Gokaraju Rangaraju Institute of Engineering and Technology

OPERATING SYSTEMS & SCI LAB

LAB MANUAL

Academic Year: 2018-19

SEMESTER-II

Prepared By:

V.Shailaja

Assistant Professor

Dept of IT

Under the Guidance of:

Dr.K.PrasannaLakshmi

Professor & HOD

Dept of IT

Gokaraju Rangaraju Institute of Engineering and Technology

(Autonomous)

OPERATING SYSTEMS & SCI LAB

PART I

Objective:

To understand the operating System functionalities System/ Software Requirement

1. Simulate the following CPU scheduling algorithms a) Round Robin b) SJF c) FCFS d) Priority
2. Simulate all file allocation strategies a) Sequential b) Indexed c) Linked
3. Simulate MVT and MFT
4. Simulate all File Organization Techniques a) Single level directory b) Two level directory
5. Simulate all page replacement algorithms a) FIFO b) LRU c) LFU
6. Simulate Paging Technique of memory management.

PART II

To understand Scilab environment and programming

1. Scilab environment
2. The Workspace and Working Directory
3. Experiment 3 – Matrix Operations
4. Experiment 4 – Sub-matrices
5. Experiment 5 – Statistics
6. Experiment 6 – Plotting Graphs
7. Experiment 7 – Plotting 3D Graphs
8. Experiment 8 – Scilab Programming Language
9. Experiment 9 – Script Files and Function Files
10. Experiment 10 – Functions in Scilab
11. Experiment 11 – File Operations
12. Experiment 12 – Reading Microsoft Excel File

OPERATING SYSTEMS & SCI LAB

Week1:

Simulate the following CPU scheduling algorithms

a) Round Robin

b) SJF

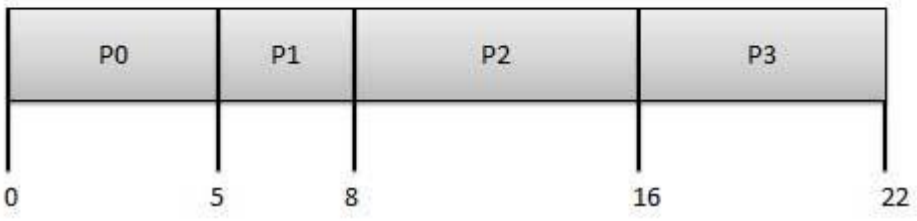
c) FCFS

d) Priority

First Come First Serve (FCFS)

- Jobs are executed on first come, first serve basis.
- It is a non-preemptive, pre-emptive scheduling algorithm.
- Easy to understand and implement.
- Its implementation is based on FIFO queue.
- Poor in performance as average wait time is high.

Process	Arrival Time	Execute Time	Service Time
P0	0	5	0
P1	1	3	5
P2	2	8	8
P3	3	6	16



Wait time of each process is as follows −

Process	Wait Time : Service Time - Arrival Time

P0	0 - 0 = 0
P1	5 - 1 = 4
P2	8 - 2 = 6
P3	16 - 3 = 13

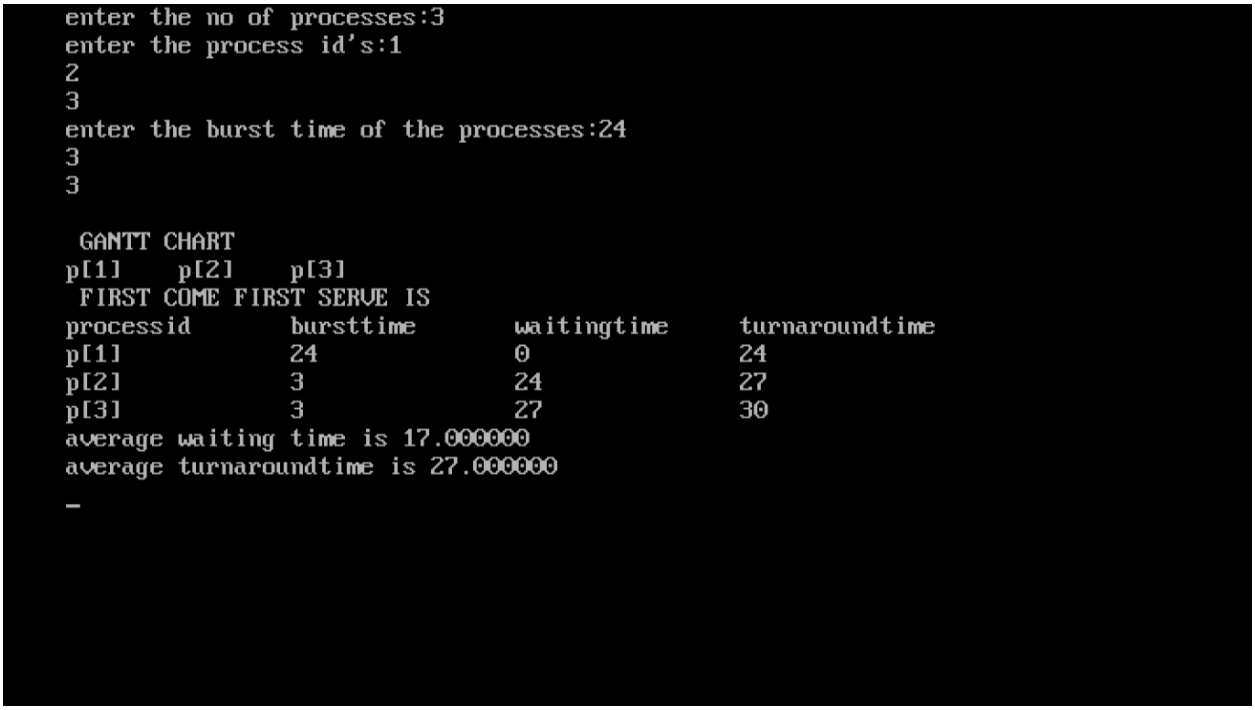
Average Wait Time: (0+4+6+13) / 4 = 5.75

FCFS Program:

```
#include<stdio.h>
#include<conio.h>
void main()
{
 int n,i,p[10],bt[10],w[10],t[10],wt=0,tat=0;
 float awt,atat;
 clrscr();
 printf("enter the no of processes:");
 scanf("%d",&n);
 printf("enter the process id's:");
 for(i=0;i<n;i++)
 scanf("%d",&p[i]);
 printf("enter the burst time of the processes:");
 for(i=0;i<n;i++)
 scanf("%d",&bt[i]);
```

```
w[0]=0;
for(i=1;i<n;i++)
w[i]=w[i-1]+bt[i-1];
for(i=0;i<n;i++)
wt=wt+w[i];
awt=(wt)/n;
t[0]=bt[0];
for(i=1;i<n;i++)
t[i]=w[i]+bt[i];
for(i=0;i<n;i++)
tat=tat+t[i];
atat=(tat)/n;
printf("\n GANTT CHART \n");
for(i=0;i<n;i++)
{
  printf("p[%d]\t",p[i]);
}
printf(" \n FIRST COME FIRST SERVE IS \n");
printf("processid\tbursttime\twaitingtime\tturnaroundtime\n");
for(i=0;i<n;i++)
{
 printf("p[%d]\t\t%d\t\t%d\t\t%d\t\n",p[i],bt[i],w[i],t[i]);
}
printf("average waiting time is %f\n",awt);
printf("average turnaroundtime is %f\n",atat);
getch();
}
```

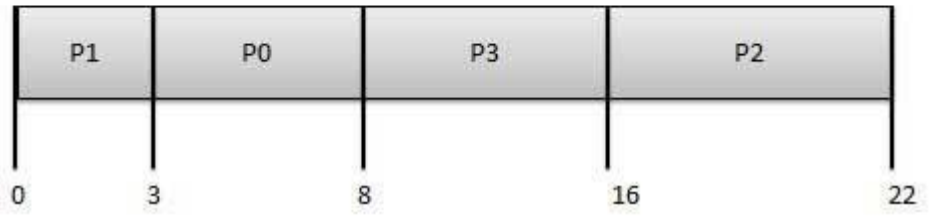
Output:


enter the no of processes:3
enter the process id's:1
2
3
enter the burst time of the processes:24
3
3
GANTT CHART
p[1] p[2] p[3]
FIRST COME FIRST SERVE IS
processid bursttime waitingtime turnaroundtime
p[1] 24 0 24
p[2] 3 24 27
p[3] 3 27 30
average waiting time is 17.000000
average turnaroundtime is 27.000000

Shortest Job First (SJF)

- This is also known as shortest job first, or SJF
- This is a non-preemptive, pre-emptive scheduling algorithm.
- Best approach to minimize waiting time.
- Easy to implement in Batch systems where required CPU time is known in advance.
- Impossible to implement in interactive systems where required CPU time is not known the processer should know in advance how much time process will take.

Process	Arrival Time	Execute Time	Service Time
P0	0	5	3
P1	1	3	0
P2	2	8	16
P3	3	6	8



Wait time of each process is as follows −

Process	Wait Time : Service Time - Arrival Time
P0	3 - 0 = 3
P1	0 - 0 = 0
P2	16 - 2 = 14
P3	8 - 3 = 5

Average Wait Time: (3+0+14+5) / 4 = 5.50

SJF Program:

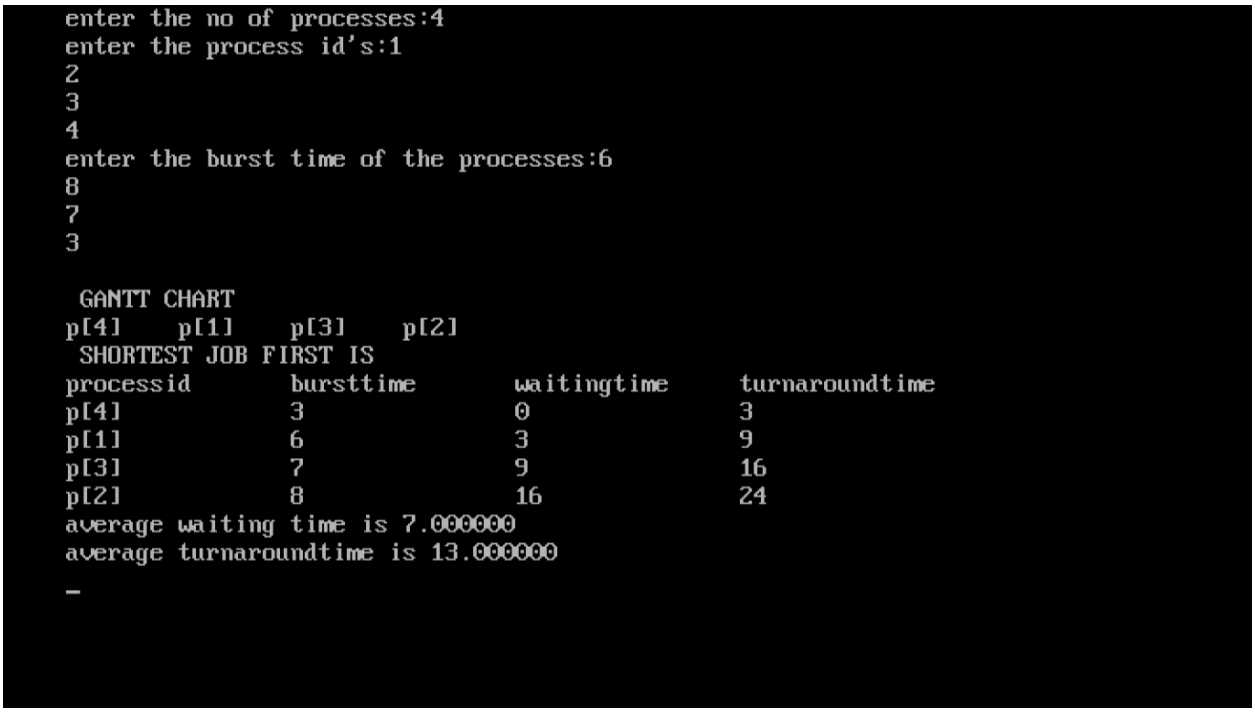
```
#include<stdio.h>

#include<conio.h>
```

```
void main()
{
 int n,i,j,p[10],bt[10],w[10],t[10],wt=0,tat=0,t1,t2;
 float awt,atat;
 clrscr();
 printf("enter the no of processes:");
 scanf("%d",&n);
 printf("enter the process id's:");
 for(i=0;i<n;i++)
 scanf("%d",&p[i]);
 printf("enter the burst time of the processes:");
 for(i=0;i<n;i++)
 scanf("%d",&bt[i]);
 for(i=0;i<n;i++)
 {
 for(j=i+1;j<n;j++)
 {
  if(bt[i]>bt[j])
  {
  t1=bt[i];
  bt[i]=bt[j];
  bt[j]=t1;

  t2=p[i];
  p[i]=p[j];
  p[j]=t2;
  }
```

```
 }
}
w[0]=0;
 for(i=1;i<n;i++)
 w[i]=w[i-1]+bt[i-1];
 for(i=0;i<n;i++)
 wt=wt+w[i];
 awt=(float)(wt)/n;
 t[0]=bt[0];
 for(i=1;i<n;i++)
 t[i]=w[i]+bt[i];
 for(i=0;i<n;i++)
 tat=tat+t[i];
 atat=(float)(tat)/n;
 printf("\n GANTT CHART \n");
 for(i=0;i<n;i++)
 {
   printf("p[%d]\t",p[i]);
 }
 printf(" \n SHORTEST JOB FIRST IS \n");
 printf("processid\tbursttime\twaitingtime\tturnaroundtime\n");
 for(i=0;i<n;i++)
 {
  printf("p[%d]\t\t%d\t\t%d\t\t%d\t\n",p[i],bt[i],w[i],t[i]);
 }
 printf("average waiting time is %f\n",awt);
 printf("average turnaroundtime is %f\n",atat);
```

```
getch();
}
```

Output:

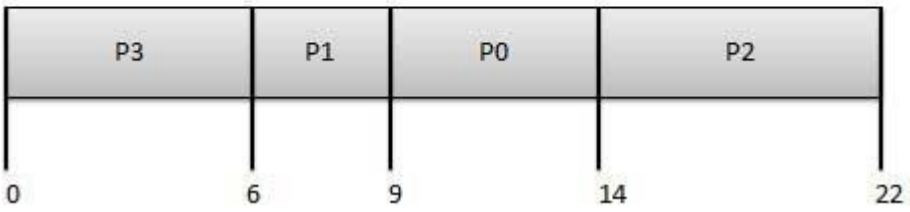


Priority Based Scheduling

Priority scheduling is a non-preemptive algorithm and one of the most common scheduling algorithms in batch systems.Each process is assigned a priority.

Process with highest priority is to be executed first and so on.Processes with same priority are executed on first come first served basis.Priority can be decided based on memory requirements, time requirements or any other resource requirement.

Process	Arrival Time	Execute Time	Priority	Service Time
P0	0	5	1	9
P1	1	3	2	6
P2	2	8	1	14
P3	3	6	3	0



Wait time of each process is as follows −

Process	Wait Time : Service Time - Arrival Time
P0	9 - 0 = 9
P1	6 - 1 = 5
P2	14 - 2 = 12
P3	0 - 0 = 0

Average Wait Time: (9+5+12+0) / 4 = 6.5

Priority Program:

```
#include<stdio.h>
#include<conio.h>
void main()
{
 int n,i,j,p[10],bt[10],pr[10],w[10],t[10],wt=0,tat=0,t1,t2,t3;
 float awt=0,atat=0;
 clrscr();
 printf("enter the no of processes:");
 scanf("%d",&n);
 printf("enter the process id's:");
 for(i=0;i<n;i++)
 scanf("%d",&p[i]);
 printf("enter the burst time of the processes:");
 for(i=0;i<n;i++)
 scanf("%d",&bt[i]);
 printf("enter the priority of the processes:");
 for(i=0;i<n;i++)
 scanf("%d",&pr[i]);
 for(i=0;i<n;i++)
 {
 for(j=i+1;j<n;j++)
 {
  if(pr[i]>pr[j])
  {
  t1=pr[i];
  pr[i]=pr[j];
```

```
 pr[j]=t1;

 t2=bt[i];
 bt[i]=bt[j];
 bt[j]=t2;

 t3=p[i];
 p[i]=p[j];
 p[j]=t3;
 }
 }
}
w[0]=0;
 for(i=1;i<n;i++)
 w[i]=w[i-1]+bt[i-1];
 for(i=0;i<n;i++)
 wt=wt+w[i];
 awt=(float)(wt)/n;
 t[0]=bt[0];
 for(i=1;i<n;i++)
 t[i]=w[i]+bt[i];
 for(i=0;i<n;i++)
 tat=tat+t[i];
 atat=(float)(tat)/n;
 printf("\n GANTT CHART \n");
 for(i=0;i<n;i++)
 {
```

```
  printf("p[%d]\t",p[i]);
 }
 printf(" \n PRIORITY IS \n");
 printf("processid\tbursttime\tpriority\twaitingtime\tturnaroundtime\n");
 for(i=0;i<n;i++)
 {
  printf("p[%d]\t\t%d\t\t%d\t\t%d\t\t%d\t\n",p[i],bt[i],pr[i],w[i],t[i]);
 }
 printf("average waiting time is %f\n",awt);
 printf("average turnaroundtime is %f\n",atat);
 getch();
}
```

Output:

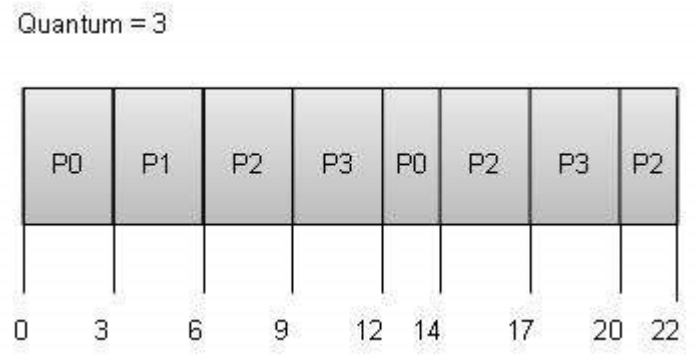
```
4
5
enter the burst time of the processes:10
1
2
1
5
enter the priority of the processes:3
1
4
5
2

 GANTT CHART
p[2]    p[5]    p[1]    p[3]    p[4]
 PRIORITY IS
processid       bursttime       priority        waitingtime     turnaroundtime
p[2]            1               1               0               1
p[5]            5               2               1               6
p[1]            10              3               6               16
p[3]            2               4               16              18
p[4]            1               5               18              19
average waiting time is 8.200000
average turnaroundtime is 12.000000
```

Round Robin Scheduling

Round Robin is the preemptive process scheduling algorithm.

Each process is provided a fix time to execute, it is called a quantum.

Once a process is executed for a given time period, it is preempted and other process executes for a given time period.Context switching is used to save states of preempted processes.



Wait time of each process is as follows −

Process	Wait Time : Service Time - Arrival Time
P0	(0 - 0) + (12 - 3) = 9
P1	(3 - 1) = 2
P2	(6 - 2) + (14 - 9) + (20 - 17) = 12
P3	(9 - 3) + (17 - 12) = 11

Average Wait Time: (9+2+12+11) / 4 = 8.5

RoundRobin Program:

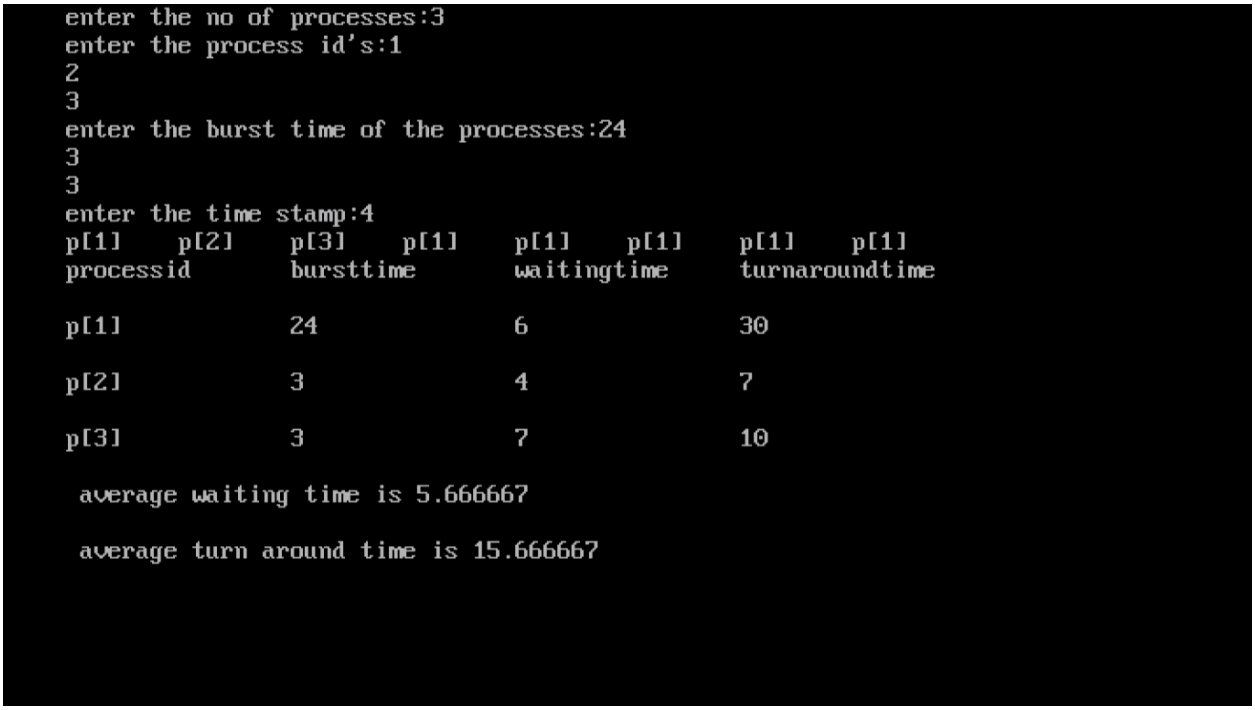
```
#include<stdio.h>

#include<conio.h>

void main()

{

 int n,p[10],bt[10],cbt[10],i,sum=0,ts,temp=0,tat[10],wt[10];
```

```
float awt=0,atat=0;
clrscr();
printf("enter the no of processes:");
scanf("%d",&n);
printf("enter the process id's:");
for(i=0;i<n;i++)
scanf("%d",&p[i]);
printf("enter the burst time of the processes:");
for(i=0;i<n;i++)
{
scanf("%d",&bt[i]);
cbt[i]=bt[i];
}
printf("enter the time stamp:");
scanf("%d",&ts);
for(i=0;i<n;i++)
{
 sum=sum+bt[i];
}
while(sum>0)
{
for(i=0;i<n;i++)
{
if(bt[i]>ts)
{
temp=temp+ts;
tat[i]=temp;
```

```
bt[i]=bt[i]-ts;
sum=sum-ts;
printf("p[%d]\t",p[i]);
}
else if(bt[i]<=ts&&bt[i]>0)
{
temp=temp+bt[i];
tat[i]=temp;
sum=sum-bt[i];
bt[i]=0;
printf("p[%d]\t",p[i]);
}
else
continue;
}
}
for(i=0;i<n;i++)
wt[i]=tat[i]-cbt[i];
printf("\nprocessid\tbursttime\twaitingtime\tturnaroundtime\n");
for(i=0;i<n;i++)
{
printf("\np[%d]\t\t%d\t\t%d\t\t%d\n",p[i],cbt[i],wt[i],tat[i]);
awt=awt+wt[i];
atat=atat+tat[i];
}
printf("\n average waiting time is %f\n",awt/n);
printf("\n average turn around time is %f\n",atat/n);
```

```
getch();

}
```

Output:



Week 2

Simulate all file allocation strategies

a) Sequential

b) Indexed

c) Linked

File Access Mechanisms

File access mechanism refers to the manner in which the records of a file may be accessed. There are several ways to access files −

- Sequential access
- Direct/Random access
- Indexed sequential access

Sequential access

A sequential access is that in which the records are accessed in some sequence, i.e., the information in the file is processed in order, one record after the other. This access method is the most primitive one. Example: Compilers usually access files in this fashion.

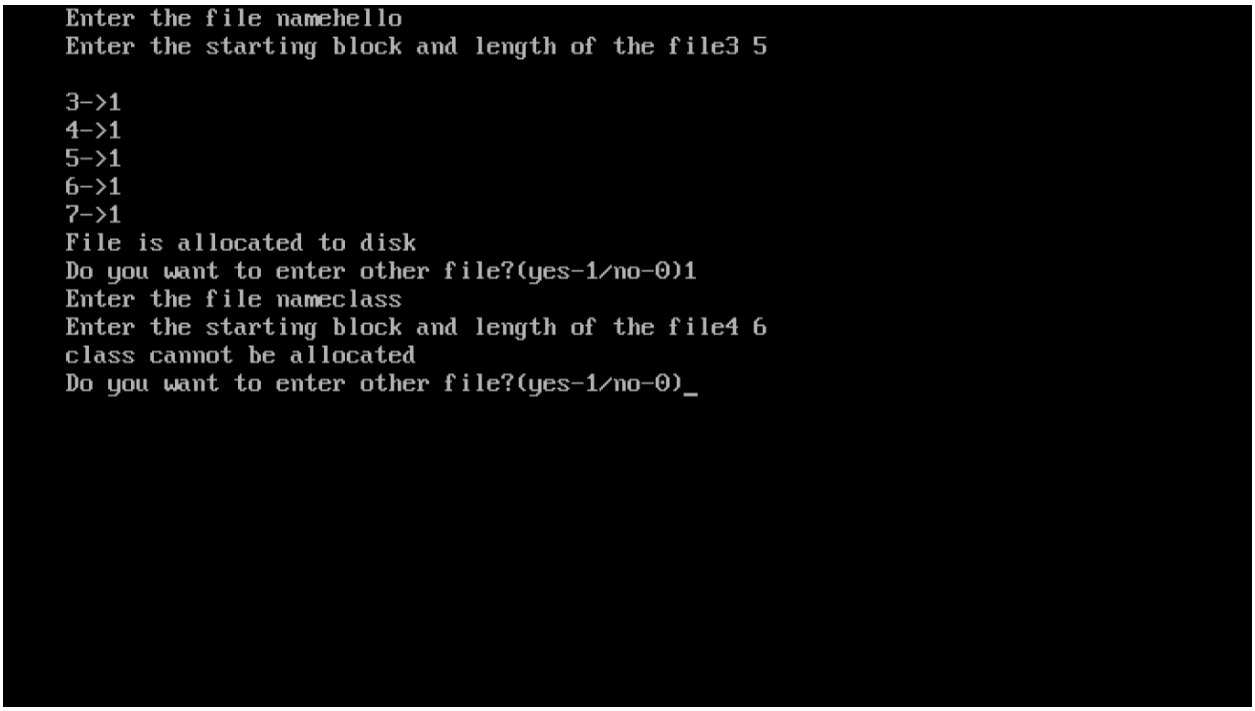
Sequential access Program:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int f[50],i,st,j,len,c,k=0;
char s[50];
clrscr();
for(i=0;i<50;i++)
f[i]=0;
while(1)
{
```

```
printf("Enter the file name");
scanf("%s",s);
printf("Enter the starting block and length of the file");
scanf("%d%d",&st,&len);
for(j=st;j<(st+len);j++)
{
if(f[j]==1)
{
printf("%s cannot be allocated",s);
k=1;
break;
}
}
if (k==0)
{
for(j=st;j<(st+len);j++)
{
f[j]=1;
printf("\n%d->%d",j,f[j]);
}
if(j==(st+len))
printf("\nFile is allocated to disk");
}
k=0;
printf("\nDo you want to enter other file?(yes-1/no-0)");
scanf("%d",&c);
if(c==0)
```

```
break;
}
getch();
}
```

Output:



Indexed sequential access

This mechanism is built up on base of sequential access.An index is created for each file which contains pointers to various blocks.Index is searched sequentially and its pointer is used to access the file directly.

Indexed sequential access Program:

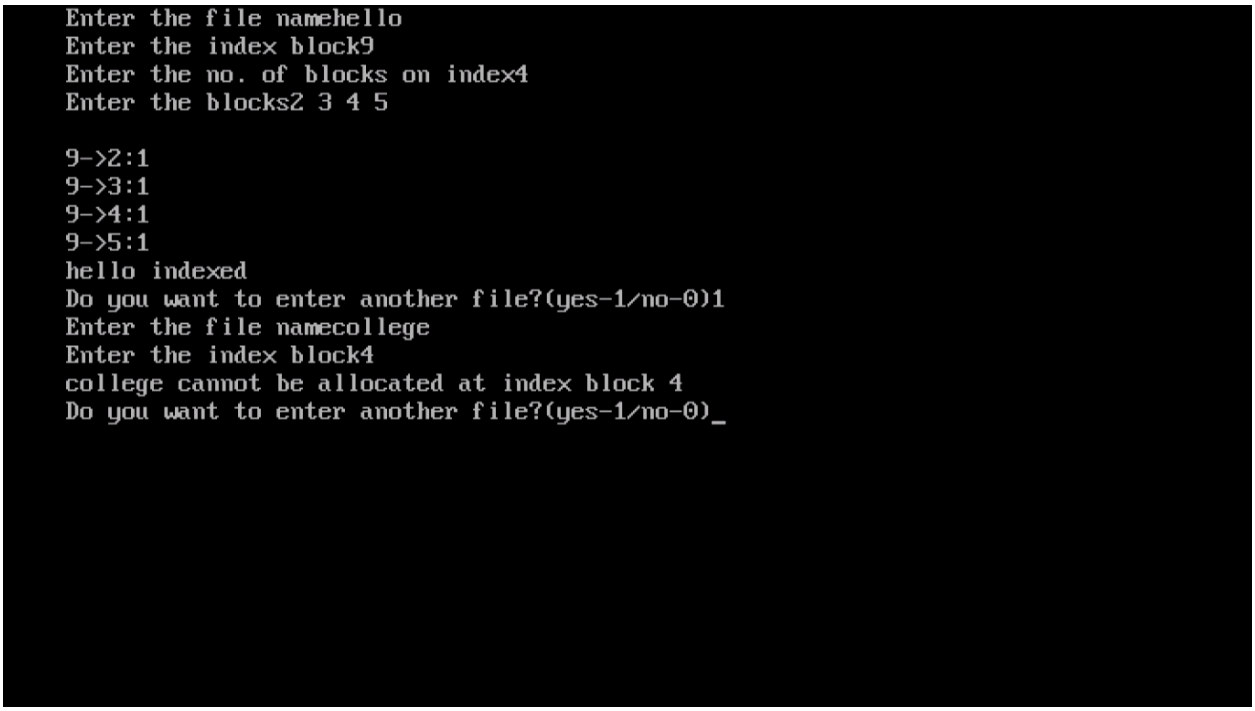
```
#include<stdio.h>
#include<conio.h>
int main()
{
 int f[50],i,j,k=0,index[50],n,c,p;
 char s[20];
 clrscr();
 for(i=0;i<50;i++)
 f[i]=0;
 while(1)
 {
  printf("Enter the file name");
  scanf("%s",s);
  printf("Enter the index block");
  scanf("%d",&p);
  if(f[p]==1)
  {
   printf("%s cannot be allocated at index block %d",s,p);
  }
  else
  {
   f[p]=1;
   printf("Enter the no. of blocks on index");
```

