# 第14章 高级I/O

## 14.1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函数以及存储映射I/O （mmap），这些都称为高级I/O。后面的内容比如进程间通信，以及以后各章中的很多实例都要使用本章所描述的概念和函数。

## 14.2非阻塞I/O

前面讲系统调用分成“低速”系统调用和其他系统调用两类。低速系统调用是可能会使进程永远阻塞的一类系统调用，它们包括下列调用：

* 如果某些文件类型（例如管道、终端设备和网络设备）的数据并不存在，则读操作可能会使调用者永远阻塞。
* 如果数据不能立即被上述同样类型的文件接受（由于在管道中无空间、网络流控制等），则写操作也会使调用者永远阻塞。
* 在某种条件发生之前，打开某些类型的文件会被阻塞，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：又例如在没有其他进程已用读模式打开该FIFO时若以只写模式打开FIFO，那么也要等待。
* 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。

前面也曾说过，虽然读、写磁盘文件会使调用者在短暂时间内阻塞，但并不能将与磁盘I/O有关的系统调用视为“低速”。

非阻塞I/O使我们可以调用open、read和write这样的I/O操作，并使这些操作不会永远阻塞。如果这种操作不能完成，则调用立即出错返回，表示该操作如继续执行将阻塞。

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)如果调用open获得描述符，则可指定O\_NONBLOCK标志。

(2)对于已经打开的一个描述符，则可用fcntl，由该函数打开O\_NONBLOCK文件状态标志。

下面例子是一个非阻塞I/O的实例，它从标准输入读500 000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write调用的结果都在标准出错上打印。函数clr\_fl类似于前面程序中的set\_fl，但与set\_fl的功能相反，它清除1个或多个标志位。

|  |
| --- |
|  |

程序运行结果如下，若标准输出是普通文件，则可以期望write只执行一次：

|  |
| --- |
|  |

但是，若标准输出是终端，则期望write有时会返回小于500000的一个数字，有时则出错返回。

下面是在一个系统上运行上述程序的结果：

|  |
| --- |
|  |

在该系统上，errno值11对应的是EAGAIN。终端驱动程序一次接收的数据量随系统而变。根据你登录系统时所使用的不同方式——是在系统控制台上登录，还是在硬接线的终端上登录，或是用伪终端在网络连接上登录——该程序运行的结果也不同。如果你在终端上在运行一窗口系统，那么也是经由伪终端设备与系统交互。

在此实例中，程序发出了数千个write调用，但是只有10-20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。后面将介绍非阻塞描述符的I/O多路转接，这是进行这种操作的一种比较有效的方法。

有时，我们可以将应用程序设计成使用多线程（见第11章），从而避免使用非阻塞I/O。如若我们能在其他线程中继续进展，则可以允许某个线程在I/O调用中阻塞。这种方法有时能简化应用程序的设计，但是线程间同步的开销有时却可能增加复杂性，于是导致得不偿失的后果。

## 14.3记录锁

若两个人同时编辑一个文件，其后果将如何呢？在很多LINUX/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），进程有时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，商用LINUX/UNIX系统提供了记录锁机制。

记录锁（record locking）的功能是：当一个进程正在读或修改文件的某个部分时，它可以阻止其他进程修改同一文件区。对于LINUX/UNIX系统而言，“记录”这个词是一种误用，因为LINUX/UNIX 系统内核根本没有使用文件记录这种概念。更适合的术语可能是字节范围锁（byte-range locking），因为它锁定的只是文件中的一个区域（也可能是整个文件）。

### 1. fcntl记录锁

前面已经给出了fcntl函数的原型，为了叙述方便，这里再重复一次。

|  |
| --- |
| #include <fcntl.h>  int fcntl (int filedes, int cmd, … /\* struct flock \*flockptr \*/ );  返回值：若成功则依赖于cmd (见下)，若出错则返回-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 1\_start; /\* offset in bytes, relative to 1\_whence \*/

short 1\_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 \*/

}

对flock结构说明如下：

* 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）。
* 要加锁或解锁区域的起始字节偏移量，这由l\_start和l\_whence两者决定。
* 区域的字节长度，由l\_len表示。
* 具有能阻塞当前进程的锁，其持有进程的ID存放在l\_pid中（仅由F\_GETLK返回）。

