Government of Karnataka

Department of Technical Education

Bengaluru

GOVERNMENT POLYTECHNIC, KARWAR

LINUX LAB JOURNAL
15CS47P
2019 - 2020

CERTIFICATE

This is to certify that Shri / Miss ____________________of **IV** Semester Diploma in **Computer Science and Engineering** has conducted the practical in **Linux Lab (15CS47P)** satisfactorily during the year 2019 - 2020

Staff Member In charge with date **Head of the Section with date**

REGISTER NUMBER

Examiner's Signature: 1.

2.

Date: *PRINCIPAL*

Place: Karwar *Govt. Polytechnic, Karwar*

<u>Programme Outcomes (PO's)</u>

1. Apply basic knowledge of mathematics, science, and engineering as it applies to Computer Science & Engineering to solve engineering problems. ***(Basic Knowledge)***
2. Apply the concepts of software engineering to build system software and application software and have the ability to design and maintain network architecture. ***(Discipline Knowledge)***
3. Plan and develop systems to solve software and network engineering problems to derive the results through experiment and practice. ***(Experiment and Practice)***
4. Create and use the techniques, algorithms, models and processes, and modern software/hardware tools necessary for computer engineering practice. ***(Engineering Tools)***
5. Produce technical solutions in global and societal context and demonstrate the need for sustainable development. ***(The Engineer and Society)***
6. ***Understand*** professional and ethical responsibilities and act accordingly in all situations. ***(Environment and Sustainability)***
7. Inculcate professional and ethical responsibilities and marshal in all situations. ***(Ethics)***
8. Function effectively as an individual and as a team member, and in multi-disciplinary environment. ***(Individual and Team Work)***
9. Communicate and present ideas effectively. ***(Communication)***
10. Self-improvement through continuous professional development, and independent and lifelong learning in the context of technological changes. ***(Lifelong learning)***

Course Outcomes (CO's)

Course Outcome		Experiment linked	CL	Linked PO	Teaching Hrs
CO1	Demonstrate installation of Linux operating system and understand the importance of Linux.	***1***	***U***	2,3,4,8,10	**06**
CO2	Appraise various command usage of files and directories.	***2 to 4***	***U, A***	2,3,4,8,10	**12**
CO3	Show the working of **vi** editor in all its modes using various commands.	***5 to 8***	***U, A***	2,3,4,8,10	**12**
CO4	Manage shell and processes using various commands.	***9 to 12***	***U, A***	2,3,4,8,10	**12**
CO5	Write Shell scripts and C programs using **vi** editor.	***13 to 21***	***A***	2,3,4,8,10	**30**
CO6	Demonstrate Linux administration and its environment	***22***	***A***	2,3,4,8,10	**06**
			Total sessions		**78**

Legends: R = Remember U= Understand; A= Apply and above levels (Bloom's revised taxonomy)

Course-PO Attainment Matrix

Course	Programme Outcomes									
	1	2	3	4	5	6	7	8	9	10
Linux Lab	**-**	**3**	**3**	**3**	**-**	**-**	**-**	**3**	**-**	**3**

Level 3- Highly Addressed, Level 2-Moderately Addressed, Level 1-Low Addressed.

Method is to relate the level of PO with the number of hours devoted to the COs which address the given PO.
If ≥40% of classroom sessions addressing a particular PO, it is considered that PO is addressed at Level
3 If 25 to 40% of classroom sessions addressing a particular PO, it is considered that PO is addressed at
Level 2 If 5 to 25% of classroom sessions addressing a particular PO, it is considered that PO is
addressed at Level 1
If < 5% of classroom sessions addressing a particular PO, it is considered that PO is considered not-addressed.

Index

Scheme of Valuation for End Examination

1	Write the syntax of commands with examples from any one exercise from PART-A	10
2	Write any one program from PART-B	10
3	Execution of PART A commands and PART B Program with result	10+10
4	Viva voce	10
	Total	**50**

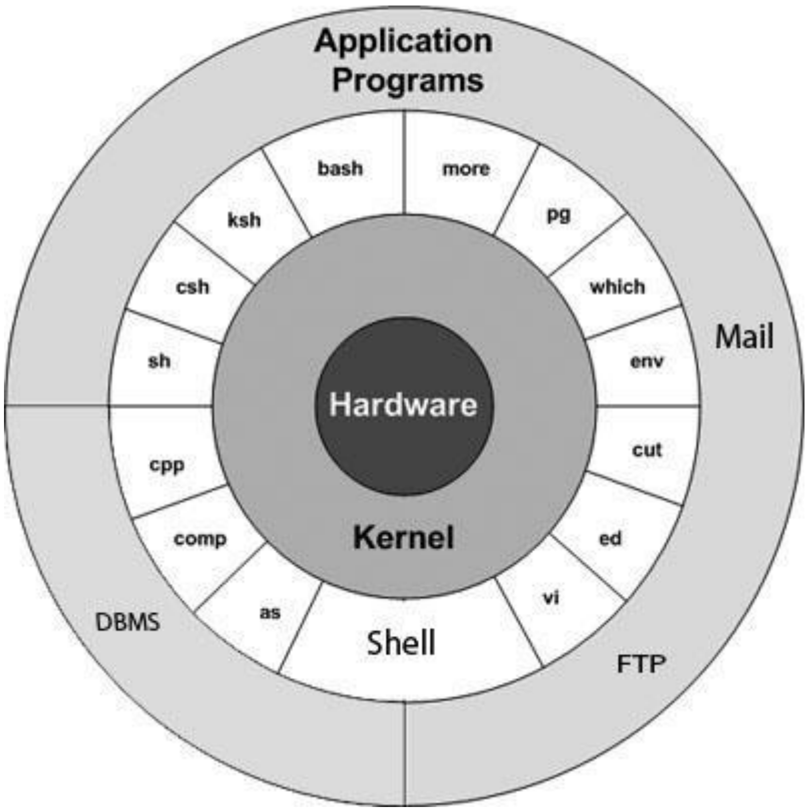
1. INTRODUCTION

UNIX is now more than 25 years old, but unfortunately the essentials have remained the same. A major part of the job learning UNIX is to master the essential command set. UNIX has a vast repertoire of commands that can solve many tasks either by working singly or in combination.

THE UNIX ARCHITECTURE

Division of labor: Kernel and Shell

The kernel interacts with the machine's hardware, and the Shell with the user. The relationship is depicted in the figure below



The Kernel is core of operating system-collection of routines mostly written in C. It is loaded into memory when the system is booted and communicates directly with the hardware. User programs (the applications) that need to access the hardware use the services of the kernel, which performs the job on user's behalf. These programs access the kernel through a set of system calls.

Apart from providing support to user programs, kernel manages the system's memory, schedules processes, decides their priorities and performs other tasks. It is also called the operating system-a program's gateway to the computer's resources.

Computers don't have any capability of translating commands into actions. That requires a **command interpreter**, a job that is handled by the outer part of the operating system-the **shell.** It is actually the interface between the user and kernel.

When you enter a command through the keyboard, the shell thoroughly examines the keyboard input for special characters .If it finds any, it rebuilds a simplified command line, and finally communicates with the kernel to see the command is executed.

The system bootstrap program loads the kernel into memory at startup. The shell is represented by **sh**(Bourne shell), **csh**(C shell), **ksh**(Korn shell) or **bash**(Bash shell).One of the shells will run to serve you when you log in.

FEATURES OF UNIX/LINUX

- A multiuser system
- A multitasking system
- The building block approach
- The toolkit
- Pattern matching
- Programming facilities.
- Documentation

LINUX INSTALLATION-STEPS FOR INSTALLING LINUX OPERATING SYSTEM

Install Virtual Box or any other equivalent s/w on the host OS:

Virtualization is the process of emulating hardware inside a virtual machine. This process of hardware emulation duplicates the physical architecture needed for the program or process to function. Virtualization allows us to create virtual version of something, such as an operating system, a server, a Storage device or network resources.

A host operating system (OS) is the original OS installed on a computer. Other operating systems are sometimes installed on a computer.

A guest OS is an operating system that is installed in a virtual machine or disk partition in addition to the host or main OS. In virtualization a single computer can run more than one OS at the same time. In a virtualization solution a guest OS can be different from the host OS.

VMware Workstation:

One of the first companies to develop a virtual product was VMware,www.vmware.com. VMware lets us to create and run a host of operating systems from one base system. We also gain the ability to drag and drop files into the virtual system and to fully configure the virtual OS. VMware Workstation even supports an option known as snapshots, which means we can set a base point to which we can easily return. To install VMware Workstation, we need to purchase a copy or download an evaluation copy. We need about 25MB to download and install VMware Workstation. On average, we will need 3GB to 8GiB for each virtual OS we install. Memory is another important issue. Although the documentation might state that a minimum of 128MB to 256MB of memory is needed, this typically won't be enough for anything more than a basic command-line install of Linux. Expect operating systems such as Windows to require much more. Insufficient memory will devastate performance on both the guest (VM) and host OS.

STEPS FOR INSTALLING BACKTRACK OS ON THE HOST OS

Step 1: Open VMware Workstation and click on create a new virtual machine.
Step 2: Select typical configuration and click on Next.
Step 3: Select Installer disc image file and click on Next.
Step 4: Give the guest OS as other and select version as other.
Step 5: Give VM name as Backtrack and select the path and Click on Next.
Step 6: Give max disk size as 15GB and select store virtual disk as a single file then
Click on Next
Step 7: Click on Finish button
Step 8: Click on power on this virtual machine

Step 9: Now the Backtrack will start. Give the command startx and press Enter
Step 10: Now select the language default as English and press forward
Step 11: Select the Country as INDIA and click forward
Step 12: Select the default keyboard layout as USA and click forward
Step 13: Select the erase the disk space and click forward
Step 14: Now click on Install and installation will begin
Step 15: Wait for backtrack to get installed in VM ware
Step 16: Click on Restart Now
Step 17: Enter login as root and toor and press Enter
Step 18: Login to backtrack by typing startx and press Enter

COMPARISON BETWEEN LINUX AND OTHER OPERATING SYSTEM

It is important to understand the differences between Linux and other operating system, such as MS-DOS, OS/2 and other implementations of UNIX for the personal computer. First of all, it should be made clear that Linux will coexist happily with other operating systems on the same machine: that is, you can run MS-DOS and OS/2 along with Linux on the same system without problems.

The Windows operating system does not provides the users access to the programming code that forms the basis for foundation of this OS. In Windows, we need to know where to find software and install it, by running executable files (.exe files) related to it. In Windows, if users want to change the desktop appearance, they have to pay and install a third party application. Windows provides a greater facility for its users by providing various drives like C: D: E: and so on. In windows, all the drives are not mounted on a single tree.

Linux OS belongs to the GNU Public License. It provides the access of code to the users of all categories. That code is basis for foundation of Linux OS. In Linux, we have a centralized location where we can search for, add or remove software co-related to packet management system. In case of Linux, users are free to make their desktop appearance in the way they desire. Users won't find "My Documents" on Ubuntu, nor will you find" Program Files" on Fedora. There is no C: or D: drives. But, there is only one single file tree and all our drives are mounted on the tree.

APPLICATION OF LINUX OPERATING SYSTEM:

- Web serving
- Networking
- Databases
- Desktops
- Scientific computing
- Home computing

2. INTERNAL AND EXTERNAL COMMANDS IN LINUX

Since ls is a program or a file having an independent existence in the /bin directory (or /usr/bin) ,it is branded as an external command. Most commands are external in nature, but there are some which are not really found anywhere, and some which are normally not executed even if they are in one of the directories specified by PATH. Take for instance the echo command:

```
#type echo
echo is a shell built-in
```

echo isn't an external command in the sense that, when you type echo, the system won't look in its PATH to locate it(even if it there in /bin).Rather, it will execute it from its own set of built-in commands that are not stored as separate files. These built-in commands, of which echo is a member, are known as **internal commands**.

Its shell that does all these work. This program starts running for you when you log in, and dies when you log out. The shell is an external command with a difference, it possesses its own set of internal commands. So if command exist both as internal command of the shell as well as an external one (in /bin or /usr/bin), the shell will accord the top priority to its own internal command of the same name.

INTERNAL COMMANDS IN LINUX:

echo : Display message

You can use echo command to display messages on the screen.

```
# echo "This is an example of the echo command"
This is an example of the echo command
```

The echo command displays text enclosed between " " on the screen. By default, the echo command displays the text and then places a newline character at the end of it. The newline character moves the cursor to the line after the text is displayed. To keep the cursor on the same line, you can use the –n option with the echo command.

```
# echo –n "This will keep the cursor on the same line"
```

Notice that the $ sign is displayed on the same line as the output.

type: Locating commands

The best and fastest way to know the location of the program that the system will execute is to use the type command:

```
# type ls
ls is /bin/ls
```

When you execute the ls command, the system fetches this file from the /bin directory and executes it.

EXTERNAL COMMANDS IN LINUX:

ls:Listing files

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

```
# ls
08_packets.html          #Numerals first
TOC.sh                   #Uppercase first
calendar                 #Then lowercase
cptodos.sh
dept.lst
emp.lst
helpdir
```

Since file containing the first three chapters have similar filenames, you can use special short-hand notation (*) to access them:

```
#ls chap*
Chap 01
Chap 02
Chap 03
```

cp: Copying a file
The cp (copy) command copies a file or a group of files. It creates an exact image of file on the disk with a different name. The syntax requires at least two filenames to be specified in the command line. When both are ordinary files, the first is copied to the second:

```
#cp chap01 unit1
```

mv: Renaming files
mv (move) has two functions---renaming a file or directory and moving a group of files to a different directory.mv doesn't create a copy of a file; it merely renames it. To rename the file chap01 to man01, you should use

```
#mv chap01 man01
```

rm: Deleting file
Files can be deleted with rm(remove).It can delete more than one file with a single instruction. The following command deletes the first three chapters of the book:

```
# rm chap01 chap02 chap03
```

cat: Displaying and creating files
cat is one of the most well-known commands of the Linux system. it is mainly used to display the contents of the small file on the terminal:

```
# cat dept.lst
01|accounts|6213
02|progs|5423
03|marketing|6521
04|personnel|2365
05|production|9876
```

OTHER COMMANDS:
cal : [The calendar]
This command prints the calendar for a month. However cal can print the calendar for the entire year too.

```
Ex. #cal 2010
```

date : [Display the system date]
This command displays the current date which shows the date and time to the nearest second.

```
Ex. #date
Thus Feb 18 08:25:23 IST 2010
```

This command can also be used with the suitable format specifies as arguments. Each format is preceded by a (+) plus symbol followed by the (%) mod operator & a single character describing the format.
For instance, you can print only the month using the format +%m

```
#date +%m
02
```

or the month name using

```
#date +%h
Feb
```

or you can combine them in one command

```
#date + "%h %m"
Feb 02
```

There are other format specifiers:

d	day of the month (1 – 31)
y	last 2 digits of year (00 – 99)
H, M and S	hour, minute & second respectively.

You can use multiple format specifiers, you must enclose them within quotes (single or double) and use a single (+) symbol before it.

bc : [Calculator]

When you invoke bc without argument the input has to be typed in each line terminated by pressing <enter>.After you have finished your work use <Ctrl-d>

```
#bc
12+5
17
<Ctrl-d>
```

The output is shown in next line. You can also use bc to perform the calculation together.

```
12*12 ; 2^3
144
```

When you divide two numbers

```
9/5=1                                   #decimal portion truncated
```

By default, bc performs truncated divisions and you have to set scale to the number of digit of precision before you perform any division.

```
Scale=2
17/7
2.42
```

bc command can convert number from one base to other i.e. when setting IP address in a network you may need to convert binary to decimal.

ibase (input base) to 2 before you provide the number:

```
ibase=2
11001010
202
```

The reverse is also possible this time with obase:

```
obase=2
14
1110
```

It also handles hexadecimal numbers

```
obase=16
14
E
```

man: [Browsing the manual pages]

Linux offers online help facilities with a man command for example

```
#man wc
```

The entire man page of the linux manual pasting to wc command displayed on the screen man presents o/p a timer man presents command syntax algorithm with brief description often supported by a suitable example. This facility is help full for a user can obtain a correct syntax of the command & fair idea about its objectives.

