1.1 INTRODUCTION

UNIX is a most popular multitasking operating system ideally suited for both standalone and distributed processing configurations. It has powerful yet easy-to-use hardware interface mechanisms and provides an excellent means of organizing and storing files on a variety of media including magnetic disks, magnetic tapes, and optical disks. UNIX systems are available to support individual users, small groups, or entire departments on a wide range of processing platforms. Most discussions on UNIX begin with history of UNIX telling about the growth of UNIX.

1.2 HISTORY OF UNIX

UNIX is a registered trademark of AT&T's UNIX System Laboratories and is the common name for a family of interactive, multi-user operating systems. UNIX is available in one version or another for virtually any computer, ranging from desktop personal computers and workstations to the most powerful supercomputers.

UNIX was originally developed on a Digital Equipment Corporation (DEC) PDP-7 in 1969 by Ken Thompson and Dennis Ritchie – employees of AT&T's Bell Laboratories. Their objective was to create a flexible, interactive processing environment suitable for programming and computer systems research projects. UNIX was ported to the DEC PDP-11 during 1970-71, and by the latter half of 1971 it was busy supporting its first real world application, a text processing and typesetting system used by Bell Labs patent department. UNIX turned out to be a success at something other than arcane computer research.

Initial work on the programming language that would become C started in 1971, and the operating system kernel was rewritten in C during 1973. Prior to that time UNIX was written entirely in assembly language. The popularity and application of UNIX increased so much that an internal systems group was formed to support the growing number of UNIX installations within Bell Labs, other AT&T departments, and the local Bell operating companies. Although at the time, AT&T was prohibited from marketing computer products. UNIX was distributed without support to a number of universities for educational purposes. It was also licensed to commercial institutions during the 1970s. During the late 1970s and early 1980s several versions of UNIX internal to UNIX System V has seen several upgrades, feature additions, and a continued rise in popularity since its initial release. Several other versions of UNIX, rooted in the educational and commercial releases of the 1970s, have also taken hold in the marketplace. The popularity of UNIX is attributed, in part, to a growing number of systems professionals who were educated in a UNIX environment. It is also attributed, in part, to its portability. About 95 percent of UNIX is written in the C programming language and assembly language is only used for hardware-dependent routines or to maximize performance. This makes UNIX easier to port than many other operating systems. In addition, vendors are motivated to port UNIX because it provides a rich and consistent environment for applications software development on a variety of different hardware platforms. AT&T and Bell labs jointly released UNIX System III. System V, which evolved out of System III, was released commercially and officially supported in 1983. At the time, the number of UNIX installations was approaching 100.000 worldwide.

1.2.1 THE UNIX FAMILY

System V is the UNIX version officially supported by AT&T's UNIX System Laboratories (USL). System V is distributed by AT&T for both microcomputers and minicomputers. Other licensed vendors distribute System V for everything from desktop machines to supercomputers. USL also produces the System V Interface Definition (SVID), which determines how well any version of UNIX complies with System V standards. Depending on the software manufacturer and the hardware platform, different versions of System V may vary. However, if a given UNIX implementation complies with the SVID, it is more receptive to applications developed on different platforms.

Berkeley Software Distribution (BSD) was developed at the University of California at Berkeley and originally released in 1977. BSD evolved from the UNIX Time-Sharing System Sixth Edition – a predecessor of System III. BSD was one of the first versions of UNIX to provide demand paging, virtual memory, and support for high-speed local area networks. In 1980, BSD 4.0 became the standard UNIX platform for the Defense advanced Research Projects Agency (DARPA) network development and the latest release, BSD 4.3, includes support for ISO/OSI networks. BSD is widely used on minicomputers.

XENIX is a version of UNIX produced and distributed explicitly for personal computers (PCs). Originally developed by Microsoft and released in 1986, XENIX is currently distributed by Santa Cruz Operation (SCO) and Interactive for computers based on Intel's 80X86 line of microprocessors. Since XENIX was designed for PCs, it is smaller and generally provides better performance than PC versions of System V. However, the performance gap between XENIX and System V (for PCs) will close as faster and more capable microprocessors, such as Intel's 80486 and 80586, reach the marketplace.

One of AT&T's objectives for System V was to set a standard for all versions of UNIX to follow. A controversy and something of a struggle within the UNIX community ensued following the release of System V. There are several competing standards exist and are under development. Each of the major players in UNIX marketplace believe their version offers something unique, something that should be preserved. AT&T has since modified its strategy. In conjunction with AT&T, Sun Microsystems developed SunOS, a version of UNIX for its workstations and file servers. In 1988 AT&T and Sun Microsystems jointly developed System V Release 4 (SVR4)which combines features from both System V and BSD incorporates important features from SunOS, BSD and XENIX. System V Release 4 also includes a C compiler that conforms to the international standard.

In 1993 Novell bought UNIX from AT&T. UNIX is the trademark of Novell. There are many variants or flavours of UNIX. Some of them are HP-UX(Hewlett -Packard), Solaris (Sunsoft), SVR4(AT&T), LINUX.

1.3 WHAT IS LINUX?

Linux is a true 32-bit operating system that runs on a variety of different platforms, including Intel, Sparc, Alpha, and Power-PC (on some of these platforms, such as Alpha, Linux is actually 64-bit). There are other ports available as well. Linux was first developed back in the early 1990s, by a young Finnish then-university student named Linus Torvalds. Linus had a "state-of-the-art" 386 box at home and decided to write an alternative to the 286-based Minix system (a small UNIX-like implementation primarily used in operating systems classes), to take advantage of the extra instruction set available on the then-new chip, and began to write a small bare-bones kernel. Eventually he announced his little project in the USENET group comp.os.minix, asking for interested parties to take a look and perhaps contribute to the project. The results have been phenomenal!

The interesting thing about Linux is, it is completely free! Linus decided to adopt the GNU Copyleft license of the Free Software Foundation, which means that the code is protected by a copyright -- but protected in that it must always be available to others.

Free means free -- you can get it for free, use it for free, and you are even free to sell it for a profit (this isn't as strange as it sounds; several organizations, including Red Hat, have packaged up the standard Linux kernel, a collection of GNU utilities, and put their own "flavour" of included applications, and sell them as distributions. Some common and popular distributions are Slackware, Red Hat, SuSe, and Debian)! The great thing is, you have access to source code which means you can customize the operating systems to your own needs, not those of the "target market" of most commercial vendors.

Linux can be considered a full-blown implementation of UNIX. However, it can not be called "Unix"; not because of incompatibilities or lack of functionality, but because the word "Unix" is a registered trademark owned by AT&T, and the use of the word is only allowable by license agreement.

Linux is every bit as supported, as reliable, and as viable as any other operating system solution (well, in my opinion, quite a bit more so!). However, due to its origin, the philosophy behind it, and the lack of a multi-million dollar marketing campaign promoting it, there are lot of myths about it. People have a lot to learn about this wonderful OS!

Features of LINUX

- It's free!
- Open Source (modifiability, extensibility, ...)
- Works on several platforms
- Robustness (after several revisions, and several people working on it)
- Widespread Usage
- Compatibility with several other platforms

1.4 UNIX System

The UNIX Operating System is described as a multi-programming, time-sharing, multi-tasking system. Let us take a brief look at these concepts.

Multi-programming
UNIX allows many programs to be executed simultaneously. This feature is called multi-programming. An executing instance of a program is called process.

Time-sharing
Multi-programming is made possible on the UNIX system by the concept of time-sharing. Since there is only a single CPU to take care of the various programs to be executed, the programs are queued and CPU time is shared among them. Each program is attended for a specific period of time, and then put back on the queue to wait its turn again as the next program in the queue is taken up.

Multi-tasking
A program in UNIX is broken down into tasks, where each task can be something like reading from or writing to the disk, or waiting for input from the user, and so on. When a program is waiting for completion of a task, the CPU, rather than waste time, starts executing the next task. Therefore, while a program is waiting for input from the user, another program could be reading from the hard disk.

1.5 HARDWARE REQUIREMENTS OF UNIX

There are a few basic requirements to run UNIX on your computer, your computer must fulfill in terms of processing power, memory size and disk space. Ideally, to run UNIX, you need a 16-bit microprocessor (80286 or preferable 80386/80486) and about 3 Megabytes of Random Access Memory (RAM) for optimum performance. The hard-disk space should be about 80 Megabytes or more. Besides this hardware, UNIX requires a considerable amount of human support. Every UNIX installation generally has a Human Manager called the System Administrator. The System Administrators tasks include supervision and control of UNIX on the installation.

1.6 SALIENT FEATURES OF THE UNIX OPERATING SYSTEM

The UNIX Operating System is available on machines with a wide range of computing power, from microcomputers to mainframes, and on different manufacturers' machines. No other system can make this claim. The popularity and success of UNIX is due to the following reasons:

Portability: The system is written in a high-level language making it easier to read, understand, change and, therefore, move to other machines. The code can be changed and compiled on a new machine. Customers can then choose from a wide variety of hardware vendors without being locked in' with a particular vendor.

Machine-independence: The system hides the machine architecture from the user, making it easier to write applications that can run on micros, minis and mainframes.

Multi-user Operations: UNIX is a multi-user system designed to support a group of users simultaneously. The system allows for the sharing of processing power and peripheral resources, while at the same time providing excellent security features.

Hierarchical File System: UNIX uses a hierarchical file structure to store information. This structure has the maximum flexibility in grouping information in a way that reflects its natural state. It allows for easy maintenance and efficient implementation.

UNIX Shell: UNIX has a simple user interface called the shell that has the power to provide the services that the user wants. It protects the user from having to know the intricate hardware details.

Pipes and-Filters: UNIX has facilities called pipes and filters which permit the user to create complex programs from simple programs.

Utilities: UNIX has over 400 utility programs for various functions. New utilities can be built effortlessly by combining existing utilities.

Background Processing: UNIX has a facility by which the user can start a process and then proceed to work on other processes while the system runs the first process in the background and the second process in the foreground. Background processing helps the user in effective utilization of time.

Software Development Tools: UNIX offers an excellent variety of tools for software development for all phases, from program editing to maintenance of software.

Maturity: UNIX is a time-tested operating system. It offers a bug-free environment and a high level of reliability.

Modular structure of a UNIX : The modular design of UNIX is one of the most outstanding aspects of the operating system. UNIX consists of number programs each program fits into a slot provided for its functioning. In case a particular program or module is not included, the rest of the operating system is not impaired. Thus in a way, every persons UNIX is unique. The user can add only those modules that he requires, thus saving on cost and valuable computer resources. UNIX versions thus are almost made to suit the user's needs. The rationale is clear as crystal. Modules of additional utilities can be patched on to the operating system as and when the need arises and the newly patched module harmonizes with the operating system like a long lost brother.

The UNIX system supports a wide variety of languages--C, FORTRAN, BASIC, PASCAL, Ada, COBOL, Lisp and Prolog.

1.7 BASIC CONCEPTS OF A MULTI-USER SYSTEM

A multi-user system consists of a single computer with several terminals attached to it. This computer can be a PC-AT, a mini or a mainframe. Various users can work on this machine through the attached terminals. These terminals can be of two types: smart and dumb.

A dumb terminal consists of a VDU and a keyboard. It has no CPU of its own, and the processing is done only by the central unit.

A smart terminal has its own CPU and peripherals, and can work independent of the central unit. The advantage is that other operating systems can be loaded onto its own hard disk. Connection to UNIX can be established when required.

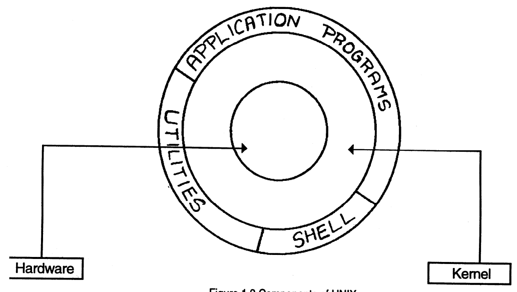


Figure 1.3 Components of UNIX

Figure 1.1 Architecture of a UNIX system.

1.8 ARCHITECTURE OR STRUCTURE OF THE UNIX SYSTEM

The UNIX system is divided into three main entities

i. Kernel

The core of the UNIX system is the Kernel – the operating system program. The Kernel controls the computer's resources, allotting them to different users and to different tasks. It interacts directly with the hardware, thus making the programs easy to write and portable across different platforms of hardware. Since the kernel communicates directly with the hardware, parts of the kernel must be customized to each system's hardware features. However, the kernel does not deal directly with a user. Instead, it starts up a separate, interactive program, called the shell, for each user.

ii. The Shell

It is the 'Command Interpreter' of the operating system. It accepts commands from the user and analyzes and interprets these commands. We know that most UNIX commands are executable C programs. The Shell interprets the users command and starts executable C programs. The Shell interprets the user commands, if the command is correct it activates the corresponding executable file and starts executing it. It then interacts with the kernel to perform the actual action of data transfer, which finally leads to the output that is displayed on the terminal. Hence the Shell acts as a middleman between the Kernel and the user of the operating system. The Kernel is basically the monarch, but you generally deal with its emissary, the Shell. In UNIX, every user who logs into the system gets his own copy of the Shell. This means that, at a particular instant, there may be several copies of the Shell running on the system with a minimum of one Shell per user.

Features of the shell
The shell is a utility program that comes with the UNIX system. However, it plays a very important role. h acts as the command interpreter for the kernel and is the interface between the user and the kernel. The UNIX system's shell includes the following major features:

Interactive Processing: Communication between the users and the system takes the form of an interactive dialogue with the shell.

Background Processing: Time-consuming, non-interactive tasks can proceed while the user continues with other processing. Therefore, the system can perform many different tasks at the same time on behalf of a single-user.

Input/Output Redirection: Programs designed to interact with a user can easily be instructed to take their input from another source, such as a file, and send their output to another destination, such as a printer.

Pipes: Programs that perform simple functions can easily be connected to perform more complex functions, minimizing the need to develop new programs.

Shell Scripts: A frequently used sequence of shell commands can be stored in a file. The name of the file can later be used to execute the stored sequence with a single command.

Shell Variables: The user can control the behavior of the shell, as well as other programs and utilities, by storing data in variables.

Programming Language Constructs: The shell includes features that allow it to be used as a programming language. These features can be used to build shell scripts that perform complex operations.

You will be looking at all these features in the chapters to come.

There are three shells mainly used with UNIX.

1. The Bourne Shell

This is one of the most widely used shells in the UNIX world. It was developed by Steve Bourne of AT&T Bell Laboratories (where else) in the late 1970s. It is the primary UNIX command interpreter and comes along with every UNIX system. The ubiquitous “dollar prompt” on UNIX installations is the trademark of the Bourne Shell.

2. The C Shell

Bill Joy, who is by now, known to you as the “Joy of UNIX” was the creator. He designed and developed this shell at the University of California at Berkeley when he was a graduate student there. The C Shell is the default Shell in the Berkeley version of UNIX and is immensely popular with UNIX programmers and university researchers.

The C Shell has a few principal advantages over the Bourne shell.

a. **A history mechanism:** The C shell remembers the commands that the user types and allows him to recall them without having to retype them. This is a great benefit, for, as you will discover, UNIX commands are often very long and have a multitude of options. Even the slightest typing mistake or a control key pressed during typing the command, and UNIX rejects it in an instant. So this mechanism is not as trivial as it seems.

b. **Aliasing:** The C shell permits you to call frequently used commands by your own formulated abbreviations. This too proves very useful at the command line. This is a type of "macro" facility that is available at the command line.

3. The Korn Shell

Developed by David Korn of AT&T, this shell was designed to be much bigger than the Bourne Shell and includes several features that make it more superior. It can completely replace the Bourne Shell in a system and has several more functions that are built into the shell making it more efficient and versatile. The Korn Shell includes all the enhancements in the C shell, like command history and aliasing, and offers a few more features itself. This shell is noticeably more efficient than the Bourne Shell.

III UNIX TOOLS AND APPLICATIONS

The outermost layer of the UNIX operating system is its tools and applications. Tools vary from one implementation of UNIX to another. Some versions of UNIX are decked with more than 400 tools and applications. These tools can be invoked from the command line itself and helps to perform the day-to-day as well as complex tasks of the system. These are placed one level above the shell and can be expanded and patched as required by the user. Unlike the Shell and the Kernel, these tools are not mandatory, hence different implementations of UNIX have varying number of tools and applications available. Apart from the utilities that are provided as part of the UNIX operating system, more than a thousand UNIX based application programs, like database management systems, word processors, accounting software and language processors, are available from independent software vendors

1.9 LOGGING IN

Once a connection is established, the system issues the login prompt. A typical login prompt looks like the following:

```
login: _
```

The prompt may be preceded by a greeting, a message, a set of instructions, a warning against tampering, and so forth. In any case, the login prompt will be the first item displayed on the terminal. In response, you should type your login name and press the carriage return key. After that, the system issues a password prompt similar to the following:

```
Password: _
```

You should respond with your assigned password and press the carriage return key.
The following example depict a typical login sequence. For this example, suppose your login name is Nachappa and your password is Manju123. The system issues the login prompt and accepts your login name:

```
login:  Nachappa<Enter>
```

Bear in mind that UNIX distinguishes between uppercase and lowercase. You must type your login name exactly as the administrator specified.

Example Nachappa, NACHAPPA, and nachappa are all distinct login names.

Following the login prompt and your reply, the system issues the password prompt:

```
login: Nachappa<Enter>
Password:
```

In response, type your password. Like the login name, your password must be typed exactly as the system administrator specified. The correct response in this example is Manju123 followed by a carriage return. In the following exampled the system administrator posted a notice that the system will be down for maintenance. Also note that there is no news, but the user has mail:

```
AT&T UNIX System V Release 4.0
Copyright ( c ) 1984, 1986, 1987, 1988, 1989, 1990 AT&T
All.Rights Reserved
last login: Sun Jan 26 17:56:19 on tty01
The system will be down today from 5:45 PM to 6:45 PM for routine
maintenance.
you have mail.
$ _
```

the system issues the command prompt following the welcome screen. The command prompt indicates that the shell is ready and waiting. The default prompt is the dollar sign character ($).

1.10 LOGGING OUT

Log out whenever you are not actively using the system, especially if you leave the area where the terminal is located. Logging out is accomplished by issuing the exit command or by typing <CTRL-d> (<CTRL-d> is not visible as it is typed – it is depicted in the examples for the benefit of the reader). Either method is generally acceptable.

exit is shell command so it must be followed by a carriage return:

```
$ exit<Enter>
```

<CTRL-d> is handled by the terminal line driver so it need not be followed by a carriage return:

```
$ <CTRL-d>
```

Unless you break the connection to the system after logging out, the system re-initiates the login sequence and issues a login prompt:

```
$ exit<Enter>

Welcome to the AT&T 386 UNIX System
System name: blackhole

login:  _
```

1.11 CHANGING YOUR PASSWORD

The system administrator assigns a password when your account is created. Thereafter, maintaining your password is your responsibility. In general, you can change your password whenever you wish with the passwd command as follows

```
$ passwd<Enter>
passwd:  Changing password for Nachappa
Old password:
New password:
Re-enter new password:
$ _
```

You must first provide your existing password. You must also type your new password twice. If it doesn't meet standards or you fail to type it the same way both times, passwd halts without changing your password. You will be allowed several attempts to get the job done:

1.12 Types of Users in UNIX

System Administrator

The System Administrator (SA) is primarily responsible for the smooth operation of the system. It is the SA job to switch on the system console. The SA also creates users and groups of users for the system, and takes backup of data to prevent loss of data due to system breakdown.

