-: INTRODUCTION:-

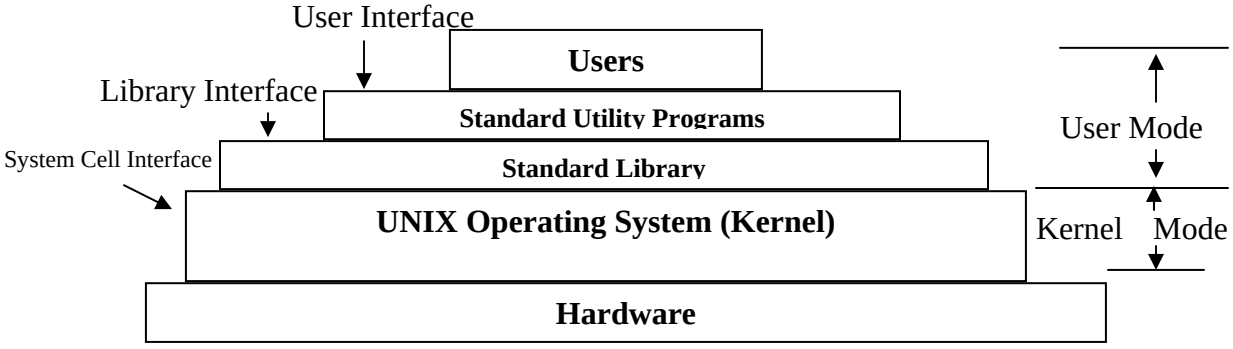
History of UNIX

In 1970, Ken Thompson and Dennis Ritchie at AT&T Bell labs developed UNIX operating system. Earlier this system was known as UNICS (Uniplexed Information and Computer Service), but later it became famous as a UNIX.

In 1973, it was re-written in 'C' programming language (higher-level language). This made UNIX the world's first portable operating system, capable of being easily ported (moved) to any hardware. UNIX became easy to understand and modifiable. This was a major advantage for UNIX. This step led to the wide acceptance of UNIX among users at that time. Today's tremendous popularity of UNIX is also responsible to this step.

The Architecture of UNIX

The various layers depicted in UNIX architecture are shown in below figure and also described below.



1. Hardware

- The bottom layer is the hardware.
- It consists of various physical devices such as CPU, memory, disks, monitor, printer, etc…
- These devices provide various services. For example, printers are used for printout purposes.

2. UNIX Operating System (Kernel):

- The next higher layer is the UNIX operating system.
- It is also called system kernel, or simply, kernel. (It is described next).
- It manages all the underlying hardware.
- It directly interacts with the hardware and provides user programs required services.
- It hides the complex details of hardware also.

- In short, it provides the simple interface between user programs and hardware.
- The main services of an operating system includes process management, memory management, file system management, I/O management etc.

3. Standard Library:

- Above operating system, next layer is for standard library.
- It contains a set of procedures, one procedure per system cell.
- These procedures are written in assembly language and used to invoke various system calls from user programs.

4. Standard Utility Programs:

- In addition to operating system and system call library, all versions of UNIX supply a large number of standard programs.
- Such programs include command processor (Shell), compilers, editors, text processing programs, file manipulation utilities, a variety of commands and so on.
- Such programs make the user tasks simpler. Users interact with them and they, in turn, interact with the operating system to get services from operating system.

5. Users:

- The top most layers are of users.
- User programs come in this layer. They interact with the system either by using library procedures to invoke system calls, or by using utility-program such as shell.

Here, some new terms came in picture such as kernel, shell, and system call. These are described below.

- **Kernel:**

- The kernel is the core of the UNIX operating system.
- UNIX uses microkernel approach, where code from the kernel is moved up in higher layers to keep it as small (thin) as possible.
- Kernel is a program, which is loaded in memory when system is turned on. It stays there and provides various services until the system is turned off.
- Kernel interacts with the hardware directly. When user program needs to use any hardware, it has to use services provided by the kernel. Special functions, called system calls, are used to request kernel. Kernel performs the job on behalf of the user process.
- In addition to providing services to user programs, kernel also provides other services like process management, memory management, file system management and so on.
- In short, kernel manages entire computer system.

● **Shell:**

- The shell is an interface between the user program and the kernel.
- When user logs-in to the system, process for shell starts execution. It terminates when user logs-out from the system.
- Users can directly interact with the shell.
- It works as a command interpreter. It accepts commands from user and translates them into the form, which the kernel can understand easily.
- It is also a programming language. It provides various programming functionalities such as looping, branching and so on.

● **System Call:**

- System calls are special functions.
- They are used to request kernel to provide various services, such as reading from a file stored on hard disk.
- They can be invoked via library procedures, or via command provided by shell, or even directly from C programs in UNIX.
- System calls are similar to user-defined functions. Difference is that they execute in the kernel mode, having fully access to all the hardware; while user defined functions execute in user mode, having no direct access to the hardware.
- Various flavors of UNIX have one thing in common. They all use the same set of system calls provided by POSIX standard. If any operating system is using other system calls, then it will not be a UNIX operating system.

● **Features of UNIX**

Many operating systems are there in market, all operating systems having their own functionality which operating system user should chose that is depends on the task. As being an operating system, UNIX contains all the features that any operating system should have, however, in addition to such common features, UNIX contains some special features, which makes it so much popular. These features are as given below.

1. Files and Process:

- In UNIX, every thing is either a file or a process.
- A file is a collection of data. They are used to store large amount of information permanently. Directories are considered as files. In addition, physical devices are also considered as files in UNIX.
- A process is a program in execution. All commands execute as processes and perform their tasks.

2. A Multi-User System:

- UNIX is a multi user system. (Dos is a single user system.)
- This means that it allows multiple users to work simultaneously on the same system.
- Different users can login from different machines into the same machine by using programs like 'TELNET'.

3. Multi Tasking System.

- UNIX is a multi-tasking system too. (Dos is a single tasking system)
- It allows multiple programs to run simultaneously.
- Among simultaneously running processes, one process will be foreground process. User can interact with this process directly. While other processes will be background processes. They execute in background without requiring user interaction.

4. The Building Block Approach:

- UNIX uses the building block approach to perform complex tasks.
- It provides a few hundred commands each of which can perform one simple job to perform complex tasks; such simple commands can be combined using pipes and filters. Thus, the small is beautiful philosophy is implemented here.

5. Portability

- Unix is written in high-level language, rather than low level assembly language
- This makes it easy to read, understand, change and move to other machine.
- Thus, it provides portability.

6. Consistent format for files

- UNIX does not provide special formats for different file types.
- It treats all files same, and provides consistent format for files in for of simple byte streams.
- It is the responsibility of the application programs to interpret file contents as per requirements.
-

7. Pattern Matching

- UNIX provides very sophisticated pattern matching capabilities.
- There is a set of special character like as '*' that can be used in pattern matching.

8. Programming facility:

- The UNIX shell is also a programming language.

- It supports all the programming features such as variables, control structures, loops so on.
- These features can be used to develop shell programs, called shell scripts. Such programs can be used to control and automate many of the system's functions.
-

9. On-line help

- Unix provides an on-line help facility for all the commands
- For this purpose, it provides a command named 'man'. By using this command, user can have an instant help on any command.

These features make the UNIX operating system popular among other operating systems.

-: TYPES OF SHELL:-

The kernel program is usually stored in a file called 'unix' whereas the shell program is un a file called 'sh'. For each user working with UNIX at any time different shell programs are running. Thus, at a particular point in time there may be several shells running in memory but only one kernel. This is because; at any instance UNIX is capable of executing only one program as the other programs waits for their turn. And since it's the kernel which executes the program one kernel is sufficient. However, different users at different terminals are trying to seek kernel's attention. And since the user interacts with the kernel through the shell different shells are necessary.

Bourne Shell (Bash)

Among all, Steve Bourne's creation, known after him as the Bourne Shell, is the most popular. Probably that's why it is bundled with every UNIX system. Or perhaps it is the other way round. Because it was bundled with every system it became popular. Whatever the cause and the effect, the fact remains that this is the shell used by many UNIX users.

C Shell (Corn)

This shell is a hit with those who are seriously into UNIX programming. It was created by Bill Joy, then pursuing his graduation at the University of California at Berkeley. It has two advantages over the Bourne Shell.

First, it allows aliasing of commands. That is, you can decide what name you want to call a command by. This proves very useful when lengthy commands which are used time and again are renamed by you. Instead of typing the entire command you can simply use the short alias at the command line.

If you want to save even more on the typing work, C shell has command history feature. This is the second benefit that comes with C Shell. Previously typed commands can be recalled,

since the C shell keeps track of all commands issued at the command line. This feature is similar to the one provided by the program DOSKEY in MS-DOS environment.

Korn Shell (Ksh)

If there was any doubt about the cause-effect relationship of the popularity of Bourne Shell and it's inclusion in every package, this adds fuel to it. The not-so-widely-used Korn Shell is very powerful, and is a superset of Bourne Shell. It offers a lot more capabilities and is decidedly more efficient than the other. It was designed to be so by David Korn of AT & T's Bell Labs.

LOGIN COMMANDS

1. passwd:

This command is used to change the password for logged in user. If your account doesn't have a password or has one that is already known to others, you should change it immediately. This is done with the passwd command.

Syntex

$passwd ↵
Enter login password: *******
New password: *******
Re-enter new password: ********
passwd (SYSTEM): passwd successfully changed for the particular user

passwd expects you to respond three times. First, it prompt for the old password. Next, it checks whether you have entered a valid password, and if you have, it then prompts for the new password.

2. logout:

This command is used to logout or terminate the session there is other way to logout with exit command and also you can terminate session with the press of ctrl +d but these commands may not be work every where to terminate the session the most reliable command is logout.

Syntex:

$logout ↵

3. Who:

This command is used to get information about users (including yourself), you can use several commands. The **who** command reports on users who are presently logged in. The who command normally reports certain information about logged-in users. By using its options, you can specify that it report information about the processes initiated by init, and that it report reboots changes to the system clock, and logoffs. If you invoke who with no option or argument, you get the following output:

Syntex:

$who

Output:

Username tty00 Date
Username tty01 Date

Option

-u This option is used to see "time since the last activity" (idle time) and the process ID number for each logged-in user. A "process ID number" is an integer number assigned by UNIX to uniquely identify a given process.

Example

$who –u

Output

Username tty00 Date Process No.
Username tty01 Date Process No.

-q This option is used to shows login IDs and a count of the logged-in users q stands for quick.

