THE UNIX ENVIRONMENT:

UNIX is a multiuser, multi processing, portable system designed to facilitate programming, text processing, communication, and many other tasks that are expected from an operating system.

It contains hundreds of simple, single –purpose functions that can be combined to do virtually every processing task imaginable. Its flexibility is demonstrated in that it is used in three different computing environments: stand-alone personal environment, time sharing systems, and client/server systems.

PERSONAL EVNIRONMENT:

Although originally designed as a multiuser environment, many users are installing UNIX on their personal computers. This trend to personal UNIX systems accelerated in the mid-1990's with the availability of Linux, a free UNIX system. The apple system X released in 2001 incorporated UNIX as its kernel.

TIME SHARING ENVIRONMENT:

In time sharing environment, many users are connected to one or more computers. Their terminals are often nonprogrammable, although today we see more and more microcomputers being used to simulate terminals.

Also in time sharing environment, the output devices like printers and auxiliary storage devices like disks are shared by all of the users. A typical college lab in which a mini computer is shared by many students is as shown in figure 1.1.

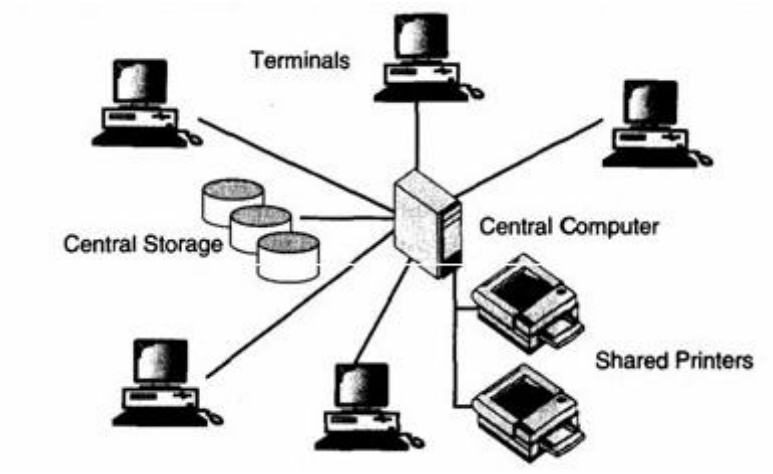


FIGURE 1.1 TIME SHARING ENVIRONMENT

In time sharing environment all of the computing must be done by the central computer. In other words, the central computer has many duties: it must control the shared resources, it must manage the shared data and printing and it must also do the computing. All of this work tends to keep the computer busy.

CLIENT/SERVER ENVIRONMENT:

A client/server environment splits the computing function between a central computer and user's computers. The users are given personal computers or workstations so that some of the computation responsibility can be moved off the central computer and assigned to the workstations.

In the client/server environment, the user's microcomputers or workstations are called the **client**. The central computer which may be a powerful microcomputer, a minicomputer, or a central mainframe system, is known as the **server**.

Because the work is shared between the user's computers and central computers, response time and monitor display are faster and the users are more productive.

UNIX STRUCTURE:

UNIX consists of four major components: the kernel, the shell, a standard set of utilities, and application programs. These components are as shown in the following figure:

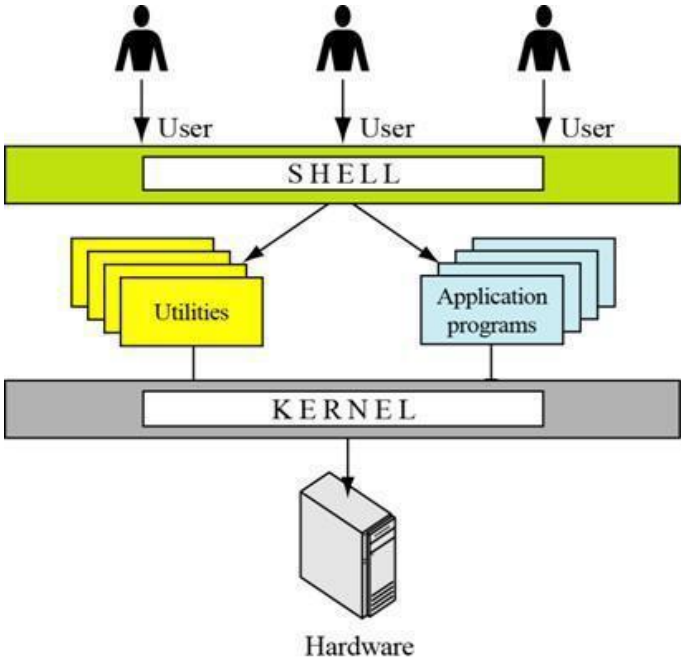


FIGURE 1.2: COMPONENTS OF UNIX

THE KERNEL:

The kernel is the heart of the UNIX system. It contains two most basic parts of the operating system: process control and resource management. All other components of the system call on the kernel to perform these services for them.

THE SHELL:

The shell is the part of the UNIX that is most visible to the user. It receives and interprets the commands entered by the user. In many respects this makes it the most important component of the UNIX structure. It is certainly the part that we, as users, get to know the most. To do anything in the system, we must give the shell a command. If the command requires a utility, the shell requests that the kernel execute the utility. If the command requires an application program, the requests that it be run.

Some of the stand UNIX shells are

- Bourne shell
- Bash
- C shell
- tcsh
- Korn

There are two major parts of the shell. The first is interpreter. The interpreter reads your commands and works with the kernel to execute them. The second part of the shell is a programming capability that allows you to write a shell (command) script.

A shell script is a file that contains shell commands that perform a useful function. It is also known as a shell program. There are three standard shells in UNIX today.

The ***Bourne shell*** developed by Steve Bourne at the AT & T labs is the oldest. Because it is the oldest and the most primitive, it is not used on many systems today. An enhanced version of Bourne shell called Bash (Bourne again shell) is used in Linux.

The ***C shell*** developed in Berkeley by Bill Joy, received its name from the fact that its commands were supposed to look like C statements. A compatible version of the C shell, tcsh is used in Linux.

The ***Korn shell*** developed by David Korn, also of the AT&T labs is the newest and most powerful. Because it was developed at AT&T labs, it is compatible with the Bourne shell.

UTILITIES:

There are hundreds of UNIX utilities. A **utility** is a standard UNIX program that provides a support process for users. Three common utilities are *text editors*, *search programs*, and *sort programs*.

Many of the system utilities are actually sophisticated applications. For example, the UNIX email system is considered a utility as are the three common text editors, **vi, emacs,** and **pico.** All four of these utilities are large systems in themselves. Other utilities are short, simple functions. For example the list (**ls**) utility displays the files that reside on a disk.

APPLICATIONS:

Applications are programs that are not a standard part of UNIX. Written by system administrators, professional programmers, or users, they provide an extended capability to the system.

ACCESSING UNIX:

To begin you need to **log in** to the system before doing any work with UNIX. Once you logged in, you enter commands and the system responds. When you have finished your work, you log out. The time spent you working with the system is known as a **session**.

USER ID:

When you work with your own computer at home, you don't need to log in or too concerned about who uses the system. When you work in a UNIX environment, however, security and user control become major concerns. Generally you cannot access the computer until you have been given permission to do so. Permissions come in the form of an account created by the system administrator (sys admin). You and your account are identified by a special code known as **user id**.

PASSWORDS:

To ensure that it is really you at the other end of the line, you must enter a password. A **password** is a secret code that you supply to the server and that is known only to you. UNIX encrypts passwords when it stores them in the server so that no one can figure out what they are. Not even the sys admin, who has absolute control over the server, can tell you what your password is if you forget it. All he or she can do is **reset** it so that you can create a new one.

INTERACTIVE SESSION:

The interactive session contains three steps: login, interaction and logout.

LOGIN:

The details of the **login** process vary from system to system. There is a general pattern to the steps, however, as listed here.

1. You must make contact with the system.
   If you are working on a local network and are always connected to a remote server, starting the **login** process is as simple as selecting an option in a menu. On the other hand, if you are making the connection from a remote location, such as from home to work, then you will need to use special connection software, often referred to as **Telnet** software.
2. Wait for the system **login** prompt.
   Once you have connected to the server, you must ask for the server to ask you to identify yourself. Note that good security requires that the server give you only the minimum information you need to make the connection. A typical **login** prompt is:
   login:
3. Type user id.
   Once the server responded with a request for you to identify yourself, enter your user id. Note that UNIX is a case-sensitive system.
4. Type your password.
   After you enter your user id, the system will prompt you for your password. The password prompt is almost always the word Password on a new line. As you type the password, it will not be displayed on the screen. This is another security caution-somebody may be watching over your shoulder to learn your password.
   If you do everything correctly you will see a shell prompt. If you make a mistake, the system will give you a cryptic error message, such as “login incorrect”, and ask for your user id again.

The default system prompt for the Bourne, bash, and korn shells is a dollar sign ($). For the C shell and tcsh shell, the prompt is a percentage sign (%).

INTERACTION:

Once you connect to the server you can enter commands that allow you to work with the computer. Typical commands allow you to work with files – edit, copy, and sort; process data and print the result; send and receive mail; and many other processing operations.

LOGOUT:

It is very important that you log out when you are through with the system. There are several reasons for this. First, it frees system resources for others who may need to use them. More

important, it is a security concern to leave to terminal logged in with no one working at it. Some unauthorized person may walkup and gain access to the system and your files, if you do not log out.

Although there are variations in the logout command, it is most typically the typed command, **logout**, at the system prompt.

A typical user session is as follows:

```
IRIX (voyager)
        This system is for the use of authorized users only.
login: gilberg
password:

UNIX BSD Release 4.0

        Welcome . . .

$ ls
file1 file2 file3

$ cat file1

        Hello World!
        .
        .
        .
$ logout
```


COMMANDS:

The basis of all UNIX interaction is the command. Commands are not unique to UNIX. Many other systems, most notably MS-DOS, use commands. While they are generally a single line entered at a console, they can also be included in executable files to form scripts.

BASIC CONCEPTS:

A UNIX command is an action request given to the UNIX shell for execution. The simplest commands are a single line entered at the command line prompt that cause a program or shell script to be executed. Often the program is a UNIX utility; it may also be an application program.

COMMAND SYNTAX:

Commands are entered at the shell prompt. You must see the prompt, such as the Korn shell $ prompt, before you can enter a command.

Every command must have a verb and may also have options and arguments. The command format is:

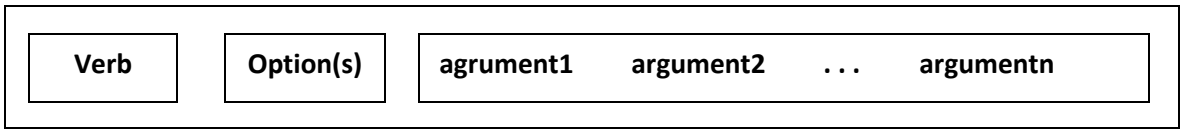
$ **verb [options] [arguments]**

The brackets indicate that the options and arguments are optional. We use this notation when we describe individual commands. If an option or argument is not in brackets, it is required.

The **verb** is the command name. The command indicates that what action is to be taken. This action concept gives us the name *verb* for action.

The **option** modifies how the action is applied. For example, when we display the date we can use an option to specify if want the time in Greenwich Mean Time or local time. Options are usually one character preceded by a minus sign or a plus sign. Many commands however have multiple options available.

