UNIX FUNDAMENTALS

COURSE MATERIAL

TABLE OF CONTENTS

Introduction to the Unix Operating System

Objectives

At end of this session, you will know about the

- **History of Unix and**
- **The salient features of Unix**

What is UNIX?
UNIX is:

- A multitasking, multi-user operating system
- The name given to a whole family of related operating systems and their most common application, utility, and compiler programs
- A rich, extensible, and open computing environment

The term *UNIX* refers to a powerful multitasking, multi-user operating system.

Once upon a time, not so long ago, everyone knew what an operating system (OS) was. It was the complex software sold by the maker of your computer system, without which no other programs could function on that computer. It spun the disks, lit the terminals, and generally kept track of what the hardware was doing and why. Application (user) programs asked the operating system to perform various functions; users seldom talked to the OS directly.

Today those boundaries are not quite so clear. The rise of graphical user interfaces, macro and scripting languages, suites of applications that can exchange information seamlessly, and the increased popularity of networks and distributed data--all of these factors have blurred the traditional distinctions. Today's computing environments consist of layers of hardware and software that interact together to form a nearly organic whole.
At its core (or, as we say in UNIX, in the kernel), however, UNIX does indeed perform the classic role of an operating system. Like the mainframe and minicomputer systems that came before, UNIX enables multiple people to access a computer simultaneously and multiple programs and activities to proceed in parallel with one another.

Unlike most proprietary operating systems, however, UNIX has given birth to a whole family of related, or variant, systems. Some differ in functionality or origin; others are developed by computer vendors and are specific to a given line of machines; still others were developed specifically as shareware or even freeware. Although these various flavors of UNIX differ from one another to some degree, they are fundamentally the same environments. All offer their own version of the most common utilities, application

programs, and languages. Those who use awk, grep, the Bourne shell, or make in one version of UNIX will find their old favorites available on other UNIX machines as well.

Those who do not care much for these programs, however, will find themselves free to substitute their own approach for getting various computing jobs done. A salient characteristic of UNIX is that it is extensible and open. By extensible, I mean that UNIX allows the easy definition of new commands, which can then be invoked or used by other programs and terminal users. This is practical in the UNIX environment because the architecture of the UNIX kernel specifically defines interfaces, or ways that programs can communicate with one another without having been designed specifically to work together.

Understanding Operating Systems

An operating system is an important part of a computer system. You can view a computer system as being built from three general components: the hardware, the operating system, and the applications. (See Figure 1.1.) The hardware includes pieces such as a central processing unit (CPU), a keyboard, a hard drive, and a printer. You can think of these as the parts you are able to touch physically. Applications are why you use computers; they use the rest of the system to perform the desired task (for example, play a game, edit a memo, send electronic mail). The operating system is the component that on one side manages and controls the hardware and on the other manages the applications.

Computer system components.

When you purchase a computer system, you must have at least hardware and an operating system. The hardware you purchase is able to use (or run) one or more different operating systems. You can purchase a bundled computer package, which includes the hardware, the operating system, and possibly one or more applications. The operating system is necessary in order to manage the hardware and the applications.

When you turn on your computer, the operating system performs a series of tasks, presented in chronological order in the next few sections.

Hardware Management, Part 1

One of the first things you do, after successfully plugging together a plethora of cables and components, is turn on your computer. The operating system takes care of all the starting functions that must occur to get your computer to a usable state. Various pieces of hardware need to be initialized. After the start-up procedure is complete, the operating system awaits further instructions. If you shut down the computer, the operating system also has a procedure that makes sure all the hardware is shut down correctly. Before turning your computer off again, you might want to do something useful, which means that one or more applications are executed. Most boot ROMs do some hardware initialization but not much. Initialization of I/O devices is part of the UNIX kernel.

Process Management

After the operating system completes hardware initialization, you can execute an application. This executing application is called a process. It is the operating system's job to manage execution of the application. When you execute a program, the operating system creates a new process. Many processes can exist simultaneously, but only one

process can actually be executing on a CPU at one time. The operating system switches between your processes so quickly that it can appear that the processes are executing simultaneously. This concept is referred to as time-sharing or multitasking.
When you exit your program (or it finishes executing), the process terminates, and the operating system manages the termination by reclaiming any resources that were being used.
Most applications perform some tasks between the time the process is created and the time it terminates. To perform these tasks, the program makes requests to the operating system, and the operating system responds to the requests and allocates necessary resources to the program. When an executing process needs to use some hardware, the operating system provides access for the process.

Hardware Management, Part 2

To perform its task, a process may need to access hardware resources. The process may need to read or write to a file, send data to a network card (to communicate with another computer), or send data to a printer. The operating system provides such services for the process. This is referred to as resource allocation. A piece of hardware is a resource, and the operating system allocates available resources to the different processes that are running.
See Table 1.1 for a summary of different actions and what the operating system (OS) does to manage them.

Table 1.1. Operating system functions.

Action	OS Does This
You turn on the computer	Hardware management
You execute an application	Process management
Application reads a tape	Hardware management
Application waits for data	Process management
Process waits while other process runs	Process management
Process displays data on screen	Hardware management
Process writes data to tape	Hardware management
You quit, the process terminates	Process management
You turn off the computer	Hardware management

From the time you turn on your computer until you turn it off, the operating system is coordinating the operations. As hardware is initialized, accessed, or shut down, the operating system manages these resources. As applications execute, request, and receive resources, or terminate, the operating system takes care of these actions. Without an operating system, no application can run and your computer is just an expensive paperweight.

The UNIX Operating System

The previous section looked at operating systems in general. This section looks at a specific operating system: UNIX. UNIX is an increasingly popular operating system. Traditionally used on minicomputers and workstations in the academic community, UNIX is now available on personal computers, and the business community has started to

choose UNIX for its openness. Previous PC and mainframe users are now looking to UNIX as their operating system solution. This section looks at how UNIX fits into the operating system model.

UNIX, like other operating systems, is a layer between the hardware and the applications that run on the computer. It has functions that manage the hardware and functions that manage executing applications.

The UNIX system is actually more than strictly an operating system. UNIX includes the traditional operating system components. In addition, a standard UNIX system includes a set of libraries and a set of applications.

Sitting above the hardware are two components: the file system and process control. Next is the set of libraries. On top are the applications. The user has access to the libraries and to the applications. These two components are what many users think of as UNIX, because together they constitute the UNIX interface.

The layers of UNIX.

The part of UNIX that manages the hardware and the executing processes is called the kernel. In managing all hardware devices, the UNIX system views each device as a file (called a device file). This allows the same simple method of reading and writing files to be used to access each hardware device. The file system manages read and write access to user data and to devices, such as printers, attached to the system. It implements security controls to protect the safety and privacy of information. In executing processes, the UNIX system allocates resources (including use of the CPU) and mediates accesses to the hardware.
One important advantage that results from the UNIX standard interface is application portability. Application portability is the ability of a single application to be executed on various types of computer hardware without being modified. This can be achieved if the application uses the UNIX interface to manage its hardware needs. UNIX's layered design insulates the application from the different types of hardware. This allows the software developer to support the single application on multiple hardware types with minimal effort. The application writer has lower development costs and a larger potential customer base. Users not only have more applications available, but can rely on being able to use the same applications on different computer hardware.

UNIX goes beyond the traditional operating system by providing a standard set of libraries and applications that developers and users can use. This standard interface allows application portability and facilitates user familiarity with the interface.

The History of UNIX

How did a system such as UNIX ever come to exist? UNIX has a rather unusual history that has greatly affected its current form.

The Early Days

In the mid-1960s, AT&T Bell Laboratories (among others) was participating in an effort to develop a new operating system called Multics. Multics was intended to supply large-scale computing services as a utility, much like electrical power. Many people who worked on the Bell Labs contributions to Multics later worked on UNIX.

In 1969, Bell Labs pulled out of the Multics effort, and the members of the Computing Science Research Center were left with no computing environment. Ken Thompson, Dennis Ritchie, and others developed and simulated an initial design for a file system that later evolved into the UNIX file system. An early version of the system was developed to take advantage of a PDP-7 computer that was available to the group.

An early project that helped lead to the success of UNIX was its deployment to do text processing for the patent department at AT&T. This project moved UNIX to the PDP-11 and resulted in a system known for its small size. Shortly afterward, the now famous C programming language was developed on and for UNIX, and the UNIX operating system itself was rewritten into C. This then radical implementation decision is one of the factors that enabled UNIX to become the open system it is today.

AT&T was not allowed to market computer systems, so it had no way to sell this creative work from Bell Labs. Nonetheless, the popularity of UNIX grew through internal use at AT&T and licensing to universities for educational use. By 1977, commercial licenses for UNIX were being granted, and the first UNIX vendor, Interactive Systems Corporation, began selling UNIX systems for office automation.

Later versions developed at AT&T (or its successor, Unix System Laboratories, now owned by Novell) included System III and several releases of System V. The two most recent releases of System V, Release 3 (SVR3.2) and Release 4 (SVR4; the most recent version of SVR4 is SVR4.2) remain popular for computers ranging from PCs to mainframes.

All versions of UNIX based on the AT&T work require a license from the current owner, UNIX System Laboratories.

Berkeley Software Distributions

In 1978, the research group turned over distribution of UNIX to the UNIX Support Group (USG), which had distributed an internal version called the Programmer's Workbench. In 1982, USG introduced System III, which incorporated ideas from several different internal versions of and modifications to UNIX, developed by various groups. In 1983, USG released the original UNIX System V, and thanks to the divestiture of AT&T, was able to market it aggressively. A series of later releases continued to introduce new features from other versions of UNIX, including the internal versions from the research group and the Berkeley Software Distribution.

While AT&T (through the research group and USG) developed UNIX, the universities that had acquired educational licenses were far from inactive. Most notably, the Computer Science Research Group at the University of California at Berkeley (UCB) developed a series of releases known as the Berkeley Software Distribution, or BSD. The original PDP-11 modifications were called 1BSD and 2BSD. Support for the Digital Equipment Corporation VAX computers was introduced in 3BSD. VAX development continued with 4.0BSD, 4.1BSD, 4.2BSD, and 4.3BSD, all of which (especially 4.2 and 4.3) had many features (and much source code) adopted into commercial products.

UNIX and Standards

Because of the multiple versions of UNIX and frequent cross-pollination between variants, many features have diverged in the different versions of UNIX. With the increasing popularity of UNIX in the commercial and government sector came the desire to standardize the features of UNIX so that a user or developer using UNIX could depend on those features.

The Institute of Electrical and Electronic Engineers (IEEE) created a series of standards committees to create standards for "An Industry-Recognized Operating Systems Interface Standard based on the UNIX Operating System." The results of two of the committees are important for the general user and developer. The POSIX.1 committee standardizes the C library interface used to write programs for UNIX. (See UNIX Unleashed, Internet Edition, Chapter 6, "The C and C++ Programming Languages.") The POSIX.2 committee standardizes the commands that are available for the general user.

In Europe, the X/Open Consortium brings together various UNIX-related standards, including the current attempt at a Common Open System Environment (COSE) specification. X/Open publishes a series of specifications called the X/Open Portability. The MOTIF user interface is one popular standard to emerge from this effort.

The United States government has specified a series of standards based on XPG and POSIX. Currently, FIPS 151-2 specifies the open systems requirements for federal purchases.

Various commercial consortia have attempted to negotiate UNIX standards as well. These have yet to converge on an accepted, stable result.

UNIX for Mainframes and Workstations

Many mainframe and workstation vendors make a version of UNIX for their machines. We will be discussing several of these variants (including Solaris from SunSoft, AIX from IBM and HP-UX from Hewlett Packard) throughout this book.

UNIX for Intel Platforms

Thanks to the great popularity of personal computers, there are many UNIX versions available for Intel platforms. Choosing from the versions and trying to find software for the version you have can be a tricky business because the UNIX industry has not settled on a complete binary standard for the Intel platform. There are two basic categories of UNIX systems on Intel hardware: the SVR4-based systems and the older, more established SVR3.2 systems.

SVR4 vendors include NCR, IBM, Sequent, SunSoft (which sells Solaris for Intel), and Novell (which sells UnixWare). The Santa Cruz Operation (SCO) is the main vendor in the SVR3.2 camp.

Source Versions of "UNIX"

Several versions of UNIX and UNIX-like systems have been made that are free or extremely cheap and include source code. These versions have become particularly attractive to the modern-day hobbyist, who can now run a UNIX system at home for little investment and with great opportunity to experiment with the operating system or make changes to suit his or her needs.

An early UNIX-like system was MINIX, by Andrew Tanenbaum. His book Operating Systems: Design and Implementations describes MINIX and includes a source listing of

the original version of MINIX. The latest version of MINIX is available from the publisher. MINIX is available in binary form for several machines (PC, Amiga, Atari, Macintosh, and SPARCStation).

The most popular source version of UNIX is Linux (pronounced "lin nucks". Linux was designed from the ground up by Linus Torvalds to be a free replacement for UNIX, and it aims for POSIX compliance. Linux itself has spun off some variants, primarily versions that offer additional support or tools in exchange for license fees. Linux has emerged as the server platform of choice for small to mid-sized Internet Service Providers and Web servers.

Making Changes to UNIX

Many people considering making the transition to UNIX have a significant base of PC-based MS-DOS and Microsoft Windows applications. There have been a number of efforts to create programs or packages on UNIX that would ease the migration by allowing users to run their existing DOS and Windows applications on the same machine on which they run UNIX. This is a rapidly changing marketplace as Microsoft evolves its Windows and Windows NT operating systems.

Summary

UNIX has a long history as an open development environment. More recently, it has become the system of choice for both commercial and some personal uses. UNIX performs the typical operating system tasks, but also includes a standard set of commands and library interfaces. The building-block approach of UNIX makes it an ideal system for creating new applications.

