EXPERIMENT NO: 1 [A]

CPU SCHEDULING: FCFS

AIM:

To write a C program to simulate the FCFS - CPU scheduling algorithm.

ALGORITHM:

Step 1: Create the number of process.

Step 2: Get the ID and Service time for each process.

Step 3: Initially, Waiting time of first process is zero and Total time for the first process is the starting time of that process.

Step 4: Calculate the Total time and Processing time for the remaining processes.

Step 5: Waiting time of one process is the Total time of the previous process.

Step 6: Total time of process is calculated by adding Waiting time and Service time.

Step 7: Total waiting time is calculated by adding the waiting time for lack process.

Step 8: Total turn around time is calculated by adding all total time of each process.

Step 9: Calculate Average waiting time by dividing the total waiting time by total number of process.

Step 10: Calculate Average turn around time by dividing the total time by the number of process.

Step 11: Display the result.

PROGRAM:

//1a First come first serve scheduling

```
#include<stdio.h>
int main()
{
    int n,i,a,b,f=0,t=0,w=0;
    printf("Enter number of processes: ");
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
        printf("Process %d\n",i+1);
        printf("Arrival Time: ");
        scanf("%d",&a);
        printf("Burst Time: ");
        scanf("%d",&b);
        f=b+f;
        t=f-a;
        w=t-b;
        printf("\nFinish Time: %d\n",f);
        printf("Turn Around Time: %d\n",t);
        printf("Waiting Time: %d\n\n\n",w);
    }
    return 0;
}
```

RESULT:

Thus the program to stimulate the FCFS scheduling algorithm is successfully executed

EXPERIMENT NO: 1 [B]

CPU SCHEDULING: SHORTEST JOB FIRST

AIM:

To write a C program to implement and simulate SJF Scheduling Algorithm.

ALGORITHM:

Step 1: Declare the array size.

Step 2: Get the number of elements to be inserted.

Step 3: Select the process, which have shortest burst, will execute first.

Step 4: If two processes have same burst length then FCFS scheduling algorithm used.

Step 5: Make the average waiting the length of next process.

Step 6: Start with the first process from it's selection as above and let other process to be in Queue.

Step 7: Calculate the total number of burst time.

Step 8: Display the values.

PROGRAM:

//1b Shortest Job First scheduling

```
#include<stdio.h>
int main()
{
    int n,i,bt[5],wt[5],tat[5],a[5],t1,t2,j,t=0,w=0;
    printf("Enter number of processes: ");
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
        printf("Enter Service Time for process %d: ",i+1);
        scanf("%d",&bt[i]);
        a[i]=i+1;
    }
    for(i=0;i<n;i++)
    {
        for(j=i+1;j<n;j++)
        {
            if(bt[i]>bt[j])
            {
                t1=bt[j];
                bt[j]=bt[i];
                bt[i]=t1;
                t1=a[j];
                a[j]=a[i];
                a[i]=t1;
            }
        }
    }
    wt[0]=0;
    printf("\nGnatt Chart for Waiting time:\n0 ");
    for(i=1;i<n;i++)
    {
        wt[i]=wt[i-1]+bt[i-1];
        w=w+wt[i];
        printf("%d ",wt[i]);
    }
    printf("\nAverage WT: %d",w/n);
    printf("\nGnatt Chart for Turn Around Time:\n");
    for(i=0;i<n;i++)
    {
        tat[i]=wt[i]+bt[i];
        t=t+tat[i];
        printf("%d ",tat[i]);
    }
```

```
    printf("\nAverage TAT: %d\n",t/n);
    printf("Table:\n");
    for(i=0;i<n;i++)
    {
        printf("P%d  %d  %d  %d\n",a[i],bt[i],wt[i],tat[i]);
    }
    return 0;
}
```

RESULT:

The C program to perform Shortest Job First Scheduling was successfully implemented and executed and the results verified.

