Exp. No:1(a) ROUND ROBIN SCHEDULING

Aim: Write a C program to implement the various process scheduling mechanisms such as Round Robin Scheduling.

Description:

Scheduling is a fundamental operating system function.
CPU scheduling is the basis of multi programming operating system. CPU scheduling algorithm determines how the CPU will be allocated to the process. These are of two types.
1. Primitive scheduling algorithms
2. Non-Primitive scheduling algorithms

1) **Primitive Scheduling algorithms**: In this, the CPU can release the process even in the middle of execution. For example: the cpu executes the process p1, in the middle of execution the cpu received a request signal from process p2, then the OS compares the priorities of p1&p2. If the priority p1 is higher than the p2 then the cpu continue the execution of process p1.Otherwise the cpu preempt the process p1 and assigned to process p2.

2) **Non-Primitive Scheduling algorithm**: In this, once the cpu assigned to a process the processor do not release until the completion of that process. The cpu will assign to some other job only after the previous job has finished.

Scheduling methodology:

Though put: It means how many jobs are completed by the CPU with in a time period.
Turn around time: The time interval between the submission of the process and the time of the completion is the turn around time.
Turn around time=Finished time – arrival time
Waiting time: it is the sum of the periods spent waiting by a process in the ready queue
Waiting time=Starting time- arrival time
Response time: it is the time duration between the submission and first response
Response time=First response-arrival time

CPU Utilization: This is the percentage of time that the processor is busy. CPU utilization may range from 0 to 100%.

Round Robin: It is a primitive scheduling algorithm it is designed especially for time sharing systems. In this, the CPU switches between the processes. When the time quantum expired, the CPU switches to another job. A small unit of time called a quantum or time slice. A time quantum is generally is a circular queue new processes are added to the tail of the ready queue.
If the process may have a CPU burst of less than one time slice then the process release the CPU voluntarily. The scheduler will then process to next process ready queue otherwise; the process will be put at the tail of the ready queue**.**

Algorithm for Round Robin:

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

(a) Average waiting time = Total waiting Time / Number of process

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

/* Program to Simulate Round Robin CPU Scheduling Algorithm */

```
#include<stdio.h>
#include<conio.h>
struct process
{
   char pn[10];
   int bt,ct,time;
}p[10];
void main()
{
  int i,full,n,tq,wt[10],tat[10], time1=0;
  float avgwt=0.0;
  clrscr();
  printf("Enter number of processes:");
  scanf("%d",&n);
  printf("Enter process name and burst time of %d process\n", n);
  for(i=0;i<n;i++)
   {
      scanf("%s%d",&p[i].pn,&p[i].bt);
      p[i].time=p[i].bt;
   }
  printf("Enter quantum:");
  scanf("%d",&tq);
  full=n;
  while(full)
   {
     for(i=0;i<n;i++)
      {
        if(p[i].bt>=tq)
         {
            p[i].bt-=tq;
```

```
                time1=time1+tq;
              }
             else if(p[i].bt!=0)
                {
                  time1+=p[i].bt;
                  p[i].bt=0;
                }
            else
              continue;
            if(p[i].bt==0)
              {
                full=full-1;
                tat[i]=time1;
              }
        }
    }
     for(i=0;i<n;i++)
      {
        p[i].ct=tat[i];
        wt[i]=tat[i]-p[i].time;
      }
    printf("---------------------------------- \n");
    printf("PN\tBt\tCt\tTat\tWt\n");
    printf("---------------------------------- \n");
     for(i=0;i<n;i++)
      {
         printf("%2s\t%2d\t%2d\t%2d\t%2d\n",p[i].pn,p[i].time,p[i].ct,tat[i],wt[i]);
         avgwt+=wt[i];
      }

    printf("----------------------------------\n");
    avgwt=avgwt/n;
    printf(" Average waiting time =
    %.2f\n",avgwt); printf("----------------------------
    ------\n"); getch();
}
```

Exp. No: 1(b) SJF SCHEDULING

Aim: Write a C program to implement the various process scheduling mechanisms such as SJF Scheduling .

Description:

Shortest Job First: The criteria of this algorithm are which process having the smallest CPU burst, CPU is assigned to that next process. If two process having the same CPU burst time FCFS is used to break the tie.

Algorithm for SJF:

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: For each process in the ready queue, calculate

(a) Waiting time for process(n)= waiting time of process (n-1) + Burst time of process(n-1)

(b) Turn around time for Process(n)= waiting time of Process(n)+ Burst time for process(n)

Step 6: Calculate

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

(d) Average Turnaround time = Total Turnaround Time / Number of

process Step 7: Stop the process

/* Program to Simulate Shortest Job First CPU Scheduling Algorithm */

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
  int i,j,n,bt[10],compt[10], wt[10],tat[10],temp;
  float sumwt=0.0,sumtat=0.0,avgwt,avgtat;
  clrscr();
  printf("Enter number of processes: ");
  scanf("%d",&n);
  printf("Enter the burst time of %d process\n",
  n); 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])
        {
          temp=bt[i];
          bt[i]=bt[j];
          bt[j]=temp;
        }
 compt[0]=bt[0];
 for(i=1;i<n;i++)
 compt[i]=bt[i]+compt[i-1];
 for(i=0;i<n;i++)
 {
   tat[i]=compt[i];
   wt[i]=tat[i]-bt[i];
   sumtat+=tat[i];
   sumwt+=wt[i];
 }
 avgwt=sumwt/n;
 avgtat=sumtat/n;
 printf("------------------------------ \n");
 printf("Bt\tCt\tTat\tWt\n");
 printf("------------------------------ \n");
 for(i=0;i<n;i++)
  {
     printf("%2d\t%2d\t%2d\t%2d\n",i,bt[i],compt[i],tat[i],wt[i]);
  }
 printf("------------------------------\n");
 printf(" Avgwt = %.2f\tAvgtat = %.2f\n",avgwt,avgtat);
 printf("------------------------------\n");
 getch();
}
```

Exp. No: 1(c)FCFS SCHEDULING

Aim: Write a C program to implement the various process scheduling mechanisms such

Description:

First-come, first-serve scheduling(FCFS): In this, which process enter the ready queue first is served first. The OS maintains DS that is ready queue. It is the simplest CPU scheduling algorithm. If a process request the CPU then it is loaded into the ready queue, which process is the head of the ready queue, connect the CPU to that process.

Algorithm for FCFS scheduling:

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: Set the waiting of the first process as „0" and its burst time as its turn around time

Step 5: for each process in the Ready Q calculate

(c) Waiting time for process(n)= waiting time of process (n-1) + Burst time of process(n-1)

(d) Turn around time for Process(n)= waiting time of Process(n)+ Burst time for process(n)

Step 6: Calculate

(e) Average waiting time = Total waiting Time / Number of process

(f) Average Turnaround time = Total Turnaround Time / Number of

process Step 7: Stop the process

