第1章　Linux操作系统

1991年3月，Linus Benedict Torvalds为他的AT386计算机买了一个多任务操作系统：Minix。他使用这个操作系统来开发自己的多任务系统，并称之为 Linux。1991年9月，他向Internet网上的其他一些Minix用户发电子邮件，发布了第一个系统原型，这样就揭开了 Linux工程的序幕。从那时起，有许多程序员都开始支持 Linux。他们增加设备驱动程序，开发应用程序，他们的目标是符合 POSIX标准。现在的Linux功能已经非常强大了，但是 Linux更吸引人的地方在于，它是免费的 (当然并不像免费啤酒那样，不是完全免费)。现在人们正在把Linux移植到其他平台上。

第2章　Linux内核

Linux的基础就是它的内核。用户可以替换某个库，或者将所有库都进行替换，但是只要Linux内核存在，它就还是Linux。内核包括设备驱动程序、内核管理、进程管理以及通信管理。内核高手总是遵循POSIX规则，该规则有时会使编程变得简单，有时会使它变得比较复杂。如果用户的程序在一个新的Linux内核版本上行为发生了变化，可能是因为实现了一个新的POSIX规则。如果读者想了解更多的关于Linux内核编程的信息，可以阅读《Linux Kernel Hacker ’s Guide》。

第3章　Linux libc包

Libc：ISO 8859.1；位于<linux/param.h>中；包括YP函数、加密函数、一些基本的影子过程(默认情况不包含)，……在libcompt中有一些为了保持兼容性而提供的老过程 (默认情况下不激活)；提供英文、法文，或者德文的错误信息；在 libcurses中有一些具有bsd 4.4lite兼容性的屏幕处理过程；在libbsd中有一些bsd兼容的过程；在libtermcap中有一些屏幕处理过程；在libdbm中有用于数据库管理的过程；在 libm中有数学过程；在crtO.o???中有执行程序的入口，在libieee???有一些字节信息(请别再笑话我了，能不能给我提供一些信息？)，在libgmon中是用户空间的配置信息。

我希望由某位Linux libc开发人员来编写本章。现在我能说的唯一的一句话是 a.out可执行格式将会变化成elf(可执行并可链接)格式(出版者注：这个变化已经发生了)，而后者又意味着在创建共享库方面的一个变化。当前这两种格式 (a.out和elf)都支持。

Linux libc包的绝大部分都是遵守库 GNU公共许可证的，尽管有些文件是遵守特殊的版权规定的，例如 crtO.o。对商业版本来说，这就意味着一个限制 ,即禁止静态链接可执行程序。在这里动态链接可执行程序又是一个特殊的例外。 FSF(自由软件基金会)的Richard Stallman说过：

在我看来，我们应该明确地允许发行不带伴随库的动态链接可执行程序，只要组成该可执行程序的对象文件按照第 5节的规定是不受限制的。… …，所以我决定现在就允许这样做。实际上，要更新LGPL将需要等到我有时间的时候，并且需要检查一下新版本。

第4章 系统调用

系统调用是向操作系统(内核)所作出的一次申请，请求操作系统做一次硬件/系统相关的操作，或者是作一次只有系统才能做的操作。以Linux 1.2为例，它总共定义了140个系统调用。有些系统调用(如close ( ))是在Linux libc中实现的。这种实现常常需要调用一个宏，而该宏最后会调用syscall ( )。传送给syscall ( )的参数是系统调用的编号，在编号后面的参数是其他一些必需的变元。如果通过实现一个新的libc库而更新了<sys/syscall.h>，则真正的系统调用编号可以在<linux/unistd.h>中找到。如果新的系统调用在libc中还没有代理程序，用户可以使用syscall ( )。下面给出一个例子，可以像下面一样使用syscall ( )来关闭一个文件(不提倡)：

```
#include <syscall.h>

extern int syscall(int, ...);

int my_close(int filedescriptor)
{
   return syscall(SYS_close, filedescriptor);
}
```

在i386体系结构下，除了系统调用编号以外，系统调用只能带5个以下的变元，这是由于受到了硬件寄存器数目的限制。如果读者是在另一个体系结构下运行Linux的，可以检查一下<asm/unistd.h>中的_syscall宏，看看硬件支持多少个变元，或者说开发人员选择支持多少个变元。这些_syscall可以用于取代syscall ( )，但是并不提倡用户这么做，这是因为由这些宏扩展而成的完整的函数有可能在库中早已存在了。

所以，只有内核高手才能去使用_syscall宏。为了说明这一点，下面给出一个使用_syscall宏的close ( )的例子：

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

_syscall1(int, close, int, filedescriptor);
```

在_syscall1宏扩展以后，得到函数close ( )。这样，我们就有两个close ( )了，一个在libc中，另一个在我们的程序中。如果系统调用失败，syscall ( )或者_syscall宏的返回值是－1；而如果系统调用成功，则返回值将是0或者更大的数值，如果系统调用失败，我们可以查看一下全局变量errno，看看到底发生了什么。

下面这些系统调用在BSD和Sys V中是可用的，但Linux 1.2不支持：

audit ( )、audition ( )、auditsvc ( )、fchroot ( )、getauid ( )、getdents ( )、getmsg ( )、mincore ( )、poll ( )、putmsg ( )、setaudit ( )、setauid ( )。

第5章 “瑞士军刀”：ioctl

ioctl代表输入/输出控制，它用于通过文件描述符来操作字符设备。 ioctl的格式如下所示：

ioctl (unsigned int fd, unsigned int request, unsigned long argument)

如果出错则返回值为－ 1，如果请求成功则返回值将大于或者等于 0，这就像其他系统调用一样。内核能区分特殊文件和普通文件。特殊文件一般可以在 /dev和/proc中找到。它们与普通文件的区别在于，它们隐藏了驱动程序的接口，并不是一个包含着文本或二进制数据的真正的(常规的)文件。这是Unix的特点，它允许用户对每一个文件都可以使用普通的读 /写操作。但是，如果用户想要对特殊文件或者普通文件进行更多的处理，用户可以使用……对了，就是ioctl。用户把ioctl用于特殊文件的机会比用于普通文件的机会要多得多，但是在普通文件上也可以使用ioctl。

第6章　Linux进程间通信

下面我们详细地介绍在Linux操作系统中实现的IPC(进程间通信)机制。

6.1　介绍

Linux IPC(进程间通信)机制为多个进程之间相互通信提供了一种方法。对 Linux C程序员来说，有许多种IPC的方法：

- 半双工Unix管道
- FIFO(命名管道)
- SYSV形式的消息队列
- SYSV形式的信号量集合
- SYSV形式的共享内存段
- 网络套接字(Berkeley形式)(本书不作介绍)
- 全双工管道(STREAMS管道)(本书不作介绍)

如果使用得当的话，这些机制将为任意 Unix系统(包括Linux)上的客户机/服务器开发提供一个坚强的框架。

6.2　半双工Unix管道

6.2.1　基本概念

简单地说，管道就是一种把一个进程的标准输出与另一个进程的标准输入相连接的方法。管道是最古老的IPC工具，自从Unix操作系统诞生以来，管道就已经存在了。它们提供了一种进程之间单向通信的方法(这就是术语“半双工”的由来)。

管道这一特征已经被广泛地使用了，甚至在 Unix命令行中(在Shell)中也用到了管道。

图3-6-1显示了建立一个管道，把 ls的输出当作sort的输入，以及把sort的输出当作lp输入进行处理的一系列过程。数据是通过一个半双工管道传输的，从管道的左边传输到管道的右边。

尽管我们中的大多数人都曾经非常频繁地在 shell脚本编程时使用管道，但在我们这么做的时候常常懒得想一下在内核一级到底发生了一些什么事。

在进程创建管道时，内核创建两个文件描述符，以供管道使用。其中一个描述符用于允许输入管道的路径(写)；而另一个用于从管道获得数据 (读)。如果仅仅是这样，管道将没有什么实用性，创建管道的进程只能用该管道与它自己通信。下图显示了在创建管道以后进程和内核的状态。

注意　出版者注：作者尚未提供3-6-1图。

从上图可以很容易地看出这两个文件描述符是如何连接在一起的。如果进程通过管道

(fd0)发送数据，它可以从fd1获得(读)那个信息。然而，相对上面这个简单的草图来说，用户希望实现的目标要大得多。当一个管道把进程与它自己相连接时，在管道中传输的数据需要经过内核。特别是在Linux下，管道实际上是用一个合法索引节点在内部表示的。当然，这个索引节点自己也驻留在内核中，并且不在任何物理文件系统的范围之内，这个特点将为用户提供一些方便的I/O方法，稍后读者就能了解到这一点。

到目前为止，管道显得毫无用处。然而，如果我们只想和自己通信，我们为什么要不辞辛苦地去创建管道呢？就目前来说，创建进程一般都是产生一个子进程。因为子进程将从父母那里继承所有打开的文件描述符，我们现在已经得到了一个多进程通信的框架(在父与子之间通信)。我们上面这个简单的草图经过升级以后如图3-6-2所示。

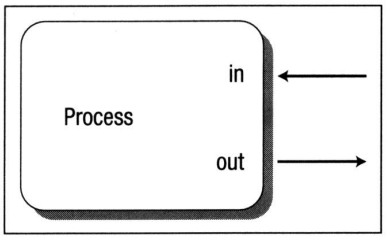

图 3-6-2

现在读者可以看到，两个进程都能访问组成管道的那两个文件描述符。就在这个时候需要作出一个关键的决定，即希望数据向哪个方向传输？是子进程向父进程发送信息，还是反过来？一旦决定以后，两个进程彼此都同意这个决定，并把它们各自所不关心的管道的某一端关掉。为了方便进行介绍，这里假设子进程进行某些处理操作，并通过管道把消息发送回父进程。经过修改以后，图3-6-2变成了图3-6-3。

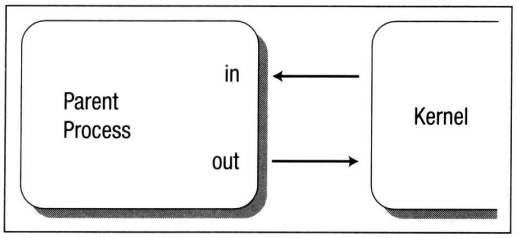

图 3-6-3

管道的创建工作到此全部完成！唯一剩下要做的事就是使用管道了。为了直接访问管道，可以使用那些用来完成低级文件I/O的系统调用(记住，管道实际上在内部是表示成合法的索引节点的)。

为了把数据发送到管道，我们使用 write ( )系统调用；而为了从管道接收数据，我们可以使用 read ( )系统调用。记住，低级文件I/O系统调用是使用文件描述符的！然而，读者需要注意的是，有些特定的系统调用，例如 lseek ( ), 对管道的文件描述符是无效的。

6.2.2 用C语言创建管道

与简单的shell例子比较起来，用高级编程语言 C创建管道要常见一些。为了用 C语言创建一个简单的管道，用户可以使用 pipe ( )系统调用。这个系统调用只需要一个变元，它是由两个整数组成的一个数组。如果调用成功，该数组将会包含管道所使用的两个新的文件描述符。在创建管道以后，进程一般会产生一个新进程 (记住子进程将继承打开的文件描述符)。

```
SYSTEM CALL: pipe();

  PROTOTYPE: int pipe( int fd[2] );
    RETURNS: 0 on success
             -1 on error: errno = EMFILE (no free descriptors)
                                  EMFILE (system file table is full)
                                  EFAULT (fd array is not valid)
```

```
NOTES: fd[0] is set up for reading, fd[1] is set up for writing
```

数组中的第一个函数(元素0)是为了读操作而创建和打开的，而第二个函数 (元素1)是为了写操作而创建和打开的。直观地说， fd1的输出变成了fd0的输入。再说一遍，通过管道传输的所有数据都将经过内核。

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

main()
{
        int     fd[2];

        pipe(fd);
        .
        .
}
```

记住，在C语言中，数组的名称实际上变成了指向它的第一个成员的指针，在上面的程序中，fd相当于&fd[0]。一旦创建了管道，就可以产生子进程了：

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

main()
{
        int     fd[2];
        pid_t   childpid;

        pipe(fd);

        if((childpid = fork()) == -1)
        {
                perror("fork");
                exit(1);
        }
        .
        .

}
```

如果父进程希望从子进程接收到数据，它应该关闭 fd1，而子进程应该关闭fd0。如果父进程希望把数据发送到子进程，它应该关闭 fd0，而子进程应该关闭fd1。因为描述符是在父进程和子进程之间共享的，所以一定要确定我们所关闭的管道的那一端是我们所不关心的。从技术的角度来看，如果管道中不需要的那一端没有显式地关闭的话，将永远不会返回 EOF。

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

```
main()
{
        int     fd[2];
        pid_t   childpid;

        pipe(fd);

        if((childpid = fork()) == -1)
        {
                perror("fork");
                exit(1);
        }

        if(childpid == 0)
        {
                /* Child process closes up input side of pipe */
                close(fd[0]);
        }
        else
        {
                /* Parent process closes up output side of pipe */
                close(fd[1]);
        }
        .
        .
}
```

正如以前所提到的那样，管道一旦创建好以后，它的文件描述符就可以像普通文件的文件描述符一样处理了。

```
/*****************************************************************************
 Excerpt from "Linux Programmer's Guide - Chapter 6"
 (C)opyright 1994-1995, Scott Burkett
 *****************************************************************************
 MODULE: pipe.c
 *****************************************************************************/
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main(void)
{
        int     fd[2], nbytes;
        pid_t   childpid;
        char    string[] = "Hello, world!\n";
        char    readbuffer[80];

        pipe(fd);

        if((childpid = fork()) == -1)
        {
```

```
                perror("fork");
                exit(1);
        }

        if(childpid == 0)
        {
                /* Child process closes up input side of pipe */
                close(fd[0]);

                /* Send "string" through the output side of pipe */
                write(fd[1], string, strlen(string));
                exit(0);
        }
        else
        {
                /* Parent process closes up output side of pipe */
                close(fd[1]);

                /* Read in a string from the pipe */
                nbytes = read(fd[0], readbuffer, sizeof(readbuffer));
                printf("Received string: %s", readbuffer);
        }

        return(0);
}
```

常常会有这样的情况，用户把子进程的文件描述符复制到标准输入或输出。这样，子进程可以用exec ( )来执行另一个程序，而那个程序将得到标准的输入或输出流。请看下面的 dup ( )系统调用：

```
SYSTEM CALL: dup();
  PROTOTYPE: int dup( int oldfd );
    RETURNS: new descriptor on success
             -1 on error: errno = EBADF (oldfd is not a valid descriptor)
                                  EBADF (newfd is out of range)
                                  EMFILE (too many descriptors for the process)
Notes: The old descriptor is not closed! Both may be used interchangeably
```

尽管老的文件描述符和新创建的文件描述符可以互换使用，但一般首先关闭一个标准的输入/输出流。系统调用dup ( )使用编号最低且未被使用的描述符作为新的描述符。

Consider:

```
        .
        .
        childpid = fork();

        if(childpid == 0)
        {
                /* Close up standard input of the child */
                close(0);

                /* Duplicate the input side of pipe to stdin */
```

```
        dup(fd[0]);
        execlp("sort", "sort", NULL);
        .
}
```

尽管文件描述符0(stdin)被关闭了，对dup ( )的调用会把管道的输入描述符 (fd0)复制到它的标准输入上，然后再调用 execlp ( )，以便让排序程序的文本段 (代码)覆盖子进程的文本段 (代码)。因为新执行的程序从它们的创建者那里继承了标准的输入 /输出流，它实际上就继承了管道的输入端作为自己的标准输入！这样一来，原来的父进程发送到管道的任何数据都将传送到排序工具那里。

用户还可以使用另一个系统调用 dup2 ( )。这个特殊的系统调用最早是由第 7版的Unix提供的，在BSD的发行版本中一直沿续了下来，现在已被 POSIX标准列入规范要求。

```
SYSTEM CALL: dup2();

PROTOTYPE: int dup2( int oldfd, int newfd );
  RETURNS: new descriptor on success
           -1 on error: errno = EBADF (oldfd is not a valid descriptor)
                                EBADF (newfd is out of range)
                                EMFILE (too many descriptors for the process)
```

注意 旧的文件描述符是用dup2 ( )关闭的！

使用这个特殊的调用，用户可以在一次系统调用中完成关闭操作以及文件描述符的复制工作。另外，该系统调用保证是原子性的，这就意味着它永远不会被一个突如其来的信号所中断。在把控制权还给内核进行信号调度以前，整个操作将会全部完成。如果用户使用原来的dup ( )系统调用，则程序员在调用它之前需要执行一次 close ( )操作。这就需要进行两次系统调用，在两次系统调用之间的短暂时间里，存在一点点危险的可能性。如果在那个稍纵即逝的瞬间到达了一个信号，则文件描述符的复制将会失败。当然， dup2 ( )会为用户解决这个问题。

Consider:

```
        .
        .
        childpid = fork();

        if(childpid == 0)
        {
                /* Close stdin, duplicate the input side of pipe to stdin */
                dup2(0, fd[0]);
                execlp("sort", "sort", NULL);
                .
                .
        }
```

6.2.3 简便方法

如果读者认为以上这种方法是一种创建和利用管道的间接方法，觉得它不够直观和简便

的话，可以选择使用下面这种方法：

```
LIBRARY FUNCTION: popen();

PROTOTYPE: FILE *popen ( char *command, char *type);
  RETURNS: new file stream on success
           NULL on unsuccessful fork() or pipe() call
Notes: creates a pipe, and performs fork/exec operations using "command."
```

以上这个标准库函数通过在内部调用 pipe ( )创建了一个半双工管道，然后它生成一个子进程，执行Bourne shell，并在shell中执行“ command ”变元。数据流动的方向由第二个变元“ type ”所决定。它的值可以是“ r ”或者“ w ”。“ r ”代表“ 读 ”，而“ w ”代表“ 写 ”，但不能既为读又为写！在Linux下，管道打开的方式是由“ type ”变元的第一个字符决定的。这样一来，如果用户把“ type ”的值置为“ rw ”，则管道只能以“ 读 ”方式打开。

可能读者会认为这个库函数非常实用，但是读者要知道，这么做所付出的代价也不小。在使用pipe ( )系统调用并自己处理fork/exec操作时，用户对管道有着很好的控制能力，但是使用这个库函数使用户失去了控制权。然而，由于这里直接使用 Bourne shell，所以允许在“ command ”变元中使用shell元字符扩展(包括通配符)。

使用popen ( )创建的管道必须用 pclose ( ) 关闭。到现在为止，读者可能已经意识到popen/pclose与标准文件流I/O函数fopen ( )和fclose ( )有着惊人的相似性。

```
LIBRARY FUNCTION: pclose();

PROTOTYPE: int pclose( FILE *stream );
  RETURNS: exit status of wait4() call
           -1 if "stream" is not valid, or if wait4() fails
Notes: waits on the pipe process to terminate, then closes the stream.
```

在由popen ( )所生成的进程上， pclose ( )函数执行一次wait4 ( )命令。当它返回时，它将摧毁管道和文件流。在这里读者可以再次体会到， pclose ( )和普通的基于流的文件I/O fclose ( )函数非常相似。

请看下面的例子。该例为排序命令打开了一个管道，接着开始处理一个字符串数组的排序：

```
/*****************************************************************************
 Excerpt from "Linux Programmer's Guide - Chapter 6"
 (C)opyright 1994-1995, Scott Burkett
 *****************************************************************************
 MODULE: popen1.c
 *****************************************************************************/

#include <stdio.h>

#define MAXSTRS 5

int main(void)
{
        int  cntr;
        FILE *pipe_fp;
        char *strings[MAXSTRS] = { "echo", "bravo", "alpha",
                                   "charlie", "delta"};
```

```
/* Create one way pipe line with call to popen() */
if (( pipe_fp = popen("sort", "w")) == NULL)
{
        perror("popen");
                exit(1);
        }

        /* Processing loop */
        for(cntr=0; cntr<MAXSTRS; cntr++) {
                fputs(strings[cntr], pipe_fp);
                fputc('\n', pipe_fp);
        }

        /* Close the pipe */
        pclose(pipe_fp);

        return(0);
}
```

因为popen ( )使用了shell来完成它的任务，所以可以使用所有的 shell扩展字符和元字符！此外，popen ( )还可以利用一些更高级的技术，例如重定向，甚至还可以利用输出管道技术。请看下面的实例调用：

```
popen("ls ~scottb", "r");
popen("sort > /tmp/foo", "w");
popen("sort | uniq | more", "w");
```

下面的这个小程序是popen ( )的另一个例子，它打开了两个管道 (一个连接到ls命令，另一个连接到sort命令)：

```
/*************************************************************************
 Excerpt from "Linux Programmer's Guide - Chapter 6"
 (C)opyright 1994-1995, Scott Burkett
 *************************************************************************
 MODULE: popen2.c
 *************************************************************************/

#include <stdio.h>

int main(void)
{
        FILE *pipein_fp, *pipeout_fp;
        char readbuf[80];

        /* Create one way pipe line with call to popen() */
        if (( pipein_fp = popen("ls", "r")) == NULL)
        {
                perror("popen");
                exit(1);
        }
```

```
    /* Create one way pipe line with call to popen() */
    if (( pipeout_fp = popen("sort", "w")) == NULL)
    {
            perror("popen");
            exit(1);
    }

    /* Processing loop */
    while(fgets(readbuf, 80, pipein_fp))
            fputs(readbuf, pipeout_fp);

    /* Close the pipes */
    pclose(pipein_fp);
    pclose(pipeout_fp);

    return(0);
}
```

为了对popen ( )进行最后的说明。我们创建了一个普通的程序。该程序在执行的命令和文件名称之间打开一个管道：

```
/*************************************************************************
 Excerpt from "Linux Programmer's Guide - Chapter 6"
 (C)opyright 1994-1995, Scott Burkett
 *************************************************************************
 MODULE: popen3.c
 *************************************************************************/

#include <stdio.h>

int main(int argc, char *argv[])
{
        FILE *pipe_fp, *infile;
        char readbuf[80];

        if( argc != 3) {
                fprintf(stderr, "USAGE:  popen3 [command] [filename]\n");
                exit(1);
        }

        /* Open up input file */
        if (( infile = fopen(argv[2], "rt")) == NULL)
        {
                perror("fopen");
                exit(1);
        }
/* Create one way pipe line with call to popen() */
if (( pipe_fp = popen(argv[1], "w")) == NULL)
{
        perror("popen");
```

```
        exit(1);
    }

    /* Processing loop */
    do {
        fgets(readbuf, 80, infile);
        if(feof(infile)) break;

        fputs(readbuf, pipe_fp);
    } while(!feof(infile));

    fclose(infile);
    pclose(pipe_fp);

    return(0);
}
```

使用以下的命令试着运行这个程序：

```
popen3 sort popen3.c
popen3 cat popen3.c
popen3 more popen3.c
popen3 cat popen3.c | grep main
```

6.2.4 管道的原子操作

如果说一个操作是“原子性”的，那么就不能以任何理由来中断该操作。整个操作都是一次性完成的。在/usr/include/posix1_lim.h中，POSIX标准对管道上的原子操作的最大缓冲区大小是这样规定的：

```
#define _POSIX_PIPE_BUF         512
```

在一次原子操作中，最多可以把512个字节写到管道，或者最多从管道读取512个字节。如果超过了这个范围，操作就将被分开，不能称之为原子操作了。然而，在Linux下，原子操作的限制在“linux/limits.h”中是这样定义的：

```
#define PIPE_BUF        4096
```

读者不难看出，Linux调整了POSIX规定的很少数目的字节数，甚至可以说是大大地调整了这个数目。当包含多个进程时，管道的原子操作变得非常重要。例如，如果写到管道的字节数目超过了一个操作所能达到的原子性限制，并且有多个进程都在写管道，则数据将会被“交叉”或者“分块”。换句话说，一个进程可能会在另一个进程进行写的时候把数据插入到管道中。

6.2.5 关于半双工管道需要注意的几个问题

- 通过创建两个管道，并在子进程中正确地重新分配好文件描述符，用户就可以创建双向管道。
- pipe ( )调用必须在调用fork ( )以前进行，否则子进程将无法继承文件描述符 (对popen ( )来说也是如此)。

• 使用半双工管道互相连接的任意进程必须位于一个相关的进程家族里。因为管道必须受到内核的限制，所以如果进程没有在管道创建者的家族里面，则该进程将无法访问管道。这一点是与命名管道(FIFO)有区别的。

6.3 命名管道

6.3.1 基本概念

命名管道的工作方式与普通的管道非常相似，但也有一些明显的区别。

• 在文件系统中命名管道是以设备特殊文件的形式存在的。
• 不同家族的进程可以通过命名管道共享数据。
• 因为所有的I/O工作都是由共享进程处理的，文件系统中保留命名管道是为了将来使用。

6.3.2 创建FIFO

有许多种方法可以创建命名管道。其中前两者可以直接用 shell来完成。

```
mknod MYFIFO p
mkfifo a=rw MYFIFO
```

上面这两个命令执行同样的操作，只有一个地方存在差异。 mkinfo命令提供了一个挂钩，可以在创建 FIFO文件之后直接改变它的许可。而如果用户使用 mknod，则需要立刻调用chmod命令。

在物理文件系统中，用户可以很快找到 FIFO文件，因为在长目录列表中， FIFO文件有一个“ p ”指示符。

```
$ ls -l MYFIFO
prw-r--r--   1 root     root            0 Dec 14 22:15 MYFIFO|
```

同时读者还可以注意到，在文件名的后面有一条竖线 (“ 管道符号 ”)。由此不难体会到运行Linux无所不在的好处。

为了用C语言创建FIFO，用户可以使用mknod ( )系统调用：

```
mknod("/tmp/MYFIFO", S_IFIFO|0666, 0);
```

the file “/tmp/MYFIFO” is created as a FIFO file. The request
ough they are affected by the umask setting as follows:

```
final_umask = requested_permissions & ~original_umask
```

rick is to use the umask() system call to temporarily zap the

```
umask(0);
mknod("/tmp/MYFIFO", S_IFIFO|0666, 0);
```

关于mknod ( )更详细的介绍，用户可以参考 man的帮助信息，现在请看用C语言创建FIFO的一个简单的例子：

```
mknod("/tmp/MYFIFO", S_IFIFO|0666, 0);
```

在上面这个例子中，文件 /tmp/MYFIFO是当作一个FIFO文件而创建的。所申请的访问许

可权限是“ 0666 ”，尽管该权限会受到如下定义的掩码的影响：

```
final_umask = requested_permissions & ~original_umask
```

一个常见的技巧是使用umask ( )系统调用暂时屏蔽掉掩码的值：

```
umask(0);
mknod("/tmp/MYFIFO", S_IFIFO|0666, 0);
```

此外， mknod ( )的第三个参数可以忽略，除非用户创建的是设备文件，在那种情况下，mknod ( )应该指明设备文件的主编号和从编号。

6.3.3 FIFO操作

对FIFO来说， I/O操作与普通的管道I/O操作基本上是一样的，但也存在着一个主要的区别。在FIFO中，必须使用一个“ open ”系统调用或者库函数来物理地建立联接到管道的通道。而对于半双工管道而言，这是不必要的，因为管道是驻留在内核中的，而不是驻留在物理文件系统上的。在下面的例子中，我们将把管道当作一个流来看待，使用 fopen ( )来打开它，并使用fclose ( )来关闭它。

请看下面这个简单的服务器进程：

```
/*****************************************************************************
 Excerpt from "Linux Programmer's Guide - Chapter 6"
 (C)opyright 1994-1995, Scott Burkett
 *****************************************************************************
 MODULE: fifoserver.c
 *****************************************************************************/

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

#include <linux/stat.h>

#define FIFO_FILE       "MYFIFO"

int main(void)
{
        FILE *fp;
        char readbuf[80];

        /* Create the FIFO if it does not exist */
        umask(0);
        mknod(FIFO_FILE, S_IFIFO|0666, 0);

        while(1)
        {
                fp = fopen(FIFO_FILE, "r");
                fgets(readbuf, 80, fp);
                printf("Received string: %s\n", readbuf);
                fclose(fp);
```

```
        }

        return(0);
}
```

因为FIFO是默认阻塞的，所以在编译了这个服务器进程之后，可以在后台运行它：

```
$ fifoserver&
```

我们稍后就将讨论FIFO的阻塞动作。首先，请看上面这个服务器进程的简单的客户前端，如下所示：

```
/*****************************************************************************
 Excerpt from "Linux Programmer's Guide - Chapter 6"
 (C)opyright 1994-1995, Scott Burkett
 *****************************************************************************
 MODULE: fifoclient.c
 *****************************************************************************/

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

#define FIFO_FILE       "MYFIFO"

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

        if ( argc != 2 ) {
                printf("USAGE: fifoclient [string]\n");
                exit(1);
        }

        if((fp = fopen(FIFO_FILE, "w")) == NULL) {
                perror("fopen");
                exit(1);
        }

        fputs(argv[1], fp);

        fclose(fp);
        return(0);
}
```

6.3.4 FIFO上的阻塞动作

一般来说FIFO总是处于阻塞状态。换句话说，如果 FIFO是为了读操作而打开的，则进程将“阻塞”，直到某些其他进程打开该FIFO并且写数据。这个阻塞动作反过来也是成立的。如果用户不希望发生阻塞，可以在 open ( )调用中使用O_NONBLOCK标志，以关闭默认的阻塞动作。

在上面那个简单的服务器进程的例子中，我们只是把它放在后台，让它在那里进行它的

阻塞动作。用户还可以选择跳转到另一个虚拟控制台，并运行客户前端，来回切换以查看所导致的动作。

6.3.5 SIGPIPE信号

最后需要注意的一点是，管道必须有人读并有人写。如果某个进程企图往管道中写数据，而没有进程去读该管道，则内核将向该进程发送 SIGPIPE信号。当在管道操作中涉及到两个以上的进程时，这个信号是非常必要的。

6.4 系统V IPC

6.4.1 基本概念

在Unix系统V中，AT&T引入了三种新的IPC方法(消息队列、信号量和共享内存)。POSIX委员会还没有完全制定好这些工具的标准，但大多数实现已经支持这三种 IPC方法。此外，Berkeley (BSD)使用套接字来作为其IPC的主要方法，而不是采用系统V的方法。Linux可以同时使用这两种IPC方法(BSD和系统V)。在稍后的某章中将会介绍套接字。

在Linux中，系统V IPC的实现是由Krishna Balasubramanian完成的，他的电子邮箱地址是balasub@cis.ohio_state.edu。

1. IPC标识符

每一个IPC对象都有一个独一无二的IPC标识符与之联系。这里所说的“ IPC对象 ”是指单个的消息队列、信号量集合或者共享内存段。在内核中这个标识符可以用于唯一地标识一个IPC对象。例如，为了访问某个特定的共享内存段，用户唯一需要的是赋给那个内存段的独一无二的ID值。

标识符的唯一性是与对象的类型相关的。为了说明这一点，假设有一个数字标识符“ 12345 ”。当然不可能会有两个消息队列都对应到这个标识符。但还是存在一种明显的可能性，即某个消息队列和某个共享内存段都具有这个数字标识符。

2. IPC关键字

为了获得一个唯一的ID号。必须使用关键字，客户进程和服务器进程双方都必须同意这个关键字，这是构造应用程序的客户/服务器框架的第一步。

当你给某人打电话时，你必须知道他的电话号码。此外，电话公司必须知道如何把你发出的呼叫转达到它最终的目的地。一旦对方提起电话，开始作出响应，联接就建立起来了。

在系统V IPC方法中，“ 电话 ”直接对应到使用的对象的类型，而“ 电话公司 ”，或者说寻找路由的方法，则与IPC关键字有着直接的联系。

关键字对不同的对象可以具有相同的值，它是通过把关键字的值硬编码进应用程序来实现的。这样做有一个缺点，就是关键字可能早已经在使用了。用户通常可以使用 ftok ( )函数来为客户进程和服务器进程生成关键字的值。

```
LIBRARY FUNCTION: ftok();
PROTOTYPE: key_t ftok ( char *pathname, char proj );
  RETURNS: new IPC key value if successful
           -1 if unsuccessful, errno set to return of stat() call
```

ftok ( )返回的关键字值是这样生成的：把索引节点号、第一个变元中文件的次设备编号，以及第二个变元中的工程标识符 (一个字符)组合起来，这就是返回值。这样的返回值不能保证是唯一的，但应用程序可以查找到是否存在冲突，如果有的话则重新试着生成关键字。

```
key_t   mykey;
mykey = ftok("/tmp/myapp", 'a');
```

在上面这个短的代码片段中，目录 /tmp/myapp与 ’ a ’ 这个标识符组合在一起。另一个通用的例子是使用当前目录：

```
key_t   mykey;
mykey = ftok(".", 'a');
```

采取什么样的关键字生成算法完全是由应用程序编程人员决定的。只要可以防止竞争条件、死锁等等，无论采取什么方法都可以。为了方便进行介绍，这里将使用 ftok ( )方法。如果用户能确定每个客户进程都是从独一无二的“ home ”目录运行的，则所产生的关键字应该能够满足需要。

无论关键字的值是如何产生的，它将用于后续的 IPC系统调用中，用来创建 IPC对象或者获取对它的访问权限。

3. ipcs命令

ipcs命令可以用来获取所有系统 V IPC对象的状态。这个工具的 Linux版本也是由 Krishna Balasubramanian编写的。

```
ipcs      -q:    Show only message queues
ipcs      -s:    Show only semaphores
ipcs      -m:    Show only shared memory
ipcs --help:     Additional arguments
```

在默认的情况下，所有三种类型的 IPC对象都将显示出来。请看下面这个 ipcs命令的输出示例：

```
------ Shared Memory Segments --------
shmid      owner      perms      bytes      nattch     status

------ Semaphore Arrays --------
semid      owner      perms      nsems      status
 ------ Message Queues --------
 msqid      owner      perms      used-bytes  messages
 0          root       660        5           1
```

从上面这个输出中读者可以找到一个具有标识符“ 0 ”的消息队列，它是由 root用户所拥有的，具有八进制形式的访问权限 660，即-rw-rw--。在此队列中有一条消息，而那个消息它的大小总共为5个字节。

ipcs命令是一个功能非常强大的工具，它使用户可以研究内核中 IPC对象的存储机制。读者应该学好它，用好它。

4. ipcrm命令

ipcrm命令可以用来从内核删除一个 IPC对象。IPC对象可以通过在用户代码中调用系统调用来删除(稍后将介绍这方面的内容)。然而，用户常常会需要手工删除 IPC对象，尤其是在开发环境下，ipcrm命令的用法非常简单：

```
ipcrm <msg | sem | shm>  <IPC ID>
```

只需指定想要删除的对象是消息队列(msg)、信号量集合(sem)还是共享内存段(shm)即可。IPC ID号可以使用ipcs命令来获得。用户必须指明对象的类型，因为在相同类型的对象之中，标识符是各不相同的(参见前面的讨论)。

6.4.2 消息队列

1. 基本概念

消息队列的最佳定义是：内核地址空间中的内部链表。消息可以顺序地发送到队列中，并以几种不同的方式从队列中获取。当然，每个消息队列都是由 IPC标识符所唯一标识的。

2. 内部和用户数据结构

要完成理解象系统 V IPC这样复杂的问题，关键是要彻底熟悉内核的几个内部数据结构。甚至对那些最基本的操作来说，直接访问这些结构中的某几个结构也是必要的，而其他的结构则停留在一个更低的级别上。

3. 消息缓冲区

我们要介绍的第一个结构是 msgbuf结构。这个特殊的数据结构可以认为是消息数据的模板。虽然定义这种类型的数据结构是程序员的职责，但是读者绝对必须知道实际上存在msgbuf类型的结构。在linux/msg.h中，它的定义如下所示：

```
/* message buffer for msgsnd and msgrcv calls */
struct msgbuf {
    long mtype;         /* type of message */
    char mtext[1];      /* message text */
};
```

在msgbuf结构中有两个成员：

• mtype —— 它是消息类型，以正数来表示。这个数必须为一个正数！

• mtext —— 它就是消息数据。

因为用户可以给特定的消息赋予一个类型，这样用户就能够在一个消息队列上进行消息的多路传送。例如，用户必须赋予客户进程一个幻数，这个幻数可以用作服务器进程所发送的消息的消息类型。而服务器本身也可以使用其他的一些数，客户可以使用这些数向它发送消息。在另一种情况下，应用程序可以把错误消息标记为具有消息类型号 1的消息，而请求消息的类型为2，等等。这种可能性是不胜枚举的。

读者需要注意的另一点是，不要被被消息数据元素 (mtext)的描述性太强的名称所误导，这个域并不是只能存放字符数据，它还能存放任意形式的任意数据。因为应用程序编程人员可以重新定义msgbuf结构，所以mtext域实际上是随机性很强的。请看下面的例子：

```
struct my_msgbuf {
        long    mtype;          /* Message type */
        long    request_id;     /* Request identifier */
        struct  client info;    /* Client information structure */
};
```

在这个定义中也有消息类型，这点与以前的定义一样，但是结构的剩余部分则被替换成两个其他的元素，其中有一个是另一种结构！这就是消息队列的可爱之处。这样一来，不论是哪种类型的数据，内核均无需作转换工作，任意信息均可以发送。

然而，对于给定消息的最大的大小，确实存在一个内部的限制。在 Linux中，它在

linux/msg.h中是这样定义的：

```
#define MSGMAX  4056   /* <= 4056 */   /* max size of message (bytes) */
```

消息总的大小不能超过4,056个字节，这其中包括mtype成员，它的长度是4个字节(long类型)。

4. 内核msg结构

内核把消息队列中的每个消息都存放在msg结构的框架中。该结构是在linux/msg.h中定义的，如下所示：

```
/* one msg structure for each message */
struct msg {
    struct msg *msg_next;   /* next message on queue */
    long  msg_type;
    char *msg_spot;         /* message text address */
    short msg_ts;           /* message text size */
};
```

- msg_next —— 这是一个指针，指向消息队列中的下一个消息。在内核寻址空间中，它们是当作一个链表存储的。
- msg_type —— 这是消息类型，它的值是在用户结构msgbuf中赋予的。
- msg_spot —— 这是一个指针，指向消息体的开始处。
- msg_ts —— 这是消息文本(消息体)的长度。
- 内核msgid_ds结构 —— IPC对象分为三类，每一类都有一个内部数据结构，该数据结构是由内核维护的。对于消息队列而言，它的内部数据结构是 msqid_ds结构。对于系统上创建的每个消息队列，内核均为其创建、存储和维护该结构的一个实例。该结构在linux/msg.h中定义，如下所示：

```
/* one msqid structure for each queue on the system */
struct msqid_ds {
    struct ipc_perm msg_perm;
    struct msg *msg_first;  /* first message on queue */
    struct msg *msg_last;   /* last message in queue */
    time_t msg_stime;       /* last msgsnd time */
    time_t msg_rtime;       /* last msgrcv time */
    time_t msg_ctime;       /* last change time */
    struct wait_queue *wwait;
    struct wait_queue *rwait;
    ushort msg_cbytes;
    ushort msg_qnum;
    ushort msg_qbytes;      /* max number of bytes on queue */
    ushort msg_lspid;       /* pid of last msgsnd */
    ushort msg_lrpid;       /* last receive pid */
};
```

虽然读者可能很少会关心这个结构的大部分成员，为了叙述的完整性，下面还是对每个成员都给出一个简短的介绍：

- msg_perm —— 它是ipc_perm结构的一个实例， ipc_perm结构是在linux/ipc.h中定义的。该成员存放的是消息队列的许可权限信息，其中包括访问许可信息，以及队列的创建者

的有关信息(如uid等等)。

- msg_first —— 链接到队列中的第一个消息(列表头部)。
- msg_last —— 链接到队列中的最后一个消息(列表尾部)。
- msg_stime —— 发送到队列的最后一个消息的时间戳(time_t)。
- msg_rtime —— 从队列中获取的最后一个消息的时间戳。
- msg_ctime —— 对队列进行最后一次变动的时间戳(稍后将作详细介绍)。
- wwait和rwait —— 是两个指针，指向内核的等待队列。如果消息队列上的某次操作使进程进入睡眠状态，则需要使用这两个成员！(类似的操作包括：队列已满，进程等待一次打开操作)。
- msg_cbytes —— 在队列上所驻留的字节的总数(即所有消息的大小的总和)。
- msg_qnum —— 当前处于队列中的消息数目。
- msg_qbytes —— 队列中能容纳的字节的最大数目。
- msg_lspid —— 发送最后一个消息的进程的PID。
- msg_lrpid —— 接收最后一个消息的进程的PID。

5. 内核ipc_perm结构

内核把IPC对象的许可权限信息存放在ipc_perm类型的结构中。例如在前面描述的某个消息队列的内部结构中，msg_perm成员就是ipc_perm类型的，它的定义是在文件linux/ipc.h中，如下所示：

```
struct ipc_perm
{
  key_t  key;
  ushort uid;   /* owner euid and egid */
  ushort gid;
  ushort cuid;  /* creator euid and egid */
  ushort cgid;
  ushort mode;  /* access modes see mode flags below */
  ushort seq;   /* slot usage sequence number */
};
```

以上所有的成员都具有相当的自扩展性。对象的创建者以及所有者(它们可能会有不同)的有关信息，以及对象的IPC关键字都是存放在该结构中的。八进制形式的访问模式也是存放在这里的，它是以一种无符号短整型的形式存储的。最后，时间片使用序列编号存放在最后面，每次通过系统调用关闭IPC对象(摧毁)时，这个值将被增加一，至多可以增加到能驻留在系统中的IPC对象的最大数目。用户需要关心这个值吗？答案是“不”。

有关这个问题，在Richard Stevens所著的《Unix Network Programming》一书的第125页中作了精辟的讨论。该书还介绍了ipc_perm结构的存在和行为在安全性方面的原因。

6. 系统调用：msgget ( )

为了创建一个新的消息队列，或者访问一个现有的队列，可以使用系统调用 msgget ( )。

```
SYSTEM CALL: msgget();

PROTOTYPE: int msgget ( key_t key, int msgflg );
  RETURNS: message queue identifier on success
           -1 on error: errno = EACCESS (permission denied)
```

```
EEXIST (Queue exists, cannot create)
EIDRM (Queue is marked for deletion)
ENOENT (Queue does not exist)
ENOMEM (Not enough memory to create queue)
ENOSPC (Maximum queue limit exceeded)
```

msgget ( )的第一个变元是关键字的值(在我们的例子中该值是调用 ftok ( )的返回值)。这个关键字的值将被拿来与内核中其他消息队列的现有关键字值相比较。比较之后，打开或者访问操作依赖于msgflg变元的内容。

• IPC_CREAT —— 如果在内核中不存在该队列，则创建它。

• IPC_EXCL —— 当与IPC_CREAT一起使用时，如果队列早已存在则将出错。

如果只使用了IPC_CREAT, msgget ( )或者返回新创建消息队列的消息队列标识符，或者会返回现有的具有同一个关键字值的队列的标识符。如果同时使用了 IPC_EXCL和IPC_CREAT，那么将可能会有两个结果。或者创建一个新的队列，或者如果该队列存在，则调用将出错，并返回－1。IPC_EXCL本身是没有什么用处的，但在与IPC_CREAT组合使用时，它可以用于保证没有一个现存的队列为了访问而被打开。

有个可选的八进制许可模式，它是与掩码进行 OR操作以后得到的。这是因为从功能上讲，每个IPC对象的访问许可权限与Unix文件系统的文件许可权限是相似的！

下面我们创建一个包装程序，它可用于打开或者创建消息队列：

```
int open_queue( key_t keyval )
{
        int     qid;

        if((qid = msgget( keyval, IPC_CREAT | 0660 )) == -1)
        {
                return(-1);
        }

        return(qid);
}
```

注意程序中使用了显式的许可权限 0660。这个小函数或者返回消息队列标识符 (int类型)，或者出错返回－1。必须出错返回－1。必须把关键字的值传给它，这是它唯一的变元。

7. 系统调用：msgsnd ( )

一旦获得了队列标识符，用户就可以开始在该消息队列上执行相关操作了。为了向队列传递消息，用户可以使用 msgsnd系统调用：

```
  SYSTEM CALL: msgsnd();

  PROTOTYPE: int msgsnd ( int msqid, struct msgbuf *msgp, int msgsz, int msgflg
);
    RETURNS: 0 on success
             -1 on error: errno = EAGAIN (queue is full, and IPC_NOWAIT was
asserted)
                                  EACCES (permission denied, no write permis-
sion)
```

```
                                        EFAULT (msgp address isn't accessable -
invalid)

                                        EIDRM  (The message queue has been removed)
                                        EINTR  (Received a signal while waiting to
write)

                                        EINVAL (Invalid message queue identifier,
nonpositive

                                                message type, or invalid message
size)

                                        ENOMEM (Not enough memory to copy message
buffer)
```

msgsnd的第一个变元是队列标识符，它是前面调用 msgget获得的返回值。第二个变元是msgp，它是一个指针，指向我们重新定义和载入的消息缓冲区。 msgsz变元则包含着消息的大小，它是以字节为单位的，其中不包括消息类型的长度 (四个字节长)。

msgflg变元可以设置为0(忽略)，也可以设置为IPC_NOWAIT。如果消息队列已满，则消息将不会被写入到队列中，控制权将被还给调用进程。如果没有指定 IPC_NOWAIT，则调用进程将被中断(阻塞)，直到可以写消息为止。

下面创建另一个发送消息的包装程序：

```
int send_message( int qid, struct mymsgbuf *qbuf )
{
        int     result, length;

        /* The length is essentially the size of the structure minus
sizeof(mtype) */
        length = sizeof(struct mymsgbuf) - sizeof(long);
        if((result = msgsnd( qid, qbuf, length, 0)) == -1)
        {
                return(-1);
        }

        return(result);
}
```

这个小函数接收到一个地址 (qbuf)，并试着把驻留在那个地址中的消息发送到消息队列标志符(qid)所指定的消息队列中， qid也是作为一个参数传送给该函数的。下面这个示例代码片段将用到前面创建的那两个包装函数：

```
#include <stdio.h>
#include <stdlib.h>
#include <linux/ipc.h>
#include <linux/msg.h>

main()
{
        int    qid;
        key_t  msgkey;
        struct mymsgbuf {
                long    mtype;          /* Message type */
                int     request;        /* Work request number */
```

```
        double  salary;         /* Employee's salary */
    } msg;

    /* Generate our IPC key value */
    msgkey = ftok(".", 'm');

    /* Open/create the queue */
    if(( qid = open_queue( msgkey)) == -1) {
            perror("open_queue");
            exit(1);
    }

    /* Load up the message with arbitrary test data */
    msg.mtype   = 1;        /* Message type must be a positive number! */
    msg.request = 1;        /* Data element #1 */
    msg.salary  = 1000.00;  /* Data element #2 (my yearly salary!) */

    /* Bombs away! */
    if((send_message( qid, &msg )) == -1) {
            perror("send_message");
            exit(1);
    }
}
```

在创建或者打开了消息队列以后，接下去用测试数据装填消息缓冲区 (注意这里没有使用字符数据，这是为了说明我们有关传送二进制信息的观点)，调用send_message，它会把消息传送到消息队列中。

现在既然消息队列有一个消息，用户可以使用 ipcs命令来查看队列的状态。下面将介绍如何真正从队列中获取消息。为了做到这点，可以使用系统调用 msgrcv ( )：

```
  SYSTEM CALL: msgrcv();
  PROTOTYPE: int msgrcv ( int msqid, struct msgbuf *msgp, int msgsz, long
mtype, int msgflg );
    RETURNS: Number of bytes copied into message buffer
             -1 on error: errno = E2BIG  (Message length is greater than msgsz,
no MSG_NOERROR)
                                  EACCES (No read permission)
                                  EFAULT (Address pointed to by msgp is in-
valid)
                                  EIDRM  (Queue was removed during retrieval)
                                  EINTR  (Interrupted by arriving signal)
                                  EINVAL (msgqid invalid, or msgsz less than 0)
                                  ENOMSG (IPC_NOWAIT asserted, and no message
exists
                                          in the queue to satisfy the request)
```

显然，第一个变元是用来指定在消息获取过程中所使用的队列的 (该值是由前面调用msgget得到的返回值)。第二个变元(msgp)代表消息缓冲区变量的地址，获取的消息将存放在这里。第三个变元(msgsz)代表消息缓冲区结构的大小，不包括 mtype成员的长度。再说一遍，可以使用下面的公式来计算大小：

```
msgsz = sizeof(struct mymsgbuf) - sizeof(long);
```

第四个变元(mtype)指定要从队列中获取的消息的类型。内核将查找队列中具有匹配类型的最老的消息，并把它的一个拷贝返回到由 msgp变元所指定的地址中。这里存在一种特殊的情况：如果mtype变元传送一个为零的值，则将返回队列中最老的消息，不管该消息的类型是什么。

如果把IPC_NOWAIT作为一个标志传送给该系统调用，而队列中没有任何消息。则该次调用将会向调用进程返回ENOMSG。否则，调用进程将阻塞，直到满足 msgrcv ( )参数的消息到达队列为止。如果在客户等待消息的时候队列被删除了，则返回 EIDRM。如果在进程阻塞并等待消息的到来时捕获到一个信号，则返回 EINTR。

请看下面这个从队列中获取消息的包装函数：

```
int read_message( int qid, long type, struct mymsgbuf *qbuf )
{
        int     result, length;

        /* The length is essentially the size of the structure minus
sizeof(mtype) */
        length = sizeof(struct mymsgbuf) - sizeof(long);

        if((result = msgrcv( qid, qbuf, length, type,  0)) == -1)
        {
                return(-1);
        }

        return(result);
}
```

在成功地从队列中获取了一个消息之后，队列中的消息项目将被摧毁。

msgflg变元中的MSG_NOERROR位提供了一些其他的功能。如果物理消息数据的大小比msgsz要大，同时又声明了MSG_NOERROR位，则消息将被截断，并只返回 msgsz个字节。一般来说，msgrcv ( )系统调用将返回－1(E2BIG)，并且该消息将保留在队列中，以供以后获取。可以利用这个特点来创建另一个包装程序，该程序使用户能够窥探队列的内部，看看是否有满足请求的消息到达：

```
int peek_message( int qid, long type )
{
        int     result, length;

        if((result = msgrcv( qid, NULL, 0, type,  IPC_NOWAIT)) == -1)
        {
                if(errno == E2BIG)
                        return(TRUE);
        }

        return(FALSE);
}
```

在上面这个程序中，细心的读者可以发现缺少缓冲区地址和长度。在这种特殊的情况下，

我们希望调用出错。然而，我们可以检测 E2BIG的返回，它显示了确实存在匹配所申请的类型的消息。如果成功，则包装程序将返回 TRUE，否则将返回 FALSE。同时请读者注意IPC_NOWAIT标志的使用，它防止了前面介绍过的阻塞行为的发生。

系统调用：msgctl ( )

通过前面介绍的那些包装程序的开发，读者现在应该知道怎样在应用程序中简单地，但同时也是聪明地创建和利用消息队列。下面再回头介绍一下如何直接地对那些与特定的消息队列相联系的内部结构进行操作。

为了在一个消息队列上执行控制操作，用户可以使用 msgctl ( )系统调用。

