Unix Operating System

By Chetan Bhojani

BCA SEM-IV

Unix Operating System

Syllabus of UNIX/LINUX

Sr. No.	Topic	Subtopics
1	Starting with Unix	Login, logout, passwd, who, whoami mkdir, cd, rm, pwd cat, cp, mv, more
2	Understanding process and multi-user environment	ps, kill banner, cal, date chmod, chown, umask write, mesg, wall, mail
3	Text editing with vi	vi modes and switching in vi Cursor movements Screen control Entering and editing text (cut, copy paste) Pattern matching
4	Advance tools	Tee, head, tail Grep, egerp, fgrep
5	Working with the text files	Sort , uniq ,cmp,comm.,diff Cut,paste,join,tr
6.	Shell programming	Writing shell scripts,interactive shell scripts Executing shell scripts using sh command Setting of variable Arithmetic of shell using expr and let statement Decision statement-if,else,elif,test,case Loops-for,while,until,break,continue Sleep,wait,null Positional parameters($1,$#,$*,$?)

The beginning of Unix operating System

Unix had a modest beginning at AT&T Bell Laboratories in 1969. That was when AT&T withdrew its team from the MULTICS project, and some of them began work on the development of a computing environment to pursue “programming research”. Ken Thompson and Dennis Ritchie then designed and built a small multi-tasking system supporting two users. This new system had an elegant file system, a command interpreter and a set of utilities. A member of the team, Brian Karnighan, presumably having MULTICS in mine, suggested the name UNICS (Then jokingly reared to as the Uniplexed Information And Computer System) for this operating system. In 1970, UNICS finally became the name by which it is known today, UNIX.

Operation systems have always i.e. till UNIX came of the seen, been design with the particular machine in mind. Similarly computers were also built for a specific operation system. They were invariably returned in assembler, which understood and access hardware well, and had good speed. Thomson and Ritchie wanted a general operation system running on more type of hardware. That implied use of high-level language with consequent sacrifice of speed.

Originally written in assembler, UNIX was rewritten in 1973 in C, which was principally authored by Ritchie. Writing the system in C made it possible to maintain it and move it different hardware platforms.
As every vendor modified and enhanced UNIX to create its own version, the orignal UNIX lost its identity as a separate product. All these versions had little resemblance with one another, and each vendor claimed that its product was “true” UNIX. The BSD releases were greatly from the System V releases, and the incompatibilities sturdily mounted while standards were being developed as to what a product had to satisfy to be called UNIX, AT&T took it upon themselves to rework mainly the BSD product, and ultimately unify the system V3.2, BSD, SunOS and XENIX flavors into a single product. And that is how SVR4 came into being.

IBM, Helwet Packard and digital equipment corporation formed the association called Open Software Foundation (OSF), and was formatted to create an alternative UNIX like operating system. However they had their own brand; IBM offered AIX, HP offered HP-UX, while DEC produced Digital UNIX. And silicon graphics offered AIRIX, system that made Jurassic park possible. The UNIX corporative spirit has also produced Linux, the free UNIX that has captivated the computing words in resent years.

Features of Unix

Unix has a number of features, mostly well and some bad, bat it's necessary to know at least some of them. Many of its features have been borrowed from other operating systems, especially MULTICS. For the benefit of causal users, these features are discussed in the forthcoming sections.

- **Multi-user system:**

In Unix, the resources are actually shared between all users, be it the memory (RAM), the CPU the chip, or the hard disk. Multi-user technology is the great socialist who has time for everyone.

For creating the illusory effect, the computer breaks up a unit of time into several segments, and each user is allocated a segments. So at any point of time, the machine will be doing the job of a single user. The moment the allocated time, the machine will be doing the job of a single user. The moment the allocated time expires, the previous job is kept in abeyance, and the next user's job is taken up. This process goes on till the clock has turned full circle, and the first user's job is taken up once again. This is the kernel does several times in one second, and keeps all ignorant and happy.

- **Multitasking system**:

In Microsoft Window is considered to be a multitasking system where you can work with Word and Excel concurrently without quitting either of them.

In Unix, a single user can also run multiple tasks concurrently. You can switch jobs between background and foreground, suspend, or even kill them. This is done by running one job normally, and the others in the background.

- **Pattern Matching**:

It is the pattern-matching feature of Unix that makes it so attractive to the end user. The * is a special character used by the system to indicate that the expression can match a number of filenames.

In Unix the most advanced and useful tools also use a special expression called a regular expression that is framed with characters from this set.

- **The tool kit:**

Unix is a collection of tools, which lets you dispense with programming work for many applications. These applications range from such mundane tasks as copying or deleting a file, to somewhat sophisticated data base operation that cab be performed by using the tools in combinations. The power of filters is simply mind-boggling and by combining them, seemingly impossible takes can be made to look easy. It is most common to find a programmer writing a program in C for which a ready-made tool is available in the UNIX tool kit.

- **Programming Facility:**

Unix was designed for a programmer. A user who isn't a programmer will certainly find the transition quite overwhelming.

The Unix shell programming language has all the necessary ingredients like control structures, loops and variables as a programming language in its own right. Though the language is somewhat difficult to learn. The shell can be combined easily with the command and other programs. Unix programmers seldom take recourse to any other language for text manipulation problems.

- **System calls and libraries:**

Unix is written in C. Though there are a couple of hundred handling specialized functions, they all do that by using a handful of functions called system calls. These calls are built into the kernel, and all library functions and utilizes are written using them.

A user who wishes to write to a file, uses the write () system call without going into the details of programming that achieves this write operation. These system calls also use the same code to access devices and files. The open () system call is used to open both a device and an ordinary file.

- **Windowing systems:**

Unix had to come up with its own GUI. The X window system provides an environment that allows you to have multiple windows where you can run applications on each window individually. It replaces the command line with a screenful of objects in the form of menus, icons, buttons and dialogue boxes.

X window supports networked applications so that an applications so that an application may run on one machine and have its display on another. Thus, the applications must be portable across a large number of hardware that supports Unix. Much database front-end software use the X Window system to run forms and reports.

What Is a Command?

A UNIX command is a series of characters that you type. These characters consist of words that are separated by white space. White space is the result of typing one or more Space or Tab keys. The first word is the name of the command. The rest of the words are called the command's arguments. The arguments give the command information that it might need, or specify varying behavior of the command. To invoke a command, simply type the command name, followed by arguments (if any), to indicate to the shell that you are done typing and are ready for the command to be executed, press Enter.

The Types of UNIX Files

There are three types of UNIX files: regular files, directories, and device files. Regular files hold executable programs and data. Executable programs are the commands (such as cat) that you enter. Data is information that you store for later use. Such information can be virtually anything: a book that you are writing, a homework assignment, or a saved spreadsheet.
Directories are files that contain other files and subdirectories, just as a filing cabinet's drawers hold related folders. Directories help you organize your information by keeping closely related files in the same place so you can find them later. For instance, you might save all your spreadsheets in a single directory instead of mixing them
As in the cat example, files can also refer to computer hardware such as terminals and printers. These device files can also refer to tape and disk drives, CD-ROM players, modems, network interfaces, scanners, and any other piece of computer hardware. Under UNIX, even the computer's memory is a file.
Although UNIX treats all files similarly, some require slightly unique treatment. For example, UNIX treats directories especially in some ways. Also, because they refer directly to the computer's hardware, device files sometimes must be treated differently from ordinary files. For instance, most files have a definite size in bytes—the number of characters they contain. Your terminal's keyboard is a device file, but how many characters does it hold? The question of file size doesn't make sense in this case. Despite these differences, UNIX commands usually don't distinguish among the various types of files.

File and Directory Names

Unlike some operating systems, UNIX gives you great flexibility in how you name files and directories. As previously mentioned, you cannot use the slash character because it is the pathname separator and the name of the file tree's root directory . However, almost everything else is legal. Filenames can contain alphabetic (both upper- and lowercase), numeric, and punctuation characters, control characters, shell wild-card characters (such as *), and even spaces, tabs, and newlines. However, just because you can do something doesn't mean you should. Your life will be much simpler if you stick with upper- and lowercase alphabetic, digits, and punctuation characters such as ., -, and _.

Note : Using shell wild-card characters such as * in filenames can cause problems. Your shell expands such wild-card characters to match other files. Suppose that you create a filename * that you want to display with cat, and you still have your files cowboys and prufrock. You might think that the command cat * will do the trick, but remember that * is a shell wild card that matches anything. Your shell expands * to match the files *, cowboys, and prufrock, so cat displays all three. You can avoid this problem by quoting the asterisk with a backslash:

```
$ cat \*
```

Filenames can be as long as 255 characters in System V Release 4 UNIX. Unlike DOS, which uses the dot character to separate the filename from a three-character suffix, UNIX does not attach any intrinsic significance to dot—a file named lots.of.italian.recipes is as legal as lots-of-Italian-recipes. However, most UNIX users follow the dot-suffix conventions listed in Table Some language compilers like cc require that their input files follow these conventions, so the table labels these conventions as "Required" in the last column.

Table: Fileconventions.

Suffix	Program	Example	Required
.c	C program files	ls.c	Yes
.f	FORTRAN program files	math.f	Yes
.pl	Perl program files	Vsc.pl	No
.h	include files	term.h	No
.d, .dir	The file is a directory	recipes.d,	No
		recipes.dir	No
.gz	A file compressed with the	Vsc.gz	Yes
	GNV project's gzip		
.Z	A compressed file	term.h.Z	Yes
.zip	A file compressed with PKZIP	book. zip	Yes

Chapter: 1 Starting with Unix/Linux

Login and Logout

Since the Unix and Linux are multi-user operating systems, the users are given their own username by the system administrator.

