LECTURE NOTES

UNIX PROGRAMMING

Name Of The Programme	**B.Tech-CSE**
Regulations	**R-16**
Year and Semester	**III Year I Semester**
Name of the Course Coordinator	**Mr A.Sandeep Kumar**
Name of the Module Coordinator	**Dr.G.J.Sunny Deol**
Name of the Program Coordinator	**Mr.N.Md.Jubair Basha**

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

KALLAM HARANADHAREDDY INSTITUTE OF TECHNOLOGY

NH-5, Chowdavaram Village, Guntur, Andhra Pradesh, India
Approved By AICET, New Delhi, Permanently Affiliated to JNTUK
Kakinada Accredited By NBA, Accredited By NAAC with A Grade

ACADEMIC YEAR 2020-21

Unit-1

Introduction to unix-Brief History-What is Unix-Unix Components-Using Unix-Commands in Unix-Some Basic Commands-Command Substitution-Giving Multiple Commands

- **Brief history:**

Many schools have contributed to Unix development, but the initial contribution by bell laboratory of AT and T and the university of California, Berkley are notable.

Bell laboratory contribution:

- In 1965 ,Massachusetts institute of technology(MIT),general electrical and bell laboratories of AT & t worked on a joint venture project called multics, which intended to develop a multi-user operating system.
- In 1969 ,AT&T withdraw itself from the multics project as the process was not satisfactory.
- On the basics of the ideas acquired while working on multics, ken Thomson and Dennis ritche developed an operating system(os) called UNICS (uniplex informatics and computing system) during the latter part of 1969.
- UNICS was developed completely is assembly language and so it was not portable.
- To achieve portability, Thomson considered implementing the system in high level language which leaded to the development of 'c' language by Ritche in 1973.
- Ritche completely recorded th entire unix system during the year 1973.
- In details of the unix implementation in'c' was mode public through a paper published in 1974.
- A system called unix system V was announced in 1963.
- With this release AT&T assured the upward compatibility all its features releases.

- The most important of the releases is the system V release 4 (SUR4) in 1991.

UCB's contribution:

- The team at Berkeley was responsible for many important technical contributions as well as the development of useful utilities.
- The development of UCB towards unix are
- EX: editor, vi editor, c-shell (csh).
- Researchers at Berkeley labs were also released some BSD-unix (Berkeley software distribution) during the spring of 1978.
- These BSD releases are 4.0 BSD(1980), 4.1 BSD(1981), 4.2 BSD(1983), 4.3 BSD(1986) and 4.4 BSD (1993).

Why so many variants:

- One of the main reasons for many variants of unix is that being a telephone company, At& T was not permitted to sell computer-based products.
- However, it could do so free of cost or for a nominal. BSD was also giving its products for free of cost , many obtained the copies of unix and worked on them .This resulted in a number of unix variants.
- Another important reason was that these systems were developed mostly by researchers for researchers and revised constantly to suit different requirements of many unix variants.

What is unix:

- Unix is an operating system .
- An operating system is a software that acts as an interface between the user and computer h/w.
- An operating system acts as a resource manager.
- Here resources mean hardware resources like the processor, the main memory, the hard disk, i/o devices and other peripherals.
- In addition to being a multi-user operating system unix gives its users, the feeling of working an independent computer system.

  Unix also provides communication facility with other users who are connected to the system either directly or indirectly via certain sort of networking

Sailent features of unix:

1. **Multi- tasking:**
   - Unix is a multi-tasking operating system.
   - It has the ability to support concurrent execution of 2 or more active process (instance of program in execution is called process).

2. **Multi-user operating system:**
   - Unix is multi-user operating system.
   - It has ability to support more than one user to login into the system simultaneously and execute programs.

Difference: the diiference between multi –tasking and multi-user system is,

- In multi-tasking, different tasks the process running concurrently belongs to one user.
- In multi-user environment different tasks belong to diff users.

3. **Portable**
   - Unix operating system is highly portable**.**
   - Compared to other os, it is very easy to port.unix on to different hardware platforms with minimal or no modifications at all i.e., because it ways developed in c language which is high level language

4. **Inter machine communication:**
   - The development of communication protocols like tcp/ip has made possible by unix operating system to usera to exchange information in the form of email and shared data.

5. **Security:**
   - **As Unix is multi-user system, there is every chance that a user may intrude into a another users area either unintentionally. But, Unix offers solid security at various levels beginning from the system startup level to accessing files as well as saving data in an encrypted form.**

6. **Library of utility and commands:**
   - Unix has good library of utilited and commands that have been used to develop newer applications.

7. **File and directory system:**
   - One of the very important key feature of any unix system is that allows users to organize and maintain these files/directories easily and maintain these files/directories easily and efficiency.

UNIX components:

Unix contains 3 major components.

1. The kernel
2. The shell
3. The file system.

In addition to these components all unix systems also contains genral utility programs.

1. The kernel:

   - The kernel is the heart of any unix os.
   - This kernel is relatively a small piece of code written in 'c' that is embedded on the hardware.
   - Every unix system has a kernel that gets automatically loaded on to the memory as soon as system is boosted.
   - The kernel is the only component that can communicate with all hardware directly.
   - Kernel manages all the system resources like memory and i/o devices ,allocated time between users and processes in the case of multi-user environment, decides process priorities, manages inter process communication and performs many other such tasks.

Monolithic kernels:

Earlier ,all the programs that were part of a kernel , were integrated, together and moved onto the memory during booting. Such integrated kernels are referred to as,' monolithic kernels'.

Micro kernel: Now -a-days ,all the programs are grouped into different modules and only the just necessary module is moved on to the memory during boating. This just necessary and sufficient module consisting of a small set of kernel program is called a " micro kernel"

Shell:

Every unix system has atleast one shell.A shell is a program that sites on kernel and acts as an agent or interface between the users and the kernel and hence the hardware.A shell is a command interpreter or processor at which the user can type in any unix command.

Types of shells:

There are different types of shells available. some of them are.

✓ The bourne shell(sh):

This is the first mojar shell to be developed and is named after its author, Stephen bourne.

- This is the most common shell and is disturbuted as the standard shell on almost all unix systems.
- This is the most common shell on almost all unix systems.

✓ C shell(csh):

- Bill joy , developed this shell at VCB as a part of the BSD release .
- It is called c shell because is syntax and usuage is very similar to the 'c' programming language .

✓ **Korn shell(ksh):**

- This shell was developed by david korn at AT &T bell labs.
- It has the both features of bourne shell and c shell and is one of the widely used shells.

✓ **Bourne-again shell(bash):**

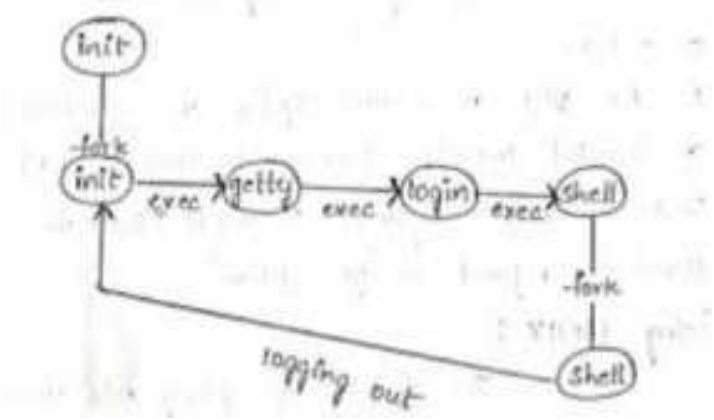
- This shell was developed by BFOX and C raney at free software foundation.
- This is a free ware shell.

2. The file system:

- A file system is another major component of a unix system.
- Unix treats every thing including hardware devices as a file.
- All the files in a unix system are organized in an inverted tree-like hierachial structure (2.1) . This structured arrangement in which files are stored is refreed as "file system".

USING UNIX

- The process of getting into unix environment is known as 'logging in' into system.
- The sequence of events in a complete log in process is as fallows:
- The user enters a login name of the getty's login prompt on the terminal.
- Gety executes the login program with the login name as the argument.
- Login requests for a password and validates it against/etc/passwd.
- Login sets up the TERM environment startup file like profile.
- The shell then prints a prompts, usually a $ and a % symbol and waits for future input. This indicates the successful entry made into a unix environment with a
  paper shell.



- When the user completes the session with system the comes out of unix environment .The process of coming out of unix environment is known as logging out.

The shell prompt:

- Successful login into a unix system is indicated by the appearance of a prompt called the shell prompt or system prompt on the terminal.
- The character that appears as a prompt depends on the shell used.
- $(dollar)-> bourne and korn shells(sh,bash,ksh)
- %(percent)-> C shell (sh and tesh)
- #(hash)->any shell as root.

Commands in Unix:

Unix has large number of commands.

Types of unix commands:

- There are 2 types of unix commands. They are
  1. External commands
  2. Internal commands

1. External commands:

A command with an independent existence in the form of a separate file is called an external command.

Ex: cat and ls . these are independently existed in a directory called the / bin directory .

When external commands and given the shell reaches these command files with the help of 'PATH' variable.

2. Internal commands:

A command that does not have an independent existence is called an internal command.
Ex. Echo,mkdir,cd,etc;

The routines for internal commands will be a part of another program (or) routine.
Ex. Echo command is internal command will be a part of another program(or) routine is the part of shell's routine "sh".

Some basic commands:-

Unix has several hundreds of commands with it. some of them are.

1. echo
2. tput
3. lty
4. who
5. uname
6. date
7. cal
8. calendar
9. passwd
10. lock
11. banner
12. cat etc..;

The echo command:-

Use:-

- The echo command is used to display messages.
- It takes zero,one(or) more no of arguments.
- Arguments may be given either as a series of individual symbols or as a string with in a pair of double quotes(" ").
- Example:

```
1 .$echo                    #Blank line is displayed
   $

2.$echo Iam studying information technology

  Iam studying information technology

  $

3.$echo "Iam studying unix programming".
```

Iam studying unix programming.

The tput command:- Use:-

- This command is used to control movement of the cursor on the screen as well as to add the certain features like blinking ,boldface and underlining to the displayed message on the screen.
- Examples :-
  - $tput clear
- Clears the screen and puts the cursor at the left top of the screen
  - $tput cup 10 20
- This command along with cup a argument and certain co-ordinates values is used to position the cursor at any required position on the screen.
  - In the above example the cursor will be placed at the tenth and twentieth column on the screen.
  - $tput lines
    48
    $
- To know the current no of rows on terminal.
  - $tput cols
    142
    $

The tty command:-

In unix, every terminal is associated with a special file, called the device file. All the device files will be present in /dev directory.

Use:-

Examples:

-

$tty

/dev/tty 01

$

- Here tty01 is the file name and will be available in /dev directory.

The who command:-

Use:-

The user can know the login details of the current users by using the who command .

Ex:- $who

```
Root console nov 19  09:35
Mgv   tty01     nov   19   09:40
dvm   tty02     nov   19   09:45
```

- The first column shows the name of the users,the second column shows devicename and third column shows login time.
  Ex:-
  $who am i
  Mgv tty01 nov 19 9:40
- The self-login details of a user can be obtained in the above example.

The uname command:-

Use:

-

- using this command one can know the details of one's unix system.
- Unix has many variants,when this command is used .it gives the name of unix system being used by the user.
  Ex:-
  $uname
  Linux
  $

The date command:-

- Using this command, the user can display the current date along with thetime nearest to the second.
  Ex:-
- $date
  Sat jan 18 11:58:00 lst 2018
  $
- $date +%m
  09
  $
- Displays only month in numeric form.
- %date +%h
  Sep
  $
- Displays the name of the month.

The cal command:-

Use:-

- This command is used to print the calendar of a specific month or a specific year.
- When this command is used without any arguments the calendar of the current month of the current year will be printed.

Ex:- $cal 02 2018

Jan 2018

Su	mo	tue	wed	th	fr	sa
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	

$

The passwd command:-

- As unix is a multi-user system due to which there is always a security threat.
- The simplest and most widely used by all individual users is use of passwds.
- For every new user,the administrator ,permita or authorities them by assigning unique password to each.

Use:-

- A user can change his/her password by using this command.

Ex:- $passwd: *****
Old passwd: *******
New passwd: *******
$

When a user attempts to change his/her password the system first asks for old password ,if it is correctly entered the system asks twice to enter new password and srts it as current password

The lock command:-

Use:-

- The lock command is used for locking a session for any required amount of time.
- This command is used generally without any arguments.

- By default the user can lock it for 30 min.
- The locking period can be changed by assigning a different value for the system variable **DEFLOGOUT.**
- When the lock command is given, the terminal asks for a password twice.

Eg: $lock -45 #locks for 45 min

The banner command:

- It is used to display banners or pasters.
- This command simply produces a blowup version of the characters that are supplied with the command.
- Ex: $ banner hello

The cat command:

- This is used to create small unix files. A file can be created by writing a command line.
- Eg: $ cat > review
  A > symbol following the command
  Means that the output
  Goes to the file name following it
  <ctrl –d>
- In the above example, 'review' is the file name after executing $cat > review, the $ disappears and the presents '>' symbol which means the system is ready to accept the contents to be placed in the file.
- If the file contents are over to exit, we have to use the command <ctrl –d>

The bc command:

- The bc command is both a calculator and a small language for writing numerical programs.
- By using this one can perform all the usual arithmetic operations as well as change of bases in the range of 2 -16
- Ex: $ bc
  Sqrt 55
  7
  Quit
  $

The spell and ispell command:

- The spell command is the first program that was developed to check for words that are wrongly spelt in a document.
- This command displays a list of misspelled words in the document used as argument.
- Eg: $cat spell – ux
  This is example
  I am using cat command

*
$ spell spell .
ux Example
Command
$

Ispell:

- The ispell command is an interactive spell – check program available in linux . when used, this command displays a screen full of information in tree sections.
- Eg: $ispell spell.ux

This is example
I am using cat command

1. Example
2. Exempler
3. Exampled

I. Ignore ignore all
II. Replace replace all
III. Add exit

The man command:

- The man command prints the details of any command of a utility on the screen.

Ex: $man pwd

Pwd(1) user commands pwd(1)
Name
Pwd - print working directory pwd(1)
Synopsis
Pwd
Description
Pwd prints the pathname of the working directory Notes

Command substitution:

- In unix, it is possible to run a command within another command.
- Ex: The date command can be run within the echo command by writing command as
- $ echo today the date is ‘date’
- Today the date is the june 7 16:25:00 2020
- when more trend is executed the shell while parsing the parameters list of the echo command treats the words that are back quoted as a command executes it and subsititutes the result of this execution at the corresponding position in the parameter list. This process is known as command subsistution.

Giving multiple commands:

- Normally a single command is given to the shell at its prompt. However there are many situations when more than one command is given in a single command line.
- One of the ways of giving the multiple commands is to use a semicolon (;) between successive commands.
- Ex: echo "giving multiple command" ; who ; who, date does not mutually interact with each other in any manner.
- They are executed one after the other, from left to right as they appear in the command line.

Unit-II

The File system –The Basics of Files-What's in a File-Directories and File Names-Permissions-I Nodes-The Directory Hierarchy, File Attributes and Permissions-The File Command knowing the File Type-The Chmod Command Changing File Permissions-The Chown Command Changing the Owner of a File-The Chgrp Command Changing the Group of a File

1.INTRODUCTION TO UNIX FILE SYSTEM:

Computers can store information on various storage media, such as Magnetic disks, Magnetic tapes, and Optical disks. A file is a named collection of related information that is recorded on secondary storage. File is a memory block of secondary storage device like disk, magnetic tape that logically stores related records permanently. Operating system provides various components to manage the data on a disk system permanently. Such as..

File manager: It manages the allocation of disk-space for the storage of user files.

Buffer manager: It is responsible for fetching data from disk storage into main memory buffers for processing, and then writing the updated data back onto the disk.

Disk Space manager: It is responsible for managing disk space on disk.

File system – provides a powerful & flexible way to organize and manage user information. File system is a Group of files and relevant information are organized in a well defined order and stored on a Hard Disk . Operating System defines a File System on Devices which is usually Hierarchical file system including UNIX.

Directory- It is a **File** that keeps a **list** of other files and sub directories on the File System .

UNIX has single File system. Devices are mounted into this File System. The disk space allotted to a linux/unix file system is made up of "**blocks**" , each of which are **512** bytes. To find out block size on your file system, use the **cmchk** command.

```
$cmchk
BSIZE=
2048
```

Fig: Disk File Block Structure

Types of File Blocks on a disk:

All blocks belonging to the file system are logically divided into :

1) Boot block (Master Boot Record)
2) Super block
3) Inode block
4) Data block

Boot block: It is the **first** block in the storage disk, numbered 0 , is called the **boot** block. It is normally unused by the filesystem and is set aside for booting procedure. It represents the beginning of the file system. It contains a program called "**Boost Strap Loader**". This program is executed during '**booting'** process to load the file system into the main memory. (Booting - means starting process of system when computer is turned ON.

Note: All file system contain only **one Boot block**.

Super block: It is the **second** block of every file system and is numbered 1. It describes the **entire file system** and its **state.** It is used to control the allocation of blocks . It contents of super block are:

a) The size of file and status of file name
b) Details of free blocks and their addresses. A file occupies a minimum of 1 Block, even it has one character in it. A block size can be of 512 bytes or multiple of 512 bytes.
c) Size of inode section of the file system.
d) It specifies "how large the file system is , how many maximum files it can accommodate, how many more files it can accommodate, how many more files can create etc."

Inode block: It contains most information regarding the **files**. Every file will have an entry in this area, identified by a **64**-bit structure referred to as I-node. The information related to all the files (not the contents) is stored in an **inode table** on the disk.
For each file, there is an inode entry in the table. Each entry is made up of **64** bytes in the table. These details are..

1) Owner of the file
2) Group to which the owner belongs
3) Type of a file
4) File access permission
5) Date and time of last access
6) Date and time of last modifications
7) No of links to the file
8) Size of the file
9) Address of blocks where the file is physically present.

Data blocks: Contains the actual file contents. An allocated block can belong to only one file in the file system.This block can't be used for storing any other file's contents unless the file which it originally belonged is deleted.

