

APPENDIX

A

UNIX

1 : OVERVIEW

1.1 : Introduction

Curriculum design and development in many universities in India is quite often not in tune with changing requirements. It is perceived by many as lagging behind, not keeping pace with developments and changing business needs. However, this is not entirely true. In fact, many universities revise their syllabi in a continuous manner, though it may not be at the rate of student or parental expectations. For example, an entry-level software engineer in the IT business sector needs to know a lot more than what he did a few years ago. Current requirements are such he/she has to be familiar with a multiuser operating system environment, must have the capability to program in such an environment, and needs to know about many more domain areas like database management, intelligence systems etc.

One has to also understand the consequent difficulties in implementing the changes in curriculum based on the changing knowledge requirements. This has been one of the factors why planners do not look into curriculum revision more frequently than expected. Computer Science, in general, and specially the core courses like algorithms and operating systems have been the critical victims of this phase-lag syndrome. Quite often, these courses are taught much later than expected in the chronological chart of the four-year degree program in engineering. One reason could be the entry of students from different branches of study into the field of computer science and, more specifically, into software engineering. But current requirements in and around these developments are such that they have to be catered to. One such essential requirement is that all engineering students must possess working knowledge of an operating system such as UNIX, and a high-level language programming language such as C, C++ in that environment. The faster they get used to this, the more productive they will be in their chosen field. The primary motivation for introducing UNIX, without a formal course on operating systems, is based on this requirement, but is also to enable fresh engineers to get onto UNIX terminals for practice as early as possible. Expectations are high and constraints are too many in delivering these. We have to prioritize even the trade-offs, as they are far too many. In this book, therefore, we have tried to bring out some of the salient features of UNIX and to map your understanding of the practicing environment through this. The transition from theory to practice may sound abrupt, theory into but it is achievable as illustrated attempt. Readers are therefore expected to bear with a lack of completeness at times and they are referred to the literature given at the end for meeting their appetite, if the present course does not cater fully to their interest.

1.2 : Scope and Objective

As described in the preceding section, it is always difficult to put practice before theory in any domain. UNIX, however, may be an exception like many other software tools. Accordingly, this part of the book on UNIX is organized into three chapters. Chapter 1 brings out the expectations and addresses some of the issues in brief, as regards the operating systems as a prelude to learning UNIX. Chapter 2 presents the overview of the UNIX operating system and provides the necessary background for practicing UNIX commands. Chapter 3 deals with practice sessions that will allow students to gain hands-on experience, even as they learn the concepts and the theory underlying the same. The chapter is primarily organized to enable students to practice these as comfortably as possible.

1.3 : Operating Systems

Present-day computer systems have the capability to gather data, perform computations, store information, communicate with other computer systems, and generate output reports. Some of those capabilities are best implemented in hardware, while others are best manifested in software. An operating system (OS) is the software that takes the raw capabilities of the hardware and builds a more practical platform for program execution. The operating system manages hardware resources. It also offers services for accessing these hardware resources and enables higher-level abstractions such as files, directories and processes.

A typical computer system has five main components: the hardware, the operating system, systems programs, application programs and users, as shown in Fig 1.1. Users are expected to interact with the system through the system programs or through the application programs.

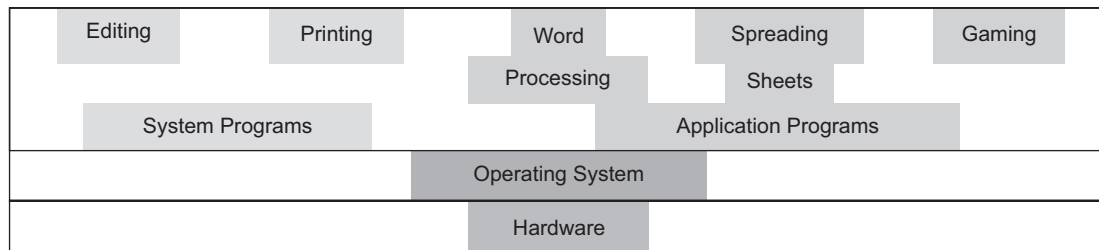


Fig. 1.1 Basic computer system

Actual work is done by the hardware. It includes the memory, the **central processing unit (CPU)**, and the input and output (I/O) devices. The operating system provides a set of services to programs. The communication between the user and the operating system has been implicitly through the programs. Systems programs are actually a set of utility programs supplied with an operating system to provide basic services for users. Examples of systems programs include a window manager for a **graphical user interface (GUI)**, a command interpreter, and programs to rename, copy or delete files. Application programs provide the computer with the functionality the users require. Examples of application programs include, for example, a software for preparing the tax details, a financial planner, a word processor, and a spreadsheet etc as shown the Fig. 1.1.

1.3.1 Hardware

The CPU is the heart and brain of a computer system. It contains a number of special-purpose registers, an **arithmetic logic unit (ALU)** and the control logic necessary to decode and execute instructions. Memory and

Input-Output (I/O) devices are usually connected to the CPU by means of a bus. The operation of the CPU is based on the control-the instructions the CPU fetches from memory, while the I/O devices are commanded by the CPU.

The operation of a CPU can be described in terms of a simple loop, where each time through the loop one instruction is executed. An instruction is first fetched from the memory location specified by the special register called the **Program Counter**. The instruction is then placed in a special register called the **Instruction Register**. The program counter is subsequently incremented so that it points to the next instruction to be executed. The instruction is then decoded to determine what action is to be performed as specified by the **opcode** (operation code) bits. The machine architecture defines which bits contain the opcode. Depending on the operation to be performed, the value of one or more operands is normally fetched from memory.

The operations specified in the opcode is finally performed. There are usually five basic categories of operations: **Movement**: Moves a value from one location (in the registers or memory) to another. **Computation**: sends one or more operand values to the ALU for performing the computation. **Conditional Branch**: If the branch condition is true, resets the program counter to point to the branch address. Note that the branch condition is always true for an unconditional branch. **Procedure call**: The program counter current value is first saved and then reset to point to the beginning of a procedure. The branch instruction specifying the saved program will allow the program to return to the current point of execution at the end of the procedure. The saved program counter may then be stored in a register, in memory, or on the stack. **Input/Output**: Transfers information of an input or output operation between the CPU and I/O device.

Traps and interrupts are events that disrupt the normal sequence of instructions executed by the CPU. A trap is an abnormal condition detected by the CPU, indicating an error. Examples of trap conditions include dividing by zero, trying to access or execute an instruction with an undefined opcode, or trying to access a nonexistent I/O device. An interrupt is a signal sent to the CPU by an external device, typically an I/O device. A CPU will check interrupts only after it has completed the processing of one instruction and before it fetches a subsequent one.

The hardware must also support multiple modes of execution in order to enable the operating system with privileges not granted to application programs. There are two most commonly supported modes of execution. These are known as **kernel (or supervisor) mode** and **user mode**. A system can enter kernel mode from user mode in three possible ways:

1. A special instruction called a **supervisor call** (SVC), or a **system call**, which is similar to a procedure call except that it sets the system state to kernel mode. Unlike procedure call instructions; supervise call instructions are not supplied with a branch address. The instruction's operand is a number that serves as an index into a vector similar to the interrupt vector. The branch address is then determined by the contents of the memory location pointed by the supervisor call operand. The vector is located in the memory controlled by the operating system so that the switch into kernel mode coincides with a jump to an operating system entry point.
2. Traps, and
3. Interrupts.

Like supervisor calls, the switch coincides with a jump to a kernel entry point. Application programs cannot change the system into kernel mode and continue to execute in their own code. On UNIX systems, one user, called the **superuser**, is given special access privileges. The superuser may read or write any file and kill any process. The superuser should not be confused with kernel mode execution. Application programs running as the superuser are granted extraordinary access rights by the kernel, but those programs do not run in kernel mode; they must use supervisor calls to request operating system services.

1.3.2 Operating System Tasks

- **Process Management:** A **process** is an executing program. Associated with a process are its code, its data, a set of resources allocated to it, and one or more “flows” of execution through its code. The operating system provides supervisor calls for managing processes and must manage the allocation of resources to processes. If multiple processes can exist simultaneously, the operating system must be capable of providing each process with an appropriate virtual environment in which it can run.
- **Memory Management:** At a minimum, memory must be shared by an application program and the operating system. On more sophisticated systems, memory can be shared by a number of processes. The operating system must manage the allocation of memory to processes and control the memory management hardware that determines which memory locations a process may access.
- **File System Management:** Computers process information. That information must be transmitted, processed and stored. A file system object is an abstract entity for storing or transmitting a collection of information. The file system is an organized collection of file system objects. The operating system must provide primitives to manipulate those objects.
- **Device Management:** A computer communicates information through its input and output devices. Processes access those devices through the operating system-supervisor calls provided for that purpose. The operating system attempts to manage those devices in a manner that allows them to be efficiently shared among the processes requiring them.

1.3.3 Operating System Types

A batch mode operating system: A batch operating system offers minimal functionality since it does not have to worry about the complications of sharing resources with multiple processes. On multi-programmed batch systems, jobs are read into a job pool stored on a disk. When one job is unable to execute because it is waiting for an I/O operation to complete, another job may be permitted to run. Concurrently executing processes greatly increases operating system complexity as regards the sharing of computer resources.

Time-shared operating system: It allows for interaction between user and process. In batch systems, all data is supplied at the time the program is inputted. This is fine for a program such as printing weekly progress reports of an Examination & Evaluation information. It is different, however, to deal with programs that must interact with the user. The operating system must support an environment that allows programs to respond to user inputs in a reasonable amount of time. The operating system has not only to share resources among the various processes, but it must also create the illusion that the processes are running simultaneously. It does this by shifting execution rapidly among all the active processes.

Real-time operating system: It is designed to provide execution support within the time constraints. It is often used as a control device in a dedicated application such as controlling scientific experiments, medical imaging systems, industrial control systems, and some display systems. It is characterized by well-defined fixed-time constraints. Real-time systems may be either *hard* or *soft* real-time. Hard real-time means that secondary storage is either limited or absent, data is stored in short-term memory, or read-only memory (ROM). It conflicts with time-sharing systems and it is not supported by general-purpose operating systems. Soft real-time has limited utility, for instance, in industrial control of robotics. However, it is extremely useful in applications like multimedia, virtual reality, etc. requiring advanced operating-system features.

1.4 ! Growth

The growth of operating systems can be traced back to the early sixties. However, over the past few decades, a variety of approaches have been used in the construction of operating-system kernels. The modern trend

is towards smaller kernels in which the application programmer is provided with enhanced levels of flexibility. This has been realized by separating the mechanism and policy, wherever possible. It has then provided simple abstractions that mimic the functionality of the hardware, rather than high-level abstractions that are excessively general. Many believe that the monolithic kernel design, a characteristic of early UNIX systems, tended to be the guiding factor for modern operating-system architecture. Its basic premise is to abstract over the machine hardware to enable high-level abstractions such as processes and file systems, which equip application developers to build systems with greater efficiency and portability. Although such high-level abstractions are suitable for many systems, there are also systems for which they are not only inadequate but also sometimes inappropriate. This would be acceptable if the developers of these systems could simply ignore the abstractions that are not suitable and attempt to create their own from scratch. However, since the operating system defines a high-level virtual machine, applications are effectively forced to use the abstractions provided. The problem with high-level abstractions is that they are quite often inefficient and excessively restrictive for many purposes. Readers can refer to the literature for further inputs on the design features of operating systems in general, and UNIX in particular, as it falls beyond the scope of this book.

2 : OPERATING SYSTEM AND SHELLS

2.1 : Introduction

This chapter introduces you to the UNIX Operating System and gives a brief introduction to UNIX shells so that you can easily migrate to the practicing sessions discussed in next chapter. Computer software can be roughly divided into two kinds: the system programs and the application programs. The system programs manage the operation of the computer itself, while the application programs solve the problems of the users. The most fundamental of all system programs is the operating system whose purpose is to control a computer's activities.

There are two main functions that an operating system performs: first, it presents the user with an equivalent of a virtual machine that is easier to program than the underlying hardware. Secondly, the operating system acts as a resource manager, by keeping track of who is using which resource, to grant resource requests, to account for usage and to mediate conflicting requests from different programs and users.

Operating systems have evolved through the years. Various operating systems differ in how they do their job and what additional features they offer. The UNIX operating system is one such operating system developed by Bell laboratories.

2.1.1 History of UNIX

UNIX is the happy outcome of the proverbial rags-to-riches story. What is now heralded as the most powerful and popular multiuser operating system had a very humble beginning in the premises of AT & T's Bell laboratories. The origin of UNIX can be traced back to 1965, when a joint venture was undertaken by Bell Telephone Laboratories, the General Electric Company and Massachusetts Institute of Technology.

The aim was to develop an operating system that could serve a large community of users and allow them to share data, if need be. This enterprise was called Multics, for Multiplexed Information and Computing Service. Even after much time, resources and efforts had been devoted to the project, the convenient, interactive computing service failed to materialise. This led Dennis Ritchie and Ken Thompson, both of AT & T, to start afresh on what their mind's eye had so illustriously envisioned. Thus, in 1969, the two along with a few others evolved what was to be the first version of the multiuser system UNIX. Though this was not tapped to the fullest, it had all the trappings of a truly potent multiuser operating system. This system was christened "UNIX" by Brian Kernighan, as a very re-minder of the ill-fated Multics. The UNIX Operating System was re-written in a high level language, C, designed and implemented by Ritchie.

2.1.2 Features of UNIX

The UNIX system has many useful features, the most important of which are its

- multitasking capability
- multiuser capability
- transportability
- large selection of powerful UNIX supplied programs
- communications and electronic mail
- library of applications software

Multitasking Capability

Multitasking means performing more than one task at a time. When you print a file and while it is printing you start editing another document, you are performing multitasking operations. Multitasking lets you

simultaneously perform tasks formerly performed sequentially. This is managed by dividing the CPU time intelligently between all processes or tasks.

Multiuser Capability

A multiuser system permits several users to use the same computer simultaneously. In a multiuser system, the same computer resources—hard disk, memory, etc.—are accessible to many users. A terminal, in turn, is a monitor and a keyboard, which are the input and output devices for the user.

