

第二章：文件

目标：

本章旨在向学员介绍Linux系统下关于文件相关的系统调用：

- 1) 掌握使用文件相关的系统调用函数
- 2) 了解I/O相关函数与系统调用函数之间的区别

时间：6 学时

教学方法：讲授PPT、实例练习



2.1 文件结构

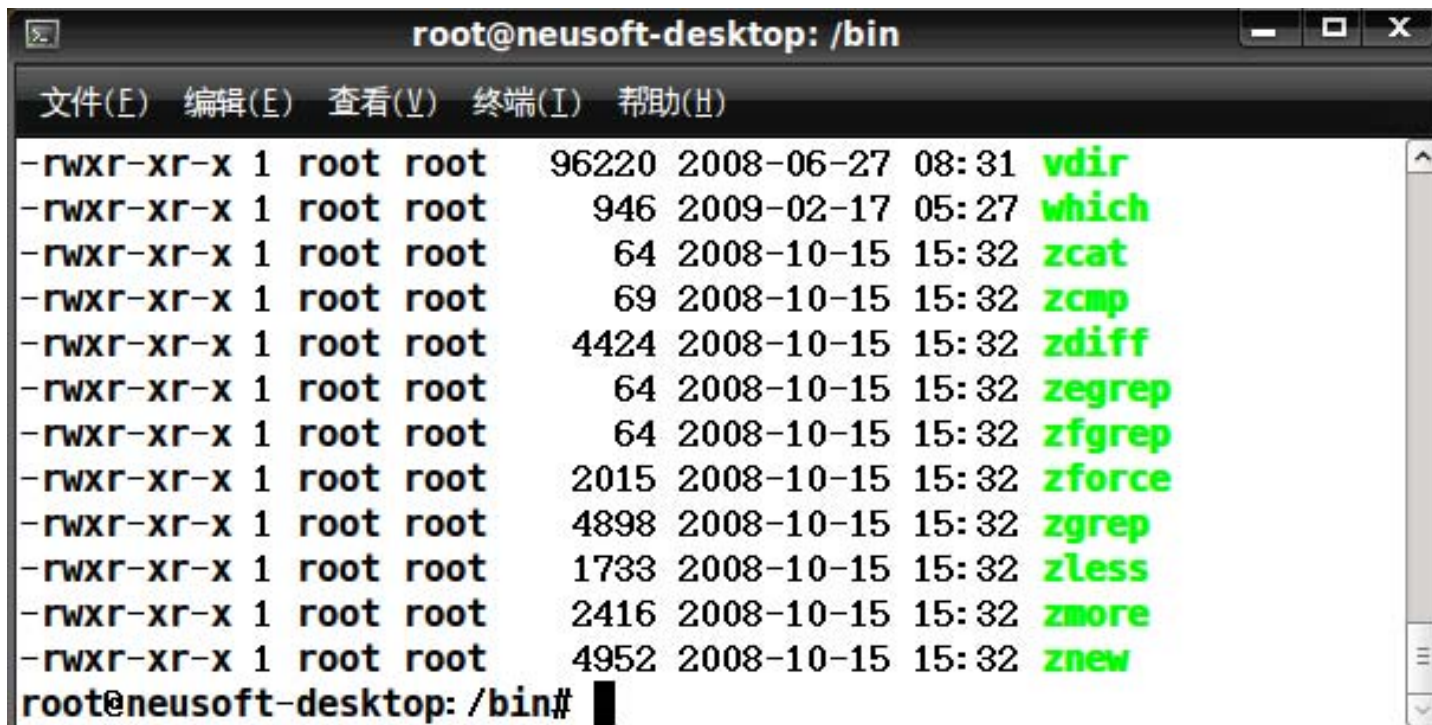
Linux中的文件提供了操作系统服务和设备维护的简单统一的接口。一般情况下可以像对待文件一样操作串行口、打印机等硬件设备。