The limitations of this commands are specified explicitly and indication of honesty of the designer.

tput clear:[Clearing the screen]

All UNIX system offers the tput command to clear the screen. However one thing is obvious
tput requires additional input to work properly. To make tput work, follow tput with the word clear

```
#tput clear          clear is an argument to tput
```

who:[who are the users?]
The who command displays an informative listing of these users:

```
#who
Root        console        Aug1    07:51       (:0)
Kumar       pts/6          Aug1    07:56       (pc123.heavens.com)
Sachin      pts/4          Aug1    02:16       (mercury.heavens.com)
```

While it's a general feature of most UNIX commands io avoid cluttering the display with header information, this command does have a header option(-H).This option prints the column headers, and when combined with the –u option, provides a more detailed list:

```
#who-Hu
NAME      LINE      TIME            IDLE     PID        COMMENTS
Root      console   Aug 1 07:51     0:48     11480      ( : 0)
Kumar     pts/10    Aug 1 07:56     0:33     11200      (pc123.heavens.com)
Sachin    pts/14    Aug 108:36       .       13678      (pc125.heavens.com)
```

One of the users shown in the first column is obviously the users who invoked the who command. To know that specifically,use the arguments am and i with who:

```
#who am i
Kumar     pts/10    Aug 1 07:56     (pe123.heavens.com)
```

uname:(knowing your machine's characteristics]
The uname command displays certain features of the OS running on your machine. By default, it simply displays the name of the OS:

```
#uname
Linux
```

The current release (-r): The -r option is used here to find out the version of your OS.

```
#uname -r
3.2.6
```

The machine name (-n): The-n option tells you the hostname:

```
#uname -n
bt
```

passwd:[changing your password]
If your account doesn t have a password or has one ie.already known to the others.you should change it immediately This is done with the passwd command.

```
#passwd
Passwd: changing password for root
Enter new login password: ****
Retype new login password: ****
passwd: password updated successfully.
```

3. WORKING WITH FILES AND DIRECTORIES

The Linux looks at everything as a file. If you write a program you add one man file to the system. Apart from file Linux file system also permits creation of folder also called directories. Just as an office has separate file cabinet to group files of similar nature. Linux also organizes their own files into these directories.

The file: Linux file system divides the file into 3 categories

1. Ordinary file	: Contains only data
2. Directory file	: Contains other files and directory
3. Device file	: Represent all hardware devices.

The reason for these distinctions is that the significance file's attribute depends on its file.

ORDINARY FILE:

It consist of stream of data resident on some permanent magnetic media. You can put anything you want into this type of file. This includes all data, source programs, object and executable code. All linux command as well as any file created by user. Commands like cat, ls etc are created as ordinary file. This file is also refer to as regular files.

The most common type of ordinary file is text file. The characteristic feature of text file is the data inside them are divided into group of line with each line terminated by new line character is not visible and does not appear in the hard copy output.

It generated by the system when you press the enter key.

DIRECTORY FILE:

A Directory contain no external data but keeps some details of file and subdirectory that it contains. The linux file system is organized with number of directory and sub directory and you can create them as and when you need.

A directory file contains two fields for each file i.e name of the file and its identification number (inode). When an ordinary file is created or removed its corresponding directory file is automatically updated by kernel with relevant information about the file.

DEVICE FILE:

The definition of a file has been broadened by linux to consider even physical devices as file. The device file is special in the sense that any output directed to it will be reflected on to respective physical device associated with the filename. When you issue a command to print a file you are really directing the files output to the file associated with the printer.

DIRECTORY RELATED COMMANDS:

pwd: Checking your current directory

It is a remarkable feature of UNIX system that like a file a user also occupies a certain slot in the file system. When you login you are placed in a specific directory of the file system. This directory is known as your current directory.

The pwd (present working directory) command tells you that.

Eg:- #pwd

```
/user/kumar
```

You see about your pathname is simply a sequence of directory name.This pathname shows your pathname with reference to top i.e root. pwd here tells you that you are placed in the directory kumar which has parent directory user which in turn is directly under root. These slashes act as delimeter to file and directory name except the first slash in a synonym for root.

cd: Changing directory

You can move around in the file system by using the CD (change directory) command when used with an argument it changes the current directory to the directory specified as the argument

Ex: #pwd

```
/user/kumar
#cd progs
#pwd
/user/kumar/progs
```

Though pwd displays the absolute path name cd does not need to use one the command cd progs here means change your sub directory to progs under the current directory using an absolute path name cause no harm either i.e use cd/user/kumar/progs for same effect when you need to switch to the /bin directory where most of the commonly used command are kept you should use absolute path name

Ex:- #pwd

```
/user/kumar/progs
#cd /bin
#pwd
/bin
```

cd can also be used without any argument

```
#pwd
/user/kumar/progs
#cd
#pwd
/user/kumar
```

mkdir: making directory

Directory can be created with mkdir (making directory) command. So the shorthand notation md is allowed in Linux The cmd is followed by the name of the directories to be created the directory path is created under the current directory as shown below

# mkdir path

You can create number of sub directories with one mkdir command:

# mkdir path abs doc

You can create directory chains with just one invocation of the command

# mkdir ppp pi/progs pi/data

This creates three sub directories---ppp and two sub directories under ppp.

Sometimes the system refuses to create a directory:

#mkdir test

mkdir: cant make directory test

This can happen due to these reasons:-

- The directory **test** may already exit
- There may be an ordinary file of the same name in current directory
- The user doesn’t have adequate authorization to create a directory. This will be considered when dealing with file permissions.

rmdir: Removing directories

The rmdir(remove directory)command removes directories (rd allowed only in Linux).You just have to do this to remove the directory ppp:

#rmdir ppp

Like mkdir, rmdir can also delete more than on directory in one shot.i.e.,

#rmdir ppp/data ppp/progs ppp

Note that you delete a directory and its subdirectories, a reverse logic has to be applied. The following directory sequence used by mkdir is invalid in rmdir:

#rmdir ppp ppp/progs ppp/data

rmdir: ppp:Directory not empty

This error message leads to two important rules that you should remember when deleting directories:

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

MANIPULATING ABSOLUTE AND RELATIVE PATHS USING CD COMMAND:

Many UNIX commands use file and directory names as arguments, which are presumed to exist in the current directory.

```
#cat login.sql
```

Will work only if the file login.sql exists in your current directory. However if you are placed in /usr and want to access login.sql , you can't obviously use the command, but rather the pathname of the file:

```
#cat  /home/student/login.sql
```

If the character of a pathname is /, the file's location must be determined with respect to root(the first/).Such a pathname is called **absolute pathname**.

If you want to have file accessed which is there in a sub-directory of the current directory then no absolute path is required.

Here, both progs and login.sql are presumed to exist in the current directory. Now if progs also contains a directory scripts under it, you won't need an absolute pathname to change to that directory:

```
cd progs/scripts
```

Here we used a relative path specification. Relative pathname, in the sense they are known.

Relative pathname:

Here, both progs and login.sql are presumed to exist in the current directory. Now if progs also contains a directory scripts under it, you won't need an absolute pathname to change to that directory:

```
#cd progs/scripts
```

Here we used a relative path specification. Relative pathname, in the sense they are known.

Using (.) and (..) in relative pathnames :

UNIX offers a shortcut- the relative pathname- that uses either the current or parent directory as reference and specifies the path relative to it. A relative pathname uses one of these cryptic symbols:

- **. (a single dot)** - This represents the current directory.
- **..(two dots)-** This represents parent directory.

Assuming that you are placed in /home/kumar/progs/data/text, you can use (..) as an argument to cd to move to the parent directory, /home/kumar/progs/data:

```
#pwd
/home/kumar/progs/data/text
#cd ..                          Moves one level up
#pwd
/home/kumar/progs/data
```

The command cd .. : Translates to this "changing your directory to the parent of the current directory".
You can combine any number of such sets of .. Separated by /s. For instance, to move to /home, you can always use cd/home. Alternatively, you can also use a relative pathname:

```
#pwd
/how/kumar/progs/data/text                Moves two levels up
#cd ..
#pwd
/home/kumar/progs/data
```

FILE RELATED COMMANDS:

cp: Copying a file

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

```
#cp chap01 unit1
```

If the destination file doesn't exist, it will first be created before copying takes. If not it will simply be overwritten without any warning from the system. So be careful when you choose your destination file.
The following examples show two ways of copying a file to the progs directory:

```
#cp chap01 progs/unit1
#cp chap01 unit1
```
cp often used with the short-hand notation .(dot) to signify the current directory as the destination. For instance, to copy the file .profile from /usr/sharma to your current directory, you can use either of the two commands:
```
#cp /usr/sharma/.profile  .profile    #destination is a file
#cp /usr/sharma/.profile .            #destination is the current directory
```
We can also copy more than one files to a single file like this:
```
#cp chap01 chap02 chap03 progs
```
Interactive copying(-i)
The –i(interactive) option, warns the user before overwriting the destination file. If unit1 exits, cp prompts for a response:
```
# cp –i chap01 unit1
cp:overwrite unit1?y
```
A y at this prompt overwrites the file, any other response leave it uncopied.

mv: Renaming files

mv (move) has two functions---renaming a file or directory and moving a group of files to a different directory.mv doesn't create a copy of a file; it merely renames it. To rename the file chap01 to man01, you should use
```
#mv chap01 man01
```
If the destination doesn't exist it will be created.mv by default doesn't prompt for overwriting the destination file if it exists.
Like cp a group of files can be moved, but only to a directory. The following command moves three files to the progs directory:
```
#mv chap01 chap02 chap03 progs
```
mv can also be to rename a directory,for instance ppp to perdir
```
#mv  ppp  perdir
```
There is –i option with mv behaving exactly like in cp. The messages are same and requires similar response.

rm: Deleting file

Files can be deleted with rm(remove).It can delete more than one file with a single instruction. The following command deletes the first three chapters of the book:
```
# rm chap01 chap02 chap03
```
rm won't remove directory, but it can remove files only from one.

Interactive deletion (-i)
Like in cp, the –I(interactive) option makes the command ask the user for confirmation before removing each file:
```
# rm –i chap01 chap02 chap03
chap01: ?y
chap02: ?n
chap03: ?y
```
A y removes the file, any other response leaves the file undeleted.

cat: Displaying and creating files

cat is one of the most well-known commands of the Linux system. it is mainly used to display the contents of the small file on the terminal:
```
# cat dept.lst
01|accounts|6213
02|progs|5423
03|marketing|6521
04|personnel|2365
05|production|9876
```
cat also accepts more than one filename as arguments

```
cat chap01 chap02
```

The contents of the second file are shown immediately after the first file without any header information. What cat has done here is concatenate the two files—hence its name.

Using cat to create a file

cat is also useful for creating a file. Enter the command cat, followed by the >(right chevron) character and the filename(for example foo):

```
# cat > foo
A > symbol following the command means that the
Output goes to the filename following it.cat used
In this way represents a rudimentary editor.
<ctrl-d>
#_
```

To verify this simply "cat" this file:

```
# cat foo
A > symbol following the command means that the
Output goes to the filename following it.cat used
In this way represents a rudimentary editor.
```

cmp: Comparing two files

There are three commands in the UNIX system that can tell you whether two files are identical. Obviously, it needs two filenames as arguments:

```
#  cmp chap01 chap02
chap01 chap02 differ: char 9,line 1
```

The two files are compared byte by byte, and the location of the first mismatch (in the ninth character of the first line)is echoed on the screen.cmp doesn't bother about possible subsequent mismatches.

If two files are identical, cmp displays no message, but returns the prompt.

```
# cmp chap01 chap01
```

comm: What is common?

comm is the command which requires two sorted files, and lists the differing entries in the different columns.

```
# cat file 1                       # cat file2
c.k.shukla                         anil agarwal
chanchal                           barun
s.dasgupta                         c.shukla
sumit                              lalit
                                   s.dasgupta
```

When comm command is run, it displays three-columnar output.

```
# comm file1 file2
        abhi
        barun
                c.shukla
chanchal
        lalit
                s.dasgupta
sumit
```

The first column contains two lines unique to the first file, and the second column shows three lines unique to the second file. The third column displays two lines(hence its name)to both files.

diff: Converting one file to another

diff is the third command that can be used to display file differences. It tells you which lines in one file have to be changed to make the two files identical. When used with the same files, it produces a detailed output:

# **diff file1 file2**

0a1,2	*Append after line 0 of first file*
>anil agarwal	*this line*
>barun	*and this line*
2c4	*Change line 2 of first file*
< chanchal	*Replacing this line*
--	*with*
>lalit	*this line*
4d5	*Delete line 4 of first file*
<sumit	*containing this line*

diff contains certain special symbols and **instructions** to indicate the changes that are required to make two files identical.Each instructions uses an address combined with an action that is applied to the first file. The instruction **0a1, 2** means appending two lines after line0,which become lines 1 and 2 in the second file,**2c4** changes line 2 which is line 4 in the second file.4d5 deletes line 4.

tar: The archival program

For creating a disk archive that contains a group of files or an entire directory structure, we use tar. For minimal use of tar we need to know these *key* options:

-c	Create an archive
-x	Extract files from archive
-t	Display files in archive
-f *arch*	Specify the archive *arch*

We'll also learn to use **gzip** and **gunzip** to compress and decompress the archive created with tar.

Creating an Archive (-c):

To create an archive we need to specify the name of the archive (with –f),the copy or write operation(-c) and the filenames as arguments. Additionally, we'll use the –v (verbose) option to display the progress while tar works. This is how we create a file archive, archive.tar, from the two uncompressed files used previously:

```
# tar –cvf archive.tar libc.html user_guide.ps
```

Let's see how we can compress this archive.

Using gzip with tar   If the created archive is very big,you may like to compress it with gzip.

```
gzip archive.tar
```

This creates a "tar-gzipped" file, archive.tar.gz. This file can be sent out by FTP or as email attachment to someone.

Extracting Files from Archive(-x) :

tar uses the –x option to extract files from an archive.

```
#tar –xvf archive.tar
```

But to extract files from a .tar.gz file(like archive.tar.gz),you must decompress it using gunzip and then run tar.

```
# gunzip archive.tar.gz
# tar –xvf archive.tar
```

Viewing the Archive(-t) :

To view the contents of the archive, use the –t(table of contents)options. It doesn't extract files, but simply shows their attributes:

```
# tar –tvf archive.tar
```

umask: default file and directory permissions

When you create files and directories, the permissions assigned to them depend on the systems default setting. The UNIX system has the following default permissions for all files and directories:

• rw-rw-rw- (octal 666) for regular files.
• rwxrwxrwx - (octal 777) for directories.

However, you don't see these permissions when you create a file or directory. Actually, this default is transformed by subtracting the user mask from it to remove one or more permissions. Let's evaluate the current value of the mask by using umask without arguments:

```
#umask
022
```

This is an octal number which has to be subtracted from the system default to obtain the actual default. This becomes 644 (666-022) for ordinary files and 755 (777-022) for directories. When you create a file on this system, it will have the permissions rw-r- -r- -

wc: Counting Lines, Words and Characters

UNIX features a universal word-counting program that also counts lines and characters. It takes one or more filenames as arguments and displays a four-columnar output. Before you use wc on the file infile, just use cat to view its contents:

Create an "infile" file by using cat command:

```
#cat > infile
#cat infile
I am the wc command
I count characters, words and lines
With options I can also make a selective count
```

You can now use wc without options to make a "word count" of the data in the file:

```
#we infile
    3      20      103 infile
```

wc counts 3 lines, 20 words and 103 characters. The filename has also been shown in the fourth column.

