CHAPTER 6

The File System

UNIX looks at everything as a file. Any UNIX file system will have tens of thousands of files. If you write a program, you add one more file to the system. When you compile and run it, you add some more. Files grow rapidly, and if they are not organized properly, you'll find it virtually impossible to access them. Proper file organization requires an elaborate directory-based storage system so a user can "place" oneself in a directory (folder), or transfer files from one directory to another.

The file system in UNIX is one of its simple and conceptually clean features. The system has proved so useful that it has been widely adopted by other systems including Windows. In this chapter, you'll handle directories and navigate freely in the file system. You'll copy and remove files in these directories and learn to display and print files. You'll also use the commands that UNIX provides to use the disk space efficiently. This chapter handles files and directories, but not their attributes (properties). File attributes along with their commands are taken up in the next chapter.

Objectives

- Learn how UNIX divides files into three categories. *(6.1)*
- Understand the considerations involved in framing a filename. *(6.2)*
- Know the parent-child relationship between files. *(6.3)*
- Learn the important directories of the UNIX file system. *(6.4)*
- Understand the difference between *absolute* and *relative pathnames*. *(6.6 and 6.8)*
- Check and change your directory with **pwd** and **cd**. *(6.5 and 6.7)*
- Create and remove directories with **mkdir** and **rmdir**. *(6.9 and 6.10)*
- Copy, remove and rename files with **cp**, **rm** and **mv**. *(6.11, 6.12 and 6.13)*
- Display and create a file with **cat**. *(6.14)*
- Identify the type of file with **file**. *(6.15)*
- Print a file with **lp** (or **lpr**). *(6.16)*
- Find out disk space usage and free space with **df** and **du**. *(6.17 and 6.18)*
- Compress a file with **compress**, **gzip** and **zip**. *(6.19)*

6.1 The File

A UNIX file is a storehouse of information for the most part, it's simply a sequence of characters. UNIX places no restriction on the structure of a file. A file contains exactly those bytes that you put into it—be it a source program, executable code or anything

else. It neither contains its own size nor its attributes (properties), including the end-of-file mark. It doesn't even contain its own name!

The founders' vision of a file is all encompassing; it includes directories and devices in its definition. Whether it's a C program you execute, or a directory you use to house groups of such programs, they are all files in the UNIX sense. This concept also extends to each and every device you find in your machine—the hard disk, printer, tape, CD-ROM drive or terminal. The shell is also a file, and so is the kernel. And if you are wondering how UNIX treats the main memory in your system, it's a file too!

The most notable feature of the UNIX file system is that UNIX makes *little* distinction between these various types of files. Many commands (which are also files themselves) work with all types of files, and you don't need to indicate specifically the type of file you are using. Many of the commands used to access a disk file are also used to access the tape drive. With limited exceptions, UNIX doesn't offer separate commands to access CD-ROMs and tapes; the same command often accesses both.

Although everything is treated as a file by UNIX, it's still necessary to divide a file into three categories:

- Ordinary file—Also known as regular file. It contains only data as a stream of characters.
- Directory file—A folder containing the names of other files and directories.
- Device file—It represents all hardware devices.

The reason why we make this distinction is that the significance of a file's attributes depends on its type. Read permission for an ordinary file means something quite different from that for a directory or a device. Moreover, you can't directly put something into a directory file, and a device file isn't really a stream of characters. While the vast majority of commands work with all types of files, some don't. For a proper understanding of the file system, you must understand the significance of these files.


Note

A UNIX file doesn't contain any of its attributes nor does it contain the end-of-file mark. The name of the file is not stored in the file either.

6.1.1 Ordinary File

The traditional file is of the **ordinary** or **regular** type. It consists of a stream of data resident on some permanent magnetic media. You can put anything you want into this type of file. This includes all data, source programs, object and executable code, all UNIX commands, as well as any files created by the user. Commands like **`cat`**, **`ls`**, and so forth are treated as ordinary or regular files.

The most common type of ordinary file is the **text file**. This is just a regular file containing printable characters. The programs that you write are text files. The UNIX commands that you use, or the C programs that you execute, are not. The characteristic feature of text files is that the data inside them are divided into groups of lines, with each line terminated by the *linefeed* (LF) character (ASCII decimal value 10). On UNIX systems, the linefeed character is also known as the newline character *(4.10.2)*, and when we talk of newline, we actually mean LF. This character isn't visible and doesn't appear in hard copy output. It's generated by the system when you press the *[Enter]* key.

6.1.2 Directory File

A **directory file** contains no external data but maintains some details of the files and subdirectories that it contains. The UNIX file system is organized with a number of such directories and subdirectories, and you can also create them as and when you need. You often need to group a set of files pertaining to a specific application and have them under a directory. This allows two or more files in separate directories to have the same filename.

A directory file contains two fields for each file—its name and identification number. (Every file has a number called the **inode number**.) If a directory houses, say, 10 files, there will be 10 such entries in the directory file. You can't, however, write directly into a directory file; that power rests only with the kernel. When an ordinary file is created or removed, its corresponding directory file is automatically updated by the kernel with the relevant information about the file.

You don't need to concern yourself with further details at this stage. The directory file is further discussed in the next chapter.

Note

It is the directory file that contains the names of all files resident in the directory.

