第 5 章

CHAPTER 5

文件I/O编程

学习 Linux 环境高级编程，首先学习的是文件的操作。因为有一句很有趣的话“Linux 下一切皆文件”。所以掌握了文件操作的方法，也就算摸到了门路。

5.1 文件和目录

首先直观地感受一下，在终端下输入命令 ls -l，如图 5-1 所示。

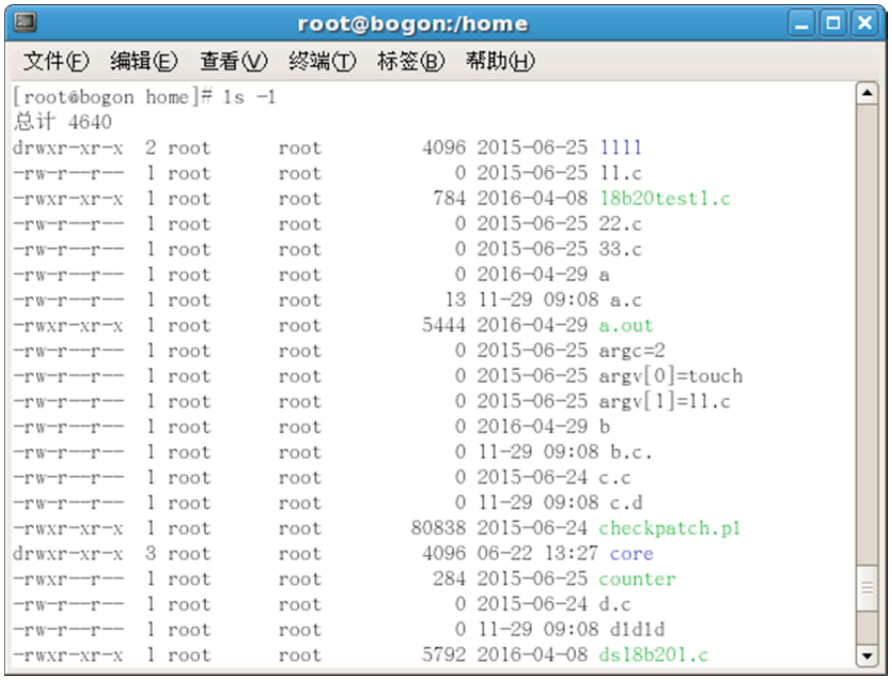

图 5-1 文件目录

图 5-1 的前 2 行为

```
drwxr-xr-x    2 root          root          4096 2015-06-25 1111
-rw-r--r--    1 root          root             0 2015-06-25 11.c
```

1．drwxr-xr-x

drwxr-xr-x 代表的是文件类型和文件权限。常用的文件类型有：

（1）-：普通文件，存各种数据。

（2）d：目录文件，存结构体，结构体内部标识这个目录中的文件名称等信息。

（3）l：链接文件，需要注意的是，软链接才是文件，而硬链接仅仅是一节点。

（4）c：字符设备，除了块设备都是字符设备，没有扇区的概念。

（5）b：块设备，所有存储类的驱动都称为块设备，包含扇区处理。

（6）p：管道设备，是用内核内存模拟的通道。

从上述说明可以看出，例子中的文件是一个目录文件，原因是第一个符号代表文件类型，d 代表此文件是一个目录文件。

2．文件权限

文件权限有：

（1）r 为读，二进制权重为 100，即 4。

（2）w 为写，二进制权重为 010，即 2。

（3）x 为执行，二进制权重为 001，即 1。

（4）-为无操作，二进制权重为 0。

（5）rwx 的顺序不可改，表示可读可写可执行。

（6）-wx 表示不能读，可写可执行。

上述就是文件权限的表示方法，文件权限是用八进制来表达的，如果一个文件有全部的权限，那么对应八进制里的数是 7（4+2+1）。同时读者会发现有多组 rwx，它所表达的不仅仅是它自身的权限。这里涉及一个分组的概念。

（1）u 组：创建者（user）；

（2）g 组：创建者所在组的成员（group）；

（3）o 组：其他人所具备的权限（other）。

也就是说，例子中的三组 rwx 都是依照上述顺序来说明权限的。例子里的文件权限就是：创建者可读可写可执行，所在组的成员可读可执行，其他成员可读不可写不可执行。

3．2

图中文件类型和权限之后是数字 2，这个 2 表示的是文件节点数，也就是说，此文件是一个目录文件。所以，目录的节点数代表该目录下的文件个数，在这里应该是有两个文件。如果此文件不是目录，只是普通文件，那么这个数字就代表硬链接的个数。关于链接的几点说明如下：

（1）链接分为硬链接和软链接（符号链接，即快捷方式）。

（2）硬链接，只是增加一个引用计数，本质上并没有物理上的增加文件。硬链接不是文件。

（3）符号链接，是在磁盘上产生一个文件，这个文件内部写入了一个指向被链接的文件的指针。

（4）采用 ln 指令，用来在文件之间创建链接，默认为创建硬链接（目录不能创建硬链接），使用选项-s 创建符号链接。硬链接指向文件本身，符号链接指向文件名称。

（5）Linux 里寻找文件的顺序是，根据文件名，找到 inode 编号，根据编号找到 inode 块，然后根据 inode 块中的属性信息找到数据块（即文件内容）。

（6）符号链接、硬链接、Windows 快捷方式都具有指向功能；但它们的区别也很明显：Windows 快捷方式指向文件的位置，符号链接是一种文件，创建链接时，系统会为符号链接重新分配一个 inode（节点）编号，但硬链接根本不是一种文件，只是一种指向。

（7）创建硬链接只是增加一个引用计数，硬链接和它的源文件共享一个 inode。

例如：

```
ln file0 file1              //为文件 file0 创建硬链接 file1
ln -s file1 file2           //为文件 file2 创建软链接,其中 file1 为刚创建的硬链接(即 file0 本身)
```

4．目录文件

工作目录是进入系统后所在的当前目录。“.”表示当前目录（工作目录），“..”表示上一级目录（父目录）。

用户目录是每创建一个用户时，就会分配的一个目录，用户名对应的目录就是用户目录，每个用户都有一个自己的主目录，主目录用“~”表示。

路径是从树形目录的某个目录层次到某个文件的一种通路。例如：

```
../../mnt/hgfs/project/linux
```

路径分为 2 种：

（1）相对路径：在工作目录下找到的一个文件路径（通路），随工作目录变化而变化。

（2）绝对路径：从根目录/开始，只有一条路径。

目录是一种特殊的文件，在该文件中存放了多个结构体数据，用来代表目录内的子目录、文件等信息。其结构如下：

```
struct direct{
        ino_t    d_ino;                    //目录文件的节点编号
        off_t    d_off;                    //目录文件开始到目录进入点的位移
        unsigned short d_reclen;           //d_name 的长度(字符串长)
        unsigned char    d_type;           //d_name 的类型
        char     d_name[256];              //文件或目录名
};
```

在前面的学习中介绍过，C 语言本身有自己的函数库，如果需要实现某个功能，包含头文件后直接调用就好。那么在操作系统中，依然会给用户提供一些功能接口 API。用户要实现某些功能必须要依赖这些 API 以及一些机制。

5.2 目录操作

1．创建目录

表头文件

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

定义函数

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

函数说明

path：目录名

mode：模式，即访问权限，包含如下选项：

（1）S_IRUSR：属主读权限；

（2）S_IWUSR：属主写权限；

（3）S_IXUSR：属主执行权限；

（4）S_IRGRP：属组读权限；

（5）S_IWGRP：属组写权限；

（6）S_IXGRP：属组执行权限；

（7）S_IROTH：其他用户读权限；

（8）S_IWOTH：其他用户写权限；

（9）S_IXOTH：其他用户执行权限。

可以使用 S_IRUSR | S_IWUSR 组合权限。可以直接使用数字，例如八进制数 0777、0666 等。例如：

```
int err=mkdir("./aaa",0777);    //在当前目录下创建一个 aaa 目录
```

返回值　返回 0 表示成功；返回−1 表示失败，如果创建成功，则在创建的目录自动创建两个子目录“.”和“..”。

示例 5.2-1　创建目录。

```
#include <stdio.h>
#include <sys/stat.h>
#include <errno.h>                          //系统错误
int main(int argc,char **argv){
      int err=mkdir(argv[1],0666);          //0666
      if(err==-1){
            printf("----- mkdir err no=%d,str=%s\n",errno,strerror(errno));
          //errno 错误编号,是系统全局,运行函数时,系统将运行的错误号写入 errno 中
          //strerror()将错误号对应的说明取出
      }
      return 0;
}
```

运行结果如下：

```
[root@localhost chapter4]# ls
mkdir1.c
[root@localhost chapter4]# gcc mkdir1.c -o mkdir1
[root@localhost chapter4]# ./mkdir1 newdir
[root@localhost chapter4]# ls
mkdir1    mkdir1.c    newdir
[root@localhost chapter4]#
```