```
    scanf("%d",&n);
    printf("Enter the blocks");
    for(i=0;i<n;i++)
    scanf("%d",&index[i]);
    for(i=0;i<n;i++)
    if(f[index[i]]==1)
    {
    printf("\nBlock cannot be allocated");
    k=1;
    break;
    }
    if(k==0)
    {
    for(i=0;i<n;i++)
    {
    f[index[i]]=1;
     printf("\n%d->%d:%d",p,index[i],f[index[i]]);
     }
    printf("\n%s indexed",s);
    }
    }
    printf("\nDo you want to enter another file?(yes-1/no-0)");
    scanf("%d",&c);
    if(c==0)
    break;
  }
//  getch();
```

return 0;

}

Output:

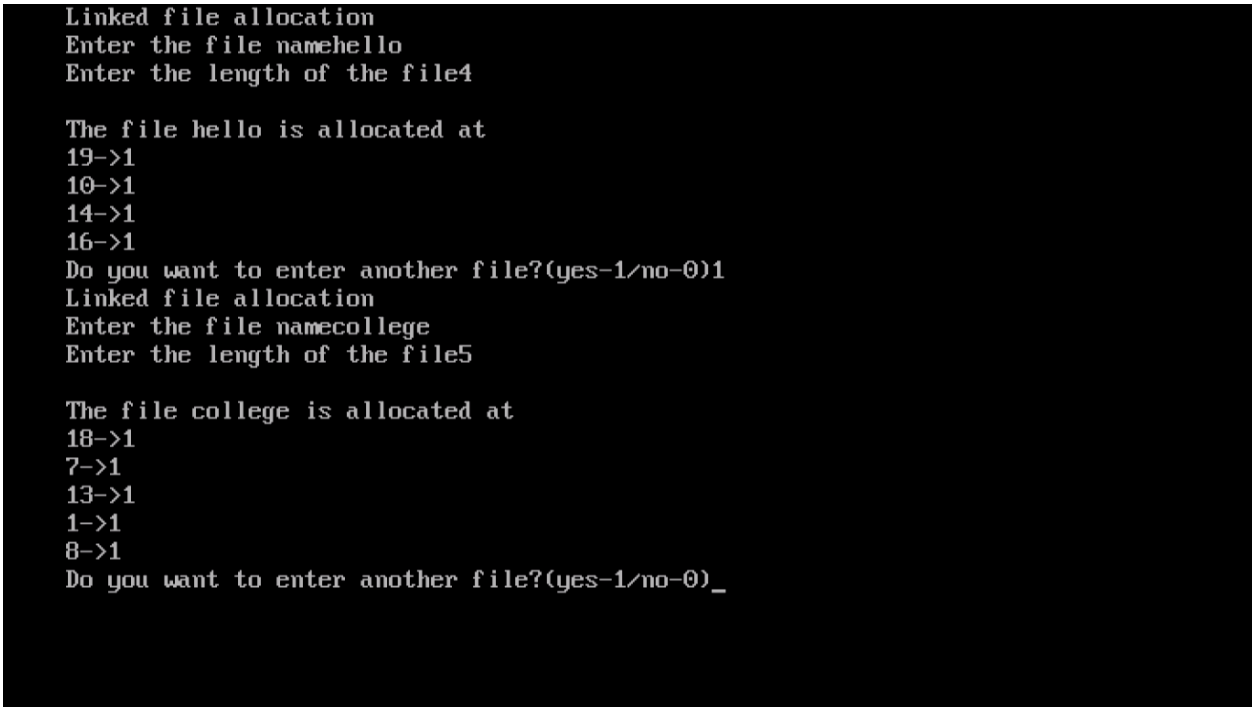


Linked Allocation Program:

```
#include<conio.h>
#include<math.h>
#include<stdlib.h>
int main()
{
 int f[20],i,n,st,len,c;
 char s[20];
 clrscr();
 n=20;
 for(i=0;i<20;i++)
 f[i]=0;
 randomize();
 while(1)
 {
 printf("Linked file allocation\n");
 printf("Enter the file name");
 scanf("%s",s);
 printf("Enter the length of the file");
   scanf("%d",&len);
   printf("\nThe file %s is allocated at",s);
   for(i=0;i<len;)
   {
     st=random(n);
     if(f[st]==0)
     {
     f[st]=1;
```

```
    i++;
    printf("\n%d->%d",st,f[st]);
    }
  }
  printf("\nDo you want to enter another file?(yes-1/no-0)");
  scanf("%d",&c);
  if(c==0)
  break;
  }
  getch();
return 0;
 }
```

Output:



Week 3:

Simulate MVT and MFT

MFT and MVT are different memory management techniques in operating systems.

MFT or fixed partitioning schemeThe OS ispartitioned into fixed sized blocks at the time of installation. For example,

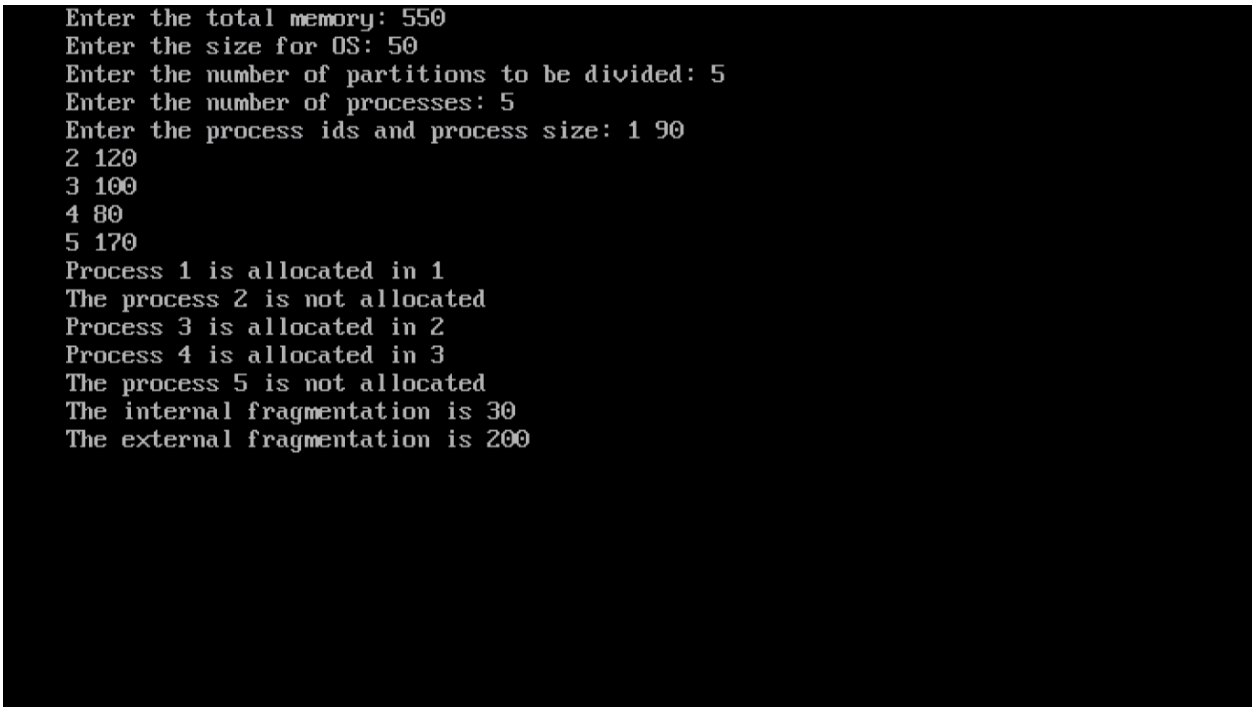
- There can be total 4 partitions and the size of each block can be 4KB. Then the
- processes which require 4KB or less memory will only get the memory.
- It is possibleto bind address at the time of compilation.
- It is notflexible because the number of blocks cannot be changed.
- There can bememory wastage due to fragmentation.

MFT Program:

```
#include<stdio.h>

#include<conio.h>

void main()

{

int ms,s,n,i,par,ps[20],p[20],size,intr=0,extr=0,k,par1,j=1;

clrscr();

printf("Enter the total memory: ");

scanf("%d",&ms);

printf("Enter the size for OS: ");

scanf("%d",&s);

ms=ms-s;

printf("Enter the number of partitions to be divided: ");

scanf("%d",&par);

printf("Enter the number of processes: ");

scanf("%d",&n);

printf("Enter the process ids and process size: ");

for(i=0;i<n;i++)
```

```
scanf("%d %d",&p[i],&ps[i]);
size=ms/par;
par1=par;
for(i=0;i<n;i++)
{
if((ps[i]<=size))
{
intr=intr+size-ps[i];
printf("Process %d is allocated in %d\n",i+1,j);
j++;
}
else
printf("The process %d is not allocated\n",i+1);
}
for(k=j;k<=par1;k++)
extr=extr+size;
printf("The internal fragmentation is %d\n",intr);
printf("The external fragmentation is %d",extr);
getch();
}
```

Output:


Enter the total memory: 550
Enter the size for OS: 50
Enter the number of partitions to be divided: 5
Enter the number of processes: 5
Enter the process ids and process size: 1 90
2 120
3 100
4 80
5 170
Process 1 is allocated in 1
The process 2 is not allocated
Process 3 is allocated in 2
Process 4 is allocated in 3
The process 5 is not allocated
The internal fragmentation is 30
The external fragmentation is 200

MVT:

- MVT or variable partitioning scheme ,No partitioning is done at the beginning.
- Memory is given to the processes as they come.
- This method is more flexible.
- Variable size of memory can be given as there is no size limitation.
- There is no internal fragmentation. But there can be external fragmentation.
- Compile time address binding cannot be done.

MVT Program:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int m,n,pp[50],ch,par[10],visited[10],mp[10],i,j,extr=0,intr=0,min,max,dec[10],k=0,t=0;
clrscr();
printf("enter the number of partitions\n");
scanf("%d",&n);
printf("enter the number of process\n");
scanf("%d",&m);
printf("enter the size of partitions\n");
for(i=0;i<n;i++)
scanf("%d",&mp[i]);
printf("enter the size of processes\n");
for(i=0;i<n;i++)
scanf("%d",&pp[i]);
do
{
for(i=0;i<n;i++)
{
```

```
visited[i]=0;
par[i]=-1;
intr=0;
extr=0;
}
printf("\nenter your choice\n 1.first fit\n 2.best fit \n 3.worst fit\n 4.exit\n");
scanf("%d",&ch);
switch(ch)
{
case 1: for(j=0;j<m;j++)
 {
 for(i=0;i<n;i++)
 {
 if(visited[i]==0)
 {
 if(mp[i]>=pp[j])
 {
 printf("process %d is allocated\n",j+1);
 intr=intr+(mp[i]-pp[j]);
 printf("\t\tinternal fragmentation in process%d is %d\n",j+1,mp[i]-pp[j]);
 par[i]=j+1;
 visited[i]=j+1;
 break;
 }
 }
 }
 if(i==n)
```

```
 printf("process %d cannot be allocated\n",j+1);
 }
 break;
case 2:for(j=0;j<m;j++)
{
for(i=0;i<n;i++)
dec[i]=mp[i]-pp[j];
min=1000;
k=-1;t=-1;
for(i=0;i<n;i++)
{
if(visited[i]==0)
{
if(dec[i]<min&&dec[i]>=0)
{
t=1;
min=dec[i];
k=i;
}
}
}
if(t==1)
{
printf("process %d is allocated at%d\n",j+1, mp[k]);
par[k]=j+1;
visited[k]=1;
printf("\t\tinternal fragmentation is %d\n",min);
```

```
intr=intr+min;
}
else
printf("process %d cannot be allocated\n",j+1);
}  //j loop is going to get incremented before break;
break;
case 3:for(j=0;j<m;j++)
{
for(i=0;i<n;i++)
dec[i]=mp[i]-pp[j];
max=-1;
k=-1;t=-1;
for(i=0;i<n;i++)
if(visited[i]==0)
if(dec[i]>max&&dec[i]>0)
{
t=1;
max=dec[i];
k=i;
}
if(t==1)
{
printf("process %d is allocated at%d\n",j+1, mp[k]);
par[k]=j+1;
visited[k]=1;
printf("\t\tinternal fragmentation is %d\n",max);
intr=intr+max;
```

```
}
else
printf("process %d cannot be allocated\n",j+1);
}
break;
case 4:exit(0);
break;
default:printf("enter a valid choice\n");
break;
  }
printf("totAL  internal fragmentation is %d\n",intr);
for(i=0;i<n;i++)
if(visited[i]==0)
extr=extr+mp[i];
printf("totAL  external fragmentation is %d\n",extr);
printf("the sequence of allocation is\n ");
for(i=0;i<n;i++)
printf("%d\t",par[i]);
}while(ch<=4);
getch();
}
```

