Lecture 4. Files and Directories

In Lectures 3 and 4 we discussed regular file I/O. In the following two lectures we'll discuss the filesystem and properties of a file.

The `stat()` Function

The `stat()` function describes the attributes of a file (owner, permissions etc.). The two versions differ by the way the file is specified (pathname vs. file descriptor), while the third one, when called on a symbolic link, returns information about the symbolic link rather than the file it points to.

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

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

All return 0 if OK, -1 if error.

Arguments:

- *`pathname`* — pathname to a file
- *`fd`* — file descriptor
- *`buf`* — a structure that the programmer must supply

The function succeeds on return when the file attributes that fill in the struct `buf` of type `stat`—a pointer to which is the second argument of `stat()`—matches one of the system defined file attributes. Below are the members of the `stat` structure, more or less, depending on the UNIX system:

```
struct stat
{
     mode_t    st_mode;  /* file type and mode (permissions) */
     ino_t     st_ino;   /* i-node number (serial number) */
     dev_t     st_dev;   /* device number (filesystem) */
     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; /* # of 512-byte blocks allocated */
};
```

Having discussed so far mainly about regular files (files and directories), we will now list all the file types under UNIX.

File types:

1. **Regular file** — a file containing data, either text or binary;
2. **Directory file** — a file containing the names of other files, and pointers to information about these files. Any process with read permission on the directory can read it, but only the kernel can write to a directory file;
3. **Character special file** — type of file used for certain devices on a system;
4. **Block special file** — type of file used for disk devices. All devices on a system are either character special files, or block special files;
5. **FIFO** — type of file used for interprocess communication (IPC) between processes. Sometimes called a pipe;
6. **Socket** — type of file used for network communication between processes, or for non-network communication between processes on a single host;
7. **Symbolic link** — type of file that points to another file.

The filetype is encoded in the `st_mode` members of the `stat` structure. It is one of the following macros (a macro is translated at compile time rather than run-time, and is simply a string substitution):

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

These are defined in /usr/include/sys/stat.h, with the mask S_IFMT defined as:

```
#define S_IFMT      0170000

#define S_ISBLK(m)  (((m) & S_IFMT)==S_IFBLK)
#define S_ISCHR(m)  (((m) & S_IFMT)==S_IFCHR)
#define S_ISDIR(m)  (((m) & S_IFMT)==S_IFDIR)
#define S_ISFIFO(m) (((m) & S_IFMT)==S_IFIFO)
#define S_ISREG(m)  (((m) & S_IFMT)==S_IFREG)
#define S_ISLINK(m) (((m) & S_IFMT)==S_IFLNK)
#define S_ISSOCK(m) (((m) & S_IFMT)==S_IFSOCK)
```

The following program takes file path names from the command line and returns the file type, using a call to `lstat()`:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.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)
          {
               printf("lstat() error\n");
               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);
     }
return 0;
}
```

To view the values of the S_IF*** constants we could use the following program:

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

int main(int argc, char *argv[])
{
     int i;
     printf("Length of a tab is %lu\n", strlen("\t"));
     char *string="|\tS_IFBLK\t\t|\t%o\t\t|\n";
     for (i=0; i<strlen("|\tS_IFBLK\t\t|\t%o\t|\n"); i++)
     {
          if (string[i]=='\t')
               printf("------");
          else
               printf("-");
     }
     printf("-\n");
     printf("|\tS_IFBLK\t\t|\t%o\t\t|\n", S_IFBLK);
     printf("|\tS_IFCHR\t\t|\t%o\t\t|\n", S_IFCHR);
     printf("|\tS_IFDIR\t\t|\t%o\t\t|\n", S_IFDIR);
     printf("|\tS_IFIFO\t\t|\t%o\t\t|\n", S_IFIFO);
     printf("|\tS_IFREG\t\t|\t%o\t\t|\n", S_IFREG);
     printf("|\tS_IFLNK\t\t|\t%o\t\t|\n", S_IFLNK);
     printf("|\tS_IFSOCK\t|\t%o\t\t|\n", S_IFSOCK);
     for (i=0; i<strlen("|\tS_IFBLK\t\t|\t%o\t|\n"); i++)
     {
          if (string[i]=='\t')
               printf("------");
          else
               printf("-");
     }
     printf("-\n");
return 0;
}
```

