第12章 高 级 I/O

12.1 引言

本章内容包括：非阻塞I/O、记录锁、系统V流机制、I/O多路转接（select和poll函数）、readv和writev函数，以及存储映照I/O（mmap）。第14章、第15章中的进程间通信，以及以后各章中的很多实例都要使用本章所述的概念和函数。

12.2 非阻塞I/O

10.5节中曾将系统调用分成两类：低速系统调用和其他。低速系统调用是可能会使进程永远阻塞的一类系统调用：

- 如果数据并不存在，则读文件可能会使调用者永远阻塞（例如读管道，终端设备和网络设备）。
- 如果数据不能立即被接受，则写这些同样的文件也会使调用者永远阻塞。
- 在某些条件发生之前，打开文件会被阻塞（例如打开一个终端设备可能需等到与之连接的调制解调器应答；又例如若以只写方式打开FIFO，那么在没有其他进程已用读方式打开该FIFO时也要等待）。
- 对已经加上强制性记录锁的文件进行读、写。
- 某些ioctl操作。
- 某些进程间通信函数（见第14章）。

虽然读、写磁盘文件会使调用在短暂时间内阻塞，但并不能将它们视为“低速”。

非阻塞I/O使我们可以调用不会永远阻塞的I/O操作，例如open,read和write。如果这种操作不能完成，则立即出错返回，表示该操作如继续执行将继续阻塞下去。

对于一个给定的描述符有两种方法对其指定非阻塞I/O：

(1) 如果是调用open以获得该描述符，则可指定O_NONBLOCK标志（见3.3节）。

(2) 对于已经打开的一个描述符，则可调用fcntl打开O_NONBLOCK文件状态标志（见3.13节）。程序3-5中的函数可用来为一个描述符打开任一文件状态标志。

> 早期的系统V版本使用标志O_NDELAY指定非阻塞方式。在这些版本中，如果无数据可读，则read返回值0。而UNIX又常将read的返回值0解释为文件结束，两者有所混淆。 POSIX.1则提供了一个非阻塞标志，它的名字和语义都与O_NDELAY不同。POSIX.1要求，对于一个非阻塞的描述符如果无数据可读，则read返回－1，并且errno被设置为EAGAIN。SVR4支持较老的O_NDELAY和POSIX.1的O_NONBLOCK，但在本书的实例中只使用 POSIX.1规定的特征。O_NDELAY的使用只是为了向后兼容，不应在新应用程序中使用。
>
> 4.3BSD为fcntl提供FNDELAY标志，其语义也稍有区别。它不只影响该描述符的文件状态标志，还将终端设备或套接口的标志更改成非阻塞的，因此影响了终端或套接口的所有用户，不只是影响共享同一文件表项的用户（4.3BSD非阻塞I/O只对终端和套接口起作用）。如果对一个非阻塞描述符的操作不能无阻塞地完

成，那么4.3BSD返回EWOULDBLOCK。4.3+BSD提供POSIX.1的O_NONBLOCK标志，但其语义却类似于4.3BSD的FNDELAY。非阻塞I/O通常用来处理终端设备或网络连接，而这些设备通常一次由一个进程使用。这就意味着 BSD语义的更改通常不会影响我们。出错返回EWOULDBLOCK而不是POSIX.1的EAGAIN，这造成了可移植性问题，必须处理这一问题。 4.3+BSD也支持FIFO，非阻塞I/O也对FIFO起作用。

实例

程序12-1是一个非阻塞I/O的实例，它从标准输入读100 000字节，并试图将它们写到标准输出上。该程序先将标准输出设置为非阻塞的，然后用 for循环进行输出，每次写的结果都在标准出错上打印。函数clr-fl类似于程序3-5中的set_f1，但与set_f1的功能相反，它清除1个或多个标志位。

程序12-1 长的非阻塞写

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

char    buf[100000];

int
main(void)
{
    int     ntowrite, nwrite;
    char    *ptr;

    ntowrite = read(STDIN_FILENO, buf, sizeof(buf));
    fprintf(stderr, "read %d bytes\n", ntowrite);

    set_fl(STDOUT_FILENO, O_NONBLOCK);  /* set nonblocking */

    for (ptr = buf; ntowrite > 0; ) {
        errno = 0;
        nwrite = write(STDOUT_FILENO, ptr, ntowrite);
        fprintf(stderr, "nwrite = %d, errno = %d\n", nwrite, errno);
        if (nwrite > 0) {
            ptr += nwrite;
            ntowrite -= nwrite;
        }
    }

    clr_fl(STDOUT_FILENO, O_NONBLOCK);  /* clear nonblocking */
    exit(0);
}
```

若标准输出是普通文件，则可以期望write只执行一次。

```
$ ls -l /etc/termcap                打印文件长度
-rw-rw-r-- 1 root   133439 Oct 11 1990 /etc/termcap
$ a.out < /etc/termcap > temp.file  测试一普通文件
read 100000 bytes
nwrite = 100000, errno = 0          一次写
```

```
$ ls -l temp.file                    检验输出文件长度
-rw-rw-r-- 1 stevens 100000 Nov 21 16:27 temp.file
```

但是，若标准输出是终端，则期望write有时会返回一个数字，有时则出错返回。下面是在一个系统上运行程序12-1的结果：

```
$ a.out < /etc/termcap 2>stderr.out   向终端输出
                                      大量输出至终端
$ cat stderr.out
read 100000 bytes
nwrite = 8192, errno = 0
nwrite = 8192, errno = 0
nwrite = -1, errno = 11               这种错211次
...
nwrite = 4096,errno = 0
nwrite = -1, errno = 11               这种错658次
...
nwrite = 4096,errno = 0
nwrite = -1, errno = 11               这种错604次
...
nwrite = 4096,errno = 0
nwrite = -1, errno = 11               这种错1047次
...
nwrite = -1, errno = 11               这种错1046次
...
nwrite = 4096, errno = 0              …等等
```

在该系统上，errno 11是EAGAIN。系统上的终端驱动程序总是一次接收 4096或8192字节。在另一个系统上，前三个write返回2005，1822和1811，然后是96次出错返回，接着是返回1846等等。每个write能写多少字节依赖于系统。

此程序若在SVR4中运行，则其结果完全不同于前面的情况。当输出到终端上时，输出整个输入文件只需要一个write。显然，非阻塞方式并不构成区别。创建一个更大的输入文件，并且系统为运行该程序增加了程序缓存。程序的这种运行方式（即输出一整个文件，只调用一次write）一直继续到输入文件长度达到约700 000字节。达到此长度后，每一个write都返回出错EAGAIN。（输入文件则决不会再输出到终端上——该程序只是连续地产生出错消息流。）

发生这种情况是因为：在SVR4中终端驱动程序通过流I/O系统连接到程序（12.4节将详细说明流）。流系统有它自己的缓存，它一次能从程序接收更多的数据。 SVR4的行为也依赖于终端类型——硬连线终端、控制台设备或伪终端。

在此实例中，程序发出了数千个write调用，但是只有20个左右是真正输出数据的，其余的则出错返回。这种形式的循环称为轮询，在多用户系统上它浪费了 CPU时间。12.5节将介绍非阻塞描述符的I/O多路转接，这是一种进行这种操作的更加有效的方法。

第17章将会用到非阻塞I/O，我们将要输出到终端设备（PostScript打印机）而且不希望在write上阻塞。

12.3 记录锁

当两个人同时编辑一个文件时，其后果将如何呢？在很多UNIX系统中，该文件的最后状态取决于写该文件的最后一个进程。但是对于有些应用程序，例如数据库，有时进程需要确保它正在单独写一个文件。为了向进程提供这种功能，较新的 UNIX系统提供了记录锁机制。（第16

章包含了使用记录锁的数据库子程序库。）

记录锁（record locking）的功能是：一个进程正在读或修改文件的某个部分时，可以阻止其他进程修改同一文件区。对于UNIX，“记录”这个定语也是误用，因为UNIX内核根本没有使用文件记录这种概念。一个更适合的术语可能是“区域锁”，因为它锁定的只是文件的一个区域（也可能是整个文件）。

12.3.1 历史

表12-1列出了各种UNIX系统提供的不同形式的记录锁。

表12-1 各种UNIX系统支持的记录锁形式

系 统	建议性	强制性	fcnt1	lockf	flock
POSIX.1	•		•		
XPG3	•		•		
SVR2	•		•	•	
SVR3, SVR4	•	•	•	•	
4.3BSD	•				•
4.3BSD Reno	•		•		•

本节的最后将说明建议性锁和强制性锁之间的区别。POSIX.1选择了以fcntl函数为基础的系统V风格的记录锁。这种风格也得到4.3BSD Reno版本的支持。

早期的伯克利版只支持BSD flock函数。此函数只锁整个文件，而不锁文件中的一个区域。但是POSIX.1的fcntl函数可以锁文件中的任一区域，大至整个文件，小至单个字节。

本书只说明POSIX.1的fcntl锁。系统V的lockf函数只是fcntl函数的一个界面。

> 记录锁是1980年由John Bass最早加到V7上的。内核中相应的系统调用入口表项是名为locking的函数。此函数提供了强制性记录锁功能，它被用在很多制造商的系统III版本中。Xenix系统采用了此函数，SVR4在Xenix兼容库中仍旧支持该函数。
>
> SVR2是系统V中第一个支持fcntl风格记录锁的版本（1984年）。

12.3.2 fcntl记录锁

3.13节中已经给出了fcntl函数的原型，为了叙说方便，这里再重复一次。

```
#include <sys/types.h>
#include <unistd.h>
#include <fcnt1.h>

int fcnt1(int filedes, int cmd, .../* struct flock *flockptr */);
```

返回：若成功则依赖于*cmd*（见下），若出错则为-1

对于记录锁，*cmd*是F_GETLK、F_SETLK或F_SETLKW。第三个参数（称其为*flockptr*）是一个指向flock结构的指针。

```
struct flock {
  short  l_type;    /* F_RDLCK, F_WRLCK, or F_UNLCK */
  off_t  l_start;   /* offset in bytes, relative to l_whence */
  short  l_whence;  /* SEEK_SET, SEEK_CUR, or SEEK_END */
  off_t  l_len;     /* length, in bytes; 0 means lock to EOF */
  pid_t  l_pid;     /* returned with F_GETLK */
};
```

flock结构说明：

• 所希望的锁类型：F_RDLCK（共享读锁）、F_WRLCK（独占性写锁）或F_UNLCK（解锁一个区域）

• 要加锁或解锁的区域的起始地址，由l_start和l_whence两者决定。l_stat是相对位移量（字节），l_whence则决定了相对位移量的起点。这与lseek函数（见3.6节）中最后两个参数类似。

• 区域的长度，由l_len表示。

关于加锁和解锁区域的说明还要注意下列各点：

• 该区域可以在当前文件尾端处开始或越过其尾端处开始，但是不能在文件起始位置之前开始或越过该起始位置。

• 如若l_len为0，则表示锁的区域从其起点（由l_start和l_whence决定）开始直至最大可能位置为止。也就是不管添写到该文件中多少数据，它都处于锁的范围。

• 为了锁整个文件，通常的方法是将l_start说明为0，l_whence说明为SEEK_SET，l_len说明为0。

上面提到了两种类型的锁：共享读锁（l_type为L_RDLCK）和独占写琐（L_WRLCK）。基本规则是：多个进程在一个给定的字节上可以有一把共享的读锁，但是在一个给定字节上的写锁则只能由一个进程独用。更进一步而言，如果在一个给定字节上已经有一把或多把读锁，则不能在该字节上再加写锁；如果在一个字节上已经有一把独占性的写锁，则不能再对它加任何读锁。在表12-2中示出了这些规则。

表12-2 不同类型锁之间的兼容性

区域当前有 \ 要求	读锁	写锁
无锁	可以	可以
一把或多把读锁	可以	拒绝
一把写锁	拒绝	拒绝

加读锁时，该描述符必须是读打开；加写锁时，该描述符必须是写打开。

以下说明fcntl函数的三种命令：

• F_GETLK　决定由*flockptr*所描述的锁是否被另外一把锁所排斥（阻塞）。如果存在一把锁，它阻止创建由*flockptr*所描述的锁，则这把现存的锁的信息写到*flockptr*指向的结构中。如果不存在这种情况，则除了将l_type设置为F_UNLCK之外，*flockptr*所指向结构中的其他信息保持不变。

• F_SETLK　设置由*flockptr*所描述的锁。如果试图建立一把按上述兼容性规则并不允许的锁，则fcntl立即出错返回，此时errno设置为EACCES或EAGAIN。

> SVR2和SVR4返回EACCES，但手册页警告将来返回EAGAIN。4.3+BSD则返回EAGAIN。POSIX.1允许这两种情况。

此命令也用来清除由*flockptr*说明的锁（l_type为F_UNLCK）。

• F_SETLKW 这是F_SETLK的阻塞版本（命令名中的W表示等待（wait））。如果由于存在其他锁，那么按兼容性规则由 *flockptr*所要求的锁不能被创建，则调用进程睡眠。如果捕捉到信号则睡眠中断。

应当了解，用F_GETLK测试能否建立一把锁，然后用F_SETLK和F_SETLKW企图建立一把锁，这两者不是一个原子操作。在这两个操作之间可能会有另一个进程插入并建立一把相关的锁，使原来测试到的情况发生变化，如果不希望在建立锁时可能产生的长期阻塞，则应使用F_SETLK，并对返回结果进行测试，以判别是否成功地建立了所要求的锁。

在设置或释放文件上的一把锁时，系统按需组合或裂开相邻区。例如，若对字节 0~99设置一把读锁，然后对字节0~49设置一把写锁，则有两个加锁区：0~49字节（写锁）及50~99（读锁）。又如，若100~199字节是加锁的区，需解锁第150字节，则内核将维持两把锁，一把用于100~149字节，另一把用于151~199字节。

实例 —— 要求和释放一把锁

为了避免每次分配flock结构，然后又填入各项信息，可以用程序 12-2中的函数lock_reg来处理这些细节。

程序12-2 加锁和解锁一个文件区域的函数

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

int
lock_reg(int fd, int cmd, int type, off_t offset, int whence, off_t len)
{
    struct flock    lock;

    lock.l_type = type;     /* F_RDLCK, F_WRLCK, F_UNLCK */
    lock.l_start = offset;  /* byte offset, relative to l_whence */
    lock.l_whence = whence; /* SEEK_SET, SEEK_CUR, SEEK_END */
    lock.l_len = len;       /* #bytes (0 means to EOF) */

    return( fcntl(fd, cmd, &lock) );
}
```

