SECTION 1 INTRODUCTION TO UNIX

Structure		Page Nos.
1.0	Introduction	5
1.1	Objectives	5
1.2	Overview of Unix	5
1.3	Unix Commands	8
1.4	Summary	18
1.5	Further Readings	18

1.0 INTRODUCTION

This section is intended to introduce you to the Unix operating system. It will provide you with a basic understanding of the Unix operating system, its file and directory structure. We have also explained the architecture and components of Unix, further the section contains different useful Unix commands, and those are explained in detail with the help of examples.

1.1 OBJECTIVES

After completing this lab manual, you should be able to:

- get basic understanding of the Unix operating system;
- understand file and directory structure of Unix;
- know the basic Unix commands; and
- know how to get Unix help.

1.2 OVERVIEW OF UNIX

Even after thirty-five years of its creation Unix is still regarded as one of the most versatile, flexible and powerful operating systems in the computer world. Before you start swimming in your Unix shell, you must find out why people still regard it as powerful. As you may know, it was created at Bell Labs in 1970, written in the C Programming Language, which was developed at the same time. It supports a large number of simultaneous users, runs with few alterations on many hardware platforms (provides some platform independence), and of course it was and is, a simple, elegant, and easy to use (at least compared to its predecessors) operating system. In the early 1980s, the two strands of Unix development – AT&T and Berkeley – continued in parallel. The Berkeley strand got a major boost from Sun Microsystems, which used the Berkeley code as the basis for its Sun OS operating system.

This section is only a brief introduction to Unix operating system and does not include information on how to use all of its capabilities. Let's see the historical development of Unix at a glance:

- Kenneth Thompson, Dennis Ritchie, and others at AT&T Bell Labs developed first Unix version in 1969-1970.
- The above system was rewritten in the programming language C in 1972-1973.
- The seventh version (V7) of Unix was released in 1979.

During these 35 years of its development, different flavors of Unix system have evolved. Some of these Unix flavors are given on following *Table 1* with the name of the organization which participated in that development.

Table 1: List of Unix Flavors

Unix flavor	Organization name
AIX	IBM
FreeBSD	FreeBSD Group
HP-UX	Hewlett-Packard Company
Irix	Silicon Graphics, Inc.
Linux	Several groups
MacOS X Server	Apple Computer, Inc.
NetBSD	NetBSD Group
OpenBSD	OpenBSD Group
OpenLinux	Caldera Systems, Inc.
Red Hat Linux	Red Hat Software, Inc.
Reliant Unix	Siemens AG
SCO Unix	The Santa Cruz Operation Inc.
Solaris	Sun Microsystems
SuSE	S.u.S.E., Inc.

Due to, Unix is multi-user and multi-tasking environment, its flexibility and portability, facilities like electronic mail and the numerous programming, text processing and scientific utilities available, Unix became popular amongst the scientific and academic communities.

Basic Unix Elements

The six basic elements of Unix are:

1) Commands
2) Files
3) Directories
4) Environment
5) Processes
6) Jobs

1) **Commands** are the instructions you give to the system to inform it what it is to do.
2) **Files** are collections of data. A file is similar to a container in which you can store documents or raw facts and figures, which are stored in directories.
3) **Directory** is similar to a file basket that contains many files. A directory can also contain other directories.
4) **Environment** is a collection of different items that explain or modify how your computing session will be carried out.
5) **Process** is a command or application running on a computer.

6) **Job** is the sequence of instructions given to a computer from the time to initiate a particular task. A job may have one or more processes in it.

In our next discussion we will use and explore these elements in detail but first you need to understand different components of the Unix operating system.

Unix Components

The Unix operating system is made up of three components–the kernel, the shell and the programs (applications). As we have earlier defined the program, so here we will only define the kernel and the shell.

The Kernel

The kernel is the core of the Unix operating system. Its main responsibilities are to allocate time and memory to programs, file management and communications in response to system calls.

The Shell

The interface between the user and the kernel is called the shell. When a user logs in, the login program checks the username and password, and then starts another program called the shell. The shell is a command line interpreter (CLI). It interprets the commands the user types in and arranges for them to be carried out. The commands are themselves programs: when they terminate, the shell gives the user another prompt (% on our systems). Shell and kernel work together, for example a user types a command, which has the effect of removing some file. The shell searches the file store for the file containing the command, and then requests the kernel, through system calls, to execute a command on file. When the process has finished running, the shell then returns the Unix prompt to the user, indicating that it is waiting for further commands.

Unix System Architecture

Broadly, the Unix architecture is divided into three layers–Application Program, System Calls and Kernel. As we can see in *Figure 1*, the layers of Unix operating system are:

<table>
<tr><th colspan="3">Application Programs Created by Users</th></tr>
<tr><td colspan="3">Shells editor and commands (who, we, grep, comp)
Compilers and system libraries</td></tr>
<tr><th colspan="3">System call interface to the kernel</th></tr>
<tr><td>Signals
terminal handling
character I/O
system terminal
drivers</td><td>File system
swapping
block I/O system
disk & tape
drivers</td><td>CPU scheduling
pages replacement
demand paging
visual memory</td></tr>
<tr><th colspan="3">Kernel interface to Hardware</th></tr>
<tr><td>Terminal
controllers
terminals</td><td>Device controllers
disks and
tapes</td><td>Memory
Controllers
Physical memory</td></tr>
</table>

Figure 1: Unix System Architecture

Everything below the system call interface and above the physical hardware is the Kernel. The Kernel provides the file system, CPU Scheduling, memory management and other operating system functions through system calls. Programs such as shell (Sh) and editors (vi) shown in the top layer interact with the Kernel by invoking a

well-defined set of system calls. The system calls instruct the Kernel to do various operations for the calling programs and exchange data between the Kernel and the program. System calls for Unix can be roughly grouped into three categories: file manipulation, process control and information manipulation. Another category can be considered for device manipulation, but since devices in Unix are treated as (special) files, the same system calls support both files and devices.

Files and Processes

Everything in Unix is either a file or a process. In Unix, Process is identified by a unique number called process identifier (PID). They are created by users using text editors, running compilers etc. and must have a file name according to Unix rules. Following are the few examples of files in Unix.

- A document.
- A program written in some high-level programming language.
- A collection of binary digits
- A directory.

The Directory Structure

Let's see how files are grouped together in the directory structure in Unix. The files are arranged in a hierarchical structure, like an inverted tree as shown in *Figure 2.* The top of the hierarchy is called the root.

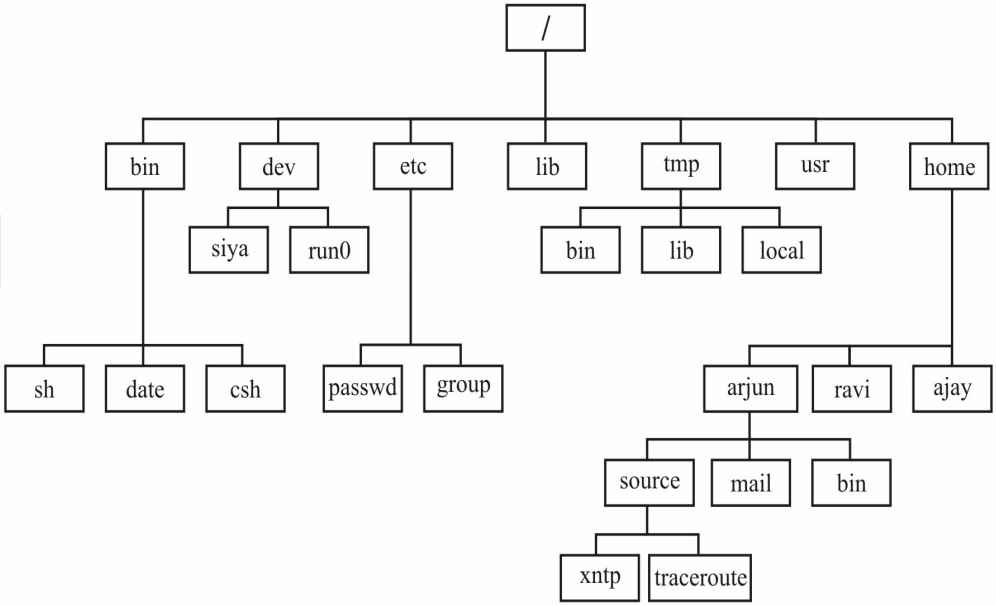


Figure 2: Directory structure of Unix

1.3 UNIX COMMANDS

When you start using Unix first you interact with a command interpreter program called the shell. Generally, Unix has two different shells C and Bourne shell. The shell you will find here in this manual is the *C shell.* The C shell command prompt ends with the percent sign (%). Another shell is the Bourne shell, which ends with a dollar sign ($).

Unix shells have their commands, syntax and control constructs. You can use these commands, syntax and control constructs to make your processing more efficient, or to perform repetitive tasks. In addition to this, you can store shell commands in a file (which is called a shell script) and run it as a simple program.

Commands

Using different commands in Unix you can manipulate files, data and environment. In this section we have explained the general syntax of Unix commands so that you can start working on it. A Unix command line consists of the name of the command followed by its arguments and ends with a RETURN. The general syntax for a command is:

Command Name [-flag options] ***filename*** **or** ***expression***

You should follow these rules with Unix commands:

1) Commands are case-sensitive.
2) Most Commands are in lowercase.
3) Commands must be entered at the shell prompt.
4) Command lines must end with a RETURN.
5) Options generally begin with a "–" (minus sign).

List

When you first log in to the Unix environment, the directory in which you enter is your home directory. The simple **ls** (list) command lists the contents of your current working directory. There may be no files visible in your home directory, in which case, the Unix prompt will be returned. List command does not show all the files in your home directory; it shows only those files whose name does not start with a dot (.). What are these dot files? Files beginning with a dot (.) are known as hidden files and generally contain important information.

? To list all files in your home directory including those whose names begin with a dot.

% ls –a

ls is an example of a command which can take options: –a is a case of an option. The options change the performance of the command. There are online manual pages (man) that inform you which options a particular command can take, and how each option modifies the performance of the command. Later on, we will discuss man pages in detail. Let see a few options of list command:

ls -a lists all the contents of the current directory, including files with initial periods, which are not usually listed.

ls -l lists the contents of the current directory in long format, including file permissions, size, and date information.

ls -s lists contents and file size in kilobytes of the current directory.

Working with directories

As you know directories are similar to files in Unix, in this section you will learn how to create or change the directories. Also, we will know about different special directories in Unix.

Make directory

Let's make a subdirectory in your home directory to hold the files you will be using and creating in this course. Use the following comments:

? To make a subdirectory called netprogs in your current working directory.

% mkdir netprogs.

? To see the directory netprogs which just you have just created

% ls.

Change directory

The command **cd** directory means change the current working directory to 'directory'. The current working directory may be thought of as the directory you are working in:

? To change to the directory you have just made, type

% cd netprogs.

Special directories

As you can see there are two special directories called (.) and (..) in the netprogs directory and in all other directories in your file system.

(.) means the current directory, so typing **% cd.** will take you to current directory which is netprogms at this moment. But you should remember that there is a space between cd and the dot. Using (.) as the name of the current directory will save a lot of typing time.

(..) means the parent of the current directory, so typing **% cd ..** will take you one directory up the hierarchy (back to your home directory). Remember typing cd with no argument always returns you to your home directory. This is very useful if you are lost in the file system.

Print working directory

Pathnames enable you to work out where you are in relation to the whole file-system. For example,

? To find out the absolute pathname of your home directory, type *cd* to get back to your home directory and then type

% pwd

The full pathname will look something like this - /a/fservb/fservb/fservb22/ gd/ hd which means that hd (your home directory) is in the directory gd (the group directory), which is located on the fservb file-server.

Understanding pathnames

What is a pathname? To understand the actual meaning of pathnames, let's takes an example where you want to see the contents of directory backups but currently you are in some other directory. Normally you can write the command as given below:

? To list the contents of your backups directory,

% ls backups

After this command you will get a message like this - *backups: No such file or directory*.

Because a backup is not your current working directory. To use a command on a file (or directory) not in the current working directory (the directory you are currently in), you must either *cd* to the correct directory, or specify its full pathname.

? To list the contents of your backups directory,

% ls netprogs/backups

Home directory

Home directories can also be referred to by the tilde ~ character. It can be used to specify paths starting at your home directory. So typing **% ls ~/netprogs** will list the contents of your netprogs directory, no matter where you are currently in the file system.

Copying Files

cp file1 file2 is the command which makes a copy of file1 in the current working directory and calls it file2 .