Finally the **argument** provides additional information to the command. For example, when displaying the contents of a file, an argument can be used to specify the name of the file. Some commands have no arguments, some accept only one argument and some accept multiple arguments. You must know for each command you use, what are the options and arguments. The general syntax or format of a command appears as below:



COMMON COMMANDS:

Date and Time command:

The **date** command displays the system date and time. If the system is local – that is one in your own area – it is the current time. If the system is remote, such as across the country somewhere, the reply will contain the time where the system is physically located. By using an option you can get the current Greenwich Mean Time (GMT). Each date response indicates what time zone is being used. For example, 17:56:52 PST indicates that the time is Pacific Standard Time. The general format of the command is as given below: **date** *options argument*

The input for date is the system itself; the date is actually maintained in the computer as a part of the operating system. Most modern hardware also has a hardware date and time clock i.e. often updated automatically to ensure that it is accurate. The **date** command sends its response to the monitor.

Example:

$ **date**

Wed Mar 6 17:56:52 PST 2002

The date command has only one user option and argument. If no option used, the time is local time. if a –u option is used, the time is GMT.

Example:

$ **date**

Wed Apr 3 08:24:19 GMT 2002

The date command argument allows you to customize the format of the date. For example you can spell out the month and day or omit them entirely rather than use the standard abbreviations. To create your own format, you use arguments. Text may appear anywhere in the argument and is displayed just as it is entered. The output display follows the command on the console. For example:

$ **date "+ Today's date is: %D. The time is: %T"**

Today's date is: 03/15/02. The time is: 15:25:16

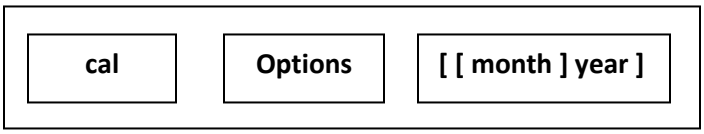
The date formats are show in the table given below:

Format Code	Explanation
a	Abbreviated weekday name, such as Mon
A	Full weekday name, such as Monday
b	Abbreviated month name, such as Jan
B	Full month name, such as January
d	Day of the month with two digits (leading zeros), such as 01, 02, . . ., 31
e	Day of the month with spaces replacing leading zeros, such as 1, 2 ,. . . . , 31
D	date in the format mm/dd/yy, such as 01/01/99
H	Military time two-digit hour, such as 00, 01, . . . , 23
I	Civilian time two-digit hour, such as 00, 01, . . . , 12
j	Julian date (day of the year), such as 001, 002, . . . , 366
m	Numeric two-digit month, such as 01, 02, ... , 12
M	Two digit minute such as 00, 01, . . . , 59

The **date** command can also be used to set the date and time, but only by a system administrator.

Calendar (cal) command:

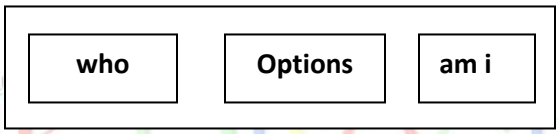
The calendar command, **cal** , displays the calendar for a specified month or for a year. It is an example of a command that has no options but uses arguments. Its general format is as show below:



As shown in the above figure, there are two arguments and no options for the calendar command. The arguments are optional: if no arguments are entered, the calendar for the current month is printed.

Who's online (who) command:

The **who** command displays all users currently logged in to the system. The general format of who command is presented as below:



The who command returns the user's name (id), terminal, and time he or she logged in. A basic **who** command is given as in the example below:

```
$ who
nb045527    ttyq0    Mar 15 15:23
tran        ttyq1    Mar 10 12:15
gilberg     ttyq5    Mar 15 14:57
```

Just knowing someone is logged in is not sufficient , however, you also want to know that he or she is active and not out getting a cup of coffee. In this case, you want to use –u **option**, which also indicates how long it has been since there was any activity on the line. This is known as **idle time**. It also returns the process id for the user. A who request with –u option is shown as below:

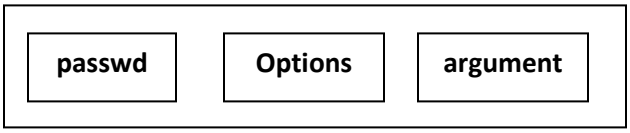
```
$ who -u
nb045527    ttyq0    Mar 15 15:23  0:41    19590
tran        ttyq1    Mar 10 12:15  old     8315
gilberg     ttyq5    Mar 15 14:57  .       2378
```

whoami:

If you key the **whoami** as command, the system returns your user id.

Change password (passwd) command:

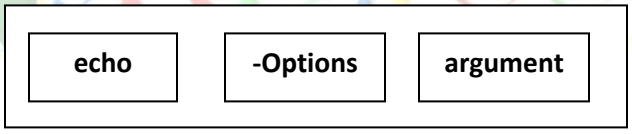
The password command, **passwd**, is used to change your password. It has no options or attributes but rather does its work through a dialog of questions and answers. The general format is shown in figure below:



It begins by asking you to enter your old password. While you might think that asking for your old password is unnecessary, it is done for security reasons: if you were to leave your area for even a minute, someone could come in and quickly change your password. Then they could later come back and gain access to your system. After you verify your password, the system asks you for a new password.

Print message (echo) command:

The **echo** command copies its argument back to the terminal. Its format appear as given below:

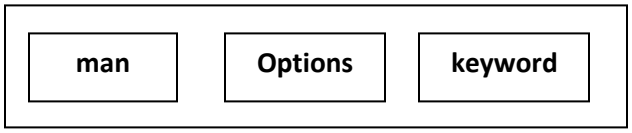


Example:

```
$ echo Hello World
Hello World
```

Online documentation (man) command:

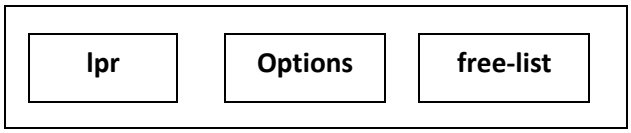
One of the most important UNIX commands is man. The man command displays online documentation. When you can't remember exactly what the options are for a command, you can quickly check the online manual and look up the answer. *There is even a manual explanation for the man command itself*. The general format of man command is as shown below:



Print (lpr) command:

The most common print utility is line printer (lpr). The line printer utility prints the contents of the specified files to either the default printer or to a specified printer. Multiple files can be printed with the same command. If no file is specified, the input comes from standard input, which is usually a keyboard unless it has been redirected.

To direct the output to a specified printer, we use the option –p. The name of the printer immediately follows the options with no spaces. Its format is presented as below:



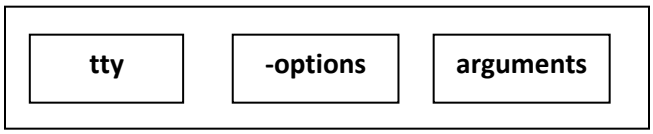
In the following example the first command prints one file to the standard printer, the second command prints three files to the standard printer, and the third command prints three files to printer lp0 (note the leading –P in the option)

```
$ lpr file1
$ lpr file1 file2file3
$ lpr –Plp0 file1 file2 file3
```


OTHER USEFUL COMMANDS:

Terminal (tty) command:

The tty utility is used to show the name of the terminal you are using. Later we will see that UNIX treats each terminal as a file, which means that the name of your terminal is actually the name of a file. The format of tty command is shown in the figure below:



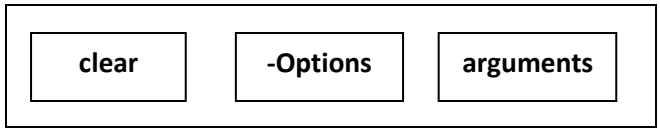
The name of the terminal file can be displayed with the tty command as in the following example:

```
$ tty
/dev/ttyq0
```

The output shows that the name of the terminal is /dev/ttyq0, or more simply, ttyq0. In UNIX the name of the terminal usually has the prefix tty.

Clear Screen (clear) command:

The clear command clears the screen and puts the cursor at the top. It is available in most systems. It's format is given as below:

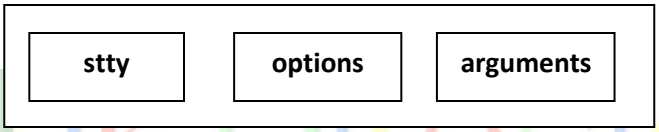


The following example demonstrates the clear command:

$ **clear**

Set Terminal (stty) command:

The set terminal (**stty**) command sets or unsets selected terminal input/output options. When the terminal is not responding properly, the set terminal command can be used to reconfigure it. Depending on the arguments, it has several uses. Its basic format is shown below:



Set Terminal without Option or Argument:

If we use the stty without any options or arguments, it shows the current common setting for your terminal. Some of these settings are communication settings, such as the baud rate. Others are control settings such as the Delete key setting (default ctrl-h). The basic command is demonstrated below:

```
$ stty
Speed 9600 baud; -parity hupcl clocal
line = 1; intr = ^A; erase = DEL; old-switch = ^@; dsusp = ^@;
brkint –inpck icrnl onlcr tab3
echo echoe echok echoke
```

Set Terminal with Options Only:

The set terminal command can be used with two options (-a and –g), neither of which allows arguments. With the –a option, it displays the current terminal option settings. With the –g option, it displays selected settings in a format that can be used as an argument to another set terminal command.

Set Terminal with Arguments:

Many of the terminal settings should be set only by a super user. Examples of these arguments strip input to seven-bit characters, force all uppercase characters to lowercase and echo characters as they are typed. One of the common user arguments sets values for the system editing control characters, such as delete.

Set Erase and Kill (ek): The ek argument sets the default erase (Delete key – ctrl+h) and kill (ctrl+c) to their defaults.

Set terminal to general configuration (sane): The sane argument sets the terminal configuration to a reasonable setting that can be used with a majority of the terminals.

$ **stty sane**

Set Erase Key (erase) by default, the Erase key is ctrl+h on the terminal. It deletes the previous character typed (in modern terminals, the Delete key also deletes the previous key). We can reconfigure the keyboard to use another key as the Delete key with the *erase* argument as shown in the following example:

$ **stty erase ^a**

The new key follows the keyword *erase* in the argument and defines the key-stroke that is to be used to erase characters. It should always be a control key + key combination, but UNIX will accept a single key.

Set Kill (kill) the Kill key deletes a whole line. By default it is ctrl+u. We can change it using the set terminal command with the *kill* argument as follows: $ **stty kill 9**

Set Interrupt Key (intr) the interrupt key interrupts or suspends a command. By default, it is ctrl + c. It can be reset using the *intr* argument as follows: $ **stty intr ^9**

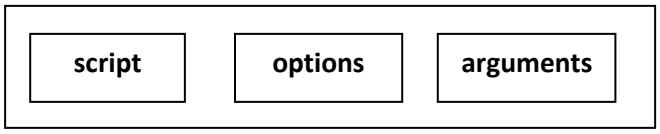
The following table summarizes the common control key commands that can be set.

Command	Attribute	Default
Erase text	*Erase*	^h
End of file key	*Eof*	^d
Kill command	*Kill*	^u
Interrupt command	*Intr*	^c
Start output	*start*	^q
Stop output		^d
Suspend command		^z

There are many more stty command options and arguments. Many of them are applicable only to the super users.