EXPERIMENT NO: 1 [C]

CPU SCHEDULING: PRIORITY SCHEDULING

AIM:

To write a C program to implement and simulate Priority Scheduling algorithm.

ALGORITHM:

STEP 1: Declare the array size.

STEP 2: Get the number of elements to be inserted.

STEP 3: Get the priority for each process and value

STEP 4: Start with the higher priority process from it's initial position let other process to be queue.

STEP 5: Calculate the total number of burst time.

STEP 6: Display the values

PROGRAM:

//1c Priority scheduling

```
#include<stdio.h>
int main()
{
    int n,i,bt[5],wt[5],tat[5],a[5],p[5],t1,j,t=0,w=0;
    printf("Enter number of processes: ");
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
        printf("Enter Service Time and Priority for process %d:
",i+1);
        scanf("%d%d",&bt[i],&p[i]);
        a[i]=i+1;
    }
    for(i=0;i<n;i++)
    {
        for(j=i+0;j<n;j++)
        {
            if(p[i]<p[j])
            {
                t1=p[j];
                p[j]=p[i];
                p[i]=t1;
                t1=a[j];
                a[j]=a[i];
                a[i]=t1;
                t1=bt[j];
                bt[j]=bt[i];
                bt[i]=t1;
            }
        }
    }
    wt[0]=0;
    printf("\nGnatt Chart for Waiting time:\n");
    for(i=1;i<n;i++)
    {
        wt[i]=wt[i-1]+bt[i-1];
        w=w+wt[i];
        printf("%d ",wt[i]);
    }
    printf("\nAverage WT: %d",w/n);
    printf("\nGnatt Chart for Turn Around Time:\n");
    for(i=0;i<n;i++)
    {
```

```
        tat[i]=wt[i]+bt[i];
        t=t+tat[i];
        printf("%d ",tat[i]);
    }
    printf("\nAverage TAT: %d\n",t/n);
    printf("Table:\n");
    for(i=0;i<n;i++)
    {
        printf("P%d    %d    %d    %d
%d\n",a[i],bt[i],p[i],wt[i],tat[i]);
    }
    return 0;
}
```

RESULT:

The C program to perform Priority Scheduling was successfully implemented and executed and the results verified.

EXPERIMENT NO: 1 [D]

CPU SCHEDULING: ROUND ROBIN

AIM:

To write a C program to implement and execute Round Robin CPU Scheduling.

ALGORITHM:

STEP 1: Declare the array size.

STEP 2: Get the number of elements to be inserted.

STEP 3: Get the value.

STEP 4: Set the time sharing system with preemption.

STEP 5: Define quantum is defined from 10 to 100ms.

STEP 6: Declare the queue as a circular.

STEP 7: Make the CPU scheduler goes around the ready queue allocating CPU to each process for the time interval specified.

STEP 8: Make the CPU scheduler picks the first process and sets time to interrupt after quantum expired dispatches the process.

STEP 9: If the process has burst less than the time quantum than the process releases the CPU.

PROGRAM:

//1d Round Robin Scheduling

```
#include<stdio.h>
int main()
{
    int bt[4],wt[4],tat[4],p[4],n=0,t=0,tt=0,w=0,ta=0,i=0,j=0;
    printf("Enter number of processes: ");
    scanf("%d",&n);
    printf("Enter Burst time for each process:\n");
    for(i=0;i<n;i++)
    {
        scanf("%d",&bt[i]);
        tt+=bt[i];
        p[i]=bt[i];
    }
    printf("Enter Quantum Time: ");
    scanf("%d",&t);
    i=0;
    while(j<tt)
    {
        if(bt[i]==0);
        else if(bt[i]<=t && bt[i]>0)
        {
            j+=bt[i];
            bt[i]=0;
            printf("%d ",j);
            tat[i]=j;
        }
        else
        {
            j+=t;
            if(bt[i]>0)
            {
                bt[i]=bt[i]-t;
                printf("%d ",j);
            }
        }
        if(i==3)
            i=0;
        else
            i++;
    }
    printf("\n");
    for(i=0;i<n;i++)
    {
```

```
        wt[i]=tat[i]-p[i];
        printf("P%d\t%d\t%d\t%d\n",i+1,p[i],tat[i],wt[i]);
        w+=wt[i];
        ta+=tat[i];
    }
    printf("\nAverage Waiting Time: %d",w/t);
    printf("\nAverage Turn Around Time: %d\n",ta/n);
    return 0;
}
```