- A line is any group of characters not containing a newline.
- A word is a group of characters not containing a space, tab or newline.
- A character is the smallest unit of information and includes a space, tab and newline.

4. BASIC FILE ATTRIBUTES

ls –l : Listing file attributes

We have already used ls command. It is the –l(long) option that is the most revealing and also the one that is most frequently used. This option will tell you not only the size of the file, but also its access rights (permissions), its last modification time and the ownership details.

Let us use ls –l to list the seven attributes of all file in the current directory

```
# ls –l
Total 72
-rw-r—r--      1    kumar       metal   19514           May 10 13:45    chap01
-rw-r—r--      1    kumar       metal    4174           May 10 15:01   chap02
-rw-rw-rw-     1    kumar       metal       84          Feb  12  12:30   dept.lst
-rw-r—r--      1    kumar       metal    9156           Mar  12  1996   genie.sh
drwxr-xr-x     2    kumar       metal      320           May  9  09:57   progs
```

this list preceded by the words "total 72",which indicates tht a total of 72 blocks are occupied by these files in the disk, each block consisting of 512 bytes(1024 in Linux).

File type and permissions:
The first column shows the type and permissions associated with each file. The first character in this column is usually a -, which indicates that the file is an ordinary one. This is, however not so for the directory progs that shows a d at the same position.
You can see a series of characters that can take the values **r**,**w**,**x** and **-**.A file can have three types of permissions-read, write and execute.

Links:
The second column indicates the number of links associated with the file. this is actually the number of filenames maintained by the system of that file.

Ownership:
When you create a file, you automatically become the owner. The third column shows kumar as the owner of all these files. Generally owner of a file have full authority to tamper with its contents, permissions and even ownership. Similarly you can create, modify or remove files in a directory if you are the owner of the directory.

Group ownership:
When a user account is opened by the system administration, he simultaneously assigns the user to some group. The fourth column stands for the group owner of the file.

File size:
The fifth column shows the size of the file in bytes, i.e., the amount of data it contains. The important thing to remember is that it is only a character count of the file and not a measure of the disk space that it occupies.

File modification time:
The sixth, seventh and eight columns indicate last modification time of the file, which is stored to the nearest second.

Filename:
The last column displays the filenames arranged in alphabetically.

The –d option: Listing directory attributes
Ls, when used with directory names, lists files in the directory, rather than the directory itself. If you want to force ls to list the directory attributes rather than its contents, you need to use the –d(directory) option:

```
# ls   –ld  progs
drwxr-xr-x     2    kumar       metal       320            May  9  09:57   progs
```

Directories can easily be identified from the long listing by first character of the first column, which here shows a **d**. For ordinary files this place always shows **-**(hyphen) and for device files either **b** or **c**

FILE PERMISSIONS:

UNIX has well defined and simple system of assigning permissions to files. Consider the first column of output of ls –l with a filename or without that represents file permissions.
UNIX follows three-tiered file protection system that determines a file's access rights. To understand how this works, let's break up the permissions string of any file (chap01) into three groups. The initial - (the first column) represents an ordinary file and is left out of the permissions:

r w x   r-x   r- -

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

The first group (rwx) has all three permissions. The file is readable, writable and executable by the *owner* of the file. The third column of the file attribute shows the owner and first permission group applies to the owner.

The second group (r-x) has a hypen in the middle slot, which indicates the absence of write permission by the *group owner* of the file.

The third group(r--) has the write and execute bits absent. This set of permissions is applicable to others, i.e., those who are neither the owner nor belong to the group.
You can set different permissions for the three categories of users-owner, group and others.

chmod: CHANGING FILE PERMISSIONS

The chmod(change mode) command is used to set the permissions of one or more files for all three of users(user, group and others).It can be run only by the user(the owner).
The command cab be used in two ways:

- In a relative manner by specifying the changes to the current permissions
- In an absolute manner by specifying the final permissions.

Relative permissions: When changing permissions in a relative manner, chmod only changes the permissions specified in the command line and leaves the other permission unchanged .In this mode it uses the following syntax:
chmod category operation permission filename(s)

chmod takes as its argument an expression comprising some letters and symbols that completely describe the user category and the type of permission being assigned or removed. The expressions contains three components:

- User category(user, group, others)
- The *operation* to be performed(assign or remove a permission)
- The type of *permission*(read, write, execute)

Abbreviations used by chmod:

Category	**Operation**	**Permission**
u-User	**+ Assigns permission**	**r Read permission**
g-Group	**- Removes permission**	**w Write permission**
o-Others		**x Execute permission**
a-All(ugo)		

Example:
```
# chmod u+x start
# chmod ugo+x start
# chmod a+x start
# chmod a-x,go+r start
# chmod o+wx start
```

Absolute permission:
When we don't need to know what a file's current permissions are, but want to set all nine permissions bits explicitly. The expression used by **chmod** here is string of three octal numbers (base 8).Octal numbers use the base 8 and octal digits have the values 0 to 7.This means that a set of three bits can represent one octal digit. If we represent the permissions of each category by one octal digit, this is how the permissions can be represented:

- Read permission-4(octal 100)
- Write permission-2(octal 010)
- Execute permission-1(octal 001)

For each category we add up numbers. For instance,6 represents read and write permissions and 7 represents all permissions.

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

```
# chmod 666 start
```
The 6 indicates read and write permissions (4+2).

chown: CHANGING FILE OWNER

chown transfers ownership of a file to a user, and it seems that it can optionally change the group as well. The command is used in this way:

chown options owner [: group] file (s)

To now renounce the ownership of the file note to Sharma, use chown in the foll:

```
#ls -l note
-rwxr----x     1      kumar          metal  347     may     10     20:30  note
#chown Sharma note; Is -1 note
-rwxr----x     2      sharma         metal  347     may     10     20:30  note
#exit
```

MANIPULATING HARD LINK AND SOFT LINK USING ln COMMAND:

ln : creating hard links:

A file is linked with the In command, which takes two filenames as arguments. The command can create both hard and soft link and has syntax similar to the one used by cp. The following command (hard) links emp.lst with employee:

```
#ln emp.lst employee                employee must not exist
```
The -i option to Is shows that they have same inode number, meaning that they are actually one and the| same file:

```
#ls -li emp.lst employee
29518  -rwxr-xr-x  2  kumar  metal  915  may  4  09:58  emp.lst
29518  -rwxr-xr-x  2  kumar  metal  915  may  4  09:58  employee
```
The link count which is normally one for unlinked file is shown to be two. You can link a third filename,emp.dat and increase the number of links to three.

```
#ln employee emp.dat; ls -l emp*
29518  -rwxr-xr-x   2   kumar   metal   915   may   4  09:58   emp.dat
29518  -rwxr-xr-x   2   kumar   metal   915   may   4  09:58   emp.lst
29518  -rwxr-xr-x   2   kumar   metal   915   may   4  09:58   employee
```

The rm command removes a file by deleting its directory entry,so we expect the same command to remove a link also:

```
#rm emp.dat
#ls -I emp.Ist employee
-rwxr-xr-x 2 kumar metal 915 may 4 09:58 emp.lst
-rwxr-xr-x 2 kumar metal 915 may 4 09:58 employee
```

So the link count has come down into two.Another rm will further bring it down to one.

Symbolic links and ln:

The symbolic link is the fourth file type considered in this text. Unlike the hard link, a symbolic link doesn't have the file contents, but simply provides the pathname of the file that actually has the contents. Being more flexible, a symbolic link is also known as a soft link.

The In command creates symbolic links also, except that you have to use-s option. This time the listing tells you a different story:

Firstly create a file by name note then,

```
#ln-s note note.sym
#ls -li note note.sym
9948   -rw-r--r--    l  kumar   group 80  feb  16   14:52  note
9952 lrwxrwxrwx 1   kumar   group 4    feb 16    15:07 note.sym>note
```

You can identify symbolic links by a character I seen in the permission field. The pointer
Notation -> note suggests that note.sym contains the pathname for the filename note. Its note and not| note.sym that actually contains the data. It's important you realize that this time we indeed have two files and they are not identical. Removing note.sym won't affect us much because we can easily recreate the link. But if we remove note,we would lose the file containing the data. In that case note.sym would point to a nonexistent file and become a dangling symbolic link.

5. VI EDITOR

vi is full-screen editor now available with all Linux systems. It is one of the most powerful editors available in any environment. vi offers cryptic and sometimes mnemonic, internal commands for editing work. It make complete use of keyboard, where practically every key has a function.vi has innumerable features, and it takes time to master most of them.
Linux features a number of **vi** editors of which **vim**, (vi improved) is the most common.
THE THREE MODES:
A vi session begins by invoking the command vi with (or without) a filename:

vi visfile

You are presented a full screen, each line beginning with a ~(tilde).This is vi's way of indicating that they are non-existent lines. For text editing, vi uses 24 of the 25 lines that are normally available in a terminal. The last line is reserved for some commands that you can enter to act on the text. This line is also used by the system messages. The filename appears in this line with the message "visfile" [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*. This is the mode where you can pass commands to act on the text, using most of the keys of the keyboard. Pressing a key doesn't show it on screen, but may perform a function like moving the cursor to the next line, or deleting a line. You can't use the command mode to enter or replace text.

There are two command mode functions that you should know at this stage the space bar and the backspace key. The spacebar takes you one character ahead, while the backspace key(or <Ctrl-h> takes you a character back. Backspacing in this mode doesn't delete text at all.

To enter text, you have to leave the command mode and enter the input mode. There are some keys which takes you to this mode, and whatever you enter shows up on the screen. Backspacing in this mode however erases all characters that the cursor passes through. To leave this mode you have to press the <Esc> key.
You have to save your file or switch to editing another file. You then have to use *ex mode* or line mode, where you can enter the instruction in the last line of the screen.
With this knowledge, we can summarize the three modes in which vi works:

- Input Mode: Where any key depressed is entered as text
- Command Mode: Where any key are used as commands to act on text
- E**x** Mode: Where ex mode commands can be entered in the last line of the screen to act on text

INPUT MODE:

Insertion of text
The simplest type of input is insertion of text. Whether the file contains any text or not, when vi is invoked the cursor is always positioned at the first character of the first line. To insert text at this position, press

i

The character doesn't show up on the screen, but pressing this key changes the mode from command to input. You start insertion with I, which put text at the left of the cursor position. If the I command is invoked with the cursor positioned on existing text, text on its right will be shifted further without being overwritten.
There are other mode of inputting text. To append text to the right of the cursor position, use

a

Followed by the text you wish to key in. After you have finished editing, press <Esc>.
With i and a, you can append several lines text in this way. They also have their uppercase counterparts performing similar functions. I inserts text at the beginning of a line, while A appends text at the end of a line.

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

```
O
```

This inserts an empty line below the current line.
O also opens a line, but above the current line. Press <Esc> after completing text input.
Replacing text:
Text is replaced with the r,R,s and S keys. To replace one single character but another, you should use

```
r                               #no <Esc> required
```

Followed by the character that replaces the one under the cursor. You can replace a single character only in this way.

SAVING TEXT AND QUITTING -THE EX MODE

Saving you work:
To enter command in this mode, enter a:,which appears at the last line of the screen, then the corresponding ex mode command and finally the <Enter> key. To save a file and remain in the editing mode, use the w (write) command.
Saving and quitting
To save file and to quit, use the x (exit) command.
Aborting editing:
It is possible to abort the editing process and quit process and quit the editing mode without saving the buffer. The q (quit) command is used to do that.

COMMAND MODE:

This is the mode you come to when you have finished entering or changing your text. Unlike in the input mode, when you press a key in the command mode, it doesn’t show up on the screen but simply performs its function.
Deletion:
The simplest text deletion is achieved with the x command. This command deletes the character under the cursor. Move the cursor to the character that needs to be deleted, and then press

```
x
```

The character under the cursor gets deleted and the text on the right shifts left to fill up the space. A repeat factor applies here, so 4x deletes the current the current characters from the right.
Navigation:
You can use the keys h, j, k and l. They are located in a line in the middle row of your keyboard. These navigation keys, when invoked as such in the command mode, move the cursor by one position.
The horizontal and vertical movements of the cursor are illustrated. Their significance is shown below:

h(or backspace)	Moves cursor left
j(or spacebar)	Moves cursor right
k	Moves cursor up
l	Moves cursor down

6. SIMPLE FILTERS

Filters tilting tools:
Group of commands, each of which accepts some data as input, performs some manipulation on it, and produces some output. Since they perform some filtering action on the data, they are appropriately called filters. Filters are the central tools of the Linux tool kit, and each filter in this tool kit performs a simple function. They all use standard input and standard output and can therefore be used with redirection and pipelines.
Henceforth, lines will also be referred to as records. Each record is separated from the next by the new line character. Fields will have same significance as the programmer normally understands. Where fields are separated by a suitable delimiter that may be '|'

THE SAMPLE DATABASE:
Henceforth you will be learning the features of several commands, including the advanced ones, text editing and shell programming with reference to a file **emp.lst**.

```
# cat emp.lst
2233   |a. k. shukla        |g.m        |sales        |12/12/52|6000
9867   |jai sharma          |director   |production   |12/03/50|4000
3456   |sumit chakrobarty   |d.g.m      |marketing    |19/04/43|6000
2345   |n.k.gupta           |chairman   |personnal    |11/03/47|7800
1008   |chanchal singhvi    |director   |sales        |03/02.90|4500
2456   |anil aggarwal       |manager    |sales        |01/09/90|4000
```

This is a text file that contains personnel database. There are 6 records (lines) in the file. Each record has 6 fields, each field separated by the delimiter |.The details of an employee are stored in one record. A person is identified by the emp-id, name, designation, department, date of birth and salary.

head: DISPLAYING THE BEGINNING OF A FILE

The **head** command, as the name implies displays the top of the file. When used without an option, it displays the first ten records of the argument file:
You have to specify line count and display, say, the first three lines of the file. Use the – symbol, followed by a numeric argument:

```
# head -3 emp.lst
2233   |a. k. shukla        |g.m      |sales      |12/12/52|6000
9867   |jai sharma          |director  |production|12/03/50|4000
3456   |sumit chakrobarty   |d.g.m     |marketing |19/04/43|6000
```

tail: DISPLAYING THE END OF A FILE

The **tail** command displays the end of the file. It provides an additional method of addressing lines, and like head displays the last ten lines when used without arguments. The last three lines are displayed in this way:

```
# tail -3 emp.lst
2345|n.k.gupta           |chairman    |personnal    |11/03/47|7800
1008|chanchal singhvi    |director       |sales      |03/02.90|4500
2456|anil aggarwal       |manager  |sales  |01/09/90|4000
```

You can address lines from beginning of the file, instead of the end. The +count option allows you to do that, where count represents the line number from where the selection should begin. Since the file contains 6 lines the last five implies using

```
#tail +4 emp.lst
```

cut: SLITTING A FILE VERTICALLY

While **head** and **tail** are used to slice a file horizontally, you can slice a file vertically with the cut command. cut identifies both columns and fields, and is a useful filter for the programmer.
The features of the **cut** and **paste** commands will be illustrated with specific references to the file **shortlist**, which stores the first five lines of **emp.lst:**

```
# head -5 emp.lst | tee shortlist
2233   |a. k. shukla          |g.m          |sales          |12/12/52|6000
9867   |jai sharma            |director     |production     |12/03/50|4000
3456   |sumit chakrobarty     |d.g.m        |marketing      |19/04/43|6000
2345   |n.k.gupta             |chairman     |personnal      |11/03/47|7800
1008   |chanchal singhvi      |director     |sales          |03/02/90|4500
```

Note the use of the **tee** facility that saves the output in the file **shortlist** and also displays it on the terminal.

