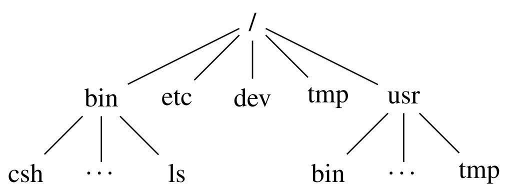
# UNIX File Hierarchy: Structure and Commands

The UNIX operating system organizes files into a tree structure with a root named by the character "/". An example of the directory tree is shown below.



All files have two types of names. An *absolute path name* is the name of the file that begins with a "/"; for example, in the above picture, the program *ls* has as its absolute path name */bin/ls* (because to find it, start at the root */*, then go to the directory *bin* in */*, then to *ls* in *bin*). A *relative path name* identifies a file relative to an arbitrary directory in the tree; for example, the relative path name of *ls* with respect to the directory *bin* is *ls*, because within the directory bin you need only look for the file ls. An easy way to keep the two types straight is to remember that an absolute path name always begins with a "/", and a relative path name never does.

Every directory has at least two subdirectories. The directory . refers to the directory containing that entry; in other words, in the above picture, */bin/.* and */bin* refer to the same directory. The . is most often used in contexts where the "current directory" is important, and not the specific name of the current directory. The directory .. refers to the parent directory; so */bin/..* is another way of saying */*, and */usr/bin/..* is the same as */usr*. This gives you a convenient way to get back up the file tree.

## File Types

There are several types of files in the file system; of these, you need to understand four different types.

An *ordinary file* (also called a *regular file*) may contain text, a program, or other data. It can be either an ASCII file, with each of its bytes being in the numerical range 0 to 127 (representing characters), or a binary file, whose bytes can be of all possible values 0 to 255.

A *directory* is a file that consist of directory entries for the files in that directory. There is one directory entry for each file. Each directory entry contains the name of the file and a pointer to the file.

A *device file* is a file that represents a device. The UNIX operating system represents physical devices (printers, terminals etc.) as filed so that the same I/O functions used to read and write regular files can be used to read from and write to these devices. Devices which do input and output a character at a time, such as terminals and printers, are represented as *character special device files*; devices which do input and output in chunks of characters (usually a multiple of 256 characters) are represented as *block special device files*.

Understanding *link files* requires you to know a little about how the system implements the concept of "file." The UNIX operating system associates a structure called an *inode* with each file; all characteristics of the file are stored there. Among things like access permissions, date and time of last access and modification, and owner, the inode identifies the data that makes up the file. The "pointer" to the file that a directory contains is simply the inode number of the file.

Two files are said to be *linked* if they refer to the same inode number. If the file *X* is linked to the file *Y*, then *X* is an alternate name for the file *Y*. For example, `cat X` will produce exactly the same effect as `cat Y`. There are two types of links. A *hard link* is simply an entry in a directory which contains an inode number that also is associated with another file name; so, using *X* and *Y* as examples, the directory entries would have two different names (one would be *X* and the other *Y*) but the inode numbers would be the same. A *symbolic link* (or *soft link*) is simply a file containing the name of the file it is linked to. The difference is that with a hard link, deleting one of the directory entries does not affect the other but with a symbolic link, deleting the file which is linked to makes the link useless. The command *ln*(1) creates both types of links; give it the `-s` option to get a symbolic link, and no options to get a hard link.

The other types depend upon the system you are using. The most common other types are *sockets* (which are used by processes to communicate with one another) and *FIFOs* (also called *named pipes*, these are very similar to sockets).

The command `ls -l` allows you to see the type of each file in a directory (among other things). The following is the output of the command `ls -alg` (the `-a` option means to list those files and directories whose names begin with a period, and the `-g` option means to show the group of the file):

```
drwxr-xr-x  6 ecs4005   student     1024 Apr 22 13:30 ./
drwxr-xr-x 74 root      student     1536 Mar 24 12:51 ../
-rw-------  1 ecs4005   student      188 Apr 13 15:53 .login
-rw-------  1 ecs4005   student        6 Mar 24 11:29 .logout
-rw-------  1 ecs4005   student      253 Apr 10 12:50 .xinitrc
-rw-r--r--  1 ecs4005   student      516 Apr 10 13:00 .twmrc
-rw-r--r--  1 ecs4005   student     1600 Apr 22 10:59 test2.out
```