Linux里的一切事物皆文件。

一个文件除了本身包含的内容以外，还有其他一些用于管理方面的属性信息，比如文件的建立、修改时间、访问权限等属性。

2.1 文件结构

- 执行ls -l显示文件详细信息



```
root@neusoft-desktop: /bin
文件(E) 编辑(E) 查看(V) 终端(I) 帮助(H)
-rwxr-xr-x 1 root root 96220 2008-06-27 08:31 vdir
-rwxr-xr-x 1 root root 946 2009-02-17 05:27 which
-rwxr-xr-x 1 root root 64 2008-10-15 15:32 zcat
-rwxr-xr-x 1 root root 69 2008-10-15 15:32 zcmp
-rwxr-xr-x 1 root root 4424 2008-10-15 15:32 zdiff
-rwxr-xr-x 1 root root 64 2008-10-15 15:32 zegrep
-rwxr-xr-x 1 root root 64 2008-10-15 15:32 zfgrep
-rwxr-xr-x 1 root root 2015 2008-10-15 15:32 zforce
-rwxr-xr-x 1 root root 4898 2008-10-15 15:32 zgrep
-rwxr-xr-x 1 root root 1733 2008-10-15 15:32 zless
-rwxr-xr-x 1 root root 2416 2008-10-15 15:32 zmore
-rwxr-xr-x 1 root root 4952 2008-10-15 15:32 znew
root@neusoft-desktop: /bin#
```

模式(mode)、链接数(links)、文件所有者(owner)、组(group)、文件大小(size)、最后修改时间(last-modified)、文件名(name)

模式包含是否为目录文件，文件所有者的读写执行权限、组的读写执行权限、其他人的读写执行权限。

2.1 文件结构

- 这些属性都被保存在inode的数据结构里，文件长度和它在磁盘上的存放地点也保存在inode里。
- inode结构体和inode编号一一对应。
- 系统使用的是文件的inode编号。
- `ls -li`可以看到inode编号和文件名的对应情况
- 综上，文件结构分为inode和block两部分。

```
roo@ubuntu:/etc$ ls -li
786441 acpi
786567 adduser.conf
787688 adjtime
786442 alternatives
786569 anacrontab
804277 anthy
786570 apg.conf
786443 apm
786444 apparmor
786445 apparmor.d
786446 appport
786447 apt
786571 at.deny
786507 lightdm
786606 locale.alias
791362 localtime
786508 logcheck
786608 login.defs
786609 logrotate.conf
786509 logrotate.d
786510 lsb-base
786610 lsb-base-logging.sh
798685 lsb-release
786612 ltrace.conf
786613 magic
786614 magic.mime
```

2.1 文件结构

inode结构体

用于存储文件的各属性，包括：

- 所有者信息：文件的owner， group；
- 权限信息： read、write和execute；
- 时间信息： 建立或改变时间（ctime）、最后读取时间（atime）、最后修改时间（mtime）；
- 标志信息： 一些flags；
- 内容信息： type， size， 以及相应的block的位置信息。

注意：不记录文件名或目录名，文件名或目录名记录在文件所在目录对应的block里。

block

用来存储文件的内容。

2.1 文件结构

创建目录或文件

当创建一个目录时，文件系统会为该目录分配一个inode和至少一个block。该inode记录该目录的属性，并指向那块block。该block记录该目录下相关联的文件或目录的inode编号和名字。

当创建一个文件时，文件系统会为该文件分配至少一个inode和与该文件大小相对应的数量的block。该inode记录该文件的属性，并指向block。

如果一个目录中的文件数太多，以至于1个block容纳不下这么多文件时，Linux的文件系统会为该目录分配更多的block。

-

2.1 文件结构

- 读取目录或文件
- 例读取/home下的test.c
- 首先根目录的inode编号固定为0.
- 通过根目录的inode编号找到其inode结构体，通过inode结构体找到其block。
- 目录的block内容为该目录下的文件的inode号与文件名字的表格
- 可以通过查询block找到home的inode编号，然后去找寻相应的inode结构体，进而找到home的block。类似的，在home的block里存放着test.c的inode编号，通过该编号可以找到test.c的内容，进而完成文件内容的读取。
- 目录数据项