The user has to enter the user name and password when the prompt appears on the screen.

login: vsc <enter>

password*:*

Where vsc is a user name. it is also known as userid or login name.
The string for password that you entered will not be displayed on the screen.
If you enter either the login name or password incorrectly, the system will display the following message.

login incorrect
wait for login retry
login:

After successful login system will show ***$*** as a prompt. If you are the system administrator and logged in as root, it will give the prompt **#**

In Linux the prompt will be

[root@localhost]#

Which shows that you are logged in as root and your machine is local host.

Logout is used to signoff a system as a current user.

passwd command

Passwd command is used to change passwords

```
$ passwd
old password:
New password:
Re-enter new password:
```

Passwd when invoked by ordinary user ,it asks for old password. After which it demands to enter new password twice. if you are logged on as root you can directly change the password.

In Linux

```
[root@localhost]passwd vsc
changing password for vsc
new password:
```

who command

who command gives the list of the users currently working on various terminals.

```
$who
vsc                 tty01      Dec 25  09:35
atmiya             tty02       Dec 25  08:50
student       tty03   Dec 25  09:45

$who –Hu

USER          LINE            LOGIN-TIME         IDLE         PID     COMMENTS

vsc           tty01           Dec 25 09:35   .            24
student tty02           Dec 25 09:40    0:30         25
```

-H is for header and –u option provides more detailed list
The idle column shows that student seems to be idling for last6 30 minutes. PID shows the process identification number.

who am i

The who command ,when used with the arguments “am” and “I”, displays a single line output only,i.e the login detail pertaining to the user who invoked this command.

```
$ who am i
vsc                 tty01           Dec 25 09:35
```

mkdir

mkdir command is used to create directories. The command is followed by the names of the directories to be created.

```
$ mkdir virani
```

This command creates the directories virani

```
$ mkdir stud1 stud2 stud3
```

This command will create three directories stud1,stud2,and stud3.

```
$ mkdir  virani virani/sci virani/arts
```

This creates three directories virani,two sub directories sci and arts under virani.The order of specifying arguments is important; you obviously can't create a sub directory before creation of its parent directory.

Sometimes the system refuses to create a directory:

```
$mkdir maths

mkdir : can't make directory maths
```

This can happen due to these reasons:
- The directory maths may already exist.
- There may be an ordinary file by that name in the current directory.
- The user doesn't have the permission to create a directory.

pwd command

In Unix, like a file, a user also occupies a certain slot of the file system. When log in, you are placed in a specific directory of the file system this directory is known as current directory. You can move around one directory to another but at any point of time only one directory is current.

At any time, you should also be able to know what your current directory is. The pwd (present working directory) tells your current directory.

```
$ pwd
/usr/virani
```

This shows the pathname. Which shows the location of current directory with reference to the root. Pwd tells you that you are placed in a directory virani, which has a parent directory usr, which is directly under root.

cd command

You can move around in the file system by using cd (change directory) command. It changes the current directory to the directory specified as the argument.

```
$ pwd
/usr/virani
$ cd student
$ pwd
/usr/virani/student
$ cd..
$ pwd
/usr/virani
```

cd.. takes you to the parent directory of the current directory
cd when used without arguments it takes to the home directory

```
$ pwd
/usr/virani/student
$ cd
$pwd
/usr
```

rmdir command

The rmdir (remove directory) command removes directories.
$ rmdir *student*

This command will remove the directory student
Like **mkdir**, **rmdir** can also delete more than one directory in at a time.

```
$ rmdir  virani virani/arts virani/sci

rmdir: virani:Directory not empty
```

The directory virani cannot be deleted because it has two sub directories *arts* and sci

- You can not delete a unless it is empty
- You cannot remove a sub directory unless you are placed in a directory, which is hierarchically above the one you have chosen to remove.

```
$cd arts

$ pwd
/user/virani/arts

$ rmdir arts

rmdir: arts: Directory does not exist
```

To remove the directory arts you must position you in the directory above *arts*,i.e.,*virani*

```
$cd /usr/virani
```

```
$pwd
/usr/virani
$rmdir arts
```

ls command

ls command is used to obtain the list of all the files in the current directory.

```
$ ls
04_emp
CASH.sh
abc
alpha
calendar
dept.lst
infile
install_log
program
```

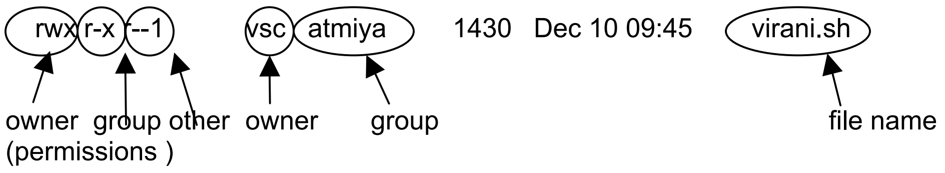
It provides the complete list of filenames in the current directory arranged in ASCII collating sequence (numbers first,uppercase and then lowercase),with one filename in each line.

ls command with options

- **ls –l command:**

```
$ ls –l
-rwxr-xr--     1     vsc     atmiya  1500    Dec 10 09:22 student
-rwxr-xr-x     1     vsc     atmiya  782     Dec 10 08:21 user.lst
drwxrwxrwx     1     vsc     atmiya  210     Dec 10 09:10 virani
-rwxrwxrwx     1     vsc     atmiya  180     Dec 10 09:15 lab.sh
```

ls –l command provides long listing with permissions.



r –read permission
w – write permission
e – execute permission
rwxr-x r-- means read, write and execute permission for file owner read and execute for group and read permission for others

- indicates an ordinary file
d indicates a directory

The third column indicates the file owner here vsc is the file owner
The fourth column shows the group owner here it is atmiya.

- **The –x option :**Output in multiple column

-x option with ls command in Sco UNIX produces a multicolumn output .Linux does this by default.

```
$ ls –x
01_page.html            CASH.sh             abc           alpha
calendar                dept.lst     infile        install_log
program
```

- **The –F option:** Identifying Directories and executables

To identify directories and executable files, the –F option should be used
It can be used with –x to produce the multi column output.

```
$ ls –Fx
01_page.html            CASH.sh*            abc/          alpha/
calendar*        dept.lst       infile        install_log
program/
```

* indicates that the file is executable, while **/** refers to a directory.

- **The –a option :** Showing hidden files
  ls doesn't how normally hidden file. –a option (all) list all the hidden files along with the other files.

  ```
  $ ls –axF
  ./               ../            .ext                 kshrc
  .profile .rhosts 01_page.html   CASH.sh*
  abc/             alpha/         calendar*            dept.lst
  infile           install_log    program/
  ```

  The files beginning with a **.** (dot) are all hidden file.

- **The –r option:** reversing the ort6 order
  You can reverse the order of the presentation with –r (reverse) option.

  ```
  $ ls –r
  program
  dept.lst
  install_log
  infile
  calendar
  alpha
  abc
  CASH.sh
  01_page.html
  ```

ls **command** (cont.)

- **The other options:**

Option	Description
-x	Displays multi column output
-F	Marks executables with * and directories with /
-r	Sort files in reverse order
-a	Shows all files including hidden files
-R	Recursive listing of all files in sub directories
-l	Long listing showing seven attributes of a file
-d	Forces listing of a directory
-t	Sort files by modification time
-u	Sorts files by access time (when used with –u option)
-i	Shows inode number of a file

cat command:

Displaying and creating files

cat command is used to display the contents of a file on the terminal.

```
$ cat user.lst
vsc
atmiya
student
lab1
lab2
account
mkt
```

```
$ cat first
This is my first Unix file
```

cat also accepts more than one filename as argument

```
$ cat  user1 user2
```

The contents of second file are shown immediately after the first without any header information. In this way ,**cat** can con**cat**enate two files. **cat** is normally used for displaying text files only. Executables ,when seen with **cat** ,will imply display junk. If you have non-printing ASCII characters in your input, you can use cat with –v option.

Using cat to create a file

Cat is also useful for creating file. Enter the command **cat** ,followed by the **>** and the file name.

```
$ cat > science
you can write your text here.

<ctrl-d>
$ _
$ cat science
 you can write your text here
```

cp command

copying a file

cp(copy) command copies a file or a group of files. It creates an exact image of the file on the disk with the different name. The syntax requires atleast666 two filenames to be specified in the command line. The first file is copied to the second.

```
$ cp user1 user2
```

If the destination file(user2) does not exist, It will first be created before copying. If not it will be overwritten without any warning from the system.

```
$ cp vi* virani
```
 It will copy all the files beginning with vi to directory virani
```
$ cp –i user1 user2
cp: overwrite user2? Y
```

The ***–i*** (interactive) option warns before overwriting the file.
Copying directory structure:
It is now possible to copy an entire directory structure with –r option .The following command copies all files and sub-directories in ***account*** to ***main***
$ cp –r account main

rm command:

Deleting files

Files can be deleted using **rm** (remove) command. It can delete more than one file with a single instruction.
$ rm user1 user2 user3
It deletes three files user1, user2, and user3.
You may remove all files from a directory with *
*$ rm user**
Interactive deletion: --

-i option makes the command ask the user for confirmation before removing each file
$rm –I user1
user1:? Y
A **Y** removes the file ,any other response leave the file undeleted.

mv command:

mv (move) is similar to the move command in dos. It has t6wo functions-renaming a file (or directory) and moving a group of file
to a different directory. mv doesn't create a copy of the file. No additional space is consumed in the disk during renaming. To rename the file ***user1*** to ***lab1***,

$mv user1 lab1
If the destination file doesn't exist, it will be created.