因为大多数锁调用是加锁或解锁一个文件区域（命令 F_GETLK很少使用），故通常使用下列五个宏，它们都定义在ourhdr.h中（见附录B）。

```
#define read_lock(fd,offset,whence,len) \
          lock_reg(fd,F_SETLK,F_RDLCK,offset,whence,len)
#define readw_lock(fd,offset,whence,len) \
          lock_reg(fd,F_SETLKW,F_RDLCK,offset,whence,len)
#define write_lock(fd,offset,whence,len) \
          lock_reg(fd,F_SETLK,F_WRLCK,offset,whence,len)
#define writew_lock(fd,offset,whence,len) \
          lock_reg(fd,F_SETLKW,F_WRLCK,offset,whence,len)
#define un_lock(fd,offset,whence,len) \
          lock_reg(fd,F_SETLK,F_UNLCK,offset,whence,len)
```

我们用与lseek函数同样的顺序定义这些宏中的三个参数。

实例 —— 测试一把锁

程序12-3定义了一个函数lock_test，可用其测试一把锁。

程序12-3 测试一个锁条件的函数

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

pid_t
lock_test(int fd, int type, off_t offset, int whence, off_t len)
{
    struct flock    lock;

    lock.l_type = type;     /* F_RDLCK or F_WRLCK */
    lock.l_start = offset;  /* byte offset, relative to l_whence */
    lock.l_whence = whence; /* SEEK_SET, SEEK_CUR, SEEK_END */
    lock.l_len = len;       /* #bytes (0 means to EOF) */

    if (fcntl(fd, F_GETLK, &lock) < 0)
        err_sys("fcntl error");

    if (lock.l_type == F_UNLCK)
        return(0);      /* false, region is not locked by another proc */
    return(lock.l_pid); /* true, return pid of lock owner */
}
```

如果存在一把锁，它阻塞由参数说明的锁，则此函数返回持有这把现存锁的进程的 ID，否则此函数返回0。通常用下面两个宏来调用此函数（它们也定义在 ourhdr.h）。

```
#define is_readlock(fd,offset,whence,len) \
            lock_test(fd,F_RDLCK,offset,whence,len)
#define is_writelock(fd,offset,whence,len) \
            lock_test(fd,F_WRLCK,offset,whence,len)
```

实例 —— 死锁

如果两个进程相互等待对方持有并且不释放（锁定）的资源时，则这两个进程就处于死锁状态。如果一个进程已经控制了文件中的一个加锁区域，然后它又试图对另一个进程控制的区域加锁，则它就会睡眠，在这种情况下，有发生死锁的可能性。

程序12-4给出了一个死锁的例子。子进程锁字节 0，父进程锁字节1。然后，它们中的每一个又试图锁对方已经加锁的字节。在该程序中使用了 8.8节中介绍的父 -子进程同步例程（TELL_xxx和WAIT_xxx）,使得对方都能建立第一把锁。运行程序12-4得到：

程序12-4 死锁检测实例

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

static void lockabyte(const char *, int, off_t);

int
main(void)
{
    int     fd;
    pid_t   pid;

        /* Create a file and write two bytes to it */
    if ( (fd = creat("templock", FILE_MODE)) < 0)
        err_sys("creat error");
    if (write(fd, "ab", 2) != 2)
```

```
            err_sys("write error");

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

        else if (pid == 0) {            /* child */
            lockabyte("child", fd, 0);
            TELL_PARENT(getppid());
            WAIT_PARENT();
            lockabyte("child", fd, 1);

        } else {                        /* parent */
            lockabyte("parent", fd, 1);
            TELL_CHILD(pid);
            WAIT_CHILD();
            lockabyte("parent", fd, 0);
        }
        exit(0);
    }

    static void
    lockabyte(const char *name, int fd, off_t offset)
    {
        if (writew_lock(fd, offset, SEEK_SET, 1) < 0)
            err_sys("%s: writew_lock error", name);

        printf("%s: got the lock, byte %d\n", name, offset);
    }
```

```
$ a.out
child:got the lock,byte 0
parent:got the lock,byte 1
child:writew_lock error:Deadlock situation detected/avoided
parent:got the lock,byte 0
```

检测到死锁时，内核必须选择一个进程收到出错返回。在本实例中，选择了子进程，这是一个实现细节。当此程序在另一个系统上运行时，一半次数是子进程接到出错信息，另一半则是父进程。

12.3.3 锁的隐含继承和释放

关于记录锁的自动继承和释放有三条规则：

(1) 锁与进程、文件两方面有关。这有两重含意：第一重很明显，当一个进程终止时，它所建立的锁全部释放；第二重意思就不很明显，任何时候关闭一个描述符时，则该进程通过这一描述符可以存访的文件上的任何一把锁都被释放（这些锁都是该进程设置的）。这就意味着如果执行下列四步：

```
fd1=open(pathname, ...);
read_lock(fd1, ...);
fd2=dup(fd1);
close(fd2);
```

则在close（fd2）后，在fd1上设置的锁被释放。如果将dup代换为open，其效果也一样：

```
fd1=open(pathname, ...);
read_lock(fd1, ...);
fd2=open(pathname, ...);
```

```
close(fd2);
```

(2) 由fork产生的子程序不继承父进程所设置的锁。这意味着，若一个进程得到一把锁，然后调用fork，那么对于父进程获得的锁而言，子进程被视为另一个进程，对于从父进程处继承过来的任一描述符，子进程要调用fcntl以获得它自己的锁。这与锁的作用是相一致的。锁的作用是阻止多个进程同时写同一个文件（或同一文件区域）。如果子进程继承父进程的锁，则父、子进程就可以同时写同一个文件。

(3) 在执行exec后，新程序可以继承原执行程序的锁。

> POSIX.1没有要求这一点。但是，SVR4和4.3+BSD都支持这一点。

12.3.4 4.3+BSD的实现

先简要地观察4.3+BSD实现中使用的数据结构，从中可以看到锁是与进程、文件相关联的。考虑一个进程，它执行下列语句（忽略出错返回）：

```
fd1 = open(pathname, ...);
write_lock(fd1, 0, SEEK_SET, 1);    /* parent write locks byte 0 */
if (fork() > 0) {      /* parent */
    fd2 = dup(fd1);
    fd3 = open(pathname, ...);
    pause();
} else {
    read_lock(fd1, 1, SEEK_SET, 1); /* child read locks byte 1 */
    pause();
}
```

图12-1显示了父、子进程暂停（执行pause( )）后的数据结构情况。

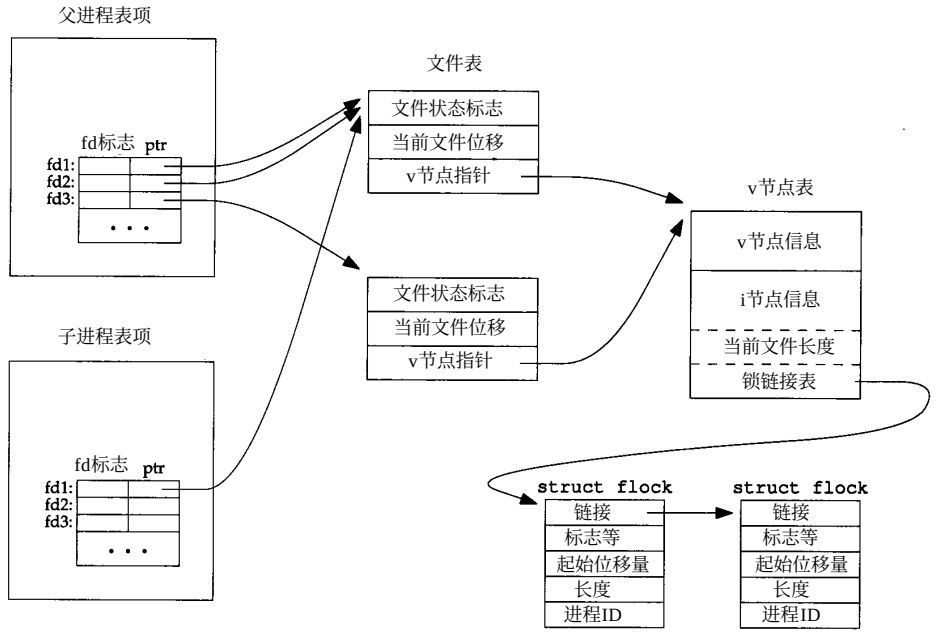

图12-1 关于记录锁的4.3+BSD数据结构

图3-3和图8-1中已显示了open、fork以及dup后的数据结构。有了记录锁后，在原来的这些图上新加了flock结构，它们由i节点结构开始相互连接起来。注意，每个 flock结构说明了一个给定进程的一个加锁区域。图中显示了两个 flock结构，一个是由父进程调用 write_lock形成的，另一个则是由子进程调用read_lock形成的。每一个结构都包含了相应进程ID。

在父进程中，关闭fd1、fd2和fd3中的任意一个都将释放由父进程设置的写锁。在关闭这三个描述符中的任意一个时，内核会从该描述符所关连的 i节点开始，逐个检查 flock连接表中各项，并释放由调用进程持有的各把锁。内核并不清楚也不关心父进程是用哪一个描述符来设置这把锁的。

实例

建议性锁可由精灵进程使用以保证该精灵进程只有一个副本在运行。起动时，很多精灵进程都把它们的进程ID写到一个各自专用的PID文件上。系统停机时，可以从这些文件中取用这些精灵进程的进程ID。防止一个精灵进程有多份副本同时运行的方法是：在精灵进程开始运行时，在它的进程ID文件上设置一把写锁。如果在它运行时一直保持这把锁，则不可能再起动它的其他副本。程序12-5实现了这一技术。

因为进程ID文件可能包含以前的精灵进程 ID，而且其长度可能长于当前进程的 ID，例如该文件中以前的内容可能是 12345\n，而现在的进程 ID是654，我们希望该文件现在只包含654\n，而不是654\n5\n，所以在写该文件时，先将其截短为 0。注意，要在设置了锁之后再调用截短文件长度的函数 ftruncate。在调用open时不能指定O_TRUNC，因为这样做会在有一个这种精灵进程运行并对该文件加锁时也会使该文件截短为 0。（如果使用强制性锁而不是建议性锁，则可使用O_TRUNC。本节最后将讨论强制性锁。）

在本实例中，也对该描述符设置了运行时关闭标志。这是因为精灵进程常常 fork并exec其他进程，并无需在另一个进程中使该文件也处在打开状态。

程序12-5　精灵进程阻止其多份副本同时运行的起动代码

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

#define PIDFILE     "daemon.pid"

int
main(void)
{
    int     fd, val;
    char    buf[10];

    if ( (fd = open(PIDFILE, O_WRONLY | O_CREAT, FILE_MODE)) < 0)
        err_sys("open error");

            /* try and set a write lock on the entire file */
    if (write_lock(fd, 0, SEEK_SET, 0) < 0) {
        if (errno == EACCES || errno == EAGAIN)
            exit(0);    /* gracefully exit, daemon is already running */
        else
            err_sys("write_lock error");
    }

            /* truncate to zero length, now that we have the lock */
```

```
    if (ftruncate(fd, 0) < 0)
        err_sys("ftruncate error");

            /* and write our process ID */
    sprintf(buf, "%d\n", getpid());
    if (write(fd, buf, strlen(buf)) != strlen(buf))
        err_sys("write error");