Record session (script) command:

The **script** command can be used to record an interactive session. When you want to start recording, key the command. To record a whole session, including the logout, make it the first command of the session. Its format is as shown below:



To stop recording, key **exit**. The session log is in the file named *typescript*. An example is shown below:

```
$ script

 Script started, file is typescript
$ date
Mon May 28 13:40:59 PDT 2001
$ who
forouzan     ttyq0   May 28 12:33  (153.18.171.128)
spk49772     ttyq3   May 28 11:27  (c296129-a.frmt1.sfba.home.com)
xf043637     ttyq4   May 28 11:25  (ACB46F15.ipt.aol.com)
$ ls -l
total 168
-rw-r--r--   1   forouzan staff    26 Apr 22   10:45  file.dat
-rw-r--r--   1   forouzan staff    49 May 14   15:42  notes.dat
$ exit
Script done, file is typescript
```

The typescript filename does not have to be used. We can give it any name by passing filename as an argument. Example:

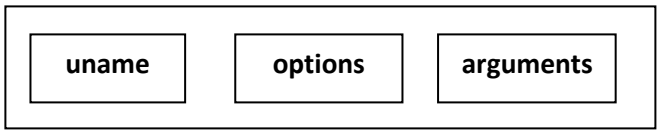
```
$ script myfilename
```

Each script command execution erases the old script file output. To append to the file rather than erase it, we use the append option (-a) as in the next example:

```
$ script -a
```

System Name (uname) command:

Each UNIX system stores data, such as its name, about itself. To see these data, we use the **uname** command. The command format appears as below:



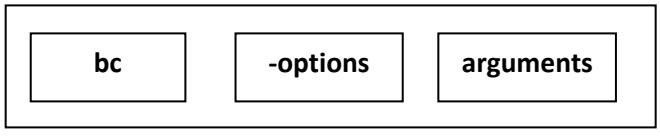
We can display all of the data using the all option (-a) or we can specify only the name (-n), operating system (default or –s), or software release (-r). There are other options that you can explore in the system documentation (**man**). Options can also be combined; for example to display the operating system and its release use –sr.

Example:

$ uname
IRIX64
$ uname –s
IRIX64
$ uname –r
6.5
$ uname –n
Challenger
$ uname –sr
IRIX64 6.5
$ uname –a
IRIX64 challenger 6.5 04191225 IP19

Calculator (bc) command:

The bc command turns UNIX in to a calculator. However, it is much more than just a calculator. In many respects, it is actually a language, similar to C, with a powerful math library ready at your fingertips. The command format is shown below:



To start the calculator, we simply key the bc command. To terminate it, we key end of the file (ctrl+d).

Simple arithmetic: It is done at the command line. It supports addition (+), subtraction (-), multiplication (*), division (/), modulus (%), and power (^).

These features are shown below:

```
$ bc
12+8
20
45 - 46
-11
34 + 34 * 3
136
34 + 34 / 3
45
8 % 3
2
24.5 ^ 67
53595873643724563625813104911439902299022950.4
```

Floating point calculations:

To use floating-point arithmetic, we must specify the number of decimal points to be used. This is done with scale expression, which sets the number of digits after the decimal in a floating-point number. Example:


```
19/3
6
scale = 2
 19/3
6.33
20/3
6.66
21/3
7.00
scale = 8
19/3
6.33333333
20/3
6.66666666
21/3
7.00000000
scale = 0
19/3
6
```

Arithmetic base calculations:

The bc calculator can be used in decimal, binary, octal, or hexadecimal bases. The base is specified by one of the two expressions: *ibase* (or simply *base*) or *obase*.

The ibase expression specifies that input will be in the specified base. The obase expression specifies the output base. If the input or output base is not defined, it is assumed to be decimal (base 10).

Example:

$ bc
ibase = 2
111
7
111*111
49
ibase=8
10
8
10*11
72
ibase = 16
1A
26
10*10
256

VI EDITOR - CONCEPTS:

As used in UNIX, editing includes both creating a new file and modifying an existing text file. An **editor** is a utility that facilitates the editing task – i.e., the creation and modification of text files. Because of their close association with text files, editors are often called text editors.

A text editor differs from a word processor in that it does not perform typographical formatting such as bolding, centering, and underlining. In other words, an editor is a basic text processor that is used to create and edit text quickly and efficiently. Editors come in two general types:

1. *line editors*
2. *screen editors*

LINE EDITORS:

In a line editor, changes are applied to a line or group of lines. To edit a line, the user must first select a line or group of lines for editing. The selection can be by line number, such as edit line 151, or through an expression that defines the line such as edit the line beginning with "Once".

Line editors are more useful when you want to make global changes over a group of lines. For example if you want to add an extra space at the beginning of each line, it is easier to use a line editor to change to change all lines at once rather than adding the space to individual lines.

However using a line editor is more complex than using a screen editor; it is necessary to know how to select the group of lines and then how to apply the change. *Two common UNIX line editors are **sed** and **ex**.*

SCREEN EDITORS:

A screen editor presents a whole screen of text at a time. The obvious major difference between line editor and screen editor is that with screen editors we can see each line of text in its context with other lines.

We can move the cursor around the screen and select a part of the text (a character or a word or a line). We can also scroll the screen – i.e., move the view of the data up or down to see different parts of the text document.

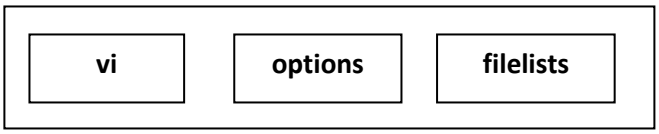
The current position in the text is shown by a cursor. The cursors form depends on the operating system. In some systems, it is a black square with white text. In others it is indicated by a blinking square or an underline. In our examples we show the cursor as a black square.

The current character is the character at the cursor. The current line is the line containing the cursor. As we search for the text the contents of the screen change. When we scroll up, the text moves down and new lines appear at the top of the screen; when we scroll down, new lines appear at the bottom of the screen.

As we enter the text, the text at the end of the line wraps to the next line, and the line at the bottom of the screen is pushed down below the screen and out of sight.

THE VI EDITOR: The vi (pronounced as vee-eye) editor is a screen editor available on most UNIX systems. When you invoke the vi editor, it copies the contents of a file to memory space known as buffer (a temporary version of the file).

Once the data have been loaded in to the buffer, the editor presents a screen full of buffer to the user for editing. If the file does not exist, an empty buffer is created. The format of vi editor is shown as below:



MODES:

COMMAND MODE:

When the vi editor is in the command mode, any key that is pressed by the user is considered as command. Commands are used to move the cursor, to delete or change part of the text, or to perform many other operations.

As soon as the command is entered, it is executed – the return key is not required. On some systems commands are known as hot keys. Of course the key ore key sequence must be a valid command. If it is not the result is unpredictable.

There are two aspects of vi editor that are frustrating to new users.

1. Most of the commands are not echoed on the screen.
2. Most commands should not be followed by a Return key. The single character or sequence of characters, pressed is the command itself.

TEXT MODE:

When the vi editor is the text mode, any key i.e. pressed by the user is considered text. The keyboard works as a typewriter. In the text mode, the characters typed by the user, if they are printable characters, are inserted in to the text at the cursor.

This means to add text in a document, we should first place the cursor at desired location. To place the cursor, however, we must be in the command mode. The typical operation is to place the cursor with a command, switch to the text mode and edits the text, then switch back to the command mode for the next operation.

CHANGING MODE:

It is clear that we must switch back and forth between vi command and text modes. To tell vi to do something, it must be in command mode; to edit text, it must in the text mode. The relationship between these two modes is summarized as follows:

- To invoke vi, you type the following command at the UNIX prompt: **$ vi filename**

- The file name is the name of the file that already exists or the name of the file that you want to create.
- When you invoke vi, you are always in command mode. During the session, you can move back and forth between the command mode and text mode.
- To exit vi, you must be in the command mode.
- There are six commands that take you to the text mode (a, A, i, I, o, and O) from the command mode.
- When you are in the text mode, you press the escape key (esc) to go to the command mode.

COMMANDS:

The vi editor is the interactive part of **vi** / **ex**. When initially entered, the text fills the buffer, and one screen is displayed. If the file is not large enough to fill the screen, the empty lines below the text on the screen will be identified with a tilde (~) at the beginning of each line. The last line on the file is a status line; the status line is also used to enter **ex** commands.

ADD COMMANDS

To insert text, you need to be in the text mode. The vi editor contains several commands to change the mode to text.

Insert mode (i and I)

We can insert text before the current position or at the beginning of the current line. The lower case insert command (i) changes to the text mode. We can then enter the text.

Command	Function
i	Inserts the text before the current character
I	Inserts text at the beginning of the current line
a	Appends the text after the current character
A	Appends the text at the end of the current line
o	Opens an empty text line for new text after the current line
O	Opens an empty text line for new text before the current line

The character at the cursor is pushed down the line as the new text is inserted. When we are through entering text, we return to the command mode by keying *esc*. The uppercase insert command (I) opens the beginning of the current line for inserting text.

Regardless of which command is used, once vi is in the insert mode, you can add as many characters or lines as needed.

Append Commands (a and A):

The basic concept behind all append commands is to add text after the specified location. In vi, we can append after the current character (a) or after the current line (A). As with the insert command, we need to escape to return the command mode.

New Line Commands (o and O):

The new line commands create an open line in the text file and insert the text provided with the command. Like insert and append after entering the text, we must use the escape key to return to the command mode.

The lowercase new line command opens a new line after the current line so you can add text here. To add text in a new line below the current line, use the lowercase new line (o) command.

If the current line spans multiple screen lines, the new line is opened after the screen line that contains the return character, not after the screen line that contains the cursor. To add the new text in a line above the current line, use the upper case new line (O) command.

CURSOR MOVE COMMANDS

To edit the text we need to move the cursors to the text to be edited. The cursor moves commands are effective only in the command mode. After the execution of a move command, the vi editor is still in the command mode. There are many cursor move commands; Among them we see eight commands here:

Command	Function
Horizontal Moves	
h, ⇐, Back space	Moves the cursor one character to the left.
l, ⇒, Spacebar	Moves the cursor one character to the right.
0	Moves the cursor to the beginning of the current line
$	Moves the cursor to the end of the current line.

Command	Function
Vertical Commands	
k, ⇑	Moves the cursor one line up
j, ⇓	Moves the cursor one line down
-	Moves the cursor to the beginning of the previous line
+, Return	Moves the cursor to the beginning of the next line

HORIZONTAL MOVE COMMANDS:

The horizontal move commands move the cursor one character to the left or right or to the beginning or end of the current line.

Move left (h, ⇐, Back space) Each of the three move left commands moves the cursor one character to the left. To move multiple characters, the key is repeated, once for each character position to be moved.

Move right (l, ⇒, Spacebar) Like the horizontal left moves, there are three horizontal right moves. Each command moves the cursor one character to the right.

Beginning of line (0) and End of the line ($):

To move to the beginning of the line containing the cursor, we use the beginning of the line command (0). Similarly, to move to the end of the current line, we use the end of the line command ($).

VERTICAL MOVE COMMANDS:

The vertical move command move the cursor up or down one line, which may be more than one screen line.

Move up (k, ⇑) The move up commands move the cursor up one line. If upward line has at least the same number of characters as the line we are moving from, the cursor is placed in the same relative position in the upward line. However if the upward line is shorter than the line we are moving from, the cursor is placed at the end of the upward line.

Move down command (j, ⇓) The move down command moves the cursor down one line. The cursor will be positioned in the lower line in the same manner as we saw for move up command. If the lower line is shorter, the cursor is placed at the end of the line.

If the lower line has the same number of the characters than the line we are moving from, the cursor is positioned at the same relative character in the lower line.