TYPES OF FILES:

A file in a UNIX system having the following types:

- Regular file / Ordinary file (-)
- Directory file (d)
- Device or Special file (c or b)
- Hidden files or Dot files (.)
- FIFO file (f)
- Link file (l)
- Socket file (s)

1. Ordinary files – An ordinary file is a file on the system that contains data, text, or program instructions.

- Used to store your information, such as some text you have written or an image you have drawn. This is the type of file that you usually work with.
- Always located within/under a directory file.
- Do not contain other files.

2. Directories – Directories store both special and ordinary files. UNIX directories are equivalent to folders. A directory file contains an entry for every file and subdirectory that it houses. If you have 10 files in a directory, there will be 10 entries in the directory. Each entry has **two** components.

i) The Filename
ii) A unique identification number for the file or directory (called the inode number)

Features:

- Branching points in the hierarchical tree.
- Used to organize groups of files.
- May contain ordinary files, special files or other directories.
- Never contain "real" information which you would work with (such as text). Basically, just used for organizing files.
- All files are descendants of the root directory, (named /) located at the top of the tree.
- In long-format output of ls –l , this type of file is specified by the "d" symbol.

3. Device or Special Files – Used to represent a real physical device such as a printer, tape drive or terminal, used for Input/Ouput (I/O) operations. On UNIX systems there are **two** types of special files for each device, **character** special files and **block** special files.

- **Character special file:** A file related to I/O and used to model serial I/O devices like Terminals ,Printers & Networks.
- **Block Special file:** A file related to memory and used to model devices like Disk drives and Magnetic tapes.

4.Hidden files or **Dot** files : A dot(.) character also used to construct a file system. Any file name beginning with a .(dot) character is called a hidden file a dot file. These are used to store some specific information regarding configuration of system. Ex: .profile, .bashrc etc.

5.FIFO file : used for making communication between two or more application programs.

2. THE BASICS OF A FILE:

FILE : A File is a sequence of Records or Bytes . A byte is a small chunk of information, typically 8 - bits long. Here, a **byte** is equivalent to a **character**. Data is usually stored in the form of records.

RECORD: It is a collection of Fields. Each record consists of a collection of related data values or items, where each value is formed of one or more bytes.

Files are the **building blocks** of any operating system. Every file has some properties .

The UNIX File Attributes are

1) File type - specifies what type of file it is.
2) Access permission - the file access permission for owner, group and others.
3) Hard link count - number of hard link of the file
4) UID - The file owner/ User ID.
5) GID -The file Group ID .
6) File size - The file size in bytes.
7) I-node no - The system I-node (Information Node) number of the file.
8) File system id - The file system id where the file is stored.
9) Last access time - the time, the file was last accessed.
10) Last modified time - the file, the file was last modified.
11) Last change time - the time, the file was last changed.

Creating A Small File:

The Unix environment allows the user to create a file to maintain their information. The creation of a file requires some editors (such as ed , vi editor) or commands (cat , touch etc.)

// creation of files with cat command

The cat (short for "concatenate") command is one of the most frequently used command in Linux/Unix like operating systems. Cat command allows us to create single or multiple files, view contain of file, concatenate files and redirect output in terminal or files.

Syntax: cat [options] name

> Redirect the output of the cat command to a file rather than standard output. The destination file will be created if it doesn't exist and will be overwritten if it does.

>> Append the output of the cat command to the end of an existing file. The destination file will be created if it doesn't exist.

| Send (or pipe) the output of the cat command into another command for further processing.

//creation of a file named as 'myfile'

$ **cat >** myfile.....here **myfile** is a file with 15 bytes.

hi how are you.

ctrl+d

// knowing file content

$ **cat** myfile

hi how are you.

//Describing the structure of a file ls -l myfile

-rwxrwxrwx 1 aliet **engg** 16 Jun 26 16:19 myfile

Data visualization in files:

OD COMMNAD: od is a program/command for displaying ("dumping") data in various human-readable output formats. The name is an acronym for "octal dump" , default it is printed in the octal data format. It can also display output in a variety of other formats, including hexadecimal, decimal, and ASCII. It is useful for visualizing data that is not in a human readable format, like the executable code of a program.

To work on this command, First create one file with numbers before using **od** command.

$ cat
numbers
1
2
3
4
5
6
7
8
9
10

Required options are **-b and -c**

-b -- shows the bytes as characters

-c -- shows the bytes as octal numbers

Example 1:

aliet@aliet-cse:~$ **od -c numbers**

output:

0000000 1 \n 2 \n 3 \n 4 \n 5 \n 6 \n 7 \n 8 \n

0000020 9 \n 1 0 \n

0000025

Here, The left side 7 -digit numbers are the **positions** in the file.

$ **od -cb numbers**

output:

0000000 1 \n 2 \n 3 \n 4 \n 5 \n 6 \n 7 \n 8 \n

061 012 062 012 063 012 064 012 065 012 066 012 067 012 070 012

0000020 9 \n 1 0 \n

071 012 061 060 012

0000025

So, we see that output was produced in octal format. The first column in the output of **od** represents the **positions** (7-bit in length) in a file. That is the ordinal number of the next character shown in octal.

Note :od command prints a **visual representation** of all bytes of a file. Possible octal values are..

Character	**octal value**
Backspace	010
\t	011
\n	012
Blank space	140
a to z	141 to 170
A to Z	101 to 126
1	061
2	062
3	063
:	:
:	:

Example 2: Text file

$ od -c myfile

0000000 h i h o w a r e y o u . \n

0000020

$ od -**cb** myfile

0000000 h i h o w a r e y o u . \n

150 151 040 150 157 167 040 141 162 145 040 171 157 165 056 012

0000020

Note: Display the byte offsets in different formats such as hexadecimal, Octal, & decimal using **-A** option The byte offset can be displayed in any of the following formats :

Hexadecimal (using -x along with -A) . Ex: $ od -Ax -c myfile
Octal (using -o along with -A). Ex: $ od -Ad -c myfile
Decimal (using -d along with -A) Ex: $ od -Ao -c input

Note: when user not to display offset information i.e file positions in a file by using '-**An**' option
$ od -An -c myfile

h i h o w a r e y o u . \n

Note: **od** command with **no option** dumps the file 16-bit , or 2-byte words and makes the magic number visible:
Ex: od /bin/ed ....ed is pure executable file in octal representation
output:

0000000 000410 025000 000462 011444 000000 000000 000000 000001
0000020 170011 016600 000002 005060 177776 010600 162706 000004

0000040 016616 000004 005720 010066 000002 005720 001376 020076
.....

Note: It is possible to copy the data from one file to another with **>** (output redirection operator)

$od –c myfile > myfile2

$cat myfile //contents of myfile
0000000 h i h o w a r e y o u . \n
0000020
$cat myfile2 //contents of myfile2
0000000 h i h o w a r e y o u . \n
0000020

Buffered output from cat: cat command reads the full file in sequence and store it in buffer and display the buffer 1's its done.
with -u option you can avoid it.
Note: cat command maintains internally a **buffer** to store data .

$ cat

$ cat
hai hai hello hello

Note: here **ctrl+d** is used to stop.
Un-Buffered cat: it uses option **–u.**

$ cat **-u**

123123

456456

3. WHATS'S IN A FILE?

Files are always used for storing data and information. Every file has its own format depends on type of a file. The format of a file is determined by the programs only. There is a wide variety of file types as well programs. Here, file types are not determined by the file system, even the Kernel also can't tell the type of a file is used. It doesn't know it. So, here "**file**" command is used to know it.

Syntax: $file

filename Examples:

$ file myfile **output**: myfile: ASCII text

$ file /bin **output**: /bin: directory

$ file /home **output**: /home: directory

$ file Desktop **output**: Desktop: directory

$ file Documents **output**: Documents: directory

$ file / **output**: /: directory

$ file /root **output** : /root: directory

$ file /etc/passwd output: /etc/passwd: ASCII text

$ file /usr/src output: /usr/src: directory

$file /bin /bin/ed /usr/src/cmd/ed.c /usr/man/man1/ed.1 ...multiple inputs are given here

output:

/bin: directory

/bin/ed: .....pure executable /bin directory maintains all executable files (in binary format)

/usr/src/cmd/ed.c: c program text

/usr/amn/man1ed.1: manual text file

4.Directories and File Names

A filename can be any sequence of ASCII characters without special characters like < , > . A file name starts with a period(.) is a hidden file, generally used by a UNIX utility.
Ex: .profile, .bashrc, .mailrc and .cshrc.

Simple rules:

1) Start your names with an alphabetical order.
2) Use dividers to separate parts of the name. (use underscore,period, and the hyphen)
3) Use an extension at the end of the filename, even though UNIX doesn't recognize extensions.
4)Never start a filename with a period. File names that start with a period are known as "hidden files" in UNIX.Generally , hidden files are created and used by the system .
5)All files have unambiguous (different) names ,starting with proper order , for example /usr/mydir/myfile.txt. because more than one user may use the same name to a file. If you want to know the files in your current directory, use **ls commad**.

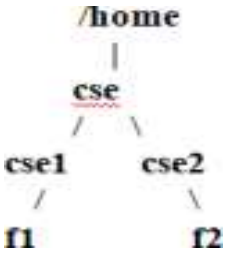
$ls

myfile
$
Each running program/file (i.e a process) has a current directory. All file names are implicitly assumed to start with the name of that directory ,unless they begin directly with a slash.
Here, user login shell & ls have a current directory. To get the current directory, use **pwd** (present/print working directory) command is used. pwd identifies the current directory.
$ pwd

/home/aliet

The current directory is an attribute of a process, not a person or a program. People have login directories, processes have current directories. If a child process creates a child process , the child inherits the current directory of its parent. But if the child then changes to a new directory , the parent is unaffected .its current directory remains the same .

Consider a **simple** directory structure and its **creation**.



$pwd

$ mkdir cse //creation of new directory
$ cd cse //changed to newly created directory.

...../cse$ **mkdir** cse1 cse2 //Creation of two sub/child directories.

.....cse$ **cd** cse1 //changed to newly created directory

$pwd

/home/cse/cse1
...../cse/cse1$ **cat >f1**
hii this is directory structure

ctrl+d

...../cse/cse1$ cat f1
hii this is directory structure
...../cse/cse1$ ls

f1

...../cse/cse1$ cd .. //back to its parent directory
...../cse/$cd cse2
...../cse/cse2/cat
> f2 This file
for CSE2
...../cse/cse2/ls

f2

...../cse/cse2$ cd .. //back to its parent directory
...../cse$
$ ls cse //Listing of two subdirectories in CSE directory.

cse1 cse2

$ ls cse/cse1 //Listing of files in cse/cse1 directory.

f1

DU COMMAND : disk usage command reports usage of disk by a recursive examination of the directory time. i.e consumption of disk space by both directory & files.

$ cd cse

.../cse$ cd cse1

../cse/cse1$ du

usage of **du** with **grep** command:

$du –a | grep myfile
4

grep command: Global Regular Expression Print .

It is used to **finding patterns** in files. This command searches one or more files for text , matches a pattern in the given file. It is also called as **Filter** command.
-used to extract information from files.
-To search the output of a command for lines related to a particular item.
-To locate files containing a particular keyword.

Syntax: grep “pattern” filename
Ex: $ grep "hi" myfile

hi how are you.

$ grep -i "Hi" myfile //ignoring the case

hi how are you.

$ **du -a | grep** myfile // disk usage

4 ./myfile.bak

4 ./myfile

$ **grep** cse **/etc/passwd** // searching for cse in /etc/passwd file.

cse:x:1001:1002::/home/cse:

cse1:x:1002:1003::/home/cse1:

DF COMMAND(DISK FREE) : df command (disk free) tells the amount of free space available on the disk. The 'df' command also stands for "disk filesystem", it is used to get full summary of available and used disk space usage of file system on Linux system.

$df //for entire filesystem

output:

Filesystem	1K-blocks	Used	Available	Use%	Mounted on
/dev/cciss/c0d0p2	78361192	23185840	51130588	32%	/
/dev/cciss/c0d0p5	24797380	22273432	1243972	95%	/home
/dev/cciss/c0d0p3	29753588	25503792	2713984	91%	/data
/dev/cciss/c0d0p1	295561	21531	258770	8%	/boot
tmpfs	257476	0	257476	0%	/dev/shm

$ df -h //human readable format

Filesystem	Size	Used	Avail	Use%	Mounted on
/dev/cciss/c0d0p2	75G	23G	49G	32%	/
/dev/cciss/c0d0p5	24G	22G	1.2G	95%	/home
/dev/cciss/c0d0p3	29G	25G	2.6G	91%	/data
/dev/cciss/c0d0p1	289M	22M	253M	8%	/boot
tmpfs	252M	0	252M	0%	/dev/shm

$ df -t ex3 //for specific file

Filesystem	**1K-blocks**	**Used**	**Available**	**Use%**	**Mounted on**
/dev/cciss/c0d0p2	78361192	23190072	51126356	32%	/
/dev/cciss/c0d0p5	24797380	22273432	1243972	95%	/home
/dev/cciss/c0d0p3	29753588	25503792	2713984	91%	/data
/dev/cciss/c0d0p1	295561	21531	258770	8%	/boot

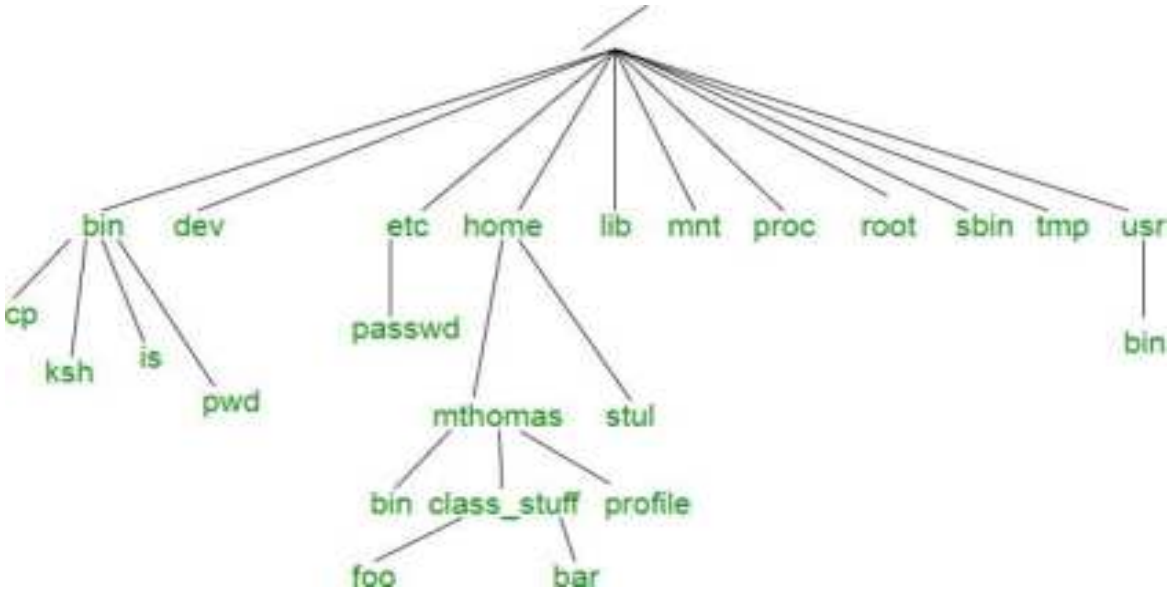
5.UNIX FILE SYSTEM / DIRECTORY STRUCURE:

The Unix file system is essentially composed of files and **directories**. Directories are special files that may contain other files.

Note: Everything in the unix system is called a file.

Unix file system is a methodology (or) technique (or) logical method of **organizing and storing** large amounts of information in a way that makes it easy to manage. A file is a smallest unit in which the information is stored. Unix file system has several important features. All data in Unix is organized into files. All files are organized into directories. These directories are organized into a tree-like structure called the file system.

Files in Unix System are organized into multi-level hierarchy structure known as a directory tree. At the very top of the file system is a directory called "root" which is represented by a "/". All other files are "descendants" of root.



The following system files (i.e. directories) are present in most Unix filesystems:

- ***/bin*** - short for binaries, this is the directory where many commonly used executable commands reside.(such as cat,rm cp etc.)
- ***/dev*** - contains device specific files
- ***/etc*** - contains system configuration files

  -Stores system administrative files and programs.

  /etc/passwd – Stores all user information.
  /etc/shadow – Stores user passwords
  /etc/group – Stores all group information
- ***/home*** - contains user directories and files. It is for all of the system's users.
- ***/lib*** - contains all library files. Essential system library files used by tools in '/bin'
- ***/mnt*** - contains device files related to mounted devices
- ***/proc*** - contains files related to system processes
- ***/root*** - the root users' home directory (note this is different than /)
- ***/sbin*** - system binary files reside here. If there is no sbin directory on your system, these files most likely reside in 'etc'
- ***/tmp*** - storage for temporary files created by programs , which are periodically removed from the filesystem
- ***/usr*** - also contains executable commands. Sub directories with files related to user tools & applications.

  /usr/include – stores standard header files .

  /usr/lib- stores standard library files .

Files are indentified by **path names**

• Files must be created before they can be used.

Path name:

In unix file system structure , each node is either a file or a directory of files, where we can maintain files and directories. It specifies a file or directory by its path name. There are TWO Path names are there.

1) *FULL, or ABSOLUTE path name*
2) *RELATIVE path name*

Full /absolute path name : It starts with the root(/), and follows the branches of the file system, each separated by /, until you reach the desired file, Example:

/home/usr/cse/sec/std.txt

Relative path name : it specifies the path relative to another, usually the current working directory to a specified file ,Example:

cse/sec/std.txt

Two special directories used in file system for moving from one level of directories to another level of directories.

. the current directory
.. the parent of the current directory .

Note: To get the directory hierarchy , use

Note:To get hierarchical structure of file system use

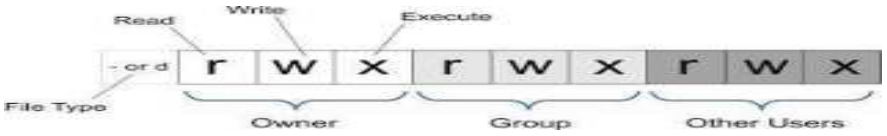
$man hier

6.Permissions:

The UNIX file system is designed to support multiple users.When many users are sharing one system,it is imortant to be able restrict access to certain foiles.The system administarator wants to prevent other users from changing important **system** files.