6.1.3 Device File

The definition of a file has been broadened by UNIX to consider even physical devices as files. This definition includes printers, tapes, floppy drives, CD-ROMs, hard disks and terminals. Although this may appear confusing initially, it's really an advantage; some of the commands used to access a disk file also work with **device files**.

The device file is special; it doesn't contain any data whatsoever. (This file is not a "sequence of characters" in contrast to what we had observed earlier.) Any output directed to it will be reflected onto the respective physical device associated with the filename. When you issue a command to print a file, you are really directing the file's output to the file associated with the printer. When you back up files onto tape, you are "symbolically" using the file associated with the tape drive. The kernel takes care of this by mapping these special files to their respective devices.

Now that you understand the three types of files, you shouldn't feel baffled by subsequent use of the word in the book. The term *file* will often be used in this book to refer to any of these types, though it will mostly be used to mean an ordinary file. The real meaning of the term should be evident from its context.

6.2 What's in a (File)name?

On most UNIX systems today, a filename can consist of up to 255 characters. Though this figure is normally never reached, if you enter more than 255 characters when specifying a filename, only the first 255 characters are effectively interpreted by the system. Some systems, however, report an error message.

Files may or may not have extensions and can consist of practically any ASCII character except the /. Just as you can have a filename beginning with a dot, you can have one which ends with a dot too. All these are valid filenames:

```
index      .last_time      LIST.
```

If you want, you can also use control characters or other unprintable characters in a filename. These filenames can be frightening, but they are still valid:

```
^V^B^D-++bcd      -{}[]      @#$%□*abcd
```

*Space before **

The first filename contains three control characters (*[Ctrl-v]* being the first). The second begins with a hyphen (a filename that can cause a lot of trouble). The third one apparently shows an embedded space in the name. Later, you'll learn to find out whether it actually contains a space or an unprintable character.

Many of these characters have special significance when used in the command line. To be specific, the shell metes out different treatment for characters like $, `, ?, * and &, and commands behave unpredictably if filenames have these characters. You should also avoid using control characters. In fact, you should use only the following characters when framing filenames:

- Alphabets and numerals
- The period (.)
- The hyphen (-)
- The underscore (_)

UNIX imposes no restrictions on the extension, if any, that a file should have. A shell script doesn't need to have the `.sh` extension, even though it helps in identification. In all cases, it's the application that imposes this restriction. For instance, the C compiler expects C program files to have `.c`; Java expects its source files to have `.java`. Windows users must also keep these two points in mind:

- A file can have many dots embedded in its name; `a.b.c.d.e` is a perfectly valid filename. A filename can also begin with a dot or end with one.
- UNIX is sensitive to case; `chap01`, `Chap01` and `CHAP01` are three different filenames, and it's possible for them to coexist in the same directory.

Tip

Avoid using any characters other than alphabets, numerals, the dot, hyphen and underscore character in framing a filename. Use uppercase sparingly.

Caution

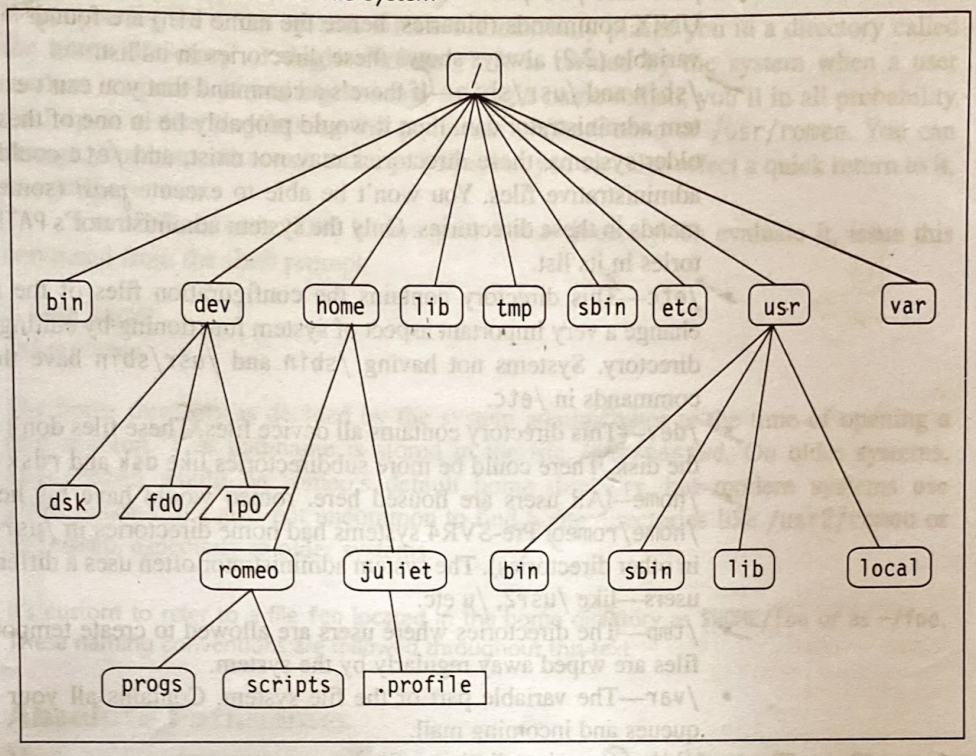
Never use a - at the beginning of a filename. You'll have a tough time getting rid of it! Moreover, many commands using this filename as an argument will instead treat it as an option and report errors. For instance, if you have a file named `-z`, **`cat -z`** won't display the file but interpret it as an invalid option.

