Linux系统编程-文件和目录

杨劲松 yjs@oldhand.org

2011.04.12

课程目标

- 掌握文件属性获取与修改相关的操作函数
  - fstat()/chmod()/chown()…
- 掌握目录操作相关的函数
  - mkdir()/rmdir()/chdir()/opendir()/readdir()…
- 掌握文件删除、修改、重命名相关的函数
  - unlink()/remove()/rename()…
- 掌握符号连接相关的函数
  - symlink()/readsymlink()…
- 了解UNIX文件系统的基础知识

参考资料

1. W.Richard Stevens《UNIX环境高级编程》
   ① 第4章文件和目录
2. 亚嵌教育《Linux 系统编程》

学习方法(建议)

1. 结合Linux内核的工作原理来理解
   - 系统函数正是内核提供给应用程序的接口
2. 熟练掌握C 语言
   - Linux内核是用C 语言写的，我们在描述内核工作原理时必然要用“指针”、“结构体”、链表这些名词来组织语言，就像只有掌握了英语才能看懂英文书一样，只有学好了C 语言才能看懂我描述的内核工作原理
3. 学会使用Man pages
4. 参考W. Richard Stevens《UNIX环境高级编程》

内容提纲

- 取得文件的属性信息
  - stat()fstat()/lstat()
  - 文件类型
  - 设置用户ID和设置组ID
  - 文件存取许可权
  - 新文件和目录的所有权
  - access()
  - umask()
  - chmod()/fchmod()
  - chown()/fchown()/lchown()
  - 文件长度
  - 文件截短
- 文件操作
  - link()/unlink()/remove()/rename()
  - 符号连接
  - symlink()/lreadlink()
  - 文件的时间
  - utime()函数
- 目录操作
  - mkdir()/rmdir()
  - 读目录
  - chdir()/fchdir()/getcwd()

1. 取得文件的属性信息

1.1.1 stat()/fstat()/lstat()

以下三个函数可以获取文件/目录的属性信息：

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

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);
```

- 三个函数的返回：若成功则为0，若出错则为-1
- 给定一个pathname的情况下：
  - 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。

1.1.2 stat()/fstat()/lstat() – struct stat

struct stat定义：

```
struct stat {
    dev_t         st_dev;      /* device */
    ino_t         st_ino;      /* inode */
    mode_t        st_mode;     /* protection */
    nlink_t       st_nlink;    /* number of hard links */
    uid_t         st_uid;      /* user ID of owner */
    gid_t         st_gid;      /* group ID of owner */
    dev_t         st_rdev;     /* device type (if inode device) */
    off_t         st_size;     /* total size, in bytes */
    blksize_t     st_blksize;  /* blocksize for filesystem I/O */
    blkcnt_t      st_blocks;   /* number of blocks allocated */
    time_t        st_atime;    /* time of last access */
    time_t        st_mtime;    /* time of last modification */
    time_t        st_ctime;    /* time of last status change */
};
```

1.1.3 stat() - example

```
#include <sys/types.h>
#include <sys/stat.h>
#include <dirent.h>
#include <pwd.h>
#include <grp.h>
#include <time.h>
#include <locale.h>
#include <langinfo.h>
#include <stdio.h>
#include <stdint.h>

struct dirent  *dp;
struct stat     statbuf;
struct passwd  *pwd;
struct group   *grp;
struct tm      *tm;
char            datestring[256];
...
/* Loop through directory entries. */
while ((dp = readdir(dir)) != NULL) {

    /* Get entry's information. */
    if (stat(dp->d_name, &statbuf) == -1)
        continue;
```

1.1.3 stat() – example (cont.)

