CHAPTER 2
UNIX FUNDAMENTALS

The UNIX operating system is a collection of programs that controls and organizes the resources and activities of a computer system. These resources consist of hardware such as the computer's memory, various peripherals such as terminals, printers, and disk drives, and software utilities that perform specific tasks on the computer system. UNIX is a multiuser, multitasking operating system that allows the computer to perform a variety of functions for many users. It also provides users with an environment in which they can access the computer's resources and utilities. This environment is characterized by its command interpreter, the shell.

In this chapter, we review a set of basic concepts for users working in the UNIX environment. As we mentioned in the preface, this book does not replace a general introduction to UNIX. A complete overview is essential to anyone not familiar with the file system, input and output redirection, pipes and filters, and many basic utilities. In addition, there are different versions of UNIX, and not all commands are identical in each version. In writing this book, we've used System V Release 2 on a Convergent Technologies' Miniframe.

These disclaimers aside, if it has been a while since you tackled a general introduction, this chapter should help refresh your memory. If you are already familiar with UNIX, you can skip or skim this chapter.

As we explain these basic concepts, using a tutorial approach, we demonstrate the broad capabilities of UNIX as an applications environment for text-processing. What you learn about UNIX in general can be applied to performing specific tasks related to text-processing.

▫ The UNIX Shell ▫

As an interactive computer system, UNIX provides a command interpreter called a shell. The shell accepts commands typed at your terminal, invokes a program to perform specific tasks on the computer, and handles the output or result of this program, normally directing it to the terminal's video display screen.

UNIX commands can be simple one-word entries like the `date` command:

```
$ date
Tue Apr 8 13:23:41 EST 1987
```

Or their usage can be more complex, requiring that you specify options and arguments, such as filenames. Although some commands have a peculiar syntax, many UNIX commands follow this general form:

command option(s) argument(s)

A *command* identifies a software program or utility. Commands are entered in lowercase letters. One typical command, `ls`, lists the files that are available in your immediate storage area, or *directory*.

An *option* modifies the way in which a command works. Usually options are indicated by a minus sign followed by a single letter. For example, `ls -l` modifies what information is displayed about a file. The set of possible options is particular to the command and generally only a few of them are regularly used. However, if you want to modify a command to perform in a special manner, be sure to consult a UNIX reference guide and examine the available options.

An *argument* can specify an expression or the name of a file on which the command is to act. Arguments may also be required when you specify certain options. In addition, if more than one filename is being specified, special *metacharacters* (such as `*` and `?`) can be used to represent the filenames. For instance, `ls -l ch*` will display information about all files that have names beginning with `ch`.

The UNIX shell is itself a program that is invoked as part of the login process. When you have properly identified yourself by logging in, the UNIX system prompt appears on your terminal screen.

The prompt that appears on your screen may be different from the one shown in the examples in this book. There are two widely used shells: the Bourne shell and the C shell. Traditionally, the Bourne shell uses a dollar sign (`$`) as a system prompt, and the C shell uses a percent sign (`%`). The two shells differ in the features they provide and in the syntax of their programming constructs. However, they are fundamentally very similar. In this book, we use the Bourne shell.

Your prompt may be different from either of these traditional prompts. This is because the UNIX environment can be customized and the prompt may have been changed by your system administrator. Whatever the prompt looks like, when it appears, the system is ready for you to enter a command.

When you type a command from the keyboard, the characters are echoed on the screen. The shell does not interpret the command until you press the *RETURN* key. This means that you can use the *erase character* (usually the *DEL* or *BACKSPACE* key) to correct typing mistakes. After you have entered a command line, the shell tries to identify and locate the program specified on the command line. If the command line that you entered is not valid, then an error message is returned.

When a program is invoked and processing begun, the output it produces is sent to your screen, unless otherwise directed. To interrupt and cancel a program before it has completed, you can press the *interrupt character* (usually *CTRL-C* or the *DEL* key). If the output of a command scrolls by the screen too fast, you can suspend the output by pressing the *suspend character* (usually *CTRL-S*) and resume it by pressing the *resume character* (usually *CTRL-Q*).

