

第1章 UNIX基础知识

1.1 引言

所有操作系统都向它们运行的程序提供服务。典型的服务有执行新程序、打开文件、读文件、分配存储区、获得当前时间等等，本书集中阐述了UNIX操作系统各种版本所提供的服务。

以严格的步进方式、不超前引用尚未说明过的术语的方式来说明 UNIX几乎是不可能的(可能也会是令人厌烦的)。本章从程序设计人员的角度快速浏览 UNIX，并对书中引用的一些术语和概念进行简要的说明并给出实例。在以后各章中，将对这些概念作更详细的说明。本章也对不熟悉UNIX的程序设计人员简要介绍了UNIX提供的各种服务。

1.2 登录

1.2.1 登录名

登录 UNIX系统时，先键入登录名，然后键入口令。系统在其口令文件，通常是 `/etc/passwd` 文件中查看登录名。口令文件中的登录项由 7个以冒号分隔的字段组成：登录名，加密口令，数字用户 ID(224)，数字组 ID(20)，注释字段，起始目录 (`/home/stevens`)，以及 shell 程序 (`/bin/ksh`)。

很多比较新的系统已将加密口令移到另一个文件中。第 6章将说明这种文件以及存取它们的函数。

1.2.2 shell

登录后，系统先显示一些典型的系统信息，然后就可以向 shell程序键入命令。shell是一个命令行解释器，它读取用户输入，然后执行命令，用户通常用终端，有时则通过文件（称为 shell脚本）向shell进行输入。常用的shell有：

- Bourne shell, `/bin/sh`
- C shell, `/bin/csh`
- KornShell, `/bin/ksh`

系统从口令文件中登录项的最后一个字段中了解到应该执行哪一个 shell。

自V7以来，Bourne shell得到了广泛应用，几乎每一个现有的UNIX系统都提供 Bourne shell。C shell是在伯克利开发的，所有 BSD版本都提供这种shell。另外，AT&T的系统V/386 R3.2和 SVR4也提供C shell（下一章将对这些不同的UNIX版本作更多说明）。KornShell是Bourne shell的后继者，它由SVR4提供。KornShell在大多数UNIX系统上运行，但在SVR4之前，通常它需要另行购买，所以没有其他两种shell流行。

本书将使用很多shell实例，以执行已开发的程序，其中将应用 Bourne shell和KornShell都具有的功能。

Bourne shell是Steve Bourne在贝尔实验室中开发的，其控制流结构使人想起Algol 68。C shell是在伯克利由Bill Joy完成的，其基础是第6版shell（不是Bourne shell）。其控制结构很像C语言，它支持一些Bourne shell没有提供的功能，如作业控制，历史机制和命令行编辑。KornShell是David Korn在贝尔实验室中开发的，它兼容Bourne shell，并且也包含了使C shell非常流行的一些功能，如作业控制、命令行编译等。

本书将使用这种形式的注释来描述历史，并对不同的UNIX实现进行比较。当我们了解了历史缘由后，采用某种特定实现技术的原因将变得清晰起来。

1.3 文件和目录

1.3.1 文件系统

UNIX文件系统是目录和文件的一种层次安排，目录的起点称为根（root），其名字是一个字符/。

目录（directory）是一个包含目录项的文件，在逻辑上，可以认为每个目录项都包含一个文件名，同时还包含说明该文件属性的信息。文件属性是：文件类型，文件长度，文件所有者，文件的许可权（例如，其他用户能否访问该文件），文件最后的修改时间等。stat和fstat函数返回一个包含所有文件属性的信息结构。第4章将详细说明文件的各种属性。

1.3.2 文件名

目录中的各个名字称为文件名（filename）。不能出现在文件名中的字符只有两个，斜线（/）和空操作符（null）。斜线分隔构成路径名（在下面说明）的各文件名，空操作符则终止一个路径名。尽管如此，好的习惯是只使用印刷字符的一个子集作为文件名字符（只使用子集的理由是：如果在文件名中使用了某些shell特殊字符，则必须使用shell的引号机制来引用文件名）。

当创建一个新目录时，自动创建了两个文件名：.（称为点）和..（称为点-点）。点引用当前目录，点-点则引用父目录。在最高层次的根目录中，点-点与点相同。

某些UNIX文件系统限制文件名的最大长度为14个字符，BSD版本则将这种限制扩展为255个字符。

1.3.3 路径名

0个或多个以斜线分隔的文件名序列（可以任选地以斜线开头）构成路径名（pathname），以斜线开头的路径名称为绝对路径名（absolute pathname），否则称为相对路径名（relative pathname）。

实例

不难列出一个目录中所有文件的名字，程序1-1是ls(1)命令的主要实现部分

程序1-1 列出一个目录中的所有文件

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

```

#include    "ourhdr.h"

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

    if (argc != 2)
        err_quit("a single argument (the directory name) is required");

    if ( (dp = opendir(argv[1])) == NULL)
        err_sys("can't open %s", argv[1]);

    while ( (dirp = readdir(dp)) != NULL)
        printf("%s\n", dirp->d_name);

    closedir(dp);
    exit(0);
}

```

ls(1)这种表示方法是UNIX的惯用方法，用以引用UNIX手册集中的一个特定项。它引用第一部分中的ls项，各部分通常用数字1至8表示，在每个部分中的各项则按字母顺序排列。假定你有一份所使用的UNIX系统的手册。

早期的UNIX系统把8个部分都集中在一本手册中，现在的趋势是把这些部分分别安排在不同的手册中：有用户专用手册，程序员专用手册，系统管理员专用的手册等等。

某些UNIX系统把一个给定部分中的手册页又用一个大写字母进一步分成若干小部分，例如，AT&T [1990e]中的所有标准I/O函数都被指明在3S部分中，例如fopen(3S)。

某些UNIX系统，例如以Xenix为基础的系统，不是采用数字将手册分成若干部分，而是用C表示命令(第1部分)，S表示服务(通常是第2、3部分)等等。

如果你有联机手册，则可用下面的命令查看ls命令手册页：

```
man 1 ls
```

程序1-1只打印一个目录中各个文件的名称，不显示其他信息，如若该源文件名为 myls.c，则可以用下面的命令对其进行编译，编译的结果送入系统默认名为 a.out的可执行文件名：

```
cc myls.c
```

某种样本输出是：

```

$ a.out /dev
.
..
MAKEDEV
console
tty
mem
kmem
null

```

此处略去多行

```
printer
$ a.out /var/spool/mqueue
can't open /var/spool/mqueue:Permission denied
$ a.out /dev/tty
can't open /dev/tty:Not a directory
```

本书将以这种方式表示输入的命令以及其输出：输入的字符以粗体表示，程序输出则以另一种字体表示。如果欲对输出添加注释，则以中文宋体表示，输入之前的美元符号 (\$) 是 shell 打印的提示符，本书将 shell 提示符显示为 \$。

注意，列出的目录项不是以字母顺序排列的，ls 命令则一般按字母顺序列出目录项。

在这 20 行的程序中，有很多细节需要考虑：

- 首先，其中包含了一个头文件 ourhdr.h。本书中几乎每一个程序都包含此头文件。它包含了某些标准系统头文件，定义了许多常数及函数原型，这些都将用于本书的各个实例中，附录 B 列出了常用头文件。

- main 函数的说明使用了 ANSI C 标准所支持的新风格（下一章将对 ANSI C 作更多说明）。

- 取命令行的第 1 个参数 argv [1] 作为列出的目录名。第 7 章将说明 main 函数如何被调用，程序如何存取命令行参数和环境变量。

- 因为各种不同 UNIX 系统的目录项的实际格式是不一样的，所以使用函数 opendir, readdir 和 closedir 处理目录。

- opendir 函数返回指向 DIR 结构的指针，并将该指针传向 readdir 函数。我们并不关心 DIR 结构中包含了什么。然后，在循环中调用 readdir 来读每个目录项。它返回一个指向 dirent 结构的指针，而当目录中已无目录项可读时则返回 null 指针。在 dirent 结构中取出的只是每个目录项的名字 (d_name)。使用该名字，此后就可调用 stat 函数 (见 4.2 节) 以决定该文件的所有属性。

- 调用了两个自编的函数来对错误进行处理：err_sys 和 err_quit。从上面的输出中可以看到，err_sys 函数打印一条消息 (“ Permission denied (许可权拒绝)” 或 “ Not a directory (不是一个目录) ”)，说明遇到了什么类型的错误。这两个出错处理函数在附录 B 中说明，1.7 节将更多地叙述出错处理。这两个出错处理函数在附录 B 中说明 1.7 节将更详细地叙述出错处理。

- 当程序将结束时，它以参数 0 调用函数 exit。函数 exit 终止程序。按惯例，参数 0 的意思是正常结束，参数值 1 ~ 255 则表示出错。8.5 节将说明一个程序 (例如 shell 或我们所编写的程序) 如何获得它所执行的另一个程序的 exit 状态。

1.3.4 工作目录

每个进程都有个工作目录 (working directory，有时称为当前工作目录 (current working directory))。所有相对路径名都从工作目录开始解释。进程可以用 chdir 函数更改其工作目录。

例如，相对路径名 doc/memo/joe 指的是文件 joe，它在目录 memo 中，而 memo 又在目录 doc 中，doc 则应是工作目录中的一个目录项。从该路径名可以看出，doc 和 memo 都应当是目录，但是却不清楚 joe 是文件还是目录。路径名 /usr/lib/lint 是一个绝对路径名，它指的是文件 (或目录) lint，而 lint 在目录 lib 中，lib 则在目录 usr 中，usr 则在根目录中。

1.3.5 起始目录

登录时，工作目录设置为起始目录 (home directory)，该起始目录从口令文件 (见 1.2 节) 中

的登录项中取得。

1.4 输入和输出

1.4.1 文件描述符

文字描述符是一个小的非负整数，内核用以标识一个特定进程正在存访的文件。当内核打开一个现存文件或创建一个新文件时，它就返回一个文件描述符。当读、写文件时，就可使用它。

1.4.2 标准输入、标准输出和标准出错

按惯例，每当运行一个新程序时，所有的 shell 都为其打开三个文件描述符：标准输入、标准输出以及标准出错。如果像简单命令 `ls` 那样没有做什么特殊处理，则这三个描述符都连向终端。大多数 shell 都提供一种方法，使任何一个或所有这三个描述符都能重新定向到某一个文件，例如：

```
ls > file.list
```

执行 `ls` 命令，其标准输出重新定向到名为 `file.list` 的文件上。

1.4.3 不用缓存的 I/O

函数 `open`、`read`、`write`、`lseek` 以及 `close` 提供了不用缓存的 I/O。这些函数都用文件描述符进行工作。

实例

如果愿意从标准输入读，并写向标准输出，则程序 1-2 可用于复制任一 UNIX 文件。

程序 1-2 将标准输入复制到标准输出

```
#include "ourhdr.h"

#define BUFSIZE 8192

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

    while ( (n = read(STDIN_FILENO, buf, BUFSIZE)) > 0)
        if (write(STDOUT_FILENO, buf, n) != n)
            err_sys("write error");

    if (n < 0)
        err_sys("read error");

    exit(0);
}
```

头文件 `<unistd.h>` (`ourhdr.h` 中包含了此头文件) 及两个常数 `STDIN_FILENO` 和 `STDOUT_FILENO` 是 POSIX 标准的一部分 (下一章将对此作更多的说明)。很多 UNIX 系统服务的函数原型，例如我们调用的 `read` 和 `write` 都在此头文件中。函数原型也是 ANSI C 标准的一部分，本章的

稍后部分将对此作更多说明。

两个常数STDIN_FILENO和STDOUT_FILENO定义在<unistd.h>头文件中，它们指定了标准输入和标准输出的文件描述符。它们的典型值是0和1，但是为了可移植性，我们将使用这些新名字。

3.9节将详细讨论BUFSIZE常数，说明各种不同的值将如何影响程序的效率。但是不管该常数的值如何，此程序总能复制任一UNIX文件。

read函数返回读得的字节数，此值用作要写的字节数。当到达文件的尾端时，read返回0，程序停止执行。如果发生了一个读错误，read返回-1。出错时大多数系统函数返回-1。

如果编译该程序，其结果送入标准的a.out文件，并以下列方式执行它：

```
a.out > data
```

那么，标准输入是终端，标准输出则重新定向至文件data，标准出错也是终端。如果此输出文件并不存在，则shell创建它。

第3章将更详细地说明不用缓存的I/O函数。

1.4.4 标准I/O

标准I/O函数提供一种对不用缓存的I/O函数的带缓存的界面。使用标准I/O可无需担心如何选取最佳的缓存长度，例如程序1-2中的BUFSIZE常数。另一个使用标准I/O函数的优点与处理输入行有关(常常发生在UNIX的应用中)。例如，fgets函数读一完整的行，而另一方面，read函数读指定字节数。

我们最熟悉的标准I/O函数是printf。在调用printf的程序中，总是包括<stdio.h>(通常包括在ourhdr.h中)，因为此头文件包括了所有标准I/O函数的原型。

实例

程序1-3的功能类似于调用read和write的前一个程序1-2，5.8节将对程序1-3作更详细的说明。它将标准输入复制到标准输出，于是也就能复制任一UNIX文件。

程序1-3 用标准I/O将标准输入复制到标准输出

```
#include "ourhdr.h"

int
main(void)
{
    int    c;

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

    if (ferror(stdin))
        err_sys("input error");

    exit(0);
}
```

函数getc一次读1个字符，然后putc将此字符写到标准输出。读到输入的最后1个字节时，getc返回常数EOF。标准输入、输出常数stdin和stdout定义在头文件<stdio.h>中，它们分别表示标准输入和标准输出文件。

1.5 程序和进程

1.5.1 程序

程序 (program) 是存放在磁盘文件中的可执行文件。使用 6 个 exec 函数中的一个由内核将程序读入存储器, 并使其执行。8.9 节将说明这些 exec 函数。

1.5.2 进程和进程 ID

程序的执行实例被称为进程 (process)。本书的每一页几乎都会使用这一术语。某些操作系统用任务表示正被执行的程序。

每个 UNIX 进程都一定有一个唯一的数字标识符, 称为进程 ID (process ID)。进程 ID 总是一非负整数。

实例

程序 1-4 用于打印进程 ID。

程序 1-4 打印进程 ID

```
#include    "ourhdr.h"

int
main(void)
{
    printf("hello world from process ID %d\n", getpid());
    exit(0);
}
```

如果要编译该程序, 其结果送入 a.out 文件, 然后执行它, 则有:

```
$ a.out
hello world from process ID 851
$ a.out
hello world from precess ID 854
```

此程序运行时, 它调用函数 getpid 得到其进程 ID。

1.5.3 进程控制

有三个用于进程控制的主要函数: fork、exec 和 waitpid (exec 函数有六种变体, 但经常把它们统称为 exec 函数)。

实例

程序 1-5 从标准输入读命令并执行

```
#include    <sys/types.h>
#include    <sys/wait.h>
#include    "ourhdr.h"

int
main(void)
```

```
{
    char    buf[MAXLINE];
    pid_t   pid;
    int     status;

    printf("%% "); /* print prompt (printf requires %% to print %) */
    while (fgets(buf, MAXLINE, stdin) != NULL) {
        buf[strlen(buf) - 1] = 0; /* replace newline with null */

        if ( (pid = fork()) < 0)
            err_sys("fork error");

        else if (pid == 0) { /* child */
            execlp(buf, buf, (char *) 0);
            err_ret("couldn't execute: %s", buf);
            exit(127);
        }

        /* parent */
        if ( (pid = waitpid(pid, &status, 0)) < 0)
            err_sys("waitpid error");
        printf("%% ");
    }
    exit(0);
}
```

UNIX的进程控制功能可以用一个较简单的程序（见程序 1-5）说明，该程序从标准输入读命令，然后执行这些命令。这是一个类似于 shell程序的基本实施部分。在这个 30行的程序中，有很多功能需要思考：

- 用标准I/O函数fgets从标准输入一次读一行，当键入文件结束字符（通常是 Ctrl-D）作为行的第1个字符时，fgets返回一个null指针，于是循环终止，进程也就终止。第 11章将说明所有特殊的终端字符（文件结束、退格字符、整行擦除等等），以及如何改变它们。

- 因为fgets返回的每一行都以新行符终止，后随一个 null字节，故用标准C函数strlen计算此字符串的长度，然后用一个 null字节替换新行符。这一操作的目的是因为 execlp函数要求的是以null结束的参数，而不是以新行符结束的参数。

- 调用fork创建一个新进程。新进程是调用进程的复制品，故称调用进程为父进程，新创建的进程为子进程。fork对父进程返回新子进程的非负进程ID，对子进程则返回0。因为fork创建一新进程，所以说它被调用一次（由父进程），但返回两次（在父进程中和在子进程中）。

- 在子进程中，调用execlp以执行从标准输入读入的命令。这就用新的程序文件替换了子进程。fork和跟随其后的exec的组合是某些操作系统所称的产生一个新进程。在UNIX中，这两个部分分成两个函数。第8章将对这些函数作更多说明。

- 子进程调用execlp执行新程序文件，而父进程希望等待子进程终止，这一要求由调用waitpid实现，其参数指定要等待的进程（在这里，pid参数是子进程ID）。waitpid函数也返回子进程的终止状态(status变量)。在此简单程序中，没有使用该值。如果需要，可以用此值精确地确定子进程是如何终止的。

- 该程序的最主要限制是不能向执行的命令传递参数。例如不能指定列出的目录名，只能对工作目录执行ls命令。为了传递参数，先要分析输入行，然后用某种约定把参数分开（很可能使用空格或制表符），然后将分隔后的各个参数传递给execlp函数。尽管如此，此程序仍可用来说明UNIX的进程控制功能。

如果运行此程序，则得到下列结果。注意，该程序使用了一个不同的提示符(%)。


```

$ a.out
% date
Fri Jun 7 15:50:36 MST 1991
% who
stevens console Jun 5 06:01
stevens tty0 Jun 5 06:02
% pwd
/home/stevens/doc/apue/proc
% ls
Makefile
a.out
shell1.c
% ^D          键入文件结束符
$           输出常规的shell提示符

```

1.6 ANSI C

本书中的所有实例都用ANSI C编写。

1.6.1 函数原型

头文件<unistd.h>包含了许多UNIX系统服务的函数原型，例如已调用过的 read，write和 getpid函数。函数原型是ANSI C标准的组成部分。这些函数原型如下列形式：

```

ssize_t read(int, void *, size_t);
ssize_t write(int, const void *, size_t);
pid_t getpid(void);

```

最后一个是说：getpid没有参数(void)，返回值的数据类型为pid_t。提供了这些函数原型后，编译程序在编译时就可以检查在调用函数时是否使用了正确的参数。在程序 1-4中，如果调用带参数的getpid(如getpid(1))，则ANSI C编辑程序将给出下列形式的出错信息：

```
line 8: too many arguments to function "getpid"
```

另外，因为编译程序知道参数的数据类型，所以如果可能，它就会将参数强制转换成所需的数据类型。

1.6.2 类属指针

从上面所示的函数原型中可以注意到另一个区别：read和write的第二个参数现在是void *类型。所有早期的UNIX系统都使用char *这种指针类型。作这种更改的原因是：ANSI C使用void *作为类属指针来代替char *。

函数原型和类属指针的组合消去了很多非ANSI C编辑程序需要的显式类型强制转换。

例如，给出了write原型后，可以写成：

```
float data[100];
write (fd, data, sizeof(data))
```

若使用非ANSI编译程序，或没有给出函数原型，则需写成：

```
write(fd, (void *)data, sizeof(data));
```

也可将void *指针特征用于malloc函数(见7.8节)。malloc的原型为：

```
void * malloc(size_t);
```

这使得可以有如下程序段：

```
int * ptr;
ptr = malloc (1000 * sizeof(int));
```

它无需将返回的指针强制转换成 `int *` 类型。

1.6.3 原始系统数据类型

前面所示的 `getpid` 函数的原型定义了其返回值为 `pid_t` 类型，这也是 POSIX 中的新规定。UNIX 的早期版本规定此函数返回一整型。与此类似，`read` 和 `write` 返回类型为 `ssize_t` 的值，并要求第三个参数的类型是 `size_t`。

以 `_t` 结尾的这些数据类型被称为原始系统数据类型。它们通常在头文件 `<sys/types.h>` 中定义(头文件 `<unistd.h>` 应已包括该头文件)。它们通常以 C typedef 说明加以定义。typedef 说明在 C 语言中已超过 15 年了(所以这并不要求 ANSI C)，它们的目的是阻止程序使用专门的数据类型(例如 `int`, `short` 或 `long`) 来允许对于一种特定系统的每个实现选择所要求的数据类型。在需要存储进程 ID 的地方，分配类型为 `pid_t` 的一个变量(注意，程序 1-5 已对名为 `pid` 的变量这样做了)。在各种不同的实现中，这种数据类型的定义可能是不同的，但是这种差别现在只出现在一个头文件中。我们只需在另一个系统上重新编辑应用程序。

1.7 出错处理

当 UNIX 函数出错时，往常返回一个负值，而且整型变量 `errno` 通常设置为具有特定信息的一个值。例如，`open` 函数如成功执行则返回一个非负文件描述符，如出错则返回 `-1`。在 `open` 出错时，有大约 15 种不同的 `errno` 值(文件不存在，许可权问题等)。某些函数并不返回负值而是使用另一种约定。例如，返回一个指向对象的指针的大多数函数，在出错时，将返回一个 `null` 指针。

文件 `<errno.h>` 中定义了变量 `errno` 以及可以赋与它的各种常数。这些常数都以 `E` 开头，另外，UNIX 手册第 2 部分的第 1 页，`intro(2)` 列出了所有这些出错常数。例如，若 `errno` 等于常数 `EACCES`，这表示产生了权限问题(例如，没有打开所要求文件的权限)。POSIX 定义 `errno` 为：

```
extern int errno;
```

POSIX.1 中 `errno` 的定义较 C 标准中的定义更为苛刻。C 标准允许 `errno` 是一个宏，它扩充成可修改的整型左值 (lvalue) (例如返回一个指向出错数的指针的函数)。

对于 `errno` 应当知道两条规则。第一条规则是：如果没有出错，则其值不会被一个例程清除。因此，仅当函数的返回值指明出错时，才检验其值。第二条是：任一函数都不会将 `errno` 值设置为 0，在 `<errno.h>` 中定义的所有常数都不为 0。

C 标准定义了两个函数，它们帮助打印出错信息。

```
#include <string.h>
char *strerror(int rnum);
```

返回：指向消息字符串的指针

此函数将 `errnum` (它通常就是 `errno` 值) 映射为一个出错信息字符串，并且返回此字符串的指针。

`perror`函数在标准出错上产生一条出错消息(基于`errno`的当前值), 然后返回。

```
#include <stdio.h>

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

它首先输出由`msg`指向的字符串, 然后是一个冒号, 一个空格, 然后是对应于`errno`值的出错信息, 然后是一个新行符。

实例

程序1-6显示了这两个出错函数的使用方法。

程序1-6 例示`strerror`和`perror`

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

int
main(int argc, char *argv[])
{
    fprintf(stderr, "EACCES: %s\n", strerror(EACCES));

    errno = ENOENT;
    perror(argv[0]);

    exit(0);
}
```

如果此程序经编译, 结果送入文件`a.out`, 则有:

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

注意, 我们将程序名(`argv[0]`, 其值是`a.out`) 作为参数传递给`perror`。这是一个标准的UNIX惯例。使用这种方法, 如程序作为管道线的一部分执行, 如:

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

则我们就能分清三个程序中的哪一个产生了一条特定的出错消息。

本书中的所有实例基本上都不直接调用`strerror`或`perror`, 而是使用附录B中的出错函数。该附录中的出错函数使用了ANSI C的可变参数表设施, 用一条C语句就可处理出错条件。

1.8 用户标识

1.8.1 用户ID

口令文件登录项中的用户ID (user ID) 是个数值, 它向系统标识各个不同的用户。系统管理员在确定一个用户的登录名的同时, 确定其用户ID。用户不能更改其用户ID。通常每个用户有一个唯一的用户ID。下面将介绍内核如何使用用户ID以检验该用户是否有执行某些操作的适当许可权。

用户ID为0的用户为根(`root`)或超级用户(`superuser`)。在口令文件中, 通常有一个登录项, 其登录名为`root`, 我们称这种用户的特权为超级用户特权。我们将在第4章中看到, 如果一个进程具有超级用户特权, 则大多数文件许可权检查都不再进行。某些操作系统功能只限于向超

级用户提供，超级用户对系统有自由的支配权。

实例

程序1-7用于打印用户ID和组ID（在下面说明）。

程序1-7 打印用户ID和组ID

```
#include    "ourhdr.h"

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

调用getuid和getgid以返回用户ID和组ID。运行该程序，产生：

```
$ a.out
uid = 224, gid = 20
```

1.8.2 组ID

口令文件登录项也包括用户的组ID（group ID），它也是一个数值。组ID也是由系统管理员在确定用户登录名时分配的。一般来说，在口令文件中有多个记录项具有相同的组ID。在UNIX下，组被用于将若干用户集合到课题或部门中去。这种机制允许同组的各个成员之间共享资源（例如文件）。4.5节将说明可以设置文件的许可权使组内所有成员都能存取该文件，而组外用户则不能。

组文件将组名映射为数字组ID，它通常是/etc/group。

对于许可权使用数值用户ID和数值组ID是历史上形成的。系统中每个文件的目录项包含该文件所有者的用户ID和组ID。在目录项中存放这两个值只需4个字节（假定每个都以双字节的整型值存放）。如果使用8字节的登录名和8字节的组名，则需较多的磁盘空间。但是对于用户而言，使用名字比使用数值方便，所以口令文件包含了登录名和用户ID之间的映射关系，而组文件则包含了组名和组ID之间的映射关系。例如UNIX ls-l命令使用口令文件将数值用户ID映射为登录名，从而打印文件所有者的登录名。

1.8.3 添加组ID

除了在口令文件中对一个登录名指定一个组ID外，某些UNIX版本还允许一个用户属于另外一些组。这是从4.2 BSD开始的，它允许一个用户属于多至16个另外的组。登录时，读文件/etc/group，寻找列有该用户作为其成员的前16个登记项就可得到该用户的添加组ID（supplementary group ID）。

1.9 信号

信息是通知进程已发生某种条件的一种技术。例如，若某一进程执行除法操作，其除数为0，则将名为SIGFPE的信号发送给该进程。进程如何处理信号有三种选择：

(1) 忽略该信号。有些信号表示硬件异常，例如，除以0或访问进程地址空间以外的单元等，

因为这些异常产生的后果不确定，所以不推荐使用这种处理方式。

(2) 按系统默认方式处理。对于0除，系统默认方式是终止该进程。

(3) 提供一个函数，信号发生时则调用该函数。使用这种方式，我们将能知道什么时候产生了信号，并按所希望的方式处理它。

很多条件会产生信号。有两种键盘方式，分别称为中断键 (interrupt key，通常是Delete键或Ctrl-C)和退出键(quit key，通常是Ctrl-\)，它们被用于中断当前运行进程。另一种产生信号的方法是调用名为kill的函数。在一个进程中调用此函数就可向另一个进程发送一个信号。当然这样做也有些限制：当向一个进程发送信号时，我们必需是该进程的所有者。

实例

回忆一下基本shell程序(见程序1-5)。如果调用此程序，然后键入中断键，则执行此程序的进程终止。产生这种后果的原因是：对于此信号(SIGINT)的系统默认动作是终止此进程。该进程没有告诉系统核对此信号作何处理，所以系统按默认方式终止该进程。

为了更改此程序使其能捕捉到该信号，它需要调用signal函数，指定当产生SIGINT信号时要调用的函数名。因此编写了名为sig_int的函数，当其被调用时，它只是打印一条消息，然后打印一个新提示符。在程序1-5中加了12行构成了程序1-8(添加的12行以行首的+号指示)。

程序1-8 从标准输入读命令并执行

```

#include    <sys/types.h>
#include    <sys/wait.h>
+ #include  <signal.h>
#include    "ourhdr.h"

+ static void sig_int(int);      /* our signal-catching function */
+
int
main(void)
{
    char    buf[MAXLINE];
    pid_t   pid;
    int     status;

+   if (signal(SIGINT, sig_int) == SIG_ERR)
+       err_sys("signal error");
+
    printf("%s "); /* print prompt (printf requires %% to print %) */
    while (fgets(buf, MAXLINE, stdin) != NULL) {
        buf[strlen(buf) - 1] = 0; /* replace newline with null */

        if ( (pid = fork()) < 0)
            err_sys("fork error");

        else if (pid == 0) { /* child */
            execlp(buf, buf, (char *) 0);
            err_ret("couldn't execute: %s", buf);
            exit(127);
        }

        /* parent */
        if ( (pid = waitpid(pid, &status, 0)) < 0)
            err_sys("waitpid error");
        printf("%s ");
    }
    exit(0);
+ }
+

```

```
+ void
+ sig_int(int signo)
+ {
+     printf("interrupt\n%% ");
+ }
```

因为大多数重要的应用程序都将使用信号，所以第10章将详细介绍信号。

1.10 UNIX时间值

长期以来，UNIX系统一直使用两种不同的时间值：

(1) 日历时间。该值是自1970年1月1日00:00:00以来国际标准时间（UTC）所经过的秒数累计值（早期的手册称UTC为格林尼治标准时间）。这些时间值可用于记录文件最近一次的修改时间等。

(2) 进程时间。这也被称为CPU时间，用以度量进程使用的中央处理机资源。进程时间以时钟滴答计算，多年来，每秒钟取为50、60或100个滴答。系统基本数据类型`clock_t`保存这种时间值。另外，POSIX定义常数`CLK_TCK`，用其说明每秒滴答数。（常数`CLK_TCK`现在已不再使用。2.5.4节将说明如何用`sysconf`函数得到每秒时钟滴答数。）

当度量一个进程的执行时间时（见3.9节），UNIX系统使用三个进程时间值：

- 时钟时间。
- 用户CPU时间。
- 系统CPU时间。

时钟时间又称为墙上时钟时间（wall clock time）。它是进程运行的时间总量，其值与系统中同时运行的进程数有关。在我们报告时钟时间时，都是在系统中没有其他活动时进行度量的。

用户CPU时间是执行用户指令所用的时间量。系统CPU时间是为该进程执行内核所经历的时间。例如，只要一个进程执行一个系统服务，例如`read`或`write`，则在内核内执行该服务所花费的时间就计入该进程的系统CPU时间。用户CPU时间和系统CPU时间的和常被称为CPU时间。

要取得任一进程的时钟时间、用户时间和系统时间很容易——只要执行命令`time(1)`，其参数是要度量其执行时间的命令，例如：

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

real    0m19.81s
user    0m0.43s
sys     0m4.53s
```

`time`命令的输出格式与所使用的shell有关。

8.15节将说明一个运行进程如何取得这三个时间。关于时间和日期的一般说明见6.9节。

1.11 系统调用和库函数

所有的操作系统都提供多种服务的入口点，由此程序向内核请求服务。各种版本的UNIX都提供经良好定义的有限数目的入口点，经过这些入口点进入内核，这些入口点被称为系统调用（system call）。系统调用是不能更改的一种UNIX特征。UNIX第7版提供了约50个系统调用，4.3+BSD提供了约110个，而SVR4则提供了约120个。

系统调用界面总是在《UNIX程序员手册》的第2部分中说明。其定义也包括在C语言中。这与很多早期的操作系统不同，这些系统按传统方式在机器的汇编语言中定义内核入口点。

UNIX所使用的技术是为每个系统调用在标准C库中设置一个具有同样名字的函数。用户进程用标准C调用序列来调用这些函数，然后，函数又用系统所要求的技术调用相应的内核服务。例如函数可将一个或多个C参数送入通用寄存器，然后执行某个产生软中断进入内核的机器指令。从应用角度考虑，可将系统调用视作为C函数。

《UNIX程序员手册》的第3部分定义了程序员可以使用的通用函数。虽然这些函数可能会调用一个或多个内核的系统调用，但是它们并不是内核的入口点。例如，`printf`函数会调用`write`系统调用以进行输出操作，但函数`strcpy`(复制一字符串)和`atoi`(变换ASCII为整数)并不使用任何系统调用。

从执行者的角度来看，系统调用和库函数之间有重大区别，但从用户角度来看，其区别并不非常重要。在本书中系统调用和库函数都以C函数的形式出现，两者都对应用程序提供服务，但是，我们应当理解，如果希望的话，我们可以替换库函数，但是通常却不能替换系统调用。

以存储器分配函数`malloc`为例。有多种方法可以进行存储器分配及其相关的无用区收集操作(最佳适应，首次适应等)，并不存在对所有程序都最佳的一种技术。UNIX系统调用中处理存储器分配的是`sbrk(2)`，它不是一个通用的存储器管理器。它增加或减少指定字节数的进程地址空间。如何管理该地址空间却取决于进程。存储器分配函数`malloc(3)`实现一种特定类型的分配。如果我们不喜欢其操作方式，则可以定义自己的`malloc`函数，它可能将使用`sbrk`系统调用。事实上，有很多软件包，它们实现自己的存储器分配算法，但仍使用`sbrk`系统调用。图1-1显示了应用程序、`malloc`函数以及`sbrk`系统调用之间的关系。

从中可见，两者职责不同，相互分开，内核中的系统调用分配另外一块空间给进程，而库函数`malloc`则管理这一空间。

另一个可说明系统调用和库函数之间的差别的例子是，UNIX提供决定当前时间和日期的界面。某些操作系统提供一个系统调用以返回时间，而另一个则返回日期。任何特殊的处理，例如正常时制和夏时制之间的转换，由内核处理或要求人为干预。UNIX则不同，它只提供一条系统调用，该系统调用返回国际标准时间1970年1月1日零点以来所经过的秒数。对该值的任何解释，例如将其变换成人们可读的，使用本地时区的时间和日期，都留给用户进程运行。在标准C库中，提供了若干例程以处理大多数情况。这些库函数处理各种细节，例如各种夏时制算法。

应用程序可以调用系统调用或者库函数，而很多库函数则会调用系统调用。这在图1-2中显示。

系统调用和库函数之间的另一个差别是：系统调用通常提供一种最小界面，而库函数通

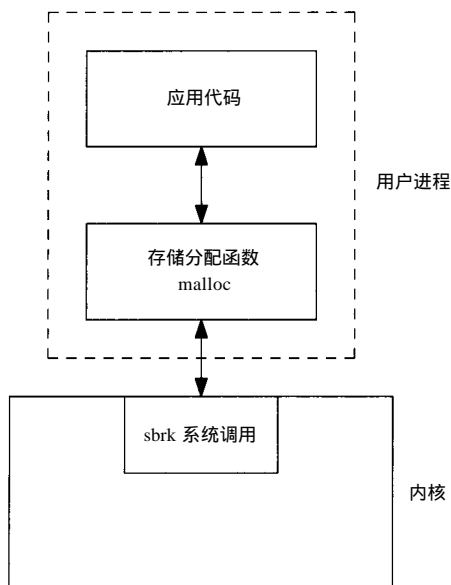


图1-1 malloc函数和sbrk系统调用

常提供比较复杂的功能。我们从 `sbrk` 系统调用和 `malloc` 库函数之间的差别中可以看到这一点，在以后当比较不带缓存的 I/O 函数（见第3章）以及标准 I/O 函数（见第5章）时，还将看到这种差别。

进程控制系统调用（`fork`、`exec` 和 `wait`）通常由用户的应用程序直接调用（请回忆程序 1-5 中的基本 shell）。但是为了简化某些常见的情况，UNIX 系统也提供了一些库函数；例如 `system` 和 `popen`。8.12 节将说明 `system` 函数的一种实现，它使用基本的进程控制系统调用。10.18 节还将强化这一实例以正确地处理信号。

为使读者了解大多数程序员应用的 UNIX 系统界面，我们不得不既说明系统调用，只介绍某些库函数。例如若只说明 `sbrk` 系统调用，那么就会忽略很多应用程序使用的 `malloc` 库函数。

本书除了必须要区分两者时，都将使用术语函数（`function`）来指代系统调用和库函数两者。

1.12 小结

本章快速浏览了 UNIX。说明了某些以后会多次用到的基本术语，介绍了一些小的 UNIX 程序的实例，从中可感知到本书的其余部分将会进一步介绍的内容。

下一章是关于 UNIX 的标准化，以及这方面的工作对当前系统的影响。标准，特别是 ANSI C 标准和 POSIX.1 标准将影响本书的余下部分。

习题

- 1.1 在系统上查证，除根目录外，目录 `.` 和 `..` 是不同的。
- 1.2 分析程序 1-4 的输出，说明进程 ID 为 852 和 853 的进程可能会发生什么情况？
- 1.3 在 1.7 节中，`perror` 的参数是用 ANSI C 的属性 `const` 定义的，而 `rerror` 的整型参数则没有用此属性定义，为什么？
- 1.4 附录 B 包含了出错处理函数 `err_sys`，当调用该函数时，保存了 `errno` 的值，为什么？
- 1.5 若日历时间存放在带符号的 32 位整型数中，那么到哪一年它将溢出？
- 1.6 若进程时间存放在带符号的 32 位整型数中，而且每秒为 100 滴答，那么经过多少天后该时间值将会溢出？

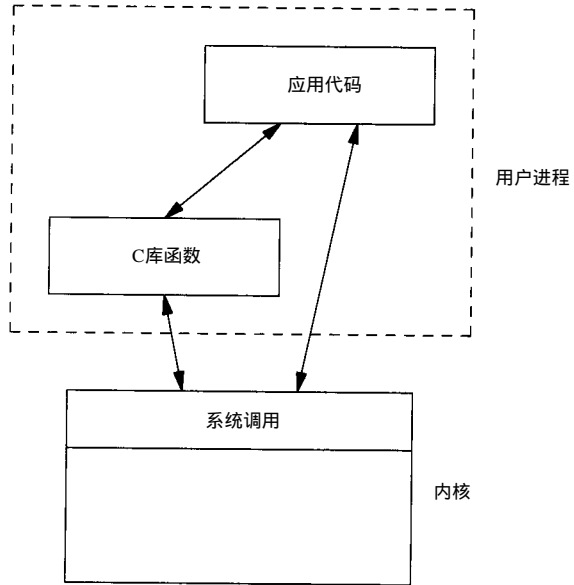


图1-2 C库函数和系统调用之间的差别

第2章 UNIX标准化及实现

2.1 引言

UNIX和C程序设计语言的标准化工作已经做了很多。虽然 UNIX应用程序在不同的 UNIX版本之间进行移植相当容易,但是 80年代UNIX版本的剧增以及它们之间差别的扩大导致很多大用户(例如美国政府)要求对其进行标准化。

本章将介绍正在进行的各种标准化工作,然后讨论这些标准对本书所列举的实际 UNIX实现的影响。所有标准化工作的一个重要部分是对每种实现必须定义的各种限制的说明,所以我们将说明这些限制以及确定它们值的多种方法。

2.2 UNIX标准化

2.2.1 ANSI C

1989年后期,C程序设计语言的ANSI标准X3.159-1989得到批准〔ANSI 1989〕。此标准已被采用为国际标准ISO/IEC 9899:1990。ANSI是美国国家标准学会,它是由制造商和用户组成的非赢利性组织。在美国,它是全国性的无偿标准交换站,在国际标准化组织(ISO)中是代表美国的成员。

ANSI C标准的意图是提供C程序的可移植性,使其能适合于大量不同的操作系统,而不只是UNIX。此标准不仅定义了C程序设计语言的语法和语义,也定义了其标准库〔ANSI 1989第4章;Plauger 1992;Kernighan及Ritchie 1988中的附录B〕。因为很多新的UNIX系统(例如本书介绍的几个UNIX系统)都提供C标准中说明的库函数,所以此库对我们来讲是很重要的。

按照该标准定义的头文件,可将该库分成 15区。表2-1中列出了C标准定义的头文件,以及下面几节中说明的另外两个标准(POSIX.1和XPG3)定义的头文件。在其中也列举了SVR4和4.3+BSD所支持的头文件。本章也将对这两种UNIX实现进行说明。

表2-1 由各种标准和实现定义的头文件

头文件	标准			实现		说明
	ANSI C	POSIX.1	XPG3	SVR4	4.3+BSD	
<assert.h>	•			•	•	验证程序断言
<cpio.h>		•		•		cpio归档值
<ctype.h>	•			•	•	字符类型
<dirent.h>		•	•	•	•	目录项(4.21节)
<errno.h>	•			•	•	出错码(1.7节)
<fcntl.h>		•	•	•	•	文件控制(3.13节)
<float.h>	•			•	•	浮点常数
<ftw.h>			•	•		文件树漫游(4.21节)

(续)

头文件	标准			实现		说明
	ANSI C	POSIX.1	XPG3	SVR4	4.3+BSD	
<grp.h>		•	•	•	•	组文件(6.4节)
<langinfo.h>			•	•		语言信息常数
<limits.h>	•			•	•	实施常数(2.5节)
<locale.h>	•			•	•	本地类别
<math.h>	•			•	•	数学常数
<nl_types.h>			•	•		消息类别
<pwd.h>		•	•	•	•	口令文件(6.2节)
<regex.h>			•	•	•	正则表达式
<search.h>			•	•		搜索表
<setjmp.h>	•			•	•	非局部goto(7.10节)
<signal.h>	•			•	•	信号(第10章)
<stdarg.h>	•			•	•	可变参数表
<stddef.h>	•			•	•	标准定义
<stdio.h>	•			•	•	标准I/O库(第5章)
<stdlib.h>	•			•	•	公用函数
<string.h>	•			•	•	字符串操作
<tar.h>		•		•		tar归档值
<termios.h>		•	•	•	•	终端I/O(第11章)
<time.h>	•			•	•	时间和日期(6.9节)
<ulimit.h>			•	•		用户限制
<unistd.h>		•	•	•	•	符号常数
<utime.h>		•	•	•	•	文件时间(4.19节)
<sys/ipc.h>			•	•	•	IPC(14.6节)
<sys/msg.h>			•	•		消息队列(14.7节)
<sys/sem.h>			•	•		信号量(14.8节)
<sys/shm.h>			•	•	•	共享存储(14.9节)
<sys/stat.h>		•	•	•	•	文件状态(第4章)
<sys/times.h>		•	•	•	•	进程时间(8.15节)
<sys/types.h>		•	•	•	•	原系统数据类型(2.7节)
<sys/utsname.h>		•	•	•		系统名(6.8节)
<sys/wait.h>		•	•	•	•	进程控制(8.6节)

2.2.2 IEEE POSIX

POSIX是一个由IEEE(电气和电子工程师学会)制订的标准族。POSIX的意思是计算机环境的可移植操作系统界面(Portable Operating System Interface for Computer Environment)。它原来指的只是IEEE标准1003.1-1988(操作系统界面)，但是，IEEE目前正在制订POSIX族中的其他有关标准。例如，1003.2将是针对shell和公用程序的标准，1003.7将是系统管理方面的标准。在1003工作组中至少有15个子委员会。

与本书相关的是1003.1操作系统界面标准，该标准定义了“POSIX依从的”操作系统必须

提供的服务。虽然1003.1标准是以UNIX操作系统为基础的，但是它又不仅仅限于UNIX和类似于UNIX的系统。确实，有些供应专有操作系统的制造商也声称这些系统将依从POSIX(同时还保有它们的所有专有功能)。

由于1003.1标准说明了一个界面(interface)而不是一种实现(implementation)，所以并不区分系统调用和库函数。所有在标准中的例程都被称为函数。

标准是不断演变的，1003.1标准也不例外。该标准的1988版，IEEE 1003.1-1988经修改后递交给ISO，没有增加新的界面或功能，但修改了文本。最终的文档作为IEEE Std.1003.1-1990正式出版[IEEE 1990]，这也就是国际标准ISO/IEC 9945-1:1990。该标准通常被称之为POSIX.1，本书将使用此标准。

IEEE 1003.1工作组此后对其又作了更多修改，它们在1993年被批准。这些修改(现在称之为1003.1a)应由IEEE作为IEEE标准1003.1-1990的附件出版，这些修改也对本书有所影响，主要是因为伯克利风格的符号链接很可能将被加到标准中作为一种要求的功能。这些修改也很可能成为ISO/IEC 9945-1:1990的一个附录。本书将用注释的方法来说明POSIX.1的1003.1a版本，指出哪些功能很可能会加到1003.1a中。

POSIX.1没有包括超级用户这样的概念。代之以规定某些操作要求“适当的优先权”，POSIX.1将此术语的定义留由具体实现进行解释。某些符合国防部安全性指导原则要求的Unix系统具有很多不同的安全级。本书仍使用传统的UNIX术语，并指明要求超级用户特权的操作。

2.2.3 X/Open XPG3

X/Open是一个国际计算机制造商组织。它提出了一个7卷本可移植性指南X/Open Portability Guide (X/Open可移植性指南)第3版[X/Open 1989]，我们将其称之为XPG3。XPG3的第2卷XSI System Interface and Headers (XSI系统界面和头文件)对类似UNIX的系统定义了一个界面，该界面定义是在IEEE Std.1003.1-1988界面的基础上制订的。XPG3包含了一些POSIX.1没有的功能。

例如，POSIX.1没有但XPG3却有的一个功能是X/Open的消息设施，该设施可由应用程序使用以在不同的语言中显示文本消息。

XPG3界面使用了ANSI C草案而不是最后的正式标准，所以在XPG3界面规格说明中包含的某些功能不再使用。这些问题很可能在将来的XPG规格说明的新版本中解决。(有关XPG4的工作正在进行，可能在1993年完成。)

2.2.4 FIPS

FIPS的含义是联邦信息处理标准(Federal Information Processing Standard)，这些标准是由美国政府出版的，并由美国政府用于计算机系统的采购。FIPS 151-1(1989年4月)基于IEEE Std.1003.1-1988及ANSI C标准草案。FIPS 151-1要求某些在POSIX.1中可选的功能。这种FIPS有时称为POSIX.1 FIPS。2.5.5节列出了FIPS所要求的POSIX.1的选择项。

POSIX.1 FIPS的影响是：它要求任一希望向美国政府销售POSIX.1依从的计算机系统的厂商应支持POSIX.1的某些可选功能。我们将不把POSIX.1 FIPS视作为另一个标准，因为实际上它只是一个更加严格的POSIX.1标准。

2.3 UNIX实现

上面一节说明了三个由各自独立的组织所制定的标准：ANSI C、IEEE POSIX以及X/Open

XPG3。但是，标准只是界面的规格说明。这些标准是如何与现实世界相关连的呢？这些标准由制造商采用，然后转变成具体实施。本书中我们感兴趣的是这些标准和它们的具体实施。

在Leffler 等著作〔1989〕的1.1节中给出了UNIX族树的详细历史和关系图。UNIX的各种版本和变体都起源于在PDP-11系统上运行的UNIX分时系统第6版（1976年）和第7版（1979年）（通常称为V6和V7）。这两个版本是在贝尔实验室以外首先得到广泛应用的UNIX系统。从这棵树上发展出三个分支：(a) AT&T分支，从此导出了系统 和系统V（被称之为UNIX的商用版本），(b) 加州大学伯克利分校分支，从此导出 4.xBSD实现，(c) 由AT&T贝尔实验室的计算科学研究中心不断开发的UNIX研究版本，从此导出第8、第9和第10版。

2.3.1 SVR4

SVR4是AT&T UNIX系统实验室的产品，它汇集了下列系统的功能：AT&T UNIX系统V第3.2版(SVR3.2)，Sun 公司的SunOS系统，加州大学伯克利分校的4.3BSD以及微软的Xenix系统（Xenix是在V7的基础上开发的，后来又采用了很多系统V的功能）。其源代码于1989年后期分发，在1990年则开始向最终用户提供。SVR4符合POSIX 1003.1标准和X/Open XPG3标准。

AT&T也出版了系统V界面定义(SVID)〔AT&T 1989〕。SVID第3版说明了UNIX系统要达到SVR4质量要求所应提供的功能。如同POSIX.1一样，SVID说明了一个界面，而不是一种实现。对于一个具体实现的SVR4应查看其参考手册，以了解其不同之处〔AT&T 1990e〕。

SVR4包含了BSD的兼容库〔AT&T 1990c〕，它提供了功能与4.3BSD对应的函数和命令。但是其中某些函数与POSIX的对应部分有所不同，本书中所有的SVR4实例都没有使用此兼容库。只有你有一些早期的应用程序，又不想改变它们时，才建议使用此兼容库，新的应用程序不应使用它。

2.3.2 4.3+BSD

BSD是由加州大学伯克利分校的计算机系统研究组研究开发和分发的。4.2BSD于1983年问世，4.3BSD则在1986年。这两个版本都在VAX小型机上运行。它们的下一个版本4.3BSD Tahoe于1988年发布，在一台称为Tahoe的小型机上运行（Leffler等的著作〔1989〕说明了4.3BSD Tahoe版）。其后又有1990年的4.3BSD Reno版，它支持很多POSIX.1的功能。下一个主要版本4.4BSD应在1992年发布。

早期的BSD系统包含了AT&T专有的源代码，它们需要AT&T许可证。为了获得BSD系统的源代码，首先需要持有AT&T的UNIX许可证。这种情况正在得到改变，在近几年来愈来愈多的AT&T源代码正被代换成非AT&T源代码，很多加到BSD系统上的新功能也来自于非AT&T方面。

1989年，伯克利将4.3BSD Tahoe中很多非AT&T源代码包装成BSD网络软件，1.0版，并使其成为公众可用的软件。其后则有BSD网络软件的2.0版，它是从4.3BSD Reno版导出的，其目的是使大部分（如果不是全部的话）4.4BSD系统不再受AT&T许可证的限制，于是其全部源代码都可为公众使用。

正如前言中所说明的，本书使用术语 4.3+BSD表示BSD系统，该系统位于BSD网络软件2.0版和即将发布的4.4BSD之间。

在伯克利所进行的UNIX开发工作是从PDP-11开始的，然后转移到VAX小型机上，接着又转移到工作站上。90年代早期，伯克利得到支持在广泛应用的80386个人计算机上开发BSD版

本，结果产生了 386BSD。这一工作是由 Bill Jolitz 完成的。其相关文档有发表在 1991 年 Dr.Dobb's Journal 上的系列文章(每月一篇)。其中很多代码出现在 BSD 网络软件 2.0 版中。

2.4 标准和实现的关系

我们已提及的标准定义了任一实际系统的子集。虽然 IEEE POSIX 正致力于在其他所需方面(例如，网络界面，进程间的通信，系统管理)制订出标准，但在编著本书写作时，这些标准还并不存在。

本书集中阐述了两个实际的 UNIX 系统：SVR4 和 4.3+BSD。因为这两个系统都宣称是依从 POSIX 的，所以我们一方面集中阐述了 POSIX.1 标准所要求的功能，同时又指出 POSIX 和这两个系统具体实现之间的差别。故 SVR4 或 4.3+BSD 特有的功能和例程都被清楚地标记出来。因为 XPG3 是 POSIX.1 的超集，所以我们还叙述了属于 XPG3，但不属于 POSIX.1 的功能。

应当了解，SVR4 和 4.3+BSD 都提供了对它们早期版本功能的兼容性（例如 SVR3.2 对 4.3BSD）。例如，SVR4 对 POSIX 规格说明中的非阻塞 I/O(O_NONBLOCK)以及传统的系统 V 方法(O_NDELAY)都提供了支持。本书将只使用 POSIX.1 的功能，但是也会提及它所替换的是哪一种非标准功能。与此相类似，SVR3.2 和 4.3BSD 以某种方法提供了可靠信号机制，这种方法也有别于 POSIX.1 标准。第 10 章将只说明 POSIX.1 的信号机制。

2.5 限制

有很多由实现定义的幻数和常数，其中有很多已被编写到程序中，或由特定的技术所确定。由于大量标准化工作的努力，已有若干种可移植的方法用以确定这些幻数和实现定义的限制。这非常有助于软件的可移植性。

以下三种类型的功能是必需的：

- 编译时间选择项（该系统是否支持作业控制）。
- 编译时间限制（短整型的最大值是什么）。
- 运行时间限制（文件名的最大字符数为多少）。

前两个，编译时间选择项和限制可在头文件中定义。程序在编译时可以包含这些头文件。但是，运行时间限制则要求进程调用一个函数以获得此种限制值。

另外，某些限制在一个给定的实现中可能是固定的(因此可以静态地在一个头文件中定义)，而在另一个实现上则可能是变动的(需要有一个运行时间函数调用)。这种类型限制的一个例子是文件名的最大字符数。系统 V 由于历史原因只允许文件名有 14 个字符，而伯克利的系统则将此增加为 255。SVR4 允许我们对每一个创建的文件系统指明是系统 V 文件系统还是 BSD 文件系统，而每个系统有不同的限制。这就是运行时间限制的一个实例，文件名的最大长度依赖于文件所处的文件系统。例如，根文件系统中的文件名长度限制可能是 14 个字符，而在某个其他文件系统中文件名长度限制可能是 255 个字符。

为了解决这些问题，提供了三种限制：

- (1) 编辑时间选择项及限制（头文件）。
- (2) 不与文件或目录相关联的运行时间限制。
- (3) 与文件或目录相关联的运行时间限制。

使事情变得更加复杂的是，如果一个特定的运行时间限制在一个给定的系统上并不改变，则将其静态地定义在一个头文件中，但是，如果没有将其定义在头文件中，则应用程序就必

须调用三个conf函数中的一个（我们很快就会对它们进行说明），以确定其运行时间值。

2.5.1 ANSI C限制

所有由ANSI C定义的限制都是编译时间限制。表 2-2中列出了文件<limits.h>中定义的C标准限制。这些常数总是定义在该头文件中，而且在一个给定系统中并不会改变。第 3列列出了ANSI C标准可接受的最小值。这用于整型长度为 16位的系统，它使用 1的补码表示。第4列列出了整型长度为32位的当前系统的值，用的是 2的补码表示法。注意，对不带符号的数据类型都没有列出其最小值，它们都应 为 0。

我们将会遇到的一个区别是系统是否提供带符号（signed）或不带符号的（unsigned）的字符值。从表 2-2中的第 4列可以看出，该特定系统使用带符号字符。从表中可以看到CHAR_MIN等于SCHAR_MIN，CHAR_MAX等于SCHAR_MAX。如果系统使用不带符号字符，则CHAR_MIN等于0，CHAR_MAX等于UCHAR_MAX。

在头文件<float.h>中，对浮点数据类型也有类似的一组定义。

我们会遇到的另一个ANSI C常数是FOPEN_MAX，这是实现保证的可同时打开的标准 I/O流的最小数，该值在头文件<stdio.h>中定义，其最小值是8。POSIX.1中的STREAM_MAX（若定义的话），则应具与FOPEN_MAX相同的值。

ANSI C在<stdio.h>中也定义了常数TMP_MAX，这是由tmpnam函数产生的唯一文件名的最大数。关于此常数我们将在5.13节中进行更多说明。

表2-2 <limits.h>中的整型值大小

名 字	说 明	最小可接受值	典 型 值
CHAR_BIT	char的位	8	8
CHAR_MAX	char的最大值	(见后)	127
CHAR_MIN	char的最小值	(见后)	- 128
SCHAR_MAX	带符号char的最大值	127	127
SCHAR_MIN	带符号char的最小值	- 127	- 128
UCHAR_MAX	不带符号char的最大值	255	255
INT_MAX	int最大值	32 767	2 147 483 647
INT_MIN	int最小值	- 32 767	- 2 147 483 648
UINT_MAX	不带符号的int的最大值	65 535	4 294 967 295
SHRT_MIN	short最小值	- 32 767	- 32 768
SHRT_MAX	short最大值	32 767	32 767
USHRT_MAX	不带符号的short的最大值	65 535	65 535
LONG_MAX	long最大值	2 147 483 647	2 147 483 647
LONG_MIN	long最小值	- 2 147 483 647	- 2 147 483 648
ULONG_MAX	不带符号的long的最大值	4 294 967 295	4 294 967 295
MB_LEN_MAX	—多字节字符常数中的最大字节数	1	1

2.5.2 POSIX限制

POSIX.1定义了很多涉及操作系统实现限制的常数，不幸的是，这是 POSIX.1中最令人迷惑不解的部分之一。

它包括33个限制和常数，它们被分成下列八类：

(1) 不变的最小值（表2-3中的13个常数）。

(2) 不变值：SSIZE_MAX。

(3) 运行时间不能增加的值：NGROUPS_MAX。

(4) 运行时间不变的值(可能不确定)：ARG_MAX, CHILD_MAX, OPEN_MAX, STREAM_MAX以及TZNAME_MAX。

(5) 路径名可变值(可能不确定)：LINK_MAX, MAX_CANON, MAX_INPUT, NAME_MAX, PATH_MAX以及PIPE_BUF。

(6) 编辑时间符号常数：_POSIX_SAVED_IDS, _POSIX_VERSION以及_POSIX_JOB_CONTROL。

(7) 执行时间符号常数：_POSIX_NO_TRUNC, _POSIX_VDISABLE以及_POSIX_CHOWN_RESTRICTED。

(8) 不再使用的常数：CLK_TCK。

在这33个限制和常数中，15个是必须定义的，其余的则按具体条件可定义可不定义。在2.5.4节中，在说明sysconf, pathconf和fpathconf函数时，我们描述了可定义可不定义的限制和常数（第4~8条）。在表2-7中我们总结了所有限制和常数。13个不变最小值则示于表2-3中。

表2-3 <limits.h>中的POSIX.1不变最小值

名 字	描 述	值
_POSIX_ARG_MAX	exec函数的参数长度	4096
_POSIX_CHILD_MAX	每个实际用户ID的子进程数	6
_POSIX_LINK_MAX	至一个文件的连接数	8
_POSIX_MAX_CANON	终端规范输入队列的字节数	255
_POSIX_MAX_INPUT	终端输入队列的可用空间	255
_POSIX_NAME_MAX	文件名中的字节数	14
_POSIX_NGROUPS_MAX	每个进程同时的添加组ID数	0
_POSIX_OPEN_MAX	每个进程的打开文件数	16
_POSIX_PATH_MAX	路径名中的字节数	255
_POSIX_PIPE_BUF	能原子地写到一管道的字节数	512
_POSIX_SSIZE_MAX	能存在ssize_t对象中的值	32 767
_POSIX_STREAM_MAX	一个进程能一次打开的标准I/O流数	8
_POSIX_TZNAME_MAX	时区名字节数	3

这些值是不变的——它们并不随系统而改变。它们指定了这些特征最严格的值。一个符合POSIX.1的实现应当提供至少这样大的值。这就是为什么将它们称为最小值的原因，虽然它们的名字都包含了MAX。另外，一个可移植的应用程序不应要求更大的值。我们将在本书的适当部分说明每一个常数的含意。

不幸的是，这些不变最小值中的某一些在实际应用中太小了。例如，目前UNIX系统每个进程可同时打开的文件数远远超过16，即使是1978年的V7也提供了20个。另外，_POSIX_PATH_MAX的最小值为255也太小了，路径名可能会超过这一限制。这意味着在编辑时不能使用这两个常数_POSIX_OPEN_MAX和_POSIX_PATH_MAX作为数组长度。

表2-3中的13个不变最小值的每一个都有一个相关的实现值，其名字是将表2-3中的名字删除前缀_POSIX_后构成的。（这13个实现值是本节开始部分所列出的2~5项：不变值、运行时不

能增加的值、运行时不变的值、以及路径名可变值。)问题是所有这13个实现值并不能确保在<limit.h>头文件中定义。某个特定值可能不在此头文件中定义的理由是：对于一个给定进程的实际值可能依赖于系统的存储器总量。如果没有在头文件中定义它们，则不能在编译时使用它们作为数组边界。所以，POSIX.1决定提供三个运行时间函数以供调用：sysconf, pathconf以及fpathconf，用它们可以在运行时间得到实际的实现值。但是，还有一个问题，因为其中某些值是由POSIX.1定义为“可能不确定的”(逻辑上无限的)，这就意味着该值没有实际上限。例如，SVR4的每个进程打开文件数在假想上是无限的，所以在SVR4中OPEN_MAX被认为是不可确定的。2.5.7节还将讨论运行时间限制不可确定的问题。

2.5.3 XPG3限制

XPG3定义了七个常数，它们总是包含在<limits.h>头文件中。POSIX.1则会把它们称之为不变最小值。它们列于表2-4中。这些值的大多数都涉及消息。

表2-4 <limits.h>中的XPG3不变最小值

名字	说明	最小可接受的值	典型值
NL_ARGMAX	调用printf和scanf的最大数字值	9	9
NL_LANGMAX	LANG环境变量中的最大字节数	14	14
NL_MSGMAX	最大消息数	32,767	32,767
NL_NMAX	在N对1映射字符中的最大字节数		1
NL_SETMAX	最大集合数	255	255
NL_TEXTMAX	消息字符串中的最大字节数	255	255
NZERO	缺省进程优先权	20	20

XPG3也定义了值PASS_MAX，作为口令中的最大有效字符数(不包括终止字符null)，它可能包含在<limits.h>中。POSIX.1则把它称之为运行时间不变的值(可能不确定)，其最小可接受的值是8。PASS_MAX值也可在运行时间用sysconf函数取得，该函数将在2.5.4节中说明。

2.5.4 sysconf、pathconf 和fpathconf 函数

我们已列出了一个实现必须支持的各种最小值，但是怎样才能找到一个特定系统实际支持的限制值呢？正如前面提到的，某些限制值在编译时是可用的，而另外一些则必须在运行时确定。我们也曾提及在一个给定的系统中某些限制值是不会更改的，而其他则与文件和目录相关联。运行时间限制是由调用下面三个函数中的一个而取得的。

```
#include <unistd.h>

long sysconf(int name);

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

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

所有函数返回：若成功为相应值，若出错为-1(见后)

最后两个函数之间的差别是一个用路径名作为其参数，另一个则取文件描述符作为参数。

表2-5中列出了这三个函数所使用的name参数。以_SC_开始的常数用作为sysconf的参数，而以_PC_开始的常数则作为pathconf或fpathconf的参数。

对于pathconf 的参数`pathname`, fpathconf 的参数`filedes` 有很多限制。如果不满足其中任何一个限制, 则结果是未定义的。

(1) `_PC_MAX_CANON`, `_PC_MAX_INPUT`以及`_PC_VDISABLE`所涉及的文件必须是终端文件。

(2) `_PC_LINK_MAX`所涉及的文件可以是文件或目录。如果是目录, 则返回值用于目录本身(不用于目录内的文件名项)。

(3) `_PC_NAME_MAX`和`_PC_NO_TRUNC`所涉及的文件必须是目录, 返回值用于该目录中的文件名。

(4) `_PC_PATH_MAX`涉及的必须是目录。当所指定的目录是工作目录时, 返回值是相对路径名的最大长度。(不幸的是, 这不是我们想要知道的一个绝对路径名的实际最大长度, 我们将在2.5.7节中再回到这一问题上来。)

(5) `_PC_PIPE_BUF`所涉及的文件必须是管道, FIFO或目录。在管道或FIFO情况下, 返回值是对所涉及的管道或FIFO的限制值。对于目录, 返回值是对在该目录中创建的任一 FIFO的限制值。

(6) `_PC_CHOWN_RESTRICTED`必须是文件或目录。如果是目录, 则返回值指明此选择项是否适用于该目录中的文件。

表2-5 对sysconf、pathconf和fpathconf的限制及name参数

限制名	说明	name 参数
ARG_MAX	exec函数的参数最大长度(字节)	_SC_ARG_MAX
CHILD_MAX	每个实际用户ID的最大进程数	_SC_CHILD_MAX
clock ticks/ second	每秒时钟滴答数	_SC_CLK_TCK
NGROUPS_MAX	每个进程的最大同时添加组ID数	_SC_NGROUPS_MAX
OPEN_MAX	每个进程最大打开文件数	_SC_OPEN_MAX
PASS_MAX	口令中的最大有效字符数	_SC_PASS_MAX
STREAM_MAX	在任一时刻每个进程的最大标准I/O流数——如若定义, 则其值一定与FOPEN_MAX相同	_SC_STREAM_MAX
TZNAME_MAX	时区名中的最大字节数	_SC_TZNAME_MAX
_POSIX_JOB_CONTROL	指明实现是否支持作业控制	_SC_JOB_CONTROL
_POSIX_SAVED_IDS	指明实现是否支持保存的设置-用户-ID和保存的设置-组-ID	_SC_SAVED_IDS
_POSIX_VERSION	指明POSIX.1版本	_SC_VERSION
_XOPEN_VERSION	指明XPG版本(非POSIX.1)	_SC_XOPEN_VERSION
LINK_MAX	文件连接数的最大值	_PC_LINK_MAX
MAX_CANON	在一终端规范输入队列的最大字节数	_PC_MAX_CANON
MAX_INPUT	终端输入队列可用空间的字节数	_PC_MAX_INPUT
NAME_MAX	文件名中的最大字节数(不包括null结束符)	_PC_NAME_MAX
PATH_MAX	相对路径名中的最大字节数(不包括null结束符)	_PC_PATH_MAX
PIPE_BUF	能原子地写到一管道的最大字节数	_PC_PIPE_BUF
_POSIX_CHOWN_RESTRICTED	指明使用chown是否受到限制	_PC_CHOWN_RESTRICTED
_POSIX_NO_TRUNC	指明若路径名长于NAME_MAX是否产生一错误	_PC_NO_TRUNC
_POSIX_VDISABLE	若定义, 终端专用字符可用此值禁止	_PC_VDISABLE

我们需要更详细地说明这三个函数的不同返回值。

(1) 如果name不是表2-5第3列中的一个合适的常数, 则所有这三个函数都返回 - 1, 并将

error设置为EINVAL。

(2) 包含MAX的12个name以及name_PC_PIPE_BUF可能或者返回该变量的值(返回值 0)，或者返回 - 1，这表示该值是不确定的，此时并不更改errno的值。

(3) 对_SC_CLK_TCK的返回值是每秒的时钟滴答数，以用于 times函数的返回值(见 8.15 节)。

(4) 对_SC_VERSION的返回值以4位数和2位数分别表示此标准的年和月。这可能或者是198808L或199009L，或此标准某个以后版本的值。

(5) 对_SC_XOPEN_VERSION的返回值表示此系统所遵从的XPG版本，其当前值是3。

(6) _SC_JOB_CONTROL和_SC_SAVED_IDS是两个可选功能。若sysconf返回 - 1(没有更改errno)则不支持相应的功能。这两个功能也可在编译时从<unistd.h>头文件中决定。

(7) 对_PC_CHOWN_RESTRICTED和_PC_NO_TRUNC的返回值若为 - 1(不改变errno)，则表示对所指定的pathname或files不支持此功能。

(8) 对_PC_VDISABLE的返回值若为 - 1(不改变errno)，则表示对所指定的pathname或files不支持此功能。若支持此功能，则返回值是被用于禁止特定终端输入字符的字符值(见表11-6)。

实例

程序2-1用于打印所有这些限制，并处理一个限制未被定义的情况。

程序2-1 打印所有可能的sysconf和pathconf值

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

static void pr_sysconf(char *, int);
static void pr_pathconf(char *, char *, int);

int
main(int argc, char *argv[])
{
    if (argc != 2)
        err_quit("usage: a.out <dirname>");

    pr_sysconf("ARG_MAX", _SC_ARG_MAX);
    pr_sysconf("CHILD_MAX", _SC_CHILD_MAX);
    pr_sysconf("clock ticks/second", _SC_CLK_TCK);
    pr_sysconf("NGROUPS_MAX", _SC_NGROUPS_MAX);
    pr_sysconf("OPEN_MAX", _SC_OPEN_MAX);
#ifdef _SC_STREAM_MAX
    pr_sysconf("STREAM_MAX", _SC_STREAM_MAX);
#endif
#ifdef _SC_TZNAME_MAX
    pr_sysconf("TZNAME_MAX", _SC_TZNAME_MAX);
#endif
    pr_sysconf("_POSIX_JOB_CONTROL", _SC_JOB_CONTROL);
    pr_sysconf("_POSIX_SAVED_IDS", _SC_SAVED_IDS);
    pr_sysconf("_POSIX_VERSION", _SC_VERSION);
    pr_pathconf("MAX_CANON", "/", _dev_tty, _PC_MAX_CANON);
    pr_pathconf("MAX_INPUT", "/", _dev_tty, _PC_MAX_INPUT);
    pr_pathconf("_POSIX_VDISABLE", "/", _dev_tty, _PC_VDISABLE);
    pr_pathconf("LINK_MAX", argv[1], _PC_LINK_MAX);
    pr_pathconf("NAME_MAX", argv[1], _PC_NAME_MAX);
    pr_pathconf("PATH_MAX", argv[1], _PC_PATH_MAX);
    pr_pathconf("PIPE_BUF", argv[1], _PC_PIPE_BUF);
```

```

pr_pathconf("_POSIX_NO_TRUNC =", argv[1], _PC_NO_TRUNC);
pr_pathconf("_POSIX_CHOWN_RESTRICTED =", argv[1], _PC_CHOWN_RESTRICTED);

exit(0);
}

static void
pr_sysconf(char *mesg, int name)
{
    long    val;

    fputs(mesg, stdout);
    errno = 0;
    if ( (val = sysconf(name)) < 0) {
        if (errno != 0)
            err_sys("sysconf error");
        fputs(" (not defined)\n", stdout);
    } else
        printf(" %ld\n", val);
}

static void
pr_pathconf(char *mesg, char *path, int name)
{
    long    val;

    fputs(mesg, stdout);
    errno = 0;
    if ( (val = pathconf(path, name)) < 0) {
        if (errno != 0)
            err_sys("pathconf error, path = %s", path);
        fputs(" (no limit)\n", stdout);
    } else
        printf(" %ld\n", val);
}

```

我们有条件地包括了两个常数，它们已被加至 POSIX.1，但不是 IEEE Std.1003.1-1988 的一部分。表 2-6 显示了在几个不同的系统上，程序 2-1 的样本输出。我们在 4.14 节中可以了解到，SVR4 S5 文件系统是可以回溯到 V7 的传统的系统 V 文件系统。UFS 是伯克利快速文件系统的 SVR4 实现。

表 2-6 配置限制的实例

限制	SunOS 4.1.1	SVR4		4.3+BSD
		S5 fileysys	UFS fileysys	
ARG_MAX	1 048 576	5 120	5 120	20 480
CHILD_MAX	133	30	30	40
每秒时钟滴答	60	100	100	60
NGROUPS_MAX	16	16	16	16
OPEN_MAX	64	64	64	64
_POSIX_JOB_CONTROL	1	1	1	1
_POSIX_SAVED_IDS	1	1	1	未定义
_POSIX_VERSION	198 808	198 808	198 808	198 808
MAX_CANON	256	256	256	255
MAX_INPUT	256	512	512	255
_POSIX_VDISABLE	0	0	0	255
LINK_MAX	32 767	1 000	1 000	32 767

(续)

限制	SunOS 4.1.1	SVR4		4.3+BSD
		S5 fileysys	UFS fileysys	
NAME_MAX	255	14	255	255
PATH_MAX	1 024	1 024	1 024	1 024
PIPE_BUF	4 096	5 120	5 120	512
_POSIX_NO_TRUNC	1	未定义	1	1
_POSIX_CHOWN_RESTRICTED	1	未定义	未定义	1

2.5.5 FIPS 151-1要求

FIPS 151-1标准(我们已在2.2.4节中提及)由于要求下列功能,所以它比POSIX.1标准更严:

- 要求下列POSIX.1可选功能: `_POSIX_JOB_CONTROL`, `_POSIX_SAVED_IDS`, `_POSIX_NO_TRUNC`, `_POSIX_CHOWN_RESTRICTED`和`_POSIX_VDISABLE`。
- `NGROUPS_MAX`的最小值是8。
- 新创建的文件或目录的组ID应设置为它所在目录的组ID(4.6节将说明此功能)。
- 已传输了一些数据后,若`read`或`write`被一个捕捉到的信号所中断,则这些函数应返回已被传输的字节数(10.5节将讨论被中断的系统调用)。
- 登录shell应定义环境变量`HOME`和`LOGNAME`。

因为美国政府购买了很多计算机系统,所以大多数POSIX的制造商都将支持这些增加的FIPS要求。

2.5.6 限制总结

我们已说明了很多限制和幻常数,其中某些必须包含在头文件中,某些可选地包含在头文件中,其他则可在运行时决定。表2-7以字母顺序总结了所有这些常数以及得到它们值的各种方法。以`_SC_`开始的运行时间名字是`sysconf`函数的参数,以`_PC_`开始的名字是`pathconf`和`fpathconf`函数的参数,如果它有最小值,则也将其列于其中。

注意,表2-3中的13个POSIX.1不变最小值示于表2-7中的最右一列。

表2-7 编译时间和运行时间限制总结

常数名	编译时间		运行时间名	最小值
	头文件	要求否		
ARG_MAX	<limits.h>	可选	<code>_SC_ARG_MAX</code>	<code>_POSIX_ARG_MAX=4096</code>
CHAR_BIT	<limits.h>	要求		8
CHAR_MAX	<limits.h>	要求		127
CHAR_MIN	<limits.h>	要求		0
CHILD_MAX	<limits.h>	可选	<code>_SC_CHILD_MAX</code>	<code>_POSIX_CHILD_MAX=6</code>
每秒时钟滴答			<code>_SC_CLK_TCK</code>	
FOPEN_MAX	<stdio.h>	要求		8
INT_MAX	<limits.h>	要求		32 767
INT_MIN	<limits.h>	要求		- 32 767
LINK_MAX	<limits.h>	可选	<code>_PC_LINK_MAX</code>	<code>_POSIX_LINK_MAX=8</code>
LONG_MAX	<limits.h>	要求		2 147 483 647

(续)

常数名	编译时间		运行时间名	最小值
	头文件	要求否		
LONG_MIN	<limits.h>	要求		- 2 147 483 647
MAX_CANON	<limits.h>	可选	_PC_MAX_CANON	_POSIX_MAX_CANON=255
MAX_INPUT	<limits.h>	可选	_PC_MAX_INPUT	_POSIX_MAX_INPUT=255
MB_LEN_MAX	<limits.h>	要求		
NAME_MAX	<limits.h>	可选	_PC_NAME_MAX	_POSIX_NAME_MAX=14
NGROUPS_MAX	<limits.h>	要求	_SC_NGROUPS_MAX	_POSIX_NGROUPS_MAX=0
NL_ARGMAX	<limits.h>	要求		9
NL_LANGMAX	<limits.h>	要求		14
NL_MSGMAX	<limits.h>	要求		32 767
NL_NMAX	<limits.h>	要求		
NL_SETMAX	<limits.h>	要求		255
NL_TEXTMAX	<limits.h>	要求		255
NZERO	<limits.h>	要求		20
OPEN_MAX	<limits.h>	可选	_SC_OPEN_MAX	_POSIX_OPEN_MAX=16
PASS_MAX	<limits.h>	可选	_SC_PASS_MAX	8
PATH_MAX	<limits.h>	可选	_PC_PATH_MAX	_POSIX_PATH_MAX=255
PIPE_BUF	<limits.h>	可选	_PC_PIPE_BUF	_POSIX_PIPE_BUF=512
SCHAR_MAX	<limits.h>	要求		127
SCHAR_MIN	<limits.h>	要求		- 127
SHRT_MAX	<limits.h>	要求		32 767
SHRT_MIN	<limits.h>	要求		- 32 767
SSIZE_MAX	<limits.h>	要求		_POSIX_SSIZE_MAX=32,767
STREAM_MAX	<limits.h>	可选	_SC_STREAM_MAX	_POSIX_STREAM_MAX=8
TMP_MAX	<stdio.h>	要求		10,000
TZNAME_MAX	<limits.h>	可选	_SC_TZNAME_MAX	_POSIX_TZNAME_MAX=3
UCHAR_MAX	<limits.h>	要求		255
UINT_MAX	<limits.h>	要求		65 535
ULONG_MAX	<limits.h>	要求		4 294 967 295
USHRT_MAX	<limits.h>	要求		65 535
_POSIX_CHOWN_RESTRICTED	<unistd.h>	可选	_PC_CHOWN_RESTRICTED	
_POSIX_JOB_CONTROL	<unistd.h>	可选	_SC_JOB_CONTROL	
_POSIX_NO_TRUNC	<unistd.h>	可选	_PC_NO_TRUNC	
_POSIX_SAVED_IDS	<unistd.h>	可选	_SC_SAVED_IDS	
_POSIX_VDISABLE	<unistd.h>	可选	_PC_VDISABLE	
_POSIX_VERSION	<unistd.h>	要求	_SC_VERSION	
_XOPEN_VERSION	<unistd.h>	要求	_SC_XOPEN_VERSION	

2.5.7 未确定的运行时间限制

我们已提及表2-7中的某些值可能是未确定的，这些值在第三列中标记为可选的 (optional)，其名字中包含MAX，或PIPE_BUF。我们遇到的问题是如果这些值没有在头文件<limits.h>中定义，那么在编译时我们就不能使用它们。但是，如果它们的值是未确定的，那么在运行时它们可能也是未定义的。让我们观察两个特殊的例子——为一个路径名分配存储器，以及决定文件描述符的数目。

1. 路径名

很多程序需要为路径名分配存储器，一般来说，在编译时就为其分配了存储器，而且使用了各种幻数（其中很少是正确的）作为数组长度：256，512，1024或标准I/O常数BUFSIZ。4.3BSD头文件<sys/param.h>中的常数MAXPATHLEN是正确值，但是很多4.3BSD应用程序并未用它。

POSIX.1试图用PATH_MAX值来帮助我们，但是如果此值是不确定的，那么仍是毫无帮助的。程序2-2是一个全书都将使用的为路径名动态地分配存储器的函数。

如若在<limits.h>中定义了常数PATH_MAX,那么就没有任何问题，如果没有，则需调用pathconf。因为pathconf的返回值是把第一个参数视为基于工作目录的相对路径名。所以指定根为第一个参数，并将得到的返回值加1作为结果值。如果pathconf指明PATH_MAX是不确定的，那么我们就只得猜测某个值。调用malloc时的+1是为了在尾端加null字符（PATH_MAX没有考虑它）。

处理不确定结果情况的正确方法与如何使用分配到的存储空间有关。例如，如果我们为getcwd调用分配空间（返回当前工作目录的绝对路径名，见4.22节），而分配到的空间太小，则返回一个出错，errno设置为ERANGE。然后可调用realloc以增加分配空间（见7.8节和练习4.18）并再试。不断重复此操作，直到getcwd调用成功执行。

程序2-2 为路径名动态地分配空间

```
#include <errno.h>
#include <limits.h>
#include "ourhdr.h"

#ifdef PATH_MAX
static int pathmax = PATH_MAX;
#else
static int pathmax = 0;
#endif

#define PATH_MAX_GUESS 1024 /* if PATH_MAX is indeterminate */
                          /* we're not guaranteed this is adequate */
char *
path_alloc(int *size)
/* also return allocated size, if nonnull */
{
    char *ptr;

    if (pathmax == 0) { /* first time through */
        errno = 0;
        if ( (pathmax = pathconf("/", _PC_PATH_MAX)) < 0) {
            if (errno == 0)
                pathmax = PATH_MAX_GUESS; /* it's indeterminate */
            else
                err_sys("pathconf error for _PC_PATH_MAX");
        } else
            pathmax++; /* add one since it's relative to root */
    }

    if ( (ptr = malloc(pathmax + 1)) == NULL)
        err_sys("malloc error for pathname");

    if (size != NULL)
        *size = pathmax + 1;
    return(ptr);
}
```

2. 最大打开文件数

在精灵进程(是在后台运行,不与终端相连接的一种进程)中一个常见的代码序列是关闭所有打开文件。某些程序中有下列形式的代码序列:

```
#include <sys/param.h>
for (i = 0; i < NOFILE; i++)
    close(i);
```

程序假定在<sys/param.h>头文件中定义了常数NOFILE。另外一些程序则使用某些<stdio.h>版本提供作为上限的常数_NFILE。某些程序则直接将其上限值定为20。

我们希望用POSIX.1的OPEN_MAX确定此值以提高可移植性,但是如果此值是不确定的,则仍然有问题,如果我们使用下列代码

```
#include <unistd.h>

for (i = 0; i < sysconf(_SC_OPEN_MAX); i++)
    close(i);
```

而且如果OPEN_MAX是不确定的,那么sysconf将返回-1,于是,for循环根本不会执行。在这种情况下,最好的选择就是关闭所有描述符直至某个任意的限制值(例如256)。如同上面的路径名一样,这并不能保证在所有情况下都能正确工作,但这却是我们所能选择的最好方法。程序2-3中使用了这种技术。

程序2-3 确定文件描述符数

```
#include <errno.h>
#include <limits.h>
#include "ourhdr.h"

#ifdef OPEN_MAX
static int openmax = OPEN_MAX;
#else
static int openmax = 0;
#endif

#define OPEN_MAX_GUESS 256 /* if OPEN_MAX is indeterminate */
/* we're not guaranteed this is adequate */

int
open_max(void)
{
    if (openmax == 0) { /* first time through */
        errno = 0;
        if ( (openmax = sysconf(_SC_OPEN_MAX)) < 0) {
            if (errno == 0)
                openmax = OPEN_MAX_GUESS; /* it's indeterminate */
            else
                err_sys("sysconf error for _SC_OPEN_MAX");
        }
    }
    return(openmax);
}
```

我们可以耐心地调用close,直至得到一个出错返回,但是从close(EBADF)出错返回并不区分无效描述符和并未打开的描述符。如果试用此技术,而且描述符9未打开,描述符10打开了,那么将停止在9上,而不会关闭10。dup函数(见3.12节)在超过了OPEN_MAX时会返回一个特定的出错值,但是用复制一个描述符一、二百次的方法来确定此值是一种极

端的方法。

SVR4和4.3+BSD的`getrlimit(2)`函数（见7.11节）以及4.3+BSD的`getdtablesize(2)`函数返回一个进程可以打开的最大描述符数，但是这两个函数是不可移植的。

`OPEN_MAX`被POSIX称为运行时间不变值，其意思是在一个进程的生命期间其值不应被改变，但是在SVR4和4.3+BSD之下，可以调用`setrlimit(2)`函数更改一个运行进程的此值（也可用C Shell的`limit`或Bourne shell和KornShell的`ulimit`命令更改）。如果系统支持这种功能，则可以将程序 2-3更改为每次调用此程序时就调用`sysconf`，而不只是第一次调用此程序时。

2.6 功能测试宏

正如前述，在头文件中定义了很多POSIX.1和XPG3的符号。但是除了POSIX.1和XPG3定义外，大多数实现在这些头文件中也加上了它们自己的定义。如果在编译一个程序时，希望它只使用POSIX定义而不使用任何实现定义的限制，那么就需定义常数`_POSIX_SOURCE`，所有POSIX.1头文件中都使用此常数。当该常数定义时，就能排除任何实现专有的定义。

常数`_POSIX_SOURCE`及其对应的常数`_XOPEN_SOURCE`被称之为功能测试宏（feature test macro）。所有功能测试宏都以下划线开始。当要使用它们时，通常在`cc`命令行中以下列方式定义：

```
cc -D_POSIX_SOURCE file.c
```

这使得在C程序包括任何头文件之前，定义了功能测试宏。如果我们仅想使用POSIX.1定义，那么也可将源文件的第一行设置为：

```
#define _POSIX_SOURCE 1
```

另一个功能测试宏是：`__STDC__`，它由符合ANSI C标准的编译程序自动定义。这样就允许我们编写ANSI C编译程序和非ANSI C编译程序都能编译的程序。例如，一个头文件可能是：

```
#ifndef __STDC__
void *myfunc(const char *, int);
#else
void *myfunc();
#endif
```

这样就能发挥ANSI C原型功能的长处，要注意在开始和结束处的两个连续的下划线常常打印成一个长下划线(如同上面一个样本源代码中一样)。

2.7 基本系统数据类型

历史上，某些UNIX变量已与某些C数据类型联系在一起，例如，历史上主、次设备号存放在一个16位的短整型中，8位表示主设备号，另外8位表示次设备号。但是，很多较大的系统需要用多于256个值来表示其设备号，于是，就需要有一种不同的技术。（确实，SVR4用32位表示设备号：14位用于主设备号，18位用于次设备号。）

头文件`<sys/types.h>`中定义了某些与实现有关的数据类型，它们被称之为基本系统数据类

型 (primitive system data type)。有很多这种数据类型定义在其他头文件中。在头文件中这些数据类型都是用C的typedef设施来定义的。它们绝大多数都以_t结尾。表2-8中列出了本书将使用的基本系统数据类型。

表2-8 基本系统数据类型

类 型	说 明
caddr_t	内存地址 (12.9节)
clock_t	时钟滴答计数器 (进程时间) (1.10节)
comp_t	压缩的时钟滴答 (8.13节)
dev_t	设备号 (主和次) (4.23节)
fd_set	文件描述符集 (12.5.1节)
fpos_t	文件位置 (5.10节)
gid_t	数值组ID
ino_t	i节点编号 (4.14节)
mode_t	文件类型, 文件创建方式 (4.5节)
nlink_t	目录项的连接计数 (4.10节)
off_t	文件长度和位移量 (带符号的) (lseek, 3.6节)
pid_t	进程ID和进程组ID (带符号的) (8.2和9.4节)
ptrdiff_t	两个指针相减的结果 (带符号的)
rlim_t	资源限制 (7.11节)
sig_atomic_t	能原子地存取的数据类型 (10.15节)
sigset_t	信号集 (10.11节)
size_t	对象 (例如字符串) 长度 (不带符号的) (3.7节)
ssize_t	返回字节计数的函数 (带符号的) (read, write, 3.7节)
time_t	日历时间的秒计数器 (1.10节)
uid_t	数值用户ID
wchar_t	能表示所有不同的字符码

用这种方式定义了这些数据类型后,在编译时就不再需要考虑随系统不同而变的实施细节,在本书中涉及到这些数据类型的地方,我们会说明为什么使用它们。

2.8 标准之间的冲突

就整体而言,这些不同的标准之间配合得是相当好的。但是我们也关注它们之间的差别,特别是ANSI C标准和POSIX.1之间的差别。(因为XPG3是一个较老的正在被修订的标准, FIPS则是一个要求更严的POSIX.1。)

ANSI C定义了函数clock,它返回进程使用的CPU时间,返回值是clock_t类型值。为了将此值变换成以秒为单位,将其除以在<time.h>头文件中定义的CLOCKS_PER_SEC。POSIX.1定义了函数times,它返回其调用者及其所有终止子进程的CPU时间以及时钟时间,所有这些值都是clock_t类型值。IEEE Std.1003.1-1988将符号CLK_TCK定义为每秒滴答数,上述clock_t值都是以此度量的。而1990 POSIX.1标准中则说明不再使用,CLK_TCK而使用sysconf函数来获得每秒滴答数,并将其用于times函数的返回值。术语是同一个,每秒滴答数,但ANSI C和POSIX.1的定义却不同。这两个标准也用同一数据类型(clock_t)来保存这些不同的值,这种差别可以在SVR4中看到,其中clock返回微秒数(CLOCK_PER_SEC是一百万),而CLK_TCK通常是50、60或100(与CPU类型有关)。

另一个可能产生冲突的区域是：在ANSI C标准说明函数时，ANSI C所说明的函数可能会没有考虑到POSIX.1的某些要求。有些函数在POSIX环境下可能要求有一个与C环境下不同的实现，因为POSIX环境中有多进程，而C语言环境则很少会考虑宿主操作系统。尽管如此，很多POSIX依从的系统为了兼容性的关系也实现ANSI C函数，`signal`函数就是一个例子。如果在不了解的情况下使用了SVR4所提供的`signal`函数(希望编写可在ANSI C环境和较早UNIX系统中运行的可兼容程序)，那么它提供了与POSIX.1 `sigaction`函数不同的语义。第10章将对`signal`函数作更多说明。

2.9 小结

在过去几年中，在UNIX不同版本的标准化方面已经取得了很大进展。本章对三个主要标准——ANSI C、POSIX和XPG3——进行了说明，也分析了这些标准对本书主要关注的两个实现：SVR4和4.3+BSD所产生的影响。这些标准都试图定义一些可能随实现而更改的参数，但是我们已经看到这些限制并不完善。本书将涉及所有这些限制和幻常数。

在本书最后的参考书目中，说明了如何订购这些标准的方法。

习题

2.1 2.7节中提到一些基本系统数据类型可以在多个头文件中定义。例如，`size_t`就在6个不同的头文件中都有定义。由于一个程序可能包含这6个不同的头文件，但是ANSI C不允许对同一个名字进行多次类型定义，如何处理这个矛盾呢？

2.2 检查系统的头文件，列出实现基本系统数据类型所用到的实际数据类型。

第3章 文 件 I/O

3.1 引言

本章开始讨论UNIX系统，先说明可用的文件I/O函数——打开文件、读文件、写文件等等。大多数UNIX文件I/O只需用到5个函数：open、read、write、lseek 以及close。然后说明不同缓存器长度对read和write函数的影响。

本章所说明的函数经常被称之为不带缓存的I/O（unbuffered I/O，与将在第5章中说明的标准I/O函数相对照）。术语——不带缓存指的是每个read和write都调用内核中的一个系统调用。这些不带缓存的I/O函数不是ANSI C的组成部分，但是是POSIX.1和XPG3的组成部分。

只要涉及在多个进程间共享资源，原子操作的概念就变成非常重要。我们将通过文件 I/O和传送给open函数的参数来讨论此概念。并进一步讨论在多个进程间如何共享文件，并涉及内核的有关数据结构。在讨论了这些特征后，将说明dup、fcntl和ioctl函数。

3.2 文件描述符

对于内核而言，所有打开文件都由文件描述符引用。文件描述符是一个非负整数。当打开一个现存文件或创建一个新文件时，内核向进程返回一个文件描述符。当读、写一个文件时，用open或creat返回的文件描述符标识该文件，将其作为参数传送给 read或write。

按照惯例，UNIX shell使文件描述符0与进程的标准输入相结合，文件描述符1与标准输出相结合，文件描述符2与标准出错输出相结合。这是UNIX shell以及很多应用程序使用的惯例，而与内核无关。尽管如此，如果不遵照这种惯例，那么很多UNIX应用程序就不能工作。

在POSIX.1应用程序中，幻数0、1、2应被代换成符号常数STDIN_FILENO、STDOUT_FILENO和STDERR_FILENO。这些常数都定义在头文件<unistd.h>中。

文件描述符的范围是0~OPEN_MAX(见表2-7)。早期的UNIX版本采用的上限值是19(允许每个进程打开20个文件)，现在很多系统则将其增加至63。

SVR4和4.3+BSD对文件描述符的变化范围没有作规定，它只受到系统配置的存储器的总量、整型字的字长以及系统管理员所配置的软性或硬性限制的约束。

3.3 open函数

调用open函数可以打开或创建一个文件。

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

int open(const char *pathname, int oflag, ... /* mode_t mode */);
```

返回：若成功为文件描述符，若出错为-1

我们将第三个参数写为 ...，这是ANSI C说明余下参数的数目和类型可以变化的方法。对

于open函数而言，仅当创建新文件时才使用第三个参数。（我们将在稍后对此进行说明。）在函数原型中此参数放置在注释中。

*pathname*是要打开或创建的文件的名称。*oflag*参数可用来说明此函数的多个选择项。用下列一个或多个常数进行或运算构成*oflag*参数(这些常数定义在<fcntl.h>头文件中)：

- O_RDONLY 只读打开。
- O_WRONLY 只写打开。
- O_RDWR 读、写打开。

很多实现将O_RDONLY定义为0，O_WRONLY定义为1，O_RDWR定义为2，以与早期的系统兼容。

在这三个常数中应当只指定一个。下列常数则是可选择的：

- O_APPEND 每次写时都加到文件的尾端。3.11节将详细说明此选择项。
- O_CREAT 若此文件不存在则创建它。使用此选择项时，需同时说明第三个参数*mode*，用其说明该新文件的存取许可权位。（4.5节将说明文件的许可权位，那时就能了解如何说明*mode*，以及如何用进程的umask值修改它。）

- O_EXCL 如果同时指定了O_CREAT，而文件已经存在，则出错。这可测试一个文件是否存在，如果不存在则创建此文件成为一个原子操作。3.11节将较详细地说明原子操作。

- O_TRUNC 如果此文件存在，而且为只读或只写成功打开，则将其长度截短为0。

- O_NOCTTY 如果*pathname*指的是终端设备，则不将此设备分配作为此进程的控制终端。9.6节将说明控制终端。

- O_NONBLOCK 如果*pathname*指的是一个FIFO、一个块特殊文件或一个字符特殊文件，则此选择项为此文件的本次打开操作和后续的I/O操作设置非阻塞方式。12.2节将说明此工作方式。

较早的系统V版本引入了O_NDELAY（不延迟）标志，它与O_NONBLOCK（不阻塞）选择项类似，但在读操作的返回值中具有两义性。如果不能从管道、FIFO或设备读得数据，则不延迟选择项使read返回0，这与表示已读到文件尾端的返回值0相冲突。SVR4仍支持这种语义的不延迟选择项，但是新的应用程序应当使用不阻塞选择项以代替之。

- O_SYNC 使每次write都等到物理I/O操作完成。3.13节将使用此选择项。

O_SYNC选择项不是POSIX.1的组成部分，但SVR4支持此选择项。

由open返回的文件描述符一定是最小的未用描述符数字。这一点被很多应用程序用来在标准输入、标准输出或标准出错输出上打开一个新的文件。例如，一个应用程序可以先关闭标准输出(通常是文件描述符1)，然后打开另一个文件，事先就能了解到该文件一定会在文件描述符1上打开。在3.12节说明dup2函数时，可以了解到有更好的方法来保证在一个给定的描述符上打开一个文件。

文件名和路径名截短

如果NAME_MAX是14，而我们却试图在当前目录中创建一个其文件名包含15个字符的新

文件，此时会发生什么呢？按照传统，早期的系统 V 版本，允许这种使用方法，但是总是将文件名截短为 14 个字符，而 BSD 类的系统则返回出错 ENAMETOOLONG。这一问题不仅仅与创建新文件有关。如果 NAME_MAX 是 14，而存在一个其文件名恰恰就是 14 个字符的文件，那么以 *pathname* 作为其参数的任一函数 (*open*, *stat* 等) 都会遇到这一问题。

在 POSIX.1 中，常数 `_POSIX_NO_TRUNC` 决定了是否要截短过长的文件名或路径名，或者返回一个出错。第 12 章将说明此值可以针对各个不同的文件系统进行变更。

FIPS 151-1 要求返回出错。

SVR4 对传统的系统 V 文件系统 (S5) 并不保证返回出错 (见表 2-6)，但是对 BSD 风格的文件系统 (UFS)，SVR4 保证返回出错，4.3+BSD 总是返回出错。

若 `_POSIX_NO_TRUNC` 有效，则在整个路径名超过 `PATH_MAX`，或路径名中的任一文件名超过 `NAME_MAX` 时，返回出错 ENAMETOOLONG。

3.4 creat 函数

也可用 `creat` 函数创建一个新文件。

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

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

返回：若成功为只写打开的文件描述符，若出错为 -1

注意，此函数等效于：

```
open(pathname, O_WRONLY | O_CREAT | O_TRUNC, mode);
```

在早期的 UNIX 版本中，`open` 的第二个参数只能是 0、1 或 2。没有办法打开一个尚未存在的文件，因此需要另一个系统调用 `creat` 以创建新文件。现在，`open` 函数提供了选择项 `O_CREAT` 和 `O_TRUNC`，于是也就不再需要 `creat` 函数了。

在 4.5 节中，我们将详细说明文件存取许可权，并说明如何指定 *mode*。

`creat` 的一个不足之处是它以只写方式打开所创建的文件。在提供 `open` 的新版本之前，如果要创建一个临时文件，并要先写该文件，然后又读该文件，则必须先调用 `creat`，`close`，然后再调用 `open`。现在则可用下列方式调用 `open`：

```
open(pathname, O_RDWR | O_CREAT | O_TRUNC, mode);
```

3.5 close 函数

可用 `close` 函数关闭一个打开文件：

```
#include <unistd.h>

int close (int fildes);
```

返回：若成功为 0，若出错为 -1

关闭一个文件时也释放该进程加在该文件上的所有记录锁。12.3节将讨论这一点。

当一个进程终止时，它所有的打开文件都由内核自动关闭。很多程序都使用这一功能而不显式地用close关闭打开的文件。实例见程序1-2。

3.6 lseek函数

每个打开文件都有一个与其相关联的“当前文件位移量”。它是一个非负整数，用以度量从文件开始处计算的字节数。(本节稍后将对“非负”这一修饰词的某些例外进行说明。)通常，读、写操作都从当前文件位移量处开始，并使位移量增加所读或写的字节数。按系统默认，当打开一个文件时，除非指定O_APPEND选择项，否则该位移量被设置为0。

可以调用lseek显式地定位一个打开文件。

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

off_t lseek(int fildes, off_t offset, int whence);
```

返回：若成功为新的文件位移，若出错为 - 1

对参数offset的解释与参数whence的值有关。

- 若whence是SEEK_SET，则将该文件的位移量设置为距文件开始处offset个字节。
- 若whence是SEEK_CUR，则将该文件的位移量设置为其当前值加offset，offset可为正或负。
- 若whence是SEEK_END，则将该文件的位移量设置为文件长度加offset，offset可为正或负。

若lseek成功执行，则返回新的文件位移量，为此可以用下列方式确定一个打开文件的当前位移量：

```
off_t currpos;

currpos = lseek(fd, 0, SEEK_CUR);
```

这种方法也可用来确定所涉及的文件是否可以设置位移量。如果文件描述符引用的是一个管道或FIFO，则lseek返回 - 1，并将errno设置为EPIPE。

三个符号常数SEEK_SET，SEEK_CUR和SEEK_END是由系统V引进的。在系统V之前，whence被指定为0(绝对位移量)，1(相对于当前位置的位移量)或2(相对文件尾端的位移量)。很多软件仍直接使用这些数字进行编码。

在lseek中的字符l表示长整型。在引入off_t数据类型之前，offset参数和返回值是长整型的。lseek是由V7引进的，当时C语言中增加了长整型。(在V6中，用函数seek和tell提供类似功能。)

实例

程序3-1用于测试其标准输入能否被设置位移量。

程序3-1 测试标准输入能否被设置位移量

```
#include <sys/types.h>
#include "ourhdr.h"

int
main(void)
{
```

```
if (lseek(STDIN_FILENO, 0, SEEK_CUR) == -1)
    printf("cannot seek\n");
else
    printf("seek OK\n");
exit(0);
}
```

如果用交互方式调用此程序，则可得：

```
$ a.out < /etc/motd
seek OK
$ cat < /etc/motd a.out
cannot seek
$ a.out < /var/spool/cron/FIFO
cannot seek
```

通常，文件的当前位移量应当是一个非负整数，但是，某些设备也可能允许负的位移量。但对于普通文件，则其位移量必须是非负值。因为位移量可能是负值，所以在比较 `lseek` 的返回值时应当谨慎，不要测试它是否小于 0，而要测试它是否等于 -1。

在 80386 上运行的 SVR4 支持 `/dev/kmem` 设备，它可以具有负的位移量。

因为位移量 (`off_t`) 是带符号数据类型 (见表 2-8)，所以文件最大长度减少一半。例如，若 `off_t` 是 32 位整型，则文件最大长度是 2^{31} 字节。

`lseek` 仅将当前的文件位移量记录在内核内，它并不引起任何 I/O 操作。然后，该位移量用于下一个读或写操作。

文件位移量可以大于文件的当前长度，在这种情况下，对该文件的下一次写将延长该文件，并在文件中构成一个空洞，这一点是允许的。位于文件中但没有写过的字节都被读为 0。

实例

程序 3-2 用于创建一个具有空洞的文件。

程序 3-2 创建一个具有空洞的文件

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

char buf1[] = "abcdefghij";
char buf2[] = "ABCDEFGHIJ";

int
main(void)
{
    int fd;

    if ( (fd = creat("file.hole", FILE_MODE)) < 0)
        err_sys("creat error");

    if (write(fd, buf1, 10) != 10)
        err_sys("buf1 write error");
    /* offset now = 10 */

    if (lseek(fd, 40, SEEK_SET) == -1)
        err_sys("lseek error");
```

```

/* offset now = 40 */
if (write(fd, buf2, 10) != 10)
    err_sys("buf2 write error");
/* offset now = 50 */

exit(0);
}

```

运行该程序得到：

```

$ a.out
$ ls -l file.hole          检查其大小
-rw-r--r--  1 stevens   50 Jul 31 05:50 file.hole
$ od -c file.hole        观察实际内容
0000000  a  b  c  d  e  f  g  h  i  j  \0  \0  \0  \0  \0  \0
0000020  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
0000040  \0  \0  \0  \0  \0  \0  \0  \0  A  B  C  D  E  F  G  H
0000060  I  J
0000062

```

使用od(1)命令观察该文件的实际内容。命令行中的-c标志表示以字符方式打印文件内容。从中可以看到，文件中间的30个未写字节都被读成为0。每一行开始的一个七位数是以八进制形式表示的字节位移量。本例调用了将在3.8节中说明的write函数。4.12节将对具有空洞的文件进行更多说明。

3.7 read函数

用read函数从打开文件中读数据。

```

#include <unistd.h>

ssize_t read(int fildes, void buff, size_t nbytes);

```

返回：读到的字节数，若已到文件尾为0，若出错为-1

如read成功，则返回读到的字节数。如已到达文件的尾端，则返回0。

有多种情况可使实际读到的字节数少于要求读字节数：

- 读普通文件时，在读到要求字节数之前已到达了文件尾端。例如，若在到达文件尾端之前还有30个字节，而要求读100个字节，则read返回30，下一次再调用read时，它将返回0(文件尾端)。

- 当从终端设备读时，通常一次最多读一行(第11章将介绍如何改变这一点)。
- 当从网络读时，网络中的缓冲机构可能造成返回值小于所要求读的字节数。
- 某些面向记录的设备，例如磁带，一次最多返回一个记录。

读操作从文件的当前位移量处开始，在成功返回之前，该位移量增加实际读得的字节数。

POSIX.1在几个方面对此函数的原型作了更改。其经典定义是：

```
int read(int fildes, char buff, unsigned nbytes);
```

首先，为了与ANSI C一致，其第二个参数由char *改为void *。在ANSI C中，类型void *用于表示类属指针。其次，其返回值必须是一个带符号整数(ssize_t)，以返回正字节数、0(表示文件尾端)或-1(出错)。最后，第三个参数在历史上是一个不带符号整数，以允许一个16位的实现可以一次读或写至65534个字节。在1990 POSIX.1标准中，引进了新的基本系统数据类型

ssize_t 以提供带符号的返回值，size_t则被用于第三个参数（见表2-7中的SSIZE_MAX常数）。

3.8 write函数

用write函数向打开文件写数据。

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

```
ssize_t write(int filedes, const void * buff, size_t nbytes);
```

返回：若成功为已写的字节数，若出错为 - 1

其返回值通常与参数*nbytes*的值不同，否则表示出错。write出错的一个常见原因是：磁盘已写满，或者超过了对一个给定进程的文件长度限制（见7.11节及习题10.11）。

对于普通文件，写操作从文件的当前位移量处开始。如果在打开该文件时，指定了O_APPEND选择项，则在每次写操作之前，将文件位移量设置在文件的当前结尾处。在一次成功写之后，该文件位移量增加实际写的字节数。

3.9 I/O的效率

程序3-3只使用read和write函数来复制一个文件。关于该程序应注意下列各点：

- 它从标准输入读，写至标准输出，这就假定在执行本程序之前，这些标准输入、输出已由shell安排好。确实，所有常用的UNIX shell都提供一种方法，它在标准输入上打开一个文件用于读，在标准输出上创建(或重写)一个文件。
- 很多应用程序假定标准输入是文件描述符0，标准输出是文件描述符1。本例中则用两个在<unistd.h>中定义的名字STDIN_FILENO和STDOUT_FILENO。
- 考虑到进程终止时，UNIX会关闭所有打开文件描述符，所以此程序并不关闭输入和输出文件。
- 本程序对文本文件和二进制代码文件都能工作，因为对UNIX内核而言，这两种文件并无区别。

程序3-3 将标准输入复制到标准输出

```
#include "ourhdr.h"
#define BUFSIZE 8192
int
main(void)
{
    int n;
    char buf[BUFSIZE];
    while ( (n = read(STDIN_FILENO, buf, BUFSIZE)) > 0)
        if (write(STDOUT_FILENO, buf, n) != n)
            err_sys("write error");
    if (n < 0)
        err_sys("read error");
    exit(0);
}
```

我们没有回答的一个问题是如何选取 `BUFSIZE` 值。在回答此问题之前，让我们先用各种不同的 `BUFSIZE` 值来运行此程序。表 3-1 显示了用 18 种不同的缓存长度，读 1 468 802 字节文件所得到的结果。

表 3-1 用不同缓存长度进行读操作的时间结果

BUFSIZE	用户 CPU (秒)	系统 CPU (秒)	时钟时间 (秒)	循环次数
1	23.8	397.9	423.4	1 468 802
2	12.3	202.0	215.2	734 401
4	6.1	100.6	107.2	367 201
8	3.0	50.7	54.0	183 601
16	1.5	25.3	27.0	91 801
32	0.7	12.8	13.7	45 901
64	0.3	6.6	7.0	22 951
128	0.2	3.3	3.6	11 476
256	0.1	1.8	1.9	5 738
512	0.0	1.0	1.1	2 869
1 024	0.0	0.6	0.6	1 435
2 048	0.0	0.4	0.4	718
4 096	0.0	0.4	0.4	359
8 192	0.0	0.3	0.3	180
16 384	0.0	0.3	0.3	90
32 768	0.0	0.3	0.3	45
65 536	0.0	0.3	0.3	23
131 072	0.0	0.3	0.3	12

程序 3-3 读文件，其标准输出则被重新定向到 `/dev/null` 上。此测试所用的文件系统是伯克利快速文件系统，其块长为 8192 字节。(块长由 `st_blksize` 表示，在 4.12 节中为 8192)。系统 CPU 时间的最小值开始出现在 `BUFSIZE` 为 8192 处，继续增加缓存长度对此时间并无影响。

我们以后还将回到这一实例上。3.13 节将用此说明同步写的效果，5.8 节将比较不带缓存所用的时间及标准 I/O 库所用的时间。

3.10 文件共享

UNIX 支持在不同进程间共享打开文件。在介绍 `dup` 函数之间，需要先说明这种共享。为此先说明内核用于所有 I/O 的数据结构。

内核使用了三种数据结构，它们之间的关系决定了在文件共享方面一个进程对另一个进程可能产生的影响。

(1) 每个进程在进程表中都有一个记录项，每个记录项中有一张打开文件描述符表，可将其视为一个矢量，每个描述符占用一项。与每个文件描述符相关联的是：

- (a) 文件描述符标志。
- (b) 指向一个文件表项的指针。

(2) 内核为所有打开文件维持一张文件表。每个文件表项包含：

- (a) 文件状态标志(读、写、增写、同步、非阻塞等)。
- (b) 当前文件位移量。

(c) 指向该文件v节点表项的指针。

(3) 每个打开文件（或设备）都有一个v节点结构。v节点包含了文件类型和对此文件进行各种操作的函数的指针信息。对于大多数文件，v节点还包含了该文件的i节点（索引节点）。这些信息是在打开文件时从盘上读入内存的，所以所有关于文件的信息都是快速可供使用的。例如，i节点包含了文件的所有者、文件长度、文件所在的设备、指向文件在盘上所使用的实际数据块的指针等等（4.14节较详细地说明了UNIX文件系统，将更多地介绍i节点。）

我们忽略了某些并不影响我们讨论的实现细节。例如，打开文件描述符表通常在用户区而不在进程表中。在SVR4中，此数据结构是一个链接表结构。文件表可以用多种方法实现——不一定是文件表项数组。在4.3+BSD中，v节点包含了实际i节点（见图3-1）。SVR4对于大多数文件系统类型，将v节点存放在i节点中。这些实现细节并不影响我们对文件共享的讨论。

图3-1显示了进程的三张表之间的关系。该进程有两个不同的打开文件——一个文件打开为标准输入（文件描述符0），另一个打开为标准输出（文件描述符为1）。

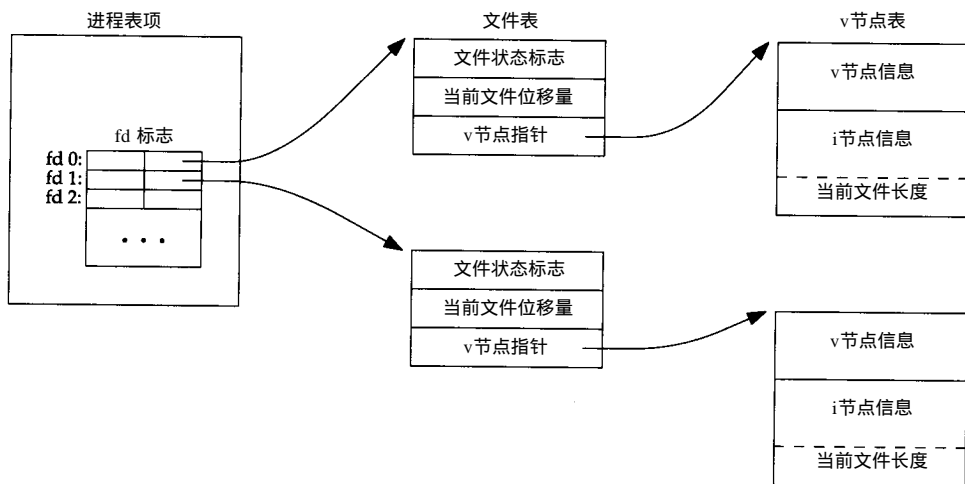


图3-1 打开文件的内核数据结构

从UNIX的早期版本〔Thompson1978〕以来，这三张表之间的基本关系一直保持至今。这种安排对于在不同进程之间共享文件的方式非常重要。在以后的章节中述及其他的文件共享方式时还会回到这张图上来。

v节点结构是近来增设的。当在一个给定的系统上对多种文件系统类型提供支持时，就需要这种结构，这一工作是由Peter Weinberger（贝尔实验室）和Bill Joy（Sun公司）分别独立完成的。Sun称此种文件系统为虚拟文件系统（Virtual File System），称与文件系统类型无关的i节点部分为v节点〔Kleiman 1986〕。当各个制造商的实现增加了对Sun的网络文件系统（NFS）的支持时，它们都广泛采用了v节点结构。

在SVR4中，v节点代换了SVR3中的与文件系统类型无关的i节点结构。

如果两个独立进程各自打开了同一文件，则有图3-2中所示的安排。我们假定第一个进

程使该文件在文件描述符 3 上打开，而另一个进程则使此文件在文件描述符 4 上打开。打开此文件的每个进程都得到一个文件表项，但对一个给定的文件只有一个 v 节点表项。每个进程都有自己的文件表项的一个理由是：这种安排使每个进程都有它自己的对该文件的当前位移量。

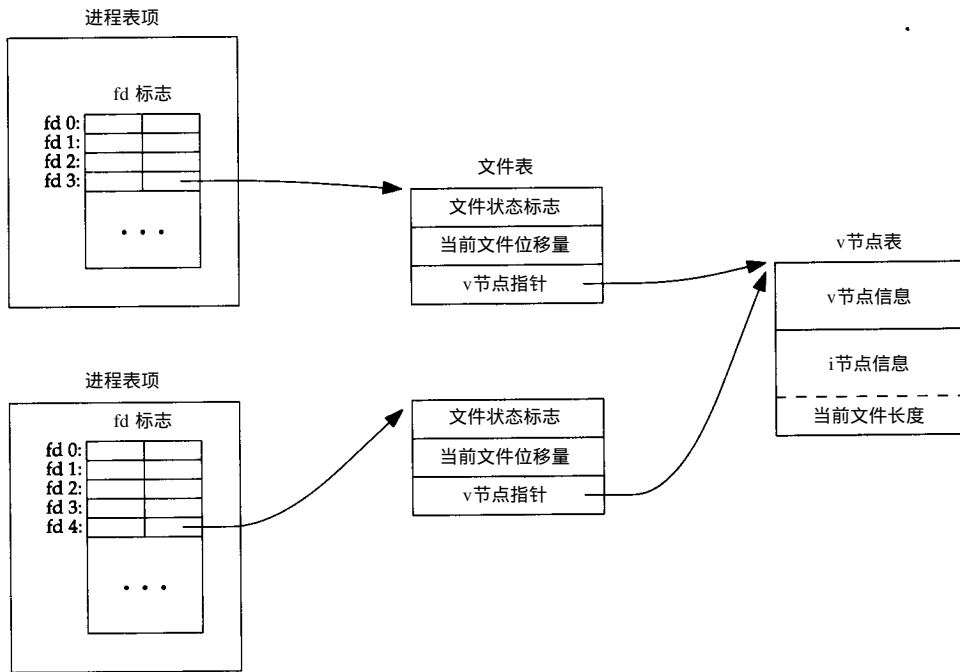


图3-2 两个独立进程各自打开同一个文件

给出了这些数据结构后，现在对前面所述的操作作进一步说明。

- 在完成每个 write 后，在文件表项中的当前文件位移量即增加所写的字节数。如果这使当前文件位移量超过了当前文件长度，则在 i 节点表项中的当前文件长度被设置为当前文件位移量（也就是该文件加长了）。

- 如果用 O_APPEND 标志打开了一个文件，则相应标志也被设置到文件表项的文件状态标志中。每次对这种具有添写标志的文件执行写操作时，在文件表项中的当前文件位移量首先被设置为 i 节点表项中的文件长度。这就使得每次写的数据都添加到文件的当前尾端处。

- lseek 函数只修改文件表项中的当前文件位移量，没有进行任何 I/O 操作。

- 若一个文件用 lseek 被定位到文件当前的尾端，则文件表项中的当前文件位移量被设置为 i 节点表项中的当前文件长度。

可能有多个文件描述符指向同一文件表项。在 3.12 节中讨论 dup 函数时，我们就能看到这一点。在 fork 后也发生同样的情况，此时父、子进程对于每一个打开的文件描述符共享同一个文件表项。

注意，文件描述符标志和文件状态标志在作用范围方面的区别，前者只用于一个进程的一个描述符，而后者则适用于指向该给定文件表项的任何进程中的所有描述符。在 3.13 节说明 fcntl 函数时，我们将会了解如何存取和修改文件描述符标志和文件状态标志。

上述的一切对于多个进程读同一文件都能正确工作。每个进程都有它自己的文件表项，其

中也有它自己的当前文件位移量。但是，当多个进程写同一文件时，则可能产生预期不到的结果。为了说明如何避免这种情况，需要理解原子操作的概念。

3.11 原子操作

3.11.1 添加至一个文件

考虑一个进程，它要将数据添加到一个文件尾端。早期的 UNIX 版本并不支持 `open` 的 `O_APPEND` 选择项，所以程序被编写成下列形式：

```
if (lseek(fd, 0L, 2) < 0)           /*position to EOF*/
    err_sys("lseek error");
if (write(fd, buff, 100) != 100)   /*and write*/
    err_sys("write error");
```

对单个进程而言，这段程序能正常工作，但若有多多个进程时，则会产生问题。（如果此程序由多个进程同时执行，各自将消息添加到一个日记文件中，就会产生这种情况。）

假定有两个独立的进程 A 和 B，都对同一文件进行添加操作。每个进程都已打开了该文件，但未使用 `O_APPEND` 标志。此时各数据结构之间的关系如图 3-2 中所示一样。每个进程都有它自己的文件表项，但是共享一个 `v` 节点表项。假定进程 A 调用了 `lseek`，它将对于进程 A 的该文件的当前位移量设置为 1500 字节(当前文件尾端处)。然后内核切换进程使进程 B 运行。进程 B 执行 `lseek`，也将其对该文件的当前位移量设置为 1500 字节(当前文件尾端处)。然后 B 调用 `write`，它将 B 的该文件的当前文件位移量增至 1600。因为该文件的长度已经增加了，所以内核对 `v` 节点中的当前文件长度更新为 1600。然后，内核又进行进程切换使进程 A 恢复运行。当 A 调用 `write` 时，就从其当前文件位移量 (1500) 处将数据写到文件中去。这样也就代换了进程 B 刚写到该文件中的数据。

这里的问题出在逻辑操作“定位档到文件尾端处，然后写”使用了两个分开的函数调用。解决问题的方法是使这两个操作对于其他进程而言成为一个原子操作。任何一个要求多于 1 个函数调用的操作都不能成为原子操作，因为在两个函数调用之间，内核有可能会临时挂起该进程(正如我们前面所假定的)。

UNIX 提供了一种方法使这种操作成为原子操作，其方法就是在打开文件时设置 `O_APPEND` 标志。正如前一节中所述，这就使内核每次对这种文件进行写之前，都将进程的当前位移量设置到该文件的尾端处，于是在每次写之前就不再需要调用 `lseek`。

3.11.2 创建一个文件

在对 `open` 函数的 `O_CREAT` 和 `O_EXCL` 选择项进行说明时，我们已见到了另一个有关原子操作的例子。当同时指定这两个选择项，而该文件又已经存在时，`open` 将失败。我们曾提及检查该文件是否存在以及创建该文件这两个操作是作为一个原子操作执行的。如果没有这样一个原子操作，那么可能会编写下列程序段：

```
if ((fd = open(pathname, O_WRONLY)) < 0)
    if (errno == ENOENT) {
        if ((fd = creat(pathname, mode)) < 0)
            err_sys("creat error");
    } else
        err_sys("open error");
```

如果在打开和创建之间，另一个进程创建了该文件，那么就会发生问题。如果在这两个函数调用之间，另一个进程创建了该文件，而且又向该文件写进了一些数据，那么执行这段程序中的 `creat` 时，刚写上去的数据就会被擦去。将这两者合并在一个原子操作中，此种问题也就不会产生。

一般而言，原子操作（atomic operation）指的是由多步组成的操作。如果该操作原子地执行，则或者执行完所有步，或者一步也不执行，不可能只执行所有步的一个子集。在 4.15 节论述 `link` 函数以及在 12.3 节中述及记录锁时，还将讨论原子操作。

3.12 dup和dup2函数

下面两个函数都可用复制一个现存的文件描述符：

```
#include <unistd.h>

int dup(int fildes);

int dup2(int fildes, int fildes2);
```

两函数的返回：若成功为新的文件描述符，若出错为 - 1

由 `dup` 返回的新文件描述符一定是当前可用文件描述符中的最小数值。用 `dup2` 则可以用 `fildes2` 参数指定新描述符的数值。如果 `fildes2` 已经打开，则先将其关闭。如若 `fildes` 等于 `fildes2`，则 `dup2` 返回 `fildes2`，而不关闭它。

这些函数返回的新文件描述符与参数 `fildes` 共享同一个文件表项。图 3-3 显示了这种情况。

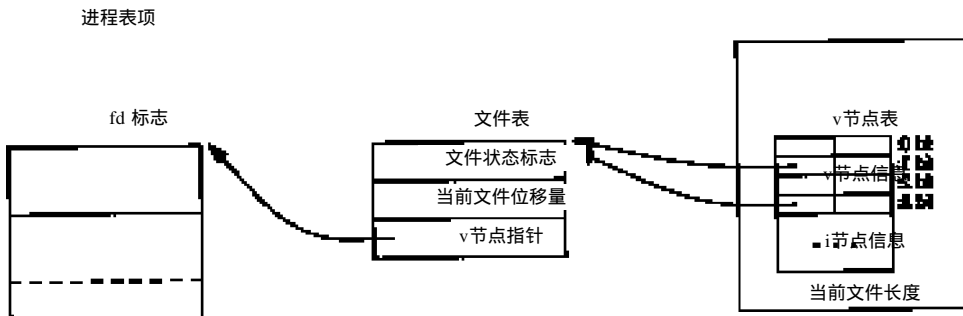


图3-3 dup(1)后内核数据结构

在此图中，我们假定进程执行了：

```
newfd = dup(1);
```

当此函数开始执行时，假定下一个可用的描述符是 3 (这是非常有可能的，因为 0, 1 和 2 由 shell 打开)。因为两个描述符指向同一文件表项，所以它们共享同一文件状态标志 (读、写、添写等) 以及同一当前文件位移量。

每个文件描述符都有它自己的一套文件描述符标志。正如我们将在下一节中说明的那样，新描述符的执行时关闭 (close-on-exec) 文件描述符标志总是由 `dup` 函数清除。

复制一个描述符的另一种方法是使用 `fcntl` 函数，下一节将对该函数进行说明。实际上，调用：

```
dup(fildes);
```

等效于：

```
fcntl (filedes, F_DUPFD, 0);
```

而调用：

```
dup2(filedes, filedes;2)
```

等效于：

```
close(filedes2);
fcntl(filedes, F_DUPFD, filedes2);
```

在最后一种情况下，dup2并不完全等同于close加上fcntl。它们之间的区别是：

(1) dup2是一个原子操作，而close及fcntl则包括两个函数调用。有可能在close和fcntl之间插入执行信号捕获函数，它可能修改文件描述符。(第10章将说明信号。)

(2) 在dup2和fcntl之间有某些不同的errno。

dup2系统调用起源于V7，然后传播至所有BSD版本。而复制文件描述符的fcntl方法则首先由系统III使用，系统V继续采用。SVR3.2选用了dup2函数，4.2BSD则选用了fcntl函数及F_DUPFD功能。POSIX.1要求dup2及fcntl的F_DUPFD功能二者兼有。

3.13 fcntl函数

fcntl函数可以改变已经打开文件的性质。

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

int fcntl(int filedes, int cmd, ... /* intarg */);
```

返回：若成功则依赖于cmd(见下)，若出错为-1

在本节的各实例中，第三个参数总是一个整数，与上面所示函数原型中的注释部分相对应。但是12.3节说明记录锁时，第三个参数则是指向一个结构的指针。

fcntl函数有五种功能：

- 复制一个现存的描述符 (cmd = F_DUPFD)。
- 获得/设置文件描述符标记 (cmd=F_GETFD或F_SETFD)。
- 获得/设置文件状态标志 (cmd=F_GETFL或F_SETFL)。
- 获得/设置异步I/O有权 (cmd=F_GETOWN或F_SETOWN)。
- 获得/设置记录锁 (cmd=F_GETLK,F_SETLK或F_SETLKW)。

我们先说明这十种命令值中的前七种(12.3节说明后三种，它们都与记录锁有关)我们将涉及与进程表项中各文件描述符相关联的文件描述符标志，以及每个文件表项中的文件状态标志，见图3-1。

• F_DUPFD 复制文件描述符filedes，新文件描述符作为函数值返回。它是尚未打开的各描述符中大于或等于第三个参数值(取为整型值)中各值的最小值。新描述符与filedes共享同一文件表项(见图3-3)。但是，新描述符有它自己的一套文件描述符标志，其FD_CLOEXEC文件描述符标志则被清除(这表示该描述符在exec时仍保持开放，我们将在第8章对此进行讨论)。

• `F_GETFD` 对应于 `filedes` 的文件描述符标志作为函数值返回。当前只定义了一个文件描述符标志 `FD_CLOEXEC`。

• `F_SETFD` 对于 `filedes` 设置文件描述符标志。新标志值按第三个参数 (取为整型值) 设置。应当了解很多现存的涉及文件描述符标志的程序并不使用常数 `FD_CLOEXEC`，而是将此标志设置为 0 (系统默认，在 `exec` 时不关闭) 或 1 (在 `exec` 时关闭)。

• `F_GETFL` 对应于 `filedes` 的文件状态标志作为函数值返回。在说明 `open` 函数时，已说明了文件状态标志。它们列于表 3-2 中。

表 3-2 对于 `fcntl` 的文件状态标志

文件状态标志	说 明
<code>O_RDONLY</code>	只读打开
<code>O_WRONLY</code>	只写打开
<code>O_RDWR</code>	读/写打开
<code>O_APPEND</code>	写时都添加至文件尾
<code>O_NONBLOCK</code>	非阻塞方式
<code>O_SYNC</code>	等待写完成
<code>O_ASYNC</code>	异步 I/O (仅 4.3+BSD)

不幸的是，三个存取方式标志 (`O_RDONLY`, `O_WRONLY`, 以及 `O_RDWR`) 并不各占 1 位。(正如同前所述，这三种标志的值各是 0、1 和 2，由于历史原因。这三种值互斥——一个文件只能有这三种值之一。) 因此首先必须用屏蔽字 `O_ACCMODE` 取得存取方式位，然后将结果与这三种值相比较。

• `F_SETFL` 将文件状态标志设置为第三个参数的值 (取为整型值)。可以更改的几个标志是：`O_APPEND`，`O_NONBLOCK`，`O_SYNC` 和 `O_ASYNC`。

• `F_GETOWN` 取当前接收 `SIGIO` 和 `SIGURG` 信号的进程 ID 或进程组 ID。12.6.2 节将论述这两种 4.3+BSD 异步 I/O 信号。

• `F_SETOWN` 设置接收 `SIGIO` 和 `SIGURG` 信号的进程 ID 或进程组 ID。正的 `arg` 指定一个进程 ID，负的 `arg` 表示等于 `arg` 绝对值的一个进程组 ID。

`fcntl` 的返回值与命令有关。如果出错，所有命令都返回 - 1，如果成功则返回某个其他值。下列三个命令有特定返回值：`F_DUPFD`, `F_GETFD`, `F_GETFL` 以及 `F_GETOWN`。第一个返回新的文件描述符，第二个返回相应标志，最后一个返回一个正的进程 ID 或负的进程组 ID。

实例

程序 3-4 取指定一个文件描述符的命令行参数，并对于该描述符打印其文件标志说明。

程序 3-4 对于指定的描述符打印文件标志

```
#include <sys/types.h>
#include <fcntl.h>
#include "ourhdr.h"

int
main(int argc, char *argv[])
{
    int    accmode, val;
```



```

if (argc != 2)
    err_quit("usage: a.out <descriptor#>");

if ( (val = fcntl(atoi(argv[1]), F_GETFL, 0)) < 0)
    err_sys("fcntl error for fd %d", atoi(argv[1]));

accmode = val & O_ACCMODE;
if (accmode == O_RDONLY)    printf("read only");
else if (accmode == O_WRONLY) printf("write only");
else if (accmode == O_RDWR) printf("read write");
else err_dump("unknown access mode");

if (val & O_APPEND)        printf(", append");
if (val & O_NONBLOCK)      printf(", nonblocking");
#if !defined(_POSIX_SOURCE) && defined(O_SYNC)
if (val & O_SYNC)         printf(", synchronous writes");
#endif
putchar('\n');
exit(0);
}

```

注意，我们使用了功能测试宏 `_POSIX_SOURCE`，并且条件编译了 POSIX.1 中没有定义的文件存取标志。下面显示了从 KornShell 调用该程序时的几种情况：

```

$ a.out 0 < /dev/tty
read only
$ a.out 1 > temp.foo
$ cat temp.foo
write only
$ a.out 2 2>>temp.foo
write only, append
$ a.out 5 5<>temp.foo
read write

```

KornShell 子句 `5<>temp.foo` 表示在文件描述符 5 上打开文件 `temp.foo` 以供读、写。

实例

在修改文件描述符标志或文件状态标志时必须谨慎，先要取得现在的标志值，然后按照希望修改它，最后设置新标志值。不能只是执行 `F_SETFD` 或 `F_SETFL` 命令，这样会关闭以前设置的标志位。

程序 3-5 是一个对于一个文件描述符设置一个或多个文件状态标志的函数。

程序 3-5 对一个文件描述符打开一个或多个文件状态标志

```

#include <fcntl.h>
#include "ourhdr.h"

void
set_fl(int fd, int flags) /* flags are file status flags to turn on */
{
    int    val;

    if ( (val = fcntl(fd, F_GETFL, 0)) < 0)
        err_sys("fcntl F_GETFL error");

    val |= flags;        /* turn on flags */

    if (fcntl(fd, F_SETFL, val) < 0)
        err_sys("fcntl F_SETFL error");
}

```

如果将中间的一条语句改为：

```
val &= ~flags;          /*turn flags off*/
```

就构成了另一个函数，我们称其为 `clr_fl`，并将在后面某个例子中用到它。此语句使当前文件状态标志值 `val` 与 `flags` 的反码逻辑与运算。

如果在程序 3-3 的开始处，加上下面一行以调用 `set_fl`，则打开了同步写标志。

```
set_fl(STDOUT_FILENO, O_SYNC);
```

这就造成每次 `write` 都要等待，直至数据已写到磁盘上再返回。在 UNIX 中，通常 `write` 只是将数据排入队列，而实际的 I/O 操作则可能在以后的某个时刻进行。数据库系统很可能需要使用 `O_SYNC`，这样一来，在系统崩溃情况下，它从 `write` 返回时就知道数据已确实写到了磁盘上。

程序运行时，设置 `O_SYNC` 标志会增加时钟时间。为了测试这一点，运行程序 3-3，它从磁盘上的一个文件中将 1.5M 字节复制到另一个文件中。然后，在此程序中设置 `O_SYNC` 标志，使其完成上述同样的工作，将两者的结果进行比较，见表 3-3。

表3-3 用同步写(O_SYNC)的时间结果

操 作	用户CPU (秒)	系统CPU (秒)	时钟时间 (秒)
取自表3-1 BUFSIZE=8192的读时间	0.0	0.3	0.3
盘文件的正常 write	0.0	1.0	2.3
O_SYNC设置的盘文件 write	0.0	1.4	13.4

表3-3中的3行都是在 `BUFSIZE` 为 8192 的情况下测量得到的。表3-1中的结果所测量的情况是读一个磁盘文件，然后写到 `/dev/null`，所以没有磁盘输出。表3-3中的第2行对应于读一个磁盘文件，然后写到另一个磁盘文件中。这就是为什么表3-3中第1, 2行有差别的原因。在写磁盘文件时，系统时间增加了，其原因是内核需要从进程中复制数据，并将数据排入队列以便由磁盘驱动器将其写到磁盘上。当写至磁盘文件时，时钟时间也增加了。当进行同步写时，系统时间稍稍增加，而时钟时间则增加为6倍。

从本例子中，我们看到了 `fcntl` 的必要性。我们的程序在一个描述符(标准输出)上进行操作，但是根本不知道由 `shell` 打开的相应文件的文件名。因为这是 `shell` 打开的，于是不能在打开时，按我们的要求设置 `O_SYNC` 标志。`fcntl` 则允许当仅知道打开文件的描述符时可以修改其性质。在说明非阻塞管道时(14.2节)，我们还将了解到，由于我们对 `pipe` 所具有的标识只是其描述符，所以也需要使用 `fcntl` 的功能。

3.14 ioctl函数

`ioctl` 函数是 I/O 操作的杂物箱。不能用本章中其他函数表示的 I/O 操作通常都能用 `ioctl` 表示。终端 I/O 是 `ioctl` 的最大使用方面(第 11 章将介绍 POSIX.1 已经用新的函数代替 `ioctl` 进行终端 I/O 操作)。

```
#include <unistd.h> /* SVR4 */
#include <sys/ioctl.h> /* 4.3+BSD

int ioctl(int fildes, int request, ...);
```

返回：若出错则为 -1，若成功则为其他值

ioctl函数不是POSIX.1的一部分，但是，SVR4和4.3+BSD用其进行很多杂项设备操作。

我们所示的原型是SVR4和4.3+BSD所使用的，而较早的伯克利系统则将第二个参数说明为unsigned long。因为第二个参数总是一个头文件中的#define名称，所以这种细节并没有什么影响。

对于ANSI C原型，它用省略号表示其余参数。但是，通常另外只有一个参数，它常常是指向一个变量或结构的指针。

在此原型中，我们表示的只是ioctl函数本身所要求的头文件。通常，还要求另外的设备专用头文件。例如，除POSIX.1所说明的基本操作之外，终端ioctl都需要头文件<termios.h>。

目前，ioctl的主要用途是什么呢？我们将4.3+BSD的ioctl操作分类示于表3-4中。

表3-4 4.3+BSD ioctl操作

类 型	常 数 名	头 文 件	ioctl 数
盘标号	DIOxxx	<disklabel.h>	10
文件I/O	FIOxxx	<ioctl.h>	7
磁带I/O	MTIOxxx	<mtio.h>	4
套接口I/O	SIOxxx	<ioctl.h>	25
终端I/O	TIOxxx	<ioctl.h>	35

磁带操作使我们可以磁带上写一个文件结束标志，反绕磁带，越过指定个数的文件或记录等等，用本章中的其他函数(read、write、lseek等)都难于表示这些操作，所以，用ioctl是对这些设备进行操作的最容易的方法。

在11.12节中存取和设置终端窗口，12.4节中说明流系统时，以及19.7节中述及伪终端的高级功能时，都将使用ioctl。

3.15 /dev/fd

比较新的系统都提供名为/dev/fd的目录，其目录项是名为0、1、2等的文件。打开文件/dev/fd/n等效于复制描述符n(假定描述符n是打开的)。

/dev/fd这种特征由Tom Duff开发，它首先出现在Research UNIX System的第8版中，SVR4和4.3+BSD支持这种特征。它不是POSIX.1的组成部分。

在函数中调用：

```
fd = open("/dev/fd/0", mode);
```

大多数系统忽略所指定的mode，而另外一些则要求mode是所涉及的文件(在这里则是标准输入)原先打开时所使用的mode的子集。因为上面的打开等效于：

```
fd = dup(0);
```

描述符0和fd共享同一文件表项(见图3-3)。例如，若描述符0被只读打开，那么我们也只对fd进行读操作。即使系统忽略打开方式，并且下列调用成功：

```
fd = open("/dev/fd/0", O_RDWR);
```

我们仍然不能对fd进行写操作。

我们也可以使用 `/dev/fd` 作为路径名参数调用 `creat`，或调用 `open`，并同时指定 `O_CREAT`。这就允许调用 `creat` 的程序，如果路径名参数是 `/dev/fd/1` 等仍能工作。

某些系统提供路径名 `/dev/stdin`、`/dev/stdout` 和 `/dev/stderr`。这些等效于 `/dev/fd/0`、`/dev/fd/1` 和 `/dev/fd/2`。

`/dev/fd` 文件主要由 shell 使用，这允许程序以对待其他路径名一样的方式使用路径名参数来处理标准输入和标准输出。例如，`cat(1)` 程序将命令行中的一个单独的 `-` 特别解释为一个输入文件名，该文件指的是标准输入。例如：

```
filter file2 | cat file1 - file3 | lpr
```

首先 `cat` 读 `file1`，接着读其标准输入（也就是 `filter file2` 命令的输出），然后读 `file3`，如若支持 `/dev/fd`，则可以删除 `cat` 对 `-` 的特殊处理，于是我们就可键入下列命令行：

```
filter file2 | cat file1 /dev/fd/0 file3 | lpr
```

在命令行中用 `-` 作为一个参数特指标准输入或标准输出已由很多程序采用。但是这会带来一些问题，例如若用 `-` 指定第一个文件，那么它看来就像开始了另一个命令行的选择项。`/dev/fd` 则提高了文件名参数的一致性，也更加清晰。

3.16 小结

本章说明了传统的 UNIX I/O 函数。因为每个 `read`、`write` 都因调用系统调用而进入内核，所以称这些函数为不带缓存的 I/O 函数。在只使用 `read` 和 `write` 情况下，我们观察了不同的 I/O 长度对读文件所需时间的影响。

在说明多个进程对同一文件进行添加操作以及多个进程创建同一文件时，本章介绍了原子操作。也介绍了内核用来共享打开文件信息的数据结构。在本书的稍后部分还将涉及这些数据结构。

我们还介绍了 `ioctl` 和 `fcntl` 函数。第 12 章还将使用这两个函数，将 `ioctl` 用于流 I/O 系统，将 `fcntl` 用于记录锁。

习题

3.1 当读/写磁盘文件时，本章中描述的函数是否有缓存机制？请说明原因。

3.2 编写一个同 3.12 节中的 `dup2` 功能相同的函数，要求不调用 `fcntl` 函数并且要有正确的出错处理。

3.3 假设一个进程执行下面的 3 个函数调用：

```
fd1 = open(pathname, oflags);
fd2 = dup(fd1);
fd3 = open(pathname, oflags);
```

画出结果图（见图 3-3）。对 `fcntl` 作用于 `fd1` 来说，`F_SETFD` 命令会影响哪一个文件描述符？`F_SETFL` 呢？

3.4 在许多程序中都包含下面一段代码：

```
dup2(fd, 0);
dup2(fd, 1);
dup2(fd, 2);
if (fd > 2)
```

```
close(fd);
```

为了说明if语句的必要性，假设fd是1，画出每次调用dup2时3个描述符项及相应的文件表项的变化情况。然后再画出fd为3的情况。

3.5 在Bourne shell和KornShell中，*digit1*>&*digit2*表示要将描述符*digit1*重定向至描述符*digit2*的同一文件。请说明下面两条命令的区别。

```
a.out > outfile 2>&1  
a.out 2>&1 > outfile
```

(提示：shell从左到右处理命令行。)

3.6 如启用添加标志打开一文件以便读、写，能否用 lseek在任一位置开始读？能否用lseek更新文件中任一部分的数据？请写一段程序以验证之。

第4章 文件和目录

4.1 引言

上一章我们说明了执行I/O操作的基本函数。其讨论围绕普通文件的I/O进行——打开一个文件，读或写一个文件。本章将观察文件系统的其他特征和文件的性质。我们将从 `stat` 函数开始，逐个说明 `stat` 结构的每一个成员以了解文件的所有属性。在此过程中，我们将说明修改这些属性的各个函数（更改所有者，更改许可权等），还将更详细地察看 UNIX 文件系统的结构以及符号连接。本章结束部分介绍对目录进行操作的各个函数，并且开发了一个以降序遍历目录层次结构的函数。

4.2 `stat`、`fstat` 和 `lstat` 函数

本章讨论的中心是三个 `stat` 函数以及它们所返回的信息。

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

int stat(const char *pathname, struct stat * buf);

int fstat(int filedes, struct stat * buf);

int lstat(const char *pathname, struct stat * buf);
```

三个函数的返回：若成功则为 0，若出错则为 -1

给定一个 `pathname`，`stat` 函数返回一个与此命名文件有关的信息结构，`fstat` 函数获得已在描述符 `filedes` 上打开的文件的有关信息。`lstat` 函数类似于 `stat`，但是当命名的文件是一个符号连接时，`lstat` 返回该符号连接的有关信息，而不是由该符号连接引用的文件的信息。（在 4.21 节中当以降序遍历目录层次结构时，需要用到 `lstat`。4.16 节将较详细地说明符号连接。）

`lstat` 函数不属于 POSIX 1003.1-1990 标准，但很可能加到 1003.1a 中。SVR4 和 4.3+BSD 支持 `lstat`。

第二个参数是个指针，它指向一个我们应提供的结构。这些函数填写由 `buf` 指向的结构。该结构的实际定义可能随实现而有所不同，但其基本形式是：

```
struct stat {
    mode_t    st_mode;      /* file type & mode (permissions) */
    ino_t     st_ino;      /* i-node number (serial number) */
    dev_t     st_dev;      /* device number (filesystem) */
    dev_t     st_rdev;     /* device number for special files */
    nlink_t   st_nlink;    /* number of links */
    uid_t     st_uid;      /* user ID of owner */
    gid_t     st_gid;      /* group ID of owner */
    off_t     st_size;     /* size in bytes, for regular files */
```

```

time_t  st_atime;   /* time of last access */
time_t  st_mtime;  /* time of last modification */
time_t  st_ctime;  /* time of last file status change */
long    st_blksize; /* best I/O block size */
long    st_blocks; /* number of 512-byte blocks allocated */
};

```

POSIX.1未定义st_rdev, st_blksize和st_blocks字段, SVR4和4.3+BSD则定义了这些字段。

注意, 除最后两个以外, 其他各成员都为基本系统数据类型(见2.7节)。我们将说明此结构的每个成员以了解文件属性。

使用stat函数最多的可能是ls -l命令, 用其可以获得有关一个文件的所有信息。

4.3 文件类型

至今我们已介绍了两种不同的文件类型——普通文件和目录。UNIX系统的大多数文件是普通文件或目录, 但是也有另外一些文件类型:

(1) 普通文件(regular file)。这是最常见的文件类型, 这种文件包含了某种形式的数据库。至于这种数据是文本还是二进制数据对于内核而言并无区别。对普通文件内容的解释由处理该文件的应用程序进行。

(2) 目录文件(directory file)。这种文件包含了其他文件的名字以及指向与这些文件有关信息的指针。对一个目录文件具有读许可权的任一进程都可以读该目录的内容, 但只有内核可以写目录文件。

(3) 字符特殊文件(character special file)。这种文件用于系统中某些类型的设备。

(4) 块特殊文件(block special file)。这种文件典型地用于磁盘设备。系统中的所有设备或者是字符特殊文件, 或者是块特殊文件。

(5) FIFO。这种文件用于进程间的通信, 有时也将其称为命名管道。14.5节将对其进行说明。

(6) 套接口(socket)。这种文件用于进程间的网络通信。套接口也可用于在一台宿主机上的进程之间的非网络通信。第15章将用套接口进行进程间的通信。

只有4.3+BSD才返回套接口文件类型, 虽然SVR4支持用套接口进行进程间通信, 但现在是经由套接口函数库实现的, 而不是通过内核内的套接口文件类型, 将来的SVR4版本可能会支持套接口文件类型。

(7) 符号连接(symbolic link)。这种文件指向另一个文件。4.16节将更多地谈及符号连接。

文件类型信息包含在stat结构的st_mode成员中。可以用表4-1中的宏确定文件类型。这些宏的参数都是stat结构中的st_mode成员。

表4-1 在<sys/stat.h>中的文件类型宏

宏	文件类型
S_ISREG()	普通文件
S_ISDIR()	目录文件
S_ISCHR()	字符特殊文件
S_ISBLK()	块特殊文件

(续)

宏	文件类型
S_ISFIFO()	管道或FIFO
S_ISLNK()	符号连接 (POSIX.1或SVR4无此类型)
S_ISSOCK()	套接字 (POSIX.1或SVR4无此类型)

实例

程序4-1取其命令行参数，然后针对每一个命令行参数打印其文件类型。

程序4-1 对每个命令行参数打印文件类型

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

int
main(int argc, char *argv[])
{
    int i;
    struct stat buf;
    char *ptr;

    for (i = 1; i < argc; i++) {
        printf("%s: ", argv[i]);
        if (lstat(argv[i], &buf) < 0) {
            err_ret("lstat error");
            continue;
        }

        if (S_ISREG(buf.st_mode)) ptr = "regular";
        else if (S_ISDIR(buf.st_mode)) ptr = "directory";
        else if (S_ISCHR(buf.st_mode)) ptr = "character special";
        else if (S_ISBLK(buf.st_mode)) ptr = "block special";
        else if (S_ISFIFO(buf.st_mode)) ptr = "fifo";
#ifdef S_ISLNK
        else if (S_ISLNK(buf.st_mode)) ptr = "symbolic link";
#endif
#ifdef S_ISSOCK
        else if (S_ISSOCK(buf.st_mode)) ptr = "socket";
#endif
        else ptr = "*** unknown mode ***";
        printf("%s\n", ptr);
    }
    exit(0);
}
```

程序4-1的样本输出是：

```
$ a.out /vmunix /etc /dev/ttya /dev/sd0a /var/spool/cron/FIFO \
> /bin /dev/prnter
/vmunix: regular
/etc: directory
/dev/ttya: character special
/dev/sd0a: block special
/var/spool/cron/FIFO: fifo
/bin: symbolic link
/dev/prnter: socket
```


其中，在第一命令行末端我们键入了一个反斜线，通知 shell要在下一行继续键入命令，然后 shell在下一行上用其第二提示符>。我们特地使用了lstat函数而不是stat函数以便检测符号连接。如若使用了stat函数，则决不会观察到符号连接。

早期的UNIX版本并不提供S_ISxxx宏，于是就需要将st_mode与屏蔽字S_IFMT逻辑与，然后与名为S_IFxxx的常数相比较。SVR4和4.3+BSD在文件<sys/stat.h>中定义了此屏蔽字和相关的常数。如若查看此文件，则可找到S_ISDIR宏定义为：

```
#define S_ISDIR (mode) (((mode) & S_IFMT) == S_IFDIR)
```

我们说过，普通文件是最主要的文件类型，但是观察一下在一个给定的系统中各种文件的比例是很有意思的。表4-2中显示了在一个中等规模的系统中的统计值。这一数据是由4.21节中的程序得到的。

表4-2 不同类型文件的计数值和比例

文件类型	计数值	比例(%)
普通文件	30 369	91.7
目录	1 901	5.7
符号连接	416	1.3
字符特殊	373	1.1
块特殊	61	0.2
套接口	5	0.0
FIFO	1	0.0

4.4 设置-用户-ID和设置-组-ID

与一个进程相关联的ID有六个或更多，它们示于表4-3中。

表4-3 与每个进程相关联的用户ID和组ID

实际用户ID	我们实际上是谁
实际组ID	
有效用户ID	
有效组ID	用于文件存取许可权检查
添加组ID	
保存设置-用户-ID	由exec函数保存
保存设置-组-ID	

- 实际用户ID和实际组ID标识我们究竟是谁。这两个字段在登录时取自口令文件中的登录项。通常，在一个登录会话期间这些值并不改变，但是超级用户进程有方法改变它们，8.10节将说明这些方法。

- 有效用户ID，有效组ID以及添加组ID决定了我们的文件访问权，下一节将对此进行说明(我们已在1.8节中说明了添加组ID)。

- 保存的设置-用户-ID和设置-组-ID在执行一个程序时包含了有效用户ID和有效组ID的副本，在8.10节中说明setuid函数时，将说明这两个保存值的作用。

在POSIX.1中，这些保存的ID是可选择的。一个应用程序在编译时可测试常数 `_POSIX_SAVED_IDS`，或在运行时以参数 `_SC_SAVED_IDS` 调用函数 `sysconf`，以判断此实现是否支持这种特征。SVR4支持此特征。

FIPS 151-1要求POSIX.1的这种可选择特征。

通常，有效用户ID等于实际用户ID，有效组ID等于实际组ID。

每个文件有一个所有者和组所有者，所有者由 `stat` 结构中的 `st_uid` 表示，组所有者则由 `st_gid` 成员表示。

当执行一个程序文件时，进程的有效用户ID通常就是实际用户ID，有效组ID通常是实际组ID。但是可以在文件方式字 (`st_mode`) 中设置一个特殊标志，其定义是“当执行此文件时，将进程的有效用户ID设置为文件的所有者 (`st_uid`)”。与此相类似，在文件方式字中可以设置另一位，它使得执行此文件的进程的有效组ID设置为文件的组所有者 (`st_gid`)。在文件方式字中的这两位被称之为设置-用户-ID (`set-user-ID`) 位和设置-组-ID (`set-group-ID`) 位。

例如，若文件所有者是超级用户，而且设置了该文件的设置-用户-ID位，然后当该程序由一个进程运行时，则该进程具有超级用户优先权。不管执行此文件的进程的实际用户ID是什么，都作这种处理。作为一个例子，UNIX程序 `passwd(1)` 允许任一用户改变其口令，该程序是一个设置-用户-ID程序。因为该程序应能将用户的新口令写入口令文件中（一般是 `/etc/passwd` 或 `/etc/shadow`），而只有超级用户才具有对该文件的写许可权，所以需要设置-用户-ID特征。因为运行设置-用户-ID程序的进程通常得到额外的许可权，所以编写这种程序时要特别谨慎。第8章将更详细地讨论这种类型的程序。

再返回到 `stat` 函数，设置-用户-ID位及设置-组-ID位都包含在 `st_mode` 值中。这两位可用常数 `S_ISUID` 和 `S_ISGID` 测试。

4.5 文件存取许可权

`st_mode` 值也包含了对文件的存取许可权位。当提及文件时，指的是前面所提到的任何类型的文件。所有文件类型（目录，字符特别文件等）都有存取许可权。很多人认为只有普通文件有存取许可权，这是一种误解。

每个文件有9个存取许可权位，可将它们分成三类，见表4-4。

在表4-4开头三行中，术语用户指的是文件所有者。`chmod(1)` 命令用于修改这9个许可权位。该命令允许我们用 `u` 表示用户（所有者），用 `g` 表示组，用 `o` 表示其他。有些书把这三种用户类型分别称之为所有者，组和世界。这会造成混乱，因为 `chmod` 命令用 `o` 表示其他，而不是所有者。我们将使用术语用户、组和其他，以便与 `chmod` 命令一致。

图中的三类存取许可权——读、写及执行——以各种方式由不同的函数使用。我们将这些不同的使用方法列在下

表4-4 9个存取许可权位，取自 `<sys/stat.h>`

st_mode屏蔽	意 义
<code>S_IRUSR</code>	用户-读
<code>S_IWUSR</code>	用户-写
<code>S_IXUSR</code>	用户-执行
<code>S_IRGRP</code>	组-读
<code>S_IWGRP</code>	组-写
<code>S_IXGRP</code>	组-执行
<code>S_IROTH</code>	其他-读
<code>S_IWOTH</code>	其他-写
<code>S_IXOTH</code>	其他-执行

面，当说明这些函数时，再进一步作讨论。

- 第一个规则是，我们用名字打开任一类型的文件时，对该名字中包含的每一个目录，包括它可能隐含的当前工作目录都应具有执行许可权。这就是为什么对于目录其执行许可权位常被称为搜索位的原因。

例如，为了打开文件 `/usr/dict/words`，需要具有对目录 `/`、`/usr`、`/usr/dict` 的执行许可权。然后，需要对该文件本身的适当许可权，这取决于以何种方式打开它（只读，读-写等）。

如果当前目录是 `/usr/dict`，那么为了打开文件 `words`，需要有对该目录的执行许可。如在指定打开文件 `words` 时，可以隐含当前目录，而不用显式地提及 `/usr/dict`，也可使用 `./words`。

注意，对于目录的读许可权和执行许可权的意义不相同。读许可权允许我们读目录，获得在该目录中所有文件名的列表。当一个目录是我们存取文件的路径名的一个分量时，对该目录的执行许可权使我们可通过该目录（也就是搜索该目录，寻找一个特定的文件名）。

引用隐含目录的另一个例子是，如果 `PATH` 环境变量（8.4节将说明）指定了一个我们不具有执行许可权的目录，那么 `shell` 决不会在该目录下找到可执行文件。

- 对于一个文件的读许可权决定了我们是否能够打开该文件进行读操作。这对应于 `open` 函数的 `O_RDONLY` 和 `O_RDWR` 标志。

- 对于一个文件的写许可权决定了我们是否能够打开该文件进行写操作。这对应于 `open` 函数的 `O_WRONLY` 和 `O_RDWR` 标志。

- 为了在 `open` 函数中对一个文件指定 `O_TRUNC` 标志，必须对该文件具有写许可权。

- 为了在一个目录中创建一个新文件，必须对该目录具有写许可权和执行许可权。

- 为了删除一个文件，必须对包含该文件的目录具有写许可权和执行许可权。对该文件本身则不需要有读、写许可权。

- 如果用6个 `exec` 函数（见 8.9节）中的任何一个执行某个文件，都必须对该文件具有执行许可权。

进程每次打开、创建或删除一个文件时，内核就进行文件存取许可权测试，而这种测试可能涉及文件的所有者（`st_uid` 和 `st_gid`），进程的有效 ID（有效用户 ID 和有效组 ID）以及进程的添加组 ID（若支持的话）。两个所有者 ID 是文件的性质，而有效 ID 和添加组 ID 则是进程的性质。内核进行的测试是：

- (1) 若进程的有效用户 ID 是 0（超级用户），则允许存取。这给予了超级用户对文件系统进行处理的最充分的自由。

- (2) 若进程的有效用户 ID 等于文件的所有者 ID（也就是该进程拥有此文件）：

- (a) 若适当的所有者存取许可权位被设置，则允许存取。

- (b) 否则拒绝存取。

适当的存取许可权位指的是，若进程为读而打开该文件，则用户-读位应为 1；若进程为写而打开该文件，则用户-写位应为 1；若进程将执行该文件，则用户-执行位应为 1。

- (3) 若进程的有效组 ID 或进程的添加组 ID 之一等于文件的组 ID：

- (a) 若适当的组存取许可权位被设置，则允许存取。

- (b) 否则拒绝存取。

- (4) 若适当的其他用户存取许可权位被设置，则允许存取，否则拒绝存取。

按顺序执行这四步。注意，如若进程拥有此文件（第(2)步），则按用户存取许可权批准或拒绝该进程对文件的存取——不检查组存取许可权。相类似，若进程并不拥有该文件。但进程属于某个适当的组，则按组存取许可权批准或拒绝该进程对文件的存取——不查看其他用户的存

取许可权。

4.6 新文件和目录的所有权

在第3章中，当说明用 `open` 或 `creat` 创建新文件时，没有说明赋予新文件的用户 ID 和组 ID 的值是什么。4.20 节将说明如何创建一个新目录以及 `mkdir` 函数。关于新目录的所有权的规则与本节将说明的新文件的所有权的规则相同。

新文件的用户 ID 设置为进程的有效用户 ID。关于组 ID，POSIX.1 允许选择下列之一作为新文件的组 ID。

- (1) 新文件的组 ID 可以是进程的有效组 ID。
- (2) 新文件的组 ID 可以是它所在目录的组 ID。

在 SVR4 中，新文件的组 ID 取决于它所在的目录的设置 - 组-ID 位是否设置。如果该目录的这一位已经设置，则新文件的组 ID 设置为目录的组 ID；否则新文件的组 ID 设置为进程的有效组 ID。

4.3+BSD 总是使用目录的组 ID 作为新文件的组 ID。

其他系统允许以一个文件系统作为单位在 POSIX.1 所允许的两种方法中选择一种，为此在 `mount(1)` 命令中使用了一个特殊标志。

FIPS 151-1 要求一个新文件的组 ID 是它所在目录的组 ID。

使用 POSIX.1 所允许的第二种方法（继承目录的组 ID）使得在某个目录下创建的文件和目录都具有该目录的组 ID。于是文件和目录的组所有权从该点向下传递。例如，在 `/var/spool` 目录中就使用这种方法。

正如前面提到的，这种设置组所有权的方法对 4.3+BSD 是系统默认的，对 SVR4 则是可选择的。在 SVR4 之下，必须设置 - 组-ID 位。更进一步，为使这种方法能够正常工作，SVR4 的 `mkdir` 函数要自动地传递一个目录的设置 - 组-ID 位（4.20 节将说明 `mkdir` 就是这样做的）。

4.7 access 函数

正如前面所说，当用 `open` 函数打开一个文件时，内核以进程的有效用户 ID 和有效组 ID 为基础执行其存取许可权测试。有时，进程也希望按其实际用户 ID 和实际组 ID 来测试其存取能力。例如当一个进程使用设置 - 用户-ID，或设置 - 组-ID 特征作为另一个用户（或组）运行时，这就可能需要。即使一个进程可能已经设置 - 用户-ID 为根，它仍可能想验证实际用户能否存取一个给定的文件。`access` 函数是按实际用户 ID 和实际组 ID 进行存取许可权测试的。（经过 4.5 节结束部分中所述的四个步骤，但将有效改为实际。）

```
#include<unistd.h>

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

返回：若成功则为 0，若出错则为 -1

其中，*mode*是表4-5中所列常数的逐位或运算。

表4-5 access函数的*mode*常数，取自<unistd.h>

<i>mode</i>	说 明
R_OK	测试读许可权
W_OK	测试写许可权
X_OK	测试执行许可权
F_OK	测试文件是否存在

实例

程序4-2显示了access函数的使用。

程序4-2 access函数实例

```
#include <sys/types.h>
#include <fcntl.h>
#include "ourhdr.h"

int
main(int argc, char *argv[])
{
    if (argc != 2)
        err_quit("usage: a.out <pathname>");

    if (access(argv[1], R_OK) < 0)
        err_ret("access error for %s", argv[1]);
    else
        printf("read access OK\n");

    if (open(argv[1], O_RDONLY) < 0)
        err_ret("open error for %s", argv[1]);
    else
        printf("open for reading OK\n");

    exit(0);
}
```

下面是该程序的一些运行结果：

```
$ ls -l a.out
-rwxrwxr-x 1 stevens 105216 Jan 18 08:48 a.out
$ a.out a.out
read access OK
open for reading OK
$ ls -l /etc/uucp/Systems
-rw-r----- 1 uucp 1441 Jul 18 15:05 /etc/uucp/Systems
$ a.out /etc/uucp/Systems
access error for /etc/uucp/Systems: Permission denied
open error for /etc/uucp/Systems: Permission denied
$ su
成为超级用户
Password:
输入超级用户口令
# chown uucp a.out
将文件用户改为uucp
# chmod u+s a.out
并打开设置-用户-ID位

$ ls -l a.out
-rwsrwxr-x 1 uucp 105216 Jan 18 08:48 a.out
```

```
# exit                                回到正常用户
$ a.out /etc/uucp/Systems
access error for /etc/uucp/Systems: Permission denied
open for reading OK
```

在本例中，设置 `-用户-ID` 程序可以确定实际用户不能读某个文件，而 `open` 函数却能打开该文件。

在上例及第8章中，我们有时要成为超级用户，以便例示某些功能是如何工作的。如果你使用多用户系统，但无超级用户许可权，那么你就不能完整地重复这些实例。

4.8 umask函数

至此我们已说明了与每个文件相关联的 9 个存取许可权位，在此基础上我们可以说明与每个进程相关联的文件方式创建屏蔽字。

`umask` 函数为进程设置文件方式创建屏蔽字，并返回以前的值。（这是少数几个没有出错返回的函数中的一个。）

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

mode_t umask(mode_t cmask);
```

返回：以前的文件方式创建屏蔽字

其中，参数 `cmask` 由表 4-4 中的 9 个常数 (`S_IRUSR`, `S_IWUSR` 等) 逐位 “或” 构成的。

在进程创建一个新文件或新目录时，就一定会使用文件方式创建屏蔽字（回忆 3.3 和 3.4 节，在那里我们说明了 `open` 和 `creat` 函数。这两个函数都有一个参数 `mode`，它指定了新文件的存取许可权位）。我们将在 4.20 节说明如何创建一个新目录，在文件方式创建屏蔽字中为 1 的位，在文件 `mode` 中的相应位则一定被转成 0。

实例

程序 4-3 创建了两个文件，创建第一个时，`umask` 值为 0，创建第二个时，`umask` 值禁止所有组和其他存取许可权。若运行此程序可得如下结果，从中可见存取许可权是如何设置的。

```
$ umask                                第一次打印当前文件方式创建屏蔽字
02
$ a.out
4 ls -l foo bar
-rw----- 1 stevens      0 Nov 16 16:23 bar
-rw-rw-rw- 1 stevens      0 Nov 16 16:23 foo
$ umask                                观察文件方式创建屏蔽字是否更改
02
```

程序 4-3 umask 函数实例

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

```

#include <fcntl.h>
#include "ourhdr.h"

int
main(void)
{
    umask(0);
    if (creat("foo", S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP |
              S_IROTH | S_IWOTH) < 0)
        err_sys("creat error for foo");

    umask(S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH);
    if (creat("bar", S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP |
              S_IROTH | S_IWOTH) < 0)
        err_sys("creat error for bar");
    exit(0);
}

```

4.9 chmod和fchmod函数

这两个函数使我们可以更改现存文件的存取许可权。

```

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

int chmod(const char *pathname, mode_t mode);

int fchmod(int fd, mode_t mode);

```

两个函数返回：若成功则为0，若出错则为-1

chmod函数在指定的文件上进行操作，而fchmod函数则对已打开的文件进行操作。

fchmod函数并不是POSIX.1的组成部分。它是SVR4和4.3+BSD的扩充部分。

为了改变一个文件的许可权位，进程的有效用户ID必须等于文件的所有者，或者该进程必须具有超级用户许可权。

参数mode是表4-6中所示常数的某种逐位或运算。

表4-6 chmod函数的mode常数，取自<sys/stat.h>

mode	说明
S_ISUID	执行时设置-用户-ID
S_ISGID	执行时设置-组-ID
S_ISVTX	保存正文
S_IRWXU	用户（所有者）读、写和执行
S_IRUSR	用户（所有者）读
S_IWUSR	用户（所有者）写
S_IXUSR	用户（所有者）执行
S_IRWXG	组读、写和执行
S_IRGRP	组读
S_IWGRP	组写
S_IXGRP	组执行

(续)

<i>mode</i>	说 明
S_IRWXO	其他读、写和执行
S_IROTH	其他读
S_IWOTH	其他写
S_IXOTH	其他执行

注意，在表4-6中，有9项是取自表4-4中的9个文件存取许可权位。我们另外加上了两个设置-ID常数(S_IS [UG] ID)，保存-正文常数(S_ISVTX)，以及三个组合常数(S_IRWX [UGO])。(这里使用了标准UNIX字符类算符 []，表示方括号算符中的任何一个字符。例如，最后一个，S_IRWX [UGO]表示了三个常数：S_IRWXU、S_IRWXG和S_IRWXO。这一字符类算符是大多数UNIX shell和很多标准UNIX应用程序都提供的正规表达式的一种形式。)

保存-正文位(S_ISVTX)不是POSIX.1的一部分。我们在下一节说明其目的。

实例

先回忆一下为例示umask函数我们运行程序4-3时，文件foo和bar的最后状态：

```
$ ls -l foo bar
-rw----- 1 stevens      0 Nov 16 16:23 bar
-rw-rw-rw- 1 stevens      0 Nov 16 16:23 foo
```

程序4-4修改了这两个文件的方式。在运行程序4-4后，我们见到的这两个文件的最后状态是：

```
$ ls -l foo bar
-rw-r--r-- 1 stevens      0 Nov 16 16:23 bar
-rw-rwlrw- 1 stevens      0 Nov 16 16:23 foo
```

在本例中，我们相对于foo的当前状态设置其许可权。为此，先调用stat获得其当前许可权，然后修改它。我们已显式地打开了设置-组-ID位、关闭了组-执行位。对普通文件这样做的结果是对该文件可以加强制性记录锁，我们将在12.3节中讨论强制性锁。注意，ls命令将组-执行许可权表示为l，它表示对该文件可以加强制性记录锁。对文件bar，不管其当前许可权位如何，我们将其许可权设置为一绝对值。

程序4-4 chmod函数实例

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

int
main(void)
{
    struct stat    statbuf;

    /* turn on set-group-ID and turn off group-execute */

    if (stat("foo", &statbuf) < 0)
        err_sys("stat error for foo");
    if (chmod("foo", (statbuf.st_mode & ~S_IXGRP) | S_ISGID) < 0)
        err_sys("chmod error for foo");

    /* set absolute mode to "rw-r--r--" */
```



```
if (chmod("bar", S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH) < 0)
    err_sys("chmod error for bar");

exit(0);
}
```

最后也要注意，在我们运行程序4-4后，ls命令列出的时间和日期并不改变。在4.18节中，我们会了解到chmod函数更新的只是i节点最近一次被更改的时间。按系统默认方式，ls -l列出的是最后修改文件内容的时间。

chmod函数在下列条件下自动清除两个许可权位。

- 如果我们试图设置普通文件的粘住位 (S_ISVTX)，而且又没有超级用户优先权，那么mode中的粘住位自动被关闭(我们将在下一节说明粘住位)。这意味着只有超级用户才能设置普通文件的粘住位。这样做的理由是可以防止不怀好意的用户设置粘住位，并试图以此方式填满交换区(如果系统支持保存-正文特征的话)。

- 新创建文件的组ID可能不是调用进程所属的组。回忆一下4.6节，新文件的组ID可能是父目录的组ID。特别地，如果新文件的组ID不等于进程的有效组ID或者进程添加组ID中的一个，以及进程没有超级用户优先数，那么设置-组-ID位自动被关闭。这就防止了用户创建一个设置-组-ID文件，而该文件是由并非该用户所属的组拥有的。

4.3+BSD和其他伯克利导出的系统增加了另外的安全性特征以试图防止保护位的错误使用。如果一个没有超级用户优先权的进程写一个文件，则设置-用户-ID位和设置-组-ID位自动被清除。如果一个不怀好意的用户找到一个他可以写的设置-组-ID和设置-用户-ID文件，即使他可以修改此文件，但失去了对该文件的特别优先权。

4.10 粘住位

S_ISVTX位有一段有趣的历史。在UNIX的早期版本中，有一位被称为粘住位 (sticky bit)。如果一个可执行程序文件的这一位被设置了，那么在该程序第一次执行并结束时，该程序正文的一个文本被保存在交换区。(程序的正文部分是机器指令部分。)这使得下次执行该程序时能较快地将其装入内存区。其原因是：在交换区，该文件是被连续存放的，而在一般的UNIX文件系统中，文件的各数据块很可能是随机存放的。对于常用的应用程序，例如文本编辑程序和编译程序的各部分常常设置它们所在文件的粘住位。自然，对交换区中可以同时存放的设置了粘住位的文件数有一定限制，以免过多占用交换区空间，但无论如何这是一个有用的技术。因为在系统再次自举前，文件的正文部分总是在交换区中，所以使用了名字“粘住”。后来的UNIX版本称之为保存-正文位 (saved-text bit)，因此也就有了常数S_ISVTX。现今较新的UNIX系统大多数都具有虚存系统以及快速文件系统，所以不再需要使用这种技术。

SVR4和4.3+BSD中粘住位的主要针对目录。如果对一个目录设置了粘住位，则只有对该目录具有写许可权的用户并且满足下列条件之一，才能删除或更名该目录下的文件：

- 拥有此文件。
- 拥有此目录。
- 是超级用户。

目录/tmp和/var/spool/uucppublic是设置粘住位的候选者——这两个目录是任何用户都可在其中

创建文件的目录。这两个目录对任一用户(用户、组和其他)的许可权通常都是读、写和执行。但是用户不应能删除或更名属于其他人的文件,为此在这两个目录的文件方式中都设置了粘住位。

POSIX.1没有定义粘住位,但SVR4和4.3+BSD则支持这种特征。

4.11 chown, fchown和lchown函数

chown函数可用于更改文件的用户ID和组ID。

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

int chown(const char *pathname, uid_t owner, gid_t group);

int fchown(int fildes, uid_t owner, gid_t group);

int lchown(const char *pathname, uid_t owner, gid_t group);
```

三个函数返回:若成功则为0,若出错则为-1

除了所引用的文件是符号连接以外,这三个函数的操作相类似。在符号连接情况下,lchown更改符号连接本身的所有者,而不是该符号连接所指向的文件。

fchown函数并不在POSIX 1003.1-1990标准中,但很可能被加到1003.1a中。SVR4和4.3+BSD都支持fchown。

只有SVR4支持lchown函数。在非SVR4系统中(POSIX.1和4.3+BSD),若chown的参数*pathname*是符号连接,则改变该符号连接的所有权,而不改变它所指向的文件的所有权。为了更改该符号连接所指向的文件的所有权,我们应指定该实际文件本身的*pathname*,而不是指向该文件的连接文件的*pathname*。

SVR4,4.3+BSD和XPG3允许将参数*owner*或*group*指定为-1,以表示不改变相应的ID。这不是POSIX.1的一部分。

基于伯克利的系统一直规定只有超级用户才能更改一个文件的所有者。这样做的原因是防止用户改变其文件的所有者从而摆脱磁盘空间限额对他们的限制。系统V则允许任一用户更改他们所拥有的文件的所有者。

按照_POSIX_CHOWN_RESTRICTED的值,POSIX.1在这两种形式的操作中选用一种。FIPS 151-1要求_POSIX_CHOWN_RESTRICTED。

