文件 I/O

一、 引言

本章开始讨论 UNIX 系统，先说明可用的文件除度 I/O 函数---打开文件、读文件、写文件等。UNIX 系统中的大多数文件 I/O 只需用到 5 个函数:open, read, write, lseek 以及 close。然后说明不同缓冲长度对 read 和 write 函数的影响。本章描述的函数经常被称为不带缓冲的 I/O(unbuffered I/O，与将在第 5 章中说明的标准 I/O 函数相对照)。术语不带缓冲指的是每个 read 和 write 都调用内核中的一个系统调用。这些不带缓冲的 I/O 函数不是 ISO C 的组成部分，但是，它们是 POSIX.I 和 Single UNIX Specification 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得非常重要。我们将通过文件 I/O 和 open 函数的参数来讨论此概念。然后，本章将进一步讨论在多个进程间如何共享文件，以及所涉及的内核有关数据结构。在描述了这些特征后，将说明 dup, fcntl, sync, fsync 和 ioctl 函数。

二、 文件描述符

对于内核而言，所有打开的文件都通过文件描述符引用。文件描述符是一个非负整数。当打开一个现有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，使用 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 系统 shell 把文件描述符 0 与进程的标准输入关联，文件描述符 1 与标准输出关联，文件描述符 2 与标准错误关联。这是各种 shell 以及很多应用程序使用的惯例，与 UNIX 内核无关。尽管如此，如果不遵循这种惯例，很多 UNIX 系统应用程序就不能正常工作。

在符合 POSIX.1 的应用程序中，幻数 0,1,2 虽然已被标准化，但应当把它们替换成符号常量 STDIN_FILENO,STDOUT_FILENO 和 STDERR_FILENO 以提高可读性。这些常量都在头文件<unistd.h>中定义。文件描述符的变化范围是 0OPEN～MAX-1。早期的 UNIX 系统实现采用的上限值是 19(允许每个进程最多打开 20 个文件)，但现在很多系统将其上限值增加至 63。

注：对于 FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8 以及 Solaris 10，文件描述符的变化范围几乎是无限的，它只受到系统配置的存储器总量、整型的字长以及系统管理员所配置的软限制和硬限制的约束。

三、 函数 open 和 openat

调用 open 或 openat 函数可以打开或创建一个文件。

```
#include <fcntl.h>

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

int openat(int fd, const char *path, int oflag, ... /* mode_t mode
    */ );
                        两函数的返回值：若成功，返回文件描述符；若出错，返回-1
```

我们将最后一个参数写为…，ISO C 用这种方法表明余下的参数的数量及其类型是可变的。对于 open 函数而言，仅当创建新文件时才使用最后这个参数(稍后将对此进行说明)。在函数原型中将此参数放置在注释中。path 参数是要打开或创建文件的名字。oflag 参数可用来说明此函数的多个选项。用下列一个或多个常量进行“或”运算构成 oflag 参数(这些常量在头文件<fcntl.h>中定义)。

O_RDONLY　　只读打开。

O_WRONLY　　只写打开。

O_RDWR　　读、写打开。

大多数实现将 O_RDONLY 定义为 0，O_WRONLY 定义为 1，O_RDWR 定义为 2，以与早期的程序兼容。

O_EXEC　　只执行打开。

O_SEARCH　　只搜索打开(应用于目录)

O_SEARCH 常量的目的在于在目录打开时验证它的搜索权限。对目录的文件描述符的后续操作就不需要再次检查对该目录的搜索权限。本书中涉及的操作系统目前都没有支持 O_SEARCH。

在这 5 个常量中必须指定一个且只能指定一个。下列常量则是可选的。

常量	含义
O_APPEND	每次写时都追加到文件的尾端。3.11 节将详细说明此选项。
O_CLOEXEC	把 FD_CLOEXEC 常量设置为文件描述符标志。3.14 节中将说明文件描述符标志。
O_CREAT	若此文件不存在则创建它。使用此选项时，open 函数需同时说明第 3 个参数 mode（openat 函数需说明第 4 个参数 mode），用 mode 指定该新文件的访问权限位（4.5 节将说明文件的权限位，那时就能了解如何指定 mode，以及如何用进程的 umask 值修改它）。
O_DIRECTORY	如果 path 引用的不是目录，则出错。
O_EXCL	如果同时指定了 O_CREAT，而文件已经存在，则出错。用此可以测试一个文件是否存在，如果不存在，则创建此文件，这使测试和创建两者成为一个原子操作。3.11 节将更详细地说明原子操作。
O_NOCTTY	如果 path 引用的是终端设备，则不将该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_NOFOLLOW	如果 path 引用的是一个符号链接，则出错。4.17 节将说明符号链接。

