第4章　文件和目录

4.1　引言

上一章我们说明了执行I/O操作的基本函数。其讨论围绕普通文件的I/O进行——打开一个文件，读或写一个文件。本章将观察文件系统的其他特征和文件的性质。我们将从stat函数开始，逐个说明stat结构的每一个成员以了解文件的所有属性。在此过程中，我们将说明修改这些属性的各个函数(更改所有者，更改许可权等)，还将更详细地察看UNIX文件系统的结构以及符号连接。本章结束部分介绍对目录进行操作的各个函数，并且开发了一个以降序遍历目录层次结构的函数。

4.2　stat、fstat和lstat函数

本章讨论的中心是三个stat函数以及它们所返回的信息。

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

int stat(const char *pathname, struct stat *buf);

int fstat(int filedes, struct stat *buf);

int lstat(const char *pathname, struct stat *buf);
```

三个函数的返回：若成功则为0，若出错则为-1

给定一个*pathname*，stat函数返回一个与此命名文件有关的信息结构，fstat函数获得已在描述符*filedes*上打开的文件的有关信息。lstat函数类似于stat，但是当命名的文件是一个符号连接时，lstat返回该符号连接的有关信息，而不是由该符号连接引用的文件的信息。(在4.21节中当以降序遍历目录层次结构时，需要用到lstat。4.16节将较详细地说明符号连接。)

> lstat函数不属于POSIX 1003.1-1990标准，但很可能加到1003.1a中。SVR4和4.3+BSD支持lstat。

第二个参数是个指针，它指向一个我们应提供的结构。这些函数填写由buf指向的结构。该结构的实际定义可能随实现而有所不同，但其基本形式是：

```
struct stat {
  mode_t   st_mode;    /* file type & mode (permissions) */
  ino_t    st_ino;     /* i-node number (serial number) */
  dev_t    st_dev;     /* device number (filesystem) */
  dev_t    st_rdev;    /* device number for special files */
  nlink_t  st_nlink;   /* number of links */
  uid_t    st_uid;     /* user ID of owner */
  gid_t    st_gid;     /* group ID of owner */
  off_t    st_size;    /* size in bytes, for regular files */
```

```
  time_t  st_atime;   /* time of last access */
  time_t  st_mtime;   /* time of last modification */
  time_t  st_ctime;   /* time of last file status change */
  long    st_blksize; /* best I/O block size */
  long    st_blocks;  /* number of 512-byte blocks allocated */
};
```

> POSIX.1未定义st_rdev，st_blksize和st_blocks字段，SVR4和4.3+BSD则定义了这些字段。

注意，除最后两个以外，其他各成员都为基本系统数据类型 (见2.7节)。我们将说明此结构的每个成员以了解文件属性。

使用stat函数最多的可能是ls -l命令，用其可以获得有关一个文件的所有信息。

4.3 文件类型

至今我们已介绍了两种不同的文件类型 —— 普通文件和目录。UNIX系统的大多数文件是普通文件或目录，但是也有另外一些文件类型：

(1) 普通文件(regular file)。这是最常见的文件类型，这种文件包含了某种形式的数据。至于这种数据是文本还是二进制数据对于内核而言并无区别。对普通文件内容的解释由处理该文件的应用程序进行。

(2) 目录文件(directory file)。这种文件包含了其他文件的名字以及指向与这些文件有关信息的指针。对一个目录文件具有读许可权的任一进程都可以读该目录的内容，但只有内核可以写目录文件。

(3) 字符特殊文件(character special file)。这种文件用于系统中某些类型的设备。

(4) 块特殊文件(block special file)。这种文件典型地用于磁盘设备。系统中的所有设备或者是字符特殊文件，或者是块特殊文件。

(5) FIFO。这种文件用于进程间的通信，有时也将其称为命名管道。14.5节将对其进行说明。

(6) 套接口(socket)。这种文件用于进程间的网络通信。套接口也可用于在一台宿主机上的进程之间的非网络通信。第15章将用套接口进行进程间的通信。

> 只有4.3+BSD才返回套接口文件类型，虽然SVR4支持用套接口进行进程间通信，但现在是经由套接口函数库实现的，而不是通过内核内的套接口文件类型，将来的SVR4版本可能会支持套接口文件类型。

(7) 符号连接(symbolic link)。这种文件指向另一个文件。4.16节将更多地述及符号连接。

文件类型信息包含在stat结构的st_mode成员中。可以用表4-1中的宏确定文件类型。这些宏的参数都是stat结构中的st_mode成员。

表4-1 在<sys/stat.h>中的文件类型宏

宏	文件类型
S_ISREG()	普通文件
S_ISDIR()	目录文件
S_ISCHR()	字符特殊文件
S_ISBLK()	块特殊文件

（续）

宏	文 件 类 型
S_ISFIFO()	管道或FIFO
S_ISLNK()	符号连接（POSIX.1或SVR4无此类型）
S_ISSOCK()	套接字（POSIX.1或SVR4无此类型）

实例

程序4-1取其命令行参数，然后针对每一个命令行参数打印其文件类型。

程序4-1 对每个命令行参数打印文件类型

```
#include    <sys/types.h>
#include    <sys/stat.h>
#include    "ourhdr.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))  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";
#ifdef  S_ISLNK
        else if (S_ISLNK(buf.st_mode))  ptr = "symbolic link";
#endif
#ifdef  S_ISSOCK
        else if (S_ISSOCK(buf.st_mode)) ptr = "socket";
#endif
        else                    ptr = "** unknown mode **";
        printf("%s\n", ptr);
    }
    exit(0);
}
```

程序4-1的样本输出是：

```
$ a.out /vmunix /etc /dev/ttya /dev/sd0a /var/spool/cron/FIFO \
> /bin /dev/printer
/vmunix: regular
/etc: directory
/dev/ttya: character special
/dev/sd0a: block special
/var/spool/cron/FIFO: fifo
/bin: symbolic link
/dev/printer: socket
```

其中，在第一命令行末端我们键入了一个反斜线，通知 shell要在下一行继续键入命令，然后shell在下一行上用其第二提示符>。我们特地使用了lstat函数而不是stat函数以便检测符号连接。如若使用了stat函数，则决不会观察到符号连接。

早期的UNIX版本并不提供S_ISxxx宏，于是就需要将st_mode与屏蔽字S_IFMT逻辑与，然后与名为S_IFxxx的常数相比较。SVR4和4.3+BSD在文件<sys/stat.h>中定义了此屏蔽字和相关的常数。如若查看此文件，则可找到S_ISDIR宏定义为：

