Chapter 1: Introduction

1. Unix Components/Architecture
2. Features of Unix
3. The UNIX Environment and UNIX Structure, Posix and Single Unix specification
4. General features of Unix commands/ command structure. Command arguments and options
5. Basic Unix commands such as echo, printf, ls, who, date,passwd, cal, Combining commands
6. Meaning of Internal and external commands
7. The type command: knowing the type of a command and locating it
8. The root login. Becoming the super user: su command

1.1 The Operating System

- An **operating system** is the software that manages the computer's hardware and provides a convenient and safe environment for running programs.
- It acts as an interface between user programs and the hardware resources that these programs access like – processor, memory, hard disk, printer & so on.
- It is loaded into memory when a computer is booted and remains active as long as the machine is up.

1.2 The UNIX Operating System

- Like DOS and Windows, there's another operating system called UNIX.
- UNIX is a giant operating system with sheer power.
- Developed by Ken Thompson and Dennis Ritchie.
- It runs practically on every Hardware and provided inspiration to the Open Source movement.
- You interact with a UNIX system through a **command interpreter** called the **shell.**

Unix Components/Architecture

- The UNIX architecture has three important agencies-
    - Division of labor: Kernel and shell
    - The file and process
    - The system calls

Division of labor: Kernel and shell

The Kernel

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

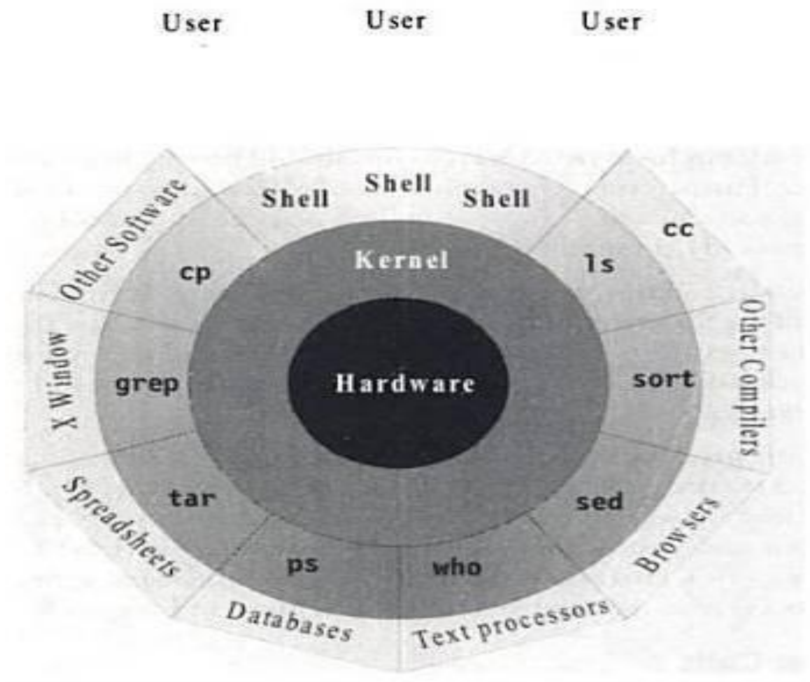
- ✓ Apart from providing support to user's program, kernel also does important housekeeping.
- ✓ It manages the system's memory, schedules processes, decides their priorities and so on.
- ✓ The kernel has to do a lot of this work even if no user program is running.
- ✓ The kernel is also called as the operating system - a programs gateway to the computer's resources.

The Shell

- ✓ Computers don't have any capability of translating commands into action.
- ✓ That requires a **command interpreter**, also called as the shell.
- ✓ Shell is acts interface between the user and the kernel.
- ✓ Most of the time, there's only one kernel running on the system, there could be several shells running – one for each user logged in.
- ✓ The shell accepts commands from user, if require rebuilds a user command, and finally communicates with the kernel to see that the command is executed.
- ✓ Example:

```
$ echo VTU     Belagavi        #Shell rebuilds echo command by removing multiple spaces
VTU Belagavi
```

- ➢ The following figure shows the kernel-shell Relationship:



<u>The file and process</u>

- ➢ Two simple entities support the UNIX – the file and the process.
- ➢ "Files have places and processes have life"

The File

- A file is just an array of bytes and can contain virtually anything.
- A file forms a Hierarchical file system.
- Every file in UNIX is part of the one file structure provided by UNIX.
- UNIX considers directories and devices as members of the file system.

The Process

- The process is the name given to the file when it is executed as a program (Process is program under execution).
- We can say process is the "time image" of an executable file.
- UNIX provides tools to control processes, move them between foreground and background and kill them.