A user at any terminal can not only use the resources of the computer but also the peripherals that may be attached, for example: a printer. One can easily appreciate how economical such a setup is than having as many computers as there are users, and also how much more convenient when the same data is to be shared by all. The heart of a UNIX installation is the host machine, often known as a server or a console.

Transportability

The UNIX system itself is very portable, the reason being that UNIX is written in C. That is, it is easier to modify the UNIX System code for installation on a new computer than to rewrite another operating system from the scratch for the new computer.

The ability to transport the UNIX system from one brand of computer to another has been the major reason for the acceptance of this system. There are two types of portability to be considered: the portability of the UNIX operating system itself and of the application programs. More than 90% of the kernel is written in C and hence can be easily transported to another machine. Also, the application programs written in higher level languages are also portable. Portable applications decrease programming costs.

UNIX System—Supplied Tools

The UNIX system comes with several supplied programs. These programs are divided into two classes; namely, integral utilities and tools, which are discussed below.

Integral utilities

Integral utilities are part of the UNIX system that provide such assistance to the operating system that they are absolutely necessary for the practical operation of a computer with UNIX. One example is the UNIX system command interpreter or the shell program. Without this you could not request your UNIX system to perform any work for you.

Tools

Tools, on the other hand, are programs that are not necessary to the computer's basic operation but provide significant additional value to UNIX. These tools include many application programs. Integral utilities are utilities that let you set up sophisticated automatic procedures to perform a series of tasks that would otherwise have to be performed as many separate actions. UNIX system tools include an extensive “electronic filing cabinet”, word processing, typesetting capabilities, and many other programs.

Communications and electronic mail

UNIX communications include the following options:

- a. Communicating between different terminals hooked into the same computer.
- b. Communicating between the users of one computer with users of another computer (the second computer being of a different brand in the same location).
- c. Communicating between computers of different sizes and types in different locations.

Third party application programs

In addition to application programs supplied by Bell, a library of over 500 UNIX application programs have been developed and sold by various computer manufacturers and software companies. UNIX application programs take advantage of the UNIX's multiuser and multitasking capabilities and can be used by more than one person at a time.

2.1.3 The Structure of the UNIX System

The UNIX system may be functionally viewed as consisting of the kernel, the shell and utilities, as shown in Fig. 2.1. The role of each of these is given briefly in the next section.

The kernel

1. schedules programs,
2. manages data/file access and storage,
3. enforces security mechanisms, and
4. performs all hardware access.

The shell

1. presents each user by a prompt,
2. interprets the commands typed by a user,
3. executes user commands, and
4. supports a custom environment for each user.

Utilities

1. File management (rm, cat, ls, rmdir, mkdir)
2. User management (passwd, chmod, chgrp)
3. Process management (kill, ps)
4. Printing (lp, troff, pr)
5. Program development tools

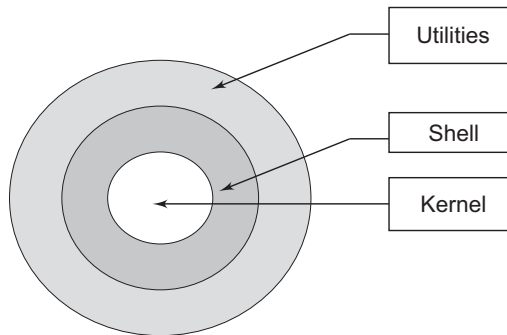


Fig. 2.1 Schematic of basic UNIX system

2.1.4 UNIX Architecture

The UNIX system can be regarded as a kind of pyramid as shown in Fig. 2.2.

At the bottom is the hardware consisting of the CPU, memory, disks, terminals and other devices. Running on the bare hardware is the UNIX operating system. Its function is to control the hardware and provide a system call interface to all the programs. These system calls allow the users to create and manage processes, files and other resources. Programs also make system calls by issuing instructions to switch from the user mode to kernel mode to start up UNIX. So a library is provided with one procedure (these procedures are written in assembly language, but can be called from C) per system call.

In addition to the operating system and system call library, there are a large number of standard programs (which may differ from version to version) such as command processors (shell), editors, compilers etc. Thus,

there are three different interfaces to UNIX: true system call interface, the library interface, and the interface formed by a set of standard utility programs.

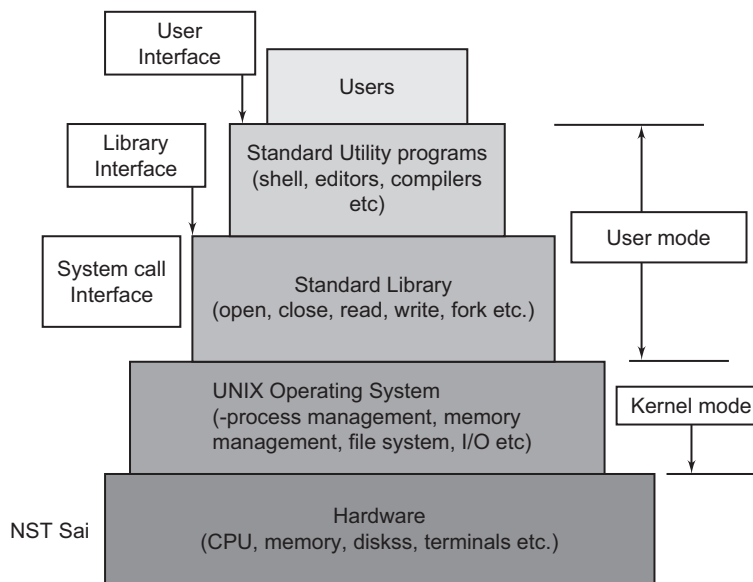


Fig. 2.2 The layered architecture of a UNIX system

2.2 : Unix File System

2.2.1 Introduction

All computer applications need to store and retrieve information. For many applications, the information must be retained for weeks, months or forever. The solution to these problems is to store information on disk or other media in units called files. Processes can then read them and write new ones if need be. A file is a unit of data that is stored on a magnetic disk (or tape). A filename identifies the data uniquely. A collection of files on the disk is called a file system. A directory is a special type of file that contains a list of filenames.

Some operating systems use a flat or one-dimensional file system, where all files reside in a single directory. The UNIX file system allows a hierarchical structure. The programs and data can be organized conveniently since files can be grouped according to usage. Files are managed by the operating system. The operating system design deals with how these files are structured, named, accessed, used, protected and implemented. That part of the operating system that deals with the files is known as the file system.

The UNIX file system is a structure for organizing information. When a process wants to read or write a file, it must first open the file. A file is opened with a `OPEN` call, whose first argument gives the path name of the file to be opened and the second one specifies whether a file is to be read, written, or both. The system checks to see if the file exists and, if it exists, it checks to see if the caller has the permission to access it in the desired way as if that is also permitted. Then the system returns a small positive integer called a file descriptor, to the caller. If the access is prohibited then it returns -1 to indicate an error.

The calls for reading and writing the file use the file descriptor to identify the file. When a process starts up, it has three descriptors available: 0 for standard input, 1 for standard output and 2 for standard error. The first file opened is given the file descriptor 3 and next one 4 and so on. When a file is closed, its file descriptor is

freed and can be allocated on a subsequent open. There are two ways to specify file names in UNIX. The first is using an absolute path starting from the root, and the second is a relative path, which is specified relative to the working directory.

2.2.2 Files

Introduction

A file is a collection of information in the form of data, an application, or documents. When a file is created, UNIX assigns the file a unique internal number called **inode**. The majority of the UNIX versions allow the file name to be of 14 characters in length, though some versions also allow longer file names. All versions of UNIX recognize at least three types of files:

Ordinary files This type of file is used to store data. Users can add data to ordinary files using an editor. Executable programs are also stored as ordinary files.

Directory files A directory file contains a list of files. Each entry in the file consists of two parts: the name of the file and a pointer to the actual file on the disk. Directories behave just like ordinary files except that some of the operations that you would use for manipulating ordinary files do not work for the directories and vice versa.

The directory is a structure for organizing files. In the list of filenames that a directory contains, the names may refer to ordinary files, special files or to other directory files. Since directories may be listed inside or other directories, complex hierarchical structures of ordinary, special and directory files result.

Special files These files are used to reference physical devices such as terminal printers, disks etc. They are read from and written to just like ordinary files but such requests cause activation of the associated physical device.

Special files are divided into two categories: block and character. A block special file is one consisting of a sequence of numbered blocks. The key property of the block special file is that each block can be individually addressed and accessed. A program can open a block special file and directly read the block 124, for instance, without reading blocks 0 to 123. Block special files are used for disks. Character special files are normally used for devices that input or output a character stream.

Some basic commands for file manipulation

- **ls: List files in a Directory** **ls** is used to display file names of the directory in an alphabetical order where you type this command. This command has many options that allow us to see the detailed and variety of information about the file.
- **cat: Concatenate and Print a File** **cat** is a short form for concatenate, which means to join together. This utility is often used to display the contents of a single file, although you can use it to display multiple files in succession. The command is of the form

\$cat filename

and it displays the contents of the **filename** on the monitor, which is the standard output device.

- **cp: Copy an Ordinary File** **cp** command allows you to create a duplicate copy of an ordinary file. The command line format for using the **cp** command is

\$cp srcfile destfile

Here the **srcfile** is the original or the source file and **destfile** is the name of the copy to be created. The important use of this command is to create a backup.

- **rm: Remove an Ordinary File** This command removes one or more ordinary files from the directory, effectively erasing the file. The **rm** utility removes a file by deleting its pointer in the appropriate directory.

In this way, the link between the filename and the physical file is destroyed, so the file can no longer be accessed. The command format is

\$rm filename...

where multiple filenames can be typed separated by spaces.

- **mv: Move (Rename) a file** **mv** command changes the name associated with a file by associating a new filename with a pointer to the physical file in the directory and deleting the link to the old filename. The command is as follows

\$mv oldfile newfile

This changes the name of the file from **oldfile** to **newfile**

- **In: Creating Filename Aliases** The UNIX file system allows more than one filename for the same physical file, which means that it is possible to have aliases for any given file. The command line format is as follows

\$ln oldfile newfile

After the **In** linking the **oldfile** and the **newfile**, both refer to the same file. So to remove a file with more than one links you have to destroy all the links.

2.2.3 File Access Permissions

Introduction

The data in the UNIX system is contained in the form of files. One may want to restrict or permit access to this data by restricting or permitting access to the files containing the data. The UNIX system allows an easy means of controlling the file access that the system users may have to all three types of files (ordinary, directory, special files).

Even for a sole user, this is a necessity so that you don't accidentally damage the contents of your files. So in a multiuser system, restricting access becomes all the more important, so that you can protect your file from being read, written, or erased by other users in the system.

Types of File Access Permissions

UNIX provides three different types of class users and three different modes of file access. These three classes of users and modes of access give rise to nine different kinds of access permissions allowed within the UNIX file system.

There are three classes of system users:

Owner (denoted by u for users) Every file has an owner. The owner is the system user who created the file. The superuser can change the individual ownership of a file if necessary. The owner has full control over restricting or permitting access to the file at any time.

Group (denoted by g) It is possible to have one or more system users own the file collectively in a kind of group ownership. The group is more than one user who may access the file. A system user who is not the file owner may access the file if he is a group owner, but he cannot restrict or permit access to those files unless he himself is the owner.

Other (denoted by o) The other category refers to any other user of the system. These are users who are neither individuals nor group owners.

In addition to classes of file users, there are three ways of accessing a file. The meaning of these access modes is somewhat different for ordinary files than it is for directories. These modes and their meanings are summarized in Table 2.1.

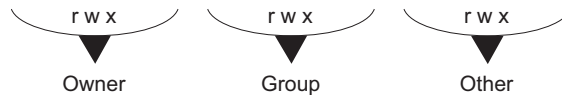
Table 2.1 File Access Modes and Their Meanings

Access Mode	Ordinary File	Directory File
Read	Allows examination of the file contents	Allows listing of files within the directory
Write	Allows changing the contents of the File	Allows creating new files and removing old ones.
Execute	Allows executing file as Command	Allows searching directory

- System users with read permission for files may read (examine) the contents of an ordinary file (for example, by using **cat**), and with read permission for directory can read the contents of a directory (by using the **ls** command).
- System users with write permission for files may write to a file and change its contents (for example, by using an editor); those with write permission for directory can use certain privileged programs to be written on a directory (allows creation and removal of files itself). Write permission is also required to delete a file.
- And finally, the system users with execute permission may execute the file as a command. Executing makes sense only if the file is an executable program or a shell script. The execute permission for the directory file is called search permission, since directories may be searched but not executed like a command. A user must have the execute permission for a directory in order to access the files named in that directory. Thus, even if you had read and write permissions for an ordinary file that was listed in a directory, unless you had an execute permission for the directory containing the ordinary file, the UNIX system would not allow you to read or write the contents of the ordinary file.

Determining File Access Permissions

The three classes of file users (owners, group and others) may be combined with the three types of access (read, write and execute) to give nine possible sets of permissions as shown here:



The presence of permission is indicated by the appropriate letter being in its correct location. The absence of a permission is indicated by a dash (–) in the same place.

For example:

r--r--r-- means this file can be read by all three user classes but cannot be written or executed by anyone and
rw-x----- means this file can be read, written and executed by the owner but is not accessible to groups and others.

Changing File Access Permissions

The **chmod** command (meaning change mode) allows you to change permission modes of one or more files or directories.

\$chmod [who] op-code permission... file...

The *who* argument tells **chmod** the user class and may be any of the following:

- **u** User (owner)
- **g** Group file owner
- **o** Other users
- **a** All users (owner, group and others)

The *op-code* argument represents the operation to be performed by **chmod**:

- **+** Add the specified permission to the existing permission
- **-** Remove the indicated permission from the existing permissions
- **=** Assign the indicated permissions.

The permission argument uses the following abbreviations:

- **r** read
- **w** write
- **x** execute

Now, for example, a command of the type,

\$chmod go-rw letter

means to remove the read and write permissions from the file **letter** for the group and the other users of the file.

2.2.4 The File System Hierarchy

Introduction

The UNIX file system has quite a few significant features. It has a hierarchical structure providing support for directories. These files are expandable, enabling of growing as required. Files are treated as byte streams so that they can contain any characters. Security rights are associated with files and directories enabling read, write, execute privileges for owner, group and others. Files may be shared enabling concurrent access. Hardware devices are treated just like files. Up until now, we have seen only a directory containing entries of ordinary files. But, a directory can contain other directories. This arrangement gives rise to the tree-like branching structure of the file system.

