ATME COLLEGE OF ENGINEERING

13th KM Stone, Bannur Road, Mysore - 560 028

DEPARTMENT OF COMPUTER SCIENCE & CYBER SECURITY

(ACADEMIC YEAR 2024)

LABORATORY MANUAL

SUBJECT: OPERATING SYSTEMS

SUB CODE: BCS303

SEMESTER: III-2024 CBCS Scheme

Composed by	Verified by	Approved by
Mr. Mahadevaswamy	Dr PAVITHRA A C	Dr. Nasreen Fathima
PROGRAMMER	FACULTY CO-ORDINATOR	HOD, CY

INSTITUTIONAL MISSION AND VISION

Objectives

- To provide quality education and groom top-notch professionals, entrepreneurs andleaders for different fields of engineering, technology and management.
- To open a Training-R & D-Design-Consultancy cell in each department, gradually introduce doctoral and postdoctoral programs, encourage basic & applied research in areas of social relevance, and develop the institute as a center of excellence.
- To develop academic, professional and financial alliances with the industry as well asthe academia at national and transnational levels.
- To cultivate strong community relationships and involve the students and the staff inlocal community service.
- To constantly enhance the value of the educational inputs with the participation of students, faculty, parents and industry.

Vision

- Development of academically excellent, culturally vibrant, socially responsible andglobally competent human resources.

Mission

- To keep pace with advancements in knowledge and make the students competitive and capable at the global level.
- To create an environment for the students to acquire the right physical, intellectual, emotional and moral foundations and shine as torch bearers of tomorrow's society.
- To strive to attain ever-higher benchmarks of educational excellence

OPERATING SYSTEMS		Semester	3
Course Code	BCS303	CIE Marks	50
Teaching Hours/Week (L:T:P: S)	3:0:2:0	SEE Marks	50
Total Hours of Pedagogy	40 hours Theory + 20 hours practicals	Total Marks	100
Credits	04	Exam Hours	3
Examination nature (SEE)	**Theory**		

Sl.NO	**Experiments**
1	Develop a c program to implement the Process system calls (fork (), exec(), wait(), create process,terminate process)
2	Simulate the following CPU scheduling algorithms to find turnaround time and waiting time a) FCFS b) SJF c) Round Robin d) Priority.
3	Develop a C program to simulate producer-consumer problem using semaphores.
4	Develop a C program which demonstrates interprocess communication between a reader process and a writer process. Use mkfifo, open, read, write and close APIs in your program.
5	Develop a C program to simulate Bankers Algorithm for DeadLock Avoidance.
6	Develop a C program to simulate the following contiguous memory allocation Techniques: a) Worst fit b) Best fit c) First fit.
7	Develop a C program to simulate page replacement algorithms: a) FIFO b) LRU
8	Simulate following File Organization Techniques a) Single level directory b) Two level directory
9	Develop a C program to simulate the Linked file allocation strategies.
10	Develop a C program to simulate SCAN disk scheduling algorithm.

Course outcomes (Course Skill Set):

At the end of the course, the student will be able to:

CO 1. Explain the structure and functionality of operating system

CO 2. Apply appropriate CPU scheduling algorithms for the given problem.

CO 3. Analyse the various techniques for process synchronization and deadlock handling.

CO 4. Apply the various techniques for memory management

CO 5. Explain file and secondary storage management strategies.

CO 6. Describe the need for information protection mechanisms

Assessment Details (both CIE and SEE)

The weightage of Continuous Internal Evaluation (CIE) is 50% and for Semester End Exam (SEE) is 50%. The minimum passing mark for the CIE is 40% of the maximum marks (20 marks out of 50) and for the SEE minimum passing mark is 35% of the maximum marks (18 out of 50 marks). A student shall be deemed tohave satisfied the academic requirements and earned the credits allotted to each subject/ course if the student secures a minimum of 40% (40 marks out of 100) in the sum total of the CIE (Continuous Internal Evaluation) and SEE (Semester End Examination) taken together.

CIE for the theory component of the IPCC (maximum marks 50)

- IPCC means practical portion integrated with the theory of the course.
- CIE marks for the theory component are **25 marks** and that for the practical component is **25 marks**.
- 25 marks for the theory component are split into **15 marks** for two Internal Assessment Tests (Two Tests, each of 15 Marks with 01-hour duration, are to be conducted) and **10 marks** for other assessment methods

mentioned in 22OB4.2. The first test at the end of 40-50% coverage of the syllabus and the second testafter covering 85-90% of the syllabus.

- Scaled-down marks of the sum of two tests and other assessment methods will be CIE marks for the theory component of IPCC (that is for **25 marks)**.
- The student has to secure 40% of 25 marks to qualify in the CIE of the theory component of IPCC.

CIE for the practical component of the IPCC

- **15 marks** for the conduction of the experiment and preparation of laboratory record, and **10 marks** for the test to be conducted after the completion of all the laboratory sessions.
- On completion of every experiment/program in the laboratory, the students shall be evaluated including viva-voce and marks shall be awarded on the same day.
- The CIE marks awarded in the case of the Practical component shall be based on the continuous evaluationof the laboratory report. Each experiment report can be evaluated for 10 marks. Marks of all experiments' write-ups are added and scaled down to **15 marks**.
- The laboratory test **(duration 02/03 hours)** after completion of all the experiments shall be conducted for 50 marks and scaled down to **10 marks.**
- Scaled-down marks of write-up evaluations and tests added will be CIE marks for the laboratory component of IPCC for **25 marks**.
- The student has to secure 40% of 25 marks to qualify in the CIE of the practical component of the IPCC.

SEE for IPCC

Theory SEE will be conducted by University as per the scheduled timetable, with common question papers forthe course (**duration 03 hours**)

1. The question paper will have ten questions. Each question is set for 20 marks.
2. There will be 2 questions from each module. Each of the two questions under a module (with a maximumof 3 sub-questions), **should have a mix of topics** under that module.
3. The students have to answer 5 full questions, selecting one full question from each module.
4. Marks scoredby the student shall be proportionally scaled down to 50 Marks

The theory portion of the IPCC shall be for both CIE and SEE, whereas the practical portion will have a CIE component only. Questions mentioned in the SEE paper may include questions from the practical component.

Suggested Learning Resources:

Textbooks

1. Abraham Silberschatz, Peter Baer Galvin, Greg Gagne, Operating System Principles 8th edition, Wiley-India, 2015

Reference Books

1. Ann McHoes Ida M Fylnn, Understanding Operating System, Cengage Learning, 6th Edition
2. D.M Dhamdhere, Operating Systems: A Concept Based Approach 3rd Ed, McGraw- Hill, 2013.
3. P.C.P. Bhatt, An Introduction to Operating Systems: Concepts and Practice 4th Edition, PHI(EEE), 2014.
4. William Stallings Operating Systems: Internals and Design Principles, 6th Edition, Pearson.

Web links and Video Lectures (e-Resources):

1. https://youtu.be/mXw9ruZaxzQ
2. https://youtu.be/vBURTt97EkA
3. https://www.youtube.com/watch?v=783KAB-tuE4&list=PLIemF3uozcAKTgsCIj82voMK3TMR0YE_f
4. https://www.youtube.com/watch?v=3-ITLMMeeXY&list=PL3pGy4HtqwD0n7bQfHjPnsWzkeRn6mkO

Activity Based Learning (Suggested Activities in Class)/ Practical Based learning

- Assessment Methods
  - Case Study on Unix Based Systems (10 Marks)
  - Lab Assessment (25 Marks)

CONTENTS

Introduction to Operating System

Installation of Operating system (Linux/Ubuntu)

Perform a case study by installing and exploring various types of operating systems on a physical or logical (virtual) machine. (Linux Installation). Instructions to Install Ubuntu Linux 18.04 (LTS) along with Windows Back Up Your Existing Data!

This is highly recommended that you should take backup of your entire data before start with the installation process.

Obtaining System Installation Media

Download latest Desktop version of Ubuntu from this link:

https://ubuntu.com/tutorials/install-ubuntu-desktop-1804#download

Booting the Installation System

There are several ways to boot the installation system. Some of the very popular ways are, booting from a CD ROM, booting from a USB memory stick, and booting from TFTP.

Here we will learn how to boot installation system using a CD ROM.

Before booting the installation system, one need to change the boot order and set CD-ROM as first boot device.

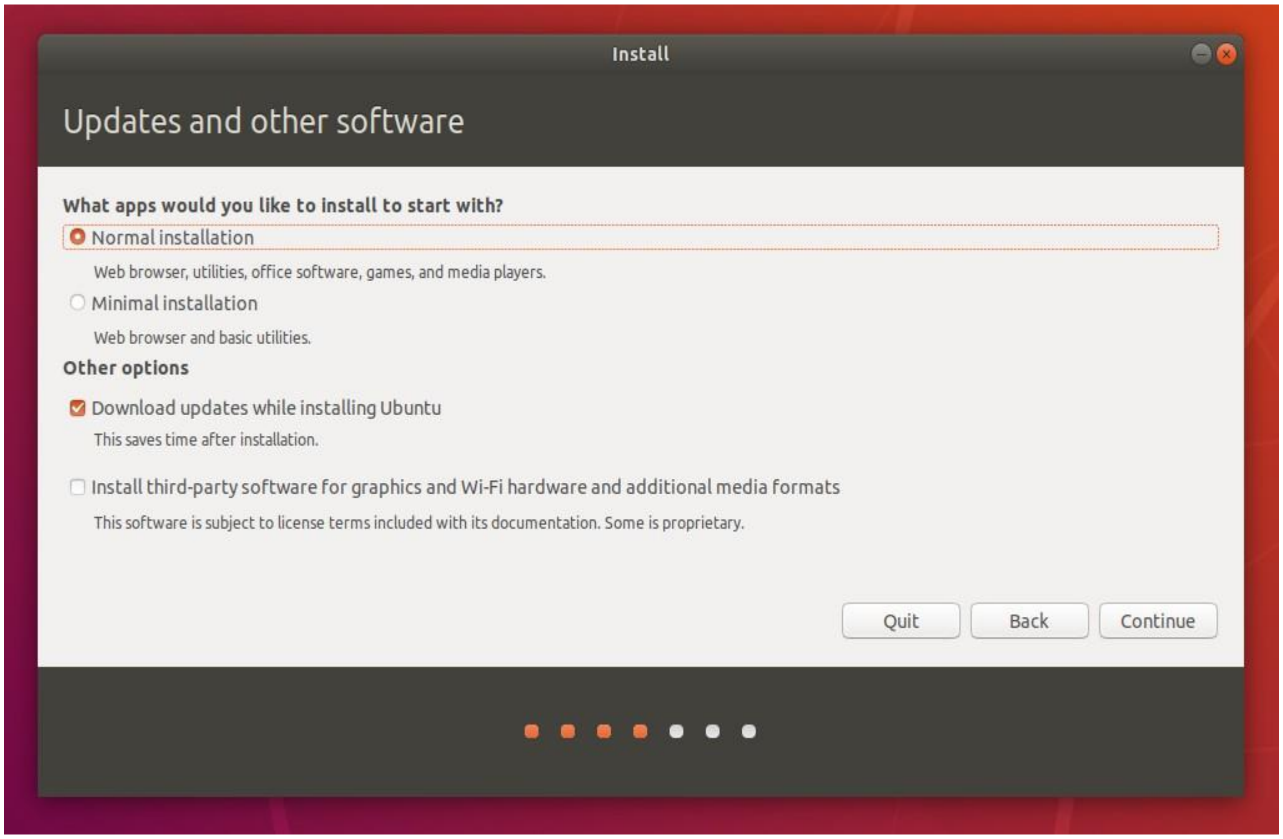
Changing the Boot Order of a Computers

As your computer starts, press the DEL-F10, HP-F9 during the initial startup screen. Depending on the BIOS manufacturer, a menu may appear. However, consult the hardware documentation for the exact key strokes.

Beneath the installation-type question are two checkboxes; one to enable updates while installing and another to enable third-party software.

- We advise enabling both Download updates and Install third-party software.
- Stay connected to the internet so you can get the latest updates while you install Ubuntu.

- If you are not connected to the internet, you will be asked to select a wireless network, if available. We advise you to connect during the installation so we can ensure your machine is up to date.



Updates and other software

What apps would you like to install to start with?

[✓ Normal installation]

Web browser, utilities, office software, games, and media players.

Minimal installation

Web browser and basic utilities.