The System Calls

- The UNIX system-comprising the kernel, shell and applications-is written in C.
- Though there are several commands that use functions called system calls to communicate with the kernel.
- All UNIX flavors have one thing in common – they use the same system calls. Eg: write, open

1.3 Features of Unix

1. A Multiuser System
    - UNIX is a multiprogramming system, it permits multiple programs to run and compete for the attention of the CPU.
    - This can happen in two ways:
        - Multiple users can run separate jobs
        - A single user can also run multiple jobs
    - A single user system where the CPU, memory and hard disks are all dedicated to a single user.
    - In UNIX, the resources are shared between all users; UNIX is also a multiuser system.

2. A Multitasking System
    - A single user can also run multiple tasks concurrently.
    - UNIX is a multitasking system.
    - It is usual for a user to edit a file, print another one on the printer, send email to a friend and browse www-all without leaving any of applications.
    - The kernel is designed to handle a user's multiple needs.
    - In a multitasking environment, a user sees one job running in the foreground; the rest run in the background.
    - User can switch jobs between background and foreground, suspend, or even terminate them.

3. The Building-block Approach
    - The designer never attempted to pack too many features into a few tools.
    - Instead, they felt "small is beautiful", and developed a few hundred commands each of which performed one simple job.
    - UNIX offers the | (filters) to combine various simple tools to carry out complex jobs.
    - By interconnecting a number of tools , we can have a large number of combinations of their usage.
    - Example:

```
$ cat note                         #cat displays the file contents
WELCOME TO BRCE
```

```
$ cat note | wc                #wc counts number of lines, words & characters in the file
1 3 15
```

4. The UNIX Toolkit
- There are general-purpose tools, text manipulation utilities (filters), compilers and interpreters , networked applications, system administration tools and shells.

5. Pattern Matching
- UNIX features very sophisticated pattern matching features.
- Example: The * (zero or more occurrences of characters) is a special character used by system to indicate that it can match a number of filenames.

6. Programming Facility
- The UNIX shell is also a programming language; it was designed for programmer, not for end user.
- It has all the necessary ingredients, like control structures, loops and variables, that establish powerful programming language.
- These features are used to design shell scripts – programs that can also invoke UNIX commands.
- Many of the system's functions can be controlled and automated by using these shell scripts.

7. Documentation
- The principal on-line help facility available is the **man** command, which remains the most important references for commands and their configuration files.
- Apart from the man documentation, there's a vast ocean of UNIX resources available on the Internet.

1.4 POSIX AND THE SINGLE UNIX SPECIFICATION

- Dennis Ritchie's decision to rewrite UNIX in C didn't make UNIX portable.
- UNIX fragmentation and absence of single standard affected the development of portable applications.
- First, AT&T creates the System V Interface Definition (SVID).
- Later, X/Open – a consortium of vendors and users, created the X/Open Portability Guide (XPG).
- Products of this specification were UNIX95, UNIX98 and UNIX03.
- Yet another group of standards, the POSIX (Portable Operating System for Computer Environment) were developed by IEEE (Institute of Electrical and Electronics Engineers).
- Two of the most important standards from POSIX are:
- POSIX.1 – Specifies the C application program interface – the system calls (Kernel).
- POSIX.2 – Deals with the Shell and utilities.
- In 2001, a joint initiative of X/Open and IEEE resulted in the unification of two standards.
- This is the Single UNIX Specification, Version 3 (SUSV3).
- The "**Write once, adopt everywhere**" approach to this development means that once software has been developed on any POSIX machine it can be easily ported to another POSIX compliant machine with minimum or no modification.

1.5 Basic Unix commands such as echo, printf, ls, who, date,passwd, cal, Combining commands

1. echo: Displaying The Message

- echo command is used is shell scripts to display a message on the terminal, or to issue a prompt for taking user input.

```
$ echo "Enter your name:\c"
Enter your name:$_
```

```
$echo $SHELL
/usr/bin/bash
```

- Echo can be used with different escape sequences

Constant	Meaning
„a"	Audible Alert (Bell)
„b"	Back Space
„f"	Form Feed
„n"	New Line
„r"	Carriage Return
„t"	Horizontal Tab
„v"	Vertical Tab
„\"	Backslash
„\0n"	ASCII character represented by the octal value n

2. printf: An Alternative To Echo

- The printf command is available on most modern UNIX systems, and is the one we can use instead of echo. The command in the simplest form can be used in the same way as echo:

```
$ printf  "Enter your name\n"
Enter your name
$_
```

- printf also accepts all escape sequences used by echo, but unlike echo, it doesn"t automatically insert newline unless the \n is used explicitly. printf also uses formatted strings in the same way the C language function of the same name uses them:

```
$ printf  "My current shell is %s\n" $SHELL
My current shell is /bin/bash
$_
```