            /* set close-on-exec flag for descriptor */
    if ( (val = fcntl(fd, F_GETFD, 0)) < 0)
        err_sys("fcntl F_GETFD error");
    val |= FD_CLOEXEC;
    if (fcntl(fd, F_SETFD, val) < 0)
        err_sys("fcntl F_SETFD error");

    /* leave file open until we terminate: lock will be held */

    /* do whatever ... */

    exit(0);
}
```

实例

在相对文件尾端加锁或解锁时需要特别小心。大多数实现按照 l_whence的SEEK_CUR或SEEK_END值，用文件当前位置或当前长度以及l_start得到绝对的文件位移量。但是，通常需要相对于文件的当前位置或当前长度指定一把锁。

程序12-6写一个大文件，一次一个字节。在每次循环中,从文件当前尾端开始处加锁直到将来可能扩充到的尾端为止（最后一个参数，长度指定为 0），并写1个字节。然后解除这把锁，写另一个字节。如果系统用“从当前尾端开始，直到将来可能扩充的尾端”这种记法来跟踪锁，那么这段程序能够正常工作。但是如果系统将相对位移量变换成绝对位移量就会有问题。在SVR4中运行此程序的确会发生问题：

程序12-6 显示相对于文件末尾的锁的问题的程序

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

int
main(void)
{
    int     i, fd;

    if ( (fd = open("temp.lock", O_RDWR | O_CREAT | O_TRUNC,
                                                FILE_MODE)) < 0)
        err_sys("open error");

    for (i = 0; i < 1000000; i++) { /* try to write 2 Mbytes */
                /* lock from current EOF to EOF */
        if (writew_lock(fd, 0, SEEK_END, 0) < 0)
            err_sys("writew_lock error");

        if (write(fd, &fd, 1) != 1)
            err_sys("write error");

        if (un_lock(fd, 0, SEEK_END, 0) < 0)
            err_sys("un_lock error");

        if (write(fd, &fd, 1) != 1)
```

```
            err_sys("write error");
        }
        exit(0);
    }
```

```
$ a.out
writew_lock error: No record locks available
$ ls -l temp.lock
-rw-r--r--    1 stevens    other     592 Nov      1 04:41 temp.lock
```

（内核返回ENOLCK。它表示内核中的锁表已用完。）分析系统是如何做的会从中得到教益。图12-2显示了第一次调用writew_lock和write之后的文件状态。

因为在writew_lock调用中，指定“直至将来可能扩充到尾端”，所以图中锁定区域超过了所写的第一个字节。

然后调用un_lock。从当前尾端处开始直至将来可能扩充到的尾端为止解锁，它将图12-2中箭头的右端缩回到第一字节位置端部。然后将第二个字节写到文件中。图12-3显示了调用un_lock以及写了第二个字节后的文件状态。

在经过第二次for循环后，在文件上共写了4个字节。图12-4显示了此时文件及锁的状态。

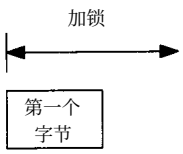

图12-2 第一次调用writew_lock和write之后的文件状态

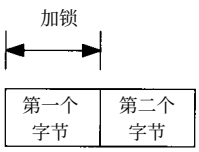

图12-3 调用un_lock和写第二字节后的状态

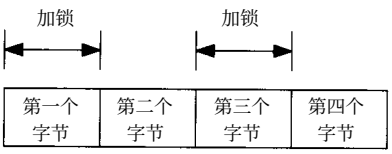

图12-4 第二次for循环后文件及锁的状态

这种情况不断重复，直至内核为该进程用完了锁结构。此时fcntl出错返回，errno设置为ENOLCK。

在此程序中，每次写的字节数是已知的，所以可将un_lock的第二个参数（其值将赋与l_start）改换成所写字节数的负值（在本程序中是－1）。这就使得un_lock去除上次加的锁。

> 实际上，在开发_db_writedat和_db_writeidx函数时，作者的系统出现了问题。16.7节给出了关于此问题的一个稍稍不同的方法。

12.3.5 建议性锁和强制性锁

考虑数据库存取例程序。如果该库中所有函数都以一致的方法处理记录锁，则称使用这些函数存取数据库的任何进程集为合作进程（*cooperating process*）。如果这些函数是唯一的用来存取数据库的函数，那么它们使用建议性锁是可行的。但是建议性锁并不能阻止对数据库文件有写许可权的任何其他进程写数据库文件。不使用协同一致的方法（数据库存取例程序）来存取数据库的进程是一个非合作进程。

强制性锁机制中，内核对每一个open、read和write都要检查调用进程对正在存取的文件是否违背了某一把锁的作用。

> 表12-1显示了SVR4提供强制性记录锁，而POSIX.1不提供。

对一个特定文件打开其设置-组-ID位，关闭其组-执行位则对该文件启动了强制性锁机制。

(回忆程序4-4)。因为当组-执行位关闭时，设置-组-ID位不再有意义，所以SVR3的设计者借用两者的这种组合来指定对一个文件的锁是强制性的而非建议性的。

如果一个进程试图读、写一个强制性锁起作用的文件，而欲读、写的部分又由其他进程加上了读、写锁，此时会发生什么呢？对这一问题的回答取决于三方面的因素：操作类型（ read或write)，其他进程保有的锁的类型（读锁或写锁)，以及有关描述符是阻塞还是非阻塞的。表12-3列出了这8种可能性。

表12-3 强制性锁对其他进程读、写的影响

	阻塞描述符，试图		非阻塞描述符，试图	
	读	写	读	写
在区域上的读锁	可以	阻塞	可以	EAGAIN
在区域上的写锁	阻塞	阻塞	EAGAIN	EAGAIN

除了表12-3中的read,write函数，其他进程的强制性锁也会对open函数产生影响。通常，即使正在打开的文件具有强制性记录锁，该打开操作也会成功。下面的 read或write依从于表12-3中所示的规则。但是，如果欲打开的文件具有强制性锁（读锁或写锁)，而且open调用中的flag为O_TRUNC或O_CREAT，则不论是否指定O_NONBLOCK,open都立即出错返回， errno设置为EAGAIN。(对O_TRUNC情况出错返回是有意义的，因为其他进程对该文件持有读、写锁，所以不能将其截短为0。对O_CREAT情况在返回时也设置errno则无意义，因为该标志的意义是如果该文件不存在则创建，由于其他进程对该文件持有记录锁，因而该文件肯定是存在的。)

这种处理方式可能导致令人惊异的结果。我们曾编写过一个程序，它打开一个文件（其模式指定为强制性锁)，然后对该文件的整体设置一把读锁，然后进入睡眠一段时间。在这段睡眠时间内，用某些常规的UNIX程序和操作符对该文件进行处理，发现下列情况：

• 可用ed编辑程序对该文件进行编辑操作，而且编辑结果可以写回磁盘！强制性记录锁对此毫无影响。对ed操作进行跟踪分析发现，ed将新内容写到一个临时文件中，然后删除原文件，最后将临时文件名改名为原文件名。于是，发现强制性锁机制对 unlink函数没有影响。

> 在SVR4中，用 truss(1)命令可以得到一个进程的系统调用跟踪信息，在4.3+BSD中，则使用ktrace(1)和kdump(1)命令。

• 不能用vi编辑程序编辑该文件。 vi可以读该文件，但是如果试图将新的数据写到该文件中，则出错返回（EAGAIN)。如果试图将新数据添加到该文件中，则 write阻塞。vi的这种行为与所希望的一样。

• 使用KornShell的>和>>算符重写或添写到该文件中，产生出错信息“ cannot creat ”。

• 在Bourne shell下使用>算符出错，但是使用>>算符则阻塞，在删除了强制性锁后再继续进行处理。(执行添加操作所产生的区别是因为： Korn Shell以O_CREAT和O_APPEND标志打开文件，而上面已提及指定O_CREAT会产生出错返回。但是， Bourne shell在该文件已存在时并不指定O_CREAT，所以open成功，而下一个write则阻塞。)

从这样一个例子中可见，在使用强制性锁时还需有所警惕。

一个别有用心的用户可以对大家都可读的文件加一把读锁（强制性)，这样就能阻止任何其他人写该文件（当然，该文件应当是强制性锁机制起作用的，这可能要求该用户能够更改该文件的许可权位)。考虑一个数据库文件，它是大家都可读的，并且是强制性锁机制起作用的。

如果一个别有用心的用户对该整个文件保有一把读锁，则其他进程不能再写该文件。

实例

程序12-7用于确定一个系统是否支持强制性锁机制。

程序12-7　确定是否支持强制性锁

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

int
main(void)
{
    int             fd;
    pid_t           pid;
    char            buff[5];
    struct stat     statbuf;

 if ( (fd = open("templock", O_RDWR | O_CREAT | O_TRUNC,
                                          FILE_MODE)) < 0)
     err_sys("open error");
 if (write(fd, "abcdef", 6) != 6)
     err_sys("write error");

     /* turn on set-group-ID and turn off group-execute */
 if (fstat(fd, &statbuf) < 0)
     err_sys("fstat error");
 if (fchmod(fd, (statbuf.st_mode & ~S_IXGRP) | S_ISGID) < 0)
     err_sys("fchmod error");

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

 } else if (pid > 0) {    /* parent */
             /* write lock entire file */
     if (write_lock(fd, 0, SEEK_SET, 0) < 0)
         err_sys("write_lock error");
     TELL_CHILD(pid);

     if (waitpid(pid, NULL, 0) < 0)
         err_sys("waitpid error");

 } else {                 /* child */
     WAIT_PARENT();       /* wait for parent to set lock */

     set_fl(fd, O_NONBLOCK);

         /* first let's see what error we get if region is locked */
     if (read_lock(fd, 0, SEEK_SET, 0) != -1)     /* no wait */
         err_sys("child: read_lock succeeded");
     printf("read_lock of already-locked region returns %d\n", errno);

         /* now try to read the mandatory locked file */
     if (lseek(fd, 0, SEEK_SET) == -1)
         err_sys("lseek error");
     if (read(fd, buff, 2) < 0)
```

```
            err_ret("read failed (mandatory locking works)");
        else
            printf("read OK (no mandatory locking), buff = %2.2s\n", buff);
    }
    exit(0);
}
```

此程序首先创建一个文件，并使强制性锁机制对其起作用。然后 fork一个子进程。父进程对整个文件设置一把写锁，子进程则将该文件的描述符设置为非阻塞的，然后企图对该文件设置一把读锁，我们期望这会出错返回，并希望看到系统返回值是 EACCES或EAGAIN。接着，子进程将文件读、写位置调整到文件起点，并试图读该文件。如果系统提供强制性锁机制，则read应返回EACCES或EAGAIN（因为该描述符是非阻塞的）。否则read返回所读的数据。在SVR4中运行此程序（该系统支持强制性锁机制），得到：

```
$ a.out
read_lock of already-locked region returns 13
read failed (mandatory locking works):No more processes
```

查看系统头文件或intro(2)手册页，可以看到错误13对应于EACCES。从例子中还可以看到，在read出错返回信息部分中包含有“ No more processes ”。这通常来自于fork，表示已用完了进程表项。

若采用4.3+BSD系统，则得到：