File Owner

The user who creates a file is said to be the owner of that file. The owner of a file can perform any operation on that file— copying, deleting and editing.

Group Owner

Consider the following situation:

A project team of five people of the Smartest Consultants Inc. is working on a software development project PROJECT-ABC for a private detective agency. The team is headed by an analyst. The other members are all programmers. The team is working on a UNIX system. Each programmer has been given a few programs to develop. The data provided by the detective agency is of a highly confidential nature, are so the data file has been created in the analyst's HOME directory. One programmer may have to link (join) program to another programmer's program in order to test the program. In this situation, each programmer is the File Owner of his or her own program files. Each program, however, would also belong to the other programmers, so that they could use the programs for linking. The project team (of five users) is said to the Group Owner for the file. In UNIX, it is possible to define the users who will belong to a group. A group users are also given a name, just as a user is given a name.

Other Users

In the example of the Smartest Consultants Inc., all users of the system who are not members of the group PROJECT-ABC are referred to as Other Users for all files that belong to that group. Other Users are users who do not need the data or programs of the Group. For example, people of the Accounts department who use the system to generate reports, or data entry operators entering data for payroll can be referred to as Other Users for the files that belong to the group PROJECT-ABC.

1.13 UNIX COMMAND FORMAT

The syntax of the UNIX command is **command -options arguments**.

Command is the name of the command , Option is a special argument which is preceded by a – sign to differentiate from the filename. Arguments are the filenames or expressions. Arguments are sometimes optional.
For example **ls –l hkg**. In this ls is the command which is used for long listing of files in the directory name hkg. l is the option which says long listing of files and hkg is the argument. Which is the name of the directory. The unix commands are case sensitive and it should be entered with lower case.

1.14 Types of UNIX commands.

Broadly there are two types of commands.

1 External command : It is the one that exists independently as a file. When an external command is issued, shell searches for this command which is stored as a file first in the current working directory. If it is not found, it searches in all directories which is listed in shell's PATH variable.
For example ls, passwd are external commands.
2. Internal commands :
These commands do not exist independently. They are the part of the shell. For example echo, cd is an internal command.

1.15 SOME BASIC COMMNDS.

echo

This command is usually used in shell scripts to display messages on the terminal.. It takes zero, one or more arguments. Arguments may be given either as a series of individual symbols or as a string within a pair of double quotes.

```
$echo

$
```

In the above example a blank line is displayed if echo is given without arguments.

```
$echo M.N.     Nachappa
M.N. Nachappa
```

If there are extra spaces between the arguments, they are adjusted and the output is printed in a standard form, with just one blank between the different arguments

```
 $ echo  "M.N.                 Nachappa"
  M.N.                 Nachappa
$
```

However, when the message is given in the form of a string argument, that message is printed as it is. In other words, when a string is given as an argument, it is printed without the adjustment of the blanks

```
$ echo name is : $name
name is : Manjunath
```

if an evaluatable argument is given, it is first evaluated and its value is printed along with the other arguments. This command is chiefly used to write output statements in the shell programs.

Date

All UNIX systems maintain the current date and time. Users can display the current date and time with the date command.

```
        $ date <Enter>
        Sat Feb 18 11:30:051ST 1989
        $
```

The tty command

The user can know name of his device file on which he is working by using tty command.

```
$tty
/dev/tty03
$
```

tput clear

The command tput clear clears the Standard output device, the screen, and positions the cursor on the top left comer of the screen

Who

The who command is used to display the names of all users who are currently logged in.

```
 $ who <Enter>
ram tty02  Feb   2     10:30
krish  tty03   Feb   2      9:40
anita  tty01   Feb    2      9:00
john  tty07  Feb    2     9:00
$
```

The who am i command displays the name of current users.

```
$ who am i <Enter>
ramesh tty02  Feb   2     10:30
$
```

The output of who command also consists of the terminal filename and the date and time the user logged in.

cat command

cat command is used to create files. Type cat command followed by > filename.

```
$cat  > sample
lion
tiger
deer
<ctrl> d
$
```

In the above example sample file is created with the typed text. To end the file we press control d to indicate end of file.

cat command When followed by a filename, all the lines in the file are displayed till the end of file.

```
$cat sample
lion
tiger
deer
$
```

cat command without the filename, however, the cat command takes its input from the standard input file as depicted:

```
$ cat
```

The cat command waits for input from the keyboard. As a user enters characters from the keyboard, they are displayed on the screen as shown:

```
$ cat
This is a chair
```

The user can keep on entering lines. To indicate the end of input, the user has to press Ctrl and d. The $ prompt then appears on the VDU.
Not all commands use the standard input file for input.

The cat command can also display more than one file as shown in the following command:

```
$ cat data1 data2 <Enter>
A sample file
Another sample file          (contents of data2)
```

The cat command can also be used to concatenate files.

$ cat file1 file2 fil3 > newfile

In the above command the create a newfile which contains contents of file file3 file3.. Suppose if newfile alreafy existing then it previous contents are lost. If you want to retain the previous contents of newfile then you can append data of file1 file2 file3 by using other form cat command.

$ cat file1 file2 file3 >> newfile

Here data file1 file2 file3 is appended to newfile.

touch command is used to create empty files.

$ touch ex1

ex1 file is created which is empty.

$touch x1 x2 x3 x4 x5 x6

Here x1 , x2, x3, x4, x5 and x6 files are created which are empty.

Chapter 2

2.1 INTRODUCTION

A file is a collection of information, which can be data, an application, documents; in fact, under UNIX, a file can contain anything. The file system refers to the way in which UNIX implements files and directories. In UNIX, a file system has the following features,

1. hierarchical structure (support for directories)
2. files are expandable (may grow as required)
3. files are treated as byte streams (can contain any characters)
4. security rights are associated with files and directories
5. read/write/execute privileges for owner/group/others
6. files may be shared (concurrent access)
7. hardware devices are treated just like files

2.2 THE DIRECTORY STRUCTURE

The UNIX file system is the means of storing and organizing programs and data as well as the interface to the computer's hardware devices. Unix uses a hierarchical directory structure where directories are organized into a tree. The directory at the root of the tree is named root and is spelled / (just a single slash character). Root lists directories: its subdirectories, which have root as their parent directory. Each subdirectory lists its own subdirectories, etc., creating a branching structure.

A file system contains three types of file, ordinary files, special files, and directories. Ordinary files include text files, data files and programs. Special files are points of interface to the computer's hardware. Directories provide the associations between files contained by a file system and the names by which those files are known.

The figure given below is a directory structure of a UNIX file system. UNIX file systems are hierarchical in structure, similar to a corporate organization chart or family tree. Appearing at the top of the file system is the root directory, in this example the root directory is named root. The root directory is the structural foundation for a file system. All other directories and files are built upon the root directory. In this example, the root directory contains references to ordinary files, data1 and data2, as well as references to other directories, bin and gomes. Any file depicted in this example could be a special file, a directory may contain references to special files as well as to ordinary files and other directories. The hierarchical nature of a UNIX file system is a consequence of the fact that directories may contain references to other directories. Therefore, directories also dictate the basic structure of a UNIX file system in addition to associating files with file names.

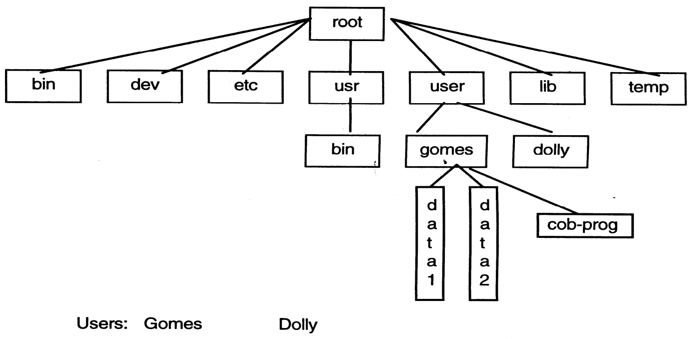


Figure 1.8 A Sample UNIX File Hierarchy

Figure 2.1 File system

Some of the common directories that are found in files are bin,etc,lib,usr,dev and tmp.
The main reason for having different directories is to keep some files together and separate from other files. For example, it is a good idea to keep files used by the system in directories like "bin", "etc", and "lib", while files created by the user in different directories like "usr".

The "bin" directory contains the executable files for most UNIX commands. UNIX commands are executable C programs or Shell Scripts devoted to carrying out single tasks at the users beckoning. Shell scripts are merely a collection of valid UNIX commands. Therefore the "bin" directory assumes a great amount of importance in a UNIX installation.

The "etc" directory contains other additional commands which are related to system maintenance and administration. It also contains several system files which store the relevant information about the users of the system, and the terminals and devices connected to the system.

The "lib" directory contains all the libraries provided by UNIX for programmers. When programs written under UNIX need to make system calls, they can use libraries provided in this directory.

In the "dev" directory, UNIX stores files that control character and block devices. UNIX has a file for each device that it has to deal with. In fact all the information that is displayed on the terminal of each user comes from a "terminal" file in this directory.

In the "usr" directory, the user work area is provided. It is general practice to create each users "home" directory within this directory so that users can store data and other files within them. The user is free to organize his home directory by creating other subdirectories in it, to contain functionally related files. Within the "usr" directory, there is another directory, "bin", in which additional UNIX commands are stored.
The "tmp" directory is the temporary directory into which most of the temporary files are kept. As compared to the others, this directory assumes the least amount of importance, both from the system and the user point of view. Files stored in this directory are automatically deleted when the system is shutdown and restarted.

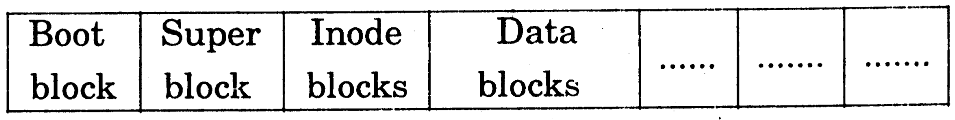
Creating a UNIX file system is analogous to formatting and partitioning disks and tapes in other operating systems. File systems are constructed on all types of media, magnetic disks, and magnetic tapes. By far the most commonly used storage media is magnetic disk – UNIX is essentially a disk operating system. A single disk may contain more than one distinct file system. Each such file system is similar in structure but occupies a dedicated region of the disk. A typical UNIX installation consists of several interconnected file systems that span several disks. The layout of the various file systems is based upon the installation's needs. The software used in an installation, the users and devices it supports, and the means of backing up programs and data all factor into the number, size and layout of an installation's component file systems.

The internal representation of a file is given by an "i-node" (pronounced as 'eye-node') which is short for "index-node". The i-node contains a description of the disk layout of the file data, and other information such as the file owner, access permissions and access times. Every file has a unique i-node, but it may have several names, each of which maps onto a single i-node. Each of these multiple-name i-nodes, that refer to the same file are called a link. When a process refers to a file, the kernel first retrieves the i-node for that file and obtains all relevant information about the file before allowing access to it. On the same lines, when a process creates a new file, the kernel allocates an unused or free i-node to it.

2.3 STRUCTURE OF UNIX FILE SYSTEM

A file system in UNIX consists of a sequence of logical blocks, each containing a series of 512, 1024, 2048 bytes or any other convenient multiple of 512 bytes, depending on the system implementation. Using large logical blocks increases the effective data transfer rate between the disk and memory, but it may also lead to reduction in its effective storage capacity. Thus the block size is usually maintained at 1024 bytes in most UNIX installations.

The file system, as a whole, has the following structure.



File System Layout

BOOT BLOCK

The beginning of the file system is represented by the file system and contains the "bootstrap" program. This program code is required for the "booting" or initialization of the operating system. When the computer is turned on the hardware reads the boot block into the memory and jumps into it. In a normal process only one boot block is needed to start up the system, however all file systems contain one boot block.

THE SUPER BLOCK

It contains information which is used to describe the state or layout of the file system. It gives information about how many blocks are there in the entire system, how large it is, how many files it can store, and where to find empty files along with other relevant information about i-nodes and so on. For example for a 1K block each block of the bitmap has 1k bytes and thus can keep track of 8191 i-nodes. For 10000 i-nodes 2 bit map blocks are needed. Since i-nodes are 32 bytes a 1k block holds up to 32 i-nodes.

THE I-NODE LIST

The super block in a file system is followed by a list of i-nodes. When a file is located its i-node is also located and brought to the i-node table, where it remains until it is closed. The main function of a files i-node is to tell us where the data blocks are. The i-node also has the mode information which tells us the kind of file it is such as regular, directory and so on. It also includes information about the protection associated with the file. The link field of the i-node records how many directory entries point to the i-node so that the file system, knows when to release the files storage. The kernel references i-nodes by an index of the i-node list.

THE DATA BLOCK

They contain file data and administrative data. An allocated data block can belong to one and only file in the file system. This is why it is a good idea not to have the block size greater than 2048 bytes, since even small files of say 1000 bytes will occupy a whole data block, thus wasting a whole 1048 or more bytes in the process.

The structure of an i-node is as follows

A typical i-node for a disk file has the following fields in UNIX.

1. The File Owner: The person who has created the file is the file owner. Ownership is divided between the owner and the group to which the owner belongs.

2. File Type: The file may be a regular file, a directory or a special file. This is explicitly specified in the i-node of the file.

3. File Access Permissions: We have seen above, the security measures that UNIX provides. Individual file permissions only serve to enhance them. These permissions are also part of the i-node of a file.

4. File Access Times: UNIX gives tremendous amount of importance to time and records for each file, its last access and modification times in the i-node.

5. Number of Links to the File: Links are different names given to one and the same file. Note that, though a file maybe given more than one name, it always has one distinct and unique i-node. Therefore we have to understand here that a link to a file is not a new copy of the file with a different name, but simply the same file with a new name. A file may be accessed by specifying its name or by specifying the names of one of its links.

6. File Size: The size of the file in bytes is also stored in its i-node.

7. Disk Addresses: These are addresses of disk blocks that contain the data in the file. A file is rarely stored on a disk in contiguous blocks. Therefore to retrieve a file, the addresses of the blocks that contain the data are stored in a tabular format in the i-node.

2.3.1 Accessing and Storing Files

Internally any file is identified by UNIX by a unique "Inode number" associated with it. We can obtain inode number associated with a file by using the command ls-i.

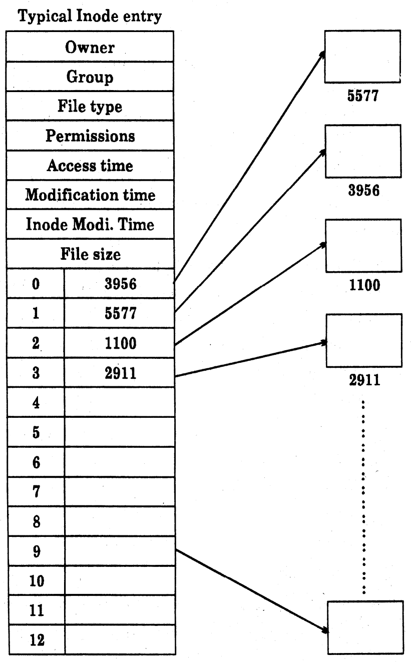
```
$ ls -i Files
Files 15836
```

Here 15836 is the inode number. A directory file contains the names of the files and sub-directories present in that directory along with an inode number for each. The inode number is nothing but an index into the inode table where the information about the file is stored. The inode number 15836 contains information about the files.

Each inode entry in the inode table consists of 13 addresses other than the information about the file, 13 addresses completely specifies where the contents of the Files are stored on the disk. These addresses may be numbered 0 through 12. Each one points to the datablock. One datablock size is 1024 bytes, i.e., I KB. The first ten addresses, 0 through 9 point to 1 KB Block on disk.

i.e., from 0th entry to 9th entry we can store 10 KB of file contents. Suppose a file of size 4 KB may have its entries as shown below.

Fig 2,2



The address 3956 signifies where the first 1 kilobytes of the File are stored. The next 1 KB is stored in 5577, and next 1 KB at 1100, last 1 KB at 2911.

These addresses may be scattered throughout the disk. Thus the addresses 0 to 9 can handle a file of a maximum size of 10 KB. For files larger than this, UNIX has a very interesting way of indicating their location. The First 10 entries points to data blocks and 3 entries points to data blocks indirectly.

From 0 to 9, each entry contains address of one data block 10th entry consists of 256 four byte slots which can store 256 more addresses. Each of these 256 addresses can point to I KB Block on disk. Therefore 10th entry can point to 256 addresses, which inturn points to 256 data blocks.

Thus, the address 10 can handle a file of a maximum size of 256 KB.

The 10th entry is called "single indirection". For a still larger file, double indirection is used. The 11th entry in inode table points to a block of 256 addresses, each of which inturn points lo another set of 256 addresses. Thus maximum file size accessible by double indirection equal to 256 x 256 KB, which is 64KB.

For even larger file, UNIX uses triple indirection. This way 12th entry in inode table points to a block of 256 addresses, each of which inturn points to another set of 256 addresses, each of which inturn points to another set of 256 addresses. Thus maximum File size accessible by triple indirection equal to 256 x 256 x 256 KB, which is 16GB.

The maximum file size UNIX providers for is the sum of sizes accessible by the 13 addressers that occur in the inode entry. I

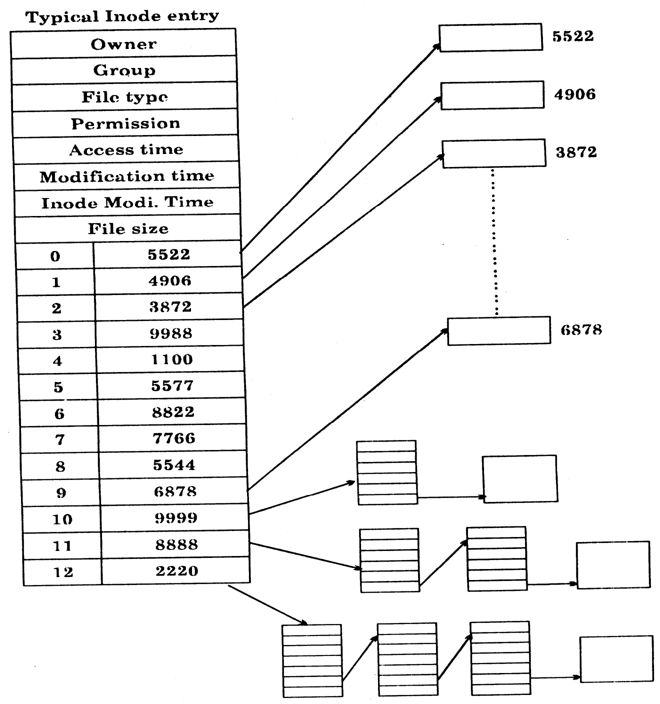


Fig 2.3

2.4 TYPES OF FILES

Ordinary Files

An ordinary file is a randomly addressable sequence of bytes. Most files, including text files, source code, programs, and so on, are ordinary files. Unlike other operation systems, UNIX does not impose a structure on or make assumptions about ordinary files. Instead, the users of an ordinary file and the programs that operate on it determine its structure. UNIX treats all ordinary files in a uniform manner regardless of their content.