6.3 The Parent–Child Relationship

All files in UNIX are "related" to one another. The file system in UNIX is a collection of all these related files (ordinary, directory and device files) organized in a hierarchical (an inverted tree) structure. This system has also been adopted by Windows, and is visually represented in Fig. 6.1.

The notable feature in every UNIX file system is that there is a supremo, which serves as the reference point for all files. This top is called **root**, and is represented by a / (frontslash). root is actually a directory file, and it has all the subdirectories of the system under it. These subdirectories, in turn, have more subdirectories and other files

FIGURE 6.1 *The UNIX File System*



under them. For instance, `bin` and `usr` are two directories directly under root, while a second `bin` and `lib` are subdirectories under `usr`.

Every file, apart from root, must have a parent, and it should be possible to trace the ultimate parentage of a file to root. If the ancestry of a file can't be traced to root, the file is simply not part of the file system. This should be easy for you to understand as we all have our own families with similar *grandparent–parent–child* relationships. Thus, the `home` directory is the parent of `romeo`, while root is the parent of `home` and the grandparent of `romeo`. If you create a file `login.sql` under the `romeo` directory, then `romeo` will be the parent of this file.

It's also obvious that in these parent–child relationships the parent is always a directory. `home` and `romeo` are both directories as they are both parents of at least one file or directory. `login.sql` is an ordinary file and can't have a directory under it.

6.4 The UNIX File System

Now let's take a cursory look at the structure of the UNIX file system. This structure has been changing constantly over the years until AT&T proposed one in its SVR4 release. Though vendor implementations vary in detail, broadly the SVR4 structure has been adopted by most vendors.

Refer to Fig. 6.1 which shows a heavily trimmed structure of a standard UNIX file system. In real life, the root directory has many more subdirectories under it than shown, but for our initial comprehension, we'll stick to these:

- /bin and /usr/bin—These are the directories where all the commonly used UNIX commands (binaries, hence the name bin) are found. Note that the PATH variable *(2.2)* always shows these directories in its list.
- /sbin and /usr/sbin—If there's a command that you can't execute but the system administrator can, then it would probably be in one of these directories. On older systems, these directories may not exist, and /etc could then contain all administrative files. You won't be able to execute *most* (some, you can) commands in these directories. Only the system administrator's PATH has these directories in its list.
- /etc—This directory contains the configuration files of the system. You can change a very important aspect of system functioning by editing a text file in this directory. Systems not having /sbin and /usr/sbin have the administrative commands in /etc.
- /dev—This directory contains all device files. These files don't occupy space on the disk. There could be more subdirectories like dsk and rdsk in this directory.
- /home—All users are housed here. romeo would have his home directory in /home/romeo. Pre-SVR4 systems had home directories in /usr (and sometimes in other directories). The system administrator often uses a different directory for users—like /usr2, /u etc.
- /tmp—The directories where users are allowed to create temporary files. These files are wiped away regularly by the system.
- /var—The variable part of the file system. Contains all your print jobs, mail queues and incoming mail.
- /lib—Contains all library files.

There are a lot more directories, and unfortunately UNIX flavors differ in name and location of many of the system directories. Truly speaking, you really don't need to know more than this right now as file system internals are of interest mainly to the system administrator and a separate chapter has been earmarked for them.

6.5 pwd: Knowing Your Current Directory

It is a remarkable feature of UNIX that, like a file, a user is also placed in a specific directory of the file system on logging in. You can move around from one directory to another, but at any point of time, you are located in only one directory. This directory is known as your **current directory**.

You would often need to know what your current directory is. The **pwd** (present working directory) command tells you that:

```
$ pwd
/home/romeo
```

This is a pathname

What you see above is a **pathname**—a sequence of directory names separated by slashes. This pathname shows your location with reference to the top—which is root. **pwd** here tells you that you are placed in the directory romeo, which has directory home as its parent, which in turn has root (the first /) as its parent. These slashes act as delimiters to file and directory names, except that the first slash is a synonym for root.

6.5.1 The Home Directory

When you log on to the system, UNIX automatically places you in a directory called the **home directory** (or **login directory**). It is created by the system when a user account is opened. If you log in using the login name romeo, you'll in all probability land up in a directory having the pathname `/home/romeo` or `/usr/romeo`. You can change your home directory when you like, but you can also effect a quick return to it, as you'll see soon.

The shell variable `HOME` knows your home directory. To evaluate it, issue this command from the shell prompt:

```
$ echo $HOME
/home/romeo
```

The home directory is decided by the system administrator at the time of opening a user account. This pathname is stored in the file `/etc/passwd`. On older systems, `/usr/romeo` would be romeo's default home directory, but modern systems use `/home/romeo`. It's also not uncommon to find home directories like `/usr2/romeo` or `/u/romeo`, especially in older systems.


Note

It's custom to refer to a file `foo` located in the home directory as `$HOME/foo` or as `~/foo`. These naming conventions are followed throughout this text.

