OPERATING SYSTEMS

LAB MANUAL

COMPUTER SCIENCE AND ENGINEERING

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 Specific Outcomes	
PSO1	**Professional Skills:** The ability to research, understand and implement computer programs in the areas related to algorithms, system software, multimedia, web design, big data analytics, and networking for efficient analysis and design of computer-based systems of varying complexity.
PSO2	**Problem-Solving Skills:** The ability to apply standard practices and strategies in software project development using open-ended programming environments to deliver a quality product for business success.
PSO3	**Successful Career and Entrepreneurship:** The ability to employ modern computer languages, environments, and platforms in creating innovative career paths, to be an entrepreneur, and a zest for higher studies.

OPERATING SYSTEMS LAB SYLLABUS

(Practical Hours: 03, Credits: 02)

Implement the following programs on Linux platform using C language.

Exp. No.	Division of Experiments	List of Experiments	Page No.
1	**CPU Scheduling Algorithms**	Write a C program to simulate the following non-preemptive CPU scheduling algorithms to find turnaround time and waiting time. a) FCFS b) SJF c) Round Robin (pre-emptive) d) Priority	2
2		*****Write a C program to simulate multi-level queue scheduling algorithm considering the following 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. Use FCFS scheduling for the processes in each queue.	7
3	**File Allocation Strategies**	Write a C program to simulate the following file allocation strategies. a) Sequential b) Indexed c) Linked	9
4	**Memory Management Techniques**	Write a C program to simulate the MVT and MFT memory management techniques.	13
5		*****Write a C program to simulate the following contiguous memory allocation techniques a) Worst-fit b) Best-fit c) First-fit	16
6		Write a C program to simulate paging technique of memory management.	20
7	**File Organization Techniques**	Write a C program to simulate the following file organization techniques a) Single level directory b) Two level directory c) Hierarchical	22
8	**Deadlock Management Techniques**	Write a C program to simulate Bankers algorithm for the purpose of deadlock avoidance.	29
9		*Write a C program to simulate disk scheduling algorithms a) FCFS b) SCAN c) C-SCAN	32
10	**Page Replacement Algorithms**	Write a C program to simulate page replacement algorithms a) FIFO b) LRU c) LFU	36
11		*Write a C program to simulate page replacement algorithms a) Optimal	41
12	**Process Synchroniza-tion**	*****Write a C program to simulate producer-consumer problem using semaphores.	44
13		*Write a C program to simulate the concept of Dining-Philosophers problem.	45
14	**Lab Questions & Assignments**		47

*****Content beyond the university prescribed syllabi

ATTAINMENT OF PROGRAM OUTCOMES & PROGRAM SPECIFIC OUTCOMES

Exp. No.	Experiment	Program Outcomes Attained	Program Specific Outcomes Attained
1	Write a C program to simulate the following non-preemptive CPU scheduling algorithms to find turnaround time and waiting time. a) FCFS b) SJF c) Round Robin d) Priority	PO1, PO2, PO4	PSO1
2	*****Write a C program to simulate multi-level queue scheduling algorithm considering the following 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. Use FCFS scheduling for the processes in each queue.	PO1, PO2, PO3, PO4, PO12	PSO1, PSO2
3	Write a C program to simulate the following file allocation strategies. a) Sequential b) Indexed c) Linked	PO1, PO2, PO4	PSO1
4	Write a C program to simulate the MVT and MFT memory management techniques.	PO1, PO2, PO4	PSO1
5	*****Write a C program to simulate the following contiguous memory allocation techniques a) Worst-fit b) Best-fit c) First-fit	PO1, PO2, PO4, PO12	PSO1
6	Write a C program to simulate paging technique of memory management.	PO1, PO2, PO4	PSO1
7	Write a C program to simulate the following file organization techniques a) Single level directory b) Two level directory c) Hierarchical	PO1, PO2, PO4	PSO1
8	Write a C program to simulate Bankers algorithm for the purpose of deadlock avoidance.	PO1, PO2, PO4	PSO1
9	*Write a C program to simulate disk scheduling algorithms a) FCFS b) SCAN c) C-SCAN	PO1, PO2, PO4, PO12	PSO1
10	Write a C program to simulate page replacement algorithms a) FIFO b) LRU c) LFU	PO1, PO2	PSO1
11	*Write a C program to simulate page replacement algorithms a) Optimal	PO1, PO2, PO12	PSO1
12	*****Write a C program to simulate producer-consumer problem using semaphores.	PO1, PO2, PO3, PO4, PO5, PO12	PSO1, PSO2
13	*Write a C program to simulate the concept of Dining-Philosophers problem.	PO1, PO2, PO3, PO4, PO5, PO12	PSO1, PSO2

*****Content beyond the university prescribed syllabi

OPERATING SYSTEMS LABORATORY

OBJECTIVE:

This lab complements the operating systems course. Students will gain practical experience with designing and implementing concepts of operating systems such as system calls, CPU scheduling, process management, memory management, file systems and deadlock handling using C language in Linux environment.

OUTCOMES:

Upon the completion of Operating Systems practical course, the student will be able to:

1. **Understand** and implement basic services and functionalities of the operating system using system calls.
2. **Use** modern operating system calls and synchronization libraries in software/ hardware interfaces**.**
3. **Understand** the benefits of thread over process and implement synchronized programs using multithreading concepts.
4. **Analyze** and simulate CPU Scheduling Algorithms like FCFS, Round Robin, SJF, and Priority.
5. **Implement** memory management schemes and page replacement schemes.
6. **Simulate** file allocation and organization techniques.
7. **Understand** the concepts of deadlock in operating systems and implement them in multiprogramming system.