Output:

```
enter the number of partitions
5
enter the number of process
5
enter the size of partitions
100
200
300
600
400
enter the size of processes
110
210
280
580
700

enter your choice
 1.first fit
 2.best fit
 3.worst fit
 4.exit
```

```
enter your choice
 1.first fit
 2.best fit
 3.worst fit
 4.exit
1
process 1 is allocated
                internal fragmentation in process1 is 90
process 2 is allocated
                internal fragmentation in process2 is 90
process 3 is allocated
                internal fragmentation in process3 is 320
process 4 cannot be allocated
process 5 cannot be allocated
totAL  internal fragmentation is 500
totAL  external fragmentation is 500
the sequence of allocation is
 -1     1       2       3       -1
enter your choice
 1.first fit
 2.best fit
 3.worst fit
 4.exit
```

```
enter your choice
 1.first fit
 2.best fit
 3.worst fit
 4.exit
3
process 1 is allocated at600
                internal fragmentation is 490
process 2 is allocated at400
                internal fragmentation is 190
process 3 is allocated at300
                internal fragmentation is 20
process 4 cannot be allocated
process 5 cannot be allocated
totAL  internal fragmentation is 700
totAL  external fragmentation is 300
the sequence of allocation is
 -1     -1      3       1       2
enter your choice
 1.first fit
 2.best fit
 3.worst fit
 4.exit
```

Week-4:

Simulate all File Organization Techniques

a) Single level directory

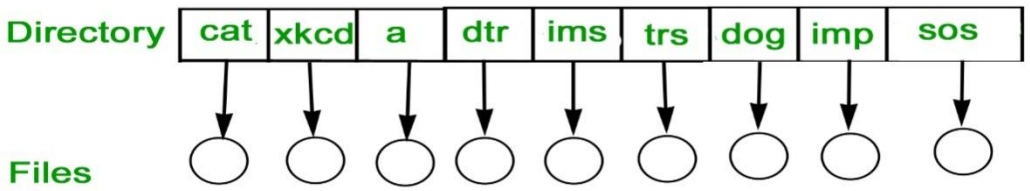
b) Two level directory

Single Level Directory

In this a single directory is maintained for all the users.

Naming problem: Users cannot have same name for two files.

Grouping problem: Users cannot group files according to their need.



Single Level Directory Program:

```
#include<stdio.h>
#include<conio.h>
#define M 30
struct directory
{
char dname[M];
int nf;
struct names
{
char fname[M];
}n[M];
}d[M];
```

```
void main()
{
int nd,i,j,k;
char fn[M];
clrscr();
printf("enter the no of directories: ");
scanf("%d",&nd);
for(i=0;i<nd;i++)
{
printf("enter the name of the directory %d: ",i+1);
scanf("%s",&d[i].dname);
}
for(i=0;i<nd;i++)
{
printf("enter the no of file in directory %d: ",i+1);
scanf("%d",&d[i].nf);
}
for(i=0;i<nd;i++)
{
for(j=0;j<d[i].nf;j++)
{
printf("enter the name of the file %d of directory %d: ",j+1,i+1);
scanf("%s",&fn);
for(k=0;k<d[i].nf;k++)
if(strcmp(fn,d[i].n[k].fname)==0)
{
j--;
```

```
printf("filename already exist.re-enter the2 file name\n");
goto x;
}
strcpy(d[i].n[j].fname,fn);
x:
}
printf("\n");
}
textcolor(30);
for(i=0;i<nd;i++)
{
cprintf("%s",d[i].dname);
printf("\t\t");
for(j=0;j<d[i].nf;j++)
printf("%s\t",d[i].n[j].fname);
printf("\n");
}
getch();
}
```

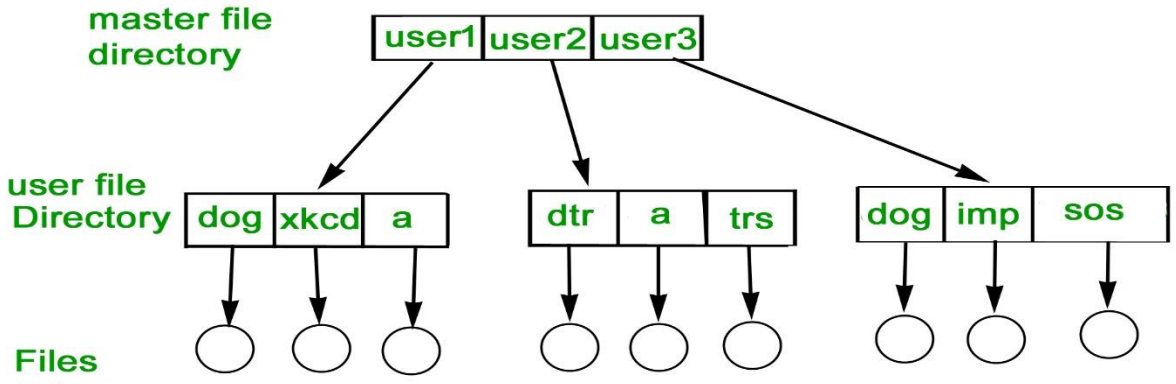
Output:

```
enter the no of directories: 2
enter the name of the directory 1: a
enter the name of the directory 2: b
enter the no of file in directory 1: 2
enter the no of file in directory 2: 2
enter the name of the file 1 of directory 1: xyz
enter the name of the file 2 of directory 1: lmn

enter the name of the file 1 of directory 2: def
enter the name of the file 2 of directory 2: ghi

a               xyz     lmn
b               def     ghi
```

Two-level directory:

In this separate directories for each user is maintained.

Path name: Due to two levels there is a path name for every file to locate that file.We can have same file name for different user.Searching is efficient in this method.



Two-level directory Program:

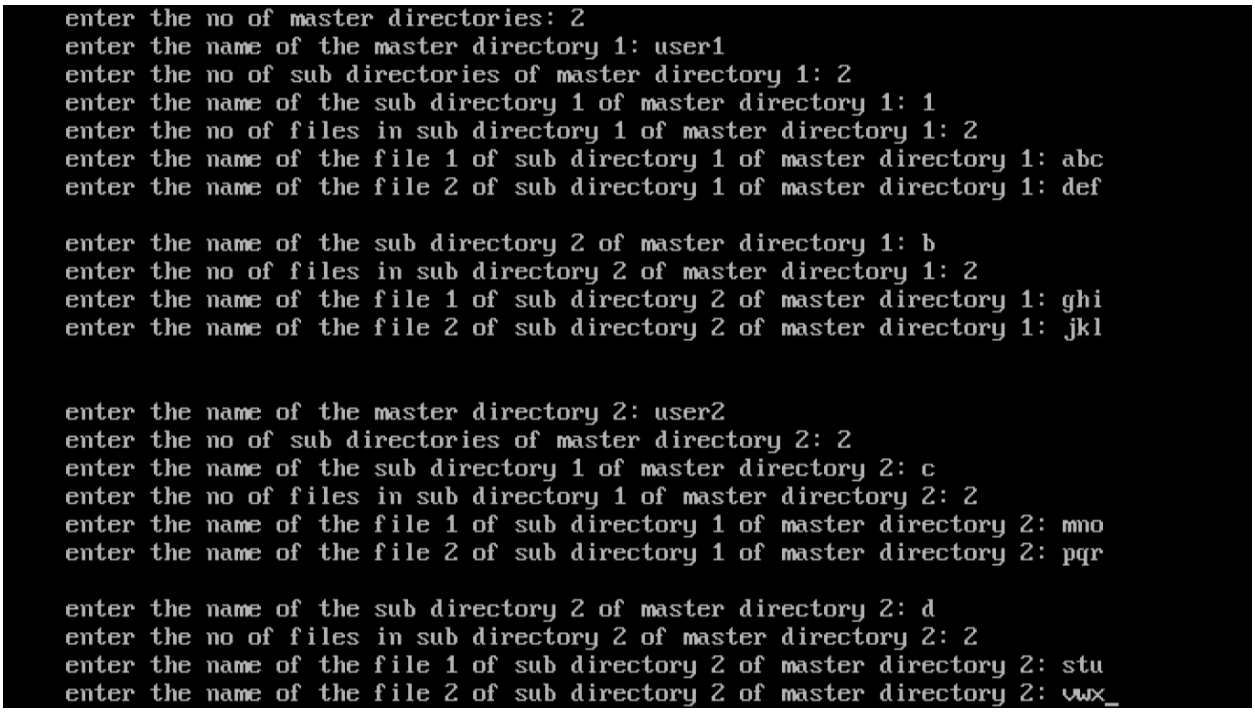
```
#include<stdio.h>
#include<conio.h>
#define M 15
struct masterdirectory
{
char mdname[M];
struct directory
{
char dname[M];
int nf;
struct names
{
char fname[M];
}n[M];
}d[M];
}md[M];
void main()
{
int nmd,nsd,i,j,k,l,m;
char fn[M],sdn[M],mdn[M];
clrscr();
printf("enter the no of master directories: ");
scanf("%d",&nmd);
for(l=0;l<nmd;l++)
{
```

```
z:
printf("enter the name of the master directory %d: ",l+1);
scanf("%s",&mdn);
for(k=0;k<nmd;k++)
if(strcmp(mdn,md[k].mdname)==0)
{
printf("the master directory name already exist.re-enter master directory\n");
goto z;
}
strcpy(md[l].mdname,mdn);
printf("enter the no of sub directories of master directory %d: ",l+1);
scanf("%d",&nsd);
for(m=0;m<nsd;m++)
{
y:
printf("enter the name of the sub directory %d of master directory %d: ",m+1,l+1);
scanf("%s",&sdn);
for(k=0;k<nsd;k++)
if(strcmp(sdn,md[l].d[k].dname)==0)
{
printf("sub directory name already exist.re-enter sub directory name\n");
goto y;
}
strcpy(md[l].d[m].dname,sdn);
printf("enter the no of files in sub directory %d of master directory %d: ",m+1,l+1);
scanf("%d",&md[l].d[m].nf);
for(j=0;j<md[l].d[m].nf;j++)
```

```
{
printf("enter the name of the file %d of sub directory %d of master directory %d:
",j+1,m+1,l+1);
scanf("%s",&fn);
for(k=0;k<md[l].d[m].nf;k++)
if(strcmp(fn,md[l].d[m].n[k].fname)==0)
{
j--;
printf("file name already exist.re-enter the file name\n");
goto x;
}
strcpy(md[l].d[m].n[j].fname,fn);
x:
}
printf("\n");
}
printf("\n");
}
printf("\n");
printf("\nTWO LEVEL DIRECTORY IMPLEMENTATION:\n----------------------------------
-------------------------------------\n");
for(k=0;k<nmd;k++)
{
printf("\n");
textcolor(20);
cprintf("%s",md[k].mdname);
printf("\t\t");
textcolor(30);
```

```
for(i=0;i<nsd;i++)
{
cprintf("%s",md[k].d[i].dname);
printf("\t\t");
for(j=0;j<md[k].d[i].nf;j++)
printf("%s\t",md[k].d[i].n[j].fname);
printf("\n\t\t");
}
}
getch();
}
```

