**PRACTICAL-1**

**Working Of Different Kernels :**

* **UNIX Architecture**
* **Types Of OS – Eg Linux, Unix, MAC, Window, etc**
* **Flavours of Linux**

**INTRODUCTION**

**UNIX** is an operating system which was first developed in the 1960s, and has been under constant development ever since. By operating system, we mean the suite of programs which make the computer work. It is a stable, multi-user, multi-tasking system for servers, desktops and laptops.

UNIX systems also have a graphical user interface (GUI) similar to Microsoft Windows which provides an easy to use environment. However, knowledge of UNIX is required for operations which aren't covered by a graphical program, or for when there are no windows interface available, for example, in a telnet session.

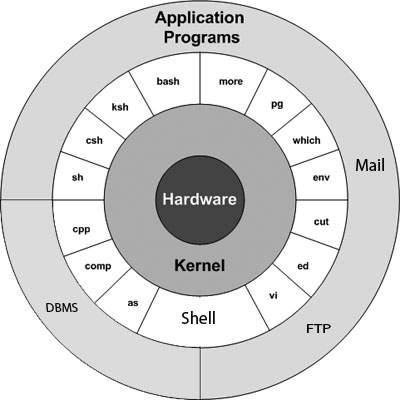
**TYPES OF UNIX**



There are many different versions of UNIX, although they share common similarities. The most popular varieties of UNIX are :

**Sun Solaris**, **GNU/Linux**, and **MacOS X**.

**UNIX ARCHITECTURE**



* **Hardware –** It consists of all hardware related information.
* **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. It allocates time and memory to programs and handles the file store and communications in response to system calls.
* **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.
* **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.
* **Application Layer –** It is the outermost layer that executes the given external applications
* **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 file system.

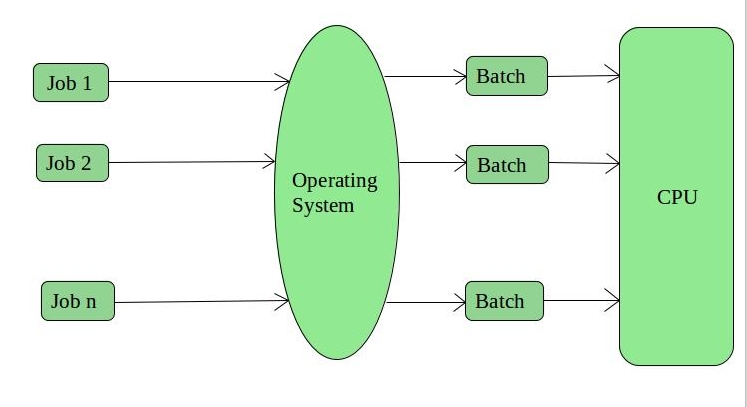
**OPERATING SYSTEMS**

An [Operating System](https://www.geeksforgeeks.org/operating-system-introduction-operating-system-set-1/) performs all the basic tasks like managing file, process, and memory. Thus operating system acts as manager of all the resources, i.e. **resource manager**. Thus operating system becomes an interface between user and machine.

# TYPES OF OPERATING SYSTEMS

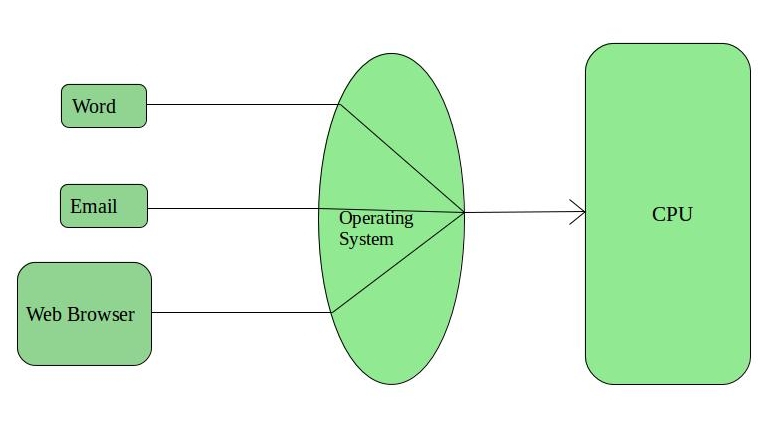
Some of the widely used operating systems are as follows-

**BATCH OPERATING SYSTEM**

  
This type of operating system does not interact with the computer directly. There is an operator which takes similar jobs having same requirement and group them into batches. It is the responsibility of operator to sort the jobs with similar needs.

**Examples of Batch based Operating System:** Payroll System, Bank Statements etc.

**TIME-SHARING OPERATING SYSTEMS**

  
Each task is given some time to execute, so that all the tasks work smoothly. Each user gets time of CPU as they use single system. These systems are also known as Multitasking Systems. The task can be from single user or from different users also. The time that each task gets to execute is called quantum. After this time interval is over OS switches over to next task.

**Examples of Time-Sharing OSs are:** Multics, Unix etc.

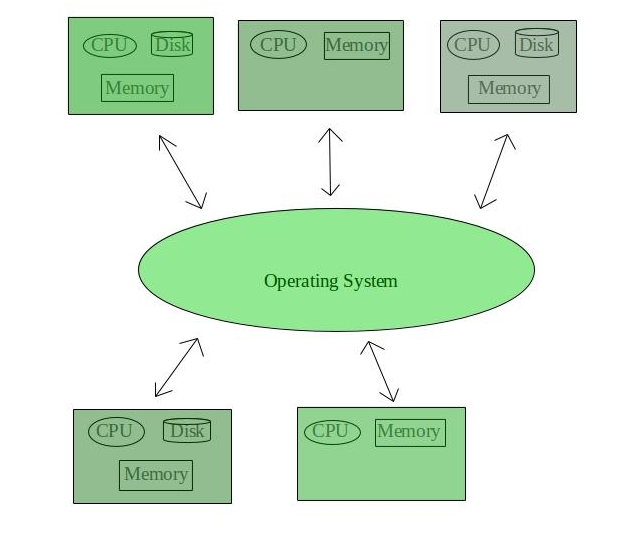
**DISTRIBUTED OPERATING SYSTEM**

These types of operating system have various autonomous interconnected computers communicate each other using a shared communication network. Independent systems possess their own memory unit and CPU. These are referred as **loosely coupled systems** or distributed systems. These system’s processors differ in size and function. The major benefit of working with these types of operating system is that it is always possible that one user can access the files or software which are not actually present on his system but on some other system connected within this network i.e., remote access is enabled within the devices connected in that network.

**Examples of Distributed Operating System are-** LOCUS etc.

### Types of Distributed Operating Systems

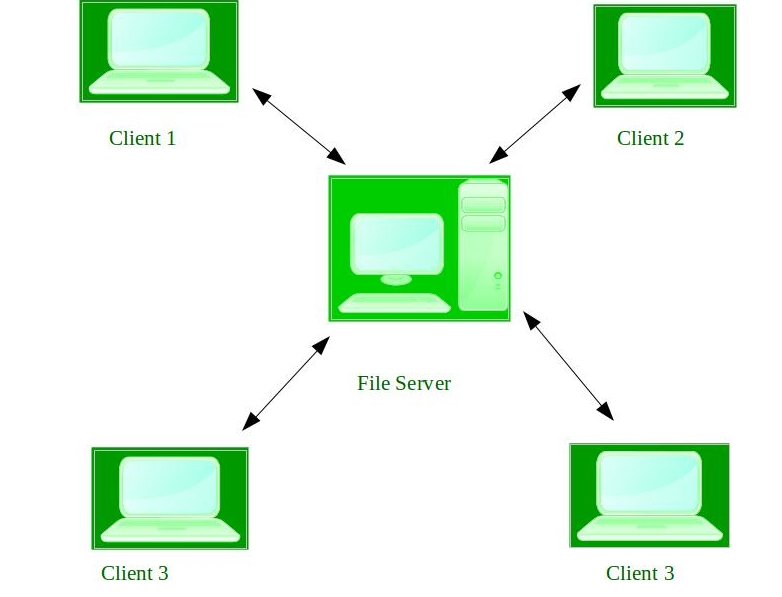
1. Client-Server Systems
2. Peer-to-Peer System

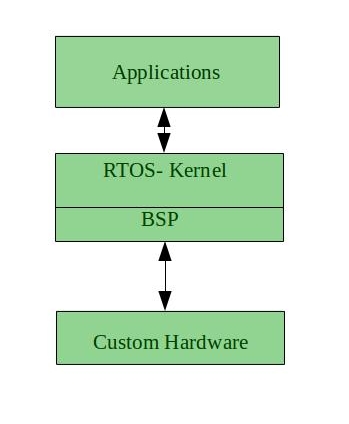


**NETWORK OPERATING SYSTEM**

These systems run on a server and provide the capability to manage data, users, groups, security, applications, and other networking functions. These type of operating systems allow shared access of files, printers, security, applications, and other networking functions over a small private network. One more important aspect of Network Operating Systems is that all the users are well aware of the underlying configuration, of all other users within the network, their individual connections etc. and that’s why these computers are popularly known as **tightly coupled systems**.

**Examples of Network Operating System are:** Microsoft Windows Server 2003, Microsoft Windows Server 2008, UNIX, Linux, Mac OS X, Novell NetWare, and BSD etc.



**REAL-TIME OPERATING SYSTEM**

It is defined as an operating system known to give max for each of the critical operations that it performs, like OS calls and interrupt handling.

**Real-time systems** are used when there are time req. are very strict like missile systems, air traffic control systems, robots etc.

**Two types of Real-Time Operating System are as follows:**

* **Hard Real-Time Systems:**  
  These OSs are meant for the applications where time constraints are very strict and even the shortest possible delay is not acceptable. These systems are built for saving life like automatic parachutes or air bags which are required to be readily available in case of any accident. Virtual memory is almost never found in these systems.
* **Soft Real-Time Systems:**  
  These OSs are for applications where for time-constraint is less strict.

## **SIMPLE BATCH SYSTEMS**

* In this type of system, there is no direct interaction between user and the computer.
* The user has to submit a job (written on cards or tape) to a computer operator.
* Then computer operator places a batch of several jobs on an input device.
* Jobs are batched together by type of languages and requirement.
* Then a special program, the monitor, manages the execution of each program in the batch.
* The monitor is always in the main memory and available for execution.

## **MULTIPROGRAMMING BATCH SYSTEMS**

* In this the operating system picks up and begins to execute one of the jobs from memory.
* Once this job needs an I/O operation operating system switches to another job (CPU and OS always busy).
* Jobs in the memory are always less than the number of jobs on disk (Job Pool).
* If several jobs are ready to run at the same time, then the system chooses which one to run through the process of CPU Scheduling.
* In Non-multiprogrammed system, there are moments when CPU sits idle and does not do any work.
* In Multiprogramming system, CPU will never be idle and keeps on processing.

## **CLUSTERED SYSTEMS**

* Like parallel systems, clustered systems gather together multiple CPUs to accomplish computational work.
* Clustering is usually performed to provide **high availability**.
* A layer of cluster software runs on the cluster nodes. Each node can monitor one or more of the others.
* If the monitored machine fails, the monitoring machine can take ownership of its storage, and restart the application(s) that were running on the failed machine. The failed machine can remain down, but the users and clients of the application would only see a brief interruption of service.

