

**OPERATING SYSTEMS**

**LABORATORY MANUAL**

**DEPARTMENT OF COMPUTER SCIENCE  
AND ENGINEERING**

## DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

### VISION

To improve the quality of technical education that provides efficient software engineers with an attitude to adapt challenging IT needs of local, national and international arena, through teaching and interaction with alumni and industry.

### MISSION

Department intends to meet the contemporary challenges in the field of IT and is playing a vital role in shaping the education of the 21st century by providing unique educational and research opportunities.

## **PROGRAMME EDUCATIONAL OBJECTIVES (PEOs)**

### **PEO1 - ANALYTICAL SKILLS**

To facilitate the graduates with the ability to visualize, gather information, articulate, analyze, solve complex problems, and make decisions. These are essential to address the challenges of complex and computation intensive problems increasing their productivity.

### **PEO2 - TECHNICAL SKILLS**

To facilitate the graduates with the technical skills that prepare them for immediate employment and pursue certification providing a deeper understanding of the technology in advanced areas of computer science and related fields, thus encouraging to pursue higher education and research based on their interest.

### **PEO3 - SOFT SKILLS**

To facilitate the graduates with the soft skills that include fulfilling the mission, setting goals, showing self-confidence by communicating effectively, having a positive attitude, get involved in team- work, being a leader, managing their career and their life.

### **PEO4 - PROFESSIONAL ETHICS**

To facilitate the graduates with the knowledge of professional and ethical responsibilities by paying attention to grooming, being conservative with style, following dress codes, safety codes, and adapting themselves to technological advancements.

### PROGRAM SPECIFIC OUTCOMES (PSOs)

After the completion of the course, B. Tech Information Technology, the graduates will have the following Program Specific Outcomes:

- 1. Fundamentals and critical knowledge of the Computer System:-** Able to Understand the working principles of the computer System and its components , Apply the knowledge to build, asses, and analyze the software and hardware aspects of it .
- 2. The comprehensive and Applicative knowledge of Software Development:** Comprehensive skills of Programming Languages, Software process models, methodologies, and able to plan, develop, test, analyze, and manage the software and hardware intensive systems in heterogeneous platforms individually or working in teams.
- 3. Applications of Computing Domain & Research:** Able to use the professional, managerial, interdisciplinary skill set, and domain specific tools in development processes, identify the research gaps, and provide innovative solutions to them.



## PROGRAM OUTCOMES (POs)

**Engineering Graduates should possess the following:**

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

## OBJECTIVES:

- To understand the functionalities of various layers of OSI model
- To explain the difference between hardware, software; operating systems, programs and files.
- Identify the purpose of different software applications.

**Week 1:** Simulate the following CPU scheduling algorithms.

a) FCFS      b) SJF      c) Round Robin      d) Priority.

**Week 2:** Write a C program to simulate producer-consumer problem using Semaphores

**Week 3:** Write a C program to simulate the concept of Dining-philosophers problem.

**Week 4:** Simulate MVT and MFT.

**Week 5:** Write a C program to simulate the following contiguous memory allocation Techniques

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

**Week 6:** Simulate all page replacement algorithms a) FIFO      b) LRU

c) OPTIMAL

**Week 7:** Simulate all File Organization Techniques

a) Single level directory   b) Two level      directory

**Week 8:** Simulate all file allocation strategies

a) Sequential   b) Indexed   c) Linked.

**Week 9:** Simulate Bankers Algorithm for Dead Lock

Avoidance. **Week 10:** Simulate Bankers Algorithm for Dead Lock Prevention. **Week 11:** Write a C program to simulate disk scheduling algorithms.

a) FCFS      b) SCAN      c) C-SCAN

## REFERENCE BOOKS:

1. An Introduction to Operating Systems, P.C.P Bhatt, 2nd edition, PHI.
2. Modern Operating Systems, Andrew S Tanenbaum, 3rd Edition, PHI

## OUTCOMES:

At the end of the course the students are able to:

- Ability to implement inter process communication between two processes.
- Ability to design and solve synchronization problems.
- Ability to simulate and implement operating system concepts such as scheduling, Deadlock management, file management, and memory management.



# THE NEOTIA UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Operating Systems Lab Manual

## TABLE OF CONTENTS

EXP.NO	NAME OF THE EXPERIMENT	PAGE.NO
1	<b>CPU SCHEDULING ALGORITHMS</b>	
	A) FIRST COME FIRST SERVE(FCFS)	1-3
	B) SHORTEST JOB FIRST(SJF)	4-6
	C) ROUND ROBIN	7-9
	D) PRIORITY	10-12
2	PRODUCER-CONSUMER PROBLEM USING SEMAPHORES	13-14
3	DINING-PHILOSOPHERS PROBLEM	15-18
4	<b>MEMORY MANAGEMENT TECHNIQUES</b>	
	A) MULTI PROGRAMMING WITH FIXED NUMBER OF TASKS(MFT)	19-21
	B) MULTI PROGRAMMING WITH VARIABLE NUMBER OF TASKS(MVT)	22-24
5	<b>CONTIGUOUS MEMORY ALLOCATION</b>	
	A) WORST FIT	25-26
	B) BEST FIT	27-28
	C) FIRST FIT	28-29
6	<b>PAGE REPLACEMENT ALGORITHMS</b>	
	A) FIRST IN FIRST OUT(FIFO)	30-32
	B) LEAST RECENTLY USED(LRU)	33-35
	C) OPTIMAL	36-39
7	<b>FILE ORGANIZATION TECHNIQUES</b>	
	A) SINGLE LEVEL DIRECTORY	40-42
	B) TWO LEVEL DIRECTORY	43-46
8	<b>FILE ALLOCATION STRATEGIES</b>	
	A) SEQUENTIAL	47-49
	B) INDEXED	50-52
	C) LINKED	53-55
9	DEAD LOCK AVOIDANCE	56-59
10	DEAD LOCK PREVENTION	60-62
11	<b>DISK SCHEDULING ALGORITHMS</b>	
	A) FCFS	63-64
	B) SCAN	65-66
	C) C-SCAN	67-69

## EXPERIMENT NO.1

### CPU SCHEDULING ALGORITHMS

#### **A). FIRST COME FIRST SERVE:**

**AIM:** To write a c program to simulate the CPU scheduling algorithm First Come First Serve (FCFS)

#### **DESCRIPTION:**

To calculate the average waiting time using the FCFS algorithm first the waiting time of the first process is kept zero and the waiting time of the second process is the burst time of the first process and the waiting time of the third process is the sum of the burst times of the first and the second process and so on. After calculating all the waiting times the average waiting time is calculated as the average of all the waiting times. FCFS mainly says first come first serve the algorithm which came first will be served first.

#### **ALGORITHM:**

Step 1: Start the process

Step 2: Accept the number of processes in the ready Queue

Step 3: For each process in the ready Q, assign the process name and the burst time

Step 4: Set the waiting of the first process as 0 and its burst time as its turnaround time

Step 5: for each process in the Ready Q calculate

a).  $\text{Waiting time (n)} = \text{waiting time (n-1)} + \text{Burst time (n-1)}$

$\text{Turnaround time (n)} = \text{waiting time(n)} + \text{Burst time(n)}$

Step 6: Calculate

a)  $\text{Average waiting time} = \text{Total waiting Time} / \text{Number of process}$

b)  $\text{Average Turnaround time} = \text{Total Turnaround Time} / \text{Number of}$

process

Step 7: Stop the process

### SOURCE CODE:

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



### INPUT

Enter the number of processes -- 3  
Enter Burst Time for Process 0 -- 24  
Enter Burst Time for Process 1 -- 3  
Enter Burst Time for Process 2 -- 3

### OUTPUT

PROCESS	BURST TIME	WAITING TIME	TURNAROUND TIME
P0	24	0	24
P1	3	24	27
P2	3	27	30
Average Waiting Time--		17.000000	
Average Turnaround Time --		27.000000	

## **B). SHORTEST JOB FIRST:**

**AIM:** To write a program to stimulate the CPU scheduling algorithm Shortest job first (Non- Preemption)

### **DESCRIPTION:**

To calculate the average waiting time in the shortest job first algorithm the sorting of the process based on their burst time in ascending order then calculate the waiting time of each process as the sum of the bursting times of all the process previous or before to that process.