```
/* Program to Simulate First Come First Serve CPU Scheduling Algorithm
*/ #include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
  int i,j,n,bt[10],compt[10],at[10], wt[10],tat[10];
  float sumwt=0.0,sumtat=0.0,avgwt,avgtat;
  clrscr();
  printf("Enter number of processes: ");
  scanf("%d",&n);
  printf("Enter the burst time of %d process\n",
  n); for(i=0;i<n;i++)
   {
     scanf("%d",&bt[i]);
   }
  printf("Enter the arrival time of %d process\n", n);
  for(i=0;i<n;i++)
```

```
 {
   scanf("%d",&at[i]);
 }
 compt[0]=bt[0]-at[0];
 for(i=1;i<n;i++)
 compt[i]=bt[i]+compt[i-1];
 for(i=0;i<n;i++)
 {
   tat[i]=compt[i]-at[i];
   wt[i]=tat[i]-bt[i];
   sumtat+=tat[i];
   sumwt+=wt[i];
 }
 avgwt=sumwt/n;
 avgtat=sumtat/n;
 printf("---------------------------------- \n");
 printf("PN\tBt\tCt\tTat\tWt\n");
 printf("---------------------------------- \n");
 for(i=0;i<n;i++)
 {
    printf("%d\t%2d\t%2d\t%2d\t%2d\n",i,bt[i],compt[i],tat[i],wt[i]);
 }
 printf("----------------------------------\n");
 printf(" Avgwt = %.2f\tAvgtat = %.2f\n",avgwt,avgtat);
 printf("----------------------------------\n");
 getch();
}
```

Exp. No: 1(d)PRIORITY SCHEDULING

Aim: Write a C program to implement the various process scheduling mechanisms such as Priority Scheduling.

Description:

Priority Scheduling: These are of two types.

One is internal priority, second is external priority. The cpu is allocated to the process with the highest priority. Equal priority processes are scheduled in the FCFS order. Priorities are generally some fixed range of numbers such as 0 to 409. The low numbers represent high priority.

Algorithm for Priority Scheduling:

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 turn around time

Step 6: For each process in the Ready Q calculate

(e) Waiting time for process(n)= waiting time of process (n-1) + Burst time of process(n-1)

(f) Turn around time for Process(n)= waiting time of Process(n)+ Burst time for process(n)

Step 7: Calculate

(g) Average waiting time = Total waiting Time / Number of process

(h) Average Turnaround time = Total Turnaround Time / Number of

process Step 8: Stop the process

/* Program to Simulate Priority CPU Scheduling Algorithm

```
*/ #include<stdio.h>
#include<conio.h>
void main()
{
  int i,j,n,bt[10],p[10],compt[10], wt[10],tat[10],temp1,temp2;
  float sumwt=0.0,sumtat=0.0,avgwt,avgtat;
  clrscr();
  printf("Enter number of processes: ");
  scanf("%d",&n);
  printf("Enter the burst time of %d process\n",
  n); for(i=0;i<n;i++)
     scanf("%d",&bt[i]);
  printf("Enter the priority of %d process\n", n);
  for(i=0;i<n;i++)
    scanf("%d",&p[i]);
```

```
 for(i=0;i<n;i++)
  for(j=i+1;j<n;j++)
  if(p[i]>p[j])
         {
          temp1=bt[i];
          bt[i]=bt[j];
          bt[j]=temp1;
          temp2=p[i];
          p[i]=p[j];
          p[j]=temp2;
         }
 compt[0]=bt[0]; wt[0]=0;
 for(i=1;i<n;i++)
 compt[i]=bt[i]+compt[i-1];
 for(i=0;i<n;i++)
 {
   tat[i]=compt[i];
   wt[i]=tat[i]-bt[i];
   sumtat+=tat[i];
   sumwt+=wt[i];
 }
 avgwt=sumwt/n; avgtat=sumtat/n;
 printf("-----------------------------\n");
 printf("Bt\tCt\tTat\tWt\n");
 printf("-----------------------------\n");
 for(i=0;i<n;i++)
  {
    printf("%2d\t%2d\t%2d\t%2d\n",bt[i],compt[i],tat[i],wt[i]);
  }
 printf("-----------------------------\n");
 printf(" Avgwt = %.2f\tAvgtat = %.2f\n",avgwt,avgtat);
 printf("-----------------------------\n");
 getch();
}
```

Exp. No:2(a)SEQUENTIAL FILE ALLOCATION

AIM: Write a C Program to implement Sequential File Allocation method.

Description:

Files are normally stored on the disks. So the main problem is how to allocate space to those files. So that disk space is utilized effectively and files can be accessed quickly. Three major strategies of allocating disc space are in wide use. Sequential, indexed and linked.

Sequential allocation :

In this allocation strategy, each file occupies a set of contiguously blocks on the disk. This strategy is best suited. For sequential files, the file allocation table consists of a single entry for each file. It shows the filenames, staring block of the file and size of the file. The main problem of this strategy is, it is difficult to find the contiguous free blocks in the disk and some free blocks could happen between two files.

Algorithm for Sequential File Allocation:

Step 1: Start the program.
Step 2: Get the number of memory partition and their sizes.
Step 3: Get the number of processes and values of block size for each process.
Step 4: First fit algorithm searches all the entire memory block until a hole which is big enough is encountered. It allocates that memory block for the requesting process.
Step 5: Best-fit algorithm searches the memory blocks for the smallest hole which can be allocated to requesting process and allocates if.
Step 6: Worst fit algorithm searches the memory blocks for the largest hole and allocates it to the process.
Step 7: Analyses all the three memory management techniques and display the best algorithm which utilizes the memory resources effectively and efficiently.
Step 8: Stop the program.

/* Program to simulate sequential file allocation strategy

```
*/ #include < stdio.h>
#include<conio.h>
void main()
{
 int f[50], i, st, len, j, c, k, count = 0;
clrscr();
for(i=0;i<50;i++)
f[i]=0;
printf("Files Allocated are : \n");
x: count=0;
printf("Enter starting block and length of files:
"); scanf("%d%d", &st,&len);
for(k=st;k<(st+len);k++)
   if(f[k]==0)
      count++;
if(len==count)
{
  for(j=st;j<(st+len);j++)
```

