OPERATING SYSTEMS

LAB MANUAL

Subject Code:	A50589
Regulations:	R13 – JNTUH
Class:	III Year I Semester (CSE)

Prepared By
Mrs.M.Pallavi
Assistant Professor

1.

PROGRAM OUTCOMES	
PO1	**Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
PO2	**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.
PO3	**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.
PO4	**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.
PO5	**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.
PO6	**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.
PO7	**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.
PO8	**Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

PO9	**Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
PO10	**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.
PO11	**Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
PO12	**Life-long learning**: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

PROGRAM EDUCATIONAL OUTCOMES

PEO1	To induce strong foundation in mathematical and core concepts, which enable them to participate in **research,** in the field of computer science.
PEO2	To be able to become the part of application development and problem solving by learning the computer **programming** methods**,** of the industry and related domains.
PEO3	To Gain the **multidisciplinary** knowledge by understanding the scope of association of computer science engineering discipline with other engineering disciplines.
PEO4	To improve the communication skills, soft skills, organizing skills which build the **professional** qualities, there by understanding the social responsibilities and ethical attitude.

OPERATING SYSTEMS LAB SYLLABUS

S. No.	List of Experiments
1	Simulate the following cpu scheduling algorithms: a) FCFS b)SJF c)Round Robin d)Priority
2	Simulate the file allocation strategies: a) Sequential b) Indexed c) Linked
3	Simulate MVT and MFT
4	Simulate all File Organization techniques A)Single level directory b)Two level c)Hierarchical d)DAG
5	Simulate Bankers Algorithm for Deadlock Avoidance
6	Simulate Bankers algorithm for Deadlock Prevention
7	Simulate all page replacement Algorithms a)FIFO b) LRU c) LFU
8	Simulate Paging Technique of memory management.
9*	*Write a C program to stimulate the following contiguous memory allocation techniques a) Worst-fit b) Best fit c) First fit
10*	*Write a C program to stimulate the disk scheduling algorithms. a) FCFS b) SCAN c) C-SCAN
11*	*Write a C program to simulate optimal page replacement algorithms
12*	*Write a C program to simulate the concept of Dining-Philosophers problem.

Faculty **HoD**

COURSE OBJECTIVES:

This lab complements the operating systems course. With this course Students are able to:

COB 1.Implement practical experience with designing and implementing concepts of operating systems

COB 2: write the code to implement and modify various concepts in Operating Systems using Linux environment.

COB3: Implement Various CPU Scheduling algorithms

COB4: : Implement Various Page replacement algorithms algorithms

COB5:. Understand and Implement Banker's Algorithm

COURSE OUTCOMES:

CO1: Understand and implement basic services and functionalities of the operating system using system calls and able to **Understand** the benefits of thread over process and implement synchronized programs using multithreading concepts.

CO2: Use modern operating system calls and synchronization libraries in software/ hardware interfaces**.**

CO3: Analyze and simulate CPU Scheduling Algorithms like FCFS, Round Robin, SJF, and Priority.

CO4 :Implement memory management schemes and page replacement schemes.

CO5: Simulate file allocation and organization techniques.

CO6: Understand the concepts of deadlock in operating systems and implement them in multiprogramming system.

ATTAINMENT OF PROGRAM OUTCOMES & PROGRAM EDUCATION OUTCOMES

Exp. No.	Experiment	Program Outcomes Attained	Program Specific Outcomes Attained
1	Simulate the following cpu scheduling algorithms: a) FCFS b)SJF c)Round Robin d)Priority	PO1, PO2, PO4	PEO1
2	Simulate the file allocation strategies: a) Sequential b) Indexed c) Linked	PO1, PO2, PO4	PEO1
3	Simulate MVT and MFT	PO1, PO2, PO4	PEO1
4	Simulate all File Organization techniques A)Single level directory b)Two level c)Hierarchical	PO1, PO2, PO4	PEO1
5	Simulate Bankers Algorithm for Deadlock Avoidance	PO1, PO2, PO4	PEO1
6	Simulate Bankers algorithm for Deadlock Prevention	PO1, PO2, PO4	PEO1,PEO2
7	Simulate all page replacement Algorithms a)FIFO b) LRU c) LFU	PO1, PO2	PEO1,PEO2
8	Simulate Paging Technique of memory management.	PO1, PO2, PO4	PEO1,PEO2
9*	*Write a C program to stimulate the following contiguous memory allocation techniques a) Worst-fit b) Best fit c) First fit	PO1, PO2, PO4, PO12	PEO1
10*	*Write a C program to stimulate the disk scheduling algorithms. a)FCFS b) SCAN c) C-SCAN	PO1, PO2, PO4, PO12	PEO1,PEO2
11*	*Write a C program to simulate optimal page replacement algorithms	PO1, PO2, PO12	PEO1
12*	*Write a C program to simulate the concept of dining philosophers problem	PO1, PO2, PO3, PO4, PO5, PO12	PEO1,PEO2

ATTAINMENT OF COURSE OBJECTIVES & COURSE OUTCOMES

Exp. No.	Experiment	COURSE OBJECTIVES attained	COURSE OUTCOMES Attained
1	Simulate the following cpu scheduling algorithms: a) FCFS b)SJF c)Round Robin d)Priority	COB1,COB3	CO3
2	Simulate the file allocation strategies: a) Sequential b) Indexed c) Linked	COB1,COB2	CO1,CO5
3	Simulate MVT and MFT	COB2	CO1,CO2
4	Simulate all File Organization techniques A)Single level directory b)Two level c)Hierarchical	COB1	CO1,CO5
5	Simulate Bankers Algorithm for Deadlock Avoidance	COB5	CO1,CO6
6	Simulate Bankers algorithm for Deadlock Prevention	COB5	CO1,CO6
7	Simulate all page replacement Algorithms a)FIFO b) LRU c) LFU	COB1,COB2,COB4	CO4
8	Simulate Paging Technique of memory management.	COB1,COB2,COB4	CO4
9*	*Write a C program to stimulate the following contiguous memory allocation techniques a) Worst-fit b) Best fit c) First fit	COB1,COB3	CO3
10*	*Write a C program to stimulate the disk scheduling algorithms. a)FCFS b) SCAN c) C-SCAN	COB1,COB3	CO3
11*	*Write a C program to simulate optimal page replacement algorithms	COB1,COB2,COB4	CO4
12*	*Write a C program to simulate the concept of dining philosophers problem	COB1,2	CO1

1.

EXPERIMENT-1

1.1 OBJECTIVE

Write a C program to simulate the following non-preemptive CPU scheduling algorithms to find turnaround time and waiting time for the above problem.
a) FCFS b) SJF c) Round Robin d) Priority

1.2 DESCRIPTION

Assume all the processes arrive at the same time.

1.2.1 FCFS CPU SCHEDULING ALGORITHM

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

1.2.2 SJF CPU SCHEDULING ALGORITHM

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

1.2.3 ROUND ROBIN CPU SCHEDULING ALGORITHM

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

1.2.4 PRIORITY CPU SCHEDULING ALGORITHM

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

AIM: Using CPU scheduling algorithms find the min & max waiting time.

HARDWARE REQUIREMENTS: Intel based Desktop Pc RAM of 512 MB

SOFTWARE REQUIREMENTS: Turbo C/ Borland C.

THEORY:

CPU SCHEDULING

Maximum CPU utilization obtained with multiprogramming

1.

CPU–I/O Burst Cycle – Process execution consists of a *cycle* of

CPU execution and I/O wait

CPU burst distribution

a) First-Come, First-Served (FCFS) Scheduling

Process	Burst Time
P1	24
P2	3
P3	3

Suppose that the processes arrive in the order: P1 , P2 , P3

The Gantt Chart for the schedule is:

Waiting time for P1 = 0; P2 = 24; P3 = 27

Average waiting time: (0 + 24 + 27)/3 = 17

ALGORITHM

1. Start

2. Declare the array size

3. Read the number of processes to be inserted

4. Read the Burst times of processes

5. calculate the waiting time of each process
   wt[i+1]=bt[i]+wt[i]
6. calculate the turnaround time of each process
   tt[i+1]=tt[i]+bt[i+1]

7. Calculate the average waiting time and average turnaround time.

8. Display the values

9. Stop

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int i,j,bt[10],n,wt[10],tt[10],w1=0,t1=0;
float aw,at;
clrscr();
printf("enter no. of processes:\n");
scanf("%d",&n);
printf("enter the burst time of processes:");
for(i=0;i<n;i++)
scanf("%d",&bt[i]);
for(i=0;i<n;i++)
{
wt[0]=0;
tt[0]=bt[0];
wt[i+1]=bt[i]+wt[i];
tt[i+1]=tt[i]+bt[i+1];
w1=w1+wt[i];
t1=t1+tt[i];
}
aw=w1/n;
at=t1/n;
printf("\nbt\t wt\t tt\n");
for(i=0;i<n;i++)
printf("%d\t %d\t %d\n",bt[i],wt[i],tt[i]);
printf("aw=%f\n,at=%f\n",aw,at);
getch();
}
```

INPUT

```
Enter no of processes
3
enter bursttime
12
8
20
```

EXPECTED OUTPUT

```
        bt wt tt

        12 0 12
        8 12 20
        20 20 40
        aw=10.666670
        at=24.00000
```

VIVA QUESTIONS

1. *What is First-Come-First-Served (FCFS) Scheduling?*
2. Why CPU scheduling is required?
3. Which technique was introduced because a single job could not keep both the CPU and the I/O devices busy?
   1) Time-sharing 2) SPOOLing 3) Preemptive scheduling 4) Multiprogramming
4. CPU performance is measured through _____.
   1) Throughput 2) MHz 3) Flaps 4) None of the above
5. Which of the following is a criterion to evaluate a scheduling algorithm?
   1 CPU Utilization: Keep CPU utilization as high as possible.
   2 Throughput: number of processes completed per unit time.
   3 Waiting Time: Amount of time spent ready to run but not running.
   4 All of the above

EXPERIMENT : 1b)

NAMEOF THE EXPERIMENT: Simulate the following CPU Scheduling Algorithms

b) **SJF**
AIM: Using CPU scheduling algorithms find the min & max waiting time.

HARDWARE REQUIREMENTS: Intel based Desktop Pc
RAM of 512 MB
SOFTWARE REQUIREMENTS:
Turbo C/ Borland C.

THEORY:
Example of Non Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	3.0	4

P1	**P3**	**P2**	**P4**
0 7 8		**12 16**	

Example of Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	3.0	4

P1	P2	P3	P2	P4	P1

Average waiting time = (9 + 1 + 0 +2)/4 = 3

ALGORITHM

1. Start

2. Declare the array size

3. Read the number of processes to be inserted

4. Read the Burst times of processes

5. sort the Burst times in ascending order and process with shortest burst time is first executed.

6. calculate the waiting time of each process
   wt[i+1]=bt[i]+wt[i]
7. calculate the turnaround time of each process
   tt[i+1]=tt[i]+bt[i+1]

8. Calculate the average waiting time and average turnaround time.

9. Display the values

10. Stop

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int i,j,bt[10],t,n,wt[10],tt[10],w1=0,t1=0;
float aw,at;
clrscr();
printf("enter no. of processes:\n");
scanf("%d",&n);
printf("enter the burst time of processes:");
for(i=0;i<n;i++)
scanf("%d",&bt[i]);
for(i=0;i<n;i++)
{
for(j=i;j<n;j++)
if(bt[i]>bt[j])
{
t=bt[i];
bt[i]=bt[j];
bt[j]=t;
}
}
for(i=0;i<n;i++)
printf("%d",bt[i]);
```

```
for(i=0;i<n;i++)
{
wt[0]=0;
tt[0]=bt[0];
wt[i+1]=bt[i]+wt[i];
tt[i+1]=tt[i]+bt[i+1];
w1=w1+wt[i];
t1=t1+tt[i];
}
aw=w1/n;
at=t1/n;
printf("\nbt\t wt\t tt\n");
for(i=0;i<n;i++)
printf("%d\t %d\t %d\n",bt[i],wt[i],tt[i]);
printf("aw=%f\n,at=%f\n",aw,at);
getch();
}
```

INPUT:

```
enter no of processes
3
enter burst time
12
8
20
```

OUTPUT:

```
bt wt tt
12 8 20
8  0 8
20 20 40
aw=9.33
at=22.64
```

VIVA QUESTIONS:

1) The optimum CPU scheduling algorithm is

   (A)FIFO (B)SJF with preemption. (C)SJF without preemption.(D)Round Robin.

2) In terms of average wait time the optimum scheduling algorithm is

   (A)FCFS (B)SJF (C)Priority (D)RR

3) What are the dis-advantages of SJF Scheduling Algorithm?
4) What are the advantages of SJF Scheduling Algorithm?
5) Define CPU Scheduling algorithm?

EXPERIMENT : 1c)

NAMEOF THE EXPERIMENT: Simulate the following CPU Scheduling Algorithms

c) Round Robin

AIM: Using CPU scheduling algorithms find the min & max waiting time.

HARDWARE REQUIREMENTS: Intel based Desktop Pc
RAM of 512 MB

SOFTWARE REQUIREMENTS:
Turbo C/ Borland C.

