MANGALAM COLLEGE OF ENGINEERING

COLLEGE VISION & MISSION STATEMENT

Vision

"To emerge as a center of excellence in technical education and research, creating employable and committed professionals."

Mission

"Inspire the learners to be globally competent engineers through innovative teaching and learning methods and imbibe a sense of social responsibility and creative inquiry in them that leads to higher learning and research."

MANGALAM COLLEGE OF ENGINEERING

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING

Vision

To be center of excellence in Artificial Intelligence and Machine Learning towards serving the greater cause of society.

Mission

Mould the young adept to state of the art academic and infrastructural facilities with modern equipment and Artificial Intelligence learning resources to create self-sustainable

Program Educational Objectives (PEO)

PEO1: To Formulate, analyze and solve Engineering problems with strong foundation in Mathematical, Scientific, Engineering fundamentals and modern computing practices through advanced curriculum.

PEO2: Analyze the requirements, realize the technical specification and design the Engineering solutions by applying the principles of Artificial Intelligence and Machine Learning.

PEO3: Demonstrate technical skills, competency in computing and Artificial Intelligence, promote collaborative learning and team work spirit through multi - disciplinary projects and diverse professional activities.

Program Specific Outcomes

PSO1: Apply the computing principles of artificial Intelligence and Machine Learning in the multi–disciplinary areas.
PSO2: Demonstrate engineering knowledge in Artificial Intelligence and Machine Learning learned to solve real time problems in various domains.

List of Experiments

EXPERIMENT - 1
Basic Linux commands

1.a) INTRODUCTION TO LINUX/UNIX

AIM:
To study the basics of UNIX/LINUX.

UNIX:
In 1969, AT & T Bell Industries, USA developed UNIXand is a multi-user operating system.
Ken Thomson and Dennis Ritchie developed UNIX OSfeatures influenced from MULTICS (Multiplexed Information and Computing Service) OS. The development of UNIX operating systemstarted in 1969, and its code was rewritten in C in 1972.

LINUX:
Linux is an open-source software and is similar to UNIX, which was created by Linus Torualds. All UNIX commands worksin Linux. The features of Linux issimilar to other OS such as Windows and UNIX.

STRUCTURE OF A LINUX OPERATING SYSTEM:
The Linux operating system consists of three parts.
  a) The UNIX Kernel
  b) The Shell
  c) The Programs - Tools and Applications

UNIX KERNEL:
Kernel is the core of the UNIX OS and is the lowest layer. It allocates processor time and memory to each program and determines when each program will run.Kernel is responsible for the control of all the tasks, schedules all the processes and carries out all the functions of the OS. Kernel decides when to stop an executing program and startanother program.

SHELL:
Shell is the command interpreter in the UNIX OS.The shell acts as an interface between the user (programs) and the kernel. It accepts commands from the user and analyses and interprets them.

UNIX is a multi-user, multi-tasking operating system in which a user can run more than one program or task at a time. The responsibility of kernel is to keep each process and user separate and to regulate access to system hardware, including CPU, memory, disk and other I/O devices.

UNIX is a layered operating system. – Layers are Hardware, Kernel and System calls, Programs
  • **Hardware layer**is the innermost layer which provides the services for the operating system.
  • The operating system, such as UNIX has two parts – **Kernel** and **Shell**

- **Kernel** interacts directly with the hardware resourcesand offers the services to the user programs.
- **User programs** interact with the kernel through a set of standard **system calls**. These system calls request services to be provided by the kernel.
- The services may be accessing a file: open, close, read, write, link, or execute a file; starting or updating accounting records; changing ownership of a file or directory; altering files to a new directory; creating, suspending, or killing a process; enabling access to hardware devices; and setting limits on system resources.

HISTORY OF UNIX

| Year	Developments
Mid of 1960's (1965)	Multiplexed Operating and Computing System (MULTICS) is the system in which the Unix operating system was used at the beginning. The MULTICS project began as a joint venture of General Electric, Massachusetts Institute for Technology and AT&TBell Laboratories.
1969	Bell Laboratories pulled out of the project in 1969.In 1969, Ken Thompson wrote the first version of Unix, called UNICS. UNICS stands for Uniplexed Operating and Computing System.
1970	The development of UNIX rises with the acquisition of a DEC (Digital Equipment Corporation) PDP-11computer. PDP-11computer has 24 kilobytes of RAM and 512 kilobytes of disk space.
1971	Sixth version of UNIX was used only in Bell Labs.UNIX was used (with the assembly-language-coded *troff*) in the Bell Labs patent department as one of the first document-processing programs.
1973	In 1973 Dennis Ritchie and Ken Thompson rewrote the Unix kernel in C. A year later,the Fifth Edition of UNIX was first licensed to universities.
1978	The Seventh Edition of UNIX, released in 1978, served as a dividing point for two divergent lines of Unix development. The two branches are known as BSD andSVR4 (System V).

FEATURES OF UNIX Operating System

The UNIX operating system is the most popular operating system. The key features that made the UNIX operating system popularare:

- Multi-user operating system
- Hierarchical file system
- Time sharing
- Background processing
- Multitasking
- Security and Protection
- Better communication
- Includescompilers, tools and utilities

- Portability
- Shell programming

THE KERNEL AND THE SHELL
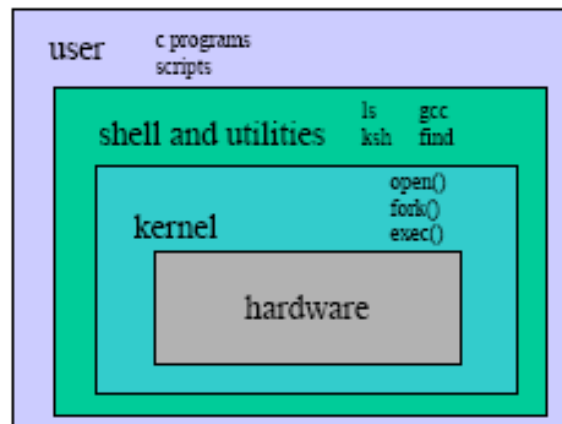
Kernelis the main control program of UNIX operating system. The kernel does not allow the user to execute its commands directly; instead, the user need to type commands on another program in the operating system called a **shell. Shell program** parses the commands, checks the syntax, translates and passes them to the kernel for execution.

The kernel will send the response when the command has been correctly interpreted and executed.The response may merely be a **prompt** for the entering next command, or displays output in the monitor.
If the kernel cannot execute the requested command due to wrong syntax, then the response will be an error message; and the user must re-enter the command with correct syntax.

Commonly used Unix operating systemshells are:
- The Bourne Shell or POSIX shell – /bin/shis a symlink to a Bourne shell.
- C shell – /bin/cshis a symlink to the C shell, or a compatible shell
- Enhanced C Shell – /bin/tcsh - Tcsh is an enhanced version of the csh
- Bash shell – /bin/bash - Bash shell interpreter is located.BourneAgain shell (called bash)
- Korn shell – /bin/ksh - Korn shell offers much more programming features and is superior in comparison to the Bash shell.

Files and processes

In UNIX operating system, everythingis represented or stored either as a file or as a process.

- A **file** is a collection of data. Files are created by users using text editors, running compilers etc.
  - Examples of files are:
  - a typed document (report, essay etc.) prepared by word or text editors.
  - the text of a program written in some high-level programming language
  - instructions comprehensible directly to the machine and incomprehensible to a casual user, for example, a collection of binary digits (an executable or binary file);
  - a directory, containing information about its contents, which may be a mixture of other directories (subdirectories) and ordinary files.

- A **process** is an program under execution which can be identified by a unique PID. PID stands for process identifier.

The Directory Structure

In UNIX like operating system, all the files are usually organized in the directory structure. The UNIX filesystem is represented as a hierarchical structure, similarto an inverted tree. The top of the file system hierarchy is called **root** (written as a "/")
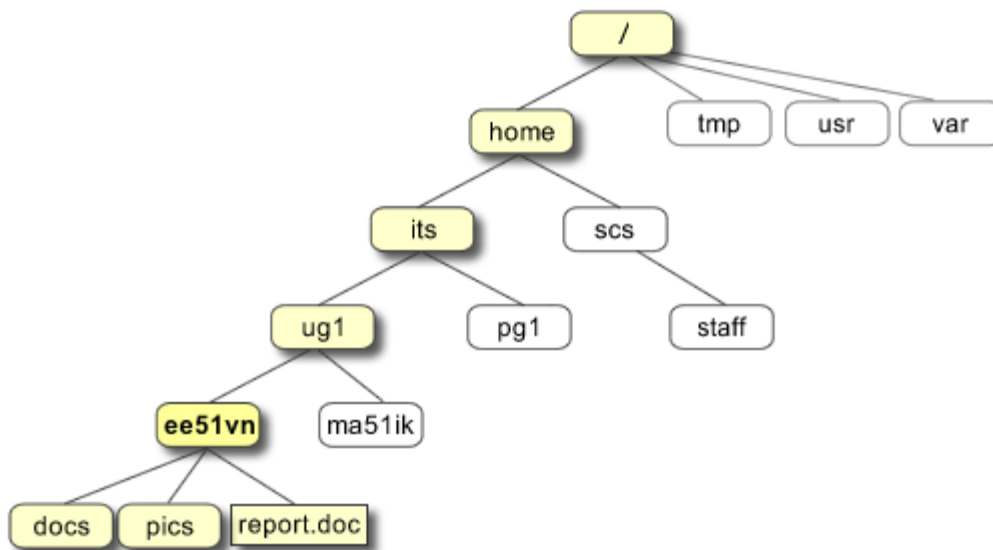
In the diagram above, the directory of the undergraduate student **"ee51vn"** contains two sub-directories (**docs** and **pics**) and a file called **report.doc**.

The full path to the file **report.doc** is **"/home/its/ug1/ee51vn/report.doc"**

The current directory (.)

In UNIX-like OS, (**.**) means the current directory, so typing
$**cd .**
NOTE: there is a space between cd and the dot

The parent directory (..)

In UNIX-like OS,(**..**) means the parent of the current directory, hence
$**cd ..**
will change one directory up the hierarchy (back to the home directory).
Note: typing cd with no argument always returns to the home directory. It is very useful when you are lost in the file system.

A **Shell** provides an interface to the Unix. It gathers input from the user and executes programs based on the user input. When a program finishes itsexecution, it displays that output of the program.

Shell provides an environment in which a user can run commands, execute programs and shell scripts..

Shell Prompt

The prompt, **$**, which is called the **command prompt**, is issued by the shell. While the prompt is displayed, you can type a command.

Shell reads your input after you press **Enter**. It determines the command you want executed by looking at the first word of your input. A word is an unbroken set of characters. Spaces and tabs separate words.

Following is a simple example of the **date** command, which displays the current date and time:

$**date**
Thu Jun 25 08:30:19 MST 2009

Shell Types

In Unix, there are two major types of shells −
- **Bourne shell** − If you are using a Bourne-type shell, the **$** character is the default prompt.
- **C shell** − If you are using a C-type shell, the % character is the default prompt.

The Bourne Shell has the following subcategories −
- Bourne shell (sh)
- Korn shell (ksh)
- Bourne Again shell (bash)
- POSIX shell (sh)

The different C-type shells follow −
- C shell (csh)
- TENEX/TOPS C shell (tcsh)

The original Unix shell was written in the mid-1970s by Stephen R. Bourne while he was at the AT&T Bell Labs in New Jersey.

Bourne shell was the first shell to appear on Unix systems, thus it is referred to as "the shell".

Bourne shell is usually installed as **/bin/sh**on most versions of Unix. For this reason, it is the shell of choice for writing scripts that can be used on different versions of Unix.

Shell Scripts

The basic concept of a shell script is a list of commands, which are listed in the order of execution. A good shell script will have comments, preceded by **#** sign, explaining the steps.

Shell scriptmay contain conditional tests, such as value X is greater than value Y, loops need to go through massive amount of data, files to read and store data, and variables to read and store data, and the shell script may include functions.

In Unix-like operating systems such as Linux, the **chmod**command is used to change the access mode of a file. The command **chmod**stands for**change mode**.

Syntax of chmod:

chmod [reference][operator][mode] file

The **reference** option is used to represent the users to whom the permissions are given. It will be either a single letter or a list of letters which specifies whom has the permissions. The references can be represented by one or more of the following letters:

Reference	Class	Description
u	owner	Permission is granted only to the owner of the file.
g	group	Permission granted to users who are members of the file's group.
o	others	users who are neither the file's owner nor members of the file's group
a	all	All three of the above, i.e., a is same as ugo

The **operator** is used to specify how the permission modes of a file must be adjusted. The three operators are as follows:

Operator Description

+	Adds the specified permission modes to the specified classes.
-	Removes the specified permission modes from the specified classes
=	The modes specified are to be made the exact modes for the specified classes

The **modes** indicate which permissions are to be granted or removed from the specified classes. There are three basic modes which correspond to the basic permissions:

r	Permission to read access to the specified file.
w	Permission to write access to the specified file. Also allows to delete the file.
x	Permission to execute the specified file, or, if it's a directory, then permission to search it.

Types of permissions can be changed using chmod command:
In Linux, ls-l command is used to list all the permissions assigned to different files. Open terminal and type ls-l command, it will list the files in the current working directory (in long format).

1.b) Basic UNIX/LINUX Commands

AIM:
To study basic UNIX commands.

THEORY:

a) date
–It is used to display the date and time.