6.6 Absolute Pathnames

Many UNIX commands use file and directory names as arguments. These files and directories are presumed to exist in the current directory. For instance, the command

```
cat login.sql
```

will work only if the file `login.sql` exists in your current directory. However, if you are placed in `/var` and you want to access `login.sql` in `/home/romeo`, you can't obviously use the above command. There are two ways of accessing this file from your current directory `/var`:

- With an **absolute pathname** which uses the root directory as the ultimate reference for the file. All path references here originate from root.
- With a **relative pathname** which uses the current directory as point of reference and specifies the path relative to it.

We'll consider relative pathnames shortly. Using an absolute pathname for `login.sql` means that you have to use this command line from `/var` to display the file:

```
cat /home/romeo/login.sql                    Can be used from any directory
```

`cat` here uses an absolute pathname, where the location of `login.sql` is specified with reference to root (the first /). When you have more than one / in a pathname, for each such /, you have to descend one level in the file system. Thus, `romeo` is one level below `home`, and two levels below root.

When you specify a file by using frontslashes to demarcate the various levels, you have a mechanism of identifying a file uniquely. No two files in a UNIX system can have identical absolute pathnames. If you have two files with the same name, they must be in different directories. This means that their pathnames will also be different. Thus, the file `/home/romeo/progs/count.pl` can coexist with the file `/home/romeo/safe/count.pl`.


Tip

If you are writing shell scripts that refer to files using an absolute pathname like `/home/romeo/progs`, you would do well to replace the component representing the home directory with the `HOME` variable—like `$HOME/progs`. If you now move your scripts to a different system where your home directory is, say `/u2/romeo`, the scripts will still work because `$HOME` always evaluates to the home directory in that system that's running the script.

6.6.1 Using the Absolute Pathname for a Command

A command runs in UNIX by executing its disk file. When you specify the **`date`** command, the system has to locate the file `date` from a list of directories specified in the `PATH` variable and then execute it. However, if you know the location of a particular command, you can also precede its name with the complete path. Since the file `date` resides in `/bin`, you can also use the absolute pathname:

```
$ /bin/date
Tue Feb 15 12:18:37 EST 2000
```

Nobody runs the **`date`** command like that. For any command that resides in the directories specified in the `PATH` variable, you don't need to use the absolute pathname. This `PATH`, you'll recall *(2.2)*, invariably has the directories `/bin` and `/usr/bin` in its list. Most UNIX commands meant for general use reside in these directories.


Note

If you execute programs residing in a directory that isn't in `PATH`, then the absolute pathname must be used. For example, to execute the program **`less`** residing in `/usr/local/bin`, you need to enter the absolute pathname:

```
/usr/local/bin/less
```

If you are frequently accessing programs in a certain directory, then it's better to include the directory itself in the `PATH`. The technique of doing that is shown in Chapter 17.

6.7 cd: Changing Directories

You can move around in the file system by using the **`cd`** (change directory) command. When used with an argument, it changes the current directory to the directory specified as the argument. To change to the directory `progs`, use **`cd progs`**. Check your current directory with **`pwd`** both before and after using **`cd`**:

```
$ pwd
/home/romeo
$ cd progs                    progs must be in current directory
$ pwd
/home/romeo/progs
```

Though **pwd** displays the absolute pathname, **cd** doesn't need to use one when the directory that you are switching to is in the current directory itself. The command **cd progs** here means: "Change your subdirectory to `progs` that is under the current directory." Using an absolute pathname causes no harm either; use **cd /home/romeo/progs** for the same effect.

Now, if `progs` also contains a directory `scripts` under it, then you have to use this command from the home directory to change to that directory:

```
cd progs/scripts                                    progs is in current directory
```

You can also see the file `convert.sh` in that directory with this command:

```
cat progs/scripts/convert.sh
```

Here, we have pathnames that have slashes between the files, but they are not absolute pathnames—rather, **relative pathnames** in their simplest form. However, to change to the `/bin` directory you need the absolute pathname:

```
$ pwd
/home/romeo/progs
$ cd /bin                                   Absolute pathname required here because
$ pwd                                                bin isn't in current directory
/bin
```

We can also navigate to `/bin` (or any directory) using a relative pathname; we are coming to that shortly. Navigation with the **cd** command using mostly absolute pathnames is illustrated in Fig. 6.2.

