第5章 文件属性管理

[单元概述]

Linux 系统中文件是由 i 节点和数据块两部分组成的，其中 i 节点保存了文件属性相关信息。本章主要针对文件属性进行讲解，包括文件属性的获取和修改，硬链接/符号链接概念，以及重定向相关概念。

本章最后设计了一个“显示文件长格式信息”项目，该项目显示文件属性信息，显示的信息包括文件类型、访问权限、文件属主、组、长度、访问日期、时间等。

[教学重点与难点]

重点：文件属性、文件类型、文件权限、硬链接和符号链接、重定向。

难点：文件属性各个成员的输出、输入输出重定向、硬链接和符号链接的区别。

5.1 获取文件属性

系统调用 stat/fstat/lstat 用于获取文件索引节点 inode 中的属性信息并保存在结构体 stat 中，如表 5.1 所示。

表 5.1　　系统调用 stat/fstat/lstat

项目	描述
头文件	#include <sys/types.h> #include <sys/stat.h> #include <unistd.h>

原型	int stat(const char *file_name, struct stat *buf);
	int fstat(int filedes, struct stat *buf);
	int lstat(const char *file_name, struct stat *buf);

功能	该函数获取指定文件的属性并放入 buf 中
参数	file_name(stat 函数)：要获取属性的文件名 filedes：要获取属性的文件的文件描述符 file_name(lstat 函数)：要获取属性的符号链接文件的文件名 buf：用于保存文件属性的 stat 结构体
返回值	如果成功获取文件属性，返回 0；如果失败，则返回－1，并且 errno 为错误码

系统调用 stat、fstat 和 lstat 的功能类似，都是获取指定文件的属性并保存在一个 stat 类型的结构体 buf 中。它们的区别是：系统调用 stat()通过给出文件名来获取该文件属性，而 fstat()则是通过文件描述符来获取文件属性，因此，fstat()在使用前，应该首先使用 open()打开该文件并得到该文件的文件描述符，而 stat()函数并不需要事先打开文件。lstat()是专门用于获取符号链接文件本身的属性的。符号链接文件是 Linux 系统中的一种“快捷方式”，它用于指向一个目标文件。所有对符号链接文件的操作都会转移到其所链接的目标文件上。当对一个符号链接文件执行 stat()时，获取的是该符号链接文件链接的目标文件的属性而不是符号链接文件的属性。因此，当我们想获得符号链接文件本身的属性时，应该使用 lstat()。

例如获取/etc/passwd 文件属性可以使用如下代码：

```
struct stat  buf;
int rt;
rt=lstat("/etc/passwd",&buf);
if(rt==-1)
    perror("lstat");
```

stat()系列函数执行成功时，获得的文件属性将填入一个 stat 类型的结构体 buf 中。在 Linux 中，struc stat 的定义是这样的：

```
struct stat {
    dev_t         st_dev;     /* 文件所在设备的设备 ID 号 */
    ino_t         st_ino;     /* inode 节点编号 */
    mode_t        st_mode;    /* 文件的类型与权限 */
    nlink_t       st_nlink;   /* 硬链接数 */
    uid_t         st_uid;     /* 文件主的用户 ID */
    gid_t         st_gid;     /* 文件主的组 ID */
    dev_t         st_rdev;    /* 设备 ID 号(如果这是一个设备文件的 inode) */
    off_t         st_size;    /* 文件总长度字节数 */
    blksize_t     st_blksize; /* 文件系统 I/O 块大小 */
    blkcnt_t      st_blocks;  /* 分配给文件的块数 */
    time_t        st_atime;   /* 文件最后访问时间 */
    time_t        st_mtime;   /* 文件的最后修改时间 */
    time_t        st_ctime;   /* t 索引节点的最后修改时间 */
};
```

对于常规文件或者符号链接文件，st_size 给出了以字节为单位的文件长度。st_blocks 给出了文件所分配的磁盘块。一般情况，一个磁盘块是 512 字节。

【基本知识】判断文件类型

判断文件的类型使用 stat 结构体中的 st_mode 成员，一般有两种方法：

方法一：

通过调用 Linux 系统定义的文件类型判断宏，当被判断的文件是该类型时，该宏返回 1，否则返回 0。文件类型判断宏如下：

S_ISREG(st_mode)　判断是否为常规文件

S_ISDIR(st_mode)　判断是否为目录文件

S_ISCHR(st_mode) 判断是否为字符设备文件
S_ISBLK(st_mode) 判断是否为块设备文件
S_ISFIFO(st_mode) 判断是否为 fifo 文件
S_ISLNK(st_mode) 判断是否为符号链接文件
S_ISSOCK(st_mode) 判断是否为 socket 文件

示例代码如下：

```
//假设 buf 为获得的文件属性,类型为结构体 struct stat。
if(S_ISREG(buf.st_mode))
    putchar('一');
```

方法二：

通过检查 st_mode 高四位的值来判断，可以将 st_mode 与文件权限掩码 S_IFMT(八进制 0170000)进行“与”操作，屏蔽掉低 12 位，保留高四位，然后与文件类型宏进行比较，文件类型宏见 4.1.4 节。

代码可以写成一个 switch 结构，如：

```
switch(buf.st_mode  & S_IFMT )
{
  case S_IFREG: printf("一"); break;//代表普通文件
  case S_IFDIR: printf("d");break;//代表目录文件
  ……
  default:printf("?");
}
```