Some commands invoke utilities that offer their own environment—with a command interpreter and a set of special "internal" commands. A text editor is one such utility, the mail facility another. In both instances, you enter commands while you are "inside" the program. In these kinds of programs, you must use a command to exit and return to the system prompt.

The return of the system prompt signals that a command is finished and that you can enter another command. Familiarity with the power and flexibility of the UNIX shell is essential to working productively in the UNIX environment.

▫ Output Redirection ▫

Some programs do their work in silence, but most produce some kind of result, or output. There are generally two types of output: the expected result—referred to as *standard output*—and error messages—referred to as *standard error*. Both types of output are normally sent to the screen and appear to be indistinguishable. However, they can be manipulated separately—a feature we will later put to good use.

Let's look at some examples. The `echo` command is a simple command that displays a string of text on the screen.

```
$ echo my name
my name
```

In this case, the input `echo my name` is processed and its output is `my name`. The name of the command—`echo`—refers to a program that interprets the command line arguments as a literal expression that is sent to standard output. Let's replace `echo` with a different command called `cat`:

```
$ cat my name
cat: Cannot open my
cat: Cannot open name
```

The `cat` program takes its arguments to be the names of files. If these files existed, their contents would be displayed on the screen. Because the arguments were not filenames in this example, an error message was printed instead.

The output from a command can be sent to a file instead of the screen by using the output redirection operator (>). In the next example, we redirect the output of the `echo` command to a file named `reminders`.

```
$ echo Call home at 3:00 > reminders
$
```

No output is sent to the screen, and the UNIX prompt returns when the program is finished. Now the `cat` command should work because we have created a file.

```
$ cat reminders
Call home at 3:00
```

The `cat` command displays the contents of the file named `reminders` on the screen. If we redirect again to the same filename, we overwrite its previous contents:

```
$ echo Pick up expense voucher > reminders
$ cat reminders
Pick up expense voucher
```

We can send another line to the file, but we have to use a different redirect operator to append (>>) the new line at the end of the file:

```
$ echo Call home at 3:00 > reminders
$ echo Pick up expense voucher >> reminders
$ cat reminders
Call home at 3:00
Pick up expense voucher
```

The `cat` command is useful not only for printing a file on the screen, but for con*cat*enating existing files (printing them one after the other). For example:

```
$ cat reminders todolist
Call home at 3:00
Pick up expense voucher
Proofread Chapter 2
Discuss output redirection
```

The combined output can also be redirected:

```
$ cat reminders todolist > do_now
```

The contents of both `reminders` and `todolist` are combined into `do_now`. The original files remain intact.

If one of the files does not exist, an error message is printed, even though standard output is redirected:

```
$ rm todolist
$ cat reminders todolist > do_now
cat: todolist: not found
```

The files we've created are stored in our *current working directory*.

Files and Directories

The UNIX file system consists of files and directories. Because the file system can contain thousands of files, directories perform the same function as file drawers in a paper file system. They organize files into more manageable groupings. The file system is hierarchical. It can be represented as an inverted tree structure with the *root directory* at the top. The root directory contains other directories that in turn contain other directories.*

On many UNIX systems, users store their files in the `/usr` file system. (As disk storage has become cheaper and larger, the placement of user directories is no longer standard. For example, on our system, `/usr` contains only UNIX software; user accounts are in a separate file system called `/work`).

*In addition to subdirectories, the root directory can contain other *file systems*. A file system is the skeletal structure of a directory tree, which is built on a magnetic disk before any files or directories are stored on it. On a system containing more than one disk, or on a disk divided into several partitions, there are multiple file systems. However, this is generally invisible to the user, because the secondary file systems are *mounted* on the root directory, creating the illusion of a single file system.

Fred's *home directory* is /usr/fred. It is the location of Fred's account on the system. When he logs in, his home directory is his current working directory. Your working directory is where you are currently located and changes as you move up and down the file system.

A *pathname* specifies the location of a directory or file on the UNIX file system. An *absolute pathname* specifies where a file or directory is located off the root file system. A *relative pathname* specifies the location of a file or directory in relation to the current working directory.