Figure 2.3 shows the structure diagrammatically.

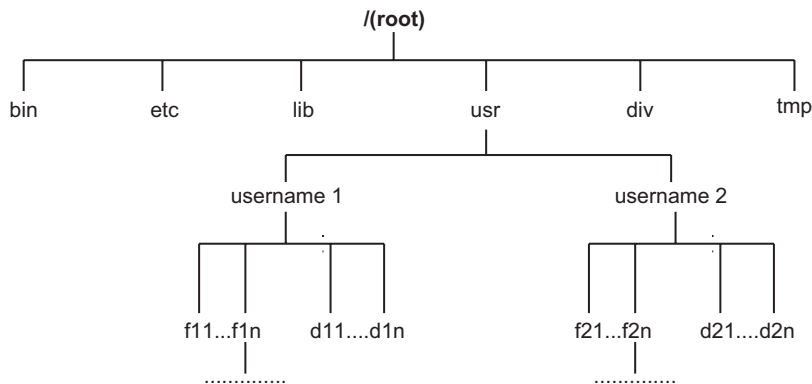


Fig. 2.3 A sample file system

The UNIX file system begins with a directory called root. The root directory is designated by a slash (/). Branching from the root are several other directories (named bin, dev, usr, tmp, lib, etc.). These directories are considered as subdirectories of the root directory and conversely the root is considered to be the parent of these directories. Each subdirectory can point to other subdirectories and to other ordinary files, which is the upside down branching as shown in the Fig. 2.3.

The main reason for having different directories is to keep some files at once together and separated from other files. For example, files that are used by the system might be kept in certain directories such as bin, lib, etc., while files created by the system users may be kept in other directories; for example, in tmp and the subdirectories of usr.

Figure 2.4 shows the representation of the entries for the directory file username 1.

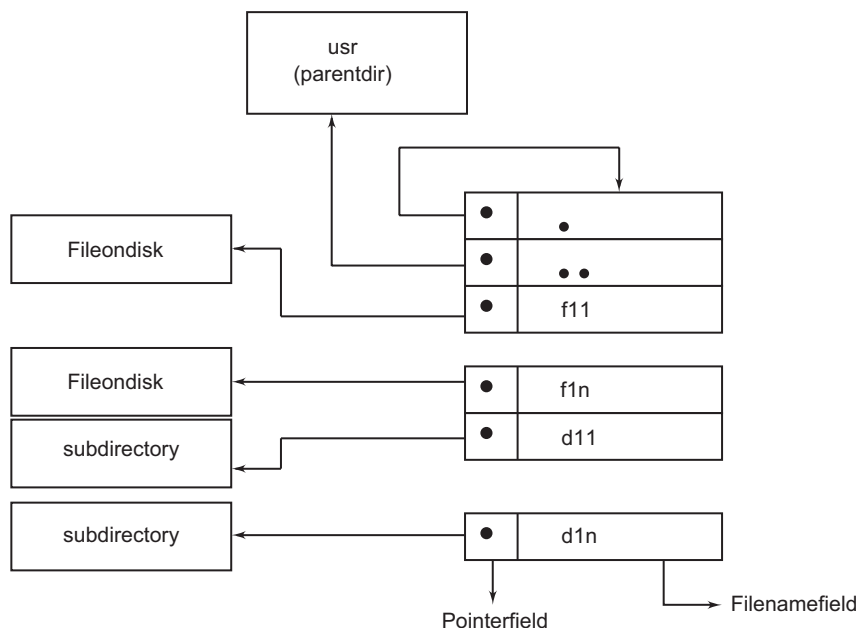


Fig. 2.4 Detail of directory structure for username1

Working Directories and Pathnames

When you first sign on to your UNIX system you begin to work in a particular directory known as your home directory. Your home directory is a unique fixed directory that is assigned when your system administrator establishes your account.

The directory in which you are working at any point of time is known as your working directory or current directory and it need not be fixed; it can be changed. Every file in UNIX has one unique absolute pathname and it begins with the root directory.

Basic Commands for Working with Directories

The following are the basic commands with which you can work when you are dealing with directories

- **pwd: Print Working Directory** The **pwd** command displays the absolute pathname of your working directory and the command line format is

\$pwd

- **mkdir: Create a Directory** You may wish to store files in a particular directory of your own creation. **mkdir** command allows you to create one or more directories. The **mkdir** has the following command line format

\$ mkdir dirname...

The argument **dirname** may be either an absolute or relative path name and you may specify more than one directory name on a single command line. If a **dirname** already exists, the **mkdir** command aborts and does not overwrite the existing directory.

- **cd: Change Directory** To change to any directory in the file system use the **cd** command (for change directory). The command format is simply

\$ cd *pathname*

where **pathname** is either an absolute or relative path name to the desired target directory. Unless you do use a full **pathname**, any filename you specify will be taken to mean a file relative to your current working directory.

- **rmdir: Remove a Directory** If you decide you no longer need a directory, you can use the command **rmdir** to remove it. To remove one or more directories use the command line format

\$ rmdir *pathname...*

The **rmdir** cannot remove a directory unless it is empty. This prevents you from accidentally removing files you wish to keep.

Input/Output in UNIX (Device files)

Like all computers, those running UNIX have I/O devices such as terminals, disks, printers and networks connected to them. Some way is needed to allow programs to access these devices. Although many solutions are available, UNIX integrates them into the file system as special files.

When you log in to the UNIX system, you are assigned a particular terminal from which you control all your processes. This terminal is called your Control Terminal. The UNIX system knows each device by a unique file name; using this file name you can refer to your control terminal or another terminal device.

Each I/O device is assigned a path name usually in **/dev**. For example, the printer may be **/dev/lp**, terminal 1 may be **/dev/tty1** and network may be **/dev/net**. When you need to refer to your terminal, you may use the **ty** command to find out the file name for your control terminal. These special files can be accessed the same way as any other files. No special commands or system calls are needed. The usual READ and WRITE system calls will do just fine. For example, the command

cp file/dev/lp

copies the file to printer causing the file to be printed. (assuming this is permitted). An additional advantage is that the usual file-protection rules apply automatically to I/O devices.

There are three major parts to every file in the UNIX file system. They are:

- the inode
- the data blocks
- the directory

The inode

Each file in the file system is described by a structure called an **inode**. To keep track of which blocks belong to which file, one can associate with each file a little table called an **inode** (short form for **index-node**), which lists the attributes and disk addresses of the file's blocks.

inodes are located in special data blocks (not used for file data) and each 512 byte block can contain as many as eight 64 byte inodes. The **inode** contains all data about the file except the file name and the actual data contained in the file. The disk addresses for the file's data blocks are contained in the **inode** area. The inodes are numbered from 2 (reserved for the root directory,/) through 65,535. i-node 1 is reserved for bad block handling. This identifying number is known as the **inode** number or the i-number.

The first few disk addresses are stored in the **inode** itself so that for small files, all the necessary information is right in the i-node. Hence, whenever a small file is opened the information is directly fetched from disk to the main memory. But for somewhat larger files, one of the addresses in the i-node is the address of a disk block called a **single indirect block**. This block contains additional disk addresses. Similarly if this still is not enough, another address in the **inode**, called a **double indirect block**, contains the address of a block that contains a list of single indirect blocks. Each of these single indirect blocks points to a few hundred data blocks. Similarly, one can also have a **triple indirect block**, and UNIX uses this scheme.

The data blocks

The data blocks are located on the disk and contain the actual data of a file. Each block can typically hold 512 characters. Some UNIX implementation use larger block sizes ranging from 1024 characters and up. Even if the file contains only one character, an entire data block must be allocated to hold this single character.

The directory

A directory contains one or more filenames. Each entry in a directory contains one filename and the **inode** number that points to the **inode** for the file. Directories also have an **inode**.

When a file is opened, UNIX uses the path name given by the user to locate the directory entry. The directory entry provides the information needed to find the disk blocks. The directory system's job is to map the name of the file onto the information needed to locate the data.

Figure 2.5(a) shows a UNIX directory entry, 2.5(b) shows the relationship between the data blocks, the **inode** and a directory that references the file and 2.5(c) shows an **inode**.

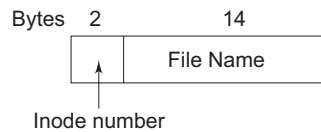


Fig. 2.5(a) A UNIX directory structure

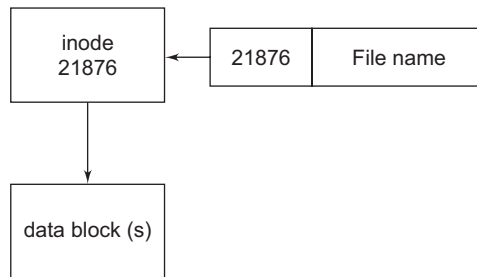


Fig. 2.5(b) The data blocks, inode and the directory

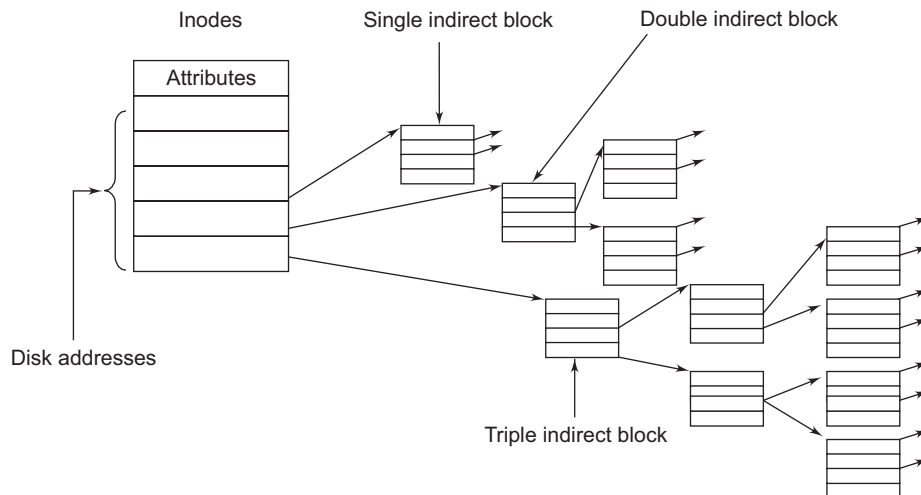


Fig. 2.5(c) An inode

2.3 : Unix Shells

To use UNIX, you must first log in by typing your name and password which the login program reads and checks. This identification is necessary to provide security as, UNIX keeps track of who owns each file. A file may be used only by authorized users. The UNIX password file contains one line for each user, containing the user's login name, numerical user id, encrypted password, home directory and other information. When a user logs in, the login program encrypts the password just read from the terminal and compares it to the one in the password file. If they agree the login is permitted, if not it is disallowed.

After a successful login the *login* program starts up the command line interpreter specified by the user's *password* file entry and then exits. Most of the command line interpreters is the shell. The **shell** initialises itself and types a prompt character, often a % or \$ sign on the screen and waits for the use to type a command line.

When the user types a command line, the **shell** extracts the first word from it, assumes it is the name of the program to be run, searches for this program and, if it finds it, runs the program. The **shell** then suspends itself until the program terminates, at which time, it tries to read the next command. The **shell** is an ordinary user program. All it needs is the ability to read from and write to the terminal, and the power to execute other programs.

Commands may take arguments which are passed to the called program as character strings. For example, the command line

\$ cp src dest

invokes the cp program with two arguments **src** and **dest**

This program interprets the first one to be the name of an existing file. It makes a copy of this file and calls the copy **dest**.

A program like the **shell** does not have to open the terminal in order to read from it or write to it. Instead, when it starts up it automatically has access to a file called **standard input**, a file called **standard output** and a file called **standard error**. Normally all three default to the terminal so that reads from the **standard input** come from the keyboard and writes to the **standard output** or **standard error** go to the screen.

2.4 : Summary

You have now been introduced to the basic features and issues in the **UNIX** operating system. You may now feel comfortable to try your hands on the UNIX system as a user. Please contact your system administrator and get yourself registered as an authorized user, with a specific login name and password. You will then be better able to appreciate the discussion in Chapter 3.

REVIEW QUESTIONS

- 2.1 Give an account of how UNIX versions came chronologically into existence?
- 2.2 Explain the main features of UNIX.
- 2.3 Explain the important constituents of UNIX structure.
- 2.4 What is a file in UNIX? And what is a file link?
- 2.5 What are the security rights associated with files and directories?

3 ! PRACTICE SESSION

3.1 ! Practice Session – I: Basic Unix Commands

3.2.1 Learning Objectives and Scope

The primary objective of this assignment is to introduce you to basic UNIX commands. The commands and utilities you will use are most common on any UNIX systems. At the end of these exercises you will be able to

- manipulate files
 - change permissions
 - count the number of lines, words, characters
 - create, delete, append, concatenate, display, move
 - extract lines
 - paginate, format the output
 - search for text strings within files
 - sort
- manipulate directories
 - create, delete
 - display contents
- change your password
- use basic commands
 - display a list of users
 - display messages and news
 - display the shell environment variables
 - obtain help on command usage
 - run programs in the background
 - time commands

The approach is interactive in nature so that the student can actively participate in the learning process while running through the contents. The purpose here is to let you perform standard operations on directories and files and to run common UNIX utilities. It will guide you to type a certain command or perform a task during the sessions. We advise you to go through this section while sitting at a terminal, with access to UNIX system, and with a username and password allocated to you by the system administrator in your institution.

3.2.2 Logging in to the UNIX Server

Follow the appropriate instructions for your system to access the UNIX server.

UNIX should normally display the following message to start this interactive session

Login:

Please take care of the following while seeking the access to the system. UNIX systems are case sensitive; upper and lower case characters are treated as different. All UNIX type systems generally use lowercase characters. All commands should be typed in lowercase characters. All users on a UNIX system are assigned their own usernames. When they logon to the system, they are placed in a HOME directory, which is a portion of the disk space reserved just for them.

