EXPERIMENT NO.1

CPU SCHEDULINGALGORITHMS

A). FIRST COME FIRST SERVE:

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

DESCRIPTION:

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

ALGORITHM:

Step 1: Start the process
Step 2: Accept the number of processes in the ready Queue
Step 3: For each process in the ready Q, assign the process name and the burst time Step 4: Set the waiting of the first process as _0'and its burst time as its turnaround time Step 5: for each process in the Ready Q calculate
a). Waiting time (n) = waiting time (n-1) + Burst time (n-1) b).
Turnaround time (n)= waiting time(n)+Burst time(n)
Step 6: Calculate
a) Average waiting time = Total waiting Time / Number of process

b) Average Turnaround time = Total Turnaround Time / Number of process

Step 7: Stop the process

SOURCE CODE:

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

B). SHORTEST JOB FIRST:

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

DESCRIPTION:

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

ALGORITHM:

Step 1: Start the process
Step 2: Accept the number of processes in the ready Queue
Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time
Step 4: Start the Ready Q according the shortest Burst time by sorting according to lowest to highest burst time.
Step 5: Set the waiting time of the first process as _0' and its turnaround time as its burst time.
Step 6: Sort the processes names based on their Burt time
Step 7: For each process in the ready queue,
calculate
a) Waiting time(n)= waiting time (n-1) + Burst time (n-1)
b) Turnaround time (n)= waiting time(n)+Burst time(n)
Step 8: Calculate
c) Average waiting time = Total waiting Time / Number of process
d) Average Turnaround time = Total Turnaround Time / Number of process
Step 9: Stop the process

SOURCE CODE :

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

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

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

INPUT

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

OUTPUT

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

C). ROUND ROBIN:

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

DESCRIPTION:

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

ALGORITHM:

Step 1: Start the process

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

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

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

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

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

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

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

Step 7: Calculate

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

d) Average Turnaround time = Total Turnaround Time / Number of process Step 8: Stop the process

SOURCE CODE

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

INPUT:

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

OUTPUT:

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

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

D). PRIORITY:

AIM: To write a c program to simulate the CPU scheduling priorityalgorithm.

DESCRIPTION:

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

ALGORITHM:

Step 1: Start the process
Step 2: Accept the number of processes in the ready Queue
Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time
Step 4: Sort the ready queue according to the priority number.
Step 5: Set the waiting of the first process as _0' and its burst time as its turnaround time
Step 6: Arrange the processes based on process priority
Step 7: For each process in the Ready Q calculate Step 8:
for each process in the Ready Q calculate

a) Waiting time(n)= waiting time (n-1) + Burst time (n-1)
b) Turnaround time (n)= waiting time(n)+Burst time(n)

Step 9: Calculate

c) Average waiting time = Total waiting Time / Number of process
d) Average Turnaround time = Total Turnaround Time / Number of process Print the results in an order.

Step10: Stop

SOURCE CODE:

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

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

INPUT

Enter the number of processes -- 5
Enter the Burst Time & Priority of Process 0 --- 10 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

VIVA QUESTIONS

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

EXPERIMENT.NO 4
MEMORY MANAGEMENT

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

AIM: To implement and simulate the MFT algorithm.

DESCRIPTION:

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

ALGORITHM:

Step1: Start the process.
Step2: Declarevariables.
Step3: Enter total memory size ms.
Step4: Allocate memory for os.
Ms=ms-os
Step5: Read the no partition to be divided n Partition size=ms/n.
Step6: Read the process no and process size.
Step 7: If process size is less than partition size allot alse blocke the process. While allocating update memory wastage-external fragmentation.

```
if(pn[i]==pn[j])f=1;
if(f==0){ if(ps[i]<=siz)
{
extft=extft+size-
ps[i];avail[i]=1; count++;
}
}
```

Step 8: Print the results

SOURCE CODE :

```
#include<stdio.h>
#include<conio.h>
main()
{
int ms, bs, nob, ef,n,
mp[10],tif=0; int i,p=0;
clrscr();
printf("Enter the total memory available (in Bytes) -- ");
scanf("%d",&ms);
printf("Enter the block size (in Bytes) -- ");
scanf("%d", &bs);
nob=ms/bs;
ef=ms - nob*bs;
printf("\nEnter the number of processes -- ");
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("Enter memory required for process %d (in Bytes)-- ",i+1);
scanf("%d",&mp[i]);
}
printf("\nNo. of Blocks available in memory--%d",nob);
printf("\n\nPROCESS\tMEMORYREQUIRED\tALLOCATED\tINTERNAL
FRAGMENTATION");
for(i=0;i<n && p<nob;i++)
{
printf("\n %d\t\t%d",i+1,mp[i]);
if(mp[i] > bs)
printf("\t\tNO\t\t---");
else
{
printf("\t\tYES\t%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

ALGORITHM:

Step1: start the process.
Step2: Declare variables.
Step3: Enter total memory size ms.
Step4: Allocate memory for os.
Ms=ms-os
Step5: Read the no partition to be divided n Partition size=ms/n.
Step6: Read the process no and process size.
Step 7: If process size is less than partition size allot alse blocke the process. While allocating update memory wastage-external fragmentation.

```
if(pn[i]==pn[j])        f=1;
if(f==0){ if(ps[i]<=size)
{
extft=extft+size-
ps[i];avail[i]=1; count++;
}
}
```

Step 8: Print the results
Step 9: Stop the process.

SOURCE CODE:

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

VIVA QUESTIONS

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

EXPERIMENT NO. 7

FILE ORGANIZATION TECHNIQUES

A) SINGLE LEVEL DIRECTORY:

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

DESCRIPTION:

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

SOURCE CODE :

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

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

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

EXPERIMENT.NO.8

FILE ALLOCATION STRATEGIES

A) SEQUENTIAL:

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

DESCRIPTION:

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

ALGORITHM:

Step 1: Start the program.
Step 2: Get the number of files.
Step 3: Get the memory requirement of each file.
Step 4: Allocate the required locations to each in sequential order a).
Randomly select a location from available location s1= random(100);

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

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

b) Allocate and set flag=1 to the allocated locations.

```
for(s=s1;s<(s1+p[i]);s++)
{
k[i][j]=s; j=j+1; b[s].bno=s;
b[s].flag=1;
}
```

Step 5: Print the results file no, length, Blocks allocated. Step 6: Stop the program

SOURCE CODE :

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

OUTPUT:

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

B) INDEXED:

AIM: To implement allocation method using chained method

DESCRIPTION:

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

ALGORITHM:

Step 1: Start the program.
Step 2: Get the number of files.
Step 3: Get the memory requirement of each file.
Step 4: Allocate the required locations by selecting a location randomly q= random(100);

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

```
q=random(100);
{
if(b[q].flag==0)
b[q].flag=1;
b[q].fno=j;
r[i][j]=q;
```

Step 5: Print the results file no, length ,Blocks allocated.
Step 6: Stop the program

SOURCE CODE :

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

OUTPUT: enter index block 9

Enter no of files on index 3 1

2 3

Allocated

File indexed

9->1:1

9->2;1

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

C) LINKED:

AIM: To implement linked file allocation technique.

DESCRIPTION:

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

ALGORTHIM:

Step 1: Start the program.

Step 2: Get the number of files.

Step 3: Get the memory requirement of each file.

Step 4: Allocate the required locations by selecting a location randomly q= random(100);

a) Check whether the selected location is free .

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

While allocating next location address to attach it to previous location

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

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

Step 6: Stop the program

SOURCE CODE :

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

OUTPUT:

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

VIVA QUESTIONS

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