Output:



Week-5:

Simulate Bankers Algorithm for Dead Lock Avoidance

Program:

```
#include<stdio.h>
#include<conio.h>
#define MAX 20
void main()
{
int np,temp1,ch,rp,pp,x,y,count,temp2,nr,i,j,k,flag;
intp[MAX],in[MAX],temp[MAX],rs[MAX],a[MAX][MAX],m[MAX][MAX],av[MAX],n[
MAX][MAX],w[MAX],f[MAX],ps[MAX];
np=temp1=count=temp2=nr=pp=i=j=k=flag=ch=rp=x=y=0;
for(i=0;i<MAX;i++)
{
p[i]=-1;
in[i]=-1;
temp[i]=-1;
av[i]=-1;
w[i]=-1;
f[i]=-1;
rs[i]=-1;
ps[i]=-1;
}
for(i=0;i<MAX;i++)
for(j=0;j<MAX;j++)
{
a[i][j]=-1;
```

```
m[i][j]=-1;
n[i][j]=-1;
}
clrscr();
printf("enter the no of processes:");
scanf("%d",&np);
printf("enter the no of resources:");
scanf("%d",&nr);
printf("enter the instances of resources\n");
for(i=0;i<nr;i++)
{
printf("%d:-",i+1);
scanf("%d",&in[i]);
}
printf("enter the process id's:\n");
for(i=0;i<np;i++)
scanf("%d",&p[i]);
//getting allocation matrix
printf("\nenter the values for allocation matrix of resources\n\t ");
for(i=1;i<=nr;i++)
printf("%d ",i);
printf("\n");
for(i=0;i<np;i++)
{
printf("for p[%d]:",i+1);
for(j=0;j<nr;j++)
scanf("%d",&a[i][j]);
```

```
}
//getting max matrix
printf("\nenter the values for the max matrix of resources\n\t ");
for(i=1;i<=nr;i++)
printf("%d ",i);
printf("\n");
for(i=0;i<np;i++)
{
printf("for p[%d]:",i+1);
for(j=0;j<nr;j++)
scanf("%d",&m[i][j]);
}
again:
//finding of available matrix
for(j=0;j<nr;j++)
{
temp1=0;
for(i=0;i<np;i++)
temp1+=a[i][j];
temp[j]=temp1;
}
for(i=0;i<nr;i++)
av[i]=in[i]-temp[i];
//need matrix
for(i=0;i<np;i++)
for(j=0;j<nr;j++)
n[i][j]=m[i][j]-a[i][j];
```

```
//printing available matrix
printf("the available matrix is:\n");
for(i=0;i<nr;i++)
printf("  %d",av[i]);
//printing of need matrix
printf("\nthe need matrix is:\n\t");
for(i=0;i<nr;i++)
printf("\t%d",i+1);
for(i=0;i<np;i++)
{
printf("\nprocess %d",i+1);
for(j=0;j<nr;j++)
printf("\t%d",n[i][j]);
}
//safety algorithm
//copying of avilable values into work
for(i=0;i<nr;i++)
w[i]=av[i];
//intializing finish to false
for(i=0;i<np;i++)
f[i]=-1;
k=count=0;
while(count!=np)
{ //while loop
 for(i=0;i<np;i++)
 { //for loop
  flag=0;
```

```
 if(f[i]==-1)
  for(j=0;j<nr;j++)
  if(n[i][j]<=w[j])
  flag++;
  else
  break;
  if(flag==nr)
  {
   for(k=0;k<nr;k++)
   w[k]=w[k]+a[i][k];
   f[i]=0;
   count++;
   ps[count-1]=i;
  }
 }//for loop ending
}//while loop ending
/*printf("\nthe total work is:");
for(i=0;i<nr;i++)
printf("  %d",w[i]);   */
printf("\nthe safe state sequence is:");
for(i=0;i<np;i++)
printf("  %d",ps[i]);
temp2=0;
for(i=0;i<np;i++)
if(f[i]==0)
temp2++;
if(temp2==np)
```

```
printf("\nthe system is in safe state");
else
printf("\ndead lock  has been occured");
op:
//request for a resource
printf("\ndo u want to request for any resource,if yes enter '0' else '1':");
scanf("%d",&ch);
if(ch==0)
{
 printf("enter for which process that you want to add a resources:");
 scanf("%d",&pp);
 for(i=0;i<np;i++)
 if(p[i]==pp)
 {
 rp=i;
 break;
 }
 printf("enter the resources:");
 for(i=0;i<nr;i++)
 scanf("%d",&rs[i]);
 for(i=0;i<nr;i++)
 if(rs[i]<=n[rp][i])
  x++;
 if(x==nr)
 {
  for(i=0;i<nr;i++)
  if(rs[i]<=av[i])
```

```
 y++;
 if(y==nr)
 {
  for(i=0;i<nr;i++)
  {
   av[i]-=rs[i];
   a[rp][i]+=rs[i];
   n[rp][i]-=rs[i];
  }
 goto again;
 }
 else
 printf("the requested resource has exceeded its available");
}
else
printf("the process has exceeded its max");
}
else if(ch==1)
printf("TERMINATED");
else
{
printf("wrong option");
goto op;
}
getch();
}
```

Output:

```
for p[2]:2 0 0
for p[3]:3 0 2
for p[4]:2 1 2
for p[5]:0 2 2

enter the values for the max matrix of resources
        1 2 3
for p[1]:7 5 3
for p[2]:3 2 2
for p[3]:9 0 2
for p[4]:2 2 2
for p[5]:4 3 3
the available matrix is:
  3  1  1
the need matrix is:
               1       2       3
process 1      7       4       3
process 2      1       2       2
process 3      6       0       0
process 4      0       1       0
process 5      4       1       1
the safe state sequence is:  3  4  1  2  0
the system is in safe state
do u want to request for any resource,if yes enter '0' else '1':1
TERMINATED_
```

Week-6:

Simulate all page replacement algorithms a) FIFO b) LRU c) LFU

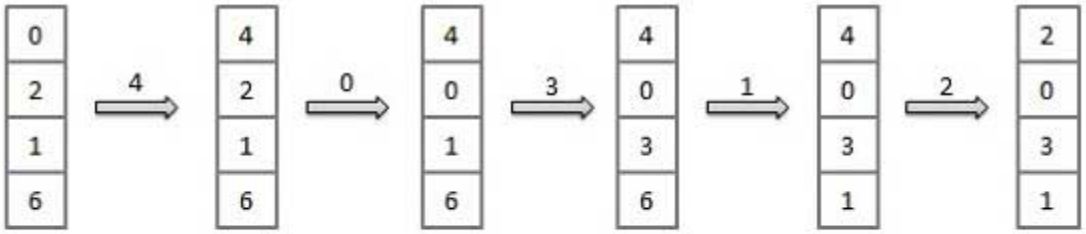
First In First Out (FIFO) algorithm

Oldest page in main memory is the one which will be selected for replacement.

Easy to implement, keep a list, replace pages from the tail and add new pages at the head.

Reference String : 0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1

Misses : x x x x x x x x x



Fault Rate = 9 / 12 = 0.75

FIFO Program:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int page[20],i,j,n,p,pf=0,frame[20],k=0,t=0,x;
clrscr();
printf("Enter the no. of frames: ");
scanf("%d",&n);
printf("Enter the no. of pages: ");
scanf("%d",&p);
printf("Enter page number sequence:-\n");
for(i=0;i<p;i++)
```

```
scanf("%d",&page[i]);
printf("The page numbers are:\n");
for(i=0;i<p;i++)
printf("%d ",page[i]);
printf("\n");
for(i=0;i<n;i++)
frame[i]=-1;
for(i=0;i<p;i++)
{
if(k==n)
k=0;
for(j=0;j<n;j++)
{
if(page[i]==frame[j])
{
t=1;
}
}
if(t!=1)
{
frame[k]=page[i];
k++;
pf=pf+1;
printf("%d:\t",page[i]);
for(x=0;x<n;x++)
{
printf("%d\t",frame[x]);
```