Example

$who –q

Output

User Name Terminal Name
User Name Terminal Name
#Users = 2

4. who am i

Sometime we logged in from several terminals under different accounts, and we forget that which username or terminal name we are using at that time this command is useful to know username and terminal name.

Syntex:

$who am i

Output

User Name Terminal Name Date and Time

5. Clear:

This command is used to clear the screen. This command works like cls command of dos.

Syntex:

$clear

UNIX FILE SYSTEM STRUCTURE

UNIX looks at everything as a file and any UNIX system has thousands of files. If you write a program, you add one more file to the system. When you compile it, you add some more. Files grow rapidly, and if any are not organized properly, you will find it difficult to locate them. Proper file organization requires a directory-based storage system. Just as an office has separate file cabinets to group files of a similar nature, UNIX also organizes its own files in directories and expects you to do that as well.

The file system in UNIX is one of its simple and conceptually clean features. It lets users access other files not belonging to them, and without infringing on security. It also offers an adequate security mechanism so that outsiders are not able to tamper with a file's contents.

The file is a container for storing information. As a first approximation, we can treat it simply as a sequence of characters. If you name a file foo and write three characters a, b and c into it, then foo will contain only the string abc and nothing else.

The Types of Files:

UNIX treats directories and devices as files as well. A directory is simply a folder where you store file and other directories. All physical devices like the hard disk, memory, CD-ROM, printer and modem are treated as files. The shell is also a file, and so it the kernel. And if you are wondering how UNIX treats the main memory in your system, it's a file too!

So we have already divided files into three categories:

1. Ordinary (Regular) File:

This is the most common file type in UNIX. All programs you write belong to this type. An ordinary file itself can be divided into to types.

(A) Text File:

A text file contains only printable characters, and you can often view the contents and make sense out of them. All C and Java program sources, shell and perl scripts are text files. A text file contains lines of characters where every line is terminated with the new line character, also known as line feed. When you press enter while inserting text, the LF character is appended to every line. You won't see this character normally, but there is a command which can make it visible.

(B) Binary File:

On the other hand, contains both printable and nonprintable characters that cover the entire ASCII range 0 – 255. Most UNIX commands are binary files, and the object code and executables that you produce by compiling C programs are also binary files. Picture, sound and video files are binary files as well. Displaying such files with a simple eat command produces unreadable output and may even disturb your terminal's settings.

2. Directory File:

A directory contains no data, but keeps some details of the files and subdirectories that it contains. The UNIX file system is organized with a number of directories and subdirectories, and you can also create them as and when you need. You often require to do that to group a set of files pertaining to a specific application. This allows two or more files in separate directories to have the same filename.

A directory file contains an entry for every file and subdirectory that it houses. If you have 20 files in a directory, there will be 20 entries in the directory. Each entry has two components.

- The file name.
- A unique identification number for the file or directory.

If directory contains an entry for a file named abc, we commonly say that the directory contains the file abc. Though we shall often be using the phrase "contains the file" rather than "contains the file name", you must not interpret the statement literally. A directory contains the file name and not the file's contents.

3. Device File:

You will also be printing files, installing software from CD-ROM r backing files t tape. All these activities are performed by reading or writing the file representing the device. For instance, you print a file by writing the file representing the printer. When you restore files from tape, you read the file associated with the tape drive. The kernel takes care of this "reflection" by mapping these special files to their respective devices. It is advantageous to treat devices as files as some of the commands used to access an ordinary file also work with device files.

Device file names are generally found inside a single directory structure /dev. A device file is indeed special; it's not really a stream of character. In fact, it doesn't contain anything at all. You will soon learn that every file has some attributes that are not stored in the file but elsewhere on the disk. It's the attributes f a device file that entirely governs the operation of the device. The kernel identifies a device from its attributes and then uses them to operate the device.

FILE PERMISSION

Before we take up file permission, you need to understand the significance of file ownership. When you create a file, your username shows up in the third column of the file's listing; you are the owner of the file. Your group name is seen in the fourth column; your group is the group owner of the file. If you copy someone else's file, you are the owner of the copy. If you can't create files in other users' home directories, it's because those directories are not owned by you.

UNIX has a simple and well-defined system of assigning permissions to files. Issue the ls –l command once again to view the permissions of a few files.

Categories of user		**Types Of Permission**	
User	u	Read	r
Group	g	Write	w
Other	o	Execute	x

```
$ls –l

-rwxr-xr--    1    kumar  metal   20500  may 10 19:21 chap02
-rwxr-xr-x    1    kumar  metal     890  Jan  29 23:17 dateval.sh
-rw-rw-rw-    1    kumar  metal      84  Feb  12 12:30 dept.lst
```

Observe the first column that represents the file permissions. These permissions are also different for the three files. UNIX follows a three-tired file protection system that determines a file's access right. To understand how this system works, let's break up the permissions string of the file chap02 into three groups. The initial – (in the first column) represents an ordinary file and is left of the permissions string:

Each group here represents a category and contains three slots, representing the read, write and execute permission of the file – in that order. r indicates read permission, which means cat can display the file. w indicates write permission; you can edit such a file with an editor. x indicates execute permission; the file can be executed as a program. The – shows the absence of the corresponding permission.

The first group (rwx) has all tree permission. The file is readable, writable and executable by the owner of the file, kumar. But do we know who the owner is? Yes we do. The third column

shows kumar as the owner and the first permissions group applies to kumar. You have to log in with the username kumar for these privileges to aplly to you.

The second group (r-x) has a hyphen in the middle slot, which indicates the absence of write permission by the group owner of the file. This group owner is metal, and all users belonging to the metal group have read and execute permissions only.

The third group (r--) has the write and executes bits absent. This set of permissions is applicable to others, i.e., those who are neither the owner kumar nor belong to the metal group. This category (others) is often referred to as the world. This file is not world-writable.

You can set different permissions for the three categories of users – owner, group and others. It's important that you understand them because a little learning here can be a dangerous thing. Faulty file permission is a sure recipe for disaster.

DIRECTORY PERMISSIONS

Directory also has their own permissions and the significance of these permissions differ a great deal from those of ordinary files. You may not have expected this, but be aware that read and write access to an ordinary file are also influenced by the permission of the directory housing them. It's possible that a file can't be accessed even though it has read permission, and can be removed even when it' write-protected. In fact, it's very easy to make it behave that way.

If the default directory permission is not altered, the chmod theory still applies. However, if they are changed, unusual things can happen. Though directory permissions are taken up later, it's worthwhile to know what the default permissions are on your system:

Categories of user		**Types Of Permission**	
User	u	Read	r
Group	g	Write	w
Other	o	Execute	x

```
$mkdir c_progs

$ls –ld c_progs

drwxr-xr-x   2   kumar metal  512   may   9    09:57  c_progs
```

First character d displays that c_progs is directory, second, third and fourth character rwx is showing that user is having all permission read, write and execute, fifth, sixth and seventh character r-x showing that group is having only read and execute permission, eighth, ninth and tenth character r-x showing that others having read and execute permission.

The default permissions of a directory on this system are rwxr-xr-x (or 755); that is what they should be. A directory must never be writable by group and others. If you find that your files are being tampered with even though they appear to be protected, check up the directory permissions.

RELATED COMMANDS

(1) ls:

This command is used see the list of files and directory at current location or specified path. This command supports us to see the list of file and directory.

Syntex:

$ls [-Option] [Path]

Options:

–x	Multicolumn output
-F	Marks executable with * directories with / and symbolic links with @
-a	Show all filenames beginning with a dot including . and ..
-R	Recursive list
-r	Sorts filenames in reverse order (ASCII collating sequence by default)
-1	One filename in each line
-l	Long listing in ASCII collating sequence showing seven attributes of a file
-d	List only directory
-t	Sorts filenames by last modification time
-lt	Sorts filenames listing by last modification time
-u	Sorts filenames by last access time
-lu	Sort by ASCII collating sequence but listing shows last access time
-lut	As above but sorted by last access time
-i	Display inode number

(2) cat

This command is use to displays files contents, creates new file, or appends data to an existing file.

Syntax 1 To display the content of file

$cat File Name

Example

$cat Example1.txt

Syntax 2 To create a new file

```
$cat  >  File Name
Enter the file content
Ctrl + d
```

Example

```
$cat  >  Example2.txt
Enter the file content
Ctrl+d
```

Syntax 3 To append existing file

```
$cat   >> File Name
Enter the file content want to append
Ctrl+d
```

Example

```
$cat  >>Example1.txt
Enter the file content want to append
Ctrl+d
```

(3) cd (Change Directory)

This command is used to change one directory forward, backward and supporting to go to root directory directly.

Syntax

cd	Change working directory to home directory.
cd ..	Change working directory to parent directory.
cd Path	Change working directory to specified directory.
cd /	Change working directory to root directory.

Example

If current directory /home/abc and user home directory is /home/guest then

$cd will take you to home directory

now current directory is /home/guest

$cd .. will take you to home directory

now current directory is /home

$cd /home/abc

now current directory is /home/abc

$cd /

now current directory is / (Root)

(4) pwd: (Path for working directory)

This command is used to see path for working directory in unix you can not see the prompt with path at that time if we want to know the what is our current path. The pwd command supports to know path for working directory.

Syntax

$pwd

Example

If current directory is /home guest then

$pwd will show you /home/guest

(5) mv command (Renaming Files and Moving a file from one place to another place)

The mv command renames files. It has two distinct functions:

- It renames a file or directory.
- It moves a group of files to a different directory.

To rename a file or directory

Syntax

$mv Old File Name New File Name

mv doesn't create a copy of the file; it merely renames it. No additional space is consumed on disk during renaming. To rename the file abc.txt to xyz.txt you can write following command.

$mv abc.txt xyz.txt

If the destination file doesn't exist, it will be created. For the above example, mv simply replaces the filename in the existing directory entry with the new name. By default, mv doesn't prompt for overwriting the destination file if it exists. So be careful.

To move a file from one place to another

Syntax

$mv file name Destination folder or path of Destination folder

Example

To move a.txt file from root to home/abc directory then following command is used from root.

$mv a.txt /home/abc

Above command will move file named a.txt from root to /home/abc

(6) cp (Copying a file)

This command is used to copy a file.

Syntax

$cp [option] Source file path or file Target path

Option

-i Asks before overwriting a destination file
-r It copies directory structure.