? Take a file stored in an open access area of the file system, and use the cp command to copy it to your netprogs directory.

🔑 % cd ~/netprogs
% cp /vol/examples/tutorial/science.txt .

Remember the dot (.) at the end of *cp* command. The dot means the current directory in Unix. The given command means copy the file *science.txt* to the current directory and keeping the same name. Assume directory /vol/examples/tutorial/ is an area to which everyone in the class has read and copy access.

Moving/Renaming files

mv file1 file2 command is used to move a file from one place to another. This has the effect of moving rather than copying the file, so you end up with only one file rather than two. It can also be used to rename a file, by moving the file to the same directory, but giving it a different name. Assume you have a science.bak file

? To move the file science.bak to your backup directory

🔑 inside the netprogs directory write % mv science.bak backups/.

Removing files and directories

To remove a file from a directory, use the rm command. For example, we are going

? To create a copy of the science.txt file then delete that file.

🔑 //when you are in netprogs directory, write//
% cp science.txt tempfile.txt
% ls (to check if it has created the file)
% rm tempfile.txt
% ls (to check if it has deleted the file)

You can use the rmdir command to remove an empty directory.

Displaying File on Screen

Clear

? To clear the terminal window of the previous commands

🔑 % clear

This will clear all text and leave you with the % prompt at the top of the screen.

cat (concatenate)

? The command cat can be used to display the contents of a file on the screen.

🔑 % cat science.txt

Less

? To display the contents of a file onto the screen a page at a time.

🔑 % less science.txt

Press the [space-bar] if you want to see another page, type [q] if you want to quit reading.

Head

? To display the first ten lines of a file to the screen.

🔑 % head science.txt

Tail

? To display the last ten lines of a file on the screen

% tail science.txt

Searching contents using less

You can search through a text file for a keyword (pattern) using the less command. For example,

? To search through science.txt for the word 'science'

% less science.txt

/science

less finds and highlights the keyword. To search the next occurrence of the word type [n].

Searching contents using grep

grep command searches files for specified words or patterns.

? To search through science.txt for the word 'science'

% grep science science.txt

After this command you will see each line containing the word science. Always remember that the grep command is case sensitive which means it distinguishes between Science and science. If you want to ignore upper/lower case distinctions, you should use -i option like % grep -i science science.txt. To search for a pattern, you must enclose it in single quotes (the apostrophe symbol). For example,

? To search for *computer science* in science.txt

% grep -i 'computer science' science.txt

Some of the other options of grep are given below:

Options of grep	Meaning
-v	Display those lines that do NOT match
-n	Precede each matching line with the line number
-c	Print only the total count of matched lines

Word count

We counts lines, words and characters in the named files, or in the standard input if no name appears. A word is a maximal string of characters delimited by spaces, tabs or newlines. If the optional argument is present, just the specified counts (lines, words or characters) are selected by the letters l, w, or c. For example,

? To count the words in science.txt

% wc -w science.txt

? To find out the lines in a file

% wc -l science.txt

Redirecting input and output

Most commands write to the standard output and many take their input from the standard input. When we run the cat command without specifying a file to read, it reads from the standard input (the keyboard), and on receiving the EOF (end of file) (^D), copies it to the standard output (the screen). We have option to redirect both the input and the output of commands. Let see how:

Output Redirection

We use the > symbol to redirect the output of a command. For example, first create two files list1 and list2. One contains six fruit names; the other contains four fruit names.

? To join (concatenate) list1 and list2 into a new file called biglist

% cat list1 list2 > biglist

Here the system is reading the contents of list1 and list2 and, then the output is redirected to the file biglist.

Input Redirection

We use the < symbol to redirect the input of a command. The input will be accepted from a file rather than the keyboard. For example,

? To sort the list of fruits

% sort < biglist

The redirect input and output can be done together as given below in an example:

? To output the sorted list to a file

% sort < biglist > slist

Who

Who command without any with it argument, lists the login name, terminal name, and login time for each current Unix user.

? To see who is on the system with you, type

% who

Pipes

Pipes (indicated by vertical bar |) create an inter-process channel between the commands. Let's see how, if you want

? To get a sorted list of user names you can write:

% who > names.txt

% sort < names.txt

But in this case we need a temporary file called names.txt. If you want to connect the output of the *who* command directly to the input of the *sort* command. This is called as inter-process channel between two commands. The symbol for a pipe is the vertical bar |

? To get a sorted list of user names

% who | sort

? To find out how many users are logged on

% who | wc -l

Wildcards

The characters * and ? are known as wildcards. Character * will match against none or more character(s) in a file (or directory) name. For example,

? To list all files in the current directory starting with list-----

% ls list*

The character? will match exactly one character. So ls? onkey will match files like donkey and monkey.

Help Manuals

There are on-line manuals, which give information about most commands. The manual pages tell you which options a particular command can take, and how each option modifies the performance of the command. man command locates and prints the section of this manual named title in the specified chapter.

We have different options given with man command. Some of these are given below:

Man command options	Meaning
– t	Phototypeset the section using troff(1).
– n	Print the section on the standard output using nroff(1).
– k	Display the output on a Tektronix 4014 terminal using troff(1) and tc(1).
– e	Appended or prefixed to any of the above causes the manual section to be

For example,

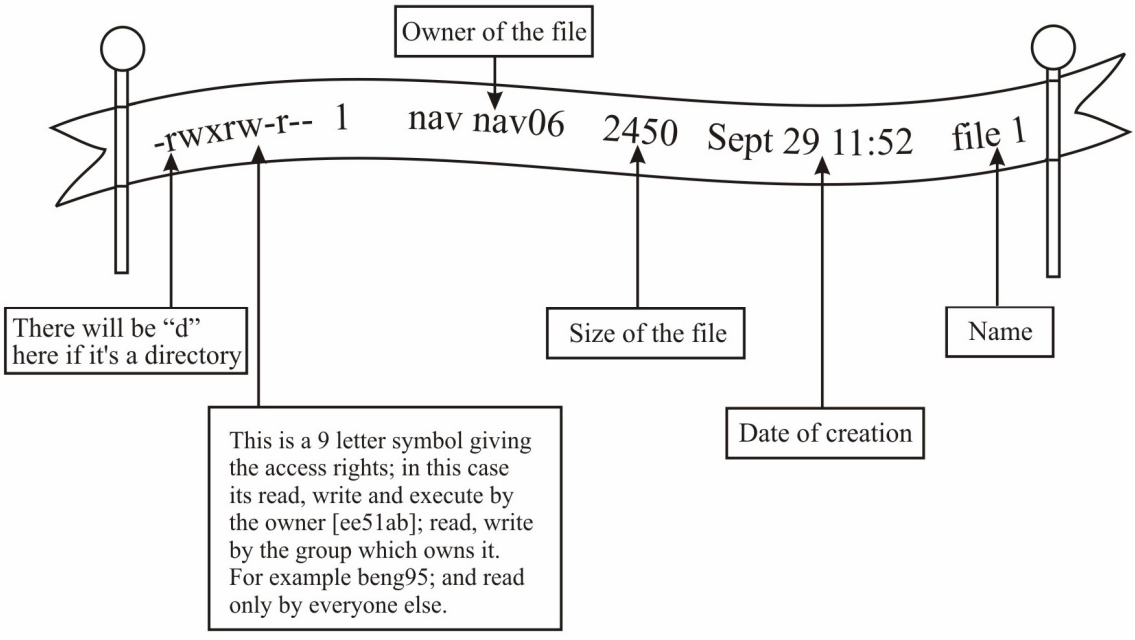
? To find out more about the wc (word count) command

% man wc

File access rights

When you give ls -l command you get lots of details about the contents of your directory, similar to the example given below:

-rwxrw-r-- 1 nav nav06 2450 Sept29 11:52 file1
-rwxrw-r-- 1 ee51ab beng95 2450 Sept29 11:52 file1



In the left-hand column is a 10 symbol string consisting of the symbols d, r, w, x, -, and, occasionally, s or S. If d is present, it will be at the left hand end of the string, and indicates a directory: otherwise - will be the starting symbol of the string. The 9 remaining symbols indicate the permissions, or access rights, and are taken as three groups of 3 each.

- The left group of 3 gives the file permissions for the user that owns the file (or directory)

- The middle group gives the permissions for the group of people to whom the file (or directory) belongs.
- The rightmost group gives the permissions for all others.

The symbols r, w, etc., have slightly different meanings depending on whether they refer to a simple file or to a directory. Access rights for files are given below:

- r (or -), indicates read permission (or otherwise), that is, the presence or absence of permission to read and copy the file.
- w (or -), indicates write permission (or otherwise), that is, the permission (or otherwise) to change a file.
- x (or -), indicates execution permission (or otherwise), that is, the permission to execute a file, where appropriate.

Access rights for directories

- r allows users to list files in the directory;
- w means that users may delete files from the directory or move files into it;
- x means the right to access files in the directory for execution.

Let us take a few examples to understand the access rights:

-rwxrwxrwx a file that everyone can read, write and execute.

-rw------- a file that only the owner can read and write - no-one else can read.

chmod

To change the access right of a file use chmod command that will change the mode of the file. Only the owner of a file can use chmod to change the permissions of a file. The options of chmod are as follows:

Symbol	Meaning
u	user
g	group
o	other
a	all
r	read
w	write (and delete)
x	execute (and access directory)
+	add permission
-	take away permission

For example,

? To remove, read, write, and execute permissions on the file biglist for the group and others

% chmod go-rwx biglist.

? To give read and write permissions on the file biglist to all.

% chmod a+rw biglist.

We can make some shortcut options for changing the mode. It can be constructed by ORing together some combination of numbers as given below:

Combination	Meaning
04000	set user ID on execution
02000	set group ID on execution
01000	save text image after execution
00400	read by owner
00200	write by owner
00100	execute (search on directory) by owner
00070	read, write, execute (search) by group
00007	read, write, execute (search) by others

Command for Processes and Jobs

A process is an executing program identified by a unique PID (process identifier).

? To see information about your processes with their associated PID and status,

% ps

A process may be in the foreground, in the background, or be suspended. In general, the shell does not return the Unix prompt until the current process has finished executing.

Some processes take a long time to run and hold up the terminal. Using the facility of multitasking the Unix prompt is returned immediately when a long process is sent to the background, and other tasks can be carried out while the original process continues executing. To make a process to run in the background, use and symbol at the end of the command line. For example, the command sleep command waits a given number of seconds before continuing, that is why % sleep 100 will hang up you for100 seconds. To utilize the time properly you can use & and send sleep in background.

? To run sleep in the background, type

% sleep 10 &

The & runs the job in the background and returns the prompt. When sleep command will finish it will return the job number and PID (process ID). Background execution is useful for jobs which will take a long time to complete. You can also suspend the process running in the foreground by holding down the [control] key and typing [z] (written as ^Z).

When a process is running, backgrounded or suspended, it will be entered onto a list along with a job number. To examine this list, type % jobs an example of a job list could be as given below with the job number in [] brackets:

[11] Suspended sleep 100

[12] Running netscape

[33] Running nedit

To restart a suspended processes write % fg %jobnumber. For example,

? To restart sleep 100

% fg %11

It is sometimes necessary to kill a process (for example, when an executing program is in an infinite loop) .To kill a job running in the foreground, type ^C (control c). To kill a suspended or background process, type % kill %jobnumber. Otherwise,

processes can be killed by finding their process numbers (PIDs) and using kill PID_number, If a process refuses to be killed, uses the -9 option, i.e., write

% kill -9 PID_number

Miscellaneous

quota

To check your current quota of amount of disk space on the file system and how much of it you have used

% quota -v

Disk free (df)

The df command reports on the space left on the file system. For example,

To find out how much space is left on the fileserver, type

% df .

Disk Utilized (du)

The du command outputs the number of kilobytes used by each subdirectory. Useful if you have gone over you quota and you want:

To find out which directory has the most files. In your home-directory,

% du

Compress

This reduces the size of a file, thus freeing valuable disk space. For example, type % ls -l science.txt and note the size of the file. Then to compress science.txt, type % compress science.txt. This will compress the file and place it in a file called science.txt.Z. To see the change in size, type ls -l again. To uncompress the file, use the uncompress command. % uncompress science.txt.Z

gzip

This also compresses a file, and is more efficient than compress. For example,

To zip science.txt, type

% gzip science.txt

This will zip the file and place it in a file called science.txt.gz

To unzip the file, use the gunzip command.

% gunzip science.txt.gz

File

File classifies the named files according to the type of data they contain, for example, ASCII (text), pictures, compressed data, etc.

