SRM VALLIAMMAI ENGINEERING COLLEGE

(An Autonomous Institution)

SRM Nagar, Kattankulathur-603203.

DEPARTMENT OF CYBER SECURITY

IT3464 - Operating Systems Lab Manual

Regulation 2023

II Year (IV semester) 2024-25

Prepared by

Ms. V. PREMA, Assistant Professor (Sr.G)/ CSE

Dr. G. SANGEETHA, Assistant Professor (Sel.G)/ CSE

Ms. PRIYADARSHINI, Assistant Professor/ CSE

Ms. M. RAGHAVI, Assistant Professor (O.G)/CYS

Ms. J. ABINAYA, Assistant Professor (O.G)/IT

PROGRAMME OUTCOMES

1. **Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis:** Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using the first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions:** Design solutions for the complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems:** Use research-based knowledge and research methods including the design of experiments, analysis and interpretation of data, and the synthesis of the information to provide valid conclusions.
5. **Modern tool usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6. **The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development. **8. Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project management and finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. **Life-long learning:** Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

PROGRAM SPECIFIC OUTCOMES (PSOs)

PSO1: Exhibit proficiency in planning, implementing and evaluating team oriented-software programming solutions to specific business problems and society needs.

PSO2: Demonstrate professional skills in applying programming skills, competency and decision making capability through hands-on experiences.

PSO3: Apply logical thinking in analyzing complex real world problems, and use professional and ethical behaviors to provide proper solutions to those problems.

PSO4: Demonstrate the ability to work effectively as part of a team in applying technology to Business and personal situations.

PROGRAM EDUCATIONAL OBJECTIVES (PEOs)

PEO1: To mould students to exhibit top performance in higher education and research and to become a State-of –theart technocrat.

PEO2: To impart the necessary background in Computer Science and Engineering by providing solid foundation in Mathematical, Science and Engineering fundamentals.

PEO3: To equip the students with the breadth of Computer Science and Engineering innovate novel solutions for the benefit of common man.

CO – PO and PSO MAPPING:

Course Outcomes	**Programme Outcomes (PO)**												**PSO**			
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**1**	**2**	**3**	**4**
CO 1	3	1	3	1	1	-	-	-	-	-	-	-	-	2	-	-
CO 2	3	1	1	2	2	-	-	-	-	-	-	-	-	2	-	-
CO 3	3	3	2	1	2	-	-	-	-	-	-	-	-	2	-	-
CO 4	1	2	2	3	2	-	-	-	-	-	-	-	-	2	-	-
CO 5	2	2	1	1	3	-	-	-	-	-	-	-	-	2	-	--
AVG	2	2	2	2	2	-	-	-	-	-	-	-	-	2	-	-

IT3464 OPERATING SYSTEMS LABORATORY L T P C

0 0 3 1.5

OBJECTIVES:

- To understand the basics of Unix command and shell programming.
- To implement various CPU scheduling algorithms.
- To implement Deadlock Avoidance and Deadlock Detection Algorithms
- To implement Page Replacement Algorithms
- To implement various memory allocation methods.
- To be familiar with File Organization and File Allocation Strategies.

LIST OF EXPERIMENTS:

1. UNIX commands and Basic Shell Programming
2. Process Management using System Calls : Fork, Exit, Getpid, Wait, Close
3. Write C programs to implement the various CPU Scheduling Algorithms
4. Implement mutual exclusion by Semaphore
5. Write C programs to avoid Deadlock using Banker's Algorithm
6. Write a C program to Implement Deadlock Detection Algorithm
7. Write C program to implement Threading
8. Write C program to Implement the paging Technique.
9. Write C programs to implement the following Memory Allocation Methods

 a. First Fit
 b.Worst Fit
 c. Best Fit

10 Write C programs to implement the various Page Replacement Algorithms
11. Write C programs to implement the various File Organization Techniques
12. Implement the following File Allocation Strategies using C programs
a. Sequential
b. Indexed
c. Linked
13. Write C programs for the implementation of various disk scheduling algorithms

COURSE OUTCOMES:

At the end of this course, the students will be able to:

CO1: Define and implement UNIX Commands.

CO2: Compare the performance of various CPU Scheduling Algorithms.

CO3: Compare and contrast various Memory Allocation Methods.

CO4: Define File Organization and File Allocation Strategies.

CO5: Implement various Disk Scheduling Algorithms

Exp. No.	List of Experiments
1	UNIX commands and Basic Shell Programming
2	Process Management using System Calls : Fork, Exit, Getpid, Wait, Close
3	Write C programs to implement the various CPU Scheduling Algorithms
4	Implement mutual exclusion by Semaphore
5	Write C programs to avoid Deadlock using Banker's Algorithm
6	Write a C program to Implement Deadlock Detection Algorithm
7	Write C program to implement Threading
8	Write C program to Implement the paging Technique
9	Write C programs to implement the following Memory Allocation Methods a. First Fit b.Worst Fit c. Best Fit
10	Write C programs to implement the various Page Replacement Algorithms
11	Write C programs to Implement the various File Organization Techniques
12	Implement the following File Allocation Strategies using C programs a. Sequential b. Indexed c. Linked
13	Write C programs for the implementation of various disk scheduling algorithms

EX.NO:1 **BASICS OF UNIX COMMANDS**

AIM:

To study and execute Unix commands.

PROCEDURE:

Unix is security conscious, and can be used only by those persons who have an account.

Telnet (Telephone Network) is a Terminal emulator program for TCP/IP networks that enables users to log on to remote servers.

To logon, type telnet server_ipaddress in run window.

User has to authenticate himself by providing username and password. Once verified, a greeting and $ prompt appears. The shell is now ready to receive commands from the user. Options suffixed with a hyphen (–) and arguments are separated by space**.**

GENERAL COMMANDS

Command	**Function**
Date	Used to display the current system date and time.
date +%D	Displays date only
date +%T	Displays time only
date +% Y	Displays the year part of date
date +% H	Displays the hour part of time
Cal	Calendar of the current month
Calyear	Displays calendar for all months of the specified year
calmonth year	Displays calendar for the specified month of the year
Who	Login details of all users such as their IP, Terminal No, User name,
who am i	Used to display the login details of the user
Tty	Used to display the terminal name
Uname	Displays the Operating System
uname –r	Shows version number of the OS (kernel).
uname –n	Displays domain name of the server
echo "txt"	Displays the given text on the screen
echo $HOME	Displays the user's home directory
Bc	Basic calculator. Press **Ctrl+d**to quit
Lpfile	Allows the user to spool a job along with others in a print queue.
man cmdname	Manual for the given command. Press **q** to exit
History	To display the commands used by the user since log on.
Exit	Exit from a process. If shell is the only process then logs out

DIRECTORY COMMANDS

Command	Function
pwd	Path of the present working directory
mkdirdir	A directory is created in the given name under the current Directory
mkdirdir1 dir2	A number of sub-directories can be created under one stroke
cd subdir	Change Directory. If the subdirstarts with / then path starts from **root** (absolute) otherwise from current working directory.
Cd	To switch to the home directory.
cd /	To switch to the root directory.
cd..	To move back to the parent directory
rmdirsubdir	Removes an empty sub-directory.

FILE COMMANDS

Command	Function
cat >filename	To create a file with some contents. To end typing press **Ctrl+d**. The >symbol means redirecting output to a file. (<for input)
cat filename	Displays the file contents.
cat >>filename	Used to append contents to a file
cpsrc des	Copy files to given location. If already exists, it will be Overwritten
cp –i src des	Warns the user prior to overwriting the destination file
cp –r src des	Copies the entire directory, all its sub-directories and files.
mv old new	To rename an existing file or directory. –i option can also be Used
mv f1 f2 f3 dir	To move a group of files to a directory.
mv –v old new	Display name of each file as it is moved.
Rmfile	Used to delete a file or group of files. –i option can also be used
rm *	To delete all the files in the directory.
rm –r *	Deletes all files and sub-directories
rm –f *	To forcibly remove even write-protected files
Ls	Lists all files and subdirectories (blue colored) in sorted manner.
Lsname	To check whether a file or directory exists.
lsname*****	Short-hand notation to list out filenames of a specific pattern.
ls –a	Lists all files including hidden files (files beginning with **.**)
ls –x dirname	To have specific listing of a directory.
ls –R	Recursive listing of all files in the subdirectories

ls –l	Long listing showing file access rights (read/write/execute-**rwx** for user/group/others-**ugo**).
cmpfile1 file2	Used to compare two files. Displays nothing if files are identical.
Wcfile	It produces a statistics of lines (**l**), words(**w**), and characters(**c**).
chmodperm file	Changes permission for the specified file. (r=4, w=2, x=1) chmod 740 file sets all rights for user, read only for groups and no rights for others

OUTPUT

GENERAL COMMANDS

```
[student@veccse  ~]date
Sat May 16 06:10:34 UTC 2020
[student@veccse ~]date +%D
05/16/20
[student@veccse ~]date +%T
10:13:11
[student@veccse ~]date +%Y
2020
[student@veccse ~]date +%H
10
[student@veccse  ~]cal
May 2020
Su  Mo  Tu  We  Th  Fr  Sa
                     1   2
 3   4   5   6   7   8   9
10  11  12  13  14  15  16
17  18  19  20  21  22  23
24  25  26  27  28  29  30
31
[student@veccse ~]cal 2020
  2020
        January                      February                      March
Su  Mo  Tu  We  Th  Fr  Sa    Su  Mo  Tu  We  Th  Fr  Sa    Su  Mo  Tu  We  Th  Fr  Sa
             1   2   3   4                             1     1   2   3   4   5   6   7
 5   6   7   8   9  10  11     2   3   4   5   6   7   8     8   9  10  11  12  13  14
12  13  14  15  16  17  18     9  10  11  12  13  14  15    15  16  17  18  19  20  21
19  20  21  22  23  24  25    16  17  18  19  20  21  22    22  23  24  25  26  27  28
```

```
26 27 28 29 30 31                23 24 25 26 27 28 29             29 30 31

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

[student@veccse ~]cal 2020

```
July 2020
Su  Mo  Tu  We  Th  Fr  Sa
                 1   2   3   4
 5   6   7   8   9  10  11
12  13  14  15  16  17  18
19  20  21  22  23  24  25
26  27  28  29  30  31
```

[student@veccse ~]who