```
#define S_ISDIR (mode) (((mode) & S_IFMT) == S_IFDIR)
```

我们说过，普通文件是最主要的文件类型，但是观察一下在一个给定的系统中各种文件的比例是很有意思的。表4-2中显示了在一个中等规模的系统中的统计值。这一数据是由 4.21节中的程序得到的。

表4-2 不同类型文件的计数值和比例

文件类型	计数值	比例(%)
普通文件	30 369	91.7
目录	1 901	5.7
符号连接	416	1.3
字符特殊	373	1.1
块特殊	61	0.2
套接口	5	0.0
FIFO	1	0.0

4.4 设置-用户-ID和设置-组-ID

与一个进程相关联的ID有六个或更多，它们示于表4-3中。

表4-3 与每个进程相关联的用户 ID和组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 添加组ID	用于文件存取许可权检查
	保存设置-用户-ID 保存设置-组-ID	由exec函数保存

• 实际用户ID和实际组ID标识我们究竟是谁。这两个字段在登录时取自口令文件中的登录项。通常，在一个登录会话期间这些值并不改变，但是超级用户进程有方法改变它们， 8.10节将说明这些方法。

• 有效用户ID，有效组ID以及添加组ID决定了我们的文件访问权，下一节将对此进行说明(我们已在1.8节中说明了添加组ID)。

• 保存的设置-用户-ID和设置-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，在8.10节中说明setuid函数时，将说明这两个保存值的作用。

> 在POSIX.1中，这些保存的ID是可选择的。一个应用程序在编译时可测试常数_POSIX_SAVED_IDS，或在运行时以参数_SC_SAVED_IDS调用函数sysconf，以判断此实现是否支持这种特征。SVR4支持此特征。
>
> FIPS 151-1要求POSIX.1的这种可选择特征。

通常，有效用户ID等于实际用户ID，有效组ID等于实际组ID。

每个文件有一个所有者和组所有者，所有者由 stat结构中的st_uid表示，组所有者则由st_gid成员表示。

当执行一个程序文件时，进程的有效用户 ID通常就是实际用户ID，有效组ID通常是实际组ID。但是可以在文件方式字(st_mode)中设置一个特殊标志，其定义是“当执行此文件时，将进程的有效用户ID设置为文件的所有者(st_uid)”。与此相类似，在文件方式字中可以设置另一位，它使得执行此文件的进程的有效组 ID设置为文件的组所有者(st_gid)。在文件方式字中的这两位被称之为设置-用户-ID(set-user-ID)位和设置-组-ID(set-group-ID)位。

例如，若文件所有者是超级用户，而且设置了该文件的设置 -用户-ID位，然后当该程序由一个进程运行时，则该进程具有超级用户优先权。不管执行此文件的进程的实际用户ID是什么，都作这种处理。作为一个例子， UNIX程序passwd(1)允许任一用户改变其口令，该程序是一个设置-用户-ID程序。因为该程序应能将用户的新口令写入口令文件中 (一般是/etc/passwd或/etc/shadow)，而只有超级用户才具有对该文件的写许可权，所以需要使用设置 -用户-ID特征。因为运行设置-用户-ID程序的进程通常得到额外的许可权，所以编写这种程序时要特别谨慎。第8章将更详细地讨论这种类型的程序。

再返回到stat函数，设置-用户-ID位及设置-组-ID位都包含在st_mode值中。这两位可用常数S_ISUID和S_ISGID测试。

4.5 文件存取许可权

st_mode值也包含了对文件的存取许可权位。当提及文件时，指的是前面所提到的任何类型的文件。所有文件类型(目录，字符特别文件等)都有存取许可权。很多人认为只有普通文件有存取许可权，这是一种误解。

每个文件有9个存取许可权位，可将它们分成三类，见表4-4。

在表4-4开头三行中，术语用户指的是文件所有者。 chmod(1)命令用于修改这9个许可权位。该命令允许我们用 u表示用户(所有者)，用g表示组，用o表示其他。有些书把这三种用户类型分别称之为所有者，组和世界。这会造成混乱，因为chmod命令用o表示其他，而不是所有者。我们将使用术语用户、组和其他，以便与chmod命令一致。

图中的三类存取许可权——读、写及执行——以各种方式由不同的函数使用。我们将这些不同的使用方法列在下

表4-4 9个存取许可权位，取自<sys/stat.h>

st_mode屏蔽	意 义
S_IRUSR	用户-读
S_IWUSR	用户-写
S_IXUSR	用户-执行
S_IRGRP	组-读
S_IWGRP	组-写
S_IXGRP	组-执行
S_IROTH	其他-读
S_IWOTH	其他-写
S_IXOTH	其他-执行

面，当说明这些函数时，再进一步作讨论。

• 第一个规则是，我们用名字打开任一类型的文件时，对该名字中包含的每一个目录，包括它可能隐含的当前工作目录都应具有执行许可权。这就是为什么对于目录其执行许可权位常被称为搜索位的原因。

例如，为了打开文件/usr/dict/words，需要具有对目录/, /usr, /usr/dict的执行许可权。然后，需要对该文件本身的适当许可权，这取决于以何种方式打开它(只读，读-写等)。

如果当前目录是/usr/dict,那么为了打开文件words，需要有对该目录的执行许可。如在指定打开文件words时，可以隐含当前目录，而不用显式地提及/usr/dict，也可使用./words。

注意，对于目录的读许可权和执行许可权的意义不相同。读许可权允许我们读目录，获得在该目录中所有文件名的列表。当一个目录是我们要存取文件的路径名的一个分量时，对该目录的执行许可权使我们可通过该目录(也就是搜索该目录，寻找一个特定的文件名)。

引用隐含目录的另一个例子是，如果 PATH环境变量(8.4节将说明)指定了一个我们不具有执行许可权的目录，那么shell决不会在该目录下找到可执行文件。

• 对于一个文件的读许可权决定了我们是否能够打开该文件进行读操作。这对应于 open函数的O_RDONLY和O_RDWR标志。

• 对于一个文件的写许可权决定了我们是否能够打开该文件进行写操作。这对应于 open函数的O_WRONLY和O_RDWR标志。

• 为了在open函数中对一个文件指定O_TRUNC标志，必须对该文件具有写许可权。

• 为了在一个目录中创建一个新文件，必须对该目录具有写许可权和执行许可权。

• 为了删除一个文件，必须对包含该文件的目录具有写许可权和执行许可权。对该文件本身则不需要有读、写许可权。

• 如果用6个exec函数（见8.9节）中的任何一个执行某个文件，都必须对该文件具有执行许可权。

进程每次打开、创建或删除一个文件时，内核就进行文件存取许可权测试，而这种测试可能涉及文件的所有者(st_uid和st_gid)，进程的有效ID (有效用户ID和有效组ID)以及进程的添加组ID (若支持的话)。两个所有者ID是文件的性质，而有效ID和添加组ID则是进程的性质。内核进行的测试是：

(1) 若进程的有效用户ID是0 (超级用户)，则允许存取。这给予了超级用户对文件系统进行处理的最充分的自由。

(2) 若进程的有效用户ID等于文件的所有者ID(也就是该进程拥有此文件)：

(a) 若适当的所有者存取许可权位被设置，则允许存取。

(b) 否则拒绝存取。

适当的存取许可权位指的是，若进程为读而打开该文件，则用户 -读位应为1；若进程为写而打开该文件，则用户-写位应为1；若进程将执行该文件，则用户-执行位应为1。

(3) 若进程的有效组ID或进程的添加组ID之一等于文件的组ID：

(a) 若适当的组存取许可权位被设置，则允许存取。

(b) 否则拒绝存取。

(4) 若适当的其他用户存取许可权位被设置，则允许存取，否则拒绝存取。

按顺序执行这四步。注意，如若进程拥有此文件 (第(2)步)，则按用户存取许可权批准或拒绝该进程对文件的存取——不查看组存取许可权。相类似，若进程并不拥有该文件。但进程属于某个适当的组，则按组存取许可权批准或拒绝该进程对文件的存取——不查看其他用户的存

取许可权。

4.6 新文件和目录的所有权

在第3章中，当说明用open或creat创建新文件时，没有说明赋予新文件的用户ID和组ID的值是什么。4.20节将说明如何创建一个新目录以及mkdir函数。关于新目录的所有权的规则与本节将说明的新文件的所有权的规则相同。

新文件的用户ID设置为进程的有效用户ID。关于组ID，POSIX.1允许选择下列之一作为新文件的组ID。

(1) 新文件的组ID可以是进程的有效组ID。

(2) 新文件的组ID可以是它所在目录的组ID。

> 在SVR4中，新文件的组ID取决于它所在的目录的设置-组-ID位是否设置。如果该目录的这一位已经设置，则新文件的组ID设置为目录的组ID；否则新文件的组ID设置为进程的有效组ID。
>
> 4.3+BSD总是使用目录的组ID作为新文件的组ID。
>
> 其他系统允许以一个文件系统作为单位在POSIX.1所允许的两种方法中选择一种，为此在mount(1)命令中使用了一个特殊标志。
>
> FIPS 151-1要求一个新文件的组ID是它所在目录的组ID。

使用POSIX.1所允许的第二种方法(继承目录的组ID)使得在某个目录下创建的文件和目录都具有该目录的组ID。于是文件和目录的组所有权从该点向下传递。例如，在/var/spool目录中就使用这种方法。

> 正如前面提到的，这种设置组所有权的方法对4.3+BSD是系统默认的，对SVR4则是可选择的。在SVR4之下，必须设置-组-ID位。更进一步，为使这种方法能够正常工作，SVR4的mkdir函数要自动地传递一个目录的设置-组-ID位（4.20节将说明mkdir就是这样做的）。

4.7 access函数

正如前面所说，当用open函数打开一个文件时，内核以进程的有效用户ID和有效组ID为基础执行其存取许可权测试。有时，进程也希望按其实际用户ID和实际组ID来测试其存取能力。例如当一个进程使用设置-用户-ID，或设置-组-ID特征作为另一个用户(或组)运行时，这就可能需要。即使一个进程可能已经设置-用户-ID为根，它仍可能想验证实际用户能否存取一个给定的文件。access函数是按实际用户ID和实际组ID进行存取许可权测试的。(经过4.5节结束部分中所述的四个步骤，但将有效改为实际。)

```
#include<unistd.h>

int access(const char *pathname, int mode);
```

返回：若成功则为0，若出错则为-1

其中，*mode*是表4-5中所列常数的逐位或运算。

表4-5 access函数的*mode*常数，取自<unistd.h>

mode	说 明
R_OK	测试读许可权
W_OK	测试写许可权
X_OK	测试执行许可权
F_OK	测试文件是否存在

实例

程序4-2显示了access函数的使用。

程序4-2 access函数实例

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

下面是该程序的一些运行结果：

```
$ ls -1 a.out
-rwxrwxr-x  1 stevens    105216 Jan 18 08:48 a.out
$ a.out a.out
read access OK
open for reading OK
$ ls -1/ etc/uucp/Systems
-rw-r----- 1 uucp         1441 Jul 18 15:05 /etc/uucp/Systems
$ a.out /etc/uucp/Systems
access error for /etc/uucp/Systems: Permission denied
open error for /etc/uucp/Systems: Permission denied
$ su                                成为超级用户
Password:                           输入超级用户口令
# chown uucp a.out                  将文件用户改为uucp
# chmod u+s a.out                   并打开设置-用户-ID位

$ ls -1 a.out                   检查所有者和SUID位
-rwsrwxr-x  1 uucp    105216 Jan 18 08:48 a.out
```

```
# exit                                  回到正常用户
$ a.out /etc/uucp/Systems
access error for /etc/uucp/Systems: Permission denied
open for reading OK
```

在本例中，设置-用户-ID程序可以确定实际用户不能读某个文件，而 open函数却能打开该文件。

> 在上例及第8章中，我们有时要成为超级用户，以便例示某些功能是如何工作的。如果你使用多用户系统，但无超级用户许可权，那么你就不能完整地重复这些实例。

4.8 umask函数

至此我们已说明了与每个文件相关联的 9个存取许可权位，在此基础上我们可以说明与每个进程相关联的文件方式创建屏蔽字。

umask函数为进程设置文件方式创建屏蔽字，并返回以前的值。 (这是少数几个没有出错返回的函数中的一个。)

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

mode_t umask(mode_t cmask);
```

返回：以前的文件方式创建屏蔽字

其中，参数*cmask*由表4-4中的9个常数(S_IRUSR,S_IWUSR等)逐位“或”构成的。

在进程创建一个新文件或新目录时，就一定会使用文件方式创建屏蔽字 (回忆3.3和3.4节，在那里我们说明了 open和creat函数。这两个函数都有一个参数 *mode*，它指定了新文件的存取许可权位)。我们将在4.20节说明如何创建一个新目录，在文件方式创建屏蔽字中为 1的位，在文件*mode*中的相应位则一定被转成0。

实例

程序4-3创建了两个文件，创建第一个时， umask值为0，创建第二个时， umask值禁止所有组和其他存取许可权。若运行此程序可得如下结果，从中可见存取许可权是如何设置的。

```
$ umask                                 第一次打印当前文件方式创建屏蔽字
02
$ a.out
4 ls -1 foo bar
-rw-------  1 stevens          0 Nov 16 16:23 bar
-rw-rw-rw-  1 stevens          0 Nov 16 16:23 foo
$ umask                                 观察文件方式创建屏蔽字是否更改
02
```

程序4-3 umask函数实例

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

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

int
main(void)
{
    umask(0);
    if (creat("foo", S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP |
                                         S_IROTH | S_IWOTH) < 0)
        err_sys("creat error for foo");

    umask(S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH);
    if (creat("bar", S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP |
                                         S_IROTH | S_IWOTH) < 0)
        err_sys("creat error for bar");
    exit(0);
}
```

4.9 chmod和fchmod函数

这两个函数使我们可以更改现存文件的存取许可权。

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

int chmod(const char *pathname, mode_t mode);

int fchmod(int filedes, mode_t mode);
```

两个函数返回：若成功则为0，若出错则为-1

chmod函数在指定的文件上进行操作，而fchmod函数则对已打开的文件进行操作。

> fchmod函数并不是POSIX.1的组成部分。它是SVR4和4.3+BSD的扩充部分。

为了改变一个文件的许可权位，进程的有效用户ID必须等于文件的所有者，或者该进程必须具有超级用户许可权。

参数*mode*是表4-6中所示常数的某种逐位或运算。

表4-6 chmod函数的*mode*常数，取自<sys/stat.h>

mode	说 明
S_ISUID	执行时设置-用户-ID
S_ISGID	执行时设置-组-ID
S_ISVTX	保存正文
S_IRWXU	用户（所有者）读、写和执行
S_IRUSR	用户（所有者）读
S_IWUSR	用户（所有者）写
S_IXUSR	用户（所有者）执行
S_IRWXG	组读、写和执行
S_IRGRP	组读
S_IWGRP	组写
S_IXGRP	组执行