A group of files can be moved to a directory. The following command moves two files to directory ***dept.***
$ mv user1 user2 user3 dept
mv can also be used to rename a directory.

$ mv main dept

more command:

paging output

If a file is too large for its content to fit in one screen, it will scroll off your screen when you view it with **cat**. This sometimes happen so fast that, before you hit <ctrl-> to stop it, quite a bit of output would have scrolled off. **more** allows a user to view a file, one screen at a time.

$ more virani

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

```
Virani (44%)
```

more also works with multiple files

```
$more user virani main
```

You first see the contents of the first file, preceded by its name. After you have finished viewing file, more pauses with the message “virani: END (next file: main)”, before displaying the contents of the file ***main***
It sequentially displays all the files, pausing whenever change of file occurs.
In the middle, you can always switch to the next or previous file using..

:n	next file
:p	previous file

Chapter: 2 Understanding process and multi-user Environment

ps command:
Or Unix Processes:

A process is simply an instance of a running program. A process is said to be born when the program starts execution, and remains alive as long as the program is active. After the execution is complete, the process is said to die. It is the kernel that is ultimately responsible for the management of these processes. It determines the time and priorities that are allotted to processes. Typically, Hundreds and thousands of processes can run in the large system. Each process is uniquely identified by a number called PID (process identifier) that is allotted by the kernel when it is born.
When we login to a Unix system, a process is immediately set up by the kernel. This is a Unix command which may be *sh* (Bourn shell), *ksh* (Korn shell) or *bash* (Bourn again shell). Any command that we type in at the prompt is actually the standard input to the shell process. This process remains alive until you logout.
To know PID of the current shell

```
$ echo $$
543
```

Parent and child process:

Every process has a parent. The parent itself is another process, and a process born from it is said to be its child. When we run the command

```
$ cat students.txt
```

A process representing the cat command is started by the sh process(the shell)
The cat process remains alive as long as the command is active. sh is said to be the parent of cat, while cat is said to be the child of sh.

Ps command:

Ps command is used to display the process status or attributes of the status.
Ps shows the processes associated with a user at that terminal.

```
$ ps
PID     TTY          TIME          CMD
423     tty02        00:00:01      login
522     tty02        00:00:01      sh
569     tty02        00:00:01      ps
```

Each line shows the PID, the terminal (tty) with which the process is associated, the cumulative processor time (TIME) that has been consumed since the process is startedand the process name (CMD).

ps with –f option:

Ps –f will give the full information of the process including the process ID of the parent process.

```
$ ps –f

UID     PID     PPID    C       STIME           TTY     TIME            CMD
root    334     1       0       12:21:42        tty02   00:00:01        /home/login
vsc     435     213     2       12:25:33        tty02   00:00:01        sh
vsc     455     311     11      12:50:00        tty02   00:00:00        ps –f
```

PPID is the process ID of the parent process. UID displays the users login name. C indicates the amount of CPU time consumed by the process. STIME shows the time the process has started. TIME shows the total time consumed by the process.

ps with –u option:

ps with –u option lets you know the activities of any user. If we want to use it for the user *vsc*

```
$ ps –u vsc
```

ps with –a option :

This option will list out the processes of all users, but it will not display the system processes

```
$ ps –a
```

ps with –e option:

This option will also list out the system processes.

```
$ ps –e
```

Running the job in background:

A multi tasking operating system like Unix lets a user do more than one job at a time. Since there can only one job in the foreground, the rest of the jobs have to run in the background. To run the jobs in the background an operator **&** is used.

```
$ vi emp.txt &
370
```

The shell returns a number, which is the PID of the invoked command. The is returned and the shell is ready to accept another command.

banner :

The command creates posters by blowing up its arguments on the screen. On each line it can display a maximum of ten characters. There are no options.

```
$ banner 'virani science college'
```

cal :

With cal command we can print the calendars for a month or the entire year. Any calendar from the year 1 to 9999 can be displayed with this command.

```
$ cal jan 1999
```

or

```
$ cal 2002
```

date:

The Unix system maintains an internal clock which will run even if the system is down.A battery backup keeps the clock ticking.

date command displays the current date and time.

```
$date
Tue    Dec    3      09:12:33      IST    2002
```

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

We can print only the month using format +%m :

```
$ date +%m
12
```

or the month name

```
$ date +%h
Dec
```

we can combine them in one command:

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

There are other format specifiers:

d	day of the month(1 to 31)
y	last two digits of the year
H, M ,and S	hour, minute, and second

chmod:

There is an important default security feature provided by Unix in the sense that it write-protects a file from all except the owner of the file, though by default, all users have read access. This is easily brought out by creating a file test:

```
$ ls –l test
-rw-r- -r- -  1    abc      group     1906      sep   5  23:39     test
```

It seems that, by default, a file doesn't have executable permission. So how does one execute such a file? To do that, the permissions of the file need to be changed. This is done with the chmod (change mode) command.

Syntax :-

$ chmod category operations filename(s)

In category we have four options,

u - user
g -group
o -others
a -all.

In operation we have three option,

+ - assign permission
- - remove permission
= - assign absolute permission

In permission we have three option,

r - read permission -4
w - write permission -2
x – execute permission -1.

Eg:-

```
$chmod u+x test
$ls –l test
-rwx r-- r--.
```

chown:

ownership of file is feature often ignored by many users. You should be knowing by now that, when file is created, the creater becomes owner (representing the category u of chmod) of the files, and the group to which the user belongs, becomes to group owner.

You must remember at the outset that it is only the owner who can change the major file attributes, like it's permission or even ownership. A group owner or others can't change this attributes. However, when you copy a file owned by someone else, the ownership of the copy is transferred to you, and you can then manipulate the attributes the copy at will. This all can done by chown command.

```
$ls –l test
        -rwxr----x   1 kumar   it   1906     sep   5  23:39     test
$chown   guest  test
        -rwxr----x   1 guest   it   1906     sep   5  23:39     test
```

umask:

The default permissions inherited by all files and directory when they are created are rw-rw-rw- (octal value 666) for regular files rwxrwxrwx (777) for directories. The default is transformed by subtracting the user mask from it to remove one or more permission. This mask is evaluated by using umask without argument.

```
# umask
022
```

Subtracting this value from the file default yields 666 – 022=644. This represents the default permission that you normally see when you create regular file. You should make sure that the umask setting is proper. Two extreme instances are shown below,

```
    umask 666      #all permission off
  umask 222      #all read-write permission on
```

What permissions will be assigned to myfile and mydir by default, and how can you control those defaults? After all, you don't want to type a chmod command every time that you create a file or directory—it would be much more convenient if they were created with the modes that you most often want.

Your umask (user-mask) controls default file and directory permissions. The command umask sets a new umask for you if you're dissatisfied with the one that the system gives you when you log in. Many users include a umask command in their login start-up files (.profile or .login). To find out your current umask, just type umask:

```
$ umask

022
```

To change your umask, enter umask and a three digit number that specifies your new umask. For instance, to change your umask to 027, enter the following command:

```
$ umask 027
```

UNIX determines the default directory modes by subtracting your umask from the octal number 777. Therefore, if your umask is 027, your default directory mode is 750.

The result of this arithmetic is a mode specification like that which you give chmod, so the effect of using the umask command is similar to using a chmod command. However, umask never sets file execute bits, so you must turn them on with chmod, regardless of your umask. To find the corresponding file permissions, you subtract your umask from 666. For example, if your umask is 022, your default file modes will be 644.

Table shows some typical umasks and the default file and directory modes that result from them. Choose one that is appropriate for you and insert it into your login start-up file. Table shows file and directory modes both numerically and symbolically, and umask values range from the most to the least secure.

Umask Value	Default file Mode	Default directory Mode
077	600 (rw———-)	700 (rwx———)
067	600 (rw———-)	710 (rwx—x—-)
066	600 (rw———-)	711 (rwx—x—x)
027	640 (rw-r——-)	750 (rwxr-xr-x)
022	644 (rw-r—r—)	755 (rwxr-xr-x)
000	666 (rw-rw-rw-)	777 (rwxrwxrwx)

Pr:

The pr command prepares a file for printing by adding suitable header, footer and formatted text. It has many options, and some of them is quite useful. A simple invocation of the command is to use it with a file name as argument.

```
$ pr dept.lst
may  06 10:38 1997    dept.lst page 1
01:accounts:3254
02:admin:5423
…blank lines…
```

pr adds one lines of margin at the top and one at the bottom. The lower portion of the page has not been shown in the example for reasons of economy. The header shows the date and time of last modification of the files, along with the filename and page number.

lpstat:

lpstat has a number of option which can provides the status information of printers and jobs. The –r option shows whether lpsched is running:

```
# lpstat -r
scheduler is running.
```

Without options, it shows the status of all requests submitted by the user who executed the command. With the –p option, you can obtained the list of jobs lined up for a specific printer:

```
# lpstat –ppr1
pr1-303  test 34562 feb 9 13:24
```

Telnet:

Every unix vendor offers the telnet and rlogin utilities in its TCP/IP package to connect to a remote unix system. Telnet belongs to the DARPA command set, while rlogin is a member of the Berkeley set of r-utilities. You must have an account on the remote machine jill to use telnet with the machine's IP address as argument:

```
        $ telnet 192.168.2.1
                login:
```

you now have to enter your login name at this prompt, and then the password to gain access to the remote machine. After you have finished, you can press <ctrl-d>, or type exit to log out and return to your local shell.
When telnet is used without the address, the system display the telnet> prompt, from where you can use its internal commands.

```
        telnet>  ! ls –l *.c
```

lp command:

printing a file in Sco Unix

lp command is used for printing in Sco Unix ,Linux uses lpr command.

```
$ lp userlist
request id is laser1-140 (1 file)
```

The prompt is returned immediately after the job has been submitted. lp notifies the request id,a combination of the printer name (laser1) and the job number(140)
The file is not printed at the time of the command is invoked, but later depending upon the number of job already lined up in the
queue. Several users can print their files in this way without conflict.

lp accepts the above request because the administrator has defined a default printer. If it is not, or if the I more than one printer in the system, you have to use –d option with the printer name

$ lp –d laser1 userlist

-t option followed by the title string prints title on the first page:

$ lp –t "List of the users" userlist

You can notify the user with the mail (**-m**) option after the file has been printed. you can also print multiple copies with **–n** option.The following command prints three copies of ***userlist*** and mail message to the user.