```
    /* Print out type, permissions, and number of links. */
    printf("%10.10s", sperm (statbuf.st_mode));
    printf("%4d", statbuf.st_nlink);

    /* Print out owner's name if it is found using getpwuid(). */
    if ((pwd = getpwuid(statbuf.st_uid)) != NULL)
        printf(" %-8.8s", pwd->pw_name);
    else
        printf(" %-8d", statbuf.st_uid);

    /* Print out group name if it is found using getgrgid(). */
    if ((grp = getgrgid(statbuf.st_gid)) != NULL)
        printf(" %-8.8s", grp->gr_name);
    else
        printf(" %-8d", statbuf.st_gid);

    /* Print size of file. */
    printf(" %9jd", (intmax_t)statbuf.st_size);

    tm = localtime(&statbuf.st_mtime);

    /* Get localized date string. */
    strftime(datestring, sizeof(datestring), nl_langinfo(D_T_FMT), tm);

    printf(" %s %s\n", datestring, dp->d_name);
}
```

1.2.1 取得文件类型

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。
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1 5章将用套接口进行进程间的通信。
7. **符号连接(symbolic link)** 这种文件指向另一个文件。

1.2.2 取得文件类型-检测宏

- 可以用以下的宏确定文件类型。这些宏的参数都是struct stat结构中的st_mode成员。

```
S_ISREG(m)  is it a regular file?

S_ISDIR(m)  directory?

S_ISCHR(m)  character device?

S_ISBLK(m)  block device?

S_ISFIFO(m) fifo?

S_ISLNK(m)  symbolic link? (Not in POSIX.1-1996.)

S_ISSOCK(m) socket? (Not in POSIX.1-1996.)
```

1.2.3 取得文件类型-st_mode定义(参考)


```
S_IFMT     0170000   bitmask for the file type bitfields
S_IFSOCK   0140000   socket
S_IFLNK    0120000   symbolic link
S_IFREG    0100000   regular file
S_IFBLK    0060000   block device
S_IFDIR    0040000   directory
S_IFCHR    0020000   character device
S_IFIFO    0010000   fifo
S_ISUID    0004000   set UID bit
S_ISGID    0002000   set GID bit (see below)
S_ISVTX    0001000   sticky bit (see below)
S_IRWXU    00700     mask for file owner permissions
S_IRUSR    00400     owner has read permission
S_IWUSR    00200     owner has write permission
S_IXUSR    00100     owner has execute permission
S_IRWXG    00070     mask for group permissions
S_IRGRP    00040     group has read permission
S_IWGRP    00020     group has write permission
S_IXGRP    00010     group has execute permission
S_IRWXO    00007     mask for permissions for others (not in group)
S_IROTH    00004     others have read permission
S_IWOTH    00002     others have write permisson
S_IXOTH    00001     others have execute permission
```

1.2.4 取得文件类型-example

FreeBSD下的test程序源码片段：

```
static int
filstat(char *nm, enum token mode)
{
        struct stat s;

        if (mode == FILSYM ? lstat(nm, &s) : stat(nm, &s))
                return 0;

        switch (mode) {
        case FILRD:
                return (eaccess(nm, R_OK) == 0);
        case FILWR:
                return (eaccess(nm, W_OK) == 0);
        case FILEX:
                /* XXX work around eaccess(2) false positives for superuser */
                if (eaccess(nm, X_OK) != 0)
                        return 0;
                if (S_ISDIR(s.st_mode) || geteuid() != 0)
                        return 1;
                return (s.st_mode & (S_IXUSR | S_IXGRP | S_IXOTH)) != 0;
        case FILEXIST:
                return (eaccess(nm, F_OK) == 0);
        case FILREG:
                return S_ISREG(s.st_mode);
        case FILDIR:
                return S_ISDIR(s.st_mode);
        case FILCDEV:
                return S_ISCHR(s.st_mode);
        case FILBDEV:
                return S_ISBLK(s.st_mode);
        case FILFIFO:
                return S_ISFIFO(s.st_mode);
        case FILSOCK:
                return S_ISSOCK(s.st_mode);
```

1.3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

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(可以用setuid()函数设置实际用户ID和有效用户ID)。
- **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**。
- **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**，在进程控制部分介绍setuid函数时，将说明这两个保存值的作用。

1.3设置-用户-ID和设置-组-ID – 进一步的解释(参考)

- 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struct 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。

```
[yjs@boss ~]$ ls -alh /usr/bin/passwd
-r-s--x--x  1 root root 21K Jun 17  2005 /usr/bin/passwd
[yjs@boss ~]$
```

1.4 文件存取许可权限

① 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。
② 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。
③ 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。
④ 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。
⑤ 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。
⑥ 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。
⑦ 如果用6个exec函数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)。

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

1.5 新文件和目录的所有权

- 在使用open()或creat()创建新文件时，或者使用mkdir()创建一个新目录时，新创建的文件/目录的所有权的规则:
  - 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:
    - 新文件的组ID可以是进程的有效组ID。
    - 新文件的组ID可以是它所在目录的组ID。

1.6.1 access()

- 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。

