第3章 文 件 I/O

3.1 引言

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

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

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

3.2 文件描述符

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

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

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

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

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

3.3 open函数

调用open函数可以打开或创建一个文件。

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

int open(const char *pathname, int oflag, .../* mode_t mode */);
                                                返回：若成功为文件描述符，若出错为- 1
```

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

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

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

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

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

在这三个常数中应当只指定一个。下列常数则是可选择的：

- O_APPEND　每次写时都加到文件的尾端。3.11节将详细说明此选择项。
- O_CREAT　若此文件不存在则创建它。使用此选择项时，需同时说明第三个参数 *mode*，用其说明该新文件的存取许可权位。 (4.5节将说明文件的许可权位，那时就能了解如何说明*mode*，以及如何用进程的umask值修改它。)
- O_EXCL　如果同时指定了O_CREAT，而文件已经存在，则出错。这可测试一个文件是否存在，如果不存在则创建此文件成为一个原子操作。 3.11节将较详细地说明原子操作。
- O_TRUNC　如果此文件存在，而且为只读或只写成功打开，则将其长度截短为 0。
- O_NOCTTY　如果*pathname*指的是终端设备，则不将此设备分配作为此进程的控制终端。9.6节将说明控制终端。
- O_NONBLOCK　如果*pathname*指的是一个FIFO、一个块特殊文件或一个字符特殊文件，则此选择项为此文件的本次打开操作和后续的 I/O操作设置非阻塞方式。 12.2节将说明此工作方式。

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

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

> O_SYNC选择项不是POSIX.1的组成部分，但SVR4支持此选择项。

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

文件名和路径名截短

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

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

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

> FIPS 151-1要求返回出错。
>
> SVR4对传统的系统V文件系统(S5)并不保证返回出错(见表2-6)，但是对BSD风格的文件系统(UFS)，SVR4保证返回出错，4.3+BSD总是返回出错。

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

3.4 creat函数

也可用creat函数创建一个新文件。

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

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

返回：若成功为只写打开的文件描述符，若出错为-1

注意，此函数等效于：

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

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

在4.5节中，我们将详细说明文件存取许可权，并说明如何指定*mode*。

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

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

3.5 close函数

可用close函数关闭一个打开文件：

```
#include <unistd.h>

int close (int filedes);
```

返回：若成功为0，若出错为-1

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

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

3.6 lseek函数

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

可以调用lseek显式地定位一个打开文件。

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

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

返回：若成功为新的文件位移，若出错为-1

对参数*offset*的解释与参数*whence*的值有关。

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

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

```
off_t    currpos;

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

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

> 三个符号常数SEEK_SET，SEEK_CUR和SEEK_END是由系统V引进的。在系统V之前，whence被指定为0(绝对位移量)，1(相对于当前位置的位移量)或2(相对文件尾端的位移量)。很多软件仍直接使用这些数字进行编码。
>
> 在lseek中的字符l表示长整型。在引入off_t数据类型之前，*offset*参数和返回值是长整型的。lseek是由V7引进的，当时C语言中增加了长整型。(在V6中，用函数seek和tell提供类似功能。)

实例

程序3-1用于测试其标准输入能否被设置位移量。

程序3-1 测试标准输入能否被设置位移量

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

int
main(void)
{
```

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

如果用交互方式调用此程序，则可得：

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

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

> 在80386上运行的SVR4支持/dev/kmem设备，它可以具有负的位移量。
>
> 因为位移量（off_t）是带符号数据类型（见表 2-8），所以文件最大长度减少一半。例如，若off_t 是32位整型，则文件最大长度是2^{31}字节。

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

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

实例

程序3-2用于创建一个具有空洞的文件。

程序3-2 创建一个具有空洞的文件

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

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

int
main(void)
{
    int     fd;

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

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

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

```
    /* offset now = 40 */

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

    exit(0);
}
```

运行该程序得到：

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

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

3.7 read函数

用read函数从打开文件中读数据。

```
#include <unistd.h>

ssize_t read(int filedes, void *buff, size_t nbytes);
                                    返回：读到的字节数，若已到文件尾为0，若出错为-1
```

如read成功，则返回读到的字节数。如已到达文件的尾端，则返回0。

有多种情况可使实际读到的字节数少于要求读字节数：

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

• 当从终端设备读时，通常一次最多读一行(第11章将介绍如何改变这一点)。

• 当从网络读时，网络中的缓冲机构可能造成返回值小于所要求读的字节数。

• 某些面向记录的设备，例如磁带，一次最多返回一个记录。

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

POSIX.1在几个方面对此函数的原型作了更改。其经典定义是：

```
int read(int filedes, char *buff, unsigned nbytes);
```

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

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

3.8 write函数

用write函数向打开文件写数据。