Other options

[✓ Download updates while installing Ubuntu]

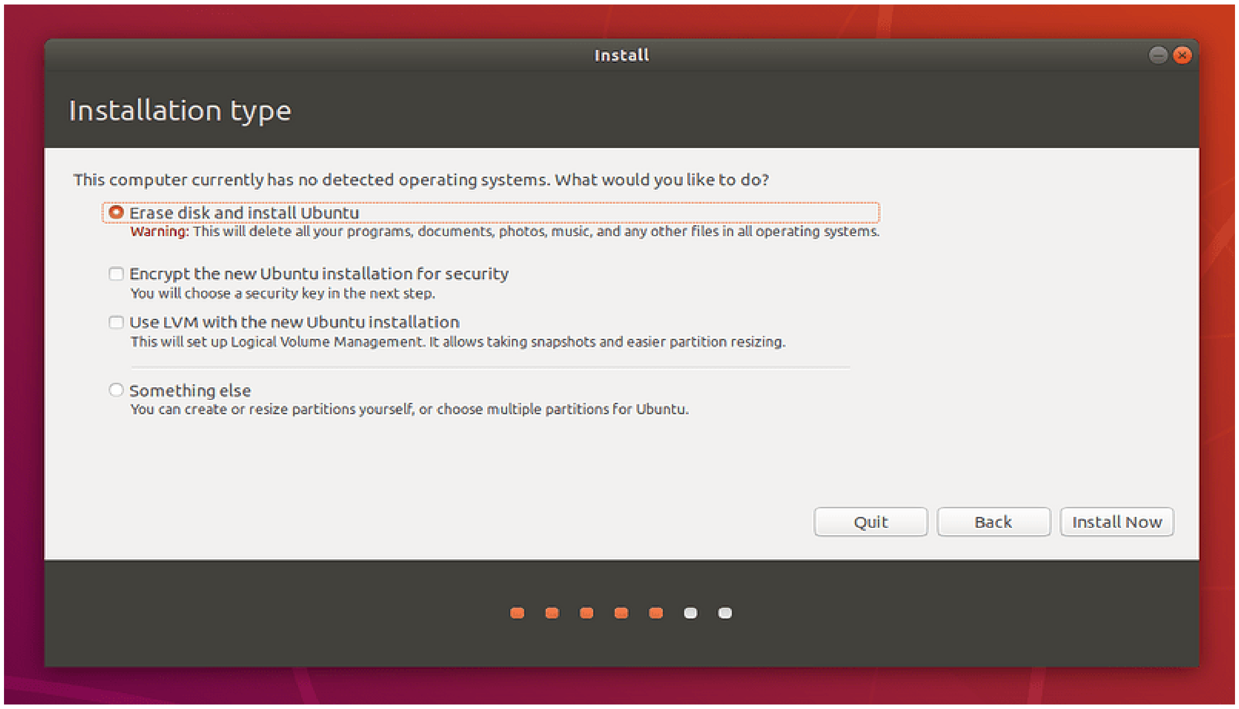
This saves time after installation.

Install third-party software for graphics and Wi-Fi hardware and additional media formats

This software is subject to license terms included with its documentation. Some is proprietary.

Allocate drive space

Use the checkboxes to choose whether you'd like to install Ubuntu alongside another operating system, delete your existing operating system and replace it with Ubuntu, or — if you're an advanced user — choose the '**Something else**' option.



Installation type

This computer currently has no detected operating systems. What would you like to do?

[✓ Erase disk and install Ubuntu]
Warning: This will delete all your programs, documents, photos, music, and any other files in all operating systems.

Encrypt the new Ubuntu installation for security
You will choose a security key in the next step.

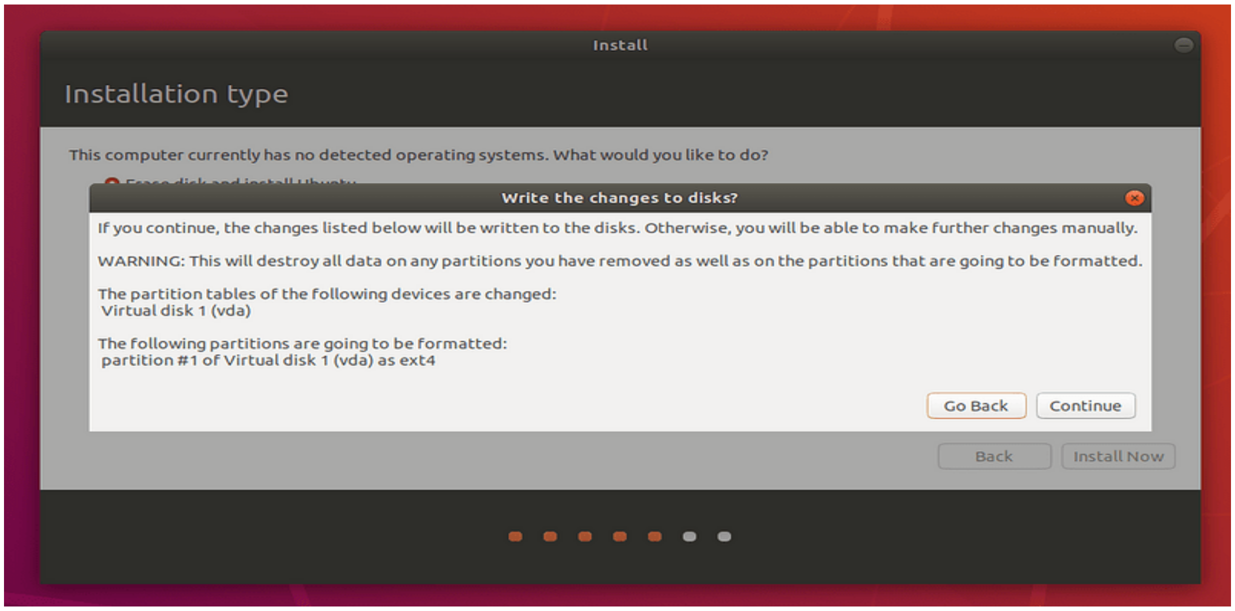
Use LVM with the new Ubuntu installation
This will set up Logical Volume Management. It allows taking snapshots and easier partition resizing.

Something else
You can create or resize partitions yourself, or choose multiple partitions for Ubuntu.

Begin installation

After configuring storage, click on the 'Install Now' button. A small pane will appear with an overview of the storage options you've chosen, with the chance to go back if the details are incorrect.

Click Continue to fix those changes in place and start the installation process.



Installation type

This computer currently has no detected operating systems. What would you like to do?

Write the changes to disks?

If you continue, the changes listed below will be written to the disks. Otherwise, you will be able to make further changes manually.

WARNING: This will destroy all data on any partitions you have removed as well as on the partitions that are going to be formatted.

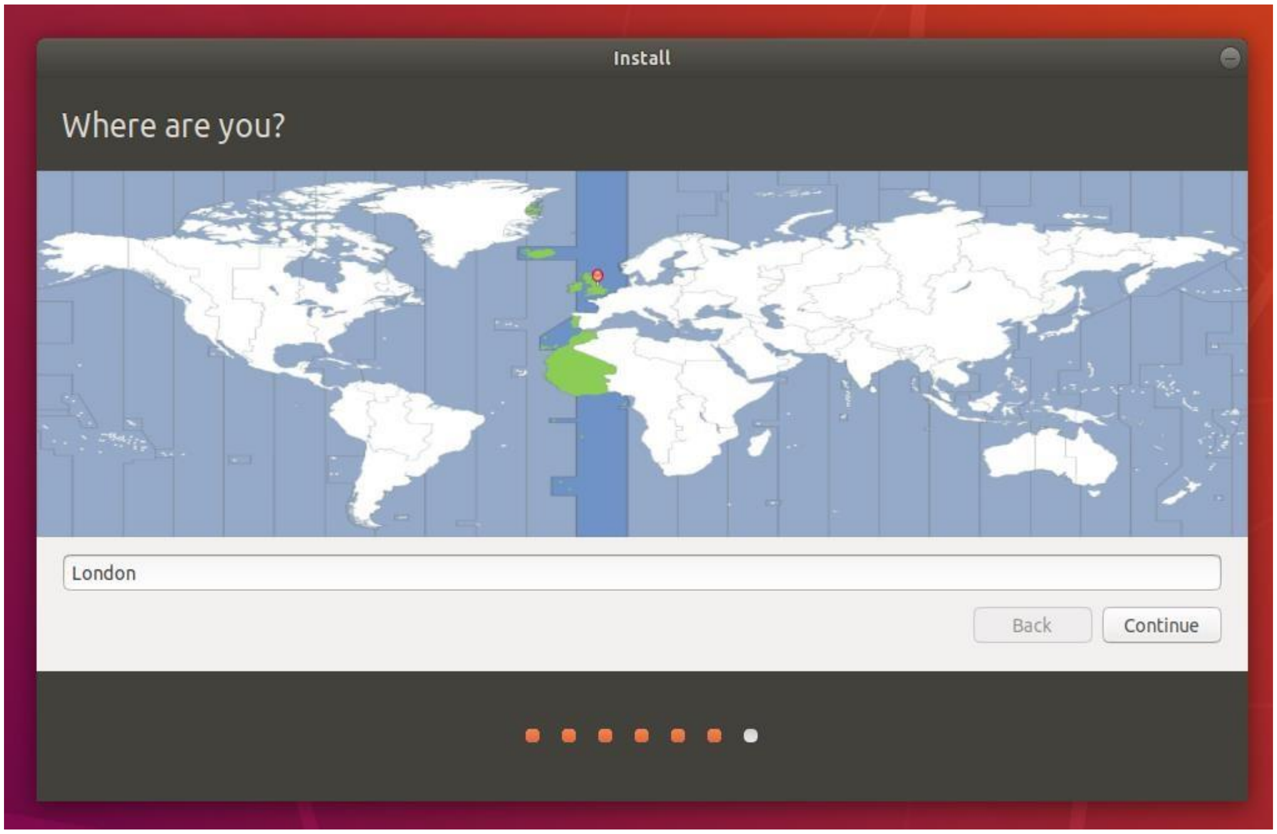
The partition tables of the following devices are changed:
Virtual disk 1 (vda)

The following partitions are going to be formatted:
partition #1 of Virtual disk 1 (vda) as ext4

Select your location

If you are connected to the internet, your location will be detected automatically. Check your location is correct and click 'Forward' to proceed.

If you're unsure of your time zone, type the name of a local town or city or use the map to select your location.



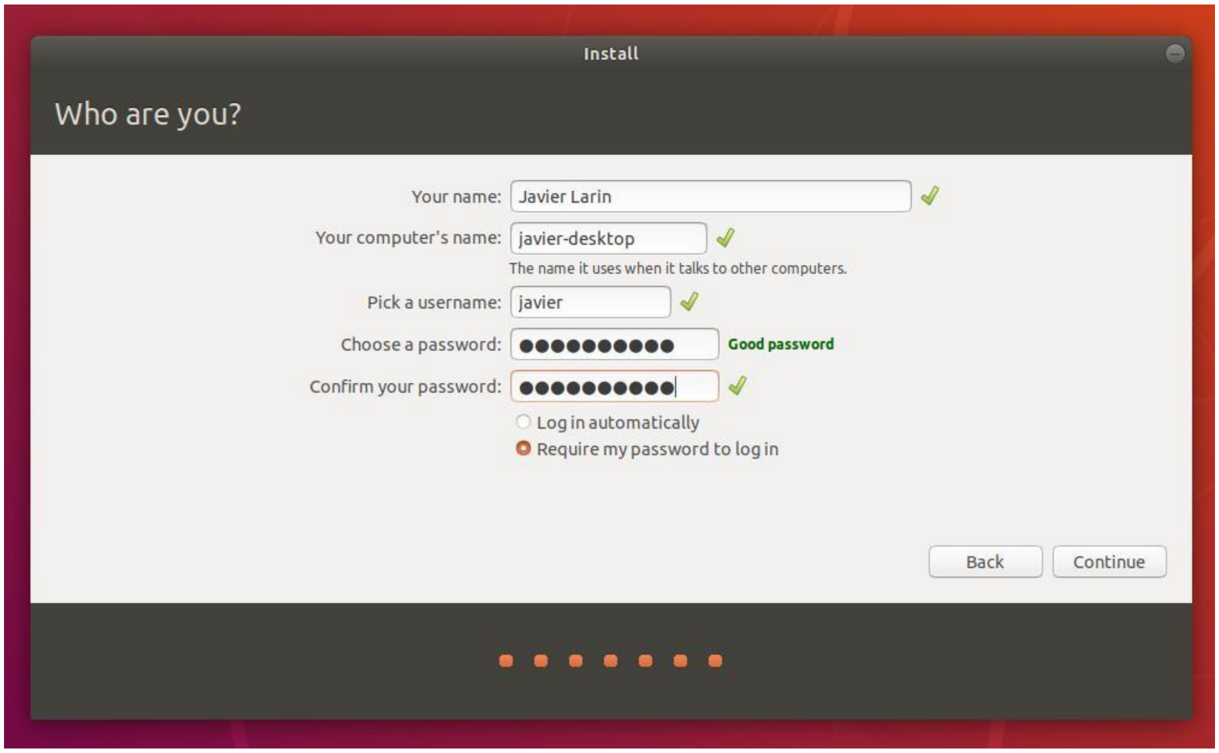
Positive: If you're having problems connecting to the Internet, use the menu in the top-right-hand corner to select a network.

Login details

Enter your name and the installer will automatically suggest a computer name and username. These can easily be changed if you prefer. The computer name is how your computer will appear on the network, while your username will be your login and account name.