O_NONBLOCK	如果 path 引用的是一个 FIFO、一个块特殊文件或一个字符特殊文件，则此选项为文件的本次打开操作和后续的 I/O 操作设置非阻塞方式。14.2 节将说明此工作模式。
O_SYNC	使每次 write 等待物理 I/O 操作完成，包括由该 write 操作引起的文件属性更新所需的 I/O。3.14 节将使用此选项。
O_TRUNC	如果此文件存在，而且为只写或读-写成功打开，则将其长度截断为 0。
O_TTY_INIT	如果打开一个还未打开的终端设备，设置非标准 termios 参数值，使其符合 Single UNIX Specification。第 18 章将讨论终端 I/O 的 termios 结构。
O_DSYNC	使每次 write 要等待物理 I/O 操作完成，但是如果该写操作并不影响读取刚写入的数据，则不需等待文件属性被更新。
O_RSYNC	使每一个以文件描述符作为参数进行的 read 操作等待，直至所有对文件同一部分挂起的写操作都完成。

由 open 和 openat 函数返回的文件描述符一定是最小的未用描述符数值。这一点被某些应用程序用来在标准输入、标准输出或标准错误上打开新的文件。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，执行打开操作前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。

四、 函数 creat

也可调用 creat 函数创建一个新文件。

```
#include <fcntl.h>
int creat (const char *path,mode_t mode);
                返回值：若成功，返回为只写打开的文件描述符；若出错，返回-1
```

此函数等价于：

open(path, O_WRONLY | O_CREAT | O_TRUNC, mode);

注： 在早期的 UNIX 系统版本中，open 的第二个参数只能是 0、1 或 2。无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 函数提供了选项 O_CREAT 和 O_TRUNC，于是也就不再需要单独的 creat 函数。

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 treat， close，然后再调用 open。现在则可用下列方式调用 open 实现：

open(path, O_RDWR|O_CREAT|O_TRUNC, mode);

五、 函数 close

可调用 close 函数关闭一个打开文件。

```
#include <unistd.h>
int close (int fd);
                                    返回值：若成功，返回 0；若出错，返回-1
```

关闭一个文件时还会释放该进程加在该文件上的所有记录锁。14.3 节将讨论这一点。当一个进程终止时，内核自动关闭它所有的打开文件。很多程序都利用了这一功能而不显式地用 c 工。se 关闭打开文件。实例见图 1-4 程序。

六、 函数 lseek

每个打开文件都有一个与其相关联的“当前文件偏移量”(current file offset)。它通常是一个非负整数，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偏移量处开始，并使偏移量增加所读写的字节数。按系统默认的情况，当打开一个文件时，除非指定）O_APPEND 选项，否则该偏移量被设置为 0。

可以调用 lseek 显式地为一个打开文件设置偏移量。

```
#include <unistd.h>
off_t lseek(int fd, off_t off_set, int whence);
                        返回值：若成功，则返回新的文件偏移量；若出错，返回-1
```

对参数 offset 的解释与参数 whence 的值有关。

若 whence 是 SEEK_SET，则将该文件的偏移量设置为距文件开始处 offset 个字节。

若 whence 是 SEEK_CUR，则将该文件的偏移量设置为其当前值加 offset，offset 可为正或负。

若 whence 是 SEEK_END，则将该文件的偏移量设置为文件长度加 offset，offset 可正可负。

若 lseek 成功执行，则返回新的文件偏移量，为此可以用下列方式确定打开文件的当前偏移量：

```
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偏移量。如果文件描述符指向的是一个管道、FIFO 或网络套接字，则 lseek 返回-1，并将 errno 设置为 ESPIPE。

七、 函数 read

调用 read 函数从打开文件中读数据。

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t nbytes);
                    返回值：读到的字节数；若已到文件尾，返回 0；若出错，返回-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的字节数：

- 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 18 章将介绍如何改变这一点)。
- 当从网络读时，网络中的缓冲机制可能造成返回值小于所要求读的字节数。
- 当从管道或 FIFO 读时，如若管道包含的字节少于所需的数量，那么 read 将只返回实际可用的字节数。
- 当从某些面向记录的设备(如磁带)读时，一次最多返回一个记录。
- 当一信号造成中断，而已经读了部分数据量时。我们将在 10.5 节进一步讨论此种情况。

八、 函数 write

调用 write 函数向打开文件写数据。

