


P.Y. Chary

UNIX / LINUX

With Advanced Shell Scripting

By Sreedevi

HAND BOOK


Technosoft
Software Training & Development

Ameerpet | Hyderabad

Tecnosoft Offered Courses

MS-OFFICE , TALLY

C with DS , C++ with DS

UNIX/LINUX

UNIX/LINUX 2 Days Only

PERL Scripting

PERL 2 Days Only

ORACLE 9i/10g/11g

D2K , PHP

ORACLE 10g DBA

SQL Server, Server DBA

TESTING TOOLS

.NET (vb,asp,c#)

DATA WAREHOUSING

JAVA,J2EE



ALLU CHARITABLE FOUNDATION

A helping hand to poorest of the poor

OUR AIMS

1. To bring computer education within the reach of the poor
2. Arranging drinking water facility to remote villages
3. Arranging small scale self employment facilities to the poor, backward and deprived women
4. Arranging three wheelers to the disabled persons
5. Arranging food and shelter to the orphans, poor kids and aged persons
6. Free medical check up campus and distribution of medicines at free of cost

ప్రజాసేవ
సామాజిక సేవ



Mr. ALLU VENKATA REDDY

Chairman & Managing Director
"AVREDDY Group of Companies"
and
Founder & Managing Director
"ALLU CHARITABLE FOUNDATION"

Democratic Work-style :

A professional management with a democratic style does allow the students enjoy a friendly environment, to make them take the best Professional team work with a start-to-finish accountability.

Faculty Expertise :

A long line of experience and expertise make our faculty the top-notch with a veteran domain strength.

Training Commitment :

Training on courses with full-fledged support, technically and morally. Timely and orderly assessment of skills to match with the corporate requirement. Extra support with a real-time experience in all deserving cases. Support on communication, personality development and interview facing skills. Simulation tests, English language skills to meet with the client requirements and the corporate culture

Value Addition :

Expert counselling to know and assert a direction to the student, Identifying the strengths and weaknesses for a strategic analysis to improve the weaker areas.



Mrs. ALLU SREEDEVI REDDY

Director
"AVREDDY Group of Companies"
and
Chairman
"ALLU CHARITABLE FOUNDATION"

The Promising Duo....

The couple... the duo.. known for their unity are dynamic, young and energetic with committed strengths in IT training areas. Mrs. Sreedevi Venkata Reddy is a known faculty, the top-notch with a long experience. The duo are committed morally and technically for a promising quality, taking their teams of faculties together. Consultancy, Development and IT enabled services have been a serious customary at A.V.R. Tecnosoft Solutions Pvt. Ltd., eTech info-systems India Pvt. Ltd. in Hitech City, and Tecno Speak Institute of english to unleash the language of success(English) the leading institutions making their long & strong presence in the field and now launching Social service activities through "ALLU CHARITABLE FOUNDATION" in remote villages.

Tecnosoft

Software Training & Development

E-Mail : tecnosoft4u@gmail.com

H.O: AMEERPET

#109, 1st Floor, Annapurna Block, Aditya Enclave, Hyd-38, AP, INDIA.

PH: 040-66839666. 66619666. 09966422225

ALLU CHARITABLE FOUNDATION

UNIX

UNIX is a CUI operating system. Operating System is an interface between hardware and applications software's. It serves as the operating system for all types of computers, including single-user personal computers and engineering workstations, multi-user microcomputers, mini computers, mainframes, and super computers, as well as special – purpose devices. The number of computers running a variant of UNIX has grown explosively, with approximately 20 million computers now running UNIX and more than 100 million people using these systems. The success of UNIX is due to many factors, including its portability to a wide range of machines, its adaptability and simplicity, the wide range of tasks that it can perform, its multi-user and multitasking nature, and its suitability for networking, which has become increasingly important as the internet has blossomed.

History of UNIX:-

Before development of UNIX operating system at AT & T Bell labs, Software team lead by Ken Thomson, Dennis Ritchie and Rudd Canday worked on MULTICS project. MULTICS stand for Multi Information Computing System. The aim of this project is to share the same data by 'n' number of users at the same time. Initially, MULTICS was developed for only two users. Based on the same concept in 1969, UNICS operating system was developed for 100's of users. UNICS stands for Uniplexed Information Computing System. Initially UNICS was written Assembly Language. In 1973, they rewritten in 'c' Language named as UNIX.

Salient Features of Unix:-

1. Multi User Capability
2. Multi Tasking Capability
3. Programming Facility
4. Portability
5. Communication (Electronic Mail)
6. Security
7. Open System
8. System Calls
9. Help Facility

1.Multi user:

Multi user operating system means more than one user shares the same system resources (hard disk, memory, printer, application software etc...,) at the same time.

2.Multi tasking:

Another highlight of Unix is that it is Multitasking, implying that it is capable of carrying out more than one job at the same time. It allows you type in a program in its editor while it simultaneously executes some other command you might have given earlier, say to sort and copy a huge file. The latter job is performed in the 'background', while in the 'foreground' you use editor, or take a directory listing or whatever else.

Depending on the priority of the task, the operating system appropriately allots small time slots(of the order of milliseconds or microseconds) to each foreground and background task.

3.Programming facility:

UNIX o/s provides shell. Shell works like a programming language. It provides commands and key words. By running these two, user can prepare efficient program.

4.Portability:

one of the main reasons for the universal popularity of Unix is that it can be ported to almost any computer system, with only the bare minimum of adoptions to suit the given computer architecture. It works with 8088 processors to super computers.

5.Communication:

UNIX provides electronic mail. The communication may be within the network of a single main computer, or between two or more such computer networks. The user can easily exchange mail, data, programs thro such networks. You may be two feet away or at two thousands miles your mail with hardly take any time to reach its destination.

6.Security:

UNIX provides three levels of security to protect data. The first is provided by assigning passwords and login names to individual users ensuring that not anybody can come and have access to your work.

At the file level, there are read, write and execute permissions to each file which decide who can access a particular file, who can modify it and who can execute it. Lastly, there is file encryption. This utility encodes your file into an unreadable format, so that even if some one succeeds in opening it, your secrets are safe.

7.Open system:

The source code for the UNIX system, and not just the executable code, has been made available to users and programmers. Because of this many people have been able to adapt the UNIX system in different ways. This openness has led to the introduction of a wide range of new features and versions customized to meet special needs. It has been easy for developers to adapt to UNIX, because the computer code for the UNIX system is straightforward, modular and compact.

8.System Calls:

Programs interact with the kernel through approximately 100 system calls. System calls tell the kernel to carry out various tasks for the program, such as opening a file, writing to a file, obtaining information about a file, executing a program, terminating a process, changing the priority of a process, and getting the time of day. Different implementations of UNIX system have compatible system calls, with each call having the same functionality. However, the internals, programs that perform the functions of system calls(usually written in the c language).

9.Help facility:

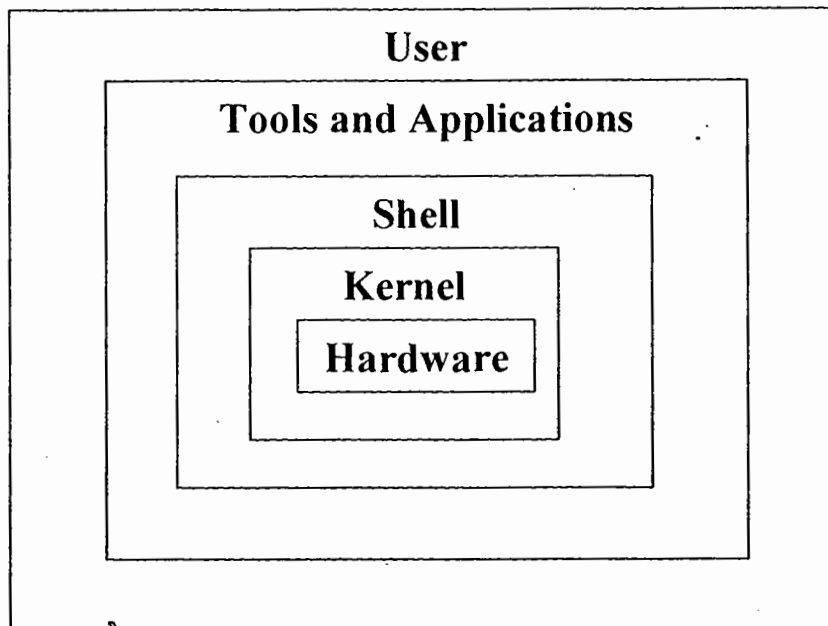
Unix provides manual pages for UNIX commands.

Difference between UNIX and windows server :

Sl . no	UNIX	WINDOWS NT
1	Unix is multi-user o/s.	Windows is a multi-user o/s.
2	Multi tasking o/s.	Multi tasking o/s.
3	To boot the UNIX o/s, 2MB RAM is enough.	To boot the windows o/s, 12MB RAM is required.
4	Unix is process based concept.	Window is process thread based concept.
5	In Unix, for every user request it creates new process.	For number of users request it creates only one process.
6	In Unix, any user process is killed it will not effect the other users.	It effects to all users.
7	Can run more than 1,00,000 transactions per minute.	Maximum number of transactions in windows o/s is 80,000 per minute.
8	There is no limit for number of users working with the server.	Limited number of users.
9	Unix is an open system.	Windows is an closed system.
10	Unix is a portable o/s	No portability.
11	Unix provides the programming facility.	No programming facility
12	It is CUI.	Windows is GUI
13	Unix is not a user friendly	It is user friendly.
14	Number of vendors(Red hat, Sun Microsoft, IBM, HP, SCO etc)	Only one vendor (Microsoft).