RESULT:

The C program to perform Round Robin Scheduling was successfully implemented and executed and the results verified.

EXPERIMENT NO: 2 [A]

FIXED MEMORY PARTITIONING

AIM:

To write a C program to implement and execute Fixed Memory Partitioning

ALGORITHM:

Step 1: Declare the necessary variables.

Step 2: Get the memory capacity and number of processes.

Step 3: Get the memory required for each process.

Step 4: If the needed memory is available for the particular process it will be allocated and the remaining memory availability will be calculated.

Step 5: If not it has to tell no further memory remaining and the process will not be allocated with memory.

Step 6: Then the external fragmentation of each process must be calculated.

PROGRAM:

//2a Fixed Memory Partitioning

```
#include<stdio.h>
int main()
{
    int mem,m,a[5],i,n,s=0,h=0,p;
    printf("Enter total memory: ");
    scanf("%d",&mem);
    mem-=100;
    printf("100 MB used by OS\nRemaining Memory: %d\n",mem);
    printf("Enter fixed memory: ");
    scanf("%d",&m);
    printf("Enter no. of processes: ");
    scanf("%d",&n);
    printf("Enter Memories:\n");
    for(i=0;i<n;i++)
    {
        scanf("%d",&a[i]);
        s+=a[i];
    }
    for(i=0;i<n;i++)
    {
        p=m-a[i];
        printf("P%d\t%d\t",i+1,a[i]);
        if(p==0)
        {
            printf("No Hole Memory\n");
        }
        else if(p<0)
        {
            printf("Memory Overflow\n");
        }
        else
        {
            printf("%d\n",p);
            h+=p;
        }
    }
    printf("Hole Memory left: %d\n",h);
    printf("Total Memory left: %d\n\n",mem-(n*m)+h);
    return 0;
}
```

RESULT:

The C program to perform Fixed Memory Partitioning was successfully implemented and executed and the results verified.

EXPERIMENT NO: 2 [B]

UNEQUAL MEMORY PARTITIONING

AIM:

To write a C program to implement and execute Unequal Memory Partitioning.

ALGORITHM:

Step 1: Declare the necessary variables.

Step 2: Get the memory capacity and number of processes.

Step 3: calculate the number of partitions the total memory has to be divided.

Step 4: Get the required memory for each process if the required memory is available in that particular partition the process will be allocated to that partition and the internal fragmentation will be calculated.

Step 5: If the required memory not available then the process will not be allocated to that partition .

Step 6: Calculate the external fragmentation and total number of fragmentation.

PROGRAM:

//2b Unequal Memory Partitioning

```
#include<stdio.h>
int main()
{
    int mem,m=0,a[5],b[5],i,n,s=0,h=0,p,q;
    printf("Enter total memory: ");
    scanf("%d",&mem);
    mem-=100;
    printf("100 MB used by OS\nRemaining Memory: %d\n",mem);
    printf("Enter no. of processes: ");
    scanf("%d",&n);
    printf("Enter Memories:\n");
    for(i=0;i<n;i++)
    {
        scanf("%d",&a[i]);
        s+=a[i];
    }
    printf("Enter number of partitions of memory: ");
    scanf("%d",&q);
    printf("Enter the memory partitions:\n");
    for(i=0;i<q;i++)
    {
        scanf("%d",&b[i]);
    }
    for(i=0;i<n;i++)
    {
        p=b[i]-a[i];
        printf("P%d\t%d\t%d\t",i+1,a[i],b[i]);
        if(p==0)
        {
            printf("No Hole Memory\n");
        }
        else
        {
            printf("%d\n",p);
            h+=p;
        }
    }
    for(i=n;i<q;i++)
    {
        m+=b[i];
    }
    printf("Hole Memory left: %d\n",h);
    printf("Total Memory left: %d\n\n",m+h);
```

```
return 0;}
```