$ lp –n3 –m userlist

canceling jobs

jobs are cancelled with cancel command

$ cancel laser1	#cancels current job on printer laser1
$ cancel laser1-140	# cancels job with request id (laser1-140)

In command

Links

When a file is copied ,both the original and copy occupy separate space in the disk. Unix allows a file to have more than one name, and yet maintain a single copy in the disk. Changes in one file will also be reflected in the other. The file is said to have more than one *links* .A file can have as many names as you want to give it, but the only thing that is common to all of them is that they all have the same inode number.

It is similar to *shortcuts* in windows, but there it is not easy to know how many shortcuts a file has. Moreover shortcuts themselves occupy a certain amount of disk space.
In Unix you can easily know from the fourth column of the *ls –l* listing.
Files are linked with **In**(link) command ,which takes two filenames as arguments. The following command links account with user

$ In account user

These links are called hard links. and have two limitations:
- You cannot have two linked files in two file systems.
- You cannot link a directory even within the same file system

To overcome this problem, symbolic links (also called soft links) were introduced. Linux uses symbolic links.
You can create a symbolic link with ***–s*** option of **ln**.

```
$ ln –s note note.sym
$ ls –l
9948     -rw-r—r--         1  vsc   atmiya   44        Dec 10 12:22 note
9952     lrwxrwxrwx  1  vsc       atmiya   4          Dec 10 14:10 note.sym->note
```

You can identify symbolic link by the character **l** seen in the permission field. The two files have different sizes. The notation **->** suggests that note.sym just contains the pathname for the file note.

Unlike, hard links, symbolic links can span multiple file systems and can also link directories. But they are removed by ***rm*** because a soft link is an ordinary file even if it points to a directory.

The find Command

One of the wonderful things about UNIX is its unlimited path names. A directory can have a subdirectory that itself has a subdirectory, and so on. This provides great flexibility in organizing your data.

Unlimited path names have a drawback, though. To perform any operation on a file that is not in your current working directory, you must have its complete path name. Disk files are a lot like flashlights: You store them in what seem to be perfectly logical places, but when you need them again, you can't remember where you put them. Fortunately, UNIX has the find command.

The find command begins at a specified point on a directory tree and searches all lower branches for files that meet some criteria. Since find searches by path name, the search crosses file systems, including those residing on a network, unless you specifically instruct it otherwise. Once it finds a file, find can perform operations on it.

Suppose you have a file named urgent.todo, but you cannot remember the directory where you stored it. You can use the find command to locate the file.

```
$ find / -name urgent.todo -print

/usr/home/stuff/urgent.todo
```

The syntax of the find command is a little different, but the remainder of this section should clear up any questions.

The find command is different from most UNIX commands in that each of the argument expressions following the beginning path name is considered a Boolean expression. At any given stop along a branch, the entire expression is true—file found—if all of the expressions are true; or false—file not found—if any one of the expressions is false. In other words, a file is found only if all the search criteria are met. For example,

```
$ find /usr/home -user marsha -size +50
```

is true for every file beginning at /usr/home that is owned by Marsha and is larger than 50 blocks. It is not true for Marsha's files that are 50 or fewer blocks long, nor is it true for large files owned by someone else.

An important point to remember is that expressions are evaluated from left to right. Since the entire expression is false if any one expression is false, the program stops evaluating a file as soon as it fails to pass a test. In the previous example, a file that is not owned by Marsha is not evaluated for its size. If the order of the expressions is reversed, each file is evaluated first for size, and then for ownership.

Another unusual thing about the find command is that it has no natural output. In the previous example, find dutifully searches all the paths and finds all of Marsha's large files, but it takes no action. For the find command to be useful, you must specify an expression that causes an action to be taken. For example,

```
$ find /usr/home -user me -size +50 -print

/usr/home/stuff/bigfile

/usr/home/trash/bigfile.old
```

Hard and Symbolic Links

The ln (link) command creates both hard and symbolic links. When you refer to the file "prufrock" in the command cat prufrock, UNIX translates the filename into an internal name. Because UNIX uses a different representation for its internal bookkeeping, you can refer to files by more than one name. A hard link is an alternative name for a file. Suppose you have a file data and you use ln to make a hard link to it called data2:

```
$ ln data data2

$ ls

data  data2
```

The name data2 now refers to exactly the same internal file as data. If you edit data, the changes will be reflected in data2 and vice versa. Data2 is not a copy of data1 but a different name for the same file. Suppose that Karen enters:

```
$ ln memo1 memo2
```

Karen now has two filenames—memo1 and memo2—that refer to the same file. Since they refer to the same internal file, they are identical except for their names. If she removes memo1, memo2 remains because the underlying file that memo2 refers to is still there. UNIX removes the internal file only after you remove all of the filenames that refer to it, in this case both memo1 and memo2. You can now see that rather than saying that rm removes a file, it's more accurate to say that it removes the file's name from the file system. When the last name for a file is gone, UNIX removes the internal file.

What good are hard links? Sometimes people working together on projects share files. Suppose that you and Joe work on a report together and must edit the same file. You want changes that you make to be reflected in Joe's copy automatically, without having Joe copy the file anew each time you change it. You also want Joe's changes to be reflected in the copy. Instead of trying to synchronize two separate files, you can make a hard link to Joe's file. Changes he makes will be reflected in your version and vice versa because you both are working with the same file even though you use different names for it.

A symbolic link (also known as a symlink) allows you to create an alias for a file, a sort of signpost in the file system that points to the real file someplace else. Suppose that you frequently look through your friend Joe's Italian recipes, but you are tired of typing:

```
$ cat /home/joe/recipes/italian/pizza/quattro_stagione
```

You could copy his recipes to your home directory, but that would waste disk space and you would have to remember to check for new recipes and copy those as well. A better solution is to create a symbolic link in your home directory that points to Joe's directory. You use ln's -s option to create symbolic links:

```
$ cd

$ ln -s /home/joe/recipes/italian italian

$ ls italian

linguini  pasta_primavera
```

Your symbolic link italian now points to Joe's recipes, and you can conveniently look at them.

There are some important differences between hard and symbolic links. First, you can't make a hard link to a directory, as in the example above. Hard links cannot cross disk partitions, and you can't make a hard link to a file in a network file system. Symbolic links can do all of these jobs.

Hard links must refer to a real file, but symbolic links may point to a file or directory that doesn't exist. Suppose that you have a symbolic link to your colleague's file /home/ann/work/project4/memos/paper.ms and she removes it. Your symlink still points to it, but the file is gone. As a result, commands like ls and cat may print potentially confusing error messages:

```
$ ls
```

```
paper.ms

$ cat paper.ms

cat: paper.ms not found
```

Why can't cat find paper.ms when ls show that it's there? The confusion arises because ls is telling you that the symbolic link paper.ms is there, which it is. Cat looks for the real file—the one the symbolic link points to—and reports an error because Ann removed it.
A final difference is that permission modes on symbolic links are meaningless. UNIX uses the permissions of the file (to which the link points) to decide whether you can read, write, or execute the file. For example, if you don't have permission to cat a file, making a symlink to it won't help; you'll still get a permission denied message from cat.

Chown:

ownership of file is feature often ignored by many users. You should know by now that, when file is created, the creator becomes owner (representing the category u of chmod) of the files, and the group to which the user belongs, becomes to group owner.

You must remember at the outset that it is only the owner who can change the major file attributes, like it's permission or even ownership. A group owner or others can't change this attributes. However, when you copy a file owned by someone else, the ownership of the copy is transferred to you, and you can then manipulate the attributes the copy at will. This all can done by chown command.

```
$ls –l test
        -rwxr----x   1 kumar   it   1906     sep   5  23:39     test
$chown   guest  test
        -rwxr----x   1 guest   it   1906     sep   5  23:39     test
```

lpstat:

lpstat has a number of options which can provides the status information of printers and jobs. The –r option shows whether lpsched is running:

```
# lpstat -r
scheduler is running.
```

Without options, it shows the status of all requests submitted by the user who executed the command. With the –p option, you can obtained the list of jobs lined up for a specific printer:

```
# lpstat –ppr1
pr1-303  test 34562 feb 9 13:24
pr1-303  test   3455 feb 9 13:29
```

UNIX communication

In a multi-user system, it is often necessary for one user to know what the other is doing. Communication through the system seems quite natural and necessary.
UNIX communication is tend to use to voice certain views which better than either speech or paper.
Some of the UNIX tools are as under:

news:

It is used to display all files that have been created in /usr/news since the last time the command was executed, so it is also known as the bulletin board.

```
e.g.:  $ news
       No news
```

mesg:

$ mesg n - It prevents other people from writing to our terminal.
$ mesg y - It enables receipt of messages sent by others.

write:

It lets you have a two-way communication with any person who is currently logged in. One user writes a message and then waits for the reply from the other.

```
e.g.:  $ write abc
       Hello,
       How are you?
I am fine.
       Hope we will meet soon.
```

If abc is not logged in, the system responds with an error message:

```
Write: abc is not logged in
```

Note: For two users to communicate with each other, both must invoke the write command.

talk:

This command is an alternative to write. It forms the basis of chat feature. The advantage of talk over write is that you can easily differentiate between the sent and received messages. It splits the screen into two horizontal windows; one shows the received data and the other is used for transmission.

```
e.g.:  $ talk abc
       Hello,
       How are you?
I am fine.
```

Hope we will meet soon.

mail:

It is used for later viewing, and the user doesn't need to be logged in. Mail messages can be saved with or without headers (s & w), and can be replied (r) while viewing it. We can also forward (m) a received message to another destination. We can delete mail (d) from the mailbox and again undo deletions (u) before quitting mail.