To report on all files in your home directory, type

% file *

history

The C shell keeps an ordered list of all the commands that you have entered. Each command is given a number according to the order it was entered. % history command shows the command history list. If you are using the C shell, you can use the exclamation character (!) to recall commands easily.

Command	Meaning
% !!	Recall last command
% !-3	Recall third most recent command
% !5	Recall 5th command in list
% !grep	Recall last command starting with grep

1.4 SUMMARY

In this section, we studied that Unix that is know as one of the most versatile, flexible and powerful operating systems in the computer world, Unix was created at Bell Labs in 1970 written in the C programming language. We studied that the command, files and directories are the basic components of Unix. We have also studied the Unix architecture, which is divided into three layers Application Program, System calls and Kernel. This section also covered the detailed description of Unix command those are used to manipulate files, data and environment. The next section of this manual covers the basic introduction to the C language.

1.5 FURTHER READINGS

1) http://www.programmersheaven.com

2) http://www.ee.surrey.ac.uk/Teaching/Unix/unixintro.html

3) http://rcsg-gsir.imsb-dsgi.nrc-cnrc.gc.ca/documents/basic/node105.html.

4) Sumitabha Das, *Unix Concepts and applications* Tata Mcgraw Hill, 2003

5) www.rice.edu/Computer/Documents/Unix/unix1.pdf

SECTION 2 INTRODUCTION TO LINUX

Structure		Page Nos.
2.0	Introduction	19
2.1	Objectives	20
2.2	Linux Operating System	20
2.3	Exploring Desktop	22
2.4	Using the Shell	24
2.5	Understanding Users and File Systems	24
2.6	Linux Utilities and Basic Commands	27
2.7	Understanding Text Processing	32
2.8	Managing Processes	34
2.9	Summary	35
2.10	References/Further Readings	35

2.0 INTRODUCTION

Linux is a free open-source operating system based on UNIX. Linus Torvalds originally created Linux with the assistance of developers from around the world. Exactingly speaking, Linux is a kernel. A kernel provides access to the computer hardware and controls access to resources. The kernel decides who will use a resource, for how long and when. You can download the Linux kernel from the official web site.

However, the Linux kernel itself is useless unless you get all the applications such as text editors, email clients, browsers, office applications, etc. Therefore, someone came up with idea of a Linux distribution. A typical Linux distribution includes:

- Linux kernel
- GNU application utilities such as text editors, browsers
- GUI (X windows)
- Office application software
- Software development tools and compilers
- Thousands of ready to use application software packages
- Linux Installation programs/scripts
- Linux post installation management tools daily work such as adding users, installing applications, etc

Linux is renowned for being both powerful and also secure. As a possible open source method Linux may be hard-wired and also enhancing by many people, several computer programmers. It's authorized Linux to be able to “evolve” extra time, developing with each development from the developer or even improvement organization. Whilst Linux began like a programmer’s just operating-system it's becoming increasingly well-liked in the customer degree. Linux can also be able to handle powerful equipment without flinching, and that's why it's a preferred amongst supercomputers and also machines.

In this section, we are going to explore the practical process related to the loading and installing of Proprietary and Open Source operating system, so far as the Proprietary operating system is concerned we are going to discuss for Windows XP and on the end of Open source operating system we are going to work with Fedora. Before beginning the actual technicalities, Let us discuss some fundamentals of operating system.

2.1 OBJECTIVES

After going through this section, you will be able to:

- list the features of Linux;
- know the components of Linux Desktop and the Shell;
- understand Linux users and file systems
- know the Utilities and Basic Commands of Linux; and
- understanding text processing and process management issues

2.2 LINUX OPERATING SYSTEM

Linux is a freely available, open source, Unix-like operating system. Written originally for the PC by Linus Torvalds, with the help of many other developers across the Internet, Linux now runs on multiple hardware platforms. Because of its speed, stability, and low cost, Linux became the fastest growing operating system for servers. Today, Linux is widely used for both basic home and office uses. It is the main operating system used for high performance business and in web servers. Linux has made a huge impact in this world.

Red Hat Linux

The first public release of Red Hat Linux (version 1.0) is dated 1994, after that there are many versions of Red Hat Linux 1.1, 2.0, 2.1, 3.0.3, 4.0, 4.1, 4.2, 5.0, 5.1, 5.2, 6.0, 6.1, 6.2, 7, 7.1, 7.2, 7.3, 8.0 and 9 over a period till the year 2003. After that Redhat and Fedora project were merged.

Fedora

Fedora is a Linux-based operating system that showcases the latest in free and open source software. Fedora is always free for anyone to use, modify, and distribute. It is built by people across the globe who work together as a community: the Fedora Project. The Fedora Project is open and anyone is welcome to join.

Fedora Core 1 was the first version of Fedora and was released in the year 2003. Since, then different versions of Fedora such as Fedora Core 2, 3, 4, 5 and 6 were released till 2006. Then new versions such as Fedora 7, 8, 9 and 10 are also released till 2008.

2.2.1 Features of Linux Operating System

The following are various features of Linux operating system:

Low Cost

There is no need to spend time and huge amount money to obtain licenses since Linux and much of its software come with the GNU General Public License. There is no need to worry about any software that you use in Linux.

Stability

Linux has high stability compared with other operating systems. There is no need to reboot the Linux system to maintain performance levels rarely. Its freezes up or slow down. It has a continuous up-times of hundreds of days or more.

Performance

Linux provides high performance on various networks. It has the ability to handle large numbers of users simultaneously.

Networking

Linux provides a strong support for network functionality; client and server systems can be easily set up on any computer running Linux. It can perform tasks like network backup faster than other operating systems.

Flexibility

Linux is very flexible. Linux can be used for high performance server applications, desktop applications, and embedded systems. You can install only the needed components for a particular use. You can also restrict the use of specific computers.

Compatibility

It runs all common Unix software packages and can process all common file formats.

Fast and Easy Installation

Linux distributions come with user-friendly installation.

Better use of Hard Disk

Linux uses its resources well enough even when the hard disk is almost full.

Multitasking

Linux is a multitasking operating system. It can handle many things at the same time.

Open Source

Linux is an Open source operating systems. You can easily get the source code for Linux and edit it to develop your personal operating system.

Powerful Kernel

The kernel is heart of the Linux operating system. It manages the resources of Linux. Resources include:

- File management
- Multitasking
- Memory management
- I/O management
- Process management
- Device management
- Networking support including IPv4 and IPv6

- Advanced features such as virtual memory, shared libraries, demand loading, shared copy-on-write executables etc

The kernel decides who will use these resources and for how long and when. It runs your programs or sets up to execute binary files. The kernel acts as an intermediary between the computer hardware and various applications.

2.3 EXPLORING DESKTOP

A **desktop** essentially refers to a graphical user interface (GUI) and applications which permit the user of the OS to interact with the system (the OS and applications) graphically using familiar keyboard and mouse commands. Unlike in the Windows world, where you essentially end up with the Windows desktop when you install Windows, in the Linux world there are different desktops that one can install and use. Historically, the two most popular were GNOME20 (pronounced either "guh nome" or simply "nome") and KDE21. GNOME is part of the GNU project and is based on the GTK+ graphical user interface toolkit/library; it was first released in 1996, and since it is now 17 years old, it is a quite mature environment. GNOME 2 was widely supported by all the major Linux distros, but with the release of GNOME 3, fan support has dropped, as the familiar desktop metaphor has been replaced by something called the Gnome Shell22—which many former GNOME users are rebelling against, for various reasons23. Nonetheless, many popular distros ship with GNOME 3 including Fedora, Mint 12, openSUSE 12.1, Arch, OpenBSD, and Debian. KDE is similar in functionality to GNOME and was first released a couple of years later in 1998. It is based on a different GUI toolkit called Qt (pronounced "cute")The desktop of Linux is termed GNOME. GNOME stands for GNU Network Object Model Environment; and GNU, as you probably know, stands for GNU's Not Unix. GNOME is a GUI and a programming environment. From the user's perspective, GNOME is like the Motif-based Common Desktop Environment (CDE) or Microsoft Windows. Behind the scenes, GNOME has many features that enable programmers to write graphical applications that work together well. The next few sections provide a quick overview of GNOME, highlighting the key features of the GNOME GUI. You can explore the details on your own.

Taking Stock of GNOME

Although what you see of GNOME is the GUI, there is much more to GNOME than the GUI. To help you appreciate it better, note the following points and key features of GNOME:

- GNOME runs on several UNIX systems, including Linux, BSD (FreeBSD, NetBSD, and OpenBSD), Solaris, HP-UX, AIX, and Silicon Graphics IRIX.

- GNOME uses the Gimp Tool Kit (GTK+) as the graphics toolkit for all graphical applications. GTK+ relies on X for output and supports multiple programming languages, including C, C++, Perl, and others. Users can easily change the look and feel of all GTK+ applications running on a system. GTK+ is licensed under the GNU Library General Public License (LGPL). See Chapter 23 for more information about GPL and LGPL.

- GNOME also uses Imlib, another library that supports displaying images on an X display. Imlib supports many different image formats, including XPM and PNG (Portable Network Graphics).

- GNOME uses the Object Management Group's Common Object Request Broker Architecture (CORBA) Version 2.2 to enable GNOME software components to communicate with one another regardless of where the components are located or what programming language is used to implement the components.

- GNOME applications support drag-and-drop operations.
- GNOME application developers write documentation using DocBook, which is a Document Type Definition (DTD) based on Standard General Markup Language (SGML). This means that you can view the manual for a GNOME application on a Web browser, such as Mozilla.
- GNOME supports standard internationalization and localization methods. This means that you can easily configure a GNOME application for a new native language.

Exploring GNOME

Assuming that you have enabled a graphical login screen during Red Hat Linux installation, you should get the GNOME GUI whenever you log in to your Linux system. The exact appearance of the GNOME display depends on the current session. The session is nothing more than the set of applications (including a window manager) and the state of these applications. The metacity window manager, which is the default window manager in GNOME, stores the session information in files located in the ~/.metacity/sessions directory (this is in your home directory). The session files are text files; if you are curious, you can browse these file with the more command.

Initially, when you don't yet have a session file, the GNOME session comes from the /usr/share/gnome/default.session file. However, as soon as the session starts, the GNOME session manager (/usr/bin/gnome-session) saves the current session information in the~/.metacity/sessions directory. A typical initial GNOME desktop produced by the session description in the default session file is shown in *Figure 1*.

Figure 1: The Initial GNOME Desktop, with the Default Session File.

As you can see from the icons on the left side of the GNOME desktop, GNOME enables you to place folders and applications directly on the desktop. This is similar to the way in which you can place icons directly on the Microsoft Windows desktop.

2.4 USING THE SHELL

Computer understands the language of 0's and 1's called binary language.

In early days of computing, instruction are provided using binary language, which is difficult for all of us, to read and write. So in Os there is special program called Shell. Shell accepts your instruction or commands in English (mostly) and if it's a valid command, it is passed to kernel.

Shell is a user program or its environment provided for user interaction. Shell is a command language interpreter that executes commands read from the standard input device (keyboard) or from a file.

Shell is not part of system kernel, but uses the system kernel to execute programs, create files etc.

Several shell available with Linux including:

BASH(Bourne Again Shell), CSH(C Shell), KSH(Korn Shell) and TCSH

Tip: To find all available shells in your system type following command: **$ cat /etc/shells**

Note that each shell does the same job, but each understand a different command syntax and provides different built-in functions.

In MS-DOS, Shell name is COMMAND.COM which is also used for same purpose, but it's not as powerful as our Linux Shells are!

2.5 UNDERSTANDING USERS AND FILE SYSTEMS

This subsection describes how the Linux kernel maintains the files in the file systems that it supports. It describes the Virtual File System (VFS) and explains how the Linux kernel's real file systems are supported.

One of the most important features of Linux is its support for many different file systems. This makes it very flexible and well able to coexist with many other operating systems. At the time of writing, Linux supports 15 file systems; ext, ext2, xia, minix, umsdos, msdos, vfat, proc, smb, ncp, iso9660, sysv, hp fs, affs and ufs, and no doubt, over time more will be added.

In Linux, as it is for Unix ™, the separate file systems the system may use are not accessed by device identifiers (such as a drive number or a drive name) but instead they are combined into a single hierarchical tree structure that represents the file system as one whole single entity. Linux adds each new file system into this single file system tree as it is mounted. All file systems, of whatever type, are mounted onto a directory and the files of the mounted file system cover up the existing contents of that directory. This directory is known as the mount directory or mount point. When the file system is unmounted, the mount directory's own files are once again revealed.

