# 502-UNIX and Shell programming

TYBCA (Sem-5)

## Unit 1

## Introduction

1.1. Features of Unix OS

1.2. System Structure

1.3. Shell & its features

1.4. Kernel

1.5. Architecture of the UNIX OS

## Unit 2. Overview

2.1 Logging in & out

2.2 I node and File Structure

2.3 File System Structure and Features

2.4 Booting Sequence & init process

2.5 File Access Permissions

## Unit 3. Shell Programming

3.1 Screen Editor "vi"
3.2 Environmental & user defined variables
3.3 Argument Processing
3.4 Shell's interpretation at prompt
3.5 Arithmetic expression evaluation
3.6 Control Structure
3.7 Redirection
3.8 Background process & priorities of process
3.9 Conditional Execution

## Unit 4. Advanced Shell Programming

4.1. Filtering utilities: grep, sed etc.

4.2. awk utility

4.3. Batch process

4.4. Splitting (cat, cut, head and tail), comparing (cmp, comm., diff), Sorting(sort), Merging & Ordering files (paste, uniq)

## Unit 5. Communication with other users

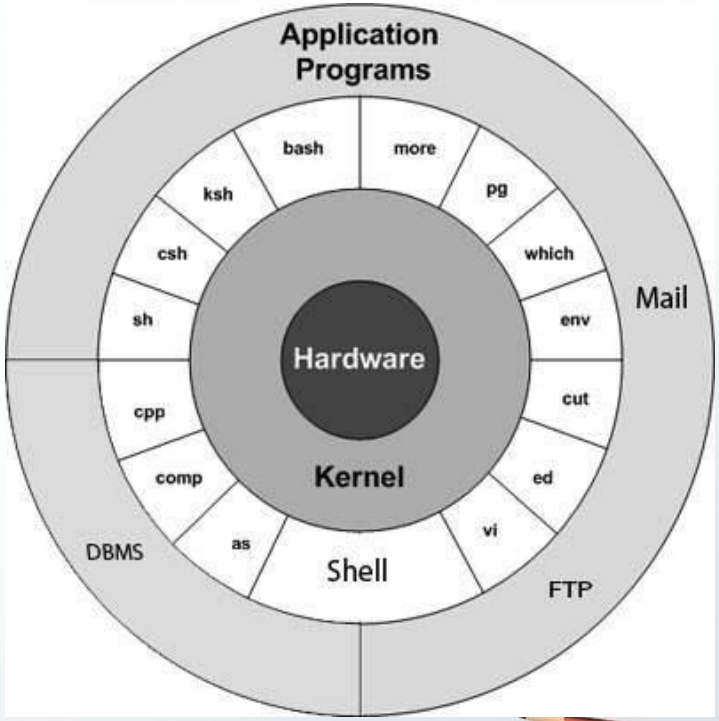
5.1 write, wall and mesg

5.2 mail, motd and news

# Unix

- UNIX is a computer operating system originally developed in 1969 by a group of AT&T at Bell Laboratory - Design by ken Thompson and Dennis Ritchie.
- Unix is called a multiuser system as well as , user can also run multiple programs at the same time; hence Unix is a multitasking environment.
- The Unix operating system is a set of programs that act as a link between the computer and the user.
- The computer programs that allocate the system resources and coordinate all the details of the computer's internals is called the **operating system** or the **kernel**.
- Users communicate with the kernel through a program known as the **shell**. The shell is a command line interpreter; it translates commands entered by the user and converts them into a language that is understood by the kernel.

# Unix Architecture



# Cont….

- **Kernel:** The kernel is the heart of the operating system. It interacts with the hardware and most of the tasks like memory management, task scheduling and file management.
- **Shell:** The shell is the utility that processes your requests. When you type in a command at your terminal, the shell interprets the command and calls the program that you want. The shell uses standard syntax for all commands. C Shell, Bourne Shell and Korn Shell are the most famous shells which are available with most of the Unix variants.
- **Commands and Utilities:** There are various commands and utilities which you can make use of in your day to day activities. **cp, mv, cat** and **grep,** etc. are few examples of commands and utilities. There are over 250 standard commands plus numerous others provided through 3rd party software. All the commands come along with various options.
- **Files and Directories:** All the data of Unix is organized into files. All files are then organized into directories. These directories are further organized into a tree-like structure called the **filesystem**.

# Features of UNIX

- Portability:
- Machine-independence:
- Multi-Tasking:
- Multi-User Operations:
- Hierarchical File System:
- UNIX shell:
- Pipes and Filters:
- Utilities:
- Software Development Tools:

# SHELL

- The shell acts as an interface between the user and the kernel.
- When a user logs in, the login program checks the username and password, and then starts another program called the shell.
- The shell is a command line interpreter (CLI). It interprets the commands the user types in and arranges for them to be carried out.
- The commands are themselves programs when they terminate, the shell gives the user another prompt.
- There are a number of different shells available, with names such as sh, csh, tcsh, ksh, bash, each with different rules of syntax.

**Types of Shell.**

- The Bourne Shell
- The C Shell
- The Korn Shell

# Kernel

- The main control program in a UNIX OS is called the kernel.
- The kernel of UNIX is the hub of the operating system: it allocates time and memory to programs and handles the file store and communications in response to system calls.
- The kernel does not allow the user to give it commands directly instead when the user types commands on the keyboard they are read by another program in the OS called a shell which checks, translates and messages them in various ways.
- The shell and the kernel work together, suppose a user types **rm myfile** (which has the effect of removing the file myfile). The shell searches the file store for the file containing the program **rm**, and then requests the kernel, through system calls, to execute the program **rm on myfile**. When the process **rm myfile** has finished running, the shell then returns the UNIX prompt **"$"** to the user, indicating that it is waiting for further commands.

# Basic Function of Kernel

1. Resource allocation
2. Process Management
3. Memory Management
4. I/O Device Management
5. Inter-Process Communication
6. Scheduling
7. System Calls and Interrupt Handling
8. Security or Protection Management

# Files and processes

- Everything in UNIX is either a file or a process.
- A process is an executing program identified by a unique PID (process identifier).

Examples of files:

+ a document (report, essay etc.)
+ the text of a program written in some high-level programming language
+ instructions comprehensible directly to the machine and incomprehensible to a casual user, for example, a collection of binary digits (an executable or binary file);
+ a directory, containing information about its contents, which may be a mixture of other directories (subdirectories) and ordinary files.

# Files and Directories

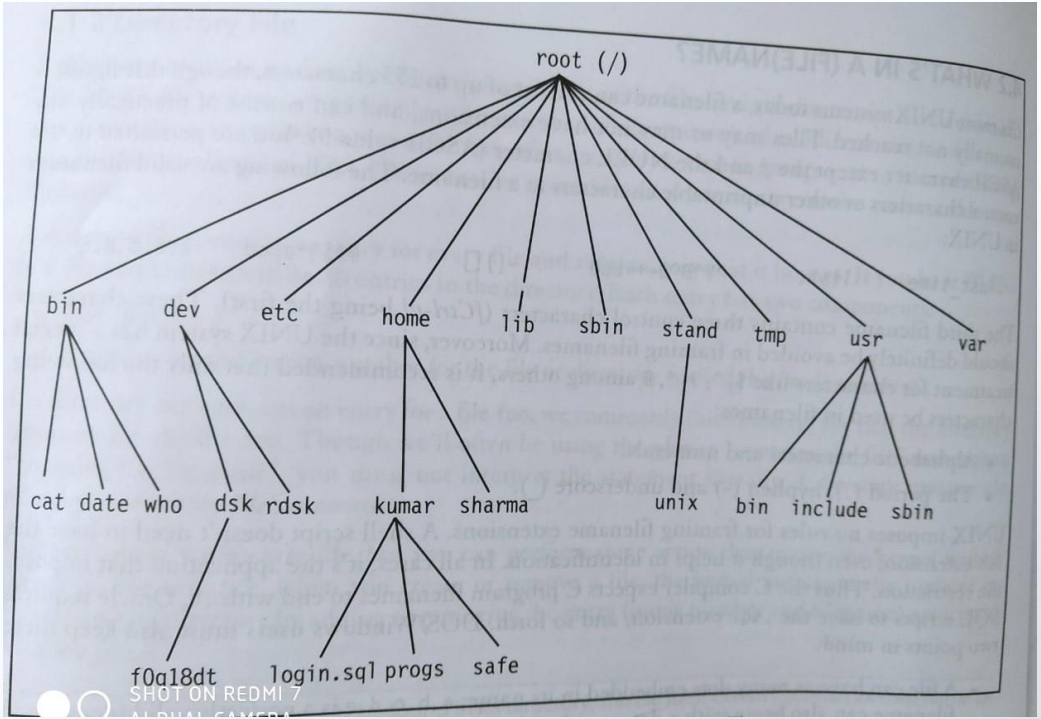
1. Ordinary files (files)
2. Directory files (directories)
3. Special files.

Files are containers for data. This data can be anything, including the text of a report, an image (picture) of a house, an executable program like a word processor, or any arbitrary data.

A special file is much like an ordinary file, and shares the same basic interface. However, special files are not stored in the file system because they represent input/output devices.

Each directory contains a number of files. A directory can contain other directories, be contained in another directory, or both

# File System Structure



# File System Structure

- The Unix file system is a hierarchical structure that allows users to store information by name. At the top of the hierarchy is the root directory, which always has the name /.
- The location of a file in the file system is called its path
  - All paths start from the root directory, so the path begins with /
  - We need to go to the home subdirectory, and the path becomes /home
  - In the home directory, we go into the rofel subdirectory. The path is now /home/rofel
  - In the rofel directory, we go into the bca subdirectory. The path is now /home/rofel/bca
  - The file quiz1.txt is in the bca directory. Appending the filename to the path, the path becomes /home/rofel/bca/quiz1.txt

**These special paths and their shorthand names are listed below:**

**/** → A single slash / specifies the root directory.

**~** →A tilde ~ specifies the current user's home directory.

**~user** →A tilde ~ followed by a user-id specifies the home directory of the given user. For example, at UBC, ~a1a1 is the home directory of the user who has the user-id a1a1.

**.** → This single dot . specifies the current working directory, that is, the directory that the user is currently in.

**..** →A pair of dots .. refers to the parent of the current working directory.

# Directory Structure

- The directory structure is hierarchical with the root directory (/)
- **/:** The slash / character alone denotes the root of the file system tree.
- **/bin and usr/bin:**These are the directories where all the commonly used unix commands (binaries) are found. PATH variable always shows these directories in its list.
- **/dev:** Stands for “devices”. Contains file representations of peripheral devices and pseudo-devices.
- **/etc:** Contains system configuration files and system databases. Your login name and password are stored in files /etc/passwd and /etc/shadow.
- **/home:** Contains the home directories for the users.
- **/var :** The variable part of the file system. Contain all your print jobs and your outgoing as well as incoming mail.
- **/lib:** Contains system libraries, and some critical files such as kernel modules or device drivers.
- **/root:** The home directory for the superuser “root” – that is, the system administrator.

- **/tmp:** A place for temporary files. Many systems clear this directory upon start up; it might have tmpfs mounted atop it, in which case its contents do not survive a reboot, or it might be explicitly cleared by a start up script at boot time.
- **/usr:** Originally the directory holding user home directories, its use has changed. It now holds executables, libraries, and shared resources that are not system critical, like the X Window System, KDE, Perl, etc. However, on some Unix systems, some user accounts may still have a home directory that is a direct subdirectory of /usr by a person).
- **/usr/include:** Stores the development headers used throughout the system. Header files are mostly used by the #include directive in C/C++ programming language.
- **/usr/lib:** Stores the required libraries and data files for programs stored within /usr or elsewhere.

**Absolute path:** It is defined as the specifying the location of a file or directory from the root directory(/). (cat /home/unix/rofel/bca/test1.txt)

**Relative path:** Relative path is defined as the path related to the present working directly(pwd). It starts at your current directory and **never starts with a /. (bca/test1.txt)**

# Processes

**A process is an executing command or program. The process is what actually performs the work of the command or program.**

+ **Foreground Process**

E.g: If you want to list all the files in your current directory, you can use the following command ,

**$ ls a*doc**

It would display all the files, which start with a and end with .doc

**a1.doc aa1.doc abc.doc a3.doc**

The process runs in the **foreground**, the output is directed to the screen. While a program is running in the foreground, no other commands can be run because the prompt would not be available until the program finishes processing and comes out.

## Background Process

The simplest way to start a background process is to add an ampersand (&) at the end of the command.

**$ls ch*.doc &**

In background mode the parent process continues to run using the keyboard and monitor as before, while the child process runs asynchronously.

## Job

A job consists of one or more processes working to perform a specific task. Each time you run a command or program on the system, you are starting a job.

# BASIC COMMMANDS

CAL : calender

DATE : show current date

ECHO : print output on screen

PRINTF : same as echo (print output on screen)

bc : basic calculator

CD : change directory

MKDIR : make directory

RMDIR : remove directory

who : who are the current users

uname : to know your machine's name

tty: knowing your terminal

HOME : to show home directory

pwd: present working directory.(current directory)

Cat : create and show the content of file

**(cat > a1 ,cat –v**(non printing character) **,cat –n** (numbering lines) **)**

(file name must be up to 255 characters long, it may or may not have any extension, it contain any character except / and null character, it is case sensitive [a1,A1])

(e.g .a1, a1., ^a1, -{}[], @#$ , a.b.c.d )

**ls (long listing directory contents)**

ls –x (output in multiple columns)

ls –F (identify directory and executable files) ls -Fx

ls –a (show hidden files) ls –axF

ls {directory name} (listing directory contents) ls –x rofel

ls –R (recursive listing) (subdirectories and files)

ls –r (sort filename in reverse order)

ls –d {directory name} list only directory name if that is available

ls –t (sort file name by last modification time) ls –lt

ls –u (sort file name by last access time) ls –lu

ls –lut

ls –i (display inode number)

**ls –l (basic file attributes) (listing file attribute) (total 7 attributes)**

_rw_r__r__ 1 kumar metal 19154 jun 10 12:45 chap01

- _rw_r__r__ : **File type and permission**
- 1 : **links**
- kumar : **ownership**
- metal : **Group ownership**
- 19154 : **File size**
- jun 10 12:45 : **Last modification time**
- chap01 : **File name**

## File ownership

When you create a file , your username show up in the third column of the file's listing so you are the owner of the file.

Your group name is seen in the fourth column, your group is the group owner of the file.