文件或目录创建后，查看权限可能不是自己通过函数设定的权限。原因是文件或目录的权限不能超过系统设定的最大权限。对于文件和目录创建之后的 file mode 权限，按照如下方法计算：

文件创建权限为

```
PERM_MAX_FILE & (mode)
```

目录创建权限为

```
PERM_MAX_DIR & (mode)
```

其中，mode 就是创建文件或目录时输入的参数。

而最大权限的计算方法如下：

```
PERM_MAX_FILE = 0666 & ~(umask) //umask 是权限掩码,为系统内建 umask 的设定值
PERM_MAX_DIR = 0777 & ~(umask)
```

例如：

```
umask 0022
PERM_MAX_FILE = 0666 & ~(umask)    = 0644
PERM_MAX_DIR = 0777 & ~(umask) = 0755
```

所以在创建文件时，指定的权限不能超过 MAX。

2．删除目录

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

说明：只能删除空目录。

3．获取当前目录及执行目录

（1）获取当前目录，即执行程序时所在的目录。

表头文件

```
#include <unistd.h>
```

定义函数

```
char *getcwd(char *buf,size_t size);
```

函数说明

buf：保存当前目录的内存地址。

size：为内存的大小。

返回值　成功则返回获取的目录，失败则返回 NULL。

示例 5.2-2　获取当前目录。

```
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>
int main(int argc,char **argv){
        char dir[256];
```

```
        getcwd(dir,256);
        printf("---- %s\n",dir);
        return 0;
    }
```

运行结果如下：

```
[root@bogon chapter4]# gcc example3.c   -o example3
[root@bogon chapter4]# ./example3
---- /home/chapter4
[root@bogon chapter4]#
```

（2）获取程序运行路径，即应用程序存放的位置。

表头文件

```
#include <unistd.h>
```

定义函数

```
int   readlink(const   char *path,   char *buf, size_t   bufsiz);
```

函数说明

path：符号链接，在 Linux 内执行程序都用"/proc/self/exe"符号链接。

buf：用来写入正在执行的文件名（包含绝对路径）的内存。

bufsiz：指明 buf 的大小。

返回值　返回实际文件名的长度，返回−1 表示失败。

示例 5.2-3　获取当前文件或者目录的路径。

```
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>
int main(int argc,char **argv){
//获取程序目录,执行时在其他目录执行
        char dir[256]={0};
        int err=readlink("/proc/self/exe",dir,256);
        if (err==−1) return −1;
        int i;
        for(i=strlen(dir)−1;i>=0 && dir[i]！='/';i--);
        dir[i]=0;
        //打开程序所在目录中的文件
        char filename[256];
        sprintf(filename,"%s/%s",dir,argv[1]);
        printf("---- %s\n",filename);
        FILE *fp=fopen(filename,"r");
        if (fp==NUL=){
            printf("---- %s\n",strerror(errno));
        }
        return 0;
    }
```

运行结果如下：

```
[root@bogon chapter4]# gcc example2.c    -o example2
[root@bogon chapter4]# ./example2    newdir
---- /home/chapter4/newdir
[root@bogon chapter4]#
```

4．获取目录或文件的状态

表头文件

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

定义函数

（1）读取指定文件或目录的状态信息。

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

（2）读取已打开的文件状态信息。

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

函数说明

path：路径或文件名。

filedes：已打开文件或目录的句柄。

buf：是 struct stat 结构的指针，其结构格式如下：

```
struct stat{
      unsigned short st_mode;          //文件保护模式(即文件类型)
      unsigned short st_nlink;         //硬链接引用数
      unsigned short st_uid;           //文件的用户标识
      unsigned short st_gid;           //文件的组标识
      unsigned long   st_size;         //文件大小
      unsigned long   st_atime;        //文件最后的访问时间
      unsigned long   st_atime_nsec;   //文件最后的访问时间的秒数的小数
      unsigned long   st_mtime;        //文件最后的修改时间
      unsigned long   st_mtime_nsec;
      unsigned long   st_ctime;        //文件最后状态的改变时间
      unsigned long   st_ctime_nsec;
              ...
    };
```

返回值　返回获取文件状态的信息。

（3）判断文件类型及访问权限。

方法一：在 struct stat 结构中，由 st_mode 字段记录了文件类型及访问权限，操作系统提供了一系列的宏来判断文件类型。如下：

```
S_ISREG(mode)          //判断是否为普通文件
S_ISDIR(mode)          //判断是否为目录文件
S_ISCHR(mode)          //判断是否为字符设备文件
S_ISBLK(mode)          //判断是否为块设备文件
```

```
S_ISFIFO(mode)          //判断是否为管道设备文件
S_ISLNK(mode)           //判断是否为符号链接
```

示例 5.2-4

```
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
int main(int argc,char **argv){
        struct stat st;
        int err= stat(argv[1],&st);
        if (err==-1) return -1;
        if (S_ISDIR(st.st_mode)) printf("isdir\n");
        else if (S_ISREG(st.st_mode)) printf("file\n");
}
```

运行结果如下：

```
[root@bogon chapter4]# gcc example4.c -o example4
[root@bogon chapter4]# ./example4
[root@bogon chapter4]# ./example4 newdir/
isdir
[root@bogon chapter4]# ./example4 example3.c
file
[root@bogon chapter4]#
```

方法二：也可以直接读取 st_mode 内的数据，不过 st_mode 内的数据是组合而成的数据，包括很多信息，需要进行“与”运算才能取出这个字段中的数据。st_mode 是用特征位来表示文件类型的，特征位的定义如下：

S_IFMT	0170000	文件类型的位遮罩
S_IFSOCK	0140000	socket
S_IFLNK	0120000	符号链接（symbolic link）
S_IFREG	0100000	一般文件
S_IFBLK	0060000	区块装置（block device）
S_IFDIR	0040000	目录
S_IFCHR	0020000	字符装置（character device）
S_IFIFO	0010000	先进先出（FIFO）
S_ISUID	0004000	文件的 set user-id on execution 位
S_ISGID	0002000	文件的 set group-id on execution 位
S_ISVTX	0001000	文件的 sticky 位
S_IRWXU	00700	文件所有者的遮罩值（即所有权限值）
S_IRUSR	00400	文件所有者具有可读取权限
S_IWUSR	00200	文件所有者具有可写入权限
S_IXUSR	00100	文件所有者具有可执行权限

S_IRWXG	00070	用户组的遮罩值（即所有权限值）
S_IRGRP	00040	用户组具有可读取权限
S_IWGRP	00020	用户组具有可写入权限
S_IXGRP	00010	用户组具有可执行权限
S_IRWXO	00007	其他用户的遮罩值（即所有权限值）
S_IROTH	00004	其他用户具有可读取权限
S_IWOTH	00002	其他用户具有可写入权限
S_IXOTH	00001	其他用户具有可执行权限

示例 5.2-5

```
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
int main(int argc,char **argv){
      struct stat st;
      int err= stat(argv[1],&st);
        if (err==−1) return -1;
                    if (st.st_mode & S_IFREG) printf("file\n");
                    else if (st.st_mode & S_IFDIR) printf("isdir\n");
        }
```

运行结果：

```
[root@bogon chapter4]# gcc example5.c -o example5
[root@bogon chapter4]# ./example5 newdir/
isdir
[root@bogon chapter4]# ./example5 example4.c
file
[root@bogon chapter4]#
```

5．目录的读写操作（标准 C 语言内的函数）

表头文件

```
#include <sys/types.h>                  //对外提供的各种数据类型,例如 size_t
#include <sys/stat.h>                   //对外提供的各种结构类型,例如 time_t
typedef int ino_t;
struct direct{
            ino_t    d_ino;             //目录文件的节点编号
            off_t    d_off;             //目录文件开始到目录进入点的位移
            unsigned short d_reclen;    //d_name 的长度(字符串长)
            unsigned char    d_type;    //d_name 的类型
            char     d_name[256];       //文件或目录名
      };
      #include <stdio.h>
```

定义函数

（1）打开目录。

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

参数　path 为目录名。

返回值　返回 DIR 类型指针(文件为 FILE *)。

（2）关闭已打开的目录。

```
int closedir(DIR *dp);
```

（3）读取目录。

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

参数　dp 为目录句柄，返回值是 opendir。

返回值　返回目录结构地址，该结构内存储指定目录下的文件信息，每读取一次则读取一个节点。目录文件本身是只读的，不可以写，如果创建目录，则用 mkdir 函数，删除目录用 rmdir 函数。