（续）

mode	说　明
S_IRWXO	其他读、写和执行
S_IROTH	其他读
S_IWOTH	其他写
S_IXOTH	其他执行

注意，在表4-6中，有9项是取自表4-4中的9个文件存取许可权位。我们另外加上了两个设置-ID常数(S_IS〔UG〕ID), 保存-正文常数(S_ISVTX)，以及三个组合常数(S_IRWX〔UGO〕)。(这里使用了标准UNIX字符类算符〔　〕，表示方括号算符中的任何一个字符。例如，最后一个，S_IRWX〔UGO〕表示了三个常数：S_IRWXU、S_IRWXG和S_IRWXO。这一字符类算符是大多数UNIX shell和很多标准UNIX应用程序都提供的正规表达式的一种形式。)

> 保存-正文位(S_ISVTX)不是POSIX.1的一部分。我们在下一节说明其目的。

实例

先回忆一下为例示umask函数我们运行程序4-3时，文件foo和bar的最后状态：

```
$ ls -1 foo bar
-rw-------  1 stevens          0 Nov 16 16:23 bar
-rw-rw-rw-  1 stevens          0 Nov 16 16:23 foo
```

程序4-4修改了这两个文件的方式。在运行程序4-4后，我们见到的这两个文件的最后状态是：

```
$ ls -1 foo bar
-rw-r--r--  1 stevens          0 Nov 16 16:23 bar
-rw-rwlrw-  1 stevens          0 Nov 16 16:23 foo
```

在本例中，我们相对于foo的当前状态设置其许可权。为此，先调用stat获得其当前许可权，然后修改它。我们已显式地打开了设置-组-ID位、关闭了组-执行位。对普通文件这样做的结果是对该文件可以加强制性记录锁，我们将在12.3节中讨论强制性锁。注意，ls命令将组-执行许可权表示为l，它表示对该文件可以加强制性记录锁。对文件bar，不管其当前许可权位如何，我们将其许可权设置为一绝对值。

程序4-4　chmod函数实例

```
#include    <sys/types.h>
#include    <sys/stat.h>
#include    "ourhdr.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);
}
```

最后也要注意到，在我们运行程序4-4后，ls命令列出的时间和日期并不改变。在4.18节中，我们会了解到chmod函数更新的只是i节点最近一次被更改的时间。按系统默认方式， ls -l列出的是最后修改文件内容的时间。

chmod函数在下列条件下自动清除两个许可权位。

• 如果我们试图设置普通文件的粘住位 (S_ISVTX)，而且又没有超级用户优先权，那么*mode*中的粘住位自动被关闭(我们将在下一节说明粘住位)。这意味着只有超级用户才能设置普通文件的粘住位。这样做的理由是可以防止不怀好意的用户设置粘住位，并试图以此方式填满交换区(如果系统支持保存-正文特征的话)。

• 新创建文件的组ID可能不是调用进程所属的组。回忆一下4.6节，新文件的组ID可能是父目录的组ID。特别地，如果新文件的组ID不等于进程的有效组ID或者进程添加组ID中的一个，以及进程没有超级用户优先数，那么设置-组-ID位自动被关闭。这就防止了用户创建一个设置-组-ID文件，而该文件是由并非该用户所属的组拥有的。

> 4.3+BSD和其他伯克利导出的系统增加了另外的安全性特征以试图防止保护位的错误使用。如果一个没有超级用户优先权的进程写一个文件，则设置 -用户-ID位和设置-组-ID位自动被清除。如果一个不怀好意的用户找到一个他可以写的设置-组-ID和设置-用户-ID文件，即使他可以修改此文件，但失去了对该文件的特别优先权。

4.10 粘住位

S_ISVTX位有一段有趣的历史。在UNIX的早期版本中，有一位被称为粘住位（sticky bit)。如果一个可执行程序文件的这一位被设置了，那么在该程序第一次执行并结束时，该程序正文的一个文本被保存在交换区。(程序的正文部分是机器指令部分。)这使得下次执行该程序时能较快地将其装入内存区。其原因是：在交换区，该文件是被连续存放的，而在一般的 UNIX文件系统中，文件的各数据块很可能是随机存放的。对于常用的应用程序，例如文本编辑程序和编译程序的各部分常常设置它们所在文件的粘住位。自然，对交换区中可以同时存放的设置了粘住位的文件数有一定限制，以免过多占用交换区空间，但无论如何这是一个有用的技术。因为在系统再次自举前，文件的正文部分总是在交换区中，所以使用了名字“ 粘住 ”。后来的UNIX版本称之为保存-正文位（saved-text bit)，因此也就有了常数 S_ISVTX。现今较新的UNIX系统大多数都具有虚存系统以及快速文件系统，所以不再需要使用这种技术。

SVR4和4.3+BSD中粘住位的主要针对目录。如果对一个目录设置了粘住位，则只有对该目录具有写许可权的用户并且满足下列条件之一，才能删除或更名该目录下的文件：

• 拥有此文件。

• 拥有此目录。

• 是超级用户。

目录/tmp和/var/spool/uucppublic是设置粘住位的候选者 —— 这两个目录是任何用户都可在其中

创建文件的目录。这两个目录对任一用户 (用户、组和其他)的许可权通常都是读、写和执行。但是用户不应能删除或更名属于其他人的文件，为此在这两个目录的文件方式中都设置了粘住位。

> POSIX.1没有定义粘住位，但SVR4和4.3+BSD则支持这种特征。

4.11 chown, fchown和lchown函数

chown函数可用于更改文件的用户ID和组ID。

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

int chown(const char *pathname, uid_t owner, gid_t group);

int fchown(int filedes, uid_t owner, gid_t group);

int lchown(const char *pathname, uid_t owner, gid_t group);
```

三个函数返回：若成功则为0，若出错则为-1

除了所引用的文件是符号连接以外，这三个函数的操作相类似。在符号连接情况下， lchown更改符号连接本身的所有者，而不是该符号连接所指向的文件。

> fchown函数并不在POSIX 1003.1-1990标准中，但很可能被加到 1003.1a中。SVR4和4.3+BSD都支持fchown。
>
> 只有SVR4支持lchown函数。在非 SVR4系统中(POSIX.1和4.3+BSD)，若chown的参数*pathname*是符号连接，则改变该符号连接的所有权，而不改变它所指向的文件的所有权。为了更改该符号连接所指向的文件的所有权，我们应指定该实际文件本身的*pathname*，而不是指向该文件的连接文件的*pathname*。
>
> SVR4，4.3+BSD和XPG3允许将参数*owner*或*group*指定为－1,以表示不改变相应的ID。这不是POSIX.1的一部分。

基于伯克利的系统一直规定只有超级用户才能更改一个文件的所有者。这样做的原因是防止用户改变其文件的所有者从而摆脱磁盘空间限额对他们的限制。系统 V则允许任一用户更改他们所拥有的文件的所有者。

> 按照_POSIX_CHOWN_RESTRICTED的值，POSIX.1在这两种形式的操作中选用一种。FIPS 151-1要求_POSIX_CHOWN_RESTRICTED。
>
> 对于SVR4，此功能是个配置选择项，而4.3+BSD则总对chown施加了限制。

回忆表2-5，该常数可选地定义在头文件 <unistd.h>中，而且总是可以用 pathconf或fpathconf函数查询。此选择项还与所引用的文件有关 —— 可在每个文件系统基础上，使该选择项起作用或不起作用。在下文中，如提及“若 _POSIX_CHOWN_RESTRICTED起作用”，则表示这适用于我们正在谈及的文件，而不管该实际常数是否在头文件中定义 (例如，4.3+BSD总有这种限制，而并不在头文件中定义此常数)。

若_POSIX_CHOWN_RESTRICTED对指定的文件起作用，则

(1) 只有超级用户进程能更改该文件的用户ID。

(2) 若满足下列条件，一个非超级用户进程可以更改该文件的组ID：

(a) 进程拥有此文件（其有效用户ID等于该文件的用户ID）。

(b) 参数*owner*等于文件的用户ID，参数*group*等于进程的有效组ID或进程的添加组ID之一。

这意味着，当_POSIX_CHOWN_RESTRICTED有效时，不能更改其他用户的文件的用户ID。你可以更改你所拥用的文件的组ID，但只能改到你所属于的组。

如果这些函数由非超级用户进程调用，则在成功返回时，该文件的设置-用户-ID位和设置-组-ID位都被清除。

4.12 文件长度

stat结构的成员st_size包含了以字节为单位的该文件的长度。此字段只对普通文件、目录文件和符号连接有意义。

> SVR4对管道也定义了文件长度，它表示可从该管道中读到的字节数，我们将在14.2中讨论管道。

对于普通文件，其文件长度可以是0，在读这种文件时，将得到文件结束指示。

对于目录，文件长度通常是一个数，例如16或512的整倍数，我们将在4.21节中说明读目录操作。

对于符号连接，文件长度是在文件名中的实际字节数。例如，

```
lrwxrwxrwx 1 root            7 Sep 25 07:14 lib -> usr/lib
```

其中，文件长度7就是路径名usr/lib的长度(注意，因为符号连接文件长度总是由st_size指示，所以符号连接并不包含通常C语言用作名字结尾的null字符)。

SVR4和4.3+BSD也提供字段st_blksize和st_blocks。第一个是对文件I/O较好的块长度，第二个是所分配的实际512字节块块数。回忆一下3.9节，其中提到了当我们将st_blksize用于读操作时，读一个文件所需的最少时间量。为了效率的缘故，标准I/O库(我们将在第5章中说明)也试图一次读、写st_blksize字节。

> 要知道，不同的UNIX版本其st-blocks所用的单位可能不是512字节块。使用此值并不是可移植的。

文件中的空洞

在3.6节中，我们提及普通文件可以包含空洞。在程序3-2中例示了这一点。空洞是由超过文件结尾端的位移量设置，并写了某些数据后造成的。作为一个例子，考虑下列情况：

```
$ ls -1 core
-rw-r--r--  1 stevens   8483248 Nov 18 12:18 core
$ du -s core
272     core
```

文件core的长度超过8M字节，而du命令则报告该文件所使用的磁盘空间总量是272个512字节块(139 264字节)(在很多伯克利类的系统上，du命令报告1024字节块块数，SVR4则报告512字

节块块数)。很明显，此文件有很多空洞。

正如我们在3.6节中提及的，read函数对于没有写过的字节位置读到的数据字节是 0。如果执行：

```
$ wc -c core
 8483248 core
```

由此可见，正常的I/O操作读至整个文件长度（带 -c选择项的wc(1)命令计算文件中的字符（字节）数)。

如果使用公用程序，例如cat(1),复制这种文件，那么所有这些空洞都被写成实际数据字节0。

```
$ cat core > core.copy
$ ls -l core*
-rw-r--r--  1 stevens   8483248 Nov 18 12:18 core
-rw-rw-r--  1 stevens   8483248 Nov 18 12:27 core.copy
$ du -s core*
272     core
16592   core.copy
```

从中可见，新文件所用的字节数是8 495 104 (512×16 592)。此长度与ls命令报告的长度之间的差别是由于文件系统使用了若干块以保持指向实际数据块的各指针。

有兴趣的读者应当参阅Bach〔1986〕的4.2节和Leffler〔1989〕的7.2节，以更详细地了解文件的物理安排。

4.13 文件截短

有时我们需要在文件尾端处截去一些数据以缩短文件。将一个文件的长度截短为 0是一个特例，用O_TRUNC标志可以做到这一点。为了截短文件可以调用函数truncate和ftruncate。

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

int truncate(const char *pathname, off_t length);

int ftruncate(int filedes, off_t length);
```