The following two examples describe two common ordinary files, text files and binary executable files. These files have specific yet very different structures. For example a text file is a sequence of ASCII characters. Spaces, tabs, punctuation marks, and other non–alphanumeric characters delimit words and newline characters separate lines.

Directories

A directory entry consists of a name and a reference (or pointer) to a file. As discussed previously, a directory entry may refer to an ordinary file, a special file, or another directory. Directory entries are also called links, they link a file with its name or names. Each file name within a directory is unique. However, a file name need not be unique across a file system, a single name may appear in any number of separate directories.

In general, directory entries provide direct access to file. The pointer refers to a system table, called an i-node, which contains a files disk layout.

Each directory, except the root directory, has a single predecessor or parent directory and may have several successors or child directories. The root directory may have children but by definition cannot have a parent. A child directory is frequently called a sub-directory. The arrangement between parent and child directories guarantees the file system's integrity and gives it the hierarchical structure.

Due to its unique position in the hierarchy, the root directory is the ancestor of all other directories. Any given directory on the system, and consequently any file, can be referenced by listing in appropriate sequence the directories that descend from the root to the given directory. Conversely, ascending through the hierarchy from any directory ultimately leads to the root directory.

Special Files

Special files represent input/output (i/o) devices, like a tty (terminal), a disk drive, or a printer. Because UNIX treats such devices as files, a degree of compatibility can be achieved between device i/o, and ordinary file i/o, allowing for the more efficient use of software. Special files can be either character special files, that deal with streams of characters, or block special files, that operate on larger blocks of data. Typical block sizes are 512 bytes, 1024 bytes, and 2048 bytes.

2.5 FILE AND PATH NAMES

The latest release of System V, SVR4, allows file names of as many as 256 characters. Prior to SVR4, UNIX file names were limited to 14 characters. As illustrated by the following example, UNIX is case sensitive – upper - and lowercase are distinct.

Example: **file, File and FILE** are all unique names.

Characters other than alphanumeric are also permitted within file names. However, the semicolon (;), ampersand (&), left parenthesis (( ), right parenthesis ( )), pipe (|), caret (^), greater-than sign (>), less-than sign (<), and minus sign (-) characters have special meaning to the command processor. These characters are frequently called meta-characters. The use of meta-characters file names is legal but difficult and should be avoided. It is common practice to name files using only alphanumeric characters, the underscore (_), and the period (.).

Files are referred to by pathnames. A pathname consists of at least a file name optionally preceded by a list of directory names. A pathname may refer to an ordinary file, a special file, or a directory since the same naming conventions apply to all file types. That is, the file name that terminates a pathname may be that of an ordinary file, a special file, or a directory.

A pathname consisting only of a file name refers to a file in the current directory. A pathname that includes a list of directories can refer to a file anywhere in the hierarchy. Directory names within a pathname are separated by slash characters (/). For this reason, slashes are not permitted within a file name except for the root directory, the root directory is referred to by a slash alone. A pathname's list of directories specifies, the appropriate order, those directories that must be traversed to locate the named file. A pathname that starts with a slash is called a full path. The leading slash refers to the root directory. The directory name following the leading slash refers to one of the root's children. In turn, it is followed by the name of one of the root directory's grandchildren, and so forth. A full path always refers to the same file since it used the root directory as a reference point. For example path is /usr/nachappa/note1/exam means exam is stored note1 directory which is in nachappa directory. Nachappa directory is in usr directory which is a subdirectory in /.
A relative path starts with something other than a slash. The first directory in the list is a child of the current directory, the second a grandchild of the current directory, and so forth. A given relative path can refer to different files since it depends on the current directory as a reference point. In Figure above, the pathname bin can refer to /bin if the current directory is the root directory, or to /usr/bin if the current directory is /usr. Also, if the root is the current directory, bin refers to /bin and usr/bin refers to /usr/bin.

Two special-purpose file names, dot (.) and dot-dot (..), appear in every directory. The file name dot refers to the directory that contains it and dot-dot refers to the parent of the directory that contains it. In other words, every directory contains a reference to itself – dot- and a reference to its parent – dot-dot. In the root directory, dot and dot-dot are equivalent since the root has no parent. Furthermore, dot-dot is essential to the structure of the file system since it provides the means for ascending from a directory to its parent.

2.7 FILE NAME GENERATION

File name generation is a convenient short hand for referring to files and directories as command arguments. The characters *, ?, [, and], when used in command arguments, specify file name patterns. The shell expands these patterns into lists consisting of all file names that both match the pattern and refer to a valid file. The expanded list is passed to the command as its arguments instead of the file name pattern. If the expanded list is empty, i.e., there are no files which match the pattern, the pattern itself is passed.

To examine file name generation, suppose that the following files are contained in the current directory:

```
. module1.c
.module2.c
.module3.c
.module12.doc
.module22.doc
.module32.doc
```

The question mark character (?) matches any single character. Therefore, the pattern module?.c is expanded by the shell into the following list:

module1.c module2.c module3.c

For example:

```
$ echo module?.c<Enter>
module1.c module2.c module3.c

$ _
```

Recall that echo outputs its arguments back to the terminal. However in this case, the single argument module?.c was expanded into the list of file names before the echo command was executed. That is, the shell replaced module?.c module1.c with module2.c module3.c before it invoked the echo command.

The asterisk character (*) matches any string of characters, including the empty or null string. The previous example could have been typed:

```
$ echo module*.c<Enter>
module1.c module2.c module3.c  $ _
```

However, it could also have been typed:

```
$ echo mod*.c
module1.c module2.c module3.c

$ _
```

Finally it could have been typed:

```
$ echo *.c<Enter>
module1.c module2.c module3.c
$ _
```

Keep in mind that where * matches strings of any length, ? matches only single characters. Consider the following two examples:

```
$ echo *.*<Enter>
module1.c module2.c module3.c module1.doc module2.doc module3.doc
$ echo *.?<Enter>
module1.c module2.c module3.c
$  _
```

In the second example, the files ending in doc didn't appear because ? cannot match a three-character string.

File name generation using brackets ([,]) is slightly more complicated. Brackets specify character classes, that is, groups or ranges of characters which meet certain criteria. A class may be enumerated explicitly, for example: [abcdefghijklmnopqrstuvwxyz]

This class matches any single lowercase character. Lexical character ranges may be used within a character class specification to save typing.
The syntax for a lexical range is: [l-u]

where l is the lower bound and u is the upper bound. The character range includes l, u and all characters that are lexically between them. The class of lowercase characters is specified as follows: [a-z] and the class of uppercase characters is specified by: [A-Z]

A character class may contain multiple range specifications. The class of all alphabetic characters is specified by: [a-zA-Z]

Character classes match single characters in a fashion similar to ?. To examine character classes, let us add the following to the list of file names:

```
 modulea.c
moduleb.c
modulec.c
```

and consider the following examples:

```
 $ echo *[a-z].c<Enter>
modulea.c moduleb.c modulec.c
$ echo *[0-9].c<Enter>
module1.c module2.c module3.c
$ echo *[ab12].c<Enter>
modulea.c moduleb.c module1.c module2.c
$ echo *[!0-9].c<Enter>
modulea.c moduleb.c module3.c
$ echo *{A-Z].c<Enter>
*[A-Z].c
$  _
```

Of note in these examples: . The class of numeric characters is specified by [0-9].
.[ab12] is an explicit class consisting of a, b, 1 and 2.

Within a character class the exclamation point (!) negates or complements the class. That is, the pattern matches any character except those characters within the class.

In the final example, the file name pattern failed to match a file. Therefore, the file name pattern is passed to the echo command unchanged.

File name patterns are used frequently, to move, copy, print and occasionally delete files; and therefore particularly important to the UNIX user. New users are well advised to practice the use of these patterns until they can produce reliable results. Many a file has been lost due to the careless or hurried use of file name patterns. If the reader is ever unsure of what a particular pattern will produce, test the pattern first with the echo command to insure it behaves as desired.

2.8 FILE COMMANDS

Simple Directory Commands

UNIX commands are entered after the $ prompt is displayed. All UNIX commands should be entered in lower-case characters.

Identifying the Current Directory

The pwd (print working directory) command is used to display the full path-name of the current directory.

```
$ pwd <Enter>
/usr/bin
$
```

Here, /usr/bin is the directory in which the user is currently working

Changing the Current Directory

The cd (change directory) command changes the current directory to the directory specified.
Assume that Ramesh has logged in, and has given the following command:

```
$ pwd <Enter>
/user/Ramesh
$ cd/user <Enter>
$ pwd <Enter>
/user
$
```

Note that the complete path-name has been specified with the cd command. UNIX also allows relative pathnames with commands.

For example, Ramesh can enter the following command after logging in, to change to the parent directory of his HOME directory.

```
$ pwd <Enter>
/user/ramesh
$ cd.. <Enter>
$ pwd <Enter>
/user
```

```
$
```

The two dots refer to the parent directory of the current directory.

Note that the cd command without any path-name always takes the user back to the

Creating a Directory

mkdir (make directory) command is used to create directories.

```
$mkdir prog-files<Enter>
$
```

The subdirectory prog-files is created under the current directory. However, the new directory does not become the current directory.

Complete path-names can be specified with mkdir.

```
$ mkdir user/Ramesh/prog-files <Enter>
```

Removing a Directory

The rmdir (remove directory) removes the directory specified.

```
$ rmdir cob-prog <Enter>
$
```

where cob-prog is the directory which is deleted

A directory can be deleted only if it is:

- Empty (does not contain files or subdirectories)
- Not the current directory.

Complete path-names may also be specified with rmdir.

```
$ rmdir user/ramesh/cob-prog <Enter>
```

In this section, we present commands commonly used to manage ordinary and special files. While these commands frequently apply to directories as well, their use is not limited to directories.

ls – List Contents of Directory

As discussed in section, ls lists the contents of directories. More specifically, it lists the entries contained in a directory along with information pertaining to those entries. In this section, we discuss options and features of ls that were not covered in section.
By default, ls lists each entry in lexical order, one to line:

```
$ ls /bin<Enter>
STTY
acctcom
ar
at
atq
atrm
awk
```

```
backup
banner
basename
batch
```

The above listing of the /usr/bin directory is too lengthy to place in the text in its entirety. It is also difficult to read as it flashes by on the terminal. The –C option instructs ls to format its output in columns:

```
$ ls –C /usr/bin<Enter>
STTY        cut         getpath     mapchan     removepkg   trchan
acctcom     date        getrange    mapkey      restore     true
crontab         getdev          lp                  putdtrp         time
ypwhich
csh         getdgrp     lpstat      pwconv      timex       zcat
csplit      getgid      ls          pwd         touch
$ _
```

Even in this format the complete listing is too lengthy to place in the text. The –C option orders directory entries vertically down each column. To produce a similar output ordered horizontally across each row:

```
$ ls –x /usr/bin<Enter>
STTY        acctcom     ar          at          atq         atrm
awk         backup      banner      basename    batch       bc
bdiff           bfs             cal                 calendar        cancel
captoinfo
cat         checkeq     chgrp       chkey       chmod       chown
write       x286        x286emul    xargs       xrestor
$ _
```

Another useful option for the ls command is –F. Similar to the –p option, -F causes ls to append a character to certain directory entries appearing in the output. Consider the following:

```
$ ls –C –F /<Enter>
altboot*    etc/        install/    opt/        service/    u/
bin@        export/     lib@        proc/       shlib/      UNIX@
dev/        home/       lost+found/ quotas      stand/      usr/
dgn*        home2/      mnt         sbin/       tmp/        var/

$ _
```

The characters appended to file names are interpreted as follows:

The entry refers to an executable program, including both binary executables and shell scripts.

/ The entry refers to a directory. @ The entry is a soft link to another directory entry.Entries that refer to special files or to ordinary files that are not executable appear as they always have. Perhaps one of the most often used options is –l. this option produces the long listing format:

```
$ ls –l /<Enter>
total 70
-r-xr-xr-x      2 root       root       1476 Oct 16       1990 altboot
lrwxrwxrwx      1 bin         bin          8 Oct 16        1990 bin -> /usr/bin
drwxrwxr-x     13 root       sys        4608 Jun   6       20:34 dev
-r-xr-xr-x      2 root       root      1476 Oct 16        1990 dgn
drwxrwxr-x     27 root       sys        3072 Jun   7       08:24 etc
drwxrwxr-x      6 root       sys         512 Jan 26       17:37 home
lrwxrwxrwx      1 bin         bin      8 Oct 16        1990 lib -> /usr/lib

lrwxrwxrwx      1 root      root   Jan 25        13:17 UNIX -> /stand/UNIX
drwxrwxr-x     19 root       sys         512 Jan 25       13:09 usr
drwxrwxr-x     16 root       sys         512 Jun   7       08:24 var
$ _
```

This example output is a subset of the complete long listing for the root directory listed previously. The long listing format provides a great deal of information about the files referred to by the directories entries. From left to right, the columns in this format are defined as follows:

Mode – the file mode consisting of the files type and access permissions. This field will be explained in more detail in section.

Links – the number of (hard) links to the file.

Owner – the login name of the files owner.

Group – the group name of the files group.

Size – the size of the file in bytes.

Month – the month when the file was last modified.

Day – the day when the file was last modified.

Time/year – the time when the file was last modified. However, if the file was not modified within the current year, this field contains the year when the file was last modified.

1 File name – the file name portion of the directory entry. If the directory entry is a soft link, this field also contains the pathname to which the link refers.

The mode field breaks down into four fields as follows:

File type (1 character). This character may take any of the following values:

d The file is a directory.

b The file is a block special file, that is, a block I/O device.
c The file is a character special file, that is, a character I/O device. Terminal devices are typical character devices.
p the file is a named pipe.
l The directory entry is a soft link.
- The file is an ordinary file.

2. Owner permissions (three characters). This field indicates the access permissions applicable to the files owner. The three characters represent, in order from left to right, read, write and execute permissions. rwx depicts the case when read, write and execute access are all permitted. A hyphen (-) in any position means that the respective access right is denied. For example, r-x indicates that read and execute access are permitted, write access is denied. If the field contains all hyphens, that is, the field appears —, all access is denied.

3. Group permissions (three character). This field indicates the access permissions applicable to the files group.

4. Other (or public) permissions (three characters) – this field indicates the access permissions applicable to all users other than the files owner and members of the files group.

It is common to use ls to list information about a directory instead of its contents. Suppose you wished to learn the status of the /usr/bin directory and issued the following command:

```
$ ls –l /usr/bin<Enter>
total 12086
-r-xr-xr-x      2 bin     bin     52172  Oct  15   1990  STTY
-rwxr-xr-x      1 bin     bin     22736  Oct  15   1990  acctcom
-r-xr-xr-x      1 bin     bin     45848  Oct  15   1990  ar
-r-xr-xr-x      1 bin     bin      6464  Oct   1   1990  ypcat
-r-xr-xr-x      1 bin     bin      6040  Oct   1   1990  ypmatch
-r-xr-xr-x      1 bin     bin      8016  Oct   1   1990  ypwhich
-r-xr-xr-x      3 bin     bin     10584 Oct   15   1990  Zcat
$ _
```

ls dutifully produces a long listing of the /usr/bin directory contents instead of a listing on the directory itself. This works correctly if the argument to ls is a file. For example:

```
$ ls –l /usr/bin/ls<Enter>
-r-xr-xr-x      2 bin     bin     12308 Oct  15   1990 /usr/bin/ls
$ _
```

When the argument to the ls command is a directory, ls lists information regarding the directories contents and not the directory itself. The –d option causes ls to list information about a directory instead of its contents:

```
$ ls –ld /usr/bin<Enter>
drwxrwxr-x    2 bin     bin     5120  Jan  25   13:15  /usr/bin
```

```
$ _
```

Here you can see the mode, owner, size and so on, for the /usr/bin directory.

cp – Copy Files

cp copies files and directories. In simplest form:

cp srcfile destfile

makes a copy of the file srcfile called destfile, where srcfile and destfile can be simple file names or pathnames. If the destination file does not exist, cp creates it. Otherwise, cp overwrites it with the contents of srcfile. For example:

```
$ cp /etc/group /tmp/group<Enter>
$ _
```

creates a copy of the group file in the /tmp directory. The following command replaces the contents of /tmp/group with the contents of the password file:

```
$ cp /etc/passwd /tmp/group<Enter>
$ _
```

The destination file name remains unchanged. However, its contents are replaced with that of /etc/passwd.

Frequently, it is necessary or desirable to copy more than one file at a time. This is accomplished with the following syntax:

cp srcfile1 [srcfile2...] destdir

where srcfile1, srcfile2 through srcfileN are ordinary files and destdir is a directory. cp creates a copy of each source file in the destination directory. Within the destination directory, and existing file with the same name as one of the source files is overwritten – new files are created as needed. The first example in this section could have been accomplished with this syntax as follows:

```
$ cp /etc/group /tmp<Enter>
$ _
```

The result is the same. A copy of the group file is placed in the /tmp directory. A more typical example is:

```
$ cp module1.c module2.c module3.c /home/Manju/tmp<Enter>
$ ls /home/Manju/tmp<Enter>
module1.c
module2.c
module3.c
$ _
```

mv – Move Files

The mv command is similar to cp except that it moves files and directories instead of copying them. Its simplest form is:

mv srcfile destfile

which moves srcfile to destfile. Unlike cp however, srcfile ceases to exist as a result of the move.
If the source and destination files appear in the same directory, moving a file amounts to changing its name:

```
$ ls mod*.c<Enter>
module1.c
$ mv module1.c module4.c<Enter>
$ ls mod*.c<Enter>
module4.c
$ _
```

Multiple files are moved with the following syntax:

mv srcfile1 [srcfile2...] destdir

The source files srcfile1, srcfile2 through srcfileN are moved from their current directory to the destination directory destdir.
Like cp, mv can be destructive. When an existing file has the same name as the destination file or appears in the destination directory and matches the name of one of the source files, it is removed and replaced with the contents of the source file. Consider the following:

```
$ ls /tmp/*.c<Enter>
/tmp/module4.c
$ mv module2.c /tmp/module4.c<Enter>
$ mv –i module3.c /tmp/module4.c<Enter>
mv: overwrite /tmp/module4.c? n<Enter>
$ _
```

In the first example, /tmp/module4.c is silently replaced with module2.c. In the second example, -it caused mv to report the existence of the destination file and prompt the user for direction. If the user responds with y, the destination file is replaced. Otherwise, it remains intact and the move does not take place.
Ordinary and special files can be moved from one file system to another. Directories may be moved only within a file system. Suppose that /home and /tmp appear on separate file systems.

rm – Remove Files

The rm command removes files. The basic syntax for rm is:

rm file1 [file2...]

where file1, file2 through fileN is a list of files to be removed. For each file name in the argument list, rm removes the associated directory entry. If the deleted entry was the only remaining hard link to the file, the file itself is removed.

