一、UNIX基础知识

1.1 引言

操作系统：一种控制硬件资源，为程序提供运行环境的软件，通常将这种软件成为内核。

系统调用：内核的接口。 公共函数库构建在系统调用上。

1.2 文件和目录

1.2.1 文件系统

文件属性：文件类型（普通文件或目录）、大小、权限、所有者等

目录：是一个包含目录项的文件，UNIX的文件系统大多不在目录项中存储属性。

1.2.2 工作目录与起始目录

每个进程都有一个工作目录，相对路径从工作目录开始解释。进程可以使用chdir函数更改当前工作目录

登录时，工作目录设置为起始目录。登录信息存储在 /etc/passwd 中

1.3 输入输出

1.3.1 文件描述符

通常是一个非负整数。

1.3.2 标准输入、输出、标准错误

每运行一个新程序，shell为其打开3个文件描述符，分别为0 ,1, 2

1.3.3 不带缓冲的 I/O

函数 open read write lseek close 提供不带缓冲区的I/O。

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

    while((n=read(STDIN_FILENO,buf,BUFFSIZE))>0)
        if(write(STDOUT_FILENO,buf,BUFFSIZE)!=n)
        {
        }
    exit(0);
}

./a.out > data   //将shell的输出重定向到data文件中
./a.out < data > data2 //将shell的输入重定向到data中，shell的输出重定向到data2中
```

1.3.4 标准IO （带缓冲区的IO）

无需担心如何选取最佳缓冲区大小

简化了对输入行的处理

标准IO遇到 \n ,或者fflush()函数的时候才会启用系统调用

```
/*用标准IO将标注输入输出到标准输出*/
int main(void)
{
    int c;
    while((c=getc(stdin))!=EOF)
    {
        if(putc(c,stdout)==EOF)
        {
        }
    }
}
```

1.4 程序和进程

1.4.1 程序

内核使用exec函数（7个exec组）将程序读入内存并执行

1.4.2 进程控制

3个用于进程控制的主要函数：fork、exec、waitpid（exec函数有7种变体）

```
int execlp(const char * file,const char * arg,……);
//execlp()会从PATH 环境变量所指的目录中查找符合参数file的文件名，找到后便执行该文件，然后将第二个以后的参数当做该文件的argv[0]、argv[1]……，最后一个参数必须用空指针(NULL)作结束。

示例：
#include<unistd.h>
int main()
{
 execlp("ls","ls","-al","/zhmc",(char *)0);
}
```

1.5 线程控制

一个进程内的所有线程共享同一地址空间、文件描述符、栈以及进程相关的属性，因为他们能访问同一存储区，所以各线程在访问共享数据时需要采取同步措施以避免不一致性。

线程ID只在所属的进程内起作用

1.6 出错处理

1.6.1 出错打印函数

```
char * strerror(int errnum);
void perror(const char *msg);
```

1.7 用户标识

1.7.1 用户ID

0 代表 root

```
int getuid(void);
```

1.7.2 组ID

将用户分组，允许组内共享资源

```
int getgid(void);
```

1.8 时间值

1.8.1 日历时间

自1970.01.01开始的时间，time_t 用于保存这种时间

1.8.2 进程时间

用于度量进程使用CPU时长，unix为每个进程维护3个时间值

时钟时间：进程运行的时间总量

用户CPU时间：执行用户指令所用的时间

系统CPU时间：执行内核程序所用的时间

2. 文件I/O

2.1 文件描述符

当打开或者创建一个新文件时，内核向进程返回一个文件描述符

0：标准输入

1：标准输出

2：标准错误

2.2 Open函数 和Openat函数

```
#include<fcntl.h>
int open(const char *path,int oflag,...)
int openat(int fd,const *path,int oflag,...)