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.

1.3 PROGRAM

1.3.1 FCFS CPU SCHEDULING ALGORITHM

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

```
        for(i=0;i<n;i++)
                printf("\n\t P%d \t\t %d \t\t %d \t\t %d", i, bt[i], wt[i], tat[i]);
        printf("\nAverage Waiting Time -- %f", wtavg/n);
        printf("\nAverage Turnaround Time -- %f", tatavg/n);
        getch();
}
```

INPUT

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

OUTPUT

PROCESS	BURST TIME	WAITING TIME	TURNAROUND TIME
P0	24	0	24
P1	3	24	27
P2	3	27	30

```
Average Waiting Time--     17.000000
Average Turnaround Time --          27.000000
```

1.3.2 SJF CPU SCHEDULING ALGORITHM

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

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

                                temp=p[i];
                                p[i]=p[k];
                                p[k]=temp;
                         }
        wt[0] =   wtavg = 0;
        tat[0] = tatavg = bt[0];
        for(i=1;i<n;i++)
        {
                wt[i] = wt[i-1] +bt[i-1];
                tat[i] = tat[i-1] +bt[i];
                wtavg = wtavg + wt[i];
                tatavg = tatavg + tat[i];
        }
        printf("\n\t PROCESS \tBURST TIME \t WAITING TIME\t TURNAROUND TIME\n");
        for(i=0;i<n;i++)
```

```
                printf("\n\t P%d \t\t %d \t\t %d \t\t %d", p[i], bt[i], wt[i], tat[i]);
        printf("\nAverage Waiting Time -- %f", wtavg/n);
        printf("\nAverage Turnaround Time -- %f", tatavg/n);
        getch();
}
```

INPUT

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

OUTPUT

PROCESS	BURST TIME	WAITING TIME	TURNAROUND TIME
P3	3	0	3
P0	6	3	9
P2	7	9	16
P1	8	16	24

Average Waiting Time -- 7.000000
Average Turnaround Time -- 13.000000

1.3.3 ROUND ROBIN CPU SCHEDULING ALGORITHM

```
#include<stdio.h>
main()
{
        int i,j,n,bu[10],wa[10],tat[10],t,ct[10],max;
        float awt=0,att=0,temp=0;
        clrscr();
        printf("Enter the no of processes -- ");
        scanf("%d",&n);

        for(i=0;i<n;i++)
        {
                printf("\nEnter Burst Time for process %d -- ", i+1);
                scanf("%d",&bu[i]);
                ct[i]=bu[i];
        }
        printf("\nEnter the size of time slice -- ");
        scanf("%d",&t);
        max=bu[0];
        for(i=1;i<n;i++)
                if(max<bu[i])
                        max=bu[i];
        for(j=0;j<(max/t)+1;j++)
                for(i=0;i<n;i++)
                        if(bu[i]!=0)
                                if(bu[i]<=t)
                                {
                                        tat[i]=temp+bu[i];
                                        temp=temp+bu[i];
                                        bu[i]=0;
                                }
                                else
                                {
                                        bu[i]=bu[i]-t;
                                        temp=temp+t;
                                }
         for(i=0;i<n;i++)
        {
                wa[i]=tat[i]-ct[i];
                att+=tat[i];
```

```
                awt+=wa[i];
        }
        printf("\nThe Average Turnaround time is -- %f",att/n);
        printf("\nThe Average Waiting time is -- %f  ",awt/n);
        printf("\n\tPROCESS\t BURST TIME \t WAITING TIME\tTURNAROUND TIME\n");
        for(i=0;i<n;i++)
                printf("\t%d \t %d \t\t %d \t\t %d \n",i+1,ct[i],wa[i],tat[i]);
        getch();
}
```

INPUT

Enter the no of processes – 3
Enter Burst Time for process 1 – 24
Enter Burst Time for process 2 -- 3
Enter Burst Time for process 3 -- 3

Enter the size of time slice – 3

OUTPUT

The Average Turnaround time is – 15.666667
The Average Waiting time is -- 5.666667

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

1.3.4 PRIORITY CPU SCHEDULING ALGORITHM

```
#include<stdio.h>
main()
{
        int p[20],bt[20],pri[20], wt[20],tat[20],i, k, n, temp;
        float wtavg, tatavg;
        clrscr();
        printf("Enter the number of processes  ---  ");
        scanf("%d",&n);

        for(i=0;i<n;i++)
        {
                p[i] = i;
                printf("Enter the Burst Time & Priority of Process %d ---  ",i);
                scanf("%d %d",&bt[i], &pri[i]);
        }
        for(i=0;i<n;i++)
                for(k=i+1;k<n;k++)
                        if(pri[i] > pri[k])
                        {
                                temp=p[i];
                                p[i]=p[k];
                                p[k]=temp;

                                temp=bt[i];
                                bt[i]=bt[k];
                                bt[k]=temp;

                                temp=pri[i];
                                pri[i]=pri[k];
                                pri[k]=temp;

                        }
        wtavg = wt[0] = 0;
        tatavg = tat[0] = bt[0];
```

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

                wtavg = wtavg + wt[i];
                tatavg = tatavg + tat[i];
        }

        printf("\nPROCESS\t\tPRIORITY\tBURST TIME\tWAITING TIME\tTURNAROUND  TIME");
        for(i=0;i<n;i++)
                printf("\n%d \t\t %d \t\t  %d \t\t %d \t\t %d ",p[i],pri[i],bt[i],wt[i],tat[i]);

        printf("\nAverage Waiting Time is  ---  %f",wtavg/n);
        printf("\nAverage Turnaround Time is --- %f",tatavg/n);
        getch();
}
```

INPUT

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

OUTPUT

PROCESS	PRIORITY	BURST TIME	WAITING TIME	TURNAROUND TIME
1	1	1	0	1
4	2	5	1	6
0	3	10	6	16
2	4	2	16	18
3	5	1	18	19

Average Waiting Time is --- 8.200000
Average Turnaround Time is --- 12.000000

EXPERIMENT 2

2.1 OBJECTIVE

*Write a C program to simulate multi-level queue scheduling algorithm considering the following 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. The priority of each process ranges from 1 to 3. Use fixed priority scheduling for all the processes.

