Operating System CS304 LAB ASSIGNMENT

SUBHOJEET GHOSH

BTECH/60017/19

PROFESSOR – MR RAYEES AHAMED KHAN

Assignment 1

Write a C program to implement First come first serve or FCFS process scheduling .

PROGRAM DESCRIPTION:-

FCFS process scheduling is a non-preemptive scheduling algorithm where the processes are scheduled or executed on the priority basis of their arrival time. The word non-preemptive means that the process which has entered the ready queue has to run along its full execution time. This means that the as long as the current process is executing no other process should get executed but rather be arranged in the ready queue based on the priority of their arrival time. As soon as the current process gets executed , the next process in the ready queue gets into execution. And thus this is algorithm goes on until all the processes in the ready queue gets executed. Therefore, this is how FCFS process scheduling works.

VARIABLE DESCRIPTION:

Variable name	Variable type	Variable Description
i , j	int	Loop variables
n	int	Input and store the number of processes
temp	int	Temporary variable used for swapping array elements/Also used as the time counter for calculating the current time of the scheduling process
at[]	int	Array to store the arrival time of the processes
bt[]	int	Array to store the burst time of the processes
tat[]	int	Array to store the turn around time of the processes
comp[]	int	Array to store the completion time of the processes
wt[]	int	Array to store the waiting time of the processes
index[]	int	Array to store the process id of the given processes
awt	float	To store the average waiting of all the processes
atat	float	To store the average turn around time of all the process

CODE FOR FCFS SCHEDULING ALGORITHM:-

```c
#include <stdio.h>
#include <conio.h>
#define max 30
void main()
{
  int i,j,n,temp,at[max],bt[max],wt[max],tat[max],comp[max],index[max];
  float awt=0,atat=0;
  printf("Enter the number of process --> ");
  scanf("%d",&n);
  printf("Enter the Burst time of the process\n");
  for(i=0;i<n;i++)
  {
    scanf("%d",&bt[i]);
    index[i]=i+1;
  }
  printf("\nEnter the arrival time of the process\n");
  temp=0;
  for(i=0;i<n;i++)
  {
    scanf("%d",&at[i]);
  }
  for(i=0;i<n;i++)
  {
    for(j=0;j<n-i-1;j++)
    {
      if(at[j]>at[j+1])
      {
        temp=at[j];
        at[j]=at[j+1];
        at[j+1]=temp;
        temp=bt[j];
        bt[j]=bt[j+1];
        bt[j+1]=temp;
        temp=index[j];
        index[j]=index[j+1];
        index[j+1]=temp;

      }
    }
  }
 temp=0;
printf("PROCESS\tARRIVAL TIME\tBURST TIME\tCOMPLETION TIME\tWAITING TIME\tTURN AROUND TIME\n");
  for(i=0;i<n;i++)
  {
    wt[i]=0;
    tat[i]=0;
    comp[i]=0;
    if(at[i]>temp)
    {
      temp=temp+1;
      i--;
      continue;
    }
    temp=temp+bt[i];
    comp[i]=temp;
    tat[i]=comp[i]-at[i];
    wt[i]=tat[i]-bt[i];
    awt=awt+wt[i];
    atat=atat+tat[i];
    printf("%d\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n",index[i],at[i],bt[i],comp[i],wt[i],tat[i]);

  }
  printf("Average waiting time => %f",awt/n);
  printf("\nAverage Turn around time => %f",atat/n);


}
```

SAMPLE INPUT:-

Enter the number of process --> 4

Enter the Burst time of the process

2 2 3 4

Enter the arrival time of the process

0 1 5 6

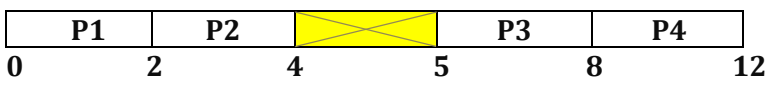
SAMPLE OUTPUT:-

PROCESS	ARRIVAL TIME	BURST TIME	COMPLETION TIME	WAITING TIME	TURN AROUND TIME
1	0	2	2	0	2
2	1	2	4	1	3
3	5	3	8	0	3
4	6	4	12	2	6

Average waiting time => 0.750000

Average Turn around time => 3.500000

GANTT CHART :



SNAPSHOT OF THE OUTPUT TERMINAL:-

```
Enter the number of process --> 4
Enter the Burst time of the process
2
2
3
4

Enter the arrival time of the process
0
1
5
6
PROCESS ARRIVAL TIME     BURST TIME     COMPLETION TIME WAITING TIME    TURN AROUND TIME
1       0               2              2               0               2
2       1               2              4               1               3
3       5               3              8               0               3
4       6               4              12              2               6
Average waiting time => 0.750000
Average Turn around time => 3.500000
```

CONCLUSION :-

FCFS scheduling algorithm is a simplest way of scheduling a process in a CPU. But it is not an efficient way to schedule the processes in a CPU. The reason is that if one process gets executed then rest of the processes are on a stand-still and it might happen that CPU is in an ideal state and not working on any process .This decreases the utilization of the CPU or in other words we can say that we are not using the full CPU power . This scheduling algorithm is beneficial in those processes where the burst time (Execution time) is less for every process so CPU can easily complete the task as fast as possible with the help of this algorithm. Although this will not allow the CPU to be ideal but still the last process in the ready queue will always take the most completion time .This is the knowledge I gained while studying Operating System and CPU scheduling (FCFS).

Assignment 2

Write a C program to implement Shortest Job First or SJF process scheduling .

PROGRAM DESCRIPTION:-

SJF scheduling is a scheduling algorithm that selects for execution the waiting process with the smallest execution time .This means that the process with the minimum burst time or execution time will have the highest priority to complete its execution. SJF is a non-preemptive scheduling algorithm. SJF is advantageous because of its simplicity and because it minimizes the average amount of time each process has to wait until its execution is complete. However it has the potential for process starvation for processes which will require a long time to complete if short processes are continually.

VARIABLE DESCRIPTION:

Variable name	Variable type	Variable Description
i , j	int	Loop variables
n	int	Input and store the number of processes
current_time	int	time counter for calculating the current time of the scheduling process
is_complete[]	int	Array to store the indication that the process is completely executed or not
at[]	int	Array to store the arrival time of the processes
bt[]	int	Array to store the burst time of the processes
tat[]	int	Array to store the turn around time of the processes
comp[]	int	Array to store the completion time of the processes
wt[]	int	Array to store the waiting time of the processes
awt	float	To store the average waiting of all the processes
atat	float	To store the average turn around time of all the process
completed	int	Counter variable
idx	int	To store the index of the process of minimum burst time
mn	int	To store the minimum burst time among all the process given

CODE FOR SJF SCHEDULING ALGORITHM:-

```c
#include <stdio.h>
#include <conio.h>
#include <string.h>
#define max 30
void main()
{
  int n,i,j,at[max],wt[max],tat[max],bt[max],comp[max];
  float atat=0,awt=0;
  int is_complete[max];
  memset(is_complete,0,sizeof(is_complete));
  printf("Enter the number of process --> ");
  scanf("%d",&n);
  for(i=0;i<n;i++)
  {
    printf("Process %d\n",i+1);
    printf("Arrival time --> ");
    scanf("%d",&at[i]);
    printf("\nBurst time --> ");
    scanf("%d",&bt[i]);
  }
  int current_time=0;
  int completed=0;
  int prev=0;
  printf("PROCESS\tARRIVAL TIME\tBURST TIME\tCOMPLETION TIME\t WAITING TIME\t TURN AROUND TIME\n");
  while(completed!=n)
  {
    int idx=-1;
    int mn=10000000;

    for(i=0;i<n;i++)
    {
      if(at[i]<=current_time && is_complete[i]==0)
      {
        if(bt[i]<mn)
        {
          mn=bt[i];
          idx=i;
        }
        if(bt[i]==mn)
        {
          if(at[i]<at[idx])
          {
            mn=bt[i];
            idx=i;
          }
        }
      }
    }
    if(idx!=-1){
      comp[idx]=current_time+bt[idx];
      tat[idx]=comp[idx]-at[idx];
      wt[idx]=tat[idx]-bt[idx];
      awt=awt+wt[idx];
      atat=atat+tat[idx];
      is_complete[idx]=1;
      completed++;
      current_time=comp[idx];
      prev=current_time;

     }
     else
     {
       current_time++;
     }
   }
   for(int i=0;i<n;i++)
   {
      printf("%d\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n",i+1,at[i],bt[i],comp[i],wt[i],tat[i]);
   }
   printf("Average waiting time => %f",awt/n);
   printf("\nAverage Turn around time => %f",atat/n);

}
```