## **MULTIPROCESSOR SYSTEMS**

A Multiprocessor system consists of several processors that share a common physical memory. Multiprocessor system provides higher computing power and speed. In multiprocessor system all processors operate under single operating system. Multiplicity of the processors and how they do act together are transparent to the others.

## **DESKTOP SYSTEMS**

Earlier, CPUs and PCs lacked the features needed to protect an operating system from user programs. PC operating systems therefore were neither multiuser nor multitasking. However, the goals of these operating systems have changed with time; instead of maximizing CPU and peripheral utilization, the systems opt for maximizing user convenience and responsiveness.

These systems are called **Desktop Systems** and include PCs running Microsoft Windows and Apple Macintosh. Operating systems for these computers have benefited in several ways from the development of operating systems for mainframes.

**HANDHELD SYSTEMS**

Handheld systems include **Personal Digital Assistants (PDAs)**, such as Palm-Pilots or Cellular Telephones with connectivity to a network such as the Internet. They are usually of limited size due to which most handheld devices have a small amount of memory, include slow processors, and feature small display screens.

# **DIFFERENT FLAVOURS OF LINUX**

Linux comes in many different guises. The basic system is the same, but the look and feel and the subsystems around it are different. Each version is produced by a different organisation with its own ethos and aims.

**UBUNTU**

<http://www.ubuntulinux.org/>

Currently the most popular distro, GUI driven Linux, based on the [Debian](http://www.debian.org/) core. The most user-friendly version for Linux newbies Unusually, Canonical provides a free [server version](http://www.linuceum.com/Server/srvUbuntuDownload.php) of Ubuntu for non- commercial use

**FEDORA**

<http://fedoraproject.org/>

Generally, Fedora requires a little more tinkering than Ubuntu or Mint, with the user having to resort to the [command line](http://www.linuceum.com/Distros/osCmdLineWhy.php) more frequently, but is more reliable and ideally suited to the slightly more adventurous user

**LINUX MINT**

<http://www.linuxmint.com/>

Currently in third place is another user friendly version of Linux. Mint adds various and comes with more applications pre-installed.

**PUPPY LINUX**

[http://puppylinux.org/main/Overview and Getting Started.htm](http://puppylinux.org/main/Overview%20and%20Getting%20Started.htm)

Small footprint (100Mb) Linux, suitable for old hardware or low specification machines. Can run easily from a [USB memory stick](http://www.linuceum.com/Distros/osInstallPuppyLinux.php) or Live CD/DVD. Includes a full [desktop GUI](http://www.linuceum.com/Distros/osDesktopPuppy.php), browser. Great for old / low specification hardware

**TINYCORE**

<http://www.tinycorelinux.com/>

Very small footprint (10Mb) Linux, suitable for old hardware or low specification machines / embedded devices. It ships with a [minimal desktop GUI](http://www.linuceum.com/Distros/osDesktopTinyCore.php) but no applications. Ideal for ancient hardware or occasional use

**MEPIS LINUX**

<http://www.mepis.org/>

There are two versions: the full version is known as Simply MEPIS but there is also a version called AntiX, which is suitable for old hardware or low specification machines. Both can run from a hard drive or direct from a Live CD/DVD.

**ZORIN OS**

<http://zorin-os.com/index.html>

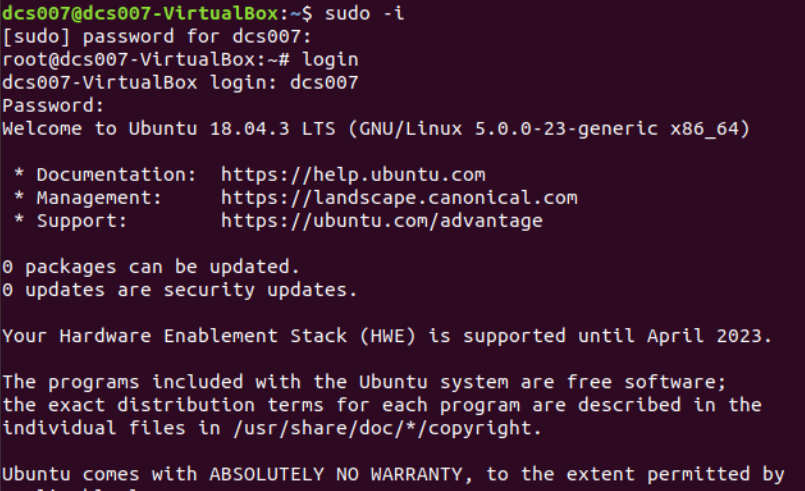
It's is easing the move from **Windows** to Linux. It comes with a full-set of applications pre-installed. Once again, well suited for those coming from a **Windows** background.

**PRACTICAL-2**

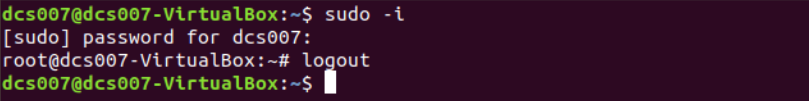
Study of Unix Architecture and the following Unix commands with option:

**USER ACCESS COMMANDS**

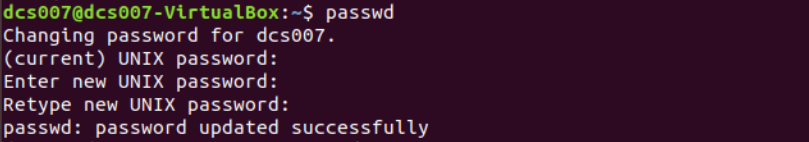
**LOGIN**:- The “login” command can be executed once we enter the root menu . The login command gives the last login and other such basic system information.



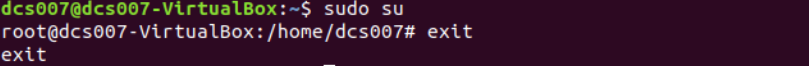
**LOGOUT:-** The “logout” command is used to logout or exit from the system root menu.



**PASSWD:-** The “passwd” command is used to change password of the current user.

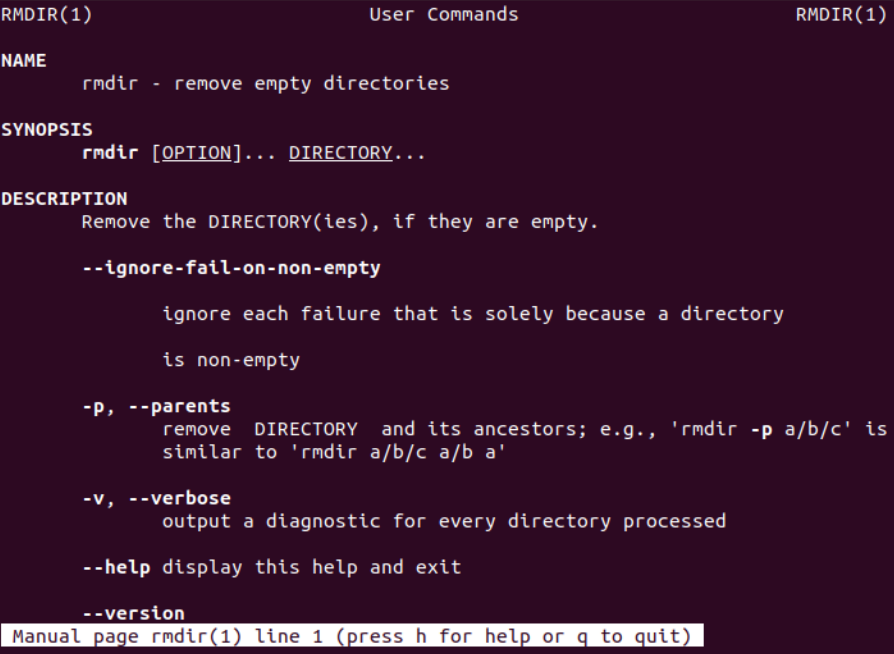


**EXIT:-** “exit ” command is used to exit from the current execution process.

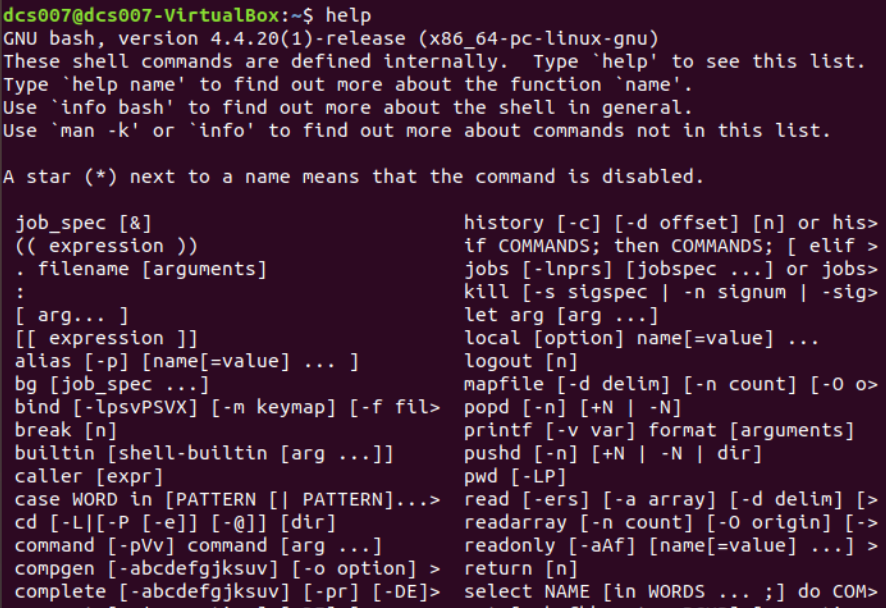


**USER HELP COMMANDS**

**MAN:-** The “man ” command gives the overall manual information for the command.

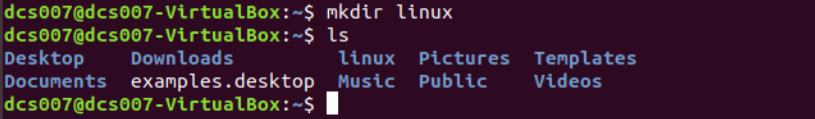


**HELP :-** The “help” command gives the detailed information about the commands.

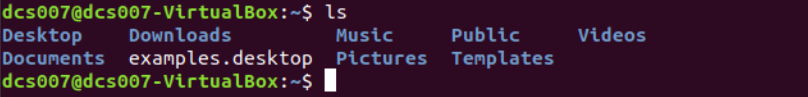


**DIRECTORY COMMANDS**

**MKDIR:-** The “mkdir” command is used to create a directory.



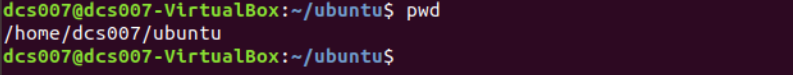
**RMDIR:-** The “rmdir” command is used to delete an existing directory.



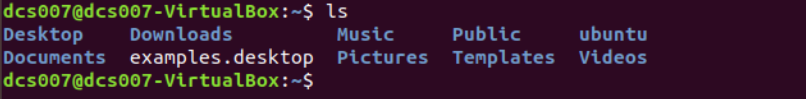
**CD:-** The “cd” command is used to change the current working directory.