- The %s format string acts as a placeholder for the value of $SHELL, and printf replaces %s with the value of $SHELL. %s is the standard format used for printing strings. printf uses many of the formats used by C"s printf function. Here are some of the commonly used ones:

%s – String

%30s – As above but printed in a space 30 characters wide
%d – Decimal integer
%6d - As above but printed in a space 30 characters wide
%o – Octal integer
%x – Hexadecimal integer
%f – Floating point number

Example:

```
$ printf  "The value of 255 is %o in octal and %x in hexadecimal\n" 255 255

The value of 255 is 377 in octal and ff in hexadecimal
```

3. <u>ls:listing directories and files</u>

- The command to list your directories and files is ls.
- With options it can provide information about the size, type of file, permissions, dates of file creation, change and access.

Syntax

ls [options] [argument]

- When no argument is used, the listing will be of the current directory. There are many very useful options for the ls command. A listing of many of them follows. When using the command, string the desired options together preceded by "-".
  - a Lists all files, including those beginning with a dot (.).
  - d Lists only names of directories, not the files in the directory
  - F Indicates type of entry with a trailing symbol: executables with *, directories with / and symbolic links
  - R Recursive list
  - u Sorts filenames by last access time
  - t Sorts filenames by last modification time
  - i Displays inode number
  - l Long listing: lists the mode, link information, owner, size, last modification (time). If the file is a symbolic link, an arrow (-->) precedes the pathname of the linked-to file.
  - x multicolumnar output

$ls –ld helpdir progs

- drwxr-xr-x 2 kumar metal 512 may 9 10:31 helpdir
- drwxr-xr-x 2 kumar metal 512 may 9 09:57 progs

<u>output in multiple columns(-x):</u>

$ ls –x

<u>Identifying Directories and executables(-F)</u>

$ ls –Fx

<u>Showing hidden files(-a)</u>

$ ls -axF

<u>Listing directory contents</u>

$ ls -x

4. who: Who Are The Users

- UNIX maintains an account of list of all users logged on to the system. The who command displays an informative listing of these users:

```
$ who
 root   tty7        2017-09-04 16:38 (:0)
 root   tty17       2017-09-04 16:38 (:0)
 $_
```

- Following command displays the header information with –H option,

```
$ who -H
NAME     LINE        TIME           COMMENT
root   tty7             2017-09-04 16:38 (:0)
$_
```

- -u option is used with who command displays detailed information of users:

```
$ who -Hu
NAME     LINE        TIME              IDLE      PID  COMMENT
root          tty7      2017-09-04 16:38  00:18      1865  (:0)
$_
```

5. date: Displaying The System Date

- One can display the current date with the date command, which shows the date and time to the nearest second:

```
$ date
Mon Sep 4 16:40:02 IST 2017
```

- The command can also be used with suitable format specifiers as arguments. Each symbol is preceded by the + symbol, followed by the % operator, and a single character describing the format. For instance, you can print only the month using the format +%m:

```
$date +%m
09
```

Or

the month name name:

```
$ date +%h
Aug
```

Or

You can combine them in one command:

```
 $ date + "%h %m"
Aug 08
```

- There are many other format specifiers, and the useful ones are listed below:
  - d – The day of month (1 - 31)
  - y – The last two digits of the year.
  - H, M and S – The hour, minute and second, respectively.
  - D – The date in the format *mm/dd/yy*
  - T – The time in the format *hh:mm:ss*

6. **Passwd: Changing your password**

- Password prevents from accessing the system
- If account doesn't have password or has one that is already known to others, should change it immediately.
- This can be done with **passwd** command:

```
$ passwd
passwd: changing password for Kumar
Enter login password: *******                    Asks for old password
New password: ********
Re-enter new password: ********
passwd (SYSTEM): passwd successfully changed for Kumar
```

- Passwd expects you to respond three times. First, it primpts for the old password. Next, it schecks whether you have entered a valid password, and if you have, it then prompts for the new password.
- Enter the new password using the password naming rules.
- Finally, passwd asks you to reenter the new password. Later, the new password is registered by the system.
- Rules for framing your own password:
  - Don't choose a password similar to the old one.
  - Don't use commonly used names like names of friends, relatives, pets and so forth.
  - Use a mix of alphabetic or numeric characters.
  - Don't write a password in an easily accessible document.
  - Change the password regularly.
  -

7. **cal: The Calendar**

- cal command can be used to see the calendar of any specific month or a complete year.