SAMPLE INPUT:-

Enter the number of process --> 4

Process 1

Arrival time --> 1 Burst time --> 3

Process 2

Arrival time --> 2 Burst time --> 4

Process 3

Arrival time --> 1 Burst time --> 2

Process 4

Arrival time --> 4 Burst time --> 4

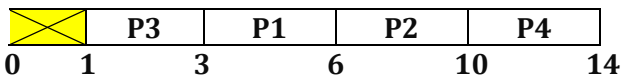
SAMPLE OUTPUT:-

PROCESS	ARRIVAL TIME	BURST TIME	COMPLETION TIME	WAITING TIME	TURN AROUND TIME
1	1	3	6	2	5
2	2	4	10	4	8
3	1	2	3	0	2
4	4	4	14	6	10

Average waiting time => 3.000000

Average Turn around time => 6.250000

GANTT CHART:-



SNAPSHOT OF THE OUTPUT TERMINAL:-

```
Enter the number of process --> 4

Process 1
Arrival time --> 1
Burst time --> 3

Process 2
Arrival time --> 2
Burst time --> 4

Process 3
Arrival time --> 1
Burst time --> 2

Process 4
Arrival time --> 4
Burst time --> 4
PROCESS ARRIVAL TIME    BURST TIME      COMPLETION TIME  WAITING TIME    TURN AROUND TIME
1       1               3               6                2              5
2       2               4               10               4              8
3       1               2               3                0              2
4       4               4               14               6              10
Average waiting time => 3.000000
Average Turn around time => 6.250000
```

CONCLUSION:-

SJF scheduling algorithm is a simple way to schedule CPU processing . It is more efficient than FCFS scheduling algorithm but still it might cause process starvation if shorter processes keeps on geting executed and longer process are not getting a chance to get into the running queue.It cannot be implemented at the level of short term CPU scheduling .It has an advantage to more optimal waiting time as it proves to provide minimum average waiting time for a given set of processes. And Lastly it is a good example for a greedy algorithm

Assignment 3

Write a C program to implement Shortest Remaining Time First or SRTF process scheduling .

PROGRAM DESCRIPTION:-

SRTF scheduling algorithm is a preemptive algorithm. In here the process never gets fully executed unless and until it has the lowest burst time among all the processes. It is also a preemptive SJF scheduling algorithm where burst time is not the priority but the burst remaining time is taken as the priority .This means that the process will execute unless any other process in the ready queue has the lower burst remaining time than the current process , if it happens then the current process gets preempted in the ready queue and the new process with a lower burst remaining time gets executed .This algorithm works as long as the burst remaining time of all the processes becomes zero.

VARIABLE DESCRIPTION:

Variable name	Variable type	Variable Description
i , j	int	Loop variables
n	int	Input and store the number of processes
current_time	int	time counter for calculating the current time of the scheduling process
is_complete[]	int	Array to store the indication that the process is completely executed or not
at[]	int	Array to store the arrival time of the processes
bt[]	int	Array to store the burst time of the processes
tat[]	int	Array to store the turn around time of the processes
comp[]	int	Array to store the completion time of the processes
wt[]	int	Array to store the waiting time of the processes
st[]	int	Array to store the Starting point of every process
rt[]	int	Array to store the response time of the process.
burst_remaining []	int	Array to store the remaining execution for the process to execute
awt	float	To store the average waiting of all the processes
atat	float	To store the average turn around time of all the process
art	float	To store the average response time
completed	int	Counter variable
idx	int	To store the index of the process of minimum burst time
mn	int	To store the minimum burst time among all the process given

CODE FOR SRTF SCHEDULING ALGORITHM:-

```c
#include <stdio.h>
#include <conio.h>
#include <string.h>
#define max 30
void main()
{
  int n,i,j,at[max],wt[max],tat[max],bt[max],comp[max],st[max],rt[max];
  float atat=0,awt=0,art=0;
  int is_complete[max];
  int burst_remaining[max];
  memset(is_complete,0,sizeof(is_complete));
  printf("Enter the number of process --> ");
  scanf("%d",&n);
  for(i=0;i<n;i++)
  {
    printf("\nProcess %d",i+1);
    printf("\nArrival time --> ");
    scanf("%d",&at[i]);
    printf("Burst time --> ");
    scanf("%d",&bt[i]);
    burst_remaining[i]=bt[i];
  }
int current_time=0;
int completed=0;
int prev=0;
printf("PROCESS\tARRIVAL TIME\tBURST TIME\tCOMPLETION TIME\tWAITING TIME\tTURN AROUND TIME\tRESPONSE TIME\n");
while(completed!=n)
{
  int idx=-1;
  int mn=100000;
  for(i=0;i<n;i++)
  {
    if(at[i]<=current_time && is_complete[i]==0)
    {
      if(burst_remaining[i]<mn)
      {
        mn=burst_remaining[i];
        idx=i;
      }
      if(burst_remaining[i]==mn)
      {
        if(at[i]<at[idx])
        {
          mn=burst_remaining[i];
          idx=i;
        }
      }

    }
  }
  if(idx!=-1)
  {
   if(burst_remaining[idx]==bt[idx])
   {
     st[idx]=current_time;
   }
   burst_remaining[idx]-=1;
   current_time++;
   prev=current_time;
   if(burst_remaining[idx]==0)
   {
     comp[idx]=current_time;
     tat[idx]=comp[idx]-at[idx];
     wt[idx]=tat[idx]-bt[idx];
     rt[idx]=st[idx]-at[idx];

     awt+=wt[idx];
     atat+=tat[idx];
     art+=rt[idx];

     is_complete[idx]=1;
     completed++;
   }
  }
  else{
```

```
      current_time++;
    }
  }
  for(int i=0;i<n;i++)
   {
      printf("%d\t%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t\t%d\n",i+1,at[i],bt[i],comp[i],wt[i],tat[i],rt[i]);
   }
   printf("Average waiting time => %f",awt/n);
   printf("\nAverage Turn around time => %f",atat/n);
    printf("\nAverage Response time=> %f",art/n);
}
```

SAMPLE INPUT:-

Enter the number of process --> 4

Process 1

Arrival time --> 0 Burst time --> 5

Process 2

Arrival time --> 1 Burst time --> 3

Process 3

Arrival time --> 2 Burst time --> 4

Process 4

Arrival time --> 4 Burst time --> 1

SAMPLE OUTPUT:-

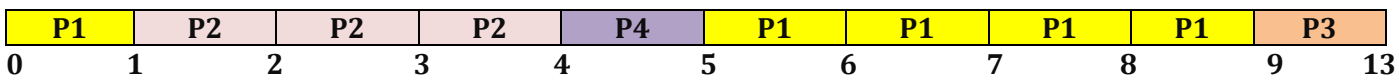
PROCESS	ARRIVAL TIME	BURST TIME	COMPLETION TIME	WAITING TIME	TURN AROUND TIME	RESPONSE TIME
1	1	3	9	4	9	0
2	2	4	4	0	3	0
3	1	2	13	7	11	7
4	4	4	5	0	1	0

Average waiting time => 2.750000

Average Turn around time => 6.000000

Average Response time=> 1.750000

GANTT CHART:-



SNAPSHOT OF THE OUTPUT TERMINAL:-

```
Enter the number of process --> 4

Process 1
Arrival time --> 0
Burst time --> 5

Process 2
Arrival time --> 1
Burst time --> 3

Process 3
Arrival time --> 2
Burst time --> 4

Process 4
Arrival time --> 4
Burst time --> 1
PROCESS        ARRIVAL TIME          BURST TIME            COMPLETION TIME       WAITING TIME          TURN AROUND TIME      RESPONSE TIME
1              0                     5                     9                     4                     9                     0
2              1                     3                     4                     0                     3                     0
3              2                     4                     13                    7                     11                    7
4              4                     1                     5                     0                     1                     0
Average waiting time => 2.750000
Average Turn around time => 6.000000
Average Response time=> 1.750000
```

CONCLUSION:-

SRTF algorithm is far more efficient than SJF as it is preemptive and every process gets an equal opportunity to be in the running queue so that it can be executed . I learnt that short processes are handled very quickly .The system also requires very little overhead since it only makes a decision when a process is added .When a new process is added the algorithm only needs to compare the currently executing process with the new process, ignoring all other processes currently waiting to execute. But every scheduling has its disadvantages .Like SJF , it has the potential for process starvation. Long processes may be held off immediately if short processes are continually added. These are the points learnt while implementing SRTF scheduling.