This produces the following output:

```
- - - - - - - - - - - - - - - - - - - - - - -
|    S_IFBLK         |    60000             |
|    S_IFCHR         |    20000             |
|    S_IFDIR         |    40000             |
|    S_IFIFO         |    10000             |
|    S_IFREG         |    100000            |
|    S_IFLNK         |    120000            |
|    S_IFSOCK        |    140000            |
- - - - - - - - - - - - - - - - - - - - - - -
```

C can only represent decimal, octal, and hexadecimal. Therefore, to explain the bitwise AND operation, we must convert from octal into binary. We get:

$$\mathbf{0}170000 = 170000_8 = \underbrace{1_2}_{001}\underbrace{7_2}_{111}\underbrace{0_2}_{000}\underbrace{0_2}_{000}\underbrace{0_2}_{000}\underbrace{0_2}_{000} = 001111000000000000_2$$

So, we want to calculate S_IFBLK & S_IFMT, they both need converted into binary:

$$\texttt{S\_IFMT} = \mathbf{0}170000_8 = 0011110000000000000_2$$
$$\texttt{S\_IFBLK}=\mathbf{0}60000_8 = 0011000000000000000_2$$

The bitwise operation will select all the 1 bits from S_IFBLK with the 1s from the mask:

$$\begin{array}{r} 0011110000000000000 \\ \otimes\ \underline{0011000000000000000} \\ 0011000000000000000 \end{array}$$

Set-User-ID and Set-Group-ID

Every process has the following associated IDs:

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

These are:

- the **real user ID** and **real group ID** — identify who's logged in. These fields are taken from the password file on login;
- the **effective user ID**, **effective group ID**, and supplementary group ID — identify who the process runs "as";
- the **saved set-user-ID** and **saved set-group-ID** contain copies of the effective user ID and the effective group ID when a program is executed. This is so that copies of the process obtained with `fork()` and `exec()` can have this info (who a process can run "as" is set with these bits).

The owner and the group owner of every file are specified by the `st_uid` and the `st_gid` of the `stat` structure. These are normally the real user ID and the real group ID. But the `st_mode` member of the `stat` structure, which we discussed earlier saying that it stores the file *type*, can also set the *file mode permissions* for that process to temporarily be those of the effective owner of the file, instead of the real user (i.e. you run as root, and the set-user-ID bit is set, you get the same file permissions, meaning you own the file).

For example, if the owner of a file is the superuser, then the effective user and group ID of that program file are set to the permissions of the superuser for the duration of the program. This happens, for instance, with the `passwd(1)` program, which enables a user to change his/her password. For the duration they use this program, users gain temporary access to the superuser's privilege to write the /etc/passwd or the /etc/shadow files.

File Permissions

The `st_mode` also stores the access permission bits for the file. There are 9 access permission bits for each file, divided into 3 categories:

`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

The `chmod(1)` command allows the user to specify `u` for user (owner), `g` for group, and `o` for other. The 3 categories (read, write, and execute) are used in different ways by different functions:

- The **execute permission** is needed to:
  - open a file by name. The execute permission is needed in each directory mentioned in the name. This is why the execute permission bit for a directory is called the *search bit*;
  - execute a file using one of the 6 `exec()` functions;

There is a difference between read and execute permission in a directory: the read permission allows you to read the directory, obtaining a list of the files in it; whereas the execute permission allows you to pass through the directory when it is a component of a pathname that you are trying to access. An executable file you try to run will have a directory path in which each upstream directory must be execute-enabled for you.

- The **read permission** for a file specifies whether we can open an existing file for reading: these are the O_RDONLY and O_RDWR flags for the `open()` function;

- The **write permission** specifies whether we can open an existing file for writing: O_WRONLY and O_RDWR for `open()`. This has to be on in order to use O_TRUNC;

- Both **write and execute permissions** are needed to:
  - create a new file;
  - delete an existing file;

The kernel performs the following tests on a file, when access is requested by a process:

1. If the effective user ID of the process is 0 (superuser), access is allowed to any file;
2. If the effective user ID of the process equals the owner ID of the file:
   (a) if the user has the type of access requested, access is allowed;
   (b) if not, access is denied;
3. If the effective group ID of the process or one of the supplementary group IDs of the process equals the group ID of the file:
   (a) if the group has the type of access requested, access is allowed;
   (b) if not, access is denied;

4. If the "other" access permission bit is set, access is allowed, if not, access is denied.

These 4 checks are performed by the kernel in sequence.

Ownership of New Files and Directories

The user ID of a new file is set to the effective user ID of the process:
1. The group ID of a new file can be the effective group ID of the process;
2. The group ID of a new file can be the group ID of the directory in which the file is being created. This way of setting group IDs upon creation ensures that group ownership of files and directories propagate down the hierarchy from that point.