To find out the pathname of our current directory, enter pwd.

```
$ pwd
/usr/fred
```

The absolute pathname of the current working directory is /usr/fred. The ls command lists the contents of the current directory. Let's list the files and subdirectories in /usr/fred by entering the ls command with the -F option. This option prints a slash (/) following the names of subdirectories. In the following example, oldstuff is a directory, and notes and reminders are files.

```
$ ls -F
reminders
notes
oldstuff/
```

When you specify a filename with the ls command, it simply prints the name of the file, if the file exists. When you specify the name of directory, it prints the names of the files and subdirectories in that directory.

```
$ ls reminders
reminders
$ ls oldstuff
ch01_draft
letter.212
memo
```

In this example, a relative pathname is used to specify oldstuff. That is, its location is specified in relation to the current directory, /usr/fred. You could also enter an absolute pathname, as in the following example:

```
$ ls /usr/fred/oldstuff
ch01_draft
letter.212
memo
```

Similarly, you can use an absolute or relative pathname to change directories using the cd command. To move from /usr/fred to /usr/fred/oldstuff, you can enter a relative pathname:

```
$ cd oldstuff
$ pwd
/usr/fred/oldstuff
```

The directory /usr/fred/oldstuff becomes the current working directory.

The cd command without an argument returns you to your home directory.

```
$ cd
```

When you log in, you are positioned in your home directory, which is thus your current working directory. The name of your home directory is stored in a shell variable that is accessible by prefacing the name of the variable (HOME) with a dollar sign ($). Thus:

```
$ echo $HOME
/usr/fred
```

You could also use this variable in pathnames to specify a file or directory in your home directory.

```
$ ls $HOME/oldstuff/memo
/usr/fred/oldstuff/memo
```

In this tutorial, /usr/fred is our home directory.

The command to create a directory is mkdir. An absolute or relative pathname can be specified.

```
$ mkdir /usr/fred/reports
```

```
$ mkdir reports/monthly
```

Setting up directories is a convenient method of organizing your work on the system. For instance, in writing this book, we set up a directory `/work/textp` and, under that, subdirectories for each chapter in the book (`/work/textp/ch01`, `/work/textp/ch02`, etc.). In each of those subdirectories, there are files that divide the chapter into sections (`sect1`, `sect2`, etc.). There is also a subdirectory set up to hold old versions or drafts of these sections.

Copying and Moving Files

You can copy, move, and rename files within your current working directory or (by specifying the full pathname) within other directories on the file system. The `cp` command makes a copy of a file and the `mv` command can be used to move a file to a new directory or simply rename it. If you give the name of a new or existing file as the last argument to `cp` or `mv`, the file named in the first argument is copied, and the copy given the new name. (If the target file already exists, it will be overwritten by the copy. If you give the name of a directory as the last argument to `cp` or `mv`, the file or files named first will be copied to that directory, and will keep their original names).

Look at the following sequence of commands:

```
$ pwd                      Print working directory
/usr/fred
$ ls -F                    List contents of current directory
meeting
oldstuff/
notes
reports/
$ mv notes oldstuff          Move notes to oldstuff directory
$ ls                         List contents of current directory
meeting
oldstuff
reports/
$ mv meeting meet.306        Rename meeting
$ ls oldstuff                List contents of oldstuff subdirectory
ch01_draft
letter.212
memo
notes
```

In this example, the `mv` command was used to rename the file `meeting` and to move the file `notes` from `/usr/fred` to `/usr/fred/oldstuff`. You can also use the `mv` command to rename a directory itself.

Permissions

Access to UNIX files is governed by ownership and permissions. If you create a file, you are the owner of the file and can set the permissions for that file to give or deny access to other users of the system. There are three different levels of permission:

r Read permission allows users to read a file or make a copy of it.
w Write permission allows users to make changes to that file.
x Execute permission signifies a program file and allows other users to execute this program.

File permissions can be set for three different levels of ownership:

owner The user who created the file is its owner.

group	A group to which you are assigned, usually made up of those users engaged in similar activities and who need to share files among themselves.
other	All other users on the system, the public.

