SRI KRISHNA COLLEGE OF TECHNOLOGY

(AN AUTONOMOUS INSTITUTION)

Affiliated to Anna University and Approved by AICTE
Accredited by NBA - AICTE and NAAC - UGC

KOVAIPUDUR, COIMBATORE-42.

23CY302

OPERATING SYSTEMSLAB MANUAL

Prepared By   Verified By

LIST OF EXPERIMENTS

Ex.No	EXPERIMENTS NAME
1.	Basic commands in Linux
2.	Shell Programming
3.	Programs using UNIX system calls
4	CPU Scheduling Algorithms
4.1	FCFS Algorithm
4.2	SJF Algorithm
4.3	SRTF Algorithm
4.4	Priority Scheduling Algorithm
4.5	Round Robin Algorithm
5.1	Producer – Consumer Algorithm using Semaphores
5.2	Dining philosopher's algorithm to demonstrate Process Synchronization.
6	Banker's algorithm for Deadlock Avoidance
7	Memory Allocation &Management Algorithms
7.1	First fit Algorithm
7.2	Next fit Algorithm
7.3	Best fit Algorithm
7.4	Worst fit Algorithm
8	Page Replacement Techniques.
8.1	FIFO page replacement Algorithm
8.2	LRU page replacement Algorithm
8.3	Optimal Page Replacement Algorithm
9	Disk Scheduling Algorithm
9.1	FCFS scheduling algorithm
9.2	SSTF (shortest seek time first) algorithm
9.3	SCAN scheduling
9.4	C-SCAN scheduling
9.5	LOOK Scheduling
9.6	C-LOOK scheduling
10	Implementation of File Organization Techniques
10.1	Contiguous Allocation Algorithm
10.2	Linked Allocation Algorithm
10.3	Indexed Allocation Algorithm

Ex : 1 **BASIC COMMANDS IN LINUX**

Objectives:

To study of general purpose Linux Commands.

Learning Outcomes:

After the completion of this experiment, student will be able to

- Understand the basic use of the Linux Command.
- Be able to use basic linux commands including man, ls, cd, cp, rm, pwd, and mkdir

GENERAL PURPOSE COMMANDS

1. WHO COMMAND:

DESCRIPTION:

This command is used to get information about all the users who are currently logged into the system.

SYNTAX:

```
$who
```

OUTPUT:

```
iicsea55 pts/12       2009-02-19 10:10 (192.168.1.66)
iicsea58 pts/8        2009-02-19 10:10 (192.168.1.49)
```

2. DATE COMMAND:

DESCRIPTION:

This command tells the Linux system to display the current Date Month Year Time.

SYNTAX:

```
$date
```

OUTPUT:

```
Thu Feb 19 10:43:29 IST 2009
```

3. CALENDAR COMMAND:

DESCRIPTION:

This command tells the Linux system to display the calendar for current month.

SYNTAX:

```
$cal
```

OUTPUT:

```
   February 2009
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
```

4. CALENDAR OPTION COMMAND:

DESCRIPTION:

This command tells the Linux system to display the calendar for current year.

SYNTAX:

$cal (year)

OUTPUT:

```
2000
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                                1 2  3  4  5                 1  2   3  4
2  3   4   5   6   7   8   6  7   8   9 10 11 12    5   6  7   8  9  10 11
9 10 11 12 13 14 15       13 14 15 16 17 18 19      12 13 14 15 16 17 18
16 17 18 19 20 21 22       20 21 22 23 24 25 26      19 20 21 22 23 24 25
23 24 25 26 27 28 29       27 28 29                  26 27 28 29 30 31
30 31
```
….

5. ECHO COMMAND:

DESCRIPTION:

This command is used to display the sentence or string given by us.

SYNTAX:

$echo (string)

OUTPUT:

```
God is Great
```

6. MAN COMMAND:

DESCRIPTION:

This command is used to get the complete details for given command.

SYNTAX:

$man (command)

OUTPUT:

```
CAL(1)              BSD General Commands Manual              CAL(1)
NAME
    cal - displays a calendar
SYNOPSIS
    cal [-smjy13] [[month] year]
DESCRIPTION
    Cal displays a simple calendar. If arguments are not specified, the cur-
    rent month is displayed.  The options are as follows:
```

7. CLEAR COMMAND:

DESCRIPTION:
This command is used to clear the screen.

SYNTAX:

```
$clear
```

OUTPUT:
The screen is cleared.

8. WHO AM I COMMAND:
This command gives information about current users

SYTNAX:

```
$who am I
```

OUTPUT:

```
Student         pts/1      feb 25 14:28(192.168.0.1555)
```

9. PWD COMMAND:

DESCRIPTION:
This command is used to show the current working directory

SYTNAX:

```
$pwd
```

OUTPUT:

```
/home/student
```

10. BC COMMAND:

DESCRIPTION:
This command is used to perform basic calculation

SYTNAX:

```
$bc
```

OUTPUT:

```
25+65
90
```

11. LIST COMMAND

DESCRIPTION:
This command is used to give the list of files in a particular directory.

SYTNAX:

```
$ls (dir name)
```

OUTPUT:

File1 file2 today

12. COMMAND: TEE

DESCRIPTION:

It is used to read the contents from standard input or from output of another command and reproduces the output to both in standard output and direct into output to one or more files.

SYNTAX: tee [options] <file name >

OUTPUT :

```
$  date | tee dat.txt
```

13.COMMAND: UNAME

DESCRIPTION:

It is used to display the details about the OS in which we are working.- n for node name

SYNTAX:

```
uname [options]
```

OUTPUT:

```
$ uname –n
administrator –Lenovo-$s510
```

<u>DIRECTORY MANIPULATION COMMAND</u>

1. MAKEDIR COMMAND

DESCRIPTION:

This command is used to create a new directory

SYTNAX:

```
$mkdir (dir)
```

OUTPUT

Required directory is created.

2. RMDIR COMMAND

DESCRIPTION:

This command is used for deleting a directory.

OPTIONS:

-i	require a conformation for deletion
-r	delete a directory without conformation

SYTNAX:

```
$rmdir (option) (dir)
```

OUTPUT:

Given directory is removed.

3. CD COMMAND:

DESCRIPTION:

This command is used to change the directory .

SYTNAX:

```
$cd (dir)
```

OUTPUT:

```
[iicsea19@localhost tariq ~]$
 Current working directory is changed.
```

<u>FILE MANIPULATION COMMANDS</u>

1. CAT COMMAND:

DESCRIPTION:

This command is used for creating and displaying the content of the file.

SYNTAX:

```
1.create: $cat>(file name)
2.display: $cat (file name)
```

OUTPUT:

```
[student@cse ~]$ cat>India
I love my India.
```

2. MORE COMMAND:

DESCRIPTION:

This command is used to display the content of the file one screen at a time

SYNTAX:

```
$more filename
```

OUTPUT:

```
$ more Vicky
Operation is performed
```

3. WC COMMAND:

DESCRIPTION:

This will print number of lines, words, characters in a file.

SYNTAX:

```
$ wc (file name)
```

OPTIONS:

l -   line

w - word

c –character

OUTPUT:

```
      $ wc –l Vicky
            2 vicky
     $ wc –w Vicky
             8 vicky
     $ wc –w Vicky
            49 vicky
```

4. MV COMMAND:

DESCRIPTION:

The command is used to move a file within a directory with different file name

SYNTAX:

$ mv (file name1) (file name2)

OUTPUT:

```
       $mv raja tar
       The file is renamed.
```

5. SPELL COMMAND:

DESCRIPTION:

It is used to find spelling errors in a file and display them as standard output in alphabetical order

SYNTAX:

$spell (file name)

OUTPUT:

```
$spell raja
cd  ppell.
```

<u>FILE COMPARISON COMMANDS</u>

1. COPY COMMAND:

DESCRIPTION:

Used to copy one or more files.

SYNTAX:

$cp (file name 1) (file name 2)

OUTPUT:

```
             $cp raja vicky
   The same file is copied with different name
```

2. CMP COMMAND:

DESCRIPTION:

This command is used to compare two file and generate the first difference.

SYNTAX:

```
$cmp (file1) (file2)
```

OUTPUT:

```
$cmp s sad
It there
```

3. COMM COMMAND:

DESCRIPTION:

This is used for comparing 2 files line by line & generates the common items.

SYNTAX:

```
$comm (file name 1) (file name 2)
```

OUTPUT:

```
$ comm S3 s5
Vicky          Antonio bendaras
```

<u>FILTER COMMANDS</u>

1. HEAD COMMAND

DESCRIPTION:

This command is used to display the first ten lines of the file without option.

SYNTAX:

```
$head (filename)
```

OUTPUT:

First ten lines of the file are displayed.

2. TAIL COMMAND

DESCRIPTION:

This command is used to display the last ten lines of the file without option.

SYNTAX:

```
$tail (file name)
```

OUTPUT:

Last ten lines of the file are displayed.

3. PASTE COMMAND

DESCRIPTION:

This command is used to paste one or more files as vertically and displays the result in standard output.

SYNTAX:

```
$paste (file1) (file2)
```

OUTPUT:

Hi I am Tariq Hi My Name is Tariq

FILE PERMISSION COMMANDS

1. CHMOD COMMAND

DESCRIPTION:

This is used to alter the access permission for owner, group, user or others.

SYNTAX:

```
$chmod (option) filename
```

OPTION:

4- read

2- write

1- execute

0- no permission

OUTPUT:

```
[student@localhost~] chmod 4 sam
---r---w---x---sam
```

COMMUNICATION COMMANDS

1. MAIL COMMAND:

DESCRIPTION:

This command is an interactive message processing system in which you send and receive electronically.

SYNTAX:

```
$ mail (user name)
```

OUTPUT:

```
Subject: hai !!! how are you …
```

2. WRITE COMMAND:

DESCRIPTION:

This command is used to communicate with the users who are currently working the operating system.

SYNTAX:

```
$ write user name
```

OUTPUT:

```
Message from student @ local host local domain on pts 10 at 12:04
```

3.WALL COMMAND:

DESCRIPTION:

This command used to broadcast by a message by super user to the entire user currently logged in.

SYNTAX:

```
$ wall message
```

OUTPUT:

```
Broad cast message from student (pts/0) ( thu feb 26 12:06:30:2009)
```

4.MESSAGE COMMAND:

DESCRIPTION:

This command is used to permit or deny message coming from the other user write to talk commands.

SYNTAX:

```
$mesg y/n
```

OUTPUT:

```
Message is accepted.
```

5. COMMAND:SSH

DESCRIPTION:

Program for logging into a remote machine and for executing commands on a remote machine.

SYNTAX:

```
ssh [options] [user]@hostname
```

EXAMPLE:

```
ssh ¬X guest@10.105.11.20
```

6. COMMAND:SCP

DESCRIPTION:

Secure copy (remote file copy program) -copies files between hosts on a network

SYNTAX:

```
 scp [options] [[user]@host1:file1] [[user]@host2:file2]
```

EXAMPLE:

```
scp file1.txt guest@10.105.11.20:~/Desktop/
```

<u>ARCHIVAL FILE COMMANDS</u>

Command : tar

Purpose : to archive a file

Syntax:tar [OPTION] DEST SOURCE

Example : tar ¬cvf /home/archive.tar /home/original

tar ¬xvf /home/archive.tar

Command : Zip
Purpose : package and compress (archive) files
Syntax:zip [OPTION] DEST SOURSE
Example : zip original.zip original

Command : unzip
Purpose : list, test and extract compressed files in a ZIP archive
Syntax:unzip filename
Example : unzip original.zip

Command : fdisk
Purpose : partition manipulator
Syntax:sudo fdisk name
Example : sudo fdisk ¬l

Command : mount
Purpose : mount a file system
Syntax:mount ¬t type device dir
Example : mount /dev/sda5 /media/target

Command : Umount
Purpose : unmount file systems
Syntax:umount [OPTIONS] dir | device...
Example : umount /media/target

Command : du
Purpose : estimate file space usage
Syntax: df [OPTION]... [FILE]...
Example : df

Command : Quota
Purpose : display disk usage and limits
Syntax:quota [OPTION]
Example : quota –v

NETWORK COMMANDS

1.ping

DESCRIPTION:

It sends packets of information to the user-defined source. If the packets are received, the destination device sends packets back. Ping can be used for two purposes

1. To ensure that a network connection can be established.
2. Timing information as to the speed of the connection.

Syntax:ping link name
Example : pink website address

2. ifconfig

DESCRIPTION:

View network configuration, it displays the current network adapter configuration. It is handy to determine if you are getting transmit (TX) or receive (RX) errors.

syntax : ifconfig
Example : ifconfig

3. netstat

DESCRIPTION:

Most useful and very versatile for finding a connection to and from the host. You can find out all the multicast groups (network) subscribed by this host by issuing
Syntax:netstat -g
Example : netstat-t,netstat-u

4. telnet
DESCRIPTION:

Connects destination host via the telnet protocol, if telnet connection establishes on any port means connectivity between two hosts is working fine.