```
#include <unistd.h>

int access(const char *path, int amode);
```

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，这就可能需要测试其存取能力，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。

```
#include <unistd.h>
...
int result;
const char *filename = "/tmp/myfile";

result = access (filename, F_OK);
```

1.6.2 access()-example

```
#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);
}
```

1.7 umask()

umask()函数为进程设置文件方式创建屏蔽字，并返回以前的值。

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

mode_t umask(mode_t mask);
```

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，在open()/creat()/mkdir()中都有一个参数mode，它指定了新文件的存取许可权位，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(即取反码)。

新创建的文件或者目录的权限为为permission & ～umask。

UNIX系统提供的umask命令的作用于umask()函数调用的作用一致。

1.8.1 chmod()/fchmod()

以下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 *path, mode_t mode);
int fchmod(int fildes, mode_t mode);
```

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。

UNIX系统提供的chmod命令的作用于chmod()函数调用的作用一致。

1.8.2 chmod()-mode参数(参考)

```
S_ISUID   04000 set user ID on execution

S_ISGID   02000 set group ID on execution

S_ISVTX   01000 sticky bit

S_IRUSR (S_IREAD)
          00400 read by owner

S_IWUSR (S_IWRITE)
          00200 write by owner

S_IXUSR (S_IEXEC)
          00100 execute/search by owner

S_IRGRP   00040 read by group

S_IWGRP   00020 write by group

S_IXGRP   00010 execute/search by group

S_IROTH   00004 read by others

S_IWOTH   00002 write by others

S_IXOTH   00001 execute/search by others
```

以上参数在<sys/stat.h>头文件中定义。

1.8.3 chmod()-examples

下面的例子给owner/group/other全部设置了读权限:

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

const char *path;
...
chmod(path, S_IRUSR|S_IRGRP|S_IROTH);
```

下面的例子给owner设置了读/写/执行权限，group/other没有任何权限：

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

const char *path;
...
chmod(path, S_IRWXU);
```

1.8.4 chmod()-examples

下面的例子给owner设置了读+写+执行权限，group/other设置了读权限:

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

#define CHANGEFILE "/etc/myfile"
...
chmod(CHANGEFILE, S_IRWXU|S_IRGRP|S_IXGRP|S_IROTH);
```

下面的例子给owner/group设置了读+写+执行权限，other设置了写权限，并用stat()来验证:

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

int status;
struct stat buffer
...
chmod("home/cnd/mod1", S_IRWXU|S_IRWXG|S_IROTH|S_IWOTH);
status = stat("home/cnd/mod1", &buffer;);
```

1.9 chown()/fchown()/lchown()

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 *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);
```

- 三个函数返回值：若成功则为0，若出错则为-1
- 三个函数的用途：
  - chown()处理文件名指向的文件
  - fchown()通过文件描述符处理打开的文件
  - lchown()处理符号连接本身
- 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。

1.10 文件长度

- struct 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。
  - 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。
  - 对于符号连接，文件长度是在文件名中的实际字节数。

