Chapter 7: The UNIX File System

The UNIX file system has a hierarchical or tree-like structure with the directory called *root* (/) as its source. The system is essentially composed of *files* and *directories*. In this chapter we describe techniques for manipulating files and directories, and commands designed to provide information about them.

7.1 Directory Structure

The UNIX system automatically puts you at a specific location in the file system when you log in. This is called your *login directory.* Typically, this is the same as your *home directory.* The name of your home directory is usually the same as your login name. Within this directory you can create files and additional directories (sometimes called *subdirectories*) in which to group the files. You can move and delete your own files and directories and control access to them.

The root of the file system is called *root* and is written as a slash (/). In other words, to change to the root directory, type:

```
% cd /
```

There is only one directory tree on a system even if several devices are mounted in that tree. (All devices are viewed as files.) The *current directory* or *working directory* is the directory that you are currently working in, which is also the directory that commands refer to by default. Files in your current directory can, therefore, be specified by their filenames only.

7.1.1 Pathnames

Wherever you can use a filename, you can also use a *pathname*, which is how you point to files that are not in your current directory. You can refer to files in other directories using either a *relative path name*, that is a path specified relative to your current directory, or with an *absolute path name*, that is a path specified relative to the root of the file system.

Absolute path names are preceded with `/`, the root directory. If a pathname does not begin with `/` it is assumed to be a relative path name. Relative path names begin with a directory or filename, a `.` (pronounced “dot”) which refers to the current directory, or `..` (pronounced “dot dot”) which refers to the directory immediately above the current directory. The character `/` also separates components of the pathname, which are directory names, except for the last one, which can be either a simple filename or a directory name.

In summary, every file has a pathname, and its absolute pathname is of the form:

```
/rootdir/dir2/... /filename
```

The following is the form of a relative pathname of a file:

```
dir_n/dir_n+1/... /filename
```

An example of an absolute path name is:

```
/usr/smith/project1/afile
```

If my current directory is `/usr/smith`, then I can refer to the file `afile` in subdirectory `project1` with a relative pathname like this:

```
project1/afile
```

Or, if my current directory is `/usr/smith/project1`, I can refer to a file named `fileb` in `/usr/smith/project2` as:

```
../project2/fileb
```

Note that you cannot necessarily tell if `fileb` is an ordinary file or a directory name. Many commands will accept a directory name, and if it *is* a directory name, the command in which it is used may perform the action on all files in the directory. This behavior can be dangerous!

7.1.2 The Home Directory

Your home directory is the top of your personal branch in the file system, and is usually designated by your username, i.e. `/<path>/<username>`.

Tilde (~)

In most UNIX shells other than **sh**, the *tilde* (**~**) stands for the home directory. Used alone, it specifies your home directory. Followed by a different user’s login name, it expands into the pathname of the home directory of that user. This is a convenient way to refer to a user’s directory, because it is independent of where the system manager may place the directory on the disk.

The use of tilde (**~**) to refer to a home directory is limited. It isn’t available in the Bourne shell, and isn’t available in FORTRAN.

Of the following three examples, the first refers to the file `def` from your own home directory, the second to the home directory of user *jones*, and the third refers to file `data1` in the subdirectory `project1` of jones' home directory.

```
~/def
~jones
~jones/project1/data1
```

To change to jones' home directory you'd enter:

```
% cd ~jones
```

logdir

FullFUE provides the command **`logdir`** which returns the full path of the specified user. **`logdir`** by itself returns the path of the invoking user. For example:

```
% logdir <username>
```

is equivalent to:

```
% echo <~username>
```

To change directories you'd enter (note the use of backquotes to use the output of the enclosed string):

```
% cd `logdir <username>`
```

In contrast to the tilde, **`logdir`** can be used within commands, scripts, FORTRAN and C programs, and other programs in all shells.

$HOME

The environment variable *HOME* is automatically set to the absolute pathname of your home directory. Environment variables are discussed in section 9.5 *Shell Variables and Environment Variables* and *HOME* is described in section 9.6 *Some Important Variables*. To see the value of *HOME*, enter:

```
% echo $HOME
```

From some other directory, you can change to your home directory or one of its subdirectories using a command like the following:

```
% cd $HOME[1]
```

or

```
% cd $HOME/mysubdir
```

---

1. **`cd`** by itself is equivalent to **`cd $HOME`**.

7.1.3 Command Line Directory Shortcuts

`.`	Current directory
`..`	Parent directory of the current directory ("up" one directory)
`~`	Your home directory (all shells but **sh**)
$HOME	Environment variable whose value is your home directory
`~<username>`	Home directory of another user (all shells but **sh**)
`/`	Root directory

7.1.4 Directories and Executables

It is appropriate at this point to mention the relationship between directories and commands. A command is simply the name of an executable file, located in some directory. To execute a command, the shell first needs to find the executable file. The shell therefore needs to be given a set of directories to search. This information is provided via the environment variable *PATH* which is a list of search directories. You can display it with the command:

```
% echo $PATH
```

PATH is explained more thoroughly in section 9.6 *Some Important Variables*. Standard UNIX commands are generally grouped in a few standard directories (e.g., `/usr/bin`), and your default *PATH* contains these. See section 9.6 *Some Important Variables* to learn how to run executables that you create and store in your own directories.