Syntax:telnet hostname port
Example : telnet hostname port

5 Quota
DESCRIPTION:

Display disk usage and limits

Syntax:quota [OPTION]
Example : quota -v

PROCESS COMMANDS

1. PS COMMAND:

DESCRIPTION:

This Command is used to tell the detail about actual process running in the system.

SYNTAX:

$ps (option)

OPTIONS:

1. –f

OUTPUT:

```
     UID      PID    PPID   C    STIME     TIMECMD
  Student     2834   2831   0    10:13     00:00:00 bash
  Student     2884   2871   0    10:13     00:00:00 bash
```

2. SLEEP COMMAND:

DESCRIPTION:

This command is used to suspend the execution of the system for an individual time

SYNTAX:

$sleep (time)

OUTPUT:

```
$ sleep 5
Operation is performed
```

3.KILL COMMAND:

DESCRIPTION:

This is used to terminate the process.

SYNTAX:

$kill (processed)

OUTPUT:

```
      $ kill 89
   Operation is performed.
```

4.AT COMMAND:

DESCRIPTION:

It is used to schedule the jobs for later execution at the specified time.

SYNTAX:

$at (time)

OUTPUT:

```
$ at 10:00
Job 1 at FRI mar 06 10:00:00 2009
```

Viva Questions

1. I want to create a directory such that anyone in the group can create a file and access any person's file in it but none should be able to delete a file other than the one created by himself.
2. Given a file find the count of lines containing word "ABC".
3. How will you find the total disk space used by a specific user?
4. Write a command sequence to find all the files modified in less than 2 days and print the record count of each.
5. How can we find the process name from its process id?
6. What is CLI & GUI?
7. What is the pwd command?
8. What are the kinds of permissions under Linux?
9. What are the different modes when using vi editor?
10. How do you terminate an ongoing process?

RESULT:

Thus the Basic Linux commands are Executed and the output is obtained successfully**.**

Ex : 2 SHELL PROGRAMMING

Objectives:

To implement Shell programming using Command syntax, Substitutions, Expansion, Simple functions, Patterns and Loops.

Learning Outcomes:

After the completion of the experiment, Student will be able to

- Understand the basic Commands in shell.
- Write simple functions, loops, Patterns, Expansions, Substitutions.

Problem Statement:

The program in shell scripting language to find,

- Greatest of three numbers
- Factorial of N Numbers.
- Sum of N numbers.
- Odd or Even Number.
- Fibonacci Series
- Multiplication Table.
- Swapping of Two Numbers.
- File manipulation.
- Palindrome or not
- Positive or negative number.
- Prime number or not.
- Area of different shapes.

Algorithm:

Greatest of three numbers:

- Start the program.
- Enter any three numbers.
- It will check the condition $a -gt $b -a $a -gt $c.
- If a is greater than print a is greater.
- If b is not greater then compare b and c.
- If b is greater than c print b is greater.
- Else print c is greater.
- Stop the program.
- Execute the program.

Execution of the Program:

$ sh filename.sh// Executing the program

Sample Coding:

// Greatest among 3 numbers

```
echo Enter 3 numbers with spaces in between

read a b c
l=$a
if [ $b -gt $l ]
then
l=$b
fi
if [ $c -gt $l ]
then
l=$c
fi
echo Largest of $a $b $c is $l
```

Sample Output:

```
Enter 3 numbers with spaces in between

3 8 5

Largest of 3 8 5 is 8
```

Test cases:

Enter Numbers in different ranges.
Include choices for executing the many concepts in one program

Sample Coding:

// Factorial

```
echo "enter the number"
read n
fact=1
i=1
while [ $i -le $n ]
do
fact=`expr $i \* $fact`
i=`expr $i + 1`
done
echo "the factorial number of $ni is $fact
```

Sample Output:

```
Enter the number :
 4
The factorial of 4 is 24.
```

Pre lab Questions:

1. How do you find out what's your shell? - echo $SHELL
2. How do you refer to the arguments passed to a shell script? - $1, $2 and so on. $0 is your script name.
3. What's the conditional statement in shell scripting? - if {condition} then … fi
4. What is the use of break command?
5. What is the use of continue command in shell scripting?
6. Tell me the Syntax of "Case statement" in Linux shell scripting?
7. What is the basic syntax of while loop in shell scripting?
8. How to make a shell script executable?
9. What is the use of "#!/bin/bash" ?
10. What is the syntax of for loop in shell script?
11. How to debug a shell script?
12. How compare the strings in shell script?
13. What are the Special Variables set by Bourne shell for command line arguments?
14. How to test files in a shell script?
15. How to perform arithmetic operation?
16. Write the Basic Syntax of do-while statement?
17. How to define functions in shell scripting?
18. How to use bc (bash calculator) in a shell script?

RESULT:

Thus the implementation of shell program using Command syntax, Substitutions, Expansion, Simple functions, Patterns and Loops was implemented successfully.

EX.NO:3 IMPLEMENTATION OF UNIX SYSTEM CALLS.

Objectives:

To write a program for implementing Process system calls, Input- Output system calls of UNIX operating system:
fork(),exec(),getpid(),exit(),wait(), Open(), Create(), Read(), Write().

Learning Outcomes:

After the completion of this experiment, student will be able to

1. To familiarize you with the process system calls, input output system calls in unix.
2. Create a new process called child process
3. Both the child and parent continue to execute the instructions following fork call.
4. The child can start execution before the parent or vice-versa.

Problem Statement:

A program to implement the following system calls

- fork()
- exec()
- wait()
- exit()
- Open()
- Create().
- Read()
- Write()

Algorithm:

Fork()

```
1. Declare a variable x to be shared by both child and parent.
2. Create a child process using fork system call.
3. If return value is -1 then
    a. Print "Process creation unsuccessful"
    b. Terminate using exit system call.
4. If return value is 0 then
    a. Print "Child process"
    b. Print process id of the child using getpid system call
    c. Print value of x
    d. Print process id of the parent using getppid system call
5. Otherwise
    a. Print "Parent process"
    b. Print process id of the parent using getpid system call
    c. Print value of x
    d. Print process id of the shell using getppid system call.
6. Stop
```

Create()

```
1. Declare a character buffer buf to store 100 bytes.
2. Get the new filename as command line argument.
3.Create a file with the given name using open system call with O_CREAT and
  O_TRUNC options.
```

```
4. Check the file descriptor.
        a) If file creation is unsuccessful, then stop.
5. Get input from the console until user types Ctrl+D
        a) Read 100 bytes (max.) from console and store onto buf using read system call
        b) Write length of buf onto file using write system call.
6. Close the file using close system call.
7. Stop
```

Read():

```
1. Declare a character buffer buf to store 100 bytes.
2. Get existing filename as command line argument.
3. pen the file for reading using open system call with O_RDONLY option.
4. Check the file descriptor.
        a) If file does not exist, then stop.
5. Read until end-of-file using read system call.
        a) Read 100 bytes (max.) from file and print it
6. Close the file using close system call.
7. Stop
```

Write():

```
1. Declare a character buffer buf to store 100 bytes.
2. Get exisiting filename as command line argument.
3. Create a file with the given name using open system call with O_APPEND option.
4. Check the file descriptor.
        a) If value is negative, then stop.
5. Get input from the console until user types Ctrl+D
        a) Read 100 bytes (max.) from console and store onto buf using read system call
        b) Write length of buf onto file using write system call.
6. Close the file using close system call.
7. Stop
```

Sample Program :

Create()

```
fd = open(argv[1], O_WRONLY|O_CREAT|O_TRUNC,
0644); if(fd < 0)
{
printf("File creation
problem\n"); exit(-1);
}
printf("Press Ctrl+D at end in a new line:\n");
while((n = read(0, buf, sizeof(buf))) > 0)
{
len = strlen(buf);
write(fd, buf, len);
}
Close(fd);
}
```

Read()

```
fd = open(argv[1],
O_RDONLY); if(fd == -1)
```

```
{
printf("%s file does not exist\n",
argv[1]);
exit(-1);
}
printf("Contents of the file %s is : \n",argv[1]);
while(read(fd, buf, sizeof(buf)) > 0)
printf("%s", buf);
close(fd);
}
```

Write()

```
fd = open(argv[1], O_APPEND|O_WRONLY|O_CREAT,0644);
if (fd < 0)
{
perror(argv[1]);
exit(-1);
}
while((n = read(0, buf, sizeof(buf))) > 0)
{
len = strlen(buf);
write(fd, buf, len);
}
close(fd);
}
```

Execution of the Program:

```
$ cc fork.c      //Compiling the Program
$ ./a.out        //Executing the program
```

Sample Coding:

```
/* C program to illustrate use of fork () & exec () system call */

#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/wait.h>

int main()
{
  pid_t  pid;
  int ret = 1;
  int status;
  pid = fork();

  if (pid == -1)  // pid == -1 means error occured
  {
    printf("can't fork, error occured\n");
    exit(EXIT_FAILURE);
  }
  else if (pid == 0)   // pid == 0 means child process created
{
       printf("child process, pid = %u\n",getpid());
       printf("parent of child process, pid = %u\n",getppid());

      char * argv_list[] = {“./program”,NULL};
      execv(argv_list[0],argv_list);
     exit(0);
  }
  else    // a positive number is returned for the pid of parent process
{
     printf("Parent Of parent process, pid = %u\n",getppid());
     printf("parent process, pid = %u\n",getpid());

       if (waitpid(pid, &status, 0) > 0)
  {

           if (WIFEXITED(status) && !WEXITSTATUS(status))
             printf("program execution successful\n");

           else if (WIFEXITED(status) && WEXITSTATUS(status))
 {
               if (WEXITSTATUS(status) == 127)
                 {
                              printf("execv failed\n");
               }
               else
                   printf("program terminated normally,"
                      " but returned a non-zero status\n");
           }
           else
              printf("program didn't terminate normally\n");
       }
       else {
          // waitpid() failed
          printf("waitpid() failed\n");
       }
     exit(0);
  }
  return 0;
}
```

Program1.c

```
#include<stdio.h>
Int main()
{
Printf(“\n I am SKCT”);
Return 0;
}
```

Execution : gcc program1.c –o program

Pre-lab Questions:

1. What is the purpose of system calls?

2. When a process creates a new process using the fork() operation, which of the following state is shared between the parent process and the child process?

a. Stack   b. Heap   c. Shared memory segments

3. State the types of system calls.

4. What is the purpose of fork(),exit(),wait() system calls.

5. List out unix system calls for file manipulation.

6. List out system calls for Device manipulation.

7. List out system calls for Communication

8. What is system calls Parameters?

9. What are the differences among a system call, a library function, and a UNIX command?

10. What are the differences among a system call, a library function, and a UNIX command?

11. List out the different input output system calls.

12. What is the purpose of I/O system calls?

13. Is it possible to see information about a process while it is being executed?

14. What is the standard convention being followed when naming files in UNIX?

15. What is pid?

16. Is it possible to access a file that does not exist?

17. What will happen if u apply read() system call for a file that has no contents?

18. Is there any restricted or least important I/O system call?

Conclusion:

Thus the C program to implement process management using the following system calls of UNIX operating system: fork(),exec(), getpid, exit, wait, open(). read(),write(),was done and the output was verified.

EX.NO 4 IMPLEMENTATION OF CPU SCHEDULING ALGORITHMS

4.1 FCFS ALGORITHM

Objectives

To implement First Come First serve Scheduling Algorithm and display Gantt chart, turnaround time and waiting time in C.