The output is separated into 7 columns. The meaning of each column in the output is shown below:

| | |
|---|---|
| 1st column | type and protection mode of the file |
| 2nd column | number of links to that file (the original file is considered a link) |
| 3rd column | owner of the file |
| 4th column | group of the file |
| 5th column | size of file in bytes |
| 6th column | date and time of last modification |
| 7th column | name of the file |

## File Access Control

In the UNIX operating system, all files are protected using a simple access control mechanism so that the owner of a file can deny other users access to his or her files. The first column of the long directory list shows the access characteristics of a file. It is in a form of 10 flags.

```
drwxr-xr-x
```

The meanings of the characters in this column are shown below:

- The first character shows the type of the file:

| | |
|---|---|
| b | block special device |
| c | character special device |
| d | directory |
| l | symbolic link |
| p | FIFO (named pipe) |
| s | socket |
| - | regular file |

- The next three characters indicate the permissions the owner of the file has:

| | |
|---|---|
| r | as character 2 means the owner can read the file |
| w | as character 3 means the owner can write the file |
| x | as character 4 means the owner can execute the file |
| s | as character 4 means the program runs as if the owner executed it, regardless of who actually executed it (called ”the setuid bit;” don’t worry about this) |
| - | instead of any of the above means the owner can’t access the file as appropriate |

- The next three characters indicate the ways in which the members of the group (commonly called “the group”) to which the file belongs can access the file:

| | |
|---|---|
| r | as character 5 means the group can read the file |
| w | as character 6 means the group can write the file |
| x | as character 7 means the group can execute the file |
| s | as character 7 means the program runs as if the group executed it, regardless of who actually executed it (called ”the setgid bit;” don’t worry about this) |
| - | instead of any of the above means the members of the group can’t access the file as appropriate |

- The last three characters indicate the ways in which all users except members of the group and the owner can access the file:

| | |
|---|---|
| r | as character 8 means the group can read the file |
| w | as character 9 means the group can write the file |
| x | as character 10 means the group can execute the file |
| t | as character 10 means the program is not swapped out when it finishes (called ”the sticky bit;” don’t worry about this) |
| - | instead of any of the above means the user can’t access the file as appropriate |

In determining access, the system checks to see if you are the owner of the file ; if so, it uses characters 2-4. If not, it checks to see if you are a member of the group of the file; if so, it uses characters 5-7. If not, it uses characters 8-10.

You can change the access characteristics of a file using the command *chmod*(1), which we will talk about in a later section.

## Directory Related Commands

There are several commands affecting directories; here is a brief description and example of each. All are fully documented in section 1 of the UNIX Programmer’s Manual.

**cd *directory***

This changes the directory you’re in. If you omit *directory*, you get put in your home directory.

**pwd**

Display the name of the current working directory. As an example, let’s say your current working directory is */usr/home/groucho*. Then:

```
% pwd
/usr/home/groucho
```

**mkdir *directory***

Create *directory*. You must have write permission in directory’s parent. I strongly recommend you use only letters, numbers, and the dash “-” (anywhere other than the first character in the name), the plus sign “+” (again, anywhere other than the first character in the name), and the underscore “_” in your directory name. You can use other characters (in fact, anything except ‘/’, but doing so may cause problems.

**rmdir *directory***

Delete *directory*. Again, you need write permission in *directory*’s parent. Also, *directory* must contain no files except . and .. or the *rmdir* will fail.

**mv *old-directory new-directory***

Rename *old-directory* to *new-directory*. This can only be done if *old-directory* and *new-directory* are on the same file system (basically, if you’re moving a directory around in your account area, don’t worry about this; but you usually can’t move one of your directories to */tmp*, for example). As an example, let’s say you create a directory called *help*, decide to rename it *nohelp*, and finally delete it:

```
% ls
% mkdir help
% ls
help
% mv help nohelp
% ls
nohelp
```

**cp -r *old-directory new-directory***

Make a copy of *old-directory* named *new-directory*. Unlike *mv*, this works across file systems.

**rm -r *directory***

Delete *directory* and all of its contents. As with *rmdir*, you need to have write permission to the parent of *directory*, *directory* itself, and all of its subdirectories. ***Be very careful with this command as you cannot recover files and directories you’ve accidentally deleted!***

## File Related Commands

There are many UNIX commands dealing with files. Here, we go through some of the more common ones. Be aware each of these has lots of options this handout does not go into, so be sure to read the manual page if you want the full scoop!