Up line (-) or Down line (+ or Return)

The up line command moves the cursor to the beginning of the line above the current line. If there is no line above the current line, the cursor does not move. Similarly, the down line command moves the cursor to the beginning of the next line. If there is no next line the cursor does not move.

DELETION COMMANDS:

Although there are several deletion commands, we discuss only two here: i.e. *x* and *dd.* There are defined in the table below:

Command	Function
x	Deletes the current character
dd	Deletes the current line

Delete character (x):

The delete character command deletes the current character - i.e. the character pointed to by the cursor. After the deletion, the cursor points to the character after the deleted character.

When the last character on the line is deleted, however the cursor moves to the character on the left. If the only character on a line is deleted, the cursor stays at the beginning of the line and further deletions have no effect.

Delete line (dd):

The delete line command deletes the current line. After the deletion, the cursor is at the beginning of the next line. If the last line is deleted, the cursor is at the beginning of the line above line above the deleted line, which is now the last line of the file.

Join command:

Two lines can be combined using the join command (J). The command can be used anywhere in the first line. After the two lines have been joined, the cursor will be at the end of the first line.

Scrolling commands:

The standard UNIX window is only 24 lines long. When editing a document longer than 24 lines, therefore, we need to scroll through the buffer to see the text. You can scroll up and your can scroll down. The direction of the scroll is relative to the text, not to the screen.

The text moves up and down. Scrolling down means moving down in the text towards the end of the file; as you move down, lines disappear at the top of the screen and new lines are added at the bottom.

Similarly, scrolling up means moving the text up – that is toward the beginning of the buffer. As you scroll up lines are added at the top of the screen, and the lines at the bottom disappear.

The following table shows six vi commands for scrolling up and down.

Command	Function
ctrl + y	Scrolls up one line.
ctrl + e	Scrolls down one line.
ctrl + u	Scrolls up half a screen (12 lines)
ctrl + d	Scrolls down half a screen (12 lines)
ctrl + b	Scrolls up whole screen (24 lines)
ctrl + f	Scrolls down whole screen (24 lines)

Line scroll commands (ctrl + y and ctrl + e):

There are two vi commands that scroll the text one line. The ctrl + y command scrolls the text up one line. When you scroll the text up, the cursor location does not change unless it is on the line at the bottom of the screen, in which case it must move up one line because the last line disappears.

To scroll down one line, use ctrl + e command. Again the cursor location does not move unless it is in the top of the screen, in which case it must move down one line because the top line disappears.

Half screen commands (ctrl + u and ctrl + d)

Half screen scrolls the cursor 12 lines up or down. This is true even if you have enlarged the screen size in your software because vi screen standard is 24 lines – half a screen is therefore 12 lines.

After the scroll, the cursor is at the beginning of the target line. The text on the screen also scrolls 12 lines unless it is already at the top or bottom, in which case only the cursor moves. If there are fewer than 12 lines to the top or the bottom, the cursor is places at the top or bottom line as appropriate. To scroll up half screen use ctrl + u command for up and ctrl +d command for down

Full page commands (ctrl + b and ctrl +f):

The full screen scrolls operate like the half screen scrolls. The exception is that they will not move the cursor to the beginning or the end of the document. If the top line is already visible on the screen and you scroll up a full screen, the cursor doesn't move.

To scroll up a full screen, use ctrl + b (b for beginning). To scroll down a full screen, use ctrl + f (f for finish).

Undo commands:

If after editing text you decide that the change was not what you really wanted, you can undo it. There are two undo commands. One reverses only the last change. Another reverses all changes to the current line. The undo commands are summarized below:

Commands	Functions
u	Undoes only the last edit.
U	Undoes all changes on the current line.

Saving and exit commands:

The save and exit command saves (writes) the contents of the buffer back into the file. The original file on the disk is changed and cannot be retrieved. After saving you are still in vi and can continue editing.

Command	Function
:w	Saves the contents of the buffer without quitting vi
:w filename	Writes the contents of buffer to new file and continues.
ZZ	Saves the contents of the buffer and exits.
:wq	Saves the contents of the buffer and exits.
:q	Exits the vi (if buffer changed will not exit).
:q!	Exits the vi without saving.

Save to New file (:w filename)

This command writes the contents of the buffer to a new file. The original file is unchanged. After the save you are still in vi, but you are editing the newly saved file.

Save and Quit (ZZ)

The save and quit command is very handy. It allows you to save your work and exit from vi. Note that ZZ command does not need the command line at the bottom of the screen; it is an immediate command and is therefore faster.

Write and Quit (:wq)

The write and quit command has the same effect as the ZZ command, but it requires that you open the command line at the bottom of the screen. Therefore you must first type a colon (:) and then w followed by q. This sequence is actually two commands: first a write command to write the buffer to the file and then a quit command to exit vi.

Quit (:q)

The quit command exits vi, but only if the buffer is unchanged. If the buffer has been changes, vi displays a message and returns to the command mode.

Quit and Don't save (:q!)

If you really want to discard the changes you have made, use the quit and don't save command. Note that you will not be warned that your changes are being discarded.

FILE SYSTEMS:

A file name can be any sequence of ASCII characters. However, we recommend that you not use some characters in a filename. For example the greater than (>) and less than (<) characters cannot be used in a file name because they are used for file redirections. On the other hand a period in UNIX does not have a special meaning as it does in other operating systems.

To make your names as meaningful as possible, we recommend that you use the following simple rules:

Start your names with an alphabetic character.

Use dividers to separate parts of the name. Good dividers are the underscore, period, and the hyphen.

Use an extension at the end of the filename, even though UNIX doesn't recognize extensions. Some applications, such as compilers, require file extensions. In addition, file extensions help you classify files so that they can be easily identified. A few common extensions are .txt for text files, .dat for data files, .bin for binary files and .c and .c++ for C and C++ files respectively.

Never start a filename with a period. Filenames that start with a period are hidden files in UNIX. Generally, hidden files are created and used by the system. They will not be seen when you list your files.

Wildcards:

Each filename must be unique. At the same time we often need to work with a group of files. For example we may want to copy or list all files belonging to a project. We can group files together using wildcards that identify portions of filenames that are different.

A **wildcard** is a token that specifies that one or more different characters can be used to satisfy a specific request. In other words, wildcards are like blanks that can be filled by any character.

There are three wildcards in UNIX: the single character (?) wildcard, the set ([…]) wildcard, and the multiple characters (*) wildcard.

Matching any single character:

The single character wildcard can be used more than once in a group. You need to understand, however, that each ? can match only one character. The following table shows the examples of single character wildcards.

File	Matches				Does Not Match	
c?	c1	c2	c3	ca	ac	cat
c?t	cat	cet	cit	c1t	cad	dac
c??t	caat	cabt	cact	c12t	cat	daat
?a?	bat	car	far	mar	bed	cur

Matching single character from a set

The single character wildcard provides powerful capabilities to group files. Sometimes it is too powerful and we need to narrow the grouping. In this case we use the character set, which specifies the characters that we want to match enclosed in brackets.

Like single character wildcard, a set matches only one character in the filename. We can, however, combine the set with fixed values, other sets, the single character wildcard, and the asterisk. Examples of wildcard set are in table below:

File	Matches				Does Not Match	
f[aoei]d	fad	fed	Fod	fid	fud	fab
f[a-d]t	fat	fbt	fct	fdt	fab	fet
f[A-z][0-9]	fA3	fa3	fr2	f^2	FA3	fa33

Matching zero or more characters:

The asterisk (*) is used to match zero or more characters in a filename. Whereas the single character wildcard matched one and only one character, the asterisk matches everything, and everything includes nothing.

File	Matches	Example	Does Not Match
*	Every file		
f*	Every file whose name begins with f	f5c2	afile, cat
*f	Every file whose name ends in f	staff	f1, faF
.	Every file whose name has a period	file.dat	bed, cur

Echo Command and File Wildcards:

One way to check the existence of files using the wildcards is the use of echo command. As we learned before (print message (echo) command), the echo command displays its argument. However, if the argument has a wildcard in it, the shell first expands the argument using the wildcard and then it displays it. It can be used with all wildcards.

You can display all three character filenames that start with f and end with t with the command as below:

$ **echo f?t**
f-t fit fat fbt fgt fwt

FILE TYPES:

UNIX provides seven file types which are given below:

1. Regular
2. Directory
3. Character special
4. Block special
5. Symbolic link
6. FIFO
7. Socket

Regular files: Regular files contain user data that need to be available for future processing. Sometimes called as ordinary files, regular files are the most common files found in a system.

Directory file: A directory is a file that contains the names and location of all files stored on a physical device.

Character Special file: A character special file represents a physical device such as a terminal that reads or writes one character at a time.

Block special file: A block special file represents a physical device such as disk, that reads or writes data a block at a time.

Symbolic link files: A symbolic link is a logical file that defines the location of another file somewhere else in the system.

FIFO files: A first-in, first-out also known as a named pipe, is a file that is used for inter process communication.

Socket: A socket is a special file that is used for network communication.

REGULAR FILES:

The most common file in UNIX is regular file. Regular files are divided by the physical format used to store the data as text or binary. The physical format is controlled by the application program or utility that processes it. UNIX views both formats as a collection of bytes and leaves the interpretation of the file format to the program that processes it.

TEXT FILES:

A text file is a file of characters drawn from the computers character set. UNIX computers use the ASCII character set. Because the UNIX shells treat data almost universally as strings of characters, the text file is the most common UNIX file.

BINARY FILES:

A binary file is a collection of data stored in the internal format of the computer. In general, there are two types of binary files: data files and program files. Data files contain application data. Program files contain instructions that make a program work. If you try to process binary file with a text-processing utility, the output will look very strange because it is not a format that can be read by people.

DIRECTORIES:

Like other operating systems, UNIX has provision for organizing files by grouping them in to directories. A directory performs the same function as a folder in a filing cabinet. It organizes related files and subdirectories in one place.

In most systems the directory hierarchy contains, at the top of the directory structure is a directory called the ***root***. Although its name is root, in the commands related to directories, it is typed as one slash (**/**). In turn each directory can contain subdirectories and files.

SPECIAL DIRECTORIES:

There are four special directories that play an important role in the directory structure. They are root directory, home directory, working directory and parent directory.

Root directory:

The root directory is the highest level in the hierarchy. It is the root of the whole file structure; therefore, it does not have a parent directory. In a UNIX environment, root directory always has several levels of subdirectories. The root directory belongs to the system administrator and can be changed by only the system administrator.

Home directory:

We use the home directory when we first log in to the system. It contains any files we create while in it and may contain personal system files such as our profile file and the command history. Our home directory is also the beginning of our personal directory structure. Each user has a home directory. The name of the home directory is the user login id or the user id.

Working directory:

The working or current directory is the one that we are in at any point in a session. When we start, the working directory is our home directory. If we have subdirectories, we will most likely move from our home directory to one or more subdirectories as needed during a session. When we change directory our working directory changes automatically.

Parent directory:

The parent directory is immediately above the working directory. When we are in our home directory, its parent is one of the system directories. When we move from our home directory to a subdirectory, our home directory becomes the parent directory.

PATHS AND PATHNAMES:

Every directory and file in the system must have a name. When we refer to files in a command, we use their filenames. To uniquely identify a file we need to specify the files path from the root directory to the file. The files path is specified by its absolute pathname, a list of all directories separated by a slash character (/).