关于加锁和解锁区域的说明还要注意下列各点：

* l\_stat是相对偏移量（字节），l\_whence则决定了相对偏移量的起点。这与lseek函数（见3.6节）中最后两个参数类似。确实，l\_whence可选用的值是SEEK\_SET、SEEK\_CUR 或 SEEK\_END。
* 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。
* 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）开始直至最大可能偏移量为止，也就是不管添写到该文件中多少数据，它们都处于锁的范围内（不必猜测会有多少字节被追加到文件之后）。
* 为了锁整个文件，我们设置l\_start和l\_whence，使锁的起点在文件起始处，并且说明长度（l\_len）为0。有多种方法可以指定文件起始处，但常用的方法是将1 \_start指定为0，l\_whence指定为SEEK\_SET。

上面提到了两种类型的锁：共享读锁（l\_type为F\_RDLCK）和独占写锁（F\_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只能有一个进程独用的一把写锁。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。下表列出了这些规则。

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | 请求 | |
| 当前区域状态 |  | 读锁 | 写锁 |
| 无锁 | 允许 | 允许 |
| 一个或多个锁 | 允许 | 拒绝 |
| 一个写锁 | 拒绝 | 拒绝 |
| 不同类型锁之间的兼容性 | | | |

上面说明的兼容性规则适用于不同进程提出的锁请求，并不适用于单个进程提出的多个锁请求。如果一个进程对一个文件区间已经有了一把锁，后来该进程又企图在同一文件区间再加一把锁，那么新锁将替换老锁。例如，若一进程在某文件的16~32字节区间有一把写锁，然后又试图在16~32字节区间加一把读锁，那么该请求将成功执行（假定其他进程此时并不试图向该文件的同一区间加锁），原来的写锁被替换为读锁。

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 | 判断由flockptr所描述的锁是否会被另外一把锁所排斥（p且塞）。如果存在一把锁，它阻止创建由flockptr所描述的锁，则把该现存锁的信息写到 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。如果试图建立一把读锁（1\_type设为 F\_RDLCK）或写锁（l\_type设为F\_WRLCK），而按上述兼容性规则不能允 许，则fcntl立即出错返回，此时errno设置为EACCES或EAGAIN。  此命令也用来清除由flockptr说明的锁（l\_typeSF\_UNLCK）。 |
| F\_SETLKW | 这是F\_SETLK的阻塞版本（命令名中的W表示等待（wait））。如果因为当前在所请求区间的某个部分另一个进程已经有一把锁，因而按兼容性规则由flockptr所请求的锁不能被创建，则使调用进程休眠。如果请求创建的锁已经可用，或者休眠由信号中断，则该进程被唤醒。 |

应当了解，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。因此不能保证在这两次fcntl调用之间不会有另一个进程插入并建立一把相关的锁，从而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锁时，系统按要求组合或裂开相邻区。例如，若字节100~199是加锁的区，需解锁第150字节，则内核将维持两把锁，一把用于字节100~149,另一把用于字节151~199。下图说明了这种情况。

|  |
| --- |
|  |
| 字节150解锁后的文件 |

假定我们又对第150字节设置锁，那么系统将会把三个相邻的加锁区合并成一个（(从字节100至199)。其结果上图中的第一图所示，于是我们又回到了出发点。

下面的程序演示了请求和释放一把锁。为了避免每次分配flock结构，然后又填入各项信息，可以用下面程序中的函数 lock\_reg来处理所有这些细节。

|  |
| --- |
|  |

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5个宏：

|  |
| --- |
|  |

我们有目的地用与lseek函数同样的顺序定义这些宏中的前三个参数。

如果存在一把锁，它阻塞由参数说明的锁请求，则此函数返回持有这把现存锁的进程ID，否则此函数返回0。通常用下面两个宏来调用此函数。

|  |
| --- |
|  |

注意，进程不能使用lock\_test函数测试它自己是否在文件的某一部分持有一把锁。 F\_GETLK命令的定义说明，返回信息指示是否有现存的锁阻止调用进程设置它自己的锁。因为F\_SETLK和F\_SETLKW命令总是替换调用进程现存的锁（若已存在），所以调用进程决不会阻塞在自己持有的锁上，于是，F\_GETLK命令决不会报告调用进程自己持有的锁。

如果两个进程相互等待对方持有并且锁定的资源时，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，则它就会休眠，在这种情况下，有发生死锁的可能性。

下面程序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，它们又都试图锁对方已经加锁的字节。在该程序中使用了前面章节中介绍的父、子进程同步例程 （TELL\_xxx和WAIT\_xxx），使得每个进程能够等待另一个进程获得它设置的第一把锁。

|  |
| --- |
|  |

运行程序，可得如下结果。

|  |
| --- |
|  |

检测到死锁时，内核必须选择一个进程接收出错返回。在本例中选择了子进程，这是一个实现细节。在某些系统上，总是子进程接到出错信息,在另一些系统上，总是父进程接到出错信息。在某些系统上，当试图使用多把锁时，有时是子进程接到出错信息，有时则是父进程接到出错信息。