Unix System Architecture

Objectives

At the end of this session you will know

- **About the layers of Unix**
  - **The Kernel**
  - **The shell**
- **How the kernel and the shell interact &**
- **The Functions and Features of a Shell**

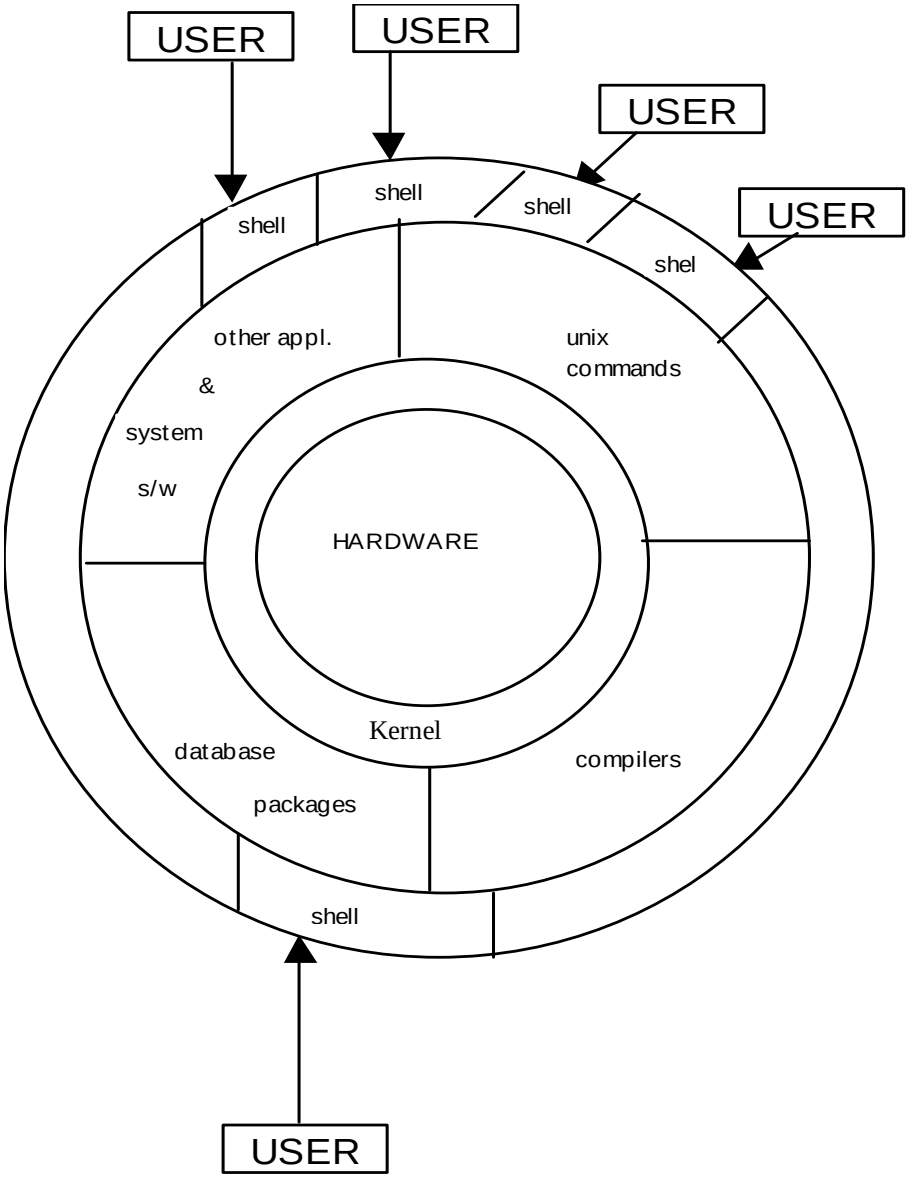
The entire Unix system is supported by a handful of essentially simple concepts. Foremost amongst them is the division of labor between two agencies, one interacting with the user, and the other with the machine's hardware. This is achieved by two abstract constituents – the Kernel and the shell. The relationship between the two is depicted below.

The UNIX Kernel

Technically speaking, the UNIX kernel is the operating system – a collection of programs mostly written in C, which communicate with the hardware directly. It provides the basic full time software connection to the hardware.

By full time, It means that the kernel is always running while the computer is turned on. When the system boots up, the kernel is loaded. Likewise, the kernel is only exited when the computer is turned off.

The UNIX kernel is built specifically for a machine when it is installed. It has a record of all the pieces of hardware it needs to talk to and knows what languages they speak (how to turn switches on and off to get a desired result). Thus, a kernel is not easily ported to another computer. Each individual computer will have its own tailor- made kernel. And if the computer's hardware configuration changes during its lie, the kernel must be "rebuilt" (told about the new pieces of hardware).



However, though the connection between the kernel and the hardware is "hardcoded" to a specific machine, the connection between the user and the kernel is generic. That is the beauty of the UNIX kernel. From your perspective, regardless of how the kernel interacts with the hardware, no matter which UNIX computer you use, you will have the same kernel interface to work with. That is because the hardware is "hidden" by the kernel.

The kernel also handles memory management, input and output requests, and process scheduling for time-shared operations (we'll talk more about what this means later).
To help it with its work, the kernel also executes daemon programs, which stay alive as long as the machine is turned on and help perform tasks such as printing or serving web documents.

However, the task of hiding the hardware is a pretty much full time job for the kernel. As such, it does not have too much time to provide for a fancy user-friendly interface. Thus, though the kernel is much easier to talk to than the hardware, the language of the kernel is still pretty cryptic.

Fortunately, the UNIX operating system has built in "shells" which wrap around the kernel and provide a much user-friendlier interface. Let's take a look at shells.

The shell

The shell, is the "sleeping beauty". Technically another Unix command, it is the interpreter of the user requests. Computers don't have any inherent capability of translating commands into action. An interpreter is required, and the shell handles that job in Unix. It takes a command from the user, deciphers it, and by exchanging information, communicates with the kernel to see that the command is executed. It is actually the interface between the user and the kernel, which effectively insulates the user from the knowledge of kernel functions. The shell is the agency which takes care of the features of redirection, using the > and | symbols. It also has a programming capability of its own.

How the Kernel and the Shell Interact

When a UNIX system is brought online, the program unix (the Kernel) is loaded into the computer's main memory, where it remains until the computer is shut down. During the bootup process, the program init runs as a background task and remains running until shutdown. This program scans the file /etc/inittab, which lists what ports have terminals and their characteristics. When an active, open terminal is found, init calls the program getty, which issues a login: prompt to the terminal's monitor. With these processes in place and running, the user is ready to start interacting with the system.

UNIX Calls the Shell at Login

During login, when you type your user name, getty issues a password: prompt to the monitor. After you type your password, getty calls login, which scans for a matching entry in the file /etc/passwd. If a match is made, login proceeds to take you to your home directory and then passes control to a session startup program; both the user name and password are specified by the entry in /etc/passwd. Although this might be a specific application program, such as a menu program, normally the session startup program is a shell program such as /bin/sh, the Bourne shell.

From here, the shell program reads the files /etc/profile and .profile, which set up the system-wide and user-specific environment criteria. At this point, the shell issues a command prompt such as $.
When the shell is terminated, the kernel returns control to the init program, which restarts the login process. Termination can happen in one of two ways: with the exit command or when the kernel issues a kill command to the shell process. At termination, the kernel recovers all resources used by the user and the shell program.

The Shell and Child Processes

In the Unix system, there are many layers of programs starting from the kernel through a given application program or command.

After you finish logging on, the shell program layer is in direct contact with the kernel. As you type a command such as ls, the shell locates the actual program file, /bin/ls, and passes it to the kernel to execute. The kernel creates a new child process area, loads the program, and executes the instructions in /bin/ls. After program completion, the kernel recovers the process area and returns control to the parent shell program. To see an example of this, type the following command:

$ps

This lists the processes you are currently running. You will see the shell program and the ps program. Now type the following:

$sleep 10 &

$ps

The first command creates a sleep child process to run in background, which you see listed with the ps command. Whenever you enter a command, a child process is created and independently executes from the parent process or shell. This leaves the parent intact to continue other work.

The Functions and Features of a Shell

It doesn't matter which of the standard shells you choose, because they all have the same purpose: to provide a user interface to UNIX. To provide this interface, all the shells offer the same basic characteristics:

- Command-line interpretation
- Reserved words
- Shell meta-characters (wild cards)
- Access to and handling of program commands
- File handling: input/output redirection and pipes
- Maintenance of variables
- Environment control
- Shell programming

Summary

The shell provides an interface between the user and the heart of UNIX--the kernel. The shell interprets command lines as input, makes filename and variable substitution, redirects input and output, locates the executable file, and initiates and interfaces programs. The shell creates child processes and can manage their execution. The shell maintains each user's environment variables. The shell is also a powerful programming language.

Introduction to the Unix File system

Objectives

At the end of this session, you will

- **Know about the internal structure of the filesystem &**
- **The different types of files in Unix**
  - **Regular files**
  - **Directory files**
  - **Device files**

The UNIX system has a very large number of files. If they are not organized in a systematic manner, it becomes impossible to locate them. It should be possible to group a set of files created by a user, and house them conveniently so that they don't get mixed up with files of other users, or even files used by the system. It should also be possible for two or more users to use the same filename without conflict and access other files not belonging to them, without infringing on security.

UNIX provides us an elaborate storage system with separate "compartments", so that one can place himself in any of these compartments, or transfer files from one compartment to another.

THE FILE

A UNIX file is a storehouse of information – it is simply a sequence of characters. UNIX places no restriction on the structure of a file, and you don't need to assume a predefined structure to work with it. A file contains exactly those bytes that you put into it, whether it represents a source program, other text or executable code. A file neither contains its size nor its attributes. It doesn't even contain the end-of-file mark. All the file attributes are kept in a separate location in the disk specially earmarked for this purpose. When a file is called up by a command or program it is this area that is looked up first before the contents are accessed.

FILE TYPES IN UNIX

Ordinary file

This type of file consists of a stream of data resident on some permanent magnetic media. This includes all data , source programs, object and executable code, all UNIX programs, as well as any files created by user. Commands like pwd, ls, etc, are treated as ordinary files or regular files.

All text files also belong to this type. Any text file in UNIX contains a sequence of lines, with each line terminated by the LF (ASCII octal value 012) which is the newline character in UNIX. When you edit a file, hitting the <Enter> key creates the LF character.

Directory File

A directory can be defined as a storehouse where other programs and sub-directories reside. A directory contains no data, but keeps an account of all the files and sub-directories that it contains. The unix file system is organized with a number of such directories and sub-directories, and you can also create them as and when you need.

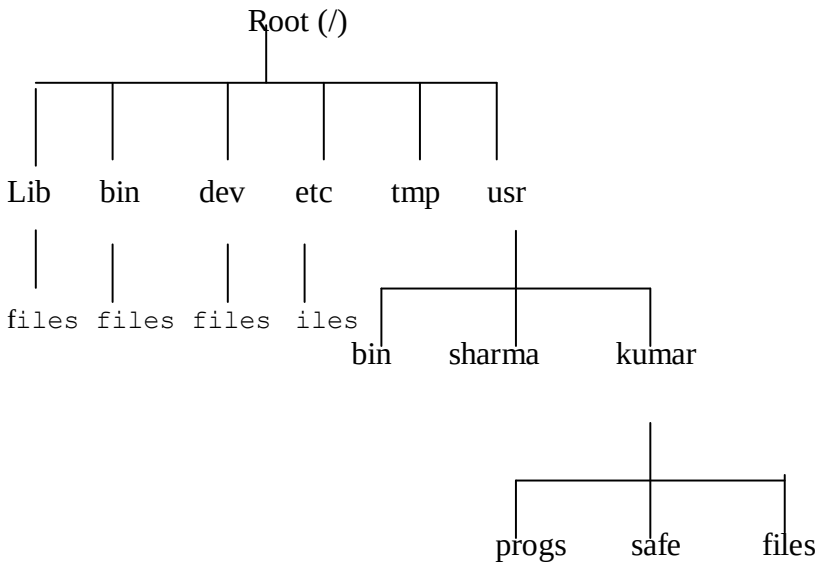
A directory file contains two fields – the name of a file and a pointer to a separate disk area, which contains the file's attributes. When a file is created or deleted, the directory file is automatically updated by the kernel with the relevant information about the file.

Device File

In UNIX even the physical devices are treated as files. The definition includes printers, tapes, floppy drives, hard disks and terminals. The device file is special in the sense that any output directed to it will be reflected onto the respective I/O device associated with the filename. The kernel takes care of this by mapping special filenames to their respective devices.

The Structure Of The File System

All files in UNIX are "related" to one another. The file system in UNIX is a collection of all these related files (ordinary, directory, device) organized in a hierarchical (an inverted tree) structure. This system is visually represented as given below,



The top of the tree, indicated by the root / serves as the reference point for all the files.
Root is actually a directory file and it has number of sub-directories under it, which intern have more sub-directories and other files.
Every file, apart from root, must have a parent, and it should be possible to trace the ultimate parentage of a file to root. If a file can't be traced to its ultimate ancestor, then it is not part of the file system.

The directories shown directly under root are normally found in every UNIX system in addition to others.

The directory /bin,/dev. /etc are system directories and you shouldn't try to tamper with these. The other directories like /usr, /tmp are the user directories.

The commonly used commands like who, cat, wc are stored in the /bin and /usr/bin directories and made available to all the users.
The /dev contains the device files of all the hardware devices.
/etc contains those utilities mostly used by the system administrator.
/tmp is used by some UNIX utilities (vi) as well as by user to store temporary files.
/usr contains all the files created by the user including his login directory.

Internal Structure Of The File System

In the UNIX system, every file has a table associated with it, which is stored in a special area of the disk, which is known as identification node (i-node). The i-node describes the file uniquely.

Every file system consists of a sequence of blocks, each block consisting of 512 bytes. Some of these blocks are not allotted to the user, and are reserved exclusively for the use of the kernel. The file system breaks the disk in to four segments as follows

1. The Boot Block

The first block, numbered 0, is called the boot block, which is normally unused by the file system, and set aside for the booting procedure. This is true for the main file system. For the other file systems, this is left unused.

2. The Super Block

This is the block numbered 1, and is used to control the allocation of disk blocks. This contains the details of the active file system, like

a. the size of the file system
b. the details of the free blocks and I-nodes

3. The i-node Blocks

This segment includes the blocks numbered 2 onwards, up to a number determined during the creation of the file system and contains the most information pertaining to the files. Every file in the file system will invariably have an entry in this area, identified by a 64-byte structure referred to as the i-node. The complete list of I-nodes is known as the I-list. Every I-node is identified by a unique number which simply references the position of the I-node in the list. This number called the I-number is nothing but the internal name of a file.

An inode is 64 bytes long. So in one physical block, there are 8 I-nodes. Each I-node contains the following attributes of the file.