The absolute pathname for a file or directory is like an address of a person. If you know only the person's name you cannot easily find that person. On the other hand if you know a person's name, street address, city, state and country, then you can locate anyone in the world.

ABSOLUTE PATHNAMES:

As we have stated that absolute pathname specifies the full path from the root to the desired directory or file. The following table lists the full pathname:

File	File Absolute Pathname	Directory Absolute Pathname
file1	/etc/file1	/etc
file2	/usr/staff/adams/file2	/usr/staff/adams
file1	/usr/staff/joan/file1	/usr/staff/joan
file1	/usr/staff/tran/file1	/usr/staff/tran
file1	/usr/staff/zeke/file1	/usr/staff/zeke

Because directories also have absolute pathnames, we also show the path to each of their directories. Absolute pathnames can be used all the time regardless of where we are in the directory hierarchy.

Disadvantage: They change as directories and their subdirectories are moved around a system by the system administrator.

RELATIVE PATHNAME:

Wherever we are working in UNIX, we are always in a directory. When we start, we are in our home directory, which is different from the root directory. Our current or working directory is the directory we are in. We can move from one directory to another any time we need by changing our working directory.

If we need to reference a file in our working directory, we can use a relative rather than the absolute path name. A **relative pathname** is the path from the working directory to the file. Therefore, when we refer to a file in the working directory, we simply use the filename with no path. This works because when a relative pathname is used. UNIX starts the search from the working directory.

Similarly from our working directory we can refer to a file in a subdirectory by using a relative path from the working directory to the file.

RELATIVE PATH NAME ABBREVIATIONS:

Home directory: For home directory the abbreviation is the tilde symbol (~). When we use the tilde, the shell uses the home directory pathname set for us by the system. When we need to refer to our own home directory, we can just use the tilde.

Working directory: The abbreviation for working directory is a dot (.). While it may seem strange that we need an abbreviation for the current directory, some UNIX commands require that the pathname for a start directory be specified even when it is the current directory. In these cases the dot abbreviation makes constructing the relative path easy.

Parent directory: The abbreviation for parent directory is tow dots (. .). The parent of any directory is the directory immediately above it in the directory path form the root. This means that every directory except the root directory has a parent.

RELATIVE PATH NAMES EXAMPLES:

Working Directory	Relative Path to file3
/etc	Use absolute pathname, Its faster.
/	usr/staff/joan/file3
/usr	staff/joan/file3
/usr/staff	joan/file3
/usr/staff/joan	file3
/usr/staff/adams	../joan/file3 or ~joan/file3
/usr/staff	../../joan/file3 or ~joan/file3

FILE SYSTEM IMPLEMENTATION:

When a disk is formatted, space is divided into several sections. Some sections contain structural information about the disk itself. The last section contains the physical files. Disk storage can be conceived of as a continuous linear storage structure, starting with track 0 on the first track surface and moving down through track 0 of all surfaces before continuing with track 1 on the first surface.

File systems:

In UNIX, a file system has four structural sections known as blocks: the boot block, the super block, the inode block, and the data block.

The boot block, super block, inode blocks are fixed at the beginning of the disk. They occupy the same locations on the disk even when the disk is reorganized. These blocks are shown in figure 1.3:

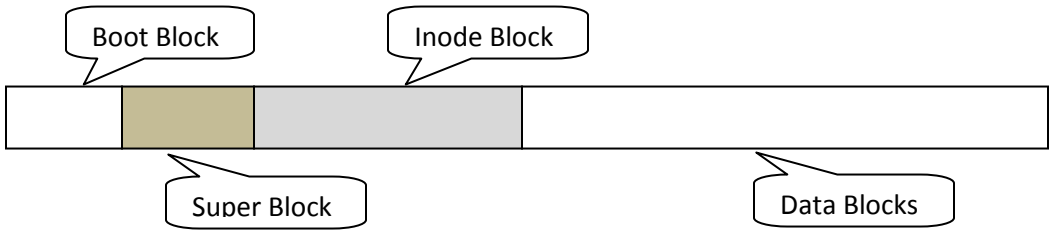


FIGURE 1.3: A DISK FILE FORMAT

BOOT BLOCK:

When an operating system is started, a small program known as the boot program is used to load the kernel in to the memory. The boot program, when present, is fond at the beginning of the disk in the boot block.

SUPER BLOCK:

The super block contains information about the file system. Stored here are such items as the total size of the disk, how many blocks are empty, and the location of bad blocks on the disk.

INODE BLOCK:

Following the super block is the inode (information node) block, which contains the information about the each file in the data block. The file information is stored in records known as inodes. There is one inode for each file on the disk. They contain information about the file, most notably the owner of the file, its file type, permissions, and address.

DATA BLOCKS:

The data block contains several types of files. First and the foremost from the user's point of view, it contains all of the user files; it is where data are stored. It also contains the special files that are related to user data: regular files, directory files, symbolic link files, and FIFO files. Finally it contains the character special, block special and socket system files.

DIRECTORY CONTENTS:

Given the concept of inodes pointing to files, the directory becomes a very simple structure. Remember that directory is itself a file. Its contents are a set of inode-file entries containing the filename and its corresponding inode. A directory is seen in the figure below:

FIGURE 1.4: A DIRECTORY OF THREE FILES

In the above figure each file is paired with an inode, which as we have seen contains the address of and other information about the file. This pairing of filename and inodes is the basis of a UNIX concept called links.

LINKS:

A link is a logical relationship between an inode and a file that relates the name of the file to its physical locations. UNIX defines two types of links: Hard Links and Symbolic Links.

HARD LINKS:

In a hard link structure, the inode in the directory links the filename directly to the physical file. While this may sound like an extra level of structure, it provides the basis for multiple file linking.

SYMBOLIC LINKS:

A symbolic or soft link is a structure in which the inode is related to the physical file through a special file known as a symbolic link. The symbolic link is not as efficient as the hard link. Soft links were designed for two specific situations: links to directories, which must be symbolic and links to files on other file systems.

MULTIPLE LINKS:

One of the advantages provided by the inode design is the ability to link two or more different filenames to one physical file. The filenames can be in the same directory or in different directories. This makes the multilink structure a convenient and efficient method of sharing files. Files are commonly shared among a team working on a large system.

Additionally, given a user with a large file system, it may be convenient to share a highly used file among several directories. In either case, the important point to note is that although there are several different references to the file, *it exists only once*.

When a file is created, an entry containing its name and inode link is stored in its directory. To share the file, the same entry can be created in another directory. As we will see, the filename in the second directory entry can use the same or a different filename. Both entries, however, link to the same inode, which in turn points to the physical file.

CURRENT AND PARENT DIRECTORIES:

We have two entries; Current directory is represented as a dot (**.**) and the parent directory is represented as double dot (**. .**). These entries provide the inode entry for these two directories so that we can reference them when necessary.

We used the option all entries (**-a**) and inode entries (**-i**). Because the current and parent directories start with a period, they are considered as hidden files and are not normally printed. Note that in this list, the current directory is listed first and the parent directory is listed second. These two directories are generally at the top of the list.

Example:

$ **ls –ai**

79944 .	79887 DirE	79937 file1
80925 . .	79942 backUpDir	79872 lnDir
79965 DirA	79906 backUpDir.mt	79871 mvDir

In the above example there are three columns each containing an inode and file name pair. Note that the inode for the current directory is 79944 and its parent directory is inode 80925.

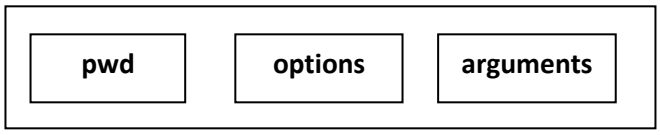
OPERATIONS UNIQUE TO DIRECTORIES:

A careful analysis of the UNIX directory and file operations reveals that some are used only with directories, some are used only with files, and some are used with both directories and files. The operations that are used only with directories are as follows:

Locate Directory (pwd) Command:

We can determine the location of the current directory in the directory structure. This is particularly used after we have navigated through the directory structure and we need to know where we are.

The command used to determine the current directory is print working directory (pwd). It has no options and no attributes. When executed, it prints the absolute path-name for the current directory. Its format is shown below:



The name print working directory is misleading because it does not actually print the working directory; it only displays it on the screen. This is one of those historical utilities. It was created when UNIX was run on teletypes.

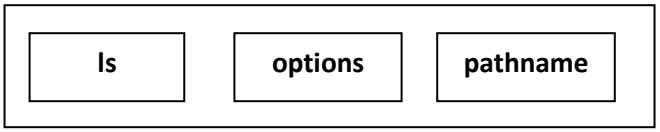
Example:

$ **pwd**

/usr/~gilberg/tran

List Directory (ls) command:

The list (**ls**) command lists the contents in a directory. Depending on the options used, it can list files, directories, or subdirectories. If the name of the directory is not provided, it displays the contents of the working directory. Its basic format is presented as follows:



Let's begin with the simplest list, a basic list of the directories and files under the working directory. This command uses no options or pathnames.

Example:

$ **ls**

BST.c	aFile	file1	memo509	saturn zFile
BST.h	binsrch.c	gnuFile	note311	statusRpt

The basic list command formats the directories and files alphabetically in columns.

Long list:

The simple list command is good for a quick review of the files in a directory, but much more information is available with the long list. Let's look carefully at the attributes shown below:

$ **ls -l**

total 2

-rw-r--r--	1	gilberg	staff	12 May 17 08:45 f-t
-rw-r--r--	1	gilberg	staff	12 May 17 08:45 f1t

-	→	File type
rw-r--r--	→	Permissions

1	→	Links
gilberg	→	Owner
staff	→	Group
12	→	File size
May 17 08:45	→	Last modified
f-t, f1t	→	Filename

File Type: There are seven file types in UNIX, which contains the file type designation, which is the first character of each line as shown in table below:

File Designation	File Type
-	Regular File
d	Directory
c	Character Special
b	Block Special
l	Symbolic Link
p	FIFO
s	Socket

Permissions: There are three sets of permissions: owner, group, and other. Each group has three possible permissions: read (r), write (w) and execute (x) in that order. If permission is granted the appropriate letter is shown. If it is not permitted, a dash is shown.

Links: Links is a count of the number of files that are linked to this directory or file.

Owner: Owner is the user name for that account that owns the file. it is usually the same as the login name.

Group: If the owner of the file is a member of a group, such as staff in above example, the group name is shown here.

File Size: The file size in bytes.

Last Modification: The date and time of last modification. If the file is more than a year old, the time is replaced by the year.

Filename: The name of the file. Normally, filename is just that, the name the owner used when the file was created. Sometimes, however, you will see some additional characters in a filename. Text editors often create a work file. When the editor is not terminated correctly, these working versions can be left hanging around.

Working files begin and/or end with a pound sign (#). Hidden files are identified with a leading period. Occasionally you will see a system file with a tilde(~) at the end of the name.

List Options:

List All Hidden files are normally not displayed in a file list. To display all files, including hidden files, use option **–a** as shown below:

$ **ls –a ~**

.	.mailrc	file2
. .	.profile	mail
.forward	.sh_history	mail.instr
.desktop-cis-b12	C-Programs	mail.staff.ids
.history	f-files	manVi

Working Directory:

The directory option (-d) displays only the working directory name. if used with the long list, it displays the working directory attributes.

Example:

$ **ls –d**

drwxr-xr-x 14 gilberg staff 3584 May 17 15:17

List User and Group ID:

The list user and group id option (-n) is the same as a long list except that the user and group id's are displayed rather than the user and group names.