Syntax: $date

Format	Purpose	Example	Result
+%m	Displays monthonly (in number)	$date+%m	06
+%h	Displays month name	$date+%h	June
+%d	Displays day of month	$date+%d	01
+%y	Print last two digits of year	$date+%y	09
+%H	Print hours	$date+%H	10
+%M	Displays minutes	$date+%M	45
+%S	Displays seconds	$date+%S	55

b) cal
-This command is used to show the calendar.

Syntax: $cal 2 2009

c) echo
- This command is used to print a message on the screen.

Syntax: $echo "text"

  d) ls
-    This command is used to list the files and directories of the current working directory.

Syntax: $ls
Lists all the files and directories

Command	Usage
ls -s (the s is lowercase)	to list files and directories with their sizes
ls -S (the S is uppercase)	to list files and directories in descending order of file size (biggest to smallest).
ls–l	Displays the list in a long list format. (Provide file statistics)
ls–t	Displays the list of files in the order of its creation time.
ls– u	Sort by access time (or show when last accessed together with –l)
ls–r	Displays the list in reverse order
ls–f	Mark directories with /, executable with* , symbolic links with @, local sockets with =,named pipes (FIFOs)with
ls– h	Displays size of the file in human readable format.
ls[a-m]*	to list all the files with filename starts with alphabets "a" to "m".
ls[a]*	to list all the files whose name begins with "a" or "A"
Eg:$ls>filelist	Output of"ls" command is written or stored to a file named "filelist".

e) lp
- This command submits a file for printing.

Syntax: $lp filename

f) man
- This command is used to provide manual help for every UNIX command.

Syntax: $man unix command

$man cat

g) who &whoami
- This command displays the data about all the users who have currently logged into the system.

whoami -This command displays the current user only.

Syntax: $who
        $whoami

h) uptime
- This command tells you how long the computer has been running since its last reboot or power-off.

Syntax: $uptime

i) uname

- This commandprints the system information such as system name,processor,hardware platform and OS type.

Syntax: $uname–a

j) hostname

- This command displays and set system host name.

Syntax: $hostname

k) bc

–This command stands for "**basic calculator**"

$bc 10/2*3 15	$ bc scale =1 2.25+1 3.35 quit	$ bc ibase=2 obase=16 11010011 89275 1010 Ā quit	$ bc sqrt(196) 14 quit
$bc for(i=1;i<3;i=i+1)I 1 2 3 quit	$ bc-l scale=2 s(3.14) 0		

FILE MANIPULATION COMMANDS

a) cat – This command can be used to create, view and concatenate files.

Creation:
Syntax:
$cat>filename

Viewing:
Syntax:
$cat filename

Add text to an existing file:
Syntax:
$cat>>filename

Concatenate:
Syntax:
$cat f1 f2>f3

$cat f1 f2>>f3 (f1, f2 and f3 are three filenames and no over writing of file f3)

b) grep – This command is used to search a particular word or pattern from a file.
Syntax: $grep search word filename
Eg: $grep anu student

c) rm – This command removesthe specified file from the file system.
Syntax: $rm filename

d) touch – This command is used to create a blank file.
Syntax: $touch filename

e) cp – This command copies the files or directories
Syntax: $cp sourcefiledestinationfile
Eg: $cp student stud

f) mv – This command is used to rename the file or directory.
Syntax: $mv old file new file
Eg: $mv–i student studentlist(-i prompt when overwrite)

g) cut– This command cuts or picks up a given number of characters or fields of the file.
Syntax:$cut<option><filename>
Eg: $cut –c filename
$cut–c1-10 student
$cut–f 3, 5 student
$ cut –f 3-8student
-c cutting columns
-f cutting fields

h) head – This command displays 10 lines from the top (head) of the file.
Syntax: $head filename
Eg:$head employee
Command to print the top two lines:
Syntax: $head-2 employee

i) tail – This command displays last 10 lines of the file
Syntax: $tail filename
Eg:$tail student

Command to print the bottom two lines;
Syntax: $tail -2 student

j) chmod – This command is used to change the permission of a file or directory.
Syntax: $chmod category operation permission file

category–It specifies the user type.

operation– It is used to assign or remove permission

Permission– It specifies the type of permission – read, write or execute

File– It specifies the filename to assign or remove permission all

Examples:

$chmod u-wx student

This command removes permission to write and execute for users on student.

$chmodu+rw,g+rw student

This command assigns read and write permission for users and groups

$chmod g=rwx student

This command assigns or changesthe permission for groups and grant read, write and execute permissions.

k) wc – This commandfinds the number of lines, word count, character count in a specified file or files. The wc stands for word count and the options of **wc command**are :–l,-w,-c

wc command displays four-columnar output.

Let file1.txt be a file contains names of fiveIndian states.

$cat file1.txt

Kerala

Assam

Arunachal Pradesh

Madhya Pradesh

Tamil Nadu

wc file1.txt

5 853file1.txt

First column - number of lines, second column - number of words, third column - number of characters present in file, fourth column - file name itself.

Category		Operation		Permission	
u	users	+	assign	r	read
g	group	-	remove	w	write
o	others	=	assign absolutely	x	execute

Syntax:

$wc –l filename

$wc –w filename

$wc–c filename

l)touch : Create a new empty file.

Syntax: touch <filename>

1.c) UNIX EDITORS

AIM:
To study various UNIX editors such as vi, ed, ex and EMACS.

THEORY:
Editor is a program that allows user to see a portion a file on the screen and modify
characters and lines by simply typing at the current position. UNIX supports a variety of Editors. They
are: - ed ex vi

EMACS
Vi - vi stands for "visual". vi is the most important and powerful editor. vi is a full screen editor that
allows user to view and edit entire document at the same time.vi editor was written in the University of
California, at Berkley by Bill Joy, who is one of the co-founder of Sun
Microsystems.

Features of vi:
It is easy to learn and has more powerful features.
It works great speed and is case sensitive. vi has powerful undo functions and has 3 modes:
  1) Command mode
  2) Insert mode
  3) Escape or ex mode

In command mode, no text is displayed on the screen.
In Insert mode, it permits users to edit, insert or replace text.
In escape mode, it displays commands at the command line.
Moving the cursor with the help of h, l, k, j, I, etc

EMACS Editor

Cursor movements / EMACS Commands:
C- means "control key", M- means "meta key"

M->	Move to end of file
M-<	Move to beginning of file
M-f	Move cursor forward one word
M-b	Move cursor backwards one word
M-v	Go backward one screen
C-v	Go forward one screen
C-n	Move cursor to next line
C-p	Move cursor to previous line
C-a	Move cursor to beginning of line
C-e	Move cursor to end of line

C-f	Move cursor forward one character
C-b	Move cursor backwards one character
C-x C-s	Save file

Deletion Commands:

C-d	Delete the character by the cursor.
delete key	Delete the character preceding the cursor
C-x DEL	deletes the previous sentence
M-k	delete the rest of the current sentence
C-k	Delete from the cursor to the end of the line
C-x u	Undo (in order to undo several previous commands)

Search and Replace in EMACS:

y	To replace the occurrence of the pattern with new string
n	Don't change the occurrence, but to skip to the next occurrence without replacing this one.
!	to replace all remaining occurrences of the file without asking again.

RESULT:
The Linux commands are executed and the output is verified.

<div align="center">

EXPERIMENT - 2
Shell programming

</div>

AIM:

To write simple shell programs by using conditional, branching and looping statements.

- a) EVEN OR ODD
- b) LEAP YEAR OR NOT
- c) BIGGEST OF THREE NUMBERS
- d) FACTORIAL OF NUMBER
- e) FIBONACCI SERIES
- f) SWAP TWO VARIABLES

2.a) Write a Shell program to check if the given number is even or odd.

ALGORITHM:

1) Start the program
2) Read the number, n.
3) Calculate "r=expr $n%2".
4) If the value of r equals 0 then print the number is even.
5) If the value of r not equal to 0 then print the number is odd.
6) Stop the program.

PROGRAM:

```
echo "Enter the Number"
read n
r=`expr $n % 2`
if [ $r -eq 0 ]
then
echo "$n is Even number"
else
echo "$n is Odd number"
fi
```

OUTPUT:

```
Enter the Number
4
4 is Even number
Enter the Number
5
5 is Odd number
```

2.b) Write a Shell program to check if the given year is leap year or not.

ALGORITHM:
1) Start the program.
2) Read year in y.
3) Calculate "x=expr $y%4".
4) If the value of x equals 0 then print the year is a leap year.
5) If the value of x not equal to 0 then print the year is not a leap year.
6) Stop the program.

PROGRAM:
```
echo "Enter the year"
read y
x=`expr $y % 4`
if [ $x -eq 0 ]
then
echo "$y is a leap year"
else
echo "$y is not a leap year"
fi
```

OUTPUT:
```
Enter the year
2004
2004 is a leap year
Enter the year
2005
2005 is not a leap year
```

2.c) Write a Program to Find Biggest in Three Numbers.

ALGORITHM:
1) Start the program.
2) Read three numbers.
3) If xis greater than yand xis greater than zthen print "x"is Big.
4) Else If "y" is greater than "z"then "z"is Big.
5) Else print "z"is Big.
6) Stop the program.

PROGRAM:
```
echo "Enter three numbers""
read x y z
if [ $x -gt $y ]&& [ $x -gt $z ]
then
```

```
echo "x is big"
else if [ $y -gt $z ]
then
echo "y is big"
else
echo "z is big"
fi
fi
```

OUTPUT:
Enter three numbers
23 54 78
zis big.

2.d) Write a Shell program to find the factorial of a number.

ALGORITHM:
1) Start the program.
2) Read the number, n.
3) Calculate "i=expr $n-1".
4) If the value of i is greater than 1 then calculate "n=expr $n * $i" and "i=expr $i – 1"
5) Print the factorial of the given number.
6) End program

PROGRAM:
```
echo -n "Enter a Number"
read n
i=`expr $n - 1`
while [ $i -ge1 ]
do
n=`expr $n \* $i`
i=`expr $i - 1`
done
echo "The Factorial of the given Number is $n"
```

OUTPUT:
Enter a Number5
The Factorial of the given Number is 120

2.e) Write a Shell program to print the Fibonacci series.

ALGORITHM:
1) Start the program.
2) Read the Limit, N.

3) Print First two terms of fibonacci series - a and b.
4) For loop from i=0 to i<N
   4.1) Print $a
   4.2) fn = a+b
   4.3) a=$b
   4.4) b=$fn
   4.5) Increment i by 1.
5) Print the Fibonacci seriesupto the given limit.
6) Stop the program.

PROGRAM:
```
# Limit N
N=7
# 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:
The Fibonacci series is :
0 1 1 2 3 5 8

2.f) Write a Shell program to swap the two integers.

ALGORITHM:
1) Start the program.
2) Read two integers a and b.
3) Calculate the swapping of two integers by using a temporary variable temp.
4) Print the swapped values of a and b.
5) Stop the program

PROGRAM:
```
echo "Enter two numbers"
read a b
temp=$a
a=$b
```

b=$temp
echo "after swapping"
echo $a $b

OUTPUT:
Enter Two Numbers
4 5
after swapping
5 4

RESULT:
       Thus, the programshave been executed successfully.

EXPERIMENT - 3
System calls of Linux operating system
(fork, exec, getpid, exit, wait, close, stat, opendir, readdir)

AIM:
To write C Programs using the following system calls of UNIX operating system fork, exec, getpid, exit, wait, close, stat, opendir, readdir.

System Calls
System call is a request to the operating system to perform some activity.
A system call is a procedure that provides the interface between a process and the operating system. It is the way by which a computer program requests a service from the kernel of the operating system.

System calls are divided into 5 categories mainly:
- Process Control
- File Management
- Device Management
- Information Maintenance
- Communication

Process Control :
Process Controlsystem calls perform the task of process creation, execution, process termination, etc. The Linux System calls under this category are fork() , exit() , exec().

File Management:
File management system calls handle file manipulation jobs such as creating a file, reading, and writing, etc. The Linux System calls under this category are open(), read(), write(), close().

Linux System Calls
In Linux, making a system call includes transferring control from unprivileged user mode to privileged kernel mode. When a system call is used, it is communicating to the operating system and in return the OS communicates to the user through the parameters that are returned to system call functions (return values).

Opendir()
The opendir() function is used to open a directory stream corresponding to the specified directory name. It also returns a pointer to the directory stream. The stream is placed at the first entry in the directory.

Readdir()
The readdir() function returns a pointer to a dirent structure representing the next directory entry in the directory stream pointed to by dirp. It returns NULL on reaching the end of the directory stream or if an error occurred.
Closedir()
The closedir() function is used to close the directory stream specified with the directory pointer, dirp.

A successful call to closedir() also closes the underlying file descriptor associated with dirp. The directory stream descriptor dirpwill not be available after this call.

fork()

The fork system call creates a new child process. It creates a *copy* of the current process as a new child process, and then both processes resume execution from the fork() call. Since it creates two processes, fork also returns two values; one to teach process. To the parent process, fork returns the *process id of the newly created child process*. To the child process, fork returns 0. The reason it returns 0 is precisely because this is an invalid process id. You would have no way of differentiating between the parent and child processes if fork returned an arbitrary positive integer to each.

A call to fork looks like this:

int pid;
if ((pid = fork()) == 0) {
/* child process executes inside here */
}
else {
/* parent process executes inside here */
}
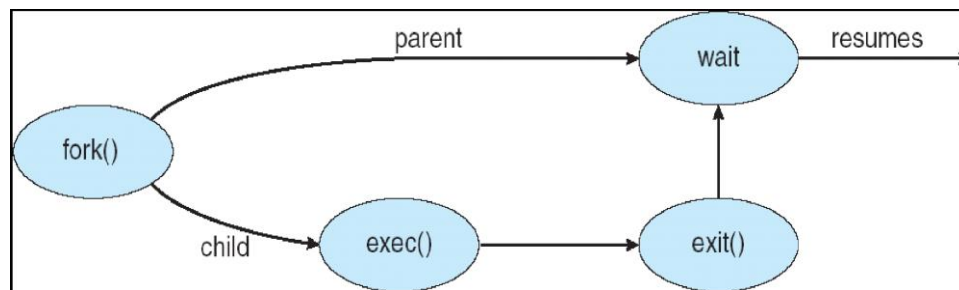
execvp&execlp

The exec functions are a family of functions that *execute* some program *within* the current process space. So, if I write a program that calls one of the exec functions, as soon as the function call succeeds the original process gets *replaced* with whateverprogram I asked the exec to execute. This is usually used along with a fork call. You would typically fork a child process, and then call exec from within the child process, to execute some other program in the new process entry created by fork.

getpid

Each process is identified by a unique *process id* (called a "pid"). The init process (which is the supreme parent to all processes) possesses id 1. All other processes have some other (possibly arbitrary) process id. The getpid system call returns the current process' id as an integer.

// …
int pid = getpid();
printf("This process' id is %d\n",pid);// …

wait & exit

In order to wait for a child process to terminate, a parent process will just execute a **wait()** system call. This call will suspend the parent process until any of its child processes terminates, at which time the **wait()** call returns and the parent process can continue.

The prototype for the **wait()** call is:

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

pid_twait(int *status);

The return value from wait is the **PID** of the child process which terminated. The parameter to **wait()** is a pointer to a location which will receive the child's exit status value when it terminates.

When a process terminates it executes an **exit()** system call, either directly in its own code, or indirectly via library code. The prototype for the **exit()** call is:

#include <std1ib.h>

void exit(int status);

The **exit()** call has no return value as the process that calls it terminates and so couldn't receive a value anyway. Notice, however, that **exit()** does take a parameter value - status. As well as causing a waiting parent process to resume execution, **exit()** also returns the status parameter value to the parent process via the location pointed to by the **wait()** parameter.

3.a) PROGRAM FOR SYSTEM CALLS OF UNIX OPERATING SYSTEMS (OPENDIR, READDIR, CLOSEDIR)

ALGORITHM:
1) Start the program.
2) Create structure variable, struct dirent.
3) Declare a variable buffer and pointer dptr.
4) Read the name of the directory.
5) Open the directory.
6) Read the contents in directory and display the content details.
7) Close the directory.
8) Stop the program

PROGRAM:
```
/* geditdircommands.c */
#include<stdio.h>
#include<stdlib.h>
#include<dirent.h>
struct dirent *dptr;
int main(int argc, char *argv[])
{
char buffer[100];
DIR *dirp;
```