Example

To copy a a.txt file from /home/abc to /home/guest following command is used

$cp /home/abc/a.txt /home/guest

Above command will copy a.txt file from /home/abc to /home/guest directory

(7) ln (Link a file)

This command is used to link a file. Links are used for sharing of files from two different locations in directory structure. There are two types of link 1) soft link 2) hard link.

Soft Link:- It links file name if source is deleted then link will not be found

Hard Link:- It links physical storage area of the file if source file is deleted still it will target the physical part of the file and data can be found.

Syntax:

$ln [Option] Source File Name Link File Name

Option

-s Creates soft link if this option is not specified with ln command then hard link will be created.

Example

File Name abc.txt following is the content of the file
Abc abc abc abc abc abc

$ ln –s abc.txt xyz.txt

Soft link will be created if abc.txt file will be deleted then user will not be able to see the content of xyz.txt because it is related to abc.txt file name.

$ ln abc.txt xyz.txt

Hard link will be created if abc.txt file will be deleted still user will be able to work will the content of abc.txt file.

(8) rm (Remove File)

This command is used to remove (delete) a file.

Syntax

$rm [Option] [Path]File Name

Options

-i	Ask before removing a file
-r	Performs recursive deletion in directory
-f	Removes write protected file also

Example

To remove a file abc.txt from /home/guest current directory is root then following command will be written.

$rm /home/guest/abc.txt

Above command will remove abc.txt file.

(9) rmdir: (Remove directory)

This command is used to remove directory form current path and specified path.

Syntex:

$rmdir [Path] Directory Name

This command is useful to remove empty directory the prime condition of this command is this that whatever the directory is, you are going to remove it that must be empty else message will display on the screen that directory is not empty.

Error Message

rmdir: dirname: cannot remove [Directory not empty]

(10) mkdir: (Make Directory)

This command is used to create a directory on current path or specified path.

Syntex:

$mkdir Directory Name

Some time you are giving right command still directory is not going to be created for following reasons.

Error Message

Mkdir: directory name: [file exists]

(1) The directory name is given with mkdir is already exist
(2) There may be a an ordinary file by that name in the current directory
(3) The permission set for the current directory don't permit

(11) Umask

This command is used to set default file permission. Whatever the file is created by the owner at that time which permission is default given to user, group and other that default permission will be set using Umask command. Umask stands for user file creation mask. The term mask implying which permissions to mask or hide. The umask value tells Unix which of the three

permission are to be denied rather than granted. The current value of umask can be easily determined by just typing umask.

To see the default permissions

Syntax

$umask

$umask
0022

Here first 0 indicates that for owner no permission is denied, Second 0 indicates that for user no permission is denied, third number 2 indicates that write permission is denied to group and fourth number two is also indicating that write permission denied to others.

Changing default permission using umask

$umask 111

This would see to it that here onwards any new file that you create would have the permissions 555 and any directory that you create would have the permission 666.

(12) Chmod (Changing File and Directory Permissions)

A file or directory is created with a default set of permissions, and this default is determined by a simple setting called umask. Generally, the default setting write-protecccts a file from all except the user (new name for owner), though all user may have read access. However, this may not be so on your system. To know your system's default, create a file xstart:

$cat > xstart

$ ls –l

-rw-r--r-- 1 kumar metal 1906 Sep 5 23:38 xstart

Syntax Relative Permission

$chmod <<User Category>> <<Operation>> <<Permission>> <<File Name>>

User Category : User, Group, Other
Operation : Assign Or remove a permission using + and –
Permission : Read, Write, Execute

Example

To assign execute permission to the user of the file xstart, we need to write following command

$chmod u+x xstart

$ls –l

-rwxr--r-- 1 kumar metal 1906 May 10 20:30 xstart

to use multiple characters to represent the user category (ugo).

$chmod ugo+x xstart or $chmod a+x xstart or $chmod +x xstart

$ls –l

-rwxr-xr-x 1 kumar metal 1906 May 10 20:30 xstart

To use more then one file at a time

$chmod u+x note note1 note2 note 3

To grant and revoke a permission at a time

$chmod a-x,go+r xstart

Abbreviations Used By chmod

Category	Operation	Permission
u user	+ Assigns Permission	r Read Permission
g group	- Remove Permission	w Write Permission
o Other	= Assigns absolute permission	x Execute permission
a All(ugo)		

Syntax Absolute Permissions

$chmod Combination of Decimal Code

Combination of Decimal Code

- Read Permission - 4 (Octal 100)
- Write Permission - 2 (Octal 010)
- Execute Permission - 1 (Octal 001)

Binary	Octal	Permissions	Significance
000	0	---	No permission
001	1	--x	Executable Only
010	2	-w-	Writable Only
011	3	-wx	Writable and Executable
100	4	r--	Readable Only
101	5	r-x	Readable and Executable
110	6	rw-	Readable and Writable
111	7	rwx	Readable, Writable and Executable

Example 1

$chmod 666 xstart;

$ls –l xstart;

-rw-rw-rw- 1 kumar metal 1906 May 10 20:30 xstart

The 6 indicates read and write permission (4 + 2). To restore the original permissions to the file, you need to remove the write permission (2) from group and others:

Example 2

$chmod 644 xstart

$ls –l xstart

-rw-r--r-- 1 kumar metal 1906 May 10 20:30 xstart

To assign all permissions to the owner, read and write permissions to the group, and only execute permission to the other, use following command.

Example 3

$chmod 761 xstart

-rwxrw---x 1 kumar metal 1906 May 10 20:30 xstart

(13) Chown: (Change Owner)

This command is used to change the owner of file. It transfers ownership of a file to a user, and it seems that it can optionally change the group as well. The command requires the user-id (UID) of the recipient, followed by one or more filenames. Changing ownership requires super user permission, so let's first change our status to that of super user with the su command.

```
$su
Password:********
#_
```

After the password is successfully entered, su returns a # prompt, the same prompt used by root. su lets us acquire super user status if we know the root password. To now renounce the ownership of the file note to sharma, use chown in the following way:

Syntax:

Chown < New Owner Name> file(s)

Example

```
#ls –s note

-rwxr----x    1    kumar    metal    347    May    10    20:30    Note

#chown sharma note;

#ls –l

-rwxr----x    1    sharma   metal    347    May    10    20:30    Note

#exit

$_
```

Once ownership of the file has been given away to sharma, the permissions that previously applied to kumar now apply to sharma. Thus, kumar can no longer edit note since there's no write privilege for group and others. He can't get back the ownership either. But he can copy the file to his own directory, in which case he becomes the owner of the copy.

(14) chgrp: (Changing Group Owner):

By default, the group owner of a file is the group to which the owner belongs. The chgrp (change group) command changes a file's group owner. A user can change the group owner of a file, but only to a group to which she also belongs. Yes, a user can belong to more than one group.

chgrp shares a similar syntax with chown. In the following example, kumar changes the group ownership of dept.lst to dba (No super user permission required):

Syntax:

chgrp <<New Group Name>> <<File Name>>

```
$ls –l dept.lst

-rw-r--r--    1    kumar  metal    139    Jun    8    16:43   dept.lst

$chgrp dba dept.lst

-rw-r--r--    1    kumar  dba      139    Jun    8    16:43   dept.lst
```

If kumar is also a member of the dba group then above command will work, if he is not a member of the dba group then super user can make the command work. Note that kumar can reverse this action and restore the previous group ownership (to metal) because he is still owner of the file and consequently retains all right related to it.

(15) Find (Locating Files)

This command is used to find a particular file.

Syntax

$find [Path]/File Name

Example 1

$find abc.txt

$find /home/guest/abc.txt

User can use wild card character with file name * means all character and ? means 1 character

Example 2

To find the file which having extension name .txt following command is used

$find *.txt

Example 3

To find the file which 2nd character is d then following command is used

$find ?d*.*

(16) Pg commands

This command is used to see the information of the document page wise.

Syntax

$pg Starting Line Number No of Lines Per Page Option File Name

Example

$pg +10 –15 –p "Page No. %d" –s abc.txt

Explanation

Above command starts displaying the contents of abc.txt file, 15 lines at a time from 10th line onwards. At the end of each displayed page a prompt comes which displays the page number on view. This prompt overrides the default ':' prompt of the pg command. The –s option ensures that the prompt is displayed in reverse video.

(17) More Command (Paging out)

This command is used to see the information of the document page wise.

Syntax

$more File Name Press q to exit

Example

$more abc.txt

Explanation

You will see the contents of abc.txt on the screen, one page at a time. At the bottom of the screen, you'll also see the filename and percentage of the file that has been viewed:

----More----(17%)

More has a couple of internal commands that don't show up on the screen when you invoke them. q, the command used to exit more, is an internal command.

Navigation of More Command

Irrespective of version, more uses the spacebar to scroll forward a page at a time. You can also scroll by small and large increment of lines or screens. To move forward one page, use

f or spacebar To move page forward

b To move back one page

Enter or j	One line forward
k	One line back
p or 1G	Beginning of file
G	End of file
q	Quit

(18) less (Paging Out)

This command is used to see the information of the document page wise. This command work just like more command and it is older version.

Syntax

$less File Name	Press q to exit

Example

$less abc.txt

Explanation

You will see the contents of abc.txt on the screen, one page at a time. At the bottom of the screen, you'll also see the filename and : that is ready to accept the navigation command those are described below.

Abc.txt :

Less has a couple of internal commands that don't show up on the screen when you invoke them. q, the command used to exit less, is an internal command.

Navigation of less Command

Irrespective of version, less uses the spacebar to scroll forward a page at a time. You can also scroll by small and large increment of lines or screens. To move forward one page, use

f or spacebar	To move page forward
b	To move back one page
Enter or j	One line forward

k	One line back
p or 1G	Beginning of file
G	End of file
q	Quit

At the end of file user has to give q to quit from less command.

(19) head

We are using cat command to view the content of files, However, if the file is bigger than 24 lines then the matter would naturally scroll off the screen. If we want to stop the scrolling we can do so by hitting the pause key and resume it by hitting any other key. This of course needs a bit of a practice otherwise the matter scrolls off the screen before you can reach for the pause key. To exercise a tighter control over the way we can view files Unix proves several utilities. Out of these the head commands help in viewing lines at the beginning of the file respectively.

Syntax:

$head –Line Number File Name

Unless otherwise specified the head command assumes that you want to display first 10 lines in the file. Should you decide to view first fifteen lines you simply have to say.

$head -15 myfile

Here with head command we can specify the number of lines if we decide to override this default value.