The –c Option: Cutting columns

cut can be used to extract specific columns from this file, say those signifying the name(second column) and the designation (third field).The name starts from column number 6 and goes up to column number 22,while the designation data occupies columns 24 through 31.Use cut with the **–c**(columns) option for cutting columns:

```
# cut –c 6-22, 24-31 shortlist
a. k. shukla          g.m
jai sharma            director
sumit chakrobarty     d.g.m
n.k.gupta             chairman
chanchal singhvi       director
```

Note that there should be no white spaces in the column list. Ranges are also permitted, and commas are used to separate the column chunks. Moreover, cut uses a special form for selecting a column from the begging and up the end of a line:

```
#cut  –c  -3,6-22,53-  shortlist
```

The expression 53- indicates column number 51 to the end of the line. Similarly -3 is same as 1-3.

The –f Option: Cutting fields

Files often don't contain fixed length records, in which case, it is better to cut fields rather than columns. Two options need to be used here **–d** (delimiter), and **–f** (field) for specifying the field list. This is how you cut second and third fields:

```
# cut  –d  \|  -f 2,3  shortlist | tee cutlist1
a. k. shukla          |g.m
jai sharma            |director
sumit chakrobarty     | d.g.m
n.k.gupta             |chairman
chanchal singhvi      |director
```

The | was escaped to prevent the shell from interpreting it as the pipeline character: alternatively, it can also be quoted. To cut out fields numbered 1,4,5,6, and save the output in cutlist2, follow a similar procedure:

```
#cut –d  "|"  –f  1,4-  shortlist  >  cutlist2
```

When you use the –f option, you shouldn't forget to use the –d option too.

paste: PASTING FILES

What you cut with the previous command can be pasted back with the **paste** command. It is special type of concatenation in that it pastes files vertically, rather than horizontally. In the previous topic, cut was used to create the two files **cutlist1** and **cutlist2** containing two cut-out portions of the same file. Using paste, you can fix them laterally:

```
# paste cutlist1 cutlist2
a. k. shukla        |g.m          2233|sales      |12/12/52|6000
jai sharma          |director     9867 |production |12/03/50|4000
sumit chakrobarty   |d.g.m        3456 |marketing |19/04/43|6000
n.k.gupta           |chairman     2345|personnal  |11/03/47|7800
chanchal singhvi    |director     1008|sales      |03/02/90|4500
```

By default, paste uses the tab character for pasting files, but you can specify a delimiter of your choice with the – **d** (delimiter) option:

```
# paste –d  \|  cutlist1  cutlist2
a. k. shukla        |g.m      |2233       |sales      |12/12/52|6000
jai sharma          |director  |9867      |production |12/03/50|4000
sumit chakrobarty   |d.g.m    |3456       |marketing  |19/04/43|6000
n.k.gupta           |chairman |2345       |personnal  |11/03/47|7800
chanchal singhvi  |director  |1008  |sales  |03/02/90|4500
```

sort: ORDERING A FILE

sort performs usual sorting functions and works quite well with variable length records.
When the command is invoked without options, the entire line is sorted:

```
# sort shortlist
1008|chanchal singhvi       |director       |sales      |03/02/90|4500
2233|a. k. shukla           |g.m            |sales      |12/12/52|6000
2345|n.k.gupta              |chairman    |personnal     |11/03/47|7800
3456|sumit chakrobarty      |d.g.m          |marketing |19/04/43|6000
9867|jai sharma             |director       |production|12/03/50|4000
```

Sorting starts with the first character of each line, and proceeds to the next character only when the characters in two lines are identical. By default, sort reorders a line in ASCII collating sequence, starting from the beginning of the line. This default sorting sequence can be altered by using certain options.

sort Options:

Unlike cut and paste, sort uses one or more contiguous spaces as default field separator. We'll be using the -k (key) POSIX option to identify keys (the fields). We'll also use the -t option to specify the delimiter.

Sorting on Primary Key (-k)**:** let's now use the -k option to sort on the second field (name). The option should be -k 2. In the same way you can use it for third, fourth fields by just taking the options can –k 3 and so on.

```
#sort -t"|" -k 2 shortlist
2233|a. k. shukla           |g.m            |sales      |12/12/52|6000
2345|n.k.gupta              |chairman    |personnal     |11/03/47|7800
3456|sumit chakrobarty      |d.g.m          |marketing |19/04/43|6000
And so on...
```

The sort order can be reversed with the -r (reverse) option. The following sequence reverses a previous sorting order:

```
#sort -t"" -r -k 2 shortlist
2345|n.k.gupta              |chairman       |personnal  |11/03/47|7800
```

```
3456|sumit chakrobarty      |d.g.m          |marketing |19/04/43|6000
2233|a. k. shukla           |g.m            |sales     |12/12/52|6000
```

Sorting on Secondary Key: You can sort on more than one key, i.e., you can provide a secondary key to sort. If the primary key is the third ficld, and the secondary ficld is the second field, then you need to specify for every-k option, where the sort ends. This is done in this way:

```
# sort -t"|" -k 3, 3 -k 2, 2 shortlist
1008|chanchal singhvi       |director       |sales     |03/02/90|4500
2233|a. k. shukla           |g.m            |sales     |12/12/52|6000
2345|n.k.gupta              |chairman       |personnal |11/03/47|7800
3456|sumit chakrobarty      |d.g.m          |marketing |19/04/43|6000
9867|jai sharma             |director       |production|12/03/50|4000
```

This sorts the file by designation and name. -k 3, 3 indicates that sorting starts on the third field and ends on the same field. You'll observe here that second field is not sorted in order, it's because in the above command we have given 3, 3 so the third field is sorted first.
Sorting on columns: You can also specify a character position within a field to be the beginning of sort. If you are to sort the file according to the year of birth, then you need to sort on the seventh and eighth

```
#sort -t"|" -k 5.7, 5.8 shortlist
2233|a. k. shukla           |g.m            |sales     |12/12/52|6000
9867|jai sharma             |director       |production|12/03/50|4000
3456|sumit chakrobarty      |d.g.m          |marketing |19/04/43|6000
2345|n.k.gupta              |chairman    |personnal    |11/03/47|7800
1008|chanchal singhvi       |director       |sales  |03/02/90|4500
```

The -k option also uses the form -k m, n where n is the character position in the mth field, So. 5.7. 5.8 mean that sorting starts on column 7 of the fifth field and ends on column 8
Numeric sort (-n): when sort acts on numerals, strange things can happen. When you sort a file containing only numbers, you get a curious result:

```
#sort numfile
10
2
27
4.
```

This is probably not what you expected, but the ASCII collating sequence places 1 above 2, and 2 above 4. That's why 10 preceded 2 and 27 preceded 4. This can be overridden by the -n(numeric) option:

```
#sort -n numfile
2
4.
10
27
```

uniq : LOCATING REPEATED LINES

If in any environment there's problem of duplicate entries creeping in due to faulty data entry. Linux offers a special tool to handle these records-the **uniq** command.
Consider a sorted file dept.lst that includes duplicate lines:

```
# cat dept.lst
01|accounts|5432
01|accounts|5432
02|admin|5467
02|admin|5467
03|marketing|6521
03|marketing|6521
03|marketing|6521
```

uniq simply fetches one copy of each record and writes it to the standard output:

```
# uniq dept.lst
01|accounts|5432
02|admin|5467
03|marketing|6521
04|personnel|2365
```

Since uniq requires a sorted file as an input, the general procedure is to sort a file and pipe the process to uniq.

tr : TRANSLATING CHARACTERS

The **tr**(translate) command is one filter that manipulates individual characters in a file. More specifically, it translates characters using one or two compact expressions. It uses an unusual syntax:

tr options expression1 expression2 standard input

The first difference that should strike you that this command takes its input only from the standard input; it doesn't take a filename as argument. By default, it translates each character in expression1 to its mapped counterpart in expression2.the first character in the first expression is replaced by the first character in the second expression, and similarly for the other characters.

You can use tr replace the **|** with a **~**(tilde),and the **/** with a **-**.Simply specify two expressions containing them in the proper sequence:

```
# tr '| /' '~ -' < emp1.lst | head -3
2233~a. k. shukla        ~g.m       ~sales       ~12-12-52~6000
1008~chanchal   singhvi   ~director   ~sales    ~03-02-90~4500
9867~jai sharma          ~director  ~production~12-03-50~4000
```

Changing the case of the text: You 'll certainly need one to change the case of the first three records from lower to upper:

```
# head –n 3 emp1.lst | tr '[a-z]' [A-Z]'
2233|A K SHUKLA            |G.M          |SALES          |12/12/52|6000
1008|CHANCHAL SINGHVI|DIRECTOR  |SALES            |03/02/90|4500
9867|JAI SHARMA            |DIRECTOR  |PRODUCTION  |12/03/50|4000
```

Deleting characters (-d):

The file emp.lst has fields separated by delimiters, and the date formatted in readable form with a /. If you need to convert this file into traditional format, use the –d(delete) option to delete the characters | and / from the file. The following command does it for the first three lines:

```
# tr –d '| /' < emp1.lst | head -3
2233   a. k. shukla        g.m       sales       1212526000
1008   chanchal singhvi   director    sales          0302904500
9867   jai sharma          director   production   1203504000
```

pr: PAGINATING FILES

the **pr** command prepares a file for printing by adding suitable headers, footers and formatted text. It has many options, and some of them are quite useful. A simple invocation of the command is to use it with a filename as argument:

```
# pr dept.lst
May 06 10:38 1997 dept.lst Page 1
01:accounts:6213
02:admin:5423
03:marketing:6521
04:personnel:2365
05:production:9876
….blank lines….
```

pr adds five lines of margin at the top and five at the bottom. The header shows the date and time of last modification of the file, along with the filename and page number.

Because pr formats its input by adding margins and a header, it is often used as a "pre-processor" before printing with the **lp** command.

7. EXPRESSIONS AND SEARCH PATTERNS

A frequent requirement of the end user or the programmer is to look for a pattern in a file. Linux has a special family of commands for handling this feature, and the three members of this family are grep, egrep and fgrep depending on the options used, they output the lines containing(or not containing) the pattern, the filenames or the line numbers.
This chapter introduce you to regular expressions, one of the most powerful features of the Linux system.

grep: SEARCHING FOR A PATTERN

grep is one of the most popular and useful Linux filters. It scans a file for the occurrence of a pattern, and can display the selected pattern, the line numbers in which they were found, or the filenames where the pattern occurs, grep can also select lines not containing the pattern.
The syntax for the **grep** command is as follows:

grep *options pattern filename(s)*

grep requires an expression to represent the pattern to be searched for ,followed by one or more filenames. The first argument (barring the option) is always treated as the expression, and the ones remaining as filenames. This is how you display lines containing the pattern “sales” from the file emp1.lst:

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

It is generally safe to quote the pattern, though in this case, it is not required. Quoting is required if the search string consists of more than one word, or uses any of the shell’s characters like *, **,** etc.
grep is representative Linux command that silently returns the prompt in case the pattern can’t be located:

```
# grep president emp.lst                          #no president found
```

The command failed because the string “president” couldn’t be located. This is one feature that is unique to grep.

When grep is used with the series of strings, it interprets the first argument as the pattern,and the rest as filenames. It then displays the filenames along with the output:

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

The expressions are not quoted yet; using them will be superfluous for the two reasons mentioned earlier. However, you will be surprised if you scan the input for the name “jai sharma”. Try invoking the command with the pattern without quotes:

```
# grep jai sharma emp.lst
grep:can’t open sharma
emp.lst:9876|jai sharma           |director  |production|12/03/50|7000
```

grep introduces shrama as a filename, and obviously fails to open such a file.However, its search continues by using the next argument, viz.emp.lst. Now, quote the pattern:

```
# grep “jai sharma”  |director  |production|12/03/50|7000
```

grep options:

Counting lines containing patterns (-c):

How many directors are there in emp1.lst and emp2.lst? The –c (count) option counts the occurrences, and the following example reveals that there are two of them in each file:

```
# grep –c ‘director’ emp*.lst
emp1.lst:2
emp2.lst:2
```

This is one of the few grep options that doesn’t display the lines at all.

Displaying line numbers (-n):

The –n (number) option can be used to display the line numbers containing the pattern, along with the lines:

```
# grep –n ‘marketing’ emp.lst
3:5678  |sumit  chakrobarty|d.g.m  |marketing|19/04/43|6000
11:6521|lalit   chowdury   |director|marketing|26/09/45|8200
14:2345|j.b.saxena          |g.m      |marketing|12/03/45|8000
15:0110|v.k.agrawal  |g.m  |marketing|31/12/40|9000
```

The line numbers are shown at the beginning of each line, separated from the actual line by a:

Deleting lines (-v):

The –v (inverse) option selects all but the lines containing the pattern. Thus, you can create a file **otherlist** containing all but directors:

```
# grep –v ‘director’ emp.lst > otherlist
wc –l otherlist
   11 otherlist                                There were 4 directors initially
```

This is a useful tool for “deleting” records, but only by using redirection, as you have done here. The records are not deleted from the original file as such, but we could create a separate file **otherlist** containing all but the directors’ records.

Displaying filenames (-l):

The –l(list) option displays only the names of files where a pattern has been found:

```
# grep –l ‘manager’ *.lst
design.lst
emp.lst
emp1.lst
emp2.lst
```

Inoring case (-i):

When you look for a name, but are not sure of the case, grep offers the –i(ignore) option, which ignores case for the pattern matching:

```
# grep –i ‘agarwal’ emp.lst
3564|sudhir Agarwal  |executive|personnel|06/07/47|7500
```

Matching multiple patterns(-e):

This locates the name “Agarwal”, but it can’t match the names ”agrawal” and “aggarwal” that are spelled in a similar manner, while possessing some minor differences. With the –e option available, you can match the three agarwals by using grep like this:

```
# grep –e “Agarwal” –e “aggarwal” –e “agrawal” emp.lst
2476|anil aggarwal   |manager  |sales        |05/01/59|5000
3564|sudhir Agarwal|executive|personnel |06/07/47|7500
0110|a.k.agrawal      |g.m          |marketing|12/31/40|9000
```

The tedium of entering such a lengthy command line is compelling enough to use regular expressions, which we’ll discuss shortly.

Option	Significance
-c	Displays count of number of occurrences
-l	Displays list of filenames only
-n	Displays line numbers along with lines
-v	Displays all but lines matching expression
-i	Ignores case for matching
-h	Omits filenames when handling multiple files
-e exp	Specifies expression with this option. Can use multiple times
-n	Displays line and n lines above and below.

Regular expressions-An introduction:

View the file emp.lst once again and locate all the agarwals. On close examination, you will see that there are three lines containing similar, but not identical patterns. They are spelled as “Agarwal”, ”aggarwal” and “agrawal”;a name is often spelled in a number of ways. It is also a frequent requirement to locate an agarwal without knowing exactly how his name is spelled. The command

```
# grep Agarwal emp.lst
 3564|sudhir Agarwal  |executive |personnel  |07/06/47|7500
```

Doesn’t help as it lists only one line, the one exactly matching the pattern. It will indeed be a tedious task to specify each pattern separately, and then check whether the line selected matches the one you are looking for.

Grep uses an expression to match a group of similar patterns. This expression is a feature of the command that uses it, and has nothing to do with the shell. It has an elaborate character set and can perform amazing matches .If an expression uses any of these characters, it is termed a regular expression.

If more than one pattern is to be matched by a single expression then it must use at least one of these characters.