Difference between Linux and windows server:

Component	Linux	Windows NT Server 4.0
Operating System	Free, or around \$49.95 for a CD-ROM distribution	Five-User version \$809 10-User version \$1129 EE 25-User Version \$3,999
Free online technical support	Yes, <u>Linux Online</u> or <u>Redhat</u>	No
Kernel source code	Yes	No
Web Server	Apache Web Server	IIS
FTP Server	Yes	Yes
Telnet Server	Yes	No
DNS	Yes	Yes, though reports indicate that it is a broken implementation with limited functionality.
Networking	TCP/IP, IPv6, NFS, SMB, IPX/SPX, NCP Server (NetWare Server), AppleTalk, plus many other protocols	TCP/IP, SMB, IPX/SPX, AppleTalk, plus many other protocols
X Window Server (For running remote GUI-based applications)	Yes	No
C and C++ compilers	Yes	No
Perl 5.0	Yes	No
Number of file systems supported	32	3
Number of GUIs	4	1

Architecture Of UNIX System**Shell:**

The shell reads your commands and interprets them as requests to execute a program or programs, which it then arranges to have carried out. Because the shell plays this role, it is called a command interpreter. Besides being a command interpreter, the shell is also a programming language. As a programming language, it permits you to control how and when commands are carried out. Shell acts as an interface between user and the kernel.

Kernel:

The kernel is the part of the operating system that interacts directly with the hardware of a computer, through device drivers that are built into the kernel. It provides set of services that can be used by programs, insulating these programs from the underlying hardware. The major functions of the kernel are to manage computer memory, to control access to the computer, to maintain file system, to handle interrupts(signals to terminate execution), to handle errors, to perform input and output services(which allow computers to interact with terminals, storage devices, and printers), and to allocate the resources of the computer(such as the cpu or input/output devices)among users. Programs interact with the kernel through approximately 100 system calls. System calls tell the kernel to carry out various tasks for the program, such as opening a file, writing to a file, obtaining

information about a file, executing a program, terminating a process, changing the priority of a process, and getting the time of day

The UNIX File System

A file is the basic structure used to store information on the UNIX system. Technically, a file is a sequence of bytes that is stored somewhere on a storage device, such as a disk. The UNIX file system provides a logical method for organizing, storing, retrieving, manipulating, and information. A file can store manuscripts and other word processing documents, instructions or programs for the computer itself, an organized database of business information, a bitmap description of a screen image, or any other kind of information stored as sequence of bytes on a computer. Files are organized into a hierarchical file system, with files grouped together into directories. Within UNIX, There are three different types of files

1. Regular files
2. Diretory files
3. Special files

1. Ordinary files:

As a user, the information that you work with will be stored as an ordinary file. Ordinary files are aggregates of characters that are treated as a unit by the UNIX system. An ordinary file can contain normal ASCII characters such as text for manuscripts or programs. Ordinary file can be created, changed, or deleted as you wish.

2.Directory files:

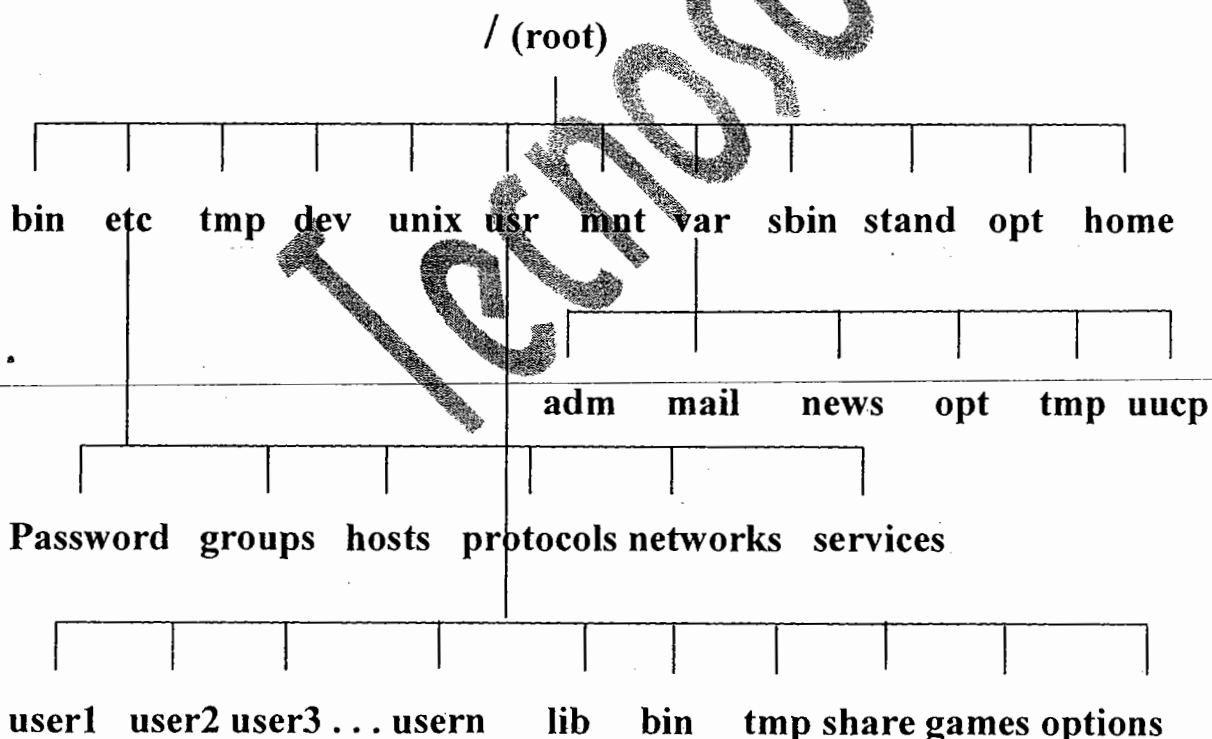
Directory is a file that holds other files and contain information about the locations and attributes of these other files. For example, a directory includes a list of all the files and subdirectories that it contains, as well as their addresses, characteristics, file types(whether they are ordinary files, symbolic links, directories, or special files), and other attributes.

3.Special files:

Special files constitute an unusual feature of the UNIX file system. A special file represents a physical device. It may be a terminal, a communications device, or a storage unit such as disk drive. From the user's perspective, the UNIX system treats special file just as it does ordinary files; that is, you can read or write to devices exactly the way you read and write to ordinary files. You can take the characters typed at your keyboard and write them to an ordinary file or a terminal screen in the same way. The UNIX system takes these read and write commands and causes them to activate the hardware connected to the device.

The Hierarchical File Structure

The UNIX file system called as hierarchical file system. In fact, within the UNIX system, there is no limit to the number of files and directories you can create in a directory that you own. File systems of this type are often called tree-structured file systems, because each directory allow you to branch off into other directories and files. The UNIX file system resembles an upside down tree.



/ (root): This is the root directory of the file system, the main directory of the entire file system, and the root directory for the superuser.

/bin: bin stands for binary. This directory contains executable files for most of the unix commands. Unix commands can be either C programs or shell programs. Shell programs are nothing but a collection of several Unix commands.

/etc: This contains system administration and configuration databases.

/dev: This contains the special(device) files that include terminals, printers, and storage devices. These files contain device numbers that identify devices to the operating system, including:

/dev/console the system console

In UNIX, similar devices are located in subdirectories of /dev. For example, all disk devices are in the subdirectory /dev/dsk.

/lib: This directory contains all the library functions provided by unix for programmers. The programs written under Unix make use of these library functions in the **lib** directory.

/sbin: This contains programs used in booting the system and in system recovery.

/opt: This is the root for the subtree containing the add-on application packages.

/home: This contains the home directories and files of all users. If your logname is tecno, your default home directory is /home/tecno.

/tmp: This contains all temporary files used by the UNIX system.

/mnt: Contains entries for removable(mountable) media such as CD-ROMs and DLT tapes.

/unix: This Directory contains unix kernel programming.

/usr: This contains other accessible directories such as /usr/lib and /usr/bin.

/usr/bin: Contains many executable programs and UNIX system utilities.

/usr/games: contains binaries for game programs and data for games.

/usr/lib: contains libraries for programs and programming languages.

/usr/sbin: contains executable programs for system administration.

/usr/share/man: contains the online manual pages.

/var: This contains the directories of all files that vary among systems. These include files that log system activity, accounting files, mail files, application packages, backup files for editors, and many other types of files that vary from system to system. Files in this directory include:

/var/adm: contains system logging and accounting files.

/var/mail: contains user mail files

/var/news: contains messages of common interest

/var/opt: is the root of a subtree containing add-on application packages.

/var/tmp: is a directory for temporary files.

/var/uucp: contains log and status files for the UUCP system

Login process:

login : tecno ↵

password : tecnosoft ↵

- system administrator prompt

\$ - user working prompt

Basic commands:

1. **\$ logname** : It displays the current user name

tecno

2. **\$ pwd** : It displays current working directory

/usr/tecno

3. **\$ clear** : It clear the screen.