示例 5.2-6　打印目录下的所有文件及目录。

```
#include <unistd.h>
#include <stdio.h>
#include <dirent.h>
#include <string.h>
#include <sys/stat.h>
#include <stdlib.h>
void printdir(char *dir, int depth)
{
        DIR *dp;
        struct dirent *entry;
        struct stat statbuf;
        if ((dp = opendir(dir)) == NULL) {
        fprintf(stderr, "Can`t open directory %s\n", dir);
        return ;
        }
        chdir(dir);
        while ((entry = readdir(dp)) != NULL) {
        lstat(entry->d_name, &statbuf);
        if (S_ISDIR(statbuf.st_mode)) {
           if (strcmp(entry->d_name, ".") == 0 ||
              strcmp(entry->d_name, "..") == 0 )
              continue;
           printf("%*s%s/\n", depth, "", entry->d_name);
           printdir(entry->d_name, depth+4);
        } else
           printf("%*s%s\n", depth, "", entry->d_name);
        }
        chdir("..");
        closedir(dp);
```

```
}
int main(int argc, char *argv[])
{
        char *topdir = ".";
        if (argc >= 2)
        topdir = argv[1];
        printf("Directory scan of %s\n", topdir);
        printdir(topdir, 0);
        printf("done.\n");
        exit(0);
}
```

运行结果如下：

```
[root@bogon chapter4]# gcc example6.c -o example6
[root@bogon chapter4]# ./example6
Directory scan of .
example2
example4.c
example2.c
example3.c
newdir/
example3
example5.c
example4
example6.c
example1.c
mkdir1
a.out
example5
example6
done.
```

5.3 文件操作

5.3.1 基本概念

1．流

流指数据的永久性存储，主要指数据以文件为单位存储在磁盘上。Linux 以字节为单位操作数据，所有的数据都是 0 或 1 的序列，如果需要让人读懂的数据，则以字符的方式显示出来，这就是所谓的文本文件。

而数据不是存在磁盘上后就永远不动，往往要被读入到内存、传送到外部设备或搬移到其他位置，所以数据不断地在流动。然而不同设备之间的连接方法差异很大，数据读取和写入的方式也不相同，所以 Linux 定义了流（stream），建立了一个统一的接口，无论数据是从内存到外设，还是从内存到文件，都使用同一个数据输入输出接口。

2．文件流和标准流

文件操作有两种方法：原始文件 I/O 和标准 I/O。

1）标准 I/O

标准 I/O 是标准 C 的输入输出库，fopen、fread、fwrite、fclose 都是标准 C 的输入输出函数。标准 I/O 都使用 FILE * 流对象指针作为操作文件的唯一识别，所以标准 I/O 是针对流对象的操作，是带缓存（内存）机制的输入输出。标准 I/O 又提供了 3 种不同方式的缓冲：

（1）全缓冲。即缓冲区被写满或是调用 fflush 后，数据才会被写入磁盘。

（2）行缓冲。即缓冲区被写满或是遇到换行符时，才会进行实际的 I/O 操作。当流涉及一个终端时（标准输入和标准输出），通常使用行缓冲。

（3）不缓冲。标准 I/O 库不对字符进行缓存处理。标准出错流 stderr 往往是不带缓存的，使得出错信息可以尽快显示出来。

2）原始 I/O

原始 I/O 又称文件 I/O，是 Linux 操作系统提供的 API，称为系统调用，是针对描述符（即一个编号）操作的，是无缓存机制。

3）文件描述符

创建一个新的文件或打开已有文件时，内核向进程返回一个非负整数（即编号），用来识别操作的是哪一个文件。对于 Linux 而言，所有打开的文件都是通过文件描述符引用的，在操作系统内部有一个宏定义了描述符的最大取值 OPENMAX，不同版本的 Linux 的取值不同。

Linux 编程使用的 open、close、read、write 等文件 I/O 函数属于系统调用的，其实现方式是用了 fctrl、ioctrl 等一些底层操作的函数。而标准 I/O 库中提供的是 fopen、fclose、fread、fwrite 等面向流对象的 I/O 函数，这些函数在实现时本身就要调用 Linux 的文件 I/O。在应用上，文件读写时二者并没区别，但是一些特殊文件，例如管道等只能使用文件 I/O 操作。

3．标准输入、标准输出和标准错误

当 Linux 执行一个程序时，会自动打开三个流：标准输入（standard input）、标准输出（standard output）和标准错误（standard error）。命令行的标准输入连接到键盘，标准输出和标准错误都连接到屏幕。对于一个程序来说，尽管它总会打开这三个流，但它会根据需要使用，并不是一定要使用。系统默认打开的三个文件描述符（为进程预定义的三个流）如下：

```
STDOUT_FILENO   0  标准输入,用于从键盘获取数据
STDIN_FILENO    1  标准输出,用于向屏幕输出数据
STDERR_FILENO   2  标准错误,用于获取错误信息
```

例如，使用标准输入输出：

```
char buf[256]="akjfkaskfdasdf";
read(STDIN_FILENO,buf,256);                 //相当于 scanf
write(STDOUT_FILENO,buf,strlen(buf));       //相当于 printf
```

4．常用设备

/dev/null　空设备，用来丢弃数据。

/dev/port　存取 I/O 的端口设备。

/dev/ttyN　N（0...） 字符终端设备。

/dev/sdaN　N（1...） SCSI 磁盘设备。

/dev/scdN　N（1...） SCSI 光驱设备。

/dev/fbN　N（0...） 帧缓冲设备（frame buffer），用于屏幕输出，多媒体操作必须使用这个设备。

/dev/mixer 混音器设备，用于调整音量的大小、各种音频的叠加，多媒体操作经常使用。

/dev/dsp 声卡数字采样和数字录音设备，用于播放声音和录音，多媒体操作必须使用。

/dev/audio 声卡音频设备，用于播放声音和录音，支持 sun 音频，较少使用。

/dev/video 视频设备，用于摄像头的视频采样（录像、照相）。

5．常用的头文件

```
#include <sys/types.h>          //操作系统对外供的各种数据类型的定义,例如 size_t
#include <sys/stat.h>           //操作系统对外提供的各种结构类型的定义,例如 time_t
#include <sys/ioctl.h>          //设备的控制函数定义
#include <sys/soundcard.h>      //声卡的结构及定义
#include <errno.h>              //对外提供的各种错误号的定义,用数字代表错误类型
#include <fcntl.h>              //文件控制的函数定义
#include <termios.h>            //串口的结构和定义
#include <unistd.h>             //C++标准库头文件
#include <sys/types.h>          //文件及设备操作函数
#include <sys/stat.h>           //文件及设备操作函数
#include <fcntl.h>              //文件及设备操作函数
```

5.3.2　检查文件及确定文件的权限

```
#include <unistd.h>
int access(const char *pathname, int mode);
```

参数　pathname 包含路径的文件名。

mode 为模式，有 5 种模式：

（1）0，检查文件是否存在；

（2）1，检查文件是否可执行 x；

（3）2，检查文件是否可写 w；

（4）4，检查文件是否可读 r；

（5）6，检查文件是否可读写 w+r。

返回值　0 为真，非零为假，无论是否为真，这个函数都会向标准错误发送信号，指明执行情况。

Linux 错误处理采用一个全局变量 errno，用来存储函数执行后的错误编码。每个错误编码都对应了一个解释，即错误提示内容。获取错误提示使用如下函数：

```
#include <errno.h>
char *strerror(int errno);              //根据错误号获取字符串
//errno 为 0 时代表成功,为非零时代表错误,由 strerror 获取提示
```

示例 5.3.2-1　判断文件是否存在。

```
//直接打开不存在的文件,看错误提示
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>
        int main(int argc,char *argv[])
{
        FILE *fp=fopen(argv[1],"r");
 printf("errno =%d,str=%s\n",errno,strerror(errno));
       //判断文件是否存在,然后决定是否打开文件,同时看错误提示
        int err=access(argv[1],0);
        printf("err=%d,errstr=%s\n",errno,strerror(errno));
        if (err==0){ //打开文件操作
                        }
       else {//创建文件操作
                        }
       }
```

将该代码文件命名为 Example5.3.2-1，编译，运行结果如下：

```
[root@bogon chapter5]# ./Example5.3.2-1 example1.c
errno =0,str=Success
err=0,errstr=Success
```

5.3.3　创建文件

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

参数　pathname 包含路径的文件名。

mode 为模式，即访问权限，包含如下选项：

（1）S_IRUSR，属主读权限；

（2）S_IWUSR，属主写权限；

（3）S_IXUSR，属主执行权限。

返回值　成功时返回文件描述符，失败时返回−1。

示例 5.3.3-1　创建文件。

```
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
int main(int argc,char **argv){
       int fd=−1;
       fd=creat(argv[1],00777);
       if (fd<0){
              printf("errno：  %s\n",strerror(errno));
       }
```

```
        else {
                printf("ok ：  %d\n",fd);
                close(fd);
        }
        return 0;
}
```