The utility **`which`** is useful in cases where a command may be ambiguous, for example due to aliasing (see section 9.7 *The Alias Command*), and you want to know exactly which executable file or files the command runs. **`which`** lists the files that would be executed if the specified command(s) had been run. The syntax for **`which`** is:

```
% which <command> [<command2> ...]
```

Each argument is expanded if it is aliased, and your path is searched for the executable files associated with the commands. See the man page for more information.

7.2 Files

An ordinary file contains ASCII characters or binary data and is considered by the UNIX system to be merely a sequence of bytes. No structure is imposed on the file and no meaning attached to its contents by the system; the meaning depends on the program that reads the file.

A directory file contains an entry for each file in that directory. The directory entry for a particular file contains the file name and *inode number*. The inode number is a volume data structure used by the file system. It has an associated entry in the *inode table* which contains other information about the file such as the owner, file protection, modification date.

A *hidden file* is an ordinary file whose name begins with a period (called "dot"). The reason they are called hidden is that the **`ls`** (list files) command does not list them by default. Use the **`-a`** option with **`ls`** to see them. Hidden files do not appear in filename expansion of *, either. Filename expansion is discussed in section 7.2.2, below.

UNIX does not support file versions. If you edit a file and save it with the same name, your earlier version is overwritten. Similarly, if you copy or rename (move) a file to a filename that already exists, the original file is overwritten.

7.2.1 Filenames

A full file specification has only two parts, the directory specification and the file name. A filename is composed of from 1 to 14 characters in old UNIX implementations and a much larger number in more recent versions (up to 255, typically). Although you can use any character in a filename except **`/`**, UNIX assigns special meaning to many characters (metacharacters), so they should be avoided (see section 9.2 *Special Characters (Metacharacters)*). It is safe to use the upper- and lowercase letters, numbers, dash (`-`), underscore (_), period (`.`), and comma (`,`). As mentioned in the previous section, files beginning with a dot (`.`) are hidden files. The "filenames" **`.`** and **`..`** (single and double dot) are reserved. The **`.`** refers to the current directory, and the **`..`** refers to the current directory's parent directory. No two files in the same directory can have the same name, but files in different directories can have the same name.

Dashes are fairly common in UNIX filenames simply because it's easier to type `my-file` than `my_file`.

Filenames are case sensitive. This means `MYFILE` is different from `Myfile` is different from `myfile` is different from `myFile`, etc.

You cannot distinguish a directory file from an ordinary file by its name, although some people make their own convention by beginning directory filenames with a capital letter, or ending them in `.d`.

Filename extensions are not required in UNIX. You can include a period and an extension in a filename to help describe the contents of the file, but it will not have special meaning to UNIX itself. However, programs can make use of extensions, for example the FORTRAN compiler expects certain extensions. Note, you can have more than one period in a filename, for example, `lex.yy.c`.

7.2.2 Filename Expansion and Wildcard Characters

The UNIX shells have a number of special characters which can be used on the command line when specifying filenames and directory names. They allow the shell to expand the argument into a set of filenames. These characters are called *wildcards*. Filename references that contain these characters are called *ambiguous file references*. Filename expansion is also called *globbing*.

The question mark (**?**) causes the shell to generate filenames which match any single character in that position. For example, `out?` matches `out1` but not `out12`.

The asterisk (*****) causes the shell to generate filenames which match any number of characters (including zero characters) in that position. For example, `myfile` matches `myf*`. The * alone means all files (except those that begin with dot (**.**), which is a special case).

A pair of brackets (**[ ]**) surrounding a list of characters causes the shell to match filenames containing the individual characters in that position. The brackets define a *character class* and each definition can only replace a single character in a filename. In other words, it is like a question mark that will only allow certain characters. For example, `memo1` and `memoa` match `memo[14a]`), but `memo3` and `memo1a` do not. A hyphen can be used to define a range of characters, for example `[a-z]` represents all lowercase characters. Thus `memo[a-z]` matches `memoa` but not `memo2` or `memoB`.

Character	Action
?	matches any single character in a filename
*****	matches any string of characters (including the empty string) in a filename
[ ]	matches any single character from the set enclosed in the brackets

Examples:

% ls out*	lists all files beginning with `out`
% ls out?	lists all files with 4-character names beginning with `out`
% ls out[ab]*	lists all files beginning with `out` followed by `a` or `b` (e.g., `outa4`)
% ls *out*	lists all files containing `out`

Filename expansion may surprise you with the results. For example, **`ls b*`** would list all files starting with b in the current directory, but it would also list the contents of all **directories** whose names start with `b` because of the way **`ls`** behaves for a directory argument. If you want to be sure of what filename expansion will result in, you can use the **`echo`** command to check it before executing a command.[1] For example, say you have a few matching files in your directory for the command:

```
% echo *out*
```

You would obtain output something like this:

```
fout fout275 inandout out1 out2 out
```

Filename expansion in **csh** can be turned off by setting the *noglob* variable:

```
% set noglob
```

To turn it back on, type **`unset noglob`**.

7.3 Manipulating Files

This section describes the basic file manipulation commands:

- listing the contents of a directory
- displaying the contents of a file
- copying and renaming a file
- deleting a file
- changing a file's access permissions

Section 7.5 *Manipulating Directories* describes the commands you can use to change and manipulate directories.

1. **`echo`** is otherwise useful for sending messages to the terminal from a script and sending known data into a pipe.

7.3.1 List Directory Contents: ls

The **ls** command, which stands for **list**, is used to list the contents of a directory. **ls** has many options, some of which are system-dependent, so only a few of them are described here. For a complete description of the command, refer to the man pages for **ls**.

ls by itself lists the names of the files and subdirectories in the current directory (in multicolumn format on some platforms), sorted alphabetically.