**PWD:-** The “pwd” command is used to get the full path of current working directory.

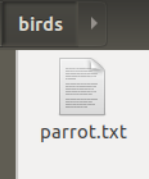


**LS:-** The “ls ” command gives the list of all the sub-directories and files present in the current working directory.



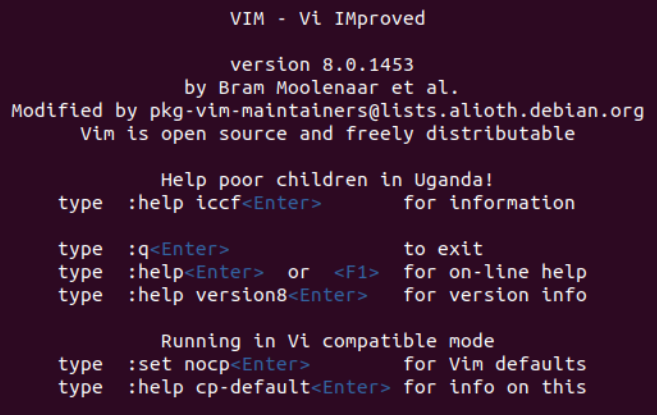
**MV:-** The “mv” command is used to move one directory to another directory . It is also used to rename any given directory.





**EDITOR COMMANDS**

**VI:-** The “vi” command is used to open the in-terminal vi editor in which can write or edit the data into files.



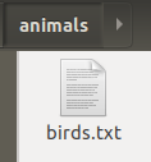
**GEDIT:-** It is powerful general purpose text editor in Linux.



**ED:-** It is used for launching the ed text editor which is a line-based text editor with minimal interface which makes it less complex for working on text files.



**FILE HANDLING COMMANDS**

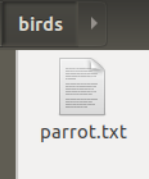


**CP:-** “cp” command is used to copy file contents to another file.

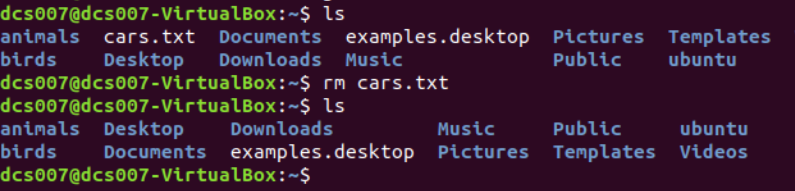


**MV:-** The “mv” command is used to move files from one directory to another.

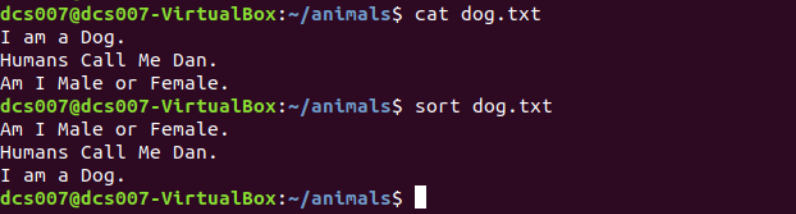




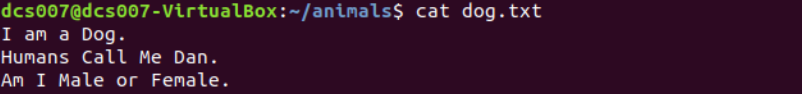
**RM:-** The “rm” command is used to remove(delete) files.



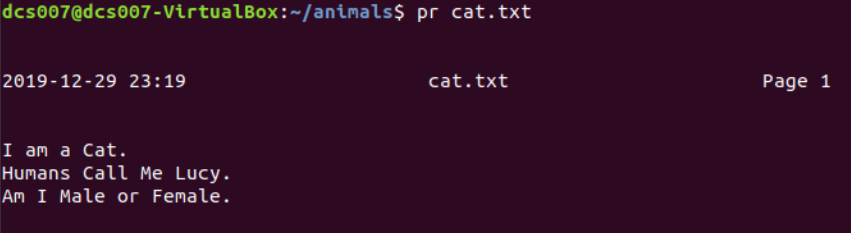
**SORT:-** The “sort” command is used to sort the data contents line wise in the file.



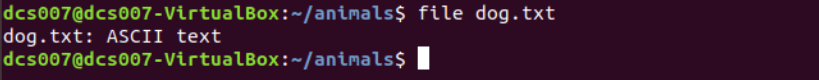
**CAT:-** “cat” command displays the file contents to on the output terminal.



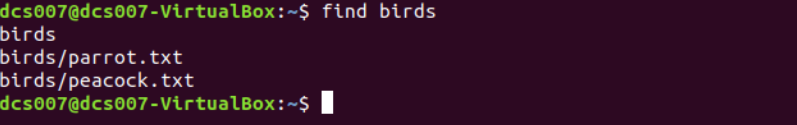
**PR:-** The “pr” command is used to display file details and its contents.



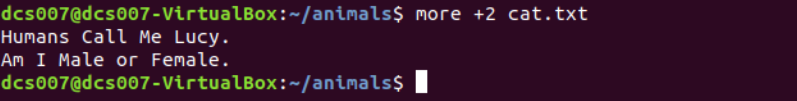
**FILE:-** “file” command returns the data type of the data stored in the file.



**FIND:-** The “find” command returns path to each and every file and folder saved in the directory entered.



**MORE:-** “more” command is used to sort and display a particular section of the content from whole file.



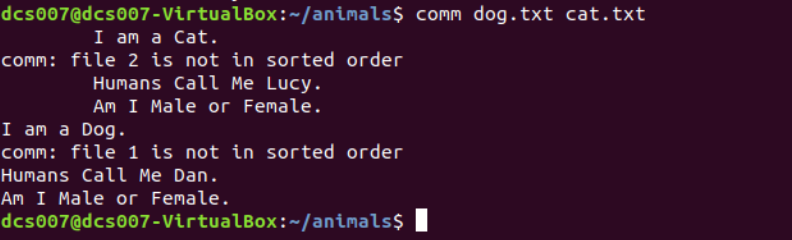
**CMP:-** When “cmp” is used for comparison between two files, it reports the location of the first mismatch to the screen if difference is found and if no difference is found i.e the files compared are identical.



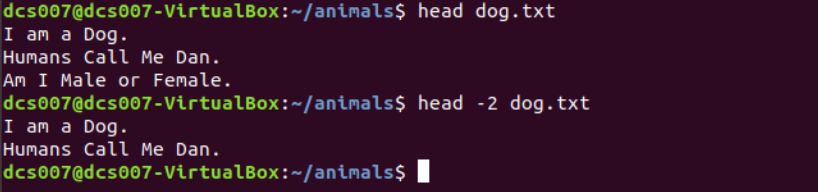
**DIFF:-** The “diff” command compares the files line by line and tells the user about what changes need to be made in the files to make both the files identical.



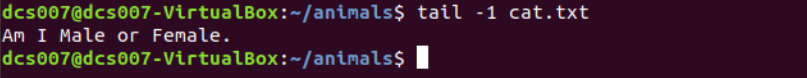
**COMM:-** The “comm” command compares two sorted files line by line and writes three columns to standard output. These columns show lines that are unique to files one, lines that are unique to file two and lines that are shared by both files. It also supports suppressing column outputs and comparing lines without case sensitivity.



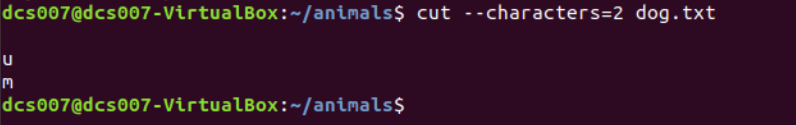
**HEAD:-** The “head” command returns the particular (entered) number of lines from the beginning of the file or any document.



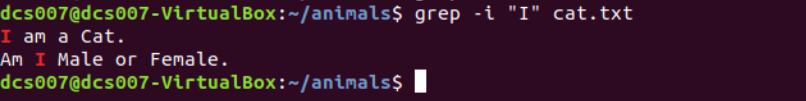
**TAIL:-** The “tail” command returns the particular (entered) number of lines from the ending of the file or any document.



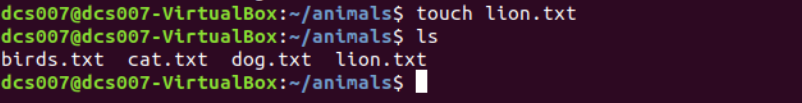
**CUT:-** The “cut” command in UBUNTU is a command for cutting out the sections from each line of files and writing the result to standard output.



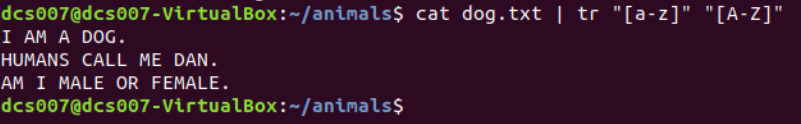
**GREP:-** The “grep” command is used to search text file for patterns. A pattern can be a word, text, numbers and more. It is one of the most useful commands on Debian/Ubuntu/ Linux and Unix like operating systems.



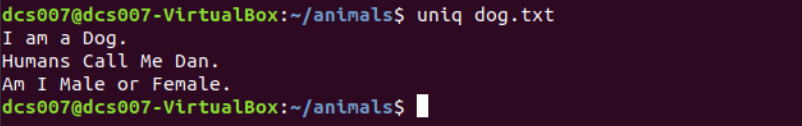
**TOUCH:-** The “touch” command is used to create new, empty files. It is also used to change the timestamps (i.e., dates and times of the most recent access and modification) on existing files and directories.



**TR:-** The tr command in UNIX is a command line utility for translating or deleting characters. It supports a range of transformations including uppercase to lowercase, squeezing repeating characters, deleting specific characters and basic find and replace.

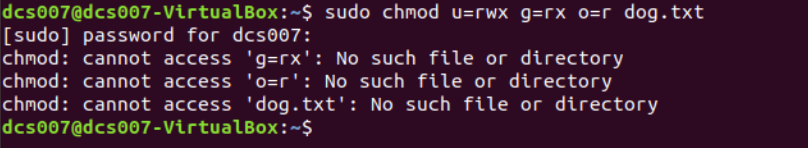


**UNIQ:-** The “**uniq**” command in Linux is a command line utility that reports or filters out the repeated lines in a file.



**SECURITY AND PROTECTION COMMANDS**

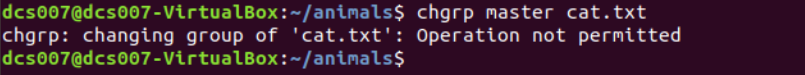
**CHMOD:-** The “chmod” command is used to change the access mode of a file.  
The name is an abbreviation of change mode.



**CHOWN:-** It is used to change the ownership and group of files, directories and links. By default, the owner of a file system object is the user that created it.