2.2 DESCRIPTION

Multi-level queue scheduling algorithm is used in scenarios where the processes can be classified into groups based on property like process type, CPU time, IO access, memory size, etc. In a multi-level queue scheduling algorithm, there will be 'n' number of queues, where 'n' is the number of groups the processes are classified into. Each queue will be assigned a priority and will have its own scheduling algorithm like round-robin scheduling or FCFS. For the process in a queue to execute, all the queues of priority higher than it should be empty, meaning the process in those high priority queues should have completed its execution. In this scheduling algorithm, once assigned to a queue, the process will not move to any other queues.

2.3 PROGRAM

```
main()
{
int p[20],bt[20], su[20], wt[20],tat[20],i, k, n, temp;
float wtavg, tatavg;
clrscr();
printf("Enter the number of processes  ---  ");
scanf("%d",&n);

for(i=0;i<n;i++)
{
        p[i] = i;
        printf("Enter the Burst Time of Process %d ---  ", i);
        scanf("%d",&bt[i]);
        printf("System/User Process (0/1) ? --- ");
        scanf("%d", &su[i]);

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

                        temp=bt[i];
                        bt[i]=bt[k];
                        bt[k]=temp;

                        temp=su[i];
                        su[i]=su[k];
                        su[k]=temp;

                 }
wtavg = wt[0] = 0;
tatavg = tat[0] = bt[0];

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

```
                wtavg = wtavg + wt[i];
                tatavg = tatavg + tat[i];
        }

        printf("\nPROCESS\t\t SYSTEM/USER PROCESS \tBURST TIME\tWAITING TIME\tTURNAROUND     TIME");
        for(i=0;i<n;i++)
                printf("\n%d \t\t %d \t\t  %d \t\t %d \t\t %d ",p[i],su[i],bt[i],wt[i],tat[i]);

        printf("\nAverage Waiting Time is  ---  %f",wtavg/n);
        printf("\nAverage Turnaround Time is --- %f",tatavg/n);
        getch();
}
```

INPUT

```
Enter the number of processes  ---  4
Enter the Burst Time of Process 0 ---  3
System/User Process (0/1) ? --- 1
Enter the Burst Time of Process 1 ---  2
System/User Process (0/1) ? --- 0
Enter the Burst Time of Process 2 ---  5
System/User Process (0/1) ? --- 1
Enter the Burst Time of Process 3 ---  1
System/User Process (0/1) ? --- 0
```

OUTPUT

PROCESS	SYSTEM/USER PROCESS	BURST TIME	WAITING TIME	TURNAROUND TIME
1	0	2	0	2
3	0	1	2	3
2	1	5	3	8
0	1	3	8	11

```
Average Waiting Time is  ---  3.250000
Average Turnaround Time is --- 6.000000
```

EXPERIMENT 3

3.1 OBJECTIVE

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

3.2 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.

3.2.1 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.

3.2.2 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.

3.2.3 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.

3.3 PROGRAM

3.3.1 SEQUENTIAL FILE ALLOCATION

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

struct fileTable
{
        char name[20];
        int sb, nob;
}ft[30];

void main()
{
        int i, j, n;
        char s[20];
        clrscr();
        printf("Enter no of files      :");
        scanf("%d",&n);

        for(i=0;i<n;i++)
        {
                printf("\nEnter file name %d          :",i+1);
                scanf("%s",ft[i].name);
                printf("Enter starting block of file %d          :",i+1);
                scanf("%d",&ft[i].sb);
                printf("Enter no of blocks in file %d :",i+1);
                scanf("%d",&ft[i].nob);
        }
        printf("\nEnter the file name to be searched-- ");
        scanf("%s",s);
        for(i=0;i<n;i++)
                if(strcmp(s, ft[i].name)==0)
```

```
                        break;
        if(i==n)
                printf("\nFile Not Found");
        else
        {
                printf("\nFILE NAME   START BLOCK   NO OF BLOCKS   BLOCKS OCCUPIED\n");
                printf("\n%s\t\t%d\t\t%d\t",ft[i].name,ft[i].sb,ft[i].nob);
                for(j=0;j<ft[i].nob;j++)
                        printf("%d, ",ft[i].sb+j);
        }
        getch();
}
```

INPUT:

```
Enter no of files   :3

Enter file name 1    :A
Enter starting block of file 1  :85
Enter no of blocks in file 1    :6

Enter file name 2    :B
Enter starting block of file 2  :102
Enter no of blocks in file 2    :4

Enter file name 3    :C
Enter starting block of file 3  :60
Enter no of blocks in file 3    :4
Enter the file name to be searched  -- B
```

OUTPUT:

FILE NAME	START BLOCK	NO OF BLOCKS	BLOCKS OCCUPIED
B	102	4	102, 103, 104, 105

3.3.2 LINKED FILE ALLOCATION

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

struct fileTable
{
char name[20];
int nob;
struct block *sb;
}ft[30];

struct block
{
        int bno;
        struct block *next;
};

void main()
{
        int i, j, n;
        char s[20];
        struct block *temp;
        clrscr();
        printf("Enter no of files      :");
        scanf("%d",&n);
        for(i=0;i<n;i++)
        {
                printf("\nEnter file name %d          :",i+1);
                scanf("%s",ft[i].name);
```