对于SVR4,此功能是个配置选择项,而4.3+BSD则总对chown施加了限制。

回忆表2-5,该常数可选地定义在头文件<unistd.h>中,而且总是可以用pathconf或fpathconf函数查询。此选择项还与所引用的文件有关——可在每个文件系统基础上,使该选择项起作用或不起作用。在下文中,如提及“若_POSIX_CHOWN_RESTRICTED起作用”,则表示这适用于我们正在谈及的文件,而不管该实际常数是否在头文件中定义(例如,4.3+BSD总有这种限制,而并不在头文件中定义此常数)。

若_POSIX_CHOWN_RESTRICTED对指定的文件起作用，则

- (1) 只有超级用户进程能更改该文件的用户ID。
- (2) 若满足下列条件，一个非超级用户进程可以更改该文件的组ID：

- (a) 进程拥有此文件（其有效用户ID等于该文件的用户ID）。
- (b) 参数`owner`等于文件的用户ID，参数`group`等于进程的有效组ID或进程的添加组ID之一。

这意味着，当_POSIX_CHOWN_RESTRICTED有效时，不能更改其他用户的文件的用户ID。你可以更改你所拥有的文件的组ID，但只能改到你所属于的组。

如果这些函数由非超级用户进程调用，则在成功返回时，该文件的设置-用户-ID位和设置-组-ID位都被清除。

4.12 文件长度

`stat`结构的成员`st_size`包含了以字节为单位的该文件的长度。此字段只对普通文件、目录文件和符号连接有意义。

SVR4对管道也定义了文件长度，它表示可从该管道中读到的字节数，我们将在14.2中讨论管道。

对于普通文件，其文件长度可以是0，在读这种文件时，将得到文件结束指示。

对于目录，文件长度通常是一个数，例如16或512的整倍数，我们将在4.21节中说明读目录操作。

对于符号连接，文件长度是在文件名中的实际字节数。例如，

```
lrwxrwxrwx 1 root          7 Sep 25 07:14 lib -> usr/lib
```

其中，文件长度7就是路径名`usr/lib`的长度(注意，因为符号连接文件长度总是由`st_size`指示，所以符号连接并不包含通常C语言用作名字结尾的`null`字符)。

SVR4和4.3+BSD也提供字段`st_blksize`和`st_blocks`。第一个是对文件I/O较好的块长度，第二个是所分配的实际512字节块数。回忆一下3.9节，其中提到了当我们将`st_blksize`用于读操作时，读一个文件所需的最少时间量。为了效率的缘故，标准I/O库(我们将在第5章中说明)也试图一次读、写`st_blksize`字节。

要知道，不同的UNIX版本其`st_blocks`所用的单位可能不是512字节块。使用此值并不是可移植的。

文件中的空洞

在3.6节中，我们提及普通文件可以包含空洞。在程序3-2中例示了这一点。空洞是由超过文件结尾端的位移量设置，并写了某些数据后造成的。作为一个例子，考虑下列情况：

```
$ ls -l core
-rw-r--r-- 1 stevens  8483248 Nov 18 12:18 core
$ du -s core
272      core
```

文件`core`的长度超过8M字节，而`du`命令则报告该文件所使用的磁盘空间总量是272个512字节块(139 264字节)(在很多伯克利类的系统上，`du`命令报告1024字节块数，SVR4则报告512字

节块数)。很明显，此文件有很多空洞。

正如我们在3.6节中提及的，`read`函数对于没有写过的字节位置读到的数据字节是0。如果执行：

```
$ wc -c core
8483248 core
```

由此可见，正常的I/O操作读至整个文件长度（带-c选择项的`wc(1)`命令计算文件中的字符（字节）数）。

如果使用公用程序，例如`cat(1)`，复制这种文件，那么所有这些空洞都被写成实际数据字节0。

```
$ cat core > core.copy
$ ls -l core*
-rw-r--r--  1 stevens   8483248 Nov 18 12:18 core
-rw-rw-r--  1 stevens   8483248 Nov 18 12:27 core.copy
$ du -s core*
272      core
16592    core.copy
```

从中可见，新文件所用的字节数是8 495 104 ($512 \times 16\,592$)。此长度与`ls`命令报告的长度之间的差别是由于文件系统使用了若干块以保持指向实际数据块的各指针。

有兴趣的读者应当参阅Bach [1986]的4.2节和Leffler [1989]的7.2节，以更详细地了解文件的物理安排。

4.13 文件截短

有时我们需要在文件尾端处截去一些数据以缩短文件。将一个文件的长度截短为0是一个特例，用`O_TRUNC`标志可以做到这一点。为了截短文件可以调用函数`truncate`和`ftruncate`。

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

int truncate(const char *pathname, off_t length);

int ftruncate(int fildes, off_t length);
```

两个函数返回；若成功则为0，若出错则为-1

这两个函数将由路径名 *pathname* 或打开文件描述符 *fildes* 指定的一个现存文件的长度截短为 *length*。如果该文件以前的长度大于 *length*，则超过 *length* 以外的数据就不再能存取。如果以前的长度短于 *length*，则其后果与系统有关。如果某个实现的处理是扩展该文件，则在以前的文件尾端和新的文件尾端之间的数据将读作0（也就是在文件中创建了一个空洞）。

SVR4和4.3+BSD提供了这两个函数。它们不是POSIX.1或XPG3的组成部分。

SVR4截短或扩展一个文件。4.3+BSD只用这三个函数截短一个文件——不能用它们扩展一个文件。

UNIX从来就没有截短文件的一种标准方法。完全兼容的应用程序必须对文件制作一个副本，在制作它时只复制所希望的数据字节。

SVR4的`fcntl`中有一个POSIX.1没有规定的命令`F_FREESP`，它允许释放一个文件中的任何一部分，而不只是文件尾端处的一部分。

程序12-5使用了ftruncate函数，以便在获得对该文件的锁后，使一个文件变空。

4.14 文件系统

为了说明文件连接的概念，先要对文件系统的结构有基本了解。同时，了解 i节点和指向一个i节点的目录项之间的区别也是很有益的。

目前有多种UNIX文件系统的实现。例如，SVR4支持两种不同类型的磁盘文件系统：传统的UNIX系统V文件系统（S5），以及统一文件系统（UFS）。在表2-6中，我们已看到了这两种文件系统的区别。UFS是以伯克利快速文件系统为基础的。SVR4也支持另外一些非磁盘文件系统，两个分布式文件系统，以及一个自举文件系统，这些文件系统都不影响下面的讨论。本节讨论传统的UNIX系统V文件系统。这种类型的文件系统可以回溯到V7。

我们可以把一个磁盘分成一个或多个分区。见图4-1，每个分区可以包含一个文件系统。

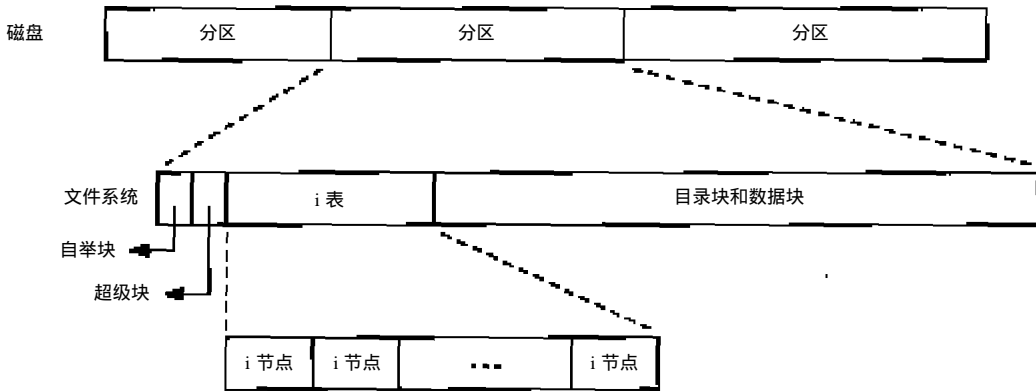


图4-1 磁盘、分区和文件系统

i节点是固定长度的记录项，它包含有关文件的信息。

在V7中，i节点占用64字节，在4.3+BSD中，i节点占用128字节。在SVR4中，在磁盘上一个i节点的长度与文件系统的类型有关：S5 i节点占用64字节，而UFS i节点占用128字节。

如果在忽略自举块和超级块情况下更仔细地观察文件系统，则可以得到图4-2中所示的情况。

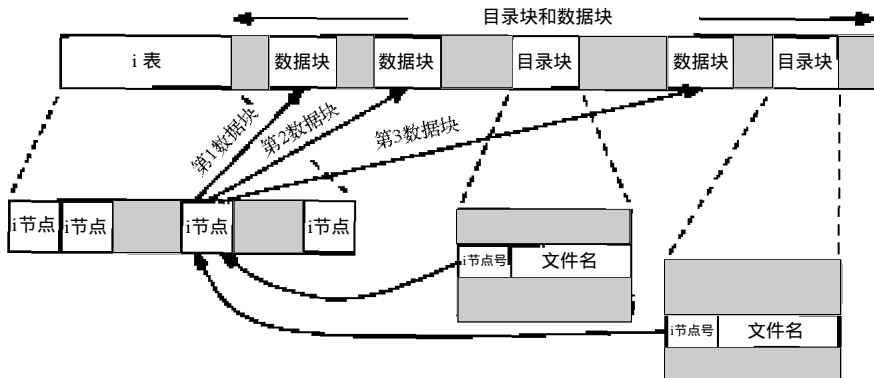


图4-2 较详细的文件系统

注意图4-2中的下列各点：

- 在图中有两个目录项指向同一 i 节点。每个 i 节点中都有一个连接计数，其值是指向该 i 节点的目录项数。只有当连接计数减少为 0 时，才可删除该文件(也就是可以释放该文件占用的数据块)。这就是为什么“解除对一个文件的连接”操作并不总是意味着“释放该文件占用的磁盘块”的原因。这也就是为什么删除一个目录项的函数被称之为 unlink 而不是 delete 的原因。在 stat 结构中，连接计数包含在 st_nlink 成员中，其基本系统数据类型是 nlink_t。这种连接类型称之为硬连接。回忆表 2-7，其中，POSIX.1 常数 LINK_MAX 指定了一个文件连接数的最大值。

- 另外一种连接类型称之为符号连接 (symbolic link)。对于这种连接，该文件的实际内容(在数据块中)包含了该符号连接所指向的文件的名字。在下列中：

```
lrwxrwxrwx 1 root      7 Sep 25 07:14 lib -> usr/lib
```

该目录项中的文件名是 lib，而在该文件中包含了 7 个数据字节 usr/lib。该 i 节点中的文件类型是 S_IFLNK，于是系统知道这是一个符号连接。

- i 节点包含了所有与文件有关的信息：文件类型、文件存取许可权位、文件长度和指向该文件所占用的数据块的指针等等。stat 结构中的大多数信息都取自 i 节点。只有两项数据存放在目录项中：文件名和 i 节点编号数。i 节点编号数的数据类型是 ino_t。

- 因为目录项中的 i 节点编号数指向同一文件系统中的 i 节点，所以不能使一个目录项指向另一个文件系统的 i 节点。这就是为什么 ln(1) 命令 (构造一个指向一个现存文件的新目录项)，不能跨越文件系统的原因。我们将在下一节说明 link 函数。

- 当在不更改文件系统的情况下为一个文件更名时，该文件的实际内容并未移动，只需构造一个指向现存 i 节点的新目录项，并删除老的目录项。例如，为将文件 /usr/lib/foo 更名为 /usr/foo，如果目录 /usr/lib 和 /usr 在同一文件系统上，则文件 foo 的内容无需移动。这就是 mv(1) 命令的通常操作方式。

我们说明了普通文件的连接计数的概念，但是对于目录文件的连接计数字段又如何呢？假定我们在工作目录中构造了一个新目录：

```
$ mkdir testdir
```

图4-3显示了其结果。注意，该图显式地显示了 . 和 .. 目录项。

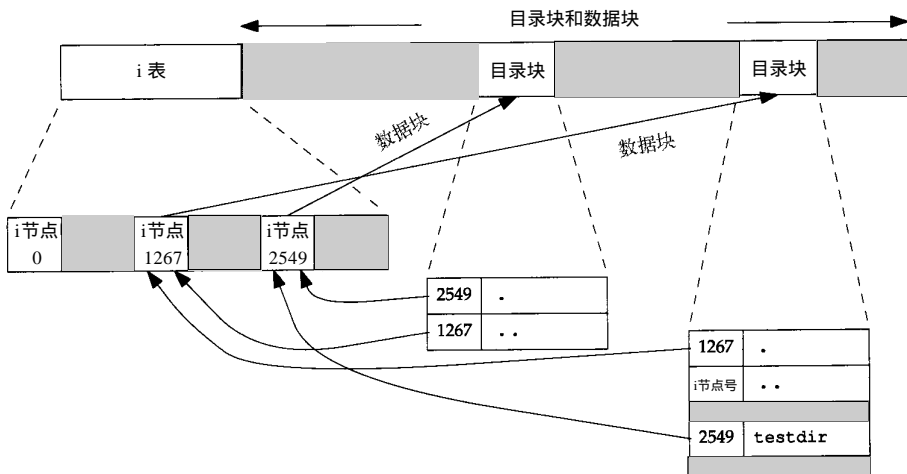


图4-3 创建了目录 testdir 后的文件系统实例

编号为2549的*i*节点，其类型字段表示它是一个目录，而连接计数为2。任何一个叶目录(不包含任何其他目录，也就是子目录的目录)其连接计数总是2，数值2来自于命名该目录(`testdir`)的目录项以及在该目录中的`.`项。编号为1267的*i*节点，其类型字段表示它是一个目录，而其连接计数则大于或等于3。它大于或等于3的原因是，至少有由三个目录项指向它：一个是命名它的目录项(在图4-3中没有表示出来)，第二个是在该目录中的`.`项，第三个是在其子目录`testdir`中的`..`项。注意，在工作目录中的每个子目录都使该工作目录的连接计数增1。

正如前面所述，这是UNIX文件系统的经典格式，在Bach [1986]一书的第4章中对此作了说明。关于伯克利快速文件系统对此所作的更改请参阅Leffler等 [1989]中的第7章。

4.15 link, unlink, remove和rename函数

如上节所述，任何一个文件可以有多个目录项指向其*i*节点。创建一个向现存文件连接的方法是使用`link`函数。

```
#include <unistd.h>

int link(const char *existingpath, const char * newpath);
```

返回：若成功则为0，若出错则为-1

此函数创建一个新目录项`newpath`，它引用现存文件`existingpath`。如若`newpath`已经存在，则返回出错。

创建新目录项以及增加连接计数应当是个原子操作(请回忆在3.11节中对原子操作的讨论)。大多数实现，例如SVR4和4.3 + BSD要求这两个路径名在同一个文件系统中。

POSIX.1允许支持跨越文件系统的连接的实现。

只有超级用户进程可以创建指向一个目录的新连接。其理由是这样做可能在文件系统中形成循环，大多数处理文件系统的公用程序都不能处理这种情况(4.16节将说明一个由符号连接引入的循环的例子)。

为了删除一个现存的目录项，可以调用`unlink`函数。

```
#include <unistd.h>

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

返回：若成功则为0，若出错则为-1

此函数删除目录项，并将由`pathname`所引用的文件的连接计数减1。如果该文件还有其他连接，则仍可通过其他连接存取该文件的数据。如果出错，则不对该文件作任何更改。

我们在前面已经提及，为了解除对文件的连接，必须对包含该目录项的目录具有写和执行许可权。正如4.10节所述，如果对该目录设置了粘住位，则对该目录必须具有写许可权，并且具备下面三个条件之一：

- 拥有该文件。
- 拥有该目录。
- 具有超级用户优先权。

只有当连接计数达到0时，该文件的内容才可被删除。另一个条件也阻止删除文件的内容——只要有进程打开了该文件，其内容也不能删除。关闭一个文件时，内核首先检查使该文件打开的进程计数。如果该计数达到0，然后内核检查其连接计数，如果这也是0，那么就删除该文件的内容。

实例

程序4-5打开一个文件，然后unlink它。执行该程序的进程然后睡眠15秒钟，接着就终止。

程序4-5 打开一个文件，然后unlink它

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

int
main(void)
{
    if (open("tempfile", O_RDWR) < 0)
        err_sys("open error");

    if (unlink("tempfile") < 0)
        err_sys("unlink error");

    printf("file unlinked\n");
    sleep(15);
    printf("done\n");

    exit(0);
}
```

运行该程序，其结果是：

```
$ ls -l tempfile          查看文件大小
-rw-r--r--  1 stevens  9240990 Jul 31 13:42 tempfile
$ df/home                检查空闲空间
Filesystem    kbytes    used    avail capacity  Mounted on
/dev/sd0h    282908   181979   726381%    /home
$ a.out &              在后台运行程序4-5
1364          shell打印其进程ID
$ file unlinked        该文件是未连接的
ls -l tempfile        观察文件是否仍然存在
tempfile not found    目录项已删除
$ df/home              检查空闲空间有无变化
Filesystem    kbytes    used    avail capacity  Mounted on
/dev/sd0h    282908   181979   72638    71%    /home
$ done                程序执行结束，关闭所有打开文件
df/home          磁盘空间有效
Filesystem    kbytes    used    avail capacity  Mounted on
/dev/sd0h    282908   172939   81678    68%    /home
9.2M字节磁盘空间有效
```

unlink的这种特性经常被程序用来确保即使是在程序崩溃时，它所创建的临时文件也不会遗留下来。进程用open或creat创建一个文件，然后立即调用unlink。因为该文件仍旧是打开的，所以不会将其内容删除。只有当进程关闭该文件或终止时（在这种情况下，内核关闭该进程所打开的全部文件），该文件的内容才被删除。

如果`pathname`是符号连接，那么`unlink`涉及的是符号连接而不是由该连接所引用的文件。

超级用户可以调用带参数`pathname`的`unlink`指定一个目录，但是通常不使用这种方式，而使用函数`rmdir`。我们将在4.20节中说明`rmdir`函数。

我们也可以使用`remove`函数解除对一个文件或目录的连接。对于文件，`remove`的功能与`unlink`相同。对于目录，`remove`的功能与`rmdir`相同。

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

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

返回：若成功则为0，若出错则为-1

ANSI C 指定 `remove` 函数删除一个文件，这更改了 UNIX 历来使用的名字 `unlink`，其原因是实现 C 标准的大多数非 UNIX 系统并不支持文件连接。

文件或目录用 `rename` 函数更名。

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

```
int rename(const char *oldname, const char * newname);
```

返回：若成功则为0，若出错则为-1

ANSI C 对文件定义了此函数（C 标准不处理目录）。POSIX.1 扩展此定义包含了目录。

根据`oldname`是指文件还是目录，有两种情况要加以说明。我们也应说明如果`newname`已经存在将会发生什么。

(1) 如果`oldname`说明一个文件而不是目录，那么为该文件更名。在这种情况下，如果`newname`已存在，则它不能引用一个目录。如果`newname`已存在，而且不是一个目录，则先将该目录项删除然后将`oldname`更名为`newname`。对包含`oldname`的目录以及包含`newname`的目录，调用进程必须具有写许可权，因为将更改这两个目录。

(2) 如若`oldname`说明一个目录，那么为该目录更名。如果`newname`已存在，则它必须引用一个目录，而且该目录应当是空目录（空目录指的是该目录中只有`.`和`..`项）。如果`newname`存在（而且是一个空目录），则先将其删除，然后将`oldname`更名为`newname`。另外，当为一个目录更名时，`newname`不能包含`oldname`作为其路径前缀。例如，不能将`/usr/foo`更名为`/usr/foo/testdir`，因为老名字（`/usr/foo`）是新名字的路径前缀，因而不能将其删除。

(3) 作为一个特例，如果`oldname`和`newname`引用同一文件，则函数不做任何更改而成功返回。

如若`newname`已经存在，则调用进程需要对其有写许可权（如同删除情况一样）。另外，调用进程将删除`oldname`目录项，并可能要创建`newname`目录项，所以它需要对包含`oldname`及包含`newname`的目录具有写和执行许可权。

4.16 符号连接

符号连接是对一个文件的间接指针，它与上一节所述的硬连接有所不同，硬连接直接指向

文件的i节点。引进符号连接的原因是为了避免硬连接的一些限制：(a)硬连接通常要求连接和文件位于同一文件系统中，(b)只有超级用户才能创建到目录的硬连接。对符号连接以及它指向什么没有文件系统限制，任何用户都可创建指向目录的符号连接。符号连接一般用于将一个文件或整个目录结构移到系统中其他某个位置。

符号连接由4.2BSD引进，后来又得到SVR4的支持。在SVR4中，传统的系统V文件系统(S5)和统一文件系统(UFS)都支持符号连接。

POSIX 1003.1-1990标准并不包括符号连接，但很可能会加到1003.1a中。

当使用以名字引用一个文件的函数时，应当了解该函数是否处理符号连接功能。也就是是否跟随符号连接到达它所连接的文件。如若该函数处理符号连接功能，则该函数的路径名参数引用由符号连接指向的文件。否则，一个路径名参数引用连接本身，而不是由该连接指向的文件。表4-7列出了本章中所说明的各个函数是否处理符号连接功能。因为 `rmdir` 并不是针对符号连接进行定义的（若 `path` 是符号连接则返回出错），所以在表4-7中没有列出这一函数。因为对符号连接的处理是由返回文件描述符的函数进行的（通常是 `open`），所以以文件描述符作为参数的函数（如 `fstat`, `fchmod` 等）也未列出。`chown` 是否跟随符号连接取决于实现——各种有关细节见4.11节。

表4-7 各个函数对符号连接的处理

函 数	不跟随符号连接	跟随符号连接
<code>access</code>		•
<code>chdir</code>		•
<code>chmod</code>		•
<code>chown</code>	•	•
<code>creat</code>		•
<code>exec</code>		•
<code>lchown</code>	•	
<code>link</code>		•
<code>lstat</code>	•	
<code>mkdir</code>		•
<code>mkfifo</code>		•
<code>mknod</code>		•
<code>open</code>		•
<code>opendir</code>		•
<code>pathconf</code>		•
<code>readlink</code>	•	
<code>remove</code>	•	
<code>rename</code>	•	
<code>stat</code>		•
<code>truncate</code>		•
<code>unlink</code>	•	

实例

使用符号连接可能在文件系统中引入循环。大多数查找路径名的函数在这种情况下发生时都

返回值为ELOOP的errno。考虑下列命令序列：

```
$ mkdir foo                创建一个新目录
$ touch foo/a              创建0长文件
$ ln -s ../foo foo/testdir 创建一符号连接
$ ls -l foo
total 1
-rw-rw-r-- 1 stevens      0 Dec  6 06:06 a
lrwxrwxrwx 1 stevens      6 Dec  6 06:06 testdir -> ../foo
```

这创建了一个目录foo，它包含了一个名为a的文件以及一个指向foo的符号连接。在图4-4中显示了这种结果，图中以圆表示目录，以正方形表示一个文件。如果我们写一段简单的程序，使用标准函数ftw(3)以降序遍历文件结构，打印每个遇到的路径名，则其输出是：

```
foo
foo/a
foo/testdir
foo/testdir/a
foo/testdir/testdir
foo/testdir/testdir/a
foo/testdir/testdir/testdir
foo/testdir/testdir/testdir/a
(更多行)
ftw returned -1:Too many levels of symbolic links
```

4.21节提供了我们自己的ftw函数版本，它用lstat代替stat以阻止它跟随符号连接。

这样一个连接很容易被删除——因为unlink并不跟随符号连接，所以可以unlink文件foo/testdir。但是如果创建了一个构成这种循环的硬连接，那么就很难删除它[⊖]。这就是为什么link函数不允许构造指向目录的硬连接的原因。（除非进程具有超级用户优先权。）

用open打开文件时，如果传递给open函数的路径名指定了一个符号连接，那么open跟随此连接到指定的文件。若此符号连接所指向的文件并不存在，则open返回出错，表示它不能打开该文件。这可能会使不熟悉符号连接的用户感到迷惑，例如：

```
$ ln -s /no/such/file myfile      创建一符号连接
$ ls myfile
myfile                            ls查到该文件
$ cat myfile                       试图察看该文件
cat: myfile: No such file or directory
$ ls -l myfile                     试-l选择项
lrwxrwxrwx 1 stevens    13 Dec  6 07:27 myfile -> /no/such/file
```

文件myfile存在，但cat却称没有这一文件。其原因是myfile是个符号连接，由该符号连接所指

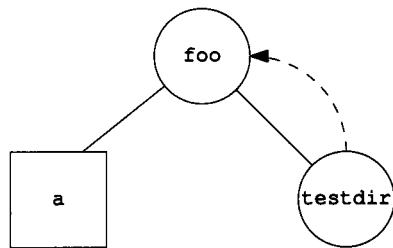


图4-4 创建一个循环的符号连接testdir

⊖ 在编写本节时，作者在自己的系统上作为一个实验做了这一点。文件系统变得错误百出，正常的fsck(1)公共程序不能解决问题。为了修复此文件系统，不得不使用了并不推荐使用的工具crlr(8)和dcheck(8)。

向的文件并不存在。ls命令的-l选择项给我们两个提示：第一个字符是l，它表示这是一个符号连接，而->表示这是一个符号连接。ls命令还有另一个选择项-F，它在是符号连接的文件名后加一个@符号，在未使用-l选择项时，这可以帮助识别出符号连接。

4.17 symlink和readlink函数

symlink函数创建一个符号连接。

```
#include <unistd.h>

int symlink(const char *actualpath, const char *sympath);
```

返回：若成功则为0，若出错则为-1

该函数创建了一个指向 *actualpath* 的新目录项 *sympath*，在创建此符号连接时，并不要求 *actualpath* 已经存在（在上一节结束部分的例子中我们已经看到了这一点）。并且，*actualpath* 和 *sympath* 并不需要位于同一文件系统中。

因为open函数跟随符号连接，所以需要有一种方法打开该连接本身，并读该连接中的名字。readlink函数提供了这种功能。

```
#include <unistd.h>

int readlink(const char *pathname, char buf, int bufsize);
```

返回：若成功则为读的字节数，若出错则为-1

此函数组合了open, read和close的所有操作。

如果此函数成功，则它返回读入 *buf* 的字节数。在 *buf* 中返回的符号连接的内容不以 null 字符终止。

4.18 文件的时间

对每个文件保持有三个时间字段，它们的意义示于表4-8中。

表4-8 与每个文件相关的三个时间值

字 段	说 明	例 子	ls(1)选择项
st_atime	文件数据的最后存取时间	read	-u
st_mtime	文件数据的最后修改时间	write	缺省
st_ctime	i节点状态的最后更改时间	chmod, chown	-c

注意修改时间(st_mtime)和更改状态时间(st_ctime)之间的区别。修改时间是文件内容最后一次被修改的时间。更改状态时间是该文件的 i 节点最后一次被修改的时间。在本章中我们已说明了很多操作，它们影响到 i 节点，但并没有更改文件的实际内容：文件的存取许可权、用户 ID、连接数等等。因为 i 节点中的所有信息都是与文件的实际内容分开存放的，所以，除了文件数据修改时间以外，还需要更改状态时间。

注意，系统并不保存对一个 i 节点的最后一次存取时间，所以 access 和 stat 函数并不更改这

三个时间中的任一个。

系统管理员常常使用存取时间来删除在一定的时间范围内没有存取过的文件。典型的例子是删除在过去一周内没有存取过的名为 a.out或core的文件。find(1)命令常被用来进行这种操作。

修改时间和更改状态时间可被用来归档其内容已经被修改或其 i节点已经被更改的那些文件。

ls命令按这三个时间值中的一个排序进行显示。按系统默认(用-l或-t选择项调用时)，它按文件的修改时间的先后排序显示。-u选择项使其用存取时间排序，-c选择项则使其用更改状态时间排序。

表4-9列出了我们已说明过的各种函数对这三个时间的作用。回忆4.14节中所述，目录是包含目录项(文件名和相关的i节点编号)的文件，增加、删除或修改目录项会影响到与其所在目录相关的三个时间。这就是在表4-9中包含两列的原因，其中一列是与该文件(或目录)相关的三个时间，另一列是与所引用的文件(或目录)的父目录相关的三个时间。例如，创建一个新文件影响到包含此新文件的目录，也影响该新文件的i节点。但是，读或写一个文件只影响该文件的i节点，而对父目录则无影响(mkdir和rmdir函数将在4.20节中说明。utime函数将在下一节中说明。6个exec函数将在4.20节中讨论。第14章将说明mkfifo和pipe函数)。

表4-9 各种函数对存取、修改和更改状态时间的作用

函 数	引用文件 (或目录)			引用文件(或目录) 的父目录			备 注
	a	m	c	a	m	c	
chmod, fchmod			•				
chown, fchown			•				
creat	•	•	•		•	•	O_CREAT新文件
creat		•	•				O_TRUNC现存文件
exec	•						
lchown			•				
link			•		•	•	
mkdir	•	•	•		•	•	
mkfifo	•	•	•		•	•	
open	•	•	•		•	•	O_CREAT新文件
open		•	•				O_TRUNC现存文件
pipe	•	•	•				
read	•						
remove			•		•	•	删除文件 = unlink
remove					•	•	删除目录 = rmdir
rename			•		•	•	对于两个参数
rmdir					•	•	
truncate, ftruncate		•	•				
unlink			•		•	•	
utime	•	•	•				
write		•	•				

4.19 utime函数

一个文件的存取和修改时间可以用 utime 函数更改。

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

int utime(const char *pathname, const struct utimbuf * times);
```

返回：若成功则为 0，若出错则为 -1

此函数所使用的结构是：

```
struct utimbuf {
    time_t actime; /*access time*/
    time_t modtime; /*modification time*/
}
```

此结构中的两个时间值是日历时间。如 1.10 节中所述，这是自 1970 年 1 月 1 日，00:00:00 以来国际标准时间所经过的秒数。

此函数的操作以及执行它所要求的优先权取决于 *times* 参数是否是 NULL。

(1) 如果 *times* 是一个空指针，则存取时间和修改时间两者都设置为当前时间。为了执行此操作必须满足下列两条件之一：(a) 进程的有效用户 ID 必须等于该文件的所有者 ID，(b) 进程对该文件必须具有写许可权。

(2) 如果 *times* 是非空指针，则存取时间和修改时间被设置为 *times* 所指向的结构中的值。此时，进程的有效用户 ID 必须等于该文件的所有者 ID，或者进程必须是一个超级用户进程。对文件只具有写许可权是不够的。

注意，我们不能对更改状态时间 *st_ctime* 指定一个值，当调用 utime 函数时，此字段被自动更新。

在某些 UNIX 版本中，touch(1) 命令使用此函数。另外，标准归档程序 tar(1) 和 cpio(1) 可选地调用 utime，以便将一个文件的时间值设置为将它归档时的值。

实例

程序 4-6 使用带 O_TRUNC 选择项的 open 函数将文件长度截短为 0，但并不更改其存取时间及修改时间。为了做到这一点，首先用 stat 函数得到这些时间，然后截短文件，最后再用 utime 函数重置这两个时间。

程序 4-6 utime 函数实例

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

int
main(int argc, char *argv[])
{
    int          i;
    struct stat  statbuf;
    struct utimbuf timebuf;
```

```

for (i = 1; i < argc; i++) {
    if (stat(argv[i], &statbuf) < 0) { /* fetch current times */
        err_ret("%s: stat error", argv[i]);
        continue;
    }
    if (open(argv[i], O_RDWR | O_TRUNC) < 0) { /* truncate */
        err_ret("%s: open error", argv[i]);
        continue;
    }
    timebuf.actime = statbuf.st_atime;
    timebuf.modtime = statbuf.st_mtime;
    if (utime(argv[i], &timebuf) < 0) { /* reset times */
        err_ret("%s: utime error", argv[i]);
        continue;
    }
}
exit(0);
}

```

以下列方式运行程序4-6：

```

$ ls -l changemod times          察看长度和最后修改时间
-rwxrwxr-x 1 stevens 24576 Dec  4 16:13 changemod
-rwxrwxr-x 1 stevens 24576 Dec  6 09:24 times
$ ls -lu changemod times        察看最后存取时间
-rwxrwxr-x 1 stevens 24576 Feb  1 12:44 changemod
-rwxrwxr-x 1 stevens 24576 Feb  1 12:44 times
$ date                          打印当天日期
Sun Feb  3 18:22:33 MST 1991
$ a.out changemod times         运行程序4-6
$ ls -l changemod times        检查结果
-rwxrwxr-x 1 stevens  0 Dec  4 16:13 changemod
-rwxrwxr-x 1 stevens  0 Dec  6 09:24 times
$ ls -lu changemod times       检查最后存取时间
-rwxrwxr-x 1 stevens  0 Feb  1 12:44 changemod
-rwxrwxr-x 1 stevens  0 Feb  1 12:44 times
$ ls -lc changemod times       更改状态时间
-rwxrwxr-x 1 stevens  0 Feb  3 18:23 changemod
-rwxrwxr-x 1 stevens  0 Feb  3 18:23 times

```

正如我们所预见的一样，最后修改时间和最后存取时间未变。但是，更改状态时间则更改为程序运行时的时间（这两个文件的最后存取时间相同的原因是，这是它们的目录用 tar命令归档时的时间）。

4.20 mkdir和rmdir函数

用mkdir函数创建目录，用rmdir函数删除目录。

```

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

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

```

返回：若成功则为0，若出错则为-1

此函数创建一个新的空目录。.*和*.. 目录项是自动创建的。所指定的文件存取许可权 *mode* 由进程的文件方式创建屏蔽字修改。

常见的错误是指定与文件相同的 *mode* (只指定读、写许可权)。但是, 对于目录通常至少要设置1个执行许可权位, 以允许存取该目录中的文件名 (见习题 4.18)。

按照4.6节中讨论的规则, 设置新目录的用户ID和组ID。

SVR4也使新目录继承父目录的设置-组-ID位。这就使得在新目录中创建的文件将继承该目录的组ID。

4.3+BSD并不要求继承此设置-组-ID位, 因为不论设置-组-ID位如何, 新创建的文件和目录总是继承父目录的组ID。

早期的UNIX版本并没有 `mkdir` 函数, 它是由4.2BSD和SVR3引进的。在早期版本中, 进程要调用 `mknod` 函数以创建一个新目录。但是只有超级用户进程才能使用 `mknod` 函数。为了避免这一点, 创建目录的命令 `mkdir(1)` 必须由根拥有, 而且打开了其设置-用户-ID位。进程为了创建一个目录, 必须用 `system(3)` 函数调用 `mkdir` 命令(1)。

用 `rmdir` 函数可以删除一个空目录。

```
#include <unistd.h>

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

返回: 若成功则为 0, 若出错则为 -1

如果此调用使目录的连接计数成为 0, 并且也没有其他进程打开此目录, 则释放由此目录占用的空间。如果在连接计数达到 0 时, 有一个或几个进程打开了此目录, 则在此函数返回前删除最后一个连接及.*和*.. 项。另外, 在此目录中不能再创建新文件。但是在最后一个进程关闭它之前并不释放此目录 (即使某些进程打开该目录, 它们在此目录下, 也不能执行其他操作, 因为为使 `rmdir` 函数成功执行, 该目录必须是空的)。

4.21 读目录

对某个目录具有存取许可权的任一用户都可读该目录, 但是只有内核才能写目录 (防止文件系统发生混乱)。回忆4.5节, 一个目录的写许可权位和执行许可权位决定了在该目录中能否创建新文件以及删除文件, 它们并不表示能否写目录本身。

目录的实际格式依赖于UNIX的具体实现。早期的系统, 例如V7, 有一个比较简单的结构: 每个目录项是16个字节, 其中14个字节是文件名, 2个字节是*i*节点编号数。而对于4.2BSD而言, 由于它允许相当长的文件名, 所以每个目录项的长度是可变的。这就意味着读目录的程序与系统相关。为了简化这种情况, UNIX现在包含了一套与读目录有关的例程, 它们是 POSIX.1 的一部分。

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

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


返回：若成功则为指针，若出错则为 NULL

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

返回：若成功则为指针，若在目录尾或出错则为 NULL

```
void rewinddir(DIR *dp);
```

```
int closedir(DIR *dp);
```

返回：若成功则为 0，若出错则为 -1

回忆一下，在程序 1-1 中（ls 命令的基本实现部分）使用了这些函数。

定义在头文件 <dirent.h> 中的 dirent 结构与实现有关。SVR4 和 4.3+BSD 定义此结构至少包含下列两个成员：

```
struct dirent {
    ino_t d_ino;           /*i-node number*/
    char  d_name[NAME_MAX+1]; /*null-terminated filename*/
}
```

POSIX.1 并没有定义 d_ino，因为这是一个实现特征。POSIX.1 在此结构中只定义 d_name 项。

注意，SVR4 没有将 NAME_MAX 定义为一个常数——其值依赖于该目录所在的文件系统，并且通常可用 fpathconf 函数取得。在 BSD 类文件系统中，NAME_MAX 的常用值是 255（见表 2-7）。但是，因为文件名是以 null 字符结束的，所以在头文件中如何定义数组 d_name 并无多大关系。

DIR 结构是一个内部结构，它由这四个函数用来保存正被读的目录的有关信息。其作用类似于 FILE 结构。FILE 结构由标准 I/O 库维护（我们将在第 5 章中对它进行说明）。

由 opendir 返回的指向 DIR 结构的指针由另外三个函数使用。opendir 执行初始化操作，使第一个 readdir 读目录中的第一个目录项。目录中各目录项的顺序与实现有关。它们通常并不按字母顺序排列。

实例

我们将使用这些目录例程编写一个遍历文件层次结构的程序，其目的是得到如表 4-2 中所示的各种类型的文件数。程序 4-7 只有一个参数，它说明起点路径名，从该点开始递归降序遍历文件层次结构。系统 V 提供了一个实际遍历此层次结构的函数 ftw(3)，对于每一个文件它都调用一个用户定义函数。此函数的问题是：对于每一个文件，它都调用 stat 函数，这就使程序跟随符号连接。例如，如果从 root 开始，并且有一个名为 /lib 的符号连接，它指向 /usr/lib，则所有在目录 /usr/lib 中的文件都两次计数。为了纠正这一点，SVR4 提供了另一个函数 nftw(3)，它具有一个停止跟随符号连接的选择项。尽管可以使用 nftw，但是为了说明目录例程的使用方法，我们还是编写了一个简单的文件遍历程序。

程序 4-7 递归降序遍历目录层次结构，并按文件类型计数

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

```

#include <limits.h>
#include "ourhdr.h"
typedef int Myfunc(const char *, const struct stat *, int);
        /* function type that's called for each filename */

static Myfunc myfunc;
static int myftw(char *, Myfunc *);
static int dopath(Myfunc *);

static long nreg, ndir, nblk, nchr, nfifo, nslink, nsock, ntot;

int
main(int argc, char *argv[])
{
    int ret;

    if (argc != 2)
        err_quit("usage: ftw <starting-pathname>");

    ret = myftw(argv[1], myfunc);        /* does it all */

    if ( (ntot = nreg + ndir + nblk + nchr + nfifo + nslink + nsock) == 0)
        ntot = 1;        /* avoid divide by 0; print 0 for all counts */
    printf("regular files = %7ld, %5.2f %%\n", nreg, nreg*100.0/ntot);
    printf("directories = %7ld, %5.2f %%\n", ndir, ndir*100.0/ntot);
    printf("block special = %7ld, %5.2f %%\n", nblk, nblk*100.0/ntot);
    printf("char special = %7ld, %5.2f %%\n", nchr, nchr*100.0/ntot);
    printf("FIFOs = %7ld, %5.2f %%\n", nfifo, nfifo*100.0/ntot);
    printf("symbolic links = %7ld, %5.2f %%\n", nslink, nslink*100.0/ntot);
    printf("sockets = %7ld, %5.2f %%\n", nsock, nsock*100.0/ntot);

    exit(ret);
}

/*
 * Descend through the hierarchy, starting at "pathname".
 * The caller's func() is called for every file.
 */

#define FTW_F 1        /* file other than directory */
#define FTW_D 2        /* directory */
#define FTW_DNR 3        /* directory that can't be read */
#define FTW_NS 4        /* file that we can't stat */

static char *fullpath;        /* contains full pathname for every file */

static int myftw(char *pathname, Myfunc *func)
/* we return whatever func() returns */
{
    fullpath = path_alloc(NULL);        /* malloc's for PATH_MAX+1 bytes */
        /* (Program 2.2) */
    strcpy(fullpath, pathname);        /* initialize fullpath */

    return(dopath(func));
}

/*
 * Descend through the hierarchy, starting at "fullpath".
 * If "fullpath" is anything other than a directory, we lstat() it,
 * call func(), and return. For a directory, we call ourself
 * recursively for each name in the directory.
 */

static int dopath(Myfunc *func)
/* we return whatever func() returns */
{
    struct stat statbuf;
    struct dirent *dirp;

```

```

DIR          *dp;
int          ret;
char        *ptr;

if (lstat(fullpath, &statbuf) < 0)
    return(func(fullpath, &statbuf, FTW_NS)); /* stat error */

if (S_ISDIR(statbuf.st_mode) == 0)
    return(func(fullpath, &statbuf, FTW_F)); /* not a directory */

/*
 * It's a directory. First call func() for the directory,
 * then process each filename in the directory.
 */

if ( (ret = func(fullpath, &statbuf, FTW_D)) != 0)
    return(ret);

ptr = fullpath + strlen(fullpath); /* point to end of fullpath */
*ptr++ = '/';
*ptr = 0;

if ( (dp = opendir(fullpath)) == NULL)
    return(func(fullpath, &statbuf, FTW_DNR));
    /* can't read directory */

while ( (dirp = readdir(dp)) != NULL) {
    if (strcmp(dirp->d_name, ".") == 0 ||
        strcmp(dirp->d_name, "..") == 0)
        continue; /* ignore dot and dot-dot */

    strcpy(ptr, dirp->d_name); /* append name after slash */

    if ( (ret = dopath(func)) != 0) /* recursive */
        break; /* time to leave */
}
ptr[-1] = 0; /* erase everything from slash onwards */

if (closedir(dp) < 0)
    err_ret("can't close directory %s", fullpath);

return(ret);
}

static int
myfunc(const char *pathname, const struct stat *statptr, int type)
{
    switch (type) {
    case FTW_F:
        switch (statptr->st_mode & S_IFMT) {
        case S_IFREG: nreg++; break;
        case S_IFBLK: nblk++; break;
        case S_IFCHR: nchr++; break;
        case S_IFIFO: nfifo++; break;
        case S_IFLNK: nlink++; break;
        case S_IFSOCK: nsock++; break;
        case S_IFDIR:
            err_dump("for S_IFDIR for %s", pathname);
            /* directories should have type = FTW_D */
        }
        break;

    case FTW_D:
        ndir++;
        break;

    case FTW_DNR:
        err_ret("can't read directory %s", pathname);

```

```

        break;

    case FTW_NS:
        err_ret("stat error for %s", pathname);
        break;

    default:
        err_dump("unknown type %d for pathname %s", type, pathname);
    }

    return(0);
}

```

在程序中，我们提供了比所要求的更多的通用性，这样做的目的是为了例示实际 `ftw` 函数的应用情况。例如，函数 `myfunc` 总是返回 0，但是调用它的函数却准备处理非 0 返回。

关于降序遍历文件系统的更多信息，以及在很多标准 UNIX 命令（如 `find`, `ls`, `tar` 等）中使用这种技术的情况，请参阅 Fowler, Korn 及 Vo [1989]。4.3+BSD 提供了一新套的目录遍历函数——请参阅 `fts(3)` 手册页。

4.22 chdir, fchdir 和 getcwd 函数

每个进程都有一个当前工作目录，此目录是搜索所有相对路径名的起点（不以斜线开始的路径名为相对路径名）。当用户登录到 UNIX 系统时，其当前工作目录通常是口令文件（`/etc/passwd`）中该用户登录项的第 6 个字段——用户的起始目录。当前工作目录是进程的一个属性，起始目录则是登录名的一个属性。进程调用 `chdir` 或 `fchdir` 函数可以更改当前工作目录。

```

#include <unistd.h>

int chdir(const char *pathname);

int fchdir(int fildes);

```

两个函数的返回：若成功则为 0，若出错则为 -1

在这两个函数中，可以分别用 `pathname` 或打开文件描述符来指定新的当前工作目录。

`fchdir` 不是 POSIX.1 的所属部分，SVR4 和 4.3+BSD 则支持此函数。

实例

因为当前工作目录是一个进程的属性，所以它只影响调用 `chdir` 的进程本身，而不影响其他进程（我们将在第 8 章较详细地说明进程之间的关系）。这就意味着程序 4-8 并不会产生我们希望得到的后果。如果编译程序 4-8，并且调用其可执行目标代码文件，则可以得到下列结果：

```

$ pwd
/usr/lib
$ mycd
chdir to /tmp succeeded
$ pwd
/usr/lib

```

从中可以看出，执行 `mycd` 程序的 shell 的当前工作目录并没有改变。由此可见，shell 应当直接调用 `chdir` 函数，所以 `cd` 命令的执行程序直接包含在 shell 程序中。

因为内核保持有当前工作目录的信息，所以我们应能取其当前值。不幸的是，内核为每个进程只保存其当前工作目录的*i*节点编号以及设备标识，并不保存该目录的完整路径名。

程序4-8 chdir函数实例

```
#include    "ourhdr.h"

int
main(void)
{
    if (chdir("/tmp") < 0)
        err_sys("chdir failed");

    printf("chdir to /tmp succeeded\n");
    exit(0);
}
```

我们需要一个函数，它从当前工作目录开始，找到其上一级的目录，然后读其目录项，直到该目录项中的*i*节点编号数与工作目录*i*节点编号数相同，这样地就找到了其对应的文件。按照这种方法，逐层上移，直到遇到根，这样就得到了当前工作目录的绝对路径名。很幸运，函数getcwd就是提供这种功能的。

```
#include<unistd.h>

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

返回：若成功则为buf，若出错则为NULL

向此函数传递两个参数，一个是缓存地址buf，另一个是缓存的长度size。该缓存必须有足够的长度以容纳绝对路径名再加上一个null终止字符，否则返回出错（请回忆2.5.7节中有关为最大长度路径名分配空间的讨论）。

某些getcwd的实现允许第一个参数buf为NULL。在这种情况下，此函数调用malloc动态地分配size字节数的空间。这不是POSIX.1或XPG3的所属部分，应予避免。

实例

程序4-9将工作目录更改至一个特定的目录，然后调用getcwd,最后打印该工作目录。如果运行该程序，则可得：

```
$ a.out
cwd = /var/spool/uucppublic
$ ls -l/usr/spool
lrwxrwxrwx  1 root  12 Jan 31 07:57 /usr/spool -> ../var/spool
```

程序4-9 getcwd函数实例

```
#include    "ourhdr.h"

int
main(void)
{
```

```

char    *ptr;
int     size;

if (chdir("/usr/spool/uucppublic") < 0)
    err_sys("chdir failed");

ptr = path_alloc(&size);    /* our own function */
if (getcwd(ptr, size) == NULL)
    err_sys("getcwd failed");

printf("cwd = %s\n", ptr);
exit(0);
}

```

注意，chdir跟随符号连接（正如在表4-7所示），但是当getcwd沿目录树上溯遇到/var/spool目录时，它并不了解该目录由符号连接/usr/spool所指向。这是符号连接的一种特性。

4.23 特殊设备文件

st_dev和st_rdev这两个字段经常引起混淆，当在11.9节讨论ttyname函数时，需要使用这两个字段。有关规则很简单：

- 每个文件系统都由其主、次设备号而为人所知。设备号所用的数据类型是基本系统数据类型dev_t。回忆图4-1，一个磁盘经常包含若干个文件系统。
- 我们通常可以使用两个大多数实现都定义的宏：major和minor来存取主、次设备号。这就意味着我们无需关心这两个数是如何存放在dev_t对象中的。

早期的系统用16位整型存放设备号：8位用于主设备号，8位用于次设备号。SVR4使用32位：14位用于主设备号，18位用于次设备号。4.3+BSD则使用16位：8位用于主设备号，8位用于次设备号。

POSIX.1说明dev_t类型是存在的，但没有定义它包含什么，或如何取得其内容。大多数实现定义了宏major和minor，但在哪一个头文件中定义它们则与实现有关。

- 系统中每个文件名的st_dev值是文件系统的设备号，该文件系统包含了该文件名和其对应的i节点。
- 只有字符特殊文件和块特殊文件才有st_rdev值。此值包含该实际设备的设备号。

实例

程序4-10为每个命令行参数打印设备号，另外，若此参数引用的是字符特殊文件或块特殊文件，则也打印该特殊文件的st_rdev值。

程序4-10 打印st_dev和st_rdev值

```

#include    <sys/types.h>    /* BSD: defines major() and minor() */
#include    <sys/stat.h>
#include    "ourhdr.h"

int
main(int argc, char *argv[])
{
    int        i;
    struct stat buf;

```

```

for (i = 1; i < argc; i++) {
    printf("%s: ", argv[i]);
    if (lstat(argv[i], &buf) < 0) {
        err_ret("lstat error");
        continue;
    }

    printf("dev = %d/%d", major(buf.st_dev), minor(buf.st_dev));

    if (S_ISCHR(buf.st_mode) || S_ISBLK(buf.st_mode)) {
        printf(" (%s) rdev = %d/%d",
            (S_ISCHR(buf.st_mode)) ? "character" : "block",
            major(buf.st_rdev), minor(buf.st_rdev));
    }
    printf("\n");
}
exit(0);
}

```

在SVR4中，为了定义宏major和minor，一定要包括头文件<sys/sysmacros.h>。运行此程序得到下面的结果：

```

$ a.out //home/stevens /dev/tty[ab]
/: dev = 7/0
/home/stevens: dev = 7/7
/dev/ttya: dev = 7/0 (character) rdev = 12/0
/dev/ttyb: dev = 7/0 (character) rdev = 12/1
$ mount 察看安装情况
/dev/sd0a on /
/dev/sd0h on /home
$ ls -l /dev/sd0[ah] /dev/tty[ab]
brw-r----- 1 root      7,    0 Jan 31 08:23 /dev/sd0a
brw-r----- 1 root      7,    7 Jan 31 08:23 /dev/sd0h
crw-rw-rw-  1 root     12,    0 Jan 31 08:22 /dev/ttya
crw-rw-rw-  1 root     12,    1 Jul   9 10:11 /dev/ttyb

```

传递给该程序的头两个参数是目录（根和/home/stevens），后两个是设备名/dev/tty[ab]，这两个设备是字符特殊设备。从程序的输出可见，根目录和/home/stevens目录的设备号不同，这表示它们位于不同的文件系统中。运行mount(1)命令证明了这一点。然后用ls命令察看由mount命令报告的两个磁盘设备和两个终端设备。这两个磁盘设备是块特殊设备，而两个终端设备则是字符特殊设备。（通常，只有块特殊设备才能包含随机存取文件系统，它们是：硬、软盘驱动器和CD-ROM等。UNIX的早期版本支持磁带存放文件系统，但这从未广泛使用过。）注意，两个终端设备（st_dev）的文件名和i节点在设备7/0上（根文件系统，它包含了/dev文件系统），但是它们的实际设备号是：12/0和12/1。

4.24 sync和fsync函数

传统的UNIX实现在内核中设有缓冲存储器，大多数磁盘I/O都通过缓存进行。当将数据写到文件上时，通常该数据先由内核复制到缓存中，如果该缓存尚未写满，则并不将其排入输出队列，而是等待其写满或者当内核需要重用该缓存以便存放其他磁盘块数据时，再将该缓存排入输出队列，然后待其到达队首时，才进行实际的I/O操作。这种输出方式被称之为延迟写（delayed write）（Bach [1986]第3章详细讨论了延迟写）。延迟写减少了磁盘读写次数，但是

却降低了文件内容的更新速度，使得欲写到文件中的数据在一段时间内并没有写到磁盘上。当系统发生故障时，这种延迟可能造成文件更新内容的丢失。为了保证磁盘上实际文件系统与缓存中内容的一致性，UNIX系统提供了sync和fsync两个系统调用函数。

```
#include <unistd.h>

void sync(void);

int fsync(int fildes);
```

返回：若成功则为0，若出错则为-1

sync只是将所有修改过的块的缓存排入写队列，然后就返回，它并不等待实际I/O操作结束。

系统精灵进程(通常称为update)一般每隔30秒调用一次sync函数。这就保证了定期刷新内核的块缓存。命令sync(1)也调用sync函数。

函数fsync只引用单个文件(由文件描述符fildes指定)，它等待I/O结束，然后返回。fsync可用于数据库这样的应用程序，它确保修改过的块立即写到磁盘上。比较一下fsync和O_SYNC标志(见3.13节)。当调用fsync时，它更新文件的内容，而对于O_SYNC，则每次对文件调用write函数时就更新文件的内容。

SVR4和4.3+BSD两者都支持sync和fsync,它们都不是POSIX.1的组成部分，但XPG3要求fsync。

4.25 文件存取许可权位小结

我们已经说明了所有文件存取许可权位，其中某些位有多种用途。表4-10列出了所有这些许可权位，以及它们对普通文件和目录文件的作用。

表4-10 文件存取许可权位小结

常数	说明	对普通文件的影响	对目录的影响
S_ISUID	设置-用户-ID	执行时设置有效用户ID	(不使用)
S_ISGID	设置-组-ID	若组执行位设置，则执行时设置有效组ID，否则使强制性锁起作用	将在目录中创建的新文件的组ID 设置为目录的组ID
S_ISVTX	粘住位	在交换区保存程序正文(若支持)	禁止在目录中删除和更名文件
S_ISUSR	用户读	许可用户读文件	许可用户读目录项
S_IWUSR	用户写	许可用户写文件	许可用户在目录中删除或创建文件
S_IXUSR	用户执行	许可用户执行文件	许可用户在目录中搜索给定路径名
S_IRGRP	组读	许可组读文件	许可组读目录项
S_IWGRP	组写	许可组写文件	许可组在目录中删除或创建文件
S_IXGRP	组执行	许可组执行文件	许可组在目录中搜索给定路径名
S_IROTH	其他读	许可其他读文件	许可其他读目录项
S_IWOTH	其他写	许可其他写文件	许可其他在目录中删除或创建文件
S_IXOTH	其他执行	许可其他执行文件	许可其他在目录中搜索给定路径名

最后9个常数分成3组。


```
S_IRWXU = S_IRUSR|S_IWUSR | S_IXUSR
S_IRWXG = S_IRGRP|S_IWGRP | S_IXGRP
S_IRWXO = S_IROTH|S_IWOTH | S_IXOTH
```

4.26 小结

本章内容围绕stat函数，详细介绍了stat结构中的每一个成员。这使我们对UNIX文件的各个属性都有所了解。对文件的所有属性以及对文件进行操作的所有函数有完整的了解对各种UNIX程序设计都非常重要。

习题

4.1 用stat函数替换程序4-1中的lstat函数，对于命令行的参数是符号连接的情况有什么变化。

4.2 表4-1指出SVR4没有提供宏S_ISLNK，但是SVR4支持符号连接并且在<sys/stat.h>中定义了S_IFLNK，如何修改ourhdr.h使得需要S_ISLNK宏的程序可以使用它？

4.3 如果文件方式创建屏蔽字是777（八进制），结果会怎样？利用umask命令验证结论。

4.4 验证关闭一个你所拥有的文件的用户读许可权，也就是不允许你访问自己的文件。

4.5 创建文件foo和bar后运行程序4-3会产生什么情况？

4.6 4.12节中讲到一个常规文件的大小可以为0，同时我们又知道st_size字段定义为目录或符号连接，那么目录和符号连接的长度是否可以不为0？

4.7 编写一个类似cp(1)的程序，它复制包含空洞的文件，但不将字节0写到输出文件中去。

4.8 4.12节ls命令的输出中，core和core.copy的存取许可权不相同，如果创建两个文件时umask没有变，说明为什么会发生这种差别。

4.9 程序4-5使用了df(1)命令来检查空闲的磁盘空间。为什么不能使用du(1)命令？

4.10 表4-9中给出unlink函数会修改文件状态改变时间，这是怎样实现的？

4.11 4.21节中，系统对可打开文件数的限制对myftw函数会产生什么影响？

4.12 4.21节中的ftw从不改变其目录，对这种处理方法进行改动：每次遇到一个目录就对之进行chdir，这样每次调用lstat时就可以使用文件名而非路径名，处理完所有的目录后执行chdir("..")。比较这种方法和正文中方法的运行时间。

4.13 每个进程都有一个根目录用于处理绝对路径名，可以通过chroot函数改变根目录。在手册中查阅此函数说明这个函数什么时候有用。

4.14 如何利用utime函数只设置两个时间值中的一个？

4.15 有些版本的finger(1)命令输出“New mail received ...”和“unread since ...”，其中...是相应的日期和时间。程序是如何决定这些日期和时间的？

4.16 用cpio(1)和tar(1)命令检查档案文件时改变了每个被归档文件的哪些时间值？复原后的文件的存取时间是多少？为什么？

4.17 file(1)命令通过读文件的第一部分并检查其内容，利用一些启发式规则确定文件的类型，如C程序、Fortran程序、shell脚本等等。UNIX提供了一条命令，它允许我们执行另一条命令，并可以跟踪该命令执行的所有系统调用。在SVR4中该命令为truss(1)，在4.3+BSD中为ktrace(1)和kdump(1)，下列使用SunOS的trace(1)跟踪file命令中的系统调用：

```
trace file a.out
```

下面是file调用的函数：

```
lstat ("a.out", 0xf7fff650) = 0
open ("a.out", 0, 0) = 3
read (3, "...", 512) = 512
fstat (3, 0xf7fff160) = 0
write (1, "a.out: demand paged execu" .., 44) = 44
a.out: demand paged executable not stripped
utime ("a.out", 0xf7fff1b0) = 0
```

为什么file命令调用utime？

4.18 UNIX对目录的深度有限制吗？编一个程序循环创建目录并修改目录名，确定叶子节点的绝对路径名的长度大于系统的 PATH_MAX限制。可以调用 getcwd得到目录的路径名吗？标准UNIX工具是如何处理长路径名的？对目录可以使用 tar或cpio命令吗？

4.19 3.15节中描述了/dev/fd的特征，如果每个用户都可以访问这些文件，则其许可权必须为rw-rw-rw-。有些程序创建输出文件时，先进行删除以确保该文件名不存在。

```
unlink (path);
if ( (fd = creat(path, FILE_MODE)) < 0)
    err_sys(...);
```

讨论一下path是/dev/fd/1的情况。

第5章 标准 I/O 库

5.1 引言

本章说明标准 I/O 库。因为不仅在 UNIX 而且在很多操作系统上都实现此库，所以它由 ANSI C 标准说明。标准 I/O 库处理很多细节，例如缓存分配，以优化长度执行 I/O 等。这样使用户不必担心如何选择使用正确的块长度（如 3.9 节中所述）。标准 I/O 库是在系统调用函数基础上构造的，它便于用户使用，但是如果不太深入地了解库的操作，也会带来一些问题。

标准 I/O 库是由 Dennis Ritchie 在 1975 年左右编写的。它是由 Mike Lesk 编写的可移植 I/O 库的主要修改版本。令人惊异的是，15 年后制订的标准 I/O 库对它只作了极小的修改。

5.2 流和 FILE 对象

在第 3 章中，所有 I/O 函数都是针对文件描述符的。当打开一个文件时，即返回一个文件描述符，然后该文件描述符就用于后读的 I/O 操作。而对于标准 I/O 库，它们的操作则是围绕流（stream）进行的（请勿将标准 I/O 术语流与系统 V 的 STREAMS I/O 系统相混淆）。当用标准 I/O 库打开或创建一个文件时，我们已使一个流与一个文件相结合。

当打开一个流时，标准 I/O 函数 `fopen` 返回一个指向 FILE 对象的指针。该对象通常是一个结构，它包含了 I/O 库为管理该流所需要的所有信息：用于实际 I/O 的文件描述符，指向流缓存的指针，缓存的长度，当前在缓存中的字符数，出错标志等等。

应用程序没有必要检验 FILE 对象。为了引用一个流，需将 FILE 指针作为参数传递给每个标准 I/O 函数。在本书中，我们称指向 FILE 对象的指针（类型为 `FILE*`）为文件指针。

在本章中，我们以 UNIX 系统为例，说明标准 I/O 库。正如前述，此标准库已移到除 UNIX 以外的很多系统中。但是为了说明该库实现的一些细节，我们选择 UNIX 实现作为典型进行介绍。

5.3 标准输入、标准输出和标准出错

对一个进程预定义了三个流，它们自动地可为进程使用：标准输入、标准输出和标准出错。在 3.2 节中我们曾用文件描述符 `STDIN_FILENO`、`STDOUT_FILENO` 和 `STDERR_FILENO` 分别表示它们。

这三个标准 I/O 流通过预定义文件指针 `stdin`、`stdout` 和 `stderr` 加以引用。这三个文件指针同样定义在头文件 `<stdio.h>` 中。

5.4 缓存

标准 I/O 提供缓存的目的是尽可能减少使用 `read` 和 `write` 调用的数量（见表 3-1，其中显示了在不同缓存长度情况下，为执行 I/O 所需的 CPU 时间量）。它也对每个 I/O 流自动地进行缓存管

理，避免了应用程序需要考虑这一点所带来的麻烦。不幸的是，标准 I/O 库令人最感迷惑的也是它的缓存。

标准 I/O 提供了三种类型的缓存：

(1) 全缓存。在这种情况下，当填满标准 I/O 缓存后才进行实际 I/O 操作。对于驻在磁盘上的文件通常是由标准 I/O 库实施全缓存的。在一个流上执行第一次 I/O 操作时，相关标准 I/O 函数通常调用 `malloc`（见 7.8 节）获得需使用的缓存。

术语刷新（`flush`）说明标准 I/O 缓存的写操作。缓存可由标准 I/O 例程自动地刷新（例如当填满一个缓存时），或者可以调用函数 `fflush` 刷新一个流。值得引起注意的是在 UNIX 环境中，刷新有两种意思。在标准 I/O 库方面，刷新意味着将缓存中的内容写到磁盘上（该缓存可以只是局部填写的）。在终端驱动程序方面（例如在第 11 章中所述的 `tcflush` 函数），刷新表示丢弃已存在缓存中的数据。

(2) 行缓存。在这种情况下，当在输入和输出中遇到新行符时，标准 I/O 库执行 I/O 操作。这允许我们一次输出一个字符（用标准 I/O `fputc` 函数），但只有在写了一行之后才进行实际 I/O 操作。当流涉及一个终端时（例如标准输入和标准输出），典型地使用行缓存。

对于行缓存有两个限制。第一个是：因为标准 I/O 库用来收集每一行的缓存的长度是固定的，所以只要填满了缓存，那么即使还没有写一个新行符，也进行 I/O 操作。第二个是：任何时候只要通过标准输入输出库要求从 (a) 一个不带缓存的流，或者 (b) 一个行缓存的流（它预先要求从内核得到数据）得到输入数据，那么就会造成刷新所有行缓存输出流。在 (b) 中带了一个在括号中的说明的理由是，所需的数据可能已在该缓存中，它并不要求内核在需要该数据时才进行该操作。很明显，从不带缓存的一个流中进行输入（(a) 项）要求当时从内核得到数据。

(3) 不带缓存。标准 I/O 库不对字符进行缓存。如果用标准 I/O 函数写若干字符到不带缓存的流中，则相当于用 `write` 系统调用函数将这些字符写至相关联的打开文件上。标准出错流 `stderr` 通常是不带缓存的，这就使得出错信息可以尽快显示出来，而不管它们是否含有一个新行字符。

ANSI C 要求下列缓存特征：

- (1) 当且仅当标准输入和标准输出并不涉及交互作用设备时，它们才是全缓存的。
- (2) 标准出错决不会是全缓存的。

但是，这并没有告诉我们如果标准输入和输出涉及交互作用设备时，它们是不带缓存的还是行缓存的，以及标准输出是不带缓存的，还是行缓存的。SVR4 和 4.3+BSD 的系统默认使用下列类型的缓存：

- 标准出错是不带缓存的。
- 如若是涉及终端设备的其他流，则它们是行缓存的；否则是全缓存的。

对任何一个给定的流，如果我们并不喜欢这些系统默认，则可调用下列两个函数中的一个更改缓存类型：

```
#include <stdio.h>

void setbuf(FILE *fp, char buf);

int setvbuf(FILE *fp, char buf, int mode, size_t size);
```

返回：若成功则为 0，若出错则非 0

这些函数一定要在流已被打开后调用（这是十分明显的，因为每个函数都要求一个有效的文件

指针作为它们的第一个参数)，而且也应在对该流执行任何一个其他操作之前调用。

可以使用 `setbuf` 函数打开或关闭缓存机制。为了带缓存进行 I/O，参数 `buf` 必须指向一个长度为 `BUFSIZ` 的缓存（该常数定义在 `<stdio.h>` 中）。通常在此之后该流就是全缓存的，但是如果该流与一个终端设备相关，那么某些系统也可将其设置为行缓存的。为了关闭缓存，将 `buf` 设置为 `NULL`。

使用 `setvbuf`，我们可以精确地说明所需的缓存类型。这是依靠 `mode` 参数实现的：

```
_IOFBF 全缓存
_IOLBF 行缓存
_IONBF 不带缓存
```

如果指定一个不带缓存的流，则忽略 `buf` 和 `size` 参数。如果指定全缓存或行缓存，则 `buf` 和 `size` 可以可选择地指定一个缓存及其长度。如果该流是带缓存的，而 `buf` 是 `NULL`，则标准 I/O 库将自动地为该流分配适当长度的缓存。适当长度指的是由 `struct` 结构中的成员 `st_blksize` 所指定的值（见 4.2 节）。如果系统不能为该流决定此值（例如若此流涉及一个设备或一个管道），则分配长度为 `BUFSIZ` 的缓存。

伯克利系统首先使用 `st_blksize` 表示缓存长度。较早的系统 V 版本使用标准 I/O 常数 `BUFSIZ`（其典型值是 1024）。即使 4.3+BSD 使用 `st_blksize` 决定最佳的 I/O 缓存长度，它仍将 `BUFSIZ` 设置为 1024。

表 5-1 列出了这两个函数的动作，以及它们的各个选择项。

表 5-1 `setbuf` 和 `setvbuf` 函数

函 数	<i>mode</i>	<i>buf</i>	缓存及长度	缓存的类型
<code>setbuf</code>		<code>nonnull</code>	长度为 <code>BUFSIZ</code> 的用户缓存	全缓存或行缓存
		<code>NULL</code>	（无缓存）	不带缓存
<code>setvbuf</code>	<code>_IOFBF</code>	<code>nonnull</code>	长度为 <i>size</i> 的用户缓存	全缓存
		<code>NULL</code>	合适长度的系统缓存	
	<code>_IOLBF</code>	<code>nonnull</code>	长度为 <i>size</i> 的用户缓存	行缓存
		<code>NULL</code>	合适长度的系统缓存	
	<code>_IONBF</code>	忽略	无缓存	不带缓存

要了解，如果在一个函数中分配一个自动变量类的标准 I/O 缓存，则从该函数返回之前，必须关闭该流。（7.8 节将对此作更多讨论。）另外，SVR4 将缓存的一部分用于它自己的管理操作，所以可以存放在缓存中的实际数据字节数少于 *size*。一般而言，应由系统选择缓存的长度，并自动分配缓存。在这样处理时，标准 I/O 库在关闭此流时将自动释放此缓存。

任何时候，我们都可强制刷新一个流。

```
#include<stdio.h>

int fflush(FILE *fp);
```

返回：若成功则为 0，若出错则为 EOF

此函数使该流所有未写的数据都被传递至内核。作为一种特殊情形，如若 `fp` 是 `NULL`，则此函数刷新所有输出流。

传送一个空指针以强迫刷新所有输出流，这是由 ANSI C 新引入的。非 ANSI C 库（例如较早的系统 V 版本和 4.3BSD）并不支持此种特征。

5.5 打开流

下列三个函数可用于打开一个标准 I/O 流。

```
#include <stdio.h>

FILE *fopen(const char *pathname, const char * type);

FILE *freopen(const char *pathname, const char * type, FILE *fp);

FILE *fdopen(int fildes, const char * type);
```

三个函数的返回：若成功则为文件指针，若出错则为 `NULL`

这三个函数的区别是：

- (1) `fopen` 打开路径名由 `pathname` 指示的一个文件。
- (2) `freopen` 在一个特定的流上（由 `fp` 指示）打开一个指定的文件（其路径名由 `pathname` 指示），如若该流已经打开，则先关闭该流。此函数一般用于将一个指定的文件打开为一个预定义的流：标准输入、标准输出或标准出错。
- (3) `fdopen` 取一个现存的文件描述符（我们可能从 `open`, `dup`, `dup2`, `fcntl` 或 `pipe` 函数得到此文件描述符），并使一个标准的 I/O 流与该描述符相结合。此函数常用于由创建管道和网络通信通道函数获得的描述符。因为这些特殊类型的文件不能用标准 I/O `fopen` 函数打开，首先必须先调用设备专用函数以获得一个文件描述符，然后用 `fdopen` 使一个标准 I/O 流与该描述符相结合。

`fopen` 和 `freopen` 是 ANSI C 的所属部分。而 ANSI C 并不涉及文件描述符，所以仅有 POSIX.1 具有 `fdopen`。

`type` 参数指定对该 I/O 流的读、写方式，ANSI C 规定 `type` 参数可以有 15 种不同的值，它们示于表 5-2 中。

表 5-2 打开标准 I/O 流的 `type` 参数

<i>type</i>	说 明
<code>r</code> 或 <code>rb</code>	为读而打开
<code>w</code> 或 <code>wb</code>	使文件成为 0 长，或为写而创建
<code>a</code> 或 <code>ab</code>	添加；为在文件尾写而打开，或为写而创建
<code>r+</code> 或 <code>r+b</code> 或 <code>rb+</code>	为读和写而打开
<code>w+</code> 或 <code>w+b</code> 或 <code>wb+</code>	使文件为 0 长，或为读和写而打开
<code>a+</code> 或 <code>a+b</code> 或 <code>ab+</code>	为在文件尾读和写而打开或创建

使用字符**b**作为*type*的一部分，使得标准 I/O系统可以区分文本文件和二进制文件。因为 UNIX内核并不对这两种文件进行区分，所以在 UNIX系统环境下指定字符**b**作为*type*的一部分实际上并无作用。

对于fdopen，*type*参数的意义则稍有区别。因为该描述符已被打开，所以 fdopen为写而打开并不截短该文件。（例如，若该描述符原来是由 open函数打开的，该文件那时已经存在，则其O_TRUNC标志将决定是否截短该文件。fdopen函数不能截短它为写而打开的任一文件。）另外，标准I/O添加方式也不能用于创建该文件（因为如若一个描述符引用一个文件，则该文件一定已经存在）。

当用添加类型打开一文件后，则每次写都将数据写到文件的当前尾端处。如若有多个进程用标准I/O添加方式打开了同一文件，那么来自每个进程的数据都将正确地写到文件中。

4.3+BSD以前的伯克利版本以及Kernighan和Ritchie [1988] 177页上所示的简单版本并不能正确地处理添加方式。这些版本在打开流时，调用 lseek到达文件尾端。在涉及多个进程时，为了正确地支持添加方式，该文件必须用 O_APPEND标志打开，我们已在3.3节中对此进行了讨论。在每次写前，做一次 lseek操作同样也不能正确工作（如同在3.11节中讨论的一样）。

当以读和写类型打开一文件时（*type*中+号），具有下列限制：

- 如果中间没有fflush、fseek、fsetpos或rewind，则在输出的后面不能直接跟随输入。
- 如果中间没有fseek、fsetpos或rewind,或者一个输出操作没有到达文件尾端，则在输入操作之后不能直接跟随输出。

按照表5-2，我们在表5-3中列出了打开一个流的六种不同的方式。

表5-3 打开一个标准I/O流的六种不同的方式

限制	r	w	a	r+	w+	a+
文件必须已存在	.			.		
擦除文件以前的内容		.			.	
流可以读
流可以写	
流只可在尾端处写			.			.

注意，在指定w或a类型创建一个新文件时，我们无法说明该文件的存取许可权位（第 3章中所述的open函数和creat函数则能做到这一点）。POSIX.1要求以这种方式创建的文件具有下列存取许可权：

```
S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH
```

除非流引用终端设备，否则按系统默认，它被打开时是全缓存的。若流引用终端设备，则该流是行缓存的。一旦打开了流，那么在对该流执行任何操作之前，如果希望，则可使用前节所述的setbuf和setvbuf改变缓存的类型。

调用fclose关闭一个打开的流。

```
#include <stdio.h>
int fclose(FILE *fp);
```

返回：若成功则为 0，若出错则为 EOF

在该文件被关闭之前，刷新缓存中的输出数据。缓存中的输入数据被丢弃。如果标准 I/O 库已经为该流自动分配了一个缓存，则释放此缓存。

当一个进程正常终止时（直接调用 `exit` 函数，或从 `main` 函数返回），则所有带未写缓存数据的标准 I/O 流都被刷新，所有打开的标准 I/O 流都被关闭。

5.6 读和写流

一旦打开了流，则可在三种不同类型的非格式化 I/O 中进行选择，对其进行读、写操作。（5.11 节说明了格式化 I/O 函数，例如 `printf` 和 `scanf`。）

(1) 每次一个字符的 I/O。一次读或写一个字符，如果流是带缓存的，则标准 I/O 函数处理所有缓存。

(2) 每次一行的 I/O。使用 `fgetc` 和 `fputs` 一次读或写一行。每行都以一个新行符终止。当调用 `fgetc` 时，应说明能处理的最大行长。5.7 节将说明这两个函数。

(3) 直接 I/O。`fread` 和 `fwrite` 函数支持这种类型的 I/O。每次 I/O 操作读或写某种数量的对象，而每个对象具有指定的长度。这两个函数常用于从二进制文件中读或写一个结构。5.9 节将说明这两个函数。

直接 I/O (direct I/O) 这个术语来自 ANSI C 标准，有时也被称为：二进制 I/O、一次一个对象 I/O、面向记录的 I/O 或面向结构的 I/O。

5.6.1 输入函数

以下三个函数可用于一次读一个字符。

```
#include <stdio.h>

int getc(FILE *fp);

int fgetc(FILE *fp);

int getchar(void);
```

三个函数的返回：若成功则为下一个字符，若已处文件尾端或出错则为 EOF

函数 `getchar` 等同于 `getc(stdin)`。前两个函数的区别是 `getc` 可被实现为宏，而 `fgetc` 则不能实现为宏。这意味着：

(1) `getc` 的参数不应当是具有副作用的表达式。

(2) 因为 `fgetc` 一定是个函数，所以可以得到其地址。这就允许将 `fgetc` 的地址作为一个参数传送给另一个函数。

(3) 调用 `fgetc` 所需时间很可能长于调用 `getc`，因为调用函数通常所需的时间长于调用宏。检验一下 `<stdio.h>` 头文件的大多数实现，从中可见 `getc` 是一个宏，其编码具有较高的工作效率。

这三个函数以 `unsigned char` 类型转换为 `int` 的方式返回下一个字符。说明为不带符号的理由是，如果最高位为 1 也不会使返回值为负。要求整型返回值的理由是，这样就可以返回所有可能的字符值再加上一个已发生错误或已到达文件尾端的指示值。在 `<stdio.h>` 中的常数 `EOF` 被要

求是一个负值，其值经常是 - 1。这就意味着不能将这三个函数的返回值存放在一个字符变量中，以后还要将这些函数的返回值与常数EOF相比较。

注意，不管是出错还是到达文件尾端，这三个函数都返回同样的值。为了区分这两种不同的情况，必须调用ferror或feof。

```
#include <stdio.h>

int ferror(FILE *fp);

int feof(FILE *fp);
```

两个函数返回：若条件为真则为非0（真），否则为0（假）

```
void clearerr(FILE *fp);
```

在大多数实现的FILE对象中，为每个流保持了两个标志：

- 出错标志。
- 文件结束标志。

调用clearerr则清除这两个标志。

从一个流读之后，可以调用ungetc将字符再送回流中。

```
#include <stdio.h>

int ungetc(int c, FILE fp);
```

返回：若成功则为c，若出错则为EOF

送回到流中的字符以后又可从流中读出，但读出字符的顺序与送回的顺序相反。应当了解，虽然ANSI C允许支持任何数量的字符回送的实现，但是它要求任何一种实现都要支持一个字符的回送功能。

回送的字符，不一定必须是上一次读到的字符。EOF不能回送。但是当已经到达文件尾端时，仍可以回送一字符。下次读将返回该字符，再次读则返回EOF。之所以能这样做的原因是一次成功的ungetc调用会清除该流的文件结束指示。

当正在读一个输入流，并进行某种形式的分字或分记号操作时，会经常用到回送字符操作。有时需要先看一看下一个字符，以决定如何处理当前字符。然后就需要方便地将刚查看的字符送回，以便下一次调用getc时返回该字符。如果标准I/O库不提供回送能力，就需将该字符存放在一个我们自己的变量中，并设置一个标志以便判别在下次需要一个字符时是调用getc，还是从我们自己的变量中取用。

5.6.2 输出函数

对应于上面所述的每个输入函数都有一个输出函数。

```
#include <stdio.h>

int putc(int c, FILE fp);

int fputc(int c, FILE fp);
```

```
int putchar(int c);
```

三个函数返回：若成功则为 `c`，若出错则为 `EOF`

与输入函数一样，`putchar(c)` 等同于 `putc(c, stdout)`，`putc` 可被实现为宏，而 `fputc` 则不能实现为宏。

5.7 每次一行 I/O

下面两个函数提供每次输入一行的功能。

```
#include <stdio.h>
char *fgets(char *buf, int n, FILE *fp);
char *gets(char *buf);
```

两个函数返回：若成功则为 `buf`，若已处文件尾端或出错则为 `NULL`

这两个函数都指定了缓存地址，读入的行将送入其中。`gets`从标准输入读，而 `fgets`则从指定的流读。

对于 `fgets`，必须指定缓存的长度 `n`。此函数一直读到下一个新行符为止，但是不超过 `n - 1` 个字符，读入的字符被送入缓存。该缓存以 `null` 字符结尾。如若该行，包括最后一个新行符的字符数超过 `n - 1`，则只返回一个不完整的行，而且缓存总是以 `null` 字符结尾。对 `fgets` 的下次调用会继续读该行。

`gets` 是一个不推荐使用的函数。问题是调用者在使用 `gets` 时不能指定缓存的长度。这样就可能造成缓存越界（如若该行长于缓存长度），写到缓存之后的存储空间中，从而产生不可预料的后果。这种缺陷曾被利用，造成 1988 年的因特网蠕虫事件。有关说明请见 1989.6. Communications of the ACM (vol.32, no.6)。 `gets` 与 `fgets` 的另一个区别是，`gets` 并不将新行符存入缓存中。

这两个函数对新行符进行处理方面的差别与 UNIX 的进展有关。早在 V7 的手册中就说明：“为了向后兼容，`gets` 删除新行符，而 `fgets` 则保持新行符。”

虽然 ANSI C 要求提供 `gets`，但请不要使用它。

`fputs` 和 `puts` 提供每次输出一行的功能。

```
#include <stdio.h>
int fputs(const char *str, FILE *fp);
int puts(const char *str);
```

两个函数返回：若成功则为非负值，若出错则为 `EOF`

函数 `fputs` 将一个以 `null` 符终止的字符串写到指定的流，终止符 `null` 不写出。注意，这并不一定是每次输出一行，因为它并不要求在 `null` 符之前一定是新行符。通常，在 `null` 符之前是一个新行符，但并不要求总是如此。

`puts` 将一个以 `null` 符终止的字符串写到标准输出，终止符不写出。但是，`puts` 然后又将一个新行符写到标准输出。

puts并不像它所对应的gets那样不安全。但是我们还是应避免使用它，以免需要记住它在最后又加上了一个新行符。如果总是使用fgets和fputs，那么就会熟知在每行终止处我们必须自己加一个新行符。

5.8 标准I/O的效率

使用前面所述的函数，我们应该对标准I/O系统的效率有所了解。程序5-1类似于程序3-3，它使用getc和putc将标准输入复制到标准输出。这两个函数可以实现为宏。

程序5-1 用getc和putc将标准输入复制到标准输出

```
#include    "ourhdr.h"

int
main(void)
{
    int    c;

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

    if (ferror(stdin))
        err_sys("input error");

    exit(0);
}
```

可以用fgetc和fputc改写该程序，这两个一定是函数，而不是宏（没有给出对源代码更改的细节）。

最后，我们还编写了一个读、写行的版本，见程序5-2。

程序5-2 用fgets和fputs将标准输入复制到标准输出

```
#include    "ourhdr.h"

int
main(void)
{
    char    buf[MAXLINE];

    while (fgets(buf, MAXLINE, stdin) != NULL)
        if (fputs(buf, stdout) == EOF)
            err_sys("output error");

    if (ferror(stdin))
        err_sys("input error");

    exit(0);
}
```

注意，在程序5-1和程序5-2中，没有显式地关闭标准I/O流。我们知道exit函数将会刷新任何未写的的数据，然后关闭所有打开的流（我们将在8.5节讨论这一点）。将这三个程序的时间与表3-1中的时间进行比较是很有趣的。表5-4中显示了对同一文件（1.5M字节，30,000行）进行操作所得的数据。

表5-4 使用标准I/O例程得到的时间结果

函 数	用户CPU (秒)	系统CPU (秒)	时钟时间 (秒)	程序正文字节数
表3-1中的最佳时间	0.0	0.3	0.3	
fgets, fputs	2.2	0.3	2.6	184
getc, putc	4.3	0.3	4.8	384
fgetc, fputc	4.6	0.3	5.0	152
表3-1中的单字节时间	23.8	397.9	423.4	

对于这三个标准I/O版本的每一个，其用户CPU时间都大于表3-1中的最佳read版本，因为每次读一个字符版本中有一个要执行150万次的循环，而在每次读一行的版本中有一个要执行30 000次的循环。在read版本中，其循环只需执行180次（对于缓存长度为8192字节）。因为系统CPU时间都相同，所以用户CPU时间的差别造成了时钟时间的差别。

系统CPU时间相同的原因是因为所有这些程序对内核提出的读、写请求数相同。注意，使用标准I/O例程的一个优点是无需考虑缓存及最佳I/O长度的选择。在使用fgets时需要考虑最大行长，但是最佳I/O长度的选择要方便得多。

表5-4中的最后一列是每个main函数的文本空间字节数（由C编译产生的机器指令）。从中可见，使用getc的版本在文本空间中作了getc和putc的宏代换，所以它所需使用的指令数超过了调用fgetc和fputc函数所需指令数。观察getc版本和fgetc版本在用户CPU时间方面的差别，可以看到在程序中作宏代换和调用两个函数在进行本测试的系统上并没有造成多大差别。

使用每次一行I/O版本其速度大约是每次一个字符版本的两倍（包括用户CPU时间和时钟时间）。如果fgets和fputs函数用getc和putc实现（例如，见Kernighan和Ritchie [1988]的7.7节），那么，可以预期fgets版本的时间会与getc版本相接近。实际上，可以预料每次一行的版本会更慢一些，因为除了现已存在的60 000次函数调用外还需增加3百万次宏调用。而在本测试中所用的每次一行函数是用memcpy(3)实现的。通常，为了提高效率，memcpy函数用汇编语言而非C语言编写。

这些时间数字的最后一个有趣之处在于：fgetc版本较表3-1中BUFSIZE = 1的版本要快得多。两者都使用了约3百万次的函数调用，而fgetc版本的速度在用户CPU时间方面，大约是后者的5倍，而在时钟时间方面则几乎是100倍。造成这种差别的原因是：使用read的版本执行了3百万次函数调用，这也就引起3百万次系统调用。而对于fgetc版本，它也执行3百万次函数调用，但是这只引起360次系统调用。系统调用与普通的函数调用相比是很花费时间的。

需要声明的是这些时间结果只在某些系统上才有效。这种时间结果依赖于很多实现的特征，而这种特征对于不同的UNIX系统却可能是不同的。尽管如此，使这样一组数据，并对各种版本的差别作出解释，这有助于我们更好地了解系统。

在本节及3.9节中我们学到的基本事实是：标准I/O库与直接调用read和write函数相比并不慢很多。我们观察到使用getc和putc复制1M字节数据大约需3.0秒CPU时间。对于大多数比较复杂的应用程序，最主要的用户CPU时间是由应用本身的各种处理花费的，而不是由标准I/O例程消耗的。

5.9 二进制I/O

5.6节中的函数以一次一个字符或一次一行的方式进行操作。如果为二进制I/O，那么我们更愿意一次读或写整个结构。为了使用getc或putc做到这一点，必须循环通过整个结构，一次

读或写一个字节。因为 `fputs` 在遇到 `null` 字节时就停止，而在结构中可能含有 `null` 字节，所以不能使用每次一行函数实现这种要求。相类似，如果输入数据中包含有 `null` 字节或新行符，则 `fgets` 也不能正确工作。因此，提供了下列两个函数以执行二进制 I/O 操作。

```
#include <stdio.h>

size_t fread(void *ptr, size_t size, size_t nobj, FILE fp);

size_t fwrite(const void *ptr, size_t size, size_t nobj, FILE fp);
```

两个函数的返回：读或写的对象数

这些函数有两个常见的用法：

(1) 读或写一个二进制数组。例如，将一个浮点数组的第 2 至第 5 个元素写至一个文件上，可以写作：

```
float data[10];

if (fwrite(&data[2], sizeof(float), 4, fp) != 4)
    err_sys("fwrite error");
```

其中，指定 `size` 为每个数组元素的长度，`nobj` 为欲写的元素数。

(2) 读或写一个结构。例如，可以写作：

```
struct {
    short count;
    long total;
    char name[NAMESIZE];
} item;

if (fwrite(&item, sizeof(item), 1, fp) != 1)
    err_sys("fwrite error");
```

其中，指定 `size` 为结构的长度，`nobj` 为 1（要写的对象数）。

将这两个例子结合起来就可读或写一个结构数组。为了做到这一点，`size` 应当是该结构的 `sizeof`，`nobj` 应是该数组中的元素数。

`fread` 和 `fwrite` 返回读或写的对象数。对于读，如果出错或到达文件尾端，则此数字可以少于 `nobj`。在这种情况下，应调用 `ferror` 或 `feof` 以判断究竟是那一种情况。对于写，如果返回值少于所要求的 `nobj`，则出错。

使用二进制 I/O 的基本问题是，它只能用于读已写在同一系统上的数据。多年之前，这并无问题（那时，所有 UNIX 系统都运行于 PDP-11 上），而现在，很多异构系统通过网络相互连接起来，而且，这种情况已经非常普遍。常常有这种情形，在一个系统上写的的数据，在另一个系统上处理。在这种环境下，这两个函数可能就不能正常工作，其原因是：

(1) 在一个结构中，同一成员的位移量可能随编译程序和系统的不同而异（由于不同的对准要求）。确实，某些编译程序有一选择项，它允许紧密包装结构（节省存储空间，而运行性能则可能有所下降）或准确对齐，以便在运行时易于存取结构中的各成员。这意味着即使在单一系统上，一个结构的二进制存放方式也可能因编译程序的选择项而不同。

(2) 用来存储多字节整数和浮点值的二进制格式在不同的系统结构间也可能不同。

在不同系统之间交换二进制数据的实际解决方法是使用较高层次的协议。关于网络协议使用的交换二进制数据的某些技术，请参阅 Stevens [1990] 的 18.2 节。

在8.13节中，我们将再回到 `fread`函数，那时将用它读一个二进制结构——UNIX的进程记账记录。

5.10 定位流

有两种方法定位标准I/O流。

(1) `ftell`和`fseek`。这两个函数自V7以来就存在了，但是它们都假定文件的位置可以存放在一个长整型中。

(2) `fgetpos`和`fsetpos`。这两个函数是新由ANSI C引入的。它们引进了一个新的抽象数据类型`fpos_t`，它记录文件的位置。在非UNIX系统中，这种数据类型可以定义为记录一个文件的位置所需的长度。

需要移植到非UNIX系统上运行的应用程序应当使用`fgetpos`和`fsetpos`。

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

```
long ftell(FILE *fp);
```

返回：若成功则为当前文件位置指示，若出错则为 - 1L

```
int fseek(FILE *fp, long offset, int whence);
```

返回：若成功则为0，若出错则为非0

```
void rewind(FILE *fp);
```

对于一个二进制文件，其位置指示器是从文件起始位置开始度量，并以字节为计量单位的。`ftell`用于二进制文件时，其返回值就是这种字节位置。为了用`fseek`定位一个二进制文件，必须指定一个字节 `offset`，以及解释这种位移量的方式。`whence`的值与3.6节中`lseek`函数的相同：`SEEK_SET`表示从文件的起始位置开始，`SEEK_CUR`表示从当前文件位置，`SEEK_END`表示从文件的尾端。ANSI C并不要求一个实现对二进制文件支持`SEEK_END`规格说明，其原因是某些系统要求二进制文件的长度是某个幻数的整数倍，非实际内容部分则充填为0。但是在UNIX中，对于二进制文件`SEEK_END`是得到支持的。

对于文本文件，它们的文件当前位置可能不以简单的字节位移量来度量。再一次，这主要也是在非UNIX系统中，它们可能以不同的格式存放文本文件。为了定位一个文本文件，`whence`一定要是`SEEK_SET`，而且`offset`只能有两种值：0（表示反绕文件至其起始位置），或是对该文件的`ftell`所返回的值。使用`rewind`函数也可将一个流设置到文件的起始位置。

正如我们已提及的，下列两个函数是C标准新引进的。

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

```
int fgetpos(FILE *fp, fpos_t *pos);
```

```
int fsetpos(FILE *fp, const fpos_t *pos);
```

两个函数返回：若成功则为0，若出错则为非0

`fgetpos`将文件位置指示器的当前值存入由 `pos`指向的对象中。在以后调用`fsetpos`时，可以使用此值将流重新定位至该位置。

5.11 格式化I/O

5.11.1 格式化输出

执行格式化输出处理的是三个printf函数。

```
#include <stdio.h>

int printf(const char *format, ...);

int fprintf(FILE *fp, const char *format, ...);
```

两个函数返回：若成功则为输出字符数，若输出出错则为负值

```
int sprintf(char *buf, const char *format, ...);
```

返回：存入数组的字符数

printf将格式化数据写到标准输出，fprintf写至指定的流，sprintf将格式化的字符送入数组buf中。sprintf在该数组的尾端自动加一个null字节，但该字节不包括在返回值中。

4.3BSD定义sprintf返回其第一个参数（缓存指针，类型为char*），而不是一个整型。ANSI C要求sprintf返回一个整型。

注意，sprintf可能会造成由buf指向的缓存的溢出。保证该缓存有足够长度是调用者的责任。

对这三个函数可能使用的各种格式变换，请参阅UNIX手册，或Kernighan和Ritchie [1988]的附录B。

下列三种printf族的变体类似于上面的三种，但是可变参数表(...)替换成了arg。

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

int vprintf(const char *format, va_list arg);

int vfprintf(FILE *fp, const char *format, va_list arg);
```

两个函数返回：若成功则为输出字符数，若输出出错则为负值

```
int vsprintf(char *buf, const char *format, va_list arg);
```

返回：存入数组的字符数

在附录B的出错例程中，将使用vsprintf函数。

关于ANSI C标准中有关可变长度参数表的详细说明请参阅Kernighan和Ritchie [1988]的7.3节。应当了解的是，由ANSI C提供的可变长度参数表例程（<stdarg.h>头文件和相关的例程）与由SVR3（以及更早版本）和4.3BSD提供的<varargs.h>例程是不同的。

5.11.2 格式化输入

执行格式化输入处理的是三个scanf函数。

```
#include<stdio.h>

int scanf(const char *format, ...);

int fscanf(FILE *fp, const char *format, ...);

int sscanf(const char *buf, const char *format, ...);
```

三个函数返回：指定的输入项数，若输入出错，或在任意变换前已至文件尾端则为 EOF

如同printf族一样，关于这三个函数的各个格式选择项的详细情况，请参阅 UNIX手册。

5.12 实现细节

正如前述，在UNIX中，标准I/O库最终都要调用第3章中说明的I/O例程。每个I/O流都有一个与其相关联的文件描述符，可以对一个流调用 `fileno` 以获得其描述符。

```
#include <stdio.h>
int fileno(FILE *fp);
```

返回：与该流相关联的文件描述符

如果要调用 `dup` 或 `fcntl` 等函数，则需要此函数。

为了了解你所使用的系统中标准 I/O 库的实现，最好从头文件 `<stdio.h>` 开始。从中可以看到：FILE 对象是如何定义的，每个流标志的定义，定义为宏的各个标准 I/O 例程（例如 `getc`）。Kernighan 和 Ritchie [1988] 中的 8.5 节含有一个简单的实现，从中可以看到很多 UNIX 实现的基本样式。Plauger [1992] 的第 12 章提供了标准 I/O 库一种实现的全部源代码。4.3 + BSD 中标准 I/O 库的实现（由 Chris Torek 编写）也是可以公开使用的。

实例

程序 5-3 为三个标准流以及一个与一个普通文件相关联的流打印有关缓存状态信息。注意，在打印缓存状态信息之前，先对每个流执行 I/O 操作，因为第一个 I/O 操作通常就造成成为该流分配缓存。结构成员 `_flag`、`_bufsiz` 以及常数 `_IONBF` 和 `_IOLBF` 是由作者所使用的系统定义的。

如果运行程序 5-3 两次，一次使三个标准流与终端相连接，另一次使它们重定向到普通文件，则所得结果是：

```
$ a.out                               stdin, stdout 和 stderr 都连至终端
enter any character                    键入新行符

one line to standard error
stream = stdin, line buffered, buffer size = 128
stream = stdout, line buffered, buffer size = 128
stream = stderr, unbuffered, buffer size = 8
stream = /etc/motd, fully buffered, buffer size = 8192
$ a.out < /etc/termcap > std.out 2> std.err
                                     三个流都重定向，再次运行该程序

$ cat std.err
one line to standard error
$ cat std.out
enter any character
stream = stdin, fully buffered, buffer size = 8192
```



```
stream = stdout, fully buffered, buffer size = 8192
stream = stderr, unbuffered, buffer size = 8
stream = /etc/motd, fully buffered, buffer size = 8192
```

程序5-3 对各个标准I/O流打印缓存状态信息

```
#include "ourhdr.h"
void pr_stdio(const char *, FILE *);
int
main(void)
{
    FILE *fp;

    fputs("enter any character\n", stdout);
    if (getchar() == EOF)
        err_sys("getchar error");
    fputs("one line to standard error\n", stderr);

    pr_stdio("stdin", stdin);
    pr_stdio("stdout", stdout);
    pr_stdio("stderr", stderr);

    if ( (fp = fopen("/etc/motd", "r")) == NULL)
        err_sys("fopen error");
    if (getc(fp) == EOF)
        err_sys("getc error");
    pr_stdio("/etc/motd", fp);
    exit(0);
}

void
pr_stdio(const char *name, FILE *fp)
{
    printf("stream = %s, ", name);
        /* following is nonportable */
    if (fp->_flag & _IONBF) printf("unbuffered");
    else if (fp->_flag & _IOLBF) printf("line buffered");
    else /* if neither of above */ printf("fully buffered");
    printf(", buffer size = %d\n", fp->_bufsiz);
}
```

从中可见，该系统的默认是：当标准输入、输出连至终端时，它们是行缓存的。行缓存的长度是128字节。注意，这并没有将输入、输出的行长限制为128字节，这只是缓存的长度。如果要将512字节的行写到标准输出则要进行四次 write 系统调用。当将这两个流重新定向到普通文件时，它们就变成是全缓存的，其缓存长度是该文件系统优先选用的 I/O 长度（从 stat 结构中得到的 st_blksize）。从中也可看到，标准出错如它所应该的那样是非缓存的，而普通文件按系统默认是全缓存的。

5.13 临时文件

标准I/O库提供了两个函数以帮助创建临时文件。

```
#include <stdio.h>

char *tmpnam(char *ptr);
```

返回：指向一唯一路径名的指针

```
FILE *tmpfile(void);
```

返回：若成功则为文件指针，若出错则为 NULL

tmpnam产生一个与现在文件名不同的一个有效路径名字符串。每次调用它时，它都产生一个不同的路径名，最多调用次数是TMP_MAX。TMP_MAX定义在<stdio.h>中。

虽然TMP_MAX是由ANSI C定义的。但该C标准只要求其值至少应为25。但是，XPG3却要求其值至少为10 000。在此最小值允许一个实现使用4位数字作为临时文件名的同时（0000~9999），大多数UNIX实现使用的却是大、小写字符。

若ptr是NULL，则所产生的路径名存放在一个静态区中，指向该静态区的指针作为函数值返回。下一次再调用tmpnam时，会重写该静态区。（这意味着，如果我们调用此函数多次，而且想保存路径名，则我们应当保存该路径名的副本，而不是指针的副本。）如若ptr不是NULL，则认为它指向长度至少是L_tmpnam个字符的数组。（常数L_tmpnam定义在头文件<stdio.h>中。）所产生的路径名存放在该数组中，ptr也作为函数值返回。

tmpfile创建一个临时二进制文件（类型wb+），在关闭该文件或程序结束时将自动删除这种文件。注意，UNIX对二进制文件不作特殊区分。

实例

程序5-4说明了这两个函数的应用。若执行程序5-4，则得：

```
$ a.out
/usr/tmp/aaaa00470
/usr/tmp/baaa00470
one line of output
```

加到临时文件名中的5位数字后缀是进程ID，这就保证了对各个进程产生的路径名各不同。

tmpfile函数经常使用的标准UNIX技术是先调用tmpnam产生一个唯一的路径名，然后立即unlink它。

程序5-4 tmpnam和tmpfile函数实例

```
#include "ourhdr.h"

int
main(void)
{
    char    name[L_tmpnam], line[MAXLINE];
    FILE    *fp;

    printf("%s\n", tmpnam(NULL));      /* first temp name */
    tmpnam(name);                      /* second temp name */
    printf("%s\n", name);

    if ( (fp = tmpfile()) == NULL)     /* create temp file */
        err_sys("tmpfile error");
    fputs("one line of output\n", fp); /* write to temp file */
    rewind(fp);                        /* then read it back */
    if (fgets(line, sizeof(line), fp) == NULL)
        err_sys("fgets error");
    fputs(line, stdout);               /* print the line we wrote */
}
```

```
    exit(0);
}
```

请回忆4.15节，对一个文件解除连接并不删除其内容，关闭该文件时才删除其内容。`tempnam`是`tmpnam`的一个变体，它允许调用者为所产生的路径名指定目录和前缀。

```
# include <stdio.h>

char    *tempnam(const char directory, const char prefix;
```

返回：指向一唯一路径名的指针

对于目录有四种不同的选择，并且使用第一个为真的作为目录：

- (1) 如果定义了环境变量 `TMPDIR`，则用其作为目录。（在7.9节中将说明环境变量。）
- (2) 如果参数 `directory` 非 `NULL`，则用其作为目录。
- (3) 将 `<stdio.h>` 中的字符串 `P_tmpdir` 用作为目录。
- (4) 将本地目录，通常是 `/tmp`，用作为目录。

如果 `prefix` 非 `NULL`，则它应该是最多包含5个字符的字符串，用其作为文件名的头几个字符。

该函数调用 `malloc` 函数分配动态存储区，用其存放所构造的路径名。当不再使用此路径名时就可释放此存储区（7.8节将说明 `malloc` 和 `free` 函数）。

`tempnam`不是POSIX.1和ANSI C的所属部分，它是XPG3的所属部分。
我们所说明的实现对应于SVR4和4.3+BSD。XPG3版本除了不支持环境变量 `TMPDIR`，其他都与此相同。

实例

程序5-5显示了 `tempnam` 的应用。

程序5-5 `tempnam`函数的应用

```
#include    "ourhdr.h"

int
main(int argc, char *argv[])
{
    if (argc != 3)
        err_quit("usage: a.out <directory> <prefix>");

    printf("%s\n", tempnam( argv[1][0] != ' ' ? argv[1] : NULL,
                          argv[2][0] != ' ' ? argv[2] : NULL) );

    exit(0);
}
```

注意，如果命令行参数（目录或前缀）中的任一个以空白开始，则将其作为 `null` 指针传送给该函数。下面显示使用该程序的各种方式。

```
$ a.out /home/stevens TEMP    指定目录和前缀
/home/stevens/TEMPAAAa00571
$ a.out " " PFX                使用默认目录：P_tmpdir
/usr/tmp/PFXAAAa00572
$ TMPDIR=/tmp a.out /usr/tmp  使用环境变量；无前缀
```

```

/tmp/AAAA00573          环境变量复设目录
$ TMPDIR=/no/such/dir a.out/tmp QQQQ
/tmp/QQQQAAAA00574     忽略无效环境目录
$ TMPDIR=/no/such/file a.out /etc/uucp MMMMM
/usr/tmp/MMMMMAAA00575  忽略无效环境和无效目录两者

```

上述选择目录名的四个步骤按序执行，该函数也检查相应的目录名是否有意义。如果该目录并不存在（例如 /no/such/dir），或者对该目录并无写许可权（例如 /etc/uucp），则跳过这些，试探对目录名的下一次选择。从本例中可以看出在路径名中如何使用进程 ID，也可看出在本实现中，P_tmpdir 目录是 /usr/tmp。设置环境变量的技术（程序名前的 TMPDIR=）适用于 Bourne shell 和 KornShell。

5.14 标准 I/O 的替代软件

标准 I/O 库并不完善。Korn 和 Vo [1991] 列出了它的很多不足之处——某些属于基本设计，但是大多数则与各种不同的实现有关。

在标准 I/O 库中，一个效率不高的不足之处是需要复制的数据量。当使用每次一行函数 `fgets` 和 `fputs` 时，通常需要复制两次数据：一次是在内核和标准 I/O 缓存之间（当调用 `read` 和 `write` 时），第二次是在标准 I/O 缓存和用户程序中的行缓存之间。快速 I/O 库 [AT&T 1990a 中的 `fiio(3)`] 避免了这一点，其方法是使读一行的函数返回指向该行的指针，而不是将该行复制到另一个缓存中。Hume [1988] 报告了由于作了这种更改，`grep(1)` 公用程序的速度增加了 2 倍。

Korn 和 Vo [1991] 说明了标准 I/O 库的另一种代替版：`sfio`。这一软件包在速度上与 `fiio` 相近，通常快于标准 I/O 库。`sfio` 也提供了一些新的特征：推广了 I/O 流，使其不仅可以代表文件，也可代表存储区；可以编写处理模块，并以栈方式将其压入 I/O 流，这样就可以改变一个流的操作；较好的异常处理等。

Krieger, Stumm 和 Unrau [1992] 说明了另一个代换软件包，它使用了映照文件——`mmap` 函数，我们将在 12.9 节中说明此函数。该新软件包称为 ASI (Alloc Stream Interface)。其程序界面类似于 UNIX 存储分配函数（`malloc`, `realloc` 和 `free`，这些将在 7.8 节中说明）。与 `sfio` 软件包相同，ASI 使用指针力图减少数据复制量。

5.15 小结

大多数 UNIX 应用程序都使用标准 I/O 库。本章说明了该库提供的所有函数，某些实现细节和效率方面的考虑。应该看到标准 I/O 库使用了缓存机制，而这种机制是产生很多问题，引起很多混淆的一个领域。

习题

- 5.1 用 `setvbuf` 完成 `setbuf`。
- 5.2 5.8 节中程序利用 `fgets` 和 `fputs` 函数拷贝文件，每次 I/O 操作只拷贝一行。若将程序中的 `MAXLINE` 改为 4，当拷贝的行超过该最大值时会出现什么情况？
- 5.3 `printf` 返回 0 值表示什么？
- 5.4 下面的代码在一些机器上运行正确，而在另外一些机器运行时出错，解释问题所在。

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

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

- 5.5 为什么tempnam限制前缀为5个字符？
- 5.6 对标准I/O流如何使用fsync函数（见4.24节）？
- 5.7 在程序1-5和1-8中打印的提示信息没有包含换行符，程序也没有调用 fflush函数，请解释提示信息是如何输出的？

第6章 系统数据文件和信息

6.1 引言

有很多操作需要使用一些与系统有关的数据文件，例如，口令文件 `/etc/passwd`和组文件 `/etc/group`就是经常由多种程序使用的两个文件。用户每次登录入 UNIX系统，以及每次执行 `ls -l`命令时都要使用口令文件。

由于历史原因，这些数据文件都是 ASCII文本文件，并且使用标准 I/O库读这些文件。但是，对于较大的系统，顺序扫描口令文件变得很花费时间，我们想以非 ASCII文本格式存放这些文件，但仍向应用程序提供一个可以处理任何一种文件格式的界面。对于这些数据文件的可移植界面是本章的主题。本章也包括了系统标识函数、时间和日期函数。

6.2 口令文件

UNIX口令文件 (POSIX.1 则将其称为用户数据库) 包含了表 6-1 中所示的各字段，这些字段包含在 `<pwd.h>` 中定义的 `passwd` 结构中。

注意，POSIX.1 只指定 `passwd` 结构中 7 个字段中的 5 个。另外 2 个元素由 SVR4 和 4.3+BSD 支持。

表6-1 /etc/passwd文件中的字段

说 明	struct passwd成员	POSIX.1
用户名	char *pw_name	•
加密口令	char *pw_passwd	
数值用户ID	uid_t pw_uid	•
数值组ID	gid_t pw_gid	•
注释字段	char *pw_gecos	
初始工作目录	char *pw_dir	•
初始shell (用户程序)	char *pw_shell	•

由于历史原因，口令文件是 `/etc/passwd`，而且是一个文本文件。每一行包含表 6-1 中所示的 7 个字段，字段之间用冒号相分隔。例如，该文件中可能有下列三行：

```
root:jheVopR58x9Fx:0:1:The superuser:/:/bin/sh
nobody:*:65534:65534:/:/
stevens:3hKVD8R58r9Fx:224:20:Richard Stevens:/home/stevens:/bin/ksh
```

关于这些登录项请注意下列各点：

- 通常有一个登录项，其用户名为 `root`，其用户 ID 是 0 (超级用户)。
- 加密口令字段包含了经单向密码算法处理过的用户口令副本。因为此算法是单向的，所

以不能从加密口令猜测到原来的口令。当前使用的算法(见Morris和Thompson〔1979〕)总是产生13个可打印字符(在64字符集〔a-zA-Z0-9./〕中)。因为用户名nobody的加密口令字段只包含一个字符(*),所以加密口令决不会与此值相匹配。此用户名可用于网络服务器,这些服务器允许我们登录到一个系统,但其用户ID和组ID(65534)并不提供优先权。用此用户ID和组ID可存取的文件只是大家都可读、写的文件。(这假定用户ID65534和组ID65534并不拥有任何文件。)在本节稍后部分我们将讨论对口令文件最近所作的更改(阴影口令)。

- 口令文件中的某些字段可能是空。如果密码口令字段为空,这通常就意味着该用户没有口令(不推荐这样做)。nobody有两个空白字段:注释字段和初始shell字段。空白注释字段不产生任何影响。空白shell字段则表示取系统默认值,通常是/bin/sh。

- 支持finger(1)命令的某些UNIX系统支持注释字段中的附加信息。其中,各部分之间都用逗号分隔:用户姓名,办公室地点,办公室电话号码,家庭电话号码。另外,如果注释字段中的用户姓名是一个&,则它被替换为登录名。例如,可以有下列记录:

```
stevens:3hKVD8R58r9Fx:224:20:Richard &, B232, 555-1111, 555-2222:
/home/stevens:/bin/ksh
```

即使你所使用的系统并不支持finger命令,这些信息仍可存放在注释字段中,因为该字段只是一个注释,并不由系统公用程序解释。

POSIX.1只定义了两个存取口令文件中信息的函数。在给出用户登录名或数值用户ID后,这两个函数就能查看相关记录。

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

struct passwd *getpwuid(uid_t uid);

struct passwd *getpwnam(const char *name);
```

两个函数返回:若成功则为指针,若出错则为NULL

getpwuid由ls(1)程序使用,以便将包含在一个i节点中的数值用户ID映照为用户登录名。getpwnam在键入登录名时由login(1)程序使用。

这两个函数都返回一个指向passwd结构的指针,该结构已由这两个函数在执行时填入了所需的信息。该结构通常是在相关函数内的静态变量,只要调用相关函数,其内容就会被重写。

如果要查看的只是一个登录名或用户ID,那么这两个POSIX.1函数能满足要求,但是也有些程序要查看整个口令文件。下列三个函数则可用于此。

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

struct passwd *getpwent(void);
```

返回:若成功则为指针,若出错或到达文件尾端则为NULL

```
void setpwent(void);

void endpwent(void);
```

POSIX.1没有定义这三个函数，但它们受到SVR4和4.3+BSD的支持。

调用getpwent时，它返回口令文件中的下一个记录。如同上面所述的两个 POSIX.1函数一样，它返回一个由它填写好的 password结构的指针。每次调用此函数时都重写该结构。在第一次调用该函数时，它打开它所使用的各个文件。在使用本函数时，对口令文件中各个记录安排的顺序并无要求。

函数setpwent反绕它所使用的文件，endpwent则关闭这些文件。在使用 getpwent查看完口令文件后，一定要调用endpwent关闭这些文件。getpwent知道什么时间它应当打开它所使用的文件(第一次被调用时)，但是它并不能知道何时关闭这些文件。

实例

程序6-1给出了函数getpwnam的一个实现。

程序6-1 getpwnam函数

```
#include <sys/types.h>
#include <pwd.h>
#include <stddef.h>
#include <string.h>

struct passwd *
getpwnam(const char *name)
{
    struct passwd *ptr;

    setpwent();
    while ( (ptr = getpwent()) != NULL) {
        if (strcmp(name, ptr->pw_name) == 0)
            break;          /* found a match */
    }
    endpwent();
    return(ptr);          /* ptr is NULL if no match found */
}
```

在程序开始处调用 setpwent是保护性的措施，以便在调用者在此之前已经调用过 getpwent的情况下，反绕有关文件使它们定位到文件开始处。getpwnam和getpwuid完成后不应使有关文件仍处于打开状态，所以应调用endpwent关闭它们。

6.3 阴影口令

在上一节我们曾提及，对UNIX口令通常使用的加密算法是单向算法。给出一个密码口令，找不到一种算法可以将其反变换到普通文本口令(普通文本口令是在Password:提示后键入的口令)。但是可以对口令进行猜测，将猜测的口令经单向算法变换成加密形成，然后将其与用户的加密口令相比较。如果用户口令是随机选择的，那么这种方法并不是很有用。但是用户往往以非随机方式选择口令(配偶的姓名、街名、宠物名等)。一个经常重复的试验是先得到一份口令文件，然后用试探方法猜测口令。(Garfinkel和Spafford [1991]的第2章对UNIX口令及口令加密处理方案的历史情况及细节进行了说明。)

为使企图这样做的人难以获得原始资料(加密口令)，某些系统将加密口令存放在另一个通常称为阴影口令(shadow password)的文件中。该文件至少要包含用户名和加密口令。与该口令相关的其他信息也可存放在该文件中。例如，具有阴影口令的系统经常要求用户在一定时

间间隔后选择一个新口令。这被称之为口令时效，选择新口令的时间间隔长度经常也存放在阴影口令文件中。

在SVR4中，阴影口令文件是 /etc/shadow。在4.3+BSD中，加密口令存放在 /etc/master.passwd 中。

阴影口令文件不应是一般用户可以读取的。仅有少数几个程序需要存取加密口令文件，例如login(1)和passwd(1)，这些程序常常设置 -用户-ID为root。有了阴影口令后，普通口令文件 /etc/passwd可由各用户自由读取。

6.4 组文件

UNIX组文件(POSIX.1称其为组数据库)包含了表6-2中所示字段。这些字段包含在 <grp.h>中所定义的group结构中。

表6-2 /etc/group文件中的字段

说 明	struct group成员	POSIX.1
组名	char *gr_name	•
加密口令	char *gr_passwd	
数字组ID	int gr_gid	•
指向各用户名指针的数组	char **gr_mem	•

POSIX.1只定义了其中三个字段。另一个字段 gr_passwd则由SVR4和4.3+BSD支持。

字段gr_mem是一个指针数组，其中的指针各指向一个属于该组的用户名。该数组以 null结尾。

可以用下列两个由POSIX.1定义的函数来查看组名或数值组ID。

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

struct group *getgrgid(gid_t);

struct group *getgrnam(const char*);
```

两个函数返回：若成功则为指针，若出错则为 NULL

如同对口令文件进行操作的函数一样，这两个函数通常也返回指向一个静态变量的指针，在每次调用时都重写该静态变量。

如果需要搜索整个组文件，则须使用另外几个函数。下列三个函数类似于针对口令文件的三个函数。

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

```
struct group *getgrent(void);
```

返回：若成功则为指针，若出错或到达文件尾端则为 NULL

```
void setgrent(void);
```

```
void endgrent(void);
```

这三个函数由SVR4和4.3+BSD提供。POSIX.1并未定义这些函数。

setgrent打开组文件(如若它尚未被打开)并反绕它。getgrent从组文件中读下一个记录，如若该文件尚未打开则先打开它。endgrent关闭组文件。

6.5 添加组ID

在UNIX中，对组的使用已经作了些更改。在V7中，每个用户任何时候都只属于一个组。当用户登录时，系统就按与口令文件相关的数字组ID，赋给他实际组ID。可以在任何时候执行newgrp(1)以更改组ID。如果newgrp命令执行成功(关于许可权规则，请参阅手册)，则实际组ID就更改为新的组ID，它将被用于后续的文件存取许可权检查。执行不带任何参数的newgrp，则可返回到原来的组。

这种组的成员关系一直维持到1983年左右。此时，4.2BSD引入了添加组ID的概念。我们不仅可以属于口令记录中组ID所对应的组，也可属于多至16个另外的组。文件存取许可权检查相应被修改为：不仅将进程的有效组ID与文件的组ID相比较，而且也将所有添加组ID与文件的组ID进行比较。

添加组ID是POSIX.1的可选特性。常数NGROUPS_MAX(见表2-7)规定了添加组ID的数量，其常用值是16。如果不支持添加组ID，则此常数值为0。

SVR4和4.3+BSD都支持添加组ID。

FIPS 151-1要求支持添加组ID，并要求NGROUPS_MAX至少是8。

使用添加组ID的优点是不必再显式地经常更改组。一个用户常常会参加多个项目，因此也就要同时属于多个组。

为了存取和设置添加组ID提供了下列三个函数：

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

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

```
int getgroups(int gidsetsize, gid_t grouplist[]);
```

返回：若成功则为添加的组ID数，若出错则为-1

```
int setgroups(int ngroups, const gid_t grouplist[]);
```

```
int initgroups(const char *username, gid_t basegid);
```

两个函数返回：若成功则为0，若出错则为-1

getgroups将进程所属用户的各添加组ID填写到数组grouplist中，填写入该数组的添加组

ID数最多为 *gidsetsize* 个。实际填写到数组中的添加组 ID 数由函数返回。如果系统常数 *NGROUPS_MAX* 为 0，则返回 0，这并不表示出错。

POSIX.1 只说明了 *getgroups*。因为 *setgroups* 和 *initgroups* 是特权操作，所以 POSIX.1 没有说明它们。但是，SVR4 和 4.3+BSD 支持所有这三个函数。

作为一种特殊情况，如若 *gidsetsize* 为 0，则函数只返回添加组 ID 数，而对数组 *grouplist* 则不作修改。（这使调用者可以确定 *grouplist* 数组的长度，以便进行分配。）

setgroups 可由超级用户调用以便为调用进程设置添加组 ID 表。*grouplist* 是组 ID 数组，而 *ngroups* 说明了数组中的元素数。

通常，只有 *initgroups* 函数调用 *setgroups*，*initgroups* 读整个组文件（用函数 *getgrent*，*setgrent* 和 *endgrent*），然后对 *username* 确定其组的成员关系。然后，它调用 *setgroups*，以便为该用户初始化添加组 ID 表。因为 *initgroups* 调用 *setgroups*，所以只有超级用户才能调用 *initgroups*。除了在组文件中找到 *username* 是成员的组，*initgroups* 也在添加组 ID 表中包括了 *basegid*。*basegid* 是 *username* 在口令文件中的组 ID。

initgroups 只有少数几个程序调用，例如 *login(1)* 程序在用户登录时调用该函数。

6.6 其他数据文件

至此已讨论了两个系统数据文件——口令文件和组文件。在日常事务操作中，UNIX 系统还使用很多其他文件。例如，BSD 网络软件有一个记录各网络服务器所提供的服务的数据文件（*/etc/services*），有一个记录协议信息的数据文件（*/etc/protocols*），还有一个则是记录网络信息的数据文件（*/etc/networks*）。幸运的是，对于这些数据文件的界面都与上述对口令文件和组文件的相似。

一般情况下每个数据文件至少有三个函数：

(1) *get* 函数：读下一个记录，如果需要还打开该文件。此种函数通常返回指向一个结构的指针。当已达到文件尾端时返回空指针。大多数 *get* 函数返回指向一个静态存储类结构的指针，如果要保存其内容，则需复制它。

(2) *set* 函数：打开相应数据文件（如果尚未打开），然后反绕该文件。如果希望在相应文件起始处开始处理，则调用此函数。

(3) *end* 函数：关闭相应数据文件。正如前述，在结束了对相应数据文件的读、写操作后，总应调用此函数以关闭所有相关文件。

另外，如果数据文件支持某种形式的关键字搜索，则也提供搜索具有指定关键字的记录的例程。例如，对于口令文件提供了两个按关键字进行搜索的程序：*getpwnam* 寻找具有指定用户名的记录；*getpwuid* 寻找具有指定用户 ID 的记录。

表 6-3 中列出了一些这样的例程，这些都是 SVR4 和 4.3+BSD 所支持的。在表中列出了针对口令文件和组文件的函数，这些已在上面对说过。表中也列出了一些与网络有关的函数。表中列出的所有数据文件都有 *get*、*set* 和 *end* 函数。

在 SVR4 中，表 6-3 中最后四个数据文件都是符号连接，连接到目录 */etc/inet* 下的同名文件上。

SVR4 和 4.3+BSD 都有类似于表中所列的附加函数，但是这些附加函数都处理系统管理文件，专用于各个实现。

表6-3 存取系统数据文件的一些例程

说明	数据文件	头文件	结构	附加的关键字搜索函数
口令	/etc/passwd	<pwd.h>	passwd	getpwnam, getpwuid
组	/etc/group	<grp.h>	group	getgrnam, getgrgid
主机	/etc/hosts	<netdb.h>	hostent	gethostbyname, gethostbyaddr
网络	/etc/networks	<netdb.h>	netent	getnetbyname, getnetbyaddr
协议	/etc/protocols	<netdb.h>	protoent	getprotobyname, getprotobynumber
服务	/etc/services	<netdb.h>	servent	getservbyname, getservbyport

6.7 登录会计

大多数UNIX系统都提供下列两个数据文件：utmp文件，它记录当前登录进系统的各个用户；wtmp文件，它跟踪各个登录和注销事件。

在V7中，包含下列结构的一个二进制记录写入这两个文件中：

```
struct utmp {
    char    ut_line[8]; /* tty line: "ttyh0", "ttyd0", "ttyp0", ... */
    char    ut_name[8]; /* login name */
    long    ut_time; /* seconds since Epoch */
};
```

登录时，login程序填写这样一个结构，然后将其写入到utmp文件中，同时也将其添写到wtmp文件中。注销时，init进程将utmp文件中相应的记录擦除（每个字节都填以0），并将一个新记录添写到wtmp文件中。读wtmp文件中的该注销记录，其ut_name字段清除为0。在系统再启动时，以及更改系统时间和日期的前后，都在wtmp文件中添写特殊的记录项。who(1)程序读utmp文件，并以可读格式打印其内容。后来的UNIX版本提供last(1)命令，它读wtmp文件并打印所选择的记录。

大多数UNIX版本仍提供utmp和wtmp文件，但其中的信息量却增加了。V7中20字节的结构在SVR2中已扩充为36字节，而在SVR4中，utmp结构已扩充为350字节。

SVR4中这些记录的详细格式请参见手册页 utmp(4)和utmpx(4)。SVR4中这两个文件都在目录/var/adm中。SVR4提供了很多函数(见getut(3)和getutx(3))读或写这两个文件。

4.3+BSD中登录记录的格式请参见手册页 utmp(5)。这两个文件的路径名是/var/run/utmp和/var/log/wtmp。

6.8 系统标识

POSIX.1定义了uname函数，它返回与主机和操作系统有关的信息。

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

int uname(struct utsname *name);
```

通过该函数的参数向其传递一个 `utsname` 结构的地址，然后该函数填写此结构。POSIX.1 只定义了该结构中至少需提供的字段（它们都是字符数组），而每个数组的长度则由实现确定。某些实现在该结构中提供了另外一些字段。在历史上，系统 V 为每个数组分配 9 个字节，其中有 1 个字节用于字符串结束符（null 字符）。

```
struct utsname {
    char sysname[9]; /* name of the operating system */
    char nodename[9]; /* name of this node */
    char release[9]; /* current release of operating system */
    char version[9]; /* current version of this release */
    char machine[9]; /* name of hardware type */
};
```

`utsname` 结构中的信息通常可用 `uname(1)` 命令打印。

POSIX.1 警告 `nodename` 元素可能并不适用于在通信网络上引用主机。此函数来自于系统 V，早期 `nodename` 元素适用于在 UUCP 网络上引用主机。

也应理解在此结构中并没有给出有关 POSIX.1 版本的信息。这应当使用 2.5.2 节中所说明的 `_POSIX_VERSION` 以获得该信息。

最后，此函数给出了一种存取该结构中信息的方法，至于如何初始化这些信息，POSIX.1 没有作任何说明。大多数系统 V 版本在构造内核时通过编译将这些信息存放在内核中。

伯克利类的版本提供 `gethostname` 函数，它只返回主机名，该名字通常就是 TCP/IP 网络上主机的名字。

```
#include <unistd.h>

int gethostname(char *name, int namelen);
```

返回：若成功则为 0，若出错则为 -1

通过 `name` 返回的字符串以 null 结尾（除非没有提供足够的空间）。`<sys/param.h>` 中的常数 `MAXHOSTNAMELEN` 规定了此名字的最大长度（通常是 64 字节）。如果宿主机联接到 TCP/IP 网络中，则此主机名通常是该主机的完整域名。

`hostname(1)` 命令可用来存取和设置主机名。（超级用户用一个类似的函数 `sethostname` 来设置主机名。）主机名通常在系统自举时设置，它由 `/etc/rc` 取自一个启动文件。

虽然此函数是伯克利所特有的，但是，SVR4 作为 BSD 兼容性软件包的一部分提供 `gethostname` 和 `sethostname` 函数，以及 `hostname` 命令。SVR4 也将 `MAXHOSTNAMELEN` 扩充为 256 字节。

6.9 时间和日期例程

由 UNIX 内核提供的基本时间服务是国际标准时间公元 1970 年 1 月 1 日 00:00:00 以来经过的秒数。1.10 节中曾提及这种秒数是以数据类型 `time_t` 表示的。我们称它们为日历时间。日历时间

包括时间和日期。UNIX在这方面与其他操作系统的区别是：(a)以国际标准时间而非本地时间计时；(b)可自动进行转换，例如变换到夏日制；(c)将时间和日期作为一个量值保存。time函数返回当前时间和日期。

```
#include <time.h>
time_t time(time_t *calptr);
```

返回：若成功则为时间值，若出错则为 - 1

时间值作为函数值返回。如果参数非 null，则时间值也存放在由 *calptr* 指向的单元内。

在很多伯克利类的系统中，time(3)只是一个函数，它调用gettimeofday(2)系统调用。

在SVR4中调用stime(2)函数，在伯克利类的系统中调用settimeofday(2)对内核中的当前时间设置初始值。

与time和stime函数相比，BSD的gettimeofday和settimeofday提供了更高的分辨率(微秒级)。这对某些应用很重要。

一旦取得这种以秒计的很大的时间值后，通常要调用另一个时间函数将其变换为人们可读的时间和日期。图6-1说明了各种时间函数之间的关系。(图中以虚线表示的四个函数localtime、mktime、ctime和strftime都受到环境变量TZ的影响。我们将在本节的最后部分对其进行说明。)

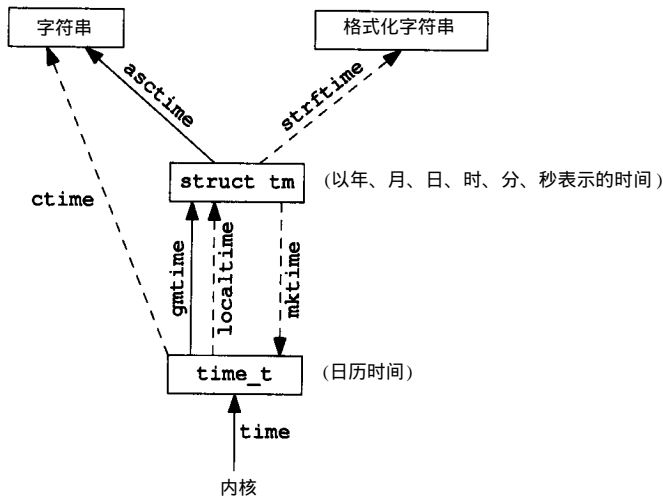


图6-1 各个时间函数之间的关系

两个函数localtime和gmtime将日历时间变换成以年、月、日、时、分、秒、周日表示的时间，并将这些存放在一个tm结构中。

```
struct tm { /* a broken-down time */
    int tm_sec; /* seconds after the minute: [0, 61] */
    int tm_min; /* minutes after the hour: [0, 59] */
    int tm_hour; /* hours after midnight: [0, 23] */
    int tm_mday; /* day of the month: [1, 31] */
    int tm_mon; /* month of the year: [0, 11] */
```

```

int  tm_year; /* years since 1900 */
int  tm_wday; /* days since Sunday: [0, 6] */
int  tm_yday; /* days since January 1: [0, 365] */
int  tm_isdst; /* daylight saving time flag: <0, 0, >0 */
};

```

秒可以超过 59 的理由是可以表示闰秒。注意，除了月日字段，其他字段的值都以 0 开始。如果夏时制生效，则夏时制标志值为正；如果已非夏时制时间则为 0；如果此信息不可用，则为负。

```

#include <time.h>

struct tm *gmtime(const time_t *calptr);

struct tm *localtime(const time_t *calptr);

```

两个函数返回：指向 tm 结构的指针

localtime 和 gmtime 之间的区别是：localtime 将日历时间变换成本地时间（考虑到本地时区和夏时制标志），而 gmtime 则将日历时间变换成国际标准时间的年、月、日、时、分、秒、周日。

函数 mktime 以本地时间的年、月、日等作为参数，将其变换成 time_t 值。

```

#include <time.h>

time_t mktime(struct tm *tm_ptr);

```

返回：若成功则为日历时间，若出错则为 -1

asctime 和 ctime 函数产生形式的 26 字节字符串，这与 date(1) 命令的系统默认输出形式类似：

```
Tue Jan 14 17:49:03 1992\n\0
```

```

#include <time.h>

char *asctime(const struct tm *tm_ptr);

char *ctime(const time_t *calptr);

```

两个函数返回：指向 null 结尾的字符串

asctime 的参数是指向年、月、日等字符串的指针，而 ctime 的参数则是指向日历时间的指针。

最后一个时间函数是 strftime，它是非常复杂的 printf 类的时间值函数。

```

#include <time.h>

size_t strftime(char *buf, size_t maxsize, const char * format,
                const struct tm *tm_ptr);

```

返回：若有空间，则存入数组的字符数，否则为 0

最后一个参数是要格式化的时间值，由一个指向一个年、月、日、时、分、秒、周日时间值的指针说明。格式化结果存放在一个长度为 maxsize 个字符的 buf 数组中，如果 buf 长度足以存放格式化结果及一个 null 终止符，则该函数返回在 buf 中存放的字符数（不包括 null 终止符），否则该

函数返回0。

*format*参数控制时间值的格式。如同printf函数一样，变换说明的形式是百分号之后跟一个特定字符。*format*中的其他字符则按原样输出。两个连续的百分号在输出中产生一个百分号。与printf函数的不同之处是，每个变换说明产生一个定长输出字符串，在*format*字符串中没有字段宽度修饰符。表6-4中列出了21种ANSI C规定的变换说明。

表6-4 strftime

格 式	说 明	例 子
%a	缩写的周日名	Tue
%A	全周日名	Tuesday
%b	缩写的月名	Jan
%B	月全名	January
%c	日期和时间	Tue Jan 14 19:40:30 1992
%d	月日：[01, 31]	14
%H	小时（每天24小时）：[00, 23]	19
%I	小时（上、下午各12小时）：[01, 12]	07
%j	年日：[001, 366]	014
%m	月：[01, 12]	01
%M	分：[00, 59]	40
%P	AM/PM	PM
%S	秒：[00, 61]	30
%U	星期日周数：[00, 53]	02
%w	周日：[0=星期日, 6]	2
%W	星期一一周数：[00, 53]	02
%x	日期	01/14/92
%X	时间	19:40:30
%y	不带公元的年：[00, 991]	92
%Y	带公元的年	1992
%Z	时区名	MST

表中第三列的数据来自于在SVR4，对应于下列时间和日期，执行strftime函数所得的结果为：

```
Tue Jan 14 19:40:30 MST 1992
```

表6-4中的大多数格式说明的意义很明显。需要略作解释的是%U和%W。%U是相应日期在该年中所属周数，包含该年中第一个星期日的周是第一周。%W也是相应日期在该年中所属的周数，不同的是包含第一个星期一的周为第一周。

SVR4和4.3+BSD都对strftime的格式字符串提供另一些扩充支持。

我们曾在前面提及，图6-1中以虚线表示的四个函数受到环境变量TZ的影响。这四个函数是：localtime、mktime、ctime和strftime。如果定义了TZ，则这些函数将使用其值以代替系统默认时区。如果TZ定义为空串（亦即TZ=），则使用国际标准时间。TZ的值常常类似于：TZ=EST5EDT，但是POSIX.1允许更详细的说明。有关TZ的详细情况请参阅POSIX.1标准 [IEEE 1990] 的8.1.1节，

SVR4 environ(5)手册页 [AT&T 1990e], 或4.3+BSD ctime(3)手册页。

本节所述的所有时间和日期函数都是由 ANSI C 标准定义的。在此基础上，POSIX.1 只增加了环境变量 TZ。

图6-1中七个函数中的五个可以回溯到 V7 (或更早)，它们是：time、localtime、gmtime、asctime和ctime。在UNIX时间功能方面近期所作的增加大多是处理非美国时区以及夏时制的更改规则。

6.10 小结

在所有UNIX系统上都使用口令文件和组文件。我们说明了各种读这些文件的函数。也介绍了阴影口令，它可以增加系统的安全性。添加组 ID 正在得到更多应用，它提供了一个用户同时可以参加多个组的方法。还介绍了大多数系统所提供的存取其他与系统有关的数据文件的类似的函数。最后，说明了ANSI C和POSIX.1提供的与时间和日期有关的一些函数。

习题

- 6.1 如果系统使用阴影文件，如何取得加密口令？
- 6.2 假设你有超级用户许可权，并且系统使用了阴影口令，重新考虑上一道习题。
- 6.3 编程调用uname并输出utsname结构中的所有字段，并与uname(1)命令的输出结果作比较。
- 6.4 编程获取当前时间并使用strftime将输出结果转换为类似于date(1)命令的缺省输出。修改环境变量TZ的值，观察输出的结果。

第7章 UNIX进程的环境

7.1 引言

下一章将介绍进程控制原语，在此之前需先了解进程的环境。本章中将学习：当执行程序时，其main函数是如何被调用的，命令行参数是如何传送给执行程序的；典型的存储器布局是什么样式；如何分配另外的存储空间；进程如何使用环境变量；进程终止的不同方式等。另外，还将说明longjmp和setjmp函数以及它们与栈的交互作用。本章结束之前，还将查看进程的资源限制。

7.2 main函数

C程序总是从main函数开始执行。main函数的原型是：

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

其中，*argc*是命令行参数的数目，*argv*是指向参数的各个指针所构成的数组。7.4节将对命令行参数进行说明。

当内核启动C程序时(使用一个exec函数，8.9节将说明exec函数)，在调用main前先调用一个特殊的起动例程。可执行程序文件将此起动例程指定为程序的起始地址——这是由连接编辑程序设置的，而连接编辑程序则由C编译程序(通常是cc)调用。起动例程从内核取得命令行参数和环境变量值，然后为调用main函数作好安排。

7.3 进程终止

有五种方式使进程终止：

(1) 正常终止：

- (a) 从main返回。
- (b) 调用exit。
- (c) 调用_exit。

(2) 异常终止：

- (a) 调用abort(见第10章)。
- (b) 由一个信号终止(见第10章)。

上节提及的起动例程是这样编写的，使得从main返回后立即调用exit函数。如果将起动例程以C代码形式表示(实际上该例程常常用汇编语言编写)，则它调用main函数的形式可能是：

```
exit( main(argc, argv) );
```

7.3.1 exit和_exit函数

exit和_exit函数用于正常终止一个程序：_exit立即进入内核，exit则先执行一些清除处理(包括调用执行各终止处理程序，关闭所有标准I/O流等)，然后进入内核。

```
#include <stdlib.h>

void exit(int status);

#include <unistd.h>

void _exit (int status);
```

我们将在8.5节中讨论这两个函数对其他进程，例如终止进程的父、子进程的影响。

使用不同头文件的原因是：`exit`是由ANSI C说明的，而`_exit`则是由POSIX.1说明的。

由于历史原因，`exit`函数总是执行一个标准 I/O 库的清除关闭操作：对于所有打开流调用 `fclose` 函数。回忆 5.5 节，这造成缓存中的所有数据都被刷新(写到文件上)。

`exit`和`_exit`都带一个整型参数，称之为终止状态 (exit status)。大多数 UNIX shell 都提供检查一个进程终止状态的方法。如果 (a) 若调用这些函数时不带终止状态，或 (b) `main` 执行了一个无返回值的 `return` 语句，或 (c) `main` 执行隐式返回，则该进程的终止状态是未定义的。这就意味着，下列经典性的 C 语言程序：

```
#include <stdio.h>
main ()
{
    printf ("hello, world \n");
}
```

是不完整的，因为 `main` 函数没有使用 `return` 语句返回(隐式返回)，它在返回到 C 的起动例程时并没有返回一个值(终止状态)。另外，若使用：

```
return(0);
```

或者

```
exit(0);
```

则向执行此程序的进程(常常是一个 shell 进程)返回终止状态 0。另外，`main` 函数的说明实际上应当是：

```
int main(void)
```

下一章将了解进程如何使程序执行，如何等待执行该程序的进程完成，然后取得其终止状态。

将 `main` 说明为返回一个整型以及用 `exit` 代替 `return`，对某些 C 编译程序和 UNIX `lint(1)` 程序而言会产生不必要的警告信息，因为这些编译程序并不了解 `main` 中的 `exit` 与 `return` 语句的作用相同。警告信息可能是“control reaches end of nonvoid function (控制到达非 void 函数的结束处)”。避开这种警告信息的一种方法是：在 `main` 中使用 `return` 语句而不是 `exit`。但是这样做的结果是不能用 UNIX 的 `grep` 公用程序来找出程序中所有的 `exit` 调用。另外一个解决方法是将 `main` 说明为返回 `void` 而不是 `int`，然后仍旧调用 `exit`。这也避开了编译程序的警告，但从程序设计角度看却并不正确。本章将 `main` 表示为返回一个整型，因为这是 ANSI C 和 POSIX.1 所定义的。我们将不理睬编译程序不必要的警告。

7.3.2 atexit函数

按照ANSI C的规定，一个进程可以登记多至32个函数，这些函数将由exit自动调用。我们称这些函数为终止处理程序（exit handler），并用atexit函数来登记这些函数。

```
#include <stdlib.h>

int atexit(void (*func)(void));
```

返回：若成功则为0，若出错则为非0

其中，atexit的参数是一个函数地址，当调用此函数时无需向它传送任何参数，也不期望它返回一个值。exit以登记这些函数的相反顺序调用它们。同一函数如若登记多次，则也被调用多次。

终止处理程序这一机制由ANSI C最新引进。SVR4和4.3+BSD都提供这种机制。系统V的早期版本和4.3BSD则都不提供此机制。

根据ANSI C和POSIX.1，exit首先调用各终止处理程序，然后按需多次调用fclose，关闭所有打开流。图7-1显示了一个C程序是如何起动的，以及它终止的各种方式。

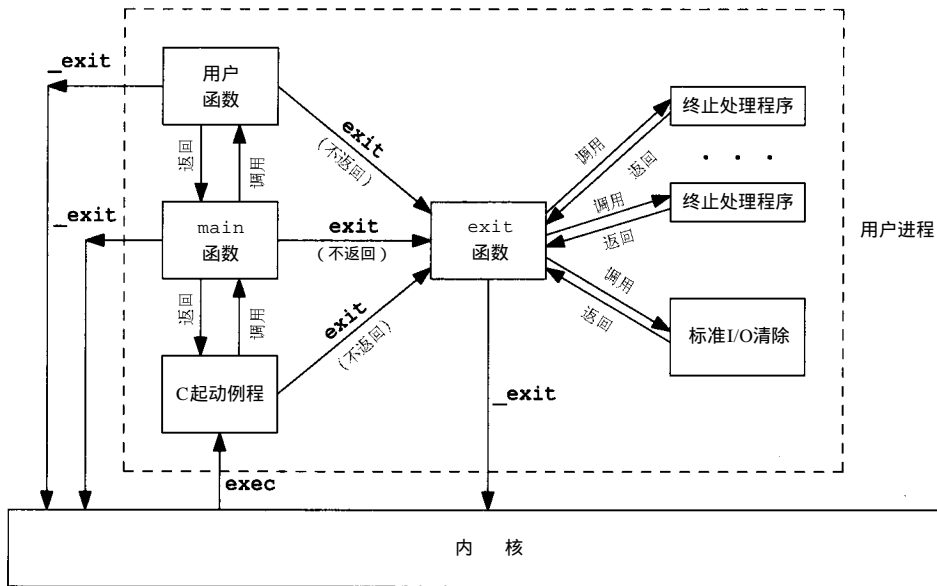


图7-1 一个C程序是如何起动和终止的

注意，内核使程序执行的唯一方法是调用一个exec函数。进程自愿终止的唯一方法是显式或隐式地(调用exit)调用_exit。进程也可非自愿地由一个信号使其终止(图7-1中没有显示)。

实例

程序7-1说明了如何使用atexit函数。执行程序7-1产生：

```
$ a.out
main is done
```

```
first exit handler
first exit handler
second exit handler
```

注意，在main中没有调用exit，而是用了return语句。

程序7-1 终止处理程序实例

```
#include "ourhdr.h"
static void my_exit1(void), my_exit2(void);
int
main(void)
{
    if (atexit(my_exit2) != 0)
        err_sys("can't register my_exit2");

    if (atexit(my_exit1) != 0)
        err_sys("can't register my_exit1");
    if (atexit(my_exit1) != 0)
        err_sys("can't register my_exit1");

    printf("main is done\n");
    return(0);
}

static void
my_exit1(void)
{
    printf("first exit handler\n");
}

static void
my_exit2(void)
{
    printf("second exit handler\n");
}
```

7.4 命令行参数

当执行一个程序时，调用exec的进程可将命令行参数传递给该新程序。这是UNIX shell的一部分常规操作。在前几章的很多实例中，我们已经看到了这一点。

实例

程序7-2将其所有命令行参数都回送到标准输出上（UNIX echo(1)程序不回送第0个参数）。编译此程序，并将其可执行代码文件定名为echoarg，则：

```
$ ./echoarg arg1 TEST foo
argv[0]: ./echoarg
argv[1]: arg1
argv[2]: TEST
argv[3]: foo
```

程序7-2 将所有命令行参数回送到标准输出

```
#include "ourhdr.h"
int
main(int argc, char *argv[])
```

```

{
    int i;
    for (i = 0; i < argc; i++) /* echo all command-line args */
        printf("argv[%d]: %s\n", i, argv[i]);
    exit(0);
}

```

ANSI C和POSIX.1都要求`argv[argc]`是一个空指针。这就使我们可以将参数处理循环改写为：

```
for(i = 0; argv[i] != NULL; i++)
```

7.5 环境表

每个程序都接收到一张环境表。与参数表一样，环境表也是一个字符指针数组，其中每个指针包含一个以`null`结束的字符串的地址。全局变量`environ`则包含了该指针数组的地址。

```
extern char **environ;
```

例如：如果该环境包含五个字符串，那么它看起来可能如图 7-2 中所示。

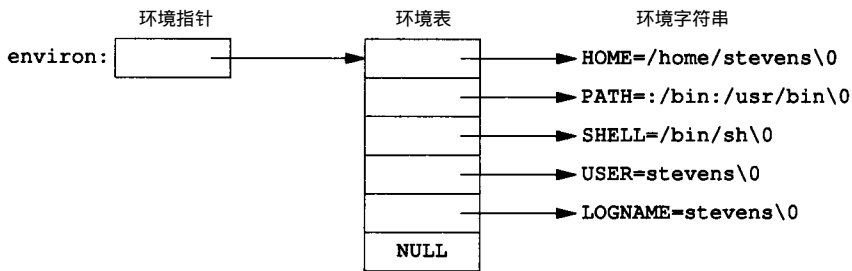


图7-2 由五个字符串组成的环境

其中，每个字符串的结束处都有一个`null`字符。我们称`environ`为环境指针，指针数组为环境表，其中各指针指向的字符串为环境字符串。

按照惯例，环境由：

```
name=value
```

这样的字符串组成，这与图 7-2 中所示相同。大多数预定义名完全由大写字母组成，但这只是一个惯例。

在历史上，大多数 UNIX 系统对`main`函数提供了第三个参数，它就是环境表地址：

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

因为 ANSI C 规定`main`函数只有两个参数，而且第三个参数与全局变量`environ`相比也没有带来更多益处，所以 POSIX.1 也规定应使用`environ`而不使用第三个参数。通常用`getenv`和`putenv`函数(7.9 节将说明)来存取特定的环境变量，而不是用`environ`变量。但是，如果要查看整个环境，则必须使用`environ`指针。

7.6 C程序的存储空间布局

由于历史原因，C 程序一直由下列几部分组成：

- 正文段。这是由 CPU 执行的机器指令部分。通常，正文段是可共享的，所以即使是经常

执行的程序(如文本编辑程序、C编译程序、shell等)在存储器中也只需有一个副本,另外,正文段常常是只读的,以防止程序由于意外事故而修改其自身的指令。

- 初始化数据段。通常将此段称为数据段,它包含了程序中需赋初值的变量。例如, C程序中任何函数之外的说明:

```
int    maxcount = 99;
```

使此变量以初值存放在初始化数据段中。

- 非初始化数据段。通常将此段称为 bss段,这一名称来源于早期汇编程序的一个操作符,意思是“block started by symbol(由符号开始的块)”,在程序开始执行之前,内核将此段初始化为0。函数外的说明:

```
long   sum[1000];
```

使此变量存放在非初始化数据段中。

- 栈。自动变量以及每次函数调用时所需保存的信息都存放在此段中。每次函数调用时,其返回地址、以及调用者的环境信息(例如某些机器寄存器)都存放在栈中。然后,新被调用的函数在栈上为其自动和临时变量分配存储空间。通过以这种方式使用栈, C函数可以递归调用。

- 堆。通常在堆中进行动态存储分配。由于历史上形成的惯例,堆位于非初始化数据段顶和栈底之间。

图7-3显示了这些段的一种典型安排方式。这是程序的逻辑布局——虽然并不要求一个具体实现一定以这种方式安排其存储空间。尽管如此,这给出了一个我们便于作有关说明的一种典型安排。

对于VAX上的4.3+BSD,正文段从0位置开始,栈顶则在0x7fffffff之下开始。在VAX机器上,堆顶和栈底之间未用的虚地址空间很大。

从图7-3还可注意到未初始化数据段的内容并不存放在磁盘程序文件中。需要存放在磁盘程序文件中的段只有正文段和初始化数据段。

size(1)命令报告正文段、数据段和 bss段的长度(单位:字节)。例如:

```
$ size /bin/cc /bin/sh
text    data    bss     dec      hex
81920   16384   664     98968    18298   /bin/cc
90112   16384   0       106496   1a000   /bin/sh
```

第4和第5列是分别以十进制和十六进制表示的总长度。

7.7 共享库

现在,很多UNIX系统支持共享库。Arnold [1986] 说明了系统V上共享库的一个早期实现, Gingell等 [1987] 则说明了SunOS上的另一个实现。共享库使得可执行文件中不再需要包含常用的库函数,而只需在所有进程都可存取的存储区中保存这种库例程的一个副本。程序第一次执行或

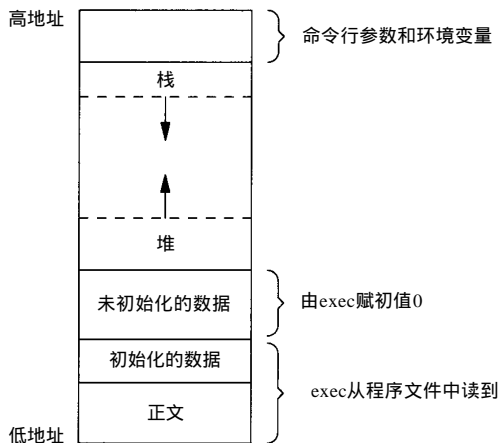


图7-3 典型的存储器安排

者第一次调用某个库函数时，用动态连接方法将程序与共享库函数相连接。这减少了每个可执行文件的长度，但增加了一些运行时间开销。共享库的另一个优点是可以库函数的新版本代替老版本而无需对使用该库的程序重新连接编辑。（假定参数的数目和类型都没有发生改变。）

不同的系统使用不同的方法使说明程序是否要使用共享库。比较典型的有 `cc(1)`和`ld(1)`命令的可选项。作为长度方面发生变化的例子，下列可执行文件（典型的`hello.c`程序）先用无共享库方式创建：

```
$ ls -l a.out
-rwxrwxr-x  1 stevens      104859 Aug  :23 a4out
$ size a.out
text      data      bss      dec      hex
49152     49152     0        98304    18000
```

如果我们再编译此程序使其使用共享库，则可执行文件的正文和数据段的长度都显著减小：

```
$ ls -l a.out
-rwxrwxr-x  1 stevens      24576 Aug  :23 a4out
$ size a.out
text      data      bss      dec      hex
8192      8192     0        16384    4000
```

7.8 存储器分配

ANSI C说明了三个用于存储空间动态分配的函数。

(1) `malloc`。分配指定字节数的存储区。此存储区中的初始值不确定。

(2) `calloc`。为指定长度的对象，分配能容纳其指定个数的存储空间。该空间中的每一位(bit)都初始化为0。

(3) `realloc`。更改以前分配区的长度(增加或减少)。当增加长度时，可能需将以前分配区的内容移到另一个足够大的区域，而新增区域内的初始值则不确定。

```
#include <stdlib.h>

void *malloc(size_t size);

void *calloc(size_t nobj, size_t size);

void *realloc(void *ptr, size_t newsize);
```

三个函数返回：若成功则为非空指针，若出错则为 `NULL`

```
void free(void *ptr)
```

这三个分配函数所返回的指针一定是适当对齐的，使其可用于任何数据对象。例如，在一个特定的系统上，如果最苛刻的对齐要求是 `double`，则对齐必须在8的倍数的地址单元处，那么这三个函数返回的指针都应这样对齐。

回忆1.6节中对类属`void *`指针和函数原型的讨论。因为这三个 `alloc`函数都返回类属指针，如果在程序中包括了`<stdlib.h>`(包含了函数原型)，那么当我们将这些函数返回的指针赋与一个不同类型的指针时，不需要作类型强制转换。

函数`free`释放`ptr`指向的存储空间。被释放的空间通常被送入可用存储区池，以后可在调用

分配函数时再分配。

`realloc`使我们可以增、减以前分配区的长度(最常见的用法是增加该区)。例如,如果先分配一个可容纳长度为512的数组的空间,并在运行时填充它,但又发现空间不够,则可调用`realloc`扩充该存储空间。如果在该存储区后有足够的空间可供扩充,则可在原存储区位置上向高地址方向扩充,并返回传送给它的同样的指针值。如果在原存储区后没有足够的空间,则`realloc`分配另一个足够大的存储区,将现存的512个元素数组的内容复制到新分配的存储区。因为这种存储区可能会移动位置,所以不应当使用任何指针指在该区中。习题4.18显示了在`getcwd`中如何使用`realloc`,以处理任何长度的路径名。程序15-27是使用`realloc`的另一个例子,用其可以避免使用编译时固定长度的数组。

注意,`realloc`的最后一个参数是存储区的`newsize`(新长度),不是新、旧长度之差。作为一个特例,若`ptr`是一个空指针,则`realloc`的功能与`malloc`相同,用于分配一个指定长度`newsize`的存储区。

此功能是由ANSI C新引进的。如果传送一个null指针,`realloc`的早期版本会失败。早期版本允许`realloc`自上次`malloc`,`realloc`或`calloc`以来所释放的块。这种技巧可回溯到V7,它利用`malloc`实现存储器紧缩的搜索策略。4.3+BSD仍支持这一功能,而SVR4则不支持。这种功能不应再使用。

这些分配例程通常通过`sbrk(2)`系统调用实现。该系统调用扩充(或缩小)进程的堆(见图7-3)。`malloc`和`free`的一个样本实现请见Kernighan和Ritchie [1988]的8.7节。

虽然`sbrk`可以扩充或缩小一个进程的存储空间,但是大多数`malloc`和`free`的实现都不减小进程的存储空间。释放的空间可供以后再分配,但将它们保持在`malloc`池中而不返回给内核。

应当注意的是,大多数实现所分配的存储空间比所要求的要稍大一些,额外的空间用来记录管理信息——分配块的长度,指向下一个分配块的指针等等。这就意味着如果写过一个已分配区的尾端,则会改写后一块的管理信息。这种类型的错误是灾难性的,但是因为这种错误不会很快就暴露出来,所以也就很难发现。将指向分配块的指针向后移动也可能会改写本块的管理信息。

其他可能产生的致命性的错误是:释放一个已经释放了的块;调用`free`时所用的指针不是三个`alloc`函数的返回值等。

因为存储器分配出错很难跟踪,所以某些系统提供了这些函数的另一种实现方法。每次调用这三个分配函数中的任意一个或`free`时都进行附加的出错检验。在调用连接编辑程序时指定一个专用库,则在程序中就可使用这种版本的函数。此外还有公共可用的资源(例如由4.3+BSD所提供的),在对其进行编译时使用一个特殊标志就会使附加的运行时间检查生效。

因为存储空间分配程序的操作对某些应用程序的运行时间性能非常重要,所以某些系统提供了附加能力。例如,SVR4提供了名为`mallopt`的函数,它使进程可以设置一些变量,并用它们来控制存储空间分配程序的操作。还可使用另一个名为`mallinfo`的函数,以对存储空间分配程序的操作进行统计。请查看所使用系统的`malloc(3)`手册页,弄清楚这些功能是否可用。

`alloca`函数

还有一个函数也值得一提,这就是`alloca`。其调用序列与`malloc`相同,但是它是在当前函数的栈帧上分配存储空间,而不是在堆中。其优点是:当函数返回时,自动释放它所使用的

栈帧，所以不必再为释放空间而费心。其缺点是：某些系统在函数已被调用后不能增加栈帧长度，于是也就不能支持 `alloca` 函数。尽管如此，很多软件包还是使用 `alloca` 函数，也有很多系统支持它。

7.9 环境变量

如同前述，环境字符串的形式是：

```
name=value
```

UNIX内核并不关心这种字符串的意义，它们的解释完全取决于各个应用程序。例如，shell使用了大量的环境变量。其中某一些在登录时自动设置（如 `HOME`，`USER`等），有些则由用户设置。我们通常在一个shell启动文件中设置环境变量以控制shell的动作。例如，若设置了环境变量 `MAILPATH`，则它告诉Bourne shell和KornShell到哪里去查看邮件。

ANSI C定义了一个函数 `getenv`，可以用其取环境变量值，但是该标准又称环境的内容是由实现定义的。

```
#include <stdlib.h>

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

返回：指向与 `name` 关联的 `value` 的指针，若未找到则为 `NULL`

注意，此函数返回一个指针，它指向 `name=value` 字符串中的 `value`。我们应当使用 `getenv` 从环境中取一个环境变量的值，而不是直接存取 `environ`。

POSIX.1和XPG3定义了某些环境变量。表 7-1列出了由这两个标准定义并受到 SVR4和4.3+BSD支持的环境变量。SVR4和4.3+BSD还使用了很多依赖于实现的环境变量。

FIPS 151-1要求登录shell必须要定义环境变量 `HOME` 和 `LOGNAME`。

表7-1 环境变量

变 量	标 准		实 现		说 明
	POSIX.1	XPG3	SVR4	4.3+BSD	
<code>HOME</code>	•	•	•	•	起始目录
<code>LANG</code>	•	•	•		本地名
<code>LC_ALL</code>	•	•	•		本地名
<code>LC_COLLATE</code>	•	•	•		本地排序名
<code>LC_CTYPE</code>	•	•	•		本地字符分类名
<code>LC_MONETARY</code>	•	•	•		本地货币编辑名
<code>LC_NUMERIC</code>	•	•	•		本地数字编辑名
<code>LC_TIME</code>	•	•	•		本地日期/时间格式名
<code>LOGNAME</code>	•	•	•	•	登录名
<code>NLS_PATH</code>		•	•		消息类模板序列
<code>PATH</code>	•	•	•	•	搜索可执行文件的路径前缀表
<code>TERM</code>	•	•	•	•	终端类型
<code>TZ</code>	•	•	•	•	时区信息

除了取环境变量值，有时也需要设置环境变量，或者是改变现有变量的值，或者是增加新的环境变量。（在下一章将会了解到，我们能影响的是当前进程及其后生成的子进程的环境，但不能影响父进程的环境，这通常是一个 shell 进程。尽管如此，修改环境表的能力仍然是很有用的。）不幸的是，并不是所有系统都支持这种能力。表 7-2 列出了由不同的标准及实现支持的各种函数。

表7-2 对于各种环境表函数的支持

函 数	标 准			实 现	
	ANSI C	POSIX.1	XPG3	SVR4	4.3+BSD
getenv	•	•	•	•	•
putenv		(可能)	•	•	•
setenv					•
unsetenv					•
clearenv		(可能)			

POSIX.1标准说明putenv和clearenv正被考虑加到POSIX.1的修订版中。

在表7-2中，中间三个函数的原型是：

```
#include <stdlib.h>

int putenv(const char *str);

int setenv(const char *name, const char *value, int rewrite);

void unsetenv(const char *name);
```

两个函数返回：若成功则为0，若出错则为非0

这三个函数的操作是：

- putenv取形式为 *name=value* 的字符串，将其放到环境表中。如果 *name* 已经存在，则先删除其原来的定义。

- setenv将 *name* 设置为 *value*。如果在环境中 *name* 已经存在，那么(a)若 *rewrite* 非0，则首先删除其现存的定义；(b)若 *rewrite* 为0，则不删除其现存定义（*name* 不设置为新的 *value*，而且也不出错）。

- unsetenv删除 *name* 的定义。即使不存在这种定义也不算出错。

这些函数在修改环境表时是如何进行操作的呢？回忆一下图 7-3，其中，环境表（指向实际 *name=value* 字符串的指针数组）和环境字符串典型地存放在进程存储空间的顶部（栈之上）。删除一个字符串很简单——只要先找到该指针，然后将所有后续指针都向下移一个位置。但是增加一个字符串或修改一个现存的字符串就比较困难。栈以上的空间因为已处于进程存储空间的顶部，所以无法扩充，即无法向上扩充，也无法向下扩充。

(1) 如果修改一个现存的 *name*:

(a) 如果新 *value* 的长度少于或等于现存 *value* 的长度，则只要在原字符串所用空间中写入新字符串。

- (b) 如果新 *value* 的长度大于原长度，则必须调用 `malloc` 为新字符串分配空间，然后将新字符写入该空间中，然后使环境表中针对 *name* 的指针指向新分配区。
- (2) 如果要增加一个新的 *name*，则操作就更加复杂。首先，调用 `malloc` 为 *name=value* 分配空间，然后将该字符串写入此空间中。
- (a) 然后，如果这是第一次增加一个新 *name*，则必须调用 `malloc` 为新的指针表分配空间。将原来的环境表复制到新分配区，并将指向新 *name=value* 的指针存在该指针表的表尾，然后又将一个空指针存在其后。最后使 `environ` 指向新指针表。再看一下图 7-3，如果原来的环境表位于栈顶之上（这是一种常见情况），那么必须将此表移至堆中。但是，此表中的大多数指针仍指向栈顶之上的各 *name=value* 字符串。
- (b) 如果这不是第一次增加一个新 *name*，则可知以前已调用 `malloc` 在堆中为环境表分配了空间，所以只要调用 `realloc`，以分配比原空间多存放一个指针的空间。然后将该指向新 *name=value* 字符串的指针存放在该表表尾，后面跟着一个空指针。

7.10 `setjmp`和`longjmp`函数

在C中，不允许使用跳越函数的 `goto` 语句。而执行这种跳转功能的是函数 `setjmp` 和 `longjmp`。这两个函数对于处理发生在很深的嵌套函数调用中的出错情况非常有用。

考虑一下程序 7-3 的骨干部分。其主循环是从标准输入读 1 行，然后调用 `do_line` 处理每一输入行。该函数然后调用 `get_token` 从该输入行中取下一个记号。一行中的第一个记号假定是某种形式的一条命令，于是 `switch` 语句就实现命令选择。我们的程序只处理一条命令，对此命令调用 `cmd_add` 函数。

程序 7-3 进行命令处理的典型程序的骨干部分

```
#include "ourhdr.h"

#define TOK_ADD 5

void do_line(char *);
void cmd_add(void);
int get_token(void);

int
main(void)
{
    char line[MAXLINE];

    while (fgets(line, MAXLINE, stdin) != NULL)
        do_line(line);
    exit(0);
}

char *tok_ptr; /* global pointer for get_token() */

void
do_line(char *ptr) /* process one line of input */
{
    int cmd;

    tok_ptr = ptr;
    while ( (cmd = get_token()) > 0) {
        switch (cmd) { /* one case for each command */
            case TOK_ADD:
                cmd_add();
        }
    }
}
```

```

        break;
    }
}

void
cmd_add(void)
{
    int    token;

    token = get_token();
    /* rest of processing for this command */
}

int
get_token(void)
{
    /* fetch next token from line pointed to by tok_ptr */
}

```

程序7-3在读命令、确定命令的类型、然后调用相应函数处理每一条命令这类程序中是非常典型的。图7-4显示了调用了cmd_add之后栈的大致使用情况。

自动变量的存储单元在每个函数的栈帧中。数组line在main的栈帧中，整型cmd在do_line的栈帧中，整型token在cmd_add的栈帧中。

如上所述，这种形式的栈安排是非常典型的，但并不要求非如此不可。栈并不一定要向低地址方向扩充。某些系统对栈并没有提供特殊的硬件支持，此时一个C实现可能要用连接表实现栈帧。

在编写如程序7-3这样的程序中经常会遇到的一个问题是，如何处理非致命性的错误。例如，若cmd_add函数发现一个错误，譬如说一个无效的数，那么它可能先打印一个出错消息，然后希望忽略输入行的余下部分，返回main函数并读下一输入行。用C语言比较难做到这一点。（在本例中，cmd_add函数只比main低两个层次，在有些程序中往往低五或更多层次。）如果我们不得不以检查返回值的方法逐层返回，那就会变得很麻烦。

解决这种问题的方法就是使用非局部跳转——setjmp和longjmp函数。非局部表示这不是在一个函数内的普通的C语言goto语句，而是在栈上跳过若干调用帧，返回到当前函数调用路径上的一个函数中。

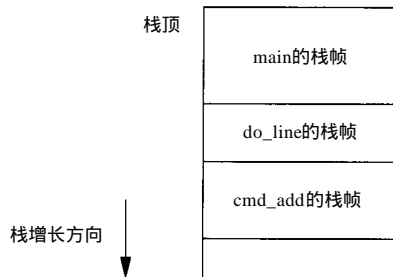


图7-4 调用cmd_add后的各个栈帧

```

#include <setjmp.h>

int setjmp(jmp_buf env);

```

返回：若直接调用则为0，若从longjmp返回则为非0

```

void longjmp(jmp_buf env, int val);

```

在希望返回到的位置调用setjmp，在本例中，此位置在main函数中。因为我们直接调用该函数，所以其返回值为0。setjmp的参数env是一个特殊类型jmp_buf。这一数据类型是某种形式的数组，其中存放在调用longjmp时能用来恢复栈状态的所有信息。一般，env变量是个全局变量，因为需从另一个函数中引用它。

当检查到一个错误时，例如在cmd_add函数中，则以两个参数调用longjmp函数。第一个就是在调用setjmp时所用的env；第二个val，是个非0值，它成为从setjmp处返回的值。使用第二个参数的原因是对于一个setjmp可以有多个longjmp。例如，可以在cmd_add中以val为1调用longjmp，也可在get_token中以val为2调用longjmp。在main函数中，setjmp的返回值就会是1或2，通过测试返回值就可判断是从cmd_add还是从get_token来的longjmp。

再回到程序实例中，表 7-1 中给出了经修改过后的 main和cmd_add函数(其他两个函数，do_line和get_token未经更改)。

程序7-4 setjmp和longjmp实例

```
#include <setjmp.h>
#include "ourhdr.h"

#define TOK_ADD 5

jmp_buf jmpbuffer;

int
main(void)
{
    char line[MAXLINE];

    if (setjmp(jmpbuffer) != 0)
        printf("error");
    while (fgets(line, MAXLINE, stdin) != NULL)
        do_line(line);
    exit(0);
}

. . .

void
cmd_add(void)
{
    int token;

    token = get_token();
    if (token < 0) /* an error has occurred */
        longjmp(jmpbuffer, 1);
    /* rest of processing for this command */
}
```

执行main时，调用setjmp，它将所需的信息记入变量jmpbuffer中并返回0。然后调用do_line，它又调用cmd_add，假定在其中检测到一个错误。在cmd_add中调用longjmp之前，栈的形式如图7-4中所示。但是longjmp使栈反绕到执行main函数时的情况，也就是抛弃了cmd_add和do_line的栈帧。调用longjmp造成main中setjmp的返回，但是，这一次的返回值是1(longjmp的第二个参数)。

7.10.1 自动、寄存器和易失变量

下一个问题是：“在main函数中，自动变量和寄存器变量的状态如何？”当longjmp返回到main函数时，这些变量的值是否能恢复到以前调用setjmp时的值（即滚回原先值），或者这些变量的值保持为调用do_line时的值(do_line调用cmd_add，cmd_add又调用longjmp)?不幸的是，对此问题的回答是“看情况”。大多数实现并不滚回这些自动变量和寄存器变量的值，而所有标准则说它们的值是不确定的。如果你有一个自动变量，而又不想使其值滚回，则可定义其为

具有volatile属性。说明为全局和静态变量的值在执行 longjmp时保持不变。

实例

下面我们通程序7-5说明在调用 longjmp后，自动变量、寄存器变量和易失变量的不同情况。如果以不带优化和带优化对此程序分别进行编译，然后运行它们，则得到的结果是不同的：

```
$ cc testjmp.c          不进行任何优化的编译
$ a.out
in f1(): count = 97, val = 98, sum = 99
after longjmp: count = 97, val = 98, sum = 99
$ cc -O testjmp.c      进行全部优化的编译
$ a.out
in f1(): count = 97, val = 98, sum = 99
after longjmp: count = 2, val = 3, sum = 99
```

注意，易失变量(sum)不受优化的影响，在 longjmp之后的值，是它在调用 f1时的值。在我们所使用的 setjmp(3)手册页上说明存放在存储器中的变量将具有 longjmp时的值，而在CPU和浮点寄存器中的变量则恢复为调用 setjmp时的值。这确实就是在运行程序7-5时所观察到的值。不进行优化时，所有这三个变量都存放在存储器中(亦即对 val的寄存器存储类被忽略)。而进行优化时，count和val都存放在寄存器中(即使count并未说明为register)，volatile变量则仍存放在存储器中。通过这一例子要理解的是，如果要编写一个使用非局部跳转的可移植程序，则必须使用volatile属性。

程序7-5 longjmp对自动，寄存器和易失变量的影响

```
#include <setjmp.h>
#include "ourhdr.h"

static void f1(int, int, int);
static void f2(void);

static jmp_buf jmpbuffer;

int
main(void)
{
    int          count;
    register int val;
    volatile int sum;

    count = 2; val = 3; sum = 4;
    if (setjmp(jmpbuffer) != 0) {
        printf("after longjmp: count = %d, val = %d, sum = %d\n",
               count, val, sum);
        exit(0);
    }
    count = 97; val = 98; sum = 99;
    /* changed after setjmp, before longjmp */
    f1(count, val, sum);      /* never returns */
}

static void
f1(int i, int j, int k)
{
    printf("in f1(): count = %d, val = %d, sum = %d\n", i, j, k);
    f2();
}
```

```
static void
f2(void)
{
    longjmp(jmpbuffer, 1);
}
```

第10章讨论信号处理程序及sigsetjmp和siglongjmp时，将再次使用setjmp和longjmp函数。

7.10.2 自动变量的潜在问题

前面已经说明了处理栈帧的一般方式，与此相关我们现在可以分析一下自动变量的一个潜在出错情况。基本规则是说明自动变量的函数已经返回后，就不能再引用这些自动变量。在整个UNIX手册中，关于这一点有很多警告。

程序7-6是一个名为open_data的函数，它打开了一个标准I/O流，然后为该流设置缓存。

程序7-6 自动变量的不正确使用

```
#include <stdio.h>

#define DATAFILE "datafile"

FILE *
open_data(void)
{
    FILE *fp;
    char databuf[BUFSIZ]; /* setvbuf makes this the stdio buffer */

    if ( (fp = fopen(DATAFILE, "r")) == NULL)
        return(NULL);

    if (setvbuf(fp, databuf, _IOLBF, BUFSIZ) != 0)
        return(NULL);

    return(fp); /* error */
}
```

问题是：当open_data返回时，它在栈上所使用的空间将由下一个被调用函数的栈帧使用。但是，标准I/O库函数仍将使用原先为databuf在栈上分配的存储空间作为该流的缓存。这就产生了冲突和混乱。为了改正这一问题，应在全局存储空间静态地（如static或extern），或者动态地（使用一种alloc函数）为数组databuf分配空间。

7.11 getrlimit和setrlimit函数

每个进程都有一组资源限制，其中某一些可以用getrlimit和setrlimit函数查询和更改。

```
#include <sys/time.h>
#include <sys/resource.h>

int getrlimit(int resource, struct rlimit * rlptr);

int setrlimit(int resource, const struct rlimit * rlptr);
```

两个函数返回：若成功则为0，若出错则为非0

对这两个函数的每一次调用都指定一个资源以及一个指向下列结构的指针。


```

struct rlimit {
    rlim_t rlim_cur;    /* soft limit: current limit */
    rlim_t rlim_max;    /* hard limit: maximum value for rlim_cur */
};

```

这两个函数不属于POSIX.1，但SVR4和4.3+BSD提供它们。

SVR4在上面的结构中使用基本系统数据类型 `rlim_t`，其他系统则将这两个成员定义为整型或长整型。

进程的资源限制通常是在系统初始化时由0进程建立的，然后由后续进程继承。在SVR4中，系统默认值可以查看文件 `/etc/conf/cf.d/mtune`。在4.3+BSD中，系统默认值分散在多个头文件中。

在更改资源限制时，须遵循下列三条规则：

(1) 任何一个进程都可将一个软限制更改为小于或等于其硬限制。

(2) 任何一个进程都可降低其硬限制值，但它必须大于或等于其软限制值。这种降低，对普通用户而言是不可逆反的。

(3) 只有超级用户可以提高硬限制。

一个无限制的由常数 `RLIM_INFINITY` 指定。

这两个函数的 *resource* 参数取下列值之一。注意并非所有资源限制都受到 SVR4和4.3+BSD 的支持。

- `RLIMIT_CORE` (SVR4及4.3+BSD) core文件的最大字节数，若其值为0则阻止创建core文件。

- `RLIMIT_CPU` (SVR4及4.3+BSD) CPU时间的最大量值(秒)，当超过此软限制时，向该进程发送 `SIGXCPU` 信号。

- `RLIMIT_DATA` (SVR4及4.3+BSD) 数据段的最大字节长度。这是图7-3中初始化数据、非初始化数据以及堆的总和。

- `RLIMIT_FSIZE` (SVR4及4.3+BSD) 可以创建的文件的最大字节长度。当超过此软限制时，则向该进程发送 `SIGXFSZ` 信号。

- `RLIMIT_MEMLOCK` (4.3+BSD) 锁定在存储器地址空间(尚未实现)。

- `RLIMIT_NOFILE` (SVR4) 每个进程能打开的最多文件数。更改此限制将影响到 `sysconf` 函数在参数 `_SC_OPEN_MAX` 中返回的值(见2.5.4节)。见程序2-3。

- `RLIMIT_NPROC` (4.3+BSD) 每个实际用户ID所拥有的最大子进程数。更改此限制将影响到 `sysconf` 函数在参数 `_SC_CHILD_MAX` 中返回的值(见2.5.4节)。

- `RLIMIT_OFILE` (4.3+BSD) 与SVR4的 `RLIMIT_NOFILE` 相同。

- `RLIMIT_RSS` (4.3+BSD) 最大驻内存集字节长度(RSS)。如果物理存储器供不应求，则内核将从进程处取回超过RSS的部分。

- `RLIMIT_STACK` (SVR4及4.3+BSD) 栈的最大字节长度。见图7-3。

- `RLIMIT_VMEM` (SVR4) 可映照地址空间的字节长度。这影响到 `mmap` 函数(见12.9节)。

资源限制影响到调用进程并由其子进程继承。这就意味着为了影响一个用户的所有后续进程，需将资源限制设置构造在 shell 之中。确实，Bourne shell 和 KornShell 具有内部 `ulimit` 命令，C shell 具有内部 `limit` 命令。(`umask` 和 `chdir` 函数也必须是 shell 内部的。)

早期的Bourne shell，例如由伯克利提供的一种，不支持ulimit命令。
较新的KornShell的ulimit命令具有-H和-S选择项，分别检查和修改硬或软限制，但它们尚未编写入文档。

实例

程序7-7打印由系统支持的对所有资源的当前软限制和硬限制。为了在SVR4和4.3+BSD之下运行此程序，必须条件编译不同的资源名。

程序7-7 打印当前资源限制

```
#include <sys/types.h>
#include <sys/time.h>
#include <sys/resource.h>
#include "ourhdr.h"

#define doit(name) pr_limits(#name, name)

static void pr_limits(char *, int);

int
main(void)
{
    doit(RLIMIT_CORE);
    doit(RLIMIT_CPU);
    doit(RLIMIT_DATA);
    doit(RLIMIT_FSIZE);

#ifdef RLIMIT_MEMLOCK
    doit(RLIMIT_MEMLOCK);
#endif
#ifdef RLIMIT_NOFILE /* SVR4 name */
    doit(RLIMIT_NOFILE);
#endif
#ifdef RLIMIT_OFILE /* 4.3+BSD name */
    doit(RLIMIT_OFILE);
#endif
#ifdef RLIMIT_NPROC
    doit(RLIMIT_NPROC);
#endif
#ifdef RLIMIT_RSS
    doit(RLIMIT_RSS);
#endif
    doit(RLIMIT_STACK);
#ifdef RLIMIT_VMEM
    doit(RLIMIT_VMEM);
#endif
    exit(0);
}

static void
pr_limits(char *name, int resource)
{
    struct rlimit limit;

    if (getrlimit(resource, &limit) < 0)
        err_sys("getrlimit error for %s", name);
    printf("%-14s ", name);
    if (limit.rlim_cur == RLIM_INFINITY)
        printf("(infinite) ");
}
```

```

else
    printf("%10ld ", limit.rlim_cur);
if (limit.rlim_max == RLIM_INFINITY)
    printf("(infinite)\n");
else
    printf("%10ld\n", limit.rlim_max);
}

```

注意，在doit宏中使用了新的ANSI C字符串创建算符(#)，以便为每个资源名产生字符串值。例如：

```
doit(RLIMIT_CORE);
```

这将由C预处理程序扩展为：

```
pr_limits("RLIMIT_CORE", RLIMIT_CORE);
```

在SVR4下运行此程序，得到：

```

$ a.out
RLIMIT_CORE          1048576          1048576
RLIMIT_CPU           (infinite)      (infinite)
RLIMIT_DATA          16777216        16777216
RLIMIT_FSIZE         2097152         2097152
RLIMIT_NOFILE         64              1024
RLIMIT_STACK         16777216        16777216
RLIMIT_VMEM          16777216        16777216

```

在4.3+BSD下运行此程序，得到：

```

$ a.out
RLIMIT_CORE          (infinite)      (infinite)
RLIMIT_CPU           (infinite)      (infinite)
RLIMIT_DATA          8388608         16777216
RLIMIT_FSIZE         (infinite)      (infinite)
RLIMIT_MEMLOCK       (infinite)      (infinite)
RLIMIT_OFILE         64              (infinite)
RLIMIT_NPROC         40              (infinite)
RLIMIT_RSS           27070464        27070464
RLIMIT_STACK         524288          16777216

```

在介绍了信号机构后，习题10.11将继续讨论资源限制。

7.12 小结

理解在UNIX环境中C程序的环境是理解UNIX进程控制特征的先决条件。本章说明了一个进程是如何起动和终止的，如何向其传递传数表和环境。虽然这两者都不是由内核进行解释的，但内核起到了从exec的调用者将这两者传递给新进程的作用。

本章也说明了C程序的典型存储器布局，以及一个进程如何动态地分配和释放存储器。详细地了解用于维护环境的一些函数是值得的，因为它们涉及存储器分配。本章也介绍了 setjmp和longjmp函数，它们提供了一种在进程内非本地转移的方法。最后介绍了 SVR4和4.3+BSD提供的资源限制功能。

习题

7.1 在80386系统上,无论使用SVR4或4.3+BSD,如果执行一个输出“hello, world”但不调用exit或return,则程序的返回代码为13(用shell检查),解释其原因。

7.2 程序7-1中的printf函数的结果何时才被真正输出?

7.3 是否有方法不使用(a)参数传递(b)全局变量这两种方法,将main中的参数argc,argv传递给它所调用的其他函数?

7.4 在有些UNIX系统中执行程序时,为什么访问不到其数据段的0单元?

7.5 用C语言的typedef定义一个新的数据类型Exitfunc处理退出,利用该类型修改atexit的原型。

7.6 如果用calloc分配一个long型的数组,数组的初始值是否为0?如果用calloc分配一个指针数组,数组的初始值是否为空指针?

7.7 在7.6节size命令的输出结果中,为什么没有给出堆和堆栈的大小?

7.8 为什么7.7节中两个文件的大小(104859和24576)不等于它们各自文本和数据大小的和?

7.9 为什么7.7节中对程序使用共享库以后改变了其可执行文件的大小?

7.10 在7.10节中我们已经说明为什么不能将一个指针返回给一个自动变量,下面的程序是否正确?

