Files and Directories

Hongik University
Eunsung Jung

Disclaimer: The slides are borrowed from many sources!

stat(2) family of functions

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

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

Returns: 0 if OK, -1 on error

- All these functions return extended attributes about the referenced file (in the case of symbolic links, lstat(2) returns attributes of the link, others return stats of the referenced file).
- Get attribute info using struct stat

Data structure: struct stat

What is this?
Try `ls –l /dev/sda`

```
struct stat {
    dev_t      st_dev;      /* device number (filesystem) */
    ino_t      st_ino;      /* i-node number (serial number) */
    mode_t     st_mode;     /* file type & mode (permissions) */
    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_blocks;   /* number of 512-byte* blocks allocated */
    long       st_blksize;  /* best I/O block size */
};
```

How to identify file types?

- The st_mode field of the struct stat encodes the type of file:
  - **regular** – most common, interpretation of data is up to application
  - **directory** – contains names of other files and pointer to information on those files. Any process can read, only kernel can write.
  - **character special** – used for character types of devices
  - **block special** – used for disk devices (typically). All devices are either character or block special.
  - **FIFO** – used for interprocess communication (sometimes called named pipe)
  - **socket** – used for network communication and non-network communication (same host).
  - **symbolic** link – Points to another file.
- Find out more in <sys/stat.h>, <asm/stat.h>.

How to identify file types?

Macro	Type of file
`S_ISREG()`	regular file
`S_ISDIR()`	directory file
`S_ISCHR()`	character special file
`S_ISBLK()`	block special file
`S_ISFIFO()`	pipe or FIFO
`S_ISLNK()`	symbolic link
`S_ISSOCK()`	socket

Figure 4.1 File type macros in `<sys/stat.h>`

Figure 4.3: Print type of file

```
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);
```

Figure 4.3: Results

```
$ ./a.out /etc/passwd /etc /dev/log /dev/tty \
> /var/lib/oprofile/opd_pipe /dev/sr0 /dev/cdrom
/etc/passwd: regular
/etc: directory
/dev/log: socket
/dev/tty: character special
/var/lib/oprofile/opd_pipe: fifo
/dev/sr0: block special
/dev/cdrom: symbolic link
```

- We can make a FIFO(named pipe)
  - mkfifo /tmp/testpipe

Set-User-ID and Set-Group-ID

- Every process has six or more IDs associated with it:

real user ID real group ID	who we really are
effective user ID effective group ID supplementary group IDs	used for file access permission checks
saved set-user-ID saved set-group-ID	saved by `exec` functions

Figure 4.5 User IDs and group IDs associated with each process

  - **Real user/group ID**: Normally, these values don't change during a login session.
  - **Effective user/group ID and supplementary group IDs**: determine our file access permissions. Normally, the effective user ID equals the real user ID, and the effective group ID equals the real group ID.
  - **Saved set-user-ID and saved set-group-ID**: contain copies of the effective user ID and the effective group ID, respectively, when a program is executed.

set-user-ID bit/set-group-ID bit

- Process user ID/group ID vs. File user ID/group ID
  - The real user ID can be obtained by calling getuid().
  - If set-user-ID bit and set-group-ID bit are set, effective user ID would be root.

File Access Permissions

- Nine permission bits for each file

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

Figure 4.6 The nine file access permission bits, from <sys/stat.h>

File Access Permissions

- r vs. x permission of a directory
  - x bit for a directory is the search bit.
    - For example, to open the file /usr/include/stdio.h , we need execute permission in the directory / , execute permission in the directory /usr , and execute permission in the directory /usr/include .
  - r bit for a directory is the read bit for the files under that directory.
    - Read permission lets us read the directory, obtaining a list of all the filenames in the directory.

File Access Permissions

- Summary
  - To open a file, need execute permission on each directory component of the path.
  - To open a file with O_RDONLY or O_RDWR, need read permission.
  - To open a file with O_WRONLY or O_RDWR, need write permission.
  - To use O_TRUNC, must have write permission.
  - To create a new file, must have write+execute permission for the directory.
  - To delete a file, need write+execute on directory, file doesn't matter
  - To execute a file (via exec family), need execute permission

File Access Permission Test by Kernel

1. If effective-uid == 0, grant access
2. If effective-uid == st_uid
   1) if appropriate user permission bit is set, grant access
   2) else, deny access
3. If effective-gid == st gid
   1) if appropriate group permission bit is set, grant access
   2) else, deny access
4. If appropriate other permission bit is set, grant access, else deny access

Ownership of new files and directories

- st_uid = effective-uid
- st_gid = ...either:
  - effective-gid of process.
  - gid of directory in which it is being created.

access(2)

```
#include <unistd.h>
int access(const char *pathname, int mode);
int faccessat(int fd, const char *pathname, int mode, int flag);
                                        Both return: 0 if OK, –1 on error
```

- Tests file accessibility on the basis of the real uid and gid. Allows setuid/setgid programs to see if the real user could access the file without it having to drop permissions to do so.

mode	Description
R_OK	test for read permission
W_OK	test for write permission
X_OK	test for execute permission

Figure 4.7 The *mode* flags for `access` function, from `<unistd.h>`

umask(2)

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

mode_t umask(mode_t numask);

                                    Returns: previous file mode creation mask
```

- umask(2) sets the file creation mode mask. Any bits that are on in the file creation mask are turned off in the file's mode.
- If a program needs to be able to ensure certain permissions on a file, it may need to turn off (or modify) the umask, which affects only the current process.

Example

disables all the group and other permission bits.

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

#define RWRWRW (S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH)

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);
}
```

Figure 4.9 Example of umask function

```
$ umask                                  first print the current file mode creation mask
002
$ ./a.out
$ ls -l foo bar
-rw-------  1 sar                         0 Dec  7 21:20 bar
-rw-rw-rw-  1 sar                         0 Dec  7 21:20 foo
$ umask                                  see if the file mode creation mask changed
002
```

chmod(2), lchmod(2) and fchmod(2):

```
#include <sys/stat.h>
int chmod(const char *pathname, mode_t mode);
int fchmod(int fd, mode_t mode);
int fchmodat(int fd, const char *pathname, mode_t mode, int flag);
```

All three return: 0 if OK, –1 on error

- Changes the permissions of an existing file.

chmod(2), lchmod(2) and fchmod(2):

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

Figure 4.11 The *mode* constants for chmod functions, from <sys/stat.h>

- Sticky Bit
  - If it was set for an executable program file, then the first time the program was executed, a copy of the program's text was saved in the swap area when the process terminated.
  - Next time, the program will be loaded into memory more quickly.
  - Not supported by Linux.

chown(2), lchown(2) and fchown(2)

```
#include <unistd.h>

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

                                    Returns: 0 if OK, -1 on error
```

- Changes st_uid and st_gid for a file. For BSD, must be superuser.
- owner or group can be -1 to indicate that it should remain the same.
- Non-superusers can change the st_gid field if both:
  - effective-user ID == st_uid and
  - owner == file's user ID and group == effective-group ID (or one of the supplementary group IDs)

File Size

- st_size member of the struct stat.
- For a regular file, a file size of 0 is allowed.
- For a symbolic link, the file size is
  - The number of bytes in the file name.
  - E.g. lrwxrwxrwx 1 root 7 Sep 25 07:14 lib -> usr/lib
- Holes in a File

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

truncate(2) and ftruncate(2)

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

Both return: 0 if OK, –1 on error

- Truncate an existing file to *length* bytes.