【基本知识】判断文件权限

文件访问权限的判断要简单一些，可以将 st_mode 与各用户读、写和执行权限进行“与”操作来判断。

例如，要判断该文件主人是否有读权限，可以执行“st_mode & S_IRUSR”，S_IRUSR 的八进制是 00400，执行“与”操作后，st_mode 只保留了代表文件主人读权限的第 8bit。因此，“与”操作的结果如果为 1，说明文件主人有读权限；否则如果结果为 0，则没有读权限。对其他的访问权限的判断也是类似的。

判断文件主人是否有读权限代码如下：

```
if( (buf.st_mode & S_IRUSR)==0)
       putchar('一');
else
       putchar('r');
```

或

```
(buf.st_mode & S_IRUSR) ?   putchar('r') : putchar('一');
```

【创新能力】判断 setuid、setgid 和 sticky 权限

判断这三个修饰权限的方式与判断文件访问权限类似，只是在文件的长格式信息显示时，这三个权限不是独立显示的，而是显示在三类用户的执行位。

setuid 显示在文件主人的执行位(显示为 s/S),setgid 显示在组成员的执行位(显示为 s/S),sticky 位显示在其他用户的执行位(显示为 t/T),因此三类用户的执行位分别会有四种显示的可能:—、x、s/t、S/T。

如果某文件具有修饰权限,则相应的执行位就显示 s/t(对应用户具有执行权限)或 S/T(对应用户没有执行权限)。

如果某文件没有修饰权限,则相应的执行位就显示 x(对应用户具有执行权限)或—(对应用户没有执行权限)。

例如判断某文件是否具有 setuid 权限,可以使用如下代码:

```
if( (buf.st_mode & S_ISUID)==0)
    (buf.st_mode & S_IXUSR)  ?   putchar('x') : putchar('-');
else
    (buf.st_mode & S_IXUSR)  ?   putchar('s') : putchar('S');
```

5.2　用户/组 ID 与名字的转换

在"ls -l"命令中,可以得到的文件的信息里包括文件的主人名及所属的组名。而系统调用 stat()得到的文件属性信息中,有 st_uid 和 st_gid。它们是文件主人的 uid 和所属的组的 gid。它们的类型分别是 uid_t 和 gid_t 类型,即 int 类型。

那么,如何根据 uid 和 gid,得到对应的用户名和组名呢? Linux 系统中提供了 getpwuid()和 getgrgid()这两个函数来实现这个功能。

在 Linux 文件系统中,/etc/passwd 文件是一个文本文件。该文件中保存有系统账户信息。该文件的每一行对应着一个账户信息,包括用户 ID、组 ID、主目录、使用的 Shell 等。其格式为:

```
用户名:密码:UID:GID:GECOS:主目录:Shell
```

函数 getpwuid()可以根据 uid 在/etc/passwd 文件中获取用户信息并在 passwd 结构体类型变量中返回,如表 5.2 所示。

表 5.2　　函数 getpwuid

项目	描述
头文件	#include <sys/types.h> #include <pwd.h>
原型	struct passwd *getpwuid(uid_t uid);
功能	该函数根据 uid 获取对应的账户信息并放入 passwd 类型的结构体变量中,并返回指向该结构体变量的指针。
参数	uid:指定的用户 id(uid)
返回值	如果成功则返回指向一个 passwd 结构体的指针;如果失败,则返回 NULL。

passwd 结构体的定义如下:

```
struct passwd {
```

```
    char          *pw_name;       /* 用户名 */
    char           *pw_passwd;    /* 用户密码 */
    uid_t          pw_uid;        /* 用户 ID */
    gid_t          pw_gid;        /* 组 ID */
    char          *pw_gecos;      /* 实际用户名 */
    char          *pw_dir;        /* 主目录 */
    char          *pw_shell;      /* Shell */
};
```

可以看出，在 passwd 结构体中，有用户名(pw_name)这项信息。这就是我们希望得到的用户名。

类似的，如表 5.3 所示的函数 getgrgid()根据文件的 gid，到/etc/group 文件中获取组名的信息。在/etc/group 文件中，保存着系统所有组的信息，包括组名、密码、组 ID 和用户列表四项信息。getgrgid()根据指定的 gid，获取相关信息并填入一个 group 结构体中，并返回指向该结构体变量的指针。

表 5.3　　函数 getgrgid

项目	描述
头文件	#include <sys/types.h> #include <grp.h>
原型	struct group *getgrgid(gid_t gid);
功能	该函数根据 gid 获取对应的组信息并放入 group 类型的结构体变量中，并返回指向该结构体变量的指针。
参数	gid:指定的组 id(gid)
返回值	如果成功则返回指向一个 group 结构体的指针；如果失败，则返回 NULL。

group 结构体的定义如下：

```
struct group
{
    char          *gr_name;       /* 组名 */
    char           *gr_passwd;    /* 组密码 */
    gid_t          gr_gid;        /* 组 ID */
    char          **gr_mem;       /* 组成员 */
};
```