Assignment 4

Write a C program to implement Round Robin or RR process scheduling .

PROGRAM DESCRIPTION:-

Round Robin scheduling is a CPU scheduling algorithm where each process is assigned a fixed time slot in a cyclic way. It is simple and easy to implement and in here its starvation free as all the processes gets fair share of CPU. It is preemptive as processes are assigned CPU only for a fixed slice of time at most.It has a ready queue and a running queue and there is high level of context switching because as soon as the time quantum quota of that process is over is preempted to the ready queue unless the burst remaining time of the process is zero.

VARIABLE DESCRIPTION:

Variable name	Variable type	Variable Description
i , j	int	Loop variables
n	int	Input and store the number of processes
tq	int	To input and store the time quantum.
current_time	int	time counter for calculating the current time of the scheduling process
is_complete[]	int	Array to store the indication that the process is completely executed or not
at[]	int	Array to store the arrival time of the processes
bt[]	int	Array to store the burst time of the processes
tat[]	int	Array to store the turn around time of the processes
comp[]	int	Array to store the completion time of the processes
wt[]	int	Array to store the waiting time of the processes
mark[]	int	Array to mark and indicate that the process is still about to be completed or not.
st[]	int	Array to store the starting point of the process when it entered the running queue to get executed.
q[]	int	Ready queue created to store the process id to get executed first
p[]	int	Array to store the process id.
awt	float	To store the average waiting of all the processes
atat	float	To store the average turn around time of all the process
completed	int	Counter variable
idx	int	To store the index of the process of minimum burst time
mn	int	To store the minimum burst time among all the process given
temp	int	Temporary variable used for swapping.
front , rear	int	To Store the front and the rear index of the queue.

CODE FOR Round Robin SCHEDULING ALGORITHM:-

```c
#include <stdio.h>
#include <conio.h>
#include <string.h>
#include <stdlib.h>

#define max 100
void push(int,int);
void pop();
int empty();
int front1();
void swap(int*,int);
int q[max],front=-1,rear=-1;

void main()
{
   int tq,n,i,idx,j,at[max],wt[max],tat[max],bt[max],comp[max],st[max],rt[max];
   float atat=0,awt=0,art=0;
   int mark[max];
   int p[max];
   int burst_remaining[max];
   printf("Enter the number of processes\n");
   scanf("%d",&n);
   printf("Enter the Time Quantam\n");
   scanf("%d",&tq);
   for(int i=0;i<n;i++)
   {
     printf("\nProcess %d",i+1);
     printf("\nArrival time --> ");
     scanf("%d",&at[i]);
     printf("Burst time --> ");
     scanf("%d",&bt[i]);
     burst_remaining[i]=bt[i];
     p[i]=i+1;
   }
 int temp=0;
 for(int i=0;i<n;i++)
 {
   for(int j=0;j<n-i-1;j++)
   {
     if(at[j]>at[j+1])
     {
       swap(at,j);
       swap(p,j);
       swap(bt,j);
       swap(burst_remaining,j);
     }
   }
 }

  int current_time=0;
  push(0,n);
  int completed=0;
  memset(mark,0,sizeof(mark));
  mark[0]=1;
  while(completed!=n)
  {
    idx=front1();
    pop();
    if(burst_remaining[idx]==bt[idx])
    {
      st[idx]=(current_time>at[idx])?current_time:at[idx];
      current_time=st[idx];
    }
    if(burst_remaining[idx]-tq>0)
    {
      burst_remaining[idx]-=tq;
      current_time+=tq;
    }
    else{
      current_time+=burst_remaining[idx];
      burst_remaining[idx]=0;
      completed++;
      comp[idx]=current_time;
      tat[idx]=comp[idx]-at[idx];
      wt[idx]=tat[idx]-bt[idx];
```

```
      rt[idx]=st[idx]-at[idx];

      awt+=wt[idx];
      atat+=tat[idx];
      art+=rt[idx];

    }
    for(int i=1;i<n;i++)
    {
      if(burst_remaining[i]>0 && at[i]<=current_time && mark[i]==0){
        push(i,n);
        mark[i]=1;
      }
    }
    if(burst_remaining[idx]>0)
    push(idx,n);

    if(empty()==1)
    {
      for(int i=1;i<n;i++)
    {
      if(burst_remaining[i]>0){
        push(i,n);
        mark[i]=1;
        break;
       }
     }
     }
   }
    temp=0;
  for(int i=0;i<n;i++)
  {
    for(int j=0;j<n-i-1;j++)
    {
      if(p[j]>p[j+1])
      {
        swap(at,j);
        swap(p,j);
        swap(bt,j);
        swap(wt,j);
        swap(rt,j);
        swap(tat,j);
        swap(comp,j);
      }
    }
  }
  printf("PROCESS\tARRIVAL TIME\tBURST TIME\tCOMPLETION TIME\tWAITING TIME\tTURN AROUND TIME\tRESPONSE TIME\n");
  for(int i=0;i<n;i++)
   {
      printf("%d\t%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t\t%d\n",p[i],at[i],bt[i],comp[i],wt[i],tat[i],rt[i]);
   }
   printf("Average waiting time => %f",awt/n);
   printf("\nAverage Turn around time => %f",atat/n);
    printf("\nAverage Response time=> %f",art/n);
  }
  void swap(int *a,int j)
  {
    int temp=0;
    temp=a[j];
    a[j]=a[j+1];
    a[j+1]=temp;
  }
  void push(int v,int x)
  {
    if(rear==x-1)
    printf("QUEUE\n");
    else
    {
      if(front==-1)
      front=0;
      rear++;
      q[rear]=v;
    }
  }
  void pop()
  {
    if(front==-1)
```

```
  printf("QUEUE\n");
  else
  {
    front++;
    if(front>rear)
    front=rear=-1;
  }
}
int empty()
{
  if(front==-1)
  return 1;
  else
  return 0;
}
int front1()
{
  if(front==-1)
   printf("QUEUE\n");
   else
   return q[front];

}
```

SAMPLE INPUT:-

Enter the number of processes 4

Enter the Time Quantam 2

Process 1

Arrival time --> 0 Burst time --> 5

Process 2

Arrival time --> 1 Burst time --> 4

Process 3

Arrival time --> 2 Burst time --> 2

Process 4

Arrival time --> 4 Burst time --> 1

SAMPLE OUTPUT:-

PROCESS	ARRIVAL TIME	BURST TIME	COMPLETION TIME	WAITING TIME	TURN AROUND TIME	RESPONSE TIME
1	0	5	12	7	12	0
2	1	4	11	6	10	1
3	2	2	6	2	4	2
4	4	1	9	4	5	4