Learning Outcomes:

After the completion of this experiment, student will be able to

- Schedule the processes or jobs to the CPU
- Will be able to create Gantt Chart and Calculate the Average waiting time and turn around time

Problem Statement:

The program to perform scheduling for the processes using first come first serve methods to display the following

1. Gantt chart
2. Average waiting time
3. Average turnaround time
4. Processes in the Queue

Algorithm:

```
1.  Create the number of process.
2.  Get the ID and Service time for each process.
3.  Initially, waiting time of first process is zero and Total time for the first process
    is the starting time of that process.
4.  Calculate the Total time and Processing time for the remaining processes.
5.  Waiting time of one process is the Total time of the previous process.
6.  Total time of process is calculated by adding Waiting time and Service time.
7.  Total waiting time is calculated by adding the waiting time for lack process.
8.  Total turnaround time is calculated by adding all total time of each process.
9.  Calculate Average waiting time by dividing the total waiting time by total
    number of process.
10. Calculate Average turnaround time by dividing the total time by the number of process.
11. Display the result.
```

Execution of the Program:

```
$ cc pgm1.c   //Compiling the Program

$ ./a.out         //Executing the program
```

Sample Coding:

```
#include<stdio.h>
int main()
{
    int AT[10],BT[10],WT[10],TT[10],n;
    int burst=0,cmpl_T;
    float Avg_WT,Avg_TT,Total=0;
    printf("Enter number of the process\n");
    scanf("%d",&n);
    printf("Enter Arrival time and Burst time of the process\n");
    printf("AT\tBT\n");
    for(int i=0;i<n;i++)
    {
        scanf("%d%d",&AT[i],&BT[i]);
    }

    // Logic for calculating Waiting time
    for(int i=0;i<n;i++)
    {
        if(i==0)
            WT[i]=AT[i];
        else
            WT[i]=burst-AT[i];
        burst+=BT[i];
        Total+=WT[i];
    }
    Avg_WT=Total/n;

    // Logic for calculating Turn around time
    cmpl_T=0;
    Total=0;
    for(int i=0;i<n;i++)
    {
        cmpl_T+=BT[i];
        TT[i]=cmpl_T-AT[i];
        Total+=TT[i];
    }
    Avg_TT=Total/n;

    // printing of outputs

    printf("Process ,Waiting_time ,TurnA_time\n");
    for(int i=0;i<n;i++)
    {
        printf("%d\t\t%d\t\t%d\n",i+1,WT[i],TT[i]);
    }
    printf("Average waiting time is : %f\n",Avg_WT);
    printf("Average turn around time is : %f\n",Avg_TT);
    return 0;
}
```

Sample Output:

```
Enter number of the process
2
Enter Arrival time and Burst time of the process
AT BT
0   4
1   8
Process ,Waiting_time ,TurnA_time
1            0              4
2            3              11
Average waiting time is : 1.500000
Average turn around time is : 7.500000
```

Pre- Lab Questions

1. What are the different ways for scheduling a job in operating systems?
2. What are the different types of Real-Time Scheduling?
3. What is a long term scheduler & short term schedulers?
4. What are the different functions of Scheduler?
5. Define non-preemptive scheduling.
6. Define dispatcher and its functions?
7. Define dispatch latency?
8. Define CPU scheduling.
9. What is a Dispatcher?
10. What is turnaround time?
11. Define Preemptive Scheduling.
12. Difference between preemptive and non preemptive scheduling.
13. How does SJF Scheduling algorithm works.
14. Tell us something about Mutex.
15. Is non-pre-emptive scheduling frequently used in a computer? Why?
16. What type of scheduling is there in RTOS?
17. What is meant by Input Queue?
18. Define Process.
19. What are the different process states available?
20. What is meant by Creating a Process?
21. What is means by Resuming a Process?
22. What is meant by Suspending a Process?
23. What is meant by Context Switching?
24. What is meant by PSW?
25. Define Mutual Exclusion.
26. What is meant by Co-operating process?

4.2 SJF ALGORITHM

Objectives

To implement Shortest Job First Scheduling Algorithm and display Gantt chart, turnaround time and waiting time in C.

Learning Outcomes:

After the completion of this experiment, student will be able to

- Know how shortest job scheduling works.
- Schedule the processes or jobs to the CPU.
- Compare the different scheduling methods.
- Will be able to create Gantt Chart and Calculate the Average waiting time and turn around time .

Problem Statement:

The program to perform scheduling for the processes using Shortest Job First method to display the following

1. Gantt chart
2. Average waiting time
3. Average turnaround time
4. Processes in the Queue

Algorithm:

```
1. Define an array of structure process with members pid, btime, wtime & ttime.
2. Get length of the ready queue, i.e., number of process (say n).
3. Obtain btime for each process.
4. Sort the processes according to their btime in ascending order.
     a. If two process have same btime, then FCFS is used to resolve the tie.
5. The wtime for first process is 0.
6. Compute wtime and ttime for each process as:
     a. wtime i+1= wtime i+ btime i b. ttime i = wtime i + btime i
7. Compute average waiting time awat and average turnaround time atur.
8. Display btime,ttime and wtime for each process.
9. Display GANTT chart for the above scheduling.
10. Display awat and atur.
11. Stop.
```

Execution of the Program:

```
$ cc pgm1.c   //Compiling the Program

$ ./a.out     //Executing the program
```

Sample Coding:

```
#include<stdio.h>
 int main()
{
    int bt[20],p[20],wt[20],tat[20],i,j,n,total=0,pos,temp;
    float avg_wt,avg_tat;
    printf("\nEnter number of process:");
    scanf("%d",&n);

    printf("\nEnter Burst Time:n");
    for(i=0;i<n;i++)
    {
        printf("p%d:",i+1);
        scanf("%d",&bt[i]);
        p[i]=i+1;
    }

   //sorting of burst times
    for(i=0;i<n;i++)
    {
        pos=i;
        for(j=i+1;j<n;j++)
        {
            if(bt[j]<bt[pos])
                pos=j;
        }

        temp=bt[i];
        bt[i]=bt[pos];
        bt[pos]=temp;

        temp=p[i];
        p[i]=p[pos];
        p[pos]=temp;
    }

    wt[0]=0;


    for(i=1;i<n;i++)
    {
        wt[i]=0;
        for(j=0;j<i;j++)
            wt[i]+=bt[j];

        total+=wt[i];
    }

    avg_wt=(float)total/n;
    total=0;
    printf("nProcesst    Burst Time    tWaiting TimetTurnaround Time");
    for(i=0;i<n;i++)
    {
       tat[i]=bt[i]+wt[i];
       total+=tat[i];
       printf("\np%d\t%d\t%d\t%d",p[i],bt[i],wt[i],tat[i]);
    }

    avg_tat=(float)total/n;
    printf("\nAverage Waiting Time=%f",avg_wt);
    printf("\nAverage Turnaround Time=%f",avg_tat);
}
```

Sample Output:

```
Enter number of process:4
Enter Burst Time:
p1:6
p2:7
p3:2
p4:9
Processt    Burst Time    Waiting Timet    Turnaround Time
p3              2              0              2
p1              6              2              8
p2              7              8              15
p4              9              15             24
Average Waiting Time=6.250000
Average Turnaround Time=12.250000
```

Pre- Lab Questions

1. What are the different ways for scheduling a job in operating systems?
2. What are the different types of Real-Time Scheduling?
3. What is a long term scheduler & short term schedulers?
4. What are the different functions of Scheduler?
5. Define non-preemptive scheduling.
6. Define dispatcher and its functions?
7. Define dispatch latency?
8. Define CPU scheduling.
9. What is a Dispatcher?
10. What is turnaround time?
11. Define Preemptive Scheduling.
12. Difference between preemptive and non preemptive scheduling.
13. How does SJF Scheduling algorithm works.
14. Tell us something about Mutex.
15. Is non-pre-emptive scheduling frequently used in a computer? Why?
16. What type of scheduling is there in RTOS?
17. What is meant by Input Queue?
18. Define Process.
19. What are the different process states available?

20. What is meant by Creating a Process?
21. What is means by Resuming a Process?
22. What is meant by Suspending a Process?
23. What is meant by Context Switching?
24. What is meant by PSW?
25. Define Mutual Exclusion.
26. What is meant by Co-operating process?

4.3 SRTF ALGORITHM

Objectives

To implement SRTF Scheduling Algorithm and display Gantt chart, turnaround time and waiting time in C.

Learning Outcomes:

After the completion of this experiment, student will be able to

- Schedule the processes or jobs to the CPU
- Will be able to create Gantt Chart and Calculate the Average waiting time and turn around time

Problem Statement:

The program to perform scheduling for the processes using first come first serve methods to display the following

1. Gantt chart
2. Average waiting time
3. Average turnaround time
4. Processes in the Queue

Algorithm:

1. Create the number of process.
2. Get the ID and Service time for each process.
3. Initially, waiting time of first process is zero and Total time for the first process is the starting time of that process.
4. Calculate the Total time and Processing time for the remaining processes.
5. Waiting time of one process is the Total time of the previous process.
6. Total time of process is calculated by adding Waiting time and Service time.
7. Total waiting time is calculated by adding the waiting time for lack process.
8. Total turnaround time is calculated by adding all total time of each process.
9. Calculate Average waiting time by dividing the total waiting time by total number of process.
10. Calculate Average turnaround time by dividing the total time by the number of process.
11. Display the result.

Execution of the Program:

```
$ cc pgm1.c   //Compiling the Program

$ ./a.out       //Executing the program
```

Sample Coding:

```
#include<stdio.h>
int main()
{
   int bt[20],p[20],wt[20],tat[20],i,j,n,total=0,pos,temp;
   float avg_wt,avg_tat;
   printf("\nEnter number of process:");
   scanf("%d",&n);

   printf("\nEnter Burst Time:n");
   for(i=0;i<n;i++)
   {
      printf("p%d:",i+1);
      scanf("%d",&bt[i]);
      p[i]=i+1;
   }

  //sorting of burst times
   for(i=0;i<n;i++)
   {
      pos=i;
      for(j=i+1;j<n;j++)
      {
         if(bt[j]<bt[pos])
            pos=j;
      }

      temp=bt[i];
      bt[i]=bt[pos];
      bt[pos]=temp;

      temp=p[i];
      p[i]=p[pos];
      p[pos]=temp;
   }

   wt[0]=0;


   for(i=1;i<n;i++)
   {
      wt[i]=0;
      for(j=0;j<i;j++)
         wt[i]+=bt[j];
```

```
        total+=wt[i];
    }

    avg_wt=(float)total/n;
    total=0;

    printf("nProcesst    Burst Time    tWaiting TimetTurnaround Time");
    for(i=0;i<n;i++)
    {
        tat[i]=bt[i]+wt[i];
        total+=tat[i];
        printf("np%d\t%d\t%d\t%d",p[i],bt[i],wt[i],tat[i]);
    }

    avg_tat=(float)total/n;
    printf("nnAverage Waiting Time=%f",avg_wt);
    printf("nAverage Turnaround Time=%fn",avg_tat);
}
```

Sample Output:

```
Enter number of process:4

Enter Burst Time:
p1:4
p2:6
p3:7
p4:2
Process    Burst Time   Waiting Timet   Turnaround Time
p4             2              0                  2
p1             4              2                  6
p2            6               6                  12
p3             7              12                   19
Average Waiting Time=5.000000
Average Turnaround Time=9.750000
```

Pre- Lab Questions

1. What are the different ways for scheduling a job in operating systems?
2. What are the different types of Real-Time Scheduling?
3. What is a long term scheduler & short term schedulers?
4. What are the different functions of Scheduler?
5. Define non-preemptive scheduling.
6. Define dispatcher and its functions?
7. Define dispatch latency?
8. Define CPU scheduling.
9. What is a Dispatcher?
10. What is turnaround time?
11. Define Preemptive Scheduling.
12. Difference between preemptive and non preemptive scheduling.

13. How does SJF Scheduling algorithm works.
14. Tell us something about Mutex.
15. Is non-pre-emptive scheduling frequently used in a computer? Why?
16. What type of scheduling is there in RTOS?
17. What is meant by Input Queue?
18. Define Process.
19. What are the different process states available?
20. What is meant by Creating a Process?
21. What is means by Resuming a Process?
22. What is meant by Suspending a Process?
23. What is meant by Context Switching?
24. What is meant by PSW?
25. Define Mutual Exclusion.
26. What is meant by Co-operating process?

4.4 PRIORITY SCHEDULING ALGORITHM

<u>Objectives:</u>

To implement Priority Scheduling Algorithm and display Gantt chart, turnaround time and waiting time using C.

<u>Learning Outcomes:</u>

After the completion of this experiment, student will be able to

- Know how Priority Scheduling works.
- Schedule the processes or jobs to the CPU.
- Compare the different scheduling methods.
- Will be able to create Gantt Chart and Calculate the Average waiting time and turn around time .

<u>Problem Statement:</u>

The program to perform scheduling for the processes using first come first serve methods to display the following

1. Gantt chart
2. Average waiting time
3. Average turnaround time
4. Processes in the Queue

<u>Algorithm:</u>

1. Get the number of process, burst time and priority.
2. Using for loop i=0 to n-1 do step 1 to 6.
3. If i=0,wait time=0,T[0]=b[0];
4. T[i]=T[i-1]+b[i] and wt[i]=T[i]-b[i].
5. Total waiting time is calculated by adding the waiting time for lack process.

6. Total turnaround time is calculated by adding all total time of each process.
7. Calculate Average waiting time by dividing the total waiting time by total number of process.
8. Calculate Average turnaround time by dividing the total time by the number of process.
9. Display the result.