The disadvantage of head is that they cannot display a range of lines. Moreover, what is displayed is final. That is, if we have displayed the first 50 lines in a file we cannot move back and view say the 10^{th} line. Unix provides two commands which offer more flexibility in viewing files.

(20) tail

We are using cat command to view the content of files, However, if the file is bigger than 24 lines then the matter would naturally scroll off the screen. If we want to stop the scrolling we can do so by hitting the pause key and resume it by hitting any other key. This of course needs a bit of a practice otherwise the matter scrolls off the screen before you can reach for the pause key. To exercise a tighter control over the way we can view files Unix proves several utilities. Out of these the tail commands help in viewing lines at the ending of the file respectively.

Syntax:

$tail –Line Number File Name

Unless otherwise specified the tail command assumes that you want to display last 10 lines in the file. Should you decide to view last fifteen lines you simply have to say.

$tail -15 myfile

Here with tail command we can specify the number of lines if we decide to override this default value.

The disadvantage of tail is that they cannot display a range of lines. Moreover, what is displayed is final. That is, if we have displayed the last 50 lines in a file we cannot move back and view say the 10^{th} line. Unix provides two commands which offer more flexibility in viewing files. These commands are pg and more.

(21) wc (Word Count)

This command is used to count number of characters, words and lines in an given input file. If file is not given, it takes input from standard input.

Syntax

$wc [Option] File Name

Options

-l	Counts number of lines
-w	Counts number of words
-c	Counts number of characters

Example

A file abc.txt having following data

1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10

$ wc abc.txt

Output

2 20 42 abc.txt

Note: Spacebar and Enter also known as on character

(22) touch (Changing the time stamp)

This command is used to change the time stamp.

Syntax

$touch MonthDateHoursMinutes File Name

Example

$ls –l

-rw-r--r-- 1 kumar metal 870 Mar 16 14:30 emp.lst

$touch 02281030

-rw-r--r-- 1 kumar metal 870 Feb 16 16:50 emp.lst

- : VI EDITOR : -

No matter what work you do with the UNIX system, you will eventually write some C programs or shell scripts. You may have to edit some of the system files at times. For all this you must learn to use an editor, and UNIX providers a very versatile one vi editor, Vi editor is a screen editor, where a portion of the file is displayed on the terminal screen, and the cursor can be moved around the screen to indicate where you want to make changes. You can select which part of the file you want to have displayed. Screen editors are also called display editors, or visual editors. VI is one of the more popular screen editors that run on the UNIX system.

- Invoking VI

It will put filename into a buffer, and display the file on the screen. If the file is larger than the screen can display, the screen will act as a window into the file. At the beginning of a session, the screen will display the first part of the file. If filename does not exist, VI will create it.

- Modes in VI editor

One of the most important aspects to remember about vi is that most of the commands fall into one of three modes:

(1) **Command mode:**

This is the default mode of vi editor. This mode is used to give some command for navigation, edition and copy or cut. This is the base mode of VI editor if user want to switch this mode to insert or input mode then user has to press i or I or a or A. If user wants to switch execute mode from this mode then user has to press **:**

(2) **Input or Insert Mode:**

This mode is used to enter data in vi editor. If user wants to insert some text to the editor then first user has to select this mode. This mode will be selected from command mode pressing I or i or a or A.

(3) **Execute Mode:**

This mode is used to save or quit from vi editor. What ever the changes user has done in file using VI editor if user wants to save, find any particular string then using this mode user can save or find the string and also whenever user wants to quit from the vi editor this mode will supports user to quit from the vi editor. To switch to this mode user has to press **:** or **/** on command mode

- Switching mode in VI

While you start VI editor at that time you will be at VI mode that will ready to accept defined command on that particular key but not any input. If you want to input text into the file you will have to go to input mode for that you will have to press “i”.

If you will press “i” at VI mode you will be at input mode here you can input any data into the file from this mode if you want to switch to VI mode then you will have to press “Esc” key. If you will press “Esc” key at input mode you will be able to switch to VI mode.

If you want to switch to command mode from the vi mode then you will have to press “:” or “/” that will support you to switch you from vi mode to command mode.

If you want to switch to command mode from the input mode then you will have to first come to the VI mode then you will be able to switch to command mode. To switch from input mode to command mode then you first will have to press “Esc” key and then after you will have to press “:” or “/” key that will support you to switch from input mode to command mode via VI mode.

- Cursor movement in VI editor

Whatever the command you are giving that will work on VI mode only.

To move around in the file, use the arrow keys: the up arrow will move the cursor up one line, the down arrow down one line. The right arrow will cause the cursor to move one position to the right. From the end of a line, it will go to the beginning of the next line.

For terminals with no functional arrow keys, four keys will move the cursor around:

h	left arrow
j	down arrow
k	up arrow
l	right arrow

- Screen control commands

Whatever the command you are giving that will work on vi mode only.

Ctrl + f
: This will move the user forward one screen in the file.

Ctrl+b
: This will move the user backward one screen in the file.

G
: This will move the cursor to the end of the file. Note, again that vi, like any other UNIX utility, is always case sensitive.

num
: This will bring the cursor to line defined line number.

- Entering, Editing, Copying data into vi editor

Whatever the command you are giving that will work on vi mode only.

x
: That will delete on character.

dd
: That will delete lines beginning at the current line.

yy
: That will copy lines beginning at the current line.

p
: That will paste copied line.

u
: This is very useful command. It cancels the effect of the previously executed command.

q

When in vi, typing will bring the user into command mode. As the : is typed, it will be displayed n the last line of the screen, and vi will wait for a command to be typed. q is such a command. This will exit vi, if no changes have been made since the last write-to-file command **:q!** will exit, even if the buffer has not been written to the file.

w

Will cause the contents of the buffer to be written to the file **:wq** will write the buffer to the file, and exit t the calling shell.

i

This will support you to switch from VI mode to input mode.

:

This will support you to switch from VI mode to command mode.

esc

This will support you to switch from input mode to vi mode.

/ This will support you to find any particular pattern from the file

-: CONCEPT OF REDIRECTION AND PIPING:-

Most UNIX commands expect input to come from a file(s), and produce output to anther file(s). "One of these files is the standard input, and is the place from which a program expects to read its input. Another is called the standard output, and it is the file to which the program writes its results. The third file is the diagnostic output (also called Standard error), and it is a file to which program writes any error responses. Generally, the standard input is taken to be the keyboard (input is typed by the user), the standard output is the terminal screen.

Standard Input, Output, Error

Many UNIX commands get input from what is called standard input and send their output to standard output (often abbreviated as standard input and standard output). Your shell sets things up s that standard input is your keyboard, and standard output is the screen.

Here's an example using the cat command. Normally, cat reads data from all of the files specified by the command line, and sends this data directly to standard output. Therefore, using the command:

/home/bca/cat history master

Display the contents of the file history followed by masters.

However, if you don't specify a filename, cat reads data from standard input and sends it back to standard output. Here's an example

/home/bca/cat > papers

Hello there
Hello there
Bye
Bye
Ctrl +d

Each line that you type is immediately echoed back by cat. When reading form standard input, you indicate the input is "finished" by sending an EOT (end of text) signal, in general, generated by pressing Ctrl + D.

(1) Redirection Output:

Now, let's say that you want to send the output of cat to a file, to save our shopping list on disk. The shell lets you redirect standard output to a file name, by using the ">" symbol. Here's how it works:

$cat amg.txt>output.txt

The > operator causes a new file to be created. If you had already created a file named output.txt, it would be deleted and replaced with the new data. If you wanted to add the new data to the old output, you could use the >> operator. For example:

$cat amg.txt>>output.txt

This will add content of amg.txt to output.txt file both file must be exists at specified path.

(2) Redirection Input:

There are two possible sources of input for UNIX commands. Programs such as ls and find get their input from the command line in the form of options and filenames. Other programs, such as cat, can get their data from the standard input as well as from the command line. Try the cat command with no options on the command line:

$cat

There is no response. Because n files are specified with the command, cat waits to get its input from your keyboard, the standard input file. The program will accept input lines from the keyboard until it sees a line which begins with Ctrl+D, which is the end of file signal for standard input.

To redirect the standard input, you use the < operator. For example, if you wanted cat to get its input from output.txt you could use the command.

$cat<output.txt

The difference between this command and

$cat output.txt

is a subtle one. In filenames provided as options to a command, you can use filename substitution. When redirecting input, you must use the name of an existing file or device.

(3) Standard Error:

Whenever the command is given wrongly at prompt there must be an error occurs to store the error message in file because we can see in windows base application that if any system error occurs that generates the log file for that particular error. In that way in UNIX such a file can be generated as standard error.

Example

$ cat > output.txt 2>errorfile

A message cannot open uoutput.txt will be written in errorfile. You can see such message with the help of cat errorfile.

Concept of Piping:

Suppose that you wanted a directory listing that was sorted by the mode file type plus permissions. To accomplish this, you might redirect the output from ls to a data file and then sort that data file. For example.

$ls –l>tempfile

$sort<tempfile

-rw-rw-r-	1	marsha	adept	1024	Jan 20 14:14 Lines.dat
-rw-rw-r-	1	marsha	adept	3072	Jan 20 14:14 Lines.idx
-rw-rw-r-	1	marsha	adept	256	Jan 20 14:14 Pages.dat

Although you get the result that you wanted, there are three drawback to this method:

You might end up with a lot of temporary file in your directory. You would have to go back and remove them. The sort program does not begins its work until the first command is complete. This is not too significant with the small amount of data used in this example, but it

can make a considerable difference with larger files. The final output contains the name of your temp file, which might not be what you had in mind.

Fortunately, there is a better way.

The pipe symbol (|) causes the standard output of the program on the left side of the pipe to be passed directly to the standard input of the program on the right side of the pipe symbol. Therefore, to get the same results as before, you can use the pipe symbol. For example

$ls –l|sort

```
-rw-rw-r-    1    marsha    adept   1024   Jan 20 14:14 Lines.dat
-rw-rw-r-    1    marsha    adept   3072   Jan 20 14:14 Lines.idx
-rw-rw-r-    1    marsha    adept   256    Jan 20 14:14 Pages.dat
```

To connect two or more operation within the same stream at that time pipe sign is used to perform such a operation in UNIX.