```
printf("\n ENTER DIRECTORY NAME");
scanf("%s", buffer);
if((dirp=opendir(buffer))==NULL)
{
printf("The given directory does not exist");
exit(1);
}
while(dptr=readdir(dirp))
{
printf("%s\n",dptr->d_name);
}
closedir(dirp);
}
```

OUTPUT:
ubuntu@ubuntu:~/skj$ geditdircommands.c
ubuntu@ubuntu:~/skj$ gccdircommands.c
ubuntu@ubuntu:~/skj$ ./a.out
ubuntu@ubuntu:~/skj$ mkdir folder1
ubuntu@ubuntu:~/skj$ cd folder1
ubuntu@ubuntu:~/skj/folder1$ cat > f1
Hai
ubuntu@ubuntu:~/skj/folder1$ cat > f2
Hello
ubuntu@ubuntu:~/skj$ ./a.out
ENTER DIRECTORY NAME folder1
.
..
f2
f1

3.b) PROGRAM FOR SYSTEM CALLS OF UNIX OPERATING SYSTEM (fork, getpid, exit)

ALGORITHM:
1) Start the program.
2) Declare three variables pid, pid1, pid2.
3) Create process using fork() system call.
4) If pid==-1, exit.
5) If pid!=-1 , get the process id using getpid() system call.
6) Display the process id of parent and child process.
7) Stop the program

PROGRAM:

```
/* geditfork.c */
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
int main()
{
int pid,pid1,pid2;
pid=fork();
if(pid==-1)
{
printf("ERROR IN PROCESS CREATION \n");
exit(1);
}
if(pid!=0)
{
pid1=getpid();
printf("\n the parent process ID is %d\n", pid1);
}
else
{
pid2=getpid();
printf("\n the child process ID is %d\n", pid2);
}
return 0;
}
```

OUTPUT:

ubuntu@ubuntu:~/skj$ gedit dircommands2.c
ubuntu@ubuntu:~/skj$ gcc dircommands2.c
ubuntu@ubuntu:~/skj$ ./a.out
the parent process ID is 11306
the child process ID is 11307

3.c) PROGRAM FOR SYSTEM CALLS OF UNIX OPERATING SYSTEMS (wait, exit, close)

AIM :

To Execute a Unix Command in a 'C' program using wait() system call.

ALGORITHM :

1) Start the program
2) Initialize variables such as pid, status,
3) Use wait() to return the parent id of the child else return -1 for an error
4) Stop the program.

PROGRAM:

```c
/* wait.c */
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
int main()
{
int pid,status,exitch;
if((pid=fork())==-1)
{
perror("error");
exit (0);
}
if(pid==0)
{
sleep(1);
printf("child process");
exit (0);
}
else
{
printf("parent process\n");
if((exitch=wait(&status))==-1)
{
perror("during wait()");
exit (0);
}
printf("\nparent existing\n");
exit (0);
}
return 0;
}
```

OUTPUT:

buntu@ubuntu:~/OSLab$ gccwait.c
ubuntu@ubuntu:~/OSLab$ ./a.out
parent process
child process
parent existing
ubuntu@ubuntu:~/OSLab$

3.d) PROGRAM FOR SYSTEM CALLS OF UNIX OPERATING SYSTEMS (PROGRAM
to get information about the file using **stat** system call.)

ALGORITHM:

1) Start the program
2) Declare structure variable struct stat
3) Allocate the size for the file by using malloc function
4) Read the filename whose status has to be displayed
5) Stop the program

PROGRAM:

```
/* stat.c */
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
#include<stdlib.h>
int main()
{
char *path,path1[10];
struct stat *nfile;
nfile=(struct stat *) malloc (sizeof(struct stat));
printf("Enter name of the file forgetting its statistics: ");
scanf("%s",path1);
stat(path1,nfile);
printf("user id %d\n",nfile->st_uid);
printf("block size : %ld\n",nfile->st_blksize);
printf("last access time: %ld\n",nfile->st_atime);
printf("time of last modification: %ld\n",nfile->st_atime);
printf("production mode: %o \n",nfile->st_mode);
printf("size of file: %ld\n",nfile->st_size);
printf("number of links: %ld\n",nfile->st_nlink);
return 0;
}
```

OUTPUT:

```
ubuntu@ubuntu:~/OSLab$ gccstat.c
ubuntu@ubuntu:~/OSLab$ ./a.out
Enter name of the file for getting  itsstatistics: stat.c
user id 999
block size : 4096
last access time: 1660724493
time of last modification: 1660724493
production mode: 100664
size of file: 686
number of links: 1
```

3.e) PROGRAM FOR SYSTEM CALLS OF UNIX OPERATING SYSTEMS (PROGRAM to get information about the file using **exec**system call.)

AIM:

To Execute a Unix Command in a 'C' program using exec() system call.

ALGORITHM:

1) Start the program.
2) Declare the variables, pid.
3) Call the functionexecv (filename,argv) to transform an executable binary file into process.
4) Repeat this process until all the executed files are displayed.
5) Stop the program.

PROGRAM:

```
// exec() System call
#include<stdio.h>
int main()
{
int pid;

// A null terminated array of character pointers

char *args[]={"/bin/ls","-l",0};
printf("\nParent process");
pid=fork();
if(pid==0)
{
execv("/bin/ls",args);
}
else
{
printf("\nChild process");
}
return 0;
}
```

OUTPUT:

total 440

-rwxrwxr-x 1 skec25 skec25 5210 Apr 16 06:25 a.out

-rw-rw-r-- 1 skec25 skec25 775 Apr 9 08:36 bestfit.c

-rw-rw-r-- 1 skec25 skec25 1669 Apr 10 09:19 correctpipe.c

-rw-rw-r-- 1 skec25 skec25 977 Apr 16 06:15 correctprio.c

-rw------- 1 skec25 skec25 13 Apr 10 08:14 datafile.dat

-rw------- 1 skec25 skec25 13 Apr 10 08:15 example.dat

-rw-rw-r-- 1 skec25 skec25 166 Apr 16 06:25 exec.c

-rw-rw-r-- 1 skec25 skec25 490 Apr 10 09:43 exit.c
Parent Process

RESULT:

The programs are compiled, executed and the output is verified.

EXPERIMENT - 4
I/O system calls of Linux operating system (open, read, write)

AIM:

To implement UNIX / Linux I/O system calls open, read, write etc.

THEORY

In UNIX operating system,all devices are represented as files. These files are located in the directory /dev. All the devices and other files are accessed in a similar way. Devices file which is specified as 'blockspecial file' with some similar characters of a disk file. A device which is specified as a 'character special file' with some characteristics that are similar to a keyboard.

UNIX system calls for I/O:

1) creat(filename, mode) – This system call is used to create a new file with the filename and specifiedmode of permission. Permission mode canalso be a represented as a number. 0666 means to grant read write permissions.
2) open(filename, mode) – This system call is used to open a file name in the specified mode (read or write). Permission mode canalso be a represented as a number. 0 is for opening in read mode, 1 for writing and 2 for both read and write mode.
3) close(fd) – This system call is used to close an opened file.
4) unlink(fd) – This system call is used to delete a file.
5) read(fd, buffer, n_to_read) – This system call is used to read data from a file specified by fd.
6) write(fd, buffer, n_to_write) - This system call is used to write data from to a file specified by fd.
7) lseek(fd, offset, whence) - This system call is used to move the read/write pointer to the specified location.

ALGORITHM:

1) Start the program.
2) Read a message.
3) Create a new file using creat command /open command
4) If fd = -1
5) Print the factorial of the given number.
6) Stop the program.

PROGRAM:

```
/* io.c */
#include<stdio.h>
#include<stdlib.h>
#include <unistd.h>
#include<fcntl.h> //file control
#include<sys/types.h>
#include<sys/stat.h>
```

```
static char message[]="hai Hello world";
int main()
{
int fd;
char buffer[80];
fd=open("IOSytemCall_file.txt",O_RDWR|O_CREAT|O_EXCL,S_IREAD|S_IWRITE);
if(fd!=-1)
{
printf("IOSytemCall_file.txt opened for read/write access\n");
write(fd,message,sizeof(message));
lseek(fd,0l,0);
if(read(fd,buffer,sizeof(message))==sizeof(message))
        printf("\"%s\" was written to IOSytemCall_file.txt\n",buffer);
else
        printf("\t Error reading IOSytemCall_file.txt\n");
}
else
exit(0);
}
```

OUTPUT:

ubuntu@ubuntu:~/OSLab$ geditio.c
ubuntu@ubuntu:~/OSLab$ gccio.c
ubuntu@ubuntu:~/OSLab$ ./a.out
IOSytemCall_file.txt opened for read/write access
"hai Hello world" was written to IOSytemCall_file.txt
ubuntu@ubuntu:~/OSLab$

RESULT:

The program is compiled, executed and the output is verified.

EXPERIMENT - 5
Inter Process Communication using Shared Memory

AIM:

C program Writer, attaches itself to the shared memory segment created in Reader Process and it reads the content of the shared memory.

ALGORITHM

1) Start the program
2) Declare the variables, shmid
3) shmat() and shmdt() are used to attach and detach shared memory segments.
   void *shmat(int shmid, const void *shmaddr, int shmflg);
   int shmdt(const void *shmaddr);
4) shmat() returns a pointer, shmaddr, to the head of the shared segment associated with a validshmid.
5) shmdt() detaches the shared memory segment located at the address indicated byshmaddr
6) SharedMemory_Writer.c creates the string and shared memory portion.
7) SharedMemory_Reader.c attaches itself to the created shared memory portion and uses the string (printf)
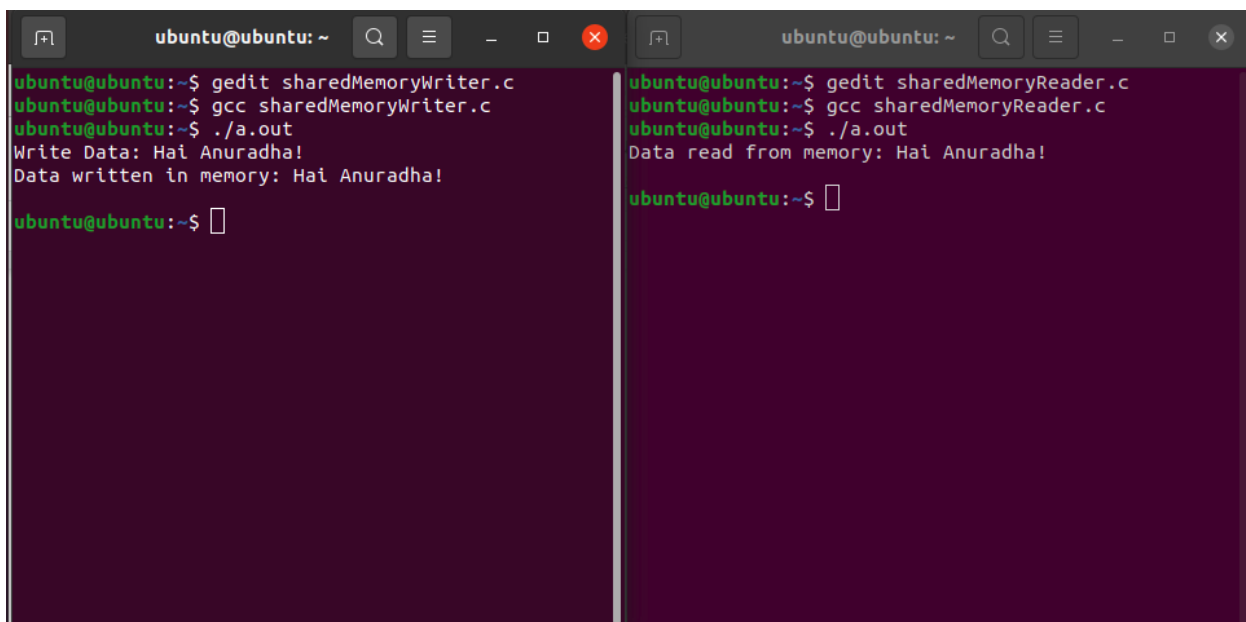8) Stop the program.

PROGRAM:

/* sharedMemoryWriter.c */
```c
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
int main()
{
/*ftok function is used to generate unique key which is for System V IPC functions */
key_t key = ftok("shmfile",65);
/*shmget functionreturns the shared memory identifier associated with key in shmid */
int shmid = shmget(key,1024,0666|IPC_CREAT);
/*shmat function is used to attach to the shared memorysegment associated with the shared memory
identifier, shmid, to the address space of the calling process. */
char *str = (char*) shmat(shmid,(void*)0,0);
printf("Write Data: ");
//scanf("%s", str);
//gets(str);
fgets(str, 25, stdin);
printf("\n Data written in memory: %s\n",str);
/* shmdtfunction detaches the shared memory segment located at the specified address from the address
space of the calling process */
shmdt(str);
```