```
}
printf("Page fault\n");
}
else
{
printf("%d:\t",page[i]);
for(x=0;x<n;x++)
printf("%d\t",frame[x]);
printf("HIT\n");
t=0;
}
}
printf("page fault:%d",pf);
getch();
}
```

Output:

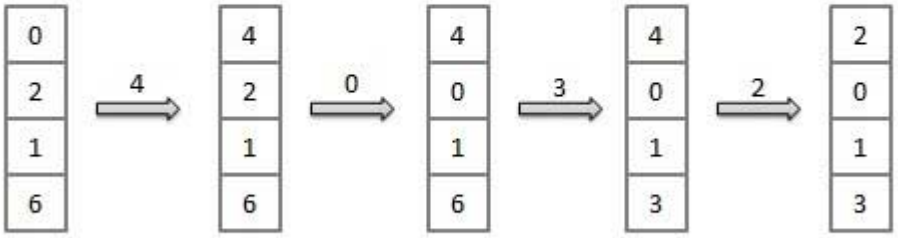
```
Enter page number sequence:-
7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1
The page numbers are:
7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1
7:      7       -1      -1      Page fault
0:      7       0       -1      Page fault
1:      7       0       1       Page fault
2:      2       0       1       Page fault
0:      2       0       1       HIT
3:      2       3       1       Page fault
0:      2       3       0       Page fault
4:      4       3       0       Page fault
2:      4       2       0       Page fault
3:      4       2       3       Page fault
0:      0       2       3       Page fault
3:      0       2       3       HIT
2:      0       2       3       HIT
1:      0       1       3       Page fault
2:      0       1       2       Page fault
0:      0       1       2       HIT
1:      0       1       2       HIT
7:      7       1       2       Page fault
0:      7       0       2       Page fault
1:      7       0       1       Page fault
page fault:15
```

Least Recently Used (LRU) algorithm

Page which has not been used for the longest time in main memory is the one which will be selected for replacement.Easy to implement, keep a list, replace pages by looking back into time.

Reference String : 0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1

Misses : x x x x x x x x



Fault Rate = 8 / 12 = 0.67

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

Let say the page reference string 7 0 1 2 0 3 0 4 2 3 0 3 2 . Initially we have 4 page slots empty.
Initially all slots are empty, so when 7 0 1 2 are allocated to the empty slots —> 4 Page faults
0 is already their so —> 0 Page fault.
when 3 came it will take the place of 7 because it is least recently used —>1 Page fault
0 is already in memory so —> 0 Page fault.
4 will takes place of 1 —> 1 Page Fault
Now for the further page reference string —> 0 Page fault because they are already available in the memory.

LRU

when a page fault occurs, throw out the page that has been unused for the longest time. This strategy is called **LRU** (**Least Recently Used**) paging.

LRU Program:

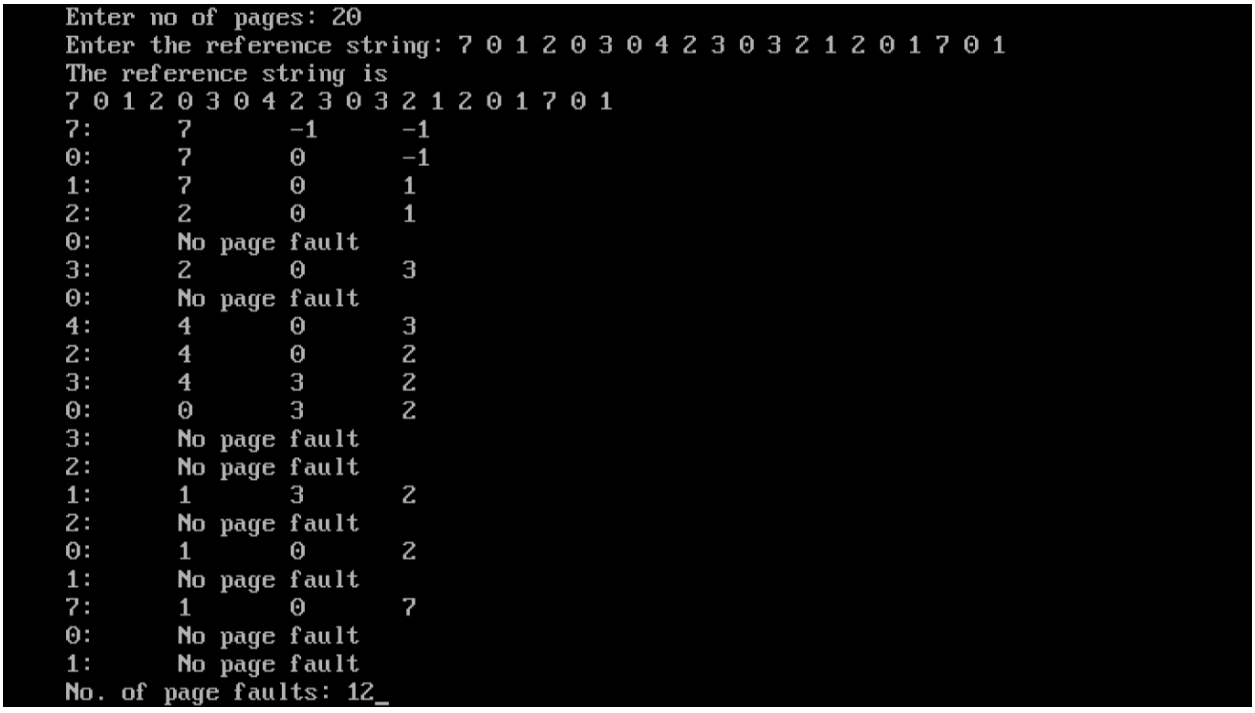
```
#include<stdio.h>

#include<conio.h>

void main()
```

```
{
int key[10],keyindex=0,count=0,a[25],min,b[10],i,m,n,fault=1,page=0,j,k=0,x,y=0;
clrscr();
printf("Enter the no of frames: ");
scanf("%d",&m);
printf("Enter no of pages: ");
scanf("%d",&n);
printf("Enter the reference string: ");
for(i=0;i<n;i++)
scanf("%d",&a[i]);

printf("The reference string is\n");
for(i=0;i<n;i++)
printf("%d ",a[i]);
printf("\n");
page=0;
for(i=0;i<m;i++)
b[i]=-1;
for(i=0;i<n;i++)
{
fault=1;
for(j=0;j<m;j++)
{
if(a[i]==b[j])
{       key[j]=count++;
fault=0;
printf("%d:\tNo page fault",a[i]);
```

```
break;
}
}
if(fault==1)
{
page=page+1;
if(k<m)
{
b[k]=a[i];
key[k]=count++;
k++;
}
else
{
min=99;
for(x=0;x<m;x++)
if(min>key[x])
{
min=key[x];
keyindex=x;
}
b[keyindex]=a[i];
key[keyindex]=count++;
}
printf("%d:\t",a[i]);
for(y=0;y<m;y++)
printf("%d\t",b[y]);
```

```
}
printf("\n");
}
printf("No. of page faults: %d",page);
getch();
}
```

Output:



Page Buffering algorithm

To get a process start quickly, keep a pool of free frames.On page fault, select a page to be replaced.Write the new page in the frame of free pool, mark the page table and restart the process.Now write the dirty page out of disk and place the frame holding replaced page in free pool.Least frequently UsedLFU) algorithm

LFU:

Least Frequently Used (LFU) is a caching algorithm in which the least frequently used cache block is removed whenever the cache is overflowed. In LFU we check the old page as well as the frequency of that page and if the frequency of the page is larger than the old page we cannot remove it and if all the old pages are having same frequency then take last i.e FIFO method for that and remove that page.

LFU Program:

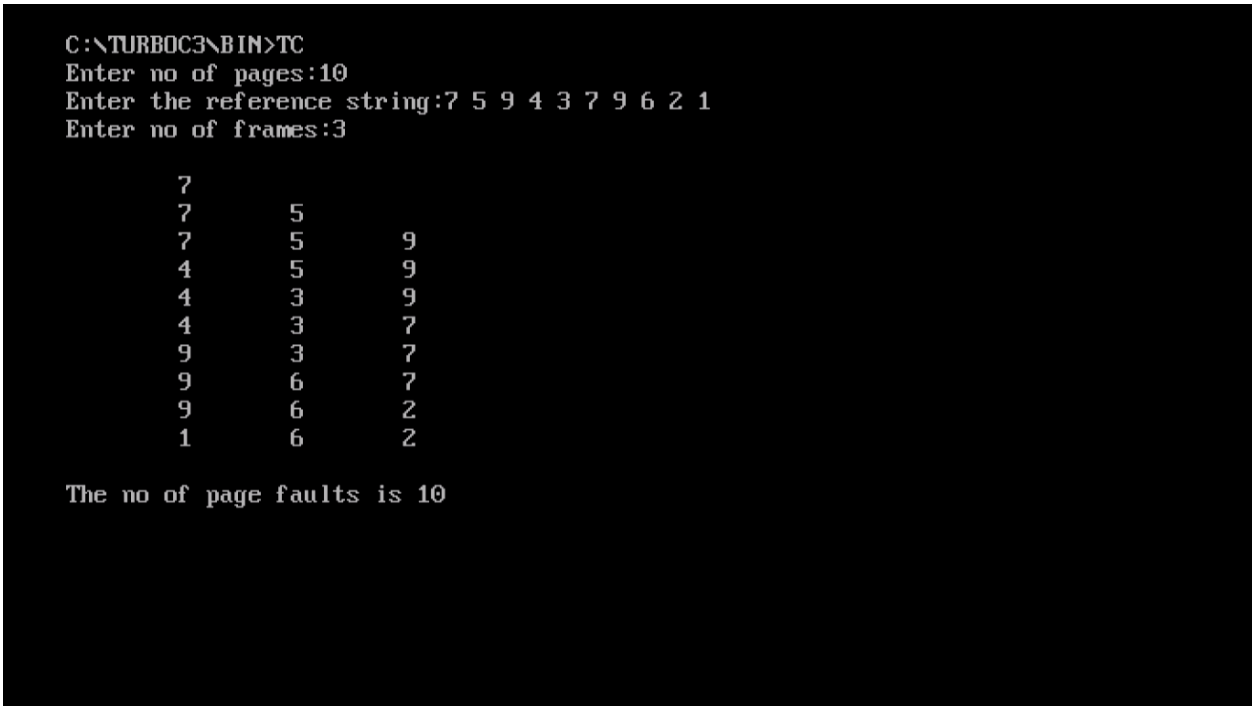
```
#include<stdio.h>

main()

{

int q[20],p[50],c=0,c1,d,f,i,j,k=0,n,r,t,b[20],c2[20];

printf("Enter no of pages:");

scanf("%d",&n);

printf("Enter the reference string:");

for(i=0;i<n;i++)

scanf("%d",&p[i]);

printf("Enter no of frames:");

scanf("%d",&f);

q[k]=p[k];

printf("\n\t%d\n",q[k]);

c++;

k++;

for(i=1;i<n;i++)

{
```

```
c1=0;
for(j=0;j<f;j++)
{
if(p[i]!=q[j])
c1++;
}
if(c1==f)
{
c++;
if(k<f)
{
q[k]=p[i];
 k++;
for(j=0;j<k;j++)
printf("\t%d",q[j]);
printf("\n");
}
else
{
for(r=0;r<f;r++)
{
c2[r]=0;
for(j=i-1;j<n;j--)
{
if(q[r]!=p[j])
c2[r]++;
else
```

```
break;
}
}
for(r=0;r<f;r++)
b[r]=c2[r];
for(r=0;r<f;r++)
{
for(j=r;j<f;j++)
{
if(b[r]<b[j])
{
t=b[r];
b[r]=b[j];
b[j]=t;
}
}
}
for(r=0;r<f;r++)
{
if(c2[r]==b[0])
q[r]=p[i];
printf("\t%d",q[r]);
}
printf("\n");
}
}
}
```