Thus, you can set read, write, and execute permissions for the three levels of ownership. This can be represented as:

```
         rwxrwxrwx
          /  |  \
```
owner group other

When you enter the command `ls -l`, information about the status of the file is displayed on the screen. You can determine what the file permissions are, who the owner of the file is, and with what group the file is associated.

```
$ ls -l meet.306
-rw-rw-r-- 1 fred  techpubs  126  March 6  10:32  meet.306
```

This file has read and write permissions set for the user `fred` and the group `techpubs`. All others can read the file, but they cannot modify it. Because `fred` is the owner of the file, he can change the permissions, making it available to others or denying them access to it. The `chmod` command is used to set permissions. For instance, if he wanted to make the file writeable by everyone, he would enter:

```
$ chmod o+w meet.306
$ ls -l meet.306
-rw-rw-rw- 1 fred  techpubs  126  March 6  10:32  meet.306
```

This translates to "add write permission (`+w`) to others (`o`)." If he wanted to remove write permission from a file, keeping anyone but himself from accidentally modifying a finished document, he might enter:

```
$ chmod go-w meet.306
$ ls -l meet.306
-rw-r--r-- 1 fred  techpubs  126  March 6  10:32  meet.306
```

This command removes write permission (`-w`) from group (`g`) and other (`o`).

File permissions are important in UNIX, especially when you start using a text editor to create and modify files. They can be used to protect information you have on the system.

▫ Special Characters ▫

As part of the shell environment, there are a few special characters (metacharacters) that make working in UNIX much easier. We won't review all the special characters, but enough of them to make sure you see how useful they are.

The *asterisk* (`*`) and the *question mark* (`?`) are filename generation metacharacters. The asterisk matches any or all characters in a string. By itself, the asterisk expands to all the names in the specified directory.

```
$ echo *
meet.306  oldstuff reports
```

In this example, the `echo` command displays in a row the names of all the files and directories in the current directory. The asterisk can also be used as a shorthand notation for specifying one or more files.

```
$ ls meet*
meet.306
$ ls /work/textp/ch*
/work/textp/ch01
/work/textp/ch02
/work/textp/ch03
/work/textp/chapter_make
```

The question mark matches any single character.

```
$ ls /work/textp/ch01/sect?
```

```
/work/textp/ch01/sect1
/work/textp/ch01/sect2
/work/textp/ch01/sect3
```

Besides filename metacharacters, there are other characters that have special meaning when placed in a command line. The *semicolon* (`;`) separates multiple commands on the same command line. Each command is executed in sequence from left to right, one before the other.

```
$ cd oldstuff;pwd;ls
/usr/fred/oldstuff
ch01_draft
letter.212
memo
notes
```

Another special character is the *ampersand* (`&`). The ampersand signifies that a command should be processed in the background, meaning that the shell does not wait for the program to finish before returning a system prompt. When a program takes a significant amount of processing time, it is best to have it run in the background so that you can do other work at your terminal in the meantime. We will demonstrate background processing in Chapter 4 when we look at the `nroff/troff` text formatter.

□ Environment Variables □

The shell stores useful information about who you are and what you are doing in *environment variables*. Entering the `set` command will display a list of the environment variables that are currently defined in your account.

```
$ set
PATH  .:bin:/usr/bin:/usr/local/bin:/etc
argv  ()
cwd   /work/textp/ch03
home  /usr/fred
shell /bin/sh
status0
TERM  wy50
```

These variables can be accessed from the command line by prefacing their name with a dollar sign:

```
$ echo $TERM
wy50
```

The `TERM` variable identifies what type of terminal you are using. It is important that you correctly define the `TERM` environment variable, especially because the `vi` text editor relies upon it. Shell variables can be reassigned from the command line. Some variables, such as `TERM`, need to be *exported* if they are reassigned, so that they are available to all shell processes.

```
$ TERM=tvi925; export TERM    Tell UNIX I'm using a Televideo 925
```

You can also define your own environment variables for use in commands.

```
$ friends="alice ed ralph"
$ echo $friends
alice ed ralph
```

You could use this variable when sending mail.

```
$ mail $friends
A message to friends
<CTRL-D>
```

This command sends the mail message to three people whose names are defined in the `friends` environment variable. Pathnames can also be assigned to environment variables, shortening the amount of typing:

```
$ pwd
/usr/fred
$ book="/work/textp"
$ cd $book
```

```
$ pwd
/work/textp
```

▫ Pipes and Filters ▫

Earlier we demonstrated how you can redirect the output of a command to a file. Normally, command input is taken from the keyboard and command output is displayed on the terminal screen. A program can be thought of as processing a stream of input and producing a stream of output. As we have seen, this stream can be redirected to a file. In addition, it can originate from or be passed to another command.