Average waiting time => 4.750000

Average Turn around time => 7.750000

Average Response time=> 1.750000

GANTT CHART:-

READY QUEUE →

P1	P2	P3	P1	P4	P2	P1

RUNNING QUEUE →

P1	P2	P3	P1	P4	P2	P1

0 2 4 6 8 9 11 12

SNAPSHOT OF THE OUTPUT TERMINAL:-

```
Enter the number of processes
4
Enter the Time Quantam
2

Process 1
Arrival time --> 0
Burst time --> 5

Process 2
Arrival time --> 1
Burst time --> 4

Process 3
Arrival time --> 2
Burst time --> 2

Process 4
Arrival time --> 4
Burst time --> 1
QUEUE
QUEUE
QUEUE
QUEUE
PROCESS ARRIVAL TIME    BURST TIME      COMPLETION TIME WAITING TIME    TURN AROUND TIME        RESPONSE TIME
1       0               5               12              7               12                      0
2       1               4               11              6               10                      1
3       2               2               6               2               4                       2
4       4               1               9               4               5                       4
Average waiting time => 4.750000
Average Turn around time => 7.750000
Average Response time=> 1.750000
```

CONCLUSION:

Round Round scheduling algorithm is an efficient way of scheduling the processes in a CPU. There is fairness since every process gets equal share of CPU. But the throughput of this algorithm is low . There is context switching. Gantt chart seem to come to big if the time quantum is very less. After all this disadvantages we can still say that due to free of starvation and very minimum ideal time of CPU making the CPU more effective in work. Round Robin Algorithm is one of the most efficient scheduling algorithm .

Assignment 5

Write a C program to implement High Response Ratio Next or HRRN process scheduling.

PROGRAM DESCRIPTION:-

HRRN is a non preemptive scheduling algorithm which works on the priority of response ratio of a given process after a given process or execution time. The process with the highest response ratio will get executed first and this response ratio changes evertime a new process has to be picked by the CPU.This Scheduling algorithm has the same advantages and disadvatages like SJF because it picks the shortest job first but also concern the waiting time of the longer jobs . Since the scheduling is non-preemptive so minimal context switching is observed .

VARIABLE DESCRIPTION:

Variable name	Variable type	Variable Description
i , j	int	Loop variables
n	int	Input and store the number of processes
current_time	int	time counter for calculating the current time of the scheduling process
is_complete[]	int	Array to store the indication that the process is completely executed or not
at[]	int	Array to store the arrival time of the processes
bt[]	int	Array to store the burst time of the processes
tat[]	int	Array to store the turn around time of the processes
comp[]	int	Array to store the completion time of the processes
wt[]	int	Array to store the waiting time of the processes
awt	float	To store the average waiting of all the processes
atat	float	To store the average turn around time of all the process
completed	int	Counter variable
idx	int	To store the index of the process of minimum burst time
hrrn	int	To store the maximum response ratio.

CODE FOR HRRN SCHEDULING ALGORITHM:-

```c
#include <stdio.h>
#include <conio.h>
#include <string.h>
#include <math.h>
#define max 30
void main()
{
  int i,j,n,at[max],bt[max],tat[max],comp[max],wt[max];
  float awt=0,atat=0;
  int is_complete[max];
  memset(is_complete,0,sizeof(is_complete));
  printf("Enter the number of process --> ");
  scanf("%d",&n);
  for(i=0;i<n;i++)
  {
    printf("Process %d\n",i+1);
    printf("Arrival time --> ");
    scanf("%d",&at[i]);
    printf("\nBurst time --> ");
    scanf("%d",&bt[i]);
  }

int temp=0;
for(i=0;i<n;i++)
{
  for(j=0;j<n-i-1;j++)
  {
    if(at[j]>at[j+1])
    {
      temp=at[j];
      at[j]=at[j+1];
      at[j+1]=temp;
      temp=bt[j];
      bt[j]=bt[j+1];
      bt[j+1]=temp;


    }
  }
}
int current_time=0,completed=0;
while(completed!=n)
{
  float hrrn=-99999;
  float temp=0;
  int idx=-1;
  for(i=0;i<n;i++)
  {
    if(at[i]<=current_time && is_complete[i]==0)
    {
      temp=((bt[i]+(current_time-at[i]))/bt[i]);
      if(hrrn<temp)
      {
        hrrn=temp;
        idx=i;
      }

    }
  }
  if(idx!=-1)
  {
    current_time+=bt[idx];
    comp[idx]=current_time;
    tat[idx]=comp[idx]-at[idx];
    wt[idx]=tat[idx]-bt[idx];
    awt+=wt[idx];
    atat+=tat[idx];
    is_complete[idx]=1;
    completed++;
  }
  else
  current_time++;
}
 printf("PROCESS\tARRIVAL TIME\tBURST TIME\tCOMPLETION TIME\t WAITING TIME\t TURN AROUND TIME\n");
  for(int i=0;i<n;i++)
```

```
   {
      printf("%d\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n",i+1,at[i],bt[i],comp[i],wt[i],tat[i]);
   }
   printf("Average waiting time => %f",awt/n);
   printf("\nAverage Turn around time => %f",atat/n);
}
```

SAMPLE INPUT:-

Enter the number of process --> 5

Process 1

Arrival time --> 0 Burst time --> 3

Process 2

Arrival time --> 2 Burst time --> 6

Process 3

Arrival time --> 4 Burst time --> 4

Process 4

Arrival time --> 6 Burst time --> 5

Process 5

Arrival time --> 8 Burst time --> 2

SAMPLE OUTPUT:-

PROCESS	ARRIVAL TIME	BURST TIME	COMPLETION TIME	WAITING TIME	TURN AROUND TIME
1	0	3	3	0	3
2	2	6	9	1	7
3	4	4	13	5	9
4	6	5	20	9	14
5	8	2	15	5	7

Average waiting time => 4.000000
Average Turn around time => 8.000000

GANTT CHART :-

P1	P2	P3	P5	P4

0 3 9 13 15 20

SNAPSHOT OF THE OUTPUT TERMINAL:-

```
Enter the number of process --> 5

Process 1
Arrival time --> 0

Burst time --> 3

Process 2
Arrival time --> 2

Burst time --> 6

Process 3
Arrival time --> 4

Burst time --> 4

Process 4
Arrival time --> 6

Burst time --> 5

Process 5
Arrival time --> 8

Burst time --> 2
PROCESS ARRIVAL TIME    BURST TIME      COMPLETION TIME  WAITING TIME    TURN AROUND TIME
1       0               3               3               0               3
2       2               6               9               1               7
3       4               4               13              5               9
4       6               5               20              9               14
5       8               2               15              5               7
Average waiting time => 4.000000
Average Turn around time => 8.000000
```

CONCLUSION:-

HRRN scheduling algorithm is another efficient way to beat SJF scheduling algorithm. I learnt that this algorithm although has a efficient way of process working in the CPU still this algorithm is not practically implemented because the burst time of the process cannot be known in advance. However for college exam purpose this algorithm is asked and it is quite helpful and easy to solve and if some knows how to code SJF scheduling then its not a difficult task to implement HRRN scheduling algorithm.

Assignment 6

Write a C program to implement Preemptive Priority Scheduling or PPS process scheduling .

PROGRAM DESCRIPTION:-

Preemptive Priority based process scheduling or PPS scheduling is another type of scheduling algorithm where external priority is provided to schedule a given number of processes. Like other scheduling algorithm (FCFS ,SJF,SRTF) where either burst time or arrival time is taken as the priority but here external priority is already there . This algorithm is preemptive , this means that the processes are checked every equal intervals of time whether the external priority is more or less than the current process. This code helps the preemption to be possible and the total code or program gets terminated only when the burst remaining time of all the processes is equal to zero.