### 2. 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) 锁与进程和文件两方面有关。这有两重含义：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 一描述符可以引用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：

fdl = open(pathname, …);

read\_lock(fdl, …);

fd2 = dup(fdl);

close(fd2);

则在close (fd2)后，在fdl上设置的锁被释放。如果将dup换为open，以打开另一描述符上的同一文件，其效果也一样：

fdl = open(pathname, ...);

read\_lock(fdl,...);

fd2 = open(pathname, ...);

close(fd2);

(2) 由fork产生的子进程不继承父进程所设置的锁。这意昧着，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，子进程需要调用fcntl才能获得它自己的锁。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。但是注意，如果对一个文件描述符设置了close-on-exec标志，那么当作为exec的一部分关闭该文件描述符时，对相应文件的所有锁都被释放了。

### 3. FreeBSD的实现

先简要地观察FreeBSD实现中使用的数据结构。这会帮助我们进一步理解规则1：锁是与进程、文件两者相关联的。

考虑一个进程，它执行下列语句（忽略出错返回）：

fdl = open(pathname,...);

write\_lock(fdl, 0, SEEK\_SET, 1); /\* parent write locks byte 0 \*/

if ((pid = fork()) > 0)

{

/\* parent \*/

fd2 = dup(fdl);

fd3 = open(pathname, ...);

} else if (pid == 0)

{

read\_lock(fdl, 1, SEEK SET, 1)；

/\* child read locks byte 1 \*/

}

pause ()

下图显示了父、子进程暂停后的数据结构情况。

|  |
| --- |
|  |
| 关于记录锁的FreeBSD数据结构 |

前面章节已显示了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lockf结构，它们由i节点结构开始相互链接起来。注意，每个lockf结构说明了一个给定进程的一个加锁区域（由偏移量和长度定义）。图中显示了两个lockf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fdl、fd2和fd3中的任意一个都将释放由父进程设置的写锁。在关闭这三个描述符中的任意一个时，内核会从该描述符所关联的i节点开始，逐个检査lockf链接表中各项，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。

在前面的程序中，我们了解到，守护进程可用一把文件锁以保证只有该守护进程的唯一副本正在运行。下面给出了lockfile函数的实现，守护进程可用该函数在文件上加锁。

|  |
| --- |
|  |

### 4. 在文件尾埔加锁

在接近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l\_start以及文件当前位置或当前长度得到绝对文件偏移量。但是，常常需要相对于文件的当前位置或当前长度指定一把锁。其原因是，我们在该文件上没有锁，所以不能调用lseek以正确无误地获得加锁时的当前文件偏移量。（在lseek和加锁调用之间，另一个进程可能改变该文件长度。）

考虑以下代码序列：

writew\_lock(fd, 0, SEEK\_END, 0);

write(fd, buf, 1);

un\_lock(fd, 0, SEEK\_END);

write(fd, buf, 1);

该代码序列所做的可能并不是你所期望的。它得到一把写锁，该写锁从当前文件尾端起，包括以后可能添加到该文件的任何数据。假定在文件尾端时执行第一个write，它给文件添写了1个字节，而该字节将被加锁。跟随其后的解锁，其作用是对以后添写到文件上的数据不再加锁，但在它之前刚添写的一个字节则保留加锁。当执行第二个写时，文件尾端又延伸了 1个字节，但该字节并未加锁。由此代码序列造成的文件锁状态示于下图。

|  |
| --- |
|  |
| 文件范围锁 |

当对文件的一部分加锁时，内核将指定的偏移量变换成绝对文件偏移量。另外，除了指定一个绝对偏移量（SEEK\_SET）之外，fcntl还允许我们相对于文件中的某个点（当前偏移量（SEEK\_CUR）或文件尾端（SEEK\_END））指定该偏移量。当前偏移量和文件尾端是可能不断变化的，而这种变化又不应影响现存锁的状态，所以内核必须独立于当前文件偏移量或文件尾端而记住锁。

如果我们想解除第一次write所写1个字节上的锁，那么应指定长度为-1。负的长度值表示在指定偏移量之前的字节数。

### 5. 建议性锁和强制性锁

考虑数据库访问例程库。如果该库中所有函数都以一致的方法处理记录锁，则称使用这些函数访问数据库的任何进程集为合作进程（cooperating process）。如果这些函数是仅有地用来访问数据库的函数，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权限的任何其他进程对数据库文件进行随意的写操作。没有使用被认可的方法（数据库函数库）访问数据库的进程是一个非合作进程。