The file type (regular, directory, etc.)
The number of links (the no. of aliases the file has)
The owner (the user-id number of the owner)
The group (the group-id number)
The file mode (the triad of the three permissions)
The number of bytes in the file
The date and time of creation
The date and time of last modification
The date and time of last access
An array of 13 pointers to the file

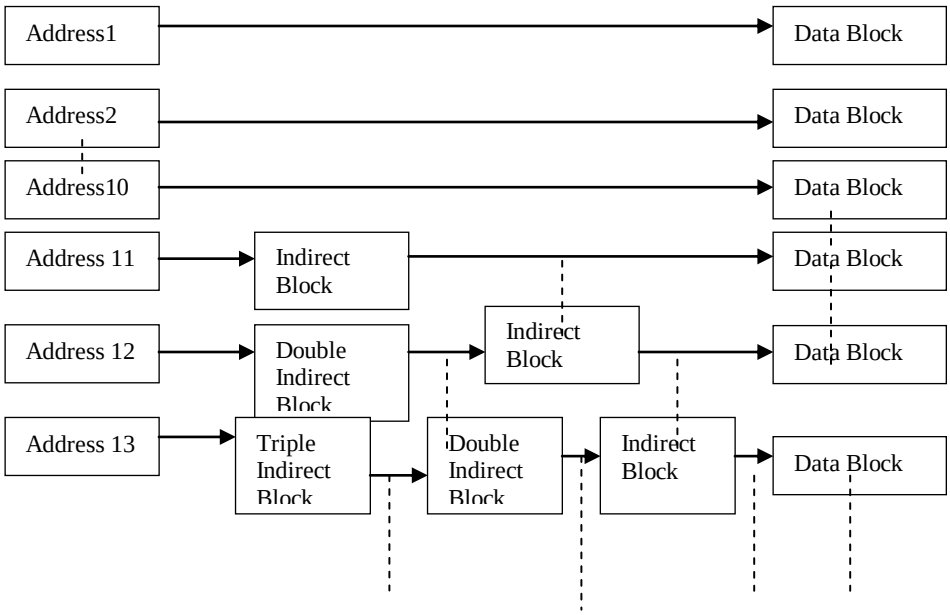
4. Data Blocks

The final segment contains a long chain of blocks for storing the contents of files. The Unix file system stores the data in physical blocks of 512 bytes. These blocks commence from the point the I-node blocks terminate. A Unix file is a sequentially organized set of blocks scattered throughout the disk. It is the array of 13 disk block addresses, which keep track of all the disk blocks containing file segments.

For regular files, the first 10 pointers are direct addresses of a file. i.e. they contain the addresses of the first ten storage blocks of a file.

If the file grows beyond 10 blocks, 11^{th} block is allocated to specify a disk block, which contains the addresses of the next 256 blocks of the file. This is called the indirect block.

When the file size grows beyond this size, the 12^{th} block, known as double indirect block is allocated. This contains the addresses of another block, which intern contains the addresses of 256 indirect blocks.
If this space is also not enough, a triple indirect block is allocated.
The following dig. Shows the Data organization in the UNIX file system.



Summary

UNIX provides an elaborate file system with separate "compartments", so that one can place himself in any of these compartments, or transfer files from one compartment to another. The I-node is the structure that contains all information about a file. The directory is just another file, which contains the filename and the I-number. The directory and the I-node tables are among the most sophisticated features of the file system, and together form a relational database with the I-number as the common field connecting the two tables.

Starting to Work With Unix

Objectives

At the end of this session, you will be able to

- **Login to the server and**
- **Disconnect from the server by logging off.**

Several people can be using a UNIX-based computer at the same time. In order for the system to know who you are and what resources you can use, you must identify yourself. In addition, since UNIX expects to communicate with you over a terminal (or a PC running terminal-emulation software), your terminal and the UNIX system must establish the ground rules that will govern the transfer of information. The process of establishing the communications session and identifying you is known as "logging in."

Once after an account is created for you, you can use the Unix system after logging in. Type your login name when you see the login prompt.

login : kumar

If a password has been allotted to you, the system will flash the following message.

password :

The password will not be echoed on the screen. If you enter an incorrect password or login name it will display the message

Login incorrect
Login :

You will be permitted to reenter your password a fixed number of times before you get it right.

If your login information is right, it will display a welcome message, Terminal type and then show you the $ prompt

Terminal type is ansi

$

Logging Out

When you are done using the system, you should log out. This will prevent other people from accidentally or intentionally getting access to your files. It will also make the system available for their use.
The normal way to log out from almost any shell is to type exit. This causes your shell to exit, or stop running.

$ exit

Summary

Unix system can be used only by those persons who maintain an "account" with the system and Unix maintains the list of accounts separately in the computer. Any user has to login to the Unix Server to use it and has to logout after finishing the work.

Directory Related Commands

Objectives

At the end of this session, you will be able to do the following

- **Find the current working directory**
- **Create your own directories**
- **Navigate through the directories using absolute and relative path**
- **List out the directory contents**
- **Delete the directories**

A directory can be defined as a storehouse where other programs and sub-directories reside. A directory contains no data, but keeps an account of all the files and sub-directories that it contains.

When you log in to the system, UNIX automatically places you in a directory called home directory, which normally has the same name as that of the login name.

To find out the directory where you are currently working, use the command

```
$ pwd
/usr/kumar
$_
```

Making A Directory – The mkdir command

Directories can be created by using the mkdir (make directory) command.

```
$ mkdir pis
```

A directory pis is created under the current directory.

A number or directories can be created as follows

```
$ mkdir pis pis/progs pis/data
```

Changing Directories – The cd command

You can move around in the file system by using the cd (change directory) command.

```
$ pwd
/usr/kumar
$ cd progs
$pwd
/usr/kumar/progs
$
```

You can use either the absolute or the relative path to go to a particular directory. When a path name begins with a / (root) , it means that it is the absolute path, otherwise it is relative with respect to the current directory.

If the pwd tells that /usr/kumar is your current directory, then

/usr/kumar/progs → absolute path
progs → Relative path

$ cd /	→ Takes you to the root
$ cd	→ Takes you to the home directory
$ cd ..	→ Takes you to the parent directory
$ cd <directory>	→ Takes you to the specified dir under the current dir

Listing out the Directory contents

Use the command ls to list out the contents of a directory.

$ ls

–x	→ Displays Multi columnar output
–F	→ Marks executables with * and directories with /
–l	→ Long listing showing the attributes of the file
–a	→ Displays all the files .,.. and those beginning with a dot.
–R	→ Recursive Listing of all the files in the sub-directories
-r	→ Sorts the files in the reverse order
-t	→ Sorts the files by modification time
-u	→ Sorts the files by access time
-i	→ Shows the inode number of a file
-s	→ Displays the no. of blocks used by a file

Removing a Directory – The rmdir command

Use the command rmdir to delete the directory

$ rmdir pis

But, the directory should be empty, before deletion.

Summary

A directory is nothing but a storehouse where other programs and sub-directories reside. A directory contains no data, but keeps an account of all the files and sub-directories that it contains. The commands mkdir, rmdir, cd can be used to work with the directories.

File Related Commands

Objectives

At the end of this session you will be able to:

- **Create & Display the contents of the files**
- **Copy, rename & remove the files.**

Create & Display the contents of the files:

cat command is used to display the contents of the file. cat, like several other UNIX commands, also accepts more than one filename as arguments

$cat chap01 chap02

The contents of the second file are shown immediately after the first file without any message or header information. cat is normally used for displaying text files only. If you have non-printing ASCII characters in your input, use the –v option to display these characters.

$ cat > abc
A > symbol following the command means that the output
goes to the filename following it. cat used in this way represents
a rudimentary editor.
<ctrl-d>

$ cat abc #displays the contents of the file abc
A > symbol following the command means that the output
goes to the filename following it. cat used in this way represents
a rudimentary editor.
$_

Copying a file – The cp Command

The cp command copies a file or a group of files. cp creates an exact image of the file on the disk with a different name.

```
$ cp chap01 unit1              #copies the file chap01 to the file unit1
$_
```

If the destination file doesn't exist, it will be created before copying takes place. If not, it will be simply overwritten.

```
$cp chap01 progs/unit1         #copies the file chap01 to the directory progs. The
                                 name of the copied file will be unit1
$cp chap01 progs               # copies the file chap01 to the directory progs. The
                                 name of the copied file will also be chap01
$ cp chap* progs               #copies all the files matching the pattern to the
                                 directory progs
$cp chap01 chap02 chap03  progs     #copies the 3 files to the dir progs
```

One important point that should be remembered is that you can't copy a file if it is read protected, and can't create a copy if the destination file or directory is write protected. You can't also copy a file into a directory you don't own. For e.g kumar cant copy the files he owns to a directory which he doesn't own, say /usr/sharma

```
$cp chap01 /usr/sharma
cp: can't create /usr/sharma/chap01
```

even though sharma may usually be able to copy these files from kumar's dir

```
$pwd
/usr/sharma
$cp /usr/kumar/chap01 .
$ls –l chap01
-rw-r—r--     1 sharma group     19514  May  10 23:45 chap01
$_
```

When sharma copies the file chap01 owned by kumar, he then becomes the owner of the copy. He can now manipulate the file permissions with chmod.

```
$cp –i chap01 unit1            #prompts you before overwriting the file
cp: overwrite unti1? y
$_
$cp –r progs  newprogs              #does recursive copy
```

Deleting the Files – The rm command

Files can be deleted with rm command. When invoked without options, it deletes the files specified in the command line.

```
$ rm chap01 chap02 chap03
```

rm won't normally remove a directory, but it can remove files from one. You can remove two chapters from the prgs directory without having to "cd" to it.

```
$ rm progs/chap01 progs/chap02
$_
$rm *        #Deletes all the files. But doesn't give any prompt messages.
$ rm –i *  # prompts before deleting the files
$ rm –i chap01 chap02
chap01: ?y
chap02: n

$rm –r *          # does recursive deletion
```

Renaming the files - The mv command

The mv command simply renames a file or a group of files. It is syntax is similar to the cp command.

```
$ mv chap01 man01
$_
```

If the destination file doesn't exist, it will be created. If it is there, it will be overwritten.

```
$mv chap01 chap02 chap03  progs
```

Moves the 3 files to the directory progs.

```
$mv pis  perdir                    # renames the directory pis to  perdir
```

Summary

A file in Unix contains just what you put into it, and the Unix system maintains the attributes of the file in a separate area of the disk and not in the file itself. The commands cat, cp and rm can be used to manipulate the files in Unix system.

File Permissions

Objectives

At the end of this session, you will be able to:

- **Learn about the file and directory permissions**
- **Learn about symbolic and octal notations**
- **Learn to set the file and directory permissions using the two notations**
- **Change the owner and groupid of a file**

UNIX follows a three-tiered file protection system, which determines the access rights that you have for a file. Each tier represents a category, and consists of a string of r's, w's and x's to represent three types of permissions.

r → read
w → write
x → execute
} permissions

You need r permission to read the file contents using say cat command, w to direct some output to the file, and x to execute the file as a program.

If any of these permissions don't apply to a specific category, then a – (hyphen) occupies the respective slot.

The command
$ ls –l
will show the list of permissions for each category .
$ ls –l note
-rw-r—r-- 1 kumar group 0 may 10 20:30 note

The – at the leftmost position indicates an ordinary file. The first set of three permissions (rw-) pertains to the owner of the file. In this case since the file was created by kumar, he is the owner of the file. The owner has both the read and write but no execute permissions on this file.

The second set of three permissions (r--) indicates only read permission for the group which owns the file, in this case, the default **group** to which kumar belongs.

The third set of three permissions (r--) indicates only read permission for those who is neither the owner nor belong to the group of the owner. This category is referred to the others.

The chmod command is used to set the three permissions for all the three categories of users of a file. It can only be used by the owner of the file and uses the following syntax.

chmod <category> <operation> <permission> <filename(s)>

The chmod command uses the following abbreviations

Category	**Operation**	**Attribute**
u-user	+ assign permission	r-read permission
g-group	- remove permission	w-write permission
o-Others	=assign absolute permission	x-execute permission
a-all		

$ chmod u+x note # assign execute permission to the owner of the file note
$ chmod ugo+x note # assign execute permission to all the users
$ chmod u+x note note1 note3
$ chmod u-x,go+r note
$ chmod o+wx note
$ chmod ugo=r note
$ chmod a=r note
$ chmod =r note

} Assign r permission to all the users

The Octal Notation

The octal notation of chmod command describes the category as well as the permission. The following octal codes are used to indicate rwx.

4 → read
2 → write
1 → execute

You can just add the appropriate codes when more than one permission has to be granted.

$ chmod 666 note
Here 6 indicate read and write permissions (4+2) to all the users.
$chmod 777 note # Assigns all permissions to all
$ chmod 000 note # removes all permissions from all categories

$ chmod –R a+x progs

Applies the chmod command recursively to all files and sub-directories in a directory. This requires only the name of the directory.

Directory Permissions

Every set of directory permissions has the d as the first character of the three-tiered sequence. The directory permissions are different from the files.

The read permission for a directory means that the list of filenames stored in that directory is accessible. If a directory has read permission, you can use ls to list out its contents. If you remove the read permission, the standard utilities won't be able to read the directory information. However, it doesn't prevent you from reading the files separately.

The absence of write permission for a directory implies that you are not permitted to create or remove files in it.

Execution privilege of a directory means that a user can pass through the directory in searching for sub-directories.

File Ownership – The chown and chgrp Commands

When a file is created, the user becomes the owner of the file, and the group to which the user belongs becomes the group owner. Thus, if sharma creates a file **notex**, he becomes the owner of the file. But when kumar copies the file from sharma's home directory, then the ownership of the copy is vested with kumar.

To change the ownership of a file or a directory, you can use the chown command,

```
$ ls –l note
-rwxrw---x    1    kumar    group    0  May  10  20:30  note

$ chmod sharma note*

$ ls –l note
-rwxrw---x    1    sharma group      0  May  10  20:30  note
```

But, once ownership is surrendered, it can't be reinstated.

Similarly to change the group ownership of the file, use the chgrp command.

To pass on the group ownership of your emp.lst file to bin, use the command

```
$ chgrp bin emp.lst
```

Summary

Unix follows a three-tiered file protection system, which determines the access rights that you have for a file. Each tier consists of a string of rs, ws and xs to represent the three types of permission. The chmod command is used in two modes symbolic and octal mode to change the file permissions.