The Regular Expression Character Sub-set

Symbols	Significance
*	Matches zero or more occurrences of previous character
.	Matches a single character
[pqr]	Matches a single character p, q or r
[c1-c2]	Matches a single character within the ASCII range represented by c1 and c2
[^pqr]	Matches a single character which is not a p, q or r
^pat	Matches pattern pat at beginning of line
pat#	Matches pattern pat at end of line

The character class:

A regular expression lets you specify a group of characters enclosed within a pair of
Square brackets [], so that the match is performed for any single character in the group.
Thus the expression

[ra]

matches either an **r** or an **a**. These meta characters can also be used to match “Agarwal” and “agrawal”. the following regular expression

[aA]g[ar][ar]wal

Matches the two names. The character class **[aA]** matches the letter **a** in both lowercase and uppercase. The model [ar] [ar] matches any of the four patterns:

aa   ar   ra   rr

of which the second and third are relevant to the present problem. As a first step, let’s use this regular expression with grep:

```
# grep “[aA]g[ar][ar]wal” emp.lst
3564|sudhir Agarwal|executive|personnel|07/06/47|7500
```

```
0110|v.k.agrawal    |g.m.       |marketing|12/31/40|9000
```

THE *:

The *(asterisk) refers to the immediately preceding character. Here it indicates that the previous character can occur many times, or not at all. The pattern

g*

Matches the single character **g**, and any number of **g**s. Because the previous character may not occur at all, it also matches a null string. Thus, apart from this null string, it also matches the following strings:

g   gg   ggg   gggg   ………..

Mark the keywords "zero or more occurrences of the previous character", which are used to describe the significance of the *.Don't make a mistake of using this expression to match a string beginning with **g**; use **gg*** instead.

How do you then match all three patterns? The third pattern "aggarwal" contains an extra g, while the other patterns don't. To include this string also, you can use the * to enhance the previous expression. The regular expression is

[aA]gg*[ar][ar]wal

Matches all three patterns. This expression now contains the "sub-expression" **gg*** because all three names contain at least one single **g**. Enhance the previous expression by including this model, and it solves the problem:

```
# grep "[aA]gg*[ar][ar]wal" emp.lst
2476|anil   aggarwal   |manager   |sales   |05/01/59|5000
3564|sudhir  Agarwal|executive|personnel|07/06/47|7500
0110|v.k.agrawal  |g.m.  |marketing|12/31/40|9000
```

THE DOT:

A **.** matches a single character. The shell uses the ? character to indicate that. The pattern

2…matches a four-character pattern beginning with a 2.The shell's equivalent pattern is 2????

THE REGULAR EXPRESSION.*:

The dot along with the *(.*) constitutes a very useful regular expression. It signifies any number of characters, or none. For instance, you may be looking for the name "j. saxena", but are not sure whether it actually exits in the file as "j.b.saxena". No problem, just embed the **.*** in the search string:

```
# grep "j.*saxena" emp.lst
2345|j.b.saxena  |g.m.   |marketing|03/12/45|8000
```

SPECIFYING PATTERN BOUNDARIES (^ AND #):

A regular expression possesses one more property, viz. that it can be used to specify the boundaries in the line within which the matching has to take place. A pattern is matched at the beginning with the ^ (caret) as the pattern prefix. Likewise, the # signifies the end of the pattern, and used to postfix it. Anchoring a pattern in this way is often necessary when it can occur in more than one place in a line, and you are interested in its occurrence only at a particular location.

Consider a simple example. Try to extract those lines where the emp-id begins with a 2.What happens if you simply use

2…

As the expression ? This won't do because the character 2, followed by three characters, can occur anywhere in the line. You must indicate to grep that the pattern occurs at the beginning of the line, and the ^ does it easily:

```
# grep "^2" emp.lst
2233|a.k.shukla     |g.m.     |sales           |12/12/52|6000
2365|barun  sengupta|director  |personnel|05/11/47|7800
2476|anil    aggarwal    |manager    |sales   |05/01/59|5000
2345|j.b.saxena  |g.m.  |marketing|03/12/45|8000
```

Similarly to select those lines where the salary lies between 7000 and 7999,you have to use the # at the end of the pattern:

```
# grep "7…#" emp.lst
9876|jai sharma          |director   |production|03/12/50|7000
2365|barun  sengupta  |director  |personnel  |05/11/47|7800
3564|sudhir Agarwal  |executive|personnel  |07/06/47|7500
```

How can you reverse the search and select only those lines where the emp-ids don't begin with a 2? You need the expression ^[^2],and the following command should do the job:

```
grep "^[^2]" emp.lst
```

egrep: EXTENDING GREP

The egrep command extends grep's pattern-matching capabilities. It offers all the options of grep, but its most useful feature is the facility to specify more than one pattern for search .each pattern is separated from the other by a | (pipe).
The regular expressions that we have used so far in grep can also be used as patterns in egrep. while grep uses some characters that are not recognized by egrep, egrep's set includes some additional characters not used by grep.

The Regular Expression Set Used by egrep

Expressions	Significance
ch+	matches one or more occurrences of character ch
ch?	matches zero or one occurrences of character ch
exp1|exp2	matches expression exp1 or exp2
(x1|x2)x3	matches expression x1x3 or x2x3

The + and ?:
The extended set includes two special characters--- + and ?.The + matches one or more instances of the previous character, while ? matches zero or one occurrence of the previous character.

b+ matches **b**,**bb**,**bbb**, etc. while **b?** matches either nothing or a single **b**. These characters restrict the scope of match compared to the *.Using this extended set, you can now have a different regular expression for matching the agarwals.

In the three **agarwal**'s that exist in emp.lst, note that the character g occurs only once or twice. So,gg? Now restricts the expansion to one or two g's only. Similarly, the model **[ar][ar]** can now be replaced by **[ar]+**,which increases the scope of expansion to one or more occurrences of either of the characters **a** and **r**.

```
# egrep –I 'agg?[ar]+wal' emp.lst
2476|anil    aggarwal    |manager    |sales    |05/01/59|5000
3564|sudhir  Agarwal|executive|personnel|07/06/47|7500
0110|v.k.agrawal  |g.m.  |marketing|12/31/40|9000
```

Searching for multiple patterns:
How do you locate both "sengupta" and "dasgupta" from the file, a thing that grep can only do using multiple –e options? Delimit the two expressions with the | and the job is done:

```
# egrep 'sengupta|dasgupta' emp.lst
```

```
2365|barun dasgupta|director|personnel|11/05/47|7800
1265|s.n.dasgupta  |manager|sales  |12/09/63|5600
```
Egrep handles the problem easily, but offers an even better alternative. You can group patterns using a pair of parentheses, as well as the pipe:

```
# egrep '(sen|das)gupta' emp.lst
2365|barun dasgupta|director|personnel|11/05/47|7800
1265|s.n.dasgupta  |manager|sales  |12/09/63|5600
```
The –f option:Storing patterns in a file:

So far, you have seen scanning the file for two patterns at the most. What do you do if there are quite a number of them? egrep offers the –f(file) option to take such patterns from the file. Let's fill up a file with some patterns:

```
# cat pat.lst
admin|accounts|sales
```
This file must contain the patterns, suitably delimiter (with pipes) in the same way as they are specified in the command line. When you execute **egrep** with **–f** option in this way

```
#egrep –f pat.lst emp.lst                    //the command takes the expression from pat.lst.
```
egrep enhances the power of **grep** by accepting both alternative patterns, as well as patterns from a file. What if there are multiple patterns, and none of them is a regular expression, but a fixed string? you can then try **fgrep**

fgrep: MULTIPLE STRING SEARCHING

fgrep, like egrep, accepts multiple patterns, both from the command line and a file, but unlike grep and egrep, doesn't accept expressions. So, if the pattern to search for is a simple string, or a group of them, fgrep is recommended.

Alternative patterns of **fgrep** are specified by separating one pattern from another by a newline character. This is unlike is **egrep**, which uses the | to delimit two expressions. You may either specify these patterns in the command line itself, or store them in a file in this way:

```
# cat pat1.lst
sales
personnel
admin
```
Now you can use **fgrep** in the same way you used **egrep**

```
#fgrep –f pat1.lst emp.lst
```

8. PROCESS MANAGEMENT COMMANDS

A process is simply an interface 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 execution is complete, the process is said to die. This process also has a name, usually the name of the program being executed. For example when you execute the **grep** command, a process named **grep** is created.

It is the kernel (and not the shell) that is ultimately responsible for the management of these processes. It determines the time and priorities that are allocated to processes so that multiple processes are able to share CPU resources. It provides a mechanism by which a process is able to execute for a finite period of time, and then relinquish control to another process.

Typically, hundreds, or even thousands of processes can run in a large systems. Each process is uniquely identified by a number called the PID (process identifier) that is allotted by the kernel when it is born.

The sh process:

When you log into a system, a process is immediately set up by the kernel. This is technically a Linux command which may be **sh** (Bourne shell), **ksh** (Korn shell) or **bash** (Bourne again shell).Any command you type in at the prompt is actually standard input to the shell process(or program).This process remains alive until you log out.

The shell maintains a set of variables that are available to the user. The sh process, identified by its PID, is stored in a special "variable" ##.To know the PID of your current shell, simply type

```
#echo ##                              //The process number of the current shell
659
```

Parents and children:

Just as a file has a parent, every process also has one. This parent itself is another process, and a process born from it is said to be its child. When you run the command

```
#cat emp.lst
```

From the keyboard, a process representing the cat command, is started by the sh process(the shell).This cat process remains active as long as the command is active.sh is said to be the parent of cat, the cat is said to be the child of sh.

Since every process has a parent, you can't have an "orphaned" process (i.e.,a process that has no parent) in Linux system.

Like a file, a process can have only one parent. Moreover, just as a directory can have more than one file under it, the multitasking nature permits a process to generate (or spawn) one or more children. This is most easily accomplished by setting up a pipeline. The command

```
#cat emp.lst | grep 'director'
```

Sets up a two processes for the two commands. These processes have the names cat and grep and both are spawned by sh.

Note: All commands don't sets up a processes. Built-in commands of the shell like pwd.cd etc. don't create processes.

ps: PROCESS STATUS

The ps command is used to display the attributes of a process.It is one of the few commands of the system that has knowledge of the kernel built into it.It reads through kernel data structures or process tables to fetch the process characteristics.

By default, ps lists out the processes associated with a user at that terminal:

```
# ps
PID     TTY    TIME     CMD
476     tty03  00:00:01 login
659     tty03  00:00:01 sh
```

```
684     tty03    00:00:00  ps
```

Like who, ps also generates a header information. 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 has been stated, and the process name(CMD).Linux shows an additional column that is usually an S(sleeping) or R(running).

You can see that your login shell (sh) has the PID 659, the same number echoed by the special variable ##.ps itself is an instance of a process identified by the PID 684.

PROCESS OPTIONS:

Displaying process ancestry (Full listing):

The ps command, by default, doesn't provide about the ancestry of a process. To get them in SCO UNIX, you have to use the –f(full) option:

```
# ps –f
UID    PID    PPID  C     STIME       TTY    TIME       CMD
Root   476    1     0     17:51:58    tty03  00:00:01   /bin/login kumar
Kumar  659    476   4     18:10:29    tty03  00;00:01  -sh
Kumar  685    659   15    18:26:44    tty03  00:00:00    ps –f
```

The PPID (parent PID) is the PID of the parent process that spawned this process. The login shell has the PID 659 and PPID 476,which means that it was set up by a system process having PID 476(the login program).The ps command itself has the PPID 659 that also happens to be the PID of the sh process.

The first column (UID) displays the user's login name. C indicates the amount of CPU time consumed by the process. STIME shows the time the process started. TIME shows the total CPU time used by the process.CMD displays the full command line with its arguments.

Displaying processes of a user (-u):

The user(-u) option lets you know the activities of any user, for instance, the user local:

```
# ps –u local
```

Displaying all user processes (-a):

The –a(all) option lists out the processes of all users, but doesn't display the system processes:

```
# ps –a
PID    TTY    TIME        CMD
662    tty02  00:00:00    ksh
705    tty04  00:00:00    sh
1005   tty01  00:00:00    ksh
1017   tty01  00:00:04    vi
680    tty03  00:00:00    ksh
1056   tty02  00:00:00    sort
1058   tty05  00:00:00    ksh
1069   tty02  00:00:00    ps
```

Five users are at work here, as evident from the terminal names displayed. Most of them seem to be users of the korn shell. Not much work seems to be going on here, except for sorting and a file editing operation with vi.

System processes (-e):

Over and above the processes that a user generates, there are a number of system processes that keep running all the time. Most of them are spawned during system startup, and some of them start when the system goes to the multi-user state. To list them you have to use –e option (or –x).

```
# ps –e
PID    TTY    TIME        CMD
0      ?      00:00:00    sched
1      ?      00:00:01    init
2      ?      00:00:00    vhand
```

```
3       ?      00:00:01      bdflush
260     ?      00:00:00      cron
282     ?      00:00:00      lpsched
308     ?      00:00:00      rwalld
336     ?      00:00:00      inetd
339     ?      00:00:00      routed
403     ?      00:00:00      mountd
408     ?      00:00:00      nfsd
```

The characteristic feature of system processes is that most of them are not associated with any terminal at all (shown by ?)Some of these processes are called daemons that do important work for the system.

RUNNING JOBS IN BACKGROUND:

A multi-tasking system lets user do more than one job at a time. Since there can be only one job in the foreground, the rest of the jobs have to run in the background. The & is the shell's operator used to run a process in the background (i.e., not wait for its death) Just terminate the command line with an &;the command will run in the background:

```
# sort –o emp.lst emp.lst &                    #the job's PID
```

The shell immediately returns a number-the PID of the invoked command (550).The prompt is returned, and the shell is ready to accept another command, even though the previous command has not been terminated yet.

kill: PREMATURE TERMINATION OF A PROCESS

The system often requires to communicate the occurrences of an event to a process. This is done by sending signal to the process. If you have program running longer than you anticipated, or if you have changed your mind and want to run something else, simply send a signal to the active process with a specific request of termination.

You can terminate a process with the kill command. The command uses one or more PID's a the arguments. Thus,

```
kill 105
```

Terminates the job with the PID 105.If you run more than one job in the background, they can all be killed with a single kill statement by specifying the PID's of all the background jobs in the command line:

```
kill 121 122 125 132 138 144
```

If all these processes have same parent, you may simply kill the parent in order to kill all its children.

Killing with signal numbers:

Kill, by default uses the signal 15 to terminate the process. The process can be killed with signal number 9, sometimes known as the sure kill signal. This signal can't be generated at the press of a key, so kill lets you use the signal number as an option:

```
kill -9 121                  #kills process 121 with signal number 9
kill –n -9 121               #same as above
```

You can also kill all processes in your system except the login shell, by using a special argument 0:

```
kill 0
```

This terminates all background processes with the signal number 15.The login shell ignores this signal, so you have to kill it by using any of the following commands:

```
kill -9 ##                   ### stores PID of current shell
kill -9 0                    #kills all processes including the login shell
kill -1 0                    #same
```

Job control in the Korn and Bash shells:
If you are using Korn or Bash shell, you can use its job control facility to manipulate jobs. You can relegate a job to the background, bring it back to the foreground, suspend or even kill it. You often have to resort to these methods when you accept a job to complete in 10 minutes, and it goes on for half an hour. Surely, you won't like to cancel it because a lot of work has been done already. To handle this situation, you will need the built-in bg, fg, suspend and kill commands.