RESULT:

The C program to perform Unequal Memory Partitioning was successfully implemented and executed and the results verified.

EXPERIMENT NO: 2 [C]

DYNAMIC MEMORY ALLOCATION

AIM:

To write a C program to implement and execute Dynamic Memory Allocation.

ALGORITHM:

Step 1: Start the program.

Step 2: Get total memory from the user.

Step 3: Print remaining memory

Step 4: Get number of processes and memories from the user.

Step 5: Calculate the total memory used and total memory left.

Step 6: Display the result.

Step 7: Stop the program.

PROGRAM:

//2c Dynamic Memory Allocation

```
#include<stdio.h>
int main()
{
    int m,n,a[5],s=0,i;
    printf("Enter total memory: ");
    scanf("%d",&m);
    m-=100;
    printf("100 MB used by OS.\nRemaining Memory: %d\n",m);
    printf("Enter number of processes: ");
    scanf("%d",&n);
    printf("Enter process memories: \n");
    for(i=0;i<n;i++)
    {
        scanf("%d",&a[i]);
        s+=a[i];
    }
    printf("Total Memory Left: %d\n",m-s);
    printf("Total Memory Used: %d\n",s);
    for(i=0;i<n;i++)
    {
        printf("P%d %d\n",i+1,a[i]);
    }
    return 0;
}
```

RESULT:

The C program to perform Dynamic Memory Allocation was successfully implemented and executed and the results verified.

EXPERIMENT NO: 3 [A]

SEQUENTIAL FILE ALLOCATION

AIM:

To write a C program to implement and execute Sequential File Allocation.

ALGORITHM:

Step 1: Start the program.

Step 2: Get the number of files.

Step 3: Get the memory requirement of each file.

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

a). Randomly select a location from available location s1= random(100);
b). Check whether the required locations are free from the selected location.
c). Allocate and set flag=1 to the allocated locations.

Step 5: Print the results fileno, length , Blocks allocated.

Step 6: Stop the program.

PROGRAM:

//3a Sequential File Allocation

```
#include<stdio.h>
int main()
{
    int a[10],n=0,l=0,i=0;
    printf("Enter Length: ");
    scanf("%d",&l);
    printf("Enter Starting Block: ");
    scanf("%d",&n);
    for(i=0;i<l;i++)
    {
        a[i]=1;
        printf("Blovk %d: %d\n",i+n,a[i]);
    }
    return 0;
}
```

RESULT:

The C program to perform Sequential File Allocation was successfully implemented and executed and the results verified.

EXPERIMENT NO: 3 [B]

LINKED FILE ALLOCATION

AIM:

To write a C program to implement and execute Linked File Allocation.

ALGORITHM:

Step 1: Start the Program.

Step 2: Get the number of files.

Step 3: Allocate the required locations by selecting a location randomly.

Step 4: Check whether the selected location is free.

Step 5: If the location is free allocate and set flag =1 to the allocated locations.

Step 6: Print the results file no, length, blocks allocated.

Step 7: Stop the execution.