Introduction to Editors

Objectives

At the end of this session, you will be able to

- **Use the vi editor**
- **Learn about the different modes in vi**
- **Learn all the editing commands**
- **Set the configuration for the vi editor in the .exrc file**

Vi is a full-screen editor available with all the UNIX systems, and is widely acknowledged as one of the most powerful editors available in any environment. The terminal type that you use is made available to vi by the environment variable TERM. Using this name, it searches either the file /etc/termcap or the respective terminal file in /usr/lib/terminfo/ to select the terminal characteristics

THE THREE MODES

A vi session begins by invoking the command vi with or without a filename

$ vi visfile

You are presented with a full empty screen, each line beginning with a ~. This is vi's way of indicating that they are non-existent lines. For text editing, vi uses 24 of the 25 lines that are normally available in a terminal. The last line is reserved for the ex commands that you can also use here, as well as the messages that the system throws out. vi works in 3 different modes as follows,

Input mode – where any key pressed is entered as a text
Command mode – where keys are used as commands to act on the text
ex mode – where ex commands can be entered in the last line to act on the text

To display the current mode , use the command

:set showmode

Following are the commands used in the input mode

Command	Function
i	Insert text to left of cursor
l	Insert text at the beginning of line
a	Appends text to the right of cursor
A	Appends text at the end of line
o	Opens line below
O	Opens line above
rch	Replaces single character under cursor with char ch(no esc)
R	Replaces text from cursor to right
s	Replaces single character under cursor with any no. of chars
S	Replaces entire line

Saving Text and Quitting – The Last Line (ex) Mode

You can switch from the command mode to the ex mode by pressing a : which appears as the ex prompt in the bottom line.You are now free to enter any ex command at this prompt. You can use the following commands

:40 to move to the 40th line
:w <enter> to save the file
:x<enter> to save and quit the editor

Repeat Factor

Vi uses to repeat factor to repeat the instruction ‘n’ no. of times. This is done by prefixing the command with a number.

30i*

will insert a series of 30 *s in one line.

The following are the commands in the command mode

Command	Function
x	Deletes the character under the cursor
dd	Deletes the current line
J	Removes the newline character between the joined lines
u	Reverse the last change
U	Reverse all the changes made to the current line
h or backspace	Moves cursor left
j	Moves cursor down

k	Moves cursor up
l or spacebar	Moves cursor right
^	Moves cursor to beginning of word of line (no repeat factor)
0 or 1	Moves cursor to the beginning of line (no repeat f actor)
$	Moves cursor to end of line
b	Moves cursor back to beginning of word
e	Moves cursor forward to end of word
w	Moves cursor forward to beginning of word

Paging & Scrolling

<control – f>	Full page forward
<control – b>	Full page backward
<control – d.>	Half page forward
<control – u>	Half page backward
<control – l>	Redraw page screen

Searching for a pattern

/pattern<enter>	Searches for the given pattern in the forward direction
?pattern <enter>	Searches for the given pattern in the backward direction
n	Repeat the previous search in forward direction
N	Repeat the previous search backwards
fch	Moves cursor forward to first occurrence of character ch in the current line

Operators

Apart from commands, vi also uses a number of operators which can be used in combination with commands They are,

d	Delete
c	Change
y	Yank(copy)
!	Filter to act on text
p	Puts text below the cursor
P	Places the text above the cursor

Customizing ex/vi

Ex/vi can be tailored to behave in a way desirable to the user. There are number of commands available in ex mode from where we can set the vi environment suitable for writing programs. These commands can also be put in a file called **".exrc "** similar to the .profile which is maintained by each user.

```
:set all
:set number
:set nonumber
:set ignorecase
:set nomagic
:set showwatch
:set tabstop=6
```

All these settings can be set in the file called **.exrc**. When vi or ex is invoked, it looks for the file .exrc in the current directory. If the variable is assigned, then the initialization instruction are controlled by the variable EXINIT.

$ EXINIT=”set number tabstop=6 ignorecase”
$_

Options in vi

vi –r <filename>
Helps salvage as much of a file possible after system crash. So if you had a lot of unsaved editing when the crash occurred, you can execute the above command after the system is up. This doesn’t guarantee complete retrieval, but in most cases has been found to recover quite a bit of the file

$ vi +lineno <filename>
This will open the file and position the cursor at the specified line

$ vi + <filename>
Positions the cursor at the end of the file

$ vi +/pattern <filename>
Positions the cursor at the line matching the pattern

Summary

vi is a powerful editor, and probably the only standard full-screen editor available with Unix systems. It operates in three modes. The command mode to enter the commands so that they can operate on the text or manipulate the cursor motion. The input mode to insert, append, replace or change text and the ex mode where ex commands can be entered in the last line to act on the text.

General - Purpose Utilities

Objectives

At the end of this session, you will be able to use the following commands

- **more command to get halted output**
- **The file command to know the file type**
- **wc to count lines, words & characters in the file**
- **Compare two files (cmp, diff & comm)**
- **find command to search for a file (find)**
- **lp command to Print a file**
- **banner command to display a blown up message**
- **cal command to display the calendar**
- **date command to display the system date**
- **tty to know your terminal**
- **who command to list the users who are working**

Halted Output - The more command

The more command allows the user to view a file, on screen at a time.The syntax of the more command is as follows :

more <options> <+linenumber> <+/pattern> <filename(s)>

$ more chap01
Normally, the message "…More…", along with the percentage of the file that has been viewed, is displayed at the bottom of the screen. You can advance to the next page by pressing the space bar (or <ENTER> in some cases). To quit more program simply press q(quit). more can also be used with multiple file names.

FILE TYPES – THE file COMMAND

The file command is used to determine the type of file, especially an ordinary file. This command has a built in mechanism of identifying the type of file by context.

$file emp.lst
emp.lst:ascii text

If the command is applied to a directory a informative list of all files in that directory will be displayed.

```
$ file progs/*
abcd:          ascii text
calender:      English text
$_
```

LINE, WORD AND CHARACTER COUNTING –THE wc COMMAND

The wc command counts lines, words and characters, depending on the options used. It takes one or more filenames as its arguments and displays a four-columnar output.

```
$wc infile
3       20      103 infile
```

wc counts 3 lines, 20 words and 103 characters.

Note:

When wc is used with multiple filenames, it produces a line for each file, as well as a total count. The following command sequence makes a count of the first three chapters:

```
$ wc chap01 chap02 chap03
305     4058    23177
550     4553 28667
377     3445    35444
```

DISPLAYING A FILE'S CONTENTS – THE od COMMAND

The od (octal dump) command displays the ASCII octal value of any file's contents. Each character is replaced by its octal value. The file odfile contains some of these characters which don't show up normally:

```
$od –b odfile
```

When used with –b option, each line displays sixteen bytes of data in ASCII octal format preceded by the position in the file of the first byte in the line, also in octal format.

COMPARING TWO FILES –THE cmp COMMAND

The cmp command is used to determine whether two files are identical in all respects.

```
$ cmp <filename1> <filename2>
$ cmp chap01 chap02
chap01 chap02 differ:char 9, line 1
$_
```

The two files are compared byte by byte, and the location of the first mismatch is echoed to the screen. If two files are identical, then the cmp displays no message, but simply returns $ prompt. The –l (list) option gives a detailed list of the byte number and the different bytes in octal for each character that differs in both the files.

comm COMMAND

The comm command compares two sorted files (in ASCII collating sequence), and compares each line of the first file with its corresponding line in the second. It displays a three-columnar output. The first column contains lines unique to the first file, while the second column shows the lines unique to the second file. The third column displays lines common (hence its name) to both files.

```
$cat file1
c.k.shukla
chanchal singhiv
s.n.dasgupta
sumit chakrobarty
$_

$cat file2
barun sengupta
c.k.shukla
anil aggarwal
lalit chowdhury
s.n.dasgupta
$_

$comm file1 file2
        barun sengupta
                c.k.shukla
        anil aggarwal
chanchal singhvi
        lalit chowdhury
                s.n.dasgupta
sumit chakrobarty
$_
```

FILE DIFFERENCES WITH diff

The diff command is used to display file differences and the lines in one file that has to be changed to make the two files identical.

```
$diff file1 file2
0a1
>arun sengupta
2c3,4
>chanchal singhvi
- - -
>anil aggarwal
>lalit chowdhury
4d5
<sumit chakrobarty
```

$_

PRINTING A FILE – THE lp COMMAND

UNIX provides a spooling facility by which you can queue up jobs for printing. This is done by lp command. It accepts more than one filename as arguments, and has a couple of useful options, which you should know. The following command prints a single copy of the first chapter:

```
$ lp chap01
Request id is 320
$_
```

The lp command notifies the job number, which you can later access with other commands. The file is not actually printed at the time the command is invoked, but later, depending on the number of the jobs already lined up in the queue. Several users can print their files in this way without conflict. There is a daemon (a process which runs periodically) which monitors this queue and prints each job in turn. The output from the printer will be a hard copy of the actual file, preceded by a title page mentioning the user name, request-id and date.

The –t (title) option, followed by a string of characters, prints the title on the first page:

```
$ lp –t "First Chapter"  chap01
Request id is 322
$_
```

The title "First Chapter" and the user name are printed in large characters, as well as in normal size. This can be suppressed by the –P option.

When used with the –m (mail) option, the user is notified by mail after the file has been printed.

The -c (copy) option prints only a copy of the file so that changes can be made in the original file even after it has been submitted to the spooler. The changes won't be reflected in the output.

You can print more than one copy with the -n (number) option. The following sequence prints three copies of the file chap01:

```
$lp –n3 chap01
Request id is 324
```

The lpstat command is used to show the status of all jobs submitted for spooling. The cancel command followed by the job number is used to cancel the specified jobs.

THE banner COMMAND

The banner command creates posters by blowing up its argument on the screen. On each line it can display a maximum of ten characters. If you have to display the word UNIX on the screen.

$banner UNIX

```
#      #      #      #      ###    #      #
#      #      # #    #       #       #   #
#      #      #  #   #       #         #
#      #      #    # #       #       #   #
######        #      #     ###     #      #
```

THE cal COMMAND

The cal (calendar) command is quite useful in printing the calendar of any particular month or the entire year. Any calendar from the year 1 to 9999 can be displayed with this command:

$ cal 1991

DISPLAYING THE SYTEM DATE with date

The UNIX system maintains an internal clock, which is meant to run perpetually. This is possible because, when the system is shut down, a battery backup keeps the clock ticking. This clock actually stores the number of seconds elapsed since January 1, 1970. The current date can be displayed by using the date command, which shows the date and time to the nearest second:

$date
Fri Dec 7 16:00:21 EST 1990
$_

The command can also be used with format specifiers as arguments.
Each format is preceded by a + symbol, followed by the % operator, and a single character describing the format.

$date +%m
12
$_

$ date +%h
Dec
$_

$date +"%h %m"
Dec 12
$_

THE who COMMAND

This command is used to display all the current users of the system. This helps to know the people working on the various terminals so that you can send them messages directly.

```
$ who
root        console     Jan 30 10:32
priya       tty01       Jan 30 14:32
uma         tty02       Jan 30 14:15
venkat tty05        Jan 30 13:17

$who –H
NAMELINE        TIME
root        console     Jan 30 10:32
priya       tty01       Jan 30 14:32
uma         tty02       Jan 30 14:15
venkat tty05        Jan 30 13:17

who -Hu
NAMELINE        TIME                IDLE    PID  COMMENTS
root        console     Jan 30 10:32        .       30
priya       tty01       Jan 30 14:32        0:40    31
$ who am i
root        console     Jan 30 10:32
```

KNOWING YOUR TERMINAL – THE tty COMMAND

The tty command is used to display the device name of the terminal which is currently in use. The command needs no argument.

```
$tty
/dev/tty01
$_
```

The output displays the complete pathname.

Locating Files with find

find recursively examines a list of files in a sub-directory to look for a file attribute. It uses a complex expression to select a file and specify an action for its disposal. It is always possible to examine a complete directory subtree rather than each sub-directory because find search is always recursive.

$ find <path list> <selection criteria> <action>

find first looks at all files in the directories specified in path list. The path list consists of one or more sub-directories, which need recursive examination. It then matches each file for one or more selection criteria. Finally it takes some action on those files that are selected

e.g

```
# find / -name .profile –print
/usr/kumar/.profile
/usr/tiwary/.profile
/usr/sharma/.profile
/.profile
#_
```

The path list (/), forming the first section of this command, here indicates that the search should start from the root directory.

find is characterized by the use of expressions which follow the path list. Every expression consists of an operator and an argument.

The first operator used here is –name, with the argument .profile.

This expression is applied to each file in the directory tree. If the expression matches the file (i.e the file has the name .profile), then the file is selected.

The third section specifies the action that is to be taken on the file, in this case a simple display on the terminal (-print).

The table below shows the major expressions used by find

Expression	**Action**
-name fname	Selects file fname
-links +x	Selects the file if it has more than x links
-user uname	Selects the file if it belongs to user uname
-group gname	Selects the file if it belongs to the group gname
-size +x[c]	Selects the file if the size of the file is greater than x blocks (characters if c is also specified)
-atime +x	Selects the file if it has been accessed in more than x days
-mtime +x	Selects the file if it has been modified in more than x days
-exec cmd	Executes UNIX command named cmd with selected files followed by {} \;
-ok cmd	Like –exec except that command is executed after user confirmation
-print	Prints selected file on the standard output
-cpio dvc	Copies selected file in cpio format to device dvc

e.g
#find . –name “*.lst” –print

locates all the files with the extension .lst
#find . –mtime –2 –print
selects all the files,modified in less than 2 days
#find . –atime +365 –print
Selects the file accessed more than a year ago
#find / -user kumar –print
selects the files owned by the user kumar
#find / -size +2000 –print | mail root
mails the files which are larger than 2000 blocks
#find / \(-user kumar –a –size +2000 \) –print
#find /usr/* \(-mtime +365 –a –atime +365 \) –exec rm {} \;
deletes all the selected files, but no prompt to the user.
#find /usr/* \(-mtime +365 –a –atime +365 \) –ok rm {} \;
deletes all the files accessed or modified more than a year ago, after prompting the user
#find –mtime –1 –cpio /dev/dsk/f0q15dt
backs up all the selected files to the floppy device in cpio format.

Summary

Unix system includes a number of handy utilities, like **more** which is a universal pager to display the file pagewise, comparison utilities (**comm, diff, cmp**) to compare the files, **find** to search for a particular file and perform an action on it. **date** to display the system date, **lp** to print a file & **who** find the users who are currently working.

Redirection, Pipes & Filters

Objectives

At the end of this session, you will be able to

- **Do i/p, o/p & error redirection**
- **Paginate files using pr command**
- **Display the beginning of a file using head**
- **Display the end of a file using tail**
- **Slit the file vertically using cut**
- **Paste the files using paste**
- **Order a file contents using sort**
- **List the Unique lines from a file using uniq and**
- **Number the file using nl**

Input-Output Redirection

Most of the commands in Unix produce the output on the terminal. Some, like editor, also take their input from the terminal. It is nearly universal that the terminal can be replaced by a file for either or both of input and output.
For example,

$ls
makes a list of filenames on your terminal. But if you say

$ls > filelist

that same list of filenames will be placed in the file filelist instead.
The symbol "**>**" means "put the output in the following file rather than on the terminal". This file will be created if it doesn't exist already or the previous contents will be overwritten. If it does, nothing is produced on your terminal.