6.7.1 Using cd without Arguments

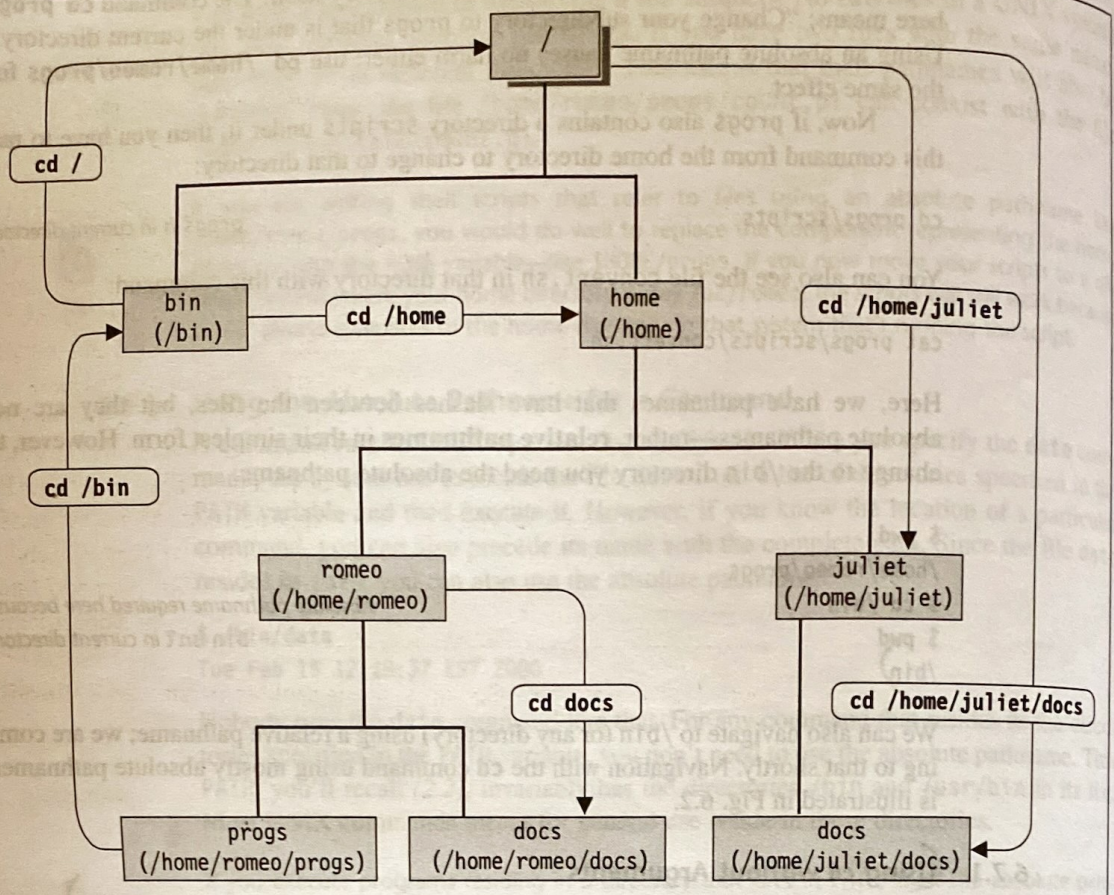
The **cd** command does something special when it is used without an argument:

```
$ pwd
/home/romeo/progs
$ cd
$ pwd
/home/romeo                    cd used without arguments reverts to home directory
```

Attention, Windows users! This command invoked without an argument doesn't indicate the current directory! It simply switches to the home directory—the directory where the user originally logged on to. Therefore, if you wander around in the file system, you can force an immediate return to your home directory by simply using **cd**:

```
$ cd /home/juliet
$ pwd
/home/juliet
$ cd                                                    Returns to home directory
$ pwd
/home/romeo
```

FIGURE 6.2 *Navigation with the* **cd** *Command*



Note: Absolute pathnames indicated in parentheses

The **cd** command can sometimes fail if you don't have proper permissions to access the directory. This doesn't normally happen unless you deliberately tamper with the permissions of the directory. File and directory permissions are discussed in the next chapter.


Note

When **cd** is invoked without arguments, it simply reverts to its home directory. It doesn't show you the current directory!

6.8 Relative Pathnames (. and ..)

In the preceding example, you changed your directory from `/home/romeo` to `/home/juliet` by using **cd** with an absolute pathname:

```
cd /home/juliet
```

It's unusual that to move to a directory which has the same parent (/home) as the current directory, you have to use an absolute pathname! Further, how do you move from here to /home then? Will you use **cd /home**? This method can be tedious when the absolute pathname is long, i.e., when you are located a number of "generations" away from root. For this, UNIX offers a shortcut—the **relative pathname**. You have seen its rudimentary form before *(6.7)*, but it's best known for its use of two cryptic symbols:

- . (a single dot)—This represents the current directory.
- .. (two dots)—This represents the parent directory.

We'll now use the .. to frame relative pathnames. You would have already guessed that to move to the parent directory, you have to use **cd ..** (two dots). Now, try this out:

```
$ pwd
/home/romeo/progs
$ cd ..                                    Moves one level up
$ pwd
/home/romeo
```

This method is compact and more useful when ascending the hierarchy. The command **cd ..** translates to this: Change your directory to the parent of the current directory.

The .. can form a component of a pathname too. You can combine any number of such sets of .. separated by /s. However, when a / is used with .. it acquires a different meaning; instead of moving down a level, it moves one level *up*. For instance, to move to /home, you can always use **cd /home**. Alternatively, you can also use a relative pathname:

```
$ pwd
/home/romeo/progs
$ cd ../..                                 Moves two levels up
$ pwd
/home
```