Finding patterns (Regular expression, gerp, egrep, fgrep)

UNIX was originally written for people using slow and clumsy teletype terminals. The fewer the number of characters typed the better. For that reason, many UNIX commands are very short two or three letters. The options, called flags, are also very short usually one character. The syntax for some commands will be described in this session along with some of the options or flags. However, many flags for many commands will not be mentioned. It is the user's responsibility to verify which flags/options are available on their machine. Short commands are usually not very mnemonic. But there are always tricks to help remember their meaning.

❍ Regular Expression

Regular expression as used in VI, grep, etc. Regular expressions are formed from single characters, concatenation of regular expressions, choice between two regular expressions, and zero or more occurrences of a regular expression.

In UNIX, regular expressions are defined using the following Meta symbols and extensions to the regular expression notation.

.	Match any single character.
[xyz]	Match any character enclosed in the brackets.
[A-B]	Match any character in the range.
[^xyz]	Complement: match any character not enclosed in the brackets.
re*	The Kleene star: match zero or more occurrences of the Re.
^re	RE match must occur at the beginning of a line.
re$	RE match must occur at the end of line.

\\ Represent the backslash

❍ grep Command

The grep command is used to read from the specified file on the command line.

Syntex:
$grep [option] RE [file(s)]

Option

-b Display, at the beginning of the output line, the number of the block in which the regular expression was found. This can be helpful in locating block numbers by context. (The first block is block zero)

-c Print the number of lines that contain the pattern, that is, the number of matching lines.

-h Prevent the name of the file that contains the matching line from being displayed at the beginning of that line.
Note:- When searching multiple files, grep normally reports not only the matching line but also the name of the file that contains it.

-i Ignore distinctions between uppercase and lowercase during comparisons.

-l Print one time the name of each file that contains lines that match the pattern-regardless of the actual number of matching lines in each file-on separate lines of the screen.

-n Precede each matching line by its line number in the file.

-s Suppress error messages about nonexistent or unreadable files.

-v Print all lines except those that contain the pattern. This reverses the logic of the search.

Examples:

1. $grep –i ‘you’ pax corn

Here the pax and corn are file name from where you want to find ‘you’ or ‘YOU’ because option i is indicating that ignoring case.

2. grep –c ‘you’ pax corn

Here you want to see that ‘you’ how many times repeated in the files. The output of above command will be this.

Output

pax : 1
corn: 5

3. grep –h 'you' pax corn

Above command will shows the output for the matching lines without specifying the files from which they came.

4. grep –iv 'you' pax corn

Above command specifies output of "every line in pax and corn that does not have 'you' or 'YOU'.

5. grep '^The' pax cron

The above command finds lines that begin with The.

6. $grep 'n$' pax corn

Above command finds lines that end with n.

⭘ egrep Command

egrp uses full regular expression in the pattern string. The syntax of egrep is the same as that for grep.

Syntex:
$egrep [option] RE [files]

The egrep command uses the samre regular expressions as the grep command, except for \(and\), and includes the following additional patters:

RE+	Matches on or more occurrence(s) or RE. (Contrast this with grep's RE* pattern, which matches zero or more occurrences of RE).
RE?	Matches zero or one occurrence of RE.
RE1|RE2	Matches either RE1 or RE2. The | acts as a logical OR operator.
(RE)	Groups multiple regular expressions.

The egrep command accepts the same command-line options as grep as well as the following additional command-line options:

-e	Search for a special expression (that is, a regular expression that begins with a-)
-f	Put the regular expression into file.

Example:

1. $egrep '[A-Z][A-Z]+' pax corn

The above command finds two or more consecutive uppercase letters.

2. $egrep 'DOS|SCO' pax corn

The above command finds each line that contains either DOS or SCO.

3. $egrep 'n(e|o)w' corn

The above command finds either new or now.

❍fgrep Command (Multiple string searching)

The fgrep command searches a file for a character string and prints all lines that contain the string. Unlike grep and egrep, fgrep interprets each character in the search string as a literal character, because fgrep has no meta characters.

Syntex:
$fgrep [options] string [files]

The options you use with the fgrep command are exactly the same as those that you use for egrep, with the addition of –x, which prints only the lines that are matched in their entirety.

Working with column and fields using cut and paste command

(1) cut command

Cut command is used to cut or pick up a given number of character or fields from the specified file. This command is support you to see specified filed. Like you have large database information but out of them you want to see some selected fields than cut command is useful in UNIX.

Syntax:

$cut [Main Options] Field or Character List [Options] File Name

Main Options:

-f This option is used to specify field list separated by TAB
-b This option is used to specify bytes list single character taken as bytes
-c This option is used to specify character list single character taken as character

It is must that you will have to used one main option with cut command.

Options:

-d This option is used to specify the character as field separator

-s This option is used to skip the lines which does not have field separator

Example 1:

```
$cat >> abc.txt
1.  one  two  three  four  five  six
2.  one  two  three  four  five  six
3.  one  two  three  four  five  six
```

(1) $cut –b 1,2 abc.txt (This will show you given byte)

```
1.
2.
3.
```

(2) $cut –b 1-4 abc.txt (This will show you range 1 to 4 byte)

```
1.  o
2.  o
3.  o
```

(3) $cut –c 1,2 abc.txt (This will show you given character)

```
1.
2.
3.
```

(4) $cut –c 1-4 abc.txt (This will show you range 1 to 4 character)

```
4.  o
5.  o
6.  o
```

Example 2:

```
$cat >>abc1.txt
1:one:two:three:four:five:six
2:one:two:three:four:five:six
3:one:two:three:four:five:six

$cut –f 1-3 –d":" abc1.txt
1:one:two
2:one:two
3:one:two
```

Example 3:

(1)

```
$cat >> abc.txt
1.  one  two  three  four  five  six
2.  one  two  three  four  five  six
3.  one  two  three  four  five  six
abcdefghijklmnopqrstuvwxyz
```

(2) $cut –f 1-4 abc.txt
$cat >> abc.txt
1. one two three
2. one two three
3. one two three
abcdefghijklmnopqrstuvwxyz

(3) $cut –f 1-4 –s abc.txt
1. one two three
2. one two three
3. one two three

Paste Command

This command is used to merges lines of files. This command prints lines consisting of sequentially corresponding lines of each given files, separated by tabs, terminated by a new line. If no files are given , the standard input is used. A filename of -means standard input.

Syntax:

$Paste [Option] File Name1 File Name2

Options:

-s Serial Paste the lines of one file at a time rather then one line from each file.

-d Delimiters tow files

Example 1

$cat >> 1.txt
1. aaa bbb ccc
2. aaa bbb ccc
3. aaa bbb ccc

$cat >> 2.txt
A) AAA BBB CCC
B) AAA BBB CCC
C) AAA BBB CCC

(1) $paste 1.txt 2.txt

1.	aaa	bbb	ccc	A)	AAA	BBB	CCC
2.	aaa	bbb	ccc	B)	AAA	BBB	CCC
3.	aaa	bbb	ccc	C)	AAA	BBB	CCC

(2) $paste -d"|" 1.txt 2.txt

```
1.  aaa  bbb  ccc  |A)  AAA  BBB  CCC
2.  aaa  bbb  ccc  |B)  AAA  BBB  CCC
3.  aaa  bbb  ccc  |C)  AAA  BBB  CCC
```

(3) $paste -s 1.txt 2.txt

```
1.  aaa  bbb  ccc  2.  aaa  bbb  ccc  3.  aaa  bbb  ccc
A)  AAA  BBB  CCC  B)  AAA  BBB  CCC  C)  AAA  BBB  CCC
```

(4) $paste -s -d"|" 1.txt 2.txt

```
1.  aaa  bbb  ccc  |2.  aaa  bbb  ccc  |3.  aaa  bbb  ccc
A)  AAA  BBB  CCC  |B)  AAA  BBB  CCC  |C)  AAA  BBB  CCC
```

Tools for sorting using sort and uniq

Sort:

This command is used to sort file with field number. This command is used to arrange your file alphabetically.

Syntax:

$sort [Options] File Name

Options

+Field Number	This option is used to define field on which you want to sort your file
-r	This option is used to sorting data in reverse order.

Note: Field number always start with 0

Example:

$cat >> auto.txt

```
1   h   j   k
4   a   x   d
3   d   l   i
2   j   o   m
```

(1) $sort auto.txt

```
1   h   j   k
2   j   o   m
```

```
        3       d       l       i
        4       a       x       d

(2)     $sort +1 auto
        4       a       x       d
        3       d       l       i
        1       h       j       k
        2       j       o       m

(3)     $sort -rauto
        4       a       x       d
        3       d       l       i
        2       j       o       m
        1       h       j       k

(4)     $sort -r +1 auto
        2       j       o       m
        1       h       j       k
        4       a       x       d
        3       d       l       i
```

Uniq Command

This command is used to finding out the duplicate and uniq entry from any particular files, Sometimes we merge the files at that time duplication is the big problem to prevent duplication of the records uniq command will support us to finding out duplication and uniq rows.

Syntax:
$uniq [Options] Filenames

Options:

- -c This will count any particular rows that how many times it is repeated in file
- -u This will remove the duplicate rows from the file and will print only non repeated rows
- -d This will finding out the duplicate rows and only print duplicate rows not uniq rows

Example:

```
$cat >> new.txt
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

(1)     $uniq new.txt
        01      accounts        6213
        02      admin           5423
        03      marketing       6521
        04      personnel       2365
        05      production      9876
        06      sales           1006

(2)     $uniq -c new.txt
        2       01      accounts        6213
        1       02      admin           5423
        3       03      marketing       6521
        1       04      personnel       2365
        1       05      production      9876
        1       06      sales           1006

(3)     $uniq -u new.txt
        02      admin           5423
        04      personnel       2365
        05      production      9876
        06      sales           1006

(4)     $uniq -d new.txt
        01      accounts        6213
        03      marketing       6521
```

- : COMPARING FILES:-

You have seen UNIX commands that works with a single file at a time. However, often a user must compare two files and determine whether they are different, and if so, just what the differences are UNIX provides commands that can help:

The **cmp** command compares two files and then simple reports the character number and line number where they differ.

The **comm** command compares two files and the report in three columns.

The **diff** command compares two files and tells you exactly where the file differ and what you must do to make them agree.

❍cmp command

The simplest command for comparing two files, cmp, simply tells you whether the files are different or not, if they are different, it tells you where in the file it sptted the first difference, if you are use cmp with no option. The command's syntax is:

Syntex:

$cmp [options] file1 file2

options

-l Gives you more information. It displays the number of each character that is different (the first character in the files is number 1), and then prints the octal value f the ASCII code of that character. (You will probably not have any use for the octal value of a character until you become a shell programming expert).