4. **\$ exit** : To logout from current user

5. **\$ date** : It displays system date & time

Sat mar 4 04:40:10 IST 2005

6. **\$ Who am i** : It displays current username, terminal number, date and time at which you logged into the system

tecno ty01 mar 4 09:30

7. \$ who : To displays the information about all the users who have logged into the system currently. i.e each user login name, terminal number, date and time thet the person logged in.

```
tecno1    ttyo1    mar 4 09:30
tecno2    tty05    mar 4 10:10
tecno3    tty06    mar 4 10:15
tecno4    tty04    mar 4 10:30
```

ie tecno 1 - login name
 tty011 - terminal name
 mar 4 - date
 09:30 - time .

8. \$ finger : It displays complete information about all the users who are logged in.

9. \$ cal : It displays previous month, current month and next month calendar.

10. \$ cal year : It display the given year calendar
 e.g.:- \$ cal 2005 ↵ It takes year from 1 to 9999

11. \$ cal month year : It display the given month calendar only.
 \$ cal 3 2005 ↵

12. #init : To change system run levels.

1. # init 0 : To shut down the system

2. # init 1 : To bring the system to single user mode.

3. # init 2 : To bring the system to multi user mode with no resource shared.

4. # init 3 : To bring the system multi user mode with source shared.

5. # init 6 : Halt and reboot the system to the default run level

13. \$ banner "tecno": It prints a message in large letters.

Creating files:

There are two commands to create files : **touch** and **cat**

1. \$ Touch filename: It creates zero byte file size.

Eg: \$ touch sample.

The size of sample file is zero bytes.

⇒ Touch does not allow you to store anything in a file. It is used for to create several empty files quickly.

Eg: \$ touch file1 file2 file3 file4 file5

2. Cat Command

Syntax

\$ cat > filename

e.g.: \$ cat > sample ↵

ctrl + d { To close a file }

(i) **\$ cat >> sample** ↵ To append data to the file

ctrl+d

(iii) **\$ cat file1 file2 file3 > file4**

This would create file4 which contains contents of file1 followed by file2 and followed by that of file3. i.e It concatenates file1, file2 and file3 contents and redirects to file4. If file4 already contains something it would be over written.

3. \$ cat < filename (or) \$cat filename : To open a file.

Eg: \$ cat sample

It displays sample file contents.

Eg: \$cat file1 file2 file3

It displays file1 contents followed by file2 then followed by file3

Removing Files

4 rm command:

To remove the given file.

Syntax

\$ rm filename

\$ rm sample : It removes sample file.

\$ rm -i filename : It asks confirmation before deleting the file.

\$ rm -i sample ↵

remove sample ? y - It removes

n - It won't remove

\$ rm file1 file2 file3 : It removes three files.

\$ rm *: It removes all files in current directory

Creating Directory

5. mkdir Command:-

To make directory

Syntax:

\$ mkdir directory name

eg: mkdir tecno

Creating multiple directories

\$mkdir dir1 dir2 dir3dirn

Change directory :

Syntax: \$cd directory name

\$cd tecnosoft

\$pwd

/usr/tecno/tecnosoft

\$cd .. : To change into parent directory.

**\$cd ** : To change to root directory.

Remove directory :

1. **\$rmdir directoryname** : To delete a directory but directory should be empty.
2. **\$rm -r directoryname** : It removes all files and sub directories, sub directory files including directory.

Copy a file :

\$cp file1 file2 : This will copy the contents of file1 into a file2. if file2 is already existed it overwrites:

\$cp -i file1 file2 : If file2 is already existed then it asks the confirmation.

Eg: **cp -i sample1 sample2 ↵**
Overwrite sample2 ?

Rename a file :

1. **\$mv old filename new filename** : To rename the file.

Comparison of files :

1. **\$cmp file1 file2** : It compares file1 and file2 . If both file contents are same number output if files are different then it displays line number and character location.

Word count :

\$wc filename : It counts number of lines, words & character and displays.

1. **\$wc file1 ↵**

5 10 70 file1

in above output, 5 represents as total no of lines, 10 represents as total no of words, 70 represents as total no of characters.

2. **\$wc file1 file2 file3 ↵**

10 30 100 file1

5	15	65	file2
7	83	120	file3
22	128	285	total

3. **\$wc -l file1** ⇒ It displays only line count
\$wc -w file1 ⇒ It displays only word count
\$wc -c file1 ⇒ It displays only character count
\$wc -lw file1 ⇒ It displays only line & word count
\$wc -wc file1 ⇒ It displays only word character count
\$wc -lc file1 ⇒ It displays only line & character count

Listing of files :

1. **ls** : It displays list of files and directories
2. **\$ls -x** : It displays width wise
3. **\$ls | pg** : It displays list of files and directories pages wise
4. **\$ls -x | pg** : It displays list of files and directories pages wise & width wise.
5. **\$ls -a** : It displays files and directories including . and .. hidden files.
6. **\$ls -F** : It displays files , directories, executable files, symbolic files.
\$ls -F
a1
a2
a3
sample\
letter *
notes @
in above output, the filename ends with /-directory, *- executable file,@ - symbolic link file.
7. **\$ls -r** : It displays files and directories in reverse order i.e descending order
8. **\$ls -R** : It displays files and directories recursively.
9. **\$ls -t** : It displays files and directories based on date & time of creation i.e. last file to first file

10. **\$ls -rt** : It displays files and directories based on date & time of creation but in reverse order. i.e. first file to last file.

11. **\$ls -i** : it displays files and directories along with inode number

12. **\$ls -l** : It displays files& directories in long list format i.e.

filetype	Permission	link	Uname	group	sizeinbytes	date	filenam e
-	Rw-r--r--	1	Tecno	Group	560	Nov 3 1.30	Sample 1
D	Rwxr-wr-w	2	Tecno	group	700	Mar 5 7:30	tecnoso ft

Different types of files :

- Ordinary file
- d directory file
- b special block file
- c special character file
- L symbolic link file.

Wild card characters (or) meta characters :

*, ?, [], -

1. **\$Ls a*** : It displays all file , starting letter is 'a'.
2. **\$Ls b*k** : It displays all files starting letter is and ending letter is k.
3. **\$Ls *k** : It displays all files ending letter is k.
4. **\$Ls a?k** : It displays all three length filenames but starting letter is a.
5. **\$Ls b??k** : It displays all 4 length filenames but starting letter is b and ending letter is k.
6. **\$ls [aeiou]*** : It displays all files but first character of the filename to be listed must be. any of the letters given within the square brackets and the remaining can be anything.
7. **\$ls [!aeiou]*** : It displays all files whose first character is anything other than a vowel
8. **\$ls [k-v]*** : It displays all files whose starting letter is between k and v.

Link files:-

A link is not a kind of file but instead is a second name for a file. When a file had two links, it is not physically present at two places, but can be referred to by either of the names. This is very useful future. If you accidentally delete a file with a single link or a single name, there is no bringing it back, as UNIX has 2 links your file is safe even if one of the links sets several. A link is severed when the file is deleted.

The concept of having several link to a file offers another advantage. If one file is to be shared b/n several users; instead of giving each user a separable copy of the same file we LAN create links of this file in each user's directory. This avoids unnecessary duplication of the same file contents in different directories.

\$ln sample mysample

The above command establishes one more link for the file sample in the form of the name my sample.

Symbolic links

Links can be used to assign more than one name to a file, but they have some important limitations. They cannot be used to assign a directory more than one name. And they cannot be used to link filenames on different computers.

These limitations can be eliminated using symbolic links. A symbolic link is a file that only contains the name of another file. When the operating system operates on a symbolic link, it is directed to the file that the symbolic points to. Symbolic links can assign more than one name to a file, and they can assign more than one name to a directory.

The **ln** command can link files within a single file system. You can also link files across file systems using the **-s**(symbolic) option to **ln**. The following example shows how you could use this feature to link a file in the **/var** file system to an entry in one of your directories within the **/home** file system.

```
$ln -s /var/x/docs/readme temp/x.readme
```

In addition to allowing links across file systems, symbolic links enable you to link directories as well as regular files.

Permissions:

Permissions for files:

There are three classes of file permissions for the three classes of users: the owner(or user) of the file. The group the file belongs to, and all other users of the system.

```
$ls -l
```

```
-rwxr-xr--
```

The first three letters of the permissions field refer to the owner's permissions; the second three to the members of the file's group; and the last to any other users.

The first three letters, **rwx**, show that the owner of the file can read it, write(w) it, and execute(x) it.

The second group of three characters, **r-x**, indicates that members of the group can read and execute the file but cannot write it. The last three characters, **r-x**, show that all others can read and execute the file but not write to it.

If you have read permission for a file, you can view its contents. Write permission means that you can alter its contents. Execute permission means that you can run the file as a program.

Permissions for Directories:

For directories, read permission allows users to list the contents of the directory, write permission allows users to create or remove files or directories inside that directory, and execute permission allows users to change to this directory using the **cd** command or use it as part of a pathname.

Changing File Permission

Chmod is the command to change file permission's or directory permission's.

Syntax

\$chmod [who] [+/-/=] [permissions] filename

in using, first specify which permissions you are changing, Second specify how they should be changed, Third, specify the file permission type

u for user or owner

g for group

o for others

+ for to add permission

- for to subtract permission

= for to **assign permission** (i.e. add specified permission and take away all other permissions, if present)

r for to read

w for write

x for to execute

Eg 1: add write permission to group members on tecno file

\$chmod g+w tecno

Eg 2: add execute permission to others and owner on tecnosoft file

\$chmod u+x,o+x tecnosoft

or

Schmod uo+x tecnosoft

Eg 3: assign read permission to others and remove write permission from group members on tecnosoft file

Schmod o=r,g-w tecnosoft

Another form of the **chmod** command lets you set permissions directly, by using a numeric(octal) code to specify them.

