# File and Directories

System Programming

2019 여름 계절학기

한양대학교 공과대학 컴퓨터소프트웨어학부
홍석준

## I. Contents

- ❑ **File attributes handling**
  - – `stat`
- ❑ **Unix file system structure and symbolic links**
- ❑ **Directory operations**

## I. stat, fstat, and lstat

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

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

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

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

- ❑ **`stat/fstat` returns a structure of information about the named file.**
- ❑ **`lstat` returns information about the symbolic link, not the file referenced by the symbolic link.**

## I. stat, fstat, and lstat

```
struct stat {
   mode_t    st_mode; /* file type & mode (permission) */
   ino_t     st_ino;  /* i-node number (serial number) */
   dev_t     st_dev;  /* device number (file system) */
   dev_t     st_rdev; /* device number for special files */
   nlink_t   st_nlink;  /* number of links */
   uid_t     st_uid;  /* user ID of owner */
   gid_t     st_gid;  /* group ID of owner */
   off_t     st_size; /* size in bytes, for regular files */
   time_t    st_atime;  /* time of last access */
   time_t    st_mtime;  /* time of last modification */
   time_t    st_ctime;  /* time of last file status change */
   long     st_blksize;/* best I/O block size */
   long     st_blocks; /* no. of disk blocks allocated */
};
```

## I. File Types

- **Encoded in the `st_mode` member of the `stat` structure**
  - Regular file
  - Directory file – pairs of (file name, pointer to information on the file)
  - Character special file, e.g. tty
  - Block special file, e.g. disk devices
  - FIFO – named pipe
  - Socket – used for network communication between processes
  - Symbolic link – A type of file pointing to another file
- **Figure 4.3**

# Figure 4.3

```
#include "apue.h“
int main(int argc, char *argv[])
{  int               i;
   struct stat       buf;
   char              *ptr;

   for (i = 1; i < argc; i++) {
     printf("%s: ", argv[i]);
     if (lstat(argv[i], &buf) < 0) {
       err_ret("lstat error");
       continue;
     }
     if (S_ISREG(buf.st_mode))
                     ptr = "regular";
     else if (S_ISDIR(buf.st_mode))
                     ptr = "directory";
     else if (S_ISCHR(buf.st_mode))
                     ptr = "character special";
     else if (S_ISBLK(buf.st_mode))
                     ptr = "block special";
     else if (S_ISFIFO(buf.st_mode))
                     ptr = "fifo";
     else if (S_ISLNK(buf.st_mode))
                     ptr = "symbolic link";
     else if (S_ISSOCK(buf.st_mode))
                     ptr = "socket";
     else
                     ptr = "** unknown mode **";
     printf("%s\n", ptr);
   }
   exit(0);
}
```

## I.Set-User-ID and Set-Group-ID

- **User IDs and group IDs associated with each process**
  - real user ID, real group ID (`st_uid & st_gid`)
  - effective user ID, effective group ID, supplementary group IDs
  - saved set-user-ID, saved set-group-ID
- ***set-user-ID*** **bit and** ***set-group-ID*** **bit in `st_mode`**
  - *When this file is executed, set the effective user/group ID of the process to be the owner/group of the file.*
  - As an example, `passwd(1)` is a set-user-ID program.

## I. File Access Permissions

- **The nine file access permission bits from `<sys/stat.h>`**

| `st_mode` mask | Meaning |
|---|---|
| S_IRUSR | User-read |
| S_IWUSR | User-write |
| S_IXUSR | User-execute |
| S_IRGRP | Group-read |
| S_IWGRP | Group-write |
| S_IXGRP | Group-execute |
| S_IROTH | Other-read |
| S_IWOTH | Other-write |
| S_IXOTH | Other-execute |

7

u+x

755

111101101

- `$ ls –l foo bar`

```
-rwxr-xr-x    1 stevens   0 Nov 16 16:23 bar
-rw-r--r--    1 stevens   0 Nov 16 16:23 foo
```

110100100

- `chmod(1)`, e.g. `chmod g+w bar`

644

## I. File Access Permissions

- **Directory**
  - R – to obtain a list of all the file names in the dir.
  - W – to create/delete a file in the directory, both X and W are necessary.
  - X - to pass through the directory comprising a pathname (e.g., execute permission in /, /usr, and /usr/include to open /usr/include/stdio.h), also called "**search bit**".
- **File**
  - R – O_RDONLY and O_RDWR for the open function
  - W – O_WRONLY, O_RDWR, O_TRUNC
  - X – exec functions
- **Q: To create a new file in a directory, which permission(s) do we need to have?**
- **Q: To delete an existing file in a dir, which permission(s)?**

## I. File Access Permissions

- ❑ user ID and group ID of the file
- ❑ effective user ID, effective group ID, and supplementary group IDs of the process

1. if the effective user ID is 0(super user), …
2. if the effective user ID equals the user ID, …
3. if the effective group ID or one of supplementary group IDs equals the group ID, …
4. if the appropriate other access permission bit is set, …

- ❑ Q: what if permissions of ‘a.out’ are as follows?:
  - -rw-rwx-r-x a.out

## ~~I.~~ Ownership of New Files and Directories

- ❑ **The user ID of a new file is set to the effective user ID of the process.**
- ❑ **POSIX.1 options for the group ID of a new file**
  - The effective group ID of the process
  - The group ID of the containing directory
    - FreeBSD, Mac OS X 10.3
    - set-group-ID bit (Linux, Solaris)

자세히 X

## I.access Function

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

**int access(const char *_pathname_, int _mode_);**

- **mode: R_OK, W_OK, X_OK, and F_OK**
- **Accessibility test based on the real user ID/group ID to verify that the real user can access the given file.**
  - not the user ID of a process?
- **Figure 4.8**

# Figure 4.8

```
#include "apue.h“
#include <fcntl.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);
}
```

## I.access Function

```
$ ls –l a.out
-rwxrwxr-w 1 sar      15945  Nov 30 12:10 a.out
$ ./a.out a.out
read access OK
open for reading OK
$ ls –l /etc/shadow
-r-------- 1 root     1315   Jul 17 2002 /etc/shadow
$ ./a.out /etc/shadow
access error for /etc/shadow: Permission denied
open error for /etc/shadow: Permission denied
$ su                          become superuser
Passwd:                       enter superuser password
$ chown root a.out            change file's user ID to root
$ chmod u+s a.out             and turn on set-user-ID bit
$ ls –l a.out                 check owner and SUID bit
-rwsrwxr-x 1 root      15945  Nov 30 12:10  a.out
$ exit                        go back to normal user
$ ./a.out  /etc/shadow
access error for /etc/shadow: Permission denied
open for reading OK
```

## I. umask Function

```
#include <sys/stat.h>
mode_t umask(mode_t cmask);
```

- It sets the file mode creation mask for the process and returns the previous value.
- Any bits that are *on* in the file mode creation mask are turned *off* in the file's *mode*.
- Figure 4.9

```
$ umask                 first print the current file mode creation mask
002
$ a.out
$ ls –l foo bar
-rw-------  1   sar  0 Nov 16 16:23 bar
-rw-rw-rw-  1   sar  0 Nov 16 16:23 foo
$ umask                 see if the file mode creation mask changed
002
```

# Figure 4.9

```
#include "apue.h“
#include <fcntl.h>
#define RWRWRW (S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH)
                   1  1  0  |  1  1  0  |  1  1  0

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

  if (creat("bar", RWRWRW) < 0)
    err_sys("creat error for bar");

  exit(0);
}
```

# I. chmod and fchmod Functions

**#include <sys/stat.h>**
**int chmod(const char *_pathname_, mode_t _mode_);**
**int fchmod(int _filedes_, mode_t _mode_);**

- Figure 4.12

umask

```
$ ls –l foo bar
-rw------- 1 sar … bar
-rw-rw-rw- 1 sar … foo
$ a.out
$ ls –l foo bar
  ???
```

| mode | Description |
|---|---|
| S_ISUID<br>S_ISGID<br>S_ISVTX | set-user-ID on execution<br>set-group-ID on execution<br>saved-text (sticky bit) |
| S_IRWXU<br>S_IRUSR<br>S_IWUSR<br>S_IXUSR | read, write, and execute by user (owner)<br>read by user (owner)<br>write by user (owner)<br>execute by user (owner) |
| S_IRWXG<br>S_IRGRP<br>S_IWGRP<br>S_IXGRP | read, write, and execute by group<br>read by group<br>write by group<br>execute by group |
| S_IRWXO<br>S_IROTH<br>S_IWOTH<br>S_IXOTH | read, write, and execute by other (world)<br>read by other (world)<br>write by other (world)<br>execute by other (world) |

# Figure 4.12

```
#include "apue.h“
int main(void)
{
  struct stat  statbuf;  /* turn on set-group-ID and turn off group-execute */

  if (stat("foo", &statbuf) < 0)
    err_sys("stat error for foo");
  if (chmod("foo", (statbuf.st_mode & ~S_IXGRP) | S_ISGID) < 0)
    err_sys("chmod error for foo");

  if (chmod("bar", S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH) < 0)
    err_sys("chmod error for bar");

  exit(0);
}
```

# I. chown, fchown, and lchown

```
#include <unistd.h>

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

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

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

- ❑ To change the user/group ID of a file.
- ❑ `lchwon` changes the owners of the symbolic link itself.

## I. `chown`, `fchown`, and `lchown`

- **If `_POSIX_CHOWN_RESTRICTED` is in effect,**
  - Only a superuser process can change the user ID of the file;
  - A nonsuperuser process can change the group ID of the file if
    - the process owns the file (the *owner* equals to the user ID of the file), and
    - the *group* equals either the effective group ID of the process or one of the process's supplementary group IDs.
  - You can't change the user ID of other users' files
  - You can change the group ID of files that you own, but only to groups that you belong to.

## I. File Size

- **`st_size` of the `stat` structure: file size in bytes**
  - Regular file
  - Directory
    - a multiple of a number such as 16 or 512
  - Symbolic link
    - the actual number of bytes in the filename
    - For example,

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

- **`st_blksize`: the preferred block size for I/O**
- **`st_blocks`: the actual number of 512-byte blocks that are allocated**

# I.File Size

- **Holes in a File**

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

- **du reports the amount of disk space used by the file (272 512-byte blocks = 139,264 bytes)**
  - BSD: 1024 byte blocks
  - Solaris: 512 byte blocks

## I. File Truncation

```
#include <unistd.h>
int truncate(const char *pathname, off_t length);
int ftruncate(int filedes, off_t length);
```

- They truncate an existing file to `length` bytes.

Thank you for your attention !!

Q and A