Permissions for files:

there are three of file permissions for the three classes of users: The owner(or user) of the file , The grouthe file belongs to and all other users of the system.To know the permissions of a file, use ls -l command .



Ex: ls –l /etc/passwd ..... for **aliet** user only -rw-r--r-- 1 **aliet** 5115 jun 25 9:40 /etc/passwd
Ex:ls -lg /etc/passwd ..... for **group admin** user
-rw-r--r-- 1 **adm** 5115 jun 25 9.45 /etc/passwd
Ex:ls -ld.....for **directory** file
drwxrwxr-x 3 cse 80 jun 25 9:45

Permissions for Directories:

Similar to files, Directories also have permissions. 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 .

PERMISSION	FILE	DIRECTORY

READ	User can open and Read the content of a file	User can list files present in directory and cannot read files of directory
WRITE	User can modify contents of file	User can add or delete files to directory
EXECUTE	User can run executable files	User can use **cd** command and can go through the directory

7. I-node:

The **inode** is a data structure in a Unix-style file system that describes a filesystem object such as a file or a directory.Every file have a unique **inode** to maintain information about the file except name & its data. Each inode stores the attributes and disk block location(s) of the object's data. File system object attributes may include metadata (times of last change, access, modification), as well as owner and permission data.

Each file is associated with an *inode*, which is identified by an integer number, often referred to as an *i-number* or *inode number*.

Inodes store information about files and directories (folders), such as file ownership, access mode (read, write, execute permissions), and file type.

The inode number indexes a table of inodes in a known location on the device. From the inode number, the kernel's file system driver can access the inode contents, including the location of the file - thus allowing access to the file.

A file's inode number can be found using the ls -i command. The ls -i command prints the i-node number in the **first column** of the report.

To get I node of a file named f5 is

$ ls – i f5

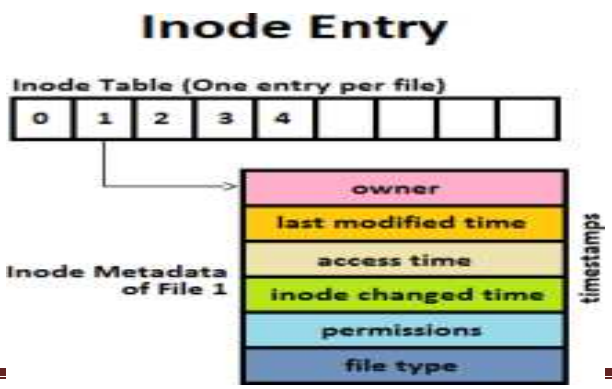
2100210 -rw-rw-r-- 1 aliet aliet 25 Jun 26 17:04 f5

An inode of a file contains....

1) Owner of the file
2) Group to which the owner belongs
3) Type of a file
4) File access permission

5) Date and time of last access
6) Date and time of last modifications
7) No of links to the file
8) Size of the file

9) Address of blocks where the file is physically present.



Ln(link) command: To create a link between two files. This command is used to creates a new name for an existing file.

Syntax: ln [options] file1 file2

Here , changes to one file are also reflected in the other file.

Types of link methods: 1)Hard link
2)Soft link / Symbolic link

Hard link: It is default link to any file. It is a UNIX path name for a file

Syntax: $ **ln** file1 file2 //here file2 should not exist in filesysytem.i.e new file.

Ex: $**ln** f2 f5

To know the **status of a link** (along with a **long list** of information) whether it is created or not. Use ls -li

$ ls -**li** f2 f5

2100210 -rw-rw-r-- 1 aliet aliet 25 Jun 26 17:04 f2

2100210 lrwxrwxrwx 2 aliet aliet 2 Jun 26 17:09 f5

When we work on **ln** command, two linked files must have the same inode number. Otherwise ,those two files are not have a link to each other.

Soft link or symbolic link:

To create symbolic link, ln command is used with option –s

$ ln -s f2 f7

$ ls -li f2 f7

2100210 -rw-rw-r-- 2 aliet aliet 25 Jun 26 17:04 f2

2100214 lrwxrwxrwx 1 aliet aliet 2 Jun 26 17:09 **f7-> f2**

Unlink : it removes or eliminates a link in between files(i.e existing link)

Syntax: unlink file1 file2

Ex: unlink f2 f7

Difference : hard link and symbolic link

Hard link	**Symbolic link**
Does not create a new inode	Create a new inode
Cannot link directories unless it is done by root	Can link directories
Cannot link files across file systems	Can link files across file system
Increase hard link count of the linked inode	Does not change hard link count of the linked inode

Note: ln command works like **cp** command, but there is a difference in that.

Difference : cp and ln command

Cp command creates a duplicated copy of file to another file with a different path name.

Where as, **ln** command saves space by not duplicating the copy here the new file will have same inode number as original file.

8.File Attributes and Permissions:

File Attributes:

Files are the **building blocks** of any operating system. Every file has some properties . Properties are nothing but attributes of a file.

The UNIX File Attributes are

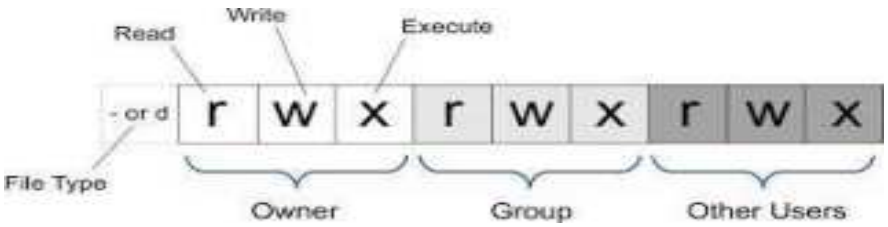
1) File type - specifies what type of file it is.
2) Access permission - the file access permission for owner, group and others.
3) Hard link count - number of hard link of the file
4) UID - The file owner/ User ID.
5) GID -The file Group ID .
6) File size - The file size in bytes.
7) I-node no - The system I-node (Information Node) number of the file.
8) File system id - The file system id where the file is stored.
9) Last access time - the time, the file was last accessed.
10) Last modified time - the file, the file was last modified.
11) Last change time - the time, the file was last changed.

Permissions:

The UNIX file system is designed to support multiple users. When many users are sharing one system, it is important to be able restrict access to certain foiles. The system administrator wants to prevent other users from changing important **system** files.

Permissions for files:

there are three 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.To know the permissions of a file, use ls -l command .



Ex: ls -l /etc/passwd.....for **aliet** user only
-rw-r--r-- 1 **aliet** 5115 jun 25 9:40 /etc/passwd

Ex:ls -lg /etc/passwd ..... for **group admin** user
-rw-r--r-- 1 **adm** 5115 jun 25 9.45 /etc/passwd

Ex:ls -ld.....for **directory** file
drwxrwxr-x 3 cse 80 jun 25 9:45

Permissions for Directories:

Similar to files, Directories also have permissions. 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 .

PERMISSION	FILE	DIRECTORY
READ	User can open and Read the content of a file	User can list files present in directory and cannot read files of directory
WRITE	User can modify contents of file	User can add or delete files to directory
EXECUTE	User can run executable files	User can use **cd** command and can go through the directory

2.The File Command knowing the File Type:

In Unix, there is a wide variety of file types. Here, file types are not determined by the file system, even the Kernel also can't tell the type of a file is used. It doesn't know it. So, here "**file**" command is used to know it.

Syntax: $file filename

Examples:

$ file myfile **output**: myfile: ASCII text

$ file /bin **output**: /bin: directory

$ file /home **output**: /home: directory

$ file Desktop **output**: Desktop: directory

$ file Documents **output**: Documents: directory

$ file / **output**: /: directory

$ file /root **output** : /root: directory

$ file /etc/passwd **output**: /etc/passwd: ASCII text

$ file /usr/src **output**: /usr/src: directory

$file /bin /bin/ed /usr/src/cmd/ed.c /usr/man/man1/ed.1 ...multiple inputs here

output:

/bin: directory
/bin/ed: .....pure executable /bin directory maintains all executable files (in binary format)
/usr/src/cmd/ed.c: c program text
/usr/amn/man1ed.1: manual text file

type command **:**

In Linux/unix system, **type** command is used for displaying information about command type. It displays if command is an alias,shell function, shell builtin, disk file, or shell reserved word.

You can use type command with other command names also.

Note: The type command will show output only when the command/executable name is available or installed in Linux System.

type [options] name [name]

Description:

-t option :

It will show a single word or string.It tells a command or name is an alias, shell reserved word, function, builtin, or disk file(external command).

$ **type -t ls** **output:** alias
$ **type -t date** **output:** file
$ **type -t declare** **output:**builtin

-p option :

The command of the disk file that would be executed. In other words, absolute path of command.

$ **type -p date** **output:** /bin/date

-P option :

Force a PATH search for each command/NAME, even if it is an alias,builtin, or function, and returns the name of the disk file that would be executed.
(We can get PATH environment value by using command echo $PATH)

$ **type -P date** **output:**/bin/date

NOTE: In below example, date command was searched in all directories listed in PATH(Environment set). And found the date command in /bin

-a option :

It display all locations have command or NAME .
It includes aliases, builtins, and functions (only in case , if -p option is not used)

$ **type -a date** **output:**date is /bin/date
$ **type -a echo** **output:** echo is a shell builtin

f option :

The -f option, suppress shell function lookup.

$ **type -f echo** **output:** echo is a shell builtin
$ **type -f date** **output:** date is /bin/date

3. The Chmod Command for Changing File Permissions:

Chmod command: **CHANGING MODE OF A FILE**

Chmod is used to change permissions on files and directories. The command chmod may be used with either letters(also known as symbolic) or numbers (also known as octal) to set the permissions. The letters used with chmod are in the table below:

Letter	permission/usage
r	Read
w	Write
x	Execute
X	Execute (only if a directory)
u	User or Owner of the file
g	Group User
o	Other user
a	All users
+	Grant permissions
-	Revoke permissions
=	Set permissions

Read, Write & Execute Permissions:

Permissions are the “rights” to act on a file or directory. The basic rights are read, write, and execute.

- **Read** - a readable permission allows the contents of the file to be viewed. A read permission on a directory allows you to list the contents of a directory.
- **Write** - a write permission on a file allows you to modify the contents of that file. For a directory, the write permission allows you to edit the contents of a directory (e.g. add/delete files).
- **Execute** - for a file, the executable permission allows you to run the file and execute a program or script. For a directory, the execute permission allows you to change to a different directory and make it your current working directory. Users usually have a default group, but they may belong to several additional groups.

Viewing File Permissions :

To view the permissions on a file or directory, issue the command ls -l <directory/file>. Remember to replace the information in the < > with the actual file or directory name. Below is sample output for the ls command:

-rw-r--r-- 1 root root 1031 Nov 18 09:22 /etc/passwd

The **first ten** characters show the access permissions. The first dash (-) indicates the type of file (d for directory, s for special file, and - for a regular file). The next three characters (**rw-**) define the owner's permission to the file. In this example, the file owner has read and write permissions only. The next three characters (**r--**) are the permissions for the members of the same group as the file owner (which in this example is read only). The last three characters (**r--**) show the permissions for all other users and in this example it is read only.

It is important to remember that the first character of the first column of a file listing denotes whether it is a directory or a file. The other nine characters are the permissions for the file/directory.

The example **drwxrw-r–** is broken down as follows:

d is a directory
rwx the user has read, write, and execute
permissions rw- the group has read and write
permissions
r– all others have read only permissions
Note that the dash (-) denotes permissions are removed. Therefore, with the "all others" group, r– translates to read permission only, the write and execute permissions were removed.

Conversely, the plus sign (+) is equivalent to granting permissions: chmod u+r , g+x <filename>
In other words, the user was given read permission and the group was given execute permission for the file. Note, when setting multiple permissions for a set, a comma is required between sets.

Chmod Octal Format:

To use the octal format, you have to calculate the permissions for each portion of the file or directory. The first ten characters mentioned above will correspond to a four digit numbers in octal. The execute permission is equal to the number one (1), the write permission is equal to the number two (2), and the read permission is equal to the number four (4). Therefore, when you use the octal format, you will need to calculate a number between 0 and 7 for each portion of the permission. A table has been provided below for clarification.

syntax: chmod <permissions> <file/directory name>

Example: chmod 777 myfile----- all permissions to all users

```
$ ls –l myfile      (or)
-rwxrwxrwx 1 aliet aliet 40 2018-06-18 14:09 myfile

$ chmod a+rwx myfile
$ ls –l myfile      (or)
-rwxrwxrwx 1 aliet aliet 40 2018-06-18 14:09 myfile

$ chmod   ugo+rwx  myfile        (or ugo=r+w+x or  ugo=rwx)

$ ls –l myfile

-rwxrwxrwx 1 aliet aliet 40 2018-06-18 14:09 myfile
```

chmod 541 myfile -----read& execute for user, read only for group , execute for others.

$ ls –l myfile
-r-xr----x 1 aliet aliet 40 2018-06-18 14:09 myfile (or)

chmod u+rx g+r u+x myfile ---- read& execute for user, read only for group , execute for others.

$ ls –l myfile
-r-xr----x 1 aliet aliet 40 2018-06-18 14:09 myfile (or)

chmod u-w g-wx u-rw myfile------read& execute for user, read only for group , execute for others.

$ ls –l myfile
-r-xr----x 1 aliet aliet 40 2018-06-18 14:09 myfile

An octal table showing the numeric equivalent for permissions is provided below.

Permissions	Binary	Octal	Description
---	000	0	No permissions
--x	001	1	Execute-only permission
-w-	010	2	Write-only permission
-wx	011	3	Write and execute permissions
r--	100	4	Read-only permission
r-x	101	5	Read and execute permissions
rw-	110	6	Read and write permissions
rwx	111	7	Read, write, and execute permissions

Permission string	Octal code	Meaning
rwxrwxrwx	777	Read, write, and execute permissions for all users.
rwxr-xr-x	755	Read and execute permission for all users. The file's owner also has write permission.
rwxr-x---	750	Read and execute permission for the owner and group. The file's owner also has write permission. Users who aren't the file's owner or members of the group have no access to the file.
rwx------	700	Read, write, and execute permissions for the file's owner only; all others have no access.
rw-rw-rw-	666	Read and write permissions for all users. No execute permissions for anybody.
rw-rw-r--	664	Read and write permissions for the owner and group. Read-only permission for all others.
rw-rw----	660	Read and write permissions for the owner and group. No world permissions.
rw-r--r--	644	Read and write permissions for the owner. Read-only permission for all others.
rw-r-----	640	Read and write permissions for the owner, and read-only permission for the group. No permission for others.
rw-------	600	Read and write permissions for the owner. No permission for anybody else.
r--------	400	Read permission for the owner. No permission for anybody else.

Example: To enable execution permission for the file "myfile" by all is

Chmod ugo+x myfile
(or) Chmod +x myfile (or) Chmod a+x myfile

Important commands:

1)dir Command:

It works like Unix/Linux **ls** command, it lists the contents of a directory.

$ dir

508
cse

2) groups command :

groups command displays all the names of groups a user is a part of like this.

$ groups

aliet adm cdrom sudo dip plugdev lpadmin sambashare

3) id command : shows user and group information for the current user or specified username .

$
id

uid=1000(aliet) gid=1000(aliet)
groups=1000(aliet),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),108(lpadmin),124(sambashare)

4) grep:

It is a search tool on Unix-like systems which can be used to search for anything whether it be a file, or a line or multiple lines in file is grep utility. It is very vast in functionality which can be attributed to the large number of options it supports like: searching using string pattern, or regular expression pattern .
Syntax: grep "pattern" filename
$ grep "hi"

myfile hi how

are you.

$ grep -i "Hi"

myfile hi how

are you.

$ sudo du -a | grep

myfile [sudo]

password for aliet:

4 ./myfile.bak

4 ./myfile

$ grep cse /etc/passwd //to search for cse in /etc/passwd

cse:x:1001:1002::/home/cse:

cse1:x:1002:1003::/home/cse1:

ls -l /etc/passwd

-rw-r--r-- 1 root root 2392 Jun 26 15:56 /etc/passwd

4. The Chown Command Changing the Owner of a File i.e CHANGE OF OWNERSHIP OF A FILE.

chown command : changes/updates the user and group ownership of a file/directory

Before that ,we need to know "how to add new users in to a system?"

1) Adding new users

syntax: $ **sudo useradd** username

Example:

$ **sudo useradd user1**..... add a new user

[sudo] password for aliet:..... enter system admin password

aliet@aliet-desktop:~$ **sudo passwd user1**

.....setting a password , here username and password are the same

Enter new UNIX password:

****

Retype new UNIX password:

****

passwd: password updated successfully

After adding of new user, you need to change that account by using **su** command.

su : Switch to user

$ **su user1**

.....switch to user1 using su command

Password:

****

user1@aliet-desktop:/home/aliet$ //now the user in user1 mode

Now , again go back to aliet user account, where you can apply **chown** command

Syntax: chown newusername filename

Ex:$ **sudo chown user1 modefile**

[sudo] password for aliet:

****

Now, the file has been changed from aliet to user1 account. Now , again login into user1 account to change users permissions. by using chmod command.

$ su user1

Password:

****

user1@aliet-desktop:/home/aliet$ **cat modefile** **//now the user in user1 node.**

hi how are you...

user1@aliet-desktop:/home/aliet$ **chmod 711 modefile**

user1@aliet-desktop:/home/aliet$ **ls -l modefile**

-rwx--x--x 1 user1 aliet 18 Jun 21 11:16 modefile

user1@aliet-desktop:/home/aliet$ **cat >> modefile**

new text is added here

user1@aliet-desktop:/home/aliet$ **cat modefile**

hi how are you...

new text is added here

5. The Chgrp Command Changing the Group of a File:

chgrp command: Like chown, this command changes the group ownership of a file.
It can be used by the owner of the file, and the user belongs to the same group can only change the group ownership of a file to another group.

This command provide the **new group name** as its **first** argument and the **name of file** as the **second** argument.