The format is:

```
% ls [<options>] [<filenames>]
```

where some of the options are:

-a	List all entries, including those that begin with **.** (dot).
-l	List in the long format, giving mode, number of links, owner, group, size in bytes, and (by default) time of last modification, by default sorted by filename.
-C	List in columns (default on some platforms)
-F	Put a / after the name of each file that is a directory, an * after the name of each file that is executable, and an @ after the name of each file that is a symbolic link.
-R	Recursively list subdirectories encountered.
-t	Sort by time stamp (latest first) instead of by name. The default time stamp is the last modification time (see **-u**).
-u	Use the time of last access for sorting if used with the **-t** option or printed in the date column if used with the **-l** option. **-ult** both sorts by and displays last access date.
-d	If the argument is a directory, list the directory itself, not its contents. Use with **-l** to get the status (e.g., permissions) of a directory.

If the argument is a directory, **ls** displays the contents of the directory. Note that this can happen unintentionally as a result of filename expansion. This behavior can be prevented with the **-d** option. The **-t** option is useful when looking at recent files:

```
% ls -lt
```

will result in the long output sorted by reverse modification date rather than by filename.

The following is a sample output of **`ls -l`**.

```
total 251
drwxr-xr-x 3 nicholls g020c 512    May 2   08:53
Tools
drwxr-xr-x 2 nicholls g020c 512    May 2   09:01
bin
-rw-r--r-- 1 nicholls g020c 446    May 4   14:09
defaults
-rw-r--r-- 1 nicholls g020c 95418  May 1   17:42
intro.lpr
-rw-r--r-- 1 nicholls g020c 0      May 10  17:51
lsout
-rw-r--r-- 1 nicholls g020c 6683   May 1   16:46
man.lpr
-rw-r--r-- 1 nicholls g020c 12258  May 9   16:16
out
```

The first line indicates the number of blocks used. The rest of the lines report on (sub)directories or files in the directory being reported on. The first column of the output is called the *mode*. The character in this first column indicates the type of file, and for our purposes here, they are:

`d`	directory
`-`	ordinary file

The next 9 characters are interpreted as three fields of three characters each, indicating the read (**`r`**), write (**`w`**), and execute (**`x`**) permissions for *owner* (sometimes called *user*), *group*, and *other*, in that order (see section 7.6.1 *File Access Permissions* for a discussion of permissions).

Next is the number of links to the file or directory. This refers to the number of different names established for it. Normally files have 1, and directories have 1 each for the directory itself, its parent directory, and each of its subdirectories. In the sample output above, notice that the directory `Tools` has 3 and `bin` has 2. Evidently, `Tools` has one subdirectory and `bin` has none.

The next fields are the login name of the owner, the group to which the owner belongs, the size of the file in bytes, the date and time the file was last modified, and, finally, the filename (which can be a directory name).

7.3.2 List File Contents: cat, less, more, head, and tail

UNIX has a number of commands that can be used for displaying the contents of a file at the terminal.

cat

cat, which stands for "con**cat**enate and print," is the standard UNIX file display; it simply prints the file to the screen.[1] When piped to **less** (see section 6.4.4 *Filters* which describes **less** as a filter), **cat** displays the file contents a screen at a time, and some simple commands may be executed at the supplied prompt.

```
% cat <filename>... [| less]
```

As its name suggests, **cat** is in fact quite useful for copying and concatenating files. Output can be piped to a file rather than to the screen, using standard output redirection (see section 6.4.2 *Standard Input and Output Redirection*). The following example concatenates the three specified files and copies them sequentially to a single file called `allthreefiles`:

```
% cat fileone filetwo filethree > allthreefiles
```

less, more

A shortcut for **cat <filename> |less** is to use **less** as a file browser:

```
% less <filename>
```

And wherever you use **less**, you can alternatively use **more**, although it is not as functional as **less**. You cannot move backwards through the file with **more**.

head, tail

head displays the first **<n>** lines of the specified file or files. If more than one file is specified, the filename is displayed before each set of file contents of **<n>** lines. **<n>** defaults to 10 lines.

```
% head [-<n>] [<filename>...]
```

tail displays the last lines of a file. Its syntax is a bit different:

```
% tail [+|-<n> lbc] [<filename>...]
```

The option **+<n>** displays the file starting **<n>** lines down from the beginning of the file, **-<n>** displays the last **<n>** lines. **l**, **b**, or **c** requests display of **<n>** lines, blocks, or characters (default is **l** lines). If more than one file is specified, the filename is displayed before each set of file contents. **<n>** defaults to 10 lines.

tail is useful when you want to see how far a process got. To display the last line of a log file, enter:

---

1. There are better ways to display a file (see **less** and **more**, which follow **cat** in this section).

```
% tail -1 <logfile>
```

7.3.3 Copy a File: cp

The command **cp** (stands for **co**p**y**) can be used to make a copy of a file, leaving the original version intact. You can copy a single file to another one (in the same or a different directory), or you can copy one or more files to a different directory, retaining the same filenames.

The syntax for these two situations varies slightly:

```
% cp [<options>] file1 targetfile
```

the file **<file1>** is copied to **<targetfile>**, where **<targetfile>** may include a path

```
% cp [<options>] <file1> [<file2> ...] <targetdirectory>
```

one or more files (<**file1> <file2>** ...) are copied to **<targetdirectory>**

If the target is a file, its contents are overwritten unless **-i** is specified, in which case you are prompted for confirmation.