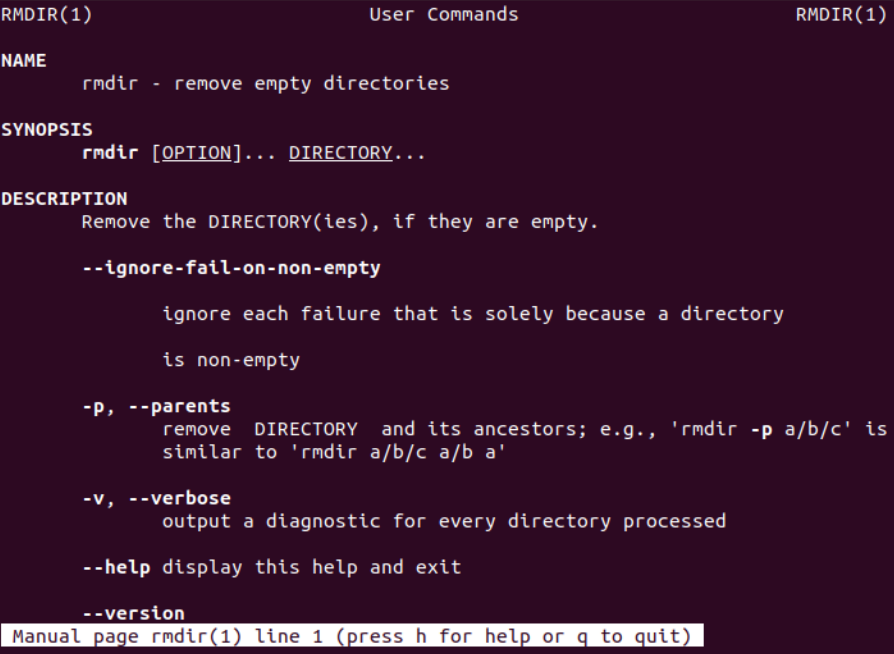


**CHGRP:-** It is used to change the Group of File or Directory. All Files in Linux basically belongs to an owner and a group.

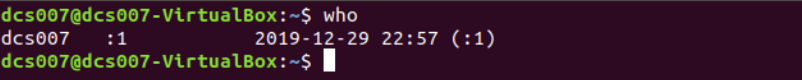


**INFORMATION COMMANDS**

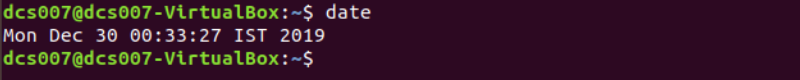
**MAN:-**The “man” command in Linux is used to display the user manual of any command that we can run on the terminal.



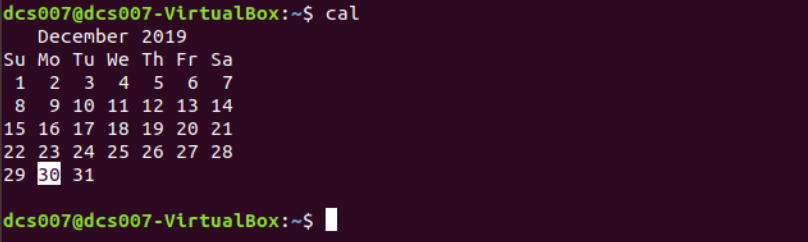
**WHO:-**The “who” command shows the information of all the users who all are logged into the system.



**DATE:-** The date command displays the current date and time, including the abbreviated day name, abbreviated month name, day of the month, the time separated by colons, the time zone name, and the year.



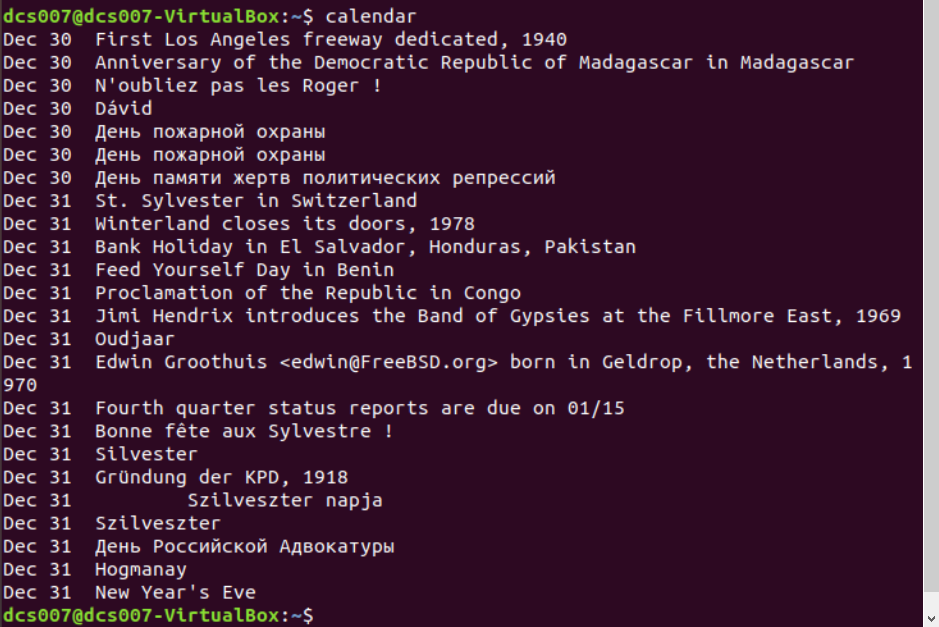
**CAL:-** Shows current month calendar on the terminal.



**TTY:-**The “tty” command displays the system information on the terminal.

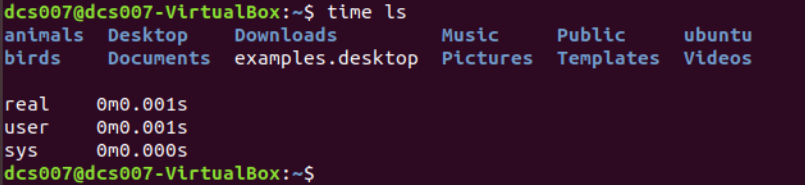


**CALENDAR:-**The “calendar” command gives the occasions of the current and the following days.

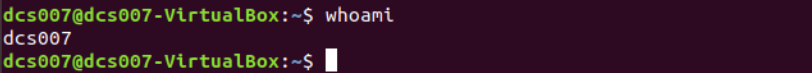


**TIME:-**The “Time” command gives the execution time for each command whenever entered to the terminal input.

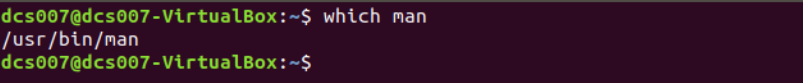
**BC:-**The “bc” command gives the copyright version and warranty details of the operating system.



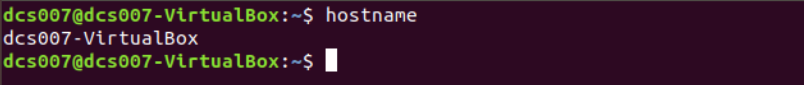
**WHOAMI:-**The “whoami” command returns the username of the current user.



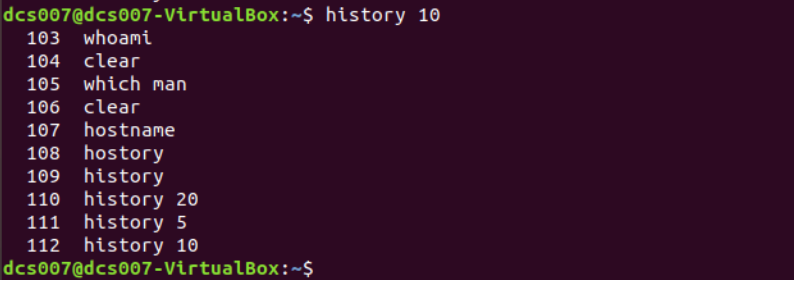
**WHICH:-** “Which” command in Linux is a command which is used to locate the executable file associated with the given command by searching it in the path environment variable.



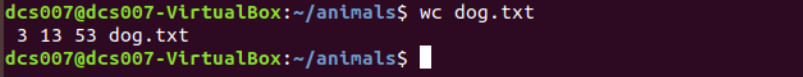
**HOSTNAME:-**“hostname” command returns the host name on which the operating system is working.



**HISTORY:-**The “history” command returns the history of all the commands that are executed on the terminal.

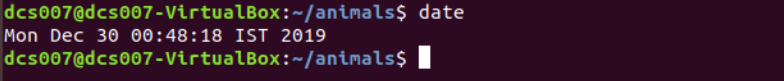


**WC:-** It is used to find out number of lines, word count, byte and characters count in the files specified in the file arguments.

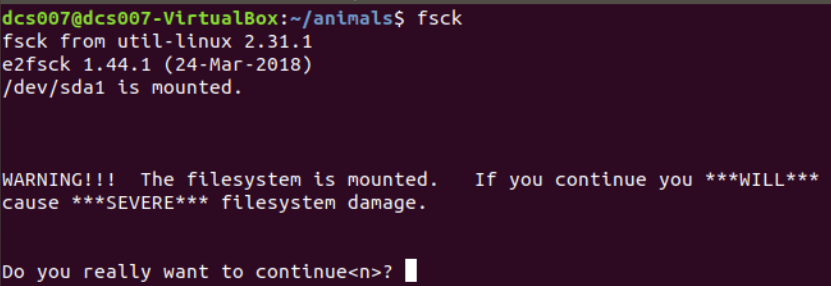


**SYSTEM ADMINISTRATOR COMMANDS**

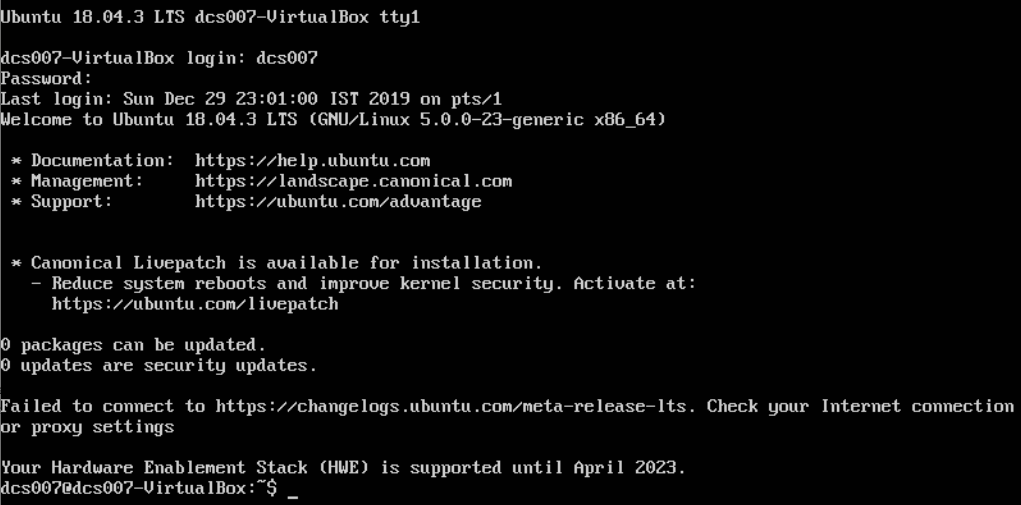
**DATE:-**The “date” command when used with proper options ,can be used to modify the system date and timezone settings.



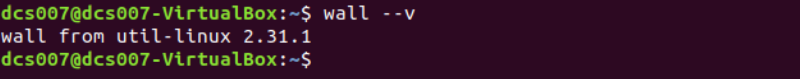
**FSCK:-** “Fsck” stands for “File System Consistency checK”. The use of “fsck” command is that you can use it to check and repair your filesystem.