```
int
fl(int val)
{
    int      *ptr;
    if (val == 0) {
        int      val;
        val = 5;
        ptr = &val;
    }
    return (*ptr + 1);
}
```

第8章 进程控制

8.1 引言

本章介绍UNIX的进程控制，包括创建新进程、执行程序 and 进程终止。还将说明进程的各种ID——实际、有效和保存的用户和组ID，以及它们如何受到进程控制原语的影响。本章也包解释器文件和system函数。本章以大多数UNIX系统所提供的进程会计机制结束。这使我们从一个不同角度了解进程控制功能。

8.2 进程标识

每个进程都有一个非负整型的唯一进程ID。因为进程ID标识符总是唯一的，常将其用做其他标识符的一部分以保证其唯一性。5.13节中的tmpnam函数将进程ID作为名字的一部分创建一个唯一的路径名。

有某些专用的进程：进程ID 0是调度进程，常常被称为交换进程(swapper)。该进程并不执行任何磁盘上的程序——它是内核的一部分，因此也被称为系统进程。进程ID 1通常是init进程，在自举过程结束时由内核调用。该进程的程序文件在UNIX的早期版本中是/etc/init，在较新版本中是/sbin/init。此进程负责在内核自举后起一个UNIX系统。init通常读与系统有关的初始化文件(/etc/rc*文件)，并将系统引导到一个状态(例如多用户)。init进程决不会终止。它是一个普通的用户进程(与交换进程不同，它不是内核中的系统进程)，但是它以超级用户特权运行。本章稍后部分会说明init如何成为所有孤儿进程的父进程。

在某些UNIX的虚存实现中，进程ID 2是页精灵进程(pagedaemon)。此进程负责支持虚存系统的请页操作。与交换进程一样，页精灵进程也是内核进程。

除了进程ID，每个进程还有一些其他标识符。下列函数返回这些标识符。

<code>#include <sys/types.h></code>	
<code>#include <unistd.h></code>	
<code>pid_t getpid(void);</code>	返回：调用进程的进程ID
<code>pid_t getppid(void);</code>	返回：调用进程的父进程ID
<code>uid_t getuid(void);</code>	返回：调用进程的实际用户ID
<code>uid_t geteuid(void);</code>	返回：调用进程的有效用户ID
<code>gid_t getgid(void);</code>	返回：调用进程的实际组ID
<code>gid_t getegid(void);</code>	返回：调用进程的有效组ID

注意，这些函数都没有出错返回，在下一节中讨论fork函数时，将进一步讨论父进程ID。4.4节中已讨论了实际和有效用户及组ID。

8.3 fork函数

一个现存进程调用fork函数是UNIX内核创建一个新进程的唯一方法(这并不适用于前节提及的交换进程、init进程和页精灵进程。这些进程是由内核作为自举过程的一部分以特殊方式创建的)。

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

pid_t fork(void);
```

返回：子进程中为0，父进程中为子进程ID，出错为-1

由fork创建的新进程被称为子进程(child process)。该函数被调用一次，但返回两次。两次返回的区别是子进程的返回值是0，而父进程的返回值则是新子进程的进程ID。将子进程ID返回给父进程的理由是：因为一个进程的子进程可以多于一个，所以没有一个函数使一个进程可以获得其所有子进程的进程ID。fork使子进程得到返回值0的理由是：一个进程只会有一个父进程，所以子进程总是可以调用getppid以获得其父进程的进程ID(进程ID 0总是由交换进程使用，所以一个子进程的进程ID不可能为0)。

子进程和父进程继续执行fork之后的指令。子进程是父进程的复制品。例如，子进程获得父进程数据空间、堆和栈的复制品。注意，这是子进程所拥有的拷贝。父、子进程并不共享这些存储空间部分。如果正文段是只读的，则父、子进程共享正文段(见7.6节)。

现在很多的实现并不做一个父进程数据段和堆的完全拷贝，因为在fork之后经常跟随着exec。作为替代，使用了在写时复制(Copy-On-Write, COW)的技术。这些区域由父、子进程共享，而且内核将它们的存取许可权改变为只读的。如果有进程试图修改这些区域，则内核为有关部分，典型的是虚存系统中的“页”，做一个拷贝。Bach [1986]的9.2节和Leffler等 [1989]的5.7节对这种特征做了更详细的说明。

实例

程序8-1例示了fork函数。如果执行此程序则得到：

```
$ a.out
a write to stdout
before fork
pid = 430, glob = 7, var = 89 子进程的变量值改变了
pid = 429, glob = 6, var = 88 父进程的变量值没有改变
$ a.out > temp.out
$ cat temp.out
a write to stdout
before fork
pid = 432, glob = 7, var = 89
before fork
pid = 431, glob = 6, var = 88
```

一般来说，在fork之后是父进程先执行还是子进程先执行是不确定的。这取决于内核所使用的调度算法。如果要求父、子进程之间相互同步，则要求某种形式的进程间通信。在程序8-1中，父进程使自己睡眠2秒钟，以此使子进程先执行。但并不保证2秒钟已经足够，在8.8节说明竞争条件时，还将谈及这一问题及其他类型的同步方法。在10.6节中，在fork之后将用信号使父、

子进程同步。

注意，程序 8-1 中 fork 与 I/O 函数之间的关系。回忆第 3 章中所述，write 函数是不带缓存的。因为在 fork 之前调用 write，所以其数据写到标准输出一次。但是，标准 I/O 库是带缓存的。回忆一下 5.12 节，如果标准输出连到终端设备，则它是行缓存的，否则它是全缓存的。当以交互方式运行该程序时，只得到 printf 输出的行一次，其原因是标准输出缓存由新行符刷新。但是当将标准输出重新定向到一个文件时，却得到 printf 输出行两次。其原因是，在 fork 之前调用了 printf 一次，但当调用 fork 时，该行数据仍在缓存中，然后在父进程数据空间复制到子进程中时，该缓存数据也被复制到子进程中。于是那时父、子进程各自有了带该行内容的缓存。在 exit 之前的第二个 printf 将其数据添加到现存的缓存中。当每个进程终止时，其缓存中的内容被写到相应文件中。

程序 8-1 fork 函数实例

```
#include <sys/types.h>
#include "ourhdr.h"

int glob = 6; /* external variable in initialized data */
char buf[] = "a write to stdout\n";

int
main(void)
{
    int var; /* automatic variable on the stack */
    pid_t pid;

    var = 88;
    if (write(STDOUT_FILENO, buf, sizeof(buf)-1) != sizeof(buf)-1)
        err_sys("write error");
    printf("before fork\n"); /* we don't flush stdout */

    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0) { /* child */
        glob++; /* modify variables */
        var++;
    } else
        sleep(2); /* parent */

    printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
    exit(0);
}
```

文件共享

对程序 8-1 需注意的另一点是：在重新定向父进程的标准输出时，子进程的标准输出也被重新定向。实际上，fork 的一个特性是所有由父进程打开的描述符都被复制到子进程中。父、子进程每个相同的打开描述符共享一个文件表项（见图 3-3）。

考虑下述情况，一个进程打开了三个不同文件，它们是：标准输入、标准输出和标准出错。在从 fork 返回时，我们有了如图 8-1 中所示的安排。

这种共享文件的方式使父、子进程对同一文件使用了一个文件位移量。考虑下述情况：一个进程 fork 了一个子进程，然后等待子进程终止。假定，作为普通处理的一部分，父、子进程都向标准输出执行写操作。如果父进程使其标准输出重新定向（很可能是由 shell 实现的），那么子进程写到该标准输出时，它将更新与父进程共享的该文件的位移量。在我们所考虑的例子中，当父进程等待子进程时，子进程写到标准输出；而在子进程终止后，父进程也写到标准输出上，

并且知道其输出会添加在子进程所写数据之后。如果父、子进程不共享同一文件位移量，这种形式的交互就很难实现。

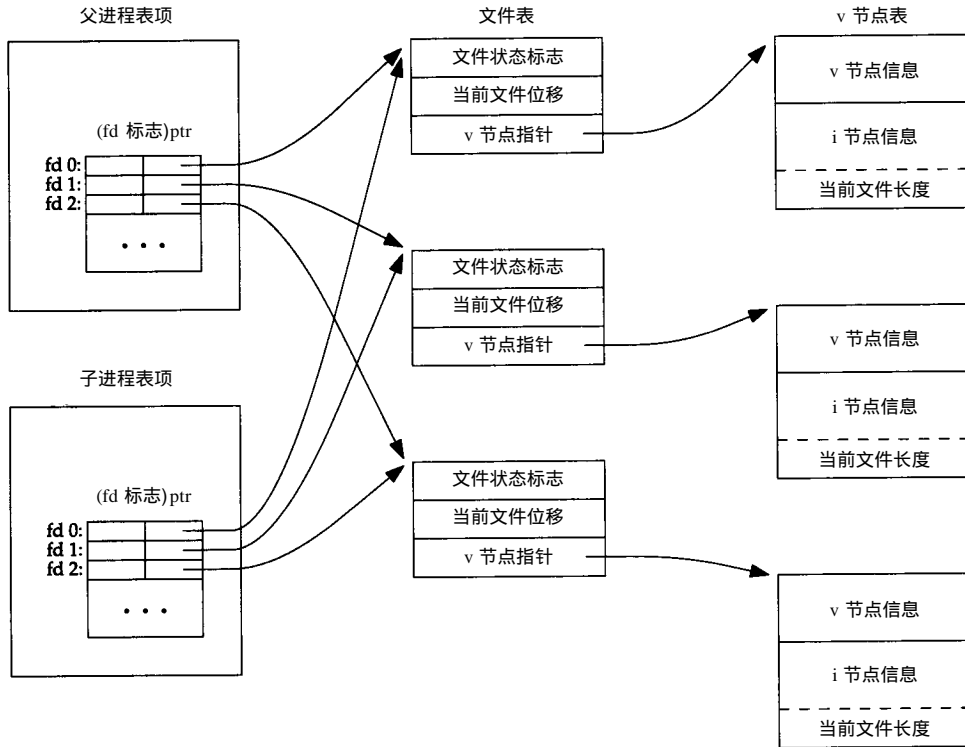


图8-1 `fork`之后父、子进程之间对打开文件的共享

如果父、子进程写到同一描述符文件，但又没有任何形式的同步（例如使父进程等待子进程），那么它们的输出就会相互混合（假定所用的描述符是在 `fork`之前打开的）。虽然这种情况是可能发生的（见程序8-1），但这并不是常用的操作方式。

在`fork`之后处理文件描述符有两种常见的情况：

(1) 父进程等待子进程完成。在这种情况下，父进程无需对其描述符做任何处理。当子进程终止后，它曾进行过读、写操作的任一共享描述符的文件位移量已做了相应更新。

(2) 父、子进程各自执行不同的程序段。在这种情况下，在 `fork`之后，父、子进程各自关闭它们不需使用的文件描述符，并且不干扰对方使用的文件描述符。这种方法是网络服务进程中经常使用的。

除了打开文件之外，很多父进程的其他性质也由子进程继承：

- 实际用户ID、实际组ID、有效用户ID、有效组ID。
- 添加组ID。
- 进程组ID。
- 对话期ID。
- 控制终端。
- 设置-用户-ID标志和设置-组-ID标志。
- 当前工作目录。

- 根目录。
- 文件方式创建屏蔽字。
- 信号屏蔽和排列。
- 对任一打开文件描述符的在执行时关闭标志。
- 环境。
- 连接的共享存储段。
- 资源限制。

父、子进程之间的区别是：

- fork的返回值。
- 进程ID。
- 不同的父进程ID。
- 子进程的tms_utime,tms_stime,tms_cutime以及tms_ustime设置为0。
- 父进程设置的锁，子进程不继承。
- 子进程的未决告警被清除。
- 子进程的未决信号集设置为空集。

其中很多特性至今尚未讨论过，我们将在以后几章中对它们进行说明。

使fork失败的两个主要原因是：(a)系统中已经有了太多的进程(通常意味着某个方面出了问题)，或者(b)该实际用户ID的进程总数超过了系统限制。回忆表 2-7，其中CHILD_MAX规定了每个实际用户ID在任一时刻可具有的最大进程数。

fork有两种用法：

(1) 一个父进程希望复制自己，使父、子进程同时执行不同的代码段。这在网络服务进程中是常见的——父进程等待委托者的服务请求。当这种请求到达时，父进程调用 fork，使子进程处理此请求。父进程则继续等待下一个服务请求。

(2) 一个进程要执行一个不同的程序。这对 shell是常见的情况。在这种情况下，子进程在从fork返回后立即调用exec(我们将在8.9节说明exec)。

某些操作系统将(2)中的两个操作(fork之后执行exec)组合成一个，并称其为spawn。UNIX将这两个操作分开，因为在很多场合需要单独使用 fork，其后并不跟随exec。另外，将这两个操作分开，使得子进程在fork和exec之间可以更改自己的属性。例如I/O重新定向、用户ID、信号排列等。在第14章中有很多这方面的例子。

8.4 vfork函数

vfork函数的调用序列和返回值与fork相同，但两者的语义不同。

vfork起源于较早的4BSD虚存版本。在Leffler等[1989]的5.7节中指出：“虽然它是特别有效率的，但是vfork的语义很奇特，通常认为它具有结构上的缺陷。”

尽管如此SVR4和4.3+BSD仍支持vfork。

某些系统具有头文件<vfork.h>，当调用vfork时，应当包括该头文件。

vfork用于创建一个新进程，而该新进程的目的是 exec一个新程序(如上节(2)中一样)。程序1-5中的shell基本部分就是这种类型程序的一个例子。vfork与fork一样都创建一个子进程，但是它并不将父进程的地址空间完全复制到子进程中，因为子进程会立即调用 exec(或exit)，于

是也就不会存访该地址空间。不过在子进程调用 `exec`或`exit`之前，它在父进程的空间中运行。这种工作方式在某些UNIX的页式虚存实现中提高了效率（与上节中提及的，在 `fork`之后跟随 `exec`，并采用在写时复制技术相类似）。

`vfork`和`fork`之间的另一个区别是：`vfork`保证子进程先运行，在它调用 `exec`或`exit`之后父进程才可能被调度运行。（如果在调用这两个函数之前子进程依赖于父进程的进一步动作，则会导致死锁。）

实例

在程序8-1中使用`vfork`代替`fork`，并做其他相应修改得到程序8-2。

程序8-2 `vfork` 函数实例

```
#include <sys/types.h>
#include "ourhdr.h"

int glob = 6; /* external variable in initialized data */

int
main(void)
{
    int var; /* automatic variable on the stack */
    pid_t pid;

    var = 88;
    printf("before vfork\n"); /* we don't flush stdio */

    if ( (pid = vfork()) < 0)
        err_sys("vfork error");
    else if (pid == 0) { /* child */
        glob++; /* modify parent's variables */
        var++;
        _exit(0); /* child terminates */
    }

    /* parent */
    printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
    exit(0);
}
```

运行该程序得到：

```
$ a.out
before vfork
pid = 607, glob = 7, var = 89
```

子进程对变量`glob`和`var`做增1操作，结果改变了父进程中的变量值。因为子进程在父进程的地址空间中运行，所以这并不令人惊讶。但是其作用的确与`fork`不同。

注意，在程序8-2中，调用了`_exit`而不是`exit`。正如8.5节所述，`_exit`并不执行标准I/O缓存的刷新操作。如果用`exit`而不是`_exit`，则该程序的输出是：

```
$ a.out
before vfork
```

从中可见，父进程`printf`的输出消失了。其原因是子进程调用了`exit`，它刷新关闭了所有标准I/O流，这包括标准输出。虽然这是由子进程执行的，但却是在父进程的地址空间中进行的，所以所有受到影响的标准I/O FILE对象都是在父进程中的。当父进程调用`printf`时，标准输出已

被关闭了，于是printf返回-1。

Leffler 等 [1989] 在5.7节中包含了fork和vfork实现方面的更多信息。习题8.1和8.2则继续了对vfork的讨论。

8.5 exit函数

如同7.3节所述，进程有三种正常终止法及两种异常终止法。

(1) 正常终止：

- (a) 在main函数内执行return语句。如在7.3节中所述，这等效于调用exit。
- (b) 调用exit函数。此函数由ANSI C定义，其操作包括调用各终止处理程序（终止处理程序在调用atexit函数时登录），然后关闭所有标准/O流等。因为ANSI C并不处理文件描述符、多进程（父、子进程）以及作业控制，所以这一定义对UNIX系统而言是不完整的。
- (c) 调用_exit系统调用函数。此函数由exit调用，它处理UNIX特定的细节。_exit是由POSIX.1说明的。

(2) 异常终止：

- (a) 调用abort。它产生SIGABRT信号，所以是下一种异常终止的一种特例。
- (b) 当进程接收到某个信号时。（第10章将较详细地说明信号。）进程本身（例如调用abort函数）、其他进程和内核都能产生传送到某一进程的信号。例如，进程越出其地址空间访问存储单元，或者除以0，内核就会为该进程产生相应的信号。

不管进程如何终止，最后都会执行内核中的同一段代码。这段代码为相应进程关闭所有打开描述符，释放它所使用的存储器等等。

对上述任意一种终止情形，我们都希望终止进程能够通知其父进程它是如何终止的。对于exit和_exit，这是依靠传递给它们的退出状态（exit status）参数来实现的。在异常终止情况，内核（不是进程本身）产生一个指示其异常终止原因的终止状态（termination status）。在任意一种情况下，该终止进程的父进程都能用wait或waitpid函数（在下一节说明）取得其终止状态。

注意，这里使用了“退出状态”（它是传向exit或_exit的参数，或main的返回值）和“终止状态”两个术语，以表示有所区别。在最后调用_exit时内核将其退出状态转换成终止状态（回忆图7-1）。下一节中的表8-1说明了父进程检查子进程的终止状态的不同方法。如果子进程正常终止，则父进程可以获得子进程的退出状态。

在说明fork函数时，一定是一个父进程生成一个子进程。上面又说明了子进程将其终止状态返回给父进程。但是如果父进程在子进程之前终止，则将如何呢？其回答是对于其父进程已经终止的所有进程，它们的父进程都改变为init进程。我们称这些进程由init进程领养。其操作过程大致是：在一个进程终止时，内核逐个检查所有活动进程，以判断它是否是正要终止的进程的子进程，如果是，则该进程的父进程ID就更改为1(init进程的ID)。这种处理方法保证了每个进程有一个父进程。

另一个我们关心的情况是如果子进程在父进程之前终止，那么父进程又如何能在做相应检查时得到子进程的终止状态呢？对此问题的回答是内核为每个终止子进程保存了一定量的信息，所以当终止进程的父进程调用wait或waitpid时，可以得到有关信息。这种信息至少包括进程ID、该进程的终止状态、以反该进程使用的CPU时间总量。内核可以释放终止进程所使用的所有存储器，关闭其所有打开文件。在UNIX术语中，一个已经终止、但是其父进程尚未对其进行善后处理（获取终止子进程的有关信息、释放它仍占用的资源）的进程被称为僵死

进程 (zombie)。ps(1)命令将僵死进程的状态打印为 Z。如果编写一个长期运行的程序，它 fork 了很多子进程，那么除非父进程等待取得子进程的终止状态，否则这些子进程就会变成僵死进程。

系统V提供了一种避免僵死进程的非标准化方法，这将在 10.7中介绍。

最后一个要考虑的问题是：一个由 init进程领养的进程终止时会发生什么？它会不会变成一个僵死进程？对此问题的回答是“否”，因为init被编写成只要有一个子进程终止，init就会调用一个wait函数取得其终止状态。这样也就防止了在系统中有很多僵死进程。当提及“一个init的子进程”时，这指的是init直接产生的进程(例如，将在9.2节说明的getty进程)，或者是其父进程已终止，由init领养的进程。

8.6 wait和waitpid函数

当一个进程正常或异常终止时，内核就向其父进程发送 SIGCHLD信号。因为子进程终止是个异步事件(这可以在父进程运行的任何时候发生)，所以这种信号也是内核向父进程发的异步通知。父进程可以忽略该信号，或者提供一个该信号发生时即被调用执行的函数(信号处理程序)。对于这种信号的系统默认动作是忽略它。第10章将说明这些选择项。现在需要知道的是调用wait或waitpid的进程可能会：

- 阻塞(如果其所有子进程都还在运行)。
- 带子进程的终止状态立即返回(如果一个子进程已终止，正等待父进程存取其终止状态)。
- 出错立即返回(如果它没有任何子进程)。

如果进程由于接收到SIGCHLD信号而调用wait，则可期望wait会立即返回。但是如果在一个任一时刻调用wait，则进程可能会阻塞。

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

pid_t wait(int *statloc);

pid_t waitpid(pid_t pid, int *statloc, int options);
```

两个函数返回：若成功则为进程ID，若出错则为-1

这两个函数的区别是：

• 在一个子进程终止前，wait使其调用者阻塞，而waitpid有一选择项，可使调用者不阻塞。

• waitpid并不等待第一个终止的子进程——它有若干个选择项，可以控制它所等待的进程。

如果一个子进程已经终止，是一个僵死进程，则wait立即返回并取得该子进程的状态，否则wait使其调用者阻塞直到一个子进程终止。如调用者阻塞而且它有多个子进程，则在其一个子进程终止时，wait就立即返回。因为wait返回终止子进程的进程ID，所以它总能了解是哪一个子进程终止了。

这两个函数的参数statloc是一个整型指针。如果statloc不是一个空指针，则终止进程的终止状态就存放在它所指向的单元内。如果不关心终止状态，则可将该参数指定为空指针。

依据传统，这两个函数返回的整型状态字是由实现定义的。其中某些位表示退出状态(正

常返回)，其他位则指示信号编号（异常返回），有一位指示是否产生了一个 core 文件等等。POSIX.1 规定终止状态用定义在 `<sys/wait.h>` 中的各个宏来查看。有三个互斥的宏可用来取得进程终止的原因，它们的名字都以 WIF 开始。基于这三个宏中哪一个值是真，就可选用其他宏来取得终止状态、信号编号等。这些都在表 8-1 中给出。在 8.9 节中讨论作业控制时，将说明如何停止一个进程。

表8-1 检查wait和waitpid所返回的终止状态的宏

宏	说 明
WIFEXITED(<i>status</i>)	若为正常终止子进程返回的状态，则为真。对于这种情况可执行 WEXITSTATUS(<i>status</i>) 取子进程传递给 exit 或 _exit 参数的低 8 位
WIFSIGNALED(<i>status</i>)	若为异常终止子进程返回的状态，则为真（接到一个不捕捉的信号）。对于这种情况，可执行 WTERMSIG(<i>status</i>) 取使子进程终止的信号编号。 另外，SVR4 和 4.3+BSD（但是，非 POSIX.1）定义宏： WCOREDUMP(<i>status</i>) 若已产生终止进程的 core 文件，则它返回真
WIFSTOPPED(<i>status</i>)	若为当前暂停子进程的返回的状态，则为真。对于这种情况，可执行 WSTOPSIG(<i>status</i>) 取使子进程暂停的信号编号

实例

程序 8-3 中的函数 `pr_exit` 使用表 8-1 中的宏以打印进程的终止状态。本章的很多程序都将调用此函数。注意，如果定义了 `WCOREDUMP`，则此函数也处理该宏。

程序 8-4 调用 `pr_exit` 函数，例示终止状态的不同值。运行程序 8-4 可得：

```
$ a.out
normal termination, exit status = 7
abnormal termination, signal number = 6 (core file generated)
abnormal termination, signal number = 8 (core file generated)
```

程序8-3 打印exit状态的说明

```
#include <sys/types.h>
#include <sys/wait.h>
#include "ourhdr.h"

void
pr_exit(int status)
{
    if (WIFEXITED(status))
        printf("normal termination, exit status = %d\n",
              WEXITSTATUS(status));
    else if (WIFSIGNALED(status))
        printf("abnormal termination, signal number = %d%s\n",
              WTERMSIG(status),
              WCOREDUMP(status) ? " (core file generated)" : "");
}
```

```
#else
    "");
#endif
    else if (WIFSTOPPED(status))
        printf("child stopped, signal number = %d\n",
            WSTOPSIG(status));
}
```

程序8-4 例示不同的exit值

```
#include <sys/types.h>
#include <sys/wait.h>
#include "ourhdr.h"

int
main(void)
{
    pid_t pid;
    int status;

    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0) /* child */
        exit(7);

    if (wait(&status) != pid) /* wait for child */
        err_sys("wait error");
    pr_exit(status); /* and print its status */

    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0) /* child */
        abort(); /* generates SIGABRT */

    if (wait(&status) != pid) /* wait for child */
        err_sys("wait error");
    pr_exit(status); /* and print its status */

    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0) /* child */
        status /= 0; /* divide by 0 generates SIGFPE */

    if (wait(&status) != pid) /* wait for child */
        err_sys("wait error");
    pr_exit(status); /* and print its status */

    exit(0);
}
```

不幸的是，没有一种可移植的方法将 WTERMSIG得到的信号编号映射为说明性的名字。(10.21节中说明了一种方法。)我们必须查看<signal.h>头文件才能知道 SIGABRT的值是6，SIGFPE的值是8。

正如前面所述，如果一个进程有几个子进程，那么只要有一个子进程终止，wait就返回。如果要等待一个指定的进程终止(如果知道要等待进程的ID)，那么该如何做呢？在早期的UNIX版本中，必须调用wait，然后将其返回的进程ID和所期望的进程ID相比较。如果终止进程不是所期望的，则将该进程ID和终止状态保存起来，然后再次调用wait。反复这样做直到所期望的进程终止。下一次又想等待一个特定进程时，先查看已终止的进程表，若其中已有要等待的进程，则取有关信息，否则调用wait。其实，我们需要的是等待一个特定进程的函数。POSIX.1

定义了waitpid函数以提供这种功能(以及其他一些功能)。

waitpid函数是新由POSIX.1定义的。SVR4和4.3+BSD都提供此函数，但早期的系统V和4.3BSD并不提供此函数。

对于waitpid的*pid*参数的解释与其值有关：

- *pid* == -1 等待任一子进程。于是在这一功能方面 waitpid与wait等效。
- *pid* > 0 等待其进程ID与*pid*相等的子进程。
- *pid* == 0 等待其组ID等于调用进程的组ID的任一子进程。
- *pid* < -1 等待其组ID等于*pid*的绝对值的任一子进程。

(9.4节将说明进程组。)waitpid返回终止子进程的进程ID，而该子进程的终止状态则通过 *statloc* 返回。对于wait，其唯一的出错是调用进程没有子进程(函数调用被一个信号中断时，也可能返回另一种出错。第10章将对此进行讨论)。但是对于waitpid，如果指定的进程或进程组不存在，或者调用进程没有子进程都能出错。

*options*参数使我们能进一步控制waitpid的操作。此参数或者是0，或者是表8-2中常数的逐位或运算。

表8-2 waitpid的选择项常数

常 数	说 明
WNOHANG	若由 <i>pid</i> 指定的子进程并不立即可用，则 waitpid不阻塞，此时其返回值为0
WUNTRACED	若某实现支持作业控制，则由 <i>pid</i> 指定的任一子进程状态已暂停，且其状态自暂停以来还未报告过，则返回其状态。WIFSTOPPED宏确定返回值是否对应于一个暂停子进程

SVR4支持两个附加的非标准的 *options*常数。WNOWAIT使系统将其终止状态已由waitpid返回的进程保持在等待状态，于是该进程就可被再次等待。对于WCONTINUED，返回由*pid*指定的某一子进程的状态，该子进程已被继续，其状态尚未报告过。

waitpid函数提供了wait函数没有提供的三个功能：

(1) waitpid等待一个特定的进程(而wait则返回任一终止子进程的状态)。在讨论popen函数时会再说明这一功能。

(2) waitpid提供了一个wait的非阻塞版本。有时希望取得一个子进程的状态，但不想阻塞。

(3) waitpid支持作业控制(以WUNTRACED选择项)。

实例

回忆一下8.5节中有关僵死进程的讨论。如果一个进程要fork一个子进程，但不要求它等待子进程终止，也不希望子进程处于僵死状态直到父进程终止，实现这一要求的诀窍是调用fork两次。程序8-5实现了这一点。

在第二个子进程中调用sleep以保证在打印父进程ID时第一个子进程已终止。在fork之后，父、子进程都可继续执行——我们无法预知哪一个会先执行。如果不使第二个子进程睡眠，则在fork之后，它可能比其父进程先执行，于是它打印的父进程ID将是创建它的父进程，而不是

init进程 (进程ID 1)。

程序8-5 fork两次以避免僵死进程

```
#include <sys/types.h>
#include <sys/wait.h>
#include "ourhdr.h"

int
main(void)
{
    pid_t  pid;

    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0) { /* first child */
        if ( (pid = fork()) < 0)
            err_sys("fork error");
        else if (pid > 0)
            exit(0); /* parent from second fork == first child */

        /* We're the second child; our parent becomes init as soon
           as our real parent calls exit() in the statement above.
           Here's where we'd continue executing, knowing that when
           we're done, init will reap our status. */

        sleep(2);
        printf("second child, parent pid = %d\n", getppid());
        exit(0);
    }

    if (waitpid(pid, NULL, 0) != pid) /* wait for first child */
        err_sys("waitpid error");

    /* We're the parent (the original process); we continue executing,
       knowing that we're not the parent of the second child. */

    exit(0);
}
```

执行程序8-5得到：

```
$ a.out
$ second child, parent pid = 1
```

注意，当原先的进程（也就是exec本程序的进程）终止时，shell打印其指示符，这在第二个子进程打印其父进程ID之前。

8.7 wait3和wait4函数

4.3+BSD提供了两个附加函数wait3和wait4。这两个函数提供的功能比POSIX.1函数wait和waitpid所提供的分别要多一个，这与附加参数*rusage*有关。该参数要求内核返回由终止进程及其所有子进程使用的资源摘要。

```
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/time.h>
#include <sys/resource.h>

pid_t wait3(int statloc, int options, struct rusagerusage);
```



```
pid_t wait4(pid_pid, int *statloc, int options, struct rusage*rusage);
```

两个函数返回：若成功则为进程 ID，若出错则为 -1

SVR4在其BSD兼容库中也提供了 wait3函数。

资源信息包括用户 CPU 时间总量、系统 CPU 时间总量、缺页次数、接收到信号的次数等。有关细节请参阅 `getrusage(2)` 手册页。这些资源信息只包括终止子进程，并不包括处于停止状态的子进程（这种资源信息与 7.11 节中所述的资源限制不同）。表 8-3 中列出了各个 wait 函数所支持的不同的参数。

表 8-3 不同系统上各个 wait 函数所支持的参数

函 数	<i>pid</i>	<i>options</i>	<i>rusage</i>	POSIX.1	SVR4	4.3+BSD
wait				•	•	•
waitpid	•	•		•	•	•
wait3		•	•		•	•
wait4	•	•	•			•

8.8 竞态条件

从本书的目的出发，当多个进程都企图对共享数据进行某种处理，而最后的结果又取决于进程运行的顺序时，则我们认为这发生了竞态条件（race condition）。如果在 fork 之后的某种逻辑显式或隐式地依赖于在 fork 之后是父进程先运行还是子进程先运行，那么 fork 函数就会是竞态条件活跃的孳生地。通常，我们不能预料哪一个进程先运行。即使知道哪一个进程先运行，那么在该进程开始运行后，所发生的事情也依赖于系统负载以及内核的调度算法。

在程序 8-5 中，当第二个子进程打印其父进程 ID 时，我们看到了一个潜在的竞态条件。如果第二个子进程在第一个子进程之前运行，则其父进程将会是第一个子进程。但是，如果第一个子进程先运行，并有足够的时间到达并执行 exit，则第二个子进程的父进程就是 init。即使在程序中调用 sleep，这也不保证什么。如果系统负担很重，那么在第二个子进程从 sleep 返回时，可能第一个子进程还没有得到机会运行。这种形式的问题很难排除，因为在大部分时间，这种问题并不出现。

如果一个进程希望等待一个子进程终止，则它必须调用 wait 函数。如果一个进程要等待其父进程终止（如程序 8-5 中一样），则可使用下列形式的循环：

```
while(getppid() !=1)
    sleep(1);
```

这种形式的循环（称为定期询问（polling））的问题是它浪费了 CPU 时间，因为调用者每隔 1 秒都被唤醒，然后进行条件测试。

为了避免竞态条件和定期询问，在多个进程之间需要有某种形式的信号机制。在 UNIX 中可以使用信号机制，在 10.16 节将说明它的一种用法。各种形式的进程间通信（IPC）也可使用，在第 14、15 章将对此进行讨论。

在父、子进程的关系中，常常出现下述情况。在 fork 之后，父、子进程都有一些事情要做。

例如，父进程可能以子进程ID更新日志文件中的一个记录，而子进程则可能要为父进程创建一个文件。在本例中，要求每个进程在执行完它的一套初始化操作后要通知对方，并且在继续运行之前，要等待另一方完成其初始化操作。这种情况可以描述如下：

```
#include "ourhdr.h"

TELL_WAIT(); /* set things up for TELL_xxx & WAIT_xxx */

if ( (pid = fork()) < 0)
    err_sys("fork error");
else if (pid == 0) { /* child */

    /* child does whatever is necessary ... */

    TELL_PARENT(getppid()); /* tell parent we're done */
    WAIT_PARENT(); /* and wait for parent */

    /* and the child continues on its way ... */
    exit(0);
}

/* parent does whatever is necessary ... */

TELL_CHILD(pid); /* tell child we're done */
WAIT_CHILD(); /* and wait for child */

/* and the parent continues on its way ... */
exit(0);
```

假定在头文件ourhdr.h中定义了各个需要使用的变量。五个例程TELL_WAIT、TELL_PARENT、TELL_CHILD、WAIT_PARENT以及WAIT_CHILD可以是宏，也可以是函数。

在后面的一些章中会说明实现这些 TELL和WAIT例程的不同方法：10.16节中说明用信号的一种实现，程序14-3中说明用流管道的一种实现。下面先看一个使用这五个例程的实例。

实例

程序8-6输出两个字符串：一个由子进程输出，一个由父进程输出。因为输出依赖于内核使进程运行的顺序及每个进程运行的时间长度，所以该程序包含了一个竞态条件。

程序8-6 具有竞态条件的程序

```
#include <sys/types.h>
#include "ourhdr.h"

static void charatime(char *);

int
main(void)
{
    pid_t pid;

    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0) {
        charatime("output from child\n");
    } else {
        charatime("output from parent\n");
    }
    exit(0);
}

static void
```

```

charatotime(char *str)
{
    char    *ptr;
    int     c;

    setbuf(stdout, NULL);          /* set unbuffered */
    for (ptr = str; c = *ptr++; )
        putc(c, stdout);
}

```

在程序中将标准输出设置为不带缓存的，于是每个字符输出都需调用一次 `write`。本例的目的是使内核能尽可能多次地在两个进程之间进行切换，以例示竞态条件。（如果不这样做，可能也就决不会见到下面所示的输出。没有看到具有错误的输出并不意味着竞态条件不存在，这只是意味着在此特定的系统上未能见到它。）下面的实际输出说明该程序的运行结果是会改变的。

```

$ a.out
output from child
output from parent
$ a.out
oouuttppuutt ffrroomm cphairledn
t
$ a.out
oouuttppuutt ffrroomm pcahrielndt

$ a.out
ooutput from parent
utput from child

```

修改程序 8-6，使其使用 `TELL` 和 `WAIT` 函数，于是形成了程序 8-7。行首标以 + 号的行是新增加的行。

程序 8-7 修改程序 8-6 以避免竞态条件

```

#include <sys/types.h>
#include "ourhdr.h"

static void charatotime(char *);

int
main(void)
{
    pid_t  pid;
+   TELL_WAIT();
+
    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0) {
+       WAIT_PARENT();          /* parent goes first */
        charatotime("output from child\n");
    } else {
+       charatotime("output from parent\n");
+       TELL_CHILD(pid);
    }
    exit(0);
}

static void

```

```

charatotime(char *str)
{
    char    *ptr;
    int     c;

    setbuf(stdout, NULL);          /* set unbuffered */
    for (ptr = str; c = *ptr++; )
        putchar(c, stdout);
}

```

运行此程序则能得到所预期的输出——两个进程的输出不再交叉混合。

程序8-7是使父进程先运行。如果将fork之后的行改变成：

```

else if (pid == 0) {
    charatotime("output from child\n");
    TELL_PARENT(getppid());
} else {
    WAIT_CHILD();          /* child goes first */
    charatotime("output from parent\n");
}

```

则子进程先运行。习题8.3继续这一实例。

8.9 exec函数

8.3节曾提及用fork函数创建子进程后，子进程往往要调用一种exec函数以执行另一个程序。当进程调用一种exec函数时，该进程完全由新程序代换，而新程序则从其main函数开始执行。因为调用exec并不创建新进程，所以前后的进程ID并未改变。exec只是用另一个新程序替换了当前进程的正文、数据、堆和栈段。

有六种不同的exec函数可供使用，它们常常被统称为exec函数。这些exec函数都是UNIX进程控制原语。用fork可以创建新进程，用exec可以执行新的程序。exit函数和两个wait函数处理终止和等待终止。这些是我们需要的基本的进程控制原语。在后面各节中将使用这些原语构造另外一些如popen和system之类的函数。

```

#include <unistd.h>

int execl(const char *pathname, const char * arg0, ... /* (char *) 0 */);

int execv(const char *pathname, char *const argv [] );

int execlp(const char *pathname, const char * arg0, ...
           /* (char *)0, char *const envp [] */);

int execve(const char *pathname, char *const argv [], char *const envp [] );

int execlp(const char *filename, const char * arg0, ... /* (char *) 0 */);

int execvp(const char *filename, char *const argv [] );

```

六个函数返回：若出错则为-1，若成功则不返回

这些函数之间的第一个区别是前四个取路径名作为参数，后两个则取文件名作为参数。当指定filename作为参数时：

- 如果 *filename* 中包含 /, 则就将其视为路径名。
- 否则就按 PATH 环境变量, 在有关目录中搜寻可执行文件。

PATH 变量包含了一张目录表 (称为路径前缀), 目录之间用冒号 (:) 分隔。例如下列 *name=value* 环境字符串:

```
PATH=/bin:/usr/bin:/usr/local/bin:.
```

指定在四个目录中进行搜索。(零长前缀也表示当前目录。在 *value* 的开始处可用 : 表示, 在行中间则要用 :: 表示, 在行尾以 : 表示。)

有很多出于安全性方面的考虑, 要求在搜索路径中决不要包括当前目录。请参见 Garfinkel 和 Spafford [1991]。

如果 `execlp` 和 `execvp` 中的任意一个使用路径前缀中的一个找到了一个可执行文件, 但是该文件不是由连接编辑程序产生的机器可执行代码文件, 则认为该文件是一个 shell 脚本, 于是试着调用 `/bin/sh`, 并以该 *filename* 作为 shell 的输入。

第二个区别与参数表的传递有关 (*l* 表示表 (list), *v* 表示矢量 (vector))。函数 `execl`、`execlp` 和 `execle` 要求将新程序的每个命令行参数都说明为一个单独的参数。这种参数表以空指针结尾。对于另外三个函数 (`execv`、`execvp` 和 `execve`), 则应先构造一个指向各参数的指针数组, 然后将该数组地址作为这三个函数的参数。

在使用 ANSI C 原型之前, 对 `execl`、`execle` 和 `execlp` 三个函数表示命令行参数的一般方法是:

```
char *arg0, char *arg1, ..., char *argn, (char *) 0
```

应当特别指出的是: 在最后一个命令行参数之后跟了一个空指针。如果用常数 0 来表示一个空指针, 则必须将它强制转换为一个字符指针, 否则它将被解释为整型参数。如果一个整型数的长度与 `char *` 的长度不同, `exec` 函数实际参数就将出错。

最后一个区别与向新程序传递环境表相关。以 `e` 结尾的两个函数 (`execle` 和 `execve`) 可以传递一个指向环境字符串指针数组的指针。其他四个函数则使用调用进程中的 `environ` 变量为新程序复制现存的环境。(回忆 7.9 节及表 7-2 中对环境字符串的讨论。其中曾提及如果系统支持 `setenv` 和 `putenv` 这样的函数, 则可更改当前环境和后面生成的子进程的环境, 但不能影响父进程的环境。)通常, 一个进程允许将其环境传播给其子进程, 但有时也有这种情况, 进程想要为子进程指定一个确定的环境。例如, 在初始化一个新登录的 shell 时, `login` 程序创建一个只定义少数几个变量的特殊环境, 而在我们登录时, 可以通过 shell 起动文件, 将其他变量加到环境中。在使用 ANSI C 原型之前, `execle` 的参数是:

```
char *pathname, char *arg0, ..., char *argn, (char *)0, charenvp[]
```

从中可见, 最后一个参数是指向环境字符串的各字符指针构成的数组的指针。而在 ANSI C 原型中, 所有命令行参数, 包括空指针, `envp` 指针都用省略号 (...) 表示。

这六个 `exec` 函数的参数很难记忆。函数名中的字符会给我们一些帮助。字母 `p` 表示该函数取 *filename* 作为参数, 并且用 PATH 环境变量寻找可执行文件。字母 `l` 表示该函数取一个参数表, 它与字母 `v` 互斥。`v` 表示该函数取一个 `argv[]`。最后, 字母 `e` 表示该函数取 `envp[]` 数组, 而不使用当前环境。表 8-4 显示了这六个函数之间的区别。

表8-4 六个exec函数之间的区别

函 数	<i>pathname</i>	<i>filename</i>	参 数 表	<i>argv[]</i>	<i>environ</i>	<i>envp[]</i>
execl	•		•		•	
execlp		•	•		•	
execle	•		•			•
execv	•			•	•	
execvp		•		•	•	
execve	•			•		•
(字母表示)		p	l	v		e

每个系统对参数表和环境表的总长度都有一个限制。在表 2-7中，这种限制是ARG_MAX。在POSIX.1系统中，此值至少是4096字节。当使用shell的文件名扩充功能产生一个文件名表时，可能会受到此值的限制。例如，命令

```
grep _POSIX_SOURCE /usr/include/*/*.h
```

在某些系统上可能产生下列形式的shell错误：

```
arg list too long
```

由于历史原因，系统V中此限制是5120字节。4.3BSD和4.3+BSD在分发时此限制是20 480字节。作者所用的系统则允许多至1M字节（见程序2-1的输出）！

前面曾提及在执行exec后，进程ID没有改变。除此之外，执行新程序的进程还保持了原进程的下列特征：

- 进程ID和父进程ID。
- 实际用户ID和实际组ID。
- 添加组ID。
- 进程组ID。
- 对话期ID。
- 控制终端。
- 闹钟尚余留的时间。
- 当前工作目录。
- 根目录。
- 文件方式创建屏蔽字。
- 文件锁。
- 进程信号屏蔽。
- 未决信号。
- 资源限制。
- tms_utime, tms_stime, tms_cutime以及tms_ustime值。

对打开文件的处理与每个描述符的exec关闭标志值有关。见图 3-1以及3.13节中对FD_CLOEXEC的说明，进程中每个打开描述符都有一个exec关闭标志。若此标志设置，则在执行exec时关闭该描述符，否则该描述符仍打开。除非特地用fcntl设置了该标志，否则系统的默认操作是在exec后仍保持这种描述符打开。

POSIX.1明确要求在exec时关闭打开目录流（见4.21节中所述的opendir函数）。这通常是由opendir函数实现的，它调用fcntl函数为对应于打开目录流的描述符设置exec关闭标志。

注意，在exec前后实际用户ID和实际组ID保持不变，而有效ID是否改变则取决于所执行程序的文件的设置-用户-ID位和设置-组-ID位是否设置。如果新程序的设置-用户-ID位已设置，则有效用户ID变成程序文件所有者的ID，否则有效用户ID不变。对组ID的处理方式与此相同。

在很多UNIX实现中，这六个函数中只有一个execve是内核的系统调用。另外五个只是库函数，它们最终都要调用系统调用。这六个函数之间的关系示于图8-2中。在这种安排中，库函数execlp和execvp使用PATH环境变量查找第一个包含名为filename的可执行文件的路径名前缀。

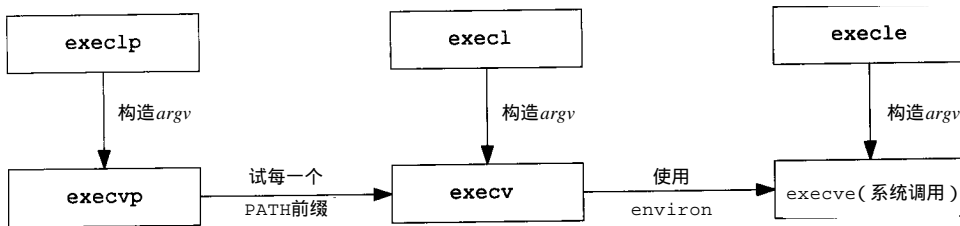


图8-2 六个exec函数之间的关系

实例

程序8-8例示了exec函数。

程序8-8 exec函数实例

```

#include <sys/types.h>
#include <sys/wait.h>
#include "ourhdr.h"

char *env_init[] = { "USER=unknown", "PATH=/tmp", NULL };

int
main(void)
{
    pid_t pid;

    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0) { /* specify pathname, specify environment */
        if (execl("/home/stevens/bin/echoall",
                 "echoall", "myarg1", "MY ARG2", (char *) 0,
                 env_init) < 0)
            err_sys("execl error");
    }
    if (waitpid(pid, NULL, 0) < 0)
        err_sys("wait error");

    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0) { /* specify filename, inherit environment */
        if (execlp("echoall",
                  "echoall", "only 1 arg", (char *) 0) < 0)
            err_sys("execlp error");
    }
    exit(0);
}

```

在该程序中先调用 `execl`，它要求一个路径名和一个特定的环境。下一个调用的是 `execlp`，它用一个文件名，并将调用者的环境传送给新程序。`execlp`在这里能够工作的原因是因为目录 `/home/stevens/bin` 是当前路径前缀之一。注意，我们将第一个参数（新程序中的 `argv[0]`）设置为路径名的文件名分量。某些 shell 将此参数设置为完全的路径名。

在程序 8-8 中要执行两次的程序 `echoall` 示于程序 8-9 中。这是一个普通程序，它回送其所有命令行参数及其全部环境表。

程序 8-9 回送所有命令行参数和所有环境字符串

```
#include    "ourhdr.h"

int
main(int argc, char *argv[])
{
    int        i;
    char       **ptr;
    extern char **environ;

    for (i = 0; i < argc; i++)    /* echo all command-line args */
        printf("argv[%d]: %s\n", i, argv[i]);

    for (ptr = environ; *ptr != 0; ptr++)    /* and all env strings */
        printf("%s\n", *ptr);

    exit(0);
}
```

执行程序 8-8 时得到：

```
$ a.out
argv[0]: echoall
argv[1]: myarg1
argv[2]: MY ARG2
USER=unknown
PATH=/tmp
argv[0]: echoall
$ argv[1]: only 1 arg
USER=stevens
HOME=/home/stevens
LOGNAME=stevens
```

其中 31 行没有显示

```
EDITOR=/usr/ucb/vi
```

注意，shell 提示出现在第二个 `exec` 打印 `argv[0]` 和 `argv[1]` 之间。这是因为父进程并不等待该子进程结束。

8.10 更改用户 ID 和组 ID

可以用 `setuid` 函数设置实际用户 ID 和有效用户 ID。与此类似，可以用 `setgid` 函数设置实际组 ID 和有效组 ID。

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

int setuid(uid_t uid);
```



```
int setgid(gid_t gid);
```

两个函数返回：若成功则为0，若出错则为-1

关于谁能更改ID有若干规则。现在先考虑有关改变用户ID的规则（在这里关于用户ID所说明的一切都适用于组ID）。

(1) 若进程具有超级用户特权，则setuid函数将实际用户ID、有效用户ID，以及保存的设置-用户-ID设置为uid。

(2) 若进程没有超级用户特权，但是uid等于实际用户ID或保存的设置-用户-ID，则setuid只将有效用户ID设置为uid。不改变实际用户ID和保存的设置-用户-ID。

(3) 如果上面两个条件都不满足，则errno设置为EPERM，并返回出错。

在这里假定_POSIX_SAVED_IDS为真。如果没有提供这种功能，则上面所说的关于保存的设置-用户-ID部分都无效。

FIPS 151-1要求此功能。
SVR4支持_POSIX_SAVED_IDS功能。

关于内核所维护的三个用户ID，还要注意下列几点：

(1) 只有超级用户进程可以更改实际用户ID。通常，实际用户ID是在用户登录时，由login(1)程序设置的，而且决不会改变它。因为login是一个超级用户进程，当它调用setuid时，设置所有三个用户ID。

(2) 仅当对程序文件设置了设置-用户-ID位时，exec函数设置有效用户ID。如果设置-用户-ID位没有设置，则exec函数不会改变有效用户ID，而将其维持为原先值。任何时候都可以调用setuid，将有效用户ID设置为实际用户ID或保存的设置-用户-ID。自然，不能将有效用户ID设置为任一随机值。

(3) 保存的设置-用户-ID是由exec从有效用户ID复制的。在exec按文件用户ID设置了有效用户ID后，即进行这种复制，并将此副本保存起来。

表8-5列出了改变这三个用户ID的不同方法。

表8-5 改变三个用户ID的不同方法

ID	exec		setuid(uid)	
	设置-用户-ID位关闭	设置-用户-ID位打开	超级用户	非特权用户
实际用户ID	不变	不变	设为uid	不变
有效用户ID	不变	设置为程序文件的用户ID	设为uid	设为uid
保存的设置-用户-ID	从有效用户ID复制	从有效用户ID复制	设为uid	不变

注意，用8.2节中所述的getuid和geteuid函数只能获得实际用户ID和有效用户ID的当前值。我们不能获得所保存的设置-用户-ID的当前值。

实例

为了说明保存的设置-用户-ID特征的用法，先观察一个使用该特征的程序。我们所观察的是伯克利tip(1)程序(系统V的cu(1)程序与此类似)。这两个程序都连接到一个远程系统，或者是直接连接，或者是拨号一个调制解调器。当tip使用调制解调器时，它必须通过使用锁文件来独

占使用它。此锁文件与UUCP程序共享，因为这两个程序可能要同时使用同一调制解调器。对其工作步骤说明如下：

(1) tip程序文件是由用户uucp拥有的，并且其设置-用户-ID位已设置。当exec此程序时，则关于用户ID得到下列结果：

实际用户ID = 我们的用户ID

有效用户ID = uucp

保存设置-用户-ID = uucp

(2) tip存取所要求的锁文件。这些锁文件是由名为uucp的用户所拥有的，因为有效用户ID是uucp，所以tip可以存取这些锁文件。

(3) tip执行setuid(getuid())。因为tip不是超级用户进程，所以这仅仅改变有效用户ID。此时得到：

实际用户ID = 我们的用户ID(未改变)

有效用户-ID = 我们的用户ID(未改变)

保存设置-用户-ID = uucp(未改变)

现在，tip进程是以我们的用户ID作为其有效用户ID而运行的。这就意味着能存取的只有我们通常可以存取的，没有额外的许可权。

(4) 当执行完所需的操作后，tip执行setuid(*uucpuid*)，其中*uucpuid*是用户uucp的数值用户ID (tip很可能在起动时调用geteuid，得到uucp的用户ID，然后将其保存起来，我们并不认为tip会搜索口令文件以得到这一数值用户ID)。因为setuid的参数等于保存的设置-用户-ID，所以这种调用是许可的(这就是为什么需要保存的设置-用户-ID的原因)。现在得到：

实际用户ID=我们的用户ID(未改变)

有效用户ID=uucp

保存设置-用户-ID=uucp(未改变)

(5) tip现在可对其锁文件进行操作以释放它们，因为tip的有效用户ID是uucp。

以这种方法使用保存的设置-用户-ID，在进程的开始和结束部分就可以使用由于程序文件的设置用户ID而得到的额外优先权。但是，进程在其运行的大部分时间只具有普通的许可权。如果进程不能在其结束部分切换回保存的设置-用户-ID，那么就不得不在全部运行时间都保持额外的许可权(这可能会造成麻烦)。

下面来看一看如果在tip运行时为我们生成一个shell进程(先fork，然后exec)将发生什么。因为实际用户ID和有效用户ID都是我们的普通用户ID(上面的第(3)步)，所以该shell没有额外的许可权。它不能存取tip运行时设置成uucp的保存的设置-用户-ID，因为该shell所保存的设置-用户-ID是由exec复制有效用户ID而得到的。所以在执行exec的子进程中，所有三个用户ID都是我们的普通用户ID。

如果程序是设置-用户-ID为root,那么我们关于tip如何使用setuid所做的说明是不正确的。因为以超级用户特权调用setuid就会设置所有三个用户ID。使上述实例按我们所说明的进行工作，只需setuid设置有效用户ID。

8.10.1 setreuid和setregid函数

4.3+BSD支持setregid函数，其功能是交换实际用户ID和有效用户ID的值。

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

```
#include <unistd.h>

int setreuid(uid_t ruid, uid_t euid);

int setregid(gid_t rgid, gid_t egid);
```

两个函数返回：若成功则为0，若出错则为-1

其作用很简单：一个非特权用户总能交换实际用户 ID 和有效用户 ID。这就允许一个设置-用户-ID 程序转换成只具有用户的普通许可权，以后又可再次转换回设置-用户-ID 所得到的额外许可权。POSIX.1 引进了保存的设置-用户-ID 特征后，其作用也相应加强，它也允许一个非特权用户将其有效用户 ID 设置为保存的设置-用户-ID。

SVR4 在其 BSD 兼容库中也提供这两个函数。

4.3BSD 并没有上面所说的保存的设置-用户-ID 功能。它用 `setreuid` 和 `setregid` 来代替。这就允许一个非特权用户前、后交换这两个用户 ID 的值，而 4.3BSD 中的 `tip` 程序就是用这种功能编写的。但是要知道，当此版本生成 shell 进程时，它必须在 `exec` 之前，先将实际用户 ID 设置为普通用户 ID。如果不这样做的话，那么实际用户 ID 就可能是 `uucp`（由 `setreuid` 的交换操作造成。），然后 shell 进程可能会调用 `setreuid` 交换两个用户 ID 值并取得 `uucp` 许可权。作为一个保护性的程序设计措施，`tip` 将子进程的实际用户 ID 和有效用户 ID 都设置成普通用户 ID。

8.10.2 seteuid和setegid函数

在对 POIX.1 的建议更改中包含了两个函数 `seteuid` 和 `setegid`。它们只更改有效用户 ID 和有效组 ID。

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

int seteuid(uid_t uid);

int setegid(gid_t gid);
```

两个函数返回：若成功则为0，若出错则为-1

一个非特权用户可将其有效用户 ID 设置为其实际用户 ID 或其保存的设置-用户-ID。对于一个特权用户则可将有效用户 ID 设置为 `uid`。（这区别于 `setuid` 函数，它更改三个用户 ID。）这一建议更改也要求支持保存的设置-用户-ID。

SVR4 和 4.3+BSD 都支持这两种函数。

图8-3给出了本节所述的修改三个不同用户 ID 的各个函数。

8.10.3 组ID

本章中所说明的一切都以类似方式适用于各个组 ID。添加组 ID 不受 `setgid` 函数的影响。

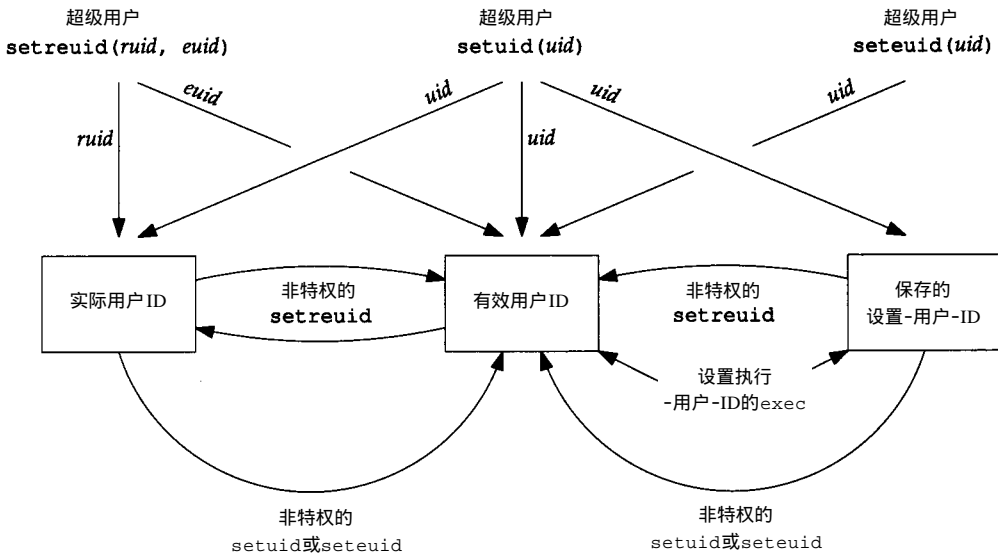


图8-3 设置不同的用户ID的各函数

8.11 解释器文件

SVR4和4.3+BSD都支持解释器文件。这种文件是文本文件，其起始行的形式是：

```
#! pathname [optional-argument]
```

在惊叹号和pathname之间的空格是可任选的。最常见的是以下列行开始：

```
#!/bin/sh
```

pathname通常是个绝对路径名，对它不进行什么特殊的处理（不使用PATH进行路径搜索）。对这种文件的识别是由内核作为exec系统调用处理的一部分来完成的。内核使调用exec函数的进程实际执行的文件并不是该解释器文件，而是在该解释器文件的第一行中pathname所指定的文件。一定要将解释器文件（文本文件，它以#!开头）和解释器（由该解释器文件第一行中的pathname指定）区分开来。

很多系统对解释器文件第一行有长度限制（32个字符）。这包括#!、pathname、可选参数以及空格数。

实例

让我们观察一个实例，从中了解当被执行的文件是个解释器文件时，内核对exec函数的参数及该解释器文件第一行的可选参数做何种处理。程序8-10调用exec执行一个解释器文件。

程序8-10 执行一个解释器文件的程序

```
#include <sys/types.h>
#include <sys/wait.h>
#include "ourhdr.h"

int
main(void)
{
    pid_t pid;
```

```

if ( (pid = fork()) < 0)
    err_sys("fork error");
else if (pid == 0) {
    /* child */
    if (execl("/home/stevens/bin/testinterp",
            "testinterp", "myarg1", "MY ARG2", (char *) 0) < 0)
        err_sys("execl error");
}

if (waitpid(pid, NULL, 0) < 0) /* parent */
    err_sys("waitpid error");
exit(0);
}

```

下面先显示要被执行的该解释器文件（只有一行）的内容，接着是运行程序 8-10 的结果。

```

$ cat /home/stevens/bin/testinterp
#!/home/stevens/bin/echoarg foo
$ a.out
argv[0]: /home/stevens/bin/echoarg
argv[1]: foo
argv[2]: /home/stevens/bin/testinterp
argv[3]: myarg1
argv[4]: MY ARG2

```

程序 `echoarg`（解释器）回送每一个命令行参数（它就是程序 7-2）。注意，当内核 `exec` 该解释器（`/home/stevens/bin/echoarg`）时，`argv[0]` 是该解释器的 *pathname*，`argv[1]` 是解释器文件中的可选参数，其余参数是 *pathname*（`/home/stevens/bin/testinterp`），以及程序 8-10 中调用 `execl` 的第二和第三个参数（`myarg1` 和 `MY ARG2`）。调用 `execl` 时的 `argv[1]` 和 `argv[2]` 已右移了两个位置。注意，内核取 `execl` 中的 *pathname* 代替第一个参数（`testinterp`），因为一般 *pathname* 包含了较第一个参数更多的信息。

实例

在解释器 *pathname* 后可跟随可选参数，它们常用于为支持 `-f` 选择项的程序指定该选择项。例如，可以以下列方式执行 `awk(1)` 程序：

```
awk -f myfile
```

它告诉 `awk` 从文件 `myfile` 中读 `awk` 程序。

在很多系统中，有 `awk` 的两个版本。`awk` 常常被称为“老 `awk`”，它是与 V7 一起分发的原始版本。`nawk`（新 `awk`）包含了很多增强功能，对应于在 Aho、Kernighan 和 Weinberger [1988] 中说明的语言。此新版本提供了对命令行参数的存取，这是下面的例子所需的。SVR4 提供了两者，老的 `awk` 既可用 `awk` 也可用 `oawk` 调用，但是 SVR4 已说明在将来的版本中 `awk` 将是 `nawk`。POSIX.2 将新 `awk` 语句就称为 `awk`，这正是本书所使用的。

在解释器文件中使用 `-f` 选择项，可以写成：

```
#!/bin/awk -f
（在解释器文件中后随 awk 程序）
```

例如，程序 8-11 是一个在 `/usr/local/bin/awkexample` 解释器文件中的程序。

程序8-11 解释器文件中的awk程序

```
#!/bin/awk -f
BEGIN {
    for (i = 0; i < ARGV; i++)
        printf "ARGV[%d] = %s\n", i, ARGV[i]
    exit
}
```

如果路径前缀之一是 `/usr/local/bin`，则可以下列方式执行程序 8-11(假定我们已打开了该文件的执行位)：

```
$ awkexample file1 FILENAME2 f3
ARGV[0] = /bin/awk
ARGV[1] = file1
ARGV[2] = FILENAME2
ARGV[3] = f3
```

执行 `/bin/awk` 时，其命令行参数是：

```
/bin/awk -f /usr/local/bin/awkexample file1 FILENAME2 f3
```

解释器文件的路径名 (`/usr/local/bin/awkexample`) 被传送给解释器。因为不能期望该解释器 (在本例中是 `/bin/awk`) 会使用 `PATH` 变量定位该解释器文件，所以只传送其路径名中的文件名是不够的。当 `awk` 读解释器文件时，因为 `#` 是 `awk` 的注释字符，所以在 `awk` 读解释器文件时，它忽略第一行。

可以用下列命令验证上述命令行参数。

```
$ su                                成为超级用户
Password:                            输入超级用户口令
# mv /bin/awk /bin/awk.save          保存原先的程序
# cp /home/stevens/bin/echoarg /bin/awk 暂时替换它
# suspend                             用作业控制挂起超级用户 shell
[1] + Stopped                        su
$ awkexample file1 FILENAME2 f3
argv[0]: /bin/awk
argv[1]: -f
argv[2]: /usr/local/bin/awkexample
argv[3]: file1
argv[4]: FILENAME2
argv[5]: f3
$ fg                                  用作业控制恢复超级用户 shell
su
# mv /bin/awk.save /bin/awk          恢复原先的程序
# exit                                终止超级用户 shell
```

在此例子中，解释器的 `-f` 选择项是必需的。正如前述，它告诉 `awk` 在什么地方得到 `awk` 程序。如果在解释器文件中删除 `-f` 选择项，则其结果是：

```
$ awkexample file1 FILENAME2 f3
/bin/awk:syntax error at source line 1
context is
    >>> /usr/local <<< /bin/awkexample
```

```
/bin/awk: bailing out at source line 1
```

因为在这种情况下命令行参数是：

```
/bin/awk /usr/local/bin/awkexample file1 FILENAME2 f3
```

于是awk企图将字符串/usr/local/bin/awkexample解释为一个awk程序。如果不能向解释器传递至少一个可选参数（在本例中是-f），那么这些解释器文件只有对shell才是有用的。

是否一定需要解释器文件呢？那也不完全如此。但是它们确实使用户得到效率方面的好处，其代价是内核的额外开销（因为内核需要识别解释器文件）。由于下述理由，解释器文件是有用的：

(1) 某些程序是用某种语言写的脚本，这一事实可以隐藏起来。例如，为了执行程序 8-11，只需使用下列命令行：

```
awkexampleoptional-arguments
```

而并不需要知道该程序实际上是一个awk脚本，否则就要以下列方式执行该程序：

```
awk -f awkexampleoptional-arguments
```

(2) 解释器脚本在效率方面也提供了好处。再考虑一下前面的例子。仍旧隐藏该程序是一个awk脚本的事实，但是将其放在一个shell脚本中：

```
awk 'BEGIN {
    for (i = 0; i < ARGV; i++)
        printf "ARGV[%d] = %s\n", i, ARGV[i]
    exit
}' $*
```

这种解决方法的问题是要求做更多的工作。首先，shell读此命令，然后试图execlp此文件名。因为shell脚本是一个可执行文件，但却不是机器可执行的，于是返回一个错误，execlp就认为该文件是一个shell脚本（它实际上就是这种文件）。然后执行/bin/sh，并以该shell脚本的路径名作为其参数。shell正确地执行我们的shell脚本，但是为了运行awk程序，它调用fork、exec和wait。用一个shell脚本代替解释器脚本需要更多的开销。

(3) 解释器脚本使我们可以使用除/bin/sh以外的其他shell来编写shell脚本。当execlp找到一个非机器可执行的可执行文件时，它总是调用/bin/sh来解释执行该文件。但是，用解释器脚本，则可编写成：

```
#!/bin/csh
(在解释器文件中后随C shell脚本)
```

再一次，我们也可将此放在一个/bin/sh脚本中(然后由其调用C shell)，但是要有更多的开销。

如果三个shell和awk没有用#作为注释符，则上面所说的都无效。

8.12 system函数

在程序中执行一个命令字符串很方便。例如，假定要将时间和日期放到一个文件中，则可使用6.9节中的函数实现这一点。调用time得到当前日历时间，接着调用localtime将日历时间变换为年、月、日、时、分、秒、周日形式，然后调用strftime对上面的结果进行格式化处理，最后将结果写到文件中。但是用下面的system函数则更容易做到这一点。

```
system("date > file");
```

ANSI C定义了system函数，但是其操作对系统的依赖性很强。

因为system不属于操作系统界面而是 shell界面，所以POSIX.1没有定义它，POSIX.2则正在对其进行标准化。下列说明与POSIX.2标准的草案11.2相一致。

```
#include <stdlib.h>

int system(const char*cmdstring);
```

返回：(见下)

如果cmdstring是一个空指针，则仅当命令处理程序可用时，system返回非0值，这一特征可以决定在一个给定的操作系统上是否支持system函数。在UNIX中，system总是可用的。

因为system在其实现中调用了fork、exec和waitpid，因此有三种返回值：

(1) 如果fork失败或者waitpid返回除EINTR之外的出错，则system返回-1，而且errno中设置了错误类型。

(2) 如果exec失败(表示不能执行shell)，则其返回值如同shell执行了exit(127)一样。

(3) 否则所有三个函数(fork,exec和waitpid)都成功，并且system的返回值是shell的终止状态，其格式已在waitpid中说明。

如果waitpid由一个捕捉到的信号中断，则system很多当前的实现都返回一个错误(EINTR)，在这种情况下system不返回一个错误的要求已被加到POSIX.2的最近草案中。(10.5节中将讨论被中断的系统调用。)

程序8-12是system函数的一种实现。它对信号没有进行处理。10.18节中将修改此函数使其进行信号处理。

shell的-c选择项告诉shell程序取下一个命令行参数(在这里是cmdstring)作为命令输入(而不是从标准输入或从一个给定的文件中读命令)。shell对以null字符终止的命令字符串进行语法分析，将它们分成分隔开的命令行参数。传递给shell的实际命令串可以包含任一有效的shell命令。例如，可以用<和>对输入和输出重新定向。

如果不使用shell执行此命令，而是试图由我们自己去执行它，那么将相当困难。首先，我们必须用execlp而不是execl，像shell那样使用PATH变量。我们必须将null符结尾的命令字符串分成各个命令行参数，以便调用execlp。最后，我们也不能使用任何一个shell元字符。

注意，我们调用_exit而不是exit。这是为了防止任一标准I/O缓存(这些缓存会在fork中由父进程复制到子进程)在子进程中被刷新。

程序8-12 system函数(没有对信号进行处理)

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

int
system(const char *cmdstring) /* version without signal handling */
{
    pid_t pid;
```



```

int    status;

if (cmdstring == NULL)
    return(1);    /* always a command processor with Unix */

if ( (pid = fork()) < 0) {
    status = -1;    /* probably out of processes */
} else if (pid == 0) {    /* child */
    execl("/bin/sh", "sh", "-c", cmdstring, (char *) 0);
    _exit(127);    /* execl error */
} else {    /* parent */
    while (waitpid(pid, &status, 0) < 0)
        if (errno != EINTR) {
            status = -1; /* error other than EINTR from waitpid() */
            break;
        }
}

return(status);
}

```

用程序8-13对这种实现的system函数进行测试(pr_exit函数定义在程序8-3中)。运行程序8-13得到：

```

$ a.out
Thu Aug 29 14:24:19 MST 1991
normal termination, exit status = 0 对于date
sh: nosuchcommand: not found
normal termination, exit status = 1 对于无此种命令
stevens console Aug 25 11:49
stevens tty0 Aug 29 05:56
stevens tty1 Aug 29 05:56
stevens tty2 Aug 29 05:56
normal termination, exit status = 44 对于exit

```

程序8-13 调用system函数

```

#include <sys/types.h>
#include <sys/wait.h>
#include "ourhdr.h"

int
main(void)
{
    int    status;

    if ( (status = system("date")) < 0)
        err_sys("system() error");
    pr_exit(status);

    if ( (status = system("nosuchcommand")) < 0)
        err_sys("system() error");
    pr_exit(status);

    if ( (status = system("who; exit 44")) < 0)
        err_sys("system() error");
    pr_exit(status);

    exit(0);
}

```

使用system而不是直接使用fork和exec的优点是：system进行了所需的各种出错处理，以及各种信号处理(在10.18节中的下一个版本system函数中)。

在UNIX的早期版本中，包括SVR3.2和4.3BSD，都没有waitpid函数，于是父进程用下列形式的语句等待子进程：

```
while ((lastpid = wait(&status)) != pid && lastpid != -1)
    ;
```

如果调用system的进程在调用它之前已经生成一个子进程（并执行一个程序），那么将引起问题。因为上面的while语句一直循环执行，直到由system产生的子进程终止才停止，如果其任意一个不是用pid标识的子进程在此之前终止，则它们的进程ID和终止状态都被while语句丢弃。实际上，由于wait不能等待一个指定的进程，POSIX.1才为此及其他一些原因定义了waitpid函数。如果不提供waitpid，对于popen和pclose函数也会发生同样的问题。

设置-用户-ID程序

如果在一个设置-用户-ID程序中调用system，那么发生什么呢？这是一个安全性方面的漏洞，决不当这样做。程序8-14是一个简单程序，它只是对其命令行参数调用system函数。

程序8-14 用system执行命令行参数

```
#include    "ourhdr.h"

int
main(int argc, char *argv[])
{
    int    status;

    if (argc < 2)
        err_quit("command-line argument required");

    if ( (status = system(argv[1])) < 0)
        err_sys("system() error");
    pr_exit(status);

    exit(0);
}
```

将此程序编译成可执行目标文件tsys。

程序8-15是另一个简单程序，它打印其实际和有效用户ID。

程序8-15 打印实际和有效用户ID

```
#include    "ourhdr.h"

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

将此程序编译成可执行目标文件printuids。运行这两个程序，得到下列结果：

```
$ tsys printuids          正常执行，无特权
real uid = 224, effective uid = 224
```

```

normal termination, exit status = 0
$ su                    成为超级用户
Password:              输入超级用户口令
# chown root tsys      更改所有者
# chmod u+s tsys       增加设置-用户-ID
# ls -l tsys           检验文件许可权和所有者
-rwsrwxr-x  1 root    105737 Aug 18 11:21 tsys
# exit                 终止超级用户 shell
$ tsys printuids
real uid = 224, effective uid = 0哎呀！这是一个安全性漏洞
normal termination, exit status = 0

```

我们给予tsys程序的超级用户许可权在system中执行了fork和exec之后仍被保持下来，也就是说执行system中shell命令的进程也具有了超级用户许可权。

如果一个进程正以特殊的许可权(设置-用户-ID或设置-组-ID)运行，它又想生成另一个进程执行另一个程序，则它应当直接使用fork和exec，而且在fork之后、exec之前要改回到普通许可权。设置-用户-ID或设置-组-ID程序决不应调用system函数。

这种警告的一个理由是：system调用shell对命令字符串进行语法分析，而shell则使用IFS变量作为其输入字段分隔符。早期的shell版本在被调用时不将此变量恢复为普通字符集。这就允许一个不怀好意的用户在调用system之前设置IFS，造成system执行一个不同的程序。

8.13 进程会计

很多UNIX系统提供了一个选择项以进行进程会计事务处理。当取了这种选择项后，每当进程结束时内核就写一个会计记录。典型的会计记录是32字节长的二进制数据，包括命令名、所使用的CPU时间总量、用户ID和组ID、起动时间等。本节将比较详细地说明这种会计记录，这样也使我们得到了一个再次观察进程的机会，得到了使用5.9节中所介绍的fread函数的机会。

任一标准都没有对进程会计进行过说明。本节的说明依据SVR4和4.3+BSD实现。SVR4提供了很多程序处理这种原始的会计数据——例如runacct和acctcom。4.3+BSD提供sa(8)命令处理并总结原始会计数据。

一个至今没有说明过的函数(acct)起动和终止进程会计。唯一使用这一函数的是SVR4和4.3+BSD的accton(8)命令。超级用户执行一个带路径名参数的accton命令起动会计处理。该路径名通常是/var/adm/pacct(早期系统中为/usr/adm/acct)。执行不带任何参数的accton命令则停止会计处理。

会计记录结构定义在头文件<sys/acct.h>中，其样式如下：

```

typedef u_short comp_t; /* 3-bit base 8 exponent; 13-bit fraction */
struct acct
{
    char    ac_flag; /* flag (see Figure 8.9).*/
    char    ac_stat; /* termination status (signal & core flag only) */
             /* (not provided by BSD systems) */
    uid_t   ac_uid; /* real user ID */

```

```

gid_t   ac_gid;   /* real group ID */
dev_t   ac_tty;   /* controlling terminal */
time_t  ac_btime; /* starting calendar time */
comp_t  ac_utime; /* user CPU time (clock ticks) */
comp_t  ac_stime; /* system CPU time (clock ticks) */
comp_t  ac_etime; /* elapsed time (clock ticks) */
comp_t  ac_mem;   /* average memory usage */
comp_t  ac_io;    /* bytes transferred (by read and write) */
comp_t  ac_rw;    /* blocks read or written */
char    ac_comm[8]; /* command name: [8] for SVR4, [10] for 4.3+BSD */
};

```

由于历史原因，伯克利系统，包括4.3+BSD都不提供ac_stat变量。

其中，ac_flag记录了进程执行期间的某些事件。这些事件见表8-6。

表8-6 会计记录中的ac_flag值

ac_flag	说 明
AFORK	进程是由fork产生的，但从未调用exec
ASU	进程使用超级用户优先权
ACOMPAT	进程使用兼容方式（仅VAX）
ACORE	进程转储core（不在SVR4）
AXSIG	进程由信号消灭（不在SVR4）

会计记录所需的各个数据（各CPU时间、传输的字符数等）都由内核保存在进程表中，并在一个新进程被创建时置初值（例如fork之后在子进程中）。进程终止时写一个会计记录。这就意味着在会计文件中记录的顺序对应于进程终止的顺序，而不是它们起动的顺序。为了确定起动顺序，需要读全部会计文件，并按起动日历时间进行排序。这不是一种很完善的方法，因为日历时间的单位是秒（见1.10节），在一个给定的秒中可能起动了多个进程。而墙上时钟时间的单位是时钟滴答（通常，每秒滴答数在50~100之间）。但是我们并不知道进程的终止时间，所知道的只是起动时间和终止顺序。这就意味着，即使墙上时间比起动时间要精确得多，但是仍不能按照会计文件中的数据重构各进程的精确起动顺序。

会计记录对应于进程而不是程序。在fork之后，内核为子进程初始化一个记录，而不是在一个新程序被执行时。虽然exec并不创建一个新的会计记录，但相应记录中的命令名改变了，AFORK标志则被清除。这意味着，如果一个进程顺序执行了三个程序A exec B, B exec C, 最后C exit)，但只写一个会计记录。在该记录中的命令名对应于程序C，但CPU时间是程序A、B、C之和。

实例

为了得到某些会计数据以便查看，运行程序8-16，它调用fork四次。每个子进程做不同的事情，然后终止。此程序所做的基本工作示于图8-4中。

程序8-17则从会计记录中选择一些字段并打印出来。

程序8-16 产生会计数据的程序

```

#include <signal.h>
#include "ourhdr.h"

int
main(void)

```

```

{
    pid_t  pid;

    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid != 0) {          /* parent */
        sleep(2);
        exit(2);                 /* terminate with exit status 2 */
    }

                                /* first child */
    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid != 0) {
        sleep(4);
        abort();                 /* terminate with core dump */
    }

                                /* second child */
    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid != 0) {
        execl("/usr/bin/dd", "dd", "if=/boot", "of=/dev/null", NULL);
        exit(7);                 /* shouldn't get here */
    }

                                /* third child */
    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid != 0) {
        sleep(8);
        exit(0);                 /* normal exit */
    }

                                /* fourth child */
    sleep(6);
    kill(getpid(), SIGKILL);     /* terminate with signal, no core dump */
    exit(6);                     /* shouldn't get here */
}

```

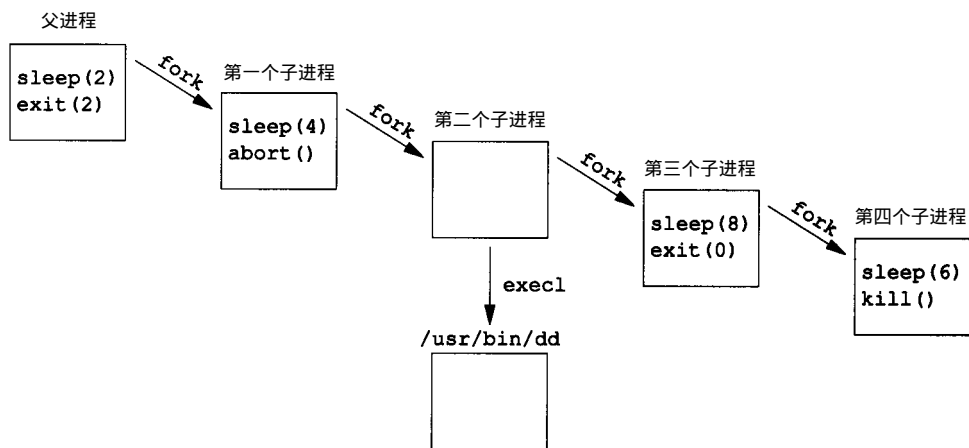


图8-4 会计处理实例的进程结构

然后，执行下列操作步骤：

(1) 成为超级用户，用accton命令起动会计事务处理。注意，当此命令结束时，会计事务处

理已经起动，因此在会计文件中的第一个记录应来自这一命令。

(2) 运行程序8-16。这会加五个记录到会计文件中(父进程一个，四个子进程各一个)。在第二个子进程中，`execl`并不创建一个新进程，所以对第二个进程只有一个会计记录。

(3) 成为超级用户，停止会计事务处理。因为在`accton`命令终止时已停止处理会计事务，所以不会在会计文件中增加一个记录。

(4) 运行程序8-17，从打印文件中选出字段并打印。

第(4)步的输出如下面所示。在每一行中都对进程加了说明，以便后面讨论。

```
accton   e =      7, chars =      64, stat =   0:   S
dd       e =     37, chars =  221888, stat =   0:   第二个子进程
a.out    e =    128, chars =      0, stat =   0:   父进程
a.out    e =    274, chars =      0, stat = 134: F 第一个子进程
a.out    e =    360, chars =      0, stat =   9: F 第四个子进程
a.out    e =    484, chars =      0, stat =   0: F 第三个子进程
```

程序8-17 打印从系统会计文件中选出的字段

```
#include <sys/types.h>
#include <sys/acct.h>
#include "ourhdr.h"

#define ACCTFILE "/var/adm/pacct"
static unsigned long compt2ulong(comp_t);

int
main(void)
{
    struct acct    acdata;
    FILE          *fp;

    if ( (fp = fopen(ACCTFILE, "r")) == NULL)
        err_sys("can't open %s", ACCTFILE);
    while (fread(&acdata, sizeof(acdata), 1, fp) == 1) {
        printf("%-*.s e = %6ld, chars = %7ld, "
            "stat = %3u: %c %c %c %c\n", sizeof(acdata.ac_comm),
            sizeof(acdata.ac_comm), acdata.ac_comm,
            compt2ulong(acdata.ac_etime), compt2ulong(acdata.ac_io),
            (unsigned char) acdata.ac_stat,
#ifdef ACORE
            /* SVR4 doesn't define ACORE */
            acdata.ac_flag & ACORE ? 'D' : ' ',
#else
            ' ',
#endif
#ifdef AXSIG
            /* SVR4 doesn't define AXSIG */
            acdata.ac_flag & AXSIG ? 'X' : ' ',
#else
            ' ',
#endif
            acdata.ac_flag & AFORK ? 'F' : ' ',
            acdata.ac_flag & ASU ? 'S' : ' ');
    }
    if (ferror(fp))
        err_sys("read error");
    exit(0);
}

static unsigned long
compt2ulong(comp_t comptime) /* convert comp_t to unsigned long */
{
    unsigned long    val;
    int              exp;
}
```

```

val = comptime & 017777; /* 13-bit fraction */
exp = (comptime >> 13) & 7; /* 3-bit exponent (0-7) */
while (exp-- > 0)
    val *= 8;
return(val);
}

```

墙上日历时间值的单位是CLK_TCK。从表2-6中可见，本系统的值是60。例如，在父进程中的sleep(2)对应于墙上日历时间128个时钟滴答。对于第一个子进程，sleep(4)变成274时钟滴答。注意，一个进程睡眠的时间总量并不精确。（第10章将返回到sleep函数。）调用fork和exit也要一些时间。

注意，ac_stat并不是进程的真正终止状态。它只是8.6节中讨论的终止状态的一部分。如果进程异常终止，则此字节中的信息只是core标志位(一般是最高位)以及信号编号数(一般是低7位)。如果进程正常终止，则从会计文件不能得到进程的退出(exit)状态。对于第一个进程，此值是128+6。128是core标志位，6是此系统上信号SIGABRT的值(它是由调用abort产生的)。第四个子进程的值是9，它对应于SIGKILL的值。从会计文件的数据中不能了解到，父进程在退出时所用的参数值是2，三个子进程退出时所用的参数值是0。

dd进程复制到第二个子进程中的文件/boot的长度是110 888字节。而I/O字符数是此值的二倍，因为读了110 888字节，然后又写了110 888字节。即使输出到null设备，仍对I/O字符数进行计算。

ac_flag值与我们所预料的相同。除调用了execl的第二个子进程以外，其他子进程都设置了F标志。父进程没有设置F标志，其原因是交互式shell曾调用过fork生成父进程，然后执行a.out文件。调用了abort的第一个子进程的core转储标志(D)打开。因为abort产生信号SIGABRT以产生core转储。该进程的X标志也打开，因为它是由信号终止的。第四个子进程的X标志也打开，但是SIGKILL信号并不产生core转储，它只是终止该进程。

最后要说明的是：第一个子进程的I/O字符数为0，但是该进程产生了一个core文件。其原因是写core文件所需的I/O并不由该进程负担。

8.14 用户标识

任一进程都可以得到其实际和有效用户ID及组ID。但是有时希望找到运行该程序的用户的登录名。我们可以调用getpwuid(getuid())，但是如果一个用户有多个登录名，这些登录名又对应着同一个用户ID，那么又将如何呢？（一个人在口令文件中可以有多个登录项，它们的用户ID相同，但登录shell则不同。）系统通常保存用户的登录名（见6.7节），用getlogin函数可以存取此登录名。

```

#include <unistd.h>

char *getlogin(void);

```

返回：若成功则为指向登录名字符串的指针，若出错则为NULL

如果调用此函数的进程没有连接到用户登录时所用的终端，则本函数会失败。通常称这些进程为精灵进程(daemon)，第13章将对这种进程专门进行讨论。

得到了登录名，就可用getpwnam在口令文件中查找相应记录以确定其登录shell等。

为了找到登录名，UNIX系统在历史上一直是调用 `ttyname` 函数（见 11.9 节），然后在 `utmp` 文件（见 6.7 节）中找匹配项。4.3+BSD 将登录名存放在进程表项中，并提供系统调用存取该登录名。

系统 V 提供 `cuserid` 函数返回登录名。此函数先调用 `getlogin` 函数，如果失败则再调用 `getpwuid(getuid())`。IEEE Std.1003.1-1988 说明了 `cuserid`，但是它以有效用户 ID 而不是实际用户 ID 来调用。POSIX.1 的 1990 最后版本删除了 `cuserid` 函数。

FIPS 151-1 要求登录 shell 定义一个环境变量 `LOGNAME`，其值为用户的登录名。在 4.3+BSD 中，此变量由 `login` 设置，并由登录 shell 继承。但是，用户可以改变环境变量，所以不能使用 `LOGNAME` 来确认用户，而应当使用 `getlogin` 函数。

8.15 进程时间

在 1.10 节中说明了墙上时钟时间、用户 CPU 时间和系统 CPU 时间。任一进程都可调用 `times` 函数以获得它自己及终止子进程的上述值。

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

clock_t times(struct tms*);
```

返回：若成功则为经过的墙上时钟时间（单位：滴答），若出错则为 -1

此函数填写由 `buf` 指向的 `tms` 结构，该结构定义如下：

```
struct tms {
    clock_t tms_utime; /* user CPU time */
    clock_t tms_stime; /* system CPU time */
    clock_t tms_cutime; /* user CPU time, terminated children */
    clock_t tms_cstime; /* system CPU time, terminated children */
};
```

注意，此结构没有包含墙上时钟时间。作为代替，`times` 函数返回墙上时钟时间作为函数值。此值是相对于过去的某一时刻度量的，所以不能用其绝对值而必须使用其相对值。例如，调用 `times`，保存其返回值。在以后某个时间再次调用 `times`，从新返回的值中减去以前返回的值，此差值就是墙上时钟时间。（一个长期运行的进程可能其墙上时钟时间会溢出，当然这种可能性极小。）

结构中两个针对子进程的字段包含了此进程已等待到的各子进程的值。

所有由此函数返回的 `clock_t` 值都用 `_SC_CLK_TCK`（由 `sysconf` 函数返回的每秒时钟滴答数，见 2.5.4 节）转换成秒数。

伯克利系统，包括 4.3BSD 继承了 V7 的 `times` 版本，它不返回墙上时钟时间。这一老版本如执行成功则返回 0，如失败则返回 -1。4.3+BSD 支持 POSIX.1 版本。

4.3+BSD 和 SVR4（在 BSD 兼容库中）提供了 `getrusage(2)` 函数，此函数返回 CPU 时间，以及指示资源使用情况的另外 14 个值。

实例

程序 8-18 将每个命令行参数作为 shell 命令串执行，对每个命令计时，并打印从 `tms` 结构取

得的值。按下列方式运行此程序，得到：

```
$ a.out "sleep 5" "date"

command: sleep 5
  real:    5.25
  user:    0.00
  sys:     0.00
  child user:    0.02
  child sys:     0.13
normal termination, exit status = 0

command: date
Sun Aug 18 09:25:38 MST 1991
  real :    0.27
  user:    0.00
  sys:     0.00
  child user:    0.05
  child sys:     0.10
normal termination, exit status = 0
```

在这个实例中，在 child user和child sys行中显示的时间是执行 shell和命令的子进程所使用的CPU时间。

程序8-18 时间以及执行命令行参数

```
#include <sys/times.h>
#include "ourhdr.h"

static void pr_times(clock_t, struct tms *, struct tms *);
static void do_cmd(char *);

int
main(int argc, char *argv[])
{
    int i;
    for (i = 1; i < argc; i++)
        do_cmd(argv[i]); /* once for each command-line arg */
    exit(0);
}

static void
do_cmd(char *cmd) /* execute and time the "cmd" */
{
    struct tms tmsstart, tmsend;
    clock_t start, end;
    int status;

    fprintf(stderr, "\ncommand: %s\n", cmd);
    if ( (start = times(&tmsstart)) == -1) /* starting values */
        err_sys("times error");
    if ( (status = system(cmd)) < 0) /* execute command */
        err_sys("system() error");
    if ( (end = times(&tmsend)) == -1) /* ending values */
        err_sys("times error");
    pr_times(end-start, &tmsstart, &tmsend);
    pr_exit(status);
}

static void
pr_times(clock_t real, struct tms *tmsstart, struct tms *tmsend)
```

```

{
    static long    clktck = 0;
    if (clktck == 0) /* fetch clock ticks per second first time */
        if ( (clktck = sysconf(_SC_CLK_TCK)) < 0)
            err_sys("sysconf error");
    fprintf(stderr, "  real: %7.2f\n", real / (double) clktck);
    fprintf(stderr, "  user: %7.2f\n",
        (tmsend->tms_utime - tmsstart->tms_utime) / (double) clktck);
    fprintf(stderr, "  sys: %7.2f\n",
        (tmsend->tms_stime - tmsstart->tms_stime) / (double) clktck);
    fprintf(stderr, "  child user: %7.2f\n",
        (tmsend->tms_cutime - tmsstart->tms_cutime) / (double) clktck);
    fprintf(stderr, "  child sys: %7.2f\n",
        (tmsend->tms_cstime - tmsstart->tms_cstime) / (double) clktck);
}

```

让我们再运行1.10节中的例子：

```

$ a.out "cd /usr/include; grep _POSIX_SOURCE /*.h > /dev/null"
command: cd /usr/include; grep _POSIX_SOURCE /*.h > /dev/null
  real:    18.67
  user:     0.00
  sys:     0.02
  child user:    0.43
  child sys:    4.13
normal termination, exit status = 0

```

如同所期望的那样，所有三个值(实际时间和子进程CPU时间)都与1.10节中的值相近。

8.16 小结

对在UNIX环境中的高级程序设计而言，完整地理解UNIX的进程控制非常重要。其中必须熟练掌握的只有几个——fork、exec族、_exit、wait和waitpid。很多应用程序都使用这些原语。fork原语也给了我们一个了解竞态条件的机会。

本章说明了system函数和进程会计，以及这些进程控制函数的应用情况。本章还说明了exec函数的另一种变体：解释器文件及它们的工作方式。对各种不同的用户 ID和组ID(实际，有效和保存的)的理解和编写安全的设置-用户-ID程序是至关重要的。

在了解进程和子进程的基础上，下一章将进一步说明进程和其他进程的关系——对话期和作业控制。第10章将说明信号机制并以此结束对进程的讨论。

习题

8.1 在程序8-2中用exit取代_exit将关闭标准输出，修改程序验证printf确实返回-1。

8.2 调用vfork后，子进程运行在父进程的地址空间中。如果不是在main函数中调用vfork，而是在vfork以后子进程从这个函数返回，那将会如何？请编写一段程序验证并且画出堆栈中的映像。

8.3 当用\$a.out执行程序8-7一次，其输出是正确的。但是若将该程序按下列方式执行多次，则其输出不正确。

```

$ a.out ; a.out ; a.out
output from parent
ooutput from parent
ouotuptut from child

```

```
put from parent
output from child
utput from child
```

原因是什么？怎样才能更正此种错误？如果使子进程首先输出，还会发生此问题吗？

8.4 在程序8-10中，调用`execl`，指定解释文件为`pathname`。如果调用`execlp`，指定`testinterp`为`filename`，并且目录`/home/stevens/bin`是路径前缀，则运行该程序时，`argv[2]`的打印输出是什么？

8.5 一个进程怎样才能获得其保存的设置-用户-ID？

8.6 编写一段程序，用于创建一个僵死进程，然后调用`system`执行`ps(1)`命令以验证该进程是僵死进程。

8.7 8.9节中提及POSIX.1要求在`exec`时关闭打开目录流。按下列方法对此进行验证：对根目录调用`opendir`，查看`DIR`结构，然后打印`exec`关闭标志。接着打开同一目录读并打印`exec`关闭标志。

第9章 进程关系

9.1 引言

在上一章我们已了解到进程之间具有关系。首先，每个进程有一个父进程。当子进程终止时，父进程会得到通知并能取得子进程的退出状态。在 8.6节说明 waitpid函数时，我们也提到了进程组，以及如何等待进程组中的任意一个进程终止。

本章将更详细地说明进程组以及 POSIX.1引入的对话期新概念。还将介绍登录 shell（登录时所调用的）和所有从登录 shell起动的进程之间的关系。

在说明这些关系时不可能不谈及信号，而谈论信号又需要很多本章介绍的概念。如果你不熟悉UNIX信号，则可能先要浏览一下第10章。

9.2 终端登录

先看一看登录到UNIX系统时所执行的各个程序。在早期的UNIX系统中，例如V7，用户用哑终端(通过RS-232连到主机)进行登录。终端或者是本地的（直接连接）或者是远程的（通过调制解调器连接）。在这两种情况下，登录都经由内核中的终端设备驱动程序。例如，在PDP-11上常用的设备是DH-11和DZ-11。因为连到主机上的终端设备数已经确定，所以同时的登录数也就有了已知的上限。下面说明的登录过程适用于使用一个RS-232终端登录到UNIX系统中。

9.2.1 4.3+BSD终端登录

登录过程在过去15年中并没有多少改变。系统管理者创建一个通常名为/etc/ttys的文件，其中，每个终端设备有一行，每一行说明设备名和传到getty程序的参数，这些参数说明了终端的波特率等。当系统自举时，内核创建进程ID 1，也就是init进程。init进程使系统进入多用户状态。init读文件/etc/ttys,对每一个允许登录的终端设备，init调用一次fork，它所生成的子进程则执行程序getty。这种情况示于图9-1中。

图9-1中各个进程的实际用户ID和有效用户ID都是0(也就是它们都具有超级用户特权)。init以空环境执行getty程序。

getty对终端设备调用open函数，以读、写方式将终端打开。如果设备是调制解调器，则open可能会在设备驱动程序中滞留，直到用户拨号调制解调器，并且线路被接通。一旦设备被打开，则文件描述符0、1、2就被设置到该设备。然后getty输出“login:”之类的信息，并等待用户键入用户名。如果终端支持多种速度，则getty可以测试特殊字符以便适当地更改终端速度（波特率）。关于getty程序以及有关数据文件的细节，请参阅

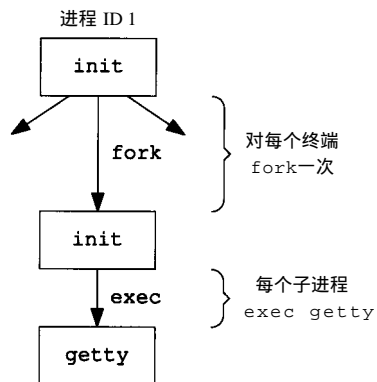


图9-1 init生成进程使终端可用于登录

UNIX手册。

当用户键入了用户名后，getty就完成了。然后它以类似于下列的方式调用login程序：

```
execl("/usr/bin/login", "login", "-p", username, (char *) 0, envp);
```

(在gettytab文件中可能会有一些选择项使其调用其他程序，但系统默认是login程序)。init以一个空环境调用getty。getty以终端名(例如TERM=foo,其中终端foo的类型取自gettytab文件)和在gettytab中的环境字符串为login创建一个环境(envp参数)。-p标志通知login保留传给它的环境,也可将其他环境字符串加到该环境中,但是不要替换它。图9-2显示了login刚被调用后这些进程的状态。

因为最初的init进程具有超级用户优先权,所以图9-2中的所有进程都有超级用户优先权。图9-2中底部三个进程的进程ID相同,因为进程ID不会因执行exec而改变。并且,除了最初的init进程,所有的进程均有一个父进程ID。

login能处理多项工作。因为它得到了用户名,所以能调用getpwnam取得相应用户的口令文件登录项。然后调用getpass(3)以显示提示“Password:”接着读用户键入的口令(自然,禁止回送用户键入的口令)。它调用crypt(3)将用户键入的口令加密,并与该用户口令文件中登录项的pw_passwd字段相比较。如果用户几次键入的口令都无效,则login以参数1调用exit表示登录过程失败。父进程(init)了解到子进程的终止情况后,将再次调用fork,其后又跟随着执行getty,对此终端重复上述过程。

如果用户正确登录,login就将当前工作目录更改为该用户的起始目录(chdir)。它也调用chown改变该终端的所有权,使用该用户成为所有者和组所有者。将对该终端设备的存取许可权改变成:用户读、写和组写。调用setgid及initgroups设置进程的组ID。然后用login所得到的所有信息初始化环境:起始目录(HOME)、shell(SHELL)、用户名(USER和LOGNAME),以及一个系统默认路径(PATH)。最后,login进程改变为登录用户的用户ID(setuid)并调用该用户的登录shell,其方式类似于:

```
execl("/bin/sh", "-sh", (char *) 0);
```

argv[0]的第一个字符-是一个标志,表示该shell被调用为登录shell。shell可以查看此字符,并相应地修改其起动过程。

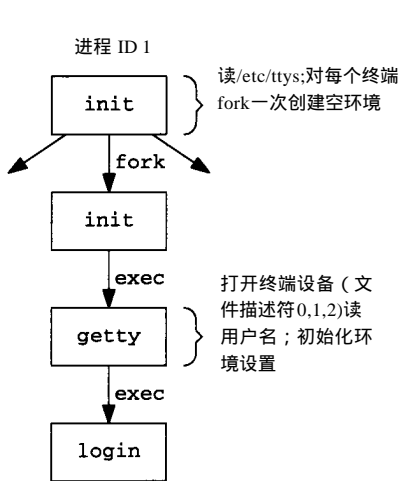


图9-2 login刚被调用后各进程的状态

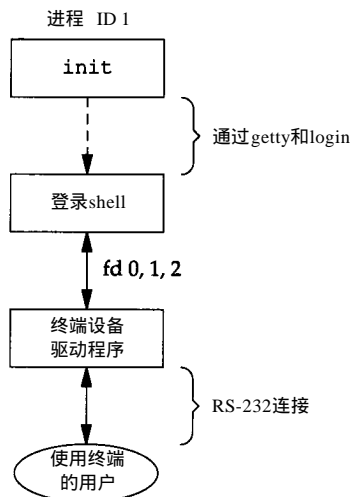


图9-3 终端登录结束后的有关进程

login所做的比上面说的要多。它可选地打印 message-of-the-day 文件，检查新邮件以及其他一些功能。但是考虑到本书的内容，我们主要关心上面所说的功能。

回忆在8.10节中对setuid函数的讨论，因为setuid是由超级用户调用的，它更改所有三个用户ID：实际、有效和保存的用户ID。login在较早时间调用的setgid对所有三个组ID也有同样效果。

到此为止，登录用户的登录shell开始运行。其父进程ID是init进程ID（进程ID 1），所以当此登录shell终止时，init会得到通知（接到SIGCHLD信号），它会对该终端重复全部上述过程。登录shell的文件描述符0, 1和2设置为终端设备。图9-3显示了这种安排。

现在，登录shell读其起动文件（Bourne shell和KornShell是.profile，C shell是.cshrc和.login）。这些起动文件通常改变某些环境变量，加上一些环境变量。例如，很多用户设置他们自己的PATH，常常提示实际终端类型（TERM）。当执行完起动文件后，用户最后得到shell的提示符，并能键入命令。

9.2.2 SVR4终端登录

SVR4支持两种形成的终端登录：(a)getty方式，这与上面对4.3+BSD所说明的一样，(b)ttymon登录，这是SVR4的一种新功能。通常，getty用于控制台，ttymon则用于其他终端的登录。

ttymon是名为服务存取设施（Service Access Facility, SAF）的一部分。按照本书的目的，我们只简单说明从init到登录shell之间工作过程，最后结果与图9-3中所示相似。init是sac（服务存取控制器）的父进程，sac调用fork，然后其子进程执行ttymon程序，此时系统进入多用户状态。ttymon监视列于配置文件中的所有终端端口，当用户键入登录名时，它调用一次fork。在此之后该子进程又执行登录用户的登录shell，于是到达了图9-3中所示的位置。一个区别是登录shell的父进程现在是ttymon，而在getty登录中，登录shell的父进程是init。

9.3 网络登录

9.3.1 4.3 + BSD网络登录

在上节所述的终端登录中，init知道哪些终端设备可用来进行登录，并为每个设备生成一个getty进程。但是，对网络登录则情况有所不同，所有登录都经由内核的网络界面驱动程序（例如：以太网驱动程序），事先并不知道将会有多少这样的登录。不是使一个进程等待每一个可能的登录，而是必须等待一个网络连接请求的到达。在4.3+BSD中，有一个称为inetd的进程（有时称为Internet superserver），它等待大多数网络连接。本书将说明4.3+BSD的网络登录中所涉及的进程序列。关于这些进程的网络程序设计方面的细节请参阅Stevens [1990]。

作为系统起动的一部分，init调用一个shell，使其执行shell脚本etc/rc。由此shell脚本起动一个精灵进程inetd。一旦此shell脚本终止，inetd的父进程就变成init。inetd等待TCP/IP连接请求到达主机，而当一个连接请求到达时，它执行一次fork，然后该子进程执行适当的程序。

我们假定到达了一个对于TELNET服务器的TCP连接请求。TELNET是使用TCP协议的远程登录应用程序。在另一个主机（它通过某种形式的网络，连接到服务器主机上）上的用户，或在同一个主机上的一个用户籍起动TELNET客户进程(client)起动登录过程：

```
telnet hostname
```

该客户进程打开一个到名为hostname的主机的TCP连接，在hostname主机上起动的程序被称为

TELNET服务器。然后，客户进程和服务器进程之间使用 TELNET应用协议通过 TCP连接交换数据。所发生的是起动客户进程的用户现在登录到了服务器进程所在的主机。（自然，用户需要在服务器进程主机上有一个有效的账号）。图9-4显示了在执行 TELNET服务器进程（称为 telnetd）中所涉及的进程序列。

然后，telnetd进程打开一个伪终端设备，并用 fork生成一个子进程（第 19章将详细说明伪终端）。父进程处理通过网络连接的通信，子进程则执行 login程序。父、子进程通过伪终端相连接。在调用exec之前，子进程使其文件描述符0,1,2与伪终端相连。如果登录正确，login就执行9.2节中所述的同样步骤——更改当前工作目录为起始目录，设置登录用户的组 ID和用户ID，以及登录用户的初始环境。然后 login用exec将其自身替换为登录用户的登录 shell。图9-5显示了到达这一点时的进程安排。

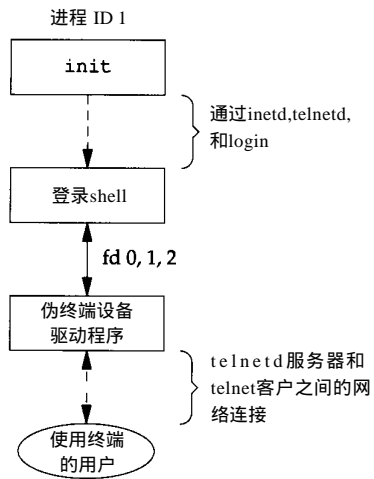
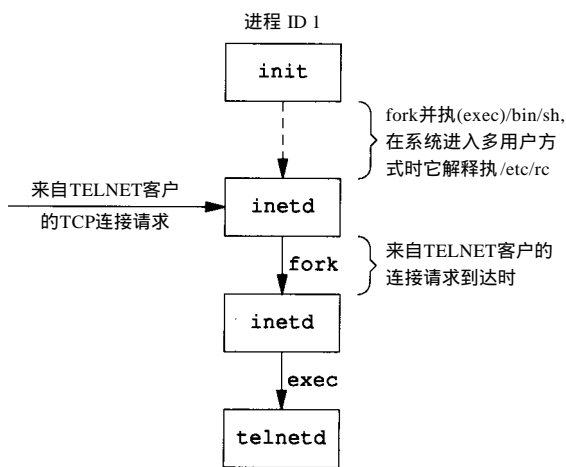


图9-4 执行TELNET服务进程中涉及的进程序列 图9-5 为网络登录设置了fd 0, 1, 2后的进程安排

很明显，在伪终端设备驱动程序和终端实际用户之间有很多事情在进行着。第 19章详细介绍伪终端时，我们会介绍与这种安排相关的所有进程。

需要理解的重点是：当通过终端（见图 9-3）或网络（见图 9-5）登录时，我们得到一个登录shell，其标准输入、输出和标准出错连接到一个终端设备或者伪终端设备上。在下一节中我们会了解到这一登录shell是一个POSIX.1对话期的开始，而此终端或伪终端则是会话期的控制终端。

9.3.2 SVR4网络登录

SVR4中网络登录的情况与4.3+BSD中的几乎一样。同样使用了inetd服务器进程，但是在SVR4中inetd是作为一种服务由服务存取控制器sac调用的，其父进程不是init。最后得到的结果与图9-5中一样。

9.4 进程组

每个进程除了有一进程ID之外，还属于一个进程组，第 10章讨论信号时还会涉及进程组。

进程组是一个或多个进程的集合。每个进程组有一个唯一的进程组 ID。进程组ID类似于进程ID——它是一个正整数，并可存放在pid_t数据类型中。函数getpgrp返回调用进程的进程

组ID。

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

pid_t getpgrp(void);
```

返回：调用进程的进程组 ID

在很多伯克利的系统中，包括 4.3+BSD，这一函数的参数是 *pid*，返回该进程的进程组。上面所示的原型是 POSIX.1 版本。

每个进程组有一个组长进程。组长进程的标识是，其进程组 ID 等于其进程 ID。

进程组组长可以创建一个进程组，创建该组中的进程，然后终止。只要在某个进程组中有一个进程存在，则该进程组就存在，这与其组长进程是否终止无关。从进程组创建开始到其中最后一个进程离开为止的时间区间称为进程组的生命期。某个进程组中的最后一个进程可以终止，也可以参加另一个进程组。

进程调用 `setpgid` 可以参加一个现存的组或者创建一个新进程组（下一节中将说明用 `setsid` 也可以创建一个新的进程组）。

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

int setpgid(pid_t pid, pid_t pgid);
```

返回：若成功则为 0，出错为 -1

这将 *pid* 进程的进程组 ID 设置为 *pgid*。如果这两个参数相等，则由 *pid* 指定的进程变成进程组组长。

一个进程只能为它自己或它的子进程设置进程组 ID。在它的子进程调用了 `exec` 后，它就不再能改变该子进程的进程组 ID。

如果 *pid* 是 0，则使用调用者的进程 ID。另外，如果 *pgid* 是 0，则由 *pid* 指定的进程 ID 被用作进程组 ID。

如果系统不支持作业控制（9.8 节将说明作业控制），那么就不定义 `_POSIX_JOB_CONTROL`，在这种情况下，此函数返回出错，`errno` 设置为 `ENOSYS`。

在大多数作业控制 shell 中，在 `fork` 之后调用此函数，使父进程设置其子进程的进程组 ID，然后使子进程设置其自己的进程组 ID。这些调用中有一个是冗余的，但这样做可以保证父、子进程在进一步操作之前，子进程都进入了该进程组。如果不这样做的话，那么就产生一个竞态条件，因为它依赖于哪一个进程先执行。

在讨论信号时，将说明如何将一个信号送给一个进程（由其进程 ID 标识）或送给一个进程组（由进程组 ID 标识）。同样，`waitpid` 则可被用来等待一个进程或者指定进程组中的一个进程。

9.5 对话期

对话期（`session`）是一个或多个进程组的集合。例如，可以有图 9-6 中所示的安排。其中，

在一个对话期中有三个进程组。通常是由 shell 的管道线将几个进程编成一组的。例如，图 9-6 中的安排可能是由下列形式的 shell 命令形成的：

```
proc1 | proc2 &
proc3 | proc4 | proc5
```

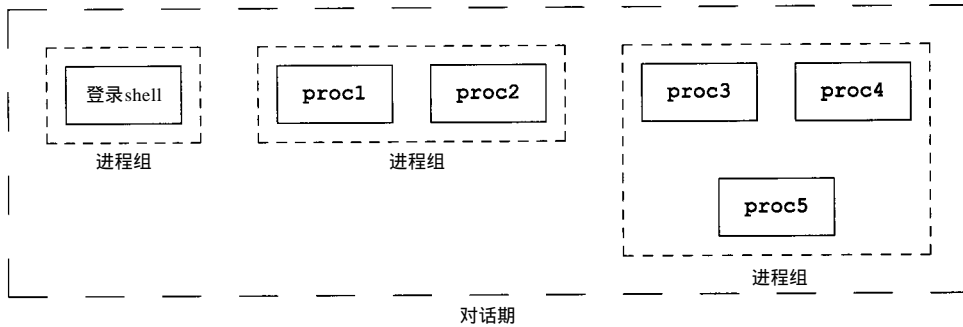


图9-6 进程组和对话期中的进程安排

进程调用 `setsid` 函数就可建立一个新对话期。

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

pid_t setsid(void);
```

返回：若成功则为进程组 ID，若出错则为 -1

如果调用此函数的进程不是一个进程组的组长，则此函数创建一个新对话期，结果为：

- (1) 此进程变成该新对话期的对话期首进程（session leader，对话期首进程是创建该对话期的进程）。此进程是该新对话期中的唯一进程。
- (2) 此进程成为一个新进程组的组长进程。新进程组 ID 是此调用进程的进程 ID。
- (3) 此进程没有控制终端（下一节讨论控制终端）。如果在调用 `setsid` 之前此进程有一个控制终端，那么这种联系也被解除。

如果此调用进程已经是一个进程组的组长，则此函数返回出错。为了保证不处于这种情况，通常先调用 `fork`，然后使其父进程终止，而子进程则继续。因为子进程继承了父进程的进程组 ID，而其进程 ID 则是新分配的，两者不可能相等，所以这就保证了子进程不是一个进程组的组长。

POSIX.1 只包括对话期首进程，而没有类似与进程 ID 和进程组 ID 的对话期 ID。显然，对话期首进程是具有唯一进程 ID 的单个进程，所以可以将对话期首进程的进程 ID 视为对话期 ID。SVR4 就是这样处理的。SVID 和 SVR4 的 `setsid(2)` 手册页谈到了以此种方式定义的对话期 ID。这是一种实现细节，它不是 POSIX.1 中定义的，4.3+BSD 也不支持它。

SVR4 有一个 `getsid` 函数，它返回一个进程的对话期 ID。此函数不是 POSIX.1 的所属部分，4.3+BSD 也不支持此函数。

9.6 控制终端

对话期和进程组有一些其他特性：

- 一个对话期可以有一个单独的控制终端（controlling terminal）。这通常是我们在其上登录的终端设备（终端登录情况）或伪终端设备（网络登录情况）。
- 建立与控制终端连接的对话期首进程，被称之为控制进程（controlling process）。
- 一个对话期中的几个进程组可被分成一个前台进程组（foreground process group）以及一个或几个后台进程组（background process group）。
- 如果一个对话期有一个控制终端，则它有一个前台进程组，其他进程组则为后台进程组。
- 无论何时键入中断键（常常是DELETE或Ctrl-C）或退出键（常常是Ctrl-\），就会造成将中断信号或退出信号送至前台进程组的所有进程。
- 如果终端界面检测到调制解调器已经脱开连接，则将挂断信号送至控制进程（对话期首进程。）这些特性示于图9-7中。

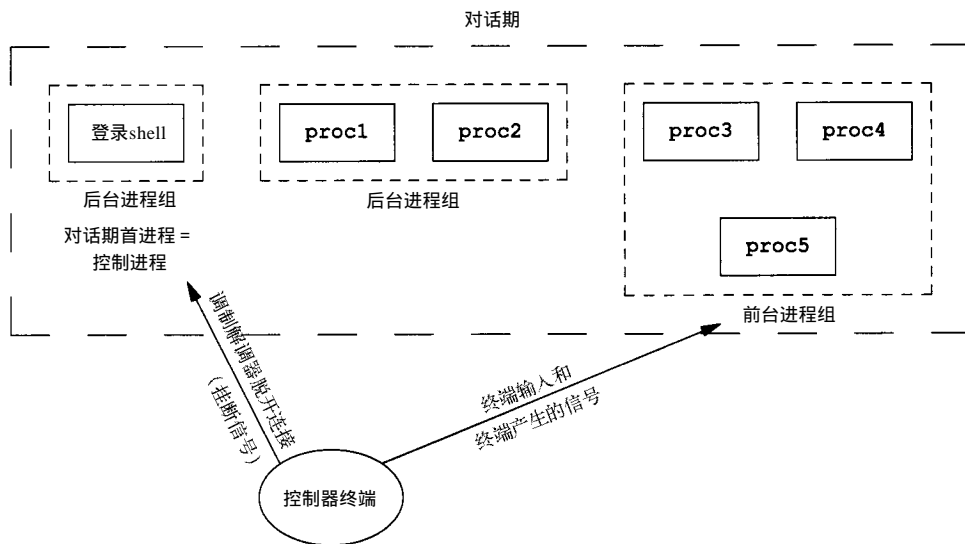


图9-7 进程组、对话期和控制终端

通常，我们不必担心控制终端——登录时，将自动建立控制终端。

系统如何分配一个控制终端依赖于实现。19.4节中将说明实际步骤。

当对话期首进程打开第一个尚未与一个对话期相关联的终端设备时，SVR4将此作为控制终端分配给此对话期。这假定对话期首进程在调用 `open` 时没有指定 `O_NOCTTY` 标志(见3.3节)。

当对话期首进程用 `TIOCSCTTY`（第三个参数是空指针）的 `request` 参数调用 `ioctl` 时，4.3+BSD为对话期分配控制终端。为使此调用成功执行，此对话期不能已经有一个控制终端（通常 `ioctl` 调用紧跟在 `setsid` 调用之后，`setsid` 保证此进程是一个没有控制终端的对话期首进程）。4.3+BSD 不使用POSIX.1中对 `open` 函数所说明的 `O_NOCTTY` 标志。

有时不管标准输入、标准输出是否重新定向，程序都要与控制终端交互作用。保证程序读写控制终端的方法是打开文件 `/dev/tty`，在内核中，此特殊文件是控制终端的同义语。自然，如果程序没有控制终端，则打开此设备将失败。

典型的例子是用于读口令的 `getpass(3)` 函数（终端回送被关闭）。这一函数由 `crypt(1)` 程序调用，而此程序则可用于管道中。例如：

```
crypt < salaries | lpr
```

它将文件 `salaries` 解密，然后经由管道将输出送至打印假脱机程序。因为 `crypt` 从其标准输入读输入文件，所以标准输入不能用于输入口令。但是，`crypt` 的一个设计特征是每次运行此程序时，都应输入加密口令，这样也就不需要将口令存放在文件中。

已经知道有一些方法可以破译 `crypt` 程序使用的密码。关于加密文件的详细情况请参见 Garfinkel 和 Spafford [1991]。

9.7 tcgetpgrp和tcsetpgrp函数

需要有一种方法来通知内核哪一个进程组是前台进程组，这样，终端设备驱动程序就能了解将终端输入和终端产生的信号送到何处（见图 9-7）。

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

pid_t tcgetpgrp(int fildes);
```

返回：若成功则为前台进程组 ID，若出错则为 -1

```
int tcsetpgrp(int fildes, pid_t pgrp);
```

返回：若成功则为 0，若出错则为 -1

函数 `tcgetpgrp` 返回前台进程组 ID，它与在 `fildes` 上打开的终端相关。

如果进程有一个控制终端，则该进程可以调用 `tcsetpgrp` 将前台进程组 ID 设置为 `pgrp`。 `pgrp` 值应当是在同一对话期中的一个进程组的 ID。 `fildes` 必须引用该对话期的控制终端。

大多数应用程序并不直接调用这两个函数。它们通常由作业控制 shell 调用。只有定义了 `_POSIX_JOB_CONTROL`，这两个函数才被定义了。否则它们返回出错。

9.8 作业控制

作业控制是伯克利在 1980 年左右加到 UNIX 的一个新特性。它允许在一个终端上启动多个作业（进程组），控制哪一个作业可以存取该终端，以及哪些作业在后台运行。作业控制要求三种形式的支持：

- (1) 支持作业控制的 shell。
- (2) 内核中的终端驱动程序必须支持作业控制。
- (3) 必须提供对某些作业控制信号的支持。

SVR3 提供了一种不同形式的作业控制，称为 shell 层。但是 POSIX.1 选择了伯克利形式的作业控制，这也是我们在这里所说明的。回忆表 2-7，如果系统支持作业控制，则定义常数 `_POSIX_JOB_CONTROL`。

FIPS 151-1 要求 POSIX.1 作业控制。

SVR4 和 4.3+BSD 支持 POSIX.1 作业控制。

从shell使用作业控制功能角度观察，可以在前台或后台起动一个作业。一个作业只是几个进程的集合，通常是一个进程管道。例如：

```
vi main.c
```

在前台起动了只有一个进程的一个作业。下面的命令：

```
pr *.c | lpr &  
make all &
```

在后台起动了两个作业。这两个后台作业所调用的进程都在后台运行。

正如前述，我们需要一个支持作业控制的 shell 以使用由作业控制提供的功能。对于早期的系统，shell 是否支持作业控制比较易于说明。C shell 支持作业控制，Bourne shell 则不支持，而 KornShell 能否支持作业控制取决于主机是否支持作业控制。但是现在 C shell 已被移植到并不支持作业控制的系统上（例如系统 V 的早期版本），而 SVR4 Bourne shell 当用名字 jsh 而不是 sh 调用时则支持作业控制。如果主机支持作业控制，则 KornShell 继续支持作业控制。各种 shell 之间的差别并不显著时，我们将只是一般地说明支持作业控制的 shell 和不支持作控制的 shell。

当起动一个后台作业时，shell 赋与它一个作业标识，并打印一个或几个进程 ID。下面的操作过程显示了 KornShell 是如何处理这一点的。

```
$ make all > Make.out &  
[1]      1475  
$ pr *.c | lpr &  
[2]      1490  
$                               键入回车  
[2] + Done      pr *.c | lpr &  
[1] + Done      make all > Make.out &
```

make 是作业号 1，所起动的进程 ID 是 1475。下一个管道线是作业号 2，其第一个进程的进程 ID 是 1490。当作业已完成而且键入回车时，shell 通知我们作业已经完成。键入回车是为了让 shell 打印其提示符。shell 并不在任何随意的时间打印后台作业的状态改变——它只在打印其提示符之前这样做。如果不这样处理，则当我们正输入一行时，它也可能输出。

我们可以键入一个影响前台作业的特殊字符——挂起键（一般采用 Ctrl-Z）与终端进行交互作用。键入此字符使终端驱动程序将信号 SIGTSTP 送至前台进程组中的所有进程，后台进程组作业则不受影响。实际上有三个特殊字符可使终端驱动程序产生信号，并将它们送至前台进程组，它们是：

- 中断字符（一般采用 DELETE 或 Ctrl-C）产生 SIGINT。
- 退出字符（一般采用 Ctrl-\）产生 SIGQUIT。
- 挂起字符（一般采用 Ctrl-Z）产生 SIGTSTP。

第 11 章中将说明可将这三个字符更改为任一其他字符，以及如何使终端驱动程序不处理这些特殊字符。

终端驱动程序必须处理与作业控制有关的另一种情况。我们可以有一个前台作业，若干个后台作业，这些作业中哪一个接收我们在终端上键入的字符呢？只有前台作业接收终端输入。如果后台作业试图读终端，那么这并不是一个错误，但是终端驱动程序检测这种情况，并且发送一个特定信号 SIGTTIN 给后台作业。这通常会停止此后台作业，而有关用户则会得到这种情况的通知，然后就可将此作业转为前台作业运行，于是它就可读终端。下列操作过程显示了这一点：

```

$ cat > temp.foo &          在后台启动，但将从标准输入读
[1]      1681
$                               键入回车
[1] + Stopped (tty input)      cat > temp.foo &
$ fg %1                       使1号作业成为前台作业
cat > temp.foo                shell告诉我们现在哪一个作业在前台
hello, world                  输入1行
^D                              键入文件结束符
$ cat temp.foo                检查该行已送入文件
hello, world

```

shell在后台启动cat进程，但是当cat试图读其标准输入（控制终端）时，终端驱动程序知道它是个后台作业，于是将SIGTIN信号送至该后台作业。shell检测到其子进程的状态改变（回忆8.6节中对wait和waitpid的讨论），并通知我们该作业已被停止。然后，用shell的fg命令将此停止的作业送入前台运行（关于作业控制命令，例如fg和bg的详细情况，以及标识不同作业的各种方法请参阅有关shell的手册页）。这样做使shell将此作业转为前台进程组（tcsetpgrp），并将继续信号(SIGCONT)送给该进程组。因为该作业现在前台进程组中，所以它可以读控制终端。

如果后台作业输出到控制终端又将发生什么呢？这是一个我们可以允许或禁止的选择项。通常，可以用stty(1)命令改变这一选择项（第11章将说明在程序中如何改变这一选择项）。下面显示了这种操作过程：

```

$ cat temp.foo &          在后台执行
[1]      1719
$ hello, world           在提示符后出现后台作业的输出
                           键入回车
[1] + Done               cat temp.foo &
$ stty tostop           禁止后台作业向控制终端输出
$ cat temp.foo &       在后台再次执行
[1]      1721
$                               键入回车，发现作业已停止
[1] + Stopped(tty output)  cat temp.foo &
$ fg %1                 将停止的作业恢复为前台作业
cat temp.foo            shell告诉我们现在哪一个作业在前台
hello, world            该作业的输出

```

图9-8摘录了我们已说明的作业控制的某些功能。穿过终端驱动程序框的实线表示：终端I/O和终端产生的信号总是从前台进程组连接到实际终端。对应于SIGTTOU信号的虚线表示，后台进程组进程的输出是否出现在终端是可选择的。

是否需要作业控制是一个有很多争论的问题。作业控制是在窗口终端广泛得到应用之前设计和实现的。很多人认为设计得好的窗口系统已经免除了对作业控制的需要。某些人抱怨作业控制的实现要求得到内核、终端驱动程序、shell以及某些应用程序的支持，是吃力不讨好的事情。某些人在窗口系统中使用作业控制，他们认为两者都需要。不管你的意见如何，作业控制是POSIX.1以及FIPS 151-1的组成部分，它还将继续存在。

9.9 shell执行程序

让我们检验一下shell是如何执行程序的，以及这与进程组、控制终端和对话期等概念的关系。为此，要再次使用ps命令。

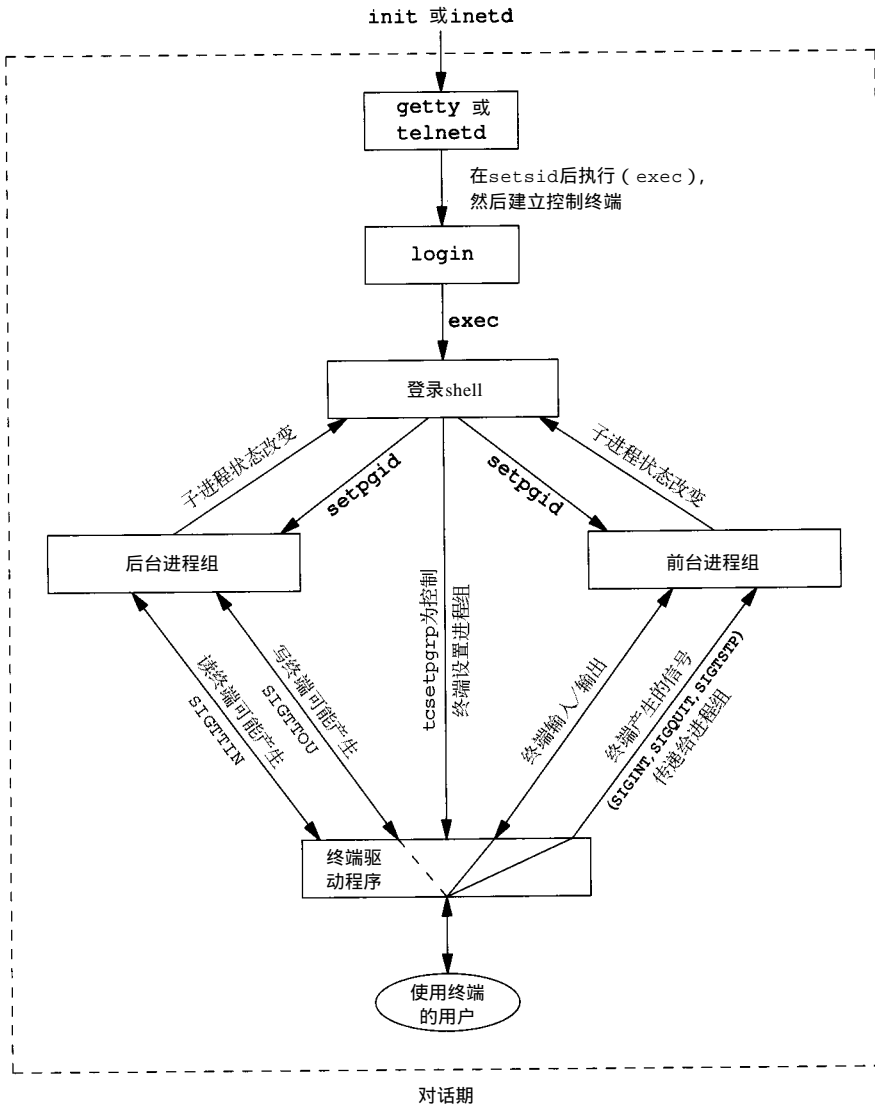


图9-8 对于前台、后台作业以及终端驱动程序作业控制功能摘要

首先使用不支持作业控制的经典的 Bourne shell。如果执行：

```
ps -xj
```

则其输出为：

```
PPID  PID  PGID  SID  TPGID  COMMAND
  1   163   163   163   163    -sh
 163   168   163   163   163    ps
```

其中，删除了一些我们现在不感兴趣的列——终端名、用户ID、CPU时间等。shell和ps命令两者位于同一对话期和前台进程组(163)中。因为163是在TPGID列中显示的进程组，所以称其为前台进程组。ps的父进程是shell，这正是我们所期望的。注意，登录shell是由login以—作为其第一个字符调用的。

不幸的是，ps(1)命令的输出在各个UNIX版本中都有所不同。在SVR4之下，使用命令ps -jl得到类似的输出，但SVR4不打印TPGID字段。在4.3+BSD之下，使用命令ps -xj -otpgid。

注意，说进程与终端进程组ID(TPGID列)相关联是用词不当。进程并没有终端进程控制组。进程属于一个进程组，而进程组属于一个对话期。对话期可能有，也可能没有控制终端。如果它确有一个控制终端，则此终端设备知道其前台进程的进程组ID。这一值可以用tcsetpgrp函数在终端驱动程序中设置（见图9-8）。前台进程组ID是终端的一个属性，而不是进程的属性。取自终端设备驱动程序的该值是ps在TPGID列中打印的值。如果ps发现此对话期没有控制终端，则它在该列打印-1。

如果在后台执行该命令：

```
ps -xj &
```

则唯一改变的值得是命令的进程ID。

PPID	PID	PGID	SID	TPGID	COMMAND
1	163	163	163	163	-sh
163	169	163	163	163	ps

因为这种shell不知道作业控制，所以后台作业没有构成另一个进程组，也没有从后台作业处取走控制终端。

现在看一看Bourne shell如何处理管道线。执行下列命令：

```
ps -xj | cat1
```

其输出是：

PPID	PID	PGID	SID	TPGID	COMMAND
1	163	163	163	163	-sh
163	200	163	163	163	cat1
200	201	163	163	163	ps

(程序cat1只是标准cat程序的一个副本，但名字不同。本节还将使用cat的另一个名为cat2的副本。在一个管道线中使用两个cat时，不同的名字可使我们将它们区分开来。)注意，管道中的最后一个进程是shell的子进程，该管道中的第一个进程则是最后一个进程的子进程。从中可以看出，shell fork一个它的副本，然后此副本再为管道线中的每条命令各fork一个进程。

如果在后台执行此管道线：

```
ps -xj | cat1 &
```

则只有进程ID改变了。因为shell并不处理作业控制，后台进程的进程组ID仍是163，如同终端进程组ID一样。

如果一个后台进程试图读其控制终端，则会发生什么呢？例如，若执行：

```
cat > temp.foo &
```

在有作业控制时，后台作业被放在后台进程组，如果后台作业试图读控制终端，则会产生信号SIGTTIN。在没有作业控制时，其处理方法是：如果该进程自己不重新定向标准输入，则shell自动将后台进程的标准输入重新定向到/dev/null。读/dev/null则产生一个文件结束。这就意味着后台cat进程立即读到文件尾，并正常结束。

上面说明了对后台进程通过其标准输入存取控制终端的适当的处理方法，但是，如果一个后台进程打开/dev/tty并且读该控制终端，又将怎样呢？对此问题的回答是“看情况”。但是这很可能不是我们所需要的。例如：

```
crypt < salaries | lpr &
```

就是这样的一条管道线。我们在后台运行它，但是 crypt程序打开/dev/tty，更改终端的特性（禁止回送），然后从该设备读，最后重置该终端特性。当执行这条后台管道时，crypt在终端上打印提示符“Password:”，但是shell读取了我们所输入的加密码口令，并企图执行其中一条命令。我们输送给shell的下一行，则被crypt进程取为口令行，于是salaries也就不能正确地译码，结果将一堆没有用的信息送到了打印机。在这里，我们有了两个进程，它们试图同时读同一设备，其结果则依赖于系统。前面说明的作业控制以较好的方式处理一个终端在多个进程间的转接。

返回到Bourne shell实例，在一条管道中执行三个进程：

```
ps -xj | cat1 | cat2
```

下面看一看shell所用的进程控制：

PPID	PID	PGID	SID	TPGID	COMMAND
1	163	163	163	163	-sh
163	202	163	163	163	cat2
202	203	163	163	163	ps
202	204	163	163	163	cat1

再重申一遍，该管道中的最后一个进程是shell的子进程，而执行管道中其他命令的进程则是该最后进程的子进程。图9-9显示了所发生的情况。

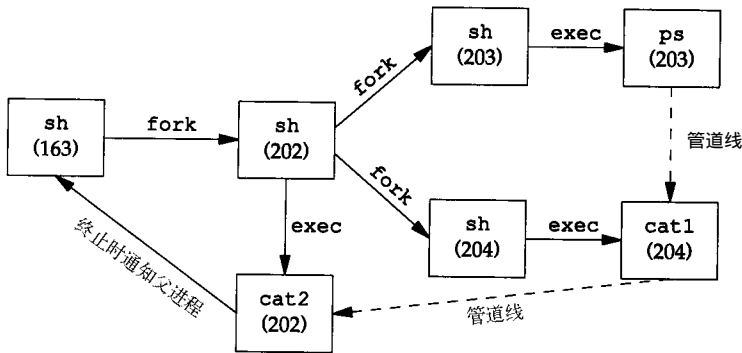


图9-9 Bourne shell执行管道线ps -xj | cat1 | cat2 时的进程

因为该道通线中的最后一个进程是登录 shell的子进程，当该进程(cat2)终止时，shell得到通知。

现在让我们用作业控制shell来检验一下同一个例子。这将显示这些shell处理后台作业的方法。在本例中将使用KornShell——用C shell得到的结果几乎是一样的。

```
ps -xj
```

其输出为：

PPID	PID	PGID	SID	TPGID	COMMAND
------	-----	------	-----	-------	---------


```
1 700 700 700 708 -ksh
700 708 708 700 708 ps
```

(从本例开始，以粗体显示前台进程组。)我们立即看到了与 Bourne shell 例子的区别。KornShell 将前台作业(ps)放入了它自己的进程组(708)。ps 命令是进程组组长进程，并是该进程组的唯一进程。进一步而言，此进程组具有控制终端，所以它是前台进程。我们的登录 shell 在执行 ps 命令时是后台进程组。但需要注意的是，这两个进程组 700 和 708 都是同一对话期的成员。事实上，在本书的实例中对话期决不会改变。

在后台执行此进程：

```
ps -xj &
```

其输出为：

```
PPID  PID  PGID  SID  TPGID  COMMAND
1 700 700 700 700 -ksh
700 709 709 700 700 ps
```

再一次，ps 命令被放入它自己的进程组，但是此时进程组(709)不再是前台进程组。这是一个后台进程组。TPGID 700 指示前台进程组是登录 shell。

按下列方式在一个管道中执行两个进程：

```
ps -xj | cat1
```

其输出为：

```
PPID  PID  PGID  SID  TPGID  COMMAND
1 700 700 700 710 -ksh
700 710 710 700 710 ps
700 711 710 700 710 cat1
```

两个进程 ps 和 cat1 都在一个新进程组(710)中，这是一个前台进程组。在本例和类似的 Bourne shell 实例之间能看到另一个区别。Bourne shell 首先创建将执行管道线中最后一条命令的进程，而此进程是第一个进程的父进程。在这里，KornShell 是两个进程的父进程。但是，如果在后台执行此管道线：

```
ps -xj | cat1 &
```

其结果显示现在 KornShell 以与 Bourne shell 相同的方式产生进程。

```
PPID  PID  PGID  SID  TPGID  COMMAND
1 700 700 700 700 -ksh
700 712 712 700 700 cat1
712 713 712 700 700 ps
```

两个进程 712 和 713 都处在后台进程组 712 中。

9.10 孤儿进程组

一个父进程已终止的进程称为孤儿进程 (orphan process)，这种进程由 init 进程收养。现在我们要说明整个进程组也可成为孤儿，以及 POSIX.1 如何处理它。

实例

考虑一个进程，它 fork 了一个子进程然后终止。这在系统中是经常发生的，并无异常之

处，但是在父进程终止时，如果该子进程停止（用作业控制）又将如何呢？子进程如何继续，以及子进程是否知道它已经是孤儿进程？程序 9-1 是这种情况的一个例子。下面要说明该程序的某些新特征。图 9-10 显示了程序 9-1 已经起动，父进程已经 fork 了子进程后的情况。

这里，假定使用了一个作业控制 shell。回忆前面所述，shell 将前台进程放在一个进程组中（本例中是 512），shell 则留在自己的组内（442）。子进程继承其父进程（512）的进程组。在 fork 之后：

- 父进程睡眠 5 秒钟，这是一种让子进程在父进程终止之前运行的一种权宜之计。
- 子进程为挂断信号（SIGHUP）建立信号处理程序。这样就能观察到 SIGHUP 信号是否已送到子进程。（第 10 章将讨论信号处理程序。）

• 子进程用 kill 函数向其自身发送停止信号（SIGTSTP）。这停止了子进程，类似于用终端挂起字符（Ctrl-Z）停止一个前台作业。

- 当父进程终止时，该子进程成为孤儿进程，共父进程 ID 成为 1，也就是 init 进程 ID。
- 现在，子进程成为一个孤儿进程组的成员。POSIX.1 将孤儿进程组（orphaned process group）定义为：该组中每个成员的父进程或者是该组的一个成员，或者不是该组所属对话期的成员。对孤儿进程组的另一种描述可以是：一个进程组不是孤儿进程组的条件是：该组中有一个进程，其父进程在属于同一对话期的另一个组中。如果进程组不是孤儿进程组，那么在属于同一对话期的另一个组中的父进程就有机会重新起动该组中停止的进程。

在这里，进程组中所有进程的进程（如进程 513 的父进程 1）属于另一个对话期。所以此进程组是孤儿进程组。

• 因为在父进程终止后，进程组成为孤儿进程组，POSIX.1 要求向新孤儿进程组中处于停止状态的每一个进程发送挂断信号（SIGHUP），接着又向其发送继续信号（SIGCONT）。

• 在处理了挂断信号后，子进程继续。对挂断信号的系统默认动作是终止该进程，为此必须提供一个信号处理程序以捕捉该信号。因此，我们期望 sig_hup 函数中的 printf 会在 pr_ids 函数中的 printf 之前执行。

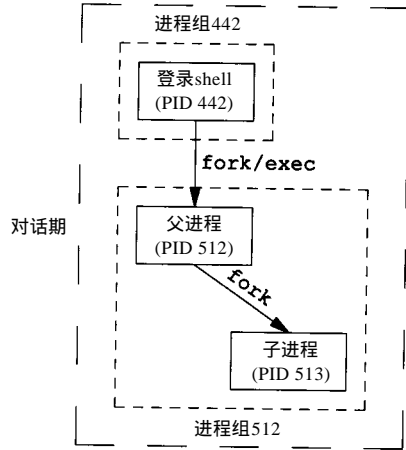


图9-10 将成为孤儿的进程组的实例

程序9-1 创建一个孤儿进程组

```
#include <sys/types.h>
#include <errno.h>
#include <fcntl.h>
#include <signal.h>
#include "ourhdr.h"

static void sig_hup(int);
static void pr_ids(char *);

int
main(void)
{
    char    c;
    pid_t   pid;

    pr_ids("parent");
```

```

if ( (pid = fork()) < 0)
    err_sys("fork error");

else if (pid > 0) { /* parent */
    sleep(5);      /* sleep to let child stop itself */
    exit(0);      /* then parent exits */
} else {          /* child */
    pr_ids("child");
    signal(SIGHUP, sig_hup); /* establish signal handler */
    kill(getpid(), SIGTSTP); /* stop ourself */
    pr_ids("child"); /* this prints only if we're continued */
    if (read(0, &c, 1) != 1)
        printf("read error from control terminal, errno = %d\n", errno);
    exit(0);
}
}

static void
sig_hup(int signo)
{
    printf("SIGHUP received, pid = %d\n", getpid());
    return;
}

static void
pr_ids(char *name)
{
    printf("%s: pid = %d, ppid = %d, pgrp = %d\n",
           name, getpid(), getppid(), getpgrp());
    fflush(stdout);
}

```

下面是程序9-1的输出：

```

$ a.out
parent: pid = 512, ppid = 442, pgrp = 512
child: pid = 513, ppid = 512, pgrp = 512
$ SIGHUP received, pid = 513
child: pid = 513, ppid = 1, pgrp = 512
read error from control terminal, errno = 5

```

注意，因为两个进程，登录 shell 和子进程都写向终端，所以 shell 提示符和子进程的输出一起出现。正如我们所期望的那样，子进程的父进程 ID 变成 1。

注意，在子进程中调用 `pr_ids` 后，程序企图读标准输入。正如前述，当后台进程组试图读控制终端时，则对该后台进程组产生 `SIGTTIN`。但在这里，这是一个孤儿进程组，如果内核用此信号停止它，则此进程组中的进程就再也不会继续。POSIX.1 规定，`read` 返回出错，其 `errno` 设置为 `EIO`（在作者所用的系统中其值是 5）。

最后，要注意的是父进程终止时，子进程变成后台进程组，因为父进程是由 shell 作为前台作业执行的。

在 19.5 节的 `pty` 程序中将会看到孤儿进程组的另一个例子。

9.11 4.3+BSD 实现

上面说明了进程、进程组、对话期和控制终端的各种属性，值得观察一下所有这些是如何实现的。下面简要说明 4.3+BSD 的实现。SVR4 实现的某些详细情况则参见 Williams [1989]。

图9-11显示了4.3+BSD的各种数据结构。

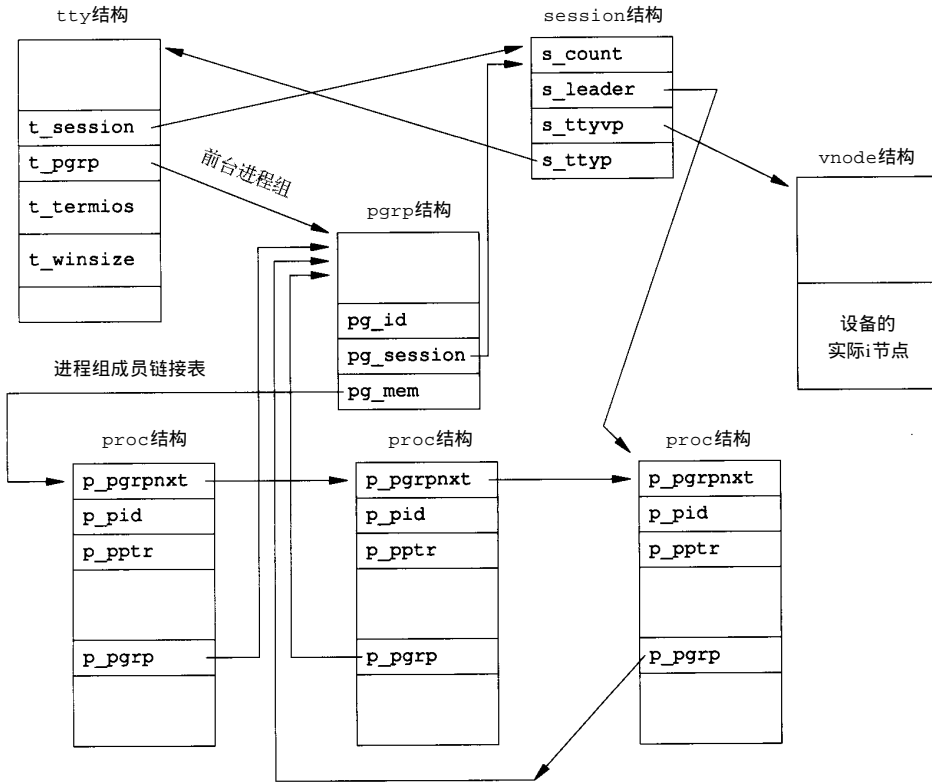


图9-11 对话期和进程组的4.3+BSD实现

下面说明图中的各个字段。从session结构开始。每个对话期都分配了这样一种结构（例如，每次调用setsid时）。

- `s_count`是该对话期中的进程组数。当此计数器减至0时，则可释放此结构。
- `s_leader`是指向对话期首进程proc结构的指针。如上所述，4.3+BSD不保持对话期ID字段，而SVR4则保持此字段。

- `s_ttyvp`是指向控制终端vnode结构的指针。
- `s_ttyp`是指向控制终端tty结构的指针。

在调用setsid时，在内核中分配一个新的对话期结构。`s_count`设置为1，`s_leader`设置为调用进程的proc结构的指针，因为新对话期没有控制终端，所以`s_ttyvp`和`s_ttyp`设置为空指针。

接着说明tty结构。每个终端设备和每个伪终端设备均在内核中分配这样一种结构（第19章将对伪终端作更多说明。）

- `t_session`指向将此终端作为控制终端的session结构（注意，tty结构指向session结构，结构也指向tty结构）。终端在失去载波信号（见图9-7）时使用此指针将挂起信号送给对话期首进程。

- `t_pgrp`指向前台进程组的pgrp结构。终端驱动程序用此字段将信号送向前台进程组。由输入特殊字符(中断、退出和挂起)而产生的三个信号被送至前台进程组。

- `t_termios`是包含所有这些特殊字符和与该终端有关信息（例如，波特率、回送打开或关闭等）的结构。第11章将再说明此结构。

• `t_winsize`是包含终端窗口当前尺寸的 `winsize`结构。当终端窗口尺寸改变时，信号 `SIGWINCH`被送至前台进程组。11.12节将说明如何设置和存取终端当前窗口尺寸。

注意，为了找到特定对话期的前台进程组，内核从 `session`结构开始，然后用 `s_ttyp`得到控制终端的 `tty`结构，然后用 `t_pgrp`得到前台进程组的 `pgrp`结构。

`pgrp`结构包含一个进程组的信息。

• `pg_id`是进程组ID。

• `pg_session`指向此进程组所属的 `session`结构。

• `pg_mem`是指向此进程组第一个进程 `proc`结构的指针。 `proc`结构中的 `p_pgrpnxt`指向此组中的下一个进程，进程组中最后一个进程 `proc`中的 `p_pgrpnxt`则为空指针。

`proc`结构包含一个进程的所有信息。

• `p_pid`包含进程ID。

• `p_pptr`是指向父进程 `proc`结构的指针。

• `p_pgrp`指向本进程所属的进程组的 `pgrp`结构。

• `p_pgrpnxt`是指向进程中下一个进程的指针。

最后还有一个 `vnode`结构。在打开控制终端设备时分配此结构。进程对 `/dev/tty`的所有访问都通过 `vnode`结构。在图9-11中，实际 `i`节点是 `v`节点的一部分。3.10节曾提及这是4.3+BSD的实现方法，而SVR4则将 `v`节点存在 `i`节点中。

9.12 小结

本章说明了进程组之间的关系——对话期，它由若干个进程组组成。作业控制是当今很多UNIX系统所支持的功能，本章说明了它是如何由支持作业控制的 `shell`实现的。在这些进程关系中也涉及到了 `/dev/tty`。

所有这些进程的关系都使用了很多信号方面的功能。下一章将详细讨论UNIX中的信号机制。

习题

9.1 考虑6.7节中说明的 `utmp`和 `wtmp`文件，为什么 `logout`记录是由4.3+BSD的 `init` 进程写的？对于网络登录的处理与此相同吗？

9.2 编写一段程序，要求调用 `fork`并在子进程中建立一个新的对话期。验证子进程变成了进程组组长且不再有控制终端。

第10章 信号

10.1 引言

信号是软件中断。很多比较重要的应用程序都需处理信号。信号提供了一种处理异步事件的方法：终端用户键入中断键，则会通过信号机构停止一个程序。

UNIX的早期版本，就已经有信号机构，但是这些系统，例如 V7所提供的信号模型并不可靠。信号可能被丢失，而且在执行临界区代码时，进程很难关闭所选择的信号。4.3BSD和SVR3对信号模型都作了更改，增加了可靠信号机制。但是这两种更改之间并不兼容。幸运的是POSIX.1对可靠信号例程进行了标准化，这正是本章所说明的。

本章先对信号机制进行综述，并说明每种信号的一般用法。然后分析早期实现的问题。在分析存在的问题之后再说明解决这些问题的方法，这样有助于加深对改进机制的理解。本章也包含了很多并非100%正确的实例，这样做的目的是为了对其不足之处进行讨论。

10.2 信号的概念

首先，每个信号都有一个名字。这些名字都以三个字符 SIG开头。例如，SIGABRT是夭折信号，当进程调用abort函数时产生这种信号。SIGALRM是闹钟信号，当由alarm函数设置的时间已经超过后产生此信号。V7有15种不同的信号，SVR4和4.3+BSD均有31种不同的信号。

在头文件<signal.h>中，这些信号都被定义为正整数（信号编号）。没有一个信号其编号为0。在10.9节中将会看到kill函数，对信号编号0有特殊的应用。POSIX.1将此种信号编号值称为空信号。

很多条件可以产生一个信号。

- 当用户按某些终端键时，产生信号。在终端上按DELETE键通常产生中断信号（SIGINT）。这是停止一个已失去控制程序的方法。（第11章将说明此信号可被映射为终端上的任一字符。）

- 硬件异常产生信号：除数为0、无效的存储访问等等。这些条件通常由硬件检测到，并将其通知内核。然后内核为该条件发生时正在运行的进程产生适当的信号。例如，对执行一个无效存储访问的进程产生一个SIGSEGV。

- 进程用kill(2)函数可将信号发送给另一个进程或进程组。自然，有些限制：接收信号进程和发送信号进程的所有者必须相同，或发送信号进程的所有者必须是超级用户。

- 用户可用kill(1)命令将信号发送给其他进程。此程序是kill函数的界面。常用此命令终止一个失控的后台进程。

- 当检测到某种软件条件已经发生，并将其通知有关进程时也产生信号。这里并不是指硬件产生条件（如被0除），而是软件条件。例如SIGURG(在网络连接上传来非规定波特率的数据)、SIGPIPE(在管道的读进程已终止后一个进程写此管道)，以及SIGALRM(进程所设置的闹钟时间已经超时)。

信号是异步事件的经典实例。产生信号的事件对进程而言是随机出现的。进程不能只是测试一个变量(例如errno)来判别是否发生了一个信号，而是必须告诉内核“在此信号发生时，请

执行下列操作”。

可以要求系统在某个信号出现时按照下列三种方式中的一种进行操作。

(1) 忽略此信号。大多数信号都可使用这种方式进行处理，但有两种信号却决不能被忽略。它们是：SIGKILL和SIGSTOP。这两种信号不能被忽略的原因是：它们向超级用户提供一种使进程终止或停止的可靠方法。另外，如果忽略某些由硬件异常产生的信号（例如非法存储访问或除以0），则进程的行为是未定义的。

(2) 捕捉信号。为了做到这一点要通知内核在某种信号发生时，调用一个用户函数。在用户函数中，可执行用户希望对这种事件进行的处理。例如，若编写一个命令解释器，当用户用键盘产生中断信号时，很可能希望返回到程序的主循环，终止系统正在为该用户执行的命令。如果捕捉到SIGCHLD信号，则表示子进程已经终止，所以此信号的捕捉函数可以调用waitpid以取得该子进程的进程ID以及它的终止状态。又例如，如果进程创建了临时文件，那么可能要为SIGTERM信号编写一个信号捕捉函数以清除临时文件（kill命令传送的系统默认信号是终止信号）。

(3) 执行系统默认动作。表10-1给出了对每一种信号的系统默认动作。注意，对大多数信号的系统默认动作是终止该进程。

表10-1列出所有信号的名字，哪些系统支持此信号以及对于信号的系统默认动作。在POSIX.1列中，表示要求此种信号。job表示这是作业控制信号（仅当支持作业控制时，才要求此种信号）。

表10-1 UNIX信号

名 字	说 明	ANSI C	POSIX.1	SVR4	4.3+BSD	缺省动作
SIGABRT	异常终止 (abort)	•	•	•	•	终止w/core
SIGALRM	超时 (alarm)		•	•	•	终止
SIGBUS	硬件故障			•	•	终止w/core
SIGCHLD	子进程状态改变		作业	•	•	忽略
SIGCONT	使暂停进程继续		作业	•	•	继续/忽略
SIGEMT	硬件故障			•	•	终止w/core
SIGFPE	算术异常	•	•	•	•	终止w/core
SIGHUP	连接断开		•	•	•	终止
SIGILL	非法硬件指令	•	•	•	•	终止w/core
SIGINFO	键盘状态请求				•	忽略
SIGINT	终端中断符	•	•	•	•	终止
SIGIO	异步I/O			•	•	终止/忽略
SIGIOT	硬件故障			•	•	终止w/core
SIGKILL	终止		•	•	•	终止
SIGPIPE	写至无读进程的管道		•	•	•	终止
SIGPOLL	可轮询事件 (poll)			•		终止
SIGPROF	梗概时间超时 (setitimer)			•	•	终止
SIGPWR	电源失效/再起动			•		忽略
SIGQUIT	终端退出符		•	•	•	终止w/core
SIGSEGV	无效存储访问	•	•	•	•	终止w/core
SIGSTOP	停止		作业	•	•	暂停进程

(续)

名字	说明	ANSI C	POSIX.1	SVR4	4.3+BSD	缺省动作
SIGSYS	无效系统调用			•	•	终止w/core
SIGTERM	终止	•	•	•	•	终止
SIGTRAP	硬件故障			•	•	终止w/core
SIGTSTP	终端挂起符		作业	•	•	停止进程
SIGTTIN	后台从控制tty读		作业	•	•	停止进程
SIGTTOU	后台向控制tty写		作业	•	•	停止进程
SIGURG	紧急情况			•	•	忽略
SIGUSR1	用户定义信号	•		•	•	终止
SIGUSR2	用户定义信号	•		•	•	终止
SIGVTALRM	虚拟时间闹钟 (<i>setitimer</i>)			•	•	终止
SIGWINCH	终端窗口大小改变			•	•	忽略
SIGXCPU	超过CPU限制 (<i>setrlimit</i>)			•	•	终止w/core
SIGXFSZ	超过文件长度限制 (<i>setrlimit</i>)			•	•	终止w/core

在系统默认动作列，“终止w/core”表示在进程当前工作目录的core文件中复制了该进程的存储图像（该文件名为core，由此可以看出这种功能很久之前就是UNIX功能的一部分）。大多数UNIX调试程序都使用core文件以检查进程在终止时的状态。在下列条件下不产生core文件：(a)进程是设置-用户-ID，而且当前用户并非程序文件的所有者，或者(b)进程是设置-组-ID，而且当前用户并非该程序文件的组所有者，或者(c)用户没有写当前工作目录的许可权，或者(d)文件太大(回忆7.11节中的RLIMIT_CORE)。core文件的许可权(假定该文件在此之前并不存在)通常是用户读/写，组读和其他读。

core文件的产生不是POSIX.1所属部分，而是很多UNIX版本的实现特征。

UNIX第6版没有检查条件(a)和(b)，并且其源代码中包含如下说明：“如果你正在找寻保护信号，那么当设置-用户-ID命令执行时，将可能产生大量的这种信号”。

4.3+BSD产生名为core.prog的文件，其中prog是被执行的程序名的前16个字符。它对core文件给予了某种标识，所以是一种改进特征。

表10-1“硬件故障”对应于实现定义的硬件故障。这些名字中有很多取自UNIX早先在PDP-11上的实现。请查看你所使用的系统的手册，以确切地确定这些信号对应于哪些错误类型。

下面比较详细地说明这些信号。

- SIGABRT 调用abort函数时（见10.17节）产生此信号。进程异常终止。
- SIGALRM 超过用alarm函数设置的时间时产生此信号。详细情况见10.10节。若由setitimer(2)函数设置的间隔时间已经过时，那么也产生此信号。
- SIGBUS 指示一个实现定义的硬件故障。
- SIGCHLD 在一个进程终止或停止时，SIGCHLD信号被送给其父进程。按系统默认，将忽略此信号。如果父进程希望了解其子进程的这种状态改变，则应捕捉此信号。信号捕捉函数中通常要调用wait函数以取得子进程ID和其终止状态。

系统V的早期版本有一个名为 SIGCLD(无H)的类似信号。这一信号具有非标准的语义，SVR2的手册页警告在新的程序中尽量不要使用这种信号。应用程序应当使用标准的 SIGCHLD 信号。10.7节将讨论这两个信号。

- SIGCONT 此作业控制信号送给需要继续运行的处于停止状态的进程。如果接收到此信号的进程处于停止状态，则系统默认动作是使该进程继续运行，否则默认动作是忽略此信号。例如，vi编辑程序在捕捉到此信号后，重新绘制终端屏幕。关于进一步的情况见 10.20节。

- SIGEMT 指示一个实现定义的硬件故障。

EMT这一名字来自PDP-11的emulator trap 指令。

- SIGFPE 此信号表示一个算术运算异常，例如除以0，浮点溢出等。

- SIGHUP 如果终端界面检测到一个连接断开，则将此信号送给与该终端相关的控制进程（对话期首进程）。见图9-11，此信号被送给session结构中s_leader字段所指向的进程。仅当终端的CLOCAL标志没有设置时，在上述条件下才产生此信号。（如果所连接的终端是本地的，才设置该终端的CLOCAL标志。它告诉终端驱动程序忽略所有调制解调器的状态行。第11章将说明如何设置此标志。）注意，接到此信号的对话期首进程可能在后台，作为一个例子见图9-7。这区别于通常由终端产生的信号（中断、退出和挂起），这些信号总是传递给前台进程组。

如果对话期前进程终止，则也产生此信号。在这种情况下，此信号送给前台进程组中的每一个进程。

通常用此信号通知精灵进程（见第13章）以再读它们的配置文件。选用SIGHUP的理由是，因为一个精灵进程不会有一个控制终端，而且通常决不会接收到这种信号。

- SIGILL 此信号指示进程已执行一条非法硬件指令。

4.3BSD由abort函数产生此信号。SIGABRT现在被用于此。

- SIGINFO 这是一种4.3+BSD信号，当用户按状态键（一般采用Ctrl-T）时，终端驱动程序产生此信号并送至前台进程组中的每一个进程（见图9-8）。此信号通常造成在终端上显示前台进程组中各进程的状态信息。

- SIGINT 当用户按中断键（一般采用DELETE或Ctrl-C）时，终端驱动程序产生此信号并送至前台进程组中的每一个进程（见图9-8）。当一个进程在运行时失控，特别是它正在屏幕上产生大量不需要的输出时，常用此信号终止它。

- SIGIO 此信号指示一个异步I/O事件。在12.6.2节中将对此进行讨论。

在表10-1中，对SIGIO的系统默认动作是终止或忽略。不幸的是，这依赖于系统。在SVR4中，SIGIO与SIGPOLL相同，其默认动作是终止此进程。在4.3+BSD中（此信号起源于4.2BSD），其默认动作是忽略。

- SIGIOT 这指示一个实现定义的硬件故障。

IOT这个名字来自于PDP-11对于输入/输出TRAP(input/output TRAP)指令的缩写。系统V的早期版本，由abort函数产生此信号。SIGABRT现在被用于此。

- SIGKILL 这是两个不能被捕捉或忽略信号中的一个。它向系统管理员提供了一种可以

杀死任一进程的可靠方法。

- SIGPIPE 如果在读进程已终止时写管道，则产生此信号。14.2节将说明管道。当套接口的一端已经终止时，若进程写该套接口也产生此信号。

- SIGPOLL 这是一种SVR4信号，当在一个可轮询设备上发生一特定事件时产生此信号。12.5.2节将说明poll函数和此信号。它与4.3+BSD的SIGIO和SIGURG信号接近。

- SIGPROF 当setitimer(2)函数设置的梗概统计间隔时间已经超过时产生此信号。

- SIGPWR 这是一种SVR4信号，它依赖于系统。它主要用于具有不间断电源(UPS)的系统上。如果电源失效，则UPS起作用，而且通常软件会接到通知。在这种情况下，系统依靠蓄电池电源继续运行，所以无须作任何处理。但是如果蓄电池也将不能支持工作，则软件通常会再次接到通知，此时，它在15~30秒内使系统各部分都停止运行。此时应当传递SIGPWR信号。在大多数系统中使接到蓄电池电压过低的进程将信号SIGPWR发送给init进程，然后由init处理停机操作。很多系统V的init实现在inittab文件中提供了两个记录项用于此种目的；powerfail以及powerwait。

目前已能获得低价格的UPS系统，它用RS-232串行连接能够很容易地将蓄电池电压过低的条件通知系统，于是这种信号也就更加重要了。

- SIGQUIT 当用户在终端上按退出键（一般采用Ctrl-\）时，产生此信号，并送至前台进程组中的所有进程（见图9-8）。此信号不仅终止前台进程组（如SIGINT所做的那样），同时产生一个core文件。

- SIGSEGV 指示进程进行了一次无效的存储访问。

名字SEGV表示“段违例（segmentation violation）”。

- SIGSTOP 这是一个作业控制信号，它停止一个进程。它类似于交互停止信号(SIGTSTP)，但是SIGSTOP不能被捕捉或忽略。

- SIGSYS 指示一个无效的系统调用。由于某种未知原因，进程执行了一条系统调用指令，但其指示系统调用类型的参数却是无效的。

- SIGTERM 这是由kill(1)命令发送的系统默认终止信号。

- SIGTRAP 指示一个实现定义的硬件故障。

此信号名来自于PDP-11的TRAP指令。

- SIGTSTP 交互停止信号，当用户在终端上按挂起键[⊖]（一般采用Ctrl-Z）时，终端驱动程序产生此信号。

- SIGTTIN 当一个后台进程组进程试图读其控制终端时，终端驱动程序产生此信号。（见9.8节中对此问题的讨论。）在下列例外情形下不产生此信号，此时读操作返回出错，errno设置为EIO：(a)读进程忽略或阻塞此信号，或(b)读进程所属的进程组是孤儿进程组。

- SIGTTOU 当一个后台进程组进程试图写其控制终端时产生此信号。（见9.8节对此问题的讨论。）与上面所述的SIGTTIN信号不同，一个进程可以选择为允许后台进程写控制终端。第11章将讨论如何更改此选择项。

如果不允许后台进程写，则与SIGTTIN相似也有两种特殊情况：(a)写进程忽略或阻塞此信

⊖ 不幸的是，术语停止(stop)有不同的意义。在讨论作业控制和信号时我们需提及停止和继续作业。但是终端驱动程序一直用术语停止表示用Ctrl-S和Ctrl-Q字符停止和起端终端输出。因此，终端驱动程序将产生交互停止信号和字符称之为挂起字符而非停止字符。