Groups command : groups command displays all the names of groups a user is a part of like this.

The **adduser** and **addgroup** commands are used to add a user and group to the system respectively according to the default configuration specified in **/etc/adduser.conf** file.

Syntax: groupadd groupname

$ **sudo** groupadd engg //creation of group named as "engg"

[sudo] password for aliet:

$ sudo useradd -m cse1
$ sudo passwd cse1

Enter new UNIX password:
****
Retype new UNIX password:
****
passwd: password updated successfully

$ sudo **userdmod** -a -G engg cse1 // add a user to agroup
$ cat >myfile

hi this is for change group user

After creation of new group, you need to create a file and chanage the permissions.

$cat myfile
hi this is for change group user

$ ls -l myfile

-rw-rw-r-- 1 aliet aliet 34 Jun 26 16:05 myfile

$ chmod 777 myfile

$ ls -l myfile
-rwxrwxrwx 1 aliet aliet 34 Jun 26 16:05 myfile

Now, user need to use **chgrp** command to the change the group owner of the file.
Syntax: chgrp newgroupname
filename Ex: $ sudo **chgrp** engg
myfile

$ ls -l myfile

-rwxrwxrwx 1 aliet engg 34 Jun 26 16:05 myfile

$ cat

>myfile hi

how are

you.

$cat myfile

hi how are you.

$ ls -l myfile

-rwxrwxrwx 1 aliet **engg** 16 Jun 26 16:19 myfile

****

6. Important & frequently asked questions

1) What is a file system? Explain Unix file system with neat diagram

2) What is a file ? Explain its attributes and permissions with examples

3) Write short notes on Inodes.

4) Explain briefly about file permissions with examples

5) Mention and Explain the commands used for changing the owner of a file as well as the group of a file.

6)Illustrate file directories with neat diagram and explain **Directory handling commands** supported by Unix. (pwd, mkdir, cd, rmdir)

7) Explain the basics of file, directories and file names ?

8)Give a short note on file and mention various types of file and explain **File handling commands** (cat, rm, cp, mv, wc)

9)Briefly explain how to change file permissions, Changing the owner of a file ,Changing the group of a file.

10)Briefly explain about **ls** command with all options and examples.

11)Explain about the file command-knowing the file type.

UNIT-III

Using the Shell-Command Line Structure-Met characters-Creating New Commands-Command Arguments and Parameters-Program Output as Arguments-Shell Variables--More on I/O Redirection-Looping in Shell Programs

Shell:

- Shell is a program that sits on the kernel and acts as an agent or interface between the users and the kernel.
- A shell is a command interpretor or a processor. As soon as the system is booted successfully, the shell presents a command line prompt(usually $ or a% symbol) At which the user can type in any unix command.

Utility:

- it a program. who, date and ls are the examples of the utility program.

Command:

- A command is all of the text type at the command line, the utility and all of the flags, filenames and other text.

Command line structure:

- A command is a program that tells the unix system to perform some action. The following structure of the command is to
- Command name [options] [arguments]
- Argument, indicates on what the command is to perform its action, usually a file or series of files.
- Option, an option modifies the command, changing the way it performs.
- We can execute a command file by simply specifying its filename. Eg: who(gives the details of users those are currently logged in)

  Date(prints today's date).
- Commands are case sensitive(i.e) LS is different from ls
- Options are generally preceded by hyphen(-) and for most commands, more than one option can be used as follows.

  Syntax:

  Command -[option][option][option]

We can also use semicolon as command terminator .
Eg: date;who

The above commands prints today's date, followed by details of users those are logged currently. The commands are separated by semicolon are executed individually from left to right.

- For most commands you can separate the options, preceding each with a hyphen.

- Syntax: command –option1 –option2 –option3
  Eg: ls –a –l -R
- Some commands have options that require parameters. options requiring parameters are usually specified separately.
  Eg: lpr –P printer 3 -#2file
- The above command sends 2 copies of file to printer 3. We can also group more than one command by using paranthesis.
  Eg: (date;who)
- We can use pipe symbol to give output to one command as input to another command.
  Eg: date;who|we
- It prints date along with the number of lines, words and characters present in the output of who command.
- We can also use'&' ampersand as command terminator. # this command prints todays date two times with time gap of 5 seconds.
  Eg: (sleep5;date&date)

METACHARCTERS:

Meta characters are the special characters, used in unix, these characters are interpreted by shell, without passing them to the commands.

Symbol	
>	output redirection
>>	output redirection(append)
<	Input redirection
<<	Input redirection(append)
*	file name wildcard. Multiple char
[ ]	used to mismatch specific char
?	used to mismatch single character
;	multiple commands
Cmd	command substitution
$(cmd)	command substitution
#	comments
&	background process
( )	group command
$	expands variables
/	escapes interpretation of followed character
|	pipeline
-	Specifies range of characters.

Although , there are number of meta characters some of them are mostly used. They Are “*”, “?”, “[]” and “-“.

Asterisk(*):

The character is used to match multiple characters that is any and all characters.

1) Eg: ls c*.

The above command displays all the files and subdirectories whose name starting with ‘c’.

2) Eg: ls x*y

The above command displays all the files starting with ‘x’and ending with ‘y’

3) Eg: $ls q?

The above command displays all the file names with two characters length but starting with character ‘a’.

SquareBrackets[]:

Brackets are used to match set of specified characters. Specific characters are separated by comma(,).

Eg: ls[a,b,c]*

Question mark(?):

The(?) question mark meta characters is used to match a single character in a file name.

Hyphen(-): Using(-) hyphen metacharcter within[] brackets is used to match a specified range of characters.

Eg: ls [a-x]*

The above command will display the file name starting with lower case letters.

Dollar($): We use '$' symbol ti retrieve the values of the variables.

Eg: x=10

In the above statement we have assigned a value to the variable x.

Echo $x

The above command retrives the value of 'x' and displays the value as10.

Creating new commands: Given a sequence of commands that is to be repeated more than a few times, it would be convenient to make it into a new command, with its own name, so you can use it like a regular command.

1 .For creating new commands ,the first step is to create new file with actual command.

For example, here I'm creating a new command 'nu' which counts number of users currently logged in that we have to create a new file with name 'nu'.

Cat>nu (or) echo who|wc -l>nu

Who|wc -l

2.'sh' is a command like other commands who, date, char etc. By using sh we can execute our commands who, date, char etc. By using sh we can execute our commands 'nu' like

i.sh<nu (or) sh nu

3. But if we want to execute 'nu' without 'sh' we have to assign execution permission to all types of users by the fallowing way.

Chmod +x nu

4. From the above step we can execute 'nu' directly .

5. But 'nu' will be executed in the directory in which is created .If we wanr to execute 'nu' from any other directory.

We have to move the file 'nu' to the directory bin.

Command arguments and parameters:-

While executing new commands ,we can pass arguments or parameters to our new commands.

1.To pass arguments or parameters ,we have to specify the parameters number preceding with $ symbol, in the command file itself.

For example let us create a new command called Ww, which returns the number of words in the specified file .

1.cat>Ww

Wc –w $

1

Creating a command file Ww

2. Making Ww

executable Chmod +x

Ww

3. Executing Ww using

sh Sh>Ww examples

The above two commands will display number of words in the file example 1.

Why because , in the command file Ww, $1 will be replaced by example1

4. Executing Ww without using sh. Move Ww to

\bin Mv Ww \bin.

5. From above step on wards , we can executed Ww, without usin sh.

Ww example1 (or) who example2

Shell variables: A part from being an interface between users and kernel and the command processor ,the shell has programming capabilities of its own .

A shell can be considered as a programming language variables are defined and used with a shell.

Rules for constructing variables are same as file name [variable names include alphanumeric characters and underscore(_)].

The variable names are case sensitive .

There are three types of variables

1. System variables (or) global variables
2. Local variables(or) user defined variables
3. Read –only variables

System variables:-

System variables are set either during the boot sequence or immediately after logging in.

The working environment under which a user works, depends entirely upo the values of these variables. These variables are also known as "Environment variables".

These are similar to global variables in general sense.

By convention these variables are written using uppercase letters only.

Eg: PATH, HOME, MAIL, SHELL, TERM, IFS etc;

i. Path variable:-

This path variable holds a list of directories in a certain order. In this list(:) separate different directories.

The value of PATH can be printed using who command.

Eg:- $echo $PATH

/user/local/sbin:/usr/sbin :/usr/bin/xll:

$

When any command is given ,the shell searches for its program in the directories listed in the PATH one by one.

ii.The HOME variable:-

When a user gets logged it he/she will be automatically placed in the home directory.

This directory is decided by the system administer at the time of opening an account for a user and is stored in the file etc/password.

The value of the path of the home directory is stored in the variable HOME.

The value of HOME variable can be known by

```
$echo $HOME
/usr/mgv
$
```

iii. The IFS variable :-(Internal field separator)

This variable holds tokens used by the shell commands to parse a string into substrings such as a word or record in to its individual fields.

By default space( ),new line (\n), new tab space(\t) system uses them as default field specifiers.

iv.The MAIL variable:-

Holds the absolute path name of the file where the users mail is kept. Mails we have received are specified by using this variable. We can set the path /usr/spool/user.

v.The Shell variable:-

it contains the name of the users shell program in the form of absolute pathname. The value of shell can be known as

```
$echo $SHELL
/Bin /Bash
$
```

Vi. **The TERM variable:-**

This variable holds the information regarding the type of terminal being used.

2. Local variables:

These variables are defined and used by specific users, they are also called user defined variables..it exist only for a short time during the execution of shell script.

Rules for constructing variable names:

- These variables are also constructed by alphanumeric characters and underscore(-).
- These are case sensitive.
  Eg: sum, SUM are different.

3. Read only variable:

The value of the read only variables cannot be manipulated. A variable can be made read only by using read only function. The variables can behave as constants in c language.

Eg: 1 echo “enter value of x”

2 read x

3 echo “value of x : $x”

4 read only x

5 x = ‘expr4x + 1’

6 echo “After increment x value : 4x”

An error will be displayed like, line5: x is read only variable.

Defining a shell variable:

A shell variable is defined using an equal to (=) operator without any space on either side of it.

Syntax: variable = value

Eg: $x = 37

The default value of a shell variable will be a null string.

Types of shell variable:

- A shell variables are of string type and the values of the variables are stored in ASCII format.
- By default shell variables are initialized as null strings.
- While writing shell programs it is not necessary to type declare or initialize shell variables.

```
$x = 37
$echo $x
 37
$echo   $xyz   # since xyz is not initialized
$                #null string will be output.
```

- Variables are concatenated by placing them adjacent to each other as,

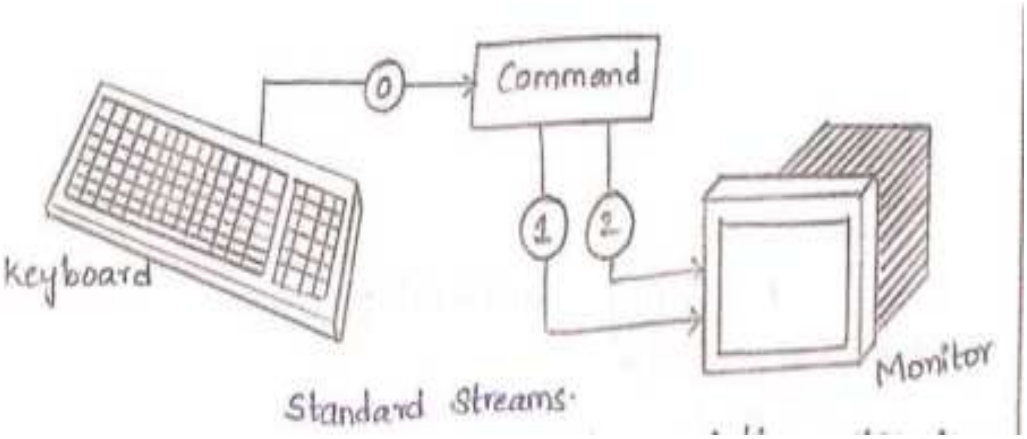
```
$x = Hello ; y = Vignan
$z =$x $y
$echo $z
Hello
vignan
$
```

- Sometimes shell variables are useful in spending up the interaction of the user with the system.
- A shell variable can also be used to replace a command

```
$count = 'wc file 1 file 2'
$$count
```

More on I/O redirection:

- Unix defines three standard streams that are used by commands.
- Each command takes its input from a stream known as standard input.
- Commands that create a output send it to a stream known as standard output.
- If an executing command encounters an error, the error message is sent to standard error.
- To reference each stream, unix assign a descriptor to each stream.
- The descriptors of each stream are,
- Standard input –o – associated with key board.
- Standard output –1– associated with Monitor.
- Standard error –2– associated with Monitor.



We can change the default file association using pipes or redirection.

Redirection:

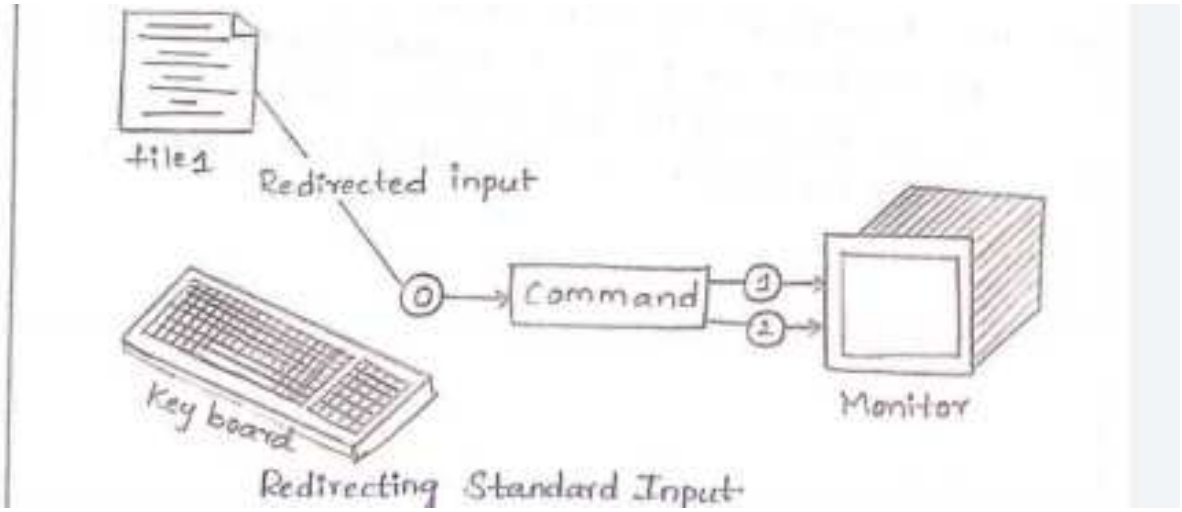
- All the streams (input, output, error) are pre assigned to the key board and the monitor. We can change the default assignments temporarily using redirection.
- Redirection is the process by which we specify that a file is to be used in place of one of the standard files, with input files, we call of input redirection with output files, we call it output redirection and with the error file, we call it error redirection.

Redirecting input:

- We can redirect the standard input from the keyboards to any text file.
- The input redirection operator is less than character(<).

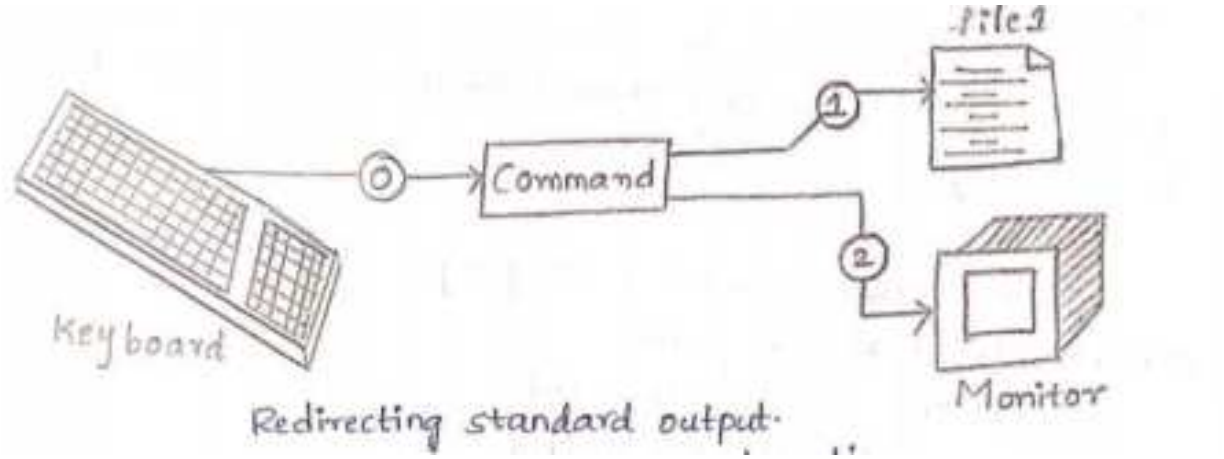
Syntax:

1. Command o< file (or)
2. Command < file 1



Eg: $wc < sample
3 20 103
$

Redirecting output:



The output redirection the commands output is copied to a file rather than displayed on the monitor.

Syntax: Command |> file 1 (or) command > files 1

Command |>| files 1(or) command >| files 1

Command |>> file 1(or) command > files 1

They are 2 ways to output the redirection

1. >output redirection
2. >>output redirection with appending.

Eg: i) $wc sample > new sample

In the above example the output has been redirected to the file called new sample.

ii) cat ex

hello IT

cat ex >>

ex1 # Hello

IT

Cat ex1

Hello

IT

Error redirection:

One of the difficulties with the standard error stream is that it is by default combined with the standard output stream on the monitor.

Eg: ls -l file1 file 2

File2: Not

available

_r_rw_rwx xx xxx 12 june 20 file

1 ls –l file 1 file2 2>errfile

_r_rw_rwx xx xxx 12 june 20 file

1 Cat errfile

Files: Not available

- In the above example there is no file named as file 2 with in the system so an error message is displayed along with the output of files1.
- To avoid ,this we can store errors or messaged exclusively in a file by redirecting them with the use of file descriptor 2 explicitly.