**INIT 2:-**There are basically 8 runlevels in ubuntu . The system is present in either of any runlevel at a time. The system is said to be in init 2 level when there is  No network but multitasking support is present .

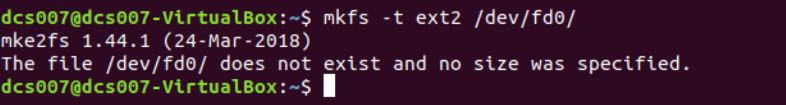


**WALL:-**  The “wall” command displays a message, or the contents of a file, or otherwise its standard input, on the terminals of all currently logged in users. The command will wrap lines that are longer than 79 characters. Short lines are whitespace padded to have 79 characters.

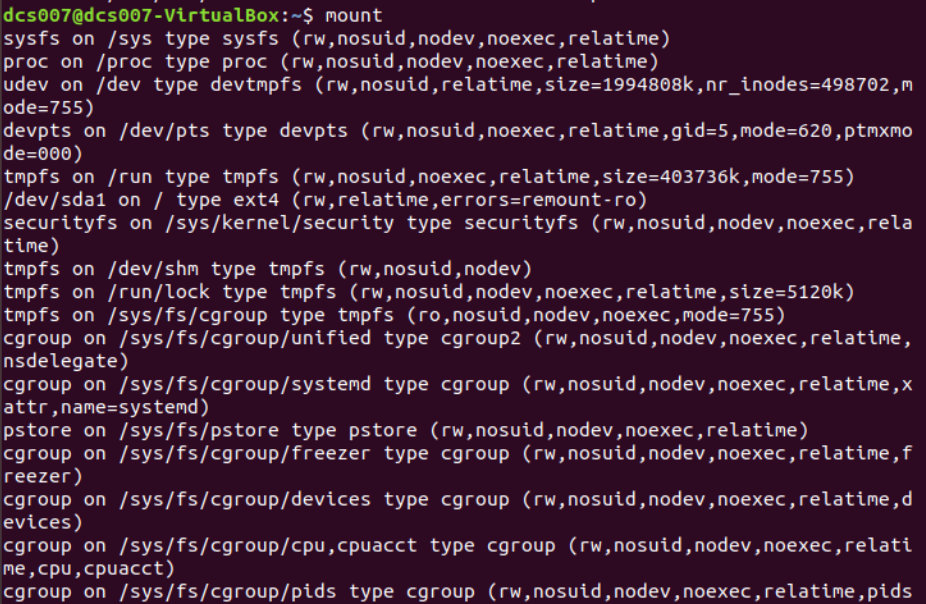


**SHUTDOWN:-**The “ shutdown” command is used to shutdown the device directly from the terminal.

**MKFS:-**The “mkfs” is used to build a Linux file system on a device, usually a [hard disk](https://www.computerhope.com/jargon/h/harddriv.htm) [partition](https://www.computerhope.com/jargon/p/partition.htm).

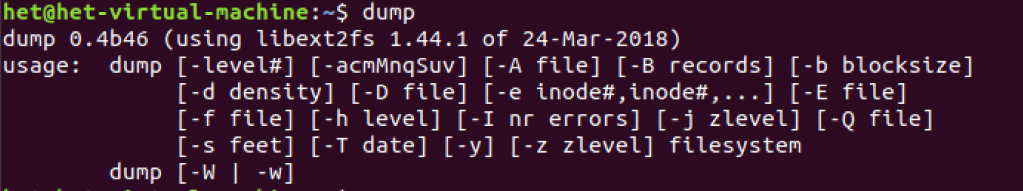


**MOUNT:-**The “mount” command serves to attach or mount the file system found on some device to the main file system of the current device.



**UNMOUNT:-** The “unmount” command serves to detach any file system found on some device from the main file system of the current device.

**DUMP:-**The dump command either dumps the whole file system or creates the system backup of the particular file system.

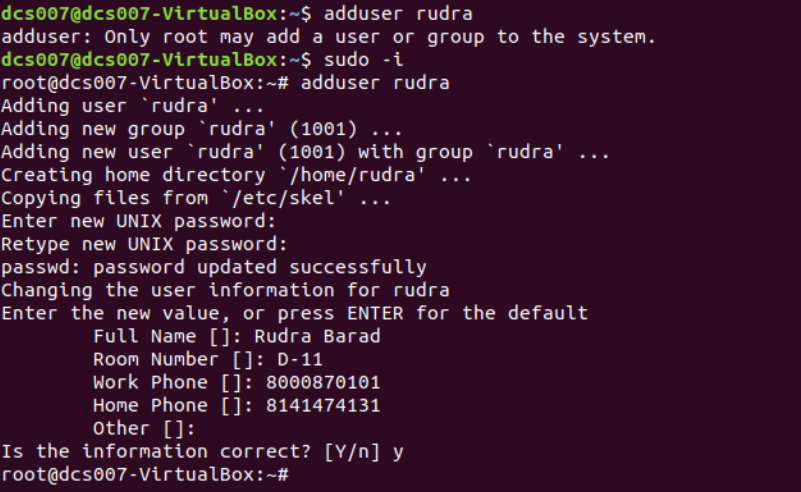


**RESTORE:-** “R**estore”** command in Linux system is used for restoring files from a backup created using dump. The restore command performs the exact inverse function of dump.



**TAR:-** The tar command stands for tape achieve, which is the most commonly used tape drive backup command used by the Linux/Unix system. It allows for you to quickly access a collection of files and placed them into a highly compressed archive file commonly called tarball, or tar, gzip, and bzip in Linux.

**ADDUSER:-** In Linux, a “adduser” command is a low-level utility that is used for adding/creating user accounts in **Linux** and other **Unix-like** operating systems.

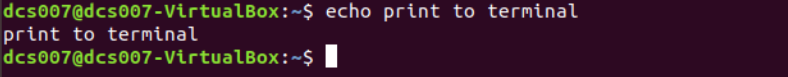


**USERDEL:-**The “userdel” command is similar to rmuser command and are both used to remove user from the current operating system.



**TERMINAL COMMANDS**

**ECHO:-** The “echo” command prints all the text that is written after it in the terminal.



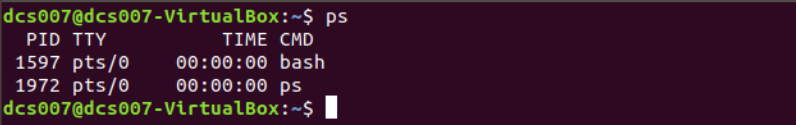
**PRINTF:-**The “printf” command is similar to the cho command but the printf command prints only one word after it in the terminal.



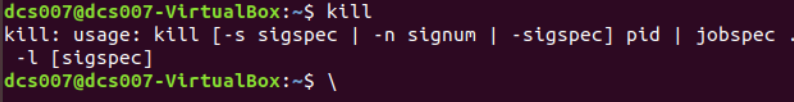
**CLEAR :-**The “clear” command clears the whole terminal screen display.

**PROCESS COMMANDS**

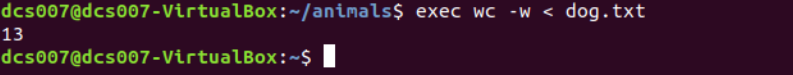
**PS:-** “ps” command is used to list the currently running processes and their PIDs along with some other information depends on different options.



**KILL:-**The “kill” command is used to kill or end any process using the process id.



**EXEC:-** The “**exec”**command in Linux is used to execute a command from the bash itself.



**PRACTICAL-3**

**AIM**

**3.1 Write a shell script which calculates nth Fibonacci number where n will be provided as input when prompted.**

**3.2 Write a shell script which takes one number from user and finds factorial of a given number.**

**3.3 Write a shell script to sort the number in ascending order. (Using array).**

**3.1 PROGRAM CODE**

# Program for Fibonacci

# Series

# Static input fo N

N=6

# First Number of the

# Fibonacci Series

a=0

# Second Number of the

# Fibonacci Series

b=1

echo "The Fibonacci series is : "

for (( i=0; i<N; i++ ))

do

    echo -n "$a "

    fn=$((a + b))

    a=$b

    b=$fn

done

**OUTPUT**



**3.2 PROGRAM CODE**

#shell script for factorial of a number

#factorial using while loop

echo "Enter a number"

read num

fact=1

while [ $num -gt 1 ]

do

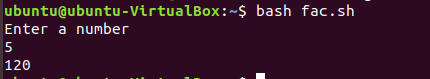
fact=$((fact \* num)) #fact = fact \* num

num=$((num - 1)) #num = num - 1

done

echo $fact

**OUTPUT**



**3.3 PROGRAM CODE**

# Declare the array of 5 subscripts to hold 5 numbers

#

echo "enter maximum number"

read n

# taking input from user

echo "enter Numbers in array:"

for (( i = 0; i < $n; i++ ))

do

read nos[$i]

done

## Prints the number befor sorting

#

echo "Original Numbers in array:"

for (( i = 0; i <= 4; i++ ))

do

echo ${nos[$i]}

done

#

# Now do the Sorting of numbers

#

for (( i = 0; i <= 4 ; i++ ))

do

for (( j = $i; j <= 4; j++ ))

do

if [ ${nos[$i]} -gt ${nos[$j]} ]; then

t=${nos[$i]}

nos[$i]=${nos[$j]}

nos[$j]=$t

fi

done

done

#

# Print the sorted number

#

echo -e "\nSorted Numbers in Ascending Order:"

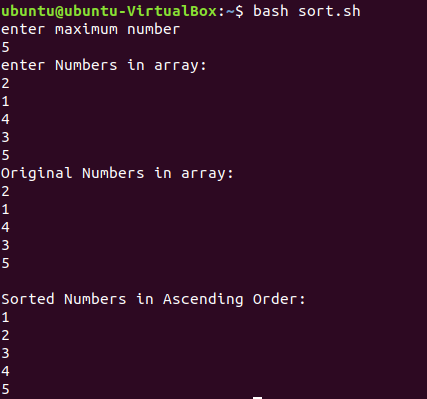
for (( i=0; i <= 4; i++ ))

do

echo ${nos[$i]}

done

**OUTPUT**



**PRACTICAL-4**

**AIM**

**Write programs using the following system calls of UNIX operating system: fork, exec, getpid, exit, wait, stat, readdir, opendir.**

**1. Write a program to execute fork() and find out the process id by getpid() system call.**

**2. Write a program to execute following system call fork(), execl(), getpid(), exit(), wait() for a process.**

**3. Write a program to find out status of named file (program of working stat() system call).**

**4. Write a program for “ls” command implementation using opendir() & readdir() system call**

**PROGRAM CODE:**

**1.**

#include <stdio.h>

#include <sys/types.h>

#include <unistd.h>

int main(void)

{

pid\_t process\_id;

pid\_t p\_process\_id;

process\_id = getpid();

p\_process\_id = getppid();

printf("The process id: %d\n",process\_id);

printf("The process id of parent function: %d\n",p\_process\_id);

return 0;

}

**2.**

#include<stdio.h>

#include<sys/stat.h>

int main()

{

struct stat sfile;

stat("stat.c", &sfile

printf("st\_mode = %o", sfile.st\_mode);

return 0;

}

**3.**

#include<stdio.h>

#include<unistd.h>

Int main(void)

{

printf("Before fork\n");

fork();

printf("after fork\n");