PROGRAM:

//3b Linked File Allocation

```
#include<stdio.h>
int main()
{
    int a[10],n=0,m=0,l=0,i=0,j=0;
    printf("Enter Length: ");
    scanf("%d",&l);
    for(i=0;i<l;i++)
    {
        a[i]=1;
    }
    printf("Enter Starting Block: ");
    scanf("%d",&n);
    printf("Enter number of blocks already allotted: ");
    scanf("%d",&m);
    printf("Enter Blocks:\n");
    for(i=0;i<m;i++)
    {
        scanf("%d",&j);
        a[j-n]=0;
    }
    for(i=0;i<l;i++)
    {
        printf("Block %d: %d\n",i+n,a[i]);
    }
    return 0;
}
```

RESULT:

The C program to perform Linked File Allocation was successfully implemented and executed and the results verified.

EXPERIMENT NO: 3 [C]

INDEXED FILE ALLOCATION

AIM:

To write a C program to implement and execute Indexed File Allocation.

ALGORITHM:

Step 1: Start the Program

Step 2: Get the number of files.

Step 3: Get the memory requirement of each file.

Step 4: Allocate the required locations by selecting a location randomly.

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

Step 7: Stop the execution.

PROGRAM:

//3c Indexed File Allocation

```
#include<stdio.h>
int main()
{
    int a[10],n=0,m=0,l=0,i=0,j=0;
    printf("Enter Length: ");
    scanf("%d",&l);
    for(i=0;i<l;i++)
    {
        a[i]=1;
    }
    printf("Enter Index Block: ");
    scanf("%d",&n);
    printf("Enter number of blocks already allotted: ");
    scanf("%d",&m);
    printf("Enter Blocks:\n");
    for(i=0;i<m;i++)
    {
        scanf("%d",&j);
        a[j-1]=0;
    }
    for(i=0;i<l;i++)
    {
        printf("%d : %d : %d\n",n,i+1,a[i]);
    }
    return 0;
}
```

RESULT:

The C program to perform Indexed File Allocation was successfully implemented and executed and the results verified.

EXPERIMENT NO: 4 [A]

DEADLOCK AVOIDANCE - BANKER'S ALGORITHM

AIM:

To write a C program to implement and execute Deadlock Avoidance - Banker's Algorithm.

ALGORITHM:

Step 1: Start the Program

Step 2: Get the values of resources and processes.

Step 3: Get the avail value.

Step 4: After allocation find the need value.

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

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

Step 7: Stop the execution

PROGRAM:

//4a Deadlock Avoidance - Banker's Algorithm

```
#include<stdio.h>
int main()
{
    int a[3][5],b[3][5],c[3],d[3],e[3],i,j,k;
    printf("Enter Allocated Memory:\n");
    for(j=0;j<5;j++)
    {
        for(i=0;i<3;i++)
        {
            scanf("%d",&a[i][j]);
        }
    }
    printf("Enter Maximum Resources:\n");
    for(j=0;j<5;j++)
    {
        for(i=0;i<3;i++)
        {
            scanf("%d",&b[i][j]);
        }
    }
    printf("Enter Total Available Memory:\n");
    for(i=0;i<3;i++)
    {
        scanf("%d",&c[i]);
        d[i]=c[i];
    }
    printf("Available Memory:\n");
    for(i=0;i<3;i++)
    {
        for(j=0;j<5;j++)
        {
            c[i]-=a[i][j];
        }
    }
    for(i=0;i<3;i++)
    {
        printf("%d ",c[i]);
    }
    printf("\nNeed\n");
    for(j=0;j<5;j++)
    {
        for(i=0;i<3;i++)
        {
```

```
            a[i][j]=b[i][j]-a[i][j];
            printf("%d ",a[i][j]);
        }
        printf("\n");
    }
    for(k=0;k<5;k++)
    {
        for(j=0;j<5;j++)
        {
            if(a[0][j]<=c[0] && a[1][j]<=c[1] && a[2][j]<=c[2])
            {
                printf(" %d %d %d\n-%d %d %d\n-------\n=%d %d
%d\n\n\n",c[0],c[1],c[2],a[0][j],a[1][j],a[2][j],c[0]-
a[0][j],c[1]-a[1][j],c[2]-a[2][j]);
                c[0]=c[0]-a[0][j]+b[0][j];
                c[1]=c[1]-a[1][j]+b[1][j];
                c[2]=c[2]-a[2][j]+b[2][j];
                for(i=0;i<3;i++)
                {
                    a[i][j]=100;
                }
                e[k]=j+1;
                break;
            }
        }
    }
    printf("\n\n");
    for(i=0;i<5;i++)
    {
        printf("P%d ",e[i]);
    }
    printf("\nTotal available Memory:\n");
    for(i=0;i<3;i++)
    {
        printf("%d ",c[i]);
    }
    printf("\n");
    return 0;
}
```