```
                printf("Enter no of blocks in file %d :",i+1);
                scanf("%d",&ft[i].nob);
                ft[i].sb=(struct block*)malloc(sizeof(struct block));
                temp = ft[i].sb;
                printf("Enter the blocks of the file    :");
                scanf("%d",&temp->bno);
                temp->next=NULL;

                for(j=1;j<ft[i].nob;j++)
                {
                        temp->next = (struct block*)malloc(sizeof(struct block));
                        temp = temp->next;
                        scanf("%d",&temp->bno);
                }
                temp->next = NULL;
        }
        printf("\nEnter the file name to be searched  --   ");
        scanf("%s",s);
        for(i=0;i<n;i++)
                if(strcmp(s, ft[i].name)==0)
                        break;
        if(i==n)
                printf("\nFile Not Found");
        else
        {
                printf("\nFILE NAME  NO OF BLOCKS   BLOCKS OCCUPIED");
                printf("\n    %s\t\t%d\t",ft[i].name,ft[i].nob);
                temp=ft[i].sb;
                for(j=0;j<ft[i].nob;j++)
                {
                        printf("%d → ",temp->bno);
                        temp = temp->next;
                }
        }
        getch();
}
```

INPUT:

```
Enter no of files      : 2

Enter file 1    : A
Enter no of blocks in file 1    : 4
Enter the blocks of the file 1        :  12 23 9 4

Enter file 2    : G
Enter no of blocks in file 2    : 5
Enter the blocks of the file 2        : 88 77 66 55 44

Enter the file to be searched   : G
```

OUTPUT:

```
FILE NAME       NO OF BLOCKS                BLOCKS OCCUPIED
G                       5                   88 → 77→  66→  55→ 44
```

3.3.3 INDEXED FILE ALLOCATION

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

struct fileTable
{
char name[20];
int nob, blocks[30];
```

```
}ft[30];

void main()
{
        int i, j, n;
        char s[20];
        clrscr();
        printf("Enter no of files     :");
        scanf("%d",&n);
        for(i=0;i<n;i++)
        {
                printf("\nEnter file name %d          :",i+1);
                scanf("%s",ft[i].name);
                printf("Enter no of blocks in file %d :",i+1);
                scanf("%d",&ft[i].nob);
                printf("Enter the blocks of the file    :");
                for(j=0;j<ft[i].nob;j++)
                        scanf("%d",&ft[i].blocks[j]);
        }

        printf("\nEnter the file name to be searched-- ");
        scanf("%s",s);
        for(i=0;i<n;i++)
                if(strcmp(s, ft[i].name)==0)
                        break;
        if(i==n)
                printf("\nFile Not Found");
        else
        {
                printf("\nFILE NAME  NO OF BLOCKS   BLOCKS OCCUPIED");
                printf("\n    %s\t\t%d\t",ft[i].name,ft[i].nob);
                for(j=0;j<ft[i].nob;j++)
                        printf("%d, ",ft[i].blocks[j]);
        }
        getch();
}
```

INPUT:

```
Enter no of files     : 2

Enter file 1    : A
Enter no of blocks in file 1    : 4
Enter the blocks of the file 1        :  12 23 9 4

Enter file 2    : G
Enter no of blocks in file 2    : 5
Enter the blocks of the file 2        : 88 77 66 55 44
Enter the file to be searched   : G
```

OUTPUT:

```
FILE NAME         NO OF BLOCKS              BLOCKS OCCUPIED
G                         5                       88,   77,   66,   55,   44
```

EXPERIMENT 4

4.1 OBJECTIVE

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

4.2 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.

4.3 PROGRAM

4.3.1 MFT MEMORY MANAGEMENT TECHNIQUE

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

main()
{
 int   ms, bs, nob, ef,n, mp[10],tif=0;
 int i,p=0;

 clrscr();
 printf("Enter the total memory available (in Bytes) -- ");
 scanf("%d",&ms);
 printf("Enter the block size (in Bytes) -- ");
 scanf("%d", &bs);
 nob=ms/bs;
 ef=ms - nob*bs;
 printf("\nEnter the number of processes -- ");
 scanf("%d",&n);
 for(i=0;i<n;i++)
 {
          printf("Enter memory required for process %d (in Bytes)-- ",i+1);
          scanf("%d",&mp[i]);
 }

 printf("\nNo. of Blocks available in memory -- %d",nob);
 printf("\n\nPROCESS\tMEMORY REQUIRED\t  ALLOCATED\tINTERNAL  FRAGMENTATION");
 for(i=0;i<n && p<nob;i++)
 {
          printf("\n   %d\t\t%d",i+1,mp[i]);
          if(mp[i] > bs)
                   printf("\t\tNO\t\t---");
          else
          {
                   printf("\t\tYES\t%d",bs-mp[i]);
                   tif = tif + bs-mp[i];
                   p++;
          }
 }
 if(i<n)
          printf("\nMemory is Full, Remaining Processes cannot be accomodated");