}

**4.**

#include <sys/types.h>

#include <sys/stat.h>

#include <unistd.h>

#include <stdio.h>

#include <stdlib.h>

#include <dirent.h>

int main(int argc, char\* argv[])

{

DIR \*mydir;

struct dirent \*myfile;

struct stat mystat;

char buf[512];

mydir = opendir(argv[1]);

while((myfile = readdir(mydir)) != NULL)

{

sprintf(buf, "%s/%s", argv[1], myfile->d\_name);

stat(buf, &mystat);

printf("%zu",mystat.st\_size);

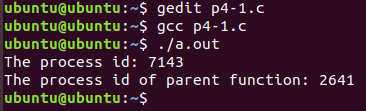
printf(" %s\n", myfile->d\_name);

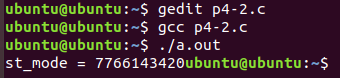
}

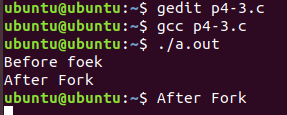
closedir(mydir);

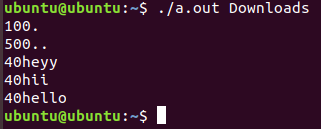
}

**OUTPUT:**









**PRACTICAL-5**

**AIM**

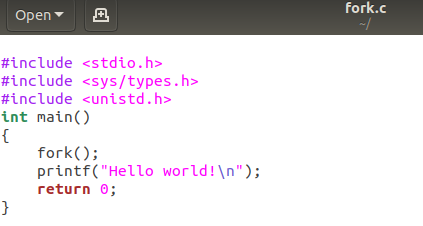
**Process control system calls:**

**A. The demonstration of fork()**

**B. execve() and wait() system calls along with zombie and orphan states.**

**PROGRAM CODE:**

**A.**



**B.**

1.

#include<stdio.h>

#include<unistd.h>

#include<sys/types.h>

#include<sys/wait.h>

void quickSort(int [],int ,int );

int partition(int [],int ,int );

void mergeSort(int [],int ,int );

void merge(int [],int ,int ,int ,int );

int main()

{

int i,j,n;

int \*status=NULL;

int arr[30];

printf("\nEnter the number of elements:");

scanf("%d",&n);

for(i=0;i<n;i++)

{

scanf("%d",&arr[i]);

}

pid\_t pid;

pid=fork();

if(pid==0)

{

printf("\n\t This is child process. ");

printf("\n\t My process id is : %d", getpid());

printf("\n\t My Parent process id is : %d", getppid());

quickSort(arr,0,n-1);

printf("\nQuicksort");

for(i=0;i<n;i++)

printf(" %d",arr[i]);

printf("\n\n");

}

else

{

printf("\n\n\t Parent process resumed after the execution of child process with PID %d", pid);

printf("\n\t My process id is : %d", getpid());

printf("\n\t My Parent process id is : %d", getppid());

mergeSort(arr,0,n-1);

printf("\nMergesort:");

for(i=0;i<n;i++)

printf(" %d",arr[i]);

printf("\n\n");

pid=wait(status);

}

}

void quickSort(int arr[],int low,int high)

{

int j;

if(low<high)

{

j=partition(arr,low,high);

quickSort(arr,low,j-1);

quickSort(arr,j+1,high);

}

}

int partition(int arr[],int low,int high)

{

int i,j,temp,pivot;

pivot=arr[low];

i=low;

j=high+1;

do

{

do

i++;

while(arr[i]<pivot && i<=high);

do

j--;

while(arr[j]>pivot);

if(i<j)

{

temp=arr[i];

arr[i]=arr[j];

arr[j]=temp;

}

}

while(i<j);

arr[low]=arr[j];

arr[j]=pivot;

return(j);

}

void mergeSort(int arr[],int low,int high)

{

int mid;

if(low<high)

{

mid=(low+high)/2;

mergeSort(arr,low,mid);

mergeSort(arr,mid+1,high);

merge(arr,low,mid,mid+1,high);

}

}

void merge(int arr[],int i1,int j1,int i2,int j2)

{

int temp[50];

int i,j,k;

i=i1;

j=i2;

k=0;

while(i<=j1 && j<=j2)

{

if(arr[i]<arr[j])

temp[k++]=arr[i++];

else

temp[k++]=arr[j++];

}

while(i<=j1)

temp[k++]=arr[i++];

while(j<=j2)

temp[k++]=arr[j++];

for(i=i1,j=0;i<=j2;i++,j++)

arr[i]=temp[j];

}

2.

#include<stdio.h>

#include<unistd.h>

#include<stdlib.h>

#include<string.h>

#include<sys/types.h>

int main(int argc,char \*argv[])

{

int i,j,c,ele;

int arr[argc];

for(i=0;i<argc-1;i++)

{

int n=atoi(argv[i]);

arr[i]=n;

}

ele=atoi(argv[i]);

i=0;

j=argc-1;

c=(i+j)/2;

while(ele!=arr[c] && i<=j)

{

if(arr[c]<ele)

i=c+1;

else

j=c-1;

c=(i+j)/2;

}

if(i<=j)

printf("Elemets found");

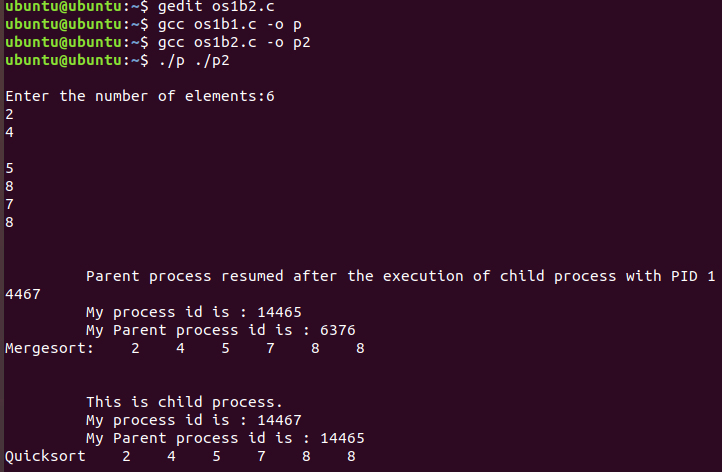
else

printf("Not found");

}

**OUTPUT:**





**PRACTICAL-6**

**AIM**

**Write a C program in UNIX to implement Process scheduling algorithms and compare.**

**A. First Come First Serve (FCFS) Scheduling**

**B. Shortest-Job-First (SJF) Scheduling**

**C. Priority Scheduling (Non-preemption) after completion extend on Preemption.**

**D. Round Robin(RR) Scheduling**

**PROGRAM CODE:**

(A)

#include<stdio.h>

void findWaitingTime(int processes[], int n, int bt[], int wt[])

{

wt[0] = 0;

for (int i = 1; i < n ; i++ )

wt[i] = bt[i-1] + wt[i-1] ;

}

void findTurnAroundTime( int processes[], int n, bt[], int wt[], int tat[])

{

// bt[i] + wt[i]

for (int i = 0; i < n ; i++)

tat[i] = bt[i] + wt[i];

}

void findavgTime( int processes[], int n, int bt[])

{

int wt[n], tat[n], total\_wt = 0, total\_tat = 0;

findWaitingTime(processes, n, bt, wt);

findTurnAroundTime(processes, n, bt, wt, tat);

printf("Processes Burst time Waiting time Turn around time\n");

for (int i=0; i<n; i++)

{

total\_wt = total\_wt + wt[i];

total\_tat = total\_tat + tat[i];

printf(" %d ",(i+1));

printf(" %d ", bt[i] );

printf(" %d",wt[i] );

printf(" %d\n",tat[i] );

}

int s=(float)total\_wt / (float)n;

int t=(float)total\_tat / (float)n;

printf("Average waiting time = %d",s);

printf("\n");

printf("Average turn around time = %d ",t);

}

// Driver code

int main()

{

//process id's

int processes[] = { 1, 2, 3};

int n = sizeof processes / sizeof processes[0];

int burst\_time[] = {10, 5, 8};

findavgTime(processes, n, burst\_time);

return 0;

}

(B)

#include<iostream>

using namespace std;

int mat[10][6];

void swap(int \*a, int \*b)

{

int temp = \*a;

\*a = \*b;

\*b = temp;

}

void arrangeArrival(int num, int mat[][6])

{

for(int i=0; i<num; i++)

{

for(int j=0; j<num-i-1; j++)

{

if(mat[j][1] > mat[j+1][1])

{

for(int k=0; k<5; k++)

{

swap(mat[j][k], mat[j+1][k]);

} }

} }

}

void completionTime(int num, int mat[][6])

{

int temp, val;

mat[0][3] = mat[0][1] + mat[0][2];

mat[0][5] = mat[0][3] - mat[0][1];

mat[0][4] = mat[0][5] - mat[0][2];

for(int i=1; i<num; i++)

{

temp = mat[i-1][3];

int low = mat[i][2];

for(int j=i; j<num; j++)

{

if(temp >= mat[j][1] && low >= mat[j][2])

{

low = mat[j][2];

val = j;

}

}

mat[val][3] = temp + mat[val][2];

mat[val][5] = mat[val][3] - mat[val][1];

mat[val][4] = mat[val][5] - mat[val][2];

for(int k=0; k<6; k++)

{

swap(mat[val][k], mat[i][k]);

} }

}

int main()

{

int num, temp;

cout<<"Enter number of Process: ";

cin>>num;

cout<<"...Enter the process ID...\n";

for(int i=0; i<num; i++)

{

cout<<"...Process "<<i+1<<"...\n";

cout<<"Enter Process Id: ";

cin>>mat[i][0];

cout<<"Enter Arrival Time: ";

cin>>mat[i][1];

cout<<"Enter Burst Time: ";

cin>>mat[i][2];

}

cout<<"Before Arrange...\n";

cout<<"Process ID\tArrival Time\tBurst Time\n";

for(int i=0; i<num; i++)

{

cout<<mat[i][0]<<"\t\t"<<mat[i][1]<<"\t\t"<<mat[i][2]<<"\n";

}

arrangeArrival(num, mat);

completionTime(num, mat);

cout<<"Final Result...\n";

cout<<"Process ID\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time\n";

for(int i=0; i<num; i++)

{

cout<<mat[i][0]<<"\t\t"<<mat[i][1]<<"\t\t"<<mat[i][2]<<"\t\t"<<mat[i][4]<<"\t\t"<<mat[i][5]<<"\n";

}

}

(C)

#include<bits/stdc++.h>

using namespace std;

struct Process

{

int pid;

int bt;

int priority;

};

bool comparison(Process a, Process b)

{

return (a.priority > b.priority);

}

void findWaitingTime(Process proc[], int n, int wt[])

{

wt[0] = 0;

for (int i = 1; i < n ; i++ )

wt[i] = proc[i-1].bt + wt[i-1] ;

}

void findTurnAroundTime( Process proc[], int n, int wt[], int tat[])

{

for (int i = 0; i < n ; i++)

tat[i] = proc[i].bt + wt[i];

}

void findavgTime(Process proc[], int n)

{

int wt[n], tat[n], total\_wt = 0, total\_tat = 0;

findWaitingTime(proc, n, wt);

findTurnAroundTime(proc, n, wt, tat);

cout << "\nProcesses "<< " Burst time "