两个函数返回；若成功则为0，若出错则为 - 1

这两个函数将由路径名 *pathname*或打开文件描述符 *filedes*指定的一个现存文件的长度截短为 *length*。如果该文件以前的长度大于 *length*，则超过 *length*以外的数据就不再能存取。如果以前的长度短于*length*，则其后果与系统有关。如果某个实现的处理是扩展该文件，则在以前的文件尾端和新的文件尾端之间的数据将读作0 (也就是在文件中创建了一个空洞)。

SVR4和4.3+BSD提供了这两个函数。它们不是POSIX.1或XPG3的组成部分。

SVR4截短或扩展一个文件。4.3+BSD只用这三个函数截短一个文件——不能用它们扩展一个文件。

UNIX从来就没有截短文件的一种标准方法。完全兼容的应用程序必须对文件制作一个副本，在制作它时只复制所希望的数据字节。

SVR4的fcntl中有一个POSIX.1没有规定的命令F_FREESP,它允许释放一个文件中的任何一部分，而不只是文件尾端处的一部分。

程序12-5使用了ftruncate函数，以便在获得对该文件的锁后，使一个文件变空。

4.14 文件系统

为了说明文件连接的概念，先要对文件系统的结构有基本了解。同时，了解 i节点和指向一个i节点的目录项之间的区别也是很有益的。

目前有多种UNIX文件系统的实现。例如，SVR4支持两种不同类型的磁盘文件系统：传统的UNIX系统V文件系统（S5），以及统一文件系统（UFS）。在表2-6中，我们已看到了这两种文件系统的一个区别。UFS是以伯克利快速文件系统为基础的。SVR4也支持另外一些非磁盘文件系统，两个分布式文件系统，以及一个自举文件系统，这些文件系统都不影响下面的讨论。本节讨论传统的UNIX系统V文件系统。这种类型的文件系统可以回溯到V7。

我们可以把一个磁盘分成一个或多个分区。见图4-1，每个分区可以包含一个文件系统。

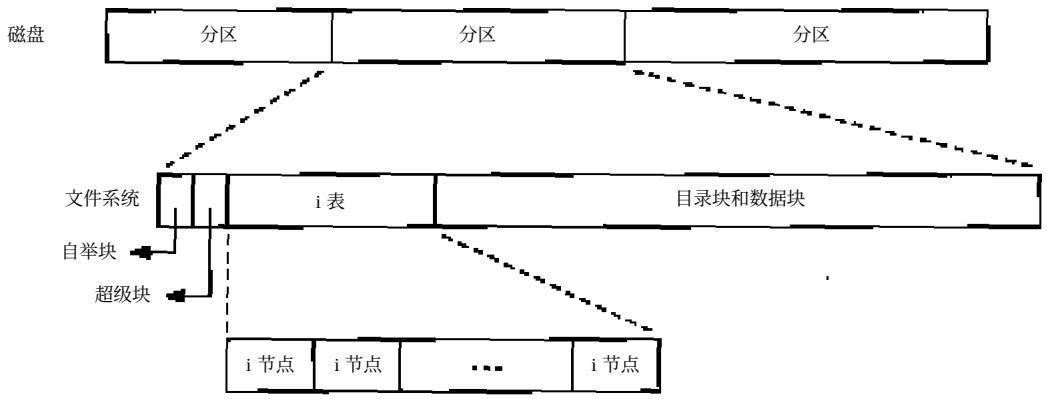

图4-1 磁盘、分区和文件系统

i节点是固定长度的记录项，它包含有关文件的信息。

> 在V7中，i节点占用64字节，在4.3+BSD中，i节点占用128字节。在SVR4中，在磁盘上一个i节点的长度与文件系统的类型有关：S5 i节点占用64字节，而UFS i节点占用128字节。

如果在忽略自举块和超级块情况下更仔细地观察文件系统，则可以得到图4-2中所示的情况。

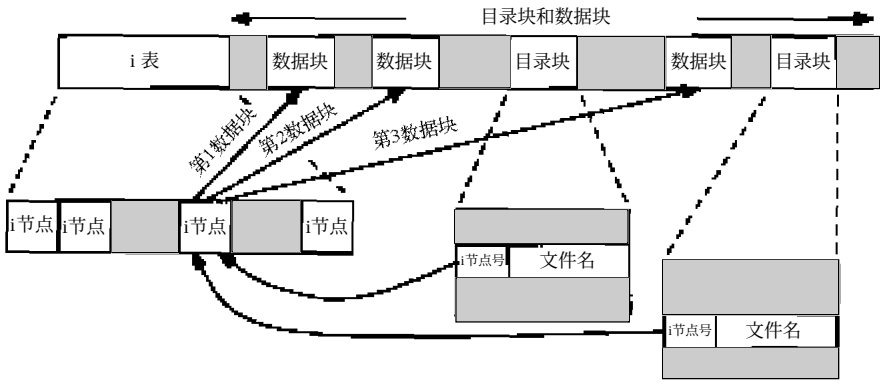

图4-2 较详细的文件系统

注意图4-2中的下列各点：

• 在图中有两个目录项指向同一i节点。每个i节点中都有一个连接计数，其值是指向该i节点的目录项数。只有当连接计数减少为0时，才可删除该文件(也就是可以释放该文件占用的数据块)。这就是为什么“解除对一个文件的连接”操作并不总是意味着“释放该文件占用的磁盘块”的原因。这也就是为什么删除一个目录项的函数被称之为unlink而不是delete的原因。在stat结构中，连接计数包含在st_nlink成员中，其基本系统数据类型是nlink_t。这种连接类型称之为硬连接。回忆表2-7，其中，POSIX.1常数LINK_MAX指定了一个文件连接数的最大值。

• 另外一种连接类型称之为符号连接（symbolic link）。对于这种连接，该文件的实际内容(在数据块中)包含了该符号连接所指向的文件的名字。在下例中：

```
lrwxrwxrwx 1 root        7 Sep 25 07:14 lib -> urs/lib
```

该目录项中的文件名是lib，而在该文件中包含了7个数据字节usr/lib。该i节点中的文件类型是S_IFLNK，于是系统知道这是一个符号连接。

• i节点包含了所有与文件有关的信息：文件类型、文件存取许可权位、文件长度和指向该文件所占用的数据块的指针等等。stat结构中的大多数信息都取自i节点。只有两项数据存放在目录项中：文件名和i节点编号数。i节点编号数的数据类型是ino_t。

• 因为目录项中的i节点编号数指向同一文件系统中的i节点，所以不能使一个目录项指向另一个文件系统的i节点。这就是为什么ln(1)命令（构造一个指向一个现存文件的新目录项），不能跨越文件系统的原因。我们将在下一节说明link函数。

• 当在不更改文件系统的情况下为一个文件更名时，该文件的实际内容并未移动，只需构造一个指向现存i节点的新目录项，并删除老的目录项。例如，为将文件/usr/lib/foo更名为/usr/foo，如果目录/usr/lib和/usr在同一文件系统上，则文件foo的内容无需移动。这就是mv(1)命令的通常操作方式。

我们说明了普通文件的连接计数的概念，但是对于目录文件的连接计数字段又如何呢？假定我们在工作目录中构造了一个新目录：

```
$ mkdir testdir
```

图4-3显示了其结果。注意，该图显式地显示了.和..目录项。

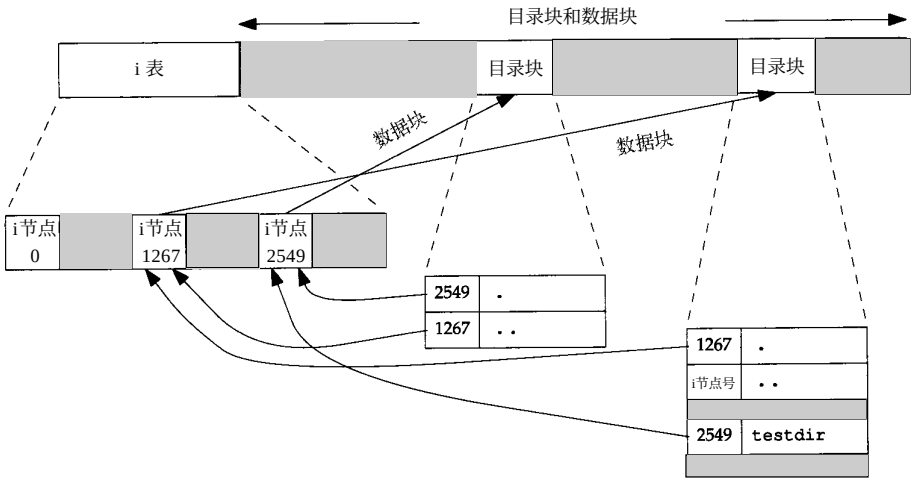

图4-3 创建了目录testdir后的文件系统实例

编号为2549的i节点，其类型字段表示它是一个目录，而连接计数为2。任何一个叶目录(不包含任何其他目录，也就是子目录的目录)其连接计数总是2，数值2来自于命名该目录（testdir）的目录项以及在该目录中的.项。编号为1267的i节点，其类型字段表示它是一个目录，而其连接计数则大于或等于3。它大于或等于3的原因是，至少有由三个目录项指向它：一个是命名它的目录项(在图4-3中没有表示出来)，第二个是在该目录中的.项，第三个是在其子目录testdir中的.. 项。注意，在工作目录中的每个子目录都使该工作目录的连接计数增1。

正如前面所述，这是UNIX文件系统的经典格式，在Bach〔1986〕一书的第4章中对此作了说明。关于伯克利快速文件系统对此所作的更改请参阅Leffler 等〔1989〕中的第7章。

4.15 link,unlink,remove和rename函数

如上节所述，任何一个文件可以有多个目录项指向其i节点。创建一个向现存文件连接的方法是使用link函数。

```
#include <unistd.h>

int link(const char *existingpath, const char *newpath);
```

返回：若成功则为0，若出错则为-1

此函数创建一个新目录项*newpath*，它引用现存文件*existingpath*。如若*newpath*已经存在，则返回出错。

创建新目录项以及增加连接计数应当是个原子操作（请回忆在3.11节中对原子操作的讨论）。

大多数实现，例如SVR4和4.3＋BSD要求这两个路径名在同一个文件系统中。

> POSIX.1允许支持跨越文件系统的连接的实现。

只有超级用户进程可以创建指向一个目录的新连接。其理由是这样做可能在文件系统中形成循环，大多数处理文件系统的公用程序都不能处理这种情况（4.16节将说明一个由符号连接引入的循环的例子）。