THEORY:

Round Robin:

Example of RR with time quantum=3

Process	Burst time
aaa	4
Bbb	3
Ccc	2
Ddd	5
Eee	1

ALGORITHM

1. Start
2. Declare the array size
3. Read the number of processes to be inserted
4. Read the burst times of the processes
5. Read the Time Quantum
6. if the burst time of a process is greater than time Quantum then subtract time quantum form the burst time
   Else
   Assign the burst time to time quantum.
7. calculate the average waiting time and turn around time of the processes.
8. Display the values
9. Stop

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int st[10],bt[10],wt[10],tat[10],n,tq;
int i,count=0,swt=0,stat=0,temp,sq=0;
float awt=0.0,atat=0.0;
clrscr();
printf("Enter number of processes:");
scanf("%d",&n);
printf("Enter burst time for sequences:");
for(i=0;i<n;i++)
{
scanf("%d",&bt[i]);
st[i]=bt[i];
}
printf("Enter time quantum:");
scanf("%d",&tq);
while(1)
{
for(i=0,count=0;i<n;i++)
{
temp=tq;
if(st[i]==0)
{
count++;
continue;
}
if(st[i]>tq)
st[i]=st[i]-tq;
else
if(st[i]>=0)
{
temp=st[i];
st[i]=0;
}
sq=sq+temp;
tat[i]=sq;
}
if(n==count)
break;
}
for(i=0;i<n;i++)
{
wt[i]=tat[i]-bt[i];
swt=swt+wt[i];
stat=stat+tat[i];
}
awt=(float)swt/n;
atat=(float)stat/n;
printf("Process_no Burst time Wait time Turn around time");
for(i=0;i<n;i++)
printf("\n%d\t %d\t %d\t %d",i+1,bt[i],wt[i],tat[i]);
printf("\nAvg wait time is %f Avg turn around time is %f",awt,atat);
getch();
```

}

Input:
Enter no of jobs
4
Enter burst time
5
12
8
20
Output:

Bt wt tt
5 0 5
12 5 13
8 13 25
20 25 45
aw=10.75000
at=22.000000

VIVA QUESTIONS:
1.Round Robin scheduling is used in

(A)Disk scheduling. (B)CPU scheduling

(C)I/O scheduling. (D)Multitasking

2. What are the dis-advantages of RR Scheduling Algoritm?

3.What are the advantages of RR Scheduling Algoritm?

4.Super computers typically employ _____.

1 Real time Operating system 2 Multiprocessors OS

3 desktop OS 4 None of the above

5. An optimal scheduling algorithm in terms of minimizing the average waiting time of a given set of processes is _____.

1 FCFS scheduling algorithm 2 Round robin scheduling algorithm

3 Shortest job - first scheduling algorithm 4 None of the above

EXPERIMENT : 1d)

NAMEOF THE EXPERIMENT: Simulate the following CPU Scheduling Algorithms

d) Priority

AIM: Using CPU scheduling algorithms find the min & max waiting time.

HARDWARE REQUIREMENTS: Intel based Desktop Pc
RAM of 512 MB
SOFTWARE REQUIREMENTS:
Turbo C/ Borland C.

THEORY:

In Priority Scheduling, each process is given a priority, and higher priority methods are executed first, while equal priorities are executed First Come First Served or Round Robin.

There are several ways that priorities can be assigned:

- Internal priorities are assigned by technical quantities such as memory usage, and file/IO operations.
- External priorities are assigned by politics, commerce, or user preference, such as importance and amount being paid for process access (the latter usually being for mainframes).

ALGORITHM

1. Start

2. Declare the array size

3. Read the number of processes to be inserted

4. Read the Priorities of processes

5. sort the priorities and Burst times in ascending order

5. calculate the waiting time of each process
wt[i+1]=bt[i]+wt[i]
6. calculate the turnaround time of each process
tt[i+1]=tt[i]+bt[i+1]

6. Calculate the average waiting time and average turnaround time.

7. Display the values

8. Stop

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
void main()
 {
  int i,j,pno[10],prior[10],bt[10],n,wt[10],tt[10],w1=0,t1=0,s;
  float aw,at;
  clrscr();
  printf("enter the number of processes:");
  scanf("%d",&n);
  for(i=0;i<n;i++)
  {
  printf("The process %d:\n",i+1);
   printf("Enter the burst time of processes:");
   scanf("%d",&bt[i]);
    printf("Enter the priority of processes %d:",i+1);
    scanf("%d",&prior[i]);
    pno[i]=i+1;
  }
    for(i=0;i<n;i++)
     {
      for(j=0;j<n;j++)
      {
       if(prior[i]<prior[j])
       {
              s=prior[i];
              prior[i]=prior[j];
              prior[j]=s;

               s=bt[i];
               bt[i]=bt[j];
               bt[j]=s;

                s=pno[i];
                pno[i]=pno[j];
                pno[j]=s;
               }
              }
      }
      for(i=0;i<n;i++)
      {
        wt[0]=0;
        tt[0]=bt[0];
             wt[i+1]=bt[i]+wt[i];
             tt[i+1]=tt[i]+bt[i+1];
             w1=w1+wt[i];
             t1=t1+tt[i];
             aw=w1/n;
             at=t1/n;
             }
             printf(" \n job \t bt \t wt \t tat \t prior\n");
               for(i=0;i<n;i++)
                printf("%d \t %d \t %d\t %d\t %d\n",pno[i],bt[i],wt[i],tt[i],prior[i]);
                printf("aw=%f \t at=%f \n",aw,at);
               getch();
   }
```

Input:
Enter no of jobs
4
Enter bursttime
10
2
4
7
Enter priority values
4
2
1
3

Output:

Bt priority wt tt
4 1 0 4
2 2 4 6
7 3 6 13
10 4 13 23
aw=5.750000
at=12.500000

VIVA QUESTIONS:

1. Priority CPU scheduling would most likely be used in a _____ os.
2. Cpu allocated process to _____ priority.
3. calculate avg waiting time=
4. Maximum CPU utilization obtained with _____
5. Using _____algorithms find the min & max waiting time. `]\oiui

EXPERIMENT- 2

2.1 OBJECTIVE

Write a C program to simulate the following file allocation strategies. a) Sequential b) Linked c)) Indexed

2.2 DESCRIPTION

A file is a collection of data, usually stored on disk. As a logical entity, a file enables to divide data into meaningful groups. As a physical entity, a file should be considered in terms of its organization. The term "file organization" refers to the way in which data is stored in a file and, consequently, the method(s) by which it can be accessed.

2.3 SEQUENTIAL FILE ALLOCATION

In this file organization, the records of the file are stored one after another both physically and logically. That is, record with sequence number 16 is located just after the 15th record. A record of a sequential file can only be accessed by reading all the previous records.

2.4 LINKED FILE ALLOCATION

With linked allocation, each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk. The directory contains a pointer to the first and last blocks of the file. Each block contains a pointer to the next block.

2.5 INDEXED FILE ALLOCATION

Indexed file allocation strategy brings all the pointers together into one location: an index block. Each file has its own index block, which is an array of disk-block addresses. The i^{th} entry in the index block points to the i^{th} block of the file. The directory contains the address of the index block. To find and read the i^{th} block, the pointer in the i^{th} index-block entry is used.

2a)AIM: Simulate the file allocation strategies using file allocation methods

HARDWARE REQUIREMENTS: Intel based Desktop Pc
RAM of 512 MB

SOFTWARE REQUIREMENTS:
Turbo C/ Borland C.

THEORY:

File Allocation Strategies:

The main problem is how to allocate disk space to the files so that disk space is utilized effectively band files can be accessed quickly. We have 3 space allocation method.

1. **Contiguous allocation (Sequential)**

   It requires each file to occupy a set of contiguous blocks on the hard disk where disk address define a linear ordering on the disk.

   Disadvantages:

   i. Difficult for finding space for a new file.

   ii. Internal and external fragmentation will be occurred.

1.

ALGORITHM:

1. Start
2. Read the number of files
3. For each file, read the number of blocks required and the starting block of the file.
4. Allocate the blocks sequentially to the file from the starting block.
5. Display the file name, starting block , and the blocks occupied by the file.
6. stop

PROGRAM:

```
#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("blocks occupiedare:");
      for(i=0;i<n;i++)
      {  printf("fileno%d",i+1);
    for(j=0;j<b[i];j++)
    printf("\t%d",c[i][j]);
    printf("\n");
      }
 getch();
}
```

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
```

VIVA QUESTIONS:

1.What file access pattern is particularly suited to chained file allocation on disk?

2. Define Sequential File allocation

3.Why we use file allocation strategies?

4.what are the advantages and dis-advantages of Sequential File allocation?

5.The average waiting time =_____.

EXPERIMENT :2 b)

NAME OF EXPERIMENT: Simulate file Allocation strategies:
b) Indexed

AIM: Simulate the file allocation strategies using file allocation methods

HARDWARE REQUIREMENTS: Intel based Desktop Pc
RAM of 512 MB

SOFTWARE REQUIREMENTS:
Turbo C/ Borland C.

THEORY:

Indexed allocation

In linked allocation it is difficult to maintain FAT – so instead of that method indexed allocation method is used. Indexed allocation method solves all the problems in the linked allocation by bringing all the pointers together into one location called index block.

ALGORITHM:

1. Start
2. Read the number of files
3. Read the index block for each file.
4. For each file, read the number of blocks occupied and number of blocks of the file.
5. Link all the blocks of the file to the index block.
6. Display the file name, index block , and the blocks occupied by the file.
7. Stop

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
main()
{
 int n,m[20],i,j,ib[20],b[20][20];
 clrscr();
 printf("Enter no. of files:");
 scanf("%d",&n);
 for(i=0;i<n;i++)
 {          printf("Enter index block :",i+1);
            scanf("%d",&ib[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,ib[i],m[i]);
printf("blocks occupiedare:");
      for(i=0;i<n;i++)
      {  printf("fileno%d",i+1);
    for(j=0;j<m[i];j++)
    printf("\t%d--->%d\n",ib[i],b[i][j]);
    printf("\n");
              }
 getch();
 }
```

OUTPUT:

```
Enter no. of files:2
 Enter index block  3
Enter blocks occupied by file1: 4
enter blocks of file1:9
  4  6  7
Enter index block  5
Enter blocks occupied by file2:2
enter blocks of file2: 10   8
  File     index  length
 1       3       4
2       5       2
blocksoccupiedare:
file1
3--->9
3--->4
3--->6
3--->7
file2
5--->10
5--->8
```

VIVA QUESTIONS:

1. What file allocation strategy is most appropriate for random access files?
2.Define File?
3.Define Directory?
4. Why we use file allocation strategies?
5.what are the advantages and dis-advantages Indexed Allocation?

EXPERIMENT : 2 c)

NAME OF EXPERIMENT: Simulate file Allocation strategies:
c) Linked

AIM: Simulate the file allocation strategies using file allocation methods

HARDWARE REQUIREMENTS: Intel based Desktop Pc
RAM of 512 MB
SOFTWARE REQUIREMENTS:
Turbo C/ Borland C.

THEORY:
Linked Allocation

Linked allocation of disk space overcomes all the problems of contiguous allocation. In linked allocation each file is a linked list of disk blocks where the disk blocks may be scattered anywhere on the disk. The directory contains a pointer to the first and last blocks of the file.

Disadvantages : Space required to maintain pointers.

ALGORITHM:

1. Start
2. Read the number of files
3. For each file, read the file name, starting block, number of blocks and block numbers of the file.
4. Start from the starting block and link each block of the file to the next block in a linked list fashion.
5. Display the file name, starting block, size of the file , and the blocks occupied by the file.
6. stop

PROGRAM:

```
#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=0;j<f[i].size;j++)
                        printf("%d--->",f[i].block[j]);
            printf("%d",f[i].block[j]);
            printf("\n");
 }
 getch();
}
```

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       20--->4--->12--->15--->45--->32--->25
rajesh  12      5       12--->6--->5--->4--->3--->2
```

VIVA QUESTIONS:

1. What file access pattern is particularly suited to chained file allocation on disk?

2. What file allocation strategy is most appropriate for random access files?

3. Mention different file allocation strategies?

4. Why we use file allocation strategies?

5 .what are the advantages and dis-advantages of each strategies?

6. The _____contains a pointer to the first and last blocks of the file.

EXPERIMENT- 3

3. OBJECTIVE

Write a C program to simulate the MVT and MFT memory management techniques

3.1 DESCRIPTION

MFT (Multiprogramming with a Fixed number of Tasks) is one of the old memory management techniques in which 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. MVT (Multiprogramming with a Variable number of Tasks) is the memory management technique in which 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.

3.2 AIM: Simulate Multiple Programming with fixed Number of Tasks (MFT**)**

3.2.1 HARDWARE REQUIREMENTS: Intel based Desktop Pc
RAM of 512 MB

3.2.2 SOFTWARE REQUIREMENTS:
Turbo C/ Borland C.

3.3 THEORY:

Multiple Programming with fixed Number of Tasks (MFT) Algorithm

Background:

IBM in their Mainframe Operating System OS/MFT implements the MFT concept. *OS/MFT* uses Fixed partitioning concept to load programs into Main memory.

Fixed Partitioning:

- In fixed partitioning concept, RAM is divided into set of fixed partition of equal Size
- Programs having the Size Less than the partition size are loaded into Memory
- Programs Having Size more then the size of Partitions Size is rejected
- The program having the size less than the partition size will lead to internal Fragmentation.
- If all partitions are allocated and a new program is to be loaded, the program that lead to Maximum Internal Fragmentation can be replaced

ALGORITHM:

Step1: start
Step2: Declare variables.
Step3: Enter total memory size.
Step4: Read the no of partitions to be divided.
Step5: Allocate memory for os.
Step6:calculate available memory by subtracting the memory of os from total memory
Step7: calculate the size of each partition by dividing available memory with no of partitions.
Step8: Read the number of processes and the size of each process.
Step9: If size of process<= size of partition then allocate memory to that process.
Step10: Display the wastage of memory.
Step11: Stop .

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
main()
{
        int ms,i,ps[20],n,size,p[20],s,intr=0;
        clrscr();
        printf("Enter size of memory:");
        scanf("%d",&ms);
        printf("Enter memory for OS:");
        scanf("%d",&s);
        ms-=s;
        printf("Enter no.of partitions to be divided:");
        scanf("%d",&n);
        size=ms/n;
        for(i=0;i<n;i++)
        {
                printf("Enter process size");
                scanf("%d ",&ps[i]);
                if(ps[i]<=size)
                {
                        intr=intr+size-ps[i];
                        printf("process%d is allocated\n",p[i]);
                }
                else
                        printf("process%d is blocked",p[i]);
        }
        printf("total fragmentation is %d",intr);
        getch();
}
```

OUTPUT:

Enter total memory size : 50

Enter memory for OS :10

Enter no.of partitions to be divided:4

Enter size of page : 10
Enter size of page : 9
Enter size of page : 9
Enter size of page : 8

Internal Fragmentation is = 4

VIVA QUESTIONS

1. The problem of fragmentation arises in _____.

1)Static storage allocation 2) Stack allocation storage

3 Stack allocation with dynamic binding 4 Heap allocation

2.Boundary registers _____.

1 Are available in temporary program variable storage

2 Are only necessary with fixed partitions

3 Track the beginning and ending the program

4 Track page boundaries

3.The principle of locality of reference justifies the use of _____.

1 Virtual Memory 2 Interrupts

3 Main memory 4 Cache memory

4. In memory management , a technique called as paging, physical memory is broken into fixed-sized blocks called _____.

1) Pages 2) Frames 3) Blocks 4) Segments

5.Demand paged memory allocation

1 allows the virtual address space to be independent of the physical memory

2 allows the virtual address space to be a multiple of the physical memory size

3 allows deadlock tobe detected in paging schemes

4 is present only in Windows NT

EXPERIMENT :3 b)

NAME OF EXPERIMENT: multiple Programming with Varible Number of Tasks (MVT) :

AIM: Simulate multiple Programming with Varible Number of Tasks (MVT)

HARDWARE REQUIREMENTS: Intel based Desktop Pc
RAM of 512 MB
SOFTWARE REQUIREMENTS:
Turbo C/ Borland C.

THEORY:

multiple Programming with Varible Number of Tasks (MVT) Algorithm

Background:

IBM in their Mainframe Operating 'System OS/MVT implements the MVT concept. OSIMVT uses Dynamic Partition concept to load programs into Main memory.

Dynamic Partitioning:

- Initially RAM is portioned according to the of programs to be loaded into Memory till such time no other program can be loaded.
- The Left over Memory is called a hole which is too small too fit any process.
- When a new program is to be into Memory Look for the partition, Which Leads to least External fragmentation and load the Program.
- The space that is not used in a partition is called as External Fragmentation

ALGORITHM:

Step1: start
Step2: Declare variables.
Step3: Enter total memory size.
Step4: Read the no of processes
Step5: Allocate memory for os.
Step6: read the size of each process
Step7:calculate available memory by subtracting the memory of os from total memory
Step8: If available memory >= size of process then allocate memory to that process.
Step9: Display the wastage of memory.
Step10: Stop .