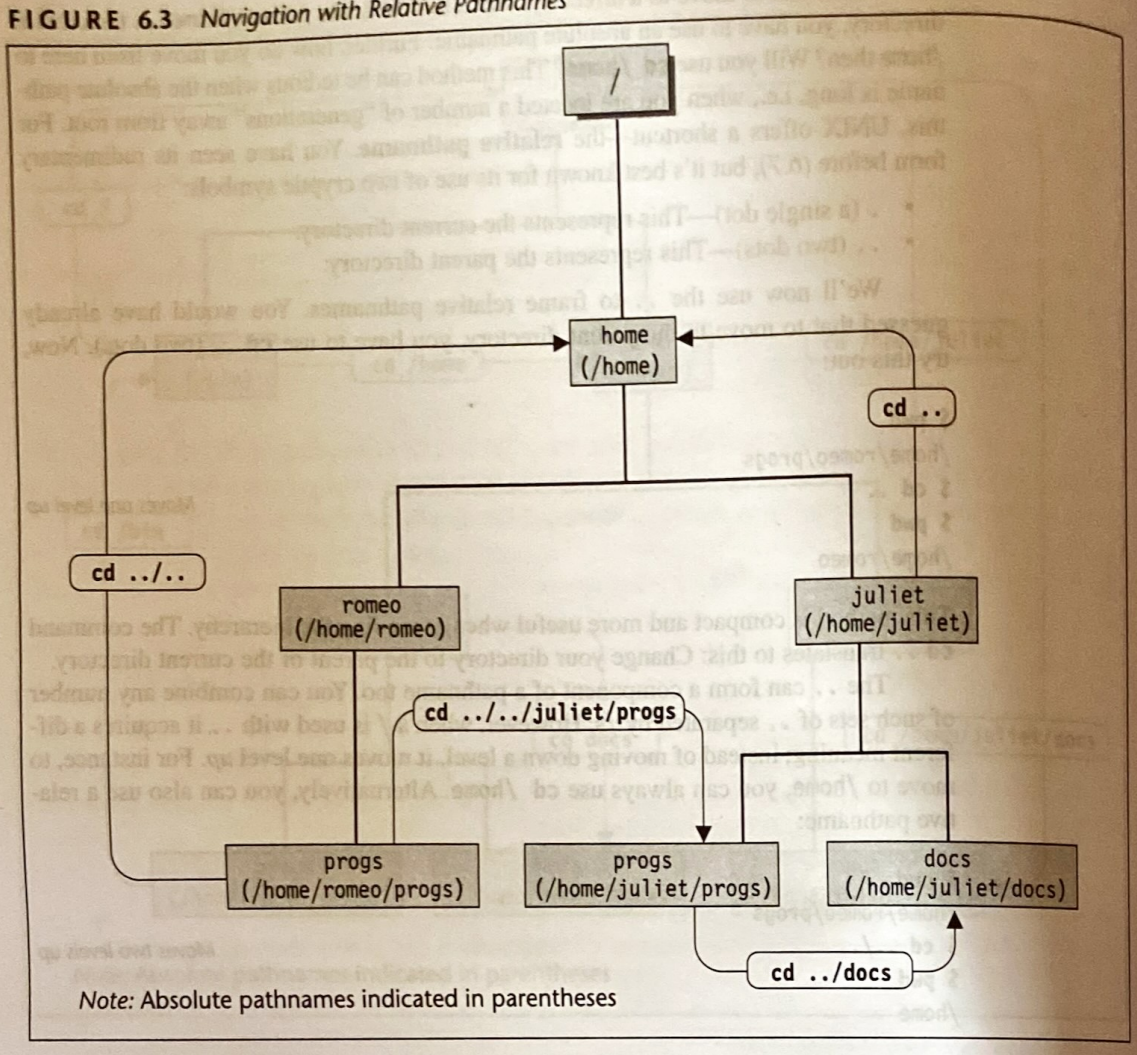
Now, how does one move from /home/romeo to /home/juliet? This requires a combination of the .. and the destination directory name in the pathname:

```
$ pwd
/home/romeo
$ cd ../juliet                             One level up and then down
$ pwd
/home/juliet
```

The use of relative pathnames using .. is depicted in Fig. 6.3. Now let's turn to the solitary dot that refers to the current directory. Any command which uses the current directory as an argument can also work with a single dot. This means that the **cp** command *(6.11)* which also uses a directory as the last argument can be used with a dot:

```
cp ../juliet/pricelist.html .
```

FIGURE 6.3 *Navigation with Relative Pathnames*



This copies the file `pricelist.html` to the current directory (`.`). Note that you didn't have to specify the filename of the copy; it's the same as the original one.


Note

When you use `../..`, the two dots on the right of the / represent the parent directory of the directory signified by the two dots on the left of the /.

You'll sometimes need to precede a *command* with `./` (a dot and a /). Let's assume that the current directory holds a **`cat`** program written by you, which is different from the UNIX **`cat`** in `/bin`. Now **`cat note`** will execute the **`cat`** command in `/bin` because `/bin` occurs earlier than `.` in PATH. (Use **`echo $PATH`** to check that out.) Using `./`, you can run your own **`cat`** and ignore the one in `/bin`:

```
./cat note                                    cat in current directory
```

Observe that here both the command and its argument reside in the current directory.


Note

Whether you should use an absolute or a relative pathname depends solely on the comparative number of keystrokes required to describe the pathname. In every case here, the relative pathname required fewer key depressions. Depending on where you are currently placed, an absolute pathname can sometimes require fewer keystrokes.