Eg:
```
$ mail abc
Subject: Communication
Mail Command.
```

wall:

It facilitates the administrator to communicate with all the users constantly. It addresses all the users simultaneously.

Eg:
```
# wall
UNIX seminar is set on the next coming Saturday.
```

Chapter: 3 Text editing with vi

VI EDITOR

vi is a full screen text editor available with all the Unix systems. Linux generally uses **vim** (**vi** **im**proved).
Three modes in **vi**:
A **vi** session begin by invoking the command ***vi*** with or without a filename

```
$ vi myfile

~
~
~
~
~
~
```

You are presented with a full empty screen, each line beginning with a **~**(tilde). This indicates that they are non-existent lines. For text editing, vi uses 24 of the 25 lines that are normally available in the terminal. The last line is reserved for some commands. This line is also used by system to display messages. The file name appears in this line with the message *"myfile"[New file]*
When you open a file with vi, the cursor is positioned at the top left-hand corner of the screen. You are said to be in command mode from where you can pass commands to act on the text.
There are three modes in Unix :

- Command mode-where keys are used as command to act upon text.
- Input mode-where any key depressed is entered as text.
- **ex** mode – where ex mode commands can be entered in the last line of the screen to act on the text.

Input mode

By default you are command mode. First enter the following command:

```
:set showmode <enter>
```

This command displays the mode at the bottom line of the screen.

** In Linux showmode setting is not necessary .vim sets it to this by default.

Before you attempt to enter text into the file , you need to change the default command mode to an input mode. There are several methods of entering into this mode, depending upon the type of input you wish to key in, but in every case the mode is terminated by pressing <esc> key.

Insertion of text

The simplest type of input is the insertion of text. whether the file contains some text or not ,when **vi** is invoked, the cursor is always positioned at the first character of the first line. To inset the text at this position press *I*,

The character doesn't show up on the screen, but pressing this key changes the mode from command mode to input mode. Further key depression will result in text being entered and displayed on the screen.
The input mode is terminated by pressing the <Esc> key which takes you back into the command mode.

You started the insertion with ***i***, which put text at the left of the cursor position. If the ***i*** is invoked with the cursor positioned on existing text, text on it right will be shifted further without being overwritten.

```
The vi editor
       |
       |        i full-screen<Esc>
       v
The vi full-screen editor
```

Append

To append text to the right of the cursor position,

a #existing text will also be shifted right

After you have finished editing press <Esc>

I inserts text at the beginning of a line, while **A** appends text at the end of the line

```
The vi full screen editor
      |
      |        a, a link of ex<Esc>
      v
The vi full-screen editor, a link of ex
```

Opening a new line with o

You can also open a new line by positioning the cursor at any point in a line and pressing o

o

This inserts an empty line below the current line.

```
vi and ex are one and the same editor
        |
        |   oIt is due to William joy<Esc>
        v
vi and ex are one and the same editor
It is due to William joy
```

O opens a line above the current line.

Replacing text

Text is replaced with **r**, **R, s**, and **S.**
To replace one single character by another, use **r** followed by the character that replaces the one under the cursor. you can replace a single character only in this way.

```
this is vi full-screen editor
  |
  |     rT
  v
This is vi full-screen editor
```

To replace more than one character, use **R** followed by the text.
This replacement is restricted to the current line only.
The **s** replaces a single character with text irrespective of its length. **s** replaces the entire irrespective of the cursor position.
S replaces the entire line.

Save and quit

Save and exit can be done using ex mode.
To enter any command in this mode enter a **:** ,which appear at the lat line of the screen.
To save a file and remain in the editing mode use **w** (write)**:**

```
:w <Enter>

"myfile", 8lines, 231 characters
```

The message shows the file name along with the number of lines and characters saved. With **w** you can optionally specify a file name as well, then the contents separately written to another file.

:w virani <Enter>

To save and quit the editor use **x** ;

```
:x <Enter>

"myfile", 8lines ,   231 characters

$_
```

It is also possible to abort the editing process and quit the editing mode without saving the buffer:

```
:q <Enter>
$_
```

Save and exit commands:

Command	Action
:w	Saves file and remains in editing mode
:x	Saves file and quits editing mode
:wq	Saves file and quits editing mode
:w emp.txt	It's like save as option of windows
:q	Quits editing mode when no changes are made
:q!	Quits editing mode after abandoning changes
:n1,n2w new.txt	writes line n1 to n2 to file new.txt
:.w new.txt	writes current line to file new.txt
:$w new.txt	writes last line to new.txt
:sh	Escape to Unix shell

Repeating Text

Suppose we want to insert a series of 10 **vsc** in one line, so instead of using I and the entering 10 vsc, we can write,

10ivsc <Esc>

Command mode

Deletion with x

x command deletes the character under the cursor. Move the cursor to the character that need to be deleted and then press **x**

The character under the cursor gets deleted and the text on the right shifts left to fill the space.
4x deletes the current character as well as three characters from the right.
Deletion with dd:

Entire lines can be removed by command **dd,**

Cursor movement

h(or backspace)	Moves the cursor left
I (or spacebar)	Moves the cursor right
K	Moves the cursor up
J	Moves the cursor down

The word oriented navigation:

B	Moves back to beginning of word
E	Moves forward to end of word
W	Moves forward to beginning of word

A repeat factor can be used.4b takes the cursor four words back.
The beginning or end of line:
To move the first character of a line
0 or **|** **#30|** moves cursor to column 30

to move to the end of the line use **$**

paging and scrolling

<ctrl-f>	scrolls full page forward
<ctrl-b>	scrolls full page backward
<ctrl-d>	scrolls half page forward
<ctrl-u>	scrolls half page backward
<ctrl-l>	redraws the screen

You must be in command mode to use this control keys.

Moving between lines

If you want to the line number of your current cursor position ,press <ctrl-g>.

“script.sh” [modified] line 403 of 541 –74%--

In vim ,in Linux the current line number and column number appears automatically. If you know the line number you can use **G** to move cursor to a specific line number.
30G #goes to line number 30

End of file is reached by simply using **G**

Pattern Search

Lines containing a string can also be located by prefixing the string with / (front slash).To locate the first occurrence of the string “virani” use,

/virani<Enter>

The cursor will be positioned at the first character of the first occurrence of this pattern. ? searches in the reverse direction.

Repeating the last pattern search:
For repeating search in the direction of the previous search made with **/** or **?** use **n** and for repeating the search in the reverse direction, use **N**

Joining lines:

Lines can be joined with **J**

4j	#joins following three lines with the current line

- The last command can be repeated using **. (dot)**
- To undo last instruction use **u**
- In vim (Linux) multiple undo and redo is possible with repeat factor C<ctrl-r> means three redo and 3u means to undo last three actions.

Chapter: 4 Advance tools

tee:

Unix provides a feature by which you can save the standard output in a file, as well as display it on the terminal (or pipe it another process). It is made possible by the tee command. Technically, it is not a feature of the shell, but a unix command available in /bin.

Tee uses standard input and standard output, which means that it can be placed anywhere in a pipeline. The additional feature it possesses is that it breaks up the input into two components; one component is saved in a file, and the other is connected to the standard output.

You can use tee to save the output of the who command in a file, as well as display it:

```
$ who | tee user.lst
        kumar   ttyo1           may 18  08:33
        guest           ttyo2           may 18  09:33
        test            ttyo3           may 18  07:33
```

since tee uses standard output, you can pipe its output to another command, say wc:

```
$ who | tee user.lst | wc –l
                3
```

you can use the device name like /dev/tty as an argument to tee:

```
$ who | tee /dev/ tty | wc –l
        kumar   ttyo1           may 18  08:33
        guest           ttyo2           may 18  09:33
        test            ttyo3           may 18  07:33
                3
```

head:

Syntax: head - <number of lines> <file name>

Purpose: Head command is user to display the top most information of the files. If no option is used it will display the first ten records of the argument file. We can also give the count number to display the respective number of lines.