$ cat f1 f2 f3 >temp
Combines all the files into one file temp.
$cat f1 f2 f3 >>temp
Combines all the files and appends it to the file temp.

$ mail mary kumar sharma < let

The symbol < (input redirection) means to take the input for the mail program from the file let instead of taking it from the terminal.

$ who > temp
$ sort < temp
$grep mary < temp

Pipes

All the previous examples reply on the same trick. Putting the output of one program into the input of another via a temporary file. But the temporary file has no other purpose. Instead it is clumsy to use such a file. This observation leads to one of the fundamental contributions of the Unix system, the idea of a pipe. A pipe is a way to connect the output of one program to the input of another program without any temporary file; a pipeline is a connection of two or more programs through pipes.

$ who | sort	print sorted list of users
$who | wc –l	Count users
$ ls | wc –l	Count files
$ls | pr –3	3-column list of filenames
$ who | grep mary	Look for particular user
$ls | pr –3 | lp	Creates and prints the 3-column list of files

Any program that reads from the terminal can read from the pipe instead; any program that writes on the terminal can write to a pipe.

Paginating Files – The pr Command

The pr command prepares a file for printing by adding suitable headers, footers and formatted text. It has many options also.

$ pr dep.lst

May 11 10:17 1990 dep.lstpage 1

01|accounts |6213
02|admin |5423
03|marketing |6521
04|personnel |2365
05|production |9876
06|sales |1006
<.. blank lines..>
$_

$pr dep.lst > prdep.lst
$ pr dep.lst | lp
Request id is 334
$ pr < dep.lst
$ pr –l72 chap01
-l option sets the page to 72 lines, of which 10 lines (5+5) will be used by the top and bottom margins, leaving 62 for the actual contents of the file.
$ pr –w132 chap01
Sets the width of the page to 132 characters.

$pr –h "Code List" dep.lst

May 11 10:17 1990 Code List page 1

01|accounts |6213
02|admin |5423
03|marketing |6521
04|personnel |2365
05|production |9876
06|sales |1006
<.. blank lines..>

-h option is used to set the header for the output.

$ pr –o20 dep.lst

May 11 10:17 1990 dep.lstpage 1

01|accounts |6213
02|admin |5423
03|marketing |6521
04|personnel |2365
05|production |9876
06|sales |1006
<.. blank lines..>

-o option specifies the offset to the output by a specified number of spaces.

$pr –3 dep.lst

This will produce the output in 3 column format

May 11 10:17 1990 dep.lstpage 1

```
01|accounts    |6213   03|marketing|6521        05|production  |9876
02|admin       |5423   04|personnel|2365        06|sales       |1006
<.. blank lines..>
```

$ pr –m dep.lst desig.lst

```
May  11  10:17  1990          page  1


01|accounts    |6213                01|chairman
02|admin       |5423                02|d.g.m
03|marketing   |6521                03|director
04|personnel   |2365                04|executive
05|production  |9876                05|g.m.
06|sales       |1006                06|manager
<.. blank lines..>
```

$ pr –m –s"%" dep.lst desig.lst

```
01|accounts    |6213%01|chairman
02|admin       |5423%02|d.g.m
03|marketing   |6521%03|director
04|personnel   |2365%04|executive
05|production  |9876%05|g.m.
06|sales       |1006%06|manager
<.. blank lines..>
```

-s is used to specify the separator while merging two files

$ pr +10 chap01
Starts printing from 10th page of the file.

$pr –d dep.lst
-d option double spaces the output

$ pr –t dep.lst
Totally omits the header and footer.

$pr –n dep.lst

Adds line numbers to the output.

Displaying the Beginning of a File – The head Command

The head command, as the name implies, displays the top of the file. Without any options, by default, it displays the first 10 lines of the file.

```
$ head emp.lst
$ head –3 emp.lst
```

This will display the first 3 lines of the file

```
$ head –2 emp.lst dep.lst
==>emp.lst <==
2233|a.k.shukla          |g.m.        |sales        |12/12/52|6000
9876|jai sharma          |director    |production   |12/03/50|7000


==>dep.lst <==
01|accounts    |6213
02|admin       |5423
$_
```

The above command displays the first 2 lines from emp.lst and dep.lst files

Displaying the End of the File – The tail Command

Complementing its **head** counterpart, the **tail** command displays the end of the file. If no count is specified, tails by default displays the last 10 lines of the file.

```
$tail –3 emp.lst
```
Will display the last 3 lines of the file emp.lst

```
$tail +11 emp.lst
```
Will display the file contents starting from line number 11.

```
$ tail –14c emp.lst
31/12/40|9000
```
Displays the last 14 characters of the file emp.lst

```
$tail –1b emp.lst | wc –c
512
```
-b displays the specified number of blocks from the file.

Slitting A File Vertically – The cut Command

While the head and tail are used to slice a file horizontally, you can slice a file vertically using the cut command. cut identifies both columns and fields and is a useful filter for the programmer.

cut <options> <character or field list> <file(s)>

cut can be used to extract specific columns from a file. If the name starts from column number 6 and goes up to the column number 22, and designation data occupies columns 24 through 32, use the following command to extract the names and desig.

$cut –c6-22,24-32 emp.lst
$cut –c-3,6-22,28-34,55- emp.lst
$cut –d"|" –f2,3 emp.lst

-d specifies the delimiter here &
-f specifies the field to be extracted.

Pasting Files –The paste Command

The paste command is used to paste the two files vertically, where as cut does it horizontally.

$ paste custlist01 custlist 02
$paste –d"|" custlist01 custlist02
-d is used to specify the delimiter for joining the files

Ordering A File – The sort Command

The sort command is used to order the contents of a file in a particular order. Sort performs it usual functions, but has a couple of unique features. Unix sort is different from other sort utilities in the sense that lines or records need not have a fixed length for it to work successfully.

$ head –3 emp.lst > shortlist
$ sort shorlist
2233|a.k. shukla |g.m. |sales |12/12/52|6000
2365|barun sengupta |director |personnel |11/05/47|7800
5423|n.k.gupta |chairman |admin |30/08/56|5400
$

Sorting starts with the first character of each line and proceeds to the next character only when the characters in two lines are identical. By default, sort reorders a line in ASCII collating sequence, starting from the beginning of the line. This sequence accords priority in the following order- white space (spaces and tabs), numerals, uppercase letters and finally lowercase characters.

Like cut and paste, sort also works on fields, and the default field separator is the space character. The –t option, followed immediately by the delimiter, overrides the default. This lets you sort on any field.

$sort –t"|" +1 shortlist
The argument +1 indicates that sorting should start after skipping the first field.

To sort on the third field, you should give

$sort –t"|" +2 shortlist

$ sort –r –t"|" +1 shortlist

Will do reverse order sorting.

$ sort –o sortedlist +3 shortlist
This will sort the file on the fourth field and place the output in the file sortedlist.
$sort –o shortlist shortlist
sort is the only Unix command where the input and output filenames can be the same.

$ sort –c shortlist
Checks whether the file has actually been sorted.

To check whether the file has been sorted on the 4^{th} field or not, you can use the following command,

$sort –t"|" +3c shortlist
sort:disorder:2365| barun sengupta |director |personnel |11/05/47|7800

Indicates that the file has not been sorted on the specified field.

Sorting On a Secondary Key

Sorting can be done on more than one field. If the primary key is the third field and the secondary key is the second field, then you can use,

$ sort –t"|" +2 –3 +1 shortlist
5423|n.k.gupta |chairman |admin |30/08/56|5400
2365|barun sengupta |director |personnel |11/05/47|7800
2233|a.k. shukla |g.m. |sales |12/12/52|6000

This sorts the file by designation and name. –3 indicates stopping of sorting after the third field and +1 indicates resumption of sort after the first field. To resume starting from the first field say +0.

Sorting On Columns

You can also specify a character position within a field to be the beginning of sort. Suppose, if you want to sort the file according to the year of birth, then you need to sort on the seventh and eighth column positions within the fifth field.

```
$ sort –t"|" +4.6 –4.9 shortlist
2365|barun sengupta   |director        |personnel       |11/05/47|7800
2233|a.k. shukla              |g.m.            |sales           |12/12/52|6000
5423|n.k.gupta         |chairman        |admin |30/08/56|5400
```

Numeric Sort

When sort acts on numerals, strange things can happen. Consider the contents of this file

```
$cat numfile
2
4
10
27
$
```

When you sort this file, you get a curious result :

```
$ sort numfile
10
2
27
4
$_
```

This is probably not what you expected, but the ASCII collating sequence places 1 above 2 and 2 above 4. That's why 10 preceeded 4. This can be overridden by the –n option.

```
$sort –n numfile
2
4
10
27
$_
```

The other options used with sort are,

-b	Ignores leading blank spaces
-d	Alphanumeric sort (dictionary sort)
-f	Folds uppercase into lowercase

The uniq Command

unique is a special filer command to handle duplicate lines in a file. uniq simply fetches one copy of the redundant records, writing them to the standard output.

```
$cat dept.lst
01|accounts   |6213
01|accounts   |6213
02|admin      |5423
03|marketing  |6521
03|marketing  |6521
03|marketing  |6521
04|personnel  |2365
05|production |9876
06|sales      |1006

$ uniq dept.lst
01|accounts   |6213
02|admin      |5423
03|marketing  |6521
04|personnel  |2365
05|production |9876
06|sales      |1006
```

uniq always requires a sorted file as input. The general procedure is to sort a file and pipe the process to uniq.

```
$ uniq dept.lst output.lst
```
Stores the unique output in the file output.lst

```
$uniq –u dep.lst
02|admin      |5423
04|personnel  |2365
05|production |9876
06|sales      |1006
```
Selects only the non-repeated lines.

```
$cut –d dep.lst
```
Selects only one copy of the repeated lines

```
$cut –c dep.lst
```
Displays the frequency of occurrence of all lines, along with the lines.

Line Numbering – The nl Command

There is a separate command in the Unix system called **1**, which has elaborate schemes for numbering lines.

```
$nl desigx.lst
1 chairman
2 d.g.m.
3 director
4 executive
5 g.m.
6 manager

$ nl –w2 –s"|" desigx.lst
1|chairman
2|d.g.m.
3|director
4|executive
5|g.m.
6|manager
$_
```

-w option is used to specify the width of the number format, and –s is to specify the separator.

```
$nl –w2 –s"|" –nrz desigx.lst
01|chairman
02|d.g.m.
03|director
04|executive
05|g.m.
06|manager
```

-n prefixes leading zeros, rz right justifies the number.

```
$ nl –w2 –s"|" –nrz –v30 –i5 desigx.lst
30|chairman
35|d.g.m.
40|director
45|executive
50|g.m.
55|manager
$
```

-v option is used to specify the initial value to number the lines and –i is used to specify the increment value.

Summary

Unix supports i/p redirection where input is taken from a file instead of the keyboard and the o/p is redirected to a file instead of the terminal. Also it supports pipes where the output of one command is transferred as input to another command. There are also special filter commands supported in Unix, which take the i/p, perform some action on it and produce the o/p

Communication & Scheduling

At the end of this session you will be able to

View the bulletin board using the news command

- **Do two-way communication using write**
- **Insulate yourself from others using mesg**
- **Communicate with other users by sending & receiving mails**
- **Address all the users using the wall command**
- **Send a message to self using the calendar command &**

- **Set delay in shell scripts using the sleep command &**
- **Execute later using at and batch commands**

In a multi-user system, it is often necessary for one user to know what the other is doing. When there are a hundred odd users sharing the system resources, some of them are invariably, located quite a distance apart. Communication through the system seems quite natural and necessary. The system administrator also requires to send messages to some, and sometimes to all of them.

The Bulletin Board – The news command

Any user to read any message that is sent by the system administrator normally invokes the news command. The administrator does that by storing the contents of the message in a file in the directory /usr/news. The typical news item is stored in the file dinner in that directory.

```
$ cat /usr/news/dinner
The Chairman invites you all to a dinner
at the Grand Hotel at 8 pm.
$_
```

Every user is expected to invoke the command news immediately after he logs in.

```
$ news
dinner (root) Fri Mar 8 20:02:10   1991
```

The Chairman invites you all to a dinner
at the Grand Hotel at 8 pm
$_

$**news**
No news # All the news items are read
$**news –n** # displays only the filenames of those messages
news : holiday meeting farewell
$**news meeting** # To display the specific message alone
$ **news –s** # To display the no. of items which have still not been read.
$ **news –a** # displays all news , regardless of already read or not

Every news item is kept as a separate file in /usr/news.
news works by noting the modification time of the file **.news_time** in the user's login directory. This is essentially a zero length file whose modification time gets updated as and when news is invoked. If the file doesn't exist, then the first invocation of the command creates it.

Message of the Day – The motd Facility

Another facility provided by the UNIX system is similar to **news,** but is normally seen as soon as the user logs in. It is called motd (message of the day). It is merely a text file /etc/motd, and not a command. The file is maintained by the system administrator, who puts any message into it that he thinks important enough to reach every user without his invoking any command specifically for the purpose. The command

cat /etc/motd

is generally kept in the file /etc/profile and is therefore, seen by every user as soon as he logs in.(This profile is executed first, before the user's own).

The two-way Communication – The write Command

The write command lets you have a two-way communication with any person who is currently logged in. One user writes his message and then waits for the reply from the other. In this way, it is possible to continue a conversation until such time as one or both the users decide to terminate it.

This is how kumar starts a dialogue with sharma

$ **write sharma**
Have you completed your program ?

I have completed mine – kumar
o
<ctrl-d>
$_

The message appears on sharma's terminal, provided he is logged in. If he is not, then the system responds with an error message:

sharma is not logged in

However, if sharma is logged in, then he will see the following message on his display:

Message from kumar (tty01) [Sat Nov 3 17:21:56]

Have you completed your program?
I have completed mine – kumar
o

Sharma now has the option of replying to this message. If he decides to do that, then he will have to invoke the write command as well, and send his message. His message then appears on kumar's terminal:

$ **write kumar**
Another hour, and it will be ready !
0-0
<control-d>
$_

When write is used interactively in this manner, both users need to invoke the command individually. This establishes two "write" channels, one for the receiver, and the other for the sender. However, if there is need for an extended communication, then one user can simply wait after he has keyed in his message, without pressing <ctrl-d>. write is so designed that it enables the user to send and receive message at the same time, without his having to quit to the shell. There will then be a series of sent and received messages appearing sequentiallly on each terminal.