```
 studentpts/1              May 16 10:05 (172.16.1.14)
```

[student@veccse ~]who am i

```
studentpts/1 May 16 10:05 (172.16.1.14)
```

```
[student@veccse  ~]tty
/dev/pts/1
[student@veccse  ~]uname
Linux
[student@veccse ~]echo "hello"
hello
[student@veccse ~]echo $HOME
/home/student
[student@veccse ~]bc bc
1.06
Copyright 1991-1994, 1997, 1998, 2000 Free Software Foundation, Inc. This is free
software with ABSOLUTELY NO WARRANTY.
For details type `warranty'.
[student@veccse  ~]man  lp
lp(1)   Easy Software Products lp(1)
NAME
lp - print files cancel - cancel jobs SYNOPSIS
lp [ -E ]  [ -c ] [ -d destination ] [ -h server ] [ -m ] [ -n num- copies [ -o option ] [ -q
priority ] [ -s ] [ -t title ] [ -H handling
] [ -P page-list ] [ file(s) ]
lp [  -E  ]  [  -c ] [ -h server ] [ -i job-id ] [ -n num-copies [ -o option ] [ -q priority ] [ -t
title ] [ -H handling ] [ -P page-list ] cancel [ -a ] [ -h server ] [ -u username ] [ id ] [
destination ] [ destination-id ]
DESCRIPTION
lpsubmits files for printing or alters a pending job. Use a filename of "-" to force
printing from the standard input.
cancelcancels existing print jobs. The -a option will remove all jobs from the
specified destination.
OPTIONS
The following options are recognized by lp:
[student@veccse  ~]history
1       date
```

```
2       date +%D
3       date +%T
4       date +%Y
5       date +%H
6       cal
7       cal 2020
8          cal 7 2020
10      who
11      who am i
12      tty
13      uname
14      uname -r
15      uname -n
16      echo "helloi"
17      echo $HOME
18      bc
19      man lp
20      history
```

DIRECTORY COMMANDS

```
[student@veccse]$  pwd
/home/student
[student@veccse ~]mkdir san
[student@veccse ~]mkdir s1 s2
[student@veccse ~]ls
 s1 s2 san   [student@veccse
~]cd s1 [student@veccse
s1]$ cd / [student@veccse
/]$ cd . .
[student@veccse /]$ rmdir s1
[student@veccse ~]$ ls
 s2 san
```

FILE COMMANDS

```
[student@vecit  ~]$  cat>test
hi welcome operating systems lab
[student@vecit ~]$ cat test
hi welcome operating systems lab [student@vecit
~]$ cat>>test fourth semester [student@vecit ~]$
cat test
hi welcome operating systems lab fourth semester
[student@vecit ~]$ cat>test1
[student@vecit ~]$ cp test test1
[student@vecit ~]$ cat test1
hi welcome operating systems lab fourth semester [student@vecit
~]$ cp -i test test1 cp: overwrite `test1'? y [student@vecit ~]$ cp
-r test test1
[student@vecit  ~]$  ls
s s2 san swap.sh temp.sh test TEST test1
[student@vecit ~]$ mv san san1 [student@vecit
~]$ ls
s s2 san1 swap.sh temp.sh test TEST test1
[student@vecit ~]$ mv test test1 san1
[student@vecit ~]$ mv -v san1 sannew
`san1' -> `sannew'
[student@vecit  ~]$  ls
s s2 sannew swap.sh temp.sh TEST
[student@vecit ~]$ cmp test test1 cmp:
test: No such file or directory
```

RESULT

Thus the study and execution of Unix commands has been completed successfully.

EX. NO: 1A: **SIMPLE SHELL PROGRAMS**

AIM:

To write simple shell scripts using shell programming fundamentals.

DESCRIPTION:

The activities of a shell are not restricted to command interpretation alone. The shell also has Rudimentary programming features. When a group of commands has to be executed regularly, they are stored in a file (with extension .sh). All such files are called shell scripts or shell programs. Shell programs run in interpretive mode.

The original UNIX came with the Bourne shell (sh) and it is universal even today. Then came a plethora of shells offering new features. Two of them, C shell (csh) and Korn shell (ksh) has been well accepted by the UNIX fraternity. Linux offers Bash shell (bash) as a superior alternative to Bourne shell.

Preliminaries

1. Comments in shell script start with #. It can be placed anywhere in a line; the shell ignores contents to its right. Comments are recommended but not mandatory
2. Shell variables are loosely typed i.e. not declared. Their type depends on the value assigned. Variables when used in an expression or output must be prefixed by $.
3. The read statement is shell's internal tool for making scripts interactive.
4. Output is displayed using echo statement. Any text should be within quotes. Escape sequence should be used with –e option.
5. Commands are always enclosed with `` (back quotes).
6. Expressions are computed using the expr command. Arithmetic operators are + - * / %. Meta characters * ( ) should be escaped with a \.
7. Multiple statements can be written in a single line separated by ;
8. The shell scripts are executed using the sh command (sh filename).

<u>Swapping values of two variables</u>

Algorithm

Step 1 : Start

Step 2 : Read the values of a and b

Step 3 : Interchange the values of a and b using another variable t as follows: t = a

a = b b = t

Step 4 : Print a and b

Step 5 : Stop

Program (swap.sh) # Swapping values

```
echo -n "Enter value for A : " read a
echo -n "Enter value for B : " read b
t=$a a=$b b=$t
echo "Values after Swapping" echo "A Value is $a"
echo "B Value is $b"
```

Output

```
[student@vecit ~]$ sh swap.sh Enter Value for A:5
Enter Value for B:6 Values after Swapping A value is 6
B values is 5 [student@vecit ~]$
```

RESULT

Thus using programming basics, simple shell scripts were executed

EX.NO.1B: CONDITIONAL CONSTRUCTS

AIM:

To write shell scripts using decision-making constructs.

DESCRIPTION:

Shell supports decision-making using if statement. The if statement like its counterpart in programming languages has the following formats. The first construct executes the statements when the condition is true. The second construct adds an optional else to the first one that has different set of statements to be executed depending on whether the condition is true or false. The last one is an elif ladder, in which conditions are tested in sequence, but only one set of statements is executed.

Operator	**Description**
-eq	Equal to
-ne	Not equal to
-gt	Greater than
-ge	Greater than or equal to
-lt	Less than
-le	Less than or equal to
-a	Logical AND
-o	Logical OR
!	Logical NOT

Odd or even

Algorithm Step 1 :
Start

```
Step 2 : Read number
Step 3 : If number divisible by 2 then Print "Number is Even"
Step 3.1 : else Print "Number is Odd"
Step 4 : Stop Program
```

# Odd or even using if-else

```
echo -n "Enter a non-zero number : " readnum
rem=`expr $num % 2` if [ $rem -eq 0 ]
then
echo "$num is Even" else
echo "$num is Odd" fi
```

Output

```
[student@vecit ~]$ sh oddeven.sh
Enter a non-zero number : 12 12 is Even
```

String comparison

Algorithm

```
Step 1 : Start
Step 2 : Read strings str1 and str2
Step 3 : If str1 = str2 then Print "Strings are the same"
Step 3.1 : else Print "Strings are distinct"
Step 4 : Stop
```

Program

```
echo -n "Enter the first string : " read s1
echo -n "Enter the second string : " read s2
if [ $s1 == $s2 ] then
echo "Strings are the same" else
echo "Strings are distinct" fi
```

Output

```
[student@vecit ~]$ sh strcomp.sh
Enter the first string :ece-a Enter the second string : ECE-A Strings are distinct
```

RESULT

Thus using if statement scripts with conditional expressions were executed

EX.NO.2: IMPLEMENTATION OF FORK, EXEC, GETPID, EXIT, WAIT, AND CLOSE SYSTEM CALLS.

AIM:

To write a program for implementing process management using the following system calls of UNIX operating system: fork, exec, getpid, exit, wait, close.

ALGORITHM:

1. Start the program.
2. Read the input from the command line.
3. Use fork() system call to create process, getppid() system call used to get the parent process ID and getpid() system call used to get the current process ID
4. execvp() system call used to execute that command given on that command line argument
5. execlp() system call used to execute specified command.
6. Open the directory at specified in command line input.
7. Display the directory contents.
8. Stop the program.

PROGRAM:

```
#include<stdio.h> main(int
 arc,char*ar[])
{
        int pid; char s[100]; pid=fork();
        if(pid<0)
                printf("error");
```

```
else if(pid>0)
{
        wait(NULL);
        printf("\n Parent Process:\n");  printf("\n\tParent
        Process id:%d\t\n",getpid());
        execlp("cat","cat",ar[1],(char*)0);
        error("can’t execute cat %s,",ar[1]);
}
```

```
        else
        {
                printf("\nChild  process:");
                printf("\n\tChildprocess parent id:\t  %d",getppid());
                printf(s,"\n\tChild process id :\t%d",getpid()); write(1,s,strlen(s));
                printf("  ");
                printf("  ");
                printf(" "); execvp(ar[2],&ar[2]);
                error("can't execute %s",ar[2]);



        }
}
```

OUTPUT:

```
[root@localhost ~]# ./a.out tst date Child process:
Child process id :
3137 Sat Apr 10 02:45:32 IST 2010
Parent  Process:
Parent Process id:3136 sd
dsaASD[root@localhost ~]# cat tst sd
dsaASD
```

RESULT:

Thus the program for process management was written and successfully executed.

EX.NO.3 A: IMPLEMENTATION OF FCFS SCHEDULING ALGORITHM

AIM

To write a C program to implement First Come First Serve scheduling algorithm.

DESRIPTION:

For FCFS scheduling algorithm, read the number of processes/jobs in the system, their CPU burst times. The scheduling is performed on the basis of arrival time of the processes irrespective of their other parameters. Each process will be executed according to its arrival time. Calculate the waiting time and turnaround time of each of the processes accordingly.

.ALGORITHM:

Step 1: Start the program.

Step 2: Get the input process and their burst time.

Step 3: Sort the processes based on order in which it requests CPU.

Step 4: Compute the waiting time and turnaround time for each process. Step 5: Calculate the average waiting time and average turnaround time. Step 6: Print the details about all the processes.

Step 7: Stop the program.

PROGRAM

PROGRAM:

```
#include<stdio.h>
Void main()
{
        int bt[50],wt[80],at[80],wat[30],ft[80],tat[80]; int
        i,n;
        float awt,att,sum=0,sum1=0; char
        p[10][5];
        printf("\nenter the number of process          ");
        scanf("%d",&n);
        printf("\nEnter the process name and burst-time:"); for(i=0;i<n;i++)
        scanf("%s%d",p[i],&bt[i]);
printf("\nEnter the arrival-time:");
for(i=0;i<n;i++)
        scanf("%d",&at[i]);
wt[0]=0;
for(i=1;i<=n;i++)
        wt[i]=wt[i-1]+bt[i-1];
ft[0]=bt[0];
for(i=1;i<=n;i++)
        ft[i]=ft[i-1]+bt[i];
printf("\n\n\t\t\tGANTT CHART\n");
printf("\n          \n");
for(i=0;i<n;i++)
        printf("|\t%s\t",p[i]);
printf("|\t\n");
printf("\n          \n");
printf("\n");
for(i=0;i<n;i++)
        printf("%d\t\t",wt[i]);
printf("%d",wt[n]+bt[n]);
printf("\n          \n");
printf("\n");
for(i=0;i<n;i++)
        wat[i]=wt[i]-at[i];
for(i=0;i<n;i++)
        tat[i]=wat[i]-at[i];
printf("\n FIRST COME FIRST SERVE\n");
printf("\n Process Burst-time Arrival-time Waiting-time Finish-time Turnaround- time\n");
for(i=0;i<n;i++)
        printf("\n\n        %d%s      \t       %d\t\t        %d      \t\t        %d\t\t        %d      \t\t
        %d",i+1,p[i],bt[i],at[i],wat[i],ft[i],tat[i]);
for(i=0;i<n;i++)
        sum=sum+wat[i];
awt=sum/n;
```

```
        for(i=0;i<n;i++)
                sum1=sum1+bt[i]+wt[i];
        att=sum1/n;
        printf("\n\nAverage waiting time:%f",awt);
        printf("\n\nAverage  turnaround  time:%f",att);
}
```

OUTPUT:

```
enter the number of process 3
Enter the process name and burst-time: p1
2
p2 3
p3 4
Enter the arrival-time:0 1 2
GANTT CHART

|       p1       |       p2       |       p3       |

0                2                5                9

FIRST COME FIRST SERVE
Process        Burst-time     Arrival-time   Waitin          Finish-time    Turnaround-time
p1             2                      0                              2              2
p2             3                      1                              5              4
p3             4                      2                              9              7
Average waiting time:1.333333 Average
turnaround time:5.333333
```

RESULT:

The FCFS scheduling algorithm has been implemented in C.

EX.NO.3B : IMPLEMENTATION OF SJF SCHEDULING ALGORITHM

AIM:

To write a C program to implement shortest job first (non-pre-emptive) scheduling algorithm.

DESCRIPTION:

For SJF scheduling algorithm, read the number of processes/jobs in the system, their CPU burst times. Arrange all the jobs in order with respect to their burst times. There may be two jobs in queue with the same execution time, and then FCFS approach is to be performed. Each process will be executed according to the length of its burst time. Then calculate the waiting time and turnaround time of each of the processes accordingly.

ALGORITHM:

Step 1: Start the program.

Step 2: Get the input process and their burst time. Step

3: Sort the processes based on burst time.

Step 4: Compute the waiting time and turnaround time for each process. Step 5:

Calculate the average waiting time and average turnaround time. Step 6: Print

the details about all the processes.

Step 7: Stop the program.

PROGRAM:

```
#include<stdio.h>
void main()
{
        int  i,j,n,bt[30],at[30],st[30],ft[30],wat[30],wt[30],temp,temp1,tot,tt[30];   float
        awt, att;
        int p[15];
        wat[1]=0;
        printf("ENTER THE NO.OF PROCESS");
        scanf("%d",&n);
        printf("\nENTER THE PROCESS NUMBER,BURST TIME AND
        ARRIVAL TIME");
```

```
for(i=1;i<=n;i++)
{
        scanf("%d\t %d\t %d",&p[i],&bt[i],&at[i]);
}
printf("\nPROCESS\tBURSTTIME\tARRIVALTIME");
for(i=1;i<=n;i++)
{
        printf("\np%d\t%d\t\t%d",p[i],bt[i],at[i]);
}
for(i=1;i<=n;i++)
{
        for(j=i+1;j<=n;j++)
        {
                if(bt[i]>bt[j])
                {
                        temp=bt[i];
                        bt[i]=bt[j];
                        bt[j]=temp;
                        temp1=p[i];
                        p[i]=p[j];
                        p[j]=temp1;
                }
        }
        if(i==1)
        {
                st[1]=0;
                ft[1]=bt[1];  wt[1]=0;
        }
        else
        {
                st[i]=ft[i-1];
                ft[i]=st[i]+bt[i];
```

```
                    wt[i]=st[i];
          }
}
printf("\n\n\t\t\tGANTT CHART\n");
printf("\n       \n");
for(i=1;i<=n;i++)
          printf("|\tp%d\t",p[i]);
printf("|\t\n");
printf("\n       \n");
printf("\n");
for(i=1;i<=n;i++)
          printf("%d  \t\t",wt[i]);
printf("%d",wt[n]+bt[n]);
printf("\n       \n");
for(i=2;i<=n;i++)
          wat[i]=wt[i]-at[i];
for(i=1;i<=n;i++)
           tt[i]=wat[i]+bt[i]-at[i];
printf("\nPROCESS\tBURSTTIME\tARRIVALTIME\tWAITINGTIME\tT
URNAROUNDTI ME\n");
for(i=1;i<=n;i++)
{
               printf("\np%d %5d  %15d  %15d  %15d",p[i],bt[i],at[i],wat[i],tt[i]);
}
for(i=1,tot=0;i<=n;i++)
          tot+=wt[i];
awt=(float)tot/n;
printf("\n\n\n AVERAGE WAITING TIME=%f",awt);
for(i=1,tot=0;i<=n;i++)
          tot+=tt[i];
att=(float)tot/n;
printf("\n\n AVERAGE TURNAROUND TIME=%f",att);
```

```
}
```

OUTPUT:

```
enter the no.of process3
enter the process number,burst time and arrival time
1 8 1
2 5 1
3 3 1
PROCESS BURSTTIM   ARRIVALTIM  WAITINGTIM  TURNAROUNDTIM
        E          E           E                E
p3           3          1           0            2
p2           5          1           2            6
p1           8          1           7            14
AVERAGE WAITING TIME=3.666667
AVERAGE TURNAROUND
TIME=7.333333
```

RESULT:

The SJF scheduling algorithm has been implemented in C.

EX.NO.3C: IMPLEMENTATION OF ROUND ROBIN SCHEDULING ALGORITHM

AIM:

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

DESCRIPTION:

For round robin scheduling algorithm, read the number of processes/jobs in the system, their CPU burst times, and the size of the time slice. Time slices are assigned to each process in equal portions and in circular order, handling all processes execution. This allows every process to get an equal chance. Calculate the waiting time and turnaround time of each of the processes accordingly.

ALGORITHM:

Step 1: Start the program.

Step 2: Get the input process and their burst time. Step
3: Sort the processes based on priority.

Step 4: Compute the waiting time and turnaround time for each process. Step 5: Calculate the average waiting time and average turnaround time. Step 6: Print the details about all the processes.

Step 7: Stop the program.

PROGRAM:

```
#include<stdio.h> voidmain()
{
        int ct=0,y[30],j=0,bt[10],cwt=0; int
        tq,i,max=0,n,wt[10],t[10],at[10],tt[10],b[10];
        float a=0.0,s=0.0;
        char  p[10][10];
        printf("\n enter the no of process:");
        scanf("%d",&n);
        printf("\nenter the time quantum");
        scanf("%d",&tq);
        printf("\nenter the process name,bursttime,arrival time");
```

```
for(i=0;i<n;i++)
{
        scanf("%s",p[i]);
        scanf("%d",&bt[i]);
        scanf("%d",&at[i]); wt[i]=t[i]=0;
        b[i]=bt[i];
}
printf("\n\t\tGANTT CHART");
printf("\n      \n"); for(i=0;i<n;i++)
{
        if(max<bt[i])
                max=bt[i];
}
while(max!=0)
{
        for(i=0;i<n;i++)
        {
                if(bt[i]>0)
                {
                        if(ct==0)
                                wt[i]=wt[i]+cwt;
                        else
                                 wt[i]=wt[i]+(cwt-t[i]);
                }
                if(bt[i]==0)
                cwt=cwt+0;
                else if(bt[i]==max)
                {
                        if(bt[i]>tq)
                        {
                                cwt=cwt+tq;
                        bt[i]=bt[i]-tq;
                        max=max-tq;
                }
                else
                {
                        cwt=cwt+bt[i];
                        bt[i]=0;
                        max=0;
                }

                printf("|\t%s",p[i]);
                y[j]=cwt;
                j++;
        }
else if(bt[i]<tq)
{
        cwt=cwt+bt[i]; bt[i]=0;
        printf("|\t%s",p[i]);
        y[j]=cwt;
        j++;
}
else if(bt[i]>tq)
{
        cwt=cwt+tq;
        bt[i]=bt[i]-tq;
        printf("|\t%s",p[i]);
         y[j]=cwt;
        j++;
}
else if(bt[i]==tq)
{
        cwt=cwt+bt[i];
```

```
                    printf("|\t%s",p[i]); bt[i]=0;
                    y[j]=cwt; j++;
          }
          t[i]=cwt;
          }
     ct=ct+1;
     }
     for(i=0;i<n;i++)
     {
          wt[i]=wt[i]-at[i];
          a=a+wt[i];
          tt[i]=wt[i]+b[i]-at[i];
          s=s+tt[i];
     }
     a=a/n; s=s/n;
     printf("\n        ");
     printf("\n0");
     for(i=0;i<j;i++)
          printf("\t%d",y[i]);
     printf("\n");
     printf("\n        "); printf("\n\t\t ROUND ROBIN\n");
     printf("\n    Process    Burst-time    Arrival-time    Waiting-time    Turnaround-
     time\n");
     for(i=0;i<n;i++)
          printf("\n\n %d%s \t %d\t\t %d \t\t %d\t\t %d", i+1, p[i], b[i], at[i], wt[i], tt[i]);
     printf("\n\nAvg waiting time=%f",a);
     printf("\n\nAvgturn  around  time=%f",s);
}
```

OUTPUT:

```
enter the no of process:3
enter the time quantum2
enter the process name, bursttime,  arrival time
p1        2       0
p2        3       1
p3       4        2
```

GANTT CHART

```
|       p1|    p2|    p3|    p2|    p3
 0       2         4         6       7       9
```

ROUND ROBIN

```
Process    Burst-time    Arrival-time    Waiting-time    Turnaround-time
 p1              2              0               0                     2
p2               3              1               3                     5
p3               4              2               3                     5
Avg Waiting Time=2.000000 Avg
Turnaround  Time=4.000000
```

RESULT

The Round Robin scheduling algorithm has been implemented in C.

EX.NO.3D: IMPLEMENTATION OF PRIORITY SCHEDULING ALGORITHM

AIM:

To write a C program to implement Priority Scheduling algorithm.

DESCRIPTION:

For priority scheduling algorithm, read the number of processes/jobs in the system, their CPU burst times, and the priorities. Arrange all the jobs in order with respect to their priorities. There may be two jobs in queue with the same priority, and then FCFS approach is to be performed. Each process will be executed according to its priority. Calculate the waiting time and turnaround time of each of the processes accordingly**.**

ALGORITHM:

Step 1: Start the program.

Step 2: Get the input process and their burst time.

Step 3: Sort the processes based on priority.

Step 4: Compute the waiting time and turnaround time for each process. Step 5:

Calculate the average waiting time and average turnaround time. Step 6: Print

the details about all the processes.

Step 7: Stop the program.

PROGRAM:

```
#include<stdio.h>
#include<string.h>
void main()
{
        int bt[30],pr[30],np; intwt[30],tat[30],wat[30],at[30],ft[30]; int
        i,j,x,z,t;
        float sum1=0,sum=0,awt,att;
        char p[5][9],y[9];
        printf("\nenter the number of process");
        scanf("%d",&np);
        printf("\nEnter the process,burst-time and priority:");
```

```
 for(i=0;i<np;i++)
             scanf("%s%d%d",p[i],&bt[i],&pr[i]);
printf("\nEnter the arrival-time:");
for(i=0;i<np;i++)
            scanf("%d",&at[i]);
 for(i=0;i<np;i++)
            for(j=i+1;j<np;j++)
            {
                        if(pr[i]>pr[j])
                        {
                                    x=pr[j];
                                    pr[j]=pr[i];
                                    pr[i]=x;
                                    strcpy(y,p[j]);
                                    strcpy(p[j],p[i]);
                                    strcpy(p[i],y);
                                    z=bt[j]; b
                                    t[j]=bt[i];
                                    bt[i]=z;


                        }
            }
wt[0]=0;
for(i=1;i<=np;i++)
            wt[i]=wt[i-1]+bt[i-1];
ft[0]=bt[0];
for(i=1;i<np;i++)
             ft[i]=ft[i-1]+bt[i];
printf("\n\n\t\tGANTT CHART\n");
printf("\n        \n");
for(i=0;i<np;i++)
            printf("|\t%s\t",p[i]);
```

```
        printf("|\t\n");
        printf("\n       \n");
        printf("\n");
        for(i=0;i<=np;i++)
                printf("%d\t\t",wt[i]);
        printf("\n       \n");
        printf("\n");
         for(i=0;i<np;i++)
                wat[i]=wt[i]-at[i];
         for(i=0;i<np;i++)
                tat[i]=wat[i]-at[i];
        printf("\nPRIORITY SCHEDULING:\n");
        printf("\nProcess  Priority  Burst-time  Arrival-time  Waiting-time  Turnaround-
        time");
        for(i=0;i<np;i++)
                printf("\n\n%d%s\t%d\t\t%d\t\t%d\t%d\t\t%d",i+1,p[i],pr[i],bt[i],a  t[i],wt[i],tat[i]);
         for(i=0;i<np;i++)
                sum=sum+wat[i];
        awt=sum/np;
        for(i=0;i<np;i++)
                sum1=sum1+tat[i];
         att=sum1/np;
        printf("\n\nAverage  waiting  time:%f",awt);  printf("\n\nAverageturn  around   time
        is:%f",att);
}
```

OUTPUT:

```
Enter the number of process3
Enter the process, burst-time and priority: p1
3 3
p2 4 2
p3 5 1
```

```
Enter the arrival-time: 0 1 2
GANTT CHART
 _______________________________________
|      p3     |      p2     |      p1     |
 ________________________________________

0             5             9            12
```

PRIORITY SCHEDULING:

```
Process   Priority   Burst-time   Arrival-time   Waiting-time   Turnaround-time
p3           1           5             0              0               0
p2           2           4             1              5               3
p1           3           3             2              9               5
Average waiting time: 3.666667 Average
turnaround time is: 2.666667
```

RESULT

The Priority scheduling algorithm has been implemented in C.

EX.NO:4 PRODUCER CONSUMER PROBLEM USING SEMAPHORE

AIM:

To write a C program to implement the Producer & consumer Problem (Semaphore)

DESCRIPTION:

Producer-consumer problem, is a common paradigm for cooperating processes. A producer process produces information that is consumed by a consumer process. One solution to the producer-consumer problem uses shared memory. To allow producer and consumer processes to run concurrently, there must be available a buffer of items that can be filled by the producer and emptied by the consumer. This buffer will reside in a region of memory that is shared by the producer and consumer processes. A producer can produce one item while the consumer is consuming another item. The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.

ALGORITHM:

Step 1: The Semaphore mutex, full & empty are initialized.

Step 2: In the case of producer process

- i) Produce an item in to temporary variable.
- ii) If there is empty space in the buffer check the mutex value for enter into the critical section.
- iii) If the mutex value is 0, allow the producer to add value in the temporary variable to the buffer.

Step 3: In the case of consumer process

- i) It should wait if the buffer is empty
- ii) If there is any item in the buffer check for mutex value, if the mutex==0, remove item from buffer
- iii) Signal the mutex value and reduce the empty value by 1.
- iv) Consume the item.

Step 4: Print the result

PROGRAM :

```
#define  BUFFERSIZE  10
int mutex,n,empty,full=0,item,item1; int
 buffer[20];
int in=0,out=0,mutex=1;
void wait(int s)
{
        while(s<0)
        {
                printf(“\nCannot add an item\n”);
                 exit(0);
        }
        s--;
}

void signal(int s)
{
        s++;
}
void producer()
{
        do
        {
                wait (empty);
                wait(mutex);
                printf(“\nEnter an item:”);
                scanf(“%d”,&item);
                buffer[in]=item;
                in=in+1;
                signal(mutex);
                signal(full);
	}
	while(in<n);
}
void consumer()
{
	do
	{
		wait(full);
		wait(mutex);
		item1=buffer[out];
		printf(“\nConsumed item =%d”,item1); out=out+1;
		signal(mutex);
		signal(empty);
	}
	while(out<n);
}
void main()
{
	printf(“Enter the value of n:”);
	scanf(“%d “,&n);
	empty=n;
	while(in<n)
		producer();
	while(in!=out)
		 consumer();
}
```

OUTPUT:

```
$ cc prco.c
$ a.out
Enter the value of n :3
Enter the item:2
Enter the item:5
Enter the item:9
consumed item=2
consumed item=5
consumed item=9
$
```

RESULT:

Thus the program for solving producer and consumer problem using semaphore was executed successfully.

EX.NO: 5 **DEADLOCK AVOIDANCE**

AIM:

To Simulate Algorithm for Deadlock avoidance

DESCRIPTION:

In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a deadlock. Deadlock avoidance is one of the techniques for handling deadlocks. This approach requires that the operating system be given in advance additional information concerning which resources a process will request and use during its lifetime. With this additional knowledge, it can decide for each request whether or not the process should wait. To decide whether the current request can be satisfied or must be delayed, the system must consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process. Banker's algorithm is a deadlock avoidance algorithm that is applicable to a system with multiple instances of each resource type

ALGORITHM:

Step 1: Start the Program

Step 2: Get the values of resources and processes. Step
3: Get the avail value.

Step 4: After allocation find the need value.

Step 5: Check whether its possible to allocate. If possible it is safe state

Step 6: If the new request comes then check that the system is in safety or not if we allow the request.

Step 7: Stop the execution

10.Stop the program

PROGRAM:

```
 #include<stdio.h>
```

```
void main()
{
          int pno,rno,i,j,prc,count,t,total;
          count=0;
          clrscr();
           printf("\n Enter number of process:");
           scanf("%d",&pno);
           printf("\n Enter number of resources:");
          scanf("%d",&rno);
           for(i=1;i< =pno;i++)
           {
                      flag[i]=0;
          }
           printf("\n Enter total numbers of each resources:");
           for(i=1;i<= rno;i++)
                     scanf("%d",&tres[i]);
           printf("\n Enter Max resources for each process:"); for(i=1;i<=
          pno;i++)
           {
                      printf("\n for process %d:",i);
                      for(j=1;j<= rno;j++)
                               scanf("%d",&max[i][j]);
           }
           printf("\n Enter allocated resources for each process:");
          for(i=1;i<= pno;i++)
           {
                     printf("\n for process %d:",i);
                    for(j=1;j<= rno;j++)
                               scanf("%d",&allocated[i][j]);
          }
          printf("\n available resources:\n"); for(j=1;j<=
           rno;j++)
```

```
{
      avail[j]=0;
     total=0;
      for(i=1;i<= pno;i++)
     {
            total+=allocated[i][j];
      }
      avail[j]=tres[j]-total;
      work[j]=avail[j];
     printf("       %d \t",work[j]);
 }
 do
 {
      for(i=1;i<= pno;i++)
      {
           for(j=1;j<= rno;j++)
           {
                 need[i][j]=max[i][j]-allocated[i][j];
           }
      }
      printf("\n Allocated matrix            Max       need");
      for(i=1;i<= pno;i++)
      {
           printf("\n");
            for(j=1;j<= rno;j++)
           {
                printf("%4d",allocated[i][j]);
            }
           printf("|"); for(j=1;j<=
           rno;j++)
           {
                printf("%4d",max[i][j]);
```