The `access()` Function

When accessing a file using `open()`, the kernel verifies the effective user ID and effective group ID. To test accessibility based on the real user ID and real group ID, the `access()` function is used.

```
#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 if error.

Arguments:
- *`pathname`* — string pathname of the file;
- *`mode`* — one of, or the bitwise OR of any of, the values in the table below;
- *`fd`* — file descriptor;
- *`flag`* — if AT_EACCESS is set, the access test is done on the effective user and group IDs, not the real user and group IDs.

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

mode	Description
`R_OK`	test for read permission
`W_OK`	test for write permission
`X_OK`	test for execute permission
`F_OK`	test for existence of file

The `faccessat()` version can use relative paths:

- if *`fd`*=AT_FDCWD, the pathname is evaluated to a file path relative to *`fd`*;
- if *`fd`* has another file value, the pathname is evaluated to the file path of the open directory referenced by *`fd`*.

The following program shows the use of `access()`:

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

int main(int argc, char *argv[])
{
     if (argc!=2)
          printf("Usage: ./a.out <pathname>\n");
     if (access(argv[1], R_OK | X_OK)<0)
          printf("Access error for %s\n", argv[1]);
     else
          printf("Read access OK.\n");

     if (open(argv[1], O_RDONLY)<0)
          printf("Open error for %s\n\n", argv[1]);
     else
          printf("Open for reading OK.\n");
return 0;
}
```

On MacOS X, the password file is in /private/var/db/dslocal/nodes/Default/users/awise.plist. However, only root can `cd` into Default and beyond. So, in order to view the contents of that directory, you'd have to run:

```
$ sudo -s
$ cd /private/var/db/dslocal/nodes/Default/users/awise.plist
```

As root, these are the permissions on awise.plist:

```
bash-3.2# ls -la awise.plist
-rw-------  1 root  wheel  82211 Feb  4 22:38 awise.plist
```

Running the program on this file yields:

```
bash-3.2#    ./myaccess    /private/var/db/dslocal/nodes/Default/
users/awise.plist
Access  error  for  /private/var/db/dslocal/nodes/Default/users/
awise.plist
Open for reading OK.
```

If I jut ran the same command without the `sudo -s`, i.e. as me, I would have gotten no permissions:

```
adrianas-mbp:Lecture4 awise$ ./myaccess /private/var/db/dslocal/
nodes/Default/users
Access error for /private/var/db/dslocal/nodes/Default/users
Open error for /private/var/db/dslocal/nodes/Default/users
```

The `umask()` Function

The `access()` function examines the file permissions based on the 9 file permission bits. The `umask()` function sets the file creation mask which is associated with every file. It returns the previous value of the file creation mask.

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

mode_t umask(mode_t cmask);
```

Returns previous value of file mode creation mask.

Argument:

- *`cmask`* — the file creation mask we want to set the file to, as the bitwise OR of any of the nine file access permission bits (we inserted the table again):

`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

The following program sets the umask of file "foo" to `rw-------` and of file "bar" to `rw-rw-rw-`.

```
#include <fcntl.h> /* for creat() */
#include <sys/stat.h> /* for the file permission bits */
#include <stdio.h> /* for printf() */

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

int main(void)
{
     umask(0);
     if (creat("foo", RWRWRW)<0)
          printf("creat() error for \"foo\"\n");
     umask(S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH);
     if (creat("bar", RWRWRW)<0)
          printf("creat() error for \"bar\"\n");
return 0;
}
```

The following program lists the octal values of the file permission bits for each of the `st_mode` umask bits:

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

int main(void)
{
     printf("----------------------------------------\n");
     printf("|\tS_IRUSR\t\t|\t%o\t|\n", S_IRUSR);
     printf("|\tS_IWUSR\t\t|\t%o\t|\n", S_IWUSR);
     printf("|\tS_IXUSR\t\t|\t%o\t|\n", S_IXUSR);
     printf("|\tS_IRGRP\t\t|\t%o\t|\n", S_IRGRP);
     printf("|\tS_IWGRP\t\t|\t%o\t|\n", S_IWGRP);
     printf("|\tS_IXGRP\t\t|\t%o\t|\n", S_IXGRP);
     printf("|\tS_IROTH\t\t|\t%o\t|\n", S_IROTH);
     printf("|\tS_IWOTH\t\t|\t%o\t|\n", S_IWOTH);
     printf("|\tS_IXOTH\t\t|\t%o\t|\n", S_IXOTH);
     printf("----------------------------------------\n");
return 0;
}
```