号, 或(b)写进程所属进程组是孤儿进程组。在这两种情况下不产生此信号, 写操作返回出错, `errno`设置为EIO。

不论是否允许后台进程写, 某些除写以外的下列终端操作也能产生此信号: `tcsetattr`, `tcsendbreak`, `tcdrain`, `tcflush`, `tcflow` 以及 `tcsetpgrp`。第11章将说明这些终端操作。

- **SIGURG** 此信号通知进程已经发生一个紧急情况。在网络连接上, 接到非规定波特率的数据时, 此信号可选择地产生。

- **SIGUSR1** 这是一个用户定义的信号, 可用于应用程序。

- **SIGUSR2** 这是一个用户定义的信号, 可用于应用程序。

- **SIGVTALRM** 当一个由 `setitimer(2)` 函数设置的虚拟间隔时间已经超过时产生此信号。

- **SIGWINCH** SVR4和4.3+BSD内核保持与每个终端或伪终端相关联的窗口的大小。一个进程可以用 `ioctl` 函数(见11.12节)得到或设置窗口的大小。如果一个进程用 `ioctl` 的设置-窗口-大小命令更改了窗口大小, 则内核将 **SIGWINCH** 信号送至前台进程组。

- **SIGXCPU** SVR4和4.3+BSD支持资源限制的概念(见7.11节)。如果进程超过了其软CPU时间限制, 则产生此信号。

- **SIGXFSZ** 如果进程超过了其软文件长度限制(见7.11节), 则SVR4和4.3+BSD产生此信号。

10.3 signal函数

UNIX信号机制最简单的界面是 `signal` 函数。

```
#include <signal.h>

void (*signal (int signo, void (func)(int))) (int);
```

返回: 成功则为以前的信号处理配置, 若出错则为 `SIG_ERR`

`signal` 函数由ANSI C定义。因为ANSI C不涉及多进程、进程组、终端 I/O 等, 所以它对信号的定义非常含糊, 以致于对 UNIX 系统而言几乎毫无用处。确实, ANSI C 对信号的说明只用了2页, 而POSIX.1的说明则用了15页。

SVR4也提供 `signal` 函数, 该函数可提供老的SVR2不可靠信号语义(10.4节将说明这些老的语义)。提供此函数主要是为了向下兼容要求此老语义的应用程序, 新应用程序不应使用它。

4.3+BSD也提供 `signal` 函数, 但是它是用 `sigaction` 函数实现的(10.14节将说明 `sigaction` 函数), 所以在4.3+BSD之下使用它提供新的可靠的信号语义。

在讨论 `sigaction` 函数时, 提供了使用该函数的 `signal` 的一个实现。本书中的所有实例均使用程序10-12中给出的 `signal` 函数。

`signo` 参数是表10-1中的信号名。 `func` 的值是: (a) 常数 `SIG_IGN`, 或(b) 常数 `SIG_DFL`, 或(c) 当接到此信号后要调用的函数的地址。如果指定 `SIG_IGN`, 则向内核表示忽略此信号。(记住有两个信号 `SIGKILL` 和 `SIGSTOP` 不能忽略。) 如果指定 `SIG_DFL`, 则表示接到此信号后的动作是系统默认动作(见表10-1中的最后1列)。当指定函数地址时, 我们称此为捕捉此信号。我们称此函数为信号处理程序 (`signal handler`) 或信号捕捉函数 (`signal-catching function`)。

signal函数的原型说明此函数要求两个参数，返回一个函数指针，而该指针所指向的函数无返回值(void)。第一个参数`signo`是一个整型数，第二个参数是函数指针，它所指向的函数需要一个整型参数，无返回值。用一般语言来描述也就是要向信号处理程序传送一个整型参数，而它却无返回值。当调用signal设置信号处理程序时，第二个参数是指向该函数(也就是信号处理程序)的指针。signal的返回值则是指向以前的信号处理程序的指针。

很多系统用附加的依赖于实现的参数来调用信号处理程序。10.21节将说明可选择的SVR4和4.3+BSD参数。

本节开头所示的signal函数原型太复杂了，如果使用下面的typedef [Plauger 1992]，则可使其简单一些。

```
typedef void Sigfunc(int);
```

然后，可将signal函数原型写成：

```
Sigfunc *signal(int, Sigfunc *);
```

我们已将此typedef包括在ourhdr.h文件中(见附录B)，并随本章中的函数一起使用。

如果查看系统的头文件<signal.h>，则可能会找到下列形式的说明：

```
#define SIG_ERR (void (*)())-1
#define SIG_DFL (void (*)())0
#define SIG_IGN (void (*)())1
```

这些常数可用于表示“指向函数的指针，该函数要一个整型参数，而且无返回值”。signal的第二个参数及其返回值就可用它们表示。这些常数所使用的三个值不一定要是-1, 0和1。它们必须是三个值而决不能是任一可说明函数的地址。大多数UNIX系统使用上面所示的值。

实例

程序10-1显示了一个简单的信号处理程序，它捕捉两个用户定义的信号并打印信号编号。10.10节将说明pause函数，它使调用进程睡眠。

程序10-1 捕捉SIGUSR1和SIGUSR2的简单处理程序

```
#include <signal.h>
#include "ourhdr.h"

static void sig_usr(int); /* one handler for both signals */

int
main(void)
{
    if (signal(SIGUSR1, sig_usr) == SIG_ERR)
        err_sys("can't catch SIGUSR1");
    if (signal(SIGUSR2, sig_usr) == SIG_ERR)
        err_sys("can't catch SIGUSR2");

    for ( ; ; )
        pause();
}

static void
sig_usr(int signo) /* argument is signal number */
{
    if (signo == SIGUSR1)
```

```

    printf("received SIGUSR1\n");
else if (signo == SIGUSR2)
    printf("received SIGUSR2\n");
else
    err_dump("received signal %d\n", signo);
return;
}

```

我们使该程序在后台运行，并且用kill(1)命令将信号送给它。注意，在UNIX中，杀死(kill)这个术语是不恰当的。kill(1)命令和kill(2)函数只是将一个信号送给一个进程或进程组。该信号是否终止该进程则取决于该信号的类型，以及该进程是否安排了捕捉该信号。

```

$ a.out &                                在后台启动进程
[1] 4720                                   作业控制shell打印作业号和进程ID
$ kill -USR1 4720                          向该进程发送 SIGUSR1
received SIGUSR1
$ kill -USR2 4720                          向该进程发送 SIGUSR2
received SIGUSR2
$ kill 4720                                 向该进程发送 SIGTERM
[1] + Terminated a.out &

```

当向该进程发送SIGTERM信号后，该进程就终止，因为它不捕捉此信号，而对此信号的系统默认动作是终止。

10.3.1 程序启动

当执行一个程序时，所有信号的状态都是系统默认或忽略。通常所有信号都被设置为系统默认动作，除非调用exec的进程忽略该信号。比较特殊的是，exec函数将原先设置为要捕捉的信号都更改为默认动作，其他信号的状态则不变（一个进程原先要捕捉的信号，当其执行一个新程序后，就自然地不能再捕捉了，因为信号捕捉函数的地址很可能在所执行的新程序文件中已无意义）。

我们经常会碰到的一个具体例子是一个交互 shell如何处理对后台进程的中断和退出信号。对于一个非作业控制shell，当在后台执行一个进程时，例如：

```
cc main.c &
```

shell自动将后台进程中对中断和退出信号的处理方式设置为忽略。于是，当按中断键时就不会影响到后台进程。如果没有这样的处理，那么当按中断键时，它不但终止前台进程，也终止所有后台进程。

很多捕捉这两个信号的交互程序具有下列形式的代码：

```

int sig_int(), sig_quit();

if (signal(SIGINT, SIG_IGN) != SIG_IGN)
    signal(SIGINT, sig_int);
if (signal(SIGQUIT, SIG_IGN) != SIG_IGN)
    signal(SIGQUIT, sig_quit);

```

这样处理后，仅当SIGINT和SIGQUIT当前并不忽略，进程才捕捉它们。

从signal的这两个调用中也可以看到这种函数的限制：不改变信号的处理方式就不能确定信号的当前处理方式。我们将在本章的稍后部分说明使用 sigaction函数可以确定一个信号的处理方式，而无需改变它。

10.3.2 进程创建

当一个进程调用fork时，其子进程继承父进程的信号处理方式。因为子进程在开始时复制了父进程存储图像，所以信号捕捉函数的地址在子进程中是有意义的。

10.4 不可靠的信号

在早期的UNIX版本中（例如V7），信号是不可靠的。不可靠在这里指的是，信号可能会被丢失——一个信号发生了，但进程却决不会知道这一点。那时，进程对信号的控制能力也很低，它能捕捉信号或忽略它，但有些很需要的功能它却并不具备。例如，有时用户希望通知内核阻塞一信号——不要忽略该信号，在其发生时记住它，然后在进程作好了准备时再通知它。这种阻塞信号的能力当时并不具备。

4.2BSD对信号机构进行了更改，提供了被称之为可靠信号的机制。然后，SVR3也修改了信号机制，提供了另一套系统V可靠信号机制。POSIX.1选择了BSD模型作为其标准化的基础。

早期版本中的一个问题是在进程每次处理信号时，随即将信号动作复置为默认值（在前面运行程序10-1时，我们通过只捕捉每种信号各一次避免了这一点）。以下是早期版本中关于如何处理中断信号的经典实例的代码：

```
int    sig_int();          /* my signal handling function */
...
signal(SIGINT, sig_int); /* establish handler */
...

sig_int()
{
    signal(SIGINT, sig_int);
        /* reestablish handler for next occurrence */
    ...
        /* process the signal ... */
}
```

由于早期的C语言版本不支持ANSI C的void数据类型，所以将信号处理程序说明为int类型。

这种代码段的一个问题是：在信号发生之后到信号处理程序中调用 signal函数之间有一个时间窗口。在此期间中，可能发生另一次中断信号。第二个信号会造成执行默认动作，而对中断信号则是终止该进程。这种类型的程序段在大多数情况下会正常工作，使得我们认为它们正确，而实际上却并不是如此。

这些早期版本的另一个问题是：在进程不希望某种信号发生时，它不能关闭该信号。进程能做的就是忽略该信号。有时希望通知系统“阻止下列信号发生，如果它们确实产生了，请记住它们。”这种问题的一个经典实例是下列程序段，它捕捉一个信号，然后设置一个表示该信号已发生的标志：

```
int    sig_int_flag;      /* set nonzero when signal occurs */

main()
{
    int    sig_int();      /* my signal handling function */
    ...
```

```

    signal(SIGINT, sig_int); /* establish handler */
    ...
    while (sig_int_flag == 0)
        pause();           /* go to sleep, waiting for signal */
    ...
}

sig_int()
{
    signal(SIGINT, sig_int); /* reestablish handler for next time */
    sig_int_flag = 1;       /* set flag for main loop to examine */
}

```

其中，进程调用 `pause` 函数使自己睡眠，直到捕捉到一个信号。当信号被捕捉到后，信号处理程序将标志 `sig_int_flag` 设置为非 0。在信号处理程序返回之后，内核将该进程唤醒，它检测到该标志为非 0，然后执行它所需做的。但是这里也有一个时间窗口，可能使操作错误。如果在测试 `sig_int_flag` 之后，调用 `paust` 之前发生信号，则此进程可能会一直睡眠（假定此信号不再产生）。于是，这次发生的信号也就丢失了。还有另一个例子，某段代码并不正确，但是大多数时间却能正常工作。要查找并排除这种类型的问题很困难。

10.5 中断的系统调用

早期 UNIX 系统的一个特性是：如果在进程执行一个低速系统调用而阻塞期间捕捉到一个信号，则该系统调用就被中断不再继续执行。该系统调用返回出错，其 `errno` 设置为 `EINTR`。这样处理的理由是：因为一个信号发生了，进程捕捉到了它，这意味着已经发生了某种事情，所以是个好机会应当唤醒阻塞的系统调用。

在这里，我们必须区分系统调用和函数。当捕捉到某个信号时，被中断的是内核中执行的系统调用。

为了支持这种特性，将系统调用分成两类：低速系统调用和其他系统调用。低速系统调用是可能会使进程永远阻塞的一类系统调用，它们包括：

- 在读某些类型的文件时，如果数据并不存在则可能会使调用者永远阻塞（管道、终端设备以及网络设备）。
- 在写这些类型的文件时，如果不能立即接受这些数据，则也可能使调用者永远阻塞。
- 打开文件，在某种条件发生之前也可能使调用者阻塞（例如，打开终端设备，它要等待直到所连接的调制解调器回答了电话）。
- `pause`（按照定义，它使调用进程睡眠直至捕捉到一个信号）和 `wait`。
- 某种 `ioctl` 操作。
- 某些进程间通信函数（见第 14 章）。

在这些低速系统调用中一个值得注意的例外是与磁盘 I/O 有关的系统调用。虽然读、写一个磁盘文件可能暂时阻塞调用者（在磁盘驱动程序将请求排入队列，然后在适当时间执行请求期间），但是除非发生硬件错误，I/O 操作总会很快返回，并使调用者不再处于阻塞状态。

可以用中断系统调用这种方法来处理的一种情况是：一个进程起动了读终端操作，而使用该终端设备的用户却离开该终端很长时间。在这种情况下进程可能处于阻塞状态几个小时甚至数天，除非系统停机，否则一直如此。

与被中断的系统调用相关的问题是必须用显式方法处理出错返回。典型的代码序列（假定

进行一个读操作，它被中断，我们希望重新启动它)可能如下列样式：

```
again:
    if ( ( n = read(fd, buff, BUFSIZE) ) < 0 ) {
        if (errno == EINTR)
            goto again;    /* just an interrupted system call */
        /* handle other errors */
    }
```

为了帮助应用程序使其不必处理被中断的系统调用，4.2BSD引进了某些被中断的系统调用的自动再启动。自动再启动的系统调用包括：ioctl、read、readv、write、writev、wait和waitpid。正如前述，其中前五个函数只有对低速设备进行操作时才会被信号中断。而wait和waitpid在捕捉到信号时总是被中断。某些应用程序并不希望这些函数被中断后再启动，因为这种自动再启动的处理方式也会带来问题，为此4.3BSD允许进程在每个信号各别处理的基础上不使用此功能。

POSIX.1允许实现再启动系统调用，但这并不是必需的。

系统V的默认工作方式是不再起启动系统调用。但是SVR4使用sigaction时（见10.14节），可以指定SA_RESTART选择项以再起启动由该信号中断的系统调用。

在4.3+BSD中，系统调用的再起启动依赖于调用了哪一个函数设置信号处理方式配置。早期的与4.3BSD兼容的sigvec函数使被该信号中断的系统调用自动再起启动。但是，使用较新的与POSIX.1兼容的sigaction则不使它们再起启动。但如同在SVR4中一样，在sigaction中可以使用SA_RESTART选择项，使内核再起启动由该信号中断的系统调用。

4.2BSD引进自动再起启动功能的一个理由是：有时用户并不知道所使用的输入、输出设备是否是低速设备。如果我们编写的程序可以用交互方式运行，则它可能读、写终端低速设备。如果在程序中捕捉信号，而系统却不提供再起启动功能，则对每次读、写系统调用就要进行是否出错返回的测试，如果是被中断的，则再进行读、写。

表10-2列出了几种实现所提供的信号功能及它们的语义。

表10-2 几种信号实现所提供的功能

函 数	系 统	信号处理程序仍被安装	阻塞信号的能力	被中断系统调用的自动再起启动
signal,	V7, SVR2, SVR3, SVR4			决不
sigset, sighold, sigrelse sigignore, sigpause	SVR3, SVR4	•	•	决不
signal, sigvec, sigblock	4.2BSD	•	•	总是
sigsetmask, sigpause	4.3BSD, 4.3+BSD	•	•	默认
sigaction, sigprocmask	POSIX.1	•	•	未说明
sigpending, sigsuspend	SVR4	•	•	可选
	4.3+BSD	•	•	可选

应当了解，其他厂商提供的 UNIX 系统可能会有不同于表 10-2 中所示的处理情况。例如，SunOS 4.1.2 中的 `sigaction` 其默认方式是再启动被中断的系统调用，这与 SVR4 和 4.3+BSD 都不同。

程序 10-12 提供了我们自己的 `signal` 函数版本，它试图重新启动被中断的系统调用（除 `SIGALRM` 信号外）。程序 10-13 则提供了另一个函数 `signal_intr`，它不进行再启动。

在所有程序实例中，我们都有目的地显示了信号处理程序的返回（如果它返回的话），这种返回可能会中断一个系统调用。

12.5 节说明 `select` 和 `poll` 函数时还将涉及被中断的系统调用。

10.6 可再入函数

进程捕捉到信号并继续执行时，它首先执行该信号处理程序中的指令。如果从信号处理程序返回（例如没有调用 `exit` 或 `longjmp`），则继续执行在捕捉到信号时进程正在执行的正常指令序列（这类似于硬件中断发生时所做的）。但在信号处理程序中，不能判断捕捉到信号时进程执行到何处。如果进程正在执行 `malloc`，在其堆中分配额外的存储空间，而此时由于捕捉到信号插入执行该信号处理程序，其中又调用 `malloc`，这时会发生什么？又例如若进程正在执行 `getpwnam`（见 6.2 节）这种将其结果存放在静态存储单元中的函数，而插入执行的信号处理程序中又调用这样的函数，这时又会发生什么呢？在 `malloc` 例子中，可能会对进程造成破坏，因为 `malloc` 通常为它所分配的存储区保持一个连接表，而插入执行信号处理程序时，进程可能正在更改此连接表。在 `getpwnam` 的例子中，正常返回给调用者的信息可能由返回至信号处理程序的信息覆盖。

POSIX.1 说明了保证可再入的函数。表 10-3 列出了这些可再入函数。图中四个带 * 号的函数并没有按 POSIX.1 说明为是可再入的，但 SVR4 SVID [AT&T 1989] 则将它们列为是可再入的。

表 10-3 信号处理程序中可以调用的可再入函数

<code>_exit</code>	<code>fork</code>	<code>pipe</code>	<code>stat</code>
<code>abort*</code>	<code>fstat</code>	<code>read</code>	<code>sysconf</code>
<code>access</code>	<code>getegid</code>	<code>rename</code>	<code>tcdrain</code>
<code>alarm</code>	<code>geteuid</code>	<code>rmdir</code>	<code>tcflow</code>
<code>cfgetispeed</code>	<code>getgid</code>	<code>setgid</code>	<code>tcflush</code>
<code>cfgetospeed</code>	<code>getgroups</code>	<code>setpgid</code>	<code>tcgetattr</code>
<code>cfsetispeed</code>	<code>getpgrp</code>	<code>setsid</code>	<code>tcgetpgrp</code>
<code>cfsetospeed</code>	<code>getpid</code>	<code>setuid</code>	<code>tcsendbreak</code>
<code>chdir</code>	<code>getppid</code>	<code>sigaction</code>	<code>tcsetattr</code>
<code>chmod</code>	<code>getuid</code>	<code>sigaddset</code>	<code>tcsetpgrp</code>
<code>chown</code>	<code>kill</code>	<code>sigdelset</code>	<code>time</code>
<code>close</code>	<code>link</code>	<code>sigemptyset</code>	<code>times</code>
<code>creat</code>	<code>longjmp*</code>	<code>sigfillset</code>	<code>umask</code>
<code>dup</code>	<code>lseek</code>	<code>sigismember</code>	<code>uname</code>
<code>dup2</code>	<code>mkdir</code>	<code>signal*</code>	<code>unlink</code>
<code>execle</code>	<code>mkfifo</code>	<code>sigpending</code>	<code>utime</code>
<code>execve</code>	<code>open</code>	<code>sigprocmask</code>	<code>wait</code>
<code>exit*</code>	<code>pathconf</code>	<code>sigsuspend</code>	<code>waitpid</code>
<code>fcntl</code>	<code>pause</code>	<code>sleep</code>	<code>write</code>

没有列入表 10-3 中的大多数函数是不可再入的，其原因为：(a) 已知它们使用静态数据结构，或 (b) 它们调用 malloc 或 free，或 (c) 它们是标准 I/O 函数。标准 I/O 库的很多实现都以不可再入方式使用全局数据结构。

要了解在信号处理程序中即使调用列于表 10-3 中的函数，因为每个进程只有一个 errno 变量，所以我们可能修改了其原先的值。考虑一个信号处理程序，它恰好在 main 刚设置 errno 之后被调用。如果该信号处理程序调用 read，则它可能更改 errno 的值，从而取代了刚由 main 设置的值。因此，作为一个通用的规则，当在信号处理程序中调用表 10-3 中列出的函数时，应当在其前保存，在其后恢复 errno。（要了解经常被捕捉到的信号是 SIGCHLD，其信号处理程序通常要调用一种 wait 函数，而各种 wait 函数都能改变 errno。）

POSIX.1 没有包括表 10-3 中的 longjmp 和 siglongjmp（10.15 节将说明 siglongjmp 函数）。这是因为在主例程以非再入方式正在更新一数据结构时可能产生信号。不是从信号处理程序返回而是调用 siglongjmp，可能使该数据结构是部分更新的。如果应用程序将要做更新全局数据结构这样的事情，而同时规定要捕捉某些信号，而这些信号的处理程序又会引起执行 sigsetjmp，则在更新这种数据结构时要阻塞此信号。

实例

在程序 10-2 中，信号处理程序 my_alarm 调用不可再入函数 getpwnam，而 my_alarm 每秒钟被调用一次。10.10 节中将说明 alarm 函数。在程序 10-2 中用其每秒产生一次 SIGALRM 信号。

运行此程序时，其结果具有随意性。通常，在信号处理程序第一次返回时，该程序将由 SIGSEGV 信号终止。检查 core 文件，从中可以看到 main 函数已调用 getpwnam，而且当信号处理程序调用此同一函数时，某些内部指针出了问题。偶然，此程序会运行若干秒，然后因产生 SIGSEGV 信号而终止。在捕捉到信号后，若 main 函数仍正确运行，其返回值却有时错误，有时正确。有时在信号处理程序中调用 getpwnam 会出错返回，其出错值为 EBADF（无效文件描述符）。

从此实例中可以看出，若在信号处理程序中调用一个不可再入函数，则其结果是不可预见的。

程序 10-2 在信号处理程序中调用不可再入函数

```
#include <pwd.h>
#include <signal.h>
#include "ourhdr.h"

static void my_alarm(int);

int
main(void)
{
    struct passwd *ptr;
    signal(SIGALRM, my_alarm);
    alarm(1);
    for ( ; ; ) {
        if ( (ptr = getpwnam("stevens")) == NULL)
            err_sys("getpwnam error");
        if (strcmp(ptr->pw_name, "stevens") != 0)
            printf("return value corrupted!, pw_name = %s\n",
                ptr->pw_name);
    }
}
```

```

static void
my_alarm(int signo)
{
    struct passwd  *rootptr;

    printf("in signal handler\n");
    if ( (rootptr = getpwnam("root")) == NULL)
        err_sys("getpwnam(root) error");
    alarm(1);
    return;
}

```

10.7 SIGCLD语义

SIGCLD和SIGCHLD这两个信号经常易于混淆。SIGCLD是系统V的一个信号名，其语义与名为SIGCHLD的BSD信号不同。POSIX.1则采用BSD的SIGCHLD信号。

BSD的SIGCHLD信号的语义与其他信号的语义相类似。子进程状态改变后产生此信号，父进程需要调用一个wait函数以检测发生了什么。

由于历史原因，系统V处理SIGCLD信号的方式不同于其他信号。如果用signal或sigset（设置信号配置的早期的与SRV3兼容性函数）设置信号配置，则SVR4继续了这一具有问题色彩的传统（即兼容性限制）。对于SIGCLD早期的处理方式是：

(1) 如果进程特地指定对该信号的配置为SIG_IGN，则调用进程的子进程将不产生僵死进程。注意，这与其默认动作（SIG_DFL）忽略（见表10-1）不同。代之以，在子进程终止时，将其状态丢弃。如果调用进程最后调用一个wait函数，那么它将阻塞到所有子进程都终止，然后该wait会返回-1，其errno则设置为ECHILD。（此信号的默认配置是忽略，但这不会造成上述语义。代之以我们必须特地指定其配置为SIG_IGN。）

POSIX.1并未说明在SIGCHLD被忽略时应产生的后果，所以这种行为是允许的。

4.3+BSD中，如SIGCHLD被忽略，则允许产生僵死子进程。如果要避免僵死子进程，则必须等待子进程。

在SVR4中，如果调用signal或sigset将SIGCHLD的配置设置为忽略，则不会产生僵死子进程。另外，使用SVR4版的sigaction，则可设置SA_NOCLDWAIT标志(见表10-5)以避免子进程僵死。

(2) 如果将SIGCLD的配置设置为捕捉，则内核立即检查是否有子进程准备好被等待，如果是这样，则调用SIGCLD处理程序。

第(2)项改变了为此信号编写处理程序的方法。

实例

10.4节曾提到进入信号处理程序后，首先要调用signal函数以再设置此信号处理程序。（在信号被复置为其默认值时，它可能被丢失，立即重新设置可以减少此窗口时间。）程序10-3显示了这一点。但此程序不能正常工作。如果在SVR2下编译并运行此程序，则其输出一行行地不断重复“SIGCLD received”。最后进程用完其栈空间并异常终止。

此程序的问题是：在信号处理程序的开始处调用signal，按照上述第(2)项，内核检查是否有需要等待的子进程（因为我们正在处理一个SIGCLD，所以确实有这种子进程），

所以它产生另一个对信号处理程序的调用。信号处理程序调用 `signal`，整个过程再次重复。

为了解决这一问题，应当在调用 `wait` 取到子进程的终止状态后再调用 `signal`。此时仅当其他子进程终止，内核才会再次产生此种信号。

如果为SIGCHLD建立了一个信号处理程序，又存在一个已终止但尚未等待的进程，则是否会产生信号？POSIX.1对此没有作说明。这样就允许前面所述的工作方式。但是，因为POSIX.1在信号发生时并没有将信号配置复置为其默认值（假定正用POSIX.1的`sigaction`函数设置其配置），于是在SIGCHLD处理程序中也就不必再为该信号指定一个信号处理程序。

务必了解你所用的系统中SIGCHLD信号的语义。也应了解在某些系统中`#define SIGCHLD`为SIGCLD或反之。更改这种信号的名字使你可以编译为另一个系统编写的程序，但是如果该程序使用该信号的另一语义，则这样的程序也不能工作。

程序10-3 不能正常工作的系统V SIGCLD处理程序

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

static void sig_cld();

int
main()
{
    pid_t  pid;

    if (signal(SIGCLD, sig_cld) == -1)
        perror("signal error");

    if ( (pid = fork()) < 0)
        perror("fork error");
    else if (pid == 0) {          /* child */
        sleep(2);
        _exit(0);
    }
    pause();    /* parent */
    exit(0);
}

static void
sig_cld()
{
    pid_t  pid;
    int    status;

    printf("SIGCLD received\n");
    if (signal(SIGCLD, sig_cld) == -1) /* reestablish handler */
        perror("signal error");

    if ( (pid = wait(&status)) < 0)    /* fetch child status */
        perror("wait error");
    printf("pid = %d\n", pid);
    return;    /* interrupts pause() */
}
```

10.8 可靠信号术语和语义

我们需要定义一些在讨论信号时会用到的术语。首先，当造成信号的事件发生时，为进程产生一个信号（或向一个进程发送一个信号）。事件可以是硬件异常（例如除以0）、软件条件（例如，闹钟时间超过）、终端产生的信号或调用 kill 函数。在产生了信号时，内核通常在进程表中设置某种形式的一个标志。当对信号做了这种动作时，我们说向一个进程递送了一个信号。在信号产生（generation）和递送（delivery）之间的时间间隔内，称信号未决（pending）。

进程可以选用“信号递送阻塞”。如果为进程产生了一个选择为阻塞的信号，而且对该信号的动作是系统默认动作或捕捉该信号，则为该进程将此信号保持为未决状态，直到该进程 (a) 对此信号解除了阻塞，或者 (b) 将对此信号的动作更改为忽略。当递送一个原来被阻塞的信号给进程时，而不是在产生该信号时，内核才决定对它的处理方式。于是进程在信号递送给它之前仍可改变对它的动作。进程调用 sigpending 函数（见 10.13 节）将指定的信号设置为阻塞和未决。

如果在进程解除对某个信号的阻塞之前，这种信号发生了多次，那么将如何呢？POSIX.1 允许系统递送该信号一次或多次。如果递送该信号多次，则称这些信号排了队。但是大多数 UNIX 并不对信号排队。代之以，UNIX 内核只递送这种信号一次。

系统 V 早期版本的手册页称 SIGCLD 信号是用排队方式处理的，但实际并非如此。代之以，内核按 10.7 节中所述方式产生此信号。AT&T [1990e] 的 sigaction(2) 手册页称 SA-SIGINFO 标志（见表 10-5）使信号可靠地排队，这也不正确。表面上此功能存在于内核中，但在 SVR4 中并不起作用。

如果有多个信号要递送给一个进程，那么将如何呢？POSIX.1 并没有规定这些信号的递送顺序。但是 POSIX.1 建议：与进程当前状态有关的信号，例如 SIGSEGV 在其他信号之前递送。

每个进程都有一个信号屏蔽字，它规定了当前要阻塞递送到该进程的信号集。对于每种可能的信号，该屏蔽字中都有一位与之对应。对于某种信号，若其对应位已设置，则它当前是被阻塞的。进程可以调用 sigprocmask（在 10.12 节中说明）来检测和更改其当前信号屏蔽字。

信号数可能会超过一个整型数所包含的二进制位数，因此 POSIX.1 定义了一个新数据类型 sigset_t，它保持一个信号集。例如，信号屏蔽字就保存在这些信号集的一个中。10.11 节将说明对信号集进行操作的五个函数。

10.9 kill 和 raise 函数

kill 函数将信号发送给进程或进程组。raise 函数则允许进程向自身发送信号。

raise 是由 ANSI C 而非 POSIX.1 定义的。因为 ANSI C 并不涉及多进程，所以它不能定义如 kill 这样要有一个进程 ID 作为其参数的函数。

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

int kill(pid_t pid, int signo);
```

```
int raise(int signo);
```

两个函数返回：若成功则为0，若出错则为-1

kill的pid参数有四种不同的情况：

- $pid > 0$ 将信号发送给进程ID为pid的进程。
- $pid == 0$ 将信号发送给其进程组ID等于发送进程的进程组ID，而且发送进程有许可权向其发送信号的所有进程。

这里用的术语“所有进程”不包括实现定义的系统进程集。对于大多数 UNIX系统，系统进程集包括：交换进程(pid 0)，init(pid 1)以及页精灵进程(pid 2)。

- $pid < 0$ 将信号发送给其进程组ID等于pid绝对值，而且发送进程有许可权向其发送信号的所有进程。如上所述一样，“所有进程”并不包括系统进程集中的进程。
- $pid == -1$ POSIX.1未定义此种情况。

SVR4和4.3+BSD用此广播信号(broadcast signal)。在广播信号时，并不把信号发送给上述系统进程集。4.3+BSD也不将广播信号发送给发送进程自身。若调用者是超级用户，则将信号发送给所有进程。如果调用者不是超级用户，则将信号发送给其实际用户ID或保存的设置-用户-ID等于调用者的实际或有效用户ID的所有进程。广播信号只能用于管理方面(例如一个超级用户进程使该系统停止运行)。

上面曾提及，进程将信号发送给其他进程需要许可权。超级用户可将信号发送给另一个进程。对于非超级用户，其基本规则是发送者的实际或有效用户ID必须等于接收者的实际或有效用户ID。如果实现支持_POSIX_SAVED_IDS(如SVR4所做的那样)，则用保存的设置-用户-ID代替有效用户ID。

在对许可权进行测试时也有一个特例：如果被发送的信号是SIGCONT，则进程可将它发送给属于同一对话期的任一其他进程。

POSIX.1将信号编号0定义为空信号。如果signo参数是0，则kill仍执行正常的错误检查，但不发送信号。这常被用来确定一个特定进程是否仍旧存在。如果向一个并不存在的进程发送空信号，则kill返回-1，errno则被设置为ESRCH。但是，应当了解，UNIX系统在经过一定时间后会重新使用进程ID，所以一个现存的具有所给定进程ID的进程并不一定就是你所想要的进程。

如果调用kill为调用进程产生信号，而且此信号是不被阻塞的，那么在kill返回之前，signo或者某个其他未决的、非阻塞信号被传送到该进程。

10.10 alarm和pause函数

使用alarm函数可以设置一个时间值(闹钟时间)，在将来的某个时刻该时间值会被超过。当所设置的时间值被超过后，产生SIGALRM信号。如果不忽略或不捕捉此信号，则其默认动作是终止该进程。

```
#include <unistd.h>

unsigned int alarm(unsigned int seconds);
```

返回：0或以前设置的闹钟时间的余留秒数

其中，参数`seconds`的值是秒数，经过了指定的`seconds`秒后会产生信号SIGALRM。要了解的是，经过了指定秒后，信号由内核产生，由于进程调度的延迟，进程得到控制能够处理该信号还需一段时间。

早期的 UNIX 版本曾提出警告，这种信号可能比预定值提前 1 秒发送。POSIX.1 则不允许这样做。

每个进程只能有一个闹钟时间。如果在调用 `alarm` 时，以前已为该进程设置过闹钟时间，而且它还没有超时，则该闹钟时间的余留值作为本次 `alarm` 函数调用的值返回。以前登记的闹钟时间则被新值代换。

如果有以前登记的尚未超过的闹钟时间，而且 `seconds` 值是 0，则取消以前的闹钟时间，其余留值仍作为函数的返回值。

虽然 SIGALRM 的默认动作是终止进程，但是大多数使用闹钟的进程捕捉此信号。如果此时进程要终止，则在终止之前它可以执行所需的清除操作。

`pause` 函数使调用进程挂起直至捕捉到一个信号。

```
#include <unistd.h>

int pause(void);
```

返回：-1，`errno` 设置为 EINTR

只有执行了一个信号处理程序并从其返回时，`pause` 才返回。在这种情况下，`pause` 返回 -1，`errno` 设置为 EINTR。

实例

使用 `alarm` 和 `pause`，进程可使自己睡眠一段指定的时间。程序 10-4 中的 `sleep1` 函数提供这种功能。

程序 10-4 `sleep` 简化但并不完整的实现

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

static void
sig_alm(int signo)
{
    return; /* nothing to do, just return to wake up the pause */
}

unsigned int
sleep1(unsigned int nsecs)
{
    if (signal(SIGALRM, sig_alm) == SIG_ERR)
        return(nsecs);
    alarm(nsecs); /* start the timer */
    pause(); /* next caught signal wakes us up */
    return( alarm(0) ); /* turn off timer, return unslept time */
}
```

程序中的 `sleep1` 函数看起来与将在 10.19 节中说明的 `sleep` 函数类似，但这种简化实现有下列问题：

(1) 如果调用者已设置了闹钟，则它被sleep1函数中的第一次alarm调用擦去。

可用下列方法更正这一点：检查第一次调用alarm的返回值，如其小于本次调用alarm的参数值，则只应等到该前次设置的闹钟时间超时。如果前次设置闹钟时间的超时时刻后于本次设置值，则在sleep1函数返回之前，再次设置闹钟时间，使其在预定时间再发生超时。

(2) 该程序中修改了对SIGALRM的配置。如果编写了一个函数供其他函数调用，则在该函数被调用时先要保存原配置，在该函数返回后再恢复原配置。

更改这一点的的方法是：保存signal函数的返回值，在返回前恢复设置原配置。

(3) 在调用alarm和pause之间有一个竞态条件。在一个繁忙的系统中，可能alarm在调用pause之前超时，并调用了信号处理程序。如果发生了这种情况，则在调用pause后，如果没有捕捉到其他信号，则调用者将永远被挂起。

sleep早期的实现与程序10-4类似，但更正了问题(1)和(2)。有两种方法可以更正问题(3)。第一种方法是使用setjmp，下面立即说明这种方法。另一种方法是使用sigprocmask和sigsuspend，10.19节将说明这种方法。

实例

SVR2中的sleep实现使用了setjmp和longjmp(见7.10节)以避免问题(3)中所说明的竞态条件。此函数的一个简化版本，称为sleep2，示于程序10-5中(为了缩短实例长度，程序中没有处理上面所说的问题(1)和(2))。

程序10-5 sleep另一个不完善的实现

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

static jmp_buf env_alm;

static void
sig_alm(int signo)
{
    longjmp(env_alm, 1);
}

unsigned int
sleep2(unsigned int nsecs)
{
    if (signal(SIGALRM, sig_alm) == SIG_ERR)
        return(nsecs);
    if (setjmp(env_alm) == 0) {
        alarm(nsecs);          /* start the timer */
        pause();              /* next caught signal wakes us up */
    }
    return( alarm(0) );      /* turn off timer, return unslept time */
}
```

在此函数中，程序10-4具有的竞态条件已被避免。即使pause从未执行，在发生SIGALRM时，sleep2函数也返回。

但是，sleep2函数中却有另一个难于察觉的问题，它涉及到与其他信号的相互作用。如果SIGALRM中断了某个其他信号处理程序，则调用longjmp会提早终止该信号处理程序。程序10-6显示了这种情况。SIGINT处理程序中的for循环语句的执行时间在作者所用的系统上超过5秒钟，也就是大于sleep2的参数值，这正是我们想要的。整型变量j说明为volatile，这样就阻止

了优化编译程序除去循环语句。执行程序 10-6 得到：

```
$ a.out
^?          键入中断字符
sig_int starting
sleep2 returned: 0
```

程序10-6 在一个捕捉其他信号的程序中调用 sleep2

```
#include <signal.h>
#include "ourhdr.h"

unsigned int sleep2(unsigned int);
static void sig_int(int);

int
main(void)
{
    unsigned int    unslept;

    if (signal(SIGINT, sig_int) == SIG_ERR)
        err_sys("signal(SIGINT) error");

    unslept = sleep2(5);
    printf("sleep2 returned: %u\n", unslept);

    exit(0);
}

static void
sig_int(int signo)
{
    int            i;
    volatile int   j;

    printf("\nsig_int starting\n");
    for (i = 0; i < 2000000; i++)
        j += i * i;
    printf("sig_int finished\n");
    return;
}
```

从中可见 sleep2 函数所引起的 longjmp 使另一个信号处理程序 sig_int 提早终止。如果将 SVR2 的 sleep 函数与其他信号处理程序一起使用，就可能碰到这种情况。见习题 10.3。

sleep1 和 sleep2 函数这两个实例的目的是告诉我们在涉及到信号时需要有精细而周到的考虑。下面几节将说明解决这些问题的方法，使我们能够可靠地，在不影响其他代码段的情况下处理信号。

实例

除了用来实现 sleep 函数外，alarm 还常用于对可能阻塞的操作设置一个时间上限值。例如，程序中有一个读低速设备的会阻塞的操作（见 10.5 节），我们希望它超过一定时间量后就一定终止。程序 10-7 实现了这一点，它从标准输入读一行，然后将其写到标准输出上。

程序10-7 带时间限制调用 read

```
#include <signal.h>
#include "ourhdr.h"

static void sig_alrm(int);
```

```
int
main(void)
{
    int    n;
    char   line[MAXLINE];

    if (signal(SIGALRM, sig_alm) == SIG_ERR)
        err_sys("signal(SIGALRM) error");
    alarm(10);
    if ( (n = read(STDIN_FILENO, line, MAXLINE)) < 0)
        err_sys("read error");
    alarm(0);

    write(STDOUT_FILENO, line, n);

    exit(0);
}

static void
sig_alm(int signo)
{
    return; /* nothing to do, just return to interrupt the read */
}
```

这种代码序列在很多UNIX应用程序中都能见到，但是这种程序有两个问题：

(1) 程序10-7有与程序10-4同样的问题：在第一次 alarm调用和read调用之间有一个竞态条件。如果内核在这两个函数调用之间使进程不能占用处理机运行，而其时间长度又超过闹钟时间，则read可能永远阻塞。大多数这种类型的操作使用较长的闹钟时间，例如 1分钟或更长一点，使这种问题不会发生，但无论如何这是一个竞态条件。

(2) 如果系统调用是自动再起动的，则当从 SIGALRM信号处理程序返回时，read并不被终止。在这种情形下，设置时间限制不会起作用。

在这里我们确实需要终止慢速系统调用。但是，POSIX.1并未提供一种可移植的方法实现这一点。

实例

让我们用longjmp再实现前面的实例(见程序10-8)。使用这种方法则无需担心一个慢速的系统调用是否被中断。

程序10-8 使用longjmp，带时间限制调用read

```
#include <setjmp.h>
#include <signal.h>
#include "ourhdr.h"

static void sig_alm(int);
static jmp_buf env_alm;

int
main(void)
{
    int    n;
    char   line[MAXLINE];

    if (signal(SIGALRM, sig_alm) == SIG_ERR)
        err_sys("signal(SIGALRM) error");

    if (setjmp(env_alm) != 0)
        err_quit("read timeout");
```

```

alarm(10);
if ( (n = read(STDIN_FILENO, line, MAXLINE)) < 0)
    err_sys("read error");
alarm(0);

write(STDOUT_FILENO, line, n);

exit(0);
}

static void
sig_alm(int signo)
{
    longjmp(env_alm, 1);
}

```

不管系统是否重新启动系统调用，该程序都会如所预期的那样工作。但是要知道，该程序仍旧有与程序10-5相同的与其他信号处理程序相互作用的问题。

如果要对I/O操作设置时间限制，则如上所示可以使用 `longjmp`，当然也要清楚它可能有与其他信号处理程序相互作用的问题。另一种选择是使用 `select`或`poll`函数，12.5.1节和12.5.2节将对它们进行说明。

10.11 信号集

我们需要有一个能表示多个信号——信号集（signal set）的数据类型。将在 `sigprocmask`（下一节中说明）这样的函数中使用这种数据类型，以告诉内核不允许发生该信号集中的信号。如前所述，信号种类数目可能超过一个整型量所包含的位数，所以一般而言，不能用整型量中的一位代表一种信号。POSIX.1定义数据类型 `sigset_t`以包含一个信号集，并且定义了下列五个处理信号集的函数。

```

#include <signal.h>

int sigemptyset(sigset_t *set);

int sigfillset(sigset_t *set);

int sigaddset(sigset_t *set, int signo);

int sigdelset(sigset_t *set, int signo);

```

四个函数返回：若成功则为0，若出错则为-1

```

int sigismember(const sigset_t *set, int signo);

```

返回：若真则为1，若假则为0

函数 `sigemptyset`初始化由 `set`指向的信号集，使排除其中所有信号。函数 `sigfillset`初始化由 `set`指向的信号集，使其包括所有信号。所有应用程序在使用信号集前，要对该信号集调用 `sigemptyset`或`sigfillset`一次。这是因为C编译程序将不赋初值的外部 and 静态度量都初始化为0，而这是否与给定系统上信号集的实现相对应并不清楚。

一旦已经初始化了一个信号集，以后就可在该信号集中增、删特定的信号。函数 `sigaddset`将一个信号添加到现存集中，`sigdelset`则从信号集中删除一个信号。对所有以信号集作为参数

的函数，都向其传送信号集地址。

实例

如果实现的信号数目少于一个整型量所包含的位数，则可用一位代表一个信号的方法实现信号集。例如，大多数4.3+BSD实现中有31种信号和32位整型。sigemptyset和sigfillset这两个函数可以在<signal.h>头文件中实现为宏：

```
#define sigemptyset(ptr) ( *(ptr) = 0 )
#define sigfillset(ptr) ( *(ptr) = ~(sigset_t)0, 0 )
```

注意，除了设置对应信号集中各信号的位外，sigfillset必须返回0，所以使用逗号算符，它将逗号算符后的值作为表达式的值返回。

使用这种实现，sigaddset打开一位，sigdelset则关闭一位，sigismember测试一指定位。因为没有信号编号值为0，所以从信号编号中减1以得到要处理的位的位编号数。程序10-9可实现这些功能。

程序10-9 sigaddset、sigdelset和sigismember的实现

```
#include <signal.h>
#include <errno.h>

#define SIGBAD(signo) ((signo) <= 0 || (signo) >= NSIG)
/* <signal.h> usually defines NSIG to include signal number 0 */

int
sigaddset(sigset_t *set, int signo)
{
    if (SIGBAD(signo)) { errno = EINVAL; return(-1); }
    *set |= 1 << (signo - 1); /* turn bit on */
    return(0);
}

int
sigdelset(sigset_t *set, int signo)
{
    if (SIGBAD(signo)) { errno = EINVAL; return(-1); }
    *set &= ~(1 << (signo - 1)); /* turn bit off */
    return(0);
}

int
sigismember(const sigset_t *set, int signo)
{
    if (SIGBAD(signo)) { errno = EINVAL; return(-1); }
    return( (*set & (1 << (signo - 1))) != 0 );
}
```

也可将这三个函数在<signal.h>中实现为各一行的宏，但是POSIX.1要求检查信号编号参数的有效性，如果无效则设置errno。在宏中实现这一点比函数要难。

10.12 sigprocmask函数

10.8节曾说明一个进程的信号屏蔽字规定了当前阻塞而不能递送给该进程的信号集。调用函数sigprocmask可以检测或更改(或两者)进程的信号屏蔽字。

```
# include <signal.h>

int sigprocmask(int how, const sigset_t *set, sigset_t *oset);
```

返回：若成功则为0，若出错则为-1

首先，*oset*是非空指针，进程的当前信号屏蔽字通过 *oset*返回。其次，若 *set*是一个非空指针，则参数*how*指示如何修改当前信号屏蔽字。表 10-4说明了*how*可选用的值。SIG_BLOCK是或操作，而SIG_SETMASK则是赋值操作。

表10-4 用sigprocmask更改当前信号屏蔽字的方法

<i>how</i>	说 明
SIG_BLOCK	该进程新的信号屏蔽字是其当前信号屏蔽字和 <i>set</i> 指向信号集的并集。 <i>set</i> 包含了我们希望阻塞的附加信号
SIG_UNBLOCK	该进程新的信号屏蔽字是其当前信号屏蔽字和 <i>set</i> 所指向信号集的交集。 <i>set</i> 包含了我们希望解除阻塞的信号
SIG_SETMASK	该进程新的信号屏蔽是 <i>set</i> 指向的值

如果*set*是个空指针，则不改变该进程的信号屏蔽字，*how*的值也无意义。

如果在调用sigprocmask后有任何未决的、不再阻塞的信号，则在 sigprocmask返回前，至少将其中之一递送给该进程。

实例

程序10-10是一个函数，它打印调用进程的信号屏蔽字所阻塞信号的名称。从程序 10-14和 10-15中调用此函数。为了节省空间，没有对表 10-1中列出的每一种信号测试该屏蔽字（见习题10.9）。

程序10-10 为进程打印信号屏蔽字

```
#include <errno.h>
#include <signal.h>
#include "ourhdr.h"

void
pr_mask(const char *str)
{
    sigset_t    sigset;
    int         errno_save;

    errno_save = errno;    /* we can be called by signal handlers */
    if (sigprocmask(0, NULL, &sigset) < 0)
        err_sys("sigprocmask error");

    printf("%s", str);
    if (sigismember(&sigset, SIGINT))    printf("SIGINT ");
    if (sigismember(&sigset, SIGQUIT))   printf("SIGQUIT ");
    if (sigismember(&sigset, SIGUSR1))   printf("SIGUSR1 ");
    if (sigismember(&sigset, SIGALRM))   printf("SIGALRM ");
    /* remaining signals can go here */
    printf("\n");
    errno = errno_save;
}
```

10.13 sigpending函数

sigpending返回对于调用进程被阻塞不能递送和当前未决的信号集。该信号集通过 *set* 参数返回。

```
#include <signal.h>

int sigpending(sigset_t*);
```

返回：若成功则为0，若出错则为-1

实例

程序10-11使用了很多前面说明过的信号功能。进程阻塞了 SIGQUIT信号，保存了当前信号屏蔽字（以便以后恢复），然后睡眠5秒钟。在此期间所产生的退出信号都被阻塞，不递送至该进程，直到该信号不再被阻塞。在5秒睡眠结束后，检查是否有信号未决，然后将 SIGQUIT 设置为不再阻塞。

注意，在设置SIGQUIT为阻塞时，我们保存了老的屏蔽字。为了解除对该信号的阻塞，用老的屏蔽字重新设置了进程信号屏蔽字（SIG_SETMASK）。另一种方法是用SIG_UNBLOCK使阻塞的信号不再阻塞。但是，应当了解如果编写一个可能由其他人使用的函数，而且需要在函数中阻塞一个信号，则不能用SIG_UNBLOCK解除对此信号的阻塞，这是因为此函数的调用者在调用本函数之前可能也阻塞了此信号。在这种情况下必须使用SIG_SETMASK将信号屏蔽字恢复为原先值。10.18节的system函数部分有这样的一个例子。

在睡眠期间如果产生了退出信号，那么此时该信号是未决的，但是不再受阻塞，所以在sigprocmask返回之前，它被递送到本进程。从程序的输出中可以看到这一点：SIGQUIT处理程序（sig_quit）中的printf语句先执行，然后再执行sigprocmask之后的printf语句。

程序10-11 信号设置和sigprocmask实例

```
#include <signal.h>
#include "ourhdr.h"

static void sig_quit(int);

int
main(void)
{
    sigset_t    newmask, oldmask, pendmask;

    if (signal(SIGQUIT, sig_quit) == SIG_ERR)
        err_sys("can't catch SIGQUIT");

    sigemptyset(&newmask);
    sigaddset(&newmask, SIGQUIT);
    /* block SIGQUIT and save current signal mask */
    if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
        err_sys("SIG_BLOCK error");

    sleep(5);      /* SIGQUIT here will remain pending */

    if (sigpending(&pendmask) < 0)
        err_sys("sigpending error");
    if (sigismember(&pendmask, SIGQUIT))
```

```

printf("\nSIGQUIT pending\n");

        /* reset signal mask which unblocks SIGQUIT */
if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
    err_sys("SIG_SETMASK error");
printf("SIGQUIT unblocked\n");

sleep(5);      /* SIGQUIT here will terminate with core file */

exit(0);
}

static void
sig_quit(int signo)
{
    printf("caught SIGQUIT\n");

    if (signal(SIGQUIT, SIG_DFL) == SIG_ERR)
        err_sys("can't reset SIGQUIT");
    return;
}

```

然后该进程再睡眠5秒钟。如果在此期间再产生退出信号，那么它就会使该进程终止，因为在上次捕捉到该信号时，已将其处理方式设置为默认动作。此时如果键入终端退出字符 Ctrl-\，则输出“QUIT(coredump)”信息，表示进程因接到SIGQUIT而终止，但是在core文件中保存了与进程有关的信息(该信息是由shell发现其子进程异常终止时打印的)。

```

$ a.out
^\                产生信号一次(在5秒之内)
SIGQUIT pending  从sleep返回后
caught SIGQUIT   在信号处理程序中
SIGQUIT unblocked 从sigprocmask返回后
^\Quit(coredump) 再次产生信号
$ a.out
\                产生信号10次(在5秒之内)
SIGQUIT pending
caught SIGQUIT   只产生信号一次
SIGQUIT unblocked
^\Quit(coredump) 再产生信号

```

注意，在第二次运行该程序时，在进程睡眠期间使SIGQUIT信号产生了10次，但是解除了对该信号的阻塞后，只向进程传送一次SIGQUIT。从中可以看出在此系统上没有将信号进行排队。

10.14 sigaction函数

sigaction函数的功能是检查或修改(或两者)与指定信号相关联的处理动作。此函数取代了UNIX早期版本使用的signal函数。在本书末尾用sigaction函数实现了signal。

```

#include <signal.h>

int sigaction(int signo, const struct sigaction *act,
              struct sigaction *oact);

```

返回：若成功则为0，若出错则为-1

其中，参数 *signo* 是要检测或修改具体动作的信号的编号数。若 *act* 指针非空，则要修改其动作。如果 *oact* 指针非空，则系统返回该信号的原先动作。此函数使用下列结构：

```
struct sigaction {
    void          (*sa_handler)(); /* addr of signal handler,
                                   or SIG_IGN, or SIG_DFL */
    sigset_t sa_mask; /* additional signals to block */
    int          sa_flags; /* signal options, Table 10-5 */
};
```

当更改信号动作时，如果 *sa_handler* 指向一个信号捕捉函数（不是常数 *SIG_IGN* 或 *SIG_DFL*），则 *sa_mask* 字段说明了一个信号集，在调用信号捕捉函数之前，该信号集要加到进程的信号屏蔽字中。仅当从信号捕捉函数返回时再将进程的信号屏蔽字恢复为原先值。这样，在调用信号处理程序时就能阻塞某些信号。在信号处理程序被调用时，系统建立的新信号屏蔽字会自动包括正被递送的信号。因此保证了在处理一个给定的信号时，如果这种信号再次发生，那么它会被阻塞到对前一个信号的处理结束为止。回忆 10.8 节，若同一种信号多次发生，通常并不将它们排队，所以如果在某种信号被阻塞时，它发生了五次，那么对这种信号解除阻塞后，其信号处理函数通常只会被调用一次。

一旦对给定的信号设置了一个动作，那么在用 *sigaction* 改变它之前，该设置就一直有效。这与早期的不可靠信号机制不同，而符合了 POSIX.1 在这方面的要求。

act 结构的 *sa_flags* 字段包含了对信号进行处理的各个选择项。表 10-5 详细列出了这些可选项的意义。

表10-5 信号处理的选择项标志 (*sa_flags*)

可选项	POSIX.1	SVR4	4.3+BSD	说明
SA_NOCLDSTOP	•	•	•	若 <i>signo</i> 是 <i>SIGCHLD</i> ，当一子进程停止时（作业控制），不产生此信号。当一子进程终止时，仍旧产生此信号（但请参阅下面说明的 <i>SVR4 SA_NOCLDWAIT</i> 可选项）
SA_RESTART		•	•	由此信号中断的系统调用自动再起动（参见 10.5 节）
SA_ONSTACK		•	•	若用 <i>sigaltstack(2)</i> 已说明了一替换栈，则此信号递送给替换栈上的进程
SA_NOCLDWAIT		•		若 <i>signo</i> 是 <i>SIGCHLD</i> ，则当调用进程的子进程终止时，不创建僵死进程。若调用进程在后面调用 <i>wait</i> ，则阻塞到它所有子进程都终止，此时返回 -1， <i>errno</i> 设置为 <i>ECHILD</i> （见 10.7 节）
SA_NODEFER		•		当捕捉到此信号时，在执行其信号捕捉函数时，系统不自动阻塞此信号。注意，此种类型的操作对应于早期的不可靠信号
SA_RESETHAND		•		对此信号的处理方式在此信号捕捉函数的入口处重置为 <i>SIG_DFL</i> 。注意，此种类型的信号对应于早期的不可靠信号
SA_SIGINFO		•		此选项对信号处理程序提供了附加信息。详细情况见 10.21 节

实例——signal函数

现在用sigaction实现signal函数。4.3+BSD也是这样做的(POSIX.1的原理阐述部分也说明这是POSIX所希望的)。SVR4则提供老的不可靠信号语义的signal函数。除非为了向后兼容而使用老的语义,在SVR4之下,也应使用下面的signal实现,或者直接调用sigaction(在SVR4下,可以在调用sigaction时指定SA_RESETHAND和SA_NODEFER选择项以实现老的语义的signal函数)。本书中所有调用signal的实例均调用程序10-12中所实现的该函数。

程序10-12 用sigaction所实现的signal函数

```

/* Reliable version of signal(), using POSIX sigaction(). */
#include <signal.h>
#include "ourhdr.h"

Sigfunc *
signal(int signo, Sigfunc *func)
{
    struct sigaction act, oact;

    act.sa_handler = func;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;
    if (signo == SIGALRM) {
#ifdef SA_INTERRUPT
        act.sa_flags |= SA_INTERRUPT; /* SunOS */
#endif
    } else {
#ifdef SA_RESTART
        act.sa_flags |= SA_RESTART; /* SVR4, 4.3+BSD */
#endif
    }
    if (sigaction(signo, &act, &oact) < 0)
        return(SIG_ERR);
    return(oact.sa_handler);
}

```

注意,必须用sigemptyset函数初始化act结构的成员。不能保证:

```
act.sa_mask = 0;
```

会做同样的事情。

我们对除SIGALRM以外的所有信号都企图设置SA_RESTART标志,于是被这些信号中断的系统调用都能再起。不希望再起由SIGALRM信号中断的系统调用的原因是:我们对I/O操作可以设置时间限制。(请回忆与程序10-7有关的讨论。)

某些系统(如SunOS)定义了SA_INTERRUPT标志。这些系统的默认方式是重新起动被中断的系统调用,而指定此标志则使系统调用被中断后不再重起动。

实例——signal_intr函数

程序10-13是signal函数的另一种版本,它阻止被中断的系统调用再起。

程序10-13 signal_intr函数

```

#include <signal.h>
#include "ourhdr.h"

Sigfunc *
signal_intr(int signo, Sigfunc *func)

```

```

{
    struct sigaction    act, oact;

    act.sa_handler = func;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;
#ifdef SA_INTERRUPT    /* SunOS */
    act.sa_flags |= SA_INTERRUPT;
#endif
    if (sigaction(signo, &act, &oact) < 0)
        return(SIG_ERR);
    return(oact.sa_handler);
}

```

如果系统定义了SA_INTERRUPT，则在sa_flags中增加该标志，这样也就阻止了被中断的系统调用的再起动。

10.15 sigsetjmp和siglongjmp函数

7.10节说明了用于非局部转移的 setjmp和longjmp函数。在信号处理程序中经常调用 longjmp函数以返回到程序的主循环中，而不是从该处理程序返回。确实，ANSI C标准说明一个信号处理程序可以返回或者调用 abort、exit或longjmp。程序10-5和10-8中已经有了这种情况。

调用longjmp时有一个问题。当捕捉到一个信号时，进入信号捕捉函数，此时当前信号被自动地加到进程的信号屏蔽字中。这阻止了后来产生的这种信号中断此信号处理程序。如果用 longjmp跳出此信号处理程序，则对此进程的信号屏蔽字会发生什么呢？

在4.3+BSD下，setjmp和longjmp保存和恢复信号屏蔽字。但是，SVR4并不做这种操作。4.3+BSD提供函数_setjmp和_longjmp，它们也不保存和恢复信号屏蔽字。

为了允许两种形式并存，POSIX.1并没有说明setjmp和longjmp对信号屏蔽字的作用，而是定义了两个新函数 sigsetjmp和siglongjmp。在信号处理程序中作非局部转移时应当使用这两个函数。

```
#include <setjmp.h>
```

```
int sigsetjmp(sigjmp_buf env, int savemask);
```

返回：若直接调用则为0，若从siglongjmp调用返回则为非0

```
void siglongjmp(sigjmp_buf env, int val);
```

这两个函数和setjmp，longjmp之间的唯一区别是sigsetjmp增加了一个参数。如果savemask非0，则sigsetjmp在env中保存进程的当前信号屏蔽字。调用 siglongjmp时，如果带非0 savemask的 sigsetjmp调用已经保存了env，则siglongjmp从其中恢复保存的信号屏蔽字。

实例

程序10-14显示了在信号处理程序被调用时，系统所设置的信号屏蔽字如何自动地包括刚被捕捉到的信号。它也例示了如何使用 sigsetjmp和siglongjmp函数。

程序10-14 信号屏蔽、sigsetjmp和siglongjmp实例

```
#include <signal.h>
#include <setjmp.h>
#include <time.h>
#include "ourhdr.h"

static void          sig_usr1(int), sig_alm(int);
static sigjmp_buf    jmpbuf;
static volatile sig_atomic_t  canjump;

int
main(void)
{
    if (signal(SIGUSR1, sig_usr1) == SIG_ERR)
        err_sys("signal(SIGUSR1) error");
    if (signal(SIGALRM, sig_alm) == SIG_ERR)
        err_sys("signal(SIGALRM) error");
    pr_mask("starting main: ");    /* Program 10.10 */

    if (sigsetjmp(jmpbuf, 1)) {
        pr_mask("ending main: ");
        exit(0);
    }
    canjump = 1;    /* now sigsetjmp() is OK */
    for ( ; ; )
        pause();
}

static void
sig_usr1(int signo)
{
    time_t  starttime;
    if (canjump == 0)
        return;    /* unexpected signal, ignore */
    pr_mask("starting sig_usr1: ");
    alarm(3);    /* SIGALRM in 3 seconds */
    starttime = time(NULL);
    for ( ; ; )    /* busy wait for 5 seconds */
        if (time(NULL) > starttime + 5)
            break;

    pr_mask("finishing sig_usr1: ");
    canjump = 0;
    siglongjmp(jmpbuf, 1);    /* jump back to main, don't return */
}

static void
sig_alm(int signo)
{
    pr_mask("in sig_alm: ");
    return;
}
```

此程序例示了另一种技术，只要在信号处理程序中调用 siglongjmp 就应使用这种技术。在调用 sigsetjmp 之后将变量 canjump 设置为非 0。在信号处理程序中检测此变量，仅当它为非 0 值时才调用 siglongjmp。这提供了一种保护机制，使得若在 jmpbuf(跳转缓存)尚未由 sigsetjmp 初始化时调用信号处理程序，则不执行其处理动作就返回（在本程序中，siglongjmp 之后程序很快就结束，但是在较大的程序中，在 siglongjmp 之后，信号处理程序可能仍旧被设置）。在一般的

C代码中（不是信号处理程序），对于longjmp并不需要这种保护措施。但是，因为信号可能在任何时候发生，所以在信号处理程序中，需要这种保护措施。

在程序中使用了数据类型 sig_atomic_t，它是由ANSI C定义的在写时不会被中断的变量类型。它意味着这种变量在具有虚存的系统上不会跨越页的边界，可以用一条机器指令对其进行存取。对于这种类型的变量总是包括ANSI类型修饰符volatile，其原因是：该变量将由两个不同的控制线——main函数和异步执行的信号处理程序存取。

图10-1显示了此程序的执行时间顺序。可将图10-1分成三部分：左面部分（对应于main），中间部分（sig_usr1）和右面部分（sig_alm）。在进程执行左面部分时，信号屏蔽字是0（没有信号是阻塞的）。而执行中间部分时，其信号屏蔽字是SIGUSR1。执行右面部分时，信号屏蔽字是SIGUSR1 | SIGALRM。

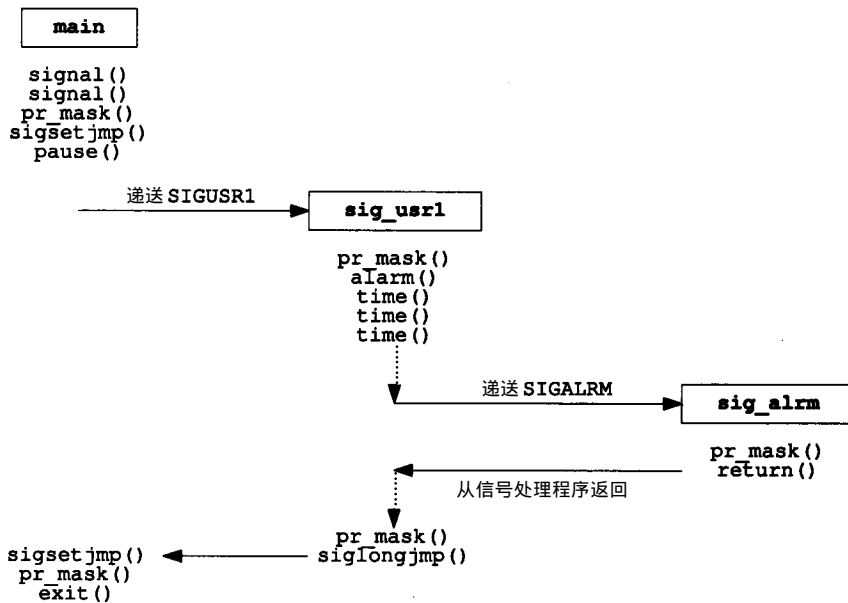


图10-1 处理两个信号的实例程序的时间顺序

执行程序10-14得到下面的输出：

```

$ a.out &                在后台启动进程
starting main:
[1] 531                  作业控制 shell打印其进程ID
$ kill -USR1 531         向该进程发送 SIGUSR1
starting sig_usr1: SIGUSR1
$ in sig_alm: SIGUSR1 SIGALRM
finishing sig_usr1: SIGUSR1
ending main:
                        键入回车
[1] + Done              a.out &
  
```

该输出与我们所期望的相同：当调用一个信号处理程序时，被捕捉到的信号加到进程的当前信号屏蔽字。当从信号处理程序返回时，原来的屏蔽字被恢复。另外，siglongjmp恢复了由sigsetjmp所保存的信号屏蔽字。

如果将程序 10-14 中的 `sigsetjmp` 和 `siglongjmp` 分别代换成 `_setjmp` 和 `_longjmp`，则在 4.3+BSD 下运行此程序，最后一行输出变成：

```
ending main: SIGUSR1
```

这意味着在调用 `_setjmp` 之后执行 `main` 函数时，其 `SIGUSR1` 是阻塞的。这多半不是我们所希望的。

10.16 sigsuspend 函数

上面已经说明，更改进程的信号屏蔽字可以阻塞或解除阻塞所选择的信号。使用这种技术可以保护不希望由信号中断的代码临界区。如果希望对一个信号解除阻塞，然后 `pause` 以等待以前被阻塞的信号发生，则又将如何呢？假定信号是 `SIGINT`，实现这一点的一种不正确的方法是：

```
sigset_t    newmask, oldmask;

sigemptyset(&newmask);
sigaddset(&newmask, SIGINT);
/* block SIGINT and save current signal mask */
if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
    err_sys("SIG_BLOCK error");

/* critical region of code */

/* reset signal mask, which unblocks SIGINT */
if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
    err_sys("SIG_SETMASK error");

pause();      /* wait for signal to occur */

/* continue processing */
```

如果在解除对 `SIGINT` 的阻塞和 `pause` 之间发生了 `SIGINT` 信号，则此信号被丢失。这是早期的不可靠信号机制的另一个问题。

为了纠正此问题，需要在一个原子操作中实现恢复信号屏蔽字，然后使进程睡眠，这种功能是由 `sigsuspend` 函数所提供的。

```
#include <signal.h>

int sigsuspend(const sigset_t *sigmask);
```

返回：-1，`errno` 设置为 `EINTR`

进程的信号屏蔽字设置为由 `sigmask` 指向的值。在捕捉到一个信号或发生了一个会终止该进程的信号之前，该进程也被挂起。如果捕捉到一个信号而且从该信号处理程序返回，则 `sigsuspend` 返回，并且该进程的信号屏蔽字设置为调用 `sigsuspend` 之前的值。

注意，此函数没有成功返回值。如果它返回到调用者，则总是返回 - 1，并且 `errno` 设置为 `EINTR` (表示一个被中断的系统调用)。

实例

程序 10-15 显示了保护临界区，使其不被指定的信号中断的正确方法。

程序 10-15 保护临界区不被信号中断

```
#include <signal.h>
```

```

#include    "ourhdr.h"

static void sig_int(int);

int
main(void)
{
    sigset_t    newmask, oldmask, zeromask;

    if (signal(SIGINT, sig_int) == SIG_ERR)
        err_sys("signal(SIGINT) error");

    sigemptyset(&zeromask);

    sigemptyset(&newmask);
    sigaddset(&newmask, SIGINT);
    /* block SIGINT and save current signal mask */
    if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
        err_sys("SIG_BLOCK error");

        /* critical region of code */
    pr_mask("in critical region: ");

        /* allow all signals and pause */
    if (sigsuspend(&zeromask) != -1)
        err_sys("sigsuspend error");
    pr_mask("after return from sigsuspend: ");

        /* reset signal mask which unblocks SIGINT */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        err_sys("SIG_SETMASK error");

        /* and continue processing ... */
    exit(0);
}

static void
sig_int(int signo)
{
    pr_mask("\nin sig_int: ");
    return;
}

```

注意，当 `sigsuspend` 返回时，它将信号屏蔽字设置为调用它之前的值。在本例中，`SIGINT` 信号将被阻塞。因此将信号屏蔽复置为原先保存的值 (`oldmask`)。

运行程序 10-15 得到下面的输出：

```

$ a.out
in critical region: SIGINT
^?                键入自定义的中断字符
in sig_int: SIGINT
after return from sigsuspend: SIGINT

```

从中可见，在 `sigsuspend` 返回时，它将信号屏蔽字恢复为调用它之前的值。

实例

`sigsuspend` 的另一种应用是等待一个信号处理程序设置一个全局变量。程序 10-16 用于捕捉中断信号和退出信号，但是希望只有在捕捉到退出信号时再继续执行 `main` 程序。此程序的样本输出是：

```

$ a.out
^?          键入我们的中断字符
interrupt
^?          再次键入我们的中断字符
interrupt
^?          再一次
interrupt
^\ $        用退出符终止

```

考虑到支持ANSI C的非POSIX系统与POSIX系统两者之间的可移植性，在一个信号处理程序中我们唯一应当做的是赋一个值给类型为 `sig_atomic_t` 的变量。POSIX.1规定得更多一些，它说明了在一个信号处理程序中可以安全地调用的函数表（见表10-3），但是如果这样来编写代码，则它可能不会正确地在非POSIX系统上运行。

程序10-16 用 `sigsuspend` 等待一个全局变量被设置

```

#include <signal.h>
#include "ourhdr.h"

volatile sig_atomic_t quitflag; /* set nonzero by signal handler */

int
main(void)
{
    void sig_int(int);
    sigset_t newmask, oldmask, zeromask;

    if (signal(SIGINT, sig_int) == SIG_ERR)
        err_sys("signal(SIGINT) error");
    if (signal(SIGQUIT, sig_int) == SIG_ERR)
        err_sys("signal(SIGQUIT) error");

    sigemptyset(&zeromask);

    sigemptyset(&newmask);
    sigaddset(&newmask, SIGQUIT);
    /* block SIGQUIT and save current signal mask */
    if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
        err_sys("SIG_BLOCK error");

    while (quitflag == 0)
        sigsuspend(&zeromask);

    /* SIGQUIT has been caught and is now blocked; do whatever */
    quitflag = 0;

    /* reset signal mask which unblocks SIGQUIT */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        err_sys("SIG_SETMASK error");

    exit(0);
}

void
sig_int(int signo) /* one signal handler for SIGINT and SIGQUIT */
{
    if (signo == SIGINT)

```

```

        printf("\ninterrupt\n");
    else if (signo == SIGQUIT)
        quitflag = 1; /* set flag for main loop */
    return;
}

```

实例

可以用信号实现父子进程之间的同步，这是信号应用的另一个实例。程序 10-17实现了 8.8节中提到的五个例程：TELL_WAIT、TELL_PARENT、TELL_CHILD、WAIT_PARENT和WAIT_CHILD。其中使用了两个用户定义的信号：SIGUSR1由父进程发送给子进程，SIGUSR2由子进程发送给父进程。程序 14-3说明了使用管道的这五个函数的另一种实现。

程序10-17 父子进程可用来实现同步的例程

```

#include <signal.h>
#include "ourhdr.h"

static volatile sig_atomic_t sigflag;
/* set nonzero by signal handler */
static sigset_t newmask, oldmask, zeromask;

static void
sig_usr(int signo) /* one signal handler for SIGUSR1 and SIGUSR2 */
{
    sigflag = 1;
    return;
}

void
TELL_WAIT(void)
{
    if (signal(SIGUSR1, sig_usr) == SIG_ERR)
        err_sys("signal(SIGUSR1) error");
    if (signal(SIGUSR2, sig_usr) == SIG_ERR)
        err_sys("signal(SIGUSR2) error");

    sigemptyset(&zeromask);

    sigemptyset(&newmask);
    sigaddset(&newmask, SIGUSR1);
    sigaddset(&newmask, SIGUSR2);
    /* block SIGUSR1 and SIGUSR2, and save current signal mask */
    if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
        err_sys("SIG_BLOCK error");
}

void
TELL_PARENT(pid_t pid)
{
    kill(pid, SIGUSR2); /* tell parent we're done */
}

void
WAIT_PARENT(void)
{
    while (sigflag == 0)
        sigsuspend(&zeromask); /* and wait for parent */

    sigflag = 0;
}

```



```

        /* reset signal mask to original value */
        if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
            err_sys("SIG_SETMASK error");
    }
    void
    TELL_CHILD(pid_t pid)
    {
        kill(pid, SIGUSR1);          /* tell child we're done */
    }

    void
    WAIT_CHILD(void)
    {
        while (sigflag == 0)
            sigsuspend(&zeromask); /* and wait for child */

        sigflag = 0;
        /* reset signal mask to original value */
        if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
            err_sys("SIG_SETMASK error");
    }
}

```

如果在等待信号发生时希望去睡眠，则 `sigsuspend` 函数可以满足此种要求（正如在前面两个例子中所示），但是如果在等待信号期间希望调用其他系统函数则将如何呢？不幸的是对此问题没有可靠的解决方法。

在程序 17-13 中我们遇到了这种情况。我们捕捉 `SIGINT` 和 `SIGALRM` 这两种信号，在信号发生时，这两个信号处理程序都设置一个全局变量。用 `signal-intr` 函数设置这两个信号处理程序，使得它们中断任一被阻塞的慢速系统调用。当阻塞在 `select` 函数调用（见 12.5.1 节），等待慢速设备的输入时很可能发生这两种信号（因为设置闹钟以阻止永远等待输入，所以这对 `SIGALRM` 是特别真实的）。我们能尽力做到的是：

```

if (intr_flag)          /* flag set by our SIGINT handler */
    handle_intr();
if (alrm_flag)         /* flag set by our SIGALRM handler */
    handle_alrm();
/* signals occurring in here are lost */
while (select( ... ) < 0) {
    if (errno == EINTR) {
        if (alrm_flag)
            handle_alrm();
        else if (intr_flag)
            handle_intr();
    } else
        /* some other error */
}

```

在调用 `select` 之前测试各全局标志，如果 `select` 返回一个中断的系统调用错误，则再次进行测试。如果在前两个 `if` 语句和后随的 `select` 调用之间捕捉到两个信号中的任意一个，则问题就发生了。正如代码中的注释所指出的，在此处发生的信号丢失了。调用相应的信号处理程序，它们设置了相应的全局变量，但是 `select` 决不会返回（除非某些数据已准备好可读）。

我们想要能够做的是下列步骤序列：

- (1) 阻塞 `SIGINT` 和 `SIGALRM`。
- (2) 测试两个全局变量以判别是否发生了一个信号，如果已发生则处理此条件。
- (3) 调用 `select`（或任何其他系统调用，例如 `read`）并解除对这两个信号的阻塞，这两个操作要作为一个原子操作。

sigsuspend函数仅当第(3)步是一个pause操作时才帮助我们。

10.17 abort函数

前面已提及abort函数的功能是使程序异常终止。

```
#include <stdlib.h>
void abort(void);
```

此函数不返回

此函数将SIGABRT信号发送给调用进程。进程不应忽略此信号。

ANSI C要求若捕捉到此信号而且相应信号处理程序返回，abort仍不会返回到其调用者。如果捕捉到此信号，则信号处理程序不能返回的唯一方法是它调用 exit、_exit、longjmp或siglongjmp。(10.15节讨论了longjmp和siglongjmp之间的区别。) POSIX.1也说明abort覆盖了进程对此信号的阻塞和忽略。

让进程捕捉SIGABRT的意图是：在进程终止之前由其执行所需的清除操作。如果进程并不在信号处理程序中终止自己，POSIX.1说明当信号处理程序返回时，abort终止该进程。

ANSI C对此函数的规格说明将这一问题留由实现决定，而不管输出流是否刷新以及不管临时文件(见5.13节)是否删除。POSIX.1的要求则进了一步，它要求如果abort调用终止进程，则它应该对所有打开的标准I/O流调用fclose的效果。但是如果abort调用并不终止进程，则它对打开流也不应有影响。正如我们将在后面所看到的，这种要求很难实现。

系统V早期的版本中，abort函数产生SIGIOT信号。更进一步，进程忽略此信号，或者捕捉它并从信号处理程序返回都是可能的，在返回情况下，abort返回到它的调用者。

4.3BSD产生SIGILL信号。在此之前，该函数解除对此信号的阻塞，将其配置恢复为SIG_DFL(终止并构造core文件)。这阻止一个进程忽略或捕捉此信号。

SVR4在产生此信号之前关闭所有I/O流。在另一方面，4.3+BSD则不做此操作。对于保护性的程序设计，如果希望刷新标准I/O流，则在调用abort之前要做这种操作。在err_dump函数中实现了这一点(见附录B)。

因为大多数UNIX tmpfile(临时文件)的实现在创建该文件之后立即调用unlink，所以ANSI C关于临时文件的警告通常与我们无关。

实例

程序10-18按POSIX.1的说明实现了abort函数。对处理打开的标准I/O流的要求是难于实现的。首先查看是否执行了默认动作，并刷新了所有标准I/O流。这并不等价于对所有打开的流调用fclose(因为只刷新，并不关闭它们)，但是当进程终止时，系统会关闭所有打开文件。如果进程捕捉此信号并返回，则刷新所有的流。(如果进程捕捉此信号，并且不返回，则不会触及标准I/O流。)没有处理的唯一条件是如果进程捕捉此信号，然后调用_exit。在这种情况下，任何未刷新的存储器中的标准I/O缓存都被丢弃。我们假定捕捉此信号，并特地调用_exit的调用者并不想要刷新缓存。

回忆10.9节，如果调用kill使其为调用者产生信号，并且如果该信号是不被阻塞的(程序10-18保证做到了这一点)，则在kill返回前该信号就被传送给该进程。这样就可确知如果对

kill的调用返回了，则该进程一定已捕捉到该信号，并且也从该信号处理程序返回。

程序10-18 abort的POSIX.1实现

```

#include <sys/signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void
abort(void)          /* POSIX-style abort() function */
{
    sigset_t          mask;
    struct sigaction  action;

    /* caller can't ignore SIGABRT, if so reset to default */
    sigaction(SIGABRT, NULL, &action);
    if (action.sa_handler == SIG_IGN) {
        action.sa_handler = SIG_DFL;
        sigaction(SIGABRT, &action, NULL);
    }

    if (action.sa_handler == SIG_DFL)
        fflush(NULL);          /* flush all open stdio streams */

    /* caller can't block SIGABRT; make sure it's unblocked */
    sigfillset(&mask);
    sigdelset(&mask, SIGABRT); /* mask has only SIGABRT turned off */
    sigprocmask(SIG_SETMASK, &mask, NULL);

    kill(getpid(), SIGABRT);   /* send the signal */

    /* if we're here, process caught SIGABRT and returned */
    fflush(NULL);              /* flush all open stdio streams */

    action.sa_handler = SIG_DFL;
    sigaction(SIGABRT, &action, NULL); /* reset disposition to default */
    sigprocmask(SIG_SETMASK, &mask, NULL); /* just in case ... */

    kill(getpid(), SIGABRT);   /* and one more time */

    exit(1); /* this should never be executed ... */
}

```

10.18 system函数

在8.12节已经有了一个system函数的实现，但是该版本并不做任何信号处理。POSIX.2要求system忽略SIGINT和SIGQUIT，阻塞SIGCHLD。在给出一个正确地处理这些信号的一个版本之前，先说明为什么要考虑信号处理。

实例

程序10-19使用8.12节中的system版本，用其调用ed(1)编辑程序。(ed很久以来就是UNIX的组成部分。在这里使用它的原因是：它是一个交互式的捕捉中断和退出信号的程序。若从shell调用ed，并键入中断字符，则它捕捉中断信号并打印问号。它也将对退出符的处理方式设置为忽略。)

程序10-19 用system调用ed编辑程序

```

#include <sys/types.h>
#include <signal.h>
#include "ourhdr.h"

static void sig_int(int), sig_chld(int);

int
main(void)
{
    int    status;

    if (signal(SIGINT, sig_int) == SIG_ERR)
        err_sys("signal(SIGINT) error");
    if (signal(SIGCHLD, sig_chld) == SIG_ERR)
        err_sys("signal(SIGCHLD) error");

    if ( (status = system("/bin/ed")) < 0)
        err_sys("system() error");
    exit(0);
}

static void
sig_int(int signo)
{
    printf("caught SIGINT\n");
    return;
}

static void
sig_chld(int signo)
{
    printf("caught SIGCHLD\n");
    return;
}

```

程序10-19用于捕捉SIGINT和SIGCHLD。若调用它则可得：

```

$ a.out
a                将正文添加至编辑器缓存
Here is one line of text
and another
.                行首的点停止添加方式
1, $p           打印第1行至最后1行，以便观察缓存中的内容
Here is one line of text
and another     将缓存写至一文件
w temp.foo      编辑器称写了37个字节
37              离开编辑器
q
caught SIGCHLD

```

当编辑程序终止时，对父进程(a.out进程)产生SIGCHLD信号。父进程捕捉它，执行其处理程序sig_chld，然后从其返回。但是若父进程正捕捉SIGCHLD信号，那么在system函数执行时，父进程中该信号的递送应当阻塞。实际上，这就是POSIX.2所说明的。否则，当system创建的子进程结束时，system的调用者可能错误地认为，它自己的一个子进程结束了。

如果再次执行该程序，在这次运行时将一个中断信号传送给编辑程序，则可得：

```

$ a.out

```

```

a                将正文添加至编辑器缓存
hello, world
.                行首的点停止添加方式
1,$p            打印第1行至最后1行
hello, world
w etmp.foo      将缓存写至一文件
13              编辑器称写了13个字节
^?              键入中断符
?                编辑器捕捉信号，打印问号
caught SIGINT   父进程执行同一操作
q                离开编辑器
caught SIGCHLD

```

回忆9.6节可知，键入中断字符可使中断信号传送给前台进程组中的所有进程。图 10-2显示了编辑程序正在进行时的进程安排。

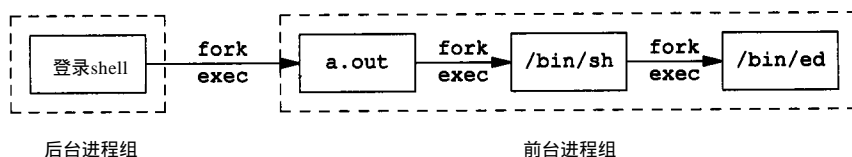


图10-2 程序10-19运行时的前、后台进程组

在这一实例中，SIGINT被送给三个前台进程。（shell进程忽略此信号。）从输出中可见，a.out进程和ed进程捕捉该信号。但是，当用system运行另一个程序（例如ed）时，不应使父、子进程两者都捕捉终端产生的两个信号：中断和退出。这两个信号只应送给正在运行的程序：子进程。因为由system执行的命令可能是交互作用命令（如本例中的ed），以及因为system的调用者在程序执行时放弃了控制，等待该执行程序的结束，所以system的调用者就不应接收这两个终端产生的信号。这就是为什么POSIX.2规定system的调用者应当忽略这两个信号的原因。

实例

程序10-20是system函数的另一个实现，它进行了所要求的信号处理。

程序10-20 system函数的POSIX.2实现

```

#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
#include <signal.h>
#include <unistd.h>

int
system(const char *cmdstring) /* with appropriate signal handling */
{
    pid_t      pid;
    int        status;
    struct sigaction ignore, saveintr, savequit;
    sigset_t   chldmask, savemask;

    if (cmdstring == NULL)
        return(1); /* always a command processor with Unix */

    ignore.sa_handler = SIG_IGN; /* ignore SIGINT and SIGQUIT */

```

```

sigemptyset(&ignore.sa_mask);
ignore.sa_flags = 0;
if (sigaction(SIGINT, &ignore, &saveintr) < 0)
    return(-1);
if (sigaction(SIGQUIT, &ignore, &savequit) < 0)
    return(-1);

sigemptyset(&chldmask);          /* now block SIGCHLD */
sigaddset(&chldmask, SIGCHLD);
if (sigprocmask(SIG_BLOCK, &chldmask, &savemask) < 0)
    return(-1);

if ( (pid = fork()) < 0) {
    status = -1;    /* probably out of processes */
} else if (pid == 0) {           /* child */
    /* restore previous signal actions & reset signal mask */
    sigaction(SIGINT, &saveintr, NULL);
    sigaction(SIGQUIT, &savequit, NULL);
    sigprocmask(SIG_SETMASK, &savemask, NULL);

    execl("/bin/sh", "sh", "-c", cmdstring, (char *) 0);
    _exit(127);    /* exec error */
} else {                        /* parent */
    while (waitpid(pid, &status, 0) < 0)
        if (errno != EINTR) {
            status = -1; /* error other than EINTR from waitpid() */
            break;
        }
}

/* restore previous signal actions & reset signal mask */
if (sigaction(SIGINT, &saveintr, NULL) < 0)
    return(-1);
if (sigaction(SIGQUIT, &savequit, NULL) < 0)
    return(-1);
if (sigprocmask(SIG_SETMASK, &savemask, NULL) < 0)
    return(-1);

return(status);
}

```

很多较早的文献中使用下列程序段忽略中断和退出信号：

```

if ( (pid = fork()) < 0)
    err_sys("fork error");
else if (pid == 0) {    /* child */
    execl(...);
    _exit(127);
}
/* parent */
old_intr = signal(SIGINT, SIG_IGN);
old_quit = signal(SIGQUIT, SIG_IGN);
waitpid(pid, &status, 0)
signal(SIGINT, old_intr);
signal(SIGQUIT, old_quit);

```

这段代码的问题是：在 fork 之后不能保证父进程还是子进程先运行。如果子进程先运行，父进程在一段时间后再运行，那么在父进程将中断信号的配置更改为忽略之前，就可能产生这种信号。由于这种原因，程序 10-20 在 fork 之前就改变对该信号的配置。

注意，子进程在调用`execl`之前要先恢复这两个信号的配置。这就允许在调用者配置的基础上，`execl`可将它们的配置更改为默认值。

system的返回值

`system`的返回值是shell的终止状态，它不总是执行命令字符串进程的终止状态。程序 8-13 中有一些例子，其结果正是我们所期望的。如果执行一条如 `date` 那样的简单命令，则其终止状态是0。执行shell命令`exit 44`，则得终止状态44。在信号方面又如何呢？

运行程序8-14，并向正在执行的命令发送一些信号。

```
$ tsys "sleep 30"
^?normal termination, exit status = 130    键入中断符
$ tsys "sleep 30"
^\\sh: 946 Quit                             键入退出符
normal termination, exit status = 131
```

当用中断信号终止 `sleep` 时，`pr_exit` 函数（见程序 8-3）认为它正常终止。当用退出键杀死 `sleep` 进程时，发生同样的事情。终止状态 130、131 又是怎样得到的呢？原来 Bourne shell 有一个在其文档中没有清楚说明的特性，其终止状态是 128 加上它所执行的命令由一个信号终止时的该信号编号数。用交互方式使用 shell 可以看到这一点。

```
$ sh                                     确保运行 Bourne shell
$ sh -c "sleep 30"
^?                                       键入中断符
$ echo $?                               打印最后一条命令的终止状态
130
$ sh -c "sleep 30"
^\\sh:962 Quit-core dumped                键入退出符
$ echo $?                               打印最后一条命令的终止状态
131
$ exit                                   离开 Bourne shell
```

在所使用的系统中，`SIGINT` 的值为 2，`SIGQUIT` 的值为 3，于是给出 shell 终止状态 130、131。

再说明几个类似的例子，这一次将一个信号直接送给 shell，然后观察 `system` 返回什么。

```
$ tsys "sleep 30" &                    这一次在后台启动它
[1]      980
$ ps                                     查看进程 ID
  PID TT STAT TIME COMMAND
  980 p3 S   0:00 tsys sleep 30
  981 p3 S   0:00 sh -c sleep 30
  982 p3 S   0:00 sleep 30
  985 p3 R   0:00 ps
$ kill -KILL 981                        杀死 shell 自身
abnormal termination, signal number = 9
[1] + Done      tsys "sleep 30" &
```

从中可见仅当 shell 本身异常终止时，`system` 的返回值才报告一个异常终止。

在编写使用 `system` 函数的程序时，一定要正确地解释返回值。如果直接调用 `fork`、`exec` 和 `wait`，则终止状态与调用 `system` 是不同的。

10.19 sleep函数

在本书的很多例子中都已使用了 `sleep` 函数，在程序 10-4 和 10-5 中有两个 `sleep` 的很完善的实现。

```
#include <unistd.h>

unsigned int sleep(unsigned int seconds);
```

返回：0或未睡的秒数

此函数使调用进程被挂起直到：

- (1) 已经过了 `seconds` 所指定的墙上时钟时间，或者
- (2) 该进程捕捉到一个信号并从信号处理程序返回。

如同 `alarm` 信号一样，由于某些系统活动，实际返回时间比所要求的会迟一些。

在第(1)种情形，返回值是 0。当由于捕捉到某个信号 `sleep` 提早返回时（第(2)种情形），返回值是未睡足的秒数（所要求的时间减实际睡眠时间）。

`sleep` 可以用 `alarm` 函数（见 10.10 节）实现，但这并不是必需的。如果使用 `alarm`，则这两个函数之间可以有交互作用。POSIX.1 标准对这些交互作用并未作任何说明。例如，若先调用 `alarm(10)`，过了 3 秒后又调用 `sleep(5)`，那么将如何呢？`sleep` 将在 5 秒后返回（假定在这段时间内没有捕捉到另一个信号），但是否在 2 秒后又产生另一个 `SIGALRM` 信号呢？这种细节依赖于实现。

SVR4 用 `alarm` 实现 `sleep`。`sleep(3)` 手册页中说明以前安排的闹钟仍被正常处理。例如，在前面的例子中，在 `sleep` 返回之前，它安排在 2 秒后再次到达闹钟时间。在这种情况下，`sleep` 返回 0。（很明显，`sleep` 必须保存 `SIGALRM` 信号处理程序的地址，在返回前重新设置它。）另外，如果先做一次 `alarm(6)`，3 秒钟之后又做一次 `sleep(5)`，则在 3 秒后 `sleep` 返回，而不是 5 秒钟。此时，`sleep` 的返回值则是未睡足的时间 2 秒。

4.3+BSD 则使用另一种技术：由 `setitimer(2)` 提供间隔计时器。该计时器独立于 `alarm` 函数，但在以前设置的间隔计时器和 `sleep` 之间仍能有相互作用。另外，尽管闹钟计时器 (`alarm`) 和间隔计时器 (`setitimer`) 是分开的，但是不幸它们使用同一 `SIGALRM` 信号。因为 `sleep` 暂时将该信号的处理程序改变为它自己的函数，所以在 `alarm` 和 `sleep` 之间仍可能有不希望的相互作用。

如果混合调用 `sleep` 和其他与时间有关的函数，它们之间有相互作用，则你应当清楚地了解你所使用的系统是如何实现 `sleep` 的。

以前伯克利类的 `sleep` 实现不提供任何有用的返回信息。这在 4.3+BSD 中已经解决。

实例

程序 10-21 是一个 POSIX.1 `sleep` 函数的实现。此函数是程序 10-4 的修改版，它可靠地处理信号，避免了早期实现中的竞态条件，但是仍未处理与以前设置的闹钟的相互作用（正如前面提到的，POSIX.1 并未对这些交互作用进行定义）。

程序10-21 sleep的可靠实现

```
#include <signal.h>
#include <stddef.h>
#include "ourhdr.h"

static void
sig_alm(void)
{
    return; /* nothing to do, just returning wakes up sigsuspend() */
}

unsigned int
sleep(unsigned int nsecs)
{
    struct sigaction  newact, oldact;
    sigset_t          newmask, oldmask, suspmask;
    unsigned int      unslept;

    newact.sa_handler = sig_alm;
    sigemptyset(&newact.sa_mask);
    newact.sa_flags = 0;
    sigaction(SIGALRM, &newact, &oldact);
    /* set our handler, save previous information */
    sigemptyset(&newmask);
    sigaddset(&newmask, SIGALRM);
    /* block SIGALRM and save current signal mask */
    sigprocmask(SIG_BLOCK, &newmask, &oldmask);

    alarm(nsecs);

    suspmask = oldmask;
    sigdelset(&suspmask, SIGALRM); /* make sure SIGALRM isn't blocked */
    sigsuspend(&suspmask);          /* wait for any signal to be caught */
    /* some signal has been caught, SIGALRM is now blocked */

    unslept = alarm(0);
    sigaction(SIGALRM, &oldact, NULL); /* reset previous action */
    /* reset signal mask, which unblocks SIGALRM */
    sigprocmask(SIG_SETMASK, &oldmask, NULL);

    return(unslept);
}
```

与程序10-4相比，为了可靠地实现 sleep，程序10-21的代码比较长。程序中没有使用任何形式的非局部转移（如程序10-5为了避免在 alarm 和 pause 之间的竞态条件所做的那样），所以对处理 SIGALRM 信号期间可能执行的其他信号处理程序没有影响。

10.20 作业控制信号

在表10-1中有六个POSIX.1认为是与作业控制有关的信号。

- SIGCHLD 子进程已停止或终止。
- SIGCONT 如果进程已停止，则使其继续运行。
- SIGSTOP 停止信号（不能被捕捉或忽略）。
- SIGTSTP 交互停止信号。
- SIGTTIN 后台进程组的成员读控制终端。

- SIGTTOU 后台进程组的成员写控制终端。

虽然仅当系统支持作业控制时，POSIX.1才要求它支持SIGCHLD，但是几乎所有UNIX版本都支持这种信号。我们已经说明了在子进程终止时这种信号是如何产生的。

大多数应用程序并不处理这些信号——交互式 shell通常做处理这些信号的所有工作。当键入挂起字符（通常是Ctrl-Z）时，SIGTSTP被送至后台进程组的所有进程。当通知 shell在前台或后台恢复一个作业时，shell向作业中的所有进程发送SIGCONT信号。与之类似的有，如果向一个进程递送了SIGTTIN或SIGTTOU信号，则根据系统默认，此进程停止，作业控制 shell了解到这一点后就通知我们。

一个例外是管理终端的进程——例如，vi(1)编辑程序。当用户要挂起它时，它需要能了解到这一点，这样就能将终端状态恢复到vi启动时的情况。另外，当在前台恢复它时，它需要将终端状态设置回所希望的状态，并需要重新绘制终端屏幕。可以在下面的例子中观察到类似vi这样的程序是如何处理这种情况的。

在作业控制信号间有某种相互作用。当对一个进程产生四种停止信号（SIGTSTP，SIGSTOP，SIGTTIN或SIGTTOU）中的任意一种时，对该进程的任一未决的SIGCONT信号就被丢弃。与之类似的是，当对一个进程产生SIGCONT信号时，对同一进程的任一未决的停止信号被丢弃。

注意，如果进程是停止的，SIGCONT的默认动作是继续一个进程，否则忽略此信号。通常，对该信号无需做任何事情。当对一个停止的进程产生一个SIGCONT信号时，该进程就继续，即使该信号是被阻塞或忽略的也是如此。

实例

程序10-22例示了当一个程序处理作业控制时所使用的通常的代码序列。该程序只是将其标准输入复制到其标准输出，但是在信号处理程序中以注释形式给出了管理屏幕的程序所执行的典型操作。当程序10-22启动时，仅当SIGTSTP信号的配置是SIG_DFL，它才安排捕捉该信号。其理由是：当此程序由不支持作业控制的shell(例如/bin/sh)所启动时，此信号的配置应当设置为SIG_IGN。实际上，shell并不显式地忽略此信号，而是init将这三个作业控制信号SIGTSTP、SIGTTIN和SIGTTOU设置为SIG_IGN。然后，这种配置由所有登录shell继承。只有作业控制shell才应将这三个信号重新设置为SIG_DFL。

当键入挂起字符时，进程接到SIGTSTP信号，然后该信号处理被调用。在此点上，应当进行与终端有关的处理：将光标移到左下角，恢复终端工作方式，等等。在将SIGTSTP重新设置为默认值(停止该进程)，并且解除了对此信号的阻塞之后，进程向自己发送同一信号SIGTSTP。因为正在处理SIGTSTP信号，而在捕捉到该信号期间系统自动地阻塞它，所以应当解除对此信号的阻塞。仅当某个进程（通常是正响应一个交互式fg命令的作业控制shell）向该进程发送一个SIGCONT信号时，该进程才继续。我们不捕捉SIGCONT信号。该信号的默认配置是继续停止的进程，当此发生时，此程序如同从kill函数返回一样继续运行。当此程序继续运行时，将SIGTSTP信号再设置为捕捉，并且做我们所希望做的终端处理（例如重新绘制屏幕）。

第18章将介绍处理特定的作业控制挂起字符的另一种方法，其中并不使用信号，而是由程序自身识别该特定字符。

程序10-22 如何处理SIGTSTP

```
#include <signal.h>
#include "ourhdr.h"

#define BUFFSIZE 1024

static void sig_tstp(int);

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

    /* only catch SIGTSTP if we're running with a job-control shell */
    if (signal(SIGTSTP, SIG_IGN) == SIG_DFL)
        signal(SIGTSTP, sig_tstp);

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

    exit(0);
}

static void
sig_tstp(int signo) /* signal handler for SIGTSTP */
{
    sigset_t mask;

    /* ... move cursor to lower left corner, reset tty mode ... */
    /* unblock SIGTSTP, since it's blocked while we're handling it */
    sigemptyset(&mask);
    sigaddset(&mask, SIGTSTP);
    sigprocmask(SIG_UNBLOCK, &mask, NULL);

    signal(SIGTSTP, SIG_DFL); /* reset disposition to default */
    kill(getpid(), SIGTSTP); /* and send the signal to ourself */
    /* we won't return from the kill until we're continued */
    signal(SIGTSTP, sig_tstp); /* reestablish signal handler */
    /* ... reset tty mode, redraw screen ... */
    return;
}
```

10.21 其他特征

本节介绍某些依赖于实现的信号的其他特征。

10.21.1 信号名字

某些系统提供数组

```
extern char *sys_siglist[];
```

数组下标是信号编号，数组中的元素是指向一个信号字符串名字的指针。

这些系统通常也提供函数 `psignal`。

```
#include <signal.h>
void psignal(int signo, const char.* msg);
```

字符串 `msg` (通常是程序名) 输出到标准出错, 后面跟着一个冒号和一个空格, 再跟着对该信号的说明, 最后是一个新行符。

SVR4和4.3+BSD都提供 `sys_siglist` 和 `psignal` 函数。

10.21.2 SVR4信号处理程序的附加参数

当调用 `sigaction` 对一个信号设置配置时, 可以指定 `sa_flags` 值 `SA_SIGINFO` (见表10-5)。这使得两个附加参数被传给信号处理程序。整型的信号编号总是作为第一个参数传送。第二个参数是一个空指针, 或者是一个指向 `siginfo` 结构的指针。(第三个参数提供进程内不同控制线程的有关信息, 在此不对它进行讨论。)

```
struct siginfo {
    int     si_signo; /* signal number */
    int     si_errno; /* if nonzero, errno value from <errno.h> */
    int     si_code; /* additional info (depends on signal) */
    pid_t   si_pid; /* sending process ID */
    uid_t   si_uid; /* sending process real user ID */
    /* other fields also */
};
```

对于由硬件产生的信号, 例如 `SIGFPE`, `si_code` 值给出附加的信息: `FPE_INTDIV` 表示整数除以0, `FPE_FLTDIV` 表示浮点数除以0等等。如果 `si_code` 小于或等于0, 则表示信号是由调用 `kill(2)` 的用户进程产生的。在此情况下, `si_pid` 和 `si_uid` 给出了发送此信号的进程的有关信息。依赖于正被捕捉的信号, 还有一些信息可用, 见 SVR4 `siginfo(5)` 手册页。

10.21.3 4.3+BSD信号处理程序的附加参数

4.3+BSD总是用三个参数调用信号处理程序:

```
handler(int signo, int code, struct sigcontext*);
```

参数 `signo` 是信号编号, `code` 给出某些信号的附加信息。例如, 对于 `SIGFPE` 的 `code` 值 `FPE_INTDIV_TRAP` 表示整数除以0。第三个参数 `scp` 与硬件有关。

10.22 小结

信号用于很多复杂的应用程序中。理解进行信号处理的原因和方式对于高级 UNIX 程序设计极其重要。本章对 UNIX 信号进行了详细而且比较深入的介绍。首先说明了早期的信号实施的问题以及它们是如何显现出来的。然后介绍了 POSIX.1 的可靠信号概念以及所有相关的函数。在此基础上接着提供了 `abort`、`system` 和 `sleep` 函数的 POSIX.1 实现。最后以观察分析作业控制信号结束。

习题

10.1 删除程序10-1中的 `for (;)` 语句, 结果会怎样? 为什么?

10.2 实现raise函数。

10.3 画出运行程序10-6时的堆栈情况。

10.4 程序10-8中利用setjmp和longjmp设置I/O操作的超时，下面的代码也常用于超时：

```
signal(SIGALRM, sig_alarm);
alarm(60);
if (setjmp (env_alarm) != 0) {
    /* handle time out */
    ...
}
...
```

这段代码有什么错误？

10.5 仅使用一个计时器(alarm或较高精度的setitimer)，构造一组函数，使得进程可以设置任一数目的计时器。

10.6 编写一段程序测试程序10-17中父进程和子进程的同步函数，要求进程创建一个文件并向文件写一个整数0，进程调用fork后父进程和子进程交替增加文件中的计数器的值，并打印哪一个进程(子进程或父进程)进行了增1操作。

10.7 在程序10-8中，若调用者捕捉了SIGABRT并从该信号处理程序中返回，为什么不是仅仅调用_exit，而要恢复缺省设置并再次调用kill？

10.8 为什么SVR4中的siginfo(见10.21节)在si_uid字段中传递实际的用户ID而非有效用户ID？

10.9 重写程序10-10，要求处理表10-1中的所有信号量，每次循环处理一个信号量而不是所有可能的信号量。

10.10 编写一段程序，要求在一个无限循环中调用sleep(60)函数，每5分钟(即5次循环)取当前的日期和时间，并打印tm_sec域。将程序执行一晚上，请解释其结果。一些程序如BSD中的cron守护进程是如何处理这类文件的？

10.11 修改程序3-3，要求：(a)将BUFSIZE改为100；(b)用signal_intr函数捕捉SIGXFSZ信号量并打印消息，然后从信号量处理程序中返回；(c)如果没有写满请求的字节数，打印write的返回值。将软资源限制RLIMIT_FSIZE(见7.11节)变为1024字节(在shell中设置软资源限制，如果不就直接在程序中的调用setrlimit)，然后拷贝一个大于1024字节的文件，在各种不同的系统上运行新程序，其结果如何？为什么？

10.12 写一段调用fwrite的程序，要求使用一个较大的缓存区(几个兆)，调用fwrite前调用alarm设置一秒钟以后调度信号量。在信号量处理程序中打印捕捉到的信号量然后返回。fwrite可以完成吗？结果如何？

第11章 终端 I/O

11.1 引言

在操作系统中，终端 I/O 处理是个非常繁琐的部分，UNIX 也不例外。在很多版本的 UNIX 手册中，终端 I/O 的手册页常常是最长的部分之一。SVID 中的 termio 手册页至少有 16 页。

70 年代后期，UNIX 终端 I/O 处理发展成两种不同的风格。一种是系统在 V7 的基础上进行了很多改变而形成的，这种风格由系统 沿续下来；另一种则是 V7 的风格，它正成为伯克利类系统的标准组成部分。如同信号一样，POSIX.1 在这两种风格的基础上制定了终端 I/O 标准。本章将介绍 POSIX.1 的终端函数，以及 SVR4 和 4.3+BSD 的增加部分。

终端 I/O 的用途很广泛，包括：终端、计算机之间的直接连接、调制解调器、打印机等等，所以它就变得非常复杂。在后面的若干章中，开发了两个例示终端 I/O 的程序：一个与 PostScript 打印机进行通信（见第 17 章），另一个涉及调制解调器以及远程计算机登录（见第 18 章）。

11.2 综述

终端 I/O 有两种不同的工作方式：

(1) 规范方式输入处理。在这种方式中，终端输入以行为单位进行处理。对于每个读要求，终端驱动程序最多返回一行。

(2) 非规范方式输入处理。输入字符不以行为单位进行装配。

如果不作特殊处理，则默认方式是规范方式。例如：若 shell 的标准输入、输出是终端，在用 read 和 write 将标准输入复制到标准输出时，终端以规范方式进行工作，每次 read 最多返回一行。处理整个屏幕的程序，例如 vi 编辑程序使用非规范方式，其原因是其命令是由不以新行符终止的一个或几个字符组成的。另外，该编辑程序使用了若干特殊字符作为编辑命令，所以它也不希望系统对特殊字符进行处理。例如，Ctrl-D 字符通常是终端的文件结束符，但在 vi 中它是向下滚动半个屏幕的命令。

V7 和 BSD 类的终端驱动程序支持三种终端输入方式：(a) 精细加工方式（输入装配成行，并对特殊字符进行处理），(b) 原始方式（输入不装配成行，也不对特殊字符进行处理），(c) cbreak 方式（输入不装配成行，但对某些特殊字符进行处理）。程序 11-10 显示了将终端设置为 cbreak 或原始方式的 POSIX.1 函数。

POSIX.1 定义了 11 个特殊输入字符，其中 9 个可以改变。本章已经用到了其中几个，例如：文件结束符（通常是 Ctrl-D），挂起字符（通常是 Ctrl-Z）。11.3 节对其中每个字符都进行了说明。

终端设备是由一般位于内核中的终端驱动程序所控制的。每个终端设备有一个输入队列，一个输出队列，见图 11-1。

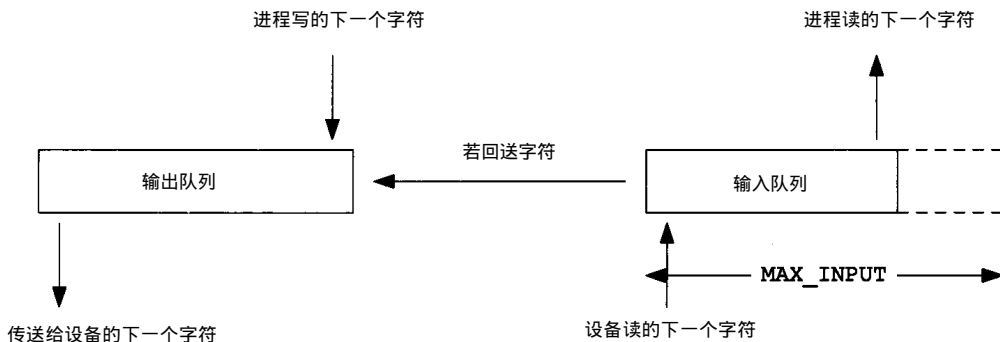


图11-1 终端设备的输入、输出队列的逻辑结构

对此图要说明下列几点：

- 如果需要回送，则在输入队列和输出队列之间有一个隐含的连接。
- 输入队列的长度MAX_INPUT（见表2-5）是有限值，当一个特定设备的输入队列已经填满时，系统作何种处理依赖于实现。当此发生时，大多数UNIX系统回送响铃字符。
- 图中没有显示另一个输入限制MAX_CANON，它是在一个规范输入行中的最大字节数。
- 虽然输出队列通常也是有限长度，但是程序不能存取定义其长度的常数。这是因为当输出队列要填满时，内核使写进程睡眠直至写队列中有可用的空间，所以程序无需关心该队列的长度。

• 我们将说明如何使用tcflush函数刷清输入或输出队列。与此类似，在说明tcsetattr函数时，将会了解到如何通知系统仅在输出队列空时改变一个终端的属性。（例如，正在改变输出属性时可能就要这样做。）我们也能通知系统，当它正在改变终端属性时，丢弃在输入队列中的任何东西。（如果正在改变输入属性，或者在规范和非规范方式之间进行转换，则可能希望这样做，以免以错误的方式对以前输入的字符进行解释。）

大多数UNIX系统在一个称为终端行规程（*terminal line discipline*）的模块中进行规范处理。它是位于内核类属读、写函数和实际设备驱动程序之间的模块，见图11-2。

12.4节讨论流I/O系统以及第19章讨论伪终端时还将使用此图。

所有我们可以检测和更改的终端设备特性都包含在 `termios` 结构中。该结构在头文件 `<termios.h>` 中定义，本章经常使用这一头文件。

```
struct termios {
    tcflag_t  c_iflag; /* input flags */
    tcflag_t  c_oflag; /* output flags */
    tcflag_t  c_cflag; /* control flags */
    tcflag_t  c_lflag; /* local flags */
    cc_t      c_cc[NCCS]; /* control characters */
};
```

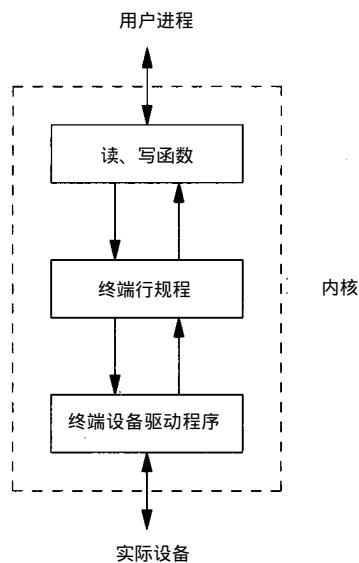


图11-2 终端行规程

粗略而言，输入标志由终端设备驱动程序用来控制输入特性（剥除输入字节的第 8 位，允许输入奇偶校验等等），输出标志则控制输出特性（执行输出处理，将新行映照为 CR/LF 等），控制标志影响到 RS-232 串行线（忽略调制解调器的状态线，每个字符的一个或两个停止位等等），本地标志影响驱动程序和用户之间的界面（回送的开或关，可视的擦除符，允许终端产生的信号，对后台作业输出的控制停止信号等）。

类型 `tcflag_t` 的长度是以保持每个标志值。它经常被定义为 `unsigned long`。`c_cc` 数组包含了所有可以更改的特殊字符。NCCS 是该数组的长度，其典型值在 11~18 之间（大多数 UNIX 实现支持的特殊字符较 POSIX.1 所定义的 11 个更多）。`cc_t` 类型的长度足以保持每个专用字符，典型的是 `unsigned char`。

系统 V 的早期版本有一个名为 `<termio.h>` 的头文件，一个名为 `termio` 的数据结构，为了区别于这些名字，POSIX.1 在这些名字后加了一个 `s`。

表 11-1 列出了所有可以更改以影响终端设备特性的终端标志。注意，POSIX.1 定义的标志是 SVR4 和 4.3+BSD 都支持的，但是它们还各有自己的扩充部分。这些扩充部分与这两个系统的各自历史发展过程有关。11.5 节将详细讨论这些标志值。

给出了表 11-1 中的所有选择项后，如何才能检测和更改终端设备的这些特性呢？表 11-2 列出了 POSIX.1 所定义的对终端设备进行操作的各个函数。（9.7 节已说明了 `tcgetpgrp` 和 `tcsetpgrp` 函数。）

表 11-1 终端标志

字 段	标 志	说 明	POSIX.1	SVR4	4.3+BSD 扩展	
c_iflag	BRKINT	接到 BREAK 时产生 SIGINT	•			
	ICRNL	将输入的 CR 转换为 NL	•			
	IGNBRK	忽略 BREAK 条件	•			
	IGNCR	忽略 CR	•			
	IGNPAR	忽略奇偶错字符	•			
	IMAXBEL	在输入队列空时振铃			•	•
	INLCR	将输入的 NL 转换为 CR	•			
	INPCK	打开输入奇偶校验	•			
	ISTRIP	剥除输入字符的第 8 位	•			
	IUCLC	将输入的大写字符转换成小写字符			•	
	IXANY	使任一字符都重新启动输出			•	•
	IXOFF	使起动/停止输入控制流起作用	•			
	IXON	使起动/停止输出控制流起作用	•			
	PARMRK	标记奇偶错	•			
c_oflag	BSDLY	退格延迟屏蔽			•	
	CRDLY	CR 延迟屏蔽			•	
	FFDLY	换页延迟屏蔽			•	
	NLDLY	NL 延迟屏蔽			•	
	OCRNL	将输出的 CR 转换为 NL			•	
	OFDEL	填充符为 DEL，否则为 NUL			•	

(续)

字 段	标 志	说 明	POSIX.1	SVR4 4.3+BSD 扩展
	OFILL	对于延迟使用填充符		•
	OLCUC	将输出的小写字符转换为大写字符		•
	ONLCR	将NL转换为CR-NL		• •
	ONLRET	NL执行CR功能		•
	ONOCR	在0列不输出CR		•
	ONOEOT	在输出中删除EOT字符		•
	OPOST	执行输出处理	•	
	OXTABS	将制表符扩充为空格		•
	TABDLY	水平制表符延迟屏蔽		•
	VTDLY	垂直制表符延迟屏蔽		•
c_cflag	CCTS_OFLOW	输出的CTS流控制		•
	CIGNORE	忽略控制标志		•
	CLOCAL	忽略解制-解调器状态行	•	
	CREAD	启用接收装置	•	
	CRTS_IFLOW	输入的RTS流控制		•
	CSIZE	字符大小屏蔽	•	
	CSTOPB	送两个停止位，否则为1位	•	
	HUPCL	最后关闭时断开	•	
	MDMBUF	经载波的流控输出		•
	PARENB	进行奇偶校	•	
	PARODD	奇校，否则为偶校	•	
c_lflag	ALTWERASE	使用替换WERASE算法		•
	ECHO	进行回送	•	
	ECHOCTL	回送控制字符为^(char)		• •
	ECHOE	可见擦除符	•	
	ECHOK	回送kill符	•	
	ECHOKE	kill的可见擦除		• •
	ECHONL	回送NL	•	
	ECHOPRT	硬拷贝的可见擦除方式		• •
	FLUSHO	刷清输出		• •
	ICANON	规范输入	•	
	IEXTEN	使扩充的输入字符处理起作用	•	
	ISIG	使终端产生的信号起作用	•	
	NOFLSH	在中断或退出键后不刷清	•	
	NOKERNINFO	STATUS不使内核输出		•
	PENDIN	重新打印		• •
	TOSTOP	对于后台输出发送SIGTTOU	•	
	XCASE	规范大/小写表示		•

注意，对终端设备，POSIX.1没有使用ioctl，而使用了表11-2中列出的12个函数。这样做的理由是：对于终端设备的ioctl函数，其最后一个参数的数据类型随执行动作的不同而不同。

虽然只有 12 个函数对终端设备进行操作，但是应当理解的是，表 11-2 中头两个函数 `tcgetattr` 和 `tcsetattr` 处理大约 50 种不同的标志（见表 11-1）。对于终端设备有大量选择项可供使用，对于一个特定设备（终端、调制解调器、激光打印机等等）又要决定所需的选择项，这些都使对终端设备的处理变得复杂起来。

表 11-2 POSIX.1 终端 I/O 函数

函 数	说 明
<code>tcgetattr</code>	取属性 (<code>termios</code> 结构)
<code>tcsetattr</code>	设置属性 (<code>termios</code> 结构)
<code>cfgetispeed</code>	得到输入速度
<code>cfgetospeed</code>	得到输出速度
<code>cfsetispeed</code>	设置输入速度
<code>cfsetospeed</code>	设置输出速度
<code>tcdrain</code>	等待所有输出都被传输
<code>tcflow</code>	挂起传输或接收
<code>tcflush</code>	刷清未决输入和 / 或输出
<code>tcsendbreak</code>	送 BREAK 字符
<code>tcgetpgrp</code>	得到前台进程组 ID
<code>tcsetpgrp</code>	设置前台进程组 ID

表 11-2 中列出的 12 个函数之间的关系示于图 11-3 中。

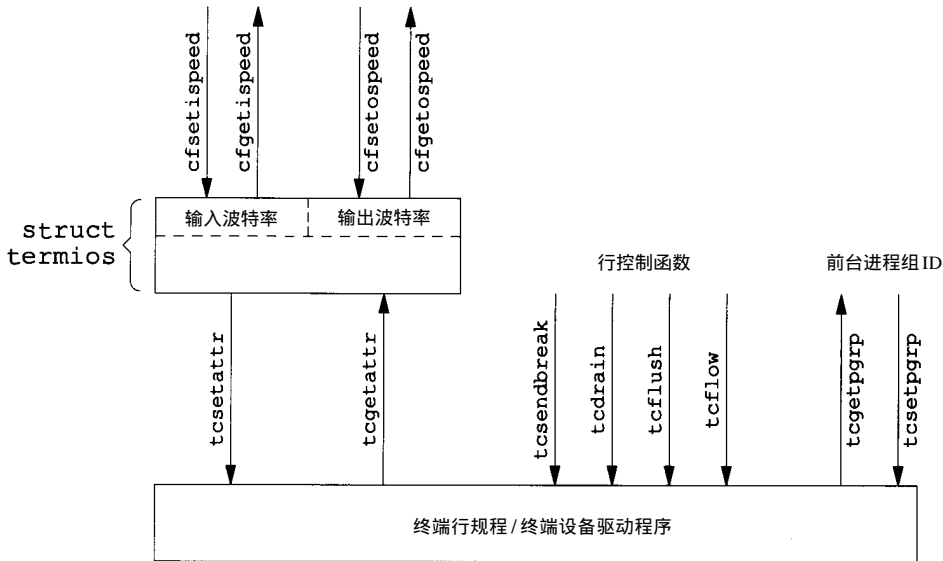


图 11-3 与终端有关的函数之间的关系

POSIX.1 没有规定在 `termios` 结构中何处存放波特率信息，这是一个依赖于实现的特性。很多早期的系统将此信息存放在 `c_cflag` 字段中。4.3+BSD 则在此结构中有两个分开的字段——一个存放输入速度，另一个则存放输出速度。

11.3 特殊输入字符

POSIX.1 定义了 11 个在输入时作特殊处理的字符。SVR4 另外加了 6 个特殊字符，4.3+BSD

则加了7个。表11-3列出了这些特殊字符。

表11-3 终端特殊输入字符

字 符	说 明	c_cc 下标	起作用, 由:		典型值	POSIX.1	SVR4 4.3+BSD 扩充
			字段	标志			
CR	回车	不能更改	c_lflag	ICANON	\r	•	
DISCARD	擦除输出	VDISCARD	c_lflag	IEXTEN	^O		• •
DSUSP	延迟挂起 (SIGTSTP)	VDSUSP	c_lflag	ISIG	^Y		• •
EOF	文件结束	VEOF	c_lflag	ICANON	^D	•	
EOL	行结束	VEOL	c_lflag	ICANON		•	
EOL2	替换的行结束	VEOL2	c_lflag	ICANON			• •
ERASE	擦除字符	VERASE	c_lflag	ICANON	^H	•	
INTR	中断信号 (SIGINT)	VINTR	c_lflag	ISIG	^?, ^C	•	
KILL	擦行	VKILL	c_lflag	ICANON	^U	•	
LNEXT	下一个字列字符	VLNEXT	c_lflag	IEXTEN	^V		• •
NL	新行	不能更改	c_lflag	ICANON	^n	•	
QUIT	退出信号 (SIGQUIT)	VQUIT	c_lflag	ISIG	^\ ^_	•	
REPRINT	再打印全部输入	VREPRINT	c_lflag	ICANON	^R		• •
START	恢复输出	VSTART	c_iflag	IXON/IXOFF	^Q	•	
STATUS	状态要求	VSTATUS	c_lflag	ICANON	^T		•
STOP	停止输出	VSTOP	c_iflag	IXON/IXOFF	^S	•	
SUSP	挂起信号 (SIGTSTP)	VSUSP	c_lflag	ISIG	^Z	•	
WERASE	擦除字	VWERASE	c_lflag	ICANON	^W		• •

在POSIX.1的11个特殊字符中, 可将其中9个更改为几乎任何值。不能更改的两个特殊字符是新行符和回车符 (\n和\r), 有些实施也不允许更改 STOP和START字符。为了进行修改, 只要更改termios结构中c_cc数组的相应项。该数组中的元素都用名字作为下标进行引用, 每个名字都以字母V开头 (见表11-3中的第3列)。

POSIX.1可选地允许禁止使用这些字符。若 _POSIX_VDISABLE有效, 则 _POSIX_VDISABLE的值可存放在 c_cc数组的相应项中以禁止使用该特殊字符。可以用 pathconf和 fpathconf函数查询此特征 (见2.5.4节)。

FIPS151-1要求支持_POSIX_VDISABLE。
 SVR4和4.3+BSD也支持此特性。SVR4将_POSIX_VDISABLE定义为0, 而4.3+BSD则将其定义为八进制数377。
 某些早期的UNIX系统所用的方法是: 若相应的特殊输入字符是0, 则禁止使用该字符。

实例

在详细说明各特殊字符之前, 先看一个更改特殊字符的程序。程序 11-1禁用中断字符, 并将文件结束符设置为Ctrl-B。

程序11-1 禁止中断字符和更改文件结束字符

```
#include <termios.h>
#include "ourhdr.h"

int
main(void)
{
    struct termios term;
```

```

long         vdisable;

if (isatty(STDIN_FILENO) == 0)
    err_quit("standard input is not a terminal device");

if ( (vdisable = fpathconf(STDIN_FILENO, _PC_VDISABLE)) < 0)
    err_quit("fpathconf error or _POSIX_VDISABLE not in effect");

if (tcgetattr(STDIN_FILENO, &term) < 0) /* fetch tty state */
    err_sys("tcgetattr error");

term.c_cc[VINTR] = vdisable;    /* disable INTR character */
term.c_cc[VEOF]  = 2;          /* EOF is Control-B */

if (tcsetattr(STDIN_FILENO, TCSAFLUSH, &term) < 0)
    err_sys("tcsetattr error");

exit(0);
}

```

对此程序要说明下列几点：

(1) 仅当标准输入是终端设备时才修改终端特殊字符。isatty（见11.9节）用于进行这种检测。

(2) 用fpathconf取_POSIX_VDISABLE值。

(3) 函数tcgetattr（见11.4节）从内核存取termios结构。在修改了此结构后，调用tcsetattr设置属性，这样就进行了我们所希望的修改。

(4) 禁止使用中断键与忽略中断信号是不同的。程序 11-1所做的是禁止使用使终端驱动程序产生SIGINT的特殊字符。但仍可使用kill函数向进程发送该信号。

下面较详细地说明各个特殊字符。我们称这些字符为特殊输入字符，但是其中有两个字符，STOP和START（Ctrl-S和Ctrl-Q）在输出时也对它们进行特殊处理。注意，这些字符中的大多数在被终端驱动程序识别并进行特殊处理后都被丢弃，并不将它们传送给执行读终端操作的进程。例外的字符是换行符（NL，EOL，EOL2）和回车符（CR）。

- CR POSIX.1的回车符。不能更改此字符。以规范方式进行输入时识别此字符。当设置了ICANON（规范方式）和ICRNL（将CR映照为NL）以及没有设置IGNCR（忽略CR）时，将CR转换成NL，并产生与NL符相同的作用。

此字符返回给读进程（多半是在转换成NL后）。

- DISCARD SVR4和4.3+BSD的删除符。在扩充方式下（IEXTEN），在输入中识别此字符。在输入另一个删除符之前或删除条件被清除之前（见FLUSHO选择项）此字符使后续输出都被删除。在处理此字符即被删除，不送向读进程。

- DSUSP SVR4和4.3+BSD的延迟-挂起作业控制字符。在扩充方式下，若作业控制被挂起并且ISIG标志被设置，则在输入中识别此字符。与SUSP字符的相同处是：延迟-挂起字符产生SIGTSTP信号，它被送至前台进程组中的所有进程（见图9-7）。但是并不是键入此字符时，而是在一个进程读控制终端时，此延迟-挂起字符才送向进程组。在处理此字符即被删除，不送向读进程。

- EOF POSIX.1的文件结束符。以规范方式进行输入时识别此字符。当键入此字符时，等待被读的所有字节都立即传送给读进程。如果没有字节等待读，则返回0。在行首输入一个EOF符是向程序指示文件结束的正常方式。在处理此字符即被删除，不送向读进程。

- EOL POSIX.1附加的行定界符，与NL作用相同。以规范方式进行输入时识别此字符。通常不使用此字符。此字符返回给读进程。

- EOL2 SVR4和4.3+BSD的附加行定界符，与NL作用相同。以规范方式输入时识别此字符。通常不使用此字符，此字符返回给读进程。
 - ERASE POSIX.1的擦除字符（退格）以规范方式输入时识别此字符。它擦除行中的前一个字符，但不会超越行首字符擦除上一行中的字符。在处理此字符即被擦除，不送向读进程。
 - INTR POSIX.1的中断字符。若设置了 ISIG标志，则在输入中识别此字符。它产生 SIGINT信号，该信号被送至前台进程组中的所有进程（见图 9-7）。在处理此字符即被删除，不送向读进程。
 - KILL POSIX.1的kill（杀死）字符。（名字“杀死”在这里又一次被误用，它应被称为行擦除符。）以规范方式输入时识别此字符。它擦除整个1行。在处理此字符即被删除，不送向读进程。
 - LNEXT SVR4和4.3+BSD的“字面上-下一个”字符。以规范方式输入时识别此字符，它使下一个字符的任何特殊含意都被忽略。这对本节提及的所有特殊字符都起作用。使用这一字符可向程序键入任何字符。在处理此字符即被删除，但输入的下一个字符则被传送给读进程。
 - NL POSIX.1的新行字符，它也被称为行定界符。不能更改此字符。以规范方式输入时识别此字符。此字符返回给读进程。
 - QUIT POSIX.1的退出字符。若设置了 ISIG标志，则在输入中识别此字符。它产生 SIGQUIT信号，该信号又被送至前台进程组中的所有进程（见图 9-7）。在处理此字符即被删除，不送向读进程。
- 回忆表 10-1，INTR和QUIT之间的区别是：QUIT字符不仅按默认终止进程，而且也产生 core文件。
- REPRINT SVR4和4.3+BSD的再打印字符。以扩充规范方式（设置了 IEXTEN和 ICANON标志）进行输入时识别此字符。它使所有未读的输入被输出（再回送）。在处理此字符即被删除，不送向读进程。
 - START POSIX.1的起动字符。若设置了 IXON标志则在输入中识别此字符；若设置 IXOFF标志，则作为输出自动产生此字符。在 IXON已设置时接收到的START字符使停止的输出（由以前输入的STOP字符造成）重新启动。在此情形下，在处理此字符即被删除，不送向读进程。
- 在IXOFF标志设置时，若输入不会使输入缓存溢出，则终端驱动程序自动地产生一 START字符以恢复以前被停止的输入。
- STATUS 4.3+BSD的状态-要求字符。以扩充、规范方式进行输入时识别此字符。它产生 SIGINFO信号，该信号又被送至前台进程组中的所有进程（见图 9-7）。另外，如果没有设置 NOKERNINFO标志，则有关前台进程组的状态信息也显示在终端上。在处理此字符即被删除，不送向读进程。
 - STOP POSIX.1的停止字符。若设置了IXON标志，则在输入中识别此字符；若IXOFF标志已设置则作为输出自动产生此字符。在IXON已设置时接收到STOP字符则停止输出。在此情形下，在处理此字符即被删除，不送向读进程。当输入一个START字符后，停止的输出重新启动。
- 在IXOFF设置时，终端驱动程序自动地产生一个STOP字符以防止输入缓存溢出。
- SUSP POSIX.1的挂起作业控制字符。若支持作业控制并且 ISIG标志已设置，则在输入中识别此字符。它产生 SIGTSTP信号，该信号又被送至前台进程组的所有进程（见图 9-7）。在处理此字符即被删除，不送向读进程。

• WERASE SVR4和4.3+BSD的字擦除字符。以扩充、规范方式进行输入时识别此字符。它使前一个字被擦除。首先，它向后跳过任一白空字符（空格或制表符），然后向后越过前一个记号，使光标处在前一个记号的第一个字符位置上。通常，前一个记号在碰到一个白空字符时即终止。但是，可用设置ALTWERASE标志来改变这一点。

此标志使前一个记号在碰到第一个非字母、数字符时即终止。在处理前，此字符即被删除，不送向读进程。

需要为终端设备定义的另一个“字符”是BREAK。BREAK实际上并不是一个字符，而是在异步串行数据传送时发生的一个条件。依赖于串行界面，可以有多种方式通知设备驱动程序发生了BREAK条件。大多数终端有一个标记为BREAK的键，用其可以产生BREAK条件，这就使得很多人认为BREAK就是一个字符。对于异步串行数据传送，BREAK是一个0值的位序列，其持续时间长于要求发送一个字节的的时间。整个0值位序列被视为是一个BREAK。11.8节将说明如何发送一个BREAK。

11.4 获得和设置终端属性

使用函数tcgetattr和tcsetattr可以获得或设置termios。这样也就可以检测和修改各种终端选择标志和特殊字符，以使终端按我们所希望的方式进行操作。

```
#include <termios.h>

int tcgetattr(int filedes, struct termios *termptr);

int tcsetattr(int filedes, int opt, const struct termios * termptr);
```

两个函数返回：若成功则为0，若出错则为-1

这两个函数都有一个指向termios结构的指针作为其参数，它们返回当前终端的属性，或者设置该终端的属性。因为这两个函数只对终端设备进行操作，所以若filedes并不引用一个终端设备则出错返回，errno设置为ENOTTY。

tcsetattr的参数opt使我们可以指定在什么时候新的终端属性才起作用。opt可以指定为下列常数中的一个：

- TCSANOW 更改立即发生。
- TCSADRAIN 发送了所有输出后更改才发生。若更改输出参数则应使用此选择项。
- TCSAFLUSH 发送了所有输出后更改才发生。更进一步，在更改发生时未读的所有输入数据都被删除（刷清）。

tcsetattr函数的返回值易于产生混淆。如果它执行了任意一种所要求的动作，即使未能执行所有要求的动作，它也返回0（表示成功）。如果该函数返回0，则我们有责任检查该函数是否执行了所有要求的动作。这就意味着，在调用tcsetattr设置所希望的属性后，需调用tcgetattr，然后将实际终端属性与所希望的属性相比较，以检测两者是否有区别。

11.5 终端选择标志

本节对表11-7中列出的各个终端选择标志按字母顺序作进一步说明，也指出该选择项出现在四个终端标志字段中的哪一个，以及该选择项是否是POSIX.1定义的，或是受到SVR4或4.3+BSD支持的。

所有列出的选择标志（除屏蔽标志外）都用一或多位表示，而屏蔽标志则定义多位。屏蔽

标志有一个定义名，每个值也有一个名字。例如，为了设置字符长度，首先用字符长度屏蔽标志CSIZE将表示字符长度的位清0，然后设置下列值之一：CS5、CS6、CS7或CS8。

由SVR4支持的6个延迟值也有屏蔽标志：BSDLY、CRDLY、FFDLY、NLDLY、TABDLY和VTDLY。对于每个延迟值的长度请参阅termio(7)手册页（AT&T [1991]）。如果指定了一个延迟，则OFILL和OFDEL标志决定是驱动器进行实际延迟还是只是传输填充字符。

实例

程序11-2例示了使用屏蔽标志取或设置一个值。

程序11-2 tcgetattr实例

```
#include <termios.h>
#include "ourhdr.h"

int
main(void)
{
    struct termios term;
    int size;

    if (tcgetattr(STDIN_FILENO, &term) < 0)
        err_sys("tcgetattr error");

    size = term.c_cflag & CSIZE;
    if (size == CS5) printf("5 bits/byte\n");
    else if (size == CS6) printf("6 bits/byte\n");
    else if (size == CS7) printf("7 bits/byte\n");
    else if (size == CS8) printf("8 bits/byte\n");
    else printf("unknown bits/byte\n");

    term.c_cflag &= ~CSIZE; /* zero out the bits */
    term.c_cflag |= CS8; /* set 8 bits/byte */

    if (tcsetattr(STDIN_FILENO, TCSANOW, &term) < 0)
        err_sys("tcsetattr error");

    exit(0);
}
```

下面说明各选择标志：

- **ALTWERASE** (`c_lflag`, 4.3+BSD) 此标志设置时，若输入了WERASE字符，则使用一个替换的字擦除算法。它不是向后移动到前一个白空字符为止，而是向后移动到第一个非字母、数字字符为止。

- **BRKINT** (`c_iflag`, POSIX.1) 若此标志设置，而IGNBRK未设置，则在接到BREAK时，输入、输出队列被刷清，并产生一个SIGINT信号。如果此终端设备是一个控制终端，则将此信号送给前台进程组各进程。

如果IGNBRK和BRKINT都没有设置，但是设置了PARMRK，则BREAK被读作为三个字节序列\377, \0和\0，如果PARMRK也没有设置，则BREAK被读作为单个字符\0。

- **BSDLY** (`c_oflag`, SVR4) 退格延迟屏蔽，此屏蔽的值是BS0或BS1。
- **CCTS_OFLOW** (`c_cflag`, 4.3+BSD) 输出的CTS流控制（见习题11.4）。
- **CIGNORE** (`c_cflag`, 4.3+BSD) 忽略控制标志。
- **CLOCAL** (`c_cflag`, POSIX.1) 如若设置，则忽略调制解调器状态线。这通常意味着该设备是本地连接的。若此标志未设置，则打开一个终端设备常常会阻塞到调制解调器回应。

- CRDLY (c_oflag, SVR4) 回车延迟屏蔽。此屏蔽的值是 CR0、CR1、CR2 和 CR3。
- CREAD (c_cflag, POSIX.1) 如若设置，则接收装置被启用，可以接收字符。
- CRTS_IFLOW (c_cflag, 4.3+BSD) 输入的 RTS 流控制（见习题 11.4）。
- CSIZE (c_cflag, POSIX.1) 此字段是一个屏蔽标志，它指明发送和接收的每个字节的位数。此长度不包括可能有的奇偶校验位。由此屏蔽定义的字段值是 CS5、CS6、CS7 和 CS8，分别表示每个字节包含 5、6、7 和 8 位。

- CSTOPB (c_cflag, POSIX.1) 如若设置，则使用两位作为停止位，否则只使用一位作为停止位。

- ECHO (c_lflag, POSIX.1) 如若设置，则将输入字符回送到终端设备。在规范方式和非规范方式下都可以回送字符。

- ECHOCTL (c_lflag, SVR4 和 4.3+BSD) 如若设置并且 ECHO 也设置，则除 ASCII TAB、ASCII NL、START 和 STOP 字符外，其他 ASCII 控制符（ASCII 字符集中的 0~37）都被回送为 ^X，其中，X 是相应控制字符代码值加 0100 所构成的字符。这就意味着 ASCII Ctrl-A 字符（01）被回送为 ^A。ASCII DELETE 字符（0177）则回送为 ^?. 如若此标志未设置，则 ASCII 控制字符按其原样回送。如同 ECHO 标志，在规范方式和非规范方式下此标志对控制字符回送都起作用。

应当了解的是：某些系统回送 EOF 字符产生的作用有所不同，其原因是 EOF 的典型值是 Ctrl-D，而这是 ASCII EOT 字符，它可能使某些终端挂断。请查看有关手册。

- ECHOE (c_lflag, POSIX.1) 如若设置并且 ICANON 也设置，则 ERASE 字符从显示中擦除当前行中的最后一个字符。这通常是在终端驱动程序中写三个字符序列：退格，空格，退格实现的。

如若支持 WERASE 字符，则 ECHOE 用一个或若干个上述三字符序列擦除前一个字。

如若支持 ECHOPRT 标志，则在这里所说明的 ECHOE 动作假定 ECHOPRT 标志没有设置。

- ECHOK (c_lflag, POSIX.1) 如若设置并且 ICANON 也设置，则 KILL 字符从显示中擦除当前行，或者输出 NL 字符（用以强调已擦除整个行）。如若支持 ECHOK 标志，则这里的说明假定 ECHOK 标志没有设置。

- ECHOK 标志 (c_lflag, SVR4 和 4.3+BSD) 如若设置并且 ICANON 也设置，则回送 KILL 字符的方式是擦去行中的每一个字符。擦除每个字符的方法则由 ECHOE 和 ECHOPRT 标志选择。

- ECHONL (c_lflag, POSIX.1) 如若设置并且 ICANON 也设置，即使没有设置 ECHO 也回送 NL 字符。

- ECHOPRT (c_lflag, SVR4 和 4.3+BSD) 如若设置并且 ICANON 和 ECHO 也都设置，则 ERASE 字符（以及 WERASE 字符，若受到支持）使所有正被擦除的字符按它们被擦除的方式打印。在硬拷贝终端上这常常是有用的，这样可以确切地看到哪些字符正被擦去。

- FFDLY (c_oflag, SVR4) 换页延迟屏蔽。此屏蔽标志值是 FF0 或 FF1。

- FLUSHO (c_lflag, SVR4 和 4.3+BSD) 如若设置，则刷清输出。当键入 DISCARD 字符时设置此标志，当键入另一个 DISCARD 字符时，此标志被清除。设置或清除此终端标志也可设置或清除此条件。

- HUPCL (c_cflag, POSIX.1) 如若设置，则当最后一个进程关闭此设备时，调制解调器控制线降低至低电平（也就是调制解调器的连接断开）。

- ICANON (c_lflag, POSIX.1) 如若设置，则按规范方式工作（见 11.10 节）。这使下列字符起作用：EOF、EOL、EOL2、ERASE、KILL、REPRINT、STATUS 和 WERASE。输入字符被装配成行。

如果不以规范方式工作，则读请求直接从输入队列取字符。在至少接到 MIN 个字节或已超过 TIME 值之前，read 将不返回。详细情况见 11.11 节。

- ICRNL (c_iflag, POSIX.1) 如若设置并且 IGNCR 未设置，即将接收到的 CR 字符转换成一个 NL 字符。

- IEXTEN (c_iflag, POSIX.1) 如若设置，则识别并处理扩充的、实现定义的特殊字符。

- IGNBRK (c_iflag, POSIX.1) 在设置时，忽略输入中的 BREAK 条件。关于 BREAK 条件是产生信号还是被读作为数据，请见 BRKINT。

- IGNCR (c_iflag, POSIX.1) 如若设置，忽略接收到的 CR 字符。若此标志未设置，而设置了 ICRNL 标志则将接收到的 CR 字符转换成一个 NL 字符。

- IGNPAR (c_iflag, POSIX.1) 在设置时，忽略带有结构错误（非 BREAK）或奇偶错的输入字节。

- IMAXBEL (c_iflag, SVR4 和 4.3+BSD) 当输入队列满时响铃。

- INLCR (c_iflag, POSIX.1) 如若设置，则接收到的 NL 字符转换成 CR 字符。

- INPCK (c_iflag, POSIX.1) 当设置时，使输入奇偶校验起作用。如若未设置 INPCK，则使输入奇偶校验不起作用。

奇偶“产生和检测”和“输入奇偶性检验”是不同的两件事。奇偶位的产生和检测是由 PARENB 标志控制的。设置该标志后使串行界面的设备驱动程序对输出字符产生奇偶位，对输入字符则验证其奇偶性。标志 PARODD 决定该奇偶性应当是奇还是偶。如果一个其奇偶性为错的字符已经来到，则检查 INPCK 标志的状态。若此标志已设置，则检查 IGNPAR 标志（以决定是否应忽略带奇偶错的输入字节），若不应忽略此输入字节，则检查 PARMRK 标志以决定向读进程应传送那种字符。

- ISIG (c_iflag, POSIX.1) 如若设置，则判别输入字符是否是要产生终端信号的特殊字符（INTR，QUIT，SUSP 和 DSUSP），若是，则产生相应信号。

- ISTRIP (c_iflag, POSIX.1) 当设置时，有效输入字节被剥离为 7 位。当此标志未设置时，则保留全部 8 位。

- IUCLC (c_iflag, SVR4) 将输入的大写字符映射为小写字符。

- IXANY (c_iflag, SVR4 和 4.3+BSD) 使任一字符都能重新启动输出。

- IXOFF (c_iflag, POSIX.1) 如若设置，则使起动-停止输入控制起作用。当终端驱动程序发现输入队列将要填满时，输出一个 STOP 字符。此字符应当由发送数据的设备识别，并使该设备暂停。此后，当已对输入队列中的字符进行了处理后，该终端驱动程序将输出一个 START 字符，使该设备恢复发送数据。

- IXON (c_iflag, POSIX.1) 如若设置，则使起动-停止输出控制起作用。当终端驱动程序接收到一个 STOP 字符时，输出暂停。在输出暂停时，下一个 START 字符恢复输出。如若未设置此标志，则 START 和 STOP 字符由进程读作为一般字符。

- MDMBUF (c_cflag, 4.3+BSD) 按照调制解调器的载波标志进行输出流控制。

- NLDLY (c_oflag, SVR4) 新行延迟屏蔽。此屏蔽的值是 NL0 和 NL1。

- NOFLSH (c_lflag, POSIX.1) 按系统默认，当终端驱动程序产生 SIGINT 和 SIGQUIT 信号时，输入、出队列都被刷新。另外，当它产生 SIGSUSP 信号时，输入队列被刷新。如若设置了 NOFLSH 标志，则在这些信号产生时，不对输入、出队列进行刷新。

- NOKERNINFO (c_lflag, 4.3+BSD) 当设置时，此标志阻止 STATUS 字符使前台进程组的状态信息显示在终端上。但是不论本标志是否设置，STATUS 字符使 SIGINFO 信号送至前台

进程组中的所有进程。

- OCRNL (c_oflag, SVR4) 如若设置, 将输出的CR字符映照为NL。
 - OFDEL (c_oflag, SVR4) 如若设置, 则输出填充字符是 ASCII DEL, 否则它是 ASCII NUL, 见OFILL标志。
 - OFILL (c_oflag, SVR4) 如若设置, 则为实现延迟, 发送填充字符(ASCII DEL或 ASCII NUL, 见OFDEL标志), 而不使用时间延迟。见 6个延迟屏蔽: BSDLY, CRDLY, FFDLY, NLDLY, TABDLY以及VTDLY。
 - OLCUC (c_oflag, SVR4) 如若设置, 将小写字符映射为大写。
 - ONLCR (c_oflag, SVR4和4.3+BSD) 如若设置, 将输出的NL字符映照为CR-NL。
 - ONLRET (c_oflag, SVR4) 如若设置, 则输出的NL字符将执行回车功能。
 - ONOCR (c_oflag, SVR4) 如若设置, 则在0列不输出CR。
 - ONOEOT (c_oflag, 4.3+BSD) 如若设置, 则在输出中删除EOT字符(^D)。在将Ctrl-D解释为挂断的终端上这可能是需要的。
 - OPOST (c_oflag, POSIX.1) 如若设置, 则进行实现定义的输出处理。关于c_oflag字的各种实现定义标志, 见表11-1。
 - OXTABS (c_oflag, 4.3+BSD) 如若设置, 制表符在输出中被扩展为空格。这与将水平制表延迟(TABDLY)设置为XTABS或TAB3产生同样效果。
 - PARENB (c_cflag, POSIX.1) 如若设置, 则对输出字符产生奇偶位, 对输入字符则执行奇偶性检验。若PARODD已设置, 则奇偶校验是奇校验, 否则是偶校验。也见INPCK、IGNPAR和PARMRK标志部分。
 - PARMRK (c_iflag, POSIX.1), 当设置时, 并且IGNPAR未设置, 则结构性错(非BREAK)和奇偶错的字节由进程读作为三个字符序列\377,\0和X, 其中X是接收到的具有错误的字节。如若ISTRIP未设置, 则一个有效的\377被传送给进程时为\377,\377。如若IGNPAR和PARMRK都未设置, 则结构性错和奇偶错的字节都被读作为一个字符\0。
 - PARODD (c_cflag, POSIX.1) 如若设置, 则输出和输入字符的奇偶性都是奇, 否则为偶。注意, PARENB标志控制奇偶性的产生和检测。
 - PENDIN (c_iflag, SVR4和4.3+BSD) 如若设置, 则在下一个字符输入时, 尚未读的任何输入都由系统重新打印。这一动作与键入REPRINT字符时的作用相类似。
 - TABDLY (c_oflag, SVR4) 水平制表延迟屏蔽。此屏蔽的值是TAB0、TAB1、TAB2或TAB3。
- XTABS的值等于TAB3。此值使系统将制表符扩展成空格。系统假定制表符所扩展的空格数到屏幕上最近一个8的倍数处为止。不能更改此假定。
- TOSTOP (c_lflag, POSIX.1) 如若设置, 并且该实现支持作业控制, 则将信号SIGTTOU送到试图与控制终端的一个后台进程的进程组。按默认, 此信号暂停该进程组中所有进程。如果写控制终端的进程忽略或阻塞此信号, 则终端驱动程序不产生此信号。
 - VTDLY (c_oflag, SVR4) 垂直制表延迟屏蔽。此屏蔽的值是VT0或VT1。
 - XCASE (c_lflag, SVR4) 如若设置, 并且ICANON也设置, 则认为终端是大写终端, 所以输入都变换为小写。为了输入一个大写字符, 在其前加一个\。与之类似, 输出一个大写字符也在其前加一个\ (这一标志已经过时, 现在几乎所有终端都支持大、小写字符)。

11.6 stty命令

上节说明的所有选择项, 在程序中都可用tcgetattr和tcsetattr函数(见11.4节)进行检查和

更改。在命令行中则用 `stty(1)` 命令进行检查和更改。`stty(1)` 命令是表 11-2 中所列的头 6 个函数的界面。如果以 `-a` 选择项执行此命令，则显示终端的所有选择项：

```
$ stty -a
speed 9600 baud; 34 rows; 80 columns;
lflags: icanon isig iexten echo echoe echok echoke -echonl echoctl
        -echoprt -altwerase -noflsh -tostop -mdmbuf -flusho -pendin
        -nokerninfo -extproc
iflags: istrip icrnl -inlcr -igncr ixon -ixoff ixany imaxbel -ignbrk
        brkint -inpck -ignpar -parmrk
oflags: opost onlcr -oxtabs
cflags: cread cs7 parenb -parodd hupcl -clocal -cstopb -crtsets
cchars: discard = ^O; dsusp = ^Y; eof = ^D; eol = <undef>;
        eol2 = <undef>; erase = ^H; intr = ^?; kill = ^U; lnext = ^V;
        quit = ^\; reprint = ^R; start = ^Q; status = ^T; stop = ^S;
        susp = ^Z; werase = ^W;
```

若在选择项名前有一个连字符，表示该选择项禁用。最后四行显示各终端特殊字符的设置（见 11.3 节）。第 1 行显示当前终端窗口的行数和列数，11.12 节将对此进行讨论。

因为 `stty` 命令是一条用户命令，而不是一个操作系统函数，所以它由 POSIX.2 说明。

系统 V 的 `stty` 在标准输入进行操作，将输出写到标准输出上。V7 和 BSD 系统则在标准输出上进行操作，将输出写到标准出错文件上。POSIX.2 的最近草案采用系统 V 的方法，4.3+BSD 也这样做。

V7 的 `stty` 手册页只有 1 页，SVR4 版本的 `stty` 手册页则有 6 页。终端驱动程序趋向于使用愈来愈多的选择项。

11.7 波特率函数

波特率 (baud rate) 是一个历史沿用的术语，现在它指的是“位/每秒”。虽然大多数终端设备对输入和输出使用同一波特率，但是只要硬件许可，可以将它们设置为两个不同值。

```
#include <termios.h>

speed_t cfgetispeed(const struct termios*);
speed_t cfgetospeed(const struct termios*);

int cfsetispeed(struct termios*, speed_t);
int cfsetospeed(struct termios*, speed_t);
```

两个函数返回：波特率值

两个函数返回：若成功为 0，出错为 -1。

两个 `cfget` 函数的返回值，以及两个 `cfset` 函数的 `speed` 参数都是下列常数之一：B50、B75、B110、B134、B150、B200、B300、B600、B1200、B1800、B2400、B4800、B9600、B19200 或 B38400。常数 B0 表示“挂断”。在调用 `tcsetattr` 时将输出波特率指定为 B0，则调制解调器的控制线就不再起作用。

使用这些函数时，应当理解输入、输出波特率是存放在图 11-3 所示的设备 `termios` 结构中的。在调用任一 `cfget` 函数之前，先要用 `tcgetattr` 获得设备的 `termios` 结构。与此类似，在调用任一

cfset函数后，应将波特率设置到 `termios` 结构中。为使这种更改影响到设备，应当调用 `tcsetattr` 函数。

如果所设置的波特率有错，则在调用 `tcsetattr` 之前，不会发现这种错误。

11.8 行控制函数

下列四个函数提供了终端设备的行控制能力。其中，参数 `filedes` 引用一个终端设备，否则出错返回，`errno` 设置为 `ENOTTY`。

```
#include <termios.h>

int tcdrain(int filedes);

int tcflow(int filedes, int action);

int tcflush(int filedes, int queue);

int tcsendbreak(int filedes, int duration);
```

四个函数返回：若成功则为 0，若出错则为 -1

`tcdrain` 函数等待所有输出都被发送。`tcflow` 用于对输入和输出流控制进行控制。`action` 参数应当是下列四个值之一。

- `TCOOFF` 输出被挂起。
- `TCOON` 以前被挂起的输出被重新启动。
- `TCIOFF` 系统发送一个 `STOP` 字符。这将使终端设备暂停发送数据。
- `TCION` 系统发送一个 `START` 字符。这将使终端恢复发送数据。

`tcflush` 函数刷清（抛弃）输入缓存（终端驱动程序已接收到，但用户程序尚未读）或输出缓存（用户程序已经写，但尚未发送）。`queue` 参数应当是下列三个常数之一：

- `TCIFLUSH` 刷清输入队列。
- `TCOFLUSH` 刷清输出队列。
- `TCIOFLUSH` 刷清输入、输出队列。

`tcsendbreak` 函数在一个指定的时间区间内发送连续的 0 位流。若 `duration` 参数为 0，则此种发送延续 0.25~0.5 秒之间。POSIX.1 说明若 `duration` 非 0，则发送时间依赖于实现。

SVR4 SVID 说明若 `duration` 非 0，则不发送 0 位。但是，SVR4 手册页中说，若 `duration` 非 0，则 `tcsendbreak` 的行为与 `tcdrain` 一样。另一个系统手册页则说，若 `duration` 非 0，则传送 0 位的时间是 $duration \times N$ ，其中 `N` 在 0.25~0.5 秒之间。从中可见，如何处理这种条件还没有统一样式。

11.9 终端标识

历史沿袭至今，在大多数 UNIX 系统中，控制终端的名字是 `/dev/tty`。POSIX.1 提供了一个运行时函数，可被调用来决定控制终端的名字。

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

```
char * ctermid(char *ptr);
```

返回：见下

如果 *ptr* 是非空，则它被认为是一个指针，指向长度至少为 `L_ctermid` 字节的数组，进程的控制终端名存放在该数组中。常数 `L_ctermid` 定义在 `<stdio.h>` 中。若 *ptr* 是一个空指针，则该函数为数组（通常作为静态变量）分配空间。同样，进程的控制终端名存放在该数组中。

在这两种情况中，该数组的起始地址被作为函数值返回。因为大多数 UNIX 系统都使用 `/dev/tty` 作为控制终端名，所以此函数的主要作用是帮助提高向其他操作系统的可移植性。

实例——`ctermid` 函数

程序 11-3 是 POSIX.1 `ctermid` 函数的一个实现。

程序 11-3 POSIX.1 `ctermid` 函数的实现

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

static char ctermid_name[L_ctermid];

char *
ctermid(char *str)
{
    if (str == NULL)
        str = ctermid_name;
    return(strcpy(str, "/dev/tty")); /* strcpy() returns str */
}
```

另外两个与终端标识有关的函数是 `isatty` 和 `ttyname`。如果文件描述符引用一个终端设备，则 `isatty` 返回真，而 `ttyname` 则返回在该文件描述符上打开的终端设备的路径名。

```
#include <unistd.h>

int isatty(int fildes);
```

返回：若为终端设备则为 1（真），否则为 0（假）

```
char *ttyname(int fildes);
```

返回：指向终端路径名的指针，若出错则为 `NULL`

实例——`isatty` 函数

如程序 11-4 所示，`isatty` 函数很容易实现。其中只使用了一个终端专用的函数 `tcgetattr`，并取其返回值。

程序 11-4 POSIX.1 `isatty` 函数的实现

```
#include <termios.h>

int
isatty(int fd)
{
    struct termios term;
```

```

    return(tcgetattr(fd, &term) != -1); /* true if no error (is a tty) */
}

```

程序11-5 测试isatty函数

```

#include    "ourhdr.h"

int
main(void)
{
    printf("fd 0: %s\n", isatty(0) ? "tty" : "not a tty");
    printf("fd 1: %s\n", isatty(1) ? "tty" : "not a tty");
    printf("fd 2: %s\n", isatty(2) ? "tty" : "not a tty");
    exit(0);
}

```

用程序11-5测试isatty函数，得到：

```

$ a.out
fd 0: tty
fd 1: tty
fd 2: tty
$ a.out </etc/passwd 2>/dev/null
fd 0: not a tty
fd 1: tty
fd 2: not a tty

```

实例——ttyname函数

ttyname函数（见程序11-6）稍长一点，因为它要搜索所有设备表项，寻找匹配项。其方法是读/dev目录，寻找具有相同设备号和i节点编号的表项。回忆4.23节，每个文件系统有一个唯一的设备号（stat结构中的st_dev字段，见4.2节），文件系统每个目录项有一个唯一的i节点号（stat结构中的st_ino字段）。在此函数中假定当找到一个匹配的设备号和匹配的i节点号时，就找到了所希望的目录项。也可验证这两个表项与st_rdev字段（终端设备的主、次设备号）相匹配，以及该目录项是一个字符特殊文件。但是，因为已经验证了文件描述符参数是一个终端设备以及一个字符特殊设备，而且在UNIX系统中，匹配的设备号和i节点号是唯一的，所以不再需要作另外的比较。

用程序11-7测试这一实现。运行程序11-7得到：

```

$ a.out </dev/console 2> /dev/null
fd 0: /dev/console
fd 1: /dev/ttyp3
fd 2: not a tty

```

程序11-6 POSIX.1 ttyname函数的实现

```

#include    <sys/types.h>
#include    <sys/stat.h>
#include    <dirent.h>
#include    <limits.h>
#include    <string.h>
#include    <termios.h>
#include    <unistd.h>

#define DEV    "/dev/"          /* device directory */

```

```

#define DEVLEN  sizeof(DEV)-1  /* sizeof includes null at end */
char *
ttyname(int fd)
{
    struct stat      fdstat, devstat;
    DIR              *dp;
    struct dirent    *dirp;
    static char      pathname[_POSIX_PATH_MAX + 1];
    char             *rval;

    if (isatty(fd) == 0)
        return(NULL);
    if (fstat(fd, &fdstat) < 0)
        return(NULL);
    if (S_ISCHR(fdstat.st_mode) == 0)
        return(NULL);

    strcpy(pathname, DEV);
    if ((dp = opendir(DEV)) == NULL)
        return(NULL);
    rval = NULL;
    while ((dirp = readdir(dp)) != NULL) {
        if (dirp->d_ino != fdstat.st_ino)
            continue;          /* fast test to skip most entries */

        strncpy(pathname + DEVLEN, dirp->d_name, _POSIX_PATH_MAX - DEVLEN);
        if (stat(pathname, &devstat) < 0)
            continue;
        if (devstat.st_ino == fdstat.st_ino &&
            devstat.st_dev == fdstat.st_dev) { /* found a match */
            rval = pathname;
            break;
        }
    }
    closedir(dp);
    return(rval);
}

```

程序11-7 测试ttyname函数

```

#include    "ourhdr.h"

int
main(void)
{
    printf("fd 0: %s\n", isatty(0) ? ttyname(0) : "not a tty");
    printf("fd 1: %s\n", isatty(1) ? ttyname(1) : "not a tty");
    printf("fd 2: %s\n", isatty(2) ? ttyname(2) : "not a tty");
    exit(0);
}

```

11.10 规范方式

规范方式很简单——发一个读请求，当一行已经输入后，终端驱动程序即返回。许多条件造成读返回。

- 所要求的字节数已读到时读即返回。无需读一个完整的行。如果读了部分行，那么也不会丢失任何信息——下一次读从前一次读的停止处开始。
- 当读到一个行定界符时，读返回。回忆 11.3 节，在规范方式中，下列字符被解释为“行

结束”：NL、EOL、EOL2和EOF。另外，在11.5节中也曾说明，如若已设置ICRNLC，但未设置IGNCR，则CR字符的作用与NL字符一样，所以它也终止一行。

在这五个行定界符中，其中只有一个EOF符在终端驱动程序对其进行处理后即被删除。其他四个字符则作为该行的最后一个字符返回调用者。

- 如果捕捉到信号而且该函数并不自动再启动（见10.5节），则读也返回。

实例——getpass函数

下面说明 getpass 函数，它读入用户在终端上键入的口令。此函数由 UNIX login(1)和 crypt(1)程序调用。为了读口令，该函数必须禁止回送，但仍可使终端以规范方式进行工作，因为用户在键入口令后，一定要键入回车，这样也就构成了一个完整行。程序 11-8是一个典型的UNIX实现。

程序11-8 getpass函数的实现

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

#define MAX_PASS_LEN 8 /* max #chars for user to enter */

char *
getpass(const char *prompt)
{
    static char buf[MAX_PASS_LEN + 1]; /* null byte at end */
    char *ptr;
    sigset_t sig, sigsave;
    struct termios term, termsave;
    FILE *fp;
    int c;

    if ( (fp = fopen(ctermid(NULL), "r+")) == NULL)
        return(NULL);
    setbuf(fp, NULL);

    sigemptyset(&sig); /* block SIGINT & SIGTSTP, save signal mask */
    sigaddset(&sig, SIGINT);
    sigaddset(&sig, SIGTSTP);
    sigprocmask(SIG_BLOCK, &sig, &sigsave);

    tcgetattr(fileno(fp), &termsave); /* save tty state */
    term = termsave; /* structure copy */
    term.c_lflag &= ~(ECHO | ECHOE | ECHOK | ECHONL);
    tcsetattr(fileno(fp), TCSAFLUSH, &term);

    fputs(prompt, fp);

    ptr = buf;
    while ( (c = getc(fp)) != EOF && c != '\n' ) {
        if (ptr < &buf[MAX_PASS_LEN])
            *ptr++ = c;
    }
    *ptr = 0; /* null terminate */
    putchar('\n', fp); /* we echo a newline */

    /* restore tty state */
    tcsetattr(fileno(fp), TCSAFLUSH, &termsave);

    /* restore signal mask */
    sigprocmask(SIG_SETMASK, &sigsave, NULL);
}
```



```

    fclose(fp);          /* done with /dev/tty */
    return(buf);
}

```

在此例中，有很多方面应当考虑：

- 调用函数 `ctermid` 打开控制终端，而不是直接将 `/dev/tty` 写在程序中。
- 只是读、写控制终端，如果不能以读、写方式打开此设备则出错返回。在有些系统中也使用一些其他约定。在 4.3+BSD 中，如果不能以读、写方式打开控制终端，则 `getpass` 从标准输入读，写到标准出错文件中。SVR4 则总是写到标准出错文件中，但只从控制终端读。

- 阻塞两个信号 `SIGINT` 和 `SIGTSTP`。如果不这样做，则在输入 `INTR` 字符时就会使程序终止，并使终端仍处于禁止回送状态。与此相类似，输入 `SUSP` 字符时将使程序暂停，并且在禁止回送状态下返回到 `shell`。在禁止回送时，选择了阻塞这两个信号。在读口令期间如果发生了这两个信号，则它们被保持，直到 `getpass` 返回前才解除对它们的阻塞。也有其他方法来处理这些信号。某些 `getpass` 版本忽略 `SIGINT`（保存它以前的动作），在返回前则将其动作恢复为以前的值。这就意味着在该信号被忽略期间所发生的这种信号都丢失。其他版本捕捉 `SIGINT`（保存它以前的动作），如果捕捉到此信号，则在重置终端状态和信号动作后，用 `kill` 函数发送此信号。没有一个 `getpass` 版本捕捉、忽略或阻塞 `SIGQUIT`，所以键入 `QUIT` 字符就会使程序夭折，并且极可能终端仍处于禁止回送状态。

- 要了解某些 `shell`，例如 `KornShell` 在以交互方式读输入时都使终端处于回送状态。这些 `shell` 是提供命令行编辑的 `shell`，因此在每次输入一条交互命令时都处理终端状态。所以如果在这种 `shell` 下调用此程序，并且用 `QUIT` 字符使其夭折，则这种 `shell` 可以恢复回送状态。不提供命令行编辑的 `shell`，例如 `Bourne shell` 和 `C shell` 将使程序夭折，并使终端仍处于不回送状态。如果对终端做了这种操作，则 `stty` 命令能使终端恢复到回送状态。

- 使用标准 I/O 读、写控制终端。我们特地将流设置为不带缓存的，否则在流的读、写之间可能会有某些相互作用（这样就需调用 `fflush`）。也可使用不带缓存的 I/O（见第 3 章），但是在这种情况下就要用 `read` 来实现 `getc`。

- 最多只可取 8 个字符作为口令。输入的多余字符则被忽略。

程序 11-9 调用 `getpass` 并且打印我们所输入的。这只是为了验证 `ERASE` 和 `KILL` 字符在正常工作（如同它们在规范方式下应该的那样）。

程序 11-9 调用 `getpass` 函数

```

#include    "ourhdr.h"
char      *getpass(const char *);
int
main(void)
{
    char    *ptr;

    if ( (ptr = getpass("Enter password:")) == NULL)
        err_sys("getpass error");
    printf("password: %s\n", ptr);

    /* now use password (probably encrypt it) ... */

    while (*ptr != 0)
        *ptr++ = 0;    /* zero it out when we're done with it */

    exit(0);
}

```

调用getpass函数的程序完成后，为了安全起见，应清除存放过用户键入的文本口令的存储区。如果该程序会产生其他用户能读的core文件（回忆10.2节，core的系统默认许可权使每个用户都能读它），或者如果某个其他进程能够设法读该进程的存储空间，则它们就能读到口令。

11.11 非规范方式

将termios结构中c_lflag字段的ICANON标志关闭就使终端处于非规范方式。在非规范方式中，输入数据不装配成行，不处理下列特殊字符：ERASE、KILL、EOF、NL、EOL、EOL2、CR、REPRINT、STATUS和WERASE。

如前所述，规范方式很容易——系统每次返回一行。但在非规范方式下，系统怎样才能知道在什么时候将数据返回给我们呢？如果它一次返回一个字节，那么系统开销就很大。（回忆表3-1，从中可以看到每次读一个字节的开销会多大。每次使返回的数据加倍，就使系统调用的开销减半。）在启动读数据之前，往往不知道要读多少数据，所以系统不能总是返回多个字节。

解决方法是：当已读了指定量的数据后，或者已经过了给定量的时间后，即通知系统返回。这种技术使用了termios结构中c_cc数组的两个变量：MIN和TIME。c_cc数组中的这两个元素的下标名为：VMIN和VTIME。

MIN说明一个read返回前的最小字节数。TIME说明等待数据到达的分秒数（秒的1/10为分秒）。有下列四种情形：

情形A：MIN>0, TIME>0

TIME说明一个字节间的计时器，在接到第一个字节时才启动它。在该计时器超时之前，若已接到MIN个字节，则read返回MIN个字节。如果在接到MIN个字节之前，该计时器已超时，则read返回已接收到的字节（因为只有当接到第一个字节时才启动，所以在计时器超时时，至少返回1个字节）。在这种情形中，在接到第一个字节之前，调用者阻塞。如果在调用read时数据已经可用，则这如同在read后，数据立即被接收到一样。

情形B：MIN>0, TIME==0

已经接到了MIN个字节时，read才返回。这可以造成read无限期的阻塞。

情形C：MIN==0, TIME>0

TIME指定了一个调用read时启动的读计时器。（与情形A相比较，两者是不同的）。在接到1个字节或者该计时器超时时，read即返回。如果是计时器超时，则read返回0。

情形D：MIN==0, TIME==0

如果有数据可用，则read最多返回所要求的字节数。若无数据可用，则read立即返回0。

在所有这些情形中，MIN只是最小值。如果程序要求的数据多于MIN个字节，那么它可能接收到所要求的字节数。这也适用于MIN==0的情形A和B。

表11-4列出了非规范方式下的四种不同情形。在表中，nbytes是read的第三个参数（返回的最大字节数）。

POSIX.1允许下标VMIN和VTIME的值分别与VEOF和VEOL相同。确实，SVR4就是这样做的。这样就提供了与系统V早期版本的兼容性。问题是从非规范方式转换为规范方式时，必须恢复VEOF和VEOL，如果不这样做，那么VMIN等于VEOF，并且它已被设置为典型值1，于是文件结束字符就变成Ctrl-A。解决这一问题最简单的方法是：在转入非规范方式时将整个termios结构保存起来。在以后再转回规范方式时恢复它。

表11-4 非规范输入的四种情形

	MIN > 0	MIN == 0
TIME > 0	A: 在计时器超时前, 读返回 [MIN, nbytes] ; 若计时器超时, 读返回 [1, MIN] (TIME=字节间计时器, 调用者可能无限阻塞)	C: 在计时器超时前, 读返回 [1, nbytes] ; 若计时器超时, 读返回 0 (TIME=读计时器。)
TIME == 0	B: 可用时, 读返回 [MIN, nbytes] , (调用者可能无限阻塞。)	D: 立即读返回 [0, nbytes]。

实例

程序11-10定义了函数tty_cbreak和tty_raw, 它们将终端分别设置为cbreak和原始方式(术语cbreak和raw来自于V7的终端驱动程序)。tty_reset函数的功能是将终端恢复为以前的工作方式。其中还提供了另外两个函数: tty_atexit, tty_termios。tty_atexit可被登记为终止处理程序, 以保证exit恢复终端工作方式。tty_termios则返回一个指向原先的规范方式termios结构的指针。第18章的调制解调器拨号程序中将使用所有这些函数。