Some options are:

-i	If the target filename exists, you are prompted for confirmation before overwriting.
-r	Used only with the **<targetdirectory>** form. Recursively copy a directory, its files, and its subdirectories to **<targetdirectory>**.

The first example below copies `myfile` to `anotherfile`, both in my current directory, prompting for verification in case `anotherfile` already exists.:

```
% cp -i myfile anotherfile
```

New users may find it useful to define **cpi** as the alias for **cp -i** to use in place of **cp** so that prompting always occurs. Section 9.7 *The Alias Command* discusses aliases.

The second example copies files `proj1` and `proj2` to another directory named `newproj` which is parallel to the current directory (has same parent directory as current):

```
% cp proj1 proj2 ../newproj
```

The third example copies the file `oldproj/proj1` to my current directory (**.**), which is a parallel directory to `oldproj` (has same parent directory). The file `proj1` keeps its name.

```
% cp ../oldproj/proj1 .
```

7.3.4 Move or Rename a File: mv

The **mv** command (stands for **m**o**v**e) allows you to rename a file in the same directory or move a file from one directory to another. If you move a file to a different directory, the file can be renamed or it can retain its original name. **mv** can also be used to move and rename directories.

```
% mv [<options>] <source1> [<source2> ...] <target>
```

Depending on whether the **<source(s)>** and **<target>** are files or directories, different actions are taken. These are described in the table below. If **<target>** is a filename, only one **<source>** file may be specified.

Source	Target	Result
file	new file name	Rename file to new name
file	existing file name	Overwrite existing file with source file contents; keep existing file name
directory	new directory name	Rename directory to new name
directory	existing directory name	Move directory such that it becomes a subdirectory of existing directory
one or more files	existing directory name	Move files to existing directory

An important option is:

-i If **<target>** exists, the user is prompted for confirmation before overwriting.

7.3.5 Reference a file: ln

The **ln** (link) command allows you to create a link in one directory to a file in the same or in a different directory, or to create a link to a different directory. Via links, a file or directory can appear to exist in multiple places, but only actually exist in one, thus conserving disk space. Links are often used to easily reference files or directories that would otherwise require a long path name.

The syntax for **ln** is similar to that for **cp** and **mv**, and in fact they are all run by the same executable.

The most commonly used options for the **ln** command are:

-i You are prompted before overwriting an existing filename.

`-s` This makes a *symbolic*, as opposed to an ordinary or *hard*, link. A symbolic link can point to a file that is in a different file system, whereas a hard link cannot.

A symbolic link displays the link and the file to which it is linked when you run **`ls -l`**; this is the only way to know the name that a file is linked to.

Note that when using the AFS file system, hard links can only be made between files that are in the same directory (the same *volume*, see section 8.5 *AFS Volumes and Quota*), so use the **`-s`** option even if you're in the same directory tree.

The syntax differs slightly for files and directories:

```
% ln [<options>] </path/to/file_name> </path/to/link_name>
```

Create the link **`<link_name>`** to reference the file file_name. If **`<link_name>`** already exists (as a link or as a file), it gets overwritten (unless you use option **`-i`**).

```
% ln -s [<other options>] </path/to/file_name>
  </path/to/link_name>
```

Create symbolic link named **`<link_name>`**, that links to **`<file_name>`** which exists in the same or another directory.

```
% ln -s [<other options>] </path/to/file_name1>
  [</path/to/file_name2> ...] <directory>
```

Create a symbolic link in **`<directory>`** to each of the listed files. The files may all exist in different directories since the **`-s`** option is used. The link names will be the same as the filenames they link to. If files of the same name but in different directories are specified, only the first file specified of that name will have a link created.

Let's look at an example:

```
% ln -s /e741/run1/e_mu2/mydata r5742
```

If `r5742` is a directory, this creates a link called `mydata` in the directory `r5742` that points to `/e741/run1/e_mu2/mydata`. You can now reference the data file as `mydata` (i.e., the same filename) as if it were in the directory `r5742`.

On the other hand, if `r5742` is not an existing directory then it represents the name of the link being created. In this case, the command establishes a link called `r5742` in the current working directory that points to the file `/e741/run1/e_mu2/mydata`. Running the command **`ls -l`** should display the following output:

```
lrw-r--r--   1 aheavey  g020          46 Aug 29 14:26 r5742
-> /e741/run1/e_mu2/mydata
```

7.3.6 Remove a File: rm

The **rm** command (stands for **r**e**m**ove) is used to remove the entries of one or more files.

```
% rm [<options>] <file>...
```

Some commonly used options are:

-i	Confirmation of removal of write-protected file occurs interactively, whether the standard input is a terminal or not. If used with the **-r** option, you are prompted about each directory before it is examined.
-r	Causes **rm** to delete the contents of the specified directory, including all its subdirectories, and the directory itself (**r**ecursive). This option should be used cautiously.

The file list can include ambiguous file references, so **rm** should be used cautiously. You can use the **echo** utility with the same ambiguous file reference to see the list generated.

Removal of a file requires write permission to its directory, but neither read nor write permission to the file itself. If the file has no write permission and the standard input is a terminal, the set of permissions is printed and you are prompted for confirmation. If the answer begins with a **y**, the file is deleted. If the standard input is not the terminal, the files are deleted without confirmation.