**chmod *mode filename* or chmod *who op permissions filename***

This changes the access permissions of the file. It has two forms, illustrated above.

In the first form, you specify an *absolute mode*, which is an octal number constructed from the logical OR of any set of the following modes:

| | |
|---|---|
| 0400 | owner can read the file |
| 0200 | owner can write to the file |
| 0100 | for a file, owner can execute it; for a directory, owner can look for a file in it (search permission) |
| 0040 | members of the file's group can read the file |
| 0020 | members of the file's group can write to the file |
| 0010 | for a file, members of the file's group can execute it; for a directory, members of the directory's group can look for a file in it (search permission) |
| 0004 | all other users can read the file |
| 0002 | all other users can write to the file |
| 0001 | for a file, all other users can execute it; for a directory, all other users can look for a file in it (search permission) |

For example,

```
chmod 751 .login
```

makes the file *.login* readable, writeable, and executable by the owner, readable and executable by any members of .login's group, and executable only by everyone else.

The *chmod*(1) command allows you to specify the mode symbolically. A symbolic mode has the form

*who op permission*

where *who* is a combination of:

| | |
|---|---|
| u | owner (user) permissions |
| g | group permissions |
| o | others' permissions |
| a | all users; same as ugo |

*op* is one of:

| | |
|---|---|
| + | to add the permission |
| - | to remove the permission |
| = | to assign the permission explicitly (all other bits for that category, owner, group, or others, will be cleared) |

and *permission* is any combination of:

| | |
|---|---|
| r | read |
| w | write |
| x | execute |

Look at the previous permission modes of the *.login* file in the previous example. Executing

```
chmod u=rx,g-r,o+r .login
```

will change the owner's access to read and execute only (the writing permission is deleted), will delete the write permission for members of the group, and will add read permission for everyone else. Here's what the two commands would do; note we use

```
ls -al .login
```

to check file status:

```
% ls -al .login
-rwxr-x--x  1 ecs4005       188 Apr 13 15:53 .login
% chmod u=rx,g-r,o+r .login
% ls -al .login
-r-x--xr-x  1 ecs4005       188 Apr 13 15:53 .login
```

**touch *file***

This creates *file* (if it does not exist) or changes the time of last access and modification of *file* (if it does exist). It's not used very often, but sometimes it's just what you need to get that *makefile* just right.

```
% ls
% touch xyzzy
% ls -l
-rw-r--r--  1 ecs4005       0 Jan 11 15:53 xyzzy
```

and after 2 minutes . . .

```
% touch xyzzy
% ls -l
-rw-r--r--  1 ecs4005       0 Jan 11 15:55 xyzzy
```

**cp *old-file new-file***
**mv *old-file new-file***
**ln *old-file new-file***
**ln -s *old-file new-file***

These commands all have the same form, and are often grouped together. The first makes a copy of *old-file* called *new-file*, the second changes the name of *old-file* to *new-file*, the third creates a hard link named *new-file* to *old-file*, and the fourth creates a symbolic link named *new-file* to *old-file*.

**cp *old-file1 old-file2 . . . directory***
**mv *old-file1 old-file2 . . . directory***
**ln *old-file1 old-file2 . . . directory***
**ln -s *old-file1 old-file2 . . . directory***

These are alternate forms of the above commands. The first makes a copy of the file(s) named *old-file1 old-file2 ...* in the directory *directory*; the new files retain the old names (but are in *directory*). The second command moves the files *old-file1 old-file2 . . .* into *directory*, the third creates hard links named *directory/old-file1, directory/old-file2, . . .* to *old-file1, old-file2, . . .*; and the fourth creates symbolic links named *directory/old-file1, directory/old-file2, . . .* to *old-file1, old-file2, . . . .*

**rm -i *file1 file2 . . .***

Delete all of *file1 file2 . . . .* The option `-i` will tell `rm` to ask before it deletes each file. You can omit it, but keeping it is highly recommended to protect yourself against errors (such as typing "`rm x *`" rather than "`rm x*`". ***Be very careful with this command as you cannot recover files you've accidentally deleted!***

**df**

This displays the amount of free disk space on all file systems:

```
% df
Filesystem     kbytes    used   avail capacity  Mounted on
/dev/xy0a       14584   10014    3111    76%    /
/dev/xy0g      686209  304550  313038    49%    /usr
```

## Acknowledgement

This document was originally written by Kevin Rich, and has been modified for ECS 36A by Matt Bishop.