将该代码文件命名为 Example5.3.3-1，编译并运行后，当前目录多了一个 hello.c 的文件。

```
[root@bogon chapter5]# ./Example5.3.3-1 hello.c
ok ：3
```

5.3.4 打开文件

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

参数 pathname 包含路径的文件名。

flags 表示文件打开方式，有如下选项：

（1）O_RDONLY，只读方式打开文件；

（2）O_WRONLY，只写方式打开文件；

（3）O_RDWR，读写方式打开文件；

（4）O_APPEND，追加模式打开文件，在写以前，文件读写指针被置于文件的末尾；

（5）O_CREAT，创建文件，若文件不存在将创建一个新文件；

（6）O_EXCL，如果通过 O_CREAT 打开文件，若文件已存在，则 open 调用失败，用于防止重复创建文件；

（7）O_TRUNC，如果文件已存在，且又是以写方式打开，则将文件清空。使用 O_CREAT 和 O_TRUNC 时，如果对设备操作，报错，且防止对设备进行创建；

（8）O_SYNC，实现 I/O 的同步，任何通过文件描述符对文件的写操作都会使调用的进程中断，直到数据被真正写入硬件中；

（9）O_NONBLOCK，非阻塞模式（非块方式)打开，当打开文件不满足于条件时，则一直等待，函数不能马上返回，直到满足条件时才打开结束。

注：以上所有选项可以组合使用。

（1）O_CREAT|O_RDWR，如果文件不存在，则创建，否则以读写方式打开文件；

（2）O_CREAT|O_EXCL，如果文件已存在，则调用失败，用来实现互斥；

（3）O_RDONLY、O_WRONLY、O_RDWR 只能选择一个；

（4）O_CREAT | O_WRONLY|O_TRUNC 组合，则与 creat 函数等价。

---

mode 模式，即访问权限，包含如下选项：

（1）S_IRUSR，属主读权限；

（2）S_IWUSR，属主写权限；

（3）S_IXUSR，属主执行权限。

返回值　返回 0 表示正确，返回非零时表示失败。

示例 5.3.4-1　打开文件。

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>
#include <fcntl.h>
int main(int argc,char *argv[])
{
int fd;
        int err=access(argv[1],0);
        printf("err=%d,errstr=%s\n",errno,strerror(errno));
        if (err==0){
              //打开文件操作
              fd=open(argv[1],O_RDWR|O_SYNC, 0666);
        }
        else {
              //创建文件操作
              fd=creat(argv[1],0666);
        }
        if (fd==-1) {
            printf("open or creat    err \n");
            return 0;
        }
}
```

将代码文件命名为 Example5.3.4-1，编译，运行结果如下：

```
[root@bogon chapter5]# ./ Example5.3.4-1    example1.c
err=0,errstr=Success
```

5.3.5　关闭文件

```
void close(int filedes);
```

参数　filedes 为文件描述符。

示例 5.3.5-1　当文件存在时，读取数据，否则创建文件并写入数据。

```
#include<stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>
#include <fcntl.h>
int main(int argc,char **argv)
        {
            int fd,err=access(argv[1],0);
```

```
        if (!err){
                fd=open(argv[1],O_RDONLY);
                char buf[4096];
                int size=read(fd,buf,200);
                printf("--fd=%d,size=%d,buf=%s\n",fd,size,buf);
                close(fd);
                        }
            else {
                fd=creat(argv[1],0777);
                char buf[256];
                scanf("%s",buf);
                int size=write(fd,buf,strlen(buf));
                close(fd);
                        }
        }
```

将该代码文件命名为 Example5.3.5-1，编译运行后，example1.c 文件被关闭，同时 example1.c 的内容也显示出来，运行结果的后 6 行为 example1.c 的内容。

```
[root@bogon chapter5]# ./ Example5.3.5-1      example1.c
--fd=3,size=200,buf=#include <stdio.h>
#include <errno.h>
int main(int argc,char **argv){
      int err=mkdir(argv[1],0666); //0666
      if (err==-1){
           printf("----- mkdir err no=%d,str=%s\n",errno,st
```

5.3.6 删除文件

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

参数 pathname 为硬链接的各种名。

本质是解决硬链接，也就是删除节点数。

说明：只是将文件的引用计数减 1，如果引用计数为 0，则删除物理文件。

示例 5.3.6-1 实现 rm -rf 名。

要点解析：删除函数时，要判断是否是目录，如果是目录则排除 .和..，并递归调函数，然后删目录；否则删文件。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <dirent.h>
int i,isr=0,isf=0,isv=0;
void startdel(char *name,int isv)
      {
```

```
        struct stat st;
        stat(name,&st);
        if (S_ISDIR(st.st_mode))
          { //如果是目录，则递归调用
          DIR *fp=opendir(name);
          if (fp!=NULL)
              {
              struct dirent* p;
              char filename[256];
              while((p=readdir(fp))!=NULL)
                {
                if (strcmp(p->d_name,".")==0||strcmp(p->d_name,"..")==0)
                continue;
                sprintf(filename,"%s/%s",name,p->d_name);
                startdel(filename,isv);
                }
              }
          closedir(fp);
          rmdir(name); //删目录
          }
        else
          unlink(name); //删文件
          if (isv) printf("del %s\n",name);
        }
        char *isenbledel(char *name){
        static char filename[256];
        memset(filename,0,256);
        //--判断文件名是否完整，如果不完整，则补全
        if (name==NULL || strcmp(name,"")==0)
          return NULL;
        else if (name[0]!='.')
              sprintf(filename,"./%s",name);
        else if (name[1]!='/')
              {
              if (name[2]=='/')
                sprintf(filename,"%s",name);
              else
                sprintf(filename,"./%s",name);
              }
        else
          sprintf(filename,"%s",name);
        return filename;
    }
    int main(int argc,char **argv)
        {
        char **ls=NULL;
        for(i=1;i<argc;i++)
          {
              if (argv[i][0]=='-')
                {
                isf=strstr(argv[i],"f")?1：0;
```

```
                isr=strstr(argv[i],"r")?1：0;
                isv=strstr(argv[i],"v")?1：0;
                }
            else
                {
                ls=&argv[i];
                break;
                }
        }
    i=0;
    char cmd[256];
    while(ls[i]!=NULL)
        {
            char *pathname=isenbledel(ls[i++]);
            if (pathname==NULL)
                continue;
            if (isf==0)
                {
                printf("delete no[yes]\n");
                scanf("%s",cmd);
            if (strcmp(cmd,"yes")!=0)
                continue;
                }
            if (isr==0)
                {
                struct stat st;
                stat(pathname,&st);
                if (S_ISDIR(st.st_mode))
                    {
                    printf("dir isn't deleted\n");
                    continue;
                    }
                }
        startdel(pathname,isv);
        }
    }
```

将该代码文件命名为 Example5.3.6-1，编译运行后，删除 hello.c 文件，提示用户是否删除，输入 yes，则该文件被删除。

```
[root@bogon chapter5]# ./ Example5.3.6-1    hello.c
delete no[yes]
yes
```

5.3.7 文件指针移动

当文件读写数据时，文件指针会自动移动，也可以通过函数来改变文件指针位置，函数如下：

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

参数　filedes 为描述符。

offset 为偏移量。

whence 表示从哪里开始，有 3 种选项：

（1）SEEK_SET　0，从头开始；

（2）SEEK_CUR　1，当前位置；

（3）SEEK_END　2，从尾开始。

返回值　是移动后当前所在位置，即字节数。

示例 5.3.7-1　用如下两种方法读取 wav 文件头：

（1）结构体方法；

（2）文件指针移动法。

```
struct WAV{
    char riff[4];              //RIFF
    long len;                  //文件大小
    char type[4];              //WAVE
    char fmt[4];               //fmt
    char tmp[4];               //空出的
    short pcm;
    short channel;             //声道数
    long sample;               //采样率
    long rate;                 //传送速率
    short framesize;           //调整数
    short bit;                 //样本位数
    char data[4];              //数据
    long dblen;                //len-sizeof(struct WAV);
};
```

几个基本概念如下所述：

（1）采样：获取音频数据；

（2）采样样本：一帧数据；

（3）采样率：每秒的帧数；

（4）格式：采样位数，即每一帧每一声道所占的内存空间；

（5）声道：双声道、单声道；

（6）t 秒内的字节数：①字节数=t * sample * bit *channel /8；②t=字节数/（sample * bit *channel /8）；③传送速率= sample * bit *channel /8，即一秒中的字节数；④调整数=channel*bit /8，即一帧的字节数。