```
return 0;
}
```

/* sharedMemoryReader.c */
```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int main()
{
/*ftok to generate unique key */
key_t key = ftok("shmfile",65);
// shmget returns an identifier in shmid
int shmid = shmget(key,1024,0666|IPC_CREAT);
// shmat to attach to shared memory
char *str = (char*) shmat(shmid,(void*)0,0);
printf("Data read from memory: %s\n",str);
//detach from shared memory
shmdt(str);
// destroy the shared memory
shmctl(shmid,IPC_RMID,NULL);
return 0;
}
```

OUTPUT:

RESULT:

The program is compiled, executed and the output is verified.

Date:

EXPERIMENT - 6
Implement Semaphores

AIM:

To write a C-program to implement the producer – consumer problem using semaphores.

THEORY:

A **semaphore** is an integer variable s that, apart from initialization, is accessed only through two standard atomic operations:

1) wait(s) - blocks while s <= 0, then decrements s; and
2) signal(s) - increments s.

A semaphore is used to limit access to a resource to a specific number of threads / processes. The initial value of the semaphore denotes the limit. A popular analogy is that a semaphore is like abouncer at a bar / club. As people enter the bar, the number of people allowed to enterdecreases. Once that number decreases to zero, people must wait to enter until other peopleleave. If the semaphore's initial value is 1, then this is known as a binary semaphore, which effectively serves the same role as a mutex.

DESCRIPTION

The producer-consumer problem illustrates the need for synchronization in systems where many processes share a resource. In the problem, two processes share a fixed-size buffer. One process produces information and puts it in the buffer, while the other process consumes information from the buffer. These processes do not take turns accessing the buffer, they both work concurrently.

Three semaphore variables are used:

- **mutex** - The lock is acquired and released using a mutex, a binary semaphore.
- **empty** - empty is a counting semaphore that is initialized on the basis of the number of slots present in the buffer, at first all the slots are empty.
- **full** - a counting semaphore with a value of zero as its starting value.

Precisely, the current value of empty denotes the number of vacant slots in the buffer, while full denotes the number of occupied slots.

ALGORITHM:

1) Let the size of the buffer be a counting semaphore, empty = 3.
2) Declare the semaphore objects mutex for mutual exclusion, empty for the number of empty positions in the buffer, and full for the number of full positions in the buffer.
3) Initialize the semaphore objects.
4) Create two functions - the function producer() and the function consumer(). Two functions producer and consumer synchronize with each other.
5) Producer()
   5.1) mutex=wait(mutex);
   5.2) full=signal(full);
   5.3) empty=wait(empty);
   5.4) Increment x as x++;
   5.5) Display the available produced item count, x.
   5.6) mutex=signal(mutex);
6) Consumer()

      6.1) mutex=wait(mutex);
      6.2) full=wait(full);
      6.3) empty=signal(empty);
      6.4) Display the consumed items, x.
      6.5) Decrement x as x--;
      6.6) mutex=signal(mutex);
  7) Destroy the semaphore objects.

PROGRAM:

```c
#include<stdio.h>
int mutex=1, full=0, empty=3, x=0;

int main()
{
int n;
void producer();
void consumer();
int wait(int);

int signal(int);
printf("\n1.PRODUCER\n2.CONSUMER\n3.EXIT\n");
while(1)
{
printf("\n Enter the choice \n");
scanf("%d",&ch);
switch(ch)
{
        case 1:
                if((mutex==1)&&(empty!=0))
                producer();
                else
                printf("BUFFER IS FULL");
        break;
        case 2:
                if((mutex==1)&&(full!=0))
                consumer();
                else
                printf("BUFFER IS EMPTY");
        break;
        case 3:
                exit(0);
        break;
}
}
return 0;
}

int wait(int s)
{
return(--s);
```

```
}
int signal(int s)
{
return(++s);
}

void producer()
{
mutex=wait(mutex);
full=signal(full);
empty=wait(empty);
x++;
printf("\nProducer produces an item%d",x);
mutex=signal(mutex);
}

void consumer()
{
mutex=wait(mutex);
full=wait(full);
empty=signal(empty);
printf("\n Consumer consumes an  item%d",x);
x--;
mutex=signal(mutex);
}
```

OUTPUT:
1.PRODUCER
2.CONSUMER
3.EXIT

Enter the choice:
1
Producer produces an item1
Enter the choice:
1
Producer produces an item2
Enter the choice:
1
Producer produces an item3
Enter the choice:
1
BUFFER IS FULL
Enter the choice:
2
Consumer consumes an item3
Enter the choice:
2
Consumer consumes an item2

Enter the choice:

2
Consumer consumes an item1
Enter the choice:
2
BUFFER IS EMPTY
Enter the choice:
2
BUFFER IS EMPTY
Enter the choice:
1
Producer produces an item1
Enter the choice:
3

Result:
Thus, the program to implement producer consumer problem has been executed successfully.

EXPERIMENT - 7
CPU scheduling algorithms

AIM:

Write a program to simulate the following non-preemptive CPU scheduling algorithms to find turnaround time and waiting time.

a) FCFS   b) SJF   c) Round Robin (pre-emptive)   d) Priority

DESCRIPTION

An Operating System is a system software that manages the computer hardware. It controls and coordinates usage of the hardware devicesand manages the various application programs.

A program under execution is called as a process. When a process executes, it changes state, usually into 5 states.

- New: The process is being created.
- Running: Instructions are being executed.
- Waiting: The process is waiting for some event to occur.
- Ready: The process is waiting to be assigned to a process.
- Terminated: The process has finished its execution.

Apart from the program code, a process includes the current activity represented in as follows:

- PC or Program Counter,
- Contents of Processor registers,
- Process Stack which contains temporary data like function parameters, return addresses and local variables
- Data section which contains global variables
- Heap for dynamic memory allocation

A Multi-programmed system will have manysimultaneously runningprocesses. Switching the CPU between these processes by the operating system can make the computer more productive. There is a Process Scheduler which selects the process among many processes that are ready for program execution on the CPU. Switching the CPU to another process must save the state of the current process and restore the state of the new process, this is called Context Switch.

Scheduling Algorithms

CPU Scheduler is responsible for selecting a process from the ready queue. There are various scheduling algorithms.Different scheduling algorithms have different properties, and the choice of a particular algorithm may favor one class of processes over another.

The scheduling criteria include

- CPU utilization: It refers to the amount of work handled by a CPU. It denotes the usage of processing resource of the computer.
- Throughput: The total number of processes completed per unit time.
- Waiting time: The sum of time spent waiting in the ready queue.
- Response time: The time from submission of a request until the first response is produced.

- Turnaround time: The time taken to complete a process. It is the total time taken from the time ofsubmission of a process to the time of completion of the process.

The different scheduling algorithms are

1) **FCFS: First Come First Serve Scheduling**
2) **SJF: Shortest Job First Scheduling**
3) **SRTF: Shortest Remaining Time First Scheduling**
4) **Round Robin Scheduling**
5) **Priority Scheduling**

A Process Scheduler is responsible for scheduling different processesto the CPU based on particular scheduling algorithms. These scheduling algorithms are either **non-preemptive** or **preemptive**.
In non-preemptive algorithms, once a process gets the CPU and enters the running state, it cannot be preempted until it completes its CPU burst time.
Preemptive scheduling is based on priority andthe scheduler may preempt a running low priority process anytime when a high priority process enters into the ready state.

First Come First Serve (FCFS)
- Jobs are executed on a first come, first serve basis.
- It is a non-preemptive scheduling algorithm.
- Easy to understand and implement.
- Its implementation is based on the FIFO queue.
- Poor in performance as average wait time is high.

DESCRIPTION:

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1, P_2, P_3
  The Gantt Chart for the schedule is:

P_1			P_2	P_3
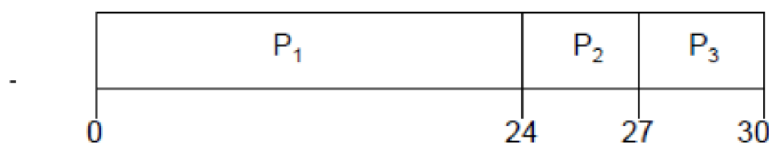
0   24   27   30

- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

Shortest Job First (SJF)
- This is also known as shortest job first, or SJF

- This is a non-preemptiveor preemptive algorithm.
- This is the 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 processor should know in advance how much time the process will take.

DESCRIPTION:
All processes are sorted according to the its CPU burst time. Theprocess with the shortest time will be scheduled next.
There are two schemes of SJF:
- **Non preemptive SJF**
  Once CPU given to the process it cannot be preempted until completes its CPU burst time.
- **Preemptive SJF or SRTF**
  If a new process arrives with less CPU burst time than the remaining time of currentexecuting process, then preempt the execution of current process. Then the shorter process with less burst time is allotted to CPU. This scheme is referred as the Shortest-Remaining-Time-First (SRTF)

SRTF: Shortest Remaining Time First Scheduling
- It is the pre-emptive form of SJF. In SRTF, the OS schedules the Job according to the remainingtime of the execution.

Priority Based Scheduling
- This is a non-preemptive algorithm.It is the most commonly used CPU scheduling algorithm in batch systems.
- Each process is assigned with a priority.
- Process with the highest priority is to be executed first and so on.
- Processes with the same priority are executed on a first come first served basis (FCFS).
- Priority can be decided based on the requirement of memory, CPU time or any other resource.

DESCRIPTION:
A priority number (integer) is associated with each process.
The process with the highest priority gets the CPU.The smallest integer means highest priority. Priority 1, 2, 3 etc.
Two schemes of priority-based scheduling:
- **Priority preemptive Scheduling** - If a new process arrives with higher priority arrives than the currentexecuting process, then it preempts the execution of current process and the higher priority process is allotted to CPU.
- **Priority non preemptive Scheduling** - Once CPU and resources are allocated to a process, the process holds it until it completes its burst time even if a higher priority process is arrived to the queue.

SJF is a priority-basedscheduling in which the priority is the CPU burst time.
***Problem*:Starvation** – The problem is low priority processes may never execute and keeps on waiting for the CPU allocation since always the high priority process keeps on executing.

*Solution:***Aging** –It is the scheduling technique that avoids starvation. After some fixed amount of time, increase the priority of eachlower priority process.

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

DESCRIPTION:
Each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue. If there are *n* processes in the ready queue and the time quantum is *q*, then each process gets $1/n$ of the CPU time in chunks of at most *q* time units at once. No process waits more than $(n-1)q$ time units.
Consider an example for round robin with quantum time=20

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

The Gantt chart is:

P_1	P_2	P_3	P_4	P_1	P_3	P_4	P_1	P_3	P_3

0   20   37   57   77   97   117   121   134   154   162
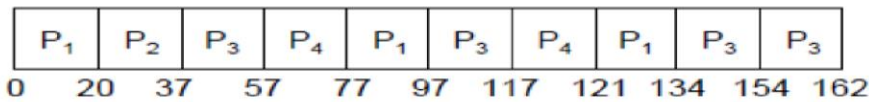
Typically, higher average turnaround than SJF, but better response

PROGRAM

7.a)FCFS

```c
/* fcfs.c*/
#include<stdio.h>
int main()
{
int burstTime[20], waitingTime[20], turnaroundTime[20], i, n;
float avgwt, avgtat;
printf("\nEnter the number of processes -- ");
scanf("%d", &n);
for(i=0;i<n;i++)
{
printf("\nEnter Burst Time for Process %d -- ", i);
scanf("%d", &burstTime[i]);
```