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
main()
{
        int i,m,n,tot,s[20];
        clrscr();
        printf("Enter total memory size:");
        scanf("%d",&tot);
        printf("Enter no. of processes:");
        scanf("%d",&n);
        printf("Enter memory for OS:");
        scanf("%d",&m);
        for(i=0;i<n;i++)
        {
                printf("Enter size of process %d:",i+1);
                scanf("%d",&s[i]);
        }
        tot=tot-m;
        for(i=0;i<n;i++)
        {
                if(tot>=s[i])
                {
                        printf("Allocate memory to  process %d\n",i+1);
                        tot=tot-s[i];
                }
                else
                        printf("process p%d is blocked\n",i+1);
        }


        printf("External Fragmentation is=%d",tot);
        getch();
}
```

OUTPUT:

```
Enter total memory size : 50
Enter no.of pages       : 4
Enter memory for OS    :10

Enter size of page  : 10
Enter size of page  : 9
Enter size of page  : 9
Enter size of page  : 10

External Fragmentation is = 2
```

VIVA QUESTIONS:

1. Explain about MFT?
2. Full form of MFT_____
3. Full form of MVT_____
4. differentiate MFT and MVT?
5. The _____ Memory is called a hole.
6. OSIMVT uses _____concept to load programs into Main memory.

7.OS/MFT uses _____ concept to load programs into Main memory.

EXPERIMENT -4

OBJECTIVE

Write a C program to simulate the following file organization techniques a) Single level directory b) Two level directory c) Hierarchical

3.1 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. In the two-level directory system, each user has own user file directory (UFD). The system maintains a master block that has one entry for each user. This master block contains the addresses of the directory of the users. When a user job starts or a user logs in, the system's master file directory (MFD) is searched. When a user refers to a particular file, only his own UFD is searched. This effectively solves the name collision problem and isolates users from one another. Hierarchical directory structure allows users to create their own subdirectories and to organize their files accordingly. A tree is the most common directory structure. The tree has a root directory, and every file in the system has a unique path name. A directory (or subdirectory) contains a set of files or subdirectories.

PROGRAM

SINGLE LEVEL DIRECTORY ORGANIZATION

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

void main()
{
        int i,ch;
        char f[30];
        clrscr();
        dir.fcnt = 0;
        printf("\nEnter name of directory -- ");
        scanf("%s", dir.dname);
        while(1)
        {
                printf("\n\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];
                        break;
                }
```

```
                Else
                        dir.fcnt--;
                break;

        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;
         default: exit(0);
        }

    }
    getch();
}
```

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

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
```

1.

```
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
```

b) TWO LEVEL DIRECTORY ORGANIZATION

```
#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\t
                Enter 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++)
```

1.

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

case 4: printf("\nEnter name of the directory -- "); scanf("%s",d);
        for(i=0;i<dcnt;i++)
        {
                if(strcmp(d,dir[i].dname)==0)
                {
                        printf("Enter the name of the file -- "); scanf("%s",f);
                        for(k=0;k<dir[i].fcnt;k++)

                        {
                                if(strcmp(f, dir[i].fname[k])==0)
                                {
                                printf("File %s is found ",f); goto jmp1;
                                }
                        }



                        printf("File %s not found",f); goto jmp1;
                }
        }
        printf("Directory %s not found",d); jmp1: break;
case 5: if(dcnt==0)
                printf("\nNo Directory's ");
        else
        {
                printf("\nDirectory\tFiles");
```

```
                                        for(i=0;i<dcnt;i++)
                                        {
                                                printf("\n%s\t\t",dir[i].dname);
                                                for(k=0;k<dir[i].fcnt;k++)
                                                        printf("\t%s",dir[i].fname[k]);
                                        }

                                }
                                break;
                        default:exit(0);
                }

        }
        getch();
}
```

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 -- 2

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

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

1.

```
Directory       Files
DIR1            A1      A2
DIR2            B1

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

Enter name of the directory – DIR
Directory not found

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

Enter name of the directory – DIR1
Enter name of the file --   A2
File A2 is deleted

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

4c) HIERARCHICAL DIRECTORY ORGANIZATION

```
#include<stdio.h>
 #include<graphics.h> struct tree_element
 {
        char name[20];
        int x, y, ftype, lx, rx, nc, level; struct tree_element *link[5];
};
typedef struct tree_element node;
void main()

 int gd=DETECT,gm; node
*root; root=NULL;
clrscr();
create(&root,0,"root",0,639,320);
clrscr();
initgraph(&gd,&gm,"c:\tc\BGI");
display(root);
getch();
closegraph();
}
create(node **root,int lev,char *dname,int lx,int rx,int x)
{
        int i, gap;
        if(*root==NULL)
        {


                (*root)=(node *)malloc(sizeof(node)); printf("Enter name
                of dir/file(under %s) : ",dname); fflush(stdin);
                gets((*root)->name); printf("enter 1 for
                Dir/2 for file :"); scanf("%d",&(*root)-
```

1.

```
                >ftype); (*root)->level=lev; (*root)-
                >y=50+lev*50; (*root)->x=x;
                (*root)->lx=lx; (*root)-
                >rx=rx; for(i=0;i<5;i++)
                        (*root)->link[i]=NULL;
                if((*root)->ftype==1)
                {
                        printf("No of sub directories/files(for %s):",(*root)->name); scanf("%d",&(*root)>nc);
                        if((*root)->nc==0)
                                        gap=rx-lx;
                        else
                                        gap=(rx-lx)/(*root)->nc;
                        for(i=0;i<(*root)->nc;i++)
                                create(&((*root)>link[i]),lev+1,(*root)>name,lx+gap*i,lx+gap*i+gap,
                                                                                lx+gap*i+gap/2);
                }
                else
                        (*root)->nc=0;
        }
}
display(node *root)
{
        int i; settextstyle(2,0,4);
        settextjustify(1,1);
        setfillstyle(1,BLUE);
        setcolor(14); if(root
        !=NULL)
        {
                for(i=0;i<root->nc;i++) line(root->x,root->y,root->link[i]-
                >x,root->link[i]->y); if(root->ftype==1) bar3d(root->x-20,root-
                >y-10,root->x+20,root>y+10,0,0); else


                fillellipse(root->x,root->y,20,20);
                outtextxy(root->x,root->y,root->name);
                for(i=0;i<root->nc;i++) display(root->link[i]);



         }
  }
```

INPUT

Enter Name of dir/file(under root): ROOT Enter 1 for Dir/2 for File: 1

No of subdirectories/files(for ROOT): 2 Enter Name of dir/file(under ROOT): USER1 Enter 1 for Dir/2 for File: 1

No of subdirectories/files(for USER1): 1

Enter Name of dir/file(under USER1): SUBDIR1 Enter 1 for Dir/2 for File: 1

No of subdirectories/files(for SUBDIR1): 2 Enter Name of dir/file(under USER1): JAVA Enter 1 for Dir/2 for File: 1

No of subdirectories/files(for JAVA): 0 Enter Name of dir/file(under SUBDIR1): VB Enter 1 for Dir/2 for File: 1

No of subdirectories/files(for VB): 0

Enter Name of dir/file(under ROOT): USER2 Enter 1 for Dir/2 for File: 1

No of subdirectories/files(for USER2): 2 Enter Name of dir/file(under ROOT): A Enter 1 for Dir/2 for File: 2

Enter Name of dir/file(under USER2): SUBDIR2 Enter 1 for Dir/2 for File: 1

No of subdirectories/files(for SUBDIR2): 2 Enter Name of dir/file(under SUBDIR2): PPL Enter 1 for Dir/2 for File: 1

No of subdirectories/files(for PPL): 2 Enter Name of dir/file(under PPL): B Enter 1 for Dir/2 for File: 2

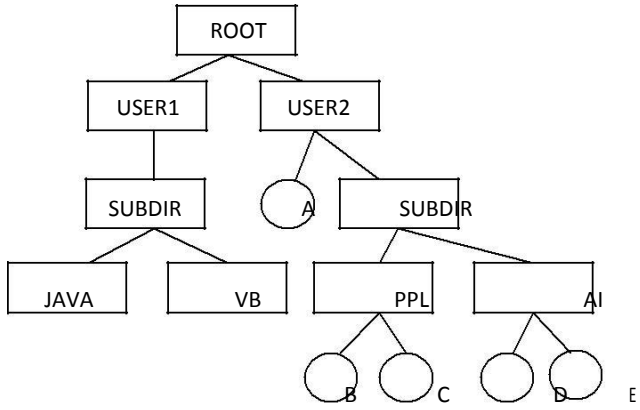
Enter Name of dir/file(under PPL): C Enter 1 for Dir/2 for File: 2

Enter Name of dir/file(under SUBDIR): AI Enter 1 for Dir/2 for File: 1

No of subdirectories/files(for AI): 2 Enter Name of dir/file(under AI): D Enter 1 for Dir/2 for File: 2

Enter Name of dir/file(under AI): E Enter 1 for Dir/2 for File: 2

OUTPUT



EXPERIMENT- 5

5.1 OBJECTIVE

Write a C program to simulate Bankers algorithm for the purpose of deadlock avoidance.

5.2 DESCRIPTION

In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are 4held by other waiting processes. This situation is called a deadlock. Deadlock avoidance is one of the techniques for handling deadlocks. This approach requires that the operating system be given in advance additional information concerning which resources a process will request and use during its lifetime. With this additional knowledge, it can decide for each request whether or not the process should wait. To decide whether the current request can be satisfied or must be delayed, the system must consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process.

Banker's algorithm is a deadlock avoidance algorithm that is applicable to a system with multiple instances of each resource type.

NAME OF EXPERIMENT: Simulate Banker's Algorithm for Deadlock Avoidance.

AIM: Simulate Banker's Algorithm for Deadlock Avoidance to find whether the system is in safe state or not.

HARDWARE REQUIREMENTS: Intel based Desktop Pc

RAM of 512 MB

SOFTWARE REQUIREMENTS: Turbo C/ Borland C.

THEORY:

DEAD LOCK AVOIDANCE

To implement deadlock avoidance & Prevention by using Banker's Algorithm.

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.

Data structures

- n-Number of process, m-number of resource types.
- Available: Available[j]=k, k – instance of resource type Rj is available.
- Max: If max[i, j]=k, Pi may request at most k instances resource Rj.
- Allocation: If Allocation [i, j]=k, Pi allocated to k instances of resource Rj
- Need: If Need[I, j]=k, Pi may need k more instances of resource type Rj,
  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 Pi, If request i=[j]=k, then process Pi wants k instances of resource type Rj.

1. if Request<=Need I go to step 2. Otherwise raise an error condition.
2. if Request<=Available go to step 3. Otherwise Pi must since the resources are available.
3. Have the system pretend to have allocated the requested resources to process Pi 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 Pi is allocated its resources. However if the state is unsafe, the Pi 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.

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
struct da {
int max[10],al[10],need[10],before[10],after[10];
}p[10];
void main() {
int i,j,k,l,r,n,tot[10],av[10],cn=0,cz=0,temp=0,c=0;
clrscr();
printf("\n Enter the no of processes:");
scanf("%d",&n);
printf("\n Enter the no of resources:");
scanf("%d",&r);
for(i=0;i<n;i++) {
printf("process %d \n",i+1);
for(j=0;j<r;j++) {
printf("maximum value for resource %d:",j+1);
scanf("%d",&p[i].max[j]);
}
```

```
for(j=0;j<r;j++) {
printf("allocated from resource %d:",j+1);
scanf("%d",&p[i].al[j]);
p[i].need[j]=p[i].max[j]-p[i].al[j];
}
}
for(i=0;i<r;i++) {
printf("Enter total value of resource %d:",i+1);
scanf("%d",&tot[i]);
}
for(i=0;i<r;i++) {
for(j=0;j<n;j++)
temp=temp+p[j].al[i];
av[i]=tot[i]-temp;
temp=0;
}
printf("\n\t max allocated needed total avail");
for(i=0;i<n;i++) {
printf("\n P%d \t",i+1);
for(j=0;j<r;j++)
printf("%d",p[i].max[j]);
printf("\t");
for(j=0;j<r;j++)
printf("%d",p[i].al[j]);
printf("\t");
for(j=0;j<r;j++)
printf("%d",p[i].need[j]);
printf("\t");
for(j=0;j<r;j++)
{
if(i==0)
printf("%d",tot[j]);
}
printf(" ");
for(j=0;j<r;j++) {
if(i==0)
printf("%d",av[j]);
}
}
printf("\n\n\t AVAIL BEFORE \t AVAIL AFTER");
for(l=0;l<n;l++)
{
for(i=0;i<n;i++)
{
for(j=0;j<r;j++)
{
if(p[i].need[j]>av[j])
cn++;
if(p[i].max[j]==0)
cz++;
}
if(cn==0 && cz!=r)
{
for(j=0;j<r;j++)
{
```

```
p[i].before[j]=av[j]-p[i].need[j];
p[i].after[j]=p[i].before[j]+p[i].max[j];
av[j]=p[i].after[j];
p[i].max[j]=0;
}
printf("\n p%d \t",i+1);
for(j=0;j<r;j++)
printf("%d",p[i].before[j]);
printf("\t");
for(j=0;j<r;j++)
printf("%d",p[i].after[j]);
cn=0;
cz=0;
c++;
break;
}
else {
cn=0;cz=0;
}
}
}
if(c==n)
printf("\n the above sequence is a safe sequence");
else
printf("\n deadlock occured");
getch();
}
```

OUTPUT:

```
//TEST CASE 1:

 ENTER THE NO. OF PROCESSES:4

 ENTER THE NO. OF RESOURCES:3
PROCESS 1
MAXIMUM VALUE FOR RESOURCE 1:3
MAXIMUM VALUE FOR RESOURCE 2:2
MAXIMUM VALUE FOR RESOURCE 3:2
ALLOCATED FROM RESOURCE 1:1
ALLOCATED FROM RESOURCE 2:0
ALLOCATED FROM RESOURCE 3:0
PROCESS 2
MAXIMUM VALUE FOR RESOURCE 1:6
MAXIMUM VALUE FOR RESOURCE 2:1
MAXIMUM VALUE FOR RESOURCE 3:3
ALLOCATED FROM RESOURCE 1:5
ALLOCATED FROM RESOURCE 2:1
ALLOCATED FROM RESOURCE 3:1
PROCESS 3
MAXIMUM VALUE FOR RESOURCE 1:3
MAXIMUM VALUE FOR RESOURCE 2:1
MAXIMUM VALUE FOR RESOURCE 3:4
ALLOCATED FROM RESOURCE 1:2
ALLOCATED FROM RESOURCE 2:1
ALLOCATED FROM RESOURCE 3:1
```

PROCESS 4
MAXIMUM VALUE FOR RESOURCE 1:4
MAXIMUM VALUE FOR RESOURCE 2:2
MAXIMUM VALUE FOR RESOURCE 3:2
ALLOCATED FROM RESOURCE 1:0
ALLOCATED FROM RESOURCE 2:0
ALLOCATED FROM RESOURCE 3:2
ENTER TOTAL VALUE OF RESOURCE 1:9
ENTER TOTAL VALUE OF RESOURCE 2:3
ENTER TOTAL VALUE OF RESOURCE 3:6

RESOURCES	ALLOCATED	NEEDED	TOTAL	AVAIL
P1 322	100	222	936	112
P2 613	511	102		
P3 314	211	103		
P4 422	002	420		

	AVAIL BEFORE	AVAIL AFTER
P 2	010	623
P 1	401	723
P 3	620	934
P 4	514	936

THE ABOVE SEQUENCE IS A SAFE SEQUENCE

VIVA QUESTIONS:

1. Differentiate deadlock avoidance and fragmentation

2.Tell me the real time example where this deadlock occurs?