```
    if(f[j]==0)
     {
       f[j]=1;
       printf("%d\t%d\n",j,f[j]);
     }
   if(j!=(st+len-1))
     printf(" The file is allocated to disk\n");
}
else
 printf(" The file is not allocated \n");
printf("Do you want to enter more file(Yes - 1/No - 0)");
scanf("%d", &c);
if(c==1)
goto   x;
else
  exit();
getch();
}
```

Exp. No:2(b)INDEXED FILE ALLOCATION

AIM: Write a C Program to implement Indexed File Allocation method.

Description:

Indexed allocation :

Indexed allocation supports both sequential and direct access files. The file indexes are not physically stored as a part of the file allocation table. Whenever the file size increases, we can easily add some more blocks to the index. In this strategy, the file allocation table contains a single entry for each file. The entry consisting of one index block, the index blocks having the pointers to the other blocks. No external fragmentation.

Algorithm for Indexed File Allocation:

Step 1: Start.

Step 2: Let n be the size of the buffer

Step 3: check if there are any producer

Step 4: if yes check whether the buffer is full

Step 5: If no the producer item is stored in the buffer

Step 6: If the buffer is full the producer has to wait

Step 7: Check there is any consumer.If yes check whether the buffer is empty

Step 8: If no the consumer consumes them from the buffer

Step 9: If the buffer is empty, the consumer has to wait.

Step 10: Repeat checking for the producer and consumer till required

Step 11: Terminate the process.

/* Program to simulate indexed file allocation strategy

```
*/ #include<stdio.h>
#include<conio.h>
#include<stdlib.h>
void main()
{
 int f[50], index[50],i, n, st, len, j, c, k,
ind,count=0; clrscr();
for(i=0;i<50;i++)
f[i]=0;
x:printf("Enter the index block: ");
scanf("%d",&ind);
if(f[ind]!=1)
{
  printf("Enter no of blocks needed and no of files for the index %d on the disk :
\n", ind);
  scanf("%d",&n);
}
else
```

```
{
  printf("%d index is already allocated \n",ind);
  goto x;
}
y: count=0;
for(i=0;i<n;i++)
{
   scanf("%d", &index[i]);
   if(f[index[i]]==0)
      count++;
}
if(count==n)
{
  for(j=0;j<n;j++)
  f[index[j]]=1;
  printf("Allocated\n");
  printf("File Indexed\n");
  for(k=0;k<n;k++)
    printf("%d-------->%d : %d\n",ind,index[k],f[index[k]]);
}
else
{
 printf("File in the index is already allocated
 \n"); printf("Enter another file indexed");
 goto y;
}
printf("Do you want to enter more file(Yes - 1/No - 0)");
scanf("%d", &c);
if(c==1)
goto x;
else
exit(0);
getch();
}
```

Exp. No:2(c)LINKED FILE ALLOCATION

AIM: Write a C Program to implement Linked File Allocation method.

Description:

Linked allocation:

It is easy to allocate the files, because allocation is on an individual block basis. Each block contains a pointer to the next free block in the chain. Here also the file allocation table consisting of a single entry for each file. Using this strategy any free block can be added to a chain very easily. There is a link between one block to another block, that's why it is said to be linked allocation. We can avoid the external fragmentation.

Algorithm for Linked File Allocation:

Step 1: Create a queue to hold all pages in memory
Step 2: When the page is required replace the page at the head of the queue
Step 3: Now the new page is inserted at the tail of the queue
Step 4: Create a stack
Step 5: When the page fault occurs replace page present at the bottom of the stack
Step 6: Stop the allocation.

```
/* Program to simulate linked file allocation strategy */
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
void main()
{
 int f[50], p,i, st, len, j, c, k, a;
clrscr();
for(i=0;i<50;i++)
f[i]=0;
printf("Enter how many blocks already allocated:
"); scanf("%d",&p);
printf("Enter blocks already allocated: ");
for(i=0;i<p;i++)
 {
  scanf("%d",&a);
  f[a]=1;
 }
x: printf("Enter index starting block and length: ");
scanf("%d%d", &st,&len);
k=len;
if(f[st]==0)
 {
  for(j=st;j<(st+k);j++)
  {
    if(f[j]==0)
     {
       f[j]=1;
       printf("%d-------->%d\n",j,f[j]);
```

```
    }
   else
    {
     printf("%d Block is already allocated \n",j);
     k++;
    }
  }
}
else
 printf("%d starting block is already allocated \n",st);
printf("Do you want to enter more file(Yes - 1/No - 0)");
scanf("%d", &c);
if(c==1)
goto   x;
else
  exit(0);
getch();
}
```

Exp. No: 3(a)MULTIPROGRAM VARIABLE TASK

AIM: Write a program to implement Dynamic allocation of memories in MVT.

Description:

MVT:

MVT stands for multiprogramming with variable number of tasks. Multiprogramming is a technique to execute number of programs simultaneously by a single processor. This is one of the memory management techniques. To eliminate the same of the problems with fixed partitions, an approach known as dynamic partitioning developed. In this technique, partitions are created dynamically, so that each process is loaded into partition of exactly the same size at that process. This scheme suffering from external fragmentation.

Algorithm for Multiprogram Variable Task:

Step1: start the process.
Step2: Declare variables.
Step3: Enter total memory size.
Step4: Allocate memory for os.
Step5: allocate total memory to the pages.
Step6: Display the wastage of memory.
Step7: Stop the process.

```
/* Program to simulate the MVT */
#include<stdio.h>
#include<conio.h>
void main()
{
        int m,i,p[15];
            char ch;
        clrscr();
        printf("Enter memory to be allocated: ");
        scanf("%d",&m);
        printf("Enter process size : ");
        scanf("%d", &p[0]);
        i=0;
        do
        {
          m=m-p[i];
          printf("\nRemaining memory is %d\n",m);
          abc:printf("Do you want to continue: ");
              fflush(stdin);
         scanf("%c",&ch);
         i++;
         if(ch=='y')
          {
            printf("Enter the process size: ");
            scanf("%d",&p[i]);
          }
        else
```

```
                {
                  printf("\nExternal fragmentation is %d",m);
                  break;
                }
               if(m<p[i])
               {
                 printf("\nRequired memory is not
                 available\n"); goto abc;
               }
            }while((ch=='y')&&(m>=p[i]));
           getch();
}
```

Exp. No: 3(b)MULTIPROGRAM FIXED TASK

AIM: Write a program to implement Dynamic allocation of memories in MVT.

Description:

MFT:

MFT stands for multiprogramming with fixed no of tasks.
MFT is the one of the memory management technique. In this technique, main memory is divided into no of static partitions at the system generated time. A process may be loaded into a partition of equal or greater size. The partition sizes are depending on o.s. in this memory management scheme the o.s occupies the low memory, and the rest of the main memory is available for user space. This scheme suffers from internal as well as external fragmentation.

Algorithm for Multiprogram Fixed Task:

Step1: start the process.
Step2: Declare variables.
Step3: Enter total memory size.
Step4: Allocate memory for operating system.
Step5: allocate total memory to the pages.
Step6: Display the wastage of memory.
Step7: Stop the process.

/* Program to simulate MFT */