为了删除一个现存的目录项，可以调用unlink函数。

```
#include <unistd.h>

int unlink(const char *pathname);
```

返回：若成功则为0，若出错则为-1

此函数删除目录项，并将由*pathname*所引用的文件的连接计数减1。如果该文件还有其他连接，则仍可通过其他连接存取该文件的数据。如果出错，则不对该文件作任何更改。

我们在前面已经提及，为了解除对文件的连接，必须对包含该目录项的目录具有写和执行许可权。正如4.10节所述，如果对该目录设置了粘住位，则对该目录必须具有写许可权，并且具备下面三个条件之一：

- 拥有该文件。
- 拥有该目录。
- 具有超级用户优先权。

只有当连接计数达到0时，该文件的内容才可被删除。另一个条件也阻止删除文件的内容——只要有进程打开了该文件，其内容也不能删除。关闭一个文件时，内核首先检查使该文件打开的进程计数。如果该计数达到0，然后内核检查其连接计数，如果这也是0，那么就删除该文件的内容。

实例

程序4-5打开一个文件，然后unlink它。执行该程序的进程然后睡眠15秒钟，接着就终止。

程序4-5 打开一个文件，然后unlink它

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

运行该程序，其结果是：

```
$ ls -1 tempfile            查看文件大小
-rw-r--r--  1 stevens   9240990 Jul 31 13:42 tempfile
$ df/home                   检查空闲空间
Filesystem   kbytes    used   avail capacity  Mounted on
/dev/sd0h    282908  181979   72638    71%    /home
$ a.out &                   在后台运行程序4-5
1364                        shell打印其进程ID
$ file unlinked             该文件是未连接的
ls -1 tempfile              观察文件是否仍然存在
tempfile not found          目录项已删除
$ df/home                   检查空闲空间有无变化
Filesystem   kbytes    used  avail capacity  Mounted on
/dev/sd0h    282908  181979  72638    71%    /home
$ done                      程序执行结束，关闭所有打开文件
df/home                     磁盘空间有效
Filesystem   kbytes    used  avail capacity  Mounted on
/dev/sd0h    282908  172939  81678    68%    /home
                            9.2M字节磁盘空间有效
```

unlink的这种特性经常被程序用来确保即使是在程序崩溃时，它所创建的临时文件也不会遗留下来。进程用open或creat创建一个文件，然后立即调用unlink。因为该文件仍旧是打开的，所以不会将其内容删除。只有当进程关闭该文件或终止时（在这种情况下，内核关闭该进程所打开的全部文件），该文件的内容才被删除。

如果*pathname*是符号连接，那么unlink涉及的是符号连接而不是由该连接所引用的文件。

超级用户可以调用带参数*pathname*的unlink指定一个目录，但是通常不使用这种方式，而使用函数rmdir。我们将在4.20节中说明rmdir函数。

我们也可以用 remove函数解除对一个文件或目录的连接。对于文件， remove的功能与unlink相同。对于目录， remove的功能与rmdir相同。

```
#include <stdio.h>

int remove(const char *pathname);
```

返回：若成功则为0，若出错则为-1

ANSI C指定remove函数删除一个文件，这更改了 UNIX历来使用的名字unlink,其原因是实现C标准的大多数非UNIX系统并不支持文件连接。

文件或目录用rename函数更名。

```
#include <stdio.h>

int rename(const char *oldname, const char *newname);
```

返回：若成功则为0，若出错则为-1

ANSI C 对文件定义了此函数（C标准不处理目录）。POSIX.1扩展此定义包含了目录。

根据*oldname*是指文件还是目录，有两种情况要加以说明。我们也应说明如果 *newname*已经存在将会发生什么。

(1) 如果*oldname*说明一个文件而不是目录，那么为该文件更名。在这种情况下，如果*newname*已存在，则它不能引用一个目录。如果*newname*已存在，而且不是一个目录，则先将该目录项删除然后将*oldname*更名为*newname*。对包含*oldname*的目录以及包含*newname*的目录，调用进程必须具有写许可权，因为将更改这两个目录。

(2) 如若*oldname*说明一个目录，那么为该目录更名。如果*newname*已存在，则它必须引用一个目录，而且该目录应当是空目录（空目录指的是该目录中只有 . 和.. 项）。如果*newname*存在（而且是一个空目录），则先将其删除，然后将*oldname*更名为*newname*。另外，当为一个目录更名时， *newname*不能包含 *oldname*作为其路径前缀。例如，不能将 /usr/foo更名为/usr/foo/testdir，因为老名字（/usr/foo）是新名字的路径前缀，因而不能将其删除。

(3) 作为一个特例，如果*oldname*和*newname*引用同一文件，则函数不做任何更改而成功返回。

如若*newname*已经存在，则调用进程需要对其有写许可权（如同删除情况一样）。另外，调用进程将删除*oldname*目录项，并可能要创建*newname*目录项，所以它需要对包含*oldname*及包含*newname*的目录具有写和执行许可权。

4.16 符号连接

符号连接是对一个文件的间接指针，它与上一节所述的硬连接有所不同，硬连接直接指向

文件的i节点。引进符号连接的原因是为了避免硬连接的一些限制： (a)硬连接通常要求连接和文件位于同一文件系统中， (b)只有超级用户才能创建到目录的硬连接。对符号连接以及它指向什么没有文件系统限制，任何用户都可创建指向目录的符号连接。符号连接一般用于将一个文件或整个目录结构移到系统中其他某个位置。

> 符号连接由4.2BSD引进，后来又得到SVR4的支持。在SVR4中，传统的系统V文件系统(S5)和统一文件系统(UFS)都支持符号连接。
>
> POSIX 1003.1-1990标准并不包括符号连接，但很可能会加到1003.1a中。

当使用以名字引用一个文件的函数时，应当了解该函数是否处理符号连接功能。也就是是否跟随符号连接到达它所连接的文件。如若该函数处理符号连接功能，则该函数的路径名参数引用由符号连接指向的文件。否则，一个路径名参数引用连接本身，而不是由该连接指向的文件。表4-7列出了本章中所说明的各个函数是否处理符号连接功能。因为 rmdir并不是针对符号连接进行定义的（若path是符号连接则返回出错），所以在表4-7中没有列出这一函数。因为对符号连接的处理是由返回文件描述符的函数进行的（通常是 open），所以以文件描述符作为参数的函数（如fstat, fchmod等）也未列出。chown是否跟随符号连接取决于实现——各种有关细节见4.11节。

表4-7 各个函数对符号连接的处理

函 数	不跟随符号连接	跟随符号连接
access		•
chdir		•
chmod		•
chown	•	•
creat		•
exec		•
lchown	•	
link		•
lstat	•	
mkdir		•
mkfifo		•
mknod		•
open		•
opendir		•
pathconf		•
readlink	•	
remove	•	
rename	•	
stat		•
truncate		•
unlink	•	

实例

使用符号连接可能在文件系统中引入循环。大多数查找路径名的函数在这种情况发生时都

返回值为ELOOP的errno。考虑下列命令序列：

```
$ mkdir foo                     创建一个新目录
$ touch foo/a                   创建0长文件
$ ln -s ../foo foo/testdir      创建一符号连接
$ ls -1 foo
total 1
-rw-rw-r-- 1 stevens       0 Dec  6 06:06 a
lrwxrwxrwx 1 stevens       6 Dec  6 06:06 testdir -> ../foo
```

这创建了一个目录foo，它包含了一个名为a的文件以及一个指向foo的符号连接。在图4-4中显示了这种结果，图中以圆表示目录，以正方形表示一个文件。如果我们写一段简单的程序，使用标准函数ftw(3) 以降序遍历文件结构，打印每个遇到的路径名，则其输出是：

```
foo
foo/a
foo/testdir
foo/testdir/a
foo/testdir/testdir
foo/testdir/testdir/a
foo/testdir/testdir/testdir
foo/testdir/testdir/testdir/a
   （更多行）
ftw returned -1:Too many levels of symbolic links
```

4.21节提供了我们自己的ftw函数版本，它用lstat代替stat以阻止它跟随符号连接。

这样一个连接很容易被删除——因为unlink并不跟随符号连接，所以可以unlink文件foo/testdir。但是如果创建了一个构成这种循环的硬连接，那么就很难删除它[⊖]。这就是为什么link函数不允许构造指向目录的硬连接的原因。（除非进程具有超级用户优先权。）

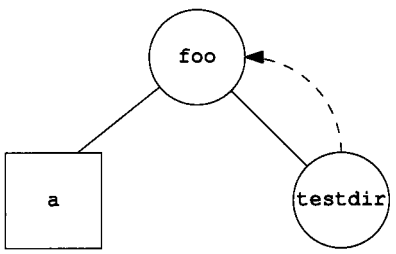

图4-4 创建一个循环的符号连接testdir

用open打开文件时，如果传递给open函数的路径名指定了一个符号连接，那么open跟随此连接到指定的文件。若此符号连接所指向的文件并不存在，则open返回出错，表示它不能打开该文件。这可能会使不熟悉符号连接的用户感到迷惑，例如：

```
$ ln -s /no/such/file myfile        创建一符号连接
$ ls myfile
myfile                              ls查到该文件
$ cat myfile                        试图察看该文件
cat: myfile: No such file or directory
$ ls -1 myfile                      试-l选择项
lrwxrwxrwx  1 stevens    13 Dec  6 07:27 myfile -> /no/such/file
```

文件myfile存在，但cat却称没有这一文件。其原因是myfile是个符号连接，由该符号连接所指

⊖ 在编写本节时，作者在自己的系统上作为一个实验做了这一点。文件系统变得错误百出，正常的 fsck(1)公共程序不能解决问题。为了修复此文件系统，不得不使用了并不推荐使用的工具 clri(8)和dcheck(8)。

向的文件并不存在。ls命令的-l选择项给我们两个提示：第一个字符是l，它表示这是一个符号连接，而－> 表示这是一个符号连接。ls命令还有另一个选择项-F，它在是符号连接的文件名后加一个@符号，在未使用-l选择项时，这可以帮助识别出符号连接。

4.17 symlink和readlink函数

symlink函数创建一个符号连接。

```
#include <unistd.h>

int symlink(const char *actualpath, const char *sympath);
```

返回：若成功则为0，若出错则为-1

该函数创建了一个指向 *actualpath*的新目录项 *sympath*，在创建此符号连接时，并不要求 *actualpath*已经存在（在上一节结束部分的例子中我们已经看到了这一点）。并且， *actualpath*和*sympath*并不需要位于同一文件系统中。

因为open函数跟随符号连接，所以需要有一种方法打开该连接本身，并读该连接中的名字。readlink函数提供了这种功能。

```
#include <unistd.h>

int readlink(const char *pathname, char *buf, int bufsize);
```