When disks are initialized (using fdisk, say) they have a partition structure imposed on them that divides the physical disk into a number of logical partitions. Each partition may hold a single file system, for example an EXT2file system. File systems organize files into logical hierarchical structures with directories, soft links and so on held in blocks on physical devices. Devices that can contain file systems are known as block devices. The IDE disk partition /dev/hda1, the first partition of the first IDE disk drive in the system, is a block device. The Linux file systems regard these block

devices as simply linear collections of blocks, they do not know or care about the underlying physical disk's geometry. It is the task of each block device driver to map a request to read a particular block of its device into terms meaningful to its device; the particular track, sector and cylinder of its hard disk where the block is kept. A file system has to look, feel and operate in the same way no matter what device is holding it. Moreover, using Linux's file systems, it does not matter (at least to the system user) that these different file systems are on different physical media controlled by different hardware controllers. The file system might not even be on the local system, it could just as well be a disk remotely mounted over a network link. Consider the following example where a Linux system has its root file system on a SCSI disk:

```
A     E     boot   etc   lib   opt   tmp   usr
C     F     cdrom  fd    proc  root  var   sbin
D     bin   dev    home  mnt   lost+found
```

Neither the users nor the programs that operate on the files themselves need know that /C is in fact a mounted VFAT file system that is on the first IDE disk in the system. In the example (which is actually my home Linux system), /E is the master IDE disk on the second IDE controller. It does not matter either that the first IDE controller is a PCI controller and that the second is an ISA controller which also controls the IDE CDROM. I can dial into the network where I work using a modem and the PPP network protocol using a modem and in this case I can remotely mount my Alpha AXP Linux system's file systems on /mnt/remote.

The files in a file system are collections of data; the file holding the sources to this chapter is an ASCII file called filesystems.tex. A file system not only holds the data that is contained within the files of the file system but also the structure of the file system. It holds all of the information that Linux users and processes see as files, directories soft links, file protection information and so on. Moreover, it must hold that information safely and securely, the basic integrity of the operating system depends on its file systems. Nobody would use an operating system that randomly lost data and files[1].

Minix, the first file system that Linux had is rather restrictive and lacking in performance.

Its filenames cannot be longer than 14 characters (which is still better than 8.3 filenames) and the maximum file size is 64MBytes. 64Mbytes might at first glance seem large enough but large file sizes are necessary to hold even modest databases. The first file system designed specifically for Linux, the Extended File system, or EXT, was introduced in April 1992 and cured a lot of the problems but it was still felt to lack performance.

So, in 1993, the Second Extended File system, or EXT2, was added.

An important development took place when the EXT file system was added into Linux. The real file systems were separated from the operating system and system services by an interface layer known as the Virtual File system, or VFS.

VFS allows Linux to support many, often very different, file systems, each presenting a common software interface to the VFS. All of the details of the Linux file systems are translated by software so that all file systems appear identical to the rest of the Linux kernel and to programs running in the system. Linux's Virtual File system layer allows you to transparently mount the many different file systems at the same time.

The Linux Virtual File system is implemented so that access to its files is as fast and efficient as possible. It must also make sure that the files and their data are kept correctly. These two requirements can be at odds with each other. The Linux VFS caches information in memory from each file system as it is mounted and used.

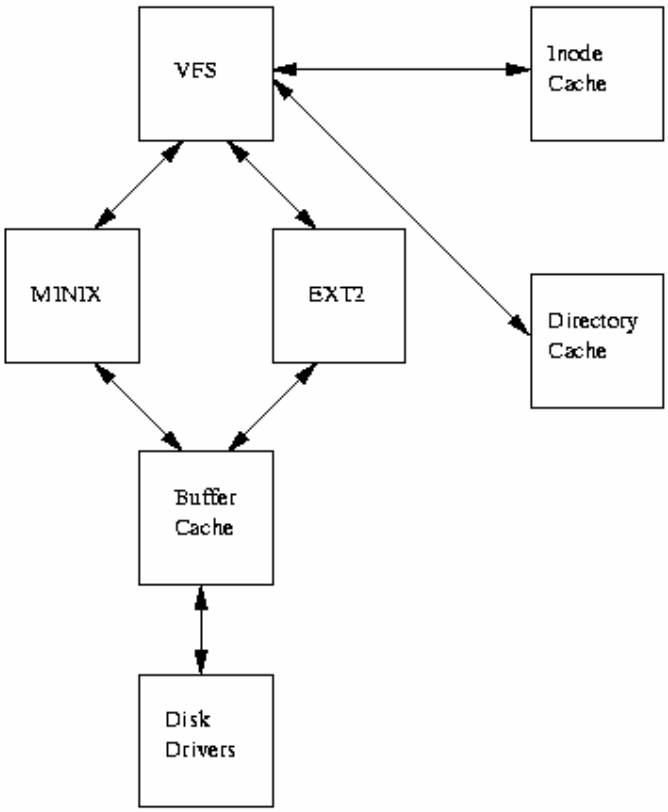


Figure 2: A Logical Diagram of the Virtual File System

The figure 2 shows the relationship between the Linux kernel's Virtual File System and it's real file systems. The virtual file system must manage all of the different file systems that are mounted at any given time. To do this it maintains data structures that describe the whole (virtual) file system and the real, mounted, file systems.

The VFS Superblock

Every mounted file system is represented by a VFS superblock; amongst other information, the VFS superblock contains the:

Device

This is the device identifier for the block device that this file system is contained in. For example, /dev/hda1, the first IDE hard disk in the system has a device identifier of *0x301*,

Inode pointers

The mounted inode pointer points at the first inode in this file system. The covered inode pointer points at the inode representing the directory that this file system is mounted on. The root file system's VFS superblock does not have a covered pointer,

Blocksize

The block size in bytes of this file system, for example 1024 bytes,

Superblock operations

A pointer to a set of superblock routines for this file system. Amongst other things, these routines are used by the VFS to read and write inodes and superblocks.

File System type

A pointer to the mounted file system's file_system_type data structure,

File System specific

A pointer to information needed by this file system,

The Linux Operating System has several user accounts. A few are preconfigured and an option of defining new is also available.

Preconfigured user accounts

– Root
  - Administrative account
  - Also called superuser
  - Can perform any operation on Linux system
  - Do not log in as root for normal work
  - Change temporarily to root user
– Regular user accounts
  - Users who log in at keyboard and use Linux system
  - Commonly associated with named individuals
– Special user account
  - Used by Linux programs
  - Created during installation of Linux
  - Vary depending on services installed

User information commands:

- id command
  – Shows effective UID
- logname command
  – View user name that you used to log in
- whoami command
  – Shows user name of currently effective UID
- groups command
  – Lists all groups you are a member of

2.6 LINUX UTILITIES AND BASIC COMMANDS

The following are some of the basic commands used in the CLI Linux operating system, however GUI option is always available, as in the case of Windows:

a) alias Alias is used to substitute a small or more familiar name in place of a long string. It is commonly used for long strings that are frequently used.

Syntax alias [name=['command']]

Name	Specifies the alias name.
Command	Specifies the command the name should be an alias for.
-a	Removes all alias definitions from the current shell execution environment.
-t	Sets and lists tracked aliases.
-x	Sets or prints exported aliases. An exported alias is defined for scripts invoked by name.

Example $alias home 'cd public_html' - Sets home to type cd public_html. Use the command 'unalias' to remove this alias.

Alias command can be used even for the following:

$alias clr clear	$alias cls clear
$alias copy cp –I	$alias del rm –I
$alias delete rm –I	$alias home cd ~
$alias md mkdir	$alias move mv –I
$alias type more	

b) awk awk utility is powerful data manipulation/scripting programming language (In fact based on the C programming Language). Use awk to handle complex task such as calculation, database handling, report creation etc.

Syntax awk -f {awk program file} filename

awk Program contains are something as follows:

```
Pattern{
    action1
    action2
    actionN
}
```

awk reads the input from given file (or from stdin also) one line at a time, then each line is compared with pattern. If pattern is match for each line then given action is taken. Pattern can be regular expressions.

Following is the summery of common awk metacharacters:

Metacharacter	Meaning
(Dot)	Match any character
*	Match zero or more character
^	Match beginning of line
$	Match end of line
\	Escape character following
[ ]	List
{ }	Match range of instance
+	Match one more preceding
?	Match zero or one preceding
|	Separate choices to match

c) cd The cd sets the working directory of a process.

Syntax cd <directory name>

Example $cd /etc

d) chmod Chmod is a utility that changes the permission of a file.

Syntax *chmod [OPTION]... MODE[,MODE]... FILE...*

Permissions
u - User who owns the file. *g* - Group that owns the file.
o - Other. *a* - All.
r - Read the file. *w* - Write or edit the file.
x - Execute or run the file as a program.

Numeric Permissions:

CHMOD can also to attributed by using Numeric Permissions:

400 read by owner 040 read by group 004 read by anybody
200 write by owner 020 write by group 002 write by anybody
100 execute by owner 010 execute by group 001 execute by anybody

Example $chmod 644 file.htm
This gives the file read/write by the owner and only read by everyone else (-rw-r--r--).

$chmod 755 file.cgi

$chmod 666 file.txt

e) chown Chown is a utility that is also used to change file ownership.

Syntax chown [R] *owner*[:*group*] *file* ...

-R recursively change file user and group IDs. For each *file* operand that names a directory, *chown* shall change the user ID (and group ID, if specified) of the directory and all files in the file hierarchy below it.

Example #chown root file.txt

f) cp The cp command is used to copy files.

Syntax $cp source destination

Example $cp abc.txt pqr.txt [copy the file abc.txt as pqr.txt]

$cp *.txt /etc/ [copy all files with .txt extension into /etc directory]

$cp a* /etc/ [copy all files starts with the letter 'a' into /etc directory]

g) date An essential command to set the date and time. Also a useful way to output current information when working in a script file.

Example $date

h) df The df command reports filesystem disk space usage.

With no arguments, 'df' reports the space used and available on all currently mounted filesystems (of all types). Otherwise, `df' reports on the filesystem containing each argument *file*.

Syntax df [*option*]... [*file*]...

Normally the disk space is printed in units of 1024 bytes, but this can be overridden.

OPTIONS

'-i' List inode usage information instead of block usage. An inode (short for index node) is contains information about a file such as its owner, permissions, timestamps, and location on the disk.

'-k' Print sizes in 1024byte blocks, overriding the default block size.

'-m' Print sizes in megabyte (i.e.; 1,048,576-byte) blocks.

Example $df –k $df –m $df –i

i) pwd To know the current working directory

Syntax pwd

Example $pwd

j) ln The ln command makes new, alternate file names for a file by hard linking, letting multiple users share one file. The ln command creates pseudonyms for files which allows them to be accessed by different names. These pseudonyms are called links. There are two different forms of the command and two different kinds of links that can be created.

Syntax **ln** [*options*] *exiting_path* [*new_path*]

ln [*options*] *exiting_paths directory*

In the first form, a new name is created called *new_path* which is a pseudonym for *existing_path*. In the second form, the last argument is taken to be a directory name and all the other arguments are paths to existing files. A link for each existing file is created in the specified directory with the same filename as the existing files.

Example Create a link named my_file in the current directory to the file /home/bill/his_file:

$ln /home/bill/his_file my_file

As above but the link is created in /home/joe/my_file:

$ln /home/bill/his_file /home/joe/my_file

To create a symbolic link, all works as above except you need to include the -s option.

For example, to make a symbolic link called Linux that points to fedora:

$ln -s Linux fedora

The only way to see that Linux is a symbolic link is by using the ls -l command (ls -l Linux). The output of this command will look much like this:

lrwxrwxrwx 1 joe users 3 2009-03-21 17:26 Linux -> fedora

k) ls The ls command shows information about files. It lists the contents of a directory in order to determine when the configurations files were last edited.

Syntax ls [-a] [-A] [-c] [-d] [-i] [-l] [-L] [n] [-r] [-R] [pathnames]

-a: shows all files, even files that are hidden (these files begin with a dot.)

-A: list all files including the hidden files. However, does not display the working directory (.) or the parent directory (..).

-c: use time of last modification of the i-node (file created, mode changed, and so forth) for sorting (-t) or printing (-l or -n).

-d: if an argument is a directory it only lists its name not its contents.

-i: for each file, print the i-node number in the first column of the report.

-l: shows you huge amounts of information (permissions, owners, size, and when last modified.)

-L: if an argument is a symbolic link, list the file or directory the link references rather than the link itself.