```
$ a.out
read_lock of already_locked region returns 35
read OK (no mandatory locking),buff=ab
```

其中，errno35对应于EAGAIN。该系统不支持强制性锁。

实例

让我们回到本节的第一个问题：当两个人同时编辑同一个文件将会怎样呢？一般的 UNIX文本编辑器并不使用记录锁，所以对此问题的回答仍然是：该文件的最后结果取决于写该文件的最后一个进程。（4.3+BSD的vi编辑器确实有一个编译选择项使运行时建议性记录锁起作用，但是这一选择项并不是默认可用的。）即使我们在一个编辑器，例如 vi中使用了建议性锁，可是这把锁并不能阻止其他用户使用另一个没有使用建议性记录锁的编辑器。

若系统提供强制性记录锁，那么可以修改常用的编辑器（如果有该编辑器的源代码）。如没有该编辑器的源代码，那么可以试一试下述方法。编写一个 vi的前端程序。该程序立即调用fork，然后父进程等待子进程终止，子进程打开在命令行中指定的文件，使强制性锁起作用，对整个文件设置一把写锁，然后运行 vi。在vi运行时，该文件是加了写锁的，所以其他用户不能修改它。当vi结束时，父进程从wait返回，此时自编的前端程序也就结束。本例中假定锁能跨越exec，这正是前面所说的SVR4的情况（SVR4是提供强制性锁的唯一系统）。

这种类型的前端程序是可以编写的，但却往往不能起作用。问题出在大多数编辑器（至少是vi和ed）读它们的输入文件，然后关闭它。只要引用被编辑文件的描述符关闭了，那么加在该文件上的锁就被释放了。这意味着，在编辑器读了该文件的内容，然后关闭了它，那么锁也就不存在了。前端程序中没有任何方法可以阻止这一点。

第16章的数据库函数库使用了记录锁以阻止多个进程的并发存取。本章则提供了时间测量以观察记录锁对进程的影响。

12.4 流

流是系统V提供的构造内核设备驱动程序和网络协议包的一种通用方法，对流进行讨论的目的是理解下列各点：

(1) 系统V的终端界面。

(2) I/O多路复用中轮询函数的使用（见12.5.2节）。

(3) 基于流管道和命名流管道的实现（见15.2和12.5.2节）。

> 流机制是由Dennis Ritchie 发展起来的〔Ritchie 1984〕,其目的是用通用、灵活的方法改写传统的字符 I/O系统并适应网络协议的需要，后来流机制被加入SVR3。SVR4则提供了对流（基于流的终端I/O系统）的全面支持。〔AT&T 1990d〕对SVR4实现进行了说明。
>
> 请注意不要将本章说明的流与标准I/O库（见5.2节）中使用的流相混淆。

流在用户进程和设备驱动程序之间提供了一条全双工通路。流无需和实际硬件设备直接对话——流也可以用作为伪设备驱动程序。图12-5示出了一个简单流的基本结构。

在流首之下可以压入处理模块。这可以用 ioctl实现。图12-6示出了一个包含一个处理模块的流。各方框之间用两根带箭头的线连接，以强调流的全双工特征。

任一个数的处理模块可以压入流。我们使用术语“压入”，是因为每一新模块总是插到流首之下，而将以前压入的模块下压。（这类似于后进、先出的栈。）图12-6标出了流的两侧，分别称为顺流（downstream）和逆流（upstream）。写到流首的数据将顺流而下传送。由设备驱动程序读到的数据则逆流向上传送。

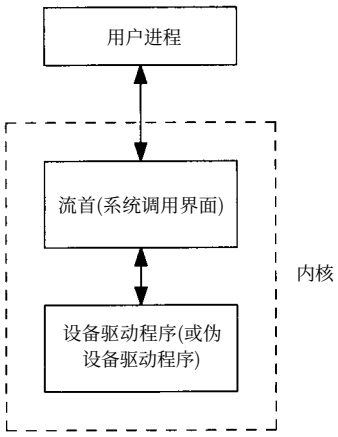

图12-5 一个简单流

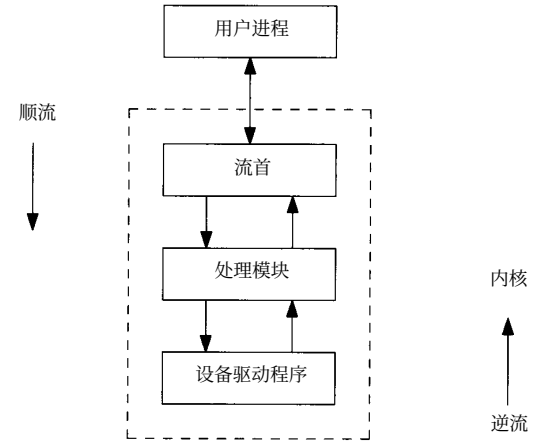

图12-6 具有处理模块的流

流模块是作为内核的一部分执行的，这类似于设备驱动程序，当构造内核时，流模块连编进入内核。大多数系统不允许将未连编进入内核的流模块压入流。

图11-2中示出了基于流的终端系统的一般结构。图中标出的“读、写函数”是流首。标注为“终端行规程”的框是一个流处理模块。该处理模块的实际名称是 ldterm。（各种流模块的手册页在〔AT&T 1990d〕的第7节和〔AT&T 1991〕的第7节中。）

用第3章中说明的函数存取流，它们是：open、close、read、write和ioctl。另外，在SVR3

内核中增加了3个支持流的新函数（getmsg、putmsg和poll），在SVR4中又加了两个处理流不同优先级波段消息的函数（getpmsg和putpmsg）。本节将说明这些新函数，为流打开的路径名通常在/dev目录之下。用ls-l查看设备名，就能判断该设备是否为流设备。所有流设备都是字符特殊文件。

虽然某些有关流的文献暗示我们可以编写处理模块，并将它们压入流中，但是编写这些模块如同编写设备驱动程序一样，需要专门的技术。

> 在流机制之前,终端是用现存的clist机制处理的。(Bach〔1986〕的10.3.1节和Leffler等〔1989〕的9.6节)分别说明SVR2和4.3BSD中的clist 机制。将基于字符的设备添加到内核中通常涉及编写设备驱动程序，将所有有关部分都安排在驱动程序中。对新设备的存取典型地通过原始设备进行，这意味着每个用户的 read,write都直通进入设备驱动程序。流机制使这种交互作用方式更加灵活，条理清晰，使得数据可以用流消息方式在流首和驱动程序之间传送，并使任意数的中间处理模块可对数据进行操作。

12.4.1 流消息

流的所有输入和输出都基于消息。流首和用户进程使用 read、write、getmsg、getpmsg、putmsg和putpmsg交换消息。在流首、各处理模块和设备驱动程序之间，消息可以顺流而下，也可以逆流而上。

在用户进程和流首之间，消息由下列几部分组成：消息类型、可选择的控制信息以及可选择的数据。表12-4列出了对应于write、putmsg和putpmsg的不同参数，所产生的不同消息类型。控制信息和数据存放在strbuf结构中：

```
struct strbuf
  int   maxlen;   /* size of buffer */
  int   len;      /* number of bytes currently in buffer */
  char *buf;      /* pointer to buffer */
};
```

表12-4 为write、putmsg和putpmsg产生的流消息的类型

函 数	控制?	数据?	波 段	标 志	产生的消息类型
write	N/A	是	不使用	不使用	M_DATA (普通)
putmsg	否	否	不使用	0	无消息发送，返回0
putmsg	否	是	不使用	0	M_DATA (普通)
putmsg	是	是或否	不使用	0	M_PROTO (普通)
putmsg	是	是或否	不使用	RS_HIPRI	M_PCPROTO (高优先级)
putmsg	否	是或否	不使用	RS_HIPRI	出错，EINVAL
putpmsg	是或否	是或否	0～255	0	出错，EINVAL
putpmsg	否	否	0～255	MSG_BAND	无消息发送，返回0
putpmsg	否	是	0	MSG_BAND	M_DATA (普通)
putpmsg	否	是	1～255	MSG_BAND	M_DATA (优先级波段)
putpmsg	是	是或否	0	MSG_BAND	M_PROTO (普通)
putpmsg	是	是或否	1～255	MSG_BAND	M_PROTO (优先级波段)
putpmsg	是	是或否	0	MSG_HIPRI	M_PCPROTO (高优先级)
putpmsg	否	是或否	0	MSG_HIPRI	出错，EINVAL
putpmsg	是或否	是或否	非0	MSG_HIPRI	出错，EINVAL

当用putmsg或putpmsg发送消息时，len指定缓存中数据的字节数。当用getmsg或getpmsg接收消息时，maxlen 指定缓存长度（使内核不会溢出缓存），而len则由内核设置，说明存放在缓存中的数据量。0长消息是允许的，len为－1说明没有控制信息或数据。

为什么需要传送控制信息和数据两者呢？提供这两者使我们可以实现用户进程和流之间的服务界面。Olander, McGrath和Israel〔1986〕说明了系统V服务界面的原先实现。AT&T[1990d]第5章详细说明了服务界面，还使用了一个简单的实例。可能最为人了解的服务界面是系统 V的传输层界面(TLI)，它提供了网络系统界面，Stevens〔1990〕第7章对此进行了说明。

控制信息的另一个例子是发送一个无连接的网络消息（数据报）。为了发送该消息，需要说明消息的内容（数据）和该消息的目的地址（控制信息）。如果不能将数据和控制一起发送，那么就要某种专门设计的方案。例如，可以用 ioctl说明地址，然后用write发送数据。另一种技术可能要求：地址占用数据的前N个字节，用write写数据。将控制信息与数据分开，并且提供处理两者的函数（putmsg和getmsg）是处理这种问题的较清晰的方法。

有约25种不同类型的消息，但是只有少数几种用于用户进程和流首之间。其余的则只在内核中顺流、逆流传送。（对于编写流处理模块的人员而言，这些消息是非常有用的，但是对编写用户级代码的人员而言，则可忽略它们。）在我们所使用的函数（ read、write、getmsg、getpmsg、putmsg和putpmsg）中，只涉及三种消息类型，它们是：

- M_DATA（I/O的用户数据）。
- M_PROTO（协议控制信息）。
- M_PCPROTO（高优先级协议控制信息）。

流中的消息都有一个排队优先级：

- 高优先级消息（最高优先级）。
- 优先波段消息。
- 普通消息（最低优先级）。

普通消息是优先波段为0的消息。优先波段消息的波段可在1~255之间，波段愈高，优先级也愈高。

每个流模块有两个输入队列。一个接收来自它上面模块的消息（这种消息从流首向驱动程序顺流传送）。另一个接收来自它下面模块的消息（这种消息从驱动程序向流首逆流传送）。在输入队列中的消息按优先级从高到低排列。表 12-4列出了针对write、putmsg和putpmsg的不同参数，产生不同优先级的消息。

有一些消息我们未加考虑。例如，若流首从它下面接收到 M_SIG消息，则产生一信号。这种方法用于终端行规程模块向相关前台进程组发送终端产生的信号

12.4.2 putmsg和putpmsg函数

putmsg和putpmsg函数用于将流消息（控制信息或数据，或两者）写至流中。这两个函数的区别是后者允许对消息指定一个优先波段。

```
#include <stropts.h>
int putmsg(int filedes, const struct strbuf *ctlptr,
           const struct strbuf *dataptr, int flag);
int putpmsg(int filedes, const struct strbuf *ctlptr,
           const struct strbuf *dataptr, int band, int flag);
```

两个函数返回：若成功则为0，若出错则为-1

对流也可以使用write函数，它等效于不带任何控制信息，*flag*为0的putmsg。

这两个函数可以产生三种不同优先级的消息：普通、优先波段和高优先级。表 12-4详细列出了这两个函数中几个参数的各种可能组合，以及所产生的不同类型的消息。

在表12-4中，消息控制列中的“否”对应于空 *ctlptr*参数，或*ctlptr->len*为－1。该列中的“是”对应于*ctlptr*非空，以及*ctlptr->len*大于等于0。这些说明同样适用于消息的数据部分（用*dataptr*代替*ctlptr*）。

12.4.3 流ioctl 操作

3.14节曾提到过ioctl函数，它能做其他I/O函数不能处理的事情。流系统中继续采用了该函数。

在SVR4中，使用ioctl可对流执行29种不同的操作。关于这些操作的说明请见 streamio(7)手册页〔AT&T1990d〕，头文件<stropts.h>应包括在使用这些操作的C代码中。ioctl的第二个参数*request*说明执行29个操作中的哪一个。所有 *request*都以I_开始。第三个参数则与 *request*有关。有时第三个参数是一个整型值，有时它是指向一个整型或一个数据结构的指针。

实例——isastream函数

有时需要确定一个描述符是否引用一个流。这与调用 isatty函数来确定一个描述符是否引用一个终端设备相类似（见11.9节）。SVR4提供isastream函数。

```
int isastream(int filedes);
```

返回：若是流返回1，否则返回0

（由于某种原因，SVR4的设计者忘记将此函数的原型放在头文件中，所以不能为此函数写一条#include语句。）

与isatty类似，它通常是用一个只对流设备才有效的 ioctl函数来进行测试的。程序12-8是该函数的一种可能的实现。它使用I_CANPUT ioctl来测试由第三个参数说明的优先波段是否可写。如果该ioctl执行成功，则它对所涉及的流并未作任何改变。

程序12-8 检查描述符是否引用流设备

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

int
isastream(int fd)
{
    return(ioctl(fd, I_CANPUT, 0) != -1);
}
```

程序12-9可用于测试此函数。

程序12-9 测试isastream函数

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

int
```

```
main(int argc, char *argv[])
{
    int     i, fd;

    for (i = 1; i < argc; i++) {
        printf("%s: ", argv[i]);
        if ( (fd = open(argv[i], O_RDONLY)) < 0) {
            err_ret("%s: can't open", argv[i]);
            continue;
        }

        if (isastream(fd) == 0)
            err_ret("%s: not a stream", argv[i]);
        else
            err_msg("%s: streams device", argv[i]);
    }
    exit(0);
}
```

运行此程序，得到很多由ioctl函数返回的出错信息。

```
$ a.out /dev/tty /dev/vidadm /dev/null /etc/motd
/dev/tty:/dev/tty:streams device
/dev/vidadm:/dev/vidadm:not a stream:Invalid argument
/dev/null:/dev/null:not a stream:No suck device
/etc/motd: /etc/motd:not a stream:Not a typewriter
```

/dev/tty在SVR之下是个流设备，这与我们所期望的一致。/dev/vidadm不是一个流设备，但是它是支持其他ioctl请求的字符特殊文件。对于不知道这种ioctl请求的设备，它返回EINVAL。/dev/null是一种不支持任何 ioctl操作的字符特殊文件，所以 ioctl返回ENODEV。最后，/etc/motd是一个普通文件，而不是字符特殊文件，所以返回 ENOTTY（这是在这种情况下的典型返回值）。

> ENOTTY（Not a typewriter）是个历史产物，当ioctl企图对并不引用字符特殊设备的描述符进行操作时，UNIX内核都返回ENOTTY。

实例

如果ioctl的参数*request*是I_LIST，则系统返回该流上所有模块的名字，包括最顶端的驱动程序。（指明最顶端的原因是：在多路转接驱动程序的情况下，有多个驱动程序。 AT&T〔1990d〕）第10章讨论了多路转接驱动程序的细节。）其第三个参数应当是指向str_list结构的指针。