```
                         }
                        printf("|");
                         for(j=1;j<= rno;j++)
                        {
                                    printf("%4d",need[i][j]);
                          }
            }
   prc=0;
  for(i=1;i<= pno;i++)
{
            if(flag[i]==0)
             {
                         prc=i;
                          for(j=1;j<= rno;j++)
                        {
                                      if(work[j]<  need[i][j])
                                    {
                                                prc=0;
                                                 break;
                                    }
                         }
             }
              if(prc!=0)
              break;
 }

 if(prc!=0)
{
            printf("\n Process %d completed",i);
            count++;
             printf("\n Available matrix:");
             for(j=1;j<= rno;j++)
```

```
                        {
                                  work[j]+=allocated[prc][j];
                                  allocated[prc][j]=0;
                                  max[prc][j]=0;
                                  flag[prc]=1;
                                   printf("    %d",work[j]);
                        }
             }
            }while(count!=pno&&prc!=0);
if(count==pno)
            printf("\nThe system is in a safe state!!");
 else
            printf("\nThe system is in an unsafe state!!");
getch();
}
```

OUTPUT

```
Enter number of process:5
Enter number of resources:3
Enter total numbers of each resources:10 5 7
Enter Max resources for each process:
 for process 1: 7 5 3
 for process 2: 3 2 2
 for process 3: 9 0 2
 for process 4: 2 2 2
 for process 5: 4 3 3
 Enter allocated resources for each process: for
 process 1: 0 1 0
 for process 2: 3 0 2
 for process 3: 3 0 2
 for process 4: 2 1 1
 for process 5: 0 0 2
```

```
available  resources:
    2       3       0



Allocated matrix          Max       need
  0   1   0|   7   5   3|   7   4   3
  3   0   2|   3   2   2|   0   2   0
  3   0   2|   9   0   2|   6   0   0
  2   1   1|   2   2   2|   0   1   1
  0   0   2|   4   3   3|   4   3   1
Process 2 completed
Available matrix:   5   3   2   Allocated
matrix                        Max       need
  0   1   0|   7   5   3|   7   4   3
  0   0   0|   0   0   0|   0   0   0
  3   0   2|   9   0   2|   6   0   0
  2   1   1|   2   2   2|   0   1   1
  0   0   2|   4   3   3|   4   3   1
Process 4 completed
Available matrix:   7   4   3   Allocated
matrix                        Max       need
  0   1   0|   7   5   3|   7   4   3
  0   0   0|   0   0   0|   0   0   0
  3   0   2|   9   0   2|   6   0   0
  0   0   0|   0   0   0|   0   0   0
  0   0   2|   4   3   3|   4   3   1
Process 1 completed
Available matrix:   7   5 3 Allocated
matrix                        Max       need
  0   0   0|   0   0   0|   0   0   0
  0   0   0|   0   0   0|   0   0   0
  3   0   2|   9   0   2|   6   0   0
  0   0   0|   0   0   0|   0    0   0
  0   0   2|   4   3   3|   4    3   1
 Process 3 completed
 Available matrix:   10       5  5
 Allocated matrix           Max       need
  0   0   O|   0   0   O|   0   0   0
  0   0   O|   0   0   O|   0   0   0
  0   0   O|   0   0   O|   0   0   0
  0   0   O|   0   0   O|   0   0   0
  0   0   2|   4   3   3|   4   3   1
 Process 5 completed
 Available matrix:   10     5  7
The system is in a safe state!!
```

RESULT:

Thus the program to implement the deadlock avoidance was executed and verified.

EX.NO:6: DEADLOCK DETECTION ALGORITHM

AIM:

To Simulate Algorithm for Deadlock detection

ALGORITHM:

Step 1: Start the Program

Step 2: Get the values of resources and processes. Step

3: Get the avail value..

Step 4: After allocation find the need value.

Step 5: Check whether its possible to allocate.

Step 6: If it is possible then the system is in safe state. Step

7: Stop the execution

PROGRAM

```
#include<stdio.h>
#include<conio.h> int
max[100][100]; i nt
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("********** Deadlock Detection Algo ************\n"); input();
        show();
        cal();
        getch();
        return 0;
}

void input()
{
        int i,j;
        printf("Enter the no of Processes\t");
         scanf("%d",&n);
        printf("Enter the no of resource 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]);
                }
        }
        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 Available\t");
        for(i=0;i<n;i++)
        {
                printf("\nP%d\t ",i+1);
                for(j=0;j<r;j++)
                {
                        printf("%d  ",alloc[i][j]);
                }
                printf("\t");
                for(j=0;j<r;j++)
                {
                        printf("%d  ",max[i][j]);
                }
                printf("\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 dead[100]; 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];
        }
}
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("\nP%d",i);
                                         if(finish[i]==1)
                                        {
                                                i=n;
                                        }
                                }
                        }
                }
        }
}
j=0;
flag=0;
for(i=0;i<n;i++)
{
        if(finish[i]==0)
        {
                dead[j]=i;
                 j++;
                flag=1;
        }
}
if(flag==1)
{
        printf("\n\nSystem is in Deadlock and the Deadlock process are\n"); for(i=0;i<n;i++)
        {
        }

}
else
        {       printf("P%d\t",dead[i]);

        }
}
```

OUTPUT:

```
Enter the no. Of processes 3
Enter the no of resources instances 3 Enter
the max matrix
3 6 8
4 3 3
```

```
3 4 4

Enter the allocation matrix 3
3 3
2 0 3
1 2 4
Enter the available resources 1
2 0
Process         Allocation          max          available
P1                   3 3 3          3 6 8         1 2 0
P2                   2 0 3          4 3 3
P3                   1 2 4          3 4 4
System is in deadlock and deadlock process are P1
P2  P3
```

RESULT:

Thus the program to implement the deadlock detection was executed successfully.

EX.NO: 7 THREADING

AIM:

To Write C program to implement Threading

ALGORITHM:

```
Step 1: Define a function func that takes a void* argument and returns a void* pointer.
Step 2: Inside the func function, detach the current thread using pthread_detach(pthread_self())
so that it can continue running independently of the parent thread.
Step 3: Print a message indicating that the function is running.
Step 4: Exit the thread using pthread_exit(NULL) so that it terminates.
Step 5: Define a function fun.
Step 6: Declare a pthread_t variable named ptid to store the ID of the new thread that will be
created.
Step 7: Inside the fun function, create a new thread using pthread_create(&ptid, NULL, &func,
NULL) with the func function as the thread function.
Step 8: Print a message indicating that the current line may be printed before the thread
terminates.
Step 9: Compare the ID of the current thread with the ID of the newly created thread using
pthread_equal(ptid, pthread_self()).
Step 10: If the IDs are equal, print a message indicating that the threads are equal.
Step 11: If the IDs are not equal, print a message indicating that the threads are not equal.
Step 12: Wait for the newly created thread to terminate using pthread_join(ptid, NULL).
Step 13: Print a message indicating that the current line will be printed after the thread ends.
Step 14: Exit the thread using pthread_exit(NULL).
```

PROGRAM:

```c
// C program to show thread functions


#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

void* func(void* arg)
{
  // detach the current thread
  // from the calling thread
  pthread_detach(pthread_self());

  printf("Inside the thread\n");

  // exit the current thread
  pthread_exit(NULL);
}

void fun()
{
  pthread_t ptid;
  // Creating a new thread
  pthread_create(&ptid, NULL, &func, NULL);
  printf("This line may be printed"
    " before thread terminates\n");

  // The following line terminates
  // the thread manually
  // pthread_cancel(ptid);

  // Compare the two threads created
  if(pthread_equal(ptid, pthread_self())
    printf("Threads are equal\n");
  else
    printf("Threads are not equal\n");

  // Waiting for the created thread to terminate
  pthread_join(ptid, NULL);

  printf("This line will be printed"
    " after thread ends\n");

  pthread_exit(NULL);
}
// Driver code
int main()
{
  fun();
  return 0;
}
```

OUTPUT:

The main thread prints:
This line may be printed before thread terminates

The new thread runs `func` and prints:
Inside the thread

The thread comparison prints:
Threads are not equal

RESULT:

Thus C program for implementing Threading has been executed successfully.

EX.NO:8 IMPLEMENTATION OF PAGING TECHNIQUE OF MEMORY MANAGEMENT

AIM:

To write a C program to implement paging concept for memory management.

ALGORIHTM:

Step 1: Start the program.

Step 2: Enter the logical memory address.

Step 3: Enter the page table which has offset and page frame. Step 4: The corresponding physical address can be calculate by, PA = [pageframe* No. of page size] + Page offset.

Step 5: Print the physical address for the corresponding logical address. Step 6: Terminate the program.

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
main()
{
        int ms, ps, nop, np, rempages, i, j, x, y, pa, offset;
        int s[10], fno[10][20];
        clrscr();
        printf("\nEnter the memory size -- ");
         scanf("%d",&ms);
        printf("\nEnter the page size -- ");
        scanf("%d",&ps);
        nop = ms/ps;
        printf("\nThe no. of pages available in memory are -- %d ",nop); printf("\nEnter
        number of processes -- ");
        scanf("%d",&np);
        rempages = nop;
        for(i=1;i<=np;i++)
```

```
	{

	printf("\nEnter no. of pages required for p[%d]-- ",i);
	scanf("%d",&s[i]);
	if(s[i] >rempages)
	{
		printf("\nMemory  is  Full");  break;
	}
	rempages = rempages - s[i];  printf("\nEnter
	pagetable for p[%d] --- ",i); for(j=0;j<s[i];j++)
		scanf("%d",&fno[i][j]);
	}

	printf("\nEnter  Logical  Address  to  find  Physical  Address  ");
	printf("\nEnter  process  no.  and  pagenumber  and  offset  --  ");
	scanf("%d %d %d",&x,&y, &offset);
	if(x>np || y>=s[i] || offset>=ps)
		printf("\nInvalid Process or Page Number or offset");
	else
	{
		pa=fno[x][y]*ps+offset;
		printf("\nThe Physical Address is -- %d",pa);
	}


	getch();
	}
```

OUTPUT

```
Enter the memory size – 1000
Enter  the  page  size  --          100
The no. of pages available in memory are            10
```

```
Enter number of processes --          3
Enter no. of pages required for p[1]--       4
Enter  pagetable  for  p[1]  ---           8         6          9          5
Enter no. of pages required for p[2]--       5
Enter  pagetable  for  p[2]  ---           1         4          5          7          3
Enter no. of pages required for p[3]--       5
Memory is Full
Enter Logical Address to find Physical Address Enter process no. and pagenumber and
offset -- 2
3
60
The Physical Address is -- 760
```

RESULT:

Thus C program for implementing paging concept for memory management has been executed successfully.