Execution of the Program:

```
$ cc pgm1.c   //Compiling the Program

$ ./a.out         //Executing the program
```

Sample Coding:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
   int et[20],at[10],n,i,j,temp,p[10],st[10],ft[10],wt[10],ta[10];
   int totwt=0,totta=0;
   float awt,ata;
   char pn[10][10],t[10];
   //clrscr();
   printf("Enter the number of process:");
   scanf("%d",&n);
   for(i=0; i<n; i++)
   {
      printf("Enter process name,arrivaltime,execution time & priority:");
      //flushall();
      scanf("%s%d%d%d",pn[i],&at[i],&et[i],&p[i]);
   }
   for(i=0; i<n; i++)
      for(j=0; j<n; j++)
      {
         if(p[i]<p[j])
         {
            temp=p[i];
            p[i]=p[j];
            p[j]=temp;
            temp=at[i];
            at[i]=at[j];
            at[j]=temp;
            temp=et[i];
            et[i]=et[j];
            et[j]=temp;
            strcpy(t,pn[i]);
            strcpy(pn[i],pn[j]);
            strcpy(pn[j],t);
         }
      }
   for(i=0; i<n; i++)
```

```
        {

            if(i==0)
            {
                st[i]=at[i];
                wt[i]=st[i]-at[i];
                ft[i]=st[i]+et[i];
                ta[i]=ft[i]-at[i];
            }
            else
            {
                st[i]=ft[i-1];
                wt[i]=st[i]-at[i];
                ft[i]=st[i]+et[i];
                ta[i]=ft[i]-at[i];
            }
            totwt+=wt[i];
            totta+=ta[i];
        }
        awt=(float)totwt/n;
        ata=(float)totta/n;
        printf("\nPname\tarrivaltime\texecutiontime\tpriority\twaitingtime\ttatime");
        for(i=0; i<n; i++)
            printf("\n%s\t%5d\t\t%5d\t\t%5d\t\t%5d\t\t%5d",pn[i],at[i],et[i],p[i],wt[i],ta[i]);
        printf("\nAverage waiting time is:%f",awt);
        printf("\nAverage turnaroundtime is:%f",ata);
        getch();
    }
```

Sample Output:

```
Enter the number of process:3
Enter process name,arrivaltime,execution time & priority:1 2 1 3
Enter process name,arrivaltime,execution time & priority:3 5 4 1
Enter process name,arrivaltime,execution time & priority:3 4 6 2

Pname   arrivaltime    executiontime   priority      waitingtime    tatime
3         5              4               1             0              4
3         4              6               2             5             11
1         2              1               3            13             14
Average waiting time is:6.000000
Average turnaroundtime is:9.666667
```

Pre-Lab Questions

1. What are the different ways for scheduling a job in operating systems?
2. What are the different types of Real-Time Scheduling?
3. What is a long term scheduler & short term schedulers?
4. What are the different functions of Scheduler?
5. Define non-preemptive scheduling.

6. Define dispatcher and its functions?
7. Define dispatch latency?
8. Define CPU scheduling.
9. What is a Dispatcher?
10. What is turnaround time?
11. Define Preemptive Scheduling.
12. Difference between preemptive and non preemptive scheduling.
13. How does Priority Scheduling algorithm works.
14. What is context switching?
15. Tell us something about Mutex.
16. Is non-pre-emptive scheduling frequently used in a computer? Why?
17. What type of scheduling is there in RTOS?
18. What is meant by Input Queue?
19. Define Process.
20. What are the different process states available?
21. What is meant by Creating a Process?
22. What is means by Resuming a Process?
23. What is meant by Suspending a Process?
24. What is meant by Context Switching?
25. What is meant by PSW?
26. Define Mutual Exclusion.
27. What is meant by Co-operating process?

4.5 ROUND ROBIN ALGORITHM

Objectives:

To implement Round Robin Scheduling Algorithm and display Gantt chart, turnaround time and waiting time using C.

Learning Outcomes:

After the completion of this experiment, student will be able to

- Know how round robin scheduling works.
- Schedule the processes or jobs to the CPU.
- Compare the different scheduling methods.
- Will be able to create Gantt Chart and Calculate the Average waiting time and turn around time .
- Recollect the looping concepts in C.

Problem Statement:

The program to perform scheduling for the processes using first come first serve methods to display the following

1. Gantt chart

2. Average waiting time
3. Average turnaround time
4. Processes in the Queue

Algorithm:

```
1. Initialize all the structure elements
2. Receive inputs from the user to fill process id,burst time and arrival time.
3. Calculate the waiting time for all the process id.
     i) The waiting time for first instance of a process is
          calculated as: a[i].waittime=count + a[i].arrivt
     ii) The waiting time for the rest of the instances of the process is calculated as:
          a) If the time quantum is greater than the remaining burst time then
             waiting time is calculated as:
                  a[i].waittime=count + tq
          b) Else if the time quantum is greater than the remaining burst time
             then waiting
             time is calculated as:
                  a[i].waittime=count - remaining burst time
4. Calculate the average waiting time and average turnaround time
5. Print the results of the step 4.
```

Execution of the Program:

```
$ cc pgm1.c        //Compiling the Program

$ ./a.out    //Executing the program
```

Sample Coding

```
#include<stdio.h>

int main()
{
    int i, limit, total = 0, x, counter = 0, time_quantum;
    int wait_time = 0, turnaround_time = 0, arrival_time[10], burst_time[10], temp[10];
    float average_wait_time, average_turnaround_time;
    printf("\nEnter Total Number of Processes: ");
    scanf("%d", &limit);
    x = limit;
    for(i = 0; i < limit; i++)
    {
        printf("\nEnter Details of Process[%d]", i + 1);

        printf("Arrival Time:");

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

        printf("Burst Time:");

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

```
            temp[i] = burst_time[i];
      }

      printf("\nEnter Time Quantum:");
      scanf("%d", &time_quantum);
      printf("\nProcess IDttBurst Timet Turnaround Timet Waiting Time\n");
      for(total = 0, i = 0; x != 0;)
      {
            if(temp[i] <= time_quantum && temp[i] > 0)
            {
                  total = total + temp[i];
                  temp[i] = 0;
                  counter = 1;
            }
            else if(temp[i] > 0)
            {
                  temp[i] = temp[i] - time_quantum;
                  total = total + time_quantum;
            }
            if(temp[i] == 0 && counter == 1)
            {
                  x--;
                  printf("\nProcess[%d]\t%d\t%d\nt%d", i + 1, burst_time[i], total -
arrival_time[i], total - arrival_time[i] - burst_time[i]);
                  wait_time = wait_time + total - arrival_time[i] - burst_time[i];
                  turnaround_time = turnaround_time + total - arrival_time[i];
                  counter = 0;
            }
            if(i == limit - 1)
            {
                  i = 0;
            }
            else if(arrival_time[i + 1] <= total)
            {
                  i++;
            }
            else
            {
                  i = 0;
            }
      }

      average_wait_time = wait_time * 1.0 / limit;
      average_turnaround_time = turnaround_time * 1.0 / limit;
      printf("\nAverage Waiting Time:t%f", average_wait_time);
      printf("\nAvg Turnaround Time:t%f", average_turnaround_time);
      return 0;
}
```

Sample output

```
Sample Output:
Enter Total Number of Processes: 4
Enter Details of Process[1]
Arrival Time:00
Burst Time:4

Enter Details of Process[2]
Arrival Time:1
Burst Time:7

Enter Details of Process[3]
Arrival Time:2
Burst Time:5

Enter Details of Process[4]
Arrival Time:3
Burst Time:6

Enter Time Quantum:2


Process ID       Burst Time       Turnaround Timer           Waiting Time

Process[1]            4                  10                             6
Process[3]             5                  17                            12
Process[4]            6                  18                             12
Process[2]            7                   21                            14
Average Waiting Time:11.000000
Avg Turnaround Time:16.500000
```

Pre-lab Questions:

1. What are the different ways for scheduling a job in operating systems?
2. What are the different types of Real-Time Scheduling?
3. What is a long term scheduler & short term schedulers?
4. What are the different functions of Scheduler?
5. Define non-preemptive scheduling.
6. Define dispatcher and its functions?
7. Define dispatch latency?
8. Define CPU scheduling.
9. What is a Dispatcher?
10. What is turnaround time?
11. Define Preemptive Scheduling.
12. Difference between preemptive and non preemptive scheduling.
13. How does Round Robin Scheduling algorithm works.
14. Why is round robin algorithm considered better than first come first served algorithm?

15. What is context switching?
16. Tell us something about Mutex.
17. Is non-pre-emptive scheduling frequently used in a computer? Why?
18. What type of scheduling is there in RTOS?
19. What is meant by Input Queue?
20. Define Process.
21. What are the different process states available?
22. What is meant by Creating a Process?
23. What is means by Resuming a Process?
24. What is meant by Suspending a Process?
25. What is meant by Context Switching?
26. What is meant by PSW?
27. Define Mutual Exclusion.
28. What is meant by Co-operating process?

Conclusion:

Thus the C program to perform CPU Scheduling for the processes and to display Gantt chart, turnaround time and waiting time was performed successfully.

EX.NO:5.1 PRODUCER – CONSUMER PROBLEM USING SEMAPHORES

Objectives:

To implement the concept of Producer – Consumer Problem using Semaphores

Learning Outcomes:

After the completion of this experiment, student will be able to

- Understand the concept of semaphores
- Know how the process can communicate.
- Will be able to create system calls for a process

Problem Statement:

The program to perform Producer – Consumer Problem using Semaphores

Algorithm:

1. mutex = 1
2. Full = 0 // Initially, all slots are empty. Thus full slots are 0
3. Empty = n // All slots are empty initially

   Producer
4. When producer produces an item then the value of "empty" is reduced by 1 because one slot will be filled now.
5. The value of mutex is also reduced to prevent consumer to access the buffer.
6. Now, the producer has placed the item and thus the value of "full" is increased by 1.
7. The value of mutex is also increased by 1 because the task of producer has been completed and consumer can access the buffer.

   Consumer
8. As the consumer is removing an item from buffer, therefore the value of "full" is reduced by 1 and the value is mutex is also reduced so that the producer cannot access the buffer at this moment.
9. Now, the consumer has consumed the item, thus increasing the value of "empty" by 1.
10. The value of mutex is also increased so that producer can access the buffer now.

Samplecode:

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

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

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

```
printf("\n1.Producer\n2.Consumer\n3.Exit");
while(1)
{
printf("\nEnter your choice:");
scanf("%d",&n);
switch(n)
{
case 1: if((mutex==1)&&(empty!=0))
producer();
else
printf("Buffer is full!!");
break;
case 2: if((mutex==1)&&(full!=0))
consumer();
else
printf("Buffer is empty!!");
break;
case 3:
exit(0);
break;
}
}
return 0;
}

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

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

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

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

```
mutex=signal(mutex);
}
```

Output

```
1.Producer
2.Consumer
3.Exit
Enter  your choice:1
Producer produces the item 1
Enter your choice:2
Consumer consumes item 1
Enter your choice:2
Buffer is empty!!
Enter your choice:1
Producer produces the item 1
Enter  your choice:1
Producer produces the item 2
Enter  your choice:1
Producer produces the item 3
Enter your choice:1
Buffer is full!!
Enter your choice:3
```

Pre- Lab Questions:

1. What are semaphores?
2. Mention the categories of semaphores.
3. How would you initialize a semaphore set?
4. What are the two main operations of semaphores?
5. What do you mean by inter process communication?
6. How can you perform a operation on semaphore?
7. What are pipes?
8. List out the importance of message queue?
9. What is semctl()?
10. How would you perform control operations in a semaphore?
11. List the parameters of semaphore control operation.

Conclusion:

Thus the C program to perform Producer – Consumer Problem using Semaphores was performed successfully.

EX.NO:5.2 DINING PHILOSOPHER’S PROBLEM TO DEMONSTRATE PROCESS SYNCHRONIZATION.

Objectives:

To implement the concept of Dining Philosopher’s Problem to demonstrate Process Synchronization.

Learning Outcomes:

After the completion of this experiment, student will be able to

- Understand the concept of semaphores
- Know how the process can communicate & synchronization.
- Will be able to create system calls for a process

Problem Statement:

The program to perform Dining Philosopher’s Problem to demonstrate Process Synchronization.

Algorithm:

1. The philosopher is instructed to think till the left fork is available, when it is available, hold it.
2. The philosopher is instructed to think till the right fork is available, when it is available, hold it.
3. The philosopher is instructed to eat when both forks are available.
4. then, put the right fork down first
5. then, put the left fork down next
6. repeat from the beginning.

Sample code :

```
#include<stdio.h>

#define n 4

int compltedPhilo = 0,i;

struct fork{
int taken;
}ForkAvil[n];

struct philosp{
int left;
int right;
}Philostatus[n];

void goForDinner(int philID){ //same like threads concept here cases implemented
if(Philostatus[philID].left==10 && Philostatus[philID].right==10)
```

```
        printf("Philosopher %d completed his dinner\n",philID+1);
//if already completed dinner
else if(Philostatus[philID].left==1 && Philostatus[philID].right==1){
            //if just taken two forks
            printf("Philosopher %d completed his dinner\n",philID+1);

            Philostatus[philID].left = Philostatus[philID].right = 10; //remembering that he
completed dinner by assigning value 10
            int otherFork = philID-1;

            if(otherFork== -1)
                otherFork=(n-1);