同样可以看出，在 group 结构体中，有组名(gr_name)这项信息。这就是我们希望得到的组名。

我们可以这样编写代码来根据 stat()得到的 uid 和 gid 获取用户名和组名(假设要获取 f1 文件的用户名)：

```
struct stat buf;
struct passwd *usr;
struct group *grp;
```

```
lstat("f1",&buf);//获得文件信息
usr=getpwuid(buf.st_uid);
printf("   %s",usr->pw_name);     //usr->pw_name 就是对应的用户名
grp=getgrgid(buf.st_gid);
printf("   %s",grp->gr_name);     //grp->gr_name 就是对应的组名
```

5.3　硬链接与符号链接

5.3.1　硬链接与符号链接的区别

根据前一章的讲解可知，每个保存在磁盘上的文件都有一个 i 节点与之对应，一个 i 节点可以对应一个或多个文件名，而与 i 节点对应的文件名的个数就是在“ls -l”命令中显示的文件链接数，也即硬链接个数。

硬链接和符号链接都是指向另一个已存文件的链接，符号链接文件相当于 Windows 中的快捷方式。它们主要有以下区别：

(1)命令。

硬链接命令：ln　原文件　硬链接文件　（注意：不能对目录创建硬链接）

符号链接命令：ln -s　原文件/目录　符号链接文件

(2)新增文件。

硬链接：不新增加真实的文件，仅增加一个指向原文件 i 节点的文件名。

符号链接：增加一个真实的文件即符号链接文件，新增的符号链接文件有自己的 i 节点，文件内容为设置符号连接时指定的原文件或目录的路径名。

(3)删除原文件/目录。

硬链接：删除原文件，使文件对应 i 节点的链接数减 1，减为 0 则删除该文件。

符号链接：删除原文件，符号链接文件失效，但该文件依然存在，如果后期又新建了一个与符号链接文件内容相同路径的文件，则符号链接文件重新有效。

(4)删除链接文件。

删除硬链接文件：使文件对应 i 节点的链接数减 1，减为 0 则删除该文件。

删除符号链接文件：对原文件/目录无任何影响。

(5)是否跨文件系统。

硬链接：不可以，不同的文件系统其文件的组织方式和结构可能不一样，因此不能随意创建跨文件系统的硬链接文件。

符号链接：可以，因为符号链接文件仅仅保存了原文件或目录的路径名，因此不受文件系统的影响。

5.3.2 相关的系统调用函数

相关的系统调用函数如表 5.4～表 5.7 所示。

表 5.4　　函数 link

项目	描述
头文件	#include <unistd.h>
原型	int link(char *pathname1, char *pathname2);
功能	创建一个硬链接文件
参数	pathname1 表示已存在文件 pthname2 表示硬链接文件
返回值	成功返回 0,失败返回－1(置 errno)

表 5.5　　函数 unlink

项目	描述
头文件	#include <unistd.h>
原型	int unlink(char *pathname);
功能	删除一个文件,如果该文件对应的 i 节点有多个文件名,则删除一个文件仅仅代表将该文件的硬链接数减 1,直至减为 0 才真正删除该文件
参数	pathname:要删除的链接文件名
返回值	成功返回 0,失败返回－1(置 errno)
备注	删除文件时,如果已经打开则要延迟到文件关闭后才真正删除

表 5.6　　函数 symlink

项目	描述
头文件	#include <unistd.h>
原型	int symlink(char *actualpath, char *sympath);
功能	创建一个符号链接文件。
参数	actualpath 表示真实存在的文件或目录 sympath 表示符号链接文件
返回值	成功返回 0,失败返回－1(置 errno)

表 5.7　　函数 readlink

项目	描述
头文件	#include <unistd.h>
原型	int readlink(char *pathname,char *buf, int bufsize);
功能	读取符号链接所指原文件名
参数	pathname:符号链接文件名 buf:存放被链接文件名的缓冲区 bufsize:缓冲区大小
返回值	成功返回实际写入缓冲区的字节数,失败返回 0

5.4　dup/dup2

5.4.1　输入输出重定向

我们知道,执行一个 Shell 命令行程序时通常会自动打开三个标准文件,即标准输入文件(stdin),通常对应终端的键盘;标准输出文件(stdout)和标准错误输出文件(stderr),这两个文件都对应终端的屏幕。进程将从标准输入文件中得到输入数据,将正常输出数据输出到标准输出文件,而将错误信息送到标准错误文件中。

以 cat 命令为例,不带参数的 cat 会从标准输入中读取数据,并将其送到标准输出。例如:

```
# cat
hello
hello
world
world
ctrl+D结束输入
```

直接使用标准输入/输出文件存在以下问题:

(1)对用户花费了很长时间从键盘输入的数据,如果想再使用这些数据时,需要重新输入,严重浪费时间和精力。

(2)输出到屏幕上的信息只能看不能进行修改和处理,不方便用户进一步操作。

为了解决上述问题,Linux 系统引入了输入输出重定向。

输入重定向是指把命令(或可执行程序)的标准输入重定向到指定的文件中。也就是说,输入可以不来自键盘,而来自一个指定的文件。所以说,输入重定向主要用于改变一个命令的输入源,特别是改变那些需要大量输入的输入源。Shell 中输入重定向的符号为<或<<,如“命令<文件名”,可将某文件的内容代替键盘输入作为某个命令的输入参数。如“cat</etc/passwd”显示某文件的内容功能。