### **ALGORITHM:**

Step 1: Start the process

Step 2: Accept the number of processes in the ready Queue

Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time

Step 4: Start the Ready Q according the shortest Burst time by sorting according to lowest to highest burst time.

Step 5: Set the waiting time of the first process as  $_0$  and its turnaround time as its burst time.

Step 6: Sort the processes names based on their Burt time

Step 7: For each process in the ready queue, calculate

a)  $\text{Waiting time}(n) = \text{waiting time}(n-1) + \text{Burst time}(n-1)$

b)  $\text{Turnaround time}(n) = \text{waiting time}(n) + \text{Burst time}(n)$

Step 8: Calculate

c)  $\text{Average waiting time} = \text{Total waiting Time} / \text{Number of process}$

d)  $\text{Average Turnaround time} = \text{Total Turnaround Time} / \text{Number of process}$

Step 9: Stop the process

### SOURCE CODE :

```
#include<stdio.h>
#include<conio.h>
main()
{
int p[20], bt[20], wt[20], tat[20], i, k, n, temp; float wtavg,
tatavg;
clrscr();
printf("\nEnter the number of processes -- ");
scanf("%d", &n);
for(i=0;i<n;i++)
{
p[i]=i;
printf("Enter Burst Time for Process %d -- ", i);
scanf("%d", &bt[i]);

}
for(i=0;i<n;i++)
for(k=i+1;k<n;k++)
if(bt[i]>bt[k])
{
temp=bt[i];
bt[i]=bt[k];
bt[k]=temp;

temp=p[i];
p[i]=p[k];
p[k]=temp;
}
wt[0] = wtavg = 0;
tat[0] = tatavg = bt[0]; for(i=1;i<n;i++)
{
wt[i] = wt[i-1] +bt[i-1];
tat[i] = tat[i-1] +bt[i];
wtavg = wtavg + wt[i];
tatavg = tatavg + tat[i];
}
printf("\n\t PROCESS \tBURST TIME \t WAITING TIME\t TURNAROUND TIME\n");
for(i=0;i<n;i++)
printf("\n\t P%d \t\t %d \t\t %d \t\t %d", p[i], bt[i], wt[i], tat[i]);
printf("\nAverage Waiting Time -- %f", wtavg/n);
printf("\nAverage Turnaround Time -- %f", tatavg/n); getch();}
```

### INPUT

Enter the number of processes -- 4  
Enter Burst Time for Process 0 -- 6  
Enter Burst Time for Process 1 -- 8  
Enter Burst Time for Process 2 -- 7  
Enter Burst Time for Process 3 -- 3

### OUTPUT

PROCESS	BURST TIME	WAITING TIME	TURNARO UND TIME
P3	3	0	3
P0	6	3	9
P2	7	9	16
P1	8	16	24
Average Waiting Time --		7.000000	
Average Turnaround Time --		13.000000	

C).

**ROUND  
ROBIN:**

**AIM:** To simulate the CPU scheduling algorithm round-robin.

**DESCRIPTION:**

To aim is to calculate the average waiting time. There will be a time slice, each process should be executed within that time-slice and if not it will go to the waiting state so first check whether the burst time is less than the time-slice. If it is less than it assign the waiting time to the sum of the total times. If it is greater than the burst-time then subtract the time slot from the actual burst time and increment it by time-slot and the loop continues until all the processes are completed.

**ALGORITHM:**

Step 1: Start the process

Step 2: Accept the number of processes in the ready Queue and time quantum (or) time slice

Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time

Step 4: Calculate the no. of time slices for each process where No. of time slice for process (n) = burst time process (n)/time slice

Step 5: If the burst time is less than the time slice then the no. of time slices =1.

Step 6: Consider the ready queue is a circular Q, calculate

- a) Waiting time for process (n) = waiting time of process(n-1)+ burst time of process(n-1) + the time difference in getting the CPU from process(n-1)
- b) Turnaround time for process(n) = waiting time of process(n) + burst time of process(n)+ the time difference in getting CPU from process(n).

Step 7: Calculate

- c) Average waiting time = Total waiting Time / Number of process
- d) Average Turnaround time = Total Turnaround Time / Number of process

Step 8: Stop the process



### SOURCE CODE

```
#include<stdio.h>
main()
{
int
i,j,n,bu[10],wa[10],tat[10],t,ct[10],max; float
awt=0,att=0,temp=0;
clrscr();
printf("Enter the no of processes -- ");
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("\nEnter Burst Time for process %d -- ", i+1);
scanf("%d",&bu[i]);
ct[i]=bu[i];
}
printf("\nEnter the size of time slice -- ");
scanf("%d",&t);
max=bu[0];
for(i=1;i<n;i++)
if(max<bu[i])
max=bu[i];
for(j=0;j<((max/t)+1);j++)
for(i=0;i<n;i++)
if(bu[i]!=0)
if(bu[i]<=t) {
tat[i]=temp+bu[i];
temp=temp+bu[i];
bu[i]=0;
}
else {
bu[i]=bu[i]-t;
temp=temp+t;
}
for(i=0;i<n;i++){
wa[i]=tat[i]-
ct[i]; att+=tat[i];
awt+=wa[i];}
printf("\nThe Average Turnaround time is -- %f",att/n);
printf("\nThe Average Waiting time is -- %f ",awt/n);
printf("\n\tPROCESS\t BURST TIME \t WAITING TIME\tTURNAROUND TIME\n");
for(i=0;i<n;i++)
printf("\t%d \t %d \t\t %d \t\t %d \n",i+1,ct[i],wa[i],tat[i]);
getch();}
```

**INPUT:**

Enter the no of processes – 3

Enter Burst Time for process 1 – 24

Enter Burst Time for process 2 -- 3

Enter Burst Time for process 3 – 3

Enter the size of time slice – 3

**OUTPUT:**

PROCESS	BURST TIME	WAITING TIME	TURNAROUND TIME
1	24	6	30
2	3	4	7
3	3	7	10

The Average Turnaround time is – 15.666667 The

Average Waiting time is 5.666667

#### **D). PRIORITY:**

**AIM:** To write a c program to simulate the CPU scheduling priority algorithm.

#### **DESCRIPTION:**

To calculate the average waiting time in the priority algorithm, sort the burst times according to their priorities and then calculate the average waiting time of the processes. The waiting time of each process is obtained by summing up the burst times of all the previous processes.

#### **ALGORITHM:**

Step 1: Start the process

Step 2: Accept the number of processes in the ready Queue

Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time

Step 4: Sort the ready queue according to the priority number.

Step 5: Set the waiting of the first process as `_0'` and its burst time as its turnaround time

Step 6: Arrange the processes based on process priority

Step 7: For each process in the Ready Q calculate Step 8:

for each process in the Ready Q calculate

a)  $\text{Waiting time}(n) = \text{waiting time}(n-1) + \text{Burst time}(n-1)$

b)  $\text{Turnaround time}(n) = \text{waiting time}(n) + \text{Burst time}(n)$

Step 9: Calculate

c)  $\text{Average waiting time} = \text{Total waiting Time} / \text{Number of process}$

d)  $\text{Average Turnaround time} = \text{Total Turnaround Time} / \text{Number of process}$  Print the results in an order.