            ForkAvil[philID].taken = ForkAvil[otherFork].taken = 0; //releasing forks
            printf("Philosopher %d released fork %d and fork
%d\n",philID+1,philID+1,otherFork+1);
            compltedPhilo++;
        }
        else if(Philostatus[philID].left==1 && Philostatus[philID].right==0){ //left already taken,
trying for right fork
                if(philID==(n-1)){
                    if(ForkAvil[philID].taken==0){ //KEY POINT OF THIS PROBLEM, THAT
LAST PHILOSOPHER TRYING IN reverse DIRECTION
                        ForkAvil[philID].taken = Philostatus[philID].right = 1;
                        printf("Fork %d taken by philosopher %d\n",philID+1,philID+1);
                    }else{
                        printf("Philosopher %d is waiting for fork %d\n",philID+1,philID+1);
                    }
                }else{ //except last philosopher case
                    int dupphilID = philID;
                    philID-=1;

                    if(philID== -1)
                        philID=(n-1);

                    if(ForkAvil[philID].taken == 0){
                        ForkAvil[philID].taken = Philostatus[dupphilID].right = 1;
                        printf("Fork %d taken by Philosopher %d\n",philID+1,dupphilID+1);
                    }else{
                        printf("Philosopher %d is waiting for Fork %d\n",dupphilID+1,philID+1);
                    }
                }
            }
            else if(Philostatus[philID].left==0){ //nothing taken yet
                    if(philID==(n-1)){
                        if(ForkAvil[philID-1].taken==0){ //KEY POINT OF THIS PROBLEM, THAT
LAST PHILOSOPHER TRYING IN reverse DIRECTION
                            ForkAvil[philID-1].taken = Philostatus[philID].left = 1;
                            printf("Fork %d taken by philosopher %d\n",philID,philID+1);
                        }else{
                            printf("Philosopher %d is waiting for fork %d\n",philID+1,philID);
                        }
                }else{ //except last philosopher case
                    if(ForkAvil[philID].taken == 0){
                        ForkAvil[philID].taken = Philostatus[philID].left = 1;
                        printf("Fork %d taken by Philosopher %d\n",philID+1,philID+1);
                    }else{
                        printf("Philosopher %d is waiting for Fork %d\n",philID+1,philID+1);
                    }
                }
        }else{}
}

int main(){
for(i=0;i<n;i++)
        ForkAvil[i].taken=Philostatus[i].left=Philostatus[i].right=0;

while(compltedPhilo<n){
/* Observe here carefully, while loop will run until all philosophers complete dinner
Actually problem of deadlock occur only thy try to take at same time
This for loop will say that they are trying at same time. And remaining status will print by go
for dinner function
*/
for(i=0;i<n;i++)
            goForDinner(i);
printf("\nTill now num of philosophers completed dinner are %d\n\n",compltedPhilo);
}

return 0;
}
```

Output

```
Fork 1 taken by Philosopher 1
Fork 2 taken by Philosopher 2
Fork 3 taken by Philosopher 3
Philosopher 4 is waiting for fork 3
Till now num of philosophers completed dinner are 0
Fork 4 taken by Philosopher 1
Philosopher 2 is waiting for Fork 1
Philosopher 3 is waiting for Fork 2
Philosopher 4 is waiting for fork 3
Till now num of philosophers completed dinner are 0
Philosopher 1 completed his dinner
Philosopher 1 released fork 1 and fork 4
Fork 1 taken by Philosopher 2
Philosopher 3 is waiting for Fork 2
Philosopher 4 is waiting for fork 3
Till now num of philosophers completed dinner are 1
Philosopher 1 completed his dinner
Philosopher 2 completed his dinner
Philosopher 2 released fork 2 and fork 1
```

```
Fork 2 taken by Philosopher 3
Philosopher 4 is waiting for fork 3
Till now num of philosophers completed dinner are 2
Philosopher 1 completed his dinner
Philosopher 2 completed his dinner
Philosopher 3 completed his dinner
Philosopher 3 released fork 3 and fork 2
Fork 3 taken by philosopher 4
Till now num of philosophers completed dinner are 3
Philosopher 1 completed his dinner
Philosopher 2 completed his dinner
Philosopher 3 completed his dinner
Fork 4 taken by philosopher 4
Till now num of philosophers completed dinner are 3
Philosopher 1 completed his dinner
Philosopher 2 completed his dinner
Philosopher 3 completed his dinner
Philosopher 4 completed his dinner
Philosopher 4 released fork 4 and fork 3
Till now num of philosophers completed dinner are 4
```

Pre- Lab Questions:

1. What is the problem in Dining Philosophers?
2. What is the solution for dining philosophers problem?
3. How many dining philosophers can eat simultaneously?
4. What is deadlock explain dining philosopher problem?
5. What is dining philosopher problem and how can it be solved using mutex locks?
6. Why the philosophers are eating spaghetti in dining philosophers problem?
7. What are pipes?
8. List out the importance of message queue?
9. How would you perform control operations in a semaphore?
10. List the parameters of semaphore control operation.

Conclusion:

Thus the C program to perform Dining Philosopher’s Problem to demonstrate Process Synchronization was performed successfully.

EX.NO: 6 **IMPLEMENTATION OF BANKERS ALGORITHM**

Objectives:

To implement of safety algorithm for deadlock avoidance using C.

Learning Outcomes:

After the completion of this experiment, student will be able to

- Understand the safe state of processes.
- Learn how deadlocks can be avoided.

Problem Statement:

The program to implement the Safety algorithm for deadlock avoidance.

Execution of the Program:

$ cc filename.c //Compiling the Program

$ ./a.out //Executing the program

Sample Coding:

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

int main()
{
int Max[10][10], need[10][10], alloc[10][10], avail[10], completed[10], safeSequence[10];
int p, r, i, j, process, count;
count = 0;

printf("Enter the no of processes : ");
scanf("%d", &p);

for(i = 0; i< p; i++)
            completed[i] = 0;

printf("\n\nEnter the no of resources : ");
scanf("%d", &r);

printf("\n\nEnter the Max Matrix for each process : ");
for(i = 0; i < p; i++)
{
            printf("\nFor process %d : ", i + 1);
            for(j = 0; j < r; j++)
                        scanf("%d", &Max[i][j]);
}

printf("\n\nEnter the allocation for each process : ");
for(i = 0; i < p; i++)
{
		printf("\nFor process %d : ",i + 1);
		for(j = 0; j < r; j++)
			scanf("%d", &alloc[i][j]);
	}

	printf("\n\nEnter the Available Resources : ");
	for(i = 0; i < r; i++)
			scanf("%d", &avail[i]);


		for(i = 0; i < p; i++)
			for(j = 0; j < r; j++)
				need[i][j] = Max[i][j] - alloc[i][j];

	do
	{
		printf("\n Max matrix:\tAllocation matrix:\n");
		for(i = 0; i < p; i++)
		{
			for( j = 0; j < r; j++)
				printf("%d ", Max[i][j]);
			printf("\t\t");
			for( j = 0; j < r; j++)
				printf("%d ", alloc[i][j]);
			printf("\n");
		}
		process = -1;
		for(i = 0; i < p; i++)
		{
			if(completed[i] == 0)//if not completed
			{
				process = i ;
				for(j = 0; j < r; j++)
				{
					if(avail[j] < need[i][j])
					{
						process = -1;
						break;
					}
				}
			}
			if(process != -1)
				break;
		}

		if(process != -1)
		{
			printf("\nProcess %d runs to completion!", process + 1);
			safeSequence[count] = process + 1;
			count++;
			for(j = 0; j < r; j++)
			{
				avail[j] += alloc[process][j];
				alloc[process][j] = 0;
				Max[process][j] = 0;
				completed[process] = 1;
			}
		}
}while(count != p && process != -1);

if(count == p)
{
		printf("\nThe system is in a safe state!!\n");
		printf("Safe Sequence : < ");
		for( i = 0; i < p; i++)
				printf("%d  ", safeSequence[i]);
		printf(">\n");
}
else
		printf("\nThe system is in an unsafe state!!");
getch();
}
```

Sample Output:

```
Enter the no of processes : 3
Enter the no of resources : 3

Enter the Max Matrix for each process :
For process 1 : 4 4 4
For process 2 : 3 4 5
For process 3 : 5 2 4

Enter the allocation for each process :
For process 1 : 3 2 1
For process 2 : 1 1 2
For process 3 : 4 1 2

Enter the Available Resources : 10 7 7

 Max matrix:    Allocation matrix:
4  4  4           3  2  1
3  4  5           1  1  2
5  2  4           4  1  2

Process 1 runs to completion!
 Max matrix:    Allocation matrix:
0  0  0           0  0  0
3  4  5           1  1  2
5  2  4           4  1  2

Process 2 runs to completion!
 Max matrix:    Allocation matrix:
0 0  0            0 0  0
0 0  0            0 0  0
5 2  4            4 1  2

Process 3 runs to completion!
The system is in a safe state!!
Safe Sequence : < 1  2  3 >
```

Test cases:

Enter the number of resources and maximum instances of each resources.

Enter the number of processes and allocation matrix of resources for each process with respect to instances.

Enter the maximum matrix of resources and check the safe state of processes.

Pre-lab questions:

1. Define Deadlock.
2. What are the necessary conditions for deadlock?
3. Define mutual exclusion.
4. Define safe state.
5. Mention the deadlock prevention mechanisms.
6. What are the deadlock avoidance algorithms?
7. What are the methods to break a deadlock?
8. What are the issues to be addressed to deal with the preemption of deadlock?
9. Differentiate preemption and no-preemption.
10. What is hold and wait?

Conclusion:

Thus the C program to implement the safety algorithm for deadlock avoidance was implemented successfully.

EX.No 7 MEMORY ALLOCATION AND MANAGEMENT TECHNIQUES (BEST FIT, WORST FIT, FIRST FIT)

Objectives:

To implement the memory allocation and management concept using C.

Learning Outcomes:

After the completion of this experiment, student will be able to

- Understand the memory management types- Best fit, worst fit and First fit.
- Learn how a process can be fitted into a memory block.

Problem Statement:

The program to implement the memory management strategies.

Execution of the Program:

$ cc filename.c //Compiling the Program

$ ./a.out //Executing the program

7.1 FIRST FIT

Algorithm :

1. Read the number of processes and number of the block from the user
2. Read the size of each block and the size of all the process requests.
3. Start allocating the processes
4. Display the results as shown below
5. Stop

Sample Coding:

```
#include <stdio.h>
int main()
{
int a[10], b[10], a1, b1, flags[10], all[10];
int i, j;
printf("\n\t\t\tMemory Management"
" Scheme -"
" First Fit\n");
for (i = 0; i < 10; i++)
{
flags[i] = 0;
all[i] = -1;
}
printf("Enter number of blocks: ");
scanf("%d", &a1);
printf("\nEnter the size of each"
" block:\n ");
for (i = 0; i < a1; i++)
{
printf("Block no.%d: ", i);
scanf("%d", &a[i]);
}
printf("\nEnter no. of "
"processes: ");
scanf("%d", &b1);
printf("\nEnter size of each"
" process:\n ");
for (i = 0; i < b1; i++)
{
printf("Process no.%d: ", i);
scanf("%d", &b[i]);
}
for (i = 0; i < b1; i++)
for (j = 0; j < a1; j++)
if (flags[j] == 0 && a[j] >= b[i])
{
all[j] = i;
flags[j] = 1;
break;
}
printf("\nBlock no.\tsize\t\t"
"process no.\t\tsize");
for (i = 0; i < a1; i++)
{
printf("\n%d\t\t%d\t\t",
 i + 1, a[i]);
if (flags[i] == 1)
{
printf("%d\t\t\t%d", all[i]
 + 1, b[all[i]]);
}
else
printf("Not allocated");
}
printf("\n");
}
```

Sample Output:

```
Memory Management Scheme - First Fit
Enter number of blocks: 3

Enter the size of each block:
 Block no.0: 12
Block no.1: 9
Block no.2: 7

Enter no. of processes: 3

Enter size of each process:
 Process no.0: 5
Process no.1: 4
Process no.2: 9

Block no.      size          process no.   size
1              12             1            5
2              9             2            4
3              7             Not allocated
```

7.2 BEST FIT

Algorithm :

1. Read the number of processes and number of blocks from the user
2. Get the size of each block and process requests
3. Then select the best memory block
4. Display the result as shown below
5. The fragmentation column will keep track of wasted memory
6. Stop

Sample Coding:

```
#include <stdio.h>
int main()
{
int a[20], b[20], c[20], b1, c1;
int i, j, temp;
static int barr[20], carr[20];
printf("\n\t\t\tMemory Management Scheme - Best Fit");
printf("\nEnter the number of  blocks:");
scanf("%d", &b1);
printf("Enter the number of processes:");
scanf("%d", &c1);
int lowest = 9999;
printf("\nEnter the size of the blocks:\n");
for (i = 1; i <= b1; i++)
{
printf("Block no.%d:", i);
scanf("%d", &b[i]);
}
printf("\nEnter the size of the processes :\n");
for (i = 1; i <= c1; i++)
{
printf("Process no.%d:", i);
scanf("%d", &c[i]);
}
for (i = 1; i <= c1; i++)
{
```

```
for (j = 1; j <= b1; j++)
{
if (barr[j] != 1)
{
temp = b[j] - c[i];
if (temp >= 0)
if (lowest > temp)
{
carr[i] = j;
lowest = temp;
}
}
}
a[i] = lowest;
barr[carr[i]] = 1;
lowest = 10000;
}
printf("\nProcess_no\tProcess_size\tBlock_no\t Block_size\tFragment");
for (i = 1; i <= c1 && carr[i] != 0; i++)
{
printf("\n%d\t\t%d\t\t%d\t\t %d\t\t%d", i,
c[i], carr[i], b[carr[i]], a[i]);
}
printf("\n");
}
```