```
struct str_list {
  int                 sl_nmods;   /* number of entries in array */
  struct str_mlist      *sl_modlist; /* ptr to first element of array */
};
```

应将sl_modlist设置为指向str_mlist结构数组的第一个元素，将sl_nmods设置为该数组中的项数。

```
struct str_mlist {
  char 1_name[FMNAMESZ+1];   /* null terminated module name */
};
```

常数FMNAMESZ定义在头文件 <sys/conf.h>中，其值常常是 8。l_name的实际长度是

FMNAMESZ+1，增加1个字节是为了存放null终止符。

如果ioctl的第三个参数是0，则该函数返回的是模块数，而不是模块名。我们将先用这种ioctl调用确定模块数，然后再分配所要求的str_mlist结构数。

程序12-10例示了I_LIST操作。由ioctl返回的名字列表并不对模块和驱动程序进行区分，因为该列表的最后一项是处于流底部的驱动程序，所以在打印时将其标明为驱动程序。

程序12-10 打印流中的模块名

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

int
main(int argc, char *argv[])
{
    int                 fd, i, nmods;
    struct str_list     list;

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

    if ( (fd = open(argv[1], O_RDONLY)) < 0)
        err_sys("can't open %s", argv[1]);
    if (isastream(fd) == 0)
        err_quit("%s is not a stream", argv[1]);

            /* fetch number of modules */
    if ( (nmods = ioctl(fd, I_LIST, (void *) 0)) < 0)
        err_sys("I_LIST error for nmods");
    printf("#modules = %d\n", nmods);

            /* allocate storage for all the module names */
    list.sl_modlist = calloc(nmods, sizeof(struct str_mlist));
    if (list.sl_modlist == NULL)
        err_sys("calloc error");
    list.sl_nmods = nmods;

            /* and fetch the module names */
    if (ioctl(fd, I_LIST, &list) < 0)
        err_sys("I_LIST error for list");

            /* print the module names */
    for (i = 1; i <= nmods; i++)
        printf("  %s: %s\n", (i == nmods) ? "driver" : "module",
                             list.sl_modlist++);
    exit(0);
}
```

```
$ who
stevens    console    Sep 25 06:12
stevens    pts001     Oct 12 07:12
$ a.out /dev/pts001
#modules= 4
  module: ttcompat
  module: ldterm
  module: ptem
  driver: pts
$ a.out /dev/console
```

```
#modules= 5
  module: ttcompat
  module: ldterm
  module: ansi
  module: char
  driver: cmux
```

在这两种情形中，顶上的两个流模块都是一样的（ttcompat和ldterm），但是余下的模块和驱动程序则不同。第19章将说明伪终端的情形。

12.4.4 write至流设备

在表12-4中可以看到write至流设备产生一个M_DATA消息。一般而言，这确实如此，但是也还有一些情况需要考虑。首先，流中顶部的一个处理模块规定了可顺流传送的最小、最大数据包长度（无法查询该模块中规定的这些值）。如果write的数据长度超过最大值，则流首将这一数据分解成最大长度的若干数据包。最后一个数据包的长度则小于最大值。

接着要考虑的是：如果向流write 0个字节，又将如何呢？除非流涉及管道或FIFO，否则就顺流发送0长消息。对于管道FIFO，为与以前版本兼容，系统的默认处理方式是忽略0长write。可以用ioctl设置实现管道和FIFO的流写方式以更改这种默认处理方式。

12.4.5 写方式

可以用ioctl取得和设置一个流的写方式。如果将*request*设置为I_GWROPT，第三个参数为指向一个整型变量的指针，则该流的当前写方式就在该整型量中返回。如果将*request*设置为I_SWROPT，第三个参数是一个整型值，则其值就成为该流新的写方式。如同处理文件描述符标志和文件状态标志（见3.13节）一样,总是应当先取当前写方式值,然后修改它,而不只是将写方式设置为某个绝对值(很可能要关闭某些原来打开的位)。

目前，只定义了两个写方式值。

• SNDZERO　对管道和FIFO的0长写会造成顺流传送一个0长消息。按系统默认，0长写不发送消息。

• SNDPIPE　在流上已出错后，若调用write或putmsg，则向调用进程发送SIGPIPE信号。

流也有读方式，我们先说明getmsg和getpmsg函数，然后再说明读方式。

12.4.6 getmsg和getpmsg函数

```
#include <stropts.h>

int getmsg(int filedes, struct strbuf *ctlptr,
           struct strbuf *datptr, int *flagptr);

int getpmsg(int filedes, struct strbuf *ctlptr,
            struct strbuf *datptr, int *bandptr, int *flagptr);
```

两个函数返回：若成功则为非负值，若出错则为-1

注意，*flagptr*和*bandptr*是指向整型的指针。在调用之前，这两个指针所指向的整型单元中应设置成所希望的消息类型，在返回时，此整型量设置为所读到的消息的类型。

如果*flagptr*指向的整型单元的值是0，则getmsg返回流首读队列中的下一个消息。如果下一个消息是高优先权消息，则在返回时，*flagptr*所指向的整型单元设置为RS_HIPRI。如果希望只接收高优先权消息，则在调用getmsg之前必须将*flagptr*所指向的整型单元设置为RS_HIPRI。

getpmsg使用了一个不同的常数集。并且它使用bandptr指明特定的优先波段。

这两个函数有很多条件来确定返回给调用者何种消息：(a)*flagptr*和*bandptr*所指向的值，(b) 流队列中消息的类型，(c) 是否指明非空dataptr和ctlptr, (d) *ctlptr->maxlen*和*dataptr->maxlen*的值。对使用getmsg而言，并不需要了解所有这些细节。如欲了解这些细节请参阅getmsg（2）手册页。

12.4.7 读方式

如果read流设备会发生些什么呢？有两个潜在的问题：(1)如果读到流中消息的记录边界将会怎样？(2)如果调用read，而流中下一个消息有控制信息又将如何？对第一种情况的默认处理方式被称为字节流方式。在这种方式中，read从流中取数据直至满足了要求，或者已经没有数据。在这种方式中，忽略流中消息的边界。对第二种情况的默认处理是，如果在队列的前端有控制消息，则read出错返回。可以改变这两种默认处理方式。

调用ioctl时，若将*request*设置为I_GRDOPT，第三个参数又是指向一个整型单元的指针，则对该流的当前读方式在该整型单元中返回。如果将*request*设置为I_SRDOPT，第三个参数是整型值，则该流的读方式设置为该值。读方式值有下列三个：

- RNORM 普通，字节流方式（与上面说明的相同）。这是默认方式。
- RMSGN 消息不删除方式。读从流中取数据直至读到所要求的字节数，或者到达消息边界。如果某次读只用了消息的一部分，则其余下部分仍留在流中，以供下一个读取。
- RMSGD 消息删除方式。这与不删除方式的区别是，如果某次读只用消息的一部分。则余下部分就被删除，不再使用。

在读方式中还可指定另外三个常数，以便设置在读到流中包含协议信息的消息时read的处理方法：

- RPROTNORM 协议-普通方式。read出错返回，errno设置为EBADMSG。这是默认方式。
- RPROTDAT 协议-数据方式。read将控制部分作为数据返回给调用者。
- RPROTDIS 协议-删除方式。read删除消息中的控制信息，但是返回消息中的数据。

实例

程序12-11是在程序3-3的基础上改写的，它用getmsg代替了read。如果在SVR4之下（其管道和终端都是用流实现的）运行此程序则得：

```
$ echo hello,world | a.out              使用流实现要求的管道
flag=0,ctl.len=-1,dat.len=13
hello,world
flag=0,ctl.len=0,dat.len=0              流挂断
$ a.out                                 使用流实现要求的终端
this is line 1
flag=0,ctl.len=-1,dat.len=15
this is line 1
and line 2
flag=0,ctl.len=-1,dat.len=11
```

```
and line 2
^D                                          键入自定义的终端EOF字符
flag=0,ctl.len=-1,dat.len=0                 文件结尾与挂断不相同
$ a.out < /etc/motd
getmsg error:Not a stream device
```

当管道被关闭时（当echo终止时），它对程序12-11表现为一个流挂断——控制长度和数据长度都设置为0。（14.2节将讨论管道。）但是对于终端，键入文件结束字符，只使返回的数据长度为0。这与挂断并不相同。如所预料的一样，将标准输入重新定向到一个非流设备， getmsg出错返回。

程序12-11 用getmsg将标准输入复制到标准输出

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

#define BUFFSIZE    8192

int
main(void)
{
    int             n, flag;
    char            ctlbuf[BUFFSIZE], datbuf[BUFFSIZE];
    struct strbuf   ctl, dat;

    ctl.buf = ctlbuf;
    ctl.maxlen = BUFFSIZE;
    dat.buf = datbuf;
    dat.maxlen = BUFFSIZE;
    for ( ; ; ) {
        flag = 0;       /* return any message */
        if ( (n = getmsg(STDIN_FILENO, &ctl, &dat, &flag)) < 0)
            err_sys("getmsg error");
        fprintf(stderr, "flag = %d, ctl.len = %d, dat.len = %d\n",
                        flag, ctl.len, dat.len);
        if (dat.len == 0)
            exit(0);
        else if (dat.len > 0)
            if (write(STDOUT_FILENO, dat.buf, dat.len) != dat.len)
                err_sys("write error");
    }
}
```

12.5 I/O多路转接

当从一个描述符读，然后又写到另一个描述符时，可以在下列形式的循环中使用阻塞 I/O：

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

这种形式的阻塞I/O到处可见。但是如果必须读两个描述符又将如何呢？如果仍旧使用阻塞I/O，那么就可能长时间阻塞在一个描述符上，而另一个描述符虽有很多数据却不能得到及时处理。所以为了处理这种情况显然需要另一种不同的技术。

让我们概略地观察一个调制解调器拨号程序的工作情况（该程序将在第 18章中介绍）。该程序读终端（标准输入），将所得数据写到调制解调器上；同时读调制解调器，将所得数据写到终端上（标准输出）。图12-7显示这种工作情况。

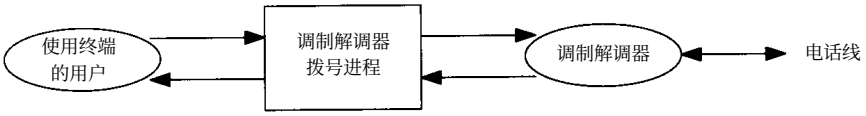

图12-7 调制解调器拨号程序概观

执行这段程序的进程有两个输入，两个输出。如果对这两个输入都使用阻塞 read，那么就可能在一个输入上长期阻塞，而另一个输入的数据则被丢失。

处理这种特殊问题的一种方法是：设置两个进程，每个进程处理一条数据通路。图 12-8中显示了这种安排。

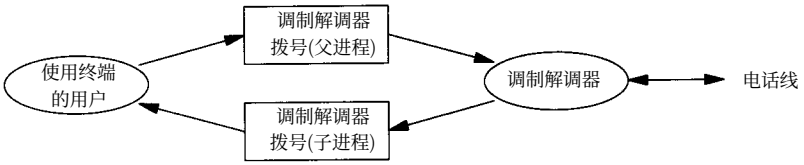

图12-8 使用两个进程实现调制解调器拨号程序

如果使用两个进程，则可使每个进程都执行阻塞 read。但是也产生了这两个进程间相互配合问题。如果子进程接收到文件结束符（由于电话线的一端已经挂断，使调制解调器也挂断），那么该子进程终止，然后父进程接收到 SIGCHLD信号。但是，如若父进程终止（用户在终端上键入了文件结束符），那么父进程应通知子进程停止工作。为此可以使用一个信号（例如 SIGUSR1）。这使程序变得更加复杂。

另一个方式是仍旧使用一个进程执行该程序，但调用非阻塞 I/O读取数据，其基本思想是：将两个输入描述符都设置为非阻塞的，对第一个描述符发一个 read。如果该输入上有数据，则读数据并处理它。如果无数据可读，则 read立即返回。然后对第二个描述符作用样的处理。在此之后，等待若干秒再读第一个描述符。这种形式的循环称为轮询。这种方法的不足之处是浪费CPU时间。大多数时间实际上是无数据可读，但是仍不断反复执行 read，这浪费了CPU时间。在每次循环后要等多长时间再执行下一轮循环也很难确定。轮询技术在支持非阻塞 I/O的系统上都可使用，但是在多任务系统中应当避免使用。

还有一种技术称之为异步 I/O（asynchronous I/O）。其基本思想是进程告诉内核，当一个描述符已准备好可以进行 I/O时，用一个信号通知它。这种技术有两个问题。第一个是并非所有系统都支持这种机制（现在它还不是 POSIX的组成部分，可能将来会是）。SVR4为此技术提供SIGPOLL信号，但是仅当描述符引用流设备时，此信号才能工作。 4.3+BSD有一个类似的信号SIGIO，但也有类似的限制 —— 仅当描述符引用终端设备或网络时才能工作。这种技术的第二个问题是，这种信号对每个进程而言只有 1个。如果使该信号对两个描述符都起作用，那么在接到此信号时进程无法判别是哪一个描述符已准备好可以进行 I/O。为了确定是哪一个描述符已准备好，仍需将这两个描述符都设置为非阻塞的，并顺序试执行 I/O。12.6节将简要说明异步I/O。

一种比较好的技术是使用I/O多路转接（I/O multiplexing）。其基本思想是：先构造一张有关描述符的表，然后调用一个函数，它要到这些描述符中的一个已准备好进行 I/O时才返回。在返回时，它告诉进程哪一个描述符已准备好可以进行 I/O。

> I/O多路转接至今还不是POSIX的组成部分。SVR4和4.3+BSD都提供select函数以执行I/O多路转接。poll函数只由SVR4提供。SVR4实际上用poll实现select。
>
> I/O多路转接在4.2+BSD中是用select函数提供的。虽然该函数主要用于终端I/O和网络I/O，但它对其他描述符同样是起作用的。SVR3在增加流机制时增加了poll函数。但在SVR4之前，poll只对流设备起作用。SVR4支持对任一描述符起作用的poll。

select和poll的可中断性

中断的系统调用的自动再起动是由4.2+BSD引进的（见10.5节），但当时select函数是不再起动的。这种特性延续到4.3+BSD，即使指定了SA_RESTART也是为此。但是，在SVR4之下，如果指定了SA_RESTART，那么select和poll也是自动再起动的。为了将软件移植到SVR4时阻止这一点，如果信号可能中断select或poll，则总是使用signal_intr函数（见程序10-13）。

12.5.1 select函数

select函数使我们在SVR4和4.3+BSD之下可以执行I/O多路转接，传向select的参数告诉内核：

(1) 我们所关心的描述符。

(2) 对于每个描述符我们所关心的条件（是否读一个给定的描述符？是否想写一个给定的描述符？是否关心一个描述符的异常条件？）。

(3) 希望等待多长时间（可以永远等待，等待一个固定量时间，或完全不等待）。

从select返回时，内核告诉我们：

(1) 已准备好的描述符的数量。

(2) 哪一个描述符已准备好读、写或异常条件。

使用这种返回值，就可调用相应的I/O函数（一般是read或write），并且确知该函数不会阻塞。