As with both cp and mv, rm can be very destructive. The –i option, demonstrated in the following example, prevents the accidental loss of files:

```
$ rm –i *.c<Enter>
rm: remove module1.c: (y/n)? n
rm: remove module2.c: (y/n)? n
rm: remove module3.c: (y/n)? n
rm: remove module4.c: (y/n)? n
$ _
```

With this option, the user is prompted for a y or n response for each file in the argument list. This is especially useful when using file name patterns to delete files. rm automatically prompts the user in a similar fashion for each file in the argument list that the user owns but is denied write permission. For example:

```
$ ls-1 /tmp/module1.c<Enter>
-r—r—r—   1 Manju   other      o Jun 25 14:34 /tmp/module1.c
$ rm /tmp/module1.c<Enter>
rm: /tmp/module1.c: 444 mode ? n
$ _
```

There are occasions where a directory and all of its descendants need to be deleted. For this purpose, the –r option causes rm to remove each directory that appears in the argument list, along with all of their files and descendants.

tail – Print Tail Portion of File

tail is similar to cat in that it prints files to standard output. However, tail allows the user to be a bit more selective. tail starts its output from a user-specified point within the file and continues until it reaches the end of the file. By default, it starts its output 10 lines form the end of the file:

```
$  tail /etc/passwd<Enter>
adm:x:4:4::0000-Admin(0000):/var/adm:
uucp:x:5:5:0000-uucp(0000):/usr/lib/uucp:
lp:x:7:8:0000-LP(0000):/home/lp: /sbin/sh
service:x:9:9:Service Login: /service:
nuucp:x:10:10:0000-uucp(0000):/var/spool/uucppublic: /usr/lib/uucp/uucico
listen:x:37:4:Network Admin: /usr/net/nls:
sync:x:67:1:0000-Admin(0000): / : /usr/bin/sync
```

```
install:x:101:1:Initial Login: /home/install:
Manju:x:1000:1:Gundu Rao: /home/Manju:
Nach:x:1001:1:Varun: /home/Nach:
$ _
```

In this example, tail printed the last 10 lines of the password file.
The starting offset can be modified so that additional or fewer lines are included in the output:

```
$ tail –5 /etc/passwd<Enter>
listen:x:37:4:Network Admin: /usr/net/nls:
sync:x:67:1:0000-Admin(0000): / : /usr/bin/sync
install:x:101:1:Initial Login: /home/install:
Manju:x:1000:1:Gundu Rao: /home/Manju:
Nach:x:1001:1:Varun: /home/Nach:
$ _
```

The option –n, where n is an integer value, specifies the starting offset from the end of the file. tail can also be instructed to key on characters or blocks instead of lines. For example, to print the last 10 characters of a file:

```
$ tail –c /etc/passwd<Enter>
home/Nach:
$ _
```

The –c option instructs tail to measure files in characters instead of lines.
The –b option calls for tail to measure files in blocks.
Replacing the hyphen with a plus sign instructs tail to start output relative to the top of the file rather than the end of the file. For example, an argument value of +8 causes tail to start its output at the eighth line, that is, to print the entire file except for the first seven lines:

```
$ tail +8 /etc/passwd<Enter>
service:x:9:9:Service Login: /service:
nuucp:x:10:10:0000-uucp(0000):/var/spool/uucppublic: /usr/lib/uucp/uucico
listen:x:37:4:Network Admin: /usr/net/nls:
sync:x:67:1:0000-Admin(0000): / : /usr/bin/sync
install:x:101:1:Initial Login: /home/install:
Manju:x:1000:1:Gundu Rao: /home/Manju:
Nach:x:1001:1:Varun: /home/Nach:
$ _
```

Perhaps the most useful option is –f. Given this option, tail does not exit when it reaches the end of the file. Instead, it continues to read the file periodically and outputs additional lines as they are appended. tail continues to wait for and print additional lines until the user halts the program. This feature, which cannot be adequately demonstrated here since its output appears over time, is especially useful for monitoring logs.

2.9 ACCESS PROTECTION

UNIX grants or denies access to a file based on the relationship that exists between the file and a user attempting to access it. There are three possible relationships:

1. The user may be the files owner.
2. The user may be a member of the group associated with the file.
3. The user may be unrelated to the file. Users unrelated to a file are considered members of the general public

Each time a user attempts to access a file, he or she is treated as the files owner, a member of the files group, or a member of the general public. Each file has separate access rights for each of these user classes. These rights, called permissions, indicate whether a member of the associated class is allowed to read, write, or execute the file. A particular access right is either granted or denied. All the following eight permutations, called the files access modes, are supported

All permissions denied
Read allowed (called read only)
Write only
Read and write allowed
Execute only
Read and execute allowed
Write and execute allowed
Read, write, and execute allowed

Each user is identified by a unique integer called a user ID. Users are also assigned to groups identified by a separate group ID. Group members usually have something in common, such as working on a common project or attending the same class. The group called other is the default group for users who are not assigned to a specific group.

Each file is assigned an owner and a group when it is created. A files owner is generally the user who created it. A files group is usually the owner's group, that is, files are generally assigned to the same group as their owner's. However, these assignments can be changed.

When a use accesses a file, that is, attempts to read, write or execute a file, the system compares the user ID of the files owner with the ID of the user attempting to access it. If the ID's are the same, the user is granted the owner's access rights. Otherwise, UNIX compares the user's group ID with the files group ID. If they match, the user is accorded the group's access rights. Otherwise, the user is allowed the public access rights. For ordinary files

Read allows the user to read the files content. Read permission is also required to copy a file.

Write allows a user to alter the files content. Write access is required to overwrite a file.

Execute allows execution of the file. Since UNIX does not distinguish between executable programs and other types of files, execute permission may be granted whether or not the file is a program.

For directories:

Read allows the user to list the directory's entries.
Write allows the user to modify a directory's content. Write permission is required to create entries in and remove entries from a directory.
Execute permits use of the directory within a pathname.

For special files:
Read permits input from the associated device.
Write allows output to the associated device.
Execute has no meaning.

A files access mode does not affect its owner's ability to change its permissions, its group, or its owner. A files owner can always change it access mode, its group, and its owner, even when all permissions are denied. However, if a files owner assigns the file to another user, only the new owner can exercise these privileges. A files owner cannot change its ownership and then reverse the process

chmod - Change File Mode

The chmod command changes the access mode for files and directories. Its basic syntax is:

chmod mode file1 [file2...]

where mode specifies the desired access mode and file1, file2 through fileN is a list of files and directories.
The mode value can be specified numerically or symbolically. Numerically, an access mode specification is a three-digit octal number. The first digit represents the owners permissions, the second represents the groups permissions, and the third represents the permissions granted to all other users. Discrete values represent read, write and execute permission; a value of 4 indicates the read access is allowed, 2 indicates that write access is allowed, and 1 indicates that execute access is allowed. The permissions granted to a particular user call is the sum of the individual read, write and execute values assigned to that user class. A 0 indicates that all permissions are denied. For example:

000	All access denied.
100	Execute by owner is allowed.
200	Write by owner is allowed.
300	Write and execute by owner is allowed.
400	Read by owner is allowed.
500	Read and execute by owner is allowed.
600	Read and write by owner is allowed.
700	read, write and execute by owner are allowed.
070	read, write and execute by group are allowed.
007	read, write and execute by other allowed.
777	read, write and execute by owner, group and other are allowed.

To change the mode of a file to read only for all users:

```
$  chmod 444 /tmp/group<Enter>
$ ls –1 /tmp/group<Enter>
-r—r—r—      1 Manju     other      572    Jun    25  12:32  /tmp/group
$  _
```

To change the mode of a file such that all users can read and execute the file but only the owner can write to the file:

```
$  chmod 755 command<Enter>
$ ls –1 command<Enter>
-rwxr-xr-x      1 Manju     other      572    Jun    25  12:32  command
$  _
```

In symbolic form, mode is expressed as follows:

user_group operation permissions

where user_group represents one or more of the owner, group and other user groups, operation indicates whether permission is enabled, disabled or assigned and permissions represents the access permissions. User classes are represented as follows:

u represents the files owner.
g represents the files group.
o represents all users other than the files owner and group.

Operations are defined as follows:

= assigns permissions.
+ grants permission.
denies permission.

Permissions are represented by the characters used in a long listing:

r represents read access.
w represents write access.
x represents execute access.

They are used, for example, to assign the owner read, write and execute permission:

```
chmod o=rwx filename
```

To add execute permission for the files group and remove write permission from all users except the files owner and group:

```
chmod g+x, o-w filename
```

CHAPTER 3

3.1 INTRODUCTION

There is a large family of UNIX programs that read some input, perform a simple transformation on it, and write some output. Examples include grep and tail to select part of the input, sort to sort it, wc to count it, and so on. Such programs are called filters. This chapter discusses the most frequently used filters and pipes and processes.

3.2 I/O REDIRECTION

In UNIX, when a user executes a command, the shell runs the command and assigns to it the keyboard as the source of input. The keyboard is referred to as the standard input file.

Let us take an example of the cat command. When followed by a filename, all the lines in the file are displayed till the end of file. Without the filename, however, the cat command takes its input from the standard input file as depicted:
$ cat   The cat command waits for input from the keyboard. As a user enters characters from the keyboard, they are displayed on the screen as shown:

```
$ cat
This is a chair
 This is a chair
Cursor waits at next line
```

The user can keep on entering lines. To indicate the end of input, the user has to press Ctrl and d. The $ prompt then appears on the VDU.

Not all commands use the standard input file for input. For example, cd command does not use the standard input file. In UNIX, the standard files are always assigned a number called the file descriptor. The file descriptor 0 is assigned to the standard input file.

Standard Output File

In UNIX, the shell assigns VDU as the destination for the output of any command that it executes. The VDU is referred to as the standard output file .
For example, when you. issue the command

```
$ ls <Enter>
```

the shell executes the command and sends its output-directory listing-to the standard output file.
Not all commands use the standard output file for output. For example, mkdir command does not use the standard output file. The file descriptor 1 is assigned to the standard output file.

Standard Error File

When an invalid command is typed at the $ prompt, the shell displays appropriate error messages on the VDU. The VDU is thus also the standard error file .
For example, the cat command followed by a filename which does not exist will generate an error message on the VDU. The file descriptor 2 is assigned to the standard error file.

Redirection

Redirection changes the assignments for the standard input, standard output and the standard error. Using redirection, the input to a command can be taken from a file other than the user's terminal keyboard. Similarly, the output of a command or the error can be written onto a disk file or onto the printer, instead of the VDU. Following are the three types of redirection in UNIX:
Input redirection
Output redirection
Error redirection
..
Input Redirection

The following example illustrates the usage of input redirection:

```
$ cat < test1 <Enter>
```

Here, < (less than symbol) implies input redirection from the named file. The cat command will take each line of the file test1 as input and display it on the VDU.
Thus, in UNIX, it is possible to specify that the standard input, instead of coming from the standard input file comes from a disk file.
The above command can also be written using the file descriptor as:

```
$ cat 0< test1 <Enter>
```

Here, 0 indicates input redirection.

Output Redirection

The following example illustrates the usage of output redirection:
$ cat testl > test2 <Enter>
Here, > (greater than symbol) implies redirection of output to the named file. The output of the cat command is written onto a disk file called test2.
In output redirection, the file to which the output is redirected is first created on disk as an empty file and then the output is sent to this file. However, if the named file (test2) already exists, its contents will be deleted before the output is written to it. If the user wants to append the output to test2 file, the command to be given is:

$ cat testl " test2 <Enter>

The previous command can also be written using the file descriptor as:

$ cat testl l> test2 <Enter>

Here, 1> indicates output redirection.

Error Redirection

The following example illustrates the usage of error redirection:
$ cat test 2> error-mesg <Enter>
Assume that the file test does not exist in the current directory. So, when a user tries to execute this command, UNIX will generate an error message since the execution is unsuccessful. This message, which would otherwise be displayed at the terminal VDU (standard error file), will now be written onto the file called error-mesg. As in the case of output redirection, error redirection also first creates the file to which the error messages are redirected and then writes the error output onto the file.

3.3 FILTERS

The grep filter

The grep filter searches a file for a particular pattern of characters, and displays all lines that contain that pattern. The pattern that is searched for in the file is referred to as the regular expression (grep stands for globally search for regular expression and print out.

grep cannot be used without specifying a regular expression. The format of the grep command is:

grep regularexpression filename

The filename is optional in the grep command. Without a filename, grep expects standard input. As a line is input, grep searches for the regular expression in the line and displays the line if it contains that regular expression. Execution stops when the user indicates end of input (by pressing Ctrl and d).

Specifying Regular Expressions

Regular expressions can be used to specify very simple patterns of characters to highly complex ones.
Some very simple patterns are shown in Table 2.1.

Regular	Expression Pattern

"A"	the character A
"F"	the character F
"New"	the word New

Table 3.1 Simple Patterns in grep

Note that regular expressions are always given in double quotes.

More complex regular expressions can be specified by using the characters summarized in Table 3.2.

Character	Matches	Example	Description
[]	matches anyone of a set of characters	grep "New[abe)"	specifies the search patterns as 'Newa' or 'Newb' or 'Newc'
[} with hyphen	matches anyone of a range of characters	grep ''New[a-c)"	specifies the search patterns as 'Newa' or 'Newb' or 'Newc'
1\	pattern following it must occur at the beginning of each line	grep "'Wew[abe]"	specifies the search patterns as 'Newa' or 'Newb' or 'Newc; but these must occur at the start of the lines
1\ within [}	pattern must not contain any character in the set specified	grep "New[Aa-c]"	specifies patterns containing the word 'New' followed by any character but a or bore
$	pattern preceding it must occur at the end of each line	grep "New[abe}$"	specifies the search patterns as 'Newa' or 'Newb' or 'Newc; but these must occur at the end of the lines
. (dot)	matches anyone character	grep ''New.[abe]"	specifies patterns containing the word 'New' followed by any character, followed by either a or b or c
\ (backslash)	ignores the special meaning of the character following it	grep ''Newl \[abcl)"	specifies one search pattern, New.[abe}, in which the dot signifies a dot character itself

Table 3.2 Characters Used in grep Patterns

Options_of the grep Filter

The grep command also has options which alter the output of the command. These are:

-n This prints each line matching the pattem along with its line number. The number is printed at th beginning of the line.
-c This prints only a count of the lines that match a pattern.
-v This prints out all those lines that do not match the pattern specified by the regular expression.

Options must be specified before the regular expression. Options can also be combined (For example, -n and -v can be used together as -nv)

The pgFilter

This filter is used to display a large file, a screenful at a time.

The usage of pg filter is:

```
$ pg datal <Enter>
```

The above command will display the contents of data1, a screenful at a time.

The WC Filter
The wc filter is used to count the number of lines, words and characters in a disk file or in the standard input.

The usage of we filter is:

```
$ wc test <Enter>
2 7 29
```

The file test has 2 lines, 7 words and 29 characters.
Table 2.3 summarizes the options of we filter.

Command	Function
wc -l	Displays the number of lines
wc -w	Displays the number of words
wc -c	Displays the number of characters

Table3.3 Options of wc filter

Since we is a filter, if no filename is provided, it uses the standard input as depicted below:

```
$ wc <Enter>
wc is a filter <Press Ctrl and d> 1
```

The tr Filter
The tr filter can be used to translate one set of characters to another.
This filter can also be used to squeeze repeated characters into one. Several commands have multi-column output and the gap between columns is more than one space. In such cases, if the user wants a particular column, cut cannot be used, since the column separator has to be a single character. Therefore, the solution is to squeeze the several spaces between columns into a single space, and then use the cut command to extract the desired columns.
The -s option is used to specify squeezing of several occurrences of a character into one character as illustrated:

```
$ who
  ram         ttyi2 Dec16    09:06
  john        ttyj3 Dec16   10:40
$
```

To make the column separator a single space, tr -s has to be used as shown:

```
$ who> temporary
$ tr -s " " temporary
ram  ttyi2 Dec16 09:06
john ttyi3 Dec16 10:40
$
```

Here, the -s option of tr works on every record, squeezing(reducing) occurrences of the string given in quotes (space, in this case) to one.

Another common usage of tr is case conversion.

```
$ tr "[a-z]" "[A-Z]" <Enter>
sumukha publications<Press Ctrl and d>
SUMUKHA PUBLICATIONS $
```

The above command converts all lowercase alphabets to uppercase.

The cut Fitter

The cut filter of UNIX is useful when specific columns from the output of certain commands (like Is, who) need to be extracted.

Consider the following example:

```
$ who > templ
$ tr -s " " templ > temp2
$ cut -d " "-£1 temp2
```

Here, the output of who command is written to a file temp1, tr -s has been used to make the column separator in the file, temp1, a single space. Finally, cut command has been used to extract only the names of the users who are logged in.

Refer to Table 2.4 for the options of cut.

Command	Function
cut -1	Displays the columns specified
cut -c	Displays the character
cut -d	Specifies the column delimiter

Table 3.4 Options of cut Filter

sort filter

Sort command is used to sort the data in text file in ascending or descending order. It cn also be used to merge sorted files.

Sort command takes zero , one or more filenames as arguments. Sort command without any arguments reads the data from the keyboard sorts the data and displays the sorted data on the screen

```
sort
Rama
Krishna
John
Govind
<ctrl d>
Govind
John
Krishna
Rama
$
```

The different options are:

-d Sorts according to dictionary and ignores punctuation
-n Sorts according to numeric order
-r Sorts in reverse order
-m merges two or more given sorted files into one sorted output file.
-v removes duplicate entries

```
$ sort names.dat > namelist
```

Here names.dat is the file which contains data to be sorted. The data is sorted and the output is stored in namelist file.

3.4 Pipes

UNIX has a feature by which filters and other commands can be combined in such a way that the standard output of one filter or command can be sent as standard input to another filter or command.
For example, to display the contents of the current directory, a screenful at a time, the user would give the following commands:

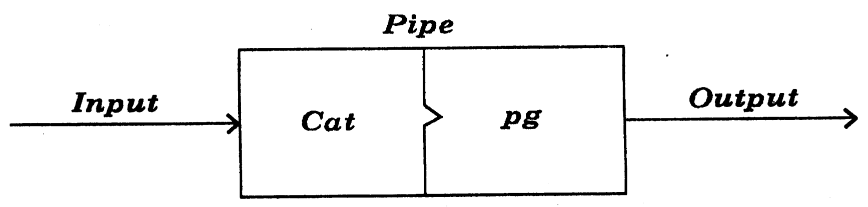
```
    $1s > tempflle <Enter>
    $ pg tempflle <Enter>
```

Here, the listing of the directory is stored in the file tempfile by the first command. This file is then used as input by pg command.
Through the UNIX pipe feature, these two steps can be done in one command without creating a temporary file as shown below:

```
    $ls | pg
    $
```

The vertical bar (|) is the pipe character, which indicates to UNIX to send the output of the command before I as input to the command after I. (Note that on the keyboard, the pipe character appears as a broken vertical line).
This pipe command would work as shown in Figure 3.1
Another advantage of the pipe feature is that programs do not have to be rewritten to perform complex tasks. Instead, the UNIX tools (commands) can be combined. There is no limit to the number of filters or co~nds in a pipe. Consider the example in Figure 2.2 to understand how a pipe works.