```
#include <unistd.h>
ssize_t write(int fd, const void *buf, size_t nbytes);
                          返回值：若成功，返回已写的字节数；若出错，返回-1
```

其返回值通常与参数 nbytes 的值相同，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偏移量处开始。如果在打开该文件时，指定了 O_APPEND 选项，则在每次写操作之前，将文件偏移量设置在文件的当前结尾处。在一次成功写之后，该文件偏移量增加实际写的字节数。

九、 I/O 的效率

实例：读取文件并将内容复制到标准输出

```
#include "apue.h"
```

```c
#include <fcntl.h>
#define BUFFSIZE 32

int main(void) {
    int n;
    int i;
    char buf[BUFFSIZE];
    int fd;
    for (i=0; i<10000; i++) {
        fd = open("./Makefile",O_RDONLY);
        while((n=read(fd,buf,BUFFSIZE)) > 0)
            if(write(STDOUT_FILENO,buf,n)!=n)
                err_sys("write error");

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

        close(fd);
    }

    exit(0);
}
```

改变 BUFFSIZE 的大小，查看用户 CPU 时间，系统 CPU 时间和时钟时间。

time ./main

十、 文件共享

UNIX 系统支持在不同进程间共享打开文件。在介绍 dup 函数之前，先要说明这种共享。为此先介绍内核用于所有 I/O 的数据结构。

内核使用 3 种数据结构（描述符表，打开文件表，v-node 表）表示打开文件，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，记录项中包含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（close_on_exec）;

   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、添写、同步和非阻塞等，关于这些标志的更多信息参加 3.14 节）;

b) 当前文件偏移量

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(v-node)结构。v 节点包含了文件类型和对此文件进行各种操作函数的指针。对于大多数文件，v 节点还包含了该文件的 i 节点（i-node，索引节点)。这些信息是在打开文件时从磁盘上读入内存的，所以，文件的所有相关信息都是随时可用的。例如，i 节点包含了文件的所有者、文件长度、指向文件实际数据块在磁盘上所在位置的指针等(4.14 节较详细地说明了典型 UNIX 系统文件系统，并将更多地介绍 i 节点)。

注：Linux 没有使用 v 节点，而是使用了通用 i 节点结构。虽然两种实现有所不同，但在概念上，v 节点与 i 节点是一样的。两者都指向文件系统特有的 i 节点结构。