User –id (UID) – represent in name and numeric form

Group –id (GID) - represent in name and numeric form

} **/etc/passwd maintain UID and GID**

## File permission

ls –l command view the permission of files

_rw_r__r__ 1 kumar metal 19154 jun 10 12:45 chap01

**File type and permission**

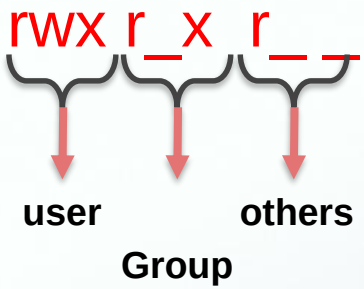
**File access rights**

_ rwx rwx rwx

**Represent ordinary file**

**9 characters represents category wise permission**

- Each group represent category and permission in three slots,
- r → read w→ write x → execute _ → absence of permission




- User can set different permission for the three categories of users i.e owner(user),group, others

cp (copying a file) cp chap1 chap2 (first dopied to second)

(cp ch1 ch2 ch 3 rofel) (cp ch* rofel)

cp -i (interactive copy) cp –i chap1 chap2

cp –R (copy directory structure (recursive copy) cp –R rofel rofelbca

rm (deleting one or more files) rm chap1 chap2 chap3 (rm chap*)

rm * (all files deleted) rm rofel/a1 rofel/a2

rm –i (interactive deletion) rm –i chap1

rm –r (R 0r r) (recursion deletion) rm –r*

rm –f (forcing removal) rm –rf *

mv (renaming (move)files or directory)

moves group of files to a different directory.

if destination file dosen't exist then create it.

doesn't prompt for overwriting.

mv ch1 chap1

mv ch1 ch2 ch3 ch4 rofel

mv rofel rofelbca (rename directory)

mv –R and mv –i

more (paging output)

more chap1 (press q for exit)

ls | more

lp (printing a file)

lp file1

lp -d {printer name} file1

lp -t “title here” file1

lp –n3 file 1 (no of copy)

file (knowing the file types)

file archive.zip

file *

wc (counting lines, words and characters)

wc file1

wc -l (lines) file1

wc –w (words) file1

wc -c (character) file1

wc chap1 chap2 chap3

cmp (comparing two files)

The two files are compare byte by byte , if two file are identical then cmp display no message. cmp chap1 chap2

comm (find common from two files)

It required two sorted files.

diff (find files differences)

display output in details.

## Compressing and Archiving the files

gzip and gunzip (.gz) compressing and decompressing files

tar (.tar) archival program

zip and unzip(.zip) compressing and archiving together

**gzip –** compress file with .gz extension and remove original file.

we can note the file before and after compression by using wc.

e.g gzip file1.txt → **gzip file1.txt.gz**

After compression

gzip –d  or gunzip (uncompressing a gzipped file)

it restore the original file.

gzip –d file1.txt

gunzip file1.txt.gz

gunzip file1.txt.gz file2.txt.gz

gzip –r (recursive compression)

gzip –r rofel (compress all file in rofel directory)

gzip –dr rofel (recursive uncompression)

tar : (archival the program)

tar –c (create an archive)

tar –x (extract files form archive)

tar –t (display files in archive)

**tar –c**

to create an archive specify name of the file , file name can be specify with the help of –f option.

to display the process while tar works use –v(verbose) option.

tar –cvf file_arch.tar file1.html file2.exe

New archive file  Files to create an archive

gzip and tar → (if archive is too large then create zip then archive)

gzip file_arch.tar (archived and compress) → file_arch.tar.gz

tar -x (extract files from archive)

tar –xvf file_arch.tar (extract two file)

tar -t(viewing the archive)

tar –tvf file_arch.tar (to show the list of archive files)

## zip and unzip (compressing and archiving together)

zip archive.zip file1.html file2.ps

Compress file name

Files to be archive

unzip archive.zip (files are restore with unzip command)

Compress file name

It doesn't overwrite with any existing file

zip –r (recursive compression) → **zip –r archive.zip .**

unzip –v (viewing the archive) → **unzip –v archive.zip**

It show the list of files with compressed and uncompressed size of each file along with percentage of compression.

## chmod (changing file permission)

- It used to set the permission of one or more files for all three categories of users. It can be run only by user(owner) and the super user.
- Command can be use by two ways

**Relative permission (by specifying the changes to the current permission)**

**Absolute permission (by specifying the final permission)**

**Relative permission :** in relative manner , chmod only changes the permission in command line and leaves the other permissions unchanged.

How to change file permission in relative manner.

**chmod category operation permission filename(s)**

**user,group,others (ugo)**

**Assign + Remove -**

**read, write , execute (rwx)**

e.g. 1. file execute by the user only.

chmod u+x test1

2. Files execute by all three category users.

chmod ugo+x test1

3. Give write permission to all category users.

chmod ugo+w test1 or chmod a+w test1 or chmod +x test1

4. To remove read permission from group and others.

chmod go-r test1

5. Assign and remove both together

chmod a-x , go+r test1.

**Absolute permission** - Use numbers to represent file permissions. When you change permissions by using the absolute mode then it represent by an octal mode number.

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

Only the current owner or superuser can use the **chmod** command to change file permissions on a file or directory.

Change permissions in absolute mode

Chmod ***nnn filename***

Specifies the octal values that represent the permissions for the file owner, file group, and others.

**E. g chmod 755 bca**

1. file execute by the user only and another have read permission only.

chmod 544 test1

2. Files execute by all three category users.

chmod 111 test1

3. Give write permission to group and other category users.

chmod 727 test1

4. To remove read permission from group and others.

chmod 733 test1

**Chown :** To change owner, change the user or group ownership of each given File to a new Owner. Chown can also change the ownership of a file to match the user or group of an existing reference file.

To change the ownership of multiple files or directories, specify them as a space-separated list.

**Syntax :** **CHOWN [OPTIONS] USER [:GROUP] FILE(s)**

USER is the user name or the user ID (UID) of the new owner. GROUP is the name of the new group or the group ID (GID).

**e.g. 1. chown bca21 chap1**

**2. chown bca11 chap2 bca**

# Vi Editor

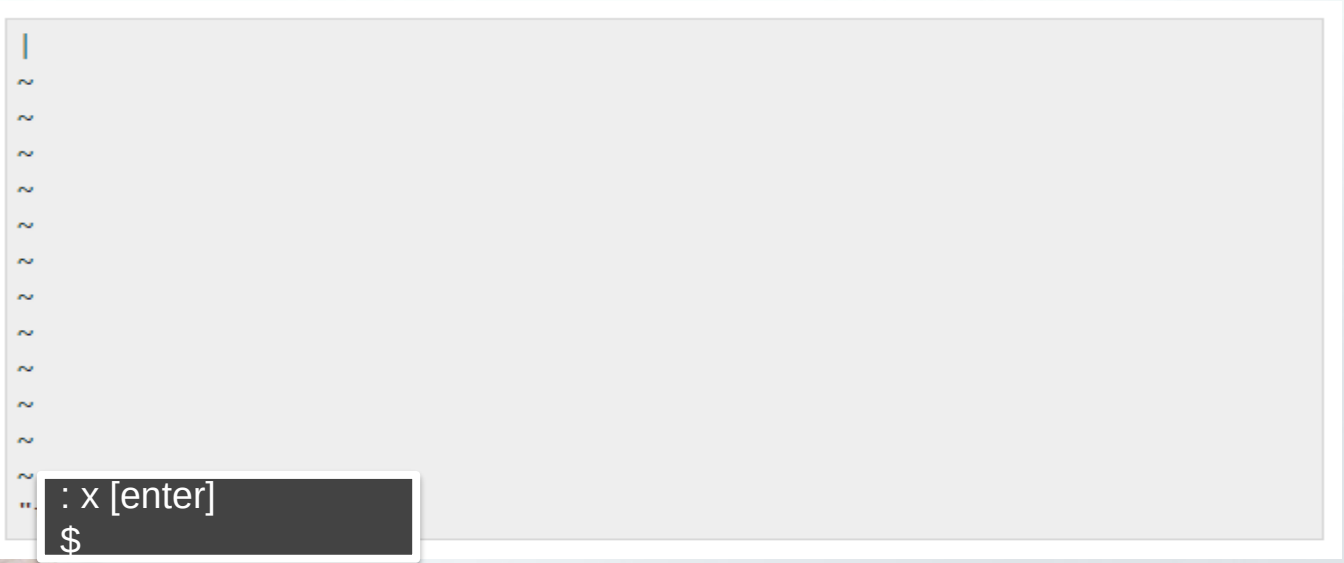
The default editor that comes with the UNIX operating system is called vi (visual editor). Using vi editor, we can edit an existing file or create a new file from scratch. we can also use this editor to just read a text file.

Syntax : **vi <filename>**

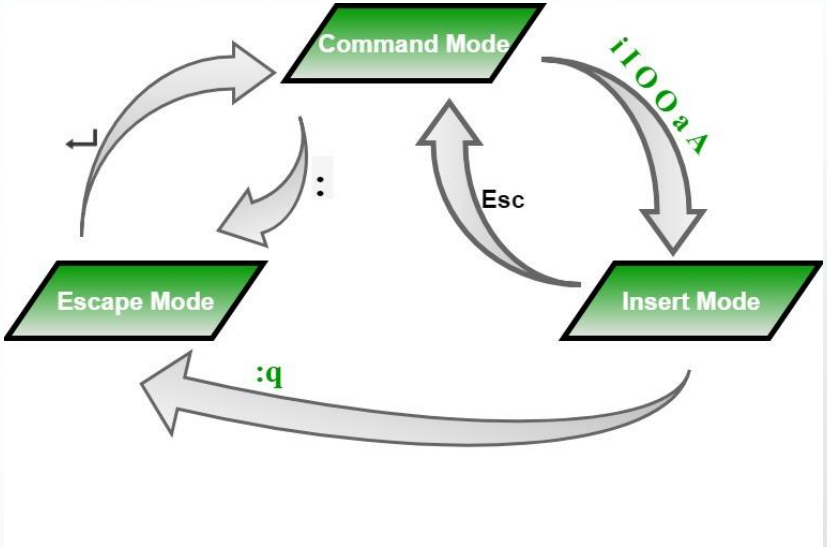
Creates a new file if it already does not exist, otherwise opens an existing file.

E.G $vi testfile

File will be open in vi editor , tilde (~) on each line following the cursor. A tilde represents an unused line. If a line does not begin with a tilde and appears to be blank, there is a space, tab, newline, or some other non-viewable character present.



# Modes of Operation in vi editor



**Command Mode:** The default mode of vi editor where every key is pressed is interpreted as a command to run on a text. this mode enables you to perform administrative tasks such as saving the files, executing the.  In this mode, whatever you type is interpreted as a command. To enter into Command Mode from any other mode, it requires pressing the **[Esc]** key. If we press [Esc] when we are already in Command Mode, then vi will beep or flash the screen.

**Insert mode:** This mode enables you to insert text into the file. Everything that's typed in this mode is interpreted as input and finally. The vi always starts in command mode. To enter text, you must be in insert mode. To come in insert mode you simply type **I** .To get out of insert mode, press the Esc key, which will put you back into command mode.

**Last Line Mode(Escape Mode):** Line Mode is invoked by typing a **colon [:],** while vi is in Command Mode. The cursor will jump to the last line of the screen and vi will wait for a command. This mode enables you to perform tasks such as saving files, executing commands.

## Input mode : entering and replacing a text

**i** : Inserts text to the left of the cursor.

**I** : Inserts text at beginning of current line.

**a** : Inserts text to the right of the cursor.

**A** : Inserts text at end of current line.

**o** : open a line below

**O** : open a line above

**r** : Replace single character under the cursor with the next character typed.

**R** : Replaces text from the cursor to right.

**s** : Replaces single character under the cursor with any number of characters.

**S** :Replaces entire line.

# Escape mode(last line mode)- saving text and quiting

| | |
|---|---|
| :w | Saves file and remains in editing mode |
| :x | Saves file and quits editing mode |
| :wq | As above |
| :w n2w.pl | Like *Save As* ..... in Microsoft Windows |
| :w! n2w.pl | As above, but overwrites existing file |
| :q | Quits editing mode when no changes are made to file |
| :q! | Quits editing mode but after abandoning changes |
| :*n1*,*n2*w build.sql | Writes lines *n1* to *n2* to file build.sql |
| :.w build.sql | Writes current line to file build.sql |
| :$w build.sql | Writes last line to file build.sql |
| :!*cmd* | Runs *cmd* command and returns to Command Mode |
| :sh | Escapes to UNIX shell |
| :recover | Recovers file from a crash |

## Navigation in vi editor

(Movement in the four direction h , j , k, l )

h→To move left. (20h moves the cursor at 20 character to the left)

l →To move right.

J→To move down

k→To move up (4k—move the cursor 4 line up)

Word Navigation (b, e w)

b – to move the cursor to the left one word at a time.

e- to move the cursor to the last character of the current word.

w- o move the cursor to the right one word at a time.

## (Movement in the line)

Press **^** to move the cursor to the start of the current line.

Press **$** to move the cursor to the end of the current line.

Use **0 or |** --- 30| moves cursoor t the column 30

## (Scrolling)

**Page Forward One Screen --**To scroll forward (move down) one screenful, press **Ctrl-F.**

**Scroll Forward One-Half Screen--**To scroll forward one half of a screen, press **Ctrl-D.**

**Page Backward One Screen --**To scroll backward (that is., move up) one screenful, press **Ctrl-B.**

**Scroll Backward One-Half Screen --**To scroll backward one half of a screen, press **Ctrl-U**

## (Absolute Movement)

ctrl+g →to show current line number.

40g → goes to line number 40

1g → goes to line number 1

g → goes to end of file.

## (Editing a text)

**Y→** copy **yy-**-Copies the current line. **10yy –** copy total 10 lines from the current line

**P →**Puts the copied text after the cursor.

**P →**Puts the yanked text before the cursor.

## (deleting and joining text)

Dd→Deletes the line the cursor is on

x→Deletes the character under the cursor location

X →Deletes the character before the cursor location

2x deletes two characters under the cursor location and 2dd deletes two lines at the current position.