This code represents a file's permissions by three octal digits; one for owner permissions, one for group permissions, and one for others. These three digits appear together as one three – digit number.

Permission	Weight
Read	4
Write	2
Execute	1
Read and write	6
Write and execute	3
Read and execute	5
Read , Write and Execute	7

Eg 1:

Schmod 700 tecnosoft

. The above command sets read, write and execute permissions for the owner only and allows no one else to do anything with the tecnosoft file.

Eg 2;

Schmod 754 tecnosoft

The above command sets all permissions for owner and sets only read and execute to group and sets only read permission to others on tecnosoft file.

Using umask to Set Permissions

The **chmod** command allows you to alter permissions on a file –by-file basis. The **umask** command allows you to do this automatically when you create any file or directory. Everyone has a default **umask** setting set up either by the system administrator or their .profile.

Umask allows you to specify the permissions of all files created after issue the **umask** command. Instead of dealing with individual file permissions, you can determine permissions for all future files with a single command. Unfortunately, using **umask** to specify permissions is rather complicated, but it is made easier if you remember 2 points:

1. **umask** uses a numeric code for representing absolute permissions just as **chmod** does. For example. 777 means read, write and execute permission for user, group and others(rwxrwxrwx)
2. You specify the permissions you want by telling **umask** what to subtract from full permission value, 777(rwxrwxrwx)

Eg 1: **\$umask 022**

The above command gives all new files in this session will be given permissions of rwxr-xr-x

Note that it corresponds to a numeric value of 755, and 755 is simply the result of subtracting the "mask" -022 in this example - from 777

Eg 2: To make sure that no one other than yourself can read, write, or execute your files, you can run the **umask** command at the beginning of your login session by putting the following line in your .profile file:

\$umask 077

i.e the above command same as **chmod 700** or **chmod go-rwx** , but **umask** applies to all files you create in your current login session after you issue the **umask** command

Changing the Owner of a File

Every file has an owner, usually the person who created it. When you create a file, you are its owner. The owner usually has broader permissions for manipulating the file than other users.

Sometimes you need to change the owner of a file; for example, if you take over responsibility for a file that previously belonged to another user.

The **chown** command takes two arguments: the login name of the new owner and name of the file.

Eg 1:- **\$chown tecno hello**

In above example, tecnosoft is the new owner of hello file.

Note: Only the owner of a file (or the superuser) can use **chown** to change its ownership.

Changing the Group of a File:

Every file belongs to a group. Sometimes, such as when new groups are set up on a system or when files are copied to a new system, you may want to change the group to which a particular file belongs. This can be done using the **chgrp**(change group) command. The **chgrp**

Command takes two arguments, the name of the new group and name of the file.

Eg 1: \$chgrp tecnosoft hello

In above example, chgrp command changes hello file into tecnosoft group.

Note: only the owner of a file (or the super user) can change the group to which this file belongs

File compression:-

There are two commands to zip the files.

1.Compress

2.Pack

Compress:-

Syntax: \$compress filename

The result of above command is filename.Z

Eg:- \$ compress sample

On compression the original file is replaced by another which has the same name with .Z extension added to it.

1. \$Compress -V sample

The optional -V(for verbose) option files compress to report how much space it saved . For example, in the above case it reported the following:

Sample:compress:76,56%-replaced with sample Z

\$Uncompress:

To get the compresses the back to its original state, we can use the uncompress utility as shown below.

\$Uncompress sample.Z

Sample. Z is deleted and the original sample is recreated back in its original form and shape.

Once the file has been compressed we cannot view it using the normal cat command. Unix provides a utility called Zcat for this purpose.

\$zcat sample.Z

It displays file contents in readable format.

Pack:

It is also able to compact a file. The compress offers a higher degree of compression as compared to pack.

Syntax : \$pack filename

Eg: \$pack sample

Sample : 37.1% compression is smaller. The packed contents are stored in the file sample.z and the original file sample is deleted. To view the contents of a packed file we can use the pcat command.

\$pcat sample .z

To get back the original file from the packed file there is a utility called unpack.

\$unpack sample.z

unpack: sample : unpacked.

Filter Commands

grep Command:

The grep command searches through one or more files for lines containing a target and then prints all of the matching lines it finds. For example, the following command prints all lines in the file `tecno_file` that contain the word "tecnosoft".

Eg1 : `$ grep tecnosoft tecno_file`

Choose your career in tecnosoft solutions

Eg2: `$ grep "tecnosoft solutions" tecno_file`

To search more than word the string should be in double quotes

Using grep for Queries:-

grep is often used to search for information in structured files or simple databases. An example of such a file is "students".

\$ cat students

101	venkat	C`	tecnosoft solutions
102	sreedevi	C++	tecnosoft solutions
103	neeraja	UNIX	tecnosoft solutions
104	narasimha	VB6.0	tecnosoft solutions
105	santhosh	UNIX	tecnosoft solutions

This is a typical example of a personal data base file. The file consists of records, each of which is terminated with a new line. A record contains several fields, separated by a field separator or delimiter. In this example, the field separator is the tab character. Database files like this can be created with an ordinary UNIX system text editor like **vi**, or with a word processor.

You can use **grep** to find all students in the file that contain the word
For example:

\$ grep UNIX students

103	neeraja	UNIX	tecnosoft solutions
105	santhosh	UNIX	tecnosoft solutions

Using grep to locate files:

If you give grep two or more files to search, it includes the name of the file before each line of output. For example, the following command searches for lines containing the string "UNIX" in all of the files in the current directory:

\$grep UNIX *

```
students : 105      santhosh      UNIX      tecnosoft solutions
students : 103      neeraja      UNIX      tecnosoft solutions
sample   : UNIX is pworefull operating system
tecno_file : The source code of UNIX system is open.
```

Searching for patterns using regular expressions

The examples so far have used **grep** to search for specific words or strings of text, but grep also allows you to search for targets defined as patterns that may match a number of different words or strings. You specify patterns for **grep** using the same kinds of *regular expressions* that were described in the discussion of text editing. In fact the name "**grip**" stands for "global regular expression and print." The rules and symbols used for forming regular expressions for **ed** and **vi** can also be used with **grep** to search for patterns. For example,

\$ grep 'ch.*se' recipes

Will find entries containing "Chinese" or "cheese." The dot(.) matches any number of characters except newline (any number includes zero, so the dot will match nothing as well). The asterisk specifies any number of repetitions; together they indicate any string of any characters. (because this pattern matches any string beginning with "ch" and ending with "se", it will also find a line containing : reach for these".)

note that in this example the target pattern "ch.*se" is enclosed in single quotation marks. This prevents the asterisk from being treated by the shell as a filename wildcard. In general, remember to put in quotes any regular expression that contains an asterisk or any other character that as special meaning for the shell.

The grep command useful options:

A large number of options are available that let you modify the way **grep** works. Three especially use full options, **-v**, **-i**, and **-I**, let you find lines that do not match the target, ignore uppercase and lowercase distinctions, and list file names only, respectively. Another helpful option, **-n**, allows you to list only the line numbers on which your target can be found.

E.g. 1: \$grep -v tecno tecno_file

It display all lines in tecno_file that do not contain the word tecno

E.g. 2: \$file * | grep -v text

It displays all files in the current directory that are not text files by piping the output of file to grep -v

fgrep Command :

The **fgrep** command is similar to **grep**, but with three main differences: you can use it to search for several targets at once, it does not allow you to use regular expressions to search for patterns, and is faster than **grep**. When you need to search a large files or several small files the differences in speed can be significant.

Searching for multiple targets :

The **grep** command prints all lines matching a particular pattern.

\$ fgrep "UNIX

>c++

>vb6.0" students

The output looks like this :

102	sreedevi	C++	tecnosoft solutions
103	neeraja	UNIX	tecnosoft solutions
104	narasimha	VB6.0	tecnosoft solutions
105	santhosh	UNIX	tecnosoft solutions

by the way, you should notice here that when you give fgrep multiple search targets, each one must be on a separate line.

Note that you have to put the search string in quotation marks when it contains several targets.

The **fgrep** command does not accept regular expressions. The target must be text strings.

Getting search targets from a file :

With the **-f** (file) option, you can tell **fgrep** to take the search targets from a file, rather than having to enter them directly . if you had a large mailing list named **customers** containing customer names and addresses, and a small file named **special** file that contained the names of special customers, you could use this option to select and print the address of the special customers from the overall list:

\$grep -f special customers

egrep Command:

The **egrep** command is the most powerful member of the **grep** command family. You can use it like **fgrep** to search for multiple targets. Like, **grep** , it allows you to use regular expressions to specify targets, but it provides a fuller, more powerful set of regular expressions than **grep**.

The **egrep** command accepts all of the basic regular expressions recognized by **grep** , as well as several useful extensions to the set.

You can tell **egrep** to search for several targets in two ways: by putting them on separate lines as in **fgrep** , or by separating them with the vertical bar or pipe symbol (**|**).