Looping in shell programs: Like other computer languages shell also has loop control structures they are

1) The while command
2) The until command
3) The for command

1) The while command:

This is one of the very widely used loop –control structures. The general syntax for while command is

While [condition]

Do

Set of statements that are to be executed repeatedly

Done

- The while, do and done are keywords.
- The set of commands between do and done keywords are repeatedly executed as long as the condition remains true.
- This is an entry- controlled loop structure(i.e) in other words to enter in to the execution loop the condition must be true.

Eg: 1) write a program to print the numbers from 1 to10 using while loop

```
i=1
While[ $i –le
10] Do
   Echo “value of I : %i”
         I = ‘expr  $i+1’
 Done
```

2) Write a program to print odd numbers from 1 to 10 using while loop i = 1

```
While( $i –le
10) Do
  Echo “value of i:
    &I” I= ‘expr $i+2’
Done
```

3) write a program to print squares of 1 to 5 numbers using while loop i=1

```
while[ $I –le
5] do
 s= ‘expr $i \* $i’
 echo = “square of I “4s
   i= ‘expr $i+1’
 done
```

The until command:

- Like the while command this command is also loop control command.
- But this command behaves exactly in an opposite manner to the while command.
- The until command repeatedly executes the set of commands that appears between the do and done as long as the condition remains false.

Syntax:

```
Until
 [condition] Do
      Set of commands to be executed repeatedly
Done
```

Eg: 1) Write a program to print the numbers from 1 to 10 using until loop

```
  I=1
until[$i –gt 10]
do
   echo “value of I :
         $i” i=’expr
         $i+1”
done
```

2) Write a program to print odd numbers from 1 to 10 using until loop

```
I=1
until[$i –gt 10]
do
   echo “value of I :
         $i” i=’expr
         $i+2”
done
```

3) Write a program to print squares of 1 to 5 using until loop

```
      I=1
Until [$i –gt 10]
do
    s = ‘expr $i\* $i’
   echo “square of I
         “:&s i=’expr
         $i+1’
done
```

The for command: This command works with a set of value generally given in the form of a list.

Syntax: for variable in list

Do

Set of commands that are executed repeatedly

Done

Eg: 1) Write a program to print numbers from 1 to 10 using for loop

```
For I in 1 2 3 4 5 6 7 8 9 10
Do
    Echo " the value of i: "$i
Done
```

2) Write a program to print odd numbers from 1 to 5 using for loop

```
For I in 1 2 3 4 5
Do
    Echo " the value of i: "$i
Done
```

3) Write a program to print squares from 1 to 5 using for loop

```
For I in 1 2 3 4 5 6 7 8 9 10
Do
    Echo 'expr $i
\*$i' Done
```

The continue and break statements:

- One can come out of the loop permanently by using a break statement (for, while,until).
- Similarly ,a continue statement is used to resume the next iteration of loop without considering the statements that appear after the continue statement within the loop.
- Eg: $ cat menu2.sh

```
ans=y
while["$ans "="y"]
do
   echo "MENU \n
1) List of files\n
2) Today's date \n
3) Process status\n
4) Users of the system\n
5) Present working directory\n
6) Quit to unix\n
   Enter your option:\c" Read choice

Case "$choice" in

        1) ls –l;;

        2) date;;

        3) ps;;

        4) who;;

        5) pwd;;

        6) break;;

        7) echo "Invalid choice"

            Continue;;
```

UNIT-IV

Filters-The Grep Family-Other Filters-The Stream Editor Sed-The AWK Pattern Scanning and processing Language-Good Files and Good Filters

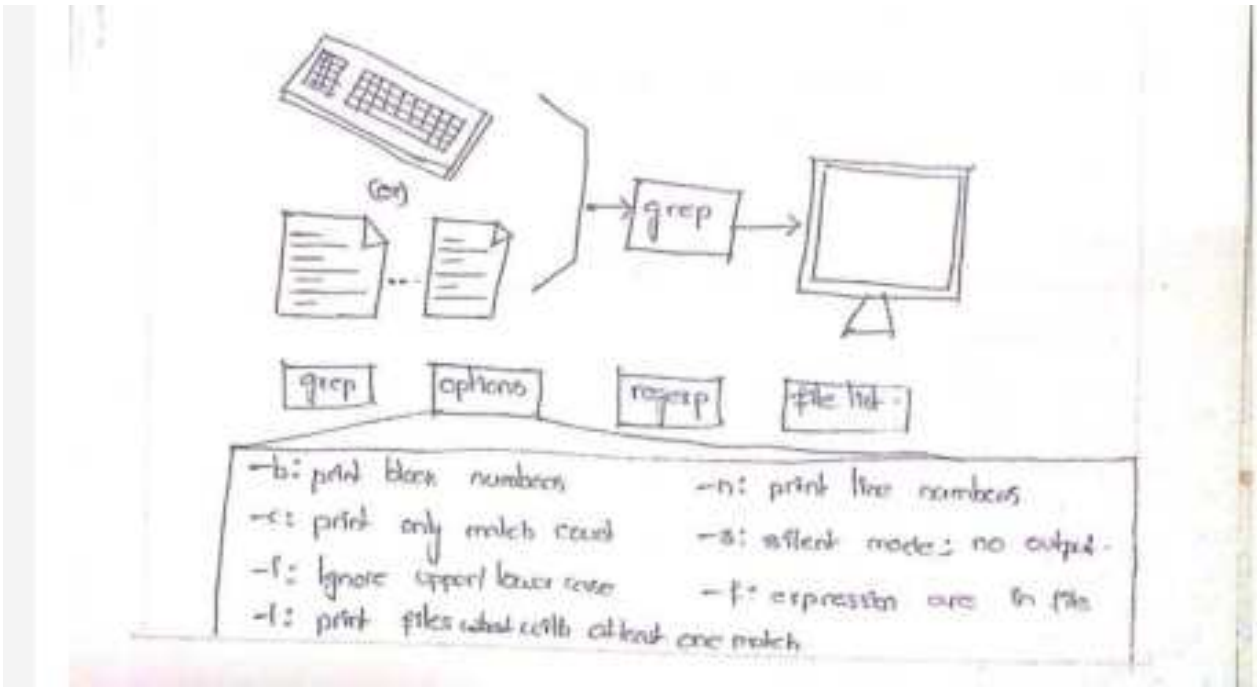
Filter:

A program or a command that read its input from the standard input, processes it in same way, and writes its output to the standard output is called a filter.

- Eg: cat, grep, tee, sort, more, head, tail, cut and paste etc.

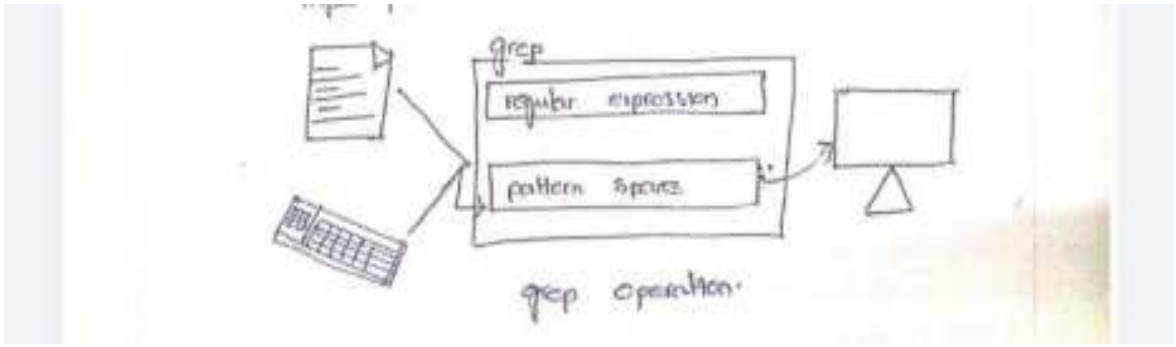
Grep family:

- Grep stands for global regular expression print.
- It is a family of program that is used to search the input file for all lines that match a specified regular expression and write them to the standard output file.



Grep operation:

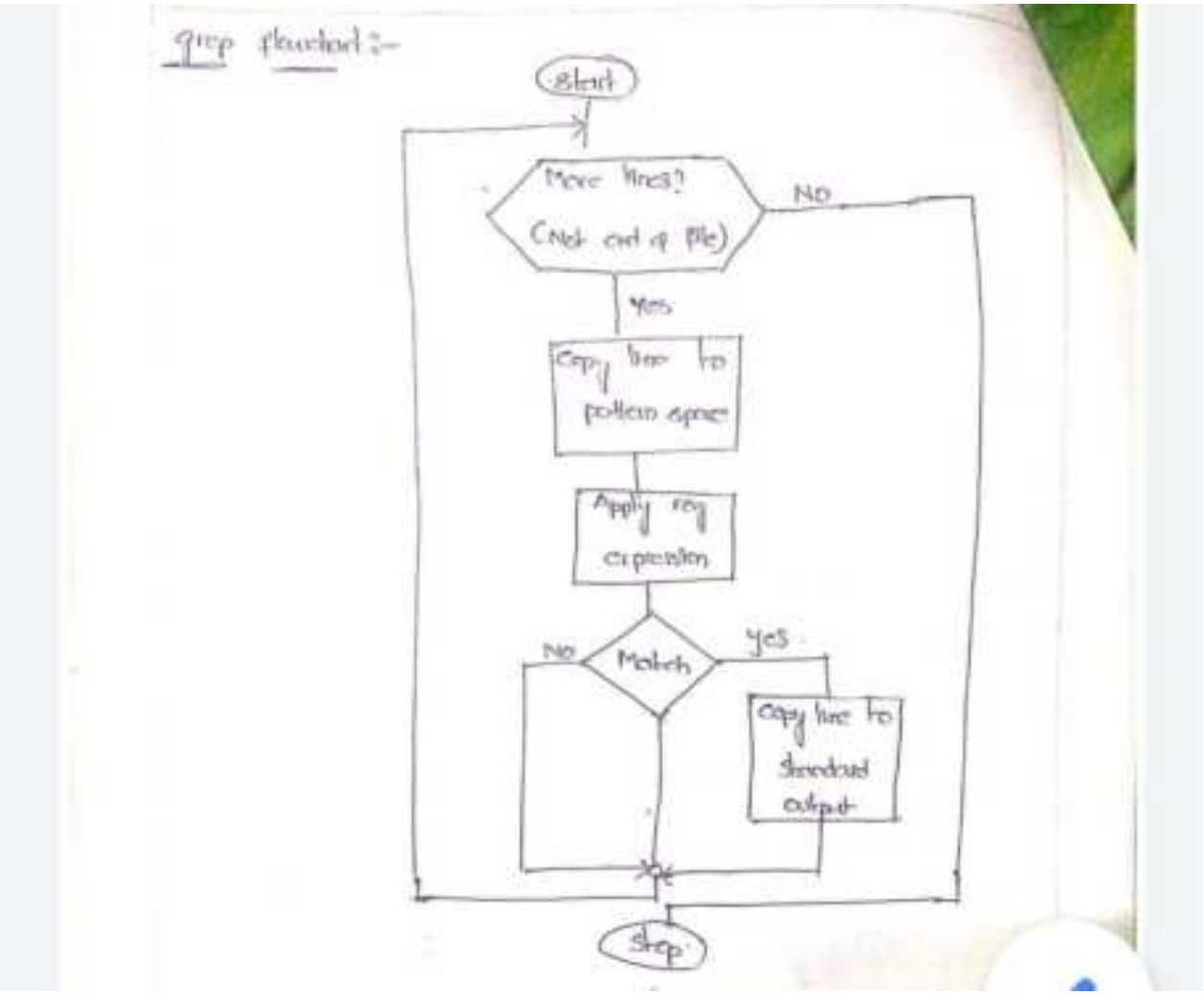
- For each line in the standard input, grep performs in the following operations.
- Copies the next input line into the pattern space. The pattern space is a buffer that can hold only one text line.
- Applies the regular expression to the pattern space.
- There is a match, the line is copied from the pattern space to the standard output.
- The grep utilizes repeat these three operations on each line in the input.



Limitations of grep:

- Grep cannot be used to add, delete or change a line.
- Grep cannot be used to print only part of a line.
- Grep cannot be used to read only part of a file.
- Grep cannot be used to select a line based on the contents of the persons or next line.

Grep Flow chart:



Grep family: They are 3 utilites in the grep family. They are

- **Grep**
- **Egrep(extended grep)**
- **Fgrep(fast grep)**

Grep family options:

Option	Explanation
-b	preceds each line by the file block number in which it is found.
-c	prints only a count of the number of lines modeling the pattern.
-I	ignores upper-lower case in matching text.
-l	prints a list of files that contain of at least one line modeling
-n	shows the line number of each line before the line.
-s	silent mode, executes utility but suppresses all output.
-v	inverse input prints lines that do not match pattern.
-x	prints only lines that entirely match pattern.
-f	list cf strings to be matched are in file.

Grep:

- The original of file marketing utilites, grep handles most of the regular expressions.
- It is generally slowe than egrep

Syntax: $grep [options] pattern [filename1][filename1]

Eg: $grep iyer student.lst

Cs018 |karthik iyer| v cspi | M| 02|05|84

In the above example, the line or record containing the name iyer is extracted using the grep command.

The egrep command:

- Egrep stands for extended grep.
- Egrep is most powerful of the three grep utilites.
- The foremost advantage of this command is that of supports multiple search patterns can be handled very easily.
- The pipe(\) character is used to mention alternate patterns.
  Syntax: $egrep '(iyer/guptha)' student.lst
  CS018 |karthik iyer iv CSE
  CS019 |sourab guptha iv IT
  $
  In the above example, both patterns are used as alternative patterns.

Fast grep (f grep):

- This command uses only fixed character patterns.
- It does not allow use of regular expressions.
- It is used for searching large files.
- This command also accepts multiple patterns.
- Whenever multiple patterns are used, they are separated by a new line charcter.

Eg: $ fgrep ‘iyer

>guptha

> murthy ‘student.lst

CS018 |karthik iyer iv CSE

CS019 |sourab guptha iv IT

The stream editor(sed):

- Sed is acronym for stream editor.
- It is extremely powerful editor by using which, one can perform quick and easy changes to a file without entering into an editor like vi or emacs and others.
- Most after it is used to extract and manipulate records or lines of medium and large size files.

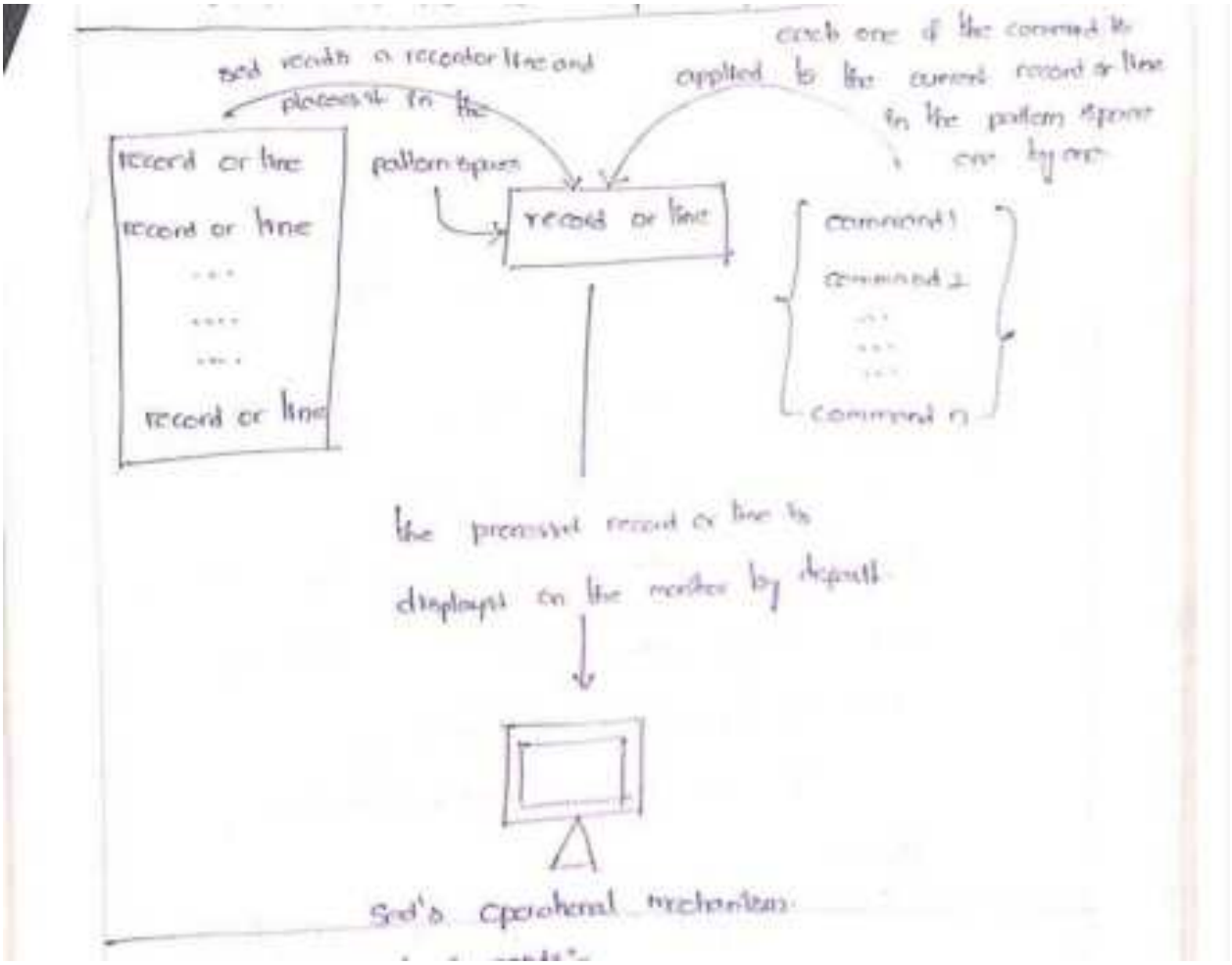
Syntax: $ sed options ‘address _ action list file list, where,

- Action - part of the address action list informs the servers about action or actions to be taken and the address part identifies a line or lines on which these actions are to be taken.
- The file list -it holds zero or more filenames from which lines are picked up one by one processed and sent onto the standard output(i.e) monitor.
- When no file is specified the i/p for the command is taken standard i/p (i.e) keyboard.