程序11-10 将终端方式设置为原始或cbreak方式

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

static struct termios save_termios;
static int ttysavefd = -1;
static enum { RESET, RAW, CBREAK } ttystate = RESET;

int
tty_cbreak(int fd) /* put terminal into a cbreak mode */
{
    struct termios buf;
    if (tcgetattr(fd, &save_termios) < 0)
        return(-1);

    buf = save_termios; /* structure copy */
    buf.c_lflag &= ~(ECHO | ICANON);
    /* echo off, canonical mode off */

    buf.c_cc[VMIN] = 1; /* Case B: 1 byte at a time, no timer */
    buf.c_cc[VTIME] = 0;

    if (tcsetattr(fd, TCSAFLUSH, &buf) < 0)
        return(-1);
    ttystate = CBREAK;
    ttysavefd = fd;
    return(0);
}

int
tty_raw(int fd) /* put terminal into a raw mode */
{
    struct termios buf;

    if (tcgetattr(fd, &save_termios) < 0)
        return(-1);
```

```

buf = save_termios; /* structure copy */
buf.c_lflag &= ~(ECHO | ICANON | IEXTEN | ISIG);
                /* echo off, canonical mode off, extended input
                processing off, signal chars off */
buf.c_iflag &= ~(BRKINT | ICRNL | INPCK | ISTRIP | IXON);
                /* no SIGINT on BREAK, CR-to-NL off, input parity
                check off, don't strip 8th bit on input,
                output flow control off */
buf.c_cflag &= ~(CSIZE | PARENB);
                /* clear size bits, parity checking off */
buf.c_cflag |= CS8;
                /* set 8 bits/char */
buf.c_oflag &= ~(OPOST);
                /* output processing off */

buf.c_cc[VMIN] = 1; /* Case B: 1 byte at a time, no timer */
buf.c_cc[VTIME] = 0;

if (tcsetattr(fd, TCSAFLUSH, &buf) < 0)
    return(-1);
ttystate = RAW;
ttysavefd = fd;
return(0);
}

int
tty_reset(int fd)          /* restore terminal's mode */
{
    if (ttystate != CBREAK && ttystate != RAW)
        return(0);

    if (tcsetattr(fd, TCSAFLUSH, &save_termios) < 0)
        return(-1);
    ttystate = RESET;
    return(0);
}

void
tty_atexit(void)          /* can be set up by atexit(tty_atexit) */
{
    if (ttysavefd >= 0)
        tty_reset(ttysavefd);
}

struct termios *
tty_termios(void)         /* let caller see original tty state */
{
    return(&save_termios);
}

```

对cbreak方式的定义是：

- 非规范方式。如本节开始处所述，这种方式不对某些输入特殊字符进行处理。这种方式仍对信号进行处理，所以用户可以键入任一终端产生的信号。调用者应当捕捉这些信号，否则这种信号就可能终止程序，并且终端将仍处于cbreak方式。

作为一般规则，在编写更改终端方式的程序时，应当捕捉大多数信号，以便在程序终止前恢复终端方式。

- 关闭回送（ECHO）标志。
- 每次输入一个字节。为此将MIN设置为1，将TIME设置为0。这是表11-4中的情形B。至少有一个字节可用时，read再返回。

对原始方式的定义是：

• 非规范方式。另外，还关闭了对信号产生字符（ ISIG ）和扩充输入字符的处理（ IEXTEN ）。关闭 BRKINT ，这样就使 BREAK 字符不再产生信号。

• 关闭回送（ ECHO ）标志。

• 关闭 ICRNL、INPCK、ISTRIP 和 IXON 标志。于是：不再将输入的 CR 字符变换为 NL（ ICRNL ），使输入奇偶校验不起作用（ INPCK ），不再剥离输入字节的第 8 位（ ISTRIP ），不进行输出流控制（ IXON ）。

• 8 位字符（ CS8 ），不产生奇偶位，不进行奇偶性检测（ PARENB ）。

• 禁止所有输出处理（ OPOST ）。

• 每次输入一个字节（ MIN=1 ， TIME=0 ）。

程序 11-11 测试原始和 cbreak 方式。运行程序 11-11 可以观察这两种终端工作方式的工作情况。

\$ a.out

```
Enter raw mode characters, terminate with DELETE
4
33
133
62
63
60
172
键入DELETE
Enter cbreak mode characters, terminate with SIGINT
1 键入Ctrl-A
10 键入退格
signal caught 键入中断符
```

程序 11-11 测试原始和 cbreak 工作方式

```
#include <signal.h>
#include "ourhdr.h"

static void sig_catch(int);

int
main(void)
{
    int i;
    char c;

    if (signal(SIGINT, sig_catch) == SIG_ERR) /* catch signals */
        err_sys("signal(SIGINT) error");
    if (signal(SIGQUIT, sig_catch) == SIG_ERR)
        err_sys("signal(SIGQUIT) error");
    if (signal(SIGTERM, sig_catch) == SIG_ERR)
        err_sys("signal(SIGTERM) error");

    if (tty_raw(STDIN_FILENO) < 0)
        err_sys("tty_raw error");
    printf("Enter raw mode characters, terminate with DELETE\n");
    while ( ( i = read(STDIN_FILENO, &c, 1) ) == 1) {
        if ((c &= 255) == 0177) /* 0177 = ASCII DELETE */
            break;
        printf("%o\n", c);
    }
    if (tty_reset(STDIN_FILENO) < 0)
        err_sys("tty_reset error");
    if (i <= 0)
        err_sys("read error");
}
```

```

if (tty_cbreak(STDIN_FILENO) < 0)
    err_sys("tty_raw error");
printf("\nEnter cbreak mode characters, terminate with SIGINT\n");
while ( (i = read(STDIN_FILENO, &c, 1)) == 1) {
    c &= 255;
    printf("%o\n", c);
}
tty_reset(STDIN_FILENO);
if (i <= 0)
    err_sys("read error");
exit(0);
}

static void
sig_catch(int signo)
{
    printf("signal caught\n");
    tty_reset(STDIN_FILENO);
    exit(0);
}

```

在原始方式中，输入的字符是 Ctrl-D(04)和特殊功能键 F7。在所用的终端上，此功能键产生6个字符：*ESC* (033)，*[* (0133)，*2* (062)，*3* (063)，*0* (060)和*z* (0172)。注意，在原始方式下关闭了输出处理（*OPOST*），所以在每个字符后没有得到回车符。另外也要注意的，在 *cbreak* 方式下，不对输入特殊字符进行处理（所以对 Ctrl-D、文件结束符和退格等不进行特殊处理），但是对终端产生的信号则进行处理。

11.12 终端的窗口大小

SVR4和伯克利系统都提供了一种功能，用其可以对当前终端窗口的大小进行跟踪，在窗口大小发生变化时，使内核通知前台进程组。内核为每个终端和伪终端保存一个 *winsize* 结构。

```

struct winsize {
    unsigned short  ws_row;      /* rows, in characters */
    unsigned short  ws_col;      /* columns, in characters */
    unsigned short  ws_xpixel;   /* horizontal size, pixels (not used) */
    unsigned short  ws_ypixel;   /* vertical size, pixels (not used) */
};

```

此结构的作用是：

- (1) 用 *ioctl*（见 3.14 节）的 *TIOCGWINSZ* 命令可以取此结构的当前值。
- (2) 用 *ioctl* 的 *TIOCSWINSZ* 命令可以将此结构的新值存放到内核中。如果此新值与存放在内核中的当前值不同，则向前台进程组发送 *SIGWINCH* 信号。（注意，从表 10-1 中可以看出，此信号的系统默认动作是忽略。）
- (3) 除了存放此结构的当前值以及在此值改变时产生一个信号以外，内核对该结构不进行任何其他操作。对结构中的值进行解释完全是应用程序的工作。

提供这种功能的目的是，当窗口大小发生变化时通知应用程序（例如 *vi* 编辑程序）。应用程序接到此信号后，它可以取得窗口大小的新值，然后重绘屏幕。

实例

程序 11-12 打印当前窗口大小，然后睡眠。每次窗口大小改变时，就捕捉到 *SIGWINCH* 信号，然后打印新的窗口大小。必须用一个信号终止此程序。

程序11-12 打印窗口大小

```

#include <signal.h>
#include <termios.h>
#ifdef TIOCGWINSZ
#include <sys/ioctl.h> /* 4.3+BSD requires this too */
#endif
#include "ourhdr.h"

static void pr_winsize(int), sig_winch(int);

int
main(void)
{
    if (isatty(STDIN_FILENO) == 0)
        exit(1);

    if (signal(SIGWINCH, sig_winch) == SIG_ERR)
        err_sys("signal error");

    pr_winsize(STDIN_FILENO); /* print initial size */
    for ( ; ; ) /* and sleep forever */
        pause();
}

static void
pr_winsize(int fd)
{
    struct winsize size;

    if (ioctl(fd, TIOCGWINSZ, (char *) &size) < 0)
        err_sys("TIOCGWINSZ error");
    printf("%d rows, %d columns\n", size.ws_row, size.ws_col);
}

static void
sig_winch(int signo)
{
    printf("SIGWINCH received\n");
    pr_winsize(STDIN_FILENO);
    return;
}

```

在一个带窗口的终端上运行此程序得到：

```

$ a.out
35 rows, 80 columns          起始长度
SIGWINCH received          更改窗口大小：捕捉到信号
40 rows, 123 columns
SIGWINCH received          再一次
42 rows, 33 columns
^? $                        键入中断以终止

```

11.13 termcap,terminfo和curses

termcap的意思是终端性能 (terminal capability)，它涉及到文本文件/etc/termcap和一套读此文件的例程。termcap这种技术是在伯克利为了支持 vi编辑器而发展起来的。termcap文件包含了对各种终端的说明：终端支持哪些功能 (行、列数、是否支持退格等)，如何使终端执行某些操作 (清屏、将光标移动到指定位置等)。把这些信息从需要编译的程序中取出来并把它放在易于编辑的文本文件中，这样就使得 vi能在很多不同的终端上运行。

然后，支持 termcap文件的一套例程也从 vi编辑程序中抽取出来，放在一个单独的 curses

(光标)库中。为使这套库可被要进行屏幕处理的任何程序使用,增加了很多功能。

termcap这种技术不是很完善的。当越来越多的终端被加到该数据文件中时,为了找到一个特定的终端就需使用较长的时间扫描此文件。此数据文件也只用两个字符的名字来标识不同的终端属性。这些缺陷导致开发另一种新技术——terminfo及其相关的curses库。在terminfo中,终端说明基本上是文本说明的编译版本,在运行时易于快速定位。terminfo由SVR2开始使用,此后所有系统V版本都使用它。

系统V使用terminfo,而4.3+BSD则使用termcap。

Goodheart [1991]对terminfo和curses库进行了详细说明。Strang, Mui和O'Reilly[1991]则对termcap和terminfo进行了说明。

不论是termcap还是terminfo都致力于本章所述及的问题——更改终端的方式、更改终端特殊字符、处理窗口大小等等。它们所提供的是在各种终端上执行典型操作(清屏、移动光标)的方法。另一方面,在本章所述问题方面curses能提供更详细的帮助。curses提供了很多函数,包括:设置原始方式、设置cbreak方式、打开和关闭回送等等。但是curses是为字符终端设计的,而当前的趋势则是向以像素为基础的图形终端发展。

11.14 小结

终端有很多特征和选择项,其中大多数都可按需进行改变。本章说明了很多更改终端操作的函数——特殊输入字符和选择标志的函数,介绍了可为终端设备设置的各个终端特殊字符以及很多选择项。

终端的输入方式有两种——规范的(每次一行)和非规范的。本章中包含了若干这两种工作方式的实例,也提供了一些函数,它们在POSIX.1终端选择项和较早的BSD cbreak及原始方式之间进行变换。本章也说明了如何取用和改变终端的窗口大小。第17和18章包含了终端I/O的另外一些实例。

习题

11.1 写一个调用tty_raw并且不恢复终端模式就终止的程序。如果系统提供reset(1)命令(SVR4和4.3+BSD中都提供),使用该命令恢复终端模式。

11.2 c_cflag字段的PARODD标志允许我们设置奇偶校验,而BSD中的tip程序也允许奇偶校验位是0或1,它是如何实现的?

11.3 如果你的系统中stty(1)命令输出MIN和TIME值,做下面的练习。两次登录系统,其中一次登录时打开vi编辑器,在另外一次登录中用stty命令确定vi设置的MIN和TIME值(终端为非规范模式)。

11.4 随着终端接口的行速度变得越来越快(19 200和38 400在当前是非常一般的),硬件的流控制就越来越重要,它包括取代XON和XOFF字符的RS-232 RTS和CTS。POSIX.1没有指定硬件流控制。在SVR4和4.3+BSD中,进程是如何开关硬件流控制?

第12章 高级 I/O

12.1 引言

本章内容包括：非阻塞 I/O、记录锁、系统 V 流机制、I/O 多路转接（select 和 poll 函数）、readv 和 writev 函数，以及存储映照 I/O（mmap）。第 14 章、第 15 章中的进程间通信，以及以后各章中的很多实例都要使用本章所述的概念和函数。

12.2 非阻塞 I/O

10.5 节中曾将系统调用分成两类：低速系统调用和其他。低速系统调用是可能会使进程永远阻塞的一类系统调用：

- 如果数据并不存在，则读文件可能会使调用者永远阻塞（例如读管道，终端设备和网络设备）
- 如果数据不能立即被接受，则写这些同样的文件也会使调用者永远阻塞。
- 在某些条件发生之前，打开文件会被阻塞（例如打开一个终端设备可能需等到与之连接的调制解调器应答；又例如若以只写方式打开 FIFO，那么在没有任何其他进程已用读方式打开该 FIFO 时也要等待）。

- 对已经加上强制性记录锁的文件进行读、写。
- 某些 ioctl 操作。
- 某些进程间通信函数（见第 14 章）。

虽然读、写磁盘文件会使调用在短暂时间内阻塞，但并不能将它们视为“低速”。

非阻塞 I/O 使我们调用不会永远阻塞的 I/O 操作，例如 open, read 和 write。如果这种操作不能完成，则立即出错返回，表示该操作如继续执行将继续阻塞下去。

对于一个给定的描述符有两种方法对其指定非阻塞 I/O：

- (1) 如果是调用 open 以获得该描述符，则可指定 O_NONBLOCK 标志（见 3.3 节）。
- (2) 对于已经打开的一个描述符，则可调用 fcntl 打开 O_NONBLOCK 文件状态标志（见 3.13 节）。程序 3-5 中的函数可用来为一个描述符打开任一文件状态标志。

早期的系统 V 版本使用标志 O_NDELAY 指定非阻塞方式。在这些版本中，如果无数据可读，则 read 返回值 0。而 UNIX 又常将 read 的返回值 0 解释为文件结束，两者有所混淆。POSIX.1 则提供了一个非阻塞标志，它的名字和语义都与 O_NDELAY 不同。POSIX.1 要求，对于一个非阻塞的描述符若无数据可读，则 read 返回 -1，并且 errno 被设置为 EAGAIN。SVR4 支持较老的 O_NDELAY 和 POSIX.1 的 O_NONBLOCK，但在本书的实例中只使用 POSIX.1 规定的特征。O_NDELAY 的使用只是为了向后兼容，不应在新应用程序中使用。

4.3BSD 为 fcntl 提供 FNDELAY 标志，其语义也稍有区别。它不只影响该描述符的文件状态标志，还将终端设备或套接口的标志更改成非阻塞的，因此影响了终端或套接口的所有用户，不只是影响共享同一文件表项的用户（4.3BSD 非阻塞 I/O 只对终端和套接口起作用）。如果对一个非阻塞描述符的操作不能无阻塞地完成

成，那么4.3BSD返回EWOULDBLOCK。4.3+BSD提供POSIX.1的O_NONBLOCK标志，但其语义却类似于4.3BSD的FNDELAY。非阻塞I/O通常用来处理终端设备或网络连接，而这些设备通常一次由一个进程使用。这就意味着 BSD语义的更改通常不会影响我们。出错返回EWOULDBLOCK而不是POSIX.1的EAGAIN，这造成了可移植性问题，必须处理这一问题。4.3+BSD也支持FIFO，非阻塞I/O也对FIFO起作用。

实例

程序12-1是一个非阻塞I/O的实例，它从标准输入读100 000字节，并试图将它们写到标准输出上。该程序先将标准输出设置为非阻塞的，然后用for循环进行输出，每次写的结果都在标准出错上打印。函数clr_fl类似于程序3-5中的set_fl，但与set_fl的功能相反，它清除1个或多个标志位。

程序12-1 长的非阻塞写

```
#include <sys/types.h>
#include <errno.h>
#include <fcntl.h>
#include "ourhdr.h"

char buf[100000];

int
main(void)
{
    int ntwrite, nwrite;
    char *ptr;

    ntwrite = read(STDIN_FILENO, buf, sizeof(buf));
    fprintf(stderr, "read %d bytes\n", ntwrite);

    set_fl(STDOUT_FILENO, O_NONBLOCK); /* set nonblocking */

    for (ptr = buf; ntwrite > 0; ) {
        errno = 0;
        nwrite = write(STDOUT_FILENO, ptr, ntwrite);
        fprintf(stderr, "nwrite = %d, errno = %d\n", nwrite, errno);
        if (nwrite > 0) {
            ptr += nwrite;
            ntwrite -= nwrite;
        }
    }

    clr_fl(STDOUT_FILENO, O_NONBLOCK); /* clear nonblocking */
    exit(0);
}
```

若标准输出是普通文件，则可以期望write只执行一次。

```
$ ls -l /etc/termcap                打印文件长度
-rw-rw-r-- 1 root 133439 Oct 11 1990 /etc/termcap
$ a.out < /etc/termcap > temp.fi 试试一普通文件
read 100000 bytes
nwrite = 100000, errno = 0          一次写
```

```
$ ls -l temp.file          检验输出文件长度
-rw-rw-r-- 1 stevens 100000 Nov 21 16:27 temp.file
```

但是，若标准输出是终端，则期望 write 有时会返回一个数字，有时则出错返回。下面是在一个系统上运行程序 12-1 的结果：

```
$ a.out < /etc/termcap 2>stderr.out  向终端输出
                                       大量输出至终端

$ cat stderr.out
read 100000 bytes
nwrite = 8192, errno = 0
nwrite = 8192, errno = 0
nwrite = -1, errno = 11                这种错211次
...
nwrite = 4096,errno = 0
nwrite = -1, errno = 11                这种错658次
...
nwrite = 4096,errno = 0
nwrite = -1, errno = 11                这种错604次
...
nwrite = 4096,errno = 0
nwrite = -1, errno = 11                这种错1047次
...
nwrite = -1, errno = 11                这种错1046次
...
nwrite = 4096, errno = 0                ...等等
```

在该系统上，errno 11 是 EAGAIN。系统上的终端驱动程序总是一次接收 4096 或 8192 字节。在另一个系统上，前三个 write 返回 2005，1822 和 1811，然后是 96 次出错返回，接着是返回 1846 等等。每个 write 能写多少字节依赖于系统。

此程序若在 SVR4 中运行，则其结果完全不同于前面的情况。当输出到终端上时，输出整个输入文件只需要一个 write。显然，非阻塞方式并不构成区别。创建一个更大的输入文件，并且系统为运行该程序增加了程序缓存。程序的这种运行方式（即输出一整个文件，只调用一次 write）一直继续到输入文件长度达到约 700 000 字节。达到此长度后，每一个 write 都返回出错 EAGAIN。（输入文件则决不会再输出到终端上——该程序只是连续地产生出错消息流。）

发生这种情况是因为：在 SVR4 中终端驱动程序通过流 I/O 系统连接到程序（12.4 节将详细说明流）。流系统有它自己的缓存，它一次能从程序接收更多的数据。SVR4 的行为也依赖于终端类型——硬连线终端、控制台设备或伪终端。

在此实例中，程序发出了数千个 write 调用，但是只有 20 个左右是真正输出数据的，其余的则出错返回。这种形式的循环称为轮询，在多用户系统上它浪费了 CPU 时间。12.5 节将介绍非阻塞描述符的 I/O 多路转接，这是一种进行这种操作的更加有效的方法。

第 17 章将会用到非阻塞 I/O，我们将要输出到终端设备（PostScript 打印机）而且不希望在 write 上阻塞。

12.3 记录锁

当两个人同时编辑一个文件时，其后果将如何呢？在很多 UNIX 系统中，该文件的最后状态取决于写该文件的最后一个进程。但是对于有些应用程序，例如数据库，有时进程需要确保它正在单独写一个文件。为了向进程提供这种功能，较新的 UNIX 系统提供了记录锁机制。（第 16

章包含了使用记录锁的数据库子程序库。)

记录锁 (record locking) 的功能是：一个进程正在读或修改文件的某个部分时，可以阻止其他进程修改同一文件区。对于 UNIX，“记录”这个定语也是误用，因为 UNIX 内核根本没有使用文件记录这种概念。一个更合适的术语可能是“区域锁”，因为它锁定的只是文件的一个区域（也可能是整个文件）。

12.3.1 历史

表12-1列出了各种UNIX系统提供的不同形式的记录锁。

表12-1 各种UNIX系统支持的记录锁形式

系 统	建议性	强制性	fcntl	lockf	flock
POSIX.1	•		•		
XPG3	•		•		
SVR2	•		•	•	
SVR3, SVR4	•	•	•	•	
4.3BSD	•				•
4.3BSD Reno	•		•		•

本节的最后将说明建议性锁和强制性锁之间的区别。POSIX.1选择了以fcntl函数为基础的系统V风格的记录锁。这种风格也得到4.3BSD Reno版本的支持。

早期的伯克利版只支持BSD flock函数。此函数只锁整个文件，而不锁文件中的一个区域。但是POSIX.1的fcntl函数可以锁文件中的任一区域，大至整个文件，小至单个字节。

本书只说明POSIX.1的fcntl锁。系统V的lockf函数只是fcntl函数的一个界面。

记录锁是1980年由John Bass最早加到V7上的。内核中相应的系统调用入口表项是名为locking的函数。此函数提供了强制性记录锁功能，它被用在很多制造商的系统III版本中。Xenix系统采用了此函数，SVR4在Xenix兼容库中仍旧支持该函数。SVR2是系统V中第一个支持fcntl风格记录锁的版本（1984年）。

12.3.2 fcntl记录锁

3.13节中已经给出了fcntl函数的原型，为了叙说方便，这里再重复一次。

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

int fcntl(int fildes, int cmd, ... /* struct flock *flockptr */);
```

返回：若成功则依赖于cmd（见下），若出错则为-1

对于记录锁，cmd是F_GETLK、F_SETLK或F_SETLKW。第三个参数（称其为flockptr）是一个指向flock结构的指针。

```

struct flock {
    short  l_type; /* F_RDLCK, F_WRLCK, or F_UNLCK */
    off_t  l_start; /* offset in bytes, relative to l_whence */
    short  l_whence; /* SEEK_SET, SEEK_CUR, or SEEK_END */
    off_t  l_len; /* length, in bytes; 0 means lock to EOF */
    pid_t  l_pid; /* returned with F_GETLK */
};

```

flock结构说明：

- 所希望的锁类型：F_RDLCK（共享读锁）、F_WRLCK（独占性写锁）或F_UNLCK（解锁一个区域）

- 要加锁或解锁的区域的起始地址，由l_start和l_whence两者决定。l_start是相对位移量（字节），l_whence则决定了相对位移量的起点。这与lseek函数（见3.6节）中最后两个参数类似。

- 区域的长度，由l_len表示。

关于加锁和解锁区域的说明还要注意下列各点：

- 该区域可以在当前文件尾端处开始或越过其尾端处开始，但是不能在文件起始位置之前开始或越过该起始位置。

- 如若l_len为0，则表示锁的区域从其起点（由l_start和l_whence决定）开始直至最大可能位置为止。也就是不管添写到该文件中多少数据，它都处于锁的范围。

- 为了锁整个文件，通常的方法是将l_start说明为0，l_whence说明为SEEK_SET，l_len说明为0。

上面提到了两种类型的锁：共享读锁（l_type为L_RDLCK）和独占写锁（L_WRLCK）。

基本规则是：多个进程在一个给定的字节上可以有一把共享的读锁，但是在一个给定字节上的写锁则只能由一个进程独用。

更进一步而言，如果在一个给定字节上已经有一把或多把读锁，则不能在该字节上再加写锁；如果在一个字节上已经有一把独占性的写锁，则不能再对它加任何读锁。

在表12-2中示出了这些规则。

加读锁时，该描述符必须是读打开；加写锁时，该描述符必须是写打开。

以下说明fcntl函数的三种命令：

• F_GETLK 决定由flockptr所描述的锁是否被另外一把锁所排斥（阻塞）。如果存在一把锁，它阻止创建由flockptr所描述的锁，则这把现存的锁的信息写到flockptr指向的结构中。如果不存在这种情况，则除了将l_type设置为F_UNLCK之外，flockptr所指向结构中的其他信息保持不变。

• F_SETLK 设置由flockptr所描述的锁。如果试图建立一把按上述兼容性规则并不允许的锁，则fcntl立即出错返回，此时errno设置为EACCES或EAGAIN。

• F_SETLKW 设置由flockptr所描述的锁。如果试图建立一把按上述兼容性规则并不允许的锁，则fcntl立即出错返回，此时errno设置为EACCES或EAGAIN。

表12-2 不同类型锁之间的兼容性

	要求	
	读锁	写锁
无锁	可以	可以
一把或多把读锁	可以	拒绝
一把写锁	拒绝	拒绝

区域当前有

SVR2和SVR4返回EACCES，但手册页警告将来返回EAGAIN。4.3+BSD则返回EAGAIN。POSIX.1允许这两种情况。

此命令也用来清除由flockptr说明的锁（l_type为F_UNLCK）。

• `F_SETLKW` 这是`F_SETLK`的阻塞版本（命令名中的`W`表示等待（wait））。如果由于存在其他锁，那么按兼容性规则由`flockptr`所要求的锁不能被创建，则调用进程睡眠。如果捕捉到信号则睡眠中断。

应当了解，用`F_GETLK`测试能否建立一把锁，然后用`F_SETLK`和`F_SETLKW`企图建立一把锁，这两者不是一个原子操作。在这两个操作之间可能会有另一个进程插入并建立一把相关的锁，使原来测试到的情况发生变化，如果不希望在建立锁时可能产生的长期阻塞，则应使用`F_SETLK`，并对返回结果进行测试，以判别是否成功地建立了所要求的锁。

在设置或释放文件上的一把锁时，系统按需组合或裂开相邻区。例如，若对字节0~99设置一把读锁，然后对字节0~49设置一把写锁，则有两个加锁区：0~49字节（写锁）及50~99（读锁）。又如，若100~199字节是加锁的区，需解锁第150字节，则内核将维持两把锁，一把用于100~149字节，另一把用于151~199字节。

实例——要求和释放一把锁

为了避免每次分配`flock`结构，然后又填入各项信息，可以用程序12-2中的函数`lock_reg`来处理这些细节。

程序12-2 加锁和解锁一个文件区域的函数

```
#include <sys/types.h>
#include <fcntl.h>
#include "ourhdr.h"

int
lock_reg(int fd, int cmd, int type, off_t offset, int whence, off_t len)
{
    struct flock    lock;

    lock.l_type = type;      /* F_RDLCK, F_WRLCK, F_UNLCK */
    lock.l_start = offset;  /* byte offset, relative to l_whence */
    lock.l_whence = whence; /* SEEK_SET, SEEK_CUR, SEEK_END */
    lock.l_len = len;      /* #bytes (0 means to EOF) */

    return( fcntl(fd, cmd, &lock) );
}
```

因为大多数锁调用是加锁或解锁一个文件区域（命令`F_GETLK`很少使用），故通常使用下列五个宏，它们都定义在`ourhdr.h`中（见附录B）。

```
#define read_lock(fd,offset,whence,len) \
    lock_reg(fd,F_SETLK,F_RDLCK,offset,whence,len)
#define readw_lock(fd,offset,whence,len) \
    lock_reg(fd,F_SETLKW,F_RDLCK,offset,whence,len)
#define write_lock(fd,offset,whence,len) \
    lock_reg(fd,F_SETLK,F_WRLCK,offset,whence,len)
#define writew_lock(fd,offset,whence,len) \
    lock_reg(fd,F_SETLKW,F_WRLCK,offset,whence,len)
#define un_lock(fd,offset,whence,len) \
    lock_reg(fd,F_SETLK,F_UNLCK,offset,whence,len)
```

我们用与`lseek`函数同样的顺序定义这些宏中的三个参数。

实例——测试一把锁

程序12-3定义了一个函数`lock_test`，可用其测试一把锁。

程序12-3 测试一个锁条件的函数

```

#include <sys/types.h>
#include <fcntl.h>
#include "ourhdr.h"

pid_t
lock_test(int fd, int type, off_t offset, int whence, off_t len)
{
    struct flock lock;

    lock.l_type = type; /* F_RDLCK or F_WRLCK */
    lock.l_start = offset; /* byte offset, relative to l_whence */
    lock.l_whence = whence; /* SEEK_SET, SEEK_CUR, SEEK_END */
    lock.l_len = len; /* #bytes (0 means to EOF) */

    if (fcntl(fd, F_GETLK, &lock) < 0)
        err_sys("fcntl error");

    if (lock.l_type == F_UNLCK)
        return(0); /* false, region is not locked by another proc */
    return(lock.l_pid); /* true, return pid of lock owner */
}

```

如果存在一把锁，它阻塞由参数说明的锁，则此函数返回持有这把现存锁的进程的 ID，否则此函数返回0。通常用下面两个宏来调用此函数（它们也定义在 ourhdr.h）。

```

#define is_readlock(fd,offset,whence,len) \
        lock_test(fd,F_RDLCK,offset,whence,len)
#define is_writelock(fd,offset,whence,len) \
        lock_test(fd,F_WRLCK,offset,whence,len)

```

实例——死锁

如果两个进程相互等待对方持有并且不释放（锁定）的资源时，则这两个进程就处于死锁状态。如果一个进程已经控制了文件中的一个加锁区域，然后它又试图对另一个进程控制的区域加锁，则它就会睡眠，在这种情况下，有发生死锁的可能性。

程序12-4给出了一个死锁的例子。子进程锁字节0，父进程锁字节1。然后，它们中的每一个又试图锁对方已经加锁的字节。在该程序中使用了 8.8节中介绍的父-子进程同步例程（TELL_XXX和WAIT_XXX），使得对方都能建立第一把锁。运行程序12-4得到：

程序12-4 死锁检测实例

```

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include "ourhdr.h"

static void lockabyte(const char *, int, off_t);

int
main(void)
{
    int fd;
    pid_t pid;

    /* Create a file and write two bytes to it */
    if ( (fd = creat("templock", FILE_MODE)) < 0)
        err_sys("creat error");
    if (write(fd, "ab", 2) != 2)

```

```

        err_sys("write error");

    TELL_WAIT();
    if ( (pid = fork()) < 0)
        err_sys("fork error");

    else if (pid == 0) {                /* child */
        lockabyte("child", fd, 0);
        TELL_PARENT(getppid());
        WAIT_PARENT();
        lockabyte("child", fd, 1);

    } else {                            /* parent */
        lockabyte("parent", fd, 1);
        TELL_CHILD(pid);
        WAIT_CHILD();
        lockabyte("parent", fd, 0);
    }
    exit(0);
}

static void
lockabyte(const char *name, int fd, off_t offset)
{
    if (writew_lock(fd, offset, SEEK_SET, 1) < 0)
        err_sys("%s: writew_lock error", name);

    printf("%s: got the lock, byte %d\n", name, offset);
}

```

```

$ a.out
child:got the lock,byte 0
parent:got the lock,byte 1
child:writew_lock error:Deadlock situation detected/avoided
parent:got the lock,byte 0

```

检测到死锁时，内核必须选择一个进程收到出错返回。在本实例中，选择了子进程，这是一个实现细节。当此程序在另一个系统上运行时，一半次数是子进程接到出错信息，另一半则是父进程。

12.3.3 锁的隐含继承和释放

关于记录锁的自动继承和释放有三条规则：

(1) 锁与进程、文件两方面有关。这有两重含意：第一重很明显，当一个进程终止时，它所建立的锁全部释放；第二重意思就不很明显，任何时候关闭一个描述符时，则该进程通过这一描述符可以存访的文件上的任何一把锁都被释放（这些锁都是该进程设置的）。这就意味着如果执行下列四步：

```

fd1=open(pathname, ...);
read_lock(fd1, ...);
fd2=dup(fd1);
close(fd2);

```

则在close (fd2) 后，在fd1上设置的锁被释放。如果将dup代换为open，其效果也一样：

```

fd1=open(pathname, ...);
read_lock(fd1, ...);
fd2=open(pathname, ...);

```



```
close(fd2);
```

(2) 由fork产生的子程序不继承父进程所设置的锁。这意味着，若一个进程得到一把锁，然后调用fork，那么对于父进程获得的锁而言，子进程被视为另一个进程，对于从父进程处继承过来的任一描述符，子进程要调用fcntl以获得它自己的锁。这与锁的作用是相一致的。锁的作用是阻止多个进程同时写同一个文件（或同一文件区域）。如果子进程继承父进程的锁，则父、子进程就可以同时写同一个文件。

(3) 在执行exec后，新程序可以继承原执行程序的锁。

POSIX.1没有要求这一点。但是，SVR4和4.3+BSD都支持这一点。

12.3.4 4.3+BSD的实现

先简要地观察4.3+BSD实现中使用的数据结构，从中可以看到锁是与进程、文件相关联的。考虑一个进程，它执行下列语句（忽略出错返回）：

```
fd1 = open(pathname, ...);
write_lock(fd1, 0, SEEK_SET, 1); /* parent write locks byte 0 */
if (fork() > 0) { /* parent */
    fd2 = dup(fd1);
    fd3 = open(pathname, ...);
    pause();
} else {
    read_lock(fd1, 1, SEEK_SET, 1); /* child read locks byte 1 */
    pause();
}
```

图12-1显示了父、子进程暂停（执行pause()）后的数据结构情况。

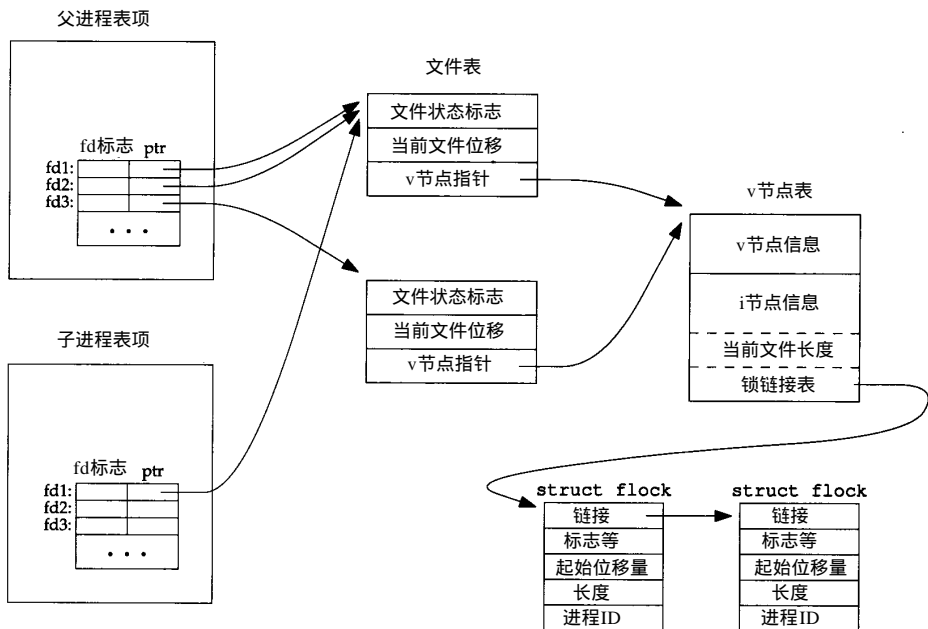


图12-1 关于记录锁的4.3+BSD数据结构

图3-3和图8-1中已显示了open、fork以及dup后的数据结构。有了记录锁后，在原来的这些图上新加了flock结构，它们由i节点结构开始相互连接起来。注意，每个flock结构说明了一个给定进程的一个加锁区域。图中显示了两个flock结构，一个是由父进程调用write_lock形成的，另一个则是由子进程调用read_lock形成的。每一个结构都包含了相应进程ID。

在父进程中，关闭fd1、fd2和fd3中的任意一个都将释放由父进程设置的写锁。在关闭这三个描述符中的任意一个时，内核会从该描述符所关联的i节点开始，逐个检查flock连接表中各项，并释放由调用进程持有的各把锁。内核并不清楚也不关心父进程是用哪一个描述符来设置这把锁的。

实例

建议性锁可由精灵进程使用以保证该精灵进程只有一个副本在运行。起动时，很多精灵进程都把它们进程ID写到一个各自专用的PID文件上。系统停机时，可以从这些文件中取用这些精灵进程的进程ID。防止一个精灵进程有多份副本同时运行的方法是：在精灵进程开始运行时，在它的进程ID文件上设置一把写锁。如果在它运行时一直保持这把锁，则不可能再起动它的其他副本。程序12-5实现了这一技术。

因为进程ID文件可能包含以前的精灵进程ID，而且其长度可能长于当前进程的ID，例如该文件中以前的内容可能是12345\n，而现在的进程ID是654，我们希望该文件现在只包含654\n，而不是654\n5\n，所以在写该文件时，先将其截短为0。注意，要在设置了锁之后再调用截短文件长度的函数ftruncate。在调用open时不能指定O_TRUNC，因为这样做会在有一个这种精灵进程运行并对该文件加锁时也会使该文件截短为0。（如果使用强制性锁而不是建议性锁，则可使用O_TRUNC。本节最后将讨论强制性锁。）

在本实例中，也对该描述符设置了运行时关闭标志。这是因为精灵进程常常fork并exec其他进程，并无需在另一个进程中使该文件也处在打开状态。

程序12-5 精灵进程阻止其多份副本同时运行的起动代码

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

#define PIDFILE "daemon.pid"

int
main(void)
{
    int    fd, val;
    char   buf[10];

    if ( (fd = open(PIDFILE, O_WRONLY | O_CREAT, FILE_MODE)) < 0 )
        err_sys("open error");

    /* try and set a write lock on the entire file */
    if (write_lock(fd, 0, SEEK_SET, 0) < 0) {
        if (errno == EACCES || errno == EAGAIN)
            exit(0); /* gracefully exit, daemon is already running */
        else
            err_sys("write_lock error");
    }

    /* truncate to zero length, now that we have the lock */
```

```

if (ftruncate(fd, 0) < 0)
    err_sys("ftruncate error");

    /* and write our process ID */
sprintf(buf, "%d\n", getpid());
if (write(fd, buf, strlen(buf)) != strlen(buf))
    err_sys("write error");

    /* set close-on-exec flag for descriptor */
if ( (val = fcntl(fd, F_GETFD, 0)) < 0)
    err_sys("fcntl F_GETFD error");
val |= FD_CLOEXEC;
if (fcntl(fd, F_SETFD, val) < 0)
    err_sys("fcntl F_SETFD error");

/* leave file open until we terminate: lock will be held */

/* do whatever ... */

exit(0);
}

```

实例

在相对文件尾端加锁或解锁时需要特别小心。大多数实现按照 `l_whence` 的 `SEEK_CUR` 或 `SEEK_END` 值，用文件当前位置或当前长度以及 `l_start` 得到绝对的文件位移量。但是，通常需要相对于文件的当前位置或当前长度指定一把锁。

程序12-6写一个大文件，一次一个字节。在每次循环中，从文件当前尾端开始处加锁直到将来可能扩充到的尾端为止（最后一个参数，长度指定为 0），并写1个字节。然后解除这把锁，写另一个字节。如果系统用“从当前尾端开始，直到将来可能扩充的尾端”这种记法来跟踪锁，那么这段程序能够正常工作。但是如果系统将相对位移量变换成绝对位移量就会有问题。在 SVR4中运行此程序的确会发生问题：

程序12-6 显示相对于文件末尾的锁的问题的程序

```

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include "ourhdr.h"

int
main(void)
{
    int i, fd;

    if ( (fd = open("temp.lock", O_RDWR | O_CREAT | O_TRUNC,
                    FILE_MODE)) < 0)
        err_sys("open error");

    for (i = 0; i < 1000000; i++) { /* try to write 2 Mbytes */
        /* lock from current EOF to EOF */
        if (writew_lock(fd, 0, SEEK_END, 0) < 0)
            err_sys("writew_lock error");

        if (write(fd, &fd, 1) != 1)
            err_sys("write error");

        if (un_lock(fd, 0, SEEK_END, 0) < 0)
            err_sys("un_lock error");

        if (write(fd, &fd, 1) != 1)

```

```

        err_sys("write error");
    }
    exit(0);
}

```

```

$ a.out
writew_lock error: No record locks available
$ ls -l temp.lock
-rw-r--r--  1 stevens  other    592 Nov  1 04:41 temp.lock

```

(内核返回ENOLCK。它表示内核中的锁表已用完。)分析系统是如何做的会从中得到教益。

图12-2显示了第一次调用writew_lock和write之后的文件状态。

因为在writew_lock调用中,指定“直至将来可能扩充到尾端”,所以图中锁定区域超过了所写的第一个字节。

然后调用un_lock。从当前尾端处开始直至将来可能扩充到的尾端为止解锁,它将图12-2中箭头的右端缩回到第一字节位置尾端部。然后将第二个字节写到文件中。图12-3显示了调用un_lock以及写了第二个字节后的文件状态。

在经过第二次for循环后,在文件上共写了4个字节。图12-4显示了此时文件及锁的状态。

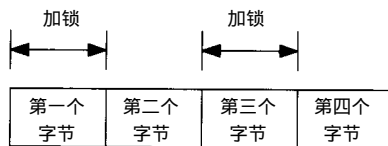
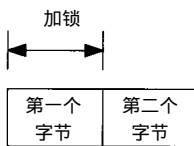
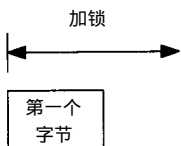


图12-2 第一次调用writew_lock和write之后的文件状态

图12-3 调用un_lock和写第二个字节后的状态

图12-4 第二次for循环后文件及锁的状态

这种情况不断重复,直至内核为该进程用完了锁结构。此时fcntl出错返回,errno设置为ENOLCK。

在此程序中,每次写的字节数是已知的,所以可将un_lock的第二个参数(其值将赋与l_start)改换成所写字节数的负值(在本程序中是-1)。这就使得un_lock去除上次加的锁。

实际上,在开发_db_writedat和_db_writeidx函数时,作者的系统出现了问题。16.7节给出了关于此问题的一个稍稍不同的方法。

12.3.5 建议性锁和强制性锁

考虑数据库存取例程序。如果该库中所有函数都以一致的方法处理记录锁,则称使用这些函数存取数据库的任何进程集为合作进程(*cooperating process*)。如果这些函数是唯一的用来存取数据库的函数,那么它们使用建议性锁是可行的。但是建议性锁并不能阻止对数据库文件有写许可的任何其他进程写数据库文件。不使用协同一致的方法(数据库存取例程序)来存取数据库的进程是一个非合作进程。

强制性锁机制中,内核对每一个open、read和write都要检查调用进程对正在存取的文件是否违背了某一把锁的作用。

表12-1显示了SVR4提供强制性记录锁,而POSIX.1不提供。

对一个特定文件打开其设置-l组-ID位,关闭其组-s-执行位则对该文件启动了强制性锁机制。

(回忆程序4-4)。因为当组-执行位关闭时，设置-组-ID位不再有意义，所以SVR3的设计者借用两者的这种组合来指定对一个文件的锁是强制性的而非建议性的。

如果一个进程试图读、写一个强制性锁起作用的文件，而欲读、写的部分又由其他进程加上了读、写锁，此时会发生什么呢？对这一问题的回答取决于三方面的因素：操作类型（read或write），其他进程保有的锁的类型（读锁或写锁），以及有关描述符是阻塞还是非阻塞的。表12-3列出了这8种可能性。

表12-3 强制性锁对其他进程读、写的影响

	阻塞描述符，试图		非阻塞描述符，试图	
	读	写	读	写
在区域上的读锁	可以	阻塞	可以	EAGAIN
在区域上的写锁	阻塞	阻塞	EAGAIN	EAGAIN

除了表12-3中的read、write函数，其他进程的强制性锁也会对open函数产生影响。通常，即使正在打开的文件具有强制性记录锁，该打开操作也会成功。下面的read或write依从于表12-3中所示的规则。但是，如果欲打开的文件具有强制性锁（读锁或写锁），而且open调用中的flag为O_TRUNC或O_CREAT，则不论是否指定O_NONBLOCK，open都立即出错返回，errno设置为EAGAIN。（对O_TRUNC情况出错返回是有意义的，因为其他进程对该文件持有读、写锁，所以不能将其截短为0。对O_CREAT情况在返回时也设置errno则无意义，因为该标志的意义是如果该文件不存在则创建，由于其他进程对该文件持有记录锁，因而该文件肯定是存在的。）

这种处理方式可能导致令人惊异的结果。我们曾编写过一个程序，它打开一个文件（其模式指定为强制性锁），然后对该文件的整体设置一把读锁，然后进入睡眠一段时间。在这段时间内，用某些常规的UNIX程序和操作符对该文件进行处理，发现下列情况：

- 可用ed编辑程序对该文件进行编辑操作，而且编辑结果可以写回磁盘！强制性记录锁对此毫无影响。对ed操作进行跟踪分析发现，ed将新内容写到一个临时文件中，然后删除原文件，最后将临时文件名改名为原文件名。于是，发现强制性锁机制对unlink函数没有影响。

在SVR4中，用truss(1)命令可以得到一个进程的系统调用跟踪信息，在4.3+BSD中，则使用ktrace(1)和kdump(1)命令。

- 不能用vi编辑程序编辑该文件。vi可以读该文件，但是如果试图将新的数据写到该文件中，则出错返回（EAGAIN）。如果试图将新数据添加到该文件中，则write阻塞。vi的这种行为与所希望的一样。

- 使用KornShell的>和>>算符重写或添写到该文件中，产生出错信息“cannot creat”。

- 在Bourne shell下使用>算符出错，但是使用>>算符则阻塞，在删除了强制性锁后再继续进行处理。（执行添加操作所产生的区别是因为：Korn Shell以O_CREAT和O_APPEND标志打开文件，而上面已提及指定O_CREAT会产生出错返回。但是，Bourne shell在该文件已存在时并不指定O_CREAT，所以open成功，而下一个write则阻塞。）

从这样一个例子中可见，在使用强制性锁时还需有所警惕。

一个别有用心的用户可以对大家都可读的文件加一把读锁（强制性），这样就能阻止任何其他人写该文件（当然，该文件应当是强制性锁机制起作用的，这可能要求该用户能够更改该文件的许可权位）。考虑一个数据库文件，它是大家都可读的，并且是强制性锁机制起作用的。

如果一个别有用心的用户对该整个文件保有一把读锁，则其他进程不能再写该文件。

实例

程序12-7用于确定一个系统是否支持强制性锁机制。

程序12-7 确定是否支持强制性锁

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <errno.h>
#include <fcntl.h>
#include "ourhdr.h"

int
main(void)
{
    int          fd;
    pid_t        pid;
    char         buff[5];
    struct stat  statbuf;

    if ( (fd = open("templock", O_RDWR | O_CREAT | O_TRUNC,
                   FILE_MODE)) < 0)
        err_sys("open error");
    if (write(fd, "abcdef", 6) != 6)
        err_sys("write error");

    /* turn on set-group-ID and turn off group-execute */
    if (fstat(fd, &statbuf) < 0)
        err_sys("fstat error");
    if (fchmod(fd, (statbuf.st_mode & ~S_IXGRP) | S_ISGID) < 0)
        err_sys("fchmod error");

    TELL_WAIT();
    if ( (pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid > 0) { /* parent */
        /* write lock entire file */
        if (write_lock(fd, 0, SEEK_SET, 0) < 0)
            err_sys("write_lock error");
        TELL_CHILD(pid);

        if (waitpid(pid, NULL, 0) < 0)
            err_sys("waitpid error");
    } else { /* child */
        WAIT_PARENT(); /* wait for parent to set lock */

        set_fl(fd, O_NONBLOCK);

        /* first let's see what error we get if region is locked */
        if (read_lock(fd, 0, SEEK_SET, 0) != -1) /* no wait */
            err_sys("child: read_lock succeeded");
        printf("read_lock of already-locked region returns %d\n", errno);

        /* now try to read the mandatory locked file */
        if (lseek(fd, 0, SEEK_SET) == -1)
            err_sys("lseek error");
        if (read(fd, buff, 2) < 0)
```

```

        err_ret("read failed (mandatory locking works)");
    else
        printf("read OK (no mandatory locking), buff = %2.2s\n", buff);
    }
    exit(0);
}

```

此程序首先创建一个文件，并使强制性锁机制对其起作用。然后 fork 一个子进程。父进程对整个文件设置一把写锁，子进程则将该文件的描述符设置为非阻塞的，然后企图对该文件设置一把读锁，我们期望这会出错返回，并希望看到系统返回值是 EACCES 或 EAGAIN。接着，子进程将文件读、写位置调整到文件起点，并试图读该文件。如果系统提供强制性锁机制，则 read 应返回 EACCES 或 EAGAIN（因为该描述符是非阻塞的）。否则 read 返回所读的数据。在 SVR4 中运行此程序（该系统支持强制性锁机制），得到：

```

$ a.out
read_lock of already-locked region returns 13
read failed (mandatory locking works):No more processes

```

查看系统头文件或 intro(2) 手册页，可以看到错误 13 对应于 EACCES。从例子中还可以看到，在 read 出错返回信息部分中包含有 “No more processes”。这通常来自于 fork，表示已用完了进程表项。

若采用 4.3+BSD 系统，则得到：

```

$ a.out
read_lock of already-locked region returns 35
read OK (no mandatory locking),buff=ab

```

其中，errno35 对应于 EAGAIN。该系统不支持强制性锁。

实例

让我们回到本节的第一个问题：当两个人同时编辑同一个文件将会怎样呢？一般的 UNIX 文本编辑器并不使用记录锁，所以对此问题的回答仍然是：该文件的最后结果取决于写该文件的最后一个进程。（4.3+BSD 的 vi 编辑器确实有一个编译选择项使运行时建议性记录锁起作用，但是这一选择项并不是默认可用的。）即使我们在一个编辑器，例如 vi 中使用了建议性锁，可是这把锁并不能阻止其他用户使用另一个没有使用建议性记录锁的编辑器。

若系统提供强制性记录锁，那么可以修改常用的编辑器（如果有该编辑器的源代码）。如没有该编辑器的源代码，那么可以试一试下述方法。编写一个 vi 的前端程序。该程序立即调用 fork，然后父进程等待子进程终止，子进程打开在命令行中指定的文件，使强制性锁起作用，对整个文件设置一把写锁，然后运行 vi。在 vi 运行时，该文件是加了写锁的，所以其他用户不能修改它。当 vi 结束时，父进程从 wait 返回，此时自编的前端程序也就结束。本例中假定锁能跨越 exec，这正是前面所说的 SVR4 的情况（SVR4 是提供强制性锁的唯一系统）。

这种类型的前端程序是可以编写的，但却往往不能起作用。问题出在大多数编辑器（至少是 vi 和 ed）读它们的输入文件，然后关闭它。只要引用被编辑文件的描述符关闭了，那么加在该文件上的锁就被释放了。这意味着，在编辑器读了该文件的内容，然后关闭了它，那么锁也就不存在了。前端程序中没有任何方法可以阻止这一点。

第 16 章的数据库函数库使用了记录锁以阻止多个进程的并发存取。本章则提供了时间测量以观察记录锁对进程的影响。

12.4 流

流是系统V提供的构造内核设备驱动程序和网络协议包的一种通用方法，对流进行讨论的目的是理解下列各点：

- (1) 系统V的终端界面。
- (2) I/O多路复用中轮询函数的使用（见12.5.2节）。
- (3) 基于流管道和命名流管道的实现（见15.2和12.5.2节）。

流机制是由Dennis Ritchie 发展起来的〔Ritchie 1984〕,其目的是用通用、灵活的方法改写传统的字符 I/O系统并适应网络协议的需要，后来流机制被加入SVR3。SVR4则提供了对流（基于流的终端I/O系统）的全面支持。〔AT&T 1990d〕对SVR4实现进行了说明。

请注意不要将本章说明的流与标准I/O库（见5.2节）中使用的流相混淆。

流在用户进程和设备驱动程序之间提供了一条全双工通路。流无需和实际硬件设备直接对话——流也可以用作伪设备驱动程序。图12-5示出了一个简单流的基本结构。

在流首之下可以压入处理模块。这可以用 `ioctl`实现。图12-6示出了一个包含一个处理模块的流。各方框之间用两根带箭头的线连接，以强调流的全双工特征。

任一数的处理模块可以压入流。我们使用术语“压入”，是因为每一新模块总是插到流首之下，而将以前压入的模块下压。（这类似于后进、先出的栈。）图12-6标出了流的两侧，分别称为顺流（downstream）和逆流（upstream）。写到流首的数据将顺流而下传送。由设备驱动程序读到的数据则逆流向上传送。

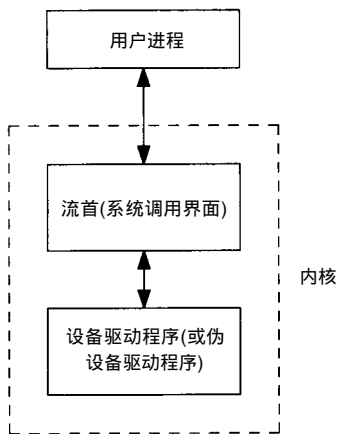


图12-5 一个简单流

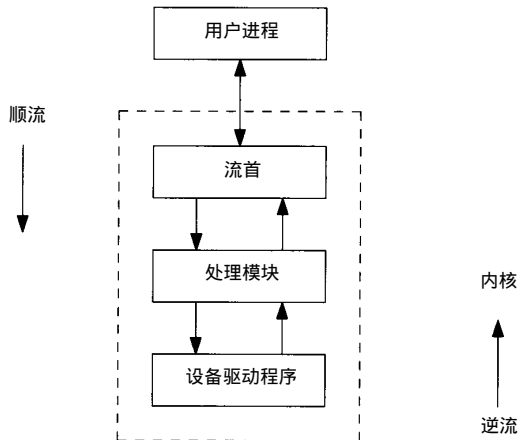


图12-6 具有处理模块的流

流模块是作为内核的一部分执行的，这类似于设备驱动程序，当构造内核时，流模块连编进入内核。大多数系统不允许将未连编进入内核的流模块压入流。

图11-2中示出了基于流的终端系统的一般结构。图中标出的“读、写函数”是流首。标注为“终端行程程”的框是一个流处理模块。该处理模块的实际名称是 `ldterm`。（各种流模块的手册页在〔AT&T 1990d〕的第7节和〔AT&T 1991〕的第7节中。）

用第3章中说明的函数存取流，它们是：`open`、`close`、`read`、`write`和`ioctl`。另外，在SVR3

内核中增加了3个支持流的新函数 (getmsg、putmsg和poll), 在SVR4中又加了两个处理流不同优先级波段消息的函数 (getpmsg和putpmsg)。本节将说明这些新函数, 为流打开的路径名通常在/dev目录之下。用ls-l查看设备名, 就能判断该设备是否为流设备。所有流设备都是字符特殊文件。

虽然某些有关流的文献暗示我们可以编写处理模块, 并将它们压入流中, 但是编写这些模块如同编写设备驱动程序一样, 需要专门的技术。

在流机制之前, 终端是用现存的clist机制处理的。(Bach [1986] 的10.3.1节和Leffler等 [1989] 的9.6节) 分别说明SVR2和4.3BSD中的clist 机制。将基于字符的设备添加到内核中通常涉及编写设备驱动程序, 将所有有关部分都安排在驱动程序中。对新设备的存取典型地通过原始设备进行, 这意味着每个用户的 read,write 都直通进入设备驱动程序。流机制使这种交互作用方式更加灵活, 条理清晰, 使得数据可以用流消息方式在流首和驱动程序之间传送, 并使任意数的中间处理模块可对数据进行操作。

12.4.1 流消息

流的所有输入和输出都基于消息。流首和用户进程使用 read、write、getmsg、getpmsg、putmsg和putpmsg交换消息。在流首、各处理模块和设备驱动程序之间, 消息可以顺流而下, 也可以逆流而上。

在用户进程和流首之间, 消息由下列几部分组成: 消息类型、可选择控制信息以及可选择的数据。表12-4列出了对应于write、putmsg和putpmsg的不同参数, 所产生的不同消息类型。控制信息和数据存放在strbuf结构中:

```
struct strbuf
{
    int maxlen; /* size of buffer */
    int len; /* number of bytes currently in buffer */
    char *buf; /* pointer to buffer */
};
```

表12-4 为write、putmsg和putpmsg产生的流消息的类型

函数	控制?	数据?	波段	标志	产生的消息类型
write	N/A	是	不使用	不使用	M_DATA (普通)
putmsg	否	否	不使用	0	无消息发送, 返回0
putmsg	否	是	不使用	0	M_DATA (普通)
putmsg	是	是或否	不使用	0	M_PROTO (普通)
putmsg	是	是或否	不使用	RS_HIPRI	M_PCPROTO (高优先级)
putmsg	否	是或否	不使用	RS_HIPRI	出错, EINVAL
putpmsg	是或否	是或否	0 ~ 255	0	出错, EINVAL
putpmsg	否	否	0 ~ 255	MSG_BAND	无消息发送, 返回0
putpmsg	否	是	0	MSG_BAND	M_DATA (普通)
putpmsg	否	是	1 ~ 255	MSG_BAND	M_DATA (优先级波段)
putpmsg	是	是或否	0	MSG_BAND	M_PROTO (普通)
putpmsg	是	是或否	1 ~ 255	MSG_BAND	M_PROTO (优先级波段)
putpmsg	是	是或否	0	MSG_HIPRI	M_PCPROTO (高优先级)
putpmsg	否	是或否	0	MSG_HIPRI	出错, EINVAL
putpmsg	是或否	是或否	非0	MSG_HIPRI	出错, EINVAL

当用putmsg或putpmsg发送消息时，len指定缓存中数据的字节数。当用getmsg或getpmsg接收消息时，maxlen指定缓存长度（使内核不会溢出缓存），而len则由内核设置，说明存放在缓存中的数据量。0长消息是允许的，len为-1说明没有控制信息或数据。

为什么需要传送控制信息和数据两者呢？提供这两者使我们可以实现用户进程和流之间的服务界面。Olander, McGrath和Israel [1986]说明了系统V服务界面的原先实现。AT&T[1990d]第5章详细说明了服务界面，还使用了一个简单的实例。可能最为人了解的服务界面是系统V的传输层界面(TLI)，它提供了网络系统界面，Stevens [1990]第7章对此进行了说明。

控制信息的另一个例子是发送一个无连接的网络消息（数据报）。为了发送该消息，需要说明消息的内容（数据）和该消息的目的地址（控制信息）。如果不能将数据和控制一起发送，那么就要某种专门设计的方案。例如，可以用ioctl说明地址，然后用write发送数据。另一种技术可能要求：地址占用数据的前N个字节，用write写数据。将控制信息与数据分开，并且提供处理两者的函数（putmsg和getmsg）是处理这种问题的较清晰的方法。

有约25种不同类型的消息，但是只有少数几种用于用户进程和流首之间。其余的则只在内核中顺流、逆流传送。（对于编写流处理模块的人员而言，这些消息是非常有用的，但是对编写用户级代码的人员而言，则可忽略它们。）在我们所使用的函数（read、write、getmsg、getpmsg、putmsg和putpmsg）中，只涉及三种消息类型，它们是：

- M_DATA (I/O的用户数据)。
- M_PROTO (协议控制信息)。
- M_PCPROTO (高优先级协议控制信息)。

流中的消息都有一个排队优先级：

- 高优先级消息（最高优先级）。
- 优先波段消息。
- 普通消息（最低优先级）。

普通消息是优先波段为0的消息。优先波段消息的波段可在1~255之间，波段愈高，优先级也愈高。

每个流模块有两个输入队列。一个接收来自它上面模块的消息（这种消息从流首向驱动程序顺流传送）。另一个接收来自它下面模块的消息（这种消息从驱动程序向流首逆流传送）。在输入队列中的消息按优先级从高到低排列。表12-4列出了针对write、putmsg和putpmsg的不同参数，产生不同优先级的消息。

有一些消息我们未加考虑。例如，若流首从它下面接收到M_SIG消息，则产生一信号。这种方法用于终端行规程模块向相关前台进程组发送终端产生的信号

12.4.2 putmsg和putpmsg函数

putmsg和putpmsg函数用于将流消息（控制信息或数据，或两者）写至流中。这两个函数的区别是后者允许对消息指定一个优先波段。

```
#include <stropts.h>
int putmsg(int fildes, const struct strbuf *ptr,
           const struct strbuf *data_ptr, int flag);
int putpmsg(int fildes, const struct strbuf *ptr,
            const struct strbuf *data_ptr, int band, int flag);
```

两个函数返回：若成功则为0，若出错则为-1

对流也可以使用 `write` 函数，它等效于不带任何控制信息，`flag` 为 0 的 `putmsg`。

这两个函数可以产生三种不同优先级的消息：普通、优先波段和高优先级。表 12-4 详细列出了这两个函数中几个参数的各种可能组合，以及所产生的不同类型的消息。

在表 12-4 中，消息控制列中的“否”对应于空 `ctlptr` 参数，或 `ctlptr->len` 为 -1。该列中的“是”对应于 `ctlptr` 非空，以及 `ctlptr->len` 大于等于 0。这些说明同样适用于消息的数据部分（用 `dataptr` 代替 `ctlptr`）。

12.4.3 流 `ioctl` 操作

3.14 节曾提到过 `ioctl` 函数，它能做其他 I/O 函数不能处理的事情。流系统中继续采用了该函数。

在 SVR4 中，使用 `ioctl` 可对流执行 29 种不同的操作。关于这些操作的说明请见 `streamio(7)` 手册页 [AT&T1990d]，头文件 `<stropts.h>` 应包括在使用这些操作的 C 代码中。`ioctl` 的第二个参数 `request` 说明执行 29 个操作中的哪一个。所有 `request` 都以 `I_` 开始。第三个参数则与 `request` 有关。有时第三个参数是一个整型值，有时它是指向一个整型或一个数据结构的指针。

实例——`isastream` 函数

有时需要确定一个描述符是否引用一个流。这与调用 `isatty` 函数来确定一个描述符是否引用一个终端设备相类似（见 11.9 节）。SVR4 提供 `isastream` 函数。

```
int isastream(int fildes);
```

返回：若是流返回 1，否则返回 0

（由于某种原因，SVR4 的设计者忘记将此函数的原型放在头文件中，所以不能为此函数写一条 `#include` 语句。）

与 `isatty` 类似，它通常是用一个只对流设备才有效的 `ioctl` 函数来进行测试的。程序 12-8 是该函数的一种可能的实现。它使用 `I_CANPUT` `ioctl` 来测试由第三个参数说明的优先波段是否可写。如果该 `ioctl` 执行成功，则它对所涉及的流并未作任何改变。

程序 12-8 检查描述符是否引用流设备

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

int
isastream(int fd)
{
    return(ioctl(fd, I_CANPUT, 0) != -1);
}
```

程序 12-9 可用于测试此函数。

程序 12-9 测试 `isastream` 函数

```
#include <sys/types.h>
#include <sys/fcntl.h>
#include "ourhdr.h"

int
```

```

main(int argc, char *argv[])
{
    int    i, fd;

    for (i = 1; i < argc; i++) {
        printf("%s: ", argv[i]);
        if ( (fd = open(argv[i], O_RDONLY)) < 0) {
            err_ret("%s: can't open", argv[i]);
            continue;
        }

        if (isastream(fd) == 0)
            err_ret("%s: not a stream", argv[i]);
        else
            err_msg("%s: streams device", argv[i]);
    }
    exit(0);
}

```

运行此程序，得到很多由*ioctl*函数返回的出错信息。

```

$ a.out /dev/tty /dev/vidadm /dev/null /etc/motd
/dev/tty:/dev/tty:streams device
/dev/vidadm:/dev/vidadm:not a stream:Invalid argument
/dev/null:/dev/null:not a stream:No suck device
/etc/motd: /etc/motd:not a stream:Not a typewriter

```

*/dev/tty*在SVR之下是个流设备，这与我们所期望的一致。*/dev/vidadm*不是一个流设备，但是它是支持其他*ioctl*请求的字符特殊文件。对于不知道这种*ioctl*请求的设备，它返回EINVAL。*/dev/null*是一种不支持任何*ioctl*操作的字符特殊文件，所以*ioctl*返回ENODEV。最后，*/etc/motd*是一个普通文件，而不是字符特殊文件，所以返回ENOTTY（这是在这种情况下的典型返回值）。

ENOTTY (Not a typewriter) 是个历史产物，当*ioctl*企图对并不引用字符特殊设备的描述符进行操作时，UNIX内核都返回ENOTTY。

实例

如果*ioctl*的参数*request*是I_LIST，则系统返回该流上所有模块的名字，包括最顶端的驱动程序。（指明最顶端的原因是：在多路转接驱动程序的情况下，有多个驱动程序。AT&T [1990d]）第10章讨论了多路转接驱动程序的细节。）其第三个参数应当是指向*str_list*结构的指针。

```

struct str_list {
    int                sl_nmods;    /* number of entries in array */
    struct str_mlist  *sl_modlist; /* ptr to first element of array */
};

```

应将*sl_modlist*设置为指向*str_mlist*结构数组的第一个元素，将*sl_nmods*设置为该数组中的项数。

```

struct str_mlist {
    char l_name[FMNAMESZ+1]; /* null terminated module name */
};

```

常数FMNAMESZ定义在头文件<sys/conf.h>中，其值常常是8。*l_name*的实际长度是

FMNAMESZ+1，增加1个字节是为了存放null终止符。

如果ioctl的第三个参数是0，则该函数返回的是模块数，而不是模块名。我们将先用这种ioctl调用确定模块数，然后再分配所要求的str_mlist结构数。

程序12-10例示了I_LIST操作。由ioctl返回的名字列表并不对模块和驱动程序进行区分，因为该列表的最后一项是处于流底部的驱动程序，所以在打印时将其标明为驱动程序。

程序12-10 打印流中的模块名

```
#include <sys/conf.h>
#include <sys/types.h>
#include <fcntl.h>
#include <stropts.h>
#include "ourhdr.h"

int
main(int argc, char *argv[])
{
    int          fd, i, nmods;
    struct str_list list;

    if (argc != 2)
        err_quit("usage: a.out <pathname>");
    if ( (fd = open(argv[1], O_RDONLY)) < 0)
        err_sys("can't open %s", argv[1]);
    if (isastream(fd) == 0)
        err_quit("%s is not a stream", argv[1]);

        /* fetch number of modules */
    if ( (nmods = ioctl(fd, I_LIST, (void *) 0)) < 0)
        err_sys("I_LIST error for nmods");
    printf("#modules = %d\n", nmods);

        /* allocate storage for all the module names */
    list.sl_modlist = calloc(nmods, sizeof(struct str_mlist));
    if (list.sl_modlist == NULL)
        err_sys("calloc error");
    list.sl_nmods = nmods;

        /* and fetch the module names */
    if (ioctl(fd, I_LIST, &list) < 0)
        err_sys("I_LIST error for list");

        /* print the module names */
    for (i = 1; i <= nmods; i++)
        printf(" %s: %s\n", (i == nmods) ? "driver" : "module",
              list.sl_modlist++);

    exit(0);
}
```

```
$ who
stevens  console  Sep 25 06:12
stevens  pts001    Oct 12 07:12
$ a.out /dev/pts001
#modules= 4
  module: ttcompat
  module: ldterm
  module: ptem
  driver: pts
$ a.out /dev/console
```

```
#modules= 5
  module: ttcompat
  module: ldterm
  module: ansi
  module: char
  driver: cmux
```

在这两种情形中，顶上的两个流模块都是一样的（ttcompat和ldterm），但是余下的模块和驱动程序则不同。第19章将说明伪终端的情形。

12.4.4 write至流设备

在表12-4中可以看到write至流设备产生一个M_DATA消息。一般而言，这确实如此，但是也还有一些情况需要考虑。首先，流中顶部的一个处理模块规定了可顺流传送的最小、最大数据包长度（无法查询该模块中规定的这些值）。如果write的数据长度超过最大值，则流首将这一数据分解成最大长度的若干数据包。最后一个数据包的长度则小于最大值。

接着要考虑的是：如果向流write 0个字节，又将如何呢？除非流涉及管道或FIFO，否则就顺流发送0长消息。对于管道FIFO，为与以前版本兼容，系统的默认处理方式是忽略0长write。可以用ioctl设置实现管道和FIFO的流写方式以更改这种默认处理方式。

12.4.5 写方式

可以用ioctl取得和设置一个流的写方式。如果将request设置为L_GWROPT，第三个参数为指向一个整型变量的指针，则该流的当前写方式就在该整型量中返回。如果将request设置为L_SWROPT，第三个参数是一个整型值，则其值就成为该流新的写方式。如同处理文件描述符标志和文件状态标志（见3.13节）一样，总是应当先取当前写方式值，然后修改它，而不只是将写方式设置为某个绝对值（很可能要关闭某些原来打开的位）。

目前，只定义了两个写方式值。

- SNDZERO 对管道和FIFO的0长写会造成顺流传送一个0长消息。按系统默认，0长写不发送消息。

- SNDPIPE 在流上已出错后，若调用write或putmsg，则向调用进程发送SIGPIPE信号。

流也有读方式，我们先说明getmsg和getpmsg函数，然后再说明读方式。

12.4.6 getmsg和getpmsg函数

```
#include <stropts.h>

int getmsg(int filedes, struct strbuf * ctlptr,
           struct strbuf * dataptr, int * flagptr);

int getpmsg(int filedes, struct strbuf * ctlptr,
            struct strbuf * dataptr, int * bandptr, int * flagptr);
```

两个函数返回：若成功则为非负值，若出错则为 -1

注意，flagptr和bandptr是指向整型的指针。在调用之前，这两个指针所指向的整型单元中应设置成所希望的消息类型，在返回时，此整型量设置为所读到的消息的类型。

如果 *flagptr* 指向的整型单元的值是 0，则 *getmsg* 返回流首读队列中的下一个消息。如果下一个消息是高优先级消息，则在返回时，*flagptr* 所指向的整型单元设置为 *RS_HIPRI*。如果希望只接收高优先级消息，则在调用 *getmsg* 之前必须将 *flagptr* 所指向的整型单元设置为 *RS_HIPRI*。

getpmsg 使用了一个不同的常数集。并且它使用 *bandptr* 指明特定的优先波段。

这两个函数有很多条件来确定返回给调用者何种消息：(a) *flagptr* 和 *bandptr* 所指向的值，(b) 流队列中消息的类型，(c) 是否指明非空 *dataptr* 和 *ctlptr*，(d) *ctlptr->maxlen* 和 *dataptr->maxlen* 的值。对使用 *getmsg* 而言，并不需要了解所有这些细节。如欲了解这些细节请参阅 *getmsg* (2) 手册页。

12.4.7 读方式

如果 *read* 流设备会发生些什么呢？有两个潜在的问题：(1) 如果读到流中消息的记录边界将会怎样？(2) 如果调用 *read*，而流中下一个消息有控制信息又将如何？对第一种情况的默认处理方式被称为字节流方式。在这种方式中，*read* 从流中取数据直至满足了要求，或者已经没有数据。在这种方式中，忽略流中消息的边界。对第二种情况的默认处理是，如果在队列的前端有控制消息，则 *read* 出错返回。可以改变这两种默认处理方式。

调用 *ioctl* 时，若将 *request* 设置为 *I_GRDOPT*，第三个参数又是指向一个整型单元的指针，则对该流的当前读方式在该整型单元中返回。如果将 *request* 设置为 *I_SRDOPT*，第三个参数是整型值，则该流的读方式设置为该值。读方式值有下列三个：

- *RNORM* 普通，字节流方式（与上面说明的相同）。这是默认方式。
- *RMSGN* 消息不删除方式。读从流中取数据直至读到所要求的字节数，或者到达消息边界。如果某次读只用了消息的一部分，则其余下部分仍留在流中，以供下一个读取。
- *RMSGD* 消息删除方式。这与不删除方式的区别是，如果某次读只用消息的一部分。则其余下部分就被删除，不再使用。

在读方式中还可指定另外三个常数，以便设置在读到流中包含协议信息的信息时 *read* 的处理方法：

- *RPROTNORM* 协议-普通方式。*read* 出错返回，*errno* 设置为 *EBADMSG*。这是默认方式。
- *RPROTDAT* 协议-数据方式。*read* 将控制部分作为数据返回给调用者。
- *RPROTDIS* 协议-删除方式。*read* 删除消息中的控制信息，但是返回消息中的数据。

实例

程序 12-11 是在程序 3-3 的基础上改写的，它用 *getmsg* 代替了 *read*。如果在 *SVR4* 之下（其管道和终端都是用流实现的）运行此程序则得：

```
$ echo hello,world | a.out          使用流实现要求的管道
flag=0,ctl.len=-1,dat.len=13
hello,world
flag=0,ctl.len=0,dat.len=0        流挂断
$ a.out                            使用流实现要求的终端
this is line 1
flag=0,ctl.len=-1,dat.len=15
this is line 1
and line 2
flag=0,ctl.len=-1,dat.len=11
```

```

and line 2
^D                               键入自定义的终端EOF字符
flag=0,ctl.len=-1,dat.len=0      文件结尾与挂断不相同
$ a.out < /etc/motd
getmsg error:Not a stream device

```

当管道被关闭时（当echo终止时），它对程序12-11表现为一个流挂断——控制长度和数据长度都设置为0。（14.2节将讨论管道。）但是对于终端，键入文件结束字符，只使返回的数据长度为0。这与挂断并不相同。如所预料的一样，将标准输入重新定向到一个非流设备，getmsg出错返回。

程序12-11 用getmsg将标准输入复制到标准输出

```

#include <stropts.h>
#include "ourhdr.h"

#define BUFFSIZE 8192

int
main(void)
{
    int n, flag;
    char ctlbuf[BUFFSIZE], datbuf[BUFFSIZE];
    struct strbuf ctl, dat;

    ctl.buf = ctlbuf;
    ctl.maxlen = BUFFSIZE;
    dat.buf = datbuf;
    dat.maxlen = BUFFSIZE;
    for ( ; ; ) {
        flag = 0; /* return any message */
        if ( (n = getmsg(STDIN_FILENO, &ctl, &dat, &flag)) < 0)
            err_sys("getmsg error");
        fprintf(stderr, "flag = %d, ctl.len = %d, dat.len = %d\n",
                 flag, ctl.len, dat.len);
        if (dat.len == 0)
            exit(0);
        else if (dat.len > 0)
            if (write(STDOUT_FILENO, dat.buf, dat.len) != dat.len)
                err_sys("write error");
    }
}

```

12.5 I/O多路转接

当从一个描述符读，然后又写到另一个描述符时，可以在下列形式的循环中使用阻塞 I/O：

```

while ( (n=read(STDIN_FILENO, buf, BUFSIZ) ) > 0)
    if (write(STDOUT_FILENO, buf, n) != n)
        err_sys("write error");

```

这种形式的阻塞 I/O 到处可见。但是如果必须读两个描述符又将如何呢？如果仍旧使用阻塞 I/O，那么就可能长时间阻塞在一个描述符上，而另一个描述符虽有很多数据却不能得到及时处理。所以为了处理这种情况显然需要另一种不同的技术。

让我们概略地观察一个调制解调器拨号程序的工作情况（该程序将在第 18 章中介绍）。该程序读终端（标准输入），将所得数据写到调制解调器上；同时读调制解调器，将所得数据写到终端上（标准输出）。图 12-7 显示这种工作情况。

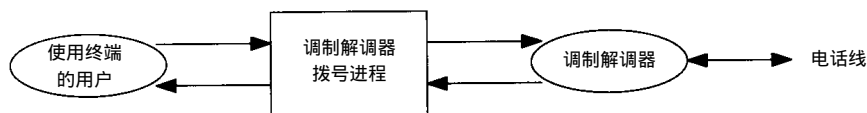


图12-7 调制解调器拨号程序概观

执行这段程序的进程有两个输入，两个输出。如果对这两个输入都使用阻塞 read，那么就可能在某一个输入上长期阻塞，而另一个输入的数据则被丢失。

处理这种特殊问题的一种方法是：设置两个进程，每个进程处理一条数据通路。图 12-8 中显示了这种安排。

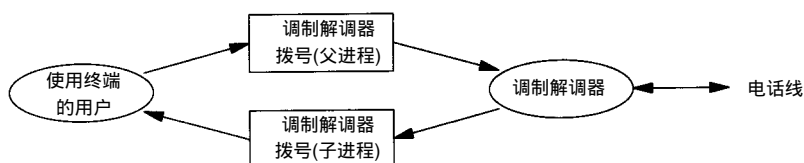


图12-8 使用两个进程实现调制解调器拨号程序

如果使用两个进程，则可使每个进程都执行阻塞 read。但是也产生了这两个进程间相互配合问题。如果子进程接收到文件结束符（由于电话线的一端已经挂断，使调制解调器也挂断），那么该子进程终止，然后父进程接收到 SIGCHLD 信号。但是，如若父进程终止（用户在终端上键入了文件结束符），那么父进程应通知子进程停止工作。为此可以使用一个信号（例如 SIGUSR1）。这使程序变得更加复杂。

另一个方式是仍旧使用一个进程执行该程序，但调用非阻塞 I/O 读取数据，其基本思想是：将两个输入描述符都设置为非阻塞的，对第一个描述符发一个 read。如果该输入上有数据，则读数据并处理它。如果无数据可读，则 read 立即返回。然后对第二个描述符作用样的处理。在此之后，等待若干秒再读第一个描述符。这种形式的循环称为轮询。这种方法的不足之处是浪费 CPU 时间。大多数时间实际上是无数据可读，但是仍不断反复执行 read，这浪费了 CPU 时间。在每次循环后要等多长时间再执行下一轮循环也很难确定。轮询技术在支持非阻塞 I/O 的系统上都可使用，但是在多任务系统中应当避免使用。

还有一种技术称之为异步 I/O (asynchronous I/O)。其基本思想是进程告诉内核，当一个描述符已准备好可以进行 I/O 时，用一个信号通知它。这种技术有两个问题。第一个是并非所有系统都支持这种机制（现在它还不是 POSIX 的组成部分，可能将来会是）。SVR4 为此技术提供 SIGPOLL 信号，但是仅当描述符引用流设备时，此信号才能工作。4.3+BSD 有一个类似的信号 SIGIO，但也有类似的限制——仅当描述符引用终端设备或网络时才能工作。这种技术的第二个问题是，这种信号对每个进程而言只有 1 个。如果使该信号对两个描述符都起作用，那么在接到此信号时进程无法判别是哪一个描述符已准备好可以进行 I/O。为了确定是哪一个描述符已准备好，仍需将这两个描述符都设置为非阻塞的，并顺序试执行 I/O。12.6 节将简要说明异步 I/O。

一种比较好的技术是使用 I/O 多路转接 (I/O multiplexing)。其基本思想是：先构造一张有关描述符的表，然后调用一个函数，它要到这些描述符中的一个已准备好进行 I/O 时才返回。在返回时，它告诉进程哪一个描述符已准备好可以进行 I/O。

I/O多路转接至今还不是POSIX的组成部分。SVR4和4.3+BSD都提供select函数以执行I/O多路转接。poll函数只由SVR4提供。SVR4实际上用poll实现select。

I/O多路转接在4.2+BSD中是用select函数提供的。虽然该函数主要用于终端I/O和网络I/O，但它对其他描述符同样是起作用的。SVR3在增加流机制时增加了poll函数。但在SVR4之前，poll只对流设备起作用。SVR4支持对任一描述符起作用的poll。

select和poll的可中断性

中断的系统调用的自动再启动是由4.2+BSD引进的（见10.5节），但当时select函数是不再起动的。这种特性延续到4.3+BSD，即使指定了SA_RESTART也是为此。但是，在SVR4之下，如果指定了SA_RESTART，那么select和poll也是自动再起动的。为了将软件移植到SVR4时阻止这一点，如果信号可能中断select或poll，则总是使用signal_intr函数（见程序10-13）。

12.5.1 select函数

select函数使我们在SVR4和4.3+BSD之下可以执行I/O多路转接，传向select的参数告诉内核：

(1) 我们所关心的描述符。

(2) 对于每个描述符我们所关心的条件（是否读一个给定的描述符？是否想写一个给定的描述符？是否关心一个描述符的异常条件？）。

(3) 希望等待多长时间（可以永远等待，等待一个固定量时间，或完全不等待）。

从select返回时，内核告诉我们：

(1) 已准备好的描述符的数量。

(2) 哪一个描述符已准备好读、写或异常条件。

使用这种返回值，就可调用相应的I/O函数（一般是read或write），并且确知该函数不会阻塞。

```
#include <sys/types.h> /* fd_set data type */
#include <sys/time.h> /* struct timeval */
#include <unistd.h> /* function prototype might be here */

int select (int maxfdp1, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
            struct timeval *tvptr);
```

返回：准备就绪的描述符数，若超时则为0，若出错则为-1

先说明最后一个参数，它指定愿意等待的时间。

```
struct timeval{
    long tv_sec; /* seconds */
    long tv_usec; /* and microseconds */
};
```

有三种情况：

- tvptr = NULL

永远等待。如果捕捉到一个信号则中断此无限期等待。当所指定的描述符中的一个已准备好或捕捉到一个信号则返回。如果捕捉到一个信号，则select返回-1，errno设置为EINTR。

- tvptr->tv_sec = 0 && tvptr->tv_usec = 0

完全不等待。测试所有指定的描述符并立即返回。这是得到多个描述符的状态而不阻塞 `select` 函数的轮询方法。

- `tvptr->tv_sec != 0 || tvptr->tv_usec != 0`

等待指定的秒数和微秒数。当指定的描述符之一已准备好，或当指定的时间值已经超过时立即返回。如果在超时时还没有一个描述符准备好，则返回值是 0，(如果系统不提供微秒分辨率，则 `tvptr->tv_usec` 值取整到最近的支持值。) 与第一种情况一样，这种等待可被捕捉到的信号中断。

中间三个参数 `readfds`、`writefds` 和 `exceptfds` 是指向描述符集的指针。这三个描述符集说明了我们关心的可读、可写或处于异常条件的各个描述符。每个描述符集存放在一个 `fd_set` 数据类型中。这种数据类型的实现可见图 12-9，它为每一可能的描述符保持了一位。

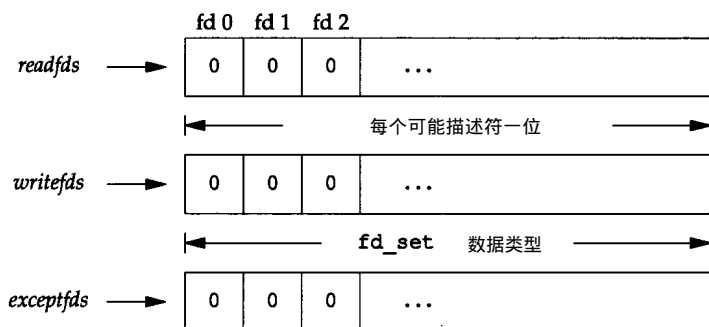


图12-9 对 `select` 指定读、写和异常条件描述符

对 `fd_set` 数据类型可以进行的处理是：(a) 分配一个这种类型的变量，(b) 将这种类型的一个变量赋与同类型的另一个变量，或 (c) 对于这种类型的变量使用下列四个宏：

```
FD_ZERO(fd_set *fdset);          /* clear all bits in fdset */
FD_SET(int fd, fd_set *fdset);   /* turn on bit for fd in fdset */
FD_CLR(int fd, fd_set *fdset);   /* turn off bit for fd in fdset */
FD_ISSET(int fd, fd_set *fdset); /* test bit for fd in fdset */
```

以下列方式说明了一个描述符集后：

```
fd_set rset;
int fd;
```

必须用 `FD_ZERO` 清除其所有位：

```
FD_ZERO (&rset);
```

然后在其中设置我们关心的各位：

```
FD_SET (fd, &rset);
FD_SET (STDIN_FILENO, &rset);
```

从 `select` 返回时，用 `FD_ISSET` 测试该集的一个给定位是否仍旧设置：

```
if (FD_ISSET(fd, &rset)){
    ...
}
```

select中间三个参数中的任意一个（或全部）可以是空指针，这表示对相应条件并不关心。如果所有三个指针都是空指针，则select提供了较sleep更精确的计时器（回忆10.19节，sleep等待整数秒，而对于select，其等待的时间可以小于1秒；其实际分辨率取决于系统时钟。）习题12.6给出了这样一个函数。

select第一个参数`maxfdp1`的意思是“最大fd加1（max fd plus 1）”。在三个描述符集中找出最高描述符编号值，然后加1，这就是第一个参数值。也可将第一个参数设置为`FD_SETSIZE`，这是一个`<sys/types.h>`中的常数，它说明了最大的描述符数（经常是256或1024）。但是对大多数应用程序而言，此值太大了。确实，大多数应用程序只应用3~10个描述符。如果将第三个参数设置为最高描述符编号值加1，内核就只需在此范围内寻找打开的位，而不必在数百位的大范围内搜索。

例如，若编写下列代码：

```
fd_set readset, writeset;

FD_ZERO(&readset);
FD_ZERO(&writeset);

FD_SET(0, &readset);
FD_SET(3, &readset);
FD_SET(1, &writeset);
FD_SET(2, &writeset);

select (4, &readset, &writeset, NULL, NULL);
```

然后，图12-10显示了这两个描述符集的情况。

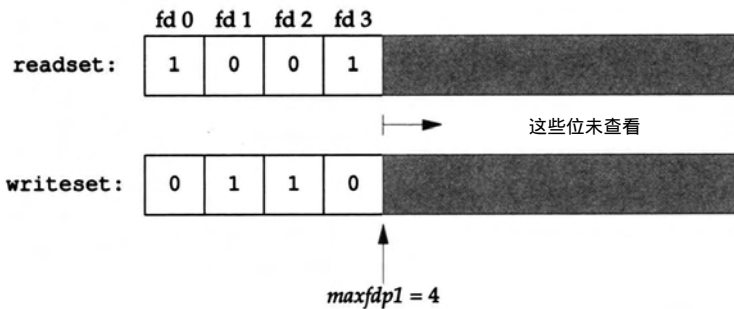


图12-10 select的样本描述符集

因为描述符编号从0开始，所以要在最大描述符编号值上加1。第一个参数实际上是要检查的描述符数（从描述符0开始）。

select有三个可能的返回值。

(1) 返回值 - 1表示出错。这是可能发生的，例如在所指定的描述符都没有准备好时捕捉到一个信号。

(2) 返回值0表示没有描述符准备好。若指定的描述符都没有准备好，而且指定的时间已经超过，则发生这种情况。

(3) 返回一个正值说明了已经准备好的描述符数，在这种情况下，三个描述符集中仍旧打开的位是对应于已准备好的描述符位。

注意，除非返回正值，否则在返回后检查描述符集是没有意义的。若捕捉到信号或计时器超时，那么描述符集的值取决于实现。确实，若计时器超时，4.3+BSD并不改变描述符集，而SVR4则清除描述符集。

在SVR4和BSD的select实现之间，有另一些差异。BSD系统总是返回每一个集中准备就绪的描述符数之和。若两个集中的同一描述符准备就绪（例如，读集和写集），则该描述符计两次。不幸，SVR4更改了这一点，若同一描述符在多个集中准备就绪，该描述符只计一次。这再一次显示了我们将会碰到的问题，直至POSIX标准化了select这样的函数才能解决此问题。

对于“准备好”的意思要作一些更具体的说明：

- (1) 若对读集（*readfds*）中的一个描述符的read不会阻塞，则此描述符是准备好的。
- (2) 若对写集（*writefds*）中的一个描述符的write不会阻塞，则此描述符是准备好的。
- (3) 若对异常条件集（*exceptfds*）中的一个描述符有一个未决异常条件，则此描述符是准备好的。现在，异常条件包括：(a)在网络连接上到达指定波特率外的数据，或者(b)在处于数据包方式的伪终端上发生了某些条件。（Stevens [1990] 的15.10节中说明了这种条件。）

应当理解一个描述符阻塞与否并不影响select是否阻塞。也就是说，如果希望读一个非阻塞描述符，并且以超时值为5秒调用select，则select最多阻塞5秒。相类似，如果指定一个无限的超时值，则select阻塞到对该描述符数据准备好，或捕捉到一个信号。

如果在一个描述符上碰到了文件结束，则select认为该描述符是可读的。然后调用read，它返回0，这是UNIX指示到达文件结尾处的方法。（很多人错误地认为，当到达文件结尾处时，select会指示一个异常条件。）

12.5.2 poll函数

SVR4的poll函数类似于select，但是其调用形式则有所不同。我们将会看到，poll与流系统紧紧相关，虽然在SVR4中，可以对任一描述符都使用该函数。

```
#include <stropts.h>
#include <poll.h>

int poll(struct pollfd farray[], unsigned long nfd, int timeout);
```

返回：准备就绪的描述符数，若超时则为0，若出错则为-1

与select不同，poll不是为每个条件构造一个描述符集，而是构造一个pollfd结构数组，每个数组元素指定一个描述符编号以及对其所关心的条件。

```
struct pollfd {
    int     fd;           /* file descriptor to check, or < 0 to ignore */
    short  events;       /* events of interest on fd */
    short  revents;      /* events that occurred on fd */
};
```

*farray*数组中的元素数由*nfd*说明。

由于某种未知的原因，SVR3说明*nfd*的类型为unsigned long，这似乎是太大了。在SVR4手册poll的原型中，第二个参数的数据类型为size_t，但在<poll.h>包含的实际原型中，第二个参数的数据类型仍说明为unsigned long。

SVR4的SVID [AT&T1989] 说明poll的第一个参数是struct pollfd *fdarray* [],而SVR4手册页 [AT&T 1990 d] 则说明该参数为struct pollfd **fdarray*。在C语言中,这两种说明是等价的。我们使用第一种说明以重申 *fdarray*指向一个结构数组,而不是指向单个结构的指针。

应将events成员设置为表12-5中所示值的一个或几个。通过这些值告诉内核我们对该描述符关心的是什么。返回时,内核设置revents成员,以说明对该描述符发生了什么事情。(注意,poll没有更改events成员,这与select不同,select修改其参数以指示哪一个描述符已准备好了。)

表12-5 poll的events和revents标志

名称	对events的输入	从revents得到的结果	说明
POLLIN	•	•	可读除高优先级外的数据,不阻塞
POLLRDNORM	•	•	可读普通(优先波段0)数据,不阻塞
POLLRDBAND	•	•	可读0优先波段数据,不阻塞
POLLPRI	•	•	可读高优先级数据,不阻塞
POLLOUT	•	•	可与普通数据,不阻塞
POLLWRNORM	•	•	与POLLOUT相同
POLLWRBAND	•	•	可写非0优先波段数据,不阻塞
POLLERR		•	已出错
POLLHUP		•	已挂起
POLLNVAL		•	此描述符并不引用一打开文件

表12-5中头四行测试可读性,接着三行测试可写性,最后三行则是异常条件。最后三行是由内核在返回时设置的。即使在events字段中没有指定这三个值,如果相应条件发生,则在revents中也返回它们。

当一个描述符被挂断后(POLLHUP),就不能再写向该描述符。但是仍可能从该描述符读取到数据。

poll的最后一个参数说明我们想要等待多少时间。如同select一样,有三种不同的情形:

- *timeout* == INFTIM 永远等待。常数INFTIM定义在<stropts.h>,其值通常是 - 1。当所指定的描述符中的一个已准备好,或捕捉到一个信号则返回。如果捕捉到一个信号,则poll返回 - 1,errno设置为EINTR。

- *timeout* == 0 不等待。测试所有描述符并立即返回。这是得到很多个描述符的状态而不阻塞poll函数的轮询方法。

- *timeout* > 0 等待*timeout*毫秒。当指定的描述符之一已准备好,或指定的时间值已超过时立即返回。如果已超时但是还没有一个描述符准备好,则返回值是 0。(如果系统不提供毫秒分辨率,则*timeout*值取整到最近的支持值。)

应当理解文件结束与挂断之间的区别。如果正在终端输入数据,并键入文件结束字符,POLLIN被打开,于是就可读文件结束指示(read返回0)。POLLHUP在revents中没有打开。如果读调制解调器,并且电话线已挂断,则在revents中将接到POLLHUP。

与select一样,不论一个描述符是否阻塞,并不影响poll是否阻塞。

12.6 异步 I/O

使用 `select` 和 `poll` 可以实现异步 I/O。关于描述符的状态,系统并不主动告诉我们任何信息,我们需要主动地进行查询(调用 `select` 或 `poll`)。如在第 10 章中所述,信号机构提供一种异步形式的通知某种事件已发生的方法。SVR4 和 4.3+BSD 提供了使用一个信号(在 SVR4 中是 `SIGPOLL`, 在 4.3+BSD 中是 `SIGIO`) 的异步 I/O 方法,该信号通知进程,对某个描述符所关心的某个事件已经发生。

我们已了解到在 SVR4 中 `select` 和 `poll` 对任意描述符都能工作。在 4BSD 中, `select` 对任意描述符都能工作。但是关于异步 I/O 却有限制。在 SVR4 中,异步 I/O 只对设备起作用。在 4.3+BSD 中,异步 I/O 只对终端和网络起作用。

SVR4 和 4.3+BSD 所支持的异步 I/O 的一个限制是每个进程只有一个信号。如果要对几个描述符进行异步 I/O,那么在进程接收到该信号时并不知道这一信号对应于哪一个描述符。

12.6.1 SVR4

在系统 V 中,异步 I/O 是流系统的一部分。它只对流设备起作用。SVR4 异步 I/O 信号是 `SIGPOLL`。

为了对一个流设备启动异步 I/O,需要调用 `ioctl`,而其第二个参数(`request`)则为 `I_SETSIG`。第三个参数则是由表 12-6 中一个或多个常数构成的整型值。这些常数在 `<stropts.h>` 中定义。

表 12-6 产生 `SIGPOLL` 信号的条件

常 数	说 明
<code>S_INPUT</code>	非高优先级的消息已到达
<code>S_RDNORM</code>	—普通消息已到达
<code>S_RDBAND</code>	—0 优先波段消息已到达
<code>S_BANDURG</code>	若此常数说明为 <code>S_RDBAND</code> , 则当一非 0 优先波段消息到达时产生 <code>SIGURG</code> 信号而非 <code>SIGPOLL</code>
<code>S_HIPRI</code>	—高优先级消息已到达
<code>S_OUTPUT</code>	写队列不再满
<code>S_WRNORM</code>	与 <code>S_OUTPUT</code> 一样
<code>S_WRBAND</code>	可发送一非 0 优先波段消息
<code>S_MSG</code>	包含 <code>SIGPOLL</code> 信号的流信号消息已到达
<code>S_ERROR</code>	<code>M_ERROR</code> 消息已到达
<code>S_HANGUP</code>	<code>M_HANGUP</code> 消息已到达

表 12-6 中“已到达”的意思是“已到达流首的读队列”。

除了调用 `ioctl` 以说明产生 `SIGPOLL` 信号的条件,也应为该信号建立一个信号处理程序。回忆表 10-1,对于 `SIGPOLL` 的默认动作是终止该进程,所以应在调用 `ioctl` 之前建立信号处理程序。

12.6.2 4.3+BSD

在 4.3+BSD 中,异步 I/O 是两个信号 `SIGIO` 和 `SIGURG` 的组合。前者是通用异步 I/O 信号,后者则只被用来通知进程在网络连接上到达了非规定波特率的数据。

为了接收SIGIO信号，需执行下列三步：

(1) 调用signal或sigaction为该信号建立一个信号处理程序。

(2) 以命令F_SETOWN（见3.13节）调用fcntl来设置进程ID和进程组ID，它们将接收对于该描述符的信号。

(3) 以命令F_SETFL调用fcntl设置O_ASYNC状态标志，使在该描述符上可以进行异步 I/O（见表3-2）。

第(3)步仅用于指向终端或网络的描述符，这是4.3+BSD异步传输设施的一个基本的限制。

对于SIGURG信号，只需执行第(1)步和第(2)步。该信号仅对于指向支持带外数据的网络连接的描述符而产生。

12.7 readv和writev函数

readv和writev函数用于在一个函数调用中读、写多个非连续缓存。有时也将这两个函数称为散布读（*scatter read*）和聚集写（*gather write*）。

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

ssize_t readv(int fd, const struct iovec iov[ ], int iovcnt);
ssize_t writev(int fd, const struct iovec iov[ ], int iovcnt);
```

两个函数返回：已读、写的字节数，若出错则为 -1

这两个函数的第二个参数是指向iovec结构数组的一个指针：

```
struct iovec {
    void *iov_base; /* starting address of buffer */
    size_t iov_len; /* size of buffer */
};
```

iov数组中的元素数由iovcnt说明。

这两个函数始于4.2BSD，现在SVR4也提供它们。

这两个函数的原型以及它们使用的iovec结构在各种有关文献资料中略有差别。如果比较它们在SVR4程序员手册〔AT&T 1990e〕、SVR4的SVID〔AT&T 1989〕，以及SVR4和4.3+BSD<sys/uio.h>头文件中的定义，那么它们之间都有差别。其部分原因是：SVID和SVR4程序员手册对应于1988 POSIX.1标准，而非1990版。上面示出的原型和结构定义对应于read和write的POSIX.1定义：缓存地址是void*，缓存长度是size_t，返回值是ssize_t。

注意，readv的第二个参数被说明为const。这取自4.3+BSD中该函数的原型，在SVR4的手册中则无此修饰词。对于readv此修饰词有效，因为并不修改iovec结构的成员——此函数只修改iov_base所指向的存储区。

4.3BSD和SVR4将iovcnt限制为16。4.3+BSD定义了常数UIO_MAXIOV，当前其值定义为1024。SVID声称常数IOV_MAX提供了系统V限制，但是它没有在SVR4的头文件中定义。

图12-11显示了readv和writev的参数和iovec结构之间的关系。writev以顺序iov[0]，iov[1]至

`iov[iovcnt-1]` 从缓存中聚集输出数据。`writenv` 返回输出的字节总数，它应等于所有缓存长度之和。
`readv` 则将读入的数据按上述同样顺序散布到缓存中。`readv` 总是先填满一个缓存，然后再填写下一个。`readv` 返回读得的总字节数。如果遇到文件结尾，已无数据可读，则返回 0。

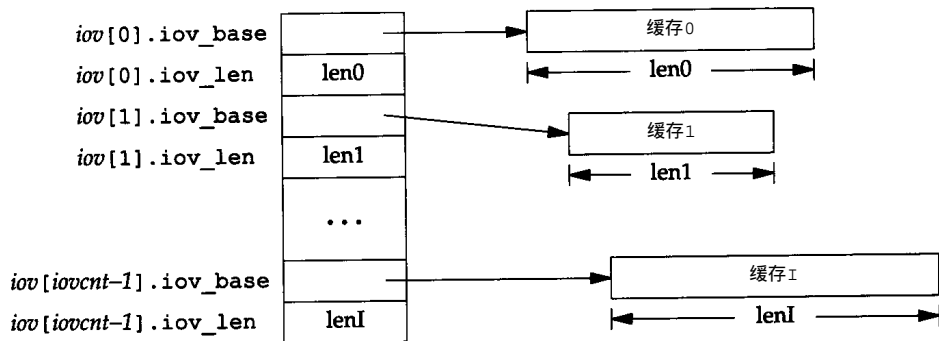


图12-11 readv和writenv的iovec结构

实例

在16.7节的 `_db_writex` 函数中，需将两个缓存连续地写到一个文件中。第二个缓存是调用者传递过来的一个参数，第一个缓存是我们创建的，它包含了第二个缓存的长度，文件中其他信息的位移量。有三种方法可以实现这一要求：

- (1) 调用 `wrtre` 两次，一次一个缓存。
- (2) 分配一个大到足以包含两个缓存的新缓存。将两个缓存的内容复制到新缓存中。然后对该缓存调用 `wrtre` 一次。
- (3) 调用 `writenv` 输出两缓存。

16.7节中使用了 `writenv`。将它与另外两种方法进行比较，对我们很有启发。表 12-7 显示了上面所述三种方法的结果。

表12-7 比较 writenv 和其他技术所得的时间结果

操 作	SPARC			80386		
	用户	系统	时钟	用户	系统	时钟
两次 write	0.2	7.2	17.2	0.5	13.1	13.7
缓存复制，然后一次 write	0.5	4.4	17.2	0.7	7.3	8.1
一次 write	0.3	4.6	17.1	0.3	7.8	8.2

所用的测试程序输出 100 字节的头文件，接着又输出 200 字节的数据。这样做 10 000 次，产生了一个百万字节的文件。该程序按上面所述方法写了 3 个版本，各运行一次，测得它们各使用的用户 CPU 时间、系统 CPU 时间和时钟时间。它们的单位都是秒。

正如我们所预料的，调用 `write` 两次的系统时间是调用 `write` 一次或调用 `writenv` 一次的两倍，这与表 3-1 的结果类似。

要注意的是：CPU 时间（用户加系统）几乎是个常数，无论是在缓存复制后用一个 `write` 还是用一个 `writenv`。两者的区别只是在用户空间（缓存复制）或系统空间（`writenv`）下执行的时间多一点。在 SPARC 系统上运行时，其和是 4.9 秒，而在 80386 系统下运行则是约 8.0 秒。

对于表 12-7 最后要说明的是，在 SPARC 上本测试所用的时钟时间主要用于磁盘数据传输上，

而在386系统上则主要用于CPU方面。

总而言之，我们一般采用 `readv` 和 `writev`，而不采用多次 `read` 和 `write`。时间结果表明采用缓存复制后用一个 `write` 与采用一个 `writev` 所用CPU时间几乎一样，但一般说来，因为前者还需要分配一个临时缓存用于存储及复制，所以后者更复杂。

12.8 readn和writen函数

某些设备，特别是终端、网络和SVR4的流设备有下列两种性质：

(1) 一次 `read` 操作所返回的数据可能少于所要求的数据，即使还没达到文件尾端。这不是一个错误，应当继续读该设备。

(2) 一次 `write` 操作的返回值也可能少于指定输出的字节数。这可能是由若干因素造成的，例如，下游模块的流量控制限制。这也不是错误，应当继续写余下的数据至该设备。（通常，只有对非阻塞描述符，或捕捉到一个信号时，才发生这种 `write` 返回。）

在读、写磁盘文件时没有这两种性质。

在第18章中，我们将写一个流管道（基于SVR4流或BSD UNIX域套接口），其中需要考虑这些特性。下面两个函数 `readn` 和 `writen` 的功能是读、写指定的N字节数据，并处理返回值小于要求值的情况。这两个函数只是按需多次调用 `read` 和 `write` 直至读、写了N字节数据。

```
#include "ourhdr.h"

ssize_t readn(int fd, void *buff, size_t nbytes);

ssize_t writen(int fd, void *buff, size_t nbytes);
```

两个函数返回：已读、写字节数，若出错则为 -1

在要将数据写到上面提到的设备上时，就可调用 `writen`，但是仅当先就知道要接收数据的数量时，才调用 `readn`（通常，调用 `read` 以接收来自这些设备的数据）。

程序12-12、程序12-13是 `writen` 和 `readn` 的一种实现。

程序12-12 writen函数

```
#include "ourhdr.h"

ssize_t writen(int fd, const void *vptr, size_t n)
{
    size_t nleft;
    ssize_t nwritten;
    const char *ptr;

    ptr = vptr; /* can't do pointer arithmetic on void* */
    nleft = n;
    while (nleft > 0) {
        if ((nwritten = write(fd, ptr, nleft)) <= 0)
            return(nwritten); /* error */

        nleft -= nwritten;
        ptr += nwritten;
    }
    return(n);
}
```

程序12-13 readn函数

```

#include    "ourhdr.h"

ssize_t    /* Read "n" bytes from a descriptor. */
readn(int fd, void *vptr, size_t n)
{
    size_t  nleft;
    ssize_t nread;
    char    *ptr;

    ptr = vptr;
    nleft = n;
    while (nleft > 0) {
        if ( (nread = read(fd, ptr, nleft)) < 0)
            return(nread); /* error, return < 0 */
        else if (nread == 0)
            break; /* EOF */

        nleft -= nread;
        ptr += nread;
    }
    return(n - nleft); /* return >= 0 */
}

```

12.9 存储映射 I/O

存储映射 I/O 使一个磁盘文件与存储空间中的一个缓存相映射。于是当从缓存中取数据，就相当于读文件中的相应字节。与其类似，将数据存入缓存，则相应字节就自动地写入文件。这样，就可以在不使用 read 和 write 的情况下执行 I/O。

为了使用这种功能，应首先告诉内核将一个给定的文件映射到一个存储区域中。这是由 mmap 函数实现的。

```

#include <sys/types.h>
#include <sys/mman.h>

caddr_t mmap(caddr_t addr, size_t len, int prot, int flag,
             int files, off_t off);

```

返回：若成功则为映射区的起始地址，若出错则为 -1

存储映照 I/O 已经用了很多年。4.1BSD (1981) 以其 vread 和 vwrite 函数提供了一种不同形式的存储映射 I/O。4.2BSD 没有使用这两个函数，而是希望使用 mmap 函数。但是由于 Leffler 等 [1989] 2.5 节中说明的理由，4.2BSD 实际并没有包含 mmap 函数。Gingell, Moran 和 Shannon [1987] 说明了 mmap 的一种实现。现在，SVR4 和 4.3+BSD 都支持 mmap 函数。

数据类型 caddr_t 通常定义为 char *。addr 参数用于指定映射存储区的起始地址。通常将其设置为 0，这表示由系统选择该映射区的起始地址。此函数的返回地址是：该映射区的起始地址。

files 指定要被映射文件的描述符。在映射该文件到一个地址空间之前，先要打开该文件。len 是映射的字节数。off 是要映射字节在文件中的起始位移量（下面将说明对 off 值有某些限制）。

在说明其余参数之前，先看一下存储映射文件的基本情况。图 12-12显示了一个存储映射文件。（见图7-3中进程存储空间的典型安排情况。）

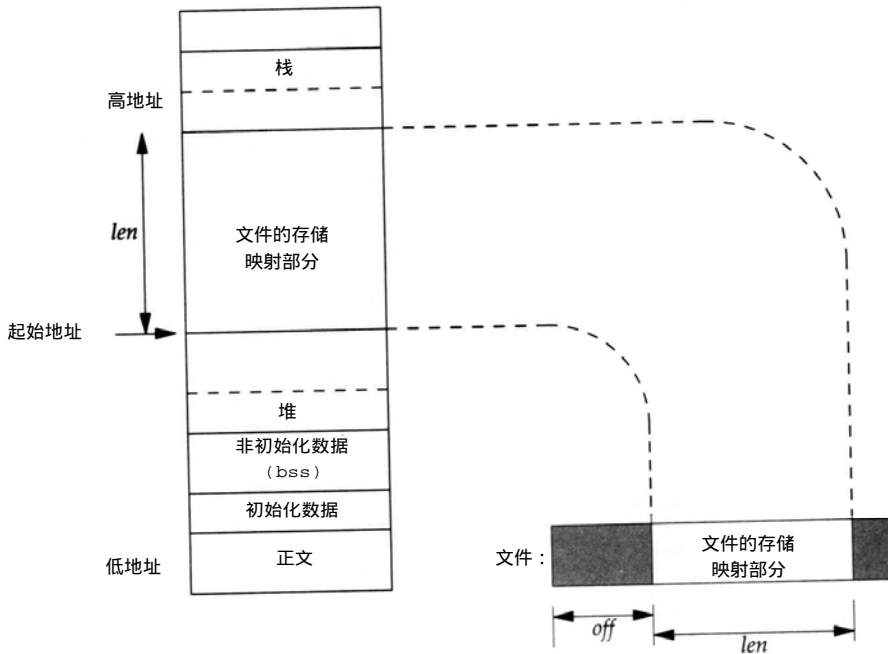


图12-12 存储映射文件的例子

在此图中，“起始地址”是mmap的返回值。在图中，映射存储区位于堆和栈之间：这属于实现细节，各种实现之间可能不同。

表12-8 存储映射区的保护

*prot*参数说明映射存储区的保护要求。

见表12-8。

对于映射存储区所指定的保护要求与文件的open方法匹配。例如，若该文件是只读打开的，那么对映射存储区就不能指定PROT_WRITE。

<i>prot</i>	说 明
PROT_READ	区域可读
PROT_WRITE	区域可写
PROT_EXEC	区域可执行
PROT_NONE	区域可存取(4.3+BSD无)

*flag*参数影响映射存储区的多种属性：

- MAP_FIXED 返回值必须等于*addr*。因为这不利于可移植性，所以不鼓励使用此标志。如果未指定此标志，而且*addr*非0，则内核只把*addr*视为何处设置映射区的一种建议。

通过将*addr*指定为0可获得最大可移植性。

- MAP_SHARED 这一标志说明了本进程对映射区所进行的存储操作的配置。此标志指定存储操作修改映射文件——也就是，存储操作相当于对该文件write。必须指定本标志或下一个标志(MAP_PRIVATE)。

- MAP_PRIVATE 本标志说明，对映射区的存储操作导致创建该映射文件的一个副本。所有后来对该映射区的存访都是存访该副本，而不是原始文件。

4.3+BSD还有另外一些MAP_XXX标志值，它们是这种现实所特有的。详细情况请参见4.3+BSD mmap(2)手册页。

*off*和*addr*的值(如果指定了MAP_FIXED)通常应当是系统虚存页长度的倍数。在SVR中，

虚存页长度可用带参数 `SC_PAGESIZE` 的 `sysconf` 函数（见 2.5.4 节）得到。在 4.3+BSD 之下，页长度由头文件 `<sys/param.h>` 中的常数 `NBPG` 定义。因为 `off` 和 `addr` 常常指定为 0，所以这种要求一般并不是问题。

因为映射文件的起动位移量受系统虚存页长度的限制，那么如果映射区的长度不是页长度的整数倍时，将如何呢？假定文件长 12 字节，系统页长为 512 字节，则系统通常提供 512 字节的映射区，其中后 500 字节被设为 0。可以修改这 500 字节，但任何变动都不会在文件中反映出来。

与映射存储区相关有两个信号：`SIGSEGV` 和 `SIGBUS`。信号 `SIGSEGV` 通常用于指示进程试图存取它不能存取的存储区。如果进程企图存取数据到用 `mmap` 指定为只读的映射存储区，那么也产生此信号。如果存取映射区的某个部分，而在存取时这一部分已不存在，则产生 `SIGBUS` 信号。例如，用文件长度映射一个文件，但在存访该映射区之前，另一个进程已将该文件截短。此时，如果进程企图存取对应于该文件尾端部分的映射区，则接收到 `SIGBUS` 信号。

在 `fork` 之后，子进程继承存储映射区（因为子进程复制父进程地址空间，而存储映射区是该地址空间中的一部分），但是由于同样的理由，`exec` 后的新程序则不继承此存储映射区。

进程终止时，或调用了 `munmap` 之后，存储映射区就被自动去除。关闭文件描述符 `filedes` 并不解除映射区。

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

int munmap(caddr_t addr, size_t len);
```

返回：若成功则为 0，若出错则为 -1

`munmap` 并不影响被映射的对象——也就是说，调用 `munmap` 并不使映射区的内容写到磁盘文件上。对于 `MAP_SHARED` 区磁盘文件的更新，在写到存储映射区时按内核虚存算法自动进行。

某些系统提供了一个 `msync` 函数，它类似于 `fsync` 函数（见 4.24 节），但对存储映射区起作用。

实例

程序 12-14 用存储映射 I/O 复制一个文件（类似于 `cp(1)` 命令）。首先打开两个文件，然后调用 `fstat` 得到输入文件的长度。在调用 `mmap` 和设置输出文件长度时都需使用输入文件长度。调用 `lseek`，然后写一个字节以设置输出文件的长度。如果不设置输出文件的长度，则对输出文件调用 `mmap` 也可以，但是对相关存储区的第一次存访会产生 `SIGBUS`。也可使用 `ftruncate` 函数来设置输出文件的长度，但是并非所有系统都支持该函数扩充文件长度（见 4.13 节）。

然后对每个文件调用 `mmap`，将文件映射到存储区，最后调用 `memcpy` 将输入缓存的内容复制到输出缓存。在从输入缓存（`src`）取数据字节时，内核自动读输入文件；在将数据存入输出缓存（`dst`）时，内核自动将数据写到输出文件中。

程序 12-14 用存储映射 I/O 复制文件

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h> /* mmap() */
#include <fcntl.h>
#include "ourhdr.h"
```

```

#ifndef MAP_FILE    /* 4.3+BSD defines this & requires it to mmap files */
#define MAP_FILE    0    /* to compile under systems other than 4.3+BSD */
#endif

int
main(int argc, char *argv[])
{
    int      fdin, fdout;
    char     *src, *dst;
    struct stat statbuf;

    if (argc != 3)
        err_quit("usage: a.out <fromfile> <tofile>");
    if ( (fdin = open(argv[1], O_RDONLY)) < 0)
        err_sys("can't open %s for reading", argv[1]);
    if ( (fdout = open(argv[2], O_RDWR | O_CREAT | O_TRUNC,
                       FILE_MODE)) < 0)
        err_sys("can't creat %s for writing", argv[1]);
    if (fstat(fdin, &statbuf) < 0) /* need size of input file */
        err_sys("fstat error");
        /* set size of output file */
    if (lseek(fdout, statbuf.st_size - 1, SEEK_SET) == -1)
        err_sys("lseek error");
    if (write(fdout, "", 1) != 1)
        err_sys("write error");
    if ( (src = mmap(0, statbuf.st_size, PROT_READ,
                    MAP_FILE | MAP_SHARED, fdin, 0)) == (caddr_t) -1)
        err_sys("mmap error for input");
    if ( (dst = mmap(0, statbuf.st_size, PROT_READ | PROT_WRITE,
                    MAP_FILE | MAP_SHARED, fdout, 0)) == (caddr_t) -1)
        err_sys("mmap error for output");
    memcpy(dst, src, statbuf.st_size); /* does the file copy */
    exit(0);
}

```

将存储区映射复制与用 read, write 进行的复制 (缓存长度为 8192) 相比较, 得到表 12-9 中所示的结果。

表12-9 read/write与mmap/memcpy比较的时间结果

操 作	SPARC			80336		
	用户	系统	时钟	用户	系统	时钟
read/write	0.0	2.6	11.0	0.0	5.3	11.2
mmap/memcpy	0.9	1.7	3.7	0.3	2.7	5.7

时间单位是秒, 被复制文件的长度是约 3 百万字节。

对于 SPARC, 两种复制方式的 CPU 时间 (用户+系统) 相同, 都是 2.6 秒。(这与表 12-7 中 writev 的情况类似)。对于 386, mmap/memcpy 方式大约是 read/write 方式的一半。

使用 mmap 时, SPARC 和 386 系统时间都减少的原因是: 内核直接对映射存储缓存作 I/O 操作。而在 read/write 方式, 内核要在用户缓存和它自己的缓存之间进行复制, 然后用其缓存作 I/O。

另一个要注意的是, 使用 mmap/memcpy 时, 时钟时间至少减半。

将一个普通文件复制到另一个普通文件中时, 存储映射 I/O 比较快。但是有一些限制, 如不能用其在某些设备之间 (例如网络设备或终端设备) 进行复制, 并且对被复制的文件进行映

射后，也要注意该文件的长度是否改变。尽管如此，有很多应用程序会从存储映射 I/O 得到好处，因为它处理的是存储空间而不是读、写一个文件，所以常常可以简化算法。从存储映射 I/O 中得益的一个例子是帧缓存设备，该设备引用一个位-映射显示。

Krieger, Stumm 和 Unrau [1992] 第5章说明了一个使用存储映射 I/O 的标准 I/O 库。

14.9 节将返回到存储映射 I/O，其中有一个例子，说明在 SVR4 和 4.3+BSD 之下如何使用存储映射 I/O 在有关进程间提供共享存储区。

12.10 小结

本章说明了很多高级 I/O 功能，其中大多数将在后面章节的例子中使用：

- 非阻塞 I/O——发一个 I/O 操作，不使其阻塞。
- 记录锁。
- 系统 V 流机制。
- I/O 多路转接——select 和 poll 函数。
- readv 和 writev 函数。
- 存储空间映射 I/O (mmap)。

习题

12.1 删除程序 12-6 for 循环中第二次调用 write 的语句后结果如何，为什么？

12.2 查看系统中 <sys/types.h> 头文件，并研究 select 和四个 FD_ 宏的实现。

12.3 <sys/types.h> 头文件中定义了 fd_set 数据类型可以处理的最大描述符数，假设需要将描述符数增加到 2048，该如何实现？

12.4 比较处理信号量集的函数（见 10.11 节）和 fd_set 描述符集的函数，并研究在你的系统上实现它们的方法。

12.5 getmsg 可以返回多少种不同的信息？

12.6 用 select 或 poll 实现一个与 sleep 类似的函数 sleep_us，不同之处是要等待指定的若干微秒。比较这个函数和 BSD 中的 usleep 函数。

12.7 是否可以利用建议性锁来实现程序 10-17 中的函数 TELL_WAIT、TELL_PARENT、TELL_CHILD、WAIT_PARENT 以及 WAIT_CHILD？如果可以，编写这些函数并测试其功能。

12.8 用 select 或 poll 测试管道的容量。将其值与第 2 章的 PIPE_BUF 的值比较。

12.9 运行程序 12-14 拷贝一个文件，检查输入文件的上一次访问时间是否改变了？

12.10 在程序 12-14 中 mmap 后调用 close 关闭输入文件，以验证关闭描述符不会使内存映射 I/O 失效。

第13章 精灵进程

13.1 引言

精灵进程 (daemon) 是生存期长的一种进程。它们常常在系统引导装入时起动, 在系统关闭时终止。因为它们没有控制终端, 所以说它们是在后台运行的。UNIX系统有很多精灵进程, 它们执行日常事物活动。

本章说明精灵的进程结构, 以及如何编写精灵进程程序, 因为精灵没有控制终端, 我们需要了解在有关事物出问题, 精灵进程如何报告出错情况。

13.2 精灵进程的特征

先来察看一些常用的系统精灵进程, 以及它们怎样和第9章中所叙述的概念: 进程组、控制终端和对话期相关联。ps(1)命令打印系统中各个进程的状态。该命令有多个选择项, 有关细节请参考系统手册。为了察看本节讨论中所需的信息, 在4.3+BSD或SunOS系统下执行:

```
ps -axj
```

选择项-a显示由其他用户所拥有的进程的状态。-x显示没有控制终端的进程的状态。-j显示与作业有关的信息: 对话期ID、进程组ID、控制终端以及终端进程组ID。在SVR4之下, 与此相类似的命令是ps -efjc (在某些符合美国国防部安全性准则要求的UNIX系统中, 只能使用ps查看自己所拥有的进程)。ps的输出大致是:

PPID	PID	PGID	SID	TT	TPGID	UID	COMMAND
0	0	0	0	?	-1	0	swapper
0	1	0	0	?	-1	0	/sbin/init -
0	2	0	0	?	-1	0	pagedaemon
1	80	80	80	?	-1	0	syslogd
1	88	88	88	?	-1	0	/usr/lib/sendmail -bd -qlh
1	105	37	37	?	-1	0	update
1	108	108	108	?	-1	0	cron
1	114	114	114	?	-1	0	inetd
1	117	117	117	?	-1	0	/usr/lib/lpd

其中, 已移去了一些我们并无兴趣的列, 例如累计CPU时间。按照顺序, 各列标题的意义是: 父进程ID、进程ID、进程组ID、终端名称、终端进程组ID (与该控制终端相关的前台进程组)、用户ID以及实际命令字符串。

这些ps命令在支持对话期ID的系统 (SunOS) 上运行, 9.5节的setsid函数中曾提及对话期ID。它是对话期首进程的进程ID。但是, 4.3+BSD系统将打印与本进程所属进程组对应的session结构的地址 (见9.11节)。

进程0、1以及2是8.2节中所述的进程。这些进程非常特殊, 存在于系统的整个生命期中。它们没有父进程ID, 没有组进程ID, 也没有对话期ID。syslogd精灵进程可用于任何为操作人

员记录系统消息的程序中。可以在一台实际的控制台上打印这些消息，也可将它们写到一个文件中（13.4.2节将对syslog设施进行说明）。sendmail是标准邮递精灵进程。update程序定期将内核缓存中的内容写到硬盘上（通常是每隔30秒）。为了做到这一点，该程序每隔30秒调用sync(2)函数一次（4.24节已对sync进行了说明）。cron精灵进程在指定的日期和时间执行指定的命令。许多系统管理任务是由cron定期地使相关程序执行而得以实现的。我们已在9.3节中提到inetd精灵进程。它监听系统的网络界面，以输入对各种网络服务器的请求。最后一个精灵进程，lpd处理对系统提出的各个打印请求。

注意，所有精灵进程都以超级用户（用户ID为0）的优先权运行。没有一个精灵进程具有控制终端——终端名称设置为问号（?）、终端前台进程组ID设置为-1。缺少控制终端可能是精灵进程调用了setsid的结果。除update以外的所有精灵进程都是进程组的首进程，对话期的首进程，而且是这些进程组和对话期中的唯一进程。update是它所在进程组（37）和对话期（37）中的唯一进程，但是该进程组的首进程（可能也是该对话期的首进程）已经终止。最后，应当引起注意的是所有这些精灵进程的父进程都是init进程。

13.3 编程规则

在编写精灵进程程序时需遵循一些基本规则，以便防止产生并不希望的交互作用。下面先说明这些规则，然后是一个按照规则编写的函数daemon_init。

(1) 首先做的是调用fork，然后使父进程exit。这样做实现了下面几点：第一，如果该精灵进程是由一条简单shell命令起动的，那么使父进程终止使得shell认为这条命令已经执行完成。第二，子进程继承了父进程的进程组ID，但具有一个新的进程ID，这就保证了子进程不是一个进程组的首进程。这对于下面就要做的setsid调用是必要的前提条件。

(2) 调用setsid以创建一个新对话期。于是执行9.5节中列举的三个操作，使调用进程：(a) 成为新对话期的首进程，(b) 成为一个新进程组的首进程，(c) 没有控制终端。

在SVR之下，有些人建议在此时再调用fork，并使父进程终止。第二个子进程作为精灵进程继续运行。这样就保证了该精灵进程不是对话期首进程，于是按照SVR4规则（见9.6节）可以防止它取得控制终端。另一方面，为了避免取得控制终端，无论何时打开一个中断设备都要指定O_NOCTTY。

(3) 将当前工作目录更改为根目录。从父进程继承过来的当前工作目录可能在一个装配的文件系统中。因为精灵进程通常在系统再引导之前是一直存在的，所以如果精灵进程的当前工作目录在一个装配文件系统中，那么该文件系统就不能被拆卸。

另外，某些精灵进程可能会把当前工作目录更改到某个指定位置，在此位置做它们的工作。例如，行式打印机假脱机精灵进程常常将其工作目录更改到它们的spool目录上。

(4) 将文件方式创建屏蔽字设置为0。由继承得来的文件方式创建屏蔽字可能会拒绝设置某些许可权。例如，若精灵进程要创建一个组可读、写的文件，而继承的文件方式创建屏蔽字，屏蔽了这两种许可权，则所要求的组可读、写就不能起作用。

(5) 关闭不再需要的文件描述符。这样使精灵进程就不再持有从其父进程继承来的某些文件描述符（父进程可能是shell进程，或某个其他进程）。但是，究竟关闭哪些描述符则与具体的精灵进程有关，所以在下面的例子中不包含此步骤。可以使用程序2-3中的open_max函数来决定最高文件描述符值，并关闭直到该值的所有描述符。

实例

程序13-1是个函数，可由想初始化成为一个精灵进程的程序调用。

程序13-1 初始化一个精灵进程

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

int
daemon_init(void)
{
    pid_t pid;

    if ( (pid = fork()) < 0)
        return(-1);
    else if (pid != 0)
        exit(0); /* parent goes bye-bye */

    /* child continues */
    setsid(); /* become session leader */

    chdir("/"); /* change working directory */

    umask(0); /* clear our file mode creation mask */

    return(0);
}
```

若daemon_init函数由main函数调用，然后进入睡眠状态，那么可以用ps命令检查该精灵进程的状态：

```
$ a.out
$ ps_axj
PPID  PID  PGID  SID   TT   TPGID  UID   COMMAND
  1    735  735   735   ?    -1     224   a.out
```

从中可以看到，该精灵进程已被正确地初始化。

13.4 出错记录

与精灵进程有关的一个问题是如何处理出错消息。因为它没有控制终端，所以不能只是写到标准出错输出上。在很多工作站上，控制台设备运行一个窗口系统，所以我们不希望所有精灵进程都写到控制台设备上。我们也不希望每个精灵进程将它自己的出错消息写到一个单独的文件中。对系统管理人员而言，如果要关心哪一个精灵进程写到哪一个记录文件中，并定期地检查这些文件，那么一定会使他感到头痛。所以，需要有一个集中的精灵进程出错记录设施。

伯克利开发了BSD syslog 设施，并广泛应用于4.2BSD。从4.xBSD导出的很多系统都支持syslog。13.4.2节将说明该设施。

系统V中从来没有一个集中的精灵进程记录设施。SVR4支持BSD风格的syslog设施，SVR4之下的inetd精灵进程使用syslog。在SVR中，syslog的基础是/dev/log流设备驱动程序，下一节将对此进行说明。

13.4.1 SVR4流log驱动程序

SVR4提供了一种流设备驱动程序，其界面具有流出错记录，流事件跟踪以及控制台记录功能。有关文献包含在〔AT&T 1990d〕的log(7)中。图13-1详细给出了这种设施的整个结构。

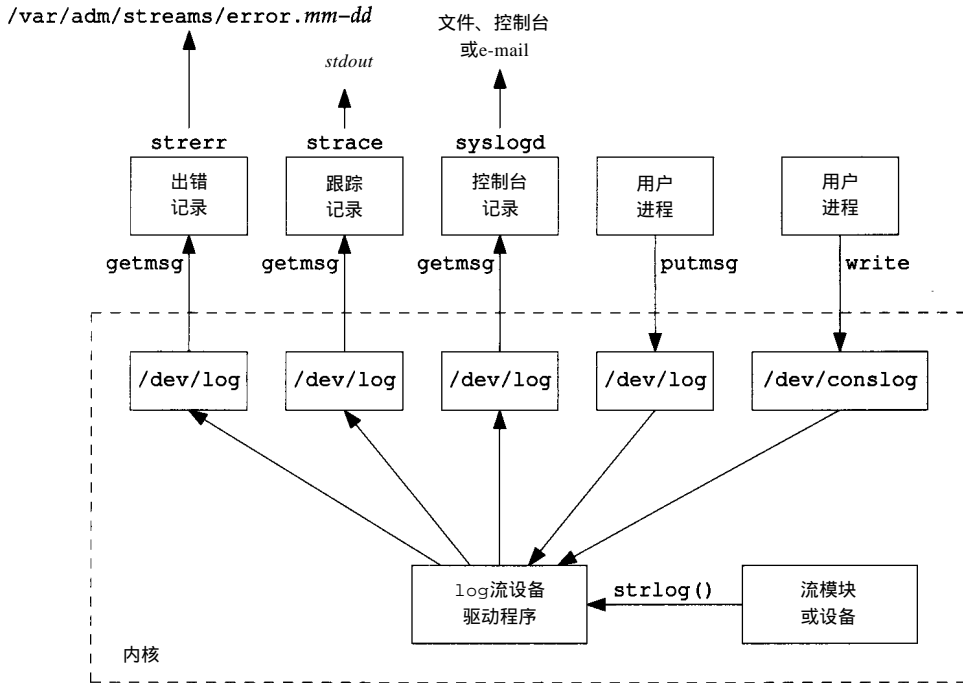


图13-1 SVR4 log 设施

有三个记录进程 (logger)：出错记录进程、跟踪记录进程以及控制台记录进程。每一条记录消息可以送给其中之一。

下面介绍三种产生记录消息的方法以及三种读记录消息的方法。

- 产生记录消息。

(1) 内核中的例程可以调用 `strlog` 以产生记录消息。这种方法通常由流模块和流设备驱动程序用于出错消息或跟踪消息（跟踪消息常用在新的流模块或驱动程序的排错中）。因为我们无意编写内核中的例程，所以不详细说明这种消息产生方法。

(2) 一个用户进程（例如一个精灵进程）可以用 `putmsg` 将消息送到 `/dev/log`。这种消息可被送到三个记录进程中的任意一个。

(3) 一个用户进程（例如一个精灵进程）可以用 `write` 将消息写到 `/dev/conslog`。这种消息只能送向控制台记录进程。

- 读记录消息。

(4) 标准的出错记录进程是 `strerr` (1M)。它将记录消息增写到在目录 `/var/adm/stream` 下的一个文件中。该文件名是 `error.mm-dd`，其中，`mm` 是月份，`dd` 是天数。`strerr` 本身是个精灵进程，通常在后台运行，它将记录消息增写到该文件中。

(5) 标准的跟踪记录进程是 `strace` (1M)。它能有选择地将一套指定的跟踪消息写至其标准输出。

(6) 标准控制台记录进程是 `syslogd`，这是一个 BSD 导出程序，下一节将对此进行叙述。此进程是个精灵进程，它读一个配置文件，然后将记录消息写至一个指定的文件（控制台是一个

文件)或登录用户,或将该消息发送给在另一台主机上的 syslog 精灵进程。

虽然上面没有提及,但用户也可以用自己的进程替换任意一个系统提供的标准精灵进程。我们可以提供自己的出错记录进程、跟踪记录进程或控制台记录进程。

每则 log 消息除消息本身外,还包含有一些其他信息。例如,由 log 驱动程序沿逆流方向发送的消息,还包含有下列消息:哪个模块产生此消息(如果该消息是由内核中的一个流模块产生的)、级别、优先级、某些标志以及消息产生的时间。有关细节请参阅手册中的 log(7)。如果使用 putmsg 产生一则 log 消息,则可以设置这些字段中的几个。如果调用 write 将一则消息发送至控制台记录进程(通过 /dev/console),则只能发送消息字符串。

图13-1中没有显示的另一种可能性是:由一个 SVR4 精灵进程调用 BSD syslog(3) 函数。用这种方法可将消息发送至控制台记录进程,这与用 putmsg 向 /dev/log 发送消息类似。使用 syslog, 可以设置消息的优先权字段。下一节将讨论此函数。

当产生了某种类型的记录消息,但是相应类型的记录进程却不在运行时, log 驱动程序丢弃该消息。

不幸的是,在 SVR4 中,使用这种 log 设施带有随意性。一些精灵进程使用它,而大多数由系统提供的精灵进程则编写成直接写向控制台。

syslog(3) 函数和 syslogd(1M) 精灵进程的有关文档在 BSD 兼容库文档部分 [AT&T 1990c], 但是它们本身并不在此库中,而是在所有用户进程(精灵进程)都可使用的标准 C 库中。

13.4.2 4.3+BSD syslog 设施

自4.2BSD以来,广泛地应用了BSD syslog设施。大多数精灵进程使用这一设施。图13-2显示了syslog设施的详细组织结构。

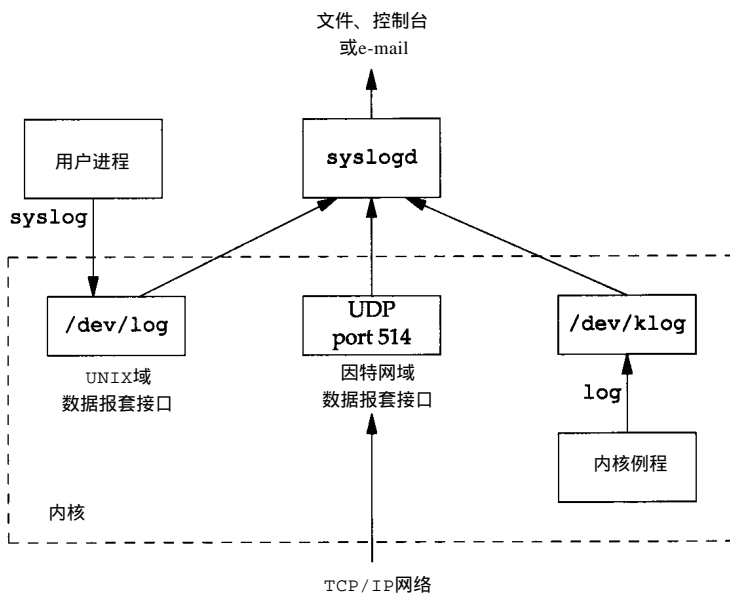


图13-2 4.3+BSD syslog设施

有三种方法产生记录消息：

(1) 内核例程可以调用log函数。任何一个用户进程通过打开和读/dev/klog设备就可以读取这些消息。因为我们无意编写内核中的例程，所以不再进一步说明此函数。

(2) 大多数用户进程（精灵进程）调用syslog(3)函数以产生记录消息。我们将在下面说明其调用序列。这使消息发送至UNIX域数据报套接口/dev/log。

(3) 在此主机上，或通过TCP/IP网络连接到此主机的某一其他主机上的一个用户进程可将记录消息发向UDP端口514。注意：syslog函数并不产生这些UDP数据报——它们要求产生此记录消息的进程具有显式的网络编程。

关于UNIX域套接口以及UDP套接口的细节，请参阅stevens [1990]。

通常，syslogd精灵进程读取三种格式的记录消息。此精灵进程在起动时读一个配置文件。一般，其文件名为/etc/syslog.conf，该文件决定了不同种类的消息应送向何处。例如，紧急消息可被送向系统管理员（若已登录），并在控制台上显示，而警告消息则可记录到一个文件中。

该设施的界面是syslog函数。

```
#include <syslog.h>
void openlog(char id*nt, int option, int facility);
void syslog(int priority, char format, ...);
void closelog(void);
```

调用openlog是可选择的。如果不调用openlog，则在第一次调用syslog时，自动调用openlog。调用closelog也是可选择的——它只是关闭被用于与syslogd精灵进程通信的描述符。

调用openlog使我们可以指定一个ident，以后，此ident将被加至每则记录消息中。ident一般是程序的名称（例如，cron、inetd等）。表13-1说明了4种可能的option。

表13-1 openlog的option参数

option	说 明
LOG_CONS	若日志消息，不能通过UNIX域数据报发送至syslogd，则将该消息写至控制台
LOG_NDELAY	立即打开UNIX域数据报套接口至syslogd精灵进程——不要等到记录第一条消息。通常，在记录第一条消息之前，该套接口不打开
LOG_PERROR	除将日志消息发送给syslog外，还将它写至标准出错。此选项仅由4.3BSD Reno及以后版本支持
LOG_PID	每条消息都包含进程ID此选择项可供对每个请求都fork一个子进程的精灵进程使用

openlog中的参数facility可以选取表13-2中列举的值。设置facility参数的目的是让配置文件可以说明，来自不同设施的消息以不同的方式进行处理。如果不调用openlog，或者以facility为0来调用它，那么在调用syslog时，可将facility作为priority参数的一个部分进行说明。

调用syslog产生一个记录消息。其priority参数是facility和level的组合，它们可选取的值分别列于facility（见表13-2）和level（见表13-3）中。level值按优先级从最高到最低按序排列。

format参数以及其他参数传至vsprintf函数以便进行格式化。在format中，每个%m都被代换成对应于errno值的出错消息字符串（strerror）。

SVR4和4.3+BSD都提供logger(1)程序，以其作为向syslog设施发送出错消息的方法。送至该程序的可选择参数可以指定facility、level以及ident。logger的意图是用于以非交互方式运行，又要产生记录消息的shell过程。

表13-2 openlog的*facility*参数

<i>facility</i>	说 明
LOG_AUTH	授权程序: login, su, getty, ...
LOG_CRON	cron和at
LOG_DAEMON	系统精灵进程: ftpd, routed, ...
LOG_KERN	内核产生的消息
LOG_LOCAL0	保留由本地使用
LOG_LOCAL1	保留由本地使用
LOG_LOCAL2	保留由本地使用
LOG_LOCAL3	保留由本地使用
LOG_LOCAL4	保留由本地使用
LOG_LOCAL5	保留由本地使用
LOG_LOCAL6	保留由本地使用
LOG_LOCAL7	保留由本地使用
LOG_LPR	行打系统: lpd, lpc, ...
LOG_MAIL	邮件系统
LOG_NEWS	Usenet网络新闻系统
LOG_SYSLOG	syslogd精灵进程本身
LOG_USER	来自其他用户进程的消息
LOG_UUCP	UUCP系统

表13-3 syslog中的*levels* (按序排列)

<i>level</i>	说 明
LOG_EMERG	紧急(系统不可使用)(最高优先级)
LOG_ALERT	必须立即修复的条件
LOG_CRIT	临界条件(例如,硬设备出错)
LOG_ERR	出错条件
LOG_WARNING	警告条件
LOG_NOTICE	正常,但重要的条件
LOG_INFO	信息性消息
LOG_DEBUG	调试排错消息(最低优先级)

logger命令的格式正由POSIX.2标准化。

实例

第17章的PostScript打印机精灵进程中,包含有下面的调用序列:

```
openlog("lprps", LOG_PID, LOG_LPR);
syslog(LOG_ERR, "open error for %s: %m", filename);
```

第一个调用将*ident*字符串设置为程序名,指定打印该进程ID,并且将系统默认的*facility*设定为行式打印机系统。对syslog的实际调用指定一个出错条件和一个消息字符串。如若不调用openlog,则第二个调用的形式可能是:

```
syslog(LOG_ERR | LOG_LPR, "open error for %s: %m", filename);
```

其中,将*priority*参数指定为*level*和*facility*的组合。

13.5 客户机-服务器模型

精灵进程常常用作服务器进程。确实，图 13-2 中，可以称 `syslogd` 进程为服务器，用户进程（客户机）用 UNIX 域数据报套接口向其发送消息。

一般而言，服务器是一个进程，它等待客户机与其联系，提出某种类型的服务要求。图 13-2 中，由 `syslogd` 服务器提供的服务是记录出错消息。

图 13-2 中，客户机和服务器之间的通信是单向的。客户机向服务器发送其服务要求，服务器则不向客户机回送任何消息。在下面有关进程通信的几章中，有大量实例，其中有客户机和服务器之间的双向通信。客户机向服务器发送要求，服务器则向客户机回送回答。

13.6 小结

在大多数 UNIX 系统中，精灵进程是一直运行的。为了初始化我们自己的精灵进程，需要一些审慎的思索并理解第 9 章中说明过的进程之间的关系。本章开发了一个可由精灵进程调用，对其自身正确地进行初始化的函数。

本章还讨论了精灵进程记录出错消息的几种方法，因为精灵进程通常没有控制终端。在 SVR4 下，可以使用流记录驱动程序，在 4.3+BSD 之下，提供了 `syslog` 设施。因为 SVR4 也提供 BSD `syslog` 设施，所以在下面的章节中，当精灵进程需要记录出错消息时，将调用 `syslog` 函数。第 17 章中，PostScript 打印机精灵进程就包含有这种情况。

习题

13.1 从图 13-2 可以看出，直接调用 `openlog` 或第一次调用 `syslog` 都可以初始化 `syslog`，此时一定要打开用于 UNIX 域的数据报套接口的特殊设备文件 `/dev/log`。如果调用 `openlog` 前，用户进程（精灵进程）先调用了 `chroot`，结果如何？

13.2 列出你的系统中所有的精灵进程，并说明它们的功能。

13.3 编写一段调用程序 13-1 中 `daemon_init` 函数的程序。调用该函数后调用 `getlogin`（见 8.14 节）查看该精灵进程是否有登录名。若程序带有 `3 > /tmp/name1` 运行时（Bourne shell 或 KornShell），则将登录名打印到文件描述符 3，并重定向到一个临时文件。在调用 `daemon_init` 和 `getlogin` 之间关闭描述符 1、2 和 3，此时再运行该程序会有什么不同？

13.4 编写一个 SVR4 精灵进程，将其作为一个控制台记录进程。细节可参阅 [AT&T 1990d] 的 `log(7)`。每次接收一则消息，并打印相关信息。再编写一个测试程序，将控制台记录消息发送给 `/dev/log` 以测试该精灵进程。

13.5 根据 13.3 节中提到的规则 (2)，通过调用两次 `fork` 修改程序 13-1，以使它在 SVR4 下不能取得控制终端。测试新的函数，并验证它不是一个对话期首进程。

第14章 进程间通信

14.1 引言

第8章说明了进程控制原语并且观察了如何调用多个进程。但是这些进程之间交换信息的唯一方法是经由fork或exec传送打开文件，或通过文件系统。本章将说明进程之间相互通信的其他技术——IPC (InterProcess Communication)。

UNIX IPC已经是而且继续是各种进程通信方式的统称，其中极少能在所有UNIX的实现中进行移植。表14-1列出了不同实现所支持的不同形式的IPC。

表14-1 UNIX IPC

IPC类型	POSIX.1	XPG3	V7	SVR2	SVR3.2	SVR4	4.3BSD	4.3+BSD
管道(半双工)	•	•	•	•	•	•	•	•
FIFOs(命令管道)	•	•		•	•	•		•
流管道(全双工)					•	•	•	•
命令流管道					•	•	•	•
消息队列		•		•	•	•		
信号量		•		•	•	•		
共享存储		•		•	•	•		
套接口						•	•	•
流					•	•		

正如上表所示，不管哪一种UNIX实现，都可依靠的唯一一种IPC是半双工的管道。表中前7种IPC通常限于同一台主机的各个进程间的IPC。最后两种；套接口和流，则支持不同主机上各个进程间IPC（关于网络IPC的详细情况，请参见Stevens [1990]）。虽然中间三种形式的IPC（消息队列、信号量以及共享存储器）在表中说明为只受到系统V的支持，但是在大多数制造商所支持的，从伯克利UNIX导出的UNIX系统中（例如，SunOS以及Ultrix），已经添加了这三种形式的IPC。

几个POSIX小组正在对IPC进行工作，但是最后结果还不很清楚，可能要到1994年甚至更迟一点与IPC有关的POSIX才能制定出来。

我们将与IPC有关的讨论分成两章。本章将讨论经典的IPC；管道、FIFO、消息队列、信号量以及共享存储器。下一章将观察SVR4和4.3+BSD共同支持的IPC的某些高级特征，包括；流管道和命名流管道，以及用这些更高级形式的IPC可以做的一些事情。

14.2 管道

管道是UNIX IPC的最老形式，并且所有UNIX系统都提供此种通信机制，管道有两种限制；

- (1) 它们是半双工的。数据只能在一个方向上流动。
- (2) 它们只能在具有公共祖先的进程之间使用。通常，一个管道由一个进程创建，然后该

进程调用fork，此后父、子进程之间就可应用该管道。

我们将会看到流管道（见15.2节）没有第一种限制，FIFO（见14.5节）和命名流管道（见15.5节）则没有第二种限制。尽管有这两种限制，半双工管道仍是最常用的IPC形式。

管道是由调用pipe函数而创建的。

```
#include <unistd.h>
int pipe(intfiledes[2]) ;
```

返回：若成功则为0，若出错则为-1

经由参数filedes返回两个文件描述符：*filedes[0]*为读而打开，*filedes[1]*为写而打开。*filedes[1]*的输出是*filedes[0]*的输入。

有两种方法来描绘一个管道，见图14-1。左半图显示了管道的两端在一个进程中相互连接，右半图则说明数据通过内核在管道中流动。

在SVR4下，管道是双全工的。两个描述符都可用于读、写。于是，图14-1中的箭头在两端都有。我们称这种双全工管道为“流管道”，下一章将详细讨论这种管道。因为POSIX.1只提供半双工管道，为了可移植性，我们假定pipe函数创建一个单方向的管道。

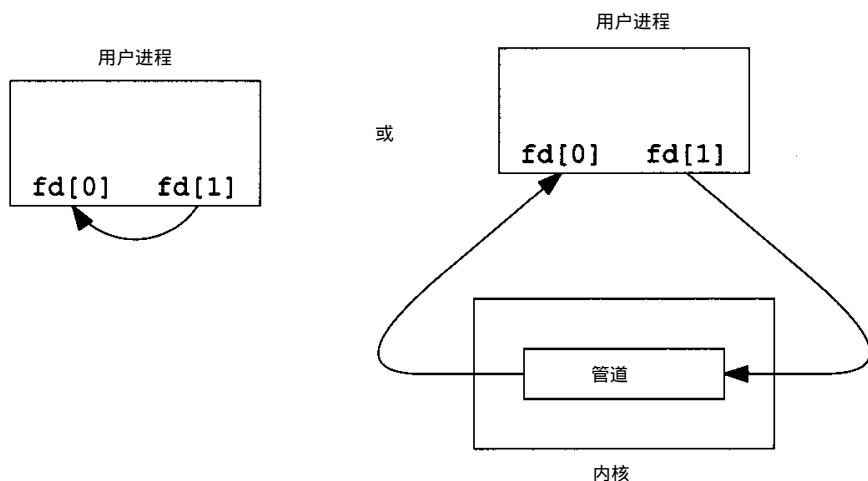


图14-1 观察UNIX管道的两种方法

fstat函数（见4.2节）对管道的每一端都返回一个FIFO类型的文件描述符，可以用S_ISFIFO宏来测试管道。

POSIX.1规定stat结构的st_size成员对于管道是未定义的。但是当fstat函数应用于管道读端的文件描述符时，很多系统在st_size中存放管道中可用于读的字节数。但是，这是不可移植的。

单个进程中的管道几乎没有任何用处。通常，调用pipe的进程接着调用fork，这样就创建了从父进程到子进程或反之的IPC通道。图14-2显示了这种情况。

fork之后做什么取决于我们想要有的数据流的方向。对于从父进程到子进程的管道，父进程关闭管道的读端（fd[0]），子进程则关闭写端（fd[1]）。图14-3显示了描述符的最后安排。

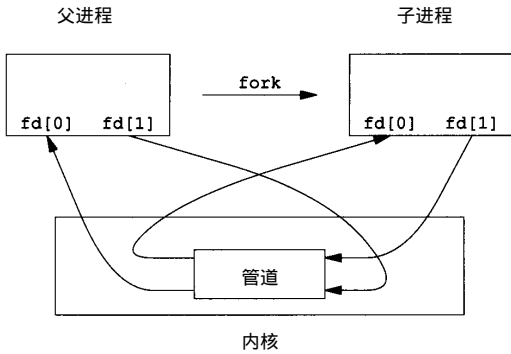


图14-2 fork之后的半双工管道

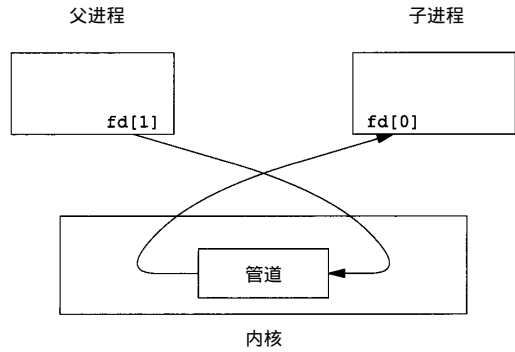


图14-3 从父进程到子进程的管道

对于从子进程到父进程的管道，父进程关

闭 fd[1]，子进程关闭fd[0]。

当管道的一端被关闭后，下列规则起作用：

(1) 当读一个写端已被关闭的管道时，在所有数据都被读取后，read返回0，以指示达到了文件结束处（从技术方面考虑，管道的写端还有进程时，就不会产生文件的结束。可以复制一个管道的描述符，使得有多个进程具有写打开文件描述符。但是，通常一个管道只有一个读进程，一个写进程。下一节介绍FIFO时，我们会看到对于一个单一的FIFO常常有多个写进程）。

(2) 如果写一个读端已被关闭的管道，则产生信号SIGPIPE。如果忽略该信号或者捕捉该信号并从其处理程序返回，则write出错返回，errno设置为EPIPE。

在写管道时，常数PIPE_BUF规定了内核中管道缓存器的大小。如果对管道进行write调用，而且要求写的字节数小于等于PIPE_BUF，则此操作不会与其他进程对同一管道（或FIFO）的write操作穿插进行。但是，若有多个进程同时写一个管道（或FIFO），而且某个或某些进程要求写的字节数超过PIPE_BUF字节数，则数据可能会与其他写操作的数据相穿插。

实例

程序14-1创建了一个从父进程到子进程的管道，并且父进程经由该管道向子进程传送数据。

程序14-1 经由管道父进程向子进程传送数据

```
#include    "ourhdr.h"

int
main(void)
{
    int    n, fd[2];
    pid_t  pid;
    char   line[MAXLINE];

    if (pipe(fd) < 0)
        err_sys("pipe error");

    if ( (pid = fork()) < 0)
        err_sys("fork error");

    else if (pid > 0) {    /* parent */
        close(fd[0]);
        write(fd[1], "hello world\n", 12);
    } else {                /* child */
```

```

        close(fd[1]);
        n = read(fd[0], line, MAXLINE);
        write(STDOUT_FILENO, line, n);
    }

    exit(0);
}

```

在上面的例子中，直接对管道描述符调用 read和write。更为有益的是将管道描述符复制为标准输入和标准输出。在此之后通常子进程调用 exec，执行另一个程序，该程序从标准输入（已创建的管道）或将数据写至其标准输出（管道）。

实例

试编写一个程序，其功能是每次一页显示已产生的输出。已经有很多 UNIX公用程序具有分页功能，因此无需再构造一个新的分页程序，而是调用用户最喜爱的分页程序。为了避免先将所有数据写到一个临时文件中，然后再调用系统中的有关程序显示该文件，我们希望将输出通过管道直接送到分页程序。为此，先创建一个管道，一个子进程，使子进程的标准输入成为管道的读端，然后 exec用户喜爱的分页程序。程序 14-2显示了如何实现这些操作。（本例要求在命令行中有一个参数说明要显示的文件的名称。通常，这种类型的程序要求在终端上显示的数据已经在存储器中。）

程序14-2 将文件复制到分页程序

```

#include <sys/wait.h>
#include "ourhdr.h"

#define DEF_PAGER "/usr/bin/more" /* default pager program */

int
main(int argc, char *argv[])
{
    int n, fd[2];
    pid_t pid;
    char line[MAXLINE], *pager, *argv0;
    FILE *fp;

    if (argc != 2)
        err_quit("usage: a.out <pathname>");
    if ( (fp = fopen(argv[1], "r")) == NULL)
        err_sys("can't open %s", argv[1]);

    if (pipe(fd) < 0)
        err_sys("pipe error");

    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid > 0) { /* parent */
        close(fd[0]); /* close read end */
        /* parent copies argv[1] to pipe */
        while (fgets(line, MAXLINE, fp) != NULL) {
            n = strlen(line);
            if (write(fd[1], line, n) != n)
                err_sys("write error to pipe");
        }
        if (ferror(fp))
            err_sys("fgets error");

        close(fd[1]); /* close write end of pipe for reader */
    }
}

```

```

    if (waitpid(pid, NULL, 0) < 0)
        err_sys("waitpid error");
    exit(0);
} else {
    /* child */
    close(fd[1]); /* close write end */
    if (fd[0] != STDIN_FILENO) {
        if (dup2(fd[0], STDIN_FILENO) != STDIN_FILENO)
            err_sys("dup2 error to stdin");
        close(fd[0]); /* don't need this after dup2 */
    }

    /* get arguments for execl() */
    if ( (pager = getenv("PAGER")) == NULL)
        pager = DEF_PAGER;
    if ( (argv0 = strrchr(pager, '/')) != NULL)
        argv0++; /* step past rightmost slash */
    else
        argv0 = pager; /* no slash in pager */

    if (execl(pager, argv0, (char *) 0) < 0)
        err_sys("execl error for %s", pager);
}
}

```

在调用fork之前先创建一个管道。fork之后父进程关闭其读端，子进程关闭其写端。子进程然后调用dup2，使其标准输入成为管道的读端。当执行分页程序时，其标准输入将是管道的读端。

当我们将一个描述符复制到另一个时（在子进程中，fd[0]复制到标准输入），应当注意该描述符的值并不已经是所希望的值。如果该描述符已经具有所希望的值，并且我们先调用dup2，然后调用close则将关闭此进程中只有该单个描述符所代表的打开文件。（回忆3.12节中所述，当dup2中的两个参数值相等时的操作。）在本程序中，如果shell没有打开标准输入，那么程序开始处的fopen应已使用描述符0，也就是最小未使用的描述符，所以fd[0]决不会等于标准输入。尽管如此，只要先调用dup2，然后调用close以复制一个描述符到另一个，作为一种保护性的编程措施，我们总是先将两个描述符进行比较。

请注意，我们是如何使用环境变量PAGER获得用户分页程序名称的。如果这种操作没有成功，则使用系统默认值。这是环境变量的常见用法。

实例

回忆8.8节中的五个函数：TELL_WAIT、TELL_PARENT、TELL_CHILD、WAIT_PARENT以及WAIT_CHILD。程序10-17提供了一个使用信号的实现。程序14-3则是一个使用管道的实现。

见图14-4，在fork之前创建了两个管道。

父进程在调用TELL_CHILD时经由上一个管道写一个字符“P”，子进程在调用TELL_PARENT时，经由下一个管道写一个字符“C”。相应的WAIT_XXX函数调用read读一个字符，没有读到字符时阻塞（睡眠等待）。

请注意，每一个管道都有一个额外的读取进程，这没有关系。也就是说除了子进程从pfd1[0]读取，父进程也有上一个管道的读端。因为父进程并没有执行对该管道的读操作，所以这不会产生任何影响。

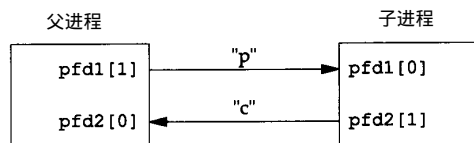


图14-4 用两个管道实现父-子进程的同步

程序14-3 使父、子进程同步的例程

```
#include "ourhdr.h"

static int pfd1[2], pfd2[2];

void
TELL_WAIT(void)
{
    if (pipe(pfd1) < 0 || pipe(pfd2) < 0)
        err_sys("pipe error");
}

void
TELL_PARENT(pid_t pid)
{
    if (write(pfd2[1], "c", 1) != 1)
        err_sys("write error");
}

void
WAIT_PARENT(void)
{
    char c;

    if (read(pfd1[0], &c, 1) != 1)
        err_sys("read error");
    if (c != 'p')
        err_quit("WAIT_PARENT: incorrect data");
}

void
TELL_CHILD(pid_t pid)
{
    if (write(pfd1[1], "p", 1) != 1)
        err_sys("write error");
}

void
WAIT_CHILD(void)
{
    char c;

    if (read(pfd2[0], &c, 1) != 1)
        err_sys("read error");
    if (c != 'c')
        err_quit("WAIT_CHILD: incorrect data");
}
```

14.3 popen和pclose函数

因为常见的操作是创建一个连接到另一个进程的管道，然后读其输出或向其发送输入，所以标准I/O库为实现这些操作提供了两个函数 `popen` 和 `pclose`。这两个函数实现的操作是：创建一个管道，`fork` 一个子进程，关闭管道的不使用端，`exec` 一个 shell 以执行命令，等待命令终止。

```
#include <stdio.h>

FILE *popen(const char*mdstring, const char type);
```

返回：若成功则为文件指针，若出错则为 NULL

```
int pclose(FILE fp*);
```

返回：*cmdstring*的终止状态，若出错则为-1

函数先执行fork，然后调用exec以执行*cmdstring*，并且返回一个标准I/O文件指针。如果*type*是"r"，则文件指针连接到*cmdstring*的标准输出（见图14-5）。

如果*type*是"w"，则文件指针连接到*cmdstring*的标准输入（见图14-6）。

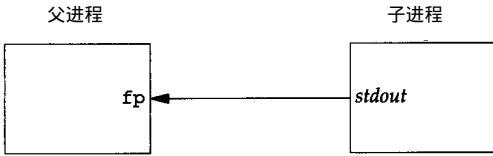


图14-5 fp=popen(*command*, "r")的结果



图14-6 fp=popen(*command*, "w")的结果

有一种方法可以帮助我们记住popen最后一个参数及其作用，这种方法就是与fopen进行类比。如果*type*是"r"，则返回的文件指针是可读的，如果*type*是"w"，则是可写的。

pclose函数关闭标准I/O流，等待命令执行结束，然后返回shell的终止状态。（我们曾在8.6节对终止状态进行过说明，system函数（见8.12节）也返回终止状态。）如果shell不能被执行，则pclose返回的终止状态与shell执行exit（127）一样。

cmdstring 由Bourne shell以下列方式执行；

```
sh -c cmdstring
```

这表示shell将扩展*cmdstring*中的任何特殊字符。例如，可以使用；

```
fp = popen("ls *.c", "r");
```

或者

```
fp = popen("cmd 2>&1", "r");
```

POSIX.1没有说明popen、pclose，因为它们与shell有交互作用，而shell是由POSIX.2说明的。我们对这两个函数的说明与POSIX.2的11.2草案相一致。该草案对这两个函数的说明与以前的实现有些区别。

实例

用popen重写程序14-2，其结果是程序14-4。使用popen减少了需要编写的代码量。

shell命令\${PAGER:-more}的意思是：如果shell变量PAGER已经定义，且其值非空，则使用其值，否则使用字符串more。

程序14-4 用popen向分页程序传送文件

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

#define PAGER "${PAGER:-more}" /* environment variable, or default */

int
main(int argc, char *argv[])
{
    char    line[MAXLINE];
    FILE    *fpin, *fpout;

    if (argc != 2)
        err_quit("usage: a.out <pathname>");
```

```

if ( (fpin = fopen(argv[1], "r")) == NULL)
    err_sys("can't open %s", argv[1]);

if ( (fpout = popen(PAGER, "w")) == NULL)
    err_sys("popen error");

    /* copy argv[1] to pager */
while (fgets(line, MAXLINE, fpin) != NULL) {
    if (fputs(line, fpout) == EOF)
        err_sys("fputs error to pipe");
}
if (ferror(fpin))
    err_sys("fgets error");
if (pclose(fpout) == -1)
    err_sys("pclose error");
exit(0);
}

```

实例——popen函数

程序14-5是我们编写的popen和pclose版本。虽然popen的核心部分与本章中以前用过的代码类似，但是增加了很多需要考虑的细节。首先每次调用 popen时，应当记住所创建的子进程的进程ID，以及其文件描述符或FILE指针。我们选择在数组 childpid中保存子进程ID，并用文件描述符作为其下标。于是，当以 FILE指针作为参数调用 pclose时，我们调用标准 I/O函数 fileno以得到文件描述符，然后取得子进程 ID，并用于调用 waitpid。因为一个进程可能调用 popen多次，所以在动态分配 childpid数组时（第一次调用 popen时），其长度可以容纳与文件描述符数相同的进程数。

调用pipe、fork以及为每个进程复制相应的文件描述符，这些操作与本章前面所述的类似。

POSIX.2要求子进程关闭在以前调用 popen时形成，当前仍旧打开的所有 I/O流。为此，在子进程中从头逐个检查 childpid数组的各元素，关闭仍旧打开的任一描述符。

若pclose的调用者已经为信号 SIGCHLD设置了一个信号处理程序，则 waitpid将返回一个 EINTR。因为允许调用者捕捉此信号（或者任何其他可能中断 waitpid调用的信号），所以当 waitpid被一个捕捉到的信号中断时，我们只是再次调用 waitpid。

如果一个信号中断了wait，pclose的早期版本返回EINTR。

pclose的早期版本在 wait期间，阻塞或忽略信号 SIGINT、SIGQUIT以及 SIGHUP。POSIX.2则不允许这一点。

程序14-5 popen和pclose函数

```

#include <sys/wait.h>
#include <errno.h>
#include <fcntl.h>
#include "ourhdr.h"

static pid_t *childpid = NULL;
/* ptr to array allocated at run-time */
static int maxfd; /* from our open_max(), Program 2.3 */

#define SHELL "/bin/sh"

FILE *
popen(const char *cmdstring, const char *type)

```

```

{
    int    i, pfd[2];
    pid_t  pid;
    FILE   *fp;

    /* only allow "r" or "w" */
    if ((type[0] != 'r' && type[0] != 'w') || type[1] != 0) {
        errno = EINVAL; /* required by POSIX.2 */
        return(NULL);
    }

    if (childpid == NULL) { /* first time through */
        /* allocate zeroed out array for child pids */
        maxfd = open_max();
        if ( (childpid = calloc(maxfd, sizeof(pid_t))) == NULL)
            return(NULL);
    }

    if (pipe(pfd) < 0)
        return(NULL); /* errno set by pipe() */

    if ( (pid = fork()) < 0)
        return(NULL); /* errno set by fork() */
    else if (pid == 0) { /* child */
        if (*type == 'r') {
            close(pfd[0]);
            if (pfd[1] != STDOUT_FILENO) {
                dup2(pfd[1], STDOUT_FILENO);
                close(pfd[1]);
            }
        } else {
            close(pfd[1]);
            if (pfd[0] != STDIN_FILENO) {
                dup2(pfd[0], STDIN_FILENO);
                close(pfd[0]);
            }
        }

        /* close all descriptors in childpid[] */
        for (i = 0; i < maxfd; i++)
            if (childpid[i] > 0)
                close(i);

        execl(SHELL, "sh", "-c", cmdstring, (char *) 0);
        _exit(127);
    }

    /* parent */
    if (*type == 'r') {
        close(pfd[1]);
        if ( (fp = fdopen(pfd[0], type)) == NULL)
            return(NULL);
    } else {
        close(pfd[0]);
        if ( (fp = fdopen(pfd[1], type)) == NULL)
            return(NULL);
    }
    childpid[fileno(fp)] = pid; /* remember child pid for this fd */
    return(fp);
}

int
pclose(FILE *fp)
{
    int    fd, stat;
    pid_t  pid;

```



```

if (childpid == NULL)
    return(-1);    /* popen() has never been called */

fd = fileno(fp);
if ( (pid = childpid[fd]) == 0)
    return(-1);    /* fp wasn't opened by popen() */

childpid[fd] = 0;
if (fclose(fp) == EOF)
    return(-1);

while (waitpid(pid, &stat, 0) < 0)
    if (errno != EINTR)
        return(-1); /* error other than EINTR from waitpid() */

return(stat);    /* return child's termination status */
}

```

实例

考虑一个应用程序，它向标准输出写一个提示，然后从标准输入读1行。使用popen，可以在应用程序和输入之间插入一个程序以对输入进行变换处理。图14-7显示了进程的安排。

对输入进行的变换可能是路径名的扩充，或者是提供一种历史机制（记住以前输入的命令）。（本实例取自POSIX.2草案。）

程序14-6是一个简单的过滤程序，它只是将输入复制到输出，在复制时将任一大写字母变换为小写字母。在写了一行之后，对标准输出进行了刷清（用fflush），其理由将在下一节介绍协同进程时讨论。

程序14-6 过滤程序，将大写字母变换成小写字母

```

#include <ctype.h>
#include "ourhdr.h"

int
main(void)
{
    int c;

    while ( (c = getchar()) != EOF) {
        if (isupper(c))
            c = tolower(c);
        if (putchar(c) == EOF)
            err_sys("output error");
        if (c == '\n')
            fflush(stdout);
    }
    exit(0);
}

```

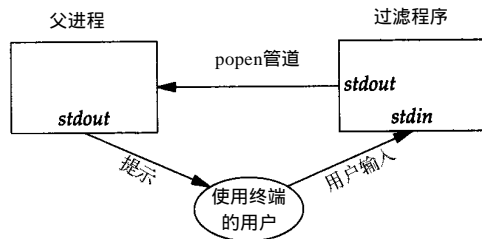


图14-7 用popen变换输入

对该过滤程序进行编译，其可执行目标代码存放在文件 myuclc中，然后在程序14-7中用popen调用它们。

因为标准输出通常是按行进行缓存的，而提示并不包含新行符，所以在写了提示之后，需要调用fflush。

程序14-7 调用大写/小写过滤程序以读取命令

```

#include <sys/wait.h>
#include "ourhdr.h"

int
main(void)
{
    char    line[MAXLINE];
    FILE    *fpin;

    if ( (fpin = popen("myuclc", "r")) == NULL)
        err_sys("popen error");

    for ( ; ; ) {
        fputs("prompt> ", stdout);
        fflush(stdout);
        if (fgets(line, MAXLINE, fpin) == NULL) /* read from pipe */
            break;
        if (fputs(line, stdout) == EOF)
            err_sys("fputs error to pipe");
    }
    if (pclose(fpin) == -1)
        err_sys("pclose error");
    putchar('\n');
    exit(0);
}

```

14.4 协同进程

UNIX过滤程序从标准输入读取数据，对其进行适当处理后写到标准输出。几个过滤进程通常在shell管道命令中线性地连接。当同一个程序产生某个过滤程序的输入，同时又读取该过滤程序的输出时，则该过滤程序就成为协同进程 (coprocess)。

KornShell提供了协同进程。Bourne shell和C shell并没有提供将进程连接起来按协同进程方式工作的方法。协同进程通常在shell的后台运行，其标准输入和标准输出通过管道连接到另一个程序。虽然要求初始化一个协同进程，并将其输入和输出连接到另一个进程的shell语法是十分奇特的（详细情况见Bolsky和Korn [1989] 中的pp.66~66），但是协同进程的工作方式在C程序中也是非常有用的。

`popen`提供连接到另一个进程的标准输入或标准输出的一个单行管道，而对于协同进程，则它有连接到另一个进程的两个单行管道——一个接到其标准输入，另一个则来自标准输出。我们先要将数据写到其标准输入，经其处理后，再从其标准输出读取数据。

实例

让我们通过一个实例来观察协同进程。进程先创建两个管道：一个是协同进程的标准输入，另一个是协同进程的标准输出。图14-8显示了这种安排。

程序14-8是一个简单的协同进程，它从其标准输入读两个数，计算它们的和，然后将结果写至标准输出。



图14-8 驱动一个协同进程——写其标准输入，读其标准输出

程序14-8 对两个数求和的简单过滤程序

```

#include "ourhdr.h"

int

```

```

main(void)
{
    int    n, int1, int2;
    char   line[MAXLINE];

    while ( (n = read(STDIN_FILENO, line, MAXLINE)) > 0) {
        line[n] = 0;          /* null terminate */
        if (sscanf(line, "%d%d", &int1, &int2) == 2) {
            sprintf(line, "%d\n", int1 + int2);
            n = strlen(line);
            if (write(STDOUT_FILENO, line, n) != n)
                err_sys("write error");
        } else {
            if (write(STDOUT_FILENO, "invalid args\n", 13) != 13)
                err_sys("write error");
        }
    }
    exit(0);
}

```

对此程序进行编译，将其可执行目标代码存入名为 add2 的文件。

程序 14-9 在从其标准输入读入两个数之后调用 add2 协同进程。从协同进程送来的值则写到其标准输出。

程序 14-9 驱动 add2 过滤程序的程序

```

#include    <signal.h>
#include    "ourhdr.h"

static void sig_pipe(int);      /* our signal handler */

int
main(void)
{
    int    n, fd1[2], fd2[2];
    pid_t  pid;
    char   line[MAXLINE];

    if (signal(SIGPIPE, sig_pipe) == SIG_ERR)
        err_sys("signal error");

    if (pipe(fd1) < 0 || pipe(fd2) < 0)
        err_sys("pipe error");

    if ( (pid = fork()) < 0)
        err_sys("fork error");

    else if (pid > 0) {          /* parent */
        close(fd1[0]);
        close(fd2[1]);
        while (fgets(line, MAXLINE, stdin) != NULL) {
            n = strlen(line);
            if (write(fd1[1], line, n) != n)
                err_sys("write error to pipe");
            if ( (n = read(fd2[0], line, MAXLINE)) < 0)
                err_sys("read error from pipe");
            if (n == 0) {
                err_msg("child closed pipe");
                break;
            }
            line[n] = 0;        /* null terminate */
            if (fputs(line, stdout) == EOF)
                err_sys("fputs error");
        }
    }
}

```

```

    }
    if (ferror(stdin))
        err_sys("fgets error on stdin");
    exit(0);
} else {
    /* child */
    close(fd1[1]);
    close(fd2[0]);
    if (fd1[0] != STDIN_FILENO) {
        if (dup2(fd1[0], STDIN_FILENO) != STDIN_FILENO)
            err_sys("dup2 error to stdin");
        close(fd1[0]);
    }
    if (fd2[1] != STDOUT_FILENO) {
        if (dup2(fd2[1], STDOUT_FILENO) != STDOUT_FILENO)
            err_sys("dup2 error to stdout");
        close(fd2[1]);
    }
    if (execl("./add2", "add2", (char *) 0) < 0)
        err_sys("execl error");
}
}

static void
sig_pipe(int signo)
{
    printf("SIGPIPE caught\n");
    exit(1);
}

```

在程序中创建了两个管道，父、子进程各自关闭它们不需使用的端口。创建两个管道的理由是：一个用做协同进程的标准输入，另一个则用做它的标准输出。然后在调用 `execl` 之前，子进程调用 `dup2` 使管道描述符移至其标准输入和输出。

若编译和运行程序 14-9，它如所希望的那样进行工作。并且，当程序 14-9 正等待输入时，若杀死 `add2` 协同进程，然后输入两个数，当程序 14-9 对管道进行写操作时，由于该管道无读进程，于是调用信号处理程序（见习题 14.4）。