Example:

$ **ls –nd**

drwxr-xr-x 2 3988 24 512 May 17 08:53

Reverse Order:

The reverse order option (-r) sorts the display in descending order (reverse order). This causes the file names starting with 'z' to be displayed before file names starting with 'a'.

Example:

$ **ls –r**

zFile	saturn	memo509	file1	aFile	BST.c
statusRpt	note311	gnuFile	binSrch.c	BST.h	

Time Sorts: There are three time sorts. They should all be used with the long list option. The basic time sort (-lt) sorts by the time stamp with the latest file first. The second time sequence is by last access (-lu). The third time sequence lists the files by the inode date change (-lc); this is basically the file creation date, although some other changes such as permission changes, affect it.

Example:

$ **ls -lt**

total 12

-rw-r- -r- -	1	gilberg	staff	14	May	19	13:33	file1
drwxr-xr- x	2	gilberg	staff	512	May	19	13:29	memos
-rw-r- -r- -	1	gilberg	staff	15	May	18	18:17	zFile
-rw-r- -r- -	1	gilberg	staff	15	May	18	18:16	aFile.tmp

Identify Directories:

We can identify the directories in a long list by the file type, which is the first character of each line. If the file type is d, the file is a directory. On a short list, however, there is no way to identify which files are directories and which are ordinary files. The –p option appends each directory name with a slash (/) as given below:

$ **ls –p**

BST.c	binSrch.c	memo509	saturn	zFile
BST.h	file1	memos/	statusRpt	
aFile.tmp	gnuFile	note311	unix7.1/	

Recursive List: It means reentering over and over again. In a list structure command, recursive means to list the directory and then list another directory, and then another one. Specifically, when we list a directory recursively, option –R, we want to list not only the contents of the directory itself but also the contents of all of its subdirectories. The recursive directory list is used primarily to study the structure of a directory. This needs to be done whenever we are considering reorganizing a directory.

Example:

$ **ls -Rp**

```
file1    file2    techNotes/    UNIX7.1/

./techNotes:
personal/        sort.doc        vi.doc

./techNotes/personal:
profile.doc
```

Print one column: There will be situations in which you want the filenames printed as a column rather than several files in one line (multicolumn). The print option for print one column is -1.

$ **ls -1**

```
BST.c
BST.h
aFile.tmp

zFile
```

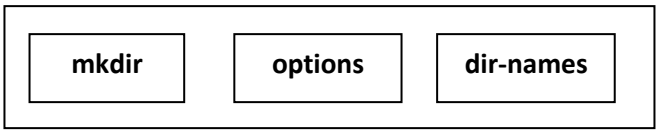
Print inode Number: Occasionally, you will need to see the inode numbers for a file. The print inode number option is –i. If you want to see all of the inodes for all files, see it for one or more files, you can list the filenames as given below:

$ **ls –li file1 zFile**

```
37585  -rw-r- - r- -    1    gilberg    staff    14    May 17 16:32 file1
37809  -rw-r- - r- -    1    gilberg    staff    15    May 18 18:17 zFile
```

Make Directory (mkdir) command:

To create a new directory, you can use the make directory command (mkdir). It has two options: permission mode and parent directories. Its format appears as below:



You can control the permission for the new directory with the mod (-m) option. If the mode is not specified, the directory will typically have a mode that includes read and execute for three sets (owner, group, others) and write only for the owner.

Example:

$ **mkdir saturnGp**

$ **ls –ld saturnGp**

drwxr-xr-x 2 gilberg staff 512 May 19 14:03 saturnGp

The second make directory option, parent (-p), creates a parent directory in the path specified by the directory name. For example, if we need to create a *memos* directory for the *saturn* project, and within memos we want to create a schedule directory we could create both with the options as shown below:

$ **mkdir –p saturnGp/memos/schedule**

$ **ls -1R saturnGp**

total 1

drwxrwxr-x 3 gilberg staff 512 May 19 14:17 memos

saturnGp/memos:
total 1

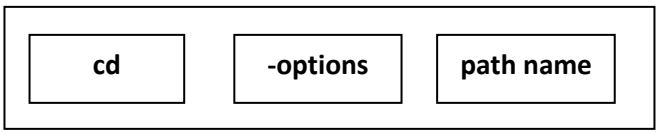
drwxrwxr-x 2 gilberg staff 512 May 19 14:17 schedule

saturnGp/memos/schedule:
total 0

Change Directory (cd) Command:

Now that we have multiple directories, we need some way to move among them – this is, to change our working directory. The change directory (cd) command does exactly that.

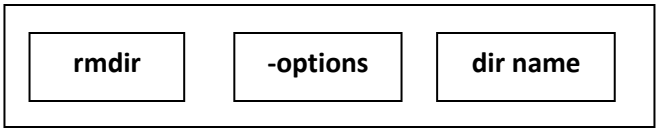
The change directory (cd) command format is given below:



There are no options for change directory command. The path name can be relative or absolute; generally it is relative. It there is no path name argument, the target is the home directory. The home directory can also be target by using the home abbreviation (cd ~).

Remove Directory (rmdir) Command:

When a directory is no longer needed, it should be removed. The remove directory (rmdir) command deletes directories. Its format is shown below:



The rmdir command cannot delete a directory unless it is empty. If it contains any files, UNIX will return an error message, "Directory not empty". When the command is executed successfully, the only response is the shell prompt.

OPERATIONS UNIQUE TO REGULAR FILES:

There are four operations that are unique to regular files; they are ***create file***, ***edit file***, ***display file***, and ***print file***. These utilities are described below:

CREATE FILE:

The most common tool to create a text file is a text editor such as ***vi***. The other utilities, such as ***cat***, that is useful to create small files.

Binary files are created by application programs written for a specific application and utilities such as c compiler.

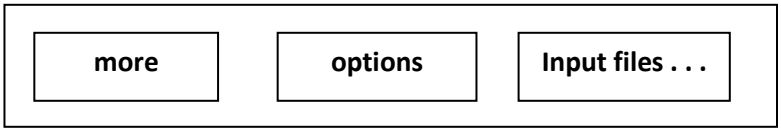
EDIT FILE:

UNIX provides several utilities to edit the text file. The most common is a basic text editor such as ***vi***. In addition, there are other utilities such as ***sed***, which provide powerful search and edit tools. All of the basic edit utilities can create a file, but only some can edit one.

DISPLAY FILE (more) COMMAND:

Although many utilities write their output to standard output (monitor), the most useful one to display a file is ***more***.

Display file command (more) allows us to set the output page size and, pauses at the end of each page to allow us to read the file. After each page, we may request one or more lines, a new page, or quit. The **more** command format is shown below:



The basic options for ***more*** command are given in the table below:

Option	Explanation
-c	Clears screen before displaying
-d	Displays error messages
-f	Does not screen wrap long lines
-l	Ignores formfeed characters
-r	Displays control characters in format ^C

Option	Explanation
-s	Squeezes multiple blank lines (leaving only on blank line in output)
-u	Suppresses text underlining
-w	Waits at end of output for user to entry any key to continue
-lines	Sets the number of lines in a screen (default is screen size – 2)
+nmbr	Starts output at the indicated line number (nmbr)
+/ptrn	Locates first occurrence of pattern (ptrn) and starts output two lines before it

The most basic use of the ***more*** used no options. In the basic form, ***more*** starts at the beginning of the file.

If the file is small (i.e. less than one screen's worth of lines), it prints all of the data and an "end" message. To return to command line, key ***enter***.

If there is more than one screen of data, ***more*** displays one screen, less two lines. At the bottom of the screen, it displays the message "***- - more - - (dd%)***". This message indicates that there are more lines in the file and how much has been displayed so far. The common more continue options are shown in the table below:

Option		Explanation	
Space	n+space	Displays next screen of output	Displays next screen of output and sets screen size to ***n*** lines.
Return	D	Advances one line	Displays half a screen
Nf	nb	Skips ***n*** screens and displays a screen	Moves back ***n*** screens and displays a screen
Q	Q	Quits **more**	Quits **more**
=		Displays current line number	
:f		Displays current file name and line number	
V		Transfers to vi editor at the current line	

PRINT FILE:

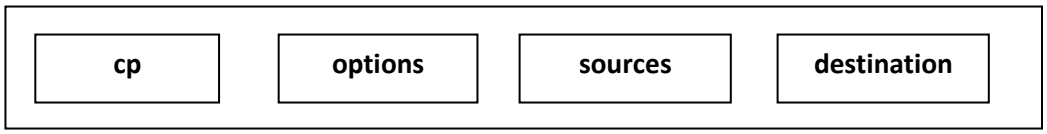
The most common print utility is line printer (lpr). The line printer utility prints the contents of the specified files to either the default printer or to a specified printer. Multiple files can be printed with the same command. If no file is specified, the input comes from the standard input, which is usually a keyboard unless it has been redirected.

OPERATIONS COMMON TO BOTH:

There are certain operations that are common to both regular files and directories; they are ***copy***, ***move***, ***rename***, ***link***, ***delete***, and ***find***. They are described as follows:

COPY (cp) COMMAND:

The copy (cp) utility creates a duplicate of a file, a set of files, or a directory. If the source is a file, the new file contains an exact copy of the data in the source file. if the source is a directory, all of the files in the directory are copied to the destination, which must be a directory. If the destination file already exists, its contents are replaced by the source file contents. The **cp** command copies both text and binary files. Its format is shown below:



The first argument lists one or more files, or one directory, to be copied. The destination identifies the filename for the new file or, when multiple files are being copied, the directory for the new files. Multiples files cannot be copied in to the same directory as the source files.

To copy a file successfully, several rules must be followed:

- The source must exist. Otherwise UNIX prints the message - **<source> - No such file or directory**
- If no destination path is specified, UNIX assumes the destination is the current directory.
- If the destination file does not exist, it is created; if it does exist, it is replaced.
- If the source is multiple files or a directory, the destination must be a directory.
- If the destination is the same directory as the source, the destination filename must be different.
- To prevent an automatic replacement of destination file, use the interactive (-i) option. When this option is specified, UNIX issues a warning message waits for reply.
- To preserve the modification times and file access permissions, use the preserve option (-p). In the absence of the preserve options, the time will be the time the file was copied, and file access permissions will be the defaults.

The copy command has three options:

Preserve Attributes Option: As stated in copy (cp) command section, when the destination file exists, its permissions, owner, and group are used rather than the source file attributes. We can force the permissions, owner and group to be changed, however by using the preserve (-p) option.

Interactive Option: We can guard against a file being accidentally deleted by a copy command by using interactive (-i) option. When the interactive option is specified, copy asks if we want to delete an existing file, if we reply ***y*** or ***yes*** the file is replaced. Otherwise if we reply ***n*** or ***no*** the copy is cancelled.

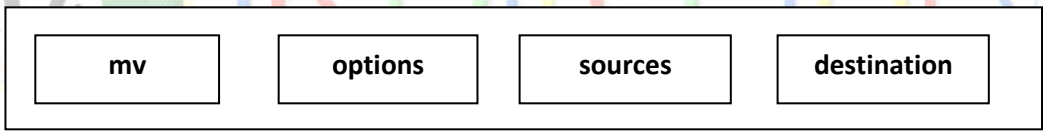
Recursive copy: Another way we can copy a collection of files is with the recursive (-r) copy. While the wildcard copy copies the matching files in a directory, the recursive copy copies the whole directory and all of its subdirectories to a new directory.