Next, enter a strong password. The installer will let you know if it's too weak.

You can also choose to enable automatic login and home folder encryption. If your machine is portable, we recommend keeping automatic login disabled and enabling encryption. This should stop people accessing your personal files if the machine is lost or stolen.



Your name: Javier Larin ✓

Your computer's name: javier-desktop ✓

The name it uses when it talks to other computers.

Pick a username: javier ✓

Choose a password: ●●●●●●●●●● Good password

Confirm your password: ●●●●●●●●●● ✓

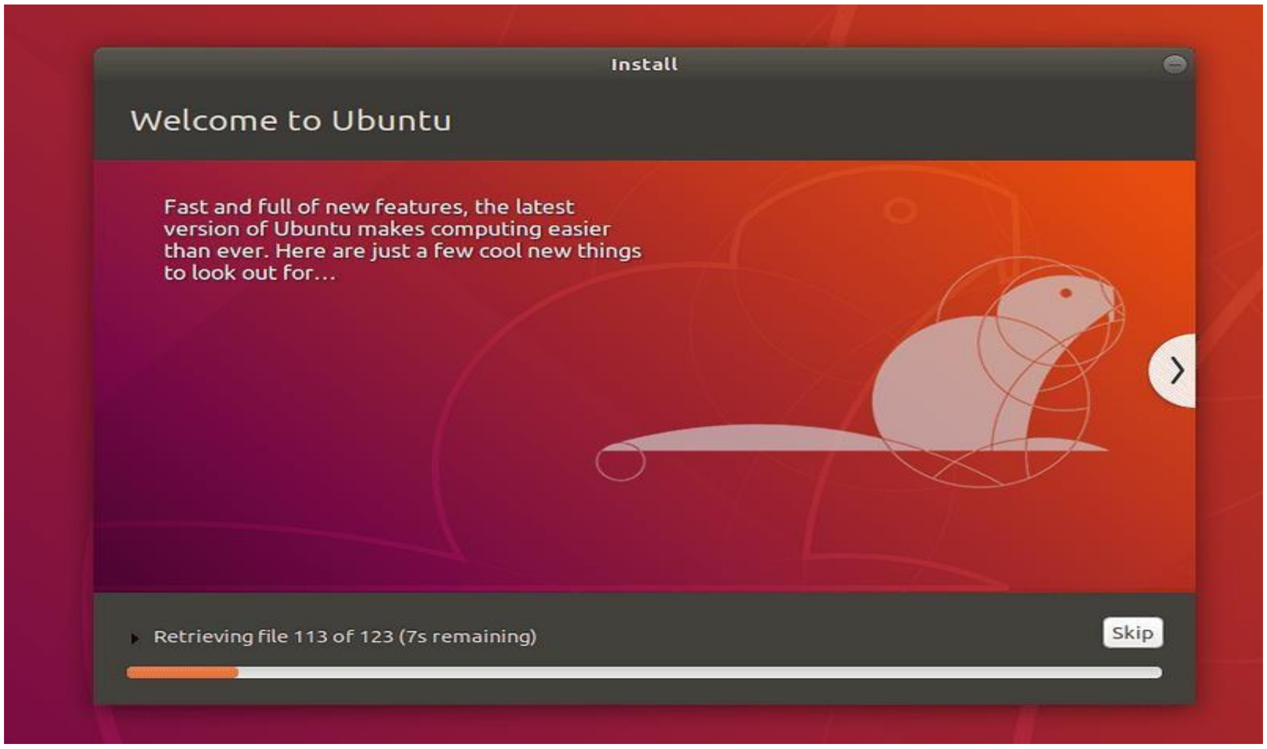
Log in automatically [✓ Require my password to log in]

:

If you enable home folder encryption and you forget your password, you won't be able to retrieve any personal data stored in your home folder.

Background installation

The installer will now complete in the background while the installation window teaches you a little about how awesome Ubuntu is. Depending on the speed of your machine and network connection, installation should only take a few minutes.


Install

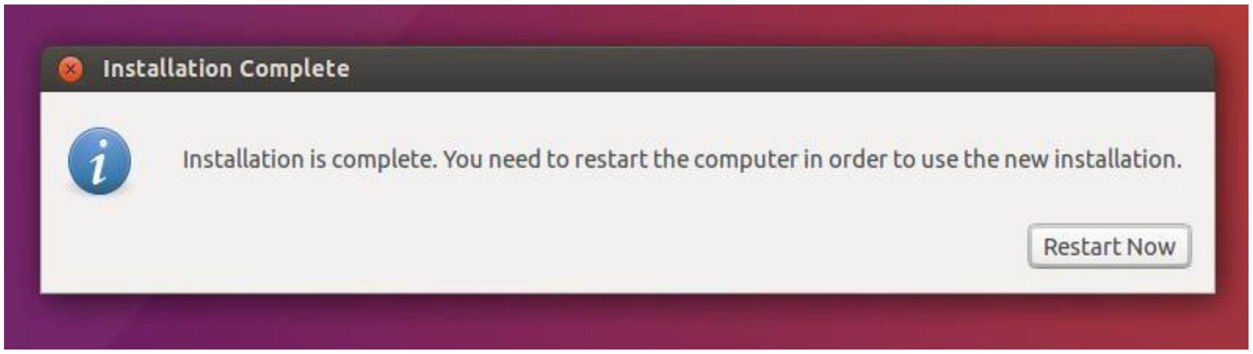
Welcome to Ubuntu

Fast and full of new features, the latest version of Ubuntu makes computing easier than ever. Here are just a few cool new things to look out for…

Retrieving file 113 of 123 (7s remaining)

Installation complete

After everything has been installed and configured, a small window will appear asking you to restart your machine. Click on Restart Now and remove either the DVD or USB flash drive when prompted. If you initiated the installation while testing the desktop, you also get the option to continue testing.


Installation Complete

Installation is complete. You need to restart the computer in order to use the new installation.

Congratulations! You have successfully installed the world's most popular Linux operating system!

It's now time to start enjoying Ubuntu!

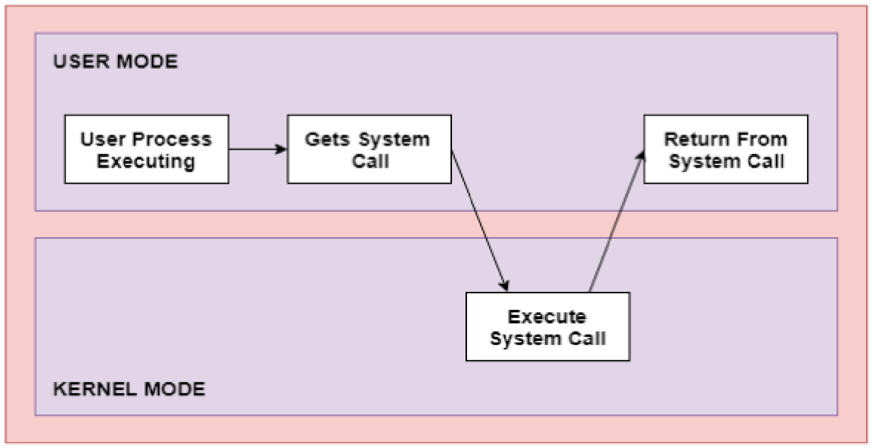
PROGRAM 1:

Develop a c program to implement the Process system calls (fork (), exec(), wait(), create process, terminate process)

System Calls:

The interface between a process and an operating system is provided by system calls. In general, system calls are available as assembly language instructions. They are also included in the manuals used by the assembly level programmers. System calls are usually made when a process in user mode requires access to a resource. Then it requests the kernel to provide the resource via a system call.

A figure representing the execution of the system call is given as follows –



In general, system calls are required in the following situations –

• If a file system requires the creation or deletion of files. Reading and writing from files also require a system call.

• Creation and management of new processes.

• Network connections also require system calls. This includes sending and receiving packets.

• Access to a hardware devices such as a printer, scanner etc. requires a system call

Types of System Calls

There are mainly five types of system calls. These are explained in detail as follows −

Process Control

These system calls deal with processes such as process creation, process termination etc.

File Management

These system calls are responsible for file manipulation such as creating a file, reading a file, writing into a file etc.

Device Management

These system calls are responsible for device manipulation such as reading from device buffers, writing into device buffers etc.

Information Maintenance

These system calls handle information and its transfer between the operating system and the user program.

Communication

These system calls are useful for interprocess communication. They also deal with creating and deleting a communication connection.

fork ( )

Used to create new process. The new process consists of a copy of the address space of the

original process. The value of process id for the child process is zero, whereas the value of process id for the parent is an integer value greater than zero.

Syntax: fork ( );

wait ( )

The parent waits for the child process to complete using the wait system call. The wait system call returns the process identifier of a terminated child, so that the parent can tell which of its possibly many children has terminated.

Syntax: wait (NULL);

exit ( )

A process terminates when it finishes executing its final statement and asks the operating

system to delete it by using the exit system call. At that point, the process may return data (output) to its parent process (via the wait system call).

Syntax: exit (0)

Fork () System Call:

AIM: To write the program to implement fork () system call.

DESCRIPTION:

Used to create new processes. The new process consists of a copy of the address space of the original process. The value of process id for the child process is zero, whereas the value of process id for the parent is an integer value greater than zero.

Syntax: Fork ( );

ALGORITHM:

Step 1: Start the program.

Step 2: Declare the variables pid and child id.

Step 3: Get the child id value using system call fork().

Step 4: If child id value is greater than zero then print as “i am in the parent process”.

Step 5: If child id! = 0 then using getpid() system call get the process id.

Step 6: Print “i am in the parent process” and print the process id.

Step 7: If child id! = 0 then using getppid() system call get the parent process id.

Step 8: Print “i am in the parent process” and print the parent process id.

Step 9: Else If child id value is less than zero then print as “i am in the child process”.

Step 10: If child id! = 0 then using getpid() system call get the process id.

Step 11: Print “i am in the child process” and print the process id.

Step 12: If child id! = 0 then using getppid() system call get the parent process id.

Step 13: Print “i am in the child process” and print the parent process id.

Step 14: Stop the program

PROGRAM :

SOURCE CODE:

```
/* fork system call */

#include<stdio.h>

#include <unistd.h>

#include<sys/types.h>

int main()

{

int id,childid;

id=getpid();

if((childid=fork())>0)

{

printf("\n i am in the parent process %d",id);

printf("\n i am in the parent process %d",getpid());

printf("\n i am in the parent process %d\n",getppid());

}
else

{

printf("\n i am in child process %d",id);

printf("\n i am in the child process %d",getpid());

printf("\n i am in the child process %d",getppid());

}

}
```

OUTPUT:

```
$ gedit fork.c

$ cc fork.c

$ ./a.out

i am in child process 3765

i am in the child process 3766

i am in the child process 3765

i am in the parent process 3765

i am in the parent process 3765

i am in the parent process 3680
```

RESULT:

Thus the program was executed and verified successfully.

ALGORITHM:

Step 1: Start the program.

Step 2: Declare the variables pid and i as integers.

Step 3: Get the child id value using the system call fork ().

Step 4: If child id value is less than zero then print “fork failed”.

Step 5: Else if child id value is equal to zero, it is the id value of the child and then start the child process to execute and perform Steps 7 & 8.

Step 6: Else perform Step 9.

Step 7: Use a for loop for almost five child processes to be called.

Step 8: After execution of the for loop then print “child process ends”.

Step 9: Execute the system call wait ( ) to make the parent to wait for the child process to get over.

Step 10: Once the child processes are terminated, the parent terminates and hence prints “Parent process ends”.

Step 11: After both the parent and the child processes get terminated it execute the wait ( ) system call to permanently get deleted from the OS.

Step 12: Stop the program.

SOURCE CODE:

```
/* wait system call */

#include <stdlib.h>

#include <errno.h>

#include<stdio.h>

#include <unistd.h>

#include <sys/types.h>
```

```
#include <sys/wait.h>

main()

{

      pid_t pid;

      int rv;

      switch(pid=fork())

{

case -1:

      perror("fork");

      exit(1);

case 0:

      printf("\n CHILD: This is the child process!\n");

      fflush(stdout);

      printf("\n CHILD: My PID is %d\n", getpid());

      printf("\n CHILD: My parent's PID is %d\n",getppid());

      printf("\n CHILD: Enter my exit status (make it small):\n ");

      printf("\n CHILD: I'm outta here!\n");

      scanf(" %d", &rv);

      exit(rv);

default:
      printf("\nPARENT: This is the parent process!\n");

      printf("\nPARENT: My PID is %d\n", getpid());

      fflush(stdout);

      wait(&rv);

      fflush(stdout);

      printf("\nPARENT: My child's PID is %d\n", pid);

      printf("\nPARENT: I'm now waiting for my child to exit()...\n");

      fflush(stdout);

      printf("\nPARENT: My child's exit status is:

      %d\n",WEXITSTATUS(rv));

      printf("\nPARENT: I'm outta here!\n");

}

}
```

OUTPUT:

```
$ gedit wait.c

$ cc wait.c

$ ./a.out

CHILD: This is the child process!

CHILD: My PID is 3821

CHILD: My parent's PID is 3820
```

```
CHILD: Enter my exit status (make it small):

CHILD: I'm outta here!

PARENT: This is the parent process!

PARENT: My PID is 3820

10

PARENT: My child's PID is 3821

PARENT: I'm now waiting for my child to exit()...

PARENT: My child's exit status is: 10

PARENT: I'm outta here!
```

Exec()

In execl() function, the parameters of the executable file is passed to the function as different arguments. With execv(), you can pass all the parameters in a NULL terminated array **argv**. The first element of the array should be the path of the executable file. Otherwise, execv() function works just as execl() function.