```
#include <unistd.h>

ssize_t write(int filedes, const void buff, size_t nbytes);
                                        返回：若成功为已写的字节数，若出错为- 1
```

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

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

3.9 I/O的效率

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

• 它从标准输入读，写至标准输出，这就假定在执行本程序之前，这些标准输入、输出已由shell安排好。确实，所有常用的UNIX shell都提供一种方法，它在标准输入上打开一个文件用于读，在标准输出上创建(或重写)一个文件。

• 很多应用程序假定标准输入是文件描述符 0，标准输出是文件描述符 1。本例中则用两个在<unistd.h>中定义的名字STDIN_FILENO和STDOUT_FILENO。

• 考虑到进程终止时，UNIX会关闭所有打开文件描述符，所以此程序并不关闭输入和输出文件。

• 本程序对文本文件和二进制代码文件都能工作，因为对 UNIX内核而言，这两种文件并无区别。

程序3-3 将标准输入复制到标准输出

```
#include    "ourhdr.h"

#define BUFFSIZE    8192

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

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

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

    exit(0);
}
```

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

表3-1 用不同缓存长度进行读操作的时间结果

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

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

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

3.10 文件共享

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

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

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

(a) 文件描述符标志。

(b) 指向一个文件表项的指针。

(2) 内核为所有打开文件维持一张文件表。每个文件表项包含：

(a) 文件状态标志(读、写、增写、同步、非阻塞等)。

(b) 当前文件位移量。

(c) 指向该文件v节点表项的指针。

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

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

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

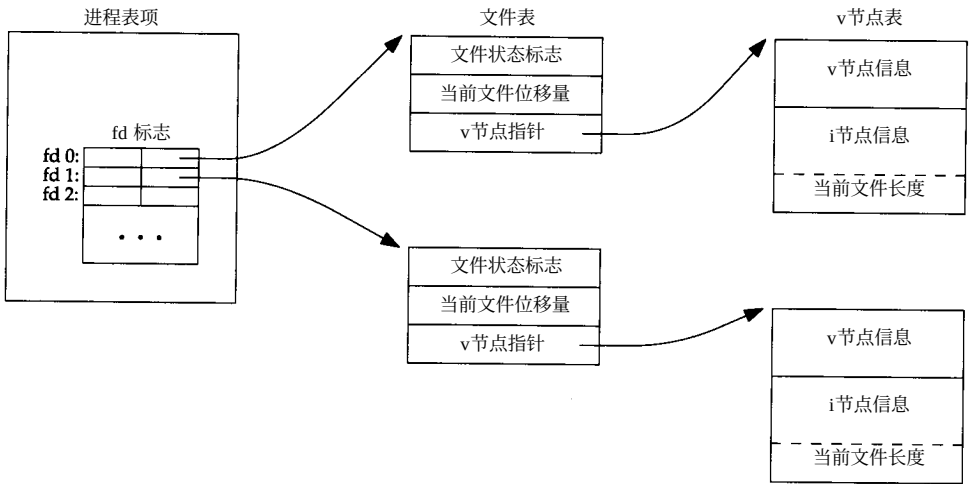

图3-1 打开文件的内核数据结构

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

> v节点结构是近来增设的。当在一个给定的系统上对多种文件系统类型提供支持时，就需要这种结构，这一工作是由Peter Weinberger（贝尔实验室）和Bill Joy（Sun公司）分别独立完成的。 Sun称此种文件系统为虚拟文件系统（ Virtual File System），称与文件系统类型无关的i节点部分为v节点〔Kleiman 1986〕。当各个制造商的实现增加了对 Sun的网络文件系统(NFS)的支持时，它们都广泛采用了 v节点结构。
>
> 在SVR4中，v节点代换了SVR3中的与文件系统类型无关的i节点结构。

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

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

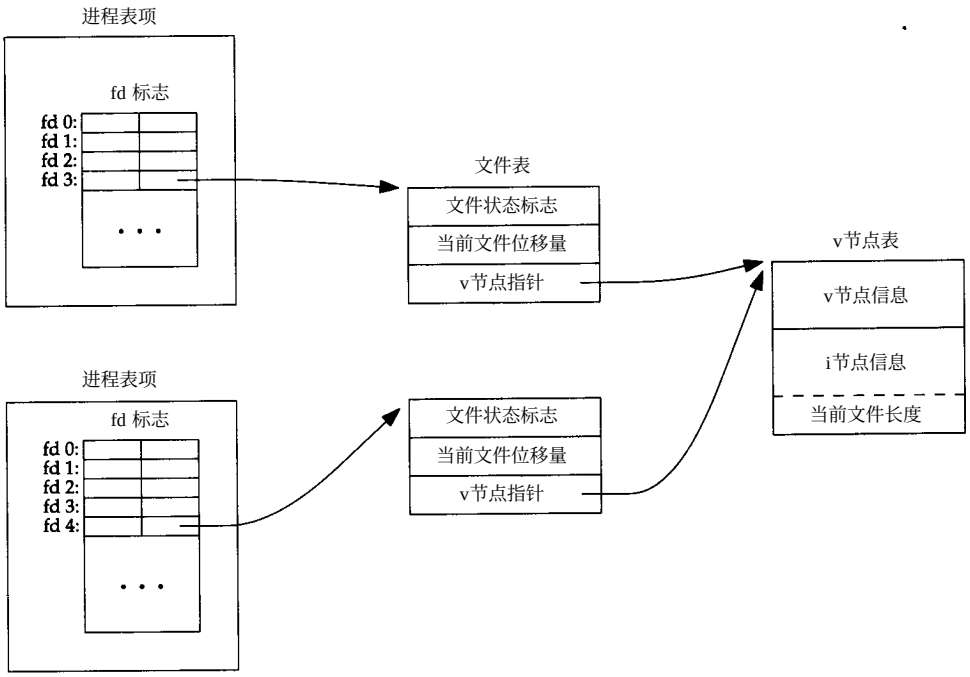

图3-2 两个独立进程各自打开同一个文件

给出了这些数据结构后，现在对前面所述的操作作进一步说明。

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

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

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

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

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

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

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

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

3.11 原子操作

3.11.1 添加至一个文件

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

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

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

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

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

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

3.11.2 创建一个文件

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

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

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

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

3.12 dup和dup2函数

下面两个函数都可用来复制一个现存的文件描述符：