强制性锁使内核对每一个open、read和write系统调用都进行检查，检查调用进程对正在访问的文件是否违背了某一把锁的作用。强制性锁有时也被称为强迫方式锁（enforcement- mode locking）。

对一个特定文件打开其设置组ID位并关闭其组执行位，则对该文件开启了强制性锁机制。因为当组执行位关闭时，设置组ID位不再有意义，所以可以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或写锁，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。下表列出了8种可能性。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 其他进程在文件区段中持有的现存锁的类型 | 阻塞描述符，试图 | | 非阻塞描述符，试图 | |
| read | write | read | write |
| 读锁 | 允许 | 阻塞 | 允许 | EAGAIN |
| 写锁 | 阻塞 | 阻塞 | EAGAIN | EAGAIN |

除了上表中的read和write函数，其他进程持有的强制性锁也会对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。后随的read或write依从于上表中所示的规则。但是，如果欲打开的文件具有强制性记录锁（读锁或写锁），而且open调用中的flag指定为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。

这种open的锁冲突处理方式可能导致令人惊异的结果。我们曾编写过一个测试程序，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，然后进入休眠一段时间。在这段休眠时间内，用某些典型的LINUX/UNIX系统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。用某些UNIX系统版本提供的系统调用跟踪特性，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为原文件名。强制性锁机制对unlink函数没有影响，于是这一切就发生了。
* 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的这种行为与所预料的一样。
* 使用bash的>和>>运算符重写或添写到该文件中，产生出错信息“cannot create”。
* 在bash下使用>运算符出错，但是使用>>运算符则阻塞，在解除了强制性锁后再继续进行处理。（这两种shell在执行添加操作时会产生这样的区别，是因为bash以 O\_CREAT和O\_APPEND标志打开文件，上面已提及指定O\_CREAT会产生出错返回，而bash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。）

产生的结果随所用操作系统版本的不同而变。从这样一个例子中可见，在使用强制性锁时还需有所警惕。从ed实例可以看到，强制性锁是可以设法避开的。

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权限位）。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。

下列程序用于确定一个系统是否支持强制性锁机制。

|  |
| --- |
|  |

此程序首先创建一个文件，并使强制性锁机制对其起作用。然后程序分裂为父进程和子进程。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(read)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LINUX运行此程序（该系统支持强制性锁机制），得到：

|  |
| --- |
|  |

让我们回到本节的第一个问题：若两个人同时编辑同一个文件将会怎样呢？一般的LINUX/UNIX系统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。

某些版本的vi编辑器使用建议性记录锁。即使我们正在使用这种版本的vi编辑器，但是它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，那么我们可以修改自己常用的编辑器（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，然后执行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了。

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通常在读完输入文件后关闭它。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，编辑器读了该文件的内容后，随即关闭了该文件，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。

## 14.4 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。但是如果必须从两个描述符读，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。

让我们观察telnet(l)命令的结构。该程序读终端（标准输入），将所得数据写到网络连接上，同时读网络连接，将所得数据写到终端上（标准输出）。在网络连接的另一端，telnetd守护进程读用户在终端上所键入的内容，并将其送给shell，这如同用户登录在远程机器上一样。telnetd守护进程将执行用户键入命令，而产生的输出通过telnet命令送回给用户，并显示在用户终端上。下图显示这种工作情景。

|  |
| --- |
|  |
| telnet程序概观 |

telnet进程有两个输入、两个输出。对这两个输入中的任一个都不能使用阻塞read，因为我们永远不知道哪一个输入有我们需要的数据。

处理这种特殊问题的一种方法是，用fork将一个进程变成两个进程，每个进程处理一条数据通路。下图显示了这种安排。

|  |
| --- |
|  |
| 使用两个进程实现telnet程序 |

如果使用两个进程，则可使每个进程都执行阻塞read。但是这也产生了问题：操作什么时候终止？如果子进程接收到文件结束符telnetd守护进程使网络连接断开，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。为此可以使用一个信号（例如SIGUSR1），但这使程序变得更加复杂。

我们可以不使用两个进程，而是用一个进程中的两个线程。这避免了终止的复杂性，但却要求处理线程之间的同步，在减少复杂性方面这可能会是得不偿失。

另一个方法是仍旧使用一个进程执行该程序，但使用非阻塞I/O读取数据。基本方法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同样的处理。在此之后，等待若干秒，然后再读第一个描述符。这种形式的循环称为轮询(polling)。这种方法的不足之处是浪费CPU时间。因为大多数时间实际上是无数据可读的，但是仍花费时间不断反复执行read系统调用。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。虽然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这种方法。