O_APPEND 每次写追加到末尾
O_CREAT 文件不存在时创建它
```

Openat解决两个问题：

让线程可以使用相对路径打开目录中的文件，而不再只能打开当前工作目录

避免TOCTTOU错误

2.3 lseek

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

offset可以大于当前文件长度，会让文件形成空洞，但空洞不会占用物理磁盘块

由于offset可正可负，所以在某些文件系统下32位系统文件最大2GB.

2.4 I/O 效率

预读技术：当检测到顺序读取时，系统视图读入比应用所要求的的更多数据

读相同大小文件时，BUFFSIZE为4096时，系统cpu时间最小，因为磁盘块大小为4096字节。

2.5 文件共享

内核使用三种数据结构表示打开文件，他们之间的关系决定了文件共享方面一个进程对另一个进程产生的影响

(1). 每个进程在进程表中都有一个记录项，记录表中包含一张打开文件描述符表，与每个文件描述符相关联的是：

a. 文件描述符标志，目前只定义了一个标志FD_CLOEXEC（如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项，每个文件表项包含：

a. 文件状态标志（读，写，同步，非阻塞等）

b. 当前文件的偏移量

c. 指向该文件v节点表项的指针

(3). 每个打开文件或者设备都有一个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，对大多数文件还包括该文件的i节点（索引节点）。这些信息是打开文件是从磁盘读入内存的，所以文件的所有相关信息是随时可用的。例如，i节点包含了文件的所有者、文件长度、指向文件的实际数据块在磁盘的具体位置的指针等。

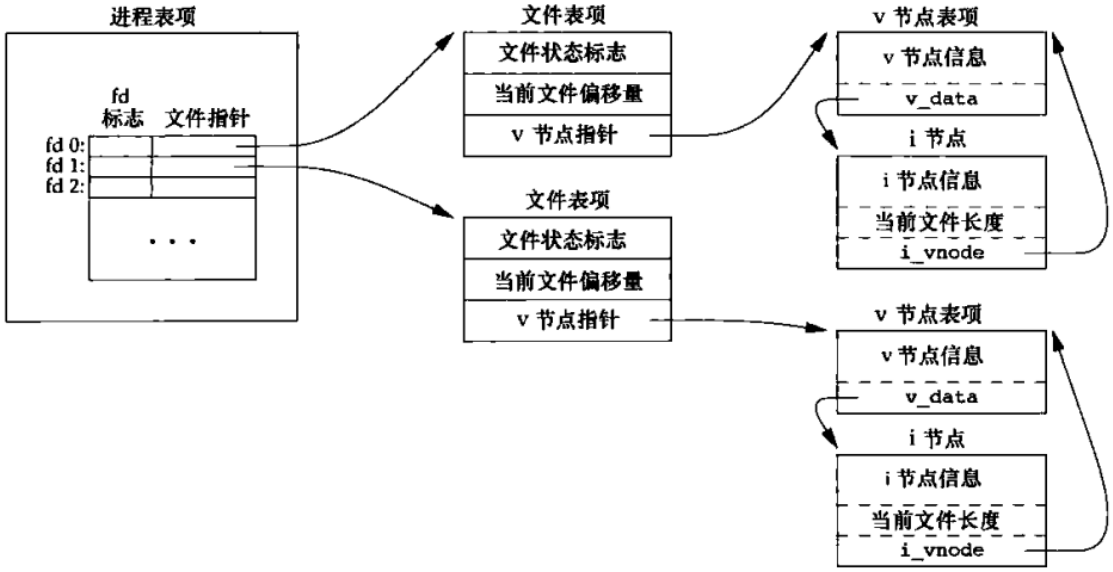

图 3-7　打开文件的内核数据结构

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，创建v节点的目的是对一个计算机系统上的多文件系统提供支持，sun公司将这种文件系统称为VFS，把与文件系统无关的i节点部分称为v节点。

linux没有使用V节点，而是采用一个与文件系统有关的i节点和一个与文件系统无关的i节点。

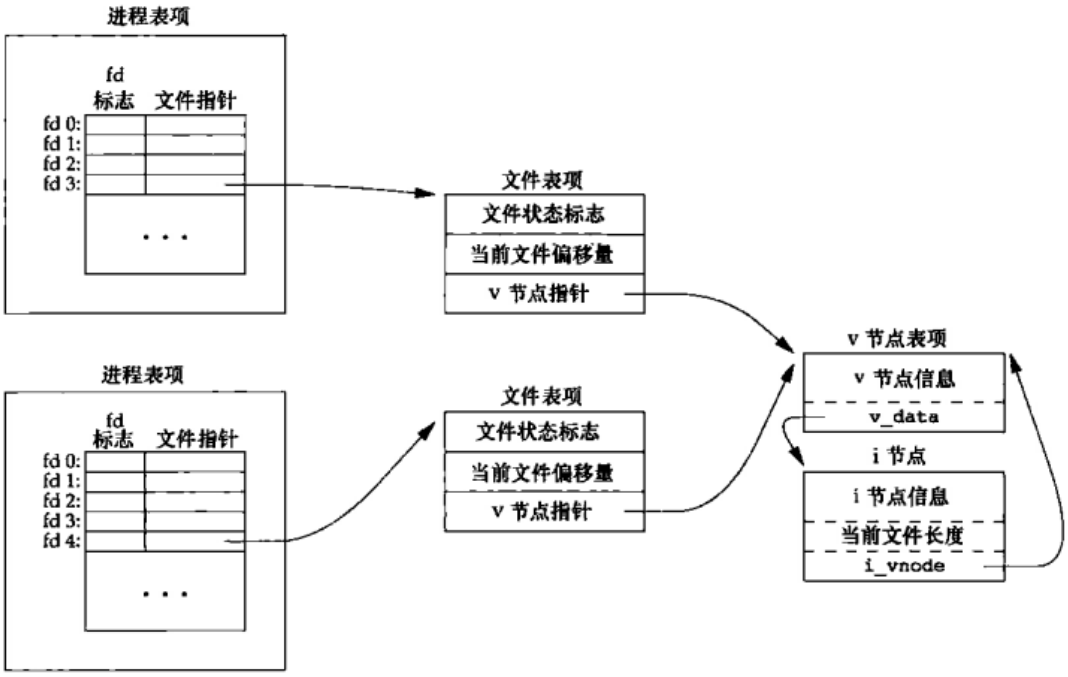

图 3-8　两个独立进程各自打开同一个文件

两个独立的进程各自打开同一个文件，每个进程都获得一个各自的文件表项，但对一个给定的文件只有一个v节点表项。也有多个文件表描述符指向同一个文件表项，如dup函数和fork函数。

文件描述符标志和文件状态描述符标志的区别：

前者只用于一个进程的一个描述符，后者则应用于指向该给定文件表象的任何进程中的所有描述符。

2.6 原子操作

2.6.1 追加到一个文件

假设两个进程同时打开一个文件，均先lseek到文件末尾，A进程先写，B进程再写时会将A进程新写的内容覆盖，出现这一问题的原因是lseek和write不是原子操作，内核为这样的操作提供了一种原子操作的方法，在每次打开文件的时候设置O_APPEND标志，使内核每次在写操作之前，都将进程的偏移量设置到该文件的末尾端处。

2.6.2 函数pread 和 pwrite