```
  SYSTEM CALL: msgctl();
  PROTOTYPE: int msgctl ( int msgqid, int cmd, struct msqid_ds *buf );
    RETURNS: 0 on success
             -1 on error: errno = EACCES (No read permission and cmd is
IPC_STAT)
                                  EFAULT (Address pointed to by buf is invalid
with IPC_SET and
                                          IPC_STAT commands)
                                  EIDRM  (Queue was removed during retrieval)
                                  EINVAL (msgqid invalid, or msgsz less than 0)
                                  EPERM  (IPC_SET or IPC_RMID command was
issued, but
                                          calling process does not have write
(alter)
                                          access to the queue)
  NOTES:
```

现在有一种普遍的观点，认为直接对内部内核数据结构进行操作会给人带来一种满足感和成就感。不幸的是，由此而给程序员方面带来的责任却不是那么轻松的，除非用户喜欢破坏自己的IPC子系统。通过使用具有一个命令可选集合的 msgctl ( )，用户可以操纵那些不太容易造成破坏的项目。请看这些命令：

IPC_STAT

获取队列的msqid_ds结构，并把它存放在buf变元所指定的地址中。

IPC_SET

设置队列的msqid_ds结构的ipc_perm成员的值。它是从buf变元中取得该值的。

IPC_RMID

从内核删除队列。

前面介绍过消息队列的内部数据结构 (msqid_ds)。对于系统中存在的每一个队列，内核为它们都维护该结构的一个实例。通过使用 IPC_STAT命令，用户可以获取该结构的一个拷贝以供检查。请看下面这个包装程序函数，它将获取内部结构，并将其拷贝到作为参数传送给它的一个地址中：

```
int get_queue_ds( int qid, struct msgqid_ds *qbuf )
{
        if( msgctl( qid, IPC_STAT, qbuf) == -1)
        {
                return(-1);
        }
```

```
        return(0);
}
```

如果不能拷贝这个内部缓冲区，函数将向调用函数返回－ 1。如果一切工作顺利，则将返回一个为零的值，并且缓冲区中将会包含队列标识符 (qid)所代表的消息队列的内部数据结构的一个拷贝。缓冲区和qid都是作为参数传送给该函数的。

既然已经拥有了队列的内部数据结构的一个拷贝，那么，用户可以操作什么属性？用户应该怎样改变它们呢？在那个数据结构中，唯一可以修改的项目就是 ipc_perm成员。它包含该队列的许可权限，以及关于拥有者和创建者的信息。然而， ipc_perm结构中唯一可以修改的成员是mode、uid和gid。用户可以改变拥有者的用户 ID，拥有者的组ID以及队列的访问许可权限。

下面创建一个包装程序函数，用于改变队列的模式。该模式必须以字符数组的形式传送(如“ 660 ”)。

```
int change_queue_mode( int qid, char *mode )
{
        struct msqid_ds tmpbuf;

        /* Retrieve a current copy of the internal data structure */
        get_queue_ds( qid, &tmpbuf);

        /* Change the permissions using an old trick */
        sscanf(mode, "%ho", &tmpbuf.msg_perm.mode);

        /* Update the internal data structure */
        if( msgctl( qid, IPC_SET, &tmpbuf) == -1)
        {
                return(-1);
        }

        return(0);
}
```

通过调用get_queue_ds包装程序函数，可以获取内部数据结构的当前拷贝。然后再调用sscanf ( )，改变相联系的msg_perm结构的方式成员。然而，在用新拷贝更新当前内部版本之前，没有发生任何变化。这是通过使用 IPC_SET命令调用msgctl ( )来实现的。

注意！用户可以修改队列的许可权限。然而在实施过程中稍微的粗心大意会导致难以预料的后果。记住，这些IPC对象不会自动消失，除非它们被正确地删除，或者系统被重启。所以，即使用ipcs命令无法看到队列，也不意味着它不存在了。

为了说明这一点，有一则在某种程度上可以称得上幽默的轶事比较说明问题。在南佛罗里达大学教一门Unix内部结构课程时，我曾经遇到过一个相对尴尬的问题。为了编译和测试一个实验例题以便在我一个星期长的教学中使用，我在上课前的晚上进入实验服务程序。在测试过程中，我发现在用于改变某消息队列的许可权限的代码中，出现了一个打字错误。我创建了一个简单的消息队列，在测试发送和接收能力时并没有什么问题，然而当我试图把队列的访问许可方式从“ 660 ”变为“ 600 ”时，所带来的后果居然是我无法访问自己的队列了。因此，在源目录的同一个目录下，我不能测试消息队列的实验例题。因为我使用 ftok ( )函数

来创建IPC关键字，所以实际上我是企图访问自己没有正确访问权限的队列。最后，在上课前的那个早晨，我联系到了本地系统管理员，花了整整一个小时向他解释这是个什么样的消息队列，而我为什么需要他执行一次ipcrm命令帮我删除该队列。

在成功地从队列获取了一个消息以后，该消息将被删除。然而，正如前面所介绍的，除非用户显式地删除它，或者系统被重启，否则 IPC对象将保留在系统中。因此，消息队列还存在于内核中，在单个消息消失以后很长时间内仍然可用。为了结束一个消息队列的生命周期，用户需要使用IPC_RMID命令来调用msgctl ( )，以便显式地删除它：

```
int remove_queue( int qid )
{
        if( msgctl( qid, IPC_RMID, 0) == -1)
        {
                return(-1);
        }

        return(0);
}
```

如果队列被删除且无错误发生，则上面这个包装程序函数将返回 0，否则就将返回－ 1。队列的删除动作是原子性的，并且以后不论用什么理由访问该队列都将失败。

8. msgtool：交互式消息队列操作程序

如果随时可以得到正确的技术方面的信息，那将会给用户带来很直接的利益，没有人能否认这一点。对于学习和研究新的技术领域，信息和资料会提供一个强有力的帮助。同理可知，如果伴随着这些技术信息还能提供一些实际应用的例子，这无疑会加速学习进程，增强学习效果。

到目前为止，本书所提供的几个有用的例子是操作消息队列的包装程序函数。虽然它们是非常有用的，但它们的表达方式还不足以保证下一步的学习和实验。为了弥补这一点，我们给出msgtool，这是一个操作IPC消息队列的交互式命令行工具。虽然它实际上经常作为一个工具用于教学目的，但是，通过利用标准 shell脚本提供消息队列功能，该工具也可以直接应用在实际的赋值操作中。

9. 背景知识

msgtool依赖于命令行变元来确定自己的行为。当从 shell脚本调用时，这个特征使它显得尤其有用。该工具提供了所有的功能，从创建、发送和接收，到改变许可权限以及最后删除队列。当前，它使用字符数组来作数据，允许用户发送文本消息。当然也可以改变这一点，使它可以发送其他类型数据的工作。我们当作一个练习把它留给读者去完成。

10. 命令行语法

发送消息

```
msgtool s (type) "text"
```

获取消息

```
msgtool r (type)
```

改变许可权限(模式)

```
msgtool m (mode)
```

删除队列

```
msgtool d
```

11. 实例

```
msgtool  s   1 test
msgtool  s   5 test
msgtool  s   1 "This is a test"
msgtool  r   1
msgtool  d
msgtool  m   660
```

12. 源代码

下面就是msgtool工具的源代码。应该在支持系统V IPC的最新内核版本上编译这段代码。在作重建时一定要在内核中使能系统V IPC。

顺便提一句，不论请求执行哪种类型的动作，这个工具在消息队列不存在时都会把它创建出来。

因为这个工具使用ftok ( )函数来生成IPC关键字的值。用户可能会遇到目录冲突的问题。如果在脚本的任意位置改变目录，它可能不会工作。另一种解决方案是把一个更复杂的路径(如“ /tmp/msgtool ”)硬编码进msgtool中，或者甚至把路径和其他可操作的变元一起通过命令行传送。

```
/*************************************************************************
 Excerpt from "Linux Programmer's Guide - Chapter 6"
 (C)opyright 1994-1995, Scott Burkett
 *************************************************************************
 MODULE: msgtool.c
 *************************************************************************
 A command line tool for tinkering with SysV style Message Queues
 *************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define MAX_SEND_SIZE 80

struct mymsgbuf {
        long mtype;
        char mtext[MAX_SEND_SIZE];


void send_message(int qid, struct mymsgbuf *qbuf, long type, char *text);
void read_message(int qid, struct mymsgbuf *qbuf, long type);
void remove_queue(int qid);
void change_queue_mode(int qid, char *mode);
void usage(void);
    int main(int argc, char *argv[])
    {
```

```
        key_t key;
        int   msgqueue_id;
        struct mymsgbuf qbuf;

        if(argc == 1)
                usage();

        /* Create unique key via call to ftok() */
        key = ftok(".", 'm');

        /* Open the queue - create if necessary */
        if((msgqueue_id = msgget(key, IPC_CREAT|0660)) == -1) {
                perror("msgget");
                exit(1);
        }

        switch(tolower(argv[1][0]))
        {
                case 's': send_message(msgqueue_id, (struct mymsgbuf *)&qbuf,
                                        atol(argv[2]), argv[3]);
                          break;
                case 'r': read_message(msgqueue_id, &qbuf, atol(argv[2]));
                          break;
                case 'd': remove_queue(msgqueue_id);
                          break;
                case 'm': change_queue_mode(msgqueue_id, argv[2]);
                          break;

                 default: usage();

        }

        return(0);
}

void send_message(int qid, struct mymsgbuf *qbuf, long type, char *text)
{
        /* Send a message to the queue */
        printf("Sending a message ...\n");
        qbuf->mtype = type;
        strcpy(qbuf->mtext, text);

        if((msgsnd(qid, (struct msgbuf *)qbuf,
                strlen(qbuf->mtext)+1, 0)) ==-1)
        {
                perror("msgsnd");
                exit(1);
        }
}

void read_message(int qid, struct mymsgbuf *qbuf, long type)
```

```
{
        /* Read a message from the queue */
        printf("Reading a message ...\n");
        qbuf->mtype = type;
        msgrcv(qid, (struct msgbuf *)qbuf, MAX_SEND_SIZE, type, 0);

        printf("Type: %ld Text: %s\n", qbuf->mtype, qbuf->mtext);
}

void remove_queue(int qid)
{
        /* Remove the queue */
        msgctl(qid, IPC_RMID, 0);
}

void change_queue_mode(int qid, char *mode)
{
        struct msqid_ds myqueue_ds;

        /* Get current info */
        msgctl(qid, IPC_STAT, &myqueue_ds);

        /* Convert and load the mode */
        sscanf(mode, "%ho", &myqueue_ds.msg_perm.mode);

        /* Update the mode */
        msgctl(qid, IPC_SET, &myqueue_ds);
}

void usage(void)
{
        fprintf(stderr, "msgtool - A utility for tinkering with msg queues\n");
        fprintf(stderr, "\nUSAGE: msgtool (s)end <type> <messagetext>\n");
        fprintf(stderr, "               (r)ecv <type>\n");
        fprintf(stderr, "               (d)elete\n");
        fprintf(stderr, "               (m)ode <octal mode>\n");
        exit(1);
}
```

6.4.3 信号量

1. 基本概念

信号量的最佳定义是：它是一种计数器，用来控制对多个进程共享的资源所进行的访问。它们常常被用作一个锁机制，在某个进程正在对特定资源进行操作时，信号量可以防止另一个进程去访问它。信号量常常被认为是系统 V的这三类IPC对象中最难掌握的一种。为了充分地理解信号量，在介绍任何系统调用和操作原理以前，将会先简要地讨论一下。

信号量这个名称实际上是个古老的铁路方面的术语，指的是在交叉路口的“ 长臂 ”，它把汽车挡住，防止它们在有火车经过时横穿铁轨。这实际上也可以说是一个简单的信号量集合。如果信号量打开(长臂向上举着)，那么资源是可以使用的(汽车可以横穿铁轨)。然而，如果信

号量关闭(长臂已经放下来了)，那么资源就不能使用(汽车必须等待)。

虽然可以用这个简单的例子来引入信号量的概念，但读者应该认识到信号量实际上是以集合的形式实现的，而不是作为单个实体实现的，认识这点很重要。当然，某个特定的信号量集合中可能只有一个信号量，就象在铁路的例子中一样。

要理解信号量这个概念，另一个方法可能就是把它们当作资源计数器。让我们把这个概念应用到另一种实际情况上。假设有一个打印假脱机程序，它可以处理多个打印机，而每个打印机需要处理多个打印请求。假脱机打印管理员将会使用信号量集合来监测对每个打印机的访问。

假设在我们团体的打印房间中有五个打印机在线。假脱机打印管理员分配一个信号量集合，它包括五个信号量，每个信号量对应系统上的一个打印机。因为在物理上每个打印机每次只能打印一个任务，在集合中的这五个信号量每个都将初始化为值等于 1，这意味着它们都在线，并且可以接受请求。

John向假脱机程序发送一个打印请求。打印管理员查看信号量集合，找到第一个值为 1的信号量。在把John的打印请求发送给物理设备之前，打印管理员使用一个为－ 1的值，使对应打印机的信号量减一。这样一来，那个信号量的值现在为 0。在系统V的信号量中，值为0就代表在那个信号量上的资源被百分之百地利用了。在我们的例子中，不能再把其他的请求发送给那个打印机了，直到信号量的值不再等于 0。

在John的打印任务完成之后，打印管理员将使对应打印机的信号量的值增加一。它的值现在又回到了1。这意味着那个打印机又可以使用了。一般来说，如果所有的 5个信号量值都为0，则表示它们都在忙着完成打印请求，即当前没有打印机可用。

尽管这是个简单的例子，但是请不要被赋予给集合中每个信号量的初值 1所迷惑了。信号量尽管被当成是资源计数器，它还可以被初始化为任何一个正的整数值，并不局限于或者为 1或者为0。如果这5个打印机每个可以同时处理 10个打印任务，用户就可以把每个信号量都初始化为10，每到达一个新任务则减去一，每完成一个打印任务则增加一。在下一章中读者将发现，信号量与共享内存段的工作关系非常紧密。共享内存段的任务就像一条看门狗，它可以防止多个进程同时写同一个内存段。

在介绍相关的系统调用之前，我们简单地了解一下在信号量操作中用到的各种内部数据结构。

2. 内部数据结构

让我们简单地了解一下内核为信号量集合所维护的数据结构。

3. 内核semid_ds结构

和消息队列一样，内核也为存在于它的寻址空间中的每个信号量集合维护一个特殊的内部数据结构。这个结构的类型是semid_ds，该类型的定义在linux/sem.h中，如下所示：

```
        /* One semid data structure for each set of semaphores in the system.
*/
        struct semid_ds {
                struct ipc_perm sem_perm;       /* permissions .. see ipc.h */
                time_t          sem_otime;      /* last semop time */
                time_t          sem_ctime;      /* last change time */
                struct sem      *sem_base;      /* ptr to first semaphore in
array */
```

```
                struct wait_queue *eventn;
                struct wait_queue *eventz;
                struct sem_undo  *undo;          /* undo requests on this array
*/
                ushort           sem_nsems;      /* no. of semaphores in array
*/
        };
```

与消息队列一样，这个结构上的操作是由某个特殊的系统调用执行的，并且不能直接对其作任何操作。下面是最重要的字段的描述：

sem_perm

这是ipc_perm结构的一个实例。它是在 linux/ipc.h中定义的，它里面存放的是信号量集合的许可权限信息，包括访问许可权限，以及关于集合的创建者的有关信息 (uid等等)。

sem_otime

最后一个semop ( )操作的时间(稍后再作详细介绍)。

sem_ctime

最后一次修改这个结构(改变方式等等)的时间。

sem_base

这是一个指针，指向数组中第一个信号量 (参见下一个结构)。

sem_undo

在这个数组中的undo请求的次数(稍后再作详细介绍)。

sem_nsems

在该信号量集合(数组)中信号量的数目。

4. 内核sem结构

在semid_ds结构中，存在一个指针，指向信号量数组的基址。每个数组成员都是 sem类型的。Sem结构也是在linux/sem.h中定义的，如下所示：

```
        /* One semaphore structure for each semaphore in the system. */
        struct sem {
                short   sempid;         /* pid of last operation */
                ushort  semval;         /* current value */
                ushort  semncnt;        /* num procs awaiting increase in
semval */
                ushort  semzcnt;        /* num procs awaiting semval = 0 */
        };
```

sem_pid

执行最后一次操作的PID(进程ID号)。

sem_semval

信号量的当前值。

sem_semncnt

等待资源变为可用状态的进程数目。

sem_semzcnt

等待资源利用率达到百分之百的进程的数目。

5. 系统调用：semget ( )

为了创建一个新的信号量集合，或者访问现有的集合，可以使用系统调用 semget ( )。

```
SYSTEM CALL: semget();

PROTOTYPE: int semget ( key_t key, int nsems, int semflg );
  RETURNS: semaphore set IPC identifier on success
           -1 on error: errno = EACCESS (permission denied)
                                EEXIST (set exists, cannot create (IPC_EXCL))
                                EIDRM (set is marked for deletion)
                                ENOENT (set does not exist, no IPC_CREAT was
used)
                                ENOMEM (Not enough memory to create new set)
                                ENOSPC (Maximum set limit exceeded)

NOTES:
```

semget ( )系统调用的第一个变元是关键字的值 (在我们的例子中，该值是调用 ftok ( )的返回值)。然后再把该关键字值与内核中现有的其他信号量集合的关键字值都比较。在比较之后，打开或者访问操作依赖于semflg变元的内容。

IPC_CREAT

如果内核中不存在这样的信号量集合，则把它创建出来。

IPC_EXCL

当与IPC_CREAT一起使用时，如果信号量集合早已存在，则操作将失败。

如果单独使用IPC_CREAT，semget ( )或者返回新创建的信号量集合的信号量集合标识符，或者返回早已存在的具有同一个关键字值的集合的标识符。如果同时使用 IPC_EXCL和IPC_CREAT，那么将有两种可能的结果：如果集合不存在，则创建一个新的集合；如果集合早已存在，则调用失败，并返回－1。IPC_EXCL本身是没有什么用处的，但当与 IPC_CREAT组合使用时，它可以用于保证没有为了访问而打开现有的信号量集合。

与系统V IPC的其他种类一样，可以把信号量集合的许可权限与掩码进行 OR操作，得到一个可选的八进制形式的许可模式。

nsems变元可以指定在新的集合中应该创建的信号量的数目。在前面所介绍的虚构的打印房间的例子中，这个值代表打印机的数目。在集合中最多能容纳的信号量的数目是在linux/sem.h中定义的，如下所示：

```
#define SEMMSL  32      /* <=512 max num of semaphores per id */
```

注意，如果用户显式地打开某个现有的集合，则 nsems变元将被忽略。

下面创建一个包装程序函数，用来打开或创建信号量集合：

```
int open_semaphore_set( key_t keyval, int numsems )
{
        int     sid;

        if ( ! numsems )
                return(-1);

        if((sid = semget( mykey, numsems, IPC_CREAT | 0660 )) == -1)
        {
                return(-1);
        }
```

```
        return(sid);
}
```

请注意函数中使用了显式的许可权限 660，这个小函数要么返回一个信号量集合标识符(整型)，要么出错并返回－1，必须把关键字值传送给它。如果是创建新集合的话，还需要传递信号量的数目，以便给它们分配空间。在本章最后所给出的例子中，读者可以注意到使用了IPC_EXCL标志，这是为了判断该信号量集合是否存在。

6. 系统调用：semop ( )

```
  SYSTEM CALL: semop();
  PROTOTYPE: int semop ( int semid, struct sembuf *sops, unsigned nsops);
    RETURNS: 0 on success (all operations performed)
             -1 on error: errno = E2BIG (nsops greater than max number of ops
allowed atomically)
                                  EACCESS (permission denied)
                                  EAGAIN (IPC_NOWAIT asserted, operation could
not go through)
                                  EFAULT (invalid address pointed to by sops
argument)
                                  EIDRM (semaphore set was removed)
                                  EINTR (Signal received while sleeping)
                                  EINVAL (set doesn't exist, or semid is
invalid)
                                  ENOMEM (SEM_UNDO asserted, not enough memory
to create the
                                          undo structure necessary)
                                  ERANGE (semaphore value out of range)
    NOTES:
```

semop ( )的第一个变元是关键字的值(在我们的例子中，它是调用semget所获得的返回值)。第二个变元(sops)是一个指针，指向将要在信号量集合上执行的操作的一个数组，而第三个变元(nsops)则是该数组中操作的个数。

sops变元指向的是类型为sembuf的结构的一个数组。sembuf结构是在linux/sem.h中定义的，如下所示：

```
        /* semop system call takes an array of these */
        struct sembuf {
                ushort  sem_num;        /* semaphore index in array */
                short   sem_op;         /* semaphore operation */
                short   sem_flg;        /* operation flags */
        };
```

sem_num

用户希望处理的信号量的编号。

sem_op

将要执行的操作(正、负，或者零)。

sem_flg

可操作的标志。

如果sem_op为负，则它的值需要从信号量中减掉。这对应着获得信号量所控制的或者监

视器所访问的资源。如果没有指定 IPC_NOWAIT，那么调用进程将睡眠，直到在信号量中已经有了能够满足需求的数量的资源(另一个进程释放了一些资源)。

如果sem_op为正，则它的值将被加到信号量上。这对应着把资源归还给应用程序的信号量集合。当不再需要资源时，应该记得把它们归还给信号量集合！

最后，如果sem_op为零(0)，则调用进程将睡眠，直到信号量的值为零为止。这对应于等待信号量的利用率达到百分之百。关于这点有一个很好的例子。如一个以超级用户权限运行的守护进程，当信号量集合达到完全的利用率时，它可以动态地调整信号量集合的大小。

为了解释semop调用，读者可以回想一下以前介绍过的有关打印房间的例子。假设我们只有一台打印机，每次只能处理一个任务。在创建信号量集合时，应该让它只含一个信号量 (只有一台打印机)，并且把那个信号量初始化为值等于 1(每次只能处理一个任务)。

每次用户需要向这台打印机发送一个任务时，首先需要确定资源是可用的。为了做到这一点，用户可以尝试着从信号量获得一个单元的资源。可以通过装入一个 sembuf数组来执行这个操作：

```
struct sembuf sem_lock = { 0, -1, IPC_NOWAIT };
```

我们解释一下上面的这个初始化结构的含义，它意味着将在信号量集合的 0号信号量上加上一个值－1。换句话说，将从集合中唯一的信号量 (第0号)获得一个单元的资源。在这里指定了IPC_NOWAIT，所以这次调用或者将立刻被满足，或者如果另一个打印任务正在打印，则调用出错。下面是在semop系统调用中使用初始化sembuf结构的一个例子：

```
if((semop(sid, &sem_lock, 1) == -1)
        perror("semop");
```

第三个变元(nsops)显示我们只执行一个操作 (在操作数组中只有一个 sembuf结构)。变元sid是信号量集合的IPC标识符。

在完成了打印任务之后，我们必须把资源归还给信号量集合，这样其他人就可以使用打印机了。

```
struct sembuf sem_unlock = { 0, 1, IPC_NOWAIT };
```

上述这个初始化结构的意义是：信号量集合中的第 0号信号量将被增加一个为 1的值。换句话说，有一个单元的资源被归还给集合。

7. 系统调用：semctl ( )

```
SYSTEM CALL: semctl();
PROTOTYPE: int semctl ( int semid, int semnum, int cmd, union semun arg );
  RETURNS: positive integer on success
           -1 on error: errno = EACCESS (permission denied)
                                EFAULT (invalid address pointed to by arg
argument)
                                EIDRM (semaphore set was removed)
                                EINVAL (set doesn't exist, or semid is
invalid)
                                EPERM (EUID has no privileges for cmd in arg)
                                ERANGE (semaphore value out of range)
NOTES: Performs control operations on a semaphore set
```

系统调用 semctl用于在信号量集合上执行控制操作。这个调用类似于系统调用 msgctl，msgctl是用于消息队列上的操作。如果读者比较一下两个系统操作的变元列表，将会发现

semctl和msgctl的列表之间差别非常小。我们知道信号量实际上是以集合的形式实现的，而不是以单个实体的形式实现的。对于信号量操作而言，不光需要传递 IPC关键字，还需要传递集合中的目标信号量。

这两个系统调用都使用到一个 cmd变元，用于指定在IPC对象上执行的命令。剩下的区别是两个调用的最后一个变元。在msgctl中，最后一个变元代表的是内核所使用的内部数据结构的一个拷贝。我们使用该结构来获取有关消息队列的内部信息，以及设置和改变队列的许可权限和拥有权。对信号量而言，它还支持其他的可操作命令，这样就需要一个更复杂的数据类型来作最后一个变元。对许多初学信号量的编程人员来说，联合体的使用是一个相当困惑的难题。我们将认真地对这个结构进行详细研究，以尽可能避免读者的困惑。

semctl ( )的第一个变元是关键字的值(在我们的例子中它是调用semget所返回的值)。第二个变元(semun)是将要执行操作的信号量的编号。大致而言，它可以看成是信号量集合的一个索引值，对于集合中的第一个信号量 (有可能只有这一个信号量)来说，它的索引值将是一个为零的值。

cmd变元代表将要在集合上执行的命令。正如读者将会看到的，这里包括我们熟悉的IPC_STAT/IPC_SET命令，以及其他一些信号量集合所特有的命令：

IPC_STAT

获取某个集合的semid_ds结构，并把它存储在semun联合体的buf变元所指定的地址中。

IPC_SET

设置某个集合的semid_ds结构的ipc_perm成员的值。该命令所取的值是从 semun联合体的buf变元中取到的。

IPC_RMID

从内核删除该集合。

GETALL

用于获取集合中所有信号量的值。整数值存放在无符号短整数的一个数组中，该数组由联合体的array成员所指定。

GETNCNT

返回当前正在等待资源的进程的数目。

GETPID

返回最后一次执行semop调用的进程的PID。

GETVAL

返回集合中某个信号量的值。

GETZCNT

返回正在等待资源利用率达到百分之百的进程的数目。

SETALL

把集合中所有信号量的值设置为联合体的 array成员所包含的对应值。

SETVAL

把集合中单个信号量的值设置为联合体的 val成员的值。

变元arg代表类型semun的一个实例。这个特殊的联合体是在 linux/sem.h中定义的，如下所示：