```
#include < stdio.h>
#include <conio.h>
void main()
{
 int i, p,a[10],c[15],temp=0,total=0;
 float b,t;
 clrscr();
 printf("Enter the total memory ");
 scanf("%f",&t);
 printf("Enter the processes :");
 scanf("%d", &p);
 b=t/p;
 for(i=0;i<p;i++)
    a[i]=b;
for(i=0;i<p;i++)
{
   lable: printf("Enter memory for %d process:
   ",i); scanf("%d",&temp);
   if(temp<=b)
   {
       c[i]=a[i]-temp;
       printf("Internal fragmentation for this block");
       printf("%d",c[i]);
    }
   else
    {
       printf("Required memory is not available");
       goto lable;
```

```
    }
   total=total+c[i];
}
printf("Total internal fragmentation %d\n",total);
getch();
}
```

Exp. No:4(a)SINGLE LEVEL DIRECTORY

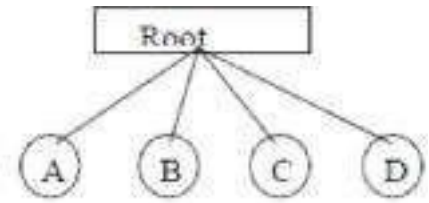
AIM: Write a program to simulate Single Level Directory.

Description:

File Organization Techniques:
a)Single Level Directory
b)Two Level
c)Hierarchical
d)General Graph Directory

The directory contains information about the files, including attributes, location and ownership. Sometimes the directories consisting of subdirectories also. The directory is itself a file, owned by the operating system and accessible by various file management routines.

Single Level Directories: It is the simplest of all directory structures, in this the directory system having only one directory, it consisting of the all files. Sometimes it is said to be root directory. The following dig. Shows single level directory that contains four files (A, B, C, D).



It has the simplicity and ability to locate files quickly. It is not used in the multi-user system, it is used on small embeddedsystem.

Algorithm for Single Level Directory Structure:

Step 1:Start

Step 2: Initialize values gd=DETECT,gm,count,i,j,mid,cir_x;
Initialize character array fname[10][20];

Step 3: Initialize graph function as
Initgraph(& gd, &gm," c:/tc/bgi");
Clear device();

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

Step 5:read number of files in variable count.

Step 6:if check i<count

Step 7: for i=0 & i<count
i increment;
Cleardevice();
setbkcolor(GREEN);
read file name;

setfillstyle(1,MAGENTA);

Step 8: mid=640/count;
cir_x=mid/3;
bar3d(270,100,370,150,0,0);
settextstyle(2,0,4);
settextstyle(1,1);
outtextxy(320,125,"rootdirectory");
setcolor(BLUE);
i++;

Step 9:for j=0&&j<=i&&cir_x+=mid
j increment;
line(320,150,cir_x,250);
fillellipse(cir_x,250,30,30);
outtextxy(cir_x,250,fname[i]);

Step 10: End

/* Program to simulate single level directory */

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
  int nf=0,i=0,j=0,ch;
  char mdname[10],fname[10][10],name[10];
  clrscr();
  printf("Enter the directory name:");
  scanf("%s",mdname);
  printf("Enter the number of files:");
  scanf("%d",&nf);
  do
   {
     printf("Enter file name to be created:");
     scanf("%s",name);
     for(i=0;i<nf;i++)
      {
        if(!strcmp(name,fname[i]))
        break;
      }
     if(i==nf)
     {
       strcpy(fname[j++],name);
       nf++;
     }
     else
      printf("There is already %s\n",name);
```

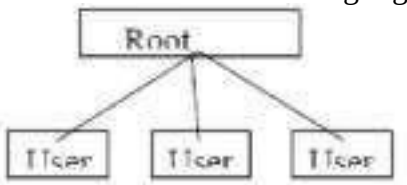
```
      printf("Do you want to enter another file(yes - 1 or no - 0):");
      scanf("%d",&ch);
   }while(ch==1);
 printf("Directory name is:%s\n",mdname);
 printf("Files names are:");
 for(i=0;i<j;i++)
    printf("\n%s",fname[i]);
getch();
}
```

Exp. No:4(b)TWO LEVEL DIRECTORY

AIM: Write a program to simulate Two Level Directory.

Description:

Two Level Directory: The problem in single level directory is different users may be accidentally using the same names for their files. To avoid this problem, each user need a private directory. In this way names chosen by one user don't interface with names chosen by a different user. The following dig 2-level directory



Here root directory is the first level directory it consisting of entries of user directory. User1, User2, User3 are the user levels of directories. A, B, C are the files.

Algorithm for Two Level Directory Structure:

Step 1:Start

Step 2: Initialize structure elements
struct tree_ element char name[20];
Initialize integer variables x, y, ftype, lx, rx, nc, level; struct tree_element *link[5];}typedef structure tree_element node;

Step 3: start main function

Step 4: Step variables gd=DETECT,gm;
node *root;
root=NULL;

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

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

Step 7: end main function

Step 8: Initialize variables i,gap;

Step 9: if check *root==NULL
(*root)=(node*)malloc(sizeof(node));
enter name of ir file name in dname;
fflush(stdin);
gets((*root)->name);

Step 10: if check lev==0||lev==1

(*root)->ftype=1;
else(*root)->ftype=2;
(*root)->level=lev;
(*root)->y=50+lev*5;
(*root)->x=x;
(*root)->lx=lx;
(*root)->rx=rx;

Step 11:for i=0&&i<5
increment i
(*root)->link[i]=NULL;
if check (*root)->ftype==1

Step 12: if check (lev==0||lev==1)
if check(*root)->level==0
print "how many users"
else print"how many files"
print (*root)->name
read (*root)->nc

Step 13:Then (*root)->nc=0;
if check(*root)->nc==0
gap=rx-lx;
else gap=(rx-lx)/(*root)->nc;

Step 14:for i=0&&i<(*root)->nc
increment i;
create(&((*root)->link[i]),lev+1,(*root)->name, lx+gap*i,lx+gap*i+gap,lx+gap*i+gap/2);
then
(*root)->nc=0;

Step 15: Initialize e display function
Initialize i
set textstyle(2,0,4);
set textjustify(1,1);
set fillstyle(1,BLUE);
setcolor(14); step 13:if check root!=NULL

Step 16:for i=0&&i<root->nc
increment i
line(root->x,root->y,root->link[i]->x,root->link[i]->y);

Step 17: if check root->ftype==1 bar3d(root->x-20,root->y-10,root->x+20,root->y+10,0,0); else fill ellipse(root->x,root->y,20,20);
out textxy(root->x,root->y,root->name);

Step 18:for i=0&&i<root->nc
increment i
display(root->link[i]);