方法一：

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

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <linux/soundcard.h>
#include <alsa/asoundlib.h>
typedef struct WAV{
        char riff[4];                   //RIFF
        long len;                       //文件大小
        char type[4];                   //WAVE
        char fmt[4];
        char tmp[4];
        short pcm;
        short channel;                  //声道数
        long sample;                    //采样率
        long rate;                      //传送速率
        short framesize;                //调整数
        short bit;                      //样本位数
        char data[4];
          long dblen;                   //len-sizeof(struct WAV);
    }wav_t;
int main(int argc,char **argv){
          //打开文件
          int fd=open(argv[1],O_RDONLY);
          if (fd==-1){
              printf("----- open err=%s\n",strerror(errno));
              return -1;
          }
          //wav_t *p=(wav_t*)malloc(sizeof(wav_t));
          //wav_t a,*p=&a;
          char buf[4096];
          wav_t *p=(wav_t*)buf;
          int size=read(fd,p,sizeof(wav_t));
          close(fd);
          printf("---%s,%d,%d,%d\n",p->type,p->channel,p->sample,p->bit);
          printf("---%d,%d\n",p->len,p->dblen);
    }
```

运行该程序，获取了 audio1.wav 文件数据，具体操作结果如下：

```
[root@bogon chapter5]# ./ Example5.3.7-1    audio1.wav
---WAVEfmt ,2,44100,8
---6303174,291939
```

方法二：

```
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/ioctl.h>
#include <stdlib.h>
```

```
#include <stdio.h>
#include <string.h>
#include <linux/soundcard.h>
#include <alsa/asoundlib.h>

int main(int argc,char **argv){
        //打开文件
        int fd=open(argv[1],O_RDONLY);
        if (fd==-1){
            printf("----- open err=%s\n",strerror(errno));
            return -1;
        }
        char type[5]={0};
        short channel;
        short bit;
        int sample;
        lseek(fd,0x08,0);
        read(fd,type,4);
        lseek(fd,0x16,0);
        read(fd,&channel,2);
        read(fd,&sample,4);
        lseek(fd,0x22,0);
        read(fd,&bit,2);
        close(fd);
        printf("--- %s,%d,%d,%d\n",type,channel,sample,bit);
    }
```

运行该程序，获取了 audio1.wav 文件数据，具体操作结果如下：

```
[root@bogon chapter5]# ./ Example5.3.7-1audio1.wav
--- WAVE,2,44100,8
```

5.3.8　其他常用函数

1．更改系统掩码函数

```
mode_t umask(mode_t mask);
```

2．修改文件权限函数

```
int chmod(const char *path, mode_t mode);          //修改没被打开的文件或目录权限
int fchmod(int fildes, mode_t mode);               //修改已打开的文件权限
```

3．截断函数

修改文件长度以及 length 长度，如果变长，则自动补 0。

```
int truncate(const char *path, off_t length);
int ftruncate(int fd, off_t length);
```

4．创建硬链接

增加节点，对应的是 unlink。

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

5．删除、移除文件或空目录

rmdir 函数用于删空目录，unlink 函数用于删文件。

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

6．重命名函数

```
int rename(const  char  *oldpath,  const char *new-path);
```

示例 5.3.8-1 读取文件内的字符串并排序，将排好序的字符串写回到文件。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/stat.h>
#include <sys/types.h>
int main(int argc,char **argv){
    if(access(argv[1],0)) {
        printf("------ err=%d,errstr=%s\n",errno,strerror(errno));
        return -1;
    }
    int fd=open(argv[1],O_RDWR);
    if (fd<0){
        printf("errno= %s\n",strerror(errno));
        return -1;
    }
    struct stat st;
    stat(argv[1],&st);
    int size=st.st_size;
    //---------------------------
    char *buf=(char *)malloc(size+1);
    bzero(buf,size+1);
    size=read(fd,buf,size);
    //---------------------------
    int i,j;
char tmp;
    for(i=0;i<size;i++){
        for(j=i;j<size;j++){
            if (buf[i] >buf[j]){
                tmp=buf[i];
                buf[i]=buf[j];
                buf[j]=tmp;
            }
        }
    }
    //---------------------------
    size=write(fd,buf,size);
    close(fd);
    return 0;
}
```

在当前目录创建 hello.c 文件，内容为“/ahow are you!”，编译运行示例 5.3.8-1，结果如下：

```
[root@bogon chapter5]# ./ Example5.3.6-2    hello.c
```

运行后，hello.c 文件的内容如下：

```
/ahow are you!

   !/aaehooruwy
```

示例 5.3.8-2　有一个结构体数组，已输入数据，将结构体数组内的数据保存到文件，要求用三种方式实现：读整块内存、循环读结构体大小、移位读取每一个成员。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/stat.h>
#include <sys/types.h>
struct STU{
    int num;
    char name[32];
};
int main(int argc,char **argv){
     struct STU stu[5]={
          {23,"asdf"},
          {34,"gadf"},
          {44,"gasdfsaf"},
          {33,"ffffff"},
          {22,"fssssssss"}
     };
     int fd;
     int ishas=!access(argv[1],0);
     if(ishas)
          fd=open(argv[1],O_RDWR);
     else
          fd=open(argv[1],O_CREAT | O_RDWR);
      if (fd<0){
            printf("errno= %s\n",strerror(errno));
            return -1;
             }
     if (ishas){
          int size=lseek(fd,0,2);
          int n=size/sizeof(struct STU);
          lseek(fd,0,0);
          struct STU *p=(struct STU *)malloc(size);
          第一种方式
          size=read(fd,p,size);
          第二种方式
```

```
            for(int i=0;i<n;i++)
            read(fd,p,sizeof(struct STU));
            第三种方式
            void *pv=p;
            for(i=0;i<n;i++) {
            read(fd,pv,4); fd+=4;
            read(fd,pv,32);fd+=32;
            }
            for(int i=0;i<n;i++) printf("%d,%s\n",p->num,p->name);
        }
        else {
            第一种方法
            int size=write(fd,stu,sizeof(struct STU)*5);
            第二种方法
            for(int i=0;i<5;i++) write(fd,&stu[i],sizeof(struct STU));
            第三种方法
            void *pv=p;
            for(i=0;i<sizeof(struct STU)*5;i++) write(fd,pv++,1);
        }
        close(fd);
        return 0;
    }

    [root@bogon chapter5]# ./ Example5.3.8-2    text
```

5.4 设备控制

设备控制，就是在设备驱动中对设备 I/O 进行管理，即对设备的一些特性进行控制，例如串口的传输波特率、马达的转速、混音器的音量、声卡的采样率等。设备控制采用统一的接口函数，然后输入特定的指令及数值。

1．硬件设备控制

硬件设备控制即设置硬件或从硬件中获取信息。函数如下：

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

参数　fd 为设备描述符。

request 是控制指令，通常是一些特定的宏或宏函数。

…通常代表是多个参数，在这里是补充参数，实际上最多只有一个参数。

返回值　返回 0 表示成功，返回非零表示失败。

所需头文件　#include <sys/ioctl.h>

对于音频设备编程，常用的设备是内部的混音器和声卡。设备文件如下：

（1）/dev/mixer 混音器设备，用于调整音量大小以及各种音频的叠加。

（2）/dev/dsp 声卡数字采样和数字录音设备，用于播放声音和录音。

2．控制混音器

对混音器的操作一般通过 ioctl 系统调用来完成，所需头文件为#include <linux/soundcard.h>，常用控制命令如下：

（1）SOUND_MIXER_VOLUME：主音量调节；
（2）SOUND_MIXER_BASS：低音控制；
（3）SOUND_MIXER_TREBLE：高音控制；
（4）SOUND_MIXER_SYNTH：FM 合成器；
（5）SOUND_MIXER_PCM：主 D/A 转换器；
（6）SOUND_MIXER_SPEAKER PC：喇叭；
（7）SOUND_MIXER_LINE：音频线输入；
（8）SOUND_MIXER_MIC：麦克风输入；
（9）SOUND_MIXER_CD：CD 输入；
（10）SOUND_MIXER_IMIX：放音音量；
（11）SOUND_MIXER_ALTPCM： D/A 转换器；
（12）SOUND_MIXER_RECLEV：录音音量；
（13）SOUND_MIXER_IGAIN：输入增益；
（14）SOUND_MIXER_OGAIN：输出增益；
（15）SOUND_MIXER_LINE1：声卡的第 1 输入；
（16）SOUND_MIXER_LINE2：声卡的第 2 输入；
（17）SOUND_MIXER_LINE3：声卡的第 3 输入。

以上控制指令的取值范围都是 0~100 的数据，按控制输入或输出，采用如下 2 个宏函数：

（1）SOUND_MIXER_READ 宏，读取混音通道的增益大小；
（2）SOUND_MIXER_WRITE 宏，写入混音通道的增益大小。

例如：