Sample Output:

```
Memory Management Scheme - Best Fit
Enter the number of blocks:5
Enter the number of processes:4

Enter the size of the blocks:
Block no.1:9
Block no.2:13
Block no.3:5
Block no.4:8
Block no.5:2

Enter the size of the processes :
Process   no.1:3
Process   no.2:4
Process   no.3:8
Process no.4:14

Process_no     Process_size   Block_no    Block_size      Fragment
1             3             3          52
2             4             4          84
3             8             1          91
```

7.3 WORST FIT

Algorithm :

1. Read total number of block and files
2. Get the size of each block and the files from the user
3. Start from the first process and find the maximum block size that can be assigned to the current process, if found then assign it to the current process.
4. If not found then leave that process and move ahead to check the rest of the processes
5. Display the result as shown below
6. The fragmentation column keeps the track of wasted memory
7. stop

Sample Coding:

```
#include <stdio.h>
int main()
{
printf("\n\t\t\tMemory Management"
" Scheme - Worst Fit");
int i, j, nblocks, nfiles, temp, top = 0;
int frag[10], blocks[10], files[10];
static int block_arr[10], file_arr[10];
printf("\nEnter the Total Number "
"of Blocks: ");
scanf("%d", &nblocks);
printf("Enter the Total Number "
"of Files: ");
scanf("%d", &nfiles);
printf("\nEnter the Size of the "
"Blocks: \n");
for (i = 0; i < nblocks; i++)
{
printf("Block No.%d:\t", i + 1);
scanf("%d", &blocks[i]);
}
printf("Enter the Size of the "
"Files:\n");
for (i = 0; i < nfiles; i++)
{
printf("File No.%d:\t", i + 1);
scanf("%d", &files[i]);
}
for (i = 0; i < nfiles; i++)
{
for (j = 0; j < nblocks; j++)
{
if (block_arr[j] != 1)
{
```

```
temp = blocks[j] - files[i];
if (temp >= 0)
{
if (top < temp)
{
file_arr[i] = j;
top = temp;
}
}
}
frag[i] = top;
block_arr[file_arr[i]] = 1;
top = 0;
}
}
printf("\nFile Number\tFile Size\t"
"Block Number\tBlock Size\tFragment");
for (i = 0; i < nfiles; i++)
{
printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d"
, i, files[i],
file_arr[i], blocks[file_arr[i]], frag[i]);
}
printf("\n");
return 0;
}
```

Sample Output:

```
        Memory Management Scheme - Worst Fit
Enter the Total Number of Blocks: 5
Enter the Total Number of Files: 4

Enter the Size of the Blocks:
Block No.1:     5
Block No.2:     7
Block No.3:     4
Block No.4:     6
Block No.5:     3
Enter the Size of the Files:
File No.1:      1
File No.2:      3
File No.3:      4
File No.4:      2

File Number     File Size       Block Number    Block Size    Fragment
0               1               4               3             2
1               3               0               5             0
2               4               0               5             0
3               2               0               5             0
```

Pre-Lab Questions

1. What is the significance of memory allocation?
2. What are the different types of memory management schemes?
3. Mention the concept of first fit.
4. What is best fit?
5. What is worst fit?
6. What is fragmentation?
7. Which memory management scheme is considered to be more effective?
8. Is it necessary that the number of files should be equal to number of memory blocks?
9. What will happen if you could not fit the files into the blocks given?

Conclusion:

Thus the C program to implement the different types of memory allocation & management schemes.

EX.NO: 8 IMPLEMENTATION OF PAGE REPLACEMENT ALGORITHMS

Objectives:

To write a C program to implement page replacement algorithm.

Learning Outcomes:

After the completion of this experiment, student will be able to

- Understand how the page faults are calculated.
- Know how the pages are replaced in memory.
- Learn about the blocks, frame and how virtual memories are used.

Problem Statement:

The program to implement page replacement for replacing the pages in the virtual memory and calculating the number of page faults.

8.1 FIFO PAGE REPLACEMENT

Algorithm:

1. Start traversing the pages.
2. Now declare the size w.r.t length of the Page.
3. Check need of the replacement from the page to memory.
4. Similarly, Check the need of the replacement from the old page to new page in memory.
5. Now form the queue to hold all pages.
6. Insert Require page memory into the queue.
7. Check bad replacemets and page faults.
8. Get no of processes to be inserted.
9. Show the values.
10. Stop

Sample Coding:

```
#include <stdio.h>
int main()
{
int referenceString[10], pageFaults = 0, m, n, s, pages, frames;
printf("\nEnter the number of Pages:\t");
scanf("%d", &pages);
printf("\nEnter reference string values:\n");
for( m = 0; m < pages; m++)
{
  printf("Value No. [%d]:\t", m + 1);
  scanf("%d", &referenceString[m]);
}
printf("\n What are the total number of frames:\t");
{
  scanf("%d", &frames);
}
int temp[frames];
for(m = 0; m < frames; m++)
{
  temp[m] = -1;
}
for(m = 0; m < pages; m++)
{
  s = 0;
  for(n = 0; n < frames; n++)
   {
      if(referenceString[m] == temp[n])
         {
            s++;
            pageFaults--;
         }
   }
   pageFaults++;
   if((pageFaults <= frames) && (s == 0))
      {
        temp[m] = referenceString[m];
      }
   else if(s == 0)
      {
        temp[(pageFaults - 1) % frames] = referenceString[m];
      }
      printf("\n");
      for(n = 0; n < frames; n++)
       {
         printf("%d\t", temp[n]);
       }
}
printf("\nTotal Page Faults:\t%d\n", pageFaults);
return 0;
}
```

Sample Output:

```
Enter the number of Pages:      5
Enter reference string values:
Value No. [1]:  5
Value No. [2]:  4
Value No. [3]:  3
Value No. [4]:  2
Value No. [5]:  1

 What are the total number of frames:  4
5       -1      -1      -1
5       4       -1      -1
5       4       3       -1
5       4       3       2
1       4       3       2

Total Page Faults:      5
```

8.2 LRU PAGE REPLACEMENT

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

Sample Coding:

```
#include<stdio.h>
main()
{
int q[20],p[50],c=0,c1,d,f,i,j,k=0,n,r,t,b[20],c2[20];
printf("Enter no of pages:");
scanf("%d",&n);
printf("Enter the reference string:");
for(i=0;i<n;i++)
        scanf("%d",&p[i]);
printf("Enter no of frames:");
scanf("%d",&f);
q[k]=p[k];
printf("\n\t%d\n",q[k]);
c++;
k++;
for(i=1;i<n;i++)
        {
                c1=0;
                for(j=0;j<f;j++)
                {
                        if(p[i]!=q[j])
                        c1++;
                }
                if(c1==f)
                {
                        c++;
                        if(k<f)
                        {
                                q[k]=p[i];
                                k++;
                                for(j=0;j<k;j++)
                                printf("\t%d",q[j]);
                                printf("\n");
                        }
                        else
                        {
```

```
                                        for(r=0;r<f;r++)
                                        {
                                                c2[r]=0;
                                                for(j=i-1;j<n;j--)
                                                {
                                                if(q[r]!=p[j])
                                                c2[r]++;
                                                else
                                                break;
                                        }
                                }
                                for(r=0;r<f;r++)
                                b[r]=c2[r];
                                for(r=0;r<f;r++)
                                {
                                        for(j=r;j<f;j++)
                                        {
                                                if(b[r]<b[j])
                                                {
                                                        t=b[r];
                                                        b[r]=b[j];
                                                        b[j]=t;
                                                }
                                        }
                                }
                                for(r=0;r<f;r++)
                                {
                                        if(c2[r]==b[0])
                                        q[r]=p[i];
                                        printf("\t%d",q[r]);
                                }
                                printf("\n");
                        }
                }
        }
        printf("\nThe no of page faults is %d",c);
        }
```

Sample Output:

```
Enter no of pages:10

Enter the reference string:7 5 9 4 3 7 9 6 2 1

Enter no of frames:3

        7

        7       5

        7       5       9

        4       5       9

        4       3       9
```

```
4    3    7
9    3    7
9    6    7
9    6    2
1    6    2
```

The no of page faults is 10

8.3 OPTIMAL PAGE REPLACEMENT

Algorithm:

- 1- Start traversing the pages.
    1. i) If set holds less pages than capacity.
        - a) Insert page into the set one by one until the size of set reaches capacity or all page requests are processed.
        - b) Simultaneously maintain the recent occurred index of each page in a map called indexes.
        - c) Increment page fault
    2. ii) Else If current page is present in set, do nothing.
        - Else
            1. a) Find the page in the set that was least recently used. We find it using index array. We basically need to replace the page with minimum index.
            2. b) Replace the found page with current page.
            3. c) Increment page faults.
            4. d) Update index of current page.
- 2 – Return page faults.

<u>Sample Coding:</u>

```
#include<stdio.h>
#include<conio.h>
main()
{
   int fr[5],i,j,k,t[5],p=1,flag=0,page[25],psz,nf,t1,u[5];
   clrscr();
   printf("enter the number of frames:");
   scanf("%d",&nf);
   printf("\n enter the page size");
   scanf("%d",&psz);
   printf("\nenter the page sequence:");
   for(i=1; i<=psz; i++)
      scanf("%d",&page[i]);
   for(i=1; i<=nf; i++)
```

```
    fr[i]=-1;
for(i=1; i<=psz; i++)
{
    if(full(fr,nf)==1)
       break;
    else
    {
        flag=0;
        for(j=1; j<=nf; j++)
        {
           if(page[i]==fr[j])
            {
                flag=1;
                printf("          \t%d:\t",page[i]);
                break;
            }
        }
        if(flag==0)
        {
            fr[p]=page[i];
            printf("          \t%d:\t",page[i]);
            p++;
        }

        for(j=1; j<=nf; j++)
            printf(" %d  ",fr[j]);
        printf("\n");
    }
}
p=0;
for(; i<=psz; i++)
{
    flag=0;
    for(j=1; j<=nf; j++)
    {
        if(page[i]==fr[j])
        {
            flag=1;
            break;
        }
    }
   if(flag==0)
    {
        p++;
        for(j=1; j<=nf; j++)
        {
            for(k=i+1; k<=psz; k++)
            {
                if(fr[j]==page[k])
                {
                    u[j]=k;
                    break;
                    }
                    else
                        u[j]=21;
                }
            }
            for(j=1; j<=nf; j++)
                t[j]=u[j];
            for(j=1; j<=nf; j++)
            {
                for(k=j+1; k<=nf; k++)
                {
                    if(t[j]<t[k])
                    {
                        t1=t[j];
                        t[j]=t[k];
                        t[k]=t1;
                    }
                }
            }
            for(j=1; j<=nf; j++)
            {
                if(t[1]==u[j])
                {
                    fr[j]=page[i];
                    u[j]=i;
                }
            }
            printf("page fault\t");
        }
        else
            printf("          \t");
        printf("%d:\t",page[i]);
        for(j=1; j<=nf; j++)
            printf(" %d ",fr[j]);
        printf("\n");
    }
    printf("\ntotal page faults:  %d",p+3);
// getch();
}
int full(int a[],int n)
{
    int k;
    for(k=1; k<=n; k++)
    {
        if(a[k]==-1)
            return 0;
    }
    return 1;
}
```

Sample Output:

```
enter the number of frames:5

 enter the page size2

enter the page sequence:1
2
                1:      1   -1   -1   -1   -1
                2:      1  2    -1   -1   -1

total page faults:  3
```

Conclusion:

Thus the C program to implement the different page replacement algorithms are executed and outputs are verified.

EX.NO: 9 IMPLEMENTATION OF DISK SCHEDULING ALGORITHMS

Objectives:

To write a C program to implement Disk Scheduling Algorithm.

Learning Outcomes:

After the completion of this experiment, student will be able to

- Understand how the disk are scheduled.
- Learn about the various disk scheduling algorithms.

Problem Statement:

The program to implement various disk scheduling algorithms.