Ex.NO: 9 IMPLEMENTATION OF MEMORY ALLOCATION TECHNIQUES

AIM:

To write a C program to implement Memory Management concept using the technique best fit, worst fit and first fit algorithms.

ALGORITHM:

1. Get the number of process.
2. Get the number of blocks and size of process.
3. Get the choices from the user and call the corresponding switch cases.
4. First fit -allocate the process to the available free block match with the size of the process
5. Worst fit –allocate the process to the largest block size available in the list
6. Best fit-allocate the process to the optimum size block available in the list
7. Display the result with allocations

PROGRAM:

```
#include<stdio.h>
main()
{
        int p[10],np,b[10],nb,ch,c[10],d[10],alloc[10],flag[10],i,j;
        printf("\nEnter the no of process:");
        scanf("%d",&np);
        printf("\nEnter the no of blocks:");
        scanf("%d",&nb);
        printf("\nEnter the size of each process:");
        for(i=0;i<np;i++)
        {
                printf("\nProcess %d:",i);
                scanf("%d",&p[i]);
        }
        printf("\nEnter the block sizes:");
        for(j=0;j<nb;j++)
        {
                printf("\nBlock %d:",j);
                scanf("%d",&b[j]);c[j]=b[j];d[j]=b[j];
        }
        if(np<=nb)
        {
                printf("\n1.First fit 2.Best fit 3.Worst fit"); do
                {
                        printf("\nEnter your choice:");
                        scanf("%d",&ch);
```

```
switch(ch)
{
            case 1: printf("\nFirst Fit\n");
                   for(i=0;i<np;i++)
                 {
                          for(j=0;j<nb;j++)
                          {
                                  if(p[i]<=b[j])
                                  {
                                  alloc[j]=p[i];printf("\n\nAlloc[%d]",all
                                  oc[j]);
                                  printf("\n\nProcess %d of size %d is
                          allocated in block:%d of size:%d",i,p[i],j,b[j]);
                                  flag[i]=0,b[j]=0;break;
                                  }
                                  else
                                          flag[i]=1;
                          }
        }
        for(i=0;i<np;i++)
        {
                if(flag[i]!=0)
                printf("\n\nProcess %d of size %d is not
        allocated",i,p[i]);
        }
        break;

case 2: printf("\nBest Fit\n");
        for(i=0;i<nb;i++)
        {
                for(j=i+1;j<nb;j++)
                {
                        if(c[i]>c[j])
                        {
                                int temp=c[i];
                                c[i]=c[j];
                                c[j]=temp;
                        }
                }
        }
```

```
printf("\nAfter sorting block sizes:");
for(i=0;i<nb;i++)
printf("\nBlock %d:%d",i,c[i]);
for(i=0;i<np;i++)
{
        for(j=0;j<nb;j++)
        {
                if(p[i]<=c[j])
                {
                        alloc[j]=p[i];printf("\n\nAlloc[%d]",all
                        oc[j]);
                        printf("\n\nProcess %d of size %d is
                        allocated in block %d of size
                        %d",i,p[i],j,c[j]);
                }       flag[i]=0,c[j]=0;break;
                else

        }
                        flag[i]=1;
}

for(i=0;i<np;i++)
{
        if(flag[i]!=0)
        printf("\n\nProcess %d of size %d is not
}       allocated",i,p[i]);
break;

case 3: printf("\nWorst Fit\n");
        for(i=0;i<nb;i++)
        {
                for(j=i+1;j<nb;j++)
                {
                        if(d[i]<d[j])
                        {
                                int temp=d[i];
                                d[i]=d[j];
                                d[j]=temp;
                        }
        }
}
printf("\nAfter sorting block sizes:");
for(i=0;i<nb;i++)
printf("\nBlock %d:%d",i,d[i]);
for(i=0;i<np;i++)
{
        for(j=0;j<nb;j++)
        {
                if(p[i]<=d[j])
                {
                alloc[j]=p[i];
                printf("\n\nAlloc[%d]",alloc[j]);
                printf("\n\nProcess %d of size %d is
                allocated in block %d of size
                %d",i,p[i],j,d[j]

                flag[i]=0,d[j]=0;break;
                }
                else
                        flag[i]=1;
        }
}
for(i=0;i<np;i++)
{
        if(flag[i]!=0)
                        printf("\n\nProcess %d of size
                        %d is not allocated",i,p[i]);
}
break;

default:        printf("Invalid Choice…!");break;
```

```
                                        }
                    }while(ch<=3);
          }
}
```

OUTPUT

```
Enter the no of process:3
Enter the no of blocks:3
Enter the size of each process:
Process 0:100
Process 1:150
Process 2:200
Enter the block sizes:
Block 0:300
Block 1:350
Block 2:200
1.First fit 2.Best fit 3.Worst fit
Enter your choice:1
Alloc[100]
Process 0 of size 100 is allocated in block 0 of size 300
Alloc[150]
Process 1 of size 150 is allocated in block 1 of size 350
Alloc[200]
Process 2 of size 200 is allocated in block 2 of size 200 Enter
your choice:2
Best Fit
After sorting block sizes are:
Block 0:200
Block 1:300
Block 2:350
Alloc[100]
Process 0 of size 100 is allocated in block:0 of size:200
Alloc[150]
Process 1 of size 150 is allocated in block:1 of size:300
Alloc[200]
Process 2 of size 200 is allocated in block:2 of size:350 enter
your choice:3
```

```
Worst Fit
After sorting block sizes are:
Block 0:350
Block 1:300
Block 2:200
Alloc[100]
Process 0 of size 100 is allocated in block 0 of size 350
Alloc[150]
Process 1 of size 150 is allocated in block 1 of size 300
Alloc[200]
Process 2 of size 200 is allocated in block 2 of size 200 Enter
your choice:6
Invalid Choice…!
```

RESULT:

Thus a UNIX C program to implement memory management scheme using Best fit worst fit and first fit were executed successfully.

EX.NO:10 IMPLEMENTATION OF THE FIFO PAGE REPLACEMENT ALGORITHMS

AIM:

To write a UNIX C program to implement FIFO page replacement algorithm.

DESCRIPTION :

The FIFO Page Replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen . There is not strictly necessary to record the time when a page is brought in. By creating a FIFO queue to hold all pages in memory and by replacing the page at the head of the queue. When a page is brought into memory, insert it at the tail of the queue.

ALGORITHM:

1. Start the process
2. Declare the size with respect to page length
3. Check the need of replacement from the page to memory
4. Check the need of replacement from old page to new page in memory
5. Format queue to hold all pages
6. Insert the page require memory into the queue
7. Check for bad replacement and page fault
8. Get the number of processes to be inserted
9. Display the values
10. Stop the process

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
main()
{
        int i, j, k, f, pf=0, count=0, rs[25], m[10], n;
        clrscr();
        printf("\n Enter the length of reference string -- ");
        scanf("%d",&n);
        printf("\n Enter the reference string -- ");
        for(i=0;i<n;i++)
                scanf("%d",&rs[i]);  printf("\n
        Enter no. of frames -- ");
        scanf("%d",&f);
        for(i=0;i<f;i++)
                m[i]=-1;
        printf("\n The Page Replacement Process is -- \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("\n The number of Page Faults using FIFO are %d",pf);
         getch();
}
```

OUTPUT

```
Enter the length of reference string – 20
Enter the reference string -- 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1 Enter
no. of frames --                3
The Page Replacement Process is –
7       -1      -1      PF No. 1
7       0       -1      PF No. 2
7       0       1       PF No. 3
2       0       1       PF No. 4
2       0       1
2       3       1       PF No. 5
2       3       0       PF No. 6
4       3       0       PF No. 7
4       2       0       PF No. 8
4       2       3       PF No. 9
0       2       3       PF No. 10
0       2       3
0       2       3
0       1       3       PF No. 11
0       1       2       PF No. 12
0       1       2
0       1       2
7       1       2       PF No. 13
7       0       2       PF No. 14
7       0       1       PF No. 15
The number of Page Faults using FIFO are 15
```

RESULT:

Thus a UNIX C program to implement FIFO page replacement is executed successfully.

EX.NO:10B IMPLEMENTATION OF LRU PAGE REPLACEMENT ALGORITHM

AIM:

To write UNIX C program a program to implement LRU page replacement algorithm.

DESCRIPTION:

The Least Recently Used replacement policy chooses to replace the page which has not been referenced for the longest time. This policy assumes the recent past will approximate the immediate future. The operating system keeps track of when each page was referenced by recording the time of reference or by maintaining a stack of references.

ALGORITHM:

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

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
main()
{
        int i, j , k, min, rs[25], m[10], count[10], flag[25], n, f, pf=0, next=1;
         clrscr();
        printf("Enter the length of reference string -- ");
```

```
scanf("%d",&n);
printf("Enter the reference string -- ");
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("\nThe Page Replacement process is -- \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("\nThe number of page faults using LRU are %d",pf);
     getch();
}
```

OUTPUT

```
Enter the length of reference string -- 20
Enter the reference string -- 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1 Enter
the number of frames -- 3
The Page Replacement process is --
7       -1      -1      PF No. -- 1
7       0       -1      PF No. -- 2
7       0       1       PF No. -- 3
2       0       1       PF No. -- 4
```

```
2         0         1
2         0         3         PF No. -- 5
2         0         3
4         0         3         PF No. -- 6
4         0         2         PF No. -- 7
4         3         2         PF No. -- 8
0         3         2         PF No. -- 9
0         3         2
0         3         2
1         3         2         PF No. -- 10
1         3         2
1         0         2         PF No. -- 11
1         0         2
1         0         7         PF No. -- 12
1         0         7

1         0         7
The number of page faults using LRU are 12
```

RESULT:

Thus a UNIX C program to implement LRU page replacement is executed successfully.

EX.NO:10C IMPLEMENTATION OF LFU PAGE REPLACEMENT ALGORITHM

AIM:

To write a program in C to implement LFU page replacement algorithm.

ALGORITHM

Step1: Start the program

Step2: Declare the required variables and initialize it. Step3;

Get the frame size and reference string from the user Step4:

Keep track of entered data elements

Step5: Accommodate a new element look for the element that is not to be used in frequently replace.