```
ssize_t pread(int fd,void *buf,size_t nbytes,off_t offset);
ssize_t pwrite(int fd,void *buf,size_t nbytes,off_t offset);
```

调用pread相当于调用lseek后调用read,但不同的是不会更新当前文件偏移量

2.6.3 创建一个文件

对于open函数的O_CREAT和O_EXCL选项，当同时指定这两个选项而文件存在时，open将失败。

2.7 函数dup 和dup2

```
//复制一个现有的文件描述符

int dup(int fd);
//返回的新文件描述符是当前可用文件描述符中的最小值
int dup2(int fd,int fd2);
//用参数fd2指定新的文件描述符，若fd2已打开,则先将其关闭，若fd等于fd2，则返回fd2，而不关闭它。否则，fd2的FD_CLOEXEC文件描述符标志将被清除，这样fd2在进程调用exec函数时是打开状态。
```

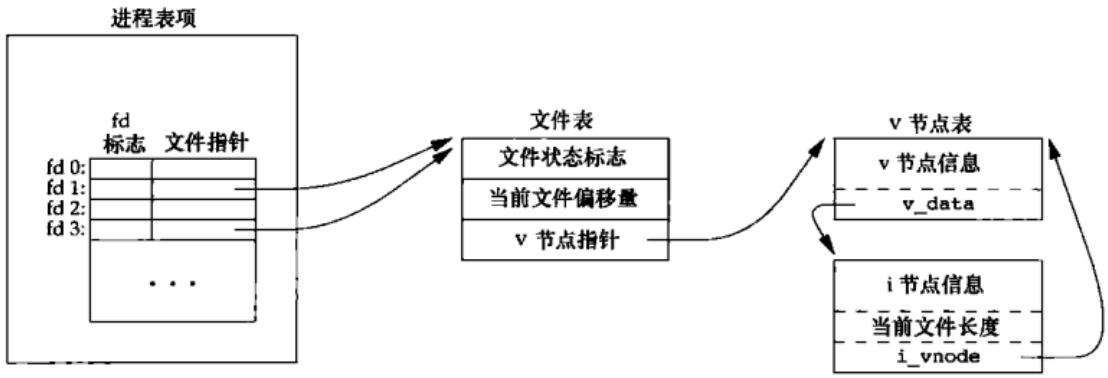

图 3-9 dup(1)后的内核数据结构

```
//复制一个文件描述符的另一种方法是使用fcntl函数
dup(fd); 等效于 fcntl(fd,F_DUPFD,0);
dup2(fd,fd2); 等效于 close(fd2); fcntl(fd,F_dupfd,fd2);的原子操作
```

2.8 sync、fsync、fdatasync

传统UNIX系统在内核中设有缓冲区告诉缓存或页高速缓存，大多数磁盘IO都通过缓冲区进行。当向文件写入数据的时候，内核通常先将数据复制到缓冲区中，然后再排入队列，晚些时候再写入磁盘，这种方式被称为延迟写。通常，当内核要重用来存放其他磁盘块数据时，他会把所有延迟写数据块写入磁盘。为了保证磁盘上的实际文件与缓冲区中的内容一致性，UNIX系统提供了sync、fsync 、fdatasync三个函数。