WILDCARD COPIES: Wildcards can be used to copy files as long as the destination is another directory. You cannot use wildcards if you are copying to and from the same directory.

MOVE (mv) COMMAND:

The move (mv) command is used to move either an individual file, a list of files, or a directory. After a move, the old filename is gone and the new file name is found at the destination.

This is the difference between a move and a copy command. After a copy, the file is physically duplicated; it exists in two places. The move format is shown below:



The first argument is the name of the file to be moved. The second argument is its destination or in the case of rename, its new name. Move has only two options: interactive (-i) and force (-f).

Interactive: If the destination file already exists, its contents are destroyed unless we use the interactive flag (-i) to request that **move** warn us. When the interactive flag is on, move asks if we want to destroy the existing file.

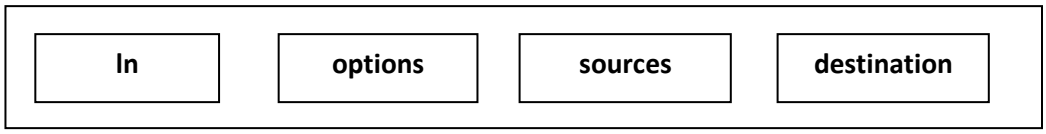
Force: When we are not allowed to write to a file, we are asked, if we want to destroy the file or not. If we are sure that we want to write it, even if it already exists, we can skip the interactive message with the force (-f) option.

RENAME (mv) COMMAND:

UNIX does not have a specific command to rename. Recall that the move (mv) command with a new name (second argument) renames the file. If the destination file is the same directory as the source file, the effect is a renaming of the file.

LINK (ln) COMMAND:

The link command receives either a file or directory as input and its output is an updated directory. The format of link command is given below:



Link has three options: symbolic, interactive and force.

Symbolic: The default link type is hard. To create a symbolic link, the symbolic option (-s) is used.

Interactive: If the destination file already exists, its contents are destroyed unless we request to be warned by using the interactive flag (-i). When the interactive flag is on, link asks if we want to destroy the existing file. This is similar to the message we get when the permissions don't allow us to write the file.

Force: When we are about to overwrite a file, we are asked if we want to destroy the file or not. If we are sure that we want to write it, even if it already exists, we can skip the interactive message with the force (-f) option.

HARD LINKS:

Hard Links to Files: To create a hard link to a file, we specify the source file and the destination file. If the destination file does not exist, it is created. If it exists, it is first removed and then re-created as a linked file.

Note: - *UNIX does not allow hard links to directories.*

Example:

```
$ ls lnDir
$
$ ln     file1    lnDir/linkedFile
$
$ ls     -i       file1
79914  file1
$ ls     -i       lnDir/linkedFile
79914  lnDir/linkedFile
```

SYMBOLIC LINKS:

When the link (ln) command is executed with no options, the result is a hard link. If we try to create a hard link to a different file system, however, it is rejected because hard links must be made within the current directory structure. To link to a different file system, therefore we must use symbolic links. We must also use symbolic links when we are linking to directories.

There is a danger with symbolic links because, although they behave like files and directories, they do not physically exist. They only point to the real directory or file. If the physical file is deleted, the file will no longer appear on a listing under its original name. It will still be available with symbolic name, but it is not accessible.

If a physical directory is deleted, the symbolic link to the directory still exists. If we try to list the symbolic directory, it lists with no files. If we try to move to it, however, we receive a message that it doesn't exist. Furthermore to delete it, we must use the delete file command (rm), not the delete directory command (rmdir).

Symbolic Link to Files: For all purposes a symbolic link functions exactly the same as a hard link, although not as efficiently.

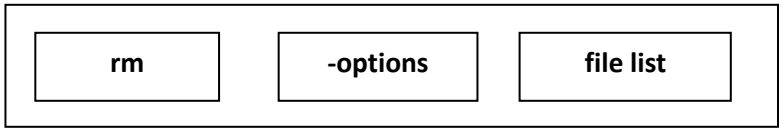
Example:

```
$ ls lnDir
linkedFile
$ ln -s file2 lnDir
$ ls -l file2
79937 file2
$ ls -il lnDir
total 2
79898 1rwxr-xr-x 1 gilberg ... 5 May 28 15:39 file2 -> file2
79914 -rwxr-xr-x 2 gilberg ... 120 May 25 15:10 linkedFile
```

Symbolic Link to Directory: The code to create a symbolic link to a directory is the same as the code to link files.

REMOVE (rm) COMMAND:

The remove (rm) utility deletes an entry from a directory by destroying its link to the file. Remember, however that there can be multiple links to a physical file. This means that a remove does not always physically delete a file. The file is deleted only if, after the remove, there are no more links to it. The remove command format is shown below:



To delete a file we must have write permission. If we try to remove a file that doesn't have it write flag set, UNIX asks for confirmation.

Remove options:

There are three options for the remove command: force, recursive, and interactive.

Force removal (-f) works just like the forced move. The file will be removed even it is write protected as long as we have write permission in the directory. If the directory is write protected, no files are removed and no warning messages are issued.

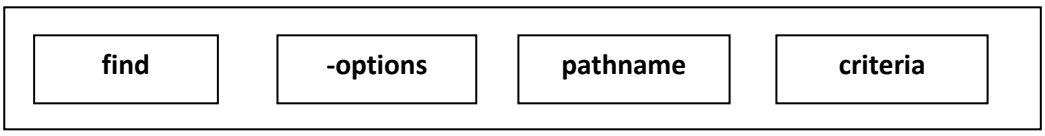
Recursive removal (-r) removes all files and empty directories in the path from the source directory. Files are deleted first, then the directory, so a directory can have files before the remove command. A directory is considered empty if all files are deleted. If a write protected file is found in the path, remove asks for confirmation before completing the remove. If the response is no, the file is not deleted, but the recursive remove command continues with other files and directories.

Wildcard remove:

Remove can be used with wildcards. This option is highly dangerous, however, and should be used with caution. It is generally considered a good idea to use the echo command to test the effect before actually executing a remove.

FIND FILE (find) COMMAND:

Given a large file environment, it can quickly become difficult to find a given file. It is not surprising , therefore that almost all operating systems provide a file search command. In UNIX, it is **find**. The format of find command is shown as follows:



Find has no options. Its first argument is the path that we want to search, usually from our home directory. The second argument is the criteria that find needs to complete its search.

Example:

$ **find DirC -name file3 -print**

DirC/DirC1/file3

List of find criteria is shown in the table below:

Criteria	Matches
-name file	Filename
-perm nnn	Permissions to nnn. Nnn must be an octal number
-perm –nnn	Permissions to bit mask, nnn. If bit mask contains 1, permission matches if is on.
-type c	File type. Valid file types are: block (b), character (c), directory (d), link (l), pipe (p), file(f), socket (s)
-link n	Number of links for a file
-group gname	Group name
-user name	User name. Numeric user id can also be used
-nouser	No name in the /etc/passwd file
-nogroup	No group name in the /etc/group file

EXECUTE ACTION:

The execute action (-exec) attribute invokes another UNIX command. Its format is shown below:

-exec command {} \;

The two braces in the action command are replaced by the path name of each file that matches the evaluation criteria. The command must end with a semicolon.

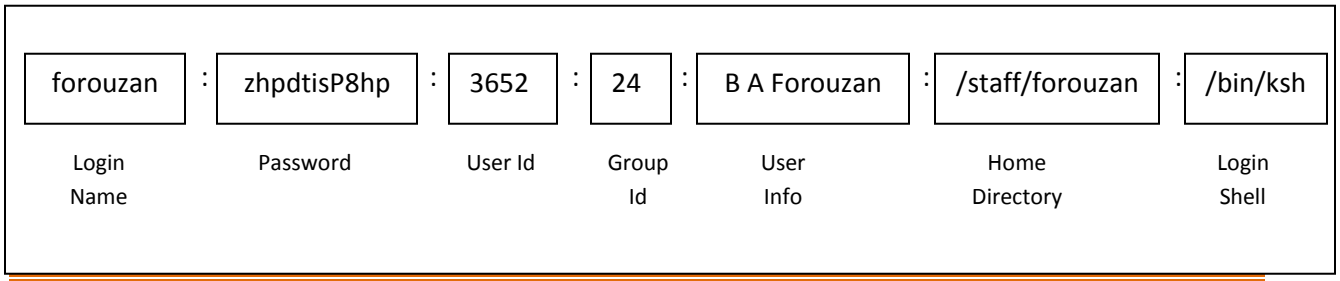
SECURITY LEVELS:

There are three levels of security in UNIX: system, directory, and file. The system security is controlled by, system administrator, a super user. The directory and file securities are controlled by the users who own them.

SYSTEM SECURITY:

System security controls who is allowed to access the system; It begins with your login id and password. When the system administrator opens an account for you, he or she creates an entry in the system password file. This file, named /etc/passwd, is located in the etc directory and contains several important pieces of information about you.

You can look at this file, but unless you are a super user, you can't, change it. The contents of an entry in our password file are shown below:



Let's examine the fields in turn:

- **Login Name:** the name you are known by to all of the other users in the system. It uniquely identifies you as one of the users in the system.
- **Password:** the one – way encrypted password that identifies you to the system.
- **User ID:** your internal user id. It is unique a number between 0 and 65,535. If you know the binary system, you will recognize 65,535 as the maximum unsigned number that can be represented in 16 bits. It also means that one UNIX system can have maximum of only 65,535 users.
    - User ID zero is reserved for the super user. Although there may be several super users in the system, there is only one super user id, zero.
- **Group ID:** similarly, group id is a unique number between 100 and 65,535 that identifies users who have common access.
- **User Information:** is used to store data about the user. Traditionally it is the user's given name. Another common use is an accounting number for systems that need to bill usage back to a user.
- **Home Directory:** the login or home directory when you first log in to the system. It is represented as the absolute pathname for your home directory.
- **Login Shell:** identifies the shell that is loaded when you login. It is also an absolute pathname.

PERMISSION CODES:

Both the directory and file security levels use a set of permission codes to determine who can access and manipulate the directory or file. The permission codes are divided in to three set of codes. The first set contains the permissions of the owner of the directory or file. The second set contains the group permissions for members in a group as identified by the group id. The third set contains the permissions for everyone else – that is the general public.

The code for each set is a triplet representing read (r), write (w) and execute (x). Read indicates that a person in that category may read a file or directory. Likewise write permission indicates that the user can change the file or directory. The last permission execute has different meanings for directories and files. For a file it indicates that the file is a program or a script that can be executed. When it is a directory, an execute permission allows access to the directory.

Permission	read (r)	write (w)	execute (x)
Directory	Read contents of directory	Add or delete entries (files) in directory using commands	Reference or move to directory
File level	Read or copy files in directory	Change or delete files	Run executable files

DIRECTORY LEVEL PERMISSIONS:

READ PERMISSION: When users have read permission for a directory, they can read the directory, which contains the names of the files and subdirectories and all of their attributes. They can then display the names and attributes with the list command. As a general rule, everyone is given read permissions for directories.

WRITE PERMISSION: When users have write permission, they can add or delete entries in a directory. This means that they can copy a file from another directory, move a file to or from the directory, remove (delete) a file. Obviously this is much more dangerous level of permissions. If you grant others permission to write to your directory, they can change its contents.

For security reasons, therefore, you generally don't grant others write permission to your directories. On the other hand, if you are maintaining a group directory within your account, it is reasonable to give group members write permission.

EXECUTE PERMISSION: Execute permission, sometimes called search permission, at the directory level allow you to reference a directory, as in a pathname or file read, or move to a directory using cd command.