```
}
waitingTime[0] =  avgwt = 0;
turnaroundTime[0] = avgtat = burstTime[0];
for(i=1;i<n;i++)
{
waitingTime[i] = waitingTime[i-1] + burstTime[i-1];
turnaroundTime[i] = turnaroundTime[i-1] + burstTime[i];
avgwt = avgwt + waitingTime[i];
avgtat = avgtat + turnaroundTime[i];
}
printf("\t PROCESS \tBURST TIME \t WAITING TIME\t TURNAROUND TIME\n");
for(i=0;i<n;i++)
            printf("\n\t P%d \t\t %d \t\t %d \t\t %d", i, burstTime[i], waitingTime[i], turnaroundTime[i]);
avgwt  =avgwt/n;
avgtat = avgtat/n;


printf("\nAverage Waiting Time - %f", avgwt);
printf("\nAverage Turnaround Time - %f", avgtat );
return 0;
}
```

TEST CASE

sa@sa-ThinkCentre-M72e:~$ cd Desktop/

sa@sa-ThinkCentre-M72e:~/Desktop$ gccfcfs.c

sa@sa-ThinkCentre-M72e:~/Desktop$. /a.out

Enter the number of processes -- 3

Enter Burst Time for Process 0 -- 24

Enter Burst Time for Process 1 -- 3

Enter Burst Time for Process 2 – 3

OUTPUT

PROCESS	BURST TIME	WAITING TIME	TURNAROUND TIME
P0	24	0	24
P1	3	24	27
P2	3	27	30

Average Waiting Time - 17.000000

Average Turnaround Time - 27.000000

7.b) SJF

```c
/* sjf.c */

#include<stdio.h>
int main()
{
int p[20], bt[20], wt[20], tat[20], i, k, n, temp;
float wtavg, tatavg;
printf("\nEnter the number of processes -- ");
scanf("%d", &n);
for(i=0;i<n;i++)
{
p[i]=i;
printf("Enter Burst Time for Process %d -- ", i);
scanf("%d", &bt[i]);
}
for(i=0;i<n;i++)
for(k=i+1;k<n;k++)
if(bt[i]>bt[k])
{
temp=bt[i];
bt[i]=bt[k];
bt[k]=temp;
temp=p[i];
p[i]=p[k];
p[k]=temp;
}
wt[0] =  wtavg = 0;
tat[0] = tatavg = bt[0];
for(i=1;i<n;i++)
{
wt[i] = wt[i-1] +bt[i-1];
tat[i] = tat[i-1] +bt[i];
wtavg = wtavg + wt[i];
tatavg = tatavg + tat[i];
}
printf("\n\t PROCESS \tBURST TIME \t WAITING TIME\t TURNAROUND TIME\n");
for(i=0;i<n;i++)
printf("\n\t P%d \t\t %d \t\t %d \t\t %d", p[i], bt[i], wt[i], tat[i]);
printf("\nAverage Waiting Time -- %f", wtavg/n);
printf("\nAverage Turnaround Time -- %f\n", tatavg/n);
return 0;
}
```

TEST CASE

sa@sa-ThinkCentre-E73:~/skj/OSLab$ geditsjf.c

sa@sa-ThinkCentre-E73:~/skj/OSLab$ gccsjf.c

sa@sa-ThinkCentre-E73:~/skj/OSLab$ ./a.out

Enter the number of processes -- 3

Enter Burst Time for Process 0 -- 5

Enter Burst Time for Process 1 -- 3

Enter Burst Time for Process 2 -- 9

OUTPUT

PROCESS	BURST TIME	WAITING TIME	TURNAROUND TIME
P1	3	0	3
P0	5	3	8
P2	9	8	17

Average Waiting Time -- 3.666667

Average Turnaround Time -- 9.333333

sa@sa-ThinkCentre-E73:~/skj/OSLab$

7.c) ROUND ROBIN

```
#include<stdio.h>
int main()
{
int count,j,n,time,remain,flag=0,time_quantum;
int wait_time=0,turnaround_time=0,at[10],bt[10],rt[10];
printf("Enter Total Process:\t ");
scanf("%d",&n);
remain=n;
for(count=0;count<n;count++)
{
printf("Enter Arrival Time and Burst Time for Process Process Number %d :",count+1);
scanf("%d",&at[count]);
scanf("%d",&bt[count]);
rt[count]=bt[count];
}

printf("Enter Time Quantum:\t");
scanf("%d",&time_quantum);
```

```
printf("\n\nProcess\t|TurnaroundTime|Waiting Time\n\n");
for(time=0,count=0;remain!=0;)
{
if(rt[count]<=time_quantum&& rt[count]>0)
{
time+=rt[count];
rt[count]=0;
flag=1;
}
else if(rt[count]>0)
{
rt[count]-=time_quantum;
time+=time_quantum;
}
if(rt[count]==0 && flag==1)
{
remain--;
printf("P[%d]\t|\t%d\t|\t%d\n",count+1,time-at[count],time-at[count]-bt[count]);
wait_time+=time-at[count]-bt[count];
turnaround_time+=time-at[count];
flag=0;
}
if(count==n-1)
count=0;
else if(at[count+1]<=time)
count++;
else
count=0;
}
printf("\nAverage Waiting Time= %f\n",wait_time*1.0/n);
printf("Avg Turnaround Time = %f",turnaround_time*1.0/n);
return 0;
}
```

TEST CASE
sa@sa-ThinkCentre-M72e:~/Desktop$ ./a.out
Enter Total Process: 4
Enter Arrival Time and Burst Time for Process Process Number 1 :01
Enter Arrival Time and Burst Time for Process Process Number 2 :0 4
Enter Arrival Time and Burst Time for Process Process Number 3 :0 6
Enter Arrival Time and Burst Time for Process Process Number 4 :0 4
Enter Time Quantum: 4

OUTPUT
Process |Turnaround Time|Waiting Time
P[1] | 1 | 0
P[2] | 5 | 1
P[4] | 13 | 9
P[3] | 15 | 9
Average Waiting Time= 4.750000
Avg Turnaround Time = 8.500000

TEST CASE 2
sa@sa-ThinkCentre-M72e:~/Desktop$ ./a.out
Enter Total Process: 4
Enter Arrival Time and Burst Time for Process Process Number 1 :0
9
Enter Arrival Time and Burst Time for Process Process Number 2 :1
5
Enter Arrival Time and Burst Time for Process Process Number 3 :2
3
Enter Arrival Time and Burst Time for Process Process Number 4 :3
4
Enter Time Quantum: 5

OUTPUT
Process |Turnaround Time|Waiting Time
P[2] | 9 | 4
P[3] | 11 | 8
P[4] | 14 | 10
P[1] | 21 | 12
Average Waiting Time= 8.500000
Avg Turnaround Time = 13.750000

7.d) PRIORITY SCHEDULING

```c
#include<stdio.h>
int main()
{
int bt[20],p[20],wt[20],tat[20],pr[20],i,j,n,total=0,pos,temp,avg_wt,avg_tat;
printf("Enter Total Number of Process:");
scanf("%d",&n);
printf("\nEnter Burst Time and Priority\n");
for(i=0;i<n;i++)
{
printf("\nP[%d]\n",i+1);
printf("Burst Time:");
```

```c
scanf("%d",&bt[i]);
printf("Priority:");
scanf("%d",&pr[i]);
p[i]=i+1;          //contains process number
}
//sorting burst time, priority and process number in ascending order using selection sort
for(i=0;i<n;i++)
{
pos=i;
for(j=i+1;j<n;j++)
{
if(pr[j]<pr[pos])
pos=j;
}
temp=pr[i];
pr[i]=pr[pos];
pr[pos]=temp;
temp=bt[i];
bt[i]=bt[pos];
bt[pos]=temp;
temp=p[i];
p[i]=p[pos];
p[pos]=temp;
}
wt[0]=0;    //waiting time for first process is zero
//calculate waiting time
for(i=1;i<n;i++)
{
wt[i]=0;
for(j=0;j<i;j++)
wt[i]+=bt[j];
total+=wt[i];
}
avg_wt=total/n;      //average waiting time
total=0;
printf("\nProcess\t   Burst Time   \tWaiting Time\tTurnaround Time");
for(i=0;i<n;i++)
{
tat[i]=bt[i]+wt[i];     //calculate turnaround time
total+=tat[i];
printf("\nP[%d]\t\t  %d\t\t   %d\t\t\t%d",p[i],bt[i],wt[i],tat[i]);
}
avg_tat=total/n;    //average turnaround time
printf("\n\nAverage Waiting Time=%d",avg_wt);
```

```
printf("\nAverage Turnaround Time=%d\n",avg_tat);
return 0;
}
```

TEST CASE

Enter Total Number of Process:4

Enter Burst Time and Priority

P[1]

Burst Time:4

Priority:1

P[2]

Burst Time:5

Priority:2

P[3]

Burst Time:5

Priority:4

P[4]

Burst Time:7

Priority:3

OUTPUT

Process	Burst Time	Waiting Time	Turnaround Time
P[1]	4	0	4
P[2]	5	4	9
P[4]	7	9	16
P[3]	5	16	21

Average Waiting Time=7

Average Turnaround Time=12

RESULT:

Thus, the program to implement various CPU scheduling algorithms has been executed successfully.

<div align="center">

EXPERIMENT - 8
Implementation of the Memory Allocation Methods for fixed partition
a) First Fit b) Worst Fit c) Best Fit

</div>

AIM:

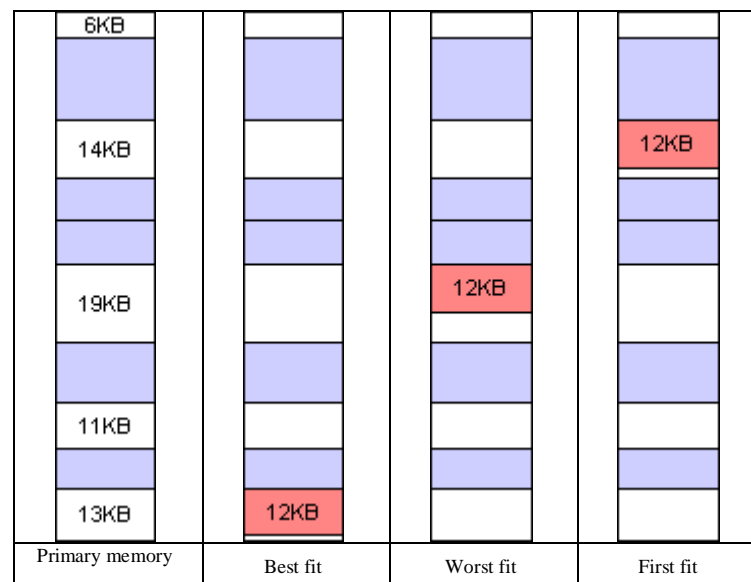Implementation of the Memory Allocation Methods for fixed partition*
a) First Fit b) Worst Fit c) Best Fit

THEORY:

Memory Management Algorithms

In an environment that supports dynamic memory allocation, the memory manager must keep a
record of the usage of each allocatable block of memory. This record could be kept by using almost any
data structure that implements linked lists. An obvious implementation is to define a free list of block
descriptors, with each descriptor containing a pointer to the next descriptor, a pointer to the block, and
the length of the block. The memory manager keeps a free list pointer and inserts entries into the list in
some order conducive to its allocation strategy. A number of strategies are used to allocate space to the
processes that are competing for memory.

- **Best fit:** The allocator places a process in the smallest block of unallocated memory in which it
  will fit. For example, suppose a process requests 12KB of memory and the memory manager
  currently has a list of unallocated blocks of 6KB, 14KB, 19KB, 11KB, and 13KB blocks. The
  best-fit strategy will allocate 12KB of the 13KB block to the process.
- **Worst fit:** The memory manager places a process in the largest block of unallocated memory
  available. The idea is that this placement will create the largest hold after the allocations, thus
  increasing the possibility that, compared to best fit, another process can use the remaining space.
  Using the same example as above, worst fit will allocate 12KB of the 19KB block to the
  process, leaving a 7KB block for future use.
- **First fit:** There may be many holes in the memory, so the operating system, to reduce the
  amount of time it spends analyzing the available spaces, begins at the start of primary memory
  and allocates memory from the first hole it encounters large enough to satisfy the request. Using
  the same example as above, first fit will allocate 12KB of the 14KB block to the process**.**

8. a) First fit

AIM:
To write a C program for the implementation of memory allocation methods for fixed partitions using the first fit.

ALGORITHM:
1) Define the max as 25.
2) Declare the variable frag[max],b[max],f[max],i,j,nb,nf,temp, highest=0, bf[max],ff[max].
3) Get the number of blocks,files,size of the blocks using for loop.
4) In for loop check bf[j]!=1, if so temp=b[j]-f[i]
5) Check highest<temp,if so assign ff[i]=j,highest=temp
6) Assign frag[i]=highest, bf[ff[i]]=1,highest=0
7) Repeat step 4 to step 6.
8) Print file no,size,blockno,size and fragment.
9) Stop the program.

PROGRAM:
```
/* first fit */
#include<stdio.h>
#define max 25
int main()
{
int frag[max],b[max],f[max],i,j,nb,nf,temp,highest=0;
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++)
{
```