Inode编号	文件名字
917505	boot
393217	home
786433	etc

2.2 文件的基本操作

文件操作是通过操作系统提供的系统调用实现对文件的创建、访问、编辑、删除等操作。

文件相关的基本系统调用如下

open	打开一个文件来读写或创建一个空文件
creat	创建一个空文件
close	关闭以前打开的文件
read	从文件中读数据
write	写数据到文件中

这些系统调用函数的说明放在man手册的第二小节

2.2 文件的基本操作

- 典型的文件操作例程:

```
#include <unistd.h>
#include <fcntl.h>
int main()
{
    int    fd;
    ssize_t nread;
    char    buf[1024];
    fd = open("data", O_RDONLY);
    if(fd == -1) exit(1);
    nread = read(fd, buf, 1024);
    if(nread == -1) exit(1);
    close(fd);
    exit(0);
}
```

分析，这个例程先以只读方式打开当前目录名为“data”的文件，如果失败，返回-1，否则返回非负整数，即文件描述符，由fd保存。随后通过read读取文件中的数据，返回值nread记录真正读出的字符个数，如果出错也返回-1。
技巧：在linux中返回-1则为出错。从一个名为errno的全局变量中可以发现错误类型。

2.2 文件的基本操作

文件描述符的概念：

一个运行中的程序被称为一个进程，他有一些与之相关的文件描述符，文件描述符是一些小的、正整数数值的数，通过他们可以访问打开的文件和设备

- 文件描述符是打开文件的进程与文件之间的连接
- 文件描述符是一个正整数的值
- 同时打开几个文件，描述符不相同
- 一个文件打开多次，描述符也不相同

2.2 文件的基本操作

```
#include<unistd.h>
/* Standard file descriptors. */
#define STDIN_FILENO 0    /* 标准输入. */
#define STDOUT_FILENO 1   /* 标准输出. */
#define STDERR_FILENO 2   /* 标准错误输出. */
```

- 注意：
- 新文件描述符总是取未用描述符中的最小值，比如说一个程序已经关闭了自己的标准输出，则再次open时就会使用“1”作为文件描述符，这样标准输出就会被定向到另外的文件或者设备。

2.2 文件的基本操作

- open 系统调用：建立了一条到文件或设备的访问路径。
- 功能为打开或创建文件
- 使用函数需要的头文件可以通过man手册查找

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

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

- 参数pathname是一个字符指针，指向文件路径及文件名。
比如： /home/guest/test

2.2 文件的基本操作

- 参数flags是一个整形参数，定义为以何种方式访问文件

O_RDONLY 只读打开文件

O_WRONLY 只写打开文件

O_RDWR 读写打开文件

以下具体使用规则请参见man手册

O_CREAT 按mode中给出的访问方式创建文件

O_EXCL 检查文件是否存在（配合O_CREAT参数使用）

O_TRUNC 强制创建文件（与O_CREAT参数配合使用时）

O_APPEND open函数打开文件后，自动调整读写指针指向文件尾

2.2 文件的基本操作

- 参数mode是可选参数，只有第二参数flags为O_CREAT时才有效，创建文件以后，马上赋予文件何种访问权限，比如：0644

1	1	0	1	0	0	1	0	0
---	---	---	---	---	---	---	---	---

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

S_IRUSR: 当前用户读权限

S_IWUSR: 当前用户写权限

S_IXUSR: 当前用户执行权限

S_IRGRP: 当前组读权限

S_IWGRP: 当前组写权限

S_IXGRP: 当前组执行权限

S_IROTH: 其他用户读权限

S_IWOTH: 其他用户写权限

S_IXOTH: 其他用户执行权限

```
open ("file", O_CREAT, S_IRUSR | S_IXOTH);
```

执行以上代码，使用ls-l查看文件权限

2.2 文件的基本操作

- write系统调用：把缓冲区buffer里的前n个字节写入与文件描述符filedes相关联的文件中。返回值是实际写出的字节数。

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

```
ssize_t write(int filedes, const void *buffer, size_t n);
```

write函数与read相对应。

2.2 文件的基本操作

- write例程:

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

int main()
{
    if ((write(1, "Here is come data\n", 18)) != 18)
        write(2, "A write error has occurred on file descriptor 1\n", 46);
    exit(0);
}
```

2.2 文件的基本操作

- read系统调用：从与文件描述符filedes相关联的文件里读入nbytes个字节的数据，并把他们放到buffer数据区中。返回值是实际写入的字节数。

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