下图显示了一个进程对应的三张表之间的关系。该进程有两个不同的打开文件：一个文件从标准输入打开（文件描述符 0)，另一个从标准输出打开（文件描述符为 1)。

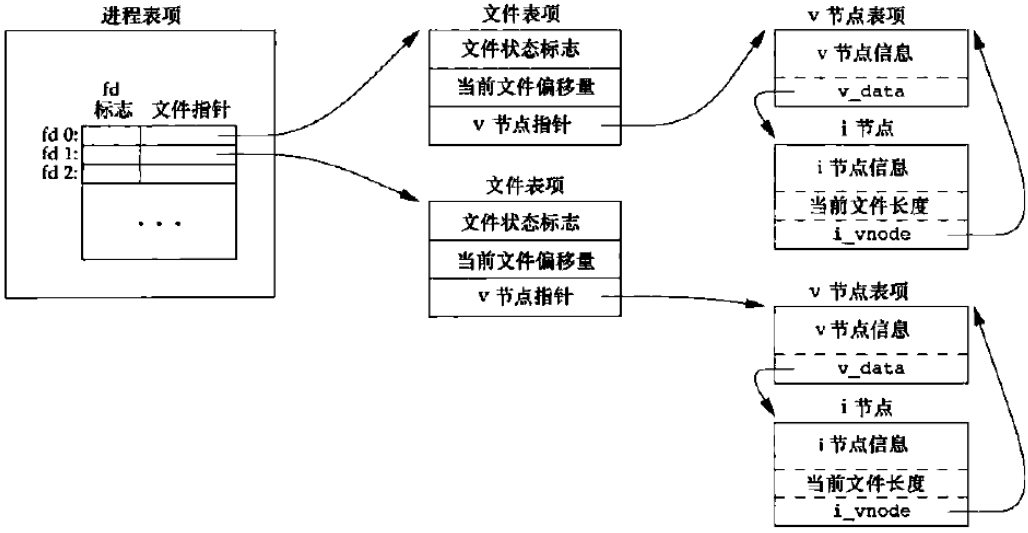

图 1 一个进程打开两个文件

如果两个独立进程各自打开了同一个文件，则有图 2 所示的关系：

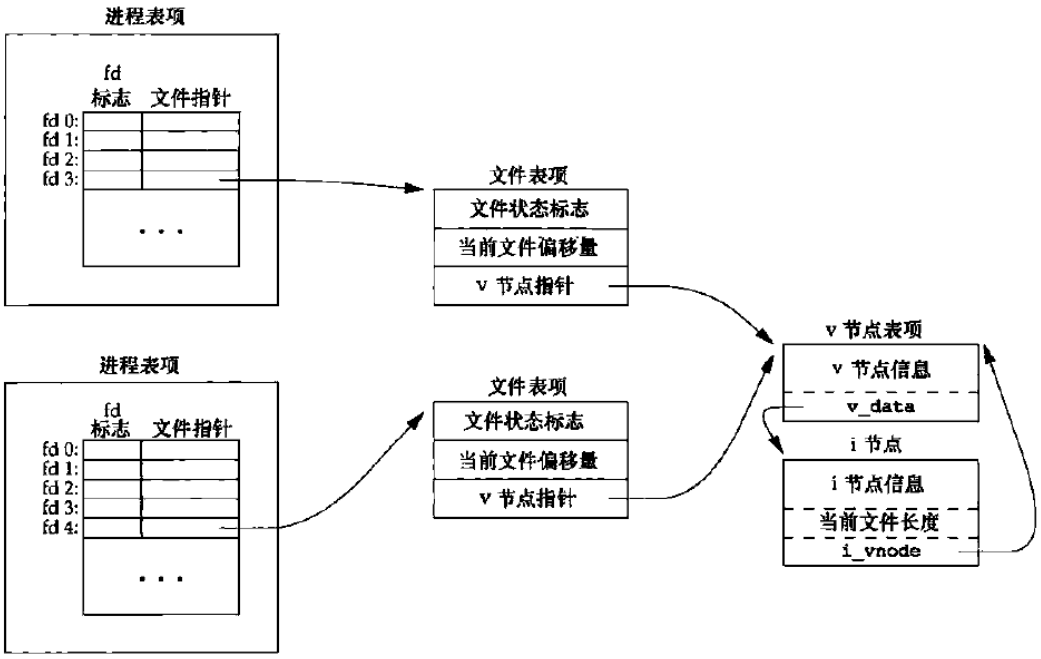

图 2 两个独立进程各自打开同一个文件

我们假定第一个进程在文件描述符 3 上打开该文件，而另一个进程在文件描述符 4 上打开该文件。打开该文件的每个进程都获得各自的一个文件表项，但对一个给定的文件只有一个 v 节点表项。之所以每个进程都获得自己的文件表项，是因为这可以使每个进程都有它自己的对该文件的当前偏移量。

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，而后者则应用于指向该给定文件表项的任何进程中的所有描述符。在 3.14 节说明 fcntl 函数时，我们将会了解如何获取和修改文件描述符标志和文件状态标志。

本节前面所述的一切对于多个进程读取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偏移量。但是，当多个进程写同一文件时，则可能产生预想不到的结果。为了说明如何避免这种情况，需要理解原子操作的概念。

十一、 原子操作

1. 追加到一个文件

考虑一个进程，它要将数据追加到一个文件尾端。早期的 UNIX 系统并不支持 open 的 O_APPEND 选项，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,buf,100) != 100) /*and write*/
```

```
    err_sys("write error");
```

对单个进程而言，这段程序能正常工作，但若有多个进程同时使用这种方法将数据追加写到同一文件，则会产生问题(例如，若此程序由多个进程同时执行，各自将消息追加到一个日志文件中，就会产生这种情况)。

假定有两个独立的进程 A 和 B 都对同一文件进行追加写操作。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 2 中所示。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，它将进程 A 的该文件当前偏移量设置为 1500 字节(当前文件尾端处)。然后内核切换进程，进程 B 运行。进程 B 执行 lseek,也将其对该文件的当前偏移量设置为 1500 字节(当前文件尾端处)。然后 B 调用 write，它将 B 的该文件当前文件偏移量增加至 1600。因为该文件的长度已经增加了，所以内核将 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，就从其当前文件偏移量(1500)处开始将数据写入到文件。这样也就覆盖了进程 B 刚才写入到该文件中的数据。

问题出在逻辑操作“先定位到文件尾端，然后写”，它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。任何要求多于一个函数调用的操作都不是原子操作，因为在两个函数调用之间，内核有可能会临时挂起进程(正如我们前面所假定的)。

