EXPERIMENT: 1

AIM: Simulate the following FCFS CPU Scheduling Algorithm

HARDWARE REQUIREMENTS: Intel based Desktop PC RAM of 512 MB

SOFTWARE REQUIREMENTS: Turbo C/ Borland C/ VS Code.

THEORY:

DESCRIPTION

Assume all the processes arrive at the same time.

For FCFS scheduling algorithm, read the number of processes/jobs in the system, their CPU burst times. The scheduling is performed on the basis of arrival time of the processes irrespective of their other parameters. Each process will be executed according to its arrival time. Calculate the waiting time and turnaround time of each of the processes accordingly.

First-Come, First-Served (FCFS) Scheduling

Process	Burst Time
P1	24
P2	3
P3	3

Suppose that the processes arrive in the order: P1 , P2 , P3 The

Gantt Chart for the schedule is:

0 24 27 30

Waiting time for P1 = 0; P2 = 24; P3 = 27

Average waiting time: (0 + 24 + 27)/3 = 17

ALGORITHM:

1. Start
2. Declare the array size
3. Read the number of processes to be inserted
4. Read the Burst times of processes
5. calculate the waiting time of each process
   wt[i+1]=bt[i]+wt[i]
6. calculate the turnaround time of each process
   tt[i+1]=tt[i]+bt[i+1]
7. Calculate the average waiting time and average turnaround time.
8. Display the values
9. Stop

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int i,j,bt[10],n,wt[10],tt[10],w1=0,t1=0;
float aw,at;
clrscr();
printf("enter no. of processes:\n");
scanf("%d",&n);
printf("enter the burst time of processes:");
for(i=0;i<n;i++)
scanf("%d",&bt[i]);
for(i=0;i<n;i++)
{
wt[0]=0;
tt[0]=bt[0];
wt[i+1]=bt[i]+wt[i];
tt[i+1]=tt[i]+bt[i+1];
w1=w1+wt[i];
t1=t1+tt[i];
}
aw=w1/n;
at=t1/n;
printf("\nbt\t wt\t tt\n");
for(i=0;i<n;i++)
printf("%d\t %d\t %d\n",bt[i],wt[i],tt[i]);
```

```
printf("aw=%f\n,at=%f\n",aw,at);
getch();
    }
```

INPUT

```
Enter the number of processes -     3
Enter Burst Time for Process 0-     24
Enter Burst Time for Process 1-     3
Enter Burst Time for Process 2-     3
```

OUTPUT

PROCESS	BURST TIME	WAITING TIME	TURNAROUND TIME
P0	24	0	24
P1	3	24	27
P2	3	27	30

Average Waiting Time- 17.000000
Average Turnaround Time- 27.000000

EXPERIMENT 2

AIM: Simulate the following CPU Scheduling Algorithms SJF

HARDWARE REQUIREMENTS: Intel based Desktop Pc RAM of 512 MB

SOFTWARE REQUIREMENTS: Turbo C/ Borland C.

THEORY:

For SJF scheduling algorithm, read the number of processes/jobs in the system, their CPU burst times. Arrange all the jobs in order with respect to their burst times. There may be two jobs in queue with the same execution time, and then FCFS approach is to be performed. Each process will be executed according to the length of its burst time. Then calculate the waiting time and turnaround time of each of the processes accordingly.

ALGORITHM

1. Start
2. Declare the array size
3. Read the number of processes to be inserted
4. Read the Burst times of processes
5. sort the Burst times in ascending order and process with shortest burst time is first executed.
6. calculate the waiting time of each process
   wt[i+1]=bt[i]+wt[i]
7. calculate the turnaround time of each process
   tt[i+1]=tt[i]+bt[i+1]
8. Calculate the average waiting time and average turnaround time.
9. Display the values
10. Stop

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int i,j,bt[10],t,n,wt[10],tt[10],w1=0,t1=0;
float aw,at;
clrscr();
printf("enter no. of processes:\n");
scanf("%d",&n);
printf("enter the burst time of processes:");
for(i=0;i<n;i++)
scanf("%d",&bt[i]);
for(i=0;i<n;i++)
{

for(j=i;j<n;j++)
if(bt[i]>bt[j])

{
t=bt[i];
bt[i]=bt[j];
bt[j]=t;
}

for(i=0;i<n;i++)
printf("%d",bt[i]);
for(i=0;i<n;i++)

{

wt[0]=0;
tt[0]=bt[0];
wt[i+1]=bt[i]+wt[i];
tt[i+1]=tt[i]+bt[i+1];
w1=w1+wt[i];
t1=t1+tt[i];

}
aw=w1/n;
at=t1/n;
printf("\nbt\t wt\t tt\n");
for(i=0;i<n;i++)
printf("%d\t %d\t %d\n",bt[i],wt[i],tt[i]);
printf("aw=%f\n,at=%f\n",aw,at);
getch();

}
```