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，并非所有系统都支持这种机制。其次，这种信号对每个进程而言只有1个（SIGPOLL或SIGIO）。如果使该信号对两个描述符都起作用（在我们正在讨论的实例中，从两个描述符读），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，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。

一种比较好的技术是使用I/O多路转接（I/O multiplexing）。先构造一张有关描述符的列表，然后调用一个函数，直到这些描述符中的一个已准备好进行I/O时，该函数才返回。在返回时，它告诉进程哪些描述符已准备好可以进行I/O。

poll、pselect和select这三个函数使我们能够执行I/O多路转接。

### 14.4.1 select和pselect 函数

在所有LINUX/UNIX平台上，select函数使我们可以执行I/O多路转接。传向select的参数告诉内核：

* 我们所关心的描述符。
* 对于每个描述符我们所关心的状态。（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状态？）
* 愿意等待多长时间（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：

* 已准备好的描述符的数量。
* 对于读、写或异常这三个状态中的每一个，哪些描述符已准备好。

使用这些返回信息，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/select.h>  int select (int maxfdpl, f d set \*restrict readfds,  fd\_set \* restrict writefds, fd\_set \* restrict exceptfds, struct time val \* re strict tvptr);  返回值：准备就绪的描述符数，若超时则返回0，若出错则返回-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：

(1) 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。 (2) 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

中间三个参数readfds，writefds，expectfds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。这种数据类型为每一可能的描述符保持了一位，其实现可如下图所示。

|  |
| --- |
|  |
| 对select指定读、写和异常条件描述符 |

对fd\_set数据类型可以进行的处理是：分配一个这种类型的变量；将这种类型的一个变量值赋予同类型的另一个变量，或对于这种类型的变量使用下列四个函数中的一个。

|  |
| --- |
| #include <sys/select.h>  int FD\_ISSET(int fd, fd set \*fdset);  返回值：若fd在描述符集中则返回非0值，否则返回0  void FD\_CLR (int fd, fd\_set \*fdset);  void FD\_SET(int fd, fd\_set \*fdset);  void FD\_ZERO (f d set \*fdset); |

这些接口可实现为宏或函数。调用FD\_ZERO将一个指定的fd\_set变量的所有位设置为0。调用FD\_SET设置一个fd\_set变量的指定位。调用FD\_CLR则将一指定位清除。最后，调用FD\_ISSET测试一指定位是否设置。

声明了一个描述符集后，必须用FD\_ZERO清除其所有位，然后在其中设置我们关心的各个位。这种操作序列如下所示：

fd\_set rset;

int fd;

FD\_ZERO(&rset)；

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的中间三个参数（指向描述符集的指针）中的任意一个或全部都可以是空指针，这表示对相应状态并不关心。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。前面讨论过，sleep等待整数秒，而对于select，其等待的时间可以小于1s, 其实际分辨率取决于系统时钟。

select的第一个参数maxfdp1的意思是“最大描述符加1”。在三个描述符集中找出最大描述符编号值，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 <sys/select.h>中的一个常量，它说明了最大的描述符数（经常是1024）。但是对大多数应用程序而言，此值太大了，多数应用程序只使用3~10个描述符。（某些应用程序使用更多的描述符，但这种UNIX程序并不具代表性。）如果将第三个参数设置为我们所关注的最大描述符编号值加1，内核就只需在此范围内寻找打开的位，而不必在三个描述符集中的数百位内捜索。 例如，若编写下列代码：

fd\_set readsec, 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)；

下图显示了这两个描述符集的情况。

|  |
| --- |
|  |
| select的示例描述符集 |

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。出错是有可能的，例如在所指定的描述符都没有准备好时捕捉到一个信号。在此种情况下，将不修改其中任何描述符集。

(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。此时，所有描述符集皆被清0。

(3) 正返回值表示已经准备好的描述符数，该值是三个描述符集中已准备好的描述符数之和，所以如果同一描述符已准备好读和写，那么在返回值中将其计为2。在这种情况下，三个描述符集中仍旧打开的位对应于已准备好的描述符。

对于“准备好”的意思要作一些更具体的说明：

* 若对读集（readfds）中的一个描述符的read操作将不会阻塞，则此描述符是准备好的。
* 若对写集（writefds）中的一个描述符的write操作将不会阻塞，则此描述符是准备好的。
* 若异常状态集（exceptfds）中的一个描述符有一个未决异常状态，则此描述符是准备好的。现在，异常状态包括(a)在网络连接上到达的带外数据，或者(b)在处于数据包模式的伪终端上发生了某些状态。
* 对于读、写和异常状态，普通文件描述符总是返回准备好。