New users may find it useful to define **rmi** as the alias for **rm -i** to use in place of **rm** so that prompting always occurs. Section 9.7 *The Alias Command* discusses aliases.

7.3.7 Copy to/Restore from Archive or Tape: tar

The **tar** utility (**t**ape **ar**chive) can be used to create, add to, list and retrieve files from an archive file. Archive files are often stored on tape. The action taken by the **tar** command depends on the *key*, which is essentially a function option. The key must be specified on the command line as if it were the first option. It may be followed by function modifiers, and then by options and/or arguments. The keys and function modifiers must be grouped together before any arguments are listed. **tar** does not require, but does allow, a dash (**-**) before the list of keys and function modifiers. The keys and functions are:

c create a new tar file

r	append specified files to tar file
t	list all files in the tar file, or all files in a specified file list
u	append new or changed files to tar file
x	unwind entire tar file or extract specified files from tar file and write each file to the directory as specified in the tar file relative to the current directory

The keys, function modifiers and options are discussed in the man pages. Be aware that they vary in some cases between UNIX flavors. The command syntax varies somewhat from key to key, so check the man pages for that information, too.

Creating a Tar File

When creating a tar file, we have a few recommendations for avoiding problems:

First, when possible, create the tar file on a machine of the same flavor as the target flavor. Occasionally a tar file doesn't unwind properly on a different platform.

Secondly, choose your working directory carefully. It is often convenient or desirable to be able to specify simple relative path names for the files to include in the tar file. For example:

```
% cd /path/to/dir
% tar cvf /tmp/filename.tar .
```

creates a tar file with all pathnames relative to `/path/to/dir`. In general you should not specify the pathname explicitly on the command line, unless it will be valid on any other system where the tar file may be unwound and used.

Thirdly, be careful choosing your target directory for the new tar file. Make sure that the target directory is outside of the directory tree that you're including in the tar file. Otherwise the tar file tries to include itself, and can grow infinitely large.

Unwinding a Tar File

To unwind a tar file, first **`cd`** to or create the target directory in which you want the tar file unwound, then unwind the product tar file:

```
% cd <target_directory>
% tar xvf <path_to_unwound_tar_file>
```

7.3.8 Compress or Expand a File: gzip, gunzip

Several utilities are available on UNIX systems for file compression. **compress** and **pack** are native UNIX utilities, and **gzip** is provided by FullFUE. We recommend you use **gzip** for file compression, and its associated utility **gunzip** for file expansion. **gunzip** recognizes and can expand files compressed with **compress** and **pack** as well as **gzip**.[1]

The file extensions **gunzip** recognizes include `.gz`, `-gz`, `.z`, `-z`, `_z`, and `.Z`. **gunzip** also recognizes the special extensions `.tgz` and `.taz` as shorthands for `.tar.gz` and `.tar.Z`, respectively. When compressing, **gzip** uses the `.tgz` extension if necessary instead of truncating a file with a `.tar` extension.

You will need to reference the man pages for details on syntax, options and usage. In their simplest forms **gzip** and **gunzip** can be used as follows, starting, for example with the original uncompressed file `bigfile`:

```
% gzip bigfile
```

The result is `bigfile.gz`, whose size is reduced with respect to `bigfile` according to Lempel-Ziv coding (LZ77), the same compression scheme used by **compress**. Whenever possible, **gzip** replaces each file by one with the extension `.gz`, while keeping the same ownership modes and access and modification times. **gzip** will only attempt to compress regular files. In particular, it will ignore symbolic links. If the compressed file name is too long for its file system, **gzip** truncates it.

Compressed files can be restored to their original form using **gunzip**, or equivalently by using the **`-d`** option with **gzip**. If the original name saved in the compressed file is not suitable for its file system, a new name is constructed from the original one to make it “legal”.

To restore `bigfile.gz` to its original name and size, enter:

```
% gunzip bigfile.gz
```

7.4 Information About Files

This section gives a cursory overview of simple uses for two very powerful commands for dealing with files: **`find`** for searching for files and **`grep`** for searching for strings within files. We also describe **`wc`** which displays the size of a file, **`od`** which creates a dump of a file, and **`file`** which can determine file type.

1. Under FullFUE on most systems (namely where **gzip** and **gunzip** are not installed in `/usr/local/bin`) you will need to run **`setup gtools`** in order to access them.

7.4.1 Find a File: find

The **find** utility tests each file in the given pathname list to see if it meets the criteria specified by the expression supplied. It does this by recursively descending the directory hierarchy for each path name. The format is:

```
% find <path-name-list> <expression>
```

<path-name-list> can contain file expansion metacharacters. Each element in **<expression>** is a separate boolean criterion. A space separating two criteria is a logical AND operator, a **-o** separating the criteria is a logical OR operator. A criterion can be negated by preceding it with an exclamation point (**!**). Criteria are evaluated from left to right unless parentheses are used to override this. Special characters must be quoted (use \) and there must be spaces on each side of the special character pair.

Some of the criteria that can be used within **<expression>** are:

-name <filename>	True if **<filename>** matches the name of the file being evaluated. Ambiguous file references can be used if enclosed in quotes.
-type <filetype>	True if the type of the file is **<filetype>**, where **<filetype>** is either d (directory) and f (ordinary file).
-atime <n>	True if the file has been accessed in **<n>** days.
-mtime <n>	True if the file has been modified in **<n>** days.
-newer <filename>	True if the file has been modified more recently than **<filename>** has.
-print	Causes the matching path names to be displayed on the screen.
-exec <command> \;	True if **<command>** returns a zero exit status. **<command>** must be terminated with a quoted semicolon (note the \). An empty pair of braces ({}) within the command represents the filename of the file being evaluated.
-ok <command> \;	Same as **-exec** except the generated command line is displayed and executed only if the user responds by typing y.