For example, the following command uses the pipe symbol to tell **egrep** to search for entries for **c**, **c++**, and **unix** in the file **students**:

\$egrep "c|c++|unix " students

101	venkat	C	tecnosoft solutions
102	sreedevi	C++	tecnosoft solutions
103	neeraja	UNIX	tecnosoft solutions
105	santhosh	UNIX	tecnosoft solutions

sort Command

The **sort** command can be used for sorting the contents of a file. Apart from sorting file, the **sort** can merge multiple sorted files and store the result in the specified output file. While sorting the **sort** command bases it comparisons on the first character in each line in the file. If the first character of two lines is same

then the second character in each line is compared and so on i.e. The sorting done according to the ASCII collating sequence.

Syntax:

\$sort filename

E.g. 1: \$sort tecno_file

This would sort the contents of tecno_file and display the sorted output on the screen.

If we want we can sort the contents of several files at one shot as in:

\$sort tecno1 tecno2 tecno3

This would sort the three files and displays the sorted output.

Instead of displaying the sorted output on the screen we can store it in a file by saying,

\$sort -oresult tecno1 tecno2 tecno3

The above command sorts the three files tecno1, tecno2 & tecno3 and saves the result in a file called **result**

If the files have already been sorted and we just want to merge them we can use:

\$sort -m tecno1 tecno2

Sometimes we may want to combine the contents of a file with the input from the keyboard and then carry out the sorting. This can be achieved by saying:

\$sort - tecno1

Where '-' stands for the standard input i.e. the keyboard

Cut Command:

Like sort, **cut** is also a filter. It cuts or picks up a given number of character or fields from the specified file. Say you have a large database of students information. If from this you want to view only a few selected fields, for instance student id and course name, **cut** is the answer.

\$cut -f 1,3 students

It displays the following output. In students file the first field is student_id and third field is course.

101	c
102	c++
103	UNIX
104	vb6.0

105 UNIX

if we are to view fields 2 through 5 we can say,

\$cut -f 2-5 students

101	venkat	C	tecnosoft solutions
102	sreedevi	C++	tecnosoft solutions
103	neeraja	UNIX	tecnosoft solutions
105	santhosh	UNIX	tecnosoft solutions

The **cut** command can also **cut** specified columns from a file and display them on the standard output. The switch used for this purpose is **-c**. for example,

\$cut -c1-15 students

as a result, the first 15 columns from each line in the file **Students** would be displayed.

Input/Output Redirection:

In all operating systems, there is a standard input device and a standard output device. In UNIX, as also in most other operating systems, the standard input device is keyboard and the standard output is the display screen.

Special files that instruct all the programs to accept standard input from the keyboard and direct the standard output to the display, are provided by Unix. The three streams, i.e. the standard input, standard output and standard error are denoted by the numbers 0, 1 and 2 respectively.

Stream	Device	Value
Standard input	Keyboard	0
Standard output	Terminal screen	1
Standard error	Terminal screen	2

Unix allows you to change the standard input and output temporarily by using what is known as redirection and piping.

Sometimes it is useful to redirect the input or output to a file or a printer. For example, you might want to redirect a directory listing from the screen to a file. Unix provides redirection symbols for the purpose

The symbol **>** implies redirection of output and the symbol **<** implies redirection of input.

The symbol > sends the output of a command to a file or a device, such as a printer.

The symbol < takes the input needed for a command from a file rather than from the keyboard.

The symbol >> adds output from a command to the end of a file without deleting the information already in the file.

E.g. Scat tecno1 > tecno2

On executing this command, the symbol > sends tecno1 file contents into tecno2 file instead of sending to output screen.

E.g. Scat tecno1 >> tecno2

on executing this command, the symbol >> sends tecno1 file contents into tecno2 then it allows the user to data into tecno2 file.

Piping:-

The Unix piping facility let us connect commands to other commands. This facility is of utmost importance in combining Unix commands and other operations. It can be really useful to redirect the output of one program so that it becomes the input of another program, thereby joining of two programs. To send the output of one command as input for another, the two commands must be joined using a pipe(|) character.

E.g. \$ls | wc -l

On executing this command, the pipe takes ls command output and redirects the output as a input to wc -l commands and display no of lines on output screen.

E.g. \$ls | wc -l > totalfiles

On executing this command, the pipe takes ls command output and redirects the output as a input to wc -l commands and counts no of lines and sends output to total files.

COMMUNICATION COMMANDS

The write command :

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

\$ write tecno2

hey there! I am back from Tirumala

Just wanted to say hello!

ctrl d

There are two prerequisites for a smooth **write** operation.:

- (a) The recipient must be logged in, else an error message is inevitable.
- (b) The recipient must have given permission for message to reach his or her terminal. This is done by saying at the \$ prompt

\$ mesg -y

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

\$mesg -n

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

mail Command :-

Using mail you can quickly and efficiently circulate memos and other written information to your co-workers. You can even send and receive mail from people outside your organization, if you and they use network computers. Mail can be sent to users who have logged in currently or even to users who haven't logged in currently. In case the user has logged in at several terminals the moment mail is sent to this user it becomes available at all the terminals.

Sending mail :

\$mail tecno1

subject: from tecnosoft solutions

hello,

how r u?

tecnosoft offering unix course with less fees.

ctrl d

To send the same mail to more than one user, you can say

```
$ mail tecno1 tecno2 tecno3
subject: from tecnosoft solutions
hello,
how r u?
tecnosoft offering unix course with less fees.
Ctrl d
```

If you are to mail a program written by you and then you of course can not be expected to type in the program after you have issued the **mail**. Command as in the above case. In such cases. In such cases you can use input redirection as shown below:

```
$ mail tecno2 tecno3 tecno4 < myprg.c
now the contents of the file myprog.c would be promptly mailed to tecno2, tecno3 and tecno4
```

Handling incoming Mail:

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

```
$ mail
```

```
SCO system V mail type ? for help
```

```
"/usr/spool/mail/tecno1": 3 message 1 new 2 unread
```

>	N	3	tecno2	Sat Mar19 11:05	9/233	unix course
	U	2	tecno3	Sat Mar19 09:30	5/413	linux
	U	1	tecno4	fri Mar 18 11:42	4/391	offer c++

&

The output shows that there are 3 messages in our mailbox of which 1 has been received since we logged in whereas 2 were lying in the mailbox unread even before we logged in. the N or a U in the first column indicates just this. Each message present in the mail box is given a number. This members are shown in the second column. The > sign indicates that the current message is message number 3. followed by this is the logname of the person who sent the mail, the date and time when the mail was received as well as the subject of each message. Observe the & displayed at the bottom. This is known as the mail prompt.

We can use several commands at this prompt. If you want to know which, you can type a / and obtain help on these commands. let us now view the second message. Just type a 2 at the & prompt.

&2

it displays second message on the output screen
to view third message just type a 3 at the & prompt

&3

it displays third message on the output screen

Finally, we exit from the mail command by typing q at the & prompt.

&q

This implies that out of the three messages we have received we have gone through 2 messages hence they have been removed from our primary mailbox file **user/spool/mail/aa12** and appended to the secondary mailbox file **/user/aa12/mbox**. One message which remains unread is still lying in the primary mailbox. One message will remains unread is still lying in the primary mailbox file. If we invoke mail once again we would be shown only the one unread message. What if we want to review the message transferred to the secondary mailbox? Simply invoke the mail command as show below:

\$ mail -f

The following table shows the operations that you can perform on mail messages.

Command	Action
+ or return	Print next message
-	Print previous message
#	Print number of current message
D	Delete the message
Dq	Delete the message and quit
M person	Mail this message to person(your own login name is the default if person is not specified)
P	Print cuurent message again
q or CTRL-D	Put undeleted mail back in /var/mail/tecno, and quit
u N	Undeleted message N
s file	Save this message. \$usr/mbox is used as a default if file is not specified
W file	Save this message without its header information, /mbox is the default
X	Put all mail back in mail file and quit
! command	Escape to the shell and run command
?	Print summary of mail commands

FINDING FILES

The **find** command helps you locate files in the file system. With the **Find** command, you can search through any part of the file system, looking for all files with a particular name. It is extremely powerful, and at times it can be a lifesaver, but it is also rather difficult to remember and to use.

An example of an common problem that **find** can help solve is locating a file that you have misplaced. For example, if you want to find a file called `tecno_file` but you can't remember where you put it, you can use **find** to search for it through all or part of your directory system.

The **find** command searches through the contents of one or more directories, including all of their sub directories. You have to tell **find** in which directory to start its search. To search through all your directories, for example, tell **find** to start in your login directory.

Eg 1: The following examples searches user tecno's directory system for the file tecno_file and prints the full pathname of any file with that name that it finds:

```
$pwd
/usr/tecno
$find . -name tecno_file -print
/usr/tecno/dir/abc/tecno_file
/usr/tecno/xyz/x1/tecno_file
/usr/tecno/aaa/a1/a2/tecno_file
```

The first argument is the name of the directory in which the search starts. In this case it is the current directory(represented by the .). The second part of the command specifies the file or files to search for , and the third part tells **find** to print the full pathnames of any matching files.

To search the entire file system, start in the system's root directory, represented by the /:

```
$find / -name tecno_file -print
```

This will find a file named tecno_file anywhere in the file system. Note that it can take a long time to complete a search of the entire file system; also keep in mind that **find** will skip any files or directories that it does not have permission to read.