Step10: Stop

### SOURCE CODE:

```
#include<stdio.h>
main()
{
int p[20],bt[20],pri[20], wt[20],tat[20],i, k, n, temp; float wtavg,
tatavg;
clrscr();
printf("Enter the number of processes --- ");
scanf("%d",&n);
for(i=0;i<n;i++){
p[i] = i;
printf("Enter the Burst Time & Priority of Process %d --- ",i); scanf("%d
%d",&bt[i], &pri[i]);
}
for(i=0;i<n;i++)
for(k=i+1;k<n;k++)
if(pri[i] > pri[k]){
temp=p[i];
p[i]=p[k];
p[k]=temp;
temp=bt[i];
bt[i]=bt[k];
bt[k]=temp;
temp=pri[i];
pri[i]=pri[k];
pri[k]=temp;
}
wtavg = wt[0] = 0;
tatavg = tat[0] = bt[0];
for(i=1;i<n;i++)
{
wt[i] = wt[i-1] + bt[i-1];
tat[i] = tat[i-1] + bt[i];

wtavg = wtavg + wt[i];
tatavg = tatavg + tat[i];
}
printf("\nPROCESS\t\tPRIORITY\tBURST TIME\tWAITING TIME\tTURNAROUND
TIME");
for(i=0;i<n;i++)
printf("\n%d \t\t %d \t\t %d \t\t %d \t\t %d ",p[i],pri[i],bt[i],wt[i],tat[i]);
printf("\nAverage Waiting Time is --- %f",wtavg/n); printf("\nAverage
Turnaround Time is --- %f",tatavg/n);
getch();}
```

### INPUT

Enter the number of processes -- 5  
Enter the Burst Time & Priority of Process 0 --- 10  
Enter the Burst Time & Priority of Process 1 --- 1  
Enter the Burst Time & Priority of Process 2 --- 2  
Enter the Burst Time & Priority of Process 3 --- 1  
Enter the Burst Time & Priority of Process 4 --- 5

3  
1  
4  
5  
2

### OUTPUT

PROCESS	PRIORITY	BURST TIME	WAITING TIME	TURNAROUND TIME
1	1	1	0	1
4	2	5	1	6
0	3	10	6	16
2	4	2	16	18
3	5	1	18	19
Average Waiting Time is ---		8.200000		
Average Turnaround Time is		12.000000		

### VIVA QUESTIONS

- 1) Define the following
  - a) Turnaround time
  - b) Waiting time
  - c) Burst time
  - d) Arrival time
- 2) What is meant by process scheduling?
- 3) What are the various states of process?
- 4) What is the difference between preemptive and non-preemptive scheduling?
- 5) What is meant by time slice?
- 6) What is round robin scheduling?



## EXPERIMENT.NO 2

**AIM:** To Write a C program to simulate producer-consumer problem using semaphores.

### DESCRIPTION

Producer consumer problem is a synchronization problem. There is a fixed size buffer where the producer produces items and that is consumed by a consumer process. One solution to the producer-consumer problem uses shared memory. To allow producer and consumer processes to run concurrently, there must be available a buffer of items that can be filled by the producer and emptied by the consumer. This buffer will reside in a region of memory that is shared by the producer and consumer processes. The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.

### PROGRAM

```
#include<stdio.
> void main()
{
    int buffer[10], bufsize, in, out, produce, consume,
    choice=0; in = 0;
    out = 0;
    bufsize = 10;
    while(choice !=3)
    {
        printf("\n1. Produce \t 2. Consume \t3. Exit");
        printf("\nEnter your choice: ");
        scanf("%d",&choice);
        switch(choice) {
            case 1: if((in+1)%bufsize==out)
                    printf("\nBuffer is Full");
                    else
                    {
                        printf("\nEnter the
                        value: "); scanf("%d",
                        &produce); buffer[in] =
                        produce;
                        in = (in+1)%bufsize;
                    }
                    break;;;
            case 2: if(in == out)
                    printf("\nBuffer is Empty");
                    else
                    {
                        consume = buffer[out];
                        printf("\nThe consumed
                        value is %d", consume); out
                        = (out+1)%bufsize;
                    }
                    break;
        }
    }
}
```

## OUTPUT

1. Produce 2. Consume 3.

Exit Enter your choice: 2

Buffer is Empty

1. Produce 2. Consume 3.

Exit Enter your choice: 1

Enter the value: 100

1. Produce 2. Consume 3.

Exit Enter your choice: 2

The consumed value is 100

1. Produce 2. Consume 3.

Exit Enter your choice: 3

### EXPERIMENT.NO 3

**AIM:** To Write a C program to simulate the concept of Dining-Philosophers problem.

#### DESCRIPTION

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

#### PROGRAM

```
int tph, philname[20], status[20], howhung, hu[20], cho; main()
{
    int i; clrscr();
    printf("\n\nDINING PHILOSOPHER PROBLEM");
    printf("\nEnter the total no. of philosophers: ");
    scanf("%d",&tph);
    for(i=0;i<tph;i++)
    {
        philname[i]=(i+1); status[i]=1;
    }
    printf("How many are hungry :
    "); scanf("%d", &howhung);
    if(howhung==tph)
    {
        printf("\n All are hungry..\nDead lock stage will occur");
        printf("\nExiting\n");
    }
    else{
        for(i=0;i<howhung;i++){
            printf("Enterphilosopher%dposition:",(i+1));
            scanf("%d",&hu[i]);
            status[hu[i]]=2;
        }
    }
}
```

```

do
{
    printf("1.One can eat at a time\t2.Two can eat at a time
\t3.Exit\nEnter your choice:");
    scanf("%d", &cho);
    switch(cho)
    {
        case 1: one();
                                break;
        case 2: two();
                break; case 3:
        exit(0);
                                }
    }while(1);
                                default: printf("\nInvalid
                                option..");
    }
}
one()
{
    int pos=0, x, i;
    printf("\nAllow one philosopher to eat at any time\n");
    for(i=0;i<howhung; i++, pos++)
    {
        printf("\nP %d is granted to eat", philname[hu[pos]]);
        for(x=pos;x<howhung;x++)
        printf("\nP %d is waiting", philname[hu[x]]);
    }
}
two()
{
    int i, j, s=0, t, r, x;
    printf("\n
    Allow two
    philosophers
    to eat at same
    time\n");
    for(i=0;i<how
    hung;i++)
    {
        for(j=i+1;j<howhung;j++)
        {
            if(abs(hu[i]-hu[j])>=1&& abs(hu[i]-hu[j])!=4)
            {
                printf("\ncombination %d \n", (s+1));
            }
        }
    }
}

```

```
t=hu[i];  
r=hu[j]; s++;  
printf("\nP %d and P %d are granted to eat", philname[hu[i]],  
        philname[hu[j]]);
```



```

    }
    {
        }
        {
            for(x=0;x<howhung;x++)
            {
                if((hu[x]!=t)&&(hu[x]!=r))
                printf("\nP %d is waiting", philname[hu[x]]);
            }
        }
    }
}

```

*INPUT*

DINING PHILOSOPHER PROBLEM

E  
n  
t  
e  
r  
t  
h  
e  
t  
o  
t  
a  
l  
n  
o  
.  
o  
f  
p  
h  
i  
l  
o  
s  
o  
p  
h  
e  
r  
s

:  
5  
H  
o  
w  
m  
a  
n  
y  
a  
r  
e

h  
u  
n  
g  
r  
y  
:  
3

Enter philosopher 1 position: 2

Enter philosopher 2 position: 4

Enter philosopher 3 position: 5

### OUTPUT

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

Allow one philosopher to eat at any time P 3 is granted to eat

P 3 is waiting

P 5 is waiting

P 0 is waiting

P 5 is granted to eat

P 5 is waiting

P 0 is waiting

P 0 is granted to eat

P 0 is waiting

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

Allow two philosophers to eat at same time  
combination 1

P 3 and P 5 are granted to eat  
P 0 is waiting

combination 2

P 3 and P 0 are granted to eat  
P 5 is waiting

combination 3

P 5 and P 0 are granted to eat  
P 3 is waiting

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

## **EXPERIMENT.NO 4**

### **MEMORY MANAGEMENT**

#### **A). MEMORY MANAGEMENT WITH FIXED PARTITIONING TECHNIQUE (MFT)**

**AIM:** To implement and simulate the MFT algorithm.

#### **DESCRIPTION:**

In this the memory is divided in two parts and process is fit into it. The process which is best suited will be placed in the particular memory where it suits. In MFT, the memory is partitioned into fixed size partitions and each job is assigned to a partition. The memory assigned to a partition does not change. In MVT, each job gets just the amount of memory it needs. That is, the partitioning of memory is dynamic and changes as jobs enter and leave the system. MVT is a more "efficient" user of resources. MFT suffers with the problem of internal fragmentation and MVT suffers with external fragmentation.

#### **ALGORITHM:**

Step1: Start the process.

Step2: Declare variables.

Step3: Enter total memory size ms.

Step4: Allocate memory for os.

$Ms = ms - os$

Step5: Read the no partition to be divided n Partition size =  $ms/n$ .

Step6: Read the process no and process size.

Step 7: If process size is less than partition size allot else block the process. While allocating update memory wastage-external fragmentation.

```
if(pn[i]==pn[j])f=1;
```

```
if(f==0){ if(ps[i]<=siz)
```

```
{
```

```
extft=extft+size-
```

```
ps[i];avail[i]=1; count++;
```

```
}
```

```
}
```

Step 8: Print the results

### SOURCE CODE :

```
#include<stdio.h>
#include<conio.h>
main()
{
int    ms,    bs,    nob,
ef,n, mp[10],tif=0; int i,p=0;
clrscr();
printf("Enter the total memory available (in Bytes) -- ");
scanf("%d",&ms);
printf("Enter the block size (in Bytes) -- ");
scanf("%d", &bs);
nob=ms/bs;
ef=ms - nob*bs;
printf("\nEnter the number of processes -- ");
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("Enter memory required for process %d (in Bytes)-- ",i+1);
scanf("%d",&mp[i]);
}
printf("\nNo.        of        Blocks        available        in        memory--%d",nob);
printf("\n\nPROCESS\tMEMORYREQUIRED\tALLOCATED\tINTERNAL
FRAGMENTATION");
for(i=0;i<n && p<nob;i++)
{
printf("\n %d\t\t%d",i+1,mp[i]);
if(mp[i] > bs)
printf("\t\tNO\t\t---");
else
{
printf("\t\tYES\t\t%d",bs-mp[i]);
tif = tif + bs-mp[i];
p++;
}
}
if(i<n)
printf("\nMemory is Full, Remaining Processes cannot be accomodated");
printf("\n\nTotal Internal Fragmentation is %d",tif);
printf("\nTotal External Fragmentation is %d",ef);
getch();
}
```



### INPUT

Enter the total memory available (in Bytes) -- 1000  
Enter the block size (in Bytes)-- 300  
Enter the number of processes -- 5  
Enter memory required for process 1 (in Bytes) -- 275  
Enter memory required for process 2 (in Bytes) -- 400  
Enter memory required for process 3 (in Bytes) -- 290  
Enter memory required for process 4 (in Bytes) -- 293  
Enter memory required for process 5 (in Bytes) -- 100  
No. of Blocks available in memory -- 3

### OUTPUT

PROCESS	MEMORY REQUIRED	ALLOCATED	INTERNAL FRAGMENTATION
1	275	YES	25
2	400	NO	-----
3	290	YES	10
4	293	YES	7

Memory is Full, Remaining Processes cannot be accommodated Total

Internal Fragmentation is 42

Total External Fragmentation is 100

## **B) MEMORY VARIABLE PARTITIONING TYPE (MVT)**

**AIM:** To write a program to simulate the MVT algorithm

### **ALGORITHM:**

Step1: start the

process. Step2: Declare variables.

Step3: Enter total memory size ms.

Step4: Allocate memory for os.

Ms=ms-os

Step5: Read the no partition to be divided n Partition size=ms/n.

Step6: Read the process no and process size.

Step 7: If process size is less than partition size allot else block the process. While allocating update memory wastage-external fragmentation.

```
if(pn[i]==pn[j])    f=1;
```

```
if(f==0){ if(ps[i]<=size)
```

```
{
```

```
extft=extft+size-
```

```
ps[i];avail[i]=1; count++;
```

```
}
```

```
}
```

Step 8: Print the results

Step 9: Stop the process.

## SOURCE CODE:

```
#include<stdio.h>
#include<conio.h>
main()
{
    int
    ms,mp[10],i, temp,n=0;
    char ch = 'y'; clrscr();
    printf("\nEnter the total memory available (in Bytes)-- ");
    scanf("%d",&ms);
    temp=ms;
    for(i=0;ch=='y';i++,n++)
    {
        printf("\nEnter memory required for process %d (in Bytes) -- ",i+1);
        scanf("%d",&mp[i]);
        if(mp[i]<=temp)
        {
            printf("\nMemory is allocated for Process %d ",i+1);
            temp = temp - mp[i];
        }
        else
        {
            printf("\nMemory is Full"); break;
        }
        printf("\nDo you want to continue(y/n) -- ");
        scanf(" %c", &ch);
    }
    printf("\n\nTotal    Memory    Available    --    %d",    ms);
    printf("\n\n\tPROCESS\t\t MEMORY    ALLOCATED    ");
    for(i=0;i<n;i++)
    printf("\n \t%d\t\t\t\t\t",i+1,mp[i]);
    printf("\n\nTotal    Memory    Allocated    is    %d",ms-temp);
    printf("\nTotal External Fragmentation is %d",temp);
    getch();
}
```

### OUTPUT:

Enter the total memory available (in Bytes) – 1000  
Enter memory required for process 1 (in Bytes) – 400  
Memory is allocated for Process 1  
Do you want to continue(y/n) -- y  
Enter memory required for process 2 (in Bytes) -- 275  
Memory is allocated for Process 2  
Do you want to continue(y/n) – y  
Enter memory required for process 3 (in Bytes) – 550

Memory is Full

PROCESS	MEMORY ALLOCATED
1	400
2	275

Total Memory Allocated is 675  
Total External Fragmentation is 325

### VIVA QUESTIONS

- 1) What is MFT?
- 2) What is MVT?
- 3) What is the difference between MVT and MFT?
- 4) What is meant by fragmentation?
- 5) Give the difference between internal and external fragmentation

## EXPERIMENT.NO 5

### MEMORY ALLOCATION TECHNIQUES

**AIM:** To Write a C program to simulate the following contiguous memory allocation techniques

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

#### DESCRIPTION

One of the simplest methods for memory allocation is to divide memory into several fixed-sized partitions. Each partition may contain exactly one process. In this multiple-partition method, when a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process. The operating system keeps a table indicating which parts of memory are available and which are occupied. Finally, when a process arrives and needs memory, a memory section large enough for this process is provided. When it is time to load or swap a process into main memory, and if there is more than one free block of memory of sufficient size, then the operating system must decide which free block to allocate. Best-fit strategy chooses the block that is closest in size to the request. First-fit chooses the first available block that is large enough. Worst-fit chooses the largest available block.

#### PROGRAM

##### *WORST-FIT*

```
#include<stdio.h>
#include<conio.h>
#define max 25
void main()
{
    int
    frag[max],b[max],f[max],i,j,nb,nf,t
    emp; static int bf[max],ff[max];
    clrscr();
    printf("\n\tMemory Management Scheme - First
    Fit"); printf("\nEnter the number of blocks:");
    scanf("%d",&nb);
    printf("Enter the number of files:");
    scanf("%d",&nf);
    printf("\nEnter the size of the
    blocks:-\n"); for(i=1;i<=nb;i++)
    {
        printf("Block %d:",i);
        scanf("%d",&b[i]);
    }
    printf("Enter the size of the files :-\n");
    for(i=1;i<=nf;i++)
    {
        printf("File %d:",i);
        scanf("%d",&f[i]);
```





### ***BEST-FIT***

```
#include<stdio.h>
#include<conio.h>
#define max 25
void main()
{
    int frag[max],b[max],f[max],i,j,nb,nf,temp,lowest=10000;
    static int bf[max],ff[max];
    clrscr();
    printf("\nEnter the number of blocks:");
    scanf("%d",&nb);
    printf("Enter the number of files:");
    scanf("%d",&nf);
    printf("\nEnter the size of the
    blocks:-\n"); for(i=1;i<=nb;i++)
    printf("Block %d:",i);
    scanf("%d",&b[i]);
    printf("Enter the size of the files :-\n");
    for(i=1;i<=nf;i++)
    {
        printf("File %d:",i);
        scanf("%d",&f[i]);
    }
    for(i=1;i<=nf;i++)
    {
        for(j=1;j<=nb;j++)
        {
            if(bf[j]!=1)
            {
                temp=b[j]-f[i]
                ; if(temp>=0)
                if(lowest>temp)
                {
                    ff[i]=j;
                    lowest=temp;
                }
            }
        }
        frag[i]=lowest; bf[ff[i]]=1; lowest=10000;
    }
    printf("\nFile No\tFile Size \tBlock
    No\tBlock Size\tFragment"); for(i=1;i<=nf &&
    ff[i]!=0;i++)

        printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d",i,f[i],ff[i],b[ff[i]],frag[i]);
    getch();
}
```