```
ssize_t read(int filedes, void *buffer, size_t nbytes);
```

参数filedes是之前open或creat调用返回的文件描述符。

参数buffer是指向数组或结构的指针。

参数是从文件中读取的字节数

2.2 文件的基本操作

- read例程:

```
#include <unistd.h>
#include <stdlib.h>
int main()
{
    char buffer[128];
    int nread;

    nread = read(0, buffer, 128);
    if (nread == -1)
        write(2, "A read error has occurred\n", 26);
    if ((write(1, buffer, nread)) != nread)
        write(2, "A write error has occurred\n", 27);
    exit(0);
}
```

2.2 文件的基本操作

- creat系统调用：功能为创建并打开文件，作用相当于
- 以O_WRONLY | O_CREAT | O_TRUNC为flags的open调用。

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

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

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

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

- 参数pathname为文件路径名
- 参数为赋予创建文件的访问权限。
- `filedes = creat("/tmp/newfile", 0644);`等价于
- `filedes = open("/tmp/newfile", O_WRONLY | O_CREAT | O_TRUNC, 0644);`

2.2 文件的基本操作

- close系统调用：功能为终止一个文件描述符与文件的关联。文件描述符可以重新被使用。
- 一个运行的程序一次性能够打开的文件数量是有限的。

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

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

```
filedes = open("file", O_RDONLY);
```

```
...
```

```
...
```

```
close(filedes);
```

执行成功返回0， 错误返回-1.

2.2 文件的基本操作

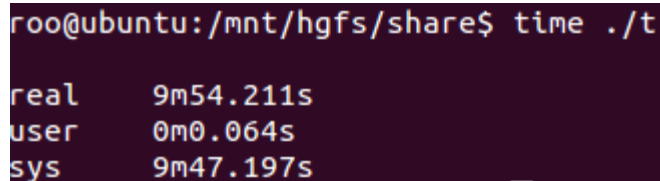
练习：请编写程序， 功能为把一个文件的内容复制到另一个文件中。

2.2 文件的基本操作

思考：如何改善拷贝大容量文件的效率？

2.2 文件的基本操作-练习对比

```
#include<unistd.h>
#include<sys/stat.h>
#include<fcntl.h>
#include<stdio.h>
int main()
{
    char c;
    int in,out;
    in = open("file.in",O_RDONLY);
    out = open("file.out",O_WRONLY|O_CREAT,S_IRUSR|S_IWUSR);
    if ((in == -1)||(out == -1))exit(1);
    while(read(in,&c,1) == 1)
        write(out,&c,1);
    close(in);
    close(out);
    exit(0);
}
```



```
roo@ubuntu:/mnt/hgfs/share$ time ./t
real    9m54.211s
user    0m0.064s
sys     9m47.197s
```

三项分别是实际使用时间、用户态使用时间、内核态使用时间

2.2 文件的基本操作-练习对比

```
#include<unistd.h>
#include<sys/stat.h>
#include<fcntl.h>
#include<stdio.h>
#include<stdlib.h>
int main()
{
    char block[1024];
    int in,out;
    int nread;
    in = open("file.in",O_RDONLY);
    out = open("file.out",O_WRONLY|O_CREAT,S_IRUSR|S_IWUSR);
    if ((in == -1)||(out == -1))exit(1);
    while(nread=read(in,block,sizeof(block)) >0)
        write(out,block,nread);
    close(in);
    close(out);
    exit(0);
}
```

```
roo@ubuntu:/mnt/hgfs/share$ time ./t2

real    0m0.513s
user    0m0.000s
sys     0m0.488s
```

2.2 文件的基本操作-练习对比

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int c;
    FILE *in,*out;
    int nread;
    in = fopen("file.in","r");
    out = fopen("file.out","w");
    if ((in == -1)||(out == -1))exit(1);
    while ((c = fgetc(in)) != EOF)
        fputc(c,out);
    close(in);
    close(out);
    exit(0);
```

```
roo@ubuntu:/mnt/hgfs/share$ time ./ttt
real    0m0.642s
user    0m0.000s
sys     0m0.616s
```