<< " Waiting time " << " Turn around time\n";

for (int i=0; i<n; i++)

{

total\_wt = total\_wt + wt[i];

total\_tat = total\_tat + tat[i];

cout << " " << proc[i].pid << "\t\t"

<< proc[i].bt << "\t " << wt[i]

<< "\t\t " << tat[i] <<endl;

}

cout << "\nAverage waiting time = "

<< (float)total\_wt / (float)n;

cout << "\nAverage turn around time = "

<< (float)total\_tat / (float)n;

}

void priorityScheduling(Process proc[], int n)

{

sort(proc, proc + n, comparison);

cout<< "Order in which processes gets executed \n";

for (int i = 0 ; i < n; i++)

cout << proc[i].pid <<" " ;

findavgTime(proc, n);

}

int main()

{

Process proc[] = {{1, 10, 2}, {2, 5, 0}, {3, 8, 1}};

int n = sizeof proc / sizeof proc[0];

priorityScheduling(proc, n);

return 0;

}

(D)

#include<iostream>

using namespace std;

void findWaitingTime(int processes[], int n,

int bt[], int wt[], int quantum)

{

int rem\_bt[n];

for (int i = 0 ; i < n ; i++)

rem\_bt[i] = bt[i];

int t = 0;

while (1)

{

bool done = true;

for (int i = 0 ; i < n; i++)

{

if (rem\_bt[i] > 0)

{

done = false;

if (rem\_bt[i] > quantum)

{

t += quantum;

rem\_bt[i] -= quantum;

}

else

{

t = t + rem\_bt[i];

wt[i] = t - bt[i];

rem\_bt[i] = 0;

} }

} if (done == true)

break;

}

}

void findTurnAroundTime(int processes[], int n, int bt[], int wt[], int tat[])

{

// bt[i] + wt[i]

for (int i = 0; i < n ; i++)

tat[i] = bt[i] + wt[i];

}

void findavgTime(int processes[], int n, int bt[], int quantum)

{

int wt[n], tat[n], total\_wt = 0, total\_tat = 0;

findWaitingTime(processes, n, bt, wt, quantum);

findTurnAroundTime(processes, n, bt, wt, tat);

cout << "Processes "<< " Burst time "

<< " Waiting time " << " Turn around time\n";

for (int i=0; i<n; i++)

{

total\_wt = total\_wt + wt[i];

total\_tat = total\_tat + tat[i];

cout << " " << i+1 << "\t\t" << bt[i] <<"\t "

<< wt[i] <<"\t\t " << tat[i] <<endl;

}

cout << "Average waiting time = "

<< (float)total\_wt / (float)n;

cout << "\nAverage turn around time = "

<< (float)total\_tat / (float)n;

}

int main()

{

int processes[] = { 1, 2, 3};

int n = sizeof processes / sizeof processes[0];

int burst\_time[] = {10, 5, 8};

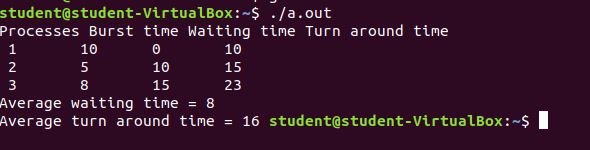
int quantum = 2;

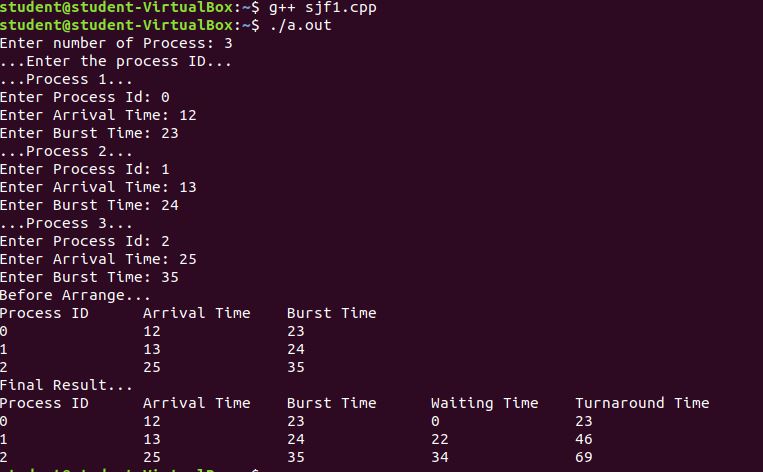
findavgTime(processes, n, burst\_time, quantum);

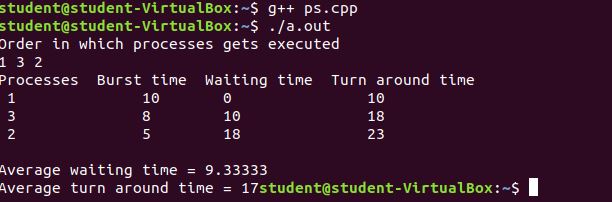
return 0;

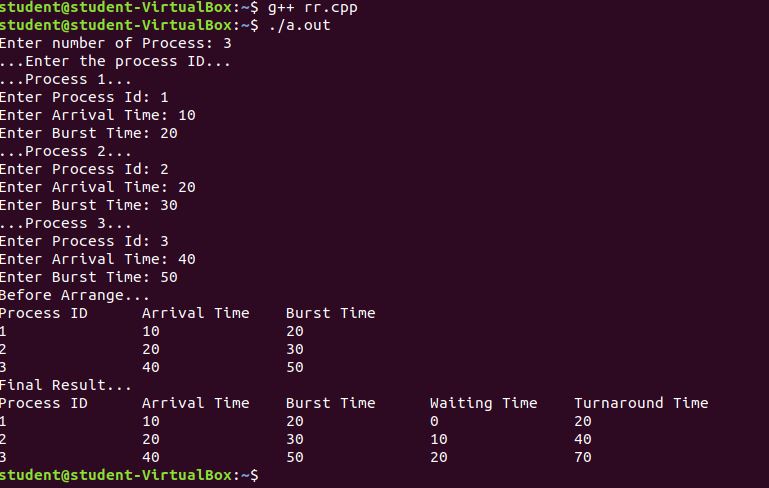
}

**OUTPUT:**

****







**PRACTICAL-7**

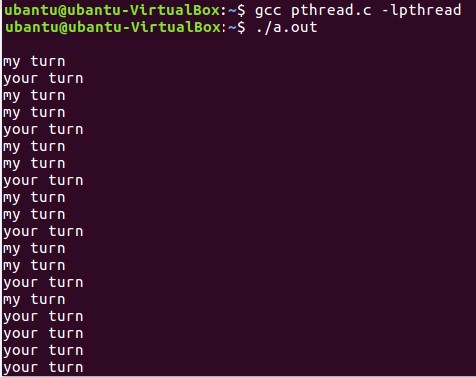
**AIM**

**Thread management using pthread library. Write a simple program to understand it.**

**PROGRAM CODE**



**OUTPUT**



**PRACTICAL-8**

**AIM**

**Write a C program in UNIX to implement Bankers algorithm for Deadlock Avoidance.**

**PROGRAM CODE**

#include <stdio.h>

int current[5][5], maximum\_claim[5][5], available[5];

int allocation[5] = {0, 0, 0, 0, 0};

int maxres[5], running[5], safe = 0;

int counter = 0, i, j, exec, resources, processes, k = 1;

int main()

{

printf("\nEnter number of processes: ");

scanf("%d", &processes);

for (i = 0; i < processes; i++)

{

running[i] = 1;

counter++;

}

printf("\nEnter number of resources: ");

scanf("%d", &resources);

printf("\nEnter Claim Vector:");

for (i = 0; i < resources; i++)

{

scanf("%d", &maxres[i]);

}

printf("\nEnter Allocated Resource Table:\n");

for (i = 0; i < processes; i++)

{

for(j = 0; j < resources; j++)

{

scanf("%d", &current[i][j]);

}

}

printf("\nEnter Maximum Claim Table:\n");

for (i = 0; i < processes; i++)

{

for(j = 0; j < resources; j++)

{

scanf("%d", &maximum\_claim[i][j]);

}

}

printf("\nThe Claim Vector is: ");

for (i = 0; i < resources; i++)

{

printf("\t%d", maxres[i]);

}

printf("\nThe Allocated Resource Table:\n");

for (i = 0; i < processes; i++)

{

for (j = 0; j < resources; j++)

{

printf("\t%d", current[i][j]);

}

printf("\n");

}

printf("\nThe Maximum Claim Table:\n");

for (i = 0; i < processes; i++)

{

for (j = 0; j < resources; j++)

{

printf("\t%d", maximum\_claim[i][j]);

}

printf("\n");

}

for (i = 0; i < processes; i++)

{

for (j = 0; j < resources; j++)

{

allocation[j] += current[i][j];

}

}

printf("\nAllocated resources:");

for (i = 0; i < resources; i++)

{

printf("\t%d", allocation[i]);

}

for (i = 0; i < resources; i++)

{

available[i] = maxres[i] - allocation[i];

}

printf("\nAvailable resources:");

for (i = 0; i < resources; i++)

{

printf("\t%d", available[i]);

}

printf("\n");

while (counter != 0)

{

safe = 0;

for (i = 0; i < processes; i++)

{

if (running[i])

{

exec = 1;

for (j = 0; j < resources; j++)

{

if (maximum\_claim[i][j] - current[i][j] > available[j])

{

exec = 0;

break;

}

}

if (exec)

{

printf("\nProcess%d is executing\n", i + 1);

running[i] = 0;

counter--;

safe = 1;

for (j = 0; j < resources; j++)

{

available[j] += current[i][j];

}

break;

}

}

}

if (!safe)

{

printf("\nThe processes are in unsafe state.\n");

break;

}

else

{

printf("\nThe process is in safe state");

printf("\nAvailable vector:");

for (i = 0; i < resources; i++)

{

printf("\t%d", available[i]);

}

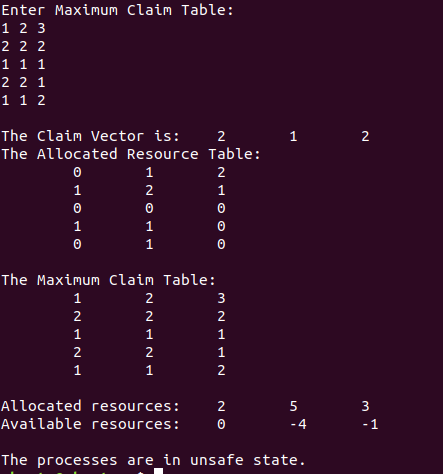
printf("\n");

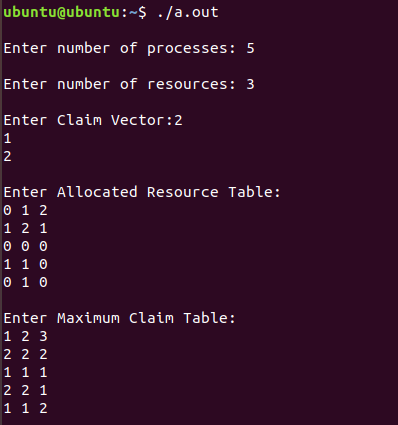
}

}

return 0;

}

**OUTPUT:**



**PRACTICAL-9**

**AIM**

**Write a C program in UNIX to perform Memory allocation algorithms and calculate Internal and External Fragmentation. (First Fit, Best Fit, Worst Fit)**