Syntax:

cal [[month] year]

- Everything within the rectangular box in optional. So cal can be used without any arguments, in which case it displays the calendar of the current month

```
$ cal
  September 2017
Su Mo Tu We Th Fr Sa
                1  2
 3 4  5     6    7 8  9
10 11 12  13  14 15 16
17 18 19  20  21 22 23
24 25 26  27  28 29 30
```

- The syntax show that cal can be used with arguments, the month is optional but year is not. To see the calendar of month August 2017, we need to use two arguments as shown below,

```
$ cal 8 2017
   August 2017
```

```
Su Mo Tu We Th Fr Sa
          1   2  3 4  5
   6 7 8   9  10 11 12
13 14 15  16  17 18 19
20 21 22  23  24 25 26
27 28 29  30  31
```

- You can"t hold the calendar of a year in a single screen page; it scrolls off rapidly before you can use [ctrl-s] to make it pause. To make cal pause using pager using the | symbol to connect them.
  $cal 2017 | more

8. Combining commands

- UNIX allows you to specify more than one command in the single command line.
  Example:

```
$ wc note; ls -l note          #Two commands combined here using ; (semicolon)
2 3 16 note
-rw-rw-r-- 1 mahesh mahesh 16 Jan 30 09:35 note
```

  In the above example, **wc** command returns number of lines, words and characters in that file, and **ls –l** command returns list all the attributes of the file
- A command line can overflow or be split into multiple lines
- UNIX terminal width is restricted to maximum 80 characters.
- Shell allows command line to overflow or be split into multiple lines.
  Example:

```
$ echo "This is                    # $ first prompt
> a three-line                     # > Second prompt
> text message"                    #Command line ends here

This is
a three-line text
message
```

1.6 INTERNAL AND EXTERNAL COMMANDS

- When the shell execute command(file) from its own set of built-in commands that are not stored as separate files in /bin directory, it is called internal command.
- They can be executed any time and are independent.
- If the command (file) has an independence existence in the /bin directory, it is called external command.
- External commands are loaded when the user requests for them. It will have an individual process.

Examples:

```
$ type echo                        # echo is an internal command
echo is shell built-in
```

```
$ type ls                               # ls is an external command
ls is /bin/ls
```

- If the command exists both as an internal and external one, shell execute internal command only.
- Internal commands will have top priority compare to external command of same name.

1.7 The type command: knowing the type of a command and locating it

- The **type** command is used to describe how its argument would be translated if used as commands. It is also used to find out whether it is built-in or external binary file.
- The type command is shell builtin.

Syntax:

```
type [Options] command names
```

- If you want to know the location of executable program (or command), use **type** command-

  Example:

  ```
  $ type date
  date is /bin/date
  $ type echo
  echo is a shell builtin
  ```

- When you execute **date** command, the shell locates this file in the **/bin** directory and makes arrangements to execute it.

1.8 The root login. Becoming the super user: su command.

- The unix system provides a special login name for the exclusive use of the administrator; it is called root. This account doesn"t need to be separately created but comes with every system. Its password is generally set at the time of installation of the system and has to be used on logging in:

  Becoming the super at login time

  ```
  login: root
  Password: ********* [Enter]

  # -
  ```

- The prompt of root is #
- Once you login as a root you are placed in **root's home directory.** Depending on the system, this directory could be / or /root.

- Administrative commands are resident in /sbin and /usr/sbi modern systems and in older system it resides in /etc.
- Roots PATH list includes detailed path, for example:

/sbin:/bin:/usr/sbin:/usr/bin:/usr/dt/bin

Becoming the super user using *su* command

$ su: Acquiring superuser status

- Any user can acquire superuser status with the su command if they know the root password. For example, the user ***abc*** becomes a superuser in this way.

```
$ su
Password: ***********                                  #Password of root user

#pwd
/home/abc
```

- Though the current directory doesn‟t change, the # prompt indicates that ***abc*** now has powers of a superuser. To be in root‟s home directory on superuser login, use **su –l.**
- User‟s often rush to the administrator with the complaint that a program has stopped running. The administrator first tries running it in a simulated environment. **Su** command when used with a – (minus), recreates the user‟s environment without the login-password route:

```
$su – abc
```

- This sequence executes **abc's .**profile and temporarily creates **abc's** environment. su runs in a separate sub-shell, so this mode is terminated by hitting **[ctrl-d]** or using **exit.**

Chapter 2:Unix files

1. Naming files.
2. Basic file types/categories.
3. Parent child relationship.
4. The home directory and the HOME variable.
5. Reaching required files- the PATH variable, manipulating the PATH, Relative and absolute pathnames.
6. Directory commands – pwd, cd, mkdir, rmdir commands.
7. The dot (.) and double dots (..) notations to represent present and parent directories and their usage in relative path names.
8. File related commands – cat, mv, rm, cp, wc and od commands.

The File

- The file is the container for storing information.
- Neither a file's size nor its name is stored in file.
- All file attributes such as file type, permissions, links, owner, group owner etc are kept in a separate area of the hard disk, not directly accessible to humans, but only to kernel.
- UNIX treats directories and devices as files as well.
- All physical devices like hard disk, memory, CD-ROM, printer and modem are treated as files.

2.1 Naming files

- A filename can consist up to 255 characters.
- File may or may not have extensions, and consist of any ASCII character expect the / & NULL character.
- Users are permitted to use control characters or other unprintable characters in a filename.
- Examples - .last_time list. @#$%*abcd a.b.c.d.e
- But, it is recommended that only the following characters be used in filenames-
    - Alphabetic characters and numerals
    - the period(.), hyphen(-) and underscore(_).

2.2 Basic file types/categories

- The UNIX has divided files into three categories:

1. **Ordinary file** – also called as regular file. It contains only data as a stream of characters.
2. **Directory file** – it contains files and other sub-directories.
3. **Device file** – all devices and peripherals are represented by files.

Ordinary File(Regular)

- the most common file type.
- Ordinary file itself can be divided into two types-

1. **Text File** – it contains only printable characters, and you can often view the contents and make sense out of them. e.g.: C and Java program files, shell and perl scripts.
2. **Binary file** – it contains both printable and unprintable characters that cover entire ASCII range. e.g.:- Most Unix commands, executable files, pictures, sound and video files are binary.

Directory File

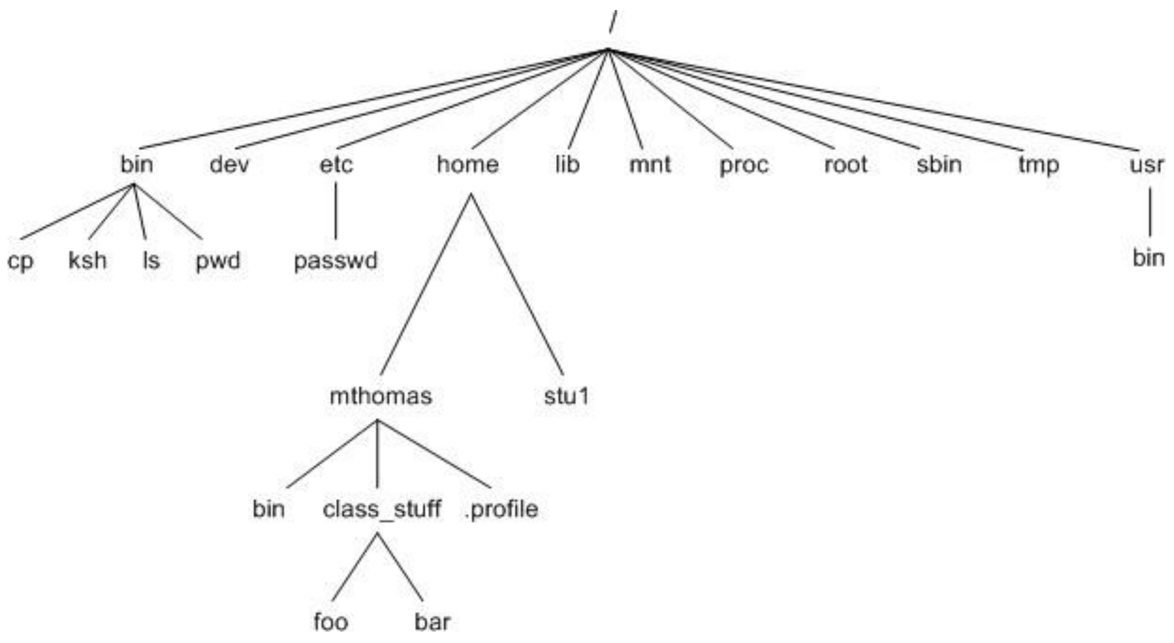
- A directory contains no data but keeps some details of the files and subdirectories that it contains.
- A directory file contains an entry for every file and subdirectories that it houses. If you have 20 files in a directory, there will be 20 entries in the directory.
- Each entry has two components-
    - the filename
    - a unique identification number for the file or directory (called as inode number).
- When any file is created or removed, the kernel automatically updates its corresponding directory by adding or removing the entry (inode number & filename) associated with that file.

Device File

- Installing software from CD-ROM, printing files and backing up data files to tape.
- All of these activities are performed by reading or writing the file representing the device.
- Advantage of device file is that some of the commands used to access an ordinary file also work with device file.
- Device filenames are generally found in a single directory structure, /dev.
- The attributes of every file is stored on the disk.
- The kernel identifies a device from its attributes and then uses them to operate the device.

2.3 Parent child relationship

- The files in UNIX are related to one another.
- The file system in UNIX is a collection of all of these files (ordinary, directory and device files) organized in a Hierarchical (an inverted tree) structure as shown in below figure,



.

- The feature of UNIX file system is that there is a top, which serves as the reference point for all files
- This top is called root and is represented by a / (Front slash).
- The root is actually a directory.
- The root directory (/) has a number of subdirectories under it.
- The subdirectories in turn have more subdirectories and other files under them.
- Every file apart from root, must have a parent, and it should be possible to trace the ultimate parentage of a file to root.
- In parent-child relationship, the parent is always a directory.
- The home directory is the parent of mthomas. / is the parent of home and the grandparent of mthomas.

2.4 The home directory and the HOME variable.

- When you logon to the system, UNIX places you in a directory called home directory.
- It is created by the system when the user account is created.
- If a user login using the login name kumar, user will land up in a directory that could have the path name /home/kumar.
- The shell variable HOME knows the home directory.

```
$echo $HOME
/home/kumar
```

2.5 Reaching required files- the PATH variable, manipulating the PATH, Relative and absolute pathnames.

and

The dot (.) and double dots (..) notations to represent present and parent directories and their usage in relative path names.

PATH

- A shell variable that contains a colon-delimited list of directories that the shell will lok through to locate a command invoked by user.
- The PATH generally includes /bin and /usr/bin for nonprivileged users and /bin and /usr/sbin for the superuser.
- Use echo to evaluate this variable in which directory list separated by colons:

```
$ echo $PATH
/bin:/usr/bin:/usr/local/bin:/usr/ccs/bin:/usr/local/java/bin:.
```

  There are six directories in this colon separated list.

- When you issue a command , the shell searches this list in the sequence specified to locate and execute it.

Absolute Pathnames:

- It shows a file's location with reference to the top, i.e., root
- It is simply a sequence of directory names separated by slashes

  e.g.: /home/kumar
- Suppose we are placed in /usr and want to access the file login.sql which is present in /home/kumar, we can give the pathname of the file as below:

```
cat /home/kumar/login.sql
```

- If the first character of a pathname is /, the file's location must be determined wrt root(the first /).such a pathname is called an absolute pathname.
- If you know the location of a particular command, you van precede its name with the complete path.
- Since **date** reside in/bin (or /usr/bin), you can use the absolute pathname:

  $ **/bin/date**

  Thu Sep 1 09:39:55 IST 2020

Relative Pathname:

- 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 /**.
- A pathname which specifies the location of a file using the symbols. and ..

  . *(single dot) –represents the current directory*

.. (two dots) –represents the parent directory

- Assume that we are in /home/kumar/progs/data/text , we can use .. as an argument to cd to move to the parent directory, /home/kumar/progs/data as shown below:

```
$ pwd
/home/kumar/progs/data/text
$ cd ..                          // moves one level up
$ pwd
/home/kumar/progs/data
```

- To move to /home , we can use the relative path name as follows:

```
$ pwd
/home/kumar/pis
$ cd ../..                // moves two levels up
$ pwd
/home
```

2.6 Directory commands – pwd, cd, mkdir, rmdir commands.

pwd: CHECKING YOUR CURRENT DIRECTORY

- pwd is print working directory
- Any time user can know the current working directory using pwd command.

```
$ pwd
/home/kumar
```

- Like HOME, pwd displays the absolute path.

cd: CHANGING THE CURRENT DIRECTORY

- User can move around the UNIX file system using cd (change directory) command.

When used with the argument, it changes the current directory to the directory specified as argument, progs: **$ pwd** /home/kumar **$ cd** progs **$ pwd** /home/kumar/progs	cd can also be used without arguments: **$ pwd** /home/kumar/progs **$cd** **$ pwd** /home/kumar
Here we are using the relative pathname of progs directory. The same can be done with the absolute pathname also. **$cd** /home/kumar/progs **$ pwd** /home/kumar/progs **$cd /bin** **$ pwd** /bin	cd without argument changes the working directory to home directory. **$cd /home/sharma** **$ pwd** /home/sharma **$cd** /home/kumar

mkdir: MAKING DIRECTORIES

- Directories are created with **mkdir** (make directory) command.
- The command is followed by names of the directories to be created. A directory patch is created under current directory like this:

```
$mkdir patch
```

- You can create a number of subdirectories with one **mkdir** command:

```
$mkdir patch dba doc
```

- For instance the following command creates a directory tree:

```
$mkdir progs progs/cprogs progs/javaprogs
```

- This creates three subdirectories – progs, cprogs and javaprogs under progs.
- The order of specifying arguments is important. You cannot create subdirectories before creation of parent directory.
- For instance following command doesn‘t work

```
$mkdir progs/cprogs progs/javaprogs progs
mkdir: Failed to make directory “progs/cprogs”; No such directory mkdir: Failed to make directory “progs/javaprogs”; No such directory
```

- System refuses to create a directory due to fallowing reasons:
  - The directory is already exists.
  - There may be ordinary file by that name in the current directory.
  - User doesn‘t have permission to create directory

rmdir: REMOVING A DIRECTORY

- The **rmdir** (remove directory) command removes the directories. You have to do this to remove progs:

```
$rmdir progs
```

- If **progs** is empty directory then it will be removed form system.
- Following command used with **mkdir** fails with **rmdir**

```
$mkdir progs progs/cprogs progs/javaprogs
rmdir: directory “progs”: Directory not empty
```

- First subdirectories need to be removed from the system then parent.
- Following command works with **rmdir**

```
$mkdir progs/cprogs progs/javaprogs progs
```

- First it removes **cprogs** and **javaprogs** form **progs** directory and then it removes **progs** fro system.
- **rmdir : Things to remember**

- You can‘t remove a directory which is not empty
- You can‘t remove a directory which doesn‘t exist in system.
- You can‘t remove a directory if you don‘t have permission to do so.

2.7 <u>File related commands – cat, mv, rm, cp, wc and od commands.</u>

cat: DISPLAYING AND CREATING FILES

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

```
$ cat cprogram.c # include
<stdioh> void main ()

{
        Printf(–hello‖);
}
```

- As like other files cat accepts more than one filename as arguments

```
$ cat ch1 ch2

It contains the contents of chapter1
It contains the contents of chapter2
```

- The contents of the second files are shown immediately after the first file without any header information. So cat concatenates two files- hence its name

cat OPTIONS

<u>Displaying Nonprinting Characters (-v)</u>

- cat without any option it will display text files. Nonprinting ASCII characters can be displayed with –v option.

<u>Numbering Lines (-n)</u>

- -n option numbers lines. This numbering option helps programmer in debugging programs.

Using cat to create a file

- **cat** is also useful for creating a file. Enter the command **cat**, followed by > character and the filename.

```
$ cat > new
This is a new file which contains some text, just to Add some
contents to the file new
[ctrl-d]
$_
```

- When the command line is terminated with **[Enter]**, the prompt vanishes. Cat now waits to take input from the user. Enter few lines; press **[ctrl-d]** to signify the end of input to the system
- To display the file contents of new use file name with **cat** command.

  $ cat new

  This is a new file which contains some text, just to Add some
  contents to the file new

<u>mv: RENAMING FILES</u>

- The mv command renames (moves) files. The main two functions are:
  1. It renames a file(or directory)
  2. It moves a group of files to different directory
- It doesn't create a copy of the file; it merely renames it. No additional space is consumed on disk during renaming.

  Eg : To rename the file csb as csa we can use the following command

  $ mv csb csa

- If the destination file doesn‘t exist in the current directory, it will be created. Or else it will just rename the specified file in mv command.
- A group of files can be moved to a directory.

  Eg : Moves three files ch1,ch2,ch3 to the directory module

  $ mv ch1 ch2 ch3 module
- Can also used to rename directory

  $ mv rename newname

- mv replaces the filename in the existing directory entry with the new name. It doesn't create a copy of the file; it renames it.
- Group of files can be moved to a directory
- mv chp1 chap2 chap3 **unix**

<u>rm: DELETING FILES</u>

- The rm command deletes one or more files.

  Eg: Following command deletes three files:

  $ rm mod1 mod2 mod3

  - Can remove two chapters from usp directory without having to cd

    Eg:

    $rm usp/marks ds/marks

  - To remove all file in a directory use *****

$ rm *

- Removes all files from that directory

<u>rm options</u>

- **Interactive Deletion (-i) :** Ask the user confirmation before removing each file:

```
$ rm -i ch1 ch2
rm: remove ch1 (yes/no)? ? y

rm: remove ch1 (yes/no)? ? n [Enter]
```

- A y removes the file (ch1) any other response like n or any other key leave the file undeleted.
- **Recursive deletion (-r or -R):** It performs a recursive search for all directories and files within these subdirectories. At each stage it deletes everything it finds.

$ rm -r * *Works as rmdir*

- It deletes all files in the current directory and all its subdirectories.
- **Forcing Removal (-f): rm** prompts for removal if a file is **write-protected.** The **-f** option overrides this minor protection and forces removal.

rm -rf* ***Deletes everything in the current directory and below***

<u>cp: COPYING A File</u>

- The **cp** command copies a file or a group of files. It creates an exact image of the file on the disk with a different name. The **syntax takes two filename** to be specified in the command line.
- When both are ordinary files, **first file is copied to second.**

$ cp csa csb

- If the destination file (csb) doesn‘t exist, it will first be created before copying takes place. If not it will simply be overwritten without any warning from the system.

Example to show two ways of copying files to the cs directory:

```
$ cp ch1 cs/module1                    #ch1 copied to module1 under cs

$ cp ch1 cs                            #ch1 retains its name under cs
```

- cp can also be used with the shorthand notation, .(dot), to signify the current directory as the destination. To copy a file „**new** from /home/user1 to your current directory, use the following command:

$cp /home/user1/new #new *destination is a file*

```
$cp /home/user1/new .                #destination is the current directory
```

- cp command can be used **to copy more than one file with a single invocation of the command.** In this case the last filename must be a directory.
  Eg: To copy the file ch1,chh2, ch3 to the module , use cp as

```
$ cp ch1 ch2 ch3 module
```

- The files will have the same name in **module**. If the files are already resident in **module**, they will be overwritten. In the above diagram module directory should already exist and cp doesn‘t able create a directory.
- UNIX system uses * as a shorthand for multiple filenames.
  Eg:

```
$ cp ch* usp                #Copies all the files beginning with ch
```

cp options

- **Interactive Copying(-i) :** The **–i** option warns the user before overwriting the destination file, If unit 1 exists, cp prompts for response

```
$ cp -i ch1 unit1
$ cp: overwrite unit1 (yes/no)? Y
```

- A y at this prompt overwrites the file, any other response leaves it uncopied.

Copying directory structure (-R) :

- It performs recursive behavior command can descend a directory and examine all files in its subdirectories.
- ***-R : behaves recursively to copy an entire directory structure***

```
$ cp -R usp newusp

$ cp -R class newclass
```

- If the **newclass/newusp** doesn‘t exist, **cp** creates it along with the associated subdirectories.

wc: COUNTING LINES,WORDS AND CHARACTERS

- wc command performs Word counting including counting of lines and characters in a specified file. It takes one or more filename as arguments and displays a four columnar output.

```
$ wc ofile
4 20 97 ofile
```

- Line: Any group of characters not containing a newline
- Word: group of characters not containing a space, tab or newline
- Character: smallest unit of information, and includes a space, tab and newline
- **wc** offers 3 options to make a specific count. –l option counts only number of lines, - w and –c options count words and characters, respectively.

```
 $ wc -l ofile
   4 ofile
 $ wc -w ofile
   20 ofile
 $ wc -c ofile
   97 ofile
```

- Multiple filenames, **wc** produces a line for each file, as well as a total count.

```
$ wc chap01 chap02 chap03
```

od: DISPLAYING DATA IN OCTAL

- **od** command displays the contents of executable files in a **ASCII octal value.**

```
$ more ofile
 this file is an example for od command
 ^d used as an interrupt key
```

- -b option displays this value for each character separately.
- Each line displays 16 bytes of data in octal, preceded by the **offset in the file of the first byte in the line.**

```
$ od –b file

0000000 164 150 151 163 040 146 151 154 145 040 151 163 040 141 156 040
0000020 145 170 141 155 160 154 145 040 146 157 162 040 157 144 040 143
0000040 157 155 155 141 156 144 012 136 144 040 165 163 145 144 040 141
0000060 163 040 141 156 040 151 156 164 145 162 162 165 160 164 040 153
0000100 145 171
```

- **-c character option**
- Now it shows the printable characters and its corresponding ASCII octal representation

```
 $ od –bc file

    od -bc ofile
    0000000 164 150 151 163 040 146 151 154 145 040 151 163 040 141 156 040
            T   h   i   s       f   i   l   e       i   s       a   n 0000020 145 170
    141 155 160 154 145 040 146 157 162 040 157 144 040 143
            e   x   a   m   p   l   e       f   o   r       o   d       c 0000040 157
```

```
155 155 141 156 144 012 136 144 040 165 163 145 144 040 141
          o    m  m   a    n    d  \n  ^    d        u    s    e    d          a 0000060 163
040 141 156 040 151 156 164 145 162 162 165 160 164 040 153
          s        a     n        i    n    t     e     r     r     u    p    t         k 0000100 145
171
          e    y
```

Some of the representation:

- The tab character, [ctrl-i], is shown as \t and the octal value 011
- The bell character , [ctrl-g] is shown as 007, some system show it as \a
- The form feed character,[ctrl-l], is shown as \f and 014
- The LF character, [ctrl-j], is shown as \n and 012
- od makes the newline character visible too.