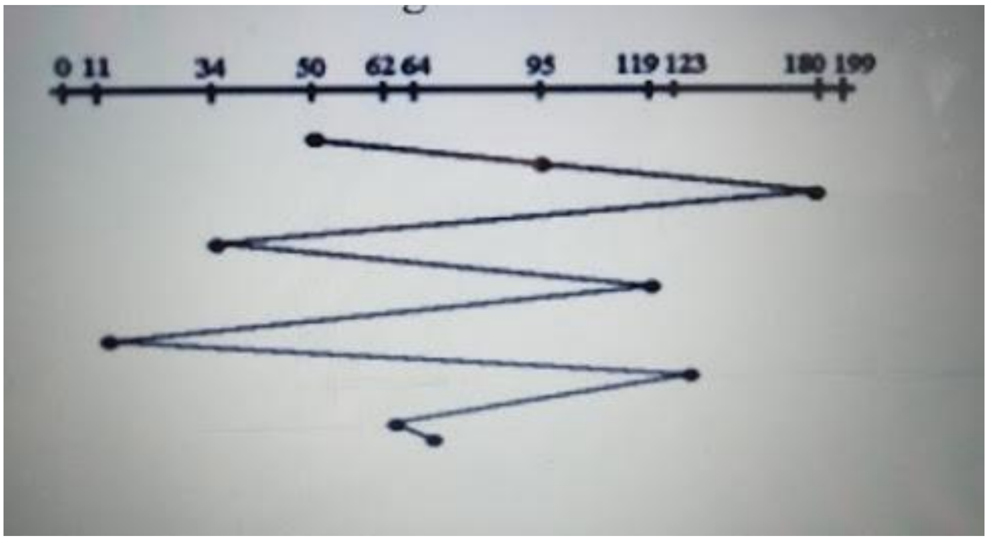
9.1 FCFS SCHEDULING ALGORITHM

Algorithm :

1. Let Request array represents an array storing indexes of tracks that have been requested in ascending order of their time of arrival. 'head' is the position of disk head.
2. Let us one by one take the tracks in default order and calculate the absolute distance of the track from the head.
3. Increment the total seek count with this distance.
4. Currently serviced track position now becomes the new head position.
5. Go to step 2 until all tracks in request array have not been serviced.

Sample Coding:

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
int RQ[100],i,n,TotalHeadMoment=0,initial;
printf("Enter the number of Requests\n");
scanf("%d",&n);
printf("Enter the Requests sequence\n");
for(i=0;i<n;i++)
scanf("%d",&RQ[i]);
printf("Enter initial head position\n");
scanf("%d",&initial);
// logic for FCFS disk scheduling
for(i=0;i<n;i++)
{
TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
initial=RQ[i];
}
printf("Total head moment is %d",TotalHeadMoment);
return 0;
}
```



Sample Output:

```
Enter the number of Request
8
Enter the Requests Sequence
95 180 34 119 11 123 62 64
Enter initial head position
50
Total head movement is 644
```

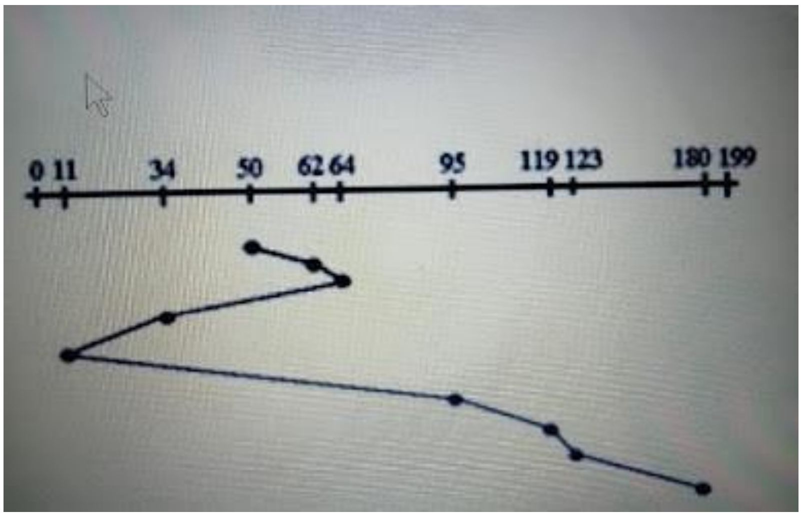
9.2 SSTF (SHORTEST SEEK TIME FIRST) ALGORITHM

Algorithm :

1. Let Request array represents an array storing indexes of tracks that have been requested. ‘head’ is the position of disk head.
2. Find the positive distance of all tracks in the request array from head.
3. Find a track from requested array which has not been accessed/serviced yet and has minimum distance from head.
4. Increment the total seek count with this distance.
5. Currently serviced track position now becomes the new head position.
6. Go to step 2 until all tracks in request array have not been serviced.

Sample Coding:

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
int RQ[100],i,n,TotalHeadMoment=0,initial,count=0;
printf("Enter the number of Requests\n");
scanf("%d",&n);
printf("Enter the Requests sequence\n");
for(i=0;i<n;i++)
scanf("%d",&RQ[i]);
printf("Enter initial head position\n");
scanf("%d",&initial);
// logic for sstf disk scheduling
/* loop will execute until all process is completed*/
while(count!=n)
{
int min=1000,d,index;
for(i=0;i<n;i++)
{
d=abs(RQ[i]-initial);
if(min>d)
{
min=d;
index=i;
}
}
TotalHeadMoment=TotalHeadMoment+min;
initial=RQ[index];
// 1000 is for max
// you can use any number
RQ[index]=1000;
count++;
}
printf("Total head movement is %d",TotalHeadMoment);
return 0;
}
```



Sample Output:

```
Enter the number of Request
8
Enter Request Sequence
95 180 34 119 11 123 62 64
Enter initial head Position
50
Total head movement is 236
```

9.3 SCAN SCHEDULING ALGORITHM

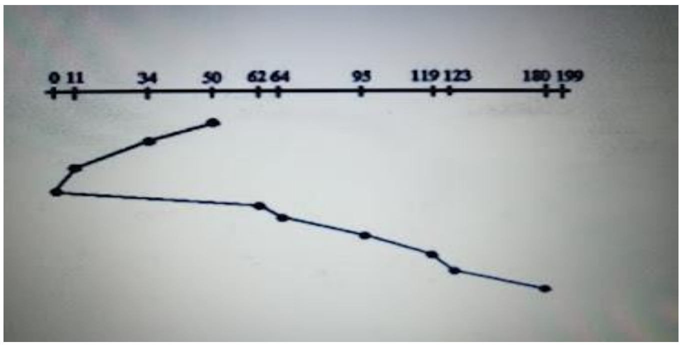
Algorithm :

1. Let Request array represents an array storing indexes of tracks that have been requested in ascending order of their time of arrival. ‘head’ is the position of disk head.
2. Let direction represents whether the head is moving towards left or right.
3. In the direction in which head is moving service all tracks one by one.
4. Calculate the absolute distance of the track from the head.
5. Increment the total seek count with this distance.
6. Currently serviced track position now becomes the new head position.
7. Go to step 3 until we reach at one of the ends of the disk.
8. If we reach at the end of the disk reverse the direction and go to step 2 until all tracks in request array have not been serviced.

Sample Coding:

```
# include<stdio.h>
#include<stdlib.h>
int main()
{
int RQ[100],i,j,n,TotalHeadMoment=0,initial,size,move;
printf("Enter the number of Requests\n");
scanf("%d",&n);
printf("Enter the Requests sequence\n");
for(i=0;i<n;i++)
scanf("%d",&RQ[i]);
printf("Enter initial head position\n");
scanf("%d",&initial);
printf("Enter total disk size\n");
scanf("%d",&size);
printf("Enter the head movement direction for high 1 and for low 0\n");
scanf("%d",&move);
// logic for Scan disk scheduling
/*logic for sort the request array */
for(i=0;i<n;i++)
{
for(j=0;j<n-i-1;j++)
{
if(RQ[j]>RQ[j+1])
{
int temp;
temp=RQ[j];
RQ[j]=RQ[j+1];
RQ[j+1]=temp;
}
}
}
int index;
for(i=0;i<n;i++)
{
if(initial<RQ[i])
{
```

```
index=i;
break;
}
}
// if movement is towards high value
if(move==1)
{
for(i=index;i<n;i++)
{
TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
initial=RQ[i];
}
// last movement for max size
TotalHeadMoment=TotalHeadMoment+abs(size-RQ[i-1]-1);
initial = size-1;
for(i=index-1;i>=0;i--)
{
TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
initial=RQ[i];
}
}
// if movement is towards low value
else
{
for(i=index-1;i>=0;i--)
{
TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
initial=RQ[i];
}
// last movement for min size
TotalHeadMoment=TotalHeadMoment+abs(RQ[i+1]-0);
initial =0;
for(i=index;i<n;i++)
{
TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
initial=RQ[i];
}
}
printf("Total head movement is %d",TotalHeadMoment);
return 0;
}
```



Sample Output:

```
Enter the number of Request
8
Enter the Requests Sequence
95 180 34 119 11 123 62 64
Enter initial head position
50
Enter total disk size
200
Enter the head movement direction for high 1 and for low 0
1
Total head movement is 337
```

9.4 C-SCAN SCHEDULING ALGORITHM

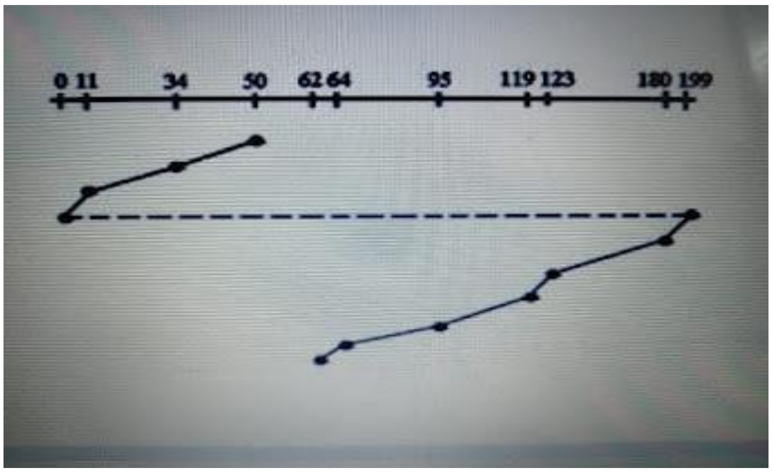
Algorithm :

1. Let Request array represents an array storing indexes of tracks that have been requested in ascending order of their time of arrival. ‘head’ is the position of disk head.
2. The head services only in the right direction from 0 to the size of the disk.
3. While moving in the left direction do not service any of the tracks.
4. When we reach the beginning(left end) reverse the direction.
5. While moving in the right direction it services all tracks one by one.
6. While moving in the right direction calculate the absolute distance of the track from the head.
7. Increment the total seek count with this distance.
8. Currently serviced track position now becomes the new head position.
9. Go to step 6 until we reach the right end of the disk.
10. If we reach the right end of the disk reverse the direction and go to step 3 until all tracks in the request array have not been serviced.

Sample Coding:

```
# include<stdio.h>
#include<stdlib.h>
int main()
{
int RQ[100],i,j,n,TotalHeadMoment=0,initial,size,move;
printf("Enter the number of Requests\n");
scanf("%d",&n);
printf("Enter the Requests sequence\n");
for(i=0;i<n;i++)
scanf("%d",&RQ[i]);
printf("Enter initial head position\n");
scanf("%d",&initial);
printf("Enter total disk size\n");
scanf("%d",&size);
printf("Enter the head movement direction for high 1 and for low 0\n");
scanf("%d",&move);
// logic for C-Scan disk scheduling
/*logic for sort the request array */
for(i=0;i<n;i++)
```

```
{
for( j=0;j<n-i-1;j++)
{
if(RQ[j]>RQ[j+1])
{
int temp;
temp=RQ[j];
RQ[j]=RQ[j+1];
RQ[j+1]=temp;
}
}
}
int index;
for(i=0;i<n;i++)
{
if(initial<RQ[i])
{
index=i;
break;
}
}
// if movement is towards high value
if(move==1)
{
for(i=index;i<n;i++)
{
TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
initial=RQ[i];
}
// last movement for max size
TotalHeadMoment=TotalHeadMoment+abs(size-RQ[i-1]-1);
/*movement max to min disk */
TotalHeadMoment=TotalHeadMoment+abs(size-1-0);
initial=0;
for( i=0;i<index;i++)
{
TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
initial=RQ[i];
}
}
// if movement is towards low value
else
{
for(i=index-1;i>=0;i--)
{
TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
initial=RQ[i];
}
// last movement for min size
TotalHeadMoment=TotalHeadMoment+abs(RQ[i+1]-0);
/*movement min to max disk */
TotalHeadMoment=TotalHeadMoment+abs(size-1-0);
initial =size-1;
for(i=n-1;i>=index;i--)
{
TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
initial=RQ[i];
}
}
printf("Total head movement is %d",TotalHeadMoment);
return 0;
}
```



Sample Output:

```
Enter the number of Request
8
Enter the Requests Sequence
95 180 34 119 11 123 62 64
Enter initial head position
50
Enter total disk size
200
Enter the head movement direction for high 1 and for low 0
1
Total head movement is 382
```

9.5 LOOK DISK SCHEDULING ALGORITHM

Algorithm :

1. Let Request array represents an array storing indexes of tracks that have been requested in ascending order of their time of arrival. ‘head’ is the position of disk head.
2. The initial direction in which head is moving is given and it services in the same direction.
3. The head services all the requests one by one in the direction head is moving.
4. The head continues to move in the same direction until all the request in this direction are finished.
5. While moving in this direction calculate the absolute distance of the track from the head.