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5s调用select，则select最多阻塞5s。相类似地，如果指定一个无限的超时值，则在该描述符数据准备好或捕捉到一个信号之前， select一直阻塞。

如果在一个描述符上碰到了文件结尾处，则select认为该描述符是可读的。然后调用read，它返回0，这是LINUX/UNIX系统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会指示一个异常状态。）

LINUX/UNIX也定义了一个select的变体，它被称为pselect。

|  |
| --- |
| #include <sys/select>  int pselect (int maxfdpl, fd\_set \*restrict readfds,  fd\_set \* restrict writ^ds, fd\_set \* re strict exceptfds, const struct timespec \* restrict tsptr, const sigset\_t \* restrict sigmask);  返回值：准备就绪的描述符数，若超时则返回0，若出错则返回-1 |

除下列几点外，pselect与select相同：

* select的超时值用timeval结构指定，但pselect使用timespec结构。timespec结构以秒和纳秒表示超时值，而非秒和微秒。如果平台支持这样精细的粒度，那么timespec就提供了更精准的超时时间。
* pselect的超时值被声明为const，这保证了调用pselect不会改变此值。
* 对于pselect可使用一可选择的信号屏蔽字。若sigmask为空，那么在与信号有关的方面, pselect的运行状况和select相同。否则，sigmask指向一信号屏蔽字，在调用pselect时，以原子操作的方式安装该信号屏蔽字。在返回时恢复以前的信号屏蔽字。

### 14.4.2 poll函数

poll函数类似于select，但是其接口则有所不同，poll函数可用于任何类型的文件描述符。

|  |
| --- |
| #include <poll.h>  int poll (struct pollfd fdarray[] , nfds\_t nfds, int timeout)；  返回值：准备就绪的描述符数，若超时则返回0，若出错则返回-1 |

与select不同，poll不是为每个状态（可读性、可写性和异常状态）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状态。

struct pollfd {

int fd;

short events;

short revents;

}；

fdarray数组中的元素数由nfds说明。

应将每个数组元素的events成员设置为下表中所示的值。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，以说明对于该描述符已经发生了什么事件。（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。）

|  |  |  |  |
| --- | --- | --- | --- |
| poll 的events和revents标志 | | | |
| 标志名 | 输人至events? | 从revents 得到结果？ | 说明 |
| POLLIN | • | • | 不阻塞地可读除高优先级外的数据(等效于POLLRDNORM｜POLLRDBAND) |
| POLLRDNORM | • | • | 不阻塞地可读普通数据（优先级波段为0） |
| POLLRDBAND | • | • | 不阻塞地可读非0优先级波段数据 |
| POLLPRI | • | • | 不阻塞地可读高优先级数据 |
| POLLOUT | • | • | 不阻塞地吋写普通数据 |
| POLLWRNORM | • | • | 与POLLOUT相同 |
| POLLWRBAND | • | • | 不阻塞地可写非0优先级波段数据 |
| POLLERR |  | • | 已出错 |
| POLLHUP |  | • | 已挂断 |
| POLLNVAL |  | • | 描述符不引用一打开文件 |

上表中前四行测试可读性，接着三行测试可写性，最后三行则是测试异常状态。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(POLLHUP)后，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愿意等待多少时间。如同select—样，有三种不同的情形：timeout == -1

永远等待。（某些系统在<stropts.h>中定义了常量INFTIM，其值通常 是-1。）当所指定的描述符中的一个已准备好，或捕捉到一个信号时则返回。如果捕捉到一个信号，则poll返回-1, errno设为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。如果正从终端输入数据，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。如果正在读调制解调器，并且电话线已挂断，则在revents中将接到POLLHUP通知。

与select—样，不论一个描述符是否阻塞，都不影响poll是否阻塞。

LINUX/UNIX在接到一信号时都不重启动poll和select，即便使用了 SA\_RESTART标志也是如此。

## 14.5 readv和writev函数

readv和writev函数用于在一次函数调用中读、写多个非连续缓冲区。有时也将这两个 函数称为散布读（scatterread）和聚集写（gatherwrite）。

|  |
| --- |
| #include <sys/uio.h>  ssize\_t readv (int filedes• const struct iovec \*iov, int iavcnt);  ssize\_t writev (int filedes, const struct iovec \*iovt int iovcnt);  两个函数返回值：若成功则返回已读、写的字节数，若出错则返回-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。其最大值受限于IOV\_MAX。下图显示了 readv和writev的参数和i ovec结构。

|  |
| --- |
|  |
| readv和writev的 iovec结构 |

writev以顺序iov[0]，iov[1]至iov[iovcnt-1]从缓冲区中聚集输出数据。writev返回输出的字节总数，通常，它应等干所有缓冲区长度之和。