RESULT:

The C program to perform Deadlock Avoidance - Banker's Algorithm was successfully implemented and executed and the results verified.

EXPERIMENT NO: 4 [B]

DEADLOCK DETECTION

AIM:

To write a C program to implement and execute Deadlock Detection.

ALGORITHM:

Step 1: Start the Program

Step 2: Get the values of resources and processes.

Step 3: Get the avail value..

Step 4: After allocation find the needed value.

Step 5: Check whether its possible to allocate.

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

Step 7: Stop the execution

PROGRAM:

//4b Deadlock Detection

```
#include<stdio.h>
int main()
{
    int a[4][5],b[4][5],c[5],d[5],i=0,j=0,k=0,f=0,s=0;
    printf("Enter Request Matrix:\n");
    for(i=0;i<4;i++)
    {
        for(j=0;j<5;j++)
        {
            scanf("%d",&a[i][j]);
        }
    }
    printf("Enter Allocate Matrix:\n");
    for(i=0;i<4;i++)
    {
        for(j=0;j<5;j++)
        {
            scanf("%d",&b[i][j]);
        }
    }
    printf("Resource Vector:\n");
    for(i=0;i<5;i++)
    {
        scanf("%d",&c[i]);
        d[i]=c[i];
    }
    printf("\n");
    for(i=0;i<4;i++)
    {
        for(j=0;j<5;j++)
        {
            c[j]-=b[i][j];
        }
    }
    printf("Available: \n");
    for(i=0;i<5;i++)
    {
        printf("%d ",c[i]);
    }
    printf("\n");
    for(i=0;i<4;i++)
    {
        s=0;
```

```
        for(j=0;j<5;j++)
        {
            s+=b[i][j];
        }
        if(s==0)
        {
            for(j=0;j<5;j++)
            {
                a[i][j]=9;
            }
            printf("No need to allocate resources to
P%d\n",i+1);
        }
    }
    for(i=0;i<4;i++)
    {
            f=0;
            for(j=0;j<5;j++)
            {
                if(c[j]<a[i][j])
                    f=1;
            }
            if(f==0)
            {
                for(j=0;j<5;j++)
                {
                    c[j]+=b[i][j];
                    a[i][j]=9;
                }
            }
    }
    printf("\n");
    printf("There is a deadlock between: ");
    for(i=0;i<4;i++)
    {
        if(a[i][0]!=9)
        {
            printf("P%d ",i+1);
        }
    }
    return 0;
}
```

RESULT: The C program to perform Deadlock Detection was successfully implemented and executed and the results verified.

EXPERIMENT NO: 5 [A]

PAGE REPLACEMENT (FCFS)

AIM:

To write a C program to implement and execute Page Replacement by FCFS.

ALGORITHM:

Step 1: Start the Program

Step 2: Read number of pages n.

Step 3: Read number of pages no.

Step 4: Read page numbers into an arraya[i].

Step 5: Initialize aval[i]=o, to check page hit.

Step 6: Print the results.

Step 7: Stop the Process.