3.How do we calculate the need for process?

4.What is the name of the algorithm to avoid deadlock?

5.Banker's algorithm for resource allocation deals with

(A)Deadlock prevention. (B)Deadlock avoidance.

(C)Deadlock recovery. (D)Mutual exclusion

EXPERIMENT- 6

6.1 OBJECTIVE

Write a C program to simulate Bankers algorithm for the purpose of deadlock prevention.

6.2 DESCRIPTION

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

Banker's algorithm is a deadlock avoidance algorithm that is applicable to a system with multiple instances of each resource type.

NAME OF EXPERIMENT: Simulate Algorithm for Deadlock prevention.

AIM: Simulate Algorithm for Deadlock prevention .

HARDWARE REQUIREMENTS: Intel based Desktop Pc
RAM of 512 MB
SOFTWARE REQUIREMENTS: Turbo C/ Borland C.

THEORY:

Deadlock Definition:
A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause (including itself).Waiting for an event could be:

- waiting for access to a critical section
- waiting for a resource Note that it is usually a non-preemptable (resource).

Conditions for Deadlock :

•Mutual exclusion: resources cannot be shared.
•Hold and wait:processes request resources incrementally, and hold on to What they've got.
•No preemption: resources cannot be forcibly taken from processes.
•Circular wait: circular chain of waiting, in which each process is waiting for a resource held by the next process in the chain.

Strategies for dealing with Deadlock :
•ignore the problem altogether
•detection and recovery
•avoidance by careful resource allocation
•prevention by structurally negating one of the four necessary conditions.

Deadlock Prevention :

Difference from avoidance is that here, the system itself is built in such a way that there are no deadlocks. Make sure atleast one of the 4 deadlock conditions is never satisfied. This may however be even more conservative than deadlock avoidance strategy.

Algorithm:

1.Start
2.Attacking Mutex condition : never grant exclusive access. but this may not be possible for several resources.
3..Attacking preemption: not something you want to do.
4.Attacking hold and wait condition : make a process hold at the most 1 resource at a time.make all the requests at the beginning. All or nothing policy. If you feel,retry. eg. 2-phase locking
5.Attacking circular wait: Order all the resources. Make sure that the requests are issued in the correct order so that there are no cycles present in the resource graph. Resources numbered 1 ... n. Resources can be requested only in increasing order. ie. you cannot request a resource whose no is less than any you may be holding.
6.Stop

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
int max[10][10],alloc[10][10],need[10][10],avail[10],i,j,p,r,finish[10]={0},flag=0;
main( )
{
clrscr( );
printf("\n\nSIMULATION OF DEADLOCK PREVENTION");
printf("Enter no. of processes, resources");
scanf("%d%d",&p,&r);printf("Enter allocation matrix");
for(i=0;i<p;i++)
for(j=0;j<r;j++)
scanf("%d",&alloc[i][j]);
printf("enter max matrix");
for(i=0;i<p;i++) /*reading the maximum matrix and availale matrix*/
for(j=0;j<r;j++)
scanf("%d",&max[i][j]);
printf("enter available matrix");
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];
fun(); /*calling function*/
if(flag==0)
{i
f(finish[i]!=1)
{
printf("\n\n Failing :Mutual exclusion");
for(j=0;j<r;j++)
{ /*checking for mutual exclusion*/
if(avail[j]<need[i][j])
avail[j]=need[i][j];
}fun();
printf("\n By allocating required resources to process %d dead lock is prevented ",i);
printf("\n\n lack of preemption");
for(j=0;j<r;j++)
{
```

```
if(avail[j]<need[i][j])
avail[j]=need[i][j];
alloc[i][j]=0;
}

fun( );
printf("\n\n daed lock is prevented by allocating needed resources");
printf(" \n \n failing:Hold and Wait condition ");
for(j=0;j<r;j++)
{ /*checking hold and wait condition*/
if(avail[j]<need[i][j])
avail[j]=need[i][j];
}
fun( );
printf("\n AVOIDING ANY ONE OF THE CONDITION, U CAN PREVENT DEADLOCK");
}
}
getch( );
}
fun()
{
while(1)
{
for(flag=0,i=0;i<p;i++)
{
if(finish[i]==0)
{
for(j=0;j<r;j++)
{
if(need[i][j]<=avail[j])
continue;
elsebreak;
}
if(j==r)
{
for(j=0;j<r;j++)
avail[j]+=alloc[i][j];
flag=1;
finish[i]=1;
}
}
}
if(flag==0)
break;
}
}
```

Output:

```
SIMULATION OF DEADLOCK PREVENTION
Enter no. of processes, resources 3, 2
enter allocation matrix 2 4 5
                        3 4 5
Enter max matrix4 3 4
                  5 6 1
Enter available matrix2
```

5

Failing : Mutual Exclusion
by allocating required resources to process dead is prevented
Lack of no preemption deadlock is prevented by allocating needed resources
Failing : Hold and Wait condition

VIVA QUESTIONS:

1. The Banker's algorithm is used for _____.

2._____ is the situation in which a process is waiting on another process,which is also waiting on another process ... which is waiting on the first process. None of the processes involved in this circular wait are making progress.

3.what is safe state?

4.What are the conditions that cause deadlock?

5.How do we calculate the need for process?

EXPERIMENT- 7

7.1 OBJECTIVE

Write a C program to simulate page replacement algorithms a) FIFO b) LRU c) LFU

7.2 DESCRIPTION

Page replacement is basic to demand paging. It completes the separation between logical memory and physical memory. With this mechanism, an enormous virtual memory can be provided for programmers on a smaller physical memory. There are many different page-replacement algorithms. Every operating system probably has its own replacement scheme. A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen. If the recent past is used as an approximation of the near future, then the page that has not been used for the longest period of time can be replaced. This approach is the Least Recently Used (LRU) algorithm. LRU replacement associates with each page the time of that page's last use. When a page must be replaced, LRU chooses the page that has not been used for the longest period of time. Least frequently used (LFU) page-replacement algorithm requires that the page with the smallest count be replaced. The reason for this selection is that an actively used page should have a large reference count.

NAME OF EXPERIMENT: Simulate page replacement algorithms:

a) FIFO

AIM: Simulate FIFO page replacement algorithms.

HARDWARE REQUIREMENTS: Intel based Desktop Pc
RAM of 512 MB

SOFTWARE REQUIREMENTS: Turbo C/ Borland C.

THEORY:

FIFO algorithm:

The simpler page replacement algorithm is a FIFO algorithm. A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replace, the oldest page is chosen. We can create a FIFO queue to hold all pages in memory. We replace the page at the head of the queue when a page is brought into memory; we insert it at the tail of the queue.

7	0	1	2	0
7	7	7	2	
	0	0	0	
		1	1	

3	0	4	2	3	0	3	2
2	2	4	4	4	0		
3	3	3	2	2	2		
4	0	0	0	3	3		

1	2	0	1
0	0		
1	1		
3	2		

7	0	1
7	7	7
1	0	0
2	2	1

ALGORITHM:

1. Start
2. Read the number of frames
3. Read the number of pages
4. Read the page numbers
5. Initialize the values in frames to -1
6. Allocate the pages in to frames in First in first out order.
7. Display the number of page faults.
8. stop

PROGRAM

FIFO PAGE REPLACEMENT ALGORITHM

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

        printf("\n The Page Replacement Process is --
        \n"); for(i=0;i<n;i++)
        {
                for(k=0;k<f;k++)
                {
                        if(m[k]==rs[i])
                                break;
                }
                if(k==f)
                {
```

```
                        m[count++]=rs[i];
                        pf++;
                }
                for(j=0;j<f;j++)
                        printf("\t%d",m[j]);
                if(k==f)
                        printf("\tPF No.
                %d",pf); printf("\n");
                if(count==f)
                        count=0;
        }
        printf("\n The number of Page Faults using FIFO are
        %d",pf);
         getch();
}
```

INPUT

Enter the length of reference string – 20
Enter the reference string -- 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1
Enter no. of frames -- 3

OUTPUT

The Page Replacement Process is –

7	-1	-1	PF No. 1
7	0	-1	PF No. 2
7	0	1	PF No. 3
2	0	1	PF No. 4
2	0	1	
2	3	1	PF No. 5
2	3	0	PF No. 6
4	3	0	PF No. 7
4	2	0	PF No. 8
4	2	3	PF No. 9
0	2	3	PF No. 10
0	2	3	
0	2	3	
0	1	3	PF No. 11
0	1	2	PF No. 12
0	1	2	
0	1	2	
7	1	2	PF No. 13
7	0	2	PF No. 14
7	0	1	PF No. 15

The number of Page Faults using FIFO are 15

7.2 LRU PAGE REPLACEMENT ALGORITHM

NAME OF EXPERIMENT: Simulate page replacement algorithms:
b) LRU

AIM: Simulate LRU page replacement algorithms

HARDWARE REQUIREMENTS: Intel based Desktop Pc
RAM of 512 MB
SOFTWARE REQUIREMENTS:
Turbo C/ Borland C.

ALGORITHM :

1. Start
2. Read the number of frames
3. Read the number of pages
4. Read the page numbers
5. Initialize the values in frames to -1
6. Allocate the pages in to frames by selecting the page that has not been used for the longest period of time.
7. Display the number of page faults.
8. stop

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
main()
{
        int i, j , k, min, rs[25], m[10], count[10], flag[25], n, f, pf=0, next=1; clrscr();
        printf("Enter the length of reference string -- "); scanf("%d",&n);
        printf("Enter the reference string -- "); for(i=0;i<n;i++)
        {
                scanf("%d",&rs[i]);
                flag[i]=0;
        }
        printf("Enter the number of frames -- "); scanf("%d",&f);
        for(i=0;i<f;i++)
        {
                count[i]=0; m[i]=-1;
        }
        printf("\nThe Page Replacement process is -- \n"); for(i=0;i<n;i++)
        {
                for(j=0;j<f;j++)
                {
                        if(m[j]==rs[i])
                        {
                                flag[i]=1;
```

```
                                        count[j]=next;
                                        next++;
                                }

                        }
                        if(flag[i]==0)
                        {
                                if(i<f)
                                {

                                        m[i]=rs[i];
                                        count[i]=next;
                                        next++;
                                }
                                Else+
                                {
                                        min=0;
                                        for(j=1;j<f;j++)
                                                if(count[min] > count[j]) min=j;

                                        m[min]=rs[i];
                                        count[min]=next;
                                        next++;
                                }
                                pf++;
                        }
                        for(j=0;j<f;j++) printf("%d\t", m[j]);
                        if(flag[i]==0)
                                printf("PF No. -- %d" , pf); printf("\n");
                }
                printf("\nThe number of page faults using LRU are %d",pf); getch();
}
```

INPUT

Enter the length of reference string -- 20

Enter the reference string -- 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1 Enter the number of frames -- 3

OUTPUT

The Page Replacement process is --

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

1 3 2
1 0 2 PF No. – 11
1 0 2
1 0 7 PF No. – 12
1 0 7

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

1.

NAME OF EXPERIMENT: Simulate page replacement algorithms:

c)LFU

AIM: Simulate LFU page replacement algorithms .

HARDWARE REQUIREMENTS: Intel based Desktop Pc
RAM of 512 MB

SOFTWARE REQUIREMENTS:
Turbo C/ Borland C.

ALGORITHM:

1. Start
2. Read the number of frames
3. Read the number of pages
4. Read the page numbers
5. Initialize the values in frames to -1
6. Allocate the pages in to frames by selecting the page that will not be used for the longest period of time.
7. Display the number of page faults.
8. stop

ROGRAM

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

main()
{
        int rs[50], i, j, k, m, f, cntr[20], a[20], min, pf=0; clrscr();
        printf("\nEnter number of page references -- "); scanf("%d",&m);

        printf("\nEnter the reference string -- "); for(i=0;i<m;i++)
                scanf("%d",&rs[i]);

        printf("\nEnter the available no. of frames -- "); scanf("%d",&f);

        for(i=0;i<f;i++)
        {




                cntr[i]=0; a[i]=-1;
        }
        Printf("\nThe Page Replacement Process is – \n"); for(i=0;i<m;i++)
        {

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

1.

```
                        break;
                }


        if(j==f)
        {
                min = 0; for(k=1;k<f;k++)
                        if(cntr[k]<cntr[min])
                                min=k;
                a[min]=rs[i];
                cntr[min]=1;
                pf++;
        }
        printf("\n");
        for(j=0;j<f;j++)
                printf("\t%d",a[j]);
        if(j==f)
                printf("\tPF No. %d",pf);
}
printf("\n\n Total number of page faults -- %d",pf);
 getch();
}
```

INPUT

Enter number of page references -- 10
Enter the reference string -- 1 2 3 4 5 2 5 2 5 1 4 3
Enter the available no. of frames -- 3

OUTPUT

The Page Replacement Process is –

1	-1	-1	PF No. 1
1	2	-1	PF No. 2
1	2	3	PF No. 3
4	2	3	PF No. 4
5	2	3	PF No. 5
5	2	3	
5	2	3	
5	2	1	PF No. 6
5	2	4	PF No. 7
5	2	3	PF No. 8

Total number of page faults -- 8

1.

EXPERIMENT -8

8.1 OBJECTIVE

Write a C program to simulate paging technique of memory management.

8.2 DESCRIPTION

In computer operating systems, paging is one of the memory management schemes by which a computer stores and retrieves data from the secondary storage for use in main memory. In the paging memory-management scheme, the operating system retrieves data from secondary storage in same-size blocks called pages. Paging is a memory-management scheme that permits the physical address space a process to be noncontiguous. The basic method for implementing paging involves breaking physical memory into fixed-sized blocks called frames and breaking logical memory into blocks of the same size called pages. When a process is to be executed, its pages are loaded into any available memory frames from their source.

AIM: To simulate paging technique of memory management.

PROGRAM

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

main()
{
        int ms, ps, nop, np, rempages, i, j, x, y, pa,
        offset; int s[10], fno[10][20];

        clrscr();

        printf("\nEnter the memory size --
        "); scanf("%d",&ms);

        printf("\nEnter the page size --
        "); scanf("%d",&ps);

        nop = ms/ps;
        printf("\nThe no. of pages available in memory are -- %d ",nop);

        printf("\nEnter number of processes --
        "); scanf("%d",&np);

        rempages = nop;

        for(i=1;i<=np;i++)
        {

                printf("\nEnter no. of pages required for p[%d]--
                ",i); scanf("%d",&s[i]);

                if(s[i] >rempages)
                {
```

```
                        printf("\nMemory is
                        Full"); break;
                }
                rempages = rempages - s[i];



                printf("\nEnter pagetable for p[%d] -,i);
                for(j=0;j<s[i];j++)
                        scanf("%d",&fno[i][j]);
        }

        printf("\nEnter Logical Address to find Physical Address ");
        printf("\nEnter process no. and pagenumber and offset ");

        scanf("%d %d %d",&x,&y, &offset);

if(x>np || y>=s[i] || offset>=ps)
        printf("\nInvalid Process or Page Number or offset");
else
{
        pa=fno[x][y]*ps+offset;
        printf("\nThe Physical Address is -- %d",pa);
}
getch();
}
```

INPUT

```
Enter the memory size – 1000
Enter the page size --          100
The no. of pages available in memory are -- 10
Enter number of processes --            3
Enter no. of pages required for p[1] --         4
Enter pagetable for p[1] ---            8       6       9       5

Enter no. of pages required for p[2] --         5
Enter pagetable for p[2] ---            1       4       5       7       3

Enter no. of pages required for p[3] --         5
```

OUTPUT

```
Memory is Full

Enter Logical Address to find Physical Address
Enter process no. and pagenumber and offset -- 2                3       60
The Physical Address is --     760
```

EXPERIMENT -9

9.1 OBJECTIVE

*Write a C program to simulate the following contiguous memory allocation techniques a) Worst-fit b) Best-fit c) First-fit

9.2 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

void main()
{
        int
        frag[max],b[max],f[max],i,j,nb,nf,temp;
        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)
                ;
        }
```

1.