INPUT

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

OUTPUT

PROCESS	BURST TIME	WAITING TIME	TURNAROUND TIME
P3	3	0	3
P0	6	3	9
P2	7	9	16
P1	8	16	24

Average Waiting Time -- 7.000000
Average Turnaround Time -- 13.000000

EXPERIMENT: 3

AIM: Simulate the following Round Robin CPU Scheduling Algorithms

HARDWARE REQUIREMENTS: Intel based Desktop Pc RAM of 512 MB

SOFTWARE REQUIREMENTS:Turbo C/ Borland C.

THEORY:

For round robin scheduling algorithm, read the number of processes/jobs in the system, their CPU burst times, and the size of the time slice. Time slices are assigned to each process in equal portions and in circular order, handling all processes execution. This allows every process to get an equal chance. Calculate the waiting time and turnaround time of each of the processes accordingly.

ALGORITHM

1. Start
2. Declare the array size
3. Read the number of processes to be inserted
4. Read the burst times of the processes
5. Read the Time Quantum
6. if the burst time of a process is greater than time Quantum then subtract time quantum form the burst time

   Else

   Assign the burst time to time quantum.
7. calculate the average waiting time and turn around time of the processes.
8. Display the values
9. Stop

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int st[10],bt[10],wt[10],tat[10],n,tq;
int i,count=0,swt=0,stat=0,temp,sq=0;
float awt=0.0,atat=0.0;

clrscr();
printf("Enter number of processes:");
scanf("%d",&n);
printf("Enter burst time for sequences:");

for(i=0;i<n;i++)
{
scanf("%d",&bt[i]);
st[i]=bt[i];
}
printf("Enter time quantum:");
scanf("%d",&tq);
while(1)
{

for(i=0,count=0;i<n;i++)
{
temp=tq;
if(st[i]==0)

{
count++;
continue;
}
if(st[i]>tq)
st[i]=st[i]-tq;
else
if(st[i]>=0)
{
temp=st[i];
st[i]=0;
}
sq=sq+temp;
tat[i]=sq;
}
if(n==count)

break;
```

```
}

for(i=0;i<n;i++)
{
wt[i]=tat[i]-bt[i];
swt=swt+wt[i];
stat=stat+tat[i];
}
awt=(float)swt/n;
atat=(float)stat/n;
printf("Process_no Burst time Wait time Turn around time");

for(i=0;i<n;i++)
printf("\n%d\t %d\t %d\t %d",i+1,bt[i],wt[i],tat[i]);
printf("\nAvg wait time is %f Avg turn around time is %f",awt,atat);
getch();