Step 19:End

/* Program to simulate two level directory */

```
#include<stdio.h>
#include<graphics.h>
struct tree_element
{
char name[20];
int x,y,ftype,lx,rx,nc,level;
struct tree_element *link[5];
};
typedef struct tree_element node;
void main()
{
int gd=DETECT,gm;
node *root;
root=NULL;
clrscr();
create(&root,0,"null",0,639,320);
clrscr();
initgraph(&gd,&gm,"c:\\tc\\bgi");
display(root);
getch();
closegraph();
}
create(node **root,int lev,char *dname,int lx,int rx,int x)
{
int i,gap;
if(*root==NULL)
{
(*root)=(node*)malloc(sizeof(node));
printf("enter name of dir/file(under %s):",dname);
fflush(stdin);
gets((*root)->name);
if(lev==0||lev==1)
(*root)->ftype=1;
else
(*root)->ftype=2;
(*root)->level=lev;
(*root)->y=50+lev*50;
(*root)->x=x;
(*root)->lx=lx;
(*root)->rx=rx;
for(i=0;i<5;i++)
(*root)->link[i]=NULL;
if((*root)->ftype==1)
{
if(lev==0||lev==1)
{
if((*root)->level==0)
printf("How many users");
else
printf("hoe many files");
printf("(for%s):",(*root)->name);
scanf("%d",&(*root)->nc);
}
```

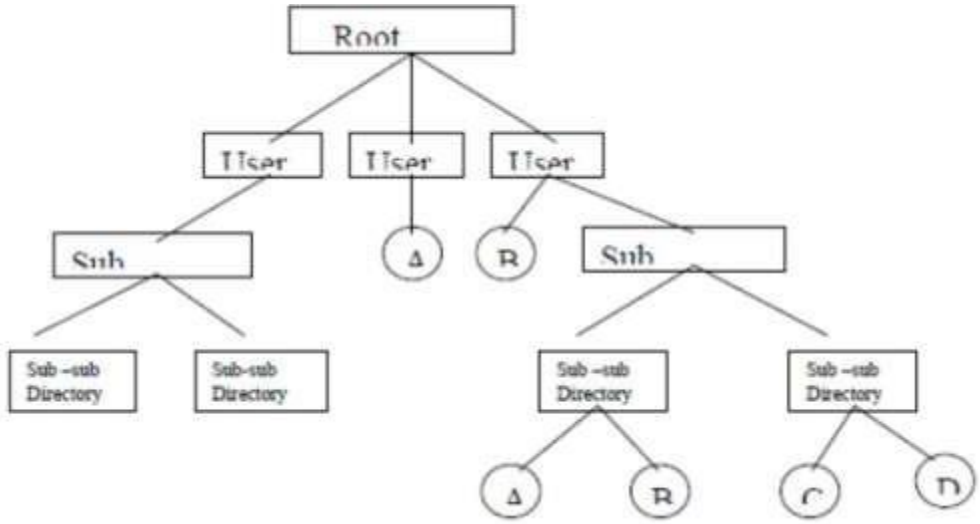
```
else
(*root)->nc=0;
if((*root)->nc==0)
gap=rx-lx;
else
gap=(rx-lx)/(*root)->nc;
for(i=0;i<(*root)->nc;i++)
create(&((*root)->link[i]),lev+1,(*root)-
>name,lx+gap*i,lx+gap*i+gap,lx+gap*i+gap/2);
}
else
(*root)->nc=0;
}
}
display(node *root)
{
int i;
settextstyle(2,0,4);
settextjustify(1,1);
setfillstyle(1,BLUE);
setcolor(14);
if(root!=NULL)
{
for(i=0;i<root->nc;i++)
{
line(root->x,root->y,root->link[i]->x,root->link[i]->y);
}
if(root->ftype==1)
bar3d(root->x-20,root->y-10,root->x+20,root->y+10,0,0);
else
fillellipse(root->x,root->y,20,20);
outtextxy(root->x,root->y,root->name);
for(i=0;i<root->nc;i++)
{
display(root->link[i]);
}
} }
```

Exp. No:4(c)HIERARCHICAL DIRECTORY

AIM: Write a program to simulate Hierarchical Directory.

Description:

Hierarchical Directory: The two level directories eliminate name conflicts among users but it is not satisfactory for users but it is not satisfactory for users with a large no of files. To avoid this, create the subdirectory and load the same type of the files into the subdirectory. So, in this method each can have as many directories are needed.



This directory structure looks like tree, that's why it is also said to be tree-level directory structure.

Algorithm for Hierarchical Directory Structure:

Step 1:Start

Step 2: define structure and declare structure variables

Step 3: start main and declare variables
Node *root
Root = NULL

Step 4: create root null
Initgraph &gd,&gm
Display root

Step 5:create a directory tree structure
If check *root==NULL
Display dir/file mane

Step 6: gets *root->name
*root-> level=lev
*root->y=50+lev*50
*root->x=x

*root->lx=lx
*root->rx = rx

Step7: for i=0 to i<5
Root->link[i]=NULL
Display sub dir/ files

Step8: if check *root->nc==0
Gap=rx-lx
Then
Gap =rx-lx/*root->nc

Step9: for i=0 to i<*root->nc
Then
*rot->nc=0

Step10: display the directory tree in graphical mood
Display nood *root
If check rooy !=NULL

Step 11: foe i=0 to i<root->nc
Line of root->x, root->y, root->link[i]->x, root->link[i]-y

Step12: if check root->ftype==1
Bar3d of root->x-20, root->y-10,root->x+20,root->y+10,0
Then
Display root->link[i]

Step 13: Stop.

/* Program to simulate hierarchical directory */

```
#include<stdio.h>
#include<graphics.h>
struct tree_element
{
char name[20];
int x,y,ftype,lx,rx,nc,level;
struct tree_element *link[5];
};
typedef struct tree_element node;
void main()
{
int gd=DETECT,gm;
node *root;
root=NULL;
clrscr();
create(&root,0,"root",0,639,320);
clrscr();
initgraph(&gd,&gm,"c:\\tc\\BGI");
display(root);
getch();
```