### INPUT

Enter the number of blocks: 3

Enter the number of files: 2

Enter the size of the blocks:-

Block 1: 5

Block 2: 2

Block 3: 7

Enter the size of the files:-

File 1: 1

File 2: 4

### OUTPUT

	File No	File Size	Block No	Block Size	Fragment
1	1		2	2	1
2	4		1	5	1

### FIRST-FIT

```
#include<stdio.h>
#include<conio.h>
#define max 25
void main()
{
    int
    frag[max],b[max],f[max],i,j,nb,nf,temp,highe
    t=0; static int bf[max],ff[max];
    clrscr();
    printf("\n\tMemory Management Scheme - Worst Fit");
    printf("\nEnter the number of blocks:");
    scanf("%d",&nb);
    printf("Enter the number of files:");
    scanf("%d",&nf);
    printf("\nEnter the size of the blocks:-\n");
    for(i=1;i<=nb;i++)
    {
        printf("Block %d:",i);
        scanf("%d",&b[i]);
    }
    printf("Enter the size of the files :-\n");
    for(i=1;i<=nf;i++)
    {
        printf("File %d:",i);
        scanf("%d",&f[i]);
    }
}
```

```

for(i=1;i<=nf;i++)
{
    for(j=1;j<=nb;j++)
    {
        if(bf[j]!=1) //if bf[j] is not allocated
        {
            temp=b[j]-f[i]
            ; if(temp>=0)
                if(highest<temp)
                {
                    }
            }
        }
        frag[i]=highest; bf[ff[i]]=1; highest=0;
    }
    ff[i]=j; highest=temp;
}
printf("\nFile_no:\tFile_size:\tBlock_no:\tBlock_size:\tFragement");
for(i=1;i<=nf;i++)
    printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d",i,f[i],ff[i],b[ff[i]],frag[i]);
getch();
}

```

### INPUT

Enter the number of blocks: 3

Enter the number of files: 2

Enter the size of the blocks:-

Block 1: 5

Block 2: 2

Block 3: 7

Enter the size of the files:-

File 1: 1

File 2: 4

### OUTPUT

File No	File Size	Block No	Block Size	Fragment
1	1	3	7	6
2	4	1	5	1

## EXPERIMENT NO.6

### PAGE REPLACEMENT ALGORITHMS

**AIM:** To implement FIFO page replacement technique.

**a) FIFO    b) LRU    c) OPTIMAL**

#### **DESCRIPTION:**

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

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

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

**OPTIMAL-** In this algorithm, pages are replaced which would not be used for the longest duration of time in the future. This algorithm will give us less page fault when compared to other page replacement algorithms.

#### **ALGORITHM:**

1. Start the process
2. Read number of pages n
3. Read number of pages no
4. Read page numbers into an array a[i]
5. Initialize avail[i]=0 .to check page hit
6.     Replace the page with circular queue, while re-placing check page availability in the frame  
     Place avail[i]=1 if page is placed in the frame Count page faults
7. Print the results.
8. Stop the process.



**A) FIRST IN FIRST**  
**OUT SOURCE CODE :**

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

```
printf("Number of page faults : %d ",pf+frsize);
getch();
}
void display()
{
int i;
printf("\n");
for(i=0;i<3;i++)
printf("%d\t",fr[i]);
}
```

### OUTPUT:

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

Number of page faults: 9

## B) LEAST RECENTLY USED

**AIM:** To implement LRU page replacement technique.

### ALGORITHM:

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

### SOURCE CODE :

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

```

{
for(i=0;i<3;i++)
{
if(fr[i]==-1)
{
fr[i]=p[j];
flag2=1; break;
}
}
}
if(flag2==0)
{
for(i=0;i<3;i++)
fs[i]=0;
for(k=j-1,l=1;l<=frsize-1;l++,k--)
{
for(i=0;i<3;i++)
{
if(fr[i]==p[k]) fs[i]=1;
}}
for(i=0;i<3;i++)
{
if(fs[i]==0)
index=i;
}
fr[index]=p[j];
pf++;
}
display();
}
printf("\n no of page faults :%d",pf+frsize);
getch();
}
void display()
{
int i; printf("\n");
for(i=0;i<3;i++)
printf("\t%d",fr[i]);
}

```

**OUTPUT:**

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

No of page faults: 7



### C) OPTIMAL

**AIM:** To implement optimal page replacement technique.

#### ALGORITHM:

1. Start Program
2. Read Number Of Pages And Frames
3. Read Each Page Value
4. Search For Page In The Frames
5. If Not Available Allocate Free Frame
6. If No Frames Is Free Replace The Page With The Page That Is Leastly Used
7. Print Page Number Of Page Faults
8. Stop process.

#### SOURCE CODE:

```
/* Program to simulate optimal page replacement
*/ #include<stdio.h>
#include<conio.h>
int fr[3], n, m;
void
display();
void main()
{
    int i,j,page[20],fs[10];
    int
    max,found=0,lg[3],index,k,l,flag1=0,flag2=0,pf=0
    ; float pr;
    clrscr();
    printf("Enter length of the reference string:
"); scanf("%d",&n);
    printf("Enter the reference string: ");
    for(i=0;i<n;i++)
        scanf("%d",&page[i]);
    printf("Enter no of frames: ");
    scanf("%d",&m);
    for(i=0;i<m;i++)
        fr[i]=-1; pf=m;
```

```

for(j=0;j<n;j++)
{
flag1=0;
flag2=0;
for(i=0;i<m;i++)
{
if(fr[i]==page[j])
{
flag1=1; flag2=1;
break;
}
}
if(flag1==0)
{
for(i=0;i<m;i++)
{
if(fr[i]==-1)
{
fr[i]=page[j]; flag2=1;
break;
}
}
}
if(flag2==0)
{
for(i=0;i<m;i++)
lg[i]=0;
for(i=0;i<m;i++)
{
for(k=j+1;k<=n;k++)
{
if(fr[i]==page[k])
{
lg[i]=k-j;
break;
}
}
}
}
found=0;
for(i=0;i<m;i++)
{
if(lg[i]==0)
{
index=i;
found = 1;

```

```

break;
}
}
if(found==0)
{
max=lg[0]; index=0;
for(i=0;i<m;i++)
{
if(max<lg[i])
{
max=lg[i];
index=i;
}
}
}
fr[index]=page[j];
pf++;
}
display();
}
printf("Number of page faults : %d\n",
pf); pr=(float)pf/n*100;
printf("Page fault rate = %f\n", pr); getch();
}
void display()
{
int i;
for(i=0;i<m;i++)
printf("%d\t",fr[i]);
printf("\n");
}

```

**OUTPUT:**

Enter length of the reference string: 12

Enter the reference string: 1 2 3 4 1 2 5 1 2 3 4 5

Enter no of frames: 3

1 -1 -1

1 2 -1

1 2 3

1 2 4

1 2 4

1 2 4

1 2 5

1 2 5

1 2 5

3 2 5

4 2 5

4 2 5

Number of page faults : 7 Page fault rate = 58.333332

**VIVA QUESTIONS**

- 1) What is meant by page fault?
- 2) What is meant by paging?
- 3) What is page hit and page fault rate?
- 4) List the various page replacement algorithm
- 5) Which one is the best replacement algorithm?