-n: The same as -l, except that the owner's UID and group's GID numbers are printed, rather than the associated character strings.

-r: reverses the order of how the files are displayed.

-R: includes the contents of subdirectories.

Example $ls –l $ls –d $ls -r

l) man Short for "manual," man displays information about commands and a keyword search mechanism for needed commands.

Syntax man <command>

Example $man find $man chown

m) passwd A quick and easy way to change passwords on a system.

Syntax passwd [-d account name]

Example #passwd

#passwd –d xyz [to delete the password of the account ‘xyz’]

n) Shutdown Shutdown is a command that turns off the computer and can be combined with variables such as -h for halt or -r for reboot.

Syntax Shutdown [-h][-r]

Example #shutdown

o) top Top provides an ongoing look at processor activity in real time. It displays a listing of the most CPU-intensive tasks on the system, and can provide an interactive interface for manipulating processes.

Syntax top

Example $top

p) vmstat The vmstat command is used to get a snapshot of everything in a system, helping admins determine whether the bottleneck is CPU, memory or I/O. Run this command to get virtual memory statistics. vmstat reports information about processes, memory, paging, block IO, traps, and cpu activity.

Syntax vmstat [-a] [-n] [*delay* [*count*]]
vmstat [-f] [-s] [-m]
vmstat [-S unit]
vmstat [-d]
vmstat [-p disk partition]
vmstat [-V]

Example $vmstat

2.7 UNDERSTANDING TEXT PROCESSING

The most commonly used text editor for Linux is vi. and its version that is usually found in PCs is vim. Following is detailed user guide for using vi editor for text processing. The vi editor has two modes:- editing mode and command mode.

Starting vi

At the Bash command prompt, type vi or vi some-filename. vi is a modal editor in which you are either in editing mode or command mode.

Getting out of vi

This is the most important thing you need to know about vi: how to get out of it. If you are in editing mode, hit ESC to enter command mode. Enter :wq to write your file and quit. To quit without writing the file, hit :q. If you really want to quit without saving the file you have made changes to hit :q!. To save the file under a different name, try :wq new-filename.

Switching Between Editing Mode and Command mode:

If you are in editing mode, hit ESC to enter command mode. If you are in command mode hit i to enter insert (editing) mode. If you cannot tell which mode you are in, trying hitting ESC several times to make sure you are in command mode, then type :set showmode. This may tell vi to let you know when you are in insert mode (it depends on the version of vi you are using).

Moving Around a Character at a Time

The original vi was written by Bill Joy (at UC Berkeley) using a computer system and keyboard that lacked arrow, page up, and page down keys. This is the reason for the stupid assignment of the keys for moving around: h (move left one character), j (move down one line), k (move up one line), and l (move right one character). On Joy's terminal, these four keys also had arrow keys on them and that is the historical reason they are still mapped that way today. However, if you have your terminal program, i.e., PuTTY, configured correctly, you should be able to use the arrow keys.

Moving Around a Word at a Time

Switch to command mode. Press w to move forward one word. Press b to move backward one word. Press nw to move forward n words, e.g., 3w to move forward three words. Press nw to move backward n words.

Moving Around the File

Switch to command mode. To move to the end of the line, hit $. To move to the beginning, hit 0. Hit 1G to go the first line of text. Hit nG to go line number n. Hit G to go to the end of file. To display line numbers, in command mode type :set number. Note that:13 is equivalent to 13G. To page down, hit Ctrl+F. To page up hit Ctrl+B.

Deleting Characters

Switch to command mode, move to the character you want to delete, and hit x to delete that character. To delete n characters hit nx, e.g., 17x to delete 17 characters. To delete all the characters from the cursor position to the end of the line hit D.

Copying a Line of Text

To copy a line of text from one place in the file to another, switch to command mode. Move to the line you want to copy. Hit yy (yank something or other, I dunno). Move the cursor to the location where you want to make the copy. Hit p (lowercase p for paste) to put the copied line after the current line. Hit P (uppercase P) to put the copied line before the current line.

Moving a Line of Text

To move a line of text from one place in the file to another, switch to command mode. Move to the line you want to copy. Hit dd. Move the cursor to the location where you want to make the move. Hit p (lowercase p) to move the line after the current line. Hit P (uppercase P) to move the line before the current line.

Deleting Lines of Text

Switch to command mode. Move to the line you want to delete. Hit dd. Hit ndd to delete n lines of text, e.g., 3dd will delete the line of text the cursor is on and the next two lines.

Cutting, Copying, Pasting Multiple Lines of Text

Use nyy to copy n lines of text to the copy buffer. Move to where you want the lines to be and hit p or P to copy them. Use ndd to delete n lines of text to the copy buffer, and then move to where you want to paste the lines and hit p or P.

Moving and Copying Text in Visual Mode

To cut, copy, and paste text that is not an entire line, enter visual mode by hitting v in command mode. Use the arrow keys to move around and select the text you want to copy or move. Once selected, hit y to copy the text to the copy buffer or d to delete the text to the buffer. Once copied (with y) or deleted with (with d) move to where you want the text to be. Hit p or P to paste the text.

Finding Text

Switch to command mode. Hit / (forward slash) and enter the string you are searching for, e.g., /cookies. The cursor will be placed on the first occurrence of cookies following the current location. Hitting n or Enter over-and-over will continue searching.

Find and Replace

Switch to command mode. To replace all occurrences of "homer" by "bart", try something like :1,100 s/homer/bart/g. The 1,100 part means search from line 1 to line 100 inclusive (I think 1,$ will search the entire file). The s stands for search. We are searching for "homer" and replacing all occurrences by "bart". The g stands for global which means it will replace all occurrences without prompting you for each one. If you want to be prompted for each one, use c instead (for confirmation).

Undo and Redo

Since you will use it the most because of vi's horrible user interface the most useful command in vi (other than :q!) is the undo command. Hitting u will undo your last change. Hitting u multiple times or using nu will perform multiple undos. Redo is Ctrl+R.

More Help

Let's say you're either insane or a masochist and want to learn more about vi. In command mode, hit :h to bring up help. The most useful thing you need to know here is how to get out of help, hit :q (like in less).

2.8 MANAGING PROCESSES

Process is kind of program or task carried out by your PC. For e.g.
$ ls -lR
ls command or a request to list files in a directory and all subdirectory in your current directory - It is a process.

Process defined as:
"*A process is program (command given by user) to perform specific Job. In Linux when you start process, it gives a number to process (called PID or process-id), PID starts from 0 to 65535.*"

Linux is multi-user, multitasking Os. It means you can run more than two process simultaneously if you wish. For e.g. To find how many files do you have on your system you may give command like:

$ ls / -R | wc -l
This command will take lot of time to search all files on your system. So you can run such command in Background or simultaneously by giving command like

$ ls / -R | wc -l &
The **ampersand** (**&**) at the end of command tells shells start process (**ls / -R | wc -l**) and run it in background takes next command immediately.

Process & PID defined as:
"*An instance of running command is called **process** and the number printed by shell is called **process-id (PID)**, this PID can be use to refer specific running process.*"

Table 1 shows most commonly used command(s) with process:

Table 1: Commonly used command

For this purpose	Use this Command	Examples*
To see currently running process	ps	**$ ps**
To stop any process by PID i.e. to kill process	kill {PID}	**$ kill 1012**
To stop processes by name i.e. to kill process	killall {Process-name}	**$ killall httpd**
To get information about all running process	ps -ag	**$ ps –ag**
To stop all process except your shell	kill 0	**$ kill 0**
For background processing (With &, use to put particular command and program in background)	Linux-command &	**$ ls / -R	wc -l &**
To display the owner of the processes along with the processes	ps aux	**$ ps aux**
To see if a particular process is running or not. For this purpose you have to use ps command in	ps ax	grep process-U-want-to see	For e.g. you want to see whether Apache web server

combination with the grep command		process is running or not then give command **$ ps ax | grep httpd**
To see currently running processes and other information like memory and CPU usage with real time updates.	top See the output of top command.	**$ top** **Note** that to exit from top command press q.
To display a tree of processes	pstree	**$ pstree**

***** To run some of this command you need to be root or equivalent user.

Note that you can only kill process which are created by yourself. A Administrator can almost kill 95-98% process. But some process can not be killed, such as VDU Process.

2.9 SUMMARY

The section widely covers the introduction of Linux operating system. This familiarizes the reader with its potential and its usefulness. The section gives an enriched source of action for practical learning of text editor vi and the process management.

2.10 REFERENCES/FURTHER READINGS

- http://www.linux.org
- http://www.fedoraproject.org
- http://www.redhat.com/docs/manuals/linux
- UNIX and Linux System Administration Handbook, Prentice Hall

SECTION 3 C PROGRAMMING

Structure **Page Nos.**

3.0 Introduction 36
3.1 Objectives 36
3.2 Introduction to C 36
3.3 Summary 46
3.4 Further Readings 47

3.0 INTRODUCTION

In the earlier unit, we introduced you to the architecture of Unix operating system, especially the Unix command required by the users. In this unit we present C language – because BSD sockets library and its interface are written in C, hence it is essential for students to learn programming in C language, also students should learn about the compilation, execution and testing of programs, system calls and how to get Unix help. C programming language is widely known for its power for these problem-solving techniques using computer. A language is a mode of communication between two people. It is necessary for those two people to understand the language in order to communicate. But even if the two people do not understand the same language, a translator can help to convert one language to the other, understood by the second person. Similar to a translator is the mode of communication between a user and a computer is a computer language. One form of the computer language is understood by the user, while in the other form it is understood by the computer. A translator (or compiler) is needed to convert from user's form to computer's form. Like other languages, a computer language also follows a particular grammar known as the syntax. In this unit we will introduce you the basics of programming language C which is essential for socket programming.

3.1 OBJECTIVES

After going through this unit, you should be able to:

- understand what is a C programming language?
- compile a C program;
- identify the syntax errors; Run and debug a C program; and
- understand what are run time and logical errors.

3.2 INTRODUCTION TO C

Computer programming languages are developed with the primary objective of facilitating a large number of people to use computers without the need for them to know in detail the internal structure of the computer. Languages are designed to be *machine-independent*. Most of the programming languages ideally designed, to execute a program on any computer regardless of who manufactured it or what model it is. Programming languages can be divided into two categories:

(i) **Low Level Languages or Machine Oriented Languages:** The language whose design is governed by the circuitry and the structure of the machine is known as the **Machine language**. This language is difficult to learn and use. It is specific to a given computer and is different for different computers i.e., these languages are **machine-dependent**. These languages have been designed to give a better machine efficiency, i.e., faster program execution. Such languages are also known as Low Level Languages. Another type of Low-

Level Language is the Assembly Language. We will code the assembly language program in the form of mnemonics. Every machine provides a different set of mnemonics to be used for that machine only depending upon the processor that the machine is using.

(ii) **High Level Languages or Problem Oriented Languages:** These languages are particularly oriented towards describing the procedures for solving the problem in a concise, precise and unambiguous manner. Every high level language follows a precise set of rules. They are developed to allow application programs to be run on a variety of computers. These languages are *machine-independent*. Languages falling in this category are FORTRAN, BASIC, PASCAL etc. They are easy to learn and programs may be written in these languages with much less effort. However, the computer cannot understand them and they need to be translated into machine language with the help of other programs known as Compilers or Translators.

Prior to writing C programs, it would be interesting to find out what really is the C language, how it came into existence and where does it stand with respect to other computer languages. We will briefly outline these issues in the following section.

History of C

C is a programming language developed at AT&T's Bell Laboratory of USA in 1972. It was designed and written by Dennis Ritchie. As compared to other programming languages such as Pascal, C allows a precise control of input and output.

Now let us see its historical development. The late 1960s were a turbulent era for computer systems research at Bell Telephone Laboratories. By 1960, many programming languages came into existence, almost each for a specific purpose. For example, COBOL was being used for Commercial or Business Applications, FORTRAN for Scientific Applications and so on. So, people started thinking why could not there be a one general-purpose language. Therefore, an International Committee was set up to develop such a language, which came out with the invention of ALGOL60. But this language never became popular because it was too abstract and too general. To improve this, a new language called Combined Programming Language (CPL) was developed at Cambridge University. But this language was very complex in the sense that it had too many features and it was very difficult to learn. Martin Richards at Cambridge University reduced the features of CPL and developed a new language called Basic Combined Programming Language (BCPL). But unfortunately it turned out to be much less powerful and too specific. Ken Thompson at AT & T's Bell Labs developed a language called B at the same time as a further simplification of CPL. But like BCPL, this was also too specific. Ritchie inherited the features of B and BCPL and added some features on his own and developed a language called C. C proved to be quite compact and coherent. Ritchie first implemented C on a DEC PDP-11 that used the Unix Operating System.