If you have invoked a command and the prompt has not yet returned, you an simply suspend the job by pressing <ctrl-z>.You will then see the following message:

```
[1]  +  Stopped                    spell uxtip02 > uxtip02.spell
```

Mind you, the job has not yet been terminated. Now if you want to do some other work in the foreground, use the bg command to push the last spell checking job to the background:

```
# bg
[1]       spell uxtip02 > uxtip02.spell&
```

The & at the end of the lime indicates that the job is now being run in the background. So,a foreground job goes to the background, first with <ctrl-z>,and then the bg command. You can ofcourse start a job in the background itself:

```
#sort permuted,index > sorted.index&
[2]      530
```

The [2] shows that this is the second job currently running in the background. You can run more jobs in this way:

```
# wc –l uxtip?? > word_count &
[3]      540
```

Now that you have three jobs running, you can have a listing of their status with the jobs commands:

```
# jobs
[3]  +  Running               wc –l uxtip?? > word_count &
[1]  -   Running              spell uxtip02 > uxtip02.spell &
[2]      Running              sort permuted.index > sorted.index &
```

You can bring any of the background to the foreground with the **fg** command. This, along with **bg**, can be used in a number of ways. With three background jobs now running, if you run **fg** without any argument, it will bring the most recent background job, i.e. the **wc** command, to the foreground.

fg and **bg** optionally also use a job identifier prefixed by a % symbol. If you specify fg %1, the first job is brought to the foreground. If you use **fg %sort**, the **sort** job comes to the foreground. If you use fg %sort, the sort job comes to the foreground.

You can also use the **stop** command to kill a background job. However, unlike the fg command which, by default, brings the most recent job to the foreground, stop needs one or more job identifiers. Thus, **stop %1** terminates execution of the first job, while **stop %sort** halts execution of **sort** command.

At any time however you can kll any background job with the buil-in kill command. Like stop it also uses same type of job identifiers, and optionally uses signal number. Thus **kill -9 %1** kills the first background job with signal number 9 while **kill -15 %spell** terminates the spell checking program.

At this point, it must be mentioned that the reason why we used <ctrl-z> for suspending a job is that,by default, this is the character set by the **stty** command.

9. INTRODUCTION TO SHELL PROGRAMMING

All shell statements and Linux commands can be entered in the command line itself. When group of commands has to be executed regularly, they are better stored in a file. All such files are called shell scripts, shell programs, or shell procedures. A shell script is an executable file which contains shell commands. The script act as a "program" by sequentially executing each command in the file. A scripting language consists of control structures, shell commands, variables and expressions.

USES OF SHELL SCRIPT:

- Customizing administrative tasks.
- System boot scripts.
- Creating simple applications.
- To kill or start multiple applications together.
- Data backup and creating snapshots.
- Monitoring your linux system.
- Whenever you find yourself doing the same task over and over again you should use shell scripting, i.e. repetitive task automation.

SHELL SPECIAL CHARACTERS:

We begin with the special set of characters that the shell uses to match filenames. We have used commands with more than one filename as arguments (e.g. cat chap01 chap02). Often, you may need to enter multiple filename in a command line. The most obvious solution is to specify all the filename separately:

```
ls chap chap01 chap02 chap03 chap04 chapx chapy chapz
```

If filenames are similar (as above) ,we can use the facility offered by the shell of representing them by a single pattern. For instance, the pattern chap represents all filenames beginning with chap.
This pattern is framed with ordinary characters (like chap) and a metacharacters (like) using well-defined rules. The pattern can then be used as an argument to the command, and the shell will expand it suitably before the command is executed. The metacharacters that are used to construct the generalized pattern for matching filenames belong to a category called wild-cards.
The * and ? : The meta characters,*, is one of the characters of the shell's wild-card set. It matches any number of characters

```
#ls chap*
Chap chap01 chap02 chap03 chap04 chapx chapy chapz
```

The next wild-card is the ?, which matches a single character. When used with same string chap (chap?) the shell matches all five-character filenames beginning with chap. Appending another "?" creates the pattern chap??, which matches six-character filename.

```
#ls chap?
chapx chapy chapz
#ls chap??
chap01 chap02 chap03
```

Matching the dot (.): If you want to list all hidden filenames in your directory having at least three characters after the dot, the dot must be matched explicitly.

```
#ls .???*
.bash_profile .exre netscape profile
```

The character class:

The character class comprises a set of characters enclosed by the rectangular brackets, [&], but it matches a single character in the class. The pattern [abcd] is a character class and it matches a single character- a, b, c or d. This can be combined with any string or another wild-card expression, so selecting chap01, chap02 and chap04

```
#ls chap0 [124]
chap01 chap02 chap04

#ls chap[1-4]           lists chap01, chap02, chap03 and chap04
#ls chap[x-z]           lists chapx chapy chapz
```

COMMENTS:

Comments help to make your code more readable. They do not affect the o/p of your program.
They are specially made for you to read. All comments in bash begin with the hash symbol: "#", except for the first line (#!/bin/bash). The first line is not a comment. Any lines after the first line that begin with"#" are a comment.

```
#!/bin/bash
#this program counts from 1 to 10
for i in 1 23 4 56789 10; do
echo Si
done
```

COMMAND SEPARATOR [SEMICOLON]:

The semicolon (;) is used as a command separator. You can run more than one command on a single line by using the command separator, placing the semicolon between each command.

```
#date; who am i
Mon Mar 27 19:15:41 PDT 2017
Root tty/s Mar 27 17:50
```

ESCAPING AND QUOTING:

Escaping: Providing a \ (backslash) before the wild-card to remove its special meaning. For instance, in the patter, the \ tells the shell that the asterisk has to be matched literally instead of being interpreted as a metacharacters.

```
rm chap\*               doesn't remove chapl chap2
```

The \ suppresses the wild-card nature of the, thus preventing the shell from performing filename expansion on it. This feature is known as escaping.

QUOTING:

Enclosing the wild-card, or even the entire pattern, within quotes (like ‘chap*’). Anything within these quotes are left alone by the shell and not interpreted. There's another way to turn off the meaning of a meta characters. When a command argument is enclosed in quotes, the meaning of all enclosed special characters is turned off.

```
#echo ‘\’               displays a V
rm 'chap*               removes file chap
```

The following example shows the protection of four special characters using single quotes:

```
#echo the character |, <,> and S are also special'.
The character, <,> and # are also special.
```

COMMAND SUBSTITUTION:

Command substitution is used to combine more than one command in a command line. The “command” construct makes available the output of command for assignment to a variable. This is also known as "back quotes" or "back ticks". The shell enables one or more command arguments to be obtained from the standard o/p of another command. This feature is called command substitution.

```
#echo "the today's date is 'date
The today's date is sat sep 7 19:01:59 IST 2002
```

You can use the two commands in a pipeline and then use the output as the argument to a third.

```
#echo "there are 'ls wc - I' files in the current directory"

There are 58 files in the current directory.
```

Command substitution is enabled when back quotes are used within double quotes. If you use single quotes, it's not.

CREATING SHELL SCRIPT:

Following steps are required to write shell script:

1. Use any editor like vi or mcedit to write shell script.
   Syntax: vi script name.sh
   Examples: vi pgml.sh
2. After writing shell script set execute permission for your script as follows
   Syntax: chmod permission your-script-name
   Examples: #chmod +x your-script-name
   #chmod +x pgml.sh
3. Execute your script as
   Syntax: sh your-script-name
   ./Your's-script-name
   Examples: #. /pgml.sh

SHELL VARIABLES:

The shell supports variables that are useful both in the command line and shell scripts. A variable assignment is of the form variable value (no spaces around-), but its evaluation requires the S as prefix to the variable name:-

```
#count =5          no # required for assignment
#echo #count       but needed for evaluation
5
```

A variable can also be assigned the value of another variable:

```
#total=#count          Assigning a value to another variable
#echo #total
```

All shell variables are initialized to null strings by default. While explicit assignment of null strings with or x=” ” or x=’ ’ is possible, you can also use this as a shorthand.

```
X=          a null string
```

A variable can be removed with unset and protected from reassignment by read-only. Both are shell internal commands.

```
Unset x          x is now undefined
Read-only x      x can't be reassigned
```

COMMAND LINE ARGUMENTS AND POSITIONAL PARAMETERS:

Shell procedures accept arguments in another situation to0-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 variable I that is evaluated with S.

```
#cat emp2.sh
#/bin/sh
#
echo "Program: #0                          # #0 contains the program name
The number of arguments specificd is ##
The arguments are #*"                      #All arguments stored in #
grep "S!" #2
echo "\nJob Over"
```

The parameter #* stores the complete set of positicnal parameters as a single string.
The parameter ## is set to the number of arguments specified. This lets you design scripts that check whether the right numbers of arguments have been entered.
The parameter #0 holds the command name itself.
Invoke this script with the pattern "director" and the filename empl.lst as the two arguments:

```
#emp2.sh director empl.lst
Program: emp2.sh
The number of arguments specified is 2
The arguments are director emp1.lst
1006 |chanchal singhvi |director |sales |103/09/38 |6700
6521 |lalit chowdary |director |marketing 26 09/45|8200
Job Over
```

When arguments are specified in this way, the first word (the command itself) is assigned to S0, 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(and using shift statement you can go beyond).

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
# grep manager emp3.lst > dev/null;echo #?
grep:can't open emp3.lst
2
```

The logical operators && and || Conditional execution:

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"
1006 |chanchal singhvi |director|sales|03/09/38|6700
6521 lalit chowdury |director |marketing 26/09/45 8200
Pattern found in a 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.Ist || echo "Pattern not found"
```

Pattern not found

EVALUATING EXPRESSION:

expr: computation and string handling:

The Bourne shell can check whether an integer is greater than another, but it doesn't have any computing features at al. it has to rely on the external expr command for that purpose. This command combines two functions in one:

- Performs arithmetic operations on integers.
- Manipulates strings.

Computation:

expr can perform the four basic arithmetic operations as well as the modulus functions:

```
# x- 3 y= 5            multiple assignments without a;
#expr 3 +5
8
#expr #x - #y
-2
#expr 3 \* 5           asterisk has to be escaped
15
#expr Sy/Sx            decimal portion truncated
1
#expr 13 % 5
3
```

String handling:

For manipulating strings, expr uses two expressions separated by a colon. The string to be worked upon is placed on the left of the:, and a regular expression is placed on its right. Depending on the composition of the expression, expr can perform three important string functions:

- Determine the length of the string
- Extract a substring
- Locate the position of a character in a string.

The length of a string: The length of a string is a relatively simple matter; the regular expression signifies to expr that it has to print the number of characters matching the pattern, i.c., the length or the entire string:

```
#expr "abcdefghijkl" :  ‘.*’          space on either side of : required
12
```

Here, expr has counted the number of occurrences of any character (.*).

Extracting a substring: expr can extract a string enclosed by the escaped characters \ (and). If you Wish to extract the 2-digit year from a 4-digit string, you must create a pattern group and extract it this way:

```
#stg-2003
#expr "$stg" : ‘..\(..\)"          extracts last two characters
03
```

It signifies that the first two characters in the value of Sstg have to be ignored and two characters have to be extracted from the third character position

Locating position of a character: expr can also return the location of the first occurrence of a character inside a string. To locate the position of the character d in the string value of #stg, you have to count the number of characters which are not d([^d]*),followed by a d:

```
#stg= abcdefgh ; expr “#stg” : ‘[^d]*d’
4
```

10. SHELL CONTROL STRUCTURES

THE if CONDITIONAL:

The if 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.

```
if condition is true                    if condition is true
then                                    then
        execute commands                        execute commands
else                                    fi
        execute commands
fi
        (Form 1)                        (Form2)
```

```
# if grep "director" emp.lst
> then echo "pattern found-job Over"
> else echo "Pattern not found"
> fi
```

grep command is executed it's output displayed, and it's return value used as input to if. This condition placed in the command line of the if statement referred as the "control command". We can used in executable program.

THE case CONDITIONAL:

The case statement is the second conditional offered by the shell. The statement matches an expression for more than one alternative and uses a compact way to permit multi-way branching .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 a 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-columns, and the entire construct is closed with esac.

# cat menu.sh

```
#!/bin/sh
echo " MENU\n
1.   List of files\n2. processes of user\n3. Today's date
4. Users of system\n5. Quit to UNIX\nEnter you option : \c"
read choice
case "#choice" in
        1) ls –l ;;
        2) ps –f ;;
        3) date ;;
        4)who                                                   ;;
        5)exit
esac
```

for: LOOPING WITH A LIST

The for loop is different in structure from the once used in other programming languages unlike while and until it doesn't taste a condition but uses a list instant. The
Syntax of construct is as follows:
Syntax1:

```
        for variable in list
         do
         execute commands
         done
```

The loop body is executed as many time as items in the list

```
ex:- for name in ruby Samuel
do
echo "#name"
done
```

Syntax2:

```
         for((exp1;exp2;exp3))
        do
        execute commands
        done
```

In the syntax given above expression1 is evaluated once before the loop is execute expression2 is evaluated before each iteration if expression2 evaluates the for loop terminals, expression3 is evaluated each time the loop is terminated.

```
        if ((a=1;a<=10;a=a+1))
        do
                echo '#a'
        done
```

while: LOOPING

Loops let you perform set of instructions repeatedly. The shell features three types of loops- while, until and for. All of them repeat the instruction set enclosed by certain keywords. While performs a set of instructions till the control command returns true exit status. The general syntax of this command is as follows :

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

The commands enclosed by do and done are executed repeatedly as long as condition remains true.

```
#/bin/sh
#
answer = y
while [ "#answer" "y" ]
do
        echo "enter the code and description: c">dev/tty
        read code description
        echo "#code|#description">> newlist
        echo "enter any more (y/n)? 'c" >/dev/tty
        read anymore
        case #anymore in
                y Y*) answer y:             #Also accepts yes, YES etc
                n*N*) answer n::            #Also accepts no, NO etc
```

```
            *) answer y ::                    #Any other reply means y

        esac
        done
```
#emp5.sh
```
Enter the code and description: 03 analgesics e
Enter anymore (y'n)? y
Enter the code and description: 04 antibiotics
Enter anymore (y n)? y
Enter the code and description: 05 OTC drugs
Enter anymore (y/n)? n
```
#cat newlist
```
03 analgesics
04 antibiotics
05 OTC drugs
```

RELATIONAL AND LOGICAL OPERATORS

Relational operators:

Bourne shell supports the following relational operators that are specific to numeric values.

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

These operators do not work for string values unless their value is numeric. It is very important to understand that all the conditional expressions should be placed inside square braces with spaces around them.

For example:

[#a <=#b] is correct whereas (#a<=#b] is incorrect

Relational operators:

Here is an example which uses all the relational operators. Assume variable a holds 10 and variable b holds 20:
```
        #cat rel.$h
        #!/bin/$h
        a 10
        b=20
        if [ #a -eq #b ]
        then
                echo "$a -eq #b: ai$ equal tob
        el$e
                echo "$a -cq $b: a i$ not cqual to b
        fi
        if [ $a -ne $b]
        then
                echo "$a -ne $b: a i$ not equal to b
        el$e
                echo "$a -ne $b: a i$ equal to b
        fi
```

```
if ( $a gt $b]
then
        echo "$a -gt $b: a i$ greater than b
el$e
        echo "$a -gt $b: a i$ not greater than b
fi
if [ $a -It $b]
then
        echo "$a -lt $b: a i$ le$$ than b
el$e
        echo "$a -lt $b: a i$ not le$$ than b
fi
if [ $a ge $b]
then
        echo "$a -ge $b: a i$ greater than or equal to b
el$e
        echo "$a -ge $b: a i$ not greater than or equal to b
fi
if [ $a -le $b]
then
        echo "$a -le #b : a i$ le$$ than or equal to b
el$e
        echo "$a -le #b: a i$ not le$$ than or equal to b
fi
```

#chmod +x rel.$h
#./rel.$h

The above script will generate the following result:

```
10-eq 20: a is not equal to b
10-ne 20: a is not equal to b
10-gt 20: a is not greater than b
10-lt 20: a is less than b
10-ge 20: a is not greater than or equal to b
10-le 20: a is less than or equal to b
```

Logical operators:
Here are logical operators that are used during shell script:

Operator	**Description**
cond1 -a cond2	True if condition1 and condition2 are true(perform AND operation)
cond1 -o cond2	True if conditional and condition2 are true(perform OR operation)
!cond1	True if conditional is false

Example:

#cat log.sh

```
#! bin bash
a=25
b-29
if [ Sa-gt 20-a Sb-gt 25 ]: then
```

```
echo "both condition satisfied"
fi
if [ Sa-gt 25 -o Sb-gt 25 ]; then
echo "only one condition is satisfied"
fi
#chmod +x log.sh
#./log.sh
Both condition satisfied
Only one condition is satisfied.
```

ADVANCED FILTERS

sed -THE STREAM EDITOR:

sed is a multipurpose tool combines work of several filters. Designed by Lee McMahon.sed is used for performing non-interactive operations. It acts on a data stream, hence its name. If you recall the filters you have learnt so far you'll find that the behaviour is wholly governed by its options you use, there was nothing else. But sed has very few options and its power is derived from the ease with which you can both select lines and frame instructions to act on the selected lines. It has an enormous number of features, almost bordering on a programming language.