```
/* arg for semctl system calls. */
union semun {
        int val;                /* value for SETVAL */
        struct semid_ds *buf;   /* buffer for IPC_STAT & IPC_SET */
        ushort *array;          /* array for GETALL & SETALL */
        struct seminfo *__buf;  /* buffer for IPC_INFO */
        void *__pad;
};
```

val

当执行SETVAL命令时将用到这个成员，它用于指定要把信号量设置成什么值。

buf

在命令IPC_STAT/IPC_SET中使用。它代表内核中所使用的内部信号量数据结构的一个拷贝。

array

用在GETALL/SETALL命令中的一个指针。它应当指向整数值的一个数组。在设置或获取集合中所有信号量的值的过程中，将会用到该数组。

剩下的变元_buf和_pad将在内核中的信号量代码中内部使用，对于应用程序开发人员来说，它们用处很少，或者说没有用处。一个明显的事实是，这两个变元是 Linux操作系统所特有的，在其他的Unix实现中没有。

因为这个特殊的系统调用被认为是所有系统 V IPC调用中最难掌握的，我们将列出它的多个使用实例。

下面的这个代码片段返回作为参数传递给它的信号量的值。在使用 GETVAL命令时，最后一个变元(联合体)将被忽略：

```
int get_sem_val( int sid, int semnum )
{
        return( semctl(sid, semnum, GETVAL, 0));
}
```

我们再看一下打印机的那个例子，假设所有五个打印机的状态都需要了解：

```
#define MAX_PRINTERS 5

printer_usage()
{
        int x;

        for(x=0; x<MAX_PRINTERS; x++)
                printf("Printer %d: %d\n\r", x, get_sem_val( sid, x ));
}
```

请再看下面这个函数，它可以用来初始化一个新的信号量的值：

```
void init_semaphore( int sid, int semnum, int initval)
{
        union semun semopts;

        semopts.val = initval;
```

```
        semctl( sid, semnum, SETVAL, semopts);
}
```

注意，semctl的最后一个变元是联合体的一个拷贝，而不是指向它的一个指针。虽然目前正在介绍的内容是把这个联合体当作变元使用的，但还是请允许我介绍一个在使用这个系统调用的时候相当普遍的一个错误。

在msgtool的实例中，我们用IPC_STAT和IPC_SET命令来改变队列的许可权限。虽然信号量也支持这些命令，它们的用法稍有不同。这是因为内部数据结构是从联合体的一个成员获取和拷贝的，而不是从单个实体获得的。读者能找出下面这段程序中的错误吗？

```
/* Required permissions should be passed in as text (ex: "660") */

void changemode(int sid, char *mode)
{
        int rc;
        struct semid_ds mysemds;

        /* Get current values for internal data structure */
        if((rc = semctl(sid, 0, IPC_STAT, semopts)) == -1)
        {
                perror("semctl");
                exit(1);
        }
        printf("Old permissions were %o\n", semopts.buf->sem_perm.mode);

        /* Change the permissions on the semaphore */
        sscanf(mode, "%o", &semopts.buf->sem_perm.mode);

        /* Update the internal data structure */
        semctl(sid, 0, IPC_SET, semopts);

        printf("Updated...\n");
```

该段代码试图为集合所使用的内部数据结构制作一个拷贝，修改许可权限，并用IPC_SET命令把它们重置回内核。然而，第一次调用 semctl将立刻返回EFAULT。或者返回最后一个变元(联合体！)的地址。此外，如果我们不去查看那次调用的错误，我们将会导致内存出错。读者知道为什么吗？

我们知道 IPC_SET/IPC_STAT命令使用的是联合体的 buf成员，它是一个指向类型semid_ds的指针。指针就是指针，不是别的什么！ buf成员必须指向一些合法的存储位置，这样我们的函数才能正常工作。请看下面这个修改过的版本：

```
void changemode(int sid, char *mode)
{
        int rc;
        struct semid_ds mysemds;

        /* Get current values for internal data structure */
```

```
/* Point to our local copy first! */
semopts.buf = &mysemds;

/* Let's try this again! */
if((rc = semctl(sid, 0, IPC_STAT, semopts)) == -1)
{
        perror("semctl");
        exit(1);
}

printf("Old permissions were %o\n", semopts.buf->sem_perm.mode);

/* Change the permissions on the semaphore */
sscanf(mode, "%o", &semopts.buf->sem_perm.mode);

/* Update the internal data structure */
semctl(sid, 0, IPC_SET, semopts);
        printf("Updated...\n");
}
```

6.4.4 semtool：交互式信号量操作程序

1. 背景知识

semtool程序依赖于命令行变元来决定自己的行为。当从 shell脚本调用它时，这个特征使它显得尤其有用。 semtool提供了所有的功能，从创建和操作，到修改许可权限以及最后删除信号量集合，它可以用于通过标准 shell脚本来控制共享资源。

2. 命令行语法

创建信号量集合

semtool c(集合中信号量的个数)

锁住某个信号量

semtool l(要锁住的信号量的编号)

解锁某个信号量

semtool u(要解锁的信号量的编号)

改变许可权限(模式)

semtool m (mode)

删除信号量集合

semtool d

3. 实例

```
semtool  c 5
semtool  l
semtool  u
semtool  m 660
semtool  d
```

4. 源代码

```
/*************************************************************************
 Excerpt from "Linux Programmer's Guide - Chapter 6"

 (C)opyright 1994-1995, Scott Burkett
 *************************************************************************
 MODULE: semtool.c
 *************************************************************************
 A command line tool for tinkering with SysV style Semaphore Sets

 ************************************************************************/

#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#define SEM_RESOURCE_MAX        1       /* Initial value of all semaphores */

void opensem(int *sid, key_t key);
void createsem(int *sid, key_t key, int members);
void locksem(int sid, int member);
void unlocksem(int sid, int member);
void removesem(int sid);
unsigned short get_member_count(int sid);
int getval(int sid, int member);
void dispval(int sid, int member);
void changemode(int sid, char *mode);
void usage(void);

int main(int argc, char *argv[])
{
        key_t key;
        int   semset_id;

        if(argc == 1)
                usage();

        /* Create unique key via call to ftok() */
        key = ftok(".", 's');

        switch(tolower(argv[1][0]))
        {
                case 'c': if(argc != 3)
                                usage();
                          createsem(&semset_id, key,  atoi(argv[2]));
                          break;
                case 'l': if(argc != 3)
                                usage();
                          opensem(&semset_id, key);
```

```
                        locksem(semset_id, atoi(argv[2]));
                        break;
                case 'u': if(argc != 3)
                                usage();
                        opensem(&semset_id, key);
                        unlocksem(semset_id, atoi(argv[2]));
                        break;
                case 'd': opensem(&semset_id, key);
                        removesem(semset_id);
                        break;
                case 'm': opensem(&semset_id, key);
                        changemode(semset_id, argv[2]);
                        break;
                 default: usage();

        }

        return(0);
}

void opensem(int *sid, key_t key)
{
        /* Open the semaphore set - do not create! */

        if((*sid = semget(key, 0, 0666)) == -1)
        {
                printf("Semaphore set does not exist!\n");
                exit(1);
        }

}

void createsem(int *sid, key_t key, int members)
{
        int cntr;
        union semun semopts;

        if(members > SEMMSL) {
                printf("Sorry, max number of semaphores in a set is %d\n",
                        SEMMSL);
                exit(1);
        }

        printf("Attempting to create new semaphore set with %d members\n",
                                members);
        if((*sid = semget(key, members, IPC_CREAT|IPC_EXCL|0666))
                        == -1)
        {
                fprintf(stderr, "Semaphore set already exists!\n");
                exit(1);
```

```
        }

        semopts.val = SEM_RESOURCE_MAX;

        /* Initialize all members (could be done with SETALL) */
        for(cntr=0; cntr<members; cntr++)
                semctl(*sid, cntr, SETVAL, semopts);
}

void locksem(int sid, int member)
{
        struct sembuf sem_lock={ 0, -1, IPC_NOWAIT};

        if( member<0 || member>(get_member_count(sid)-1))
        {
                fprintf(stderr, "semaphore member %d out of range\n", member);
                return;
        }

        /* Attempt to lock the semaphore set */
        if(!getval(sid, member))
        {
                fprintf(stderr, "Semaphore resources exhausted (no lock)!\n");
                exit(1);
        }

        sem_lock.sem_num = member;

        if((semop(sid, &sem_lock, 1)) == -1)
        {
                fprintf(stderr, "Lock failed\n");
                exit(1);
        }
        else
                printf("Semaphore resources decremented by one (locked)\n");

        dispval(sid, member);
}

void unlocksem(int sid, int member)
{
 struct sembuf sem_unlock={ member, 1, IPC_NOWAIT};
 int semval;

 if( member<0 || member>(get_member_count(sid)-1))
 {
        fprintf(stderr, "semaphore member %d out of range\n", member);
        return;
 }

 /* Is the semaphore set locked? */
```

```
        semval = getval(sid, member);
        if(semval == SEM_RESOURCE_MAX) {
                fprintf(stderr, "Semaphore not locked!\n");
                exit(1);
        }

        sem_unlock.sem_num = member;

        /* Attempt to lock the semaphore set */
        if((semop(sid, &sem_unlock, 1)) == -1)
        {
                fprintf(stderr, "Unlock failed\n");
                exit(1);
        }
        else
                printf("Semaphore resources incremented by one (unlocked)\n");

        dispval(sid, member);
}

void removesem(int sid)
{
        semctl(sid, 0, IPC_RMID, 0);
        printf("Semaphore removed\n");
}

unsigned short get_member_count(int sid)
{
        union semun semopts;
        struct semid_ds mysemds;

        semopts.buf = &mysemds;

        /* Return number of members in the semaphore set */
        return(semopts.buf->sem_nsems);
}
int getval(int sid, int member)
{
        int semval;

        semval = semctl(sid, member, GETVAL, 0);
        return(semval);
}

void changemode(int sid, char *mode)
{
        int rc;
        union semun semopts;
        struct semid_ds mysemds;

        /* Get current values for internal data structure */
```

```
        semopts.buf = &mysemds;

        rc = semctl(sid, 0, IPC_STAT, semopts);

        if (rc == -1) {
                perror("semctl");
                exit(1);
        }

        printf("Old permissions were %o\n", semopts.buf->sem_perm.mode);

        /* Change the permissions on the semaphore */
        sscanf(mode, "%ho", &semopts.buf->sem_perm.mode);

        /* Update the internal data structure */
        semctl(sid, 0, IPC_SET, semopts);

        printf("Updated...\n");

}

void dispval(int sid, int member)
{
        int semval;

        semval = semctl(sid, member, GETVAL, 0);
        printf("semval for member %d is %d\n", member, semval);
}

void usage(void)
{
        fprintf(stderr, "semtool - A utility for tinkering with semaphores\n");
        fprintf(stderr, "\nUSAGE:  semtool4 (c)reate <semcount>\n");
        fprintf(stderr, "                (l)ock <sem #>\n");
        fprintf(stderr, "                (u)nlock <sem #>\n");
        fprintf(stderr, "                (d)elete\n");
        fprintf(stderr, "                (m)ode <mode>\n");
        exit(1);
}
```

5. semstat：semtool的伴随程序

作为一个附赠品，下面给出一个称为 semstat的伴随程序的源代码。 semstat程序显示出集合中每个信号量的值，该集合是由 semtool所创建的。

```
/*************************************************************************
 Excerpt from "Linux Programmer's Guide - Chapter 6"
 (C)opyright 1994-1995, Scott Burkett
 *************************************************************************
 MODULE: semstat.c
 *************************************************************************
 A companion command line tool for the semtool package.  semstat displays
```

```
 the current value of all semaphores in the set created by semtool.
 ****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int get_sem_count(int sid);
void show_sem_usage(int sid);
int get_sem_count(int sid);
void dispval(int sid);

int main(int argc, char *argv[])
{
        key_t key;
        int   semset_id;

        /* Create unique key via call to ftok() */
        key = ftok(".", 's');

        /* Open the semaphore set - do not create! */
        if((semset_id = semget(key, 1, 0666)) == -1)
        {
                printf("Semaphore set does not exist\n");
                exit(1);
        }

        show_sem_usage(semset_id);
        return(0);
}

void show_sem_usage(int sid)
{
        int cntr=0, maxsems, semval;

        maxsems = get_sem_count(sid);

        while(cntr < maxsems) {
                semval = semctl(sid, cntr, GETVAL, 0);
                printf("Semaphore #%d:  --> %d\n", cntr, semval);
                cntr++;
        }
}

int get_sem_count(int sid)
{
        int rc;
        struct semid_ds mysemds;
```

```
        union semun semopts;

        /* Get current values for internal data structure */
        semopts.buf = &mysemds;

        if((rc = semctl(sid, 0, IPC_STAT, semopts)) == -1) {
                perror("semctl");
                exit(1);
        }

        /* return number of semaphores in set */
        return(semopts.buf->sem_nsems);
}

void dispval(int sid)
{
        int semval;

        semval = semctl(sid, 0, GETVAL, 0);
        printf("semval is %d\n", semval);
}
```

6.4.5 共享内存

1. 基本概念

共享内存的最佳定义是：它是对将要在多个进程间映射和共享的内存区域 (段)所作的映射。这是IPC最快捷的方式，因为这里没有什么中介 (例如管道、消息队列等等)。相反，消息直接从某内存段映射，并映射到调用进程的寻址空间。内存段可被某进程创建，并接着写到任意数量的进程中去，或者是从任意数量的进程中读数据。

2. 内部和用户数据结构

让我们简单地了解一下内核为共享内存段所维护的数据结构。

3. 内核shmid_ds结构

和消息队列以及信号量集合一样，对于当前存在于内核的寻址空间中的每个共享内存段，内核均为其维护着一个特殊的内部数据结构。这个结构的类型是 shmid_ds，它是在linux/shm.h中定义的，如下所示：

```
        /* One shmid data structure for each shared memory segment in the
system. */
        struct shmid_ds {
                struct ipc_perm shm_perm;       /* operation perms */
                int     shm_segsz;              /* size of segment (bytes) */
                time_t  shm_atime;              /* last attach time */
                time_t  shm_dtime;              /* last detach time */
                time_t  shm_ctime;              /* last change time */
                unsigned short  shm_cpid;       /* pid of creator */
                unsigned short  shm_lpid;       /* pid of last operator */
                short   shm_nattch;             /* no. of current attaches */
```

```
                                                    /* the following are private
    */

                    unsigned short   shm_npages;     /* size of segment (pages) */
                    unsigned long   *shm_pages;      /* array of ptrs to frames ->
    SHMMAX */
                    struct vm_area_struct *attaches; /* descriptors for attaches */
            };
```

在这个结构上所进行的操作是由一个特殊的系统调用执行的，并且还不能直接对它进行操作，下面是对一些更重要的域的描述：

shm_perm

它是ipc_perm结构的一个实例，它是在 linux/ipc.h中定义的，该域存放着该内存段的许可权限信息，包括访问许可权限，以及有关内存段的创建者的信息 (uid等等)。

shm_segsz

内存段的大小(以字节为单位测量)。

shm_atime

最后一次进程连接到该内存段的时间。

shm_dtime

最后一个进程与该内存段断开连接的时间。

shm_ctime

对这个内存段进行最后一次修改 (改变模式等等)的时间。

shm_cpid

创建进程的PID。

shm_lpid

对该内存段进行最后一次操作的进程的 PID。

shm_nattch

当前与该内存段相连接的进程的数量。

4. 系统调用：shmget ( )

为了创建一个新的共享内存段，或者访问一个现有的共享内存段，可以使用 shmget ( )系统调用。

```
  SYSTEM CALL: shmget();

  PROTOTYPE: int shmget ( key_t key, int size, int shmflg );
    RETURNS: shared memory segment identifier on success
             -1 on error: errno = EINVAL (Invalid segment size specified)
                                  EEXIST (Segment exists, cannot create)
                                  EIDRM (Segment is marked for deletion, or was
removed)
                                  ENOENT (Segment does not exist)
                                  EACCES (Permission denied)
                                  ENOMEM (Not enough memory to create segment)

  NOTES:
```

这个系统调用对我们来说早已不陌生了，它与消息队列以及信号量集合对应的调用惊人

的相似。

shmget的第一个变元是关键字的值(在我们的例子中它是由调用 ftok ( )所返回的值)。然后，这个值将与内核中现有的其他共享内存段的关键字值相比较。在比较之后，打开和访问操作都将依赖于shmflg变元的内容。

IPC_CREAT

如果在内核中不存在该内存段，则创建它。

IPC_EXCL

当与IPC_CREAT一起使用时，如果该内存段早已存在，则此次调用将失败。

如果只使用IPC_CREAT，shmget( )或者将返回新创建的内存段的段标识符，或者返回早已存在于内核中的具有相同关键字值的内存段的标识符。如果同时使用 IPC_CREAT和IPC_EXCL，则可能会有两种结果，如果该内存段不存在，则将创建一个新的内存段；如果内存段早已存在，则此次调用失败，并将返回－ 1。IPC_EXCL本身是没有什么用处的，但在与IPC_CREAT组合使用时，它可用于保证没有一个现有的内存段为了访问而打开着。

再说一遍，用户可以把许可方式与掩码进行 OR操作，得到一个可选的八进制形式的许可模式。

下面我们创建一个包装程序函数，用于查找或者创建一个共享内存段：

```
int open_segment( key_t keyval, int segsize )
{
        int     shmid;

        if((shmid = shmget( keyval, segsize, IPC_CREAT | 0660 )) == -1)
        {
                return(-1);
        }

        return(shmid);
}
```

注意，程序中使用了显式的许可权限 0660。这个小函数或者将返回一个共享内存段标识符(整型)，或者在出错时返回－ 1。关键字的值和所申请的段的大小 (以字节为单位)都是作为变元而传递给该函数的。

一旦进程获得了给定内存段的合法 IPC标识符，它的下一步操作就是连接该内存段，或者把该内存段映射到自己的寻址空间中。

5. 系统调用：shmat ( )

```
  SYSTEM CALL: shmat();

  PROTOTYPE: int shmat ( int shmid, char *shmaddr, int shmflg);
    RETURNS: address at which segment was attached to the process, or
             -1 on error: errno = EINVAL (Invalid IPC ID value or attach
address passed)
                                  ENOMEM (Not enough memory to attach segment)
                                EACCES (Permission denied)
 NOTES:
```

如果addr变元值等于零(0)，则内核将试着查找一个未映射的区域。这是我们推荐使用的方法。用户可以指定一个地址，但通常该地址只用于访问所拥有的硬件，或者解决与其他应用程序的冲突。SHM_RND标志可以与标志变元进行OR操作，结果再置为标志变元，这样可以让传送的地址页对齐(舍入到最相近的页面大小)。

此外，如果把SHM_RDONLY标志与标志变元进行OR操作，结果再置为标志变元，这样映射的共享内存段只能标记为只读方式。

这个调用使用起来可能是最简单的了。请看下面这个包装程序函数，它的一个参数是某内存段的合法IPC标识符，它将返回那个内存段所连接的地址：

```
char *attach_segment( int shmid )
{
        return(shmat(shmid, 0, 0));
}
```

一旦正确地连接上了一个内存段，并且进程拥有一个指向该内存段始地址的指针，则对那个内存段进行读和写就象引用和间接引用指针一样简单！请注意不要丢了原始指针的值！如果把它丢了，则用户将没有办法访问该内存段的基址 (开始)。

6. 系统调用：shmctl( )

```
  SYSTEM CALL: shmctl();
  PROTOTYPE: int shmctl ( int shmqid, int cmd, struct shmid_ds *buf );
    RETURNS: 0 on success
             -1 on error: errno = EACCES (No read permission and cmd is
IPC_STAT)
                                  EFAULT (Address pointed to by buf is invalid
with IPC_SET and
                                          IPC_STAT commands)
                                  EIDRM  (Segment was removed during retrieval)
                                  EINVAL (shmqid invalid)
                                  EPERM  (IPC_SET or IPC_RMID command was
issued, but
                                          calling process does not have write
(alter)
                                          access to the segment)
       NOTES:
```

这个调用与消息队列的msgctl调用是完全类似的。正因为如此，我们将不会过于详细地去介绍它，合法的命令值是：

IPC_STAT

获取内存段的shmid_ds结构，并把它存储在buf变元所指定的地址中。

IPC_SET

设置内存段shmid_ds结构的ipc_perm成员的值，此命令是从buf变元中获得该值的。

IPC_RMID

标记某内存段，以备删除。

IPC_RMID命令并不真正地把内存段从内存中删除。相反，它只是标记上该内存段，以备将来删除。只有当前连接到该内存段的最后一个进程正确地断开了与它的连接，实际的删除

操作才会发生。当然，如果当前没有进程与该内存段相连接，则删除将立刻发生。

为了正确地断开与其共享内存段的连接，进程需要调用 shmdt系统调用。

7. 系统调用：shmdt ( )

```
SYSTEM CALL: shmdt();

PROTOTYPE: int shmdt ( char *shmaddr );
  RETURNS: -1 on error: errno = EINVAL (Invalid attach address passed)
```

当某进程不再需要一个共享内存段时，它必须调用这个系统调用来断开与该内存段的连接。正如前面所介绍的那样，这与从内核删除内存段是两回事！在成功完成了断开连接操作以后，相关的shmid_ds结构的shm_nattch成员的值将减去一。如果这个值减到零 (0)，则内核将真正删除该内存段。

8. shmtool：交互式共享内存操作程序

背景知识

我们最后一个关于系统V IPC对象的例子将是shmtool，它是一个命令行工具，用于创建、读、写和删除共享内存段。再说一遍，与前面的例子一样，在任何操作中，如果内存段事先不存在，则它将被创建出来。

9. 命令行语法

向内存段写字符串

```
shmtool w "text"
```

从内存段获取字符串

```
shmtool r
```

改变许可权限(模式)

```
shmtool m (mode)
```

删除内存段

```
shmtool d
```

10. 实例

```
shmtool  w   test
shmtool  w   "This is a test"
shmtool  r
shmtool  d
shmtool  m   660
```

11. 源代码

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define SEGSIZE 100

main(int argc, char *argv[])
{
        key_t key;
```

```
    int   shmid, cntr;
    char  *segptr;

    if(argc == 1)
            usage();

    /* Create unique key via call to ftok() */
    key = ftok(".", 'S');

    /* Open the shared memory segment - create if necessary */
    if((shmid = shmget(key, SEGSIZE, IPC_CREAT|IPC_EXCL|0666)) == -1)
        {
                printf("Shared memory segment exists - opening as client\n");

                /* Segment probably already exists - try as a client */
                if((shmid = shmget(key, SEGSIZE, 0)) == -1)
                {
                        perror("shmget");
                        exit(1);
                }
        }
        else
        {
                printf("Creating new shared memory segment\n");
        }

        /* Attach (map) the shared memory segment into the current process */
        if((segptr = shmat(shmid, 0, 0)) == -1)
        {
                perror("shmat");
                exit(1);
        }

        switch(tolower(argv[1][0]))
        {
                case 'w': writeshm(shmid, segptr, argv[2]);
                          break;
                case 'r': readshm(shmid, segptr);
                          break;
                case 'd': removeshm(shmid);
                          break;
                case 'm': changemode(shmid, argv[2]);
                          break;
                 default: usage();

        }
}

writeshm(int shmid, char *segptr, char *text)
{
```

```
        strcpy(segptr, text);
        printf("Done...\n");
}

readshm(int shmid, char *segptr)
{
        printf("segptr: %s\n", segptr);
}

removeshm(int shmid)
{
        shmctl(shmid, IPC_RMID, 0);
        printf("Shared memory segment marked for deletion\n");
}

changemode(int shmid, char *mode)
{
        struct shmid_ds myshmds;

        /* Get current values for internal data structure */
        shmctl(shmid, IPC_STAT, &myshmds);

        /* Display old permissions */
        printf("Old permissions were: %o\n", myshmds.shm_perm.mode);

        /* Convert and load the mode */
        sscanf(mode, "%o", &myshmds.shm_perm.mode);

        /* Update the mode */
        shmctl(shmid, IPC_SET, &myshmds);

        printf("New permissions are : %o\n", myshmds.shm_perm.mode);
}

usage()
{
        fprintf(stderr, "shmtool - A utility for tinkering with shared
memory\n");
        fprintf(stderr, "\nUSAGE:  shmtool (w)rite <text>\n");
        fprintf(stderr, "                (r)ead\n");
        fprintf(stderr, "                (d)elete\n");
        fprintf(stderr, "                (m)ode change <octal mode>\n");
        exit(1);
}
```

第7章 声音编程

一台PC机至少有一个声音设备：内部扬声器。但是，用户还可以买一块声卡插入自己的计算机，以提供更复杂的声音设备。读者可以查看《 Linux Sound User ’s Guide》或者《Sound HOWTO》来阅读有关声卡的内容。

7.1 内部扬声器编程

不管读者相信还是不相信， PC机上的扬声器是Linux控制台的一部分，因此它是个字符设备。所以，可以使用ioctl ( )请求来操作它。对内部扬声器而言，存在下面两个请求：

1. KDMKTONE

使用内核定时器产生一段特定时间长的哔哔声。

例子：ioctl (fd, KDMKTONE, (long) argument).

2. KIOCSOUND

产生一个持续的哔哔声，或者中止当前正在发出的哔哔声。

例子：ioctl (fd, KIOCSOUND, (int) tone).