UNIX 系统为这样的操作提供了一种原子操作方法，即在打开文件时设置 O_APPEND 标志。正如前一节中所述，这样做使得内核在每次写操作之前，都将进程的当前偏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。

2. 函数 pread 和 pwrite

Single UNIX Specification 包括了 XSI 扩展，该扩展允许原子性地定位并执行 I/O。pread 和 pwrite 就是这种扩展。

```
#include <unistd.h>
ssize_t pread(int fd, void *buf, size_t nbytes, off_t offset);
                    返回值：读到的字节数，若已到文件尾，返回 0；若出错，返回-1
ssize_t pwrite(int fd, const void *buf, size_t nbytes, off_t offset);
                            返回值：若成功，返回已写的字节数；若出错，返回-1
```

调用 pread 相当于调用 lseek 后调用 read，但是 pread 又与这种顺序调用有下列重要区别。

- 调用 pread 时，无法中断其定位和读操作。

- 不更新当前文件偏移量。

调用 pwrite 相当于调用 lseek 后调用 write，但也与它们有类似的区别。

3. 创建一个文件

对 open 函数的 O_CREAT 和 O_EXCL 选项进行说明时，我们已见到另一个有关原子操作的例子。当同时指定这两个选项，而该文件又已经存在时，open 将失败。我们曾提及检查文件是否存在和创建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(path, mode)) < 0)
            err_sys("creat error");
    }
    else {
        err_sys("open error");
    }
}
```

如果在 open 和 creat 之间，另一个进程创建了该文件，就会出现问题。若在这两个函数调用之间，另一个进程创建了该文件，并且写入了一些数据，然后，原先进程执行这段程序中的 creat，这时，刚由另一进程写入的数据就会被擦去。如若将这两者合并在一个原子操作中，这种问题也就不会出现。

一般而言，原子操作(atomic operation)指的是由多步组成的一个操作。如果该操作原子地执行，则要么执行完所有步骤，要么一步也不执行，不可能只执行所有步骤的一个子集。在 4.15 节描述 link 函数以及在 14.3 节中说明记录锁时，还将讨论原子操作。

十二、 函数 dup 和 dup2

下面两个函数都可用来复制一个现有的文件描述符。

```
#include <unistd.h>
int dup(int fd);
int dup2(int fd,int fd2);
                两函数的返回值：若成功，返回新的文件描述符；若出错，返回-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。对于 dup2，可以用 fd2 参数指定新描述符的值。如果 fd2 已经打开，则先将其关闭。如若 fd 等于 fd2，则 dup2 返回 fd2，而不关闭它。否则，fd2 的 FD_CLOEXEC 文件描述符标志就被清除，这样 fd2 在进程调用 exec 时是打开状态。

这些函数返回的新文件描述符与参数 fd 共享同一个文件表项，如图 3 所示。

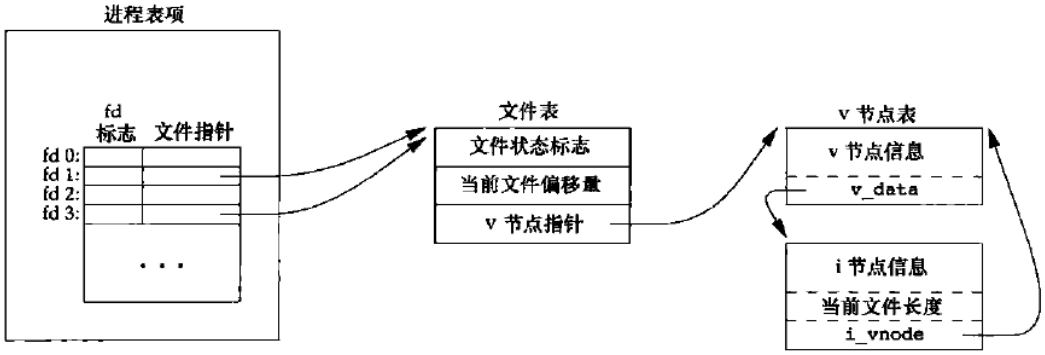

图 3 dup(1)后的内存数据结构

在此图中，我们假定进程启动时执行了：

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(这是非常可能的，因为 0,1 和 2 都由 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、追加等)以及同一当前文件偏移量。

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）标志总是由 dup 函数清除。

复制一个描述符的另一种方法是 fcntl 函数，3.14 节将对该函数进行说明。实际上，

调用

dup(fd);