练习对比结论

- 第一个例子必须进行两百万次左右的系统调用
- 第二个例子采用了大数据块的拷贝、用的还是系统调用、但是只做2000次的系统调用就足够了。
- 第三个例子虽然不如第二个例子速度快，但是比第一个例子已经好了很多了，原因在于stdio库在FILE结构里使用了一个缓冲区，只有在缓冲区填满的时候才进行底层系统调用。

2.2 文件的基本操作

- 其他和文件有关的函数:
- lseek 移动文件中特定字节的读写指针
- perror 错误处理
- stat、fstat、lstat 获得文件属性
- chmod 修改文件权限及特殊属性
- chown 修改文件所有者和所有组
- link 创建一个文件的链接
- unlink 删除连接
- symlink 创建符号链接

2.2 文件的基本操作

- `fstat`、`stat`和`lstat`系统调用显示文件的状态信息

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

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

- `fstat`返回与一个打开的文件描述符关联着的文件的状态信息，这些信息将被写到一个`stat`结构里（见后页），其地址作为参数被传递过去
- `stat`与`lstat`使用文件名查看文件状态信息。
- 两者基本一致，但当文件是一个符号连接时，`lstat`返回符号连接本身信息，`stat`返回该连接指向的文件的信息。

2.2 文件的基本操作

- stat结构体主要成员:

st_mode	文件权限和文件类型信息
st_ino	与该文件关联的inode
st_dev	保存文件的设备
st_uid	文件属性的UID号
st_gid	文件属性的GID号
st_atime	文件上一次被访问的时间
st_ctime	文件的权限、所有者、组或内容上一次改变的时间
st_mtime	文件的内容上一次被修改的时间
st_nlink	该文件上硬连接的个数

2.2 文件的基本操作

- 例程：获得文件的大小

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

int main()
{
    struct stat infobuf;
    if (stat("/etc/passwd", &infobuf) == -1)
        perror("/etc/passwd");
    else
        printf("The size of /etc/passwd is %d\n",
            infobuf.st_size);
}
```

2.2 文件的基本操作

- stat结构中返回的st_mode标志还有一些关联的宏定义在头文件sys/stat.h中，其中包括对访问权限、文件类型以及部分掩码的定义，具体可以参加man手册中的描述
- st_mode结构：

文件类型				特殊位			用户			组			其他		
				u	g	s	r	w	x	r	w	x	r	w	x

- 文件类型包括：

#define	S_IFBLK	0060000	文件是一个特殊块设备
#define	S_IFDIR	0040000	文件是一个目录
#define	S_IFCHR	0020000	文件是一个特殊的字符设备
#define	S_IFIFO	0010000	文件是一个FIFO设备
#define	S_IFREG	0100000	文件是一个普通文件
#define	S_FLNK	0120000	文件是一个符号链接

2.2 文件的基本操作

其他模式标志包括：

The following flags are defined for the `st_mode` field:

<code>S_IFMT</code>	<code>0170000</code>	bit mask for the file type bit fields
<code>S_IFSOCK</code>	<code>0140000</code>	socket
<code>S_IFLNK</code>	<code>0120000</code>	symbolic link
<code>S_IFREG</code>	<code>0100000</code>	regular file
<code>S_IFBLK</code>	<code>0060000</code>	block device
<code>S_IFDIR</code>	<code>0040000</code>	directory
<code>S_IFCHR</code>	<code>0020000</code>	character device
<code>S_IFIFO</code>	<code>0010000</code>	FIFO
<code>S_ISUID</code>	<code>0004000</code>	set UID bit
<code>S_ISGID</code>	<code>0002000</code>	set-group-ID bit (see below)
<code>S_ISVTX</code>	<code>0001000</code>	sticky bit (see below)
<code>S_IRWXU</code>	<code>00700</code>	mask for file owner permissions
<code>S_IRUSR</code>	<code>00400</code>	owner has read permission
<code>S_IWUSR</code>	<code>00200</code>	owner has write permission
<code>S_IXUSR</code>	<code>00100</code>	owner has execute permission
<code>S_IRWXG</code>	<code>00070</code>	mask for group permissions
<code>S_IRGRP</code>	<code>00040</code>	group has read permission

2.2 文件的基本操作

- 例程：判断是否为目录