返回：若成功则为读的字节数，若出错则为-1

此函数组合了open, read和close的所有操作。

如果此函数成功，则它返回读入 *buf*的字节数。在*buf*中返回的符号连接的内容不以null字符终止。

4.18 文件的时间

对每个文件保持有三个时间字段，它们的意义示于表4-8中。

表4-8 与每个文件相关的三个时间值

字 段	说 明	例 子	ls(1)选择项
st_atime	文件数据的最后存取时间	read	-u
st_mtime	文件数据的最后修改时间	write	缺省
st_ctime	i节点状态的最后更改时间	chmod, chown	-c

注意修改时间(st_mtime)和更改状态时间(st_ctime)之间的区别。修改时间是文件内容最后一次被修改的时间。更改状态时间是该文件的i节点最后一次被修改的时间。在本章中我们已说明了很多操作，它们影响到i节点，但并没有更改文件的实际内容：文件的存取许可权、用户ID、连接数等等。因为i节点中的所有信息都是与文件的实际内容分开存放的，所以，除了文件数据修改时间以外，还需要更改状态时间。

注意，系统并不保存对一个i节点的最后一次存取时间，所以access和stat函数并不更改这

三个时间中的任一个。

系统管理员常常使用存取时间来删除在一定的时间范围内没有存取过的文件。典型的例子是删除在过去一周内没有存取过的名为 a.out或core的文件。find(1)命令常被用来进行这种操作。

修改时间和更改状态时间可被用来归档其内容已经被修改或其 i节点已经被更改的那些文件。

ls命令按这三个时间值中的一个排序进行显示。按系统默认 (用-l或-t选择项调用时)，它按文件的修改时间的先后排序显示。 -u选择项使其用存取时间排序， -c选择项则使其用更改状态时间排序。

表4-9列出了我们已说明过的各种函数对这三个时间的作用。回忆 4.14节中所述，目录是包含目录项（文件名和相关的 i节点编号）的文件，增加、删除或修改目录项会影响到与其所在目录相关的三个时间。这就是在表 4-9中包含两列的原因，其中一列是与该文件 (或目录)相关的三个时间，另一列是与所引用的文件（或目录）的父目录相关的三个时间。例如，创建一个新文件影响到包含此新文件的目录，也影响该新文件的 i节点。但是，读或写一个文件只影响该文件的i节点，而对父目录则无影响（mkdir和rmdir函数将在4.20节中说明。utime函数将在下一节中说明。6个exec函数将在4.20节中讨论。第14章将说明mkfifo和pipe函数）。

表4-9 各种函数对存取、修改和更改状态时间的作用

函 数	引用文件（或目录）			引用文件（或目录）的父目录			备 注
	a	m	c	a	m	c	
chmod,fchmod			•				
chown,fchown			•				
creat	•	•	•		•	•	O_CREAT新文件
creat		•	•				O_TRUNC现存文件
exec	•						
lchown			•				
link			•		•	•	
mkdir	•	•	•		•	•	
mkfifo	•	•	•		•	•	
open	•	•	•		•	•	O_CREAT新文件
open		•	•				O_TRUNC现存文件
pipe	•	•	•				
read	•						
remove			•		•	•	删除文件 = unlink
remove					•	•	删除目录 = rmdir
rename			•		•	•	对于两个参数
rmdir					•	•	
truncate,ftruncate		•	•				
unlink			•		•	•	
utime	•	•	•				
write		•	•				

4.19 utime函数

一个文件的存取和修改时间可以用utime函数更改。

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

int utime(const char *pathname, const struct utimbuf *times);
```

返回：若成功则为0，若出错则为-1

此函数所使用的结构是：

```
struct utimbuf {
time_t actime;    /*access time*/
time_t modtime;   /*modification time*/
}
```

此结构中的两个时间值是日历时间。如1.10节中所述，这是自1970年1月1日，00:00:00以来国际标准时间所经过的秒数。

此函数的操作以及执行它所要求的优先权取决于*times*参数是否是NULL。

(1) 如果*times*是一个空指针，则存取时间和修改时间两者都设置为当前时间。为了执行此操作必须满足下列两条件之一：(a)进程的有效用户ID必须等于该文件的所有者ID，(b)进程对该文件必须具有写许可权。

(2) 如果*times*是非空指针，则存取时间和修改时间被设置为*times*所指向的结构中的值。此时，进程的有效用户ID必须等于该文件的所有者ID，或者进程必须是一个超级用户进程。对文件只具有写许可权是不够的。

注意，我们不能对更改状态时间st_ctime指定一个值，当调用utime函数时，此字段被自动更新。

在某些UNIX版本中，touch(1)命令使用此函数。另外，标准归档程序tar(1)和cpio(1)可选地调用utime,以便将一个文件的时间值设置为将它归档时的值。

实例

程序4-6使用带O_TRUNC选择项的open函数将文件长度截短为0，但并不更改其存取时间及修改时间。为了做到这一点，首先用stat函数得到这些时间，然后截短文件，最后再用utime函数重置这两个时间。

程序4-6 utime函数实例

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

int
main(int argc, char *argv[])
{
    int             i;
    struct stat     statbuf;
    struct utimbuf  timebuf;
```

```
    for (i = 1; i < argc; i++) {
        if (stat(argv[i], &statbuf) < 0) {  /* fetch current times */
            err_ret("%s: stat error", argv[i]);
            continue;
        }
        if (open(argv[i], O_RDWR | O_TRUNC) < 0) {  /* truncate */
            err_ret("%s: open error", argv[i]);
            continue;
        }
        timebuf.actime  = statbuf.st_atime;
        timebuf.modtime = statbuf.st_mtime;
        if (utime(argv[i], &timebuf) < 0) {     /* reset times */
            err_ret("%s: utime error", argv[i]);
            continue;
        }
    }
    exit(0);
}
```

以下列方式运行程序4-6：

```
$ ls -l changemod times          察看长度和最后修改时间
-rwxrwxr-x  1 stevens   24576 Dec  4 16:13 changemod
-rwxrwxr-x  1 stevens   24576 Dec  6 09:24 times
$ ls -lu changemod times         察看最后存取时间
-rwxrwxr-x  1 stevens   24576 Feb  1 12:44 changemod
-rwxrwxr-x  1 stevens   24576 Feb  1 12:44 times
$ date                           打印当天日期
Sun Feb  3 18:22:33 MST 1991
$ a.out changemod times          运行程序4-6
$ ls -l changemod times          检查结果
-rwxrwxr-x  1 stevens       0 Dec  4 16:13 changemod
-rwxrwxr-x  1 stevens       0 Dec  6 09:24 times
$ ls -lu changemod times         检查最后存取时间
-rwxrwxr-x  1 stevens       0 Feb  1 12:44 changemod
-rwxrwxr-x  1 stevens       0 Feb  1 12:44 times
$ ls -lc changemod times         更改状态时间
-rwxrwxr-x  1 stevens       0 Feb  3 18:23 changemod
-rwxrwxr-x  1 stevens       0 Feb  3 18:23 times
```

正如我们所预见的一样，最后修改时间和最后存取时间未变。但是，更改状态时间则更改为程序运行时的时间（这两个文件的最后存取时间相同的原因是，这是它们的目录用 tar命令归档时的时间）。

4.20 mkdir和rmdir函数

用mkdir函数创建目录，用rmdir函数删除目录。

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

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

返回：若成功则为0，若出错则为-1

此函数创建一个新的空目录。. 和.. 目录项是自动创建的。所指定的文件存取许可权 *mode*由进程的文件方式创建屏蔽字修改。

常见的错误是指定与文件相同的*mode* (只指定读、写许可权)。但是，对于目录通常至少要设置1个执行许可权位，以允许存取该目录中的文件名（见习题4.18)。

按照4.6节中讨论的规则，设置新目录的用户ID和组ID。

> SVR4也使新目录继承父目录的设置-组-ID位。这就使得在新目录中创建的文件将继承该目录的组ID。
>
> 4.3+BSD并不要求继承此设置-组-ID位，因为不论设置-组-ID位如何，新创建的文件和目录总是继承父目录的组ID。
>
> 早期的UNIX版本并没有mkdir函数，它是由4.2BSD和SVR3引进的。在早期版本中，进程要调用mknod函数以创建一个新目录。但是只有超级用户进程才能使用mknod函数。为了避免这一点，创建目录的命令mkdir(1)必须由根拥有，而且打开了其设置-用户-ID位。进程为了创建一个目录，必须用 system(3)函数调用mkdir命令(1)。

用rmdir函数可以删除一个空目录。

```
#include <unistd.h>

int rmdir(const char *pathname);
```

返回：若成功则为0，若出错则为-1

如果此调用使目录的连接计数成为0，并且也没有其他进程打开此目录，则释放由此目录占用的空间。如果在连接计数达到0时，有一个或几个进程打开了此目录，则在此函数返回前删除最后一个连接及. 和.. 项。另外，在此目录中不能再创建新文件。但是在最后一个进程关闭它之前并不释放此目录（即使某些进程打开该目录，它们在此目录下，也不能执行其他操作，因为为使rmdir函数成功执行，该目录必须是空的)。

4.21 读目录

对某个目录具有存取许可权的任一用户都可读该目录，但是只有内核才能写目录（防止文件系统发生混乱)。回忆4.5节，一个目录的写许可权位和执行许可权位决定了在该目录中能否创建新文件以及删除文件，它们并不表示能否写目录本身。

目录的实际格式依赖于UNIX的具体实现。早期的系统，例如V7，有一个比较简单的结构：每个目录项是16个字节，其中14个字节是文件名，2个字节是i节点编号数。而对于4.2BSD而言，由于它允许相当长的文件名，所以每个目录项的长度是可变的。这就意味着读目录的程序与系统相关。为了简化这种情况， UNIX现在包含了一套与读目录有关的例程，它们是 POSIX.1的一部分。

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

DIR *opendir(const char *pathname);
```

返回：若成功则为指针，若出错则为NULL

```
struct dirent *readdir(DIR *dp);
```

返回：若成功则为指针，若在目录尾或出错则为NULL

```
void rewinddir(DIR *dp);

int closedir(DIR *dp);
```

返回：若成功则为0，若出错则为-1

回忆一下，在程序1-1中（ls命令的基本实现部分）使用了这些函数。

定义在头文件<dirent.h>中的dirent结构与实现有关。SVR4和4.3+BSD定义此结构至少包含下列两个成员：