}
```

INPUT

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

Enter the size of time slice – 3

OUTPUT

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

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

EXPERIMENT: 4

AIM: Simulate the following Priority CPU Scheduling

HARDWARE REQUIREMENTS: Intel based Desktop Pc RAM of 512 MB

SOFTWARE REQUIREMENTS: Turbo C/ Borland C.

THEORY:

In Priority Scheduling, each process is given a priority, and higher priority methods are executed first, while equal priorities are executed First Come First Served or Round Robin. There are several ways that priorities can be assigned:

- Internal priorities are assigned by technical quantities such as memory usage, and file/IO operations.
- External priorities are assigned by politics, commerce, or user preference, such as importance and amount being paid for process access (the latter usually being for mainframes).

ALGORITHM

1. Start
2. Declare the array size
3. Read the number of processes to be inserted
4. Read the Priorities of processes
5. sort the priorities and Burst times in ascending order
5. calculate the waiting time of each process
   wt[i+1]=bt[i]+wt[i]
6. calculate the turnaround time of each process
   tt[i+1]=tt[i]+bt[i+1]
6. Calculate the average waiting time and average turnaround time.
7. Display the values
8. Stop

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
void main()
 {
 int i,j,pno[10],prior[10],bt[10],n,wt[10],tt[10],w1=0,t1=0,s;
 float aw,at;
 clrscr();
 printf("enter the number of processes:");
 scanf("%d",&n);
 for(i=0;i<n;i++)
 {
 printf("The process %d:\n",i+1);
  printf("Enter the burst time of processes:");
  scanf("%d",&bt[i]);
   printf("Enter the priority of processes %d:",i+1);
   scanf("%d",&prior[i]);
   pno[i]=i+1;
 }
   for(i=0;i<n;i++)
    {
     for(j=0;j<n;j++)
     {
      if(prior[i]<prior[j])
      {
            s=prior[i];
            prior[i]=prior[j];
            prior[j]=s;

             s=bt[i];
             bt[i]=bt[j];
             bt[j]=s;

              s=pno[i];
              pno[i]=pno[j];
              pno[j]=s;
             }
            }
     }
     for(i=0;i<n;i++)
     {
       wt[0]=0;
       tt[0]=bt[0];
           wt[i+1]=bt[i]+wt[i];
           tt[i+1]=tt[i]+bt[i+1];
           w1=w1+wt[i];
           t1=t1+tt[i];
           aw=w1/n;
```

```
        at=t1/n;
        }
        printf(" \n job \t bt \t wt \t tat \t prior\n");
         for(i=0;i<n;i++)
          printf("%d \t %d \t %d\t %d\t %d\n",pno[i],bt[i],wt[i],tt[i],prior[i]);
         printf("aw=%f \t at=%f \n",aw,at); getch();
}
```

INPUT

```
Enter the number of processes --  5
Enter the Burst Time & Priority of Process 0 --- 10          3
Enter the Burst Time & Priority of Process 1 --- 1           1
Enter the Burst Time & Priority of Process 2 --- 2           4
Enter the Burst Time & Priority of Process 3 --- 1           5
Enter the Burst Time & Priority of Process 4 --- 5           2
```

OUTPUT

PROCESS	PRIORITY	BURST TIME	WAITING TIME	TURNAROUND TIME
1	1	1	0	1
4	2	5	1	6
0	3	10	6	16
2	4	2	16	18
3	5	1	18	19

```
Average Waiting Time is ---        8.200000
Average Turnaround Time is         --- 12.000000
```

EXPERIMENT 5

AIM: Write a C program to simulate the following contiguous memory allocation techniques

a) Worst-fit b) Best-fit c) First-fit

HARDWARE REQUIREMENTS: Intel based Desktop Pc RAM of 512 MB

SOFTWARE REQUIREMENTS: Turbo C/ Borland C.

THEORY:

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

ALGORITHM:

1. Start the process
2. Declare the size
3. Get the number of processes to be inserted
4. Allocate the first hole that is big enough searching
5. Start at the beginning of the set of holes
6. If not start at the hole that is sharing the pervious first fit search end
7. Compare the hole
8. if large enough then stop searching in the procedure