PROGRAM:

//5a FCFS Page Replacement

```
#include<stdio.h>
int main()
{
    int a[3]={-1,-1,-1},n=0,pf=0,i=0,j=0;
    printf("Enter number of frames: ");
    scanf("%d",&n);
    for(i=0;i<19;i++)
    {
        scanf("%d",&n);
        if(n!=a[0] && n!=a[1] && n!=a[2])
        {
            a[j]=n;
            if(j<2)
                j++;
            else
                j=0;
            pf++;
        }
    }
    printf("Total Page Faults: %d",pf);
    return 0;
}
```

RESULT:

The C program to perform Deadlock Detection was successfully implemented and executed and the results verified.

EXPERIMENT NO: 5 [B]

PAGE REPLACEMENT (LRU)

AIM:

To write a C program to implement and execute Page Replacement using Least Recently Used Algorithm.

ALGORITHM:

Step 1: Start the Program.

Step 2: Declare the size.

Step 3: Get the number of pages to be issued.

Step 4: Declare counter and stack.

Step 5: Select the LRU page by counter value.

Step 6: Stack them according the selection.

Step 7: Stop the execution.

PROGRAM:

//5b Least Recently Used Page Replacement

```
#include<stdio.h>
int main()
{
    int a[3]={-1,-1,-1},b[3]={4,3,2},i,j,p,pf=0,n;
    printf("Enter frame size: ");
    scanf("%d",&n);
    for(i=0;i<20;i++)    {
        scanf("%d",&p);
        if(a[0]!=p && a[1]!=p && a[2]!=p)        {
            if(b[0]>b[1] && b[0]>b[2])
            {
                a[0]=p;
                b[0]=1;
                b[1]++;
                b[2]++;
            }
            else if(b[1]>b[0] && b[1]>b[2])
            {
                a[1]=p;
                b[1]=1;
                b[0]++;
                b[2]++;
            }
            else
            {
                a[2]=p;
                b[2]=1;
                b[0]++;
                b[1]++;
            }
            pf++;
        }
        else        {
            for(j=0;j<n;j++)            {
                b[j]++;
                if(a[j]==p)                {
                    b[j]=1;
                }
            }
        }
    }
    printf("Number of page faults: %d",pf);
    return 0;}
```

RESULT:

The C program to perform Page Replacement using Least Recently Used Algorithm was successfully implemented and executed and the results verified.

EXPERIMENT NO: 6

MEMORY MANAGEMENT USING PAGING TECHNIQUE

AIM:

To write a C program to implement Memory Management using Paging Technique.

ALGORITHM:

Step 1: Read all the necessary input from the keyboard.

Step 2: Pages – Logical memory is broken into fixed- sized blocks.

Step 3: Frames – physical memory is broken into fixed – sized blocks.

Step 4: Calculate the physical address using the following

*Physical address=(frame number * Frame size)+ offset*

Step 5: Display the physical address.

Step 6: Stop the execution.

PROGRAM:

//6 Paging

```
#include<stdio.h>
int main()
{
    int n,p,i,sp[5];
    printf("Enter number of pages: ");
    scanf("%d",&n);
    printf("Enter page size: ");
    scanf("%d",&p);
    for(i=0;i<n;i++)
    {
        sp[i]=malloc(p);
        printf("Page %d Address %u\n",i+1,sp[i]);
    }
    return 0;
}
```

RESULT:

The C program to perform Memory Management using Paging Technique was successfully implemented and executed and the results verified.

EXPERIMENT NO: 7 [A]

SINGLE FILE ORGANIZATION

AIM:

To write a C program to implement and execute Single File Organization.

ALGORITHM:

Step 1: Start the Program.

Step 2: Initialize values gd=DETECT,gm,count,i,j,mid,cir_x.

Step 3: Initialize graph function.

Step 4: Set back ground color with setbkcolor();

Step 5: Read number of files in variable count.

Step 6: check i<count; mid=640/count;

Step 7: Stop the execution.

PROGRAM:

//7a Single File Organization

```
#include <stdio.h>
#include <stdlib.h>
#include <process.h>
#include <dirent.h>
#include <sys/types.h>
#include <string.h>
int main()
{
    DIR *d;
    struct dirent *dir;
    int choice, count=0, index=0, check, i;
    FILE *f = NULL;
    char fname[50], name[256][256];
    do    {
        char loc[50] ="D:\\oslab\\";
        printf("Enter Filename to save (with the extension): ");
        scanf("%s",&fname);

        d = opendir("D:\\oslab\\");
        if(d)        {
            while((dir = readdir(d)) != NULL)            {
                strcpy(name[count], dir->d_name);
                count++;
            }
            closedir(d);
        }
        for(i=0; i<count; i++)        {
            check = strcmp(fname, name[i]);
            if(check==0)            {
                printf("File already present!!\n");
                goto last;
            }
        }
        strcat(loc,fname);
        printf("%s", &loc);
        f = fopen(loc,"w+");
        if(f!=NULL)
            printf("File Saved\n");
        else
            printf("File not Saved\n");
        printf(" Files in Directory: \n");
        system("dir D:\\oslab\\");
    last:
```

```
    printf(" Want to save another file? (1/0): ");
    scanf("%d",&choice);
} while(choice==1);
return 0;}
```

RESULT:

The C program to perform Single File Organization was successfully implemented and executed and the results verified.

EXPERIMENT NO: 7 [B]

MULTIPLE FILE ORGANIZATION

AIM:

To write a C program to implement and execute Multiple File Organization.

ALGORITHM:

Step 1: Start the Program

Step 2: Initialize structure elements

Step 3: Start main function

Step 4: Set variables gd =DETECT, gm;

Step 5: Create structure using create (&root,0,"null",0,639,320);

Step 6: initgraph(&gd,&gm,"c:\tc\bgi");

Step 7: Stop the execution

PROGRAM:

//7b Multiple File Organisation

```
#include <stdio.h>
#include <stdlib.h>
#include <process.h>
#include <sys/types.h>
#include <dirent.h>
#include <string.h>
int main(){
    DIR *d;
    struct dirent *dir;
    int choice, count=0, index=0, check, i, choice2;
    FILE *f = NULL;
    char fname[50], dname[50], name[256][256];
    printf("Folders Present in the Folder 'Multiple' are: \n");
    printf("1) DBMS\t\t2) OS\n");
    do    {
        char loc1[50]="dir D:\\Multiple\\",
loc[50]="D:\\Multiple\\";
        printf("Enter Folder name to save the file into: ");
        scanf("%s",&dname);
        if(strcmp(dname,"dbms")!=0 && strcmp(dname,"os")!=0)
{
            printf("Wrong Directory name Entered..!!\nEnter one
of these:\n");
            printf("1) DBMS\t\t2) OS\n");
            goto last;
        }
        strcat(loc,dname);
        strcat(loc,"\\");
        printf("Enter Filename to save (with the extension): ");
        scanf("%s",&fname);
        d = opendir(loc);
        if(d)        {
            while((dir = readdir(d)) != NULL)            {
                strcpy(name[count], dir->d_name);
                count++;
            }
            closedir(d);
        }
        for(i=0; i<count; i++)        {
            check = strcmp(fname, name[i]);
            if(check==0)            {
                printf("File already present!!\n");
                goto last;
```

```
            }
        }
        strcat(loc,fname);
        f = fopen(loc,"w+");
        if(f!=NULL)
            printf("File Saved\n");
        else
            printf("File not Saved\n");
        printf("Want to see the files in this directory?
(1/0)");
        scanf("%d", &choice2);
        if(choice2==1)        {
            strcat(loc1,dname);
            printf(" Files in Directory: \n");
            system(loc1);
        }
    last:
        printf(" Want to save another file? (1/0): ");
        scanf("%d",&choice);
    } while(choice==1);
    return 0;
}
```

RESULT:

The C program to perform Multiple File Organization was successfully implemented and executed and the results verified.