For many years the *de facto* standard for C was the version supplied with the Unix Version 5 operating system. The growing popularity of microcomputers led to the creation of a large number of C implementations. At the source code level most of these implementations were highly compatible. However, since no standard existed there were discrepancies. To overcome this situation, ANSI established a committee in 1983 that defined an ANSI standard for the C language.

Salient features of C

C is a general purpose, structured programming language. Among the two types of programming languages discussed earlier, C lies in between these two categories. That's why it is often called a ***middle level language.*** It means that it combines the elements of high-level languages with the functionality of assembly language. It provides relatively good programming efficiency (as compared to machine-oriented language) and relatively good machine efficiency as compared to high-level

languages). As a middle level language, C allows the manipulation of bits, bytes and addresses – the basic elements with which the computer executes the inbuilt and memory management functions. The flexibility of C allows it to be used for systems programming as well as for application programming.

C is commonly called a structured language because of structural similarities to ALGOL and Pascal. The distinguishing feature of a structured language is compartmentalization of code and data. Structured language is one that divides the entire program into modules using top-down approach where each module executes one job or task. Structured language is one that allows only one entry and one exit point to/from a block or module. It is easy for debugging, testing, and maintenance if a language is a structured one. C supports several control structures such as **while, do-while and for** and various data structures such as **strucs, files, arrays** etc. as would be seen in the later units of this course. The basic unit of a C program is a **function -** C's standalone subroutine**.** The structural component of C makes the programming and maintenance easier.

C Program Structure

As we have already seen, to solve a problem there are three main things to be taken into consideration. Firstly, what should be the output? Secondly, what should be the inputs that will be required to produce this output? Thirdly, the steps of instructions which may be used for these inputs to produce the required output. As stated earlier, every programming language follows a set of rules; therefore, a program written in C also follows predefined rules known as syntax. C is a case sensitive language**.** All C programs consist of one or more functions. One function that must be present in every C program is **main ()**. This is the first function called up when the program execution begins. Basically, **main()** outlines what a program does. Although **main** is not given in the keyword list, it cannot be used for naming a variable. The structure of a C program is illustrated in *Figure1* where functions func1 ( ) through funcn ( ) represent user defined functions.

```
Preprocessor directives
Global data declarations
main ( )      /* main function*/
{
        Declaration part;
        Program statements;
 }
  /*User defined functions*/
func1( )
{
        ……
}
 func2 ( )
{
        ……
 }
 .
 .
 .
 funcn ( )
 {
        ……
 }
```

Figure 1: Structure of a C Program

A Simple C Program

From the above sections, you have become familiar with a programming language and structure of a C program. Its now time to write a simple C program. This program will illustrate how to print out the message "This is a C program".

Example 2.1: Write a program to print a message on the screen.

```
/*Program to print a message*/
#include <stdio.h>              /* header file*/
main()                          /* main function*/
{
  printf("This is a C program\n");  /* output statement*/
}
```

Though the program is very simple, a few points must be noted.

Every C program contains a function called **main()**. This is the starting point of the program. This is the point from where the execution begins. It will usually call other functions to help perform its job. Some functions we have to write and others are used from the standard libraries.

#include <stdio.h> is a reference to a special file called stdio.h which contains information that must be included in the program at the time when it is compiled. The inclusion of this required information will be handled automatically by the compiler. You will find it at the beginning of almost every C program. Basically, all the statements starting with # in a C program are called preprocessor directives. These will be considered in the later units. Just remember, that this statement allows you to use some predefined functions such as, *printf(),* in this case.

main() declares the start of the function, while the two curly brackets { } show the start and finish of the function. Curly brackets in C are used to group statements together as a function, or in the body of a loop or a block. Such a grouping is known as a compound statement or a block. Every statement within a function ends with a terminator semicolon (;).

printf("This is a C program\n"); prints the words on the screen. The text to be printed is enclosed in double quotes. The **\n** at the end of the text tells the program to print a new line as part of the output. That means now if we give a second printf() statement, it will be printed in the next line.

Comments may appear anywhere within a program, as long as they are placed within the delimiters **/*** and ***/.** Such comments are helpful in identifying the program's principal features or in explaining the underlying logic of various program features.

While useful for teaching, such a simple program has few practical uses. Let us consider something rather more practical. Let us look into the example that describes complete program development life cycle given below:

Example

Develop an algorithm, flowchart and program to add two numbers.

Algorithm

1) Start

2) Input the two numbers ***a*** and ***b***

3) Calculate the sum as ***a+b***

4) Store the result in ***sum***

5) Display the result

6) Stop.

Flowchart

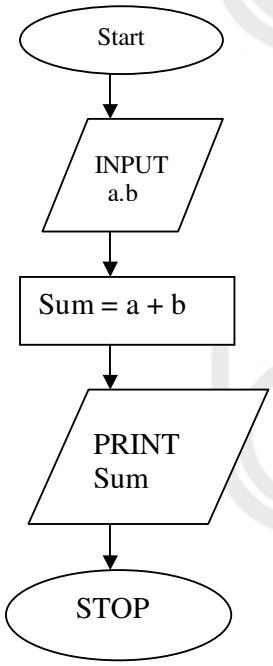


Figure 2: Flowchart to add two numbers

Program

```
#include <stdio.h>

main()
{
  int a,b,sum;                    /* variables declaration*/
   printf("\n Enter the values for a and b: \n");
  scanf("%d, %d", &a, &b);
  sum=a+b;

  printf("\nThe sum is %d",sum);     /*output statement*/
}
```

OUTPUT

Enter the values of a and b:

2 3

The sum is 5.

In the above program considers two variables *a* and *b*. These variables are declared as integers **(int)**, which is the data type to indicate integer values. The next statement is the printf() statement meant for prompting the user to input the values of ***a*** and ***b***. scanf() is the function to intake the values into the program provided by the user. Next comes the processing / computing part which computes the **sum**. Again the **printf()** statement is a bit different from the first program; it includes a format specifier (%d). The format specifier indicates the kind of value to be printed. We will study about other data types and format specifiers in detail in the following units. In the printf() statement above, sum is not printed in double quotes because we want its value to be printed. The number of format specifiers and the variable should match in the printf() statement.

At this stage, we don't go much into detail. However, in the following units you will be learning all these details.

Writing a Program

A C program can be executed on platforms such as DOS, Unix etc. like DOS, **Unix** also stores C program in a file with extension is ***.c***. This identifies it as a C program. The easiest way to enter your text is using a text editor like *vi*, *emacs* or *xedit*. To edit a file called testprog.c using vi type.

$ vi testprog.c

The editor is also used to make subsequent changes to the program.

Compiling a Program

After you have written the program, the next step is to save the program in a file with extension. c. This program is in high-level language. But this language is not understood by the computer directly. So, the next step is to convert the high-level language program (source code) to machine language (object code). This task is performed by a software or program known as a **compiler**. Every language has its own compiler that converts the source code to object code. The compiler will compile the program successfully if the program is syntactically correct; else the object code will not be produced. This is explained pictorially in *Figure 3*.

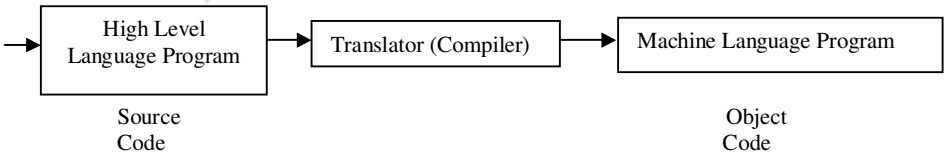


Figure 3: Process of Translation

The C Compiler

If the name of the program file is testprog.c, the simplest method is compile it to type cc testprog.c

This will compile testprog.c, and, if successful, will produce an executable file called ***a.out***. If you want to give the executable file any other name, you can type

cc testprog.c -o testprog

This will compile ***testprog.c***, creating an executable file testprog.

Syntax and Semantic Errors

Every language has an associated grammar, and the program written in that language has to follow the rules of that grammar. For example, in English a sentence such a "Shyam, is playing, with a ball". This sentence is syntactically incorrect because commas should not come the way they are in the sentence.

Likewise, C also follows certain syntax rules. When a C program is compiled, the compiler will check that the program is syntactically correct. If there are any syntax errors in the program, those will be displayed on the screen with the corresponding line numbers.

Let us consider the following program.

Example 2.3: Write a program to print a message on the screen.

```
/* Program to print a message on the screen*/
#include <stdio.h
main( )
{
printf("Hello, how are you\n")
```

Let the name of the program be **test.c** .If we compile the above program as it is we will get the following errors:

Error test.c 1:No file name ending;
Error test.c 5: Statement missing ;
Error test.c 6: Compound statement missing }

Edit the program again, correct the errors mentioned and the corrected version appears as follows:

```
#include <stdio.h>
main( )
{
  printf ("Hello, how are you\n");
}
```

Apart from syntax errors, another type of errors that are shown while compilation, are semantic errors. These errors are displayed as warnings. These errors are shown if a particular statement has no meaning. The program does compile with these errors, but it is always advised to correct them also, since they may create problems while execution. The example of such an error is that say you have declared a variable but have not used it, and then you get a warning "code has no effect". These variables are unnecessarily occupying the memory.

Link and Run the C Program

After compilation, the next step is linking the program. Compilation produces a file with an extension **.obj**. Now this **.obj** file cannot be executed since it contains calls to functions defined in the standard library (header files) of C language. These functions have to be linked with the code you wrote. C comes with a standard library that provides functions that perform most commonly needed tasks. When you call a function that is not the part of the program you wrote, C remembers its name. Later the linker combines the code you wrote with the object code already found in the standard library. This process is called *linking.* In other words, Linker is a program that links separately compiled functions codes together into one program. It combines the functions in the standard C library with the code that you wrote. The output of the linker in an executable program.

Unix also includes a very useful program called **make**. **Make** allows very complicated programs to be compiled quickly, by reference to a configuration file (usually called makefile). If your C program is a single file, you can usually use make by simply typing –

Make testprog

This will compile **testprog.c** as well as link your program with the standard library so that you can use the standard library functions such as printf() and put the executable code in **testprog.**

Linker Errors

If a program contains syntax errors then the program does not compile, but it may happen that the program compiles successfully but we are unable to get the executable file, this happens when there are certain linker errors in the program. For example, the object code of a certain standard library function is not present in the standard C library; the definition for this function is present in the header file that is why we do not get a compiler error. Such kinds of errors are called linker errors. The executable file would be created successfully only if these linker errors are corrected.

Logical and Runtime Errors

After the program is compiled and linked successfully we execute the program. Now there are three possibilities:

1) The program executes and we get correct results,

2) The program executes and we get wrong results, and

3) The program does not execute completely and aborts in between.

The first case simply means that the program is correct. In the second case, we get wrong results; it means that there is some logical mistake in our program. This kind of error is known as **logical error**. This error is the most difficult to correct. This error is corrected by debugging. Debugging is the process of removing the errors from the program. This means manually checking the program step by step and verifying the results at each step. Suppose we have to find the average of three numbers and we write the following code:

Example: Write a C program to compute the average of three numbers

```
/* Program to compute average of three numbers *?
#include<stdio.h>
main( )
{
  int a,b,c,sum,avg;
  a=10;
  b=5;
  c=20;
  sum = a+b+c;
  avg = sum / 3;
  printf("The average is %d\n", avg);
}
```

OUTPUT

The average is 8.

The exact value of average is 8.33 and the output we got is 8. So we are not getting the actual result, but a rounded off result. This is due to the logical error. We have declared variable **avg** as an integer but the average calculated is a real number, therefore only the integer part is stored in **avg.** Such kinds of errors which are not detected by the compiler or the linker are known as **logical errors**.

The third kind of error is only detected during execution. Such errors are known as **run time errors**. These errors do not produce the result at all. The program execution stops in between and a run time error message are flashed on the screen. Let us look at the following example:

Example: **Write a program to divide a sum of two numbers by their difference**

```
/* Program to divide a sum of two numbers by their difference*/
#include <stdio.h>
 main( )
 {
  int a, b;
  float c;

  a=10;
  b=10;
```

```
    c = (a+b) / (a-b);
    printf("The value of the result is %f\n",c);
}
```

The above program will compile and link successfully, it will execute till the first *printf()* statement and we will get the message in this statement, as soon as the next statement is executed we get a runtime error of "Divide by zero" and the program halts. Such kinds of errors are **runtime errors**.