输出重定向是指把命令(或可执行程序)的标准输出或标准错误输出重新定向到指定文件中。这样,该命令的输出就不显示在屏幕上,而是写入到指定文件中。Shell 中输出重定向的符

号为>或>>,如“命令>文件名”,可将某命令的输出从屏幕上改为输出到某文件中。

不论是输入还是输出重定向,其实质就是将标准输入、标准输出对应的文件描述符“复制”到某个文件的文件描述符上。这个“复制”的过程可以使用 dup/dup2 实现。

5.4.2 dup/dup2 函数

如表 5.8 所示的 dup 和 dup2 都是对文件描述符 oldfd 进行复制,得到 oldfd 的一个拷贝。复制后得到的文件描述符与源文件描述符一样,指向文件表中相同的一个打开文件表项。因此这两个文件描述符共享锁、文件读写指针等。例如,当对一个文件描述符执行了 lseek()修改了文件读写指针后,另一个文件描述符的读写指针也发生变化(其实它们使用的是同一个读写指针)。

表 5.8 系统调用 dup/dup2

项目	描述
头文件	#include <unistd.h>
原型	int dup(int oldfd); int dup2(int oldfd, int newfd);
功能	对指定的文件描述符进行复制
参数	olddf:要复制的原文件描述符 newfd:复制后的新文件描述符
返回值	如果成功,返回复制的新文件描述符;如果失败,则返回-1,并且 errno 为错误码

dup 和 dup2 的区别是:dup 将 oldfd 复制到用户文件描述符表中最小可用文件描述符,而 dup2 可以指定将 oldfd 复制为 newfd。

例如,一个进程执行了两个 open(),分别打开了 f1 和 f2 文件,得到两个文件描述符 fd1 和 fd2,然后对 f1 进行复制:

```
……..
   fd1 = open("f1", O_RDONLY);
      fd2 = open ("f2", O_RDONLY);
     dup(fd1);
…..
```

在执行上述代码后,将文件 f1 的文件描述符(文件描述符 fd1 的值为 3)复制为 5。内存中相关数据结构如图 5.1 所示。

【知识验证】dup 对文件读写的影响

本例对第四章的 fileopenon.c 进行改造,两次打开同一个文件进行读写操作,然后对第一个文件描述符调用 dup 并写入,然后再通过三个文件描述符读取文件内容并观察结果。

(1)源程序(fileopenone2.c)。

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <unistd.h>
4. #include <fcntl.h>
```

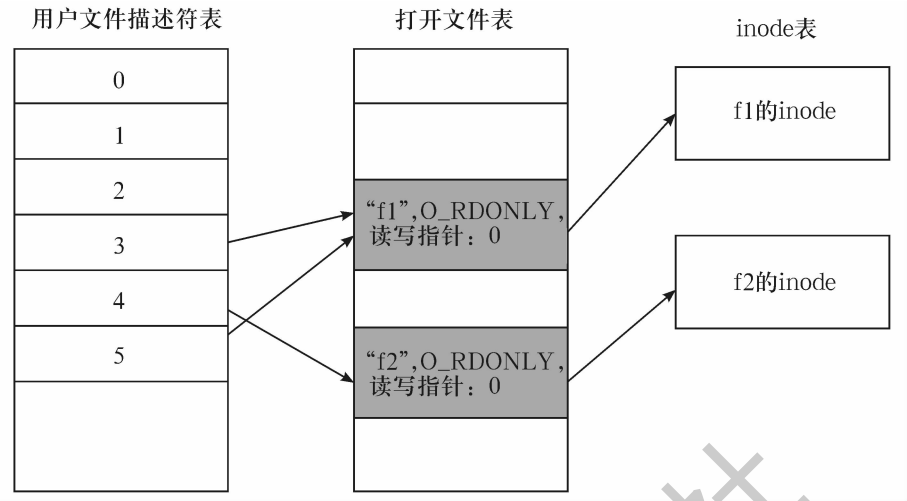

图 5.1　dup 执行后内存数据结构的内容

```
5. main()
6. {
7.      int fd1,fd2,fd3;
8.      int num;
9.      char buf[20];
10.     fd1 = open("f1",O_RDWR|O_TRUNC);
11.     if(fd1==-1)
12.     {
13.         perror("open");
14.         exit(1);
15.     }
16.     printf("fd1 is %d \n",fd1);
17.     fd2 = open("f1",O_RDWR);
18.     if(fd2==-1)
19.     {
20.         perror("open");
21.         exit(1);
22.     }
23.     printf("fd2 is %d \n",fd2);
24.     fd3=dup(fd1);
25.     printf("fd3 is %d \n",fd3);
26.     num=write(fd1,"hello world!",12);
27.     printf("fd1:write num= %d bytes into f1\n",num);
28.     num=read(fd2, buf,20);
29.     buf[num]=0;
30.     printf("fd2:read   %d bytes from f1: %s\n",num, buf);
31.     num=read(fd3, buf,20);
```

```
32.     buf[num]=0;
33.     printf("fd3:read  %d bytes from f1: %s\n",num, buf);
34.     close(fd1);
35.     close(fd2);
36.     close(fd3);
37.}
```

(2)编译。