9. Display the values

10. Stop the process

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
#define max 25

void main()
{
        int frag[max],b[max],f[max],i,j,nb,nf,temp,highest=0;
        static int bf[max],ff[max];
        clrscr();

        printf("\n\tMemory Management Scheme - Worst
        Fit"); printf("\nEnter the number of blocks:");
        scanf("%d",&nb);
        printf("Enter the number of files:");
        scanf("%d",&nf);
        printf("\nEnter the size of the blocks:-\n");
        for(i=1;i<=nb;i++)
        {
                printf("Block %d:",i);
                scanf("%d",&b[i]);
        }
        printf("Enter the size of the files :-\n");
        for(i=1;i<=nf;i++)
        {
                printf("File %d:",i);
                scanf("%d",&f[i]);
        }
        for(i=1;i<=nf;i++)
        {
                for(j=1;j<=nb;j++)
                {
                        if(bf[j]!=1)      //if bf[j] is not allocated
                        {
                                temp=b[j]-f[i];
                                if(temp>=0)
                                        if(highest<temp)
                                        {
                                                ff[i]=j;
```

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

INPUT

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

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

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

OUTPUT

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

EXPERIMENT 6

AIM: Write a program for page replacement policy using a) LRU b) FIFO c) Optimal.

HARDWARE REQUIREMENTS: Intel based Desktop Pc RAM of 512 MB

SOFTWARE REQUIREMENTS: Turbo C/ Borland C.

THEORY:

Page replacement is basic to demand paging. It completes the separation between logical memory and physical memory. With this mechanism, an enormous virtual memory can be provided for programmers on a smaller physical memory. There are many different page-replacement algorithms. Every operating system probably has its own replacement scheme. A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen. If the recent past is used as an approximation of the near future, then the page that has not been used for the longest period of time can be replaced. This approach is the Least Recently Used (LRU) algorithm. LRU replacement associates with each page the time of that page's last use. When a page must be replaced, LRU chooses the page that has not been used for the longest period of time. Least frequently used (LFU) page-replacement algorithm requires that the page with the smallest count be replaced. The reason for this selection is that an actively used page should have a large reference count.

ALGORITHM:

1. First of all, find the location of the desired page on the disk.
2. Find a free Frame:
    a) If there is a free frame, then use it.
    b) If there is no free frame then make use of the page-replacement algorithm in order to select the victim frame.
    c) Then after that write the victim frame to the disk and then make the changes in the page table and frame table accordingly.
3. After that read the desired page into the newly freed frame and then change the page and frame tables.
4. Restart the process.

PROGRAM:

(1) FIFO PAGE REPLACEMENT ALGORITHM

```
#include<stdio.h>
#include<conio.h>
main()
{
        int i, j, k, f, pf=0, count=0, rs[25], m[10], n;
        clrscr();
        printf("\n Enter the length of reference string -- ");
        scanf("%d",&n);
        printf("\n Enter the reference string -- ");
        for(i=0;i<n;i++)
                scanf("%d",&rs[i]);
        printf("\n Enter no. of frames -- ");
        scanf("%d",&f);
         for(i=0;i<f;i++)
                m[i]=-1;

        printf("\n The Page Replacement Process is -- \n");
        for(i=0;i<n;i++)
        {
                for(k=0;k<f;k++)
                {
                        if(m[k]==rs[i])
                                break;
                }
                if(k==f)
                {
                        m[count++]=rs[i];
                        pf++;
                }
                for(j=0;j<f;j++)
                        printf("\t%d",m[j]);
                if(k==f)
                        printf("\tPF No. %d",pf);
                printf("\n");
                if(count==f)
                        count=0;
        }
        printf("\n The number of Page Faults using FIFO are %d",pf);
        getch();
        }
```

INPUT

Enter the length of reference string – 20
Enter the reference string -- 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1
Enter no. of frames -- 3

OUTPUT

The Page Replacement Process is –

7	-1	-1	PF No. 1
7	0	-1	PF No. 2
7	0	1	PF No. 3
2	0	1	PF No. 4
2	0	1	
2	3	1	PF No. 5
2	3	0	PF No. 6
4	3	0	PF No. 7
4	2	0	PF No. 8
4	2	3	PF No. 9
0	2	3	PF No. 10
0	2	3	
0	2	3	
0	1	3	PF No. 11
0	1	2	PF No. 12
0	1	2	
0	1	2	
7	1	2	PF No. 13
7	0	2	PF No. 14
7	0	1	PF No. 15

The number of Page Faults using FIFO are 15

(2) **LRU PAGE REPLACEMENT ALGORITHM**

```
#include<stdio.h>
#include<conio.h>
main()
{
        int i, j , k, min, rs[25], m[10], count[10], flag[25], n, f,
        pf=0, next=1; clrscr();
        printf("Enter the length of reference string -- ");
        scanf("%d",&n);
        printf("Enter the reference string -- ");
        for(i=0;i<n;i++)
        {
                scanf("%d",&rs[i]);
                flag[i]=0;
        }
        printf("Enter the number of frames -- ");
        scanf("%d",&f);
        for(i=0;i<f;i++)
        {
```

```
                count[i]=0;
                m[i]=-1;
        }
        printf("\nThe Page Replacement process is -- \n");
        for(i=0;i<n;i++)
        {
                for(j=0;j<f;j++)
                {
                        if(m[j]==rs[i])
                        {
flag[i]=1;

                                count[j]=next;
                                next++;
                        }

                }
                if(flag[i]==0)
                {
                        if(i<f)
                        {
                                m[i]=rs[i];
                                count[i]=next;
                                next++;
                        }
                        else
                        {
                                min=0;
                                for(j=1;j<f;j++)
                                        if(count[min] > count[j])
                                                min=j;

                                m[min]=rs[i];
                                count[min]=next;
                                next++;
                        }
                        pf++;
                }
                for(j=0;j<f;j++)
                        printf("%d\t", m[j]);
                if(flag[i]==0)
                        printf("PF No. -- %d" , pf);
                printf("\n");
        }
        printf("\nThe number of page faults using LRU are
        %d",pf); getch();
}
```

INPUT

Enter the length of reference string -- 20
Enter the reference string -- 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1
Enter the number of frames -- 3

OUTPUT

The Page Replacement process is --

7	-1	-1	PF No. -- 1
7	0	-1	PF No. -- 2
7	0	1	PF No. -- 3
2	0	1	PF No. -- 4
2	0	1	
2	0	3	PF No. -- 5
2	0	3	
4	0	3	PF No. -- 6
4	0	2	PF No. -- 7
4	3	2	PF No. -- 8
0	3	2	PF No. -- 9
0	3	2	
0	3	2	
1	3	2	PF No. -- 10
1	3	2	
1	0	2	PF No. -- 11
1	0	2	
1	0	7	PF No. -- 12
1	0	7	
1	0	7	

The number of page faults using LRU are 12

(3) **LFU PAGE REPLACEMENT ALGORITHM**

```
#include<stdio.h>
#include<conio.h>

main()
{
        int rs[50], i, j, k, m, f, cntr[20], a[20], min, pf=0;
        clrscr();
        printf("\nEnter number of page references -- ");
        scanf("%d",&m);

        printf("\nEnter the reference string -- ");
        for(i=0;i<m;i++)
                scanf("%d",&rs[i]);

        printf("\nEnter the available no. of frames -- ");
        scanf("%d",&f);

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

```
        {
                cntr[i]=0;
                a[i]=-1;
        }
        Printf("\nThe Page Replacement Process is – \n");
        for(i=0;i<m;i++)
        {

                for(j=0;j<f;j++)
                        if(rs[i]==a[j])
                        {
                                cntr[j]++;
                                break;
                        }
                if(j==f)
                {
                        min = 0;
                        for(k=1;k<f;k++)
                                if(cntr[k]<cntr[min])
                                        min=k;
                        a[min]=rs[i];
                        cntr[min]=1;
                        pf++;
                }
                printf("\n");
                for(j=0;j<f;j++)
                        printf("\t%d",a[j]);
                if(j==f)
                        printf("\tPF No. %d",pf);
        }
        printf("\n\n Total number of page faults -- %d",pf);
        getch();
}
```

INPUT

Enter number of page references -- 10
Enter the reference string -- 1 2 3 4 5 2 5 2 5 1 4 3
Enter the available no. of frames -- 3

OUTPUT

The Page Replacement Process is –

1	-1	-1	PF No. 1
1	2	-1	PF No. 2
1	2	3	PF No. 3
4	2	3	PF No. 4
5	2	3	PF No. 5
5	2	3	
5	2	3	
5	2	1	PF No. 6
5	2	4	PF No. 7

5 2 3 PF No. 8

Total number of page faults -- 8

(4) **OPTIMAL PAGE REPLACEMENT ALGORITHM**

```
#include<stdio.h>
#include<conio.h>

    int n;
    main()
    {
            int seq[30],fr[5],pos[5],find,flag,max,i,j,m,k,t,s;
            int count=1,pf=0,p=0;
            float pfr;
            clrscr();
            printf("Enter maximum limit of the sequence: ");
            scanf("%d",&max);
            printf("\nEnter the sequence: ");
            for(i=0;i<max;i++)
                    scanf("%d",&seq[i]);
                    printf("\nEnter no. of frames: ");
                    scanf("%d",&n);
                    fr[0]=seq[0];
                    pf++;
                    printf("%d\t",fr[0]);
                    i=1;
                    while(count<n)
                    {
                            flag=1;
                            p++;
                            for(j=0;j<i;j++)
                            {
                                    if(seq[i]==seq[j])
                                    flag=0;
                            }
                            if(flag!=0)
                            {
                                    fr[count]=seq[i];
                                    printf("%d\t",fr[count]);
                                    count++;
                                    pf++;
                            }
                            i++;
                    }
                    printf("\n");
```

```
for(i=p;i<max;i++)
{
        flag=1;
        for(j=0;j<n;j++)
        {
                if(seq[i]==fr[j])
                        flag=0;
        }
        if(flag!=0)
        {
                for(j=0;j<n;j++)
                {
                        m=fr[j];
                        for(k=i;k<max;k++)
                        {
                                if(seq[k]==m)
                                {
                                        pos[j]=k;
                                        break;
                                }
                                else
                                        pos[j]=1;
                          }
                }
                for(k=0;k<n;k++)
                {
                        if(pos[k]==1)
                                flag=0;
                }
                if(flag!=0)
                        s=findmax(pos);
                if(flag==0)
                {
                        for(k=0;k<n;k++)
                        {
                                if(pos[k]==1)
                                {
                                        s=k;
                                        break;
                                }
                          }
                }
                fr[s]=seq[i];
                for(k=0;k<n;k++)
                        printf("%d\t",fr[k]);
                pf++;
                printf("\n");
```

```
                        }
                }
                pfr=(float)pf/(float)max;
                printf("\nThe no. of page faults are %d",pf);
                printf("\nPage fault rate %f",pfr);
                getch();
}
int findmax(int a[])
{
        int max,i,k=0;
        max=a[0];
        for(i=0;i<n;i++)
        {
                if(max<a[i])
                {
                        max=a[i];
                        k=i;
                }
        }
        return k;
}
```

INPUT

Enter number of page references -- 10
Enter the reference string -- 1 2 3 4 5 2 5 2 5 1 4 3
Enter the available no. of frames -- 3

OUTPUT

The Page Replacement Process is –

1	-1	-1	PF No. 1
1	2	-1	PF No. 2
1	2	3	PF No. 3
4	2	3	PF No. 4
5	2	3	PF No. 5
5	2	3	
5	2	3	
5	2	1	PF No. 6
5	2	4	PF No. 7
5	2	3	PF No. 8

Total number of page faults -- 8

EXPERIMENT 7

AIM: Write a program to implement Banker's algorithm for deadlock avoidance.

HARDWARE REQUIREMENTS: Intel based Desktop Pc, RAM of 512 MB

SOFTWARE REQUIREMENTS: Turbo C/ Borland C.

THEORY:

When a new process enters a system, it must declare the maximum number of instances of each resource type it needed. This number may exceed the total number of resources in the system. When the user request a set of resources, the system must determine whether the allocation of each resources will leave the system in safe state. If it will the resources are allocation; otherwise the process must wait until some other process release the resources.