-s Prints nothing but returns an appropriate result code (0 if there are no differences, 1 if there are one or more differences). This option is useful when you write shell scripts.

Examples:

$cat >na1
allen Christopher
babinchak david
best betty
bloom dennis
boelhower joseph
bose anita
cacossa ray
delucia joseph

$cat > na2
allen Christopher
babinchak David
best betty
boelhower joseph
bose
cacossa ray
delucia joseph

Note that the first difference between the two files is on the second lines. The D in David in the second file is the 29th character, counting all new line characters at the ends of lines.

$cmp –s na1 na2
$echo $?

output:

1

The variable ? is the shell variable that contains the result code of the last command, and $? Is its value. The value 1 on the last line indicates that cmp found at least one difference between the two files. If is there no difference found at the time of comparing two files then output will be 0.

⭘comm command (Finding What is Common)

The comm. Command compare two sorted files line by line. comm. prints lines that are common, and lines that are unique, to two input files. The two files must be sorted before comm. can be used. The filename – means the standard input. With no options, comm. produces three column output. Column one contains lines unique to files1, column two contains lines unique to file2, and column three contains lines common to both files.

Syntex:

$comm [options] [-help] [-version] file1 file2

The options -1, -2, and -3 suppress printing of the corresponding columns.

-help Print a usage message and exit with a nonzero status.

-version Print version information on standard output then exit.

⭘diff command (Converting One File to Other)

The diff command is much more powerful than the cmp command. It shows you the differences between two files by outputting the editing changes that you would need to make to convert one file to the other.

Syntex:

$diff [options] file1 file2

-b Ignores trailing blanks, and treats all other strings f blanks as equivalent to one another.
-i Ignores uppercase and lowercase distinctions.
-t Preserves indentation level of the original file by expanding tabs in the output.
-w Ignores all blanks (spaces and tabs).

First, let's look at the two files that show what diff does:

Let's apply diff to the files na1 and na2 (the files with which cmp was demonstrated):

Example:

$diff na1 na2

2c2
<babinchak david
-
>babinchak David
4d3
<bloom dennis
6c5
<bose anita
-
>bose

These editor commands are quite different from that diff printed before. The first four lines show

2c2
<babinchak david
-
>babinchak David

Which means that you can change the second line of file1 to match the second line of file 2 by executing the command, which means change line 2 of file 1 to line 2 f file2. Note that both the line from file1 prefaced with <-and the line from file2-prefaced with >- are displayed, separated by a line consisting of three dashes.

The next command sys to delete line 4 from file1 to bring it into agreement with file2 up to but not including line3 of file2. Finally, notice that there is another change command, 6c5, which says change line 6 of file 1 by replacing it with line 5 of file2.

You can use the –i option to tell diff to ignore the case of the characters, as follows:

$diff –i na1 na2
4d3
<bloom dennis
6c5
<bose anita
-
>bose

The –c option causes the differences to be printed in context, that is, the output displays several of the lines above and below a line in which diff finds a difference. Each difference is marked with one of the following:

An exclamation point (!) indicates that corresponding lines in the two files are similar but not the same.

A minus sign (-) means that the line is not in file2.

A plus sign (+) means that the line is in file2 but not in file1.

$diff –c na1 na2

Changing information in files using tr and sed

(1) tr (Translating Characters)

This command is used to translate character for any particular files. The tr filter manipulates individual character in a line.

Syntex:

$tr expression 1 expression2

Example 1

```
$cat  >> abc.txt
abcd   abcd   abcd   abcd   abcd
abcd   abcd   abcd   abcd   abcd
ctrl+d
$tr '[a-z]' '[A-Z]' < abc.txt
ABCD ABCD ABCD ABCD ABCD
ABCD ABCD ABCD ABCD ABCD
$
```

Example 2 with same file

```
$tr '[a|b|c]' '[o]' < abc.txt
oood   oood   oood   oood   oood
oood   oood   oood   oood   oood
$
```

Example 3 with same file

```
$tr 'abc' 'opq' < abc.txt
opqd   opqd   opqd   opqd   opqd
opqd   opqd   opqd   opqd   opqd
$
```

option –d Delete given character from the files.

Example 4

```
$tr –d '/a' < abc.txt
bcd    bcd    bcd    bcd    bcd
bcd    bcd    bcd    bcd    bcd
$
```

sed (Stream editor)

sed is a multipurpose tool which combines the work of several filters. It is derived from ed. The sed performs non interactive operations on a data stream hence its name. It uses very few options but has a host of feature that allow you to select lines and run instructions on them. Learning sed will prepare you well for perl which uses many of these feature.

sed uses instructions to act n text. An instructin combines an address for selecting lines, with an actions to be taken on them, as shown by the syntax:

Syntex:

sed [option] 'address action' file name

The address and action are enclosed within single quotes. Addressing in sed is done in two ways

By one or two line numbers (like 3,7)
By specifying a /- enclosed pattern which occurs in a line (like /From:/)

Common files

$cat >> abc.txt

```
1.   abcd   abcd   abcd   abcd   abcd
2.   abcd   abcd   abcd   abcd   abcd
3.   abcd   abcd   abcd   abcd   abcd
4.   abcd   abcd   abcd   abcd   abcd
5.   abcd   abcd   abcd   abcd   abcd
6.   abcd   abcd   abcd   abcd   abcd
```

Example1 Line addressing

```
$sed '3q' abc.txt
1.      abcd    abcd    abcd    abcd    abcd
2.      abcd    abcd    abcd    abcd    abcd
3.      abcd    abcd    abcd    abcd    abcd
$
```

Example2 Print line from anywhere

```
$sed –n '5,6p' abc.txt
5.      abcd    abcd    abcd    abcd    abcd
6.      abcd    abcd    abcd    abcd    abcd
$
```

Example3 Print line from anywhere (Negative Action)

```
$sed –n '4,$!p' abc.txt
1.      abcd    abcd    abcd    abcd    abcd
2.      abcd    abcd    abcd    abcd    abcd
3.      abcd    abcd    abcd    abcd    abcd
$
```

Example4 Using multiple instructions

```
$sed –n –e '1p' –e '3p' abc.txt
1.      abcd    abcd    abcd    abcd    abcd
3.      abcd    abcd    abcd    abcd    abcd
$
```

Displaying data in octal

od command:

Many files (especially executables) contain nonprinting characters, and most UNIX commands don't display them properly. The file od file contains some of these characters that don't show up normally. More about od command

```
$cat > odfile
White space includes a
The ctrl+G character rings a bell
The ctrl+L character skips a page
ctrl+d
```

The apparently incomplete first line actually contains a tab. The od command makes these commands visible by displaying the ASCII octal value of its input. The –b option displays this value for each character separately. Here's a trimmed output:

$od –b odfile

```
0000000  127  150  151  164  145  040  163  160  141  143  145  040  151  156  143  154
0000020  165  144  145  163  040  141  040  011  012  124  150  145  040  007  040  143 etc…
```

Each line displays 16 bytes of data in octal, preceded by the offset (position) in the file of the first byte in the line. In the absence of proper mapping it's difficult to make sense out of this output, but when the –b and –c option are combined, the output is friendlier:

$od –bc odfile

```
0000000  127  150  151  164  145  040  163  160  141  143  145  040  151  156  143  154
           W    h    i    t    e         s    p    a    c    e         i    n    c    l
0000020  165  144  145  163  040  141  040  011  012  124  150  145  040  007  040  143
           u    d    e    s         a         \t   \n   T    h    e         007       c
```

etc……

Each line is now replaced with tow. The octal representations are shown in the first line. The printable character and escape sequences are shown in the second line. The first character in the first line is the letter W having the octal value 127. You will recall having used some of the escape sequences with echo. Let's have a look at their various representations.

The tab character Ctrl-i is shown as \t and the octal value 011.
The ball character Ctrl-g is shown as 007. Some systems show it as \a.
The formfeed character, Ctrl-l is shown as \f and 014.
The LF character Ctrl-j is shown as \n and 012. Note that od makes the newline character visible too.

Tools for mathematical calculations

- **bc command**

This command is used to have calculator at prompt.

Syntax

```
$bc
-
```

Examples

To do simple calculation

```
$bc
5+6
11
Ctrl+d
```

To do multiple calculation

```
$bc
5+6;5*2
11
10
```

To have decimal point

```
$bc
scale=2
5/2
2.50
```

To have binary to decimal

```
$bc
ibase=2
1111
15
```

To have decimal to binary

```
$bc
obase=2
15
1111
```

- **factor command**

When factor command is invoked without an argument, it waits for a number to be typed in. If you type in a positive number less then 2^46. It will factories the number and print its prime factors; each one is printed the proper number of times. Then it waits for another number. It exits if it encounters a zero or any non-numeric character.

Example

```
$factor
15
        3
        5

28
        2
        2
        7
q               to quit
```

$

If factor is invoked with an argument, it factors the number as above and then exits.

Monitoring input and output command

- **tee**

tee is an external command and not a feature of the shell. It handles a character stream by duplicating its input. It saves one copy in a file and writes the other to standard output. tee can be placed anywhere in a pipeline. It creates new file and stored enter data and also give the output at a time to standard output.

Syntax

$tee [other command] filename

Other command

User can place other command using pipeline.

Example 1

Creating a.txt file using tee command.

```
                        $tee abc
                        abc abc abc
Output Displayed to next row   abc abc abc
                        xyz xyz xyz
Output Displayed to next row   xyz xyz xyz
                        Ctrl + d
```

Example 2

Using pipeline

$who | tee user.txt

Username Terminal No. Date and Time

Same information will be stored in user.txt file.

● **script (Recording your session)**

To store your login session or any particular session script command is used.

Syntax:

$script File Name

Example:

$script abc

script started, output file is abc

$ls
bbb xyz.txt
$wc bbb
1 1 1 bbb
$exit
script done, output file is xyz

$cat
$ls
bbb xyz.txt
$wc bbb
1 1 1 bbb
$exit
script done, Date and Time will be shown here

Tools for displaying date and time

● **cal**

cal is a handy tools that you can invoke any time to see the calendar of any specific moth, or a complete year. To see the calendar for the month of July 2009, provide the month number and year as the two arguments to cal:

Syntax
$cal [Month Number Year]

Month Number Year

This option will show you the calendar for the particular month of particular year. To see the Aug 2009 calendar then following command will be invoked.

$cal 8 2009
Above command will show you the calendar for the month of august 2009.