 printf("\n\nTotal Internal Fragmentation is %d",tif);
 printf("\nTotal External Fragmentation is %d",ef);
 getch();
```

}

INPUT

Enter the total memory available (in Bytes) -- 1000
Enter the block size (in Bytes)-- 300
Enter the number of processes – 5
Enter memory required for process 1 (in Bytes) -- 275
Enter memory required for process 2 (in Bytes) -- 400
Enter memory required for process 3 (in Bytes) -- 290
Enter memory required for process 4 (in Bytes) -- 293
Enter memory required for process 5 (in Bytes) -- 100

No. of Blocks available in memory -- 3

OUTPUT

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

Memory is Full, Remaining Processes cannot be accommodated
Total Internal Fragmentation is 42
Total External Fragmentation is 100

4.3.2 MVT MEMORY MANAGEMENT TECHNIQUE

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

main()
{
 int   ms,mp[10],i, temp,n=0;
 char ch = 'y';

 clrscr();
 printf("\nEnter the total memory available (in Bytes)-- ");
 scanf("%d",&ms);
 temp=ms;
 for(i=0;ch=='y';i++,n++)
 {
          printf("\nEnter memory required for process %d (in Bytes) -- ",i+1);
          scanf("%d",&mp[i]);
          if(mp[i]<=temp)
          {
                   printf("\nMemory is allocated for Process %d ",i+1);
                   temp = temp - mp[i];
          }
          else
          {
                   printf("\nMemory is Full");
                   break;
          }
          printf("\nDo you want to continue(y/n) -- ");
          scanf(" %c", &ch);
 }
 printf("\n\nTotal Memory Available -- %d", ms);

 printf("\n\n\tPROCESS\t\t MEMORY ALLOCATED  ");
 for(i=0;i<n;i++)
          printf("\n \t%d\t\t%d",i+1,mp[i]);
 printf("\n\nTotal Memory Allocated is %d",ms-temp);
 printf("\nTotal External Fragmentation is %d",temp);
```

```
 getch();
}
```

INPUT

```
Enter the total memory available (in Bytes) --          1000

Enter memory required for process 1 (in Bytes) --       400
Memory is allocated for Process 1
Do you want to continue(y/n) --      y

Enter memory required for process 2 (in Bytes) --       275
Memory is allocated for Process 2
Do you want to continue(y/n) --      y

Enter memory required for process 3 (in Bytes) --       550
```

OUTPUT

```
Memory is Full
Total Memory Available -- 1000

PROCESS          MEMORY ALLOCATED
1                        400
2                        275

Total Memory Allocated is 675
Total External Fragmentation is      325
```

EXPERIMENT 5

5.1 OBJECTIVE

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

5.2 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.

5.3 PROGRAM

5.3.1 WORST-FIT

```
#include<stdio.h>
#include<conio.h>
#define max 25

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]);
        }
        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

5.3.2 BEST-FIT

```
#include<stdio.h>
#include<conio.h>
#define max 25

void main()
{
        int frag[max],b[max],f[max],i,j,nb,nf,temp,lowest=10000;
        static int bf[max],ff[max];
        clrscr();

        printf("\nEnter the number of blocks:");
        scanf("%d",&nb);
        printf("Enter the number of files:");
        scanf("%d",&nf);
        printf("\nEnter the size of the blocks:-\n");
        for(i=1;i<=nb;i++)
                printf("Block %d:",i);scanf("%d",&b[i]);

        printf("Enter the size of the files :-\n");
        for(i=1;i<=nf;i++)
        {
                printf("File %d:",i);
                scanf("%d",&f[i]);
        }
        for(i=1;i<=nf;i++)
        {
                for(j=1;j<=nb;j++)
                {
                        if(bf[j]!=1)
                        {
                                temp=b[j]-f[i];
                                if(temp>=0)
                                        if(lowest>temp)
                                        {
                                                ff[i]=j;
```

```
                                                lowest=temp;
                                        }
                                }
                        }
                        frag[i]=lowest;
                        bf[ff[i]]=1;
                        lowest=10000;
                }
                printf("\nFile No\tFile Size \tBlock No\tBlock Size\tFragment");
                for(i=1;i<=nf && ff[i]!=0;i++)
                        printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d",i,f[i],ff[i],b[ff[i]],frag[i]);
                getch();
}
```

INPUT

Enter the number of blocks: 3
Enter the number of files: 2

Enter the size of the blocks:-
Block 1: 5
Block 2: 2
Block 3: 7

Enter the size of the files:-
File 1: 1
File 2: 4

OUTPUT

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

5.3.3 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 6

6.1 OBJECTIVE

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

6.2 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.

6.3 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 7

7.1 OBJECTIVE

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

7.2 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.

7.3 PROGRAM

7.3.1 SINGLE LEVEL DIRECTORY ORGANIZATION

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

void main()
{
        int i,ch;
        char f[30];
        clrscr();
        dir.fcnt = 0;
        printf("\nEnter name of directory --    ");
        scanf("%s", dir.dname);
        while(1)
        {
                printf("\n\n1. Create File\t2. Delete File\t3. Search File \n
                                        4. Display Files\t5. Exit\nEnter your choice -- ");
                scanf("%d",&ch);
                switch(ch)
                {
                        case 1:  printf("\nEnter the name of the file --   ");
                                 scanf("%s",dir.fname[dir.fcnt]);
                                 dir.fcnt++;
                                 break;
                        case 2:  printf("\nEnter the name of the file -- ");
                                 scanf("%s",f);
                                 for(i=0;i<dir.fcnt;i++)
                                 {
                                        if(strcmp(f, dir.fname[i])==0)
                                        {
                                                printf("File %s is deleted ",f);
                                                strcpy(dir.fname[i],dir.fname[dir.fcnt-1]);
                                                break;
                                        }
                                 }
                                 if(i==dir.fcnt)
                                        printf("File %s not found",f);
```

```
                                        else
                                                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

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
```