The output of this program is the following octal permission bits table:

S_IRUSR	400
S_IWUSR	200
S_IXUSR	100
S_IRGRP	40
S_IWGRP	20
S_IXGRP	10
S_IROTH	4
S_IWOTH	2
S_IXOTH	1

For example, if we bitwise OR S_IRUSR with S_IRGRP, we get:

$$\texttt{S\_IRUSR} = \mathbf{0}400 = 400_8 = \underbrace{4_2}_{100}\underbrace{0_2}_{000}\underbrace{0_2}_{000} = 100000000_2$$

$$\texttt{S\_IRGRP} = \mathbf{0}040 = 040_8 = \underbrace{0_2}_{000}\underbrace{4_2}_{100}\underbrace{0_2}_{000} = 000100000_2$$

After the bitwise OR operation, we get:

$$\begin{array}{r} 100000000 \\ \oplus 000100000 \\ \hline 100100000 \end{array}$$

Because of the choice of octal value for each bit, in binary any bitwise OR operation expands into a binary value with 1s in all the "set" places.

The `chmod` and `fchmod` Functions

These two functions are used to change the file access permissions for a file.

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

int chmod(const char *pathname, mode_t mode);
int fchmod(int fd, mode_t mode);
```

Both return 0 if OK, -1 if error.

In order to change the permission bits of a file, the effective user ID of the process must be equal to or higher than the owner ID of the file.

The mode is the bitwise OR of the constants in the following table, a superset of the 9 file permission modes (including a bitwise AND of each 3 per user, group, and others), and of the set-user-ID, set-group-ID, and sticky bit:

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)

The following program turns on the execute permission on file “foo”:

```
#include <unistd.h>
#include <sys/stat.h> /* for the file permission bits */
#include <stdio.h> /* for printf() */

#define XXX (S_IXUSR | S_IXGRP | S_IXOTH)

int main(void)
{
     struct stat statbuf;
     if (stat("foo", &statbuf)<0)
```

```
          printf("stat() error for \"foo\"\n");
     /* turn on user execute, group execute, and others execute
*/
     if (chmod("foo", (statbuf.st_mode | XXX))<0)
          printf("chmod() error for \"foo\"\n");
return 0;
}
```

The Sticky Bit

This is the constant S_ISVTX (saved text). Historically, if it was set for an executable program file, a copy of the program's binary was saved in the swap area when first executed. This way it would load more quickly into memory when called the next time. Examples of such programs were the text editor and C compiler. The number of files that could be saved into the swap area was limited by the swap space. The swap was cleared only on reboot.

Nowadays, the sticky bit is used for directories. If set, a file in the respective directory can be removed or renamed only if the user has write permission for that directory and one of the following:

- user owns the file
- user owns the directory
- user is root

The `chown()`, `fchown()`, and `lchown()` Functions

The `chown()` family of functions are used to change the user ID and the group ID of a file.

```
#include <unistd.h>

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

All three return 0 if OK, -1 if error.

The third version of the function operates on a symbolic link, and changes the owner of the symbolic link itself, rather than the file it points to.

The `pathconf()` system call can query the <unistd.h> header file for the value of the _POSIX_CHOWN_RESTRICTED constant. If set, then:

1. Only a superuser process can change the user ID for the file;
2. A non-superuser process can change the group ID of a file if:
   - the process owns the file (effective user ID==user ID of the file);
   - owner is specified as -1 or equals the user ID of the file;
   - group equals either the effective group ID of the process or one of the supplementary group IDs of the process.

When _POSIX_CHOWN_RESTRICTED is on, you can't change the user ID of other people's files. And you can only change group IDs on files you own, but only to groups that you belong to.

Homework (due Tuesday, Feb-16-2016):

1. Write a small program that illustrates the use of the `chown()` system call on a file.
2. Write a program that displays all the information from the `stat` structure about a file in a "formatted" table, with the constant names on the left, and their binary values on the right. In particular, the `st_mode` member of the `stat` structure, which encodes the file type and the permissions mode, should be broken down into the values of its components, given that each bit has a meaning. List the meanings of each bit of `st_mode` and its values for your file. Convert into binary using the % operator on the octal value of `st_mode`, then test against the values obtained by selecting each bit using bitwise operations.

There are 16 bits for st_mode (no two leading 0 bits as in the mask examples):
- 1 for set-user-ID
- 1 for set-group-ID
- 1 for the sticky bit
- 4 for the file type
- 9 for access permissions