```
printf("\nThe no of page faults is %d",c);

}
```

Output:



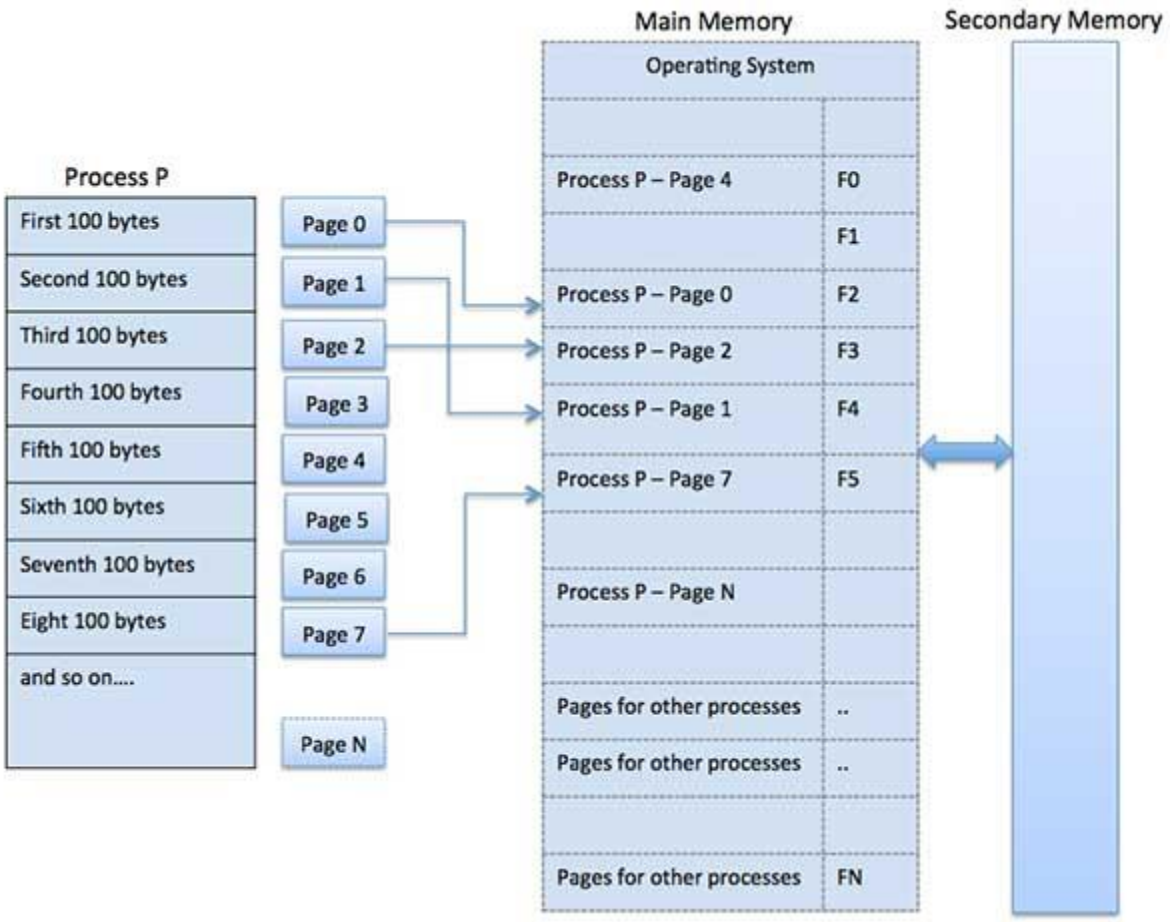
Week-7:

Simulate Paging Technique of memory management.

Paging

Paging is a memory management technique in which process address space is broken into blocks of the same size called pages (size is power of 2, between 512 bytes and 8192 bytes). The size of the process is measured in the number of pages.

Similarly, main memory is divided into small fixed-sized blocks of (physical) memory called frames and the size of a frame is kept the same as that of a page to have optimum utilization of the main memory and to avoid external fragmentation.



Address Translation

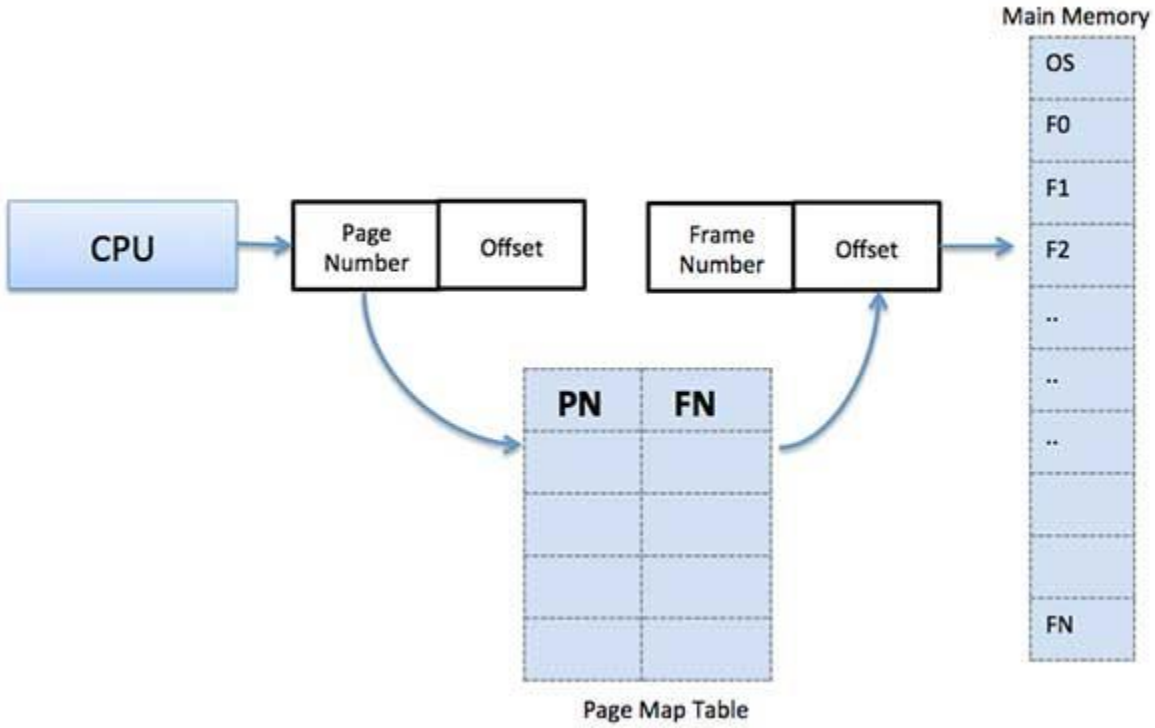
Page address is called logical address and represented by page numberand the offset.

Logical Address = Page number + page offset

Frame address is called physical address and represented by a frame number and the offset.

Physical Address = Frame number + page offset

A data structure called page map table is used to keep track of the relation between a page of a process to a frame in physical memory.

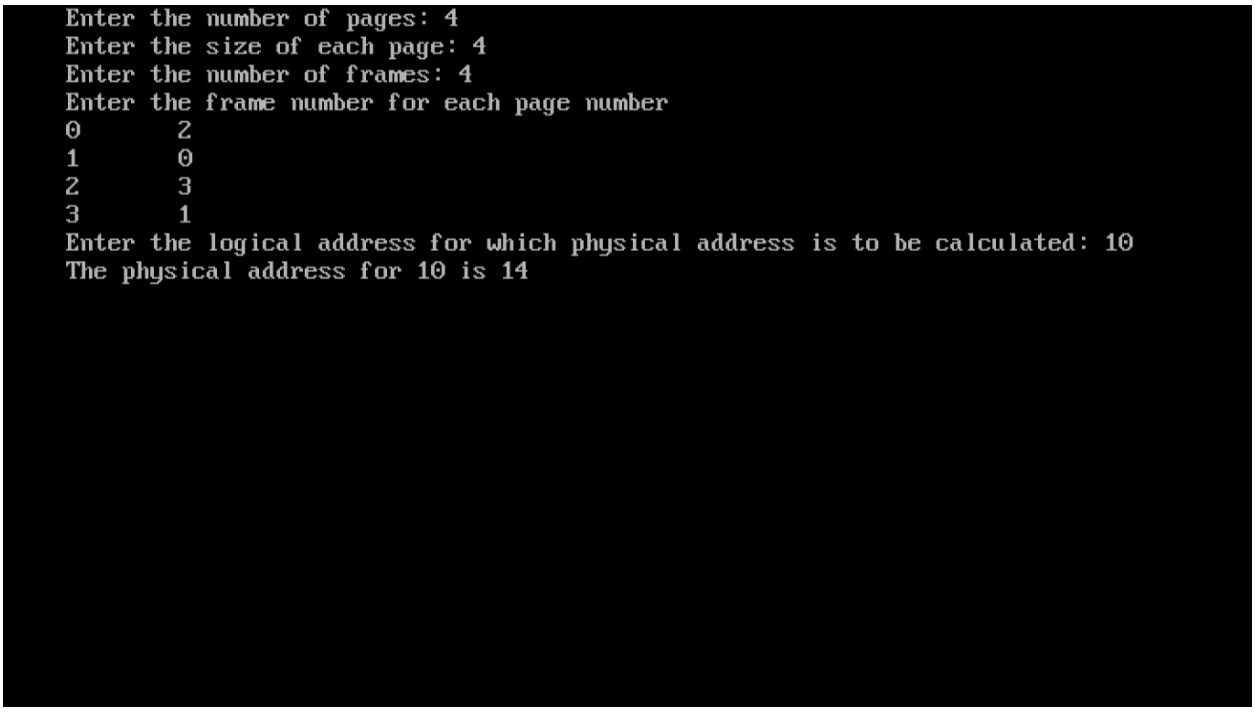


Program:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int p,ps,f,fs,pt[20],i,la,pn,offset,pa;
clrscr();
printf("Enter the number of pages: ");
scanf("%d",&p);
printf("Enter the size of each page: ");
scanf("%d",&ps);
printf("Enter the number of frames: ");
scanf("%d",&f);
fs=ps;
```

```
printf("Enter the frame number for each page number\n");
for(i=0;i<p;i++)
{
printf("%d\t",i);
scanf("%d",&pt[i]);
}
printf("Enter the logical address for which physical address is to be calculated: ");
scanf("%d",&la);
pn=la/ps;
offset=la%ps;
pa=pt[pn]*fs+offset;
printf("The physical address for %d is %d ",la,pa);
getch();
}
```

Output:


Enter the number of pages: 4
Enter the size of each page: 4
Enter the number of frames: 4
Enter the frame number for each page number
0 2
1 0
2 3
3 1
Enter the logical address for which physical address is to be calculated: 10
The physical address for 10 is 14