变元由两部分组成，低位字是声调 (tone)的值，而高位字是声音持续的时间周期。 tone的值并不是指声音频率。 PC主板定时器8254的时钟频率是1.19MHz，所以它是1190000/秒。周期长度是以时钟滴答的次数为单位测量的。上面这两个 ioctl调用将立即返回，所以用户可以使用这种方法产生哔哔声，而无需阻塞程序。

KDMKTONE应该用来产生警告信息，因为用户无需考虑将这种声音停下来。

KIOCSOUND可以用来演奏悦耳的音乐，就像在例子程序 splay中一样。为了停下它的声音，用户可以使用为0的tone值来调用KIOCSOUND。

7.2 声卡编程

作为一个程序员，了解当前 Linux系统中是否插了一块声卡是很重要的。有一个检查的方法是查看/dev/sndstat。如果打开/dev/sndstat失败，并且errno等于ENODEV，那么说明没有激活声音驱动程序，这意味着用户无法得到内核声音驱动程序的帮助。除此之外，用户还可以试着去打开 /dev/dsp，也可以达到查看声卡的目的。如果它没有连接到 pcsnd驱动程序，则open ( )操作将不会失败。

如果用户希望在硬件级别上使用声卡，则通过使用 outb ( )和inb ( )调用的一些组合，将可以检测到用户正在查找的声卡。

用户在应用程序中使用声音驱动程序，可能会造成这些应用程序在其他 i386系统上也能正常运行。这是因为聪明的程序员们已经决定在 Linux、isc、FreeBSD以及大部分其他基于i386的系统上使用相同的驱动程序。如果 Linux是在提供同一个声音设备接口的其他体系结构下运行，则这个特征在移植程序时会大有帮助。声卡不是 Linux控制台的一部分，它是一个特殊的设备。声卡主要提供三个重要的特征：

- 数字取样输入/输出
- 频率调制输出
- MIDI接口

这三个特征都有它们自己的设备驱动程序接口。数字取样的接口是 /dev/dsp。频率调制的接口/dev/sequencer，而MIDI接口是/dev/midi。声音设备(如音量、平衡或者贝斯)可以通过/dev/mixer接口来控制。为了兼容性的需要，还提供了一个 /dev/audio设备，该设备可用于读SUN_law的声音数据，但它是映射到数字取样设备的。

如果读者猜想自己可以使用ioctl ( )来操作这些设备，那么读者猜对了。 ioctl ( )请求是在<linux/soundcard.h>中定义的，它们以SNDCTL_开头。

因为我自己并没有声卡，所以希望有志之士接着完善本章。

第8章　字符单元图形

本章讨论的是基于字符的屏幕输入和输出，而不是基于像素的屏幕输入和输出。这里所提到的字符，是指像素的一种组合，它可以根据字符集而变化。用户的图形卡早已提供了一种或多种的字符集，默认是以文本(字符集)模式进行操作的，这是因为文本的处理速度比像素图形的处理速度要快得多。对终端来说，用户除了可以把它用作简单的 (沉默的)并且是烦人的文本显示器以外，还可以用它们来完成很多任务。下面将介绍怎样利用　Linux终端尤其是Linux控制台所提供的特殊功能。

- printf、sprintf、fprintf、scanf、sscanf、fscanf —— 使用Linux的这些函数，用户可以把格式化的字符串输出到标准输出和标准错误中，或者输出到其他定义为 FILE *stream形式的流中(例如文件)。Scanf(……)提供了一个类似的方法，可以从 Linux读取格式化的输入。
- termcap —— 在ASCII文件/etc/termcap中，有一个终端描述项目的集合，它就是终端功能数据库(TERMinal CAPabilities database)。在这里用户可以找到一些信息，介绍如何显示特殊字符，如何执行操作(删除、插入字符或者行，等等)，以及怎样初始化一个终端。这个数据库可以通过编辑器vi来使用。系统包含一些视图库函数，可以读和使用终端功能(termcap (3x))。使用这个数据库，程序就可以用相同的代码处理各种终端了。termcap数据库和库函数只提供了对终端的低级访问。程序员必须亲自完成诸如改变属性或颜色、参数化输出以及优化等工作。
- terminfo数据库 —— 终端信息数据库(TERMinal INFOrmation database)是基于termcap数据库的，它也介绍了终端的功能。但是所处的级别要比 termcap高。使用terminfo数据库，程序可以轻易地改变屏幕的属性，并且可以使用特殊的键，如功能键，等等。该数据库位于/usr/lib/terminfo/[A-z, 0-9]*中。每个文件描述一个终端。
- curses —— Terminfo是个不错的数据库，非常适合在程序中用来进行终端处理。而(BSD_)CURSES库则使用户可以对终端进行高级访问，它是基于 terminfo数据库的。Curses允许用户在屏幕上打开并操作一个窗口，它提供了一个完整的输入和输出函数的集合，并可以在150多个终端上用不依赖于终端的方式改变视频的属性。 curses库可以在/usr/lib/libcurses.a中找到。这是一个BSD版本的curses库。
- ncurses —— Ncurses是一个更高级的版本。在版本1.8.6中，它应该与定义在SYSVR4中的AT&T的curses库兼容。它具有一些扩展的功能，如颜色操作、输出的特殊优化、终端特定的优化，等等。该库已经在大量的系统上测试通过，如 Sun OS、HP和Linux。笔者建议用户使用ncurses，而不要使用其他库。在SYSV Unix系统上(如Sun的Solaris)，应该存在一个curses库，它的功能与ncurses相同(实际上solaris的curses提供的函数更多，并提供鼠标支持)。

下面几节将介绍怎样使用不同的服务包来访问终端。在　Linux下，用户拥有 termcap的GNU版本，用户可以使用ncurses，而不是curses。

8.1 libc中的I/O函数

8.1.1 格式化输出

printf(……)函数的功能是提供格式化的输出，并允许变元的转换。

```
int fprintf(FILE *stream, const char *format, ...),
```

上面这个函数将对输出 (在……中填写的变元)转换，并把它写到 stream中，同时还将把format中定义的格式也写到stream中。该函数将会返回实际写的字符的个数；如果出错则返回一个负数。

format包含两种类型的对象：

1. 需要输出的普通字符。
2. 关于如何转换或者格式化变元的信息。

格式信息必须以%打开，后面是该格式的值，再后面是需要转换的字符 (如果要打印%自身，需要使用%%)。格式的可能值如下：

标志

－

格式化的变元在域内向左靠输出 (默认的方式是在变元域中向右靠输出)。

＋

每个数打印时都要带上符号，例如 +12或者－2.32

Blank

当第一个字符不是符号时，需要插入一个空格

0

对于数字转化而言，如果域的宽度定义得超过数字的位数，则在数字的左边用 0来填充。

#

根据变元转换的不同，输出也将不同

- 0 第一个数字为0。
- x或X 在变元的前面会打印0x或0X。
- e、E、f或F 输出将带有小数点。
- g或G 变元尾部的0会打印出来。

```
A number for the minimal field width.
```

转换以后的变元打印的宽度至少应该与变元本身一样大。通过在格式中指定一个数值，用户就可以让字段宽度稍微大一点。如果格式化的变元比较小，则字段的宽度将用 0或者空格来填充。

```
A point to separate the field width and the precision.
A number for the precision.
```

这个转换的可能值如表8-1所示。

```
int printf(const char *format, ...)
```

这个函数与fprintf (stdout, ……)相同。

```
int sprintf(char *s, const char *format, ...)
```

这个函数与printf (…)相同，只有一点区别，即输出将被写到字符指针 S中(带一个后缀\0)。

用户必须为S分配足够的内存空间。

```
vprintf(const char *format, va_list arg)
vfprintf(FILE *stream, const char *format, va_list arg)
vsprintf(char *s, const char *format, va_list arg)
```

这三个函数与前面的函数相同，不过变元列表被设置为 arg。

表8-1 libc-printf转换

注意 出版者注：作者尚未提供此表。

8.1.2 格式化输入

就像在格式化输出中使用printf (…)一样，用户可以把scanf (…)用于格式化输入。

```
int fscanf(FILE *stream, const char *format, ...)
```

fscanf (…)从stream中读数据，并按照在format中定义的规则对其进行转换。转换结果将放置在变元……中(**注意**：所有变元**必须**是指针)。当format中的转换规则已经用完时，读操作即告结束。如果第一个转换遇到文件尾，或者发生了某些错误，则 fscanf (…)将返回EOF，否则它将返回转换以后变元的数目。

format可以存放有关如何格式化输入变元的规则 (见下表8-2)。

它还可以包括：

- 空格或者制表符，这两者均被忽略。
- 任意的普通字符(除了%以外)，这些字符必须出现在输入的对应位置上。
- 转换规则，它的构成如下：一个 %号，可选的符号*(这个符号使fscanf (…)可以赋予一个变元)、一个可选的数字、一个可选的符号 h、l或L(它规定了输入目标的长度)以及转换字符。

```
int scanf(const char *format, ...)
```

该函数与fscanf (stdin, …)类似。

```
int sscanf(char *str, const char *format, ...)
```

该函数与scanf(…)相同，除了它的输出是来自 str以外。

表8-2 libc-scanf转换

注意 出版者注：作者尚未提供此表。

8.2 termcap库

8.2.1 前言

termcap库是termcap数据库所提供的API，该库可以在/etc/termcap/中找到。库函数允许进行如下动作：

- 获得当前终端的描述：tgetent (…)。
- 在描述中查找信息：tgetnum (…)、tgetflag (…)、tgetstr (…)。
- 以终端特定的格式编码数字参数：tparam (…)、tgoto (…)。
- 计算并执行填充：tputs (…)。

如果程序中用到termcap库，则它必须包含.h头文件，并与这些头文件相链接。

注意 出版者注：作者尚未提供表

termcap函数是一些独立于终端的过程，但它们只允许程序员对终端进行低级访问。如果需要稍高级别的服务包，应该使用curses或者ncurses。

8.2.2 获得终端描述

```
int tgetent(void *buffer, const char *termtype)
```

在Linux操作系统上，当前终端的名称包含在环境变量 TERM中。所以termtype是调用(3)的返回结果。

对buffer而言，在使用termcap的GNU版本时无需分配任何内存，这一点实际上根据我们对Linux的认识也应该想到了。否则，如果使用的不是 GNU版本的termcap，用户就必须分配2,048个字节。(以前缓冲区只需要1,024个字节，但是现在这个大小已经翻倍了。)

tgetent (…)在成功时将返回1；而如果找到数据库却没有对应TERM的项目，则返回0；如果出错则返回其他不同的值。

下面的例子说明了tgetent (…)的用法：

注意 出版者注：作者尚未提供任何例子。

默认情况下，termcap使用/etc/termcap/作数据库。如果环境变量TERMCAP被设置成其他的值，例如 $HOME/mytermcap，则所有的函数将不再使用 /etc/termcap，而使用$HOME/mytermcap。TERMCAP前面没有斜线，这个定义好的值将用作终端的名称。

8.2.3 查看终端描述

每一段信息都被称为一个权能(capability)，每个权能都是两个字母的代码，这两个字母代码的后面是该权能的值。权能的类型包括：

- 数值型：例如co —— 列的数目。
- 布尔型或标志：例如hc —— 硬拷贝终端。
- 字符串：例如st —— 设置制表符停止。

每个权能都只有一种值的类型(co永远是数值型、hc永远是标志，而st永远是字符串)。因为存在三种不同类型的值，所以相应地需要用三个函数来查询它们。 char *name是权能的两个字母的代码。

```
int tgetnum(char *name)
```

它获得类型为数值型的权能的值，例如 co的值。如果权能是可用的， tgetnum (…)将返回该数值，否则返回1。(注意，返回值非负。)

```
int tgetflag(char *name)
```

它获得一个布尔型的权能值(或者标志型)。如果该标志存在，则返回1，否则返回0。

```
char *tgetstr(char *name, char **area)
```

它获得一个字符串型的权能值。如果该值存在，则返回一个指向该字符串的指针；否则，如果不存在则返回 NULL。在GNU版本中，如果 area为NULL，则termcap将自己分配内存。termcap将不会再引用该指针。所以在离开程序之前别忘了释放 name。我们提倡使用这个方法，因为用户不知道指针将需要多少空间，所以应该让 termcap帮用户完成分配工作。

8.2.4 termcap权能

布尔型权能	
5i	打印机将不回送到屏幕上
am	自动对齐，这意味着自动换行
bs	Control+H (8 dec.)执行一个后退操作
bw	在最左一列作后退操作将转换到前一行的最右边
da	显示保留在上面的屏幕
db	显示保留在下面的屏幕
eo	按空格将会删除光标位置的所有字符
es	在状态行中可以使用Esc组合键和特殊字符
gn	普通设备
hc	这是个硬拷贝终端
HC	当光标不在最底下一行时将难以看到它
hs	有一个状态行
hz	Hazeltine错误，终端不能打印发音符号
in	终端在空白区域插入null，而不是插入空格
km	终端有一个元键
mi	光标移动是以插入方式工作的
ms	光标移动是以standout/underline方式工作的
NP	没有填充字符
NR	ti不会返转te
nx	不有填充，但必须使用XON/XOFF
os	终端可能会叠印
ul	尽管终端不能叠印，但它可以划下划线
xb	Beehive信号，F1发送ESCAPE，F2发送^c
xn	新行/环绕信号
xo	终端使用XON/XOFF协议
xs	在标准输出文本上打印的文本将显示在标准输出中
xt	Teleray信号，破坏性制表符，以及奇数standout方式

数值型权能

字符串权能

8.3 Ncurses简介

本章将使用下述的术语：

- 窗口：这是一个内部表述，它包含屏幕某部分的映像。 WINDOW是在curses.h中定义的。
- 屏幕：它是一个窗口，它的大小就是整个屏幕的大小 (从左上角到右下角)。

• 终端：它是一个特殊的屏幕，它包含着当前屏幕是什么样的有关信息。
• 变量：在curses.h中定义了如下一些变量和常量：

WINDDW *curscr：	当前屏幕
WINDOW *stdscr：	标准屏幕
int LINES：	终端上的行数
int COLS：	终端上的列数
bool TRUE：	真标志，1
bool FALSE：	假标志，0
int ERR：	错误标志，－1
int OK：	OK标志，0

• 函数：在函数的描述中，变元一般具有如下类型：

win：	WINDOW*
bf：	bool
ch：	chtype
str：	char*
chstr：	chtype*
fmt：	char*
否则：	int

一般来说，使用ncurses库的程序看起来像下面一样：

注意 出版者注：作者尚未提供例子程序。

ncurses.h中定义了ncurses的变量和类型，例如WINDOW和函数原型，所以应该包含这个头文件。它将会自动包含stdio.h、ncurses/unctrl.h、stdarg.h和stddef.h。

initscr ( )可以用来初始化ncurses数据结构，并可用于读入正确的terminfo文件。内存也是在这个函数中分配的。如果发生了错误，则 initscr将返回ERR，否则将返回一个指针。此外，屏幕将被刷新并被初始化。

endwin ( )将清除所有已经分配的ncurses资源，并把tty方式恢复到在调用initstr ( )以前的状态。在调用ncurses库的任意其他函数之前，必须先调用 initscr ( )；而在用户退出程序之前，必须调用endwin ( )。如果用户希望在多个终端上执行输出，则用户可以使用 newterm (…)，而不要使用initscr ( )。

可以用下面的方法编译该程序：

注意 出版者注：原书缺少正文。

用户可以加入自己愿意加入的任何标志 (gcc (1))，因为ncurses.h的路径已经改变了，所以用户必须加入下面一行：

注意 出版者注：原书缺少正文。

否则将无法找到ncurses.h、nterm.h、termcap.h以及unctrl.h。Linux可能的其他标志为：
• 2：让gcc做一些优化工作。
• _ansi：用于ansi一致性C代码。
• _Wall：将打印出所有的警告信息。
• _m486：将对Intel 486使用优化的代码(该二进制代码也可用在Intel 386上)。

ncurses库可在/usr/lib/中找到。ncurses库总共有三种版本：

• libncurses.a —— 普通的ncurses库。

• libdcurses.a —— 用于调试的ncurses库。

• libpcurses.a —— 用于配置的ncurses库(不知是否因为1.8.6 libpcurses.a不再存在的原因)。

• libcurses.a —— 实际上不是ncurses库的第四个版本，它是原始的 BSD curses(在笔者的Slackware 2.1.0中，它是bsd服务包)。

屏幕的数据结构称为窗口，它是在 ncurses.h中定义的。在某种程度上，窗口就像内存中的一个字符数组，程序员可以对其进行操作，无需输出到终端。默认的窗口大小就是终端的大小，用户可以用newwin (…)创建其他的窗口。

为了可以更好地更新物理终端， ncurses声明了另一个窗口。这是一个关于终端实际是什么样子的映像，并且它还是一个关于终端应该是什么样子的映像。输出必须在用户调用 refresh ( )时进行。然后Ncurses就可以使用Linux中的信息来更新物理终端了。库函数在更新进程中将使用内部优化，这样用户就可以改变不同的窗口，并立即以最优的方式更新屏幕。

通过使用ncurses函数，用户可以对数据结构窗口进行操作。以 w开头的函数将允许用户指定窗口；而其他函数一般用于影响 Linux。以mv开头的函数将首先把光标移动到位置 y, x上。

有一个字符类型是 chtype，这是一种无符号长整型，它可用于存放一些附加的信息 (属性等等)。

Ncurses使用数据库。一般来说该数据库位于 /lib/terminfo/，ncurses将在那里查找本地终端定义。如果用户希望在不改变原始的 terminfo的情况下测试终端的一些定义，用户可以设置环境变量。ncurses将检查这个变量，并使用存放在那里的定义，而不使用 /usr/lib/terminfo/中的定义。

当前ncurses版本是1.8.6 ( )。

注意 出版者注：当前版本是4.2, www.gnu.ai.mit.edu/software/ncurses/ncurses.html。

在本章的末尾，读者可以读到一个表，它总结了 BSD_Curses、ncurses以及Sun Os 5.4的curses。如果读者希望查找某个特定的函数以及它是在哪里定义的，则可以查找那个表。

8.4 初始化

• WINDOW *initscr ( ) —— 在使用ncurses的程序中，一般首先应该调用这个函数。在有些情况下，在调用initscr ( )以前，可能会需要调用slk_init (int)、filter ( )、ripoffline (…)或者use_env (bf)。当使用多个终端时 (或者可能测试权能时)，用户可以使用 newterm (…)来取代initscr ( )。

initscr ( )将读入正确的terminfo文件，初始化ncurses数据结构，并为其分配内存，将其设置为终端所具有的值。该函数将返回一个指针；而如果出错则返回 ERR。用户无需初始化该指针。

initscr ( )将为用户完成初始化工作。如果它的返回值是 ERR，则用户的程序将退出。这是因为没有ncurses函数可以正常工作。

• SCREEN *newterm (char *type, FILE *outfd, FILE *intd)

如果具有多个终端输出，则对用户访问的每个终端都应该调用 newterm (…)，而不用initscr ( )。type是终端的名称，包括在$TERM中(如ansi, xterm, vt100等等)。outfd是输出指针，

而infd是输入指针。对使用newterm (…)打开的每个终端都应该调用endwin ( )来退出。

• SCREEN *set_term (SCREEN *new)

通过使用 set_term(SCREEN)，用户可以切换当前终端。所有的函数都将在set_term(SCREEN)设置的当前终端上起作用。

• int endwin ( )

endwin ( )将执行清理工作，把终端模式恢复到调用 initscr ( )以前的状态，并把光标移动到屏幕的左上角。在调用endwin ( )退出程序之前，不要忘了关闭所有打开的窗口。

在调用endwin ( )之后还可以调用refresh ( )，它将把终端恢复到调用initscr ( )以前的状态(可视模式)，否则屏幕将被清除(不可视模式)。

• int isendwin ( )

如果调用endwin ( )之后还调用了refresh ( )，则该函数返回TRUE，否则返回FALSE。

• void delscreen (SCREEN *sp)

当SCREEN不再需要时，在调用完endwin ( )之后，调用delscreen (SCREEN)将可以释放所有占用的资源。(注，此函数尚未实现。)

8.5 窗口

窗口可以创建、删除、移动、拷贝、按掀、复制等等。

• WINDOW *newwin (nlines, ncols, begy, begx)

begy和begx是窗口左上角的坐标。nlines是一个整数，它存放着行的数目，而ncols也是一个整数，它存放着列的数目。

图3-8-1 Ncurses-newwin方案

注意 出版者注：原书缺图。

该窗口的左上角位于第10行，第10列，该窗口有10行，60列。如果nlines为零，则窗口将有LINES_begy行。同理可知，如果ncols为零，则窗口将有COLS_begx列。

如果用户调用newwin且把所有的参数设置为零，则打开的窗口的大小将与屏幕的大小相同。

使用……(原书缺)我们可以在屏幕的中间打开一个窗口，不管它的尺寸有多大：

(原书缺)

这命令将在屏幕的中间打开一个 22行和70列的窗口。在打开窗口以前应该查看屏幕的大小。在Linux控制台中，我们可以有 25行以上，80列以上，但是在xterm中情况却并不一定如此(它们是可以缩放的)。

此外，用户还可以使用(……)把两个窗口调整为屏幕大小：

在例子目录中有一些C程序，读者可以阅读这些程序，找到更多的解释。

• int delwin (win)

它删除窗口win。如果存在子窗口，则在删除 win以前先要删除这些子窗口。这个函数将释放win所占据的所有资源。在调用 endwin ( )之前用户应该删除所有的窗口。

• int mvwin (win, by, bx)

它将把窗口移到坐标(by，bx)处。如果这个坐标将把窗口移出屏幕边界的范围，则此函数

什么也不做，并返回ERR。

• WINDOW *subwin (origwin, nlines, ncols, begy, begx)

它返回一个位于origwin窗口中间的子窗口。如果用户改变这两个窗口 (origwin或者那个新窗口)中的一个，则这种改变将会同时反映到这两个窗口上。在下一次调用 refresh ( )之前，先要调用touchwin (origwin)。

begx和begy是相对于屏幕的，而不是相对于 origwin的。

• WINDOW *derwin (origwin, nlines, ncols, begy, begx)

此函数与subwin (…)相同，只不过这里的 begx和begy是相对于窗口origwin的，而不是相对于屏幕的。

• int mvderwin (win, y, x)

此函数将把win移到父窗口内。(注意：此函数尚未实现)。

• WINDOW *dupwin (win)

此函数复制窗口win。

• Duplicate the window win.
• `int syncok(win, bf)`
• `void wsyncup(win)`
• `void wcursyncup(win)`
• `void wsyncdown(win)` (注：尚未实现)
• `int overlay(win1, win2)`
• `int overwrite(win1, win2)`
• overlay (…)将把win1中的所有文本拷贝到win2中，但是不拷贝空格。 overwrite (…)也是做文本拷贝工作的函数，但它拷贝空格。
• int copywin (win1, win2, sminrow, smincol, dminrow, dmincol, dmaxrow, dmaxcol, overlay)
• 它所做的工作与overlay (…)和overwrite (…)相似，但是该函数还可以让程序员选择拷贝窗口的哪个区域。

8.6 输出

• `int addch(ch)`
• `int waddch(win, ch)`
• `int mvaddch(y, x, ch)`
• `int mvwaddch(win, y, x, ch)`

这些函数可以用于把字符输出到窗口。它们将操作窗口，为了把字符写到屏幕上，用户必须调用函数 refresh ( )。addch (…)和waddch (…)把字符 ch输出到屏幕上或者 win中，mvaddch (…)和mvwaddch (…)执行相同的工作，但它们首先会把光标移动到位置 y, x上。

• `int addstr(str)`
• `int addnstr(str, n)`

- int waddstr(win, str)
- int waddnstr(win, str, n)
- int mvaddstr(y, x, str)
- int mvaddnstr(y, x, str, n)
- int mvwaddstr(win, y, x, str)
- int mvwaddnstr(win, y, x, str, n)

这些函数把字符串输出到某个窗口，它们的作用等价于对 addch (…)进行一系列的调用。str是一个以null结束的字符串(“ blafoo\0 ”)。名称中有字母w的函数把字符串str写到窗口win中，而其他函数则把字符串输出到屏幕上。名称中有字母 n的函数将输出字符串中的n个字符。如果n的值为－1，则输出整个字符串str。

- int addchstr(chstr)
- int addchnstr(chstr, n)
- int waddchstr(win, chstr)
- int waddchnstr(win, chstr, n)
- int mvaddchstr(y, x, chstr)
- int mvaddchnstr(y, x, chstr, n)
- int mvwaddchstr(win, y, x, chstr)
- int mvwaddchnstr(win, y, x, chstr, n)

这些函数把chstr拷贝到窗口映射中(或者win中)。起始位置是当前光标所处的位置。名称中有字母n的函数将输出chstr中的n个字符。如果n的值为－1，则输出整个字符串 chstr。光标的位置不移动，并且不检查控制字符。这些函数比 addstr (…)那些过程的运行速度要快。 chstr是指向chtype数组的一个指针。

- int echochar(ch)
- int wechochar(win, ch)

这两个函数等价于先调用 addch (…) (waddch (…))，再接着调用 refresh ( ) (wrefresh (win))。

8.6.1 格式化输出

```
int printw(fmt, ...)

int wprintw(win, fmt, ...)

int mvprintw(y, x, fmt, ...)

int mvwprintw(win, y, x, fmt, ...)

int vwprintw(win, fmt, va_list)
```

这些函数对应于printf (…)以及Linux中的其他函数。

在服务包中printf (…)用于格式化输出。用户可以定义一个输出字符串，在其中包含不同类型的变量。

如果需要使用vwprintw (…)，用户也必须包含.h头文件。

8.6.2 插入字符/行

```
int insch(c)

int winsch(win, c)

int mvinsch(y,x,c)

int mvwinsch(win,y,x,c)
```