A *pipe* is formed when the output of one command is sent as input to the next command. For example:

```
$ ls | wc
```

might produce:

```
10      10      72
```

The `ls` command produces a list of filenames which is provided as input to `wc`. The `wc` command counts the number of lines, words, and characters.

Any program that takes its input from another program, performs some operation on that input, and writes the result to the standard output is referred to as a *filter*. Most UNIX programs are designed to work as filters. This is one reason why UNIX programs do not print "friendly" prompts or other extraneous information to the user. Because all programs expect—and produce—only a data stream, that data stream can easily be processed by multiple programs in sequence.

One of the most common uses of filters is to process output from a command. Usually, the processing modifies it by rearranging it or reducing the amount of information it displays. For example:

```
$ who                               List who is on the system, and at which terminal
peter       tty001      Mar  6 17:12
walter      tty003      Mar  6 13:51
chris       tty004      Mar  6 15:53
val         tty020      Mar  6 15:48
tim         tty005      Mar  4 17:23
ruth        tty006      Mar  6 17:02
fred        tty000      Mar  6 10:34
dale        tty008      Mar  6 15:26
$ who | sort                        List the same information in alphabetic order
chris       tty004      Mar  6 15:53
dale        tty008      Mar  6 15:26
fred        ttY000      Mar  6 10:34
peter       tty001      Mar  6 17:12
ruth        tty006      Mar  6 17:02
tim         tty005      Mar  4 17:23
val         tty020      Mar  6 15:48
walter      tty003      Mar  6 13:51
$
```

The `sort` program arranges lines of input in alphabetic or numeric order. It sorts lines alphabetically by default. Another frequently used filter, especially in text-processing environments, is `grep`, perhaps UNIX's most renowned program. The `grep` program selects lines containing a pattern:

```
$ who | grep tty001          Find out who is on terminal 1
peter      tty001      Mar  6 17:12
```

One of the beauties of UNIX is that almost any program can be used to filter the output of any other. The pipe is the master key to building command sequences that go beyond the capabilities provided by a single program and allow users to create "custom" programs of their own to meet specific needs.

If a command line gets too long to fit on a single screen line, simply type a backslash followed by a carriage return, or (if a pipe symbol comes at the appropriate place) a pipe symbol followed by a carriage return. Instead of executing the command, the shell will give you a secondary prompt (usually >) so you can continue the line:

```
$ echo This is a long line shown here as a demonstration |
>  wc
      1      10      49
```

This feature works in the Bourne shell only.

▫ Shell Scripts ▫

A *shell script* is a file that contains a sequence of UNIX commands. Part of the flexibility of UNIX is that anything you enter from the terminal can be put in a file and executed. To give a simple example, we'll assume that the last command example (`grep`) has been stored in a file called `whoison`:

```
$ cat whoison
who | grep tty001
```

The permissions on this file must be changed to make it executable. After a file is made executable, its name can be entered as a command.

```
$ chmod +x whoison
$ ls -l whoison
-rwxrwxr-x    1 fred     doc          123 Mar  6 17:34 whois
$ whoison
peter      tty001        Mar  6 17:12
```

Shell scripts can do more than simply function as a batch command facility. The basic constructs of a programming language are available for use in a shell script, allowing users to perform a variety of complicated tasks with relatively simple programs.

The simple shell script shown above is not very useful because it is too specific. However, instead of specifying the name of a single terminal line in the file, we can read the name as an argument on the command line. In a shell script, `$1` represents the first argument on the command line.

```
$ cat whoison
who | grep $1
```

Now we can find who is logged on to any terminal:

```
$ whoison tty004
chris      tty004        Mar  6 15:53
```

Later in this book, we will look at shell scripts in detail. They are an important part of the writer's toolbox, because they provide the "glue" for users of the UNIX system—the mechanism by which all the other tools can be made to work together.