In the previous list, **+<n>** means more than **<n>** , **-<n>** means less than **<n>**, **<n>** means exactly **<n>** .

Note that **find** doesn't do anything with the found files, it doesn't even display the names, unless instructed to.

Examples:

- Search the current directory and all subdirectories for the file `lostfile`:

```
% find . -name lostfile -print
```

- List all files ending in `.html` in your `/wwwork` subdirectory:

```
% find wwwork -name '*.html' -print
```

- This command will prompt you if you want to execute **`more`** on each file that begins with the letter `d` in the current directory and all subdirectories (Enter `y` if you want the file displayed.):

```
% find . -name 'd*' -ok more {} \;
```

- List all files in the current directory that *don't* begin with `m`:

```
% find . ! -name 'm*' -print
```

- Find all files in the current directory and all subdirectories that contain the string *hello*:

```
% find . -exec grep -l "hello" {} \;
```

- Remove all files in your directory tree that are named `a.t` or have the extension of `.o` and haven't been accessed in a week:

```
% find ~\(-name a.t -o -name '*.o') -atime +7
              -exec rm {} \;
```

Note that using the `find` command takes up a lot of system resources.

In particular on AFS systems, you may accidentally end up searching servers all over the world if the top of the search is at the root directory (**`/`**). Generally you should be careful to only search the part of the UNIX tree that interests you. Here is an example:

Look for **`<filename>`** starting at the root directory (**`/`**), and exclude searches in the `/afs` and `/nfs` branches:

```
% find / \( -name /afs -prune \) -o \( -name /nfs -prune \) -o
  -name <filename> -print
```

7.4.2 Search for a Pattern: grep

The **`grep`** utility searches the contents of one or more files for a pattern.

The format is:

```
% grep [<options>] <pattern> [<file> ...]
```

Some of the options are:

`-c`	Display only a count of lines that contain the pattern.
`-i`	Ignore upper/lower case distinctions during comparisons.

-l Display only the name of each file that contains one or more matches.

The pattern can be a simple string or a regular expression (see section 6.4.5 *Regular Expressions*). You must quote regular expressions that contain special characters, spaces, or tabs (this can be done by enclosing the entire expression within single quotation marks).

Examples:

- Find all non-hidden files in the current directory containing the string *smith*:

```
% grep -i smith *
```

- Search the file `abc` for a string beginning with *f*, followed by 0 or more *r*'s, and ending in *og* (e.g., *frog, fog, frrog*):

```
% grep 'fr*og' abc
```

- Search the file `myfile` for a line beginning with a **T**:

```
% grep '^T' myfile
```

or

```
% less myfile | grep '^T'
```

- Search `/usr/jones/junk` for the characters *file* followed by a number (e.g., *file1, file3*):

```
% grep 'file[0-9]' /usr/jones/junk
```

- Display a line if Smith is logged in:

```
% who | grep smith
```

- Show all processes being run by Smith:

```
% ps -ef | grep smith
```

- Show all environmental variables containing **<string>** in their name or their translation:

```
% env | grep <string>
```

- Show all aliases containing **<string>** in their name or their translation:

```
% alias | grep <string>
```

7.4.3 Count a File: wc

The **wc** command, which stands for **w**ord **c**ount, counts the number of lines, words, and characters there are in the named files, or in the standard input if the argument is absent. If there is more than one file, **wc** totals the count as well.

```
% wc [-lwc] [<names>]
```

The options **l**, **w**, and **c** may be used in any combination to specify that a subset of **l**ines, **w**ords, and **c**haracters be reported. The default is **-lwc.**

UNIX users frequently count things by piping them into **wc**. For example, to display the number of users logged into the system, you can execute:

```
% who | wc -l
```

7.4.4 Dump a File: od

The **od** (**o**ctal **d**ump) command can be used to examine the contents of a file in various formats: octal, decimal, hexadecimal, and ASCII. The default is octal.

The format is:

```
% od [<options>] [<file>] [<offset>] [|less]
```

If **<file>** is not included, standard input is assumed. The options are:

-c	Produces a character dump.
-d	Produces a decimal dump.
-o	Produces an octal dump.
-x	Produces a hexadecimal dump.

The **-c** option prints non-printable characters as a printable character preceded by a backslash: **\0** is null, **\b** is backspace, **\f** is form-feed, **\n** is new-line, **\r** is return, and **\t** is tab.

The **<offset>** specifies where in the file the dump is to begin, if different than the beginning of the file. It is of the form **[+]<n>[.][b]**. The **+** is only necessary if you have no file specified so that the command interpreter knows this is the offset not the file. Without **.** or **b**, **<n>** indicates the dump starts at (octal) byte **<n>** of the file. A **.** displays **<n>** in decimal, a **b** in 512-byte blocks.

We recommend that you always pipe the output of **od** to **less** (or **more**) so that you can manipulate it. Large files can be unwieldy, and you may not be able to stop the output once it's going!