7.3.2 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;
                        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
4. Search File          5. Display         6. Exit        Enter your choice --      1

Enter name of directory  --    DIR1
Directory created
```

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

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

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

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

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

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

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

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

Directory       Files
DIR1            A1      A2
DIR2            B1

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

Enter name of the directory – DIR
Directory not found

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

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

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

7.3.3 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()
{
```

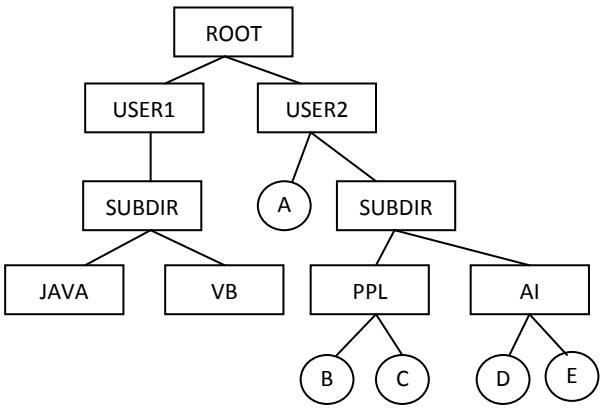
```
        int gd=DETECT,gm;
        node *root;
        root=NULL;
        clrscr();
        create(&root,0,"root",0,639,320);
        clrscr();
        initgraph(&gd,&gm,"c:\tc\BGI");
        display(root);
        getch();
        closegraph();
}
create(node **root,int lev,char *dname,int lx,int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

```
        }
    }
```

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 8

8.1 OBJECTIVE

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

8.2 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.

8.3 PROGRAM

```
#include<stdio.h>
struct file
{
        int all[10];
        int max[10];
        int need[10];
        int flag;
};
void main()
{
        struct file f[10];
        int fl;
        int i, j, k, p, b, n, r, g, cnt=0, id, newr;
        int avail[10],seq[10];
        clrscr();
        printf("Enter number of processes -- ");
        scanf("%d",&n);
        printf("Enter number of resources -- ");
        scanf("%d",&r);
        for(i=0;i<n;i++)
        {
                printf("Enter details for P%d",i);
                printf("\nEnter allocation\t -- \t");
                for(j=0;j<r;j++)
                        scanf("%d",&f[i].all[j]);
                printf("Enter Max\t\t -- \t");
                for(j=0;j<r;j++)
                        scanf("%d",&f[i].max[j]);
                f[i].flag=0;
        }
        printf("\nEnter Available Resources\t -- \t");
        for(i=0;i<r;i++)
                scanf("%d",&avail[i]);

        printf("\nEnter New Request Details -- ");
        printf("\nEnter pid \t -- \t");
        scanf("%d",&id);
        printf("Enter Request for Resources \t -- \t");
        for(i=0;i<r;i++)
        {
                scanf("%d",&newr);
                f[id].all[i] += newr;
```

```
                avail[i]=avail[i] - newr;
        }

        for(i=0;i<n;i++)
        {
                for(j=0;j<r;j++)
                {
                        f[i].need[j]=f[i].max[j]-f[i].all[j];
                        if(f[i].need[j]<0)
                                f[i].need[j]=0;
                }
        }
        cnt=0;
        fl=0;
        while(cnt!=n)
        {
                g=0;
                for(j=0;j<n;j++)
                {
                        if(f[j].flag==0)
                        {
                                b=0;
                                for(p=0;p<r;p++)
                                {
                                        if(avail[p]>=f[j].need[p])
                                                b=b+1;
                                        else
                                                b=b-1;
                                }
                                if(b==r)
                                {
                                        printf("\nP%d is visited",j);
                                        seq[fl++]=j;
                                        f[j].flag=1;
                                        for(k=0;k<r;k++)
                                                avail[k]=avail[k]+f[j].all[k];
                                        cnt=cnt+1;
                                        printf("(");
                                        for(k=0;k<r;k++)
                                                printf("%3d",avail[k]);
                                        printf(")");
                                        g=1;
                                }
                        }
                }
                if(g==0)
                {
                        printf("\n REQUEST NOT GRANTED -- DEADLOCK OCCURRED");
                        printf("\n SYSTEM IS IN UNSAFE STATE");
                        goto y;
                }
        }
        printf("\nSYSTEM IS IN SAFE STATE");
        printf("\nThe Safe Sequence is -- (");
        for(i=0;i<fl;i++)
                printf("P%d  ",seq[i]);
        printf(")");
y: printf("\nProcess\t\tAllocation\t\tMax\t\t\tNeed\n");
        for(i=0;i<n;i++)
        {
                printf("P%d\t",i);
                for(j=0;j<r;j++)
```

```
                        printf("%6d",f[i].all[j]);
                for(j=0;j<r;j++)
                        printf("%6d",f[i].max[j]);
                for(j=0;j<r;j++)
                        printf("%6d",f[i].need[j]);
                printf("\n");
        }
        getch();
}
```

INPUT

```
Enter number of processes       –       5
Enter number of resources       --      3
Enter details for P0
Enter allocation        --      0       1       0
Enter Max               --              7       5       3

Enter details for P1
Enter allocation        --      2       0       0
Enter Max               --      3       2       2

Enter details for P2
Enter allocation        --      3       0       2
Enter Max               --      9       0       2

Enter details for P3
Enter allocation        --      2       1       1
Enter Max               --      2       2       2

Enter details for P4
Enter allocation        --      0       0       2
Enter Max               --      4       3       3

Enter Available Resources  --   3   3       2
Enter New Request Details --
Enter pid       --      1
Enter Request for Resources     --      1         0       2
```

OUTPUT