## EXPERIMENT NO. 7

### FILE ORGANIZATION TECHNIQUES

#### A) SINGLE LEVEL DIRECTORY:

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

#### **DESCRIPTION:**

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

#### **SOURCE CODE :**

```
#include<stdio.h>
struct
{
    char
    dname[10],fname[10][10]; int
    fcnt;
}dir;

void main()
{
    int i,ch; char
    f[30]; clrscr();
    dir.fcnt = 0;
    printf("\nEnter name of directory --
    "); scanf("%s", dir.dname);
    while(1)
    {
        printf("\n\n1. Create File\t2. Delete File\t3. Search File \n
        4. Display Files\t5. Exit\nEnter your choice -- ");
        scanf("%d",&ch);
        switch(ch)
        {
            case 1: printf("\nEnter the name of the file -- ");
                scanf("%s",dir.fname[dir.fcnt]);
                dir.fcnt++; break;
            case 2: printf("\nEnter the name of the file -- ");
                scanf("%s",f);
                for(i=0;i<dir.fcnt;i++)
                {
                    if(strcmp(f, dir.fname[i])==0)
                    {
                        printf("File %s is deleted ",f); strcpy(dir.fname[i],dir.fname[dir.fcnt-1]); break;
                    }
                }
            }
        }
    }
```



```

}
if(i==dir.fcnt)
printf("File %s not found",f);
else

```

```

d
i
r
f
c
n
t
-
-
;
b
r
e
a
k
;

```

```

case 3:    printf("\nEnter the name of the file -- ");
scanf("%s",f);
for(i=0;i<dir.fcnt;i++)
{
if(strcmp(f, dir.fname[i])==0)
{
printf("File %s is found ", f);
break;
}
}
if(i==dir.fcnt)
printf("File %s not found",f);
break;
case 4:    if(dir.fcnt==0)
printf("\nDirectory Empty");
else
{
printf("\nThe Files are -- ");
for(i=0;i<dir.fcnt;i++)
printf("\t%s",dir.fname[i]);
}
break;

getch();}

```

```

}

```

```
default: exit(0);  
}
```

The Neotia University

## OUTPUT:

Enter name of directory -- CSE

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

Enter the name of the file -- A

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

Enter the name of the file -- B

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

Enter the name of the file -- C

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

The Files are -- A B C

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

Enter the name of the file – ABC File

ABC not found

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

Enter the name of the file – B

File B is deleted

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

```
#include<stdio.h>

struct
{
    char
    dname[10],fname[10][10]; int
    fcnt;
}dir[10];

void main()
{
    int i,ch,dcnt,k;
    char f[30], d[30];
    clrscr(); dcnt=0;
    while(1)
    {
        printf("\n\n1. Create Directory\t2. Create File\t3. Delete File");
        printf("\n4. Search File\t\t5. Display\t6. Exit\tEnter your choice --");
        scanf("%d",&ch);
        switch(ch)
        {
            case 1: printf("\nEnter name of directory -- ");
                    scanf("%s", dir[dcnt].dname);
                    dir[dcnt].fcnt=0;
                    dcnt++;
                    printf("Directory created"); break;
            case 2: printf("\nEnter name of the directory -- ");
                    scanf("%s",d);
                    for(i=0;i<dcnt;i++)
                        if(strcmp(d,dir[i].dname)==0)
                        {
                            printf("Enter name of the file -- ");
                            scanf("%s",dir[i].fname[dir[i].fcnt]);
                        }
                    break;
            case 3: printf("\nEnter name of file to be deleted -- ");
                    scanf("%s",f);
                    for(i=0;i<dcnt;i++)
                        if(strcmp(f,dir[i].fname[dir[i].fcnt])==0)
                        {
                            printf("File %s deleted",f);
                            dir[i].fcnt--;
                            break;
                        }
                    break;
            case 4: printf("\nEnter file name to be searched -- ");
                    scanf("%s",f);
                    for(i=0;i<dcnt;i++)
                        if(strcmp(f,dir[i].fname[dir[i].fcnt])==0)
                        {
                            printf("File %s found",f);
                            break;
                        }
                    break;
            case 5: printf("\nDisplaying files in directory");
                    for(i=0;i<dcnt;i++)
                        printf("\nFile %s",dir[i].fname[dir[i].fcnt]);
                    break;
            case 6: break;
        }
    }
}
```

```

        dir[i].fent++;
        printf("File created");
    }
    if(i==dcnt)
        printf("Directory %s not found",d);
        break;
case 3: printf("\nEnter name of the directory -- ");
        scanf("%s",d);
        for(i=0;i<dcnt;i++)
        for(i=0;i<dcnt;i++)
        {
            if(strcmp(d,dir[i].dname)==0)
            {
                printf("Enter name of the file -- ");
                scanf("%s",f);
                for(k=0;k<dir[i].fent;k++)
                {
                    if(strcmp(f, dir[i].fname[k])==0)
                    {
                        printf("File %s is deleted ",f);
                        dir[i].fent--;
                        strepy(dir[i].fname[k],dir[i].fname[dir[i].fent]);
                        goto jmp;
                    }
                }
            }
        }
        printf("File %s not found",f); goto jmp;
    }
}
printf("Directory %s not found",d);
jmp : break;
case 4: printf("\nEnter name of the directory -- ");
        scanf("%s",d);
        for(i=0;i<dcnt;i++)
        {
            if(strcmp(d,dir[i].dname)==0)
            {
                printf("Enter the name of the file -- ");
                scanf("%s",f);
                for(k=0;k<dir[i].fent;k++)
                {
                    if(strcmp(f, dir[i].fname[k])==0)
                    {
                        printf("File %s is found ",f); goto jmp1;
                    }
                }
            }
        }

        printf("
File %s

```

not  
found",f

```
); goto  
jmp1;
```

Page 44

```
}  
}
```

The Neotia University



```

        printf("Directory %s not found",d); jmp1: break;
    case 5: if(dcnt==0)
        printf("\nNo Directory's ");
        else
        {
            printf("\nDirectory\tFiles");
            for(i=0;i<dcnt;i++)
            {
                printf("%s\n",
                    getc(f));
            }
            break;
        }
    default:exit(0);
}
}
getch();
}

```

p  
r  
i  
n  
t  
f  
(  
"  
\  
n  
%  
s  
\  
t  
\  
t  
"  
,  
d  
i  
r  
[  
i  
]  
.  
d  
n  
a  
m  
e  
)  
;  
f  
o  
r  
(  
k  
=  
0

; k < d i r [ i ] . f e n t ; k + + ) p r i n t f ( " \ t % s " , d i r [ i ] . f n a m e [ k ]

) ;

The Neotia University

## OUTPUT

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

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

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

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

Enter name of the file -- A2  
File created

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

## VIVA QUESTIONS

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

## **EXPERIMENT.NO.8**

### **FILE ALLOCATION STRATEGIES**

#### **A) SEQUENTIAL:**

**AIM:** To write a C program for implementing sequential file allocation method

#### **DESCRIPTION:**

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

#### **ALGORITHM:**

Step 1: Start the program.

Step 2: Get the number of files.

Step 3: Get the memory requirement of each file.

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

Randomly select a location from available location  $s1 = \text{random}(100)$ ;

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

```
if(b[s1].flag==0){  
for  
(j=s1;j<s1+p[i];j++){  
if((b[j].flag)==0)count++;  
}
```

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

- b) Allocate and set flag=1 to the allocated locations. for( $s=s1$ ;  $s<(s1+p[i])$ ;  $s++$ )

```
{  
k[i][j]=s; j=j+1; b[s].bno=s;  
b[s].flag=1;  
}
```

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

6: Stop the program

### SOURCE CODE :

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

**OUTPUT:**

Enter the starting block & length of file 4

10 4->1

5->1

6->1

7->1

8->1

9->1

10->1

11->1

12->1

13->1

The file is allocated to disk.



## **B) INDEXED:**

**AIM:** To implement allocation method using chained method

### **DESCRIPTION:**

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

### **ALGORITHM:**

Step 1: Start the program.

Step 2: Get the number of files.

Step 3: Get the memory requirement of each file.

Step 4: Allocate the required locations by selecting a location randomly  $q = \text{random}(100)$ ;

a) Check whether the selected location is free .