A sed instruction:

Everything in sed is an instruction. An instruction combines an address for selecting lines with an action to be taken on them as shown by the syntax:

```
sed options 'address action' file(s)
```

sed has two ways of addressing lines:

*By line number
*By specifying a pattern which occurs in a line

In the first form, address specifies either one line number to select a single line, or a set of two to select a group of contiguous lines. Likewise; the second form can use one or two patterns. The action can be either insertion, deletion or substitution of text (table).sed has its own family of internal commands which transform selected lines/text in several ways.

Line addressing:

sed follows the line addressing as, you either specify a line or a pair of them ,to limit the boundaries of the selection. The action is appended to this address. For instance, the instruction 3q can be broken to the address 3 and action q(quit).When the instruction is enclosed within quotes and followed by one or more filenames, you can simulate head -3 in this way:

```
# sed '3q' emp.lst                                  #quits after line no. 3
2233|a. k. shukla        |g.m       |sales       |12/12/52|6000
9867|jai sharma          |director  |production|12/03/50|4000
3456|sumit chakrobarty|d.g.m  |marketing |19/04/43|6000
```

sed also uses the p(print) command to print the output.But notice what happens when you use two line addresses along with p command:

```
# sed '1,2p' emp.lst
2233|a. k. shukla       |g.m      |sales       |12/12/52|6000
2233|a. k. shukla       |g.m       |sales       |12/12/52|6000
9867|jai sharma         |director  |production|12/03/50|4000
9867|jai sharma         |director  |production|12/03/50|4000
3456|sumit   chakrobarty|d.g.m   |marketing   |19/04/43|6000
2345|n.k.gupta          |chairman|personnal  |11/03/47|7800
1008|chanchal   singhvi   |director   |sales   |03/02.90|4500
2456|anil aggarwal  |manager  |sales  |01/09/90|4000
```

….more lines with each line displayed only once.

You probably intended to select first two lines of emp.lst with the instruction 1,2p. Instead,the first two lines have been printed twice. Hence we can know that sed with its p command by default prints all lines on the standard output, in addition to all lines affected by action.

The –n option: suppressing duplicate line printing

To overcome problem of printing duplicate lines, you should use the –n option whenever you use the p command. Thus previous command should have been written as follows:

```
#sed –n '1,2p' emp.lst
2233|a. k. shukla       |g.m       |sales       |12/12/52|6000
9867|jai sharma         |director  |production|12/03/50|4000
```

And to select last line of file,use #

```
#sed –n '#p' emp.lst
2456|anil aggarwal      |manager   |sales       |01/09/90|4000
```

Negating the action (!):

sed also has a negation operator(!),which can be used with any action. For instance, selecting the first 2 lines is just same as not selecting lines 3 through the end. The next to previous command sequence can be written in this way too:

Selecting lines from anywhere:

```
#sed –n '3, #!p' emp.lst                    #Don't print lines 3 to the end
```

The address and action are enclosed with a pair of single quotes.

Selecting lines from middle:

sed can also select lines from the middle of a file,to select lines from 9 through 11,you have

```
#sed –n '9-11p' emp.lst
```

Selecting multiple group of lines:

sed is not restricted to selecting contiguous groups of lines.

```
#sed –n '1-2p              #3 addresses in one command, using only a single
7,9p                       #pair of quotes
#p' emp.lst                #selects last line
```

Using multiple instruction:

There is also an alternative method of entering previous sequence of instructions. The –e option allows you to enter as many instructions as you wish

```
#sed –n –e '1-2p' –e '7-9p' –e '#p' emp.lst
```

Inserting and changing text:

sed can also insert new text and change existing text in a file. If you remember doing similar things in vi.sed uses i(insert),a(append) and c(change) commands. But there are important differences too. Let's append 2 records to input on emp.lst and store new output in empnew.lst

```
#sed '#a\
>2033|a. k. bose|g.m      |sales      |1612/52|6500\
>9877|jayant    |director |production|12/03/50|4500
>' emp.lst > empnew.lst
```

First enter instruction #a which appends text at the end of file. Then enter \ before pressing the <enter> key. You can key in as many lines as you wish. Each line except the last line has to be terminated by \ before hitting the <enter> key.sed identifies line without \ as the last line of input.

Double spacing text:

Consequence of not using an address with these commands. The inserted or changed text then is placed after or before every line of file

```
sed 'i\                                     #inserts before every line
                                            #this is blank line
' emp.lst
```

Inserts a blank line before each line of file printed. This is another way of double spacing text, **a** would insert a blank line after each selected line.

Context addressing:

The second form of addressing lets you specify a pattern as well. When you specify a single pattern all lines containing the pattern are selected.

```
# sed -n '/director/p' emp.lst
```

This method of addressing lines by specifying context is known as context addressing. You can also specify a comma-separated pair of context addresses to select group of lines. Line and context addresses can also be mixed.

```
# sed –n '/dasgupta/,/saxsena/p' emp.lst
# sed –n '1,/dasgupta/p' emp.lst
```

Deleting lines:

Using d(delete) command, sed can evaluate grep's –v option to select lines not containing the pattern. Either of following commands

```
# sed '/director/d' emp.lst > olist          # -n option not to be used with d
# sed -n '/director/!p' emp.lst > olist
```

Selects all but lines containing "director" and save them in list.

Writing selected lines to a file:

The (w) write command makes it possible to write selected lines in a separate file. Save the records of directors in a disk in this way.

```
# sed –n '/director/w dlist' emp.lst              #no display when using –n option
```

Since sed accepts more than one address, you can perform a full context splitting of file emp.lst. You can also have

```
# sed –n '/director/w dlist
          /manager/w mlist
         /executive/w elist ' emp.lst
```

Substitution(s):

sed's strongest feature is substitution, achieved with its s(substitute) command.
It lets you replace a pattern in its input with something else.
The syntax for such a command can be described as
[Address] s/string1/string2/flag
Here string1 will be replaced preferred by string2 in all lines specified by the address. If address is not specified substitution will be preferred by all lines containing string1. Using this syntax let's first replace word "director" by "member" in first five lines of emp.lst

```
# sed '1,5s/director/member/' emp.lst.
```

To broaden search so that it affects all lines you may either use global address 1, # or simply drop address altogether. In the absence of an address sed acts on all lines. i.e.

```
sed '1,#s/director/member/' emp.lst
sed 's/director/member/' emp.lst
```

Remember that in absence of –n and p options, all lines are displayed whether substitution has been preferred or not.

sed also uses regular expressions for patterns to be substituted. To replace all occurrences of "agarwal", "aggrawal" and "aggarwal" by simply "Agarwal" use regular expression as

```
# sed –n 's/[Aa]gg*[ar][ra]wal/Agarwal/p' emp.lst
```

This time –n and p command ensured that only lines affected by substitution are displayed; unaffected lines are suppressed.

Global substitution:

Suppose you require to replace all occurrence of director character in first line by #. See what happens when you use following command.

```
# sed 's/ | / # /' emp.lst | head -3
2233#a. k. shukla      |g.m     |sales        |12/12/52|6000
9867#jai sharma        |director  |production|12/03/50|4000
9867#jai sharma        |director  |production|12/03/50|4000
```

This only replaces first occurrence of the | in all lines other pipes remain unaffected. All the previous substitutions also acted on first occurrences in the line, you couldn't know that because string "director" never occurred more than once in same line. To replace all occurrences, you need to use g (global) flag at the end of instruction. Referred to as global substitution. i.e.

# sed 's/ | / # /g' emp.lst | head -3

```
2233#a. k. shukla          #g.m     #sales            #12/12/52#6000
9867#jai sharma            #director  #production#12/03/50#4000
9867#jai sharma            #director  #production#12/03/50#4000
```

Compressing multiple spaces:

How do you delete trailing spaces from second, third and fourth fields? Regular expression required in source string needs to signify one or more occurrences of a space, followed by a |:

# sed 's/ *|/ | /g ' emp.lst | head -2

```
2233|a.k.agarwal|g.m|sales|12/12/1990|1200
9876|jai sharma|director|production|13/12/2000|5000
```

The remembered pattern:

There's another way of performing substitution. Scan entire file for say string "director" and replace first occurrence in each line matched.

sed –n '/director/s/director/member/p' emp.lst

Here the address /director/ appears to be redundant; without using it will have same output. However, you must understand this form also because it widens scope of substitution. It's possible that you may like to replace a string in all lines containing a different string. For example, you may like to change the designation of only that director who heads the following marketing function. Use this form to scan the file for pattern "marketing" and then perform required substitution.

# sed –n 'marketing/s/director/member/p' emp.lst

```
1234|lalit chowdury   |member   |marketing |09/12/45|2500
```

The above sequence however be condensed if the scanned pattern is same as the substituted pattern i.e. source string. This was the case in next to previous example.sed remembers scanned patterns and lets you use // (two front slashes) as source string if it's same as the scanned pattern. You then need not to specify pattern there. Two forms are then equivalent.

sed 'director/s//member' emp.lst

sed 's/director/member' emp.lst

The second form, though require two fewer keystrokes. This empty (or null) regular expression enclosed within the two front slashes and is understood by sed to represent search pattern.

The repeated pattern (&):

When a pattern is the source string also occurs in replacement, you can use special characters & to represent it. Both the following commands do same thing.

sed 's/director/executive director &/' emp.lst

sed '/director/s//executive &/' emp.lst

The &, known as repeated pattern, expands to the entire source string. The & is the only other special character you can use in replacement string.

Internal commands used by sed:-

Commands	Description
i	inserts before line
a	appends after line
c	changes line(s)
d	deletes line(s)
p	prints line(s)on standard o/p
q	quits after reading up to addressed line
=	prints line number addressed
s/s1/s2	substitute string1 by string2
r flname	places contents of the file flname after line
w flname	writes addressed lines of file flname

awk: AN ADVANCED FILTER

awk is an interpreted programming language which focuses on processing text. It was designed to execute complex pattem-matching operations on streams of textual data. It makes heavy use of strings, associative arrays and regular expressions.

SIMPLE awk FILTERING

awk is a little awkward to use at first, but if you feel comfortable with sed, then you'll find a friend in awk. The syntax is as follows:

```
awk options 'selection_criteria {action}' file(s)
```

The selection criteria filters input and selects lines for the action component to act upon. This component is enclosed within curly braces. The selection criteria and actions constitute an awk program that is surrounded by a set of single quotes.

A typically complete awk command specifies the selection criteria and action. The following command selects the directors from emp.lst

```
#awk ‘/director/ {print}' emp.lst
9876|jai Sharma          |director     |production    |12/03/50     |7000
2345|barun sengupta      |director     |personnel     |11/05/47     |7800
1006|chanchal singhvi    |director     |sales         |03/09/38     |6700
6521|lalit chowdary      |director     |marketing  |26/09/45  |8200
```

The selection_criteria section (/director/) selects lines that are processed in the action section ({print}). An awk program must have either the selection criteria or the action, or both, but within single quotes. Double quotes will create problems unless used judiciously. For pattern matching, awk uses regular expressions in sed-style:

```
#awk –F”|” ‘/sa[kx]s*ena/’ emp.lst
3212|shyam  saxena     |d.g.m  |accounts     |12/03/50      |6000
2345|j.b. saksena  |g.m |marketing  |11/05/47  |8000
```

SPLITTINGA LINE INTO FIELDS

awk uses the special parameter, #0, to indicate the entire line. It also identifies tields by S1, #2, #3. awk uses a contiguous sequence of spaces and tabs as a single delimiter. You can use awk to print the name, designation, department and salary of all the sales people.

```
#awk -F"" /sales/ {print S2, 53, #4, #6} emp.st
a.k.shukla            .m           sales   6000
chanchal singhvi     director      sales   6700
s.n.dasgupta         manager       sales   5600
anil Aggarwal        manager       sales   5000
```

Notice that a, (comma) has been used to delimit the field specifications. This ensures that each field is separated from the other by a space.

If you want to select lines 3 to 6, all you have to do is use the built-in variable NR to specify the line numbers:

```
#awk –F”|” ‘NR==2,  NR==5 {print NR,#2,#3,#6}’ emp.lst
2 nk.gupta           chairman       5400
3 v.k.agrawal        g.m            9000
4 j.b.saxena         g.m            8000
5 sumit chakraborty  d.g.m          6000
```

NR is one of those built-in variables used in awk programs, and is one of the many operators employed in comparison tests.

printf: FORMATTING OUTPUT

The C-like printf statements, you can use awk as a stream formatter, awk accepts most of the formats used by the printf function used in C, but the %s format will be used for string data, and d for numeric.

```
#awk -F"|" ‘/[aA]gg?[ar]+wal/ {
>printf “%3d %-20s %-12s %d\n", NR, S2, #3, S6 }' emp.lst
```

```
4 v.k.aggarwal       g.m          9000
9 sudhir agarwal     executive    7500
15 anil aggarwal     manager      5000
```

Number Comparison

sed can also handle numbers - both integer and floating type - and make relational tests on them. You can now print pay- slips for those people whose basic pay exceeds 7500:

```
#awk –F”|” ‘#6> 8000 {
>printf  “%-20s %-12s %d\n", #2, #3, #6 }' emp.lst
v.k.agrawal          g.m          9000
lalit chowdary       director     8200
```

You can also combine regular expression matching with numeric comparison to locate those, either born in 1945 or drawing a basic pay greater than 8000:

```
#awk –F”|”  ‘#6>8000 || #5-/45#/’ emp.lst
0110 |v.k.agrawal       |g.m          |marketing       |31/12/40        |9000
2345 |j.b.saxena        |g.m          |marketing       |12/03/45        |8000
6521 |lalit chowdary    |director     |marketing  |26/09/45  |8200
```

THE BEGIN AND END SECTIONS

If you have to print something before processing the first line, for example, a heading, the BEGIN section can be used quite gainfully. Similarly, the END section is useful in printing some totals after processing is over.

The **BEGIN** and **END** sections are optional and take the form

```
BEGIN {action}               both require curly braces
END {action}
#vi empawk.awk
BEGIN {
printf "tit Employee Abstract \n'n"
} #6 > 7500 {
kount++; tot+=#6
printf "%3d %-20s %-12s %d'n", kount, #2, #3, S6
}
END {
printf "In\t The average basic pay is %6din", tot/kount
}
#chmod +x empawk.awk
```

The BEGIN section here prints a suitable heading, offset by two tabs (tt), while the END section prints the average pay (tot/kount) for the selected lines. To execute this program, use the-f option:

```
#awk -F" -f empawk.awk emp.lst
        Employee abstract
1 v.k.agrawal        g.m          9000
2 j.b.saxena         g.m          8000
```

ARRAYS

An array is also a variable except that this variable can store a set of value or elements. Each element is accessed by a subscript called the index. Array are indexed using numbers, they usually start at 0 and go to N-1 the number of element in an array.