```
P1 is visited(  5  3  2)
P3 is visited(  7  4  3)
P4 is visited(  7  4  5)
P0 is visited(  7  5  5)
P2 is visited( 10  5  7)
SYSTEM IS IN SAFE STATE
The Safe Sequence is -- (P1  P3  P4  P0  P2  )
```

Process	Allocation	Max	Need
P0	0 1 0	7 5 3	7 4 3
P1	3 0 2	3 2 2	0 2 0
P2	3 0 2	9 0 2	6 0 0
P3	2 1 1	2 2 2	0 1 1
P4	0 0 2	4 3 3	4 3 1

EXPERIMENT 9

9.1 OBJECTIVE

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

9.2 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

9.3 PROGRAM

9.3.1 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
```

9.3.2 SCAN DISK SCHEDULING ALGORITHM

```
#include<stdio.h>
main()
{
        int t[20], d[20], h, i, j, n, temp, k, atr[20], tot, p, sum=0;

        clrscr();
        printf("enter the no of tracks to be traveresed");
        scanf("%d'",&n);
        printf("enter the position of head");
        scanf("%d",&h);
        t[0]=0;t[1]=h;
        printf("enter the tracks");
        for(i=2;i<n+2;i++)
                scanf("%d",&t[i]);
        for(i=0;i<n+2;i++)
        {
                for(j=0;j<(n+2)-i-1;j++)
                {       if(t[j]>t[j+1])
                                {
                                        temp=t[j];
                                        t[j]=t[j+1];
                                        t[j+1]=temp;
        }   }   }
        for(i=0;i<n+2;i++)
                if(t[i]==h)
                        j=i;k=i;
        p=0;
        while(t[j]!=0)
        {
                atr[p]=t[j];
                j--;
                p++;
        }
        atr[p]=t[j];
        for(p=k+1;p<n+2;p++,k++)
                atr[p]=t[k+1];
        for(j=0;j<n+1;j++)
        {
                if(atr[j]>atr[j+1])
                        d[j]=atr[j]-atr[j+1];
                else
                        d[j]=atr[j+1]-atr[j];
                sum+=d[j];
        }
        printf("\nAverage header movements:%f",(float)sum/n);
        getch();
}
```

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

9.3.3 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)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 10

10.1 OBJECTIVE

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

10.2 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.

10.3 PROGRAM

10.3.1 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

10.3.2 LRU PAGE REPLACEMENT ALGORITHM

```
#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

10.3.3 LFU PAGE REPLACEMENT ALGORITHM

```
#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 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.h>
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++;
                        }
                        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])
                {
                        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 producer-consumer problem using semaphores.

12.2 DESCRIPTION

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

12.3 PROGRAM

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

OUTPUT

```
1. Produce        2. Consume        3. Exit
Enter your choice: 2
Buffer is Empty
1. Produce        2. Consume        3. Exit
Enter your choice: 1
Enter the value: 100
1. Produce        2. Consume        3. Exit
Enter your choice: 2
The consumed value is 100
1. Produce        2. Consume        3. Exit
Enter your choice: 3
```

EXPERIMENT 13

13.1 OBJECTIVE

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

13.2 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.

13.3 PROGRAM

```
int tph, philname[20], status[20], howhung, hu[20], cho;
main()
{
        int i;
        clrscr();
        printf("\n\nDINING PHILOSOPHER PROBLEM");
        printf("\nEnter the total no. of philosophers: ");
        scanf("%d",&tph);
        for(i=0;i<tph;i++)
        {
                philname[i] = (i+1);
                status[i]=1;
        }
        printf("How many are hungry : ");
        scanf("%d", &howhung);
        if(howhung==tph)
        {
                printf("\nAll are hungry..\nDead lock stage will occur");
                printf("\nExiting..");
        }
        else
        {
                for(i=0;i<howhung;i++)
                {
                        printf("Enter philosopher %d position: ",(i+1));
                        scanf("%d", &hu[i]);
                        status[hu[i]]=2;
                }
                do
                {
                        printf("1.One can eat at a time\t2.Two can eat at a time\t3.Exit\nEnter your choice:");
                        scanf("%d", &cho);
                        switch(cho)
                        {
                                case 1:  one();
                                         break;
                                case 2:  two();
                                         break;
                                case 3:  exit(0);
                                default: printf("\nInvalid option..");
                        }
```

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

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

LAB QUESTIONS & ASSIGNMENTS

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

2.1 PRE-LAB QUESTIONS

1. What is multi-level queue CPU Scheduling?
2. Differentiate between the general CPU scheduling algorithms like FCFS, SJF etc and multi-level queue CPU Scheduling?
3. What are CPU-bound I/O-bound processes?

2.2 POST-LAB QUESTIONS

1. What are the parameters to be considered for designing a multilevel feedback queue scheduler?
2. Differentiate multi-level queue and multi-level feedback queue CPU scheduling algorithms?
3. What are the advantages of multi-level queue and multi-level feedback queue CPU scheduling algorithms?

2.3 ASSIGNMENT QUESTIONS

1. Write a C program to simulate multi-level queue scheduling algorithm considering the following 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 each process priority to be from 1 to 3. Use priority scheduling for the processes in each queue.

EXPERIMENT 3

3.1 PRE-LAB QUESTIONS

1. Define file?
2. What are the different kinds of files?
3. What is the purpose of file allocation strategies?

3.2 POST-LAB QUESTIONS

1. Identify ideal scenarios where sequential, indexed and linked file allocation strategies are most appropriate?
2. What are the disadvantages of sequential file allocation strategy?
3. What is an index block?
4. What is the file allocation strategy used in UNIX?