Syntax:int execv (const char *path, char *const argv[]);

ALGORITHM:

Step 1: Start the program.

Step 2: Include the necessary header files.

Step 3: Print execution of exec system call for the ls Unix command.

Step 4: Execute the execv function using the appropriate syntax for the Unix command ls.

Step 5: The list of all files and directories of the system is displayed.

Step 6: Stop the program.

PROGRAM :

SOURCE CODE:

```
/* execv system call */

#include<stdio.h>

#include<sys/types.h>

main(int argc,char *argv[])

{

printf("before execv\n");

execv("/bin/ls",argv);

printf("after execv\n");

}
```

OUTPUT:

```
$ gedit execv.c

$ cc execv.c

$ ./a.out

before execv

a1 aaa aaa.txt abc a.out b1 b2 comm.c db db1 demo2 dir1 direc.c execl.c execv.c f1.txt

fflag.c file1 file2 fork.c m1 m2 wait.c xyz
```

RESULT:

Thus the program was executed and verified successfully

PROGRAM 2:

Simulate the following CPU scheduling algorithms to find turnaround time and waiting time a) FCFS b) SJF c) Round Robin d) Priority.

CPU Scheduling Algorithms:

Scheduling of processes/work is done to finish the work on time.

Below are different times with respect to a process.

Arrival Time : Time at which the process arrives in the ready queue.

Completion Time : Time at which process completes its execution.

Burst Time : Time required by a process for CPU execution.

Turn Around Time : Time Difference between completion time and arrival time.

Turn Around Time = Completion Time - Arrival Time

Waiting Time(W.T) : Time Difference between turn around time and burst time.

Waiting Time = Turn Around Time - Burst Time

Why do we need scheduling?

A typical process involves both I/O time and CPU time. In a uniprogramming system like MS DOS, time spent waiting for I/O is wasted and CPU is free during this time. In multiprogramming systems, one process can use CPU while another is waiting for I/O. This is possible only with process scheduling.

Objectives of Process Scheduling Algorithm

- Max CPU utilization [Keep CPU as busy as possible]
- Fair allocation of CPU.
- Max throughput [Number of processes that complete their execution per time unit]
- Min turnaround time [Time taken by a process to finish execution]

• Min waiting time [Time a process waits in ready queue]

• Min response time [Time when a process produces first response]

Different Scheduling Algorithms:

First Come First Serve (FCFS): Simplest scheduling algorithm that schedules according to arrival times of processes. First come first serve scheduling algorithm process that requests the CPU first is allocated the CPU first. It is implemented by using the FIFO queue. When a process enters the ready queue, its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue. FCFS is a non – preemptive scheduling algorithm.

Note: First come first serve suffers from convoy effect.

Shortest Job First (SJF): Process which has the shortest burst time is scheduled first. If two processes have the same bust time, then FCFS is used to break the tie. It is a non-preemptive scheduling algorithm

AIM: To write a C program to implement FCFS CPU scheduling algorithm.

DESCRIPTION:

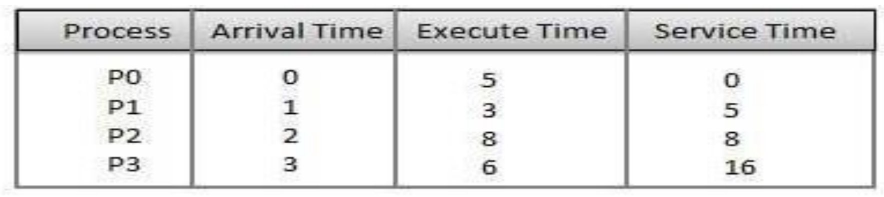
• Jobs are executed on FCFS (First Come, First Serve) basis.

• It is a non-preemptive, preemptive scheduling algorithm.

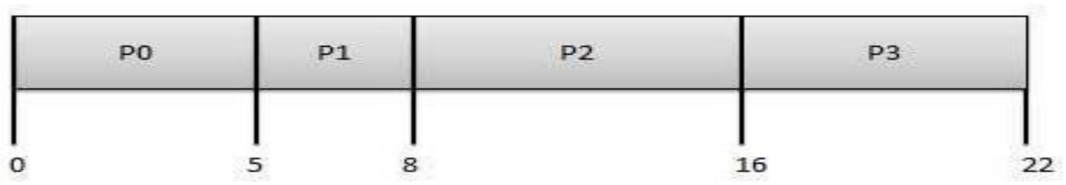
• Easy to understand and implement.

• Its implementation is based on FIFO (First In First Out) queue.

• Poor in performance as average wait time is high.



Process	Arrival Time	Execute Time	Service Time
P0	0	5	0
P1	1	3	5
P2	2	8	8
P3	3	6	16



Wait time of each process is as follows –

Process	Wait Time : Service Time - Arrival Time
P0	0 - 0 = 0
P1	5 - 1 = 4
P2	8 - 2 = 6
P3	16 -3=16

Average Wait Time: (0+4+6+13) / 4 = 5.75

ALGORITHM:

Step 1: Start the program.

Step 2: Create the number of process.

Step 3: Get the ID and Service time for each process.

Step 4: Initially, Waiting time of first process is zero and Total time for the first process is the starting time of that process.

Step 5: Calculate the Total time and Processing time for the remaining processes.

Step 6: Waiting time of one process is the Total time of the previous process.

Step 7: Total time of process is calculated by adding Waiting time and Service time.

Step 8: Total waiting time is calculated by adding the waiting time for lack process.

Step 9: Total turn around time is calculated by adding all total time of each process.

Step 10: Calculate Average waiting time by dividing the total waiting time by total number of process.

Step 11: Calculate Average turn around time by dividing the total time by the number of process.

Step 12: Display the result.

Step 13: Stop the program.

PROGRAM:

SOURCE CODE:

```
/* A program to simulate the FCFS CPU scheduling algorithm */
#include<stdio.h>
int main()
{
char pn[10][10];
int arr[10],bur[10],star[10],finish[10],tat[10],wt[10],i,n;
int totwt=0,tottat=0;
printf("Enter the number of processes:");
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("Enter the Process Name, Arrival Time & Burst Time:");
scanf("%s%d%d",&pn[i],&arr[i],&bur[i]);

}

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

{

if(i==0)

{

star[i]=arr[i];

wt[i]=star[i]-arr[i];

finish[i]=star[i]+bur[i];

tat[i]=finish[i]-arr[i];

}

else

{

star[i]=finish[i-1];

wt[i]=star[i]-arr[i];

finish[i]=star[i]+bur[i];

tat[i]=finish[i]-arr[i];

}

}

printf("\nPName Arrtime Burtime Start TAT Finish");
```

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

{

printf("\n%s\t%6d\t\t%6d\t%6d\t%6d\t%6d",pn[i],arr[i],bur[i],star[i],tat[i],finish[i]);

totwt+=wt[i];

tottat+=tat[i];

}

printf("\nAverage Waiting time:%f", (float)totwt);

printf("\nAverage Turn Around Time:%f", (float)tottat);

}
```

OUTPUT:

```
$ cc fcfs.c
$ ./a.out
Enter the number of processes: 3
Enter the Process Name, Arrival Time & Burst Time: 1    2   3
Enter the Process Name, Arrival Time & Burst Time: 2    5   6
Enter the Process Name, Arrival Time & Burst Time: 3    6   7

PName       Arrtime           Burtime      Start         TAT      Finish

   1            2                 3           2            3          5

   2            5                 6          5             6          11

   3            6                 7          11            12         18

Average Waiting time: 1.666667
```

Average Turn Around Time: 7.000000

/* A program to simulate the SJF CPU scheduling algorithm */

AIM:To write a C program to implement SJF CPU scheduling algorithm.

DESCRIPTION:

- This is also known as shortest job first, or SJF
- This is a non-preemptive, pre-emptive scheduling algorithm.
- Best approach to minimize waiting time.
- Easy to implement in Batch systems where required CPU time is known in advance.
- Impossible to implement in interactive systems where required CPU time is not known.
- The processer should know in advance how much time process will take. Given: Table of processes, and their Arrival time, Execution time

Process	Arrival Time	Execution Time	Service Time
P0	0	5	0
P1	1	3	5
P2	2	8	14
P3	3	6	8

Waiting time of each process is as follows –

Process	Waiting Time
P0	0-0=0
P1	5 - 1 = 4
P2	14 - 2 = 12
P3	8 - 3 = 5

Average Wait Time: (0 + 4 +12 + 5)/4 = 21 / 4=5.25

ALGORITHM:

Step 1: Start the program.

Step 2: Get the number of process.

Step 3: Get the id and service time for each process.

Step 4: Initially the waiting time of first short process as 0 and total time of first short is process the service time of that process.

Step 5: Calculate the total time and waiting time of remaining process.

Step 6: Waiting time of one process is the total time of the previous process.

Step 7: Total time of process is calculated by adding the waiting time and service time of each process.

Step 8: Total waiting time calculated by adding the waiting time of each process.

Step 9: Total turn around time calculated by adding all total time of each process.

Step 10: Calculate average waiting time by dividing the total waiting time by total number of process.

Step 11: Calculate average turn around time by dividing the total waiting time by total number of process.

Step 12: Display the result.

Step 13: Stop the program.

PROGRAM:

SOURCE CODE: /* A program to simulate the SJF CPU scheduling algorithm */

```
#include<stdio.h>

#include<string.h>
```

```
Void  main()

{

int i=0,pno[10],bt[10],n,wt[10],temp=0,j,tt[10];

float sum,at;

printf("\n Enter the no of process ");

scanf("\n %d",&n);

printf("\n Enter the burst time of each process");

for(i=0;i<n;i++) { printf("\n p%d",i);

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

}

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

{

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

 {

if(bt[i]>bt[j])

 {

 temp=bt[i];

 bt[i]=bt[j];

 bt[j]=temp;

 temp=pno[i];
pno[i]=pno[j];
 pno[j]=temp;
 } } }
 wt[0]=0;
 for(i=1;i<n;i++)
 {
wt[i]=bt[i-1]+wt[i-1];
sum=sum+wt[i];
 }
printf("\n process no \t burst time\t waiting time \t turn around time\n");
for(i=0;i<n;i++)
 {
 tt[i]=bt[i]+wt[i];
at+=tt[i];
printf("\n p%d\t\t%d\t\t%d\t\t%d",i,bt[i],wt[i],tt[i]);
}
printf("\n\n\t Average waiting time%f\n\t Average turn around time%f", sum, at);
}
```

OUTPUT:

```
$ cc sjf.c

$ ./a.out

Enter the no of process 5

Enter the burst time of each process

p0 1
p1  5
p2 2
p3 3
p4 4


process no     burst time     waiting time    turn around time

    p0              1             0              1

    p1              2             1              3

    p2              3             3              6

    p3              4             6              10

    p4             5             10              15


Average waiting time 4.000000
```

Average turn around time 7.000000

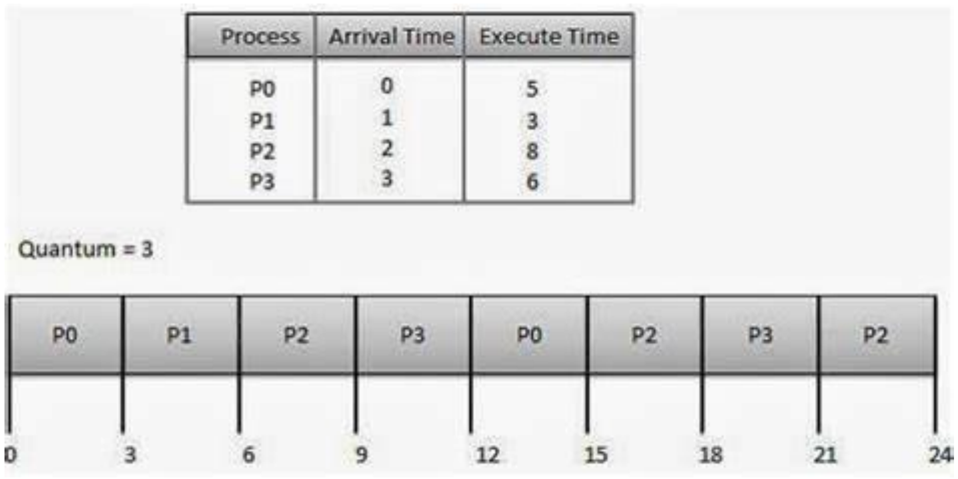
RESULT: Thus the program was executed and verified successfully.

Write C program to simulate Round Robin CPU scheduling algorithm.

AIM: To write a C program to implement Round Robin CPU scheduling algorithm.

DESCRIPTION

- Round Robin is the preemptive process scheduling algorithm.
- Each process is provided a fix time to execute, it is called a quantum.
- Once a process is executed for a given time period, it is preempted and other process executes for a given time period.
- Context switching is used to save states of preempted processes.



Process	Arrival Time	Execute Time
P0	0	5
P1	1	3
P2	2	8
P3	3	6

Quantum = 3

ALGORITHM:

Step 1: Start the program.
Step 2: Initialize all the structure elements.
Step 3: Receive inputs from the user to fill process id, burst time and arrival time.
Step 4: Calculate the waiting time for all the process id.

i. The waiting time for first instance of a process is calculated as: a[i].waittime=count + a[i].arrivt.

ii. The waiting time for the rest of the instances of the process is calculated as:

a) If the time quantum is greater than the remaining burst time then waiting time is calculated as: a[i].waittime=count + tq.

b) Else if the time quantum is greater than the remaining burst time then waiting time is calculated as: a[i].waittime=count - remaining burst time

Step 5: Calculate the average waiting time and average turnaround time

Step 6: Print the results of the step 4.

Step 7: Stop the program.

PROGRAM :

SOURCE CODE:

```
/* A program to simulate the Round Robin CPU scheduling algorithm */
#include<stdio.h>
struct process
{
int burst,wait,comp,f;
 }p[20]={0,0};
int main()
{
int n,i,j,totalwait=0,totalturn=0,quantum,flag=1,time=0;
printf("\nEnter The No Of Process :");
 scanf("%d",&n);
printf("\nEnter The Quantum time (in ms) :");
scanf("%d",&quantum);
for(i=0;i<n;i++)
{
printf("Enter The Burst Time (in ms) For Process #%2d :",i+1);
```

```
scanf("%d",&p[i].burst);
p[i].f=1;
}
printf("\nOrder Of Execution \n");
printf("\nProcess Starting Ending Remaining");
printf("\n\t\tTime \tTime \t Time");
while(flag==1)
{
flag=0; for(i=0;i<n;i++)
{
if(p[i].f==1)
{
flag=1; j=quantum;
if((p[i].burst-p[i].comp)>quantum)
 {

p[i].comp+=quantum;
}
else
{ p[i].wait=time-p[i].comp;
j=p[i].burst-p[i].comp;
p[i].comp=p[i].burst;
  p[i].f=0;
 }
printf("\nprocess  #  %-3d  %-10d  %-10d  %-10d",  i+1,  time,  time+j,  p[i].burst-
p[i].comp);
time+=j;
}
 }
}
printf("\n\n--------------------");
printf("\nProcess \t Waiting Time TurnAround Time ");
for(i=0;i<n;i++)
 {
printf("\nProces%-12d%-15d%-15d",i+1,p[i].wait,p[i].wait+p[i].burst);
totalwait=totalwait+p[i].wait;
 totalturn=totalturn+p[i].wait+p[i].burst;
}
printf("\n\nAverage\n -------------------- ");
printf("\nWaiting  Time: %fms",totalwait/(float)n); p
rintf("\nTurnAround Time : %fms\n\n",totalturn/(float)n);
return 0;
}
```

OUTPUT:

```
$ cc rr.c
 $ ./a.out

 Enter The No Of Process: 3
 Enter The Quantum time (in ms): 5
 Enter The Burst Time (in ms) For Process # 1: 25
 Enter The Burst Time (in ms) For Process # 2: 30
 Enter The Burst Time (in ms) For Process # 3: 54
 Order Of Execution
 Process     Starting  Ending  Remaining
               Time     Time      Time
process # 1    0         5        20
process # 2    5         10        25
process # 3    10         15        49
process # 1    15         20        15
process # 2    20         25        20
process # 3    25         30        44
process # 1    30         35        10
process # 2    35         40        15
process # 3    40         45        39
process # 1    45         50        5
process # 2    50         55        10
process # 3    55         60        34
process # 1    60         65        0
process # 2    65         70        5
process # 3    70         75        29
process # 2    75         80        0
process # 3    80         85        24
process # 3    85         90        19
process # 3    90         95        14
process # 3    95         100        9
Process # 3    100         105        4
process # 3    105         109        0


Process           Waiting Time     TurnAround Time
Process # 1             40               65
Process # 2          50            80
Process # 3          55            109
Average ------------------ Waiting Time: 48.333333ms
TurnAround Time: 84.666667ms
```

RESULT: Thus the program was executed and verified successfully.

PROGRAM :

SOURCE CODE:

/* A program to simulate the Priority scheduling algorithm */

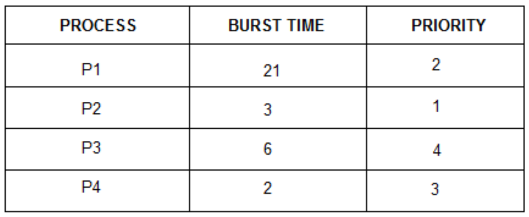
Priority scheduling is a non-preemptive algorithm and one of the most common scheduling algorithms in batch systems. Each process is assigned first arrival time (less arrival time process first) if two processes have same arrival time, then compare to priorities (highest process first). Also, if two processes have same priority then compare to process number (less process number first). This process is repeated while all process get executed.

Implementation –

- First input the processes with their arrival time, burst time and priority.
- First process will schedule, which have the lowest arrival time, if two or more processes will have lowest arrival time, then whoever has higher priority will schedule first.
- Now further processes will be schedule according to the arrival time and priority of the process. (Here we are assuming that lower the priority number having higher priority). If two process priority are same then sort according to process number.
  Note: In the question, They will clearly mention, which number will have higher priority and which number will have lower priority.

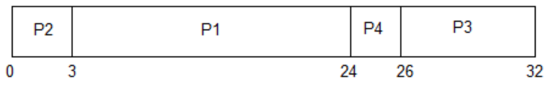
- Once all the processes have been arrived, we can schedule them based on their priority.

Consider the below table for processes with their respective CPU burst times and the priorities.



PROCESS	BURST TIME	PRIORITY
P1	21	2
P2	3	1
P3	6	4
P4	2	3

The GANTT chart for following processes based on Priority scheduling will be,



The average waiting time will be, (0 + 3 + 24 + 26)/4 = 13.25 ms

SOURCE CODE

```
#include<stdio.h>
void main()
 {
  int x,n,p[10],pp[10],pt[10],w[10],t[10],awt,atat,i;
  printf("Enter the number of process : ");
  scanf("%d",&n);
  printf("\n Enter process : time priorities \n");
  for(i=0;i<n;i++)
   {
    printf("\nProcess no %d : ",i+1);
```

```
     scanf("%d %d",&pt[i],&pp[i]);
     p[i]=i+1;
    }
  for(i=0;i<n-1;i++)
   {
     for(int j=i+1;j<n;j++)
     {
      if(pp[i]<pp[j])
      {
        x=pp[i];
        pp[i]=pp[j];
        pp[j]=x;
        x=pt[i];
        pt[i]=pt[j];
        pt[j]=x;
        x=p[i];
        p[i]=p[j];
        p[j]=x;
      }
   }
}
w[0]=0;
awt=0;
t[0]=pt[0];
atat=t[0];
for(i=1;i<n;i++)
 {
   w[i]=t[i-1];
   awt+=w[i];
   t[i]=w[i]+pt[i];
   atat+=t[i];
 }
printf("\n\n Job \t Burst Time \t Wait Time \t Turn Around Time   Priority \n");
for(i=0;i<n;i++)
printf("\n %d \t\t %d  \t\t %d \t\t %d \t\t %d \n",p[i],pt[i],w[i],t[i],pp[i]);
awt/=n;
atat/=n;
printf("\n Average Wait Time : %d \n",awt);
printf("\n Average Turn Around Time : %d \n",atat);

}
```

OUTPUT:

```
 $ cc rr.c
  $ ./a.out

Enter the number of process : 4
Enter process : time priorities
Process no 1 : 3 1

Process no 2 : 4 2

Process no 3 : 5 3

Process no 4 : 6 4
```

```
Job      Burst Time          Wait Time     Turn Around Time    Priority

4             6                   0               6               4

3             5                   6              11               3

2             4                  11               15              2

1             3                  15                18             1

Average Wait Time : 8

Average Turn Around Time : 12
```

PROGRAM 3:

Develop a C program to simulate producer-consumer problem using semaphores.

To solve the Producer-Consumer problem three **semaphores variable** are used :

Semaphores are variables used to indicate the number of resources available in the system at a particular time. semaphore variables are used to achieve `Process Synchronization.

Full

The full variable is used to track the space filled in the buffer by the Producer process. It is initialized to 0 initially as initially no space is filled by the Producer process.

Empty

The Empty variable is used to track the empty space in the buffer. The Empty variable is initially initialized to the

BUFFER-SIZE as initially, the whole buffer is empty.

Mutex

Mutex is used to achieve mutual exclusion. mutex ensures that at any particular time only the producer or the consumer is accessing the buffer.

Mutex - mutex is a binary semaphore variable that has a value of 0 or 1.

We will use the Signal() and wait() operation in the above-mentioned semaphores to arrive at a solution to the Producer-Consumer problem.

Signal() - The signal function increases the semaphore value by 1.

Wait() - The wait operation decreases the semaphore value by 1.

PROGRAM

SOURCE CODE:

```
#include <stdio.h>
#include <stdlib.h>
int mutex = 1; // Initializing the mutex variable with the value 1.
int full = 0; // Initializing the full variable with the value 0.
```

```
Int empty = 10, data = 0; // empty variable will store the number of empty slots in the
buffer
void producer()// A function that will resemble producers' production of data
{
       --mutex; // decrementing the value of mutex
      ++full; // Increase the number of full slots
    --empty; // decrementing the number of slots available
    data++;// incrementing data which means that the data is produced
    printf("\nProducer produces item number: %d\n", data);
     ++mutex; // incrementing the value of mutex
}
void consumer()// A function that will resemble the consumer's consumption of data
{
   --mutex;
    --full;
    ++empty;
 printf("\nConsumer consumes item number: %d.\n", data);
    data--;
   ++mutex;
}
int main()
{
    int n, i;
    printf("\n1. Enter 1 for Producer"
           "\n2. Enter 2 for Consumer"
           "\n3. Enter 3 to Exit");
    for (i = 1; i > 0; i++)
```

```
{
   printf("\nEnter your choice: ");
   scanf("%d", &n);
 switch (n) // using switch case as there can be multiple types of choice.
   {
   case 1: if ((mutex == 1) && (empty != 0))
      {
         producer();
      }
            else
      {
         printf("The Buffer is full. New data cannot be produced!");
      }
      break;
   case 2:
      if ((mutex == 1) && (full != 0))
      {
         consumer();
      }
      else
      {
         printf("The Buffer is empty! New data cannot be consumed!");
      }
      break;
    case 3:
    exit(0);
    break;
```

```
        }
    }
}
```

OUTPUT:

```
 $ cc rr.c

  $ ./a.out

1. Enter 1 for Producer
2. Enter 2 for Consumer
3. Enter 3 to Exit
Enter your choice: 1

Producer produces item number: 1

Enter your choice: 1

Producer produces item number: 2

Enter your choice: 1

Producer produces item number: 3

Enter your choice: 2

Consumer consumes item number: 3.

Enter your choice: 2

Consumer consumes item number: 2.

Enter your choice: 2

Consumer consumes item number: 1.

Enter your choice: 2
```

```
The buffer is empty! New data cannot be consumed!

Enter your choice: 1

Producer produces item number: 1

Enter your choice: 1

Producer produces item number: 2

Enter your choice: 1
Producer produces item number: 3


Enter your choice: 1


Producer produces item number: 4


Enter your choice: 1


Producer produces item number: 5

Enter your choice: 1

Producer produces item number: 6

Enter your choice: 1

Producer produces item number: 7

Enter your choice: 1

Producer produces item number: 8

Enter your choice: 1
```

```
Producer produces item number: 9

Enter your choice: 1

Producer produces item number: 10

Enter your choice: 1

The buffer is full. New data cannot be produced!

Enter your choice: 2

Consumer consumes item number: 10.

Enter your choice: 2

Consumer consumes item number: 9.

Enter your choice: 2

Consumer consumes item number: 8.

Enter your choice: 2

Consumer consumes item number: 7.

Enter your choice: 2

Consumer consumes item number: 6.

Enter your choice: 2

Consumer consumes item number: 5.

Enter your choice: 2

Consumer consumes item number: 4.
```

```
Enter your choice: 2

Consumer consumes item number: 3.

Enter your choice: 2

Consumer consumes item number: 2.

Enter your choice: 2

consumer consumes item number: 1.

Enter your choice: 2

The buffer is empty! New data cannot be consumed!

Enter your choice: 2

The buffer is empty! New data cannot be consumed!