等效于：

fcntl(fd,F_DUPFD,0);

而调用

dup2(fd,fd2);

则等效于：

fcntl(fd,F_DUPFD,fd2);

在后一种情况下，dup2 并不完全等同于 close 加上 fcntl。它们之间的区别具体如下：

1) dup2 是一个原子操作，而 close 和 fcntl 包括两个函数调用。有可能在 close 和 fcntl 之间调用了信号捕获函数，它可能修改文件描述符(第 10 章将说明信号)。如果不同的线程改变了文件描述符的话也会出现相同的问题(第 11 章将说明线程)。
2) dup2 和 fcntl 有一些不同的 errno。

十三、 函数 sync, fsync 和 fdatasync

传统的 UNIX 系统实现在内核中设有缓冲区高速缓存或页高速缓存，大多数磁盘 I/O 都通过缓冲区进行。当我们向文件写入数据时，内核通常先将数据复制到缓冲区中。然后排入队列，晚些时候再写入磁盘。这种方式被称为延迟写(delayed write)

通常，当内核需要重用缓冲区来存放其他磁盘块数据时，它会把所有延迟写数据块写入磁盘。为了保证磁盘上实际文件系统与缓冲区中内容的一致性，UNIX 系统提供了 sync、fsync 和 fdatasync 三个函数。

```
iinclude<unistd.h>
int fsync(int fd);
int fdatasync(int jd);
                                        返回值：若成功，返回 0；若出错，返回-1
void svnc(void);
```

sync 只是将所有修改过的块缓冲区排入写队列，然后就返回，它并不等待实际写磁盘操作结束。

通常，称为 update 的系统守护进程周期性地调用(一般每隔 30 秒)sync 函数。这就保证了定期冲洗(flush)内核的块缓冲区。命令 sync(1)也调用 sync 函数。

fsync 函数只对由文件描述符 fd 指定的一个文件起作用，并且等待写磁盘操作结束才返回。fsync 可用于数据库这样的应用程序，这种应用程序需要确保修改过的块立即写到磁盘上。

fdatasync 函数类似于 fsync，但它只影响文件的数据部分。而除数据外，fsync 还会同步更新文件的属性。

十四、 函数 fcntl

fcntl 函数可以改变已经打开文件的属性。

```
#include<fcntl.h>
int fcntl(int fd, int cmd, ... /* int arg*/);
                        返回值:若成功，则依赖于 cmd(见下)；若出错，返回一 1
```

在本节的各实例中，第 3 个参数总是一个整数，与上面所示函数原型中的注释部分对应。但是在 14.3 节说明记录锁时，第 3 个参数则是指向一个结构的指针。

fcntl 函数有以下 5 种功能。

(1)复制一个已有的描述符(cmd=F_DUPFD 或 F_DUPFD_CLOEXEC)。

(2)获取/设置文件描述符标志(cmd=F_GETFD 或 F_SETFD)。

(3)获取/设置文件状态标志((cmd=F_GETFL 或 F_SETFL)。

(4)获取/设置异步 I/O 所有权((cmd=F_GETOWN 或 F_SETOWN)。

(5)获取/设置记录锁(cmd=F_GETLK、F_SETLK 或 F_SETLKW)。

十五、 函数 ioctl

ioctl 函数一直是 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 是使用 ioctl 最多的地方(在第 18 章中将看到，POSIX.1 已经用一些单独的函数代替了终端 I/O 操作)。

```
#include <unistd.h> /*System V*/
#include <sys/ioctl.h> /*BSD and Linux*/
int ioctl(int fd, int request, ...);
```

十六、 /dev/fd

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 这一功能是由 Tom Duff 开发的.它首先出现在 Research UNIX 系统的第 8 版中，本书说明的所有 4 种系统(FreeBSD 8.0，Linux 3.2.0，Mac OS X 10.6.8 和 Solaris 10)都支持这一功能。它不是 POSDC.1 的组成部分。

十七、 小结

本章说明了 UNIX 系统提供的基本 I/O 函数。因为 read 和 write 都在内核执行，所以称这些函数为不带缓冲的 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。我们也观察了许多将已写入的数据冲洗到磁盘上的方法，以及它们对应用程序性能的影响。

在说明多个进程对同一文件进行追加写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。在本书的稍后还将涉及这些数据结构。

我们还介绍了 ioctl 和 fcntl 函数，本书后续部分还会涉及这两个函数。第 14 章还将 fcntl 用于记录锁，第 18 章和第 19 章将 ioctl 用于终端设备。