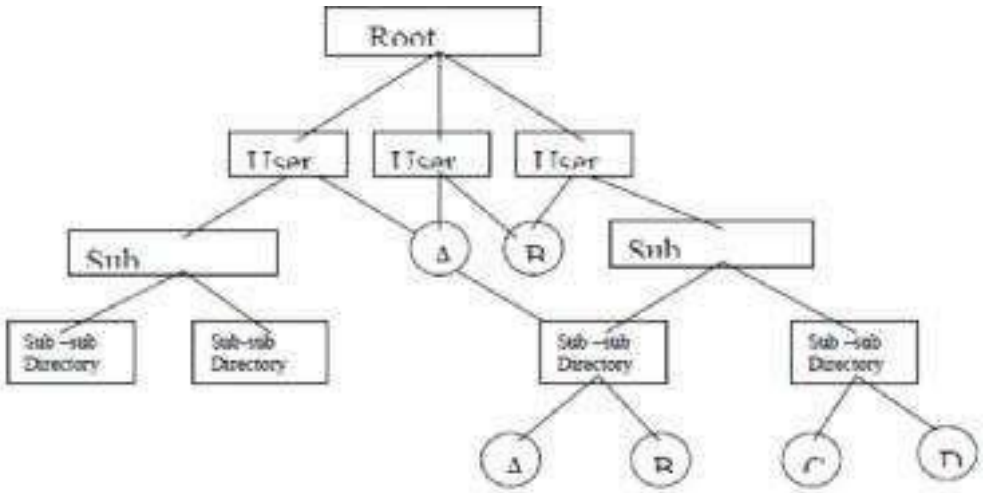
```
closegraph();
}
create(node **root,int lev,char *dname,int lx,int rx,int x)
{
int i,gap;
if(*root==NULL)
{
(*root)=(node *)malloc(sizeof(node));
printf("Enter name of dir/file(under %s) : ",dname);
fflush(stdin);
gets((*root)->name);
printf("enter 1 for Dir/2 for file :");
scanf("%d",&(*root)->ftype);
(*root)->level=lev;
(*root)->y=50+lev*50;
(*root)->x=x;
(*root)->lx=lx;
(*root)->rx=rx;
for(i=0;i<5;i++)
(*root)->link[i]=NULL;
if((*root)->ftype==1)
{
printf("No of sub directories/files(for %s):",(*root)->name);
scanf("%d",&(*root)->nc);
if((*root)->nc==0)
gap=rx-lx;
else
gap=(rx-lx)/(*root)->nc;
for(i=0;i<(*root)->nc;i++)
create(&((*root)->link[i]),lev+1,(*root)-
>name,lx+gap*i,lx+gap*i+gap,lx+gap*i+gap/2);
}
else
(*root)->nc=0;
}
}
display(node *root)
{
int i;
settextstyle(2,0,4);
settextjustify(1,1);
setfillstyle(1,BLUE);
setcolor(14);
if(root !=NULL)
{
for(i=0;i<root->nc;i++)
{
line(root->x,root->y,root->link[i]->x,root->link[i]->y);
}
if(root->ftype==1)
bar3d(root->x-20,root->y-10,root->x+20,root->y+10,0,0);
else
fillellipse(root->x,root->y,20,20);
outtextxy(root->x,root->y,root->name);
for(i=0;i<root->nc;i++)
{
display(root->link[i]);
}
}
}
```

Exp. No:4(d)GENERAL GRAPH DIRECTORY

AIM: Write a program to simulate General Graph Directory.

Description:

General graph Directory: When we add links to an existing tree structured directory, the tree structure is destroyed, resulting in a simple graph structure. This structure is used to traversing is easy and file sharing also possible.



/* Program to General Graph Directory */

```
#include<stdio.h>
#include<conio.h>
#include<graphics.h>
#include<string.h>
struct tree_element
{
char name[20];
int x,y,ftype,lx,rx,nc,level;
struct tree_element *link[5];
};
typedef struct tree_element node;
typedef struct
{
char from[20];
char to[20];
}link;
link L[10];
int nofl;
node * root;
void main()
{
int gd=DETECT,gm;
root=NULL;
clrscr();
create(&root,0,"root",0,639,320);
read_links();
clrscr();
initgraph(&gd,&gm,"c:\\tc\\BGI");
draw_link_lines();
```

```
display(root);
getch();
closegraph();
}
read_links()
{
int i;
printf("how many links");
scanf("%d",&nofl);
for(i=0;i<nofl;i++)
{
printf("File/dir:");
fflush(stdin);
gets(L[i].from);
printf("user name:");
fflush(stdin);
gets(L[i].to);
}
}
draw_link_lines()
{
int i,x1,y1,x2,y2;
for(i=0;i<nofl;i++)
{
search(root,L[i].from,&x1,&y1);
search(root,L[i].to,&x2,&y2);
setcolor(LIGHTGREEN);
setlinestyle(3,0,1);
line(x1,y1,x2,y2);
setcolor(YELLOW);
setlinestyle(0,0,1);
}
}
search(node *root,char *s,int *x,int *y)
{
int i;
if(root!=NULL)
{
if(strcmpi(root->name,s)==0)
{
*x=root->x;
*y=root->y;
return;
}
else
{
for(i=0;i<root->nc;i++)
search(root->link[i],s,x,y);
}
}
}
create(node **root,int lev,char *dname,int lx,int rx,int x)
{
int i,gap;
if(*root==NULL)
{
(*root)=(node *)malloc(sizeof(node));
printf("enter name of dir/file(under %s):",dname);
fflush(stdin);
gets((*root)->name);
```

```
printf("enter 1 for dir/ 2 for file:");
scanf("%d",&(*root)->ftype);
(*root)->level=lev;
(*root)->y=50+lev*50;
(*root)->x=x;
(*root)->lx=lx;
(*root)->rx=rx;
for(i=0;i<5;i++)
(*root)->link[i]=NULL;
if((*root)->ftype==1)
{
printf("no of sub directories /files (for %s):",(*root)->name);
scanf("%d",&(*root)->nc);
if((*root)->nc==0)
gap=rx-lx;
else
gap=(rx-lx)/(*root)->nc;
for(i=0;i<(*root)->nc;i++)
create( & ( (*root)->link[i] ) , lev+1 , (*root)-
>name,lx+gap*i,lx+gap*i+gap,lx+gap*i+gap/2);
}
else
(*root)->nc=0;
}
}
/* displays the constructed tree in graphics mode */
display(node *root)
{
int i;
settextstyle(2,0,4);
settextjustify(1,1);
setfillstyle(1,BLUE);
setcolor(14);
if(root !=NULL)
{
for(i=0;i<root->nc;i++)
{
line(root->x,root->y,root->link[i]->x,root->link[i]->y);
}
if(root->ftype==1)
bar3d(root->x-20,root->y-10,root->x+20,root->y+10,0,0);
else
fillellipse(root->x,root->y,20,20);
outtextxy(root->x,root->y,root->name);
for(i=0;i<root->nc;i++)
{
display(root->link[i]);
}}}
```

Exp. No:5 BANKER'S DEADLOCK AVOIDANCE

Aim: To Simulate Bankers Algorithm for Deadlock Avoidance.