J→Joins the current line with the next one. A count of j commands join many lines.

j→Joins the current line with the next one. A count joins that many lines

**U→**Restores the current line to the state it was in before the cursor entered the line.

**u→**This helps undo the last change that was done in the file. Typing 'u' again will re-do the change

## Pattern Matching and Wild card characters

***** matches **zero or more character(s)** in a file (or directory) name.

**?** It will match exactly one character.

**[ijk]** A single character either I, j or k

**[x-z]** a single character that is within the asci range of the characters x and z.

**[!ijk]** a single character that is not an I , j or k

**[!x-z]** a single character that is not within the asci range of the characters x and z.

## Examples :

**$ ls chap1 chap2 chap3 → ls chap*** (it will match all files and directory start with chap)

**$ echo* → (list of all files in current directory)**

**$ ls ???.txt → (match .txt file having at least 3 character)**

**$ ls .??? →(all hidden file names having at least 3 character after .(dot))**

**$ ls emp*.txt → (it will match all text files start with emp)**

**$ ls chap? → (it will display file name like , chap1, chapx , chap5, chapy )**

**$ cp * rofel → (copy all files from cuurent directory into rofel directory)**

**$ rm * →(all the files will be deleted from current directory)**

ls *.c → list all files with extension c

mv *.. /bin →moves all files to bin subdirectory of parent directory.

cp ????? Progs → copies to progs directory all files with 5 character long.

ls chap0[124]→list chap01, chap02 and chap04

ls chap0[1-4]→list chap01, chap02 , chap03 and chap04

ls chap0[!124] →list all file except chap01, chap02 and chap04

lp chap[0-1][0-9] → print all files from chap00, chap01… to chap19.

Ls chap[x-z]→ list chapx, chapy, and chap

*.[co]→mathches all file name with single character extension .c and .o

*.[!co]→mathches all file name with single character extension but not .c and .o

*.[!o]→mathches all file name with single character extension but not .o

[a-zA-z]*→ matching all file name beginning with an alphabets

[!a-zA-Z]*→matching all file name don’t begin with an alphabets

## Unix Environment Variables

An important Unix concept is the environment, which is defined by environment variables. A variable is a string to which we assign a value. The value assigned could be a number, text, filename, device, or any other type of data.

For example, first we set a variable **TEST** and then we access its value using the echo command: (environment variable used with $)

E.g. TEST="ROFEL BCA"

echo $TEST

ROFEL BCA

The environment variables are set without using the $ sign (TEST)but while accessing them we use the $ sign as prefix.($TEST).


## Environment Variable

### PATH

When you type any command on the command prompt, the shell has to locate the command before it can be executed.

The PATH variable specifies the locations in which the shell should look for commands. Usually the Path variable is set as follows,

**$PATH=/bin:/usr/bin**

**$**

### HOME

HOME Indicates the home directory of the current user.

### PWD

PWD Indicates the current working directory as set by the cd command.

### RANDOM

RANDOM Generates a random integer between 0 and 32,767 each time it is referenced.

**UID**

UID Expands to the numeric user ID of the current user, initialized at the shell start-up.

**E.G**

**$ echo $HOME**

**/root**

**$ echo $PATH**

**/usr/local/bin:/bin:/usr/bin:/home**

# Escaping and Quoting

**Escaping(\)** – providing a backslash(\) before the wild card to remove (escape) its special meaning.

Placing a \ immediately before a metacharacter turns off its special meaning.

**e.g** (*, here * is not interpreted as a wildcard characters.)

rm chap* → will remove all the file starting with chap but…

rm chap * → doesn't remove all the file starting with chap but remove chap file only.

**echo the new line character is \\n**


**Quoting(` `)** – there's another way to turn off the meaning of metacharacters. enclosing a wild card or even the entire pattern, within quotes (like `chap*`). Anything within this quotes are left alone by the shell and not interpreted.

e.g

echo '\' → display \

rm 'chap*' → removes file chap*

rm "my document .doc" → removes file my document.doc

# Redirection

It used in the shell to reassign the standard input and standard output of a command. We see command output and error message on the terminal (display), and we sometimes provide command input through the terminal(keyboard).

They perform all terminal related activity with the three files that the shell makes available to every command.

**0-standard input ==** the file representing input, which is connected to the keyboard.

**1-standard output ==** the file representing output, which is connected to the display.

**2-standard error ==** the file representing error message, which is connected to the display.

## Output Redirection (output redirection)

The '**>**' symbol is used for output (STDOUT) redirection.

e.g ls –l > list1

Here the output of command ls -l is re-directed to file "list1" instead of your screen. Then show the output by using → cat list1

If there is an existing file with the same name, the redirected command will delete the contents of that file and then it may be overwritten.

If you do not want a file to be overwritten but want to add more content to an existing file, then you should use **'>>'** operator.

All command display output on the terminal actually write to the standard output file not directly to the terminal.

The terminal , the default destination.

A file using the redirection symbol > and >>.

As input to another program using a pipeline(|).

You can redirect standard output, to not just files, but also devices!

$ cat music.mp3 > /dev/aud1



## Input redirection (standard input)

The **'<'** symbol is used for input(STDIN) redirection.

For e.g we can use cat and wc command to read a file but when they are use without argument then they read the file representing the standard input. It can represent three input sources,

The keyboard, default source.

A file using redirection with the < symbol

Another program using pipe (|) sign

e.g

wc < list1

In above example file name is missing so it will take a file name as list1.

## Standard Error

When you enter an incorrect command or try to open a non-existent file, certain diagnostic messages show up on the screen. This is the standard error stream whose default destination is the terminal.

e.g. cat a1

cat : cannot open a1

cat command fail to open file a1 and writes to standard error. You can redirect this error,

E.g cat a1 > errorfile (error redirect to errorfile)

cat errorfile

cat: cannot open a1

**Two Special Files**

1. /dev/null

File size is always zero. You may not save output in this file. It simply accepts any stream without growing in size. This facility is useful in redirecting error messages away from the terminal so they don't appear on the screen.

2. /dev/tty

This is a special file in the unix system. It is not the file that represents standard output or standard error. Commands generally don't write to this file but you will need to redirect some statements in shell script to this file.

## Pipes

Pipe is used to combine two or more commands, and the output of one command acts as input to another command, and this command's output may act as input to the next command and so on. It can also be visualized as a temporary connection between two or more commands/ programs/ processes. A pipe is a form of redirection (transfer of standard output to some other destination).

e.g cmd1 | cmd2 | cmd 3 (ls | more) who | wc –l

## tee

It is an external command. It saves one copy in a file and writes the other to standard output(terminal). **tee** can be placed anywhere in a pipeline.

e.g **ls –l** (display output of ls –l on terminal)

**ls –l | tee file1** (output can not be display on terminal but store in the file1)

**cat file1** (show the output from file1)

## Backquote (`)

The backquote ` is used for command substitution.

When the shell encounters a string between backquotes `cmd` it executes cmd and replaces the backquoted string with the standard output of cmd, with any trailing newlines deleted.

> **A backquote is not a quotation sign.** It has a very special meaning. Everything you type between backticks is evaluated (executed) by the shell before the main command and the *output* of that execution is used by that command,



## Command Substitution

Command substitution is the mechanism by which the shell performs a given set of commands and then substitutes their output in the place of the commands.

Command substitution is generally used to assign the output of a command to a variable.

```
DATE=`date`
echo "Date is $DATE"
```

```
Date is Thu Jul 2 03:59:57 MST 2009
```

## Process

A process is simply an instance of a running program.

A process is said to be **born** when the program starts execution and remains alive till the process is active.

After the execution the process is said to be **died**.

A process is also has a name (name of the program (cmd) is a process name)

The **kernel** is responsible for the management of a process.

Some attributes of every process are maintained by the kernel in memory in a separate structure called the **process table.**

Two important attribute of a process are,

1. The process id (PID)- each process is uniquely identified by a unique integer called the PID . That is allotted by the kernel when the process is born. PID is useful for kill the process.

2. The parent process id(PPID)- the **pid of the parent** is also available as a process attribute. when several processes have the same PPID.

**To know the PID of the current shell use $$ → echo $$**

**(291)**

## Parents and children process

### Parent Process

A parent process is one that creates a child process. A parent process may have multiple child processes but a child process only one parent process.

All the processes in operating system are created when a process executes the fork() system call except the startup process. The process that used the fork() system call is the parent process.

On the success of a fork() system call, the PID of the child process is returned to the parent process and 0 is returned to the child process.

On the failure of a fork() system call, -1 is returned to the parent process and a child process is not created.

## Child Process

A child process is a process created by a parent process in operating system using a fork() system call. A child process may also be called a **subprocess** or a subtask.

A child process is created as its parent process's copy and inherits most of its attributes. If a child process has no parent process, it was created directly by the kernel.

If a child process exits or is interrupted, then a SIGCHLD signal is send to the parent process. **e.g cat emp.lst**

A process representing the cat command , it started by the shell process. This cat process remain active as long as the command is active. **The shell is said to be the parent of cat , while cat is said to be the child of the shell.**

## Process Commands

ps : to know the process attributes

**$ps**

**PID TTY TIME CMD**

**18358 ttyp3 00:00:00 sh (it is a login shell of this user)**

**(it will show PID, terminal (TTY) , cumulative process time (TIME) and process name (CMD))**

**ps -f** → detailed listing which also show the parent of every process (- f → full)

| UID | PID | PPID | C | STIME | TTY | TIME | CMD |
|---|---|---|---|---|---|---|---|
| rofel | 6738 | 3662 | 0 | 10:23:03 | pts/6 | 0:00 | vi newfile |

**Where,**

UID – owner (user id) of every process

PID – Process ID

PPID – Parent Process id

C – CPU utilization of process

STIME – Time of Process started

**ps -u <username>** →displaying a process of user only. To know activity of any user.

$ps –u bca

**(PID TTY TIME CMD )**

**ps –a** → list the process of all users but doesn't display the system process.

(-a →all)

$ps –a

**(PID TTY TIME CMD )**

**ps –e or ps –A**

displaying all processes including system and user process both , apart form the user processes , a number of system processes keep running all the time.

**$ps –e**

| PID | TTY | TIME | CMD |
|---|---|---|---|
| 0 | ? | 0:01 | sched |
| 1 | ? | 0:00 | init |
| 2 | ? | 4:36 | fflush |
| 3 | pts/4 | 2:20 | vi |

System process are easily identified by **?** In the **TTY** column.

## Mechanism of Process Creation

There are 3 distinct phase in mechanism of process creation and uses 3 system calls: fork(), exec() and wait().

**fork():** A process in UNIX is created with the fork system call It Creates a child process. A new process is created because an existing process creates an exact copy of itself. This child process has the same environment as its parent but only the PID is different. This procedure is known as forking.

**exec():** forking create a process but it is not enough to run a new program so After forking the process, the address space of the child process is overwritten by the new process data. This is done through exec call to the system. No new process is created over here. The PID & PPID remains unchanged.

**wait():** The parent then executes wait system call to wait for the child process to complete its execution. It picks up the exit status of the child and then continue with its other function.

**(The important attributes that are inherited by the child process from its parents are: Real UID and GID, PGID, Nice value, Environment setting, Current working directory, memory segments etc.)**

## Internal Command

Internal commands are something which is built into the shell. For the shell built in commands, the execution speed is really high. It is because no process needs to be spawned for executing it.

**For example**, when using the "cd" command, no process is created. The current directory simply gets changed on executing it, echo

## External Command

External commands are not built into the shell. These are executables present in a separate file. When an external command has to be executed, a new process has to be spawned and the command gets executed.

**For example,** when you execute the "cat" command, which usually is at /usr/bin, the executable /usr/bin/cat gets executed , ls

## How the shell is created??

When the system moves to multiuser mode, **init** forks and exec **getty** for every active communication line. Every time **getty** print the login prompt on the respective terminal and then goes off to sleep.


- When a user attempt to log in , **getty** wakes up and **fork-exec** the login program to verify the login name and password entered.
- On the successful login **, login fork-exec** the process representing the login shell.

- **Init** goes off to sleep and waiting for the death of its children.
- When the user logs out , her shell is killed and the death is intimated to **init.**
- **Init** then wakes up and spawned another **getty** for the next **login.**

**Init** is a daemon process that continues running until the system is shut down.

## Process states and Zombies.

- A process after creation is in the **runnable state** before it actually run it is in **running state**.
- While the process is in **running** it may be invoke a disk I/O when it has nothing to do except wait for the I/O to complete.
- Then the process moves to **sleeping state** , and it may be awake when the I/O operaion is over.
- A process can also be **suspended** by pressing a key (ctrl+z). (Kill the process)

## Zombie

- The process whose parent don’t wait for their death move to the **zombie state.**
- When a **process dies**, it immediately moves to **zombie state**. It remain in this state until the parent picks up the child’s exits status from the process table. After that kernel free the process table entry.
- A **zombie is a harmless dead** child but you can’t kill it.
- It also possible for the parent itself to die before the child dies.
- The child then become an **orphan** and kernel make init the parent of all orphan.
- A process which has finished the execution but still has entry in the process table to report to its parent process is known as a **zombie process.**

## Running jobs (process) in the background

- In a multitasking os user perform more than one job at a time.
- So there can be only one job in the foreground and rest of the jobs have to run in background.

There are two different way to run a process in background ,

1. & operator
2. nohup command

## &: no logging out

**“&”** is a shell operator used to run a process in the background. In this case parent doesn’t wait for the child’s death. Just terminate the command line with an **&** and the command will run in the background.

e.g $ sort emp.lst

emp.lst will be sort and display output on terminal

e.g $ sort emp.lst &

519

$ _

**Shell return PID of the invoked command and the prompt is return so the shell is ready to accept the another command. but the previous command has not been terminated yet.**

## nohup : log out safely

- When user logs out , all job stop their working because their shell is killed.
- When the parent is killed, its children are also normally killed.
- But **nohup command** , prefix with any command , it permits execution of the process even after the user has logged out. But you must use **&** sign with it.

**e.g** **$ nohup sort emp.lst &**

**586 ← PID** (sending output to nohup.out file)

**So you can safely logout of the system without aborting the command**

# nice (job execution with low priority)

- In unix OS all processes are usually executed with equall priority if there is a high-priority process or low priority process.
- But nice command is used with & operator to reduce the priority of jobs. More important jobs can have greater access to the system resource.
- Nice command reduce priority of any command.
- To run a job with a low priority, the command name should be prefix with nice,

**nice wc -l emp.lst** **or** **nice wc –l emp.lst**

- nice is a built in command , its values are system dependent and range from 1 to 19.
- A higher nice value implies a lower priority.

**nice –n**

-n option is used to increase priority value.