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

Safety Algorithm

Work and Finish be the vector of length m and n respectively, Work=Available and Finish[i] =False.

Find an i such that both
Finish[i] =False
Need<=Work
If no such I exists go to step 4.
work=work Allocation, Finish[i] =True;
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.

if Request<=Need I go to step 2. Otherwise raise an error condition.
if Request<=Available go to step 3. Otherwise Pi must since the resources are available.

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:

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:

```
#include<stdio.h>
#include<conio.h>
struct da {
int max[10],al[10],need[10],before[10],after[10];
}p[10];

void main() {

int i,j,k,l,r,n,tot[10],av[10],cn=0,cz=0,temp=0,c=0; clrscr();
printf("\n Enter the no of processes:");
scanf("%d",&n);
```

```
printf("\n Enter the no of resources:");
scanf("%d",&r);
for(i=0;i<n;i++) {
printf("process %d \n",i+1);
for(j=0;j<r;j++) {
printf("maximum value for resource %d:",j+1);

scanf("%d",&p[i].max[j]);
}
for(j=0;j<r;j++) {
printf("allocated from resource %d:",j+1);

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

p[i].need[j]=p[i].max[j]-p[i].al[j];
}
}
for(i=0;i<r;i++) {

printf("Enter total value of resource %d:",i+1);
scanf("%d",&tot[i]);
}
for(i=0;i<r;i++) {
for(j=0;j<n;j++)
temp=temp+p[j].al[i];

av[i]=tot[i]-temp;
temp=0;
}
printf("\n\t max allocated needed total avail");
for(i=0;i<n;i++) {
printf("\n P%d \t",i+1);

for(j=0;j<r;j++)
printf("%d",p[i].max[j]);
printf("\t");
for(j=0;j<r;j++)
printf("%d",p[i].al[j]);
printf("\t");
for(j=0;j<r;j++)
printf("%d",p[i].need[j]);
printf("\t");
for(j=0;j<r;j++)
{
if(i==0)
printf("%d",tot[j]);
}
```

```
printf(" ");
for(j=0;j<r;j++) {
if(i==0)

printf("%d",av[j]);
}
}
printf("\n\n\t AVAIL BEFORE \t AVAIL AFTER");

for(l=0;l<n;l++)
{
for(i=0;i<n;i++)
{
for(j=0;j<r;j++)
{
if(p[i].need[j]>av[j])

cn++;
if(p[i].max[j]==0)
cz++;
}
if(cn==0 && cz!=r)
{
for(j=0;j<r;j++)
{
p[i].before[j]=av[j]-p[i].need[j];
p[i].after[j]=p[i].before[j]+p[i].max[j];
av[j]=p[i].after[j];
p[i].max[j]=0;
}
printf("\n p%d \t",i+1);
for(j=0;j<r;j++)
printf("%d",p[i].before[j]);
printf("\t");
for(j=0;j<r;j++)
printf("%d",p[i].after[j]);
cn=0;
cz=0;
c++;
break;
}
else {
cn=0;cz=0;
}

}
}
```

```
if(c==n)
printf("\n the above sequence is a safe sequence"); else
printf("\n deadlock occured");
getch();
}
```