User can see also all year calendar for the particular year for that user has to write year number after cal command.

Example
$cal 2009

Above command will show you 12month calendar for the year of 2009.

With cal, you can produce the calendar for any month or year between the year 1 and 9999. This should serve our requirements.

- **date (Displaying date and time)**

Unix has a date command that shows the date and time in the form used on the Internet.

Syntax:

$date
Fri Aug 19 13:28:34 IST 2005
$_

date is a valid command, and it displays both the date and time. Notice another security feature of UNIX; the command doesn't prompt you to change either the date or time. This facility is available only to the administrator, and the strange thing is that he uses the same command.

- : COMMUNICATION RELATED COMMANDS: -

- **telnet (Remote Login)**

The telnet command will allow you to log on to a remote machine. If yu have an account on a host in a local network (or on the Internet), you can use telnet with the hostname or IP address as argument:

Syntax

$telnet IP Address

Example

$telnet 192.168.35.12….
Trying 192.168.35.12.
Connected to 192.168.35.12.
Escape character is '^]'.

SunOS 5.8
Login:

You now have to enter your login name at this prompt, and then the password to gain access to the remote machine. As long as you are logged in, anything you type is sent to the remote machine, and your machine just acts as a dumb terminal. Any files you use or command that you run will always be on the remote machine. After you have finished, you can press ctrl+d or type exit to logout and return to your local shell.

The escape character lets you make a temporary escape t the telnet> prompt so you can execute a command on your local machine. To invoke it, press [Ctrl-]]. YYou can then use the ! with a UNIX command, say ls to list files on the local machine:

$ctrl+]
telnet>!ls –l

A telnet session is closed in the same way as any login session. Use [Ctrl + d] or the command appropriate to the shell. If you are at the telnet> prompt, use quit, an internal command of telnet.

- **wall (Communicating with users)**

The wall command addresses all users simultaneously. Most UNIX systems don't permit users to run this command and reserve it for the sole use of the administrator.

#wall
The machine will be shut down today
At 14:30 hrs. The backup will be at 13:30 hrs
Ctrl+d

All users currently logged in will receive this message on their terminal. This command is routinely executed b the administrator—especially before shutdown of the system.

- **mtod (For message of the day)**

The 'message of the day' like the news, is typed by the superuser in the file /etc/mtod. In case of the news command it is entirely the user's discretion whether to read the news or not. As against this the user doesn't have a choice whether to read the message of the day or not. Reading the message of the day is mandatory, as it is displayed as soon as you login. As soon as you login a file /etc/profile file gets executed. This file contains a command cat /etc/motd. As a result the contents of /etc/motd are bound to get displayed on the screen. When I logged in I found the following message on my screen.

The air-conditioning system needs a maintenance check! Hence the system would be down between 2.00 to 5.00 PM. See you at 5.00

The /etc/profile is like our good old AUTOEXEC.BAT of DOS. It gets executed every time the user logs in. In addition every user has his own .profile file in his home directory using which he can customize his working environment. The sequence of executing of these files is /etc/profile followed by .profile.

As you must have guessed it is only the superuser who can change the contents of the file /etc/mtod.

- **Write (Communicating with users)**

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

Syntax

$write user name

Example

$write user2
Hey there! I am back to the office
Ctrl+d

On executing this command the message would be relayed to the user whose login name is user2. He would hear a beep on his terminal, followed by the message:

Message from user1 on unix (tt3a) [Thu Oct 15 17:13:58]….
Hey there! I am back to the office
(end of message)

There are two prerequisites for a smooth write operation:

1) The recipient must be logged in, else an error message is ineitable.
2) The recipient must have give permission for messages to reach his or her terminal. This is done by saying the $ prompt.

$mesg –y

If you are expecting nothing of consequence and do not wish to be disturbed by social trivia like the one we just saw, you can deny write permission to your terminal by saying.

$mesg –n

A superuser however can write to any terminal, irrespective of whether mesg has been set –y or –n.

- **mail**

Using mail you can quickly and efficiently circulate memos and other written information to your co-workers, including directions for the out of town party this Saturday and the latest meanderings. You can even send and receive mail from people outside your organization, if you and they use networked computers.

write command required that the user to whom the message is to be sent not only to be logged in, but also open to messages. Unlike this, mail can be sent to users who have logged in currently or even to users who haven't logged in currently. In case the user has logged in at several terminals the moment mail is sent to this user it becomes available at all the terminals.

Sending mail

Syntax

```
$mail   user name
subject:-Unix Course
Body of the message
Ctrl+d
```

Sending mail more then one user

Syntax

```
$mail   user name1  user name2  user name3
subject:-Unix Course
Body of the message
Ctrl+d
```

Sending mail from the file

Syntax

```
$mail   user name1  user name2  user name3 < abc.txt
```

Handling Incoming Mail

The incoming mail received by a user is stored in a mailbox. Each user is given a mailbox whose name is same as the log name of the user. For example, for user aa1 the filename would be aa1 and it would be present in /user/spool/mail directory. All mail received from different

sources is appended to this mailbox file. However while viewing mail the messages are shown separately as if they are being read from separate files.

To read the mail that has been received we simply say mail at the shell prompt.

```
$mail
mail 1  User Name     Day and Date  Time   Subject
mail 2  User Name     Day and Date  Time   Subject
mail 3  User Name     Day and Date  Time   Subject
&
```

& displayed at the bottom. This is known as the mail prompt. We can issue several commands at this prompt. If you want to know which, you can type a? and obtain help on these commands.

```
&2
Mail 2 will be displayed
```

&q to quit from the mail box.

- **news**

The system administrator is the sole person who can make news under the UNIX OS. He types the information which he wants everyone on the network to know of in different files in /user/news directory. Whenever we log in if any fresh news has come in since we logged out last time then a message is displayed on our terminal saying.

```
News: rally tennis appeal
```

Where rally, tennis and appeal are the names of the files in which the news items are available.

We may either choose to ignore it, or if any news item interests us we may decide to peruse the same. To read the news, we have to say at the dollar prompt:

```
$news
rally
Content of the files will be displayed here

tennis
Content of the files will be displayed here

appeal
Content of the files will be displayed here
```

If you want to peruse only a particular item from the several items then you can say.

$news rally
rally
Content of the files will be displayed here

There are two more options available with the news command. The –n option only lists the names of the news from the /user/news directory that have not yet been read by you.

-s option which provides a count of the unread new items in the /user/news directory.

$news –s
5 news items.

If you try news –s after you have gone through all the news item it promptly displays the message 'No news'.

- **finger**

finger command is used to tell you which users are connected and which, if any, can receive messages. It displays a list of all those who have logged in and places a * next to those terminals where mesg is set to –n.

Syntax

$finger –i

Above command will displays following output

Login	TTY	When	Idle
ABC	*tty10	Fri Oct 13 17:25	8 minutes 12 seconds
XYZ	tty3	Fri Oct 13 17:21	49 Seconds
JKL	*tty4	Fri Oct 13 16:21	

- : PROCESS RELATED COMMANDS:-

- **ps (Process Status)**

Display status of current running processes. By default, it shows process ID, terminal with which process is associated, time consumed by the process since it has been started and process name (generally command name or program name).

Syntax

$ps [Option]

-f	Full listing showing the PPID (Parent Process ID) of each process
-e or A	All processes including user and system processes
-u	Process of user usr only
-l	Long listing showing memory-related information
-t	Processes running on terminal term, (say /dev/console)

Example

```
$ps
```

Simple ps command will display following output

PID	TT	TIME	COMMAND
36724	tty10	0	ksh.exe
37796	tty10	0	ps.exe

- **To run processes in background**

To run a process in the background, Unix provides the ampersand (&) symbol. While executing a command, if this symbol is placed at the end of the command then the command will be executed in the background. When you run a process in the background a number is displayed on the screen.

This number is nothing but the PID of the process that you have just executed in the background. Let's understand this with an example.

```
$sort employee.dat > emp.out &
17653
$
```

The task of sorting the file employee.dat and storing the output in emp.out has now been assigned to the background, letting the user free to carry out any other task in the foreground.

Advantages and limitation of background processes

(1) On terminal of background process no success or failure is reported on the screen. Then how do we keep track of it? That is where the PID displayed on the screen comes in handy. We can search for this PID in the output of ps to verify whether the process is still running or has been terminated.

(2) The output of a background process should always be redirected to a file. Otherwise you would get a garbled screen showing the output of the background process along with whatever you are doing in the foreground.

(3) With too many processes running in the background the overall system performance is likely to degrade.

(4) If you log out while some of your processes are running in the background all these processes would be abandoned halfway through. This is natural because all your processes are children/grandchildren/great grandchildren of your sh (shell) process. And when we log out the sh process dies along with all its children.

- **nice (changing the process priorities)**

Though all processes are equal, some processes are more equal than others. They can be made so by increasing their priority. The processes with higher priority would obviously get a time slot earlier and would be fired earlier than the other processes in the queue.

The priority of a process is decided by a number associated with it. This number is called 'nice' value of the process. Though paradoxical, higher the nice value of a process lower is its priority. The nice value of a process can range from 0 to 39, with 20 as the default nice value of a process. Thus, a process with a nice value 25 would execute slower then the one with a nice value 20.

Example 1

```
$nice cat employee.dat
```

This would increase the nice value of our cat process from 20 to 30 Since, we didn't specify the increment, an increment of 10 got assumed and the nice value got correspondingly incremented. If we so desire, we can specify the value of the increment.

```
$nice – 15 cat employee.dat
```

Now the cat process will have a nice value 35 (20+15). Note that the increment can range from 0 to 19. By incrementing the nice value we are putting cat on a lower priority and hence it would be executed slower than what it does normally.

```
$nice --15 cat employee.dat
```

Above command will reduce the nice value of a process but this kind operation can be done by only administrator. If any user wants to put his/her process with higher priority then he or she has to request to the administrator to put his or her process in higher priority.

Note that the priority of a process can be changed at the time of firing the process at command prompt. Once the process has been submitted to the process queue its priority cannot be changed.

User can see the nice values of the various process running in memory using ps –l command.

- **kill (Killing the process)**

When kill command is invoked, it sends a termination signal t the process being killed. A signal is a mechanism to communicate with a process. These signals have been given numbers. In the above example we have not communicated the signal number to the process to be killed. Hence, the default signal is same as the one generated when you hit the Del key in the middle of a ls or cat command. This default signal however is not very powerful and may not be able to kill a process at all times. A good example is the sh process which cannot be terminated by this default signal.