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

```
q=random(100);
```

```
{
```

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

```
b[q].flag=1;
```

```
b[q].fno=j;
```

```
r[i][j]=q;
```

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

Step 6: Stop the program

### SOURCE CODE :

```
#include<stdio.h>
int
f[50],i,k,j,inde[50],n,c,count=0,p;
main()
{
clrscr();
for(i=0;i<50;i++)
) f[i]=0;
x: printf("enter index block\t");
scanf("%d",&p);
if(f[p]==0)
{
f[p]=1;
printf("enter no of files on index\t");
scanf("%d",&n);
}
else
{
printf("Block already allocated\n");
goto x;
}
for(i=0;i<n;i++)
scanf("%d",&inde[i]);
for(i=0;i<n;i++)
if(f[inde[i]]==1)
{
printf("Block already allocated");
goto x;
}
for(j=0;j<n;j++)
f[inde[j]]=1;
printf("\n    allocated");
printf("\n file indexed");
for(k=0;k<n;k++)
printf("\n %d->%d:%d",p,inde[k],f[inde[k]]);
printf(" Enter 1 to enter more files and 0 to exit\t");
scanf("%d",&c);
if(c==1)
goto
x; else
exit();
getch();
}
```

**OUTPUT:** enter index block 9

Enter no of files on index 3 1

2 3

Allocated

File indexed

9->1:1

9->2;1

9->3:1 enter 1 to enter more files and 0 to exit

### C) LINKED:

**AIM:** To implement linked file allocation technique.

#### DESCRIPTION:

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

#### ALGORITHM:

Step 1: Start the program.

Step 2: Get the number of files.

Step 3: Get the memory requirement of each file.

Step 4: Allocate the required locations by selecting a location randomly  $q = \text{random}(100)$ ;

a) Check whether the selected location is free .

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

While allocating next location address to attach it to previous location

```
for(i=0; i<n; i++)
{
for(j=0; j<s[i]; j++)
{
q=random(100);
if(b[q].flag==0) b[q].flag=1;
b[q].fno=j;
r[i][j]=q;
if(j>0)
{
}
}
}
p=r[i][j-1]; b[p].next=q;}
```

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

Step 6: Stop the program

### SOURCE CODE :

```
#include<stdio.h>
main()
{
int f[50],p,i,j,k,a,st,len,n,c;
clrscr();
for(i=0;i<50;i++) f[i]=0;
printf("Enter how many blocks that are already
allocated"); scanf("%d",&p);
printf("\nEnter the blocks no.s that are already allocated");
for(i=0;i<p;i++)
{
scanf("%d",&a);
f[a]=1;
}
X:
printf("Enter the starting index block
& length"); scanf("%d%d",&st,&len); k=len;
for(j=st;j<(k+st);j++)
{
if(f[j]==0)
{ f[j]=1;
printf("\n%d->%d",j,f[j]);
}
else
{
printf("\n %d->file is already
allocated",j);
k++;
}
}
printf("\n If u want to enter one
more file? (yes-1/no-0)");
scanf("%d",&c);
if(c==1)
goto
X;
else
exit();
getch( );}
```

## OUTPUT:

Enter how many blocks that are already allocated 3 Enter the blocks no.s that are already allocated 4 7 Enter the starting index block & length 3 7 9

3->1

4->1 file is already

allocated 5->1

6->1

7->1 file is already

allocated 8->1

9->1file is already

allocated 10->1

11->1

12->1

## VIVA QUESTIONS

- 1) List the various types of files
- 2) What are the various file allocation strategies?
- 3) What is linked allocation?
- 4) What are the advantages of linked allocation?
- 5) What are the disadvantages of sequential allocation methods?



## EXPERIMENT.NO 9

### DEAD LOCK AVOIDANCE

**AIM:** To Simulate bankers algorithm for Dead Lock Avoidance (Banker's Algorithm)

#### DESCRIPTION:

Deadlock is a situation where in two or more competing actions are waiting for the other to finish, and thus neither ever does. When a new process enters a system, it must declare the maximum number of instances of each resource type it needed. This number may exceed the total number of resources in the system. When the user request a set of resources, the system must determine whether the allocation of each resources will leave the system in safe state. If it will the resources are allocation; otherwise the process must wait until some other process release the resources.

Data structures

- n-Number of process, m-number of resource types.
- Available: Available[j]=k, k – instance of resource type R<sub>j</sub> is available. Max:  
If • max[i, j]=k, P<sub>i</sub> may request at most k instances resource R<sub>j</sub>.
- Allocation: If Allocation [i, j]=k, P<sub>i</sub> allocated to k instances of resource R<sub>j</sub> Need: If  
• Need[I, j]=k, P<sub>i</sub> may need k more instances of resource type R<sub>j</sub>, Need[I, j]=Max[I, j]-  
Allocation[I, j];

#### *Safety Algorithm*

1. Work and Finish be the vector of length m and n respectively, Work=Available and Finish[i] =False.
2. Find an i such that both  
Finish[i] • =False  
• Need≤Work If  
no such I exists go to step 4.
3. work= work + Allocation, Finish[i] =True;
4. if Finish[1]=True for all I, then the system is in safe state.

#### Resource request algorithm

Let Request i be request vector for the process P<sub>i</sub>, If request i=[j]=k, then process P<sub>i</sub> wants k instances of resource type R<sub>j</sub>.

1. if Request≤Need I go to step 2. Otherwise raise an error condition.
2. if Request≤Available go to step 3. Otherwise P<sub>i</sub> must since the resources are available.
3. Have the system pretend to have allocated the requested resources to process P<sub>i</sub> by modifying the state as follows;  
Available=Available-Request I;  
Allocation I=Allocation +Request I;  
Need i=Need i- Request I;  
If the resulting resource allocation state is safe, the transaction is completed and process P<sub>i</sub> is allocated its resources. However if the state is unsafe, the P<sub>i</sub> must wait for Request i and the old resource-allocation state is restored.

### ALGORITHM:

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

### SOURCE CODE :

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
    int alloc[10][10],max[10][10];
    int avail[10],work[10],total[10];
    int i,j,k,n,need[10][10];
    int m;
    int    count=0,c=0;
    char
    finish[10]; clrscr();
    printf("Enter the no. of processes and
    resources:"); scanf("%d%d",&n,&m);
    for(i=0;i<=n;i++)
    finish[i]='n';
    printf("Enter the claim matrix:\n");
    for(i=0;i<n;i++)
    for(j=0;j<m;j++)
    scanf("%d",&max[i][j]);
    printf("Enter the allocation matrix:\n");
    for(i=0;i<n;i++)
    for(j=0;j<m;j++)
    scanf("%d",&alloc[i][j]);
    printf("Resource vector:");
    for(i=0;i<m;i++)
    scanf("%d",&total[i]);
    for(i=0;i<m;i++)
    avail[i]=0;
    for(i=0;i<n;i++)
```