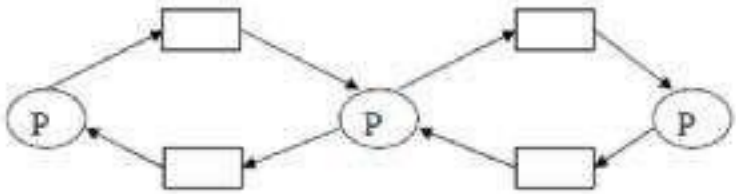
Description:

Deadlock: A process request the resources, the resources are not available at that time, so the process enter into the waiting state. The requesting resources are held by another waiting process,both are in waiting state, this situation is said to be Deadlock.

A deadlocked system must satisfied the following 4 conditions. These are:

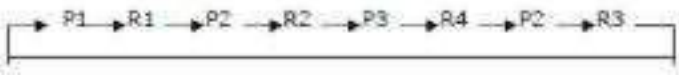
(i) Mutual Exclusion: Mutual Exclusion means resources are in non-sharable mode only, it means only one process at a time can use a process.

(ii) Hold and Wait: Each and every process is the deadlock state, must holding at least one resource and is waiting for additional resources, that are currently being held by another process.



(iii) No Preemption: No Preemption means resources are not released in the middle of the work, they released only after the process has completed its task.

(iv) Circular Wait: If process P1 is waiting for a resource R1, it is held by P2, process P2 is waiting for R2, R2 held by P3, P3 is waiting for R4, R4 is held by P2, P2 waiting for resource R3, it is held by P1.



Deadlock Avoidance: It is one of the method of dynamically escaping from the deadlocks. In this scheme, if a process request for resources, the avoidance algorithm checks before the allocation of resources about the state of system. If the state is safe, the system allocate the resources to the requesting process otherwise (unsafe) do not allocate the resources. So taking care before the allocation said to be deadlock avoidance.

Banker's Algorithm: It is the deadlock avoidance algorithm, the name was chosen because the bank never allocates more than the available cash.

Available: A vector of length „m" indicates the number of available resources of each type. If available[j]=k, there are „k" instances of resource types Rj available.

Allocation: An nxm matrix defines the number of resources of each type currently allocated to each process. If allocation[i,j]=k, then process Pi is currently allocated „k" instances of resources type Rj.

Max: An nxm matrix defines the maximum demand of each process. If max[i,j]=k, then Pi may request at most „k" instances of resource type Rj.

Need: An nxm matrix indicates the remaining resources need of each process. If need[I,j]=k, then Pi may need „k" more instances of resource type Rj to complete this task. There fore, 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
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.

Algorithm for Banker's Deadlock Avoidance:

1. Start the program.
2. Get the values of resources and processes.
3. Get the avail value.
4. After allocation find the need value.
5. Check whether its possible to allocate.
6. If it is possible then the system is in safe state.
7. Else system is not in safety state.
8. If the new request comes then check that the system is in safety.
9. or not if we allow the request.
10. stop the program.

/* Program to Simulate Bankers Algorithm for Dead Lock Avoidance */

```
#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();
                  }
```

Exp.No:6 BANKER'S DEADLOCK PREVENTION

Aim: To Simulate Bankers Algorithm for Deadlock Prevention.

/* Program to Simulate Bankers Algorithm for Dead Lock Prevention */

```
#include<stdio.h>
#include<conio.h>
void main()
{
        char job[10][10];
        int time[10],avail,tem[10],temp[10];
        int safe[10];
        int ind=1,i,j,q,n,t;
        clrscr();
        printf("Enter no of jobs: ");
        scanf("%d",&n);
        for(i=0;i<n;i++)
        {
                printf("Enter name and time: ");
                scanf("%s%d",&job[i],&time[i]);
        }
        printf("Enter the available resources:");
        scanf("%d",&avail);
        for(i=0;i<n;i++)
        {
                temp[i]=time[i];
                tem[i]=i;
        }
        for(i=0;i<n;i++)
                for(j=i+1;j<n;j++)
                {
                        if(temp[i]>temp[j])
                        {
                                t=temp[i];
                                temp[i]=temp[j];
                                temp[j]=t;
                                t=tem[i];
                                tem[i]=tem[j];
                                tem[j]=t;
                         }
                  }
        for(i=0;i<n;i++)
        {
                q=tem[i];
                if(time[q]<=avail)
                {
                        safe[ind]=tem[i];
                        avail=avail-tem[q];
                        //printf("%s",job[safe[ind]]);
```

```
                        ind++;
                    }
                    else
                    {
                        printf("No safe sequence\n");
                    }
                }
                printf("Sa
                fe
                sequence
                is:");

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

                printf(\n",j
                ob[safe[i]],
                time[safe[i
                ]]);
                getch();
}
```

Exp. No: 7(a)FIFO PAGE REPLACEMENT ALGORITHM

AIM: To implement FIFO (First In First Out) page replacement algorithm.

Description:

FIFO (First in First Out) algorithm: FIFO is the simplest page replacement algorithm, the idea behind this is, "Replace a page that page is oldest page of main memory" or "Replace the page that has been in memory longest". FIFO focuses on the length of time a page has been in the memory rather than how much the page is being used.

Algorithm for FIFO Page Replacement:

Step 1: Create a queue to hold all pages in memory
Step 2: When the page is required replace the page at the head of the queue
Step 3: Now the new page is inserted at the tail of the queue

/* Program to simulate FIFO page replacement */

```
#include<stdio.h>
#include<conio.h>
int fr[3],m;
void display();
void main()
{
   int i,j,page[20];
   int flag1=0,flag2=0,pf=0;
   int n, top=0;
   float pr;
   clrscr();
   printf("Enter length of the reference string: ");
   scanf("%d",&n);
   printf("Enter the reference string: ");
   for(i=0;i<n;i++)
       scanf("%d",&page[i]);
   printf("Enter no of frames: ");
   scanf("%d",&m);
   for(i=0;i<m;i++)
      fr[i]=-1;
   for(j=0;j<n;j++)
     {
       flag1=0;
       flag2=0;
       for(i=0;i<m;i++)
          {
            if(fr[i]==page[j])
              {
                flag1=1;
                flag2=1;
                break;
              }
```

```
                }

            if(flag1==0)
              {
                for(i=0;i<m;i++)
                  {
                    if(fr[i]==-1)
                      {
                        fr[i]=page[i];
                        flag2=1;
                        break;
                      }
                  }
              }
            if(flag2==0)
              {
                fr[top]=page[j];
                top++;
                pf++;
                  if(top>=m)
                      top=0;
                }
          display();
       }
     pf+=m;
     printf("Number of page faults : %d\n", pf);
     pr=(float)pf/n*100;
     printf("Page fault rate = %f \n", pr);
     getch();
     }
     void display()
     {
      int i;
      for(i=0;i<m;i++)
          printf("%d\t", fr[i]);
     printf("\n");
}
```

}

Exp. No:7(b)LRU PAGE REPLACEMENT ALGORITHM

AIM: To implement page replacement algorithm LRU (Least Recently Used)

Description:

LRU (Least Recently Used): the criteria of this algorithm is "Replace a page that has been used for the longest period of time". This strategy is the page replacement algorithm looking backward in time, rather than forward.