```
#gcc  -o  fileopenone2  fileopenone2.c
```

(3)运行。

```
#./fileopenone2
```

(4)运行结果

输出的结果为：

```
[root@bogon 5]# ./fileopenone2
fd1 is 3
fd2 is 4
fd3 is 5
fd1:write num=12 bytes into f1
fd2:read  12 bytes from f1: hello world!
fd3:read  0 bytes from f1:
```

通过第四章读者已经了解每次调用 open()都会得到一个新的打开文件表项，即使两次打开的同一个文件也会有各自的文件读写指针。

刚调用 open()后，fd1 和 fd2 两个文件描述符对应的读写指针都指向文件 f1 的开始位置。此时通过文件描述符 fd1 向文件 f1 中写入了 12 个字符“hello world!”，成功写入之后其读写指针为 12。而这时另一个文件描述符 fd2 对应的读写指针仍为 0，指向文件的开始位置，因此接下来通过 fd2 从文件中读数据时，会将刚才写入的字符串读出。

在本例中队 fd1 调用了 dup()函数进行文件描述符复制返回 fd3，可以看到 fd3 与 fd1 和 fd2 的值都不同，但是在通过 fd1 写入 12 个字符后，通过 fd3 读取文件内容却什么也读不到，因为复制文件描述符后 fd3 和 fd1 使用的是同一个打开文件表项，fd1 写完后读写指针已经到了文件结尾，fd3 从文件结尾读取文件内容自然就什么也读不到了。

【基本知识】输入输出重定向

在程序中，如何通过 dup/dup2 来实现输入输出重定向呢？每个进程在创建时，都自动打开了三个标准输入输出设备文件：标准输入设备文件、标准输出设备文件和标准错误输出设备文件，它们的文件描述符分别是 0、1 和 2。当进程执行过程中，要使用 printf 输出时，都会将结果写入文件描述符为 1 的标准输出设备文件中；此时如果执行下列代码，就实现了将输出重定向到了文件 f1 中：

```
fd1 = open ("f1", O_RDWR);
close (1);
dup (fd1);
```

原因是当执行了 close(1)后，文件描述符 1 成为最小可用文件描述符。此时执行 dup(fd1)就将 f1 文件的文件描述符 fd1 复制到了 1 中，这样文件描述符 1 就不再指向标准输出设备文件

而是指向文件 f1 了，以后在执行 printf 这样的输出语句时，结果就写入了文件 f1 中而不是输出到屏幕上。

同理，下面的语句可以实现输入重定向：

```
fd1 = open ("f1", O_RDWR);
close ( 0);
dup (fd1);
```

提示：使用 dup 进行输入输出重定向后，文件描述符 0 和 1 原来指向键盘和屏幕，后来指向了文件；如果要将 0 和 1 恢复指向键盘和屏幕，可以事先使用 dup2 用其他文件描述符来指向键盘和屏幕，然后再调用 dup2 恢复回来。

5.5　文件属性的修改

5.5.1　修改文件属性

有时候，我们需要修改文件的某个属性。例如改变文件的访问权限：去掉其他用户对文件的读权限；改变文件的属主关系等等。那么到底哪些属性是可以修改的，哪些属性是不可以修改的呢？

表 5.9　　**索引节点 inode 的修改**

字段	描述	改变属性
st_dev	文件所在设备的 ID 号	不能修改
st_ino	inode 节点编号	不能修改
st_mode	文件的类型与权限	chmod 命令或函数修改文件访问权限
st_nlink	硬链接数	link()或 ln 命令
st_atime	最后一次访问文件的时间	utime()或访问文件
st_mtime	文件内容最后改变的时间	utime()或修改文件
st_ctime	索引节点最后改变的时间	通过改变文件属性间接改变
st_uid	文件主标识符	chown 命令或函数
st_gid	组标识符	chown 命令或函数
st_blocks	文件的数据块数	通过改变文件大小间接改变
st_size	文件长度	通过改变文件大小间接改变
st_blksize	文件系统 I/O 块大小	不能修改

可以看出，inode 节点中有些属性信息是可以修改的，系统提供了专门的系统调用。如 chmod()可以修改文件访问权限，chown()可以修改文件主人及组。utime()可以修改文件的访问时间及修改时间。

inode 节点中还有些属性信息是不能修改的，如 inode 节点编号 st_ino、文件所在设备的 ID 号 st_dev 等。

另外，还有一些属性信息虽然不能直接去修改，但是可以通过其他操作间接改变。如，当我们向文件中增加新的内容，会间接影响到文件的长度 st_size 字段的值。

5.5.2 改变文件属主及组 chown/fchown/lchown

如表 5.10 所示的系统调用 chown()/fchown()/lchown()，用于改变文件的属主及组。

表 5.10　　系统调用 chown/fchown/lchown

项目	描述
头文件	#include <sys/types.h> #include <unistd.h>

原型	int chown(const char *path, uid_t owner, gid_t group);
	int fchown(int fd, uid_t owner, gid_t group);
	int lchown(const char *path, uid_t owner, gid_t group);

功能	改变指定文件的主人及组
参数	path(chown 函数)：指定的文件路径名 fd：指定文件的文件描述符 file_name(lchown 函数)：指定的符号链接文件的文件路径名 owner：新的文件主 ID group：新的组 ID
返回值	如果成功，返回 0；如果失败，则返回－1，并且 errno 为错误码

chown()将 path 指定的文件的主人及组改为 owner 和 group。fchown()则需先将文件打开，并通过文件描述符进行操作。对于一个符号链接文件，要修改符号链接文件本身的主人和组，可以使用 lchown()。

只有超级用户可以改变一个文件的主人和组。文件的主人可以将他的一个文件所属的组改为他所在的任何一个组。除上述情况外，其他用户不能修改文件的主人和组。

【知识验证】修改文件组

假设系统中有一个用户组 u2，其 gid 为 502。以 root 身份编写并运行程序 chowntest.c，创建一个 f1 文件(文件主为 root，组为 root 组)，将其文件组改为 u2 组。

(1)源程序(chowntest.c)。