Enter your choice: 3
```

PROGRAM 4:

Develop a C program which demonstrates interprocess communication between a reader process and a writer process. Use mkfifo, open, read, write and close APIs in your program

In computing, a named pipe (also known as a **FIFO**) is one of the methods for inter-process communication.

- It is an extension to the traditional pipe concept on Unix. A traditional pipe is "unnamed" and lasts only as long as the process.
- A named pipe, however, can last as long as the system is up, beyond the life of the process. It can be deleted if no longer used.
- Usually a named pipe appears as a file and generally processes attach to it for inter-process communication. A FIFO file is a special kind of file on the local storage which allows two or more processes to communicate with each other by reading/writing to/from this file.
- FIFO special file is entered into the filesystem by calling mkfifo() in C. Once we have created a FIFO special file in this way, any process can open it for reading or writing, in the same way as an ordinary file. However, it has to be open at both ends simultaneously before you can proceed to do any input or output operations on it.

Creating a FIFO file: In order to create a FIFO file, a function calls i.e. mkfifo is used.

int mkfifo(const char *pathname, mode_t mode);

mkfifo() makes a FIFO special file with name ***pathname***. Here ***mode*** specifies the FIFO's permissions. It is modified by the process's umask in the usual way: the permissions of the created file are (mode & ~umask).

Using FIFO: As named pipe(FIFO) is a kind of file, we can use all the system calls associated with it i.e. open, read, write, close.

Example Programs to illustrate the named pipe: There are two programs that use the same FIFO.

Program 1 writes first, then reads. The program 2 reads first, then writes. They both keep doing it until terminated.

P1: Write First

```
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    int fd;
    char * myfifo = "/tmp/myfifo";
     mkfifo(myfifo, 0666);
     char arr1[80], arr2[80];
    while (1)
    {
     fd = open(myfifo, O_WRONLY);
     fgets(arr2, 80, stdin);
     write(fd, arr2, strlen(arr2)+1);
     close(fd);
     fd = open(myfifo, O_RDONLY);
     read(fd, arr1, sizeof(arr1));
     printf("User2: %s\n", arr1);
     close(fd);
    }
    return 0;
}
```

P2: Read first

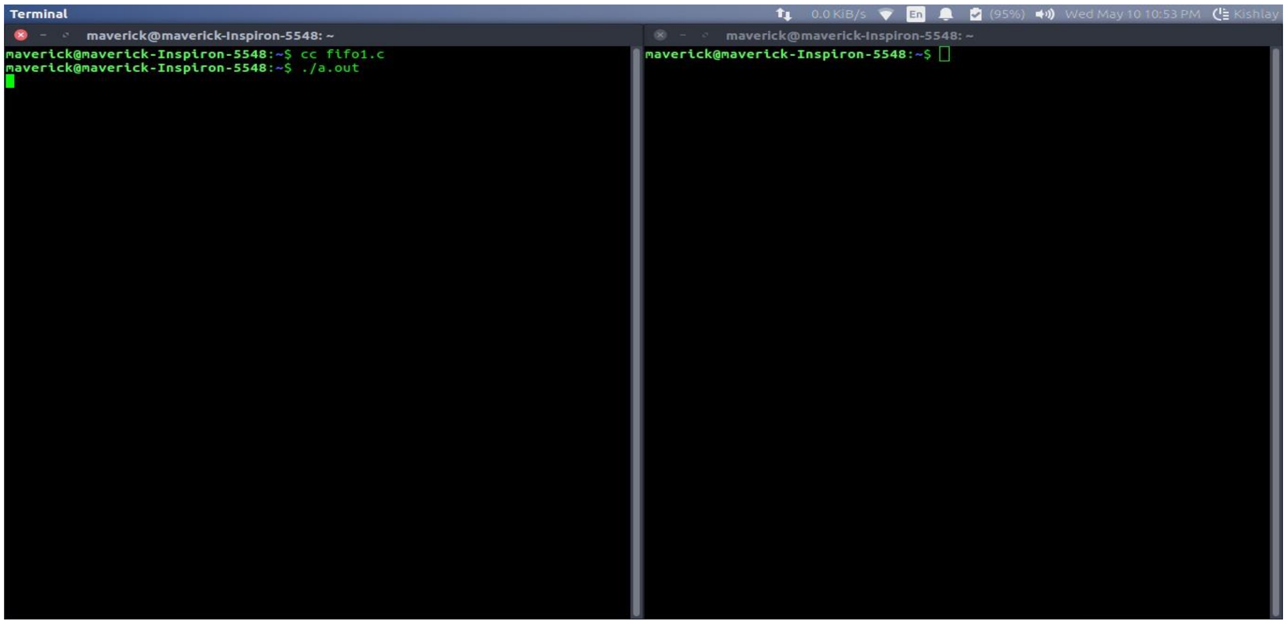
```
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    int fd1;
     char * myfifo = "/tmp/myfifo";
      mkfifo(myfifo, 0666);
    char str1[80], str2[80];
    while (1)
    {
        fd1 = open(myfifo,O_RDONLY);
        read(fd1, str1, 80);
        printf("User1: %s\n", str1);
        close(fd1);
        fd1 = open(myfifo,O_WRONLY);
        fgets(str2, 80, stdin);
        write(fd1, str2, strlen(str2)+1);
        close(fd1);
    }
    return 0;
}
```

OUTPUT:

```
$ cc fifo1.c
$ ./a.out
```

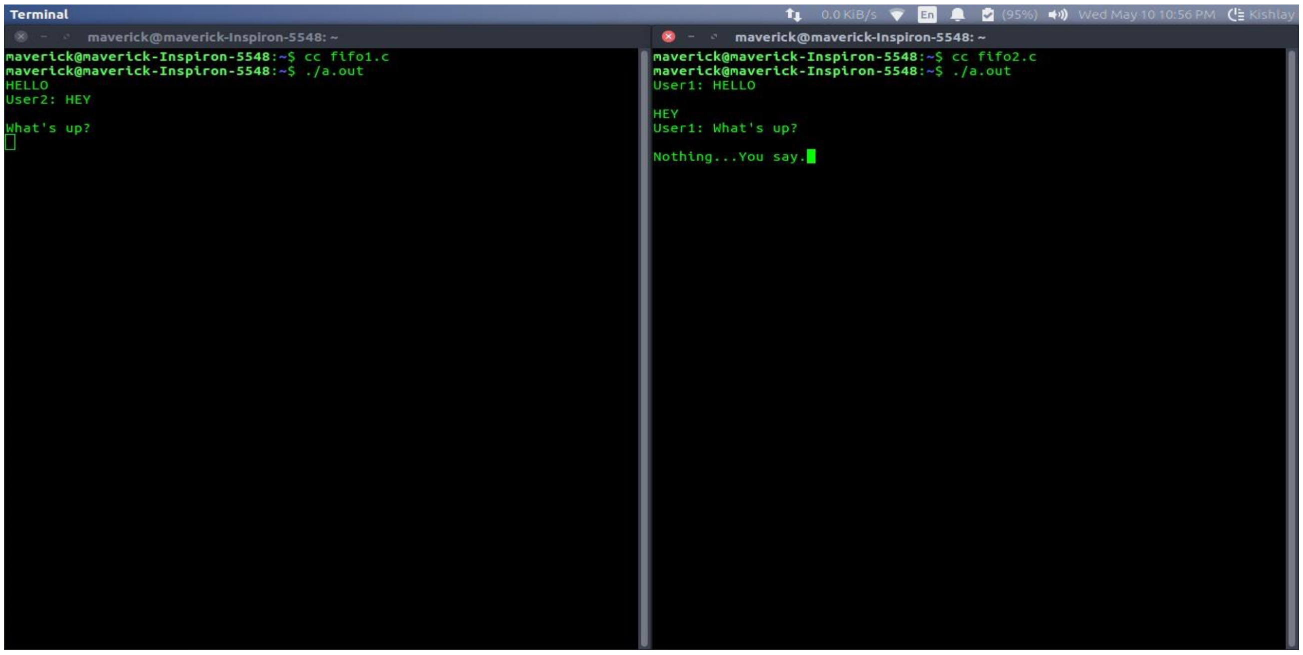
Open two terminals and run the program

```
$ cc fifo2.c
$ ./a.out
```


Terminal
0.0 KiB/s
En
(95%)
Wed May 10 10:53 PM
Kishlay
maverick@maverick-Inspiron-5548: ~
maverick@maverick-Inspiron-5548: ~

```
maverick@maverick-Inspiron-5548:~$ cc fifo1.c
maverick@maverick-Inspiron-5548:~$ ./a.out
```

```
maverick@maverick-Inspiron-5548:~$
```



```
maverick@maverick-Inspiron-5548:~$ cc fifo1.c
maverick@maverick-Inspiron-5548:~$ ./a.out
HELLO
User2: HEY

What's up?
```

```
maverick@maverick-Inspiron-5548:~$ cc fifo2.c
maverick@maverick-Inspiron-5548:~$ ./a.out
User1: HELLO

HEY
User1: What's up?

Nothing...You say.
```

PROGRAM 5:

Develop a C program to simulate Bankers Algorithm for DeadLock Avoidance.

DEAD LOCK AVOIDANCE

AIM: Simulate bankers algorithm for Dead Lock Avoidance (Banker‘s Algorithm)

DESCRIPTION:

Deadlock is a situation where in two or more competing actions are waiting f or the other to finish, and thus neither ever does. When a new process enters a system, it must declare the maximum number of instances of each resource type it needed. This number may exceed the total number of resources in the system.

When the user request a set of resources, the system must determine whether the allocation of each resources will leave the system in safe state. If it will the resources are allocation; otherwise the process must wait until some other process release the resources.

Data structures Allocation:

If Allocation [i, j]=k, Pi allocated to k instances of resource Rj Need: If

Need[I, j]=k, Pi may need k more instances of resource type Rj, Need[I, j]=Max[I, j]-Allocation[I, j];

Safety Algorithm

Work and Finish be the vector of length m and n respectively,

Work=Available and Finish[i] =False.

1. Find an i such that both
2. Finish[i] =False Need<=Work If no such I exists go to step 4.
3. work= work + Allocation, Finish[i] =True;
4. if Finish[1]=True for all I, then the system is in safe state. Resource request algorithm
   Let Request i be request vector for the process Pi, If request i=[j]=k, then process Pi wants k instances of resource type Rj.
1. if Request<=Need I go to step 2. Otherwise raise an error condition.
2. if Request<=Available go to step 3. Otherwise Pi must since the resources are available.
3. Have the system pretend to have allocated the requested resources to process Pi by modifying the state as follows;

Available=Available-Request I; Allocation I=Allocation +Request I; Need i=Need i- Request I;
If the resulting resource allocation state is safe, the transaction is completed and process Pi is allocated its resources. However if the state is unsafe, the Pi must wait for Request i and the old resource-allocation state is restored.

ALGORITHM:

1. Start the program.
2. Get the values of resources and processes.
3. Get the avail value.
4. After allocation find the need value.
5. Check whether its possible to allocate.
6. If it is possible then the system is in safestate.
7. Else system is not in safety state.
8. If the new request comes then check that the system is in safety.
9. or not if we allow the request.
10. stop the program.
11. End.

SOURCE CODE:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int work[5],avl[5],alloc[10][10],l;
int need[10][10],n,m,I,j,avail[10],max[10][10],k,count,i,fcount=0,pr[10];
char finish[10]={'f','f','f','f','f','f','f','f','f','f'};
printf("\n enter the no of process");
scanf("%d",&n);
printf("\n enter the no of resources");
scanf("%d",&m);
printf("\n enter the total no of resources");
for(i=1;i<=m;i++)
scanf("%d",&avail[i]);
printf("\n enter the max resources req by each pr alloc matrix");
for(i=1;i<=n;i++)
for(j=1;j<=m;j++)
scanf("%d",&max[i][j]);
printf("\n process allocation matrix");
for(i=1;i<=n;i++)
for(j=1;j<=m;j++)
scanf("%d",&alloc[i][j]);
for(i=1;i<=n;i++)
for(j=1;j<=m;j++)
need[i][j]=max[i][j]-alloc[i][j];
for(i=1;i<=n;i++)
```

```
{
k=0;
for(j=1;j<=m;j++)
{
k=k+alloc[i][j];
}
avl[i]=avl[i]-k;
work[i]=avl[i];
}
for(k=1;k<=n;k++)
for(i=1;i<=n;i++)
{
count=0;
for(j=1;j<=m;j++)
{
if((finish[i]=='f')&&(need[i][j]<=work[i]))
count++;
}
if(count==m)
{
for(l=1;l<=m;l++)
work[l]=work[l]+alloc[i][l];
finish[i]='t';
pr[k]=i;
break;
}
}
```

```
for(i=1;i<=n;i++)
if(finish[i]=='t')
fcount++;
if(fcount==n)
{
}
else
printf("\n the system is in safe state");
for(i=1;i<=n;i++)
printf("\n %d",pr[i]);
printf("\n the system is not in safe state");
//getch();
}
```

Expected Output:

```
Enter the no of process 5
Enter the no of resources 3
Enter the total no of resources10 5 7
Enter the max resource req. by each pr alloc matrix
7 5 3
3 2 2
9 0 2
2 2 2
4 3 3
Process allocation matrix
0 1 0
2 0 0
3 0 2
```

```
2 1 1
0 2 2
The system is in safe state1
1
3
4
5
2
```

PROGRAM 6

Develop a C program to simulate the following contiguous memory allocation Techniques:

a) Worst fit b) Best fit c) First fit.