Example: head - 3 abc.txt

```
        ABC
        PQR
        XYZ
```

As shown here it will display first three line of this file.

tail:

Syntax: tail - < number of lines> <file name>
Purpose: As in the case of head it displays the top most lines in the case of tail command it displays end lines of the file. By default if no number is given it will display the last 10 lines of the file.
Example: tail – 3 abc.txt

```
ROPE
HOPE
SHOPE
```

As given it will display last three lines of the files.

grep:

Syntax: grep < field name > <file name >
Purpose: With the help of this command we can get the list of the particular records, which are similar to the condition given. Grep requires an expression to represent the pattern to be searched for followed by one or more filenames.
Example: grep " sales " emp.txt

```
1.   Chanchal  singhvi | sales
2.   Barun sengupta | sales
```

Syntax: grep –c
Purpose: With the help of this command we can get the number of the records present in the file as per the field given.
Example: grep –c " sales " emp.txt

```
emp.txt   2
```

Syntax: grep –n
Purpose: With the help of this command we can get the number of line on which the records exist and the record.
Example: grep –c " sales " emp.txt

```
3: Chanchal singhvi | sales
10: Barun sengupta | sales
```

Syntax: grep –v
Purpose: It will display the total number of lines containing the record.
Example: grep –v " sales " emp.txt
Syntax: grep –l
Purpose: It will display the file names only where the pattern has been found.
Example: grep –l "manager" *.lst

```
desig.lst
emp.lst
```

Syntax: grep –I
Purpose: This option is used to avoid case sensitiveness of the test and the pattern given as argument.
Example: grep –I "agarwal" emp.lst

Sudhir Agarwal

Syntax: grep –e
Purpose: With the help of this option we can get the list of the records, which are spelled in similar manner as the pattern with different order test.
Example: grep –e “agarwal”

Sudhir Agarwal
Anil aggarwal
V.K. agrawal

egrep: Extending grep

The egrep command extends the grep’s pattern matching capabilities. It offers all the options of grep,but it’s most useful feature is the facility to specify more than one pattern for search. Each pattern is separated by a pipe (|)
The regular expressions that we have used in grep can also be used as patterns in *egrep.* egrep uses some additional characters , not used by grep.

Expression	Significance
ch+	Matches one or more occurrence of character ch
ch?	Matches zero or more occurrence of character ch
exp1|exp2	Matches expression exp1 or exp2

The extended set includes two special characters- + and ?. The + matches one or more instances of the previous characters, while ? matches zero or one occurrence of the previous character.
a+ matches a,aa,aaa etc. while a? matches either nothing or a single a.
Serching for multiple patterns:
We can use egrep for multiple patterns
We can locate both “sengupta” and “dasgupta” from the file.

```
$ egrep ‘sengupta|dasgupta’ emp.lst
2365|barun sengupta |director|7800
1265|s.n.dasgupta|manager|6500
```

We can also use:

```
$ egrep ‘(sen|das)gupta’ emp.lst
2365|barun sengupta |director|7800
1265|s.n.dasgupta|manager|6500
```

The –f option:

we can take the patterns from the file with the help of –f option.

```
$ cat pat.txt
sales | personnel |HRD
```

The file must contain the patterns , suitably delimited(with pipes) in the same way as they are on the command line.

```
$ egrep –f pat.txt emp.lst
```

The command takes the expression from the file pat.txt

fgrep:

fgrep accepts multiple patterns both from the command line and a file, but unlike grep and egrep, doesn't accept regular expressions. So, if the pattern to search for is a simple string, or a group of them, fgrep is recommended.
Alternative patterns in fgrep are specified by separating one pattern from another by the newline character. This is unlike in egrep, which uses the | to delimit two expressions. You may either specify these patterns in the command line itself

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

Now you can use fgrep in the same way you used egrep:

```
        fgrep –f pat.lst emp.lst
```

Chapter: 5 Working with the text files

Sort:

Syntax: sort < filename>

Purpose: On giving this command it will display the list in the sorted form. Sorting starts with the first character of each line and proceeds to the next character only when the characters in two lines are identical. By default sort records a line starting from the beginning of the line. The sequence of ordering by the sort commands is

1) White space
2) Numerals
3) Uppercase letters
4) Lowercase letters

Example: sort shortlist

There are two more options for using the sort with different arguments

Sort –t "|" +<no of field to be left> <file name>

This command will display the list in the sorting order of the second field i.e. the list will be sorted on the bases of the second field.

E.g.: sort –t "|" + 1 shortlist

Here +1 indicate the shorting should start after skipping the first field.

Sort –t "|" –r +<no of field to be left> <file name>

It will give the listing in the reverse format.

E.g.: sort –t "|" –r +1 shortlist

uniq:

unique command will remove the duplicate entries.

```
$ cat dept.data
01|account|1123
02|account|2341
03|admin|2131
04|account|1132
05|sales|4211
```

```
$ unique dept.data

01|account|1123
02|admin|2131
03|sales|4211
```

cmp: comparing two files

We may require to know whether the two files are identical or not. cmp(compare) command is used for the same.
Syntax:

```
        $ cmp <file1> <file2>
$ cmp note1 note2
```

The two files are compared byte by byte, and the location of the first mismatch is echoed to the screen.
cmp with –l option gives detailed list of the byte number and the differing bytes in octal for each character that differs in the both files:

```
$ cmp –l note1 note2

3       143     145             # Third character has the ASCII values 143 and 145
6       170     166
8       167     173
```

If the two files are identical, cmp will display no message but it will return the $ prompt.

comm:

comm. requires two sorted files and compares each line of the file with its corresponding line in the second.

$ cat note1
```
c.k.shukla
chanchal sanghvi
s.n.dasgupta
sumit chakrobarty
```

$ cat note2
```
barun sengupta
c.k.shukla
anil agrawal
lalit chowdhery
s.n.dasgupta
```

```
$ comm note1 note2

                barun sengupta
                        c.k.shukla
                anil agrawal
chanchal sanghvi
                lalit chowdhery
                        s.n.dasgupta
sumit chakrobarty
```

The first column contains two lines unique to the first file, while second column shows three lines unique to the second file. The third column displays two lines common to both the files.

diff:

Syntax: diff <file name 1> <file name 2>

Purpose: Diff command is used to display the differences present in the two files by comparing them. It also gives the information of line on which the non-identical data is present, which can become helpful to males the two files identical.
Diff use certain special symbols and instructions to indicate the changes that are required to make two files identical.

Example: diff list 1 list 2

Cut

cut can be used to extract specific columns from the file(such as name-the second field and the designation-the third field). The names start from column 6 through 32. Use cut with –c option for cutting columns.

```
$ cut –c 6-22,24-32 list
```

The –f option:

```
$ cut –d\| -f 2,3 list |tee cutlist
```

-d is used to specify a delimiter while –f is used for fields.

Paste

It pastes the files vertically.

```
$ paste list1     list2
```

By default paste uses the tab character for pasting files, but we can specify a delimiter with –d option.

```
$ paste –d\| list1 list2
```

Chapter: 6 Shell Programming

Introduction

You can do many things without having an extensive knowledge of how they actually work. For example, you can drive a car without understanding the physics of the internal combustion engine. A lack of knowledge of electronics doesn't prevent you from enjoying music from a CD player. You can use a UNIX computer without knowing what the shell is and how it works. However, you will get a lot more out of UNIX if you do.
Three shells are typically available on a UNIX system: Bourne, Korn, and C shells.

- What a shell is
- What a shell does for you
- How a shell relates to the overall system

The Kernel and the Shell

As the shell of a nut provides a protective covering for the kernel inside, a UNIX shell provides a protective outer covering. When you turn on, or "boot up," a UNIX-based computer, the program Unix is loaded into the computer's main memory, where it remains until you shut down the computer. This program, called the kernel, performs many low-level and system-level functions. The kernel is responsible for sending basic instructions to the computer's processor. The kernel is also responsible for running and scheduling processes and for carrying out all input and output. The kernel is the heart of a UNIX system. There is one and only one kernel.
As you might suspect from the critical nature of the kernel's responsibilities, the instructions to the kernel are complex and highly technical. To protect the user from the complexity of the kernel, and to protect the kernel from the shortcomings of the user, a protective shell is built around the kernel. The user makes requests to a shell, which interprets them, and passes them on to the kernel. The remainder of this section explains how this outer layer is built.
Once the kernel is loaded to memory, it is ready to carry out user requests. First, though, a user must log in and make a request. For a user to log in, however, the kernel must know who the user is and how to communicate with him. To do this, the kernel invokes two special programs, getty and login. For every user port—usually referred to as a tty—the kernel invokes the getty program. This process is called spawning. The getty program displays a login prompt and continuously monitors the communication port for any type of input that it assumes is a user name.

When getty receives any input, it calls the login program, as shown in Figure 10.2. The login program establishes the identity of the user and validates his right to log in. The login program checks the password file. If the user fails to enter a valid password, the port is returned to the control of a getty. If the user enters a valid password, login passes control by invoking the program name found in the user's entry in the password file. This program might be a word processor or a spreadsheet, but it usually is a more generic program called a shell.

three shells, developed independently, have become a standard part of UNIX. They are

- The Bourne shell, developed by Stephen Bourne
- The Korn shell, developed by David Korn
- The C shell, developed by Bill Joy

This variety of shells enables you to select the interface that best suits your needs or the one with which you are most familiar.

The Functions of a Shell

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

- Command line interpretation
- Program initiation
- Input-output redirection
- Pipeline connection
- Substitution of filenames
- Maintenance of variables
- Environment control
- Shell programming

1.**Command Line Interpretation**

When you log in, starting a special version of a shell called an interactive shell, you see a shell prompt, usually in the form of a dollar sign ($), a percent sign (%), or a pound sign (#). When you type a line of input at a shell prompt, the shell tries to interpret it. Input to a shell prompt is sometimes called a command line. The basic format of a command line is
Command arguments
Command is an executable UNIX command, program, utility, or shell program. The arguments are passed to the executable. Most UNIX utility programs expect arguments to take the following form:
options filenames
For example, in the command line

```
$ ls -l file1 file2
```

There are three arguments to ls, the first of which is an option, while the last two are file names. One of the things the shell does for the kernel is to eliminate unnecessary information. For a computer, one type of unnecessary information is white space; therefore, it is important to know what the shell does when it sees white space. White space consists of the space character, the horizontal tab, and the new line character. Consider this example:

```
$ echo part A     part B     part C

part A part B part C
```

Here, the shell has interpreted the command line as the echo command with six arguments and has removed the whitespace between the arguments. For example, if you were printing headings for a report and you wanted to keep the whitespace, you would have to enclose the data in quotation marks, as in

```
$ echo 'part A     part B     part C'

part A     part B     part C
```

The single quotation mark prevents the shell from looking inside the quotes. Now the shell interprets this line as the echo command with a single argument, which happens to be a string of characters including whitespace.