Once into the write program, you can execute any UNIX command in the same way as you do with ed/ex, and with the same operator (!).

!ls *.txt

When sharma is logged in to more than one terminal, say on tty03 and tty08, then the command

$ **write sharma**

Sends the message to the terminal with the lowest number, i.e tty03, But if you really want to send the message to the terminal tty08, you can also do that. write also accepts the terminal name as the second argument.

$ **write sharma tty08**

Insulation from Other users – The mesg Command

Communication, single or two-way, can be disconcerting to a user who might be watching the output of a very important program on his terminal at that instant. He obviously wouldn't like the screen to be disturbed by such unexpected intrusions. In that case he can use the mesg command to insulate himself. The command

$ **mesg n**
prevents other people from writing to his terminal. while

$ **mesg y**
enables receipt of such messages. By default, the terminal behaves as if it is in this mode.

If you want to know the status of your terminal is in, then simply use mesg without arguments.
$ **mesg**
is y

Using the mailbox – The mail command

The mail command is one of the most popular in the UNIX repertoire of electronic mail. Unlike write, it enables sending of mail to a user even if he is not logged in. It is mainly used for non-interactive communication, especially when there is no urgency.

Like write, mail uses standard input, and a primitive way of using this program is to take the input from the keyboard.

$ **mail sharma**
The new system will start functioning from next month
Convert your files by next-week –kumar
<ctrl-d>
$_

mail is essentially a single-channel communication command, and the sender doesn't use it for conversation. The message doesn't appear on the receiver's terminal either. If the receiver is running a program, then mail waits for program execution to finish before flashing the following message:

You have mail

mail differs from write in another way; the receiver needn't be logged in for the message to reach him. mail operates on a different principle altogether. It saves th e message in a "mailbox" which normally is placed in the directory /usr/mail, and has the same as the login name. In this case, sharma's mail is appended to the file /usr/mail/sharma. When sharma logs in, he will first be greeted by the above message. The advantage of mail is you needn't handle it immediately.

```
$ mail          #To read the incoming mails

From kumar Thu sep 13 16:25:46 EST 1990
The new system will start functioning from next month
Convert your files by next-week –kumar
?
```

Once mail responds with a ?, you can see the next message, if there is one , by using the concept of relative addressing that you saw in ex. Enter a + at the prompt to see further mail, and a – to display the immediately preceding message. Any message can be accessed by simply entering the number itself. For instance,

```
?3
```
shows you the message numbered 3.

You can decide to save any particular message in a separate file rather than the default mailbox used by the system. The w (write) command (with a filename as argument) is used to do that.

```
?w note3      # saves the current message in the file note3
? w 1 2 3 note3  # Append the messages to the file note3
```

The following table shows the list of commands used with the mail command

Command	**Action**
+	prints the next message
-	Prints the previous message
N	prints the message numbered N
h	prints the headers of all the messages
d N	deletes the message N (otherwise the current msg)
u N	undeletes the message N (the current if not specified)
s fname	saves the current msg in the file fname with headers
w fname	saves the current msg without headers
m usr-list	forwards mail to users in the list
r usr-list	replies to sender as well as the users in the usr-list
q	quit from mail program
p	prints mail
! <cmd>	Runs UNIX command <cmd>

Addressing All users – The wall Command

The wall command has more urgency than the others, as it addresses all users. Though the command is available in /etc, it can be invoked by any user by employing the absolute path of the command. There are no arguments, and the standard input is used for text input

$ /etc/wall
The machine will be shutdown today
at 14:30 hrs. The backup will be at 13:30 hrs
<ctrl-d>
$_

Every user currently logged in will receive this message on his terminal, duly preceded by the same header line that accompanies the write command.

Sending a message to self – The calendar command

The calendar command provides a useful reminder mechanism for a user. It is a sort of an engagement diary, which consists of textual information. The command searches a text file named calendar in the current directory for lines containing any date, which either represents the current date, or tomorrow. It displays the matched lines on the standard output. For such mechanism to be effective, there should be a file named calendar in the current directory, which can look something like this

$ **cat calendar**

Nov 1, 1990 the target for this month is Rs 3.2 crore
Board meeting on second November, 1990 at 10 a.m.
Half-yearly results should be published by Nov 3
Lunch with the Chairman on November 6
$_

$date
Thu Nov 1 12:03:18 EST 1990

$calendar
Nov 1, 1990 the target for this month is Rs 3.2 crore
Board meeting on second November, 1990 at 10 a.m.
$_

Delay in Shell Scripts – The sleep Command

When you use shell scripts, you may occasionally need to introduce some delay in certain sections of the script. Sometimes, it is necessary to let the user see some

message on the screen for a predefined period before the script starts doing something else. sleep is a command which does it.

```
$ sleep 100; echo "100 seconds have elapsed"
100 seconds have elapsed
$_
```

The message appears exactly 100 seconds after the commands have been invoked. This is a simple form of scheduling commands so that they can be invoked now, to be executed later.

Execute later – The at and batch commands

UNIX provides more sophisticated facilities to schedule a job to be run at a specified time of day rather than now. This facility is also useful because the load on any UNIX system varies greatly throughout the day. It is thus preferable to schedule less urgent jobs at a time when the system overheads are low.

at takes as its argument the time the job is to be executed.

```
$ at  14:08
empawk2.sh
<ctrl-d>
job 657535140.a         at  Fri Nov   2  14:08:00 1990
$_
```

This means that at 2:08 p.m. today, the script file empawk2.sh has to be executed, which incidentally contains an awk program. The job number is also shown, along with the date and time at which the job is meant to be executed.

```
$ at 1508 < empawk2.sh        # takes the i/p from the file (time can be
                                  specified without a : )
$ at 15:08
empawk2.sh > rep.lst
<ctrl-d>
job 657538680.a         at Fri  Nov  2   15:08:00         1990
$_
```

The standard output of the command can also be redirected to a file. However, any error messages that may be generated while executing a program will, in the absence of redirection, continue to be mailed to the user.

```
$ at noon
$ at now + 1 year
$ at 3:08pm + 1 day
$ at 15:08 December 18, 1991
$ at 3:08pm Dec 18, 1991 + 4 years
```

```
$ at 9am Mon
$ at 9am tomorrow
```

The at Queue

The at queue is displayed with the –l option. This shows, for each submitted job, its number (derived from the number of seconds elapsed since 1970), the scheduled date of execution, as well as the day of the week.

```
$ at –l
657613800.a          Sat    Nov    3  12:00:00    1900
689075220.a          Sat    Nov    3  15:17:00    1990
$_
```

This listing shows only the jobs scheduled by you not by others.

```
$ at –r 657613800.a          # Removes the job from the queue
```

The batch command too schedules jobs for later execution, but unlike at, scheduling is done in such a way that the jobs are executed as soon as the system load permits. The command doesn't take any arguments, and uses an internal algorithm to decide when the job will be executed. This is a useful command since it prevents too many CPU-hungry jobs from running at the same-time, as they can bring the machine to a complete standstill.

```
$ batch < empawk2.sh
job 657540282.b at       Fri    Nov    2  15:34:42    1990
$_
```

Any job scheduled with batch also goes to the at queue, and can also be removed with the command at –r, provided, of course, the at command is invoked before the job has been executed.

The access to the use of at and batch are restricted in many UNIX systems. This security feature is controlled by the files at.allow and at.deny. Both are expected to be present in /usr/lib/cron, though it is not necessary to have these files.

The at.allow will contain the list of user names who are permitted to use the at and batch commands. The at.allow will be checked first to see the user list. If not present, the system checks for at.deny. If present, the users listed there are barred from using these commands. If neither is present, only administrator is allowed to invoke these commands.

Summary

Unix allows its users to communicate with each other interactively using the write command. It also has the mail command using which, a user can send a mail to another user even if he has not logged on. It also has commands to schedule the jobs that are to be executed later, to send the reminder messages to the users and to send messages to all the users at the same time.

PROCESSES

Objectives

At the end of this session, you will be able to

- **Learn about the mechanism of process creation in UNIX**
- **Print the process status**
- **Run a job in background**
- **Wait for the completion of a background process**
- **Set the priority for the processes and terminate the processes**

Overview of Processes

A process is simply an instance of a running program. When any program is executed, it gives rise to a process. The process is said to be born when the program starts execution, and remains alive as long as the program is active. After the execution of the program is complete, the process is said to die.

This process also has a name, usually the name of the program being executed. For example, when you execute the grep command, a process named grep is created.

Since UNIX is a multi-tasking system, more than one process can run at a time. Typically, hundreds or even thousands of processes can run in a larger system. A number called the PID (the process identifier) which is allotted by the kernel when it is born uniquely identifies each process. This is a number between 0 and 32,767.

But in any UNIX system with a single CPU, there is actually only one process, which can run at a time. The kernel is responsible for the management of these processes. It determines among other things, the time that is allotted to the processes, their priorities and so on.

Therefore, in these time-sharing systems, it is the job of the kernel to ensure that multiple processes are able to share the CPU resources. It provides a mechanism by which a process is able to execute for a finite period of time and then relinquish control to another process. The kernel has to frequently store these active processes in the disk before calling them for running. This happens more than once a second, making the user transparent to the switching process.

The sh Process

When a user logs into the system, the kernel immediately sets up a process called **sh**. This process or the program remains alive until the user logs off the system, and any command that you type in at the prompt is actually the standard input to this program.

The shell maintains a set of variables, which are available to the user;
You can use the special shell variable "$$" to know the PID number for the shell process.

```
$ echo $$
30
$_
```

Parents And Children

Just as a file has a parent, every process also has its own creator, which itself is another process. The process responsible for giving birth to another is called the parent, and the born process created thus is said to be its child.

When you run a command from the command line, the shell interprets it, and then executes the corresponding program. Another process is started by the shell, which remains active as long as the command is active.

e.g
$ cat emp.lst

Sets up a process, which is identified by the name cat. This process is initiated by the sh process and the latter is said to be the parent of the cat process. The cat is said to be the child of sh. Since every process has a parent, you can't have an "Orphaned" process in the UNIX system. The ancestry of every process can be traced to one ultimate process, which happens to be the first that is set up when the system is booted (PID 0) This process doesn't have an ancestor. Just as a directory can have more than one file under it, a process can also generate (spawn) one or more processes.

The command

$ cat emp.lst | grep 'director'

Sets up two processes for the commands. The two processes have the names cat and grep and are set to be spawned by sh. However, a process can have only one parent. And after giving birth to a child, it waits for its death.

When a process is completed, it sends a signal to its parent informing it of its death. Control is thus reverted back to the parent process, which can give birth to other processes and wait for them to die as well.

Because, UNIX is a multi tasking Os, it is possible for a process to spawn more than one child. However, if you can arrange to have the parent die, then the children of the parent process will also automatically die which is the default behavior of the processes. But this can also be altered.

Process Status – The ps command

To know the status of all the processes that are currently running in your current login session, you can use the ps command.

```
$ ps
PID    TTY    TIME        COMMAND
30     01     0:03        sh
56     01     0:00        ps
$_
```

where

PID	is the process Id
TTY	the terminal with which the process is associated
TIME	the cumulative processor time that has been consumed since the Process has been started
COMMAND	The process name

To get the information about the ancestry of a process, use –f option

```
$ ps –f
UID    PID    PPID   C      STIME      TTY   TIME        COMMAND
Kumar  30       1    0      11:18:08   01    0:03           -sh
Kumar  61      30   13      11:24:01   01    0:00        ps –f
```

Where

UID	is the user's login name
PID	is the process id
PPID	is the PID of the parent process
C	is the amount of CPU time consumed by the process
STIME	chronological time that has elapsed since the process started

To get the activities of any user at a time, use –u option

```
$ ps –u sharma
```

Use –a option to list out the processes of all the users

```
$ ps –a
```

Use the option –l to get the detailed listing of attributes of each process.

```
$ ps –l
```

Multiple Jobs in Background - & and the nohup command

A multi-tasking system lets a user do more than one job at a time. The & operator is provided by the shell to run a process, and not wait for its death. After the command line is terminated with an &, the command will run in the background.

```
$ sort –o emp.dat emp.lst
71
$_
```

On invoking this command, shell immediately returns a number which is the PID of background process and comes back to the $ prompt, ready to process the next command. Execute the ps –f to see the status of the background process.

Continue Process

Background processes cease to run, however, when a user logs out. This happens because, when his shell dies. And when the parent dies, its children also die. The UNIX system permits a variation in this default behavior. It provides a nohup (no hang-up) command to permit execution of a process even after the user logged off.

```
$ nohup sort emp.lst &
101
Sending output to nohup.out
$_
```

The & operator must terminate the command line as usual, and the shell returns the PID number of the process. Since no output filename was specified with the sort command, nohup sends the default output to a file nohup.out. You can now log off the system and still be assured that the command has not been aborted. Use the ps command from another terminal to observe the effect of nohup.

```
$ps –f –u kumar
UID     PID   PPID  C     STIME        TTY     TIME     COMMAND
Kumar  101     1    45    14:52:09      01      0:13 sort emp.lst
$_
```

Here the PPID of the sort process has been assigned with the process id (1)
of the process init which is the parent of the shell

Waiting for completion of Background Processes - The wait command

Background execution becomes meaningful if more than one job has to be executed concurrently. It is pointless to execute a job in the background and then remain idle in the foreground.

But some applications may require you to run one final program, which uses some files created by background jobs. In such an event, you can execute this program only after all

the background jobs have completed execution. The wait command, by default, checks whether all background processes have been completed.

```
$temp=/tmp/kumar
$cut –c1-4 oldlist > $temp.01 &
121
$cut –c5-21 oldlist > $temp.02 &
122
$cut –c22-30 oldlist > $temp.03 &
123
$ cut –c31-40 oldlist > $temp.04 &
124
$ cut –c41-48 oldlist > $temp.05 &
125
$ cut –c49- oldlist > $temp.06 &
144
$_

$ wait
$paste –d"|" $temp.??
```

wait is a built-in shell command, not a executable program. When this command is executed no process is spawned. It sends the shell into a wait state so that it can acknowledge the death of all the children. It can also be used with an argument. Thus

```
$ wait 138
```

Waits for the background job bearing the PID number 138. It doesn't wait for the completion of other background jobs, which could be running at the same time.

Premature Termination of a Process – The kill Command

If you have a program running longer than you anticipated, or if you have changed your mind and want to run something else simply send a signal to the active process with a specific request of termination. Pressing the interrupt key of your machine does this. When a process receives the signal, it may ignore it, terminate it or do something else.