DESCRIPTION

One of the simplest methods for memory allocation is to divide memory into several fixed-sized partitions. Each partition may contain exactly one process. In this multiple-partition method, when a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process. The operating system keeps a table indicating which parts of memory are available and which are occupied. Finally, when a process arrives and needs memory, a memory section large enough for this process is provided. When it is time to

load or swap a process into main memory, and if there is more than one free block of memory of sufficient size, then the operating system must decide which free block to allocate. Best-fit strategy chooses the block that is closest in size to the request.

First-Fit

This is a very basic strategy in which we start from the beginning and allot the first hole, which is big enough as per the requirements of the process. The first-fit strategy can also be implemented in a way where we can start our search for the first-fit hole from the place we left off last time.

Best-Fit

This is a greedy strategy that aims to reduce any memory wasted because of internal fragmentation in the case of static partitioning, and hence we allot that hole to the process, which is the smallest hole that fits the requirements of the process. Hence, we need to first sort the holes according to their sizes and pick the best fit for the process without wasting memory.

Worst-Fit

This strategy is the opposite of the Best-Fit strategy. We sort the holes according to their sizes and choose the largest hole to be allotted to the incoming process. The idea behind this allocation is that as the process is allotted a large hole, it will have a lot of space left behind as internal fragmentation. Hence, this will create a hole that will be large enough to accommodate a few other processes.

PROGRAM

WORST-FIT

```
#include<stdio.h>
#define max 25
void main()
{
int
frag[max],b[max],f[max],i,j,nb,nf,t
emp; static int bf[max],ff[max];
printf("\n\tMemory Management Scheme - First Fit");
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]);
}
```

OUT PUT

```
$ cc mm.c
$ ./a.out

Enter the number of blocks: 3
Enter the number of files: 2
Enter the size of the blocks:-
Block 1: 5
Block 2: 2
Block 3: 7
Enter the size of the files:-
File 1: 1
File 2: 4
```

OUTPUT

```
File No     File Size     Block No     Block Size    Fragment
1                1             1             5              4
2                4              3             7             3
```

Program :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\tBlockSize\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]);
}
```

OUTPUT

```
$ cc best.c
$./a.out
Enter the number of blocks: 3
Enter the number of files: 2
Enter the size of the blocks:-
Block 1: 5
Block 2: 2
Block 3: 7
Enter the size of the files:-
File 1: 1
File 2: 4
OUTPUT
File No      File Size     Block No    Block Size    Fragment
1               1             2            2            1
2               4              1             5           1
```

Program :FIRST-FIT

```
#include<stdio.h>
#define max 25
void main()
{
```

```
int
frag[max],b[max],f[max],i,j,nb,nf,temp,highes
t=0; static int bf[max],ff[max];
printf("\n\tMemory Management Scheme - Worst Fit");
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)
{
}
}
frag[i]=highest; bf[ff[i]]=1; highest=0;
}
ff[i]=j; highest=temp;
}
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

```
$ CC first.c
$./a.out
Enter the number of blocks: 3
Enter the number of files: 2
Enter the size of the blocks:-
Block 1: 5
Block 2: 2

Block 3: 7

Enter the size of the files:-

File 1: 1

File 2: 4

OUTPUT

File No    File Size    Block No   Block Size    Fragment

1              1             3               7             6

2              4            1               5             1
```

PROGRAM 7:

Develop a C program to simulate page replacement algorithms: a) FIFO b) LRU

DESCRIPTION:

Page replacement algorithms are an important part of virtual memory management and it helps the OS to decide which memory page can be moved out making space for the currently needed page. However, the ultimate objective of all page replacement algorithms is to reduce the number of page faults.

FIFO-This is the simplest page replacement algorithm. In this algorithm, the operating system keeps track of all pages in the memory in a queue, the oldest page is in the front of the queue. When a page needs to be replaced page in the front of the queue is selected for removal.

LRU-In this algorithm page will be replaced which is least recently used

ALGORITHM:

1. Start the process

2. Read number of pages n

3. Read number of pages no

4. Read page numbers into an array a[i]

5. Initialize avail[i]=0 .to check page hit

6. Replace the page with circular queue, while re-placing check page availability in the frame

   Place avail[i]=1 if page is placed in theframe Count page faults

7. Print the results.

8. Stop the process.

A) FIRST IN FIRST OUT

SOURCE CODE:

```
#include<stdio.h>
int fr[3];
void main()
{
void display();
int i,j,page[12]={2,3,2,1,5,2,4,5,3,2,5,2};
int
flag1=0,flag2=0,pf=0,frsize=3,top=0;
for(i=0;i<3;i++)
{
fr[i]=-1;
}
for(j=0;j<12;j++)
{
```

```
flag1=0; flag2=0;
 for(i=0;i<12;i++)
{
if(fr[i]==page[j])
{
flag1=1; flag2=1; break;
}}
if(flag1==0)
{
for(i=0;i<frsize;i++)
 {
if(fr[i]==-1)
{
fr[i]=page[j]; flag2=1;
break;
}
}
}
if(flag2==0)
{
fr[top]=page[j];
top++;
pf++;
if(top>=frsize)
top=0;
}
display();
 }
printf("Number of page faults : %d ",pf+frsize);
}
void display()
{
int i; printf("\n");
for(i=0;i<3;i++)
printf("%d\t",fr[i]);
}
```

OUTPUT:

```
$cc a2.c
$ ./a.out
2   -1   -1
2    3   -1
2    3   -1
2     3  1
5    3    1
5    2   1
5    2   4
5    2    4
3    2   4
```

```
3   2  4

3   5  4

3   5  2

Number of page faults: 9
```

B) LEAST RECENTLY USED

AIM: To implement LRU page replacement technique.

ALGORITHM:

1. Start the process
2. Declare the size
3. Get the number of pages to be inserted
4. Get the value
5. Declare counter and stack
6. Select the least recently used page by counter value
7. Stack them according the selection.
8. Display the values
9. Stop the process

SOURCE CODE :

```
#include<stdio.h>
int fr[3];
```

```
void main()
{
void display();
int p[12]={2,3,2,1,5,2,4,5,3,2,5,2},i,j,fs[3];
int index,k,l,flag1=0,flag2=0,pf=0,frsize=3;
for(i=0;i<3;i++)
{
fr[i]=-1;
}
for(j=0;j<12;j++)
{
flag1=0,flag2=0;
for(i=0;i<3;i++)
{
if(fr[i]==p[j])
{
flag1=1;
flag2=1; break;
}
}
if(flag1==0)
{
for(i=0;i<3;i++)
{
```

```
if(fr[i]==-1)
{
fr[i]=p[j]; flag2=1;
break;
}
}
}
if(flag2==0)
{
for(i=0;i<3;i++)
fs[i]=0;
for(k=j-1,l=1;l<=frsize-1;l++,k--)
{
for(i=0;i<3;i++)
{
if(fr[i]==p[k])
 fs[i]=1;
}}
for(i=0;i<3;i++)
{
if(fs[i]==0)
index=i;
}
fr[index]=p[j];
```

```
pf++;
}
display();
 }
printf("\n no of page faults :%d",pf+frsize);


 }
void display()
 {
int i; printf("\n");
for(i=0;i<3;i++)
printf("\t%d",fr[i]);
}
```

OUTPUT:

```
2   -1   -1
2    3   -1
2    3   -1
2    3    1
2    5    1
2    5    1
2    5    4
2    5    4
3    5    4
```

```
3    5    2

3    5    2

3     5   2

No of page faults: 7
```

LRU

AIM: To Simulate LRU page replacement algorithms.

ALGORITHM:

Step 1: Start the program.

Step 2: Read the number of frames.

Step 3: Read the number of pages.

Step 4: Read the page numbers.

Step 5: Initialize the values in frames to -1.

Step 6: Allocate the pages in to frames by selecting the page that has not been used for the longest period of time.

Step 7: Display the number of page faults. Step 8: Stop the program.

PROGRAM :

SOURCE CODE: /* A program to simulate LRU Page Replacement Algorithm */

```
#include<stdio.h>
Int main()
 {
 int a[5],b[20],p=0,q=0,m=0,h,k,i,q1=1,j,u,n;
char f='F';
printf("Enter the number of pages:");
scanf("%d",&n);
printf("Enter %d Page Numbers:",n);
```

```
for(i=0;i<n;i++) scanf("%d",&b[i]); for(i=0;i<n;i++)
 {
if(p==0)
{
if(q>=3)
q=0;
a[q]=b[i];
if(q1<3) { q1=q; }
}
printf("\n%d",b[i]);
printf("\t"); for(h=0;h<q1;h++)
printf("%d",a[h]);
if((p==0)&&(q<=3))
{
 printf("-->%c",f);  m++; } p=0; if(q1==3)
{
 for(k=0;k<q1;k++)
 {
if(b[i+1]==a[k]) p=1;
}
for(j=0;j<q1;j++)
{
 u=0; k=i;
while(k>=(i-1)&&(k>=0))
{
if(b[k]==a[j])
u++;
 k--;
}
 if(u==0)
q=j;
}
}
else
{
for(k=0;k<q;k++)
 {
if(b[i+1]==a[k])
 p=1;
 }
}
}
 printf("\nNo of faults:%d",m);
}
```

OUTPUT:

```
$ cc lru.c
$ ./a.out
 Enter the number of pages: 12
Enter 12 Page Numbers:


 2          2-->F
 3          23-->F
2          23
1          231-->F
5          531-->F
2          521-->F
4          524-->F
5          524
3          324-->F
2          324
5          354-->F
2          352-->F
```

No of faults: 7

RESULT: Thus the program was executed and verified successfully.

PROGRAM 8

Simulate following File Organization Techniques

a) Single level directory b) Two level directory

SINGLE LEVEL DIRECTORY:

AIM: Program to simulate Single level directory file organization technique.

DESCRIPTION: The directory structure is the organization of files into a hierarchy of folders. In a single-level directory system, all the files are placed in one directory. There is a root directory which has all files. It has a simple architecture and there are no sub directories. Advantage of single level directory system is that it is easy to find a file in the directory.

SOURCE CODE :

```
#include<stdio.h>

#include<stdlib.h>

#include<string.h>

struct

{

char dname[10],f

name[10][10];

int fcnt; }dir;

void main()

{

int i,ch;

char f[30];
```

```
dir.fcnt = 0;
printf("\nEnter name of directory -- ");
scanf("%s", dir.dname);
while(1)
{
printf("\n\n1. Create File\t2. Delete File\t3. Search File \n 4. Display Files\t5. Exit\nEnter your choice -- ");
scanf("%d",&ch);
switch(ch)
{
 case 1: printf("\nEnter the name of the file -- ");
scanf("%s",dir.fname[dir.fnt]);
dir.fcnt++;
 break;
case 2: printf("\nEnter the name of the file -- ");
scanf("%s",f);
 for(i=0;i<dir.fname[i]==0)
{
if(strcmp(f, dir.fname[i])==0)
{
```

```
printf("File %s is deleted ",f);

strcpy(dir.fname[i],dir.fname[dir.fcnt-1]);

break;

}

}

if(i==dir.fcnt)

printf("File %s not found",f);

else

dir.fcnt--;

break;

case 3: printf("\nEnter the name of the file -- ");

scanf("%s",f);

for(i=0;i<dir.fcnt;i++)

{

if(strcmp(f, dir.fname[i])==0)

{

printf("File %s is found ", f);

break;

}

}
```

```
if(i==dir.fcnt)

printf("File %s not found",f);

break;

case 4: if(dir.fcnt==0)

printf("\nDirectory Empty");

else

{

printf("\nThe Files are -- ");

for(i=0;i<dir.fcnt;i++)

printf("\t%s",dir.fname[i]);

}

break;

default: exit(0);

}

}

}
```

OUTPUT:

```
$cc f1.c
$./a.out
```

```
Enter name of directory -- CSE

1. Create File 2. Delete File 3. Search File

4. Display Files 5. Exit Enter your choice – 1

Enter the name of the file -- A

1. Create File 2. Delete File 3. Search File

4. Display Files 5. Exit Enter your choice – 1

Enter the name of the file -- B

1. Create File 2. Delete File 3. Search File

4. Display Files 5. Exit Enter your choice – 1

Enter the name of the file -- C

1. Create File 2. Delete File 3. Search File

4. Display Files 5. Exit Enter your choice – 4

The Files are -- A B C

1. Create File 2. Delete File 3. Search File

4. Display Files 5. Exit Enter your choice – 3

Enter the name of the file – ABC File

ABC not found

1. Create File 2. Delete File 3. Search File

4. Display Files 5. Exit Enter your choice – 2

Enter the name of the file – B

File B is deleted

1. Create File 2. Delete File 3. Search File

4. Display Files 5. Exit Enter your choice – 5
```

TWO LEVEL DIRECTORY

AIM: Program to simulate two level file organization technique

Description:

In the two-level directory system, each user has own user file directory (UFD). The system maintains a master block that has one entry for each user. This master block contains the addresses of the directory of the users. When a user job starts or a user logs in, the system's master file directory (MFD) is searched. When a user refers to a particular file, only his own UFD is searched.