The pipe command would work as shown in the figure below.



Working of a pipe

Command 1	| Command 2 |	Command 3 |	Command 4
Should give	Should take	Should take	Should take
Standard	Standard i/p	Standard i/p from	Standard
Output to	from command 1	Command 2 &	input
Command 2	& give standard	give standard	from
	O/p to command 3	o/p to command 4	command 3

3.4 Process

The concepts of time sharing and multitasking in Unix can be generically understood whilst dealing with processes. By itself, the hundreds of programs that come with Unix are not processes. A program is elevated to the status of a process only when it starts executing. Therefore a process is defined as the instance of an executing program.

3.4.1 States of a Process:

Since Unix systems usually have a single CPU, obviously only one process can be executing at one time. Thus a major function of the Kernel is to provide control and support for the many programs that wish to obtain the CPU at the same time.
The most logical approach to this problem is to have a process "scheduler" that decides which process should grab the attention of the CPU. Its like a real-world rat race where only the best man wins. There are six states that a process goes through in a multitasking time-sharing operating system. The following diagram elucidates:

Figure 3,1

The first process is "submit", wherein the process is submitted by the scheduler into the process queue according to a fixed priority. After the process is submitted, there is a certain time interval where it remains in the queue. This is called the "hold" state of the process. When the process was finally the next in the queue to receive the attention of the CPU, it was elevated to the "ready" state.

And finally the big moment arrived. The process gained the attention of the CPU and attained the "running" state.

Each process is allotted a "time slice" of the CPU time for execution. If it manages to complete by then, well and good. Or as soon as the time slice of a particular process ends, the CPU starts running another process, returns this process to the "ready" state and places it back in the process queue. The CPU is said to have performed a "context switch" to a new process and all the necessary parameters and values of the old process are saved for retrieval when its next time slice arrives. This process has to now wait in the ready state till its next time slice.

Some processes need to break to perform disk I/O. The processor cant sit around waiting for such processes to finish with the relatively slow disk I/O procedure. Such processes are placed in the "wait" state until I/O is completed and then placed in the "ready" state in the process queue when the requisite I/O has been completed. A lesson for them to use their CPU time slice better next time.

A process that has completed executing fully is placed in the "completed" state and removed from the process queue.

Thus we can conclude from the above discussion that Unix is a "pre-emptive" operating system where a particular process is only allowed to execute upto the allotted time slice of the CPU after which it is placed back in the queue until its time slice arrives again. The arrival of the next time slice depends upon the scheduling algorithm.

This is in contrast to other contemporary operating systems like MS-Windows where each process gains full CPU time, until and unless it passes control to another process. This allows a particular process to monopolize the CPU. Therefore the operating system is referred to as "non-preemptive".

3.4.2 Process Status: The "ps" command

This command displays the processes that are currently active in the system. These are the processes that have attained the “running” state with the CPU. The ps command also displays additional information about the processes like the process name, process id, the terminal number on which the process is active and the time that has been utilized by the process since it was initiated. In a nutshell, the ps command gives a snapshot of the activity that is going on in the system.

In its simplest form, the ps command by itself displays the information about processes active at the users terminal only.

```
$  ps
  PID        TTY       TIME         COMMAND
  22267        2c        0:03          sh
  22388        2c        0:00          ps
$
```

We have two processes running now. The one called sh is the Unix Shell. This shell is the Bourne Shell. This process is born the minute you login and dies when you log out of the system. Throughout your Unix session, the Shell will always remain active.

The next process was the ps command itself. The process-ids (PIDs) associated with each process are unique and assigned by the system. No two processes with have the same process-ids.If you want additional information about the processes that are active at your terminal, you could opt for a full listing using the –f option.

The ps –f command will display a full listing of processes and commands active at a particular terminal along with a header for each field. Check this out by typing:

```
$  ps –f
    UID    PID       PPID   C   STIME     TTY    TIME     COMMAND
    userl  22267         1   2  12:48:54     2c     0:04       -sh
    userl  22577   22267  12  15:11:48     2c     0:00       ps  -f
$
```

The UID field displays the user-id as the logname of the user of the terminal. The following field, PID shows the process-ids assigned by the system to the processes that are currently active at that terminal. The process-ids, as you will see later, are most useful when it comes to ‘killing’ a process.

The PPID stands for “parent-process-id” and displays the process id of the parent process that spawned the child process. Because the system created a Shell process (sh) for you, the instant you logged in, its parent-id is 1. However, for each user, the PID of the Shell will differ since it is just a number that is assigned to the process as and when you log in.

The Shell has a parent process, whereas it is your shell that gave birth to the ps process. Hence you will observe above that the PPID of the ps-f command is the same as the PID of the Shell.

The STIME is the “starting” time of the process, whereas TIME denotes the cumulative execution time for that process.

Finally, the "C" field denotes the amount of CPU resources that the process has used recently. The Kernel uses this information to decide which of the processes will get the attention of the CPU next. This is known as process scheduling and is one of the principal duties performed by the Kernel. The Kernel will let the process with a lower C value have control of the CPU, before the process with a higher C value is given a hearing.

3.4.3 System Processes:

We have, so far, examined processes associated with each individual user, but this is only the tip of the iceberg. There are several long-lived processes that support the activities of the system and transient processes that live and die when the system goes about its business independently of individual users.

The –e option along with ps command will list every process that is currently active on the system. The ps –e command helps you examine which processes are running in the background, or what the machine is doing behind your back. The output of this command is very useful in diagnosing process-related problems.

```
$   ps   -f
UID      PID    PPID    C      STIME    TTY        TIME      COMMAND
root       0        0    0      Mar 16       ?          0:10       sched
root       1        0    0      Mar 16       ?         20:25       /etc/init
root       2        0    0      Mar 16       ?          0:36       vhand
root       3        0    0      Mar 16       ?          1:22       bdflush
root 24849        1    0     115304      01           0:01       /etc/getty  tty01  m
root 15899        1    0            ?       ?           0:01       /etc/getty  tty03  m
root 15885        1    0            ?       ?           0:01       /etc/getty  tty02  m
root    236        0    0      Mar 16       ?          0:19       /etc/cron
userl22267        1    1     12:48:54      2c           0:04        -sh
userl24919  22267   31    12:17:22      2c           0:01       ps  -ef
$
```

There are several items of interest in this output. This is the first, inside look at processes that we have ever taken. The first process to execute when the machine boots is the scheduler (called sched in the above output). The scheduler is the heart of the time-sharing design of Unix. The scheduler is responsible for determining which of the processes is ready to execute and allocates each process the required system resources, like CPU time-slice, memory and other important considerations.

The scheduler has the value PID 0, and in turn starts off the intialization (or the /etc/init) process. Since the path is so explicitly specified, you must have guessed that it is a file on disk, existing in the /etc directory. The init process gets the value PID 1.

The vhand process is the"virtual handler", which handles the virtual memory of the system and manages to swap the active process between the disk and the real memory, as they run or as they wait in queue to be processed by the system. The vhand process is mainly responsible for managing the system memory in a multitasking environment. The sched and vhand processes are the ones which make up the core or the kernel. Thus the PID of the vhand process is 2 since this the third process to start after booting.

Finally, the bdflush (short for buffer to disk flush), has the job of managing the dist I/O for the system. Thus the disk is updated with the latest information in the buffers, because only data which is changed is written to the disk everytime. The crucial bdflush process is assigned a PID of 3.

The processes such as vhand, bdflush and sched are stored in the file /unix which is the Unix kernel. These system processes are called DAEMONS.

The daemon is defined as a process that runs without the user requesting it to do so. The daemon process is user-independent and terminal-independent. A daemon is, alternately, defined as a process that runs in the background in order to be available at all times.

3.4.4 TYPES OF PROCESS

Processes are classified into three categories.
1. interatice processes
2. Non-interative processes
3. Daemons

Interactive processes(Foreground processes)
All the user processes which are created by users a with the shell, act upon the directions of the users and are normally attached to the terminal are called interactive processes.
Non-interactive processes
Certain processes can be made to run independent of terminals. Such processes that run without any attachment to a terminal are Non-interactive processes.

Daemon processes
The daemon is defined as a process that runs without the user requesting it to do so. The daemon process is user-independent and terminal-independent. A daemon is, alternately, defined as a process that runs in the background in order to be available at all times.

3.4.4 Process control
Running A Process in the Background:

There are some tasks that are time-consuming, and require no input from the user. If these tasks are executed normally, the user may have to wait for a long period of time before the task is completed, to resume the next task. A typical example of such a task is sorting a huge file and redirecting its output to another file on disk. The sorting operation requires a considerable amount of time and the user cannot resume any other task till this task is completed.

But when you are working with Unix, you can break these limitations. Unix offers the user the facility to run such processes in the "background" and resume other tasks in the "foreground". To assign a process to the background, Unix provides the ampersand (&) symbol. Simply type the command and at the end of it, place the & symbol. The process will start executing in the background and you will instantly see a number followed by the Shell prompt. This number is the process-id of the process that you just assigned to the background.

If you have a task that exclusively requires user-interaction, in the foreground, and you want a task to be urgently executed, you can now do it straightaway from the Shell prompt. Without waiting for the first task to terminate. Incredible. And most convenient too. Lets get started right away.

```
$ sort  massivefile  >  sorted.file&
13579
$
```

The task is now assigned to the background leaving the foreground all clear for the user.

If you run a background process and log out of the system, without checking its status, the process will be unceremoniously terminated, since all processes are linked to the Shell which dies when you log out. The remedy to this is the nohup command which will be dealt with in the next section.

The "nohup" command

This command assumes a great amount of importance when it comes to background processes. Unix provides the nohup command to ensure that your background process is not killed or terminated when you log out of the system.

To use the nohup command, involves no hang-ups at all. Just place the word nohup before the command, and the & at the end of the command, to signify that the process should be a background process.

```
$ nohup sort –d bigfile > file.srt&
13772
$
```

3.4.5 Parents And Child proceses

A process is said to be 'born' when it starts executing and 'dead' after it terminates. At this rate, the morality rate for processes in Unix is quite low. Though we must admit that the "daemons", due to their obvious supernatural powers, manage to live as long as the system is alive. Others live and die. And nobody gives a damn.

Very much like life itself, a new process can be born only if another process starts it. The new process is called the 'child' process, whereas the process that starts it, is called the 'parent'. A 'parent' may have multiple children but a child can have only one parent. That's the tragedy of the Unix family. Its always a one-parent family. But all said and done, it's a happy family.

A child process, once started, may in turn 'spawn' new children. And the cycle goes on. Though one never generally speaks of a 'grandparent' of a process, it is possible to trace the ancestry of any process, right to the point where the system was started.

Fig 2.3

3.4.6 How to Kill Processes

For various reasons, a user may want to terminate a process. There may be too many processes running on his terminal which may affect the speed and performance of his current active process. Certain background processes may have gone awry. The remedy to all the above, is the kill command.

Its time to take up arms and go on the rampage. One of the first diagnostic procedures that one must resort to, is the ps command. Now's where the process-ids of the processes actually count. Take note of the PIDs of the processes that are currently active.

```
$ kill processidno
```

for example `$kill 5847`

It terminates the process having process id number 5847

The "nice" Command

In a multitasking system, processes are fighting and grappling to gain what is called "priority". It's a small word with large implications. When a process gains priority, it obtains the attention of the CPU much faster than the other eligible suitors. And the timing is what makes all the difference. The CPU, like a fair maiden, is most sought-after, and the earlier you make the queue, the better.

A process executing on Unix can be assigned a "priority" that ranges from 0 to 39. Normally when a process executes, it is assigned a "priority" of 20. This is a default value that the system assigns as the "priority" for the process. The point to be noted is that, if the "priority" of the process is made lower than 20, it will execute faster and vice-versa. Yes, it is rather paradoxical that the higher the "priority" of a process, the slower it will execute.

The super user is permitted the facility to decrease the “priority” of processes by giving a negative increment to the default priority, thus making the process gain the attention of the CPU earlier.

For example, if a normal user types:

```
$ nice ls
```

his “priority” will increase. Even though no actual increment is specified, the default increment of 10 is taken, hence the process “priority” increases to (20+10) 30. This process will execute much slower than the normal ls –alR command, whose “priority” will still be 20.

```
$ nice –15 cat /etc/passwd
```

The priority of this process now becomes (20-15) 5 and the process executes very fast.

Therefore, to keep a time-consuming program from typing up the whole system, run the program in the background with a high “priority”, thus giving it low preference.

3.4.6 PROCESS CREATION

There are 3 distinct phases in the creation of a process, using 3 important system calls fork ( ), exec( ) & wait ( )•

* fork

A process in UNIX is created with the fork system call, which creates a copy of the process that invokes it. When a process is forked in this way, the child gets a new PID. The forking mechanism is responsible for the multiplication of processes in the system.

The only way a new process is created by the UNIX Kernel is when an existing process calls the fork function.

Eg. : pid_t fork (void);

Returns : 0 in child

process ID of child in parent,

-1 on error.

The new process created by fork is called the child process. This function is called once but returns twice. The only difference in the returns is that the return value in the child is 0 when the return value in the parent is the process ID of the new child.

The reason the child's process Id is returned to the parent is because a process can have more than one child, so there is no function that allows a process to obtain the process Id's of its children. The reason fork returns 0 to the child is because a process can have only a single parent, so the child can always call getppid to obtain the process Id of its parent.

* exec:

The parent overwrites the image that it has just created with the copy of the program that has to be executed. This is done with the exec System call & the parent is said to exec this process. No additional process is created here, the existing program is simply replaced with the new program. This process has the same PID as the child that was just forked.

The exec system call causes a calling process to change its context & execute a different program. There are six versions of the exec system call. They all have the same function but they differ from each other in their argument lists.

* Wait

The parent can execute wait system call to keep waiting for the child process to complete. When the child process has completed execution, it sends a termination signal to the parent. The parent is then free to continue with its other functions.

EXIT COMMAND:

$ exit ; It is a command used for signing off or suspending the session.

There are three ways for a process to terminate normally and two forms of abnormal termination.

1) Normal Termination:
   a. Executing a return from the main function. This is equivalent to calling exit.
   b. Calling the exit function. The function is defined by ANSI C and includes the calling of all exit handlers that have been registered by calling at exit and closing all standard input streams.
   c. Calling the exit function. This function is called by exit I and handles the UNIX - specific details.

In most, UNIX implementations exit (3) is a function in the standard C library while - exit (2) is a system call.

2. Abnormal Termination:

Calling abort. This is a special case of the next item, Since it generates the SIGABRT signals.

3.4.7 SYTEM CALLS

Sometimes we need to use the elemental features of the UNIX system kernel available through system call interface. A process accesses system resources through system call.

In one way or the other, all processes use kernel through system calls interface to access files, to communicate with other processes, and to control processes. System call is subroutine call that causes the kernel to do some operation,. The lowest level of system call is related to I/O operation. When a process makes a system call it goes to kernel mode, after executing the system call it switches from kernel mode to user mode. For example the system calls related to file are open(), read(), write() , close(). System calls related to processes are fork(), exec(). Wait() etc.

3.4.8 Library functions

The UNIX system simplifies the job of building software tools not only because of the programming tools such as vi, lex, yaac but also because of the tool kit of subroutines To use the subroutines provided with the UNIX system effectively, you must know what they can do. You should also know how to add the tools you develop to your tool kit.

The standard UNIX system library provides subroutines for doing buffered input and output, manipulating strings and memory, classifying and converting characters, manipulating time, allocating storage, and accessing variables in the environment. These software building blocks save you time when you program a problem solution. They also provide a portable, standard interface to the UNIX system and to other operating systems that support them. These subroutines are recommended in preference to the system calls.

The heart of the subroutine library is the Standard I/O library, referred to as stdio. The Standard I/O library is a large set of integrated I/O subroutines such as fopen() with a consistent interface and was designed to be called from C programs. Standard I/O also optimizes the size of data passed between such devices as disks, terminals, and programs. For example, Standard I/O knows that I/O operations involving disks are optimally handled in the logical block size of the disk, typically lK byte blocks, and that output destined for a terminal is better handled in lines ending with a newline. The software tools you build should use the Standard I/O subroutines for all input and output.

To access the subroutines in the Standard I/O library, you must include the Standard I/O definition file in your program code, preferably at the beginning, directly after the comments section.

Chapter 4

4.1 Introduction

The System Administrator (SA) is primarily responsible for the smooth operation of the system. It is the job System Administrator to switch on the system console. The System Administrator also creates users and groups of users for the system, and takes backup of data to prevent loss of data due to system breakdown.

4.2 Duties of a System Administrator

A system administrator is solely responsible for the successful management of a system. Below is given a concise list of duties that are performed by a system administrator.

. Starting and shutting down the system.
. User management: System administrator is responsible for adding,
modifying and removing users. It is at this stage that users accounts are opened or removed, passwords are given or withdrawn, minimum working environment is provided via the system-wide profile file and so on.

. Disk space management Commands such as du, df, compress and uncompress and others are used for this purpose.
. Taking backups and restoring files: Depending upon the importance of the data, backups of all important files are taken at regular intervals of time. Some of the commands used for the purpose of taking backups and restoration of required files or entire file systems are cpio, tar and dump.
. Responsible for the installation of the software. Responsible for all the local events: The system administrator is responsible for communicating with all the users informing about their and system activities.

To sum up, the system administrator is responsible for the terminals management, the users management, the software management, the system hardware management, the files and file system management, monitoring the activities of the entire system and accounting of the system usage.

4.3 Privileges of a System Administrator

The system administrator has tremendous powers. There are several commands that are reserved only for his exclusive use.

One of the very important privileges that a system administrator has is that he or she can change the attributes of any file notwithstanding the permissions associated with it.

Another important privilege that an administrator has is that he or she can remove a file using the rm command as well as initiate or kill any process.
A system administrator can use the passwd command to assign a new password to any user even without knowing the old password. Thus, a system administrator has a privilege of changing anybody's password without knowing it. Of course, once a user loses a password there is no way to restore it. However the superuser can assign a new one even without the knowledge of the old password.

Another privilege that a system administrator has is that he or she can reset the system time using the date command.

The date command with a numeric argument of eight-character length that represents the month, day of the month and time in the format MMDDHHmm is used to set the time as shown below. This is a case where the command behaves differently in the hands of a system administrator.

```
#date 12031934
```

A system administrator also has the privilege of communicating with all the users not-withstanding the write permissions associated with the terminals.

The system administrator can limit the maximum size of a file that a user can be permitted to create (using the ulimit command). Further he can allow or deny specific users from using commands such as at, batch and corn.

The system administrator can restrict the activities of a user by providing a restricted version of the shell (rsh) or by allowing the user to run only at specific script by making a suitable entry in the .profile file.

A system administrator has a lot of responsibilities (duties) as well as privileges. Thus, a system administrator is also known as a superuser.