程序 15-1 将提供这一实例的另一个版本，它使用一个全双工管道而不是两个半双工管道。

实例

在协同进程 `add2`（见程序 14-8）中，使用了 UNIX I/O: `read` 和 `write`。如果使用标准 I/O 改写该协同进程，其后果是什么呢？程序 14-10 即改写后的版本。

程序 14-10 对两个数求和的滤波程序，使用标准 I/O

```

#include    "ourhdr.h"

int
main(void)
{
    int    int1, int2;
    char    line[MAXLINE];

    while (fgets(line, MAXLINE, stdin) != NULL) {
        if (sscanf(line, "%d%d", &int1, &int2) == 2) {
            if (printf("%d\n", int1 + int2) == EOF)
                err_sys("printf error");
        } else {
            if (printf("invalid args\n") == EOF)

```

```

        err_sys("printf error");
    }
}
exit(0);
}

```

若程序 14-9 调用此新的协同进程，则它不再工作。问题出在系统默认的标准 I/O 缓存机制上。当程序 14-10 被调用时，对标准输入的第一个 `fgets` 引起标准 I/O 库分配一个缓存，并选择缓存的类型。因为标准输入是个管道，所以 `isatty` 为假，于是标准 I/O 库由系统默认是全缓存的。对标准输出也有同样的处理。当 `add2` 从其标准输入读取而发生堵塞时，程序 14-9 从管道读时也发生堵塞，于是产生了死锁。

对将要执行的这样一个协同进程可以加以控制。在程序 4-10 中的 `while` 循环之前加上下面 4 行：

```

if (setvbuf(stdin, NULL, _IOLBF, 0) != 0)
    err_sys("setvbuf error");
if (setvbuf(stdout, NULL, _IOLBF, 0) != 0)
    err_sys("setvbuf error");

```

这使得当有一行可用时，`fgets` 即返回，并使得当输出一新行符时，`printf` 即执行 `fflush` 操作。对 `setvbuf` 进行了这些显式调用，使得程序 14-10 能正常工作。

如果不能修改程序，则需使用其他技术。例如，如果在程序中使用 `awk(1)` 代替 `add2` 作为协同进程，则下列命令行不能工作：

```

#!/bin/awk/ -f
{ print $1 + $2 }

```

不能工作的原因还是标准 I/O 的缓存机制问题。但是，在这种情况下不能改变 `awk` 的工作方式（除非有 `awk` 的源代码）。

对这种问题的一般解决方法是使被调用（在本例中是 `awk`）的协同进程认为它的标准输入和输出被连接到一个终端。这使得协同进程中的标准 I/O 例程对这两个 I/O 流进行行缓存，这类似于前面所做的显式 `setvbuf` 调用。第 19 章将用伪终端实现这一点。

14.5 FIFO

FIFO 有时被称为命名管道。管道只能由相关进程使用，它们共同的祖先进程创建了管道。但是，通过 FIFO，不相关的进程也能交换数据。

第 14 章已经提及 FIFO 是一种文件类型。`stat` 结构（见 4.2 节）成员 `st_mode` 的编码指明文件是否是 FIFO 类型。可以用 `S_ISFIFO` 宏对此进行测试。

创建 FIFO 类似于创建文件。确实，FIFO 的路径名存在于文件系统中。

```

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

int mkfifo(const char *pathname, mode_t mode);

```

返回：若成功则为 0，若出错则为 -1

`mkfifo` 函数中 `mode` 参数的规格说明与 `open` 函数中的 `mode` 相同（见 3.3 节）。新 FIFO 的用户和组的所有权与 4.6 节所述的相同。

一旦已经用 `mkfifo` 创建了一个 FIFO，就可用 `open` 打开它。确实，一般的文件 I/O 函数（`close`、`read`、`write`、`unlink` 等）都可用于 FIFO。

mkfifo是POSIX.1首先提出的。SVR3用mknod(2)系统调用创建FIFO。而在SVR4中，mkfifo调用mknod创建FIFO。

POSIX.2已经建议了一个mkfifo(1)命令。SVR4和4.3+BSD现在支持此命令。于是，用一条shell命令就可以创建一个FIFO，然后用一般的shell I/O重新定向对其进行存取。

当打开一个FIFO时，非阻塞标志(O_NONBLOCK)产生下列影响：

(1) 在一般情况下(没有说明O_NONBLOCK)，只读打开要阻塞到某个其他进程为写打开此FIFO。类似，为写而打开一个FIFO要阻塞到某个其他进程为读而打开它。

(2) 如果指定了O_NONBLOCK，则只读打开立即返回。但是，如果没有进程已经为读而打开一个FIFO，那么只写打开将出错返回，其errno是ENXIO。

类似于管道，若写一个尚无进程为读而打开的FIFO，则产生信号SIGPIPE。若某个FIFO的最后一个写进程关闭了该FIFO，则将为该FIFO的读进程产生一个文件结束标志。

一个给定的FIFO有多个写进程是常见的。这就意味着如果不希望多个进程所写的数据互相穿插，则需考虑原子写操作。正如对于管道一样，常数PIPE_BUF说明了可被原子写到FIFO的最大数据量。

FIFO有两种用途：

(1) FIFO由shell命令使用以便将数据从一条管道线传送到另一条，为此无需创建中间临时文件。

(2) FIFO用于客户机-服务器应用程序中，以在客户机和服务器之间传递数据。

我们各用一个例子来说明这两种用途。

实例——用FIFO复制输出流

FIFO可被用于复制串行管道命令之间的输出流，于是也就不需要写数据到中间磁盘文件中(类似于使用管道以避免中间磁盘文件)。

管道只能用于进程间的线性连接，然而，因为FIFO具有名字，所以它可用于非线性连接。

考虑这样一个操作过程，它需要对一个经过过滤的输入流进行两次处理。图14-9表示了这种安排。

使用FIFO以及UNIX程序tee(1)，就可以实现这样的过程而无需使用临时文件。(tee程序将其标准输入同时复制到其标准输

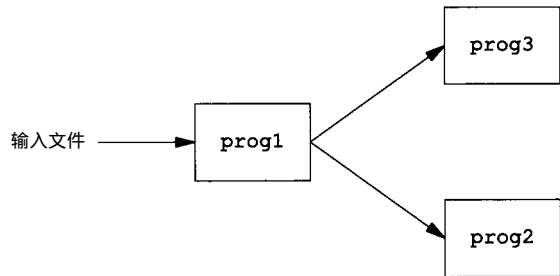


图14-9 对一个经过过滤的输入流进行两次处理

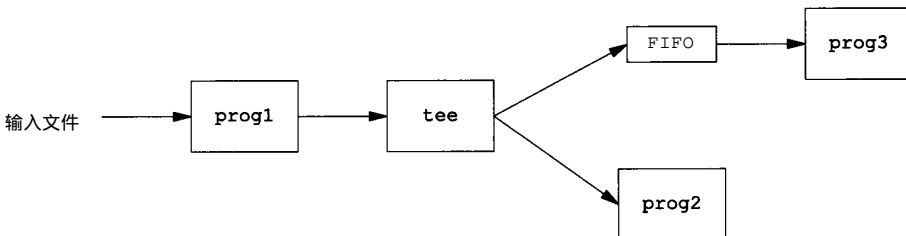


图14-10 使用FIFO和tee将一个流发送到两个不同的进程

出以及其命令行中包含的命名文件中。)

```
mkfifo fifol
prog3 < fifol &
prog1 < infile | tee fifol | prog2
```

创建FIFO，然后在后台起动prog3，它从FIFO读数据。然后起动prog1，用tee将其输出发送到FIFO和prog2。图14-10显示了有关安排。

实例——客户-服务器使用FIFO进行通信

FIFO的另一个应用是在客户机和服务器之间传送数据。如果有一个服务器，它与很多客户机有关，每个客户机都可将其请求写到一个该服务器创建的众所周知的FIFO中（“众所周知”的意思是；所有需与服务器联系的客户机都知道该FIFO的路径名）。图14-11显示了这种安排。因为对于该FIFO有多个写进程，客户机发送给服务器的请求其长度要小于PIPE_BUF字节。这样就能避免客户机各次写之间的穿插。

在这种类型的客户机-服务器通信中使用FIFO的问题是：服务器如何将回答送回各个客户机。不能使用单个FIFO，因为服务器会发出对各个客户机请求的响应，而请求者却不可能知道什么时候去读才能恰恰得到对它的响应。一种解决方法是每个客户机都在其请求中发送其进程ID。然后服务器为每个客户机创建一个FIFO，所使用的路径名是以客户机的进程ID为基础的。例如，服务器可以用名字/tmp/serv1.XXXXX创建FIFO，其中XXXXX被替换成客户机的进程ID。图14-12显示了这种安排。

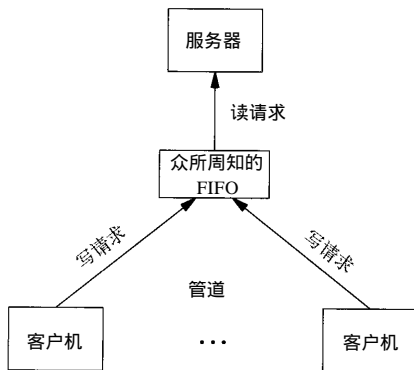


图14-11 客户机用FIFO向服务器发送请求

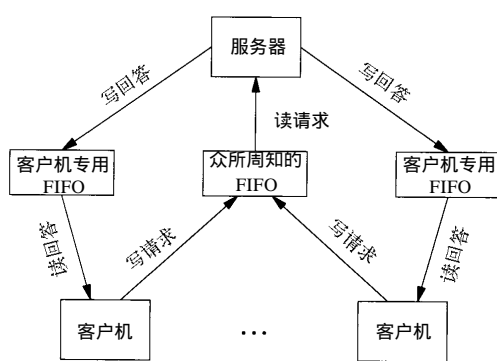


图14-12 客户机-服务器用FIFO进行通信

这种安排可以工作，但也有一些不足之处。其中之一是服务器不能判断一个客户机是否崩溃终止，这就使得客户机专用的FIFO会遗留在文件系统中。另一个是服务器必须捕捉SIGPIPE信号，因为客户机在发送一个请求后没有读取响应就可能终止，于是留下一个有写进程（服务器）而无读进程的客户机专用FIFO。

按照图14-12中的安排，如果服务器以只读方式打开众所周知的FIFO（因为它只需读该FIFO），则每次客户机数从1变成0，服务器就将在FIFO中读到一个文件结束标记。为使服务器免于处理这种情况，一种常见的技巧是使服务器以读-写方式打开该FIFO（见习题14.10）。

14.6 系统V IPC

三种系统V IPC：消息队列、信号量以及共享存储器之间有很多相似之处。以下各节将说明这些IPC的各自特殊功能，本节先介绍它们类似的特征。

这三种IPC源自于1970年的一种称为Columbus UNIX的UNIX内部版本。后来它们被加到SV上。

14.6.1 标识符和关键字

每个内核中的IPC结构（消息队列、信号量或共享存储段）都用一个非负整数的标识符（identifier）加以引用。例如，为了对一个消息队列发送或取消息，只需知道其队列标识符。与文件描述符不同，IPC标识符不是小的整数。当一个IPC结构被创建，以后又被删除时，与这种结构相关的标识符连续加1，直至达到一个整型数的最大正值，然后又回转到0。（即使在IPC结构被删除后也记住该值，每次使用此结构时则增1，该值被称为“槽使用顺序号”。它在ipc_perm结构中，下一节将说明此结构。）

无论何时创建IPC结构（调用msgget、semget或shmget），都应指定一个关键字（key），关键字的数据类型由系统规定为key_t，通常在头文件<sys/types.h>中被规定为长整型。关键字由内核变换成标识符。

有多种方法使客户机和服务器在同一IPC结构上会合：

(1) 服务器可以指定关键字IPC_PRIVATE创建一个新IPC结构，将返回的标识符存放在某处（例如一个文件）以便客户机取用。关键字IPC_PRIVATE保证服务器创建一个新IPC结构。这种技术的缺点是：服务器要将整型标识符写到文件中，然后客户机在此后又读文件取得此标识符。

IPC_PRIVATE关键字也可用于父、子关系进程。父进程指定IPC_PRIVATE创建一个新IPC结构，所返回的标识符在fork后可由子进程使用。子进程可将此标识符作为exec函数的一个参数传给一个新程序。

(2) 在一个公用头文件中定义一个客户机和服务器都认可的关键字。然后服务器指定此关键字创建一个新的IPC结构。这种方法的问题是关键字可能已与一个IPC结构相结合，在此情况下，get函数（msgget、semget或shmget）出错返回。服务器必须处理这一错误，删除已存在的IPC结构，然后试着再创建它。

(3) 客户机和服务器认同一个路径名和课题ID（课题ID是0~255之间的字符值），然后调用函数ftok将这两个值变换为一个关键字（函数ftok在手册页stdipc（3）中说明）。然后在方法(2)中使用此关键字。ftok提供的唯一服务就是由一个路径名和课题ID产生一个关键字。因为一般来说，客户机和服务器至少共享一个头文件，所以一个比较简单的方法是避免使用ftok，而只是在该头文件中存放一个大家都知道的关键字。这样做还避免了使用另一个函数。

三个get函数（msgget、semget和shmget）都有两个类似的参数key和一个整型的flag。如若满足下列条件，则创建一个新的IPC结构（通常由服务器创建）：

(1) key是IPC_PRIVATE，或

(2) key当前未与特定类型的IPC结构相结合，flag中指定了IPC_CREAT位。为访问现存的队列（通常由客户机进行），key必须等于创建该队列时所指定的关键字，并且不应指定IPC_CREAT。

注意，为了访问一个现存队列，决不能指定IPC_PRIVATE作为关键字。因为这是一个特殊的键值，它总是用于创建一个新队列。为了访问一个用IPC_PRIVATE关键字创建的现存队列，一定要知道与该队列相结合的标识符，然后在其他IPC调用中（例如msgsnd、msgrcv）使用该标识符。

如果希望创建一个新的IPC结构，保证不是引用具有同一标识符的一个现行IPC结构，那么必须在`flag`中同时指定`IPC_CREAT`和`IPC_EXCL`位。这样做了以后，如果IPC结构已经存在就会造成出错，返回`EEXIST`（这与指定了`O_CREAT`和`O_EXCL`标志的`open`相类似）。

14.6.2 许可权结构

系统V IPC为每一个IPC结构设置了一个`ipc_perm`结构。该结构规定了许可权和所有者。

```
struct ipc_perm {
    uid_t    uid;           /* owner's effective user id */
    gid_t    gid;           /* owner's effective group id */
    uid_t    cuid;         /* creator's effective user id */
    gid_t    cgid;         /* creator's effective group id */
    mode_t    mode;         /* access modes */
    ulong    seq;           /* slot usage sequence number */
    key_t    key;           /* key */
}
```

在创建IPC结构时，除`seq`以外的所有字段都赋初值。以后，可以调用`msgctl`、`semctl`或`shmctl`修改`uid`、`gid`和`mode`字段。为了改变这些值，调用进程必须是IPC结构的创建者或超级用户。更改这些字段类似于对文件调用`chown`和`chmod`。

`mode`字段的值类似于表4-4中所示的值，但是对于任何IPC结构都不存在执行许可权。另外，消息队列和共享存储使用术语“读”和“写”，而信号量则用术语“读”和“更改”。表14-2中对每种IPC说明了6种许可权。

表14-2 系统V IPC许可权

许可权	消息队列	信号量	共享存储
用户读	<code>MSG_R</code>	<code>SEM_R</code>	<code>SHM_R</code>
用户写(更改)	<code>MSG_W</code>	<code>SEM_A</code>	<code>SHM_W</code>
组读	<code>MSG_R >> 3</code>	<code>SEM_R >> 3</code>	<code>SHM_R >> 3</code>
组写(更改)	<code>MSG_W >> 3</code>	<code>SEM_A >> 3</code>	<code>SHM_W >> 3</code>
其他读	<code>MSG_R >> 6</code>	<code>SEM_R >> 6</code>	<code>SHM_R >> 6</code>
其他写(更改)	<code>MSG_W >> 6</code>	<code>SEM_A >> 6</code>	<code>SHM_W >> 6</code>

14.6.3 结构限制

三种形式的系统V IPC都有我们可能会遇到的内在限制。这些限制的大多数可以通过重新配置而加以更改。当叙述每种IPC时，都会指出它的限制

在SVR4中，这些值以及它们的最小、最大值都在文件`/etc/conf/cf.d/mtune`中。

14.6.4 优点和缺点

系统V IPC的主要问题是；IPC结构是在系统范围内起作用的，没有访问计数。例如，如果创建了一个消息队列，在该队列中放入了几则消息，然后终止，但是该消息队列及其内容并不被删除。它们余留在系统中直至：由某个进程调用`msgrcv`或`msgctl`读消息或删除消息队列，或某个进程执行`ipcrm(1)`命令删除消息队列；或由正在再起动的系统删除消息队列。将此与管道`pipe`相比，那么当最后一个访问管道的进程终止时，管道就被完全地删除了。对于FIFO而言虽

然当最后一个引用 FIFO 的进程终止时其名字仍保留在系统中，直至显式地删除它，但是留在 FIFO 中的数据却在此时全部删除。

系统 V IPC 的另一个问题是；这些 IPC 结构并不按名字为文件系统所知。我们不能用第 3、4 章中所述的函数来存取它们或修改它们的特性。为了支持它们不得不增加了十多个全新的系统调用（msgget、semop、shmat 等）。我们不能用 ls 命令见到它们，不能用 rm 命令删除它们，不能用 chmod 命令更改它们的存取权。于是，也不得不增加了全新的命令 ipcs 和 ipcrm。

因为这些 IPC 不使用文件描述符，所以不能对它们使用多路转接 I/O 函数：select 和 poll。这就使得一次使用多个 IPC 结构，以及用文件或设备 I/O 来使用 IPC 结构很难做到。例如，没有某种形式的忙-等待循环，就不能使一个服务器等待一个消息放在两个消息队列的任一个中。

Andrade、Carges 以及 Kovach [1989] 对使用系统 V IPC 的一个实际事务处理系统进行了综述。他们认为系统 V IPC 使用的名字空间（标识符）是一个优点而不是前面所说的问题，理由是使用标识符使一个进程只要使用单个函数调用（msgsnd）就能将一个消息发送到一个队列，而其他形式的 IPC 则通常要求 open、write 和 close。这种论据是不真实的。为了避免使用一个关键字和调用 msgget，客户机总要以某种方式获得服务器队列的标识符。分派给特定队列的标识符取决于在创建该队列时，有多少消息队列已经存在，取决于自内核自举以来，内核中将分配给新队列的表项已经使用了多少次。这是一个动态值，不能被猜测或事先存放在一个头文件中。正如 14.6.1 节所述，至少服务器应将分配给队列的标识符写到一个文件中以便客户机读取。

这些作者列举的消息队列的其他优点是：(a) 它们是可靠的，(b) 流是受到控制的，(c) 面向记录，(d) 可以用非先进先出方式处理。正如在 12.4 节中所见，流也具有所有这些优点，虽然在向一个流发送数据之前，需要一个 open，在结束时需要一个 close。表 14-3 对这些不同形式的 IPC 的某些特征进行了比较。

表 14-3 不同形式 IPC 之间特征的比较

类 型	无连接?	可靠?	流控制?	记录?	消息类型或优先权?
消息队列	否	是	是	是	是
流	否	是	是	是	是
UNIX 流套接口	否	是	是	否	否
UNIX 数据报套接口	是	是	否	是	否
FIFO	否	是	是	否	否

(第 15 章将对 UNIX 流和数据报套接口进行简要说明。) 表中的“无连接”指的是无需先调用某种形式的 open，就能发送消息的能力。正如前述，因为需要有某种技术以获得队列标识符，所以我们并不认为消息队列具有无连接特性。因为所有这些形式的 IPC 都限制用在单主机上，所以它们都是可靠的。当消息通过网络传送时，丢失消息的可能性就要加以考虑。流控制的意思是：如果系统资源短缺（缓存）或者如果接收进程不能再接收更多消息，则发送进程就要睡眠。当流控制条件消失时，发送进程应自动地被唤醒。

表 14-3 中没有表示的一个特征是：IPC 设施能否自动地为每个客户机自动地创建一个到服务器的唯一连接。第 15 章将说明，流以及 UNIX 流套接口可以提供这种能力。

下面三节顺次对三种形式的系统 V IPC 进行详细说明。

14.7 消息队列

消息队列是消息的链接表，存放在内核中并由消息队列标识符标识。我们将称消息队列为

“队列”，其标识符为“队列ID”。msgget用于创建一个新队列或打开一个现存的队列。msgsnd用于将新消息添加到队列尾端。每个消息包含一个正长整型类型字段，一个非负长度以及实际数据字节（对应于长度），所有这些都将在将消息添加到队列时，传送给msgsnd。msgrcv用于从队列中取消息。我们并不一定要以先进先出次序取消息，也可以按消息的类型字段取消息。

每个队列都有一个msqid_ds结构与其相关。此结构规定了队列的当前状态。

```
struct msqid_ds {
    struct ipc_perm  msg_perm; /* see Section 14.6.2 */
    struct msg       *msg_first; /* ptr to first message on queue */
    struct msg       *msg_last; /* ptr to last message on queue */
    ulong           msg_cbytes; /* current # bytes on queue */
    ulong           msg_qnum; /* # of messages on queue */
    ulong           msg_qbytes; /* max # of bytes on queue */
    pid_t           msg_lspid; /* pid of last msgsnd() */
    pid_t           msg_lrpid; /* pid of last msgrcv() */
    time_t          msg_stime; /* last-msgsnd() time */
    time_t          msg_rtime; /* last-msgrcv() time */
    time_t          msg_ctime; /* last-change time */
};
```

两个指针msg-first和msg-last分别指向相应消息在内核中的存放位置，所以它们对用户进程而言是无价值的。结构的其他成员是自定义的。

表14-4列出了影响消息队列的系统限制（见14.6.3节）。

表14-4 影响消息队列的系统限制

名字	说明	典型值
MSGMAX	可发送的最长消息的字节长度	2048
MSGMNB	特定队列的最大字节长度（亦即队列中所有消息之和）	4096
MSGMNI	系统中最大消息队列数	50
MSGTOL	系统中最大消息数	50

调用的第一个函数通常是msgget，其功能是打开一个现存队列或创建一个新队列。

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgget(key_t key, int flag);
```

返回：若成功则为消息队列ID，若出错则为-1

14.6节说明了将key变换成一个标识符的规则，并且讨论是否创建一个新队列或访问一个现存队列。当创建一个新队列时，初始化msqid_ds结构的下列成员：

- ipc-perm结构按14.6.2节中所述进行初始化。该结构中mode按flag中的相应许可权位设置。这些许可权用表14-2中的常数指定。

- msg_qnum、msg_lspid、msg_lrpid、msg_stime和msg_rtime都设置为0。
- msg_ctime设置为当前时间。
- msg_qbytes设置为系统限制值。

若执行成功，则返回非负队列ID。此后，此值就可被用于其他三个消息队列函数。

`msgctl`函数对队列执行多种操作。它以及另外两个与信号量和共享存储有关的函数 (`semctl`和`shmctl`)是系统V IPC的类似于`ioctl`的函数 (亦即垃圾桶函数)。

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgctl(int msqid, int cmd, struct msqid_dsbuf);
```

返回：若成功则为0，出错则为-1

`cmd`参数指定对于由`msqid`规定的队列要执行的命令：

- `IPC_STAT` 取此队列的`msqid_ds`结构，并将其存放在`buf`指向的结构中。
- `IPC_SET` 按由`buf`指向的结构中的值，设置与此队列相关的结构中的下列四个字段：`msg_perm.uid`、`msg_perm.gid`、`msg_perm.mode`和`msg_qbytes`。此命令只能由下列两种进程执行：一种是其有效用户ID等于`msg_perm.cuid`或`msg_perm.uid`；另一种是具有超级用户特权的进程。只有超级用户才能增加`msg_qbytes`的值
- `IPC_RMID` 从系统中删除该消息队列以及仍在该队列上的所有数据。这种删除立即生效。仍在用这一消息队列的其他进程在它们下一次试图对此队列进行操作时，将出错返回`EIDRM`。此命令只能由下列两种进程执行：一种是其有效用户ID等于`msg_perm.cuid`或`msg_perm.uid`；另一种是具有超级用户特权的进程。

这三条命令 (`IPC_STAT`、`IPC_SET`和`IPC_RMID`) 也可用于信号量和共享存储。

调用`msgsnd`将数据放到消息队列上。

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgsnd(int msqid, const void ptr, size_t nbytes, int flag);
```

返回：若成功则为0，若出错则为-1

正如前面提及的，每个消息都由三部分组成，它们是：正长整型类型字段、非负长度 (`nbytes`) 以及实际数据字节 (对应于长度)。消息总是放在队列尾端。

`ptr`指向一个长整型数，它包含了正整型消息类型，在其后立即跟随了消息数据。(若`nbytes`是0，则无消息数据。) 若发送的最长消息是512字节，则可定义下列结构：

```
struct mymsg {
    long mtype; /* positive message type */
    char mtext[512]; /* message data of length nbytes */
};
```

于是，`ptr`就是一个指向`mymsg`结构的指针。接收者可以使用消息类型以非先进先出的次序取消息。

`flag`的值可以指定为`IPC_NOWAIT`。这类似于文件I/O的非阻塞I/O标志 (见12.2节)。若消息队列已满 (或者是队列中的消息总数等于系统限制值，或队列中的字节总数等于系统限制值)，则指定`IPC_NOWAIT`使得`msgsnd`立即出错返回`EAGAIN`。如果没有指定`IPC_NOWAIT`，则进程阻塞直到 (a) 有空间可以容纳要发送的消息，或 (b) 从系统中删除了此队列，或 (c) 捕捉

到一个信号，并从信号处理程序返回。在第二种情况下，返回 EIDRM (“标志符被删除”)。最后一种情况则返回 EINTR。

注意，对消息队列删除的处理不是很完善。因为对每个消息队列并没有设置一个引用计数器（对打开文件则有这种计数器），所以删除一个队列使得仍在这一队列的进程在下次对队列进行操作时出错返回。信号量机构也以同样方式处理其删除。删除一个文件则要等到使用该文件的最后一个进程关闭了它，才能删除文件的内容。

msgrcv从队列中取用消息。

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgrcv(int msqid, void *ptr, size_t nbytes, long type, int flag);
```

返回：若成功则为消息数据部分的长度，若出错则为 -1

如同msgsnd中一样，*ptr*参数指向一个长整型数（返回的消息类型存放在其中），跟随其后的是存放实际消息数据的缓存。*nbytes*说明数据缓存的长度。若返回的消息大于 *nbytes*，而且在*flag*中设置了MSG_NOERROR，则该消息被截短（在这种情况下，不通知我们消息截短了）。如果没有设置这一标志，而消息又太长，则出错返回 E2BIG（消息仍留在队列中）。

参数*type*使我们可以指定想要哪一种消息：

- *type* == 0 返回队列中的第一个消息。
- *type* > 0 返回队列中消息类型为*type*的第一个消息。
- *type* < 0 返回队列中消息类型值小于或等于*type*绝对值，而且在这种消息中，其类型值又最小的消息。

非0*type*用于以非先进先出次序读消息。例如，若应用程序对消息赋优先权，那么 *type*就可以是优先权值。如果一个消息队列由多个客户机和一个服务器使用，那么 *type*字段可以用来包含客户机进程ID。

可以指定*flag*值为IPC_NOWAIT，使操作不阻塞。这使得如果没有所指定类型的消息，则msgrcv出错返回ENOMSG。如果没有指定IPC_NOWAIT，则进程阻塞直至（a）有了指定类型的消息，或（b）从系统中删除了此队列（出错返回EIDRM），或（c）捕捉到一个信号并从信号处理程序返回（出错返回EINTR）。

实例——消息队列与流管道的时间比较

如若需要客户机和服务器之间的双向数据流，可以使用消息队列或流管道（15.2节将介绍流管道，它与管道类似，但是是全双工的）。

表14-5显示了在两个不同系统上这两种技术在时间方面的比较。测试程序先创建IPC通道，调用fork，然后从父进程向子进程发送20M字节数据。数据发送的方式是：对于消息队列，调用10 000次msgsnd，每个消息长度为2000字节；对于流管道，调用10 000次write，每次写2000字节。时间都以秒为单位。

在SPARC上，流管道是用UNIX域套接口实现的。在SVR4之下，pipe函数提供流管道（使用12.4节所述的流机制）。

从这些数字中可见，消息队列原来的实施目的是提供比一般IPC更高速度的进程通信方法，

但现在与其他形式的IPC相比，在速度方面已经没有什么差别了。（在原来实施消息队列时，唯一的其他形式的IPC是半双工管道。）考虑到使用消息队列具有的问题（见14.6.4节），我们得出的结论是，在新的应用程序中不应当再使用它们。

表14-5 消息队列和流管道的时间比较

操 作	SPARC,SunOS 4.1.1			80386,SVR4		
	用户	系统	时钟	用户	系统	时钟
消息队列	0.8	10.7	11.6	0.7	19.6	20.1
流管道	0.3	10.6	11.0	0.5	21.4	21.9

14.8 信号量

信号量与已经介绍过的IPC机构（管道、FIFO以及消息队列）不同。它是一个计数器，用于多进程对共享数据对象的存取。为了获得共享资源，进程需要执行下列操作：

(1) 测试控制该资源的信号量。

(2) 若此信号量的值为正，则进程可以使用该资源。进程将信号量值减1，表示它使用了一个资源单位。

(3) 若此信号量的值为0，则进程进入睡眠状态，直至信号量值大于0。若进程被唤醒后，它返回至(第1)步。

当进程不再使用由一个信息量控制的共享资源时，该信号量值增1。如果有进程正在睡眠等待此信号量，则唤醒它们。

为了正确地实现信息量，信号量值的测试及减1操作应当是原子操作。为此，信号量通常是在内核中实现的。

常用的信号量形式被称之为双态信号量(binary semaphore)。它控制单个资源，其初始值为1。但是，一般而言，信号量的初值可以是任一正值，该值说明有多少个共享资源单位可供共享应用。

不幸的是，系统V的信号量与此相比要复杂得多。三种特性造成了这种并非必要的复杂性：

(1) 信号量并非是一个非负值，而必需将信号量定义为含有一个或多个信号量值的集合。当创建一个信号量时，要指定该集合中的各个值

(2) 创建信息量（semget）与对其赋初值（semctl）分开。这是一个致命的弱点，因为不能原子地创建一个信号量集合，并且对该集合中的所有值赋初值。

(3) 即使没有进程正在使用各种形式的系统V IPC，它们仍然是存在的，所以不得不为这种程序担心，它在终止时并没有释放已经分配给它的信号量。下面将要说明的undo功能就是假定要处理这种情况的。

内核为每个信号量设置了一个semid_ds结构。

```
struct semid_ds {
    struct ipc_perm  sem_perm; /* see Section 14.6.2 */
    struct sem      *sem_base; /* ptr to first semaphore in set */
    ushort         sem_nsems; /* #of semaphores in set */
    time_t         sem_otime; /* last-semop() time */
    time_t         sem_ctime; /* last-change time */
};
```

对用户而言，sem_base指针是没有价值的，它指向内核中的sem结构数组，该数组中包含了sem_nsems个元素，每个元素各对应于集合中的一个信号量值。

```

struct sem {
    ushort semval;      /* semaphore value always >= 0 */
    pid_t  sempid;     /* pid for last operation */
    ushort semncnt;    /* # processes awaiting semval > currval */
    ushort semzcnt;    /* # processes awaiting semval = 0 */
};

```

表14-6列出了影响信号量集合的系统限制（见14.6.3节）。

表14-6 影响信号量的系统限制

名 字	说 明	典型值
SEMMVMX	任一信号量的最大值	32 767
SEMAEM	任一信号量的最大终止时调整值	16 384
SEMMNI	系统中信号量集的最大数	10
SEMMNS	系统中信号量集的最大数	60
SEMMSL	每个信号量集中的最大信号量数	25
SEMMNU	系统中undo结构的最大数	30
SEMUME	每个undo结构中的最大undo项数	10
SEMOPM	每个semop调用所包含的最大操作数	10

要调用的第一个函数是semget以获得一个信号量ID。

```

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget(key_t key, int nsems, int flag);

```

返回：若成功则返回信号量ID，若出错则为-1

14.6.1节说明了将key变换为标识符的规则，讨论了是否创建一个新集合，或是引用一个现存的集合。但创建一个新集合时，对semid_ds结构的下列成员赋初值：

- 按14.6.2节中所述，对ipc_perm结构赋初值。该结构中的mode被设置为flag中的相应许可权位。这些许可权是用表14-2中的常数设置的。

- sem_otime设置为0。
- sem_ctime设置为当前时间。
- sem_nsems设置为nsems。

nsems是该集合中的信号量数。如果是创建新集合（一般在服务器中），则必须指定nsems。如果引用一个现存的集合（一个客户机），则将nsems指定为0。

semctl函数包含了多种信号量操作。

```

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semctl(int semid, int semnum, int cmd, union semun arg);

```

返回：（见下）

注意，最后一个参数是个联合（union），而非指向一个联合的指针。

```
union semun {
    int          val;          /* for SETVAL */
    struct semid_ds *buf;     /* for IPC_STAT and IPC_SET */
    ushort      *array;      /* for GETALL and SETALL */
};
```

*cmd*参数指定下列十种命令中的一种，使其在 *semid*指定的信号量集合上执行此命令。其中有五条命令是针对一个特定的信号量值的，它们用 *semnum*指定该集合中的一个成员。*semnum* 值在0和 *nsems - 1*之间（包括0和 *nsems - 1*）。

- **IPC_STAT** 对此集合取 *semid_ds*结构，并存放在由 *arg.buf*指向的结构中。
- **IPC_SET** 按由 *arg.buf*指向的结构中的值设置与此集合相关结构中的下列三个字段值：*sem_perm.uid*, *sem_perm.gid*和 *sem_perm.mode*。此命令只能由下列两种进程执行：一种是其有效用户ID等于 *sem_perm.cuid*或 *sem_perm.uid*的进程；另一种是具有超级用户特权的进程。
- **IPC_RMID** 从系统中删除该信号量集合。这种删除是立即的。仍在用此信号量的其他进程在它们下次意图对此信号量进行操作时，将出错返回 *EIDRM*。此命令只能由下列两种进程执行：一种是具有有效用户ID等于 *sem_perm.cuid*或 *sem_perm.uid*的进程；另一种是具有超级用户特权的进程。

- **GETVAL** 返回成员 *semnum*的 *semval*值。
- **SETVAL** 设置成员 *semnum*的 *semval*值。该值由 *arg.val*指定。
- **GETPID** 返回成员 *semnum*的 *sempid*值。
- **GETNCNT** 返回成员 *semnum*的 *semncnt*值。
- **GETZCNT** 返回成员 *semnum*的 *semzcnt*值。
- **GETALL** 取该集合中所有信号量的值，并将它们存放在由 *arg.array*指向的数组中。
- **SETALL** 按 *arg.array*指向的数组中的值设置该集合中所有信号量的值。

对于除 **GETALL**以外的所有 **GET**命令，*semctl*函数都返回相应值。其他命令的返回值为0。函数 *semop*自动执行信号量集合上的操作数组。

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semop(int semid, struct sembuf semoparray[], size_t nops);
```

返回：若成功则为0，若出错则为-1

*semoparray*是一个指针，它指向一个信号量操作数组。

```
struct sembuf {
    ushort sem_num; /* member # in set (0,..1, nsems-1 */
    short  sem_op; /* operation(negative, 0, or pasitive */)
    short  sem_flg; /* IPC_NOWAIT, SEM_UNDO */
};
```

*nops*规定该数组中操作的数量（元素数）。

对集合中每个成员的操作由相应的 *sem_op*规定。此值可以是负值、0或正值。（下面的讨论将提到信号量的 *undo*标志。此标志对应于相应 *sem_flg*成员的 *SEM_UNDO*位。）

(1) 最易于处理的情况是 *sem_op*为正。这对应于返回进程占用的资源。*sem_op*值加到信号

量的值上。如果指定了undo标志,则也从该进程的此信号量调整值中减去 sem_op。

(2) 若sem_op为负,则表示要获取由该信号量控制的资源。

如若该信号量的值大于或等于 sem_op的绝对值(具有所需的资源),则从信号量值中减去 sem_op的绝对值。这保证信号量的结果值大于或等于0。如果指定了undo标志,则 sem_op的绝对值也加到该进程的此信号量调整值上。

如果信号量值小于 sem_op的绝对值(资源不能满足要求),则:

- (a) 若指定了IPC_NOWAIT,则出错返回EAGAIN;
- (b) 若未指定IPC_NOWAIT,则该信号量的semncnt值加1(因为将进入睡眠状态),然后调用进程被挂起直至下列事件之一发生:
 - i. 此信号量变成大于或等于 sem_op的绝对值(即某个进程已释放了某些资源)。此信号量的semncnt值减1(因为已结束等待),并且从信号量值中减去 sem_op的绝对值。如果指定了undo标志,则 sem_op的绝对值也加到该进程的此信号量调整值上。
 - ii. 从系统中删除了此信号量。在此情况下,函数出错返回ERMID。
 - iii. 进程捕捉到一个信号,并从信号处理程序返回,在此情况下,此信号量的 semncnt值减1(因为不再等待),并且函数出错返回EINTR。

(3) 若sem_op为0,这表示希望等待到该信号量值变成0。

如果信号量值当前是0,则此函数立即返回。

如果信号量值非0,则:

- (a) 若指定了IPC_NOWAIT,则出错返回EAGAIN;
- (b) 若未指定IPC_NOWAIT,则该信号量的semncnt值加1(因为将进入睡眠状态),然后调用进程被挂起,直至下列事件之一发生:
 - i. 此信号量值变成0。此信号量的semzcnt值减1(因为已结束等待)。
 - ii. 从系统中删除了此信号量。在此情况下,函数出错返回ERMID。
 - iii. 进程捕捉到一个信号,并从信号处理程序返回。在此情况下,此信号量的 semzcnt值减1(因为不再等待),并且函数出错返回EINTR。

semop具有原子性,因为它或者执行数组中的所有操作,或者一个也不做。

exit时的信号量调整

正如前面提到的,如果在进程终止时,它占用了经由信号量分配的资源,那么就会成为一个问题。无论何时只要为信号量操作指定了SEM_UNDO标志,然后分配资源(sem_op值小于0),那么内核就会记住对于该特定信号量,分配给我们多少资源(sem_op的绝对值)。当该进程终止时,不论自愿或者不自愿,内核都将检验该进程是否还有尚未处理的信号量调整值,如果有,则按调整值对相应量值进行调整。

如果用带SETVAL或SETALL命令的semctl设置一信号量的值,则在所有进程中,对于该信号量的调整值都设置为0。

实例——信号量与记录锁的时间比较

如果多个进程共享一个资源,则可使用信号量或记录锁。对这两种技术在时间上的差别进行比较是有益的。

若使用信号量,则先创建一个包含一个成员的信号量集合,然后对该信号量值赋初值 1。为了分配资源,以 sem_op为 - 1调用semop,为了释放资源,则以 sem_op为+1调用semop。对每个操作都指定SEM_UNDO,以处理在未释放资源情况下进程终止的情况。

若使用记录锁，则先创建一个空文件，并且用该文件的第一个字节（无需存在）作为锁字节。为了分配资源，先对该字节获得一个写锁，释放该资源时，则对该字节解锁。记录锁的性质确保了，当有一个锁的进程终止时，内核会自动释放该锁。

表14-7显示了在两个不同系统上，使用这两种不同技术进行锁操作所需的时间。在各种情况中，资源都被分配，然后释放10 000次。这同时由三个不同的进程执行。表14-7中所示的时间是三个进程的总计，单位是秒。

表14-7 信号量锁和记录锁的时间比较

操 作	SPARC,SunOS 4.1.1			80386,SVR4		
	用户	系统	时钟	用户	系统	时钟
带undo的信号量	0.9	13.9	15.0	0.5	13.1	13.7
建议性纪录锁	1.1	15.2	16.5	2.1	20.6	22.9

在SPARC上，记录锁与信号量锁相比，在系统时间方面要多耗用10%。在80386上，多耗用约50%。

虽然记录锁稍慢于信号量锁，但如果只需锁一个资源（例如共享存储段）并且不需要使用系统V信号量的所有花哨的功能，则宁可使用记录锁。理由是：(a)使用简易，(b)进程终止时，会处理任一遗留下的锁。

14.9 共享存储

共享存储允许两个或多个进程共享一给定的存储区。因为数据不需要在客户机和服务器之间复制，所以这是最快的一种IPC。使用共享存储的唯一窍门是多个进程之间对一给定存储区的同步存取。若服务器将数据放入共享存储区，则在服务器做完这一操作之前，客户机不应当去取这些数据。通常，信号量被用来实现对共享存储存取的同步。（不过正如前节最后部分所述，记录锁也可用于这种场合。）

内核为每个共享存储段设置了一个shmid_ds结构。

```
struct shmid_ds {
    struct ipc_perm shm_perm; /* see Section 14.6.2 */
    struct anon_map *shm_amp; /* pointer in kernel */
    int shm_segsz; /* size of segment in bytes */
    ushort shm_lkcnt; /* number of times segment is being locked */
    pid_t shm_lpid; /* pid of last shmop() */
    pid_t shm_cpid; /* pid of creator */
    ulong shm_nattch; /* number of current attaches */
    ulong shm_cnattch; /* used only for shminfo */
    time_t shm_atime; /* last-attach time */
    time_t shm_dtime; /* last-detach time */
    time_t shm_ctime; /* last-change time */
};
```

表14-8列出了影响共享存储的系统限制（见14.6.3节）。

调用的第一个函数通常是shmget，它获得一个共享存储标识符。

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

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

int shmget(key_t key, int size, int flag);
```

返回：若成功则为共享内存 ID，若出错则为 -1

表14-8 影响共享存储的系统限制

名字	说明	典型值
SHMMAX	共享存储段的最大字节数	131 072
SHMMIN	共享存储段的最小字节数	1
SHMMNI	系统中共享存储段的最大段数	100
SHMSEG	每个进程，共享存储段的最大段数	6

14.6.1节说明了将`key`变换成一个标识符的规则，以及是创建一个新共享存储段或是存访一个现存的共享存储段。当创建一个新段时，初始化`shmid_ds`结构的下列成员：

- `ipc_perm`结构按14.6.2节中所述进行初始化。该结构中的`mode`按`flag`中的相应许可权位设置。这些许可权用表14-2中的常数指定。

- `shm_lpid`、`shm_nattach`、`shm_atime`、以及`shm_dtime`都设置为0。

- `shm_ctime`设置为当前时间。

`size`是该共享存储段的最小值。如果正在创建一个新段（一般在服务器中），则必须指定其`size`。如果正在存访一个现存的段（一个客户机），则将`size`指定为0。

`shmctl`函数对共享存储段执行多种操作。

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmctl(int shmid, int cmd, struct shmid_ds buf);
```

返回：若成功则为0，若出错则为 -1

`cmd`参数指定下列5种命令中一种，使其在`shmid`指定的段上执行。

- `IPC_STAT` 对此段取`shmid_ds`结构，并存放在由`buf`指向的结构中。

- `IPC_SET` 按`buf`指向的结构中的值设置与此段相关结构中的下列三个字段：

`shm_perm.uid`、`shm_perm.gid`以及`shm_perm.mode`。此命令只能由下列两种进程执行：一种是其有效用户ID等于`shm_perm.cuid`或`shm_perm.uid`的进程；另一种是具有超级用户特权的进程。

- `IPC_RMID` 从系统中删除该共享存储段。因为每个共享存储段有一个连接计数（`shm_nattach`在`shmid_ds`结构中），所以除非使用该段的最后一个进程终止或与该段脱接，否则不会实际上删除该存储段。不管此段是否仍在使用，该段标识符立即被删除，所以不能再用`shmat`与该段连接。此命令只能由下列两种进程执行：一种是其有效用户ID等于`shm_perm.cuid`或`shm_perm.uid`的进程；另一种是具有超级用户特权的进程。

- `SHM_LOCK` 锁住共享存储段。此命令只能由超级用户执行。

- `SHM_UNLOCK` 解锁共享存储段。此命令只能由超级用户执行。

一旦创建了一个共享存储段，进程就可调用`shmat`将其连接到它的地址空间中。

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

void *shmat(int shmid, void *addr, int flag);
```

返回：若成功则为指向共享存储段的指针，若出错则为 -1

共享存储段连接到调用进程的哪个地址上与 *addr* 参数以及在 *flag* 中是否指定 SHM_RND 位有关。

(1) 如果 *addr* 为 0，则此段连接到由内核选择的第一个可用地址上。

(2) 如果 *addr* 非 0，并且没有指定 SHM_RND，则此段连接到 *addr* 所指定的地址上。

(3) 如果 *addr* 非 0，并且指定了 SHM_RND，则此段连接到 ($addr - (addr \bmod SHMLBA)$) 所表示的地址上。SHM_RND 命令的意思是：取整。SHMLBA 的意思是：低边界地址倍数，它总是 2 的乘方。该算式是将地址向下取最近 1 个 SHMLBA 的倍数。

除非只计划在一种硬件上运行应用程序（这在当今是不大可能的），否则不用指定共享段所连接到的地址。所以一般应指定 *addr* 为 0，以便由内核选择地址。

如果在 *flag* 中指定了 SHM_RDONLY 位，则以只读方式连接此段。否则以读写方式连接此段。

shmat 的返回值是该段所连接的实际地址，如果出错则返回 -1。

当对共享存储段的操作已经结束时，则调用 shmdt 脱接该段。注意，这并不从系统中删除其标识符以及其数据结构。该标识符仍然存在，直至某个进程（一般是服务器）调用 shmctl（带命令 IPC_RMID）特地删除它。

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmdt(void *addr);
```

返回：若成功则为 0，若出错则为 -1

addr 参数是以前调用 shmat 时的返回值。

实例

内核将以地址 0 连接的共享存储段放在什么位置上与系统密切相关。程序 14-11 打印一些信息，它们与指定系统将不同类型的数据放在什么位置有关。在一个特定的系统上运行此程序，其输出如下：

```
$ a.out
array[] from 18f48 to 22b88
stack around f7fffb2c
malloced from 24c28 to 3d2c8
shared memory attached from f77d0000 to f77e86a0
```

图 14-13 显示了这种情况，这与图 7-3 中所示的典型存储区布局类似。注意，共享存储段紧靠在栈之下。实际上，在共享存储段和栈之间有大约 8M 字节的未用地址空间。

程序14-11 打印不同类型的数据所存放的位置

```

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include "ourhdr.h"

#define ARRAY_SIZE 40000
#define MALLOC_SIZE 100000
#define SHM_SIZE 100000
#define SHM_MODE (SHM_R | SHM_W) /* user read/write */

char array[ARRAY_SIZE]; /* uninitialized data = bss */

int
main(void)
{
    int shmid;
    char *ptr, *shmptr;

    printf("array[] from %x to %x\n", &array[0], &array[ARRAY_SIZE]);
    printf("stack around %x\n", &shmid);

    if ( (ptr = malloc(MALLOC_SIZE)) == NULL)
        err_sys("malloc error");
    printf("malloced from %x to %x\n", ptr, ptr+MALLOC_SIZE);

    if ( (shmid = shmget(IPC_PRIVATE, SHM_SIZE, SHM_MODE)) < 0)
        err_sys("shmget error");
    if ( (shmptr = shmat(shmid, 0, 0)) == (void *) -1)
        err_sys("shmat error");
    printf("shared memory attached from %x to %x\n",
           shmptr, shmptr+SHM_SIZE);
    if (shmctl(shmid, IPC_RMID, 0) < 0)
        err_sys("shmctl error");

    exit(0);
}

```

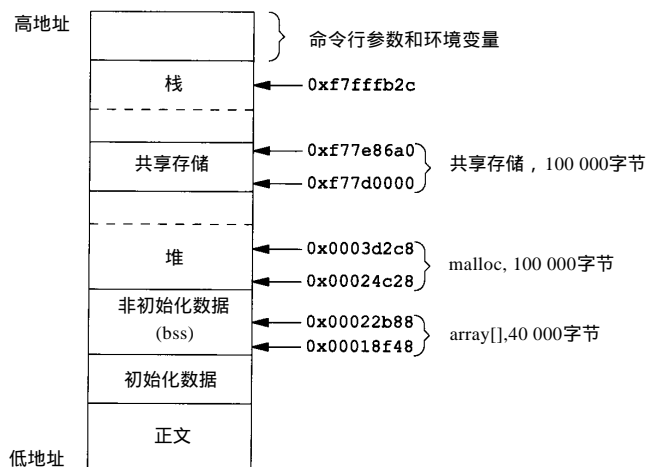


图14-13 特定系统上的存储区布局

实例——/dev/zero的存储映射

共享存储可由不相关的进程使用。但是，如果进程是相关的，则 SVR4提供了一种不同的

技术。

设备/dev/zero在读时，是0字节的无限资源。此设备也接收写向它的任何数据，但忽略此数据。我们对此设备作为IPC的兴趣在于，当对其进行存储映射时，它具有一些特殊性质：

- 创建一个未名存储区，其长度是mmap的第二个参数，将其取整为系统上的最近页长。
- 存储区都初始化为0。
- 如果多个进程的共同祖先进程对mmap指定了MAP_SHARED标志，则这些进程可共享此存储区。

程序14-12是使用此特殊设备的一个例子。它打开此/dev/zero设备，然后指定一个长整型调用mmap。注意，一旦该存储区被映射了，就能关闭此设备。然后，进程创建一个子进程。因为在调用mmap时指定了MAP_SHARED，所以一个进程写到存储映照区的数据可由另一进程见到。（如果已指定MAP_PRIVATE，则此程序不会工作。）

然后，父、子进程交替运行，使用8.8节中的同步函数各自对共享存储映射区中的一个长整型数加1。存储映射区由mmap初始化为0。父进程先对它进行增1操作，使其成为1，然后子进程对其进行增1操作，使其成为2，然后父进程使其成为3……注意，当在update函数中，对长整型值增1时，必须使用括号，因为增加的是其值，而不是指针。

程序14-12 在父、子进程间使用/dev/zero存储映射I/O的IPC

```
#include <sys/types.h>
#include <sys/mman.h>
#include <fcntl.h>
#include "ourhdr.h"

#define NLOOPS 1000
#define SIZE sizeof(long) /* size of shared memory area */

static int update(long *);

int
main()
{
    int fd, i, counter;
    pid_t pid;
    caddr_t area;

    if ( (fd = open("/dev/zero", O_RDWR)) < 0)
        err_sys("open error");
    if ( (area = mmap(0, SIZE, PROT_READ | PROT_WRITE,
                    MAP_SHARED, fd, 0)) == (caddr_t) -1)
        err_sys("mmap error");
    close(fd); /* can close /dev/zero now that it's mapped */
    TELL_WAIT();
    if ( (pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid > 0) { /* parent */
        for (i = 0; i < NLOOPS; i += 2) {
            if ( (counter = update((long *) area)) != i)
                err_quit("parent: expected %d, got %d", i, counter);
            TELL_CHILD(pid);
            WAIT_CHILD();
        }
    } else { /* child */
        for (i = 1; i < NLOOPS + 1; i += 2) {
            WAIT_PARENT();
            if ( (counter = update((long *) area)) != i)
```

```

        err_quit("child: expected %d, got %d", i, counter);
        TELL_PARENT(getppid());
    }
}
exit(0);
}
static int
update(long *ptr)
{
    return( (*ptr)++ ); /* return value before increment */
}

```

以上述方式使用/dev/zero的优点是：在调用mmap创建映射区之前，无需存在一个实际文件。映射/dev/zero自动创建一个指定长度的映射区。这种技术的缺点是：它只能由相关进程使用。如果在无关进程之间需要使用共享存储区，则必须使用shmXXX函数。

实例——匿名存储映射

4.3+BSD提供了一种类似于/dev/zero的设施，称为匿名存储映射。为了使用这种功能，在调用mmap时指定MAP_ANON标志，并将描述符指定为-1。结果得到的区域是匿名的（因为它并不通过一个文件描述符与一个路径名相结合），并且创建一个存储区，它可与后代进程共享。

为了使程序14-12应用4.3+BSD的这种特征，做了两个修改：(a) 删除/dev/zero的open条语句，(b) 将mmap调用修改成下列形式：

```

if ( (area = mmap(0, SIZE, PROT_READ | PROT_WRITE,
                 MAP_ANON | MAP_SHARED, -1, 0)) == (caddr_t) -1)

```

在此调用中，指定了MAP_ANON标志，并将文件描述符设置为-1。程序14-12的其余部分则不加改变。

14.10 客户机-服务器属性

下面详细说明客户机和服务器的属性，这些属性受到它们之间所使用的IPC的不同类型的影响。

最简单的关系类型是使客户机fork并执行所希望的服务器。在fork之前先创建两个单向管道以使数据可在两个方向传输。图14-8是这种形式的一个例子。被执行的服务器可能是设置-用户-ID的程序，这使它具有了特权。查看客户机的实际用户ID就可以决定客户机的身份。（回忆8.9节，从中可了解到在exec前后实际用户ID和实际组ID并没有改变。）

在这种安排下，可以构筑一个“开放式服务器”（15.4节即提供了这种客户机和服务器的一种实现）。它为客户机开放文件而不是客户机调用open函数。这样就可以增加在正常的UNIX用户/组/其他许可权之上或之外的附加的许可权检查。假定服务器是设置-用户-ID程序，这给予了它附加的许可权（很可能是root许可权）。服务器用客户机的实际用户ID以决定是否给予它对所要求的文件的存取。使用这种方式，可以构筑一个服务器，它允许某种用户通常没有的存取权。

在此例子中，因为服务器是父进程的子进程，所以它能做的一切是将文件内容传送给父进程。这种方式对一般文件工作得很好，同时，也可被用于专用设备文件。我们希望能做的是使服务器打开所要的文件，并将文件描述符送回。父进程可向子进程传送打开文件描述符，而子进程则不能向父进程传回一个描述符（除非使用将在下一章介绍的专门编程技术）。

下一种服务器类型已显示于图14-12中，服务器是一个精灵进程，客户机则用某种形式的

IPC与其联系。可以将管道用于这种形式的客户机-服务器关系。要求有一种命名的IPC，例如FIFO或消息队列。对于FIFO，如果服务器必需将数据送回客户机，则对每个客户机都要有单独使用的FIFO。如果客户机-服务器应用程序只有客户机向服务器送数据，则只需要一个众所周知的FIFO。（系统V行式打印机假脱机程序使用这种形式的客户机-服务器。客户机是lp(1)命令，服务器是lpsched进程。因为只有从客户机到服务器的数据流，没有任何数据需送回客户机，所有只需使用一个FIFO。

使用消息队列则存在多种可能性：

(1) 在服务器和客户机之间可以只使用一个队列，使用每个消息的类型字段指明谁是消息的接受者。例如，客户机可以用类型字段为1发送它们的消息。在要求之中应包括客户机的进程ID。此后，服务器在发送响应消息时，将类型字段设置为客户机的进程ID。服务器只接受类型字段为1的消息（msgrcv的第四个参数），客户机则只接受类型字段等于它们的进程ID的消息。

(2) 另一种方法是每个客户机使用一个单独的消息队列。在向服务器发送第一个请求之前，每个客户机先创建它自己的消息队列，创建时使用关键字IPC_PRIVATE。服务器也有它自己的队列，其关键字或标识符是所有客户机知道的。客户机将其第一个请求送到服务器的众所周知的队列上，该请求中应包含其客户机消息队列的队列ID。服务器将其第一个响应送至客户机队列，此后的所有请求和响应都在此队列上交换。

使用这种技术的一个问题是：每个客户机专用队列通常只有一个消息在其中——或者是对服务器的一个请求，或者是对客户机的响应。这似乎是对有限的系统资源（消息队列）的浪费，可以用一个FIFO来代替。另一个问题是服务器需从多个队列读消息。对于消息队列，select和poll都不起作用

使用消息队列的这两种技术都可以用共享存储段和同步方法（信号量或记录锁）实现。使用共享存储段的问题是，一次只能有一个消息在共享存储段中——类似于队列限制为只能有一个消息。为此，在使用共享存储IPC时，通常每个客户机使用一个共享存储段。

这种类型的客户机-服务器关系（客户机和服务器是无关系进程）的问题是：服务器如何准确地标识客户机。除非服务器正在执行一种非特权操作，否则服务器知道谁是客户机是很重要的。例如，若服务器是一个设置-用户-ID程序，就有这种要求。虽然，所有这几种形式的IPC都经由内核，但是它们并未提供任何措施使内核能够标识发送者。

对于消息队列，如果在客户机和服务器之间使用一个专用队列（于是一次只有一个消息在该队列上），那么队列的msg_lspid包含了对方进程的进程ID。但是当客户机将请求发送给服务器时，我们想要的是客户机的有效用户ID，而不是它的进程ID。现在还没有一种可移植的方法，在已知进程ID情况下用其可以得到有效用户ID。（确实，内核在进程表项中保持有这两种值，但是除非彻底检查内核存储空间，否则已知一个，无法得到另一个。）

我们将在15.5.2中使用下列技术，使服务器可以标识客户机。同样的技术也可用于FIFO、消息队列、信号量或共享存储。下面的说明具体针对按图14-12中的方式使用FIFO情况。客户机必须创建它自己的FIFO，并且设置FIFO的文件存取许可权，使得只允许用户-读，用户-写。假定服务器具有超级用户特权（或者它很可能并不关心客户机的真实标识），所以服务器仍可读、写此FIFO。当服务器在众所周知的FIFO上接受到客户机的第一个请求时（它应当包含客户机专用FIFO的标识），服务器调用针对客户机专用FIFO的stat或fstat。服务器所采用的假设是：客户机的有效用户ID是FIFO的所有者（stat结构的st_uid字段）。服务器验证该FIFO只有用户-读、用户-写许可权。服务器还应检查是该FIFO的三个时间量（stat结构中的st_atime, st_mtime和st_ctime字段），要检查它们与当前时间是否很接近（例如不早于当前时间15秒或30秒）。如果

一个有预谋的客户机可以创建一个 FIFO，使另一个用户成为其所有者，并且设置该文件的许可权为用户-读和用户-写，那么在系统中就存在了其他基础性的安全问题。

为了在系统 V IPC 中应用这种技术，回想一下与每个消息队列、信号量、以及共享存储段相关的 `ipc_perm` 结构，其中 `cuid` 和 `cgid` 字段标识 IPC 结构的创建者。以 FIFO 为例，服务器应当要求客户机创建该 IPC 结构，并使客户机将存取权设置为只允许用户-读和用户-写。服务器也应检验与该 IPC 相关的时间量与当前时间是否很接近（因为这些 IPC 结构在显式地删除之前一直存在）。

在 15.5.1 节中，将会看到进行这种身份验证工作的一种更好方法是内核提供客户机的有效用户 ID 和有效组 ID。SVR4 在进程之间传送文件描述符时可以做到这一点。

14.11 小结

本章详细说明了进程间通信的多种形式；管道、命名管道（FIFO）以及另外三种 IPC 形式，通常称之为系统 V IPC——消息队列、信号量和共享存储。信号量实际上是同步原语而不是 IPC，常用于共享资源的同步存取，例如共享存储段。对于管道，说明了 `popen` 的实现，说明了协同进程，以及使用标准 I/O 库缓存机制时可能遇到的问题。

在时间方面，对消息队列与流管道、信号量与记录锁做了比较后，提出了下列建议：学会使用管道和 FIFO，因为在大量应用程序中仍可有效地使用这两种基本技术。在新的应用程序中，要尽可能避免使用消息队列以及信号量，而应当考虑流管道和记录锁，因为它们与 UNIX 内核的其他部分集成得要好多。共享存储段有其应用场合，而 `mmap` 函数（见 12-9 节）则可能在以后的版本中起更大作用。

下一章将介绍一些更先进的 IPC 形式，它们由更加新的系统，例如 SVR4 和 4.3+BSD 提供。

习题

- 14.1 在程序 14-2 中父进程代码的末尾，如删除 `waitpid` 前的 `close`，结果将如何？
- 14.2 在程序 14-2 中父进程代码的末尾，如删除 `waitpid`，结果将如何？
- 14.3 如果 `popen` 的参数是一个不存在的命令会有什么结果？编写一段程序测试一下。
- 14.4 删除程序 14-9 中的信号量处理程序，执行程序并终止子进程。输入一行后，怎样能说明父进程是由 SIGPIPE 终止的？

14.5 将程序 14-9 中进行管道读、写的 `read` 和 `write` 用标准 I/O 库代替。

14.6 POSIX.1 加入 `waitpid` 函数的理由之一是 POSIX.1 前的大多数系统不能处理下面的代码。

```
if ( (fp = popen("/bin/true", "r")) == NULL )
    ...
if ( (rc = system("sleep 100")) == -1 )
    ...
if (pclose (fp) == -1)
    ...
```

若在这段代码中不使用 `waitpid` 函数会如何？用 `wait` 代替呢？

14.7 当一个管道被写者关闭后，解释 `select` 和 `poll` 如何处理该管道的输入描述符？当一个管道的读端被关闭时，请重做此习题以查看该管道的输出描述符。编两个测试程序，一个用 `select` 一个用 `poll`，并确定答案是否正确。

14.8 如果 `popen` 以 `type` 为 `r` 执行 `cmdstring` 并将结果写到标准出错输出，结果如何？

14.9 `popen` 会使得 shell 执行它的 `cmdstring` 参数，当 `cmdstring` 终止时会产生什么结果？（提

示：画出包含的所有进程。)

14.10 大多数UNIX系统允许读写FIFO，但是POSIX.1特别声明没有定义为读写而打开FIFO。请用非阻塞方法实现为读写而打开FIFO。

14.11 除非文件包含敏感或机密数据，否则允许其他用户读文件不会造成损害。但是，如果一个恶意进程读取了被一个服务器和几个客户机进程使用的消息队列中的一条消息后会有什么结果？恶意进程需要知道哪些信息就可以读消息队列？

14.12 编写一段程序完成下面的工作：循环五次创建一个消息队列，并打印队列的标识符，然后删除队列。接着再循环五次利用关键字IPC_PRIVATE创建消息队列并将一条消息放在队列中，程序终止后用`ipcs(1)`查看消息队列。解释队列标识符的变化。

14.13 描述如何在共享存储段中建立一个数据对象的连接列表。列表指针如何保存？

14.14 画出程序14-12的变量*i*在父进程和子进程中随时间变化的值。由`update`函数返回该值，其类型为长整型，保存在共享存储区。假设`fork`后子进程先运行。

14.15 使用14.9节的`shmXXX`函数代替共享存储映射区重写程序14-12。

14.16 使用14.8节系统V提供的信号量函数重写程序14-12实现父进程与子进程间的交替。

14.17 使用建议性记录锁定方法重写程序14-12实现父进程与子进程间的交替。

14.18 解释`mmap`函数的文件描述符参数如何在4.3+BSD系统匿名存储映射方式下允许不相关进程间的存储共享。

第15章 高级进程间通信

15.1 引言

上一章说明了各种UNIX系统提供的IPC经典方法，包括：管道、FIFO、消息队列、信号量和共享存储。本章介绍某些高级的IPC以及它们的应用方法，包括：流管道和命名流管道。使用这些机制，可以在进程间传送打开文件描述符。在分别为每一个客户进程提供一个通道的系统中，这些通信机制使客户进程能与精灵服务进程会合。4.2BSD和SVR3.2最早提供这些高级形式的IPC，但是至今尚未广泛使用，也缺少参考文献。本章中很多思想来自 Pressotto和Ritchie [1990] 的论文。

15.2 流管道

流管道是一个双向（全双工）管道。单个流管道就能向父、子进程提供双向的数据流。图15-1显示了观察流管道的两种方式。它与图14-1的唯一区别是双向箭头连线，因为流管道是全双工的。

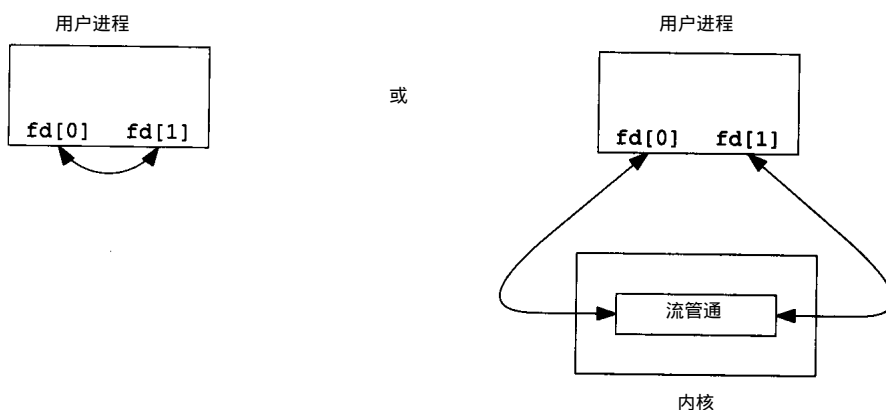


图15-1 观察流管道的两种方式

实例

下面用一个流管道再次实现了程序14-9的协作进程实例。程序15-1是新的main函数。add2协作进程与程序14-8中的相同。程序15-1调用了创建一个流管道的新函数s_pipe。（下面将说明该函数的SVR4和4.3+BSD版本。）

程序15-1 用流管道驱动add2过滤进程的程序

```
#include <signal.h>
#include "ourhdr.h"
```

```

static void sig_pipe(int);      /* our signal handler */

int
main(void)
{
    int     n, fd[2];
    pid_t   pid;
    char    line[MAXLINE];

    if (signal(SIGPIPE, sig_pipe) == SIG_ERR)
        err_sys("signal error");

    if (s_pipe(fd) < 0)        /* only need a single stream pipe */
        err_sys("pipe error");

    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid > 0) {        /* parent */
        close(fd[1]);
        while (fgets(line, MAXLINE, stdin) != NULL) {
            n = strlen(line);
            if (write(fd[0], line, n) != n)
                err_sys("write error to pipe");
            if ( (n = read(fd[0], line, MAXLINE)) < 0)
                err_sys("read error from pipe");
            if (n == 0) {
                err_msg("child closed pipe");
                break;
            }
            line[n] = 0;      /* null terminate */
            if (fputs(line, stdout) == EOF)
                err_sys("fputs error");
        }
        if (ferror(stdin))
            err_sys("fgets error on stdin");
        exit(0);
    } else {                  /* child */
        close(fd[0]);
        if (fd[1] != STDIN_FILENO) {
            if (dup2(fd[1], STDIN_FILENO) != STDIN_FILENO)
                err_sys("dup2 error to stdin");
        }
        if (fd[1] != STDOUT_FILENO) {
            if (dup2(fd[1], STDOUT_FILENO) != STDOUT_FILENO)
                err_sys("dup2 error to stdout");
        }
        if (execl("./add2", "add2", NULL) < 0)
            err_sys("execl error");
    }
}

static void
sig_pipe(int signo)
{
    printf("SIGPIPE caught\n");
    exit(1);
}

```

父程序只使用fd[0]，子程序只使用fd[1]。因为流管道的每一端都是全双工的，所以父进程读、写fd[0]，而子程序将fd[1]复制到标准输入和标准输出。图15-2显示了由此构成的描述符。

s_pipe函数定义为与标准pipe函数类似。它的调用参数与pipe相同，但返回的描述符以读-写方式打开。

实例——SVR4下的s_pipe函数

程序15-2是s_pipe函数的SVR4版本。它只是调用创建全双工管道的标准pipe函数。

程序15-2 s_pipe函数的SVR4版本

```
#include "ourhdr.h"
int
s_pipe(int fd[2]) /* two file descriptors returned in fd[0] & fd[1] */
{
    return( pipe( fd ) );
}
```

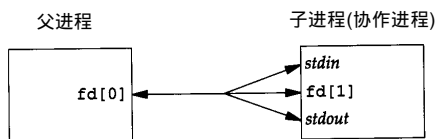


图15-2 为协作进程安排描述符

在系统V的早期版本中也可以创建流管道，但要进行的处理较多。有关在SVR3.2下创建流管道的详细情况，请参阅Stevens [1990]。

图15-3显示了SVR4之下管道的基本结构。它主要是两个相互连接的流首。

因为管道是一种流设备，故可将处理模块压入管道的任一端。15.5.1节将用此技术提供一个可以装配的命名管道。

实例——4.3+BSD之下的s_pipe函数

程序15-3是s_pipe函数的BSD版本。此函数在4.2BSD及以后的各版本中起作用。它创建一对互连的UNIX域流套接口。

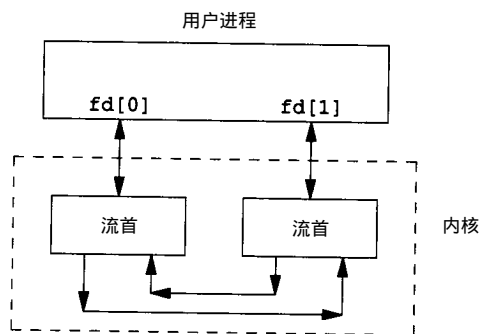


图15-3 SVR4之下的管道

自4.2BSD开始，常规的管道已用此方式实现。但是，当调用pipe时，第一个描述符的写端和第二个描述符的读端都被关闭。为获得全双工管道，必须直接调用socketpair。

程序15-3 s_pipe函数的BSD版本

```
#include <sys/types.h>
#include <sys/socket.h>
#include "ourhdr.h"
int
s_pipe(int fd[2]) /* two file descriptors returned in fd[0] & fd[1] */
{
    return( socketpair(AF_UNIX, SOCK_STREAM, 0, fd) );
}
```

15.3 传送文件描述符

在进程间传送打开文件描述符的能力非常有用。用此可以对客户机 / 服务器应用进行不同的设计。它允许一个进程（一般是服务器）处理与打开一个文件有关的所有操作（涉及的细节可能是：将网络名翻译为网络地址、拨号调制解调器、协商文件锁等。）以及向调用进程返回一描述符，该描述符可被用于以后的所有 I/O 函数。打开文件或设备的所有细节对客户而言都是透明的。

4.2BSD支持传送打开描述符，但其实实施中有些错误。4.3BSD排除了这些错误。SVR3.2及以上版本都支持传送打开描述符。

下面进一步说明“从一个进程向另一个进程传送一打开文件描述符”的含义。回忆图 3-2，其中显示了两个进程，它们打开了同一文件。虽然它们共享同一 v 节点表，但每个进程都有它自己的文件表项。

当从一个进程向另一个进程传送一打开文件描述符时，我们想要发送进程和接收进程共享同一文件表项。图 15-4 显示了所希望的安排。在技术上，发送进程实际上向接受进程传送一个指向一打开文件表项的指针。该指针被分配存放在接收进程的第一个可用描述符项中。（注意，不要得到错觉以为发送进程和接收进程中的描述符编号是相同的，通常它们是不同的。）这种情况与在 fork 之后，父、子进程完全共享一个打开文件表项相同（见图 8-1）。

当发送进程将描述符传送给接收进程后，通常它关闭该描述符。发送进程关闭该描述符并不造成关闭该文件或设备，其原因是该描述符对应的文件仍需为接收进程打开（即使接收进程尚未接收到该描述符）。

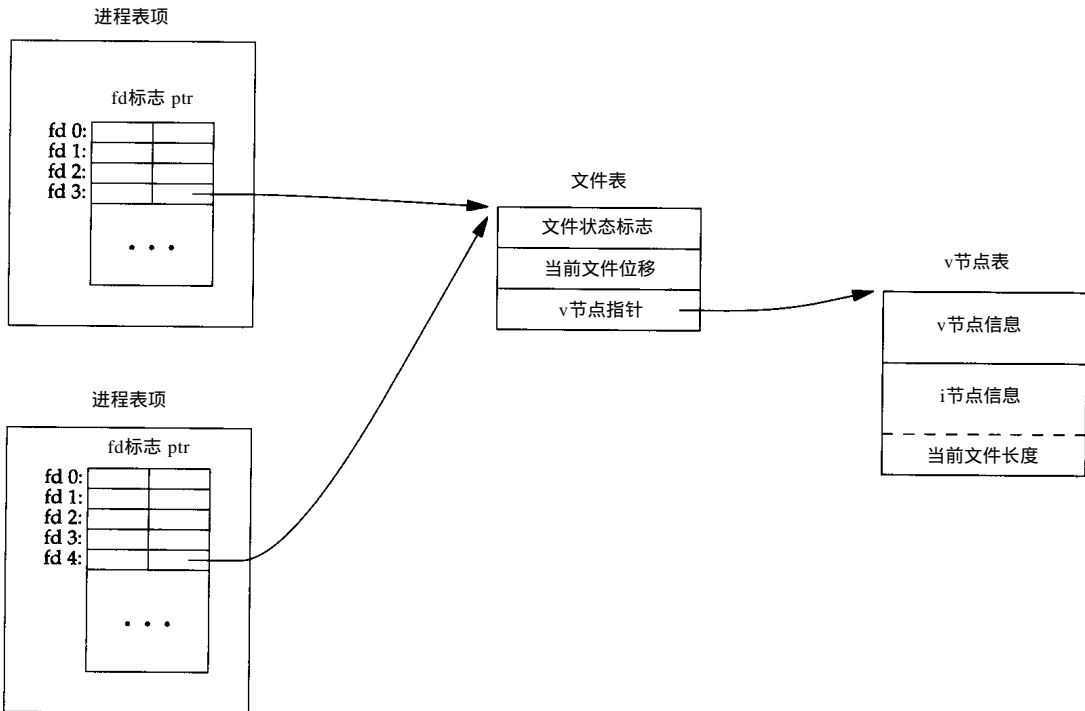


图15-4 从上一进程传送一个打开文件至下一进程

下面定义本章使用的三个函数（第18章也将使用）以发送和接收文件描述符。本节将会给出对于SVR4和4.3+BSD的这三个函数的不同实现。

```
#include "ourhdr.h"

int send_fd(int spipefd, int filedес);

int send_err(int spipefd, int status, const char * errmsg);
                两个函数返回：若成功则为0，若出错则为-1

int recv_fd(int spipefd, ssize_t (userfunc)(int, const void *, size_t));
                返回：若成功则为文件描述符，若出错则 <0
```

当一个进程（通常是服务器）希望将一个描述符传送给另一个进程时，它调用 `send_fd` 或 `send_err`。等待接收描述符的进程（客户机）调用 `recv_fd`。

`send_fd` 经由流管道 `spipefd` 发送描述符 `filedes`。`send_err` 经由流管道 `spipefd` 发送 `errmsg` 和 `status` 字节。`status` 的值应在 `-1 ~ -255` 之间

客户机调用 `recv_fd` 接收一描述符。如果一切正常（发送者调用了 `send_fd`），则作为函数值返回非负描述符。否则，返回值是由 `send_err` 发送的 `status`（`-1 ~ -255` 之间的一个值）。另外，如果服务器发送了一条出错消息，则客户机调用它自己的 `userfunc` 处理该消息。`userfunc` 的第一个参数是常数 `STDERR_FILENO`，然后是指向出错消息的指针及其长度。客户机常将 `userfunc` 指定为 UNIX 的 `write` 函数。

我们实现了用于这三个函数的我们自己制定的协议。为发送一描述符，`send_fd` 先发送两个0字节，然后是实际描述符。为了发送一条出错消息，`send_err` 发送 `errmsg`，然后是1个0字节，最后是 `status` 字节的绝对值（`1~255`）。`recv_fd` 读流管道中所有字节直至 null 字符。null 字符之前的所有字符都送给调用者的 `userfunc`。`recv_fd` 读到的下一个字节是 `status` 字节。若 `status` 字节为0，那么一个描述符已传送，否则表示没有接收到描述符。

`send_err` 函数在将出错消息写到流管道后，即调用 `send_fd` 函数。这示于程序 15-4 中。

程序15-4 send_err函数

```
#include "ourhdr.h"

/* Used when we had planned to send an fd using send_fd(),
 * but encountered an error instead. We send the error back
 * using the send_fd()/recv_fd() protocol. */

int
send_err(int clifd, int errcode, const char *msg)
{
    int    n;

    if ( (n = strlen(msg)) > 0)
        if (writen(clifd, msg, n) != n) /* send the error message */
            return(-1);

    if (errcode >= 0)
        errcode = -1; /* must be negative */

    if (send_fd(clifd, errcode) < 0)
        return(-1);

    return(0);
}
```

以下三节介绍了在SVR4、4.3BSD和4.3+BSD下，两个函数send_fd和recv_fd的实际实现。

15.3.1 SVR4

在SVR4之下，文件描述符用两个ioctl命令在一流管道中交换，这两个命令是：I_SENDFD和I_RECVFD。为了发送一描述符，将ioctl的第三个参数设置为实际描述符。这示于程序15-5中。

程序15-5 SVR4的send_fd函数

```

#include    <sys/types.h>
#include    <stropts.h>
#include    "ourhdr.h"

/* Pass a file descriptor to another process.
 * If fd<0, then -fd is sent back instead as the error status. */
int
send_fd(int clifd, int fd)
{
    char    buf[2];        /* send_fd()/recv_fd() 2-byte protocol */
    buf[0] = 0;            /* null byte flag to recv_fd() */
    if (fd < 0) {
        buf[1] = -fd;     /* nonzero status means error */
        if (buf[1] == 0)
            buf[1] = 1;   /* -256, etc. would screw up protocol */
    } else {
        buf[1] = 0;       /* zero status means OK */
    }

    if (write(clifd, buf, 2) != 2)
        return(-1);

    if (fd >= 0)
        if (ioctl(clifd, I_SENDFD, fd) < 0)
            return(-1);
    return(0);
}

```

当接收一描述符时，ioctl的第三个参数是一指向strecvfd结构的指针。

```

struct strecvfd {
    int    fd;            /* new descriptor */
    uid_t  uid;          /* effective user ID of sender */
    gid_t  gid;          /* effective group ID of sender */
    char   fill[8];
};

```

recv_fd读流管道直到接收到双字节协议的第一个字节（null字节）。当发出带I_RECVFD命令的ioctl时，在流读首处的第一条消息应当是一个描述符，它是由I_SENDFD发来的，或者得到一条出错消息。这示于程序15-6中。

程序15-6 SVR4的recv_fd函数

```

#include    <sys/types.h>
#include    <stropts.h>
#include    "ourhdr.h"

/* Receive a file descriptor from another process (a server).
 * In addition, any data received from the server is passed

```



```

* to (*userfunc)(STDERR_FILENO, buf, nbytes). We have a
* 2-byte protocol for receiving the fd from send_fd(). */

int
recv_fd(int servfd, ssize_t (*userfunc)(int, const void *, size_t))
{
    int          newfd, nread, flag, status;
    char         *ptr, buf[MAXLINE];
    struct strbuf dat;
    struct strrecvfd recvfd;

    status = -1;
    for ( ; ; ) {
        dat.buf = buf;
        dat.maxlen = MAXLINE;
        flag = 0;
        if (getmsg(servfd, NULL, &dat, &flag) < 0)
            err_sys("getmsg error");
        nread = dat.len;
        if (nread == 0) {
            err_ret("connection closed by server");
            return(-1);
        }
        /* See if this is the final data with null & status.
        Null must be next to last byte of buffer, status
        byte is last byte. Zero status means there must
        be a file descriptor to receive. */
        for (ptr = buf; ptr < &buf[nread]; ) {
            if (*ptr++ == 0) {
                if (ptr != &buf[nread-1])
                    err_dump("message format error");
                status = *ptr & 255;
                if (status == 0) {
                    if (ioctl(servfd, I_RECVFD, &recvfd) < 0)
                        return(-1);
                    newfd = recvfd.fd; /* new descriptor */
                } else
                    newfd = -status;
                nread -= 2;
            }
        }
        if (nread > 0)
            if ((*userfunc)(STDERR_FILENO, buf, nread) != nread)
                return(-1);

        if (status >= 0) /* final data has arrived */
            return(newfd); /* descriptor, or -status */
    }
}

```

15.3.2 4.3BSD

不幸的是，对于4.3BSD以及在其基础上构造的SunOS和Ulrix，以及从4.3BSD Reno开始的后续版本必须提供不同的实现。

为了交换文件描述符，调用 `sendmsg(2)`和`recvmsg(2)`函数。这两个函数的参数中都有一个指向 `msg_hdr`的指针，该结构包含了所有关于要发送和接收消息的信息。该结构定义在 `<sys/socket.h>` 头文件中，在BSD4.3之下，其样式是：

```

struct msg_hdr {
    caddr_t      msg_name; /* optional address */

```

```

int          msg_namelen; /* size of address */
struct iovec *msg_iov;   /* scatter/gather array */
int          msg_iovlen; /* # elements in msg_iov array */
caddr_t      msg_accrighs; /* access rights sent/received */
int          msg_accrighslen; /* size of access rights buffer */
};

```

头两个元素通常用于在网络连接上发送数据报文，在这里，目的地址可以由每个数据报文指定。下面两个元素使我们可以指定缓存的数组（散布读和聚集写），这如同对readv和writev函数（见12.7节）的说明一样。最后两个元素处理存取权的传送和接收。当前唯一定义的存取权是文件描述符。存取权仅可跨越一个UNIX域套接口传送（即在4.3BSD之下作为流管道所使用的）。为了发送或接收一文件描述符，将msg_accrighs设置为指向该整型描述符，将msg_accrighslen设置为描述符的长度（即整型的长度）。仅当此长度非0时，才传送或接收描述符。

程序15-7是4.3BSD的send_fd函数。

程序15-7 4.3BSD的send_fd函数

```

#include <sys/types.h>
#include <sys/socket.h> /* struct msghdr */
#include <sys/uio.h>    /* struct iovec */
#include <errno.h>
#include <stddef.h>
#include "ourhdr.h"

/* Pass a file descriptor to another process.
 * If fd<0, then -fd is sent back instead as the error status. */

int
send_fd(int clifd, int fd)
{
    struct iovec    iov[1];
    struct msghdr   msg;
    char           buf[2]; /* send_fd()/recv_fd() 2-byte protocol */

    iov[0].iov_base = buf;
    iov[0].iov_len  = 2;

    msg.msg_iov     = iov;
    msg.msg_iovlen  = 1;
    msg.msg_name    = NULL;
    msg.msg_namelen = 0;

    if (fd < 0) {
        msg.msg_accrighs = NULL;
        msg.msg_accrighslen = 0;
        buf[1] = -fd; /* nonzero status means error */
        if (buf[1] == 0)
            buf[1] = 1; /* -256, etc. would screw up protocol */
    } else {
        msg.msg_accrighs = (caddr_t) &fd; /* addr of descriptor */
        msg.msg_accrighslen = sizeof(int); /* pass 1 descriptor */
        buf[1] = 0; /* zero status means OK */
    }
    buf[0] = 0; /* null byte flag to recv_fd() */

    if (sendmsg(clifd, &msg, 0) != 2)
        return(-1);

    return(0);
}

```

在sendmsg调用中，发送双字节协议数据（null和status字节）和描述符。

为了接收一文件描述符，从流管道读，直至读到 null字节，它位于最后的 status字节之前。null字节之前是一条出错消息，它来自发送者。这示于程序 15-8。

程序15-8 4.3BSD的recv_fd函数

```
#include <sys/types.h>
#include <sys/socket.h> /* struct msghdr */
#include <sys/uio.h> /* struct iovec */
#include <stddef.h>
#include "ourhdr.h"

/* Receive a file descriptor from another process (a server).
 * In addition, any data received from the server is passed
 * to (*userfunc)(STDERR_FILENO, buf, nbytes). We have a
 * 2-byte protocol for receiving the fd from send_fd(). */

int
recv_fd(int servfd, ssize_t (*userfunc)(int, const void *, size_t))
{
    int          newfd, nread, status;
    char         *ptr, buf[MAXLINE];
    struct iovec iiov[1];
    struct msghdr msg;

    status = -1;
    for ( ; ; ) {
        iiov[0].iov_base = buf;
        iiov[0].iov_len  = sizeof(buf);
        msg.msg_iov      = iiov;
        msg.msg_iovlen   = 1;
        msg.msg_name     = NULL;
        msg.msg_namelen  = 0;
        msg.msg_accrightright = (caddr_t) &newfd; /* addr of descriptor */
        msg.msg_accrightrightlen = sizeof(int); /* receive 1 descriptor */

        if ( (nread = recvmmsg(servfd, &msg, 0)) < 0)
            err_sys("recvmmsg error");
        else if (nread == 0) {
            err_ret("connection closed by server");
            return(-1);
        }

        /* See if this is the final data with null & status.
         * Null must be next to last byte of buffer, status
         * byte is last byte. Zero status means there must
         * be a file descriptor to receive. */
        for (ptr = buf; ptr < &buf[nread]; ) {
            if (*ptr++ == 0) {
                if (ptr != &buf[nread-1])
                    err_dump("message format error");
                status = *ptr & 255;
                if (status == 0) {
                    if (msg.msg_accrightrightlen != sizeof(int))
                        err_dump("status = 0 but no fd");
                    /* newfd = the new descriptor */
                } else
                    newfd = -status;
                nread -= 2;
            }
        }
    }
    if (nread > 0)
        if ((*userfunc)(STDERR_FILENO, buf, nread) != nread)
```

```

        return(-1);
    if (status >= 0)    /* final data has arrived */
        return(newfd); /* descriptor, or -status */
    }
}

```

注意，该程序总是准备接收一描述符（在每次调用 `recvmsg` 之前，设置 `msg_accrights` 和 `msg_accrighslen`），但是仅当在返回时 `msg_accrighslen` 非0，才确实接收到一描述符。

15.3.3 4.3+BSD

从4.3BSD Reno开始，更改了 `msg_hdr` 结构的定义。在以前版本中被称之为“存取权”的最后两个元素改称为“辅助数据”。另外，在该结构结束处增加了一个新成员 `msg_flags`。

```

struct msg_hdr {
    caddr_t    msg_name;        /* optional address */
    int        msg_namelen;     /* size of address */
    struct iovec *msg_iov;      /* scatter/gather array */
    int        msg_iovlen;     /* # elements in msg_iov array */
    caddr_t    msg_control;    /* ancillary data */
    u_int      msg_controllen; /* size of ancillary data */
    int        msg_flags;      /* flags on received message */
};

```

现在，`msg_control` 字段指向一个 `cmsghdr`（控制消息头）结构。

```

struct cmsghdr {
    u_int  cmsg_len; /* data byte count, including header */
    int    cmsg_level; /* originating protocol */
    int    cmsg_type; /* protocol-specific type */
    /* followed by the actual control message data */
};

```

为了发送一文件描述符，将 `cmsg_len` 设置为 `cmsghdr` 结构长度加一个整型（描述符）的长度。将 `cmsg_level` 设置为 `SOL_SOCKET`，`cmsg_type` 设置为 `SCM_RIGHTS`，这表明正在传送的是存取权（SCM表示套接口级控制消息）。实际描述符的存放位置紧随在 `cmsg_type` 字段之后，使用 `MSG_DATA` 宏以获得指向该整型数的指针。程序 15-9 示出了 4.3BSD Reno 之下的 `send_fd` 函数。

程序15-9 4.3BSD的 `send_fd` 函数

```

#include <sys/types.h>
#include <sys/socket.h> /* struct msg_hdr */
#include <sys/uio.h>    /* struct iovec */
#include <errno.h>
#include <stddef.h>
#include "ourhdr.h"

static struct cmsghdr *cmptr = NULL; /* buffer is malloc'ed first time */
#define CONTROLLEN (sizeof(struct cmsghdr) + sizeof(int))
                /* size of control buffer to send/rcv one file descriptor */

/* Pass a file descriptor to another process.
 * If fd<0, then -fd is sent back instead as the error status. */

int
send_fd(int clifd, int fd)
{
    ...
}

```

```

    struct iovec    iov[1];
    struct msghdr  msg;
    char          buf[2]; /* send_fd()/recv_fd() 2-byte protocol */

    iov[0].iov_base = buf;
    iov[0].iov_len  = 2;
    msg.msg_iov     = iov;
    msg.msg_iovlen  = 1;
    msg.msg_name    = NULL;
    msg.msg_namelen = 0;
    if (fd < 0) {
        msg.msg_control    = NULL;
        msg.msg_controllen = 0;
        buf[1] = -fd; /* nonzero status means error */
        if (buf[1] == 0)
            buf[1] = 1; /* -256, etc. would screw up protocol */
    } else {
        if (cmptr == NULL && (cmptr = malloc(CONTROLLEN)) == NULL)
            return(-1);
        cmptr->cmsg_level  = SOL_SOCKET;
        cmptr->cmsg_type   = SCM_RIGHTS;
        cmptr->cmsg_len    = CONTROLLEN;
        msg.msg_control    = (caddr_t) cmptr;
        msg.msg_controllen = CONTROLLEN;
        *(int *)CMSG_DATA(cmptr) = fd; /* the fd to pass */
        buf[1] = 0; /* zero status means OK */
    }
    buf[0] = 0; /* null byte flag to recv_fd() */
    if (sendmsg(clifd, &msg, 0) != 2)
        return(-1);
    return(0);
}

```

为了接收一描述符（见程序 15-10），我们为 cmsghdr 结构和一描述符分配了足够的存储区，设置 msg_control 使其指向所分配到的存储区，然后调用 recvmsg。

程序 15-10 4.3BSD Reno 的 recv_fd 函数

```

#include <sys/types.h>
#include <sys/socket.h> /* struct msghdr */
#include <sys/uio.h> /* struct iovec */
#include <stddef.h>
#include "ourhdr.h"

static struct cmsghdr *cmptr = NULL; /* malloc'ed first time */
#define CONTROLLEN (sizeof(struct cmsghdr) + sizeof(int))
/* size of control buffer to send/recv one file descriptor */
/* Receive a file descriptor from another process (a server).
 * In addition, any data received from the server is passed
 * to (*userfunc)(STDERR_FILENO, buf, nbytes). We have a
 * 2-byte protocol for receiving the fd from send_fd(). */
int
recv_fd(int servfd, ssize_t (*userfunc)(int, const void *, size_t))
{
    int          newfd, nread, status;
    char        *ptr, buf[MAXLINE];
    struct iovec iov[1];
    struct msghdr msg;
    status = -1;
    for ( ; ; ) {

```

```

    iov[0].iov_base = buf;
    iov[0].iov_len  = sizeof(buf);
    msg.msg_iov     = iov;
    msg.msg_iovlen  = 1;
    msg.msg_name    = NULL;
    msg.msg_namelen = 0;
    if (cmptr == NULL && (cmptr = malloc(CONTROLLEN)) == NULL)
        return(-1);
    msg.msg_control = (caddr_t) cmptr;
    msg.msg_controllen = CONTROLLEN;
    if ( (nread = recvmmsg(servfd, &msg, 0)) < 0)
        err_sys("recvmmsg error");
    else if (nread == 0) {
        err_ret("connection closed by server");
        return(-1);
    }
    /* See if this is the final data with null & status.
       Null must be next to last byte of buffer, status
       byte is last byte. Zero status means there must
       be a file descriptor to receive. */
    for (ptr = buf; ptr < &buf[nread]; ) {
        if (*ptr++ == 0) {
            if (ptr != &buf[nread-1])
                err_dump("message format error");
            status = *ptr & 255;
            if (status == 0) {
                if (msg.msg_controllen != CONTROLLEN)
                    err_dump("status = 0 but no fd");
                newfd = *(int *)CMSG_DATA(cmptr); /* new descriptor */
            } else
                newfd = -status;
            nread -= 2;
        }
    }
    if (nread > 0)
        if ((*userfunc)(STDERR_FILENO, buf, nread) != nread)
            return(-1);
    if (status >= 0) /* final data has arrived */
        return(newfd); /* descriptor, or -status */
}
}
}

```

15.4 open服务器第1版

目前，使用文件描述符传送技术开发了一个open服务器：它是一个可执行程序，由一个进程执行以打开一个或多个文件。该服务器不是将文件送回调用进程，而是送回一个打开文件描述符。这使该服务器对任何类型的文件（例如调制解调器线或网络连接）而不单是普通文件都能起作用。这也意味着，用IPC交换最小量的信息——从客户机到服务器传送文件名和打开方式，而从服务器到客户机返回描述符。文件内容则不需用IPC传送。

将服务器设计成一个单独的可执行程序有很多优点：

(1) 任一客户机都易于和服务器联系，这类似于客户机调用一库函数。不需要将一特定服务编码在应用程序中，而是设计一种可供重用的设施。

(2) 如若需要更改服务器，那么也只影响一个程序。相反，更新一库函数可能要更改调用此库函数的所有程序（用连编程序重新连接）。共享库函数可以简化这种更新。

(3) 服务器可以是设置-用户-ID程序，于是使其具有客户机没有的附加许可权。注意，一

个库函数（或共享库函数）不能提供这种能力。

客户机创建一流管道，然后调用 `fork` 和 `exec` 以调用服务器。客户机经流管道发送请求，服务器经管道回送响应。定义客户机和服务器间的协议如下：

(1) 客户机经流管道向服务器发送下列形式的请求：

```
open <pathname> <openmode>\0
```

`<openmode>` 是 `open` 函数的第二个参数，以十进制表示。该请求字符串以 `null` 字节结尾。

(2) 服务器调用 `send_fd` 或 `send_err` 回送一打开描述符或一条出错消息。

这是一个进程向其父进程发送一打开描述符的实例。15.6节将修改此实例，其中使用了一个精灵服务器，它将一个描述符发送给完全无关的进程。

程序15-11是头文件 `open.h`，它包括标准系统头文件，并且定义了各个函数原型。

程序15-11 open.h头文件

```
#include <sys/types.h>
#include <errno.h>
#include "ourhdr.h"

#define CL_OPEN "open"          /* client's request for server */

/* our function prototypes */
int csopen(char *, int);
```

程序15-12是 `main` 函数，其中包含一个循环，它先从标准输入读一个路径名，然后将该文件复制至标准输出。它调用函数 `csopen` 以与 `open` 服务器联系，从其返回一打开描述符。

程序15-12 main函数

```
#include "open.h"
#include <fcntl.h>

#define BUFFSIZE 8192

int
main(int argc, char *argv[])
{
    int n, fd;
    char buf[BUFFSIZE], line[MAXLINE];

    /* read filename to cat from stdin */
    while (fgets(line, MAXLINE, stdin) != NULL) {
        line[strlen(line) - 1] = 0; /* replace newline with null */

        /* open the file */
        if ( (fd = csopen(line, O_RDONLY)) < 0)
            continue; /* csopen() prints error from server */

        /* and cat to stdout */
        while ( (n = read(fd, buf, BUFFSIZE)) > 0)
            if (write(STDOUT_FILENO, buf, n) != n)
                err_sys("write error");
        if (n < 0)
            err_sys("read error");
        close(fd);
    }

    exit(0);
}
```

程序15-13是函数csopen，它先创建一流管道，然后进行服务器的fork和exec操作。

程序15-13 csopen函数

```

#include    "open.h"
#include    <sys/uio.h>    /* struct iovec */

/* Open the file by sending the "name" and "oflag" to the
 * connection server and reading a file descriptor back. */

int
csopen(char *name, int oflag)
{
    pid_t    pid;
    int    len;
    char    buf[10];
    struct iovec    iov[3];
    static int    fd[2] = { -1, -1 };

    if (fd[0] < 0) {    /* fork/exec our open server first time */
        if (s_pipe(fd) < 0)
            err_sys("s_pipe error");
        if ( (pid = fork()) < 0)
            err_sys("fork error");
        else if (pid == 0) {    /* child */
            close(fd[0]);
            if (fd[1] != STDIN_FILENO) {
                if (dup2(fd[1], STDIN_FILENO) != STDIN_FILENO)
                    err_sys("dup2 error to stdin");
            }
            if (fd[1] != STDOUT_FILENO) {
                if (dup2(fd[1], STDOUT_FILENO) != STDOUT_FILENO)
                    err_sys("dup2 error to stdout");
            }
            if (execl("./opend", "opend", NULL) < 0)
                err_sys("execl error");
        }
        close(fd[1]);    /* parent */
    }

    sprintf(buf, " %d", oflag);    /* oflag to ascii */
    iov[0].iov_base = CL_OPEN " ";
    iov[0].iov_len = strlen(CL_OPEN) + 1;
    iov[1].iov_base = name;
    iov[1].iov_len = strlen(name);
    iov[2].iov_base = buf;
    iov[2].iov_len = strlen(buf) + 1;    /* +1 for null at end of buf */
    len = iov[0].iov_len + iov[1].iov_len + iov[2].iov_len;
    if (writev(fd[0], &iov[0], 3) != len)
        err_sys("writev error");

    /* read descriptor, returned errors handled by write() */
    return( recv_fd(fd[0], write) );
}

```

子进程关闭管道的一端，父进程关闭另一端。子进程也为它所执行的服务器将管道的一端复制到其标准输入和标准输出0（另一种可选择的方案是将描述符fd[1]的ASCII表示形式作为一个参数传送给服务器。）

父进程将请求发送给服务器，请求中包含路径名和打开方式。最后，父进程调用recv_fd以返回描述符或错误消息。如果服务器返回一错误消息则调用write，向标准出错输出该消息。

现在，观察open服务器。其程序是opend，它由子进程执行（见程序15-13）。先观察

opend.h头文件（见程序15-14），它包括了系统头文件，并且说明了全局变量和函数原型。

程序15-14 opend.h头文件

```
#include <sys/types.h>
#include <errno.h>
#include "ourhdr.h"

#define CL_OPEN "open" /* client's request for server */

/* declare global variables */
extern char errmsg[]; /* error message string to return to client */
extern int oflag; /* open() flag: O_XXX ... */
extern char *pathname; /* of file to open() for client */

/* function prototypes */
int cli_args(int, char **);
void request(char *, int, int);
```

main函数（见程序15-15）经流管道（它的标准输入）读来自客户机的请求，然后调用函数request。

程序15-15 main函数

```
#include "opend.h"

/* define global variables */
char errmsg[MAXLINE];
int oflag;
char *pathname;

int
main(void)
{
    int nread;
    char buf[MAXLINE];

    for ( ; ; ) { /* read arg buffer from client, process request */
        if ( (nread = read(STDIN_FILENO, buf, MAXLINE)) < 0)
            err_sys("read error on stream pipe");
        else if (nread == 0)
            break; /* client has closed the stream pipe */

        request(buf, nread, STDOUT_FILENO);
    }
    exit(0);
}
```

程序15-16中的request函数承担全部工作。它调用函数buf_args将客户机请求分解成标准argv型的参数表，然后调用函数cli_args处理客户机的参数。如果一切正常，则调用open打开相应文件，接着调用send_fd，经由流管道（它的标准输出）将描述符回送给客户机。如果出错则调用send_err回送一则出错消息，其中使用了前面说明的客户机-服务器协议。

客户机请求是一个空的中断的字符串，其参数由空格分隔。程序15-17中的buf_args函数将字符串分解成标准argv型参数表，并调用用户函数处理参数。本节稍后及第18章将用到该函数。我们使用ANSI C函数strtok将字符串分割成参数。

程序15-16 request函数

```
#include "opend.h"
#include <fcntl.h>
```

```

void
request(char *buf, int nread, int fd)
{
    int    newfd;

    if (buf[nread-1] != 0) {
        sprintf(errmsg, "request not null terminated: %*.s\n",
                nread, nread, buf);
        send_err(fd, -1, errmsg);
        return;
    }

    /* parse the arguments, set options */
    if (buf_args(buf, cli_args) < 0) {
        send_err(fd, -1, errmsg);
        return;
    }

    if ( (newfd = open(pathname, oflag)) < 0) {
        sprintf(errmsg, "can't open %s: %s\n",
                pathname, strerror(errno));
        send_err(fd, -1, errmsg);
        return;
    }

    /* send the descriptor */
    if (send_fd(fd, newfd) < 0)
        err_sys("send_fd error");
    close(newfd);    /* we're done with descriptor */
}

```

程序15-17 buf_args函数

```

#include    "ourhdr.h"

#define MAXARGC    50 /* max number of arguments in buf */
#define WHITE     " \t\n" /* white space for tokenizing arguments */

/* buf[] contains white-space separated arguments. We convert it
 * to an argv[] style array of pointers, and call the user's
 * function (*optfunc)() to process the argv[] array.
 * We return -1 to the caller if there's a problem parsing buf,
 * else we return whatever optfunc() returns. Note that user's
 * buf[] array is modified (nulls placed after each token). */

int
buf_args(char *buf, int (*optfunc)(int, char **))
{
    char    *ptr, *argv[MAXARGC];
    int     argc;

    if (strtok(buf, WHITE) == NULL)    /* an argv[0] is required */
        return(-1);
    argv[argc = 0] = buf;

    while ( (ptr = strtok(NULL, WHITE)) != NULL) {
        if (++argc >= MAXARGC-1)    /* -1 for room for NULL at end */
            return(-1);
        argv[argc] = ptr;
    }
    argv[++argc] = NULL;

    return( (*optfunc)(argc, argv) );
    /* Since argv[] pointers point into the user's buf[],

```

```

    user's function can just copy the pointers, even
    though argv[] array will disappear on return. */
}

```

buf_args调用的服务器函数是cli_args（见程序15-18）。它验证客户机发送的参数是否正确，然后将路径名和打开方式存放在全局变量中。

这样也就完成了open服务器，它由客户机执行fork和exec而调用。在fork之前创建了一个流管道，然后客户机和服务器用其进行通信。在这种安排下，每个客户机都有一服务器。

在下一节观察了客户机-服务器连接后，我们将在15.6节重新实现一个open服务器，其中一个精灵进程作为服务器，所有客户机都与其进行联系。

程序15-18 cli_args函数

```

#include    "opend.h"

/* This function is called by buf_args(), which is called by
 * request().  buf_args() has broken up the client's buffer
 * into an argv[] style array, which we now process. */

int
cli_args(int argc, char **argv)
{
    if (argc != 3 || strcmp(argv[0], CL_OPEN) != 0) {
        strcpy(errmsg, "usage: <pathname> <oflag>\n");
        return(-1);
    }

    pathname = argv[1];    /* save ptr to pathname to open */
    oflag = atoi(argv[2]);
    return(0);
}

```

15.5 客户机-服务器连接函数

对于相关进程（例如，父进程和子进程）之间的IPC，流管道非常有用。前节所述的open服务器使用未命名的流管道能从子进程向父进程传送文件描述符。但是当处理无关进程时（例如，若服务器是一精灵进程），则需要使用有名的流管道。

可以先构造一未命名流管道（用s_pipe函数），然后对每一端加上一文件系统路径名。精灵进程服务器将只创建流管道的一端，并对该端加上一名字。这样，无关的客户机可以向服务者的流管道端发送消息，从而与精灵进程会聚。这类似于图14-11中所示的情况，在该图中客户机使用FIFO发送它们的请求。

一种更好的方法是：服务器创建一名字公开的流管道的一端，然后客户机连接至该端。另外，每次一个新客户机连至服务器的命名流管道时，就在客户机和服务器之间创建一条全新的流管道。这样，每次一个新客户机连接至服务器，以及客户机终止时，服务器都会得到通知。SVR4和4.3+BSD都支持这种形式的IPC。本节将开发三个函数，客户机-服务器可以使用这些函数以建立上述针对每个客户机的连接。

```

#include "ourhdr.h"

int serv_listen(const char*nm);

```

返回：若成功则返回为文件描述符，若出错则 <0

首先，一个服务器应当宣布，它愿意听取客户机在一个众所周知名字上的连接，该名字是在文件系统中的路径名。为此调用 `serv_listen`，其参数 `name` 是服务器的众所周知名字。客户机希望与服务器连接时使用此名字。该函数的返回值是命名流管道服务器端的文件描述符。

一旦服务器已调用 `serv_listen`，它将调用 `serv_accept` 等待客户连接到达。

```
#include "ourhdr.h"

int serv_accept(int listenfd, uid_t *uidptr);
```

返回：若成功则返回为文件描述符，若出错则 <0

`listenfd` 是 `serv_listen` 返回的描述符。在客户机连接到服务器众所周知的名字上之前，此函数并不返回。当客户机连接至服务器时，自动创建一条全新的流管道，其新描述符作为该函数的值返回。另外，客户机的有效用户 ID 通过指针 `uidptr` 存储。

客户机为与服务器连接只需调用 `cli_conn` 函数。

```
#include "ourhdr.h"

int cli_conn(const char *name);
```

返回：若成功则返回为文件描述符，若出错则 <0

客户指定的 `name` 应当与服务器调用 `serv_listen` 时宣布的相同。返回的描述符引用连接至服务器的流管道

使用上述三个函数，就可编写服务器精灵进程，它可以管理任一数量的客户机。唯一的限制是单个进程可用的描述符数，服务器对于每一个客户机连接都需要一个描述符。因为这些函数处理的都是普通文件描述符，所以服务器使用 `select` 或 `poll` 就可在所有客户机之间多路转接 I/O 请求。最后，因为客户机-服务器连接都是流管道，所以可以经由连接传送打开描述符。

下面两节将说明在 SVR4 和 4.3+BSD 之下这三个函数的实现。第 18 章开发一个通用的连接服务器时，也将使用这三个函数。

15.5.1 SVR4

SVR4 提供装配的流以及一个名为 `connld` 的流处理模块，用其可以提供与服务器有唯一连接的命名流管道。

装配流和 `connld` 模块是由 Presotto 和 Ritchie [1990] 为 Research UNIX 系统开发的，后来由 SVR4 采用。

首先，服务器创建一未命名流管道，并将流处理模块 `connld` 压入一端。图 15-5 显示了这一处理结果。

然后，使压入 `connld` 的一端具有一路径名。SVR4 提供 `fattach` 函数实现这一点。任一进程（例如客户机）打开此路径名就引用该管道的命名端。

程序 15-19 使用了 20 余行代码实现 `serv_listen` 函数。

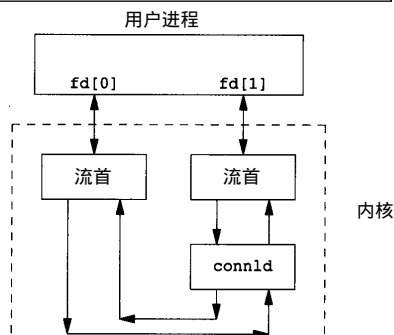


图15-5 在一端压入 `connld` 模块后的流管道

程序15-19 SVR4的serv_listen函数

```

#include <sys/types.h>
#include <sys/stat.h>
#include <stropts.h>
#include "ourhdr.h"

#define FIFO_MODE (S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH)
/* user rw, group rw, others rw */

int /* returns fd if all OK, <0 on error */
serv_listen(const char *name)
{
    int tempfd, fd[2], len;

    /* create a file: mount point for fattach() */
    unlink(name);
    if ( (tempfd = creat(name, FIFO_MODE)) < 0)
        return(-1);
    if (close(tempfd) < 0)
        return(-2);

    if (pipe(fd) < 0)
        return(-3);

    /* push connld & fattach() on fd[1] */
    if (ioctl(fd[1], I_PUSH, "connld") < 0)
        return(-4);
    if (fattach(fd[1], name) < 0)
        return(-5);

    return(fd[0]); /* fd[0] is where client connections arrive */
}

```

当另一进程对管道的命名端（connld模块压入端）调用open时，发生下列处理过程：

- (1) 创建一个新管道。
- (2) 该新管道的一个描述符作为open的返回值回送给客户机。
- (3) 另一个描述符在命名管道的另一端（亦即不是压入 connld的端）传送给服务器。服务器以带I_RECVFD命令的ioctl接受该新描述符。

假定服务器用fattach函数加到其管道的众所周知的名字是/tmp/serv1。图15-6显示了客户机调用：

```
fd=open("/tmp/serv1", O_RDWR);
```

并返回后产生的结果。

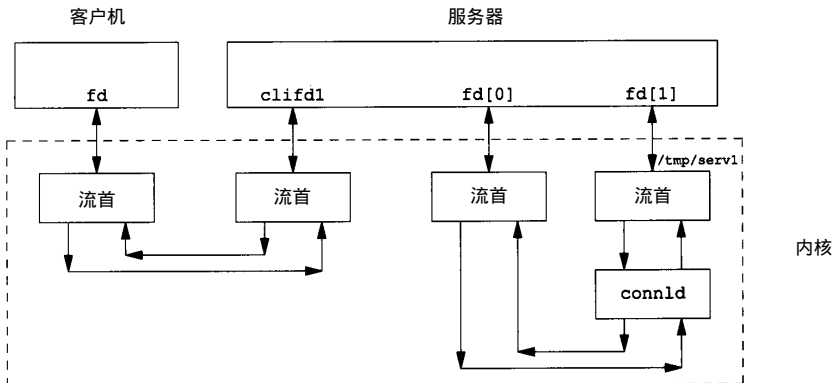


图15-6 客户机-服务器在命名管道上的连接

客户机和服务器之间的管道是open创建的，被打开的路径名实际上是一命名管道，其中压入了connld模块。客户机得到由open返回的文件描述符fd。服务器处的新文件描述符是clifdl，它是由服务器在描述符fd[0]上以I_RECVFD命令调用ioctl而接收到的。一旦服务器在fd[1]上压入了connld模块，并对fd[1]外接上一个名字，它就不再使用fd[1]。

服务器调用程序15-20中的serv_accept函数等待客户机连接到达。

程序15-20 SVR4的serv_accept函数

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stropts.h>
#include "ourhdr.h"

/* Wait for a client connection to arrive, and accept it.
 * We also obtain the client's user ID. */

int /* returns new fd if all OK, -1 on error */
serv_accept(int listenfd, uid_t *uidptr)
{
    struct strrecvfd recvfd;

    if (ioctl(listenfd, I_RECVFD, &recvfd) < 0)
        return(-1); /* could be EINTR if signal caught */

    if (uidptr != NULL)
        *uidptr = recvfd.uid; /* effective uid of caller */

    return(recvfd.fd); /* return the new descriptor */
}
```

在图15-6中，serv_accept的第一个参数应当是描述符fd[0]，serv_accept的返回值是描述符clifdl。

客户机调用程序15-21中的cli_conn函数起动对服务器的连接。

程序15-21 SVR4的cli_conn函数

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

/* Create a client endpoint and connect to a server. */

int /* returns fd if all OK, <0 on error */
cli_conn(const char *name)
{
    int fd;

    /* open the mounted stream */
    if ( (fd = open(name, O_RDWR)) < 0)
        return(-1);
    if (isastream(fd) == 0)
        return(-2);

    return(fd);
}
```

我们对返回的描述符是否引用一个流设备进行了两次检查，以便处理服务器没有起动，但该路径名却存在于文件系统中的情况。（在SVR4下，几乎没有什么理由去调用cli_conn，而不

是直接调用 open。下一节将看到，在 BSD 系统之下，cli_conn 函数要复杂得多，因此编写 cli_conn 函数就很必要。)

15.5.2 4.3+BSD

在 4.3+BSD 之下，为了用 UNIX 域套接口连接客户机和服务器，需要有一套不同的操作函数。因为应用 socket、bind、listen、accept 和 connect 函数的大部分细节与其他网络协议有关（参见 Stevens [1990]），所以此处不详细展开。

因为 SVR4 也支持 UNIX 域套接口，所以本节所示代码同样可在 SVR4 之下工作。

程序 15-22 包含了 serv_listen 函数。它是服务器调用的第一个函数。

程序 15-22 4.3+BSD 的 serv_listen 函数

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include "ourhdr.h"

/* Create a server endpoint of a connection. */

int /* returns fd if all OK, <0 on error */
serv_listen(const char *name)
{
    int fd, len;
    struct sockaddr_un unix_addr;

    /* create a Unix domain stream socket */
    if ( (fd = socket(AF_UNIX, SOCK_STREAM, 0)) < 0)
        return(-1);

    unlink(name); /* in case it already exists */

    /* fill in socket address structure */
    memset(&unix_addr, 0, sizeof(unix_addr));
    unix_addr.sun_family = AF_UNIX;
    strcpy(unix_addr.sun_path, name);
#ifdef SCM_RIGHTS /* 4.3BSD Reno and later */
    len = sizeof(unix_addr.sun_len) + sizeof(unix_addr.sun_family) +
        strlen(unix_addr.sun_path) + 1;
    unix_addr.sun_len = len;
#else /* vanilla 4.3BSD */
    len = strlen(unix_addr.sun_path) + sizeof(unix_addr.sun_family);
#endif

    /* bind the name to the descriptor */
    if (bind(fd, (struct sockaddr *) &unix_addr, len) < 0)
        return(-2);

    if (listen(fd, 5) < 0) /* tell kernel we're a server */
        return(-3);

    return(fd);
}
```

首先，调用 socket 函数创建一个 UNIX 域套接口。然后，填充 sockeraddr_un 结构，将一个众所周知的路径名赋与该套接口。该结构是调用 bind 函数的一个参数。然后调用 listen 以通知内核：本服务器正等待来自客户机的连接。（listen 的第二个参数是 5，它是最大的未决连接请求数，

内核将这些请求对该描述符进行排队。大多数实现强制该值的上限为5。))

客户机调用cli_conn函数(见程序15-23)起动与服务器的连接。

程序15-23 4.3+BSD的cli_conn函数

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/stat.h>
#include <sys/un.h>
#include "ourhdr.h"

/* Create a client endpoint and connect to a server. */

#define CLI_PATH "/var/tmp/" /* +5 for pid = 14 chars */
#define CLI_PERM S_IRWXU /* rwx for user only */

int /* returns fd if all OK, <0 on error */
cli_conn(const char *name)
{
    int fd, len;
    struct sockaddr_un unix_addr;

    /* create a Unix domain stream socket */
    if ( (fd = socket(AF_UNIX, SOCK_STREAM, 0)) < 0)
        return(-1);

    /* fill socket address structure w/our address */
    memset(&unix_addr, 0, sizeof(unix_addr));
    unix_addr.sun_family = AF_UNIX;
    sprintf(unix_addr.sun_path, "%s%05d", CLI_PATH, getpid());
#ifdef SCM_RIGHTS /* 4.3BSD Reno and later */
    len = sizeof(unix_addr.sun_len) + sizeof(unix_addr.sun_family) +
        strlen(unix_addr.sun_path) + 1;
    unix_addr.sun_len = len;
#else /* vanilla 4.3BSD */
    len = strlen(unix_addr.sun_path) + sizeof(unix_addr.sun_family);
    if (len != 16)
        err_quit("length != 16"); /* hack */
#endif

    unlink(unix_addr.sun_path); /* in case it already exists */
    if (bind(fd, (struct sockaddr *) &unix_addr, len) < 0)
        return(-2);
    if (chmod(unix_addr.sun_path, CLI_PERM) < 0)
        return(-3);

    /* fill socket address structure w/server's addr */
    memset(&unix_addr, 0, sizeof(unix_addr));
    unix_addr.sun_family = AF_UNIX;
    strcpy(unix_addr.sun_path, name);
#ifdef SCM_RIGHTS /* 4.3BSD Reno and later */
    len = sizeof(unix_addr.sun_len) + sizeof(unix_addr.sun_family) +
        strlen(unix_addr.sun_path) + 1;
    unix_addr.sun_len = len;
#else /* vanilla 4.3BSD */
    len = strlen(unix_addr.sun_path) + sizeof(unix_addr.sun_family);
#endif

    if (connect(fd, (struct sockaddr *) &unix_addr, len) < 0)
        return(-4);

    return(fd);
}
```


调用socket函数以创建客户端的UNIX域套接口，然后客户机专用的名字填入 socketaddr_un 结构。该路径名的最后5个字符是客户机的进程ID（我们可以查证此结构的长度是14个字符，以避免UNIX域套接口早期实现的某些错误）。在路径名已经存在的情况下调用 unlink，然后再调用bind将一名字赋与客户机的套接口，这就创建了文件系统中的路径名，该文件的类型是套接口。接着调用 chmod，它关闭除 user_read，user_write 和 user_execute 以外的存取权。在 serv_accept 中，服务器检查该套接口的这些许可权和用户ID，以验证用户的身份。

然后，以服务器众所周知的路径名填充另一个 socketaddr_un 结构。最后，connect 函数启动与服务器的连接。

创建每个客户机与服务器的唯一连接是通过在 serv_accept 函数中调用 accept 函数实现的（见程序15-24）。

程序15-24 4.3+BSD的serv_accept函数

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/stat.h>
#include <sys/un.h>
#include <stddef.h>
#include <time.h>
#include "ourhdr.h"

#define STALE 30 /* client's name can't be older than this (sec) */

/* Wait for a client connection to arrive, and accept it.
 * We also obtain the client's user ID from the pathname
 * that it must bind before calling us. */

int /* returns new fd if all OK, <0 on error */
serv_accept(int listenfd, uid_t *uidptr)
{
    int          clifd, len;
    time_t       staletime;
    struct sockaddr_un  unix_addr;
    struct stat   statbuf;

    len = sizeof(unix_addr);
    if ( (clifd = accept(listenfd, (struct sockaddr *) &unix_addr, &len)) < 0)
        return(-1); /* often errno=EINTR, if signal caught */

    /* obtain the client's uid from its calling address */
#ifdef SCM_RIGHTS /* 4.3BSD Reno and later */
    len -= sizeof(unix_addr.sun_len) - sizeof(unix_addr.sun_family);
#else /* vanilla 4.3BSD */
    len -= sizeof(unix_addr.sun_family); /* len of pathname */
#endif
    unix_addr.sun_path[len] = 0; /* null terminate */

    if (stat(unix_addr.sun_path, &statbuf) < 0)
        return(-2);
#ifdef S_ISSOCK /* not defined for SVR4 */
    if (S_ISSOCK(statbuf.st_mode) == 0)
        return(-3); /* not a socket */
#endif
    if ((statbuf.st_mode & (S_IRWXG | S_IRWXO)) ||
        (statbuf.st_mode & S_IRWXU) != S_IRWXU)
        return(-4); /* is not rwx----- */

    staletime = time(NULL) - STALE;
    if (statbuf.st_atime < staletime ||
```

```

    statbuf.st_ctime < staletime ||
    statbuf.st_mtime < staletime)
        return(-5); /* i-node is too old */

    if (uidptr != NULL)
        *uidptr = statbuf.st_uid; /* return uid of caller */

    unlink(unix_addr.sun_path); /* we're done with pathname now */

    return(clifd);
}

```

服务器在调用accept中堵塞以等待客户机调用cli_conn。当accept返回时，其返回值是连向客户机的全新的描述符（这类似于SVR4中connld模块所做的）。另外，accept也通过其第二个参数（指向socketaddr_un结构的指针）返回客户机赋予其套接口的路径名（它包含客户机的进程ID）。用null字节结束此路径名，然后调用stat。这使我们验证此路径名确实是一个套接口，其许可权user_read，user_write和user_execute。我们也验证与该套接口相关的三个时间不超过30秒。（time函数返回自UNIX纪元经过的时间和日期，它们都以秒计。）如果所有这些检查都通过，则认为该客户机的身份（其有效用户ID）是该套接口的所有者。虽然这种检查并不完善，但却是现有系统所能做得最好的。（如果内核能像SVR4I_RECVFD做的那样，将有效用户ID返回给accept，那就更好一些。）

图15-7显示了cli_conn调用返回后的这种连接，假定服务器众所周知的名字是/tmp/serv1。请将此图与图15-6相比较。

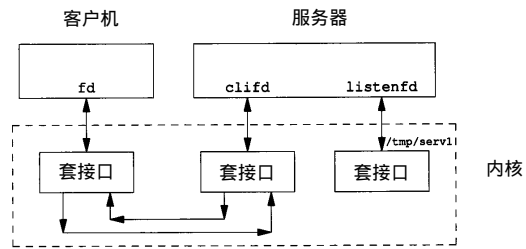


图15-7 UNIX域套接口上客户机-服务器连接

15.6 open服务器第2版

在15.4节中，客户机调用fork和exec构造了一个open服务器，它说明了如何从子程序向父程序传送文件描述符。本节将开发一个精灵进程样式的open服务器。一个服务器处理所有客户机的请求。由于避免使用了fork和exec，我们期望这一设计会更有效的。在客户机和服务器之间仍将使用上一节说明的三个函数：serv_listen、serv_accept和cli_conn。这一服务器将表明：一个服务器可以处理多个客户机，为此使用的技术是12.5节中说明的select和poll函数。

本节所述的客户机类似于15.4节中的客户机。确实，文件main.c是完全相同的（见程序15-12）。在open.h头文件（见程序15-11）中则加了下面1行：

```
#define CS_OPEN "/home/stevens/open" /* server's well-known name */
```

因为在这里调用的是cli_conn而非fork和exec，所以文件open.c与程序15-13完全不同。这示于程序15-25中。

程序15-25 csopen函数

```

#include "open.h"
#include <sys/uio.h> /* struct iovec */

/* Open the file by sending the "name" and "oflag" to the
 * connection server and reading a file descriptor back. */

int

```

```

csoopen(char *name, int oflag)
{
    int            len;
    char           buf[10];
    struct iovec   iov[3];
    static int     csfd = -1;

    if (csfd < 0) { /* open connection to conn server */
        if ( (csfd = cli_conn(CS_OPEN)) < 0)
            err_sys("cli_conn error");
    }

    sprintf(buf, " %d", oflag); /* oflag to ascii */
    iov[0].iov_base = CL_OPEN " ";
    iov[0].iov_len  = strlen(CL_OPEN) + 1;
    iov[1].iov_base = name;
    iov[1].iov_len  = strlen(name);
    iov[2].iov_base = buf;
    iov[2].iov_len  = strlen(buf) + 1;
    /* null at end of buf always sent */
    len = iov[0].iov_len + iov[1].iov_len + iov[2].iov_len;
    if (writev(csfd, &iov[0], 3) != len)
        err_sys("writev error");

    /* read back descriptor */
    /* returned errors handled by write() */
    return( recv_fd(csfd, write) );
}

```

客户机与服务器之间使用的协议仍然相同。

让我们先查看服务器。头文件 `opend.h` (见程序 15-26) 包括了标准头文件, 并且说明了全局变量和函数原型。

程序 15-26 `open.h` 头文件

```

#include <sys/types.h>
#include <errno.h>
#include "ourhdr.h"

#define CS_OPEN "/home/stevens/opend" /* well-known name */
#define CL_OPEN "open" /* client's request for server */

/* declare global variables */
extern int debug; /* nonzero if interactive (not daemon) */
extern char errmsg[]; /* error message string to return to client */
extern int oflag; /* open flag: O_XXX ... */
extern char *pathname; /* of file to open for client */

typedef struct { /* one Client struct per connected client */
    int fd; /* fd, or -1 if available */
    uid_t uid;
} Client;

extern Client *client; /* ptr to malloc'ed array */
extern int client_size; /* # entries in client[] array */
/* (both manipulated by client_XXX() functions) */

/* function prototypes */
int cli_args(int, char **);
int client_add(int, uid_t);
void client_del(int);
void loop(void);
void request(char *, int, int, uid_t);

```

因为此服务器处理所有客户机, 所以它必须保存每个客户机连接的状态。这是用定义在 `opend.h` 头文件中的 `client` 数组实现的。程序 15-27 定义了三个处理此数组的函数。

程序15-27 处理client数组的三个函数

```
#include "opend.h"

#define NALLOC 10 /* #Client structs to alloc/realloc for */

static void
client_alloc(void) /* alloc more entries in the client[] array */
{
    int i;
    if (client == NULL)
        client = malloc(NALLOC * sizeof(Client));
    else
        client = realloc(client, (client_size + NALLOC) * sizeof(Client));
    if (client == NULL)
        err_sys("can't alloc for client array");
    /* have to initialize the new entries */
    for (i = client_size; i < client_size + NALLOC; i++)
        client[i].fd = -1; /* fd of -1 means entry available */
    client_size += NALLOC;
}

/* Called by loop() when connection request from a new client arrives */
int
client_add(int fd, uid_t uid)
{
    int i;
    if (client == NULL) /* first time we're called */
        client_alloc();
again:
    for (i = 0; i < client_size; i++) {
        if (client[i].fd == -1) { /* find an available entry */
            client[i].fd = fd;
            client[i].uid = uid;
            return(i); /* return index in client[] array */
        }
        /* client array full, time to realloc for more */
        client_alloc();
        goto again; /* and search again (will work this time) */
    }
}

/* Called by loop() when we're done with a client */
void
client_del(int fd)
{
    int i;
    for (i = 0; i < client_size; i++) {
        if (client[i].fd == fd) {
            client[i].fd = -1;
            return;
        }
    }
    log_quit("can't find client entry for fd %d", fd);
}
```

第一次调用client_add时，它调用client_alloc、client_alloc又调用malloc为该数组的10个登记项分配空间。在这10个登记项全部用完后，再调用client_add，使realloc分配附加空间。依靠这种动态空间分配，我们无需在编译时限制client数组的长度。

如果出错，那么因为假定服务器是精灵进程，所以这些函数调用 `log_` 函数（见附录 B）。`main` 函数（见程序 15-28）定义全局变量，处理命令行选择项，然后调用 `loop` 函数。如果以 `-d` 选择项调用服务器，则它以交互方式运行而非精灵进程。当测试些服务器时，使用交互运行方式。

程序 15-28 `main` 函数

```

#include    "opend.h"
#include    <syslog.h>

        /* define global variables */
int        debug;
char       errmsg[MAXLINE];
int        oflag;
char       *pathname;
Client     *client = NULL;
int        client_size;

int
main(int argc, char *argv[])
{
    int     c;

    log_open("open.serv", LOG_PID, LOG_USER);

    opterr = 0;      /* don't want getopt() writing to stderr */
    while ( (c = getopt(argc, argv, "d")) != EOF) {
        switch (c) {
            case 'd':      /* debug */
                debug = 1;
                break;

            case '?':
                err_quit("unrecognized option: -%c", optopt);
        }
    }

    if (debug == 0)
        daemon_init();

    loop();      /* never returns */
}

```

`loop` 函数是服务器的无限循环。我们将给出该函数的两种版本。程序 15-29 是使用 `select` 的一种版本。（在 4.3+BSD 和 SVR4 之下工作），程序 15-30 是使用 `poll`（用于 SVR4）的另一种版本。

程序 15-29 使用 `select` 的 `loop` 函数

```

#include    "opend.h"
#include    <sys/time.h>

void
loop(void)
{
    int     i, n, maxfd, maxi, listenfd, clifd, nread;
    char    buf[MAXLINE];
    uid_t   uid;
    fd_set  rset, allset;

    FD_ZERO(&allset);

        /* obtain fd to listen for client requests on */

```

```

if ( (listenfd = serv_listen(CS_OPEN)) < 0)
    log_sys("serv_listen error");
FD_SET(listenfd, &allset);
maxfd = listenfd;
maxi = -1;

for ( ; ; ) {
    rset = allset;          /* rset gets modified each time around */
    if ( (n = select(maxfd + 1, &rset, NULL, NULL, NULL)) < 0)
        log_sys("select error");

    if (FD_ISSET(listenfd, &rset)) {
        /* accept new client request */
        if ( (clifd = serv_accept(listenfd, &uid)) < 0)
            log_sys("serv_accept error: %d", clifd);
        i = client_add(clifd, uid);
        FD_SET(clifd, &allset);
        if (clifd > maxfd)
            maxfd = clifd; /* max fd for select() */
        if (i > maxi)
            maxi = i;      /* max index in client[] array */
        log_msg("new connection: uid %d, fd %d", uid, clifd);
        continue;
    }

    for (i = 0; i <= maxi; i++) { /* go through client[] array */
        if ( (clifd = client[i].fd) < 0)
            continue;
        if (FD_ISSET(clifd, &rset)) {
            /* read argument buffer from client */
            if ( (nread = read(clifd, buf, MAXLINE)) < 0)
                log_sys("read error on fd %d", clifd);
            else if (nread == 0) {
                log_msg("closed: uid %d, fd %d",
                    client[i].uid, clifd);
                client_del(clifd); /* client has closed conn */
                FD_CLR(clifd, &allset);
                close(clifd);
            } else /* process client's request */
                request(buf, nread, clifd, client[i].uid);
        }
    }
}
}
}

```

此函数调用 `serv_listen` 以创建服务器对于客户机连接的端点。此函数的其余部分是一个循环，它以 `select` 调用开始。在 `select` 返回后，两个条件可能为真：

(1) 描述符 `listenfd` 可能准备好读，这意味着新客户机已调用了 `cli_conn`。为了处理这种情况。我们将调用 `serv_accept`，然后更新 `client` 数组以及与该新客户机相关的簿记消息。（跟踪作为 `select` 第一个参数的最高描述符编号。也跟踪使用中的 `client` 数组的最高下标。）

(2) 一个现存的客户机的连接可能准备好读。这意味这下列两事件之一：(a) 该客户机已经终止，或 (b) 该客户机已发送一新请求。如果 `read` 返回 0（文件结束），则可认为一客户机终止。如果读返回值大于 0 则可判定有一新请求需处理，调用 `request` 处理此新的客户机请求。

用 `allset` 描述符集跟踪当前使用的描述符。当新客户机连至服务器时，此描述符集的适当位被打开。当该客户机终止时，适当位就被关闭。

因为客户机的所有描述符都由内核自动关闭（包括与服务器的连接），所以我们知道什么时候一客户机终止，该终止是否自愿。这与系统 V IPC 机构不同。

使用poll函数的loop函数示于程序15-30中。

程序15-30 使用poll的loop函数

```

#include    "opend.h"
#include    <poll.h>
#include    <stropts.h>

void
loop(void)
{
    int          i, maxi, listenfd, clifd, nread;
    char         buf[MAXLINE];
    uid_t        uid;
    struct pollfd *pollfd;

    if ( (pollfd = malloc(open_max() * sizeof(struct pollfd))) == NULL)
        err_sys("malloc error");

        /* obtain fd to listen for client requests on */
    if ( (listenfd = serv_listen(CS_OPEN)) < 0)
        log_sys("serv_listen error");
    client_add(listenfd, 0); /* we use [0] for listenfd */
    pollfd[0].fd = listenfd;
    pollfd[0].events = POLLIN;
    maxi = 0;
    for ( ; ; ) {
        if (poll(pollfd, maxi + 1, INFTIM) < 0)
            log_sys("poll error");

        if (pollfd[0].revents & POLLIN) {
            /* accept new client request */
            if ( (clifd = serv_accept(listenfd, &uid)) < 0)
                log_sys("serv_accept error: %d", clifd);
            i = client_add(clifd, uid);
            pollfd[i].fd = clifd;
            pollfd[i].events = POLLIN;
            if (i > maxi)
                maxi = i;
            log_msg("new connection: uid %d, fd %d", uid, clifd);
        }

        for (i = 1; i <= maxi; i++) {
            if ( (clifd = client[i].fd) < 0)
                continue;
            if (pollfd[i].revents & POLLHUP)
                goto hungup;
            else if (pollfd[i].revents & POLLIN) {
                /* read argument buffer from client */
                if ( (nread = read(clifd, buf, MAXLINE)) < 0)
                    log_sys("read error on fd %d", clifd);
                else if (nread == 0) {
hungup:
                    log_msg("closed: uid %d, fd %d",
                        client[i].uid, clifd);
                    client_del(clifd); /* client has closed conn */
                    pollfd[i].fd = -1;
                    close(clifd);
                } else /* process client's request */
                    request(buf, nread, clifd, client[i].uid);
            }
        }
    }
}

```

为使打开描述符的数量能与客户机数量相当，我们动态地为 pollfd结构分配空间（函数 open_max见程序2-3）。

client数组的第0个登记项用于listenfd描述符。于是，client数组中的客户机下标号与pollfd数组中所用的下标号相同。新客户机连接的到达由listenfd描述符中的POLLIN指示。如同前述，调用serv_accept以接收该连接。

对于一个现存的客户机，应当处理来自poll的两个不同事件：客户机终止由POLLHUP指示，以及来自现存客户的一个新要求由POLLIN指示。回忆习题14.7，在还有数据在流首可读时，挂起消息能到达流首。对于管道，在处理挂起前，我们希望先读所有数据。但是对于服务器，当从客户机接收到挂起消息时，能关闭该流连接，于是也就丢弃了仍在流上的所有数据。因为已经不能回送任何响应，所以也就没有理由再去处理仍在流上的任何请求。

如同本函数的select版本，调用request函数（见程序15-31）处理来自客户机的新请求。此函数类似于其早期版本（见程序15-16）。它调用同一函数buf_args（见程序15-17），buf_args又调用cli_args（见程序15-18）。

程序15-31 request函数

```
#include "opend.h"
#include <fcntl.h>

void
request(char *buf, int nread, int clifd, uid_t uid)
{
    int newfd;

    if (buf[nread-1] != 0) {
        sprintf(errmsg, "request from uid %d not null terminated: %*.s\n",
                uid, nread, nread, buf);
        send_err(clifd, -1, errmsg);
        return;
    }
    log_msg("request: %s, from uid %d", buf, uid);

    /* parse the arguments, set options */
    if (buf_args(buf, cli_args) < 0) {
        send_err(clifd, -1, errmsg);
        log_msg(errmsg);
        return;
    }

    if ( (newfd = open(pathname, oflag)) < 0) {
        sprintf(errmsg, "can't open %s: %s\n",
                pathname, strerror(errno));
        send_err(clifd, -1, errmsg);
        log_msg(errmsg);
        return;
    }

    /* send the descriptor */
    if (send_fd(clifd, newfd) < 0)
        log_sys("send_fd error");
    log_msg("sent fd %d over fd %d for %s", newfd, clifd, pathname);
    close(newfd); /* we're done with descriptor */
}
```

这样就完成了open服务器，它使用一个精灵进程处理所有的客户机请求。

15.7 小结

本章集中讨论了如何在进程间传送文件描述符，及服务器如何接受来自客户机的连接，并演示了SVR4和4.3+BSD中的实现过程。目前大多数UNIX系统都提供这些高级IPC功能。第18章还将再次用到本章所述的函数。

本章给出了open服务器的两个版本。一个版本由客户机用fork和exec直接调用，另一版本为处理所有客户机请求的精灵进程服务器。这两个版本均采用了15.3节所述的文件描述符传送和接收函数。第二个版本还采用了15.5节所述的客户机-服务器连接函数以及12.5节所述的多路转接函数。

习题

15.1 改写程序15-1，要求是：对于流管道使用标准I/O库函数代替read和write。

15.2 使用本章说明的文件描述符传送函数以及8.8节中说明的父-子进程同步例程，编写具有下列功能的程序：该程序调用fork，然后子进程打开一现存文件并将打开的描述符传给父进程。父进程读该文件的当前位移量，并打印它以便验证。若此文件如上述从子进程传递到父进程，则父、子进程应共享同一文件表项，所以当子进程每次更改该文件当前位移量，那么这种更改同样影响到父进程的描述符。使子进程将该文件定位至一个不同位移量，并通知父进程。

15.3 程序15-14和15.15分别定义和说明了全局变量，两者的区别是什么？

15.4 改写bug_args函数（见程序15-17），删除其中对argv数组长度的编译时间限制。请用动态存储分配。

15.5 说明优化程序15-29和程序15-30中loop函数的方法，并实现之。

第16章 数据库函数库

16.1 引言

80年代早期，UNIX环境被认为不适合运行多用户数据库系统（见 Stonebraker [1981] 和 Weinberger [1982]）。早期的系统，如 V7，因为没有提供任何形式的IPC机制（除了半双工管道），也没有提供任何形式的记录锁机制，所以确实不适合运行多用户数据库。新一些的系统，象SVR4和4.3+BSD，则为运行可靠的、多用户的数据库系统提供了一个适合的环境。很多商业公司在许多年前就已提供这种系统。

本章将设计一个简单的、多用户数据库的函数库。通过此函数库提供的 C语言函数，其他程序可以访问数据库中的记录。这个C函数库只是一个完整的数据库的很小的一部分，并不包括其他很多部分，如查询语言等，关于其他部分可以参阅专门介绍数据库的书。我们感兴趣的是一个数据库函数库与UNIX的接口，以及这些接口与前面各章节的关系（如12.3节的记录锁）。

16.2 历史

dbm(3) 是一个在UNIX系统中很流行的数据库函数库，它由 Ken Thompson开发，使用了动态散列结构。最初，它与 V7一起提供，并出现在所有伯克利的版本中，也包含在伯克利的SVR4兼容函数库中。Seltzer和Yigit [1991] 中有关于dbm函数库使用的动态散列算法历史的详细介绍，以及这个库的其他实现方法。但是，这些实现的一个致命缺点是它们都不支持多个进程对数据库的并发修改。它们都没有提供并发控制（如记录锁）。

4.3+BSD提供了一个新的库db(3)，这个库支持三种不同的访问方式：面向记录、散列和 B树。同样，db也没有提供并发控制（这一点在 db手册的BUGS栏中说得很清楚）。Seltzer和Olson [1992] 中说以后的版本将提供像大部分商业数据库系统一样的并发控制功能。

绝大部分商业的数据库函数库提供多进程并发修改一个数据库所需要的并发控制。这些系统一般都使用12.3节中介绍的建议记录锁，并且用B+树来实现他们的数据库。

16.3 函数库

本节将定义数据库函数库的C语言接口，下一节再讨论其实现。

当打开一个数据库时，通过返回值得到一个DB结构的指针。这一点很像通过fopen得到一个FILE结构的指针（见5.2节），以及通过opendir得到一个DIR结构的指针（见4.21节）。我们将用此指针作为参数来调用以后的数据库函数。

```
#include "db.h"
```

```
DB *db_open(const char*name, int oflag, int mode);
```

返回：若成功则为DB结构的指针，若失败则为NULL

```
void db_close(DBdb*);
```

如果db_open成功返回，则将建立两个文件：*pathname.idx*和*pathname.dat*，*pathname.idx*是索引文件，*pathname.dat*是数据文件。*oflag*被用作第二个参数传递给open（见3.3节），表明这些文件的打开模式（只读、读写或如果文件不存在则建立等）。如果需要建立新的数据库，*mode*将作为第三个参数传递给open（文件访问权限）。

当不再使用数据库时，调用db_close来关闭数据库。db_close将关闭索引文件和数据文件，并释放数据库使用过程中分配的所有用于内部缓冲的存储空间。

当向数据库中加入一条新的记录时，必须提供一个此记录的关键字，以及与此关键字相联系的数据。如果此数据库存储的是人事信息，关键字可以是雇员号，数据可以是此雇员的姓名、地址、电话号码、受聘日等等。我们的实现要求关键字必须是唯一的（比方说，不会有两个雇员记录有同样的雇员号）。

```
#include "db.h"

int db_store(DB *db, const char *key, const char *data, int flag);
```

返回：若成功则为0，若错误则为非0(见下)

*key*和*data*是由NULL结束的字符串。它们可以包含除了NULL外的任何字符，如换行符。

*flag*只能是DB_INSERT（加一条新记录）或DB_REPLACE（替换一条已有的记录）。这两个常数定义在db.h头文件中。如果使用DB_REPLACE，而记录不存在，则返回值为-1。如果使用DB_INSERT，而记录已经存在，则返回值为1。

通过提供关键字*key*可以从数据库中取出一条记录。

```
#include "db.h"

char *db_fetch(DB *db, const char *key,);
```

返回：若成功则为指向数据的指针，若记录没有找到则为NULL

如果记录找到了，返回的指针指向与关键字联系在一起的数据。

通过提供关键字*key*，也可以从数据库中删除一条记录。

```
#include "db.h"

int db_delete(DB *db, const char *key);
```

返回：若成功则为0，若记录没有找到则为-1

除了通过关键字访问数据库外，也可以一条一条记录地访问数据库。因此，首先调用db_rewind回到数据库的第一条记录，再调用db_nextrec顺序地读每个记录。

```
#include "db.h"

void db_rewind(DB *db);

char *db_nextrec(DB *db, char *key);
```

返回：若成功则返回指向数据的指针，若到达数据库的尾端则为NULL

如果*key*是非NULL的指针，db_nextrec将当前记录的关键字存入*key*中。

db_nextrec不保证记录访问的次序，只保证每一条记录被访问恰好一次。如果顺序存储三条关键字分别为A、B、C的记录，则无法确定db_nextrec将按什么顺序返回这三条记录。它可

能按B、A、C的顺序返回，也可能按其他顺序。实际的顺序由数据库的实现决定。

这七个函数提供了数据库函数库的接口。接下来介绍实现。

16.4 实现概述

大多数数据库访问的函数库使用两个文件来存储信息：一个索引文件和一个数据文件。索引文件包括索引值（关键字）和一个指向数据文件中对应数据记录的指针。有许多技术可用来组织索引文件以提高按关键字查询的速度和效率，散列表和 B+树是两种常用的技术。我们采用固定大小的散列表来组织索引文件结构，并采用链表法解决散列冲突。在介绍 `db_open` 时，曾提到将建立两个文件：一个以 `.idx` 为后缀的索引文件和一个以 `.dat` 为后缀的数据文件。

我们将关键字和索引以 NULL 结尾的字符串形式存储——它们不能包含任一的二进制数据。有些数据库系统用二进制的形式存储数值数据（如用 1、2 或 4 个字节存储一个整数）以节省空间，这样一来使函数复杂化，也使数据库文件在不同的平台间移植比较困难。比方说，网络上有两个系统使用不同的二进制格式存储整数，如果想要这两个系统都能够访问数据库就必须解决这个问题（今天不同体系结构的系统共享文件已经很常见了）。按照字符串形式存储所有的记录，包括关键字和数据，能使这一切变得简单。这确实会需要更多的磁盘空间，但随着磁盘技术的发展，这渐渐不再构成问题。

`db_store` 要求对每个关键字，最多只有一个对应的记录。有些数据库系统允许多条记录使用同样的关键字，并提供方法访问与一个关键字相关的所有记录。另外，我们只有一个索引文件，这意味着每个数据记录只能有一个关键字。有些数据库允许一条记录拥有多个关键字，并且对每一个关键字使用一个索引文件。当加入或删除一条记录时，要对所有的索引文件进行相应的修改。（一个有多个索引的例子是雇员库文件，可以将雇员号作为关键字，也可以将雇员的社会保险号作为关键字。由于一般雇员的姓名并不保证唯一，所以名字不能作为关键字。）

图 16-1 是数据库实现的基本结构。索引文件由三部分组成：空闲链表指针、散列表和索引记录。图 16-1 `ptr` 字段中实际存储的是以 ASCII 字符串形式记录的文件中的位移量。

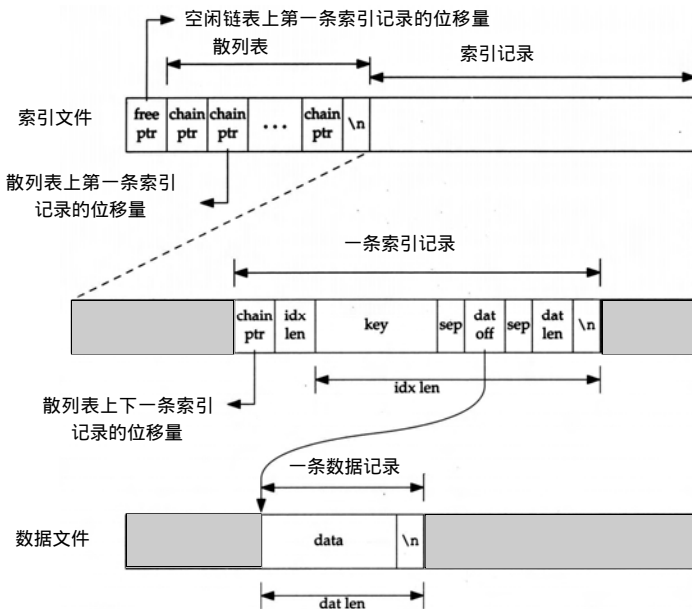


图 16-1 索引文件和数据文件结构

当给定一个关键字要在数据库中寻找一条记录时，`db_fetch`根据关键字计算散列值，由此散列值可确定一条散列链（`chain ptr`（链表指针）字段可以为0，表示一条空的散列链）。沿着这条散列链，我们可以找到所有有同样散列值的索引记录。当遇到一个索引记录的 `chain ptr` 字段为0时，表示到达了此散列链的末尾。

下面来看一个实际的数据库文件。程序16-1建立了一个新的数据库，并且加入了三条记录。由于所有的字段都以ASCII字符串的形式存储在数据库中，所以可以用任何标准的UNIX工具来查看索引文件和数据文件。

```
$ ls -l db4.*
-rw-r--r-- 1 stevens28 Oct 30 06:42 db4.dat
-rw-r--r-- 1 stevens28 Oct 30 06:42 db4.idx
$ cat db4.idx
 0 53 35 0
 0 10Alpha:0:6
 0 10beta:6:14
17 11gamma:20:8
$ cat db4.dat
data1
Data for beta
record3
```

程序16-1 建立一个数据库并向其写三条记录

```
#include "db.h"

int
main(void)
{
    DB *db;

    if ( (db = db_open("db4", O_RDWR | O_CREAT | O_TRUNC,
                      FILE_MODE)) == NULL)
        err_sys("db_open error");

    if (db_store(db, "Alpha", "data1", DB_INSERT) != 0)
        err_quit("db_store error for alpha");
    if (db_store(db, "beta", "Data for beta", DB_INSERT) != 0)
        err_quit("db_store error for beta");
    if (db_store(db, "gamma", "record3", DB_INSERT) != 0)
        err_quit("db_store error for gamma");

    db_close(db);
    exit(0);
}
```

为了使这个例子简单，将每个 `ptr` 字段的大小定为4个ASCII字符，将散列表的大小（散列表的条数）定为3。由于每一个 `ptr` 记录的是一个文件位移量，所以4个ASCII字符限制了一个索引文件或数据文件的大小最多只能为10 000字节。16.8节做性能方面的测试时，将 `ptr` 字段的大小设为6（这样文件大小可以达到1 000 000字节），将散列表的大小设为100。

索引文件的第一行为

```
0 53 35 0
```

分别为空闲链表指针（0表示空闲链表为空），和三个散列表表的指针：53、35和0。下一行

```
0 10Alpha:0:6
```

显示了一条索引记录的结构。第一个4字符的字段（0）表示这一条记录是此散列链的最后一条。

下一个4字符的字段(10)为`idx len`,(索引记录长度)表示此索引记录剩下部分的长度。通过两个read操作来读取一条索引记录:第一个read读取这两个固定长度的字段(`chain ptr`和`idx len`),然后再根据`idx len`来读取后面的不定长部分。剩下的三个字段为:`key`(关键字)、`dat off`(数据记录的位移量)和`dat len`(数据记录的长度)。这三个字段用`sep`(分隔符)隔开,在这里使用的是分号。由于此三个字段都是不定长的,所以需要有一个专门的分隔符,而且这个分隔符不能出现在关键字中。最后用一个`\n`(回车符)结束这一条索引记录。由于在`idx len`中已经有了记录的长度,所以这个回车符并不是必须的,加上回车符是为了把各条索引记录分开,这样就可以用标准的UNIX工具如`cat`和`more`来查看索引文件。`key`是将记录加入数据库时选择的关键字。数据记录在数据文件中的位移为0,长度为6。从数据文件中可看到数据记录确实从0开始,长度为6个字节。(与索引文件一样,这里自动在每条数据记录的后面加上一个回车符,以便于使用UNIX工具。在调用`db_fetch`时,此回车符不作为数据返回。)

如果在这个例子中跟踪三个散列链,可以看到第一条散列链上的第一条记录的位移量是53(`gamma`)。这条链上下一条记录的位移量为17(`Alpha`),并且是这条链上的最后一条记录。第二条散列链上的第一条记录的位移量是35(`beta`),且是此链上最后一条记录。第三条散列链为空。

请注意索引文件中索引记录的顺序和数据文件中对应数据记录的顺序与程序16-1中调用`db_store`的顺序一样。由于在调用`db_open`时使用了`O_TRUNC`标志,索引文件和数据文件都被截断,整个数据库相当于从新初始化。在这种情形下,`db_store`将新的索引记录和数据记录添加到对应的文件末尾。后面将看到`db_store`也可以重复使用这两个文件中因删除记录而生成的空间。

在这里使用固定大小的散列表作为索引是一个妥协。当每个散列链均不太长时,这个方法能保证快速地查找。我们的目的是能够快速地查找任一的关键字,同时又不使用太复杂的数据结构,如B树或动态可扩充散列表。动态可扩充散列表的优点是能保证仅用两次磁盘操作就能找到数据记录(详见Selter和Yigit[1991])。B树能够用关键字的顺序来遍历数据库(采用散列表的`db_nextrec`函数就做不到这一点)。

16.5 集中式或非集中式

当有多个进程访问数据库时,有两种方法可实现库函数:

(1) 集中式 由一个进程作为数据库管理者,所有的数据库访问工作由此进程完成。库函数通过IPC机制与此中心进程进行联系。

(2) 非集中式 每个库函数独立申请并发控制(加锁),然后调用它自己的I/O函数。

使用这两种技术的数据库都有。UNIX系统中的潮流是使用非集中式方法。如果有适当的加锁函数,因为避免使用了IPC,那么非集中式方法一般要快一些。图16-2描绘了集中式方法的操作。

图中特意表示出IPC像绝大多数UNIX的消息传送一样需要经过操作系统内核(14.9节的共享存储不需要这种经过内核的拷贝)。我们看到,在集中式方式下,中心控制进程将记录读出,然后通过IPC机制将数据送给请求进程。注意到中心控制进程是唯一的通过I/O操作存取数据库文件的进程。

集中式的优点是能够根据需要来对操作模式进行控制。例如,可以通过中心进程给不同的进程赋予不同的优先级,而非集中式方法则很难做到。在这种情况下只能依赖于操作内核的磁盘I/O调度策略和加锁策略(如当三个进程同时等待一个锁开锁时,哪个进程下一个得到锁)。

图16-3描绘了非集中式方法，本章的实现就是采用这种方法。使用库函数访问数据库的用户进程是平等的，它们通过使用记录锁机制来实现并发控制。

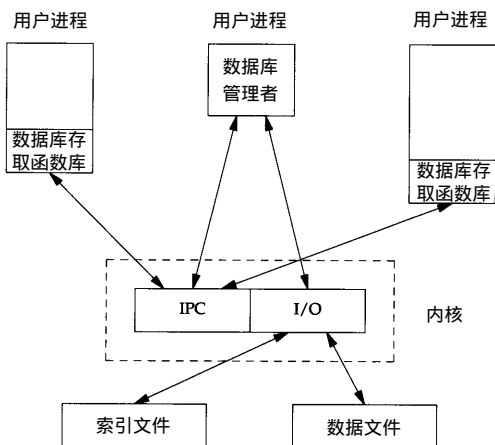


图16-2 集中式数据库访问

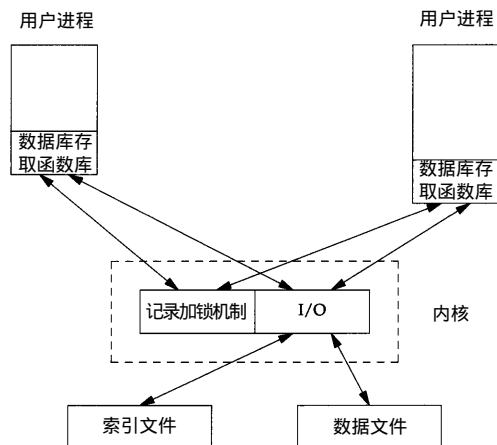


图16-3 非集中式数据库访问

16.6 并发

由于很多系统的实现都采用两个文件（一个索引文件和一个数据文件）的方法，所以我们也可以使用这种方法，这要求我们能够控制对两个文件的加锁。有很多方法可用来对两个文件进行加锁。

16.6.1 粗锁

最简单的加锁方法是将这两个文件中的一个作为锁，并要求调用者在对数据库进行操作前必须获得这个锁。我们将这称为粗锁（coarse locking）。比方说，可以认为一个进程对索引文件的0字节加了读锁后，才能获得读整个数据库的权力。一个进程对索引文件的0字节加了写锁后，才获得修改整个数据库的权力。可以使用UNIX的记录锁机制来控制每次可以有多个读进程，而只能由一个写进程（见表12-2）。db_fetch和db_nextrec函数将获得读锁，而db_delete、db_store以及db_open则获得写锁。（db_open获得写锁的原因是如果要创建新文件的话，要在索引文件前端建立空闲区链表以及散列链表。）

粗锁的问题是它限制了最大程度的并发。用粗锁时，当一个进程向一条散列链中加一条记录时，其他进程无法访问另一条散列链上的记录。

16.6.2 细锁

下面用称为细锁（fine locking）的方法来加强粗锁以提高并发度。我们要求一个读进程或写进程在操作一条记录前必须先获得此记录所在散列链的读锁或写锁。允许对一条散列链同时可以有多个读进程，而只能有一个写进程。另外，一个写进程在操作空闲区链表（如db_delete或db_store）前，必须获得空闲区链表的写锁。最后，当db_store向索引文件或数据文件加一条新记录时，必须获得对应文件相应区域的写锁。

我们期望细锁能比粗锁提供更高的并发度。16.8节将给出一些实际的比较测试结果。16.7

节给出了采用细锁的实现，并详细讨论了锁的实现（粗锁是实现的简化）。

在源文件中，直接调用了read，readv，write和writev。没有使用标准I/O函数库。虽然使用标准I/O函数库也可以使用记录锁，但是需要非常复杂的缓存管理。我们不希望例如 fgets返回的数据是10分钟之前读入标准I/O缓存的，而被另一个进程在5分钟之前修改了。

我们对并发讨论依据的是数据库函数库的简单的需求。商业系统一般有更多的需要。关于并发更多的细节可以参见Data[1982]的第3章。

16.7 源码

我们从程序16-2中的头文件db.h开始。所有函数以及调用此函数库的用户进程都包含这一头文件。

程序16-2 db.h头文件

```
#include <sys/types.h>
#include <sys/stat.h> /* open() & db_open() mode */
#include <fcntl.h> /* open() & db_open() flags */
#include <stddef.h> /* NULL */
#include "ourhdr.h"

/* flags for db_store() */
#define DB_INSERT 1 /* insert new record only */
#define DB_REPLACE 2 /* replace existing record */

/* magic numbers */
#define IDXLEN_SZ 4 /* #ascii chars for length of index record */
#define IDXLEN_MIN 6 /* key, sep, start, sep, length, newline */
#define IDXLEN_MAX 1024 /* arbitrary */
#define SEP ':' /* separator character in index record */
#define DATLEN_MIN 2 /* data byte, newline */
#define DATLEN_MAX 1024 /* arbitrary */

/* following definitions are for hash chains and free list chain
in index file */
#define PTR_SZ 6 /* size of ptr field in hash chain */
#define PTR_MAX 999999 /* max offset (file size) = 10**PTR_SZ - 1 */
#define NHASH_DEF 137 /* default hash table size */
#define FREE_OFF 0 /* offset of ptr to free list in index file */
#define HASH_OFF PTR_SZ /* offset of hash table in index file */

typedef struct { /* our internal structure */
    int idxfd; /* fd for index file */
    int datfd; /* fd for data file */
    int oflag; /* flags for open()/db_open(): O_*** */

    char *idxbuf; /* malloc'ed buffer for index record */
    char *datbuf; /* malloc'ed buffer for data record */
    char *name; /* name db was opened under */
    off_t idxoff; /* offset in index file of index record */
                /* actual key is at (idxoff + PTR_SZ + IDXLEN_SZ) */
    size_t idxlen; /* length of index record */
                /* excludes IDXLEN_SZ bytes at front of index record */
                /* includes newline at end of index record */
    off_t datoff; /* offset in data file of data record */
    size_t datlen; /* length of data record */
                /* includes newline at end */
    off_t ptrval; /* contents of chain ptr in index record */
    off_t ptroff; /* offset of chain ptr that points to this index record */
    off_t chainoff; /* offset of hash chain for this index record */
    off_t hashoff; /* offset in index file of hash table */
}
```



```

int    nhash; /* current hash table size */
long   cnt_delok; /* delete OK */
long   cnt_delerr; /* delete error */
long   cnt_fetchok; /* fetch OK */
long   cnt_fetcherr; /* fetch error */
long   cnt_nextrec; /* nextrec */
long   cnt_stor1; /* store: DB_INSERT, no empty, appended */
long   cnt_stor2; /* store: DB_INSERT, found empty, reused */
long   cnt_stor3; /* store: DB_REPLACE, diff data len, appended */
long   cnt_stor4; /* store: DB_REPLACE, same data len, overwrote */
long   cnt_storerr; /* store error */
} DB;

typedef unsigned long   hash_t; /* hash values */

/* user-callable functions */
DB      *db_open(const char *, int, int);
void     db_close(DB *);
char     *db_fetch(DB *, const char *);
int      db_store(DB *, const char *, const char *, int);
int      db_delete(DB *, const char *);
void     db_rewind(DB *);
char     *db_nextrec(DB *, char *);
void     db_stats(DB *);

/* internal functions */
DB      *_db_alloc(int);
int      _db_checkfree(DB *);
int      _db_dodelete(DB *);
int      _db_emptykey(char *);
int      _db_find(DB *, const char *, int);
int      _db_findfree(DB *, int, int);
int      _db_free(DB *);
hash_t   _db_hash(DB *, const char *);
char     *_db_nextkey(DB *);
char     *_db_readdat(DB *);
off_t    _db_readidx(DB *, off_t);

off_t    _db_readptr(DB *, off_t);
void     _db_writedat(DB *, const char *, off_t, int);
void     _db_writeidx(DB *, const char *, off_t, int, off_t);
void     _db_writeptr(DB *, off_t, off_t);

```

该程序定义了实现的基本限制。如果要支持更大的数据库的话，这些限制也可以修改。其中一些定义为常数的值也可定义为变量，只是会使实现复杂一些。例如，设定散列表的大小为137，一个也许更好的方法是让db_open的调用者根据数据库的大小通过参数来设定这个值，然后将这个值存在索引文件的最前面。

在DB结构中记录一个打开的数据库的所有信息。db_open函数返回一个DB结构的指针，这个指针被用于其他所有函数。

选择用db_开头来命名用户可调用的库函数，用_db_开头来命名内部函数。

程序16-3中定义了函数db_open。它打开索引文件和数据文件，必要的话初始化索引文件。通过调用_db_alloc来为DB结构分配空间，并初始化此结构。

程序16-3 db_open函数

```

#include    "db.h"

/* Open or create a database.  Same arguments as open(). */
DB *

```

```

db_open(const char *pathname, int oflag, int mode)
{
    DB          *db;
    int         i, len;
    char        asciiptr[PTR_SZ + 1],
               hash[(NHASH_DEF + 1) * PTR_SZ + 2];
               /* +2 for newline and null */
    struct stat statbuff;

    /* Allocate a DB structure, and the buffers it needs */
    len = strlen(pathname);
    if ( (db = _db_alloc(len)) == NULL)
        err_dump("_db_alloc error for DB");

    db->oflag = oflag;      /* save a copy of the open flags */

    /* Open index file */
    strcpy(db->name, pathname);
    strcat(db->name, ".idx");
    if ( (db->idxfd = open(db->name, oflag, mode)) < 0) {
        _db_free(db);
        return(NULL);
    }

    /* Open data file */
    strcpy(db->name + len, ".dat");
    if ( (db->datfd = open(db->name, oflag, mode)) < 0) {
        _db_free(db);
        return(NULL);
    }

    /* If the database was created, we have to initialize it */
    if ((oflag & (O_CREAT | O_TRUNC)) == (O_CREAT | O_TRUNC)) {
        /* Write lock the entire file so that we can stat
           the file, check its size, and initialize it,
           as an atomic operation. */
        if (writew_lock(db->idxfd, 0, SEEK_SET, 0) < 0)
            err_dump("writew_lock error");

        if (fstat(db->idxfd, &statbuff) < 0)
            err_sys("fstat error");
        if (statbuff.st_size == 0) {
            /* We have to build a list of (NHASH_DEF + 1) chain
               ptrs with a value of 0. The +1 is for the free
               list pointer that precedes the hash table. */
            sprintf(asciiptr, "%*d", PTR_SZ, 0);
            hash[0] = 0;
            for (i = 0; i < (NHASH_DEF + 1); i++)
                strcat(hash, asciiptr);
            strcat(hash, "\n");

            i = strlen(hash);
            if (write(db->idxfd, hash, i) != i)
                err_dump("write error initializing index file");
        }
        if (un_lock(db->idxfd, 0, SEEK_SET, 0) < 0)
            err_dump("un_lock error");
    }

    db->nhash = NHASH_DEF; /* hash table size */
    db->hashoff = HASH_OFF; /* offset in index file of hash table */
                          /* free list ptr always at FREE_OFF */
    db_rewind(db);

    return(db);
}

```

如果数据库正被建立，则必须加锁。考虑两个进程试图同时建立同一个数据库的情况。第一个进程运行到调用 `fstat`，并且在 `fstat` 返回后被内核切换。这时第二个进程调用 `db_open`，发现索引文件的长度为 0，并初始化空闲链表和散列链表。第二个进程继续运行，向数据库中添加了一条记录。这时第二个进程被阻塞，第一个进程继续运行，并发现索引文件的大小为 0（因为第一个进程是在 `fstat` 返回后才被切换），所以第一个进程重新初始化空闲链表和散列链表，第二个进程写入的记录就被抹去了。要避免发生这种情况的方法是进行加锁，可以使用 12.3 节中的 `readw_lock`，`writew_lock` 和 `un_lock` 这三个函数。

`db_open` 调用程序 16-4 中定义的函数 `_db_alloc` 来为 DB 结构分配空间，包括一个索引缓存和一个数据缓存。

程序 16-4 `_db_alloc` 函数

```

#include    "db.h"

/* Allocate & initialize a DB structure, and all the buffers it needs */
DB *
_db_alloc(int namelen)
{
    DB      *db;

    /* Use calloc, to init structure to zero */
    if ( (db = calloc(1, sizeof(DB))) == NULL)
        err_dump("calloc error for DB");

    db->idxfd = db->datfd = -1;          /* descriptors */

    /* Allocate room for the name.
       +5 for ".idx" or ".dat" plus null at end. */
    if ( (db->name = malloc(namelen + 5)) == NULL)
        err_dump("malloc error for name");

    /* Allocate an index buffer and a data buffer.
       +2 for newline and null at end. */
    if ( (db->idxbuf = malloc(IDXLEN_MAX + 2)) == NULL)
        err_dump("malloc error for index buffer");
    if ( (db->datbuf = malloc(DATLEN_MAX + 2)) == NULL)
        err_dump("malloc error for data buffer");

    return (db);
}

```

索引缓存和数据缓存的大小在 `db.h` 中定义。数据库函数库可以通过让这些缓存按需要动态扩张来得到加强。其方法可以是记录这两个缓存的大小，然后在需要更大的缓存时调用 `realloc`。

在函数 `_db_free`（见程序 16-5）中，这些缓存将被释放，同时打开的文件被关闭。`db_open` 在打开索引文件和数据文件时如果遇到错误，则调用 `_db_free` 释放资源。`db_close`（见程序 16-6）也调用 `_db_free`。

函数 `db_fetch`（见程序 16-7）根据给定的关键字来读取一条记录。它调用 `_db_find` 在数据库中查找一条索引记录，如果找到，再调用 `_db_readdat` 来读取对应的数据记录。

程序 16-5 `_db_free` 函数

```

#include    "db.h"

/* Free up a DB structure, and all the malloc'ed buffers it

```

```

* may point to. Also close the file descriptors if still open. */
int
_db_free(DB *db)
{
    if (db->idxfd >= 0 && close(db->idxfd) < 0)
        err_dump("index close error");
    if (db->datfd >= 0 && close(db->datfd) < 0)
        err_dump("data close error");
    db->idxfd = db->datfd = -1;

    if (db->idxbuf != NULL)
        free(db->idxbuf);
    if (db->datbuf != NULL)
        free(db->datbuf);
    if (db->name != NULL)
        free(db->name);
    free(db);
    return(0);
}

```

程序16-6 db_close函数

```

#include    "db.h"

void
db_close(DB *db)
{
    _db_free(db);    /* closes fds, free buffers & struct */
}

```

程序16-7 db_fetch函数

```

#include    "db.h"

/* Fetch a specified record.
 * We return a pointer to the null-terminated data. */
char *
db_fetch(DB *db, const char *key)
{
    char    *ptr;

    if (_db_find(db, key, 0) < 0) {
        ptr = NULL;    /* error, record not found */
        db->cnt_fetcherr++;
    } else {
        ptr = _db_readdat(db); /* return pointer to data */
        db->cnt_fetchok++;
    }

    /* Unlock the hash chain that _db_find() locked */
    if (un_lock(db->idxfd, db->chainoff, SEEK_SET, 1) < 0)
        err_dump("un_lock error");
    return(ptr);
}

```

函数_db_find(见程序16-8)通过遍历散列链来查找记录。db_fetch, db_delete和db_store这几个需要根据关键字查找记录的函数都调用它。

程序16-8 _db_find函数

```

#include    "db.h"

/* Find the specified record.
 * Called by db_delete(), db_fetch(), and db_store(). */

int
_db_find(DB *db, const char *key, int writelock)
{
    off_t    offset, nextoffset;

    /* Calculate hash value for this key, then calculate byte
     * offset of corresponding chain ptr in hash table.
     * This is where our search starts. */

    /* calc offset in hash table for this key */
    db->chainoff = (_db_hash(db, key) * PTR_SZ) + db->hashoff;
    db->ptroff = db->chainoff;

    /* Here's where we lock this hash chain. It's the
     * caller's responsibility to unlock it when done.
     * Note we lock and unlock only the first byte. */
    if (writelock) {
        if (writew_lock(db->idxfd, db->chainoff, SEEK_SET, 1) < 0)
            err_dump("writew_lock error");
    } else {
        if (readw_lock(db->idxfd, db->chainoff, SEEK_SET, 1) < 0)
            err_dump("readw_lock error");
    }

    /* Get the offset in the index file of first record
     * on the hash chain (can be 0) */
    offset = _db_readptr(db, db->ptroff);

    while (offset != 0) {
        nextoffset = _db_readidx(db, offset);
        if (strcmp(db->idxbuf, key) == 0)
            break;        /* found a match */

        db->ptroff = offset;    /* offset of this (unequal) record */
        offset = nextoffset;    /* next one to compare */
    }

    if (offset == 0)
        return(-1);        /* error, record not found */

    /* We have a match. We're guaranteed that db->ptroff contains
     * the offset of the chain ptr that points to this matching
     * index record. _db_dodelete() uses this fact. (The chain
     * ptr that points to this matching record could be in an
     * index record or in the hash table.) */
    return(0);
}

```

_db_find的最后一个参数指明需要加什么样的锁，0表示读锁，1表示写锁。我们知道，db_fetch需要加读锁，而db_delete和db_store需要加写锁。_db_find在获得需要的锁之前将等待。

_db_find中的while循环遍历散列链中的索引记录，并比较关键字。函数_db_readidx用于读取每条索引记录。

请注意阅读_db_find中的最后一条注释。当沿着散列链进行遍历时，必须始终跟踪当前索引记录的前一条索引记录，其中有一个指针指向当前记录。这一点在删除一条记录时很有用，因为必须修改当前索引记录的前一条记录的链指针。

先来看看_db_find调用的一些比较简单的函数。_db_hash（见程序16-9）根据给定的关键字计算散列值。它将关键字中的每一个ASCII字符乘以这个字符在字符串中以1开始的索引号，将这些结果加起来，除以散列表的大小，将余数作为这个关键字的散列值。

程序16-9 _db_hash函数

```
#include    "db.h"

/* Calculate the hash value for a key. */

hash_t
_db_hash(DB *db, const char *key)
{
    hash_t    hval;
    const char *ptr;
    char      c;
    int       i;

    hval = 0;
    for (ptr = key, i = 1; c = *ptr++; i++)
        hval += c * i;    /* ascii char times its 1-based index */

    return(hval % db->nhash);
}
```