Diagrammatic Representation of Program Execution Process

The *Figure 4* shows the diagrammatic representation of the program execution process.

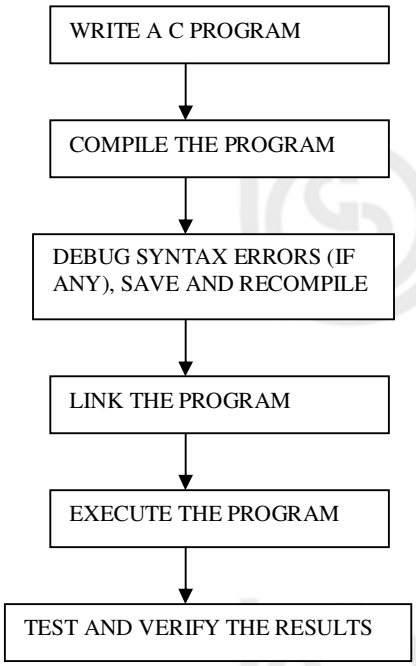


Figure 4: Program Execution Process

Debugging and Optimisation

In the previous section you have studied the basics C programming, and got elementary knowledge for compiling and executing programs but you need some more detailed information about compilation, execution, debugging, optimization, linking and standards libraries of Unix operating system.

Debugging

Debugging of a program includes many things like running it step by step, setting the break point before a given command is executed, checking at contents of variables during program execution, etc. To relate between the executable program and the original source code, we should ask the compiler to insert information to the resulting executable program to help the debugger or tester. This information is called "debug information". To add the debugging information to your program you should compile it with "-g" flag option, as shown below in the command.

- **% cc -g Prog_Debug.c -o Prog_Debug**

The '-g' option informs the compiler to use debug info. You will find that the resulting file is larger in size than the file created without '-g' flag. This difference in size is due to the additional debug information. But if you want to remove this information you can use the strip command, as given below:

- **% strip Prog_Debug**

Creating an Optimized Code

Once you have created a program and debugged it, you should compile it into an efficient and optimize code, you may ask why should I optimize the code? After optimization of a code, it runs faster and it takes lesser space. To create an optimized program **prog1**, you can write the following command:

- **% cc -O Prog1.c -o Prog1**

The '-O' option informs the compiler to optimize the code. Because various optimization algorithms are applied to code so the compiler process it slower than normal compilation with '-O' option. Further you can explore different optimization levels associated with '-O' option using Unix help.

This also means the compilation will take longer, as the compiler tries to apply various optimization algorithms to the code. This optimization is supposed to be conservative, in that it ensures that the code will still perform the same functionality as it did when compiled without optimization (well, unless there are bugs in our compiler). Usually we can define an optimization level by adding a number to the '-O' flag. The higher the number - the better optimized the resulting program will be, and the slower the compiler will complete the compilation.

Compiler Warnings

In general the compiler generates error messages only. However, we can instruct the compiler to give us details about warnings also. Testing and removing all warning from the program improve the quality of your program. To get the compiler to use all types of warnings, use a command as given below:

- **% cc -Wall Prog1.c -o Prog1**

Multi Source Programs

Till now you learned how to compile a single-source program but you can compile multi source programs also. Unix provides two possible ways to compile a multi-source C program. The first is to use a single command line to compile all the files. For example, you have a program whose source is found in files "main.c", "hello.c" and "world.c". you can compile as given below:

- **% cc main.c hello.c world.c -o hello_world**

This will cause the compiler to compile each of the given files separately, and then link them altogether to one executable file named "hello_world". A different way to compile multi source C programs is to divide the compilation process into two phases compiling and linking. Let's see the previous example following a different method:

cc -c main.cc

cc -c hello.c

cc -c world.c

cc main.o hello.o world.o -o hello_world

The first 3 commands have used -c option (which inform the compiler only to create an object file) and each have taken one source file, and compiled it into something called "object file", with the same names, but with a ".o" extension. The 4th command is to link all object files into one program. The linker invoked by the compiler takes all the symbols from the 3 object files, and links them together into one executable file hello_world.

System Library

For most system calls and library functions we have to include an appropriate header file. e.g., stdio.h, math.h. To use a function, ensure that you have made the required #includes in your C file. Then the function can be called as though you had defined it

yourself. To compile a program with including functions from math.h library header file, you can write the command as given below:

- **cc mathprog.c -o mathprog -lm**

The "-lm" option gives an instruction to the compiler to link the maths library with the program.

Unix System Calls

System calls are functions that a programmer can call to perform the services of the operating system. Some of them involve access to data that users must not be permitted to corrupt or even change. It is difficult to determine what is a library routine like printf() and what is a system call sleep(). To obtain information about a system call or library routine, you can read the Unix manual pages. You can read these manual pages by writing the following command for system call and to library routine:

- **% man 2 read // if read is a system call**
- **% man 3 read // if read is a library routine**

Note: All of the entries in Section 2 of the manuals are system calls, and all of the entries in Section 3 are library routines. As you know Unix system calls are interfaces, between the Unix kernel and the user programs and system calls are the only way to access kernel facilities such as the file system, the multitasking mechanisms, the inter-process communication and the socket primitives.

How does a C programmer actually issue a system call? There is no difference between a system call and any other function call. For example, the read system call might be issued like this:

nav= read(fd, buf, numbytes);

Where the implementation of the read may vary. It is better to understand that system calls are simply C subroutines. System calls takes longer time than a simple subroutine call because a system call involves a context switch between user and kernel. Generally, system calls return some value either it was successful or it failed. For example, read system call returns the number of bytes read, -1 is returned if an error occurred. When we use read () system call it should be as we have given below:

```
if ((nav = read(fd, buf, numbytes)) = = -1)
    {
     printf("Read operation failed\n");
     exit(1);
    }
```

To add more interactivity in programs you can use "perror" the library routine. It gives the description of the error condition stored in error number (errno which contains a code that indicates the reason). So, that you can handle the errors in a better way, an example is given below:

```
  if ((nav= read(fd, buf, numbytes)) == -1)
   {
    perror("read");
    exit(1);
   }
```

Output: "file does not exist" on an error.

3.3 SUMMARY

In this section we studied about the basic introduction to the C language. C was developed at AT&T's Bell Laboratory of USA in 1972 written by Dennis Ritchie. C is a general purpose, structured programming language often called a *middle level*

language. C program can be executed on platforms such as DOS, Unix etc. like DOS, Unix also stores C program in a file with extension is *.c*. This identifies it as a C program. The easiest way to enter your text is using a text editor like *vi*, *emacs* or *xedit*. We studied the procedure and command required to compile, run and debug the C program, further we have studied the difference between run time and logical errors.

The section 4 of this manual will cover the exercises on the Network programming and administration.

3.4 FURTHER READINGS

1) Brian W.Kernighan and Dennis M. Ritchie; *The C programming language* Prentice Hall.
2) W. Richard Stevens, "*UNIX Network Programming*", Prentice Hall.
3) C language, MCS-011 course material of MCA, IGNOU.
4) http://www.programmersheaven.com.
5) http://www.ee.surrey.ac.uk/Teaching/Unix/unixintro.html.

SECTION 4 NETWORK PROGRAMMING AND ADMINISTRATION EXERCISES

Structure **Page Nos.**

4.0	Introduction	48
4.1	Objectives	48
4.2	Lab Sessions	49
4.3	List of Unix Commands	52
4.4	List of TCP/IP Ports	54
3.5	Summary	58
3.6	Further Readings	58

4.0 INTRODUCTION

In the earlier sections, you studied the Unix, Linux and C language basics. This section contains more practical information to help you know best about Socket programming, it contains different lab exercises based on Unix and C language. We hope these exercises will provide you practice for socket programming. Towards the end of this section, we have given the list of Unix commands frequently required by the Unix users, further we have given a list of port numbers to indicate the TCP/IP services which will be helpful to you during socket programming.

To successfully complete this section, the learner should have the following knowledge and skills prior to starting the section. S/he must have:

- Studied the corresponding course material of BCS-052 and completed the assignments.
- Proficiency to work with Unix and Linux and C interface.
- Knowledge of networking concepts, including network operating system, client-server relationship, and local area network (LAN).

Also, to successfully complete this section, the learner should adhere to the following:

- Before attending the lab session, the learner must already have written steps/algorithms in his/her lab record. This activity should be treated as home-work that is to be done before attending the lab session.
- The learner must have already thoroughly studied the corresponding units of the course material (BCS-052) before attempting to write steps/algorithms for the problems given in a particular lab session.
- Ensure that you include comments in your lab exercises. This is a practice, which will enable others to understand your program and enable you to understand the program written by you after a long time.

4.1 OBJECTIVES

After completing this section, you should be able to:

- understand the practical issue of TCP/IP;
- know the different task performed for network administration;

- develop network applications; and
- know the TCP/IP services.

4.2 LAB SESSIONS

It contains different lab exercises based on Unix, Linux and C language to provide you hand-on experience on Unix and Linux, to sharpen your programming skills and to provide knowledge necessary for developing network applications. We hope these exercises will provide the learners, practice for socket programming. Before attending the lab session, the learner must have already written steps/algorithms in his/her lab record. This activity should be treated as homework that is to be done before attending the lab session. The learner must have already thoroughly studied the corresponding units of the course material (BCS-052) before attempting to write steps/algorithms for the problems given in a particular lab session. Ensure that you include comments in your lab exercises. This is a practice, which will enable others to understand your program and enable you to understand the program written by you after a long time.

Session 1: Unix Basics

This session is your first introduction with Unix. You can try different commands available in Unix for system and network administrator. Let us start:

Exercise 1: Run the following commands and write the use of each command:

ipconfig	ping	telnet	diskperf	netdiag
netstat	pathping	ftp/tftp	fc	sfc
nbtstat	rcp	lpr	tracert	verifier
nslookup	route	lpq	net session	drivers
nettime	rsh	chkdsk	hostname	net account

Exercise 2: Find your Ethernet physical address. Configure the IPv6 address in your eth0 interface and update the DNS server according your addresses.

Exercise 3: Write the command to remove read permission for Group and Other on the file 'green'.

Exercise 4: Write the command to add search permission for everyone for your home directory (~/).

Exercise 5: Find all files in your directory starting at ~ that were modified or accessed within the last 2 days.

Exercise 6: Find and print all files in your file space whose size is less than 50 bytes.

Session 2: Socket Setup and Data Transfer

Exercise 1: Write the code in C language for a function **swap**, which exchanges two-socket address structures. Give at least two implementations in your solutions. Also compare the solutions and explain why and how it is better.

Exercise 2: Write a new function named inet_pton_loose that handles these scenarios:

- If the address family is AF_INET and inet_pton returns 0, call inet_aton and see if it succeeds.

- If the address family is AF_INET6 and inet_pton returns 0, call inet_aton and if it succeeds, return the IPv4-mapped IPv6 address.

Exercise 3: Write the client and server programs in C language for establishing termination of connection between client and server using TCP. Assume the server can handle only one client.

Exercise 4: Write the client and server programs in C language for simple data (hello) transfer between client and server using UDP. Client will send *hello server* message to the server program. In its reply the server will send *hello client* message. The server and client programs should reside on different computers in a network.

Exercise 5: Write the Echo client and server programs in C language using UDP. The Echo clients should verify whether the text string they received from the server is the same text string that they sent or not. Also use the shutdown system call to improve the performance programs.

Session 3: Basic Linux Administration

Exercise 1: Create a new user guest1 in the Guest Group using GUI tool in Linux. Now add another new user in a different group than the group of guest. Assume both users wants to write on a same text file "myfile", create an optimal file permission for this text file.

Exercise 2: When new media (like a flash drive, an external hard drive, an SD card, etc) is added to a system, it must be mounted to the file system. Show how Mounting/Unmounting is done in Linux File Systems with read only permission.

Exercise 3: Write a command to enable authentication for single-user mode on Linux.

Exercise 4: Write a command to disable Interactive Hotkey Startup at Boot in Linux.

Exercise 5: Setup a Network Connection in Linux. Configure it for Wireless networking. Once internet connection is established use **ss, netstat, tcptrack, Iftop** commands and check the results.

Session 4: Advanced Data Transfer

Exercise 1: Write the client and server programs in C language for connectionless communication between two different Unix computers in the same TCP/IP network. The server process receive a byte from the client process should and send back an should acknowledgement to the client process.