To reference in any way a subdirectory or file under a directory, you must have executed permissions to all directories in the absolute pathname of that subdirectory or file. The user permissions for directories, therefore, generally include both read and execute. To grant read without execute permission is a contradiction because without execute no one, including the owner, may access the directory for any reason.

Example:

$ **ls –R permissionTest**
file1 ptSubDir

permissionTest/ptSubDir:
file2

If permission changed to rw- (no x) for user then:

$ **ls –R permissionTest**
Cannot access permissionTest/file1: Permission denied
Cannot access permissionTest/ptSubDir: Permission denied

$ **more permissionTest/file1**
Cannot open permissionTest/file1: Permission denied

FILE LEVEL PERMISSIONS:

File permissions are similar to directory permissions, except that they pertain to a file rather than a directory.

READ PERMISSION: users who have file **read permission** can read or copy a file. Files that contain public information generally have read permission. Private files, however, should be read only the user (owner). Of course, group files should be readable by anyone in the group.

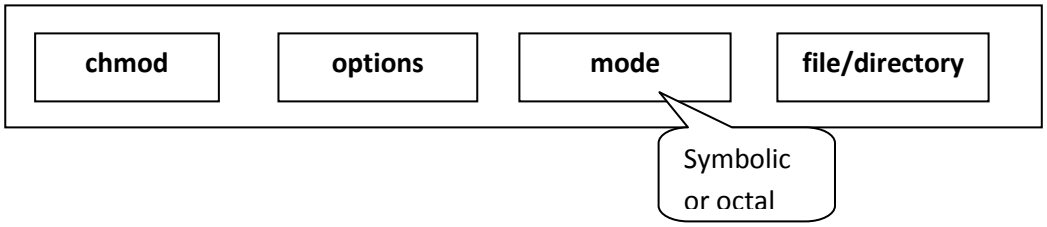
WRITE PERMISSION: Files with **write permission** can be changed. They can also be deleted. As with directories, you generally restrict write permissions to yourself (user) and other users in your group.

EXECUTE PERMISSION: With files, **execute permission** means that you can execute (run) programs, utilities, and scripts. Whether or not you grant execute permissions to others depends on what the program does.

CHECKING PERMISSIONS: To check the permissions of a file or directory, we use the long list command (ls –l).

CHANGING PERMISSIONS:

When a directory or a file is created, the system automatically assigns default permissions. The owner of the directory or file can change them. To change the permissions we use the **chmod** command, whose format is as given below:



SYMBOLIC CODES: When we use symbolic codes, we tell UNIX what permissions we want to set and it does all of the work for us. As we know that there are three sets of permissions: user, group, and other, each set use its first letter as a mnemonic identifier.

Thus, **u** represents user, **g** represents group and **o** represents others. If we want to set all three groups at the same time, we use a set of **a** for all. No groups also defaults to all, but it is better to use **a** when we want all. There are three sets of operators.

To assign absolute permissions to a set, we use the assignment operator(=). In this case current permissions for the set are replaced by the new permissions. To change only one or two of the permissions in a set and leave the others as they are currently set, we use a plus sign (+) to add permissions. To remove one or two permissions and leave the others alone, we use a minus sign (-).

Finally the permissions are represented by their symbolic letter: ***r*** represents read, ***w*** represents write and ***x*** represents execute. One, two or three symbolic letters can be used in each command. Each set of symbolic codes must be separated by commas, and there can be no spaces.

Example:

chmod u=rwx,g+w,o-w memo.doc

Command	**Interpreter**
chmod u=rwx file	Sets read (r), write (w), execute (x) permissions for user.
chmod g=rx file	Sets only read(r) and execute (x) for group; write (w) denied.
chmod g+x file	Adds execute (x) permission for group; read and write unchanged.
chmod a+r file	Adds read (r) to all users; write and execute unchanged.
chmod o-w file	Removes others write (w) permission; read and execute unchanged.

Octal codes:

A faster way to enter permissions is to use the octal equivalent of the codes. You must realize, however, that when using the octal codes, all the permission codes are changed. It is not like the symbolic modes where you need to specify only what you want to change. With octal codes, you must completely represent all of the user codes each time.

In an octal digit, there are three bit positions. The three different permissions for each set of codes correspond to the three different bit positions in an octal digit. The fist bit represents the read permission, the second bit represents the write permission and the third bit represents the execute permission. This relationship of octal bits positions to the permissions is seen in figure 1.5

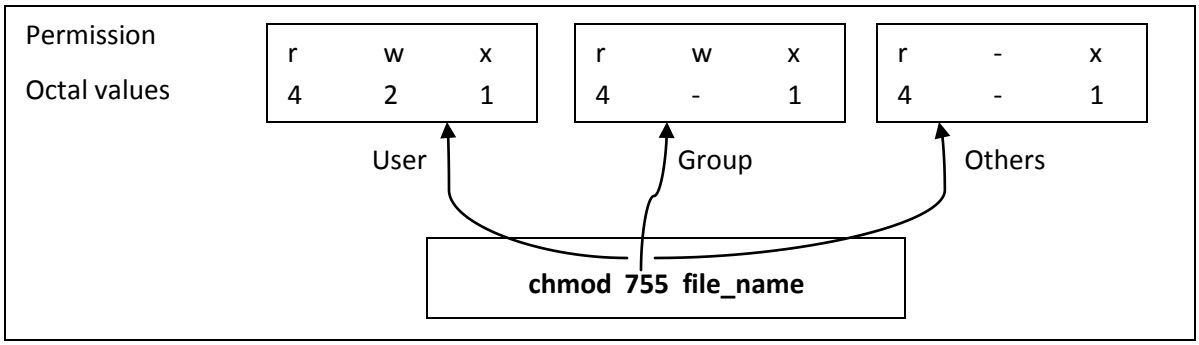


FIGURE 1.5: OCTAL chmod COMMANDS

There is one major difference between the symbolic mode commands and the octal commands. The octal commands set every permission. We cannot set just one or two leaving the others unchanged, as we can with the symbolic **+** operator. When you use octal commands therefore you should first check the settings with a long list command **ls** to determine the current settings.

The following table shows some of the more common permission commands.

Command	Description
chmod 777 file	All permissions on for all three settings
chmod 754 directory	User all, group read + execute; others read only
chmod 664 file	User and group read + write, others read only
chmod 644 file	User read + write, group and others read only
chmod 711 program	User all , group and others execute only

Options:

There is only one option, recursion (-R). The chmod recursion works just as in other commands. Starting with current working directory, it changes the permissions of all files and directories in the directory. it then moves to the subdirectories and recursively changes all of their permissions.

USER MASKS (TOPIC BEYOND SYLLABUS):

The permissions are initially set for a directory or file using a three digit octal system variable, the user mask (**mask**). Defined initially by the system administrator when your account is created, the mask contains the octal settings for the permissions that are to be *removed* from the default when a directory or a file is created. You can change the settings by creating a *mask* entry in your login file.

When a new directory or file is created, the number in the mask is used to set the default permissions. The default permissions are 777 for a directory and 666 for a file. The following table shows how the mask is used to create the default permissions.

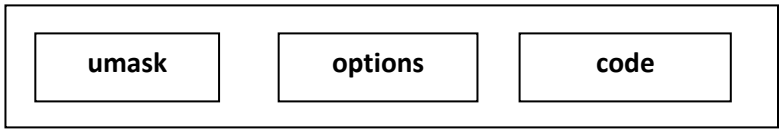
mask	Directory permission (default 777)	File permission (default 666)
0	7 (read/write/execute)	6 (read/write)
1	6 (read/write)	6 (read/write)
2	5 (read/execute)	4 (read)
3	4 (read)	4 (read)
4	3 (write/execute)	2 (write)
5	2 (write)	2 (write)
6	1 (execute)	0 (none)
7	0 (none)	0 (none)

Each mask digit means remove the corresponding digits from the default permission. For example mask digit 1 means remove 1 from directory and file permission. For directories the default permission is 7 (4 + 2 + 1), so we remove the 1 and the result is 6 (4 + 2). The default permission for a file, on other hand is 6 (4 + 2) so there is no 1 to be removed. The result is therefore also 6 (4 + 2). The following table shows some of the more common settings.

mask	Directory permission (default 777)	File permission (default 666)
000 (Public)	777 (rwx rwx rwx)	666 (rw- rw- rw-)
011 (Public)	777 (rwx rw- rw-)	666 (rw- rw- rw-)
022 (Write Protected)	755 (rwx r-x r-x)	644 (rw- r- - r- -)
007 (Project Private)	770 (rwx rwx - - -)	660 (rw- rw- - - -)
077 (Private)	700 (rwx - - - - - -)	600 (rw- - - - - - -)

USER MASK (umask) COMMAND:

The user mask displayed and set with the umask command as shown in the figure below:



To display the current user mask setting, use the **umask** command with no arguments. To set it, use the command with the new mask setting.

Example:

$ **umask**
000
$ **umask 022**
022

CHANGING OWNERSHIP AND GROUP:

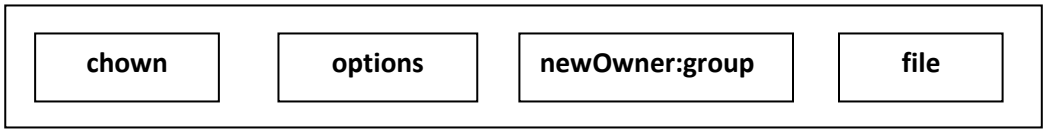
Every directory and file has an owner and a group. When you create a directory or a file, you are the owner and your group is the group. There are two commands that allow the owner and group to be changed. *The change ownership (**chown**) command can change the owner or the owner and group. The change group (**chgrp**) command can change only the group.*

CHANGE OWNERSHIP (chown) COMMAND:

The owner and optionally the group are changed with the change ownership (chown) command. The new owner may be a login name or a user id (UID). The group is optional.

When group is used, it is separated from the owner by a colon or a period. The group may be a group name or group id (GID), and the new owner must be a member of the group. The group does not have to be changed when the owner is changed unless the new owner is not a member of the current group.

Only the current owner or super user may change the ownership or group. This means that once the ownership is changed, the original owner cannot claim it back. Either the new owner or the system administrator must change it back. The format of ownership command is given as follows:



When the recursive option (-R) is used with a directory, all files in the directory and all subdirectories and their files are changed recursively. Note that after the owner is changed, it no longer displays in the directory list.

Example:

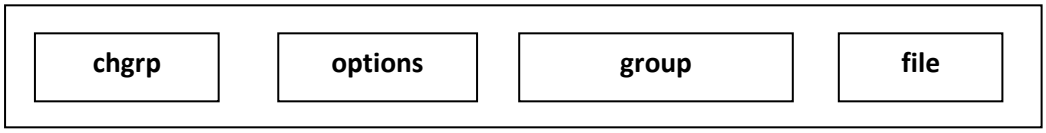
$ **ls -l**
total 2

-rw-r- -r- -	1	rfg3988	staff	120	Aug	30	2002	file1
-rw-r- -r- -	1	rfg3988	staff	120	Aug	30	2002	file2

$ **chown forouzan file1**
$ **ls -l**
total 1
-rw-r- -r- - 1 rfg3988 staff 120 Aug 30 2002 file2

CHANGE GROUP (chgrp) COMMAND:

To change the group without changing the owner, you use the change group (chgrp) command. This command format is shown below:



This command operates the same as the change owner command.