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

如果需将两个缓冲区内容连续地写到一个文件中。第二个缓冲区是调用者传递过来的一个参数，第一个缓冲区是我们创建的，它包含了第二个缓冲区的长度以及在文件中其他信息的偏移量。有三种方法可以实现这一要求：

(1) 调用write两次，一次-个缓冲区。

(2) 分配一个大到足以包含两个缓冲区的新缓冲区。将两个缓冲区的内容复制到新缓冲区中。然后对该缓冲区调用write一次。

(3) 调用writev输出两个缓冲区。

将writev与另外两种方法进行比较，对我们是很有启发的。

一般来说调用write两次的系统时间较调用write或writev一次要长。在缓冲区复制后跟随一个write所用的CPU时间（用户加系统）要少于调用writev一次所耗费的CPU时间。对于单write情况，我们先将用户层次的两个缓冲区复制至一个中间缓冲区，然后当调用write时内核将该中间缓冲区中的数据复制至其内部缓冲区。对于writev的情况，因为内核只需将数据直接复制进其内部缓冲区，所以复制工作应当少一些。但是，对于这种少量数据，使用writev的固定开销大干得益。随着需复制数据的增加，程序中复制缓冲区的开销也会增多。此时，writev这种替代方法就会有更大的吸引力。

总之，应当用尽量少的系统调用次数来完成任务。如果只写少量的数据，会发现自己复制数据然后使用一次write会比用writev更合算。但也可能发现，这样获得的性能提升并不值得，因为管理中间缓冲区会增加程序的复杂度。

## 14.6 readn和writen函数

管道、FIFO以及某些设备，特别是终端、网络有下列两种性质：

(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也可能是这样。这不是错误，应当继续读该设备。

(2) —次write操作的返回值也可能少干指定输出的字节数。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的中途返回。）

在读、写磁盘文件时从未见到过这种情况，除非文件系统用完了空间，或者我们接近了配额限制，而不能将要求写的数据全部写出。

通常当读、写一个管道、网络设备或终端时，我们需要考虑这些特性。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 "laworks.h"  ssize\_t readn(int filedes, void \*buf, size—t nbytes);  ssize\_t writen (int filedes, void \*buf, Bize\_t nbytes);  两个函数返回值：已读、写字节数，若出错则返回-1 |

readn和wri ten函数并非任何标准的组成部分。在要将数据写到上面提到的文件类型上时，就可调用writen，但是只有当事先就知道要接收数据的数量时，才调用readn（通常只调用read接收来自这些设备的数据）。下面包含了一种readn和writen的实现。

|  |
| --- |
|  |

注意，若在已经读、写了一些数据后出错，则这两个函数返回已传输的数据量，而非出错返回。与此类似，在读时如达到文件尾，而且在此之前已成功地读了一些数据，但尚未满足所要求的量，则readn返回已复制到调用者缓冲区中的字节数。

## 14.7存储映射I/O

存储映射I/O（Memory-mapped I/O）使一个磁盘文件与存储空间中的一个缓冲区相映射。于是当从缓冲区中取数据，就相当于读文件中的相应字节。与此类似，将数据存入缓冲区，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/mman.h>  void \*mmap(void \*addrg size\_t lent int prot, int flag, int filedes, off\_t off);  返回值：若成功则返回映射区的起始地址，若出错则返回MAP\_FAILED |

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偏移量（下面将说明对off值有某些限制）。

prot参数说明对映射存储区的保护要求，见下表。

|  |  |
| --- | --- |
| 映射存储区的保护要求 | |
| prot | 说明 |
| PROT\_READ | 映射区可读 |
| PROT\_WRITE | 映射区可写 |
| PROT\_EXEC | 映射区可执行 |
| PROT\_NONE | 映射区不可访问 |

可将prof参数指定为PROT\_NONE，或者是PROT\_READ、PROT\_WRITE、PR0T\_EXEC任意组合的按位或。对指定映射存储区的保护要求不能超过文件open模式访问权限。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。