Example:

#cat array.awk

```
#!/usr/bin/awk -f
BEGIN {
some_array[1]- "hello"
some_array[2]- "everybody"
some_array[3] = "!"
print some array[1], some array[2], some_array[3]
```

Output: chmod +x array.awk

```
./array.awk
Hello everybody!
```

Associative (HASH) Arrays

awk arrays are associative, where information is held as key-value pairs often referred to as hash. The index is the key that is saved internally as a string. Instead of using a fixed integer to index the array, you can use a value, a string, to identify each element in the associative array. When we set an array element using mon[1]="mon", awk converts the number 1 to a string. Just type the below program directly in the terminal and execute it:

```
#awk BEGIN
>direction["N"]="North"; direction["S"]="South":
>direction["E"]="East"; direction["W]="West";
>printf("N is %s and W is %s\n", direction ["N"], direetion["W"]):
>mon[1]="jan"; mon["1"]="January"; mon ["01"]="JAN";
>printf("mon [1] is %s\n", mon[1]):
>printf("mon[01] is also %s\n", mon[01]):
>printf("mon[ \"1\" ] is also %s\n", mon["1"]):
>printf("but mon[\"01\"] is %s\n", mon["01"]):
>}'
```

Output:

```
N is North and W is West
mon [1] is January
mon [01] is also January
mon ["1"] is also January
But mon ["01"] is JAN
```

ENVIRON []: The Environment Array

awk maintains the associative array, ENVIRON [ ], to store all environment variables. This POSIX requirement is met by recent versions of awk including nawk (new awk) and gawk (GNU awk).Just type the below command in the terminal and execute it.

```
#nawk 'BEGIN {
>print "HOME" "=" ENVIRON [*HOME"]
>print "PATH" "=" ENVIRON ["PATH"]
>}'
```

Output:

```
HOME=/users1/home/staff/sumit
PATH=/usr/bin :: /usr/local/bin :: /usr/ccs/bin
```

11. LINUX SYSTEM ADMINISTRATION

DISK MANAGEMENT UTILITIES:

No matter how many disks are added to the system, there will always be a scramble for space. Users often forget to remove the file they no longer require. Even otherwise, there are a number of files, especially in the directories /tmp and /usr/tmp that tend to accumulate during the day. If this build-up is not checked, the entire disk space will eventually be eaten up, resulting in a slowdown of system functioning.

Linux has a number of commands that can aid you in this task like the **df** and **du** commands, which can also be issued by any user. Both these commands report disk usage or the space in terms of blocks, the minimum unit of measurement of disk space, which is 1024 bytes in Linux.

df: Reporting free space

df(disk free) reports the amount of free space available on disk.The output always reports for each file system separately:

```
# 	df
```

du:Disk usage:

You will often need to find out the consumption of a specific directory tree rather than an entire file system. The du(disk usage) command reports usage by a recursive examination of the directory tree.

By default, du lists the usage of each sub-directory of its argument, and finally produces a summary. The list can often be quite big, and more often than not, you may be interested only in a single figure that takes into account all these sub-directories. For this, the -s (summary) option is quite convenient:

```
# du /home/sales/tml
```

User management:

The term user is not meant to be only a person; it can mean the name of a project or an application, where a group of users having similar functions can use the same username to use the system. It is thus quite common to have username like marketing, accounts, mis, etc.

The Linux provides a number of services related to the creation and maintenance of user accounts. This includes setting of the new user's password. For command line experts **useradd**, **usermod** and **userdel** commands.

useradd:Adding users

From time to time, you have to add users to the system. You have to associate each user with a group and decide the permissions that have to be set for the user and her group. Adding a user involves setting of the parameters, most of them in **/etc/passwd**:

- A user identification number(UID) and username
- A group identification number(GID) and group name
- The login shell
- The mailbox
- The password

usermod and userdel: Modifying and removing users

usermod is used for modifying some of the parameters set with useradd. Users sometimes need to change their login shell, and the following command line sets the C shell as the login shell for the user oracle:

```
usermod –s /bin/csh oracle
```

Users are removed from the system with **userdel**. The following command removes oracle from the system:

```
userdel oracle
```

fdformat: Formatting floppy diskettes

Before you use a floppy for backup purposes, you need to format it first. This is done with the fdformat with the raw device name as argument:

```
fdformat /dev/fd0h1440
```

This command formats a 1.44MB floppy. The formatting process is followed by verification where most of the errors are encountered.

12. LINUX ENVIRONMENT

ENVIRONMENT VARIABLES

Environment variables are a set of dynamic named values which affect the processes or program on a computer. It exists in every operating system i.e. the types may vary. It can be created, edited, saved and deleted. It gives information about the system behavior. Environment variables can change the way software or programs behave.
Shell variables are of two types - local and environment. PATH, HOME and SHELL are **environment variables**. They are so called because they are available in the user's total environment- the sub-shells that run shell scripts, and mail commands and editors.
#echo #PATH-A list of directory paths. When the user types a command without providing the full path, this list is checked to see whether it contains a path that leads to the command.
#echo #HOME-indicates where a user's home directory is located in the file system.
#echo #LOGNAME-this variable shows your username. When you wander around in the file system, you may sometimes forget your login name. From time to time, just make sure you know which account you logged in to.
#echo #PWD-This variable points to the current directory. Equivalent to the output of the command pwd when called without arguments.

COMMAND HISTORY

Bash and kom support a versatile history feature that treats a previous command as an event and associates it with an event number. Using this number, you can recall previous commands, edit them if required and re-execute them. The shell may also saye all-commands in a history life.
The history command displays the history list showing the event number of every previously executed command. A command is recalled by using this event number with a symbol like (! or r).
While bash displays the complete history list with history.

```
#history 5 bash
```

The output shows the event number and command associated with that number and it could look like this:

```
35 ps
36 Is -1
37 pwd
38 history 5
```

TERMINAL VARIABLE STTY COMMAND AND ITS OPTION

tty: KNOWING YOUR TERMINAL

Since UNIX treats even terminals as files, it's reasonable to expect a command that tells you the filename of the terminal you are using. It's the tty (teletype) command, an obvious reference to device that has now become obsolete. The command is simple and needs no arguments:

```
#tty
/dev/pts/10
```

The terminal filename is in 10 resident in the pts directory. This directory in turn is under the /dev directory.

stty: DISPLAYING AND SETTING TERMINAL CHARACTERISTICS

The stty command helps straighten these things out; it both displays and changes settings. Sty uses a very large number of keywords. The -a (all) option displays the current settings. A trimmed output is presented below:

#stty-a

```
Speed 34800 baud; rows =25; column 80; ypixels 0; xpixels -0;
intr c; quit =^\; erase = ^?; kill = ^u;
cof d; col = <undef>; col2 =<undef>, swtch =<undef>;
start q; stop =^s; susp =^z; dsusp =^y;
isig icanon-xcase echo echoe echok -echonl-noflsh
-tostop echoctl-echoprt echoke-defecho-fluso -pendin iexten
```

The setting intr =^c signifies that ctrl -c interrupts a program. The erase character is ctrl -h and the kill character is ctrl -u. The eof (end of file) character is set to ctrl -d.

PROFILES

The Bourne family uses .profile as the login script. As an added feature, the profile used by bash can have one of three names -.bash profile, .profile and .bash login. Issue the Is -a command and see whether you can locate one of these files in your home directory. This login script should be added to your directory at the time of user creation, but you can create one even if it's not there, It is really a shell script that is executed by the shell when one logs on to the system.

.profile is file containing commands that is executed automatically every time user logs in to the system. It may contain any linux command including some system parameter initialization, setting alias, exporting some variables etc. . profile is basically to set user environment in linux, each user can have their own .profile file located at user/home directory.

To see .profile file you can go to /home/username directory, type ls -a (to list all files including hidden files) and then cat .profile

Example of .**profile**:

```
Cat .profile
USER =/root/.profile
MAIL-/var/mail/root
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin and so on
PSI= 'S'
PS2->
TERM=xterm
Today's date is Fri Mar 23 21:57:56 IST 2018
```

The profile contains the commands that are meant to be executed only once in a session.

PART – B

13. Write a shell script to display current date, time, username and directory.

```
#!/bin/bash
#
echo  "hello,  $LOGNAME"
echo "current date is `date`"
echo "user is `who i am`"
echo "current directory is `pwd`"
```

Output:

```
[root@bt~]# vi pgm1.sh
[root@bt~]# chmod +x pgm1.sh
[root@bt~]# ./pgm1.sh
hello, root
current date is Tue Mar 13 05:53:20 IST 2018
user is root pts/0 2018-03-13 05:47 (:0.0)
current directory is /root
```

14. Write script to determine whether given file exist or not, file name is supplied as command line argument, also check for sufficient number of command line argument

```
#!/bin/bash
FILE=$1
if [ -f $FILE ]; then
echo "file '$FILE' Exits"
else
echo "The file '$FILE' does not exists"
fi
```

Output:

```
[root@bt~]# vi pgm2.sh
[root@bt~]# chmod +x pgm2.sh
[root@bt~]# ./pgm2.sh
File ‘ ‘ Exists
[root@bt~]# ./pgm2.sh aaa
The file 'aaa' does not exists.
```

15. Write shell script to show various system configuration like:

- **a) Currently logged user name and his long name**
- **b) Current shell**
- **c) Your home directory**

```
echo "*********System Configuration Details*************"
echo "**************************************************"
echo "Currently logged User Name:$USER"
echo "Log Name:$LOGNAME"
echo "Current Shell Name:$SHELL"
echo "Home directory: $HOME"
echo "**************************************************"
```

Output:

```
[root@bt~]# vi pgm3.sh
[root@bt~]# chmod +x pgm3.sh
[root@bt~]# ./pgm3.sh
*********System Configuration Details*************
**************************************************
Currently logger User Name: root
Log Name: root
Current Shell Name: /bin/bash
Home directory: /root
**************************************************
```

16. Write shell script to show various system configuration like:

- **a) Your operating system type**
- **b) Your current path setting**
- **c) Your current working directory**
- **d) Show all available shells**

```
echo "********System Configuration Details***********"
echo "*********************************************"
echo "Operating system Type:`uname -a`"
echo " "
echo "Current Path Setting:$PATH"
echo " "
echo "Current Working Directory:`pwd`"
echo " "
echo "Show all available Shells: `cat /etc/shells`"
echo "*********************************************"
```

Output:

```
[root@bt~]# vi pgm4.sh
[root@bt~]# chmod +x pgm4.sh
[root@bt~]# ./pgm4.sh
********System Configuration Details***********
*********************************************
Operating System type: Linux bt 3.2.6
Current Path setting:
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/etc/alternatives/gem-
bin:/etc/alternatives/gem-bin.
Current Working Directory: /root
Show all available shells: # /etc/shells : valid login shells
/bin/csh
/bin/sh
/usr/bin/es
/usr/bin/ksh
/bin/ksh
/usr/bin/screen
*********************************************
```

17. Write a Shell script to accept any two file names and check their file permissions.

```
echo "Enter the name of first file:"
read file1
echo "Enter the name of second file"
read file2
fp1=`ls -l $file1|cut -c 2-10`
fp2=`ls -l $file2|cut -c 2-10`
if [ $fp1 = $fp2 ]
then
echo "Files $file1 and $file2 have same permission $fp1"
else
echo "permission of first file $file1:"
echo $fp1
echo "permission of second file $file2:"
echo $fp2
fi
```

Output:

```
[root@bt~]# vi pgm5.sh
[root@bt~]# chmod +x pgm5.sh
[root@bt~]# ./pgm5.sh
Enter the name of first file:
aa.txt
Enter the name of second file:
bb.txt
Files aa.txt and bb.txt have same permission rw-r--r--

[root@bt~]# ./pgm5.sh
Enter the name of first file:
pgm1.sh
Enter the name of second file:
pgm2.sh
Permission of first file pgm1.sh: rwxr--r--
Permission of second file pgm2.sh: rw-r--r-r
```

18. Write a Shell script to read a file name and change the existing file permissions.

```
echo "changing permission using octal notation"
echo "enter the file name"
read fi
echo "enter the permission to change for each of user [UGO]"
echo "for user"
read u
echo "for group"
read g
echo "for others"
read o
chmod $u$g$o $fi
echo "The new permission of file $fi is as follows"
ls -l $fi
```

Output:

```
[root@bt~]# vi pgm6.sh
[root@bt~]# chmod +x pgm6.sh
[root@bt~]# ./pgm6.sh
Changing permission using octal notation:
Enter the file name:
pat.lst Enter the permission to change for each of user[UGO]
for user
4
for group
6
for others
1
The new permission of file pat.lst is as follows
-r--rw---x root root 23 2017-03-05 05:27 pat.lst
```

19. Write a shell script to print current month calendar and to replace the current day number by '*'or '' respectively.**

```
#!/bin/bash
today=`date +%e`
if [ $today -lt 10 ]
then
cal|sed s/"$today"/*/g
else
cal|sed s/"$today"/**/g
fi
```

Output:

```
[root@bt~]# vi pgm7.sh
[root@bt~]# chmod +x pgm7.sh
[root@bt~]# ./pgm7.sh
           JAN 2018
Su  Mo  Tu  We  Th  Fr   Sa
     1   2    3   4    5    6
 7   8    9  10  11   12  13
14  15  16  17  18  19  20
21  **  23  24  25  26  27
28  29  30  31
```

20. Write a C-program to fork a child process and execute the given Linux commands.

```
#include<stdio.h>
#include<sys/wait.h>
int main(void)
{
int pid;
int status;
printf ("Hello World! \n");
pid = fork();
if (pid == -1)
{
perror ("bad fork");
exit(1);
} if (pid ==0 )
printf ("I am the child process. \n);
else
{
wait(&status)
printf ("I am the parent process. \n);
}
}
```

Output:

```
[root@bt~]# vi pgm8.c
[root@bt~]# gcc pgm8.c
[root@bt~]# ./a.out
Hello World!
I am the child process.
I am the parent process.
```

21. Write a C-program to fork a child process, print owner process ID and its parent process ID.

```
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
int main()
{
pid_t p1;
int pid,ppid;
p1=fork();
if(p1==-1)
{
printf("child process id not created\n);
exit(0);
}
pid=getpid();
ppid=getppid();
printf("The child process id is %d\n",pid);
printf("The parent process id is %d\n",ppid);
return 0;
}
```

Output:

```
[root@bt~]# vi pgm9.c
[root@bt~]# gcc pgm9.c
[root@bt~]# ./a.out
The child process id is 4572
The parent process id is 1234
The child process id is 5678
The parent process id is 1
```

22. Write a C-program to prompt the user for the name of the environment variable, check its validity and print an appropriate message.

```
#include<stdio.h>
#include<conio.h>
int main()
{
char x[10],ans,*y;
do
{
printf("Enter the Variable:\n");
scanf("%s",x);
y=getenv(x);
if(y!=NULL)
printf("%s is an environment variable",x);
else
printf("%s is not an environment variable",x);
printf("Do you want to continue(Y or y/N or n)?");
getchar( );
ans=getchar();
}
while(ans=='y'||ans=='Y');
exit(0);
}
```

Output:

```
[root@bt~]# vi pgm10.c
[root@bt~]# gcc pgm.10.c
[root@bt~]# ./a.out
Enter the variable: HOME
HOME is an environmental variable
Do you want to continue(y or Y/Nor n)?y

Enter the variable: cd
cd is not environmental variable.
Do you want to continue(y or Y/Nor n)?y

Enter the variable: PATH
PATH is an environmental variable
Do you want to continue(y or Y/Nor n)?y
```