```
#include <sys/types.h>/* fd_set data type */
#include <sys/time.h> /* struct timeval */
#include <unistd.h>   /* function prototype might be here */
int select (int maxfdp1, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
            struct timeval *tvptr);
```

返回：准备就绪的描述符数，若超时则为0，若出错则为-1

先说明最后一个参数，它指定愿意等待的时间。

```
struct  timeval{
    long  tv_sec;  /* seconds */
    long  tv_usec; /* and microseconds */
};
```

有三种情况：

• *tvptr*= =NULL

永远等待。如果捕捉到一个信号则中断此无限期等待。当所指定的描述符中的一个已准备好或捕捉到一个信号则返回。如果捕捉到一个信号，则select返回－1， errno设置为EINTR。

• *tvptr->tv_sec*= =0 && *tvptr->tv_usec*= =0

完全不等待。测试所有指定的描述符并立即返回。这是得到多个描述符的状态而不阻塞select函数的轮询方法。

• *tvptr->tv_sec* ! =0 | | *tvptr->tv_usec*! =0

等待指定的秒数和微秒数。当指定的描述符之一已准备好，或当指定的时间值已经超过时立即返回。如果在超时时还没有一个描述符准备好，则返回值是 0，（如果系统不提供微秒分辨率，则*tvptr->tv_usec*值取整到最近的支持值。）与第一种情况一样，这种等待可被捕捉到的信号中断。

中间三个参数*readfds*、*writefds*和*exceptfds*是指向描述符集的指针。这三个描述符集说明了我们关心的可读、可写或处于异常条件的各个描述符。每个描述符集存放在一个 fd_set数据类型中。这种数据类型的实现可见图12-9，它为每一可能的描述符保持了一位。

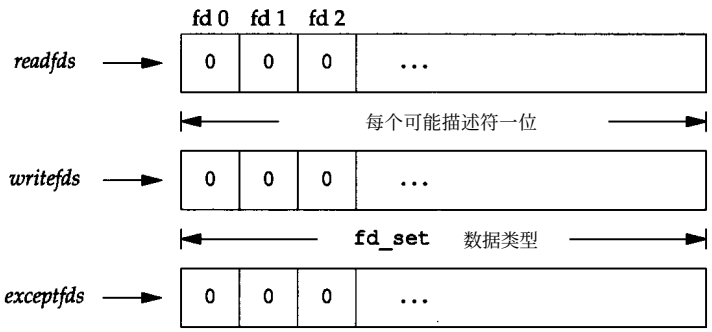

图12-9 对select指定读、写和异常条件描述符

对fg_set数据类型可以进行的处理是：(a)分配一个这种类型的变量，(b)将这种类型的一个变量赋与同类型的另一个变量，或(c)对于这种类型的变量使用下列四个宏：

```
FD_ZERO(fd_set *fdset);           /* clear all bits in fdset */
FD_SET(int fd, fd_set *fdset);    /* turn on bit for fd in fdset */
FD_CLR(int fd, fd_set *fdset);    /* turn off bit for fd in fdset */
FD_ISSET(int fd, fd_set *fdset);  /* test bit for fd in fdset */
```

以下列方式说明了一个描述符集后：

```
fd_set  rset;
int     fd;
```

必须用FD_ZERO清除其所有位：

```
FD_ZERO (&rset);
```

然后在其中设置我们关心的各位：

```
FD_SET (fd,&rset);
FD_SET (STDIN_FILENO,&rset);
```

从select返回时，用FD_ISSET测试该集中的一个给定位是否仍旧设置：

```
if (FD_ISSET(fd, &rset)){
...
}
```

select中间三个参数中的任意一个（或全部）可以是空指针，这表示对相应条件并不关心。如果所有三个指针都是空指针，则select提供了较sleep更精确的计时器（回忆10.19节，sleep等待整数秒，而对于select，其等待的时间可以小于1秒；其实际分辨率取决于系统时钟。）习题12.6给出了这样一个函数。

select第一个参数*maxfdp1*的意思是“最大fd加1（max fd plus 1）”。在三个描述符集中找出最高描述符编号值，然后加1，这就是第一个参数值。也可将第一个参数设置为FD_SETSIZE，这是一个<sys/types.h>中的常数，它说明了最大的描述符数（经常是256或1024）。但是对大多数应用程序而言，此值太大了。确实，大多数应用程序只应用3~10个描述符。如果将第三个参数设置为最高描述符编号值加1，内核就只需在此范围内寻找打开的位，而不必在数百位的大范围内搜索。

例如，若编写下列代码：

```
fd_set readset, writeset;

FD_ZERO(&readset);
FD_ZERO(&writeset);

FD_SET(0, &readset);
FD_SET(3, &readset);
FD_SET(1, &writeset);
FD_SET(2, &writeset);

select (4, &readset, &writeset, NULL, NULL);
```

然后，图12-10显示了这两个描述符集的情况。

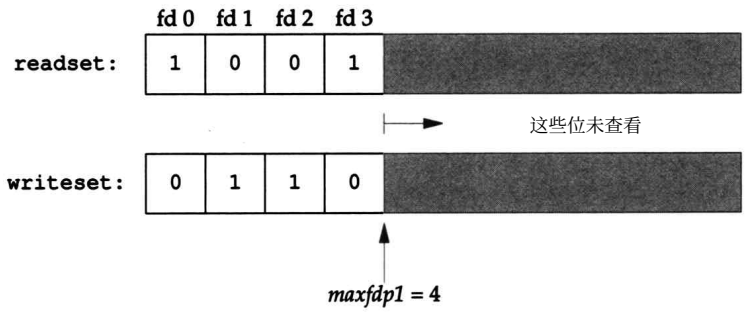

图12-10　select的样本描述符集

因为描述符编号从0开始，所以要在最大描述符编号值上加1。第一个参数实际上是要检查的描述符数（从描述符0开始）。

select有三个可能的返回值。

(1) 返回值－1表示出错。这是可能发生的，例如在所指定的描述符都没有准备好时捕捉到一个信号。

(2) 返回值0表示没有描述符准备好。若指定的描述符都没有准备好，而且指定的时间已经超过，则发生这种情况。

(3) 返回一个正值说明了已经准备好的描述符数，在这种情况下，三个描述符集中仍旧打开的位是对应于已准备好的描述符位。

注意，除非返回正值，否则在返回后检查描述符集是没有意义的。若捕捉到信号或计时器超时，那么描述符集的值取决于实现。确实，若计时器超时，4.3+BSD并不改变描述符集，而SVR4则清除描述符集。

在SVR4和BSD的select实现之间，有另一些差异。BSD系统总是返回每一个集中准备就绪的描述符数之和。若两个集中的同一描述符准备就绪（例如，读集和写集），则该描述符计两次。不幸，SVR4更改了这一点，若同一描述符在多个集中准备就绪，该描述符只计一次。这再一次显示了我们将会碰到的问题，直至POSIX标准化了select这样的函数才能解决此问题。

对于“准备好”的意思要作一些更具体的说明：

(1) 若对读集（*readfds*）中的一个描述符的read不会阻塞，则此描述符是准备好的。

(2) 若对写集（*writefds*）中的一个描述符的write不会阻塞，则此描述符是准备好的。

(3) 若对异常条件集（*exceptfds*）中的一个描述符有一个未决异常条件，则此描述符是准备好的。现在，异常条件包括：(a)在网络连接上到达指定波特率外的数据，或者 (b)在处于数据包方式的伪终端上发生了某些条件。（Stevens〔1990〕的15.10节中说明了这种条件。）

应当理解一个描述符阻塞与否并不影响 select是否阻塞。也就是说，如果希望读一个非阻塞描述符，并且以超时值为5秒调用select，则select最多阻塞5秒。相类似，如果指定一个无限的超时值，则select阻塞到对该描述符数据准备好，或捕捉到一个信号。

如果在一个描述符上碰到了文件结束，则select认为该描述符是可读的。然后调用read，它返回0，这是UNIX指示到达文件结尾处的方法。（很多人错误地认为，当到达文件结尾处时，select会指示一个异常条件。）

12.5.2 poll函数

SVR4的poll函数类似于select，但是其调用形式则有所不同。我们将会看到，poll与流系统紧紧相关，虽然在SVR4中，可以对任一描述符都使用该函数。

```
#include <stropts.h>
#include <poll.h>

int poll(struct pollfd fdarray[],unsigned long nfds,int timeout);
```

返回：准备就绪的描述符数，若超时则为0，若出错则为-1

与select不同，poll不是为每个条件构造一个描述符集，而是构造一个 pollfd结构数组，每个数组元素指定一个描述符编号以及对其所关心的条件。