这些函数把字符ch插入到光标的左边，光标后面的所有字符则向右移动一个位置。在这一行最右端的字符可能会丢失。

```
int insertln()

int winsertln(win)
```

这两个函数在当前行的上方插入一个空行 (最底下的一行将被丢失)。

```
int insdelln(n)

int winsdelln(win, n)
```

如果n为正数，则这些函数将在适当的窗口的当前光标上方插入 n行(这样一来最底下的n行将丢失)；如果n为负数，则光标下面的n行将被删除，余下的行将上升，顶替它们的位置。

```
int insstr(str)

int insnstr(str, n)

int winsstr(win, str)

int winsnstr(win, str, n)

int mvinsstr(y, x, str)

int mvinsnstr(y, x, str, n)
int mvwinsstr(win, y, x, str)
int mvwinsnstr(win, y, x, str, n)
```

这些函数将在当前光标的左边插入str(字符的个数不能超过一行的限度)。在光标右边的字符将右移，如果到达行尾，则字符将丢失，光标位置不变。

y和x是指在插入str以前先要把光标移动到的坐标， n是要插入的字符的数目 (如果n为0则插入整个字符串)。

8.6.3 删除字符/行

```
int delch()
```

```
int wdelch(win)

int mvdelch(y, x)

int mvwdelch(win, y, x)
```

删除光标左边的字符，并把光标右边余下的字符向左移动一个位置。
y和x是在进行删除操作以前要把光标移动到的坐标。

```
int deleteln()

int wdeleteln(win)
```

删除光标下面的一行，并把下面所有的其他行都向上移动一个位置。此外，屏幕最底下的一行将被清除。

8.6.4 方框和直线

```
int border(ls, rs, ts, bs, tl, tr, bl, br)

int wborder(win, ls, rs, ts, bs, tl, tr, bl, br)

int box(win, vert, hor)
```

这些函数在窗口的边界 (或者win的边界)画上方框。在下面的表格中，读者将可以看到字符，以及它们的默认值。当用零去调用 box (…)时将会用到这些默认值。在下面的图中读者可以看到方框中字符的位置。

表8-3 Ncurses：边框字符

图3-8-2 Ncurses：方框字符

注意 出版者注：表和图原书缺。

```
int vline(ch, n)

int wvline(win, ch, n)

int hline(ch, n)

int whline(win, ch, n)
```

这些函数将从当前光标位置开始画一条水平线或者垂直线。 ch是画线所使用的字符， n是要画的字符的个数，光标位置并不移动。

8.6.5 背景字符

```
void bkgdset(ch)

void wbkgdset(win, ch)
```

这两个函数设置窗口或者屏幕的背景字符和属性。 ch的属性将和窗口中所有非空格的字

符的属性进行OR操作。背景是窗口的一部分，将不会随着滚动、输入或输出而改变。

```
int bkgd(ch)
int wbkgd(win, ch)
```

它们将把背景字符改为ch，把属性改为ch的属性。

8.7 输入

```
int getch()

int wgetch(win)

int mvgetch(y, x)

int mvwgetch(win, y, x)
```

getch ( )将从终端读取输入，读取的方式取决于是否设置了延迟模式。如果设置了延迟模式，则getch ( )将一直等待，直到用户按下一个键为止；如果没有设置延迟模式，则它将返回输入缓冲区中的数据，如果输入缓冲区为空，则它将返回 ERR。mvgetch (…)和mvwgetch (…)首先把光标移动到位置 (y,x)上。名称中有 w字母的函数将从与窗口 win相关的终端读取输入，getch ( )和mvgetch (…)则从屏幕相关的终端读取。

如果使能了keypad (…)，在用户按下某个功能键时， getch ( )将返回一个代码，该代码在.h头文件中被定义为 KEY_*宏。如果用户按下 Esc键(它可能会是某个组合功能键的第一个键)，则ncurses将启动一个一秒钟的定时器，如果在这一秒钟时间内没有按完其余的键，则返回Esc。否则就返回功能键的值。 (如果需要的话，可以使用 notimeout ( )来关闭第二个定时器。)

```
int ungetch(ch)
```

这个函数将把字符ch送回输入缓冲区。

```
int getstr(str)
int wgetstr(win, str)
int mvgetstr(y, x, str)
int mvwgetstr(win, y, x, str)
int wgetnstr(win, str, n)
```

这些函数的作用相当于对getch ( )进行一系列的调用，直到接收到一个新行。行中的字符存放在str中(所以，在调用getstr (…)之前，不要忘记给字符指针分配内存)。如果打开了回送，则字符串将被显示出来 (使用noecho ( )可以关闭回送)，而用户的删除字符以及其他特殊字符也会被解释出来。

```
chtype inch()

chtype winch(win)
chtype mvinch(y, x)
chtype mvwinch(win, y, x)
```

这些函数从屏幕或窗口返回一个字符，因为返回值的类型是 chtype，所以还包括了属性信息。这一信息可以使用常量 A_*从字符中扩展得到。

```
int instr(str)
int innstr(str, n)
int winstr(win, str)
int winnstr(win, str, n)
int mvinstr(y, x, str)
int mvinnstr(y, x, str, n)
int mvwinstr(win, y, x, str)
int mvwinnstr(win, y, x, str, n)
```

这些函数从屏幕或窗口返回一个字符串。(注：这些函数尚未实现)

```
int inchstr(chstr)
int inchnstr(chstr, n)
int winchstr(win, chstr)
int winchnstr(win, chstr, n)
int mvinchstr(y, x, chstr)
int mvinchnstr(y, x, chstr, n)
int mvwinchstr(win, y, x, chstr)
int mvwinchnstr(win, y, x, chstr, n)
```

这些函数将从屏幕或窗口返回一个 chtype字符串。该字符串包含了每个字符的属性信息(注：尚未实现，在ncurses库中没有包含lib_inchstr)。

格式化输入

```
int scanw(fmt, ...)
int wscanw(win, fmt, ...)
int mvscanw(y, x, fmt, ...)
int mvwscanw(win, y, x, fmt, ...)
int vwscanw(win, fmt, va_list)
```

这些函数类似于scanf (…)。如果在调用这些函数之前调用了 wgetstr (…)，则它的结果将会成为这些函数的输入。

8.8 选项

8.8.1 输出选项

```
int idlok(win, bf)
void idcok(win, bf)
```

这两个函数为窗口使能或者关闭终端的 insert/delete特征(idlok (…)针对一行，而idcok (…)则针对字符)。(注：idcok (…)尚未实现)

```
void immedok(win, bf)
```

如果bf设置为TRUE，则对窗口win的每一次改变都将导致物理屏幕的一次刷新。这将使程序的性能降低，所以默认的值是 FALSE。(注：此函数尚未实现)

```
int clearok(win, bf)
```

如果bf值为TRUE，则下一次调用wrefresh (win)时将会清除屏幕，并完全地把它重新画一遍(就像用户在编辑器vi中按下Ctrl+L一样)。

```
int leaveok(win, bf)
```

默认的行为是，ncurses让物理光标停留在上次刷新窗口时的同一个位置上。不使用光标的程序可以把leaveok (…)设置为TRUE，这样一般可以节省光标移动所需要的时间。此外，ncurses将试图使终端光标不可见。

```
int nl()
int nonl()
```

这两个函数控制新行的平移。使用 nl ( )可以打开平移，这样在回车时就会平移到新的一行，在输出时就会走行。而 nonl ( )可以把平移关上。关上平移之后，ncurses做光标移动操作时速度就会快一些。

8.8.2 输入选项

```
int keypad(win, bf)
```

如果bf为TRUE，该函数在等待输入时会使能用户终端的键盘上的小键盘。ncurses将返回一个键代码，该代码在 .h头文件中被定义为 KEY_*宏，它是针对小键盘上的功能键和方向键的。对于PC键盘来说，这一点是非常有帮助的，因为这样用户就可以使能数字键和光标键。

```
int meta(win, bf)
```

如果bf为TRUE，从getch ( )返回的键代码将是完整的8位(最高位将不会被去掉)。

```
int cbreak()
int nocbreak()
int crmode()
int nocrmode()
```

cbreak ( )和nocbreak ( )将把终端的CBREAK模式打开或关闭。如果CBREAK打开，则程序就可以立刻使用读取的输入信息。如果 CBREAK关闭，则输入将被缓存起来，直到产生新的一行(注意：crmode ( )和nocrmode ( )只是为了提供向上兼容性，不要使用它们)。

```
int raw()
int noraw()
```

这两个函数将把RAW模式打开或关闭。RAW与CBREAK相同，它们的区别在于 RAW模式不处理特殊字符。

```
int echo()
int noecho()
```

如果把echo ( )设置为TRUE，则用户所敲的输入将会回送并显示出来，而 noecho ( )则对此保持沉默。

```
int halfdelay(t)
```

此函数与cbreak()相似，但它要延迟t秒钟。

```
int nodelay(win, bf)
```

终端将被设置为非阻塞模式。如果没有任何输入则 getch ( )将返回ERR，否则如果设置为FALSE，则getch ( )将等待，直到用户按下某个键为止。

```
int timeout(t)
int wtimeout(win, t)
```

笔者提倡大家使用这两个函数，而不要使用 halfdelay (t)和nodelay (win, bf)。getch ( )的结

果取决于t的值。如果t是正数，则读操作将被阻塞t毫秒；如果t为零，则不发生任何阻塞；如果t是负数，则程序将阻塞，直到有输入为止。

```
int notimeout(win, bf)
```

如果bf为TRUE，则getch ( )将使用一个特殊的定时器 (一秒钟长)。到时间以后，再对以Esc等键打头的输入序列进行解释。

```
int typeahead(fd)
```

如果fd是-1，则不检查超前键击，否则ncurses将使用文件描述符fd来进行这些检查。

```
int intrflush(win, bf)
```

当bf为TRUE时使能该函数。在终端上按下任意中断键 (quit、break…)时，所有的输出将会刷新到tty驱动程序队列中。

```
void noqiflush()
void qiflush()
```

(注，这两个函数尚未实现)

8.8.3 终端属性

```
int baudrate()
```

此函数返回终端的速度，以bps为单位。

```
char erasechar()
```

此函数返回当前删除的字符。

```
char killchar()
```

此函数返回当前杀死的字符。

```
int has_ic()
int has_il()
```

如果终端具有插入/删除字符的能力，则has_rc()将返回TRUE。如果终端具有插入/删除行的能力，则has_il ( )将返回TRUE，否则这两个函数将返回ERR。(注：尚未实现)

```
char *longname()
```

此函数所返回的指针允许用户访问当前终端的描述符。

```
chtype termattrs()
```

(注：此函数尚未实现)

```
char *termname()
```

这个函数从用户环境中返回TERM的内容。(注：尚未实现)

8.8.4 使用选项

读者现在已经了解了窗口选项和终端方式，下面将介绍它们的用法。

首先，在Linux上用户必须使用小键盘。这将使用户可以使用 PC键盘上的光标键和数字键。

目前有两种类型的输入：

1. 程序希望用户输入一个键，然后根据这个键再调用某个函数。 (例如，等待用户按“ q ”，按“ q ”代表quit)。

2. 程序希望用户在屏幕上的某个位置输入一个字符串。例如，数据库中的某个目录或者地址。

首先我们使用下面的选项和方式，这样while循环将可以正确地工作。

在用户按下某个键之前，程序将一直挂起。如果用户所按的是“ q”，则调用退出函数，否则需要一直等待其他输入。

程序需要依次判断switch语句的条件，直到找到符合条件的输入函数。使用KEY_*宏可以检查特殊的键，例如小键盘上的光标键。在文件浏览器中循环看起来如下所示：

(译者注：原文缺)

在这一秒钟时间内，我们只需要设置echo ( )，用户输入的字符将会显示到屏幕上。如果想把字符显示在用户所希望的位置，可以使用函数move (…)或者wmove (…)。

或者用户也可以打开一个窗口，在窗口中有一个掩模 (与窗口不同的一些其他颜色)，然后要求用户输入一个字符串：

注意 出版者注：作者尚未提供实例。

在实例目录中有一些C语言程序，读者可以在这些程序中了解到更多的情况。

8.9 更新终端

正如以前所介绍的那样， ncurses窗口是内存中的一个映像。这意味着在进行刷新以前，对窗口的任意改变将不会显示在屏幕上，这将优化屏幕的输出，因为这样一来用户就可以进行大量的操作，然后再一次性地把它们刷新到屏幕上。否则，每一次改变窗口都将刷新到终端，这将会降低程序的性能。

```
int refresh()
int wrefresh(win)
```

refersh ( )将把窗口映像拷贝到终端，而wrefresh (win)将把窗口映像拷贝到win，并使它看起来象原来的样子。

```
int wnoutrefresh(win)
int doupdate()
```

wnoutrefresh (win)将会只拷贝到窗口win，这意味着在终端上将不进行任何输出，但是虚拟屏幕实际上看起来象程序员所希望的那样。 doupdate ( )将输出到终端上。程序可以改变许多窗口，对每个窗口都调用一次wnoutrefresh (win)，然后再调用一次doupdate ( )来更新物理屏幕。

例如下面的程序，该程序中用到两个窗口，通过修改一些文本行可以改变这两个窗口。用户可以使用wrefresh (win)来编写函数changewin (win)。

(译者注：原文未提供例题)

这样做会让ncurses更新终端两次，会降低程序的执行速度。我们使用 doupdate ( )来改写函数changewin (win)，这样主函数性能会好一些。

```
int redrawwin(win)
int wredrawln(win, bline, nlines)
```

如果在往屏幕上输出新内容时需要清除一些行或者整个屏幕，可以使用这两个函数。 (可能这些行已经被破坏了或者由于其他的原因。)

```
int touchwin(win)
int touchline(win, start, count)
int wtouchln(win, y, n, changed)
int untouchwin(win)
```

这些函数通知ncurses整个win窗口已经被改动过了，或者从start直到start+count的这些行已经被改动过了。例如，如果用户有一些重叠的窗口（正如在example.c中一样），对某个窗口的改动不会影响其他窗口的映像。

wtouchln (…)将按掀从y开始的n行。如果change的值是TRUE，则这些行被按掀过了，否则就还未被按掀过(改变或未改变)。

untouchwin (win)将把窗口win标记为自上次调用refresh ( )以来还未被按掀。

```
int is_linetouched(win, line)
int is_wintouched(win)
```

通过使用这两个函数，用户可以检查自从上次调用 refresh ( )以来，第line行或者窗口win是否已被按掀过。

8.10 视频属性与颜色

属性是特殊的终端权能，当往屏幕上输出字符时需要用到它。字符可以打印成粗体、带下划线的、闪烁的等等。在 ncurses中，用户可以打开或关闭属性，以获得更佳的输出效果。下面列出了一些可能的属性。

表8-4 ncurses：属性

注意 出版者注：作者尚未提供此表。

ncurses定义了八种颜色，在带有彩色支持的终端上用户可以使用这些颜色。首先，调用start_color ( )初始化颜色数据结构，然后使用 has_colors ( )检查终端权能。start_color ( )将初始化COLORS和COLOR_PAIR。前者是终端所支持的最多的颜色数目，而后者是用户可以定义的色彩对的最大数目。

表8-5 ncurses：颜色

注意 出版者注：作者尚未提供此表。

这两个属性可以使用OR操作组合起来。“ COLORPAIRS_1 COLORS_1 ”

注意 出版者注：作者尚未提供代码。

```
int color_content(color, r, g, b)
```

此函数获取color的颜色成份r, g和b。

那么，如何把属性和颜色组合起来呢？有些终端，例如 Linux中的控制台，具有一些颜色，而其他终端却没有(xterm、VS100等等)。下面的代码可以解决这个问题：

注意 出版者说明：作者尚未提供源代码。

首先，函数CheckColor调用start_color ( )初始化颜色，如果当前终端有彩色的话，则函数has_colors ( )将返回TRUE。我们检查了这一点以后，调用 init_pair (…)把前景色和背景色组

合起来，再调用wattrset (…)为特定的窗口设置这些颜色对。此外，如果我们使用的是黑白终端，还可以单独使用wattrset (…)来设置属性。

如果要在xterm中获取颜色，我认为最佳方法是使用 ansi_xterm，以及来自Midnight Commander的terminfo项目。用户可以获取 ansi_xterm和Midnight Commander的源代码(mc_x.x.tar.gz)，然后编译ansi_xterm，并对mc_x.x.tar.gz文档中的xterm.ti和vt100.ti使用tic命令。执行ansi_xterm，把它试验出来。

8.11 光标和屏幕坐标

```
int move(y, x)
int wmove(win, y, x)
```

move ( )将移动光标，而wmove (win)则从窗口win中移动光标。对输入/输出函数来说，还定义了其他的一些宏，在调用特定函数之前，这些宏可以移动光标。

```
int curs_set(bf)
```

这个函数将把光标置为可见或者不可见，如果终端具有这个功能的话。

```
void getyx(win, y, x)
```

getyx (…)将返回当前光标位置。(注意：这是一个宏)

```
void getparyx(win, y, x)
```

如果win是个子窗口，getparyx (…)将把该窗口对应父窗口的坐标存储在y和x中，否则y和x都将为－1。(注：此函数尚未实现)

```
void getbegyx(win, y, x)
void getmaxyx(win, y, x)
int getmaxx(win)
int getmaxy(win)
```

这些函数把窗口win的开始坐标和大小坐标存放在y和x中。

```
int getsyx(int y, int x)
int setsyx(int y, int x)
```

getsyx (…)把虚拟屏幕光标存放在y和x中，而setsyx (…)则设置这个坐标。如果y和x是－1，用户调用getsyx (…)将会设置leaveok。

8.12 滚动

```
int scrollok(win, bf)
```

当光标在屏幕的右下角并且输入了一个字符 (或者新的一行)时，如果bf为TRUE，则窗口win中的文本将上滚一行。如果bf为FALSE，则鼠标留在原来的位置上。

当滚动特征打开时，使用下面的函数可以滚动窗口中的内容。 (注意：当用户在窗口的最后一行输入一个新行时，也应该发生相应的滚动操作，所以在使用 scrollok (…)时要十分小心，否则可能会得到出乎意料的结果。)

```
int scroll(win)
```

此函数将使窗口向上滚动一行(数据结构中的行也向上滚动)。

```
int scrl(n)
int wscrl(win, n)
```

这两个函数将使屏幕或者窗口 win向上向下滚动，滚动方向取决于整数 n的值。如果n是正数，则窗口向上滚动n行，否则如果n是负数，则窗口向下滚动 n行。

```
int setscrreg(t, b)
int wsetscrreg(win, t, b)
```

这两个函数设置一个软滚动区。

下面的代码将向读者说明怎样在屏幕上获得滚动文本的效果，也可以参见实例目录中的 .c文件。

注意 出版者注：作者尚未提供代码。

在程序中，我们有一个18行66列的窗口，我们希望在其中滚动文本。 S[ ]是存放文本的字符数组。 Max_s是s[ ]中最后一行的编号。 Clear_line将使用窗口的当前属性打印空白字符，从当前光标位置一直打印到本行的结束， (属性不同于clrtoeol所使用的A_NORMAL)。Beg是当前显示在屏幕上的 s[ ]文本的最后一行。 Scroll是个枚举类型，它告诉函数应该做些什么，并把文本的NEXT行或者PREV行显示出来。

8.13 小键盘

```
WINDOW *newpad(nlines, ncols)
WINDOW *subpad(orig, nlines, ncols, begy, begx)
int prefresh(pad, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol)
int pnoutrefresh(pad, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol)
int pechochar(pad, ch)
```

8.14 软标签

```
int slk_init(int fmt)
int slk_set(int labnum, char *label, int fmt)
int slk_refresh()
int slk_noutrefresh()
char *slk_label(int labnum)
int slk_clear()
int slk_restore()
int slk_touch()
int slk_attron(chtype attr)
int slk_attrset(chtype attr)
int slk_attroff(chtype attr)
```

这些函数是与attron (attr)、attrset (attr)和attroft (attr)相对应的，但它们尚未实现。

8.15 杂项

```
int beep()
int flash()
char *unctrl(chtype c)
char *keyname(int c)
int filter()
```

(注：以上函数尚未实现。)

```
void use_env(bf)
int putwin(WINDOW *win, FILE *filep)
```

(注：以上函数尚未实现。)

```
WINDOW *getwin(FILE *filep)
```

(注：此函数尚未实现。)

```
int delay_output(int ms)
int flushinp()
```

8.16 低级访问

```
int def_prog_mode()
int def_shell_mode()
int reset_prog_mode()
int reset_shell_mode()
int resetty()
int savetty()
int ripoffline(int line, int (*init)(WINDOW *, int))
int napms(int ms)
```

8.17 屏幕转储

```
int scr_dump(char *filename)
```

(注：此函数尚未实现。)

```
int scr_restore(char *filename)
```

(注：此函数尚未实现。)

```
int scr_init(char *filename)
```

(注：此函数尚未实现。)

```
int scr_set(char *filename)
```

(注：此函数尚未实现。)

8.18 Termcap模拟

```
int tgetent(char *bp, char *name)
int tgetflag(char id[2])
int tgetnum(char id[2])
char *tgetstr(char id[2], char **area)
char *tgoto(char *cap, int col, int row)
int tputs(char *str, int affcnt, int (*putc)())
```

8.19 Terminfo函数

```
int setupterm(char *term, int fildes, int *errret)
int setterm(char *term)
int set_curterm(TERMINAL *nterm)
int del_curterm(TERMINAL *oterm)
int restartterm(char *term, int fildes, int *errret)
```

(注：以上函数尚未实现。)

```
char *tparm(char *str, p1, p2, p3, p4, p5, p6, p7, p8, p9)
p1 - p9 long int.
int tputs(char *str, int affcnt, int (*putc)(char))
int putp(char *str)
int vidputs(chtype attr, int (*putc)(char))
int vidattr(chtype attr)
int mvcur(int oldrow, int oldcol, int newrow, int newcol)
int tigetflag(char *capname)
int tigetnum(char *capname)
int tigetstr(char *capname)
```

8.20 调试函数

- void _init_trace()
- void _tracef(char *, ...)
- char *_traceattr(mode)
- void traceon()
- void traceoff()

8.21 Terminfo权能

8.21.1 布尔型权能

变 量	权能名称	初始化	描 述
auto_left_margin	bw	bw	cub1从最后一列换行到第0列
auto_right_margin	am	am	终端的边界自动对齐
back_color_erase	bce	ut	屏幕以背景色清除
can_change	ccc	cc	终端可以重新定义现有的颜色
ceol_standout_glitch	xhp	xs	标准输出不会被覆盖所清除(hp)
col_addr_glitch	xhpa	YA	对hpa/mhpa大写字符而言只作正向移动
cpi_changes_res	cpix	YF	改变字符间距将会影响解析度
cr_cancels_micro_mode	crxm	YB	使用cr关闭宏模式
eat_newline_glitch	xenl	xn	在80列之后将忽略新行(Concept)
erase_overtrike	eo	eo	可以用空格来删除叠印
generic_type	gn	gn	通用行类型(如dialup, switch)
hard_copy	hc	hc	硬拷贝终端
hard_cursor	chts	HC	光标很难看到
has_meta_key	km	km	有一个元键(shift, 设置奇偶校验位)
has_print_wheel	daisy	YC	打印机需要操作员来改变字符集
has_status_line	hs	hs	有一个额外的“状态行”
hue_lightness_saturation	hls	hl	终端只使用HLS颜色表示法(Tektronix)
insert_null_glitch	in	in	插入模式, 能识别空行
lpi_changes_res	lpix	YG	改变行距将影响解析度
memory_above	da	da	显示可以保留在屏幕上方
memory_below	db	db	显示可以保留在屏幕下方

(续)

变 量	权能名称	初始化	描 述
move_insert_mode	mir	mi	在插入模式下可以安全地移动
move_standout_mode	msgr	ms	在标准输出模式下可以安全地移动
needs_xon_xoff	nxon	nx	不能填充，需要nxon/xoff
no_esc_ctl_c	xsb	xb	Beehive信号（F1=Escape，F2=Ctrl C）
non_rev_rmcup	nrrmc	NR	smcup不能反转rmcup
no_pad_char	npc	NP	填充字符不存在
non_dest_scroll_region	ndscr	ND	滚动区不可摧毁
over_strike	os	os	终端可以叠印
prtr_silent	mc5i	5i	打印机不向屏幕回送
row_addr_glitch	xvpa	YD	vhp/mvpa大写字母只能作正向移动
semi_auto_right_margin	sam	YE	打印在最后一列将导致cr
status_line_esc_ok	eslok	es	在状态行上可以使用Esc键
dest_tabs_magic_smso	xt	xt	制表符不可用(Teleray 1061)
tilde_glitch	hz	hz	Hazel_tine；不能打印 ' s
transparent_underline	ul	ul	下划线字符叠印
xon_coff	xon	xo	终端使用xon/xoff握手机制