```
1. #include <sys/stat.h>
2. main()
3. {
4.     struct stat buf;
5.     unlink("f1");
6.     system("touch f1");
7.     stat("f1",&buf);
8.     printf("old userid:%d  grpid:%d\n",buf.st_uid,buf.st_gid);
9.     chown("f1",-1,502);
10.    stat("f1",&buf);
11.    printf("new userid:%d  grpid:%d\n",buf.st_uid,buf.st_gid);
12.}
```

在程序 chowntest.c 中，首先通过 system 函数执行 touch 命令创建 f1 文件（如果该文件存在则先删除）。然后通过 stat()获取该文件的属性信息，并输出其中的 uid 和 gid。

接下来调用 chown("f1",－1,502)将 f1 文件的组改为 gid 为 502 的组，文件主不修改。修改后再次调用 stat()获取修改后文件的属性并再次输出 uid 和 gid，以观察其变化。

(2)编译。

```
#gcc  -o  chowntest  chowntest.c
```

(3)运行。

```
#./chowntest
```

(4)运行结果。

```
[root@localhost book]#  ./chowntest
old userid:0  grpid:0
new userid:0  grpid:502
```

5.5.3　改变文件访问权限 chmod/fchmod

系统调用 chmod/fchmod 用于改变文件的访问权限，如表 5.11 所示。

表 5.11　　系统调用 chmod/fchmod

项目	描述
头文件	#include ＜sys/types.h＞ #include ＜stat.h＞
原型	int chmod(const char *path, mode_t mode); int fchmod(int fildes, mode_t mode);
功能	改变指定文件的访问权限
参数	path:指定的文件路径名 fd:指定文件的文件描述符 mode:新的文件访问权限
返回值	如果成功，返回 0;如果失败，则返回－1，并且 errno 为错误码

chmod()将 path 指定的文件的访问权限改为 mode。fchmod()则需先将文件打开，并通过文件描述符进行操作。

5.5.4　改变文件时间 utime

系统调用 utime()用于改变文件的访问时间和修改时间，如表 5.12 所示。

表 5.12　　系统调用 utime

项目	描述
头文件	#include ＜sys/types.h＞ #include ＜utime.h＞
原型	intutime(const char *filename, struct utimbuf *buf);
功能	改变指定文件的访问时间和修改时间。

参数	filename:要改变时间的文件名 buf:新的访问时间和修改时间
返回值	如果成功,返回 0;如果失败,则返回-1,并且 errno 为错误码

utime()将 filename 指定的文件的访问时间和修改时间改为 buf 中的新时间。buf 是一个 utimbuf 类型的结构体,用于存放新的访问时间和修改时间,其定义如下:

```
struct utimbuf {
        time_t actime;     /* 访问时间 */
        time_t modtime;    /* 修改时间 */
};
```

5.5.5 改变文件长度 truncate/ftruncate

系统调用 truncate()/ftruncate()用于改变文件的长度,如表 5.13 所示。

表 5.13 系统调用 truncate/ftruncate

项目	描述
头文件	#include <sys/types.h> #include <unistd.h>
原型	int truncate(const char *path, off_t length); int ftruncate(int fd, off_t length);
功能	改变指定文件的长度
参数	path:要改变长度的文件名 fd:要改变长度的文件描述符 length:新的文件长度
返回值	如果成功,返回 0;如果失败,则返回-1,并且 errno 为错误码

truncate/ftruncate 将 path 或 fd 指定的文件的长度改为 length。如果文件原长度大于 length,则超出的部分会被截掉;如果文件原长度小于 length,则文件被延长,延长的部分用 0 来填充。

在改变文件长度时,文件的读写指针不受影响。

对于 ftruncate(),文件应先以写的方式打开;对于 truncate(),用户对该文件应该具有写权限。

5.6 项目:显示文件长格式信息

5.6.1 项目分析与设计

Linux 中最常用的一个命令就是 ls 命令。ls 命令用于查看目录信息。ls 命令有很多参数选项,其中 ls-l 可以以长格式显示文件的属性信息,显示的信息包括文件类型、访问权限、文件

属主、组、长度、访问日期、时间等。

那么这个功能是如何实现的呢？我们自己能否编写一个这样的程序呢？实际上，开发这样一个程序是很有意义的。因为我们在开发项目时，不可避免地总要编写访问文件的程序，而访问文件之前，往往需要先得到文件的详细属性信息，然后再决定如何去访问文件。

在本节，我们就开发这样一个小项目。下面我们首先来了解如何从索引节点中获取文件属性。

在前面介绍过，在 Linux 系统中文件包括两部分：数据块和索引节点 inode。文件的属性信息保存在文件的索引节点 inode 中，文件的数据内容存储在数据块中。Linux 系统中提供了读取文件属性和修改文件属性的系统调用。系统调用 stat/fstat/lstat 用于获取文件索引节点 inode 中的属性信息；系统调用 chown/fchown/lchown、chmod/fchmod、utime、truncate/ftruncate、rename 等用于修改文件的相关属性信息。

要想显示文件的详细属性信息，首先要得到文件的属性信息。文件的属性信息包含在文件的索引节点中，显示的信息应该包括文件的 inode 节点编号、文件名、文件主人及组名、文件长度、文件最后修改时间等。因此该项目实现的步骤如下：

(1)根据用户输入的文件名，读取该文件的索引节点。

(2)从索引节点中提取所需的各项信息，经过必要的转换后输出。

5.6.2　项目实施

通过 lstat 系统调用，读取索引节点中的属性信息。源代码(myll.c)如下。

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <sys/stat.h>
4.  #include <linux/fs.h>
5.  #include <time.h>
6.  #include <dirent.h>
7.  #include <errno.h>
8.  #include <grp.h>
9.  #include <pwd.h>
10. void print_size(struct stat *statp)
11. {
12.     switch (statp->st_mode & S_IFMT)
13.     {
14.     case S_IFCHR:
15.     case S_IFBLK:
16.         printf(" %u, %u ", (unsigned)(statp->st_rdev >> 8),
17.         (unsigned)(statp->st_rdev & 0xFF));
18.         break;
19.     default:
20.         printf(" %u ", (unsigned long)statp->st_size);
21.     }
22. }
23. void print_date(struct stat *statp)
24. {
```

```
    time_t now;
    double diff;
    char buf[100], *fmt;
    if (time(&now) == -1)
    {
        printf(" ???????????");
        return;
    }
    diff = difftime(now, statp->st_mtime);
    if (diff < 0 || diff > 60 * 60 * 24 * 182.5) /* roughly 6 months */
        fmt = "%b %e %Y";
    else
        fmt = "%b %e %H:%M";
    strftime(buf, sizeof(buf), fmt, localtime(&statp->st_mtime));
    printf(" %s ", buf);
}
void printlong(char *name)
{
    struct stat buf;
    struct passwd *user;
    struct group *grp;
    char linkname[64];
    char dirfilename[64];
    int rt;
    rt=lstat(name,&buf);
    if(rt==-1)
    {
        perror("in printlong:lstat");
        return;
    }
    switch(buf.st_mode & S_IFMT)
    {
        case S_IFDIR: printf("d");break;
        case S_IFLNK: printf("l");break;
        case S_IFREG: printf("-");break;
        case S_IFBLK: printf("b");break;
        case S_IFCHR: printf("c");break;
        case S_IFSOCK: printf("s");break;
        case S_IFIFO: printf("p");break;
        default:printf("?");
    }
    putchar((buf.st_mode & S_IRUSR) ?'r':'-');
    putchar((buf.st_mode & S_IWUSR) ?'w':'-');
    if(buf.st_mode & S_ISUID)
        putchar((buf.st_mode & S_IXUSR) ?'s':'S');
    else
        putchar((buf.st_mode & S_IXUSR) ?'x':'-');
    putchar((buf.st_mode & S_IRGRP) ?'r':'-');
```

```
73.     putchar((buf.st_mode & S_IWGRP) ? 'w' : '-');
74.     if(buf.st_mode & S_ISGID)
75.         putchar((buf.st_mode & S_IXGRP) ? 's' : 'S');
76.     else
77.         putchar((buf.st_mode & S_IXGRP) ? 'x' : '-');
78.     putchar((buf.st_mode & S_IROTH) ? 'r' : '-');
79.     putchar((buf.st_mode & S_IWOTH) ? 'w' : '-');
80.     if(buf.st_mode & S_ISVTX)
81.         putchar((buf.st_mode & S_IXOTH) ? 't' : 'T');
82.     else
83.         putchar((buf.st_mode & S_IXOTH) ? 'x' : '-');
84.     printf(" %u ",buf.st_nlink);
85.     user=getpwuid(buf.st_uid);
86.     printf(" %s ",user->pw_name);
87.     grp=getgrgid(buf.st_gid);
88.     printf(" %s ",grp->gr_name);
89.     print_size(&buf);
90.     print_date(&buf);
91.     if((buf.st_mode & S_IFMT) == S_IFLNK)
92.     {
93.         rt=readlink(name,linkname,sizeof(linkname));
94.         linkname[rt]=0;
95.         printf(" %s-> %s ",name,linkname);
96.     }
97.     else
98.         printf("%s",name);
99.     printf("\n");
100. }
101. int checkfiletype(char *name)
102. {
103.     struct stat buf;
104.     int typeflag;
105.     lstat(name,&buf);
106.     switch(buf.st_mode & S_IFMT)
107.     {
108.         case S_IFREG: typeflag=1;break;
109.         case S_IFDIR: typeflag=2;break;
110.         case S_IFLNK: typeflag=3;break;
111.         case S_IFCHR: typeflag=4;break;
112.         case S_IFBLK: typeflag=5;break;
113.         case S_IFSOCK: typeflag=6;break;
114.         case S_IFIFO: typeflag=7;break;
115.         default:typeflag=0;
116.     }
117.     return typeflag;
118. }
119. main(int argc, char *argv[])
120. {
```

```
121.      int rt=-1;
122.      if(argc! =3)
123.      {
124.          printf("Usage: %s -l filename\n",argv[0]);
125.          exit(1);
126.      }
127.      if(strcmp(argv[1],"-l")! =0)
128.      {
129.          printf("Usage: %s -l filename\n",argv[0]);
130.          exit(1);
131.      }
132.      rt=checkfiletype(argv[2]);
133.      switch(rt)
134.      {
135.          case 0:printf("unknown file type\n");exit(1);
136.          case 1:
137.          case 3:
138.          case 4:
139.          case 5:
140.          case 6:
141.          case 7:printlong(argv[2]);return;
142.      }
143. }
```