1.0 Type your allocated username.
UNIX asks for your password.

1.1 Type your allocated password.

Note that the password is not displayed on the screen as you enter it. An example is shown below with username "harshini"

Login: **harshini**
Password:

The system checks the username and password you typed. If the username and password typed by you are correct, it will log you in. It does this by running a shell program for you. The shell program is your interface to the system. Once you have successfully logged in, you can proceed to the next section. If you cannot login, please ask your system administrator for help and guidance to access what is denied for you.

pwd (print working directory)

When a user logs in to a UNIX system, they are located in their own directory space. Users are generally located off the /usr directory. The **pwd** (print working directory) command displays the pathname of the current directory you are in. This is helpful when you want to know exactly where you are.

1.2 Type the pwd command, then *enter the command response on the line below*.
\$ **pwd**

\$
ls(list files)

This command is similar to DIR command used in **DOS**. It displays a list of all the files in the directory. Please note that if you are a new user, you may not have many files in your home directory.

1.3 Type the command
\$ **ls**

How many files were listed? _____

You will note that the shell prompt reappears, and there is no file listing of the directory contents. This does not mean that there are no files stored there. Remember that UNIX systems support hidden files.

UNIX systems extend the power of commands by using special flags or switches. These switches are one of the most powerful features of UNIX systems. Switches are preceded with a - symbol.

1.4 Type the command (to illustrate the use of switches)
\$ **ls -la**

How many files were listed? _____

The switch **l** stands for long listing, and the switch **a** is for all files, including directories and hidden files. For example, UNIX responds with the following listing of the directory. Response for your command will be different (yours will be different, this is only an example)

```
$ ls -la
total 9
-rw—— 1 rait comp 537 Jul 20 12:09 .profile
```

We can easily understand how the following are organized in the response:

permission settings	-rw (read and write)
number of links to file	1
owner of file	rait
group	cs
size of file in bytes	537
date/time of last modification	Jul 20 12:09
filename	.profile

The first part reveals the permission settings of the files. These have the following codes:

Directory	Read	Write	Execute	No access
D	r	w	X	—

There are normally three sets of permission settings (one set for the owner of the file, one set for the group to which the user belongs, and another set for other users on the system) specified in the following order, as illustrated in the example:

specifies whether a directory entry
permissions for the owner of the file
permissions for group members
permissions for others

1.5 What are the permission settings for the file *.harshini* in the current directory?

owner
group
others

cat (concatenate)

This utility is used to combine files or typing files to the screen. The cat command sends its output to the console screen (in UNIX we call this standard out or *stdout* for short).

1.6 Type the following command to view the file profile on the screen.

\$ cat .profile

The format of this command specifies that the cat utility is to use the file profile as its input, and send the output to the console screen. How to copy the data from the key board to a file has been illustrated below:

1.7 Type the command

\$ cat - > temp

This specifies that the input is from stdin (- means the keyboard), and writes the output to the file called temp (> means dump into).

1.8 Type the following lines of text

(ignore any typing mistakes, and do **not use** the cursor arrow keys).

Children like harshini always like to laugh as they love fun always

1.9 Press **CTRL d** to terminate text entry, and the shell prompt returns.

What is the command used to view the file temp on the screen.

The cat utility can also be used to join many files together into a single file. The following command (do not type this)

\$ cat temp .profile > tempS

copies the two files (temp and .profile) to a file called *tempS*.

1.9 *What is the command you would type to copy the file temp to a new file called tempq using the cat utility.*

The > is used for appending to an existing file, as in this example (do not type this).

\$ cat temp >> temps

cp (copy)

This command stands for copy, and is used for copying one file to another.

1.10 Type the command
\$ cp .profile tempq

This copies the file *.profile* to another called *tempq*. If *tempq* had already existed, its previous contents would've been erased.

Files can also be copied to another directory. The command (do not type this)

\$ cp * /usr/tmp

would copy all the files in the current directory to the directory */usr/tmp*.

1.11 *What is the command you would type to copy the file temp to a new file called temp1 using the cp utility.*

rm (remove)

The rm utility is used for erasing files and directories.

1.12 Type the command
\$ rm temp

This removes the file. Once a file is removed, it cannot be restored. To cover situations where mistakes might occur, a switch **-i** is normally appended to this command. It will request a Yes or No response before deleting the file.

1.13 Type the command
\$ rm -i temp

NOTE that switches are written before the filenames.

1.14 Answer **Y** to the prompt so that *temp* is removed.

head

This is used to view the first few lines of a file. It accepts a switch specifying the number of lines to view. The command (do not type this)

\$ head -2 harsh

would list the first 2 lines of the file *harsh* on the console.

1.15 What is the command to view the first 3 lines of the file *.profile*.

tail

This is used to view the last few lines of a file. It accepts a switch specifying the number of lines to view. The command (do not type this)

\$ tail -2 usha

would list the last 2 lines of the file *usha* on the console.

1.16 What is the command to view the first 5 lines of the file *.profile*.

lp (line printer)

This command is used to obtain a hardcopy printout of a file as illustrated below:.

\$ lp rani

wc (word count)

This utility displays a count of the number of characters, words and lines in a file. For example, to count in a file named *harshini*, we have the following:

1.17 Type the command
\$ wc harshini

This prints out a list of the number of lines, words, and characters respectively.

The switches for this utility are

-l	-c	-w
print line count	print character count	print word count

1.18 What is the command to count the number of words in the file *.profile*.

1.19 What is the command to count the number of lines in the file *.profile*.

pr (print)

The print utility displays the file to stdout. The switches for this command are given in table below:

-ln	set page length in lines to n (example, -120 sets page length to 20 lines)
-wn	set line length in characters to n
-n	number output lines
-t	do not print page header or trailer

-digit	number of columns to use (example, -2 sets columns to two)
+digit	start at page number (example, +1 sets start at page 1)

1.20 Type the command
\$ pr -2 -n -120 harshini

This prints the file *harshini* to stdout, using two columns per line, numbering each line, using a page length of 20 lines per page.

who
This command displays a list of the people registered on the system. Using the -u switch displays a list of those people currently logged into the UNIX server.

1.21 Type the command,
\$ who -u

talk
This command sends interactive messages between users.

1.22 Identify a user from the list printed by the previous who command.

1.23 Enter the command (but replace username with the name of an actual user)
\$ talk username
1.24 Exchange a few messages.

When a user receives a message, s/he should respond. The interactive exchange is cancelled by using the **Ctrl D** keys (some systems may use CTRL C). When this occurs, the message end of message appears on the other terminal, and control is returned to the shell.

1.25 Terminate the message exchange by pressing Ctrl D.

mesg (message)
This command permits or denies messages invoked using write. The switches it uses are -y and -n

1.26 Type the following command to disable messages.
\$ mesg n
1.27 Type the following command (but replace usname with your UNIX login name)
\$ talk username
What was the message printed on the console screen?

1.28 Re-enable messages by typing the command
\$ mesg y

news
The news command keeps the user informed of current events. Do not use this command at this time.

grep
The grep command searches a file for a pattern. The following command (do not type this)

\$ grep 'bagunnava' temp

searches the file *temp* in the current directory for the text string *bagunnava*. Using the *** wildcard specifies all files, and using the switch *-n* will display on which line the text string is found.

The following command (do not type this)

```
$ grep 'emantavu' *
```

searches all files in the current directory for the text string 'emantavu'.

1.29 What is the command to search the file *.profile* for the text string 'TERM' (case sensitive).

time

This command times how long it takes to execute a given command.

1.30 Type the command,
\$ **time ls -la**

The command responds with the elapsed time during the execution of the command-the time spent in the system and the time spent in execution of the command.

1.31 How long does it take to execute the *who* command?

cal (calendar)

This command prints a calendar to the screen. Please do not use this command at this time.

man (manual)

This command locates and displays the reference page for a specified item.

1.32 Type the command
\$ **man df**

The *:* on the screen represents a prompt. Pressing enter will continue the display. To quit from the manual, type *q*.

Present a brief summary of the command function performed by df in the space below.

Piping commands

UNIX systems offer pipelining. This feature is extremely powerful. It means that the output of one utility can be fed directly into the input of another. The *|* symbol indicates pipelining in UNIX systems. To illustrate how this works, consider the following command (do not type this command).

```
$ pr -2 -n temp | wc -l
```

This creates a pipe between the output of the *pr* utility and the input of the utility *wc*. The command runs *pr*, which takes the file *temp*, adds line numbers to it and formats it using two columns per line, feeding the output into *wc* which prints the line count on the stdout.

A pipe inter-connects two programs together

HINT: A pipe is used when a command (program) is used on both sides. In the above example, *pr* and *wc* are both commands, so they are connected using a pipe. If the right side was a filename or standard-device name,

then you use a redirection symbol to connect the program to the device-name or file. This command is wrong. A command connects to another command using a pipe rather than a redirect symbol!

\$ pr temp > wc

Exercise 1 Attempt answering the following

- 1.1 What is the command for creating a file called *harshu4*, which is a copy of *harshu5* but with line numbers for each line of the file?
- 1.2 What is the command for getting a line numbered printout (hard copy) of file *harshini* (which does not have line numbers) ?
- 1.3 What is the command to get a printout (hard copy) of the directory listing ?

passwd (password)

Every user in UNIX is always assigned a password. For changing the current password, **passwd** utility can be used. When the user types the following command (do not type this command),

\$ passwd

the utility asks for the new password twice, evidently for confirmation that you do not misspell it. The user types in the new password, but notice that it will not displayed on the screen when typed (for security reasons). The new password would, however, be immediately operative. If the user were to log in again, the system would ask for this new password to be inserted before allowing access.

mv (move)

The **mv** command is used for moving or renaming files.

1.33 Type the command
\$ mv harsh sarat

This renames the file *harsh* to *sarat*.
As another example, *the following command*

\$ mv harsh /xxx/yyy

moves the file *harsh* from home directory into the directory */xxx/yyy*

Directory Management

UNIX systems support hierarchical directory structures, which are managed by the following commands:

pwd	Print current working directory
cd	Change directory
mkdir	Make a subdirectory
rmdir	Remove a subdirectory

pwd (print working directory)

This command prints the current working directory on the console screen. In UNIX, the hard disk area is divided into directories, much like any book is segmented into chapters and paragraphs. The directories form a hierarchical level, which simplifies the organization of the files on the system. The top-most directory in UNIX is called **root**, and consists of a number of subdirectories grouped according to function.

1.34 In the space provided below, enter your current directory (use the *pwd* command).

When you use the *ls* command to list the file contents of a directory, those entries which are subdirectories are preceeded with a 'd' character. For example, look at the following screen display:

```
ls -la
total 27 files
drwxr-sr-x      3      n_savi   512    Nov   14   11:05  .
drwxr-sr-x     46      root    1124    Apr   23   16:47  ..
-rw-r--r--      1      n_savi  2201    Mar    3   19:02  .profile
drwxr-s--       2      n_savi   512    Nov   24   12:05  datafiles
-rw-r-----     1      n_savi     0    Dec   24   12:05  dirlist
-rw-r-----     1      n_savi    44    Jan    5    08:59  tmp
```

Please take note that **datafiles** is a subdirectory. There are two other subdirectory entries in the listing.

- . the current directory
- .. the parent directory

cd (change directory)

This command is used to change the current directory.

1.35 For each of the following commands, enter the current working directory after the command is executed.

cd .	
cd ..	
cd /	
cd \$HOME	

mkdir (make directory)

This command makes a subdirectory under the current directory.

1.36 State the command for creating a sub-directory called *sarat* ?.

rmdir (remove directory)

This command removes (deletes) a subdirectory under the current directory.

1.37 State the command for deleting a sub-directory called *sarat*

The Shell

This is probably the most powerful feature of UNIX. It is the user interface to the system, and has a number of built-in commands. The shell may also execute a command file (called a shell script).

set

Typing the set command will print out a list of all the current shell environments. This is a list of all the variables and settings that the shell currently has.

The majority of these are loaded when you first log in. The first thing the shell does is execute the file *profile* in the user home directory!

1.38 Type the command
\$ set | more
1.39 Enter in the space provided below, the environment variables printed by the set command.

HOME	
LOGNAME	
PATH	
PS1	
TERM	

Shell Variables

The shell supports user and system variables. An example of system variables are

PS1	shell prompt
PS2	secondary line shell prompt
HOME	home directory
LOGNAME	login name

Variables are assigned values using the = sign.

1.40 Type the following
\$ PS1="where are you:"

This changes the system prompt PS1 from a \$ to the text 'where are you:'

1.41 Type the command
where are you: **dir="ls -la"**

This variable *dir* can now be used as a command. Before executing it, the shell will expand it out. Shell variables, when used as commands, are preceded with a \$ symbol.

1.42 Type the command
\$ dir

The shell expands this out to the command 'ls -la' then executes it, resulting in a long listing of the current directory.

1.43 RESET the shell prompt PS1 back to the \$ character.

echo

The echo command echoes what follows.

1.44 Type the command
\$ echo "hello guys"

The shell responds with

\$ echo "hello guys"
hello guys
\$

Echo can also be used to print out shell variables.

1.45 Type the following command sequence.

```
$ sample="Hello guys"
```

```
$ echo $sample
```

1.46 What was printed as a result of the command?

1.47 Type set to view the added commands to the shell environment.

Running a secondary shell A secondary copy of the shell is invoked by typing (do not type this command yet)

```
$ sh
```

```
$
```

Commands are given to a secondary shell to execute, in either the foreground or background mode. The secondary shell will terminate when the command is finished.

1.48 What does the following command do, assuming the file `cmds` contains the text string `ls -la`?

Note:

To check this, run the **vi** editor enter the string **ls -la**. Save the contents to a file named **cmds**. Execute the command **chmod +x cmds** to flag the file as executable.

1.49 Then type the commands below

```
$ sh < cmds > dir.dat
```

```
_____
_____
_____
```

A secondary shell is terminated by typing **exit**.

Processing in the background

UNIX also runs processes in the background. When a command is appended with the **&** symbol, it is run in the background and the shell prompt immediately returns. This is handy for such work as printing documents, as it allows you to continue doing something else instead of waiting for the document to be finished printing.

Each process in UNIX is identified by a Process Identification Number (PID). When a background process is started off, UNIX will print its PID number in square brackets.