. VARIABLE DESCRIPTION:

Variable name	Variable type	Variable Description
i , j	int	Loop variables
n	int	Input and store the number of processes
current_time	int	time counter for calculating the current time of the scheduling process
is_complete[]	int	Array to store the indication that the process is completely executed or not
burst_remaining[]	int	Array to store the burst remaining time of the given process
process[]	int	Array to store the process id
priority[]	int	Array to store the external priority provided by user.
at[]	int	Array to store the arrival time of the processes
bt[]	int	Array to store the burst time of the processes
tat[]	int	Array to store the turn around time of the processes
comp[]	int	Array to store the completion time of the processes
wt[]	int	Array to store the waiting time of the processes
awt	float	To store the average waiting of all the processes
atat	float	To store the average turn around time of all the process
completed	int	Counter variable
idx	int	To store the index of the process of minimum burst time
mx	int	To store the maximum priority of all the given process

CODE FOR PPS SCHEDULING ALGORITHM:-

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
#include <stdlib.h>
#define max 30
void main()
{
  int n,i,j,at[max],wt[max],tat[max],bt[max],comp[max],process[max],priority[max],st[max],rt[max];
  float atat=0,awt=0,art=0;
  int is_complete[max]; int burst_remaining[max];
  memset(is_complete,0,sizeof(is_complete));
  printf("Enter the number of process --> ");
  scanf("%d",&n);
  for(i=0;i<n;i++)
  {
    printf("\nProcess %d",i+1);
    printf("\nArrival time --> ");
    scanf("%d",&at[i]);
    printf("Burst time     --> ");
    scanf("%d",&bt[i]);
    printf("Priority given --> ");
    scanf("%d",&priority[i]);
    process[i]=i+1;
    burst_remaining[i]=bt[i];
  }
  int current_time=0;
  int completed=0;
  int prev=0;
  while(completed!=n)
  {
    int idx=-1;
    int mx=-1;
    for(i=0;i<n;i++)
    {
      if(at[i]<=current_time && is_complete[i]==0)
      {
        if(priority[i]>mx)
        {
          mx=priority[i];
          idx=i;
        }
        if(priority[i]==mx)
        {
          if(at[i]<at[idx])
          {
            mx=priority[i];
            idx=i;
          }
        }
      }
    }
    if(idx!=-1)
    {
      if(burst_remaining[idx]==bt[idx])
     {
       st[idx]=current_time;

     }

     burst_remaining[idx]-=1;
     current_time++;
     prev=current_time;
     if(burst_remaining[idx]==0)
     {
       comp[idx]=current_time;
       tat[idx]=comp[idx]-at[idx];
       wt[idx]=tat[idx]-bt[idx];
       rt[idx]=st[idx]-at[idx];

       awt+=wt[idx];
       atat+=tat[idx];
       art+=rt[idx];
       is_complete[idx]=1;
       completed++;
       current_time=comp[idx];
```

```
      prev=current_time;
    }

   }
   else
   {
     current_time++;
   }
 }
 printf("PROCESS  PRIORITY  AT\t  BT\t  CT\t  WT\t  TAT\t  RT\n");
 for(int i=0;i<n;i++)
  {
     printf("%d\t %d\t   %d\t   %d\t   %d\t   %d\t   %d\t   %d\n",process[i],priority[i],at[i],bt[i],comp[i],wt[i],tat[i],rt[i]);
  }
  printf("Average waiting time => %f",awt/n);
  printf("\nAverage Turn around time => %f",atat/n);
  printf("\nAverage Response time=> %f",art/n);
}
```

SAMPLE INPUT:-

Enter the number of process --> 5

Process 1

Arrival time --> 0 Burst time --> 4 Priority given --> 2

Process 2

Arrival time --> 1 Burst time --> 3 Priority given --> 3

Process 3

Arrival time --> 2 Burst time --> 1 Priority given --> 4

Process 4

Arrival time --> 3 Burst time --> 5 Priority given --> 5

Process 5

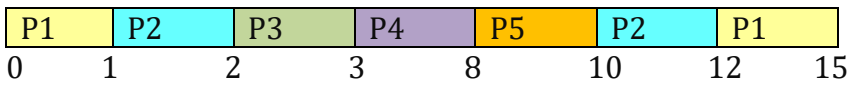
Arrival time --> 4 Burst time --> 2 Priority given --> 5

SAMPLE OUTPUT:-

PROCESS	PRIORITY	ARRIVAL TIME	BURST TIME	COMPLETION TIME	WAITING TIME	TURN AROUND TIME	RESPONSE TIME
1	2	0	4	15	11	15	0
2	3	1	3	12	8	11	0
3	4	2	1	3	0	1	0
4	5	3	5	8	0	5	0
5	5	4	2	10	4	6	4

Average waiting time => 4.600000
Average Turn around time => 7.600000
Average Response time=> 0.800000

GANTT CHART:-



SNAPSHOT OF THE OUTPUT TERMINAL:-

```
Enter the number of process --> 5

Process 1
Arrival time --> 0
Burst time     --> 4
Priority given --> 2

Process 2
Arrival time --> 1
Burst time     --> 3
Priority given --> 3

Process 3
Arrival time --> 2
Burst time     --> 1
Priority given --> 4

Process 4
Arrival time --> 3
Burst time     --> 5
Priority given --> 5

Process 5
Arrival time --> 4
Burst time     --> 2
Priority given --> 5
PROCESS  PRIORITY  AT     BT      CT      WT      TAT     RT
1        2         0       4       15      11      15      0
2        3         1       3       12      8       11      0
3        4         2       1       3       0       1       0
4        5         3       5       8       0       5       0
5        5         4       2       10      4       6       4
Average waiting time => 4.600000
Average Turn around time => 7.600000
Average Response time=> 0.800000
```

CONCLUSION:-

PPS scheduling algorithm is based on priority and preemption prevails unless there is a tie among the priority then FCFS algorithm is then comes into consideration .This type of algorithm is efficient for those processes which have low burst time because if a process have low priority and higher burst time then it will take much longer time to get excecuted . this leads to starvation. As per the knowledge gained by me is that this preemption mode is best for real time operating system. One thing to be noticed that in PPS the waiting of the process having the highest priority will always be zero. The priority given can be from low to high or high to low.

Assignment 7

Write a C program to implement Non-Preemptive Priority Scheduling or NPPS process scheduling .

PROGRAM DESCRIPTION:-

Non-Preemptive Priority based process scheduling or PPS scheduling is another type of scheduling algorithm where external priority is provided to schedule a given number of processes. Like other scheduling algorithm (FCFS ,SJF,SRTF) where either burst time or arrival time is taken as the priority but here external priority is already there . This algorithm is non-preemptive , this means that the highest priority of the process is taken and then the whole process gets executed with its full execution time.This code or program get terminated only when all the process gets executed .

. VARIABLE DESCRIPTION:

Variable name	Variable type	Variable Description
i , j	int	Loop variables
n	int	Input and store the number of processes
current_time	int	time counter for calculating the current time of the scheduling process
is_complete[]	int	Array to store the indication that the process is completely executed or not
process[]	int	Array to store the process id
priority[]	int	Array to store the external priority provided by user.
at[]	int	Array to store the arrival time of the processes
bt[]	int	Array to store the burst time of the processes
tat[]	int	Array to store the turn around time of the processes
comp[]	int	Array to store the completion time of the processes
wt[]	int	Array to store the waiting time of the processes
awt	float	To store the average waiting of all the processes
atat	float	To store the average turn around time of all the process
completed	int	Counter variable
idx	int	To store the index of the process of minimum burst time
mx	int	To store the maximum priority of all the given process

CODE FOR NPPS SCHEDULING ALGORITHM:-

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
#include <stdlib.h>
#define max 30
void main()
{
  int n,i,j,at[max],wt[max],tat[max],bt[max],comp[max],process[max],priority[max],st[max],rt[max];
  float atat=0,awt=0,art=0;
  int is_complete[max];
  memset(is_complete,0,sizeof(is_complete));
  printf("Enter the number of process --> ");
  scanf("%d",&n);
  for(i=0;i<n;i++)
  {
    printf("\nProcess %d",i+1);
    printf("\nArrival time --> ");
    scanf("%d",&at[i]);
    printf("Burst time     --> ");
    scanf("%d",&bt[i]);
    printf("Priority given --> ");
    scanf("%d",&priority[i]);
  process[i]=i+1;
}
int current_time=0;
int completed=0;
int prev=0;
while(completed!=n)
{
  int idx=-1;
  int mx=-1;
  for(i=0;i<n;i++)
  {
    if(at[i]<=current_time && is_complete[i]==0)
    {
      if(priority[i]>mx)
      {
        mx=priority[i];
        idx=i;
      }
      if(priority[i]==mx)
      {
        if(at[i]<at[idx])
        {
          mx=priority[i];
          idx=i;
        }
      }
    }
  }
  if(idx!=-1)
  {
   st[idx]=current_time;
   comp[idx]=current_time+bt[idx];
   tat[idx]=comp[idx]-at[idx];
   wt[idx]=tat[idx]-bt[idx];
   rt[idx]=st[idx]-at[idx];

   awt+=wt[idx];
   atat+=tat[idx];
   art+=rt[idx];
   is_complete[idx]=1;
   completed++;
   current_time=comp[idx];
   prev=current_time;

  }
  else
  {
    current_time++;
  }
}
printf("PROCESS  PRIORITY  AT\t  BT\t  CT\t  WT\t  TAT\t  RT\n");
for(int i=0;i<n;i++)
 {
```

```
      printf("%d\t %d\t   %d\t   %d\t   %d\t   %d\t   %d\t   %d\n",process[i],priority[i],at[i],bt[i],comp[i],wt[i],tat[i],rt[i]);
   }
   printf("Average waiting time => %f",awt/n);
   printf("\nAverage Turn around time => %f",atat/n);
   printf("\nAverage Response time=> %f",art/n);
}
```