You can tell **find** to look in several directories by giving each directory as an argument. The following command first searches the current directory and its sub directories and then look in /tmp/project and its sub directories.

```
$find ./tmp/project -name tecno_file -print
```

You can use wildcard symbols with **find** to search for files even if you don't know their exact names. For example, if you are not sure whether the file you are looking for was called tecno_data, tecno.data, or ndata, but you know that it ended in data, you can use the pattern ***data** as the name to search for

```
$find -name "*data" -print
```

Running find in the background

If necessary you can search through the entire system by telling **find** to start in the root directory , /. Remember, though, that it can take **find** a long time to search through a large directory and its sub directories, and searching the whole file system , starting at / , can take a very long time on large system. If you need to run a command like this that will take a long time, you can use the multitasking feature of UNIX to run it as a background job, which allows you to continue doing other work while find carries out its search.

To run a command in the background, you end it with an ampersand(&). The following command line runs **find** in the background to search the whole file system and send its output to **tecno_found**:

```
$find / -name tecno_file -print > found_tecno &
```

The advantage of running a command in the background is that you can go on to run other commands without waiting for the background job to finish.

Getting Information About File Types

Sometimes you just want to know what kind of information a file contains. For example, you may decide to put all your shell scripts together in one directory. You know that several scripts are scattered about in several directories, but you don't know their names, or you aren't sure you remember all of them. Or you may want to print all of the text files in the current directory, whatever their content.

You can use several of the commands already discussed to get limited information about file contents. But the most complete and most useful command for getting information about the type of information contained in file is **file**.

file reports the type of information contained in each of the files you give it. The following shows typical output from using **file** on all of the files in current directory:

```
$file *
```

examples:	directory
tecnol:	ascii text
dirlink:	ascii text
cx:	commands text
linkfile:	symbolic link to dirlink
hello:	executable file
send:	English text
tag:	data

```
$file hello.c
```

```
c program file
```

VIEWING LONG FILES

pg command

The **pg** command displays one screen of text at a time and prompts you for a command after each screen. You can use the various **pg** commands to move back and forth by one or more lines, by half screens, or by full screens. You can also search for and display the screen containing a particular string of text.

Eg 1: The following command displays the file **tecnosoft1** one screen at a time:

\$pg tecnosoft1

To display the next screen of text, press RETURN. To move back one page, type the hyphen or minus sign (-). You can also move forward or backward several screens by typing a plus or minus sign followed by the number of screens and hitting RETURN. For example, **+3** moves ahead three screens, and **-3** moves back three.

You use **1** to move one or more lines forward or backward. For example, **-5l** moves back five lines. To move a half screen at a time, type **d** or press CTRL-D. If you want, you can tell **pg** how many lines to display at one time by using a command line option.

\$pg -10 tecno_file

This causes **pg** to display screens of ten lines. To quit **pg**, type **q** or **Q**, or press the BREAK or DELETE key.

more Command

In addition to **pg**, UNIX has another pager, **more**. Like **pg**, **more** allows you to move through a file by lines, half screens, or full screens, and it lets you move backward or forward in a file and search for patterns.

Eg 1: To display the file **tecno_1** use **more** this way:

\$more tecno_1

To tell **more** to move ahead by a screen, you press the SPACEBAR. To move ahead one line, you press RETURN. The commands for half-screen motions, **d** and CTRL-D, are the same as in **pg**. To move backward by a screen, you use **b** or CTRL-B.

Editor

Three editors available in almost all versions of unix. There are :

- 1.ed editor
2. ex editor
- 3.vi editor

Command for copying and paste text

Command	Function
yw	Yanks word from cursor position
yy	Yanks line from cursor position
y\$	Yanks line from cursor position to end of line
y0	Yanks line from cursor position to beginning of line
p	Pastes last yanked buffer

Commands for Quitting vi

Command	Function
ZZ	Writes the buffer to the file and quits vi
:wq	Writes the buffer to the file and quits vi
:w filename and :q	Writes the buffer to the file filename (new) and quits vi
:w! filename and :q	Overwrites the existing file filename with the contents of the buffer and quits vi.
:q!	Quits vi whether or not changes made to the buffer were written to a file. Does not incorporate changes made to the buffer since the last write (:w) command.
:q	Quits vi changes made to the buffer were written to a file.
:set nu	Setting line numbers for file
:set nonu	Removing line numbers

SHELL SCRIPTING

Tecnosoft Solutions

Courses offered

MS-OFFICE ,
ACCOUNTING PACKAGES
C with DS , C++ with DS ,
UNIX/LINUX ,UNIX 2 Days ,
LINUX ADMIN,
PERL Scripting ,
ORACLE 9i/10g ,D2K,
ORACLE 10g DBA ,
SQL Server,
TESTING TOOLS ,
VB 6.0 , VC++ ,
.NET (vb,asp.c#) ,
JAVA , J2EE ,
DATA WAREHOUSING ,

SHELL SCRIPTING

SHELL:-shell is command line interpreter. It takes commands from user and executes them. it is an interface between user and kernel.

The three most widely used UNIX shells are Bourne shell .korn shell and c shell.

Shell	Developed by	Shell prompt	Execution command
Bourne shell	Steve bourne	\$	Sh
Korn shell	David korn	\$	Ksh
C shell	Bill joy, California university student	%	Csh

Each shell has merits and demerits of its own. moreover the shell scripts written for one shell may not work with the other shell. This is because different shells use different mechanisms to execute the commands in the shell script. Bourne shell since it is one of the most widely used unix shells in existence today.

Almost all UNIX implementations offer the Bourne shell as part of their standard configuration. it is smaller than the other two shells and therefore more efficient for most shell processing. However, it lacks features offered by the c and the korn shell.

All shell programs written for the Bourne shell are likely to work with the korn shell. the reverse however may not be true, this is so since the facilities like arrays, command aliasing and history mechanism available in the korn shell are not supported by the bourne shell.

The c shell programming language resembles the c language and is quite different from the language of the bourne shell. only the very basic shell scripts will run under both the c and bourne shell; a vast majority will not shell keeps track of commands as you enter them(history) and allows you to go back and execute them again without typing the command. or ,if you want to, you can recall them, make modifications, and then execute the new command.

Shell program :-

a shell program is nothing but a series of such commands. instead of specifying one job at a time, we give the shell a to-do list –a program – that carries out an entire procedure. such programs are known as “shell scripts”.

When to use shell scripts:

1. Customizing your work environments. for example, every time you log in if you want to see the current date, a welcome message and the list of users who have logged in you can write a shell script for the same.
2. automating your daily tasks. for example, you may want to back up all your programs at the end of day. this can be done using a shell script.
3. Automating repetitive tasks. for example, the repetitive task of compiling a c program, linking it with some libraries and executing the executable code can be assigned to a shell script.
4. Executing important system procedures like shutting down the system, formatting a disk, creating a file system, mounting the file system, letting the users the floppy and finally unmounting the disk.
5. Performing same operation on many files. Or example, you may want to replace a string printf with a string myprintf in all the c programs present in a directory.

Shell Variables:-

Variable is a data name and it is used to store value. Variable value can change during execution of the program.

Variable's in unix are two types.

1. Unix-defined variables or system variables
2. user defined variables

Unix-defined variables:

These are standard variables which are always accessible the shell provides the values for these variables these variable are usually used by the system itself and govern the environment we work under. If we so desire we can change the values of these variables as per our preferences and customize the system environment.

The list of all system variables and their values can be displayed by saying at the \$prompt,

\$set

HOME=/usr/tecnosoft

HZ=100

IFS=

LOGNAME=tecnosoft

MAIL=/usr/spool/mail/tecnosoft

MAILCHECK=600

OPTIND=1

PATH=/bin:/usr/bin:/usr/tecnosoft:/bin:.

PS1=\$

PS2=>

SHELL=/bin/sh

TERM=vt100

TZ=IST-5:30

Variable	Meaning
PS1	Primary shell prompt
PS2	The system prompt 2, default value is ">"
PATH	Defines the path which the shell must search in order to execute any command or file
HOME	Stores the default working directory of the user.
LOGNAME	Stores the login name of the user
MAIL	Defines the file where the mail of the user is stored
MAILCHECK	Defines the duration after which the shell checks whether the user has received any mail. By default its value is 600 (seconds)
IFS	Defines the Internal Field Separator, which is a space, a tab or a new line
SHELL	Defines the name of your default working shell
TERM	Defines the name of the terminal on which you are working
TZ	Defines the name of the time Zone in which we are working

User Defined Variables

These are defined by user and are used most extensively in shell programming.

Rules for creating User Defined Shell Variables:-

1. The first character of a variable name should be alphabet or underscore
2. no commas or blanks are allowed within a variable name
3. variables names should be of any reasonable length
4. variable names are case sensitive. that is name, Name, nAme, name are all different variable names
5. variable name shouldn't be a reserve word

Shell Keywords:-

Keywords are the Words whose meaning has already been explained to the shell. The keywords are also called as "Reserve Words".

The list of keywords available in Bourne Shell are

Echo	Until
if	trap
Read	case
else	wait
Set	esac
fi	eval
Unset	break
while	exec
Readonly	continue
do	ulimit
Shift	Exit
Done	Umask
Export	return
For	

1.Echo

echo command is used to display the messages on the screen and is used to display the value stored in a shell variable.

Eg 1: \$echo "tecnosoft is a training institute"

Tecnosoft is a training institute

Note: double quotes are option in echo statement

Eg 2: \$echo "today date is : `date` "

Today date is sat mar 4 04:40:10 IST 2005

Note: the unix command should be in back quotes in echo statement otherwise it treat as text

Eg 3: \$echo "my file has `wc -l file1` lines

My file has 10 lines

Eg 4: \$echo my log name is : `logname`"

My logname is tecnosoft

Eg 5: \$echo "my present working directory is : `pwd`"

My present working directory is : /usr/tecnosoft/abc

Shell variables

Eg 1: \$a=10

Note: there are no predefined data types in unix. Each and every thing it treat as character. Each character occupies 1 byte of memory.