```
struct pollfd {
  int    fd;       /* file descriptor to check, or < 0 to ignore */
  short  events;   /* events of interest on fd */
  short  revents;  /* events that occurred on fd */
};
```

*fdarray*数组中的元素数由*nfds*说明。

由于某种未知的原因，SVR3说明*nfds*的类型为unsigned long，这似乎是太大了。在SVR4手册poll的原型中，第二个参数的数据类型为 size_t，但在<poll.h>包含的实际原型中，第二个参数的数据类型仍说明为 unsigned long。

SVR4的SVID〔AT&T1989〕说明poll的第一个参数是struct pollfd *fdarray*[ ],而SVR4手册页〔AT&T 1990 d〕则说明该参数为struct pollfd **fdarray*。在C语言中，这两种说明是等价的。我们使用第一种说明以重申 *fdarray*指向一个结构数组，而不是指向单个结构的指针。

应将events成员设置为表12-5中所示值的一个或几个。通过这些值告诉内核我们对该描述符关心的是什么。返回时，内核设置revents成员，以说明对该描述符发生了什么事件。（注意，poll没有更改events成员，这与select不同，select修改其参数以指示哪一个描述符已准备好了。）

表12-5 poll的events和revents标志

名 称	对events的输入	从revents得到的结果	说 明
POLLIN	•	•	可读除高优级外的数据，不阻塞
POLLRDNORM	•	•	可读普通(优先波段0 数据，不阻塞
POLLRDBAND	•	•	可读0优先波段数据，不阻塞
POLLPRI	•	•	可读高优先级数据，不阻塞
POLLOUT	•	•	可与普通数据，不阻塞
POLLWRNORM	•	•	与POLLOUT相同
POLLWRBAND	•	•	可写非0优先波段数据，不阻塞
POLLERR		•	已出错
POLLHUP		•	已挂起
POLLNVAL		•	此描述符并不引用一打开文件

表12-5中头四行测试可读性，接着三行测试可写性，最后三行则是异常条件。最后三行是由内核在返回时设置的。即使在 events字段中没有指定这三个值，如果相应条件发生，则在revents中也返回它们。

当一个描述符被挂断后（POLLHUP），就不能再写向该描述符。但是仍可能从该描述符读取到数据。

poll的最后一个参数说明我们想要等待多少时间。如同 select一样，有三种不同的情形：

• *timeout* == INFTIM　永远等待。常数INFTIM定义在<stropts.h>,其值通常是－1。当所指定的描述符中的一个已准备好，或捕捉到一个信号则返回。如果捕捉到一个信号，则poll返回－1，errno设置为EINTR。

• *timeout* == 0　不等待。测试所有描述符并立即返回。这是得到很多个描述符的状态而不阻塞poll函数的轮询方法。

• *timeout* > 0　等待*timeout*毫秒。当指定的描述符之一已准备好，或指定的时间值已超过时立即返回。如果已超时但是还没有一个描述符准备好，则返回值是 0。（如果系统不提供毫秒分辨率，则*timeout*值取整到最近的支持值。）

应当理解文件结束与挂断之间的区别。如果正在终端输入数据，并键入文件结束字符，POLLIN被打开，于是就可读文件结束指示（read返回0）。POLLHUP在revents中没有打开。如果读调制解调器，并且电话线已挂断，则在 revents中将接到POLLHUP。

与select一样，不论一个描述符是否阻塞，并不影响poll是否阻塞。

12.6 异步I/O

使用select和poll可以实现异步I/O。关于描述符的状态,系统并不主动告诉我们任何信息,我们需要主动地进行查询（调用 select或poll）。如在第10章中所述，信号机构提供一种异步形式的通知某种事件已发生的方法。SVR4和4.3+BSD提供了使用一个信号（在SVR4中是SIGPOLL，在4.3+BSD中是SIGIO）的异步I/O方法，该信号通知进程，对某个描述符所关心的某个事件已经发生。

> 我们已了解到在 SVR4中select和poll对任意描述符都能工作。在 4BSD中，select对任意描述符都能工作。但是关于异步 I/O却有限制。在SVR4中，异步I/O只对设备起作用。在4.3+BSD中，异步I/O只对终端和网络起作用。

SVR4和4.3+BSD所支持的异步I/O的一个限制是每个进程只有一个信号。如果要对几个描述符进行异步I/O，那么在进程接收到该信号时并不知道这一信号对应于哪一个描述符。

12.6.1 SVR4

在系统V中，异步 I/O是流系统的一部分。它只对流设备起作用。 SVR4异步 I/O信号是SIGPOLL。

为了对一个流设备起动异步 I/O，需要调用 ioctl，而其第二个参数（ *request*）则为I_SETSIG。第三个参数则是由表12-6中一个或多个常数构成的整型值。这些常数在 <stropts.h> 中定义。

表12-6 产生SIGPOLL信号的条件

常 数	说 明
S_INPUT	非高优先级的消息已到达
S_RDNORM	一普通消息已到达
S_RDBAND	一0优先波段消息已到达
S_BANDURG	若此常数说明为S_RDBAND，则当一非0优先波段消息到达时产生SIGURG信号而非SIGPOLL
S_HIPRI	一高优先级消息已到达
S_OUTPUT	写队列不再满
S_WRNORM	与S_OUTPUT一样
S_WRBAND	可发送一非0优先波段消息
S_MSG	包含SIGPOLL信号的流信号消息已到达
S_ERROR	M_ERROR消息已到达
S_HANGUP	M_HANGUP消息已到达

表12-6中“ 已到达 ”的意思是“ 已到达流首的读队列 ”。

除了调用ioctl以说明产生SIGPOLL信号的条件，也应为该信号建立一个信号处理程序。回忆表10-1，对于SIGPOLL的默认动作是终止该进程，所以应在调用ioctl之前建立信号处理程序。

12.6.2 4.3+BSD

在4.3+BSD中，异步I/O是两个信号SIGIO和SIGURG的组合。前者是通用异步I/O信号，后者则只被用来通知进程在网络连接上到达了非规定波特率的数据。

为了接收SIGIO信号，需执行下列三步：

(1) 调用signal或sigaction为该信号建立一个信号处理程序。

(2) 以命令F_SETOWN（见3.13节）调用fcntl来设置进程ID和进程组ID，它们将接收对于该描述符的信号。

(3) 以命令F_SETFL调用fcntl设置O_ASYNC状态标志，使在该描述符上可以进行异步 I/O（见表3-2）。

第(3)步仅用于指向终端或网络的描述符，这是4.3+BSD异步传输设施的一个基本的限制。

对于SIGURG信号，只需执行第(1)步和第(2)步。该信号仅对于指向支持带外数据的网络连接的描述符而产生。

12.7 readv和writev函数

readv和writev函数用于在一个函数调用中读、写多个非连续缓存。有时也将这两个函数称为散布读（*scatter read*）和聚集写（*gather write*）。

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

ssize_t readv(int filedes, const struct iovec iov[ ], int iovcnt);

ssize_t writev(int filedes, const struct iovec iov[ ], int iovcnt);
```

两个函数返回：已读、写的字节数，若出错则为 -1

这两个函数的第二个参数是指向iovec结构数组的一个指针：

```
struct iovec {
  void   *iov_base;   /* starting address of buffer */
  size_t  iov_len;    /* size of buffer */
};
```

*iov*数组中的元素数由*iovcnt*说明。

> 这两个函数始于4.2BSD，现在SVR4也提供它们。
>
> 这两个函数的原型以及它们使用的 iovec结构在各种有关文献资料中略有差别。如果比较它们在 SVR4程序员手册〔AT&T 1990e〕、SVR4的SVID〔AT&T 1989〕，以及SVR4和4.3+BSD<sys/uio.h>头文件中的定义，那么它们之间都有差别。其部分原因是： SVID和SVR4程序员手册对应于 1988 POSIX.1标准，而非1990版。上面示出的原型和结构定义对应于 read和write的POSIX.1定义：缓存地址是void *，缓存长度是size_t,返回值是ssize_t。
>
> 注意，readv的第二个参数被说明为 const。这取自4.3+BSD中该函数的原型，在SVR4的手册中则无此修饰词。对于 readv此修饰词有效，因为并不修改 iovec结构的成员——此函数只修改iov_base所指向的存储区。
>
> 4.3BSD和SVR4将*iovcnt*限制为16。4.3+BSD定义了常数UIO_MAXIOV，当前其值定义为 1024。SVID声称常数 IOV_MAX提供了系统 V限制，但是它没有在SVR4的头文件中定义。

图12-11显示了readv和writev的参数和iovec结构之间的关系。 writev以顺序*iov*[0], *iov*[1]至

iov[iovcnt-1] 从缓存中聚集输出数据。writev返回输出的字节总数，它应等于所有缓存长度之和。

readv则将读入的数据按上述同样顺序散布到缓存中。 readv总是先填满一个缓存，然后再填写下一个。readv返回读得的总字节数。如果遇到文件结尾，已无数据可读，则返回0。

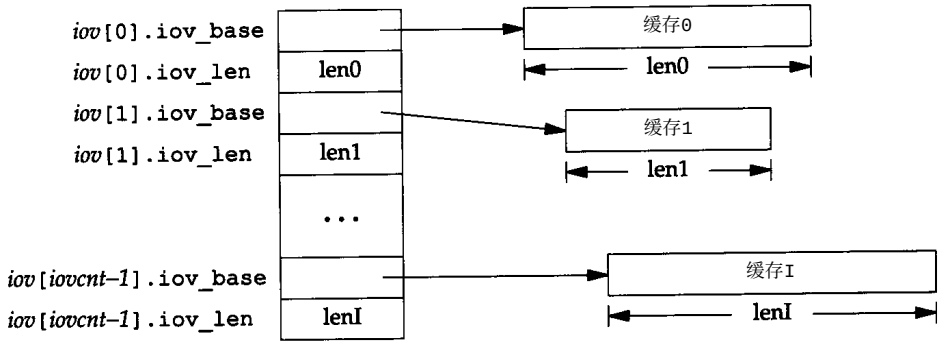

图12-11 readv和writev的iovec结构

实例

在16.7节的_db_writeidx函数中，需将两个缓存连续地写到一个文件中。第二个缓存是调用者传递过来的一个参数，第一个缓存是我们创建的，它包含了第二个缓存的长度，文件中其他信息的位移量。有三种方法可以实现这一要求：

(1) 调用wrtre两次,一次一个缓存。

(2) 分配一个大到足以包含两个缓存的新缓存。将两个缓存的内容复制到新缓存中。然后对该缓存调用wrtre一次。

(3) 调用writev输出两缓存。

16.7节中使用了writev。将它与另外两种方法进行比较，对我们很有启发。表 12-7显示了上面所述三种方法的结果。

表12-7 比较writev和其他技术所得的时间结果

操 作	SPARC			80386		
	用户	系统	时钟	用户	系统	时钟
两次write	0.2	7.2	17.2	0.5	13.1	13.7
缓存复制，然后一次write	0.5	4.4	17.2	0.7	7.3	8.1
一次write	0.3	4.6	17.1	0.3	7.8	8.2

所用的测试程序输出100字节的头文件，接着又输出200字节的数据。这样做10 000次，产生了一个百万字节的文件。该程序按上面所述方法写了 3个版本，各运行一次，测得它们各使用的用户CPU时间、系统CPU时间和时钟时间。它们的单位都是秒。

正如我们所预料的，调用 write两次的系统时间是调用 write一次或调用writev一次的两倍，这与表3-1的结果类似。

要注意的是：CPU时间（用户加系统）几乎是个常数，无论是在缓存复制后用一个 write还是用一个writev。两者的区别只是在用户空间（缓存复制）或系统空间（ writev）下执行的时间多一点。在SPARC系统上运行时，其和是4.9秒，而在80386系统下运行则是约8.0秒。

对于表12-7最后要说明的是，在SPARC上本测试所用的时钟时间主要用于磁盘数据传输上，

而在386系统上则主要用于CPU方面。

总而言之，我们一般采用readv和writev，而不采用多次read和write。时间结果表明采用缓存复制后用一个write与采用一个writev所用CPU时间几乎一样，但一般说来，因为前者还需要分配一个临时缓存用于存储及复制，所以后者更复杂。

12.8 readn和writen函数

某些设备,特别是终端、网络和SVR4的流设备有下列两种性质：

(1) 一次read操作所返回的数据可能少于所要求的数据，即使还没达到文件尾端。这不是一个错误，应当继续读该设备。

(2) 一次write操作的返回值也可能少于指定输出的字节数。这可能是由若干因素造成的，例如，下游模块的流量控制限制。这也不是错误，应当继续写余下的数据至该设备。(通常，只有对非阻塞描述符，或捕捉到一个信号时，才发生这种write返回。)

在读、写磁盘文件时没有这两种性质。

在第18章中，我们将写一个流管道（基于SVR4流或BSD UNIX域套接口），其中需要考虑这些特性。下面两个函数readn和writen的功能是读、写指定的N字节数据，并处理返回值小于要求值的情况。这两个函数只是按需多次调用read和write直至读、写了N字节数据。

```
#include"ourhdr.h"

ssize_t readn(int filedes, void *buff, size_t nbytes);

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

两个函数返回：已读、写字节数，若出错则为-1

在要将数据写到上面提到的设备上时，就可调用writen，但是仅当先就知道要接收数据的数量时，才调用readn（通常，调用read以接收来自这些设备的数据）。

程序12-12、程序12-13是writen和readn的一种实现。

程序12-12 writen函数

```
#include    "ourhdr.h"

ssize_t                     /* Write "n" bytes to a descriptor. */
writen(int fd, const void *vptr, size_t n)
{
    size_t      nleft;
    ssize_t     nwritten;
    const char  *ptr;

    ptr = vptr; /* can't do pointer arithmetic on void* */
    nleft = n;
    while (nleft > 0) {
        if ( (nwritten = write(fd, ptr, nleft)) <= 0)
            return(nwritten);       /* error */

        nleft -= nwritten;
        ptr   += nwritten;
    }
    return(n);
}
```

程序12-13 readn函数

```
#include    "ourhdr.h"

ssize_t                     /* Read "n" bytes from a descriptor. */
readn(int fd, void *vptr, size_t n)
{
    size_t  nleft;
    ssize_t nread;
    char    *ptr;

    ptr = vptr;
    nleft = n;
    while (nleft > 0) {
        if ( (nread = read(fd, ptr, nleft)) < 0)
            return(nread);      /* error, return < 0 */
        else if (nread == 0)
            break;              /* EOF */

        nleft -= nread;
        ptr   += nread;
    }
    return(n - nleft);      /* return >= 0 */
}
```

12.9 存储映射I/O

存储映射I/O使一个磁盘文件与存储空间中的一个缓存相映射。于是当从缓存中取数据，就相当于读文件中的相应字节。与其类似，将数据存入缓存，则相应字节就自动地写入文件。这样，就可以在不使用read和write的情况下执行I/O。

为了使用这种功能，应首先告诉内核将一个给定的文件映射到一个存储区域中。这是由mmap函数实现的。

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

caddr_t mmap(caddr_t addr, size_t len, int prot, int flag,
             int filedes, off_t off);
```