8.21.2 数值型权能

变 量	权能名称	初始值	描 述
bit_image_entwining	bitwin	Yo	在SYSV中未作描写
buffer_capacity	bufsz	Ya	在打印前缓存的字节的数目
columns	cols	co	在一行中列的数目
dot_vert_spacing	spinv	Yb	在水平方向上点与点的距离，以每英寸多少点为单位
dot_horz_spacing	spinh	Yc	在垂直方向上针之间的距离，以每英寸多少针为单位
init_tabs	it	it	每#个空格算一个制表符的位置
label_height	lh	lh	每个标签多少行
label_width	lw	lw	每个标签多少列
lines	lines	li	屏幕或页面上行的数目
lines_of_memory	lm	lm	如果>lines则表示内存中的行数，0意味着可变
magic_cookie_glitch	xmc	sg	smso或rmso所剩下的空白字符的数目
max_colors	colors	Co	在屏幕上颜色的最大数目
max_micro_address	maddr	Yd	在micro_…_address中的最大值
max_micro_jump	mjump	Ye	在parm_…_micro中的最大值
max_pairs	pairs	pa	在屏幕上颜色对的最大数目
micro_col_size	mcs	Yf	在宏模式中字符间距的大小
micro_line_size	mls	Yg	在宏模式中行距的大小
no_color_video	ncv	NC	不能使用彩色的视频属性
number_of_pins	npins	Yh	在打印头中针的数目
num_labels	nlab	Nl	屏幕上标签的数量
output_res_char	orc	Yi	水平解析度，以每行单元数为单位
output_res_line	orl	Yj	垂直解析度，以每行单元数为单位
output_res_horz_inch	orhi	Yk	水平解度，以每英寸单元数为单位
output_res_vert_inch	orvi	Yl	垂直解析度，以每英寸单元数为单位
padding_baud_rate	pb	pb	在需要cr/nl填充时最低的波特率
virtual_terminal	vt	vt	虚拟终端号(Unix系统)
width_status_line	wsl	ws	状态行的第n列

(下面的数值型权能是在 SYSV term结构中定义的，但在 man帮助中还没有提供对它们的描述。我们的解释来自term结构的头文件。)

(续)

变 量	权能名称	初始值	描 述
bit_image_type	bitype	Yp	位映像设备的类型
buttons	btns	BT	鼠标按键的数目
max_attributes	ma	ma	终端能够处理的最多的组合属性
maximum_windows	wnum	MW	可定义窗口的最大数目
print_rate	cps	Ym	打印速率，以每秒字符数为单位
wide_char_size	widcs	Yn	在双宽度模式中字符间距的大小

8.21.3 字符串型权能

变 量	权能名称	初始值	描 述
acs_chars	acsc	ac	图形字符集对 一def = vt100
alt_scancode_esc	scesa	S8	扫描码模拟的另一种换码(默认值是VT100)
back_tab	cbt	bt	向后tab (p)
bell	bel	bl	声音信号(响铃)(p)
bit_image_repeat	birep	Xy	把位映像单元重复＃1 ＃2次(使用tparm)
bit_image_newline	binel	Zz	移动到位映像的下一行(使用tparm)
bit_image_carriage_return	bicr	Yv	移动到同一行的开头(使用tparm)
carriage_return	cr	cr	回车(p*)
change_char_pitch	cpi	ZA	改变为每英寸＃个字符
change_line_pitch	lpi	ZB	改变为每英寸＃行
change_res_horz	chr	ZC	改变水平解度
change_res_vert	cvr	ZD	改变垂直解析度
change_scroll_region	csr	cs	把滚动区改变为从＃1行到＃2行(VT100)(PG)
char_padding	rmp	rP	与ip相似，但它用在插入模式中
char_set_names	csnm	Zy	字符集名称的列表
clear_all_tabs	tbc	ct	清除所有的制表符停止(p)
clear_margins	mgc	MC	清除所有的页边
clear_screen	clear	cl	清除屏幕和home光标(p*)
clr_bol	el1	cb	清除到行首
clr_eol	el	ce	清除到行尾(P)
clr_eos	ed	cd	清除到显示的末尾(p*)
code_set_init	csin	ci	多个代码集合的初始化序列
color_names	colornm	Yw	＃1号颜色的名称
column_address	hpa	ch	设置光标列(PG)
command_character	cmdch	CC	在原型中终端可以设置的cmd字符
cursor_address	cup	cm	屏幕光标移动到＃1行＃2列(PG)
cursor_down	cud1	do	下移一行
cursor_home	home	ho	Home光标(如果没有环的话)
cursor_invisible	civis	vi	使光标不可见
cursor_left	cub1	le	把光标向左移一个空格
cursor_mem_address	mrcup	CM	内存相对的光标寻址
cursor_normal	cnorm	ve	使光标以最普通的外形显示(undo vs/vi)
cursor_right	cuf1	nd	不具有破坏性的空白(光标向右移)
cursor_to_ll	ll	ll	最后一行，第一列(如果没有环的话)
cursor_up	cuu1	up	Upline(光标向上移)

(续)

变量	权能名称	初始值	描述
cursor_visible	cvvis	vs	使光标可见
define_bit_image_region	defbi	Yx	定义方形的位映像区(使用tparm)
define_char	defc	ZE	定义字符集中的某个字符
delete_character	dch1	dc	删除字符(p*)
delete_line	dl1	dl	删除行(p*)
device_type	devt	dv	显示语言/代码集支持
dis_status_line	dsl	ds	关闭状态行
display_pc_char	dispc	S1	显示PC字符
down_half_line	hd	hd	向下移动半行(向前换1/2行)
ena_acs	enacs	eA	使能另一个字符集合
end_bit_image_region	endbi	Yy	结束位映像区(使用tparm)
enter_alt_charset_mode	smacs	as	开始另一个字符集(p)
enter_am_mode	smam	SA	打开自动对齐特征
enter_blink_modeblink	mb		打开字符闪烁效果
enter_bold_mode	bold	md	打开粗体(特别亮)模式
enter_ca_mode	smcup	ti	启动使用环的程序的字符串
enter_delete_mode	smdc	dm	删除模式(输入)
enter_dim_mode	dim	mh	打开半亮模式
enter_doublewide_mode	swidm	ZF	使能双倍宽度模式
enter_draft_quality	sdrfq	ZG	设置草图效果的打印方式
enter_insert_mode	smir	im	插入模式(输入)
enter_italics_mode	sitm	ZH	使能斜体字模式
enter_leftward_mode	slm	ZI	使能向左回车移动
enter_micro_mode	smicm	ZJ	使能宏移动功能
enter_near_letter_quality	snlq	ZK	设置NLQ打印
enter_normal_quality	snrmq	ZL	设置一般质量的打印方式
enter_pc_charset_mode	smpch	S2	输入PC字符显示模式
enter_protected_mode	prot	mp	打开保护模式
enter_reverse_mode	rev	mr	打开反转视频模式
enter_scancode_mode	smsc	S4	输入PC扫描码
enter_secure_mode	invis	mk	打开空白模式(字符不可见)
enter_shadow_mode	sshm	ZM	使能阴影打印模式
enter_standout_mode	smso	so	开始标准输出模式
enter_subscript_mode	ssubm	ZN	使能下标打印
enter_superscript_mode	ssupm	ZO	使能上标打印
enter_underline_mode	smul	us	开始下划线模式
enter_upward_mode	sum	ZP	使能向上回车移动
enter_xon_mode	smxon	SX	打开xon/xoff握手机制
erase_chars	ech	ec	删除#1个字符(PG)
exit_alt_charset_mode	rmacs	ae	终止可选的字符集(P)
exit_am_mode	rmam	RA	关闭自动对齐方式
exit_attribute_mode	sgr0	me	关闭所有属性
exit_ca_mode	rmcup	te	终止使用环的程序的字符串
exit_delete_mode	rmdc	ed	终止删除模式
exit_doublewide_mode	rwidm	ZQ	关闭双倍宽度打印方式
exit_insert_mode	rmir	ei	结束插入模式
exit_italics_mode	ritm	ZR	关闭斜体打印模式

(续)

变 量	权能名称	初始值	描 述
exit_leftward_mode	rlm	ZS	使能右向(普通的)回车移动
exit_micro_mode	rmicm	ZT	关闭宏移动能力
exit_pc_charset_mode	rmpch	S3	关闭PC字符显示
exit_scancode_mode	rmsc	S5	关闭PC扫描码模式
exit_shadow_mode	rshm	ZU	关闭阴影打印模式
exit_standout_mode	rmso	se	结束标准输出模式
exit_subscript_mode	rsubm	ZV	关掉下标打印方式
exit_superscript_mode	rsupm	ZW	关掉上标打印方式
exit_underline_mode	rmul	ue	结束下划线模式
exit_upward_mode	rum	ZX	打开向下(普通的)回车移动
exit_xon_mode	rmxon	RX	关掉xon/xoff握手机制
flash_screen	flash	vb	可视响铃(不能移动光标)
form_feed	ff	ff	硬拷贝终端页面的换页(p*)
from_status_line	fsl	fs	从状态行返回
init_1string	is1	i1	终端初始化字符串
init_2string	is2	i2	终端初始化字符串
init_3string	is3	i3	终端初始化字符串
init_file	if	if	所包含的文件名称
init_prog	iprog	iP	初始化程序的路径名
initialize_color	initc	Ic	初始化颜色的定义
initialize_pair	initp	Ip	初始化颜色对
insert_character	ich1	ic	插入字符(P)
insert_line	il1	al	加入一个新的空白行(p*)
insert_padding	ip	ip	在插入的字符之后再插入填充字符(p*)
key_a1	ka1	K1	小键盘左上方的键
key_a3	ka3	K3	小键盘右上方的键
key_b2	kb2	K2	小键盘中央的键
key_backspace	kbs	kb	由回退键所发送
key_beg	kbeg	1	开始键
key_btab	kcbt	kB	向右一tab键
key_c1	kc1	K4	小键盘左下角的键
key_c3	kc3	K5	小键盘右下角的键
key_cancel	kcan	2	取消键
key_catab	ktbc	ka	由clear_all_tabs键发送
key_clear	kclr	kC	由清除屏幕或者删除键发送
key_close	kclo	3	关闭键
key_command	kcmd	4	命令键
key_copy	kcpy	5	拷贝键
key_create	kcrt	6	创建键
key_ctab	kctab	kt	由clear_tab键发送
key_dc	kdch1	kD	由删除字符键发送
key_dl	kdl1	kL	由删除行键发送
key_down	kcud1	kd	由终端向下光标键发送
key_eic	krmir	kM	在插入模式中由rmir或smir发送
key_end	kend	7	结束键
key_enter	kent	8	输入/发送键
key_eol	kel	kE	由clear_to_end_of_line键发送

(续)

变 量	权能名称	初始值	描 述
key_eos	ked	kS	由clear_to_end_of_screen键发送
key_exit	kext	9	退出键
key_find	kfnd	0	查找键
key_help	khlp	%1	帮助键
key_home	khome	kh	由home键发送
key_ic	kich1	kl	由ins char/enter ins mode键发送
key_il	kil1	kA	由插入行发送
key_left	kcub1	kl	由终端向左键发送
key_ll	kll	kH	由home_down键发送
key_mark	kmrk	%2	标记键
key_message	kmsg	%3	消息键
key_move	kmov	%4	移动键
key_next	knxt	%5	下一个键
key_npage	knp	kN	由下页键发送
key_open	kopn	%6	打开键
key_options	kopt	%7	选项键
key_ppage	kpp	kP	由前页键发送
key_previous	kprv	%8	前一键
key_print	kprt	%9	打印键
key_redo	krdo	%0	redo键
key_refrence	kref	&1	引用键
key_refresh	krfr	&2	刷新键
key_replace	krpl	&3	替换键
key_restart	krst	&4	重启键
key_resume	kres	&5	恢复键
key_right	kcuf1	kr	由终端向右键发送
key_save	ksav	&6	保存键
key_sbeg	kBEG	&9	按下开始键的同时按下shift键
key_scancel	kCAN	&0	按下取消键的同时按下shift键
key_scommand	kCMD	*1	按下命令键的同时按下shift键
key_scopy	kCPY	*2	按下拷贝键的同时按下shift键
key_screate	kCRT	*3	按下创建键的同时按下shift键
key_sdc	kDC	*4	按下删除字符键的同时按下shift键
key_sdl	kDL	*5	按下删除行键的同时按下shift键
key_select	kslt	*6	选择键
key_send	kEND	*7	按下结束键的同时按下shift键
key_seol	kEOL	*8	按下行尾键的同时按下shift键
key_sexit	kEXT	*9	按下退出键的同时按下shift键
key_sf	kind	kF	由前滚/下滚键发送
key_sfind	kFND	*0	按下查找键的同时按下shift键
key_shelp	kHLP	#1	按下帮助键的同时按下shift键
key_shome	kHOM	#2	按下Home键的同时按下shift键
key_sic	kIC	#3	按下插入字符键的同时按下shift键
key_sleft	kLFT	#4	按下向左键的同时按下shift键
key_smessage	kMSG	%a	按下消息键的同时按下shift键
key_smove	kMOV	%b	按下移动键的同时按下shift键
key_snext	kNXT	%c	按下向后键的同时按下shift键

(续)

变 量	权能名称	初始值	描 述
key_soptions	kOPT	%d	按下选项键的同时按下shift键
key_sprevious	kPRV	%e	按下向前键的同时按下shift键
key_sprint	kPRT	%f	按下打印键的同时按下shift键
key_sr	kri	kR	由后滚/下滚键发送
key_sredo	kRDO	%g	按下redo键的同时按下shift键
key_sreplace	kRPL	%h	按下替换键的同时按下shift键
key_sright	kRIT	%l	按下向右键的同时按下shift键
key_srsume	kRES	%j	按下恢复键的同时按下shift键
key_ssave	kSAV	!1	按下保存键的同时按下shift键
key_ssuspend	kSPD	!2	按下中断键的同时按下shift键
key_sundo	kUND	!3	按下取消键的同时按下shift键
key_stab	khts	kT	由set_tab键发送
key_suspend	kspd	&7	中断键
key_undo	kund	&8	取消键
key_up	kcuul	ku	由终端的向上键发送
keypad_local	rmkx	ke	不处于“小键盘发送”方式之中
keypad_xmit	smkx	ks	把终端置为“小键盘发送”方式
lab_f0	lf0	l0	如果不是f0的话，则为功能键f0的标签
lab_f1	lf1	l1	如果不是f1的话，则为功能键f1的标签
lab_f2	lf2	l2	如果不是f2的话，则为功能键f2的标签
lab_f3	lf3	l3	如果不是f3的话，则为功能键f3的标签
lab_f4	lf4	l4	如果不是f4的话，则为功能键f4的标签
lab_f5	lf5	l5	如果不是f5的话，则为功能键f5的标签
lab_f6	lf6	l6	如果不是f5的话，则为功能键f6的标签
lab_f7	lf7	l7	如果不是f7的话，则为功能键f7的标签
lab_f8	lf8	l8	如果不是f8的话，则为功能键f8的标签
lab_f9	lf9	l9	如果不是f9的话，则为功能键f9的标签
lab_f10	lf10	la	如果不是f10的话，则为功能键f10的标签
label_on	smln	LO	打开软标签
label_off	rmln	LF	关闭软标签
meta_off	rmm	mo	关闭“元模式”
meta_on	smm	mm	打开“元模式”(8位)
micro_column_address	mhpa	ZY	近似宏调整的列—地址,
micro_down	mcud1	ZZ	近似宏调整的光标—向下
micro_left	mcutb1	Za	近似宏调整的光标—向左
micro_right	mcuf1	Zb	近似宏调整的光标—向右
micro_row_address	mvpa	Zc	近似宏调整的行—地址
micro_up	mcuu1	Zd	近似宏调整的光标—向上
newline	nel	nw	新行(行为近似于cr后跟lf)
order_of_pins	porder	Ze	匹配软件以及打印头中的针
orig_colors	oc	oc	重置所有的颜色对
orig_pair	op	op	把默认的颜色对设置为原始的那个
pad_char	pad	pc	填充字符(非空)
parm_dch	dch	DC	删除#1字符(PG*)
parm_delete_line	dl	DL	删除#1行(PG*)
parm_down_cursor	cud	DO	把光标向下移#1行(PG*)
parm_down_micro	mcud	Zf	近似宏调用的cub

(续)

变 量	权能名称	初始值	描 述
parm_ich	ich	IC	插入＃1个空白符号(PG*)
parm_index	indn	SF	向上滚动＃1行(PG)
parm_insert_line	il	AL	加入＃1个新的空白行(PG*)
parm_left_cursor	cub	LE	把光标向左移＃1个空格(PG)
parm_left_micro	mcub	Zg	近似宏调整中的cub
parm_right_cursor	cuf	Rl	把光标向右移＃1个空格(PG*)
parm_right_micro	mcuf	Zh	近似宏调整中的cuf
parm_rindex	rin	SR	回滚＃1行(PG)
parm_up_cursor	cuu	UP	把光标上移＃1行(PG*)
parm_up_micro	mcuu	Zi	近似宏调整中的cuu
pkey_key	pfkey	pk	把功能键＃1定义为字符＃2的类型
pkey_local	pfloc	pl	把功能键＃1定义为执行字符串＃2
pkey_xmit	pfx	px	把功能键＃1定义为发送字符串＃2
pkey_plab	pfxl	xl	把功能键＃1定义为发送＃2，并显示＃3
plab_norm	pln	pn	编程标签＃1，以显示字符串＃2
print_screen	mc0	ps	打印屏幕内容
prtr_non	mc5p	pO	打开打印机，打印＃1个字节
prtr_off	mc4	pf	关闭打印机
prtr_on	mc5	po	打开打印机
repeat_char	rep	rp	把字符＃1重复＃2次(PG*)
req_for_input	rfi	RF	输入请求
reset_1string	rs1	r1	把终端完全置为sane方式
reset_2string	rs2	r2	把终端完全置为sane方式
reset_3string	rs3	r3	把终端完全置为sane方式
reset_file	rf	rf	包含重置字符串的文件名称
restore_cursor	rc	rc	把光标置为上一个屏幕上的位置
row_address	vpa	cv	垂直绝对位置(设置行)(PG)
save_cursor	sc	sc	保存光标位置(P)
scancode_escape	scesc	S7	为了扫描码模拟按下Esc键
scroll_forward	ind	sf	把文本向上滚动(P)
scroll_reverse	ri	sr	把文本向下滚动(P)
select_char_set	scs	Zj	选择字符集
set0_des_seq	s0ds	s0	切换到代码集0(EUC集0，ASCII)
set1_des_seq	s1ds	s1	切换到代码集1
set2_des_seq	s2ds	s2	切换到代码集2
set3_des_seq	s3ds	s3	切换到代码集3
set_a_background	setab	AB	使用ANSI设置背景颜色
set_a_foreground	setaf	AF	使用ANSI设置前景颜色
set_attributes	sgr	sa	定义视频属性(PG9)
set_background	setb	Sb	设置当前背景颜色
set_bottom_margin	smgb	Zk	设置当前行的底部边界
set_bottom_margin_parm	smgbp	ZI	从bottomset_color_band的＃1行或＃2行设置底行
setcolor	Yz		改变＃1号色带颜色
set_color_pair	scp	sp	设置当前颜色对
set_foreground	setf	Sf	设置当前前景色
set_left_margin	smgl	ML	设置当前行的左边界
set_left_margin_parm	smglp	Zm	在＃1行(＃2行)设置左(右)边界

(续)

变 量	权能名称	初始值	描 述
set_lr_margin	smglr	ML	设置左右边界
set_page_length	slines	YZ	把页的长度设置为#1行(使用tparm)
set_right_margin	smgr	MR	把右边界设置为当前列
set_right_margin_parm	smgrp	Zn	把右边界设置为#1列
set_tab	hts	st	在当前列的所有行设置制表符
set_tb_margin	smgtb	MT	设置上下边界
set_top_margin	smgt	Zo	把上边界设置为当前行
set_top_margin_parm	smgtp	Zp	把上边界设置为#1行
set_window	wind	wi	当前窗口是从#1行到#2行，从#3列到#4列
start_bit_image	sbim	Zq	开始打印位映像图形
start_char_set_def	scsd	Zr	开始定义字符集
stop_bit_image	rbim	Zs	结束打印位映像图形
stop_char_set_def	rcsd	Zt	结束定义字符集
subscript_characters	subcs	Zu	下标字符的列表
superscript_characters	supcs	Zv	上标字符的列表
tab	ht	ta	跳转到下面8个空格硬件的制表符位置
these_cause_cr	docr	Zw	这些字符导致CR
to_status_line	tsl	ts	跳到状态行，第1列
underline_char	uc	uc	给某字符划下划线，并移过它
up_half_line	hu	hu	上移半行(反转1/2行)
xoff_character	coffc	XF	XON字符
xon_character	xonc	XN	XOFF字符

(下面的字符串权能是在SYSVr终端结构中定义的，但在man帮助信息中还未作描述，对它们的解释是从终端结构头文件中得到的。)

label_format	fln	Lf	??
set_clock	sclk	SC	设置时钟
display_clock	dclk	DK	显示时钟
remove_clock	rmclk	RC	删除时钟
create_window	cwin	CW	把窗口#1定义为从#2行,#3列到#4行，#5列
goto_window	wingo	WG	跳到窗口#1
hangup	hup	HU	挂起电话
dial_phone	dial	DI	拨电话号码#1
quick_dial	qdial	QD	拨电话号码#1，但不做进度检查
tone	tone	TO	选择接触声调拨叫
pulse	pulse	PU	选择脉冲拨叫
flash_hook	hook	fh	闪光切换分支
fixed_pause	pause	PA	暂停2～3秒
wait_tone	wait	WA	等待拨叫声音
user0	u0	u0	用户字符串#0
user1	u1	u1	用户字符串#1
user2	u2	u2	用户字符串#2
user3	u3	u3	用户字符串#3
user4	u4	u4	用户字符串#4
user5	u5	u5	用户字符串#5
user6	u6	u6	用户字符串#6

(续)

变 量	权能名称	初始值	描 述
user7	u7	u7	用户字符串 #7
user8	u8	u8	用户字符串 #8
user9	u9	u9	用户字符串 #9
get_mouse	getm	Gm	surses应获得按钮事件
key_mouse	kmous	Km	??
mouse_info	minfo	Mi	鼠标状态信息
pc_term_options	pctrm	S6	PC终端选项
req_mouse_pos	reqmp	RQ	请求鼠标位置报告
zero_motion	zerom	Zx	后继字符没有移动

8.22 [N]Curses函数概述

下面介绍不同的(n)curses服务包。第一列是bsd_curses(如Slackware 2.1.0中，与Sun OS 4.x中一样)，第二列是sysv_curses(在Sun OS 5.4/Solaris 2中)，第三列是ncurses(版本1.8.6)。第四列的描述是摘自描述该函数的文本(如果有对该函数的描述的话)。

注意 出版者注：作者未提供此表。

x
服务包中有这个函数。
n
此函数尚未实现。
(尚待完成)

第9章　I/O端口编程

通常一台pc机至少有两个串行接口和一个并行接口。这些接口是特殊的设备，它们以如下方式映射：

- 它们是RS232串行设备0～n，这里n的大小取决于用户的硬件。
- 它们是并行设备0～n，这里n的值取决于用户的硬件。
- 它们是游戏杆设备0～n。

/dev/ttys*和/dev/cua*设备之间的差异之处在于怎样处理 open( )函数的调用。/dev/cua*设备常用作调用设备，通过调用 open( )可以获得其他的默认设置；而 /dev/ttys*设备则将为到达的调用以及发出的调用而初始化。

注意　出版者注：/dev/cua*设备现在正逐渐淘汰。

在默认情况下，设备通常由那些打开该设备的进程来控制。通常 ioctl()请求可以处理所有这些特殊的设备，但POSIX更喜欢定义新函数，以便根据结构 termios的不同来处理异步终端。这两种方法都需要包含〈termios.h〉。

1) ioctl 方法：TCSBRK、TCSBRKP、TCGETA(获得属性)、TCSETA(设置属性)、终端I/O控制(TIOC)请求：TIOCGSOFTCAR(设置软回车)、TIOCSSOFTCAR(获得软回车)、TIOCSCTTY(设置控制 tty)、TIOCMGET(获得modemline)、TIOCMSET(设置modemline)、TIOCGSERIAL、TIOCSSERIAL、TIOCSERCONFIG、TIOCSERGWILD、TIOCSERSWILD、TIOCSERGSTRUCT、TIOCMBIS、TIOCMBIC、……

2) POSIX方法：tcgetattr( )、tcsetattr( )、tcsendbreak( )、tcdrain( )、tcflush( )、tcflow( )、tcgetpgrp( )、tcsetpgrp( )、cfsetispeed( )、cfgetispeed( )、cfsetospeed( )、cfgetospeed( )

3) 其他方法：对硬件使用 outb和inb，就好像在编程时在没在打印机的情况下使用找印机端口。

9.1　鼠标编程

鼠标或者与串行口相连接，或者直接连接到 AT总线上。不同的鼠标所发送的数据也不同，这使得鼠标编程比较难实现。但是，　Andrew Haylett是个好人，他提供了自己程序集的版权，这意味着用户可以在自己的过程中使用他的鼠标过程。在这部分的内容中，读者可以读到带有版权声明的程序集1.8的预发行版本。x11早已提供了一个优秀的鼠标 API，所以Andrew的过程只运用于非x11的应用程序。

在该程序集服务包中，用户可以使用 mouse.c和mouse.h这两个模块。为了获取鼠标事件，用户只需调用ms_init( )和get_ms_event( )这两个函数。ms_init需要下面这10个变元：

1) int acceleration　它是一个加速因子。如果用户移动鼠标的距离超过 delta个像素，则鼠标移动的速度将取决于这个值的大小。

2) int baud　它是用户鼠标所使用的bps速率(通常为1,200)。

3) int delta 它是像素的数量。鼠标的移距离必须超过这个值，才能启运加速因子进行加速。

4) char *device 它是鼠标设备的名称 (例如，/dev/mouse)。

5) int toggle 在初始化时用于切换DTR或RTS，或者同时切换这两种鼠标modem线。

6) int sample 鼠标的灵敏度(dpi)(通常为100)。

7) mouse_type mouse 鼠标的标识符，笔者的鼠标的标识符是 P_MSC(Mouse Syctems公司)。

8) int slack 它是边框处slack的数量，如果slack为-1，则当鼠标光标在屏幕边框处时，用户再移动鼠标，鼠标将停在边框上；如果 slack的值大于等于0，则当鼠标光标在边框处，用户再把它移动slack个像素位置以后，鼠标光标将跳转到另一端。