7.4.5 Determine File Type: file

The **file** utility can be used to determine the file type of a file according to its contents. It bases its guesses on a list of "magic numbers" recorded in a "magic file", `/etc/magic`. Some of the file types are:

- ASCII text
- C program text
- directory

• executable

file determines the filetype by looking at the beginning of the file and comparing it to entries in the magic file. The command format is:

```
% file <filename>...
```

7.5 Manipulating Directories

This section describes the commands you can use to organize and use the UNIX directory structure. It describes how to make and remove directories, and move from one directory to another. Listing the contents of a directory (files and subdirectories) was described in a previous section. Section 7.6.2 *Directory Permissions* explains the meaning of access permissions as applied to directories.

7.5.1 Print Working Directory: pwd

The **pwd** command (for **p**rint **w**orking **d**irectory) displays the path name of your working (current) directory. The command format is:

```
% pwd
```

7.5.2 List Directory Contents: ls

The **ls** command, which stands for **list**, is used to list the contents of a directory. **ls** has many options, some of which are system-dependent. A few of them are described in section 7.3.1 *List Directory Contents: ls*. For a complete description of the command, refer to the man pages for **ls**.

7.5.3 Change Directory: cd

When you first log in to the system, you are placed in your home directory, which is then also your current working directory. You can use the **cd** command (for **c**hange **d**irectory) to change your current working directory. The command format is:

```
% cd [<directory>]
```

You can specify a complete path or a relative path. You can use **..** (for the parent directory) in your pathname. You must have execute permission (which provides search permission in this case) on a directory to **cd** to it.

If **<directory>** is not specified, you are returned to your home directory.

The following examples illustrate moving to different directories:

- your home directory

  % cd

- a subdirectory called `Tools`

  % cd Tools

- a colleague's home directory (using absolute pathname)

  % cd /usr/jones

- a colleague's subdirectory (using tilde)

  % cd ~jones/ourfiles

- a parallel directory (has same parent directory as current directory)

  % cd ../Tools

7.5.4 Make a Directory: mkdir

The **mkdir** command (for **m**a**k**e **dir**ectory) is used to create a directory. The command format is:

```
% mkdir <dirname> ...
```

If a pathname is not specified, the directory is created as a subdirectory of the current working directory. Directory creation requires write access to the parent directory. The owner ID and group ID of the new directory are set to those of the creating process.

Examples:

- create a subdirectory in the current working directory

  % mkdir progs

- create one with a full pathname (the directory `Tools` must already exist)

  % mkdir /usr/nicholls/Tools/Less

7.5.5 Copy a Directory

The most straightforward way of copying a directory and its contents is to pipe the output of the **ls** command (see section 7.3.1 *List Directory Contents: ls*) into the file copy facility **cpio** (see the man pages). However this technique does not copy subdirectories.

First, create the destination directory using **mkdir** (see section 7.5.4 *Make a Directory: mkdir*), if it doesn't already exist. Secondly, from the source directory, run the command (shown with recommended options; see man pages for option information):

```
% ls | cpio -dumpV <destination_dir>
```

The **<destination_dir>** must be specified relative to the source directory.

If you need to copy a directory structure, then use the **tar** utility instead. It is described in section 7.3.7 *Copy to/Restore from Archive or Tape: tar*. The following sequence of commands (shown on a single line) copies a structure from the source directory to the destination directory. The **<destination_dir>** is taken as relative to the **<source_dir>**:

```
% cd <source_dir>; tar cf - . | (cd <destination_dir>; tar xfBp
  -)
```

The "**-**" is used for the name of the tar file (argument to the **f** option) so that **tar** writes to the standard output or reads from the standard input, as appropriate.

7.5.6 Move (Rename) a Directory: mv or mvdir

See section 7.3.4 *Move or Rename a File: mv* for information on **mv**. To move a directory (**<olddirname>**) and its contents to a different position in the directory tree, use the command format:

```
% mvdir <olddirname> <newdirpath>
```

If **<newdirpath>** exists already, then the directory gets moved to **<newdirpath>/<olddirname>**. Note that the two arguments cannot be in the same path. For example:

```
% mvdir x/y x/z
```

is ok, but

```
% mvdir x/y x/y/z
```

is not ok.

7.5.7 Remove a Directory: rmdir

You can remove a directory with the **rmdir** command. The directory must contain no files or subdirectories, and you must have write permission to the parent directory.

```
% rmdir <dirname> ...
```

You can use an absolute or relative pathname.

You can also use **rm -r** as described in Section 7.3.6 *Remove a File: rm*. **rm -r** will delete a directory, all subdirectories, and all files. **This command should be used with extreme caution.**

For example, the following command deletes the directory `temp`, all subdirectories of `temp` and all files contained in those directories, prompting before each removal, and confirming removal of write-protected files (**`-i`**):

```
% rm -ir /usr/jones/temp
```

7.6 File and Directory Permissions

7.6.1 File Access Permissions

The UNIX file system allows you to control read, write, and execute access to your files on the basis of user (owner), group, and other (everyone else).[1] In this section we will consider only the standard UNIX file permissions.

