SYLLABUS
OPERATING SYSTEMS LAB
* mandatory

1. Basic Linux commands
2. Shell programming -Command syntax -Write simple functions with basic tests, loops, patterns
3. System calls of Linux operating system:* fork, exec, getpid, exit, wait, close, stat, opendir, readdir
4. Write programs using the I/O system calls of Linux operating system (open, read, write)
5. Implement programs for Inter Process Communication using Shared Memory *
6. Implement Semaphores*
7. Implementation of CPU scheduling algorithms. a) Round Robin b) SJF c) FCFS d) Priority *
8. Implementation of the Memory Allocation Methods for fixed partition* a) First Fit b) Worst Fit c) Best Fit
9. Implement l page replacement algorithms a) FIFO b) LRU c) LFU*
10. Implement the banker's algorithm for deadlock avoidance. *
11. Implementation of Deadlock detection algorithm
12. Simulate file allocation strategies. b) Sequential b) Indexed c) Linked
13. Simulate disk scheduling algorithms. * c) FCFS b)SCAN c) C-SCAN

EXPERIMENT 1
Basic Linux commands

AIM:

To study about the basics of UNIX and Basic UNIX Commands.

UNIX:

It is a multi-user operating system. Developed at AT& T Bell Industries, USA in 1969.Ken Thomson along with Dennis Ritchie developed it from MULTICS (Multiplexed Information and Computing Service) OS. By1980, UNIX had been completely rewritten using C language.

LINUX:

It is similar to UNIX, which is created by Linus Torualds. All UNIXcommands works in Linux. Linux is a open source software. The main feature of Linux is coexisting with other OS such as windows and UNIX.

STRUCTURE OF A LINUXSYSTEM:

It consists of three parts.

a) UNIX kernel

b) Shells

c) Tools andApplications

UNIX KERNEL:

Kernel is the core of the UNIX OS. It controls all tasks, schedule all Processes and carries out all the functions of OS. Decides when one programs tops and another starts.

SHELL:

Shell is the command interpreter in the UNIX OS. It accepts command from the user and analyses and interprets them

Commands

a) date

–used to check the
date and time Syn:$date

Format	Purpose	Example	Result
+%m	To display only month	$date+%m	06
+%h	To display month name	$date+%h	June
+%d	To display day of month	$date+%d	O1
+%y	To display last two digits of years	$date+%y	09
+%H	To display hours	$date+%H	10
+%M	To display minutes	$date+%M	45
+%S	To display seconds	$date+%S	55

b) cal

–used to display the calendar Syn:$cal 2 2009

c) echo

–used to print the message on the screen.

Syn:$echo “text”

d) ls

–used to list the files. Your files are kept in a directory.

Syn:$ls –s

All files (include files with prefix) ls–l Lodetai (provide file statistics)

ls–t Order by creation time

ls– u Sort by access time (or show when last accessed together with–l)

ls–s Order by size

ls–r Reverseorder

ls–f Mark directories with /,executable with* , symbolic links with @, local sockets with =, named pipes(FIFOs)with

ls–s Show file size

ls– h“ Human Readable”, show file size in Kilo Bytes & Mega Bytes (h can be used together with –l or)

ls[a-m]*List all the files whose names begin with alphabetsFrom„a‟to„m‟ls[a]*List All the files whose name begins with „a‟or„A‟

Eg:$ls>mylist Output of „ls‟command is stored to disk file named„mylist‟

e) lp

–used to take printouts
Syn:$lp filename

f)man

–used to provide manual help on every UNIX commands.

Syn:$man unix command

$man cat

g) who &whoami

–it displays data about all users who have logged into the system currently.
The next command displays about current user only.
Syn:$who$whoami

h) uptime

–tells you how long the computer has been running since its last reboot or power-off.

Syn:$uptime

i)uname

–it displays the system information such as hardware platform, system name and processor, OS type.

Syn:$uname–a

j) hostname

–displays and set system host name
Syn:$ hostname

k) bc

–stands for„best calculator‟

FILE MANIPULATION COMMANDS

a) **cat**–this create, view and concatenatefiles.

Creation:

Syn:$cat>filename

Viewing:

Syn:$cat filename

Add text to an existing file:

Syn:$cat>>filename

Concatenate:

Syn:$catfile1file2>file3

$catfile1file2>>file3 (no over writing of file3)

b) **grep**–used to search a particular word or pattern related to that word from the file. Syn:$grep search wordfilename

Eg:$grep anu student

c) **rm**–deletes a file from the file system Syn:$rmfilename

d) **touch**–used to create a blankfile.

Syn:$touch file names

e) **cp**–copies the files or directories Syn:$cpsource file destination file

Eg:$cp student stud

f) **mv**–to rename the file or directory syn:$mv old file newfile

Eg:$mv–i student student list(-i prompt when overwrite)

g) **cut**–it cuts or pickup a given number of character or fields of the file. Syn:$cut<option><filename>

Eg: $cut –c filename

$cut–c1-10emp

$cut–f 3,6emp

$ cut –f 3-6 emp

-c cutting columns

-f cutting fields

h) **head**–displays10 lines from the head(top)of a given file Syn:$headfilename

Eg:$head student

To display the top two lines:

Syn:$head-2student

i) **tail**–displays last 10 lines of the file Syn:$tail filename

Eg:$tail student

To display the bottom two lines;

Syn:$ tail -2 student

j) **chmod**–used to change the permissions of a file ordirectory. Syn:$ch mod category operation permission file Where, Category–is the user type

Operation–is used to assign or remove permission Permission–is the type of permission

File–are used to assign or remove permission all

Examples:

$chmodu-wx student

Removes write and execute permission for users

$chmodu+rw,g+rwstudent

Assigns read and write permission for users and groups

$chmodg=rwx student

Assigns absolute permission for groups of all read, write and execute permissions

k) **wc**–it counts the number of lines, words, character in a specified file(s) with the options as–l,-w,-c

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

Syn: $wc –l filename

$wc –w filename

$wc–c filename

VIVA

Question 1: Which are the Linux Directory Commands?

There are 5 main Directory Commands in Linux:
pwd: Displays the path of the present working directory.
Syntax: `$ pwd`
ls: Lists all the files and directories in the present working directory.
Syntax: `$ ls`
cd: Used to change the present working directory.
Syntax: `$ cd <path to new directory>`
mkdir: Creates a new directory
Syntax: `$ mkdir <name (and path if required) of new directory>`

rmdir: Deletes a directory
Syntax: `$ rmdir <name (and path if required) of directory>`

Question 2:What is the redirection operator?

The redirection operator is used for redirecting the output of a specific command as an input to another command. The following are the two ways of using this:

1. '>' overwrites the file's existing content or creates a new one.
2. '>>' adds new content to the end of an existing file or creates a new one.

Question 3: Explain the 'ls' command in Linux
The ls command is used to list the files in a specified directory. The general syntax is:

$ ls <options> <directory>

For example, if you want to list all the files in the Example directory, then the command will be as follows:

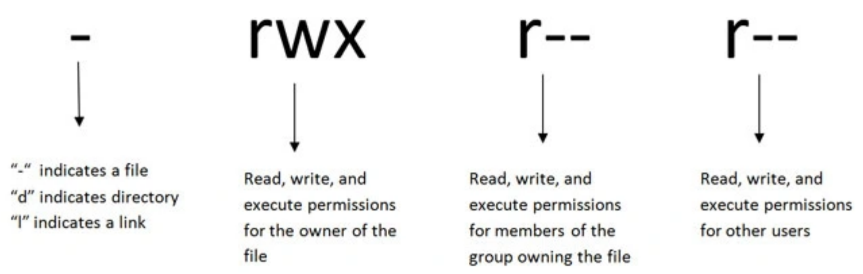
$ ls Example/

Question 4: How to change directory permissions in Linux?

To change directory permissions in Linux, use the following:chmod +rwx filename to add permissions
chmod -rwx directoryname to remove permissions.
chmod +x filename to allow executable permissions.
chmod -wx filename to take out write and executable permissions.
Note that "r" is for read, "w" is for write, and "x" is for execute.
This only changes the permissions for the owner of the file.

Question 4.2 What are the three permission groups?

There are three options for permission groups available to you in Linux. These are
- owners: these permissions will only apply to owners and will not affect other groups.
- groups: you can assign a group of users specific permissions, which will only impact users within the group.
- all users: these permissions will apply to all users, and as a result, they present the greatest security risk and should be assigned with caution.


-
rwx
r--
r--
"-" indicates a file
"d" indicates directory
"l" indicates a link
Read, write, and execute permissions for the owner of the file
Read, write, and execute permissions for members of the group owning the file
Read, write, and execute permissions for other users

Question 4.3: What are the three kinds of file permissions in Linux?
There are three kinds of file permissions in Linux:

Read (r): Allows a user or group to view a file.
Write (w): Permits the user to write or modify a file or directory.
Execute (x): A user or grup with execute permissions can execute a file or view a directory.

Question 4.5: How to change Linux permissions in numeric code
Use numbers instead of "r", "w", or "x".
Permission numbers are:

0 = ---
1 = --x
2 = -w-
3 = -wx
4 = r-
5 = r-x
6 = rw-
7 = rwx

For example:
chmod 777 foldername
will give read, write, and execute permissions for everyone.
chmod 700 foldername
will give read, write, and execute permissions for the user only.
chmod 327 foldername
will give write and execute (3) permission for the user, w (2) for the group, and read, write, and execute for the users

RESULT

The study about basics of UNIX and Basic UNIX Commands is done successfully.

EXPERIMENT 2
System calls

System calls of Linux operating system:* fork, exec, getpid, exit, wait, close, stat, opendir, readdir

2.1

PROGRAM

```
#include<stdio.h>
#include<dirent.h>
#include<stdlib.h>
struct dirent *dptr;
int main(int argc, char *argv[])
{
char buff[100];
DIR *dirp;
printf("enter the directory name");
scanf("%s",buff);
if((dirp=opendir(buff))==NULL)
{
printf("the given directory does not exist");
exit(1);
}
while(dptr=readdir(dirp))
{
printf("%s\n",dptr->d_name);
}
closedir(dirp);
}
```

OUTPUT

```
simat@simat-Veriton-Series:~$ gcc 1.c
simat@simat-Veriton-Series:~$ ./a.out
enter the directory name sreepathy
..
1
1.c
.
simat@simat-Veriton-Series:~$ ./a.out
enter the directory name welcome
```

```
the given directory does not exist
```

2.2

PROGRAM

```
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
void main()
{
int pid,pid1,pid2;
pid=fork();
if(pid==-1)
{
printf("error in process creation");
exit(1);
}
if(pid!=0)
{
pid1=getpid();
printf("the parent process id is %d",pid1);
}
else
{
pid2=getpid();
printf(" the child process id is %d",pid2);
}
}
```

OUTPUT

```
simat@simat-Veriton-Series:~$ gcc 2.c
simat@simat-Veriton-Series:~$ ./a.out
the parent process id is 4731 the child process id is 4732
```

2.3

PROGRAM

```
#include <unistd.h>
 int main(void) {
  char *binaryPath = "/bin/ls";
  char *arg1 = "-l";
  char *arg2 = "/home/simat";

  execl(binaryPath, binaryPath, arg1, arg2, NULL);

  return 0;
}
```

OUTPUT

```
simat@simat-Veriton-Series:~$ ./a.out
total 11484
drwxr-xr-x 3 simat simat        4096 Dec 16 12:05 006
-rw-rw-r-- 1 simat simat         360 Feb 13 10:09 1.c
drwxr-xr-x 2 simat simat        4096 Dec 20 15:07 21
-rw-rw-r-- 1 simat simat         307 Feb 13 10:17 2.c
-rwxrwxr-x 1 simat simat        8296 Feb 13 11:35 a.out
-rw-r--r-- 1 simat simat         173 Nov 16 14:43 arth.l
-rw-r--r-- 1 simat simat        2070 Nov 16 14:53 arth.tab.h
-rw-r--r-- 1 simat simat         507 Nov 16 14:53 arth.y
-rw-r--r-- 1 simat simat         642 Jan  3 10:48 bc.c
-rw-rw-r-- 1 simat simat         184 Feb  2 15:41 calc.l
-rw-r--r-- 1 simat simat       44730 Feb  2 15:45 calc.tab.c
-rw-r--r-- 1 simat simat        2050 Feb  2 15:45 calc.tab.h
```

2.4

PROGRAM

```
#include<stdio.h>
#include<sys/stat.h>
int main()
{
  //pointer to stat struct
  struct stat sfile;

  //stat system call
  stat("sreepathy", &sfile);

  //accessing st_mode (data member of stat struct)
  printf("st_mode = %o", sfile.st_mode);
  return 0;
}
```

OUTPUT
```
simat@simat-Veriton-Series:~$ gcc stat.c
simat@simat-Veriton-Series:~$ ./a.out
st_mode = 40775
```
2.5
PROGRAM
```
#include<stdio.h>
#include<sys/wait.h>
#include<unistd.h>
 int main()
{
	if (fork()== 0)
	printf("HC: hello from child\n");
	else
	{
	printf("HP: hello from parent\n");
	wait(NULL);
	printf("CT: child has terminated\n");
	}

	printf("Bye\n");
	return 0;
}
```
OUTPUT
```
simat@simat-Veriton-Series:~$ gcc wait.c
simat@simat-Veriton-Series:~$ ./a.out
HP: hello from parent
HC: hello from child
Bye
CT: child has terminated
Bye
```
VIVA

Question 1: Explain Process Management System Calls in Linux

The System Calls to manage the process are:

- **fork () : Used to create a new process**
- **exec() : Execute a new program**
- **wait() : Wait until the process finishes execution**
- **exit() : Exit from the process**

And the System Calls used to get Process ID are:

- **getpid():- get the unique process id of the process**

- **getppid():- get the parent process unique id**

Question 2

How to terminate a running process in Linux?

Every process has a unique process id. To terminate the process, we first need to find the process id. The ps command will list all the running processes along with the process id. And then we use the kill command to terminate the process.

Question 3: What is fork() System Call?

System call fork() is used to create processes. It takes no arguments and returns a process ID. The purpose of fork() is to create a new process, which becomes the child process of the caller. After a new child process is created, both processes will execute the next instruction following the fork() system call. Therefore, we have to distinguish the parent from the child. This can be done by testing the returned value of fork():

- **If fork() returns a negative value, the creation of a child process was unsuccessful.**
- **fork() returns a zero to the newly created child process.**
- **fork() returns a positive value, the process ID of the child process, to the parent.**

The returned process ID is of type pid_t defined in sys/types.h.

Question 4 :Difference between fork() and exec()

- The fork() acts as a system call that helps in creating processes.
- The exec() also refers to a system call that helps in creating processes.
- There is a primary difference between fork and exec. The fork, on the one hand, generates new processes while simultaneously preserving its parent process.
- The exec, on the other hand, creates new processes but doesn't preserve the parent process simultaneously.

Question 5 : What is a wait system call

The system call wait() function blocks the calling process until one of its child processes exits or a signal is received.

RESULT

System calls of Linux operating system is familiarized

EXP 3
Memory Allocation Methods

AIM
To Implement Memory Allocation Methods a) First Fit b) Worst Fit c) Best Fit

a) First Fit

ALGORITHM
—
PROGRAM : First Fit
```
#include <stdio.h>
#define max 25
int i, j, k = 0, nb, nf, temp = 0, high = 0, flag = 0;

void firstfit(int b[], int f[])
{
    for (i = 1; i <= nf; i++)
    {
        for (j = 1; j <= nb; j++)
        {
            temp = b[j] - f[i];
            if (temp >= 0)
            {
                k = j;
                printf("\nFile Size %d is put in %d partition", f[i], b[k]);
                b[k] = temp;
                flag = 1;
                break;
            }
        }
        if (flag == 0)
            printf("\nFile Size %d must Wait", f[i]);
        flag = 0;
    }
}

int main()
{
    int b[max], f[max];
    printf("\nMemory 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]);
    }
    firstfit(b, f);
    return 0;
}
```

Output: First Fit

```
Memory Management Scheme -First Fit
Enter the number of blocks : 5
Enter the number of files : 4
Enter the size of the blocks :
Block 1 : 100
Block 2 : 500
Block 3 : 200
Block 4 : 300
Block 5 : 600
Enter the size of the files :-
File 1: 212
File 2: 417
File 3: 112
File 4: 426
File Size 212 is put in 500 partition
File Size 417 is put in 600 partition
File Size 112 is put in 288 partition
File Size 426 must Wait
```

b) Worst Fit

ALGORITHM

—

PROGRAM : Worst Fit

```
#include <stdio.h>
#define max 25
int i, j, k = 0, nb, nf, temp = 0, highest = 0, flag = 0;

void worstfit(int b[], int f[])
{
    for (i = 1; i <= nf; i++)
    {
        for (j = 1; j <= nb; j++)
        {
            temp = b[j] - f[i];
            if (temp >= 0)
            {
                if (highest < temp)
                {
                    k = j;
                    highest = temp;
                }
            }
        }
        if (highest != 0)
            printf("\nFile Size %d is put in %d partition", f[i], b[k]);
        else
            printf("\nFile Size %d must wiat", f[i]);
        b[k] = highest;
        highest = 0;
    }
}

int main()
{
    int b[max], f[max];
    printf("\nMemory 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]);
    }
    worstfit(b, f);
    return 0;
}
```

Output : Worst Fit

```
Memory Management Scheme -Worst Fit
Enter the number of blocks : 5
Enter the number of files : 4
Enter the size of the blocks :
Block 1 : 100
Block 2 : 500
Block 3 : 200
Block 4 : 300
Block 5 : 600
Enter the size of the files :-
File 1: 212
File 2: 417
File 3: 112
File 4: 426
File Size 212 is put in 600 partition
File Size 417 is put in 500 partition
File Size 112 is put in 388 partition
File Size 426 must wait
```

c) Bestfit

ALGORITHM

—

PROGRAM : Best Fit

```
#include <stdio.h>
#define max 25
int i, j, k = 0, nb, nf, temp = 0, lowest = 999, flag = 0;

void bestfit(int b[], int f[])
{
    for (i = 1; i <= nf; i++)
    {
        for (j = 1; j <= nb; j++)
        {
            temp = b[j] - f[i];
            if (temp >= 0)
            {
                if (lowest > temp)
                {
                    k = j;
                    lowest = temp;
                }
            }
        }
        if (lowest != 999)
            printf("\nFile Size %d is put in %d partition", f[i], b[k]);
        else
            printf("\nFile Size %d must wiat", f[i]);

        b[k] = lowest;
        lowest = 999;
    }
}

int main()
{
    int b[max], f[max];
    printf("\nMemory 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]);
    }
    bestfit(b, f);
    return 0;
```

```
}
```

Output : Best Fit

```
Memory Management Scheme -Best Fit
Enter the number of blocks : 5
Enter the number of files : 4
Enter the size of the blocks :
Block 1 : 100
Block 2 : 500
Block 3 : 200
Block 4 : 300
Block 5 : 600
Enter the size of the files :-
File 1: 212
File 2: 417
File 3: 112
File 4: 426
File Size 212 is put in 300 partition
File Size 417 is put in 500 partition
File Size 112 is put in 200 partition
File Size 426 is put in 600 partition
```

VIVA

What do you mean by an operating system? What are its basic functions?

Operating System (OS) is basically a software program that manages and handles all resources of a computer such as hardware and software. An OS is responsible for managing, handling, and coordinating overall activities and sharing of computer resources. **It acts as an intermediary among users of computer and computer hardware.**

What are the different types of OS?

- Batched OS (Example: Payroll System, Transactions Process, etc.)
- Multi - Programmed OS (Example: Windows O/S, UNIX O/S, etc.)
- Time Sharing OS (Example: Multics, etc.)
- Distributed OS (LOCUS, etc.)
- Real-Time OS (PSOS, VRTX, etc.)

RESULT

Implementation of the Memory Allocation Methods for fixed partition is done successfully

Experiment 4
CPU scheduling algorithms

AIM

To implement the CPU scheduling algorithms. a) FCFS b) SJF c) Priority d) Round Robin

4.1 FCFS

ALGORITHM

```
1. Begin
2. Input the processes along with their burst time (bt).
3. Find turnaround time for all processes
4. Find waiting time (wt) for all processes.
    a. wt[0] = 0 ( Process 1 need not to wait)
    b. Find waiting time for all other processes
           wt= tat-bt
5. average waiting time =  total_waiting_time / no_of_processes.
6. average turnaround time =  total_turn_around_time/no_of_processes.
7. Display waiting time, turnaround time of each process
8. Display average waiting time,average turnaround time
9. End
```

PROGRAM : FCFS

```
#include <stdio.h>
struct process
{
   int pid, bt, ct, wt, tat;
} p[20];
int main()
{
   int n, i;
   printf("\nEnter the Number of Processes : ");
   scanf("%d", &n);
   for (i = 0; i < n; i++)
   {
      p[i].pid = i;
      printf("\nEnter the Burst time for Process %d : ", i);
      scanf("%d", &p[i].bt);
```

```
    }

    p[0].ct = 0;
    for (i = 0; i < n; i++)
    {
        p[i].ct = p[i - 1].ct + p[i].bt;
        p[i].tat = p[i].ct;
    }

    p[0].wt = 0;
    for (i = 0; i < n; i++)
        p[i].wt = p[i].tat - p[i].bt;

    printf("\nProcess execution order:\n");
    for (i = 0; i < n; i++)
        printf("P%d->", p[i].pid);

    printf("\n\tPID\t\tBurst Time\t Turnaround Time\t Waiting Time\n");
    for (i = 0; i < n; i++)
        printf("\n\t%d\t\t%d\t\t\t%d\t\t\t%d", p[i].pid, p[i].bt, p[i].tat, p[i].wt);
    printf("\n");

    float avgtat = 0, avgwt = 0;
    for (i = 0; i < n; i++)
    {
        avgtat += p[i].tat;
        avgwt += p[i].wt;
    }
    printf("Average Turn Around Time = %.2f\n", avgtat / n);
    printf("Average Waiting Time = %.2f\n", avgwt / n);

    return 0;
}
```

OUTPUT : FCFS

```
 Enter the Number of Processes : 4
Enter the Burst time for Process 0 : 21
Enter the Burst time for Process 1 : 3
Enter the Burst time for Process 2 : 6
Enter the Burst time for Process 3 : 2

Process execution order:
P0->P1->P2->P3->

      PID          Burst Time          Turnaround Time           Waiting Time
```

```
    0             21            21                0
    1             3             24                21
    2             6             30                24
    3             2             32                30
```

4.2 SJF

ALGORITHM

1. Begin
2. Input the processes along with their burst time (bt).
3. Sort the processes according to burst time, lowest to highest
4. Find turnaround time for all processes
5. Find waiting time (wt) for all processes.
    a. wt[0] = 0 (Process 1 in the sorted list need not to wait)
    b. Find waiting time for all other processes wt= tat-bt
6. average waiting time = total_waiting_time / no_of_processes.
7. average turnaround time = total_turn_around_time/no_of_processes.
8. Sort the processes according to the process id
9. Display waiting time, turnaround time of each process
10. Display average waiting time,average turnaround time
11. End

PROGRAM : SJF

```
#include <stdio.h>
struct process
{
   int pid, bt, ct, wt, tat;
} p[20], s;

int main()
{
   int n, i, j;
   printf("\n Enter the no. of processes : ");
   scanf("%d", &n);
   for (i = 0; i < n; i++)
   {
      p[i].pid = i;
      printf("\n Enter the burst time for Process %d : ", i);
      scanf("%d", &p[i].bt);
```

```
    }
    for (i = 0; i < n - 1; i++)
        for (j = 0; j < n - i - 1; j++)
            if (p[j].bt > p[j + 1].bt)
            {
                s = p[j];
                p[j] = p[j + 1];
                p[j + 1] = s;
            }
    p[-1].ct = 0;
    for (i = 0; i < n; i++)
    {
        p[i].ct = p[i - 1].ct + p[i].bt;
        p[i].tat = p[i].ct;
    }
    p[0].wt = 0;
    for (i = 1; i < n; i++)
        p[i].wt = p[i].tat - p[i].bt;
    printf("\nProcess execution order:\n");
    for (i = 0; i < n; i++)
        printf("P%d->", p[i].pid);
    for (i = 0; i < n - 1; i++)
        for (j = 0; j < n - i - 1; j++)
            if (p[j].pid > p[j + 1].pid)
            {
                s = p[j];
                p[j] = p[j + 1];
                p[j + 1] = s;
            }
    printf("\n\tPID\t\tBurst Time\t Turnaround Time\t Waiting Time\n");
    for (i = 0; i < n; i++)
        printf("\n\t%d\t\t%d\t\t\t%d\t\t\t%d", p[i].pid, p[i].bt, p[i].tat, p[i].wt);
    printf("\n");
    return 0;
}
```

Output : SJF

```
Enter the Number of Processes : 4
Enter the Burst time for Process 0 : 21
Enter the Burst time for Process 1 : 3
Enter the Burst time for Process 2 : 6
Enter the Burst time for Process 3 : 2

Process execution order:
P3->P1->P2->P0->

    PID           Burst Time     Turnaround Time      Waiting Time

     0               21               32                    11
     1               3                5                     2
     2               6                11                    5
     3               2                2                     0
```

4.3 Priority
Algorithm:

PROGRAM : Priority
```
#include <stdio.h>
struct process
{
    int pid, bt, ct, wt, tat, priority;
} p[20], s;
int main()
{
    int n, i, j;
    printf("\n Enter the no. of processes : ");
    scanf("%d", &n);
    for (i = 0; i < n; i++)
    {
        p[i].pid = i;
        printf("\n Enter the Priority for Process %d : ", i);
        scanf("%d", &p[i].priority);
        printf("\n Enter the burst time for Process %d : ", i);
        scanf("%d", &p[i].bt);
    }
    for (i = 0; i < n - 1; i++)
        for (j = 0; j < n - i - 1; j++)
            if (p[j].priority > p[j + 1].priority)
            {
                s = p[j];
                p[j] = p[j + 1];
                p[j + 1] = s;
            }
    p[-1].ct = 0;
    for (i = 0; i < n; i++)
    {
        p[i].ct = p[i - 1].ct + p[i].bt;
        p[i].tat = p[i].ct;
    }
    p[0].wt = 0;
    for (i = 1; i < n; i++)
        p[i].wt = p[i].tat - p[i].bt;
    printf("\nProcess execution order:\n");
    for (i = 0; i < n; i++)
        printf("P%d->", p[i].pid);
    for (i = 0; i < n - 1; i++)
        for (j = 0; j < n - i - 1; j++)
            if (p[j].pid > p[j + 1].pid)
            {
                s = p[j];
                p[j] = p[j + 1];
                p[j + 1] = s;
            }
    printf("\n\tPID\tPriority\tBurst Time\t Turnaround Time\t Waiting Time\n");
    for (i = 0; i < n; i++)
        printf("\n\t%d\t%d\t\t%d\t\t\t%d\t\t\t%d", p[i].pid, p[i].priority, p[i].bt, p[i].tat, p[i].wt);
    printf("\n");

    float avgtat = 0, avgwt = 0;
    for (i = 0; i < n; i++)
    {
        avgtat += p[i].tat;
        avgwt += p[i].wt;
    }
    printf("Average Turn Around Time = %.2f\n", avgtat / n);
    printf("Average Waiting Time = %.2f\n", avgwt / n);

    return 0;
}
```

Output : Priority

```
 Enter the no. of processes : 4
 Enter the Priority for Process 0 : 3
 Enter the burst time for Process 0 : 21
 Enter the Priority for Process 1 : 1
 Enter the burst time for Process 1 : 6
 Enter the Priority for Process 2 : 2
 Enter the burst time for Process 2 : 3
 Enter the Priority for Process 3 : 4
 Enter the burst time for Process 3 : 2

Process execution order:
```

```
P1->P2->P0->P3->

   PID     Priority            Burst Time       Turnaround Time        Waiting Time

    0          3                  21                  30                        9
    1          1                  6                   6                         0
    2          2                  3                   9                         6
    3          4                  2                   32                        30
```

4.4 Round Robin
Algorithm:

PROGRAM : Round Robin

```
#include <stdio.h>
#define MAX 20
int rq[20], front = 0, rear = -1;
struct process
{
   int pid, bt, btcopy, wt, tat;
} p[20], s;

void enqueue(int pid)
{
   if (rear > MAX - 1)
      return;
   else
   {
      rear = rear + 1;
      rq[rear] = pid;
   }
}

int dequeue()
{
   int ele, i;
   if (rear == -1)
      return -1;
   else
   {
        ele = rq[0];
        for (i = 0; i <= rear; i++)
            rq[i] = rq[i + 1];
        rear--;
        return ele;
    }
}

int main()
{
    int n, i, j, sum = 0, ts, id, ct = 0;
    printf("\nEnter the no. of processes : ");
    scanf("%d", &n);
    printf("\nEnter the Time Slice : ");
    scanf("%d", &ts);
    for (i = 0; i < n; i++)
    {
        p[i].pid = i;
        printf("\nEnter the burst time for Process %d : ", i);
        scanf("%d", &p[i].bt);
        p[i].btcopy = p[i].bt;
    }
    for (i = 0; i < n; i++)
    {
        enqueue(i);
    }
    id = dequeue();
    printf("\nExecution order: ");
    printf(" P%d->", id);
    while (id != -1)
    {
        if (p[id].bt > ts)
        {
            ct = ct + ts;
            p[id].tat = ct;
            p[id].bt = p[id].bt - ts;
            if (p[id].bt != 0)
                enqueue(p[id].pid);
        }
        else
        {
            ct = ct + p[id].bt;
            p[id].tat = ct;
            p[id].bt = 0;
        }
        id = dequeue();
        if (id != -1)
            printf(" P%d->", id);
    }
    for (i = 0; i < n; i++)
        p[i].wt = p[i].tat - p[i].btcopy;
    printf("\n\n\n\tPID\t\tBurst Time\t\t Turnaround Time\t Waiting Time\n");
    for (i = 0; i < n; i++)
        printf("\n\t%d\t\t\t%d\t\t\t%d\t\t\t%d", p[i].pid, p[i].btcopy, p[i].tat, p[i].wt);
    printf("\n");
    return 0;
}
```

Output : Round Robin

```
Enter the no. of processes : 4
Enter the Time Slice : 5
Enter the burst time for Process 0 : 21

Enter the burst time for Process 1 : 3
Enter the burst time for Process 2 : 6
Enter the burst time for Process 3 : 2

Execution order:  P0-> P1-> P2-> P3-> P0-> P2-> P0-> P0-> P0->


        PID         Burst Time            Turnaround Time      Waiting Time

         0              21                      32                    11
         1              3                       8                     5
         2              6                       21                    15
         3              2                       15                    13
```

VIVA

1. **What is a process?**

   The process is basically a program that is currently under execution.

2. **What are the different states of a process?**
   a. **New.** The process is being created.
   b. **Running.** Instructions are being executed.
   c. **Waiting.** The process is waiting for some event to occur (such as an I/O completion or reception of a signal).
   d. **Ready.** The process is waiting to be assigned to a processor.
   e. **Terminated.** The process has finished execution.

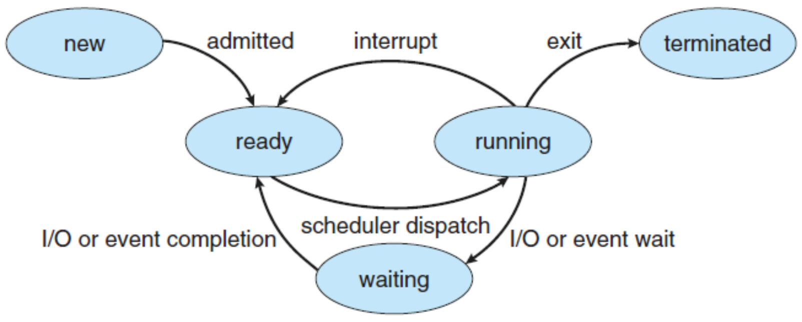


Figure 3.2 Diagram of process state.

3. **What is Context Switching?**

- Switching the CPU to another process requires performing a state save of the current process and a state restore of a different process. This task is known as a context switch.
- **Context-switch time is pure overhead, because the system does no useful work while switching.**

4. **What are different types of CPU Scheduling**

There 2 types of CPU Scheduling:

Non-Preemptive Scheduling

- In this scheduling, once the resources (CPU cycles) are allocated to a process, the process holds the CPU till it gets terminated or reaches a waiting state.

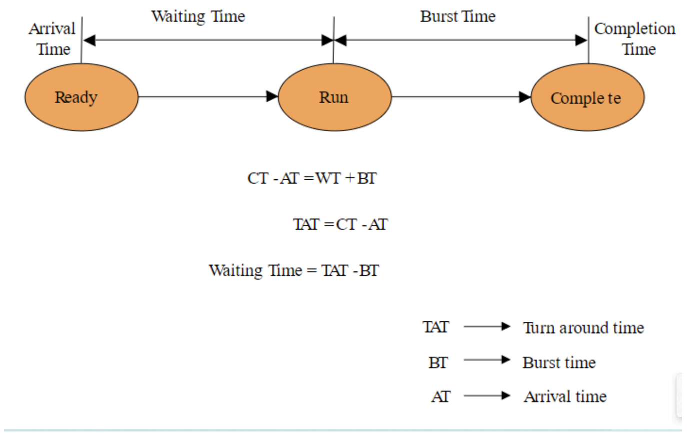
Preemptive Scheduling:
- Preemptive scheduling is used when a process switches from running state to ready state or from the waiting state to ready state.

5. What are the objectives of Process Scheduling Algorithms

- **Maximize CPU Utilization**: Keep CPU as busy as possible.
- **MaximumThroughput** : Number of processes that complete their execution per time unit should be maximized.
- **Minimum turnaround time**: Time taken by a process to finish execution should be the least.
- **Minimum waiting time:** Waiting time for a process must be minimum and the process should not starve in the ready queue.
- **Minimum response time**: It means that the time when a process produces the first response should be as less as possible.

6. What are the different Terminologies in CPU Scheduling algorithm

Arrival Time	: Time at which the process arrives in the ready queue.
Completion Time	: Time at which process completes its execution.
Burst Time	: Time required by a process for CPU execution.
Turn Around Time	: Time Difference between completion time and arrival time.
Waiting Time	: Time Difference between turn around time and burst time.



7. Compare different scheduling algorithms

Algorithm	Allocation Criteria	Average waiting time (AWT)	Preemption	Starvation
FCFS	**According to the arrival time of the processes, the CPU is allocated.**	**Large.**	**No**	**No**
SJF	**Based on the lowest CPU burst time (BT).**	**Smaller than FCFS**	**No**	**Yes**

SRTF	**Same as SJF the allocation of the CPU is based on the lowest CPU burst time (BT). But it is preemptive.**	**Depending on some measures e.g., arrival time, process size, etc**	**Yes**	**Yes**
RR	**According to the order of the process arrives with fixed time quantum (TQ)**	**Large as compared to SJF and Priority scheduling.**	**Yes**	**No**
Priority Pre-emptiv e	**According to the priority. The bigger priority task executes first**	**Smaller than FCFS**	**Yes**	**Yes**
Priority non-preem ptive	**According to the priority with monitoring the new incoming higher priority jobs**	**Preemptive Smaller than FCFS**	**No**	**Yes**

Experiment 5

IPC using Shared Memory

Aim: To Implement programs for Inter Process Communication using Shared Memory

ALGORITHM

Writer Process

1. Begin

2. Generate a unique key using function ftok()
3. Create a shared memory segment using shmget and get an identifier for the shared memory segment.
4. Attach to the shared memory segment created using shmat
5. Write the data to the reader to the shared memory segment created
6. Detach from the shared memory segment using shmdt()
7. End

Reader Process

1. Begin
2. Generate a unique key using function ftok()
3. Create a shared memory segment using shmget and get an identifier for the shared memory segment.
4. Attach to the shared memory segment created using shmat
5. Enter the data to the reader to the shared memory segment created
6. Detach from the shared memory segment using shmdt()
7. To destroy the shared memory call shmctl()
8. End

Program:

Code : Writer

```
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <string.h>
int main()
{
   key_t key = ftok("shmfile", 65);
   int shmid = shmget(key, 1024, 0666 | IPC_CREAT);
   char *str = (char *)shmat(shmid, (void *)0, 0);
   printf("Write Data : ");
   gets(str);
```

```
    printf("Data written in memory : %s\n", str);
    shmdt(str);
    return 0;
}
```

Code : Reader

```
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
int main()
{
    key_t key = ftok("shmfile", 65);
    int shmid = shmget(key, 1024, 0666 | IPC_CREAT);
    char *str = (char *)shmat(shmid, (void *)0, 0);
    printf("Data read from memory: %s\n", str);
    shmdt(str);
    shmctl(shmid, IPC_RMID, NULL);
    return 0;
}
```

Output :

```
simat@simat-desktop:~$ gcc writer.c
simat@simat-desktop:~$ ./a.out
Write Data : hai
Data written in memory: hai

simat@simat-desktop:~$ gcc reader.c
simat@simat-desktop:~$ ./a.out
Data read from memory: hai
```

VIVA

1. What is independent and cooperating processes?

   Independent Process
   A process is independent if it cannot affect or be affected by the other processes executing in the system.
   Any process that does not share data with any other process is independent

Cooperating process can affect or be affected by other processes, including sharing data

2. What is IPC?

Inter-process communication (IPC) is a mechanism that allows processes to communicate with each other and synchronize their actions. The communication between these processes can be seen as a method of co-operation between them. Processes can communicate with each other through both:

1. Shared Memory
2. Message passing

3. What is the difference between Shared memory and message passing?

.No	**Shared Memory Model**	**Message Passing Model**
1.	The shared memory region is used for communication.	A message passing facility is used for communication.
2.	Communicating processes reside on the same machine	Communicating processes reside on remote machines connected through a network.
3..	system calls are made only to establish the shared memory.	message passing is implemented through kernel intervention (system calls).
4	Faster communication strategy.	Relatively slower communication strategy.
5.	No kernel intervention.	It involves kernel intervention.
6	It can be used in exchanging larger amounts of data.	It can be used in exchanging small amounts of data.

4. What is Producer consumer problem?

The Producer-Consumer problem is a classic synchronization problem in operating systems.

The problem is defined as follows: there is a fixed-size buffer and a Producer process, and a Consumer process.

The Producer process creates an item and adds it to the shared buffer. The Consumer process takes items out of the shared buffer and “consumes” them.

Certain conditions must be met

1. The Producer process must not produce an item if the shared buffer is full.
2. The Consumer process must not consume an item if the shared buffer is empty.
3. At any given instance, only one process should be able to access the shared buffer and make changes to it.

5. What are the steps to be followed while implementing shared memory?

- A shared-memory region resides in the address space of the process creating the shared-memory segment.
- Other processes that wish to communicate using this shared-memory segment must attach it to their address space.

To use shared memory, we have to perform two basic steps:

1. Request a memory segment that can be shared between processes to the operating system.
2. Associate a part of that memory or the whole memory with the address space of the calling process.

EXPERIMENT 6

PRODUCER-CONSUMER PROBLEM USING SEMAPHORES

AIM

To simulate producer-consumer problem using semaphores.

```
#include <stdio.h>

#include <stdlib.h>
```

```
int mutex = 1, full = 0, empty = 3, x = 0;

int signal(int s)

{

    return (++s);

}

int wait(int s)

{

    return (--s);

}

void producer()

{

    empty = wait(empty);

    mutex = wait(mutex);

    x++;

    printf("\nProducer Produces the item %d ", x);

    mutex = signal(mutex);

    full = signal(full);

}

void consumer()

{

    full = wait(full);

    mutex = wait(mutex);

    printf("\nConsumer Consumes the item %d ", x);

    x--;

    mutex = signal(mutex);

    empty = signal(empty);
```

```
}
void main()
{
    int n;
    while (1)
    {
        printf("\n1.PRODUCER\n2.CONSUMER\n3.EXIT\n");
        printf("\nENTER YOUR CHOICE\n");
        scanf("%d", &n);
        switch (n)
        {
        case 1:
            if ((mutex == 1) && (empty != 0))
                producer();
            else
                printf("\nBUFFER IS FULL\n");
            break;
        case 2:
            if ((mutex == 1) && (full != 0))
                consumer();
            else
                printf("\nBUFFER IS EMPTY\n");
            break;
        case 3:
            exit(0);
            break;
        }
    }
}
```

VIVA

1. **What is meant by Process Synchronization?**

Process Synchronization is the task of synchronizing cooperative processes such that the processes do not have access to shared data or resources at the same time. This helps avoid inconsistency in data.

2. **What is race condition?**

A Race condition is a scenario that occurs in a multithreaded environment due to multiple threads sharing the same resource or executing the same piece of code. If not handled properly, this can lead to an undesirable situation, where the output state is dependent on the order of execution of the threads.

3. **What is meant by a Critical Section?**

Critical Section is a code segment where a shared variable is accessed and/or updated. The critical section should be treated as an atomic operation to avoid race conditions in the critical section. This is done by allowing only one thread to enter the critical section at any given point in time. The other threads should get access to it only after the previous thread is exited the critical section.

4. **What is meant by Semaphore?**

Semaphore is one of the most significant techniques to manage concurrent access to resources.

A semaphore is a signaling mechanism used to synchronize access to a resource. It is implemented using a flag (an integer variable) that is used to ensure that only the maximum allowed number of threads/processes can enter the critical section at a point in time.

5. **Explain two types of semaphores.**

Binary Semaphore: Initial value of the semaphore flag is 1. This ensures that only one thread can enter the critical section at a time. The value of the flag would either be 1 or 0.

Counting Semaphore: Initial value of the semaphore flag can be any positive number. The value of the flag denotes the maximum number of threads that can be in the critical section at any point in time.

6. **What are the different kinds of operations that are possible on semaphore?**

There are basically two atomic operations that are possible:

Wait()

Signal()

Implementation

```
wait () {

  while(s == 0); //Notice the ; at the end.

    s--;}

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

EXPERIMENT 7
BANKERS ALGORITHM

AIM

To Implement the banker's algorithm for deadlock avoidance

ALGORITHM

Safety Algorithm -the algorithm for finding out whether or not a system is in a safe state.
Let Work and Finish be vectors of length m and n, respectively.

```
1. Begin
2. Initialize Work = Available and  Finish[i] = false for i = 0, 1, ..., n − 1.
3. Find an index i such that both
      1. Finish[i] == false
      2. Needi ≤ Work
   If no such i exists, go to step 4.
4. Work =Work + Allocationi
   Finish[i] = true
   Go to step 2.
5.. If Finish[i] == true for all i, then the system is in a safe state.
6. End
```

PROGRAM

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int avail[100];
int total_res[100];
int total_alloc[100];
int req[100];
struct process
{
   int pid;
   int max[100];
   int alloc[100];
   int need[100];
   int finish;
} p[20], temp;
void main()
{
   int i, j, m, n, flag = 0, x = 0, k = 0, sequence[20], availtemp[100], a = 0, f = 0, work[50];
   printf("ENTER THE NUMBER OF RESOURCES : ");
```

```
scanf("%d", &m);
printf("MAXIMUM RESOURCE COUNT FOR : \n");
for (j = 0; j < m; j++)
{
    printf("\tRESOURCE %d : ", j);
    scanf("%d", &total_res[j]);
}
printf("\nENTER THE NUMBER OF PROCESSES : ");
scanf("%d", &n);
for (i = 0; i < n; i++)
{
    p[i].pid = i;
    printf("\nMAXIMUM ALLOCATION FOR PROCESS %d : ", p[i].pid);
    for (j = 0; j < m; j++)
        scanf("%d", &p[i].alloc[j]);
    printf("\nMAXIMUM REQUIREMENT FOR PROCESS %d : ", p[i].pid);
    for (j = 0; j < m; j++)
        scanf("%d", &p[i].max[j]);
    for (j = 0; j < m; j++)
        total_alloc[j] = total_alloc[j] + p[i].alloc[j];
    for (j = 0; j < m; j++)
        p[i].need[j] = p[i].max[j] - p[i].alloc[j];
    p[i].finish = 0;
}
printf("\n Matrix of Total Allocation\n");
for (i = 0; i < m; i++)
    printf("%d", total_alloc[i]);
printf("\n AVAILABLE MATRIX\n");
for (i = 0; i < m; i++)
    avail[i] = total_res[i] - total_alloc[i];
for (i = 0; i < m; i++)
{
    work[i] = avail[i];
    printf("%d", avail[i]);
}

for (i = 0; i < (n - 1) * m; i++)
    availtemp[i] = 0;

while (a < m)
{
    for (i = 0; i < n; i++)
    {
        if (p[i].finish != 1)
```

```
            {
                for (j = 0; j < m; j++)
                {
                    if (work[j] >= p[i].need[j])
                    {
                        flag = 1;
                    }
                    else
                    {
                        flag = 0;
                        break;
                    }
                }
                if (flag != 0)
                {
                    sequence[x] = p[i].pid;
                    x++;
                }
                f = i * m;
                for (j = 0; j < m; j++)
                {
                    if (flag != 0)
                    {
                        work[j] += p[i].alloc[j];
                        availtemp[f] = work[j];
                        f++;
                        p[i].finish = 1;
                    }
                }
            }
        }
        a++;
    }

    printf("\n PROCESS \tMAXIMUM \tALLOCATED \t NEED \t\t AVAIL \n");
    for (i = 0; i < n; i++)
    {
        printf("\n  %d \t", p[i].pid);
        printf(" \t ");
        for (j = 0; j < m; j++)
        {
            printf("%d ", p[i].max[j]);
        }
        printf(" \t ");
        for (j = 0; j < m; j++)
        {
            printf("%d ", p[i].alloc[j]);
        }
        printf(" \t ");
        for (j = 0; j < m; j++)
        {
            printf("%d ", p[i].need[j]);
        }
        printf(" \t ");
        for (j = i * m; j < m * (i + 1); j++)
        {
            printf("%d ", availtemp[j]);
        }
        printf("\n");
    }

    for (i = 0; i < n; i++)
    {
        if (p[i].finish == 0)
            f = 1;
        else
            f = 0;
    }
    if (f == 1)
        printf("\nSystem is Not Safe");
    else
    {
        printf("Sequence of Execution\n");
        for (i = 0; i < x; i++)
        {
            printf("P%d->", sequence[i]);
        }
        printf("\nSystem is Safe");
    }
}
```

Output :

```
ENTER THE NUMBER OF RESOURCES : 3
MAXIMUM RESOURCE COUNT FOR :
        RESOURCE 0 : 10
        RESOURCE 1 : 5
        RESOURCE 2 : 7
```

```
ENTER THE NUMBER OF PROCESSES : 5
MAXIMUM ALLOCATION FOR PROCESS 0 : 0 1 0
MAXIMUM REQUIREMENT FOR PROCESS 0 : 7 5 3
MAXIMUM ALLOCATION FOR PROCESS 1 : 2 0 0
MAXIMUM REQUIREMENT FOR PROCESS 1 : 3 2 2
MAXIMUM ALLOCATION FOR PROCESS 2 : 3 0 2
MAXIMUM REQUIREMENT FOR PROCESS 2 : 9 0 2
MAXIMUM ALLOCATION FOR PROCESS 3 : 2 1 1
MAXIMUM REQUIREMENT FOR PROCESS 3 : 4 2 2
MAXIMUM ALLOCATION FOR PROCESS 4 : 0 0 2
MAXIMUM REQUIREMENT FOR PROCESS 4 : 5 3 3
 Matrix of Total Allocation
725
 AVAILABLE MATRIX
332
 PROCESS      MAXIMUM          ALLOCATED       NEED        AVAIL

     0             7 5 3              0 1 0           7 4 3         7 5 5

     1             3 2 2              2 0 0          1 2 2          5 3 2

     2             9 0 2              3 0 2          6 0 0          10 5 7

     3             4 2 2              2 1 1          2 1 1          7 4 3

     4             5 3 3              0 0 2          5 3 1          7 4 5

Sequence of Execution

P1->P3->P4->P0->P2->

System is Safe
```

EXPERIMENT 8
Disk Scheduling Algorithms

AIM

Simulate disk scheduling algorithms. a) FCFS b)SCAN c) C-SCAN

PROGRAM 1 : FCFS

```
// FCFS Disk Scheduling algorithm
#include <stdio.h>
#include <math.h>
#define MAX 20

void FCFS(int arr[],int head,int size)
{
        int seek_count = 0;
        int cur_track, prev_track=head,distance;

        for(int i=0;i<size;i++)
        {
                cur_track = arr[i];

                // calculate absolute distance
                distance = fabs(prev_track - cur_track);

                // increase the total count
                seek_count += distance;

                // accessed track is now new head
                prev_track = cur_track;
        }

        printf("Total number of seek operations: %d\n",seek_count);

        // Seek sequence would be the same
        // as request array sequence
        printf("Seek Sequence is\n");
```

```
          for (int i = 0; i < size; i++) {
                    printf("%d->",arr[i]);
          }
}


int main()
{

          int arr[MAX],head,n,i,t;

          printf("Enter the total no. of tracks");
          scanf("%d",&t);

          printf("Enter the no. of requests");
          scanf("%d",&n);

          printf("Enter the track requests");
          for(i=0;i<n;i++)
             scanf("%d",&arr[i]);

          printf("Enter the position of read write head");
          scanf("%d",&head);

          FCFS(arr,head,n);

          return 0;
}
```

OUTPUT

```
Enter the total no. of tracks200
Enter the no. of requests9
Enter the track requests 60 143 15 185 85 120 33 28 146
Enter the position of read write head70
Total number of seek operations: 736
Seek Sequence is
60->143->15->185->85->120->33->28->146->
```

PROGRAM 2 : SCAN

```
#include <stdio.h>
int main()
{
    int head, req[20], n, i, j, seektime = 0, max, flag = 0, seq[20], c = 0, temp;
    printf("Enter the Max number of Cylinders : ");
    scanf("%d", &max);
    printf("Enter the Number of Requests : ");
    scanf("%d", &n);
    printf("Enter the requests : ");
    for (i = 0; i < n; i++)
    {
        scanf("%d", &req[i]);
        if (req[i] > max - 1)
            flag = 1;
    }
    if (flag == 1)
        printf("Process cannot complete");
    else
    {
        for (i = 0; i < n - 1; i++)
        {
            for (j = 0; j < n - i - 1; j++)
            {
                if (req[j] > req[j + 1])
                {
                    temp = req[j];
                    req[j] = req[j + 1];
                    req[j + 1] = temp;
                }
            }
        }
        printf("Enter the current position of the head : ");
        scanf("%d", &head);
        j = 0;
        while (req[j] <= head)
        {
            j++;
        }
        for (i = j; i < n; i++)
        {
            seektime += req[i] - head;
            head = req[i];
            seq[c] = req[i];
            c++;
        }
        seektime += (max - 1) - req[n - 1];
        head = max - 1;
        for (i = j - 1; i >= 0; i--)
        {
            seektime += head - req[i];
            head = req[i];
            seq[c] = req[i];
            c++;
        }
        printf("Seektime =%d\n", seektime);
        printf("Seek Sequence : ");
        for (i = 0; i < c; i++)
            printf("%d->", seq[i]);
    }
}
```

Output : SCAN

```
Enter the Max number of Cylinders : 200
Enter the Number of Requests : 7
Enter the requests : 82 170 43 140 24 16 190
Enter the current position of the head : 50
Seektime =332
Seek Sequence : 82->140->170->190->43->24->16->
```

PROGRAM 2 : C-SCAN

```
#include <stdio.h>
int main()
{
    int head, current, req[20], n, i, j, seektime = 0, max, flag = 0, seq[20], c = 0, temp;
    printf("Enter the Max number of Cylinders : ");
    scanf("%d", &max);
    printf("Enter the Number of Requests : ");
    scanf("%d", &n);
    printf("Enter the requests : ");
    for (i = 0; i < n; i++)
    {
        scanf("%d", &req[i]);
        if (req[i] > max - 1)
            flag = 1;
```

```
}
if (flag == 1)
    printf("Process cannot complete");
else
{
    for (i = 0; i < n - 1; i++)
    {
        for (j = 0; j < n - i - 1; j++)
        {
            if (req[j] > req[j + 1])
            {
                temp = req[j];
                req[j] = req[j + 1];
                req[j + 1] = temp;
            }
        }
    }
    printf("Enter the current position of the head : ");
    scanf("%d", &head);
    j = 0;
    while (req[j] <= head)
    {
        j++;
    }
    for (i = j; i < n; i++)
    {
        seektime += req[i] - head;
        head = req[i];
        seq[c] = req[i];
        c++;
    }
    seektime += (max - 1) - req[n - 1];
    seektime += max - 1;
    head = 0;
    for (i = 0; i < j; i++)
    {
        seektime += req[i] - head;
        head = req[i];
        seq[c] = req[i];
        c++;
    }
    printf("Seektime =%d\n", seektime);
    printf("Seek Sequence : ");
    for (i = 0; i < c; i++)
```

```
            printf("%d->", seq[i]);
    }
}
```

Output : C-SCAN

```
Enter the Max number of Cylinders : 200
Enter the Number of Requests : 7
Enter the requests : 82 170 43 140 24 16 190
Enter the current position of the head : 50
Seektime =391
Seek Sequence : 82->140->170->190->16->24->43->
```

VIVA

1. **What is disk scheduling**
   Disk scheduling is done by operating systems to schedule I/O requests arriving for the disk. Disk scheduling is also known as I/O scheduling.

2. **Compare disk scheduling algorithms based on criteria,starvation, seek time,load, waiting and response time, throughput, disadvantages**

FCFS	SCAN	C-SCAN
First Come-First Serve	Elevator Algorithm	Circular SCAN
It services the IO requests in the order in which they arrive.	The disk arm moves into a particular direction till the end, satisfying all the requests coming in its path, and then it turns back and moves in the reverse direction satisfying requests coming in its path.	The arm of the disk moves in a particular direction servicing requests until it reaches the last cylinder, then it jumps to the last cylinder of the opposite direction without servicing any request then it turns back and starts moving in that direction servicing the remaining requests.
There is no starvation in this algorithm, every request is serviced.	Starvation is still possible	It will never cause starvation to any requests.
does not optimize the seek time.	Improved variance of seek time as compared to SSTF.	improved variance of seek time as compared to SCAN.
when the load on a disk is light and small systems where efficiency is not important.	Under a light load, SCAN policy is best.	Under a heavy load, C-SCAN policy is best.
May not provide the best possible service in terms of good waiting & response time.	Low variance occurs in waiting & response time.	Uniform waiting time & better response time is provided.
It has extremely low throughput due to lengthy seeks.	It has higher throughput than FCFS,	It maintains a high level of throughput by avoiding discrimination against the innermost and outermost cylinders.
Disadvantages: The short processes which are at the back of the queue have to wait for the long process at the front to finish	***Disadvantages:*** Long waiting time occurs for the cylinders which are just visited by the head. In SCAN the head moves till the end of the disk despite the absence of requests to be serviced.	***Disadvantages:*** More seek movements are caused in C-SCAN compared to SCAN Algorithm. In C-SCAN even if there are no requests left to be serviced the Head will still travel to the end of the disk.

3. **Explain the terms: seek time, rotational latency,disk access time, disk response time**

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.

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.

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.

EXPERIMENT 9

PAGE REPLACEMENT

AIM

To implement page replacement algorithms a) FIFO b) LRU

a)FIFO
ALGORITHM

PROGRAM : FIFO (First in First Out)
```
#include <stdio.h>
int main()
{
    int reference[50], frame[50], fsize, i, j, flag = 0, c = 0, n, fault = 0;
    float miss, hit;
    printf("Enter the number of reference :  ");
    scanf("%d", &n);
    printf("Enter the references : ");
    for (i = 0; i < n; i++)
        scanf("%d", &reference[i]);
    printf("Enter the frame size : ");
    scanf("%d", &fsize);
    for (i = 0; i < fsize; i++)
        frame[i] = -1;
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < fsize; j++)
        {
            if (frame[j] == reference[i])
                flag = 1;
        }
        if (flag != 1)
        {
            fault++;
            frame[c] = reference[i];
            c++;
        }
        if (c == fsize)
            c = 0;
        flag = 0;
    }
    printf("Total number of faults = %d\n", fault);
    miss = ((float)fault / n) * 100;
    hit = ((float)(n - fault) / n) * 100;
    printf("Total number of references = %d\n", n);
    printf("Miss ratio = %.2f%\n", miss);
    printf("Hit ratio = %.2f%\n", hit);

    return 0;
}
```

Output : FIFO

```
Enter the number of reference :  13
Enter the references : 5 7 6 0 7 1 7 2 0 1 7 1 0
Enter the frame size : 4
Total number of faults = 7
Total number of references = 13
Miss ratio = 53.85%
Hit ratio = 46.15%
```

b)LRU
ALGORITHM

PROGRAM - LRU (LEAST RECENTLY USED)

```
#include <stdio.h>
int main()
{
     int reference[50], frame[50], fsize, i, j, flag = 0, c = 0, n, fault = 0, k, recent[50], temp, flag1 =
0;
    float miss, hit;
    printf("Enter the number of reference :  ");
    scanf("%d", &n);
    printf("Enter the references : ");
    for (i = 0; i < n; i++)
        scanf("%d", &reference[i]);
    printf("Enter the frame size : ");
    scanf("%d", &fsize);
    for (i = 0; i < fsize; i++)
    {
        fault++;
        frame[i] = reference[i];
        recent[i] = reference[i];
        c++;
    }
```

```
recent[c]=0;
for (i = fsize; i < n; i++)
{
    for (k = 0; k < c; k++)
    {
        if (reference[i] == recent[k])
        {
            flag1 = 1;
            break;
        }
    }
    if (flag1 == 1)
    {
        temp = recent[k];
        for (j = k; j < c; j++)
            recent[j] = recent[j + 1];
        recent[c - 1] = temp;
    }
    else
    {
        recent[c] = reference[i];
        c++;
        recent[c]=0;
    }
    flag1=0;

    for (j = 0; j < fsize; j++)
    {
        if (frame[j] == reference[i])
            flag = 1;
    }
    if (flag != 1)
    {
        fault++;
        for (k = 0; k < c; k++)
        {
            for (j = 0; j < fsize; j++)
            {
                if (recent[k] == frame[j])
                {
                    frame[j] = reference[i];
                    goto end;
                }
            }
            }
        }
    end:
        flag = 0;
    }

    printf("Total number of faults = %d\n", fault);
    miss = ((float)fault / n) * 100;
    hit = ((float)(n - fault) / n) * 100;
    printf("Total number of references = %d\n", n);
    printf("Miss ratio = %.2f%\n", miss);
    printf("Hit ratio = %.2f%\n", hit);

    return 0;
}
```

Output : LRU

```
Enter the number of reference :  13
Enter the references : 5 7 6 0 7 1 7 2 0 1 7 1 0
Enter the frame size : 3
Total number of faults = 9
Total number of references = 13
Miss ratio = 69.23%
Hit ratio = 30.77%
```

VIVA

1. Which of the page replacement algorithms suffers from Belady’s Anomaly?
Answer: FIFO
Bélády’s anomaly is the name given to the phenomenon where increasing the number of page frames results in an increase in the number of page faults for a given memory access pattern.

2. **Optimal page replacement algorithm** i replaces the page whose demand in the future is least as compared to other pages from frames (secondary memory).
The purpose of this algorithm is to minimize the number of page faults. Also, one of the most famous abnormalities in the paging technique is "Belady's Anomaly", which is least seen in this algorithm.

3. **Page fault**: A page fault occurs when a program attempts to access data or code that is in its address space, but is not currently located in the system RAM.

4. **Virtual memory** is a memory management technique where secondary memory can be used as if it were a part of the main memory. Virtual memory is a common technique used in a computer's operating system (OS).

5. Difference: PAging and segmentation

S.NO	Paging	Segmentation
1.	In paging, the program is divided into fixed or mounted size pages.	In segmentation, the program is divided into variable size sections.
2.	For the paging operating system is accountable.	For segmentation compiler is accountable.
3.	Page size is determined by hardware.	Here, the section size is given by the user.
4.	It is faster in comparison to segmentation.	Segmentation is slow.
5.	**Paging could result in internal fragmentation.**	**Segmentation could result in external fragmentation.**
6.	In paging, the logical address is split into a page number and page offset.	Here, the logical address is split into section number and section offset.
7.	Paging comprises a page table that encloses the base address of every page.	While segmentation also comprises the segment table which encloses segment number and segment offset.
8.	The page table is employed to keep up the page data.	Section Table maintains the section data.

9.	In paging, the operating system must maintain a free frame list.	In segmentation, the operating system maintains a list of holes in the main memory.
10.	Paging is invisible to the user.	Segmentation is visible to the user.
11.	In paging, the processor needs the page number, and offset to calculate the absolute address.	In segmentation, the processor uses segment number, and offset to calculate the full address.
12.	It is hard to allow sharing of procedures between processes.	Facilitates sharing of procedures between the processes.
13	In paging, a programmer cannot efficiently handle data structure.	It can efficiently handle data structures.
14.	This protection is hard to apply.	Easy to apply for protection in segmentation.
15.	The size of the page needs always be equal to the size of frames.	There is no constraint on the size of segments.
16.	A page is referred to as a physical unit of information.	A segment is referred to as a logical unit of information.
17.	Paging results in a less efficient system.	Segmentation results in a more efficient system.

SHELL PROGRAMMING

AIM
To write simple functions with basic tests, loops, patterns in shell

ALGORITHM

PROGRAM

Steps:

gedit filename.sh
bash filename.sh

8.1 EVEN OR ODD

```
echo "enter the number"
read num
if [ `expr $num % 2` -eq 0]
then
echo "number is even"
else
echo "number is odd"
fi
```

8.2 BIGGEST OF 2 NUMBERS

```
echo "enter the number"
read a b
if [ $a -gt $b]
then
echo "A is big"
else
echo "B is big"
fi
```

8.3 BIGGEST OF THREE NUMBERS

```
echo "enter three numbers"
read a b c
if [ $a -gt $b ] && [ $a -gt $c]
then
echo "A is big"
else if [ $b -gt $c ]
then
echo "B is big"
else
echo "C is big"
fi
fi
```

8.4 FACTORIAL OF A NUMBER USING FOR LOOP

```
echo "Enter a number"

# Read the number
read num

fact=1

for((i=2;i<=num;i++))
{
 fact=$((fact * i))
}

echo  "factorial of $num is $fact"
```

8.5 FACTORIAL OF A NUMBER USING WHILE LOOP

```
# shell script for factorial of a number
# factorial using while loop

echo "Enter a number"

# Read the number
read num
fact=1

# -gt is used for '>' Greater than sign
while [ $num -gt 1 ]
do
  fact=$((fact * num))
  num=$((num - 1))
done

# Printing the value of the factorial
echo  "factorial of $num is $fact"
```

c) Bestfit

ALGORITHM

—

PROGRAM : Best Fit

```
#include <stdio.h>
#define max 25
int i, j, k = 0, nb, nf, temp = 0, lowest = 999, flag = 0;

void bestfit(int b[], int f[])
{
    for (i = 1; i <= nf; i++)
    {
        for (j = 1; j <= nb; j++)
        {
            temp = b[j] - f[i];
            if (temp >= 0)
            {
                if (lowest > temp)
                {
                    k = j;
                    lowest = temp;
                }
            }
        }
        if (lowest != 999)
            printf("\nFile Size %d is put in %d partition", f[i], b[k]);
        else
```

```
            printf("\nFile Size %d must wiat", f[i]);

        b[k] = lowest;
        lowest = 999;
    }
}

int main()
{
    int b[max], f[max];
    printf("\nMemory 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]);
    }
    bestfit(b, f);
    return 0;
}
```

Output : Best Fit

```
Memory Management Scheme -Best Fit
Enter the number of blocks : 5
Enter the number of files : 4
Enter the size of the blocks :
Block 1 : 100
Block 2 : 500
Block 3 : 200
Block 4 : 300
Block 5 : 600
Enter the size of the files :-
File 1: 212
File 2: 417
File 3: 112
File 4: 426
File Size 212 is put in 300 partition
File Size 417 is put in 500 partition
File Size 112 is put in 200 partition
File Size 426 is put in 600 partition
```