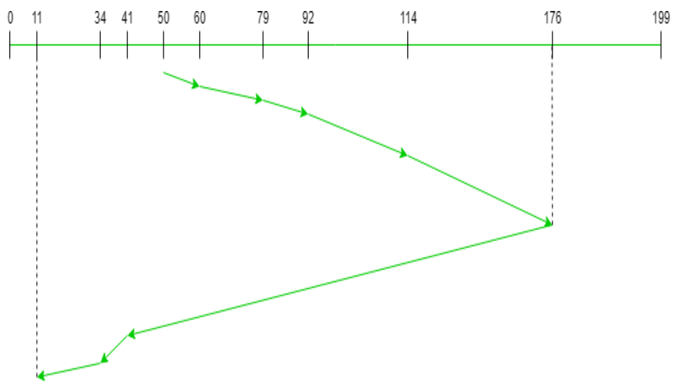
6. Increment the total seek count with this distance.
7. Currently serviced track position now becomes the new head position.
8. Go to step 5 until we reach at last request in this direction.
9. If we reach where no requests are needed to be serviced in this direction reverse the direction and go to step 3 until all tracks in request array have not been serviced.

Sample Coding:

```
# include<stdio.h>
#include<stdlib.h>
int main()
{
int RQ[100],i,j,n,TotalHeadMoment=0,initial,size,move;
printf("Enter the number of Requests\n");
scanf("%d",&n);
printf("Enter the Requests sequence\n");
for(i=0;i<n;i++)
scanf("%d",&RQ[i]);
printf("Enter initial head position\n");
scanf("%d",&initial);
printf("Enter total disk size\n");
scanf("%d",&size);
printf("Enter the head movement direction for high 1 and for low 0\n");
scanf("%d",&move);
// logic for look disk scheduling
/*logic for sort the request array */
for(i=0;i<n;i++)
{
for(j=0;j<n-i-1;j++)
{
if(RQ[j]>RQ[j+1])
{
int temp;
temp=RQ[j];
RQ[j]=RQ[j+1];
RQ[j+1]=temp;
}
}}

int index;
for(i=0;i<n;i++)
{
if(initial<RQ[i])
{
index=i;
break;
}
}
// if movement is towards high value
if(move==1)
{
for(i=index;i<n;i++)
{
```

```
TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
initial=RQ[i];
}
for(i=index-1;i>=0;i--)
{
TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
initial=RQ[i];
}
}
// if movement is towards low value
else
{
for(i=index-1;i>=0;i--)
{
TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
initial=RQ[i];
}
for(i=index;i<n;i++)
{
TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
initial=RQ[i];
}
}
printf("Total head movement is %d",TotalHeadMoment);
return 0;
}
```



Sample Output:

```
Enter the number of Request
8
Enter the Requests Sequence
95 180 34 119 11 123 62 64
Enter initial head position
50
Enter the head movement direction for high 1 and for low 0
1
Total head movement is 299
```

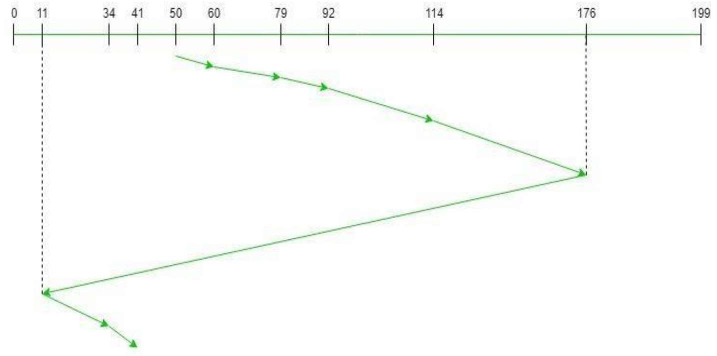
9.6 C-LOOK DISK SCHEDULING ALGORITHM

Algorithm :

1. Let Request array represents an array storing indexes of the tracks that have been requested in ascending order of their time of arrival and **head** is the position of the disk head.
2. The initial direction in which the head is moving is given and it services in the same direction.
3. The head services all the requests one by one in the direction it is moving.
4. The head continues to move in the same direction until all the requests in this direction have been serviced.
5. While moving in this direction, calculate the absolute distance of the tracks from the head.
6. Increment the total seek count with this distance.
7. Currently serviced track position now becomes the new head position.
8. Go to step 5 until we reach the last request in this direction.
9. If we reach the last request in the current direction then reverse the direction and move the head in this direction until we reach the last request that is needed to be serviced in this direction without servicing the intermediate requests.
10. Reverse the direction and go to step 3 until all the requests have not been serviced.

Sample Coding:

```
# include<stdio.h>
#include<stdlib.h>
int main()
{
int RQ[100],i,j,n,TotalHeadMoment=0,initial,size,move;
printf("Enter the number of Requests\n");
scanf("%d",&n);
printf("Enter the Requests sequence\n");
for(i=0;i<n;i++)
scanf("%d",&RQ[i]);
printf("Enter initial head position\n");
scanf("%d",&initial);
printf("Enter total disk size\n");
scanf("%d",&size);
printf("Enter the head movement direction for high 1 and for low 0\n");
scanf("%d",&move);
// logic for C-look disk scheduling
/*logic for sort the request array */
for(i=0;i<n;i++)
{
for( j=0;j<n-i-1;j++)
{
if(RQ[j]>RQ[j+1])
{
int temp;
temp=RQ[j];
RQ[j]=RQ[j+1];
RQ[j+1]=temp;
}}
}
int index;
for(i=0;i<n;i++)
{
if(initial<RQ[i])
{
index=i;
break;
}}
// if movement is towards high value
if(move==1)
{
for(i=index;i<n;i++)
{
TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
initial=RQ[i];
}
for( i=0;i<index;i++)
{
TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
initial=RQ[i];
}}
// if movement is towards low value
else
{
for(i=index-1;i>=0;i--)
{
TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
initial=RQ[i];
}
for(i=n-1;i>=index;i--)
{
TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
initial=RQ[i];
}}
printf("Total head movement is %d",TotalHeadMoment);
return 0;
}
```



Sample Output:

```
Enter the number of Request
8
Enter the Requests Sequence
95 180 34 119 11 123 62 64
Enter initial head position
50
Enter the head movement direction for high 1 and for low 0
1
Total head movement is 322
```

Conclusion:

Thus the C program to implement the different disk scheduling algorithms are executed and outputs are verified.

EX.NO: 10 IMPLEMENTATION OF FILE ORGANIZATION TECHNIQUES

Objectives:

To write a C program to implement different file organization techniques available in operating system.

Learning Outcomes:

After the completion of this experiment, student will be able to

- Understand how the files are organized.
- Know how the continuous and non-continuous memory spaces are allocated.
- Learn about the different file allocation methods used.

Problem Statement:

The program to implement different file allocation techniques in operating system.

10.1 CONTIGUOUS (SEQUENTIAL) FILE ALLOCATION ALGORITHM

Algorithm:

```
Step 1: Start the program.
Step 2: Get the number of memory partition and their sizes.
Step 3: Get the number of processes and values of block size for each process.
Step 4: First fit algorithm searches all the entire memory block until a hole which is big enough
        is encountered. It allocates that memory block for the requesting process.
Step 5: Best-fit algorithm searches the memory blocks for the smallest hole which can be
        allocated to requesting process and allocates if.
Step 6: Worst fit algorithm searches the memory blocks for the largest hole and allocates it to
        the process.
Step 7: Analyses all the three memory management techniques and display the best algorithm
        which utilizes the memory resources effectively and efficiently.
Step 8: Stop the program
```

Sample Coding:

```
#include<stdio.h>
#include<conio.h>
main()
{
int n,i,j,b[20],sb[20],t[20],x,c[20][20];
clrscr();
printf("Enter no.of files:");
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("Enter no. of blocks occupied by file%d",i+1);
scanf("%d",&b[i]);
printf("Enter the starting block of file%d",i+1);
scanf("%d",&sb[i]);
t[i]=sb[i];
for(j=0;j<b[i];j++)
c[i][j]=sb[i]++;
}
printf("Filename\tStart block\tlength\n");
for(i=0;i<n;i++)
printf("%d\t %d \t%d\n",i+1,t[i],b[i]);
printf("Enter file name:");
scanf("%d",&x);
printf("File name is:%d",x);
printf("length is:%d",b[x-1]);
printf("blocks occupied:");
for(i=0;i<b[x-1];i++)
printf("%4d",c[x-1][i]);
getch();
}
```

Sample Output:

```
Enter no.of files: 2
Enter no. of blocks occupied by file1 4
Enter the starting block of file1 2
Enter no. of blocks occupied by file2 10
Enter the starting block of file2 5
Filename Start block length
1 2 4
2 5 10
Enter file name: rajesh
File name is: rajesh
length is: 12803
blocks occupied : 0
```

10.2 LINKED FILE ALLOCATION ALGORITHM

Algorithm:

```
Step 1: Create a queue to hold all pages in memory
Step 2: When the page is required replace the page at the head of the queue
Step 3: Now the new page is inserted at the tail of the queue
Step 4: Create a stack
Step 5: When the page fault occurs replace page present at the bottom of the stack
Step 6: Stop the allocation.
```

Sample Coding:

```
#include<stdio.h>
#include<conio.h>
struct file
{
char fname[10];
int start,size,block[10];
}f[10];
main()
{
int i,j,n;
clrscr();
printf("Enter no. of files:");
```

```
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
    printf("Enter file name:");
    scanf("%s",&f[i].fname);
    printf("Enter starting block:");
    scanf("%d",&f[i].start);
    f[i].block[0]=f[i].start;
    printf("Enter no.of blocks:");
    scanf("%d",&f[i].size);
    printf("Enter block numbers:");
    for(j=1;j<=f[i].size;j++)
    {
    scanf("%d",&f[i].block[j]);
    }
    }
    printf("File\tstart\tsize\tblock\n");
for(i=0;i<n;i++)
    {
    printf("%s\t%d\t%d\t",f[i].fname,f[i].start,f[i].size);
    for(j=1;j<=f[i].size-1;j++)
    printf("%d--->",f[i].block[j]);
    printf("%d",f[i].block[j]);
    printf("\n");
    }
    getch();
    }
```

Sample Output:

```
    Enter no. of files:2
    Enter file name:venkat
    Enter starting block:20
    Enter no.of blocks:6
    Enter block numbers: 4
    12
    15
    45
    32
    25
    Enter file name:rajesh
    Enter starting block:12
    Enter no.of blocks:5
    Enter block numbers:6
    5
    4
    3
    2
    File start size block
    venkat 20 6 4--->12--->15--->45--->32--->25
    rajesh 12 5 6--->5--->4--->3--->2
```

10.3 INDEXED FILE ALLOCATION ALGORITHM

Algorithm:

```
Step 1: Start.
Step 2: Let n be the size of the buffer
Step 3: check if there are any producer
Step 4: if yes check whether the buffer is full
Step 5: If no the producer item is stored in the buffer
Step 6: If the buffer is full the producer has to wait
Step 7: Check there is any cosumer. If yes check whether the buffer is empty
Step 8: If no the consumer consumes them from the buffer
Step 9: If the buffer is empty, the consumer has to wait.
Step 10: Repeat checking for the producer and consumer till required
Step 11: Terminate the process.
```

Sample Coding:

```
#include<stdio.h>
#include<conio.h>
main()
{
int n,m[20],i,j,sb[20],s[20],b[20][20],x;
clrscr();
printf("Enter no. of files:");
scanf("%d",&n);
for(i=0;i<n;i++)
{ printf("Enter starting block and size of file%d:",i+1);
scanf("%d%d",&sb[i],&s[i]);
printf("Enter blocks occupied by file%d:",i+1);
scanf("%d",&m[i]);
printf("enter blocks of file%d:",i+1);
for(j=0;j<m[i];j++)
scanf("%d",&b[i][j]);
} printf("\nFile\t index\tlength\n");
for(i=0;i<n;i++)
{
printf("%d\t%d\t%d\n",i+1,sb[i],m[i]);
}printf("\nEnter file name:");
scanf("%d",&x);
printf("file name is:%d\n",x);
i=x-1;
printf("Index is:%d",sb[i]);
printf("Block occupied are:");
for(j=0;j<m[i];j++)
printf("%3d",b[i][j]);
getch();
}
```

Sample Output:

```
Enter no. of files:2
Enter starting block and size of file1: 2 5
Enter blocks occupied by file1:10
enter blocks of file1:3
```

```
2 5 4 6 7 2 6 4 7
Enter starting block and size of file2: 3 4
Enter blocks occupied by file2:5
enter blocks of file2: 2 3 4 5 6
File index length
1 2 10
2 3 5
Enter file name: venkat
file name is: venkat
Index is: 12803
Block occupied are: 0
```

Conclusion:

Thus the C program to implement the different file allocation techniques are executed and outputs are verified.