Eg 2: \$b=2000

In above example a occupies 2 bytes and b occupies 4 bytes of memory

Reading of variable

\$ is the operator to read variable value

Eg 1: \$n=100

\$echo \$n

100

Eg 2: \$name="Tecnosoft"

\$echo \$name

tecnosoft

\$echo welcome to \$name

welcome to tecnosoft

eg 3: \$now=date

\$now

sat mar 4 04:40:10 IST 2005

Eg 4: \$mypath=/usr/tecnosoft/abc/a1/a2

\$cd \$mypath

now it changes to a2 directory, to check say at \$ prompt pwd command

\$pwd

/usr/tecnosoft/abc/a1/a2

Null Variables

A variable which has been defined but has not been given any value is known as a null variable. A null variable can be created in any of the following ways.

1.\$n=""

2.\$n=''

3.\$n=

\$echo n

on echoing a null variable, only a blank line appears on the screen.

Constant:

Constant is a fixed value. It doesn't change during execution of the program

\$a=20

\$readonly a

when the variable are made readonly, the shell does not allow us to change their values. so a value can read but can't change.

Note: If we want the shell to forget about a variable altogether, we use the unset command.

\$unset a

on issuing the above command the variable a and with it the value assigned to it are erased from the shell's memory.

Escape sequence characters

Sl.No	Escape character	Meaning	Example
1	"\07"	Bell sound	\$echo "hello\07"
2	"\n"	New line	\$echo "hello \n tecnosft" hello tecnosoft
3	"\t"	Tab space	\$echo "hello \t tecnosoft" hello tecnosoft
4	"\b"	Back space	\$echo "hello\btecnosoft helltecnosoft
5	"\r"	Carriage return i.e it places cursor beginning of the current line	\$echo "welcome to \r tecnosoft" tecnosofto
6	"\c"	It places cursor at end of the current statement	\$echo "hello... \c" hello...
7	"\""	Double quote	\$echo "\"hello\"" "hello"
8	"\'"	Single quote	\$echo "\'hello\'" 'hello'
9	"\""	Back slash	\$echo "\\hello\\ \\hello\\
10	"\033[0m"	Normal characters	\$echo "\033[0m tecnosoft" tecnosoft
11	"\033[1m"	Bold characters	\$echo "\033[1m tecnosoft" tecnosoft
12	"\033[5m"	Blinkin charcters	\$echo "\033[5m tecnosoft"
13	"\033[7m"	Reverse video charcters	\$echo "\033[7m tecnosoft"

Shell Programs

1.write a program to display list of files,the current working user's list and present working directory

vi sp1

```
ls -x  
who  
pwd
```

:wq(save and quit)

execution of shell program:

sh is the command to execute bourne shell programs

```
eg : $sh sp1  
or  
$chmod 744 sp1  
$sp1
```

2)write program to display address

vi sp2

```
echo "Tecnosoft Solutions"  
echo "No:109, 1st floor,"  
echo "Annapurna block,"  
echo "Aditya Enclave,"  
echo "Ameerpet,"  
echo "Hyderabad"  
echo "ph.no:5583966 / 94403 27911 "
```

:wq(save and quit)

3)write program count no of users are currently logged into the system

vi sp3

```
echo "There are `who | wc -l` users"
```

```
:wq(save and quit)
```

4)write program read name and diplay

vi sp4

```
echo "what is your name?"
```

```
read name
```

```
echo "hello $name"
```

```
:wq(save and quit)
```

Note :- read is command to read variable value form the user.read command reads value form keyboard upto space or enter key.

Eg 1: read a

Eg 2: read a b c

5)write program read 2 numbers and display

vi sp5

```
echo "enter 2 numbers:\c"
```

```
read a b
```

```
echo "your numbers are $a and $b"
```

```
:wq(save and quit)
```

OPERATORS:

- 1.Arithmetic operators
- 2.Relational operators
 - a.Numeric comparision operators
 - b.String comparision operators
3. Logical operators

1.Arithmetic Operators

Operator	Meaning
+	Addition
-	Substraction
*	Multiplication
/	Division
%	Modulus division

2.Numeric Comparision Operators

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

3.String Comparision Operators

Operator	Meaning
>	Greater than
<	Less than
=	Equal to
!=	Not equal to

4.Logical Operators

Operator	Meaning
-a	Logical AND
-o	Logical OR
!	Logical NOT

6)write program to read 2 numbers and display sum,difference,product and division

vi sp6

```
echo "enter 2 numbers"
read a b
c=`expr $a + $b`
echo "a+b=$c"
c=`expr $a - $b`
echo "a-b=$c"
c=`expr $a \* $b`
echo "a*b=$a"
c=`expr $a / $b`
echo "a/b=$c"
```

:wq(save and quit)

Note : expr is the command to evaluating arithmetic expressions. But expr is capable of carrying out only integer arithmetic.

7)write program to read 2 float numbers and display sum,difference,product and division

vi sp7

```
echo "enter 2 float numbers"
read a b
c=`echo $a + $b | bc`
echo "a+b=$c"
```



```
c=`echo $a - $b | bc`  
echo "a-b=$c"  
c=`echo $a \* $b | bc`  
echo "a*b=$a"  
c=`echo $a / $b | bc`  
echo "a/b=$c"
```

```
:wq(save and quit)
```

CONTROL STATEMENTS:

There are four types of control instructions in shell. They are:

1. Sequence Control Instruction
2. Selection or Decision Control Instructions
3. Repetition or Loop Control Instruction
4. Case Control Instruction

The sequence control instruction ensures that the instructions are executed in the same order in which they appear in the program. Decision and case control instructions allow the computer to take a decision as to which instruction is to be executed next. The loop control instruction helps computer to execute a group of statements repeatedly.

Decision Control Statement

1. if-then-fi statement
2. if-then-else-if statement
3. if-then-elif-fi statement
4. case-esac statement

1.if-then-fi statement**syntax:**

if control command

fi

The if statement of unix is concerned with the the exit status of a command. The exit status indicates whether the command was executed successfully or not. The exit status of a command is 0 if it has been executed successfully, 1 otherwise.

8) write a program to change directory**vi sp8**

echo enter directory name

read dname

if cd \$dname

then

echo "changed to \$dname"

pwd

fi

:wq(save and quit)

2.if-then-else-fi Statement**syntax**

if condition

then

else

fi

the exit status of the control command is 0 then it executes then statements otherwise it executes else statements.

9)Write a program to copy a file**vi sp9**

```
echo enter source filename and target file name
read src trg
if cp $src $trg
then
echo file copied successfully
else
echo failed to copy the file
fi

:wq(save and quit)
```

10)Write a program to search string in a file**vi sp10**

```
echo "enter a file name"
read fname
echo "enter to string to search"
read str
if grep $str $fname
then
echo "$str is found in the file $file"
else
echo "$str is not found in $fname"
fi

:wq(save and quit)
```

11) write a program find greatest number of 2 numbers

vi sp11

```
echo enter two numbers
read a b
if [ $a -gt $b ]
then
echo $a is the greatest value
else
echo $b is the greatest value
fi
```

:wq(save and quit)

12) write a program to check given no is even or odd

vi sp12

```
echo enter a number
read n
if [ `expr $n % 2` -eq 0 ]
then
echo $n is even number
else
echo $n is odd number
fi
```

:wq(save and quit)

The Test Command

If constructs depends upon whether or not the condition results into true or not.

If constructs are generally used in conjunction with the test command.

The test command helps us to find the contents of a variable, the number of variables and the type of file or kind of file permission. The test command returns an exit status after evaluating the condition.

Syntax:

If test condition

Then

Commands

Else

Commands

fi

13) write a program to check how many users working on the system**vi sp13**

total=`who | wc -l`

if test \$total -eq 1

then

echo "you are the only user working..."

else

echo "There are \$total users working..."

fi

:wq(save and quit)

14) Write a program to check given number is +ve or -ve number**vi sp14**

echo "enter a number"

read num

if test \$num -gt 0

then

echo "\$num is +ve number"

else

echo "\$num is -ve number"

fi

:wq(save and quit)

15) write a program to find student result**vi sp15**

```
echo "enter three subject marks:"
read m1 m2 m3
if [ $m1 -ge 40 ]
then
    if [ $m2 -gt 40 ]
    then
        if [ $m3 -gt 40 ]
        then
            echo "PASS"
        else
            echo "FAIL"
        fi
    else
        echo "FAIL"
    fi
else
    echo "FAIL"
fi
:wq(save and quit)
```

16)write a program to print greeting**vi sp16**

```
hour=`date | cut -c 12,13`

if [ $hour -ge 0 -a $hour -le 11 ]
then
    echo "Good Morning"
else
    if [ $hour -ge 12 -a $hour -le 17 ]
    then
        echo "Good Afternoon"
```

```
    else
        echo "Good Evening"
    fi
fi
```

```
:wq(save and quit)
```

File Test Commands

The test command has several options for checking the status of a file.