```
SOUND_MIXER_READ(SOUND_MIXER_MIC)              //代表从设备中获取麦克风的输入增益
SOUND_MIXER_WRITE(SOUND_MIXER_VOLUME)          //代表设置设备的主音量大小
```

关于增益值的数据保存在 16 位数据中，其中：

（1）对于只有一个混音通道的单声道设备，增益大小保存在低位字节中。

（2）对于支持多个混音通道的双声道设备，增益大小实际上包括两个部分，分别代表左、右两个声道的值，其中低位字节保存左声道的音量，高位字节则保存右声道的音量。

示例 5.4-1　控制音量大小。

```
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <dirent.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <fcntl.h>
#include <sys/soundcard.h>
```

```
int setvol(int lvol,int rvol)
{
int fd=open("/dev/mixer",O_RDWR);
short vol =lvol<<8 | rvol;
int err=ioctl(fd,MIXER_WRITE(SOUND_MIXER_VOLUME),&vol);
close(fd);
}
int getvol(int *lvol,int *rvol)
{
int fd=open("/dev/mixer",O_RDWR);
short vol;
int err=ioctl(fd,MIXER_READ(SOUND_MIXER_VOLUME),&vol);
*lvol=vol >>8;
*rvol=vol & 0xFF;
close(fd);
        }
int main(int argc,char **argv)
{
setvol(atoi(argv[1]),atoi(argv[2]));
        }
```

3．控制采样设备

WAVE 文件作为多媒体中使用的声波文件格式之一，它以 RIFF 格式为标准。每个 WAVE 文件的头四个字节便是“RIFF”。WAVE 文件由文件头和数据体两大部分组成。

在音频文件中，有三个至关重要的数据，即声道、采样频率、采样大小。

采样指以固定的时间间隔对波形的值进行抽取，采样频率，即一秒钟内采样的次数。采样频率越高，声音保真度越好。常用的采样率有：

（1）8000Hz：电话所用采样频率， 对于人的说话已经足够。

（2）22050Hz：无线电广播所用采样频率。

（3）32000Hz：miniDV、数码视频 camcorder、DAT（LP mode）所用采样频率。

（4）44100Hz：音频 CD 的采样频率，也常用于 MPEG-1 音频（VCD，SVCD，MP3）。

（5）47250Hz：PCM 录音机所用采样频率。

（6）48000Hz：miniDV、数字电视、DVD、DAT、电影和专业音频所用的数字声音采样频率。

（7）50000Hz：商用数字录音机所用采样频率。

（8）50400Hz：三菱 X-80 数字录音机所用采样频率。

（9）96000Hz：DVD-Audio、LPCM DVD 音轨、BD-ROM 音轨和 HD-DVD 音轨所用采样频率。

采样大小指一个音频采样数据所占的字节数，目前只有 8 位和 16 位。声道数目通常有单声道、双声道两种。

通过上面的信息可知：

一个采样数据字节数=声道数目×采样大小/8（字节）；

一秒采样数据字节数=采样频率×声道数目×采样大小/8（字节）；

音频数据长度=播放时间×采样频率×声道数目×采样大小/8（字节）；

音频文件长度=文件头长度＋音频数据长度。

对混音器的操作一般都通过 ioctl 系统调用来完成，常用控制命令如下：

```
#include <linux/soundcard.h>
SNDCTL_DSP_SETFMT          //采样大小,也称采样格式,取值有 8 或 16
SNDCTL_DSP_SPEED           //采样频率,取值有 8000、11025、22050、32000、44100、47250、48000、50000
SNDCTL_DSP_STEREO          //立体声,取值 0 或 1
SNDCTL_DSP_CHANNELS        //设置声道数目
SOUND_PCM_READ_RATE        //设置 PCM 转换速度
SNDCTL_DSP_GETBLKSIZE      //设置 DSP 块空间的大小
SNDCTL_DSP_SETFRAGMENT//设置声卡驱动程序中的内核缓冲区的大小(分为读或写)
SNDCTL_DSP_SYNC            //控制读写文件同步
SNDCTL_DSP_GETOSPACE       //获取输出空间
SNDCTL_DSP_GETISPACE       //获取输入空间
SNDCTL_DSP_RESET           //DSP 复位
```

示例 5.4-2　实现 wav 音乐播放。

libao.h 文件代码如下：

```
#ifndef LIBAO_H
#define LIBAO_H

typedef enum {dsp,alsa} ao_type_t;
typedef struct AOINFO{
        ao_type_t type;                         //设备类型编号
        char *name;                             //oss、asla 等
        char *author;                           //作者
        char *time;                             //编写时间
        char *describe;                         //模块描述
} ao_info_t;

typedef struct LIBAO{
    ao_info_t *info;                            //模块信息
    int (*open)(char *devname,int flags);       //打开设备
    int (*read)(char *buf,int size);            //输入数据
    int (*write)(char *buf,int size);           //输出数据
    int (*ioctl)(int cmd,void *data);           //控制设备
    int (*close)();                             //关闭设备
} ao_t;

//extern ao_t *getdev(ao_type_t type);
extern ao_t ao_dsp;
#endif
```

libao.c 文件代码如下：

```
#include <stdlib.h>
#include "libao.h"
//extern ao_t ao_alsa;
extern ao_t ao_dsp;
ao_t *ao_ls[]={
```

```
        //&ao_alsa,
        &ao_dsp,
        NULL
};

ao_t *getdev(ao_type_t type){
        int i=0;
        while(1){
        if (ao_ls[i]->info->type ==type) return ao_ls[i];
        else if (ao_ls[i]==NULL) return NULL;
        i++;
        }
}
```

dsp.c 文件代码如下：

```
#include <stdio.h>
#include "libao.h"
static int dsp_fd;
static int dsp_open(char *devname,int flags){          //打开设备
             int dsp_fd=open(devname,flags);
             return dsp_fd;
             }
static int dsp_read(char *buf,int size){               //输入数据
            int size1=read(dsp_fd,buf,size);
            return size1;
}

static int dsp_write(char *buf,int size){              //输出数据
         int sze=write(dsp_fd,buf,size);
              return sze;
}

static int dsp_ioctl(int cmd,void *data){              //控制设备
              int ss=ioctl(dsp_fd,cmd,data);

}

static int dsp_close(){                                //关闭设备
         close(dsp_fd);
              return 0;
}

ao_info_t dsp_info={
          0,                                           //设备类型编号
         "dsp",                                        //oss、asla 等
         "ztl",                                        //作者
         "2016-1-29",                                  //编写时间
         "用于 dsp 音频设备接口"                       //块描述
};
```

```
ao_t ao_dsp={
        &(dsp_info),
    dsp_open,
    dsp_read,
    dsp_write,
    dsp_ioctl,
    dsp_close,
};
```

lib.h 文件代码如下：

```
#define LIB_H
#ifndef LIB_H
extern int minit(handler_t handler);
//播放函数,参数为 wav 音频文件名,如果暂停后继续播放,则可输入 NULL
extern int mplay(char *name);
//录音函数,参数为 wav 音频文件名
extern int mrecode(char *name);
//暂停,可暂停当前录音或播放
extern int mpause();
//停止,可停止当前录音或播放
extern int mstop();
//设置播放位置,参数为秒
extern int msetpos(int t);
//前进,参数为秒
extern int fastforward(int diff);
//后退,参数为秒
extern int fastretreat(int diff);
//设置音量,音量值范围为 0~100
extern int fastretreat(int diff);
//设置音量,音量值范围为 0~100
extern int setvolume(int left,int right);
//获取音量,音量值范围为 0~100
extern int getvolume(int *left,int *right);
//销毁
extern int destory();
#endif
```

libmplay.h 文件代码如下：

```
#ifndef MPLAY_H
#define MPLAY_H
//枚举,表示各种状态
typedef enum {STOP,PLAY,RECODE,PAUSE,READY} state_t;
//函数指针类型,表示播放状态
/*参数 state 为 STOP 时,val 为 0
                为 PLAY 时,val  为当前已播放时间
                为 RECODE 时,val 为当前已录音时间
                为 PAUSE 时,val 为 0
                为 READY 时,val 为播放前可播时间
*/
typedef void (*handler_t)(state_t state,int val);
```

```
//初始化函数,参数为回调函数句柄
int minit(handler_t handler);
//播放函数,参数为 wav 音频文件名,如果暂停后继续播放,则可输入 NULL
int mplay(char *name);
//录音函数,参数为 wav 音频文件名
int mrecode(char *name);
//暂停,可暂停当前录音或播放
int mpause();
//停止,可停止当前录音或播放
int mstop();
//设置播放位置,参数为秒
int msetpos(int t);
//前进,参数为秒
int fastforward(int diff);
//后退,参数为秒
int fastretreat(int diff);
//设置音量,音量值范围为 0~100
int setvolume(int left,int right);
//获取音量,音量值范围为 0~100
int getvolume(int *left,int *right);
//销毁
int destroy();
#endif
```

libmplay.c 文件代码如下：