Algorithm for LRU Page Replacement:

Step 1: Create a queue to hold all pages in memory

Step 2: When the page is required replace the page at the head of the queue

Step 3: Now the new page is inserted at the tail of the queue

Step 4: Create a stack

Step 5: When the page fault occurs replace page present at the bottom of the stack

```
/* Program to simulate LRU page replacement
*/ #include<stdio.h>
#include<conio.h>
void display();
void main()
{
   int i,j,page[20],fs[10],n,m;
   int index,k,l,flag1=0,flag2=0,pf=0;
   int fr[10];
   float pr;
   clrscr();
   printf("Enter length of the reference string: ");
   scanf("%d",&n);
   printf("Enter the reference string: ");
   for(i=0;i<n;i++)
       scanf("%d",&page[i]);
   printf("Enter no of frames: ");
   scanf("%d",&m);
   for(i=0;i<m;i++)
      fr[i]=-1;
   for(j=0;j<n;j++)
     {
       flag1=0;
       flag2=0;
       for(i=0;i<m;i++)
         {
           if(fr[i]==page[j])
             {
               flag1=1;
               flag2=1;
               break;
```

```
                }

            }
        if(flag1==0)
          {
           for(i=0;i<m;i++)
             {
                if(fr[i]==-1)
                  {
                fr[i]=page[j];
                     flag2=1;
                break;
                  }
             }
          }
       if(flag2==0)
          {
             for(i=0;i<m;i++)
               fs[i]=0;
             for(k=j-1,l=1;l<=m;l++,k--)
               {
                  for(i=0;i<m;i++)
                     {
                   if(fr[i]==page[k])
                           fs[i]=1;
                     }
               }
            for(i=0;i<m;i++)
              {
                  if(fs[i]==0)
                index=i;
             }
               fr[index]=page[j];
            pf++;
       }
 for(i=0;i<m;i++)
       printf("%d\t", fr[i]);
 printf("\n");
 }
pf+=m;
printf("Number of page faults : %d\n", pf);
pr=(float)pf/n*100;
printf("Page fault rate = %f \n", pr);
getch();
}
```

AIM: To implement page replacement algorithms

Optimal (The page which is not used for longest time)

Description:

LFU (Least Frequently Used): The least frequently used algorithm "select a page for replacement, if the page has not been used for the often in the past" or "Replace page that page has smallest count" for this algorithm each page maintains as counter which counter value shows the least count, replace that page. The frequency counter is reset each time is page is loaded.

Algorithm for Optimal(LFU) Page Replacement:

Here we select the page that will not be used for the longest period of time.

Step 1: Create an array.

Step 2: When the page fault occurs replace page that will not be used for the longest period of time.

```
/* Program to simulate optimal page replacement */
#include<stdio.h>
#include<conio.h>
int fr[3], n, m;
void display();
void main()
{
   int i,j,page[20],fs[10];
   int max,found=0,lg[3],index,k,l,flag1=0,flag2=0,pf=0;
   float pr;
   clrscr();
   printf("Enter length of the reference string: ");
   scanf("%d",&n);
   printf("Enter the reference string: ");
   for(i=0;i<n;i++)
       scanf("%d",&page[i]);
   printf("Enter no of frames: ");
   scanf("%d",&m);
   for(i=0;i<m;i++)
      fr[i]=-1;
   pf=m;
   for(j=0;j<n;j++)
    {
      flag1=0;
      flag2=0;
      for(i=0;i<m;i++)
        {
          if(fr[i]==page[j])
            {
              flag1=1;
```

```
                flag2=1;
                break;
              }
            }
          if(flag1==0)
            {
              for(i=0;i<m;i++)
                {
                  if(fr[i]==-1)
                    {
                            fr[i]=page[j];
                      flag2=1;
                            break;
                    }
                }
            }
          if(flag2==0)
            {
              for(i=0;i<m;i++)
                lg[i]=0;
              for(i=0;i<m;i++)
                {
                  for(k=j+1;k<=n;k++)
                    {
                      if(fr[i]==page[k])
                        {
                          lg[i]=k-j;
                          break;
                        }
                    }
                }
              found=0;
              for(i=0;i<m;i++)
                {
                  if(lg[i]==0)
                    {
                            index=i;
                      found = 1;
                      break;
                    }
                }
              if(found==0)
                {
                  max=lg[0];
                  index=0;
                  for(i=0;i<m;i++)
                          {
                            if(max<lg[i])
                              {
                                max=lg[i];
```

```
                                index=i;
                               }
                        }
      }
      fr[index]=page[j];
      pf++;
 }
 display();
 }
printf("Number of page faults : %d\n", pf);
pr=(float)pf/n*100;
printf("Page fault rate = %f \n", pr);
getch();
}
void display()
{
   int i;
  for(i=0;i<m;i++)
  printf("%d\t",fr[i]);
  printf("\n");
}
```

Exp. No: 8PAGING TECHNIQUE OF MEMORY MANAGEMENT

AIM: To implement the Memory management policy- Paging.

Description: Paging is an efficient memory management scheme because it is non-contiguous memory allocation method. The basic idea of paging is the physical memory (main memory) is divided into fixed sized blocks called frames, the logical address space is divided into fixed sized blocks, called pages, but page size and frame size should be equal. The size of the frame or a page is depending on operating system.
In this scheme the operating system maintains a data structure that is page table, it is used for mapping purpose. The page table specifies the some useful information, it tells which frames are there and so on. The page table consisting of two fields, one is the page number and other one is frame number. Every address generated by the CPU divided into two parts, one is page number and second is page offset or displacement. The pages are loaded into available free frames in the physical memory.

Algorithm for Paging Technique:

Step 1: Read all the necessary input from the keyboard.
Step 2: Pages - Logical memory is broken into fixed - sized blocks.
Step 3: Frames – Physical memory is broken into fixed – sized blocks.
Step 4: Calculate the physical address using the following
Physical address = (Frame number * Frame size) + offset
Step 5: Display the physical address.
Step 6: Stop the process.

/* Program to simulate paging technique of memory management
***/** #include<stdio.h>

```
void main()
{
   int p, ps, i;
   int *sa;
   clrscr();
   printf("Enter how many pages: ");
   scanf("%d",&np);
   printf("Enter page size: ");
   scanf("%d",&ps);
   for(i=0;i< np;i++)
     {
        sa[i]=(int)malloc(ps);
        printf("Page %d address is %d\n", i, sa[i]);
      }
getch();
}
```