SAMPLE INPUT:-

Enter the number of process --> 5

Process 1

Arrival time --> 0 Burst time --> 4 Priority given --> 2

Process 2

Arrival time --> 1 Burst time --> 3 Priority given --> 3

Process 3

Arrival time --> 2 Burst time --> 1 Priority given --> 4

Process 4

Arrival time --> 3 Burst time --> 5 Priority given --> 5

Process 5

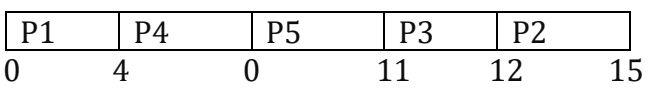
Arrival time --> 4 Burst time --> 2 Priority given --> 5

SAMPLE OUTPUT:-

PROCESS	PRIORITY	ARRIVAL TIME	BURST TIME	COMPLETION TIME	WAITING TIME	TURN AROUND TIME	RESPONSE TIME
1	2	0	4	4	0	4	0
2	3	1	3	15	11	14	11
3	4	2	1	12	9	10	9
4	5	3	5	9	1	6	1
5	5	4	2	11	5	7	5

Average waiting time => 5.200000
Average Turn around time => 8.200000
Average Response time=> 5.200000

GANTT CHART:-



SNAPSHOT OF THE OUTPUT TERMINAL:-

```
Enter the number of process --> 5

Process 1
Arrival time --> 0
Burst time     --> 4
Priority given --> 2

Process 2
Arrival time --> 1
Burst time     --> 3
Priority given --> 3

Process 3
Arrival time --> 2
Burst time     --> 1
Priority given --> 4

Process 4
Arrival time --> 3
Burst time     --> 5
Priority given --> 5

Process 5
Arrival time --> 4
Burst time     --> 2
Priority given --> 5
PROCESS  PRIORITY  AT     BT      CT      WT      TAT     RT
1        2         0       4       4       0       4       0
2        3         1       3       15      11      14      11
3        4         2       1       12      9       10      9
4        5         3       5       9       1       6       1
5        5         4       2       11      5       7       5
Average waiting time => 5.200000
Average Turn around time => 8.200000
Average Response time=> 5.200000
```

CONCLUSION:-

NPPS scheduling algorithm is similar to what I learnt from FCFS .Since both the algorithm are non-preemptive that's why they have some similar features . There is only one difference is that in FCFS arrival time was taken as the priority and in here external priority is given . So having the knowledge of FCFS algorithm NPPS was like icing on the cake . But there are some demerits while using this algorithm. In here starvation is there which makes it quite in efficient . That why the preemptive method of priority scheduling is taken into consideration over NPPS due to low starvation and higher utilization of CPU process.

Assignment 8

Write a C program to implement Earliest Deadline First or EDF process scheduling .

PROGRAM DESCRIPTION:-

EDF is a dynamic priority scheduling algorithm used in real time operating system to place processes in the priority queue. When ever a scheduling event occurs the queue will be searched for the process closest to its deadline. This process is the next to be schedule for execution. EDF is optimal scheduling algorithm on preemptive uniprocessors. The code written here is based on dynamical priority change. First based on the periods the hyperperiod or the lcm is taken out with the help of some block of code.

After that on the basis of that and its particular deadline given . processes gets excecuted and preempted as according to the deadline priority.

VARIABLE DESCRIPTION:

Variable name	Variable type	Variable Description
i,j	int	Loop Variable
d	int	used for allotting position
k	int	lcm of the periods
n	int	total number of processes
c	int	counter variable
nextTime	int	to store the next time
time	int	to note the current time
preemptive_count	int	to count the number of preemption
exec[]	float	Array to store the execution time
period[]	float	Array to store the period
individual_util[]	float	Array to store the utilization factor
flag[]	float	Array to store the condition values
release[]	float	Array to store the release time
deadline[]	float	Array to store the deadline
instance[]	float	Array to store the instant process
ex[]	float	Array to store the temp period time
responsemax[]	float	To store the max response time
responsemin[]	float	to store the min response time
tempmax	float	temporary variable
util	float	to store the total utilization
a,b	int	function variable
small	float	to store the smallest period
smallindex	int	to store the smallest period index.

CODE FOR EDF SCHEDULING ALGORITHM:-

```
#include <stdio.h>
#include <string.h>
int gcd(int a,int b){
  if(b==0)
  return a;
  else
  gcd(b,a%b);}
int lcm(int a,int b){
  return ((a*b)/gcd(a,b));}
int hyperperiod(float period[],int n){
  int k=period[0];
  n--;
  while(n>=1){
    k=lcm(k,period[n--]);}
  return k;}
int edf(float *period,int n,int t,float *deadline){
  int i,small=10000.0f,smallindex=0;
  for(i=0;i<n;i++){
    if(period[i]<small && (period[i]-t)<=deadline[i]){
      small=period[i];
      smallindex=i; }}
  if(small==10000.0f){
    return -1;}
  return smallindex;}

int main(){
  int i,j,n,c,d,k,nextTime=0,time=0,task,preemption_count;
  float exec[30],period[30],individual_util[30],flag[30],release[30];
  float deadline[30],instance[30],ex[30],responsemax[30],responsemin[30],tempmax;
  float util=0;
  printf("\nEARLIEST DEADLINE FIRST SCHEDULLING\n");
  printf("Enter the number of process -> ");
  scanf("%d",&n);
  printf("\n Enter the data\n");
  for(i=0;i<n;i++){
    printf("\nRelease time -> ");
    scanf("%f",&release[i]);
    printf("Period time -> ");
    scanf("%f",&period[i]);
    printf("Exceution time -> ");
    scanf("%f",&exec[i]);
    printf("Deadline -> ");
    scanf("%f",&deadline[i]);
  }
for(i=0;i<n;i++){
  individual_util[i]=exec[i]/period[i];
  util+=individual_util[i];
  responsemax[i]=exec[i];
  deadline[i]=period[i];
  instance[i]=0.0f;}
util=util*100;
if(util>100){
  printf("Utilization factor = %0.2f \n\nScheduling is not possible as Utilization factor is above 100\n",util);}
else{
  printf("Utilization factor = %0.2f \n\nScheduling is  possible as Utilization factor is below 100\n",util);
  printf("\nHyperperiod of the given task set is : %d\n\n",k=hyperperiod(period,n));
  c=0;
  while(time<k){
    nextTime=time+1;
    task=edf(period,n,time,deadline);
    if(task==-1){
      printf("(%d,%d) -->  ",time,nextTime);
      printf("idle\n ");
      time++;
      continue;}
```

```
    instance[task]++;
    printf("(%d,%d) -->  ",time,nextTime);
    printf("Task %d\n",(task+1));
    ex[c++]=task;
    if(instance[task]==exec[task]){
      tempmax=nextTime-(period[task]-deadline[task]);
      if(instance[task]<tempmax){
        responsemax[task]=tempmax; }
      else{
      responsemin[task]=instance[task];}
      if(deadline[task]==k){
        responsemin[task]=responsemax[task];}
      period[task]+=deadline[task];
      instance[task]=0.0f;}
    time++;}
  for(i=0;i<n;i++){
    printf("\n\nMaximum Response time of Task %d = %f",i+1,responsemax[i]);
    printf("\n\nMinimum Response time of Task %d = %f",i+1,responsemin[i]);}
  preemption_count=0;
  for(i=0;i<k;i=j) {
   flag[i]=1;
   d=ex[i];
   for(j=i+1;d==ex[j];j++){
     flag[d]++;}
   if(flag[d]==exec[d]){
     flag[d]=1;}
   else{
     flag[d]++;
     preemption_count++;}}
  printf("\n\nPreemptio count = %d",preemption_count); }
return 0;}
```