```
struct dirent {
  ino_t d_ino;              /*i-node number*/
  char  d_name[NAME_MAX+1] ;   /*null-terminated filename*/
}
```

> POSIX.1并没有定义d_ino,因为这是一个实现特征。POSIX.1在此结构中只定义d_name项。

注意，SVR4没有将NAME_MAX定义为一个常数——其值依赖于该目录所在的文件系统，并且通常可用fpathconf函数取得。在BSD类文件系统中，NAME_MAX的常用值是255（见表2-7）。但是，因为文件名是以null字符结束的，所以在头文件中如何定义数组d_name并无多大关系。

DIR结构是一个内部结构，它由这四个函数用来保存正被读的目录的有关信息。其作用类似于FILE结构。FILE结构由标准I/O库维护（我们将在第5章中对它进行说明）。

由opendir返回的指向DIR结构的指针由另外三个函数使用。opendir执行初始化操作，使第一个readdir读目录中的第一个目录项。目录中各目录项的顺序与实现有关。它们通常并不按字母顺序排列。

实例

我们将使用这些目录例程编写一个遍历文件层次结构的程序，其目的是得到如表4-2中所示的各种类型的文件数。程序4-7只有一个参数，它说明起点路径名，从该点开始递归降序遍历文件层次结构。系统V提供了一个实际遍历此层次结构的函数ftw(3)，对于每一个文件它都调用一个用户定义函数。此函数的问题是：对于每一个文件，它都调用stat函数，这就使程序跟随符号连接。例如，如果从root开始，并且有一个名为/lib的符号连接，它指向/usr/lib,则所有在目录/usr/lib中的文件都两次计数。为了纠正这一点，SVR4提供了另一个函数nftw(3)，它具有一个停止跟随符号连接的选择项。尽管可以使用nftw，但是为了说明目录例程的使用方法，我们还是编写了一个简单的文件遍历程序。

程序4-7 递归降序遍历目录层次结构，并按文件类型计数

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

```
#include    <limits.h>
#include    "ourhdr.h"
typedef int Myfunc(const char *, const struct stat *, int);
                /* function type that's called for each filename */

static Myfunc   myfunc;
static int      myftw(char *, Myfunc *);
static int      dopath(Myfunc *);

static long nreg, ndir, nblk, nchr, nfifo, nslink, nsock, ntot;

int
main(int argc, char *argv[])
{
    int     ret;

    if (argc != 2)
        err_quit("usage:  ftw  <starting-pathname>");

    ret = myftw(argv[1], myfunc);       /* does it all */

    if ( (ntot = nreg + ndir + nblk + nchr + nfifo + nslink + nsock) == 0)
        ntot = 1;       /* avoid divide by 0; print 0 for all counts */
    printf("regular files  = %7ld, %5.2f %%\n", nreg,  nreg*100.0/ntot);
    printf("directories    = %7ld, %5.2f %%\n", ndir,  ndir*100.0/ntot);
    printf("block special  = %7ld, %5.2f %%\n", nblk,  nblk*100.0/ntot);
    printf("char special   = %7ld, %5.2f %%\n", nchr,  nchr*100.0/ntot);
    printf("FIFOs          = %7ld, %5.2f %%\n", nfifo, nfifo*100.0/ntot);
    printf("symbolic links = %7ld, %5.2f %%\n", nslink,nslink*100.0/ntot);
    printf("sockets        = %7ld, %5.2f %%\n", nsock, nsock*100.0/ntot);

    exit(ret);
}

/*
 * Descend through the hierarchy, starting at "pathname".
 * The caller's func() is called for every file.
 */

#define FTW_F   1       /* file other than directory */
#define FTW_D   2       /* directory */
#define FTW_DNR 3       /* directory that can't be read */
#define FTW_NS  4       /* file that we can't stat */

static char *fullpath;      /* contains full pathname for every file */

static int                  /* we return whatever func() returns */
myftw(char *pathname, Myfunc *func)
{
    fullpath = path_alloc(NULL);    /* malloc's for PATH_MAX+1 bytes */
                                    /* (Program 2.2) */
    strcpy(fullpath, pathname);     /* initialize fullpath */

    return(dopath(func));
}
/*
 * Descend through the hierarchy, starting at "fullpath".
 * If "fullpath" is anything other than a directory, we lstat() it,
 * call func(), and return.  For a directory, we call ourself
 * recursively for each name in the directory.
 */
static int                  /* we return whatever func() returns */
dopath(Myfunc* func)
{
    struct stat     statbuf;
    struct dirent   *dirp;
```

```
    DIR             *dp;
    int             ret;
    char            *ptr;

    if (lstat(fullpath, &statbuf) < 0)
        return(func(fullpath, &statbuf, FTW_NS));   /* stat error */

    if (S_ISDIR(statbuf.st_mode) == 0)
        return(func(fullpath, &statbuf, FTW_F));    /* not a directory */

    /*
     * It's a directory.  First call func() for the directory,
     * then process each filename in the directory.
     */

    if ( (ret = func(fullpath, &statbuf, FTW_D)) != 0)
        return(ret);

    ptr = fullpath + strlen(fullpath);  /* point to end of fullpath */
    *ptr++ = '/';
    *ptr = 0;

    if ( (dp = opendir(fullpath)) == NULL)
        return(func(fullpath, &statbuf, FTW_DNR));
                                        /* can't read directory */

    while ( (dirp = readdir(dp)) != NULL) {
        if (strcmp(dirp->d_name, ".") == 0  ||
            strcmp(dirp->d_name, "..") == 0)
                continue;       /* ignore dot and dot-dot */

        strcpy(ptr, dirp->d_name);  /* append name after slash */

        if ( (ret = dopath(func)) != 0)     /* recursive */
            break;  /* time to leave */
    }
    ptr[-1] = 0;    /* erase everything from slash onwards */

    if (closedir(dp) < 0)
        err_ret("can't close directory %s", fullpath);

    return(ret);
}
static int
myfunc(const char *pathname, const struct stat *statptr, int type)
{
    switch (type) {
    case FTW_F:
        switch (statptr->st_mode & S_IFMT) {
        case S_IFREG:   nreg++;     break;
        case S_IFBLK:   nblk++;     break;
        case S_IFCHR:   nchr++;     break;
        case S_IFIFO:   nfifo++;    break;
        case S_IFLNK:   nslink++;   break;
        case S_IFSOCK:  nsock++;    break;
        case S_IFDIR:
            err_dump("for S_IFDIR for %s", pathname);
                    /* directories should have type = FTW_D */
        }
        break;

    case FTW_D:
        ndir++;
        break;

    case FTW_DNR:
        err_ret("can't read directory %s", pathname);
```

```
            break;

        case FTW_NS:
            err_ret("stat error for %s", pathname);
            break;

        default:
            err_dump("unknown type %d for pathname %s", type, pathname);
        }

        return(0);
    }
```

在程序中，我们提供了比所要求的更多的通用性，这样做的目的是为了例示实际 ftw函数的应用情况。例如，函数myfunc总是返回0，但是调用它的函数却准备处理非0返回。

关于降序遍历文件系统的更多信息，以及在很多标准UNIX命令（如find, ls, tar等）中使用这种技术的情况，请参阅 Fowler,Korn及Vo〔1989〕。4.3+BSD提供了一新套的目录遍历函数——请参阅fts (3) 手册页。

4.22 chdir, fchdir 和getcwd 函数

每个进程都有一个当前工作目录，此目录是搜索所有相对路径名的起点 (不以斜线开始的路径名为相对路径名)。当用户登录到 UNIX系统时，其当前工作目录通常是口令文件(/etc/passwd)中该用户登录项的第6个字段——用户的起始目录。当前工作目录是进程的一个属性，起始目录则是登录名的一个属性。进程调用chdir或fchdir函数可以更改当前工作目录。

```
#include <unistd.h>

int chdir(const char *pathname);

int fchdir(int filedes);
```

两个函数的返回：若成功则为0，若出错则为-1

在这两个函数中，可以分别用*pathname*或打开文件描述符来指定新的当前工作目录。

> fchdir不是POSIX.1的所属部分，SVR4和4.3+BSD则支持此函数。

实例

因为当前工作目录是一个进程的属性，所以它只影响调用 chdir的进程本身，而不影响其他进程（我们将在第8章较详细地说明进程之间的关系）。这就意味着程序4-8并不会产生我们希望得到的后果。如果编译程序4-8，并且调用其可执行目标代码文件，则可以得到下列结果：

```
$ pwd
/usr/lib
$ mycd
chdir to /tmp succeeded
$ pwd
/usr/lib
```

从中可以看出，执行mycd程序的shell的当前工作目录并没有改变。由此可见， shell应当直接调用chdir函数，所以cd命令的执行程序直接包含在shell程序中。

因为内核保持有当前工作目录的信息，所以我们应能取其当前值。不幸的是，内核为每个进程只保存其当前工作目录的i节点编号以及设备标识，并不保存该目录的完整路径名。

程序4-8 chdir函数实例

```
#include    "ourhdr.h"

int
main(void)
{
    if (chdir("/tmp") < 0)
        err_sys("chdir failed");

    printf("chdir to /tmp succeeded\n");
    exit(0);
}
```

我们需要一个函数，它从当前工作目录开始，找到其上一级的目录，然后读其目录项，直到该目录项中的i节点编号数与工作目录i节点编号数相同，这样地就找到了其对应的文件。按照这种方法，逐层上移，直到遇到根，这样就得到了当前工作目录的绝对路径名。很幸运，函数getcwd就是提供这种功能的。

```
#include<unistd.h>

char *getcwd(charbuf, size_tsize);
```

返回：若成功则为buf，若出错则为NULL

向此函数传递两个参数，一个是缓存地址 *buf*，另一个是缓存的长度 *size*。该缓存必须有足够的长度以容纳绝对路径名再加上一个null终止字符，否则返回出错（请回忆2.5.7节中有关为最大长度路径名分配空间的讨论）。

> 某些getcwd的实现允许第一个参数 *buf*为NULL。在这种情况下，此函数调用malloc动态地分配*size*字节数的空间。这不是POSIX.1或XPG3的所属部分，应予避免。

实例

程序4-9将工作目录更改至一个特定的目录，然后调用 getcwd,最后打印该工作目录。如果运行该程序，则可得：

```
$ a.out
cwd = /var/spool/uucppublic
$ ls -l/usr/spool
lrwxrwxrwx  1 root  12 Jan 31 07:57 /usr/spool -> ../var/spool
```

程序4-9 getcwd函数实例

```
#include    "ourhdr.h"

int
main(void)
{
```

```
    char    *ptr;
    int     size;

    if (chdir("/usr/spool/uucppublic") < 0)
        err_sys("chdir failed");

    ptr = path_alloc(&size);    /* our own function */
    if (getcwd(ptr, size) == NULL)
        err_sys("getcwd failed");

    printf("cwd = %s\n", ptr);
    exit(0);
}
```

注意，chdir跟随符号连接（正如在表4-7所示），但是当getcwd沿目录树上溯遇到/var/spool目录时，它并不了解该目录由符号连接/usr/spool所指向。这是符号连接的一种特性。

4.23 特殊设备文件

st_dev和st_rdev这两个字段经常引起混淆，当在11.9节讨论ttyname函数时，需要使用这两个字段。有关规则很简单：

- 每个文件系统都由其主、次设备号而为人所知。设备号所用的数据类型是基本系统数据类型dev_t。回忆图4-1，一个磁盘经常包含若干个文件系统。
- 我们通常可以使用两个大多数实现都定义的宏： major和minor来存取主、次设备号。这就意味着我们无需关心这两个数是如何存放在dev_t对象中的。

> 早期的系统用16位整型存放设备号： 8位用于主设备号， 8位用于次设备号。SVR4使用32位：14位用于主设备号， 18位用于次设备号。4.3+BSD则使用16位：8位用于主设备号，8位用于次设备号。
>
> POSIX.1说明dev_t类型是存在的，但没有定义它包含什么，或如何取得其内容。大多数实现定义了宏major和minor，但在哪一个头文件中定义它们则与实现有关。

- 系统中每个文件名的st_dev值是文件系统的设备号，该文件系统包含了该文件名和其对应的i节点。
- 只有字符特殊文件和块特殊文件才有st_rdev值。此值包含该实际设备的设备号。

实例

程序4-10为每个命令行参数打印设备号，另外，若此参数引用的是字符特殊文件或块特殊文件，则也打印该特殊文件的st_rdev值。

程序4-10 打印st_dev和st_rdev值