4.3.1 Becoming a Superuser

The username of the superuser's account is root. Many administrative tasks and their associated commands require superuser status.

There are two ways to become a superuser. The first one is to log into the console directly as root. The second way is to execute the command su after logging in under another username, as shown below.

```
$/bin/su password: #

#not echoed>
```

Once the su command is entered, the system prompts for the superuser's password. If the correct password is entered then the system displays a pound sign (#) indicating that the user has now become the superuser successfully.

4.4 UNIX SECURITY

Computer time as well as information (data) stored in computers are valuable resources that require protection. System security is a very important part of any multiuser system like Unix. The two important factors that are considered to have required system security are

. Keeping unauthorized people away from gaining access to the system.
. Keeping even authorized users away from tampering with system files or other user's file.

Unix has many different types of security measures built in to it. The most basic, oldest and the one that is being very widely used even these days is the password security method. Another method is to restrict the capabilities of specific users by making them use a restricted version of the standard shell. By using a command, called the erypt command important and highly sensitive individual files can be made secure.

4.4.1 PaSSWOrd Security: /ete/passwd and fete/shadow Files

I Passwords are nothing but keys by using which, one can enter into a system. The method of providing security by assigning unique passwords to every individual user is one of the oldest as well as the most widely accepted practices even these days. No one is permitted to use a system without knowing and recording complete information about him or her. These days password security is managed with the help of two special files called /etc/passwd and etc/shadow files.

/etc/passwd File The complete information about the user is obtained and recorded in a separate file called the / ete/ passwd file during the opening of an account for the user. This file can be read by any user but can be edited only by the superuser. Each user will have a line pertaining to him or her on this file. Each of these lines are made up of seven fields that are separated by colons (:). The following line shows the general format of each line followed by the explanation of each field in the line.

User:password: UID: GID: comment: home: shell

Below is given the purpose of each of the fields.

. user: This field holds the login name of the user.
. password: This field holds the password in the encrypted form. An asterisk (*) in this field indicates that one cannot login to the system with this user or login name. A x in this field indicates that the encrypted password is on a separate file called the shadow file that will be present in the same directory-the etc directory.
. UID: This field holds the numerical lD of the user.
. GID: This field holds the default group ID of the user. A user maybe a member of many groups.
. comment: This field holds some sort of descriptive comments (typically the user's full name, address, etc,). Certain commands like fi nger use this information. This field is also called as GECOS field.
. home: This field holds the absolute pathname of the user's home directory.
. shell: This field holds the information about the user's shell or command interpreter. This is the shell into which a user enters into as soon as he or she logs in.

For example manju:x::123:456: manjunath:/home/nsm:/usr/bin/sh
Here, the user is manju. The x in the second field indicates that the password is present on a separate file called the / etc/ shadow file. The UID is 123 and GID is 456. The fifth field gives more information about the user manju. As already mentioned earlier, commands such as finger use this information. The login or home directory of manju is /home/nsm and his shell is /usr/bin/sh.

The / etc/ shadow File As explained above, the second field of a /etc/ passwd file holds a user's password in the encrypted form. Past experience has shown that these encrypted passwords can be easily obtained by any user and can be decrypted. So encrypted password is not stored in /etc/passwd file. It is stored in /etc/shadow file which is accessible by only the superuser. /etc/shadow file also contains some additional information about password aging.

4.4.1 Account Management

Any user who can make use of the system is called an account. The system must therefore be provided with information of the user so that he may be allowed to work on the system. It is the systems administrator who can allow a particular user to work on the system.

Adduser

In most of the modem system a shell script called adduser is available which helps to add the user.

Syntax

The general syntax to use the adduser shell script is

```
adduser <login-name> <name> use-id > <HOME-directory- name>
```

Example

```
# adduser ravi ravindra 181'/usr/ravi
```

Deleting an Account from the System

delusr

Again a special shell script called deluser is available that helps to delete a user directly without making changes through vi.

Syntax

The general syntax is

```
del <login-id> <Ye/No> <HOME-directory-name>
```

Example

```
Deluser manju /usr/manju
```

Which ever account has the login name ravi will be deleted from the system and he won't be allowed to login.

4.5 Making a File System

When the system is booted the root file system is always loaded and this file system consists of directories like /etc, /dev. /bin etc. But other file system like the /usr may have to be created and mounted on the root file system under the root directory.

- To make a file system the mkfs program stored under the /etc directory is used.

- To run the /etc/mkfs program two parameters are to be passed to the program. One is the storage device name on which the file system is to be made and another is the capacity of the storage device in KB. A file system is however made only if the/dev directory contains the device driver to the storage device.

Syntax

Thus the general syntax for mkfs program is

```
/etc/mkfs /dev/ <device -name> <device-capacity>
```

Example

```
/etc/mkfs/dev/fd096ds15360
```

The device mentioned is the floppy disk with size 360KB having 96 tpi with 15spt.

4.6 Mounting the File System

As said making the file system is not the last thing to do to be able to use it. After a file system is made it has to be mounted to the root file system at the required location. To mount a file system the /etc/mount program is used. This program takes in two parameters one is the device name on which the required file system to be mounted is stored and another is the directory name in the root file system where the file system is to be mounted.

Syntax

The general syntax to use the /etc/mount program is

```
/etc/mount <device-name> <direcotory-name>
```

Example

```
/etc/mount/dev/fd096ds15/usr/ravi
```

The above eg. shows how a file system created on the device /dev/fd096ds15 is mounted onto the directory /usr/ravi under the root file system.

4.7 Unmounting a File System

The file system that was created in the above eg. was on a floppy disk in say A drive. To unmount the file system one can't just remove the disk from the drive. It has to be formally unmounted from the root file system using the /etc/umount program which takes in the device name on which the file system to be demounted resides.

Syntax

The general syntax for the /etc/umount is

```
/etc/umount <device-name>
```

Example

```
/etc/umount/dev/fd096ds15
```

4.8 Backing Up and Restoring Files

The files stored on the system may belong to not just one user but many and if any user deletes it then others using the file will also loose it. Moreover files on such a sensitively organized system as UNIX have higher chances of being lost or corrupted and hence file backing up is necessary.

The most widely used backing up programs are the tar and the cpio.

4.8.1 The tar Program

tar is a short for tape archive and helps to take back up of specified files on a specified storage device.

Syntax

To back up files the general syntax is

```
tar -[cvr]f <device-name> <path-name>
```

-c	⇒	option informs tar to back up from the beginning of the storage device. So if any file already exists it will be overwritten.
-r	⇒	option informs tar to append the files at the end of the storage device. So if any file exists on the disk it will be retained.
-v	⇒	option makes tar work verbosely i.e. give information of the file while backing up.
*f	⇒	option is obligatory and is used just before the device name. Thus the f option is used to specify the device name.

Example

```
tar -cf /dev/fd096ds15 /usr/ravi/*.c
```

This will back files with .c as their extension names from the directory /usr/ravi and store it on to the device/dev/fd096dsl5 from the beginning of the device.

Syntax

To restore files the general syntax is

tar-xf<device-name> <file-name>

-x (option specifies to tar that restoration is to be done. x is a short for extract.

Example

tar-xf/dev/fd096ds15*c

will restore the files ending with .c from the device /dev/fd096ds15

4.8.2 The cpio Program

cpio is a short for copy input-output This program works in two modes. One in output mode to take backups and another in input mode to restore files. Syntax

Syntax

To back up files the general syntax is

cpio-o<device-name>

Example

cpio-o/dev/fd096ds15

This makes the cpio program work in the output mode because of the -o option and files are backed up on to the specified device. But which files ??. The files names of the files to be backed up are to be specified from the keyboard.

Hence the following commands will also backup the respective files.

$ ls | cpio > /dev/fd096ds15

$ find /usr/manju -name "*c" –print | cpio -o > /dev/fd096ds15

In the first case the output of the list command will be given to the cpio program which will be backed up while in the second the files ending with ‘.c’ 'from the directory /usr/ravi will only be backed up.

- To restore files the general syntax is
  cpio-i <file-names> <device-name>

Example

$ cpio -i "*.c" /dev/fd096ds15

The above command makes the cpio work in the input mode because of the -l option and restores files whose name end with *.c* from the specified device.

4.9 Maintenance of the File System

UNIX provides a program called fsck which checks the file system if it is correct and if anything is wrong it sets things right. The fsck program resides in the /etc directory.

Syntax

The general syntax to run the fsck program is

```
$ /etc/fsck [-yn] [directory-name]
```

The fsck program works interactively with the user. As the fsck program finds any error with the file system before clearing 'it the program ask for confirmation from the user.

4.10 SECONDARY STORAGE MANGEMENT

Whatever the amount of disk space one has, always there will be a need for some more. One can attribute such a need for a number of reasons like the following.

1. Many number of files that are not so important as well as not being used, might be just present there and need either to be deleted or preserved as backups.
2. The allotted space for a file system or a user may be being underutilized and someone else may be in need of additional space. Such a situation needs the space to be re-appropriated.
3. The space is available yet it is not possible to create new files due to the non-availability off inodes.
4. Addition of new users.
5. New applications that are space thirsty and so on.

Before the space is actually re-appropriated, it is necessary to know how much space is available or how much space is actually being utilized, Unix has commands such as df and du using which, one can know the free space available and space utilized respectively.

4.10.1 The df Command

The df (disk free) command is used to find the amount of disk space available on a file system. If a particular file system is not specified then this command reports the free space available on all the file systems. Given below is an example that displays the free space available on all the file systems.

```
$df
I        (ldevl dsk/cd1) , 2965 blocks 51007 inodes
I    usr (ldevldsk/cOdOs214534 blocks203028 inodes

$
```

The above output shows that this system has two file systems. It gives the information about the number of free blocks and free inodes. Each block is usually of a 512 byte. However, on some systems, block size may be 1024 bytes.

4.10.2 The du Command

The du (disk usage) command is used to find out how much disk space has been used by each sub-directory as well as each file under a current directory, By default, this command generates the reports in terms of the blocks used by present working director.

```
$du

10      ./manju
$
```

Here 10 blocks are used by manju directory.

4.11 Shutting down the system

System administrator has to shutdown the system, if required at the end of the day. Abrupt switching of the system will lead to problems such as file system corruption. In order to switch off the system systematically shutdown command is used. Shutdown notifies all the users to logout. It unmounts all the file systems and writes all the file system information back to the disk. The command is shutdown.

```
# shutdown
```

Chapter5

Unix Editors

5.1 INTRODUCTION

The Editor is an utility program that we use to make changes to the contents of the file. Text editor is an editor designed to deal with files containing strings of characters in a particular character set. Moreover you can view what you have already, before making changes. With a text editor you can interactively move through a file, inspecting things and deciding if you want to change them, and instruct the text editor to make the changes permanent.

There are two types of editors

1. Interactive editors

Example vi

2. Non Interactive editors Example sed (stream editor)

5.2 Interactive Editor
The interactive editors fall into two categories, namely Line Editor, and Screen Editor.
5,3 Line Editor
In Line Editor you can give commands to the editor to perform various operations on the lines such as insertion of new lines, changing of new lines, copying or moving of lines to different
locations in the file. Two line editors that are available are ex and ed.
5.4 Screen Editor
Screen Editors can be called as display editors and visual editors. In screen editors you can display a portion of a file, you can move the cursor anywhere in the screen.
The file is first copied into temporary scratchpad area, called buffer and the changes are made in that buffer. When the changes are over you must save the file. Any changes that you made during the editing session do not appear in the original file until you save the file.
5.5 Invoking vi
·
We call the editor by the vi command:
Example $vi unix.doc

and your file unix. doc is read into the editor buffer, the terminal screen is cleared, and a portion of the file is displayed on the terminal screen. This happens if unix.doc exist else it is created.

This is often termed as a window of the file. The cursor can be moved around on the screen to indicate where to make the changes.

$vi +15 unlx.doc
The above command will display the initial window and the curser is positioned at the beginning of fifteenth line.

If power goes off while you are entering into the file then the unix system will save the file automatically. You can retrieve file using -r option.
Example $vl -r unix.doc
Getting out of vi
If you want to save your changes and exit the editor then type.

:wq or :x or *zz*
·
If you don't want to save your changes then just type
:q!<enter>

5.5.1 vi Modes
There are three types of modes in vi, namely
·
Command or Edit Mode
·
Insert Mode
·
Replace Mode

Command Mode
·
The moment you give vi command you are placed into command mode. In command mode you can move the cursor, delete text, change text, perform searches, and exit vi to

return to the shell prompt. In the command mode we cannot add text. We need to use Insert Mode to add text.

.
To go to insert mode from command mode press the key i and to go to command from insert mode press <Esc> key.

5.5.2 Basic Cursor Movements

You can move the cursor in different ways.

.
It can be moved left and right along the screen a character at a time, a word at a time or to the beginning or end of a line.

.
It can be moved up and down the screen.

.
It can directly go to specified line or string to search for.

.
Apart from the arrow keys you can use certain other keys to move about in the command mode. These are as described in the following table.

Key	Position after the command
left arrow or h	One character to the left
right arrow or I	One character to the right
up arrow or k	One line up
down arrow or j	One line down
W or W	One word forward
b or B	One character backward
Return or +	Beginning Of The New Line
-	Beginning of the previous line
0	Beginning of the current line
$	End of the current line
H	Beginning of the screen
M	Beginning of middle line on screen
L	Beginning of last line on screen

. All the above commands except the last 5 can be prefixed with a number to repeat it that many times.

~ Example

3w moves the cursor 3 words forward.

5.5.3 Scrolling to the Text

If your file goes beyond the screen, you can use the following commands to scroll the screen forward and backward.

Ctrl F	Next page (Scroll Forward)
Ctrl B	Previous page (Scroll Backward)
Ctrl D	1/2 Screen forward
CTRL U	1/2 Screen backward
Ctrl E	Displays next line off the screen
CTRL Y	Displays the previous line off the screen

5.5.4 Insertion Mode

To enter some lines of text, you have to go to the insert mode. The following commands can be used to go to insert mode.

i	Insert before the cursor.
i	Insert in the beginning o the current line.
a	Insert after the cursor.
A	Insert at the end of the current line.
0	Insert a line after the current line.
a	Insert a line before the current line.

You can use the arrow keys to go to any place of the entered text.

5.5.6 Editing an Existing File

Invoke vi with the name of the file to be modified.

Example $vl unlx.doc

Use the arrow key to position the curser and use the commands i, I, a, A, 0 or 0 to insert text in the file. The following commands are used to delete text in the edit (command) mode:

x	Delete the current character.
dd	Deletes the current line.
nx	Deletes n characters from the current character.
ndd	deletes n lines from the current line.
dw	Deletes a word at current character position.
ndw	Deletes n words from current character position.

The following commands are used to copy the, text in edit mode.

yy	Yanks 1 line. Take the cursor to the line after which you want the yanked lines to be copied and press p
ny	Yanks n lines.

k,ltm	Copies a range of lines i.e k to I after the m th line.

5.5.7 Replace Mode

.

If you want to replace a character with another character then you have to go to the replace mode.
The following command is used to replace one character with another character.

rc Place the cursor below the character to be replaced and press r. Then enter the character that you want to replace with.

R< text> Replaces characters by keyed in text until Escape key is pressed.

5.5.8 Undo Changes

If you want to undo the changes then you can use u command in edit mode.

5.5.9 Searching

The command "/pattern" is used to locate a pattern starting from the current curser location.

Example /utility

5.5.10 Replacing Text

.

After searching a string you may want to replace the original string. The following commands are given by pressing the shift key and: (colon). This places the curser at the bottom of the screen. Use command "s" for substitution.

Syntax

The syntax structure for this is

```
:s/unix/zenix/
```

The above command will search for a string(unix) and will substitute that string with new string 'zenix'. It will replace the first occurrence of the string only.

.

If you want to replace the string for all the occurences then use g option. The command

```
:s/unix/zenix/g
```

will replace the search string (unix) for all the occurrences with the replace_string (zenix).

.

5.5.11 Assigning Line Numbers

.

If you want to assign line numbers for the statements then use the following command
:set number

.

If you don't want the line numbers then use the following command
:set nonumber

.

If you want to directly go to nth line then use the following command
:n <enter>

5.5.12 Transferring part of a file to some other file

.

If you want to save some portion of the current file to another file then use the following command
: <range> w newfile

Example
If you want to transfer lines from 10 to 20th line then the command for this is as follows

```
:10,20w newfile
```

newfile is the file to which you are transferring the lines.
These lines will overwrite the contents of newfile. If you want to append these lines to the newfile then use the following command

```
:10,20w» newfilename
```

6.1 INTRODUCTION

The shell sits between you and the operating system, acting as a command interpreter. It reads your terminal input and translates the commands into actions taken by the system. The shell is analogous to command.com in DOS. When you log into the system you are given a default shell. When the shell starts up it reads its startup files and may set environment variables, command search paths, and command aliases, and executes any commands specified in these files.

The original shell was the Bourne shell, sh. Every UNIX platform will either have the Bourne shell, or a Bourne compatible shell available. It has very good features for controlling input and output, but is not well suited for the interactive user. To meet the latter need the C shell, csh, was written and is now found on most, but not all, UNIX systems. It uses C type syntax, the language UNIX is written in, but has a more awkward input/output implementation. It has job control, so that you can reattach a job running in the background to the foreground. It also provides a history feature which allows you to modify and repeat previously executed commands.

The default prompt for the Bourne shell is $ (or #, for the root user). The default prompt for the C shell is %.

Numerous other shells are available from the network. Almost all of them are based on either sh or csh with extensions to provide job control to sh, allow in-line editing of commands, page through previously executed commands, provide command name completion and custom prompt, etc. Some of the more well known of these may be on your favorite Unix system: the Korn shell, ksh, by David Korn and the Bourne Again Shell, bash, from the Free Software Foundations GNU project, both based on sh, the T-C shell, tcsh, and the extended C shell, cshe, both based on csh. Below we will describe some of the features of sh and csh so that you can get started.

6.2 Features of the UNIX shell

- **interactive**
  The shell presents a prompt and waits for the user to enter a command. After the return key is pressed, the shell processes the command and when the command is finished, the shell re-displays the prompt. This process continues until the user exits the shell, by typing exit or by pressing ctrl-d, at which time the user is logged out of the UNIX host.
- **runs programs in the background**
  Non-interactive tasks which do not require keyboard input or display output can be run in the background as a separate task. The user continues working with other inter-active programs. Examples of this are printing or sorting files.
- **input/output redirection**
  Programs designed to use the standard input device (keyboard) and standard

output device (display) can have their input and output devices redirected to other devices. For example, a program which generally reads from the keyboard can be redirected to read from a file instead. A program which writes its output to the display can be instructed to redirect its output to the printer or a file.