SAMPLE INPUT:-

Enter the number of process -> 3

Enter the data

Release time -> 0

Period time -> 4

Exceution time -> 1

Deadline -> 4

Release time -> 0

Period time -> 6

Exceution time -> 2

Deadline -> 6

Release time -> 0

Period time -> 8

Exceution time -> 3

Deadline -> 8

SAMPLE OUTPUT:-

Utilization factor = 95.83

Scheduling is possible as Utilization factor is below 100

Hyperperiod of the given task set is : 24

(0,1) --> Task 1
(1,2) --> Task 2
(2,3) --> Task 2
(3,4) --> Task 3
(4,5) --> Task 1
(5,6) --> Task 3
(6,7) --> Task 3
(7,8) --> Task 2
(8,9) --> Task 1
(9,10) --> Task 2
(10,11) --> Task 3
(11,12) --> Task 3
(12,13) --> Task 1
(13,14) --> Task 3
(14,15) --> Task 2
(15,16) --> Task 2
(16,17) --> Task 1
(17,18) --> Task 3
(18,19) --> Task 2
(19,20) --> Task 2
(20,21) --> Task 1
(21,22) --> Task 3
(22,23) --> Task 3
(23,24) --> ideal.

Maximum Response time of Task 1 = 1.000000

Minimum Response time of Task 1 = 1.000000

Maximum Response time of Task 2 = 4.000000

Minimum Response time of Task 2 = 2.000000

Maximum Response time of Task 3 = 7.000000

Minimum Response time of Task 3 = 0.000000

Preemptio count = 7

SNAPSHOT OF THE OUTPUT WINDOW:-

```
EARLIEST DEADLINE FIRST SCHEDULLING
Enter the number of process -> 3

 Enter the data

Release time -> 0
Period time -> 4
Exceution time -> 1
Deadline -> 4

Release time -> 0
Period time -> 6
Exceution time -> 2
Deadline -> 6

Release time -> 0
Period time -> 8
Exceution time -> 3
Deadline -> 8
Utilization factor = 95.83

Scheduling is  possible as Utilization factor is below 100

Hyperperiod of the given task set is : 24
```

```
(0,1) -->  Task 1
(1,2) -->  Task 2
(2,3) -->  Task 2
(3,4) -->  Task 3
(4,5) -->  Task 1
(5,6) -->  Task 3
(6,7) -->  Task 3
(7,8) -->  Task 2
(8,9) -->  Task 1
(9,10) -->  Task 2
(10,11) -->  Task 3
(11,12) -->  Task 3
(12,13) -->  Task 1
(13,14) -->  Task 3
(14,15) -->  Task 2
(15,16) -->  Task 2
(16,17) -->  Task 1
(17,18) -->  Task 3
(18,19) -->  Task 2
(19,20) -->  Task 2
(20,21) -->  Task 1
(21,22) -->  Task 3
(22,23) -->  Task 3
(23,24) -->  idle


Maximum Response time of Task 1 = 1.000000

Minimum Response time of Task 1 = 1.000000

Maximum Response time of Task 2 = 4.000000
```

```
Minimum Response time of Task 2 = 2.000000

Maximum Response time of Task 3 = 7.000000

Minimum Response time of Task 3 = 0.000000

Preemptio count = 7
```

CONCLUSION:-

EDF Scheduling algorithm is an optimal way of scheduling any process in the CPU. Real time processes in which priority keeps on varies gets easily sorted out with the help of this algorithm.The main concept in this scheduling is how the algorithm chooses the process and makes a virtual ready and running queue.EDF algorithm does not need the tasks or processes to be periodic and also the tasks o processes requires a fixed CPU burst time.In EDF,any executing task can be preempted if any other periodic instance with an earlier deadline is ready for execution and becomes active.

Assignment 9

Write a C program to implement Rate Monotonic process scheduling .

PROGRAM DESCRIPTION:-

Rate Monotonic algorithm is priority algorithm that belonged to the static priority scheduling category of Real time operating system. It is preemptive in nature. The priority is decided according to the cycle time of the processes that are involved .If the process has a small job duration ,then it has the highest priority. Thus if a process with the highest priority starts execution, then it will preempt the other running processes. The Priority of a process is inversely proportional to the period it will run.

This property of priority is used in this code to solve the rate monotonic algorithm.

VARIABLE DESCRIPTION:

Variable name	Variable type	Variable Description
i,j	int	Loop Variable
n	int	To store the number of processes
exec[]	int	Array to store the execution
period[]	int	Array to store the period time
rt[]	int	to store the response time
ready[]	int	to store the ready value
pr_id[]	int	Array to store the process id
utilization	float	to Store the total utilization of the CPU
totalexecutiontime	int	To store the total execution time
lcm_pr	int	To Store the lcm of the periods
a,b	int	function variables

CODE FOR RATE MONOTONIC SCHEDULING ALGORITHM:-

```c
#include <stdio.h>
#include <conio.h>
#include <math.h>
int gcd(int a,int b)
{
  if(b==0)
  return a;
  else
  gcd(b,a%b);
}
int lcm(int a,int b)
{
  return ((a*b)/gcd(a,b));
}
int hyperperiod(int *period,int n)
{
  int k=period[0];
  n--;
  while(n>=1)
  {
    k=lcm(k,period[n--]);

  }
  return k;
}
void swap(int *a,int j)
{
  int temp=a[j];
  a[j]=a[j+1];
  a[j+1]=temp;
}
int main()
{
  int i,j,n,exec[30],period[30],rt[30],ready[30],pr_id[30];
  float utilization = 0;
  int totalExecutionTime=0,lcm_pr;
  printf("Enter the number of process -> ");
  scanf("%d",&n);
  for(i=0;i<n;i++)
  {
    printf("P %d\n",(i+1));

    printf("Period time -> ");
    scanf("%d",&period[i]);
    printf("Exceution time -> ");
    scanf("%d",&exec[i]);
    pr_id[i]=i+1;
    ready[i]=1;
    rt[i]=exec[i];

  }
  for(i=0;i<n;i++)
  {
    for(j=0;j<n-i-1;j++)
    {
      if(period[j]>period[j+1])
      {
        swap(period,j);
        swap(exec,j);
        swap(pr_id,j);
        swap(rt,j);
        swap(ready,j);
      }
    }
  }
```

```
lcm_pr=hyperperiod(period,n);
printf("\nProcess No.\t\tExecution time \t\t Period\n");
 for(int i=0 ; i<n ; i++)
 {
      totalExecutionTime+=(lcm_pr/period[i])*exec[i];
      printf("%d\t\t\t%d\t\t\t%d\n",pr_id[i],exec[i],period[i]);
 }
 if(totalExecutionTime > lcm_pr){
      printf("\n\nTotal execution time is more than LCM so RMS is not feasible here\n");
      return 0;
  }
  utilization=(totalExecutionTime*1.0/lcm_pr) *100;
  printf("Utilization -> %f \n",utilization);
  printf("LCM -> %d \n",lcm_pr);
  printf("\nTIME\t\tRUNNING PROCESS\n");
  for( i=0 ; i<lcm_pr ; i++){
      if(i>=period[0])
          for(int j=0 ; j<n ; j++){
              if(!(i%period[j]))
                  ready[j]=1;
          }
      for(int j=0 ; j<n ; j++){
          if(ready[j]==1){
              if(rt[j]>0){
                  if(!(--(rt[j]))){
                      ready[j]=0;
                      rt[j]=exec[j];
                  }
              }
              printf("%d\t\t\tP%d\n",i,pr_id[j]);
              break;
          }
          if(j == n-1)
                 printf("(%d) Empty Slot\n",i);
      }

    }
}
```