1.11 文件截短

如果需要在文件尾端处截去一些数据以缩短文件，可以调用函数truncate()和ftruncate()。

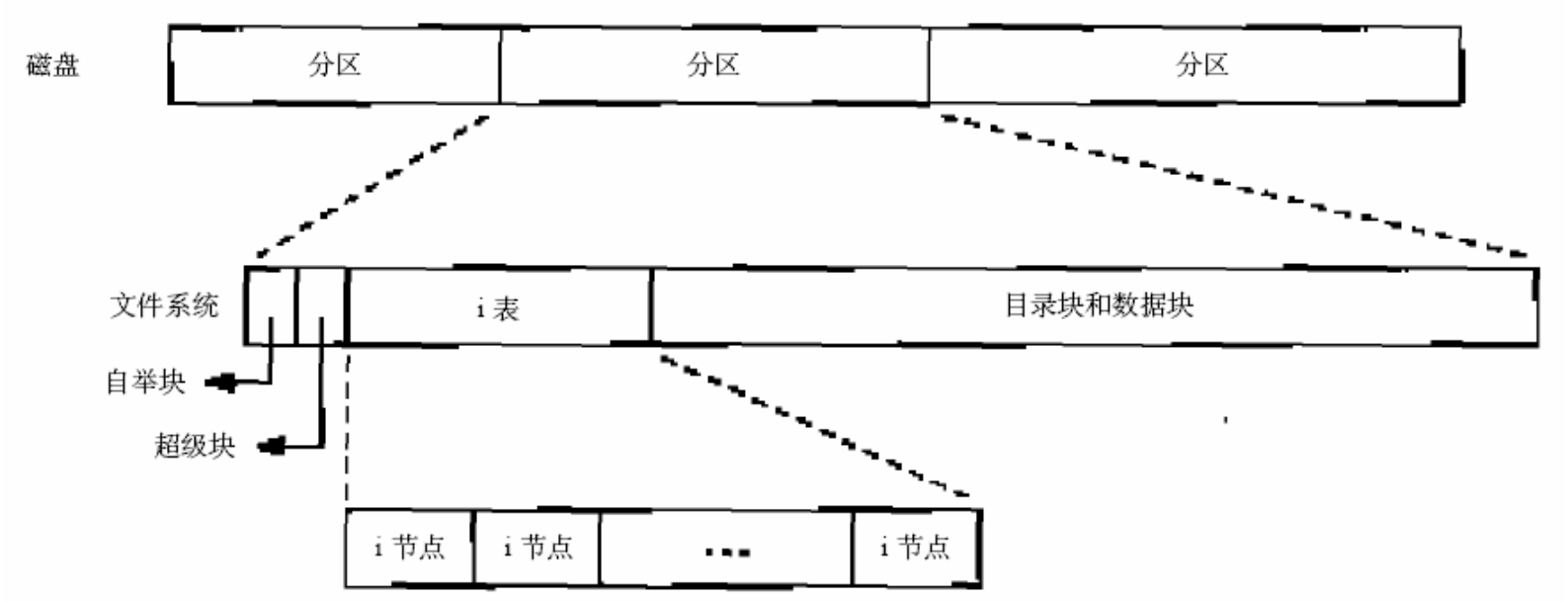
```
#include <unistd.h>
#include <sys/types.h>

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

- 两个函数返回值：若成功则为0，若出错则为-1，并设置errno
- 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)。
- 将一个文件的长度截短为0是一个特例，在open()/creat()调用时，设定O_TRUNC标志可以做到这一点。

2. 文件操作

2.1 传统文件系统介绍(供学员参考)

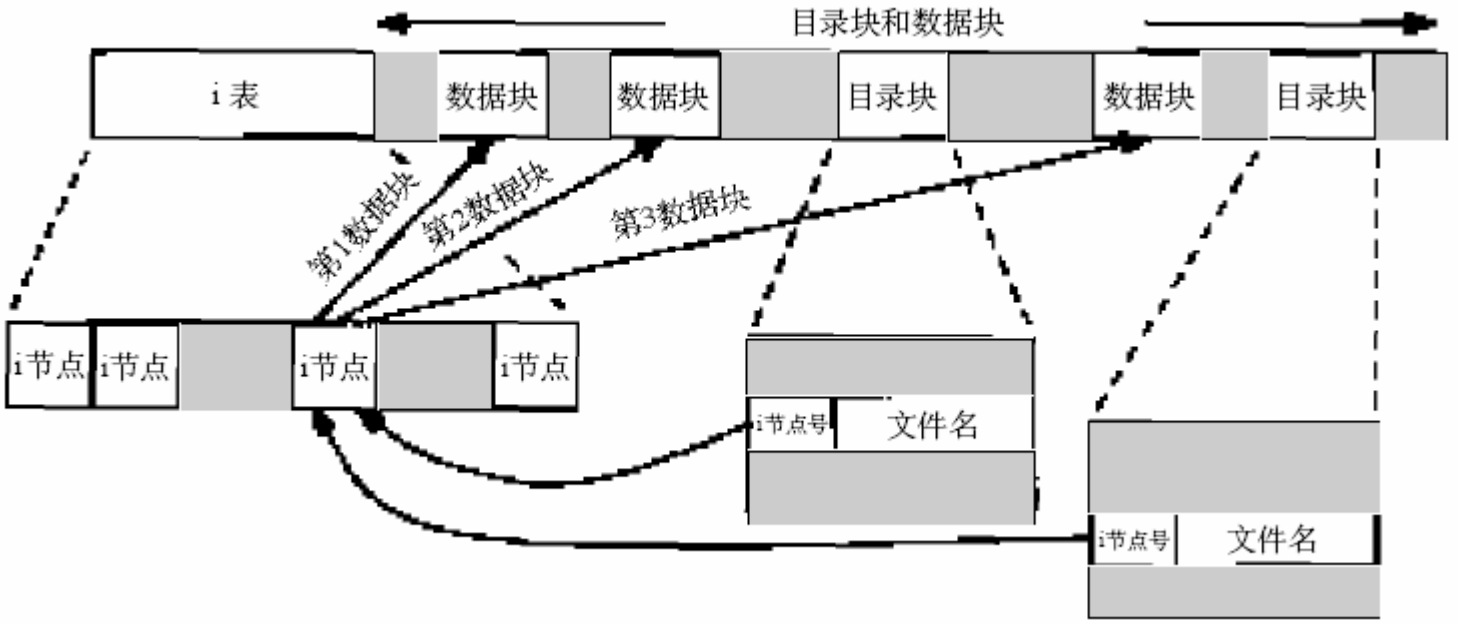
传统的UNIX系统V文件系统，这种类型的文件系统可以回溯到V7，我们可以把一个磁盘分成一个或多个分区，每个分区可以包含一个文件系统：



磁盘、分区和文件系统

2.1 传统文件系统介绍(供学员参考)

忽略自举块和超级块情况下，更仔细地观察文件系统：



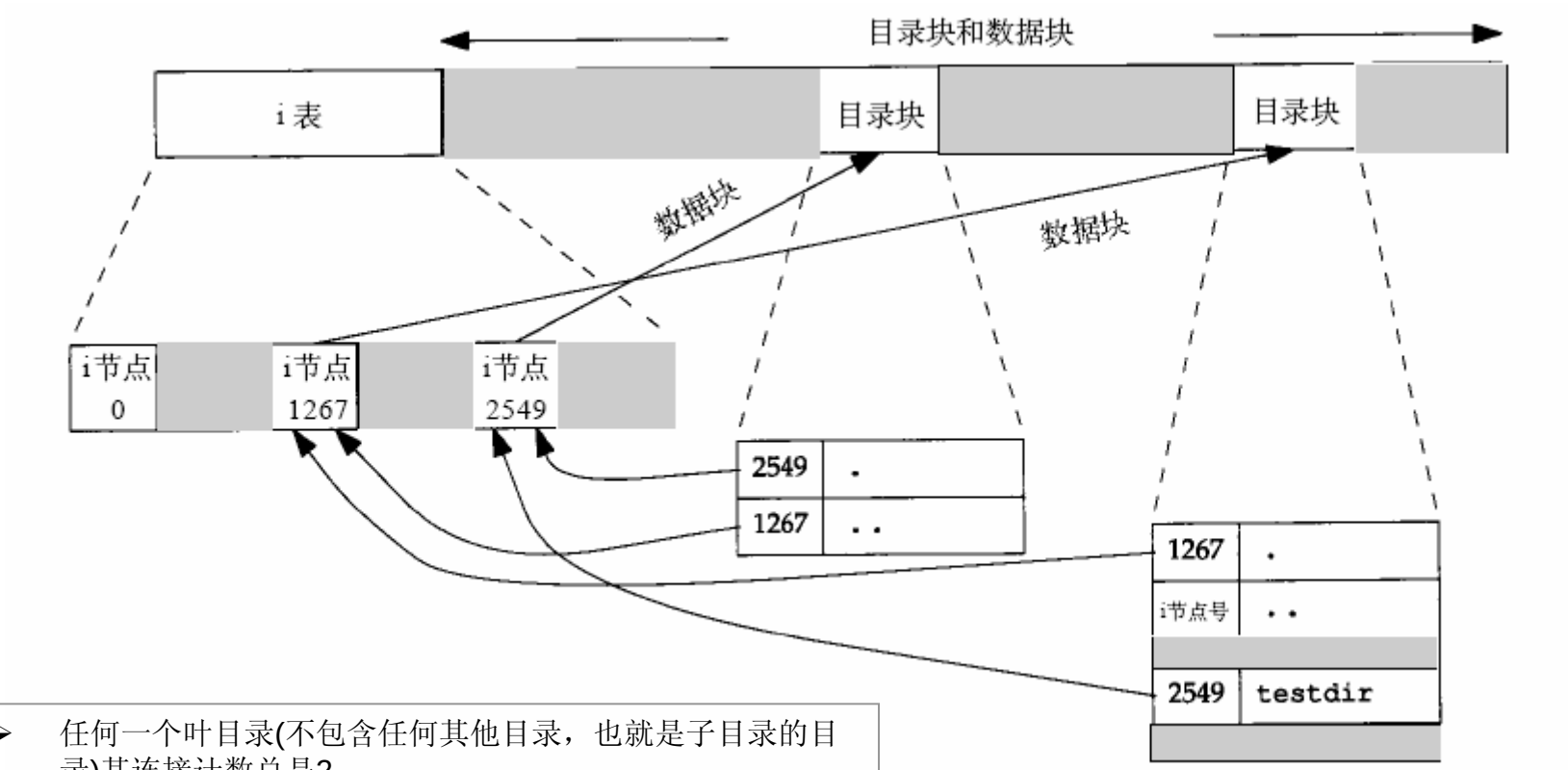
较详细的文件系统

2.1 传统文件系统介绍(供学员参考)

- 在图中有两个目录项指向同一i-node。每个i-node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)。
- 符号连接(symbolic link)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。符号连接i-node中的文件类型是S_IFLNK，于是系统知道这是一个符号连接。
- i-node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。struct stat结构中的大多数信息都取自i-node。只有两项数据存放在目录项中：文件名和i-node编号数。i-node编号数的数据类型是ino_t。
- 因为目录项中的i-node编号数指向同一文件系统中的i-node，所以不能使一个目录项指向另一个文件系统的i-node。
- 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-node的新目录项，并删除老的目录项。

2.1 传统文件系统介绍(供学员参考)

一个文件系统的某个部分(实例):



- 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
- 任何一个非叶目录(该目录下还有子目录)，其连接数≥3

2.2 link()/unlink()/remove()/rename()

- link()
- unlink()
- remove()
- rename()

2.2.1 link()

创建一个向现存文件连接(即通常所说的硬连接)的方法是使用link()函数:

```
#include <unistd.h>

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

- 函数调用成功则返回0，若出错则返回-1，并设置errno。
- 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
- 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。

2.2.1 link()-examples

example 1:

```
#include <unistd.h>

char *path1 = "/home/cnd/mod1";
char *path2 = "/modules/pass1";
int   status;
...
status = link (path1, path2);
```

example 2:

```
#include <unistd.h>

#define LOCKFILE "/etc/ptmp"
#define PASSWDFILE "/etc/passwd"
#define SAVEFILE "/etc/opasswd"
...
/* Save current password file */
link (PASSWDFILE, SAVEFILE);

/* Remove current password file. */
unlink (PASSWDFILE);

/* Save new password file as current password file. */
link (LOCKFILE,PASSWDFILE);
```

2.2.2 unlink()

删除一个文件引用(即删除一个现存的目录项)，可以调用unlink()函数:

```
#include <unistd.h>

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

- 函数调用成功则返回0，若出错则返回-1，并设置errno。
- 函数删除目录项，并将由*path*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。
- 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。
  - ✓ 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*path*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*path*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()

2.2.2 unlink() – example1

```
#include <unistd.h>

char *path = "/modules/pass1";
int   status;
...
status = unlink(path);
```

2.2.2 unlink() – example2

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

#define LOCKFILE "/etc/ptmp"

int pfd;  /* Integer for file descriptor returned by open call. */
FILE *fpfd;  /* File pointer for use in putpwent(). */
...
/* Open password Lock file. If it exists, this is an error. */
if ((pfd = open(LOCKFILE, O_WRONLY| O_CREAT | O_EXCL, S_IRUSR
    | S_IWUSR | S_IRGRP | S_IROTH)) == -1)  {
    fprintf(stderr, "Cannot open /etc/ptmp. Try again later.\n");
    exit(1);
}

/* Lock file created; proceed with fdopen of lock file so that
   putpwent() can be used.
 */
if ((fpfd = fdopen(pfd, "w")) == NULL) {
    close(pfd);
    unlink(LOCKFILE);
    exit(1);
}
```

2.2.3 remove()

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 *path);
```

- 函数调用成功返回0，出错返回-1，并设置errno
- 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。

Example:

```
#include <stdio.h>

int status;
...
status = remove("/home/cnd/old_mods");
```

2.2.4 rename()

文件或目录用rename()函数更名:

```
#include <stdio.h>

int rename(const char *old, const char *new);
```

- 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。
- 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。
- 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。

2.2.4 rename()-example

```
#include <stdio.h>

int status;
...
status = rename("/home/cnd/mod1", "/home/cnd/mod2");
```

2.3 符号连接

- 符号连接(symbolic link)是对一个文件的间接指针，它与硬连接有所不同，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：
  - 硬连接通常要求连接和文件位于同一文件系统中，
  - 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。
- 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。

2.3 符号连接(跟随关系)

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	•	

2.4 symlink()/readlink()

- symlink()函数创建了一个指向*actualpath(path1)*的新目录项*sympath(path2)*，在创建此符号连接时，并不要求*actualpath(path1)*已经存在

```
#include <unistd.h>

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

- readlink()函数提供了打开该连接本身，并读该连接中的名字的功能(不能使用open()/read()/close()方法，因为open函数跟随符号连接)。

```
#include <unistd.h>

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

- openlink()函数组合了open(), read()和close()的所有操作。
- 如果openlink()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('\0')字符终止。如果出错则返回-1，并设置errno。

2.5 文件的时间

对每个文件保持有三个时间字段:

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

2.5 文件的时间

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		•	•				

2.6 utime()

utime()用于修改文件的存取时间和修改时间:

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

int utime(const char *filename, const struct utimbuf *buf);

#include <sys/time.h>

int utimes(char *filename, struct timeval *tvp);
```

参数：

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

- 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：
  - ✓ 进程的有效用户ID必须等于该文件的所有者ID，
  - ✓ 进程对该文件必须具有写许可权。
- ✓ 如果*times*是非空指针，则存取时间和修改时间被设置为*t i m e s*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。
- ✓ 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

3. 目录操作

3.1 mkdir()/rmdir()

创建目录:

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

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

删除目录:

```
#include <unistd.h>

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

3.2 opendir()/readdir()/closedir()

打开目录:

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

#include <dirent.h>

int closedir(DIR *dir);
```

关闭目录:

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

#include <dirent.h>

int closedir(DIR *dir);
```

3.2 opendir()/readdir()/closedir()

读目录:

```
#include <unistd.h>
#include <linux/dirent.h>
#include <linux/unistd.h>

_syscall3(int, readdir, uint, fd, struct dirent *, dirp, uint, count);

int readdir(unsigned int fd, struct dirent *dirp, unsigned int count);
```

参数定义:

```
  struct dirent
  {
      long d_ino;                 /* inode number */
      off_t d_off;                /* offset to this dirent */
      unsigned short d_reclen;    /* length of this d_name */
      char d_name [NAME_MAX+1];   /* file name (null-terminated) */
  }
```

3.3 rewinddir()

重设目录操作位置回头部：

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

#include <dirent.h>

void rewinddir(DIR *dir);
```

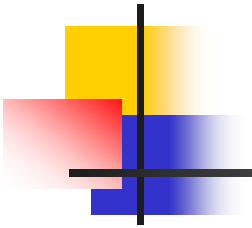
3.4 chdir()/fchdir()/getcwd()

调用chdir()或fchdir()函数可以更改当前工作目录:

```
#include <unistd.h>

int chdir(const char *path);
int fchdir(int fd);
```

获取当前工作目录的绝对路径名:

```
#include <unistd.h>

char *getcwd(char *buf, size_t size);
char *get_current_dir_name(void);
char *getwd(char *buf);
```



Let's DO it!

Thanks for listening!

☺