Exercise 2: Write the client and server programs in C language, where the server can exchange text with many client processes. A client process starts the communication with an input "start". After this the client process waits for the answer from the server. If server permits, it can further send any text message (with restriction of not more than 1000 words in a day). The communication goes on in this way until the client process sends the message "stop" to the server.

Session 5: Flow and Error Control

Exercise 1: Assume Client program is running on Machine A and server program on B. Write a program to ensure that the data received by server on machine B is the same data which was sent by the client program on machine A. Implement the scheme through which you can recover/calculate the lost

data. Write the client and server programs in C language for showing the result.

Exercise 2: Write programs in C language for implementing the sliding window protocol of window size 5.

Session 6: Routing

Exercise 1: Write the client and server programs in C language for implementing the broadcasting in the local network.

Exercise 2: Write the program in C language for implementing the IP Routing protocol using Address tables.

Session 7: Utility Development

Exercise 1: Write the program in C language for implementing the utility similar to "Ping".

Help: Ping is actually an acronym for the words 'Packet INternet Groper'. The Ping utility is essentially a system administrator's tool that is used to see if a computer is operating and also to see if network connections are intact. Ping uses the Internet Control Message Protocol (ICMP) echo function, which is detailed in RFC 792.

Exercise 2: Write the program in C language for implementing address resolution using DNS tables.

Session 8: Advance Linux Administration

Exercise 1: Configure and launch an FTP server than install and test an FTP client in Linux.

Exercise 2: Configure a remote server and transfer a Directory to Remote Server.

Exercise 3: Create and Configure samba Server in Linux. Also, transfer files from client side.

Exercise 4: At the root level, check the status of services apmd, sendmail, and cups. Use chkconfig to turn cups off in runlevels 2, 3, 4, and 5. Use the run level editor GUI to turn 'cups' back on.

Exercise 5: Assume a user is running HTTP at the port number 80. You are required to change port number of HTTP for your client to 8080.

Session 9: Mail Transfer

Exercise 1: Write the program in C language for implementing the client for Simple mail transfer protocol.

Exercise 2: Write the program in C language for implementing the server for Simple mail transfer protocol. Where Server can handle maximum 5 clients concurrently.

Session 10: Client/Server Computing

Exercise 1: Write the client and server programs in C language, where client will send a file containing a C-program, server will compile and executes the file given by the client and if error occurs during compilation or execution server will send back the appropriate message to the client otherwise server will send the executable file to the client.

4.3 LIST OF UNIX COMMANDS

In this appendix, we have summarized some of the basic Unix commands you need to get started. For further details on UNIX commands use the **man** command.

Setup and Status Commands

logout	end your UNIX session
passwd	change password by prompting for old and new passwords
stty	set terminal options
date	display or set the date
finger	display information about users
ps	display information about processes
env	display or change current environment
set	C shell command to set shell variables
alias	C shell command to define command abbreviations
history	C shell command to display recent commands

File and Directory Commands

cat	concatenate and display file(s)
more	paginator - allows you to browse through a text file
less	more versatile paginator than more
mv	move or rename files
cp	copy files
rm	remove files
ls	list contents of directory
mkdir	make a directory
rmdir	remove a directory
cd	change working directory
pwd	print working directory name
du	summarize disk usage
chmod	change mode (access permissions) of a file or directory
file	determine the type of file
quota -v	displays current disk usage for this account
ls -a	list all files and directories
cd ~	change to home-directory
cd ..	change to parent directory
head file	display the first few lines of a file
tail file	display the last few lines of a file
grep 'keyword' file	search a file for keywords
command > file	redirect standard output to a file

command >> file	append standard output to a file
command < file	redirect standard input from a file
command1 | command2	pipe the output of command1 to the input of command2
cat file1 file2 > file0	concatenate file1 and file2 to file0
sort	sort data

Editing Tools

pico	simple text editor
vi	screen oriented (visual) display editor
diff	show differences between the contents of files
grep	search a file for a pattern
sort	sort and collate lines of a file (only works on one file at a time)
wc	count lines, words, and characters in a file
Look	look up specified words in the system dictionary
awk	pattern scanning and processing language
gnuemacs	advanced text editor

Formatting and Printing Commands

lpq	view printer queue
lpr	send file to printer queue to be printed
Lprm	remove job from printer spooling queue
enscript	converts text files to POSTSCRIPT format for printing
lprloc	locations & names of printers, prices per page
pacinfo	current billing info for this account

Program Controls, Pipes, and Filters

CTRL-C	interrupt current process or command
CTRL-D	generate end-of-file character
CTRL-S	stop flow of output to screen
CTRL-Q	resume flow of output to screen
CTRL-Z	suspend current process or command
jobs	lists background jobs
bg	run a current or specified job in the background
fg	bring the current or specified job to the foreground
fg %1	foreground job number 1
!!	repeat entire last command line
!$	repeat last word of last command line
sleep	suspend execution for an interval
kill	terminate a process

nice	run a command at low priority
renice	alter priority of running process
&	run process in background when placed at end of command line
>	redirect the output of a command into a file
>>	redirect and append the output of a command to the end of a file
<	redirect a file to the input of a command
>&	redirect standard output and standard error of a command into a file (C shell only)
|	pipe the output of one command into another
kill %1	kill job number 1
ps	list current processes
kill 26152	kill process number 26152
who	list users currently logged in
a2ps -Pprinter textfile	print text file to named printer
lpr -Pprinter psfile	print postscript file to named printer
*	match any number of characters
?	match one character
man command	read the online manual page for a command
whatis command	brief description of a command
apropos keyword	match commands with keyword in their man pages
ls -lag	list access rights for all files
command &	run command in background

Other Tools and Applications

pine	electronic mail
bc	desk calculator
man	print UNIX manual page to screen
elm	another electronic mail program

4.4 LIST OF TCP/IP PORTS

The Internet Assigned Numbers Authority (IANA) specifies TCP/IP port numbers. As we discussed earlier in the course, all the port numbers are divided into three categories based on port number ranges: the Well Known Ports, the Registered Ports, and the Dynamic and/or Private Ports. The well known ports are those from 0 through 1023, registered ports are those from 1024 through 49151 and the Dynamic and/or Private Ports are those from 49152 through 65535. These ports are not used by any defined application. The following tables indicate the official (if the application-port combination is in the Internet Assigned Numbers Authority list) well-known and registered ports numbers.

Well-Known Ports (0 to 1023)

Port number	Description
0	Reserved; do not use
1	TCPMUX
5	RJE (Remote Job Entry)
7	ECHO protocol
9	DISCARD protocol
13	DAYTIME protocol
17	QOTD (Quote of the Day) protocol
18	Message Send Protocol
19	CHARGEN (Character Generator) protocol
20	FTP - data port
21	FTP - control (command) port
22	SSH (Secure Shell) - used for secure logins, file transfers and port forwarding
23	Telnet protocol - unencrypted text communications
25	SMTP - used for sending E-mails
37	TIME protocol
38	Route Access Protocol
39	Resource Location Protocol
41	Graphics
42	Host Name Server
49	TACACS Login Host protocol
53	DNS (Domain Name Server)
67	BOOTP (BootStrap Protocol) server; also used by DHCP (Dynamic Host Configuration Protocol)
68	BOOTP client; also used by DHCP
69	TFTP (Trivial File Transfer Protocol)
70	Gopher protocol
79	Finger protocol
80	HTTP (Hyper Text Transfer Protocol) - used for transferring web pages
88	Kerberos - authenticating agent
109	POP2 (Post Office Protocol version 2) - used for retrieving E-mails
110	POP3 (Post Office Protocol version 3) - used for retrieving E-mails
113	Ident - old server identification system, still used by IRC servers to identify its users

118	SQL Services
119	NNTP (Network News Transfer Protocol) - used for retrieving newsgroups messages
123	NTP (Network Time Protocol) - used for time synchronization
137	NetBIOS NetBIOS Name Service
138	NetBIOS NetBIOS Datagram Service
139	NetBIOS NetBIOS Session Service
143	IMAP4 (Internet Message Access Protocol 4) - used for retrieving E-mails
156	SQL Service
161	SNMP (Simple Network Management Protocol)
162	SNMPTRAP
179	BGP (Border Gateway Protocol)
194	IRC (Internet Relay Chat)
213	IPX
369	Rpc2portmap
389	LDAP (Lightweight Directory Access Protocol)
401	UPS Uninterruptible Power Supply
427	SLP (Service Location Protocol)
443	HTTPS - HTTP Protocol over TLS/SSLSSL (encrypted transmission)
445	Microsoft-DS (Active Directory, Windows shares, Sasser-worm, Agobot, Zobotworm)
445	Microsoft-DS SMB file sharing
464	Kpasswd
465	SMTP over SSL - CONFLICT with registered Cisco protocol
500	Isakmp
514	syslog protocol - used for system logging
530	Rpc
540	UUCP (Unix-to-Unix Copy Protocol)
542	commerce (Commerce Applications) (RFC maintained by: Randy Epstein [repstein at host.net])
554	RTSP (Real Time Streaming Protocol)
563	Nntp protocol over TLS/SSL (NNTPS)
587	email message submission (SMTP) (RFC 2476)
591	FileMaker 6.0 Web Sharing (HTTP Alternate, see port 80)
593	HTTP RPC Ep Map
636	LDAP over SSL (encrypted transmission)
666	id Software's Doom multiplayer game played over TCP (666 is a reference to the Number of the Beast)
691	MS Exchange Routing

873	rsync File synchronisation protocol
989	Ftp Protocol (data) over TLS/SSL
990	Ftp Protocol (control) over TLS/SSL
992	Telnet protocol over TLS/SSL
993	IMAP4 over SSL (encrypted transmission)
995	POP3 over SSL (encrypted transmission)

Registered Ports (1024 – 49151)

Port	Description
1080	SOCKS proxy
1099	RMI Registry
1099	RMI Registry
1194	OpenVPN
1198	The cajo project Free dynamic transparent distributed computing in Java
1214	Kazaa
1223	TGP: "TrulyGlobal Protocol" aka "The Gur Protocol"
1337	menandmice.com DNS (not to be confused with standard DNS port). Often used on compromised/infected computers - "1337" a "Leet speak" version of "Elite". See unregistered use below.
1352	IBM Lotus Notes/Domino RCP
1387	Computer Aided Design Software Inc LM (cadsi-lm)
1387	Computer Aided Design Software Inc LM (cadsi-lm)
1414	IBM MQSeries
1433	Microsoft SQL database system
1434	Microsoft SQL Monitor
1434	Microsoft SQL Monitor
1494	Citrix MetaFrame ICA Client
1547	Laplink
1547	Laplink
1723	Microsoft PPTP VPN
1723	Microsoft PPTP VPN
1863	MSN Messenger
1900	Microsoft SSDP Enables discovery of UPnP devices
1935	Macromedia Flash Communications Server MX
1984	Big Brother - network monitoring tool
2000	Cisco SCCP (Skinny)
2000	Cisco SCCP (Skinny)
2427	Cisco MGCP
2809	IBM WebSphere Application Server Node Agent

2967	Symantec AntiVirus Corporate Edition
3050	gds_db
3050	gds_db
3074	xbox live
3128	HTTP used by web caches and the default port for the Squid cache
3306	MySQL Database system
3389	Microsoft Terminal Server (RDP) officially registered as Windows Based Terminal (WBT)
3396	Novell NDPS Printer Agent
3689	DAAP Digital Audio Access Protocol used by Apple's ITunes
3690	Subversion version control system
3784	Ventrilo VoIP program used by Ventrilo
3785	Ventrilo VoIP program used by Ventrilo
6891-6900	MSN Messenger (File transfer)
6901	MSN Messenger (Voice)
11371	OpenPGP HTTP Keyserver

4.5 SUMMARY

In this section, we have given different lab exercises based on Unix, Linux and C language to provide to practical experience of socket programming. These exercises will help to develop/manage different network application by your own. Further this section covered the list of Unix commands frequently required by the Unix users, and a list of port numbers to indicate the TCP/IP services which will helpful to you during socket programming.

4.6 FURTHER READINGS

1) Brian W.Kernighan and Dennis M. Ritchie; *The C programming language* Prentice Hall.

2) Douglas E. Comer and David L. Stevens, "*Internetworking with TCP/IP. Vol.3: Client-server programming and applications BSD socket version*", Prentice Hall.

3) W. Richard Stevens, "*TCP/IP Illustrated. Vol. 1: The protocols*", Addison Wesley.

4) W. Richard Stevens, "*UNIX Network Programming*", Prentice Hall.

5) http://www.linux.org

6) UNIX and Linux System Administration Handbook, Prentice Hall