```
        for(i=1;i<=nf;i++)
        {
                for(j=1;j<=nb;j++)
                {
                        if(bf[j]!=1)
                        {
                                temp=b[j]-
                                f[i];
                                if(temp>=0
                                )
                                {
                                        ff[i]=j;
                                        break;
                                }
                        }
                }

                frag[i]=temp;
                bf[ff[i]]=1;
        }
        printf("\nFile_no:\tFile_size :\tBlock_no:\tBlock_size:\tFragement"); for(i=1;i<=nf;i++)
                                        printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d",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	1	5	4
2	4	3	7	3

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);
```

1.

```
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

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,highest=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)
                                {
                                        ff[i]=j;
                                        highest=temp;
                                }
                }
        }
          frag[i]=highest;
          bf[ff[i]]=1;
          highest=0;
}
printf("\nFile_no:\tFile_size :\tBlock_no:\tBlock_size:\tFragement"); for(i=1;i<=nf;i++)
                                                printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d",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 -10

10.1 OBJECTIVE

*Write a C program to simulate diskscheduling algorithms a) FCFS b) SCAN c) C-SCAN

10.2 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

10.3 PROGRAM

10.3.1 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();

}
```

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	45
58	3
60	2
70	10
18	52
90	72
150	60
160	10
184	24

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 traveresed"); scanf("%d'",&n);
        printf("enter the position of head"); scanf("%d",&h);
        t[0]=0;t[1]=h; printf("enter the tracks");
        for(i=2;i<n+2;i++)
                scanf("%d",&t[i]);
        for(i=0;i<n+2;i++)
        {
                for(j=0;j<(n+2)-i-1;j++)
                { if(t[j]>t[j+1])
                                {
                                        temp=t[j];
                                        t[j]=t[j+1];
                                        t[j+1]=temp;
        } } }
        for(i=0;i<n+2;i++)
                if(t[i]==h)
                        j=i;k=i;
        p=0;
        while(t[j]!=0)
        {
                atr[p]=t[j]; j--;
                p++;
        }
        atr[p]=t[j];
        for(p=k+1;p<n+2;p++,k++)
                atr[p]=t[k+1];
```

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();
}
```

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) CSCAN DISK SCHEDULING ALGORITHM

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

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+}
        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

EXPERIMENT 11

11.1 OBJECTIVE

*Write a C program to simulate page replacement algorithms a) Optimal

11.2 DESCRIPTION

Optimal page replacement algorithm has the lowest page-fault rate of all algorithms and will never suffer from Belady's anomaly. The basic idea is to replace the page that will not be used for the longest period of time. Use of this page-replacement algorithm guarantees the lowest possible page fault rate for a fixed number of frames. Unfortunately, the optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string.

11.3 PROGRAM

```
#include<stdio>
 int n;
main()
{
        int
        seq[30],fr[5],pos[5],find,flag,max,i,j,m,k,t,s;
        int count=1,pf=0,p=0;
        float
        pfr;
        clrscr(
        );
        printf("Enter maximum limit of the sequence:
        "); scanf("%d",&max);
        printf("\nEnter the sequence:
        ");
        for(i=0;i<max;i++)
                scanf("%d",&seq[i]);
                printf("\nEnter no. of frames:
                "); scanf("%d",&n);
                fr[0]=seq[0];
                pf++;
                printf("%d\t",fr[0]);
                i=1;
                while(count<n)
                {
                        flag=1;
                        p++;
                        for(j=0;j<i;j++)
                        {
                                if(seq[i]==seq[j])
                                flag=0;
                        }
                        if(flag!=0)
                        {
                                fr[count]=seq[i];
                                printf("%d\t",fr[count]);
                                count++;
                                pf++;
```

1.

```
                }
                i++;
        }
        printf("\n");
        for(i=p;i<max;i++)
        {
                flag=1;
                for(j=0;j<n;j++)
                {
                        if(seq[i]==fr[j])
                                flag=0;
                }
                if(flag!=0)



        {
                for(j=0;j<n;j++)
                {
                        m=fr[j];
                        for(k=i;k<max;k++)
                        {
                                if(seq[k]==m)
                                {
                                        pos[j]=k;
                                        break;
                                }
                                else
                                                                                pos[j]=1;
                        }
                }
                for(k=0;k<n;k++)
                {
                        if(pos[k]==1)
                                flag=0;
                }
                if(flag!=0)
                        s=findmax(pos);
                if(flag==0)
                {
                        for(k=0;k<n;k++)
                        {
                                if(pos[k]==1)
                                {
                                        s=k;
                                        break;
                                }
                        }
                }
                fr[s]=seq[i];
                for(k=0;k<n;k++)
                        printf("%d\t",fr[k]);
                pf++;
                printf("\n");
```

1.

```
                        }
                }
                pfr=(float)pf/(float)max;
                printf("\nThe no. of page faults are %d",pf); printf("\nPage fault rate %f",pfr);
                getch();

}
int findmax(int a[])
{
        int max,i,k=0;
        max=a[0];
        for(i=0;i<n;i++)
        {
                if(max<a[i])
                {
                        max=a[i];
                        k=i;
                }
        }
        return k;
}
```

INPUT

Enter number of page references -- 10

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

EXPERIMENT 12

OBJECTIVE

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

12.1 DESCRIPTION

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

12.2 PROGRAM

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

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

INPUT

DINING PHILOSOPHER PROBLEM
Enter the total no. of philosophers: 5
How many are hungry : 3
Enter philosopher 1 position: 2
Enter philosopher 2 position: 4
Enter philosopher 3 position: 5

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.

1.

1.

1.

1.

1.

1.

1.

1.

1.

1.

1.

1.

```
                                        Else
                                                dir.fcnt--;
                                        break;

                                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;
                                default: exit(0);
                        }

                }
                getch();
        }
```

OUTPUT:

```
Enter name of directory --    CSE
1. Create File      2. Delete File   3. Search File
4. Display Files    5. Exit          Enter your choice – 1

Enter the name of the file --   A

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

Enter the name of the file --   B

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

Enter the name of the file --   C
1. Create File      2. Delete File   3. Search File
4. Display Files    5. Exit          Enter your choice – 4

The Files are -- A B  C

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

1.

```
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
                                                    23
```

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

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

TWO LEVEL DIRECTORY ORGANIZATION

```
#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\t
Enter 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]);
                                                dir[i].fcnt++;
                                                printf("File created");
                                                break;
                                        }
                                if(i==dcnt)
                                        printf("Directory %s not found",d);
                                break;
```

1.

```
case 3: printf("\nEnter name of the directory -- ");
        scanf("%s",d);
        for(i=0;i<dcnt;i++)
        {
        if(strcmp(d,dir[i].dname)==0)
        {
                printf("Enter name of the file -- ");
                scanf("%s",f);
                for(k=0;k<dir[i].fcnt;k++)
                {
                        if(strcmp(f, dir[i].fname[k])==0)
```

```
                                        {
                                        printf("File %s is deleted
                                        ",f); dir[i].fcnt--;
                                        strcpy(dir[i].fname[k],dir[i].fname[dir[i].fcnt]);
                                        goto jmp;
                                        }
                                }
                                printf("File %s not found",f);
                                goto jmp;
                        }
                        }
                        printf("Directory %s not found",d);
                        jmp : break;

                case 4: printf("\nEnter name of the directory -- ");
                        scanf("%s",d);
                        for(i=0;i<dcnt;i++)
                        {
                                if(strcmp(d,dir[i].dname)==0)
                                {
                                        printf("Enter the name of the file -- ");
                                        scanf("%s",f); for(k=0;k<dir[i].fcnt;k++)

                                        {
                                                if(strcmp(f, dir[i].fname[k])==0)
                                                {
                                                printf("File %s is found
                                                ",f); goto jmp1;
                                                }
                                        }
                                        printf("File %s not found",f);
                                        goto jmp1;
                                }
                        }
                        printf("Directory %s not found",d);
                jmp1: break;
                case 5: if(dcnt==0)
                                printf("\nNo Directory's ");
                        else
                        {
                                printf("\nDirectory\tFiles");
                                for(i=0;i<dcnt;i++)
                                {
                                        printf("\n%s\t\t",dir[i].dname);
                                        for(k=0;k<dir[i].fcnt;k++)
                                                printf("\t%s",dir[i].fname[k]);
                                }
                        }
                        break;
                default:exit(0);
                }

        }
        getch();
}
```

OUTPUT:

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

1.

4. Search File 5. Display 6. Exit Enter your choice -- 1

Enter name of directory -- DIR1
Directory created

1.

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

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

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

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

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

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

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

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

Directory	Files
DIR1	A1 A2
DIR2	B1

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

Enter name of the directory – DIR
Directory not found

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

Enter name of the directory – DIR1
Enter name of the file -- A2
File A2 is deleted

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

HIERARCHICAL DIRECTORY ORGANIZATION

```
#include<stdio.h>
 #include<graphics.h>
struct tree_element
{
        char name[20];
        int x, y, ftype, lx, rx, nc, level;
        struct tree_element *link[5];
};
typedef struct tree_element node;
void main()
```

```
 int
gd=DETECT,gm;
node *root;
root=NULL; clrscr();
create(&root,0,"root",0,639,320);
clrscr();
initgraph(&gd,&gm,"c:\tc\BGI");
display(root);
getch();
closegraph();
}
create(node **root,int lev,char *dname,int lx,int rx,int x)
{
int i, gap;
if(*root==NULL)
{
(*root)=(node *)malloc(sizeof(node)); printf("Enter
name of dir/file(under %s) : ",dname);
fflush(stdin);
gets((*root)->name);
printf("enter 1 for Dir/2 for file
:"); scanf("%d",&(*root)->ftype);
(*root)->level=lev; (*root)-
>y=50+lev*50; (*root)->x=x;
(*root)->lx=lx;
(*root)->rx=rx;
for(i=0;i<5;i++)
(*root)->link[i]=NULL;
if((*root)->ftype==1)
{
printf("No of sub directories/files(for %s):",(*root)->name);
scanf("%d",&(*root)>nc); if((*root)->nc==0)
gap=rx-lx;
else
gap=(rx-lx)/(*root)->nc;
for(i=0;i<(*root)->nc;i++)
create(&((*root)>link[i]),lev+1,(*root)>name,lx+gap*i,lx+gap*i+gap,
lx+gap*i+gap/2);
}
else
(*root)->nc=0;
}
}
display(node *root)
{
int i;
settextstyle(2,0,4);
settextjustify(1,1);
setfillstyle(1,BLUE);
setcolor(14); if(root
!=NULL)
{
for(i=0;i<root->nc;i++) line(root->x,root->y,root->link[i]-
>x,root->link[i]->y); if(root->ftype==1) bar3d(root->x-
20,root->y-10,root->x+20,root>y+10,0,0); else


fillellipse(root->x,root->y,20,20);
```

```
outtextxy(root->x,root->y,root-
>name); for(i=0;i<root->nc;i++)
display(root->link[i]);
```


1.

```
        }
  }
```

INPUT

Enter Name of dir/file(under root): ROOT Enter 1 for Dir/2 for File: 1

No of subdirectories/files(for ROOT): 2 Enter Name of dir/file(under ROOT): USER1 Enter 1 for Dir/2 for File: 1

No of subdirectories/files(for USER1): 1

Enter Name of dir/file(under USER1): SUBDIR1 Enter 1 for Dir/2 for File: 1

No of subdirectories/files(for SUBDIR1): 2 Enter Name of dir/file(under USER1): JAVA Enter 1 for Dir/2 for File: 1

No of subdirectories/files(for JAVA): 0 Enter Name of dir/file(under SUBDIR1): VB Enter 1 for Dir/2 for File: 1

No of subdirectories/files(for VB): 0

Enter Name of dir/file(under ROOT): USER2 Enter 1 for Dir/2 for File: 1

No of subdirectories/files(for USER2): 2 Enter Name of dir/file(under ROOT): A Enter 1 for Dir/2 for File: 2

Enter Name of dir/file(under USER2): SUBDIR2 Enter 1 for Dir/2 for File: 1

No of subdirectories/files(for SUBDIR2): 2 Enter Name of dir/file(under SUBDIR2): PPL Enter 1 for Dir/2 for File: 1

No of subdirectories/files(for PPL): 2 Enter Name of dir/file(under PPL): B Enter 1 for Dir/2 for File: 2

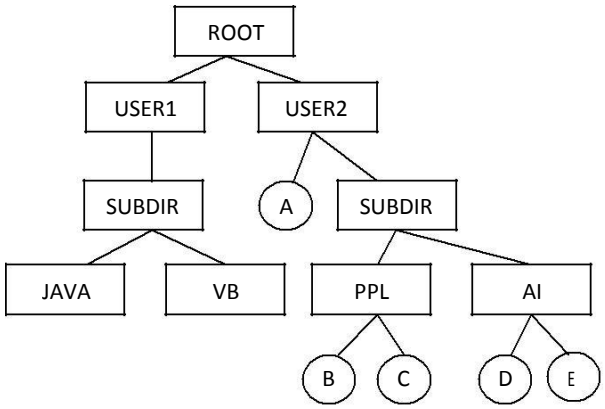
Enter Name of dir/file(under PPL): C Enter 1 for Dir/2 for File: 2

Enter Name of dir/file(under SUBDIR): AI Enter 1 for Dir/2 for File: 1

No of subdirectories/files(for AI): 2 Enter Name of dir/file(under AI): D Enter 1 for Dir/2 for File: 2

Enter Name of dir/file(under AI): E Enter 1 for Dir/2 for File: 2

OUTPUT



EXPERIMENT- 4

4. OBJECTIVE

Write a C program to simulate the MVT and MFT memory management techniques

1.

4.1 DESCRIPTION

MFT (Multiprogramming with a Fixed number of Tasks) is one of the old memory management techniques in which 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. MVT (Multiprogramming with a Variable number of Tasks) is the memory management technique in which 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.