_db_find调用的下一个函数是_db_readptr（见程序16-10）。这个函数能够读取以下三种不同的链表指针中的任意一种：（1）索引文件最开始的空闲链表指针，（2）散列表中指向散列链的第一条索引记录的指针，（3）每条索引记录头的指向下一条记录的指针（这里的索引记录既可以处于一条散列表链中，也可以处于空闲链表中）。这个函数的调用者应做好必要的加锁，此函数不进行任何加锁。

程序16-10 _db_readptr函数

```
#include    "db.h"

/* Read a chain ptr field from anywhere in the index file:
 * the free list pointer, a hash table chain ptr, or an
 * index record chain ptr. */

off_t
_db_readptr(DB *db, off_t offset)
{
    char    asciiptr[PTR_SZ + 1];

    if (lseek(db->idxfd, offset, SEEK_SET) == -1)
        err_dump("lseek error to ptr field");
    if (read(db->idxfd, asciiptr, PTR_SZ) != PTR_SZ)
        err_dump("read error of ptr field");

    asciiptr[PTR_SZ] = 0;    /* null terminate */
    return(atol(asciiptr));
}
```

_db_find中的while循环通过调用_db_readidx来读取各条索引记录。_db_readidx（见程序16-11）是一个较长的函数，这个函数读取索引记录，并将索引记录的信息存储到适当的字段中。

程序16-11 _db_readidx函数

```
#include    "db.h"
#include    <sys/uio.h>    /* struct iovec */
```

```

/* Read the next index record. We start at the specified offset in
 * the index file. We read the index record into db->idxbuf and
 * replace the separators with null bytes. If all is OK we set
 * db->datoff and db->datlen to the offset and length of the
 * corresponding data record in the data file. */

off_t
_db_readidx(DB *db, off_t offset)
{
    int            i;
    char          *ptr1, *ptr2;
    char          *asciiptr[PTR_SZ + 1], asciilen[IDXLEN_SZ + 1];
    struct iovec   iov[2];

    /* Position index file and record the offset. db_nextrec()
     * calls us with offset==0, meaning read from current offset.
     * We still need to call lseek() to record the current offset. */
    if ( (db->idxoff = lseek(db->idxfd, offset,
        offset == 0 ? SEEK_CUR : SEEK_SET)) == -1)
        err_dump("lseek error");

    /* Read the ascii chain ptr and the ascii length at
     * the front of the index record. This tells us the
     * remaining size of the index record. */
    iov[0].iov_base = asciiptr;
    iov[0].iov_len  = PTR_SZ;
    iov[1].iov_base = asciilen;
    iov[1].iov_len  = IDXLEN_SZ;
    if ( (i = readv(db->idxfd, &iov[0], 2)) != PTR_SZ + IDXLEN_SZ) {
        if (i == 0 && offset == 0)
            return(-1); /* EOF for db_nextrec() */
        err_dump("readv error of index record");
    }

    asciiptr[PTR_SZ] = 0; /* null terminate */
    db->ptrval = atol(asciiptr); /* offset of next key in chain */
    /* this is our return value; always >= 0 */
    asciilen[IDXLEN_SZ] = 0; /* null terminate */
    if ( (db->idxlen = atol(asciilen)) < IDXLEN_MIN ||
        db->idxlen > IDXLEN_MAX)
        err_dump("invalid length");

    /* Now read the actual index record. We read it into the key
     * buffer that we malloced when we opened the database. */
    if ( (i = read(db->idxfd, db->idxbuf, db->idxlen)) != db->idxlen)
        err_dump("read error of index record");
    if (db->idxbuf[db->idxlen-1] != '\n')
        err_dump("missing newline"); /* sanity checks */
    db->idxbuf[db->idxlen-1] = 0; /* replace newline with null */

    /* Find the separators in the index record */
    if ( (ptr1 = strchr(db->idxbuf, SEP)) == NULL)
        err_dump("missing first separator");
    *ptr1++ = 0; /* replace SEP with null */

    if ( (ptr2 = strchr(ptr1, SEP)) == NULL)
        err_dump("missing second separator");
    *ptr2++ = 0; /* replace SEP with null */

    if (strchr(ptr2, SEP) != NULL)
        err_dump("too many separators");

    /* Get the starting offset and length of the data record */
    if ( (db->datoff = atol(ptr1)) < 0)
        err_dump("starting offset < 0");
    if ( (db->datlen = atol(ptr2)) <= 0 || db->datlen > DATLEN_MAX)
        err_dump("invalid length");
}

```

```

        return(db->ptrval);    /* return offset of next key in chain */
    }

```

我们调用 `readv` 来读取索引记录开始处的两个固定长度的字段：指向下一条索引记录的链表指针和索引记录剩下的不定长部分的长度。然后，索引记录剩下的部分被读入：关键字、数据记录的位移量和数据记录的长度。我们并不读数据记录，这由调用者自己完成。例如，在 `db_fetch` 中，在 `_db_find` 按关键字找到索引记录前是不去读取数据记录的。

下面回到 `db_fetch`。如果 `_db_find` 找到了索引记录，则调用 `_db_readdat` 来读取对应的数据记录。这是一个很简单的函数（见程序 16-12）。

程序16-12 `_de_readdat`函数

```

#include    "db.h"

/* Read the current data record into the data buffer.
 * Return a pointer to the null-terminated data buffer. */

char *
_db_readdat(DB *db)
{
    if (lseek(db->datfd, db->datoff, SEEK_SET) == -1)
        err_dump("lseek error");

    if (read(db->datfd, db->datbuf, db->datlen) != db->datlen)
        err_dump("read error");
    if (db->datbuf[db->datlen - 1] != '\n')    /* sanity check */
        err_dump("missing newline");
    db->datbuf[db->datlen - 1] = 0;    /* replace newline with null */

    return(db->datbuf);    /* return pointer to data record */
}

```

我们从 `db_fetch` 开始，现在已经读取了索引记录和对应的数据记录。注意到，只在 `_db_find` 中加了一个读锁。由于对这条散列链加了读锁，所以其他进程在这段时间内不可能对这条散列链进行修改。

下面来看函数 `db_delete`（见程序 16-13）。它开始时和 `db_fetch` 一样，调用 `_db_find` 来查找记录，只是这里传递给 `_db_find` 的最后一个参数为 1，表示要对这一条散列链加写锁。

`db_delete` 调用 `_db_dodelete`（见程序 16-14）来完成剩下的工作（在后面将看到 `db_store` 也调用 `_db_dodelete`）。`_db_dodelete` 的大部分工作是修正空闲链表以及与关键字对应的散列链。

当一条记录被删除后，将其关键字和数据记录设为空。本章后面将提到的函数 `db_nextrec` 要用到这一点。

`_db_dodelete` 对空闲链表加写锁，这样能防止两个进程同时删除不同链表上的记录时产生相互影响，因为我们将被删除的记录移到空闲链表上，这将改变空闲链表，而一次只能有一个进程能这样做。

程序16-13 `db_delete`函数

```

#include    "db.h"

/* Delete the specified record */

int
db_delete(DB *db, const char *key)

```



```

{
    int    rc;

    if (_db_find(db, key, 1) == 0) {
        rc = _db_dodelete(db); /* record found */
        db->cnt_delok++;
    } else {
        rc = -1; /* not found */
        db->cnt_delerr++;
    }

    if (un_lock(db->idxfd, db->chainoff, SEEK_SET, 1) < 0)
        err_dump("un_lock error");
    return(rc);
}

```

程序16-14 _db_dodelete函数

```

#include    "db.h"

/* Delete the current record specified by the DB structure.
 * This function is called by db_delete() and db_store(),
 * after the record has been located by _db_find(). */

int
_db_dodelete(DB *db)
{
    int    i;
    char   *ptr;
    off_t  freeptr, saveptr;

    /* Set data buffer to all blanks */
    for (ptr = db->datbuf, i = 0; i < db->datlen - 1; i++)
        *ptr++ = ' ';
    *ptr = 0; /* null terminate for _db_writedat() */

    /* Set key to blanks */
    ptr = db->idxbuf;
    while (*ptr)
        *ptr++ = ' ';

    /* We have to lock the free list */
    if (writew_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
        err_dump("writew_lock error");

    /* Write the data record with all blanks */
    _db_writedat(db, db->datbuf, db->datoff, SEEK_SET);

    /* Read the free list pointer. Its value becomes the
     chain ptr field of the deleted index record. This means
     the deleted record becomes the head of the free list. */
    freeptr = _db_readptr(db, FREE_OFF);

    /* Save the contents of index record chain ptr,
     before it's rewritten by _db_writeidx(). */
    saveptr = db->ptrval;

    /* Rewrite the index record. This also rewrites the length
     and the data offset, and the data length,
     none of which has changed, but that's OK. */
    _db_writeidx(db, db->idxbuf, db->idxoff, SEEK_SET, freeptr);

    /* Write the new free list pointer */
    _db_writeptr(db, FREE_OFF, db->idxoff);

    /* Rewrite the chain ptr that pointed to this record

```

```

        being deleted. Recall that _db_find() sets db->ptroff
        to point to this chain ptr. We set this chain ptr
        to the contents of the deleted record's chain ptr,
        saveptr, which can be either zero or nonzero. */
    _db_writeptr(db, db->ptroff, saveptr);

    if (un_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
        err_dump("un_lock error");

    return(0);
}

```

`_db_dodelete`通过调用函数`_db_writedat`（见程序16-15）清空数据记录。这时`_db_writedat`并不对数据文件加写锁，这是因为`db_delete`对这条记录索引的散列链加了写锁，这已经保证不会有其他进程能够读写这条记录。本章后面讲述`db_store`时，将看到在另一种情况下，`_db_writedat`将数据添加到文件的末尾，并加锁。

`_db_writedat`调用`writew`来写数据记录及回车符。不能够假设调用者传递进来的字符缓存有空间让我们在其后面再添加一个回车符。回忆12.7节，在那里曾讨论过一个`writew`要比两个`write`快。

接着`_db_dodelete`修改索引记录。让这条记录的链表指针指向空闲链表的第一条记录（如果空闲链表为空，则这个链表指针置为0），空闲链表指针也被修改，指向当前删除的这条记录，这样就将这条删除的记录移到了空闲链表首。空闲链表实际上很象一个先进先出的堆栈。

程序16-15 `_db_writedat`函数

```

#include    "db.h"
#include    <sys/uio.h>    /* struct iovec */

/* Write a data record. Called by _db_dodelete() (to write
   the record with blanks) and db_store(). */

void
_db_writedat(DB *db, const char *data, off_t offset, int whence)
{
    struct iovec    iov[2];
    static char     newline = '\n';

    /* If we're appending, we have to lock before doing the lseek()
       and write() to make the two an atomic operation. If we're
       overwriting an existing record, we don't have to lock. */
    if (whence == SEEK_END)    /* we're appending, lock entire file */
        if (writew_lock(db->datfd, 0, SEEK_SET, 0) < 0)
            err_dump("writew_lock error");

    if ( (db->datoff = lseek(db->datfd, offset, whence)) == -1)
        err_dump("lseek error");
    db->datlen = strlen(data) + 1;    /* datlen includes newline */

    iov[0].iov_base = (char *) data;
    iov[0].iov_len = db->datlen - 1;
    iov[1].iov_base = &newline;
    iov[1].iov_len = 1;
    if (writew(db->datfd, &iov[0], 2) != db->datlen)
        err_dump("writew error of data record");

    if (whence == SEEK_END)
        if (un_lock(db->datfd, 0, SEEK_SET, 0) < 0)
            err_dump("un_lock error");
}

```

我们并没有为索引文件和数据文件的空闲记录设置各自的空闲链表。这是因为当一条记录被删除时，对应的索引记录被加入了空闲链表，而这条索引记录中有指向数据文件中数据记录的指针。有其他很多更好的处理记录删除的方法，只是它们会更复杂一些。

程序16-16列出了函数 `_db_writeidx`，`_db_dodelete`调用此函数来写一条索引记录。和 `_db_writedat`一样，这一函数也只有添加新索引记录时才需要加锁。`_db_dodelete`调用此函数是为了重写一条索引记录，在这种情况下调用者已经在散列链上加了写锁，所以不再需要加另外的锁。

程序16-16 `_db_writeidx`函数

```
#include "db.h"
#include <sys/uio.h> /* struct iovec */

/* Write an index record.
 * _db_writedat() is called before this function, to set the fields
 * datoff and datlen in the DB structure, which we need to write
 * the index record. */

void
_db_writeidx(DB *db, const char *key,
             off_t offset, int whence, off_t ptrval)
{
    struct iovec    iov[2];
    char           asciiptrlen[PTR_SZ + IDXLEN_SZ + 1];
    int            len;

    if ( (db->ptrval = ptrval) < 0 || ptrval > PTR_MAX)
        err_quit("invalid ptr: %d", ptrval);

    sprintf(db->idxbuf, "%s%c%d%c%d\n",
            key, SEP, db->datoff, SEP, db->datlen);
    if ( (len = strlen(db->idxbuf)) < IDXLEN_MIN || len > IDXLEN_MAX)
        err_dump("invalid length");
    sprintf(asciiptrlen, "%*d*d", PTR_SZ, ptrval, IDXLEN_SZ, len);

    /* If we're appending, we have to lock before doing the lseek()
     * and write() to make the two an atomic operation. If we're
     * overwriting an existing record, we don't have to lock. */
    if (whence == SEEK_END) /* we're appending */
        if (writew_lock(db->idxfd, ((db->nhash+1)*PTR_SZ)+1,
                       SEEK_SET, 0) < 0)
            err_dump("writew_lock error");

    /* Position the index file and record the offset */
    if ( (db->idxoff = lseek(db->idxfd, offset, whence)) == -1)
        err_dump("lseek error");

    iov[0].iov_base = asciiptrlen;
    iov[0].iov_len  = PTR_SZ + IDXLEN_SZ;
    iov[1].iov_base = db->idxbuf;
    iov[1].iov_len  = len;
    if (writev(db->idxfd, &iov[0], 2) != PTR_SZ + IDXLEN_SZ + len)
        err_dump("writev error of index record");

    if (whence == SEEK_END)
        if (un_lock(db->idxfd, ((db->nhash+1)*PTR_SZ)+1, SEEK_SET, 0) < 0)
            err_dump("un_lock error");
}
```

`_db_dodelete`调用的最后一个函数是 `_db_writeptr`（见程序16-17）。它被调用两次——一次

用来写空闲链表指针，一次用来写散列链表指针（指向被删除索引记录的指针）。

程序16-17 _db_writeptr函数

```
#include    "db.h"

/* Write a chain ptr field somewhere in the index file:
 * the free list, the hash table, or in an index record. */

void
_db_writeptr(DB *db, off_t offset, off_t ptrval)
{
    char    asciiptr[PTR_SZ + 1];

    if (ptrval < 0 || ptrval > PTR_MAX)
        err_quit("invalid ptr: %d", ptrval);
    sprintf(asciiptr, "%*d", PTR_SZ, ptrval);

    if (lseek(db->idxfd, offset, SEEK_SET) == -1)
        err_dump("lseek error to ptr field");
    if (write(db->idxfd, asciiptr, PTR_SZ) != PTR_SZ)
        err_dump("write error of ptr field");
}
```

程序16-18中列出了最长的数据库函数 db_store。它从调用 _db_find 开始，以查看这个记录是否存在。如果记录存在且标志为 DB_REPLACE，或记录不存在且标志为 DB_INSERT，这些都是允许的。替换一条已存在的记录，指的是该记录的关键字一样，而数据很可能不一样。

注意到因为 db_store 可能会改变散列链，所以调用 _db_find 的最后一条参数说明要对散列链加写锁。

如果要加一条新记录到数据库中，则调用函数 _db_findfree（见程序16-19）在空闲链表中搜索一个有同样关键字大小和同样数据大小的已删除的记录。

_db_findfree 中的 while 循环遍历空闲链表以搜寻一个有对应关键字大小和数据大小的索引记录项。在这个简单的实现中，只有当一个已删除记录的关键字大小及数据大小与新加入的记录的关键字大小及数据大小一样时才重用已删除的记录的空间。其他更好的算法一般更复杂。

_db_findfree 需要对空闲链表加写锁以避免与其他使用空闲链表的进程互相影响。当一条记录从空闲链表上取下来后，就可以打开这个写锁。_db_dodelete 也要修改空闲链表。

程序16-18 db_store函数

```
#include    "db.h"

/* Store a record in the database.
 * Return 0 if OK, 1 if record exists and DB_INSERT specified,
 * -1 if record doesn't exist and DB_REPLACE specified. */

int
db_store(DB *db, const char *key, const char *data, int flag)
{
    int    rc, keylen, datlen;
    off_t    ptrval;

    keylen = strlen(key);
    datlen = strlen(data) + 1;    /* +1 for newline at end */
    if (datlen < DATLEN_MIN || datlen > DATLEN_MAX)
        err_dump("invalid data length");
```

```

/* _db_find() calculates which hash table this new record
   goes into (db->chainoff), regardless whether it already
   exists or not. The calls to _db_writeptr() below
   change the hash table entry for this chain to point to
   the new record. This means the new record is added to
   the front of the hash chain. */

if (_db_find(db, key, 1) < 0) {      /* record not found */
    if (flag & DB_REPLACE) {
        rc = -1;
        db->cnt_storerr++;
        goto doreturn;      /* error, record does not exist */
    }

    /* _db_find() locked the hash chain for us; read the
       chain ptr to the first index record on hash chain */
    ptrval = _db_readptr(db, db->chainoff);

    if (_db_findfree(db, keylen, datlen) < 0) {
        /* An empty record of the correct size was not found.
           We have to append the new record to the ends of
           the index and data files */
        _db_writedat(db, data, 0, SEEK_END);
        _db_writeidx(db, key, 0, SEEK_END, ptrval);
        /* db->idxoff was set by _db_writeidx(). The new
           record goes to the front of the hash chain. */
        _db_writeptr(db, db->chainoff, db->idxoff);
        db->cnt_stor1++;
    } else {
        /* We can reuse an empty record.
           _db_findfree() removed the record from the free
           list and set both db->datoff and db->idxoff. */
        _db_writedat(db, data, db->datoff, SEEK_SET);
        _db_writeidx(db, key, db->idxoff, SEEK_SET, ptrval);
        /* reused record goes to the front of the hash chain. */
        _db_writeptr(db, db->chainoff, db->idxoff);
        db->cnt_stor2++;
    }
} else {
    /* record found */
    if (flag & DB_INSERT) {
        rc = 1;
        db->cnt_storerr++;
        goto doreturn;      /* error, record already in db */
    }

    /* We are replacing an existing record. We know the new
       key equals the existing key, but we need to check if
       the data records are the same size. */
    if (datlen != db->datlen) {
        _db_dodelete(db); /* delete the existing record */

        /* Reread the chain ptr in the hash table
           (it may change with the deletion). */
        ptrval = _db_readptr(db, db->chainoff);

        /* append new index and data records to end of files */
        _db_writedat(db, data, 0, SEEK_END);
        _db_writeidx(db, key, 0, SEEK_END, ptrval);
        /* new record goes to the front of the hash chain. */
        _db_writeptr(db, db->chainoff, db->idxoff);
        db->cnt_stor3++;
    }
}

```

```

    } else {
        /* same size data, just replace data record */
        _db_writedat(db, data, db->datoff, SEEK_SET);
        db->cnt_stor4++;
    }
}
rc = 0; /* OK */
doreturn: /* unlock the hash chain that _db_find() locked */
if (_un_lock(db->idxfd, db->chainoff, SEEK_SET, 1) < 0)
    err_dump("un_lock error");
return(rc);
}

```

程序16-19 _db_findfree函数

```

#include "db.h"

/* Try to find a free index record and accompanying data record
 * of the correct sizes. We're only called by db_store(). */

int
_db_findfree(DB *db, int keylen, int datlen)
{
    int rc;
    off_t offset, nextoffset, saveoffset;

    /* Lock the free list */
    if (writew_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
        err_dump("writew_lock error");

    /* Read the free list pointer */
    saveoffset = FREE_OFF;
    offset = _db_readptr(db, saveoffset);

    while (offset != 0) {
        nextoffset = _db_readidx(db, offset);
        if (strlen(db->idxbuf) == keylen && db->datlen == datlen)
            break; /* found a match */

        saveoffset = offset;
        offset = nextoffset;
    }

    if (offset == 0)
        rc = -1; /* no match found */
    else {
        /* Found a free record with matching sizes.
         * The index record was read in by _db_readidx() above,
         * which sets db->ptrval. Also, saveoffset points to
         * the chain ptr that pointed to this empty record on
         * the free list. We set this chain ptr to db->ptrval,
         * which removes the empty record from the free list. */

        _db_writeptr(db, saveoffset, db->ptrval);
        rc = 0;

        /* Notice also that _db_readidx() set both db->idxoff
         * and db->datoff. This is used by the caller, db_store(),
         * to write the new index record and data record. */
    }

    /* Unlock the free list */
    if (_un_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
        err_dump("un_lock error");
    return(rc);
}

```

回到函数db_store，在调用_db_find后，有四种可能：

(1) 加入一条新的记录，而_db_findfree没有找到对应大小的空闲记录。这意味着要将这条新记录添加到索引文件和数据文件的末尾。通过调用_db_writeptr将新记录加到对应的散列链的链首。

(2) 加入一条新记录，且_db_findfree找到对应大小的空闲记录。这条空闲记录被_db_findfree从空闲链表上移下来，新的索引记录和数据记录被写入，然后通过调用_db_writeptr将新记录加到对应的散列链的链首。

(3) 要替换一条记录，而新数据记录的长度与已存在的记录的长度不一样。调用_db_dodelete将老记录删除，然后将新记录添加到索引文件和数据文件的末尾（也可以用其他方法，如可以再找一找是否有适当大小的已删除的记录项）。最后调用_db_writeptr将新记录加到对应的散列链的链首。

(4) 要替换一条记录，而新数据记录的长度与已存在的记录的长度恰好一样。这是最容易的情况，只需要重写记录即可。

在向文件的末尾添加索引记录或数据记录时，需要加锁（回忆在程序 12-6中遇到的相对文件尾加锁问题）。在第(1)和第(3)种情况中，db_store调用_db_writeidx和_db_writedat时，第3个参数为0，第4个参数为SEEK_END。这里，第4个参数作为一个标志用来告诉这两个函数，新的记录将被添加到文件的末尾。_db_writeidx用到的技术是对索引文件加写锁，加锁的范围从散列链的末尾到文件的末尾。这不会影响其他数据库的读用户和写用户（这些用户将对散列链加锁），但如果其他用户此时调用db_store来添加数据则会被锁住。_db_writedat使用的方法是对整个数据文件加写锁。同样这也不会影响其他数据库的读用户和写用户（它们甚至不对数据文件加锁），但如果其他用户此时调用db_store来向数据文件添加数据则会被锁住（见习题 16.3）。

最后两个函数是db_nextrec和db_rewind，这两个函数用来读取数据库的所有记录。通常在下列形式的循环中的使用这两个函数：

```
db_rewind(db) ;
while( (ptr = db_nextrec(db, key)) != NULL) {
    /* process record */
}
```

前面曾警告过，记录的返回没有一定的次序——它们并不按关键字的顺序。

函数db_rewind（见程序 16-20）将索引文件定位在第一条索引记录（紧跟在散列表后面）。

程序 16-20 db_rewind函数

```
#include "db.h"

/* Rewind the index file for db_nextrec().
 * Automatically called by db_open().
 * Must be called before first db_nextrec().
 */

void
db_rewind(DB *db)
{
    off_t offset;

    offset = (db->nhash + 1) * PTR_SZ; /* +1 for free list ptr */

    /* We're just setting the file offset for this process
     * to the start of the index records; no need to lock.
     */
}
```

```

+1 below for newline at end of hash table. */
if ( (db->idxoff = lseek(db->idxfd, offset+1, SEEK_SET)) == -1)
    err_dump("lseek error");
}

```

当db_rewind定位好索引文件后，db_nextrec一条一条按顺序读取索引记录。在程序 16-21 中可以看到，db_nextrec并不使用散列链表。由于db_nextrec除了读取散列链表中的记录外，还读取已删除的记录，所以必须检查记录是否是已被删除的（关键字为空），并忽略这些已删除的记录。

如果db_nextrec在循环中被调用时数据库正被修改，则db_nextrec返回的记录只是变化中的数据库在某一时刻的快照（snapshot）。db_nextrec总是返回一条“正确”的记录，也就是说它不会返回一条已删除的记录。但有可能一条记录刚被db_nextrec返回后就被删除。类似的，如果db_nextrec刚跳过一条已删除的记录，这条记录的空间就被一条新记录重用，除非用db_rewind并重新遍历一遍，否则看不到这条新的记录。如果通过db_nextrec获得一份数据库的准确的“冻结”的快照很重要，则应作到在这段时间内没有添加和删除。

下面来看db_nextrec使用的加锁。并不使用任何的散列链表，也不判断每条记录属于哪条散列链。所以有可能当db_nextrec读取一条记录时，其索引记录正在被删除。为了防止这种情况，db_nextrec对空闲链表加读锁，这样就可避免与_db_dodelete和_db_findfree相互影响。

程序16-21 db_nextrec函数

```

#include    "db.h"
/* Return the next sequential record.
 * We just step our way through the index file, ignoring deleted
 * records.  db_rewind() must be called before this function is
 * called the first time.
 */
char *
db_nextrec(DB *db, char *key)
{
    char    c, *ptr;

    /* We read lock the free list so that we don't read
     * a record in the middle of its being deleted. */
    if (readw_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
        err_dump("readw_lock error");

    do {
        /* read next sequential index record */
        if (_db_readidx(db, 0) < 0) {
            ptr = NULL;    /* end of index file, EOF */
            goto doreturn;
        }
        /* check if key is all blank (empty record) */
        ptr = db->idxbuf;
        while ( (c = *ptr++) != 0 && c == ' ')
            ;    /* skip until null byte or nonblank */
    } while (c == 0);    /* loop until a nonblank key is found */
    if (key != NULL)
        strcpy(key, db->idxbuf);    /* return key */
    ptr = _db_readdat(db);    /* return pointer to data buffer */
    db->cnt_nextrec++;
doreturn:

```



```

    if (un_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
        err_dump("un_lock error");
    return(ptr);
}

```

16.8 性能

为了测试这个数据库函数库，也为了获得一些时间上的测试数据，我们编写了一个测试程序。这个程序接受两个命令行参数：要创建的子进程的个数和每个子进程向数据库写的数据库记录的条数（*nrec*）。然后创建一个空的数据库（通过调用 `db_open`），通过 `fork` 创建指定数目的子进程，等待所有子进程结束。每个子进程执行以下步骤：

- (1) 向数据库写 *nrec* 条记录。
- (2) 通过关键字读回 *nrec* 条记录。
- (3) 执行下面的循环 *nrec* × 5 次。
 - (a) 随机读一条记录。
 - (b) 每循环 37 次，随机删除一条记录。
 - (c) 每循环 11 次，随机添加一条记录并读取这条记录。
 - (d) 每循环 17 次，随机替换一条记录为新记录。间隔地一次用同样大小的记录替换，一次用比以前更长的记录替换。
- (4) 将此进程写的所有记录删除。每删除一条记录，随机地寻找 10 条记录。

随着函数的调用次数增加，DB 结构的 `cnt_xxx` 变量记录对一个数据库进行的操作数。每个子进程的操作数一般都会与其他子进程不一样，因为每个子进程用来选择记录的随机数生成器是每个子进程根据其进程号来初始化的。当 *nrec* 为 500 时，每个子进程的较典型的操作记录数见表 16-1。

表 16-1 *nrec* 为 500 时，每个子进程调用的操作的典型次数

操 作	计 数
<code>db_store</code> , <code>DB_INSERT</code> 无空记录, 添加到尾部	675
<code>db_store</code> , <code>DB_INSERT</code> , 重用空记录空间	170
<code>db_store</code> , <code>DB_REPLACE</code> , 长度不同, 添加到尾部	100
<code>db_store</code> , <code>DB_REPLACE</code> , 长度相同	100
<code>db_store</code> , 记录没找到	20
<code>db_fetch</code> , 记录找到	8300
<code>db_fetch</code> , 记录没找到	750
<code>db_delete</code> , 记录找到	840
<code>db_delete</code> , 记录没找到	100

读取的次数大约是存储和删除的 10 倍，这可能是许多数据库应用程序的一般情况。

每一个子进程进行这些操作（读取，存储和删除）时，只对此子进程写的记录进行。由于所有的子进程对同一个数据库进行操作（虽然对不同的记录），所以所有的并发控制都会被使用到。数据库中的记录总条数随子进程数按比例增加。（当只有一个子进程时，一开始有 *nrec* 条记录写入数据库。当有两个子进程时，一开始有 *nrec* × 2 条记录写入数据库，依此类推。）

我们通过运行测试程序的三个不同版本来比较加粗锁和加细锁提供的并发，并且比较三种不同的加锁方式（不加锁、建议锁和强制锁）。第一个版本加细锁，用 16.7 节中的程序。第二

个版本通过改变加锁的调用而使用粗锁，16.6节对此已介绍过。第三个版本将所有加锁函数均去掉，这样可以计算加锁的开销。通过改变数据库文件的许可标志位，还可以使第一和第二个版本（加细锁和加粗锁）使用建议锁或强制锁（本节所有的测试报告中，对加细锁只测试了采用强制锁的情况）。

本章所有的测试都在一台运行SVR4的80386系统上进行。

16.8.1 单进程的结果

表16-2显示了只有一个子进程运行的结果，*nrec*分别为500，1000，2000。

表16-2 单进程，不同的*nrec*和不同的加锁方法

<i>nrec</i>	不 加 锁			建 议 锁						强 制 锁		
				粗 锁			细 锁			细 锁		
	用户	系统	时钟	用户	系统	时钟	用户	系统	时钟	用户	系统	时钟
500	15	68	84	16	78	94	15	79	94	16	92	109
1000	61	340	402	63	360	425	63	366	430	71	412	488
2000	157	906	1068	158	936	1096	158	934	1097	159	1081	1253

最后12列显示的是以秒为单位的时间。在所有的情况下，用户 CPU时间加上系统 CPU时间都基本上等于总的运行时间。这一组测试受 CPU限制而不是受磁盘操作限制。

中间6列（建议锁）对加粗锁和加细锁的结果基本一样。这是可以理解的，因为对于单个进程来说加粗锁和加细锁并没有区别。

比较不加锁和加建议锁，可以看到加锁的调用对系统 CPU时间增加了3%到15%。即使这些锁实际上并没有使用过（只有一个进程），fcntl系统调用仍会有一些时间的开销。用户 CPU时间对四种不同的加锁基本上一样，这是因为用户代码基本上是一样的（除了调用fcntl的次数有些不同外）。

关于表16-2要注意的最后一点是强制锁比建议锁增加了大约15%的系统开销。由于对加强制细锁和加建议细锁的调用次数是一样的，故添加的系统开销来自read和write。

最后的运行测试是有多个子进程的不加锁的程序。与预想的一样，结果是随机的错误。一般情况包括：加入到数据库中的记录找不到，测试程序异常退出等。几乎每次运行测试程序，就有不同的错误发生。这是典型的竞态——多个进程在没有任何加锁的情况下修改同一个文件，错误情况不可预测。

16.8.2 多进程的结果

下一组测试主要目的是比较粗锁和细锁的不同。前面说过我们认为由于加细锁时数据库的各个部分被加锁的时间比加粗锁少，所以加细锁应能够提供更好的并发。表16-3显示了对*nrec*取500，子进程数目从1到12的测试结果。

所有的用户时间、系统时间和总时间的单位均为秒。所有这些时间均是父进程与所有子进程的总和。关于这些数据有许多需要考虑。

第8列是加建议粗锁与加建议细锁的运行总时间的时间差。从中可以看到使用细锁得到了多大的并发度。在运行测试的系统上，当使用不多于7个进程时，加粗锁要快。即使是在使用多于7个进程后，使用细锁的时间减少也不大（大约1%），这使我们怀疑使用那么多代码来实现细锁是否值得。

表16-3 nrec=500时不同加锁方法的比较

进程	建议锁							强制锁			
	粗锁			细锁			Δ	细锁			Δ
	用户	系统	时钟	用户	系统	时钟		用户	系统	时钟	
1	16	79	96	16	83	99	3	16	96	112	16
2	42	230	273	43	237	281	8	43	271	315	14
3	79	454	536	81	464	547	11	78	545	626	18
4	128	753	884	132	757	892	8	123	888	1015	17
5	185	1123	1315	196	1173	1376	61	189	1366	1560	16
6	262	1601	1870	270	1611	1888	18	264	1931	2205	20
7	351	2164	2526	354	2174	2537	11	341	2527	2877	16
8	451	2801	3264	454	2766	3230	-34	438	3298	3750	19
9	565	3513	4092	569	3483	4067	-25	548	4148	4712	19
10	684	4293	5000	688	4215	4925	-75	658	5048	5732	20
11	812	5151	5987	811	5043	5876	-111	797	6198	7020	23
12	958	6075	7058	960	5992	6980	-78	937	7298	8265	22

我们认为从粗锁到细锁总时间会减少，但也认为对系统时间来说细锁仍将比粗锁高，对任何的进程数都应是这样。这样认为的原因是对细锁调用了更多次的 fcntl。如果将表 16-1 中的 fcntl 调用的次数加起来，平均对粗锁有 22 110 次，细锁 25 680 次（表 16-1 中的每个操作对于粗锁要调用两次 fcntl，而对于细锁前三个 db_store 及记录删除（记录找到）需要调用四次 fcntl）。基于此，我们认为由于增加了 16% 的 fcntl 的调用会增加细锁的系统时间。所以测试中进程超过 7 个后，加细锁的系统时间反而下降使人疑惑。

最后一列是从加建议细锁到加强制细锁的系统时间的百分比增量。这与在表 16-2 中看到的强制锁增加约 15%~20% 的系统开销是一致的。

由于所有这些测试的用户代码几乎一样（对加建议细锁和强制细锁增加了一些 fcntl 调用），我们认为对每一行的用户 CPU 时间应基本一样。但在测试中，从加建议粗锁到加建议细锁总要增加约 1%~3% 的用户时间，而从加建议细锁到加强制细锁的用户时间总要减少 1%~3%。关于这一点，没有什么明显的解释。

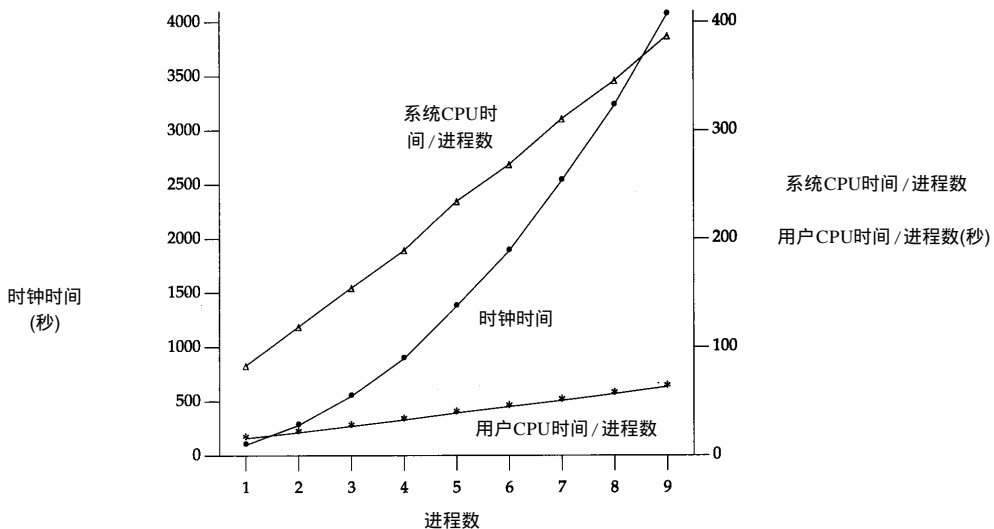


图16-4 表16-3中使用建议细锁的数据

表16-3的第一行与表16-2中的`nrec`取500的那一行很相似。这与预期相一致。

图16-4是表16-3中加建议细锁的数据图。我们绘制了进程数从1到9的总时间（没有绘制10，11和12，因为那要太多垂直坐标空间），也绘制了用户CPU时间除以进程数后的每进程用户CPU时间，另外还绘制了每进程系统CPU时间。

注意到这两个每进程CPU时间都是线性的，但总时间不是线性的。如果把表16-3某一行中的用户CPU时间与系统CPU时间加起来，并与总时间比较，两者的差距随进程数目增多而增大。可能的原因是当进程数增大时操作系统使用的CPU时间增多。操作系统的开销会是总时间增加，但不会影响单个进程的CPU时间。

用户CPU时间随进程数增加的原因可能是因为数据库中有更多的记录。每一条散列链更长，所以`_db_find`函数平均要运行更长时间来找到一条记录。

16.9 小结

本章详细介绍了一个数据库函数库的设计与实现。出于介绍的目的，我们使这个函数库尽可能的小和简单，但也包括了多进程并发控制需要的对记录加锁的功能。

我们使用不同数目的进程，四种不同的加锁方法：不加锁，建议锁（细锁和粗锁），和强制锁，研究了数据库的性能。可以看到建议锁比不加锁在总时间上增加了约10%，强制锁比建议锁耗时再增加约10%。

习题

16.1 在`_db_dodelete`中使用的加锁是比较保守的。例如，如果等到真正要用空闲链表时再加锁，则可获得更大的并发度。如果将调用`writew_lock`移到调用`_db_writedat`和`_db_readptr`之间会发生什么呢？

16.2 如果`db_nextrec`不对空闲链表加读锁而被读的记录正在被删除，描述在怎样的情况下`db_nextrec`会返回一个正确的关键值但是数据记录却是空的（也就不正确了）。

16.3 在讨论了函数`db_store`后我们描述了`_db_writeidx`和`_db_writedat`的加锁。我们说过这种加锁不会干涉除了调用`db_store`外的其他的读进程和写进程。如果改为强制锁，这还成立吗？

16.4 怎样把`fsync`集成到这个数据库函数库中？

16.5 建立一个新的数据库并写入一些数据。写一个程序调用`db_nextrec`来读数据库中的记录，并调用`_db_hash`来计算每条记录的散列值。根据每条散列链上的记录树画出直方图。程序16-9中的函数是否足够？

16.6 修改数据库函数，使得索引文件中散列链的数目可以在数据库建立时指定。

16.7 如果你的系统支持网络文件系统，如Sun的网络文件系统（NFS）或AT&T的远程文件共享（RFS），比较两种情况的性能：（a）数据库与测试程序在同一台机器上，（b）数据库与测试程序在不同的机器上。这个数据库函数库提供的记录锁机制还能工作吗？

第17章 与PostScript 打印机通信

17.1 引言

我们现在开发一个可以与 PostScript打印机通信的程序。PostScript打印机目前使用很广，它一般通过RS-232端口与主机相连。这样就使得我们有可能使用第 11章中的终端I/O函数。同样，与PostScript打印机通信是全双工的，在发送数据给打印机时也要准备好从打印机读取状态消息。这样，又有可能使用 12.5节中的I/O多路转接函数：select 和poll。所开发的这个程序基于James Clark 所写的lprps程序。这个程序和其他一些程序共同组成 lprps软件包，可以在comp.sources.misc新闻组中找到（Volume 21，1991年7月）。

17.2 PostScript 通信机制

关于PostScript打印机所需要知道的第一件事就是我们并不是发送一个文件给打印机去打印——而是发送一个PostScript程序给打印机让它去执行。在PostScript打印机中通常有一个PostScript解释器来执行这个程序，生成输出的页面。如果PostScript程序有错误，PostScript打印机（实际上是PostScript解释器）返回一个错误消息，或许还会产生其他输出。

下面的PostScript程序在输出页面上生成一个熟悉的字符串“hello, world”（这里并不叙述PostScript编程，详细情况请参见Adobe Systems [1985和1986]，而是着重在与PostScript打印机的通信上）。

```
%!  
/Times-Roman findfont  
15 scalefont           % point size of 15  
setfont                % establish current font  
300 350 moveto         % x=300 y=350 (position on page)  
(hello, world) show    % output the string to current page  
showpage               % and output page to output device
```

如果将PostScript程序中的setfont改变为ssetfont，再把它发送到PostScript打印机，结果是什么也没有被打印。相反的，从打印机得到以下消息：

```
%% [ Error: undefined; OffendingCommand: ssetfont ]%%  
%% [ Flushing: rest of job (to end-of-file) will be ignored ]%%
```

这些错误消息随时都可能产生，这也是处理PostScript打印机复杂的地方。我们不能只是将整个PostScript程序发送给打印机后就不管了——还必须处理这些潜在的错误消息（本章所说的“打印机”，就是指PostScript解释器）。

PostScript打印机通常通过RS-232串口与主机相连。这就如同终端的连接一样，所以第 11章中的终端I/O函数在这里也适用（PostScript打印机也可以通过其他方式连接到主机上，例如逐渐流行的网络接口。但目前占主导地位的还是串口相连）。图17-1显示了典型的工作过程。一个PostScript程序可以产生两种形式的输出：通过showpage操作输出到打印机页面上，或者通过print操作输出到它的标准输出（在这里是与主机的串口连接）。

PostScript解释器发送和接受的是7位ASCII字符。PostScript程序可包含所有可打印的ASCII

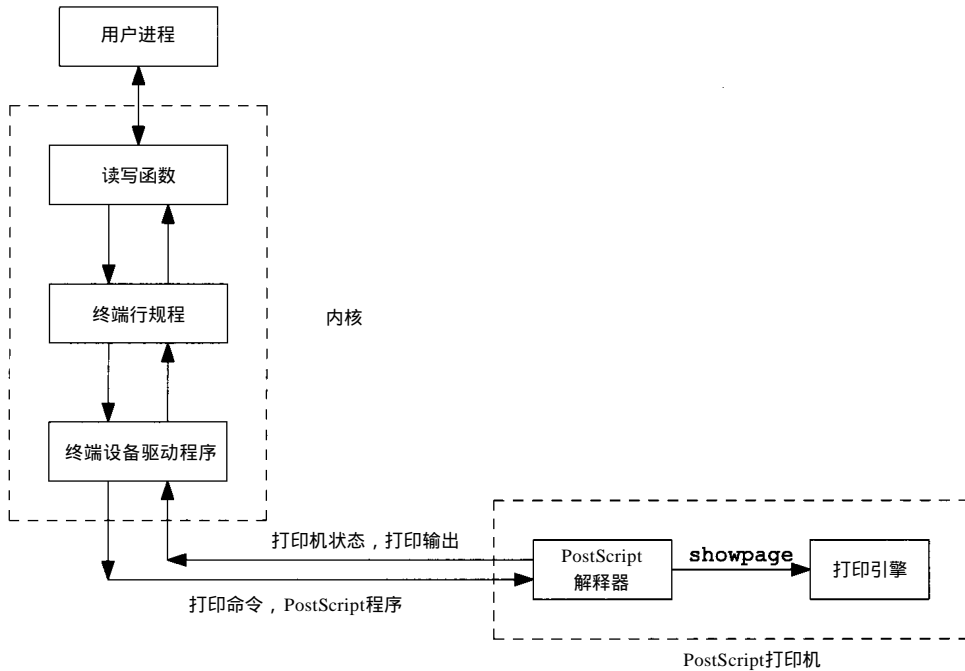


图17-1 用串口连接与PostScript打印机通信

字符。一些不可以打印的字符有着特殊的含义（见表17-1）。

表17-1 从主机发送到PostScript打印机的特殊字符

字符	八进制值	说明
Control-C	003	中断，导致PostScript解释器中断正在执行的操作。通常，这会终止正在解释的程序
Control-D	004	文件终止符
Line feed	012	行终止符，PostScript换行字符。如果顺序接收到回车和换行符，则只有换行符传给PostScript解释器
Return	015	行终止符。它被解释为换行符
Control-Q	021	开始输出（XON流控制）
Control-S	023	结束输出（XOFF流控制）
Control-T	024	状态查询。PostScript解释器返回一个一行的状态消息

PostScript的文件终止符（Ctrl-D）用来同步打印机和主机。发送一个PostScript程序到打印机，然后再发送一个EOF到打印机。当打印机执行完PostScript程序后，它就发回一个EOF。

当PostScript解释器正在执行一个PostScript程序时，可以向它发送一个中断（Ctrl-C）。这通常使正在被打印机执行的程序终止。

状态查询消息（Ctrl-T）会使得打印机返回一个一行的状态消息。所有的打印机消息都是如下格式：

```
%% [ key : val ] %%
```

所有可能出现在状态消息中的key:val对，被分号分开。回忆前面例子的返回消息：

```
%% [ Error: undefined; OffendingCommand: ssetfont ]%%
%% [ Flushing: rest of job (to end-of-file) will be ignored ]%%
```

这个状态消息具有这个格式：

```
%% [ status : idle ]%%
```

除了idle（没有作业）外，其他状态指示还有 busy（正在执行一个 PostScript程序）、waiting（正在等待执行PostScript程序）、printing（打印中）、initializing（初始化）和printing test page（正打印测试页）。

现在来考虑PostScript解释器自己产生的状态消息。可以看到以下的消息：

```
%% [ Error:error ; OffendingCommand operator ]%%
```

总共大约会发生 25种不同的 error。通常的 error有dictstackunderflow, invalidaccess, typecheck, 和undefined。这里的operator是产生错误的PostScript操作。

一个打印机的错误用以下形式来指示：

```
%% [ PrinterErrorreason ]%%
```

这儿的reason一般是Out Of Paper（缺纸）、Cover Open（盖打开）或Miscellaneous Error（其他错误）。

当错误发生时，PostScript解释器经常会发出另外一个消息：

```
%% [ Flushing : rest of job (to end-of-file) will be ignored ] %%
```

我们看一下在特殊字符序列%%[和]%%中的字符串，为了处理这个消息，必须分析该字符串。一个PostScript程序也可以通过PostScript的print操作来产生输出。这个输出应当传给发送程序给打印机的用户（虽然这并不是打印程序所期望解释的）。

表17-2列出了PostScript解释器传送给主机的特殊字符。

表17-2 PostScript解释器传送给主机的特殊字符

字 符	八进制值	说 明
Control-D	004	文件终止符
Line feed	012	换行符，如果在 PostScript解释器的标准输出写入一个换行符，则它被解释为一个回车符加上一个换行符
Control-Q	021	开始输出（XON流控制）
Control-S	023	结束输出（XOFF流控制）

17.3 假脱机打印

本章所开发的程序通过两种方式发送 PostScript程序给PostScript打印机：单独的方式或者通过BSD行式打印机假脱机系统。通常使用假脱机系统，但提供一个独立的方式也很有用，如用于测试等。

UNIX SVR4同样提供了一个假脱机打印系统，在 AT&T手册〔1991〕第一部分以lp开头的册页中可以找到假脱机系统的详细资料。Stevens〔1990〕的第13章详细说明了BSD和pre-SVR4的假脱机系统。本章并不着重在假脱机系统上，而在于与 PostScript打印机的通信。

在BSD的假脱机系统中，以如下形式打印一个文件：

```
lpr -pps main.c
```

这个命令发送文件 main.c 到名为 ps 的打印机。如果没有指定 -pps 的选项，那么或者输出到 PRINTER 环境变量对应的打印机上，或者输出到缺省的 lp 打印机上。所用的打印机参数可以在 /etc/printcap 文件中查到。图 17-2 是对应一个 PostScript 打印机的一项。

```
lp|ps:\
:br#19200:lp=/dev/ttyb:\
:sf:sh:rw:\
:fc#0000374:fs#0000003:xc#0:xs#0040040:\
:af=/var/adm/psacct:lf=/var/adm/pslog:sd=/var/spool/pslpd:\
:if=/usr/local/lib/psif:
```

图 17-2 一个 PostScript 打印机对应的 printcap 项

第一行给出了该项的名称，ps 或者 lp。br 的值指定了波特率是 19 200。lp 指定了该打印机的特殊设备文件路径名。sf 是格式送纸，sh 是指打印作业的开始加入一个打印页头，rw 指定打印机以读写方式打开。如 17.2 节所述，这一项是 PostScript 打印机所必须的。

下面四个域指定了在旧版本 BSD 风格的 stty 结构中需要打开和关闭的位（这里对此进行叙述是因为大多数使用 printcap 文件的 BSD 系统都支持这种老式的设置终端方式的方法。在本章的源程序文件中，可以看到如何使用第 11 章所述的 POSIX.1 函数来设置所有的终端参数。）首先，fc 掩码清除 sg_flags 元素中的下列值：EVENP 和 ODDP（关闭了奇偶校验）、RAW（关闭 raw 模式）、CRMOD（关闭了输入输出中的 CR/LF 映射）、ECHO（关闭回显）和 LCASE（关闭输入输出中的大小写映射）。然后，fs 掩码打开了以下位：CBREAK（一次输入一个字符）和 TANDEM（主机产生 Ctrl-S，Ctrl-Q 流控制）。接着，xc 掩码清除了本地模式字中各位。最后，xs 掩码设置了本地模式字中的下列两位：LDECCTQ（Ctrl-Q 重新开始输出，Ctrl-S 则停止输出）和 LLITOUT（压缩输出转换）。

af 和 lf 字符串分别指定了记帐文件和日志文件。sd 指定了假脱机的目录，而 if 指定了输入过滤程序。

输入过滤程序可被所有的打印文件所调用，格式如下：

```
filter -n loginname -h hostname acctfile
```

这里还有几个可选的参数（这些参数有可能被 PostScript 打印机所忽略）。要打印的文件在标准输入中，打印机（printcap 文件中的 lp 项）设在标准输出。标准输入也可以是一个管道。

使用一个 PostScript 打印机，输入过滤程序首先查询输入文件的开始两个字符，确定这个文件是 ASCII 文本文件还是 PostScript 程序，通常的惯例是前两个字符为 %! 表示这是一个 PostScript 程序。如果这个文件是 PostScript 程序，lprps 程序（下面将详细讨论）就把它发送到打印机。如果这个文件是文本文件，就使用其他程序将它转换成 PostScript 程序。

printcap 文件中提到的 psif 过滤程序是 lprps 软件包提供的。这个包中的 texttps 可以将文本文件转换成 PostScript 程序。图 17-3 概略表示了这些程序。

图中有一个程序 psrev 没有表示出来，该程序将 PostScript 程序生成的页面反转过来，当 PostScript 打印机在正面而不是在反面打印输出时，就可以使用此程序。

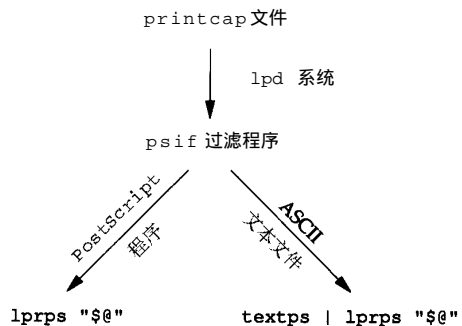


图 17-3 lprps 系统示意图

以下将介绍lprps程序的设计和编码。

17.4 源码

先看一下main调用的函数，以及它们是如何与打印机交互作用的。表 17-3详细表明了这种交互作用。表中第二列指定该函数是否可以通过接受 SIGINT信号而中断。第三列指定了各个函数的超时设置（以秒为单位）。注意，当发送用户的PostScript程序到打印机时，没有超时设置。这是因为一个 PostScript程序可能用任一长的时间来执行。函数 get_page行中的“our PostScript program”是指程序 17-9，这个程序是用来记录当前页码的。

表17-3 main程序调用的函数

函 数	可中断?	超时?	发送给打印机	从打印机得到
get_status	否	5	Ctrl-T	%%[status: idle]%%
get_page	否	30	我们的PostScript程序	%%[pagecount n]%%
			EOF	
			EOF	
send_file	是	无	用户的Postscript程序	
			EOF	
			EOF	
get_page	否	30	我们的Postscript程序	%%[pagecount n]%%
			EOF	
			EOF	

程序17-1列出了头文件lprps.h。该文件包含在所有的源文件中。该头文件包含了各个源程序所需的系统头文件，定义了全局变量和全局函数的原型。

程序17-1 lprps.h头文件

```
#include <sys/types.h>
#include <sys/time.h>
#include <errno.h>
#include <signal.h>
#include <syslog.h> /* since we're a daemon */
#include "ourhdr.h"

#define EXIT_SUCCESS 0 /* defined by BSD spooling system */
#define EXIT_REPRINT 1
#define EXIT_THROW_AWAY 2

#define DEF_DEVICE "/dev/ttyb" /* defaults for debug mode */
#define DEF_BAUD B19200

/* modify following as appropriate */
#define MAILCMD "mail -s \"printer job\" %s@%s < %s"

#define OBSIZE 1024 /* output buffer */
#define IBSIZE 1024 /* input buffer */
#define MBSIZE 1024 /* message buffer */

/* declare global variables */
extern char *loginname;
extern char *hostname;
extern char *acct_file;
extern char eofc; /* PS end-of-file (004) */
extern int debug; /* true if interactive (not a daemon) */
extern int in job; /* true if sending user's PS job to printer */
```

```

extern int  psfd;          /* file descriptor for PostScript printer */
extern int  start_page; /* starting page# */
extern int  end_page;    /* ending page# */

extern volatile sig_atomic_t  intr_flag; /* set if SIGINT is caught */
extern volatile sig_atomic_t  alrm_flag; /* set if SIGALRM goes off */

extern enum status {      /* printer status */
    INVALID, UNKNOWN, IDLE, BUSY, WAITING
} status;

/* global function prototypes */
void  do_acct(void);      /* acct.c */

void  clear_alrm(void);  /* alarm.c */
void  handle_alrm(void);
void  set_alrm(unsigned int);

void  get_status(void);  /* getstatus.c */

void  init_input(int);   /* input.c */
void  proc_input_char(int);
void  proc_some_input(void);
void  proc_upto_eof(int);

void  clear_intr(void);  /* interrupt.c */
void  handle_intr(void);
void  set_intr(void);

void  close_mailfp(void); /* mail.c */
void  mail_char(int);
void  mail_line(const char *, const char *);

void  msg_init(void);    /* message.c */
void  msg_char(int);
void  proc_msg(void);

void  out_char(int);     /* output.c */

void  get_page(int *);  /* pagecount.c */

void  send_file(void);  /* sendfile.c */

void  block_write(const char *, int); /* tty.c */
void  tty_flush(void);
void  set_block(void);
void  set_nonblock(void);
void  tty_open(void);

```

文件vars.c（见程序17-2）定义了全局变量。

程序17-3从执行main函数开始。因为此程序一般是作为精灵进程运行的，所以main函数调用log_open函数（见附录B）。不能将错误消息写到标准错误上——而应使用13.4.2节中描述的syslog设施。

程序17-2 声明全局变量

```

#include    "lprps.h"

char  *loginname;
char  *hostname;
char  *acct_file;
char  eofc = '\004';          /* Control-D = PostScript EOF */

int  psfd = STDOUT_FILENO;
int  start_page = -1;

```

```
int     end_page = -1;
int     debug;
int     in_job;

volatile sig_atomic_t  intr_flag;
volatile sig_atomic_t  alrm_flag;

enum status      status = INVALID;
```

程序17-3 main函数

```
#include "lprps.h"

static void usage(void);

int
main(int argc, char *argv[])
{
    int     c;

    log_open("lprps", LOG_PID, LOG_LPR);

    opterr = 0;      /* don't want getopt() writing to stderr */
    while ( (c = getopt(argc, argv, "cdh:i:l:n:x:y:w:")) != EOF) {
        switch (c) {
            case 'c':      /* control chars to be passed */
            case 'x':      /* horizontal page size */
            case 'y':      /* vertical page size */
            case 'w':      /* width */
            case 'l':      /* length */
            case 'i':      /* indent */
                break; /* not interested in these */

            case 'd':      /* debug (interactive) */
                debug = 1;
                break;

            case 'n':      /* login name of user */
                loginname = optarg;
                break;

            case 'h':      /* host name of user */
                hostname = optarg;
                break;

            case '?':
                log_msg("unrecognized option: -%c", optopt);
                usage();
        }
    }

    if (hostname == NULL || loginname == NULL)
        usage();      /* require both hostname and loginname */

    if (optind < argc)
        acct_file = argv[optind];      /* remaining arg = acct file */

    if (debug)
        tty_open();

    if (atexit(close_mailfp) < 0)      /* register func for exit() */
        log_sys("main: atexit error");

    get_status();

    get_page(&start_page);

    send_file();      /* copies stdin to printer */
}
```

```

    get_page(&end_page);
    do_acct();
    exit(EXIT_SUCCESS);
}

static void
usage(void)
{
    log_msg("lprps: invalid arguments");
    exit(EXIT_THROW_AWAY);
}

```

然后处理命令行参数，很多参数会被PostScript打印机忽略。-d标志指示这个程序是交互运行，而不是作为精灵进程。如果设置了这个标志，则需要初始化终端模式（tty_open）。将函数close_mailfp指定为退出处理程序。

然后可以调用在表17-3中提到的函数：取得打印机状态保证它是就绪的（get_status），得到打印机的起始页码（get_page），发送文件（PostScript程序）到打印机（send_file），得到打印机的结束页码（get_page），写记帐记录（do_acct），然后终止。

文件acct.c定义了函数do_acct（见程序17-4）。它在main函数的结尾处被调用，用来写下记帐记录。记帐文件的路径和名字从printcap文件中的相应记录项（见图17-2）获得，并作为命令行的最后一个参数。

程序17-4 do_acct函数

```

#include    "lprps.h"

/* Write the number of pages, hostname, and loginname to the
 * accounting file. This function is called by main() at the end
 * if all was OK, by printer_flushing(), and by handle_intr() if
 * an interrupt is received. */

void
do_acct(void)
{
    FILE    *fp;

    if (end_page > start_page &&
        acct_file != NULL &&
        (fp = fopen(acct_file, "a")) != NULL) {
        fprintf(fp, "%7.2f %s:%s\n",
                (double)(end_page - start_page),
                hostname, loginname);
        if (fclose(fp) == EOF)
            log_sys("do_acct: fclose error");
    }
}

```

从历史上看，所有的BSD打印过滤程序都使用%7.2f printf格式，把输出的页数写到记帐文件中。这样就允许光栅设备不使用页数，而以英尺为单位报告输出长度。

下面一个文件是tty.c（见程序17-5），它包含了所有的终端I/O函数。这些函数调用第3章中提到的函数（fcntl、write和open）和第11章中的POSIX.1终端函数（tcflush、tcgetattr、tsetattr、cfsetispeed和cfsetospeed）。如果允许发生写阻塞，那么要调用block_write函数。如果不希望发生阻塞，则调用set_nonblock函数，然后再调用read或者write函数。因为PostScript是一个全双工的设备，打印机有可能发送数据回来（例如出错消息等），所以不希望阻塞一个方向的写操作。

如果打印机发送错误消息时，我们正因向其发送数据而处于阻塞状态，则会出现死锁。

一般是内核为终端输入和输出进行缓存，所以如果发生错误，则可以调用 `tty_flush` 来刷新输入和输出队列。

如果以交互方式运行该程序，那么 `main` 函数将调用函数 `tty_open`。需要把终端设为非规范模式，设置波特率和其他一些终端标志。注意各种 PostScript 打印机的这些设置并不都一样。检查你的打印机手册确定它的设置。（数据的位数可能是 7 位或 8 位，起始位、停止位的数目以及奇偶校验等都可能因打印机而异。）

程序 17-5 终端函数

```
#include "lprps.h"
#include <fcntl.h>
#include <termios.h>

static int      block_flag = 1;      /* default is blocking I/O */

void
set_block(void)      /* turn off nonblocking flag */
{
    int      val;

    /* called only by block_write() below */

    if (block_flag == 0) {
        if ( (val = fcntl(psf, F_GETFL, 0)) < 0)
            log_sys("set_block: fcntl F_GETFL error");
        val &= ~O_NONBLOCK;
        if (fcntl(psf, F_SETFL, val) < 0)
            log_sys("set_block: fcntl F_SETFL error");

        block_flag = 1;
    }
}

void
set_nonblock(void) /* set descriptor nonblocking */
{
    int      val;

    if (block_flag) {
        if ( (val = fcntl(psf, F_GETFL, 0)) < 0)
            log_sys("set_nonblock: fcntl F_GETFL error");
        val |= O_NONBLOCK;
        if (fcntl(psf, F_SETFL, val) < 0)
            log_sys("set_nonblock: fcntl F_SETFL error");

        block_flag = 0;
    }
}

void
block_write(const char *buf, int n)
{
    set_block();
    if (write(psf, buf, n) != n)
        log_sys("block_write: write error");
}

void
tty_flush(void)      /* flush (empty) tty input and output queues */
{
    if (tcflush(psf, TCIOFLUSH) < 0)
        log_sys("tty_flush: tcflush error");
}
```

```

void
tty_open(void)
{
    struct termios term;

    if ( (psfd = open(DEF_DEVICE, O_RDWR)) < 0)
        log_sys("tty_open: open error");

    if (tcgetattr(psfd, &term) < 0) /* fetch attributes */
        log_sys("tty_open: tcgetattr error");
    term.c_cflag = CS8 | /* 8-bit data */
                CREAD | /* enable receiver */
                CLOCAL; /* ignore modem status lines */
                /* no parity, 1 stop bit */
    term.c_oflag &= ~OPOST; /* turn off post processing */
    term.c_iflag = IXON | IXOFF | /* Xon/Xoff flow control */
                IGNBRK | /* ignore breaks */
                ISTRIP | /* strip input to 7 bits */
                IGNCR; /* ignore received CR */
    term.c_lflag = 0; /* everything off in local flag:
                    disables canonical mode, disables
                    signal generation, disables echo */
    term.c_cc[VMIN] = 1; /* 1 byte at a time, no timer */
    term.c_cc[VTIME] = 0;
    cfsetispeed(&term, DEF_BAUD);
    cfsetospeed(&term, DEF_BAUD);
    if (tcsetattr(psfd, TCSANOW, &term) < 0) /* set attributes */
        log_sys("tty_open: tcsetattr error");
}

```

该程序处理两个信号：SIGINT和SIGALRM。处理SIGINT是对BSD假脱机系统调用的任何一种过滤程序的要求。如果打印机作业被 lprm(1)命令删除，那么这个信号被发送给过滤程序。使用SIGALRM来设置超时，对这两个信号用类似的方式处理：提供了 set_XXX函数来建立信号处理器，clear_XXX函数来取消这个信号处理器。如果有信号传送给这个进程，信号处理器在设置一个全局的标记 intr_flag和alarm_flag后返回。程序的其他部分可在适当的时间来检测这些标记，有一个明显的时间是在 I/O函数返回错误 EINTR时，该程序然后调用 handle_intr或者 handle_alarm来处理这种情况。调用 signal_intr函数（见程序10-13）来中断一个慢速的系统调用。程序17-6是处理SIGINT信号的interrupt.c文件。

当一个中断发生时，必须发送PostScript的中断字符（Ctrl-C）给打印机，接着发送一个文件终止符(EOF)。这通常引起PostScript解释器终止它正在解释的程序，然后等待从打印机返回的EOF（稍后将描述proc_upto_eof函数）。此时读取最后的页码，写下记帐记录，然后就可以终止了。

程序17-6 处理中断信号的interrupt.c文件

```

#include    "lprps.h"

static void
sig_int(int signo) /* SIGINT handler */
{
    intr_flag = 1;
    return;
}

/* This function is called after SIGINT has been delivered,
 * and the main loop has recognized it. (It not called as
 * a signal handler, set_intr() above is the handler.) */

```

```

void
handle_intr(void)
{
    char    c;

    intr_flag = 0;
    clear_intr();          /* turn signal off */

    set_alarm(30);        /* 30 second timeout to interrupt printer */

    tty_flush();          /* discard any queued output */
    c = '\003';
    block_write(&c, 1);    /* Control-C interrupts the PS job */
    block_write(&eofc, 1); /* followed by EOF */
    proc_upto_eof(1);     /* read & ignore up through EOF */

    clear_alarm();

    get_page(&end_page);
    do_acct();
    exit(EXIT_SUCCESS);   /* success since user lprm'ed the job */
}

void
set_intr(void)           /* enable signal handler */
{
    if (signal_intr(SIGINT, sig_int) == SIG_ERR)
        log_sys("set_intr: signal_intr error");
}

void
clear_intr(void)        /* ignore signal */
{
    if (signal(SIGINT, SIG_IGN) == SIG_ERR)
        log_sys("clear_intr: signal error");
}

```

表17-3写明了哪些函数设置超时。仅在以下情况下设置超时：查询打印机状态 (get_status)、读取打印机的页码 (get_page) 或者正中断打印机时 (handle_intr)。如果发生了超时，只需要记录下错误，过一段时间后终止。程序 17-7 是 alarm.c 文件。

程序 17-7 处理超时的 alarm.c 文件

```

#include    "lprps.h"

static void
sig_alarm(int signo)    /* SIGALRM handler */
{
    alarm_flag = 1;
    return;
}

void
handle_alarm(void)
{
    log_ret("printer not responding");
    sleep(60);          /* it will take at least this long to warm up */

    exit(EXIT_REPRINT);
}

void          /* Establish the signal handler and set the alarm. */
set_alarm(unsigned int nsec)

```

```

{
    alarm_flag = 0;
    if (signal_intr(SIGALRM, sig_alm) == SIG_ERR)
        log_sys("set_alarm: signal_intr error");
    alarm(nsec);
}

void
clear_alarm(void)
{
    alarm(0);
    if (signal(SIGALRM, SIG_IGN) == SIG_ERR)
        log_sys("clear_alarm: signal error");
    alarm_flag = 0;
}

```

程序17-8是函数`get_status`，这个函数由`main`函数调用。它发送一个`Ctrl-T`到打印机以获取打印机的状态消息。打印机回送一行消息。如果接收到的消息是：

```
%%[ status : idle ]%%
```

则意味着打印机准备好执行一个新的作业。这个消息由函数`proc_some_input`读取和处理（下面将讨论这个函数）。

程序17-8 `get_status`函数

```

#include    "lprps.h"

/* Called by main() before printing job.
 * We send a Control-T to the printer to fetch its status.
 * If we timeout before reading the printer's status, something
 * is wrong. */

void
get_status(void)
{
    char    c;

    set_alarm(5);          /* 5 second timeout to fetch status */
    tty_flush();
    c = '\024';
    block_write(&c, 1);    /* send Control-T to printer */

    init_input(0);
    while (status == INVALID)
        proc_some_input(); /* wait for something back */

    switch (status) {
    case IDLE: /* this is what we're looking for ... */
        clear_alarm();
        return;

    case WAITING: /* printer thinks it's in the middle of a job */
        block_write(&eofc, 1); /* send EOF to printer */
        sleep(5);
        exit(EXIT_REPRINT);

    case BUSY:
    case UNKNOWN:
        sleep(15);
        exit(EXIT_REPRINT);
    }
}

```


如果接收到下列消息：

```
%%[ status: waiting ]%%
```

则说明打印机正等待我们发送更多的数据以用于当前正打印的作业，这很可能是前一打印作业出了些问题。为了清除这个状态，发送给打印机一个EOF终止符。

PostScript打印机维护着一个页码计数器。这个计数器每打印一页就会增加，它即使在关闭电源时也会保存。为了读此计数器，需要发送给打印机一个PostScript程序。文件pagecount.c（见程序17-9）包含了这个小PostScript程序（含有大约10个PostScript操作符）和函数get_page。

程序17-9 pagecount.c文件——得到打印机的计数器值

```
#include "lprps.h"

/* PostScript program to fetch the printer's pagecount.
 * Notice that the string returned by the printer:
 * %%[ pagecount: N ]%%
 * will be parsed by proc_msg(). */

static char pagecount_string[] =
    "(%[ pagecount: ) print " /* print writes to current output file */
    "statusdict begin pagecount end " /* push pagecount onto stack */
    "20 string " /* creates a string of length 20 */
    "cvx " /* convert to string */
    "print " /* write to current output file */
    "( ]%%) print "
    "flush\n"; /* flush current output file */

/* Read the starting or ending pagecount from the printer.
 * The argument is either &start_page or &end_page. */

void
get_page(int *ptrcount)
{
    set_alarm(30); /* 30 second timeout to read pagecount */

    tty_flush();
    block_write(pagecount_string, sizeof(pagecount_string) - 1);
    /* send query to printer */

    init_input(0);
    *ptrcount = -1;
    while (*ptrcount < 0)
        proc_some_input(); /* read results from printer */

    block_write(&eofc, 1); /* send EOF to printer */
    proc_upto_eof(0); /* wait for EOF from printer */

    clear_alarm();
}
```

pagecount_string数组包含了这个小PostScript程序。虽然我们可以用如下方法得到并打印页码：

```
statusdict begin pagecount end = flush
```

但我们希望得到类似于打印机返回的状态消息的输出格式：

```
%% [ pagecount N ]%%
```

然后，proc_some_input函数处理这个消息，其方式与处理打印机的状态消息相类似。

程序17-10中的函数send_file由main函数调用，它将用户的PostScript程序发送到打印机上。

程序17-10 send_file函数

```

#include    "lprps.h"

void
send_file(void)    /* called by main() to copy stdin to printer */
{
    int    c;

    init_input(1);
    set_intr();    /* we catch SIGINT */

    while ( (c = getchar()) != EOF)    /* main loop of program */
        out_char(c);    /* output each character */
    out_char EOF;    /* output final buffer */

    block_write(&eofc, 1);    /* send EOF to printer */
    proc_upto_eof(0);    /* wait for printer to send EOF back */
}

```

这个函数主要是一个while循环，首先读取标准输入（getchar），然后调用函数out_char把字符发送给打印机。当在标准输入上遇到EOF时，就发送一个EOF给打印机（指示作业完成），然后等待从打印机返回一个EOF(proc_upto_eof)。

回忆图17-1，连接在串口的PostScript解释器的输出可能是打印机状态消息或者是PostScript的print操作符的输出。所以，我们所认为的“文件被打印”甚至可能一页都不输出。这个PostScript程序文件执行后，把它的结果送回主机。PostScript不是一种编程语言，但是，有时确实需要发送一个PostScript程序到打印机并将结果返回主机，而不是打印在纸上。一个例子是取页码数的PostScript程序，用其可以了解打印机的使用情况。

```

%!
statusdict begin pagecount end =

```

如果从PostScript解释器返回的不是状态消息，就以电子邮件形式发送给用户。程序17-11的mail.c完成这一功能。

程序17-11 mail.c文件

```

#include    "lprps.h"

static FILE *mailfp;
static char temp_file[L_tmpnam];
static void open_mailfp(void);

/* Called by proc_input_char() when it encounters characters
 * that are not message characters. We have to send these
 * characters back to the user. */

void
mail_char(int c)
{
    static int done_intro = 0;

    if (in_job && (done_intro || c != '\n')) {
        open_mailfp();
        if (done_intro == 0) {
            fputs("Your PostScript printer job "
                "produced the following output:\n", mailfp);
            done_intro = 1;
        }
        putc(c, mailfp);
    }
}

```

```

/* Called by proc_msg() when an "Error" or "OffendingCommand" key
 * is returned by the PostScript interpreter. Send the key and
 * val to the user. */
void
mail_line(const char *msg, const char *val)
{
    if (in_job) {
        open_mailfp();
        fprintf(mailfp, msg, val);
    }
}

/* Create and open a temporary mail file, if not already open.
 * Called by mail_char() and mail_line() above. */
static void
open_mailfp(void)
{
    if (mailfp == NULL) {
        if ( (mailfp = fopen(tmpnam(temp_file), "w")) == NULL)
            log_sys("open_mailfp: fopen error");
    }
}

/* Close the temporary mail file and send it to the user.
 * Registered to be called on exit() by atexit() in main(). */
void
close_mailfp(void)
{
    char    command[1024];
    if (mailfp != NULL) {
        if (fclose(mailfp) == EOF)
            log_sys("close_mailfp: fclose error");
        sprintf(command, MAILCMD, loginname, hostname, temp_file);
        system(command);
        unlink(temp_file);
    }
}

```

每次在打印机返回一个字符，而且这个字符不是状态消息的一部分时，那么就调用函数 `mail_char`（下面将讨论函数 `proc_input_char`，它调用 `mail_char`）。只有当函数 `send_file` 正发送一个文件给打印机时，变量 `in_job` 才被设置。在其他时候，例如正在读取打印机的状态消息或者打印机的页码计数器值时，它都不会被设置。然后调用函数 `mail_line`，它将一行写入邮件文件中。

当第一次调用函数 `open_mailfp` 时，它生成一个临时文件并把它打开。函数 `close_mailfp` 由 `main` 函数设置为终止处理程序，当调用 `exit` 时就会调用该函数。如果此时临时邮件文件已经产生，那么关闭这个文件，邮件传给用户。

如果发送一行的 PostScript 程序：

```

%!
statusdict begin pagecount end =

```

来获得打印机的页码计数，那么返回的邮件消息是：

```

Your PostScript printer job produced the following output:
11185

```

`output.c`（见程序 17-12）包含了函数 `out_char`，`send_file` 调用此函数以便将字符输出到打印机。

程序17-12 output.c 文件

```

#include    "lprps.h"

static char outbuf[OBSIZE];
static int  outcnt = OBSIZE;    /* #bytes remaining */
static char *outptr = outbuf;

static void out_buf(void);

/* Output a single character.
 * Called by main loop in send_file(). */

void
out_char(int c)
{
    if (c == EOF) {
        out_buf();    /* flag that we're all done */
        return;
    }

    if (outcnt <= 0)
        out_buf();    /* buffer is full, write it first */

    *outptr++ = c;    /* just store in buffer */
    outcnt--;
}

/* Output the buffer that out_char() has been storing into.
 * We have our own output function, so that we never block on a write
 * to the printer.  Each time we output our buffer to the printer,
 * we also see if the printer has something to send us.  If so,
 * we call proc_input_char() to process each character. */

static void
out_buf(void)
{
    char    *wptr, *rptr, ibuf[IBSIZE];
    int     wcnt, nread, nwritten;
    fd_set  rfd, wfd;

    FD_ZERO(&wfd);
    FD_ZERO(&rfd);
    set_nonblock();    /* don't want the write() to block */
    wptr = outbuf;    /* ptr to first char to output */
    wcnt = outptr - wptr;    /* #bytes to output */
    while (wcnt > 0) {
        FD_SET(psfd, &wfd);
        FD_SET(psfd, &rfd);
        if (intr_flag)
            handle_intr();
        while (select(psfd + 1, &rfd, &wfd, NULL, NULL) < 0) {
            if (errno == EINTR) {
                if (intr_flag)
                    handle_intr();    /* no return */
            } else
                log_sys("out_buf: select error");
        }
        if (FD_ISSET(psfd, &rfd)) {    /* printer is readable */
            if ( (nread = read(psfd, ibuf, IBSIZE)) < 0)
                log_sys("out_buf: read error");
            rptr = ibuf;
            while (--nread >= 0)
                proc_input_char(*rptr++);
        }
        if (FD_ISSET(psfd, &wfd)) {    /* printer is writeable */

```

```

        if ( (nwritten = write(psf, wptr, wcnt)) < 0)
            log_sys("out_buf: write error");
        wcnt -= nwritten;
        wptr += nwritten;
    }
}
outptr = outbuf;    /* reset buffer pointer and count */
outcnt = OBSIZE;
}

```

当传送给 `out_char` 的参数是 EOF 时，表明输入已经结束，最后的输出缓存内容应当发送到打印机。

函数 `out_char` 把每个字符放到输出缓存中，当缓存满了时调用 `out_buf` 函数。编写 `out_buf` 函数必须小心：我们发送数据到打印机，打印机也可能同时送回数据。为了避免写操作的阻塞，必须把描述符设置为非阻塞的。（回忆程序 12-1）。使用 `select` 函数来多路转接双向的 I/O：输入和输出。在读取和写入时都设置同一个描述符。还有一种可能是 `select` 函数可能会被一个信号（SIGINT）所中断，所以必须在任何错误返回时对此进行检查。

如果从打印机接收到异步输入，则调用 `proc_input_char` 来处理每一个字符。这个输入可能是打印机状态消息或者发送给用户的邮件。

当向打印机 `write` 时，必须处理 `write` 返回的计数比期望的数量少的情况。同样，回忆程序 12-1 的例子，其中在每次 `write` 时终端可以接收任一数量的数据。

文件 `input.c`（见程序 17-13），定义了处理所有从打印机输入的函数。这种输入可以是打印机状态消息或者给用户的输出。

程序 17-13 `input.c` 文件——读取和处理打印机的输入

```

#include    "lprps.h"

static int eof_count;
static int ignore_input;
static enum parse_state {    /* state of parsing input from printer */
    NORMAL,
    HAD_ONE_PERCENT,
    HAD_TWO_PERCENT,
    IN_MESSAGE,
    HAD_RIGHT_BRACKET,
    HAD_RIGHT_BRACKET_AND_PERCENT
} parse_state;

/* Initialize our input machine. */

void
init_input(int job)
{
    in_job = job;    /* only true when send_file() calls us */
    parse_state = NORMAL;
    ignore_input = 0;
}

/* Read from the printer until we encounter an EOF.
 * Whether or not the input is processed depends on "ignore". */

void
proc_upto_eof(int ignore)
{
    int    ec;

    ignore_input = ignore;

```

```

    ec = eof_count;      /* proc_input_char() increments eof_count */
    while (ec == eof_count)
        proc_some_input();
}

/* Wait for some data then read it.
 * Call proc_input_char() for every character read. */
void
proc_some_input(void)
{
    char    ibuf[IBSIZE];
    char    *ptr;
    int     nread;
    fd_set  rfd;

    FD_ZERO(&rfd);
    FD_SET(psf, &rfd);
    set_nonblock();
    if (intr_flag)
        handle_intr();
    if (alarm_flag)
        handle_alarm();
    while (select(psf + 1, &rfd, NULL, NULL, NULL) < 0) {
        if (errno == EINTR) {
            if (alarm_flag)
                handle_alarm();      /* doesn't return */
            else if (intr_flag)
                handle_intr();      /* doesn't return */
        } else
            log_sys("proc_some_input: select error");
    }
    if ((nread = read(psf, ibuf, IBSIZE)) < 0)
        log_sys("proc_some_input: read error");
    else if (nread == 0)
        log_sys("proc_some_input: read returned 0");

    ptr = ibuf;
    while (--nread >= 0)
        proc_input_char(ptr++);      /* process each character */
}

/* Called by proc_some_input() above after some input has been read.
 * Also called by out_buf() whenever asynchronous input appears. */
void
proc_input_char(int c)
{
    if (c == '\004') {
        eof_count++;      /* just count the EOFs */
        return;
    } else if (ignore_input)
        return;          /* ignore everything except EOFs */

    switch (parse_state) {      /* parse the input */
    case NORMAL:
        if (c == '%')
            parse_state = HAD_ONE_PERCENT;
        else
            mail_char(c);
        break;
    case HAD_ONE_PERCENT:
        if (c == '%')
            parse_state = HAD_TWO_PERCENT;
        else {
            mail_char('%'); mail_char(c);

```

```
        parse_state = NORMAL;
    }
    break;
case HAD_TWO_PERCENT:
    if (c == '[') {
        msg_init();          /* message starting; init buffer */
        parse_state = IN_MESSAGE;
    } else {
        mail_char('%'); mail_char('%'); mail_char(c);
        parse_state = NORMAL;
    }
    break;
case IN_MESSAGE:
    if (c == ']')
        parse_state = HAD_RIGHT_BRACKET;
    else
        msg_char(c);
    break;
case HAD_RIGHT_BRACKET:
    if (c == '%')
        parse_state = HAD_RIGHT_BRACKET_AND_PERCENT;
    else {
        msg_char(']'); msg_char(c);
        parse_state = IN_MESSAGE;
    }
    break;
case HAD_RIGHT_BRACKET_AND_PERCENT:
    if (c == '%') {
        parse_state = NORMAL;
        proc_msg();          /* we have a message; process it */
    } else {
        msg_char(']'); msg_char('%'); msg_char(c);
        parse_state = IN_MESSAGE;
    }
    break;
default:
    abort();
}
}
```

每当等待从打印机返回EOF时就会调用函数proc_upto_eof。

函数proc_some_input从串口读取。注意我们调用select函数来确定什么时候该描述符是可以读取的，这是因为select函数通常被一个捕捉到的信号所中断——它并不自动地重起。因为select函数能被SIGALRM或SIGINT所中断，故我们并不希望它重起。回忆一下12.5节中关于select函数被正常中断的讨论，同样回忆10.5节中设置SA_RESTART来说明当一个特定信号出现时，应当自动重起动的I/O函数。但是因为并不总是有一个附加的标志，使得我们可以说明I/O函数不应当重起。如果不设置SA_RESTART，我们只能依赖系统的缺省值，而这可能是自动重新起被中断的I/O函数。当从打印机返回输入时，我们以非阻塞模式读取，得到打印机准备就绪的数据，然后调用函数proc_input_char来处理各个字符。

处理打印机发送给我们的消息是由proc_input_char完成的。必须检查每一个字符并记住状态。变量parse_state跟踪记录当前状态。调用msg_char函数把序列%%[以后所有的字符串储存在消息缓存中。当遇到结束序列]%%时，调用proc_msg来处理消息。除了开始%%[和最后]%%序列以及二者之间的状态消息其他字符串，都被认为是用户的输出，被邮递给用户（调用mail_char）。

现在查看那些处理由输入函数积累消息的函数。程序 17-14 是文件 message.c。

当检测到 `%%[` 后，调用函数 `msg_init`，它只是初始化缓存计数器。然后对于消息中的每一个字符都调用 `msg_char` 函数。

函数 `proc_msg` 将消息分解为 `key:val` 对，并检查 `key`。调用 ANSI C 的 `strtok` 函数将消息分解为记号，每个 `key: val` 对用分号分隔。

一个以下形式的消息：

```
%%[ Flushing : rest of job (to end-of-file) will be ignored ]%%
```

引起函数 `printer_flushing` 被调用。它清理终端的缓存，发送一个 EOF 给打印机，然后等待打印机返回一个 EOF。

如果接收到一个以下形式的消息：

```
%%[ PrinterErrorreason]%%
```

则调用 `log_msg` 函数来记录这个错误。`key` 为 `Error` 的出错消息邮递传回用户。这些一般是 PostScript 程序的错误。

如果返回一个 `key` 为 `status` 的状态消息，它很可能是由于函数 `get_status` 发送给打印机一个状态请求 (Ctrl-T) 而引起的。我们查看 `val`，并按照它来设置变量 `status`。

程序 17-14 message.c 文件，处理从打印机返回的消息

```
#include "lprps.h"
#include <ctype.h>

static char msgbuf[MBSIZE];
static int msgcnt;
static void printer_flushing(void);

/* Called by proc_input_char() after it's seen the "%%[" that
 * starts a message. */

void
msg_init(void)
{
    msgcnt = 0;    /* count of chars in message buffer */
}

/* All characters received from the printer between the starting
 * %%[ and the terminating ]%% are placed into the message buffer
 * by proc_some_input(). This message will be examined by
 * proc_msg() below. */

void
msg_char(int c)
{
    if (c != '\0' && msgcnt < MBSIZE - 1)
        msgbuf[msgcnt++] = c;
}

/* This function is called by proc_input_char() only after the final
 * percent in a "%%[ <message> ]%%" has been seen. It parses the
 * <message>, which consists of one or more "key: val" pairs.
 * If there are multiple pairs, "val" can end in a semicolon. */

void
proc_msg(void)
{
    char *ptr, *key, *val;
    int n;

    msgbuf[msgcnt] = 0;    /* null terminate message */
    for (ptr = strtok(msgbuf, ";"); ptr != NULL;
```



```

                                ptr = strtok(NULL, ";")) {
while (isspace(*ptr))
    ptr++;
key = ptr;
if ( (ptr = strchr(ptr, ':')) == NULL)
    continue; /* missing colon, something wrong, ignore */
*ptr++ = '\0'; /* null terminate key (overwrite colon) */
while (isspace(*ptr))
    ptr++; /* skip leading spaces in val */
val = ptr;
                                /* remove trailing spaces in val */
ptr = strchr(val, '\0');
while (ptr > val && isspace(ptr[-1]))
    --ptr;
*ptr = '\0';
if (strcmp(key, "Flushing") == 0) {
    printer_flushing(); /* never returns */
} else if (strcmp(key, "PrinterError") == 0) {
    log_msg("proc_msg: printer error: %s", val);
} else if (strcmp(key, "Error") == 0) {
    mail_line("Your PostScript printer job "
             "produced the error '%s'.\n", val);
} else if (strcmp(key, "status") == 0) {
    if (strcmp(val, "idle") == 0)
        status = IDLE;
    else if (strcmp(val, "busy") == 0)
        status = BUSY;
    else if (strcmp(val, "waiting") == 0)
        status = WAITING;
    else
        status = UNKNOWN; /* "printing", "PrinterError",
                           "initializing", or "printing test page". */
} else if (strcmp(key, "OffendingCommand") == 0) {
    mail_line("The offending command was '%s'.\n", val);
} else if (strcmp(key, "pagecount") == 0) {
    if (sscanf(val, "%d", &n) == 1 && n >= 0) {
        if (start_page < 0)
            start_page = n;
        else
            end_page = n;
    }
}
}
}

/* Called only by proc_msg() when the "Flushing" message
 * is received from the printer. We exit. */
static void
printer_flushing(void)
{
    clear_intr(); /* don't catch SIGINT */
    tty_flush(); /* empty tty input and output queues */
    block_write(&eofc, 1); /* send an EOF to the printer */
    proc_upto_eof(1); /* this call won't be recursive,
                     since we specify to ignore input */
    get_page(&end_page);
    do_acct();
    exit(EXIT_SUCCESS);
}

```

key为OffendingCommand一般总是与其他key: val对一起出现, 如

```
%% [ Error : stackunderflow ; OffendingCommand : pop ]%%
```

则在送回给用户的邮件中就要添加一行。

最后, 函数get_page (见程序17-9) 中的PostScript程序产生一个pagecount的key。我们调用sscanf把val转换为二进制, 设置起始或结束页面值变量。函数get_page中的while循环在等待这个变量变成非负值。

17.5 小结

本章实现了一个完整的程序——它发送一个PostScript程序给连接在RS-232端口的PostScript打印机。这给我们一个实践机会, 把前些章所介绍的很多函数用到一个实用的程序中: I/O多路转接、非阻塞I/O、终端I/O和信号等。

习题

17.1 我们需要使用lprps来打印标准输入的文件, 它也可能是一个管道。因为程序psif一定要查看输入文件的前两个字节, 那么应当如何开发psif程序(见图17-3)来处理这种情况呢?

17.2 实现psif过滤程序, 处理前一个习题中的实例。

17.3 参考Adobe Systems [1998] 12.5节中关于在PostScript程序中字体请求的处理, 修改本章中的lprps程序以处理字体请求。

第18章 调制解调器拨号器

18.1 引言

与调制解调器相关的程序要处理如此种类繁多的调制解调器很困难。在大多数 UNIX系统中总有两个程序来处理调制解调器。第一个是远程登录程序，它允许我们拨通另外的计算机、登录和使用远程系统。在系统V中这个程序叫做cu，而BSD则称它为tip。它们完成类似的工作，而且都可以处理很多不同类型的调制解调器。另一个使用调制解调器的程序是 uucico，它是UUCP包的一部分。问题是不同种类调制解调器的具体特性一般都包含在这些程序的内部，所以，如果想写其他使用调制解调器的程序，就不得不做与这些程序类似的工作。同样，如果想要改变这些程序，使其不通过调制解调器，而利用其他介质通信（例如网络连接），那么也要做很大的改动。

本章开发了一个程序来处理调制解调器所有需要处理的细节。我们把所有这些的细节都集中到这个程序中，而不是分散在多个程序里（这个程序的构思来自于 Presotto 和Ritchie [1990]所描述的连接服务器）。为了使用这个程序，必须能如 15.3节所说明的那样调用它，并使它传回文件描述符。然后，用这个程序来开发远程登录程序（类似 cu和tip ）。

18.2 历史

cu(1) 命令（意思是call UNIX）是在V7中出现的。但它只能处理一种特殊的自动拨号单元（ACU）。伯克利的Bill Shannon修改了cu，并把它实现在4.2BSD的tip(1)中。这之间的最大改变是使用了一个文本文件/etc/remote来存放所有的系统信息（电话号码、优先的拨号器、波特率、奇偶校验、流控制等）。这个版本的tip支持六种不同的拨号单元和调制解调器，如要支持其他种类的调制解调器则要修改源码。

与cu和tip一样，UUCP系统也可以使用调制解调器和自动拨号单元。UUCP对不同的调制解调器进行加锁，因此多个UUCP的实例可以同时运行。这样，tip和cu程序就不得不遵循UUCP协议，避免与UUCP冲突。在BSD系统中，UUCP使用了它自己的拨号函数。这些函数被连接到UUCP的可执行程序中，这样增加新的调制解调器也需要修改源码。

SVR2提供了一个dial(3)函数来将调制解调器拨号的一致特性归纳到一个库函数中。这个函数由cu使用，但UUCP不使用。这是一个标准的C库函数，所以可以被一般程序使用。

Honey DanBer UUCP系统是将调制解调器命令从C源程序中抽取出来，将它们放在一个Dialers文件中。这就允许不修改源码就可以加入新类型的调制解调器。但是cu和UUCP所使用的访问Dialers文件的函数不是很通用。这说明cu和UUCP可以不重新开发代码去处理Dialers文件中的拨号信息，但除了cu和UUCP以外的程序并不能使用这个文件。

在所有这些版本的cu、tip和UUCP中，加锁保证了在同一时间只有一个程序使用某一设备。因为这些程序工作在不同系统中，早期的版本不提供记录锁，而使用一个早期形式的文件加锁，这会导致当一个程序崩溃时，该锁文件仍旧保留，所以又开发特殊的技术来处理这种情况。（对特殊设备文件不能使用记录锁，所以记录锁也不能完全解决问题。）

18.3 程序设计

我们来分析一下调制解调器拨号器所应该具有的特性：

(1) 它必须在不改动源码的情况下支持新增加的调制解调器类型。

为了达到这个目标，我们使用了Honey DanBer的Dialers文件。我们将所有使用这个文件来拨号调制解调器的代码都放到一个精灵进程服务器中，这样任何程序都可以使用 15.5节中的客户机-服务器函数来访问它。

(2) 一定要使用一些特定形式的锁，以保证当那些持有锁的程序在非正常结束时能自动释放它的锁。以前那些专门的技术，如那些在大多数 cu和UUCP版本中仍然使用的技术，都不应再使用。

我们用一个服务器精灵进程来处理所有的设备加锁。因为 15.5节中的客户机-服务器函数会在客户机终止时自动通知服务器，所以这个精灵进程能释放进程所持有的任何加锁。

(3) 新的程序一定要能够使用我们所开发的所有特性。开发一个新的处理调制解调器的程序不应当什么都要自己实现，它拨任何类型的调制解调器应该就像函数调用一样简单方便。为此，我们让中央服务器精灵进程处理所有与拨号有关的操作，并返回一个文件描述符。

(4) 客户机程序，例如cu和tip，不应当需要特别权限。这些程序不应当是设置 -用户-ID程序。但是要给予服务器精灵进程特殊权限，允许客户机程序运行时无需特权。

显然我们不能改动已有的cu、tip和UUCP程序，但应该让其他程序在我们工作的基础上实现起来更加简单。当然，我们也一定要充分吸取已有的UNIX 拨号程序的优点。

图18-1描述了客户机-服务器工作模式的结构。

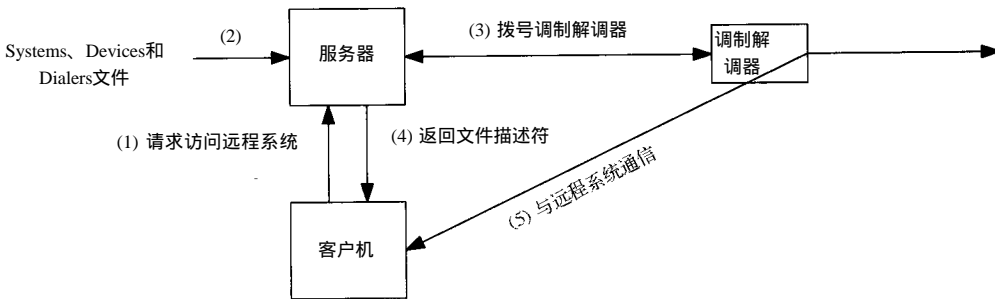


图18-1 客户机-服务器工作模式示意图

建立与远程系统的通信用过程如下：

(1) 起动服务器。

(2) 客户机起动，使用cli_conn函数（见15.5节）建立与服务器的连接。客户机向服务器发出一个请求，请求拨号远程系统。

(3) 服务器读取Systems、Devices和Dialers配置文件来决定如何拨号远程系统（下一节将讲述这些文件）。如果正使用一个调制解调器，在对应的Dialers配置文件中就包含了这个特定调制解调器的所有命令。

(4) 服务器打开该调制解调器设备并拨号该调制解调器。这需要一些时间（一般为15~30秒）。服务器处理所有对该调制解调器的加锁，以避免各用户间的冲突。

(5) 如果拨号成功，服务器返回一个该调制解调器设备的文件描述符给客户机。在 15.3节中的函数可以发送和接受这个描述符。

(6) 客户机直接与远程系统通信。服务器不再参与这个过程。客户机读写上一步返回的文件描述符即可。

客户机与服务器的通信用第(2)~(5)步是通过一个流管道进行的。当客户机完成与远程系统的通信时，客户机关闭该流管道。服务器发现该管道关闭后，释放对调制解调器设备的加锁。

18.4 数据文件

这一节将讲述Honey DanBer UUCP系统所使用的三个文件：Systems、Devices和Dialers。在这些文件中有很多UUCP所使用的域。本节不详细讲述这些域和UUCP系统本身。参见Redman [1989]可得到更详细的信息。

表18-1分栏列出了Systems文件中的六个域。

表18-1 Systems 文件

name	time	type	class	phone	login
host1	任意	ACU	19200	5551234	不使用
host1	任意	ACU	9600	5552345	不使用
host1	任意	ACU	2400	5556789	不使用
modem	任意	modem	19200	-	不使用
laser	任意	laser	19200	-	不使用

*name*域是远程系统的名字。例如，可以使用 `cu host1` 这种形式的命令。这里要注意的是，我们可以对同一远程系统建立多个项。系统按顺序尝试拨号这些项。表18-1中名为 *modem* 和 *laser* 的项对应于与调制解调器和激光打印机的直接连接。并不需要拨号来连接这些设备，但对它们仍需要打开合适的终端连线，并处理好加锁问题。

*Time*域指定了拨号的星期和时间。这是一个UUCP的域。*Type*域指定了对这个特定的*name*使用Devices文件中的哪一项。*Class*域是指线路速率（波特率）。*Phone*域指定那些*type*为ACU项的电话号码，而其他项的*phone*域是一个连字符。最后一个域 *login*，是一个字符串。它是在UUCP中远程登录时所使用的，这里不使用这个域。

Devices文件包含了调制解调器和那些直接连接的主机的信息。表18-2列出了这个文件中的五个域。*type*域与Systems文件中*type*域对应。*class*域也一定要与Systems文件中对应的*class*域一致，它通常指定了线路速率。

表18-2 Devices 文件

type	line	line	class	dialer
ACU	cua0	-	19200	tbfast
ACU	cua0	-	9600	tb9600
ACU	cua0	-	2400	tb2400
ACU	cua0	-	1200	tb1200
modem	ttya	-	19200	direct
laser	ttyb	-	19200	direct

设备的实际名称是对*line*字段加前缀/dev/。在这个例子中，实际设备是/dev/cua0，/dev/ttya和/dev/ttyb。另外一个域*line2*，没有被使用。

最后一个域*dialer*，与Dialers文件中对应项一致。对于直接相连的项则为direct。

表18-3显示了Dialers文件的格式。这个文件包含了所有调制解调器的拨号命令。

表18-3 Dialers文件

<i>dialer</i>	<i>sub</i>	<i>handshake</i>
tb9600	=W-,	" " \dA\pA\pA\pTQ0S2=255S12=255s50=6s58=2s68=255\r\c OK\r \EATDT\T\r\c CONNECT\s9600 \r\c ""
tbfast	=W-,	" " \dA\pA\pA\pTQ0S2=255S12=255s50=255s58=2s68=255s110=1s111=30\r\c OK\r \EATDT\T\r\c CONNECT\sFAST

表中只列出两项，没有列出 Devices 中的 tb1200 和 tb2400 项。handshake 域本应该写在同一行中，因为版面的限制，我们把它放在两行上。

dialer 域与 Devices 文件中的行相对应。sub 域则指定了电话号码中等号和减号的替代字符。在表 18-3 中，这个域表明了用 w 代替等号，逗号代替减号。这样就允许 Systems 文件中的电话号码中含有等号（意思是等待拨号音）和减号（意思是暂停）。在不同的调制解调器上，这两个字符的含义不同，将它们替换成何种字符，需在 Dialers 文件中指定。

最后一个域 handshake，包含了实际的拨号指令。它是一连串以空格分开的字符串，称为期望-发送串。我们期望（即一直读取，直到得到匹配字符串）得到第一个字符串，然后发送（写入）第二个字符串。作为一个例子，让我们来看 tbfast 项。这个项是用于 PEP (Packetized Ensemble Protocol) 模式的 Telebit Trailblazer 调制解调器。

(1) 第一个期望字符串是空，意思是“期望空”。这总是成功的。

(2) 发送第二个字符串，这个字符串以 \d 开头，\d 表示暂停两秒。然后发送 A。再暂停半秒 (\p)，发送另外一个 A，暂停，再发送一个 A，再暂停。接着，发送余下的以 T 开头的字符串。这些都是设置调制解调器的命令。 \r 发送一个回车， \c 表明在发送字符串结尾不要开始新行。

(3) 从调制解调器读取，直到得到字符串 OK\r (\r 表示回车)。

(4) 下一个发送串以 \E 开头，这允许进行回应检查：每次发送给调制解调器一个字符，就一直读取直到有回应。然后发送四个字符 ATDT。下一个特殊字符 \T，是指使用替代的电话号码。然后是一个回车符，然后是 \c 是指在发送字符串后不要开始新行。

(5) 最后的期望字符串是等待调制解调器返回 CONNECT FAST。(\s 意思是单个空格。)

当收到最后的期望字符串后，拨号就完成了。（当然，在 handshake 字符串中可能出现其他更多的特殊字符序列，这里就不详细说明了。）

现在来总结一下，对这三个文件的操作。

(1) 使用远程系统的名称，在 Systems 文件中找到相同 name 的第一项。

(2) 在 Devices 文件中找到对应的项，其 type 域和 class 域与 Systems 文件中项的相应域匹配。

(3) 在 Dialer 文件中找到与 Devices 文件 dialer 域对应的项。

(4) 拨号。

这个过程如果失败，有两个原因：(1) 对应于 Devices 文件中 line 域的设备已经被其他人所使用，(2) 拨号不成功。（例如，远程系统电话占线，或者远程系统关机不响应电话等）。第二种情况一般可以通过对调制解调器读写超时来确定。（见习题 18.10）。不管出现哪一种情况，都要回到拨号的第(1)步，然后选择 Systems 文件中同一远程系统的下一项。如同我们在表 18-1 中看到的，一个特定的主机可以有多个项，每个主机可以有多个电话号码（同一个电话号码也可以对应多个设备）。

在 Honey DanBer 系统中还有其他我们没有用到的文件。如 Dialcodes 指定了 Systems 文件中电话号码的缩写，而 Sysfiles 文件允许指定 Systems、Devices、Dialers 文件的替代文件。

18.5 服务器设计

现在我们开始描述一下服务器。有两个因素影响服务器的设计。

(1) 拨号过程可能会延续一段时间 (15~30秒), 所以服务器一定要创建一个子进程来处理实际的拨号。

(2) 服务器的精灵进程 (父进程) 一定要管理所有的加锁。

图18-2显示了这个过程。

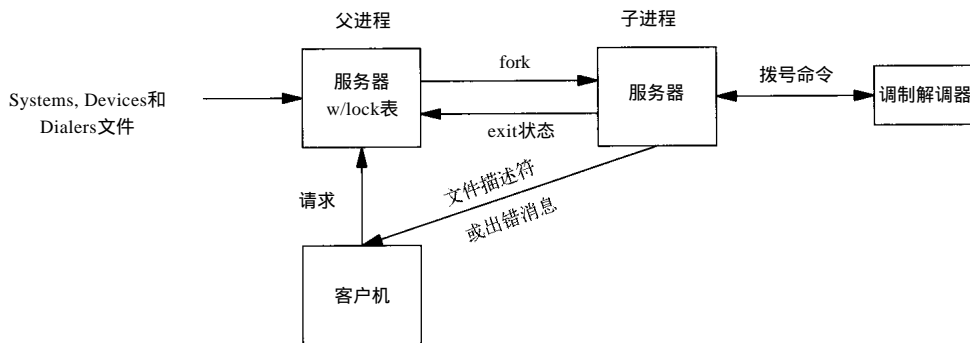


图18-2 调制解调器拨号器的工作过程

服务器的工作过程如下：

(1) 父进程在服务器的众所周知名字处接收从客户机发来的请求。如 15.5节所述, 这在客户机-服务器之间生成了一个流管道。父进程就像 15.6节中的open服务器一样, 要同时处理多个客户机。

(2) 基于客户机要联系的远程系统的名字, 父进程查询Systems文件和Devices文件找到匹配的项。父进程同时也维护一个加锁表, 记录哪些设备在被使用, 这样它就不查询那些被使用的项了。

(3) 如果发现匹配项, 则fork一个子进程来进行实际的拨号。(父进程这时可以处理其他客户机请求)。如果成功, 子进程就在客户机指定的流管道上将调制解调器的文件描述符传给客户机 (这个管道在fork时也被复制了), 并调用exit(0)。如果发生了错误 (例如, 电话线占线、没有响应等), 子进程调用exit(1)。

(4) 子进程结束时, 会发送信号 SIGCHLD通知父进程。父进程就得到子进程的结束状态 (waitpid)。

如果子进程成功, 父进程就不用再做其他事情。在客户机结束使用调制解调器之前, 必须一直对调制解调器加锁。客户机指定的客户机-父进程之间的流管道就一直打开着。这样, 当客户机终止时, 父进程得到通知, 然后释放对设备的加锁。

如果子进程不成功, 父进程就从Systems文件中尝试找下一个匹配项。如果找到了对远程系统的另一项, 父进程返回上一步, 创建一个新的子进程来拨号。如果没有找到新的匹配项, 父进程调用send_err (见程序15-4) 后关闭与客户机的流管道。

与每一个客户机有一个连接使子进程在必要时能将调试输出发回给客户机。发生问题时, 客户机常常想要看到整个实际拨号过程。

18.6 服务器源码

服务器包括17个源文件。表18-4详细说明了父进程和子进程所使用的文件, 以及这些文件

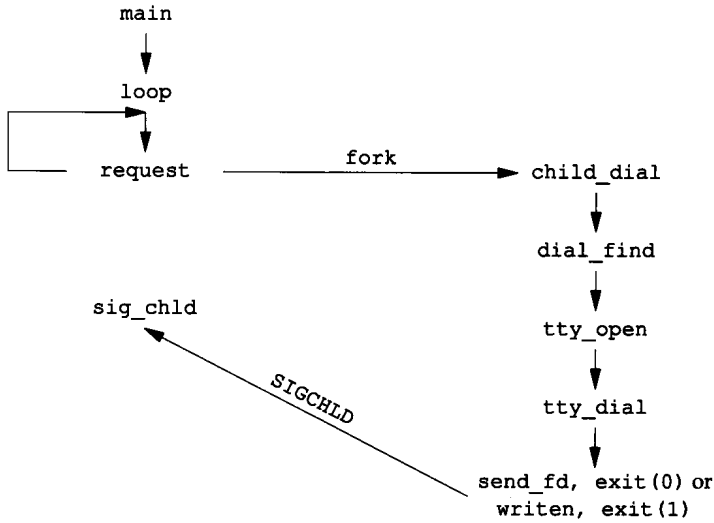


图18-3 服务器的函数调用过程

中所包含的函数。图18-3描述了不同函数的调用过程。

表18-4 服务器源文件

源文件	父进程/ 子进程	函 数
childdial.c	C	child_dial
cliargs.c	P	cli_args
client.c	P	client_alloc, client_add, client_del, client_sigchld
ctlstr.c	C	ctl_str
debug.c	C	DEBUG, DEBUG_NONL
devfile.c	P	dev_next, dev_rew, dev_find
dialfile.c	C	dial_next, dial_rew, dial_find
expectstr.c	C	expect_str, exp_read, sig_alm
lock.c	P	find_line, lock_set, lock_rel, is_locked
loop.c	P	loop, cli_done, child_done
main.c	P	main
request.c	P	request
sendstr.c	C	send_str
sigchld.c	P	sig_chld
sysfile.c	P	sys_next, sys_rew, sys_posn
ttydial.c	C	tty_dial
ttyopen.c	C	tty_open

程序18-1是calld.h头文件，它被包含在所有这些源文件中。calld.h包含几个系统头文件，定义了一些基本的常量，声明了全局变量。

程序18-1 calld.h头文件

```

#include <sys/types.h>
#include <errno.h>
#include <signal.h>
#include "ourhdr.h"

#define CS_CALL "/home/stevens/calld" /* well-known name */
#define CL_CALL "call"
#define MAXSYSNAME 256
#define MAXSPEEDSTR 256
  
```



```

#define NALLOC 10          /* #structs to alloc/realloc for */
                          /* Client structs (client.c), Lock structs (lock.c) */
#define WHITE " \t\n"     /* for separating tokens */
#define SYSTEMS "./Systems" /* my own copies for now */
#define DEVICES "./Devices"
#define DIALERS  "./Dialers"

/* declare global variables */
extern int clifd;
extern int debug; /* nonzero if interactive (not daemon) */
extern int Debug; /* nonzero for dialing debug output */
extern char errmsg[]; /* error message string to return to client */
extern char *speed; /* speed (actually "class") to use */
extern char *sysname; /* name of system to call */
extern uid_t uid; /* client's uid */
extern volatile sig_atomic_t chld_flag; /* when SIGCHLD occurs */
extern enum parity { NONE, EVEN, ODD } parity; /* specified by client */

typedef struct { /* one Client struct per connected client */
    int fd; /* fd, or -1 if available */
    pid_t pid; /* child pid while dialing */
    uid_t uid; /* client's user ID */
    int chlldone; /* nonzero when SIGCHLD from dialing child recvd:
                  1 means exit(0), 2 means exit(1) */
    long sysftell; /* next line to read in Systems file */
    long foundone; /* true if we find a matching sysfile entry */
    int Debug; /* option from client */
    enum parity parity; /* option from client */
    char speed[MAXSPEEDSTR]; /* option from client */
    char sysname[MAXSYSNAME]; /* option from client */
} Client;

extern Client *client; /* ptr to malloc'ed array of Client structs */
extern int client_size; /* # entries in client[] array */
/* (both manipulated by client_XXX() functions) */

typedef struct { /* everything for one entry in Systems file */
    char *name; /* system name */
    char *time; /* (e.g., "Any") time to call (ignored) */
    char *type; /* (e.g., "ACU") or system name if direct connect */
    char *class; /* (e.g., "9600") speed */
    char *phone; /* phone number or "-" if direct connect */
    char *login; /* uucp login chat (ignored) */
} Systems;

typedef struct { /* everything for one entry in Devices file */
    char *type; /* (e.g., "ACU") matched by type in Systems */
    char *line; /* (e.g., "cua0") without preceding "/dev/" */
    char *line2; /* (ignored) */
    char *class; /* matched by class in Systems */
    char *dialer; /* name of dialer in Dialers */
} Devices;

typedef struct { /* everything for one entry in Dialers file */
    char *dialer; /* matched by dialer in Devices */
    char *sub; /* phone number substitution string (ignored) */
    char *expsend; /* expect/send chat */
} Dialers;

extern Systems systems; /* filled in by sys_next() */
extern Devices devices; /* filled in by dev_next() */
extern Dialers dialers; /* filled in by dial_next() */

/* our function prototypes */
void child_dial(Client *); /* chlldial.c */

```

```

int      cli_args(int, char **);           /* cliargs.c */
int      client_add(int, uid_t);          /* client.c */
void     client_del(int);
void     client_sigchild(pid_t, int);

void     loop(void);                      /* loop.c */

char     *ctl_str(char);                  /* ctlstr.c */

int      dev_find(Devices *, const Systems *); /* devfile.c */
int      dev_next(Devices *);
void     dev_rew(void);

int      dial_find(Dialers *, const Devices *); /* dialfile.c */
int      dial_next(Dialers *);
void     dial_rew(void);

int      expect_str(int, char *);         /* expectstr.c */

int      request(Client *);               /* request.c */

int      send_str(int, char *, char *, int); /* sendstr.c */

void     sig_chld(int);                   /* sigchild.c */

long     sys_next(Systems *);              /* sysfile.c */
void     sys_posn(long);
void     sys_rew(void);

int      tty_open(char *, char *, enum parity, int); /* ttyopen.c */

int      tty_dial(int, char *, char *, char *, char *); /* ttydial.c */

pid_t    is_locked(char *);               /* lock.c */
void     lock_set(char *, pid_t);
void     lock_rel(pid_t);

void     DEBUG(char *, ...);              /* debug.c */
void     DEBUG_NONL(char *, ...);

```

我们定义了一个Client结构，它包含了每一客户机的所有信息。这是一个对程序 15-26中类似结构的扩展。在创建一个子进程为客户机拨号和子进程终止之间，可以处理任一多的其他客户机。这个结构同时包含了我们所需要的其他信息，如尝试找到 Systems文件中的其他项，重新拨号等。

我们同样为Systems、Devices、Dialers文件中每一项定义了一个结构。

程序18-2是服务器的main函数。因为这个程序一般是作为精灵进程运行，故提供了一个 -d 的命令行选择项，允许交互式运行。

程序18-2 main函数

```

#include    "calld.h"
#include    <syslog.h>

          /* define global variables */
int       clifd;
int       debug; /* daemon's command line flag */
int       Debug; /* Debug controlled by client, not cmd line */
char      errmsg[MAXLINE];
char      *speed;
char      *sysname;
uid_t     uid;
Client    *client = NULL;

```

```

int      client_size;
Systems  systems;
Devices  devices;
Dialers  dialers;
volatile sig_atomic_t chld_flag;
enum parity parity = NONE;

int
main(int argc, char *argv[])
{
    int    c;

    log_open("calld", LOG_PID, LOG_USER);
    opterr = 0;      /* don't want getopt() writing to stderr */
    while ( (c = getopt(argc, argv, "d")) != EOF) {
        switch (c) {
            case 'd':      /* debug */
                debug = 1;
                break;

            case '?':
                log_quit("unrecognized option: -%c", optopt);
        }
    }

    if (debug == 0)
        daemon_init();

    loop();      /* never returns */
}

```

当使用-d选择项后，所有对log_XXX函数的调用(见附录B)都送到标准错误。否则它们就会被syslog记录下来。

函数loop是服务器程序的主循环(见程序18-3)。它使用了select函数来复制不同的描述符。

程序18-3 loop.c文件

```

#include    "calld.h"
#include    <sys/time.h>
#include    <errno.h>

static void cli_done(int);
static void child_done(int);

static fd_set  allset; /* one bit per client conn, plus one for listenfd */
                /* modified by loop() and cli_done() */

void
loop(void)
{
    int      i, n, maxfd, maxi, listenfd, nread;
    char     buf[MAXLINE];
    Client   *cliptr;
    uid_t    uid;
    fd_set   rset;

    if (signal_intr(SIGCHLD, sig_chld) == SIG_ERR)
        log_sys("signal error");

    /* obtain descriptor to listen for client requests on */
    if ( (listenfd = serv_listen(CS_CALL)) < 0)
        log_sys("serv_listen error");
}

```

```

FD_ZERO(&allset);
FD_SET(listenfd, &allset);
maxfd = listenfd;
maxi = -1;

for ( ; ; ) {
    if (chld_flag)
        child_done(maxi);
    rset = allset; /* rset gets modified each time around */
    if ( (n = select(maxfd + 1, &rset, NULL, NULL, NULL)) < 0) {
        if (errno == EINTR) {
            /* caught SIGCHLD, find entry with chlddone set */
            child_done(maxi);
            continue; /* issue the select again */
        } else
            log_sys("select error");
    }

    if (FD_ISSET(listenfd, &rset)) {
        /* accept new client request */
        if ( (clifd = serv_accept(listenfd, &uid)) < 0)
            log_sys("serv_accept error: %d", clifd);

        i = client_add(clifd, uid);
        FD_SET(clifd, &allset);
        if (clifd > maxfd)
            maxfd = clifd; /* max fd for select() */
        if (i > maxi)
            maxi = i; /* max index in client[] array */
        log_msg("new connection: uid %d, fd %d", uid, clifd);
        continue;
    }

    /* Go through client[] array.
       Read any client data that has arrived. */

    for (cliptr = &client[0]; cliptr <= &client[maxi]; cliptr++) {
        if ( (clifd = cliptr->fd) < 0)
            continue;
        if (FD_ISSET(clifd, &rset)) {
            /* read argument buffer from client */
            if ( (nread = read(clifd, buf, MAXLINE)) < 0)
                log_sys("read error on fd %d", clifd);

            else if (nread == 0) {
                /* The client has terminated or closed the stream
                   pipe. Now we can release its device lock. */

                log_msg("closed: uid %d, fd %d",
                        cliptr->uid, clifd);
                lock_rel(cliptr->pid);
                cli_done(clifd);
                continue;
            }

            /* Data has arrived from the client. Process the
               client's request. */

            if (buf[nread-1] != 0) {
                log_quit("request from uid %d not null terminated:"
                        " %*.s", uid, nread, nread, buf);
                cli_done(clifd);
                continue;
            }
        }
        log_msg("starting: %s, from uid %d", buf, uid);
    }
}

```

```

        /* Parse the arguments, set options. Since
           we may need to try calling again for this
           client, save options in client[] array. */
        if (buf_args(buf, cli_args) < 0)
            log_quit("command line error: %s", buf);
        cliptr->Debug = Debug;
        cliptr->parity = parity;
        strcpy(cliptr->sysname, sysname);
        strcpy(cliptr->speed, (speed == NULL) ? "" : speed);
        cliptr->childdone = 0;
        cliptr->sysftell = 0;
        cliptr->foundone = 0;

        if (request(cliptr) < 0) {
            /* system not found, or unable to connect */
            if (send_err(cliptr->fd, -1, errmsg) < 0)
                log_sys("send_err error");
            cli_done(cliid);
            continue;
        }
        /* At this point request() has forked a child that is
           trying to dial the remote system. We'll find
           out the child's status when it terminates. */
    }
}

}

}

/* Go through the client[] array looking for clients whose dialing
   children have terminated. This function is called by loop() when
   chld_flag (the flag set by the SIGCHLD handler) is nonzero. */

static void
child_done(int maxi)
{
    Client *cliptr;

again:
    chld_flag = 0; /* to check when done with loop for more SIGCHLDs */
    for (cliptr = &client[0]; cliptr <= &client[maxi]; cliptr++) {
        if ( ( cliid = cliptr->fd) < 0)
            continue;
        if (cliptr->childdone) {
            log_msg("child done: pid %d, status %d",
                cliptr->pid, cliptr->childdone-1);

            /* If the child was successful (exit(0)), just clear
               the flag. When the client terminates, we'll read
               the EOF on the stream pipe above and release
               the device lock. */

            if (cliptr->childdone == 1) { /* child did exit(0) */
                cliptr->childdone = 0;
                continue;
            }

            /* Unsuccessful: child did exit(1). Release the device
               lock and try again from where we left off. */

            cliptr->childdone = 0;
            lock_rel(cliptr->pid); /* unlock the device entry */
            if (request(cliptr) < 0) {
                /* still unable, time to give up */
                if (send_err(cliptr->fd, -1, errmsg) < 0)

```

```

        log_sys("send_err error");
        cli_done(clifd);
        continue;
    }
    /* request() has forked another child for this client */
}
}
if (chld_flag) /* additional SIGCHLDs have been caught */
    goto again; /* need to check all chlldone flags again */
}

/* Clean up when we're done with a client. */

static void
cli_done(int clifd)
{
    client_del(clifd); /* delete entry in client[] array */
    FD_CLR(clifd, &allset); /* turn off bit in select() set */
    close(clifd); /* close our end of stream pipe */
}

```

loop函数初始化client数组，建立一个对SIGCHLD信号的处理程序。我们不调用signal，而调用了signal_intr，这样当有信号返回时，每一个慢速的系统调用都可以被中断。然后 loop函数调用serv_listen（见程序15-19和程序15-22）。loop函数的其他部分是一个基于select函数的无限循环，它检查如下两种情况：

(1) 如果传来一个新的客户机连接请求，则调用 serv_accept（见程序15-20和15-24）。函数 client_add为这个新的客户机在client数组中增加一项。

(2) 浏览整个client数组，看是否有客户机终止或者新的客户机请求到达。

当一个客户机终止拨号时（不管是否自愿），它的客户机专用的通向服务器的流管道被关闭，我们从管道上得到一个文件终止符。这时，可以释放这个客户机所拥有的全部加锁，并删除client数组中的项。

当从客户机收到请求时，则调用 request函数（函数buf_args见程序15-17）。如果客户机调用的远程系统的名字有效，而且找到相对应的 Devices项，request函数就会创建一个子进程，然后返回。

在loop函数运行中，子进程终止这一外部事件随时都可能发生。如果在 select函数中阻塞，则select函数返回一个EINTR错误。因为在loop函数的其他地方也可能出现子进程终止信号，所以在这个循环中每次调用select之前都检测标志chld_flag，如果有，则调用child_done来处理子进程终止。

loop函数浏览整个client数组，检查每一项的chlldone标志。如果子进程成功，就不需要做其他事情。但如果子进程以状态1终止时（exit(1)），则调用request函数来尝试使用Systems文件中的下一项。

程序18-4是cli_args，这个函数在客户机请求到达时被loop函数中的buf_args所调用。它处理客户机送来的命令行参数。注意这个函数根据命令行参数设置全局变量，然后loop函数将这些全局变量复制到client数组的相应的项中，这些选择项只影响一个客户机请求。

程序18-5是client.c，它定义了处理client数组的函数组。程序18-5和程序15-27的区别在于在这里一定要根据进程的ID（见函数client_sigchld）来查询所需要的项。

程序18-6是文件lock.c。其中的函数管理父进程中的lock数组。同上面的client数组一样，调用realloc来给lock数组动态分配空间，以避免编译时间限制。

程序18-4 cli_args函数

```
#include "calld.h"
/* This function is called by buf_args(), which is called by loop().
 * buf_args() has broken up the client's buffer into an argv[] style
 * array, which is now processed. */
int
cli_args(int argc, char **argv)
{
    int    c;
    if (argc < 2 || strcmp(argv[0], CL_CALL) != 0) {
        strcpy(errmsg, "usage: call <options> <hostname>");
        return(-1);
    }
    Debug = 0;      /* option defaults */
    parity = NONE;
    speed = NULL;
    opterr = 0;     /* don't want getopt() writing to stderr */
    optind = 1;    /* since we call getopt() multiple times */
    while ( (c = getopt(argc, argv, "des:o")) != EOF) {
        switch (c) {
            case 'd':
                Debug = 1; /* client wants DEBUG() output */
                break;
            case 'e':      /* even parity */
                parity = EVEN;
                break;
            case 'o':      /* odd parity */
                parity = ODD;
                break;
            case 's':      /* speed */
                speed = optarg;
                break;
            case '?':
                sprintf(errmsg, "unrecognized option: -%c\n", optopt);
                return(-1);
        }
    }
    if (optind < argc)
        sysname = argv[optind]; /* name of host to call */
    else {
        sprintf(errmsg, "missing <hostname> to call\n");
        return(-1);
    }
    return(0);
}
```

程序18-5 client.c程序

```
#include "calld.h"
static void
client_alloc(void) /* alloc more entries in the client[] array */
{
    int    i;
    if (client == NULL)
        client = malloc(NALLOC * sizeof(Client));
    else
```

```

    client = realloc(client, (client_size + NALLOC) * sizeof(Client));
    if (client == NULL)
        err_sys("can't alloc for client array");

        /* have to initialize the new entries */
    for (i = client_size; i < client_size + NALLOC; i++)
        client[i].fd = -1; /* fd of -1 means entry available */
    client_size += NALLOC;
}

/* Called by loop() when connection request from a new client arrives */
int
client_add(int fd, uid_t uid)
{
    int    i;

    if (client == NULL) /* first time we're called */
        client_alloc();
again:
    for (i = 0; i < client_size; i++) {
        if (client[i].fd == -1) { /* find an available entry */
            client[i].fd = fd;
            client[i].uid = uid;
            return(i); /* return index in client[] array */
        }
        /* client array full, time to realloc for more */
        client_alloc();
        goto again; /* and search again (will work this time) */
    }
}

/* Called by loop() when we're done with a client */
void
client_del(int fd)
{
    int    i;

    for (i = 0; i < client_size; i++) {
        if (client[i].fd == fd) {
            client[i].fd = -1;
            return;
        }
    }
    log_quit("can't find client entry for fd %d", fd);
}

/* Find the client entry corresponding to a process ID.
 * This function is called by the sig_chld() signal
 * handler only after a child has terminated. */
void
client_sigchld(pid_t pid, int stat)
{
    int    i;

    for (i = 0; i < client_size; i++) {
        if (client[i].pid == pid) {
            client[i].childdone = stat; /* child's exit() status +1 */
            return;
        }
    }
    log_quit("can't find client entry for pid %d", pid);
}

```


程序18-6 管理设备锁的函数组

```
#include "calld.h"

typedef struct {
    char *line; /* points to malloc()'ed area */
              /* we lock by line (device name) */
    pid_t pid; /* but unlock by process ID */
              /* pid of 0 means available */
} Lock;
static Lock *lock = NULL; /* the malloc'ed/realloc'ed array */
static int lock_size; /* #entries in lock[] */
static int nlocks; /* #entries currently used in lock[] */

/* Find the entry in lock[] for the specified device (line).
 * If we don't find it, create a new entry at the end of the
 * lock[] array for the new device. This is how all the possible
 * devices get added to the lock[] array over time. */

static Lock *
find_line(char *line)
{
    int i;
    Lock *lptr;

    for (i = 0; i < nlocks; i++) {
        if (strcmp(line, lock[i].line) == 0)
            return(&lock[i]); /* found entry for device */
    }

    /* Entry not found. This device has never been locked before.
     Add a new entry to lock[] array. */

    if (nlocks >= lock_size) { /* lock[] array is full */
        if (lock == NULL) /* first time through */
            lock = malloc(NALLOC * sizeof(Lock));
        else
            lock = realloc(lock, (lock_size + NALLOC) * sizeof(Lock));
        if (lock == NULL)
            err_sys("can't alloc for lock array");

        lock_size += NALLOC;
    }

    lptr = &lock[nlocks++];
    if ((lptr->line = malloc(strlen(line) + 1)) == NULL)
        log_sys("malloc error");
    strcpy(lptr->line, line); /* copy caller's line name */
    lptr->pid = 0;
    return(lptr);
}

void
lock_set(char *line, pid_t pid)
{
    Lock *lptr;

    log_msg("locking %s for pid %d", line, pid);
    lptr = find_line(line);
    lptr->pid = pid;
}

void
lock_rel(pid_t pid)
{
    Lock *lptr;
```

```

    for (lptr = &lock[0]; lptr < &lock[nlocks]; lptr++) {
        if (lptr->pid == pid) {
            log_msg("unlocking %s for pid %d", lptr->line, pid);
            lptr->pid = 0;
            return;
        }
    }
    log_msg("can't find lock for pid = %d", pid);
}

pid_t
is_locked(char *line)
{
    return( find_line(line)->pid ); /* nonzero pid means locked */
}

```

lock数组中的每一项都与一个line (Devices文件的第二个域) 相关联。因为这些加锁函数不知道该数据文件中所有的line值，所以每当一个新line被第一次加锁时，就在lock数组中增加一项。函数find_line处理加锁。

下面的三个源文件处理三个数据文件：Systems, Devices和Dialers。每个文件有个XXX_next函数，它读取文件中的下一行，调用ANSI C函数strtok把这一行分成多个域。程序18-7处理Systems文件。

程序18-7 读取Systems文件的函数

```

#include    "callid.h"

static FILE *fpsys = NULL;
static int  syslineno; /* for error messages */
static char sysline[MAXLINE];
           /* can't be automatic; sys_next() returns pointers into here */

/* Read and break apart a line in the Systems file. */

long
sys_next(Systems *sysptr) /* return >0 if OK, -1 on EOF */
/* structure is filled in with pointers */
{
    if (fpsys == NULL) {
        if ( (fpsys = fopen(SYSTEMS, "r")) == NULL)
            log_sys("can't open %s", SYSTEMS);
        syslineno = 0;
    }
again:
    if (fgets(sysline, MAXLINE, fpsys) == NULL)
        return(-1); /* EOF */
    syslineno++;

    if ( (sysptr->name = strtok(sysline, WHITE)) == NULL) {
        if (sysline[0] == '\n')
            goto again; /* ignore empty line */
        log_quit("missing 'name' in Systems file, line %d", syslineno);
    }
    if (sysptr->name[0] == '#')
        goto again; /* ignore comment line */

    if ( (sysptr->time = strtok(NULL, WHITE)) == NULL)
        log_quit("missing 'time' in Systems file, line %d", syslineno);

    if ( (sysptr->type = strtok(NULL, WHITE)) == NULL)
        log_quit("missing 'type' in Systems file, line %d", syslineno);
}

```

```

if ( (sysptr->class = strtok(NULL, WHITE)) == NULL)
    log_quit("missing 'class' in Systems file, line %d", syslineno);
if ( (sysptr->phone = strtok(NULL, WHITE)) == NULL)
    log_quit("missing 'phone' in Systems file, line %d", syslineno);
if ( (sysptr->login = strtok(NULL, "\n")) == NULL)
    log_quit("missing 'login' in Systems file, line %d", syslineno);
return(ftell(fpsys)); /* return the position in Systems file */
}

void
sys_rew(void)
{
    if (fpsys != NULL)
        rewind(fpsys);
    syslineno = 0;
}

void
sys_posn(long posn) /* position Systems file */
{
    if (posn == 0)
        sys_rew();
    else if (fseek(fpsys, posn, SEEK_SET) != 0)
        log_sys("fseek error");
}

```

函数sys_next由request调用，其功能是读取Systems文件的下一项。

对于每一个客户机，必须记住当前位置（Client结构的sysftell成员变量）。这样如果一个子进程拨号远程系统没有成功，则可以知道是Systems文件中的哪一项失败，然后尝试其他项。这个位置可通过调用标准的I/O函数ftell得到，可以用fseek函数复位。

程序18-8包含了读取Devices文件的函数。

程序18-8 读取Devices文件的函数

```

#include "calld.h"

static FILE *fpdev = NULL;
static int devlineno; /* for error messages */
static char devline[MAXLINE];
/* can't be automatic; dev_next() returns pointers into here */

/* Read and break apart a line in the Devices file. */

int
dev_next(Devices *devptr) /* pointers in structure are filled in */
{
    if (fpdev == NULL) {
        if ( (fpdev = fopen(DEVICES, "r")) == NULL)
            log_sys("can't open %s", DEVICES);
        devlineno = 0;
    }
}

again:
if (fgets(devline, MAXLINE, fpdev) == NULL)
    return(-1); /* EOF */
devlineno++;

if ( (devptr->type = strtok(devline, WHITE)) == NULL) {
    if (devline[0] == '\n')
        goto again; /* ignore empty line */
    log_quit("missing 'type' in Devices file, line %d", devlineno);
}

```

```

    }
    if (devptr->type[0] == '#')
        goto again;          /* ignore comment line */

    if ( (devptr->line = strtok(NULL, WHITE)) == NULL)
        log_quit("missing 'line' in Devices file, line %d", devlineno);

    if ( (devptr->line2 = strtok(NULL, WHITE)) == NULL)
        log_quit("missing 'line2' in Devices file, line %d", devlineno);

    if ( (devptr->class = strtok(NULL, WHITE)) == NULL)
        log_quit("missing 'class' in Devices file, line %d", devlineno);

    if ( (devptr->dialer = strtok(NULL, WHITE)) == NULL)
        log_quit("missing 'dialer' in Devices file, line %d", devlineno);

    return(0);
}

void
dev_rew(void)
{
    if (fpdev != NULL)
        rewind(fpdev);
    devlineno = 0;
}

/* Find a match of type and class */

int
dev_find(Devices *devptr, const Systems *sysptr)
{
    dev_rew();
    while (dev_next(devptr) >= 0) {
        if (strcmp(sysptr->type, devptr->type) == 0 &&
            strcmp(sysptr->class, devptr->class) == 0)
            return(0);          /* found a device match */
    }
    sprintf(errmsg, "device '%s'/'%s' not found\n",
            sysptr->type, sysptr->class);
    return(-1);
}

```

可以看到，request函数调用dev_find函数来确定type域、class域与Systems文件中对应域相同的项。

程序18-9是读取Dialers文件的函数。

程序18-9 读取Dialers文件的函数

```

#include    "calld.h"

static FILE *fpdial = NULL;
static int  diallineno;          /* for error messages */
static char dialline[MAXLINE];
        /* can't be automatic; dial_next() returns pointers into here */

/* Read and break apart a line in the Dialers file. */

int
dial_next(Dialers *dialptr) /* pointers in structure are filled in */
{
    if (fpdial == NULL) {
        if ( (fpdial = fopen(DIALERS, "r")) == NULL)
            log_sys("can't open %s", DIALERS);
    }
}

```

```

        diallineno = 0;
    }
again:
    if (fgets(dialline, MAXLINE, fpdial) == NULL)
        return(-1);    /* EOF */
    diallineno++;

    if ( (dialptr->dialer = strtok(dialline, WHITE)) == NULL) {
        if (dialline[0] == '\n')
            goto again;    /* ignore empty line */
        log_quit("missing 'dialer' in Dialers file, line %d", diallineno)
    }
    if (dialptr->dialer[0] == '#')
        goto again;    /* ignore comment line */

    if ( (dialptr->sub = strtok(NULL, WHITE)) == NULL)
        log_quit("missing 'sub' in Dialers file, line %d", diallineno);

    if ( (dialptr->expsend = strtok(NULL, "\n")) == NULL)
        log_quit("missing 'expsend' in Dialers file, line %d",diallineno);

    return(0);
}

void
dial_rew(void)
{
    if (fpdial != NULL)
        rewind(fpdial);
    diallineno = 0;
}

/* Find a dialer match */

int
dial_find(Dialers *dialptr, const Devices *devptr)
{
    dial_rew();
    while (dial_next(dialptr) >= 0) {
        if (strcmp(dialptr->dialer, devptr->dialer) == 0)
            return(0);    /* found a dialer match */
    }
    sprintf(errmsg, "dialer '%s' not found\n", dialptr->dialer);
    return(-1);
}

```

可以看到函数child_dial调用dial_find函数来查找dialer域与一个特定设备匹配的项。

从表18-4可以发现Systems和Devices文件由父进程处理，而子进程处理Dialers文件。这也是整个设计的目的——父进程发现一个匹配的未被加锁的设备，然后创建一个子进程进行实际的拨号工作。

现在看一下程序18-10中的request函数。loop函数调用此函数，来确定对应一个特定的远程主机的而且未被加锁的拨号设备。为了达到这个目的，loop函数查找Systems和Devices文件。如果发现匹配项，则创建一个子进程。除了远程主机名字，还允许客户机指定拨号速度。例如，对于表18-1中的Systems文件，客户机的拨号请求可能是这样的：

```
call -s 9600 host1
```

这样就忽略了表18-1中速率不是9600的host1项。

需要注意的是，除非知道了子进程的ID，否则不能使用lock_set函数来记录设备锁，但是在创建子进程之前又必须检查设备是否被加锁。因为总是要先加锁，然后才启动子进程拨号，

因此使用TELL_WAIT函数（见程序10-17）来同步父进程和子进程。同样需要注意：虽然检查函数is_locked和实际的加锁函数set_lock是两个分立的操作（也就是说，不是一个原子操作），但是这并没有竞争的问题，这是因为request函数只能被一个服务器精灵进程所调用——它不能被多个进程调用。

如果request返回0，则创建一个子进程来开始拨号。如果request返回1时，则说明或者远程系统的名称无效，或者所有可能的设备都已经被加锁。

程序18-10 request函数

```
#include "call.h"

int request(Client *cliptr) /* return 0 if OK, -1 on error */
{
    pid_t pid;
    errmsg[0] = 0;
    /* position where this client left off last (or rewind) */
    sys_posn(cliptr->sysftell);
    while ( (cliptr->sysftell = sys_next(&systems)) >= 0) {
        if (strcmp(cliptr->sysname, systems.name) == 0) {
            /* system match */
            /* if client specified a speed, it must match too */
            if (cliptr->speed[0] != 0 &&
                strcmp(cliptr->speed, systems.class) != 0)
                continue; /* speeds don't match */

            DEBUG("trying sys: %s, %s, %s, %s", systems.name,
                  systems.type, systems.class, systems.phone);
            cliptr->foundone++;

            if (dev_find(&devices, &systems) < 0)
                break;
            DEBUG("trying dev: %s, %s, %s, %s", devices.type,
                  devices.line, devices.class, devices.dialer);
            if ( (pid = is_locked(devices.line)) != 0) {
                sprintf(errmsg, "device '%s' already locked by pid %d\n",
                        devices.line, pid);
                continue; /* look for another entry in Systems file */
            }

            /* We've found a device that's not locked.
               fork() a child to do the actual dialing. */
            TELL_WAIT();
            if ( (cliptr->pid = fork()) < 0)
                log_sys("fork error");
            else if (cliptr->pid == 0) { /* child */
                WAIT_PARENT(); /* let parent set lock */
                child_dial(cliptr); /* never returns */
            }
            /* parent */
            lock_set(devices.line, cliptr->pid);
            /* let child resume, now that lock is set */
            TELL_CHILD(cliptr->pid);
            return(0); /* we've started a child */
        }
    }
    /* reached EOF on Systems file */
    if (cliptr->foundone == 0)
        sprintf(errmsg, "system '%s' not found\n", cliptr->sysname);
}
```

```

else if (errmsg[0] == 0)
    sprintf(errmsg, "unable to connect to system '%s'\n",
            cliptr->sysname);
return(-1);    /* also, cliptr->sysftell is -1 */
}

```

父进程专用函数的最后一个是 sig_chld，它是 SIGCHLD 信号的处理程序。这个函数在程序 18-11 中。

程序 18-11 sig_chld 信号处理程序

```

#include    "callld.h"
#include    <sys/wait.h>

/* SIGCHLD handler, invoked when a child terminates. */

void
sig_chld(int signo)
{
    int    stat, errno_save;
    pid_t    pid;

    errno_save = errno;    /* log_msg() might change errno */
    chld_flag = 1;
    if ( (pid = waitpid(-1, &stat, 0)) <= 0)
        log_sys("waitpid error");

    if (WIFEXITED(stat) != 0)
        /* set client's chlddone status for loop() */
        client_sigchld(pid, WEXITSTATUS(stat)+1);
    else
        log_msg("child %d terminated abnormally: %04x", pid, stat);

    errno = errno_save;
    return;    /* probably interrupts accept() in serv_accept() */
}

```

当一个子进程终止时，必须在 client 数组的适当位置上记录下它的终止状态和进程 ID。调用函数 client_sigchld（见程序 18-5）来完成这个工作。

需要注意，这里违反了第 10 章中的原则——一个信号处理程序只处理一个全局变量，而不做其他。这里调用了 waitpid 和函数 client_sigchld（见程序 18-5），后者对信号而言是安全的，它所做的只是在 client 数组的对应项中做记录。它并不创建或删除项（那样的话将是不可重入的），它也不调用任何系统函数。

POSIX.1 定义的 waitpid 对信号而言是安全的（见表 10-3）。如果不从信号处理程序中调用 waitpid，那么当 chld_flag 标志非 0 时父进程必须调用 waitpid。但是因为主循环查询 chld_flag 之前子进程有可能终止，则要或者在每个子进程终止时对 chld_flag 加 1（这样主循环就知道要调用多少次 waitpid），或者在循环内使用 WNOHANG 标志，并调用 waitpid（见表 8-3）。最简单的方法还是从信号量处理程序中调用 waitpid，在 client 数组中记录下信息。

现在讨论客户机在拨号远程系统时需调用的一些函数。当 request 调用 child_dial 来创建子进程（见程序 18-12）时，会用到这些函数。

程序 18-12 child_dial 函数

```

#include    "callld.h"

```

```

/* The child does the actual dialing and sends the fd back to
 * the client. This function can't return to caller, must exit.
 * If successful, exit(0), else exit(1).
 * The child uses the following global variables, which are just
 * in the copy of the data space from the parent:
 *     cliptr->fd (to send DEBUG() output and fd back to client),
 *     cliptr->Debug (for all DEBUG() output), cliptr->parity,
 *     systems, devices, dialers. */

void
child_dial(Client *cliptr)
{
    int    fd, n;

    Debug = cliptr->Debug;
    DEBUG("child, pid %d", getpid());

    if (strcmp(devices.dialer, "direct") == 0) { /* direct tty line */
        fd = tty_open(systems.class, devices.line, cliptr->parity, 0);
        if (fd < 0)
            goto die;
    } else { /* else assume dialing is needed */
        if (dial_find(&dialers, &devices) < 0)
            goto die;
        fd = tty_open(systems.class, devices.line, cliptr->parity, 1);
        if (fd < 0)
            goto die;
        if (tty_dial(fd, systems.phone, dialers.dialer,
                    dialers.sub, dialers.expsend) < 0)
            goto die;
    }

    DEBUG("done");
    /* send the open descriptor to client */
    if (send_fd(cliptr->fd, fd) < 0)
        log_sys("send_fd error");
    exit(0); /* parent will see this */

die:
    /* The child can't call send_err() as that would send the final
     * 2-byte protocol to the client. We just send our error message
     * back to the client. If the parent finally gives up, it'll
     * call send_err(). */
    n = strlen(errmsg);
    if (writen(cliptr->fd, errmsg, n) != n) /* send error to client */
        log_sys("writen error");
    exit(1); /* parent will see this, release lock, and try again */
}

```

如果所使用的设备是直接相连的，那么只要调用 `tty_open` 来打开终端设备并设置终端设备参数。但如果这个设备是调制解调器，那么就要使用三个函数：`dial_find`（确定 `Dialers` 文件中的符合要求的项）、`tty_open` 和 `tty_dial`（进行实际的拨号）。

如果 `child_dial` 成功了，它调用 `send_fd`（见程序 15-5 和程序 15-9）把文件描述符传给客户机，并调用 `exit(0)`。否则，它将出错消息传给客户机，并调用 `exit(1)`。客户机专用的流管道在创建时被复制，所以子进程可以直接将文件描述符或出错消息传给客户机。

客户机可以在发送给服务器的命令中使用 `-d` 选择项，与此对应，设置客户机专用标志变量 `Debug`。程序 18-13 的 `DEBUG` 和 `DEBUG_NONL` 函数使用了此标志，将调试信息传回客户机。当拨号遇到问题时，这个调试信息就有用了。这两个函数主要被子进程所调用，父进程也会在 `request` 函数中调用它们（见程序 18-10）。

程序18-13 调试函数

```
#include "callid.h"
#include <stdarg.h>

/* Note that all debug output goes back to the client. */

void
DEBUG(char *fmt, ...) /* debug output, newline at end */
{
    va_list args;
    char line[MAXLINE];
    int n;

    if (Debug == 0)
        return;
    va_start(args, fmt);
    vsprintf(line, fmt, args);
    strcat(line, "\n");
    va_end(args);

    n = strlen(line);
    if (writen(clifd, line, n) != n)
        log_sys("writen error");
}

void
DEBUG_NONL(char *fmt, ...) /* debug output, NO newline at end */
{
    va_list args;
    char line[MAXLINE];
    int n;

    if (Debug == 0)
        return;
    va_start(args, fmt);
    vsprintf(line, fmt, args);
    va_end(args);

    n = strlen(line);
    if (writen(clifd, line, n) != n)
        log_sys("writen error");
}
```

程序18-14是tty_open函数。这个函数用来打开设备并设置设备工作模式，它适用于调制解调器设备或直接相连的设备。Systems文件中的class域和Devices文件指定了线路速率，客户机还可以指定校验方式。

程序18-14 tty_open函数

```
#include "callid.h"
#include <fcntl.h>
#include <termios.h>

/* Open the terminal line */

int
tty_open(char *class, char *line, enum parity parity, int modem)
{
    int fd, baud;
    char devname[100];
    struct termios term;

    /* first open the device */
    strcpy(devname, "/dev/");
```

```

strcat(devname, line);
if ( (fd = open(devname, O_RDWR | O_NONBLOCK)) < 0) {
    sprintf(errmsg, "can't open %s: %s\n",
            devname, strerror(errno));
    return(-1);
}
if (isatty(fd) == 0) {
    sprintf(errmsg, "%s is not a tty\n", devname);
    return(-1);
}

/* fetch then set modem's terminal status */
if (tcgetattr(fd, &term) < 0)
    log_sys("tcgetattr error");

if (parity == NONE)
    term.c_cflag = CS8;
else if (parity == EVEN)
    term.c_cflag = CS7 | PARENB;
else if (parity == ODD)
    term.c_cflag = CS7 | PARENB | PARODD;
else
    log_quit("unknown parity");
term.c_cflag |= CREAD | /* enable receiver */
                HUPCL; /* lower modem lines on last close */
/* 1 stop bit (since CSTOPB off) */

if (modem == 0)
    term.c_cflag |= CLOCAL; /* ignore modem status lines */

term.c_oflag = 0; /* turn off all output processing */
term.c_iflag = IXON | IXOFF | /* Xon/Xoff flow control (default) */
                IGNBRK | /* ignore breaks */
                ISTRIP | /* strip input to 7 bits */
                IGNPAR; /* ignore input parity errors */
term.c_lflag = 0; /* everything off in local flag:
                  disables canonical mode, disables
                  signal generation, disables echo */

term.c_cc[VMIN] = 1; /* 1 byte at a time, no timer */
term.c_cc[VTIME] = 0; /* (See Figure 18.10) */

if (strcmp(class, "38400") == 0) baud = B38400;
else if (strcmp(class, "19200") == 0) baud = B19200;
else if (strcmp(class, "9600") == 0) baud = B9600;
else if (strcmp(class, "4800") == 0) baud = B4800;
else if (strcmp(class, "2400") == 0) baud = B2400;
else if (strcmp(class, "1800") == 0) baud = B1800;
else if (strcmp(class, "1200") == 0) baud = B1200;
else if (strcmp(class, "600") == 0) baud = B600;
else if (strcmp(class, "300") == 0) baud = B300;
else if (strcmp(class, "200") == 0) baud = B200;
else if (strcmp(class, "150") == 0) baud = B150;
else if (strcmp(class, "134") == 0) baud = B134;
else if (strcmp(class, "110") == 0) baud = B110;
else if (strcmp(class, "75") == 0) baud = B75;
else if (strcmp(class, "50") == 0) baud = B50;
else {
    sprintf(errmsg, "invalid baud rate: %s\n", class);
    return(-1);
}

cfsetispeed(&term, baud);
cfsetospeed(&term, baud);

if (tcsetattr(fd, TCSANOW, &term) < 0) /* set attributes */
    log_sys("tcsetattr error");

DEBUG("tty open");

```

```

    clr_fl(fd, O_NONBLOCK);    /* turn off nonblocking */
    return(fd);
}

```

因为有时要打开连接在调制解调器上的终端必须先检测到调制解调器的载波才行，所以我们以非阻塞方式打开终端设备。我们是在向外拨号，而不是向内拨号，所以不愿意等待。在这个函数的结尾处调用 `clr_fl` 函数来清除这种非阻塞方式。在函数 `tty_open` 中，调制解调器和直接相连的线路的唯一区别就是对直接相连的线路要设置 `CLOCAL` 位。

详细的拨号过程在函数 `tty_dial`（见程序 18-15）中实现。这个函数只在调制解调器拨号时被调用，打开直接相连设备时不调用。

程序 18-15 `tty_dial` 函数

```

#include    "calld.h"

int
tty_dial(int fd, char *phone, char *dialer, char *sub, char *expsend)
{
    char    *ptr;

    ptr = strtok(expsend, WHITE);    /* first expect string */
    for ( ; ; ) {
        DEBUG_NONL("expect = %s\nread: ", ptr);
        if (expect_str(fd, ptr) < 0)
            return(-1);

        if ( (ptr = strtok(NULL, WHITE)) == NULL)
            return(0);    /* at the end of the expect/send */
        DEBUG_NONL("send = %s\nwrite: ", ptr);
        if (send_str(fd, ptr, phone, 0) < 0)
            return(-1);

        if ( (ptr = strtok(NULL, WHITE)) == NULL)
            return(0);    /* at the end of the expect/send */
    }
}

```

这个函数只是调用一个函数来处理期望得到的字符串，调用另外一个函数来处理发送的字符串。当没有发送或期望字符串时工作就完成了（这里并不处理表 18-3 中的 `sub` 域）。

程序 18-16 是输出发送字符串的 `send_str` 函数。为了不使得这个程序太过臃肿，我们没有实现每一个转义序列——我们只保证这个程序能处理表 18-3 中的 `Dialers` 文件。

程序 18-16 `send_str` 函数

```

#include    "calld.h"

int
send_str(int fd, char *ptr, char *phone, int echocheck)
{
    char    c, tempc;

    /* go though send string, converting escape sequences on the fly */
    while ( (c = *ptr++) != 0) {
        if (c == '\\') {
            if (*ptr == 0) {
                sprintf(errmsg, "backslash at end of send string\n");
                return(-1);
            }
        }
    }
}

```

```

c = *ptr++;      /* char following backslash */

switch (c) {
case 'c':      /* no CR, if at end of string */
    if (*ptr == 0)
        goto returnok;
    continue; /* ignore if not at end of string */

case 'd':      /* 2 second delay */
    DEBUG_NONL("<delay>");
    sleep(2);
    continue;

case 'p':      /* 0.25 second pause */
    DEBUG_NONL("<pause>");
    sleep_us(250000); /* Exercise 12.6 */
    continue;

case 'e':
    DEBUG_NONL("<echo check off>");
    echocheck = 0;
    continue;

case 'E':
    DEBUG_NONL("<echo check on>");
    echocheck = 1;
    continue;

case 'T':      /* output phone number */
    send_str(fd, phone, phone, echocheck); /* recursive */
    continue;

case 'r':
    c = '\r';
    break;

case 's':
    c = ' ';
    break;

    /* room for lots more case statements ... */

default:
    sprintf(errmsg, "unknown send escape char: \\%s\n",
            ctl_str(c));
    return(-1);
}
}

DEBUG_NONL("%s", ctl_str(c));
if (write(fd, &c, 1) != 1)
    log_sys("write error");
if (echocheck) { /* wait for char to be echoed */
    do {
        if (read(fd, &tempc, 1) != 1)
            log_sys("read error");
        DEBUG_NONL("{%s}", ctl_str(tempc));
    } while (tempc != c);
}
}

c = '\r'; /* if no \c at end of string, CR written at end */
DEBUG_NONL("%s", ctl_str(c));
if (write(fd, &c, 1) != 1)
    log_sys("write error");

```

```

returnok:
    DEBUG("");
    return(0);
}

```

send_str调用函数ctl_str函数将ASCII控制字符转换成可以打印的形式。程序 18-17是ctl_str函数。

程序18-17 ctl_str函数

```

#include    "calld.h"

/* Make a printable string of the character "c", which may be a
 * control character.  Works only with ASCII. */

char *
ctl_str(char c)
{
    static char tempstr[6];    /* biggest is "\177" + null */
    c &= 255;
    if (c == 0)
        return("\\0");        /* really shouldn't see a null */
    else if (c < 040)
        sprintf(tempstr, "%c", c + 'A' - 1);
    else if (c == 0177)
        return("DEL");
    else if (c > 0177)
        sprintf(tempstr, "\\%03o", c);
    else
        sprintf(tempstr, "%c", c);
    return(tempstr);
}

```

在整个拨号过程中最困难的是识别期望字符串。程序 18-18就是来完成这项工作的函数expect_str。(对于发送字符串,我们只实现了Dialers文件所提供的特性的一个子集。)

程序18-18 读取和识别期望字符串的函数组

```

#include    "calld.h"

#define EXPALRM    45        /* alarm time to read expect string */

static int    expalarm = EXPALRM;
static void    sig_alarm(int);
static volatile sig_atomic_t    caught_alarm;

static size_t    exp_read(int, char *);

int
expect_str(int fd, char *ptr)
{
    char    expstr[MAXLINE], inbuf[MAXLINE];
    char    c, *src, *dst, *inptr, *cmpptr;
    int    i, matchlen;

    if (strcmp(ptr, "\\\"") == 0)
        goto returnok;        /* special case of "" (expect nothing) */

    /* copy expect string, converting escape sequences */
    for (src = ptr, dst = expstr; (c = *src++) != 0; ) {

```

```

if (c == '\\') {
    if (*src == 0) {
        sprintf(errmsg, "invalid expect string: %s\n", ptr);
        return(-1);
    }
    c = *src++;      /* char following backslash */
    switch (c) {
        case 'r':    c = '\r'; break;
        case 's':    c = ' '; break;
        /* room for lots more case statements ... */
        default:
            sprintf(errmsg, "unknown expect escape char: \\%s\n",
                ctl_str(c));
            return(-1);
    }
}
*dst++ = c;
}
*dst = 0;
matchlen = strlen(expstr);

if (signal(SIGALRM, sig_alm) == SIG_ERR)
    log_quit("signal error");
caught_alm = 0;
alarm(expalarm);

do {
    if (exp_read(fd, &c) < 0)
        return(-1);
} while (c != expstr[0]); /* skip until first chars equal */

cmpptr = inptr = inbuf;
*inptr = c;

for (i = 1; i < matchlen; i++) { /* read matchlen chars */
    inptr++;
    if (exp_read(fd, inptr) < 0)
        return(-1);
}

for ( ; ; ) { /* keep reading until we have a match */
    if (strncmp(cmpptr, expstr, matchlen) == 0)
        break; /* have a match */
    inptr++;
    if (exp_read(fd, inptr) < 0)
        return(-1);
    cmpptr++;
}
returnok:
alarm(0);
DEBUG("\nextpect: got it");
return(0);
}

size_t /* read one byte, handle timeout errors & DEBUG */
exp_read(int fd, char *buf)
{
    if (caught_alm) { /* test flag before blocking in read */
        DEBUG("\nread timeout");
        return(-1);
    }
    if (read(fd, buf, 1) == 1) {
        DEBUG_NONL("%s", ctl_str(*buf));
        return(1);
    }
}

```

```
    if (errno == EINTR && caught_alarm) {
        DEBUG("\nread timeout");
        return(-1);
    }
    log_sys("read error");
}

static void
sig_alarm(int signo)
{
    caught_alarm = 1;
    return;
}
```

我们首先复制期望字符串，转换特殊字符。匹配方法是从调制解调器读取字符直到该字符与期望字符串的第一个字符匹配，然后再读取与期望字符串同样多的字符。从这开始，从调制解调器连续地读取字符到缓存中，把它们与期望字符串比较，直至得到整个匹配串（当然还有更好的匹配算法，我们所选的这个算法只是为了简化编程。从调制解调器读取的字符一般 50 个一组读入，期望字符串的长度一般是 10~20 个字符）。

每当尝试匹配一个期望串时，都必须设置一个警告标志，警告是我们可以确定没有收到匹配字符串的唯一方式。

现在介绍完了服务器精灵进程。这个精灵进程所做的其实就是打开一个终端设备和使用调制解调器拨号。至于打开终端设备以后的工作取决于客户机。下面将看到一个提供了类似于 `cu` 和 `tip` 界面的客户机，它允许我们对远程系统拨号并登录。

18.7 客户机设计

客户机与服务器之间的界面只是若干行代码。客户机生成一个命令行，发送到服务器，然后收到一个文件描述符或者一个错误消息。客户机的设计着重于客户机如何处理返回的文件描述符。这一节描述了一个类似于 `cu` 和 `tip` 程序的 `call` 客户机程序。这个程序允许我们对远程系统拨号，并登录。远程系统并不一定是一个 UNIX 系统。我们可以使用这个程序来同那些与本机通过 RS-232 串口连接的系统或设备进行通信。

18.7.1 终端行规程

图 12-7 和 12-8 给出了一个调制解调器拨号器的概况。图 18-4 则是图 12-7 的扩充。这里要注意的是，在用户和调制解调器之间有两个行规程，并假设我们使用这个程序来拨号一个远程 UNIX 系统。（回忆程序 12-10 的输出，与一个基于流的终端系统相比，图 18-4 只是一个简化。事实上可能有多个流组成这个行规程，可能有多个模块构成终端设备驱动程序。此外没有显式地表明流首。）

图 18-4 本地系统中调制解调器上方的两个虚线框中的过程是由服务器的 `tty_open` 函数（见程序 18-14）建立的。该函数设置虚线框中的终端行规程为非规范模式。本地系统中的调制解调器被服务器函数 `tty_dial` 所拨号（见程序 18-15）。终端行规程的虚线框和 `call` 进程之间的两个箭头对应于服务器端返回的文件描述符。（这里把一个描述符显示为两个箭头，是为了重申它是一个全双工的描述符）。

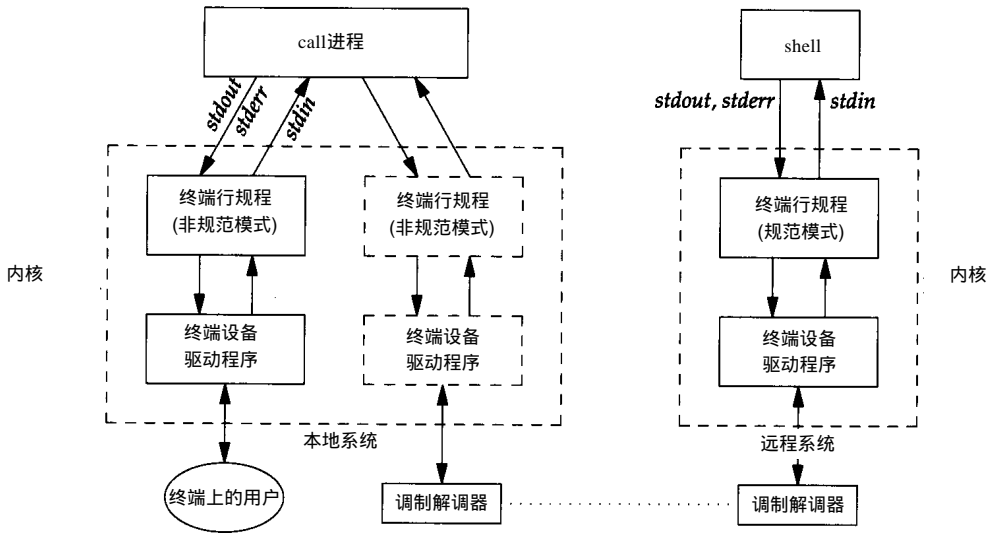


图18-4 调制解调器拨号器登录远程 UNIX 主机过程的示意图

远程系统shell下方的行规程框被登录进程设置为规范模式。当拨通远程系统时，我们希望输入的特殊字符（如文件结束、删除行等）被远程主机上的行规程模块所认识。这样，就必须将终端（标准输入、标准输出、call进程的标准出错）上方的行规程模块设置为非规范模式。

18.7.2 一个进程还是两个进程

在图18-4中，我们看到的call程序只有一个进程。因为要读取两个描述符、写两个描述符，所以要求支持像select或者poll一样的I/O多路转接函数。我们同样也可以把客户机程序设计为如图12-8中的两个进程：一个父进程、一个子进程。图18-5就显示了这两个进程以及它们下方的行规程。从历史上看，cu和tip就像图18-5中所示，一直是两个进程。这是因为早期的UNIX系统不支持I/O多路转接函数。

我们基于以下两个原因采用一个进程：

(1) 采用两个进程会使得客户机的终止变得复杂。如果在一行的开始处输入~（一个波浪符号加上一个点）来终止连接，子进程认识了这个符号并终止，父进程却必须捕捉到SIGCHLD信号才终止。

如果这个连接是由远程系统终止的，或者线路断掉了，父进程从调制解调器描述符读取到文件终止符号而检测到这情况。然后，父进程必须通知子进程，这样子进程才会终止。

使用一个进程则避免了终止时的进程间通信。

(2) 我们要在客户机实现一个文件传送函数，类似于cu和tip程序中的put和take命令。我们在标准输入中输入这些命令，在一行开始处输入一个波浪号（缺省的转义字符）。如果采用两个进程，这些命令被子进程所识别（见图18-5）。但是客户机接收到的文件（使用take命令），会使用调制解调器描述符，而这时这个描述符正在被父进程读取。这样的话，为了实现take命令，子进程就必须通知父进程，让父进程停止从调制解调器读取。父进程可能会在读取描述符

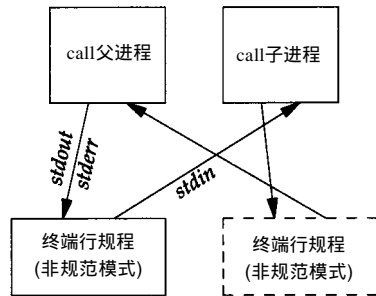


图18-5 两个进程的call程序

时被阻塞，这样还需要一个信号来中断父进程的读取。当子进程结束后，还需要另外通知父进程继续读取调制解调器。这样经常会使情况变得混乱。

一个单一的进程会简化整个客户机。但是，若只使用一个进程，则无法只是停止子进程的作业，而BSD的tip程序则支持这种特性。它允许停止子进程而父进程仍然继续运行。这意味着所有的终端输入被重定向到shell，而不是子进程中，这使我们可以工作在本地系统，仍然可以看到远程系统产生的输出。当在远程系统上运行一个很长时间的作业，又希望在本地系统中看到它产生的输出的情况，这个功能就很方便。

现在来看一下实现客户机的源码。

18.8 客户机源码

因为客户机并不处理与远程系统相连接的细节，与服务器比较，客户机比较小些。客户机程序中大约有一半用于处理那些类似于take和put的命令。

程序18-19是call.h头文件，它被包含在所有的源文件中。

程序18-19 call.h 头文件

```
#include <sys/types.h>
#include <sys/time.h>
#include <errno.h>
#include <termios.h>
#include "ourhdr.h"

#define CS_CALL "/home/stevens/calld" /* well-known server name */
#define CL_CALL "call" /* command for server */

/* declare global variables */
extern char escapec; /* tilde for local commands */
extern char *src; /* for take and put commands */
extern char *dst; /* for take and put commands */

/* function prototypes */
int call(const char *);
int doescape(int);
void loop(int);
int prompt_read(char *, int (*)(int, char **));
void put(int);
void take(int);
int take_put_args(int, char **);
```

发送给服务器的命令和服务器的众所周知名字必须与程序 18-1 中的值保持一致。

程序18-20是客户机的main函数。

程序18-20 main函数

```
#include "call.h"

/* define global variables */
char escapec = '~';
char *src;
char *dst;

static void usage(char *);

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

```

{
    int          c, remfd, debug;
    char        args[MAXLINE];

    args[0] = 0;    /* build arg list for conn server here */
    opterr = 0;    /* don't want getopt() writing to stderr */
    while ( (c = getopt(argc, argv, "des:o")) != EOF) {
        switch (c) {
            case 'd':    /* debug */
                debug = 1;
                strcat(args, "-d ");
                break;

            case 'e':    /* even parity */
                strcat(args, "-e ");
                break;

            case 'o':    /* odd parity */
                strcat(args, "-o ");
                break;

            case 's':    /* speed */
                strcat(args, "-s ");
                strcat(args, optarg);
                strcat(args, " ");
                break;

            case '?':
                usage("unrecognized option");
        }
    }
    if (optind < argc)
        strcat(args, argv[optind]); /* name of host to call */
    else
        usage("missing <hostname> to call");

    if ( (remfd = call(args)) < 0) /* place the call */
        exit(1);    /* call() prints reason for failure */
    printf("Connected\n");

    if (tty_raw(STDIN_FILENO) < 0) /* user's tty to raw mode */
        err_sys("tty_raw error");
    if (atexit(tty_atexit) < 0) /* reset user's tty on exit */
        err_sys("atexit error");

    loop(remfd);    /* and do it */

    printf("Disconnected\n\r");
    exit(0);
}

static void
usage(char *msg)
{
    err_quit("%s\nusage: call -d -e -o -s<speed> <hostname>", msg);
}

```

main函数处理命令行参数，把它们保存在args数组中，再把它们发送给服务器。call函数与服务器联系并返回远程系统的文件描述符。

图18-4终端上方的行规程模块被tty_raw函数（见程序11-10）设置为非规范模式。为了在完成工作后重置终端，我们将tty_atexit函数设置为终止处理程序。

调用函数loop把输入到调制解调器的所有东西和从调制解调器读取的东西复制到终端上。

程序18-21中的call函数联系服务器以得到一个调制解调器的文件描述符。如前所述，它只

用了若干行代码就完成了这项工作。

程序18-21 call函数

```
#include "call.h"
#include <sys/uio.h> /* struct iovec */

/* Place the call by sending the "args" to the calling server,
 * and reading a file descriptor back. */

int
call(const char *args)
{
    int          csfd, len;
    struct iovec iov[2];

    /* create connection to conn server */
    if ( (csfd = cli_conn(CS_CALL)) < 0)
        err_sys("cli_conn error");

    iov[0].iov_base = CL_CALL " ";
    iov[0].iov_len  = strlen(CL_CALL) + 1;
    iov[1].iov_base = (char *) args;
    iov[1].iov_len  = strlen(args) + 1;
    /* null at end of args always sent */
    len = iov[0].iov_len + iov[1].iov_len;
    if (writev(csfd, &iov[0], 2) != len)
        err_sys("writev error");

    /* read back descriptor */
    /* returned errors handled by write() */
    return( recv_fd(csfd, write) );
}
```

loop函数处理两个输入流和两个输出流之间的多路转接。可以使用 poll或者select，这取决于本地系统提供什么函数。程序18-22使用poll来实现。

程序18-22 使用poll函数的loop函数

```
#include "call.h"
#include <poll.h>
#include <stropts.h>

/* Copy everything from stdin to "remfd",
 * and everything from "remfd" to stdout. */

#define BUFFSIZE 512

void
loop(int remfd)
{
    int          bol, n, nread;
    char         c, buff[BUFFSIZE];
    struct pollfd fds[2];

    setbuf(stdout, NULL); /* set stdout unbuffered */
                          /* (for printf's in take() and put() */
    fds[0].fd = STDIN_FILENO; /* user's terminal input */
    fds[0].events = POLLIN;
    fds[1].fd = remfd; /* input from remote (modem) */
    fds[1].events = POLLIN;

    for ( ; ; ) {
```

```
if (poll(fds, 2, INFTIM) <= 0)
    err_sys("poll error");

if (fds[0].revents & POLLIN) { /* data to read on stdin */
    if (read(STDIN_FILENO, &c, 1) != 1)
        err_sys("read error from stdin");

    if (c == escapec && bol) {
        if ( (n = doescape(remfd)) < 0)
            break; /* user wants to terminate */
        else if (n == 0)
            continue; /* escape seq has been processed */

        /* else, char following escape was not special,
           so it's returned and echoed below */
        c = n;
    }
    if (c == '\r' || c == '\n')
        bol = 1;
    else
        bol = 0;

    if (write(remfd, &c, 1) != 1)
        err_sys("write error");
}
if (fds[0].revents & POLLHUP)
    break; /* stdin hangup -> done */

if (fds[1].revents & POLLIN) { /* data to read from remote */
    if ( (nread = read(remfd, buff, BUFSIZE)) <= 0)
        break; /* error or EOF, terminate */

    if (written(STDOUT_FILENO, buff, nread) != nread)
        err_sys("written error to stdout");
}
if (fds[1].revents & POLLHUP)
    break; /* modem hangup -> done */
}
}
```

loop函数的基本循环只是在等待从调制解调器或终端来的数据。当从终端读取到数据时，就把它拷贝到调制解调器，反过来也一样。稍微复杂一点的是，要识别一行的第一个转义字符（波浪线）。

要注意，从终端（标准输入）每次读取一个字符，但从调制解调器每次读取一个缓存的内容。每次从终端读取一个字符的原因之一是，我们必须在新行开始处注意每一个字符以识别可能的特殊字符。虽然每次I/O读取一个字符浪费了CPU时间（见表3-1），但是一般来说从终端的输入要比从远程系统得到的内容要少的多。（在使用本程序的一个定量测试中，本地每一个输入，远程系统就有大约100个输出。）

当检测到转义字符时，就调用doescape函数来处理这个输入的命令（见程序18-23）。我们这里只支持5个命令。简单的命令就在这个函数内部实现，复杂的命令，如take和put，则通过各自的函数实现。

- 一个句号终止客户机。对于某些设备，如激光打印机，这是终止客户机的唯一方法。当登录到远程系统后（见图18-4），退出远程系统会引起调制解调器掉线，loop函数就会从调制解调器得到挂断信号。

- 如果系统支持作业控制，则识别作业控制挂起字符，并挂起客户机。注意直接识别这个字符要比使用行规程识别它而产生SIGSTOP信号要简单些（见程序10-22）。那样，在停止之前不得不重置终端模式，而继续执行时又要重置。

• 调制解调器描述符中英镑符号产生 BREAK。使用 POSIX.1 的 `tcsendbreak` 函数来实现这个 BREAK (见 11.8 节)。这个 BREAK 经常会引起远程系统的 `getty` 或者 `ttymon` 程序改变线路速率。(见 9.2 节)

• `take` 和 `put` 命令需要分别调用不同的函数。区分这两个命令的方法是记住它们说明了客户机在本地系统要做的操作：从远程系统取 (`take`) 一个文件或者送 (`put`) 一个文件给远程系统。

程序 18-24 是实现 `take` 命令的代码。`take` 函数先是调用 `prompt_read` (见程序 18-25)，它显示 `^[take]` 提示来响应 `^t` 命令。`prompt_read` 函数然后从终端读取输入的一行，包含源路径 (远程系统上的文件) 和目标路径 (本机上的文件)。把读取的结果储存在全局变量 `src` 和 `dst` 中。

程序 18-23 `escape` 函数

```
#include "call.h"
#include <signal.h>

/* Called when first character of a line is the escape character
 * (tilde). Read the next character and process. Return -1
 * if next character is "terminate" character, 0 if next character
 * is valid command character (that's been processed), or next
 * character itself (if the next character is not special). */

int
doescape(int remfd)
{
    char c;

    if (read(STDIN_FILENO, &c, 1) != 1) /* next input char */
        err_sys("read error from stdin");

    if (c == escapec) /* two in a row -> process as one */
        return(escapec);

    else if (c == '.') { /* terminate */
        write(STDOUT_FILENO, "~.\n\r", 4);
        return(-1);
    }

#ifdef VSUSP
    } else if (c == tty_termios()->c_cc[VSUSP]) { /* suspend client */
        tty_reset(STDIN_FILENO); /* restore tty mode */
        kill(getpid(), SIGTSTP); /* suspend ourself */

        tty_raw(STDIN_FILENO); /* and reset tty to raw */
        return(0);
    }
#endif