```c
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 \n",i,f[i],ff[i],b[ff[i]],frag[i]);
return 0;
}
```

OUTPUT:
ubuntu@ubuntu: ~/OSLab$ **geditbestfit.c**
ubuntu@ubuntu: ~/OSLab$ **gccfirstfit.c**
ubuntu@ubuntu: ~/OSLab$ **./a.out**

Memory Management Scheme - First Fit
Enter the number of blocks:3
Enter the number of files:3
Enter the size of the blocks:-
Block 1:9
Block 2:5
Block 3:8
Enter the size of the files :-
File 1:5
File 2:7
File 3:3

File_no:	File_size :	Block_no:	Block_size:	Fragement
1	5	1	9	4
2	7	3	8	1
3	3	2	5	2

8. b) Worst Fit - Memory Allocation Methods for fixed partition

AIM:
To write a C program for implementing memory allocation methods for fixed partition using the worst fit.

ALGORITHM:
1) Define the max as 25.
2) Declare the variable frag[max],b[max],f[max],i,j,nb,nf,temp, highest=0, bf[max],ff[max].
3) Get the number of blocks,files,size of the blocks using for loop.
4) In for loop check bf[j]!=1, if so temp=b[j]-f[i]
5) Check highest<temp,if so assign ff[i]=j,highest=temp
6) Assign frag[i]=highest, bf[ff[i]]=1,highest=0
7) Repeat step 4 to step 6.
8) Print file no,size,blockno,size and fragment.
9) Stop the program.

PROGRAM:
```
/* worst fit */
#include<stdio.h>
#define max 25
{ int frag[max],b[max],f[max],i,j,nb,nf,temp;
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)
{
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 \n",i,f[i],ff[i],b[ff[i]],frag[i]);
return 0;
}
```

OUTPUT:
ubuntu@ubuntu: ~/OSLab$ **geditbestfit.c**
ubuntu@ubuntu: ~/OSLab$ **gccworstfit.c**
ubuntu@ubuntu: ~/OSLab$ **./a.out**

Memory Management Scheme - Worst Fit
Enter the number of blocks:3
Enter the number of files:3
Enter the size of the blocks:-
Block 1:9
Block 2:5
Block 3:8
Enter the size of the files :-
File 1:5
File 2:7
File 3:3

File_no:	File_size :	Block_no:	Block_size:	Fragement
1	5	1	9	4
2	7	3	8	1
3	3	2	5	2

8. c) Best Fit - Memory Allocation Methods for fixed partition

AIM:

To write a program to implement the best fit algorithm for memory management.

ALGORITHM:
1) Define the max as 25.
2) Declare the variable frag[max],b[max],f[max],i,j,nb,nf,temp, highest=0, bf[max],ff[max].
3) Get the number of blocks,files,size of the blocks using for loop.
4) In for loop check bf[j]!=1, if so temp=b[j]-f[i]
5) Check lowest>temp,if so assign ff[i]=j,highest=temp
6) Assign frag[i]=lowest, bf[ff[i]]=1,lowest=10000
7) Repeat step 4 to step 6.
8) Print file no,size,block no,size and fragment.
9) Stop the program.

PROGRAM:
```c
/* best fit */
#include<stdio.h>
#define max 25
int main()
{
int frag[max], b[max], f[max], i, j, nb, nf, temp, lowest=10000;
int bf[max], ff[max];
printf("\n\tMemory Management Scheme - Best 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)
{
if(lowest>temp)
{
ff[i]=j;
lowest=temp;
}
}
else
printf ("No block suits the file size ");
}
}
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 \n",i,f[i],ff[i],b[ff[i]],frag[i]);
return 0;
}
```

OUTPUT:
ubuntu@ubuntu: ~/OSLab$ **geditbestfit.c**
ubuntu@ubuntu: ~/OSLab$ **gccbestfit.c**
ubuntu@ubuntu: ~/OSLab$ **./a.out**

Memory Management Scheme - Best Fit
Enter the number of blocks:3
Enter the number of files:3
Enter the size of the blocks:-
Block 1:9
Block 2:5
Block 3:8
Enter the size of the files :-
File 1:5
File 2:7
File 3:3

File No	File Size	Block No	Block Size	Fragment
1	5	2	5	0

2   7   3   8   1

3   3   1   9   6

RESULT:

Thus, the program to implement memory management schemes has been executed successfully.

EXPERIMENT - 9
Implement l page replacement algorithms a) FIFO b) LRU c) LFU

AIM:
Implementation of page replacement algorithms a) FIFO b) LRU c) LFU

THEORY:
Paging is a process of reading data from, and writing data to, the secondary storage. It is a memory management scheme that is used to retrieve processes from the secondary memory in the form of pages and store them in the primary memory. The main objective of paging is to divide each process in the form of pages of fixed size. These pages are stored in the main memory in frames. Pages of a process are only brought from the secondary memory to the main memory when they are needed.

When an executing process refers to a page, it is first searched in the main memory. If it is not present in the main memory, a page fault occurs.

- Page Fault is the condition in which a running process refers to a page that is not loaded in the main memory.

In such a case, the OS has to bring the page from the secondary storage into the main memory. This may cause some pages in the main memory to be replaced due to limited storage. A Page Replacement Algorithm is required to decide which page needs to be replaced.

Page Replacement Algorithm decides which page to remove, also called swap out when a new page needs to be loaded into the main memory. Page Replacement happens when a requested page is not present in the main memory and the available space is not sufficient for allocation to the requested page. When the page that was selected for replacement was paged out, and referenced again, it has to be read in from disk, and this requires for I/O completion. This process determines the quality of the page replacement algorithm: the lesser the time waiting for page-ins, the better is the algorithm.

A page replacement algorithm tries to select which pages should be replaced so as to minimize the total number of page misses. There are many different page replacement algorithms. These algorithms are evaluated by running them on a particular string of memory reference and computing the number of page faults. The fewer the page faults the better is the algorithm for that situation.

- If a process requests a page and that page is found in the main memory then it is called page hit, otherwise page miss or page fault.

Some Page Replacement Algorithms:
- First In First Out (FIFO)
- Least Recently Used (LRU)
- LFU
- Optimal Page Replacement

First In First Out (FIFO)

This is the simplest page replacement algorithm. In this algorithm, the OS maintains a queue that keeps track of all the pages in memory, with the oldest page at the front and the most recent page at the back. When there is a need for page replacement, the FIFO algorithm, swaps out the page at the front of the queue, that is the page which has been in the memory for the longest time. The page fault range increases as the number of allocated frames also increases.

Least Recently Used (LRU)

The LRU stands for the Least Recently Used. It keeps track of page usage in the memory over a short period of time. It works on the concept that pages that have been highly used in the past are likely to be significantly used again in the future. It removes the page that has not been utilized in the memory for the longest time. LRU is the most widely used algorithm because it provides fewer page faults than the other methods.

Least Frequently Used (LFU)

The LFU page replacement algorithm stands for the Least Frequently Used. In the LFU page replacement algorithm, the page with the least visits in a given period of time is removed. It replaces the least frequently used pages. If the frequency of pages remains constant, the page that comes first is replaced first.

9. a) Page replacement algorithms - FIFO

AIM:

To write a C program for implementation of FIFO page replacement algorithm.

ALGORITHM:

1) Start the program.
2) Declare the variables - f, n, count, pf and rs[], m[].
3) Enter the number of page references.
4) Enter the required page references.
5) Enter the number of frames.
6) FIFO page replacement selects the page that has been in memory the longest time and when the page must be replaced the oldest page is chosen.
7) When a page is brought into memory, it is inserted at the tail of the queue.
8) Initialize all theframes as -1 or empty.
9) Print the total number of page faults.
10) Stop the program.

PROGRAM:

/* FIFO Page Replacement */
```
#include<stdio.h>
int main()
{
```

```c
int i, j, k, f, pf=0, count=0, rs[25], m[10], n;

printf("\n Enter the number of page references : ");
scanf("%d",&n);
printf("\n Enter the page references : ");
for(i=0;i<n;i++)
scanf("%d",&rs[i]);
printf("\n Enter number of frames : ");
scanf("%d",&f);
for(i=0;i<f;i++)
        m[i]=-1;
printf("\n FIFO Page Replacement\n");
for(i=0;i<n;i++)
{
for(k=0;k<f;k++)
{
if(m[k]==rs[i])
break;
}
if(k==f)
{
m[count++]=rs[i];
pf++;
}
for(j=0;j<f;j++)
printf("\t%d",m[j]);
if(k==f)
printf("\tPF No. %d",pf);
printf("\n");
if(count==f)
count=0;
}
printf("\nTotal number of page faults using FIFO is %d \n",pf);
return 0;
}
```

Output:
ubuntu@ubuntu:~/OSLab$ geditfifoPageReplace.c
ubuntu@ubuntu:~/OSLab$ gccfifoPageReplace.c
ubuntu@ubuntu:~/OSLab$ ./a.out

Enter the number of page references : 5
Enter the page references : 2 1 5 1 3
Enter number of frames : 3

FIFO Page Replacement

2	-1	-1	PF No. 1
2	1	-1	PF No. 2
2	1	5	PF No. 3
2	1	5	
3	1	5	PF No. 4

Total number of page faults using FIFO is 4

9. b) LRU

PROGRAM:

/* lruPageReplace.c*/

```c
#include<stdio.h>
int main()
{
int i, j , k, min, rs[25], m[10], count[10], flag[25], n, f, pf=0, next=1;

printf("Enter the number of page references : ");
scanf("%d",&n);

printf("Enter the page references : ");
for(i=0;i<n;i++)
{
scanf("%d",&rs[i]);
flag[i]=0;
}
printf("Enter the number of frames : ");
scanf("%d",&f);

for(i=0;i<f;i++)
{
count[i]=0;
m[i]=-1;
}
printf("\n  LRU Page Replacement  \n");

for(i=0;i<n;i++)
{
for(j=0;j<f;j++)
{
if(m[j]==rs[i])
{
```

```
flag[i]=1;
count[j]=next;
next++;
}
}

if(flag[i]==0)
{
if(i<f)
{
m[i]=rs[i];
count[i]=next;
next++;
}
else
{
min=0;

for(j=1;j<f;j++)
if(count[min] > count[j])
min=j;

m[min]=rs[i];
count[min]=next;
next++;
}
pf++;

}
for(j=0;j<f;j++)
printf("%d\t", m[j]);
if(flag[i]==0)

printf("PF No. -- %d" , pf);

printf("\n");
}

printf("\nTotal number of page faults using LRU are %d \n",pf);
return 0;

}
```

OUTPUT:

ubuntu@ubuntu:~/OSLab$ geditlruPageReplace.c

ubuntu@ubuntu:~/OSLab$ gcclruPageReplace.c

ubuntu@ubuntu:~/OSLab$ ./a.out

Enter the number of page references : 7

Enter the page references : 2 1 2 1 5 3 5

Enter the number of frames : 3

LRU Page Replacement

2	-1	-1	PF No. -- 1
2	1	-1	PF No. -- 2
2	1	-1	
2	1	-1	
2	1	5	PF No. -- 3
3	1	5	PF No. -- 4
3	1	5	

Total number of page faults using LRU are 4

9. c) LFU Page Replacement

PROGRAM:

```
/* lfuPageReplace.c*/
#include<stdio.h>

int main()
{
int rs[50], i, j, k, m, f, cntr[20], a[20], min, pf=0;

printf("\nEnter number of page references : ");
scanf("%d",&m);

printf("\nEnter the page references : ");

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

printf("\nEnter the number of frames : ");
scanf("%d",&f);

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

```
a[i]=-1;
}

printf("\n LFU Page Replacement \n");
for(i=0;i<m;i++)
{
for(j=0;j<f;j++)
if(rs[i]==a[j])
{
cntr[j]++;
break;
}

if(j==f)
{
min = 0;
for(k=1;k<f;k++)

if(cntr[k]<cntr[min])

min=k;
a[min]=rs[i];

cntr[min]=1;

pf++;
}
printf("\n");

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

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

if(j==f)
printf("\tPF No. %d",pf);

}
printf("\n\n Total number of page faults are %d \n",pf);
return 0;
}
```

OUTPUT:
ubuntu@ubuntu:~/OSLab$ geditlfuPageReplace.c
ubuntu@ubuntu:~/OSLab$ gcclfuPageReplace.c

ubuntu@ubuntu:~/OSLab$ ./a.out

Enter number of page references : 7
Enter the page references : 2 1 2 1 5 3 5
Enter the number of frames : 3
LFU Page Replacement

2	-1	-1	PF No. 1
2	1	-1	PF No. 2
2	1	-1	PF No. 2
2	1	-1	PF No. 2
2	1	5	PF No. 3
2	1	3	PF No. 4
2	1	5	PF No. 5

 Total number of page faults are 5

RESULT:

Thus, the program to implement various page replacement algorithms has been executed successfully.

EXPERIMENT - 10

Implement the banker's algorithm for deadlock avoidance. *

AIM:
To write a C program to implement banker's algorithm for deadlock avoidance.

THEORY:
The banker's algorithm is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation for predetermined maximum possible amounts of all resources, then makes an "s-state" check to test for possible activities, before deciding whether allocation should be allowed to continue.

Banker's algorithm is termed so since it is used in the banking system to check whether a loan can be allowed to a person or not. Assume there are n number of account holders in a bank and the total sum of their money is S. If a person applies for a loan, then the bank first subtracts the loan amount from the total money that bank has and if the remaining amount is greater than S then only the loan is sanctioned. It is done because if all the account holders come to withdraw their money, then the bank can easily do it.

That is, the bank would never allocate its money in such a way that it can no longer satisfy the needs of all its customers. The bank would always try to be in a safe state.

Data structures used to implement the Banker's Algorithm are as follows:
Let 'n' be the number of processes in the system and 'm' be the number of resources types.

Available :
It is a one-dimensional array of size 'm' representing the number of available resources of each type.
Available[j] = k means there are 'k' instances of resource type R

Max :
It is a two-dimensional array of size 'n*m' that states the maximum request of each process in a system.
Max[i, j] = k means process Pi may request at most 'k' instances of resource type Rj.