```
int fsync(int fd);
int fdatasync(int fd);
void sync(void);
```

sync只是将所有修改过的块缓冲区排入写队列，并不等待磁盘的操作结束。

通常，称为update的系统守护进程周期性的调用sync函数定期冲刷块缓冲区，命令sync(1)也调用sync函数。

fsync函数只对fd指定的一个文件起作用，并且等待磁盘操作结束才会返回。可用于数据库这种需要确保修改过的块立即写到磁盘上的应用程序

fdatasync函数类似fsync，但只影响文件的数据部分，而除了数据部分外，fsync函数还会更新文件的属性

2.9 函数fcntl

```
int fcntl(int fd,int cmd,...)

fcntl函数有以下5种功能。
1. 复制一个已有描述符  (cmd=F_DUPFD或cmd= DUPFD_CLOEXEC)
2. 获取/设置 文件描述符标志 （cmd=F_GETFD或F_SETFD）
3. 获取/设置文件描述符标志 （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）

返回值：出错均返回-1
cmd=F_DUPFD 返回新的文件描述符
cmd=F_GETFL cmd=F_GETFD 返回相应的标志
cmd=F_GETOWN 返回进程或进程组ID
```

F_DUPFD

复制文件描述符fd，新的描述符是尚未打开的、大于等于第三个参数的描述符最小值，与fd共享文件表项，但有自己的一套文件系统标志，其FD_CLOEXE文件描述符标志被清除（这表示该描述符在exec时仍然保持有效）

F_DUPFD_CLOEXEC

复制文件描述符，设置与新文件符关联的FD_CLOEXEC文件描述符标志的值，返回新文件描述符

F_GETFD

对应于fd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fd设置文件描述符标志，新标志按照第三个参数设置。

F_GETFL

对应于fd的文件状态标志位作为函数值返回

F_SETFL

将文件状态标志设置为第三个参数的值

F_GETOWN

获取当前接收SIGIO和SIGURG信号的进程ID或进程组ID

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。

```
F_GETFL实例

int main(int argc,char *argv[])
{
    int val;
    if(val=fcntl(atoi(argv[1]),F_GETFL,0))<0)
        err_sys("fcntl error for fd %d",atoi(argv[1]));
    switch(val&O_ACCMODE)
    {
            case: O_RDONLY: printf("read only"); break;
            case: O_WRONLY: printf("write only"); break;
            case: O_RDWR  : printf("read write"); break;
            default:break;
    }
    if(val&O_APPEND) printf("O_APPEND");
    if(val&O_NONBLOCK) printf("O_NONBLOCK");
    if(val&O_SYNC) printf("O_SYNC");
    #if !defined(_POSIX_C_SOURCE)&&defined(O_FSYNC)&&(O_FSYNC != O_SYNC)
        if(val&O_FSYNC) printf("O_FSYNC");
    #endif
}
//atoi函数 字符串转整形
```

```
F_SETFD 实例
void set_fl(int fd,int flags)
{
    int val;
    if((val=fcntl(fd,F_GETFL,0))<0)
        ERR_SYS("error");
    val |=flags;
    if(fcntl(fd,F_SETFL,val)<0)
        err_sys("errot");
}
```

2.10 函数ioctl

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

2.11 /dev/fd

打开文件 /dev/fd/n等效于复制描述符n (假定文件描述符n是打开的)。

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

大多数系统忽略指定的mode，一些系统要求必须是被引用文件初始化打开模式的子集。

linux系统中实现/dev/fd 是个例外，它将/dev/fd 下的文件描述符映射成指向底层物理文件的符号链接。在linux使用创建函数creat，参数为/dev/fd/1时要特别小心，在/dev/fd文件使用creat会导致底层文件被截断。

3.文件和目录

3.1 函数 stat、fstat、fstatat、lstat

```
返回与此命名文件相关的信息结构
int stat(const char *restrict pathname,struct stat *restrict buf);
int fstat(int fd,struct stat *buf);
//前两个函数会透过软链接读取文件结构信息
int lstat(const char *restrict pathname,struct stat *restrict buf);
当命名的文件是一个符号链接时，返回符号链接的相关信息，而不是返回引用文件的相关信息
int fstatat(int fd,const char* restrict pathname,struct stat *restrict buf,int
flag);
//参数fd不是文件描述符，当fd=AT_SYMLINK_NOFOLLOW,返回符号链接本身的信息，相当于lstat
//fd=AT_FDCWD时，且pathname为相对路径，会计算相对当前目录的pathname参数。
//pathname为绝对路径，忽略fd

struct stat {
    unsigned long  st_dev; //device number (file system)
    unsigned long  st_ino; //i-node number （serial number）
    unsigned short st_mode; // file type & mode (permission)
    unsigned short st_nlink; //num of links
    unsigned short st_uid;
    unsigned short st_gid;
    unsigned long  st_rdev; //device number for special file
    unsigned long  st_size;   //size in bytes ,for regular file
    unsigned long  st_blksize; //best IO block size
    unsigned long  st_blocks; //num of disk blocks allocated
    unsigned long  st_atime;
    unsigned long  st_atime_nsec; //time of last access
    unsigned long  st_mtime;
    unsigned long  st_mtime_nsec; //time of last modification
    unsigned long  st_ctime;
    unsigned long  st_ctime_nsec; //time of last file status change
};
```

3.2 文件类型

普通文件

目录文件 包含了其他文件的名字以及指向文件信息的指针，只有内核能写目录文件。

块特殊文件 提供对设备（如磁盘）带缓冲的访问，每次访问以固定长度为单位进行

字符特殊文件 提供对设备的不带缓冲的访问，每次访问的长度可变

FIFO 命名管道 用于进程间通信

socket 用于进程间的网络通信

符号链接 这种类型的文件指向另一个文件

这些类型信息包含在stat结构的st_mode成员中

```c
#include "apue.h"
int
main(int argc, char *argv[])
{
    int i;
    struct stat buf;
    char *ptr;
    for (i = 1; i < argc; i++) {
        printf("%s: ", argv[i]);
        if (lstat(argv[i], &buf) < 0) {
            err_ret("lstat error");
            continue;
        }
        if (S_ISREG(buf.st_mode))
            ptr = "regular";
        else if (S_ISDIR(buf.st_mode))Section 4.3 File Types 97
            ptr = "directory";
        else if (S_ISCHR(buf.st_mode))
            ptr = "character special";
        else if (S_ISBLK(buf.st_mode))
            ptr = "block special";
        else if (S_ISFIFO(buf.st_mode))
            ptr = "fifo";
        else if (S_ISLNK(buf.st_mode))
            ptr = "symbolic link";
        else if (S_ISSOCK(buf.st_mode))
            ptr = "socket";
        else
            ptr = "** unknown mode **";
        printf("%s\n", ptr);
    }
    exit(0);
}
```

```
$ ./a.out /etc/passwd /etc /dev/log /dev/tty \
> /var/lib/oprofile/opd_pipe /dev/sr0 /dev/cdrom
/etc/passwd: regular
/etc: directory
/dev/log: socket
/dev/tty: character special
/var/lib/oprofile/opd_pipe: fifo
/dev/sr0: block special
/dev/cdrom: symbolic link
```

3.3 设置用户ID与组ID

实际用户 ID 实际组 ID	我们实际上是谁
有效用户 ID 有效组 ID 附属组 ID	用于文件访问权限检查
保存的设置用户 ID 保存的设置组 ID	由 exec 函数保存

图 4-5　与每个进程相关联的用户 ID 和组 ID

实际用户ID和实际组ID取自登录文件中的口令项

有效用户ID，有效组ID，附属组ID决定了访问文件的权限

保存的设置用户ID和保存的设置组ID在执行一个程序时包含了有效用户ID和有效组ID的副本

每个文件都有一个所有者和一个组所有者，在stat结构体中1定义

通常有效用户ID等于实际用户ID,有效组ID等于实际组ID，但可以在文件模式字st_mode中设置一个特殊标志，可以将进程的有效用户ID设置为文件所有者的用户ID（st_uid），另一标志位可将有效组ID设置为文件的组所有者ID

文件ID -----> 进程ID

3.4 文件访问权限

当用名字打开任何类型文件时，对该名字包含的每一个目录，包括它可能隐含的当前工作目录都应具有执行权限读权限和执行权限的意义是不相同的。读权限允许我们读目录，获得在该目录中所有文件名的列表。执行权限允许我们通过该目录（即搜索该目录，寻找一个特定的文件名）。

为了在一个目录中创建新文件，必须对该目录有写和执行的权限。

为了在一个目录中删除文件，必须对该目录有写和执行的权限，对文件本身无权限要求。

使用exec函数族执行任何一个文件，必须对该文件具有执行权限，该文件还必须是一个普通文件。

进程没打开、创建、删除一个文件时，内核就进行文件访问权限测试，具体测试流程如下：

1. 若进程有效用户ID为0，允许访问
2. 有效用户ID等于文件所有者ID（进程拥有此文件），若文件的访问权限位被设置（读写执行），则允许访问。
3. 若进程的有效组ID 或附属组ID之一等于文件组，若组适当的访问权限被设置，则允许访问
4. 若其他用户适当访问权限被设置，则允许访问，否则拒绝访问。

3.5 新文件和目录所有权

新文件的用户ID设置为进程的有效用户ID

新文件的组ID可以是进程的有效组ID，也可以是所在目录的组ID

3.6 函数access 和函数 faccessat

```
//按照实际用户ID和实际组ID测试访问权限
int access(const char *pathname,int mode);
int faccessat(int fd,const char *pathname,int mode ,int flag);
mode : R_OK测试读权限 W_OK 测试写权限 X_OK 测试执行权限
faccessat 函数与access函数在下面两种情况是相同的：一种是pathname参数为绝对路径
                                                    ：一种是fd=FDCWD 且pathname为相对路径
flag参数会影响faccessat函数的行为，如果flag设置为AT_EACCESS,访问检查用的是有效用户ID和有效组ID
```

```
#include "apue.h"
#include <fcntl.h>
int
main(int argc, char *argv[])
{
    if (argc != 2)
        err_quit("usage: a.out <pathname>");

    if (access(argv[1], R_OK) < 0)
        err_ret("access error for %s", argv[1]);
    else
        printf("read access OK\n");

    if (open(argv[1], O_RDONLY) < 0)
        err_ret("open error for %s", argv[1]);
    else
        printf("open for reading OK\n");

    exit(0);
}

$ ls -l a.out
-rwxrwxr-x 1 sar 15945 Nov 30 12:10 a.out
$ ./a.out a.out
read access OK
open for reading OK
$ ls -l /etc/shadow
-r-------- 1 root 1315 Jul 17 2002 /etc/shadow
$ ./a.out /etc/shadow
access error for /etc/shadow: Permission denied
open error for /etc/shadow: Permission denied
$ su become superuser
Password: enter superuser password
# chown root a.out change file's user ID to root
# chmod u+s a.out and turn on set-user-ID bit
# ls -l a.out check owner and SUID bit
-rwsrwxr-x 1 root 15945 Nov 30 12:10 a.out
# exit go back to normal user
$ ./a.out /etc/shadow
access error for /etc/shadow: Permission denied
open for reading OK
```

3.7 函数umask

```
//为进程设置文件模式创建屏蔽字
mode_t umask(mode_t cmask);
cmask 由S_IRUSR、S_IWUSR等9个标志位或运算构成，为1则屏蔽相应权限
```

```
#include "apue.h"
#include <fcntl.h>
#define RWRWRW (S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH)
int
main(void)
{
    umask(0);
    if (creat("foo", RWRWRW) < 0)
    err_sys("creat error for foo");
    umask(S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH);
    if (creat("bar", RWRWRW) < 0)
    err_sys("creat error for bar");
    exit(0);
}

$ umask first print the current file mode creation mask
002
$ ./a.out
$ ls -l foo bar
-rw------- 1 sar 0 Dec 7 21:20 bar
-rw-rw-rw- 1 sar 0 Dec 7 21:20 foo
$ umask see if the file mode creation mask changed
002
```

在编写创建新文件的程序时，如果确保指定的访问权限位已经激活，必须在进程运行时修改umask值.

更改进程的文件模式常见屏蔽字并不影响父进程(通常是shell)的屏蔽字

4.8 函数chmod fchmod fchmodat

```
int chmod(const char *pathname ,mode_t mode); //对指定文件进行操作
int fchmod(int fd,mode_t mode); //对已打开文件进行操作
int fchmodat(int fd,const char *pathname ,mode_t mode,int flag);
chmod与fchmodat在下面两种情况是相同的：一种是pathname参数为绝对路径一种是fd=FDCWD 且pathname为相对路径
否则fchmodat计算相对于打开路径（由fd参数指向）的pathname
flag参数用于改变fchmodat的行为，当设置了AT_SYMLINK_NOFOLLOW标志时，fchmodat并不会跟随符号链接.
为了改变一个文件的权限位，进程的有效用户ID必须等于文件所有者ID，或者该进程拥有超级用户权限
参数mode部分说明：
S_ISUID：执行时设置进程用户ID
S_ISGID：执行时设置进程组ID
S_ISVTX：保存正文
```