Signals in UNIX are identified by a number. They all have the associated name or the definition, the list of that are applicable to your machine are available in the file called /usr/include/sys/signal.h.

Each signal notifies to a process that an even has occurred. This event can be a hang-up (logging out), represented by the signal number 1, or the normal software termination indicated by the signal 15. The kernel uses the signal 18 to indicate the death of a child to the parent. Pressing the Interrupt key makes the kernel send a signal numbered 2, having the name SIGINT.

A Process can be terminated by using the kill command. The command uses one or more PID numbers as its arguments.

$kill 105
$kill 121 122 125 132 138 144 # kills more than one process

However, if one background process gives birth to other child processes, then you may kill only the parent process in order to kill all its children. But if **init** inherits the children of the killed process (as with the nohup command), you should then kill them separately by providing all these PID numbers as arguments in the command line.

kill, by default, uses the signal number 15 (SIGTERM) to terminate the process. Some programs simply ignore this signal and continue execution normally. These programs are designed in such a way that SIGTERM is ignored. In that case, the signal number 9 (SIGKILL) is required to kill any process. This is sometimes known as the sure kill signal. But this cant be generated at the press of a key.

$ kill –9 121 # Sure kill , generates the signal SIGKILL

$ kill 0 # kills all the processes you own, except the login shell

Job execution with Low priority – The nice Command

Processes in UNIX system are usually executed with equal priority. UNIX offers the nice command to reduce the priority of jobs. This is a helpful feature, especially when used with the & operator. It enables the low-priority but CPU –intensive jobs to be run with reduced priority so that more important job has greater access to the system resources. To run a job with a low priority, the command name should be prefixed with **nice**

$ nice wc –l uxmanual

or better still with

$ nice wc –l uxmanul &
131
$_
By default, nice reduces the priority of any process by 10 units, the default nice value of a process being around 20. The above command will have the effect of increasing this nice value by 10 units.

$ nice –15 wc –l uxmanual &

This reduces the priority by 15 units. This argument value can vary from 0 to 19 and in any case will reduce the priority of a process. The user can't increase the priority of a process. That power is reserved for the super user. The nice and priority values are displayed with the –l option of ps.

Summary

The process is an important entity of the Unix system. Since Unix is a multi-tasking system, it is possible to create multiple processes. The scheme for organizing these processes is well defined. Each process has a unique number allotted to it. The ps command is used to display the attributes of a process. Unix also permits you to control the processes by setting priorities.

It also allows you to run processes at the background without user intervention.

Advanced Filters – I

Objectives

At the end of this session, you will be able to

- **Search for a pattern in the file using the grep command**
- **Extend the functionality of grep using egrep**
- **Do Multiple pattern matching using fgrep and**
- **Translate characters using tr**

There is a large family of UNIX programs that read simple input, perform simple transformation on it, and write some output. Examples include grep, tr & sed.

Searching for a pattern – The grep command

A frequent requirement of the end user or the programmer is to look for a pattern in a file. grep is a such a filter command used for scanning a file for a regular expression.

grep <options> <pattern> <filename(s)>

Most of the options of grep are shared by its other members also (egrep and fgrep). The important options are listed below,

Option	Significance
-c	Displays the count of the no. of occurrences
-l	Displays the list of filenames which contain the pattern
-n	Displays the line numbers along with the lines
-v	Displays all but the lines matching the expression
-I	Ignores the case for matching
-f fname	Takes expressions from the file fname ,egrep&fgrep only
-x	Displays the line matched in entirety , fgrep only

You can use grep to display from the file emp.lst, the lines containing the string "sales".

```
$grep sales emp.lst
2233|a.k shukla          |g.m.      |sales      |12/12/52|6000
1006|chanchal singhvi    |director  |sales      |03/09/38|6700
1265|s.n. dasgupta       |manager   |sales      |12/09/63|5600
2476|anil aggarwal       |manager   |sales      |01/05/59|5000
```

$_

$grep president emp.lst
$_
The command failed because, the string president couldn't be located.

```
$grep director emp1.lst emp2.lst
emp1.lst:1006|chanchal singhvi     |director     |sales        |03/09/38|6700
emp1.lst:6521|lalit chowdury       |director     |marketing    |26/09/45|8200
emp2.lst:9876|jai sharma           |director     |production   |12/03/50|7000
emp2.lst:2365|barun sengupta       |director     |personnel    |11/05/47|7800
$_
```

```
$ cat emp?.lst | grep director          #omit the filenames
1006|chanchal singhvi     |director     |sales        |03/09/38|6700
6521|lalit chowdury       |director     |marketing    |26/09/45|8200
9876|jai sharma           |director     |production   |12/03/50|7000
2365|barun sengupta       |director     |personnel    |11/05/47|7800
$_
```

```
$ grep 'jai sharma' emp.lst
9876|jai sharma           |director     |marketing    |12/03/50|7000
$_
```

Here the string 'jai sharma' is included within quotes, otherwise 'jai' will be taken as the string to be searched in the file file 'sharma'

```
$str='jai sharma'
$ grep "$str" emp.lst
9876|jai sharma           |director     |marketing    |12/03/50|7000
$_
```

grep Options

```
$ grep –c 'director' emp?.lst
emp1.lst:2
emp2.lst:2
$_
```

-c option will list out the no. of lines matching the pattern

```
$grep –n director emp.lst
6:1006|chanchal singhvi     |director     |sales        |03/09/38|6700
11:6521|lalit chowdury      |director     |marketing    |26/09/45|8200
2:9876|jai sharma           |director     |production   |12/03/50|7000
```

```
4:2365|barun sengupta        |director     |personnel     |11/05/47|7800
$_

$ grep –v 'director' emp.lst | tee otherlist
2233|a.k.shukla            | g.m.        |sales         |12/12/52|6000
5678|sumit chakrobarty      |d.gm.        |marketing     |19/04/43|6000
1265|s.n. dasgupta          |manager      |sales         |12/09/63|5600
2476|anil aggarwal          |manager      |sales         |01/05/59|5000
2345|j.b. saxena            |g.m.         |marketing     |12/03/45|8000
0110|v.k.agarwal            |g.m.         |marketing     |31/12/40|9000
$_
```

-v option will list the lines which don't contain the pattern.

```
$grep –l 'manager' *.lst
desig.lst
emp.lst
emp1.lst
emp2.lst
empn.lst
$_
```

-l option will list only the filenames that contain the pattern

```
$ grep –I 'agarwal' emp.lst
3564|sudhir Agarwal          |executive    |personnel     |06/07/47|7500
$_
```

-i option ignores the case for pattern matching.

Regular Expressions

grep is not just used to search a file for simple strings. It is possible that you may be looking for a name, but don't know exactly how it is spelt. You may be interested in the occurrence of a pattern only at certain locations like beginning or end of the line.

grep also accepts regular expressions for a pattern to do all these.
A regular expression is a string of ordinary and meta characters which can be used to match more than one type of pattern. The following are the list of regular expressions used by grep.

Expression	**Significance**
ch*	Matches zero or more occurrence of the previous char **ch**.
[pqr]	Matches a single character p,q or r
[c1-c2]	Matches a single character with in the ASCII range represented by the characters c1 and c2
[^pqr]	Matches a single character which is not p,q or r
^pattern	Matches the pattern at the beginning of a line

pattern$	Matches the pattern at the end of the line
ch\{m\}	Matches m occurrences of the character ch
ch\{m,n\}	Matches a range of m to n occurrences of the char. ch
ch\{m,\}	Matches a minimum of m occurrences of ch
\(pattern\)	Matches the pattern enclosed by \(and \) with the tag \n where n is the integer between 1 and 9

While these characters are special to the command, and some of them to the shell, adequate care has to be taken to ensure that the shell doesn't interfere in affairs, which legitimately concern the command. That is why all regular expressions used by grep are enclosed within quotes.

Examples

```
$grep '[cC]ho[wu]dh*ury' emp.lst
4290|jayant Choudhury      |executive  |production  |07/09/50|6000
6521|lalit chowdury        |directory  |marketing   |26/09/45|8200
$_
```

We accounted for case by using the character class [cC]. The sequence h* indicates that there can be either no h, or as many h's at that position.

Regular expressions can be used to make more meaningful and complex searches. To select those lines where the salary is 8000 or more, search for a character which belongs to the class [8-10], followed by three more characters (represented by…). Anchor the pattern suitably to the end of the line with the $.

```
$ grep '[8-10]…$' emp.lst
6521|lalit chowdury        |director   |marketing   |26/09/45|8200
2345|j.b. saxena           |g.m.       |marketing   |12/03/45|8000
0110|v.k. agrawal          |g.m.       |marketing   |31/12/40|9000
```

To find out how many are directors in this list, use the following command.

```
$ grep 'director.*[8-10]…$' emp.lst
6521|lalit chowdury        |director   |marketing   |26/09/45|8200
$_

$grep –c 'director.*[8-10]…$' emp.lst
1
$_

$ grep –v '^[ \t]*$' abc
```

Deletes the lines containing white space from the file.

Extending grep – The egrep

There are certain limitations with grep, like, you can't search for more than one pattern. For example, you cannot search for sengupta and dasgupta together. But the egrep command, authored by Alfred Aho, extends the pattern-matching capabilities.

```
$ egrep 'sales| marketing' emp.lst > emp1.lst
$ egrep –v 'sales|marketing' emp.lst > emp2.lst
$_
```

The extended regular expression set used by egrep

Expression	Significance
ch+	Matches one or more occurrences of the char. ch
ch?	Matches zero or one occurrence of the char ch.
exp1 | exp2	Matches the expression exp1 or exp2
(x1|x2)x3	Matches the expression x1x3 or x1x2

```
$egrep 'sengupta|dasgupta' emp.lst
$egrep '(sen|das)gupta' emp.lst
```

Multiple String Searching – The fgrep

fgrep, like egrep, accepts alternative patterns, both from the command line, as well as from a file, but unlike grep and egrep, it doesn't accept regular expressions. So, if the pattern to search for is a simple string, or a group of them, then fgrep is recommended. It is faster than its two fellow members, and should be used while using fixed strings.

```
$ fgrep 'sales
> personnel
> admin' emp.lst
```

Will list out all the sales, personnel and admin employees.

egrep and fgrep can take the patterns from a file which are stored in separate lines.

```
$cat pat.lst
sales
personnel
admin
$
```

```
$fgrep –f pat.lst emp.lst
```

Translating Characters - The tr Command

tr is an important filter which maps characters. The command takes only the standard input as the source of its data; it doesn't take a filename as its argument. When invoked without options, it translates each character in expression1 to its mapped counterpart in expression2. The first character in the first expression is replaced by the first character in the second expression, and similarly for the other characters.

Syntax:

tr <options> <expression1> <expression2> <standard input>

Examples

```
$ tr '|' '~' < emp.lst
2233~a.k. shukla          ~g.m.        ~sales       ~12/12/52~6000
9876~jai sharma           ~director    ~production  ~12/03/50~7000
5678~sumit chakrobarty    ~d.g.m.      ~marketing   ~19/04/43~6000
2365~barun sengupta       ~director    ~personnel   ~11/05/47~7800
5423~n.k. gupta           ~chairman    ~admin       ~30/08/56~5400
1006~chanchal singhvi     ~director    ~sales       ~03/09/38~6700
6213~karuna ganguly       ~g.m         ~accounts    ~05/06/62~6300
1265~s.n. dasgupta        ~manager     ~sales       ~12/09/63~5600
4290~jayant Choudhury     ~executive   ~production  ~07/09/50~6000
2476~anil aggarwal        ~manager     ~sales       ~01/05/59~5000
6521~lalit chowdury       ~director    ~marketing   ~26/09/45~8200
3212~shyam saksena        ~d.g.m       ~accounts    ~12/12/55~6000
3564~sudhir Agarwal       ~executive   ~personnel   ~06/07/47~7500
2345~j.b. saxena          ~g.m.        ~marketing   ~12/03/45~8000
0110~v.k.agrawal          ~g.m.        ~marketing   ~31/12/40~9000
$
```

$head –3 emp.lst | tr '[a-z]' '[A-Z]'

Changes all the lowercase letters to uppercase.

$ tr –s ' ' < emp.lst

-s option squeezes the multiple consecutive occurrences of its argument to a single character. You can use it to compress multiple blank spaces to a single space.

Summary

This chapter highlights a very important aspect of the Unix system – the database features of the tool kit. grep is used to locate a pattern or report on the details of its occurrence. You can output the matching records, their line numbers, as well as the number of times a pattern occurs in a file. Unix also supports egrep and fgrep, which are used with multiple expressions.

Programming With The Shell

Objectives

At the end of this session you will be able to

- **Write interactive shell scripts**
- **Learn to use all the looping and conditional programming constructs in your shell script**
- **Learn to pass command line arguments to the shell program and**
- **Interrupt the program during its execution**

You've seen that the shell is used to interpret command lines, maintain variables, and execute programs. The shell is also a programming language. You can store a set of shell commands in file. This is known as a shell script or shell programming. By combining commands and variable assignments with flow control and decision making, you have a powerful programming tool. Using the shell as a programming language, you can automate recurring tasks, write reports, and build and manipulate your own data files.

Shell Startup--Environment Control

When a user begins a session with UNIX and the shell is executed, the shell creates a specified environment for the user. The following sections describe these processes.

Shell Environment Variables

Shell maintains a set of variables, which are known as system variables or environmental variables. Some of these variables are set during the booting sequence and some after the user logs in. These variable values can even be altered to hold your values.

Use the set command to have a complete list of all these variables.

```
$ set
HOME=/usr/kumar
IFS=
MAIL=/usr/mail/kumar
PATH=/bin:/usr/bin:.
PS1=$
SHELL=/bin/sh
```

```
TERM=ansi
$_
```

The PATH Variable

```
$ echo $PATH
/bin:/usr/bin:.
$_
```

This shows the list of directories that have to be scanned by the shell while hunting for a command. In this case, the search will begin from /bin, through /usr/bin, and finally to the current directory indicated by a .(dot). This sequence makes the commonly used UNIX commands to universally available to every user.

You can modify this value to include another directory name in its list. You can store all your shell programs in a separate directory named progs and then modify PATH to include this directory in its search path.

```
$PATH=$PATH:/usr/kumar/progs
$echo $PATH
/bin:/usr/bin:./usr/kumar/progs
$_
```

The HOME Variable

When you log in, UNIX normally places you in a directory named after your login name. This directory is called the home directory or login directory, and is controlled by the variable HOME. You can switch from any directory to your home directory by using the evaluated variable $HOME as an argument to the cd command.

```
$pwd
/usr/bin
$ cd $HOME
$pwd
/usr/kumar
$_
```

The cd command used without arguments also resets the current directory to the directory assigned to HOME.

examples:

```
$ echo $TERM
$ echo $PATH
```

You can easily change the variables the same way you assign values to any shell variable.