Syntax

$kill PID (Process ID)

To delete process forcefully then following syntax is used

$kill -9 PID

User can see the process id using ps cmmand.

Another important point. You can kill only your processes and not those fired by other users. Super user of course enjoys an altogether different status. He can kill any of the user's processes easily.

- **at**

This command is capable of executing UNIX commands at a future date and time. The UNIX command can be specified at the command prompt or can be stored in a file and the at command can use this file to execute the commands. Both these facilities are exemplified below.

```
$at 17:00
echo "It's 5 PM! Backup your files and logout"
Ctrl + d
Job 853158864.a at Wed Jun 14 17:00:00 IST 1996
```

At 5 PM in the evening you would see a message on your screen "It's 5 PM! Backup your files and logout".

- **batch**

Instead of we specifying that our commands be executed at a precise moment in time sometimes we may let the system decide the best time for executing our commands. The way to achieve this is through a command called batch. When we submit our jobs using this command, Unix executes our job when it is relatively free and the system load is light. Since

the time of execution of our commands is left for the system to decide we don't specify the time while executing the batch command.

```
$batch
echo "Hello Word"
Cltr + d
Job 692322435.b at Fri Jun 14 17:00:00 IST 1996
```

Once again note that the 'b' extension given to our job-id signifies that it has been submitted using the batch command.

- **Cron (Running Jobs Periodically)**

At and batch command are meant for one-time execution but corn executes programs at regular intervals. It is mostly dormant, but every minute it wakes up and looks in a control file (the crontab file) in /var/spool/cron/crontabs for instructions to be performed at that instant. After executing them, it goes back to sleep, only to wake up the next minute.

A user may also be permitted to place a crontab file after his/her login name in the crontab directory kumar has to place his crontab commands in the file /var/spool/cron/crontabs/kumar. This location is, however, system-dependent. A specimen entry in the file /var/spool/cron/crontabs/kumar can look like this:

```
00-10  17  *  3,6,9,12          5  find /  -newer       .last_time       -print   backuplist
```

The first field (legal value 00 to 59) specifies the number of minutes after the hour when the command is to be executed. The range 00-10 schedules execution every minute in the first 10 minutes of the hours. The second field (17, i.e. 5 p.m.) indicates the hours in 24 hour format for scheduling (legal value 1 to 24).

The third field (legal value 1 to 31) control the day of the month. This field (here, an asterisk), read with the other tow, implies that the command is to be executed every minute, for the first 10 minutes, starting at 5 p.m. ever day. The fourth field (3,6,9,12) specifies the month (legal values 1 to 12). The fifth field (5 – Friday) indicates the days of the week (legal values 0 to 6), Sunday having the value 0.

- **crontab**

User can also create your own crontab files with vi in the format shown below

```
00-10  17  *  3,6,9,12          5  find /  -newer       .last_time       -print   backuplist
```

The first field (legal value 00 to 59) specifies the number of minutes after the hour when the command is to be executed. The range 00-10 schedules execution every minute in the first 10

minutes of the hours. The second field (17, i.e. 5 p.m.) indicates the hours in 24 hour format for scheduling (legal value 1 to 24).

The third field (legal value 1 to 31) control the day of the month. This field (here, an asterisk), read with the other tow, implies that the command is to be executed every minute, for the first 10 minutes, starting at 5 p.m. ever day. The fourth field (3,6,9,12) specifies the month (legal values 1 to 12). The fifth field (5 – Friday) indicates the days of the week (legal values 0 to 6), Sunday having the value 0.

You will then need to use the crontab command to place the file in the directory containing crontab files for cron to read the file again:

crontab cron.txt cron.txt contains cron commands

If kumar runs this command, a file named kumar will be created in /var/spool/cron/crontabs containing the contents of cron.txt. In this way, different users can have crontab files named after their user-ids.

- **wait: (Waiting for process to complete)**

The wait command built into most shells (including all the shells discussed in this book) will wait for completion of all background processes or a specific background process. Usually, wait is used in scripts, but occasionally you may want to use it interactively to wait for a particularly important background job or to pause until all of your current background jobs complete so you will not load the system with your next job. The command will wait for a particular PID. If you are using a job control shell, you can use a job identified instead of a PID.

Example:

$job1 & job2 & wait job2 id (Process Id job1 [01] & job2 [02]

$job 1 & job2 & wait [02]

- **sleep**

Most interactive use of wait in can be replaced by notify. The notify command tells the shell not to wait until issuing a new prompt before telling you about the completion of all or some background jobs. The command notifies will tell shell to give asynchronous notification of job completion. The command notifies jobid will tell the shell to give asynchronous notification for a particular job.

Example

$sleel 30 & sleep 10 & notify %2

[2] Done Sleep 10
[1] Running Sleep 20
$

When you do this example, don't type anything after hitting return to enter notify %2. The notification appears as soon as job 2 finishes.

- : SHEELL PROGRAMMING:-

Shell Variables

- **System Variable**

PS2	:-	Secondary prompt string (default >)
PATH	:-	Search path for commands; multiple pathnames are separated with a colon (default /bin: /usr/bin)
HOME	:-	Default argument for the cd command; contains the pathname of the home directory.
LOGNAME	:-	Holds user's login name
MAIL	:-	Name of file to check for incoming mail
IFS	:-	Internal field separator (default space, tab or new line)
SHELL	:-	Pathname of the shell
TERM	:-	Specifies your terminal type
MAILCHECK	:-	Specifies how often to check for mail in $MAIL or $MAILPATH. If set to 0, mail is checked before each prompt (default 600 seconds).

User Variable

Set

The set statement displays all variables available in the current shell. This command will show the list of all variables and their expression.

Unset

To remove the definition of a function from the shell we can use the unset command. For example,

```
$unset HOME
```

Now user does't have HOME variable that stored the home directory path.

echo

This command is used to print something on the screen. This command is just like printf in c language. To print value of any variable using echo that variable should have $ followed by variable name.

read

This command is used to input a value of any variable at the time of running program. This command is just like scanf in c language.

Use of Logical Operator

Shell allows usage of three logical operations while performing a test. These are:

(a) –a (read as AND)
(b) –o (read as OR)
(c) ! (read as NOT)

The first two operators, -a and –o, allow two or more conditions to be combined
to be combined in a test. Let us see how they are used in a program Consider the following problem.

Relational Operator

-lt	Less then
-gt	Greater then
-le	Less then equal to
-ge	Greater then equal to
-eq	Equal to
-ne	Not equal to

If and Fi conditions

- if then fi

Like most programming languages, shell to uses the keyword if to implement the decision control instruction. The simplest form of the if statement looks like this:

Syntax:

```
if [ Condition ];then
        True Block
fi
```

If you are familiar with any high level language you must have used some form of an if statement. The UNIX shell if statement is different from others at least on one count. In most high level languages the if statements are usually concerned with the values of variables – is age greater than 20, is answer equal to yes or some such condition.

Example

```
echo "Enter a Number \c"
read N
if [ $N –eq 10 ]; then
echo "Enter number is 10"
fi
```

- if…then….else….fi

The if statement by itself will execute a single command, or a group of commands, when the exit status of the control command is 0. It does nothing when the exit status is 1. Can we execute one group of commands if the exits status is 0 and another group if the exit status is 1? of course. This is what is the purpose of the else statement, which is demonstrated in the following shell script.

Syntax:

```
if  [ Condition ];then
        True Block
else
        False Block
fi
```

Example

```
echo "Enter a Number \c"
read N
if [ $N –eq 10 ]; then
echo "Enter number is 10"
else
echo "Enter number is not 10"
fi
```

- if…then….elif….fi

Example

```
echo “Enter a Number \c”
read N
if [ $N –eq  0 ]; then
echo “Enter number is 0”
elif [ $N –gt 0 ];then
echo “Enter number is greater then 0”
else
echo “Enter number is less then 0”
fi
```

The Case Control Structure

Another way of controlling the sequence of execution is using the case control structure. Though if-else constructs can be nested, and abundance of conditions may make tracing the control in a program difficult. Hence, for a program where we are required to select from several alternatives a case control structure is the answer. The general form of the case control instruction is given below.

Syntax

```
case value in

choice 1)
        Statement block;;
choice 2)
        Statement block;;
choice )
        Statement block;;
*)
        Statement block;
case
```

Example

```
Echo “Enter a number from 1 to 3: \c”
read num
```

```
case $num in
        1)  echo you entered 1;;
        2)  echo you entered 2;;
        3)  echo you entered 3;;
        *)  echo you entered not 1,2,3;
esac
```

Looping in shell scripting

It is often the case in programming that you want to do something a fixed number of times. The while loop is ideally suited for such cases. Let us look at a simple example which uses a while loop.

Syntax

```
while [ Condition ]
do
        statement block
done
```

Example

```
count=1
while [ $count –le 3 ]
do
        echo $count
done
```

The program executes all commands between do and done 3 times. These commands form what is called the 'body' of the while loop. Immediately following the while is a control command. So long as the exit status of the control command is true, all commands within the body of the while loop keep getting executed repeatedly. To begin with, the variable count is initialized to 1 and every time the simple interest logic is executed the value of count is incremented by one. The variable count is many a times called either a 'loop counter' or an 'index variable'.

- **The Until Loop**

An until loop looks like this:

```
unitl [ Condition ]
do
        Statement Block
done
```

Example

```
count=1
while [ $count –ge 3 ]
do
        echo $count
done
```

The statements within the until loop keep on getting executed till the exit status of the control command remains false (1). When the exit status becomes ture (0), the control passes to the first command that follows the body of the until loop (i.e. the first command after done).

There is a minor difference between the working of while and until loops. The while loop executes till the exit status of the control command is true and terminates when the exit status becomes false. Unlike this the until loop executes till the exit status of the control command is false and terminates when this status becomes true. This difference is brought about more clearly by the following programs.

- **Using for with command line arguments**

The for loop is more frequently used as compared to the while and the until loops. Its working is also different than the other two loops. The for allows us to specify a list of values which the control variable in the loop can take. The loop is then executed for each value mentioned in the list.

Syntax

```
for variable in value1 value2 value3…
do
        Statement block
done
```

Here the for loop would execute the same sequence of commands for values mentioned in the list following the keyword in. Here is a simple example of the for loop.

Example

```
For $word in High on a hill was a lovely mountain
do
        echo $word
done
```

Output

```
High
on
a
hill
was
a
lovely
mountain
```