SOURCE CODE :

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
struct
{
char dname[10],fname[10][10];
int fcnt;
}
dir[10];
void main()
{
int i,ch,dcnt,k;
char f[30], d[30];
dcnt=0;
```

```
while(1)
{
printf("\n\n1. Create Directory\t2. Create File\t3. Delete File");
printf("\n4. Search File\t\t5. Display\t6. Exit\t Enter your choice --");
scanf("%d",&ch);
switch(ch)
{
case 1: printf("\nEnter name of directory -- ");
scanf("%s", dir[dcnt].dname);
dir[dcnt].fcnt=0;
dcnt++;
printf("Directory created");
break;
case 2: printf("\nEnter name of the directory -- ");
scanf("%s",d);
for(i=0;i<dcnt;i++)
if(strcmp(d,dir[i].dname)==0)
{
printf("Enter name of the file -- ");
scanf("%s",dir[i].fname[dir[i].fcnt]);
dir[i].fcnt++;
printf("File created");
}
if(i==dcnt)
```

```
printf("Directory %s not found",d);
break;
case 3: printf("\nEnter name of the directory -- ");
scanf("%s",d);
for(i=0;i<dcnt;i++)
for(i=0;i<dcnt;i++)
{
if(strcmp(d,dir[i].dname)==0)
{
printf("Enter name of the file -- ");
scanf("%s",f);
for(k=0;k<dir[i].fcnt;k++)
{
if(strcmp(f, dir[i].fname[k])==0)
{
printf("File %s is deleted ",f);
dir[i].fcnt--;
strcpy(dir[i].fname[k],dir[i].fname[dir[i].fcnt]);
goto jmp;
}
}
printf("File %s not found",f); goto jmp;
}
}
```

```
printf("Directory %s not found",d);
jmp : break;
case 4: printf("\nEnter name of the directory -- ");
scanf("%s",d);
for(i=0;i<dcnt;i++)
{
if(strcmp(d,dir[i].dname)==0)
{
printf("Enter the name of the file -- ");
scanf("%s",f);
for(k=0;k<dir[i].fcnt;k++)
{
if(strcmp(f, dir[i].fname[k])==0)
{
printf("File %s is found ",f); goto jmp1;
}
}
printf("File %s not found",f); goto jmp1;
}
}
printf("Directory %s not found",d); jmp1: break;
case 5: if(dcnt==0)
printf("\nNo Directory's ");
else
```

```
{
printf("\nDirectory\tFiles");
for(i=0;i<dcnt;i++)
{
printf("\n%s\t\t",dir[i].dname);
for(k=0;k<dir[i].fcnt;k++)
printf("\t%s",dir[i].fname[k]);
}
}
break;
default:exit(0);
}
}
}
```

OUTPUT

$ cc f2.c

$./a.out

```
1. Create Directory 2. Create File 3. Delete File
4. Search File 5. Display 6. Exit
Enter your choice -- 1
Enter name of directory -- DIR1 Directory created
1. Create Directory 2. Create File 3. Delete File
4. Search File 5. Display 6. Exit Enter your choice -- 1
Enter name of directory -- DIR2 Directory created
```

```
1. Create Directory 2. Create File 3. Delete File
4. Search File 5. Display 6. Exit Enter your choice -- 2
Enter name of the directory – DIR1
Enter name of the file -- A1
File created
1. Create Directory 2. Create File 3. Delete File
4. Search File 5. Display 6. Exit
Enter your choice -- 2
Enter name of the directory – DIR1
Enter name of the file -- A2
File created
1. Create Directory 2. Create File 3. Delete File
4. Search File 5. Display 6.
Exit Enter your choice – 6
```

VIVA QUESTIONS

1. Define directory?
2. List the different types of directory structures?
3. What is the advantage of hierarchical directory structure?
4. Which of the directory structures is efficient? Why?
5. What is acyclic graph directory?

PROGRAM 9

Develop a C program to simulate the Linked file allocation strategies.

A. SEQUENTIAL:

AIM:

C program for implementing sequential file allocation method.

DESCRIPTION:

The most common form of file structure is the sequential file in this type of file, a fixed format is used for records. All records (of the system) have the same length, consisting of the same number of fixed length fields in a particular order because the length and position of each field are known, only the values of fields need to be stored, the field name and length for each field are attributes of the file structure.

ALGORITHM:

```
Step 1: Start the program.

Step 2: Get the number of files.

Step 3: Get the memory requirement of each file.

Step 4: Allocate the required locations to each in sequential order a).

Randomly select a location from available location s1= random(100);

a) Check whether the required locations are free from the selected location.

if(b[s1].flag==0)

{

for (j=s1;j<s1+p[i];j++)

{

if((b[j].flag)==0)
```

```
count++;
}
if(count==p[i])
break;
}
```

a) Allocate and set flag=1 to the allocated locations.

```
 for(s=s1;s<(s1+p[i]);s++)
{
k[i][j]=s;
 j=j+1;
 b[s].bno=s;
b[s].flag=1;
}
```

Step 5: Print the results file no, length, Blocks allocated. Step

Step 6: Stop the program

SOURCE CODE :

```
#include<stdio.h>
Int main()
{
int f[50],i,st,j,len,c,k;
for(i=0;i<50;i++)
```

```
f[i]=0;

X:

printf("\n Enter the starting block & length of file");

scanf("%d%d",&st,&len);

for(j=st;j<(st+len);j++)

if(f[j]==0)

{

f[j]=1;

printf("\n%d->%d",j,f[j]);

}

else

{

printf("Block already allocated");

break;

}

if(j==(st+len))

printf("\n the file is allocated to disk");

printf("\n if u want to enter more files?(y-1/n-0)");

scanf("%d",&c);

if(c==1)
```

```
goto X;

else

exit();

}
```

OUTPUT:

```
Enter the starting block & length of file 4 10

4->1

5->1

6->1

7->1

8->1

9->1

10->1

11->1

12->1

13->1

The file is allocated to disk.
```

B. INDEXED

AIM: To implement allocation method using chained method

DESCRIPTION:

In the chained method file allocation table contains a field which points to starting block of memory. From it for each bloc a pointer is kept to next successive block. Hence,there is no external fragmentation.

ALGORITHM:

Step 1: Start the program.

Step 2: Get the number of files.

Step 3: Get the memory requirement of each file.

Step 4: Allocate the required locations by selecting a location randomly q= random(100);

a) Check whether the selected location is free .

b) If the location is free allocate and set flag=1 to the allocated locations.

```
q=random(100);
{
if(b[q].flag==0)
b[q].flag=1;
b[q].fno=j;
r[i][j]=q;
```

Step 5: Print the results file no, length ,Blocks allocated.

Step 6: Stop the program

SOURCE CODE :

```
#include<stdio.h>

int f[50],i,k,j,inde[50],n,c,count=0,p;

main()

{

for(i=0;i<50;i++)

f[i]=0;

x: printf("enter index block\t");

scanf("%d",&p);

if(f[p]==0)

{

f[p]=1;

printf("enter no of files on index\t");

scanf("%d",&n);

}

else

{

printf("Block already allocated\n");

goto x; }

for(i=0;i<n;i++)
scanf("%d",&inde[i]);

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

if(f[inde[i]]==1)

{

printf("Block already allocated");

goto x;

}

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

f[inde[j]]=1;

printf("\n allocated");

printf("\n file indexed");

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

printf("\n %d->%d:%d",p,inde[k],f[inde[k]]);

printf(" Enter 1 to enter more files and 0 to exit\t");

scanf("%d",&c);

if(c==1)

goto x;

else

exit();

}
```

OUTPUT:

```
$ CC index.c
$ ./a.out
 enter index block 9
Enter no of files on index 3 1
2 3
Allocated
File indexed
9->1:1
9->2;1
9->3:1 enter 1 to enter more files and 0 to exit
```

C) LINKED:

AIM: To implement linked file allocation technique.

DESCRIPTION: In the chained method file allocation table contains a field which points to starting block of memory. From it for each bloc a pointer is kept to next successive block. Hence, there is no external fragmentation

ALGORTHIM:

Step 1: Start the program.

Step 2: Get the number of files.

Step 3: Get the memory requirement of each file.

Step 4: Allocate the required locations by selecting a location randomly q= random(100);

a) Check whether the selected location is free .

b) If the location is free allocate and set flag=1 to the allocated locations.

While allocating next location address to attach it to previous location

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

{

for(j=0;j<s[i];j++)

{
q=random(100); if(b[q].flag==0)

b[q].flag=1;

b[q].fno=j;

r[i][j]=q;

if(j>0)

{

}

}

p=r[i][j-1]; b[p].next=q;}
```

Step 5: Print the results file no, length ,Blocks

allocated.

Step 6: Stop the program

SOURCE CODE :

```
#include<stdio.h>

main()

{

int f[50],p,i,j,k,a,st,len,n,c;
```

```
for(i=0;i<50;i++) f[i]=0;

printf("Enter how many blocks that are already allocated");

scanf("%d",&p);

printf("\nEnter the blocks no.s that are already allocated");

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

{

scanf("%d",&a);

f[a]=1;

}

X:

printf("Enter the starting index block &

length"); scanf("%d%d",&st,&len); k=len;

for(j=st;j<(k+st);j++)

{

if(f[j]==0)

{

 f[j]=1;

printf("\n%d->%d",j,f[j]);

}

else
{
printf("\n %d->file is already
allocated",j);
k++;
}
}
printf("\n If u want to enter onemore file? (yes-1/no-0)");
scanf("%d",&c);
if(c==1)
goto X;
else
exit();
}
```

OUTPUT:

```
Enter how many blocks that are already allocated 3 Enter the blocks no.s
that are already allocated 4 7 Enter the starting index block & length 3 7 9
3->1
4->1 file is already allocated
5->1
6->1
7->1 file is already allocated
```

```
8->1
9->1file is already allocated
10->1
11->1
12->1
```

VIVA QUESTIONS

1) List the various types of files

2) What are the various file allocation strategies?

3) What is linked allocation?

4) What are the advantages of linked allocation?

5) What are the disadvantages of sequential allocation methods?

PROGRAM 10:

Develop a C program to simulate SCAN disk scheduling algorithm

DESCRIPTION

One of the responsibilities of the operating system is to use the hardware efficiently. For the disk drives, meeting this responsibility entails having fast access time and large disk bandwidth. Both the access time and the bandwidth can be improved by managing the order in which disk I/O requests are serviced which is called as disk scheduling. The simplest form of disk scheduling is, of course, the first-come, first-served (FCFS) algorithm. This algorithm is intrinsically fair, but it generally does not provide the fastest service. In the SCAN algorithm, The disk arm starts at one end, and moves towards the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk. At the other end, the direction of head movement is reversed, and servicing continues. The head continuously scans back and forth across the disk. C-SCAN is a variant of SCAN designed to provide a more uniform wait time. Like SCAN, C-SCAN moves the head from one end of the disk to the other, servicing requests along the way. When the head reaches the other end, however, it immediately returns to the beginning of the disk without servicing any requests on the return trip

SCAN DISK SCHEDULING ALGORITHM

SOURCE CODE

```
#include<stdio.h>

main()

{

int t[20], d[20], h, i, j, n, temp, k, atr[20], tot, p, sum=0;

printf("enter the no of tracks to be traveresed");

scanf("%d'",&n);

printf("enter the position of head");

scanf("%d",&h);
```

```
t[0]=0;t[1]=h;

printf("enter the tracks");

for(i=2;i<n+2;i++)

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

for(i=0;i<n+2;i++)

{

for(j=0;j<(n+2)-i-1;j++)

{

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

{

temp=t[j];

t[j]=t[j+1];

t[j+1]=temp;

} } }

for(i=0;i<n+2;i++)

if(t[i]==h)

j=i;k=i;

p=0;

while(t[j]!=0)

{
```

```
atr[p]=t[j]; j--;

p++;

}

atr[p]=t[j];

for(p=k+1;p<n+2;p++,k++)

atr[p]=t[k+1];

for(j=0;j<n+1;j++)

{

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

d[j]=atr[j]-atr[j+1];

else

d[j]=atr[j+1]-atr[j];

sum+=d[j];

}

printf("\nAverage header movements:%f",(float)sum/n);

}
```

INPUT

```
Enter no.of tracks:9

Enter track position:55 58 60 70 18 90 150 160 184
```

OUTPUT

```
Tracks traversed Difference between tracks

150                    50

160                   10

184                    24

90                     94

70                    20

60                    10

58                     2

55                     3

18                    37
```