```
#include<stdio.h>
#include "libmplay.h"
#include "libao.h"
#include "lib.h"
#include <linux/ioctl.h>
#include<linux/soundcard.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include<errno.h>
#include<malloc.h>
#include <sys/ioctl.h>

typedef struct WAV{
        char riff[4];           //RIFF
        long len;               //文件大小
        char type[4];           //WAVE
        char fmt[4];
        char tmp[4];
        short pcm;
        short channel;          //声道数
        long sample;            //采样频率
        long rate;              //传送速率
```

```
            short framesize;            //调整数
            short bit;                  //样本位数
            char data[4];
              long dblen;
        }wav_t;
handler_t pp;
int ld,sd,dif,avfd;
state_t state;
char mz[23];
//struct ao_t *p=ao_dsp;
int minit(handler_t handler)
{
      //printf("--------8\n");
      pp=handler;
}
  void *run(void *arg)
{
  int t=0;
  int i=0;
  while(state) {
        int i=0;
        char stt[4096];
        ld=open(mz,O_RDONLY);
        wav_t sty;
        int size=read(ld,&sty,44);
        if(size==-1) printf("%s\n",strerror(errno));
        sd=ao_dsp.open("/dev/dsp",O_WRONLY);
        int a=sty.sample;
        int b=sty.channel;
        int c=sty.bit;
        ioctl(sd,SNDCTL_DSP_SPEED,&a);
        ioctl(sd,SNDCTL_DSP_CHANNELS,&b);
        ioctl(sd,SNDCTL_DSP_SETFMT,&c);
        int size1;
        int len=0;
    while((size1=read(ld,stt,4000))!=0) {
        while(state==RECODE) usleep(20000);
            if(dif!=0) lseek(ld,dif,1);
            dif=0;
            len+=size1;
            t=len/(a*b*c/8);
            pp(state,t);
            write(sd,stt,size1);
              }
  }
     close(ld);
     close(sd);
     }
int mplay(char *name)
{
   strcpy(mz,name);
```

```
        if(state==0)
       {
        state=PLAY;
        pthread_t id;
        printf("---------9\n");
        pthread_create(&id,NULL,run,NULL);
           }
       else
       state=PLAY;
}
int mrecode(char *name)
{
    strcpy(mz,name);
    int fd=open("/dev/dsp",O_RDONLY);
    char buf[4095];
    int speed=44100;
    int bit=16;
    int channel=2;
    wav_t head={"RIFF",44,"WAVE","fmt ",0,0x10,channel,speed,speed*bit*channel/8,
         channel*bit/8,bit,"data",0};
        avfd=open(mz,O_CREAT | O_WRONLY);
         if (fd==-1){
                        printf("---- open dsp err=%s\n",strerror(errno));
                        return -1;
                        }
          ioctl(fd,SNDCTL_DSP_SPEED,&speed);
          ioctl(fd,SNDCTL_DSP_CHANNELS,&channel);
          ioctl(fd,SNDCTL_DSP_SETFMT,&bit);
          write(avfd,&head,sizeof(head));
          int len=0,t=0;
    while(1) {
              int size=read(fd,buf,4000);
              write(avfd,buf,size);
              len+=size;
              t=len/(channel*bit*speed/8);
                if(t>5) break;
                   }
       int all=len+44;
       lseek(avfd,0x04,0);
       write(avfd,&all,4);
       lseek(avfd,0x28,0);
       write(avfd,&len,4);
       close(fd);
       close(avfd);

//暂停,可暂停当前录音或播放
}
int mpause()
{
     state=RECODE;
     pp(state,0);
```

```
//停止,可停止当前录音或播放
}
int mstop()
{
   return -1;
//设置播放位置,参数为秒
}
int msetpos(int t)
{
   int sx=lseek(ld,0,1);
   return sx;

//前进,参数为秒
}
int fastforward(int diff)
{
   dif=diff;
//后退,参数为秒
}
int fastretreat(int diff)
{
dif=diff;
//设置音量,音量值范围为 0~100
}
int setvolume(int left,int right)
{
    int vol=right<<8 | left;
    ioctl(sd,MIXER_WRITE(SOUND_MIXER_VOLUME),&vol);
//获取音量,音量值范围为 0~100
}
int getvolume(int *left,int *right)
{
   int vol1;
   ioctl(sd,MIXER_READ(SOUND_MIXER_VOLUME),&vol1);
   *right=(vol1>>8)&0xff;
   *left=vol1&0xff;
//销毁
}
int destroy()
{

}
```

test.c 文件代码如下：

```
#include<stdio.h>
#include "lib.h"
int curstate,sq;
void fun(int state,int val)
{
```

```
        curstate=state;
}
int main(int argc,char **argv)
{
   printf("*************************************************\n");
   printf("p 播放        q 暂停      r 录音       s 停止        d 销毁       \n");
   printf("f 播放位置   n 前进      t 后退       e 设置音量   h 获取音量 \n");
   printf("*************************************************\n");
   char buf[23];
    while(1) {
        if(sq==-1) break;
        scanf("%s",buf);
        if(strcmp(buf,"p")==0) {
        if(curstate==3) {
          minit(NULL);
          mplay(NULL);
                }
          else {
                  minit(fun);
                  mplay(argv[1]);
                    }
             }
       else if(strcmp(buf,"s")==0)
            sq=mstop();
            else if(strcmp(buf,"q")==0)
               mpause();
           else if(strcmp(buf,"f")==0)
             {
            int a=msetpos();
              printf("--%d\n",a);
             }
          else if(strcmp(buf,"n")==0)
          {
               fastforward(900000);
          }
          else if(strcmp(buf,"t")==0)
               fastretreat(-500000);
          else if(strcmp(buf,"e")==0)
               {
                mpause();
                int b,c;
                scanf("%d %d",&b,&c);
                setvolume(b,c);
                }
          else if(strcmp(buf,"h")==0)
                {
                  mpause();
                  int vol,d,f;
                  getvolume(&d,&f);
                  vol=f<<8 | d;
                  printf("%d %d %d",d,f,vol);
```

```
                }
            else if(strcmp(buf,"r")==0)
                mrecode(argv[1]);
    }
}
```

编译代码，运行程序，结果如下：

```
[root@bogon Example5.4-2]# ls
audio1.wav   libao.c   lib.h          libmplay.h   test
dsp.c        libao.h   libmplay.c     music.WAV    test.c
[root@bogon Example5.4-2]# gcc -l pthread libao.c dsp.c libmplay.c test.c -o test
[root@bogon Example5.4-2]# ./test music.WAV
*********************************************************
p 播放        q 暂停      r 录音       s 停止        d 销毁
f 播放位置    n 前进      t 后退       e 设置音量    h 获取音量
*********************************************************
p
---------9
```

5.5　Linux 时间编程

在 Linux 系统下，经常需要输出系统当前的时间，涉及获取一些关于时间的信息，时间主要有世界标准时间和日历时间。

协调世界时（Coordinated Universal Time，UTC），又称为世界标准时间，也就是大家所熟知的格林威治标准时间（Greenwich MeanTime，GMT）。

日历时间（Calendar Time），是用“从一个标准时间点”（如 1970 年 1 月 1 日 0 点）到此时经过的秒数来表示的时间。

5.5.1　取得目前的时间

表头文件

```
#include<time.h>
```

定义函数

```
time_t time(time_t *t);
```

函数说明　此函数会返回从公元 1970 年 1 月 1 日的 UTC 时间从 0 时 0 分 0 秒算起到现在所经过的秒数。如果 t 不是空指针，此函数也会将返回值存到 t 指针所指的内存。

返回值　成功则返回秒数，失败则返回（(time_t) -1）值。

示例 5.5.1-1　time 函数使用方法。

```
#include<time.h>
#include <stdio.h>
main(){
    int seconds= time((time_t*)NULL);
```

```
        printf("%d\n",seconds);
}
```

取得时间命令的程序运行结果如下：

```
[root@localhost chapter5]# gcc Example5.5.1-1.c -o Example5.5.1-1
[root@localhost chapter5]# ./Example5.5.1-1
1489710775
```

5.5.2 取得目前时间和日期

表头文件

```
#include<time.h>
```

定义函数

```
struct tm*gmtime(const time_t*timep);
```

函数说明 gmtime()将参数 timep 所指的 time_t 结构中的信息转换成真实世界所使用的时间日期表示方法，然后将结果由结构 tm 返回。

tm 的结构定义如下：

```
struct tm
{
        int tm_sec;
        int tm_min;
        int tm_hour;
        int tm_mday;
        int tm_mon;
        int tm_year;
        int tm_wday;
        int tm_yday;
        int tm_isdst;
};
```

其中，int tm_sec 代表目前秒数，正常范围为 0~59，但允许至 61 秒；int tm_min 代表目前分数，范围为 0~59；int tm_hour 是从午夜算起的小时数，范围为 0~23；int tm_mday 是目前月份的日数，范围为 1~31；int tm_mon 代表目前月份，从一月算起，范围为 0~11；int tm_year 是从 1900 年算起至今的年数；int tm_wday 是一星期的日数，从星期一算起，范围为 0~6；int tm_yday 是从今年 1 月 1 日算起至今的天数，范围为 0~365；int tm_isdst 是日光节约时间的旗标。此函数返回的时间日期未经时区转换，是 UTC 时间。

返回值 返回结构体 tm 代表的是目前的 UTC 时间。

示例 5.5.2-1 gmtime()函数的使用方法。

```
#include<time.h>
#include <stdio.h>
main()
{
```

```
        char *wday[]={"Sun","Mon","Tue","Wed","Thu","Fri","Sat"};
        time_t timep;
        struct tm *p;
        time(&timep);
        p=gmtime(&timep);
        printf("%d-%d-%d\n",(1900+p->tm_year),(1+p->tm_mon),p->tm_mday);
        printf("%s%d:%d:%d\n",wday[p->tm_wday],p->tm_hour,p->tm_min, p->tm_sec);
}
```

取得目前时间和日期程序运行结果如下：

```
[root@localhost chapter5]# gcc Example5.5.2-1.c -o Example5.5.2-1
[root@localhost chapter5]# ./Example5.5.2-1
2017-3-17
Fri0:34:32
```

5.5.3　取得当地目前时间和日期

表头文件

```
#include<time.h>
```

定义函数

```
struct tm *localtime(const time_t * timep);
```

函数说明　localtime()将参数 timep 所指的 time_t 结构中的信息转换成真实世界所使用的时间日期表示方法，然后将结果由结构 tm 返回。此函数返回的时间日期已经转换成当地时区。

返回值　返回结构体 tm，代表目前的当地时间。

示例 5.5.3-1　localtime()函数的使用方法。

```
#include<time.h>
#include <stdio.h>
main(){
      char *wday[]={"Sun","Mon","Tue","Wed","Thu","Fri","Sat"};
      time_t timep;
      struct tm *p;
      time(&timep);
      p=localtime(&timep);
      printf("%d-%d-%d    ",(1900+p->tm_year), (1+p->tm_mon),p->tm_mday);
      printf("%s%d:%d:%d\n", wday[p->tm_wday], p->tm_hour, p->tm_min, p->tm_sec);
}
```

取得当地目前时间和日期程序运行结果如下：

```
[root@localhost chapter5]# gcc Example5.5.3-1.c -o Example5.5.3-1
[root@localhost chapter5]# ./Example5.5.3-1
2017-3-17    Fri8:35:50
```

5.5.4 将时间结构数据转换成经过的秒数

表头文件

```
#include<time.h>
```

定义函数

```
time_t mktime(struct tm * timeptr);
```

函数说明 mktime()用来将参数 timeptr 所指的 tm 结构数据转换成从公元 1970 年 1 月 1 日 0 时 0 分 0 秒算起至今的 UTC 时间所经过的秒数。

返回值 返回经过的秒数。

示例 5.5.4-1 用 time()取得时间(秒数),利用 localtime()转换成结构 tm,再利用 mktine()将结构 tm 转换成原来的秒数。

```
#include<time.h>
#include <stdio.h>
main(){
      time_t timep;
      struct tm *p;
      time(&timep);                    //取得时间(秒数)
      printf("time(): %d\n",timep);
      p=localtime(&timep);
      timep=mktime(p);
      printf("time()->localtime()->mktime():%d\n",timep);
}
```

Mktime()函数程序运行结果如下:

```
[root@localhost chapter5]# gcc Example5.5.4-1.c -o Example5.5.4-1
[root@localhost chapter5]# ./Example5.5.4-1
time(): 1489711031
time()->localtime()->mktime():1489711031
```

5.5.5 设置目前时间

表头文件

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

定义函数

```
int settimeofday(const struct timeval *tv,const struct timezone *tz);
```

函数说明 settimeofday()把目前时间设成由 tv 所指的结构信息,当地时区信息则设成 tz 所指的结构。详细的说明请参考 gettimeofday()。

注意:只有 root 权限才能使用此函数修改时间。

返回值 成功则返回 0,失败则返回−1。

5.5.6　取得当前时间

表头文件

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

定义函数

```
int gettimeofday ( struct timeval * tv , struct timezone * tz )
```

函数说明　gettimeofday()把目前的时间有 tv 所指的结构返回，当地时区的信息则放到 tz 所指的结构中。

timeval 结构定义如下：

```
struct timeval
{
        long tv_sec;            //秒
        long tv_usec;           //微秒
};
```

timezone 结构定义如下：

```
struct timezone
{
        int tz_minuteswest;     //和 Greenwich 时间差了多少分钟
        int tz_dsttime;         //日光节约时间的状态
};
```

返回值　成功则返回 0，失败则返回−1。

EFAULT 指针 tv 和 tz 所指的内存空间超出存取权限。

示例 5.5.6-1　gettimeofday()函数的使用方法。

```
#include<stdio.h>
#include<sys/time.h>
#include<unistd.h>
main()
{
      struct timeval tv;
      struct timezone tz;
      gettimeofday (&tv , &tz);
      printf("tv_sec; %d\n", tv.tv_sec);
      printf("tv_usec; %d\n",tv.tv_usec);
      printf("tz_minuteswest; %d\n", tz.tz_minuteswest);
      printf("tz_dsttime, %d\n",tz.tz_dsttime);
}
```

取得目前的时间程序运行结果如下：

```
[root@localhost chapter5]# gcc Example5.5.6-1.c -o Example5.5.6-1
[root@localhost chapter5]# ./Example5.5.6-1
tv_sec; 1489711185
```

```
tv_usec; 123826
tz_minuteswest; -480
tz_dsttime, 0
```

5.5.7 将时间和日期以 ASCII 码格式表示

表头文件

```
#include<time.h>
```

定义函数

```
char * asctime(const struct tm * timeptr);
```

函数说明 将 tm 格式的时间转化为字符串，若再调用相关的时间日期函数，此字符串可能会被破坏。此函数与 ctime 不同处在于传入的参数是不同的结构。

返回值 返回一字符串表示目前当地的时间日期。

示例 5.5.7-1 asctime 函数的使用方法。

```
#include <time.h>
#include<stdio.h>
main()
{
      time_t timep;
      time (&timep);
      printf("%s",asctime(gmtime(&timep)));
}
```

asctime 命令程序运行结果如下：

```
[root@localhost chapter5]# gcc Example5.5.7-1.c -o Example5.5.7-1
[root@localhost chapter5]# ./Example5.5.7-1
Fri Mar 17 00:41:02 2017
```

5.5.8 将时间和日期以字符串格式表示

表头文件

```
#include<time.h>
```

定义函数

```
char *ctime(const time_t *timep);
```

函数说明 将日历时间转化为本地时间的字符串形式，若再调用相关的时间日期函数，此字符串可能会被破坏。

返回值 返回一字符串表示目前当地的时间日期。

示例 5.5.8-1 ctime 函数的使用方法。

```
#include<time.h>
#include<stdio.h>
```

```
main()
{
    time_t timep;
    time (&timep);
    printf("%s",ctime(&timep));
}
```

ctime 命令程序运行结果如下：

```
[root@localhost chapter5]# gcc Example5.5.8-1.c -o Example5.5.8-1
[root@localhost chapter5]# ./Example5.5.8-1
Fri Mar 17 08：43：00 2017
```

示例 5.5.8-2　写一段程序，实现本地时间和格林尼治时间的转换。

```
#include <time.h>
#include <stdio.h>
int main(void){
    struct tm *local;
    time_t t;
    //获取日历时间
    t=time(NULL);
    //将日历时间转化为本地时间
    local=localtime(&t);
    //打印当前的小时值
    printf("Local hour is: %d\n",local->tm_hour);
    //将日历时间转换为格林威治时间
    local=gmtime(&t);
    printf("UTC hour is: %d\n",local->tm_hour);
    return 0;
}
```

运行结果如下：

```
[root@localhost chapter5]# gcc Example5.5.8-2.c -o Example5.5.8-2
[root@localhost chapter5]# ./Example5.5.8-2
Local hour is: 8
UTC hour is: 0
```

习题与练习

1．文件 I/O 编程指的是什么?可以用哪些方法实现？
2．基于 C 语言的库函数文件编程和基于 Linux 系统的文件编程的区别是什么？
3．什么是标准时间？什么是日历时间？
4．编写写入数据程序，将一串字符串 a~f 写入到/tmp/test.txt 中。
5．编写读取文件程序，将/tmp/test.txt 中内容读出。
6．编写时间转换程序，将当前时间转换为格林威治时间。
7．使用 creat 函数创建一个文件，当没有文件名输入时，提示用户输入一个文件名。