Operational Mechanism of sed:

- Sed reads in one line at a time, holds it in a memory space called pattern space and acts on it as mentioned in the sed command.
- If then reads in the next line, acts on it in the same manner and so on.
- By default all the processed lines are sent to the standard output (i.e) the monitor.

- The sed processing does not effect the original content of the file any ways. If requires, the processed output can be written on to a separate file.



Modes of giving sed command:

They are 2 ways for giving sed commands, they are

Inline method: sed instructions can be given by either inline or separate file.

Separate file: This method Is used only when there are no. of instructions to be used or instructions are usually entered repeatedly.

- The file that holds a number of sed commands **(i**.e) sed's address-action component is generally referred to as script file.

Addressing: Line and context addressing:

- By default sed considers every line of the input file, takes the desired action and then displays it on the standard output
- In the practice, it may not be necessary to pick up of lines and act on each of them.
- The basis on which only required lines are placed up for same form of processing is called addressing.

They are 2 forms of addressing
1) line addressing
2) context addressing

1) **Line addressing**: The simplest basis upon which only the required lines are placed up is to use line numbers , this method of using a line number or a group of line numbers to pickup only required lines is known as line addressing.

2) **Context addressing:** Another to pick up only method that is popularly used to employ a search pattern and to pick up only such lines that contain the search pattern. This method of crying a pattern to pick up one or more lines is called context addressing.

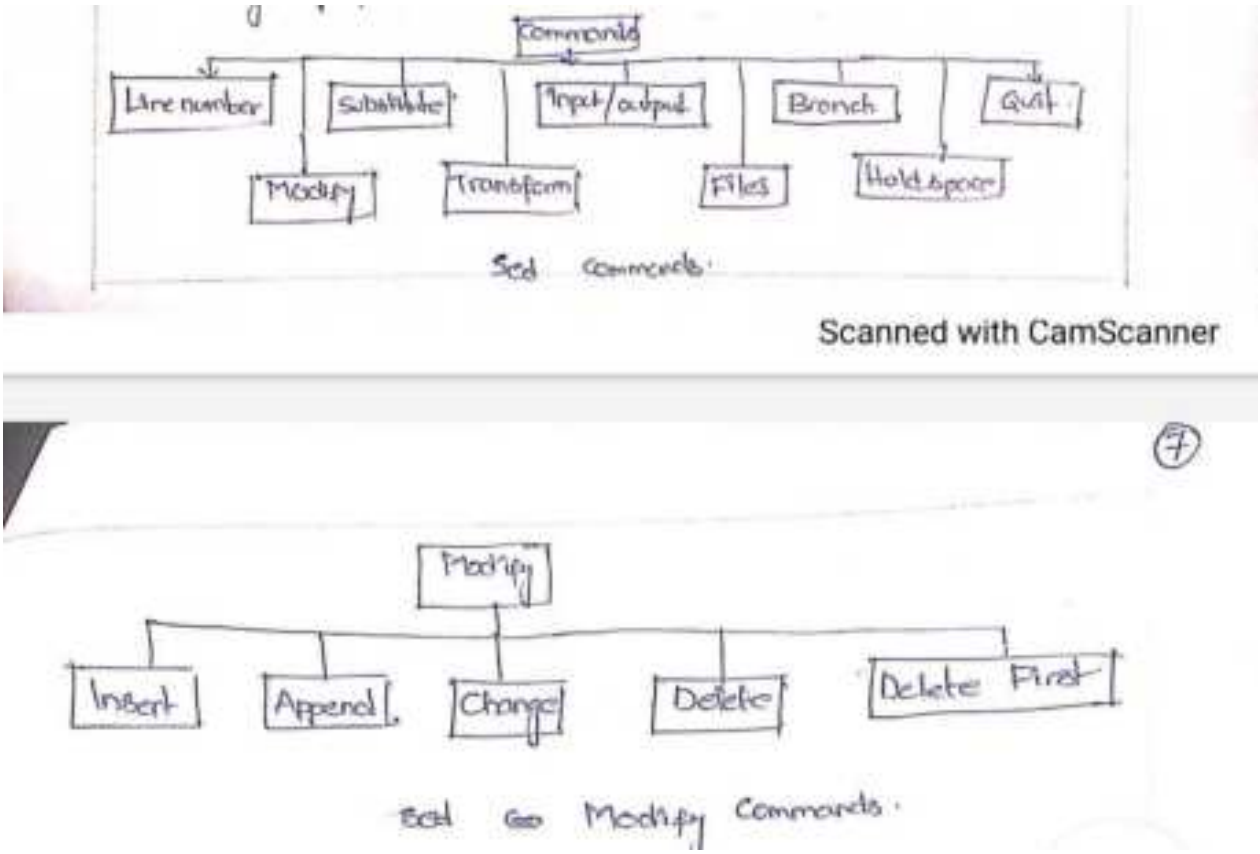
Multiple instructions:

Multiple commands are given employing one of the following methods.

1) Using –e option with every command
2) Using the quote method in which case each command has to be given in a separate line.
3) Using a script file (i.e)a test file that holds a command per line.

Sed commands:

There are 25 commands that can be used in a instruction. We group them in to categories based on how they perform their task.



All modify commands apply to the whole line, you cannot modify just part of a line.

Insert command:

- Insert adds one or more lines directly to the output before the address.
- This command can only be used with the single line and a ref of lines.

```
$ sed  '\p\
> This is inserted line
>' file1
This is the inserted line
This is sed
This is line number command
End of file.
```

The append command:

One or more lines or records can appended to an existing file or database by using the append command.

Eg: The append command by appending a dashed line separator after every line.

Script name:- appending sed.sed

a\

$a\

\ $ sed –f appending sed file 1

Output:

This is sed

This is line number command

End of the file

Change command: it replaces a modified line with next text.

Eg:

Script Name: Change sed

2c\

This is change command

$ sed –f changed .sed file 1

This is sed

This is change command

End of files

Delete pattern space command:

- It comes in 2 versions
- When a lowercase delete command(d) is used, it deletes the entire pattern spaces.
- When delete all lines in the pattern space that begin with uppercatutese “o”.

Substitute command:

- **Pattern substitution is one of the most powerful commands in sed.**
- In general substitute replaces text that is selected by a regular expression with a replacement string.
- With it, we can add, delete or change text in one or more lines.

Input and output commands:

The sed utility automatically needs text from the input file and writes data to the output files usually the standard output.

There are 5 input/output commands. They are

- Next(n)
- Print()
- List(l)
- Append next(N)
- Print first line(p)

AWK:

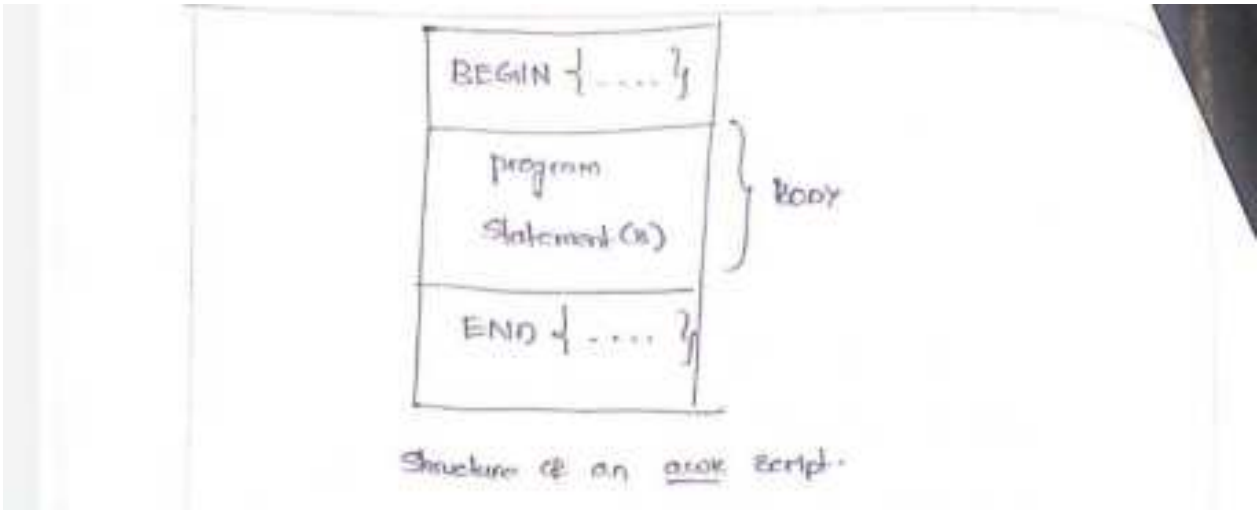
- AWk is a filter that was originally developed in 1977 by aho wernberger and kernighan as a pattern scanning language.
- The name awk is derived from the first letters of its developers surnames.
- It is programming language c- like control structures, functions and variables.
- It was designed to work with structured files and text patterns.
- One of the very important features of the filter is that is operates at the field level.

Syntax of AN awk program statement: The general format of awk command line is $ awk options 'program' file list

- Use $ options is optional
- File list will have 0 or more input filenames and
- Program will have 1 or more statements having the following general format.
  Pattern{action}

- **The pattern component of a programs statement indicates the basis for a line or record relation and manipulation.**
- The action part of every program statement is surrounded by a pair of curly brackets.
- This action part is made up of c-like statements, which performs actions on the lines or records selected based upon the pattern component.
- The pattern can be simple words or regular expressions as egrep or they can be more complicated conditions like in c language.
- Awk employs only 2 options, namely –f and –F options.
- -F specifies the input field separator.
- -f specifies that the program is on a separate file.

The structure of AWK script: Generally on awk script will be made up and time sections called the BEGIN, BODY and END sections.



Structure of an awk script.

- Awk script can have only one or more of these sections, however an awk program must contain at least one of these sections.

The BEGIN Section:

- The section is recognized by the keyword or the pattern BEGIN.
- All the instructions present in the section are executed once before the awk actually start reading and executing program statements from the body.
- Instructions in this section are used for performing tasks such as initializing variables, generating headings and other similar tasks, which must be completed before the body processing starts.

- A typical BEGIN section statement that identifies the input field separator as line as the colon(☺ character
  'BEGIN {FS":"}/FS – Field separator

The END Section:

- The end section is recognized by the keyword or the pattern end.
- All the instructions present in this section are executed once after the last line of the input has be processed.
- Instructions in this section are used for generating summary reports.
- A typical END section statement that prints number of records processed looks as
  'END { print NR}

The body section: This section contains one or more actual program statements.

Operational mechanism of awk:

- The working of awk is exactly similar to that of the stream editor sed.
- Like sed, awk also picks up the records or lines from the input file one by one and applies all the program statements present on the program file to each line.
- Here, applying all the program lines means that pattern portion of every program statement is compared with the presently picked up line one by one.
- Whenever the pattern portion of a program statement models, the action mentioned in the action portion of the program statement is carried out on the present input line**.**

UNIT-V

Shell Programming-Shell Variables-The Export Command-The Profile File a Script Run During Starting-The First Shell Script-The read Command-Positional parameters-The $? Variable knowing the exit Status-More about the Set Command-The Exit Command-Branching Control Structures-Loop Control Structures-The Continue and Break Statement-The Expr Command: Performing Integer Arithmetic-Real Arithmetic in Shell Programs-The here Document(<<)-The Sleep Command-Debugging Scripts-The Script Command-The Eval Command-The Exec Command

Shell variables:

A part from during an interface between users and kernel and the command processor, the shell also has programming capabilities of its own.

As any other language, variables are defined and used with a shell

There are 3 types of shell variables they are

1. **system variables**
2. **local variables**
3. **read – only variables**

1. system variables:

- System variables are set either during the boot sequence or immediately after logging in
- The working environment under which a user works, depends entirely upon the values of these variables depends entirely upon the values of these variables. These variables are also known as "environment variable"
- These are similar to global variables in general sense.
- By convention these variables are written using appearance letters only.
  Eg: PATH, HOME, MAIL, SHELL,TERM, IFS etc

PATH variable:

- The path variable holds a list directories in a certain order. In this list(:) separate different directories.
- The value of PATH can be printed using echo command.

```
Eg: $echo $ PATH
    /usr/local/ sbin:/sbin:/usr/bin/x11:
    $
```

- When any command is given , the shell searches for its program in the directories listed in the PATH one by one.

The HOME variable:

- When a user gets logged in he/she will be automatically placed in the home directory.
- The directory is decided an account for a user and is stored in the file etc/passwd.
- The value of the path of the home directory is stored in the variable HOME.
- The value of HOME variable can be known by
- $echo $ HOME
- /USR/MGV

The IFS VARIABLE (INTERNAL FIELD SEPERATOR)

- The variable holds tokens used by the shell commands to parse a string into substrings such as a word or record into its individual fields.
- By default space(),newline(\n), new tab spa e(\t) system uses them as default field specifiers.

The MAIL variable:

- Holds the absolute path name of the file where the users mail is kept.

- Mails we have received are specified by using this variable. We can also set the path /usr/spool/user.

The SHEEL variable:

- Contains the name of the users a shell program in the form as absolute pathname.
- The value of shell can be known by
- $ echo $ SHELL
- /bin /bash
- $

The TERM variable:

- The variable holds the information regarding the type of terminal being used.

2. Local variables:

- These variables are defined and used by specific users they are also called user defined variables.
- These variables exist only for a short time during the execution of shell script.

Rules for constructing variable names:

- These variables are also constructed by alphanumeric characters and underscore(_).
- They are case sensitive.
- Eg: sum, SUM are different.
- Defining a shell variable:
- A shell variable is defined using an equal to (=) operator without any space on either sides of it.

  Syntax: variable = value

  Eg: $X = 37
- The default value of shell variables will be a null string.
- Type of a shell variables:
- All shell variables are of string type and the values of variables are stored in ASCII format.
- By default shell variables are initialized as null strings.
- While writing shell programs it is not necessary to type declare or intialize shell variables.

Evaluating a shell variable:

- A shell variables are evaluated by prefixing the variable name with as $.

- When the shell reads a commad line, all words that are preceded by a $ are identified and evaluated as variables unless otherwise the $ is despecialized.
- Eg: $x = 37

```
$ echo $x
37
$ echo $xyz #since xyz is not initialized
$                # null string will be output
```

- Variables are concentrated by placing them adjacent to each other as

```
$x = hello; y = vignan
$Z = $x $y
$echo $x
Hello vignan
$
```

- Some times shell variables are useful in spending up the interaction of the user with the system.
- A shell variable can also be used to replace a command.
- $count = ‘wc file 1 file2 ‘
- $$count

3. read only variables:

- The value of the read only variables can not be manipulated . A variable can be made read only by using read only function.
- The variables can behave as constant in c language.

Eg:

```
1. echo “enter value of x”
2. read x
3. Echo “value of x : $x”
4. Read only x
5. X = ‘expr$x +1’
6. Echo “after increment x value: $x”
```

An error will be displayed like,

Line 5: x is read only variable

4. The expert command:-

- Values of the variables set or changed in one program will not be available to other programs..
- It is possible to make the values available across all programs or processes by using the export command.

1. Ex:- $x=10
2. $echo $x
3. 10
4. >echo $x
5. $x=10
6. $export x
7. $sh
8. echo $x
9. 10

5. **The profile file –**A script run during starting

- Every user has a .profile of his or her own. This file is a shell script that will be present in the home directory of the user.
- As this file resides in HOME directory, it gets executed as soon as the user login
- The system administrator provides each user with a profile that will be just sufficient to have a minimum working environment.
- However, in a practical situation, these profiles can be large the user can then edit and customize the same according to his or her convenience.
- Because this file is automatically executed on login, it is called the AUTO.EXEC.BAT file of unix.
- The contents of a typical, .profile file of a user is given here

Ex:-$cat .profile

```
#user $HOME /.profile –commands executed at login time
Home =/home/mgv/programs
PATH=$PAYH:$HOME/bin:/usr/bin/x11:/usr/hosts::
MAIL=/usr/spool/mail/$LOGNAME          #mailbox location
IFS=
PS1="$1
"
PS2=">"
Echo "today's date is 'date' "
News
Echo "you are now in the $HOME directory"
$
```

- As already mentioned the user can customize the operating environment to suit his or her requirements by manipulating system variables, adding and modifying statements in the .profile file.

6.The first shell script:-

Most , often using one command to get a task done is found to be inadequate.

- In other words, more than one command is required to get the task done.
- In such circumstances all the necessary commands that are required are put in a separate file in the required sequence, and the file is executed.
- Such a set commands that are taken together as a single unit within a file and executed at stretch is called a shell program or a shell script.
- Below is given an example of a shell script that is present in a file called my - script.sh.
- Eg: cat my – script.sh

```
Clear
Echo “this is my first shell script”
Echo “today’s date is ‘date / cut –d”. “-f 1-3”
Echo “good luck”
```

Executing a shell :

- A shell script can be executed in 2 ways , the straight farword method of executed a shell script is by using the shell command sh as shown below.

Eg: $sh my- script.sh

```
This is my first shell script
Today’s date is tue nov 26
Good luck
```

The other way of executing a shell script is to first assign the execute permission to the script file, using the chmod command and the executing the script by using the script file name directly.

Eg: $chmod u + x my-script.sh

```
$ my-script.sh
This is my first shell script Today’s date is tue nov 26 Good luck
```

Comments:

- Comments are used to explain the purpose and logic of the program and commands used in the program that are not obvious.
- In shell scripts comments are written using the hash(#) character as the first character of the comment line.

Eg:

```
# this is comment line
# comments are written as a part of documentation
```

Date # show the current date

- The read command:
- The read command is used to give input to a shell program interactively.
- This command reads just ne line and assigns the line to one or more shell variables.
- Following is a script that is in a file called the readname.sh
- The script just reads in a name and displays it on the month.
- Eg: $cat readname.sh
- Echo what is your name \?
- Read name
- Echo the is $ name
- $
- The file readname.sh is executed, after giving the executed permission to this file.