5.6.3 项目编译与运行

(1)编译。

```
#gcc  -o  myll  myll.c
```

(2)运行。

```
#./myll  -l  文件名(除目录外)
```

(3)可能的运行结果如图 5.2 所示。

```
[root@localhost ch05]# ./myll  /etc/passwd
Usage:./myll  -l  filename

[root@localhost ch05]# ./myll -l /etc/passwd
-rw-r--r-- 1  root  root  1929  Jul  8  2011 /etc/passwd

[root@localhost ch05]# ./myll -l /dev/cdrom
lrwxrwxrwx 1  root  root  3  Nov  1 10:28  /dev/cdrom->hdc

[root@localhost ch05]# ./myll -l /dev/ttyS0
crw-rw---- 1  root  uucp  4,64  Nov 14 10:48 /dev/ttyS0

[root@localhost ch05]# ./myll -l /dev/sda1
brw-r----- 1  root  disk  8,1  Nov  1 10:28 /dev/sda1

[root@localhost ch05]# ./myll -l /dev/initctl
prw------- 1  root  root  0  Nov  1 10:30 /dev/initctl

[root@localhost ch05]# ./myll -l /dev/log
srw-rw-rw- 1  root  root  0  Nov  1 10:29 /dev/log
```

图 5.2 运行结果

在程序 myll.c 中，通过 touch 命令创建一个 f1 文件，然后调用 stat()获取该文件的属性信息并显示。对于文件的索引节点编号 st_ino、链接数 st_nlink、文件主的用户 ID 号 st_uid、文件主组 ID 号 st_gid、文件块个数 st_blocks、块大小 st_blksize 等数值型属性，直接通过 printf()将结果输出。

文件的大小通过函数 print_size()输出。该函数对文件的类型进行判断，如果是字符设备文件 IFCHR 或者块设备文件 S_IFBLK，由于其长度为 0，则输出 st_rdev 中的设备号。如果不是设备文件，则输出文件的长度字段 st_size。

文件的类型在本程序中只做了简单的判断和输出。如果是目录文件 S_IFDIR 则输出“directory”；如果是符号链接文件 S_IFLNK 则输出“symbolic link”；如果是常规文件 S_IFREG 则输出“regular file”；其他类型的文件都输出“other type”。

文件的访问权限根据判断的结果按照“rwxrwxrwx”的格式输出。

[习题]

一、选择题

1. 可以使用(　　)系统调用获得符号链接所引用文件名称。

A. link　　B. symlink　　C. readlink　　D. softlink

2. 获得工作路径名称的系统调用是(　　)。

A. getcwd　　B. getpwuid　　C. getgrgid　　D. getlogin

3. 通过文件属性中的 uid 获得文件拥有者名字的系统调用是(　　)。

A. getcwd　　B. getpwuid　　C. getgrgid　　D. getlogin

4. 通过文件属性中的 gid 获得文件所属组名的系统调用是(　　)。

A. getcwd　　B. getpwuid　　C. getgrgid　　D. getlogin

5. 根据文件路径来改变文件权限使用的系统调用是(　　)。

A. chown　　B. chmod　　C. fchmod　　D. fchown

二、填空题

1. 使用系统调用_____可以设置和得到文件模式的屏蔽字。

2. 创建硬链接使用系统调用_____，创建符号链接使用系统调用_____。

3. 获得工作路径名称的系统调用是_____。

4. 可以使用_____系统调用获取文件属性信息。

5. chmod、chown、utime 都可以修改文件 i 节点的信息，其中 chmod 的功能是_____，chown 的功能是_____，utime 的功能是_____。

6. 若实现将标准输出重定向到文件描述符为 6 对应的文件上，则应使用语句_____。

三、判断并解释原因

1. 一个文件的硬链接中，第一个创建的硬链接与其他硬链接相比总是最后一个被删除。

2. 可以对普通文件和目录文件创建硬链接和符号链接。
3. 给一个文件创建硬链接时，如果新的链接文件已经存在，则覆盖之。
4. 一个符号链接不能再引用另一个符号链接。
5. lstat 系统调用可以获得某符号链接所引用的文件的 i 节点信息。

四、简答题

1. 回答 stat、fstat、lstat 三个系统调用的区别。
2. dup 和 dup2 有哪些区别和联系？
3. 回答硬链接和符号链接的区别。

五、编程题

1. 编写程序 pro3.c，将字符串“hello world”通过输出重定向方式写入文件 f1 中。
2. 实现“ls -l 文件名”功能。