4.2 AIM: Simulate Multiple Programming with fixed Number of Tasks (MFT**)**

4.2.1 HARDWARE REQUIREMENTS: Intel based Desktop Pc
RAM of 512 MB

4.2.2 SOFTWARE REQUIREMENTS:
Turbo C/ Borland C.

4.3 THEORY:

Multiple Programming with fixed Number of Tasks (MFT) Algorithm

Background:

IBM in their Mainframe Operating System OS/MFT implements the MFT concept. *OS/MFT* uses Fixed partitioning concept to load programs into Main memory.

Fixed Partitioning:

- In fixed partitioning concept, RAM is divided into set of fixed partition of equal Size
- Programs having the Size Less than the partition size are loaded into Memory
- Programs Having Size more then the size of Partitions Size is rejected
- The program having the size less than the partition size will lead to internal Fragmentation.
- If all partitions are allocated and a new program is to be loaded, the program that lead to Maximum Internal Fragmentation can be replaced

ALGORITHM:

Step1: start
Step2: Declare variables.
Step3: Enter total memory size.
Step4: Read the no of partitions to be divided.
Step5: Allocate memory for os.
Step6:calculate available memory by subtracting the memory of os from total memory
Step7: calculate the size of each partition by dividing available memory with no of partitions.
Step8: Read the number of processes and the size of each process.
Step9: If size of process<= size of partition then allocate memory to that process.
Step10: Display the wastage of memory.
Step11: Stop .

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
main()
{
        int ms,i,ps[20],n,size,p[20],s,intr=0;
        clrscr();
        printf("Enter size of memory:");
        scanf("%d",&ms);
        printf("Enter memory for OS:");
        scanf("%d",&s);
```

```
        ms-=s;
        printf("Enter no.of partitions to be divided:");
        scanf("%d",&n);
        size=ms/n;
        for(i=0;i<n;i++)
        {
                printf("Enter process size");
                scanf("%d ",&ps[i]);
                if(ps[i]<=size)
                {
                        intr=intr+size-ps[i];
                        printf("process%d is allocated\n",p[i]);
                }
                else
                        printf("process%d is blocked",p[i]);
        }
        printf("total fragmentation is %d",intr);
        getch();
}
```

OUTPUT:

Enter total memory size : 50

Enter memory for OS :10

Enter no.of partitions to be divided:4

Enter size of page : 10
Enter size of page : 9
Enter size of page : 9
Enter size of page : 8

Internal Fragmentation is = 4

VIVA QUESTIONS

1. The problem of fragmentation arises in _____.

   1)Static storage allocation 2) Stack allocation storage

   3 Stack allocation with dynamic binding 4 Heap allocation

2.Boundary registers _____.

   1 Are available in temporary program variable storage

   2 Are only necessary with fixed partitions

   3 Track the beginning and ending the program

   4 Track page boundaries

3.The principle of locality of reference justifies the use of _____.

   1 Virtual Memory 2 Interrupts

   3 Main memory 4 Cache memory

4. In memory management , a technique called as paging, physical memory is broken into fixed-sized blocks called _____.

   1) Pages 2) Frames 3) Blocks 4) Segments

5.Demand paged memory allocation

   1.

1 allows the virtual address space to be independent of the physical memory

2 allows the virtual address space to be a multiple of the physical memory size

3 allows deadlock tobe detected in paging schemes

4 is present only in Windows NT

EXPERIMENT :4 b)

NAME OF EXPERIMENT: multiple Programming with Varible Number of Tasks (MVT) :

AIM: Simulate multiple Programming with Varible Number of Tasks (MVT)

HARDWARE REQUIREMENTS: Intel based Desktop Pc
RAM of 512 MB

SOFTWARE REQUIREMENTS:
Turbo C/ Borland C.

THEORY:

multiple Programming with Varible Number of Tasks (MVT) Algorithm

Background:

IBM in their Mainframe Operating 'System OS/MVT implements the MVT concept. OSIMVT uses Dynamic Partition concept to load programs into Main memory.

Dynamic Partitioning:

- Initially RAM is portioned according to the of programs to be loaded into Memory till such time no other program can be loaded.
- The Left over Memory is called a hole which is too small too fit any process.
- When a new program is to be into Memory Look for the partition, Which Leads to least External fragmentation and load the Program.
- The space that is not used in a partition is called as External Fragmentation

ALGORITHM:

Step1: start
Step2: Declare variables.
Step3: Enter total memory size.
Step4: Read the no of processes
Step5: Allocate memory for os.
Step6: read the size of each process
Step7:calculate available memory by subtracting the memory of os from total memory
Step8: If available memory >= size of process then allocate memory to that process.
Step9: Display the wastage of memory.
Step10: Stop .

PROGRAM:

1.

```
#include<stdio.h>
#include<conio.h>
main()
{
        int i,m,n,tot,s[20];
        clrscr();
        printf("Enter total memory size:");
        scanf("%d",&tot);
        printf("Enter no. of processes:");
        scanf("%d",&n);
        printf("Enter memory for OS:");
        scanf("%d",&m);
        for(i=0;i<n;i++)
        {
                printf("Enter size of process %d:",i+1);
                scanf("%d",&s[i]);
        }
        tot=tot-m;
        for(i=0;i<n;i++)
        {
                if(tot>=s[i])
                {
                        printf("Allocate memory to  process %d\n",i+1);
                        tot=tot-s[i];
                }
                else
                        printf("process p%d is blocked\n",i+1);
        }
        printf("External Fragmentation is=%d",tot);
        getch();
}
```

OUTPUT:

```
Enter total memory size : 50
Enter no.of pages       : 4
Enter memory for OS     :10

Enter size of page  : 10
Enter size of page  : 9
Enter size of page  : 9
Enter size of page  : 10

External Fragmentation is = 2
```

VIVA QUESTIONS:

7. Explain about MFT?
8. Full form of MFT_____
9. Full form of MVT_____
10. differentiate MFT and MVT?
11. The _____ Memory is called a hole.

12. OSIMVT uses _____concept to load programs into Main memory.

7.OS/MFT uses _____ concept to load programs into Main memory.

EXPERIMENT- 5

5.3 OBJECTIVE

Write a C program to simulate Bankers algorithm for the purpose of deadlock avoidance.

5.4 DESCRIPTION

In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are 4held by other waiting processes. This situation is called a deadlock. Deadlock avoidance is one of the techniques for handling deadlocks. This approach requires that the operating system be given in advance additional information concerning which resources a process will request and use during its lifetime. With this additional knowledge, it can decide for each request whether or not the process should wait. To decide whether the current request can be satisfied or must be delayed, the system must consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process.

Banker's algorithm is a deadlock avoidance algorithm that is applicable to a system with multiple instances of each resource type.

NAME OF EXPERIMENT: Simulate Banker's Algorithm for Deadlock Avoidance.

AIM: Simulate Banker's Algorithm for Deadlock Avoidance to find whether the system is in safe state or not.

HARDWARE REQUIREMENTS: Intel based Desktop Pc

1.

RAM of 512 MB

SOFTWARE REQUIREMENTS: Turbo C/ Borland C.

THEORY:

DEAD LOCK AVOIDANCE

To implement deadlock avoidance & Prevention by using Banker's Algorithm.

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.

Data structures

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

Safety Algorithm

5. Work and Finish be the vector of length m and n respectively, Work=Available and Finish[i] =False.
6. Find an i such that both
   - Finish[i] =False
   - Need<=Work

   If no such I exists go to step 4.
7. work=work+Allocation, Finish[i] =True;
8. 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 Pi, If request i=[j]=k, then process Pi wants k instances of resource type Rj.

4. if Request<=Need I go to step 2. Otherwise raise an error condition.
5. if Request<=Available go to step 3. Otherwise Pi must since the resources are available.
6. Have the system pretend to have allocated the requested resources to process Pi 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 Pi is allocated its resources. However if the state is unsafe, the Pi must wait for Request i and the old resource-allocation state is restored.

ALGORITHM:

11. Start the program.
12. Get the values of resources and processes.
13. Get the avail value.
14. After allocation find the need value.
15. Check whether its possible to allocate.
16. If it is possible then the system is in safe state.
17. Else system is not in safety state.
18. If the new request comes then check that the system is in safety.
19. or not if we allow the request.
20. stop the program.

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
struct da {
int max[10],al[10],need[10],before[10],after[10];
}p[10];
void main() {
int i,j,k,l,r,n,tot[10],av[10],cn=0,cz=0,temp=0,c=0;
clrscr();
printf("\n Enter the no of processes:");
scanf("%d",&n);
printf("\n Enter the no of resources:");
scanf("%d",&r);
for(i=0;i<n;i++) {
printf("process %d \n",i+1);
for(j=0;j<r;j++) {
printf("maximum value for resource %d:",j+1);
scanf("%d",&p[i].max[j]);
}
for(j=0;j<r;j++) {
printf("allocated from resource %d:",j+1);
scanf("%d",&p[i].al[j]);
p[i].need[j]=p[i].max[j]-p[i].al[j];
}
}
for(i=0;i<r;i++) {
printf("Enter total value of resource %d:",i+1);
scanf("%d",&tot[i]);
}
for(i=0;i<r;i++) {
for(j=0;j<n;j++)
temp=temp+p[j].al[i];
av[i]=tot[i]-temp;
temp=0;
}
printf("\n\t max allocated needed total avail");
for(i=0;i<n;i++) {
printf("\n P%d \t",i+1);
for(j=0;j<r;j++)
printf("%d",p[i].max[j]);
printf("\t");
for(j=0;j<r;j++)
printf("%d",p[i].al[j]);
printf("\t");
for(j=0;j<r;j++)
printf("%d",p[i].need[j]);
printf("\t");
for(j=0;j<r;j++)
{
if(i==0)
printf("%d",tot[j]);
}
printf(" ");
for(j=0;j<r;j++) {
if(i==0)
printf("%d",av[j]);
}
}
printf("\n\n\t AVAIL BEFORE \t AVAIL AFTER");
for(l=0;l<n;l++)
{
```

```
for(i=0;i<n;i++)
{
for(j=0;j<r;j++)
{
if(p[i].need[j]>av[j])
cn++;
if(p[i].max[j]==0)
cz++;
}
if(cn==0 && cz!=r)
{
for(j=0;j<r;j++)
{

p[i].before[j]=av[j]-p[i].need[j];
p[i].after[j]=p[i].before[j]+p[i].max[j];
av[j]=p[i].after[j];
p[i].max[j]=0;
}
printf("\n p%d \t",i+1);
for(j=0;j<r;j++)
printf("%d",p[i].before[j]);
printf("\t");
for(j=0;j<r;j++)
printf("%d",p[i].after[j]);
cn=0;
cz=0;
c++;
break;
}
else {
cn=0;cz=0;
}
}
}
if(c==n)
printf("\n the above sequence is a safe sequence");
else
printf("\n deadlock occured");
getch();
}
```

OUTPUT:

```
//TEST CASE 1:

 ENTER THE NO. OF PROCESSES:4

 ENTER THE NO. OF RESOURCES:3
PROCESS 1
MAXIMUM VALUE FOR RESOURCE 1:3
MAXIMUM VALUE FOR RESOURCE 2:2
MAXIMUM VALUE FOR RESOURCE 3:2
ALLOCATED FROM RESOURCE 1:1
ALLOCATED FROM RESOURCE 2:0
ALLOCATED FROM RESOURCE 3:0
PROCESS 2
MAXIMUM VALUE FOR RESOURCE 1:6
MAXIMUM VALUE FOR RESOURCE 2:1
MAXIMUM VALUE FOR RESOURCE 3:3
ALLOCATED FROM RESOURCE 1:5
ALLOCATED FROM RESOURCE 2:1
ALLOCATED FROM RESOURCE 3:1
PROCESS 3
```

MAXIMUM VALUE FOR RESOURCE 1:3
MAXIMUM VALUE FOR RESOURCE 2:1
MAXIMUM VALUE FOR RESOURCE 3:4
ALLOCATED FROM RESOURCE 1:2
ALLOCATED FROM RESOURCE 2:1
ALLOCATED FROM RESOURCE 3:1
PROCESS 4
MAXIMUM VALUE FOR RESOURCE 1:4
MAXIMUM VALUE FOR RESOURCE 2:2
MAXIMUM VALUE FOR RESOURCE 3:2
ALLOCATED FROM RESOURCE 1:0
ALLOCATED FROM RESOURCE 2:0
ALLOCATED FROM RESOURCE 3:2
ENTER TOTAL VALUE OF RESOURCE 1:9
ENTER TOTAL VALUE OF RESOURCE 2:3
ENTER TOTAL VALUE OF RESOURCE 3:6

RESOURCES	ALLOCATED	NEEDED	TOTAL	AVAIL
P1 322	100	222	936	112
P2 613	511	102		
P3 314	211	103		
P4 422	002	420		

	AVAIL BEFORE	AVAIL AFTER
P 2	010	623
P 1	401	723
P 3	620	934
P 4	514	936

THE ABOVE SEQUENCE IS A SAFE SEQUENCE

VIVA QUESTIONS:

1. Differentiate deadlock avoidance and fragmentation

2.Tell me the real time example where this deadlock occurs?

3.How do we calculate the need for process?

4.What is the name of the algorithm to avoid deadlock?

5.Banker's algorithm for resource allocation deals with

(A)Deadlock prevention. (B)Deadlock avoidance.

(C)Deadlock recovery. (D)Mutual exclusion

EXPERIMENT- 6

12.1 OBJECTIVE

Write a C program to simulate Bankers algorithm for the purpose of deadlock prevention.

6.2 DESCRIPTION

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

Banker's algorithm is a deadlock avoidance algorithm that is applicable to a system with multiple instances of each resource type.

NAME OF EXPERIMENT: Simulate Algorithm for Deadlock prevention.

AIM: Simulate Algorithm for Deadlock prevention .

HARDWARE REQUIREMENTS: Intel based Desktop Pc
RAM of 512 MB
SOFTWARE REQUIREMENTS: Turbo C/ Borland C.

THEORY:

Deadlock Definition:
A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause (including itself).Waiting for an event could be:

- waiting for access to a critical section
- waiting for a resource Note that it is usually a non-preemptable (resource).

Conditions for Deadlock :

•Mutual exclusion: resources cannot be shared.
•Hold and wait:processes request resources incrementally, and hold on to

What they've got.
•No preemption: resources cannot be forcibly taken from processes.
•Circular wait: circular chain of waiting, in which each process is waiting for a resource held by the next process in the chain.

Strategies for dealing with Deadlock :
•ignore the problem altogether
•detection and recovery
•avoidance by careful resource allocation
•prevention by structurally negating one of the four necessary conditions.