    } else if (c == '#') { /* generate break */
        tcsendbreak(remfd, 0);
        return(0);

    } else if (c == 't') { /* take a file from remote host */
        take(remfd);
        return(0);

    } else if (c == 'p') { /* put a file to remote host */
        put(remfd);
        return(0);
    }

    return(c); /* not a special character */
}
```

程序18-24 处理take命令

```
#include "call.h"

#define CTRLA 001 /* eof designator for take */

static int rem_read(int);
static char rem_buf[MAXLINE];
static char *rem_ptr;
static int rem_cnt = 0;

/* Copy a file from remote to local. */

void
take(int remfd)
{
    int n, linecnt;
    char c, cmd[MAXLINE];
    FILE *fpout;

    if (prompt_read("[take] ", take_put_args) < 0) {
        printf("usage: [take] <sourcefile> <destfile>\n\r");
        return;
    }

    /* open local output file */
    if ( (fpout = fopen(dst, "w")) == NULL) {
        err_ret("can't open %s for writing", dst);
        putc('\r', stderr);
        fflush(stderr);
        return;
    }

    /* send cat/echo command to remote host */
    sprintf(cmd, "cat %s; echo %c\r", src, CTRLA);
    n = strlen(cmd);
    if (write(remfd, cmd, n) != n)
        err_sys("write error");

    /* read echo of cat/echo command line from remote host */
    rem_cnt = 0; /* initialize rem_read() */
    for ( ; ; ) {
        if ( (c = rem_read(remfd)) == 0)
            return; /* line has dropped */
        if (c == '\n')
            break; /* end of echo line */
    }

    /* read file from remote host */
    linecnt = 0;
    for ( ; ; ) {
        if ( (c = rem_read(remfd)) == 0)
            break; /* line has dropped */
        if (c == CTRLA)
            break; /* all done */

        if (c == '\r')
            continue; /* ignore returns */
        if (c == '\n') /* but newlines are written to file */
            printf("\r%d", ++linecnt);
        if (putc(c, fpout) == EOF)
            break; /* output error */
    }
    if (ferror(fpout) || fclose(fpout) == EOF) {
        err_msg("output error to local file");
        putc('\r', stderr);
        fflush(stderr);
    }
}
```

```

    }
    c = '\n';
    write(remfd, &c, 1);
}

/* Read from remote. Read up to MAXLINE, but parcel out one
 * character at a time. */

int
rem_read(int remfd)
{
    if (rem_cnt <= 0) {
        if ( (rem_cnt = read(remfd, rem_buf, MAXLINE)) < 0)
            err_sys("read error");
        else if (rem_cnt == 0)
            return(0);
        rem_ptr = rem_buf;
    }
    rem_cnt--;
    return(*rem_ptr++ & 0177);
}

```

在take函数为写而打开本地文件后，发送以下命令到远程主机：

```
cat sourcefile; echo ^A
```

这使远程主机执行cat命令，然后回显Ctrl-A的ASCII字符。我们从远程主机的返回信息中查看Ctrl-A字符，当发现后，就知道文件传送已经结束了。注意同样必须读取发送的命令行的回应，只有确定了命令的回应后，才能开始接受cat命令的输出。

当从远程文件中读取时，查找新行的标志，记录下新行的数目。在左边界显示这个行号，覆盖原有的行号（在printf中使用回车终止一行，而没有使用换行符）。这就在终端上提供了一个可视的文件传送进度指示，最后在结尾处显示最终的行号。

客户机源文件中也包含了rem_read函数，这个函数被用来从远程主机读取每个字符。每次读取一个缓存的大小，但每次只返回拨号者一个字符。

最初，take命令的程序每次读取一个字符，就像以前的cu和tip程序一样。十年以前，那时1200波特率的调制解调器还被认为是高速，这样做是可以的。但是使用现在快速的调制解调器，它们以9600或以上的波特率将字符传送给终端设备，那么字符就会丢失。作者使用的是一个PEP模式Telebit T2500的调制解调器，即使在本地主机和远程主机都使用了流量控制情况下，用cu和tip命令时还是会遇到这个问题，当传送一个大的文本文件（大约75 000字节）时，大约在一半的时候字符丢失了，只得重新传送。

解决方法是对rem_read函数重新编码，每次读取一个缓存的字符。这样做会将CPU的时间减少到大约1/3左右（传送这个75 000字节的文件，从16秒减少到5秒），而且每次都提供了可信的文件传送。在rem_read函数中临时加入一个计数器，可以看到每次调用时读取到多少字符。表18-5显示了一个结果。

表18-5 文件传送时读取的字节数

字节数	次数	字节数	次数	字节数	次数	字节数	次数
1	1	28	2	39	1	55	1
13	1	29	1	40	1	56	9

(续)

字节数	次数	字节数	次数	字节数	次数	字节数	次数
16	1	32	1	46	1	57	751
17	1	33	1	48	2	58	530
22	1	34	1	51	2	59	2
24	1	35	1	52	2	114	1
25	4	37	1	53	1	115	1
26	3	38	1	54	1	194	1

在这个结果中，只有一次是返回一个字节，99%次是返回57或者58个字节。这个改动把读取的次数从75 000次减少到1 329次。

注意表18-5中每次返回的字节数，这时行规程模块通过 `tty_open` 函数（见程序18-14）将它的MIN设置为1且TIME设置为0。这就是11.11节中的实例B。在这里要注意MIN仅仅只是最小值，如果要求的数量比最小值多而且远程系统已经准备好，则可以读取到更多的字节。当MIN被设置为1时，我们并没有被限制为每次读取一个字节。

程序18-25是两个辅助的函数 `take_put_args` 和 `prompt_read`。其中 `prompt_read` 会被 `take` 和 `put` 两函数调用，并以 `take_put_args` 作为它的一个参数，然后，由 `buf_args` 函数调用（见程序15-17）。

程序18-25 `take_put_args`和`prompt_read`函数

```
#include    "call.h"

/* Process the argv-style arguments for take or put commands. */

int
take_put_args(int argc, char **argv)
{
    if (argc == 1) {
        src = dst = argv[0];
        return(0);
    } else if (argc == 2) {
        src = argv[0];
        dst = argv[1];
        return(0);
    }
    return(-1);
}

static char cmdargs[MAXLINE];
        /* can't be automatic; src/dst point into here */

/* Read a line from the user. Call our buf_args() function to
 * break it into an argv-style array, and call userfunc() to
 * process the arguments. */

int
prompt_read(char *prompt, int (*userfunc)(int, char **))
{
    int    n;
    char   c, *ptr;

    tty_reset(STDIN_FILENO);    /* allow user's editing chars */

    n = strlen(prompt);
```



```

if (write(STDOUT_FILENO, prompt, n) != n)
    err_sys("write error");

ptr = cmdargs;
for ( ; ; ) {
    if ( (n = read(STDIN_FILENO, &c, 1)) < 0)
        err_sys("read error");
    else if (n == 0)
        break;
    if (c == '\n')
        break;
    if (ptr < &cmdargs[MAXLINE-2])
        *ptr++ = c;
}
*ptr = 0;          /* null terminate */

tty_raw(STDIN_FILENO);    /* reset tty mode to raw */

return( buf_args(cmdargs, userfunc) );
        /* return whatever userfunc() returns */
}

```

函数prompt_read从终端读入一行，然后调用buf_args将这一行分解为标准的参数列表，然后用take_put_args来处理这些参数。需要注意的是终端被重置到规范模式来读取参数，当输入命令行时允许使用标准的编辑字符。

最后的客户机函数是put，见程序18-26。这个函数被调用来拷贝一个本地文件到远程主机。

程序18-26 put 函数

```

#include    "call.h"

/* Copy a file from local to remote. */

void
put(int remfd)
{
    int    i, n, linecnt;
    char   c, cmd[MAXLINE];
    FILE   *fpin;

    if (prompt_read("[put] ", take_put_args) < 0) {
        printf("usage: [put] <sourcefile> <destfile>\n\r");
        return;
    }

    /* open local input file */
    if ( (fpin = fopen(src, "r")) == NULL) {
        err_ret("can't open %s for reading", src);
        putc('\r', stderr);
        fflush(stderr);
        return;
    }

    /* send stty/cat/stty command to remote host */
    sprintf(cmd, "stty -echo; cat >%s; stty echo\r", dst);
    n = strlen(cmd);
    if (write(remfd, cmd, n) != n)
        err_sys("write error");
    tcdrain(remfd);    /* wait for our output to be sent */
    sleep(4);          /* and let stty take effect */
}

```

```

        /* send file to remote host */
linecnt = 0;
for ( ; ; ) {
    if ( (i = getc(fpin)) == EOF)
        break;          /* all done */
    c = i;
    if (write(remfd, &c, 1) != 1)
        break;          /* line has probably dropped */
    if (c == '\n')      /* increment and display line counter */
        printf("\r%d", ++linecnt);
}

    /* send EOF to remote, to terminate cat */
c = tty_termios()->c_cc[VEOF];
write(remfd, &c, 1);
tcdrain(remfd);      /* wait for our output to be sent */
sleep(2);
tcflush(remfd, TCIOFLUSH); /* flush echo of stty/cat/stty */
c = '\n';
write(remfd, &c, 1);

if (ferror(fpin)) {
    err_msg("read error of local file");
    putc('\r', stderr);
    fflush(stderr);
}
fclose(fpin);
}

```

就像take命令一样，我们发送一个命令字符串给远程系统。这次命令是：

```
stty -echo; cat destfile; stty echo
```

我们必须关闭回送（echo），否则整个文件将回送给我们。为了终止cat命令，发送文件终止符（一般采用Ctrl-D）。这要求本机和远程主机使用相同的文件终止符。另外，文件中不能含有远程系统中的ERASE和KILL特殊字符。

18.9 小结

本章讨论了两个不同的程序：一个是服务器的精灵进程用来拨号远程系统，另一个是远程登录程序，它使用服务器连接远程系统。服务器也可以被其他需要与远程系统连接的程序或其他与主机通过异步终端端口连接的硬件设备所使用。

服务器的设计类似与15.6节中的open服务器，需要使用流管道，每个客户机要连接到服务器上，并传递文件描述符。这些高级的进程间通信特性允许我们建立具有我们所需特定功能（见18.3节）的客户机-服务器应用。

客户机类似UNIX系统中的cu和tip程序，但在本章的例子中并不需要关心拨号、与UUCP锁定的文件是否冲突或者如何建立调制解调器行规程模块等细节。服务器将处理这些细节。它使我们集中在客户机所关心的事情上，例如如何提供一个可信的文件传输机制等。

习题

- 18.1 怎样才能避免18.3节中的第(1)步（手工起动服务器）？
- 18.2 在程序18-4中，如果不将optind设置为1会出现什么情况？
- 18.3 如果在request函数（见程序18-10）中创建一个子进程后，在子进程终止前修改了Systems文件会出现什么情况？

18.4 7.8节提到在可重新分配的存储区域中要谨慎使用指针，因为这区域可以移动。但在18.3节中，为什么能在可重新分配的client数组上使用cliptr指针呢？

18.5 如果take或put命令的路径参数中含有一个分号，会出现什么情况呢？

18.6 修改服务器，使得它在启动时一次读取三个数据文件并保存在存储区域中。那么，如果数据文件被修改了，服务器应当怎么处理呢？

18.7 程序18-21中，为什么要在填充writev函数中的结构时，发送一个参数args？

18.8 用select函数代替poll函数来实现程序18-22。

18.9 如何确定用put命令传送的文件不包含可以被远程系统上行规程解释的字符？

18.10 dialing 函数越早发现拨号失败，才能越早尝试 Systems文件中的下一项。如果可以确定远程系统的电话占线，而不是等到计时器 expect_str超时才确定拨号失败，就可以节约15~20秒时间。为了处理这些错误，在4.3BSD UUCP的期望-发送字符串中增加了期望字符串ABORT。它加上一个说明字符串，就可以放弃当前的拨号。例如，在表18-3最后期望字符串CONNECT\Sfast的前面，可以加上ABORT BUSY来实现这个功能。

第19章 伪终端

19.1 引言

第9章介绍进行终端登录时，需要通过一个终端设备自动提供终端的语义。在终端和运行程序之间有一个终端行规程（见图 11-2），通过这个规程我们能够在终端上设置特殊字符（退格、行删除、中断等）。但是，当一个登录请求到达网络连接时，终端行规程并不是自动被加载到网络连接和登录程序 shell 之间的。图9-5显示了一个伪终端设备驱动程序被用来提供终端语义。

除了用于网络登录，伪终端还被用在其他方面，本章将对此进行介绍。我们将首先提供 SVR4和4.3+BSD系统下用于创建伪终端的函数，然后使用这些函数编写一个程序用来调用 `pty`。我们将看到这个程序的各种使用：在输入字符和终端显示之间进行转换（BSD的码转换程序）和运行协同进程来避免程序 14-10中遇到的缓存问题。

19.2 概述

伪终端（pseudo terminal）这个名词暗示了与一个应用程序相比，它更加像一个终端。但事实上，伪终端并不是一个真正的终端。图 19-1显示了使用伪终端的进程的典型结构。其中关键点如下：

(1) 通常一个进程打开伪终端主设备然后调用 `fork`。子进程建立了一个新的对话，打开一个相应的伪终端从设备，将它复制成标准输入、标准输出和标准出错，然后调用 `exec`。伪终端从设备成为子进程的控制终端。

(2) 对于伪终端从设备之上的用户进程来说，其标准输入、标准输出和标准出错都能当作终端设备使用。用户进程能够调用第 11章中讲到的所有输入/输出函数。但是因为是在伪终端从设备之下并没有真正的设备，无意义的函数调用（改变波特率、发送中断符、设置奇偶校验等）将被忽略。

(3) 任何写到伪终端主设备的输入都会作为从设备端的输入，反之亦然。事实上所有从设备端的输入都来自于主设备上的用户进程。这看起来就像一个流管道（见图 15-3），但从设备上的终端行规程使我们拥有普通管道之外的其他处理能力。

图19-1显示了BSD系统中的伪终端结构。19.3.2节将介绍如何打开这些设备。在 SVR4系统中伪终端是使用流系统来创建的（见 12.4节）。图19-2详细描述了SVR4系统中各个伪终端模块

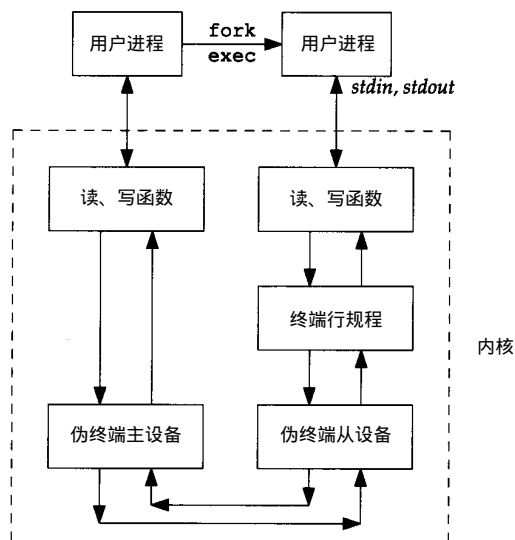


图19-1 典型的伪终端进程结构

之间的关系。虚线框中的两个流模块是可选的。请注意从设备上的三个流模块同程序 12-10 (网络登录) 的输出是一样的。19.3.1节将介绍如何组织这些流模块。

从现在开始将简化以上图示, 首先不再画出图 19-1 的“读、写功能”或图 19-2 的流首。使用缩写“pty”表示伪终端, 并将图 19-2 中所有伪终端从设备之上的流模块集合表示为“终端行规程”模块。

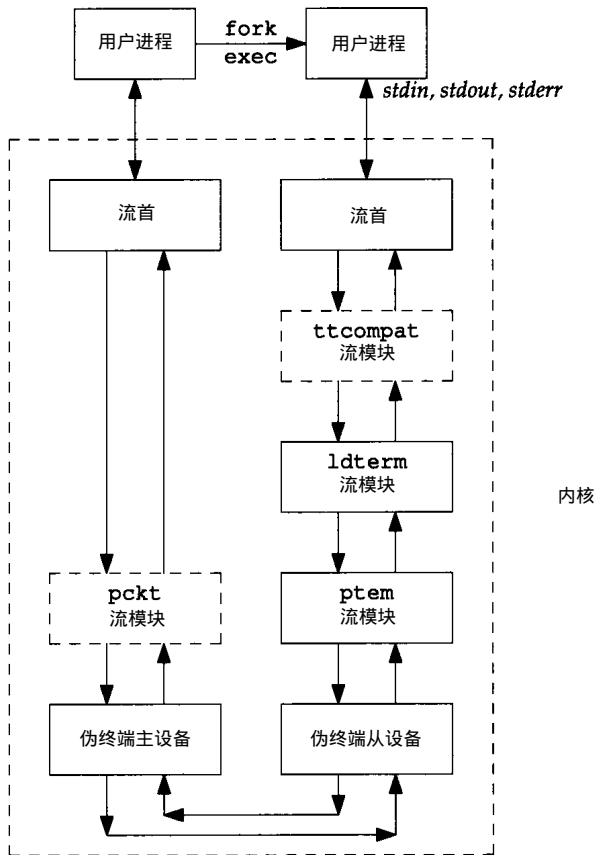


图19-2 SVR4下的伪终端结构

19.2.1 网络登录服务器

伪终端用于构造网络登录服务器。典型的例子是 telnetd 和 rlogind 服务器。Stevens [1990] 第15章详细讨论了提供 rlogin 服务的步骤。一旦登录 shell 运行在远端主机上, 即可得到如图 19-3 的结构。同样的结构也用于 telnetd 服务器。

在 rlogind 服务器和登录 shell 之间有两个 exec 调用, 这是因为 login 程序通常是在两个 exec 之间检验用户是否合法的。

本图的一个关键点是驱动伪终端主设备的进程通常同时在读写另一个 I/O 流。本例中另一个 I/O 流是 TCP/IP。这表示该进程必然使用了某种形式的如 select 或 poll 那样的 I/O 多路转接 (见 12.5 节), 或被分成两个进程。回忆 18.7 节讨论过的一个进程和两个进程的比较。

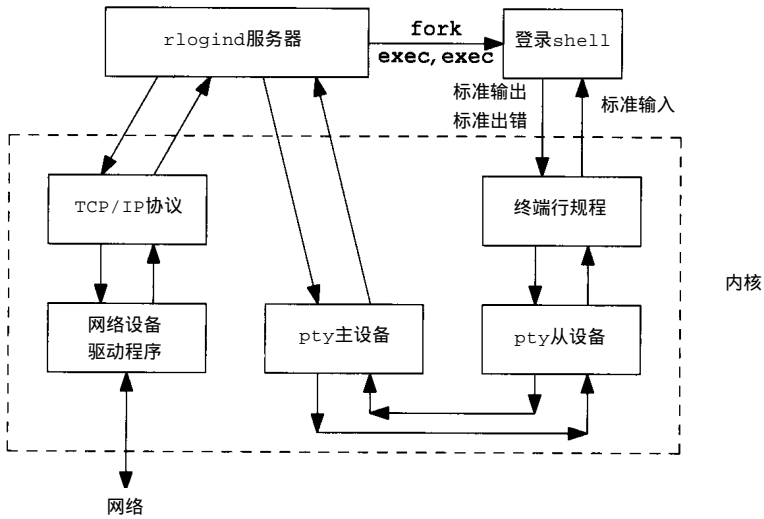


图19-3 rlogind服务器的进程组织结构

19.2.2 script程序

script程序是随SVR4和4.3+BSD提供的，该程序将终端对话期间所有的输入和输出信息在一个文件中做一个拷贝。它通过将自己置于终端和登录 shell的一个新的调用之间来完成这个工作。图19-4详细描述了script程序相关的交互。这里特别指出 script程序通常是从登录 shell起动的，该shell然后等待程序的结束。

script程序运行时，在伪终端从设备之上终端规程的所有输出都被复制到一个 script文件中（通常叫做typescript）。因为击键通常被行规程的模块回显，该 script文件也包括了输入的内容。但是，因为口令不被回显，该 script文件不会包含口令。

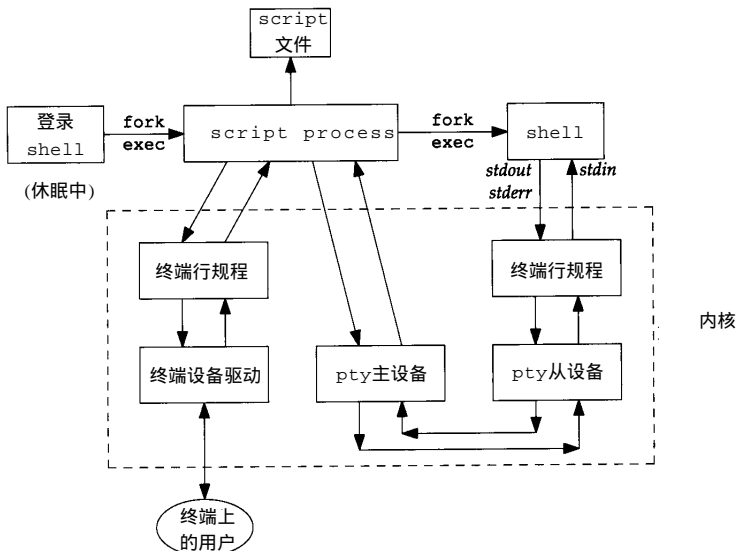


图19-4 script程序

本书中所有运行程序并显示其输出的实例都是由 script 程序实现的，这样避免了手动拷贝程序输出可能带来的错误。

在19.5节开发一个通用的pty程序后，我们将看到一个巧妙的shell脚本能够将它转化成一个script程序。

19.2.3 expect程序

伪终端可以用来使交互式的程序运行在非交互的状态中。许多程序需要一个终端来运行，18.7节中的call进程就是一个例子。它假定标准输入是一个终端并在启动时将其设置为初始模式（见程序18-20）。该程序不能被shell脚本用来运行以自动拨号到远程系统，登录，取出信息和注销。

同修改所有交互式程序来支持批处理模式的操作比较，一个更好的解决方法是通过一个script来驱动交互式程序。expect程序[Libes 1990;1991]提供了这样的方法。类似于19.5节的pty程序，它使用伪终端来运行其他程序。并且，expect还提供了一种编程语言用于检查程序的输出，以确定用什么作为输入发送给该程序。当一个交互式的程序开始从一个脚本运行时，不能仅仅是将脚本中的所有内容输入到程序中去。相应的，要通过检查程序的输出来决定下一步输入的内容。

19.2.4 运行协同进程

在程序14-10所示的例子中，不能调用使用标准I/O库进行输入、输出的协同进程，这是因为当通过管道与协同进程进行通讯时，标准I/O库会将标准输入和输出的内容放到缓存中，从而引起死锁。如果协同进程是一个已经编译的程序而我们又没有源程序，则无法在源程序中加入fflush语句来解决这个问题。图8显示了一个进程驱动协同进程的情况。我们只需将一个伪终端放到两个进程之间，见图19-5。



图19-5 用伪终端驱动一个协同进程

现在协同进程的标准输入和标准输出就像终端设备一样，所以标准I/O库会将这两个流设置为行缓存。

父进程有两种不同的方法在自身和协同进程之间获得伪终端（这种情况下的父进程可以类似程序14-9，使用两个管道和协同进程进行通讯；或者像程序15-1那样，使用一个流管道）。一个方法是父进程直接调用pty_fork函数（见19.4节）而不是fork。另一种方法是exec该pty程序，将协同进程作为参数（见19.5节）。我们将在说明pty程序后介绍这两种方法。

19.2.5 观看长时间运行程序的输出

使用任一个标准shell，都可以将一个需要长时间运行的程序放到后台运行。但是如果将该程序的标准输出重定向到一个文件，并且如果它产生的输出不多，我们就不能方便地监控程序

的进展，这是因为标准I/O库会将标准输出放在缓存中保存。我们看到的将只是成块的输出结果，有时甚至可能是8192字节一块。

如果有源程序，则可以加入fflush调用。另一种方法是，可以在pty程序下运行该程序，让标准I/O库认为输出是终端。图19-6说明了这个结构，我们将这个缓慢输出的程序称为slowout。从登录shell到pty进程的fork/exec箭头用虚线表示，以强调pty进程是作为后台任务运行的。

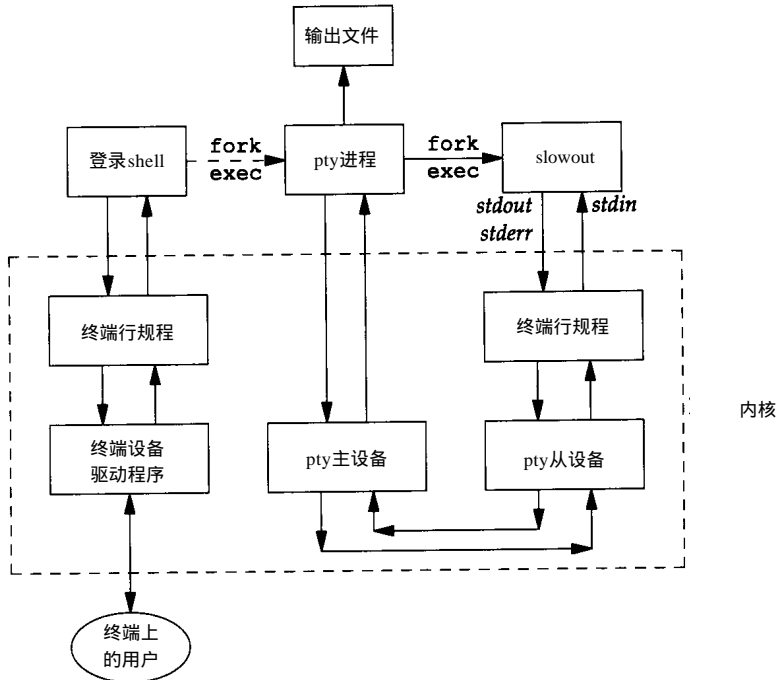


图19-6 使用伪终端运行一个缓慢输出的程序

19.3 打开伪终端设备

在SVR4和4.3+BSD系统中打开伪终端设备的方法有所不同。我们提供两个函数来处理所有细节：ptym_open用来打开下一个有效的伪终端主设备，ptys_open用来打开相应的从设备。

```
#include "ourhdr.h"
```

```
int ptym_open(char *pts_name);
```

返回：若成功则为伪终端主设备文件描述符，否则为 -1

```
int ptys_open(int fdm, char *pts_name);
```

返回：若成功则为伪终端从设备文件描述符，否则为 -1

通常我们不直接调用这两个函数——函数pty_fork（见19.4节）调用它们并fork出一个子进程。

ptym_open决定下一个有效的伪终端主设备并打开该设备。这个调用必须分配一个数组来存放主设备或从设备的名称，并且如果调用成功，相应的主设备或从设备的名称会通过pts_name返回。这个名称和ptym_open返回的文件描述符将传给ptys_open，该函数用来打开一个从设备。

在讲解pty_fork函数之后，使用两个函数来打开这两个设备的原因将会很明显。通常，一个进程调用ptym_open来打开一个主设备并且得到从设备的名称。该进程然后 fork子进程，子进程在调用setsid建立新的对话后调用ptys_open来打开从设备。这就是从设备如何成为子进程的控制终端的过程。

19.3.1 SVR4

SVR4系统下所有伪终端的流实现细节在 AT&T [1990d] 第12章中有所说明，还描述了以下三个函数：grantpt(3)，unlockpt(3)，和ptsname(3)。

伪终端主设备是/dev/ptmx。这是一个流的增殖设备（clone device）。这意味着当我们打开该增殖设备，其open例程自动决定第一个未被使用的伪终端主设备并打开这个设备。（下一节将看到在伯克利系统中，我们必须自己找到第一个未被使用的伪终端主设备。）

程序19-1 SVR4的伪终端打开函数

```
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>
#include <fcntl.h>
#include <stropts.h>
#include "ourhdr.h"

extern char *ptsname(int); /* prototype not in any system header */

int
ptym_open(char *pts_name)
{
    char *ptr;
    int fdm;

    strcpy(pts_name, "/dev/ptmx"); /* in case open fails */
    if ( (fdm = open(pts_name, O_RDWR)) < 0)
        return(-1);

    if (grantpt(fdm) < 0) { /* grant access to slave */
        close(fdm);
        return(-2);
    }
    if (unlockpt(fdm) < 0) { /* clear slave's lock flag */
        close(fdm);
        return(-3);
    }
    if ( (ptr = ptsname(fdm)) == NULL) { /* get slave's name */
        close(fdm);
        return(-4);
    }

    strcpy(pts_name, ptr); /* return name of slave */
    return(fdm); /* return fd of master */
}

int
ptys_open(int fdm, char *pts_name)
{
    int fds;

    /* following should allocate controlling terminal */
    if ( (fds = open(pts_name, O_RDWR)) < 0) {
        close(fdm);
    }
}
```

```
        return(-5);
    }
    if (ioctl(fds, I_PUSH, "ptem") < 0) {
        close(fdm);
        close(fds);
        return(-6);
    }
    if (ioctl(fds, I_PUSH, "ldterm") < 0) {
        close(fdm);
        close(fds);
        return(-7);
    }
    if (ioctl(fds, I_PUSH, "ttcompat") < 0) {
        close(fdm);
        close(fds);
        return(-8);
    }
    return(fds);
}
```

首先打开设备/dev/ptmx并得到伪终端主设备的文件描述符。打开这个主设备自动锁定了对应的从设备。

然后调用grantpt来改变从设备的许可权。执行如下操作：(a) 将从设备的所有权改为有效用户ID；(b) 将组所有权改为组tty；(c) 将许可权改为只允许用户-读，用户-写和组-写。将组所有权设置为tty并允许组-写许可权是因为程序wall(1)和write(1)的设置-用户-ID为组tty。调用函数grantpt执行/usr/lib/pt_chmod。该程序的设置-用户-ID为root，因此它能够修改从设备的所有者和许可权。

函数unlockpt用来清除从设备的内部锁。在打开从设备前必须做这件事情。并且必须调用ptsname来得到从设备的名称。这个名称的格式是/dev/pts/*NNN*。

文件中接下来的函数是ptys_open，该函数真正被用来打开一个从设备。在SVR4系统中，如果调用者是一个还没有控制终端的对话期首进程，open就会分配一个从设备作为控制终端。如果不希望函数自动做这件事，可以在调用时指明O_NOCTTY标志。

打开从设备后，将三个流模块放在从设备的流上。ptem是伪终端虚拟模块，ldterm是终端行规程模块。这两个模块合在一起像一个真正的终端模块一样工作。ttcompat提供了向老系统如V7、4BSD和Xenix的ioctl调用的兼容性。这是一个可选的模块，但是因为它自动尝试控制台登录和网络登录（见程序12-10的输出），我们将其加到从设备的流中。

调用这两个函数的结果是得到：伪终端主设备的文件描述符和从设备的文件描述符。

19.3.2 4.3+BSD

在4.3+BSD系统中必须自己来确定第一个可用的伪终端主设备。为达到这个目的，从/dev/pty0开始并不断尝试，直到成功打开一个可用的伪终端主设备或试完所有设备。在打开设备时，将看到两种可能的错误：EIO指设备已经被使用；ENOENT表示设备不存在。对于后一种情况，可以停止搜索，因为所有的伪终端设备都在被使用中。一旦成功地打开一个例如名为/dev/pty*MN*的伪终端主设备，那么对应的从设备的名称为/dev/tty*MN*。

程序19-2中的函数ptys_open打开该从设备。我们在该函数中调用chown和chmod，必须意识到调用这两个函数的进程必须有超级用户许可权。如果必须改变所有权，那么这两个函数调用必须放在一个设置-用户-ID的root用户的可执行函数中，这类似于4.3+BSD系统下的grantpt函数。

在4.3+BSD系统之下打开pty从设备不具有像分配作为控制终端的设备那样的副作用。下一节将探讨如何在4.3+BSD系统下分配控制终端。

这个函数尝试16种不同的伪终端主设备：从/dev/ptyp0到/dev/ptyTf。具体有效的pty设备号取决于两个因素：(a)内核中配置的号码；(b)/dev目录下的特殊文件号。对于任何程序来说，有效的号码是(a)和(b)中较小的一个。并且，即使(a)和(b)中小小的值大于64，许多现有的BSD应用（如telnetd，rlogind等等）会搜索程序19-2中第一个for循环中的pqrs。

程序19-2 4.3+BSD的伪终端open函数

```
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>
#include <fcntl.h>
#include <grp.h>
#include "ourhdr.h"

int
ptym_open(char *pts_name)
{
    int    fdm;
    char   *ptr1, *ptr2;

    strcpy(pts_name, "/dev/ptyXY");
    /* array index: 0123456789 (for references in following code) */

    for (ptr1 = "pqrstuvwxyzPQRST"; *ptr1 != 0; ptr1++) {
        pts_name[8] = *ptr1;
        for (ptr2 = "0123456789abcdef"; *ptr2 != 0; ptr2++) {
            pts_name[9] = *ptr2;

            /* try to open master */
            if ( (fdm = open(pts_name, O_RDWR)) < 0) {
                if (errno == ENOENT) /* different from EIO */
                    return(-1); /* out of pty devices */
                else
                    continue; /* try next pty device */
            }

            pts_name[5] = 't'; /* change "pty" to "tty" */
            return(fdm); /* got it, return fd of master */
        }
    }
    return(-1); /* out of pty devices */
}

int
ptys_open(int fdm, char *pts_name)
{
    struct group *grp_ptr;
    int gid, fds;

    if ( (grp_ptr = getgrnam("tty")) != NULL)
        gid = grp_ptr->gr_gid;
    else
        gid = -1; /* group tty is not in the group file */

    /* following two functions don't work unless we're root */
    chown(pts_name, getuid(), gid);
}
```

```

chmod(pts_name, S_IRUSR | S_IWUSR | S_IWGRP);
if ( (fds = open(pts_name, O_RDWR)) < 0) {
    close(fdm);
    return(-1);
}
return(fds);
}

```

19.4 pty_fork函数

现在使用上一节介绍的两个函数：ptym_open和ptys_open，编写我们称之为pty_fork的函数。这个新函数具有如下功能：打开主设备和从设备，建立作为对话期管理者的子进程并使其具有控制终端。

```

#include <sys/types.h>
#include <termios.h>
#include <sys/ioctl.h> /* 4.3+BSD defines struct winsize here */
#include "ourhdr.h"

pid_t pty_fork(int *ptrfdm, char *slave_name,
              const struct termios *slave_termios,
              const struct winsize *slave_winsize);

```

返回：子进程中为0，父进程中为子进程的进程ID，若错误则为-1

pty主设备的文件描述符通过ptrfdm指针返回。

如果slave_name不为空，从设备的名称被存放在该指针指向的存储区中。调用者必须为该存储区分配空间。

如果指针slave_termios不为空，该指针所引用的结构将初始化从设备的终端行规程。如果该指针为空，系统将从设备的termios结构初始化为一个由具体应用定义的初始状态。类似的，如果slave_winsize指针不为空，该指针所引用的结构将初始化从设备的窗口大小。如果该指针为空，winsize结构通常被初始化为0。

程序19-3显示了这个程序的代码。调用相应的ptym_open和ptys_open函数，这个函数在SVR4和4.3+BSD系统下都可以使用。

在打开伪终端主设备后，fork将被调用。正如前面提到的，要等到调用setid建立新的对话期后才调用ptys_open。当调用setsid时，子进程还不是一个进程组的首进程（想一想为什么？）因此9.5节列出的三个操作被使用：（a）子进程作为对话的管理者创建一个新的对话期；（b）子进程创建一个新的进程组；（c）子进程没有控制终端。在SVR4系统中，当调用ptys_open时，从设备成为了控制终端。在4.3+BSD系统中，必须调用ioctl并使用参数TIOCSCTTY来分配一个控制终端。然后termios和winsize这两个结构在子进程中被初始化。最后从设备的文件描述符被复制到子进程的标准输入、标准输出和标准出错中。这表示由子进程所exec的进程都会将上述三个句柄同伪终端从设备联系起来。

在调用fork后，父进程返回伪终端主设备的描述符并返回。下一节将在pty程序中使用pty_fork。

程序19-3 pty_fork函数

```

#include <sys/types.h>
#include <termios.h>
#ifdef TIOCGWINSZ
#include <sys/ioctl.h> /* 4.3+BSD requires this too */
#endif
#include "ourhdr.h"

pid_t
pty_fork(int *ptrfdm, char *slave_name,
        const struct termios *slave_termios,
        const struct winsize *slave_winsize)
{
    int    fdm, fds;
    pid_t  pid;
    char   pts_name[20];

    if ( (fdm = ptym_open(pts_name)) < 0)
        err_sys("can't open master pty: %s", pts_name);

    if (slave_name != NULL)
        strcpy(slave_name, pts_name); /* return name of slave */

    if ( (pid = fork()) < 0)
        return(-1);

    else if (pid == 0) { /* child */
        if (setsid() < 0)
            err_sys("setsid error");

            /* SVR4 acquires controlling terminal on open() */
        if ( (fds = ptys_open(fdm, pts_name)) < 0)
            err_sys("can't open slave pty");
        close(fdm); /* all done with master in child */

#ifdef TIOCSCTTY && !defined(CIBAUD)
            /* 4.3+BSD way to acquire controlling terminal */
            /* !CIBAUD to avoid doing this under SunOS */
        if (ioctl(fds, TIOCSCTTY, (char *) 0) < 0)
            err_sys("TIOCSCTTY error");
#endif

            /* set slave's termios and window size */
        if (slave_termios != NULL) {
            if (tcsetattr(fds, TCSANOW, slave_termios) < 0)
                err_sys("tcsetattr error on slave pty");
        }
        if (slave_winsize != NULL) {
            if (ioctl(fds, TIOCSWINSZ, slave_winsize) < 0)
                err_sys("TIOCSWINSZ error on slave pty");
        }

            /* slave becomes stdin/stdout/stderr of child */
        if (dup2(fds, STDIN_FILENO) != STDIN_FILENO)
            err_sys("dup2 error to stdin");
        if (dup2(fds, STDOUT_FILENO) != STDOUT_FILENO)
            err_sys("dup2 error to stdout");
        if (dup2(fds, STDERR_FILENO) != STDERR_FILENO)
            err_sys("dup2 error to stderr");
        if (fds > STDERR_FILENO)
            close(fds);
        return(0); /* child returns 0 just like fork() */
    } else { /* parent */
        *ptrfdm = fdm; /* return fd of master */
        return(pid); /* parent returns pid of child */
    }
}

```

19.5 pty程序

编写pty程序的目的是为了用键入：

```
pty prog arg1 arg2
```

来代替：

```
prog arg1 arg2
```

这样使我们可以用pty来执行另一个程序，该程序在一个自己的对话期中执行，并和一个伪终端连接。

看一下pty程序的源码。程序19-4包含main函数。它调用上一节的pty_fork函数。

程序19-4 pty程序的main函数

```
#include <sys/types.h>
#include <termios.h>
#ifdef TIOCGWINSZ
#include <sys/ioctl.h> /* 4.3+BSD requires this too */
#endif
#include "ourhdr.h"

static void set_noecho(int); /* at the end of this file */
void do_driver(char *); /* in the file driver.c */
void loop(int, int); /* in the file loop.c */

int
main(int argc, char *argv[])
{
    int fdm, c, ignoreeof, interactive, noecho, verbose;
    pid_t pid;
    char *driver, slave_name[20];
    struct termios orig_termios;
    struct winsize size;

    interactive = isatty(STDIN_FILENO);
    ignoreeof = 0;
    noecho = 0;
    verbose = 0;
    driver = NULL;

    opterr = 0; /* don't want getopt() writing to stderr */
    while ( (c = getopt(argc, argv, "d:einv")) != EOF) {
        switch (c) {
            case 'd': /* driver for stdin/stdout */
                driver = optarg;
                break;

            case 'e': /* noecho for slave pty's line discipline */
                noecho = 1;
                break;

            case 'i': /* ignore EOF on standard input */
                ignoreeof = 1;
                break;

            case 'n': /* not interactive */
                interactive = 0;
                break;

            case 'v': /* verbose */
                verbose = 1;
                break;

            case '?':
                err_quit("unrecognized option: -%c", optopt);
        }
    }
}
```

```

    }
}
if (optind >= argc)
    err_quit("usage: pty [ -d driver -einv ] program [ arg ... ]");
if (interactive) { /* fetch current termios and window size */
    if (tcgetattr(STDIN_FILENO, &orig_termios) < 0)
        err_sys("tcgetattr error on stdin");
    if (ioctl(STDIN_FILENO, TIOCGWINSZ, (char *) &size) < 0)
        err_sys("TIOCGWINSZ error");
    pid = pty_fork(&fdm, slave_name, &orig_termios, &size);
} else
    pid = pty_fork(&fdm, slave_name, NULL, NULL);
if (pid < 0)
    err_sys("fork error");
else if (pid == 0) { /* child */
    if (noecho)
        set_noecho(STDIN_FILENO); /* stdin is slave pty */

    if (execvp(argv[optind], &argv[optind]) < 0)
        err_sys("can't execute: %s", argv[optind]);
}
if (verbose) {
    fprintf(stderr, "slave name = %s\n", slave_name);
    if (driver != NULL)
        fprintf(stderr, "driver = %s\n", driver);
}
if (interactive && driver == NULL) {
    if (tty_raw(STDIN_FILENO) < 0) /* user's tty to raw mode */
        err_sys("tty_raw error");
    if (atexit(tty_atexit) < 0) /* reset user's tty on exit */
        err_sys("atexit error");
}
if (driver)
    do_driver(driver); /* changes our stdin/stdout */
loop(fdm, ignoreeof); /* copies stdin -> ptym, ptym -> stdout */
exit(0);
}

static void
set_noecho(int fd) /* turn off echo (for slave pty) */
{
    struct termios stermios;

    if (tcgetattr(fd, &stermios) < 0)
        err_sys("tcgetattr error");

    stermios.c_lflag &= ~(ECHO | ECHOE | ECHOK | ECHONL);
    stermios.c_oflag &= ~(ONLCR);
    /* also turn off NL to CR/NL mapping on output */

    if (tcsetattr(fd, TCSANOW, &stermios) < 0)
        err_sys("tcsetattr error");
}

```

下一节检测pty程序的不同使用时，将会探讨多种的行命令选择项。

在调用pty_fork前，我们取得了termios和winsize结构的值，将其传递给pty_fork。通过这种方法，伪终端从设备具有和现在的终端相同的初始状态。

从pty_fork返回后，子进程关闭了伪终端从设备的回显，并调用execvp来执行命令行指定的程序。所有的命令行参数将成为程序的参数。

父进程在调用exit时执行原先设置的退出处理程序，它复原终端状态，将用户终端设置为初始模式（可选）。下一节将讨论do_driver函数。

接下来父进程调用函数loop（见程序19-5）。该函数仅仅是将所有标准输入拷贝到伪终端主设备，并将伪终端主设备接收到的所有内容拷贝到标准输出。同18.7节一样，我们有两个选择——一个进程还是两个进程？为了有所区别，这里使用两个进程，尽管使用select或poll的单进程也是可行的。

程序19-5 loop函数

```
#include <sys/types.h>
#include <signal.h>
#include "ourhdr.h"

#define BUFFSIZE 512

static void sig_term(int);
static volatile sig_atomic_t sigcaught; /* set by signal handler */

void
loop(int ptym, int ignoreeof)
{
    pid_t child;
    int nread;
    char buff[BUFFSIZE];
    if ( (child = fork()) < 0 ) {
        err_sys("fork error");
    } else if (child == 0) { /* child copies stdin to ptym */
        for ( ; ; ) {
            if ( (nread = read(STDIN_FILENO, buff, BUFFSIZE)) < 0 )
                err_sys("read error from stdin");
            else if (nread == 0)
                break; /* EOF on stdin means we're done */

            if (written(ptym, buff, nread) != nread)
                err_sys("written error to master pty");
        }

        /* We always terminate when we encounter an EOF on stdin,
           but we only notify the parent if ignoreeof is 0. */
        if (ignoreeof == 0)
            kill(getppid(), SIGTERM); /* notify parent */
        exit(0); /* and terminate; child can't return */
    }

    /* parent copies ptym to stdout */
    if (signal_intr(SIGTERM, sig_term) == SIG_ERR)
        err_sys("signal_intr error for SIGTERM");

    for ( ; ; ) {
        if ( (nread = read(ptym, buff, BUFFSIZE)) <= 0 )
            break; /* signal caught, error, or EOF */

        if (written(STDOUT_FILENO, buff, nread) != nread)
            err_sys("written error to stdout");
    }

    /* There are three ways to get here: sig_term() below caught the
     * SIGTERM from the child, we read an EOF on the pty master (which
     * means we have to signal the child to stop), or an error. */
    if (sigcaught == 0) /* tell child if it didn't send us the signal */

```



```

        kill(child, SIGTERM);
    return;    /* parent returns to caller */
}

/* The child sends us a SIGTERM when it receives an EOF on
 * the pty slave or encounters a read() error. */

static void
sig_term(int signo)
{
    sigcaught = 1;    /* just set flag and return */
    return;    /* probably interrupts read() of ptym */
}

```

注意，当使用两个进程时，如果一个终止，那么它必须通知另一个。我们用 SIGTERM 进行这种通知。

19.6 使用pty程序

接下来看一下pty程序的不同例子，了解一下使用不同命令行选择项的必要性。

如果使用KornShell，我们执行：

```
pty ksh
```

得到一个运行在一个伪终端下的新的shell。

如果文件ttyname同程序11-7相同，可按如下方式执行pty程序：

```

$ who
stevens console Feb 6 10:43
stevens tty0 Feb 6 15:00
stevens tty1 Feb 6 15:00
stevens tty2 Feb 6 15:00
stevens tty3 Feb 6 15:48
stevens tty4 Feb 7 14:28
$ pty ttyname
fd 0: /dev/tty0
fd 1: /dev/tty0
fd 2: /dev/tty0

```

*tty4*是正在使用的最高终端设备
在pty上运行程序11-7
*tty5*是下一个有效的pty设备号

19.6.1 utmp文件

6.7节讨论了记录当前UNIX系统登录用户的utmp文件。那么在伪终端上运行程序的用户是否被认为登录了呢？如果是远程登录，telnetd和rlogind，显然伪终端上的用户应该在utmp中拥有相应条目。但是，从窗口系统或运行script程序，在伪终端上运行shell的用户是否应该在utmp中拥有相应条目呢？这个问题一直没有一个统一的认识。有的系统有记录，有的没有。如果没有记录的话，who(1)程序一般不会显示正在被使用的伪终端。

除非utmp允许其他用户的写许可权，否则一般的程序将不能对其进行写操作。某些系统提供这个写许可权。

19.6.2 作业控制交互

当在pty上运行作业控制shell时，它能够正常地运行。例如，

```
pty ksh
```

在pty上运行KornShell。我们能够在这个新 shell下运行程序和使用作业控制，如同在登录 shell 中一样。但如果在pty下运行一个交互式程序而不是作业控制 shell，比如：

```
pty cat
```

一切正常直到键入作业控制的暂停字符。在 SVR4和4.3+BSD系统中作业控制暂停字符将会被显示为^Z而被忽略。在SunOS4.1.2中，cat进程终止，pty进程终止，回到初始登录shell。

为了明白其中的原因，我们需要检查所有相关的进程、这些进程所属的进程组和对话期。图19-7显示了pty cat运行的结构图。

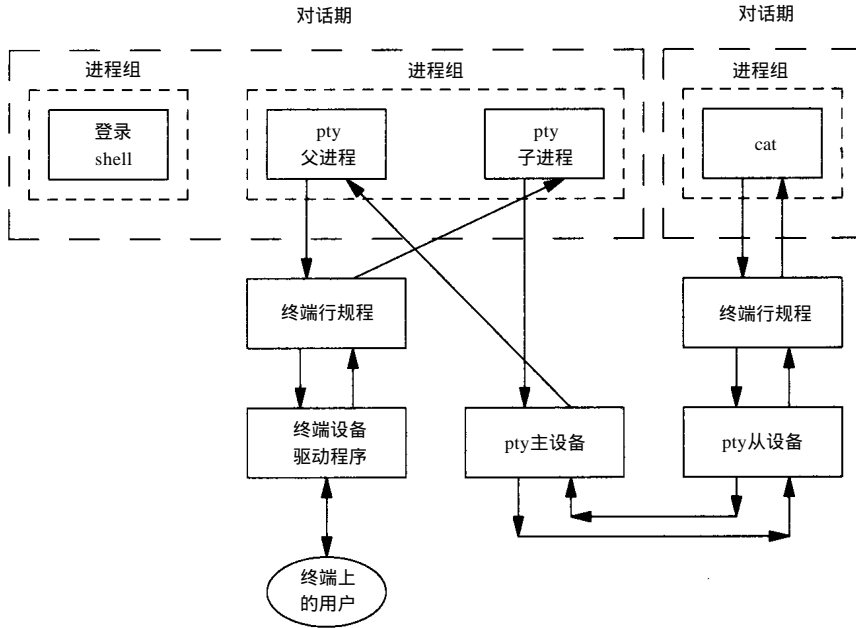


图19-7 pty cat的进程组和对话期

当键入暂停字符 (Ctrl-Z)，它将被 cat 进程下的行规程模块所识别，这是因为 pty 将终端 (在pty父进程之下) 设置为初始模式。但内核不会终止 cat 进程，这是因为它属于一个孤儿进程组 (见9.10节)。cat 的父进程是pty的父进程，属于另一个对话期。

不同的系统处理这种情况的方法也不同。在 POSIX.1 中这个 SIGTSTP 信号不被发送给进程。早期的伯克利系统递送一个进程甚至不能捕获的 SIGKILL。这就是我们在 SunOS4.1.2 中看到的。(POSIX.1 Rationale 建议用 SIGHUP 作为更好的替代，因为进程能够捕获它。)用程序 8-17 查看 cat 进程的终止状态，可以发现该进程确实由 SIGKILL 信号终止。

在 SVR4 和 4.3+BSD 系统中，我们修改了程序 10-22 来观察结果。修改后的程序能够在捕获 SIGTSTP 后进行打印，并在捕获 SIGCONT 信号后再打印一次并继续执行。这说明 SIGTSTP 被进程捕获，但是当进程试图发送信号给自己来暂停本进程的时候，内核立即发送 SIGCONT 信号使之继续执行。内核将不会让进程被作业控制停止。SVR4 和 4.3+BSD 系统的这种处理方法比起发送一个 SIGKILL 的方法来显得不那么激烈。

当使用 pty 来运行作业控制 shell 时，被这个新 shell 调用的作业将不是任何孤儿进程组的成员，这是因为作业控制 shell 总是属于同一个对话期。在这种情况下，Ctrl-Z 被发送到被 shell 调用的进程，而不是 shell 本身。

让被pty调用的进程能够处理作业控制信号的唯一的方法是：另外增加一个命令行标志让pty子进程能够自己认识作业暂停字符，而不是让该字符通过其他行规程模块。

19.6.3 检查长时间运行程序的输出

另一个使用pty进行作业控制交互的例子见图19-6。如果运行一个程序：

```
pty slowout > file.out &
```

当子进程试图从标准输入（终端）读入数据时，pty进程立刻停止运行。这是因为该作业是一个后台作业并且当它试图访问终端时会使作业控制停止。如果将标准输入重定向使得pty不从终端读取数据，如：

```
pty slowout < /dev/null > file.out &
```

那么pty程序立即终止，这是因为它从标准输入读取到一个文件结束符。解决问题的方法是使用-i选择项。这个选择项的含义是忽略来自标准输入的文件结束符：

```
pty -i slowout < /dev/null > file.out &
```

这个标志导致在遇到文件结束符时，程序19-5的子进程终止，但子进程不会使父进程也终止。相反的，父进程一直将伪终端从设备的输出拷贝到标准输出（本例中的file.out）。

19.6.4 script程序

使用pty程序，可以用下面的方式实现BSD系统中的script（1）程序。

```
#!/bin/sh
```

```
pty "${SHELL:-/bin/sh}" | tee typescript
```

一旦执行这个script程序，即可以运行ps来观察进程之间的关系。图19-8显示了这些关系。

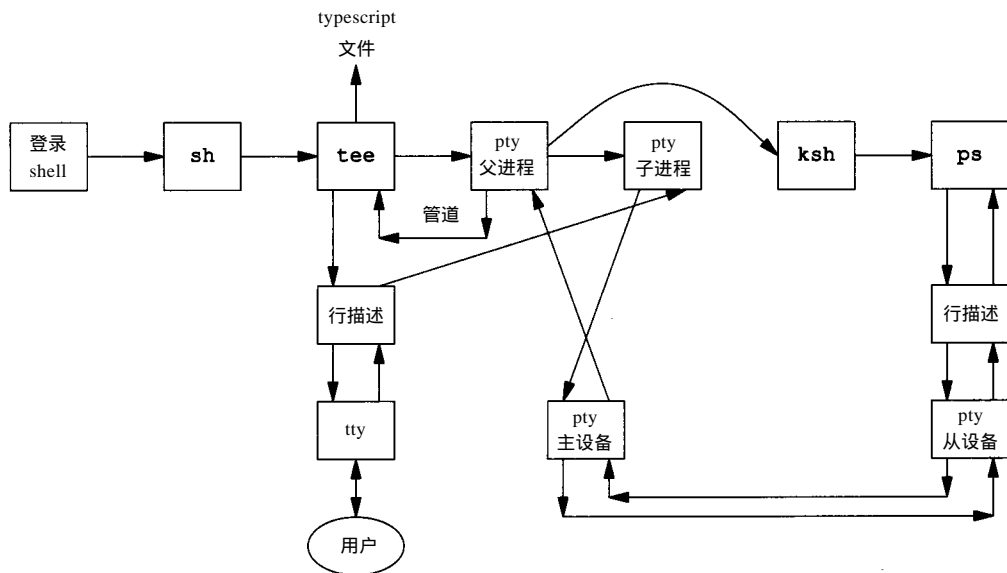


图19-8 script shell脚本

在这个例子中，假设SHELL变量是KornShell（可能是/din/ksh）。如前面所述，script仅仅是将新的shell（和它调用的所有的子进程）的输出拷贝出来，但是因为伪终端从设备上的行规程模块通常允许回显，故绝大多数键入都被写到typescript文件中去。

19.6.5 运行协同进程

在程序 14-9 中，我们不能让协同进程使用标准 I/O 函数，其原因是标准输入和输出不是终端，其输入和输出将被放到缓存中。如果用

```
if (execl("./pty", "pty", "-e", "add2", (char *) 0) < 0)
```

替代：

```
if (execl("./add2", "add2", (char *) 0) < 0)
```

在 pty 下运行协同进程，该程序即使使用了标准 I/O 仍然可以正确运行。

图 19-9 显示了在使用伪终端作为协同进程的输入和输出的情况。框中的“驱动程序”是前面提到过的改变了 execl 的程序 14-9。这是图 19-5 的一个扩充，它显示了所有的进程间联系和数据流。

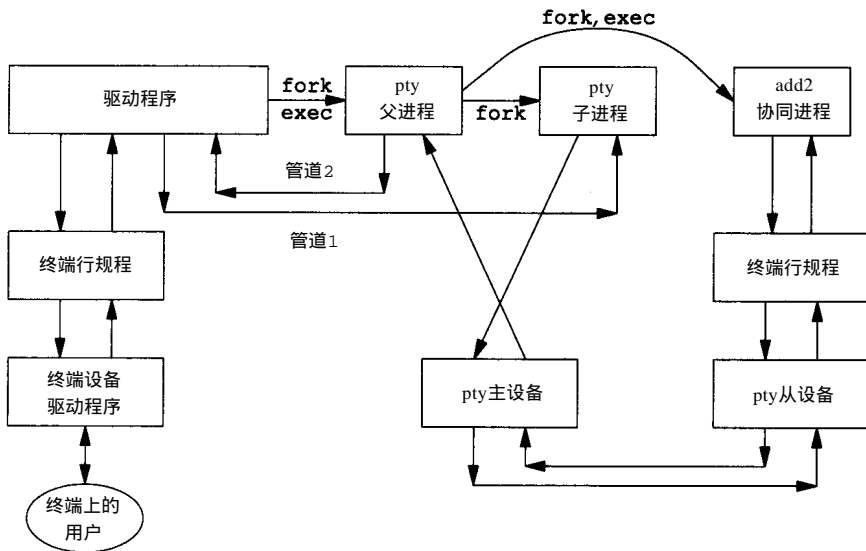


图 19-9 运行一协同进程，以 pty 作为其输入和输出

这个例子显示了对于 pty 程序 -e（不回显）选择项的重要性。pty 不以交互方式运行，这是因为它的标准输入不是一个终端。在程序 19-4 中 interactive 标志默认为 false，这是因为对 isatty 调用的返回结果是 false。这意味着在真正的终端之上的行规程保持在典型模式下并允许回显。指定 -e 选择项后，关掉了伪终端从设备上的行规程模块的回显。如果不这样做，则键入的每一个字符都将被两个行规程模块显示两次。

我们还要用 -e 选择项关闭 termios 结构的 ONLCR 标志，防止所有的协同进程的输出被回车和换行符终止。

在不同的系统上测试这个例子会遇到 12.8 节描述 readn 和 writen 函数时提到的问题。当描述符不是引用普通的磁盘文件时，从 read 返回的读取数据量可能因实现不同而有所区别。协同进程使用 pty 时，如果调用通过管道的 read 而返回结果不到一行，将输出不可预测的结果。解决的方法不是使用程序 14-9 而是使用修改过的使用标准 I/O 库的习题 14.5 的程序，将两个管道都设置为行缓存。这样 fgets 函数将会读完一个整行。程序 14-9 的 while 循环假设送到协同进程的每一行都会带来一行的返回结果。

19.6.6 用非交互模式驱动交互式程序

虽然让 pty 运行所有的协同进程是非常诱人的想法，但如果协同进程是交互式的，就不能

正常工作。问题在于pty只是将其标准输入复制到pty，并将来自pty的复制到其标准输出。而并不关心具体得到什么数据。

举个例子，我们可以在pty直接与调制解调器对话之下运行18.7节的call客户机：

```
pty call t2500
```

这样做不比直接键入call t2500有什么优点，但我们可能希望用一个shell命令运行call程序来取得调制解调器的一些内部寄存器的内容。如果t2500.cmd包括两行：

```
aatn?
```

```
~.
```

第一行打印出调制解调器的寄存器值，第二行终止call程序。但是如果运行：

```
pty -l < t2500.cmd call t2500
```

结果就不是我们希望得到的。事实上文件t2500.cmd的内容首先被送到了调制解调器。当交互运行call程序时我们等待调制解调器显示“Connected”，但是pty程序不知道这样做。这就是为什么需要比pty更巧妙的程序，如expect，自脚本文件运行交互式程序。

在pty上即使运行程序14-9也不能正常工作，这是因为程序14-9认为它在一个管道写入的每一行都会在另一个管道产生一行。而且，14-9程序总是先发送一行到系统进程，然后再读取一行。在上面的例子中，我们需要先收到行“Connected”，然后再发送数据。

这里有一些使用shell命令驱动交互式程序的方法。可以在pty上增加一种命令语言和一个解释器。但是一个适当的命令语言可能十倍于pty程序的大小。另一种方法是使用命令语言并用pty_fork函数来调用交互式程序，这正是expect程序所做的。

我们将采用一种不同的方法，使用选择项-d让pty程序同一个管理输入和输出的驱动进程连接起来。该驱动进程的标准输出是pty的标准输入，反之亦然。这有点像协同进程，只是在pty的“另一边”。此种进程结构与图19-9中所示的几乎相同，但是在这种情况下由pty来完成驱动进程的fork和exec。而且我们将在pty和驱动进程之间使用一个单独的流管道，而不是两个半双工管道。

程序19-6是do_driver函数的源码，该函数被pty（见程序19-4）的main函数在使用-d选项时调用。

程序19-6 pty程序的do_driver函数

```
#include <sys/types.h>
#include <signal.h>
#include "ourhdr.h"

void
do_driver(char *driver)
{
    pid_t    child;
    int      pipe[2];

    /* create a stream pipe to communicate with the driver */
    if (s_pipe(pipe) < 0)
        err_sys("can't create stream pipe");

    if ( (child = fork()) < 0)
        err_sys("fork error");

    else if (child == 0) {          /* child */
        close(pipe[1]);

        /* stdin for driver */
        if (dup2(pipe[0], STDIN_FILENO) != STDIN_FILENO)
```

```
err_sys("dup2 error to stdin");

    /* stdout for driver */
    if (dup2(pipe[0], STDOUT_FILENO) != STDOUT_FILENO)
        err_sys("dup2 error to stdout");
    close(pipe[0]);

    /* leave stderr for driver alone */

    execlp(driver, driver, (char *) 0);
    err_sys("execlp error for: %s", driver);
}

close(pipe[0]);    /* parent */

if (dup2(pipe[1], STDIN_FILENO) != STDIN_FILENO)
    err_sys("dup2 error to stdin");

if (dup2(pipe[1], STDOUT_FILENO) != STDOUT_FILENO)
    err_sys("dup2 error to stdout");
close(pipe[1]);

/* Parent returns, but with stdin and stdout connected
   to the driver. */
}
```

通过编写自己的驱动程序的方法，可以随意地驱动交互式程序。即使驱动程序有和 pty 连接在一起的标准输入和标准输出，它仍然可以通过 /dev/tty 同用户交互。这个解决方法仍不如 expect 程序通用，但是它提供了一种不到 50 行代码的选择方案。

19.7 其他特性

伪终端还有其他特性，我们在这里简略提一下。AT&T[1990d]和4.3+BSD系统的操作手册有更详细的内容。

19.7.1 打包模式

打包模式能够使伪终端主设备了解到伪终端从设备的状态变化。在 SVR4 系统中可以将流模块 pckt 压入主设备端来设置这种模式。图 19-2 显示了这种可选模式。在 4.3+BSD 系统中可以通过 TIOCPKT 的 ioctl 来设置这种模式。

SVR4 和 4.3+BSD 系统中具体的打包模式有所不同。在 SVR4 系统中，读取伪终端主设备的进程必须调用 getmsg 从流中取得数据，这是因为 pckt 模块将一些事件转化为无数据的流消息。在 4.3+BSD 系统中每一次从伪终端主设备的读操作都会在可选数据之后返回状态字节。

无论实现的方法是什么样的，打包模式的目的是，当伪终端从设备之上的行规程模块出现以下事件时，通知进程从伪终端主设备读取数据：读入队列被刷新；写出队列被刷新；输出被停止（如：Ctrl-S）；输出重新开始；XON/XOFF 流开关被关闭后重新打开；XON/XOFF 流开关被打开后重新关闭。这些事件被 rlogin 客户机和 rlogid 服务器等使用。

19.7.2 远程模式

伪终端主设备可以用 TIOCREMOTE 的 ioctl 将伪终端从设备设置成远程模式。虽然 SVR4 和 4.3+BSD 系统使用同样的命令来打开或关闭这个特性，但是在 SVR4 系统中 ioctl 的第三个参数是一个整型数，而 4.3+BSD 中是一个指向整型数的指针。

当伪终端主设备将伪终端从设备设置成这种模式时，它通知伪终端从设备之上的行规程模块对从主设备收到的任何数据都不要进行处理，无论它是不是从设备的 `termios` 结构的规范或非规范标志。远程模式适用于窗口管理器这种进行自己的行编辑的应用模式。

19.7.3 窗口大小变化

伪终端主设备上的进程可以用 `TIOCSWINSZ` 的 `ioctl` 来设置从设备的窗口大小。如果新的大小和老的不同，一个 `SIGWINCH` 信号将被发送到伪终端从设备的前台进程组。

19.7.4 信号发生

读写伪终端主设备的进程可以向伪终端从设备的进程组发送信号。在 SVR4 系统中，可以通过 `TIOCSIGNAL` 的 `ioctl` 完成这个功能，第三个参数就是信号的数值。在 4.3+BSD 中通过 `TIOCSIG` 的 `ioctl` 来完成，第三个参数就是指向信号编号值的指针。

19.8 小结

本章首先说明了在 SVR4 和 4.3+BSD 系统中打开伪终端的代码。然后用此代码提供了用于多种不同应用的通用的 `pty_fork` 函数。这个函数是小程序（`pty`）的基础。并且讨论了许多伪终端的属性。

伪终端在大多数 UNIX 系统中每天都被用来进行网络登录。我们检查了伪终端的许多其他用途，从 `script` 程序到使用批处理脚本来驱动交互式程序。

习题

19.1 当用 `telnet` 或 `rlogin` 远程登录到一个 BSD 系统上时，像我们在 19.3.2 节讨论过的那样，伪终端从设备的所有权和许可权被设置。该过程是如何发生的？

19.2 修改 4.3+BSD 系统中的 `ptys_open` 函数，使之调用一个设置 - 用户 - ID 程序来改变伪终端从设备的所有权和许可权（像 SVR4 系统中的 `grantpt` 函数所做的）。

19.3 使用 `pty` 程序来决定你的系统初始化 `termios` 结构和 `winsize` 结构的值。

19.4 重写 `loop` 函数（见程序 19-5），使之成为一个使用 `select` 或 `poll` 的单个进程。

19.5 在子进程中，`pty_fork` 返回后，标准输入、标准输出和标准出错都以读写方式打开。能够将标准输入变成只读，另两个变成只写吗？

19.6 在图 19-7 中，指出哪个进程组是前台的，哪个进程组是后台的，并指出对话期管理者。

19.7 在图 19-7 中，当键入文件终止符时，进程终止的顺序是什么？如果可能的话，用进程计数来修改之。

19.8 `script(1)` 程序通常开始时在输出文件头增加一行，结束时在输出文件末尾增加一行。将这个特性加到本章简单的 shell 脚本中。

19.9 解释为什么在下面的例子中，文件 `data` 的内容被输出到终端上，而程序 `ttyname` 只产生输出而从不读取输入。

```
$ cat data                                一个有两行的文件
hello,
world
$ pty -i < data ttyname                   -i 忽略stdin的文件结束标志
hello,                                     这两行来自何处？
```

```
world
fd 0: /dev/ttyp5
fd 1: /dev/ttyp5
fd 2: /dev/ttyp5
```

我们期望ttyname输出这三行

19.10 写一个调用pty_fork的程序，该程序有一个子进程，该子进程 exec另一个要求你编写的程序。子进程调用的新的程序能够捕获 SIGTERM和SIGWINCH。当捕获到消息时，该程序要打印出来，并且对于后一种消息，还要打印终端窗口大小。然后让父进程向 19.7节描述过的，有ioctl的伪终端从设备的进程组发送 SIGTERM消息。从伪终端从设备读回消息验证捕获的消息。接下来用父进程设置伪终端从设备窗口的大小，并读回伪终端从设备的输出。让父进程退出并确定是否要伪终端从设备进程也终止，如果要终止，应如何终止？

附录A 函数原型

本附录包含了正文中说明过的标准 UNIX、POSIX和ANSI C的函数原型。通常我们想了解的是函数的参数 (fgets的哪一个参数是文件指针 ?)和返回值 (sprintf返回的是指针还是计数值 ?)。

这些函数原型还说明了要包含哪些头文件，以获得特定常数的定义，或获得 ANSI C函数原型，以帮助在编译时进行错误检测。

```
void    _exit(int status);
        <unistd.h>
        此函数不返回

void    abort(void);
        <stdlib.h>
        此函数不返回

int     access(const char *pathname, int mode);
        <unistd.h>
        mode: R_OK, W_OK, X_OK, F_OK
        返回：若成功则为 0，若出错则为 - 1

unsigned
int     alarm(unsigned int seconds);
        <unistd.h>
        返回：0或上次设置的 alarm的剩余秒数

char    *asctime(const struct tm *tm_ptr);
        <time.h>
        返回：指向以 null终止的字符串的指针

int     atexit(void (*func)(void));
        <stdlib.h>
        返回：若成功则为 0，若出错则为非 0

void    *calloc(size_t nobj, size_t size);
        <stdlib.h>
        返回：若成功则为非空指针，若出错则为 NULL

speed_t cfgetispeed(const struct termios *term_ptr);
        <termios.h>
        返回：波特率值

speed_t cfgetospeed(const struct termios *term_ptr);
        <termios.h>
        返回：波特率值

int     cfsetispeed(struct termios *term_ptr, speed_t speed);
        <termios.h>
        返回：若成功则为 0，若出错则为 - 1

int     cfsetospeed(struct termios *term_ptr, speed_t speed);
        <termios.h>
        返回：若成功则为 0，若出错则为 - 1

int     chdir(const char *pathname);
        <unistd.h>
        返回：若成功则为 0，若出错则为 - 1
```

```
int      chmod(const char *pathname, mode_t mode);
          <sys/types.h>
          <sys/stat.h>
          mode: S_IS[UG]ID, S_ISVTX, S_I[RWX] (USR|GRP|OTH)
          返回: 若成功则为0, 若出错则为 - 1

int      chown(const char *pathname, uid_t owner, gid_t group);
          <sys/types.h>
          <unistd.h>
          返回: 若成功则为0, 若出错则为 - 1

void     clearerr(FILE *fp);
          <stdio.h>

int      close(int fildes);
          <unistd.h>
          返回: 若成功则为0, 若出错则为 - 1

int      closedir(DIR *dp);
          <sys/types.h>
          <dirent.h>
          返回: 若成功则为0, 若出错则为 - 1

void     closelog(void);
          <syslog.h>

int      creat(const char *pathname, mode_t mode);
          <sys/types.h>
          <sys/stat.h>
          <fcntl.h>
          mode: S_IS[UG]ID, S_ISVTX, S_I[RWX] (USR|GRP|OTH)
          返回: 若成功则为只写打开的文件描述符, 若出错则为 - 1

char     *ctermid(char *ptr);
          <stdio.h>
          返回: 控制终端的路径名

char     *ctime(const time_t *calptr);
          <time.h>
          返回: 指向以null终止的字符串的指针

int      dup(int fildes);
          <unistd.h>
          返回: 若成功则为新文件描述符, 若出错则为 - 1

int      dup2(int fildes, int fildes2);
          <unistd.h>
          返回: 若成功则为新文件描述符, 若出错则为 - 1

void     endgrent(void);
          <sys/types.h>
          <grp.h>

void     endpwent(void);
          <sys/types.h>
          <pwd.h>

int      execl(const char *pathname, const char *arg0, ... /* (char *) 0 */);
          <unistd.h>
          返回: 若出错则为 - 1, 若成功则无返回

int      execlp(const char *pathname, const char *arg0, ... /* (char *) 0,
          char *const envp[] */);
          <unistd.h>
          返回: 若出错则为 - 1, 若成功则无返回

int      execlp(const char *filename, const char *arg0, ... /* (char *) 0 */);
          <unistd.h>
          返回: 若出错则为 - 1, 若成功则无返回
```

```

int     execv(const char *pathname, char *const argv[]);
        <unistd.h>
        返回：若出错则为 - 1，若成功则无返回

int     execve(const char *pathname, char *const argv[], char *const envp[]);
        <unistd.h>
        返回：若出错则为 - 1，若成功则无返回

int     execvp(const char *filename, char *const argv[]);
        <unistd.h>
        返回：若出错则为 - 1，若成功则无返回

void    exit(int status);
        <stdlib.h>
        无返回

int     fchdir(int filedes);
        <unistd.h>
        返回：若成功则为 0，若出错则为 - 1

int     fchmod(int filedes, mode_t mode);
        <sys/types.h>
        <sys/stat.h>
        mode: S_IS[UG]ID, S_ISVTX, S_I[RWX] (USR|GRP|OTH)
        返回：若成功则为 0，若出错则为 - 1

int     fchown(int filedes, uid_t owner, gid_t group);
        <sys/types.h>
        <unistd.h>
        返回：若成功则为 0，若出错则为 - 1

int     fclose(FILE *fp);
        <stdio.h>
        返回：若成功则为 0，若出错则为 - 1

int     fcntl(int filedes, int cmd, ... /* int arg */);
        <sys/types.h>
        <unistd.h>
        <fcntl.h>
        cmd: F_DUPFD, F_GETFD, F_SETFD, F_GETFL, F_SETFL
        返回：若成功则取决于 cmd，若出错则为 - 1

FILE    *fdopen(int filedes, const char *type);
        <stdio.h>
        type: "r", "w", "a", "r+", "w+", "a+",
        返回：若成功则为文件指针，若出错则为 NULL

int     feof(FILE *fp);
        <stdio.h>
        返回：若已至流的文件尾端则为非 0 值(真)，否则为 0 假)

int     ferror(FILE *fp);
        <stdio.h>
        返回：若流出错则为非 0 值(真)，否则为 0 假)

int     fflush(FILE *fp);
        <stdio.h>
        返回：若成功则为 0，若出错则为 EOF

int     fgetc(FILE *fp);
        <stdio.h>
        返回：若成功则为下一个字符，若已至文件尾端或出错则为 EOF

int     fgetpos(FILE *fp, fpos_t *pos);
        <stdio.h>
        返回：若成功则为 0，若出错则为非 0

```

```

char *fgets(char *buf, int n, FILE *fp);
    <stdio.h>
    返回：若成功则为 buf，若已至文件尾端或出错则为 NULL

int fileno(FILE *fp);
    <stdio.h>
    返回：与该流相结合的文件描述符

FILE *fopen(const char *pathname, const char *type);
    <stdio.h>
    type: "r", "w", "a", "r+", "w+", "a+",
    返回：若成功则为文件指针，若出错则为 NULL

pid_t fork(void);
    <sys/types.h>
    <unistd.h>
    返回：子进程中为 0，父进程中为子进程 ID，若出错则为 - 1

long fpathconf(int filedes, int name);
    <unistd.h>
    name: _PC_CHOWN_RESTRICTED, _PC_LINK_MAX, _PC_MAX_CANON,
          _PC_MAX_INPUT, _PC_NAME_MAX, _PC_NO_TRUNC,
          _PC_PATH_MAX, _PC_PIPE_BUF, _PC_VDISABLE
    返回：若成功则为相应值，若出错则为 - 1

int fprintf(FILE *fp, const char *format, ...);
    <stdio.h>
    返回：若成功则为已输出的字符数，若出错则为负值

int fputc(int c, FILE *fp);
    <stdio.h>
    返回：若成功则为 c，若出错则为 EOF

int fputs(const char *str, FILE *fp);
    <stdio.h>
    返回：若成功则为非负值，若出错则为 EOF

size_t fread(void *ptr, size_t size, size_t nobj, FILE *fp);
    <stdio.h>
    返回：读到的对象数

void free(void *ptr);
    <stdlib.h>

FILE *freopen(const char *pathname, const char *type, FILE *fp);
    <stdio.h>
    type: "r", "w", "a", "r+", "w+", "a+",
    返回：若成功则为文件指针，若出错则为 NULL

int fscanf(FILE *fp, const char *format, ...);
    <stdio.h>
    返回：赋值输入项数，若在任一变换前输入出错或 EOF，则返回 EOF

int fseek(FILE *fp, long offset, int whence);
    <stdio.h>
    whence: SEEK SET, SEEK CUR, SEEK_END
    返回：若成功则为 0，若出错则为非 0

int fsetpos(FILE *fp, const fpos_t *pos);
    <stdio.h>
    返回：若成功则为 0，若出错则为非 0

int fstat(int filedes, struct stat *buf);
    <sys/types.h>
    <sys/stat.h>
    返回：若成功则为 0，若出错则为 - 1

int fsync(int filedes);
    <unistd.h>
    返回：若成功则为 0，若出错则为 - 1

```

```

long    ftell(FILE *fp);
        <stdio.h>
        返回：若成功则为当前文件位置指示器，若出错则为 - 1 L

int     ftruncate(int filedes, off_t length);
        <sys/types.h>
        <unistd.h>
        返回：若成功则为 0，若出错则为 - 1

size_t  fwrite(const void *ptr, size_t size, size_t nobj, FILE *fp);
        <stdio.h>
        返回：写的对象数

int     getc(FILE *fp);
        <stdio.h>
        返回：若成功则为下一个字符，若已至文件尾端或出错则为 EOF

int     getchar(void);
        <stdio.h>
        返回：若成功则为下一个字符，若已至文件尾端或出错则为 EOF

char    *getcwd(char *buf, size_t size);
        <unistd.h>
        返回：若成功则为 buf，若出错则为 NULL

gid_t   getegid(void);
        <sys/types.h>
        <unistd.h>
        返回：调用进程的有效组 I D

char    *getenv(const char *name);
        <stdlib.h>
        返回：与 name 相关连的 value 的指针，若没有找到，则为 NULL

uid_t   geteuid(void);
        <sys/types.h>
        <unistd.h>
        返回：调用进程的有效用户 I D

gid_t   getgid(void);
        <sys/types.h>
        <unistd.h>
        返回：调用进程的实际组 I D

struct  group *getgrent(void);
        <sys/types.h>
        <grp.h>
        返回：若成功则为指针，若已至文件尾端或出错则为 NULL

struct  group *getgrgid(gid_t gid);
        <sys/types.h>
        <grp.h>
        返回：若成功则为指针，若出错则为 NULL

struct  group *getgrnam(const char *name);
        <sys/types.h>
        <grp.h>
        返回：若成功则为指针，若出错则为 NULL

int     getgroups(int gidsetsize, gid_t grouplist[]);
        <sys/types.h>
        <unistd.h>
        返回：若成功则为添加组 I 数，若出错则为 - 1

int     gethostname(char *name, int namelen);
        <unistd.h>
        返回：若成功则为 0，若出错则为 - 1

```

```

char    *getlogin(void);
        <unistd.h>
        返回：若成功则为login名字字符串的指针，若出错则为NULL

int     getmsg(int filedes, struct strbuf *ctlptr, struct strbuf *dataptr, int *flagptr);
        <stropts.h>
        *flagptr: 0, RS_HIPRI
        返回：若成功则为非负值，若出错则为-1

pid_t   getpgrp(void);
        <sys/types.h>
        <unistd.h>
        返回：调用进程的进程组ID

pid_t   getpid(void);
        <sys/types.h>
        <unistd.h>
        返回：调用进程的进程ID

int     getpmsg(int filedes, struct strbuf *ctlptr, struct strbuf *dataptr, int *bandptr,
               int *flagptr);
        <stropts.h>
        *flagptr: 0, MSG_HIPRI, MSG_BAND, MSG_ANY
        返回：若成功则为非负值，若出错则为-1

pid_t   getppid(void);
        <sys/types.h>
        <unistd.h>
        返回：调用进程的父进程ID

struct  passwd *getpwent(void);
        <sys/types.h>
        <pwd.h>
        返回：若成功则为指针，若已至文件尾端或出错则为NULL

struct  passwd *getpwnam(const char *name);
        <sys/types.h>
        <pwd.h>
        返回：若成功则为指针，若出错则为NULL

struct  passwd *getpwuid(uid_t uid);
        <sys/types.h>
        <pwd.h>
        返回：若成功则为指针，若出错则为NULL

int     getrlimit(int resource, struct rlimit *rlptr);
        <sys/time.h>
        <sys/resource.h>
        返回：若成功则为0，若出错则为非0

char    *gets(char *buf);
        <stdio.h>
        返回：若成功则为buf，若已至文件尾或出错则为NULL

uid_t   getuid(void);
        <sys/types.h>
        <unistd.h>
        返回：调用进程的实际用户ID

struct  tm *gmtime(const time_t *calptr);
        <time.h>
        返回：指向一时间结构的指针

int     initgroups(const char *username, gid_t basegid);
        <sys/types.h>
        <unistd.h>
        返回：若成功则为0，若出错则为-1

```

```

int      ioctl(int filedes, int request, ...);
          <unistd.h>      /* SVR4 */
          <sys/ioctl.h>   /* 4.3+BSD */
          返回：若出错则为 - 1，若成功则为其他

int      isastream(int filedes);
          返回：若为流设备则为 1（真），否则为 0（假）

int      isatty(int filedes);
          <unistd.h>
          返回：若为终端设备则为 1（真），否则为 0（假）

int      kill(pid_t pid, int signo);
          <sys/types.h>
          <signal.h>
          返回：若成功则为 0，若出错则为 - 1

int      lchown(const char *pathname, uid_t owner, gid_t group);
          <sys/types.h>
          <unistd.h>
          返回：若成功则为 0，若出错则为 - 1

int      link(const char *existingpath, const char *newpath);
          <unistd.h>
          返回：若成功则为 0，若出错则为 - 1

struct
tm      *localtime(const time_t *calptr);
          <time.h>
          返回：指向一时间结构的指针

void     longjmp(jmp_buf env, int val);
          <setjmp.h>
          不返回

off_t    lseek(int filedes, off_t offset, int whence);
          <sys/types.h>
          <unistd.h>
          whence: SEEK_SET, SEEK_CUR, SEEK_END
          返回：若成功则为新的文件位移，若出错则为 - 1

int      lstat(const char *pathname, struct stat *buf);
          <sys/types.h>
          <sys/stat.h>
          返回：若成功则为 0，若出错则为 - 1

void     *malloc(size_t size);
          <stdlib.h>
          返回：若成功则为非空指针，若出错则为 NULL

int      mkdir(const char *pathname, mode_t mode);
          <sys/types.h>
          <sys/stat.h>
          mode: S_IS[UG]ID, S_ISVTX, S_I[RWX] (USR|GRP|OTH)
          返回：若成功则为 0，若出错则为 - 1

int      mkfifo(const char *pathname, mode_t mode);
          <sys/types.h>
          <sys/stat.h>
          mode: S_IS[UG]ID, S_ISVTX, S_I[RWX] (USR|GRP|OTH)
          返回：若成功则为 0，若出错则为 - 1

time_t   mktime(struct tm *imptr);
          <time.h>
          返回：若成功则为日历时间，若出错则为 - 1

caddr_t  mmap(caddr_t addr, size_t len, int prot, int flag, int filedes, off_t off);
          <sys/types.h>
          <sys/mman.h>
          prot: PROT_READ, PROT_WRITE, PROT_EXEC, PROT_NONE
          flag: MAP_FIXED, MAP_SHARED, MAP_PRIVATE
          返回：若成功则为映射区的起始地址，若出错则为 - 1

```

```

int      msgctl(int msqid, int cmd, struct msqid_ds *buf);
        <sys/types.h>
        <sys/ipc.h>
        <sys/msg.h>
        cmd: IPC_STAT, IPC_SET, IPC_RMID
        返回: 若成功则为0, 若出错则为 - 1

int      msgget(key_t key, int flag);
        <sys/types.h>
        <sys/ipc.h>
        <sys/msg.h>
        flag: 0, IPC_CREAT, IPC_EXCL
        返回: 若成功则为消息队列 ID, 若出错则为 - 1

int      msgrcv(int msqid, void *ptr, size_t nbytes, long type, int flag);
        <sys/types.h>
        <sys/ipc.h>
        <sys/msg.h>
        flag: 0, IPC_NOWAIT, MSG_NOERROR
        返回: 若成功则为消息数据部分的长度, 若出错则为 - 1

int      msgsnd(int msqid, const void *ptr, size_t nbytes, int flag);
        <sys/types.h>
        <sys/ipc.h>
        <sys/msg.h>
        flag: 0, IPC_NOWAIT
        返回: 若成功则为0, 若出错则为 - 1

int      mmap(caddr_t addr, size_t len);
        <sys/types.h>
        <sys/mman.h>
        返回: 若成功则为0, 若出错则为 - 1

int      open(const char *pathname, int oflag, ... /*, mode_t mode */);
        <sys/types.h>
        <sys/stat.h>
        <fcntl.h>
        oflag: O_RDONLY, O_WRONLY, O_RDWR;
              O_APPEND, O_CREAT, O_EXCL, O_TRUNC,
              O_NOCTTY, O_NONBLOCK, O_SYNC
        mode: S_IS[UG]ID, S_ISVTX, S_I[RWX](USR|GRP|OTH)
        返回: 若成功则为文件描述符, 若出错则为 - 1

DIR      *opendir(const char *pathname);
        <sys/types.h>
        <dirent.h>
        返回: 若成功则为指针, 若出错则为 NULL

void     openlog(char *ident, int option, int facility);
        <syslog.h>
        option: LOG_CONS, LOG_NDELAY, LOG_PERROR, LOG_PID
        facility: LOG_AUTH, LOG_CRON, LOG_DAEMON, LOG_KERN,
                 LOG_LOCAL[0-7], LOG_LPR, LOG_MAIL, LOG_NEWS,
                 LOG_SYSLOG, LOG_USER, LOG_UUCP

long     pathconf(const char *pathname, int name);
        <unistd.h>
        name: _PC_CHOWN_RESTRICTED, _PC_LINK_MAX, _PC_MAX_CANON,
              _PC_MAX_INPUT, _PC_NAME_MAX, _PC_NO_TRUNC,
              _PC_PATH_MAX, _PC_PIPE_BUF, _PC_VDISABLE
        返回: 若成功则为相应值, 若出错则为 - 1

int      pause(void);
        <unistd.h>
        返回: - 1, errno设置为EINTR

int      pclose(FILE *fp);
        <stdio.h>
        返回: cmdstring的终止状态, 若出错则为 - 1

```



```

void    perror(const char *msg);
        <stdio.h>

int     pipe(int fildes[2]);
        <unistd.h>
        返回：若成功则为 0，若出错则为 - 1

int     poll(struct pollfd fdarray[], unsigned long nfds, int timeout);
        <stropts.h>
        <poll.h>
        返回：准备就绪的描述符数，超时返回 0，若出错则返回 - 1

FILE    *popen(const char *cmdstring, const char *type);
        <stdio.h>
        type: "r", "w"
        返回：若成功则为文件指针，若出错则为 NULL

int     printf(const char *format, ...);
        <stdio.h>
        返回：若成功则为输出的字符数，若输出错则返回负值

void    psignal(int signo, const char *msg);
        <signal.h>

int     putc(int c, FILE *fp);
        <stdio.h>
        返回：若成功则为 c，若出错则为 EOF

int     putchar(int c);
        <stdio.h>
        返回：若成功则为 c，若出错则为 EOF

int     putenv(const char *str);
        <stdlib.h>
        返回：若成功则为 0，若出错则为非 0

int     putmsg(int fildes, const struct strbuf *ctlptr, const struct strbuf *dataptr,
               int flag);
        <stropts.h>
        flag: 0, RS_HIPRI
        返回：若成功则为 0，若出错则为 - 1

int     putpmsg(int fildes, const struct strbuf *ctlptr, const struct strbuf *dataptr,
                int band, int flag);
        <stropts.h>
        flag: 0, MSG_HIPRI, MSG_BAND
        返回：若成功则为 0，若出错则为 - 1

int     puts(const char *str);
        <stdio.h>
        返回：若成功则为非负值，若出错则为 EOF

int     raise(int signo);
        <sys/types.h>
        <signal.h>
        返回：若成功则为 0，若出错则为 - 1

ssize_t read(int fildes, void *buff, size_t nbytes);
        <unistd.h>
        返回：若成功则为读到的字节数，若已至文件尾端则为 0，若出错则为 -

struct
dirent *readdir(DIR *dp);
        <sys/types.h>
        <diret.h>
        返回：若成功则为指针，若出错则为 NULL

int     readlink(const char *pathname, char *buf, int bufsize);
        <unistd.h>
        返回：若成功则为读到的字节数，若出错则为 - 1

```

```

ssize_t  readv(int filedes, const struct iovec iov[], int iovcnt);
          <sys/types.h>
          <sys/uio.h>
          返回：若成功则为读到的字节数，若出错则为 - 1

void     *realloc(void *ptr, size_t newsize);
          <stdlib.h>
          返回：若成功则为非空指针，若出错则为 NULL

int      remove(const char *pathname);
          <stdio.h>
          返回：若成功则为 0，若出错则为 - 1

int      rename(const char *oldname, const char *newname);
          <stdio.h>
          返回：若成功则为 0，若出错则为 - 1

void     rewind(FILE *fp);
          <stdio.h>

void     rewinddir(DIR *dp);
          <sys/types.h>
          <dirent.h>

int      rmdir(const char *pathname);
          <unistd.h>
          返回：若成功则为 0，若出错则为 - 1

int      scanf(const char *format, ...);
          <stdio.h>
          返回：赋值的输入项数，若在任一变换前输入出错或 EOF 则返回 EOF

int      select(int maxfdp1, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
                struct timeval *tvp);
          <sys/types.h>
          <sys/time.h>
          <unistd.h>
          返回：准备就绪的描述符数，超时返回 0，若出错则为 - 1
          FD_ZERO(fd_set *fdset);
          FD_SET(int filedes, fd_set *fdset);
          FD_CLR(int filedes, fd_set *fdset);
          FD_ISSET(int filedes, fd_set *fdset);

int      semctl(int semid, int semnum, int cmd, union semun arg);
          <sys/types.h>
          <sys/ipc.h>
          <sys/sem.h>
          cmd: IPC_STAT, IPC_SET, IPC_RMID, GETPID, GETNCNT, GETZCNT,
               GETVAL, SETVAL, GETALL, SETALL
          返回：(与命令有关)

int      semget(key_t key, int nsems, int flag);
          <sys/types.h>
          <sys/ipc.h>
          <sys/sem.h>
          flag: 0, IPC_CREAT, IPC_EXCL
          返回：若成功则为信号量 ID，若出错则为 - 1

int      semop(int semid, struct sembuf semoparray[], size_t nops);
          <sys/types.h>
          <sys/ipc.h>
          <sys/sem.h>
          返回：若成功则为 0，若出错则为 - 1

void     setbuf(FILE *fp, char *buf);
          <stdio.h>

int      setegid(gid_t gid);
          <sys/types.h>
          <unistd.h>
          返回：若成功则为 0，若出错则为 - 1

```

```

int      setenv(const char *name, const char *value, int rewrite);
          <stdlib.h>
          返回：若成功则为0，若出错则为非0值

int      seteuid(uid_t uid);
          <sys/types.h>
          <unistd.h>
          返回：若成功则为0，若出错则为-1

int      setgid(gid_t gid);
          <sys/types.h>
          <unistd.h>
          返回：若成功则为0，若出错则为-1

void     setgrent(void);
          <sys/types.h>
          <grp.h>

int      setgroups(int ngroups, const gid_t grouplist[]);
          <sys/types.h>
          <unistd.h>
          返回：若成功则为0，若出错则为-1

int      setjmp(jmp_buf env);
          <setjmp.h>
          返回：若被直接调用则为0，若从longjmp返回则为非0值

int      setpgid(pid_t pid, pid_t pgid);
          <sys/types.h>
          <unistd.h>
          返回：若成功则为0，若出错则为-1

void     setpwent(void);
          <sys/types.h>
          <pwd.h>

int      setregid(gid_t rgid, gid_t egid);
          <sys/types.h>
          <unistd.h>
          返回：若成功则为0，若出错则为-1

int      setreuid(uid_t ruid, uid_t euid);
          <sys/types.h>
          <unistd.h>
          返回：若成功则为0，若出错则为-1

int      setrlimit(int resource, const struct rlimit *rlptr);
          <sys/time.h>
          <sys/resource.h>
          返回：若成功则为0，若出错则为非0值

pid_t    setsid(void);
          <sys/types.h>
          <unistd.h>
          返回：若成功则为进程组ID，若出错则为-1

int      setuid(uid_t uid);
          <sys/types.h>
          <unistd.h>
          返回：若成功则为0，若出错则为-1

int      setvbuf(FILE *fp, char *buf, int mode, size_t size);
          <stdio.h>
          mode: _IOFBF, _IOLBF, _IONBF
          返回：若成功则为0，若出错则为非0值

void     *shmat(int shmid, void *addr, int flag);
          <sys/types.h>
          <sys/ipc.h>
          <sys/shm.h>
          flag: 0, SHM_RND, SHM_RDONLY
          返回：若成功则为共享存储段指针，若出错则为-1

```

```

int      shmctl(int shmid, int cmd, struct shmid_ds *buf);
        <sys/types.h>
        <sys/ipc.h>
        <sys/shm.h>
        cmd: IPC_STAT, IPC_SET, IPC_RMID,
              SHM_LOCK, SHM_UNLOCK
        返回: 若成功则为0, 若出错则为-1

int      shmdt(void *addr);
        <sys/types.h>
        <sys/ipc.h>
        <sys/shm.h>
        返回: 若成功则为0, 若出错则为-1

int      shmget(key_t key, int size, int flag);
        <sys/types.h>
        <sys/ipc.h>
        <sys/shm.h>
        flag: 0, IPC_CREAT, IPC_EXCL
        返回: 若成功则为共享存储段ID, 若出错则为-1

int      sigaction(int signo, const struct sigaction *act, struct sigaction *oact);
        <signal.h>
        返回: 若成功则为0, 若出错则为-1

int      sigaddset(sigset_t *set, int signo);
        <signal.h>
        返回: 若成功则为0, 若出错则为-1

int      sigdelset(sigset_t *set, int signo);
        <signal.h>
        返回: 若成功则为0, 若出错则为-1

int      sigemptyset(sigset_t *set);
        <signal.h>
        返回: 若成功则为0, 若出错则为-1

int      sigfillset(sigset_t *set);
        <signal.h>
        返回: 若成功则为0, 若出错则为-1

int      sigismember(const sigset_t *set, int signo);
        <signal.h>
        返回: 若真则为1, 若假则为0

void     siglongjmp(sigjmp_buf env, int val);
        <setjmp.h>
        不返回

void     (*signal(int signo, void (*func)(int)))(int);
        <signal.h>
        返回: 信号的以前配置, 若出错则为SIG_ERR

int      sigpending(sigset_t *set);
        <signal.h>
        返回: 若成功则为0, 若出错则为-1

int      sigprocmask(int how, const sigset_t *set, sigset_t *oset);
        <signal.h>
        how: SIG_BLOCK, SIG_UNBLOCK, SIG_SETMASK
        返回: 若成功则为0, 若出错则为-1

int      sigsetjmp(sigjmp_buf env, int savemask);
        <setjmp.h>
        返回: 若直接调用则为0, 若从siglongjmp调用返回则为非0

int      sigsuspend(const sigset_t *sigmask);
        <signal.h>
        返回: -1, errno设置为EINTR

```

```

unsigned
int      sleep(unsigned int seconds);
        <unistd.h>
        返回: 0或未睡足的秒数

int      sprintf(char *buf, const char *format, ...);
        <stdio.h>
        返回: 存入数组中的字符数

int      sscanf(const char *buf, const char *format, ...);
        <stdio.h>
        返回: 赋值的输入项数, 若在任一变换前输入出错或 EOF则为 EOF

int      stat(const char *pathname, struct stat *buf);
        <sys/types.h>
        <sys/stat.h>
        返回: 若成功则为 0, 若出错则为 - 1

char     *strerror(int errnum);
        <string.h>
        返回: 消息字符串指针

size_t   strftime(char *buf, size_t maxsize, const char *format, const struct tm *tmpr);
        <time.h>
        返回: 如有空间为存入数组的字符数, 否则为 0

int      symlink(const char *actualpath, const char *sympath);
        <unistd.h>
        返回: 若成功则为 0, 若出错则为 - 1

void     sync(void);
        <unistd.h>

long     sysconf(int name);
        <unistd.h>
        name: _SC_ARG_MAX, _SC_CHILD_MAX, _SC_CLK_TCK,
              _SC_NGROUPS_MAX, _SC_OPEN_MAX, _SC_PASS_MAX,
              _SC_STREAM_MAX, _SC_TZNAME_MAX, _SC_JOB_CONTROL,
              _SC_SAVED_IDS, _SC_VERSION, _SC_XOPEN_VERSION
        返回: 若成功则为对应的值, 若出错则为 - 1

void     syslog(int priority, char *format, ...);
        <syslog.h>

int      system(const char *cmdstring);
        <stdlib.h>
        返回: shell的终止状态

int      tcdrain(int filedes);
        <termios.h>
        返回: 若成功则为 0, 若出错则为 - 1

int      tcflow(int filedes, int action);
        <termios.h>
        action: TCOOFF, TCOON, TCIOFF, TCION
        返回: 若成功则为 0, 若出错则为 - 1

int      tcflush(int filedes, int queue);
        <termios.h>
        queue: TCIFLUSH, TCOFLUSH, TCIOFLUSH
        返回: 若成功则为 0, 若出错则为 - 1

int      tcgetattr(int filedes, struct termios *termpr);
        <termios.h>
        返回: 若成功则为 0, 若出错则为 - 1

pid_t    tcgetpgrp(int filedes);
        <sys/types.h>
        <unistd.h>
        返回: 若成功则为前台进程组的进程组 ID 若出错则为 - 1

```

```

int      tcsendbreak(int filedes, int duration);
        <termios.h>
        返回：若成功则为 0，若出错则为 - 1

int      tcsetattr(int filedes, int opt, const struct termios *termpr);
        <termios.h>
        opt: TCSANOW, TCSADRAIN, TCSAFLUSH
        返回：若成功则为 0，若出错则为 - 1

int      tcsetpgrp(int filedes, pid_t pgrp);
        <sys/types.h>
        <unistd.h>
        返回：若成功则为 0，若出错则为 - 1

char     *tempnam(const char *directory, const char *prefix);
        <stdio.h>
        返回：指向一唯一路径名的指针

time_t   time(time_t *calptr);
        <time.h>
        返回：若成功则为时间值，若出错则为 - 1

clock_t   times(struct tms *buf);
        <sys/times.h>
        返回：若成功则为过去的墙上时钟时间（单位：滴答），若出错则为 - 1

FILE     *tmpfile(void);
        <stdio.h>
        返回：若成功则为文件指针，若出错则为 NULL

char     *tmpnam(char *ptr);
        <stdio.h>
        返回：指向唯一路径名的指针

int      truncate(const char *pathname, off_t length);
        <sys/types.h>
        <unistd.h>
        返回：若成功则为 0，若出错则为 - 1

char     *ttyname(int filedes);
        <unistd.h>
        返回：指向终端路径名的指针，若出错则为 NULL

mode_t   umask(mode_t cmask);
        <sys/types.h>
        <sys/stat.h>
        返回：以前的文件方式创建屏蔽

int      uname(struct utsname *name);
        <sys/utsname.h>
        返回：若成功则为非负值，若出错则为 - 1

int      ungetc(int c, FILE *fp);
        <stdio.h>
        返回：若成功则为 c，若出错则为 EOF

int      unlink(const char *pathname);
        <unistd.h>
        返回：若成功则为 0，若出错则为 - 1

void     unsetenv(const char *name);
        <stdlib.h>

int      utime(const char *pathname, const struct utimbuf *times);
        <sys/types.h>
        <utime.h>
        返回：若成功则为 0，若出错则为 - 1

int      vfprintf(FILE *fp, const char *format, va_list arg);
        <stdarg.h>
        <stdio.h>
        返回：若成功则为输出字符数，若输出错则为负值

```

```

int    vprintf(const char *format, va_list arg);
        <stdarg.h>
        <stdio.h>
        返回：若成功则为输出字符数，若输出错则为负值

int    vsprintf(char *buf, const char *format, va_list arg);
        <stdarg.h>
        <stdio.h>
        返回：存入数组的字符数

pid_t  wait(int *statloc);
        <sys/types.h>
        <sys/wait.h>
        返回：若成功则为进程ID，若出错则为 - 1

pid_t  wait3(int *statloc, int options, struct rusage *rusage);
        <sys/types.h>
        <sys/wait.h>
        <sys/time.h>
        <sys/resource.h>
        options: 0, WNOHANG, WUNTRACED
        返回：若成功则为进程ID，若出错则为 - 1

pid_t  wait4(pid_t pid, int *statloc, int options, struct rusage *rusage);
        <sys/types.h>
        <sys/wait.h>
        <sys/time.h>
        <sys/resource.h>
        options: 0, WNOHANG, WUNTRACED
        返回：若成功则为进程ID，若出错则为 - 1

pid_t  waitpid(pid_t pid, int *statloc, int options);
        <sys/types.h>
        <sys/wait.h>
        options: 0, WNOHANG, WUNTRACED
        返回：若成功则为进程ID，若出错则为 - 1

ssize_t write(int filedes, const void *buff, size_t nbytes);
        <unistd.h>
        返回：若成功则为写的字节数，若出错则为 - 1

ssize_t writew(int filedes, const struct iovec iov[], int iovcnt);
        <sys/types.h>
        <sys/uio.h>
        返回：若成功则为写的字节数，若出错则为 - 1

```

附录B 其他源代码

B.1 头文件

正文中的大多数程序都包含头文件 `ourhdr.h`，这示于程序 B-1 中。其中定义了常数（例如 `MAXLINE`）和我们自编函数的原型。

因为大多数程序包含下列头文件：`<stdio.h>`、`<stdlib.h>`（其中有 `exit` 函数原型），以及 `<unistd.h>`（其中包含所有标准 UNIX 函数的原型），所以 `ourhdr.h` 包含了这些系统头文件，同时还包含了 `<string.h>`。这样就减少了本书正文中所有程序的长度。

程序 B-1 头文件 `ourhdr.h`

```
/* Our own header, to be included *after* all standard system headers */

#ifndef __ourhdr_h
#define __ourhdr_h

#include <sys/types.h> /* required for some of our prototypes */
#include <stdio.h> /* for convenience */
#include <stdlib.h> /* for convenience */
#include <string.h> /* for convenience */
#include <unistd.h> /* for convenience */

#define MAXLINE 4096 /* max line length */

#define FILE_MODE (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
/* default file access permissions for new files */
#define DIR_MODE (FILE_MODE | S_IXUSR | S_IXGRP | S_IXOTH)
/* default permissions for new directories */

typedef void Sigfunc(int); /* for signal handlers */

/* 4.3BSD Reno <signal.h> doesn't define SIG_ERR */
#if defined(SIG_IGN) && !defined(SIG_ERR)
#define SIG_ERR ((Sigfunc *)-1)
#endif

#define min(a,b) ((a) < (b) ? (a) : (b))
#define max(a,b) ((a) > (b) ? (a) : (b))

/* prototypes for our own functions */
char *path_alloc(int *); /* Program 2.2 */
int open_max(void); /* Program 2.3 */
void clr_fl(int, int); /* Program 3.5 */
void set_fl(int, int); /* Program 3.5 */
void pr_exit(int); /* Program 8.3 */
void pr_mask(const char *); /* Program 10.10 */
Sigfunc *signal_intr(int, Sigfunc *); /* Program 10.13 */

int tty_cbreak(int); /* Program 11.10 */
int tty_raw(int); /* Program 11.10 */
int tty_reset(int); /* Program 11.10 */
void tty_atexit(void); /* Program 11.10 */
#ifndef ECHO /* only if <termios.h> has been included */
struct termios *tty_termios(void); /* Program 11.10 */
#endif

```



```

#endif

void    sleep_us(unsigned int);    /* Exercise 12.6 */
ssize_t readn(int, void *, size_t); /* Program 12.13 */
ssize_t writen(int, const void *, size_t); /* Program 12.12 */
int     daemon_init(void);        /* Program 13.1 */

int     s_pipe(int *);            /* Programs 15.2 and 15.3 */
int     recv_fd(int, ssize_t (*func)(int, const void *, size_t));
                                         /* Programs 15.6 and 15.8 */
int     send_fd(int, int);        /* Programs 15.5 and 15.7 */
int     send_err(int, int, const char *); /* Program 15.4 */
int     serv_listen(const char *); /* Programs 15.19 and 15.22 */
int     serv_accept(int, uid_t *); /* Programs 15.20 and 15.24 */
int     cli_conn(const char *);   /* Programs 15.21 and 15.23 */
int     buf_args(char *, int (*func)(int, char **));
                                         /* Program 15.17 */

int     pty_open(char *);         /* Programs 19.1 and 19.2 */
int     ptys_open(int, char *);   /* Programs 19.1 and 19.2 */
#ifdef TIOCGWINSZ
pid_t   pty_fork(int *, char *, const struct termios *,
                const struct winsize *); /* Program 19.3 */
#endif

int     lock_reg(int, int, int, off_t, int, off_t);
                                         /* Program 12.2 */
#define read_lock(fd, offset, whence, len) \
        lock_reg(fd, F_SETLK, F_RDLCK, offset, whence, len)
#define readw_lock(fd, offset, whence, len) \
        lock_reg(fd, F_SETLKW, F_RDLCK, offset, whence, len)
#define write_lock(fd, offset, whence, len) \
        lock_reg(fd, F_SETLK, F_WRLCK, offset, whence, len)
#define writew_lock(fd, offset, whence, len) \
        lock_reg(fd, F_SETLKW, F_WRLCK, offset, whence, len)
#define un_lock(fd, offset, whence, len) \
        lock_reg(fd, F_SETLK, F_UNLCK, offset, whence, len)

pid_t   lock_test(int, int, off_t, int, off_t);
                                         /* Program 12.3 */

#define is_readlock(fd, offset, whence, len) \
        lock_test(fd, F_RDLCK, offset, whence, len)
#define is_writelock(fd, offset, whence, len) \
        lock_test(fd, F_WRLCK, offset, whence, len)

void    err_dump(const char *, ...);    /* Appendix B */
void    err_msg(const char *, ...);
void    err_quit(const char *, ...);
void    err_ret(const char *, ...);
void    err_sys(const char *, ...);

void    log_msg(const char *, ...);    /* Appendix B */
void    log_open(const char *, int, int);
void    log_quit(const char *, ...);
void    log_ret(const char *, ...);
void    log_sys(const char *, ...);

void    TELL_WAIT(void);              /* parent/child from Section 8.8 */
void    TELL_PARENT(pid_t);
void    TELL_CHILD(pid_t);
void    WAIT_PARENT(void);
void    WAIT_CHILD(void);

#endif /* __ourhdr_h */

```

程序中先包括一般系统头文件,然后再包括 ourhdr.h,这样就能解决某些系统之间的差别(例如 4.3BSD Reno中没有定义 SIG_ERR),并且也可定义一些我们的函数原型,而这些仅当包括一般系统头文件之后才是需要的。当在原型中引用未定义的结构时,某些 ANSI C 编译程序会认为不正常。

B.2 标准出错处理例程

我们提供了两个出错处理例程,它们可用于本书中大多数实例以处理各种出错情况。一个例程以 err_ 开头,并向标准出错文件输出一条出错消息。另一个例程以 log_ 开头,用于精灵进程(见第13章),它们多半没有控制终端。

提供了这些出错处理函数后,只要在程序中写一行代码就可以进行出错处理,例如:

```
if (出错条件)
    err_dump(带任意参数的printf格式);
```

这样也就不再需要使用下列代码:

```
if (出错条件) {
    char buff[200];
    sprintf(buff,带任意参数的printf格式);
    perror(buff);
    abort();
}
```

我们的出错处理函数使用了 ANSI C 的变长参数表功能。其详细说明见 Kernighan 和 Ritchie [1998] 的 7.3 节。应当注意的是这一 ANSI C 功能与早期系统(例如 SVR3 和 4.3BSD)提供的 varargs 功能不同。宏的名字相同,但更改了某些宏的参数。

表B-1列出了各个出错处理函数之间的区别。

表B-1 标准出错处理函数

函数	strerror(errno)?	终止?
Err_ret	是	return;
Err_sys	是	exit(1);
Err_dump	是	abort();
Err_msg	否	return;
Err_quit	否	exit(1);
Log_ret	是	return;
Log_sys	是	exit(2);
Log_msg	否	return;
Log_quit	否	exit(2);

程序B-2包括了输出至标准出错文件的各个出错处理函数。

程序B-2 输出至标准出错文件的出错处理函数

```
#include <errno.h> /* for definition of errno */
#include <stdarg.h> /* ANSI C header file */
#include "ourhdr.h"

static void err_doit(int, const char *, va_list);
```

```
char    *pname = NULL;      /* caller can set this from argv[0] */

/* Nonfatal error related to a system call.
 * Print a message and return. */

void
err_ret(const char *fmt, ...)
{
    va_list    ap;
    va_start(ap, fmt);
    err_doit(1, fmt, ap);
    va_end(ap);
    return;
}

/* Fatal error related to a system call.
 * Print a message and terminate. */

void
err_sys(const char *fmt, ...)
{
    va_list    ap;
    va_start(ap, fmt);
    err_doit(1, fmt, ap);
    va_end(ap);
    exit(1);
}

/* Fatal error related to a system call.
 * Print a message, dump core, and terminate. */

void
err_dump(const char *fmt, ...)
{
    va_list    ap;

    va_start(ap, fmt);
    err_doit(1, fmt, ap);
    va_end(ap);
    abort();          /* dump core and terminate */
    exit(1);         /* shouldn't get here */
}

/* Nonfatal error unrelated to a system call.
 * Print a message and return. */

void
err_msg(const char *fmt, ...)
{
    va_list    ap;

    va_start(ap, fmt);
    err_doit(0, fmt, ap);
    va_end(ap);
    return;
}

/* Fatal error unrelated to a system call.
 * Print a message and terminate. */

void
err_quit(const char *fmt, ...)
{
    va_list    ap;
```

```

    va_start(ap, fmt);
    err_doit(0, fmt, ap);
    va_end(ap);
    exit(1);
}

/* Print a message and return to caller.
 * Caller specifies "errnoflag". */

static void
err_doit(int errnoflag, const char *fmt, va_list ap)
{
    int    errno_save;
    char   buf[MAXLINE];

    errno_save = errno;    /* value caller might want printed */
    vsprintf(buf, fmt, ap);
    if (errnoflag)
        sprintf(buf+strlen(buf), ": %s", strerror(errno_save));
    strcat(buf, "\n");
    fflush(stdout);        /* in case stdout and stderr are the same */
    fputs(buf, stderr);
    fflush(NULL);        /* flushes all stdio output streams */
    return;
}

```

程序B-3包括了各log_XXX出错处理函数。若进程不以精灵进程方式进行，那么调用者应当定义变量debug，并将其设置为非0值。在这种情况下，出错消息被送至标准出错文件。若debug标志为0，则使用syslog设施（见13.4.2节）。

程序B-3 用于精灵进程的处理函数

```

/* Error routines for programs that can run as a daemon. */

#include <errno.h>    /* for definition of errno */
#include <stdarg.h>   /* ANSI C header file */
#include <syslog.h>
#include "ourhdr.h"

static void log_doit(int, int, const char *, va_list ap);

extern int  debug;    /* caller must define and set this:
                       nonzero if interactive, zero if daemon */

/* Initialize syslog(), if running as daemon. */

void
log_open(const char *ident, int option, int facility)
{
    if (debug == 0)
        openlog(ident, option, facility);
}

/* Nonfatal error related to a system call.
 * Print a message with the system's errno value and return. */

void
log_ret(const char *fmt, ...)
{
    va_list    ap;

    va_start(ap, fmt);
    log_doit(1, LOG_ERR, fmt, ap);
    va_end(ap);
}

```

```
    return;
}

/* Fatal error related to a system call.
 * Print a message and terminate. */

void
log_sys(const char *fmt, ...)
{
    va_list    ap;

    va_start(ap, fmt);
    log_doit(1, LOG_ERR, fmt, ap);
    va_end(ap);
    exit(2);
}

/* Nonfatal error unrelated to a system call.
 * Print a message and return. */

void
log_msg(const char *fmt, ...)
{
    va_list    ap;

    va_start(ap, fmt);
    log_doit(0, LOG_ERR, fmt, ap);
    va_end(ap);
    return;
}

/* Fatal error unrelated to a system call.
 * Print a message and terminate. */

void
log_quit(const char *fmt, ...)
{
    va_list    ap;

    va_start(ap, fmt);
    log_doit(0, LOG_ERR, fmt, ap);
    va_end(ap);
    exit(2);
}

/* Print a message and return to caller.
 * Caller specifies "errnoflag" and "priority". */

static void
log_doit(int errnoflag, int priority, const char *fmt, va_list ap)
{
    int        errno_save;
    char       buf[MAXLINE];

    errno_save = errno; /* value caller might want printed */
    vsprintf(buf, fmt, ap);
    if (errnoflag)
        sprintf(buf+strlen(buf), ": %s", strerror(errno_save));
    strcat(buf, "\n");
    if (debug) {
        fflush(stdout);
        fputs(buf, stderr);
        fflush(stderr);
    } else
        syslog(priority, buf);
    return;
}
```

附录C 习题答案

第1章

1.1 利用ls(1)命令中的下面两个选择项：-i——显示文件或目录的i节点数目（关于i节点在4.14节中会详细讨论）；-d——如果参数是一目录，只列出其名字，而不是目录中的所有文件。

执行命令的结果为：

```
$ ls -ldi /etc/. /etc/.. -i要求打印i节点的数量
3077 drwxr-sr-x  7 bin  2048 Aug 5 20:12 /etc/./
 2 drwxr-xr-x 13 root  512 Aug 5 20:11 /etc/./
$ls -ldi ./ ../          .和..的i节点数均为2
 2 drwxr-xr-x 13 root  512 Aug 5 20:11 ./
 2 drwxr-xr-x 13 root  512 Aug 5 20:11 ../
```

1.2 UNIX是多任务系统，所以，在程序1-4运行的同时其他两个进程也在运行。

1.3 假如perror的ptr参数是一个指针，则perror就可以改变ptr所指串的内容。所以利用限定词const使得perror不能修改ptr所指的串。而strerror的参数是错误号，由于其是整数类型并且C传递的是参数值，因此strerror不能修改参数的值，也就没有必要使用const属性。（如果C中函数参数的处理不是很清楚，可参见Kernighan和Ritchie〔1998〕5.2节。）

1.4 调用fflush，fprintf和vprintf函数可修改errno的值。如果它的值变了但没有保存，则最终显示的错误信息是不正确的。

在过去开发的许多程序中，都可以发现不保存errno的情况，典型的错误信息是“Not a typewriter（打字机不存在）”。5.4节中标准I/O库根据标准I/O流是否指向终端设备而改变流的缓存器。istty（见11.9节）通常用来判断流是否指向终端设备，如果流不指向终端设备，errno可能置为ENOTTY，从而引起该错误。程序C-1显示了这一特性。

程序C-1 errno和printf的交互作用

```
#include <stdio.h>

/*
 * The following prints errno=25 (ENOTTY) under 4.3BSD and SVR2,
 * when stdout is redirected to a file.
 * Under SVR4 and 4.3+BSD it works OK.
 */

int
main()
{
    int fd;
    extern int errno;

    if ( (fd = open("/no/such/file", 0)) < 0) {
        printf("open error: ");
        printf("errno = %d\n", errno);
    }
    exit(0);
}
```

执行上面的程序，结果为：

```
$ grep BSD /etc/motd
4.3 BSD UNIX #29: Thu Mar 29 11:14:13 MST 1990
$ a.out
open error: error = 2      工作正常，stdout是一个终端
$ a.out > temp.foo
$ cat temp.foo
open error: error = 25    错误
```

1.5 2038年。

1.6 大约248天。

第2章

2.1 下面是4.3+BSD中使用的技术。在<machine/ansi.h>中，用大写字母定义可在多个头文件中出现的基本数据类型。例如：

```
#ifndef _ANSI_H_
#define _ANSI_H_

#define _CLOCK_T_ unsigned long
#define _SIZE_T_ unsigned int

...

#endif /* _ANSI_H_ */
```

以下面的顺序可以在这6个头文件中分别定义size_t。

```
#ifdef _SIZE_T_
typedef _SIZE_T_ size_t;
#undef _SIZE_T_
#endif
```

这样，实际上只执行一次typedef。

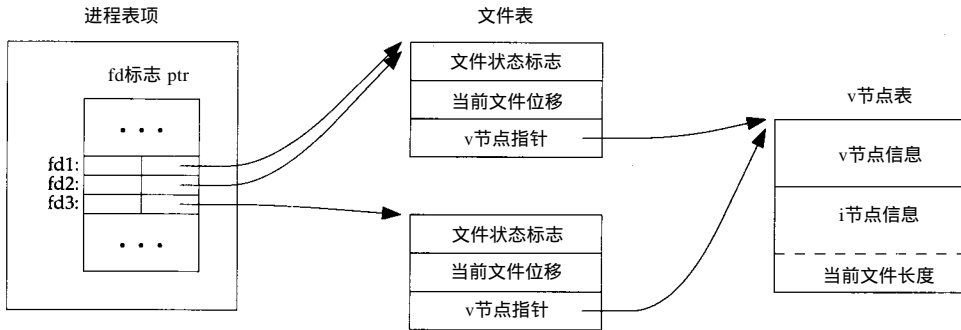
第3章

3.1 所有的磁盘I/O都要经过内核的块缓存器，唯一例外的是对原始磁盘设备的I/O，但是我们不考虑这种情况（Bach〔1986〕的第3章讲述了这种缓存器的操作）。既然read或write的数据都要被内核缓存，那么术语“无缓存装置的I/O”指的是在用户的进程中对这两个函数不会自动缓存，每次read或write就要进行一次系统调用。

3.3 每次调用open函数就分配一个文件表项，如果两次打开的是相同的文件，则两个文件表项指向相同的v节点。调用dup引用已存在的文件表项（此处指fd1的文件表项），见图C-1。当F_SETFD作用于fd1时，只影响fd1的文件描述符标志；F_SETFL作用于fd1时，则影响fd1及fd2的文件描述符标志。

3.4 如果fd是1，执行dup2(fd, 1)后返回1，但是没有关闭描述符1（见3.12节）。调用3次dup2后，3个描述符指向相同的文件表项，所以不需要关闭描述符。

如果fd是3，调用3次dup2后，有4个描述符指向相同的文件表项，所以需要关闭描述符3。



图C-1 open和dup的结果

3.5 shell从左到右处理命令行,所以

```
a.out > outfile 2>&l
```

首先设置标准输出到outfile,然后执行dups将标准输出复制到描述符2(标准错误)上,其结果是将标准输出和标准错误设置为相同的文件,即描述符1和2指向相同的文件表项。而对于命令行

```
a.out 2 >&l >outfile
```

由于首先执行dups,所以描述符2成为终端(假设命令是交互执行的),标准输出重定向到outfile。结果是描述符1指向outfile的文件表项,描述符2指向终端的文件表项。

3.6 这种情况之下,仍然可以用lseek和read函数读文件中任意一处的内容。但是write函数在写数据之前会自动将文件位移量设置为文件尾,所以写文件时只能从文件尾开始,不能在任一位置。

第4章

4.1 stat函数总是顺一个符号连接向前,所以修改后的程序不会显示文件类型是“符号连接”。例如:/bin是/usr/bin的一个符号连接,但是stat函数的结果只显示/bin是一个目录,而不说明它是一个符号连接。若一个符号连接指向一不存在的文件,则stat出错返回。

4.2 将下面的几行语句加入<ourhdr.h>

```
#if defined (S_IFLNK) && !defined(S_ISLNK )
#define S_ISLNK(mode) (((mode) & S_IFMT) == S_IFLNK)
#endif
```

这是一个我们编写的头文件如何屏蔽某些系统差别的实例。

4.3 关闭了该文件的所有存取许可权。

```
$ umask 777
$ data > temp.foo
$ ls -l temp.foo
----- l stevens 29 Jan 14 06:39 temp.foo
```

4.4 下面的命令表示关闭用户读许可权的情况。

```
$ data > foo
$ chmod u-r foo 关闭用户读许可权
$ ls -l foo 验证文件的许可权
--w-rw-r-- l stevens 29 Jul 31 09:00 foo
$ cat foo 读文件
```



```
cat: foo: Permission denied
```

4.5 如果用open或creat创建已经存在的文件，则该文件的存取许可权不变。程序 4-3可以验证这点。

```
$ rm foo bar          删除文件
$ data > foo          创建文件
$ data > bar
$ chmod a-r foo bar   关闭所有的读许可权
$ ls -l foo bar       验证其许可权
--w--w---- 1 stevens  29 Jul 31 10:47 bar
--w--w---- 1 stevens  29 Jul 31 10:47 foo
$ a.out               运行程序4-3
$ ls -l foo bar       检查文件的许可权和大小
--w--w---- lstevens  0 Jul 31 10:47 bar
--w--w---- lstevens  0 Jul 31 10:47 foo
```

可以看出存取许可权没有改变，但是文件长度缩短了。

4.6 目录的长度从来不会是0，因为它总是包含 .和 ..两项。符号连接的长度指其路径名包含的字符数，由于路径名中至少有一个字符，所以长度也不为0。

4.8 当创建新的core文件时，内核对其存取许可权有一个默认设置，在本例中是 rw-r--r--。这一默认值可能会可能不会被 umask 的值修改。shell对创建的重定向的新文件也有一个默认的访问许可权，本例中为rw-rw-rw-。这个值总是被当前的umask修改，在本例中umask为02。

4.9 不能使用du的原因是它需要文件名，如：

```
du tempfile
```

或目录名，如：

```
du .
```

只有当unlink函数返回时才释放tempfile的目录项，du命令没有计算仍然被tempfile占用的空间。本例中只能使用df命令察看文件系统中实际可用的自由空间。

4.10 如果被删除的链接不是该文件的最后一个链接，则该文件不会删除。此时，文件的状态改变时间被更新。如果是最后一个链接被删除，则该文件将被物理删除。这时再去更新文件的状态改变时间就没有意义，因为包含文件所有信息的i节点将会随着文件的删除而被释放。

4.11 用opendir打开一个目录后，循环调用函数dopath。假设opendir使用一个文件描述符，并且只有处理完目录后调用closedir才释放描述符，这就意味着每次打开目录就要降一级使用另外一个描述符。所以系统可打开的描述符数就限制了文件系统中树的深度。SVR4中的ftw允许调用者指定使用的描述符数，这隐含着该实现可以关闭描述符并且重用它们。

4.13 chroot函数用于辅助因特网文件传输程序（FTP）中的安全性。系统中没有帐号的用户（也称为匿名FTP）放在一个单独的目录下，利用chroot将此目录当作新的根目录就可以阻止用户访问此目录以外的文件。

chroot也用于在另一台机器上构造一文件系统层次结构的一个副本，然后修改此副本，但不更改原来的文件系统。这可用于测试新软件包的安装。chroot只能由超级用户使用，一旦更改了一个进程的root，该进程及其后代进程就再也不能恢复至原先的root。

4.14 首先调用stat函数取得文件的三个时间值，然后调用utime设置期望的值。我们不希望在调用utime时改变的值就是stat中相应的值。

4.15 finger(1)对邮箱调用stat函数，最近一次的修改时间是上一次接收邮件的时间，最近存取时间是上一次读邮件的时间。

4.16 对cpio来说，既可以改变文件的访问时间（`st_atime`）和修改时间（`st_mtime`），也可以都不改变。cpio的-a选项可以在读文件后重新设置文件的存取时间，改变文件的存取时间。另一方面，-m将文件的修改时间和存取时间保存为归档时的值。

对tar来说，在抽取文件时，其默认方式是复原归档时的修改时间，但是-m选择项则将修改时间设置为抽取文件时的时间。无论tar在何种情况，文件的存取时间均是抽取文件时的时间。

由于不能修改状态改变时间（`utime`也只能改变访问时间和修改时间），所以没有将其保存在文档上。

4.17 `read`改变了文件存取时间，为了消除这一影响，有些版本的 `file(1)`调用`utime`恢复文件的存取时间，但是这样做会修改文件的状态改变时间。

4.18 内核对目录的深度没有内在的限制，但是如果路径名的长度超出了 `PATH_MAX`，则有许多命令会失败。程序 C-2创建了一个深度为100的目录树，每一级目录名有45个字符。利用`getcwd`可以得到第100级目录的绝对路径名（需要多次调用`realloc`申请一个足够大的缓存）。

程序C-2 创建深目录树

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

#define DEPTH 100 /* directory depth */
#define MYHOME "/home/stevens"
#define NAME "alonglonglonglonglonglonglonglongname"

int
main(void)
{
    int i, size;
    char *path;

    if (chdir(MYHOME) < 0)
        err_sys("chdir error");

    for (i = 0; i < DEPTH; i++) {
        if (mkdir(NAME, DIR_MODE) < 0)
            err_sys("mkdir failed, i = %d", i);
        if (chdir(NAME) < 0)
            err_sys("chdir failed, i = %d", i);
    }
    if (creat("afile", FILE_MODE) < 0)
        err_sys("creat error");

    /*
     * The deep directory is created, with a file at the leaf.
     * Now let's try and obtain its pathname.
     */

    path = path_alloc(&size);
    for ( ; ; ) {
        if (getcwd(path, size) != NULL)
            break;
        else {
            err_ret("getcwd failed, size = %d", size);
            size += 100;
            if ( (path = realloc(path, size)) == NULL)
```

```

        err_sys("realloc error");
    }
}
printf("length = %d\n%s\n", strlen(path), path);

exit(0);
}

```

运行后得到：

```

$ a.out
getcwd failed, size = 1025: Result too large
getcwd failed, size = 1125: Result too large
...
33行
getcwd failed, size = 4525: Result too large
length = 4613

```

显示4613字节的路径名

但是由于文件名太长了，不能用tar或cpio对该目录建立档案文件，而且也不能用rm -r命令删除该目录。（怎样才能删除该目录树？）

4.19 /dev目录关闭了一般用户的写许可权，所以用户不能删除目录中的文件，即unlink失败。

第5章

5.2 fgets函数读入数据，直到行结束或缓存满（当然会留出一个字节存放终止字符）。同样，fputs只负责将缓存的内容输出，而并不考虑缓存中是否包含换行符。所以，如果将MAXLINE设得很小，这两个函数仍然会正常工作，只不过被执行的次数要比MAXLINE值较大的时候多。

如果这些函数删除或添加换行符（如gets和puts），则必需保证缓存足够大。

5.3 当printf没有输出任何字符时，如：printf("")，则返回0。

5.4 这是一个比较常见的错误。getc以及getchar的返回值是整型，而不是字符型。由于EOF经常定义为-1，那么如果系统使用的是有符号的字符类型，程序还可以正常工作。但如果使用的是无符号字符类型，那么返回的EOF被保存到字符c后将不再是-1，所以，程序会进入死循环。

5.5 5个字符长的前缀、4个字符长的进程内唯一标识再加5个字符长的系统内唯一标识（进程ID）刚好组成14位的UNIX传统文件长度限制。

5.6 使用方法为：先调用fflush后调用fsync，fsync所使用的参数由fileno函数获得。如果不调用fflush，所有的数据仍然在内存缓存中，此时调用fsync将没有任何效果。

5.7 当程序交互运行时，标准输入和输出设备均为行缓存方式。每次调用fgets时标准输出设备将自动刷清。

第6章

6.1 在SVR4系统中，用户手册中讲述了存取阴影口令文件的函数。我们不能使用6.2节所述函数返回的pw_passwd变量来比较加密口令。正确的方法是使用阴影口令文件中对应用户的加密口令来进行比较。

在4.3+BSD系统中，口令文件的阴影是自动建立的。仅当调用者的用户ID为0时，getpwnam或getpwuid函数返回的passwd结构中的pw_passwd字段才包含有加密口令。

6.2 在SVR4系统中，程序C-3将输出加密口令。当然，除非有超级用户许可权，否则调用getspnam将返回EACCES错误。

程序C-3 在SVR4系统中输出加密口令

```
#include <sys/types.h>
#include <shadow.h>
#include "ourhdr.h"

int
main(void) /* SVR4 version */
{
    struct spwd *ptr;

    if ( (ptr = getspnam("stevens")) == NULL)
        err_sys("getspnam error");

    printf("sp_pwdp = %s\n",
           ptr->sp_pwdp == NULL || ptr->sp_pwdp[0] == 0 ?
           "(null)" : ptr->sp_pwdp);

    exit(0);
}
```

在4.3+BSD系统中，具有超级用户许可权时，程序C-4将输出加密口令。否则pw_passwd的返回值为星号(*)。

程序C-4 在4.3+ BSD系统中输出加密口令

```
#include <sys/types.h>
#include <pwd.h>
#include "ourhdr.h"

int
main(void) /* 4.3+BSD version */
{
    struct passwd *ptr;

    if ( (ptr = getpwnam("stevens")) == NULL)
        err_sys("getpwnam error");

    printf("pw_passwd = %s\n",
           ptr->pw_passwd == NULL || ptr->pw_passwd[0] == 0 ?
           "(null)" : ptr->pw_passwd);

    exit(0);
}
```

6.4 程序C-5以date格式输出日期。

程序C-5 以date(1)的格式输出日期和时间

```
#include <time.h>
#include "ourhdr.h"

int
main(void)
{
    time_t    caltime;
    struct tm *tm;
    char      line[MAXLINE];

    if ( (caltime = time(NULL)) == -1)
        err_sys("time error");
    if ( (tm = localtime(&caltime)) == NULL)
        err_sys("localtime error");
```

```

if (strftime(line, MAXLINE, "%a %b %d %X %Z %Y\n", tm) == 0)
    err_sys("strftime error");
fputs(line, stdout);

exit(0);
}

```

程序C-5的运行结果如下：

```

$ echo $TZ                                作者使用的默认值
MST7
$ a.out
Wed Jan 15 06:48:57 MST 1992
$ TZ=EST5EDT a.out                        美国东海岸
Wed Jan 15 08:49:06 EST 1992
$ TZ=JST-9 a.out                           日本
Wed Jan 15 22:49:12 JST 1992

```

第7章

7.1 原因在于printf的返回值（输出的字符数）变成了main函数的返回码。当然，并不是所有的系统都会出现该情况。

7.2 当程序处于交互运行方式时，标准输出设备通常处于行缓存方式，所以当输出换行符时，上次的结果才被真正输出。如果标准输出设备被定向到一个文件而处于完全缓存方式，则当标准I/O清理操作执行时，结果才真正被输出。

7.3 由于argc和argv不像environ一样保存在全局变量中，所以在大多数UNIX系统中没有其他办法。

7.4 当C程序复引用一个空指针出错时，执行该程序的进程将终止，于是可以利用这种方法终止进程。

7.5 定义如下：

```

typedef void Exitfunc(void) ;
int atexit(Exitfunc*);

```

7.6 calloc将分配的内存空间初始化为0。但是ANSI C并不保证0值与浮点0或空指针的值相同。

7.7 只有通过exec函数执行一个程序时，才会分配堆和堆栈。

7.8 可执行文件包含了用于调试core文件的符号表信息，用strip(1)可以删除这些信息，对两个a.out文件执行这条命令，它们的大小减为98 304和16 384。

7.9 没有使用共享库时，可执行文件的大部分都被标准I/O库所占用。

7.10 这段代码不正确。因为在if语句中定义了自动变量val，所以当if中的复合语句结束时，该变量就不存在了，但是在if语句之外又用指针引用已经不存在的自动变量val。

第8章

8.1 用下面几行代替程序8-2中调用printf的语句：

```

i = printf("pid = %d, glob = %d, var = %d\n",
          getpid( ), glob, var);
sprintf (buf, "%d\n", i);

```

```
write (STDOUT_FILENO, buf, strlen(buf));
```

注意要定义变量*i*和*buf*。

这里假设子进程调用 `exit` 时只关闭标准 I/O 流，并不关闭与标准输出相关的文件描述符 `STDOUT_FILENO`。有些版本的标准 I/O 库会关闭与标准输出相关的文件描述符从而引起写失败，这种情况就调用 `dup` 将标准输出复制到另一个描述符，`write` 则使用新复制的文件描述符。

8.2 可以通过程序 C-6 来说明这个问题。

程序 C-6 错误使用 `vfork` 的例子

```
#include <sys/types.h>
#include "ourhdr.h"

static void f1(void), f2(void);

int
main(void)
{
    f1();
    f2();
    _exit(0);
}

static void
f1(void)
{
    pid_t pid;

    if ( (pid = vfork()) < 0)
        err_sys("vfork error");
    /* child and parent both return */
}

static void
f2(void)
{
    char buf[1000]; /* automatic variables */
    int i;

    for (i = 0; i < sizeof(buf); i++)
        buf[i] = 0;
}

```

当函数 `f1` 调用 `vfork` 时，父进程的堆栈指针就指向 `f1` 的栈帧，见图 C-2。`vfork` 使得子进程先执行然后从 `f1` 返回，接着子进程调用 `f2` 并且覆盖了 `f1` 的堆栈区间，在 `f2` 中子进程将自动变量 `buf` 的值置为 0，即将堆栈中的 1000 个字节的值都置为 0。从 `f2` 返回后父进程继续执行调用 `_exit`，这时堆栈中 `main` 以下的内容已经被 `f2` 修改了，但是父进程仍然以为调用了 `vfork` 后从 `f1` 返回。返回信息虽然保存在堆栈中，但是可能已经被子进程修改了。对这个例子，父进程继续执行的结果要依赖于实际的 UNIX 系统。（如：返回信息保存在堆栈的具体位置，修改动态变量时覆盖了哪些信息等等。）通常的结果是一个 core 文件。

8.3 在程序 8-7 中我们先让父进程输出，但是当父进程输出完毕子进程要输出时，要让父进程终止。是父进程先终止还是子进程先执行输出要依赖

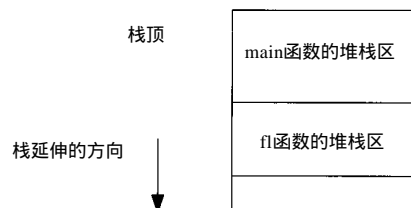


图 C-2 调用 `vfork` 时的栈帧

于内核对两个进程的调度。shell在父进程终止后会开始执行其他程序，这样也许仍会影响子进程。要避免这种情况就是在子进程完成输出后才终止父进程。用下面的语句替换程序中 fork 后面的代码。由于只有终止父进程才能开始下一个程序，所以不会出现上面的情况。

```
else if (pid == 0) {
    WAIT_PARENT();           /* parent goes first */
    charatime("output from child\n");
    TELL_PARENT(getppid()); /* tell parent we're done */
} else {
    charatime("output from parent\n");
    TELL_CHILD(pid);        /* tell child we're done */
    WAIT_CHILD();          /* wait for child to finish */
}
```

8.4 对argv[2]打印的是相同的值 (/home/stevens/bin/testinterp)。原因是execlp在结束时调用了execve，并且与直接调用execl的路径名相同。

8.5 不提供返回保存的设置-用户-ID的函数，我们必须在进程开始时保存有效的用户ID。

8.6 程序C-7创建了一个僵死进程。

程序C-7 创建一个僵死进程并用ps查看其状态

```
#include "ourhdr.h"

int
main(void)
{
    pid_t pid;

    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0) /* child */
        exit(0);

    /* parent */
    sleep(4);

    system("ps");

    exit(0);
}
```

执行程序结果如下 (ps (1) 用Z表示僵死进程)：

```
$ a.out
PID TT STTIME COMMAND
5940 p3 S 0:00 a.out
5941 p3 Z 0:00 <defunct> 僵死进程
5942 p3 S 0:00 sh -c ps
5943 p3 R 0:00 ps
```

第9章

9.1 因为init是login shell的父进程，当登录shell终止时它收到SIGCHLD信号量，所以init进程知道什么时候终端用户注销。

网络登录没有包含init，相应的注销项是由一个处理登录并监测注销的进程写的（本例中为telnetd）。

第10章

10.1 当程序第一次接收到发送给它的信号量时就终止了。因为一捕捉到信号量 `pause` 函数就返回。

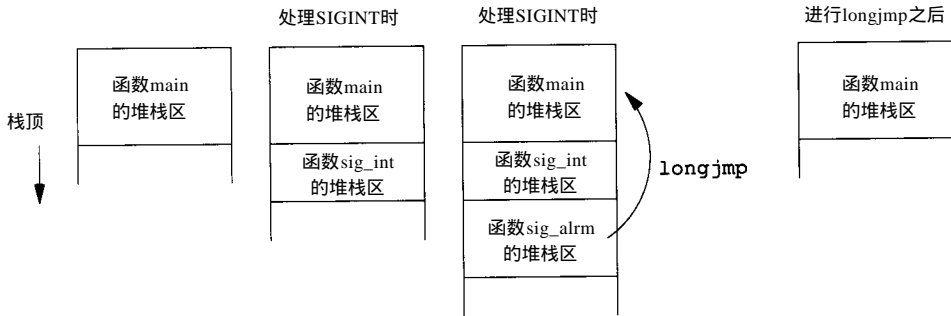
10.2 程序C-8实现了 `raise` 函数。

程序C-8 `raise` 函数的实现

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

int
raise(int signo)
{
    return( kill(getpid(), signo) );
}
```

10.3 见图C-3。



图C-3 `longjmp` 前后的堆栈状态

从 `sig_alarm` 通过 `longjmp` 返回 `main`，有效地避免了继续执行 `sig_int`。

10.4 如果进程在调用 `alarm` 和 `setjmp` 之间被内核阻塞了，`alarm` 时间走完之后就调用信号量处理程序，然后调用 `longjmp`。但是由于没有调用 `setjmp`，所以没有设置 `env_alarm` 缓存区。如果 `longjmp` 的跳转缓存没有被 `setjmp` 初始化，则说明没有定义 `longjmp` 的操作。

10.5 参见 Don Libes 的 “Implementing Software Timers” (*C Users Journal*, Vol. 8, no. 11, Nov. 1990) 中的例子。

10.7 如果仅仅调用 `_exit`，则进程终止状态就不能表示该进程是由于 `SIGABRT` 信号量而终止的。

10.8 如果信号量是由其他用户的进程发出的，进程必须设置用户的 ID 为根或者是接收进程的所有者，否则 `kill` 不能执行。所以实际的用户 ID 为信号量的接收者提供了更多的信息。

10.10 对于本书中所用的系统，大约每 60~90 分钟增加一秒，这个误差是因为每次调用 `sleep` 都要调度一次将来的时间事件，但是由于 CPU 调度，有时我们并没有在事件发生时被叫醒。另外一个原因是开始运行进程和调用 `sleep` 都需要一定的时间。

BSD 中的 `cron` 每分钟都要取当前时间，它首先设置一个休眠周期，然后在下一分钟开始时唤醒。大多数调用是 `sleep(60)`，偶尔有一个 `sleep(59)` 用于在下一分钟同步。但是若在进程中花费了许多时间执行命令或者系统的负载重调度慢，这时休眠值可能远小于 60。

10.11 在SVR4中，从来没有调用过 SIGXFSZ的信号量处理程序，一旦文件的大小达到1024时，write就返回24。

在4.3+BSD中，文件大小达到1500字节时调用该信号处理程序，write返回-1并且errno设置为EFBIG。

SunOS4.1.2的情况与SVR4类似，但是调用了该信号量处理程序。

系统V在文件大小达到软资源限制时无错返回一个较小的数，而BSD判断文件超出限制时错误返回，没有写任何数据。

10.12 结果依赖于标准I/O库的实现——fwrite如何处理一个被中断的写。

第11章

11.1 注意由于终端是非正规模式，所以要用换行符而不是回车终止reset命令。

11.2 它为128个字符建了一张表，根据用户的要求设置奇偶校验位。然后使用8位I/O处理奇偶位的产生。

11.3 在SVR4中运行stty -a，并且将标准输入重定向到运行vi的终端，结果显示vi设置MIN为1，TIME为1。reads等待至少敲入一个字符，但是该字符输入后，只对后继的字符等待十分之一秒即返回。

11.4 在SVR4中使用扩展的通用终端接口。参见AT&T〔1991〕手册中的termiox(7)。在4.3+BSD中使用c_cflag字段的CCTS_OFLOW和CRTS_IFLOW标志，参见表11-1。

第12章

12.1 程序运行正常，不会发生ENOLCK的错误。第一次循环调用writew_lock、write和un_lock。调用un_lock后只保留了第一个字节的锁，第二次循环时，调用writew_lock使得新锁与第一个字节的锁合并，图C-4是第二次循环的结果。

每循环一次就扩展一个字节的锁，内核将这些锁合并后就只保持了一个锁，因此符合锁结构的定义。

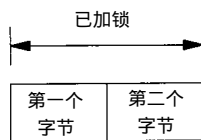
12.2 在SVR4和4.3+BSD中，fd_set是只包含一个成员的结构，该成员为一个长整型数组。数组中每一位对应于一个描述符。四个FD_宏通过开关或测试指定的位来操纵这个数组。将之定义为一个包含数组的结构而不仅仅是一个数组的原因是：通过C语言的赋值语句，可以使fd_set类型变量相互赋值。

12.3 在SVR4和4.3+BSD中允许用户在头文件<sys/types.h>前定义常数FD_SETSIZE。例如下面的代码可以使fd_set数据类型包含2048个描述符。

```
#define FD_SETSIZE 2048
#include <sys/types.h>
```

12.4 下面列出了功能类似的函数。

FD_ZERO	sigemptyset
FD_SET	sigaddset
FD_CLR	sigdelset
FD_ISSET	sigismember



图C-4 第二次循环后锁的状态

没有与sigfillset对应的FD_xxx函数。对信号量来说，指向信号量集合的指针是第一个参数，信号量数是第二个参数；对于描述符来说，描述符数是第一个参数，指向描述符集合的指针是第二个参数。

12.5 最多五种信息：数据，数据长度，控制信息，控制信息的长度和标志。

12.6 利用select实现的程序见C-9，利用poll实现的程序见C-10。

程序C-9 用select实现sleep_us函数

```
#include <sys/types.h>
#include <sys/time.h>
#include <stddef.h>
#include "ourhdr.h"

void
sleep_us(unsigned int nusecs)
{
    struct timeval tval;

    tval.tv_sec = nusecs / 1000000;
    tval.tv_usec = nusecs % 1000000;
    select(0, NULL, NULL, NULL, &tval);
}
```

程序C-10 用poll实现sleep_us函数

```
#include <sys/types.h>
#include <poll.h>
#include <stropts.h>
#include "ourhdr.h"

void
sleep_us(unsigned int nusecs)
{
    struct pollfd dummy;
    int timeout;

    if ( (timeout = nusecs / 1000) <= 0)
        timeout = 1;
    poll(&dummy, 0, timeout);
}
```

BSD中的usleep(3)使用setitimer设置间隔计时器，并且执行8个系统调用。它可以正确地和其他计时器交互作用，而且即使捕捉到信号量也不会被中断。

12.7 不行。我们可以使TELL_WAIT创建一个临时文件，其中一个字节用作为父进程的锁，另一个字节用作为子进程的锁。WAIT_CHILD使得父进程等待子进程的锁，TELL_PARENT使得子进程释放子进程的锁。但是问题在于调用fork后，子进程释放了所有的锁导致子进程不能具有任何它自己的锁而开始执行。

12.8 用select的方法见程序C-11，使用poll的情况类似。

程序C-11 用select计算管道的性能

```
#include <sys/types.h>
#include <sys/time.h>
#include "ourhdr.h"

int
```

```

main(void)
{
    int            i, n, fd[2];
    fd_set        writeset;
    struct timeval tv;

    if (pipe(fd) < 0)
        err_sys("pipe error");
    FD_ZERO(&writeset);

    for (n = 0; ; n++) { /* write 1 byte at a time until pipe is full */
        FD_SET(fd[1], &writeset);
        tv.tv_sec = tv.tv_usec = 0; /* don't wait at all */
        if ( (i = select(fd[1]+1, NULL, &writeset, NULL, &tv)) < 0)
            err_sys("select error");
        else if (i == 0)
            break;
        if (write(fd[1], "a", 1) != 1)
            err_sys("write error");
    }
    printf("pipe capacity = %d\n", n);
    exit(0);
}

```

在SVR4和SunOS 4.1.1中使用select和poll计算出的结果等于表2-6的值。在4.3+BSD中使用select计算的结果为3073。

12.9 在SVR4、4.3+BSD和SunOS 4.1.2中，程序12-14确实修改了输入文件的最近一次存取时间。

第13章

13.1 如果进程调用chroot就不能打开/dev/log，解决的办法是在chroot之前调用选择项为LOG_NDELAY的openlog。这样即使调用了chroot之后，仍然可以打开特定的设备文件（UNIX与数据报套接口）并生成一个有效的描述符。

13.3 程序C-12是一种解决方案。

程序C-12 调用daemon_init获得注册名

```

#include    "ourhdr.h"

int
main(void)
{
    char    *ptr, buff[MAXLINE];

    daemon_init();

    close(0);
    close(1);
    close(2);

    ptr = getlogin();
    sprintf(buff, "login name: %s\n",
            (ptr == NULL) ? "(empty)" : ptr);
    write(3, buff, strlen(buff));
    exit(0);
}

```

结果依赖于不同的系统实现和是否关闭文件描述符1、2和3。关闭描述符影响结果的原因是：当程序开始执行时与控制终端连接，调用daemon_init后关闭3个描述符就意味着getlogin没

有控制终端，所以不能在 utmp 文件中看到登录项。

但是在 4.3+BSD 中，登录名是由进程表维护的，并且可以通过 fork 复制。也就是说除非其父进程没有登录名（如系统自引导时调用 init），否则进程总能获得其登录名。

第14章

14.1 如果写管道端总是不关闭，则读者就决不会看到文件的结束符。页面调度程序就会一直阻塞在读标准输入。

14.2 父进程向管道写完最后一行以后就终止了，然后读者读到管道的结尾时自动关闭管道。但是由于子进程（页面调度程序）要等待输出的页，所以父进程可能比子进程领先一个管道缓存器。如果在一个交互式命令行 shell 上运行，当父进程终止时 shell 会改变终端的模式并提示用户。由于大部分页面调度程序在等待处理下一个页面时将终端设置为非正规模式，所以终止父进程就会影响页面调度程序。

14.3 因为执行了 shell，所以 popen 返回一个文件指针。但是 shell 不能执行不存在的命令，因此在标准错误上显示下面信息后终止，其退出状态为 1，调用 pclose 就将该退出状态返回。

```
sh: a.out: not found
```

14.4 当父进程终止时，用 shell 看它的终止状态。对于 Bourne shell 和 KornShell 所用的命令是

```
echo $?
```

打印的结果是 128 加信号量数。

14.5 首先加入下面的定义，

```
FILE *fpin, *fpout;
```

然后用 fdopen 关联管道描述符和标准 I/O 流，并将流设置为行缓存的，在 while 循环从标准输入读之前作下面的工作。

```
if ( (fpin = fdopen(fd2[0], "r")) == NULL)
    err_sys("fdopen error");
if ( (fpout = fdopen(fd1[1], "w")) == NULL)
    err_sys("fdopen error");
if (setvbuf(fpin, NULL, _IOLBF, 0) < 0)
    err_sys("setvbuf error");
if (setvbuf(fpout, NULL, _IOLBF, 0) < 0)
    err_sys("setvbuf error");
```

while 中的 read 和 write 用下面的语句代替：

```
if (fputs(line, fpout) == EOF)
    err_sys("fputs error to pipe");
if (fgets(line, MAXLINE, fpin) == NULL) {
    err_msg("child closed pipe");
    break;
}
```

14.6 虽然 system 函数调用了 wait，但是终止的第一个子进程是由 popen 产生的，所以它将再次调用 wait 并一直阻塞到 sleep 完成，然后 system 返回。当 pclose 调用 wait 时，由于没有子进程可等待所以返回出错，导致 pclose 也返回出错。

14.7 select 表明描述符是可读的。调用 read 读完所有的数据后返回 0 就表明到达了文件尾端。对于 poll（假设管道是一个流设备）来说，若返回 POLLHUP 也许仍有数据可以读。但是一旦读完了所有的数据 read 就返回 0，即表明到达了文件尾端。poll 读完了所有的数据后并不返回 POLLIN。

对于被读者关闭的指向管道的输出描述符来说，select表明描述符是可写的，调用write时产生SIGPIPE信号量，如果忽略该信号量或从信号量处理程序中返回时write就返回EPIPE错误。而对于poll，如果管道是一个流设备，poll就对该描述符返回POLLHUP。

14.8 子进程向标准出错写的内容同样也在父进程的标准出错中出错。只要在cmdstring中包含重定向2>&1命令，就可以将标准出错发送给父进程。

14.9 popen生成一个子进程，子进程通过exec执行Bourne shell。然后shell再调用fork，最后由shell的子进程执行命令串。当cmdstring终止时shell恰好在等待该事件，然后shell退出，而这一事件又是pclose中waitpid所等待的。

14.10 解决的办法是打开FIFO两次，一次读一次写。我们绝不会使用为写而打开的描述符，但是使该描述符打开就可在客户数从1变为0时，阻止产生文件终止。打开FIFO两次需要注意下列操作方式：第一次以非阻塞、只读方式open，第二次以阻塞、只写方式open。（如果用非阻塞、只写方式打开将返回错误。）然后关闭读描述符的非阻塞属性。参见程序C-13。

程序C-13 以非阻塞方式打开FIFO进行读写操作

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

#define FIFO "temp.fifo"

int
main(void)
{
    int    fdrd, fdwr;

    unlink(FIFO);
    if (mkfifo(FIFO, FILE_MODE) < 0)
        err_sys("mkfifo error");

    if ( (fdrd = open(FIFO, O_RDONLY | O_NONBLOCK)) < 0)
        err_sys("open error for reading");
    if ( (fdwr = open(FIFO, O_WRONLY)) < 0)
        err_sys("open error for writing");

    clr_fl(fdrd, O_NONBLOCK);

    exit(0);
}
```

14.11 随意读取现行队列中的消息会干扰客户机-服务器协议，导致丢失客户机请求或者服务器的响应。由于队列允许所有的用户读，所以进程只要知道队列的标识符就可以读队列。

14.13 由于服务器和客户机都可能将段连接到不同的地址，所以在共享存储段中不存放实际物理地址。相反，当在共享存储段中建立链表时，指针的值设置为共享存储段内的位移。位移量为所指目标的实际地址减去共享存储段的起始地址。

14.14 表C-1显示了相关的事件。

表C-1 程序14-12中父子进程间的切换过程

父进程中i的值	子进程中i的值	共享变量的值	update的返回值	备注
		0		由mmap初始化
	1			子进程先运行，然后阻塞

(续)

父进程中i的值	子进程中i的值	共享变量的值	update的返回值	备注
0		1		父进程运行
		2	0	父进程阻塞 子进程恢复
		3	1	子进程阻塞 父进程恢复
2	3	3		父进程阻塞
		4	2	父进程阻塞
		5	3	子进程阻塞 父进程恢复
4	5			子进程阻塞 父进程恢复

第15章

15.3 说明 (declaration) 指定了标识符集合的属性 (例如数据类型), 如果说明的同时分配了存储单元就是定义 (definition)。

在头文件 `open.h` 中用 `extern` 说明了三个全局变量, 这时并没有为它们分配存储单元, 在文件 `main.c` 中定义了三个全局变量, 有时会在定义时就初始化全局变量, 但通常使用 C 的默认值。

15.5 `select` 和 `poll` 都返回就绪的描述符个数。当将这些就绪描述符都处理完后, 操作 `client` 数组的循环就可以结束。

第16章

16.1 `_db_dodelete` 中保守的加锁操作是为了避免和 `db_nextrec` 发生竞态条件。如果没有使用写锁保护 `_db_writedat` 调用, 则有可能在 `_db_nextrec` 读某个记录时擦去该记录: `db_nextrec` 首先读入一个索引记录, 发现该记录非空, 则接着读入记录内容, 但是在它调用 `_db_readidx` 后 `_db_readdat` 前, 该记录却被 `_db_dodelete` 删除了。

16.2 假定 `db_nextrec` 调用了 `_db_readidx`, 它将记录的关键字读入索引缓存进行处理。但该处理过程被内核调度进程打断, 另一个执行的进程刚好调用 `db_delete` 删除了这一条记录, 使得索引文件和记录文件中对应的内容都被清空。当第一个进程恢复执行并调用 `_db_readdat` (在 `db_nextrec` 函数体中) 时, 返回的是空记录。所以 `db_nextrec` 中的读锁使得读入记录索引的过程和读入记录内容的过程是一个原子操作 (至少对其他操作同一数据库的并发进程中的写操作而言)。

16.3 强制锁对其他的读者和写者产生了影响——其他的读和写操作都被阻塞, 直到 `_db_writeidx` 和 `_db_writedat` 设置的锁被解除。

第17章

17.1 `psif` 必须读取文件的前两个字节并且与 `%!` 进行比较。如果文件是可以随机定位的, 则可以 `rewind` 文件, 并调用 `lprps` 或 `textps`。如果文件不可随机定位, 则只能将该两个字节重新放回标准输入设备。此时一个可行的办法是: 建立一个管道并 `fork` 一个子进程。然后父进程将其标准输入设备设置为管道, 并执行 `textps` 或 `lprps`。然后子进程将它读到的两个字节写入管道, 再接着将文件的其他部分输出。

第18章

18.2 通常 `getopt` 只用来处理单个参数列表。在 `getopt` 函数的初始化数据段，全局变量 `optind` 被初始化为 1。但是在我们的服务器中，调用 `getopt` 来处理多个参数列表——每个客户机一个，所以必须在为每个客户机首次调用 `getopt` 前均重新初始化 `optind`。

18.3 我们使用了 `Client` 结构维护 `Systems` 文件的位移量。如果在保存了位移量后、再次使用之前修改了该文件，则有可能上次保存的位移量不再指向以前指向的行。虽然服务器可以检测到文件是否被修改了（如何检测？），但是我们无法再将位移量恢复到原来的正确位置。当文件修改后，我们唯一的办法是不让有关用户再登录进来。

18.4 只有当 `client_add` 被调用时，才能通过 `realloc` 移动 `client` 数组。因为 `client_add` 是在 `select` 后，而不是在使用 `cliptr` 的循环中。

18.5 发送到远端系统的不同命令可能会被混淆起来。可以在 `take_put_args` 中加入一些检查功能实现命令的区分。

18.6 一个常用的方法是：要求用户在修改任何文件后通知服务器，以使服务器可以重新读取文件。`SIGHUP` 命令就是经常用来完成这项任务的。

18.9 可以在远端执行 `stty` 命令，然后分析其输出结果。但是考虑到不同 UNIX 平台的 `stty` 命令输出结果差别很大，这种方式实现起来较为困难。

第19章

19.1 `telnetd` 和 `rlogind` 两个服务器均以超级用户的许可权运行，所以它们都可以成功地调用 `chown` 和 `chmod`。

19.3 执行：

```
pty -n stty -a
```

以避免伪终端从设备的 `termios` 结构和 `winsize` 结构初始化。

19.5 很不幸，`fcntl` 的 `F_SETFL` 命令不允许改变读-写状态。

19.6 有三个进程组：(1) 登录 shell，(2) `pty` 父进程和子进程，(3) `cat` 进程。前两个进程组与登录 shell 组成了一个对话期，其中，登录 shell 为对话期首进程。第二个对话期仅包含 `cat` 进程。第一个进程组（登录 shell）是后台进程组，其他两个进程组是前台进程组。

19.7 当接受到文件终止符时，首先是 `cat` 终止，然后是伪终端从设备及伪终端主设备。接着，正从伪终端主设备读取的 `pty` 父进程产生一个文件终止符，该父进程将 `SIGTERM` 信号发送给子进程，子进程终止（子进程不捕捉该信号）。最后，父进程调用 `main` 函数尾端的 `exit(0)`。

程序 8-17 中相关的输出为：

```
cat      e =      270, chars =      274, stat =      0:
pty      e =      262, chars =       40, stat =     15: F      X
pty      e =      288, chars =     188, stat =      0:
```

19.8 这可通过使用 shell 的 `echo` 和 `date(1)` 命令实现：

```
#!/bin/sh
(echo "Script started on " `date`;
pty "${SHELL:-/bin/sh}";
echo "Script done on " `date`) | tee typescript
```

19.9 伪终端上的行规程能够回应，故 `pty` 可以读取其标准输入，并写向伪终端主设备。尽管程序（`ttyname`）从不读取输入，该回应也可通过伪终端上的行规程模块实现。

参 考 书 目

Adobe Systems Inc. 1985. *PostScript Language Tutorial and Cookbook*. Addison-Wesley, Reading, Mass.

“蓝皮书”。

Adobe Systems Inc. 1986. *PostScript Language Reference Manual*. Addison-Wesley, Reading, Mass.

“红皮书”。此书1985年版的附录D包含了通过串行线与PostScript打印机通信的详细信息。1986年版则删除了此部分信息。

Adobe Systems Inc. 1988. *PostScript Language Program Design*. Addison-Wesley, Reading, Mass.

“绿皮书”。此书第12章说明了如何为PostScript打印机编写打印假脱机程序。

Aho, A. V., Kernighan, B. W., and Weinberger, R.J. 1988. *The AWK Programming Language*. Addison-Wesley, Reading, Mass.

本书对awk程序设计语言进行了完整的说明。本书所说明的awk有时被称之为nawk (new awk)。

Andrade, J.M., Carges, M. T., and Kovach, K. R. 1989. "Building a Transaction Processing System on UNIX Systems", *Proceedings of the 1989 USENIX Transaction Processing Workshop*, pp.13-22(May), Pittsburgh, Pa.

说明AT&T Tuxedo事务处理系统。

ANSI. 1989. "American National Standard for Information Systems—Programming Language C", X3.159-1989, ANSI(Dec.).

C语言及标准函数库的官方标准。

此标准可向Global Engineering Documents订购，编号是+1 800 854 7179或+1 714 261 1455。

Arnold, J. Q. 1986. "Shared Libraries on UNIX System V", *Proceedings of the 1986 Summer USENIX Conference*, pp.395-404, Atlanta, Ga.

说明SVR3中共享库的实现。

AT&T. 1989. *System V Interface Definition, Third Edition*. Addison-Wesley, Reading, Mass.

本书为四卷本，说明系统V的源代码界面和运行时的行为。其第3版对应于SVR4。1991年出版了第5卷，它包含了第1~4卷中更新的命令和函数部分。

AT&T. 1990a. *UNIX Research System Programmer's Manual, Tenth Edition, Volume I*. Saunders College Publishing, Fort Worth, Tex.

这是Research UNIX第10版 (V10) 的《UNIX程序员手册》。它包含了传统的UNIX手册页 (第1~9节)。

AT&T. 1990b. *UNIX Research System Papers, Tenth Edition, Volume II*. Saunders College Publishing, Fort Worth, Tex.

Research UNIX第10版 (V10) 第2卷，它包含了说明该系统各个方面的40篇文章。

AT&T. 1990c. *UNIX System V Release 4 BSD/XENIX Compatibility Guide*. Prentice-Hall,

Englewood Cliffs, N. J.

包含说明兼容库的手册页。

AT&T. 1990d. *UNIX System V Release 4 Programmer's Guide. STREAMS*. Prentice-Hall, Englewood Cliffs, N. J.

说明SVR4的STREAMS (流) 系统。

AT&T. 1990e. *UNIX System V/386 Release 4 Programmer's Reference Manual*. Prentice-Hall, Englewood Cliffs, N. J.

本书是针对Intel80386处理器的SVR4实现的程序员参考手册。它包含：第1节(命令)、第2节(系统调用)、第3节(子例程)、第4节(文件格式)和第5节(其他)。

AT&T. 1991. *UNIX System V/386 Release 4 System Administrator's Reference Manual*. Prentice-Hall, Englewood Cliffs, N. J.

本书是针对Intel80386处理器的SVR4实现的管理员参考手册。它包含：第1节(命令)、第4节(文件格式)、第5节(其他)、第7节(特殊文件)。

Bach, M.J. 1986. *The Design of the UNIX Operating System*. Prentice-Hall, Englewood Cliffs, N.J.

本书详细说明UNIX操作系统的设计和实现。虽然本书并不提供UNIX源代码(因为这是AT&T的财产),但提供并讨论了UNIX内核使用的很多算法及数据结构。本书说明的是SVR2。

Bolsky, M. I., and Korn, D. G. 1989. *The KornShell Command and Programming Language*. Prentice-Hall, Englewood Cliffs, N.J.

Chen, D., Barkley, R. E., and Lee, T. P. 1990. "Insuring Improved VM Performance: Some NoFault Policies", *Proceedings of the 1990 Winter USENIX Conference*, pp.11-22, Washington, D.C.

本书说明对SVR4虚存实现的更改,其目的是改善该系统,特别是fork和exec的性能。

Comer, D.E. 1979. "The Ubiquitous B-Tree", *ACM Computing Surveys*, vol. 11, no.2, pp.121-137(June).

Date, C.J. 1982. *An Introduction to Database Systems, Volume II*. Addison-Wesley, Reading, Mass.

Fowler, G. S., Korn, D. G., and Vo, K. P. 1989. "An Efficient File Hierarchy Walker", *Proceedings of the 1989 Summer USENIX Conference*, pp. 173-188, Baltimore, Md.

说明一个新库函数,其作用是遍历文件系统层次结构。

Garfinkel, S., and Spafford, G. 1991. *Practical UNIX Security*. O'Reilly & Associates, Sebastopol, Calif.

本书详细说明UNIX的安全性。

Gingell, R.A., Lee, M., Dang, X.T., and Weeks, M. S. 1987. "Shared Libraries in SunOS", *Proceedings of the 1987 Summer USENIX Conference*, pp.131-145, Phoenix, Ariz.

Gingell, R.A., Moran, J.P., and Shannon, W. A. 1987. "Virtual Memory Architecture in SunOS", *Proceedings of the 1987 Summer USENIX Conference*, pp.81-94, Phoenix, Ariz.

说明mmap函数的起始实现,以及虚存设计中的有关问题。

Goodheart, B. 1991. *UNIX Curses Explained*. Prentice-Hall, Englewood Cliffs, N.J.

本书详细说明terminfo和curses函数库。

Hume, A.G. 1988. "A Tale of Two Greps", *Softw. Pract. and Exper.*, vol. 18, no.11, pp.1063-1072.

IEEE. 1990. "Information Technology—Portable Operating System Interface(POSIX)Part 1: System Application Program Interface(API)[C Language]",1003.1-1990, IEEE(Dec.).

这是第一个POSIX标准，它定义了基于UNIX操作系统的C语言系统界面标准。这常被称为POSIX.1.

Kernighan, B.W., and Pike, R. 1984. *The UNIX Programming Environment*. Prentice-Hall, Englewood Cliffs, N.J.

本书是对UNIX程序设计附加细节的参考书，包含了许多UNIX命令和公用程序，例如grep、sed、awk以及Bourne shell。

Kernighan, B.W., and Ritchie, D.M. 1988. *The C Programming Language, Second Edition*. Prentice-Hall, Englewood Cliffs, N.J.

本书说明C程序设计语言的ANSI标准。附录B中包含了ANSI标准定义的函数库说明。

Kleiman, S.R. 1986."Vnodes: An Architecture for Multiple File System Types in Sun Unix",*Proceedings of the 1986 Summer USENIX Conference*,pp.238-247, Atlanta, Ga.

说明了原先的v节点实现。

Korn, D. G., and Vo, K.P. 1991."SFIO: Safe/Fast String/File IO," *Proceedings of the 1991 Summer USENIX Conference*, pp.235-255, Nashville, Tenn.

说明了标准I/O函数库的一种替换软件，用下列方式发送 e-mail就可得到它：echo 'send atgifts/sfio.shar'|mail netlib @ research.att.com.

Krieger, O., Stumm, M., and Unrau, R. 1992. "Exploiting the Advantages of Mapped Files for Stream I/O", *Proceedings of the 1992 Winter USENIX Conference*, pp. 27-42, San Francisco, Calif.

一种标准I/O函数库的替换软件，它基于映射文件。

Leffler, S.J., McKusick, M.K., Karels, M.J., and Quarterman, J.S. 1989. *The Design and Implementation of the 4.3BSD UNIX Operatin System*. Addison-Wesley, Reading, Mass.

本书对4.3BSD UNIX系统进行完整的说明，所说明的是4.3BSD的Tahoe版。

Libes, D. 1990."expect: Curing Those Uncontrollable Fits of Interaction",*Proceedings of the 1990 Summer USENIX Conference*, pp.183-192, Anaheim, Calif.

对expect程序及其实现的说明。

Libes, D. 1991."expect: Scripts for Controlling Interactive Processes",*Computing Systems*, vol. 4, no.2,pp.99-125(Spring).

本文提供了很多expect脚本。

Morris, R., and Thomopson, K. 1979."UNIX Password Security",*Communications ACM*, vol.22, no.11, pp.594-597(Nov.).

说明UNIX口令方案设计的历史演变。

Nemeth, E., Snyder, G., and Seebass, S. 1989. *UNIX System Administration Handbook*. Prentice-Hall, Englewood Cliffs, N.J.

对如何管理UNIX系统作了详细说明。

Olander, D. J., McGrath, G. J., and Israel, R. K. 1986."A Framework for Networking in System V",*Proceedings of the 1986 Summer USENIX Conference*, pp.38-45, Atlanta, Ga.

本文说明系统V服务界面，系统和TLI的原先实现。

Plauger, P.J. 1992. *The Standard C Library*. Prentice-Hall, Englewood Cliffs, N.J.

本书是一本ANSI C函数库的全书，包含了该库完整的C语言实现。

Presotto, D. L., and Ritchie, D. M. 1990. "Interprocess Communication in the Ninth Edition UNIX System", *Softw.Pract. and Exper.*, vol.20, no.S1, pp.S1/3-S1/17(June).

本文说明UNIX第九版提供的IPC设施，它是由AT&T贝尔实验室的信息科学研究部开发的。这种IPC的基础是流输入、输出系统，它也包括全双工管道，通过它在进程之间可以传送文件描述符，还包括对服务器的唯一客户连接。本文的一个副本也刊载在 AT&T [1990b]。

Redman, B. E. 1989."UUCP UNIX-to-UNIX Copy", in *UNIX Networking*, eds.S. G. Kochan and P.H. Wood, pp.5-48. Howard W. Sams and Company, Indianapolis, Ind.

本章包含了有关Honey DanBer UUCP的附加详细情况，也详细说明了UUCP程序的历史演变。

Ritchie, D. M. 1984. "A Stream Input-Output System", *AT&T Bell Laboratories Technical Journal*, vol.63, no.8, pp. 1897-1910(Oct.).

关于流的第一篇文章。

Seltzer, M., and Olson, M. 1992."LIBTP: Portable, Modular Transactions for UNIX", *Proceedings of the 1992 Winter USENIX Conference*, pp.9-25, San Francisco, Calif.

说明对db(3)库的修改，它来自于实现了事务的4.3+BSD。

Seltzer, M., and Yigit, O. 1991."A New Hashing Package for UNIX", *Proceedings of the 1991 Winter USENIX Conference*, pp.173-184, Dallas, Tex.

说明dtm(3)库及它的各种实现，以及一种新的散列处理软件包。

Stevens, W. R. 1990. *UNIX Network Programming*. Prentice-Hall, Englewood Cliffs, N.J.

详细说明UNIX下的网络程序设计。

Stonebraker, M. R. 1981. "Operating System Support for Database Management", *Communications ACM*, vol.24, no.7, pp.412-418(July).

Strang, J., Mui, L., and O'Reilly, T. 1991. *termcap & terminfo, Third Edition*. O'Reilly & Associates, Sebastopol, Calif.

一本有关termcap和terminfo的参考书。

Thompson, K. 1978. "UNIX Implementation", *Bell Syst. Technical Journal*, vol.57, no.6, pp.1931-1946(July-Aug.).

说明UNIX V 7的某些实现细节。

Weinberger, P. J. 1982."Making UNIX Operating Systems Safe for Database", *Bell Syst. Technical Journal*, vol.61, no.9, pp.2407-2422(Nov.).

说明在早期UNIX系统中实现数据库的某些问题。

Williams, T. 1989."Session Management in System V Release 4", *Proceedings of the 1989 Winter USENIX Conference*, pp.365-375, San Diego, Calif.

说明POSIX.1所要求的，在SVR4中实现的对话期结构，包括进程组、作业控制和控制终端。本书也说明了现存方法的安全性方面。

X/Open. 1989. *X/Open Portability Guide*. Prentice-Hall, Englewood Cliffs, N.J.

本书为七卷本，包括下列各部分：命令和公用程序 (Vol.1)、系统界面和头文件 (Vol.2)、补充定义 (Vol.3)、程序设计语言 (Vol.4)、数据管理 (Vol.5)、窗口管理 (Vol.6) 以及网络服务 (Vol.7)。