6.9 mkdir: Making Directories

Directories are created with the **mkdir** (make directory) command. The command is followed by the names of the directories to be created. A directory `patch` is created under the current directory like this:

```
mkdir patch
```

mkdir takes multiple arguments; you can create a number of subdirectories with one **mkdir** command:

```
mkdir patch dbs doc                    Three directories created
```

So far, simple enough, but the UNIX system goes further and lets you create a directory tree with just one invocation of the command. For instance, the following command creates a directory tree:

```
mkdir pis pis/progs pis/data           Creates the directory tree
```

This creates three subdirectories—`pis`, and two subdirectories under `pis`. The order of specifying the arguments is important; you obviously can't create a subdirectory before creation of its parent directory. For instance, you can't enter

```
$ mkdir pis/data pis/progs pis
mkdir: can't access pis/.
mkdir: can't access pis/.
```

Note that even though the system failed to create the two subdirectories `progs` and `data`, it has still created the `pis` directory.

Sometimes, the system refuses to create a directory:

```
$ mkdir test
mkdir: Failed to make directory "test"; Permission denied
```

This can happen for these reasons:

- The directory `test` may already exist.
- There may be an ordinary file by that name in the current directory.
- The permissions set for the current directory don't permit the creation of files and directories by the user.

We'll take up file permissions in the next chapter.

6.10 rmdir: Removing Directories

The **rmdir** (remove directory) command removes directories. You have to do this to remove the directory pis:

```
rmdir pis
```

Like **mkdir**, **rmdir** can also delete more than one directory in one shot. For instance, the three directories and subdirectories just created with **mkdir** can be removed by using **rmdir** with a reversed set of arguments:

```
rmdir pis/data pis/progs pis                    Arguments reversed this time
```

Note that when you delete a directory and its subdirectories, a reverse logic has to be applied. The following directory sequence used by **mkdir** is invalid in **rmdir**:

```
$ rmdir pis pis/progs pis/data
rmdir: pis: Directory not empty
```

Have you observed one thing from the error message? **rmdir** has silently deleted the lower level subdirectories, progs and data. There are two important rules that you should keep in mind when deleting directories:

- You can't use **rmdir** to delete a directory unless it is empty. In this case, the pis directory couldn't be removed because of the existence of the subdirectories progs and data under it.
- You can't remove a subdirectory unless you are placed in a directory which is hierarchically above the one you have chosen to remove. (This restriction doesn't apply in Linux.)

The first rule follows logically from the example above. But the UNIX **rm** command which works with ordinary files can be used with a special option (-r) to remove a complete directory structure. This is discussed in Section 6.12.1.

To illustrate the second cardinal rule, try removing the progs directory by executing the command from the same directory itself:

```
$ cd progs
$ pwd
/home/romeo/pis/progs
$ rmdir .                                   Trying to remove current directory works in Linux
rmdir: .: Can't remove current directory or ..
```

To remove this directory, you must position yourself in the directory above progs, i.e., pis, and then remove it from there:

```
$ cd ..
$ pwd                                                    Moves to parent directory
/home/romeo/pis
$ rmdir progs
$ _                                                      Command succeeded
```

The **mkdir** and **rmdir** commands work only with those directories *owned* by the user. Generally, a user is the owner of her home directory, and she can create and remove subdirectories (as well as regular files) in this directory or in any subdirectories created by her. However, she normally won't be able to create or remove files and directories in other users' directories. The concept of ownership will be discussed in the next chapter.


Note

Using **rmdir**, you can't remove a subdirectory unless it's empty, and you are positioned in its parent directory. However, with the **rm** command, you can remove any directory whether it is empty or not. Linux makes an exception; it allows you to delete the current directory with **rmdir .** but only if it is empty.

6.11 cp: Copying Files

We'll now take up the three basic commands that you'll need to use with files—**cp** (copy), **rm** (remove) and **mv** (move or rename). In this section, we take up **cp**; the other two are discussed in the next two sections.

The **cp** command copies a file or a group of files. It creates an exact image of the file on the disk with a different name. The syntax requires at least two filenames to be specified in the command line. When both are ordinary files, the first is copied to the second:

```
cp chap1 unit1
```

If the destination file (`unit1`) doesn't exist, it will first be created before copying takes place. If not, it will simply be overwritten without any warning from the system. So be careful when you choose your destination filename. Just check with an **ls** command whether or not the file exists.

What happens if `unit1` exists and is a directory instead of an ordinary file? In that case, `chap1` is copied to that directory with the same name. It's like using

```
cp chap1 unit1/chap1                    Same as cp chap1 unit1
```

Using a directory name as the destination, you can copy multiple files into the directory using a single **cp** command.

cp is often used with the shorthand notation **.** (dot) to signify the current directory as the destination. For instance, to copy the file `.profile` from `/home/juliet` to your current directory, you can use either of the two commands:

```
cp /home/juliet/.profile .profile       Destination is a file
cp /home/juliet/.profile .              Destination is current directory
```

Obviously, the second one is preferable because it requires fewer keystrokes.

As discussed before, **cp** can also be used to copy more than one file with a single invocation of the command. In that case, the last filename *must* be a directory. For instance, to copy the files `chap01`, `chap02` and `chap03` to the `progs` directory, you'll have to use **cp** like this:

```
cp chap01 chap02 chap03 progs           progs must be a directory
```

The files retain their original names in the progs directory. If these files are already resident in progs, **cp** will overwrite them. For the above command to work, the progs directory must exist because **cp** won't create it.