Deadlock Prevention :
Difference from avoidance is that here, the system itself is built in such a way that there are no deadlocks. Make sure atleast one of the 4 deadlock conditions is never satisfied. This may however be even more conservative than deadlock avoidance strategy.
Algorithm:

1.Start
2.Attacking Mutex condition : never grant exclusive access. but this may not be possible for several resources.
3..Attacking preemption: not something you want to do.
4.Attacking hold and wait condition : make a process hold at the most 1 resource at a time.make all the requests at the beginning. All or nothing policy. If you feel,retry. eg. 2-phase locking
5.Attacking circular wait: Order all the resources. Make sure that the requests are issued in the correct order so that there are no cycles present in the resource graph. Resources numbered 1 ... n. Resources can be requested only in increasing order. ie. you cannot request a resource whose no is less than any you may be holding.
6.Stop

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
int max[10][10],alloc[10][10],need[10][10],avail[10],i,j,p,r,finish[10]={0},flag=0;
main( )
{
clrscr( );
printf("\n\nSIMULATION OF DEADLOCK PREVENTION");
printf("Enter no. of processes, resources");
scanf("%d%d",&p,&r);printf("Enter allocation matrix");
for(i=0;i<p;i++)
for(j=0;j<r;j++)
scanf("%d",&alloc[i][j]);
printf("enter max matrix");
for(i=0;i<p;i++) /*reading the maximum matrix and availale matrix*/
for(j=0;j<r;j++)
scanf("%d",&max[i][j]);
printf("enter available matrix");
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];
fun(); /*calling function*/
if(flag==0)
{i
f(finish[i]!=1)
{
printf("\n\n Failing :Mutual exclusion");
for(j=0;j<r;j++)
```

```
{ /*checking for mutual exclusion*/
if(avail[j]<need[i][j])
avail[j]=need[i][j];
}fun();
printf("\n By allocating required resources to process %d dead lock is prevented ",i);
printf("\n\n lack of preemption");
for(j=0;j<r;j++)
{
if(avail[j]<need[i][j])
avail[j]=need[i][j];
alloc[i][j]=0;
}
fun( );
printf("\n\n daed lock is prevented by allocating needed resources");
printf(" \n \n failing:Hold and Wait condition ");
for(j=0;j<r;j++)
{ /*checking hold and wait condition*/
if(avail[j]<need[i][j])
avail[j]=need[i][j];
}
fun( );
printf("\n AVOIDING ANY ONE OF THE CONDITION, U CAN PREVENT DEADLOCK");
}
}
getch( );
}
fun()
{
while(1)
{
for(flag=0,i=0;i<p;i++)
{
if(finish[i]==0)
{
for(j=0;j<r;j++)
{
if(need[i][j]<=avail[j])
continue;
elsebreak;
}
if(j==r)
{
for(j=0;j<r;j++)
avail[j]+=alloc[i][j];
flag=1;
finish[i]=1;
}
}
}
if(flag==0)
break;
}
}
```

Output:

```
SIMULATION OF DEADLOCK PREVENTION
Enter no. of processes, resources 3, 2
enter allocation matrix 2 4 5
                        3 4 5
Enter max matrix4 3 4
          1.
```

5 6 1
Enter available matrix2
5
Failing : Mutual Exclusion
by allocating required resources to process dead is prevented
Lack of no preemption deadlock is prevented by allocating needed resources
Failing : Hold and Wait condition

VIVA QUESTIONS:

2. The Banker's algorithm is used for _____.

2._____ is the situation in which a process is waiting on another process,which is also waiting on another process ... which is waiting on the first process. None of the processes involved in this circular wait are making progress.

3.what is safe state?

4.What are the conditions that cause deadlock?

5.How do we calculate the need for process?

EXPERIMENT- 7

7.1 OBJECTIVE

Write a C program to simulate page replacement algorithms a) FIFO b) LRU c) LFU

7.2 DESCRIPTION

Page replacement is basic to demand paging. It completes the separation between logical memory and physical memory. With this mechanism, an enormous virtual memory can be provided for programmers on a smaller physical memory. There are many different page-replacement algorithms. Every operating system probably has its own replacement scheme. A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen. If the recent past is used as an approximation of the near future, then the page that has not been used for the longest period of time can be replaced. This approach is the Least Recently Used (LRU) algorithm. LRU replacement associates with each page the time of that page's last use. When a page must be replaced, LRU chooses the page that has not been used for the longest period of time. Least frequently used (LFU) page-replacement algorithm requires that the page with the smallest count be replaced. The reason for this selection is that an actively used page should have a large reference count.

NAME OF EXPERIMENT: Simulate page replacement algorithms:

a) FIFO

AIM: Simulate FIFO page replacement algorithms.

1.

HARDWARE REQUIREMENTS: Intel based Desktop Pc

RAM of 512 MB

SOFTWARE REQUIREMENTS: Turbo C/ Borland C.

THEORY:

FIFO algorithm:

The simpler page replacement algorithm is a FIFO algorithm. A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replace, the oldest page is chosen. We can create a FIFO queue to hold all pages in memory. We replace the page at the head of the queue when a page is brought into memory; we insert it at the tail of the queue.

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

ALGORITHM:

9. Start
10. Read the number of frames
11. Read the number of pages
12. Read the page numbers
13. Initialize the values in frames to -1
14. Allocate the pages in to frames in First in first out order.
15. Display the number of page faults.
16. stop

PROGRAM

FIFO PAGE REPLACEMENT ALGORITHM

```
#include<stdio.h>
#include<conio.h>
main()
{
```

1.

```
int i, j, k, f, pf=0, count=0, rs[25], m[10],
n; clrscr();
printf("\n Enter the length of reference string -- ");
scanf("%d",&n);
printf("\n Enter the reference string --
"); for(i=0;i<n;i++)
        scanf("%d",&rs[i]);
printf("\n Enter no. of frames -- ");
scanf("%d",&f);
for(i=0;i<f;i++)
        m[i]=-1;

printf("\n The Page Replacement Process is --
\n"); for(i=0;i<n;i++)
{
        for(k=0;k<f;k++)
        {
                if(m[k]==rs[i])
                        break;
        }
        if(k==f)
        {
                m[count++]=rs[i];
                pf++;
        }
        for(j=0;j<f;j++)
                printf("\t%d",m[j]);
        if(k==f)
                printf("\tPF No.
        %d",pf); printf("\n");
        if(count==f)
                count=0;
}
printf("\n The number of Page Faults using FIFO are
%d",pf); getch();
```


1.

}

INPUT

Enter the length of reference string – 20
Enter the reference string -- 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1
Enter no. of frames -- 3

OUTPUT

The Page Replacement Process is –

7	-1	-1	PF No. 1
7	0	-1	PF No. 2
7	0	1	PF No. 3
2	0	1	PF No. 4
2	0	1	
2	3	1	PF No. 5
2	3	0	PF No. 6
4	3	0	PF No. 7
4	2	0	PF No. 8
4	2	3	PF No. 9
0	2	3	PF No. 10
0	2	3	
0	2	3	
0	1	3	PF No. 11
0	1	2	PF No. 12
0	1	2	
0	1	2	
7	1	2	PF No. 13
7	0	2	PF No. 14
7	0	1	PF No. 15

The number of Page Faults using FIFO are 15

7.3 LRU PAGE REPLACEMENT ALGORITHM

NAME OF EXPERIMENT: Simulate page replacement algorithms:
b) LRU

AIM: Simulate LRU page replacement algorithms

HARDWARE REQUIREMENTS: Intel based Desktop Pc
RAM of 512 MB
SOFTWARE REQUIREMENTS:
Turbo C/ Borland C.

ALGORITHM :

9. Start
10. Read the number of frames
11. Read the number of pages
12. Read the page numbers
13. Initialize the values in frames to -1
14. Allocate the pages in to frames by selecting the page that has not been used for the longest period of time.

1.

15. Display the number of page faults.
16. stop

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
main()
{
        int i, j , k, min, rs[25], m[10], count[10], flag[25], n, f, pf=0, next=1; clrscr();
        printf("Enter the length of reference string -- "); scanf("%d",&n);
        printf("Enter the reference string -- "); for(i=0;i<n;i++)
        {
                scanf("%d",&rs[i]);
                flag[i]=0;
        }
        printf("Enter the number of frames -- "); scanf("%d",&f);
        for(i=0;i<f;i++)
        {
                count[i]=0; m[i]=-1;
        }
        printf("\nThe Page Replacement process is -- \n"); for(i=0;i<n;i++)
        {
                for(j=0;j<f;j++)
                {
                        if(m[j]==rs[i])
                        {
                                flag[i]=1;

                        count[j]=next;
                        next++;
                }

        }
        if(flag[i]==0)
        {
                if(i<f)
                {
                        m[i]=rs[i];
                        count[i]=next;
                        next++;
                }
                Else+
                {
                        min=0;
                        for(j=1;j<f;j++)
                                if(count[min] > count[j]) min=j;

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

1.

```
                        printf("PF No. -- %d" , pf); printf("\n");
        }
        printf("\nThe number of page faults using LRU are %d",pf); getch();
}
```

INPUT

Enter the length of reference string -- 20

Enter the reference string -- 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1 Enter the number of frames -- 3

OUTPUT

The Page Replacement process is --

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

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

1.

1.

NAME OF EXPERIMENT: Simulate page replacement algorithms:

c)LFU

AIM: Simulate LFU page replacement algorithms .

HARDWARE REQUIREMENTS: Intel based Desktop Pc
RAM of 512 MB

SOFTWARE REQUIREMENTS:
Turbo C/ Borland C.

ALGORITHM:

9. Start
10. Read the number of frames
11. Read the number of pages
12. Read the page numbers
13. Initialize the values in frames to -1
14. Allocate the pages in to frames by selecting the page that will not be used for the longest period of time.
15. Display the number of page faults.
16. stop

ROGRAM

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

main()
{
        int rs[50], i, j, k, m, f, cntr[20], a[20], min, pf=0; clrscr();
        printf("\nEnter number of page references -- "); scanf("%d",&m);

        printf("\nEnter the reference string -- "); for(i=0;i<m;i++)
                scanf("%d",&rs[i]);

        printf("\nEnter the available no. of frames -- "); scanf("%d",&f);

        for(i=0;i<f;i++)
        {
                cntr[i]=0; a[i]=-1;
        }
        Printf("\nThe Page Replacement Process is – \n"); for(i=0;i<m;i++)
        {

                for(j=0;j<f;j++)
                        if(rs[i]==a[j])
                        {
                                cntr[j]++;
                                break;
                        }
                if(j==f)
                {
                        min = 0; for(k=1;k<f;k++)
                                if(cntr[k]<cntr[min])
                                        min=k;
```

```
                    a[min]=rs[i];
                    cntr[min]=1;
                    pf++;
            }
            printf("\n");
            for(j=0;j<f;j++)
                    printf("\t%d",a[j]);
            if(j==f)
                    printf("\tPF No. %d",pf);
}
printf("\n\n Total number of page faults -- %d",pf);
 getch();
}
```

INPUT

Enter number of page references -- 10
Enter the reference string -- 1 2 3 4 5 2 5 2 5 1 4 3
Enter the available no. of frames -- 3

OUTPUT

The Page Replacement Process is –

1	-1	-1	PF No. 1
1	2	-1	PF No. 2
1	2	3	PF No. 3
4	2	3	PF No. 4
5	2	3	PF No. 5
5	2	3	
5	2	3	
5	2	1	PF No. 6
5	2	4	PF No. 7
5	2	3	PF No. 8

Total number of page faults -- 8

EXPERIMENT -8

12.3 OBJECTIVE

Write a C program to simulate paging technique of memory management.

12.4 DESCRIPTION

In computer operating systems, paging is one of the memory management schemes by which a computer stores and retrieves data from the secondary storage for use in main memory. In the paging memory-management scheme, the operating system retrieves data from secondary storage in same-size blocks called pages. Paging is a memory-management scheme that permits the physical address space a process to be noncontiguous. The basic method for implementing paging involves breaking physical memory into fixed-sized blocks called frames and breaking logical memory into blocks of the same size called pages. When a process is to be executed, its pages are loaded into any available memory frames from their source.

PROGRAM

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

main()
{
        i
        n
        t

        m
        s
        ,

        p
        s
        ,

        n
        o
        p
        ,

        n
        p
        ,

        r
        e
        m
        p
        a
        g
        e
        s
        ,

        i
```

1.

```
,
j
,
x
,
y
,
p
a
,
o
f
f
s
e
t
;
i
n
t
s
[
1
0
]
,
f
n
o
[
1
0
]
[
2
0
]
;

clrscr();

p
r
i
n
t
f
(
"
\
n
E
n
t
e
```

1.

r

t
h
e

m
e
m
o
r
y

s
i
z
e

-
-

"
)
;

s
c
a
n
f
(
"
%
d
"
,
&
m
s
)
;

p
r
i
n
t
f
(
"
\
n
E
n
t
e
r

t
h
e

p
a

1.

```
g
e

s
i
z
e

-
-

"
)
;

s
c
a
n
f
(
"
%
d
"
,
&
p
s
)
;

nop = ms/ps;
printf("\nThe no. of pages available in memory are --
%d ",nop);

p
r
i
n
t
f
(
"
\
n
E
n
t
e
r

n
u
m
b
e
r

o
f

p
r
```

1.

```
o
c
e
s
s
e
s

-
-

"
)
;

s
c
a
n
f
(
"
%
d
"
,
&
n
p
)
;

rempages = nop;

for(i=1;i<=np;i++)
{

        p
        ri
        n
        t
        f
        (
        "
        \
        n
        E
        n
        t
        e
        r
        n
        o
        .
        o
        f
        p
        a
        g
        e
        s
        r
        e
```

1.

```
q
u
ir
e
d
f
o
r
p
[
%
d
]
-
-
"
,i
);
s
c
a
n
f
(
"
%
d
"
,
&
s
[i
]
);

if(s[i] >rempages)
{
        p
        r
        i
        n
        t
        f
        (
        "
        \
        n
        M
        e
        m
        o
        r
        y

        i
        s

        F
        u
        l
        l
        "
        )
        ;
```

1.

```
                b
                r
                e
                a
                k
                ;
        }
        rempages = rempages - s[i];

        p
        r
        i
        n
        t
        f
        (
        "
        \
        n
        E
        n
        t
        e
        r

        p
        a
        g
        e
        t
        a
        b
        l
        e

        f
        o
        r

        p
        [
        %
        d
        ]

        -
        -
        -

        "
        ,
        i
        )
        ;

        f
        o
        r
        (
        j
        =
        0
        ;
```

1.

```
                        j
                        <
                        s
                        [
                        i
                        ]
                        ;
                        j
                        +
                        +
                        )
                                scanf("%d",&fno[i][j]);
                }

                printf("\n
                Enter
                Logical
                Address
                to find
                Physical
                Address
                ");
                printf("\n
                Enter
                process
                no. and
                pagenum
                ber and
                offset --
                ");

                scanf("%d %d %d",&x,&y, &offset);
        if(x>np || y>=s[i] || offset>=ps)
                printf("\nInvalid Process or Page Number or offset");
        else
        {
                pa=fno[x][y]*ps+offset;
                printf("\nThe Physical Address is -- %d",pa);
        }
        getch();
}
```

INPUT

```
Enter the memory size – 1000
Enter the page size --          100
The no. of pages available in memory are -- 10
Enter number of processes --            3
Enter no. of pages required for p[1] --         4
Enter pagetable for p[1] ---            8       6       9       5

Enter no. of pages required for p[2] --         5
Enter pagetable for p[2] ---            1       4       5       7       3

Enter no. of pages required for p[3] --         5
```

OUTPUT

```
Memory is Full

Enter Logical Address to find Physical Address
Enter process no. and pagenumber and offset -- 2                3       60
```

The Physical Address is -- 760

EXPERIMENT - 9

9.1 OBJECTIVE

*Write a C program to simulate the following contiguous memory allocation techniques a) Worst-fit b) Best-fit c) First-fit

9.2 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.

1.

PROGRAM

a) WORST-FIT

```
#include<stdio.h>
#
i
n
c
l
u
d
e
<
c
o
n
i
o
.
h
>

#
d
e
f
i
n
e

m
a
x

2
5


void main()
{
        i
        n
        t

        f
        r
        a
        g
        [
        m
        a
        x
        ]
        ,
        b
        [
        m
        a
        x
        ]
        ,
        f
```

1.