```
#include <stdio.h>
#include <sys/stat.h>
int main()
{
    struct stat infobuf;
    if (stat("./abc", &infobuf) == -1)
    {
        perror("/abc");
        return -1; }
    if ((infobuf.st_mode & 0170000) == 0040000)
        printf("this is a directory\n");
    else
        printf("this is a file\n");
    return 0;
}
```

这句话也可以使用定义好的宏，修改为
if ((infobuf.st_mode & S_IFMT) == S_IFDIR)



2.2 文件的基本操作

- 解释st_mode标志的掩码包括：

S_IFMT	文件类型
S_IRWXU	所属者的读/写/执行权限
S_IRWXG	所属组的读/写/执行权限
S_IRWXO	其他用户的读/写/执行权限

- 辅助确定文件类型的宏定义

S_ISBLK	测试是否是特殊的块设备文件
S_ISCHR	测试是否是特殊的字符设备文件
S_ISDIR	测试是否是目录
S_ISFIFO	测试是否是FIFO设备
S_ISREG	测试是否是普通文件
S_ISLNK	测试是否是符号链接文件

2.2 文件的基本操作

- 例程：测试一个文件，不是目录

```
struct stat statbuf;  
mode_t modes;  
  
stat("filename",&statbuf);  
...  
modes = statbuf.st_mode;  
if (!S_ISDIR(modes))  
....
```

2.2 文件的基本操作

- 例程：测试一个文件，不是目录,设置了属主的执行权限、不再有其他权限，使用下面代码：

```
struct stat statbuf;  
mode_t modes;  
  
stat("filename",&statbuf);  
...  
modes = statbuf.st_mode;  
if (!S_ISDIR(modes) && ((modes & S_IRWXU) == S_IXUSR))  
....
```

2.2 文件的基本操作

- lseek系统调用：改变已打开文件中指针位置

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

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

filedes参数为文件描述符

offset参数为表示新位置相对起始位置的字节数

start_flag参数含义：

SEEK_SET offset是从文件的起始文件开始算，通常值为0

SEEK_CUR offset相对文件读写的当前位置而言，通常值为1

SEEK_END offset相对文件尾而言，通常值为2

2.2 文件的基本操作

- perror函数

很多系统函数的调用会因为各种各样的原因而操作失败，在操作失败的时候，它们会设置外部变量error的值来指明自己失败的原因。程序必须在函数报告出错后检查errno变量，因为它很有可能被下个程序调用函数覆盖。

perror函数可以用来在错误出现时候报告出错误码。

perror函数把error变量中报告的当前错误映射到一个字符串，并把它输出到标准错误输出流。

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

```
void perror(const char *s);
```

如果s不为空，错误信息会先增加字符串s的内容，再打印错误信息。

2.2 文件的基本操作

例程：显示错误信息perror

```
int sample
{
    int fd;
    fd = open("file", O_RDONLY);
    if (fd == -1)
    {
        perror("Cannot open file");
        return;
    }
    ....
}
```

运行结果：

```
Cannot open file: No such file or directory
Cannot open file: Interrupted system call
```

2.2 文件的基本操作

- chmod系统调用

修改文件或目录的访问权限

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

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

path参数: 指定被修改权限的文件

mode参数: 修改的权限设置

2.2 文件的基本操作

- chown系统调用

改变文件或目录的所有者和组

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

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

用户ID和组ID数值通过getuid和getgid调用获得

2.2 文件的基本操作

- 例程：修改文件权限及所有者

```
#include <unistd.h>
#include <sys/stat.h>
int main()
{
    chmod("abc",04764);
    chmod("bcd",S_ISUID|S_IRWXU|S_IRGRP|S_IWGRP|S_IROTH);
    chown("abc",1000,1000);
    return 0;
}
```

2.2 文件的基本操作

- unlink 删除链接
- link 创建链接
- symlink 创建符号链接

返回值0表示成功，-1表示失败

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

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