The IFS Variable

IFS contains a string of characters which are used as word separators in the command line. The string normally consists of the space, tab, and the newline characters.

```
$echo "$IFS" | od –bc
0000000          \t    \n    \n
               040  011   012   012
0000004
$_
```

where the space character is represented by the ASCII octal value 040.

The MAIL Variable

MAIL determines where, in which directory the incoming mails are to be stored for a user. This directory is searched by the shell every time a user logs in. If any information is found here, then the shell informs the user of this with the familiar message "You have mail"

The Variables PS1 and PS2

Unix has two prompts, stored in PS1 and PS2. The primary prompt string PS1 is the one you normally see. PS2 is the secondary prompt which you see while using the sed and awk commands.

```
$PS1="*"
*_
echo $PS1
*
*_
```

The SHELL Variable

SHELL determines the type of shell that a user sees on logging in. There are a host of shells that accompany any UNIX system and you can select the one you like the most. Besides the most popular Bourne shell, you have even C and Korn shells, which are known by the program csh and ksh respectively.

The TERM Variable

TERM indicates the terminal type being used. There are some utilities that are terminal dependent, and they require knowing the type of terminal you are using, especially the vi editor wont work properly, if the TERM variable is not set properly.

Shell Script that explains the functions of the system variables
$ cat sysvar.sh

```
# This script explains the functions of the system variables
```

```
echo “Our home directory is $HOME”
echo “The two prompt strings are $PS1 and $PS2”
echo “And the path for command search is $PATH”
```

Make the Shell Script Executable & Run

```
$chmod u+x sysvar.sh
$sysvar.sh
Our home directory is /usr/kumar
The two prompt strings are $ and >
And the path for the command search is /bin:/usr/bin:.
$_
```

The script Executed During Login Time – The .profile

After immediately a user logs in, shell executes a hidden file called **.profile** which is present in the home directory of the user. This file can contain all the commands to be executed and the environmental variable initializations that are to be done automatically after the user logins in.

Making the Shell Scripts Interactive – The read statement

To take input from the user interactively, the read statement can be used with one or more variables.

$ cat emp1.sh

```
# script : emp1.sh – Interactive version
# The pattern and filename to be supplied by the user
tput clear   #clears the screen
echo “\n Enter the pattern to be searched : \c”
read pname
echo “\n Enter the file to be used \c”
read flname
echo “\n Searching for $pname from file $flname \n”
grep “$pname” $flname
echo “\nSelected records shown above”
```

$_

Special parameters related to command line arguments

Like C language, shell also accepts arguments in command line. This non-interactive method of specifying the arguments is quite useful for scripts
requiring few inputs.

When arguments are specified along with the name of the shell procedure, they are assigned to certain special "variables" or "parameters". The first argument is read by the shell into a special parameter $1, the second argument into $2, and so on. These variables are appropriately called positional parameters.

Apart from these positional parameters, there are some special parameters also available, which are listed below,

$* Stores the complete set of positional parameters as a single string
$0 Stores the name of the program
$# Stores the count of arguments passed

$ cat emp2.sh

```
echo "program : $0"
echo "The number of arguments specified is $#
echo "The arguments are $*"
grep "$1" $2
echo "\n Job Over"
```

$_

Other Special Parameters

Shell maintains certain special parameters the details of which are given below,

Shell Parameter	**Significance**
$$	PID of the current shell
$!	PID number of last background job
$?	Exit status of last command

e.g

```
$echo $$
49
$ kill –9 $$                    # Kills the login shell

$ sort –o emp.out  emp.lst &
345
$ kiil $!                       # kills the last background process

$grep director emp.lst > /dev/null ; echo $?   # Exit status of the command
0
```

```
$ grep manager emp.lst > /dev/null ; echo $?
1
$ grep manager emp3.lst > /dev/null ; echo $?
 grep : can't open emp3.lst
```

Conditional Execution - The Logical Operators && and ||

The && operator is used by the shell in the same sense as it is used in awk and C. It delimits two commands. The second command is executed only when the first succeeds.
e.g

```
$ grep 'director' emp.lst && echo "Pattern found in file"
1001|Chanchal singhvi          |director|sales  |03/09/38|6700
pattern found in file
$_
```

In the case of || operator, the second command is executed only if the first command fails.

```
$ grep 'director' emp.lst || echo "Pattern Not found"
Pattern Not found
$_
$ cat emp.lst > /dev/null &&          # checks whether the file is existing
> grep 'manager' emp.lst ||           # checks for the pattern manager
> echo "Pattern Not found"            # displays the message if no pattern
Pattern Not found
$_
```

Script Termination – The exit statement

The exit statement is used to prematurely terminate a program. When this statement is encountered in a script, execution is halted and control is returned to the calling program, in most cases the shell.
e.g

```
$ grep "$1" $2 || exit 2
$ echo  "Pattern Found – Job Over"
```

The exit statement also takes an optional argument. When you specify one, the script will terminate with a return value as specified. Otherwise, the return value will be zero (true) and this will be assigned to the parameter $?.

The if Conditional

The if statement, takes two-way decisions, depending on the fulfillment of a certain condition. If evaluates a condition which accompanies its command line. If the condition is fulfilled, then the sequence of commands following it is executed.

```
 if <condition is true >
 then
<execute commands>
else
        <execute commands>
fi
```

```
if <condition is true >
then
<execute commands>
elif <condition is true>
then
        <execute commands>
else
        <execute commands>
fi
```

e.g

```
$ if grep "director" emp.lst
> then
>     echo "Pattern Found – Job over"
> else
>     echo "Pattern Not found"
>fi
```

if's Companion – test

test is an internal feature of shell which evaluates the condition placed on its right, and returns either true or false exit status. This return value is used by if for taking decisions. For this, test uses certain operators to evaluate the condition which are listed below,

Operator	Meaning
-eq	Equal to
-ne	Not Equal to
-gt	Greater than
-ge	Greater than or Equal to
-lt	Less than
-le	Less than or Equal to

test doesn't display any output, but simply returns a value, which is assigned to the parameter $?.

e.g.

```
$x=5
$y=7
$z=7.2
$test $x –eq $y ; echo $?
1
$ test $x –lt $y ; echo $?
0
$test $z –eq $y ; echo $?
0
$ test $z –gt $y ; echo $?
```

1
$_

The last 2 statements show that 7.2 is not greater than 7 but equal to 7. This is the major drawback of Bourneshell, which makes it unsuitable for testing decimal numbers.

e.g
$ cat emp3.sh

```
if test $# -ne 3 # 3 arguments not entered
 then
        echo "You have not keyed in 3 arguments"
        exit 3
else
        if grep "$1" $2 > $3
        then
                echo "Pattern Found – Job Over"
        else
                echo "Pattern Not found – Job Over"
        fi
fi
```

$_

test – file tests
test can be used to test the various file attributes. For example, you can test whether a file has the necessary read, write or execute permissions. The list of file and string related tests are given below,

Test	**Exit Status**
-f <file>	True if <file> exists and is a regular file
-r <file>	True if <file> exists and is readable
-w <file>	True if <file> exists and is writable
-x <file>	True if <file> exists and is executable
-d <file>	True if <file> exists and is a directory
-s <file>	True if <file> exists and has a size greater than zero.

String Related	
-n stg	True if string stg is not a null string
-z stg	True if string stg is a null string
s1=s2	True if string s1 = s2
s1!=s2	True if string s1 is not equal to s2
stg	True if string stg is assigned and not null
Logical Operators	
-a	Logical AND
-o	Logical OR

$ cat filetest.sh

```
echo “Enter the name of the file : \c”
read flname

if [ ! –f $flname ]
then
        echo “File does not exist”
elif [ ! –r $flname ]
then
        echo “File is not readable”
elif [ ! –w $flname ]
then
        echo “File is not writable”
else
        echo “File is both readable and writable”
fi
```

$_

$ cat emp5.sh

```
echo “Enter the string to be searched : \c”
read pname
if [ -z “$pname” ]
then
        echo “You have not entered the string”
        exit 1
fi
echo “Enter the file to be used : \c”
read flname

if [ ! –n “$flname” ]
then
        echo “You have not entered the filename”
        exit 2
else
        echo “Search for $pname in $flname ? : \c “
        read answer

        if [ “$answer” =  “y” ]
        then
                grep “$pname” “$flname” || echo “Pattern Not Found”
        else
                exit 3
        fi
fi
```

$ cat emp6.sh

```
if [ -n "$pname" –a –n "$flname" ]
then
        grep "$pname" "$flname" || echo "pattern Not Found"
else
        echo "Atleast one input was a null string"
        exit 1
fi
```

THE case STATEMENT

The case statement is the second conditional construct offered by the shell. The syntax of the case construct is given below,

```
case <expression> in
        <pattern1>) <execute commands>;;
        <pattern2>) <execute commands>;;
        <pattern3>) <execute commands>;;
        <……>
        <……>
esac
```

$cat menu.sh

```
echo "1.List"
echo "2.Processes of user"
echo "3.Today's Date"
echo "4.Users of system"
echo "5.Quit to UNIX"

echo" Enter your option :\c"
read choice
echo
case "$choice" in
1)  ls –l;;
2)  ps –f;;
3)  date ;;
4)  who;;
5)  exit ;;
esac
```

$

LOOPING –THE WHILE STATEMENT

Loops let you perform a set of instructions repeatedly. The shell features three types of loops –while,until and for. The first two are basically complementary to each other. All

of them repeat the instruction set enclosed by certain keywords as often as the control command permits/

The while statement should be quite familiar to most programmers. It repeatedly performs a set of instructions till the control command returns a
true exit status. The general syntax of this command is as follows:

```
while <condition is true>
do
        <execute commands>
done
```

The keywords here are do and done. The set of instructions enclosed by do and done are to be performed as long as the condition remains true. Like in the if statement, this condition is actually the return value of a UNIX command or program. This means that you can use the test statement here also, with its associated expressions, numeric and string comparisons, and file tests.
The following loop displays the ps –f output after a regular interval :

```
$x=5
$while [ $x  -gt 3 ]
do
        ps –f
        sleep 5
done
```

Any loop is treated as complete only when the words do and done are entered in the right order. The assignment x=5 is made before the while construct, and then the condition [$ -gt 3] is tested. It is true, and the loop executes. After the first iteration, the condition is again tested, and the instructions are again executed. The result is a display every five seconds of the ps –f command output with an infinite loop. You can terminate the session by pressing the interrupt key.

break and continue

break statement, causes control to break out of the loop. On the other hand, the continue statement suspends execution of all statements following it and switches control to the top of the loop for the next iteration.

The script dentry1.sh makes use of all these features

$ cat dentry1.sh

```
flname=desig.lst
while echo “Designation code :\c”
```

```
do
        read desig


case "$desig" in
        [0-9] [0-9]) if grep "^$desig" $flname >/dev/null # If code exists
                        then
                                echo "Code exists"
                                continue
                        fi;;
                    *) echo  "Invalid code"
                        continue;;
esac
while echo "Description   :\c"
do
        read desc
        case "$desc" in
        *[!\a-zA-Z]*)  echo "Can contain only alphabets and spaces"
                        continue;;
                        ") echo "Description not entered"
                         continue;;
                        *) echo "$desig | $desc ">> $flname
                        break
        esac
done
echo "\nWish to continue?(u/n): \c"
read answer
        case "$answer" in
                [yY]*) continue;;
                    *) break;;
        esac
done
echo "Program terminated"
```

$_

The until STATEMENT

The until statement complements the while construct in the sense that the loop body here is executed repeatedly as long as the condition remains false.
It is another way of viewing the whole thing.

```
$until false
do
        ps –f
        sleep 5
done
```

The for Loop

The for loop is different in structure from the ones used in other programming languages. There is no next statement here, neither can a step be specified. Unlike while and until, it doesn't test a condition but uses a list instead. The syntax of this construct is as follows:

```
for variable in list
do
        <execute commands>
done
for x in 1 2  3 4
do
        echo "The value of x is $x"
done
The value of x is 1
The value of x is 2
The value of x is 3
The value of x is 4
$_

for var in $PATH $HOME $MAIL
do
        echo "$var"
done
/bin:/usr/bin:.
/usr/kumar
/usr/mail/kumar
$_


for file in *.c
do
        cc $file
done
$_
```

THE set AND shift STATEMENTS

The set statement makes it possible to convert its arguments into positional parameters. A simple use of this statement can be

```
$ set 6767 4545 9789
$_
```

This assigns the value 6767 to the positional parameter $1, 4545 to $2 and 9789 to $3. It also sets the other parameters $# and $*.You can verify this by echoing each parameter in turn:

```
$ echo $1
6767
$ echo $2
4545
$ echo $3
 9789
$ echo $#
3
$ echo $*
6767 4545 9789
$_
```

This feature is especially useful for picking up individual fields from the output of a program. Thus, using command substitution, you can assign values to the shell's positional parameters with the output of the date command.

```
$ set `date`
$ echo $*
Wed Jan 24 14:02:33 EST 2001
$echo $2 $3 $6
Jan 24 2001
$_
```

shift statement is used to transfer the contents of a positional parameter to its immediate lower numbered one. This goes on as many times as the statement is called. When called once, $2 becomes $1, $# becomes $2 and so on. Try this on the positional parameters that were filled up with the date command:

```
$ echo  $*
Wed Jan 24 14:02:33 EST 2001
$ echo $1 $2 $3
Wed Jan 24
$shift
$ echo $1 $2 $3
Jan 24 14:02:33
$ shift
24 14:02:33 EST
$_
```

The important thing to remember is that the contents of the leftmost parameter, i.e. $1 are lost every time shift is invoked. In this way you can access more than nine positional parameters in a script, which you couldn't have done earlier. So if a script now uses twelve arguments, you can shift thrice and then use the ninth parameter :

```
$ echo $#
12
$shift
```

```
$shift
$shift
$echo $#
9
$_
```

Summary

The shell is a remarkable language, which will fascinate a programmer. It has most of the standard features of modern day languages, including variables, conditionals and loops. Besides, the facility to use the exit status of a program to control the program flow is a a highlight of the Unix system. Since programming is a built-in feature of the shell, that again leads to further advantages. This gives it a certain kind of power, which is somewhat unique in the programming world.