3.3 ASSIGNMENT QUESTIONS

1. Write a C program to simulate a two-level index scheme for file allocation?

EXPERIMENT 4

4.1 PRE-LAB QUESTIONS

1. What is the purpose of memory management unit?
2. Differentiate between logical address and physical address?
3. What are the different types of address binding techniques?
4. What is the basic idea behind contiguous memory allocation?
5. How is dynamic memory allocation useful in multiprogramming operating systems?

4.2 POST-LAB QUESTIONS

1. Differentiate between equal sized and unequal sized MFT schemes?
2. What is the advantage of MVT memory management scheme over MFT?

4.3 ASSIGNMENT QUESTIONS

1. Write a C program to simulate MFT memory management scheme with unequal sized partitions.

EXPERIMENT 5

5.1 PRE-LAB QUESTIONS

1. Differentiate between the memory management schemes MFT and MVT?
2. What is dynamic memory allocation?
3. What is external fragmentation?

5.2 POST-LAB QUESTIONS

1. Which of the dynamic contiguous memory allocation strategies suffer with external fragmentation?
2. What are the possible solutions for the problem of external fragmentation?
3. What is 50-percent rule?
4. What is compaction?
5. Which of the memory allocation techniques first-fit, best-fit, worst-fit is efficient? Why?

5.3 ASSIGNMENT QUESTIONS

1. Write a C program to implement compaction technique.

EXPERIMENT 6

6.1 PRE-LAB QUESTIONS

1. What are the advantages of noncontiguous memory allocation schemes?
2. What is the process of mapping a logical address to physical address with respect to the paging memory management technique?
3. Define the terms – base address, offset?

6.2 POST-LAB QUESTIONS

1. Differentiate between paging and segmentation memory allocation techniques?
2. What is the purpose of page table?
3. Whether the paging memory management technique suffers with internal or external fragmentation problem. Why?
4. What is the effect of paging on the overall context-switching time?

6.3 ASSIGNMENT QUESTIONS

1. Write a C program to simulate two-level paging technique.
2. Write a C program to simulate segmentation memory management technique.

EXPERIMENT 7

7.1 PRE-LAB QUESTIONS

1. Define directory?
2. Describe the general directory structure?
3. List the different types of directory structures?

7.2 POST-LAB QUESTIONS

1. Which of the directory structures is efficient? Why?
2. Which directory structure does not provide user-level isolation and protection?
3. What is the advantage of hierarchical directory structure?

7.3 ASSI7GNMENT QUESTIONS

1. Write a C to simulate acyclic graph directory structure?
2. Write a C to simulate general graph directory structure?

EXPERIMENT 8

8.1 PRE-LAB QUESTIONS

1. Define resource. Give examples.
2. What is deadlock?
3. What are the conditions to be satisfied for the deadlock to occur?

8.2 POST-LAB QUESTIONS

1. How can be the resource allocation graph used to identify a deadlock situation?
2. How is Banker's algorithm useful over resource allocation graph technique?
3. Differentiate between deadlock avoidance and deadlock prevention?

8.3 ASSIGNMENT QUESTIONS

1. Write a C program to implement deadlock detection technique for the following scenarios?
    a. Single instance of each resource type
    b. Multiple instances of each resource type

EXPERIMENT 9

9.1 PRE-LAB QUESTIONS

1. What is disk scheduling?
2. List the different disk scheduling algorithms?
3. Define the terms – disk seek time, disk access time and rotational latency?

9.2 POST-LAB QUESTIONS

1. What is the advantage of C-SCAN algorithm over SCAN algorithm?
2. Which disk scheduling algorithm has highest rotational latency? Why?

9.3 ASSIGNMENT QUESTIONS

1. Write a C program to implement SSTF disk scheduling algorithm?

EXPERIMENT 10

10.1 PRE-LAB QUESTIONS

1. Define the concept of virtual memory?
2. What is the purpose of page replacement?
3. Define the general process of page replacement?
4. List out the various page replacement techniques?
5. What is page fault?

10.2 POST-LAB QUESTIONS

1. Which page replacement algorithm suffers with the problem of Belady's anomaly?
2. Define the concept of thrashing? What is the scenario that leads to the situation of thrashing?

10.3 ASSIGNMENT QUESTIONS

1. Write a C program to simulate LRU-approximation page replacement algorithm?
   a. Additional-Reference bits algorithm
   b. Second-chance algorithm

EXPERIMENT 11

11.1 PRE-LAB QUESTIONS

1. What are the benefits of optimal page replacement algorithm over other page replacement algorithms?

11.2 POST-LAB QUESTIONS

1. Why can't the optimal page replacement technique be used in practice?

EXPERIMENT 12

12.1 PRE-LAB QUESTIONS

1. What is the need for process synchronization?
2. Define a semaphore?
3. Define producer-consumer problem?

12.2 POST-LAB QUESTIONS

1. Discuss the consequences of considering bounded and unbounded buffers in producer-consumer problem?
2. Can producer and consumer processes access the shared memory concurrently? If not which technique provides such a benefit?

12.3 ASSIGNMENT QUESTIONS

1. Write a C program to simulate producer-consumer problem using message-passing system.

EXPERIMENT 13

13.1 PRE-LAB QUESTIONS

1. Differentiate between a monitor, semaphore and a binary semaphore?
2. Define clearly the dining-philosophers problem?

13.2 POST-LAB QUESTIONS

1. Identify the scenarios in the dining-philosophers problem that leads to the deadlock situations?

13.3 ASSIGNMENT QUESTIONS

1. Write a C program to simulate readers-writers problem using monitors?