Step6: Count the number of page fault and display the value Step7:

Terminate the program

PROGRAM

```
#include<stdio.h>
#include<conio.h>
main()
{
        int rs[50], i, j, k, m, f, cntr[20], a[20], min, pf=0; clrscr();
        printf("\nEnter number of page references -- ");
         scanf("%d",&m);
        printf("\nEnter the reference string -- ");
        for(i=0;i<m;i++)
                scanf("%d",&rs[i]);
        printf("\nEnter the available no. of frames -- ");
         scanf("%d",&f);
        for(i=0;i<f;i++)
        {
```

```
        cntr[i]=0;
        a[i]=-1;
    }
    Printf(“\nThe Page Replacement Process is – \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 -- %d",pf); getch();
}
```

OUTPUT

```
Enter number of page references -- 10
Enter the reference string --           1 2 3 4 5 2 5 2 5 1 4 3
Enter the available no. of frames              3
 The Page Replacement Process is –
1       -1      -1      PF No. 1
1       2       -1      PF No. 2
1       2       3       PF No. 3
4       2       3       PF No. 4
5       2       3       PF No. 5
5       2       3
5       2       3
5       2       1       PF No. 6
5       2       4       PF No. 7
5       2       3       PF No. 8
Total number of page faults---------8
```

RESULT:

Thus the C programs to implement LFU page replacement algorithm was executed successfully.

a. Additional-Reference bits algorithm

b. Second-chance algorithm

EX.NO:11 FILE ORGANIZATION TECHNIQUES

AIM:

To Write a C program to simulate the following file organization techniques

a) Single level directory b) Two level directory

DESCRIPTION

The directory structure is the organization of files into a hierarchy of folders. In a single-level directory system, all the files are placed in one directory. There is a root directory which has all files. It has a simple architectureand there are no sub directories. Advantage of single level directory system is that it is easy to find a file in thedirectory. In the two-level directory system, each user has own user file directory (UFD). The system maintains a master block that has one entry for each user. This master block contains the addresses of the directory of the users. When a user job starts or a user logs in, the system's master file directory (MFD) is searched. When a user refers to a particular file, only his own UFD is searched. This effectively solves the name collision problem and isolates users from one another

PROGRAM

1. SINGLE LEVEL DIRECTORY ORGANIZATION

```
#include<stdio.h>
struct
{
char dname[10],fname[10][10];
int fcnt;
}dir;
void main()
{
int i,ch;
char f[30];
clrscr();
dir.fcnt = 0;
printf("\nEnter name of directory -- ");
scanf("%s", dir.dname);
while(1)
{
printf("\n\n 1. Create File\t2. Delete File\t3. Search File \n 4. Display Files\t5. Exit\nEnter your choice -- ");
scanf("%d",&ch);
switch(ch)
{
case 1: printf("\n Enter the name of the file -- ");
scanf("%s",dir.fname[dir.fcnt]);
dir.fcnt++;
break;
case 2: printf("\n Enter the name of the file -- ");
scanf("%s",f);
for(i=0;i<dir.fcnt;i++)
{
if(strcmp(f, dir.fname[i])==0)
{
printf("File %s is deleted ",f);
strcpy(dir.fname[i],dir.fname[dir.fcnt-1]);
break;
}
}
if(i==dir.fcnt)
printf("File %s not found",f);
else
dir.fcnt--;
break;
case 3: printf("\n Enter the name of the file -- ");
scanf("%s",f);
for(i=0;i<dir.fcnt;i++)
{
if(strcmp(f, dir.fname[i])==0)
{
printf("File %s is found ", f);
break;
}
}
if(i==dir.fcnt)
printf("File %s not found",f);
break;
case 4: if(dir.fcnt==0)
printf("\n Directory Empty");
else
{
printf("\n The Files are -- ");
for(i=0;i<dir.fcnt;i++)
printf("\t%s",dir.fname[i]);
}
break;
default: exit(0);
}
}
getch();
}
```

OUTPUT:

```
Enter name of directory -- CSE

1. Create File 2. Delete File 3. Search File
4. Display Files 5. Exit Enter your choice – 1

Enter the name of the file -- A

1. Create File 2. Delete File 3. Search File
4. Display Files 5. Exit Enter your choice – 1

Enter the name of the file -- B

1. Create File 2. Delete File 3. Search File
4. Display Files 5. Exit Enter your choice – 1

Enter the name of the file -- C

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

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

The Files are -- A B C

1. Create File 2. Delete File 3. Search File
4. Display Files 5. Exit Enter your choice – 3

Enter the name of the file – ABC

File ABC not found
1. Create File 2. Delete File 3. Search File
4. Display Files 5. Exit Enter your choice – 2

Enter the name of the file – B
File B is deleted

1. Create File 2. Delete File 3. Search File
4. Display Files 5. Exit Enter your choice – 5
```

TWO LEVEL DIRECTORY ORGANIZATION

```c
#include<stdio.h>
struct
{
char dname[10],fname[10][10];
int fcnt;
}dir[10];
void main()
{
int i,ch,dcnt,k;
char f[30], d[30];
clrscr();
dcnt=0;
while(1)
{
printf("\n\n 1. Create Directory\t 2. Create File\t 3. Delete File");
printf("\n 4. Search File \t \t 5. Display \t 6. Exit \t Enter your choice -- ");
scanf("%d",&ch);
switch(ch)
{
case 1: printf("\n Enter name of directory -- ");
scanf("%s", dir[dcnt].dname);
dir[dcnt].fcnt=0;
dcnt++;
printf("Directory created");
break;
case 2: printf("\n Enter name of the directory -- ");
scanf("%s",d);
for(i=0;i<dcnt;i++)
if(strcmp(d,dir[i].dname)==0)
{
printf("Enter name of the file -- ");
scanf("%s",dir[i].fname[dir[i].fcnt]);
dir[i].fcnt++;
printf("File created");
break;
}
if(i==dcnt)
printf("Directory %s not found",d);
break;
case 3: printf("\nEnter name of the directory -- ");
scanf("%s",d);
for(i=0;i<dcnt;i++)
{
if(strcmp(d,dir[i].dname)==0)
{
printf("Enter name of the file -- ");
scanf("%s",f);
for(k=0;k<dir[i].fcnt;k++)
{
if(strcmp(f, dir[i].fname[k])==0)
{
printf("File %s is deleted ",f);
dir[i].fcnt--;
strcpy(dir[i].fname[k],dir[i].fname[dir[i].fcnt]);
goto jmp;
}
}
printf("File %s not found",f);
goto jmp;
}
}
printf("Directory %s not found",d);
jmp : break;
case 4: printf("\nEnter name of the directory -- ");
scanf("%s",d);
for(i=0;i<dcnt;i++)
{

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

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

<u>OUTPUT:</u>

```
1. Create Directory 2. Create File 3. Delete File
4. Search File 5. Display 6. Exit Enter your choice -- 1

Enter name of directory -- DIR1
Directory created


1. Create Directory 2. Create File 3. Delete File
4. Search File 5. Display 6. Exit Enter your choice -- 1

Enter name of directory -- DIR2

Directory created

1. Create Directory 2. Create File 3. Delete File
4. Search File 5. Display 6. Exit Enter your choice -- 2

Enter name of the directory – DIR1
Enter name of the file -- A1
File created

1. Create Directory 2. Create File 3. Delete File
4. Search File 5. Display 6. Exit Enter your choice -- 2

Enter name of the directory – DIR1
Enter name of the file -- A2
File created

1. Create Directory 2. Create File 3. Delete File
4. Search File 5. Display 6. Exit Enter your choice -- 2

Enter name of the directory – DIR2
Enter name of the file -- B1
File created

1. Create Directory 2. Create File 3. Delete File
```

```
4. Search File 5. Display 6. Exit Enter your choice -- 5
Directory Files
DIR1 A1 A2
DIR2 B1

1. Create Directory 2. Create File 3. Delete File
4. Search File 5. Display 6. Exit Enter your choice -- 4

Enter name of the directory – DIR
Directory not found

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

File A2 is deleted

1. Create Directory 2. Create File 3. Delete File
4. Search File 5. Display 6. Exit Enter your choice – 6
```

RESULT :

Thus the program to implement the Sequential file allocation was executed successfully.

EX.NO:12A SEQUENTIAL FILE ALLOCATION AIM:

To implement sequential file allocation technique.

ALGORITHM:

Step 1: Start the program. Step
2: Get the number of files.
Step 3: Get the memory requirement of each file.
Step 4: Allocate the required locations to each in sequential order.
a). Randomly select a location from available location s1= random(100); b). Check whether the required locations are free from the selected location. c). Allocate and set flag=1 to the allocated locations.
Step 5: Print the results fileno, length , Blocks allocated. Step 6: Stop the progr

PROGRAM

```
#include<stdio.h>
main()
{
        int f[50],i,st,j,len,c,k;
        clrscr();
        for(i=0;i<50;i++)
                f[i]=0;
        X:
        printf("\n Enter the starting block & length of file");
        scanf("%d%d",&st,&len);
        for(j=st;j<(st+len);j++)
                if(f[j]==0)
                {
                        f[j]=1;
                        printf("\n%d->%d",j,f[j]);
                }
                else
                {
                        printf("Block already allocated");
                         break;
                }
        if(j==(st+len))
                printf("\n the file is allocated to disk"); printf("\n
        if u want to enter more files?(y-1/n-0)");
        scanf("%d",&c);
         if(c==1)
                goto X;
         else
                exit();
getch();
}
```

OUTPUT

```
Output: Enter the starting block & length of file 4 10 4-
>1
5->1
6->1
7->1
8->1
9->1
10->1
11->1
12->1
13->1
The file is allocated to disk
If you want to enter more files? (Y-1/N-0)
```

RESULT :

Thus the program to implement the Sequential file allocation was executed successfully.

EX.NO:12B LINKED FILE ALLOCATION

AIM:

To write a C program to implement File Allocation concept using the technique Linked List Technique.

ALGORITHM:

```
Step 1: Start the Program
Step 2: Get the number of files.
Step 3: Allocate the required locations by selecting a location randomly Step 4:
Check whether the selected location is free.
Step 5: If the location is free allocate and set flag =1 to the allocated locations.
Step 6: Print the results file no, length, blocks allocated.
Step 7: Stop the execution
```

PROGRAM:

```
#include<stdio.h>
 main()
{
        int f[50],p,i,j,k,a,st,len,n,c;
        clrscr();
        for(i=0;i<50;i++)
                f[i]=0;
        printf("Enter how many blocks that are already allocated"); scanf("%d",&p);
        printf("\nEnter the blocks no.s that are already allocated"); for(i=0;i<p;i++)
        {
                scanf("%d",&a);
                f[a]=1;
        }
        X: printf("Enter the starting index block & length"); scanf("%d%d",&st,&len);
```

```
            k=len;
            for(j=st;j<(k+st);j++)
            {
                if(f[j]==0)
                 {
                                                          f[j]=1;
                        printf("\n%d->%d",j,f[j]);

                }
                else
                {
                        printf("\n %d->file is already allocated",j); k++;

                }
            }
printf("\n If u want to enter one more file? (yes-1/no-0)"); scanf("%d",&c);
            if(c==1)
                goto X;
             else
                exit();
    getch( );
    }
```

OUTPUT :

```
Enter how many blocks are already allocated 3 Enter the
blocks no's that are already allocated 4 7 9 Enter the starting
index block & length 3 7
    3-> 1
    4-> File is already allocated 5->1
    6->1
    7-> File is already allocated 8->1
    9-> File is already allocated 10->1
    11->1
    12->1
    If u want to enter one more file? (yes-1/no-0)
```

RESULT:

Thus the program to implement the linked file allocation was executed successfully

EX.NO:12C INDEXED FILE

ALLOCATION AIM:

To write a C program to implement file Allocation concept using the technique indexed allocation Technique

ALGORITHM:

Step 1: Start the Program

Step 2: Get the number of files.

Step 3: Get the memory requirement of each file.

Step 4: Allocate the required locations by selecting a location randomly. Step 5: Print the results file no,length, blocks allocated.

Step 6: Stop the execution.

PROGRAM

```
#include<stdio.h>
int f[50],i,k,j,inde[50],n,c,count=0,p;
 main()
{
        clrscr();
        for(i=0;i<50;i++)
                f[i]=0;
        x: printf("enter index block\t"); scanf("%d",&p);
        if(f[p]==0)
        {
                f[p]=1;
                printf("enter no of files on index\t"); s canf("%d",&n);


        }
        else
        {       printf("Block already allocated\n"); goto x;


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

```
            for(i=0;i<n;i++)
    if(f[inde[i]]==1)
    {
            printf("Block already allocated");
            goto x;
    }
    for(j=0;j<n;j++)
            f[inde[j]]=1; printf("\n
    allocated"); printf("\n file
    indexed"); for(k=0;k<n;k++)
            printf("\n %d->%d:%d",p,inde[k],f[inde[k]]);
    printf(" Enter 1 to enter more files and 0 to exit\t"); s
    scanf("%d",&c);
     if(c==1)
            goto x;
      else
            exit();
getch();
}
```

OUTPUT :

```
Enter index block 9
Enter no of files on index 3 1 2 3
Allocated
File indexed 9-> 1:1
9-> 2:1
9->3:1
Enter 1 to enter more files and 0 to exit.
```

RESULT : Thus the program to implement the indexed file allocation was executed successfully

EX.NO: 13 DISK SCHEDULING

ALGORITHMS AIM:

Write a C program to simulate disk scheduling algorithms

a) FCFS b) SCAN c) C-SCAN

DESCRIPTION

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

PROGRAM

FCFS DISK SCHEDULING ALGORITHM

```
#include<stdio.h>
 main()
{
        int t[20], n, I, j, tohm[20], tot=0;
        float avhm;
        clrscr();
        printf("enter the no.of tracks"); scanf("%d",&n);
        printf("enter the tracks to be traversed");
        for(i=2;i<n+2;i++)
```

```
                scanf(“%d”,&t*i+);
        for(i=1;i<n+1;i++)
        {
                tohm[i]=t[i+1]-t[i];
                if(tohm[i]<0)
                        tohm[i]=tohm[i]*(-1);
        }
        for(i=1;i<n+1;i++)
                tot+=tohm[i];
        avhm=(float)tot/n;
        printf(“Tracks traversed\tDifference between tracks\n”); for(i=1;i<n+1;i++)
                printf(“%d\t\t\t%d\n”,t*i+,tohm*i+);
         printf("\nAverage header movements:%f",avhm); getch();
}
```

OUTPUT

```
Enter no.of tracks:9
Enter track position:55     58     60     70     18     90     150    160    184
Tracks traversed        Difference between tracks
55                       45
58                      3
60                      2
70                      10
18                      52
90                      72
150                     60
160                     10
184                     24
Average header movements:30.888889
```

SCAN DISK SCHEDULING ALGORITHM

```
#include<stdio.h>
 main()
{
        int t[20], d[20], h, i, j, n, temp, k, atr[20], tot, p, sum=0;
        clrscr();
        printf("enter the no of tracks to be traveresed");
        scanf("%d'",&n);
        printf("enter the position of head");
         scanf("%d",&h);
        t[0]=0;t[1]=h;
        printf("enter the tracks");
        for(i=2;i<n+2;i++)
                scanf("%d",&t[i]);
        for(i=0;i<n+2;i++)
        {
                for(j=0;j<(n+2)-i-1;j++)
                {
                        if(t[j]>t[j+1])
                        {
                                temp=t[j];
                                t[j]=t[j+1];
                                t[j+1]=temp;
                        }
                }
        }
        for(i=0;i<n+2;i++)
                if(t[i]==h)
```

```
                    j=i;k=i;
        p=0;
        while(t[j]!=0)
        {
                atr[p]=t[j];
                 j--;
                p++;
        }
        atr[p]=t[j];
         for(p=k+1;p<n+2;p++,k++)
                atr[p]=t[k+1];
        for(j=0;j<n+1;j++)
        {
                if(atr[j]>atr[j+1])
                        d[j]=atr[j]-atr[j+1];
                else
                        d[j]=atr[j+1]-atr[j];
                sum+=d[j];
        }
        printf("\nAverage header movements:%f",(float)sum/n);
        getch();
}
```

OUTPUT

```
Enter no.of tracks:9
Enter track position:55     58     60     70     18     90     150     160     184
Tracks traversed     Difference between tracks
150     50
160     10
1841    24
90      94
```

```
70      20

60      10

58      2

55      3

18      37

Average header movements: 27.77
```

RESULT:

Thus the program to implement disk Scheduling algorithm has been executed and verified

CONTENT BEYOND SYLLABUS

EX.NO.14 DINING-PHILOSOPHERS

PROBLEM. AIM:

Write a C program to simulate the concept of Dining-Philosophers problem.

DESCRIPTION

The dining-philosophers problem is considered a classic synchronization problem because it is an example of a large class of concurrency-control problems. It is a simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner. Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five single chopsticks. When a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors). A philosopher may pick up only one chopstick at a time. Obviously, she cam1ot pick up a chopstick that is already in the hand of a neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks. When she is finished eating, she puts down both of her chopsticks and starts thinking again. The dining-philosophers problem may lead to a deadlock situation and hence some rules have to be framed to avoid the occurrence of deadlock.

PROGRAM

```
int tph, philname[20], status[20], howhung, hu[20], cho;
main()
{
        int i;
        clrscr();
        printf("\n\nDINING PHILOSOPHER PROBLEM");
        printf("\nEnter the total no. of philosophers: ");
        scanf("%d",&tph);
        for(i=0;i<tph;i++)
        {
                philname[i] = (i+1);
```

```
          status[i]=1;
}
printf("How many are hungry : ");
 scanf("%d", &howhung);
 if(howhung==tph)
{
          printf("\nAll are hungry..\nDead lock stage will occur");
          printf("\nExiting..");
}
else
{
          for(i=0;i<howhung;i++)
          {
                    printf("Enter philosopher %d position: ",(i+1));
                    scanf("%d", &hu[i]);
                    status[hu[i]]=2;
          }
          do
          {
          printf("1.One can eat at a time\t2.Two can eat at a time\t3.Exit\nEnter
          your choice:");
          scanf("%d", &cho);
          switch(cho)
          {
                     case  1:    one();
                                 break;
                     case 2:    two();
                                 break;
                      case 3: exit(0);
                    default:  printf("\nInvalid  option..");
```

```
                }
                }while(1);
        }
}
one()
{

        int pos=0, x, i;
        printf("\nAllow one philosopher to eat at any time\n"); for(i=0;i<howhung;
         i++, pos++)
        {
                printf("\nP %d is granted to eat", philname[hu[pos]]); for(x=pos;x<howhung;x++)
                        printf("\nP %d is waiting", philname[hu[x]]);

        }
}
two()
{

        int i, j, s=0, t, r, x;
        printf("\n Allow two philosophers to eat at same time\n"); for(i=0;i<howhung;i++)
        {
                for(j=i+1;j<howhung;j++)
                {
                        if(abs(hu[i]-hu[j])>=1&& abs(hu[i]-hu[j])!=4)
                        {
                                printf("\n\ncombination %d \n", (s+1));
                                t=hu[i];
                                r=hu[j];
                                 s++;
                                printf("\nP %d and P %d are granted to eat",
                                philname[hu[i]],philname[hu[j]]);
                                for(x=0;x<howhung;x++)
                                {
                                        if((hu[x]!=t)&&(hu[x]!=r))
                                                printf("\nP %d is waiting", philname[hu[x]]);
                                }
                        }
                }
        }
}
```

OUTPUT

```
DINING PHILOSOPHER PROBLEM
Enter the total no. of philosophers: 5 How
many are hungry : 3
Enter philosopher 1 position: 2
Enter philosopher 2 position: 4
Enter philosopher 3 position: 5
1.      One can eat at a time        2.Two can eat at a time      3.Exit
Enter your choice: 1
Allow one philosopher to eat at any time P 3 is granted to eat P 3
is waiting P 5 is waiting P 0 is waiting
P 5 is granted to eat P 5 is waiting P
0 is waiting
P 0 is granted to eat P 0 is waiting
 1.One can eat at a time       2.Two can eat at a time       3.Exit Enter your choice: 2
Allow two philosophers to eat at same time combination 1
P 3 and P 5 are granted to eat P 0 is waiting
combination 2
P 3 and P 0 are granted to eat P 5 is waiting
combination 3
P 5 and P 0 are granted to eat P 3 is waiting
```

```
1.One can eat at a time      2.Two can eat at a time      3.Exit
 Enter your choice: 3
```

RESULT:

Thus the program to implement the dining Philosopher was executed and verified.