2.Program Initiation

When the shell finishes interpreting a command line, it initiates the execution of the requested program. The kernel actually executes it. To initiate program execution, the shell searches for the executable file in the directories specified in the PATH environment variable. When it finds the executable file, a subshell is started for the program to run. You should understand that the subshell can establish and manipulate its own environment without affecting the environment of its parent shell. For example, a subshell can change its working directory, but the working directory of the parent shell remains unchanged when the subshell is finished.

3.Input-output Redirection

It is the responsibility of the shell to make this happen. The shell does the redirection before it executes the program. Consider these two examples, which use the wc word count utility on a data file with 5 lines:

```
$ wc -l fivelines

5 fivelines

$ wc -l <fivelines

5
```

This is a subtle difference. In the first example, wc understands that it is to go out and find a file named fivelines and operate on it. Since wc knows the name of the file it displays it for the user. In the second example, wc sees only data, and does not know where it came from because the shell has done the work of locating and redirecting the data to wc, so wc cannot display the file name.

4.Pipeline Connection

Since pipeline connections are actually a special case of input-output redirection in which the standard output of one command is piped directly to the standard input of the next command, it follows that pipelining also happens before the program call is made. Consider this command line:

```
$ who | wc -l

5
```

In the second example, rather than displaying its output on your screen, the shell has directed the output of who directly to the input of wc.

5.Substitution of Filenames

Metacharacters can be used to reference more than one file in a command line. It is the responsibility of the shell to make this substitution. The shell makes this substitution before it executes the program. For example,

```
$ echo *

file1 file2 file3 file3x file4
```

Here, the asterisk is expanded to the five filenames, and it is passed to echo as five arguments. If you wanted to echo an asterisk, we would enclose it in quotation marks.

6.Maintenance of Variables

The shell is capable of maintaining variables. Variables are places where you can store data for later use. You assign a value to a variable with an equal (=) sign.

```
$ LOOKUP=/usr/mydir
```

Here, the shell establishes LOOKUP as a variable, and assigns it the value /usr/mydir. Later, you can use the value stored in LOOKUP in a command line by prefacing the variable name with a dollar sign ($). Consider these examples:

```
$ echo $LOOKUP

/usr/mydir

$ echo LOOKUP

LOOKUP
```

7.Environment Control

When the login program invokes your shell, it sets up your environment, which includes your home directory, the type of terminal you are using, and the path that will be searched for executable files. The environment is stored in variables called environmental variables. To change the environment, you simply change a value stored in an environmental variable. For example, to change the terminal type, you change the value in the TERM variable, as in

```
$ echo $TERM

vt100

$ TERM=ansi

$ echo $TERM

ansi
```

Shell programming

You've seen that the shell is used to interpret command lines, maintain variables, and execute programs. The shell also is a programming language. By combining commands and variable assignments with flow control and decision-making, you have a powerful programming tool.

Stephen Bourne at Bell Laboratories, where UNIX was originally developed, wrote the Bourne shell. Because it is found on most UNIX systems, many software developers work under the assumption that the Bourne shell is available on a UNIX system. This use does not mean that it is the best shell, but simply that it is the most common. Other shells, most notably the Korn shell, were written to enhance the Bourne shell, so shell programs written for Bourne run under the Korn shell. In some literature, the Bourne shell is called the UNIX system Version 7 shell.

The first exposure most people have to the Bourne shell is as an interactive shell. After logging on the system and seeing any messages from the system administrator, the user sees a shell prompt. For users other than the super-user, the default prompt for the interactive Bourne shell is a dollar sign ($). When you see the dollar sign ($), the interactive shell is ready to accept a line of input, which it interprets. When the super-user logs in, he or she sees the pound sign (#) as a prompt. It is a reminder that as super-user some of the built-in protections are not available and that extra care is necessary in this mode.

NOTE: On UNIX systems the super-user, also referred to as root, is without restriction. The super-user can write to any directory and can remove any file. File permissions do not apply to the super-user. The system administrator usually closely holds the password for the super-user.

Variables in Unix:

In Unix there are two types of variables.

1. System variables:
2. User variables:

1) System Variables:

The Unix system is controlled by a number of shell variables that are separately set by the system, some during the boot sequence, and some after logging in. These variables are called system variables or environment variables.

In system variables there are some types of it.

The path variable:

PATH is a variable that instructs the shell about the route it should follow to locate any executable command. If you wish to include the directory /home/henry/progs in your search list, you need to redefine this variable:

```
PATH=$PATH:/home/henry/progs
```

The HOME Variable:

When you log in, UNIX normally place you in a directory named after your login name. This directory is called the home or login directory, and is available in the variable HOME:

```
$ echo $HOME
/home/henry
```

The home directory for a user is set by the pertaining to her in the file /etc/passwd.

The IFS Variable:

IFS contains a string of characters that are used a s word separators in the command line. The string normally consists of the space, tab and the newline characters. All these characters are invisible, but the blank line following it suggests that the newline character is part of this string. You can confirm the contents of this variable by taking it octal dump:

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

The MAIL Variable:

MAIL determines where all incoming mail addressed to the user is to be stored. This directory is searched by the shell every time a user logs in. If any information is found here, the shell informs the user with the familiar message “You have mail”.

The SHELL Variable:

SHELL determines the type of shell that a user sees in logging in.
Today, there are a host of shells that accompany any UNIX system, and you can select the one you like the most. The C shell is known by the program csh, and the Korn shell by ksh. Linux offers bash, tcsh and pdksh; bash is the standard Linux shell.

If you wish to use any of these as the default shell, then the variable needs to be assigned accordingly. However, this assignment is made by the system after reading the last field in /etc/passwd. The system administrator usually sets up your login shell when creating a user account, through he can change it whenever you make a request.

Theoretically, all shell statements and UNIX commands can be entered in the command line itself. However, when a group of commands has to be executed regularly, they are better stored in a file. All such files are called shell scripts, shell programs, m shell procedures. There is absolutely no restriction on the extension such filenames should have, though it is a universal convention that the extension .sh be used for shell programs. The following shell script script.sh has a sequence of three UNIX commands:

```
$ cat script.sh
 # Sample shell script                    #Use # to comment lines
  date
  cal  'date "+%m 19%y"'
  echo 'Calendar for the current month shown above'
```

You can use the vi' editor to create this script. You can execute this file in two ways. One way is to use the sh command along with the filename: . sh script.sh Alternatively, you can first use chmod to make the file executable, and then run it by simply invoking the filename:

```
$ chmod +x script .sh
$ script .sh                                        # Don't need to use sh here
```

read: Interactive Scripts

The read statement is the shell's internal tool for taking input from the user, i.e., making scripts interactive. It is used with one or more variables, and input supplied through the standard input is read into these variables.
The script emp .sh uses read to take a search string and filename from the terminal: $ cat empi. sh

```
# Script: empi. sh - Interactive version
# The pattern and filename to be supplied by the user
echo "\nEnter-the pattern to be searched: \c"      # \n for a newline
read pname
echo "\nEnter the file to be used: \c"
read flname
echo "\nSearching for $pname from file $f1name\n"
grep "$pname" $f1name
echo "\nSe1ected records shown above\n"
```

If you are using the bash shell in Linux, you can't use the escape sequences \c and \n unless you use echo with the -e option also.
Single **read** statement can be used with one or more variables to let you enter multiple arguments:
read pname f l name

COMMAND LINE ARGUMENTS—POSITIONAL PARAMETERS

Shell procedures accept arguments in another situation too—when you specify them in the command line itself.

When arguments are specified with a shell procedure, they are assigned to certain special "variables", or rather ***positional parameters***.
The first argument is read by the shell into the parameter $1, the second argument into $2, and so on. You can't technically call them shell variables because all variables are evaluated with a $ before the variable name. In the case of $1, you really don't have a liable 1 that is evaluated with $.

```
$ cat emp2.sh
echo "Program: $0                                    # $0 contains the program name
The number of arguments specified is $#
The arguments are $*"
 grep "$1" $2
echo "\n job over"
```
The parameter $* stores the complete set of positional parameters as a single string. $# is set to the number of arguments specified. This lets you design scripts that check whether the right number of arguments have been entered. The command itself is stored in the parameter $0.
Invoke this script with the pattern "director" and the filename emp1.lst as the two arguments
$ emp2.sh director emp1.lst
When arguments are specified in this way, the first word (the command itself) is assigned to $0, the second word (the first argument) to $1, and the third word (the second argument) to $2. You can use more positional parameters in this way up to $9

Since the script accepts three arguments, how do you search for a multi-word pattern, say "barun sengupta" from emp2.1st? When the string is quoted, the shell understands it as a single argument:

```
$ emp2.sh "barun sengupta" emp2.lst
 Program : emp2.sh
 The number of arguments specified is 2
The arguments are barun sengupta emp2.lst
 2365| barun senauota | director I personnel |11/05/47|7800
```

if has been set at 2, as you can see above. If you had not used quotes, then the line would have been printed twice, first for "barun", and then for "sengupta".

The Parameter $?