- **programs can be chained or connected together via pipes**
  The output of one program can be fed directly into another program by connecting the two programs via a pipe. This allows a user to create powerful new commands by chaining existing commands together.
- **wild-card characters are supported in filenames**
  The handling of files is simplified by using wild-card characters to match files which match particular patterns. Common operations can thus be performed on a group of common files using a single command.
- **script files**
  A number of commonly used commands can be stored in a file, which when executed, runs each command as though it has been typed from the command line. A sequence of commands can be executed by executing the file which contains the command. This simplifies repetitious commands.
- **environment variables**
  The user can customize and control the behavior of the shell by using special variables that the shell supports. The variables can also be used by application programs and shell script files to control their behavior. An example of a shell variable is the prompt string used to display the shell prompt sign ($).
- **macro language**
  The shell supports a simple language definition for creating shell script files. These can be used to generate very complex command sequences.

A shell script (shell program) is a file containing a set of commands to be executed (run) by the shell in sequence as it reads the file. In its simplest form, a shell script is simply a way to save a set of commands that are often executed, such as the initialization commands for your login session that are stored in the .login script file, so you don't have to re-type the set every time you want to run it. Instead, you simply run the script that contains all the commands. Strictly speaking, everything that you can put into a shell script can also be executed interactively by typing on the command line, although the looping constructs can be cumbersome. This gives you the chance to test out the syntax of various shell constructs.

The shell provides tools to make shell scripts more powerful, even full-fledged programs. Shell programming is organized around the concepts of "substitution" and "flow-of-control".

- Substitution is used to manipulate values within the script. It involves using the values of variables as part of a command; and taking the output of a command to be the new value of a variable.
- Flow-of-control refers to common programming constructs such as loops and if-then-else statements that are used to control which statements in the script are executed in which order, often depending upon the value of a variable; or to cause repetitive execution of a set of commands with different variable or input values.

6.2.1 FEATURES OF C SHELL

Some of the features of the C shell are listed here:

Customizable environment.
Abbreviate commands. (Aliases.)
History. (Remembers commands typed before.)
Job control. (Run programs in the background or foreground.)
Shell scripting. (One can write programs using the shell.)
Keyboard shortcuts.

6.3 Special characters in BOURNE Shell

Some characters are special to the shell, and in order to enter them, one has to precede it with a backslash (\). Some are listed here with their meaning to the shell.

< > Output redirection**.**
| Pipes.
***** Matches any string of zero or more characters.
? Matches any single character.
[ ] Matches any set of characters contained in brackets.
{ } Matches any comma-separated list of words.
; Used to separate commands.
& Also used to separate commands, but puts them in the background.
\ Quote the following character.
$ Obtains the value of the variable.
' Take text enclosed within quotes literally.
` Take text enclosed within quotes as a command, and replace with output.
" Take text enclosed within quotes literally, after substituting any variables.

6.4 SHELL VARIABLES

A variable in the shell can be any name containing up to 20 letters and digits which starts with a letter or an underscore and first one being letter. Variables names are case sensitive so DAY, Day and day are all different. It is not good practice to define more than one variable with the same name but different case. In fact it has become standard for shell variables to be given all upper case characters.

A variable is a name or placeholder used to store one or more values. Variables give a high level of generality to everyday interactive shell usage, and are heavily used in shell programming.

In interactive shell a variable must be referenced (used) with a dollar sign. This dollar sign tells the shell that what follows is a variable, and that the shell should make the appropriate substitution. There are three types variables. They are system variables, local or user defined variables and Read only variables.

6.4.1 System variables

They are set either during the boot sequence or immediately after logging in.. They are also known as environment variables. Some of them are **PATH, HOME,IFS,MAIL,SHELL** and **TERM**.

PATH varible holds a list of directories in a cetain order. Colon(:) is used to separate different directories.. The current value of this varible can be seen by using echo command.

```
$ echo $PATH

/bin:/usr/bin:/usr/manju/bin:/usr/nach/bin:.

$
```

This Bourne Shell variable will describe the directories that will be searched looking for the program or command that you want to execute. The Bourne Shell looks in several directories for a file that has the same name as the command that you entered. The PATH variable controls this search path. Normally, the first directory searched is the current working directory. If the program is not found, the search continues in the /bin and then the /usr/bin directory and so on. If the program is not found in one of these directories, the Bourne Shell reports that the program or command can't be found (or executed).

HOME

The first Bourne Shell variable that we will look at is the HOME variable. By default, the home directory is the current working directory after you login. The system administrator determines your home directory when you establish an account and places that information in the /etc/passwd file. When you login, the BourneShell gets that pathname and assigns it to the HOME variable.

When you enter a cd command with no argument, the utility takes the name of the directory from the HOME variable and makes it the current working directory. If you change the HOME variable to another directory pathname, the utility will make the new directory the current working directory.

```
 $echo $HOME
 /user/nach
 $cd
```

```
$pwd

/user/nach
```

IFS

This is the internal-field separator Bourne Shell variable. You can always use a space or tab to separate characters on the command line. When you assign the IFS variable to another character, you can also use this character as the field separator.

MAIL

The MAIL variable contains the name of the file that the mail (and mailx) utilities use to store your mail. Usually, the absolute pathname of this file is /usr/mail/name, where name is your login name.

Example:

```
$MAIL=/usr/mail/nach
```

PS1

This is the Bourne Shell prompt which lets you know that the shell is waiting for you to give it a command. The default BourneShell prompt is a dollar sign ($). The shell stores the prompt as a string variable in PS1. When you change the value of this variable, the appearance of the prompt will change. When you are working on several different machines, it might be useful to have the prompt be the name of the machine you are working on.

```
$pwd
/user0/nach
$PS1='nach'
nach:
```

Notice that prompt is now nach:

PS2

This variable is called the secondary prompt. If the command is not completed on one line and must be continued on the next line, the prompt for that continued line is PS2. The default is >. This prompt indicates that the Bourne Shell is expecting you to finish the previous command line.

Sample Session:

```
$echo 'demonstration of prompt string
>2'
demonstration of prompt string
2
$PS2='Continue? '
$echo 'demonstration of
Continue? prompt string 2'
demonstration of
prompt string 2
$
```

Notice how the secondary prompt was changed to "**Continue?** ".

SHELL

This variable contains name of the users shell program in the form of absolute path name.Value of the SHELL variable is known by using echo command.

```
$echo $SHELL
/bin/sh

$
```

6.4.3 Local Variables

These variables are local to a particular user's shell. As these variables are defined and used by specific users, the are also called user defined variables. Shell allows you to have variables, just like any programming languages. Variables do not need to be declared. To set a shell variable, use

VAR=value

and to use the value of the variable later, use

$VAR

or

${VAR}

The latter syntax is useful if the variable name is immediately followed by other text:

```
$COLOR=yellow
$echo This looks $COLORish
$echo This seems ${COLOR}ish
```

prints

```
This looks
This seems yellowish
```

There is only one type of variable in shell strings. This is somewhat limited, but is sufficient for most purposes. By default value of a shell variable is null string.

```
$echo $x

$
```

Here x is not assigned value. It contains null string.

Variables are concatenated by placing them adjacent to each other.

```
$DAY=Tues
 $d1=day
$ echo $DAY$d1

Tuesday
```

Let us see the way in which variables are accessed should be examined. While it is true that a variable must be accessed with a $ in front, that is not the end of the story. It should be remembered form the discussion of quoting special characters that a text string enclosed in double quotes will allow the shell to make variable substitutions while a string in single quotes will not. Consider the following example,

```
$ name="Ram Kumar"
$ echo ``His name is $name"
His name is Ram Kumar"
$ echo 'His name is $name'
His name is $name
```

where the single quoted text string is echoed verbatim. Another complication that can arise when accessing variables as part of a longer string is, for example, the variable DAY is assigned the value ``Tues'', and then an attempt is made to make the string ``Tuesday'' containing the variable DAY the following problem arises:

```
$ set DAY=Tues
$ echo $DAYday
Dayday: Undefined Variable
$
```

What has happened here is that there is no whitespace between the variable name and the rest of the string. When the shell parses the command line, it interprets everything following the $ and up until whitespace as the variable name. Since DAYday was not defined as a variable (given a value) the shell displays an error message. To alleviate this problem, curly braces ({}) can be used. If a curly brace follows directly after a $, everything inside of the closed curly brace pair will be considered a variable name.

```
$ set DAY=Tues
$ echo ${DAY}day
Tuesday
```

It is also important to note that when assigning a value to a variable, the shell considers everything between the equals sign and the next whitespace to be the value of the variable. It is thus important that no whitespace (other than what is meant to be used) is introduced during a variable assignment. If whitespace is to be used, the appropriate quotation characters must be used.

```
$person = "Krishna"

$echo \$person
  $person
  $
```

In the above example the variable person is preceded by a dollar sign ($) but the dollar sign has a backslash (\) ahead of it. The backslash has the effect of canceling the special meaning of the character following the backslash. In this case, the special meaning of the dollar sign is ignored and the substitution is not done.

```
  $echo '$person'
  $person
  $
```

The single quote marks (') causes the characters between the marks to be taken as literal. The shell makes no attempt to interpret the meanings of these characters. The shell passes these characters on with no substitution.

A shell varibale can also be used to replace a command.

```
$count=`wc file1`

$ $count

23 456 23456 file1

$
```

6.4.4 readonly variables

If a variable has had a value assigned, and you want to make sure that its value is not subsequently changed, you may designate a variable as a readonly variable with the following command:

readonly variable

From this point on, variable cannot be reassigned. This ensures that a variable won't be accidentally changed.Readonly variables that can be read by the user, but not written to or changed explicitly.

```
$readonly x=5
```

The value x cannot be changed to any value

6.4.5 Export

When a shell executes a program, it sets up a new environment for the program to execute in. This is called a subshell. In the Bourne shell, variables are considered to be local variables; in other words, they are not recognized outside the shell in which they were assigned a value. You can make a variable available to any subshells you execute by exporting it using the export command. Your variables can never be made available to other users.

6.5 How to execute a shell program?

1. Create file using text editor, enter the statements and save it
2. Make the shell script executable by using the command $ chmod u+x filename
 or by another command **$ sh filename**.

Shell scripts are often written to handle some of the more tedious tasks that a user encounters on a regular basis. A simple shell script called shell_ex is shown in the following example.

```
# This is a very simple shell procedure
 echo "It is created with the basic echo command "
 echo "and three other very basic commands "
 echo
 ps
 echo
 who
 echo
 ls
```

With the exception of the line that starts with hash marks (#), the script is a list of simple Unix commands. Almost any line starting with a hash mark will be ignored by the shell and hence indicate programmer comments. The powerful feature of shell scripts over simply writing the commands on a command line is that scripts can contain many types of safety, logging, and other features to provide a worry free and organized working environment.

6.6 Inserting Comments

Comment entries can be included in a shell script by prefixing statements with the # symbol. The shell, on encountering #, ignores what follows in that line.

6.7 Command Substitution

Recall the usage of pipes in joining commands by sending the standard output of one command as standard input for another. Another way of using more than one command in a single command line is through command substitution.
Suppose a user wants to display the following message on the screen:

The date is (output of the date command)

To do so, the user can enter the following command:

echo "The date is 'date' "

The command date is enclosed in single backward quotes. The shell first replaces the enclosed command of the output, and then executes the entire command.
Command substitution can also be used to store the output of a command in a variable. For example,

ctr='ls *.c | wc-1'

The variable ctr will now contain a count of the number of files in the current directory whose names end in c.

6.8 Reading Input Into a Shell Variable

Shell scripts can also take input from stdin. User input allows shell programs to be fully interactive, which will add to their generality. The read statement is used for this purpose. The read statement will cause the shell to accept an input string to be read in until a newline character is encountered. The shell removes all white space (with the obvious exception of newline characters) but does not interpret filename or character substitutions. The shell takes the input string and breaks it up into substrings which are surrounded by white space. The read statement has the following form:

$ **read Variable1 Variable2 Variable3 ... VariableN**

where each variable is an expected substring. If, for example, the user is expected to input two filenames, the read statement would be

$ read file1 file2

If the user accidentally enters three files rather than two, the second variable will be assigned the last two file names. If on the other hand, there are more variables than substrings entered, the unmatched variables will be null, or empty. For example a shell script is created withname reads which contains the following statements.

```
 echo "Please enter three strings"
read a b c
echo $a $b $c
echo $c
echo $b
echo $a
```

Values of three variables are to be entered with spaces in between.

6.9 SHELL ARITHMETIC USING expr

All variables in UNIX are string variables. Therefore to carry out arithmetic operations, the variables must be converted to numeric format. This conversion is performed using the expr command. expr is used to carry out basic arithmetic operations including modulo division on integers.

The operators used for arithmetic operations are:

+	Plus for addition
-	Minus for subtraction
*	Star or Asterisk for multiplication. This operator must be despecialized or escaped as \ *. Otherwise the shell reads it as a wildcard character.
/	Slash for division
%	Percentage for modulo division.

An important limitation of expr is that it is capable of performing integer calculations only. For precise and important computations, any of the UNIX calculators such as be can be used.

```
$ expr 2 + 3
5
$
$ expr  10 - 10
0
$
```

```
$ expr  15 / 3
5
$
$ expr 10 / 2 5
0
$
$ expr  20 \* 3
60
$
$ expr 10 % 3
3
$
$ a=10
$b=5
$ expr $a + $b
15
$ echo “expr $a +  $b “
expr 10 + 5
$echo `expr  $a + $b`                back quote executes
15
$
```

Hierarchy of arithmetic operators is as below:

Priority	Operators
1	/ * %
2	+ -

The hierarchy can be overcome by using parentheses, since top priority is given to evaluation of expression within parentheses. The parentheses must also be preceded by a backslash \ (i.e., \c and \).

expr command can also be used to perform certain string manipulations such as length of a string.

```
$ expr "krishna"

7
$
```

6.10 Escape Sequence

An escape sequence is a two-character (mostly) string beginning with a \ (backslash) character. When the escape sequence is placed at the end of an echo string argument, the command executes accordingly. For example \c places the' cursor in the same line that displays the output:

```
$ echo " Enter the value:\c"
enter the value:$
```

AI! escape sequences are not two character strings. The escape sequence \ 007 is used for a beep sound. Other common escape sequence characters are:

\t	A tab of 8 character positions
\n	A newline character equivalent to pressing Enter.
\007	Sounds the system bell

Example program to find the sum of three numbers

```
Clear
Echo enter three nubers
Read num1, num2, num3
Sum='expr $num1  +  $num2  +  $num3`
Echo sum of three numbers is $sum
```

6.11 Test command

The test utility evaluates expressions and returns a condition indicating whether or not the expression is true (equal to zero) or false (not equal to zero). There are no options with this utility. The format for this utility is as follows:

syntax : **test expression**

Expression is composed of constants, variables, and operators. Expressions will be looked at in greater detail later with some examples. There are a few items that need to be mentioned that apply to expressions. Expressions can contain one or more evaluation criteria that test will evaluate. A -a that separates two criteria is a logical AND operator. In this case, both criteria must evaluate to true in order for test to return a value of true. The -o is the logical OR operator. When this operator separates two criteria, one or the other (or both) must be true for test to return a true condition.

You can negate any criterion by preceding it with an exclamation mark (!). Parentheses can be used to group criteria. If there are no parentheses, the -a (logical AND operator) takes precedence over the -o (logical OR operator). The test utility will evaluate operators of equal precedence from left to right.

Within the expression itself, you must put special characters, such as parentheses, in quote marks so the Bourne Shell will not evaluate them but will pass them to test.

Since each element (evaluation criterion, string, or variable) in an expression is a separate argument, each must be separated by a space.

The test utility will work from the command line but it is more often used in a script to test input or verify access to a file.

Another way to do the test evaluation is to surround the expression with left and right brackets. A space character must appear after the left bracket and before the right bracket.

[expression]

6.11.1 Test on Numeric Values

Test expressions can be in many different forms. The expressions can appear as a set of evaluation criteria. The general form for testing numeric values is:

int1 op int2

This criterion is true if the integer int has the specified algebraic relationship to integer int2.

The valid operators (op) are:

-eq	**equal**
-ne	**not equal**
-gt	**greater than**
-lt	**less than**

-ge	**greater than or equal**
-le	**less than or equal**

$a=10;b=5

$ test $a –gt $b

Here ; is used to separate the statements. Here expression $a –gt $b is evaluated it is true it returns 1.

$a=5;b=10

$[$a –gt $b]

Here expression $a –gt $b is evaluated, it is false and returns 0.

6.11.2 Test on Character Strings

The evaluation criterion for character strings is similar to numeric comparisons. The general form is:

string1 op string2

The operators (op) are:

string1 = string2	**true if string1 and string 2 are equal**
string1 != string2	**true if string1 and string2 are not equal**
string1	**true if string1 is not the null string**

$a=lmn;b=lmnr

$[$a = $b]

Here the exprsion $a = $b is evaluated. It is false because string are not eqal. So it returns 0(false).

6.11.3 Test on File Types

The test utility can be used to determine information about file types. A few of them are listed here:

-r filename	**true if filename exists and is readable**
-w filename	**true if filename exists and is writable**
-x filename	**true if filename exists and is executable**
-f filename	**true if filename exists and it is a plain file**
-d filename	**true if filename exists and it is a directory.**
-s filename	**true if filename exits and it contains information (has a size greater than 0 bytes)**

Example:

```
$test -d new_dir
```

If new_dir is a directory, this criterion will evaluate to true.
If it does not exist, then it will be false.

6.12 if then-fi

The format for this construct is:

```
if expression
then
     commands
 fi
```

The if statement evaluates the expression and then returns control based on this status. The fi statement marks the end of the if, notice that fi is if spelled backward.

The if statement executes the statements immediately following it if the expression returns a true status. If the return status is false, control will transfer to the statement following the fi.

```
if [  $a –eq 5  -o  $a –le 10 ]
  then
    echo " it is a valid number"
```

Here if variable a value is equal to 5 or less than equal to 10 then condition is true and echo command is executed.

A shell program to find whether user entered name is a directory or not.

```
# To find whether directory or not
echo " enter name:"
read name
if [ -d $name ]
echo " entered name is a directory"
fi
echo "entered name is not a directory"
```

In the above program –d name finds whether it is a directory or not. If it is true then the echo statement after if is executed. If –d name returns 0 then echo statement after fi is executed.

6.13 If-then-else-fi

Syntax

```
 if  condition
then
           commands
else
           commands
fi
```

In the above statement the commands mentioned after the else will be executed if the specified condition is false.

```
if  [ x –gt  0]
      then
         y='expr $x + 1`
         echo "y has the value $y
else
         echo x is either 0 or negative
if
```

Here if x is +ve then the value of y is incremented by 1otherwise it prints the message x is either 0 or negative.

6.14 if-then-else if-else-fi statement

If a variable is to be tested for a number of values then the if-then-else if-else-fi statement is to be used.