1.50 Type the command

```
$ ls -l > dirlist&
```

NOTE that the shell prompt immediately returns, and a number in brackets is printed. The **&** character signals to the shell that the command is to run in the background.

```
8828
```

```
$
```

This is the PID number of the background task. You can have more than one background task. Sometimes, you may want to shut down a background task. The **kill** command is used to shutdown a task, and it accepts the PID number of the task to shutdown.

sleep

This command suspends execution for a specified time interval, expressed in seconds.

1.51 Type the command,
\$ sleep 500&

This will run the sleep program in the background for 500 seconds.

pstree (process status, tree display)

This command reports on the active processes being run.

1.52 Type the command
\$ pstree -p

You may even see the sleep process listed if it is still running in the background.

Note: If pstree is not available on your system, try ps as an alternative.

User Profile Scripts

When a user logs into the UNIX system, a special user script file (.profile) is run which configures the environment for them. This script file is initially created by the system manager, but can be altered by each user to customize the environment to suit his or her own requirements.

The file .profile resides in the users login directory, and is not normally visible with the ls command unless the -a option is specified (its a hidden file).

It contains a series of commands that create shell variables and specify paths for commands. A portion of the file .profile for user ec401 looks like,

PATH=:/bin:/usr/bin:/\$HOME/bin:
export PATH MAIL

1.53 Type this command to view your .profile script
\$ cat .profile

vi (vee-eye editor)

Provided with UNIX is a simple text editor. The editor has two modes: a command mode and a text entry mode. The default mode is the command mode.

Commands used by vi (pronounced vee-eye),

i	enter text mode insert (use ESC to return to command mode)
x	delete character
dw	delete word
dd	delete line
:x	Save and exit
:q!	Quit and abandon any changes

The vi editor is invoked by (do not type this) **\$ vi filename**

1.54 Use vi to create the following text file, using the filename telephone.list
Harsh 5307233
Savi 5441212
Uma 5335454
Ram 545 8087
Deepika 5244321

```
Saras 655 7231
Usha 4343431
```

1.55 Save the file and return to the shell.

1.56 WRITE THE COMMAND YOU WOULD USE TO EXTRACT Harsh's PHONE NUMBER FROM THE FILE telephone.list AND DISPLAY IT ON THE SCREEN (using grep).

1.57 EDIT the file telephone.list, and add the following lines.

```
Harsh 434321
Deepika 2003000
Uma 4356711
```

There is now more than one occurrence of similar data (usernames) in the file. Using grep to search for user 'deepika' will find two occurrences. In a large file with hundreds of identical names (like a phone book), using grep in this manner results in wasted time.

If we are aware of some of the digits (like area code) for a certain user, then this can also be extracted, by re-piping the output of the first grep into a second grep command.

Consider this example,

```
$ grep 'harsh' phone.list | grep '434'
```

The first part extracts all line containing the text string harsh from the file phone.list, then pipes those lines to the next command, which searches the lines fed to it for the string '434'.

1.58 Write the command in the space below, to extract 'Deepika's' number that begins with '20'.

sort

The sort command sorts lines of input from files and writes the result on the standard output. The format of the sort command is,

sort [options] +pos1 -pos2 files...

where the options are

-f	ignore case
-ofileout	specifies output to be written to fileout
+pos1	start column for sort
-pos2	end column for sort
-n	numeric comparison

Numbering of columns start at 0. Columns are separated by tabs or space characters. The following data illustrates the numbering of the columns.

Harshini 5344477 Calgary Canada

Column 0	Column 1	Column 2	Column 3
Harshini	5344477	Calgary	Canada

1.59 Type the command
\$ sort -ophone.sort telephone.list

This sorts the file phone.list into alphabetical order, placing the result in the file phone.sort.

1.60 Enter the command below that will sort the file telephone.list by phone number.

1.61 Type the command, then view the file on the screen to verify if it worked.

uniq (unique)

This command reads the input file and compares adjacent lines. The second and succeeding copies of repeated lines are removed, and what is left is written to the output file. The format of this command is **uniq [options] +n -n file...** where the options are,

-c	Precede each line with a count of the number of times it occurred
-d	Write one copy of the repeated lines
-u	Write non-repeated lines only
+n	Ignore the first n characters
-n	Ignore the first n fields

1.62 Type the command,
\$ uniq -u telephone.list > newtelephone.list
This strips out any duplications in telephone.list and writes the resultant output to a new file.

diff (difference)

This command compares two text files, and displays differences between them. The format for this command is **diff [options] file1 file2**, where the options are,

-b	Ignore trailing blanks
-e	Produce a script of commands for ed which will change file2 to file1
-f	Similar to -e, but not for use with ed
-h	Fast less-rigorous comparison. Do not use with -e and -f

1.63 Type the command,
\$ diff telephone.list newtelephone.list
1.64 State the screen output generated by running the preceding diff command.

tee

This command is used to split the output into two streams, thus allowing the generation of an intermediate file to assess correct operation.

1.65 Type the command,
\$ grep 'Harsh' telephone.list | tee telephone.tmp | grep '530'

This command splits the output of the first grep command, and sends it to the file telephone.tmp as well as the second command.

1.66 Enter in the space provided below, the contents of the file telephone.tmp

chmod (change file modification rights)

This command alters the permission settings for a file or directory. Its format is

chmod [who] [permission] file...

The appendix contains a detailed description of this command.

Giving others access via chmod

Using this command, a user can allow other users access to their files, or create executable files of shell commands.

1.67 Type the command
\$ ls -la

The listing shows that the current rwx access for the group 'others' is currently set to —. This permission setting prevents other users from accessing information stored in your home directory.

1.68 Type the command sequence,
\$ cd \$HOME
\$ mkdir datafiles
\$ chmod o+r datafiles

The command sequence first changes the working directory to your home directory, creates a subdirectory called datafiles, then changes the permission settings allowing all other users read access.

Shell Script files

Commands can be stored in a file and executed by the shell.

1.69 Using vi, create a file called 'cmd.file' which contains the command 'ls -la'.

1.70 Type the command,
\$ cmd.file

1.71 What is the message printed by the shell?

UNIX creates files as text. In order to execute a file, you must have execute permission first. By using the ls command, you will see that the file is not flagged as executable.

1.72 USE the chmod command to add executable rights to the file cmd.file
 Enter the command you used in the space provided below.

1.73 Type the command
\$ **cmd.file**
1.74 What is now printed by the shell?

LOGGING OUT

There are two ways to finish a UNIX session: either typing **exit** or some systems use CTRL-D. A TCP/IP Telnet session is usually terminated by typing exit or closing the telnet application.

1.75 Type the following command to exit this session.
\$ **exit**

Summary of Command Syntax

cal	displays calendar month = 1 to 12 year = 1 to 9999																				
\$cat file	displays file																				
cd	return to login directory																				
cd dirname	change working directory to dirname																				
chmod [who] op-code permission... file...	who: <table><tr><td>o</td><td>others</td></tr><tr><td>g</td><td>group</td></tr><tr><td>u</td><td>Login owner</td></tr><tr><td>a</td><td>all of above</td></tr></table> op-code: <table><tr><td>+</td><td>add permission</td></tr><tr><td>-</td><td>remove permission</td></tr><tr><td>=</td><td>assign absolute permission</td></tr></table> permission: <table><tr><td>r</td><td>read</td></tr><tr><td>w</td><td>write</td></tr><tr><td>x</td><td>execute</td></tr></table>	o	others	g	group	u	Login owner	a	all of above	+	add permission	-	remove permission	=	assign absolute permission	r	read	w	write	x	execute
o	others																				
g	group																				
u	Login owner																				
a	all of above																				
+	add permission																				
-	remove permission																				
=	assign absolute permission																				
r	read																				
w	write																				
x	execute																				
cp file1 file2	copy file1 and name new copy as file2																				
diff [options] file1 file2	find difference between file1 and file2 options: <table><tr><td>-b</td><td>ignore spaces and tabs in comparison</td></tr><tr><td>-e</td><td>ed script option</td></tr><tr><td>-f</td><td>opposite order script</td></tr><tr><td>-h</td><td>simple, fast, less-rigorous comparison</td></tr></table>	-b	ignore spaces and tabs in comparison	-e	ed script option	-f	opposite order script	-h	simple, fast, less-rigorous comparison												
-b	ignore spaces and tabs in comparison																				
-e	ed script option																				
-f	opposite order script																				
-h	simple, fast, less-rigorous comparison																				
echo [arg]...	echo arg to standard output																				
find pathname-list... condition...																					

options:

-atime n	last accessed day n
-exec cmd	execute cmd if file meets conditions
-group gname	file owner gname
-links n	number of links to file
-mtime n	last modified day n
-name file	specify filename
-newer file	file modified more recently than file
-ok cmd	execute cmd only on 'y' response
-print	display the file pathname
-size n	file size in blocks
-type c	specify file-type
	b block
	c character
	d directory
	f plain file
-user uname	file owner uname

grep [options] pattern file...

find lines in file matching pattern options:

-c	count of matching lines
-e	use if pattern begins with '-'
-h	header display suppressed
-l	list of filenames containing pattern
-n	line number of lines which match
-v	variant: all lines except matches
-y	lowercase pattern will match uppercase

lp [options] file...

place file in line printer queue options:

-c	make copy of file for queue
-ddestprn	specify destprn as destination printer
-fformname	use the form specified by formname
-m	report by mail when printed
-nx	print x copies
-r	remove file after placing in queue

	-ttitle specify title as banner
	-w write message on terminal after printing
ls [options] name...	list contents of directory options: -a all entries -d directory status information -l long format -r reverse order listing -s size in blocks -t sort by time modification -u sort by last access time -x multi-column across page
man [options] section title	locates and displays pages of unix manual options: -a all matching title -f first matching title -w prints pathname to manual section
mesg [options]	report or set write permission on terminal options: -n deny non-user write permission -y enable non-user write permission
mkdir dirname...	create directory
mv file1 file2	change name of file1 to file2
mv dir1 dir2	rename dir1 directory to dir2 directory
mv file... dname	move one or more files to directory dname
passwd	change your password
pr [options] file...	formatted printing of file... options: -a multi-column across page -d double space output -f use form-feed character as page break -h "title" customized page head title -ln page length n lines -m print all files... in multiple columns

ps [options] process number

-n produce n-column output
+n start printing at page n
-sc separate columns with
character c
-t no header printed
-wn page width n characters
print information about active processes

options:

-a all processes except group
leaders and non-terminal
processes
-e all processes
-l long listing

pwd

display current working directory

rm [options] file...

remove one or more files

options:

-f force removal of write-protected
files
-i interactive delete
-r recursively delete directory

sort [options] [+pos1] [-pos2] [-ofileout] filein...

options:

-b ignore leading blanks and tabs
-c check if sorted
-d dictionary order
-f ignore case
-i ignore non-printable characters
in non-numeric

comparisons

-M compare as months
-m merge the files, input files are
already sorted
-n sort on first numeric field
-r reverse order of sort
-tx field separator is x
-u eliminate duplicate lines

+pos1, -pos2

restrict sort key to beginning at pos1 and ending
at pos2

-ofileout	write output to outfile
tail [+~n] file	copy file to stdout +n from beginning to ~n from end of file
tee [options] file...	divert a copy into file...
	options:
	-a append to file
	-i ignore interrupt signals
	-u unbuffered output
uniq [options] input [output]	remove repeated adjacent lines in input file when copying to output file options:
	-u display lines not repeated in input
	-d display one copy of repeated lines in input
	-c precede each line with number of times
repeated	
	-n ignore the first n fields
	+n ignore the first c characters
wc [options] file...	count lines, words and characters in file... options:
	-c count characters
	-l count lines
	-w count words
who	list user login name, terminal and login time
write user [ttyname]	write to user on terminal, ttyname the ttyname is optional.

3.2 : Practice Session – II : Basic VI Commands

3.2.1 Learning Objectives and Scope

The primary objective of this assignment is to introduce you to UNIX editor vi (pronounced as vee-eye). Scope of learning is expected to be that you will be able to do the following tasks:

1. insert, delete, copy, append and move text
2. position the cursor
3. save, undelete and quit
4. search, search and replace text

vi

The **vi** editor (pronounced vee-eye) is UNIX's standard editor. The purpose of the following exercises is to enable you to understand some of the important features of this editor. **vi** provides a window of 20 lines

into the file being edited. You can move to any line and make changes (insert, delete, over-write). Unix was designed to support simple ASCII terminals command line. **vi** has two basic modes of operation, **command** mode and **edit** mode.

The default mode is the command mode. In this mode, editing of user text cannot be performed. **vi** awaits the appropriate command from the user before performing the operation. In general, **vi** will return to the command mode after executing the command.

The advantage of this approach is that commands can be executed from files, and documents can be formatted or re-arranged by simply running a menu script of commands in **vi**.

Pressing the **ESC** key will cause the terminal to beep. When this happens, you are in command mode.

Loading vi

Log on to the UNIX system using your user account.

2.0 To load **vi** and begin an editing session, type,
vi newfile

This command loads the editor and, because the file does not already exist, creates it ready for use. The editor is now in command mode and is awaiting a command.

Moving the cursor

The following keys control cursor movement.

K	j	h	l
up	down	left	right

When **vi** is loaded on a new file, the cursor is restricted to the upper left position on the screen, and cannot be moved using the cursor keys.

Important

If you make a mistake typing text into the file, you cannot use the delete or cursor keys to correct the mistakes. Complete the line, mistakes and all, then enter command mode by pressing the **ESC** key. Next, position the cursor over the mistakes and use the appropriate commands to delete, replace or insert the correct characters.

Text input mode (append, insert, open)

The commands

a
i
o
O

allow text to be entered, starting at the upper left corner of the screen, for a new file. **When used on an existing file, the commands act as follows:**

a	Appends to right of the current cursor position
i	Inserts to the left of the current cursor position
o	Inserts one line down from the current cursor position
O	Inserts one line up from the current cursor position

2.1 Append the following text to the file.
Where does Harshini live in calgary ?

```
Let me ponder
1111303 POLARIS Forest Avenue.
```

If you make any typing mistakes do not correct them yet, enter all the lines first. Remember, you cannot use the cursor keys, backspace or the delete key.