9) int maxx 用户当前终端在x方向上的解析度。在使用默认字体的情况下，字符宽度为10个像素，所以整个屏幕x方向的解析度是10*80-1。

10) int maxy 用户终端在y方向上的解析度，使用默认的字体，一个字符是 12个像素高。所以整个屏幕y方向的解析度是12*25-1。

get_ms_event( )只需要一个参数，即指向结构 ms_event的指针。如果get_ms_event( )返回－1，则函数将出错。在成功时它将返回 0，而结构ms_event中将包含实际的鼠标状态。

9.2 调制解调器编程

请参见实例miniterm.c。

使用termios来控制rs232端口。

使用Hayes命令来控制调制解调器。

9.3 打印机编程

请参见实例checklp.c。

不要使用termios来控制打印机端口。如果需要时要使用 ioctl和inb/outb。

使用Epson、Postscript、PCL等命令来控制打印机。

〈linux/lp.h〉

ioctl调用：LPCHAR、LPTIME 、LPABORT、LPSETIRQ、LPGETIRQ、LPWAIT

用于状态和控制端口的inb/out

9.4 游戏杆编程

请参见游戏杆可装入内核模块服务器包中的实例 js.c。

〈linux/joystick.h〉

ioctl调用：JS_SET_CAL、JS_GET_CAL、JS_SET_TIMEOUT、JS_GET_TIMEPOUT、JS_SET_TIMELIMIT、JS_GET_TIMELIMIT、JS_GET_ALL、JS_SET_ALL。在/dev/jsn上执行一次读操作将返回结构JS_DATA_TYPE。

第10章　把应用程序移植到Linux上

10.1　介绍

把Unix应用程序移植到Linux上是相当容易的。Linux以及它所使用的GNU C库，在设计时就已经充分地考虑到了应用程序的可移植性。这就意味着许多应用程都可以简单地使用make进行编译，只要它们没有利用特定实现的一些不知名特征，并且没有紧密地依赖于未定义或者未写入文档的行为(例如某个特定的系统调用)。

Linux基本上服从IEEE Std 1003.1-1988(POSIX.1)标准，但也并不保证一定都是这样。与此相似，Linux也支持和实现了许多SVIP和BSD Unix中的特征，但是并不是在所有情况下都支持这些特征，一般来说，Linux在设计时已经被设计成与其他Unix实现相兼容，可以让移植应用程序更容易，在许多情况下它改进或纠正了这些实现中的某些行为。

作为一个例子，在poll操作中，Linux会真正减少传送给select系统调用的timeout变元。而其他某些实现可能根本不修改该值。而如果不按照这个规定，应用程序在Linux下编译时可能会崩溃。BSD和SunOS select系统调用提供的man帮助信息说：在“将来的实现中”该系统调用会修改timeout指针。不幸的是，许多应用程序仍然假设那个值不会改变。

本章的目标就是概述有关把应用程序移植到Linux上的问题，重点是介绍Linux、POSIX.1、SVID和BSD在以下方面存在的差异：信号处理、终端I/O、进程控制和消息集成，以及可移植条件编译。

10.2　信号处理

近年来，Unix的不同实现所给出的信号的定义也不相同，信号的语义也有所不同。当前主要有两类信号：可靠的和不可靠的。不可靠信号是指它们的信号处理程序调用以后并不保留安装。如果程序希望此信号保留安装，则这种“单发”信号必须在信号处理程序内部重新安装信号处理程序。正因为如此，我们假设在处理程序重要新安装以前该信号再次到达，这时就构成了一个竞争条件。它的后果是要么丢失了信号，要么激活该信号的原始行为(例如杀死进程)。所以，这些信号是“不可靠的”，因为信号的获取操作和重新安装操作不是原子操作。

在不可靠信号语义下，当被信号中断时，系统调用并不自动重启。所以，为了让程序能处理所有情况，在每一次系统调用以后，程序都需要检查errno的值，如果值为EINTR则重新发出系统调用。

同理可知，不可靠信号语义并不提供获得原子中断操作的简单方法(中断操作是指把进程置为睡眠状态，直到某信号到达为止)。因为重新安装的信号处理程序是不可靠的，所以有些时候信号到达时程序却无动于衷。

另一方面，在可靠信号语义下，信号处理程序在调用时就保留安装，这时重新安装所带

来的竞争条件就避免了。而且，特定的系统调用可以重启，通过 POSIX的sigsuspend函数还可以提供原子中断操作。

10.2.1 SVR4、BSD和POSIX.1下的信号

SCV4所实现的信号提供以下函数：signal、sigset、sighold、sigrelse、sgignore和sigpause。SVR4下的signal函数与典型的Unix V7下的信号相同，只提供不可靠信号。另外的函数确实提供了信号处理程序和自动重新安装，但不支持系统调用的重新启动。

BSD支持函数signal、sigvec、sigblock、sigsetmask和sigpause。所有这些函数均提供可靠信号，系统调用是默认重启的。但如果程序员愿意，他可以关闭这一特征。

POSIX.1提供函数sigaction、sigprocmask、sigpending和sigsuspend。注意这里没有signal函数，因为根据POSIX.1标准，它的价值不大。这些函数提供了可靠信号，但 POSIX没有定义系统调用的重启行为。如果在 SVR4和BSD下使用sigaction，则系统调用的重启在默认情况下是关闭的。但如果指定了信号标志 SA_RESTART，则该特征也可以打开。

所以，在程序中使用信号的最佳方法是使用函数 sigaction，它将允许程序员显式地指定信号处理程序的行为。然而，在许多应用程序中仍然使用 signal函数，而读者可以看到，上面所列出的signal在SVR4和BSD下语义是不同的。

10.2.2 Linux信号选项

在 Linux下，sigaction结构的sa_flags成员定义了如下一些值：

- SA_NOCLDSTOP：当子进程停止执行时，无需发送 SIGCHLD信号。
- SA_RESTART：当被某信号处理程序中断时，强迫特定的系统调用重新启动。
- SA_NOMASK：关闭信号掩码(在信号处理程序执行时它会阻塞信号)。
- SA_ONESHOT：在信号处理程序执行以后清除该处理程序。注意 SVR4使用SA_RESETHAND表示同一个操作。
- SA_INTERRUPT：在Linux下定义，但未使用。在 SunOS下，系统调用是自动重启的，而该标志可以关闭那个行为。
- SA_STACK：当前它被用于信号栈操作。

注意POSIX.1只定义了SA_NOCLDSTOP，而SVR4下定义的许多其他选项在 Linux下均不可用。在移植使用 sigaction的应用程序时，用户可能需要修改 sa_flags的值，以便得到正确的行为。

10.2.3 Linux下的信号

在Linux下，signal函数等价于使用SA_ONESHOT和SA_NOMASK选项来调用sigaction。也就是说，它对应着SVR4所使用的典型的不可靠信号语义。

如果用户想让信号使用 BSD语义，大部分Linux系统都提供了一个与 BSD兼容的库，可以与它链接。为了使用该库，用户需要在编译命令行中加入如下选项：

```
-I/usr/include/bsd -lbsd
```

当移植的应用程序使用信号时，请密切关注程序对于信号处理程序的使用作出什么假设，并修改代码(以正确的定义进行编译)，以获得正确的行为。

10.2.4 Linux支持的信号

Linux几乎支持SVR4、BSD和POSIX所提供的所有信号，但以下几个信号Linux不支持：

- SIGEMT不被支持，在SVR4和BSD下它对应着一个硬件错误。
- SIGINFO不被支持，SVR4下它可以用来处理键盘信息请求。
- SIGSYS不被支持。在SVR4和BSD中它指的是非法系统调用。如果用户与libbsd相链接，该信号被重新定义为SIGUNUSED。
- SIGABRT和SIGIOT相同。
- SIGIO、SIGPOLL和SIGURG相同。
- SIGBUS被定义为SIGUNUSED。从技术上讲，在Linux环境中不存在“总线错误”。

10.3 终端I/O

与信号一样，终端I/O控制在SVR4、BSD和POSIX.1下的实现各不相同。

SVR4使用termio结构，以及终端设备上的几个ioctl调用(如TCSETA、TCGETA等等)，以获取和设置termio结构的成员值。该结构定义如下：

```
struct termio {
  unsigned short c_iflag;  /* Input modes */
  unsigned short c_oflag;  /* Output modes */
  unsigned short c_cflag;  /* Control modes */
  unsigned short c_lflag;  /* Line discipline modes */
  char c_line;             /* Line discipline */
  unsigned char c_cc[NCC]; /* Control characters */
};
```

在BSD下，sgtty结构可用于几个ioctl调用中，如TIOCGETP、TIOCSETP等等。

在POSIX下，termios结构是与POSIX.1所定义的几个函数一起使用的。(如tcsetattr和tcgetattr等)。termios结构与SVR4所使用的结构termios相同，但是它们的类型不同(POSIX使用的类型为tcflag_t，而不再是unsigned short)，而NCCS则用于c_cc数组的大小。

在Linux下，POSIX.1 termios和SVR4 termio都直接受内核支持。这意味着如果用户使用这两种方法之一来访问终端I/O，它应该直接在Linux下编译。如果读者有什么疑问，可以很容易地把使用termio的代码修改为使用termios，这种修改只需对这两种方法稍有了解即可。然而用户永远不会需要作这种修改。但是，如果程序要在termio结构中使用c_line，用户一定要小心。对于几乎所有的应用程序来说，它应该是N_TTY，如果程序假定其他一些行规则也可用，则程序员将陷入困境。

如果用户的程序使用BSD sgtty实现，用户可以像上面介绍的那样与libbsd.a进行链接。这将提供ioctl的一个替代品，它将以内核使用的POSIX termios调用重新提交终端I/O请求。在编译这样的程序时，如果没有定义TIOCGETP等符号，则用户需要链接libbsd。

10.4 进程信息和控制

系统必须向程序(如ps、top和free)提供一些方法，以便它们从内核获取有关进程和系统资源的信息。类似地，调试器和其他类似的工具也需要有控制和监察运行进程的能力。不同版

本的Uinx通过许多接口提供了这些特征，几乎所有这些特征都是与计算机相关的，或者紧密地和特定内核设计联系在一起的。到目前为止，对于这种进程与内核之间的交互，还没有一种广泛接受的接口。

10.4.1 kvm过程

许多系统使用kvm_open、kvm_nlist和kvm_read等过程来直接访问内核数据结构，这些访问是通过/dev/ kmem设备进行的。一般来说，这些程序将打开/dev/ kmem，读入内核的符号表，使用该表查找运行的内核中的数据，并使用过程读取内核地址空间中适当的地址。因为这将需要用户程序和内核同意以这种方式读入的数据结构的大小和格式，所以这样的程序在移植时，对每次内核变动以及CPU类型变化，它都需要重新建立。

10.4.2 ptrace和/proc文件系统

在4.3BSD和SVID中，使用ptreace系统调用可以控制进程，并从该进程读取数据。该系统调用通常由调试器使用，以捕获运行进程的执行，或测试其状态。在 SVR4下，ptrace被/proc文件系统所取代，/proc文件系统看上去就象一个目录，每个运行进程在它里面都有一个对应的文件项目，该项目的名称就是运行进程的 ID号。用户程序可以打开感兴趣的进程所对应的文件，并在它上面发出几个 ioctl调用，以控制它的执行，或者从内核获取进程的信息。相同的道理 ，程序可以通过文件描述符从 /proc文件系统读取进程地址空间中的数据，或者把进程地址空间的数据写入到/proc文件系统中。

10.4.3 Linux下的进程控制

在 Linux下，ptrace系统调用可用于进程控制，它的工作过程与 4.3BSD中类似。为了获取进程和系统信息， Linux还提供了/proc文件系统，但它的语义有很大的差别，在 Linux下，/proc包括许多文件，这些文件可以提供通用系统信息，如内存使用、负载平均、装入模块统计，以及网络统计等。这些文件一般是通过 read和write来访问的，它们的内容可以使用 scanf来过滤。 Linux的/proc文件系统还为每个运行进程都提供了一个目录项，它的名称就是进程ID，它包含的文件项目存放如下信息：命令行、到当前工作目录和可执行文件的链接，以及打开的文件描述符等等。内核随时提供所有这些信息，以响应 read请求。这一实现与Plan 9中的/proc文件系统相似，但它确实也有缺点，例如，对于工具 ps(它的作用是列出所有运行进程的信息列表)来说，需要使用许多目录，打开并且读许多文件。相比较而言，在其他 Unix系统上，kvm过程只需要几个系统调用就能直接读内核数据。

很明显，因为每个实现的差别如此之大，移植那些使用它们的应用程序将会是个很困难的工作。必须指出的是， SVR4的/proc文件系统与Linux中的/proc差别是非常大的，它们不能用在同一个上下文中。一般来说，使用 kvm过程或者SVR4/proc文件系统的任何程序都不是真正可移植的，这些代码段对每个操作系统都需要重写。

Linux ptrace调用与BSD中的ptrace几乎相同，它们存在以下几点差异：

- BSD下的请求PTRACE_PEEKUSER在Linux下的名称是PTRACE_PEEKUSR，而BSD下的请求PTRACE_POKEUSER在Linux下的名称则是PTRACE_POKEUSR。
- 进程注册程序可以调用 PTRACE_POKEUSR请求，带上/usr/include/linux/ptrace.h中的偏

移量来进行设置。

- 不支持SunOS请求PTRACE_{READ，WRITE}{TEXT，DATA}，同时也不支持PTRACE_SETACBKPT，PTRACE-SETWRBKPT，PTRACE_CLRBKPT，或者PTRACE_DUMPCORE。这些请求只会对少量现有程序造成影响。

Linux并不提供kvm过程来从用户程序中读取内核地址空间，但有些程序 (最重要的是kmem_ps)实现了这些过程。一般来说，它们不是可移植的。使用 kvm过程的任意代码可能需要依赖于内核中特定符号或数据结构—但作出这样的假设是不可靠和不安全的。使用 kvm过程应被视为是系统结构相关的。

10.5 可移植条件编译

如果用户需要对现有代码进行修改，以便把它移植到 Linux下，用户可以使用ifdef…endif对，用它们包含与Linux相关的代码部分—或者对应到其他实现上的代码。关于如何选择基于操作系统进行编译的代码部分，目前还没有实际的标准，但大多数程序都使用这样一个约定，为系统V的代码定义SVR4，为BSD代码定义BSD，并为与Linux相关的代码定义 linux。

Linux所使用的GNU C库允许用户在编译时定义各种宏，以便打开该库的各种特征。这些宏如下所示：

- _ _STRICT_ANSI_ _：只对ANSI C特征。
- _POSIX_SOURCE：对POSIX.1特征。
- _POSIX_C_SOURCE：如果定义为 1，则对应 POSIX.1特征；如果定义为 2，则对应POSIX.2特征。
- _BSD_SOURCE：ANSI、POSIX和BSD特征。
- _SVID_SOURCE：ANSI、POSIX和系统V特征。
- _GNU_SOURCE：ANSI、POSIX、BSD、SVID和GNU扩展。如果以上这些宏均来定义，则它是默认值。

如果用户自己定义了 _BSD_SOURCE，则还需为库定义一个 _FAVOR_BSD定义。这将导致某些代码会选择BSD行为，而不选择POSIX或SVR4。例如，如果定义了 _FAVOR_BSD，则setjmp和longjmp将保存和恢复信号掩码，而 getpgrp将接收一个PID变元。注意，用户必须链接libbsd，以便使本章前面曾经介绍过的特征可以采取 BSD式的行为。

在Linux下，gcc自动定义了大量的宏，用户可以在程序中使用这些宏。它们是：

- _ _GNUC_ _(GNU C的主版本号，例如2)
- _ _GNUC_MINOR_ _(GNU C的次版本号，例如5)
- unix
- i386
- linux
- _ _ unix_ _
- _ _i386_ _
- _ _linux_ _
- _ _unix
- _ _i386

• _ _linux

许多程序使用#ifdef linux来包围与Linux相关的代码。通过使用这些编译时的宏，用户可以轻易地调整现有代码，加入或者剔除所需的修改，以便把程序移植到 Linux下。注意，因为Linux一般支持比较多的系统 V形式的特征。所以在为系统 V和为BSD所写的程序中，最好是使用系统V版本的代码进行移植，另外，用户也可以从基于 BSD的代码移植，并链接到libbsd。

10.6 补充说明

本章介绍了大部分移植过程中会遇到的问题，但没有介绍系统调用的丢失和流的丢失。前者在系统调用一章中介绍过；有人认为可装入的流模块应该位于 pub/systems/linux/isdn中的ftp.uni_stuttgart.de。

附录　以字母顺序排列的系统调用

表A-1　以字母顺排列的系统调用

系统调用	描　述
_exit	与exit相似，但动作较少(m+c)
accept	接受套接字上的联接(m+c!)
access	检查用户对某文件的许可权限(m+c)
acct	尚未实现(mc)
adjtimex	设置/获取内核时间变量(-c)
afs_syscall	保留的andrew文件系统调用(-)
alarm	在某特定时刻发送SIGALRM(m+c)
bdflush	把某个污染缓冲区刷新到磁盘上(-c)
bind	为进程间通信命名一个套接字(m!c)
break	尚未实现(一)
brk	改变数据段的大小(mc)
chdir	改变工作目录(m+c)
chmod	改变文件属性(m+c)
chown	改变文件所有权(m+c)
chroot	设置新的根目录(mc)
clone	参见fork(m-)
close	通过调用关闭一个文件(m+c)
connect	连接两个套接字(m!c)
creat	创建文件(m+c)
create_module	为可装入内核模块分配空间(-)
delete_module	卸载一个内核模块(-)
dup	创建文件描述符复制(m+c)
dup2	复制文件描述符(m+c)
execl、execlp、execle, …	参见execve(m+!c)
execve	执行某文件(m+c)
exit	终止一个程序(m+c)
fchdir	通过调用改变工作目录
fchmod	参见chmod(mc)
fchown	改变文件的所有权(mc)
fclose	通过调用关闭文件(m+!c)
fcntl	文件/文件描述符控制
flock	改变文件锁定(m!c)
fork	创建字进程(m+c)
fpathconf	通过调用获取文件的有关信息(m+!c)
fread	从流中读取二进制数据的数组(m+!c)
fstat	获取文件状态(m+c)
fstatfs	通过调用获取文件系统状态(mc)
fsync	把文件高速缓存写到磁盘上(mc)
ftime	获取从1970年1月1日以来的时区十秒数信息
ftruncate	改变文件大小(mc)

(续)

系统调用	描 述
fwrite	把二进制数据的数组写入流中(m+!c)
get_kernel_syms	获取内核符号表或它的大小
getdomainname	获取系统的域名(m!c)
getdtablesize	获取文件描述符表的大小(m!c)
getegid	获取有效的组id(m+c)
geteuid	获取有效的用id(m+c)
getgid	获取真正的组id(m+c)
getgroups	获取补充组(m+c)
gethostid	获取唯一的主机标识符(m!c)
gethostname	获取系统主机名(m!c)
getitimer	获取间隔定时器的值(mc)
getpagesize	获取系统页的大小(m-!c)
getpeername	获取相连接的同等套接字的地址(m!c)
getpgid	获取某进程的父进程的组id(+c)
getpgrp	获取当前进程的父进程的组id(m+c)
getpid	获取当前进程的进程id(m+c)
getppid	获取父进程的进程id(m+c)
getpriority	获取进程/组/用户的伏先级(mc)
getrlimit	获取资源限制(mc)
getrusage	获取资源的利用率(m)
getsockname	获取套接字的地址(m!c)
getsockopt	获取套接字的选项设置(m!c)
gettimeofday	获取1970年1月1日以来的时区十秒数信息(mc)
getuid	获取真正的uid(m+c)
getty	尚未实现()
idle	使进程成为可以交换的候选进程(mc)
init_module	插入一个可装入的内核模块(-)
ioctl	操作字符设备(mc)
ioperm	设置一些I/O端口的许可权限(m-c)
iopl	设置所有I/O端口的许可权限(m-c)
ipc	进程间通信(-c)
kill	向进程发送信号(m+c)
killpg	向进程组发送信号(mc!)
klog	参见syslog(-!)
link	为现有的文件创建硬连接(m+c)
listen	监听套接字连接(m!c)
llseek	大型文件所使用的lseek(-)
lock	尚未实现( )
lseek	改变某文件描述符的指针的位置(m+c)
lstat	获取文件状态(mc)
mkdir	创建目录(m+c)
mknod	创建设备(mc)
mmap	把文件映射到内存(mc)
modify_ldt	读或写本地描述符表(-)
mount	挂装一个文件系统(mc)
mprotect	读、写或执行保护内存(-)
mpx	尚未实现( )

(续)

系统调用	描 述
msgctl	ipc消息控制(m!c)
msgget	获取一个ipc消息队列的id(m!c)
msgrcv	接收一个ipc消息(m!c)
msgsnd	发送ipc消息(m!c)
munmap	从内存取某消息的文件映射(mc)
nice	改变进程优先级(mc)
oldfstat	不再使用
oldlstat	不再使用
oldolduname	不再使用
oldstat	不再使用
olduname	不再使用
open	打开文件(m+c)
pathconf	获取文件的有关信息(m+!c)
pause	睡眠，直到信号到达为止(m+c)
personality	改变lbcs当前执行域(-)
phys	尚未实现(m)
pipe	创建管道(m+c)
prof	尚未实现( )
profil	执行时间配置(m!c)
ptrace	跟踪子进程(mc)
quotactl	尚未实现
read	从文件中读数据(m+c)
readv	从文件读数据块(m!c)
readdir	读目录(m+c)
readlink	获取符号连接的内容(mc)
reboot	重启(-mc)
recv	从相连接的套接字接收消息(m!c)
recvfrom	从套接字接收消息(m!c)
rename	删除或者重命名一个文件(m+c)
rmdir	删除一个空目录(m+c)
sbrk	参见brk(mc!)
select	睡眠，直到在文件描述符上执行一个动作(mc)
semctl	ipc信号量控制(m!c)
semget	ipc获取信号量集合标识符(m!c)
semop	在信号量集合成员上执行的ipc操作(m!c)
send	把消息发送到相连接的套接字(m!c)
sendto	把消息发送到套接字(m!c)
setdomainname	设置系统的域名(mc)
setfsgid	设置文件系统组id( )
setfsuid	设置文件系统用户id( )
setgid	设置真正的组id(m+c)
setgroups	设置补充组(mc)
sethostid	设置唯一的主机标识符(mc)
sethostname	设置系统的主机名(mc)
setitimer	设置间隔定时器(mc)
setpgid	设置进程的组id(m+c)
setpgrp	没有效果(mc!)

(续)

系统调用	描 述
setpriority	设置进程/组/用户的优先级(mc)
setregid	设置真正的和有效的组id(mc)
setreuid	设置真正的和有效的用户id(mc)
setrlimit	设置资源限制(mc)
setsid	创建会话(+c)
setsockopt	改变进程的选项(mc)
settimeofday	设置自1970年1月1日以来的时区十秒数信息
setuid	设置真正的用户id(m+c)
setup	初始化设备并挂装根目录(-)
sgetmask	参见siggetmask(m)
shmat	把共享内存连接到数据段上(m!c)
shmctl	ipc操作共享内存(m!c)
shmdt	从数据段上断开共享内存的连接(m!c)
shmget	获取/创建共享内存段(m!c)
shutdown	关闭套接字(m!c)
sigaction	设置/获取信号处理程序(m+c)
sigblock	阻塞信号(m!c)
siggetmask	获取当前进程的信号阻塞(!c)
signal	设置信号处理程序(mc)
sigpause	使用新的信号掩码；直到到达一个信号(mc)
sigpending	获取追加的并且是阻塞的信号(m+c)
sigprocmask	设置/获取当前进程的信号阻塞(+c)
sigreturn	尚未使用( )
sigsetmask	设置当前进程的信号阻塞(c!)
sigsuspend	取代sigpause(m+c)
sigvec	参见sigaction(m!)
socket	创建套接字通信端点(m!c)
socketcall	套接字调用的组合(-)
socketpair	创建两个相互连接的套接字(m!c)
ssetmask	参见sigsetmask(m)
stat	获取文件状态(m+c)
statfs	获取文件系统状态(mc)
stime	设置1970年1月1日以来的秒数(mc)
stty	尚未实现
swapoff	停止交换到文件/设备中(m-c)
swapon	开始交换到文件/设备中(m-c)
symlink	创建到某文件的符号链接(m+c)
sync	同步内存和磁盘缓冲区(mc)
syscall	按编号执行系统调用(-!c)
sysconf	获取某系统变量的值(m+!c)
sysfs	获取配置的文件系统的有关信息( )
sysinfo	获得Linux系统的信息(m-)
syslog	操作系统登录(m-c)
system	执行shell命令(m!c)
time	获取自1970年1月1日以来的秒数(m+c)
times	获取进程时间(m+c)
truncate	改变文件大小(mc)

(续)

系统调用	描 述
ulimit	获取/设置文件限制(c!)
umask	设置文件创建掩码(m+c)
umount	挂装文件系统(mc)
uname	获取系统信息(m+c)
unlink	在不忙时删除某文件(m+c)
uselib	使用共享库(m-c)
ustat	尚未实现(c)
utime	修改索引节点时间项(m+c)
utimes	参见utime(m!c)
vfork	参见fork(m!c)
vhangup	可视地挂起当前tty(m-c)
vm86	进入虚拟的8086模式(m-c)
wait	等待进程中止(m+!c)
wait3	bsd下等待特定进程(m+!c)
wait4	bsd下等待特定进程(mc)
waitpid	等待特定进程(m+c)
write	把数据写到文件中(m+c)
wretev	把数据块写到文件中(m!c)

说明

- (m) 存在对应的man帮助页。
- (+) 服从POSIX的规定。
- (-) Linux相关。
- (c) 在libc中。
- (!) 不是一个单独的系统调用，需要使用另一个系统调用。