The parameter $? stores the exit status of the last command. It has the value 0 if the command succeeds, and a non-zero value if it fails.
For example, if grep fails to find a pattern, the return value is 1, and if the file scanned is unreadable in the first place, the return value is 2. In any case, return values exceeding 0 are to be interpreted as failure of the command.
Try using grep in different ways:

```
$grep director emp.lst >/dev/null; echo $?
0
$ grep manager emp.lst >/dev/null; echo $?
```

```
1                                        # "manager" doesn't exist
```

```
$ grep manager emp2.lst >/dev/null; echo $?
grep: can't open emp2.lst
 2
```

You should note that all assignments to the positional and special parameters are automatically made by the shell.

To find *out whether a command executed successfully* or *not, simply use echo $? After the command. 0 indicates success, other values point to failure.*

Shell parameter	Significance
$1, $2 etc	The positional parameter
$*	Complete set of positional parameters as a single string
$#	Number of arguments specified in command line
$0	Name of executed command
$?	Exit status of last command
$!	PID of last background job

THE LOGICAL OPERATORS && AND | |—CONDITIONAL EXECUTION

There is no logic built into the script emp1.sh that can prevent display of the message "Selected records shown above" even if the pattern is not found. That is because the exit status of grep was not used to control the direction of the program. The shell provides two operators to control execution of a command depending on the success or failure of the previous command— the && and ||. The && operator is used by the shell in the same sense as it is used in C. It delimits two commands; the second command is executed only when the first succeeds.

You can use it with the **grep** command in this way:

```
$ grep 'director' empl.lst && echo "pattern found in file"
1006lchancha1 sanghvi |director |sales |03/09/38|6700
6521| lalit  chowdury |director |marketing |26/09/45|8200
 pattern found in file
```

The || operator is used to execute the command following it only when the previous command fails. If you "grep" a pattern from a file without success, you can notify the failure.

```
$ grep 'manager' emp2.lst  || echo "Pattern not found"
Pattern not found
```

THE if CONDITIONAL

Theif statement, like its counterpart in other programming languages, takes two-way decisions, depending on the fulfillment of a certain condition. In the shell, the statement uses the following forms, much like the one used in other languages:

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

Form1

```
if condition is true
 then
            execute commands
 fi
```

Form 2

If evaluates a condition that accompanies its "command line". If the condition is fulfilled (i.e., returns a true exit status), the sequence of commands following it is executed. Every if has must have a corresponding fi. The else statement (in Form 1), if present, specifies the action in case the condition is not fulfilled. This statement is not always required, as shown in Form 2.

What makes shell programming so powerful is that the condition can be a simple one like comparing two values, *or checking the return value of any UNIX program.* All UNIX commands return a value, as we saw with **cat** and grep. In the next example, **grep** is first executed, and if uses its return value to control the program flow:

```
 $ if grep "director" emp.lst
>then echo "Pattern found - Job Over
>else echo "Pattern not found"
>fi

 9876|jai Sharma|director |production|7000
2365|barun sengupta| director|personnel|7600
 1006lchancha1 sanghvi | director|sales|6500
6521|lalit chowdhery| director|marketing|8000
 Pattern found-job over
```

Every ***if*** must have an accompanying ***then*** and ***fi***, and optionally ***else***

THE case CONDITIONAL

The case statement is the second conditional offered by the shell. It doesn't have a parallel in most languages including perl. The statement matches an expression for more than one alternative, and uses a compact construct to permit multi-way branching. It also handles string tests, but in a more efficient manner than if. The general syntax of the case statement is as follows:

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

case matches the expression first for pattern1, and if successful, executes the commands associated with it. If it doesn't, then it falls through and matches pattern2, and so on. Each command list is terminated by a pair of semi-colons, and the entire construct is closed with esac (reverse of case). Consider a simple example using this construct. You can devise a script menu.sh, which accepts values from 1 to 5, and performs some action depending on the number keyed in.

```
$ cat menu.sh
echo "          MENU\n
1.  List of fi1es\n 2. Processes of user\n 3. Today's Date
 4. Users of system\n 5.Quit to UNIX\nEnter your option: \c"
read choice
case "$choice" in
 1)  ls –l ;;
 2)  ps -f ;;
 3)  date ;;
 4)   who ;;
 5)  exit                # ;; not really required for the last option
```

The five menu choices are displayed with a multi-line echo statement, case matches the value of the variable $choice for the strings 1, 2, 3, 4 and 5. It then relates each value to a command that has to be executed.

For example, if the user enters a 1, the ls -l command will be executed. Script termination is achieved by entering 5, or any other value not matched in the program logic. You can see today's date by choosing the third option: $ menu.sh

```
              MENU
1.  List of files
2.  Processes of user
3.  Today's Date
4.  Users of system
5.  Quit to UNIX
Enter your option: 3
 Sat Sep 13 16:41:16 1ST 1997
```

The *until* Loop

An until loop looks like this:

```
     until control command doesn't return true
     do
          this
          and this
          and this
          and this
  done
```

The statements within the until loop keep on getting executed till exit status of the control command remains false (1). When the exit status becomes true (0), the control passes to the first command 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 command is true and terminates when the exit status becomes false.
Unlike this the until loop executes till the exit status of the cont; command is false and terminates when this status becomes true.

* Prints numbers from 1 to 10 using until

```
until [$i-gt 10]
do
    echo $i
    i=expr$i+r
```

Did you notice the difference in the way the loops have been written in the two programs given above. The while loop continues its job till the test remains true and terminates when the test fails. Unlike this, the until loop continues till the test remains false and terminates when the test becomes true. Apart from this peculiarity the while and the until loops behave exactly identically.

Sleep, wait and null commands:

Sleep and Wait:- you may occasionally need to introduce some delay in a shell script to let the user see some message on the screen before the script starts doing something else. You may also like to make a check regularly(say, once a minute) for an event to occur(say, for a file to spring into existence). Sleep is a unix command that introduces this delay. It is used with an argument that specifies the number of seconds for which the shell will pause or sleep before it resume execution.

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

The message appears exactly 100 seconds after the commands have been invoked. The special feature of this command is that it doesn't incur significant overheads while it is sleeping.

Wait :- wait is a shell built-in that checks whether all background processes have been completed. This can be quite useful when you have run a job in the background, and now want to make sure the command is completed so that you can run yet another program. You can use the wait command to wait for the completion of the last background job with or without the process PIDs:

```
wait                #waits for completion of all background processes
wait 139       #waits for completion of process PID 139
```

Shell Scripts:

1. Make a simple calculator.
2. To display following details in report.
   (i) Basic Salary
   (ii) HRA (25%)
   (iii) DA (10%)
   (iv) Gross Salary = Basic Salary + HRA + DA
3. To accept number as command line and display sum of all digit.
4. To sort the content of the file.
5. To accept Invoice number & display other details (Inv-date, Inv-amount, Inv-party) of Invoice
6. Accept seat no and display other details (stu_name, tot_mark, class) of a marksheet.

7. Write a script that takes command line argument and reports on whether it is a directory, a file or something else.
8. Write a script that counts word on several files.
9. Display the information of a user (i.e. user id, group id,default directory etc.)When a username is entered at the command line
10. Take the filename as a command line argument and check whether it is a file or a directory as well as check for the read write and execute permissions.
11. If a character is entered through the keyboard. Find out if it is a vowel or a digit.
12. Create a menu as Follows:

```
                Selection Menu
        1.  List of Users currently logged in
        2.   Process Status
        3.  Disk Usage
        4.  Exit
        Enter your choice [1-4]:
```

Display the information as per the users selection

Shree M. & N. Virani Science College
And
Shree Gyanyagna College of Pure and Applied Science

B.Sc (I.T) Sem-III Preliminary Examination –Oct 2002

Operating System and Unix

Time: 3 Hours Total Marks: 100

Que: 1 Answer the following questions(Any Four) [20]

1. Inter process Communications
2. Functions of the Operating System
3. Features of Unix
4. Virtual Memory
5. Vi editor with different modes

Que:2 [A]Explain the following commands(Any Seven) [14]

1. rlogin
2. chmod
3. sort
4. lp
5. doscp
6. chown
7. ps
8. df
9. split

[B] Write a note on IO management [8]

[OR]

Write a note on process management

Que:3 Answer the following questions (Any three) [15]

1. Discuss Piping and redirection in Unix
2. Discuss the use of positional parameters in Unix
3. Explain the loop Structure in shell programming
4. Explain decision making statements in Unix

Que: 4 [A] write a note on (Any Four) [20]

1. Shell variables in Unix
2. Backup and Restore in Unix
3. Head and tail command
4. Mounting and un-mounting a file system in Unix
5. grep command

[B] Explain the term (Any three) [3]

1. Process
2. System call
3. Super user
4. Multitasking

Que: 5 write the shell scripts (Any Four) [20]

1. Display the information of a user (i.e. user id, group id,default directory etc.)When a username is entered at the command line
2. Take the filename as a command line argument and check whether it is a file or a directory as well as check for the read write and execute permissions.
3. If a character is entered through the keyboard. Find out if it is a vowel or a digit.
4. Create a menu as Follows:

   Selection Menu

   5. List of Users currently logged in
   6. Process Status
   7. Disk Usage
   8. Exit

   Enter your choice [1-4]:

Display the information as per the users selection

5. In a company an employee is paid as under

   If Basic salary is less than Rs. 5000 then HRA=10% of Basic salary
   DA =50% of BS
   If Basic Salary is equal or above Rs.5000 then HRA=Rs.1000,
   DA =60% of Basic salary
   If basic salary is inputted from the keyboard calculate the gross salary.
   Gross salary=Basic salary+HRA+DA