```
int link(const char *path1, const char *path2);
```

```
int symlink(const char *path1, const char *path2);
```

unlink实质是删除该文件所在目录的一条目录数据项，即删除目录文件中的一个文件名及其inode号，并不是真正删除文件的block。当该文件的链接计数达到0，则会标志该block为未用。

link就是添加硬链接，symlink添加软链接

2.3 目录操作

- 相关系统调用：

opendir	打开目录
readdir	读目录
closedir	关闭目录
telldir	定位目录
seekdir	查找目录

2.3 目录操作

- opendir()返回指向目录流的指针DIR*

```
#include <sys/types.h>
#include <dirent.h>
DIR *opendir(const char *dirname);
```

2.3 目录操作

- readdir函数

函数返回一个指向结构的指针，结构里保存着目录流drip中下一个目录项的有关信息

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

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

dirent结构中包含目录数据信息包括以下：

```
struct dirent
{
    long d_ino; /* 索引节点inode号 */
    off_t d_off; /* 在目录文件中的偏移 */
    unsigned short d_reclen; /* 文件名长 */
    unsigned char d_type; /* 文件类型 */
    char d_name [NAME_MAX+1]; /* 文件名，最长255字符 */
}
```

2.3 目录操作

- 例程：显示某目录中的文件

```
#include <stdio.h>
#include <stdlib.h>
#include <dirent.h>
int main()
{
    DIR *dir_ptr;
    struct dirent *direntp;
    if ((dir_ptr = opendir("/home")) == NULL)
        perror("can not open /home");
    else
    {
        while ((direntp = readdir(dir_ptr))!= NULL)
            printf("%s\n",direntp->d_name);
        closedir(dir_ptr);
    }
    return 0;
}
```

2.3 目录操作

- telldir函数

返回值记录着一个目录流的当前位置

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

```
#include <dirent.h>
```

```
long int telldir(DIR *dirp);
```


2.3 目录操作

- telldir例程:

```
#include <stdio.h>
int main()
{
    DIR *dir_ptr;
    struct dirent *direntp;
    int dir_loc;
    if ((dir_ptr = opendir("./testdir")) == NULL)
        perror("can not open ./testdir");
    else {
        while ((direntp = readdir(dir_ptr)) != NULL)
        {
            printf("%s: ",direntp->d_name);
            dir_loc=telldir(dir_ptr);
            printf("%d\n",dir_loc);    }
            closedir(dir_ptr);  }
    return 0;
}
```

2.3 目录操作

- seekdir函数

设置目录流dirp的目录项指针

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

void seekdir(DIR *dirp, long int loc);
```

loc的值用来设置指针位置，通过telldir调用获得

2.3 目录操作

- `closedir`函数

关闭一个目录流并释放与之关联的资源

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

int closedir(DIR *dirp);
```

2.3 目录操作

小结:

- 文件与目录都有内容和属性，文件的内容是任意数据，目录的内容只能是文件名和子目录名的列表
- 内核提供系统调用读取目录的内容，读取和修改文件的属性
- 文件类型、文件访问权限和特殊属性编码存储在一个16位整数中，可以通过掩码技术读取这些信息
- 文件所有者和组的信息是以id形式存在的，id与用户、组名的联系通过文件passwd和group实现

2.3 目录操作

- 其他系统调用

<code>mkdir</code>	创建目录
<code>rmdir</code>	删除空目录
<code>chdir</code>	改变当前目录

2.3 目录操作

- mkdir和rmdir系统调用

```
#include <sys/stat.h>
int mkdir(const char *path, mode_t mode);

#include <unistd.h>
int rmdir(const char *path);
```

- mkdir创建一个新的目录节点，并把它链接到文件系统树
- rmdir从目录树删除一个节点，这个目录必须为空

2.3 目录操作

- chdir系统调用与getcwd函数
进出目录与确定当前工作目录

```
#include <unistd.h>
int chdir(const char *path);

#include <unistd.h>
char *getcwd(char *buf, size_t size);
```

Neusoft

Beyond Technology

Copyright © 2008 版权所有 东软集团