```

for(j=0;j<m;j++)
    avail[j]+=alloc[i][j];
for(i=0;i<m;i++)
    work[i]=avail[i];
for(j=0;j<m;j++)
    work[j]=total[j]-work[j];
for(i=0;i<n;i++)
    for(j=0;j<m;j++)
        need[i][j]=max[i][j]-alloc[i][j]
; A:
for(i=0;i<n;i++)
{
    c=0;
    for(j=0;j<m;j++)
        if((need[i][j]<=work[j])&&(finish[i]=='n'))
            c++;
    if(c==m)
    {
        printf("All the resources can be allocated to Process %d", i+1);
        printf("\n\nAvailable resources are:");
        for(k=0;k<m;k++)
        {
            work[k]+=alloc[i][k];
            printf("%4d",work[k]);
        }
        printf("\n");
        finish[i]='y';
        printf("\nProcess %d executed?:%c \n",i+1,finish[i]);
        count++;
    }
}
if(count!=n)
    goto A;
else
    printf("\n System is in safe mode");
printf("\n The given state is safe state");
getch();
}

```

## OUTPUT

Enter the no. of processes and resources: 4 3

Enter the claim matrix:

3 2 2

6 1 3

3 1 4

4 2 2

Enter the allocation matrix:

1 0 0

6 1 2

2 1 1

0 0 2

Resource vector: 9 3 6

All the resources can be allocated to Process 2

Available resources are: 6 2 3

Process 2 executed?: y

All the resources can be allocated to Process 3 Available resources are: 8 3 4

Process 3 executed?: y

All the resources can be allocated to Process 4 Available resources are: 8 3 6

Process 4 executed?: y

All the resources can be allocated to Process 1

Available resources are: 9 3 6

Process 1 executed?: y

System is in safe mode

The given state is safe state

## VIVA QUESTIONS

- 1) What is meant by deadlock?
- 2) What is safe state in banker's algorithms?
- 3) What is banker's algorithm?
- 4) What are the necessary conditions where deadlock occurs?
- 5) What are the principles and goals of protection?

## EXPERIMENT.NO 10 DEAD LOCKPREVENTION

**AIM:** To implement deadlock prevention technique

### **Banker's Algorithm:**

When a new process enters a system, it must declare the maximum number of instances of each resource type it needed. This number may exceed the total number of resources in the system. When the user request a set of resources, the system must determine whether the allocation of each resources will leave the system in safe state. If it will the resources are allocation; otherwise the process must wait until some other process release the resources.

### **DESCRIPTION:**

Data structures

- 
- n-Number of process, m-number of resource types.
- 
- Available: Available[j]=k, k – instance of resource type R<sub>j</sub> is available.
- Max: If max[i, j]=k, P<sub>i</sub> may request at most k instances resource R<sub>j</sub>.  
Allocation: If Allocation [i, j]=k, P<sub>i</sub> allocated to k instances of resource R<sub>j</sub> Need:  
If Need[L, j]=k, P<sub>i</sub> may need k more instances of resource type R<sub>j</sub>,  
Need[L, j]=Max[L, j]-Allocation[L, j];

*Safety Algorithm*

Work and Finish be the vector of length m and n respectively, Work=Available and Finish[i] =False.

Find an i such  
that both •Finish[i] =False  
•Need≤Work

If no such I exists go to step 4.

5. work=work+Allocation, Finish[i] =True;

if Finish[1]=True for all I, then the system is in safe state



### ALGORITHM:

1. Start the program.
2. Get the values of resources and processes.
3. Get the avail value.
4. After allocation find the need value.
5. Check whether its possible to allocate.
6. If it is possible then the system is in safe state.
7. Else system is not in safety state
8. Stop the process.

### SOURCE CODE :

```
#include<stdio.h>
#include<conio.h>
> void main()
{
    char job[10][10];
    int time[10],avail,tem[10],temp[10]; int
    safe[10]; int ind=1,i,j,q,n,t;
    clrscr();
    printf("Enter no of jobs: ");
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
        printf("Enter name and time:
        "); scanf("%s%d",&job[i],&time[i]);
    }
    printf("Enter the available
    resources:"); scanf("%d",&avail);
    for(i=0;i<n;i++)
    {
        temp[i]=time[i];
        tem[i]=i;
    }
    for(i=0;i<n;i++)
    for(j=i+1;j<n;j++)
    {
        if(temp[i]>temp[j])
        {
            t=temp[i];
```



```

temp[i]=temp[j];
temp[j]=t; t=temp[i];
tem[i]=tem[j];
tem[j]=t;
}
}
for(i=0;i<n;i++)
{
q=tem[i];
if(time[q]<=avail)
{
safe[ind]=tem[i];
avail=avail-tem[q];
printf("%s",job[safe[ind]])
; ind++;
}
else
{
printf("No safe sequence\n");
}
}
printf("Safe sequence is:");
for(i=1;i<ind; i++)
printf("%s %d\n",job[safe[i]],time[safe[i]]);
getch();
}

```

### OUTPUT:

```

Enter no of jobs:4
Enter name and time: A 1
Enter name and time: B 4
Enter name and time: C 2
Enter name and time: D 3
Enter the available resources: 20
Safe sequence is: A 1, C 2, D 3, B 4.

```

## EXPERIMENT.NO 11

**AIM:** To Write a C program to simulate disk scheduling algorithms

a)FCFS            b) SCAN    c) C-SCAN

### DESCRIPTION

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

### PROGRAM

#### A) FCFS DISK SCHEDULING ALGORITHM

```
#include<stdio.h>
> main()
{
    int t[20], n, I, j, tohm[20], tot=0; float avhm;
    clrscr();
    printf("enter the no.of tracks");
    scanf("%d",&n);
    printf("enter the tracks to be traversed");
    for(i=2;i<n+2;i++)
        scanf("%d",&t*i
+); for(i=1;i<n+1;i++)
    {
        tohm[i]=t[i+1]-t[i];
        if(tohm[i]<0)
            tohm[i]=tohm[i]*(-1
);
    }
    for(i=1;i<n+1;i++)
        tot+=tohm[i]
; avhm=(float)tot/n;
    printf("Tracks traversed\tDifference between tracks\n");
    for(i=1;i<n+1;i++)
        printf("%d\t\t\t%d\n",t*i,tohm*i+);
        printf("\nAverage header movements:%f",avhm);
    getch();
}
```

The Neotia University

**INPUT**

Enter no.of tracks:9

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

**OUTPUT**

Tracks traversed

Difference between tracks

55  
58  
60  
70  
18  
90  
150  
160  
184

Average header movements:30.888889

## B) SCAN DISK SCHEDULING ALGORITHM

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

```
> main()
```

```
{
```

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

```
    clrscr();
```

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

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

```
    printf("enter the position of head");
```

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

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

```
    printf("enter the
```

```
    tracks");
```

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

```
        scanf("%d",&t[i]
```

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

```
    {
```

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

```
    {
```

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

```
    {
```

```
    temp=t[j];
```

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

```
    t[j+1]=temp
```

```
    ;
```

```
    } } }
```

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

```
    +) if(t[i]==h)
```

```
        j=i;k=i;
```

```
    p=0;
```

```
    while(t[j]!=0)
```

```
    {
```

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

```
        j--, p++;
```

```
    }
```

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

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

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

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

```
    {
```

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

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

```
        else
```

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

```
        sum+=d[j];
```

```
    }
```

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

```
getch();}
```

The Neotia University



**INPUT**

Enter no.of tracks:9

Enter track position:55

58 60 70 18  
90 150 160 184

**OUTPUT**

Tracks traversed      Difference between tracks

150	50
160	10
184	24
90	94
70	20
60	10
58	2
55	3
18	37

Average header movements: 27.77

### C) C-SCAN DISK SCHEDULING ALGORITHM

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

```
> main()
```

```
{
```

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

```
    clrscr();
```

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

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

```
    printf("enter the position of head");
```

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

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

```
    printf("enter      total  
tracks"); scanf("%d",&tot);
```

```
    t[2]=tot-1;
```

```
    printf("enter      the      tracks");
```

```
    for(i=3;i<=n+2;i++)
```

```
        scanf("%d",&t[i]
```

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

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

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

```
        {
```

```
for(i=0;i<=n+2;i++) if(t[i]==h);  
    j=i;break;
```

```
t  
e  
m  
p  
=  
t  
[  
j  
]  
;  
t  
[  
j  
]  
=  
t  
[  
j  
+  
1  
]  
;  
t  
[  
j  
+
```

```
1  
]  
=  
t  
e  
m  
p  
}
```

```
p=0;  
while(t[j]!=tot-1)  
{  
    atr[p]=t[j]  
    ; j++;  
    p++;  
}  
atr[p]=t[j]  
; p++;  
i=0;  
while(p!=(n+3) && t[i]!=t[h])  
{  
    atr[p]=t[i]; i++;  
    p++;  
}
```

```
for(j=0;j<n+2;j++)
{
    if(atr[j]>atr[j+1])
        d[j]=atr[j]-atr[j+1];
    else
        d[j]=atr[j+1]-atr[j];
    sum+=d[j];
}
printf("total header movements%d",sum);
printf("avg is %f", (float)sum/n);
getch();
}
```

### INPUT

Enter the track position : 55 58 60  
70 18  
90 150  
160 184  
Enter starting position : 100

### OUTPUT

Tracks traversed	Difference Between tracks
150	50
160	10
184	24
18	240
55	37
58	3
60	2
70	10
90	20

Average seek time : 35.7777779