**e.g $ nice –n 5 wc –l emp.lst & (nice value increased by 5 units)**

**To show the nice value use ps –o nice command.**

# kill (premature termination of a process)

- The kill command is used for terminating one or more processes.
- It is an internal  command.
- By default the kill command will send a TERM signal to a process allowing the process to perform any cleanup operations before shutting down.
- The kill command is used primarily to terminate or restart processes.
- The command use one more PID as its arguments.
- To kill a process first find out the process identifier number or PID of the process to be killed, then pass the PID number to the kill command.

**e.g $ kill 105**

**PID**

This command terminate the job(process) having the **PID 105**. it is premature termination.

If you don't remember PID then use ps command then use kill command .

**$ps**

**PID TTY TIME CMD**

**18358 ttyp3 00:00:00 sh (it is a login shell of this user)**

**$ kill 18358**

**$ kill 105 121 1211 123**

User can kill more than one process at a time. Just specify all their **PID** with kill command

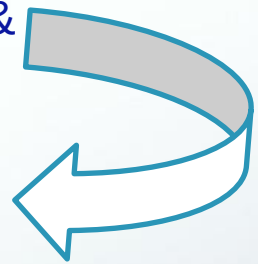
If all these process have the same parent, you may simply kill the parent then all its children gets kill.



# Killing the last background job ($!)

- The system variable $! Store the PID of the last background job.
- So you can kill the last background job without using ps to find out its PID.
- To know the PID of current shell →$$
- To show last backgroung job use & at the end of any command.

e.g $ sort –o emp.lst &



345

$ kill $! → Kill the sort command

# Job control

- at and batch command (execute later → scheduling a job)
- Unix provide a facility to schedule a job to run at a specific time of day.

**at : one time execution**

- It will take an argument as time for the job to be executed and display the at > prompt.
- Input has to be supplied from the standard input.

**e.g $ at 11:00**

at > program1.sh

[ctrl+d]

The particular job goes to the queue and at 11:00 a.m today the script program1.sh will be excuted.
at show the job number and date and time of schedule execution.

- at command specify the key word , now, noon, midnight and tomorrow.
- It use + symbol to act as an operator.
- That word can be used with this operator include hours, day, week, months and years.

at 15 -------24 hour format

at 5pm

at 3:08pm

at noon ------- at 12 hrs. today

at now +1 year ----- at current time after one year

at 3:08pm +1 day -----at 3:08 pm tomorrow

at 15:08 December 20, 2020

at 9pm tomorrow

at –l job can be listed
at –r jobs can be remove



## batch : execute in batch queue

Batch command also schedules jobs for later execution,

but unlike at, jobs are executed as soon as the system load permits.

This command doesn't take any arguments.

**e.g $ batch < program1.sh**

Command will be executed using /usr/bin/bash
Job 104185.b at sun sep 13 12:00 2020

## crontab - Running a job periodically

- at and batch – for one time execution.
- Cron executes program at regular intervals.
- It is mostly dormant , but every minute it wakes up and looks in a control (crontab) file to be perform some instruction.
- Each time contains a set of six fields seprated by white space.

**00-10 17 * 3,6,9,12 5 sort emp.lst > newsort**

## 00-10 17 * 3,6,9,12 5 sort emp.lst > newsort

- **First field (00-59 values**) specifies the number of minutes after the hour when the command is to be executed . The range 00-10 schedules execution every minute in the first 10 minutes of the hour.
- **The second field (17 i.e. 5 pm)** indicates the hour in 24 hour format for scheduling.(1 to 24)
- **The third field (1 to 31**) control the day of the month. (***** means the command is to be executed every minute) for the first 10 min. starting at 5 p.m everyday.
- **The fourth field (3,6,9,12**) specifies the month (1 to 12)
- **The fifth field (5-Friday**) indicate the day of the week. (i.e 0 to 6 – Sunday having 0)

- So sort command will be executed every minute in first 10 min after 5 p.m, every Friday of the month march, June , September and December (of everyday)

- crontab creating a crontab file which contain cron command.
- cron is mainly used by the system administrator to perform housekeeping chores, like removing outdated files or collecting data on system performance.

## File systems and Inodes

- In unix files are organized in a hierarchical way.
- Unix file hierarchy as a “file system” i.e all the files and directory are held together in a big superstructure i.e **root**.
- Every file system has a directory structure headed by root. Every file is associated with a table that contain all the information about a file – except its name and content.
- That table is called **inode(index node)** and it is access by **inode number.**
- It’s a directory that stores the inode number along with the file name.
- When you use command with a file name as an argument , the kernel first locates the inode number of the file from directory then read the inode to fetch data relevant to the file.
- For showing all inode information use **ls -i(inode).**

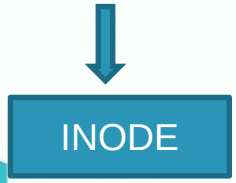
# inode contains the following attributes of a file:

- File type(regular , directory, device)
- File permission (rwx, rwx ,rwx for UGO)
- Number of links
- UID of owner
- GID of group owner
- File size in bytes
- Date and Time of last modification
- Date and time of last access
- Date and time of last change of the inode
- An array of pointer that keep track of all disk blocks used by the file.

**For e.g**

**$ ls -il emp.lst**

**9059 -rw-r- - r- - 1 bca rofel 51813 sep 11 10:15 emp.lst**



# Link

- A link in UNIX is a pointer to a file.
- Like pointers in any programming languages, links in UNIX are pointers pointing to a file or a directory.
- Creating links is a kind of shortcuts to access a file. Links allow more than one file name to refer to the same file.
- A directory entry can have an Inode pointing to another file.

There are two types of links :

- Hard Links
- Soft Link or Symbolic links

These links behave differently when the source of the link (what is being linked to) is moved or removed.

**HARD LINK**

- Hard links always refer to the source, even if moved or removed. When a hard link is made, then the i-numbers of two different directory file entries point to the **same inode**.
- Each hard linked file is assigned the same Inode value as the original, therefore they reference the same physical file location. Hard links more flexible and remain linked even if the original or linked files are moved throughout the file system.
- ls -l command shows all the links in the link column .
- Links have actual file contents
- Removing any link, just reduces the link count, but doesn't affect other links.
- We cannot create a hard link for a directory to avoid recursive loops.
- If original file is removed then the link will still show the content of the file.

## Symbolic link

- Symbolic or a soft link is a special type of file containing links or references to another file or directory in the form of a path. The path may be relative or absolute. Symbolic links are not updated.
- A soft link is similar to the file shortcut feature which is used in Windows Operating systems. Each soft linked file contains a separate Inode value that points to the original file. As similar to hard links, any changes to the data in either file is reflected in the other. Soft links can be linked across different file systems, although if the original file is deleted or moved, the soft linked file will not work correctly (called hanging link).
- Soft Link contains the path for original file and not the contents.
- Removing soft link doesn't affect anything but removing original file, the link becomes "dangling" link which points to nonexistent file.
- A soft link can link to a directory.
- Link can be confirm by using ls –li (inode --- same inode number for link file)

# ln (creating a link(hard and soft))

- File is linked with ln command.
- Which take two file name as arguments.

**e.g $ ln emp.lst employee** ---employee file must does not exist

- With using ls –li command to show they both have same inode number. (link)

**$ ls –li emp.lst employee.**

29518 -rwxr-xr-x 2 bca rofel 512 sep 11 9:50:40 emp.lst
29518 -rwxr-xr-x 2 bca rofel 512 sep 11 9:50:40 emp.lst

- You can link multiple files, but then destination filename must be a directory.

**e.g $ ln chap* bca_files** (bca_files is a directory)

To remove a link use rm command

**$ rm emp.lst employee**

**ln –s (symbolic link)**

**$ ln –s emp.lst emp.sym**

To check symbolic link use → **$ ls –li emp.lst emp.sym**

```
29518 -rwxr-xr-x  1 bca rofel  512  sep 11 9:50:40 emp.lst
29518 1rwxr-xr-x  1 bca rofel  512  sep 11 9:50:40 emp.sym→emp.lst
```

# umask (show default file and directory permission)

umask is a number which defines the default permissions which are not to be given on a file. A umask of 022 means not to give the write permission to the group(022) and others(022) by default. "user file-creation mode mask"

**Following are default permission for all files and directories**.

- rw- rw- rw-  (octal 666) for regular file
- rwx rwx rwx (octal 777) for directories
- Defaut umask value is 022

The umask specifies the permissions you do not want given by default to newly created files and directories.

umask works by doing a bitwise AND with the bitwise complement of the umask. Bits that are set in the umask correspond to permissions that are not automatically assigned to newly created files.

**e.g.** we create a file say "file1". The permissions given for this file will be the result coming from the substraction of the umask from the default value ,

Default: 666
umask : 022
---
Result : 644

644 is the permission to be given on the file "file1".

644 means read and write for the User(644), read only for the group(644) and others(644).

## Unix three time stamps

ls –l --- Time of last file modification

ls –lu --- Time of last access

ls –lc --- Time of last inode modification

ls –lt --- listing in order of their modification time

ls –lut --- listing in order of their access time

# touch : (changing the time stamp)

- It is used to create, change and modify timestamps of a file. Basically, there are two different commands to create a file ,
- **cat** : It is used to create the file with content.
- **touch**: It is used to create a file without any content.

The file created using touch command is empty. This command can be used when the user doesn't have data to store at the time of file creation.

touch command can be used to create the multiple numbers of files at the same time but these files would be empty while creation.

touch command creates new file if it does't exist , otherwise it just change the time stamp of the file

e.g touch options expression filename(s)

$ touch emp.lst → without option and expression , both time are set to current time.

$ touch 09151050 emp.lst → expression consist of 8 digit number using format (MMDDhhmm—month,day,hour,minute)

touch –a (change only access time)

$ touch –a 08201150 emp.lst

touch –m (change only modification time)

$ touch –m 06121020 emp.lst

**Check the output of this command use ls –l or ls –lu or ls -lt**

# Find (locating a file)

- The Unix find command is a powerful utility to search for files or directories.
- The search can be based on different criteria, and the matching files can be run through defined actions. This command recursively descends the file hierarchy for each specified path name.
- The find command in UNIX is a command line utility for walking a file hierarchy. It can be used to find files and directories and perform subsequent operations on them.
- It supports searching by file, folder, name, creation date, modification date, owner and permissions. By using the - exec other UNIX commands can be executed on files or folders found.

***Syntax: find path_list selection_criteria action***

## How find command works,

- First it recursively examines all files in the directories specified in path_list.
- Then matches each file for one or more selection_criteria.
- Finally, it take some action on those selected files.

Example.

**$ find / -name emp.lst -print**

- ( / )path list indicates that the search should start from root directory.
- ( -name emp.lst ) selection criteria , consist of an expression in form of –Operator. It matches the file that has a name emp.lst . (. Is also used)
- (-print) the third argument specifies action –print to be taken on the file, (display on terminal)

**$ find . –name “*.c” –print (all file with the extension of c)**

find used to match group of file names With relative pathnames.

**$ find . –name ‘[A-Z]*’ –print (all file whose name sbegin with an uppercase letter)**

**Selection criteria options:**

1. -inum (locating a file by inode number)

**$ find / -inum 13975 -print**

find all filenames that have the same(13975) inode number.

2. -type and –perm (locating a file by file type and permission)

-type option follow by f , d and l i.e selected files may be ordinary (f), directory(d) and symbolic link(l).

**$ find . -type d -print (locate all directory of your current directory)**

-perm option specifies the permission to match. i.e in octal number like (666,777, 644)

**$ find $HOME -perm 777 –type d -print**

it select file having read , write and execute permission for all category of users only for directories.

3. –mtime and –atime (file's modification and access time – find unused file)

**$ find . –mtime -2 -print**

-2 means less than 2 days , it will select files from the current directory that have been modified in less than 2 days.

Here negative value (-) is used with mtime and positive value used with atime(+)

**$ find /home -atime +365 –print**

it will select file from the home directory that have not been access for more than a year.

**→Find operators (! (not) , -o (or) and –a (and))**

**$ find . ! –name “*.c” -print**

it will find all the files from current directory except .c extension file.

**$ find /home \( -name “*.sh” -o -name “*.pl” \) -print**

it will find files with extension .sh as well as extension .pl both

**$ find $HOME \(-perm 777 -a -type d \) -print**

it will find directories with permission read , write and excutes of all three groups.

## Find Action

**-print** → prints (display on terminal ) selected files on standard output.

**-ls** → executes ls –lids command on selected files.

```
$ find  .type f –mtime +2  ls
```

**-exec cmd** → executes unix command cmd followed by { } \.

```
$ find $HOME –type f – atime +365 –exec rm {} \
```

# Simple filter Commands:

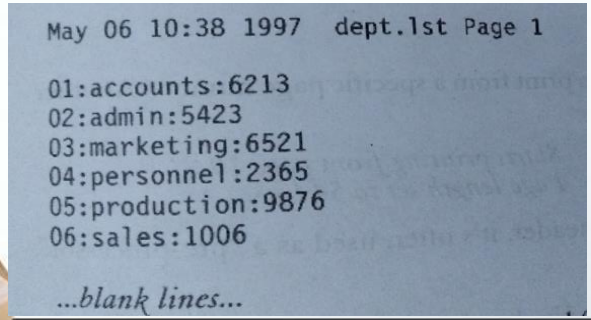
## pr : (paginating files)

The pr command prepares a text file for printing.

pr adds headers that include the filename, date and time, and the page number.

It is use file name as an argument.

```
$ pr   dept.lst
```




**pr command add  lines of margine at the top and five at the bottom.**

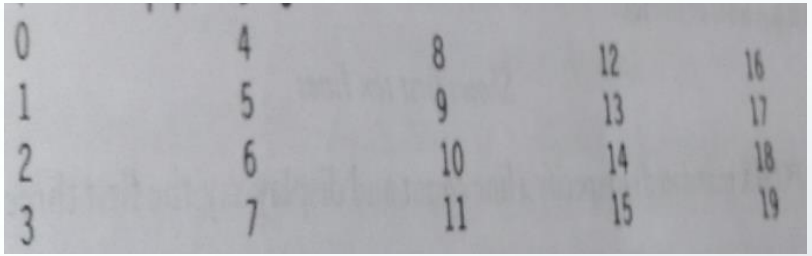
## pr options

**pr –*k*** (k is an integer) – print output in k columns. pr obtain its input from the standard output of another program.

**$ pr -5 file 1 → output is display in 5 columns.**

**-t →** to suppress header and footer.

**$ file 1| pr –t -5**



**-h** → you can set header with your choice.

**-d** → double space input, reduce clutter.

**-n** → number lines

**-o n** → set lines by n spaces, increases left margin of page.

**$ pr –t -n -d -o 10 dept.lst**

**1 01:accounts:6213**

**2 02:admin:5423**

**3 03:marketing:6521**

← OUTPUT

**Examples:**

**$ pr +10 chap1** → start printing from page number 10

**$ pr -l 54 chap1** → -l is page length. Which is set to 54.

**$ pr -h "Department List " dept.lst** → set header as user's choice

# head – displaying the beginning of a file.

head command display the top of the file. When use without an option , it display the first 10 lines of the file.

$ head  emp.lst  -- show first 10 lines from file.

**head –n** (-n specify a line count and display.)

$ head –n 5  emp.lst

or

$ head -5 emp.lst

} both command give same output, it display first 5 lines from file

E.G display last modified file . → ls –lt | head -1

display last 5 modified file. → ls -lt | head -5

# tail – displaying the end of the file

It is the complementary of head command .The tail command, as the name implies, print the last N number of data of the given input. By default it prints the last 10 lines of the specified files. If more than one file name is provided then data from each file is precedes by its file name.

$ tail  emp.lst  → show last 10 lines from file.

**tail –n** (-n specify a line count and display.)

$ tail –n 5  emp.lst

or

$ tail -5 emp.lst

} both command give same output, it display last 5 lines from file

***+count*** **(+5)** → it allows you to represent the line number from where the selection should begin.

$  tail  +11 emp.lst  → from 11th line onwards print all lines till end.

**tail –c → extracting bytes rather than lines**

-c option follow by a positive or negative integer depends on weather the extraction is perform relative to a beginning or end of the file.

$ tail – 25 a1.txt → show only last 25 characters from a1.txt file.

$ tail +25 a1.txt → show everything after skipping 25 characters.



# cut – slitting a file vertically

Extract columns and fields from file by using cut command. (use for fixed length line)

cut –c (cutting columns)

To extract specific columns you need to follow –c option with a list of column numbers, delimited by a comma.

Range can be specified by hyphen (-).

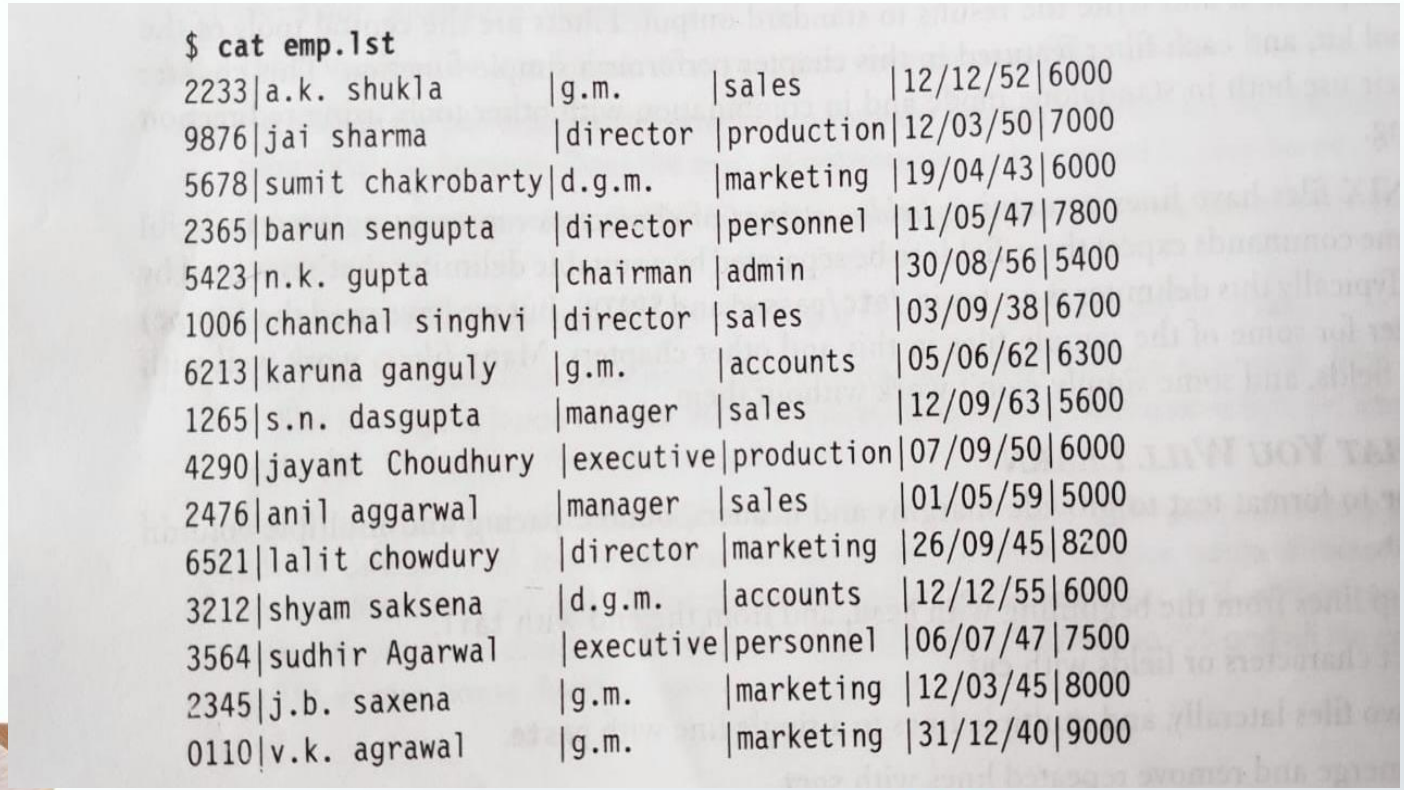
Example: $ cut -c 5 emp.lst --- only 5th column display

$ cut -c 5, 7 , 9 emp.lst -----only 5th ,7th and 9th column display

$ cut –c 6-22 , 24-32 emp.lst - range

$ cut –c 50- emp.lst ---display column no. 50 to end of line.

$ cut -c -5 empl.lst --- same as 1-5



```
$ cat emp.lst
2233|a.k. shukla       |g.m.      |sales      |12/12/52|6000
9876|jai sharma        |director  |production |12/03/50|7000
5678|sumit chakrobarty|d.g.m.    |marketing  |19/04/43|6000
2365|barun sengupta    |director  |personnel  |11/05/47|7800
5423|n.k. gupta        |chairman  |admin      |30/08/56|5400
1006|chanchal singhvi  |director  |sales      |03/09/38|6700
6213|karuna ganguly    |g.m.      |accounts   |05/06/62|6300
1265|s.n. dasgupta     |manager   |sales      |12/09/63|5600
4290|jayant Choudhury  |executive |production |07/09/50|6000
2476|anil aggarwal     |manager   |sales      |01/05/59|5000
6521|lalit chowdury    |director  |marketing  |26/09/45|8200
3212|shyam saksena     |d.g.m.    |accounts   |12/12/55|6000
3564|sudhir Agarwal    |executive |personnel  |06/07/47|7500
2345|j.b. saxena       |g.m.      |marketing  |12/03/45|8000
0110|v.k. agrawal      |g.m.      |marketing  |31/12/40|9000
```

## cut –f (cutting fields)

To extract the useful information you need to cut by fields rather than columns. List of the fields number specified must be separated by comma or pipes. Ranges are not described with -f option. cut uses tab as a default field delimiter but can also work with other delimiter by using -d option.

**Note: Space is not considered as delimiter in UNIX**.

**Example :** $ cut –d “|” -f 2,3,5 emp.lst --cut whole field 2nd , 3rd and 5th fields.

$ cut –d \| -f 2,3,5 emp.lst --cut whole field 2nd , 3rd and 5th fields.

$ cut –d “|” -f 2,3,5 emp.lst > cutlist1

$ cut –d “|” -f 4,5,6 emp.lst | tee cutlist2

$ who | cut –d” “ –f 1 --- cut user list from who output.

# Paste (pasting files)

Paste command is one of the useful commands in Unix. It is used to **join files horizontally** by outputting lines consisting of lines from each file specified, separated by tab as delimiter, to the standard output.

Examples: **$ paste cutlist1 cutlist2 ---join two file horizontally.**

```
a.k. shukla        |g.m.       2233|sales      |12/12/52|6000
jai sharma         |director   9876|production |12/03/50|7000
sumit chakrobarty  |d.g.m.     5678|marketing  |19/04/43|6000
barun sengupta     |director   2365|personnel  |11/05/47|7800
n.k. gupta         |chairman   5423|admin      |30/08/56|5400
```

**$ paste –d" |" cutlist1 cutlist 2 -- join two file horizontally, with using delimeter "|"**

```
a.k. shukla        |g.m.       |2233|sales      |12/12/52|6000
jai sharma         |director   |9876|production |12/03/50|7000
sumit chakrobarty  |d.g.m.     |5678|marketing  |19/04/43|6000
barun sengupta     |director   |2365|personnel  |11/05/47|7800
n.k. gupta         |chairman   |5423|admin      |30/08/56|5400
```

$ cut –d “|” –f 2 ,3 emp.lst | paste -d “|” cutlist 1 –

$ cut –d”|” –f 1,4 emp.lst | paste –d ”| “ - cutlist1

--- cutting and join together

```
a.k. shukla       |g.m.      |2233|sales      |12/12/52|6000
jai sharma        |director  |9876|production|12/03/50|7000
```

**paste –s (join lines)**

Example: paste -s –d “| |\n” address book

```
anup kumar|anup_k@yahoo.com|24569083
vinod sharma|vinod_sharma@hotmail.com|34586532
madhuri_bahl|madhuri@heavens.com|39034943
```

```
$ cat addressbook
anup kumar
anup_k@yahoo.com
24569083
vinod sharma
vinod_sharma@hotmail.com
34586532
madhuri_bahl
madhuri@heavens.com
39034943
```

## sort : (ordering a file)

- SORT command is used to sort a file, arranging the records in a particular order. By default, the sort command sorts file assuming the contents are ASCII. Using options in sort command, it can also be used to sort numerically.
- SORT command sorts the contents of a text file, line by line.
- By default entire line is sorted.
- It supports sorting alphabetically, in reverse order, by number, by month and can also remove duplicates.
- The sort command can also sort by items not at the beginning of the line, ignore case sensitivity and return whether a file is sorted or not. Sorting is done based on one or more sort keys extracted from each line of input.
- Blank space is the default field separator.

## Example:

**$ sort emp.lst**

It reorders a lines in ASCII collating sequence –whitespace , numerals, uppercase letters and lowercase letters.

We can use **–t** option as a delimeter.

## Options:

**sort -k : sorting on primary key.**

**$ sort –t"|" –k 2 dept.lst --- sorting perform on second filed.**

**Sort order can be reversed with –r option.**

**$ sort -t"|" -r -k 2 dept.lst --- descending order**

**$ sort –t "|" –k 2r dept.lst**

**sorting on secondary key** **(sorting can perform on more than one key)**

**$ sort –t"|" –k 3,3 -k 2,2 dept.lst**

sorting can perform first on 3rd column (i.e starting with 3 and ending with 3)

**Sorting on columns** (-k option use with m.n i.e mth filed and nth column)

**$ sort –t"|" –k 5.6 , 5.8 dept.lst**

sorting can be start from 6th column of 5th field and end from 8th column on 5th field)

**Numeric sort**

**$ sort –n numfile**

**Removing repeated lines**

**sort –u** (unique , it remove repeated lines form a file)

**$ sort –t "|" –u dept.lst**

**Sort output redirect to the file (sort –o)** specify output file name

$ sort –o  sort_dept  –k  3 dept.lst

Sort Output store
in sort_dept file

$ sort  –o dept.lst  dept.lst  -- output stored in same file

**check weather the file has actually sorted or not. (-c)**

$ sort –c  sort_dept

$ _  ---file is sorted

## Sort with multiple file name as an arguments (-m)

**$ sort –m file1 file2 file3**

Table 12.1 **sort** Options

| *Option* | *Description* |
|---|---|
| -t*char* | Uses delimiter *char* to identify fields |
| -k *n* | Sorts on *n*th field |
| -k *m*,*n* | Starts sort on *m*th field and ends sort on *n*th field |
| -k *m*.*n* | Starts sort on *n*th column of *m*th field |
| -u | Removes repeated lines |
| -n | Sorts numerically |
| -r | Reverses sort order |
| -f | Folds lowercase to equivalent uppercase (case-insensitive sort) |
| -m *list* | Merges sorted files in *list* |
| -c | Checks if file is sorted |
| -o *flname* | Places output in file *flname* |

# uniq : locate repeated and nonrepeated lines

uniq command filters out the matching lines from the input and writes the filtered data to the output file. It display unique lines.

ungiq requires a sorted file as input.

Example : $ cat dept.lst

```
01|accounts|6213
01|accounts|6213
02|admin|5423
03|marketing|6521
03|marketing|6521
03|marketing|6521
04|personnel|2365
05|production|9876
06|sales|1006
```

$ uniq dept.lst

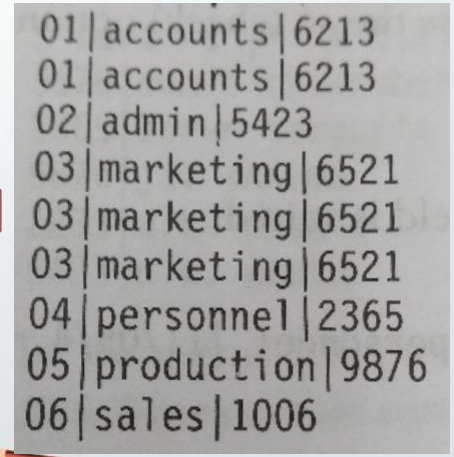
```
01|accounts|6213
02|admin|5423
03|marketing|6521
04|personnel|2365
05|production|9876
06|sales|1006
```

uniq required sorted file so the procedure is to sort the file first and then use uniq command.

**$ sort dept.lst | uniq**

**Options :**

**sort –u** -- selecting a nonrepeated lines. It select only lines that are not repeated. **$ sort dept.lst | uniq –u**




02| admin|5423

06| sales |1006

04|personnel | 2365

05|production|9876

**$ cut –d"|" –f 2 dept.lst | sort | uniq –u**

admin sales personnel production

**uniq –d** -- selecting a duplicate lines , select only one copy of repeated lines.

```
01|accounts|6213
01|accounts|6213
02|admin|5423
03|marketing|6521
03|marketing|6521
03|marketing|6521
04|personnel|2365
05|production|9876
06|sales|1006
```

$ cut –d"|" –f 2 dept.lst | sort | uniq –d

accounts

marketing

**uniq –c** ---count frequency of occurrence of all lines along with the lines. $ cut –d"|" –f 2 dept.lst | sort | uniq –c

2 accounts

1 admin

3 marketing

1 personnel

1 production

1 sales

## tr – translating characters

The tr command in UNIX is a command line utility for translating or deleting characters.

It will filter manipulate individual characters in a line.

tr command take input only from standard input, it doesn't take a file name as argument.

It translate each character in expression1 with expression2 .

The first character in the first expression is replace with the first character in the second expression.

syntax : **tr options expression1 expression2 standard input**

**e.g tr '|' '~' < emp.lst -- replace | with ~ in the file**

- Length of the two expression should be equal.
- You can define the two expression as two separate variable and then evaluate them.

Example:

```
expr1='|'      ;    expr2='~'
tr  "$expr1"  "$expr2"  < emp.lst
```

Change case of text

```
$ head –n 3 emp.lst  |  tr   '[a-z]'   '[A-Z]'
```

-- change the case of first three lines from upper to lower.

Translate character using asci octal values

```
$  tr  '|' '\012'  <  emp.lst
```

-- 012 is a new line character that is \n

## tr options

tr –d : deleting the characters

$ tr –d '|\' < emp.lst | head –n 3 -- it will delete | \ from file

tr -s :Compressing multiple consecutive characters . (squeeze)

$ tr –s ' ' < emp.lst --remove multiple spaces, set one space

tr -c : complementing values of expression (complement)

$ tr –cd '|/' < emp.lst -- delete all the characters except | and / with combine –cd


## Advance filter commands  using regular expression:

### grep :

grep scans its input for a pattern and displays lines containing the pattern, the line number or file name where the pattern occurs. The grep filter searches a file for a particular pattern of characters, and displays all lines that contain that pattern. The pattern that is searched in the file is referred to as the regular expression.

### sed: Stream Editor

It can perform lot's of function on file like, searching, find and replace, insertion or deletion. Though most common use of SED command in UNIX is for substitution or for find and replace.

SED command in unix supports regular expression which allows it perform complex pattern matching.

## grep :searching for a pattern

Syntax: grep options pattern filename(s)

grep scan its input for a pattern and displays lines containing the pattern, the line number or filenames where the pattern occurs.

grep searches for pattern in one or more filenames , or the standard input if no file name is specified.

Example: $ grep “sales” emp.lst

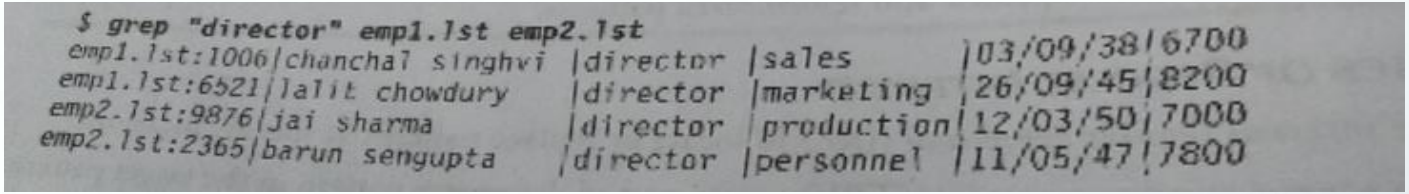
-- display line containing the string sales from the file emp.lst

grep is a filter , it can search its standard input for the pattern , and also save the standard output in a file.

$ who | grep rofel > a1.out

grep is used with multiple file names,

$ grep “director”  emp1.lst  emp2.lst  emp3.lst




$ grep ‘jai sharma’  emp.lst

-- quoting is necessary when the pattern contain multiple words.

grep –i : ignoring the case (when you are not sure of the case for searching pattern)

$ grep –i ‘manager’ emp.lst

it will locate the pattern (Manager / MANAGER) using this.

grep –v : deleting lines (inverse)

it select all the lines except those containing the pattern.

$ grep –v ‘director’ emp.lst

it will display all lines except those contain director

grep -n : displaying line number

it display line number containing a pattern, along with the lines.

$ grep -n ‘marketing’ emp.lst


grep – c : counting lines containing patterns.

$ grep –c ‘director’ emp.lst -- it will find director from file emp.lst and return no.

4

$ grep –c director emp*.lst -- find pattern from multiple files

emp.lst :4

emp1.lst :2

emp2.lst : 3

emp3.lst :2

grep -l : displaying filenames.

$ grep –l ‘manager’ *.lst -- it display only the names of files containing the pattern.

emp.lst

emp1.lst

grep –e : matching multiple pattern

$ grep –e “saksena” -e “saxena” emp.lst

grep -f : taking a pattern from file

you can place all the pattern in a separate file , one pattern per line and use –f option with grep to take pattern from file.

**Example: cat > pattern** → 

```
manager
admin
tester
accountant
```

$ grep –f pattern emp.lst -- it will take a pattern from file and search from another file.

## grep options :

| Option | Significance |
|---|---|
| -i | Ignores case for matching |
| -v | Doesn't display lines matching expression |
| -n | Displays line numbers along with lines |
| -c | Displays count of number of occurrences |
| -l | Displays list of filenames only |
| -e *exp* | Specifies expression with this option. Can use multiple times. Also used for matching expression beginning with a hyphen. |
| -x | Matches pattern with entire line (doesn't match embedded patterns) |
| -f *file* | Takes patterns from *file*, one per line |
| -E | Treats pattern as an extended regular expression (ERE) |

# Regular expression

| *Symbols or Expression* | *Matches* |
|---|---|
| * | Zero or more occurrences of the previous character |
| g* | Nothing or g, gg, ggg, etc. |
| . | A single character |
| .* | Nothing or any number of characters |
| [pqr] | A single character *p*, *q* or *r* |
| [c1-c2] | A single character within the ASCII range represented by *c1* and *c2* |
| [1-3] | A digit between 1 and 3 |
| [^pqr] | A single character which is not a *p*, *q* or *r* |
| [^a-zA-Z] | A nonalphabetic character |
| ^pat | Pattern *pat* at beginning of line |
| pat$ | Pattern *pat* at end of line |
| bash$ | bash at end of line |
| ^bash$ | bash as the only word in line |
| ^$ | Lines containing nothing |

# Basic Regular Expression(BRE)

## Character Class :[ ]

A regular expression lets you specify a group of characters enclosed within a pair of rectangular brackets, [ ].

e.g [aA] --it matches **either a or A .**

[aA]g[ar][ar]wal – it will match **Agarwal** and **agrawal**

$ grep "[aA]g[ar][ar]wal " emp.lst

```
3564|sudhir Agarwal   |executive|personnel |07/06/47|7500
0110|v.k. agrawal     |g.m.     |marketing |12/31/40|9000
```

Inverse (^) : regular expression use the (^) caret sign for invert the operation. When the character class begins with this character , all character other than the group are match.

$ grep "[^A-Za-z]" emp.lst -- it matches a single nonalphabetic character string.

**The * (zero or more occurrences of the previous character)**

The * refers to the immediately preceding character. It indicates that the previous character can occur many times or not.

Example : a* → aa aaa aaaa aaaaa

$ grep "[aA]gg*[ar][ar]wal" emp.lst

```
2476|anil aggarwal    |manager  |sales      |05/01/59|5000
3564|sudhir Agarwal   |executive|personnel  |07/06/47|7500
0110|v.k. agrawal     |g.m.     |marketing  |12/31/40|9000
```

**The dot (.) : matches a single character.**

Example : 2... --matches four character pattern beginning with a 2. (2???)

The regular expression .* -- matches any number of character or none.

$ grep "j. *saxena" emp.lst

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

## Specifying pattern location (^ and $)

**^ (caret) :** for matching at the beginning of a line.

**$ :** for matching at the end of a line.

Example: $ grep "^2" emp.lst

--extract those lines where 1st column (emp_id) begins with 2

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

$ grep "7...$" emp.lst

--extract those lines where the salary lies between 7000 and 7999.

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

## Reverse search

$ grep "^[^2]" emp.lst →extract only those lines where the emp_id don't begin with 2.

$ ls –l | grep "^d" → display only the directories. (starting with d in ls –l)

# Extended Regular Expression (ERE)

| Expression | Significance |
|---|---|
| *ch+* | Matches one or more occurrences of character *ch* |
| *ch?* | Matches zero or one occurrence of character *ch* |
| *exp1\|exp1* | Matches *exp1* or *exp2* |
| GIF\|JPEG | Matches GIF or JPEG |
| *(x1\|x2)x3* | Matches *x1x3* or *x2x3* |
| (lock\|ver)wood | Matches lockwood or verwood |

**ERE : used to match dissimilar pattern with a single expression**.

grep use ERE with –E option otherwise use egrep but without -e

**The + and ?**

**+** -- Matches one or more occurrences of the previous character.

**?** -- Matches zero or one occurrences of the previous character.

**Example :** b+ --matches b , bb , bbb , bbbb

b? --matches single instance of b or nothing.

$ grep –E "[Aa]gg?arwal" emp.lst --matches gg?

```
2476|anil aggarwal    |manager  |sales      |05/01/59|5000
3564|sudhir Agarwal   |executive|personnel  |07/06/47|7500
```

## Matching multiple patterns( |, (and) )

| - is a delimiter of multiple patterns.

( ) – group of patterns.

Example : $ grep –E 'sengupta | dasgupta' emp.lst

$ grep -E '(sen | das)gupta' emp.lst

## sed – Stream Editor

Most common use of SED command in UNIX is for substitution or for find and replace.

- It is a powerful text stream editor. Can do insertion, deletion, search and replace(substitution).
- Unix supports regular expression which allows it perform complex pattern matching.

**Syntax:** sed options 'address action' file(s)

address and action are enclosed within a single quotes.

Addressing in sed is done in two ways:

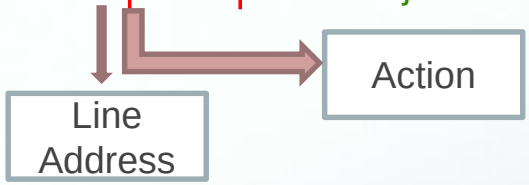
By one or two line numbers (like, 3,7)

By specifying a /-enclosed pattern which occurs in a line.(/ and /)

## Line addressing :

Instruction is enclosed within quotes. First instruction is an address(line number) and second is action(q-quit, p-print).

Example: $ sed '3q' emp.lst --- just like head –n 3 (after 3rd line quit)




Sed command output display a selected lines as well as all lines. So suppress this behavior use –n option.

$ sed –n '1,2p' emp.lst -- print first two lines only.

$ sed –n '9,11p' emp.lst --print lines 9 to 11

$ - select only last line

```
$ sed –n '$p' emp.lst
```

Selecting a multiple group of files (using a single pair of quotes)

```
$ sed –n '1,2p
  7,9p
  $p' emp.lst
```

**3 address in one command , using a single pair of quotes**

Negating the Action (!) (not operator)

```
$ sed –n '3,$!p' emp.lst
```

**-- don't print line 3 to last**

Using multiple instruction (-e)

$ sed –n –e '1,2p' -e '7,9p' -e '$p' emp.lst --multiple group of lines

Take instruction from file (-f)

$ cat >line_add

1,2p
7,9p
$p

$ sed -n –f line_add emp.lst

-f option is used with multiple files as well.

$ sed –n –f line_add emp*.lst

Combine –f and –e option.

$ sed –n –e '/saxena/p' -f line_add -f line_add1 emp*

## Context addressing

The second form of addressing called context addressing. The pattern must be bounded in **/ and /.**

Example: $ sed –n '/director/p' emp.lst **-- just like grep to match a pattern**

$ sed –n '/director/,/manager/p' emp.lst

**-- specify two pattern separated by comma.**

$ sed –n '1,/sales/p' emp.lst **-mixed line and context addressing**

**Using regular expression**

```
$ sed –n '/[aA]gg*[ar][ar]wal/p'  emp.lst
$ sed  –n '/sa[kx]s*ena/p
                /gupta/p' emp.lst
```

regular expressions like grep

**Anchoring characters (^ and $)**

```
$  sed –n '/^2/p' emp.lst
$ sed –n '/50..../p' emp.lst
```

regular expressions in grep and sed are common

It is more powerful tools compare to any other filtering command.

## Writing a selected lines to a file(w)

You can use the w(write) command to write the selected lines to a separate file.

```
$ sed –n '/director/w  dir_list' emp.lst
```

**-- copy all line contains director in dir_list file**

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

**-- copy all line contains director,manager and execuives  in three separate file.**

```
$ sed –n '1,500w file1
          501,$w' file2'  emp.lst
```

**Saves first 500 lines in file1 and the rest of the lines in file2**

## Text Editing

Inserting and changing lines (i,a,c)

**i – for inserting a text.**

```
$ sed '1i\
> #include <stdio.h>\
>#include <conio.h>
>' prg1.c  > $$
```

Add two line at the beginning of a c program.
1i – which insert text at line number 1
Need to use \ before [enter]
No need to write \ in last line of input
Redirect output to a temporary file

-- now we must move this file to prg1.c to be able to use it like,

```
$ mv $$ prg1.c
$ head –n 2 prg1.c
```

**Double spacing text** (insert a blank line after every line **–append file(a)** )

```
$  sed ‘a\


‘ emp.lst
```

i is also used instead of a but i inserted a blank line before each selected line

**Deleting a lines(d)**

```
$ sed ‘/manager/d’ emp.lst
```

→ -n option not to be used with d

both the command selects all lines except those containing manager

```
$ sed –n ‘/manager/!p ‘ emp.lst
```

**Deleting a blank lines** (blank line consist of any number of spaces, tab or nothing)

```
$ sed  ‘/^[    ]*$/d’  emp.lst     -- space and tab inside [  ]
```

**Substitution (s) (replace a pattern in its input with something else)**

syntax : [address] s/expression1/expression2/flags

Here, expression1 is replace with expression2 in all lines specified by [address]. If address is not specified , the substitution is performed for all matching lines.

Example: $ sed 's/ | / : /' emp.lst

it replace only the first instance of | in a line has been replaced with a colon :

So you need to use g(global) flag to replace all the pipes.

$ sed 's/ | / : /g' emp.lst

$ sed '1,3s/ | / : /' emp.lst -- replace character in first three line only.

$ sed ‘1.5s / manager / g.m /’ emp.lst

--replace a word manager with g.m in the first five lines.

**Use regular expression for substitution.**

$ sed ‘s /[Aa]gg*[ar][ar]wal/ Agarwal /g ‘ emp.lst

--to replace all occurances of Agarwal, Aggarwal and Agrawal with simply Agarwal.

**Use Anchoring characters (^ and $)**

$ sed ‘s/^/2/’ emp.lst -- add 2 as prefix to all emp.id.

$ sed ‘s/$/.00/’ emp.lst --you can add the suffix .00 at the end of the line.

**Performing multiple substitution:** (press enter at the end of each instruction)

replace three HTML tags

```
$ sed 's/<I>/<EM>/g
>s/<B>/<STRONG>/g
>s/<U>/<EM>/g ' form1.html
```

**Compressing Multiple Spaces**

```
$ sed 's/    *| / | / g' emp.lst
```

-- delete trailing spaces from fields.

## Remembered Pattern (//)

sed ‘s/director/member/’ emp.lst

sed ‘/director/s/ /member/’ emp.lst

sed ‘/director/s/director/member/’ emp.lst

These three commands perform a same job.

-- the second command (//) remember the scanned pattern, and store it in //. It representing regular expression which is interpreted as a remembered pattern.

sed ‘s/|//g’ emp.lst →remove every | from file

-- you can write // slashes in target field then you can remove that pattern from file.

sed –n ‘/marketing/s/director/member /p’ emp.lst

--replace a string in all lines containing a different string.

## Basic Regular Expression

- **The Repeated pattern**
  It use single symbol, & to make the entire source pattern appear at the destination also.
- **The Interval Regular Expression**
  This expression uses the characters { and } with a single or a pair of numbers between them.
- **The Tagged Regular Expression**
  This expression groups pattern with ( and ) represents them at the destination with numbered tag.

## Repeated Pattern (&)

- when source pattern also occurs at the destination. We can use special character & to represent it. & is the only other special character you can use in the target expression.

Example : $ sed 's/director/executive director/' emp.lst

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

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

All these commands replace director with executive director

## Interval Regular Expression (IRE)

-Match pattern at any specified location, it will use an escape pair of curly braces ,

ch\{m\} – The Metachacter ch can occur m times.

ch\{m,n\} – here , ch can occur between m and n times.

ch\{m,\} – here, ch can occur at least m times.

- All form of **IRE** have the single character regular expression **ch** as the first element.
- It can either be .(dot) or a character class.
- It is followed by a pair of escaped curly braces containing either a single number m, or a range between m and n to determine the number of times the character preceding it can occur.
- The values of ma and n can't exceedn255.

Example : $ grep '[0-9]\{10\}' file 1

– it will extract any numeral character (0-9) can occur 10 times from file.

```
$ cat teledir.txt
jai   sharma 25853670
chanchal singhvi 9831545629
anil  aggarwal 9830263298
shyam saksena 23217847
lalit chowdury 26688726
```

```
$ grep '[0-9]\{10\}' teledir.txt
chanchal singhvi 9831545629
anil  aggarwal 9830263298
```

--Display the listing for those files that have the write bit set either for group or others.

```
$ ls –l  | sed –n ‘/^.\{5,8\}w /p ‘
```

--extracting lines based on length

```
$ sed –n ‘/.\{101,\} /p’ emp.lst        -- line length at least 101
$ grep  ‘^.\{101,150\}$’ emp.lst        -- line length between 101 and 150
```

## Tagged Regular Expression (TRE)

It relates to breaking up a line into groups and then extracting one or more of these groups. It require two regular expression to be specified –one for source and one for target.

Example : you need to extract number as \([0-9]*\)

series of nonalphabetic characters \([^a-zA-Z]*\)

Every group pattern automatically acquires the numeric lable n, i.e 1st group represented as \1 , the second one \2 and so on…

Example: $ sed 's/\([a-z]*\) *\([a-z]*\)/\2 , \1/' teledir.txt | sort

```
$ cat teledir.txt
jai   sharma 25853670
chanchal singhvi 9831545629
anil  aggarwal 9830263298
shyam saksena 23217847
lalit chowdury 26688726
```

```
aggarwal, anil 9830263298
chowdury, lalit 6688726
saksena, shyam 3217847
sharma, jai 5853670
singhvi, chanchal 9831545629
```

we can convert the date format to yyyymmdd from original format. We can generate 19yymmdd format from existing.

In TRE you can use ^ as well instead of /. sed doesn't compulsorily use a / to a delimiter pattern for substitution.

```
$ cat emp.lst
2233|a.k. shukla       |g.m.      |sales      |12/12/52|6000
9876|jai sharma        |director  |production|12/03/50|7000
5678|sumit chakrobarty|d.g.m.    |marketing  |19/04/43|6000
```

Example: sed 's^ \(..\)/ \(..\)/ \(..\) ^ 19\3\2\1^' emp.lst | head –n 3

```
$ sed 's^\(..\)/\(..\)/\(..\)^19\3\2\1^' emp.lst | head -n 3
2233|a.k. shukla       |g.m.      |sales      |19521212|6000
9876|jai sharma        |director  |production|19500312|7000
5678|sumit chakrobarty|d.g.m.    |marketing  |19430419|6000
```

Table 13.4 Internal Commands Used by **sed**

| *Command* | *Description* |
|---|---|
| i, a, c | Inserts, appends and changes text |
| d | Deletes line(s) |
| 10q | Quits after reading the first 10 lines |
| p | Prints line(s) on standard output |
| 3,$p | Prints lines 3 to end (-n option required) |
| $!p | Prints all lines except last line (-n option required) |
| /begin/,/end/p | Prints lines enclosed between begin and end (-n option required) |
| q | Quits after reading up to addressed line |
| r *fname* | Places contents of file *fname* after line |
| w *fname* | Writes addressed lines to file *fname* |
| = | Prints line number addressed |
| s/*s1*/*s2*/ | Replaces first occurrence of expression *s1* in all lines with expression *s2* |
| 10,20s/-/:/ | Replaces first occurrence of - in lines 10 to 20 with a : |
| s/*s1*/*s2*/g | Replaces all occurrences of expression *s1* in all lines with expression *s2* |
| s/-/:/g | Replaces all occurrences of - in all lines with a : |

DONE

# Shell Programming (UNIX Shell Script)

- UNIX shell scripts are text files that contain sequences of UNIX commands.
- A shell usually interprets a single line of input, but we can also create a file containing a number of lines of commands to be interpreted.
  - This file is a program known as a shell script
  - The program can also contain control structures (if-then, loops)
  - Shell scripts allow a sequence of commands to be executed automatically
- A programmer creates shell scripts with a text (vi) editor.
- Shell scripts do not have to be converted into machine language by a compiler.
- This is because the UNIX shell acts as an interpreter when reading script files.
- Use **.sh extension** for shell script.
- As an interpreter reads the statements in a program file, it immediately translates them into executable instructions, and causes them to run.

- A shell script consists of a sequence of built-in and nonbuilt-in commands separated by ; or NEWLINE.
- Comments begin with a #.

## Shell variables

- Shell variables are symbolic names that can access values stored in memory Operators.
- The environment variables, and shell variables.
- Shell variables are those you create at the command line or in a shell script.
- Environment and configuration variables have standard names, such as HOME, PATH, SHELL, USERNAME, and PWD.

## Input and Output

- The command **echo** is used to display words to the standard output.
- To prevent a newline after the last word, use echo -n .

```
echo “hello”
echo –n date
```

- To read user input, the metavariable $ is used.
- Making a script interactive use **read** statement for taking an input form user.

```
echo “ Enter file name: “
read fname
```

## Exmaple:

**Step 1**: **In Vi editor create the shell script : script.sh**

**Step : 2**

```
echo “Today’s date : `date`”
echo “ calender: “
cal
echo “ My shell is : $SHELL”
```

**Step : 3 run Shell Script**

to run the shell script make it executable first **chmod +x script.sh**

**sh script.sh**

# Command Line argument

- Unix shell script also accept arguments from the command line.
- When arguments are specified with a shell script , they assign a special variable,(parameter)
- Parameters can be passed to the script from the command line.
- Inside the script, these parameters may be referenced by using the positional parameters

  $0, $1, $2,$*,$# etc.

# Special Parameters used by shell

**$0** refers to the command name (the name of the shell script).

**$1** is the first argument read by shell.

**$2** is the second argument read by shell, and so on.

**$*** is stores the complete set of positional parameters as a single string.

**$#** is set to number of arguments specified .

**$@** each quoted string treated as a separate argument.

**$?** The exit status of the last command executed.

**$$** The process number(PID) of the current shell.

**$!** The process number of the last background command(JOB)

**sh script1.sh manager emp**

# Exit Status

- The $? variable represents the exit status of the previous command.
- Exit status is a numerical value returned by every command upon its completion. As a rule, most commands return an exit status of **0 if they were successful**, and **1 if they were unsuccessful**.
- Some commands return additional exit statuses for particular reasons. For example, some commands differentiate between kinds of errors and will return various exit values depending on the specific type of failure.

Example :

```
$ grep director  emp.lst  > /dev/null  ; echo $?
0                                      --success
$ grep manager  emp.lst  > /dev/null  ; echo $?
1                                      --Failure  in finding pattern
```

# Logical Operator (&& AND ||)

Shell provides two operators.

cmd1 && cmd2

Logical **AND**. If both the operands are true, then the condition becomes true otherwise false.

cmd1 || cmd2 --Logical OR. If one of the operands is true, then the condition becomes true.

**Example :**

**$ grep 'director' emp.lst && echo "pattern found"**

o/p

pattern found

**$ grep 'manager' emp.lst || echo "pattern not found"**

pattern not found

## Conditional Statements : (if and switch case)

```
if [ expression ]
then
    execute statement
fi
```

```
if [ expression ]
then
        statement1
 else
        statement2
fi
```

```
if [ expression1 ]
then
        statement1
elif [ expression2 ]
then
        statement2
else
        statement3
fi
```

```
case  [expression] in
    Pattern 1) Statement 1;;
    Pattern 2) Statement 2;;
     …..
     Pattern n) Statement n;;
 esac
```

Case statement works as a switch statement if specified value match with the pattern then it will execute a block of that particular pattern.

When a match is found all of the associated statements until the double semicolon (;;) is executed.

A case will be terminated when the last command is executed. If there is no match, the exit status of the case is zero.

# Test and [ ] – to evaluate an expression

- It is used with if statement to evaluate an expression.

-Need test statement because the true and false values returned by expression can't be directly handled by if.

**-test work in three ways,**

compare two numbers.

compare two strings or a single one for a null value.

checks a file 's attribute.

## Numeric Comparison:

Numeric comparison operator used by test begin with –(hyphen) like,

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

## Example:

```
$ x=5 ; y=7; z=5.5
$ test  $x –eq $y      -- equall to
$ test  $x –ne $y      -- not equall to
$ test  $x –lt  $y     -- less than
$ test  $x –gt $y      -- grater than
$ test  $x –ge $y      -- grater than or equall to
$ test  $x –le $y      -- less than or equall to
```

## String comparison

test can be used to compare strings with set of operators like,

| *Test* | *True if* |
|---|---|
| *s1* = *s2* | String *s1* = *s2* |
| *s1* != *s2* | String *s1* is not equal to *s2* |
| -n *stg* | String *stg* is not a null string |
| -z *stg* | String *stg* is a null string |
| *stg* | String *stg* is assigned and not null |
| *s1* == *s2* | String *s1* = *s2* (Korn and Bash only) |

# File tests

test can be used to test the various file attributes like its type or its permissions.

| Test | True if File |
|---|---|
| -f *file* | *file* exists and is a regular file |
| -r *file* | *file* exists and is readable |
| -w *file* | *file* exists and is writable |
| -x *file* | *file* exists and is executable |
| -d *file* | *file* exists and is a directory |
| -s *file* | *file* exists and has a size greater than zero |
| -e *file* | *file* exists (Korn and Bash only) |
| -u *file* | *file* exists and has SUID bit set |
| -k *file* | *file* exists and has sticky bit set |
| -L *file* | *file* exists and is a symbolic link (Korn and Bash only) |
| *f1* -nt *f2* | *f1* is newer than *f2* (Korn and Bash only) |
| *f1* -ot *f2* | *f1* is older than *f2* (Korn and Bash only) |
| *f1* -ef *f2* | *f1* is linked to *f2* (Korn and Bash only) |

# expr : computation and string handling

- The expr command in Unix evaluates a given expression and displays its corresponding output. It is used for:
- Basic operations like addition, subtraction, multiplication, division, and modulus on integers.

## Example:

```
$  x=3  y=5
$ expr 3 + 5     → 8
$ expr $x - $y  → -2
$ expr $x  \*  $y  → 15
$ expr $y / $ x   → 1
$ expr  13  % 5  → 3
```

Evaluating regular expressions, string operations like substring, length of strings etc.

**Determine the length of the string**

```
$ expr "rofel"  :  '.*'   → 5
```

**Extract a substring**

```
$ str1 ="2003"
$ expr  "$str1" : ' ..\(..\)'   → 03
```

**Locate the position of a character**

```
$ str1="Rofel BCA"
$ expr "$str1"  :  '[^B]*B'  → 7
```

# Looping

## while

```
while condition is true
do
        Statement to be executed
done
```

Example:

```
a=0
while [ $a –lt 10 ]
do
   echo $a
   a=`expr $a + 1`
done
```

## for :Looping with list

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

The for loop operate on lists of items. It repeats a set of commands for every item in a list.

Example:

```
for a in 1 2 3 4 5 6 7 8 9 10
do
      echo $a
done
```

## Basename : (changing the filename extension)-external command

It extract the base filename from an absolute pathname.

Example: $ basename /home/rofel/bca/script1.sh o/p→ script1.sh

When base name used with two arguments, it strips off the second argument from the first argument.

Example: $ basename script1.sh sh o/p→ script1.

This feature are used to rename filename extensions from txt to doc or anything else.

Example:

```
for file in *.txt
do
    bname=`basename $file  txt`
   mv  $file  ${bname}doc
done
```

## set (manipulating positional parameters) –internal commands

Set assigns its arguments to the positional parameters $1,$2 and so on. It is useful for picking up individual fields from the output of a program. Set is convert its arguments to positional parameters.

$ set 123 456 789

It assign the value 123 to $1 , 456 to $2 and 789 to $3.

example: $ set `date`

$ echo $*

wed Jul 10 10:20:50 IST 2020

$ echo “The date today is $2 $3 , $6”

The date today is jul 10, 2020

# shift (shifting argument left)

Shift transfer the contents of a positional parameter to its immediate lower numbered.

Example : : $ set `date`

```
$ echo  “$@”
wed Jul 10 10:20:50   IST  2020
$ echo $1 $2 $3
wed   jul  10
$ shift
$ echo $1 $2 $3
jul 10 10:20:50
$ shift 2
$ echo $1 $2 $3
10:20:50   IST   2020
```

# The Here Document (<<)

The shell uses the << symbol so read data from the same file containing the script. It is known as here document , it specifying that the data is here rather than in a separate file.

Standard input can also take input from a here document.

This feature is useful when used with commands that don't accept a filename as argument .

Example :

```
$ sh emp1.sh << END
>Sales
>emp.lst
>END
```

# What is awk?

- created by: Aho, Weinberger, and Kernighan
- scripting language used for manipulating data and generating reports

- versions of awk
  - awk, nawk, mawk, pgawk, …
  - GNU awk: gawk



# What can you do with awk?

- awk operation:
  - scans a file line by line
  - splits each input line into fields
  - compares input line/fields to pattern
  - performs action(s) on matched lines
- Useful for:
  - transform data files
  - produce formatted reports
- Programming constructs:
  - format output lines
  - arithmetic and string operations and conditionals and loops

# Basic awk Syntax

```
awk [options] 'selsction criteria  {action}' file(s)
```

Options:

-F to change input field separator

-f to name script file

Action :

{print}

Example : $ awk '/director/ ' emp.lst -default print

$ awk '/director/ {print} ' emp.lst

$ awk '/director/ {print $0} ' emp.lst -$0 is the complete line

# Arithmetic Operators

| Operator | Meaning | Example |
|---|---|---|
| + | Add | x + y |
| - | Subtract | x – y |
| * | Multiply | x * y |
| / | Divide | x / y |
| % | Modulus | x % y |
| ^ | Exponential | x ^ y |

Example:

```
% awk '$3 * $4 > 500 {print $0}' file
```

# Relational Operators (comparision operator)

| Operator | Meaning | Example |
|---|---|---|
| < | Less than | x < y |
| < = | Less than or equal | x < = y |
| == | Equal to | x == y |
| != | Not equal to | x != y |
| > | Greater than | x > y |
| > = | Greater than or equal to | x > = y |
| ~ | Matched by reg exp | x ~ /y/ |
| !~ | Not matched by req exp | x !~ /y/ |

# Logical Operators

| Operator | Meaning | Example |
|---|---|---|
| && | Logical AND | a && b |
| \|\| | Logical OR | a \|\| b |
| ! | NOT | ! a |

Examples:

```
$ awk '($2 > 5) && ($2 <= 15)
                        {print $0}' file
$ awk '$3 == 100 || $4 > 50' file
```

# Output Statements

**`print`**

print easy and simple output

**`printf`**

print formatted (similar to C printf)

**`sprintf`**

format string (similar to C sprintf)


# Function: print

- Writes to standard output
- Output is terminated by ORS
  - default ORS is newline
- If called with no parameter, it will print $0
- Printed parameters are separated by OFS,
  - default OFS is blank
- Print control characters are allowed:
  - \n \f \a \t \\ …




# print example

```
% awk '{print}' grades
john 85 92 78 94 88
andrea 89 90 75 90 86

% awk '{print $0}' grades
john 85 92 78 94 88
andrea 89 90 75 90 86

% awk '{print($0)}' grades
john 85 92 78 94 88
andrea 89 90 75 90 86
```



# print Example

```
% awk '{print $1, $2}' grades
john 85
andrea 89

% awk '{print $1 "," $2}' grades
john,85
andrea,89
```


# print Example

```
% awk '{OFS="-";print $1 , $2}' grades
john-85
andrea-89

% awk '{OFS="-";print $1 "," $2}' grades
john,85
andrea,89
```



# awk variables

- A user can define any number of variables within an awk script
- The variables can be numbers, strings, or arrays
- Variable names start with a letter, followed by letters, digits, and underscore
- Variables come into existence the first time they are referenced; therefore, they do not need to be declared before use
- All variables are initially created as strings and initialized to a null string “”

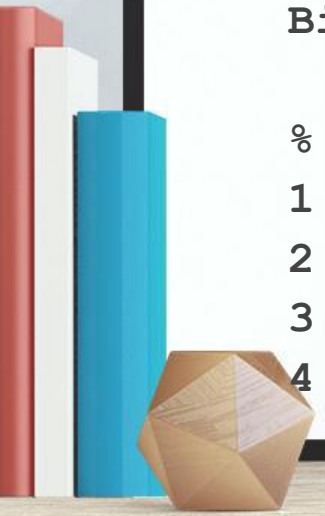
# Some System Variables (Built-in variables )

| | |
|---|---|
| FS | Field separator (default=whitespace) |
| RS | Record separator (default=\n) |
| NF | Number of fields in current record |
| NR | Number of the current record |
| OFS | Output field separator (default=space) |
| ORS | Output record separator (default=\n) |
| FILENAME | Current filename |

# Example: Records and Fields

```
% cat emps
Tom Jones          4424     5/12/66 543354
Mary Adams         5346     11/4/63 28765
Sally Chang        1654     7/22/54 650000
Billy Black        1683     9/23/44 336500

% awk '{print NR, $0}' emps
1 Tom Jones        4424     5/12/66 543354
2 Mary Adams       5346     11/4/63 28765
3 Sally Chang      1654     7/22/54 650000
4 Billy Black      1683     9/23/44 336500
```




# Example: Space as Field Separator

```
% cat emps
Tom Jones          4424     5/12/66 543354
Mary Adams         5346     11/4/63 28765
Sally Chang        1654     7/22/54 650000
Billy Black        1683     9/23/44 336500

% awk '{print NR, $1, $2, $5}' emps
1 Tom Jones 543354
2 Mary Adams 28765
3 Sally Chang 650000
4 Billy Black 336500
```



# Example: Colon as Field Separator

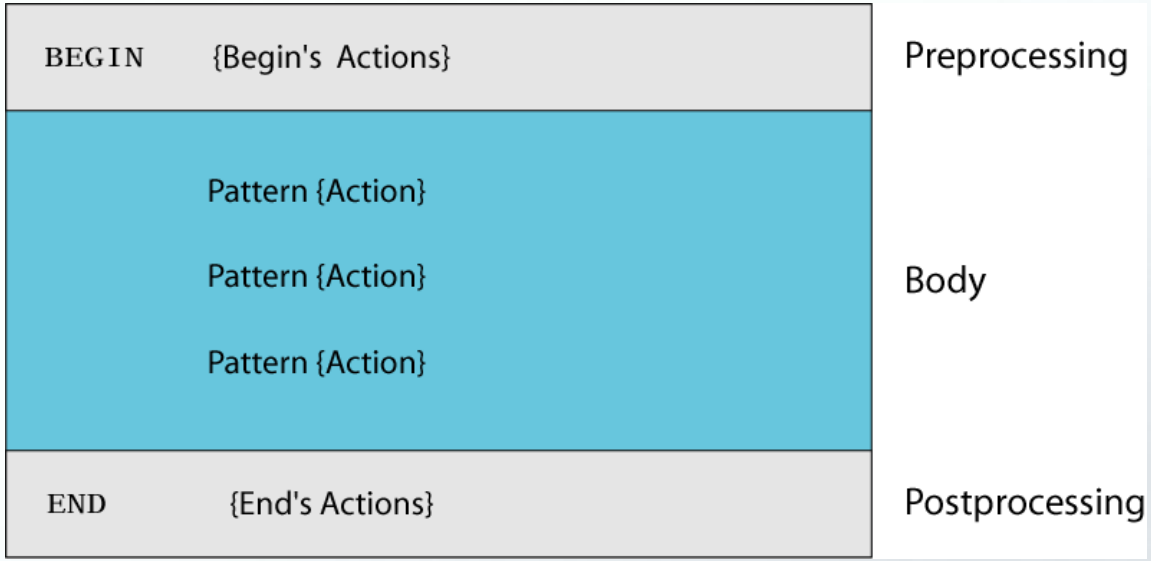
```
% cat em2
Tom Jones:4424:5/12/66:543354
Mary Adams:5346:11/4/63:28765
Sally Chang:1654:7/22/54:650000
Billy Black:1683:9/23/44:336500

% awk -F: '/Jones/{print $1, $2}' em2
        Tom Jones 4424
```




# awk Scripts (Begin And End section)

- awk scripts are divided into three major parts:




- comment lines start with #

# awk Scripts

BEGIN: pre-processing

performs processing that must be completed before the file processing starts (i.e., before awk starts reading records from the input file)

useful for initialization tasks such as to initialize variables and to create report headings



# awk Scripts

BODY: Processing

contains main processing logic to be applied to input records

like a loop that processes input data one record at a time:

- if a file contains 100 records, the body will be executed 100 times, one for each record



# awk Scripts

END: post-processing

contains logic to be executed after all input data have been processed

logic such as printing report grand total should be performed in this part of the script



# Arrays in awk

Syntax:

```
arrayName[index] = value
```

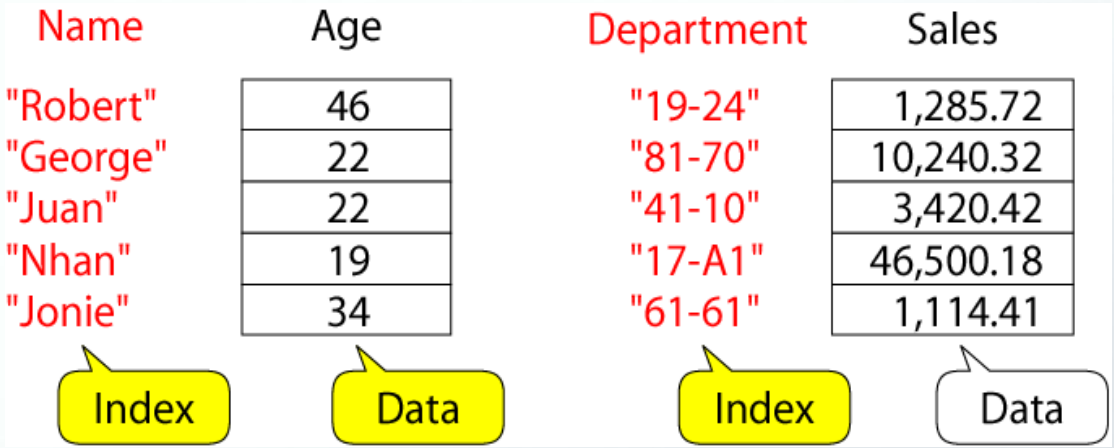
Examples:

```
list[1] = "one"
list[2] = "three"
```


# Associative(hash) Arrays -information is held as key value paired.

The index is the key that is saved internally as a string.

+ awk arrays can use string as index



# awk builtin functions

| *Function* | *Description* |
|---|---|
| `int(`*x*`)` | Returns integer value of *x* |
| `sqrt(`*x*`)` | Returns square root of *x* |
| `length` | Returns length of complete line |
| `length(`*x*`)` | Returns length of *x* |
| `substr(`*stg*`,`*m*`,`*n*`)` | Returns portion of string of length *n*, starting from position *m* in string *stg* |
| `index(`*s1*`,`*s2*`)` | Returns position of string *s2* in string *s1* |
| `split(`*stg*`,`*arr*`,` *ch*`)` | Splits string *stg* into array *arr* using *ch* as delimiter; returns number of fields |
| `system("`*cmd*`")` | Runs UNIX command *cmd* and returns its exit status |

# Examples

```
Int()
Input : $ awk 'BEGIN{print int(3.534);print int(4);print int(-5.223);print int(-5);}‘
 Output : 3 4 -5 -5
```

```
Sqrt ()
Input : $ awk 'BEGIN{print sqrt(16);print sqrt(0);print sqrt(-12);}'
Output : 4 0
```

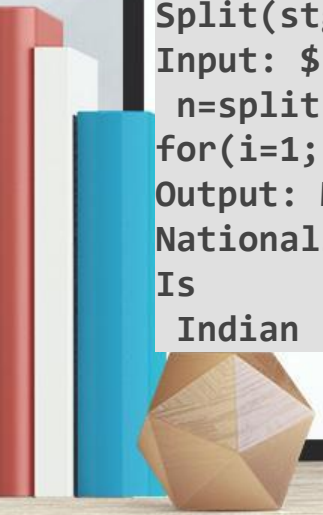
```
Length()
Input: $ awk 'BEGIN{print length(“ROFEL BCA")}'
Output: 9
```

```
Index(s1,s2)
Input: awk 'BEGIN{print index("Graphic", "ph"); print index("University",
"abc")}'
Output: 4 0
```

```
substr(stg,m,n)
Input: $ awk 'BEGIN{print substr(“Unix and Shell Programming", 9)}'
Output: Shell Programming

Input: $ awk 'BEGIN{print substr(" Unix and Shell Programming ", 9,5)}'
Output: Shell
```

```
Split(stg,arr,ch)
Input: $ awk 'BEGIN{string="My Nationality Is Indian"; fieldsep=" ";
 n=split(string, array, fieldsep);
for(i=1; i<=n; i++){printf("%s\n", array[i]);}}'
Output: My
Nationality
Is
 Indian
```




# Control flow – if statement

```
Syntax: if (condition is true)
        {
                action1
        }
        else
        {
                action2
        }
```

Example:

```
awk –F"|" '{ if ($3 == "manager" || $3 == "g.m" )
                {print $1 $3}
             else
                {print $0} ' emp
```

# for Loop

Syntax:

```
for(i=1;i<=n;i++)
        statement
```

Example:

```
for (x in deptSales)
{
      print x, deptSales[x]
}
```

# for Loop for arrays

```
for (var in array)
      statement
```


# while Loop

Syntax:

```
while (logical expression)
        statement
```

Example:

```
i = 1
while (i <= NF)
{
        print i, $i
        i++
}
```


# do-while Loop

Syntax:

```
do
        statement
while (condition)
```

statement is executed at least once, even if condition is false at the beginning

Example:

```
 i = 1
do {
  print $0
  i++
} while (i <= 10)
```



# Thank you