```
#include    <sys/types.h>   /* BSD: defines major() and minor() */
#include    <sys/stat.h>
#include    "ourhdr.h"

int
main(int argc, char *argv[])
{
    int         i;
    struct stat buf;
```

```
        for (i = 1; i < argc; i++) {
            printf("%s: ", argv[i]);
            if (lstat(argv[i], &buf) < 0) {
                err_ret("lstat error");
                continue;
            }

            printf("dev = %d/%d", major(buf.st_dev),  minor(buf.st_dev));

            if (S_ISCHR(buf.st_mode) || S_ISBLK(buf.st_mode)) {
                printf(" (%s) rdev = %d/%d",
                        (S_ISCHR(buf.st_mode)) ? "character" : "block",
                        major(buf.st_rdev), minor(buf.st_rdev));
            }
            printf("\n");
        }
        exit(0);
    }
```

在SVR4中，为了定义宏major和minor，一定要包括头文件<sys/sysmacros.h>。运行此程序得到下面的结果：

```
$ a.out / /home/stevens /dev/tty[ab]
/: dev = 7/0
/home/stevens: dev = 7/7
/dev/ttya: dev = 7/0 (character) rdev = 12/0
/dev/ttyb: dev = 7/0 (character) rdev = 12/1
$ mount   察看安装情况
/dev/sd0a on /
/dev/sd0h on /home
$ ls -1 /dev/sd0[ah] /dev/tty[ab]
brw-r-----  1 root       7,    0 Jan 31 08:23 /dev/sd0a
brw-r-----  1 root       7,    7 Jan 31 08:23 /dev/sd0h
crw-rw-rw-  1 root      12,    0 Jan 31 08:22 /dev/ttya
crw-rw-rw-  1 root      12,    1 Jul  9 10:11 /dev/ttyb
```

传递给该程序的头两个参数是目录（根和/home/stevens），后两个是设备名/dev/tty[ab]，这两个设备是字符特殊设备。从程序的输出可见，根目录和/home/stevens目录的设备号不同，这表示它们位于不同的文件系统中。运行mount(1)命令证明了这一点。然后用ls命令察看由mount命令报告的两个磁盘设备和两个终端设备。这两个磁盘设备是块特殊设备，而两个终端设备则是字符特殊设备。（通常，只有块特殊设备才能包含随机存取文件系统，它们是：硬、软盘驱动器和CD-ROM等。UNIX的早期版本支持磁带存放文件系统，但这从未广泛使用过。）注意，两个终端设备（st_dev）的文件名和i节点在设备7/0上（根文件系统，它包含了/dev文件系统），但是它们的实际设备号是：12/0和12/1。

4.24 sync和fsync函数

传统的UNIX实现在内核中设有缓冲存储器，大多数磁盘I/O都通过缓存进行。当将数据写到文件上时，通常该数据先由内核复制到缓存中，如果该缓存尚未写满，则并不将其排入输出队列，而是等待其写满或者当内核需要重用该缓存以便存放其他磁盘块数据时，再将该缓存排入输出队列，然后待其到达队首时，才进行实际的I/O操作。这种输出方式被称之为延迟写（delayed write）（Bach〔1986〕第3章详细讨论了延迟写）。延迟写减少了磁盘读写次数，但是

却降低了文件内容的更新速度，使得欲写到文件中的数据在一段时间内并没有写到磁盘上。当系统发生故障时，这种延迟可能造成文件更新内容的丢失。为了保证磁盘上实际文件系统与缓存中内容的一致性，UNIX系统提供了sync和fsync两个系统调用函数。

```
#include <unistd.h>

void sync(void);

int fsync(int filedes);
```

返回：若成功则为0，若出错则为-1

sync只是将所有修改过的块的缓存排入写队列，然后就返回，它并不等待实际I/O操作结束。

系统精灵进程(通常称为update)一般每隔30秒调用一次sync函数。这就保证了定期刷新内核的块缓存。命令sync(1)也调用sync函数。

函数fsync只引用单个文件(由文件描述符filedes指定)，它等待I/O结束，然后返回。fsync可用于数据库这样的应用程序，它确保修改过的块立即写到磁盘上。比较一下fsync和O_SYNC标志(见3.13节)。当调用fsync时，它更新文件的内容，而对于O_SYNC，则每次对文件调用write函数时就更新文件的内容。

> SVR4和4.3+BSD两者都支持sync和fsync,它们都不是POSIX.1的组成部分，但XPG3要求fsync。

4.25 文件存取许可权位小结

我们已经说明了所有文件存取许可权位，其中某些位有多种用途。表4-10列出了所有这些许可权位，以及它们对普通文件和目录文件的作用。

表4-10 文件存取许可权位小结

常 数	说 明	对普通文件的影响	对目录的影响
S_ISUID	设置-用户-ID	执行时设置有效用户ID	（不使用）
S_ISGID	设置-组-ID	若组执行位设置，则执行时设置有效组ID，否则使强制性锁起作用	将在目录中创建的新文件的组ID设置为目录的组ID
S_ISVTX	粘住位	在交换区保存程序正文（若支持）	禁止在目录中删除和更名文件
S_ISUSR	用户读	许可用户读文件	许可用户读目录项
S_IWUSR	用户写	许可用户写文件	许可用户在目录中删除或创建文件
S_IXUSR	用户执行	许可用户执行文件	许可用户在目录中搜索给定路径名
S_IRGRP	组读	许可组读文件	许可组读目录项
S_IWGRP	组写	许可组写文件	许可组在目录中删除或创建文件
S_IXGRP	组执行	许可组执行文件	许可组在目录中搜索给定路径名
S_IROTH	其他读	许可其他读文件	许可其他读目录项
S_IWOTH	其他写	许可其他写文件	许可其他在目录中删除或创建文件
S_IXOTH	其他执行	许可其他执行文件	许可其他在目录中搜索给定路径名

最后9个常数分成3组。

```
S_IRWXU = S_IRUSR|S_IWUSR|S_IXUSR
S_IRWXG = S_IRGRP|S_IWGRP|S_IXGRP
S_IRWXO = S_IROTH|S_IWOTH|S_IXOTH
```

4.26 小结

本章内容围绕stat函数，详细介绍了stat结构中的每一个成员。这使我们对UNIX文件的各个属性都有所了解。对文件的所有属性以及对文件进行操作的所有函数有完整的了解对各种UNIX程序设计都非常重要。

习题

4.1 用stat 函数替换程序4-1中的lstat 函数，对于命令行的参数是符号连接的情况有什么变化。

4.2 表4-1指出SVR4没有提供宏S_ISLNK，但是SVR4支持符号连接并且在 <sys/stat.h> 中定义了S_IFLNK，如何修改ourhdr.h使得需要S_ISLNK宏的程序可以使用它？

4.3 如果文件方式创建屏蔽字是777（八进制），结果会怎样？利用umask命令验证结论。

4.4 验证关闭一个你所拥有的文件的用户读许可权，也就是不允许你访问自己的文件。

4.5 创建文件foo和bar后运行程序4-3会产生什么情况？

4.6 4.12节中讲到一个常规文件的大小可以为 0，同时我们又知道 st_size 字段定义为目录或符号连接，那么目录和符号连接的长度是否可以为0？

4.7 编写一个类似cp(1)的程序，它复制包含空洞的文件，但不将字节0写到输出文件中去。

4.8 4.12节ls命令的输出中，core和core.copy的存取许可权不相同，如果创建两个文件时umask没有变，说明为什么会发生这种差别。

4.9 程序4-5使用了df(1)命令来检查空闲的磁盘空间。为什么不能使用du(1)命令?

4.10 表4-9中给出unlink函数会修改文件状态改变时间，这是怎样实现的？

4.11 4.21节中，系统对可打开文件数的限制对myftw函数会产生什么影响？

4.12 4.21节中的ftw 从不改变其目录，对这种处理方法进行改动：每次遇到一个目录就对之进行chdir，这样每次调用lstat时就可以使用文件名而非路径名，处理完所有的目录后执行chdir ("..")。比较这种方法和正文中方法的运行时间。

4.13 每个进程都有一个根目录用于处理绝对路径名，可以通过 chroot函数改变根目录。在手册中查阅此函数说明这个函数什么时候有用。

4.14 如何利用utime函数只设置两个时间值中的一个？

4.15 有些版本的finger(1)命令输出“ New mail received ... ”和“ unread since ... ”，其中...是相应的日期和时间。程序是如何决定这些日期和时间的？

4.16 用cpio(1)和tar(1)命令检查档案文件时改变了每个被归档文件的哪些时间值？复原后的文件的存取时间是多少？为什么？

4.17 file(1)命令通过读文件的第一部分并检查其内容，利用一些启发式规则确定文件的类型，如C程序、Fortran程序、shell脚本等等。UNIX提供了一条命令，它允许我们执行另一条命令，并可以跟踪该命令执行的所有系统调用。在 SVR4中该命令为truss(1)，在4.3+BSD中为ktrace(1)和kdump(1)，下例使用SunOS的 trace(1)跟踪file命令中的系统调用：

```
trace file a.out
```

下面是file调用的函数：

```
lstat ("a.out", 0xf7fff650) = 0
open ("a.out", 0, 0) = 3
read (3,""..., 512) = 512
fstat (3,0xf7fff160) = 0
write (1, "a.out: demand paged execu" .., 44) = 44
a.out: demand paged executable not stripped
utime ("a.out", 0xf7fff1b0) = 0
```

为什么file命令调用utime？

4.18　UNIX对目录的深度有限制吗？编一个程序循环创建目录并修改目录名，确定叶子节点的绝对路径名的长度大于系统的 PATH_MAX限制。可以调用 getcwd得到目录的路径名吗？标准UNIX工具是如何处理长路径名的？对目录可以使用 tar或cpio命令吗？

4.19　3.15节中描述了/dev/fd的特征，如果每个用户都可以访问这些文件，则其许可权必须为rw-rw-rw-。有些程序创建输出文件时，先进行删除以确保该文件名不存在。

```
unlink (path);
if ( (fd = creat(path, FILE_MODE)) < 0)
    err_sys(...);
```

讨论一下path是/dev/fd/1的情况。