Note that in the AFS file system, file permissions are mediated by Access Control Lists (ACLs) that are set on a directory level. The standard UNIX file permissions don't apply in this case except for the owner permissions, which apply to all users. AFS file permissions are treated in section 8.6 *File and Directory Permissions***.**

To determine the current permissions, use the long form of the **`ls`** command, **`ls -l`**. Referring to the example below, the nine characters immediately following the first field represent the one-bit flags known as the *mode bits* that control file access. A dash indicates a bit is not set, **`r`** stands for **`r`**ead

1. Note **o** is for **other** and not for **owner** as on VMS.

access, **w** for **w**rite access, and **x** for e**x**ecution access. The first set of three characters refer to owner permission, the middle three for group permission, and the last three for all other user classes.

```
     total 251
     drwxr-xr-x 3 nicholls g020c 512    May 2   08:53
     Tools
     drwxr-xr-x 2 nicholls g020c 512    May 2   09:01
     bin
     -rw-r--r-- 1 nicholls g020c 446    May 4   14:09
     defaults
     -rw-r--r-- 1 nicholls g020c 95418  May 1   17:42
     intro.lpr
     -rw-r--r-- 1 nicholls g020c 0      May 10 17:51
     lsout
     -rw-r--r-- 1 nicholls g020c 6683   May 1   16:46
     man.lpr
-rw-r--r-- 1 nicholls g020c 12258 May 9  16:16 out
```

In the example, ignoring the directory files (which have a **d** in position 1), the owner has **rw** access to the files, whereas group and others have read (**r**) access only.

chmod

The **chmod** command, which stands for **ch**ange **mod**e, is used to change access permissions of a file or directory:

```
% chmod <mode> <filename> ...
```

or

```
% chmod <mode> <directory> ...
```

In the *absolute form* of the mode where the level of protection is specified in octal format, **<mode>** looks like 741 or 554, for example, where each of the three octal numbers represents the sum of the permissions granted to its class: user, group, and other, in that order. The three types of permission have the values:

read	4 (100 octal)
write	2 (010 octal)
execute	1 (001 octal)

For example, a mode of 741 means owner can read, write, and execute (4+2+1=7); group can read (4+0+0=4); and others can execute the file (0+0+1=1).

To give this permission to a file `test`, you would enter:

```
% chmod 741 test
```

You can use an alternate form of **<mode>** in the **chmod** command in which **<mode>** is a three-character field specifying an action to be taken. The action is to add or subtract one or more permissions from one or more user classes. It takes the form:

```
<who> <operator> <permission(s)>
```

These three positions within the field take the following characters:

<who>	represents the user class or classes; it takes any combination of **u**, **g**, **o**, and **a** for **u**ser (user is really the owner), **g**roup, **o**ther and **a**ll, respectively, where *all* includes the three individual classes
<operator>	**+** or **-** for adding or subtracting permissions, or = for setting a specific permission and resetting all other permissions for the specified user class(es)
<permission(s)>	any combination of **r**, **w**, and **x** for **r**ead, **w**rite, and e**x**ecute, indicating the permissions to be permitted, denied, or reset.

Examples of the **chmod** command:

- Remove group execute permission to the file `progs`:

  ```
  % chmod g-x progs
  ```

- For the files `out` and `out1`, add group read and write, and deny write to other:

  ```
  % chmod g+rw,o-w out out1
  ```

- Set group read permission and reset all other group permissions to `myfile`:

  ```
  % chmod g=r myfile
  ```

Note that classes of users or levels of protection not specified in a command are not modified in this form of the command (with the exception that **=** resets other permissions).

umask

With the **umask** command you can specify a mask that the system uses to set access permissions when a file is created. In order to understand **umask** you need to know that access permission at file creation is application-dependent.

Each command or application sets a file permission in its *open* command.[1] The system then "subtracts" any user-defined mask, resulting in the final access permission for the file. You can set a umask by this command:

```
% umask [<ooo>]
```

where **<ooo>** stands for three octal digits. The user-specified "mask", **<ooo>**, has the same positional structure as described above for **chmod**, but specifies permissions that should be **removed** (disallowed).

For example, a mask of 022 removes no permissions from owner, and removes write permission from group and others. Thus a file normally created with 777 would become 755 (this would appear as `rwxr-xr-x` in the format put out by the command **ls -l**). The following command could be put in your `.cshrc` or `.profile`.

```
% umask 022
```

The meaning of permissions applied to directories is described in Section 7.6.2 *Directory Permissions*.

7.6.2 Directory Permissions

See section 8.6.2 *Directory Permissions via Access Control Lists (ACLs)* for AFS systems.

You can grant or deny permission for directories as well as files, and protection assigned to a directory file takes precedence over the permissions of individual files in the directory.

- Read permission for a directory allows you to read the names of the files contained in that directory with the **ls** command, but not to use them.
- Write permission for a directory allows you to create files in that directory *or* to delete any file in the directory, regardless of the file protection on the files themselves. It does not allow you to see the files or use them without **r** and **x** directory permission. In other words, write permission to a directory allows you to alter the contents of the directory itself, but not to alter, except to remove, files *in* the directory (which is controlled by the file's permissions).
- Execute permission allows you to list the contents of the directory.

File access permissions of directory files are changed with the **chmod** command (see section 7.6.1 *File Access Permissions*).

1. Normally only the loader creates files with execute permission.

7.7 Temporary Directories

By convention, there are directories named `/tmp` (and sometimes `/usr/tmp`) where programs and users can store temporary files. Many programs (e.g., compilers) write temporary files there or in the area specified by the environment variable *TMPDIR*. Since these are public areas, it is necessary to manage this space, which means that you cannot count on files being retained in these directories.

Many systems on site have fairly small `/tmp` areas and therefore you must be careful not to fill up this space. In general, you should only use `/tmp` for very temporary, small files. On many systems files in `/tmp` will disappear after a reboot or after existing for a week. You can set *TMPDIR* to a different location if there is not enough space in these areas.

Contact the administrators of the particular system to find out what the current policy is on the machine.