OUTPUT:

```
//TEST CASE 1:

 ENTER THE NO. OF PROCESSES:4

 ENTER THE NO. OF RESOURCES:3

PROCESS 1
MAXIMUM VALUE FOR RESOURCE 1:3
MAXIMUM VALUE FOR RESOURCE 2:2
MAXIMUM VALUE FOR RESOURCE 3:2
ALLOCATED FROM RESOURCE 1:1
ALLOCATED FROM RESOURCE 2:0
ALLOCATED FROM RESOURCE 3:0

PROCESS 2
MAXIMUM VALUE FOR RESOURCE 1:6
MAXIMUM VALUE FOR RESOURCE 2:1
MAXIMUM VALUE FOR RESOURCE 3:3
ALLOCATED FROM RESOURCE 1:5
ALLOCATED FROM RESOURCE 2:1
ALLOCATED FROM RESOURCE 3:1

PROCESS 3
MAXIMUM VALUE FOR RESOURCE 1:3
MAXIMUM VALUE FOR RESOURCE 2:1
MAXIMUM VALUE FOR RESOURCE 3:4
ALLOCATED FROM RESOURCE 1:2
ALLOCATED FROM RESOURCE 2:1
ALLOCATED FROM RESOURCE 3:1

PROCESS 4
MAXIMUM VALUE FOR RESOURCE 1:4
MAXIMUM VALUE FOR RESOURCE 2:2
MAXIMUM VALUE FOR RESOURCE 3:2
ALLOCATED FROM RESOURCE 1:0
ALLOCATED FROM RESOURCE 2:0
ALLOCATED FROM RESOURCE 3:2
ENTER TOTAL VALUE OF RESOURCE 1:9
ENTER TOTAL VALUE OF RESOURCE 2:3
ENTER TOTAL VALUE OF RESOURCE 3:6



 RESOURCES   ALLOCATED  NEEDED        TOTAL  AVAIL
 P1      322        100      222              936     112
 P2      613        511      102
 P3      314        211      103
 P4      422        002      420

           AVAIL  BEFORE  AVAIL  AFTER
 P 2          010     623
 P 1          401     723
```

P 3	620	934
P 4	514	936

THE ABOVE SEQUENCE IS A SAFE SEQUENCE

EXPERIMENT 8

AIM: Write a program to implement reader/writer problem using semaphore.

HARDWARE REQUIREMENTS: Intel based Desktop Pc, RAM of 512 MB

SOFTWARE REQUIREMENTS: Turbo C/ Borland C.

THEORY:

The readers-writers problem relates to an object such as a file that is shared between multiple processes. Some of these processes are readers i.e. they only want to read the data from the object and some of the processes are writers i.e. they want to write into the object.
The readers-writers problem is used to manage synchronization so that there are no problems with the object data. For example - If two readers access the object at the same time there is no problem. However if two writers or a reader and writer access the object at the same time, there may be problems.
To solve this situation, a writer should get exclusive access to an object i.e. when a writer is accessing the object, no reader or writer may access it. However, multiple readers can access the object at the same time.
This can be implemented using semaphores.

Algorithm:

1. Reader will run after Writer because of read semaphore.
2. Writer will stop writing when the write semaphore has reached 0.
3. Reader will stop reading when the read semaphore has reached 0.

In writer, the value of write semaphore is given to read semaphore and in reader, the value of read is given to write on completion of the loop.

Program:

```
#include<semaphore.h>
#include<stdio.h>
#include<stdlib.h>
sem_t x,y;
pthread_t tid;
pthread_t writerthreads[100],readerthreads[100];
int readercount;
```

```
void *reader(void* param)
{
   sem_wait(&x);
   readercount++;
   if(readercount==1)
   sem_wait(&y);
   sem_post(&x);
   printf("\n%d reader is inside",readercount);
   sem_wait(&x);
   readercount--;
   if(readercount==0)
   {
      sem_post(&y);
   }
   sem_post(&x);
   printf("\n%d Reader is leaving",readercount+1);
}

void *writer(void* param)
{
   printf("\nWriter is trying to enter");
   sem_wait(&y);
   printf("\nWriter has entered");
   sem_post(&y);
   printf("\nWriter is leaving");
}

int main()
{
   int n2,i;
   printf("Enter the number of readers:");
   scanf("%d",&n2);
   int n1[n2];
   sem_init(&x,0,1);
   sem_init(&y,0,1);
   for(i=0;i<n2;i++)
   {
      pthread_create(&writerthreads[i],NULL,reader,NULL);
      pthread_create(&readerthreads[i],NULL,writer,NULL);
   }
   for(i=0;i<n2;i++)
   {
      pthread_join(writerthreads[i],NULL);
      pthread_join(readerthreads[i],NULL);
   }

}
```

Output:

reader is inside
reader is leaving
reader is inside
reader is leaving
writer is trying to enter
writer has entered
writer is leaving