```
$ ls -l foo bar
-rw------- 1 sar 0 Dec 7 21:20 bar
-rw-rw-rw- 1 sar 0 Dec 7 21:20 foo

#include "apue.h"
int
main(void)
```

```
    {
    struct stat statbuf;
    /* turn on set-group-ID and turn off group-execute */
    if (stat("foo", &statbuf) < 0)
        err_sys("stat error for foo");
    if (chmod("foo", (statbuf.st_mode & ~S_IXGRP) | S_ISGID) < 0)
        err_sys("chmod error for foo");
    /* set absolute mode to "rw-r--r--" */
    if (chmod("bar", S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH) < 0)
        err_sys("chmod error for bar");
    exit(0);
}

$ ls -l foo bar
-rw-r--r-- 1 sar 0 Dec 7 21:20 bar
-rw-rwSrw- 1 sar 0 Dec 7 21:20 foo

时间和日期没有改变，chmod更新的只是i节点最后一次被更改的时间
ls -l 列出的是最后一次文件被更改的时间
```

chmod函数 对某些系统的普通文件粘着位赋予特殊含义，在这些系统上试图设置粘着位但没有root权限，mode中的粘着位被关闭，防止用户恶意设置粘着位，影像系统性能。

新文件的组ID可能不是调用进程所属的组，而是父目录的组ID，这种情况下没有root权限，设置组ID位被自动关闭

防止了用户创建一个设置组ID文件，而该文件并非该用户所属组拥有的。

4.9 粘着位

当粘着位被设置，程序正文副本被保存在交换区内（交换区被当做连续文件处理），如今已不再需要

当对一个目录设置粘着位，只有具有写权限的用户满足 拥有此文件/拥有此目录/root 时，才能删除或重命名该目录下文件

4.10 函数chown、fchown 、fchownat、lchown

```
更改用户ID和组ID，如果两个参数owner或者group中的任何一个是-1，则对应的ID不变
int chown(const char *pathname, uid_t owner, gid_t group);
int fchown(int fd, uid_t owner, gid_t group);
int fchownat(int fd, const char *pathname, uid_t owner, gid_t group,int flag);
int lchown(const char *pathname, uid_t owner, gid_t group);
```

当引用的文件不是符号链接时，这几个函数操作类似

当是符号链接时，lchown和fchownat(设置了AT_SYMLINK_NOFOLLOW标志)对符号链接本身操作

fchown函数改变fd参数指向的打开文件的所有者，不能用于改变符号链接所有者

fchownat函数 当fd=AT_FDCWD并且pathname为相对路径时

1.当flag设置了AT_SYMLINK_NOFOLLOW，fchownat函数与lchown函数类似

2.当flag清除了AT_SYMLINK_NOFOLLOW， fchownat函数与chown函数类似

当 fd 设置为打开目录的文件描述符，并且pathname参数是一个相对路径名，fchownat函数计算相对路径

4.11 文件长度

stat结构成员st_size表示以字节为单位的文件的长度，此字段只对普通文件、目录文件和符号链接有意义

对目录，文件长度通常是一个数（如16或512）的整数倍。

对符号链接，文件长度是在文件名中的实际字节数 lib -> usr/lib 7个字节

用cat命令复制有空洞的文件到新文件时，新文件会被0填满

4.12 文件截断

```
int truncate(const char *pathname,off_t length);
int ftruncate(int fd,off_t length);
```
将一个现有文件截断，当以前的长度大于length时，以后的数据为私有数据不能访问，当小于length时，空余数据补0

4.13 文件系统

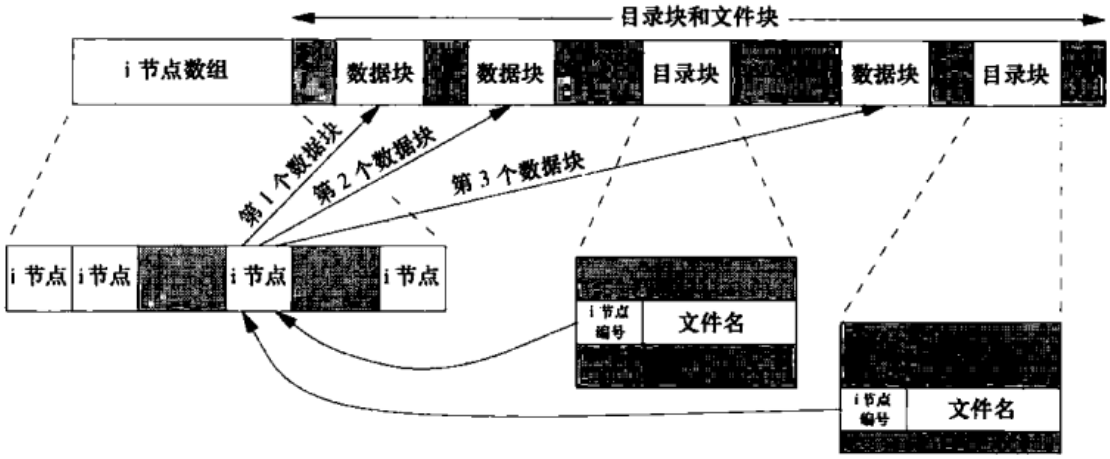

图 4-14　较详细的柱面组的 i 节点和数据块

硬链接：每一个i节点都有一个链接计数，当链接计数为0时才可删除该文件，在stat结构体中包含在st_nlink结构体中，这种类型的链接被称为硬链接

符号链接：文件的实际内容(在数据块中)包含了该符号链接指向文件的名字，该i节点的文件类型为S_IFLNK

只有两项重要的数据存放在目录项中：文件名和i节点编号

一个目录项的不能指向另一个文件系统的i节点，这也是ln命令不能跨文件系统的原因

若在同一个文件系统中，为一个文件重命名时，该文件的实际内容并未移动，只需要构造一个指向现有i节点的新目录项，并删除老目录项，这是mv命令通常的操作方式

对目录文件的引用计数

任何一个叶目录的链接计数总为2,数值2来自目录项自身和.目录项

在父目录中的每一个子目录都使该父目录的链接计数增加1

4.14 函数link linkat unlink unlinkat remove

```
//创建一个硬链接
#include <unistd.h>
int link(const char *existingpath, const char *newpath);
int linkat(int efd, const char *existingpath, int nfd, const char *newpath,
int flag); //可以指定相对路径或绝对路径 可以决定是创建符号链接的链接还是符号链接指向文件的链接
```

创建新目录项和增加链接计数应该是一个原子操作

```
删除目录项 i节点的引用计数减1
int unlink(const char *pathname);
int unlinkat(int fd, const char *pathname, int flag);
//flag参数 给出一种方法可以使unlinkat函数删除目录
//pathname是符号链接,那么unlink函数删除该符号链接，而不是删除该链接引用的文件
```

unlink的另一个作用就是创建临时文件，在文件open后立即使用unlink函数

```
#include "apue.h"
#include <fcntl.h>
int main(void)
{
    if (open("tempfile", O_RDWR) < 0)
    err_sys("open error");
    if (unlink("tempfile") < 0)
    err_sys("unlink error");
    printf("file unlinked\n");
    sleep(15);
    printf("done\n");
    exit(0);
}
//这种特性被用来确保即使程序崩溃时，其创建的临时文件也不会被留下来
```

4.15 函数rename和 renameat

```
重命名文件目录
#include <stdio.h>
int rename(const char *oldname, const char *newname);
int renameat(int oldfd, const char *oldname, int newfd,
const char *newname);
//若oldname指的是一个文件，且newname已经存在，此时newname不能是目录，将新文件删除，然后进程
将文件的目录项的旧文件名改为新文件名
//若oldname是一个目录，且newname存在，则该目录应该为空目录，将旧目录删除，然后将文件目录项的旧
文件名该为新文件名
//若oldname newname引用符号链接，则处理符号链接本身
//不能对 . ..重命名
```

4.16 符号链接

引入符号链接为了避免一些限制：

1.硬链接通常要求链接和文件处于同一文件系统中

2.只有超级用户才能创建指向文件目录的硬链接

使用符号链接可能引入一个循环，大多数查找路径名的函数在这种情况下返回errno=ELOOP

```
$ mkdir foo //make a new directory
$ touch foo/a  //create a 0-length file
$ ln -s ../foo foo/testdir //create a symbolic link
$ ls -l foo
```

4.17 创建和读取符号链接

```
#include <unistd.h>
int symlink(const char *actualpath, const char *sympath);
int symlinkat(const char *actualpath, int fd, const char *sympath);
//Both return: 0 if OK, -1 on error
```

在创建符号链接时，并不要求actualpath存在,且actualpath与sympath可以不位于同一文件系统中

```
//由于open函数跟随符号链接，需要一种方法打开该链接本身，并读取链接中的名字
ssize_t readlink(const char* restrict pathname, char *restrict buf,
                    size_t bufsize);
ssize_t readlinkat(int fd, const char* restrict pathname,
                        char *restrict buf, size_t bufsize);
//Both return: number of bytes read if OK, -1 on error
```

4.18 文件的时间

st_atim 文件数据最后访问的时间 read

st_mtim 文件数据最后修改的时间 write

st_ctim 文件数据最后更改的时间