Eg: $chmod u+ x readname.sh
$ readname.sh
What is your name
Murthy
The name is murthy

Multiple arguments:

The read command can take multiple arguments. In other words the value for more than one variable can be input or assigned, using a single read command.

Eg: In which values to three variables a, b and c may be read is given here

$ read a b c

- There could be possibility that number of values input are either less or more than the number of read's arguments.
- In such situations values are assigned as follows
- If number of values input are less the of arguments, then the argument or variables to which values are not input will be initiated t null.
- If the number of values input are more than the number of arguments, the first (n-1) values are assigned to the (n-1) arguments and all the remaining input values assigned to the nth argument.
- This is an important feature.
- Eg: let w, x, y, z are the 4 input values. The execution of the $read a b c command, assigns w to a, x to b, and y to c.

The read only function:

- Variables, the values of which can be only be read but not be manipulated, are called read – only variables.
- Any required variable can be made read – only by using the read only function.

Eg: $cat example

```
Echo input a value of
x Read x
Echo the value of x is
$x Read only x
X= 'expr $ x +1'
Echo the value of x now is $x
$
```

the execution of the above shell program gives the output as,

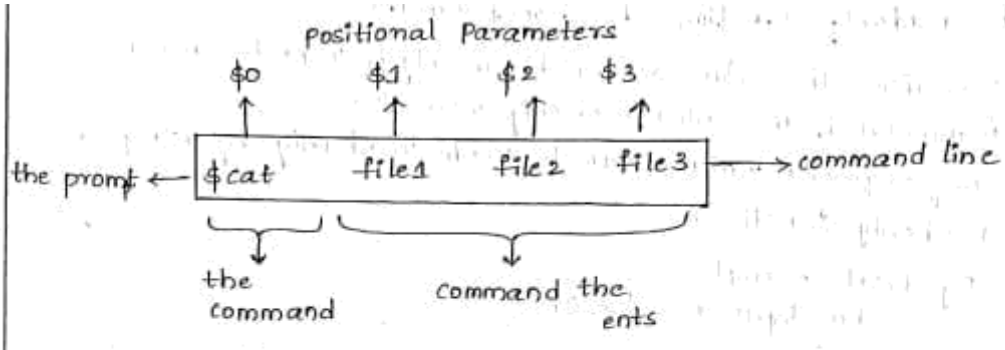
```
$ sh example
Input a value of x
The value of x is
4
Example: line 5: x : read only variables
$
```

- The read – only variable becomes a non –read only variable when it is exported, using the export command.

Positional parameters:

- Arguments submitted in the shell script are called positional parameters as the first argument is pasr no 1, second argument is passed on as parameter no2 and so on.
- Actually these parameters are stored in certain special shell variables.
- There are nine such variables that capture and hold values given in a command line. These are $1, $2,……$9. The $1 holds the first argument , the $2 holds the second argument.
- As the arguments are assigned as values to the special values $1, $2, $3 and so on, depending upon their physical position the command line they are called positional parameters.

1:33 PM anit5 unix



$0 is a special shell variable that holds the parameter number 0 (zero), th

* $#, $* and $@ variables:-

In addition to he special variables, there are three more special variables used by the shell

ters

→ The $* variable holds the list of all the arguments. It may be observed that the $# variable is similar to argc and the $* variable is similar to argv[] in C.

→ Like $*, the $@ variable also holds the list of arguments present in the command line.

→ When these variables $* and $@ are used within quote marks, the contents of $* is considered as a single string whereas in the case of $@ each of the arguments independently quoted and considered as independent string arguments.

Scanned with CamScanner

06

The script pasex.sh given below accepts any number of arguments and displays the contents of all the special variables and some of the positional parameters.

Eg:- cat .sh

```
echo 'no of arguments : '$#
echo 'values of parameters : '$*
echo 'first param value : '$1
```

Difference between $* and $@:

- The following example illustrates the subtitle difference between $* and $@ special variables.
- The files fa, fb, fc are assumed to be present in the current working directory.

Eg: $cat pos-par1.sh

```
#more than one existing file name to be given at the command line.
ls "$*""
Ls "$@"
$
```

- The following output is obtained when the above script is run with the existing filenames fa, fb, fc as arguments.

```
$ sg pos-par1.sh fa fb fc
Ls: fa fb fc: no such file or
directory Fa fb fc
$
```

The set command:

Assigning values to positional parameters can be assigned using the set command as shown in the following example.

Eg: $ set friends is need are friends inneed.

- The execution of the above command line, assigns friends to the parameter $1, in to the parameter $2, need to the parameter $3 and so on.
- These assignments can be verified by using the echo command as shown below.

```
$ echo $1 $4 $6
Friends are inneed
$
```

Positional parameters and excess arguments:

- In certain situations are may give more than nine arguments in a command line.
- The results in excess values to be assigned to nine positional parameters.
- An example that illustrates the assignment of excess arguments to positional parameters follows.
- Eg: set everyone has the capacity to learn from mistakes. He learns a lot from experience.
- $ echo $1 $2 $3 $4 $5 $6 $7 $8 $9 $10 $11
- Everyone has the capacity to learn from mistakes. He everyone[].

- In the above example an attempt to echo the value of $10 has resulted in everyone 0 ($1's value and a zero) an the value of $11 has resulted in everyone1('$1's value and a 1).
- The shift command – handling excess command line arguments:
- It is known that they are only nine positional parameters are therefore only a maximum of nine arguments can be given in a command line.
- In case , more than 9 arguments are given in a command line, no error is indicated.
- However, the behavior will be ambiguous as illustrated in the previous section such type of situations are handled using the shift command.
- When used, the shift statement shifts out the values assigned to the positional parameters to the left by an integer value mentioned with the shift statements as its arguments.

  Eg: $shift 5

  $ echo $1 $2 $3 $4 $5 $6 $7 $8 $9

  Learn from mistakes. he learns a lot from experience.

  $
- Thus the statement $ shift 5 moves the parameters values toth valu the left by 5 positions.
- Actually it shifts out the first 5 values and assigns the 6th value of 41, **7th** value of $2 and soon.
- It should be noted that when certain values are shifted using a shift command. Shifted values would be lost.
- An example that illustrates the use of shift command the above example can be studied along with the previous example.

The $? Variable - knowing the exit status:

- When ever a command, that is, a program, is run it may either get executed successfully and yield a result or it may not get executed successfully.
- Whenever a command is successfully executed the program returns o.
- However, if a command is not executed successfully a value than 0 will be returned .
- Logically, a 0 is considered as true and a non-zero value is considered as false.
- The returned values are called program exit status value and will be available in one of the shell's special variables called $?
- An exit status value available in $? Is normally used in decision making in shell programs. Eg: $cat sample.sh

  Cat: sample .sh :No such file or directory

  $echo $?

  1 $

The exit command:

- The command is used to terminate the execution of a script as and when required.
- This command can optionally use a numeric arguments.
- If no arguments are used this command returns a zero.

Branching control structures:

- Like programs written in computer language shell program statements also get executed sequentially in the order in which they appear.
- There are a number of situations where one has to change the order of execution of program statements
- This needs a sort of decision making thus we use control structures that are used to shift the point of execution are called branching control statements.

1) if conditional:

This is the simplest of all the branching control structures. It has the following general format.

```
Syntax:     if test _ expression
                Then
                    True – block
            If
```

- when the shell comes across an if construct, it evaluates the test expression first.
- If this evaluation results in a true exit status, then the commands will be executed.
- An if construct may also have an else block as shown below

```
If test _ expression
  Then
      True - block
Els
e       False – block

Fi
```

- If the condition is true then the true block will be executed, otherwise false – block will be executed. A par from two forms discussed above, the if has branching.

```
tIf test – expression
    Then
        Command(s)
Elif test- expression
    Then
        Command(s)
Else
```

```
        Command(s)
    Fi
```

The test command:

- This is a built in shell command that evaluates the expression given to its as an argument and returns true if the evaluation of the expression returns a zero or false if the evaluation returns non-zero.
- This command can be used directly using the keyboard test or indirectly by using square brackets.

```
Eg:  ans = "y"
     If test "$ans" = "y"
       Then
           Echo "yes"
     Fi
```

Numeric tests:

- In the numeric tests , two numbers are compared using relational operators that are listed along with their meaning.

Operator	meaning
-eq	equal to
-ne	not equal to
-gt	greater than
-gc	greater than or equal to
-lt	less than
-lc	less than or equal to

```
Eg: read x y
    If ( 4x –lt
       4y) Then
     Else
          Echo " x is greater than
    Els
          y" Echo " x is less than
     e
          y"
     Fi
```

The case command:

- This command provides the multi way decision – making facility.
- Basically it works on pattern matching.

Syntax:

```
Case string – value
in Pattern 1)
command
        Command;;
Pattern2)  command
        Command;;
…….
…….
Pattern) command
        Command;;
*) echo "none of the patterns matched";;
Esac
```

Loop control statements:

Like all the computer languages shell also has loop control structures they are

1) the while command
2) the until command
3) the for command

The while command:

This is one of the very widely used loop – control structures

```
While [ condition ]
Do
    Set of statements that are to be executed
Done
```

- The while, do and done are keywords.
- The repeatedly executed as long as the condition remains true.
- This is an entry – controlled loop structure .

Eg: below is the program to print the numbers from 1 to 10 using while loop.

```
I = 1
While [ $i =lc 10
] Do
    Echo "value of i:4i"
        I= ' expr $i+1'
Done
```

The until command:

- Like the while command this command is also loop control commands.

- But this command behaves exactly in an opposite manner to the while command.
- The until command repeatedly executes the set of commands that appears between the do and done as long as the condition remains false.

  Syntax: until [condition]
  Do
  Set of commands to be executed repeatedly
  Done
- write a program to print the numbers from 1 to 10 using until loop I

```
= 1
Until [ $1 –gt
10] Do
  Echo " value of i:
$i" I='expr $i+1'
done
```

The for command:

This command works with a set of values generally given in a form of list.

Syntax: for var in
list Do
Statements
done

eg: to print 1 to10 numbers on screen

```
for I in 1 2 3 4 5 6 7 8 9 10
Do
  Echo " the value of I" $i
done
```

The continue and break statements:

- one can come out of the loop permanantly by using a break statement (for, while, until).
- Similarly ,a continue statement is used to resume the next iteration of loop without considering the statements that appear the continue statement with in the loop.

Eg:

```
$cat menu2.sh
Ans = y
While [ "$ans " = "y"]n

Do
   Echo "menu\n
1. List of files\n
2. today's date\n
3.process status\n
4.users of the system\n
5. Present working directory\n
6.quit to unix\
Enter your option;\c"
Case "$ choice" in
1) ls –l;;
2) date;;
3)ps;;
4)who;;
5)pwd;;
6)break;;
7)echo "invalid choice"
      Continue;;
Esac
    Done
```

The expr command: performing integer arithmetic:

- This command is used to carry out the basic structure operations including modulo division on integers.
- This command is used only when arithmetic operations are sample and are few.
- An important limitation of performing arithmetic with this command is that it is slow.
- The operators used for performing arithmetic operations are +(plus) for addition – (minus) for subtraction, *(multiplication), /(slash) for division and %(percentage) for modulo division.
- While using the command one must remember that on either sides of arithmetic operators a blank or a tab must be present and the operator, must be despecialized must be present , otherwise shell treats it are a wild card.

```
Eg: $x = 3; y = 5
$expr 3 + 5
9
$expr $x -  $y
-3
-2
$expr3\*5
15
$expr 4y/$x
1
$expr 13%5
3
$
```

- The expr command is often used with command substitution to assign values to variables as shown below.

```
$x=6; y=2;z='expr $x+$y'
$x='expr $x+1'        #implementation of x++
$
```

- A part from performing numeric computations the expr command can also perform certain string manipulations some of them are:
- To determine the length of a given string.
- To extract a sub-string from a given string.
- To locate the position of a character in a string.
- During any of the string manipulations using expr ,one has to use two expressions, separated by a colon(:).
- The string to be handled in placed on the left side of the colon(:) and a regular expression is placed on its right.
- It should be noted that a blank must be present on either side of the colon(:).

Ex:- $ expr "quality": '.*'

```
6
$
```

- A substring is extracted using a tagged regular expressions (TRE) as shown below.

```
$stg=1949
$expr '$stg: '..\(..)\'
49
$
```

- As already mentioned , expr works only on integers. As such ,one has to think how are real numbers handled. This situation is managed using the echo and be command .

```
$c='echo $a+$b|bc\
```

Real arithmatic in shell programs:-

- It is possible to perform integer computations with the shell.
- In other words, it is not possible to perform the real arithmetic with the shell.
- But real arithmetic can be managed using the basic calculator (bc) along with the scale function.

Area of triangle:-

- The script given below accepts the basic and the height of a triangle ,computes its area and outputs the same.
- The scale function sets the precision to two digits after the decimal point.

Ex:- $cat triangle _area.sh

```
 echo "\n Enter a value of base: \c"
read base
echo "\n Enter a value of height : \c"
read height
area = 'echo "scale=2 \n ½ * $base *$height"|bc'
echo "The area of the triangle is $area"
$
```

Degree Fahrenheit to degree Celsius:-

- The script below accepts temperature in degree Fahrenheit ,converts it into degree Celsius and outputs the same.

```
$cat degree_conv.sh
echo "\n Enter a Fahrenheit value :\c"
read    fahr
cel = 'echo "scale =2\n 5/9*($fahr-32)"|bc'
echo "The equivalent degree Celsius= $cel"
$
```

- These two examples have been successfully run with the Bourne shell.
- However, the escape characters will be effective only with the use of the –n option with the echo command.

Gross salary of an employee:-

- A shell script that computes the gross salary of an employee according to the fallowing rules is given below.
- The basic salary is input interactively through the keyboard.
- If the basic salary is <1500 then HRA=10% of the basic and DA=90% of the basic.
- If basic salary is >=1500 then is HAR=rs 500 and DA=98% of the

```
#salary computation example
Echo "\n enter basic salary: \e"
read basic
if [ $baic -lt 1500 ]
then
   hra= 'echo "scale=2;$basic *10/100" |bc'
   da= 'echo " scale =2;$basic *90/100" |bc '
else
   hra=500
   da= 'echo "scale =2; $basic *98/100" |bc'
fi
gsalary= 'echo "scale =2;$basic +$hra+$da "
|bc' echo "gross salary =Rs.$gsalary"
   $
```

- Two typical outputs of the execution of are bove mentioned script are given below

```
$salary.sh
Enter basic salary :
Rs.1200 Gross salary=
rs.2400.00
$salary.sh
Enter basic salary:1675
Gross
salary=rs.3816.50
```

The here document(<<):-

- It is known that a command like grep works with files.
- For example, the command $grep "karthik" student.lst searches for all the lines that have the string karthik in the file student.lst.
- The mechanism involved here is to search for the file student.lst, which will be else where, to open it and search for karthik on it.
- In fact, grep is also a program present on a file, which is also has to opened.
- Thus , in the example two files are accessed and opened.
- In unix it is possible to include the document on which the system has to operate along with the command itself as shown in the fallowing example.

```
$grep "^03" <<end #this is the command the ***
>01 : architecture : 456
>02:computer science : 556
>03 : electronics : 656
>04 : mechanical : 756          >end
```

03: electronics : 656
$

- In the above example, the double less than symbol (<<) informs the shell that the document on which the command has to operate is here itself.
- The word that fallows the << characters is called the end marker (delimiter).
- As seen in the example above the end marker that fallows the command line must appear in the document as its last line.
- Every thing lies in between the command line and the end marker is taken as the input.
- The document that lies between the command line with the << characters and the end marker is known as the here document.
- One of the advantages of using a here document is that only one file has to be opened and therefore execution will be faster.
- Another advantage of the here document is that it can successfully used with communication command mail, write, wall and others that does not accept file names as their arguments.

The sleep command:

- Using the command the user can make the system to sleep that is pause for some fixed period of time.
- $ sleep 60; echo " the system was sleeping for 60 seconds"
- The system was sleeping for 60 seconds
- $
- The facility is particularly useful to introduce required amount of delay in a shell script whenever required.

Debugging scripts:

- It is necessary to debug the shell script.
- Shell scripts can be debugged either with the execute trace option (-x) or verbose option (-v)
- With these options one can check the value of all the variables involved and view the logical flow and the program.
- Debug options can be used in 2 ways. In one method the options are used in the command line during the execution of scripts.
- For example, in order to debug a script called mesg.sh, the command line will be $sh –x mesh.gh good luck, where good luck is argument in the script.
- The execute trace option(-x) prints each command, preceded by a plus (+) sign, before it is executed.

- It also replaces the value of each variable accessed in the statement.

```
$ cat mesh.sh
# to print message for required number of times.
Set –x
Echo " how many no. of times the message is to be displayed?"
Read count
Until [ 4 count –eq 0 ]
Do
   Echo 4*
   Count = ' expr $ count -
1' Done
Set + x
$
$sh mesg.sh Good Luck
+Echo 'how many no. of times the message is to displayed?'
How many no. of times the message is to be displayed/
+read count
3
+ [ '3 –eq 0 ']'
+echo Good
Luck Good Luck
++ expr 3 -1
+count = 2
+'['2 –eq 0']'
+echo Good
Luck Good Luck
++expr 2 -1
+count = 1
+'[' 1 –eq 0']'
+echo Good Luck
++expr 1 -1
+count = 0
+'[' 0 –eq 0']'
$
```

The use of the verbose option –v behaves almost similarly.

The script command:

- This command is used to record an interactive session.

- It is invoked either without any argument or with a filename as its argument.
- When this command Is used without any argument the session is recorded in a file called the typescript, by default.
- However, when this command is used with a filename as its arguments the session is recorded in the argument file.
- In either cases, as soon as the script command is used, a message is displayed.
- $ script
- Script started, file is typescript
- $script darsha
- Script started, file is darsha.
- One the script command is invoked, everything that is done at the terminal will be automatically recorded in the corresponding log file.
- Recording a session is terminated using the exit command.
- A recorded session may be displayed on the terminal or printed or \ pr command.
- A session can be appended to an existing log file using the append option –a, as shown in the following example
- $ script –a
- $
- The session that starts now will be appended to the earlier contents of the default log file typescript.

Advantage:

- Session of trial runs for a new script written by a user are recorded for the purpose of analyzing its behavior and take corrective measures, if required.