```
[
m
a
x
]
,
i
,
j
,
n
b
,
n
f
,
t
e
m
p
;

s
t
a
t
i
c

i
n
t

b
f
[
m
a
x
]
,
f
f
[
m
a
x
]
;

clrscr();

pri
ntf
("\
n\t
M
em
or
y
M
an
ag
```

1.

```
em
en
t
Sc
he
me
-
Fir
st
Fit
");
pri
ntf
("\
nE
nt
er
th
e
nu
m
be
r
of
blo
cks
:");
sca
nf(
"%
d",
&n
b);
p
r
i
n
t
f
(
"
E
n
t
e
r

t
h
e

n
u
m
b
e
r

o
f

f
i
```

1.

```
les:"
);
scanf("%d",&nf);
printf("\nEnter the size of the blocks:-
```

1.

```
\
n
"
)
;

f
o
r
(
i
=
1
;
i
<
=
n
b
;
i
+
+
)
{
        p
        i
        n
        t
        f
        (
        "
        B
        l
        o
        c
        k

        %
        d
        :
        "
        ,
        i
        )
        ;

        s
        c
        a
        n
        f
        (
        "
        %
        d
        "
        ,
        &
        b
        [
        i
        ]
        )
```

1.

```
        ;
}
printf("Enter the size of the files:-\n");
for(i=1;i<=nf;i++)
```

1.

```
{
        p
        r
        i
        n
        t
        f
        (
        "
        F
        i
        l
        e

        %
        d
        :
        "
        ,
        i
        )
        ;

        s
        c
        a
        n
        f
        (
        "
        %
        d
        "
        ,
        &
        f
        [
        i
        ]
        )
        ;
}
 for(i=1;i<=nf;i++)
{
        for(j=1;j<=nb;j++)
        {
                if(bf[j]!=1)
                {
                        t
                        e
                        m
                        p
                        =
                        b
                        [
                        j
                        ]
                        -
                        f
                        [
                        i
                        ]
                        ;
```

1.

```
                                        i
                                        f
                                        (
                                        t
                                        e
                                        m
                                        p
                                        >
                                        =
                                        0
                                        )
                                        {
                                                ff[i]=j;
                                                break;
                                        }
                                }
                        }
                        frag[i]=temp;
                        bf[ff[i]]=1;
                }
                printf("\nFile_no:\tFile_size
                :\tBlock_no:\tBlock_size:\tFragement");
                for(i=1;i<=nf;i++)
printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d",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	1	5	4
2	4	3	7	3

BEST-FIT

```
#include<stdio.h>
#
i
n
c
l
u
d
e
```

1.

```
<
c
o
n
i
o
.
h
>

#
d
e
f
i
n
e

m
a
x

2
5


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++)
       {
                p
                r
                i
                n
                t
                f
                (
                "
                F
```

1.

```
i
l
e

%
d
:
"
,
i
)
;

s
c
a
n
f
(
"
%
d
"
,
&
f
[
i
]
)
;
}
 for(i=1;i<=nf;i++)
{
  for(j=1;j<=nb;j++)
{
if(bf[j]!=1)
{
t
e
m
p
=
b
[
j
]
-
f
[
i
]
;

i
f
(
t
e
m
p
>
=
```

1.

```
                                        0
                                        )
                                                if(lowest>temp)
                                                {
                                                        ff[i]=j;
                                                lowest=te
                                                mp;
                                                  }
                                        }
                                }
                                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>
#
i
n
c
l
u
d
e
<
c
o
n
i
o
.
```

1.

```
h
>

#
d
e
f
i
n
e

m
a
x

2
5


void main()
{
        int
        frag[max],b[max],f[max],i,j,nb,nf,
        temp,highest=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++)
        {
                p
                r
                i
                n
                t
                f
                (
                "
                B
                l
                o
                c
                k

                %
                d
                :
                "

                ,
                i
                )
                ;
```

1.

```
s
c
a
n
f
(
"
%
d
"
,
&
b
[
i
]
)
;
}
 printf("Enter the
 size of the files :-
 \n");
 for(i=1;i<=nf;i++)
 {
p
r
i
n
t
f
(
"
F
i
l
e

%
d
:
"
,
i
)
;

s
c
a
n
f
(
"
%
d
"
,
&
f
[
i
]
)
;
```

1.

```
                }
                for(i=1;i<=nf;i++)
                {
                for(j=1;j<=nb;j++)
                {
                        if(bf[j]!=1)  //if bf[j] is not allocated
                        {
                                te
                                mp
                                =b[
                                j]-
                                f[i];
                                if(t
                                em
                                p>
                                =0)
                                        if(highest<temp)
                                        {
                                                ff[i]=j;
                                                highest=temp;
                                        }
                        }
                }
                  frag[i]=highest;
                  bf[ff[i]]=1;
                  highest=0;
          }
          printf("\nFile_no:\tFile_size
          :\tBlock_no:\tBlock_size:\tFragement"); for(i=1;i<=nf;i++)
        printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d",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 - 10

10.1 OBJECTIVE

*Write a C program to simulate diskscheduling algorithms a) FCFS b) SCAN c) C-SCAN

10.2 DESCRIPTION

One of the responsibilities of the operating system is to use the hardware efficiently. For the disk drives, meeting this responsibility entails having fast access time and large disk bandwidth. Both the access time and the bandwidth can be improved by managing the order in which disk I/O requests are serviced which is called as disk scheduling. The simplest form of disk scheduling is, of course, the first-come, first-served (FCFS) algorithm. This algorithm is intrinsically fair, but it generally does not provide the fastest service. In the SCAN algorithm, the disk arm starts at one end, and moves towards the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk. At the other end, the direction of head movement is reversed, and servicing continues. The head continuously scans back and forth across the disk. C-SCAN is a variant of SCAN designed to provide a more uniform wait time. Like SCAN, C-SCAN moves the head from one end of the disk to the other, servicing requests along the way. When the head reaches the other end,

however, it immediately returns to the beginning of the disk without servicing any requests on the return trip

PROGRAM

FCFS DISK SCHEDULING ALGORITHM

```
#include<stdio.h>
main()
{
        i
        n
        t

        t
        [
        2
        0
        ]
        ,

        n
        ,

        I
        ,

        j
        ,

        t
        o
        h
        m
        [
        2
        0
        ]
        ,

        t
        o
        t
        =
        0
        ;

        f
        l
        o
        a
        t

        a
        v
        h
        m
        ;

        clrscr();
        p
        r
        i
```

1.

```
n
t
f
(
“
e
n
t
e
r

t
h
e

n
o
.
o
f

t
r
a
c
k
s
”
)
;

s
c
a
n
f
(
“
%
d
”
,
&
n
)
;

p
r
i
n
t
f
(
“
e
n
t
e
r

t
h
e
```

1.

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]

1.

```
-
t
[
i
]
;

i
f
(
t
o
h
m
[
i
]
<
0
)

t
o
h
m
[
i
]
=
t
o
h
m
[
i
]
*
(
-
1
)
;

}
for(i=1;i<n+1;i++)
tot+=tohm[i];
avhm=(float)tot/n;
printf
(“Tra
cks
trave
rsed\
tDiffe
rence
betw
een
track
s\n”);
for(i=
1;i<n
+1;i+
+)
```

1.

```
p
r
i
n
t
f
(
“
%
d
\
t
\
t
\
t
\
t
%
d
\
n
”
,
t
*
i
+
,
t
o
h
m
*
i
+
)
;

p
r
i
n
t
f
(
"
\
n
A
v
e
r
a
g
e

h
e
a
d
e
r
```

1.

```
m
o
v
e
m
e
n
t
s
:
%
f
"
,
a
v
h
m
)
;

g
e
t
c
h
(
)
;
}
```

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	45
58	3
60	2
70	10
18	52
90	72
150	60
160	10
184	24

1.

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
        traveresed");
        scanf("%d'",&n);
        print
        f("en
        ter
        the
        posit
        ion
        of
        head
        ");
        scanf
        ("%d
        ",&h)
        ;
        t
        [
        0
        ]
        =
        0
        ;
        t
        [
        1
        ]
        =
        h
        ;

        p
        r
        i
        n
        t
        f
        (
        "
        e
        n
        t
        e
        r

        t
        h
        e
```

1.

```
t
r
a
c
k
s
"
)
;

f
o
r
(
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)
{
        a
        t
        r
        [
        p
        ]
        =
        t
        [
        j
        ]
        ;

        j
        -
        -
        ;
```

1.

```
            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++)
{
            i
                        f
                        (
                        a
                        t
                        r
                        [
                        j
                        ]
                        >
                        a
                        t
                        r
                        [
                        j
                        +
                        1
                        ]
                        )

                        d
                        [
                        j
                        ]
                        =
                        a
                        t
                        r
                        [
                        j
                        ]
                        -
                        a
                        t
                        r
                        [
                        j
                        +
                        1
                        ]
                        ;
            else
                        d
            [
            j
            ]
            =
            a
            t
            r
            [
            j
            +
            1
            ]
            -
```

1.

```
                a
                t
                r
                [
                j
                ]
                ;

                s
                u
                m
                +
                =
                d
                [
                j
                ]
                ;
        }
        printf("\nAverage header
        movements:%f",(float)su
        m/n); getch();
}
```

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
16	
0	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
```

```
traveresed"); 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])
                {
                        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;break;
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)su
```

1.

```
        m/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

EXPERIMENT 11

11.4 OBJECTIVE

*Write a C program to simulate page replacement algorithms a) Optimal

11.5 DESCRIPTION

Optimal page replacement algorithm has the lowest page-fault rate of all algorithms and will never suffer from Belady's anomaly. The basic idea is to replace the page that will not be used for the longest period of time. Use of this page-replacement algorithm guarantees the lowest possible page fault rate for a fixed number of frames. Unfortunately, the optimal page-replacement algorithm is difficult to implement,

because it requires future knowledge of the reference string.

11.6 PROGRAM

11.7

```
include<stdio.h> int n;
main()
{
int
seq[30],fr[5],
pos[5],find,fla
g,max,i,j,m,k,
t,s;          int
count=1,pf=0,
p=0; float pfr;
clrscr();
printf("Enter
maximum limit
of              the
sequence:    ");
scanf("%d"ax);
printf("\nEnter
the   sequence:
");
for(i=0;i<m++)
scanf("
%d",&s
eq[i]);
printf("
\nEnter
no. of
frames:
");
scanf(%
d
fr[0]=seq[0];
pf++;
printf("%d\t",fr[0]);
i=1;
while(count<n)
{
flag=1;
p++;
for(j=0;j<i;j++)
{
if(seq[i]==seq[j]
flag=0;
}if(flag!=0)
{
fr[count]=seq[i];
printf("%d\t",fr[count]);
count++
pf++;

i++;
}
printf("\n");
for(i=p;i<max;i++)
{
flag=1;
for(j=0;j<n;j++)
```

```
{
if(seq[i]==fr[j])
                                        flag=0;
                                }
                                if(flag!=0)

                        {
                                for(j=0;j<n;j++)
                                {
                                        m=fr[j];
                                        for(k=i;k<max;k++)
                                        {
                                                if(seq[k]==m)
                                                {
                                                        pos[j]=k;
                                                        break;
                                                }
                                                else
                                                        pos[j]=1;
                                        }
                                }
                                for(k=0;k<n;k++)
                                {
                                        if(pos[k]==1)
                                                flag=0;
                                }
                                if(flag!=0)
                                        s=findmax(pos);
                                if(flag==0)
                                {
                                        for(k=0;k<n;k++)
                                        {
                                                if(pos[k]==1)
                                                {
                                                        s=k;
                                                        break;
                                                }
                                        }
                                }
                                fr[s]=seq[i];
                                for(k=0;k<n;k++)
                                        printf("%d\t",fr[k]);
                                pf++;
                                printf("\n");
                        }
                }
                pfr=(float)pf/(float)max;
                printf("\nThe no. of page faults are
                %d",pf); printf("\nPage fault rate %f",pfr);
                getch();
}
int findmax(int a[])
{
        int max,i,k=0;
        max=a[0];
        for(i=0;i<n;i++)
        {
                if(max<a[i])
```

1.

```
                {
                        max=a[i];
                        k=i;
                }
        }
        return k;
}
```

INPUT

Enter number of page references -- 10

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

Enter the available no. of frames -- 3

OUTPUT

The Page Replacement Process is –

1	-1	-1	PF No. 1
1	2	-1	PF No. 2
1	2	3	PF No. 3
4	2	3	PF No. 4
5	2	3	PF No. 5
5	2	3	
5	2	3	
5	2	1	PF No. 6
5	2	4	PF No. 7
5	2	3	PF No. 8

Total number of page faults -- 8

EXPERIMENT 12

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

DESCRIPTION

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

PROGRAM

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

```
                                printf("1.One can eat at a time\t2.Two can eat at a time\t3.Exit\nEnter your
                                choice:"); scanf("%d", &cho);
                                switch(cho)
                                {
                                        case 1:  one();
                                                 break;
                                        case 2:  two();
                                                 break;
                                        case 3:  exit(0);
                                        default: printf("\nInvalid option..");
                                }

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

INPUT

DINING PHILOSOPHER PROBLEM
Enter the total no. of philosophers: 5
How many are hungry : 3
Enter philosopher 1 position: 2
Enter philosopher 2 position: 4
Enter philosopher 3 position: 5

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 1:

1.1 PRE-LAB QUESTIONS

1. Define operating system?
2. What are the different types of operating systems?
3. Define a process?
4. What is CPU Scheduling?
5. Define arrival time, burst time, waiting time, turnaround time?

1.2 POST-LAB QUESTIONS

1. What is the advantage of round robin CPU scheduling algorithm?
2. Which CPU scheduling algorithm is for real-time operating system?
3. In general, which CPU scheduling algorithm works with highest waiting time?
4. Is it possible to use optimal CPU scheduling algorithm in practice?
5. What is the real difficulty with the SJF CPU scheduling algorithm?

1.3 ASSIGNMENT QUESTIONS

1. Write a C program to implement round robin CPU scheduling algorithm for the following given scenario. All the processes in the system are divided into two categories – system processes and user processes. System processes are to be given higher priority than user processes. Consider the time quantum size for the system processes and user processes to be 5 msec and 2 msec respectively.
2. Write a C program to simulate pre-emptive SJF CPU scheduling algorithm.

EXPERIMENT 2: