OPERATING SYSTEMS

LABORATORY MANUAL

B.TECH
(II YEAR – II SEM)

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND DATA SCIENCE

INDEX

EXPERIMENT.NO 2
MEMORY MANAGEMENT

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

AIM: To implement and simulate the MFT algorithm.

DESCRIPTION:

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

SOURCE CODE :

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

INPUT

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

Enter the block size (in Bytes)-- 300

Enter the number of processes – 5

Enter memory required for process 1 (in Bytes) -- 275

Enter memory required for process 2 (in Bytes) -- 400

Enter memory required for process 3 (in Bytes) -- 290

Enter memory required for process 4 (in Bytes) -- 293

Enter memory required for process 5 (in Bytes) -- 100

No. of Blocks available in memory -- 3

OUTPUT

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

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

Total External Fragmentation is 100

B) MEMORY VARIABLE PARTIONING TYPE (MVT)

AIM: To write a program to simulate the MVT algorithm

SOURCE CODE:

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

OUTPUT:

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

Memory is Full

Total Memory Available – 1000

PROCESS	MEMORY ALLOCATED
1	400
2	275

Total Memory Allocated is 675
Total External Fragmentation is 325

EXPERIMENT.NO 3

DEAD LOCK AVOIDANCE

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

DESCRIPTION:

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

Data structures

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

Safety Algorithm

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

   If no such I exists go to step 4.
3. work= work + Allocation, Finish[i] =True;
4. if Finish[1]=True for all I, then the system is in safe state.

Resource request algorithm

Let Request i be request vector for the process Pi, If request i=[j]=k, then process Pi wants k instances of resource type Rj.

1. if Request<=Need I go to step 2. Otherwise raise an error condition.
2. if Request<=Available go to step 3. Otherwise Pi must since the resources are available.
3. Have the system pretend to have allocated the requested resources to process Pi by modifying the state as follows;

   Available=Available-Request I;

   Allocation I=Allocation +Request I;

   Need i=Need i- Request I;

   If the resulting resource allocation state is safe, the transaction is completed and process Pi is allocated its resources. However if the state is unsafe, the Pi must wait for Request i and the old resource-allocation state is restored.

SOURCE CODE :

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

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

OUTPUT

```
Enter the no. of processes and resources: 4 3
Enter the claim matrix:
3 2 2
6 1 3
3 1 4
4 2 2
Enter the allocation matrix:
1 0 0
6 1 2
2 1 1
0 0 2
Resource vector:9 3 6
All the resources can be allocated to Process 2
Available resources are: 6 2 3
Process 2 executed?:y
All the resources can be allocated to Process 3 Available resources
are: 8 3 4
Process 3 executed?:y
All the resources can be allocated to Process 4 Available resources
are: 8 3 6
Process 4 executed?:y
All the resources can be allocated to Process 1
Available resources are: 9 3 6
Process 1 executed?:y
System is  in safe mode
The given state is safe state
```

EXPERIMENT NO.4

PAGE REPLACEMENT ALGORITHMS

AIM: To implement FIFO page replacement technique.

a) FIFO b) LRU c) OPTIMAL

DESCRIPTION:

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

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

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

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

A) FIRST IN FIRST OUT

SOURCE CODE :

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

```
if(fr[i]==-1)
{
fr[i]=page[j]; flag2=1; break;
}
}
}
if(flag2==0)
{
fr[top]=page[j];
top++;
pf++;
if(top>=frsize)
top=0;
}
display();
}
printf("Number of page faults : %d ",pf+frsize);getch();
}
void display()
{
int     i;     printf("\n");
for(i=0;i<3;i++)
printf("%d\t",fr[i]);
}
```

OUTPUT:

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

Number of page faults: 9

B) LEAST RECENTLY USED

AIM: To implement LRU page replacement technique.

SOURCE CODE :

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

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

OUTPUT:

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

No of page faults: 7

C) LFU

AIM: To implement LFU page replacement technique.

Introduction to LFU page replacement :

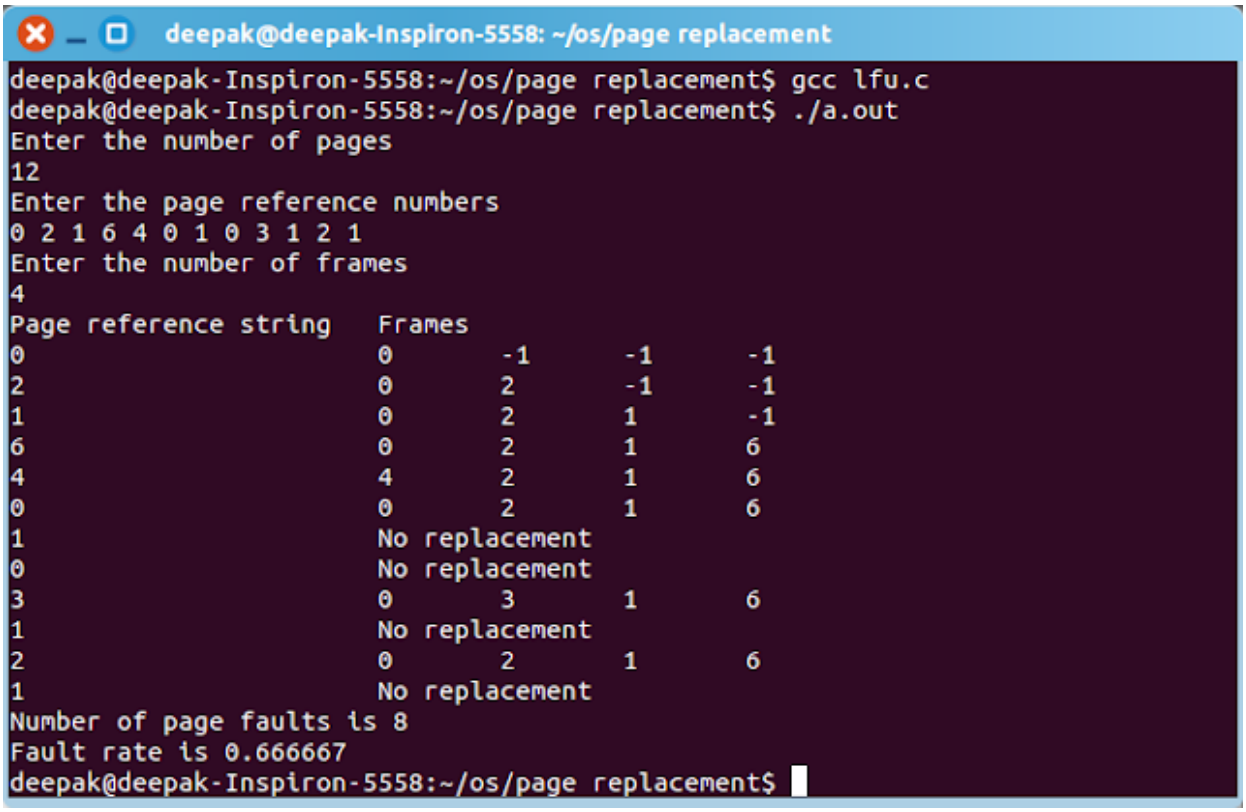
The least frequently used(LFU) page-replacement algorithm requires that the page with the smallest count be replaced. The reason for this selection is that an actively used page should have a large reference count. A problem arises, however, when a page is used heavily during the initial phase of a process but then is never used again. Since it was used heavily, it has a large count and remains in memory even though it is no longer needed. One solution is to shift the counts right by1bit at regular intervals,forming an exponentially decaying average usage count.

C program for LFU page replacement :

```
#include<stdio.h>
void print(int frameno,int frame[])
{
        int j;
          for(j=0;j<frameno;j++)
          printf("%d\t",frame[j]);
          printf("\n");
}
int main()
{
int i,j,k,n,page[50],frameno,frame[10],move=0,flag,count=0,count1[10]={0},repindex,leastcount;
float rate;
printf("Enter the number of pages\n");
scanf("%d",&n);
printf("Enter the page reference numbers\n");
  for(i=0;i<n;i++)
 scanf("%d",&page[i]);
printf("Enter the number of frames\n");
scanf("%d",&frameno);
for(i=0;i<frameno;i++)
     frame[i]=-1;
 printf("Page reference string\tFrames\n");
for(i=0;i<n;i++)
 {
  printf("%d\t\t\t",page[i]);
 flag=0;
for(j=0;j<frameno;j++)
{
if(page[i]==frame[j])
{
flag=1;
count1[j]++;
printf("No replacement\n");
break;
}
}
if(flag==0&&count<frameno)
```

```
{
 frame[move]=page[i];
count1[move]=1;
move=(move+1)%frameno;
count++;
print(frameno,frame);
}
 else if(flag==0)
{
repindex=0;
leastcount=count1[0];
for(j=1;j<frameno;j++)
{
if(count1[j]<leastcount)
{
repindex=j;
leastcount=count1[j];
}
 }
frame[repindex]=page[i];
 count1[repindex]=1;
 count++;
print(frameno,frame);
  }
 }
rate=(float)count/(float)n;
printf("Number of page faults is %d\n",count);
 printf("Fault rate is %f\n",rate);
return 0;
}
```

OUTPUT :


deepak@deepak-Inspiron-5558: ~/os/page replacement
deepak@deepak-Inspiron-5558:~/os/page replacement$ gcc lfu.c
deepak@deepak-Inspiron-5558:~/os/page replacement$ ./a.out
Enter the number of pages
12
Enter the page reference numbers
0 2 1 6 4 0 1 0 3 1 2 1
Enter the number of frames
4
Page reference string Frames
0 0 -1 -1 -1
2 0 2 -1 -1
1 0 2 1 -1
6 0 2 1 6
4 4 2 1 6
0 0 2 1 6
1 No replacement
0 No replacement
3 0 3 1 6
1 No replacement
2 0 2 1 6
1 No replacement
Number of page faults is 8
Fault rate is 0.666667
deepak@deepak-Inspiron-5558:~/os/page replacement$

EXPERIMENT.NO 5
MEMORY ALLOCATION TECHNIQUES

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

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

PROGRAM

WORST-FIT

```
#include<stdio.h>
#include<conio.h>
#define max 25
void main()
{
        int
        frag[max],b[max],f[max],i,j,nb,nf,t
        emp; static int bf[max],ff[max];
        clrscr();
```

```
printf("\n\tMemory Management Scheme - First Fit");
printf("\nEnter the number of blocks:");
scanf("%d",&nb);
printf("Enter the number of files:");
scanf("%d",&nf);
printf("\nEnter the size of the blocks:-\n");
for(i=1;i<=nb;i++)
{
        printf("Block %d:",i);
        scanf("%d",&b[i]);
}
printf("Enter the size of the files :-\n");
for(i=1;i<=nf;i++)
{
        printf("File %d:",i);
        scanf("%d",&f[i]);
```

```
            }
            for(i=1;i<=nf;i++)
            {
                  for(j=1;j<=nb;j++)
                  {
                        if(bf[j]!=1)
                        {
                              temp=b[j]-
                              f[i];
                              if(temp>=0)
                              {
                                    ff[i]=j;
                                    break;
                              }
                        }
                  }
                  frag[i]=temp;
                  bf[ff[i]]=1;
            }
            printf("\nFile_no:\tFile_size :\tBlock_no:\tBlock_size:\tFragement");
            for(i=1;i<=nf;i++)
            printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d",i,f[i],ff[i],b[ff[i]],frag[i]);
            getch();
      }
```

INPUT

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

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

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

OUTPUT

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

BEST-FIT

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

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

INPUT

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

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

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

OUTPUT

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

FIRST-FIT

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

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

INPUT

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

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

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

OUTPUT

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

EXPERIMENT.NO.6

FILE ALLOCATION STRATEGIES

A) SEQUENTIAL:

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

DESCRIPTION:

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

SOURCE CODE :

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

OUTPUT:

Enter the starting block & length of file 4 10
4->1
5->1
6->1
7->1
8->1
9->1
10->1
11->1
12->1
13->1
The file is allocated to disk.

B) INDEXED:

AIM: To implement allocation method using chained method

DESCRIPTION:

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

SOURCE CODE :

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

```
if(c==1)
goto  x;
else
exit();
getch();
}
```

OUTPUT: enter index block 9

Enter no of files on index 3 1

2 3

Allocated

File indexed

9->1:1

9->2;1

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

C) LINKED:

AIM: To implement linked file allocation technique.

DESCRIPTION:

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

SOURCE CODE :

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

```
else
exit();
getch( );}
```

OUTPUT:

```
Enter how many blocks that are already allocated 3 Enter the blocks no.s
that are already allocated 4 7 Enter the starting index block & length 3 7 9
3->1
4->1 file is already allocated
5->1
6->1
7->1 file is already allocated
8->1
9->1file is already allocated
10->1
11->1
12->1
```

EXPERIMENT.NO 7

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

a) FCFS b) SSTF c) C-SCAN

DESCRIPTION

One of the responsibilities of the operating system is to use the hardware efficiently. For the disk drives, meeting this responsibility entails having fast access time and large disk bandwidth. Both the access time and the bandwidth can be improved by managing the order in which disk I/O requests are serviced which is called as disk scheduling. The simplest form of disk scheduling is, of course, the first-come, first-served (FCFS) algorithm. This algorithm is intrinsically fair, but it generally does not provide the fastest service.

C-SCAN is a variant of SCAN designed to provide a more uniform wait time. Like SCAN, C-SCAN moves the head from one end of the disk to the other, servicing requests along the way. When the head reaches the other end, however, it immediately returns to the beginning of the disk without servicing any requests on the return trip

PROGRAM

A) FCFS DISK SCHEDULING ALGORITHM

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

INPUT:

INPUT Enter no.of tracks:9

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

OUTPUT

Tracks traversed	Difference between tracks
55	45
58	3
60	2
70	10
18	52
90	72
150	60
160	10
184	24

Average header movements:30.888889

B)SSTF DISK SCHEDULING ALGORITHM

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

Output:

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

B) SCAN DISK SCHEDULING ALGORITHM

```
#include <stdio.h>
#include <stdlib.h>
#define LOW 0
#define HIGH 199
int main(){
  int queue[20];
  int head, max, q_size, temp, sum;
  int dloc; //location of disk (head) arr
  printf("%s\t", "Input no of disk locations");
  scanf("%d", &q_size);
  printf("%s\t", "Enter head position");
  scanf("%d", &head);
  printf("%s\n", "Input elements into disk queue");
  for(int i=0; i<q_size; i++){
    scanf("%d", &queue[i]);
  }
  queue[q_size] = head; //add RW head into queue
  q_size++;
  //sort the array
  for(int i=0; i<q_size;i++){
    for(int j=i; j<q_size; j++){
      if(queue[i]>queue[j]){
        temp = queue[i];
        queue[i] = queue[j];
        queue[j] = temp;
      }
    }
  }

  max = queue[q_size-1];
  //locate head in the queue
  for(int i=0; i<q_size; i++){
    if(head == queue[i]){
      dloc = i;
      break;
    }
  }

  if(abs(head-LOW) <= abs(head-HIGH)){
      for(int j=dloc; j>=0; j--){
        printf("%d --> ",queue[j]);
      }
```

```
        for(int j=dloc+1; j<q_size; j++){
          printf("%d --> ",queue[j]);
        }
        } else {
        for(int j=dloc+1; j<q_size; j++){
            printf("%d --> ",queue[j]);
        }
        for(int j=dloc; j>=0; j--){
            printf("%d --> ",queue[j]);
        }
      }
      sum  = head + max;
      printf("\nmovement of total cylinders %d", sum);
      return 0;
    }
```

Output:

```
Input no of disk locations       9
Enter head position           100
Input elements into disk queue
55       58       60       70       18       90       150      160      184
150 --> 160 --> 184 --> 100 --> 90 --> 70 --> 60 --> 58 --> 55 --> 18 -->
movement of total cylinders 284
```