**PROGRAM CODE**

#include<stdio.h>

#define max 25

void main()

{

int frag[max],b[max],f[max],i,j,nb,nf,temp,highest=0;

static int bf[max],ff[max];

printf("\nEnter the number of blocks:");

scanf("%d",&nb);

printf("Enter the number of files:");

scanf("%d",&nf);

printf("\nEnter the size of the blocks:-\n");

for(i=1;i<=nb;i++)

{

printf("Block %d:",i);

scanf("%d",&b[i]);

}

printf("Enter the size of the files :-\n");

for(i=1;i<=nf;i++)

{

printf("File %d:",i);

scanf("%d",&f[i]);

}

for(i=1;i<=nf;i++)

{

for(j=1;j<=nb;j++)

{

if(bf[j]!=1) //if bf[j] is not allocated

{

temp=b[j]-f[i];

if(temp>=0)

if(highest<temp)

{

ff[i]=j;

highest=temp;

}

}

}

frag[i]=highest;

bf[ff[i]]=1;

highest=0;

}

printf("\nFile\_no:\tFile\_size :\tBlock\_no:\tBlock\_size:\tFragement");

for(i=1;i<=nf;i++)

printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d",i,f[i],ff[i],b[ff[i]],frag[i]);

}

**Best Fit:**

#include<stdio.h>

#define max 25

void main()

{

int frag[max],b[max],f[max],i,j,nb,nf,temp,lowest=10000;

static int bf[max],ff[max];

printf("\nEnter the number of blocks:");

scanf("%d",&nb);

printf("Enter the number of files:");

scanf("%d",&nf);

printf("\nEnter the size of the blocks:-\n");

for(i=1;i<=nb;i++)

{

printf("Block %d:",i);

scanf("%d",&b[i]);

}

printf("Enter the size of the files :-\n");

for(i=1;i<=nf;i++)

{

printf("File %d:",i);

scanf("%d",&f[i]);

}

for(i=1;i<=nf;i++)

{

for(j=1;j<=nb;j++)

{

if(bf[j]!=1)

{

temp=b[j]-f[i];

if(temp>=0)

if(lowest>temp)

{

ff[i]=j;

lowest=temp;

}

}

}

frag[i]=lowest;

bf[ff[i]]=1;

lowest=10000;

}

printf("\nFile No\tFile Size \tBlock No\tBlock Size\tFragment");

for(i=1;i<=nf && ff[i]!=0;i++)

printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d",i,f[i],ff[i],b[ff[i]],frag[i]);

}

**Worst Fit:**

#include<stdio.h>

#define max 25

void main()

{

int frag[max],b[max],f[max],i,j,nb,nf,temp;

static int bf[max],ff[max];

printf("\nEnter the number of blocks:");

scanf("%d",&nb);

printf("Enter the number of files:");

scanf("%d",&nf);

printf("\nEnter the size of the blocks:-\n");

for(i=1;i<=nb;i++)

{

printf("Block %d:",i);

scanf("%d",&b[i]);

}

printf("Enter the size of the files :-\n");

for(i=1;i<=nf;i++)

{

printf("File %d:",i);

scanf("%d",&f[i]);

}

for(i=1;i<=nf;i++)

{

for(j=1;j<=nb;j++)

{

if(bf[j]!=1)

{

temp=b[j]-f[i];

if(temp>=0)

{

ff[i]=j;

break;

}

}

}

frag[i]=temp;

bf[ff[i]]=1;

}

printf("\nFile\_no:\tFile\_size :\tBlock\_no:\tBlock\_size:\tFragement");

for(i=1;i<=nf;i++)

printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d",i,f[i],ff[i],b[ff[i]],frag[i]);

}

**OUTPUT**

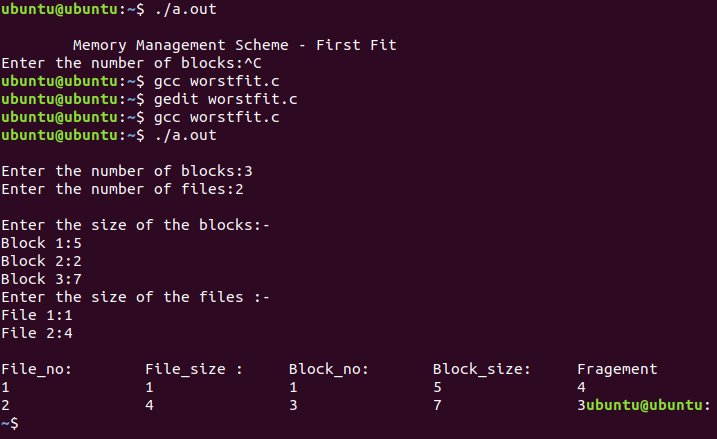
**First Fit**



**Best Fit**



**Worst Fit**



**PRACTICAL-10**

**AIM**

**Thread synchronization using counting semaphores and mutual exclusion using mutex.**

**PROGRAM CODE**

#include<stdio.h>

#include<semaphore.h>

#include<sys/types.h>

#include<pthread.h>

#include<unistd.h>

#include<stdlib.h>

#define BUFFER\_SIZE 1

pthread\_mutex\_t mutex;

sem\_t empty,full;

int buffer[BUFFER\_SIZE];

int counter;

pthread\_t tid;

void \*producer();

void \*consumer();

void insert\_item(int);

int remove\_item()

void initilize()

{ pthread\_mutex\_init(&mutex,NULL);

sem\_init(&full,0,0);

sem\_init(&empty,0,BUFFER\_SIZE); }

void \*producer() {

int item,wait\_time;

wait\_time=rand()%5;

sleep(wait\_time)%5;

item=rand()%10;

sem\_wait(&empty);

pthread\_mutex\_lock(&mutex);

printf("Producer produces %d\n\n",item);

insert\_item(item);

pthread\_mutex\_unlock(&mutex);

sem\_post(&full);

}

void \*consumer() {

int item,wait\_time;

wait\_time=rand()%5;

sleep(wait\_time);

sem\_wait(&full);

pthread\_mutex\_lock(&mutex);

item=remove\_item();

printf("Consumer consumes %d\n\n",item);

pthread\_mutex\_unlock(&mutex);

sem\_post(&empty);

}

void insert\_item(int item)

{ buffer[counter++]=item; }

int remove\_item()

{ return buffer[--counter]; }

int main() {

int n1,n2;

int i;

printf("Enter number of Producers");

scanf("%d",&n1);

printf("Enter number of Consumers");

scanf("%d",&n2);

initilize();

for(i=0;i<n1;i++)

pthread\_create(&tid,NULL,producer,NULL);

for(i=0;i<n2;i++)

pthread\_create(&tid,NULL,consumer,NULL);

sleep(5); }

**OUTPUT**

**PRACTICAL-11**

**AIM**

**Write a C program in UNIX to implement inter process communication (IPC) using Semaphore.**

**PROGRAM CODE**

#include<stdio.h>

#include<sys/ipc.h>

#include<sys/shm.h>

#include<sys/types.h>

#include<string.h>

#include<errno.h>

#include<stdlib.h>

#include<unistd.h>

#include<string.h>

#define SHM\_KEY 0x12345

struct shmseg {

int cntr;

int write\_complete;

int read\_complete;

};

void shared\_memory\_cntr\_increment(int pid, struct shmseg \*shmp, int total\_count);

int main(int argc, char \*argv[]) {

int shmid;

struct shmseg \*shmp;

char \*bufptr;

int total\_count;

int sleep\_time;

pid\_t pid;

if (argc != 2)

total\_count = 10000;

else {

total\_count = atoi(argv[1]);

if (total\_count < 10000)

total\_count = 10000;

}

printf("Total Count is %d\n", total\_count);

shmid = shmget(SHM\_KEY, sizeof(struct shmseg), 0644|IPC\_CREAT);

if (shmid == -1) {

perror("Shared memory");

return 1;

}

shmp = shmat(shmid, NULL, 0);

if (shmp == (void \*) -1) {

perror("Shared memory attach");

return 1;

}

shmp->cntr = 0;

pid = fork();

if (pid > 0) {

shared\_memory\_cntr\_increment(pid, shmp, total\_count);

} else if (pid == 0) {

shared\_memory\_cntr\_increment(pid, shmp, total\_count);

return 0;

} else {

perror("Fork Failure\n");

return 1;

}

while (shmp->read\_complete != 1)

sleep(1);

if (shmdt(shmp) == -1) {

perror("shmdt");

return 1;

}

if (shmctl(shmid, IPC\_RMID, 0) == -1) {

perror("shmctl");

return 1;

}

printf("Writing Process: Complete\n");

return 0;

}

void shared\_memory\_cntr\_increment(int pid, struct shmseg \*shmp, int total\_count) {

int cntr;

int numtimes;

int sleep\_time;

cntr = shmp->cntr;

shmp->write\_complete = 0;

if (pid == 0)

printf("SHM\_WRITE: CHILD: Now writing\n");

else if (pid > 0)

printf("SHM\_WRITE: PARENT: Now writing\n");

//printf("SHM\_CNTR is %d\n", shmp->cntr);

for (numtimes = 0; numtimes < total\_count; numtimes++) {

cntr += 1;

shmp->cntr = cntr;

sleep\_time = cntr % 1000;

if (sleep\_time == 0)

sleep(1);

}

shmp->write\_complete = 1;

if (pid == 0)

printf("SHM\_WRITE: CHILD: Writing Done\n");

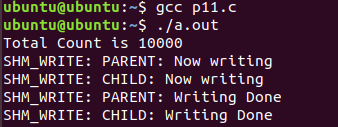
else if (pid > 0)

printf("SHM\_WRITE: PARENT: Writing Done\n");

return;

}

**OUTPUT**



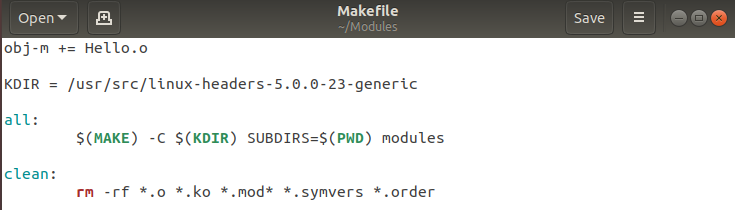
**PRACTICAL-12**

**AIM**

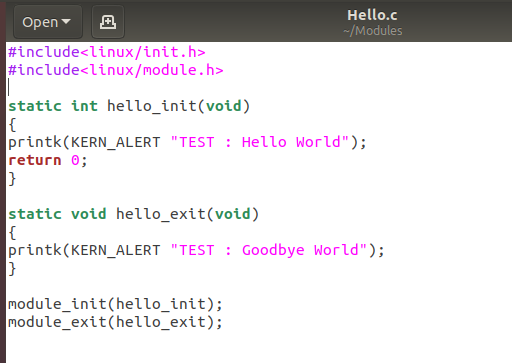
**Kernel space programming: Implement and add a loadable kernel module to Linux kernel, demonstrate using insmod, lsmod and rmmod commands. A sample kernel space program should print the "Hello World" while loading the kernel module and "Goodbye World" while unloading the kernel module.**

**PROGRAM CODE**

Makefile:



Hello.c:



**OUTPUT**