The UNIX system uses a set of special characters called **metacharacters** that you can use for matching more than one file. A detailed discussion on these characters is taken up in Chapter 8, but here we'll consider the *—one of the characters of the metacharacter set. If there were only three files in the current directory having the common string chap, you could compress the above sequence using the * as a suffix to chap:

```
cp chap* progs                    Copies all files beginning with chap
```

We'll continue to use the * as a shorthand for multiple filenames sharing a common string. If you are keen on using the other special characters, then look up Section 8.2.


Caution

cp overwrites without warning the destination file if it exists. Before using it, run **ls** to check if the file exists, and then use **cat** to check the contents of the file.

6.11.1 cp Options

Interactive Copying (-i) The -i (interactive) option warns the user before overwriting the destination file. If unit1 exists, **cp** prompts for a response:

```
$ cp -i chap1 unit1                    unit1 is an ordinary file here
cp: overwrite unit1? y
```

A y at this prompt overwrites the file; any other response leaves it uncopied. If you are afraid that you may accidentally overwrite a file, you can later use your knowledge of *aliases (17.4)* and *shell functions (19.10)* to make **cp** behave in this interactive manner by default.

Copying Directory Structures (-r) It's now possible to copy an entire directory structure with the -r (recursive) option. The following command copies all files and subdirectories in progs to newprogs:

```
cp -r progs newprogs
```

Since the process is recursive and copies all files resident in the subdirectories, **cp** here also has to create the subdirectories if it doesn't find them during the copying process. You can move entire directory structures this way.


Note

If the file to be copied is read-protected, it won't be possible to copy it. **cp** also won't work when the destination file exists and is write-protected. These issues are discussed and resolved in the next chapter.

6.12 rm: Deleting Files

The **rm** command deletes files and makes space available on disk. It normally operates silently and should be used with caution. It can delete more than one file with a single instruction:

```
rm chap01 chap02 chap03                    rm chap* should sometimes do
```

Unless used with the `-r` option, **rm** won't remove a directory, but it can remove files from one. You can remove the two chapters from the `progs` directory without having to "cd" to it:

```
rm progs/chap01 progs/chap02               Or rm progs/chap0[12]
```

You may sometimes need to delete all files of a directory, as part of a cleaning-up operation. The *, when used by itself, represents all files, and you can then use **rm** like this:

```
$ rm *                                     All files gone!
$ _
```

Windows users, beware! When you delete files in this fashion, the system won't prompt you with the message `Are you sure?` or `All files in directory will be deleted` before removing the files! The $ prompt will return silently, suggesting that the work has been done.

Caution Never issue a command like **rm *** before you check your current directory with **pwd** and use **ls** to list the files.

6.12.1 rm Options

Interactive Deletion (-i) There will often be a number of files that you'd like to delete, and some you'd like to retain. If you have 20 files to delete, you really won't want to specify all their names in the command line. You can then use **rm**'s `-i` (interactive) option, which makes the command ask the user for confirmation before removing each file:

```
$ rm -i chap01 chap02 chap03
chap01: ?y
chap02: ?n
chap03: ?y
```

A y removes the file; any other response leaves the file undeleted.

Recursive (and Dangerous) Deletion (-r) With the `-r` option, **rm** performs a tree walk—a thorough recursive search for all subdirectories and files within these

subdirectories. It then deletes all of them. **rm** *won't normally remove directories, but when used with this option, it will*. Therefore, when you issue the command

```
rm -r *                                    Current directory tree removed completely!
```

you'll delete all files in the current directory, as well as all subdirectories and their files. You must be doubly sure of what you are doing before you execute an **rm** command like this: it can be even more disastrous than **rm *.** Check your current directory and look at the listing of all its files and directories to ensure that there is not a single file that is important to you. If you don't have a backup, then these files will be lost forever.

rm won't delete any file if it's write-protected, but may prompt you for removal. The **-f** option overrides this protection also. It will force removal even if the files are write-protected:

```
rm -rf *                                   This is even more dangerous!
```

Note that you can't normally delete a file that you don't *own*. (Strictly speaking, this statement is not true.) **rm** doesn't send files to the Trash Bin (like Windows and X Window do); a file removed with **rm** is gone forever.

Caution

Unlike **rmdir**, **rm -r** can remove subdirectories even if they are not empty. If the root user (the super user) issues the command **rm -rf *** in the root directory, the entire UNIX system will be wiped out from the hard disk!

6.13 mv: Renaming Files

mv renames (moves) files and directories. It doesn't create a copy of the file but merely renames it. This means that no additional space is consumed in the disk by using **mv**. It has two functions:

- Renames a file (or directory).
- Moves a group of files to a different directory.

We'll first use **mv** on an ordinary file. To rename the file `chap01` to `man01`, you should use

```
mv chap01 man01                            man01 can also be a directory
```

If the destination file doesn't exist, it will be created. By default, **mv** doesn't prompt for overwriting the destination file if it exists. So be careful, again.

Like **cp**, a group of files can be moved, but only to a directory. The following command moves three files to the `progs` directory:

```
mv chap01 chap02 chap03 progs              progs must be a directory
```

mv can also be used to rename a directory and is used in exactly the same way as an ordinary file. There is a **-i** option available with **mv** also, which behaves exactly like in **cp**. So you can protect your files from being overwritten by using this option. The messages are the same, and require a similar response.