Option	Meaning
-s file	True if the file exists and has a size greater than 0
-f file	True if the file exists and is not a directory
-d file	True if the file exists and is a directory file
-c file	True if the file exists and is character special file
-b file	True if the file exists and is a block special file
-r file	True if the file exists and you have a read permission to it
-w file	True if the file exists and you have a write permission to it
-x file	True if the file exists and you have a execute permission to it

17) Write a program to check for ordinary file and display its contents

```
vi sp17
```

```
echo enter a file name
read fname
if test -f $fname
then
cat $fname
else
echo "given file is not ordinary file"
fi
:wq(save and quit)
```

18) Write a program to check give file is ordinary or directory file

vi sp18

```
echo "enter a file name:"
read fname
if [ -f $fname ]
then
cat $fname
elif [ -d $fname ]
then
ls
else
echo $fname is not file and not a directory
fi

:wq(save and quit)
```

19) Write a program to check read permission

vi sp19

```
echo "enter a file name"
read fname
if [ -r $fname ]
cat $fname
else
chmod u+r $fname
cat $fname
fi
```

:wq(save and quit)

20) Write a program to append data to the file

vi sp20

```
echo "enter a filename"
read $fname
if [ -f $fname ]
then
    if [ -w $fname ]
    then
        echo "enter data to file to stop press ctrl+d..."
        cat >> $fname
    else
        chmod u+w $fname
        echo "enter data to file to stop press ctrl+d..."
        cat >> $fname
    fi
else
    echo "enter data to file to stop press ctrl+d..."
    cat > $fname
fi

:wq(save and quit)
```

String Test Commands

Condition	Meaning
String1 = string2	True if the strings are same
String1 != string2	True if the strings are different
-n string1	True if the length of string is greater than 0
-z string	True if the length of the string is zero

21)write a program to compare two strings

vi sp21

```
echo "enter first string:"  
read str1  
echo "enter second string:"  
read str2  
if test $str1 = $str2  
then  
echo "both strings are equal"  
else  
echo "strings are not equal"  
fi
```

:wq(save and quit)

22)Write a program check given string is empty or not

vi sp22

```
echo enter a string  
read str  
if [ -z $str ]  
then  
echo "string is empty"  
else  
echo 'given string is not empty'  
fi
```

:wq(save and quit)

Case Control Statement**Syntax**

Case value in
Choice1)

;;

choice2)

;;

choice3)

;;

:

:

choicen)

;;

*)

;;

esac

Firstly, the expression following the case keyword is evaluated. The value that it yields is then matched, one by one against the potential choices(choice1, choice2 and choice3 in the above form). When a match is found, the shell executes all commands in that case up to;;. This pair of semicolons placed at the end of each choice are necessary.

23)Sample program for case**vi sp23**

```
echo "enter a number between 1 to 4"\c"
read num
case $num in
1)echo "you entered 1"
;;
2)echo "you entered 2"
;;
3)echo "you entered 3"
;;
4)echo "you entered 4"
;;
*)echo "invalid number. enter number between 1 to 4 only"
;;
esac
```

:wq(save and quit)

24)Write a program to check given character is upper case alphabet or lower case alphabet or digit or special character**vi sp24**

```
echo "enter a single character"
read ch
case $ch in
[a-z])echo "you entered a small case alphabet"
;;
[A-Z])echo "you entered a upper case alphabet"
;;
[0-9])echo "you entered a digit"
;;
?)echo "you entered a special character"
;;
```

```
*)echo "you entered more than one character"
```

```
;;
```

```
esac
```

```
:wq(save and quit)
```

25)write a program to display file contents or write on to file or execute based on user choice

vi sp25

```
echo "enter a file name:\c"
```

```
read fname
```

```
echo "Main Menu"
```

```
echo "-----"
```

```
echo "r. read mode"
```

```
echo "w.write mode"
```

```
echo "x. execute mode"
```

```
echo "enter mode:\c"
```

```
read mode
```

```
case $mode in
```

```
r)
```

```
if [ -f $fname -a -r $fname ]
```

```
then
```

```
cat $fname
```

```
fi
```

```
;;
```

```
w)
```

```
if [ -f $fname -a -w $fname ]
```

```
then
```

```
echo "enter dat to file at end press ctrl+d:"
```

```
cat>$fname
```

```
fi
```

```
;;
```

```
x)
```

```
if [ -f $fname -a -x $fname ]
```

```
then
```

```
chmod u+x $fname
$fname
fi
;;
*)echo "you entered invalid mode..."
.;;
esac
```

:wq(save and quit)

26)write a menu driven program which has following options:

- 1.contents of current directory
- 2)list of users who have currently logged in
- 3)present working directory
- 4)calendar
- 5)exit

vi sp26

```
echo "Main Menu"
echo "-----"
echo "1.list of files"
echo "2.list of users"
echo "3.present working directory"
echo "4.display calendar"
echo "5.exit"
```

```
echo "enter your choice:\c"
read choice
case $choice in
1) ls -x
;;
2)who
;;
3)pwd
;;
```

```
4)echo "enter month and year"
  read m y
  cal $m $y
  ;;
5)banner "thank you"
  sleep 1
  exit 0
  ;;
*)echo "choice is wrong try again"
  ;;
esac

:wq(save and quit)
```

The Exit Statement

To terminate execution of shell program

Looping Control Statements

A loop involves repeating some portion of the program either a specified no of times or until a particular condition is being satisfied. There are three methods by way of which we can repeat a part of a program. There are

- 1) Using a **while** statement
- 2) Using a **until** statement
- 3) Using a **for** statement

1) While Statement

syntax

```
while [ condition ]
do
-----
-----
-----
done
```

the statements within the while loop would keep on getting executed till the condition is true. When the condition is false, the control transfers to after done statement.

27)Write a program display numbers 1 to 10**vi sp27**

```
echo "the number form 1 to 10 are:"
i=1
while [ $i -le 10 ]
do
echo $i
i=`expr $i + 1`
done

:wq(save and quit)
```

28)Write a program copy file from root to user directory**vi sp28**

```
flag=1
while [ $flag -eq 1 ]
do
echo "enter a file name:\c"
read fname
cp $fname /usr/tecno/$fname
echo "$fname copied....."
echo "do u wish to continue [1=yes/0=no]:"
read flag
done

:wq(save and quit)
```


The Break Statement

When the keyword break is encountered inside any loop, control automatically passes to the first statement after the loop.

The Continue Statement

When the keyword continue is encountered inside any loop, control automatically passes to the beginning of the loop.

29) write a program display file contents if file existing

vi sp29

```
x=0
while test $x = 0
do
echo "enter a file name:\c"
read fname
if test ! -f $fname
then
echo "$fname is not found...."
continue
else
break
fi
done
cat $fname | more
```

:wq(save and quit)

The Until Loop**Syntax**

until [condition]

do

done

The statements within the until loop keep on getting executed till the condition is false. When the condition is true the control transfers to after done statement.

30)Write a program print numbers 1 to 10**vi sp30**

```
i=1
until [ $i -gt 10 ]
do
echo $i
i=`expr $i + 1`
done
```

```
:wq(save and quit)
```

The True and False Command

To execute the loop an infinite no of times.

31)Write sample program for true command**vi sp31**

```
while true
do
clear
banner "hello"
sleep 1
clear
banner "Tecnosoft"
sleep 1
done
```

```
:wq(save and quit)
```

Note: The above program executes continuous to stop execution press ctrl + break.

32) Write sample program for false command

```
vi sp32
```

```
until false
do
clear
banner "hello"
sleep 1
clear
banner "Tecnosoft"
sleep 1
done
```

```
:wq(save and quit)
```

The Sleep Command

The sleep command stops the execution of the program the specified no of seconds.

The For Loop Syntax

```
For variable in value1 value2 value3 . . . . . value
do
-----
-----
-----
done
```

The for allows us to specify a list of values which the variable in the loop can take. The loop is then executed for each value mentioned in the list.

33)write a program to demonstrate for loop

vi sp33

```
for i in 1 2 3 4 5
do
echo $i
done
```

:wq(save and quit)

34)Write a program to demonstrate for loop

vi sp34

```
for i in tecno soft is a training institute.
do
echo $i
done
```

:wq

35)Write a program to display all files in current directory

vi sp35

```
for I in *
do
if test -f $i -a -r $i
then
cat $i | more
sleep 1
clear
fi
done
```

:wq(save and quit)

36) Write a program to display all sub-directories in the current directory

vi sp36

```
for I in *  
do  
if [ -d $i ]  
then  
echo $i  
fi  
done
```

:wq(save and quit)

Positional Parameters

When the arguments are passed with the command line, shell puts each word on the command line into special variables. For this, the shell uses something called as "positional parameters". These can be thought of as variables defined by the shell. They are nine in number, named \$1 through \$9.

Consider the following statement, where sp37 is any executable shell script file and the remaining are the arguments.

\$sp37 tecnosoft solutions is a computer training and development institute

on entering such command, each word is automatically stored serially in the positional parameters. Thus, \$0 would be assigned sp37, \$1 would be assigned "tecnosoft", \$2 "solutions", \$3 is "is" and so on, till "institute", which is assigned to \$9.

37) write program to copy a file using positional parameters

vi sp37

```
if cp $1 $2
then
echo 'file copied successfully'
else
echo "failed to copy"
fi
```

:wq(save and quit)

\$chmod 744 sp37

\$sp37 a1 a2

in above command it passes a1 to \$1 and a2 into \$2, then it copies a1 contents to a2.

S* - Contains entire string of arguments

S# - Contains the no for arguments specified in the command