Allocation :
It is a two-dimensionalarray of size 'n*m' that states the number of resources of each type currently assigned to each process.
Allocation[i, j] = k means process Pi is currently allocated 'k' instances of resource type Rj

Need :
It is a two-dimensional array of size 'n*m' that specifies the remaining resource need of each process.
Need [i, j] = k means process Pi currently need 'k' instances of resource type Rj
Need [i, j] = Max [i, j] – Allocation [i, j]

ALGORITHM:

1) Start the program.
2) Declare the memory for the process.
3) Read the number of processes, resources, allocation matrix and available matrix.
4) Compare each and every process using the banker"s algorithm.
5) If the process is in safe state then it is a not a deadlock process otherwise it is a deadlockprocess
6) Display the result of state of process
7) Stop the program

PROGRAM:

```c
/* bankers algorithm */
#include<stdio.h>
int max[100][100];
int alloc[100][100];
int need[100][100];
int avail[100];
int n,r;
void input();
void show();
void cal();
int main()
{
int i,j;
printf("\n Banker's Algorithm:");
input();
show();
cal();
return 0;
}
void input()
{
int i,j;
printf("Enter the no of Processes\t");
scanf("%d",&n);
printf("Enter the no of resources instances\t");
scanf("%d",&r);
printf("Enter the Max Matrix\n");
for(i=0;i<n;i++)
{
for(j=0;j<r;j++)
{
scanf("%d",&max[i][j]);
}
}
```

```c
printf("Enter the Allocation Matrix\n");
for(i=0;i<n;i++)
{
for(j=0;j<r;j++)
{
scanf("%d",&alloc[i][j]);
}
}
printf("Enter the available Resources\n");
for(j=0;j<r;j++)
{
scanf("%d",&avail[j]);
}
}
void show()
{
int i,j;
printf("Process \t Allocation \t Max \t Need Matrix\t Available ");
for(i=0;i<n;i++)
{
printf("\nP%d \t\t\t ",i+1);
for(j=0;j<r;j++)
{
printf("%d ",alloc[i][j]);
}
printf("\t\t\t");
for(j=0;j<r;j++)
{
printf("%d ",max[i][j]);
}
//find need matrix
for(j=0;j<r;j++)
{
need[i][j]=max[i][j]-alloc[i][j];
}

printf("\t\t");
for(j=0;j<r;j++)
{
printf("%d ",need[i][j]);
}
printf("\t\t\t");
if(i==0)
{
```

```
for(j=0;j<r;j++)
printf("%d ",avail[j]);
}
}
}
void cal()
{
int finish[100],temp,need[100][100],flag=1,k,c1=0;
int safe[100];
int i,j;
for(i=0;i<n;i++)
{
finish[i]=0;
}
//find need matrix
for(i=0;i<n;i++)
{
for(j=0;j<r;j++)
{
need[i][j]=max[i][j]-alloc[i][j];
}
}
printf("\n");
while(flag)
{
flag=0;
for(i=0;i<n;i++)
{
int c=0;
for(j=0;j<r;j++)
{
if((finish[i]==0)&&(need[i][j]<=avail[j]))
{
c++;
if(c==r)
{
for(k=0;k<r;k++)
{
avail[k]+=alloc[i][j];
finish[i]=1;
flag=1;
}
printf("P%d->",i+1);
if(finish[i]==1)
```

```
{
i=n;
}
}
}
}
}
}
for(i=0;i<n;i++)
{
if(finish[i]==1)
{
c1++;
}
else
{
printf("P%d->",i);
}
}
if(c1==n)
{
printf("\n The system is in safe state \n");
}
else
{
printf("\n Process are in dead lock");
printf("\n System is in unsafe state \n");
}
}
```

OUTPUT:
ubuntu@ubuntu:~/OSLab$ geditbankers.c
ubuntu@ubuntu:~/OSLab$ gccbankers.c
ubuntu@ubuntu:~/OSLab$ ./a.out

Banker's Algorithm:
Enter the no of Processes　　　　2
Enter the no of resources instances　　　3
Enter the Max Matrix
3 1 3
2 2 1
Enter the Allocation Matrix
1 1 2
1 2 1

Enter the available Resources
1 0 1

Process	Allocation	Max	Need Matrix	Available
P1	1 1 2	3 1 3	2 0 1	1 0 1
P2	1 2 1	2 2 1	1 0 0	

P2->P1->
 The system is in safe state

ubuntu@ubuntu:~/OSLab$ geditbankers.c
ubuntu@ubuntu:~/OSLab$ gccbankers.c
ubuntu@ubuntu:~/OSLab$ ./a.out

Banker's Algorithm :
Enter the no of Processes   5
Enter the no of resources instances   3
Enter the Max Matrix
7 5 3
3 2 2
9 0 2
2 2 2
4 3 3
Enter the Allocation Matrix
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2
Enter the available Resources
3 3 2

Process	Allocation	Max	Need Matrix	Available
P1	0 1 0	7 5 3	7 4 3	3 3 2
P2	2 0 0	3 2 2	1 2 2	
P3	3 0 2	9 0 2	6 0 0	
P4	2 1 1	2 2 2	0 1 1	
P5	0 0 2	4 3 3	4 3 1	

P2->P4->P5->P3->P1->
 The system is in safe state

RESULT:

Thus, the banker's algorithmfor deadlock avoidance has been executed successfully.

Date:

EXPERIMENT - 11

Simulate disk scheduling algorithms

a) FCFS b) SCAN c) C-SCAN

AIM

Simulate disk scheduling algorithms.

a) FCFS   b)SCAN   c) C-SCAN

THEORY:
The operating system schedules the incoming I/O requests for the disk access and this is known as the **disk scheduling.** Disk scheduling is also called as I/O scheduling.

Importance of disk scheduling:
- Multiple I/O requests may arrive from different processes butat a time only one I/O request can be served by the disk controller. Therefore, other I/O requests need to wait in the waiting queue and need to be scheduled.
- Two or more requests may be far from each other so can result in greater disk arm movement.
- Hard drives are one of the slowest parts of the computer system and thus need to be accessed in an efficient manner.

Disk scheduling algorithms are used to schedule multiple I/O requests for disk access.
The **purpose** of disk scheduling algorithms is to reduce the total seek time.

Important terms:
- **Seek Time:** Seek time is the time taken to locate the disk arm to a specified track where the data is to be read or write. So, the disk scheduling algorithm that gives minimum average seek time is better.
- **Rotational Latency**: Rotational Latency is the time taken by the desired sector of disk to rotate into a position so that it can access the read/write heads. So, the disk scheduling algorithm that gives minimum rotational latency is better.
- **Transfer Time:** Transfer time is the time to transfer the data. It depends on the rotating speed of the disk and number of bytes to be transferred.
- **Disk Access Time:** Disk Access Time is:
- **Disk Access Time = Seek Time + Rotational Latency + Transfer Time**

- **Disk Response Time**: Response Time is the average of time spent by a request waiting to perform its I/O operation.
- **Average Response time** is the response time of all requests. Variance Response Time is a measure of how individual requests are serviced with respect to average response time. So the disk scheduling algorithm that gives minimum variance response time is better.

Disk Scheduling Algorithms

FCFS: First Come First Serve is the simplest disk scheduling algorithm. In FCFS, the requests are served in the same order as they arrive to the disk queue.

- **Order of request is- (82,170,43,140,24,16,190)**
- **Current position of Read/Write head is : 50**
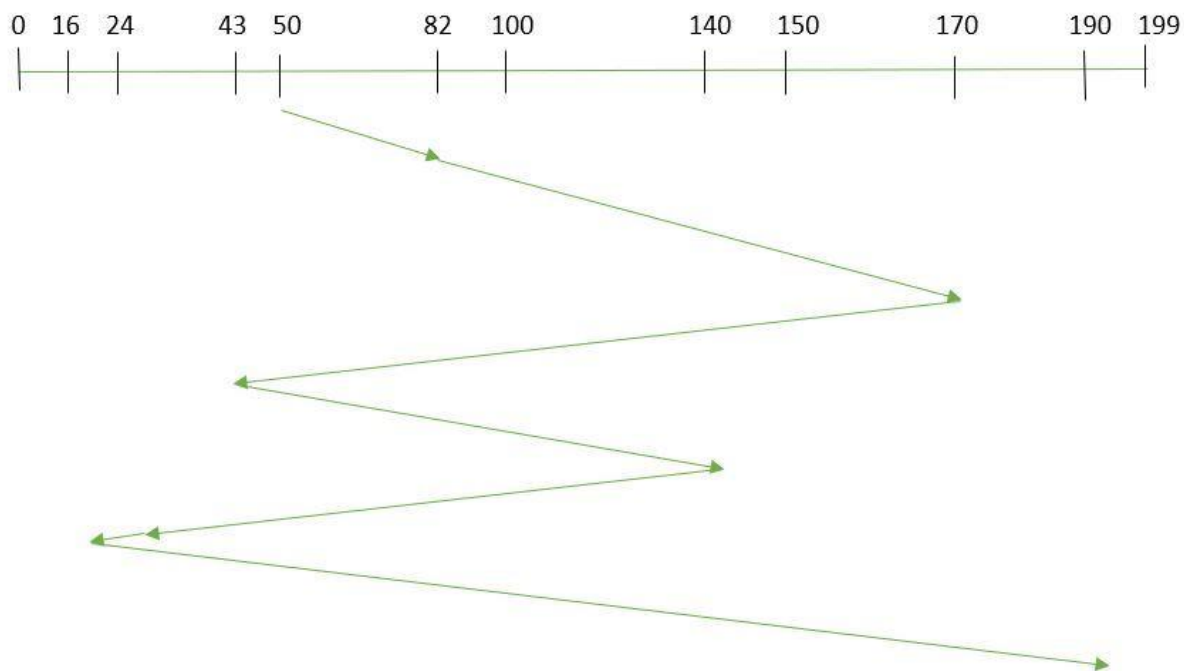
SSTF: In SSTF (Shortest Seek Time First), requests having the shortest seek time are executed first. So, the seek time of every request is calculated in advance in the queue and then they are scheduled according to their calculated seek time. As a result, the request near the disk arm will get executed first. SSTF is certainly an improvement over FCFS as it decreases the average response time and increases the throughput of the system.

- SSTF stands for Shortest Seek Time First.
- This algorithm services that next request which requires the least number of head movements from its current position regardless of the direction.
- It breaks the tie in the direction of head movement.

- **Order of request is- (82,170,43,140,24,16,190)**
- **Current position of Read/Write head is : 50**
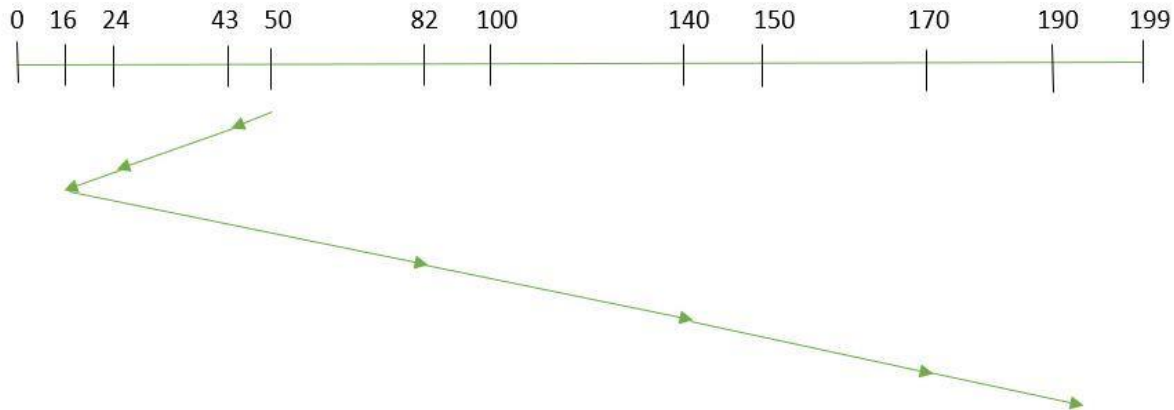
SCAN: In SCAN algorithm the disk arm moves into a particular direction and services the requests coming in its path and after reaching the end of disk, it reverses its direction and again services the request arriving in its path. So, this algorithm works as an elevator and hence is also known as elevator algorithm. As a result, the requests at the midrange are serviced more and those arriving behind the disk arm will have to wait.

- As the name suggests, this algorithm scans all the cylinders of the disk back and forth.
- Head starts from one end of the disk and move towards the other end servicing all the requests in between.
- After reaching the other end, head reverses its direction and move towards the starting end servicing all the requests in between.
- The same process repeats.

- **Order of the requests to be addressed are-82,170,43,140,24,16,190.**
- **The Read/Write arm is at 50, and it is also given that the disk arm should move "towards the larger value".**
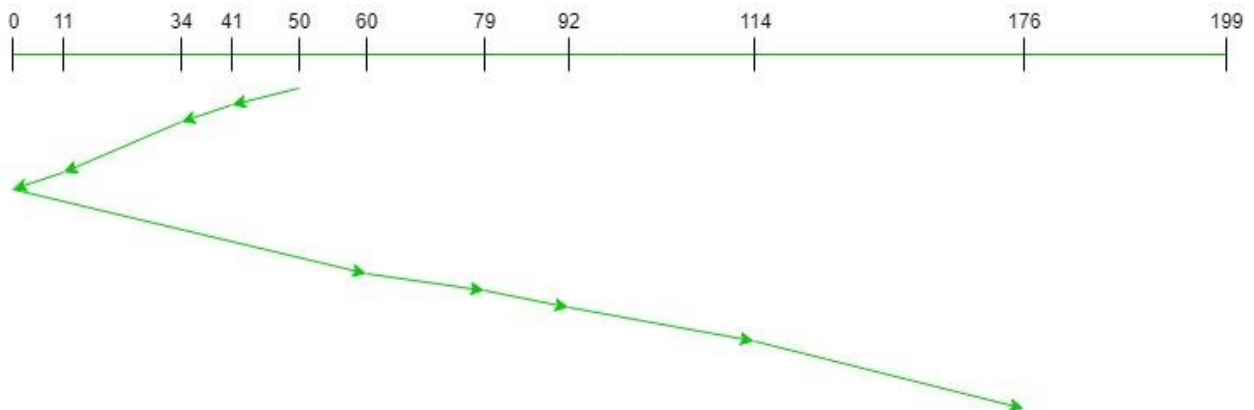
CSCAN: In SCAN algorithm, the disk arm again scans the path that has been scanned, after reversing its direction. So, it may be possible that too many requests are waiting at the other end or there may be zero or few requests pending at the scanned area.