返回：若成功则为映射区的起始地址，若出错则为 -1

存储映照I/O已经用了很多年。4.1BSD（1981）以其vread和vwrite函数提供了一种不同形式的存储映射I/O。4.2BSD没有使用这两个函数，而是希望使用mmap函数。但是由于Leffler等[1989]2.5节中说明的理由，4.2BSD实际并没有包含mmap函数。Gingell,Moran和Shannon〔1987〕说明了mmap的一种实现。现在，SVR4和4.3+BSD都支持mmap函数。

数据类型caddr_t通常定义为char *。*addr*参数用于指定映射存储区的起始地址。通常将其设置为0，这表示由系统选择该映射区的起始地址。此函数的返回地址是：该映射区的起始地址。

*filedes*指定要被映射文件的描述符。在映射该文件到一个地址空间之前，先要打开该文件。*len*是映射的字节数。*off*是要映射字节在文件中的起始位移量（下面将说明对*off*值有某些限制）。

在说明其余参数之前，先看一下存储映射文件的基本情况。图 12-12显示了一个存储映射文件。（见图7-3中进程存储空间的典型安排情况。）

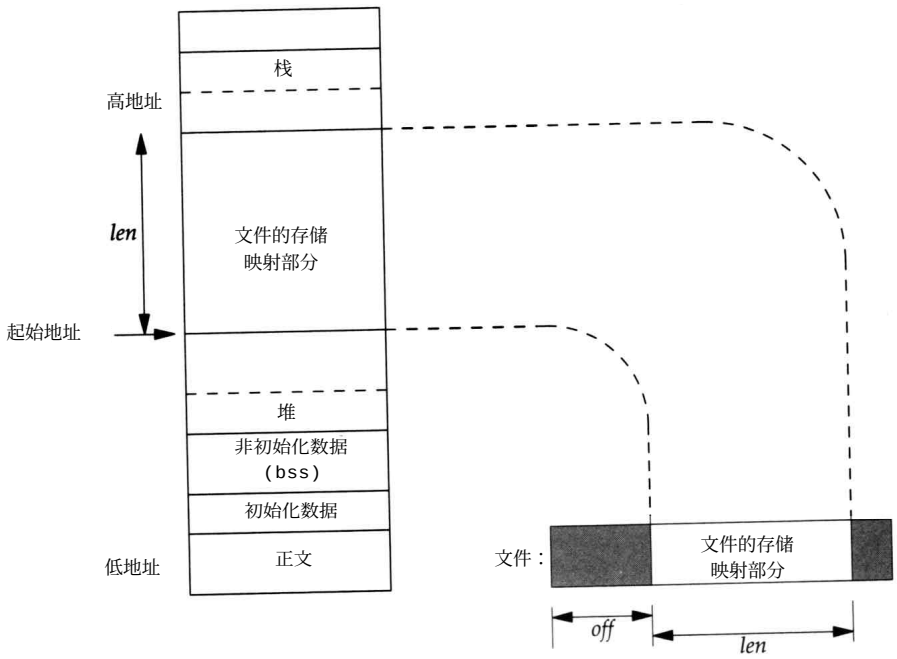

图12-12　存储映射文件的例子

在此图中，“起始地址”是mmap的返回值。在图中，映射存储区位于堆和栈之间：这属于实现细节，各种实现之间可能不同。

*proc*参数说明映射存储区的保护要求。见表12-8。

表12-8　存储映射区的保护

prot	说　明
PROT_READ	区域可读
PROT_WRITE	区域可写
PROT_EXEC	区域可执行
PROT_NONE	区域可存取(4.3+BSD无)

对于映射存储区所指定的保护要求与文件的open方法匹配。例如，若该文件是只读打开的，那么对映射存储区就不能指定PROT_WRITE。

*flag*参数影响映射存储区的多种属性：

• MAP_FIXED　返回值必须等于*addr*。因为这不利于可移植性，所以不鼓励使用此标志。如果未指定此标志，而且*addr*非0，则内核只把*addr*视为何处设置映射区的一种建议。

通过将*addr*指定为0可获得最大可移植性。

• MAP_SHARED　这一标志说明了本进程对映射区所进行的存储操作的配置。此标志指定存储操作修改映射文件 —— 也就是，存储操作相当于对该文件write。必须指定本标志或下一个标志（MAP_PRIVATE）。

• MAP_PRIVATE　本标志说明，对映射区的存储操作导致创建该映射文件的一个副本。所有后来对该映射区的存访都是存访该副本，而不是原始文件。

4.3+BSD还有另外一些MAP_xxx标志值，它们是这种现实所特有的。详细情况请参见4.3+BSD mmap（2）手册页。

*off*和*addr*的值（如果指定了MAP_FIXED）通常应当是系统虚存页长度的倍数。在SVR中，

虚存页长度可用带参数SC_PAGESIZE的sysconf函数（见2.5.4节）得到。在4.3+BSD之下，页长度由头文件<sys/param.h>中的常数NBPG定义。因为*off*和*addr*常常指定为0，所以这种要求一般并不是问题。

因为映射文件的起动位移量受系统虚存页长度的限制，那么如果映射区的长度不是页长度的整数倍时，将如何呢？假定文件长12字节，系统页长为512字节，则系统通常提供512字节的映射区，其中后500字节被设为0。可以修改这500字节，但任何变动都不会在文件中反映出来。

与映射存储区相关有两个信号：SIGSEGV和SIGBUS。信号SIGSEGV通常用于指示进程试图存取它不能存取的存储区。如果进程企图存数据到用 mmap指定为只读的映射存储区，那么也产生此信号。如果存取映射区的某个部分，而在存取时这一部分已不存在，则产生 SIGBUS信号。例如，用文件长度映射一个文件，但在存访该映射区之前，另一个进程已将该文件截短。此时，如果进程企图存取对应于该文件尾端部分的映射区，则接收到 SIGBUS信号。

在fork之后，子进程继承存储映射区（因为子进程复制父进程地址空间，而存储映射区是该地址空间中的一部分），但是由于同样的理由，exec后的新程序则不继承此存储映射区。

进程终止时，或调用了 munmap之后，存储映射区就被自动去除。关闭文件描述符 *filedes* 并不解除映射区。

```
#include <sys/types.h>
#include <sys/mman.h>
int munmap(caddr_t addr, size_t len);
```

返回：若成功则为0，若出错则为-1

munmap 并不影响被映射的对象 —— 也就是说，调用munmap并不使映射区的内容写到磁盘文件上。对于MAP_SHARED区磁盘文件的更新，在写到存储映射区时按内核虚存算法自动进行。

> 某些系统提供了一个msync函数，它类似于fsync函数（见4.24节），但对存储映射区起作用。

实例

程序12-14用存储映射I/O复制一个文件（类似于cp(1)命令）。首先打开两个文件，然后调用fstat得到输入文件的长度。在调用 mmap和设置输出文件长度时都需使用输入文件长度。调用lseek，然后写一个字节以设置输出文件的长度。如果不设置输出文件的长度，则对输出文件调用mmap也可以，但是对相关存储区的第一次存访会产生 SIGBUS。也可使用ftruncate函数来设置输出文件的长度，但是并非所有系统都支持该函数扩充文件长度（见 4.13节）。

然后对每个文件调用mmap，将文件映射到存储区，最后调用 memcpy将输入缓存的内容复制到输出缓存。在从输入缓存（ src）取数据字节时，内核自动读输入文件；在将数据存入输出缓存（dst）时，内核自动将数据写到输出文件中。

程序12-14 用存储映射I/O复制文件

```
#include    <sys/types.h>
#include    <sys/stat.h>
#include    <sys/mman.h>    /* mmap() */
#include    <fcntl.h>
#include    "ourhdr.h"
```

```
#ifndef MAP_FILE    /* 4.3+BSD defines this & requires it to mmap files */
#define MAP_FILE    0   /* to compile under systems other than 4.3+BSD */
#endif

int
main(int argc, char *argv[])
{
    int         fdin, fdout;
    char        *src, *dst;
    struct stat statbuf;

    if (argc != 3)
        err_quit("usage: a.out <fromfile> <tofile>");

    if ( (fdin = open(argv[1], O_RDONLY)) < 0)
        err_sys("can't open %s for reading", argv[1]);

    if ( (fdout = open(argv[2], O_RDWR | O_CREAT | O_TRUNC,
                                                FILE_MODE)) < 0)
        err_sys("can't creat %s for writing", argv[1]);

    if (fstat(fdin, &statbuf) < 0)  /* need size of input file */
        err_sys("fstat error");

            /* set size of output file */
    if (lseek(fdout, statbuf.st_size - 1, SEEK_SET) == -1)
        err_sys("lseek error");
    if (write(fdout, "", 1) != 1)
        err_sys("write error");

    if ( (src = mmap(0, statbuf.st_size, PROT_READ,
                    MAP_FILE | MAP_SHARED, fdin, 0)) == (caddr_t) -1)
        err_sys("mmap error for input");

    if ( (dst = mmap(0, statbuf.st_size, PROT_READ | PROT_WRITE,
                    MAP_FILE | MAP_SHARED, fdout, 0)) == (caddr_t) -1)
        err_sys("mmap error for output");

    memcpy(dst, src, statbuf.st_size);  /* does the file copy */

    exit(0);
}
```

将存储区映射复制与用read，write进行的复制（缓存长度为8192）相比较，得到表12-9中所示的结果。

表12-9 read/write与mmap/memcpy比较的时间结果

操 作	SPARC			80336		
	用户	系统	时钟	用户	系统	时钟
read/write	0.0	2.6	11.0	0.0	5.3	11.2
mmap/memcpy	0.9	1.7	3.7	0.3	2.7	5.7

时间单位是秒，被复制文件的长度是约3百万字节。

对于SPARC，两种复制方式的CPU时间（用户+系统）相同，都是2.6秒。（这与表12-7中writev的情况类似）。对于386，mmap/memcpy方式大约是read/write方式的一半。

使用mmap时，SPARC和386系统时间都减少的原因是：内核直接对映射存储缓存作I/O操作。而在read/write方式，内核要在用户缓存和它自己的缓存之间进行复制，然后用其缓存作I/O。

另一个要注意的是，使用mmap/memcpy时，时钟时间至少减半。

将一个普通文件复制到另一个普通文件中时，存储映射 I/O比较快。但是有一些限制，如不能用其在某些设备之间（例如网络设备或终端设备）进行复制，并且对被复制的文件进行映

射后，也要注意该文件的长度是否改变。尽管如此，有很多应用程序会从存储映射 I/O得到好处，因为它处理的是存储空间而不是读、写一个文件，所以常常可以简化算法。从存储映射I/O中得益的一个例子是帧缓存设备，该设备引用一个位 -映射显示。

Krieger,Stumm和Unrau〔1992〕第5章说明了一个使用存储映射I/O的标准I/O库。

14.9节将返回到存储映射I/O，其中有一个例子，说明在SVR4和4.3+BSD之下如何使用存储映射I/O在有关进程间提供共享存储区。

12.10 小结

本章说明了很多高级I/O功能，其中大多数将在后面章节的例子中使用：

- 非阻塞I/O —— 发一个I/O操作，不使其阻塞。
- 记录锁。
- 系统V流机制。
- I/O多路转接 —— select和poll函数。
- readv和writev函数。
- 存储空间映射I/O（mmap）。

习题

12.1 删除程序12-6for循环中第二次调用write的语句后结果如何，为什么？

12.2 查看系统中<sys/types.h>头文件，并研究select和四个FD_宏的实现。

12.3 <sys/types.h>头文件中定义了fd_set数据类型可以处理的最大描述符数，假设需要将描述符数增加到2048，该如何实现？

12.4 比较处理信号量集的函数（见10.11节）和fd_set描述符集的函数，并研究在你的系统上实现它们的方法。

12.5 getmsg可以返回多少种不同的信息？

12.6 用select或poll实现一个与sleep类似的函数sleep_us，不同之处是要等待指定的若干微秒。比较这个函数和BSD中的usleep函数。

12.7 是否可以利用建议性锁来实现程序 10-17中的函数TELL_WAIT、TELL_PARENT、TELL_CHILD、WAIT_PARENT以及WAIT_CHILD？如果可以，编写这些函数并测试其功能。

12.8 用select或poll测试管道的容量。将其值与第2章的PIPE_BUF的值比较。

12.9 运行程序12-14拷贝一个文件，检查输入文件的上一次访问时间是否改变了？

12.10 在程序12-14中mmap后调用close关闭输入文件，以验证关闭描述符不会使内存映射I/O失效。