The eval command:

- As it is already known when a command is given , the shell scans the command line once, makes command substitutions if required, evaluates variables, if necessary , interprets special characters, if any and then executes it.
- The use of eval command makes the shell to scan the command line once more, that is second time and then actually executes the command line.
- B = a
- C = b
- Eval echo \$$c
- A
- The first two statements in this example are assignment statements.

- When the shell comes across the third statement , because of eval, it first scans the statement once for any possible pre evaluation or subsistution.
- Here because of the meta character \ the file $ is overlooked and the next variable $c gets evaluated resulting in b.
- After this evaluation the third statement will be equivalent to echo $b.
- Then this statement gets executed as usual by the shell resulting in as the answer.
- The exec command:
- This command has the following 2 abilites.
- Running a command with out creating a new process.
- Redirecting a standard input, output or error of a shell script from with in the script.
- Normally when the shell executes a program or a command that is not internal to it, a new process is created by it.
- However, when a command is run using the exec command the new program is overlayed on the current process gets terminated.
- Obviously, in this case, the process of the shell itself gets terminated and therefore the current user is logged out.
  $exec date
  Fri jun 18 15:10:00 ist
  2018 Login:
- The exec command can run both scripts and compiled programs.
- As no process is created, the command on the program executed using the exec command runs quicker.
- As already mentioned this command can be used to close the standard input called in file, the exec open it with any file the user wants to read.
- To change the standard input to a file called in file, the exec command is shown below.
- Exec < in file
- Hence forth any command that reads data from the standard input will do so from the file in file.
- Redirection of standard output is done similarly.
- The command exec>out file redirects all the subsequent output to the file out file.
- Standard input and standard output can be reassigned their default values by using the exec once again.
- For example the command exec</dev/tty reassigns the standard input back to the terminal.

- $ cat exec
- Echo enter filename
- Read filename
- Exec < $filename
- While readline
- Do
- Echo $ line
- Done
- Exec </dev/tty>
- $

UNIT-VI

The Process-The Meaning-Parent and Child Processes-Types of Processes-More about Foreground and Background processes-Internal and External Commands-Process Creation-The Trap Command-The Stty Command-The Kill Command-Job Control.

Process:

- A process is an abstract concept by using which one can explain understand and control the execution of a program in a operating system.
- A process is defined as a program in execution.
- Unix being a multiuser and multi tasking system at there could be a several programs belonging to different users or the same user running at the same cpu.
- All these programs shares the same cpu.
- The kernel generates or spawns processes for every program under execution and allocates definite and equal cpu slot time slots to these various programs.
- Each of these processes have a unique identification number allocated to it by the kernel called process identification numbers or pid's.
- Mathematically, a process is represented by the tuple (process id, code, data, register values, pc value),
- Where process id (pid) is the unique identification number that is used to identify the process uniquely from other processes code is the program code that is under execution , data is the data used during is under execution, register values are the values in cpu registers and pc value is the address in the program counter from where the execution of the program starts.

Parent and child process: In unix, a process is responsible for generating another process.

- A process that generates another process is called the parent of the newly generated process, called the child.
- Eg: when a command like $cat sample.lst is given, the shell creates a process for running the cat command.
- Thus the shell sh (ksh or bash) being a process, generates another process (cat).
- Here the shell process is the parent process and the cat process is child process.

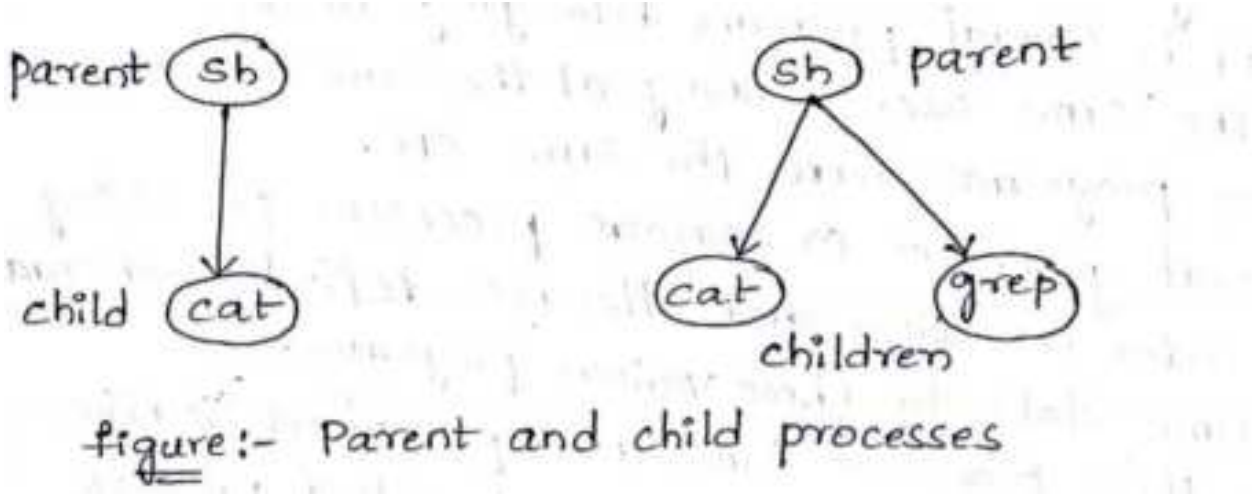


figure:- Parent and child processes

- In general, a parent process waits for the complete execution of its child process a parent waits for its child to die.
- However sometimes of its child a parent may die before its child.
- In such cases the child is said to be orphan.
- Generally the orphan processes are attached to the init ph process with pid's.

Program and a process: All the processes get themselves arranged in the form of a hierarchial inverted tree like structure.

- This is similar to file organization.
- This is only difference between these 2 organizations is that the file organization is locational where as process organization is temporal.

Types of process: process within unix are classified in to three general categories as

1) Interactive processes.
2) Non – interactive processes
3) Daemons.

1) **Interactive processes** : All the user process, which are created by users with the shell sct upon the directions of the users and are normally attached to the terminal are called interactive processes or foreground processes.
2) **Non – interactive processes**: Certain processes can be made to run independent of terminals. Such processes that run without any attachment to a terminal called non – interactive processes. These types of processes are also called background processes.
3) **Daemons**: All the processes that keep running always without holding up any terminals and keep waiting for certain instructions either from the system or the user and then immediately get into action are called daemons. Swapper, init, corn, bdflush , vhandle are some of daemons. The daemons came into existence as soon as the system is booted and will be alive till the system is shut down. One cannot kill these processes prematurely.

More about foreground and background processes: commands that hold up the terminal during their execution are called foreground processes.

- The chief advantage of foreground processing is that, no further commands can be given from the terminal as long as the older one is running.
- This advantage becomes significant when a currently running process is big and takes a lot of time for processing.
- It is possible to make processes to run without using the terminal.
- Such processes takes their input from some file and process it without holding up the terminal and write their output on the another file are called background processes.
- Typical jobs that could be run in background are sorting of large database file or locating a file in a big file system.

Running a command in the background: A command is made to run in the background by terminating the command line with an ampersand character as shown in the following example.

Eg: $ sort -o student.lst student.lst &

567

$

Some of these problems could be due to any one of the following.

- The success or failure of the background process are not reported.
- The user has to find it out. For this purpose the identification number is used.

- The output has to be redirected to a file as otherwise display on the monitor gets mixed up.
- Too many processes running in the background degrades the overall efficiency of the system.
- There is danger of the user logging out when some processes are dtill running in the background.

Internal and external commands: The classification of commands depending on whether they generate separate processes are not upon their running is discussed here.

- Most of the commands such as cat, who and others generate separate processes as soon as they are used.
- Commands that generate separate processes upon their running are called external commands.
- Some commands such as mkdir, rm ,cd and others do not generate new processes when they are used ; such commands are called internal commands.

Process creation: There are 3 distinct phases in the creation of a processes. They are

1) Forking
2) Overlaying and execution and
3) Waiting

- These 3 phases are taken care by making calls to the system routines fork(), exec() and wait() respectively.
- Forking is the first phase in the creation of a process by a process.
- The calling process (parent) makes a call to the system routine fork() which then makes an exact copy of itself.
- The copy will be of the memory of the calling process at the time of fork() system call and not of the complete program the calling process was started with.
- Right after the fork() there will be 2 processes with identical memory images.
- Each one of these 2 processes has to return values.
- There will be 2 return values. The fork parent process returns the pid of the new process, the child process just created , where as the fork of child returns a 0(zero).
- In case a new child process is not created a -1 is returned.
- Immediately after forking, the parent keeps waiting for the child process to complete its task.
- It awaknes only when it receives a complete signal from the child after, which it will be free to continue with its other functions.
- The child process inherits almost the entire environment of the calling process.

- In other words the child process will have the same priority same signal handling settings, same group and user ids, same current directory and so on.
- However, children will not inherit the local variables and will have different pid's.
- In the second phase, the parent makes a system call to one of the exec() functions.
- The system call simply overwrites the text and data area of the child process by the text and data of the new program and then starts executing this new program.
- At the end of overlaying and execution , a call is made to the exit() function that terminates the child and sends a signal back to the parent after which, the parent becomes free to continue with its other functions.
- The entire mechanism of process creation is pictorially shown below.

nal

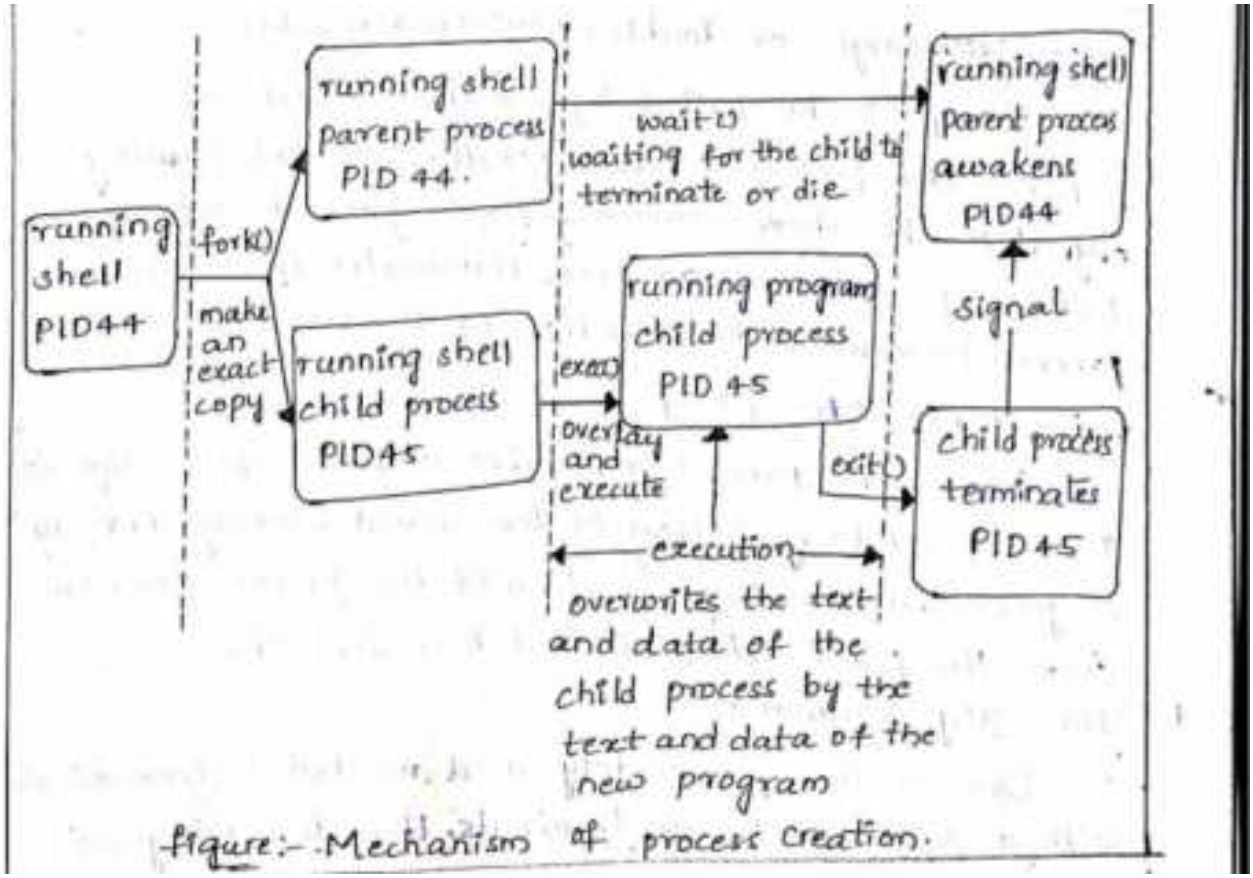


figure:- Mechanism of process creation.

The trap command: Normally signals are used to prematurely terminate the execution of a process either intentionally or uninbentionally .

- The trap command is used to trap one or more signals and then decide.A bout the further care of action .

- If no action is mentioned, then the signal just trapped and the execution of the program resumes from the point of rom where it had been left off. The general format of trap command is

  $ trap [commands] signal – numbers

- The command part is opertional. When it is present all the commands present in this part are executed one by one as soon as the process receives one of the signals specified in the signal - number list.
- The commands used, must be enclosed using either single or double quotation marks.

  1) 4 trap "echo killed by signal 15; exit" 15

  When the process receives a kill command, causing signal 15, the above command first gives the message killed by signal 15 and terminates the current killed by signal 15 and then terminates the current process because of the execution of the exit command

  2) $ trap "ls -l" 1 2 3

  When the process generates anyone of the signals 1, 2 or 3, along listing of the current working directory is generated and then execution of the process resumes from the point where it had been left off.

The stty command:

One of the most widely used methods in communicate with the system is to use terminals, that is via keywords.

- There are certain combination of keys, on these terminals, which control the behavior of any program in execution.
- Eg: we have been of

  1) <ctrl –m> (^m), that is the <RETURN> key to end a command line and execute the command.

  2) <ctrl –c> (^c) to interrupt a current process and to come back to the shell.

  3) <ctrl –s> (^s) to pause display on the monitor.

  4) <ctrl –d> (^d) to indicate end of file and so on.
- The sty command is used to see or verify the setting of different keys on the keyboard.
- The user can have a short listing of the settings by using the command without any arguments.
- In order to see all the settings, it has to be used with the –a (all) option.
  Eg: $ sty -a

```
Speed 9600 baud ; I speed 9600 baud; line = o(tty);
Erase = ^? ; kill =^ U; eof = ^d; intr =^c; stop = ^s;
Echo echoe .....

$
```

The output shown above is just illustration.

- From the output one can see that the terminal speed is 9600 bauds, one can see that the terminal speed Is used to indicate ^U is used for killing the line, ^D is used to indicate end of a file, because of echo everything
- Typed at the keyboard gets echoed on the displayed terminal backspacing over a character retains its display and so on.
- This command can also be used to change the key settings as shown in the following examples.

  Eg: $ stty -echo
  $ stty eof \ ^a
- It is recommend not to play around with the terminal settings.
- This may lead to improve working of the terminal.

The kill command: There are certain situation when one likes to terminate a process permanently. some of these situations are

- When the machine has hung.
- When a running program has gone in to an endless loop.
- When a program is doing unintended things.
- When the system performance goes below acceptable levels because of too many background process.
- Terminating a process prematurely is called killing.
- Killing a foreground process is straight far word.
- This is done by using either the DEL key or the BREAK key.
- However, to kill a background process the kill command is used.
- This command is given with the pid of the process to be killed as its arguments.
- If the pid is not known the ps command is used to know the same.
- Eg: A process having an identification number 555 can be killed using the kill command as shown in the following example.

  $ kill 555
- A kill command, when invoked sends a termination signal to the process being killed.
- When used without any option, it sends 15 as its default signal number.

- The signal number 15 is known as the software termination signal and is ignored by many processes. and the shell process sh, ignores signal 15. In other words signal and is ignored by many processes.
- At such times , one can use signal number 9, the sure kill signal, to terminate a process forcibly as shown in the following example.

$ kill -9 666 # 666 is the number of the process.

Job control:

Unix is a multi tasking system and there will be many number of jobs or processes running simultaneously in a unix environment.

- How many as well as which processes are ruuning currently.
- Terminate either a misbehaving or unwanted process.
- Modify the priority of a process.
- To push a process in to background, required process to
- To bring, required process to the foreground and so on.
- In unix , there exist many commands by using which, one can perform any of the job control activites.
  Eg: the ps command is used to know the details of currently running process.
- The kill command is used to prematurely kill a process and so on.

Job control commands:

- The jobs , fg and bg
- A command or command line with a number of commands put together or a script is generally referred to as a job
- In unix , as one can run commands in the background, there could be a number of commands, that is , process running in the background.
- Also there could be a command – a process – running in the foreground.

The Jobs command: A list of all the current jobs obtained using the jobs command as shown below

[ksh]jobs # the {ksh} prompt has been used intentionally

[1] + running sort emp. Data / grep 'Banglore'> address . lst &

[2] – running 1000

& [ksh]

- In the above output, a+ (plus) and – (minus) that appear after the job number mark the current and previous jobs, respectively.
- The word running indicates beings the job is current being executed.

The fg command: The command is used to bring a job that is being executed in the background currently to the foreground.

- This command can be either used without any arguments or with a job number as its arguments.
- Some simple illustrations are given here,
- [ksh] fg # brings the most recent background process to the foreground.
- [ksh] fg %2 # brings job number 2 to the foreground.
- [ksh] fg % sort #brings the job name of which begins with the sort too the foreground.
- As seen from the above examples when ever a job number is used as an argument with a job control command, it must be preceded by a % (sign).
- Here it may be noted that the current job may be referred to by using any one of the representations - %1 or % + or %%.
- Also it may be noted that the first few characters of a command sequence can be used to referred a to job as shown in the last example.

The Bg command: A new an can be made to run in the background by using & (ampersand) at the end of the command line.

- The currently running foreground process is first suspended by using ,< ctrl –z> keys , and then making it to run in the background by using the bg command.
- By assuming that the currently running process has been suspended right now, the following command line puts it in the background.

Eg: [ksh] bg % 1 # resumes job number 1 in the background