```
#include <unistd.h>

int dup(int filedes);

int dup2(int filedes, int filedes2);
```

两函数的返回：若成功为新的文件描述符，若出错为-1

由dup返回的新文件描述符一定是当前可用文件描述符中的最小数值。用dup2则可以用*filedes2*参数指定新描述符的数值。如果*filedes2*已经打开，则先将其关闭。如若*filedes*等于*filedes2*，则dup2返回*filedes2*，而不关闭它。

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

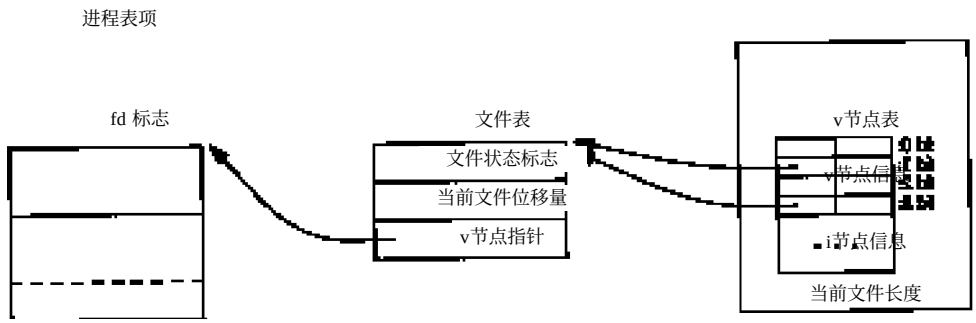

图3-3 dup(1)后内核数据结构

在此图中，我们假定进程执行了：

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

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

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

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

```
dup(filedes);
```

等效于：

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

而调用：

```
dup2(filedes, filedes2)
```

等效于：

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

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

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

(2) 在dup2和fcntl之间有某些不同的errno。

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

3.13 fcntl函数

fcntl函数可以改变已经打开文件的性质。

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

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

返回：若成功则依赖于*cmd*(见下)，若出错为-1

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

fcntl函数有五种功能：

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

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

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

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

• F_SETFD　对于*filedes* 设置文件描述符标志。新标志值按第三个参数(取为整型值)设置。

应当了解很多现存的涉及文件描述符标志的程序并不使用常数 FD_CLOEXEC，而是将此标志设置为0(系统默认，在exec时不关闭)或1(在exec时关闭)。

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

表3-2　对于fcntl的文件状态标志

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

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

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

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

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

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

实例

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

程序3-4　对于指定的描述符打印文件标志

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

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

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

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

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

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

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

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

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

实例

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

程序3-5是一个对于一个文件描述符设置一个或多个文件状态标志的函数。

程序3-5 对一个文件描述符打开一个或多个文件状态标志

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

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

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

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

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

如果将中间的一条语句改为：

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

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

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

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

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

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

表3-3 用同步写(O_SYNC)的时间结果

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

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

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

3.14 ioctl函数

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

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

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

返回：若出错则为-1，若成功则为其他值

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

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

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

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

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

表3-4 4.3+BSD ioctl操作

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

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

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

3.15 /dev/fd

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

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

在函数中调用：

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

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

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

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

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

我们仍然不能对fd进行写操作。

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

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

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

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

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

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

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

3.16 小结

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

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

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

习题

3.1 当读/写磁盘文件时，本章中描述的函数是否有缓存机制？请说明原因。

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

3.3 假设一个进程执行下面的3个函数调用：

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

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

3.4 在许多程序中都包含下面一段代码：

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

```
        close(fd);
```

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

3.5 在Bourne shell和KornShell中，*digit*1>&*digit*2表示要将描述符*digit*1重定向至描述符*digit*2的同一文件。请说明下面两条命令的区别。

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

（提示：shell从左到右处理命令行。）

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