2.2 Now press ESC to return to the command mode. Note that the cursor remains in the text area.

Deleting and Changing text

The three most used commands to alter text are

X	erase the character at the cursor
r	Replace the character at the cursor
dd	delete the entire line where the cursor is

All three commands are executed in command mode, and return to the command mode after executing.

2.3 Position the cursor over the first 1 on the third line of text.

2.4 Press the x key to erase the first 1. The text should now look like

```
Where does Harshini live in calgary ?
Let me ponder
111303 POLARIS Forest Avenue.
```

To erase the remaining two 1s in the last line, we could continue using the x command three more times. However, we can precede this command with the number of characters to delete.

2.5 Typing the command 2x shall give you the following result on the screen

```
Where does Harshini live in calgary ?
Let me ponder
1303 POLARIS Forest Avenue.
```

2.6 Position the cursor over the phrase 'Let me ponder?'. Erase the line using the dd command.

2.7 Position the cursor over the 'c' of the word 'calgary', and using the r command, capitalize the word. You would see which of the following?

```
Where does Harshini live in Calgary ?
Let me ponder
1303 POLARIS Forest Avenue.
or
Where does Harshini live in Calgary?
Let me ponder
1111303 POLARIS Forest Avenue.
```

Undoing changes

There are times that you make changes and immediately realize a mistake has been made. The **vi** editor provides means to undo previous commands. The **u** command undoes the previous command. The **U** command undoes all the changes made on the current line.

If you make a mistake, and wish to restore the current line to its original state,

2.8 State the command to use and verify

Saving the editor buffer and remaining in vi

You are strongly encouraged to save editing changes regularly. The command to write the editor buffer is

:w

If you decide that you do not want to over-write the existing file, but rather save the changes to a new file, then follow the **:w** command with the name of the new file,

:w newfile2

Exiting vi

The commands used to exit vi are,

ZZ writes the buffer to disk into the original file, then returns to the shell

:wq command is same as the command **ZZ**

:q! quits the editor, abandons the buffer, and returns to the shell

2.9 Quit vi and return to the shell

Additional cursor positioning commands

We have already introduced the keys h, j, k, l for moving the cursor left, down, up and right respectively.

We shall introduce four additional keys that control the cursor movement.

b	Move the cursor to the beginning of the previous word
e	Move the cursor to the end of the next word
0	Move to the beginning of the line (zero)
\$	Move to the end of the line

EXERCISES

2.10 The command sequence you would use to change this text (initial cursor position is shown at the 'W' of 'What')

Where is Colt Engineering Corporation?

It is in the capital of New Zealand ?

Calgary.

to the following

Where is Colt Engineering Corporation?

It is in the capital of Canada ?

Calgary.

Command used	What it does ?

- 2.11 Develop the answers for the following questions
1. Identify the command to move up one line?

2. Identify the command to move down one line?

3. Identify the command to delete a line?

4. Identify the command to abandon the editor and return to the shell?

5. Identify the command to undo the last command?

6. Identify the command to position the cursor at the end of the current line?

7. Identify the command to write the buffer to the file and remain in vi?

8. Identify the command to erase the character at the cursor position?

9. Identify the command to position the cursor at the beginning of the previous word?

Screen Scrolling and Paging

The commands to scroll the screen up and down a page at a time (12 lines) are

ctrl-d	scroll down
ctrl-u	scroll up

For very long files, vi provides a go to line number command.

200G	go to line 3000
------	-----------------

To position the cursor at the **last line** in the file, type G
To position the cursor at the **beginning of the file**, type 1G
This command **displays the line number** of the cursor position

ctrl-g	Display current line number
--------	-----------------------------

Searching

Another method of positioning within a file is to search for a string of characters. When in command mode, any string preceded by a / indicates a *search forward command*. The cursor is positioned at the first occurrence of the text string. The command n will search *forwards* for the *next occurrence*. The ? command is used for searching *backwards*.

- 2.12 Identify the commands needed for the following:
1. Use the cursor go to command to position the cursor at the first line in the file.

2. Search forward for the word 'Calgary'

3. Identify the current line number.

Delete, Change, Rearranging text

The dd command has already been covered. This command *deletes the current line*.
The dw command *deleted the current word*. Note that the delete command begins with d, followed by the scope of the command (d for a line, w for a word).
In this section, the operators delete, change and yank will be discussed, and the scopes will be words, lines, sentences and paragraphs.
These commands will be used to delete, duplicate, change and re-arrange text. The following tables summarize the operators and scopes discussed in this section.

Operator	Action
D	deletes text into a temporary buffer. Recovered using the 'put' command.
Y	yanks a copy of text into a temporary buffer. The copy is then pasted using the 'put' command.
P	puts whatever last deleted or yanked, after or below the cursor.
C	same as delete followed by insert. Performs the delete operation then enters text insert mode. Must press ESC to go back to command mode.

Delete, Yank, Put and Change Operators

Scope	Action
E	from the cursor to the end of the current word.
w	from the cursor to the beginning of the next word, including the space.
b	from the letter before the cursor backwards to the beginning of the word.
\$	from the cursor to the end of the line.
0	from just before the cursor to the beginning of the line.
)	from the cursor to the beginning of the next sentence.
(	from just before the cursor backwards to the beginning of the sentence.
}	from the cursor to the end of the paragraph.
{	from just before the cursor backwards to the beginning of the paragraph.

Scope operators

The **dd**, **cc**, and **yy** commands affect the entire line.

Delete [D]

The current file being edited looks like

Who lives in Canada? and where?
Harshini at Calgary

2.13 Position the cursor at the end of the text. Enter the text insert mode and enter the following paragraph.

Harshini is a good girl and intelligent always like to sing and dance. She is very keen on studies. She likes French fries.

2.14 Position the cursor on the first letter of the second paragraph. The screen now looks like

Who lives in Canada ? and where ?
Harshini at Calgary
Harshini is a good girl and intelligent always like to sing and dance. She is very keen on studies. She likes French fries.

2.15 Identify the Command for Deleting the current word.

2.16 Identify the command for deleting the current sentence.

2.17 Position the cursor over the first character of the word 'French'.

2.18 Identify the command for deleting from the current cursor position to the end of the line.

Hint: The screen is expected to have the following at the end
Who lives in Canada ?
and where ?
Harshini at Calgary
She is very keen on studies. She likes

Yank and Put

Temporary buffers store the contents of all data deleted. The contents of this buffer can be accessed and 'put' or pasted anywhere in the text.

Un-named temporary buffers The temporary buffers are numbered 1 to 9, and hold each preceding deletion. Thus, buffer 1 holds the last deletion, buffer 2 the next previous, and so on. The temporary buffers can be accessed using the **put** command, by preceding the command with the buffer number preceded by a double quote ("np).

2.19 What put command is needed to restore the last deletion of 'French fries' to its original place in the text ?

The **yank** command copies the specified text into temporary buffers, leaving the original text present in the text file.

Named buffers There are 26 named buffers for use by the yank command. These buffers are named 'a' to 'z'. The command **"ay}** yanks from cursor to end of paragraph into buffer **a**

2.20 Position the cursor over the first letter of the word 'French'. Type the following command to yank the word into the temporary buffer. "ayw

Change

- 2.21** Position the cursor over the first character of the word 'French'. What is the change command to change the word to 'Indian'.
- 2.22** Press the ESC key to return to the command mode. Position the cursor over the first letter of the word 'Indian'. Using a combination of the delete and put commands, we have to replace this with the previously yanked copy of the word 'French'. Identify the necessary command.
- 2.23** Check the contents of the text file.

Moving Blocks

The **move** command moves a block of lines from one point in the source file to another. This options works on line numbers. To turn-on the display of line numbers within vi, **type** ctrl-n

The format of the **move** command is startline, endlinemafterdestline.

Following command moves lines 1 to 3 to the end of the file.

- 1,3m\$
- 2.24** What is the Command for moving lines 1 and 2 to the end of the file.
- 2.25** Check the contents of the text file.
- 2.29** QUIT the file, abandon and exit to the shell.
- 2.30** Logout of the UNIX system by pressing CTRL-D or typing exit.
- 2.31** Answer the following questions
1. Identify the command to scroll the screen down 12 lines.
 2. Identify the command to position the cursor on the first line.

3. Identify the command to display the current line number.
4. Identify the command to position the cursor on the last line in the file.

2.32 Match the commands shown on the left to the functions shown on the right. Assume all the commands are given in the command mode only, none are given in the text input mode.

35G		scrolls screen down 1/2 page
3yw		moves cursor down 1 line
r2		stores four lines in buffer c
/fun		prints line number of the current line
[control-g]		moves cursor to left one character
2dd		replaces character under cursor with number 2
J		yanks out 3 words
"c4dd		substitutes funny for fun
[control-d]		deletes 2 lines
H		puts cursor on line 35
		finds the word "fun"

- 2.33** Answer the following.
1. How do you indicate file is new while you invoke the editor with a file?
 2. What does the :q command do ?
 3. When you first enter the editor with an existing file and type a, the append command to add text, where is the text placed?
 4. What is the command to save the editor buffer contents?
 5. Where does the insert command I place new text?
 6. Identify the command that will save the first three lines of the buffer to a file called 'temp2'.
 7. How do you exit from the text input mode?
 8. Identify the command(s) to correct the following misspelled word, "soemetime."
 9. Identify the command(s) to delete the last five lines of the buffer.

Summary of Command Syntax

1. Text Mode Input - End with ESC

A	append text after the cursor
I	insert text before the cursor
O	Open a new line below cursor
O	Open a new line above cursor
R	replace characters on screen, starting at the cursor

2. Command Mode—after execution return to command mode

R	Replaces the character under the cursor
/happy	Searches sequence, look for next occurrence of 'happy'
?lark	Searches sequence, look for previous occurrence of 'lark'
N	used after / or ? to advance to next occurrence of the search pattern
U	undoes the last command
U	undoes all the changes on the current line
X	deletes character under the cursor
[del] or [ctrl-h]	deletes character to left of cursor
[ctrl-d]	scrolls screen down one half page at a time
[ctrl-u]	scrolls screen up one half page at a time
NG	Positions the cursor at line n in the file
[ctrl-g]	Identifies the line number where the cursor is located

3. Operators used in the Command Mode

d	deletes indicated text starting at the cursor.
dw	deletes a word
dd	deletes a line
3dd	3dd deletes three lines
	deleted text is stored in a temporary buffer, whose contents can be printed using the p command.
c	deletes indicated text starting at the cursor, then enters the text input mode.
ce	deletes from the cursor to the end of the word
y	copies the indicated text, starting at the cursor, and stores it in a buffer. There are nine unnamed buffers that store the last nine delete or yank operations and 26 named buffers (a-z) that can also be used for storage. A double quote is used to specify the name of the buffer. "cy\$ stores the text from the cursor to the end of the line in buffer c
P	puts the delete or yank buffer contents after the cursor or on the next line.
p	puts the last item yanked or deleted back into the file just after the cursor
"cp	puts the contents of buffer c after the cursor
P	identical to the p command, but places the text before the cursor

4. Scope for use with operators

e	the end of the current word
w	from the cursor to the beginning of the next word, including the space
b	from the character before the cursor, backwards, to the beginning of the word
\$	from the cursor to the end of the line
0	from just before the cursor to the beginning of the line
)	from the cursor to the beginning of the next sentence (. ! ? return or two spaces)
(	from just before the cursor back to the beginning of the sentence which contains the cursor

}	from the cursor to the end of the paragraph. A paragraph begins after a blank line
{	from just before the cursor back to the beginning of a paragraph

5.0 Exiting the editor

:w	Writes the buffer into the current file :3,10w popcorn writes lines 3-10 to the file called 'popcorn' :w writes the entire buffer to the current file
:q	quits buffer after using the :w command
:wq	Writes and quits
:q!	quits editor without writing buffer contents to the file
ZZ	write and quit. Same as :wq

3.3 : Practice Session—III: Basic Awk Programming

3.3.1 Learning Objectives and Scope

The primary objective of this assignment is to introduce you to basic **awk** utilities available in UNIX. You are expected to learn the following:

- Search patterns
- Match Patterns
- Perform actions on files

awk PROGRAM

An awk program consists of a number of patterns and associated actions. Actions are enclosed using curly braces, and separated using semi-colons.

```
pattern { action }
pattern { action }
```

awk scans input lines one after the other, searching each line to see if it matches a set of patterns or conditions specified in the awk program. An action is specified for each pattern. The action is performed when the pattern matches that of the input line.

When awk scans an input line, it breaks it down into a number of fields. Fields are separated by a space or tab character. Fields are numbered beginning at one, and the dollar symbol (\$) is used to represent a field. For instance, the following line in a file.

```
Harshini likes honey because it is so sweet.
```

This line has eight fields. They are as follows:

```
$1      Harshini
$2      likes
$3      honey
$4      because
$5      it
$6      is
$7      so
$8      sweet.
```

Field zero is specified by **\$0** and it refers to the entire line. **AWK** scans lines either from a file or files or through standard input. Consider the following simple awk program.

```
{ print $0 }
```

There is no pattern here to match, only an action is expressed. This means that for every line encountered, perform the action. The action *prints* field 0 (the entire line). Now using a text editor, create a file called *harsawk1* and place the above statement in it. Save the file and return to the Unix shell prompt. To run the above program

3.1 Type following command
`awk -f harsawk 1 /etc/group`

awk interprets the actions specified in the program file *harsawk1*, and applies this to each line read from the file */etc/group*. The effect is to print out each input line read from the file, in effect, displaying the file on the screen (same as the UNIX command *cat*). In order to search for an occurrence of a string in an input line, we have to specify it as a pattern and enclose it using a forward slash symbol. For example, we have to search each input line for the string *sweet* and the print the entire line.

3.2 Use the following command.

```
/sweet/ { print $0 }
```

3.3 Edit *harsawk1* and change the search string to your username. Run the program on the files */etc/group* and */etc/passwd*

```
awk -fharsawk1 /etc/group
awk -fharsawk1 /etc/passwd
```

3.4 Please note that in the previous example there was no pattern specified, what is then the difference in the output of this program.