在说明flag参数之前，先看一下存储映射文件的基本情况。下图显示了一个存储映射文件。

|  |
| --- |
|  |
| 存储映射文件的例子 |

在上图中，“起始地址”是mmap的返回值。映射存储区位于堆和找之间，这属于实现细节，各种实现之间可能不尽相同。

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。

如果来指定此标志，而且addr非0，则内核只把addr视为在何处设置映射区的一种建议，但是不保证会使用所要求的地址。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，也就是说，存储操作相当于对该文件的write。必须指定本标志或下一个标志（MAP \_PRIVATE），但不能同时指定两者。

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私有副本。

所有后来对该映射区的引用都是引用该副本，而不是原始文件。（此标志的一种用途是用于调试程序，它将一程序文件的正文部分映射至一存储区，但允许用户修改其中的指令。任何修改只影响程序文件的副本，而不影响原文件。）

每种实现都可能还有另外一些MAP\_xxx标志值，它们是这种实现所特有的。详细情况请参见系统的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。因为off和addr常常指定为0，所以这种要求一般并不重要。

因为映射文件的起始偏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字节被设置为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。于是，我们不能用mmap将数据添加到文件中。为了做到这一点，我们必须首先加长该文件。

与映射存储区相关的有SIGSEGV和SIGBUS两个信号。信号SIGSEGV通常j于指示进程试图访问对它不可用的存储区。如果进程企图存数据到mmap指定为只读的映射存储区，那么也产生此信号。如果访问映射区的某个部分，而在访问时这一部分实际上已不存在，则产生SIGBUS信号。例如，用文件长度映射了一个文件，但在引用该映射区之前，另一个进程已将该文件截短，此时，如果进程企图访问对应于该文件已截去部分的映射区，则会接收到 SIGBUS 信号。

在调用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，调用exec后的新程序则不继承此存储映射区。

调用mprotect可以更改一个现存映射存储区的权限。

|  |
| --- |
| #include <sys/mman.h>  int mprotect (void \*addr, size\_t len, int prot);  返回值：若成功则返回0，若出错则返回-1 |

prot的许可值与mmap中prot参数一样。地址参数的值必须是系统页长的整数倍。

如果在共享存储映射区中的页已被修改，那么我们可以调用msync将该页冲洗到被映射的文件中。msync函数类似于fsync，但作用于存储映射区。

|  |
| --- |
| #include <sys/mman.h>  int msync (void \*addr• size\_t len, int flags) ;  返回值：若成功则返回0，若出错则返冋-1 |

如果映射是私有的，那么不修改被映射的文件。与其他存储映射函数一样，地址必须与页边界对齐。

flags供参数使我们对如何冲洗存储区有某种程度的控制。我们可以指定MS\_ASYNC标志以简化被写页的调度。如果我们希望在返回之前奪待写操作完成，则可指定MS\_SYNC标志。一定要指定MS\_ASYNC和MS\_SYNC中的一个。

MS\_INVALIDATE是一个可选标志，使用它们以通知操作系统丢弃与底层存储器没有同步的任何页。若使用了此标志，某些实现将丢弃在指定范围中的所有页，但这并不是所期望的。

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

|  |
| --- |
| #include <sys/mman.h>  int munmap (caddr\_t addr, size\_t len);  返回值：若成功则返回0，若出错则返回-1 |

munmap不会影响被映射的对象，也就是说，调用munmap不会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。在解除了映射后，对于MAP\_PRIVATE存储区的修改被丢弃。

下面的代码用存储映射I/O复制一个文件。

|  |
| --- |
|  |

该程序首先打开两个文件，然后调用fstat得到输入文件的长度。在为输入文件调用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引用会产生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。

然后对每个文件调用mmap，将文件映射到存储区，最后调用memcpy将输入缓冲区的内容复制到输出缓冲区。在从输入缓冲区(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冲区(dst)时，内核自动将数据写到输出文件中。

如果考虑到时钟时间，那么mmap和memcpy方式较read和write方式要快。这是合情合 理的。使用mmap和memcpy时做的工作要少。用read和write时，要先将数据从内核缓冲区复制到应用程序缓冲区（read），然后又将应用程序缓冲区中的数据复制至内核缓冲区(write)。用mmap和memcpy时，则直接将映射到应用程序地址空间的一个内核缓冲区中的数据复制到另一个同样映射到应用程序地址空间中的内核缓冲区中。

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，例如，不能用其在某些设备（例如网络设备或终端设备）之间进行复制，并且在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，某些应用程序会从存储映射I/O中得到好处，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对帧缓冲区设备的操作，该设备引用一个位图式显示(bit-mapped display)。

## 14.8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。
* I/O多路转接——select和poll函数（后面的很多实例将用到这两个函数）。
* readv和writev函数（后面的很多实例也将用到这两个函数）。
* 存储映射I/O (mmap)。