- Circular-SCAN Algorithm is an improved version of the SCAN Algorithm.
- Head starts from one end of the disk and move towards the other end servicing all the requests in between.
- After reaching the other end, head reverses its direction.
- It then returns to the starting end without servicing any request in between.
- The same process repeats.

- **Order of the requests to be addressed are-82,170,43,140,24,16,190.**
- **The Read/Write arm is at 50, and it is also given that the disk arm should move "towards the larger value".**
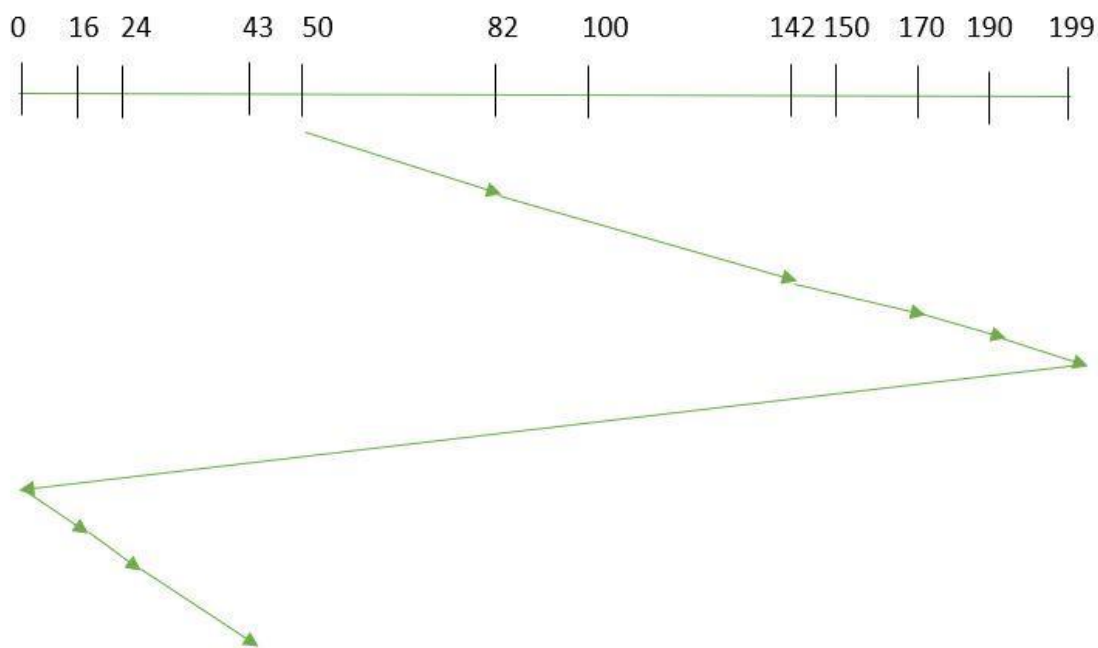
PROGRAM:

11. a) FCFS Disk scheduling algorithms

```
/* fcfsDiskAlgo */
#include<stdio.h>
int main()
{
        int queue[100],n,head,i,j,k,seek=0,diff;
        float avg;
printf(" \n FCFS Disk Scheduling Algorithm: \n");
        // printf("Enter the size of Queue\t");
printf("Enter the number of requests\t");
```

```
scanf("%d",&n);
        // printf("Enter the Queue\t");
printf("\nEnter the Request sequence \t");
        for(i=1;i<=n;i++)
        {
scanf("%d",&queue[i]);
        }
printf("Enter the initial head position\t");
scanf("%d",&head);
queue[0]=head;
printf("\n");
        for(j=0;j<=n-1;j++)
        {
                diff=abs(queue[j+1]-queue[j]);
                seek+=diff;
printf("Move from %d to %d with Seek %d\n",queue[j],queue[j+1],diff);
        }
printf("\nTotal Seek Time is %d\t",seek);
        avg=seek/(float)n;
printf("\nAverage Seek Time is %f\t",avg);
        return 0;
        }
```

OUTPUT:

ubuntu@ubuntu:~/OSLab$ geditfcfsDiskAlgo.c
ubuntu@ubuntu:~/OSLab$ gccfcfsDiskAlgo.c
ubuntu@ubuntu:~/OSLab$ ./a.out

FCFS Disk Scheduling Algorithm :

Enter the number of requests   5
Enter the Request sequence   25 36 54 125 145
Enter the initial head position   50
Move from 50 to 25 with Seek 25
Move from 25 to 36 with Seek 11
Move from 36 to 54 with Seek 18
Move from 54 to 125 with Seek 71
Move from 125 to 145 with Seek 20

Total Seek Time is 145
Average Seek Time is 29.000000

11. b) SCAN Disk Scheduling Algorithm

PROGRAM:
// C Program to Simulate SCAN Disk Scheduling Algorithm

```
#include<stdio.h>
int absoluteValue(int); // Declaring function absoluteValue

intmain()
{
  int queue[25],n,headposition,i,j,k,seek=0, maxrange,
  difference,temp,queue1[20],queue2[20],temp1=0,temp2=0;
  float averageSeekTime;
printf(" \n SCAN Disk Scheduling Algorithm: \n");

  // Reading the maximum Range of the Disk.
  printf("Enter the maximum range of Disk: ");
  scanf("%d",&maxrange);
  // Reading the number of Queue Requests(Disk access requests)
  printf("Enter the number of queue requests: ");
  scanf("%d",&n);

  // Reading the initial head position.(ie. the starting point of execution)
  printf("Enter the initial head position: ");
  scanf("%d",&headposition);

  // Reading disk positions to be read in the order of arrival
  printf("Enter the disk positions to be read(queue): ");
  for(i=1;i<=n;i++)   // Note that i varies from 1 to n instead of 0 to n-1
  {
  scanf("%d",&temp);  //Reading position value to a temporary variable

  //Now if the requested position is greater than current headposition,
  //then pushing that to array queue1
  if(temp>headposition)
  {
  queue1[temp1]=temp; //temp1 is the index variable of queue1[]
  temp1++; //incrementing temp1
  }
  else     //else if temp < current headposition,then push to array queue2[]
  {
  queue2[temp2]=temp; //temp2 is the index variable of queue2[]
  temp2++;
  }
```

```
}
// Sort the two arrays
//SORTING array queue1[] in ascending order
for(i=0;i<temp1-1;i++)
{
for(j=i+1;j<temp1;j++)
{
if(queue1[i]>queue1[j])
{
temp=queue1[i];
queue1[i]=queue1[j];
queue1[j]=temp;
}
}
}

//SORTING array queue2[] in descending order
for(i=0;i<temp2-1;i++)
{
for(j=i+1;j<temp2;j++)
{
if(queue2[i]<queue2[j])
{
temp=queue2[i];
queue2[i]=queue2[j];
queue2[j]=temp;
}
}
}
//Copying first array queue1[] into queue[]
for(i=1,j=0;j<temp1;i++,j++)
{
queue[i]=queue1[j];
}

//Setting queue[i] to maxrange because the head goes to
//end of disk and comes back in scan Algorithm
queue[i]=maxrange;

//Copying second array queue2[] after that first one is copied, into queue[]
for(i=temp1+2,j=0;j<temp2;i++,j++)
{
queue[i]=queue2[j];
}
```

```
  //Setting queue[i] to 0. Because that is the innermost cylinder.
  queue[i]=0;

  //At this point, we have the queue[] with the requests in thecorrect order of execution as per scan
algorithm.
  //Now we have to set 0th index of queue[] to be the initial headposition.
  queue[0]=headposition;

  // Calculating SEEK TIME. seek is initially set to 0 in the declaration part.

  for(j=0; j<=n; j++) //Loop starts from headposition. (ie. 0th index of queue)
  {
  // Finding the difference between next position and current position.
  difference = absoluteValue(queue[j+1]-queue[j]);

  // Adding difference to the current seek time value
  seek = seek + difference;

  // Displaying a message to show the movement of disk head
  printf("Disk head moves from position %d to %d with Seek %d \n",
  queue[j], queue[j+1], difference);
  }

  // Calculating Average Seek time
  averageSeekTime = seek/(float)n;

  //Display Total and Average Seek Time(s)
  printf("Total Seek Time= %d\n", seek);
  printf("Average Seek Time= %f\n", averageSeekTime);
  return 0;
}

// Defining function absoluteValue
int absoluteValue(int x)
{
  if(x>0)
  {
  return x;
  }
  else
  {
  return x*-1;
  }
}
```

OUTPUT:

ubuntu@ubuntu:~/OSLab$ geditscanDiskScheduleAlgo.c
ubuntu@ubuntu:~/OSLab$ gccscanDiskScheduleAlgo.c
ubuntu@ubuntu:~/OSLab$ ./a.out

SCAN Disk Scheduling Algorithm:
Enter the maximum range of Disk: 180
Enter the number of queue requests: 5
Enter the initial head position: 50
Enter the disk positions to be read(queue): 25 36 54 125 145
Disk head moves from position 50 to 54 with Seek 4
Disk head moves from position 54 to 125 with Seek 71
Disk head moves from position 125 to 145 with Seek 20
Disk head moves from position 145 to 180 with Seek 35
Disk head moves from position 180 to 36 with Seek 144
Disk head moves from position 36 to 25 with Seek 11
Total Seek Time= 285
Average Seek Time= 57.000000
ubuntu@ubuntu:~/OSLab$

11. c) C-SCAN Disk Scheduling Algorithm

PROGRAM:

```c
/* c-scan.c */
#include<stdio.h>
#include<stdlib.h>
int main()
{
  int RQ[100],i,j,n,TotalHeadMoment=0,initial,size,move;
printf(" \n C-SCAN Disk Scheduling Algorithm: \n");

  printf("Enter the number of Requests\t");
  scanf("%d",&n);
  printf("Enter the Requests sequence\t");
  for(i=0;i<n;i++)
  scanf("%d",&RQ[i]);
  printf("Enter initial head position\t");
  scanf("%d",&initial);
  printf("Enter total disk size\t");
  scanf("%d",&size);
  printf("Enter the head movement direction for high 1 and for low 0 -\t");
  scanf("%d",&move);

  // logic for C-Scan disk scheduling
  /*logic for sort the request array */
  for(i=0;i<n;i++)
  {
  for( j=0;j<n-i-1;j++)
  {
  if(RQ[j]>RQ[j+1])
  {
  int temp;
  temp=RQ[j];
  RQ[j]=RQ[j+1];
  RQ[j+1]=temp;
  }
  }
  }

  int index;
  for(i=0;i<n;i++)
  {
  if(initial<RQ[i])
```

```
{
index=i;
break;
}
}


// if movement is towards high value
if(move==1)
{
for(i=index;i<n;i++)
{
TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
initial=RQ[i];
}
//  last movement for max size
TotalHeadMoment=TotalHeadMoment+abs(size-RQ[i-1]-1);
/*movement max to min disk */
TotalHeadMoment=TotalHeadMoment+abs(size-1-0);
initial=0;
for( i=0;i<index;i++)
{
TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
initial=RQ[i];
}
}
// if movement is towards low value
else
{
for(i=index-1;i>=0;i--)
{
TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
initial=RQ[i];
}
//  last movement for min size
TotalHeadMoment=TotalHeadMoment+abs(RQ[i+1]-0);
/*movement min to max disk */
TotalHeadMoment=TotalHeadMoment+abs(size-1-0);
initial =size-1;
for(i=n-1;i>=index;i--)
{
TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
initial=RQ[i];

}
```

```
  }
  printf("\nTotal head movement is %d\n",TotalHeadMoment);
  return 0;
}
```

OUTPUT:

ubuntu@ubuntu:~/OSLab$ gedit c-scan.c
ubuntu@ubuntu:~/OSLab$ gcc c-scan.c
ubuntu@ubuntu:~/OSLab$ ./a.out

C-SCAN Disk Scheduling Algorithm:
Enter the number of Requests   3
Enter the Requests sequence   54 1 5
Enter initial head position   50
Enter total disk size   100
Enter the head movement direction for high 1 and for low 0 - 1

Total head movement is 153

ubuntu@ubuntu:~/OSLab$ gedit c-scan.c
ubuntu@ubuntu:~/OSLab$ gcc c-scan.c
ubuntu@ubuntu:~/OSLab$ ./a.out

C-SCAN Disk Scheduling Algorithm:
Enter the number of Requests   3
Enter the Requests sequence   54 1 5
Enter initial head position   50
Enter total disk size   100
Enter the head movement direction for high 1 and for low 0 - 0

Total head movement is 194
ubuntu@ubuntu:~/OSLab$

RESULT:

Thus, the various disk scheduling algorithms has been executed successfully.