3.5 Type the following command.

This runs the program *who* and sends its output of who is logged on the system to the awk program which scans each line for the search string. It will thus list out a line containing your login name, terminal number and login date/time.

```
who | awk -f harsawk1
```

3.6 Change the contents of *harsawk1* to read (replace the search string with your login name)
`/sweet/ { print $1, $2 }`
 What do you expect the output of the program to be?
 What fields will it print out?)

3.7 Now type the command

```
who | awk -f harsawk1
```

What do you see now ? How is the output different than the earlier output?

awk programs are particularly suitable for generating reports or forms. We shall use the following textual data as the input file. This file is called **awksample**. A heading has been provided here for clarity. Remember that there is no header in the data file.

Contents of the file **awksample** is shown below, which has five fields.

code	name	family	Skills	Experience
J100	designer	A1	C	4
J101	analyst	B2	C++	4
J102	architect	C4	JDC	5
J103	programmer	D3	Siebel	6
J110	manager	D2	Clarify	6
J111	designer	A1	Arbor	8
J111	BDM	B2	TeMIP	9
J113	designer	C4	NetCom	10
J200	manager	F5	Kenon	20
J200	Consultant	G2	java	30
J202	Consultant	G2	unix	10
J203	Researcher	G2	oracle	30

Pattern Selection

This involves specifying a pattern to match for each input line scanned. The following awk program (harshini2) compares field one (\$1) and if the field matches the string "J111", the specific action is performed (the entire line is printed).

```
$1 == "J111" {print $0 }
```

Note: The == symbol represents an equality test. Thus, in the above pattern, it compares the string of field one against the constant string "J111", and performs the action if it matches.

Create Program

```
$ cat -> harshini2
$1 == "J111" { print $0 }
< ctrl-d>
$
```

Run Program

```
$ awk -f harshini2 awksample
```

Sample Program Output

```
J111      GM      A1      Arbor      8
J111      BDM     B2      TeMIP     9
```

The program output includes all input lines where the code is "J111".

3.8 Write an awk program which prints out all input lines where name is designer.

Comments in awk Programs

Comments have to begin with the hash (#) symbol and continue till the end of the line. The awk program below adds a comment to a previous awk program shown earlier.

```
# The program harshini3 is same as harshini2 except that it has a comment
$1 == "J111" { print $0 }
```

Comments can also be placed anywhere on the line. The example below illustrates how the comment is placed after the action.

```
$1 == "J111" { print $0 } # print all records where the code is J111
```

Remember that the comment ends at the end of the line. Please notice that the following program is wrong, as the closing brace of the action is treated as part of the comment.

```
$1 == "J111" { print $0 #print out all records }
```

Relational Expressions

We have already used “ the equal to ” symbols in our program. Detailed below are the other relational operators that are useful for comparing expressions.

<	less than
< =	less than or equal to
==	equal to
!=	not equal
> =	greater than or equal to
>	greater than
~	matches
!~	does not match

BEGIN and END Statements

- BEGIN** The action associated with this keyword is executed before the first input line is read.
- END** The action associated with this keyword is executed after all input lines have been processed.

```
# harshini4, an awk program to display all input lines for jobs
# with experience less than 6
$5 < 6 { print $0 }
```

3.9 harshini4 Program Output

J100	designer	A1	C	4
J101	analyst	B2	C++	4
J102	architect	C4	JDC	5

```
# harshini5
# an awk program to print the job with skill of arbor
$4 == "Arbor" { print $2, $3, $5 }
```

3.10 harshini5 Program Output

designer	A1	8
----------	----	---

3.11 Find the output of the following :

```
$2 != "designer" { print $0 }
```

```
$5 > 20 { print $4 }
```

```
$2 !~ /Con/ { print $0 }
```

3.12 Write an awk program to display all the details in the family G2

```
# harshini6
```

```
# list all jobs in family A1
```

```
$3 ~ /A1/ { print "Skill = ", $4, " Experience = ", $5 }
```

3.13 harshini6 Program Output

```
Skill = C Experience = 4
```

```
Skill = Arbor Experience = 8
```

Following is the text file named *abcawk* for exercises **3.14 - 3.33**

Type	Memory (Kb)	Location	Serial #	HD Size (Mb)
XT	640	D402	MG0010	0
386	2048	D403	MG0011	100
486	4096	D404	MG0012	270
386	8192	A423	CC0177	400
486	8192	A424	CC0182	670
286	4096	A423	CC0183	100
286	4096	A425	CC0184	80
Mac	4096	B407	EE1027	80
Apple	4096	B406	EE1028	40
68020	2048	B406	EE1029	80
68030	2048	B410	EE1030	100
\$unix	16636	A405	CC0185	660
"trs80"	64	Z101	EL0020	0

3.14 Verify the following programs and the respective outputs using the file *abcawk*

```
#abcawk1
```

```
$1 == "286" { printf("Location : "); print $3 }
```

abcawk1 Program Output

```
Location : A423
```

```
Location : A425
```

```
#abcawk2
```

```
$1 == "286" { printf("Location is %s\n", $3 ); }
```

abcawk2 Program Output

Location is A423
Location is A425

```
#abcawk3
$1=="286" { printf("Location = %s, serial # = %s\n", $3, $4 ); }
```

abcawk3 Program Output

Location = A423, serial # = CC0183
Location = A425, serial # = CC0184

```
#abcawk4
$1=="486" { printf("Location = %s, disk = %dKb\n", $3, $5 ); }
```

abcawk4 Program Output

Location = D404, disk = 270Kb
Location = A424, disk = 670Kb

```
#abcawk5
# formatting the output using a field width
$1=="286" { printf("Location = %10s, disk = %5dKb\n", $3, $5); }
```

abcawk5 Program Output

Location = A423, disk = 100Kb
Location = A425, disk = 80Kb

- 3.15 Use the file *abcawk*. Write** an awk program which counts the number of computers which have 8192Kb or greater amounts of memory, then prints the number found at the end of the program.
- 3.16 Use the file *abcawk*. Write** an awk program which sums the disk space of all computers, then prints the total disk space at the end of the program.

Regular Expressions

awk provides pattern searching which is more comprehensive than the simple examples outlined previously. These patterns are called *regular expressions*, and are similar to those supported by other UNIX utilities like **grep**. The simplest regular expression is a string enclosed in slashes:

`/386/`

In the above example, any input line containing the string `386` will be printed. To restrict a search to a specific field, the match (or not match) symbol is used. In the following example, field one of the input line is searched for the string `386`.

`$1 ~ /386/`

In regular expressions, the following symbols are metacharacters with special meanings.

`\ ^ $ . [ ] * + ? ( ) |`

`^` matches the first character of a string
`$` matches the last character of a string

. matches a single character of a string
 [] defines a set of characters
 () used for grouping
 | specifies alternatives

#abcaawk6, display all x8x computer types
 \$1 ~ /[86]/ { print \$0 }

3.17 abcaawk6 Program Output

386	2048	D403	MG0011	100
486	4096	D404	MG0012	270
386	8192	A423	CC0177	400
486	8192	A424	CC0182	670
286	4096	A423	CC0183	100
286	4096	A425	CC0184	80
68020	2048	B406	EE1029	80
68030	2048	B410	EE1030	100
"trs80"	64	Z101	EL0020	0

Note: In this example, field one is searched for the character '8' and '6', in any order of occurrence and position.

If the first character after the opening bracket ([]) is a caret (^) symbol, this complements the set so that it matches any character NOT IN the set. The following example (myawk17) shows this, matching field one with any character except "8" or "6".

#abcaawk7
 # display all which do not contain 2, 3, 4, 6 or 8 in first field
 \$1 ~ /^[^23468]/ { print \$0 }

3.18 abcaawk7 Program Output

XT	640	D402	MG0010	0
Mac	4096	B407	EE1027	80
Apple	4096	B406	EE1028	40
68020	2048	B406	EE1029	80
68030	2048	B410	EE1030	100
\$unix	16636	A405	CC0185	660
"trs80"	64	Z101	EL0020	0

3.19 Why are the lines containing "68020", "68030" and "trs80" also displayed in the output of Exercise 3.18 ?

Verify the outputs 3.20 and 3.21 with respect to the file **abcaawk**
 #abcaawk8

```
# display all lines where field one contains A-Z, a-z
$1 ~ /[a-zA-Z]/ { print $0 }
```

3.20 abcawk8 Program Output

XT	640	D402	MG0010	0
Mac	4096	B407	EE1027	80
Apple	4096	B406	EE1028	40
\$unix	16636	A405	CC0185	660
"trs80"	64	Z101	EL0020	0

```
#abcawk9
# illustrate multiple searching using alternatives
/(Apple|Mac|68020|68030)/ { print $0 }
```

3.21 abcawk9 Program Output

Mac	4096	B407	EE1027	80
Apple	4096	B406	EE1028	40
68020	2048	B406	EE1029	80
68030	2048	B410	EE1030	100

3.22 Write an awk program which prints out all input lines for computers which belong to the school of management (using metacharacters).

3.23 Write an awk program which prints out all input lines for computer types which begin with a dollar (\$) symbol (using metacharacters).

3.24 Write an awk program which lists all computers of type "286", "386" and "486" which have hard disk.

3.25 Write an awk program to print out the percentage (to one decimal place) of computers which have 2048Kb or less of memory.

3.26 Write an awk program to calculate and print out (to three decimal places) the natural logarithm of a value entered from the keyboard.

3.27 Write an awk program to find and print out the longest computer name

3.28 Write an awk program to print out only those computers which are type "486" with 4096Kb or more memory. Use an **if** statement to perform this.

3.29 Write an awk program to print out the total number of computers held. Use a **while** statement to perform this.

3.30 Write an awk program to print out the total number of computers held. Use a **for** statement to perform this.

- 3.31 Write** an awk program to print out the total disk space for computer types “286”, “386” and “486”.
- 3.32 Write** an awk program to print out a sorted list of all “286”, “386” and “486” computers sorted according to disk size.
- 3.33 Write** an awk program to list all details of type “286” computers. Prefix the list with the current date (Hint: see the previous example, and the UNIX command *date*).

awk Statements Summary

1. Command Line

awk *program filenames*

awk -f *program-file filenames*

awk -Fs

(sets field separator to string *s*, **-Ft** sets separator to tab)

2. Patterns

BEGIN

END

/regular expression/

relational expression

pattern & & pattern

pattern || pattern

(pattern)

!pattern

pattern, pattern

3. Control Flow Statements

if (*expr*) *statement* [**else** *statement*]

if (*subscript in array*) *statement* [**else** *statement*]

while (*expr*) *statement*

for (*expr* ; *expr* ; *expr*) *statement*

for (*var in array*) *statement*

do *statement* **while** (*expr*)

break

continue

next

exit [*expr*]

return [*expr*]

4. Input Output

close(*filename*) close file

getline sets \$0 from next input line, set NF, NR, FNR

getline < *file* sets \$0 from next input line of file, set NF

getline *var* sets *var* from next input line, set NR, FNR

getline *var* < *file* sets *var* from next input line of file

print prints current input line

print *expr-list* prints expressions

print *expr-list* > *file* prints expressions to file

printf *fmt, expr-list* formats and print

printf *fmt, expr-list* > *file* formats and print to file

system(*cmd-line*) executes command *cmd-line*, returns status

5. Functions

func *name*(*parameter list*) { *statement* }

function *name* (*parameter list*) { *statement* }

function-name (*expr*, *expr*, ...)

6. String Functions

gsub(*r,s,t*) substitutes *s* for *r* in *t* globally, returns number of substitutions

index(*s,t*) returns position of string *t* in *s*, 0 if not present

length(*s*) returns length of *s*

match(*s,r*) returns position in *s* where *r* occurs, 0 if not present

split(*s,a,r*) splits *s* into array *a* on *r*, returns number of fields

sprintf(*fmt, expr-list*) returns *expr-list* formatted according to format string specified by *fmt*

sub(*r,s,t*) substitutes *s* for first *r* in *t*, returns number of substitutions

substr(*s,p,n*) returns substring of *s* length *n* starting at position *p*

7. Arithmetic Functions

atan2(*y,x*) arctangent of *y/x* in radians

cos(*x*) cosine of *x*, with *x* in radians

exp(*x*) exponential function of *x*

int(*x*) integer part of *x* truncated towards 0

log(*x*) natural logarithm of *x*

rand() random number between 0 and 1

sin(*x*) sine of *x*, with *x* in radians

sqrt(*x*) square root of *x*

srand(*x*) *x* is new seed for **rand**()

8. Operators (increasing precedence)

= += -= *= /= %= ^= assignment

?: conditional expression

|| logical or

& & logical and

~ !~ regular expression match, negated match

< <= > >= != == relationals

blank string concatenation

+ - add, subtract

* / % multiply, divide, modulus

+ - ! unary plus, unary minus, logical negation

^ exponential

++ — increment, decrement

\$ field

9. Regular Expressions (increasing precedence)

c matches no-metacharacter *c*

\ *c* matches literal character *c*

. matches any character except newline

^ matches beginning of line or string

\$ matches end of line or string

[*abc...*] character class matches any of *abc...*

[^*abc...*] negated class matches any but *abc...* and newline

r1 | *r2* matches either *r1* or *r2*

r1r2 concatenation: matches *r1*, then *r2*

<code>r+</code>	matches one or more <code>r</code> 's
<code>r*</code>	matches zero or more <code>r</code> 's
<code>r?</code>	matches zero or more <code>r</code> 's
<code>(r)</code>	grouping: matches <code>r</code>

10. Built-In Variables

ARGC	number of command-line arguments
ARGV	array of command-line arguments (0.. ARGC -1fR)
FILENAME	name of current input file
FNR	input line number number in current file
FS	input field separator (default blank)
NF	number of fields in input line
NR	number of input lines read so far
OFMT	output format for numbers (default=%. 6g)
OFS	output field separator (default=space)
ORS	output line separator (default=newline)
RS	input line separator (default=newline)
RSTART	index of first character matched by match()
RLENGTH	length of string matched by match()
SUBSEP	subscript separator (default="\034")

11. Implementation Limits

100 fields
 2500 characters per input line
 2500 characters per output line
 1024 characters per individual field
 1024 characters per printf string
 400 characters maximum quoted string
 400 characters in character class
 15 open files
 1 pipe

12. MSDOS to UNIX file conversion

dtox harshini > harshini.unx converts a MSDOS file to UNIX format.
xtod harshini.unx > harshini converts a UNIX file to MSDOS format.