The general syntax is

```
        if condition1
         then
                        commands
                else
                if  condition2
                 then
                        commands
                else
                        commands
                 fi
        fi
```

A menu program to select the choice of food.

```
# to demonstrate if-then-else if-else-fi  statement
Clear
Echo " 1. South Indian food
Echo " 2. North Indian food"

Echo " enter your choice  ( 1 or 2):"
Read choice
If test choice –eq 1
Then
   Echo "you have selected south Indian food"
Else

If [ $choice –eq 2  ]
Then
          Echo" you have selected north Indian food"
    Else
          Echo " "invalid choice"
      fi
fi
```

6.15 if-then-elif -else-fi statement

multilevel if-then-else statements can be written using elif command. elif is equivalent to else + if. The elif construct combines the else and if statements and allows you to construct a nested set of if then else structures.

```
        if Condition 1
         then
```

```
                    commands

          elif  condition 2
           then
                    commands
          elif condition3
          then
                    commands
                else
                             commands
           fi
```

A shell script to identify the type of a file.

```
echo " enter name:"
read name
if [ -d $name ]
echo " entered name is a directory"
elif [ -f $name]
 then
echo "entered name is file"
elif [ -r $name]
 then
echo " Entered name is readable file
elif [ -w $name]
 then
echo " Entered name is writable file
elif [ -x $name]
 then
echo " Entered name is executable file
else
  echo  "no such name exists"
fi
```

6.16 CASE STATEMENT

The case construct works like C's switch statement, except that it matches patterns instead of numerical values.   A chain of if-else statement can also be used in such situations. However, if-else becomes cumbersome when the number of nested levels increases. At such times the case statement can be used to select from several alternatives. Its syntax is

```
case value in
       patternl) command

                     command;;

pattern2) command
```

```
            command;;
 patternN) command
             command;;

          *) command

             command;;


esac
```

The value is a string; this is generally either a variable or a backquoted command. The value is compared against the patterns until a match is found. The shell then executes all the statements up to the two semicolons that are next to each other. Next, control is transferred beyond esac.

The case statements start with keyword case and end with the keyword

esac, reverse of case. The default *) pattern gets executed when no match is found.

Case cannot handle relational and file tests, but only string tests. It also effective when string is fetched through command substitution. *Case patterns (or labels) can be in any order. Case labels are not limited to a single character or single digit. It can also be a string. More then one case label can be combined in a single statement using the or (I) operator. The default case is optional. If default case is not included and no case is satisfied, control is transferred beyond esac. * The default case if included must be the last choice of case statement.

```
# An example with the case statement
# Reads a command from the user and processes it
echo "Enter your command (who, list, or cal)"
read command
case "$command" in
        who)
                echo "Running who..."
                who
                ;;
        list)
                echo "Running ls..."
                ls
                ;;
        cal)
```

```
                    echo "Running cal..."
                    cal
                    ;;
          *)
                    echo "Bad command, your choices are: who, list, or cal"
                    ;;
esac
```

```
# Script to demonstrate case command
clear
echo Enter a number  (0-9)"
 read number

case $number  in


0)echo Zero ;;
1)echo One ;;
3)echo Three ;;
4)echo Four ;;
5)echo Five;;
6)echo Six ;;
7)echo Seven ;;
8)echo Eight;;
9)echo Nine
*) echo Enter no 1to 9 ;;
esac
```

```
# a program to demonstrste arithmetic operations
clear
echo Enter a choice(1-4)"
 read choice
echo " enter two values"
read a, b

case $choice  in
```

```
        case  1)  result = `expr $a + $b``;;

  case   2)  result = `expr $a - $b`;;

  case   3)  result = `expr  $a */ $b`;;

  case   4)   result = `expr $a /  $b`;;

   *)    echo “choice not correct”;;

esac
echo The result is:$result
```

```
# To show case patterns with multiple values
clear
echo "Enter a number from 1 to 9"
read num
case $num in
1 | 3 | 5 | 7|9) echo Odd Number ;;
 2 | 4|I 6|8) echo Even Number ;;
                  *) echo A Number ;;

esac
```

In the above example, when the input number is 1 or 3 or 5 ,7 or 9, the in first case pattern statement will be executed and when the number is 2 or, 4 ,6 or 8, the second case pattern statement is executed. If more than one digit number or letters are entered, case treats them as strings and compares them.

```
# Script to show pattern matching using meta-characters
clear
echo "Enter a single character please"
read char

case $char in
[A-Z]) echo You entered a Capital letter ;;
[a-z]) echo You entered a lowercase letter ;;
```

```
 [0-9]) echo You entered a digit ;;
?) echo Your entered a special symbol ;;
*) echo you entered more than one character ;;

esac
```

In the above example, the first three case levels check for upper case, lowercase letters and digits respectively. The ? is a wildcard character that represents a single character. * represents the default case. Patterns can also contain strings separated by comma within curly braces { }. For example x {yz, lm, dfg} will expand to xyz, xlm, or xdfg.

The above example displays the menu as shown. It displays the list of files if input value is 2, shows process status if choice is 3, current users for value 4, current working directory when choice is 5 and exits when choice is 6. If choice is anything else, the "Invalid option" message is displayed.

6.17 LOOP-CONTROL STRUCTURES

Looping in a program allows a portion of a program to be repeated as long as the programmer wishes. This can be for a specified number of iterations (or loops), or it can be until a particular condition is met. For instance, a programmer might want to repeat a particular operation on every file in a particular directory. Rather than rewrite the section of the program that carries out the operation over and over for each file, the operation can be written once and iterated as many times as required. Loop control also allows for programs that are more general. Rather than having to pre-specify how many iterations are required for a particular task, the programmer uses conditions to control the iterations. The Bourne shell provides a rich variety of loop control constructs. Depending on what is needed the programer can chose the construct that best suit his needs.

The three types of looping constructs available are:

1. **The while loop**
2. **The for loop**
3. **The until loop**

6.17.1 The while Loop

The while loop is one of the most common and widely used loop control structure. The general. syntax of while command is:

```
While control command
 do

Command1
command2
-
-
done
```

while, do and done are keywords. The commands in the while loop between do and done keywords are repeatedly executed as long as the condition remains true (or exit status of control command is 0). Once the exit status becomes false (or 1), control passes to next statement after the loop (after done).

The following script segment counts backwards from 10 to 1:

```
number=10
while [ $number -ge 1 ]
   do
   echo $number
   number=`expr $number - 1`
   done
```

```
# Script to add 10 numbers
clear
echo -e "\n Enter an integer."
i=0
sum=0
while [ $i -le 10 ]
do
echo enter number
read num
sum=`expr $sum + $num
i=`expr $i  +  1`
done
echo “sum of ten numbers is:$sum”
```

```
# Shell script to count the number of vowels in a string. clear
echo Enter a string :"
```

```
read  str

len='expr length  $str`
echo "length of string = $len"
 i=1
actr=0
ectr=0
ictr=0
 uctr=0
octr=0
othr=0
 while [ $i -le $len ]
do
ch= 'expr $str | cut -c $i'
case $ch in
a|A) actr='expr $actr + 1';;
 e|E) ectr='expr $ectr + 1';;
```

```
i|I) ictr='expr $ictr + 1';;
o|O) octr='expr $octr + 1';;
 u|U) uctr='expr $uctr + 1';;
*) othr='expr $othr + l' ;;

esac
i='expr $i + l' done
done
echo "Number of a or A = $actr"
echo "Number of e or E = $ectr"
 echo "Number of i or 1 = $ictr"
echo "Number of 0 or 0 = $octr"
 echo "Number of u or U = $uctr"
echo "Number of other characters = $othr"

# To check a given number for prime number
 clear
echo "Enter the number m, m > 1:"
read m
i=2
flag=1
while [ $i -lt $m –a $flag –eq 0  ]
do
  if [ 'expr $m % $i' -eq 0 ]
then
flag=0
fi
done
if test $flag –eq  0
then
echo $m is a not a Prime number
else
echo $m is a Prime number
```

6.17.2 FOR loop

The **for:in:do** construct is used to repeat a group of commands once for each item in a provided list. The construct has the following form:

```
for VARIABLE in LIST
  do
  COMMAND LIST
  done
```

where VARIABLE is a variable name assigned each item in LIST during the execution of COMMAND LIST. What happens is as follows: the variable takes the value of the first item in the list and then executes the command list; after the command list has been passed through, the variable is assigned the value of the second item in the list, and so on, until the list has been exhausted. Searching for the occurence of a string in a file could be done like the following:

```
# A script to look for the occurence of a string in a file
# Usage: match [string] [file]
#
for word in `cat $2`
  do
  if [ ``$word'' = ``$1'' ]
  then
    echo ``Found $1 in file $2''
  else
    :
  fi
  done
```

where the first parameter passed to the script is string and the second is the file thought to contain the string. Notice that after the else statement a colon has been placed. This is the null command which tells the computer to do nothing. This is clearly an unnecessary section of the script and was only added to demonstrate the use of the null command.

If the list is omitted from the for statement, each parameter in the command line will be passed to word.

```
for each $m 1 2 3 4 5 6 7 8 9
    echo "value of $m"
end
```

This will print 1 2 3 4 5 6 7 8 9

6.17.3 UNTIL command

Another construct which is very similar to the while loop is the until construct. The construct works in precisely the same manner with the one exception that it repeats a series of commands until a condition is met(as long as the condition is false). The until loop looks like

```
until condition
   do
   commands
   done
```

To count backwards, as in the above example, the until loop would be used as follows:

```
number=10
until [ $number -lt 1 ]
   do
   echo $number
   number=`expr $number - 1`
   done
```

6.18 Exiting Loops Early

In the three looping constructs examined above, the loop would continue to execute until a specific condition was met (a boolean condition in the while and until loops, or completion of a certain number of loops in the for loop). There are times however when it would be beneficial to exit a loop early. The Bourne shell provides two methods for exiting from a loop early, the break command, and the continue command.

6.18.1 Break command

It skips remaining statements and exits from the loop. The break command will exit from the current loop and program control will be resumed directly after the loop construct exited from.

```
for name in *
do
  if [ ! -r $name ] ; then
    echo "Cannot read $name, quitting loop"
    break
  fi
  echo "Found file or directory $name"
done
```

In the above example loops over files and directories, quits if one is not readable. Here name in * means all the files in the current directory is taken and it will be passed one by one to name variable.

6.18.2 continue command

The second method for exiting a loop early is the continue command. This command behaves very much like the break command with the exception that it jumps out of the current loop and places control at the next iteration of the same loop structure. It is used skip remaining statements and return to top of loop.

```
for name in *
do
  if [ ! -f $name ] ; then
    continue
  fi
  echo "Found file $name"
done
```

In the above example loops over files, skips directories

When used in case structures, the break command is a pleasant alternative to the exit command for handling unwanted choices as it allows for control to be passed to another section of the program rather than exiting the program entirely.

6.19 POSITIONAL PARAMETERS

As in the case of C program information can be conveyed to a shell program in the form of command line arguments. With shell scripts these command line arguments are called positional parameters. Obviously, this method of passing on the values to a shell script is a non-interactive method.

Arguments submitted with a shell script are called positional parameters as the first argument is passed on as parameter no 1, second argument is passed on as parameter no 2 and so on. Actually these parameters are stored in certain special shell variables. There are nine such variables that capture and hold values given in a command line. These are $1, $2, ..., $9. The $1 variable holds the first argument, the $2 variable holds the second argument and so on. As the arguments are assigned as values to the special variables $1, $2, $3 and so on, depending upon their physical positions in the command line they are called positional parameters.

For example consider a shell script total which sums all the values supplied as command line arguments.

```
#This program demonstrate command line arguments
clear
sum=`expr $1 + $2 +$3 + $4 + $5`
echo sum of values: $sum

$ sum 3 4 57 8 9 23
104
$
```

Here 3 4 57 8 9 23 are positional parameters. They are stored in the variables $1,$2,$3,$4,$5, $6 and $0 contains the command sum.

This example uses the first command-line argument instead of asking the user to type a command

Store the following in a file named case2.sh and execute it

```
# An example with the case statement
# Reads a command from the user and processes it
# Execute with one of
#         sh case2.sh who
#         sh case2.sh ls
#         sh case2.sh cal
echo "Took command from the argument list: '$1'"
case "$1" in
        who)
                echo "Running who..."
                who
                ;;
        list)
                echo "Running ls..."
                ls
                ;;
        cal)
                echo "Running cal..."
                cal
                ;;
        *)
                echo "Bad command, your choices are: who, list, or cal"
                ;;
esac
```

I

6.19.1 The $0 Variable

$0 is a special shell variable that holds the parameter number 0 (zero), the program name

6.19.2 $#, $* and $@ Variables

In addition to the already mentioned special variables, there are three mon special variables used by the shell. They are $#, $* and $@. The $# variable holds a count of the total number of parameters, that is, arguments, The $* variable holds the *list of all the arguments.* It may be observed that $# variable is similar to argc and the $* variable is similar to argv[] in C. Like $*, the $@ variable also holds the list of all the arguments present in the command line. However when these variables $* and $@ are used within quote marks, the contents of $* is considered as a single string whereas in the case of$@ each of the arguments is independently quoted and considered as independent string arguments.

6.19.3 Understanding Positional Parameters

Let us modify the script total which displays the contents of all the special variables and some of positional parameters. The study of this script and its output gives a better understanding of the special variables as well as the positional parameters.

```
#This program demonstrate command line arguments
clear
sum=`expr $1 + $2 +$3 + $4 + $5`
echo sum of values: $sum
echo Program name is :$0
echo The number of arguments are $#
echo The arguments are $*
 echo  The first argument is $1
 echo  The second argument is $2
   echo  The third argument is $3
   echo  The fourth argument is $4
    echo  The fifth argument is $5
    echo The sixth argument is $6
```

The output of a trial run of the above program total is as follows.

```
sum of values:104
Program name is :total
The number of arguments are 5
    The arguments are 3 4 57 8 9 23
```

```
 The first argument is 3
       The second argument is 4
         The third argument is 57
         The fourth argument is 8
          The fifth argument is 9
          The sixth argument is 23
```

The number of arguments that can be given at command line is 9. This is the limitation. One can overcome this by using shift command. We will discuss that in the next section.

6.19.5 The set Command

It is used to assign values to positional parameters. As discussed in the earlier sections of this chapter, arguments of a script get automatically assigned to positional parameters as their values. It should be noted that values to positional parameters cannot be assigned using the equal to (=) operator. However values to positional parameters can be assigned using the set command as shown in the following example.

```
$set friends in need are friends indeed.
$
```

The execution of the above command line assigns friends to the parameter $1, in to the parameter $2, need to the parameter $3 and so on. These assignments can be verified by using the echo command as shown below.

```
$echo $1 $4$6
 friends are indeed.
$
```

Displaying date in a required format Given below is a shell script using which today's date is displayed in the format, *day date month year* using the set command, date command and positional parameters.

```
Set `date`
Echo $1 S3 $2 $6
```

The output upon the execution of the script is
Tue 27 jun 2007

6.19.6 Positional Parameters and Excess Arguments In certain situations one may give more than nine arguments in a command line. This results in excess values to be assigned to the nine positional parameters. An example that illustrates the assignment of excess arguments to positional parameters follows.

```
$set In the above example an attempt to display the value of variable p q
$echo $1 $2 $3 $4 $5 $6 $7 $8 $9 $10 $11
```

```
In the above example an attempt to display the In1 the1
$
```

In the above example an attempt to echo the value of $10 has resulted in **In** ($l's value and a zero) and the value of $11 has resulted in **the1** ($l's value and a 1). Such situations are handled using the shift command.

6.19.7 shift Command

It is known that there are only nine positional parameters and therefore only a maximum of nine arguments can be given in a command line. In case more than nine arguments are given in a command line, no error is indicated. However, the behaviour will be ambiguous as illustrated in the previous section. Such type of situations are handled using the shift command. When used, the shift statement shifts out the values assigned to the positional parameters to the *left* by an integer value mentioned with the shift statement as its argument. Thus the statement $shift 5 moves the parameter values to the left by five positions. Actually it shifts out the first five values and assigns the 6th value to $1, 7th value to $2 and so on. It should be noted that when certain values are shifted using a shift command, shifted values would be lost. An example that illustrates the use of a shift command is given below. This example has to be studied along with the previous example.

```
$ shift 5
$echo $1 $2 $3 $4 $5 $6 $7 $8 $9 $10 $11
attempt to display the value of variable p q
$
```

When used without any argument the shift command shifts the contents of the positional parameters by just one position to the left.

6.19.8 THE $? VARIABLE

Whenever a command, that is, a program, is run it may either get executed successfully and yield a result or it may not get executed successfully.
Whenever, a command is successfully executed the program returns a 0 (zero). However, if a command is not executed successfully a value other than 0 (zero) will be returned. Logically, a 0 (zero) is considered as true and a non-zero value is considered as *false.* These returned values are called exit status values and will be available in one of the shell’s special variable $?.
An exit status value available in $? is normally used in decision making in shell programs.

For example, let sample.sh be a non-existent file and an attempt is made to list its contents as shown in the following example. The exit status will be a non-zero value.

```
$cat sample.sh
cat: sample.sh: No such file or directory
$echo $?
1
$
```

```
$ a=5
$test a-gt 0
$echo $?
0
$
$grep 'swetha' student.lst
$echo $?     # search failed
1
$
```

Let us display the existing file sample.

```
Cat sample
Ram
Krish
John
$ echo $?
0
$
```

Here execution of command was successful so it $? Value contains 0. It means true**.**

6.20 UNIX SYSTEM COMMUNICATION

Since UNIX is time sharing **s**ystem communication between the different users is very much required. We discuss some of the commands that facilitate communication.

6.21 THE WRITE COMMAND

The write command can be used by any user to write something on some one else's terminal, provided the recipient of the message permits communication.

Eg: Suppose user4 needs to communicate with user5.

```
$ write user5
Hello Kumud, Today UNIX class is there, Please attend.
```

ctrl D should be typed to indicate the end of the message.

On executing this command the message will be transferred to user5. There will be a beep on his terminal followed by the message.

Now user5 may respond to the message by using write command.

Conditions for the write Command

1) The recipient must be logged in, else an error message will be displayed.

The recipient must have given permission for messages to reach his or her terminal. This is done by saying at the $ prompt

$ mesg - y

2) To deny write permission to any terminal can be done by saying.

$ mesg - n.

A super user can write to any terminal, perspective of whether mesg has been set to -y or -no

It is better to know who all are logged in and who allow messages to be written to their terminals and then we can run the write command.

6.20.2 THE WALL COMMAND(Write to all)

The wall command can only be used by the superuser (or) System Administrator. The superuser has full freedom to write to any user on the network. Wall command enables the super user to 'write to all' irrespective of whether the users have given write permission to their terminals or not. Suppose if the UNIX system is going to be shutdown within 5 minutes time, the super user has to warn all concerned users and notify them to save whatever they are working on.

The wall command is shown below.

```
# wall
System shutting down in 10 minutes. You are required to save the contents.
(ctrl d)
```

All users who are logged in will hear a beep and see a message flashed on their screen as follows.

```
Broadcast Message from root(tty01) on UNIX. Jul 19 09:25 2007
System shutting down in 10 minutes. You are required to save the contents.
```

Command prompt is shown as # and not $, as that is the prompt the superuser works at.

6.20.3 MAIL COMMAND

It is used to send the mail to the users. Mail command is used for that purpose

$ mail username

For example

```
Mail student1
Subject:  Test at 12.30 pm
Make it convenient to attend the test
Ctrl d
$
```

If the user has not logged in also mail will be stored in the mailbox. Usually when user logs in message will be displayed that you have a mail. One can read the mail by giving once again the command mail.