3.4 | Practice Session—IV: Basic Mail Commands

3.4.1 Learning Objectives and Scope

The primary objective of this assignment is to introduce you to basic mail commands. You are expected to be able to do the following tasks:

- Send messages
- Transfer files
- Send personal reminders
- Communicate to remote systems

MailBoxes

UNIX holds mail messages in the following two mailboxes:

system mailbox (/usr/spool/mail/)
 user mailbox (....../mbox)

Mail arrives in the system mailbox, and is saved in your user mailbox after you have read it. The user mailbox is normally located in the respective \$HOME directory.

Mail Message Format

A mail message is exchanged between people. Mail messages consist of two parts

header	contains information about sender, receiver and subject
body	actual content of message

Mail Headers

Mail headers have the following construction:

To:	This specifies the name of the recipients of the message (mandatory)
Subject:	Title describing the message (optional)
Cc:	List of people to receive a carbon copy (optional)
Bcc:	List of people to receive blind carbon copy (they do not see user names in the received message. Optional)

Modes of Operation

The mail program operates in two main modes

compose mode	messages are created
command mode	manage your mail (list/edit/delete/print/save messages)

Typing the command **mail** runs the mail program and checks for mail. If there is mail, command mode is entered. If there is no mail, the program returns immediately. Compose mode is entered when you invoke the mail program and specify a user as an argument as follows

```
mail harshini
```

Here harshini is the username.

Entering mail Compose Mode

This mode is entered by invoking the mail program from the UNIX command line prompt and specifying a user to whom the mail should be sent. The example below invokes the mail program and names the recipient of the message as **sravya**. The user is placed into compose mode, where the message contents is filled in (DO NOT type this example!).

```
mail sravya
```

When the mail program starts, it reads a mail script file (if one is present) located in the user home directory. This specifies options for how the mail program works. One of the options is *asksub*, which tells the mail program to ask for a subject heading. If this option is set, then the mail program prompts you to enter a subject line,

```
Subject:
```

Enter the string "**How are you Sraveee**" as the subject content for this message. Subject fields are limited to a single line. The mail program is now in **compose mode**.

4.1 Enter each of the following lines. At the end of each line press.

```
Hi ! Harshu and Sradavees !
Do you know UNIX ?
Prabhu is not in town
I am entering the contents of this
message in compose mode. Once I press the return key
I cannot edit previous lines.
Have a nice day - Deepika.
```

To finish entering the message, press **CTRL-d**. The mail message is sent, and the mail program returns you to the shell prompt.

Please take note of the following:

- To delete a line in compose mode, press CTRL-U
- To view the message before sending it, press ~p
- To abort the message, press the DELETE key twice
(this has been mapped to CTRL-BS on the IBM-PC keyboard, and will save the message in the file dead-letter)
- To send the message, press CTRL-d

Sending a mail message contained in a file to a user

Suppose the mail message already exists in a file named *harshini* and it is to be sent.

You can send it by typing the command given below:

```
mail deepika < harshini
```

which sends the text file *harshini* to the user *deepika*.

4.2 Create a file name “deepika” which contains a message to send.

```
cat - > tmpfile
Hello and welcome. This message was in
a file named deepika.
```

4.3 Send the file to the user harshini

```
mail harshini < tmpfile
```

Subject Content Specification while sending a message

The command line option **-s** specifies the subject header for mail to use. The following command

```
mail -s "We are not going to Navratna today at 6.00 pm" harshini < calgary
```

sends the file *calgary* to the user *harshini* and inserts the text string “ We are not going to Navratna today at 6.00 pm ” into the subject field of the message header.

4.4 Send the file “rekha” to yourself, using a subject header string “I have nothing to say as on today”

Entering mail command mode

This mode is entered when you invoke the mail program without arguments and there is mail waiting for you.

Type

```
mail
```

The mail program displays a title message and lists all available mail headers:

SCO System V Mail (version 3.2) Type ? for help.

“/usr/spool/mail/harshini”: 3 messages 3 new

```
N 3 harita      Wed Jul 31 15:02 10/299
N 2 deepika    Wed Jul 31 15:01 9/278 Hi! Ranjani how are you
>N1 sindhuja   Wed Jul 31 15:00 12/415 How are you
&
```

This initial screen displays the subject fields of messages that have arrived. The format of the display is

Type	Message_number	From_User	Date/Time	Subject
N	denotes a new message			
>	denotes the current message			
&	mail prompt symbol			

It is important to observe that the message number 3 does not have a subject heading because the mail message was sent from a file and the -s option was not specified on the command line while the mail program was invoked.

Getting help

For the display of help while running the mail program, you have to type a question (?) mark

!	shell escape
cd [directory]	chdir to directory or home if none given
delete [msglist]	deletes messages
edit [msglist]	edits messages
from [msglist]	gives header lines of messages
header	prints page of active message headers
list	lists all commands (no explanations)
mail user	sends mail to specific user
next	go to and types next message
preserve [msglist]	preserves messages in mailbox
quit	quits, preserving unread messages
reply [message]	replies to message, including all recipients
Reply [msglist]	replies to the authors of the messages
save [msglist] file	appends messages to file
top [msglist]	prints top 5 lines of messages
type [msglist]	prints messages
undelete [msglist]	restores deleted messages
xit	quits, preserving all messages
z [-]	displays next [last] page of 10 headers
[msglist] is optional and specifies messages by number, author, subject or type.	
The default is the current message.	

Current Message Reading

To read the current message denoted by the > symbol, press the return key. The mailer then displays the message. A sample message is shown below.

Message 1:

From sindhuja Mon May 31 15:00:20 1993
From: sindhuja@hotmail.com (P Sindhuja)
X-Mailer: SCO System V Mail (version 3.2)

To: harshini
Subject: how are you
Date: Wed, 31 Jul 93 15:00:16 NZT
Message-ID: <9305311500.aa12036@giasbin01.vsnl.net.in>
Status: RO

Harshini
 Hi! How are you?
 Have a nice day at club house games show
 I miss you all. Deepika and I will come to see you
 Bye ! Sindhu

Note : Read messages are not held in the system mailbox when you exit the mail program.
 If you want to save these messages after reading them, you must do so before exiting the mail program.
 To read the next message, press the return key. The mailer responds with

Message 2:

From deepika Wed Jul 31 15:01:48 1993
 From: deepika@yahoo.com (B Deepika)
 X-Mailer: SCO System V Mail (version 3.2)
 To: harshini
 Subject: Hi! Ranjani how are you
 Date: Wed, 31 Jul 93 15:01:47 NZT
 Message-ID: <9305311501.aa12046@giasbin01.vsnl.net.in>
 Status: RO
 Hi ! Fine thank you
 Deepika

Viewing mail headers

The **header** command in compose mode lists the headers of all mail messages. The message header contains

- N if the message is new
- the message number (1, 2, 3.. n)
- the username of the sender
- the date and time of arrival
- the number of lines/the number of bytes in the message
- the subject field contents

Type the header command to list the available messages. Your display should now look like this: & header

```
N 3 harita      Wed Jul 31 15:02 10/299
N 2 deepika     Wed Jul 31 15:01 9/278   Hi! Ranjani how are you
>N 1 sindhuja   Wed Jul 31 15:00 12/415   How are you
&
```

Reading Specific Messages

The **type** command in compose mode is used to view specific messages. As an argument, it accepts the message number or senders name. If no argument is given, it displays the current selected message denoted by the symbol >.

Examples

type 4	types message 4
type harshini	types the message from user harshini
type	types the current highlighted message

4.5 For the above example if you give the command **type** without arguments.
 Message number 1 from sindhuja will be displayed.

Mail Message Printing

The **l** command is used for printing specified mail messages.

- 4.6

For printing all mail messages received from the user *harshini*, Command is **I harshini**
- 4.7

For printing the current message Command is **I**

Mail Message Saving

The **hold** or **preserve** command always saves the messages in the system mailbox. The messages will be visible the next time you enter mail. When you leave the mail with the **quit** command, the message is normally saved in the person's mailbox. It will not appear in the list of mail messages the next time, when you invoke the mail program.

- 4.8

Type the command **hold** to preserve the current message.
- Follow this command by using the **header** command to display the message headers. What do you see on screen for the present example?
Do you see any P symbol, if so, where? what does this mean?

Current Mail Message Saving to a file

Quite often, users wish to store the important messages in a file. Such a thing is facilitated by the **save** command and by specifying a file to save the message in.

s myfolder
saves the current message in the file *myfolder*. **This file is now treated as a mail folder by the mail program.**

- 4.9

If you **type** the command
s 4 thankyou
Message number 4 gets saved in a mail folder named *thankyou*.

Some Useful Commands

- | | |
|-----------------------------|--|
| delete sarada | deletes the message from user sarada |
| delete 3 | deletes message number 3 |
| reply 4 | activates reply to message number 4 |
| reply | activates reply to the current message |
| f 11 harshini | forwards message number 11 to user harshini |
| f 5 harshini deepika | forwards message number 5 to harshini and deepika |
| mail -f myfolder | mail will use a different mail folder |
| !ls -la | do a directory listing |
| !sh | run a shell |
- Note : Last two commands starts with ! are actually to invoke shell commands from mailer

Editing in Compose Mode

The mail program accepts special key sequences in the compose mode. These commands have to begin with the tilde (~) symbol. Using the command sequence ~v invokes the UNIX editor **vi**, and permits you to edit the message you have entered. After exiting vi, you are back into mail compose mode, where you can continue entering lines or send the message by pressing ctrl-d.

- 5.10

IF You Type
~?
then press ~p

the following command whilst in compose mode.
It displays list of available escape sequences
it lists previous entered lines whilst in compose mode,

Communication to Remote Sites

You can now practice sending mails across using terminals. For example, send a mail from the username “kartik” to username “hemant” and in verse. Practice the commands for naming, message saving in printing.

You can now send mail to other people at remote sites by specifying their specific mail address like the ones namely sai@mahindrabt.com or sai@hotmail.com etc. This only works if you have INTERNET access.

3.5 : Future Scope

In order to gain the experience coupled with a grasp of the underlying concepts, readers are advised to refer to the literature available in plenty. A customary course on operating systems and computer organization is a prerequisite to understand the intricacies of the UNIX operating system and, potentially, to use it in a practical environment. A list of books is cited in the end for your reference and further, use.

BIBLIOGRAPHY

- Alan Southerton, *Modern Unix*, Wiley, 1992.
- Anatole Olczak, *The Korn Shell User and Programming Manual*, Addison Wesley, 1992.
- Barry Rosenberg, *Korn Shell Programming Tutorial*, Addison Wesley, 1991.
- Brent Heslop and David Angell, *Mastering SunOS*, Sybex, 1990.
- Chris Negus and Larry Schumer, *Guide to the Unix Desktop*, Unix Press, 1992.
- Daniel Gilly and O' Reilly Staff, *Unix in a Nutshell*, O' Reilly, 2nd ed. 1992 (for System V and Sularis 2).
- Debra Cameron and Bill Rosenblatt, *Learning GNU Emacs*, O'Reilly, 1992.
- Don Libes and Sandy Ressler, *Life with Unix—A Guide for Everyon*, Prentice-Hall.
- Douglas Topham, *Portable Unix*, Wiley, 1992.
- Ed Dunphy, *The Unix Industry*, OED, 1991.
- Gail Anderson and Paul Anderson, *The Unix C Shell Field Guide*, Prentice-Hall, 1986.
- Harley Hahn, *A Student's Guide to Unix*, McGraw-Hill, 1993.
- Hewlett-Packard, *The Ultimate Guide to the vi and ex Text Editors*, Addison Wesley, 1989.
- James Gardner, *Learning Unix*, Sams, 1991.
- Jerry Peek, Tim O'Reilly and Mike Loukides (and other contributors), *Unix Power Tools*, O'Reilly/Bantam, 1993.
- John Levine and Margaret Levine Young, *Unix for Dummies*, IDG, 1993.
- John Valley, *Unix Desktop Guide to the Korn Shell*, Sams, 1992.
- Kaare Christian, *The Unix Operating System*, Wiley, 2nd ed. 1988.
- Kenneth Rosen, Richard Rosinski and James Farber, *Unix System V Release 4: An Introduction*, McGraw-Hill, 1990.
- Linda Lamb, *Learning the vi Editor*, O'Reilly, 1990.
- Lowell Arthur, *Unix Shell Programming*, Wiley, 2nd ed. 1990.
- M. Schoonover, J. Bowie and W. Arnold, *GNU Emacs Unix Text Editing and Programming*, Addison Wesley, 1992.
- Mark Sobell, *A Practical Guide to the Unix System V Release 4*, Benjamin Cummings, 2nd ed. 1991.
- Martin Arick, *Unix C Shell - Desk Reference*, QED Technical, 1992.
- Maurice Bach, *The Design of the Unix Operating System*, Prentice-Hall, 1986.
- Mitchell Waite, Donald Martin and Stephen Prata, *The Waite Group's Unix System V Primer*, Sams, 2nd ed. 1992.
- Morris Bolsky and David Korn, *The Korn Shell Command and Programming Language*, Prentice-Hall, 1989.

- Paul Abrahams and Bruce Larson, *Unix for the Impatient*, Addison Wesley, 1992.
- Pete Holsberg, *Unix Desktop Guide to Tools*, Sams, 1992.
- Peter Norton and Harley Hahn, *Peter Norton's Guide to Unix*, Bantam Computer, 1991.
- Ralph Roberts and Mark Boyd, *Desktop Guide to Emacs*, Sams, 1991.
- Richard Stallman, *GNU EMACS Manual*, Free Software Foundation, 7th ed. 1991.
- S. Leffler, M. McKusick, M. Karels and J. Quarterman, *The Design and Implementation of the 4.3 BSD Unix Operating System*, Addison Wesley, 1990.
- SSC staff, *SCC reference cards*, Specialized Systems Consultants, 1984-93.
- Stephen Coffin, *Unix System V Release 4: The Complete Reference*, McGraw-Hill, 1990.
- Stephen Kochan and Patrick Wood, *Unix Shell Programming*, Hayden, 1990.
- Ted Timar, *The Frequently Asked Questions List*, 93/3/18 (frequently updated).