SAMPLE INPUT:-

Enter the number of process -> 3

P1

Period time -> 20 Exceution time -> 3

P2

Period time -> 5 Exceution time -> 2

P3

Period time -> 10 Exceution time -> 2

SAMPLE OUTPUT:-

Process No. Execution time Period
2 2 5
3 2 10
1 3 20
Utilization -> 75.000000
LCM -> 20

TIME RUNNING PROCESS
0 P2
1 P2
2 P3
3 P3
4 P1
5 P2
6 P2
7 P1
8 P1
(9) Empty Slot
10 P2
11 P2
12 P3
13 P3
(14) Empty Slot
15 P2
16 P2
(17) Empty Slot
(18) Empty Slot
(19) Empty Slot

SNAPSHOT OF THE OUTPUT TERMINAL:-

```
Enter the number of process -> 3
P 1
Period time -> 20
Exceution time -> 3
P 2
Period time -> 5
Exceution time -> 2
P 3
Period time -> 10
Exceution time -> 2

Process No.            Execution time          Period
2                      2                      5
3                      2                      10
1                      3                      20
Utilization -> 75.000000
LCM -> 20
```

```
TIME            RUNNING PROCESS
0                       P2
1                       P2
2                       P3
3                       P3
4                       P1
5                       P2
6                       P2
7                       P1
8                       P1
(9) Empty Slot
10                      P2
11                      P2
12                      P3
13                      P3
(14) Empty Slot
15                      P2
16                      P2
(17) Empty Slot
(18) Empty Slot
(19) Empty Slot
```

CONCLUSION:-

Rate monotonic scheduling algorithm along with many other scheduling algorithm has its advantages and disadvantages. Some i learnt while i was engaged in doing this algorithm.

Advantages:-

It is easy to implement

If any static priority assignment algorithm can meet the deadline then rate monotonic scheduling can also be the same and its optimal.

Disadvantages:-

It is very difficult to support aperiodic and sporadic tasks under RMA

RMA is not optimal when task and period deadline differ.

Assignment 10

Write a C program to implement Multilevel Queue Feedback Scheduling Algorithm.

PROGRAM DESCRIPTION:-

Multilevel queue feedback scheduling algorithm is one the the most sufficient mode of scheduling algorithm where 3 queues are used with different scheduling algorithms and everytime it executes a process it works for a certain time quantum and then it goes from the higher queue to the folloing lower queues. And at last gets executed properly. This algorithm works on time quantum basis.

VARIABLE DESCRIPTION:

Variable name	Variable type	Variable Description
i , j	int	Loop variables
n	int	Input and store the number of processes
current_time	int	time counter for calculating the current time of the scheduling processes
is_complete[]	int	Array to store the indication that the process is completely executed or not
at[]	int	Array to store the arrival time of the processes
AT	int	To store the arrival times of the processes getting distributed to different queues (1) for round robin (2) for Non Preemptive priority & (3) for FCFS schedulings
BT	int	To store the Burst times of the processes getting distributed to different queues (1) for round robin (2) for Non Preemptive priority & (3) for FCFS schedulings
priority2[]	int	To set priority to which queue the process should enter
bt[]	int	Array to store the burst time of the processes
TAT	int	Array to store the turn around time of the processes
COMP	int	Array to store the completion time of the processes
WT	int	Array to store the waiting time of the processes
awt	float	To store the average waiting of all the processes
atat	float	To store the average turn around time of all the process
art	float	To store the response average response of the processes
completed	int	Counter variable
idx	int	To store the index of the process of minimum burst time
Q1,Q2,Q3	int	To the structure queues values
RT	int	To store the remaining burst time left before its get preempted.
tq1,tq2	int	To store the time quantam for the round robin process

CODE FOR MULTILEVEL QUEUE FEEDBACK SCHEDULING ALGORITHM:-

```
#include<stdio.h>

struct process
{
    char name;
    int AT,BT,WT,TAT,RT,CT;
}Q1[10],Q2[10],Q3[10];/*Three queues*/

int n;
void sortByArrival()
{
struct process temp;
int i,j;
for(i=0;i<n;i++)
    {
        for(j=i+1;j<n;j++)
            {
                if(Q1[i].AT>Q1[j].AT)
                    {
                          temp=Q1[i];
                          Q1[i]=Q1[j];
                          Q1[j]=temp;
                    }
            }
    }
}

int main()
{
     int i,j,k=0,r=0,time=0,tq1=5,tq2=8,flag=0;
     char c;
     printf("Enter no of processes:");
     scanf("%d",&n);
     for(i=0,c='A';i<n;i++,c++)
     {
         Q1[i].name=c;
         printf("\nEnter the arrival time and burst time of process %c: ",Q1[i].name);
         scanf("%d%d",&Q1[i].AT,&Q1[i].BT);
         Q1[i].RT=Q1[i].BT;/*save burst time in remaining time for each process*/

    }
sortByArrival();
time=Q1[0].AT;
printf("Process in first queue following RR with qt=5");
printf("\nProcess\t\tRT\t\tWT\t\tTAT\t\t");
for(i=0;i<n;i++)
{

  if(Q1[i].RT<=tq1)
  {

       time+=Q1[i].RT;/*from arrival time of first process to completion of this process*/
       Q1[i].RT=0;
       Q1[i].WT=time-Q1[i].AT-Q1[i].BT;/*amount of time process has been waiting in the first queue*/
       Q1[i].TAT=time-Q1[i].AT;/*amount of time to execute the process*/
       printf("\n%c\t\t%d\t\t%d\t\t%d",Q1[i].name,Q1[i].BT,Q1[i].WT,Q1[i].TAT);

  }
  else/*process moves to queue 2 with qt=8*/
  {
      Q2[k].WT=time;
      time+=tq1;
      Q1[i].RT-=tq1;
      Q2[k].BT=Q1[i].RT;
      Q2[k].RT=Q2[k].BT;
      Q2[k].name=Q1[i].name;
      k=k+1;
      flag=1;
   }
}
if(flag==1)
 {printf("\nProcess in second queue following RR with qt=8");
  printf("\nProcess\t\tRT\t\tWT\t\tTAT\t\t");
}for(i=0;i<k;i++)
   {
    if(Q2[i].RT<=tq2)
     {
       time+=Q2[i].RT;/*from arrival time of first process +BT of this process*/
       Q2[i].RT=0;
       Q2[i].WT=time-tq1-Q2[i].BT;/*amount of time process has been waiting in the ready queue*/
```

```
      Q2[i].TAT=time-Q2[i].AT;/*amount of time to execute the process*/
      printf("\n%c\t\t%d\t\t%d\t\t%d",Q2[i].name,Q2[i].BT,Q2[i].WT,Q2[i].TAT);

    }
    else/*process moves to queue 3 with FCFS*/
    {
      Q3[r].AT=time;
      time+=tq2;
      Q2[i].RT-=tq2;
      Q3[r].BT=Q2[i].RT;
      Q3[r].RT=Q3[r].BT;
      Q3[r].name=Q2[i].name;
      r=r+1;
      flag=2;
    }
  }

{if(flag==2)
printf("\nProcess in third queue following FCFS ");
}
for(i=0;i<r;i++)
{
    if(i==0)
            Q3[i].CT=Q3[i].BT+time-tq1-tq2;
        else
            Q3[i].CT=Q3[i-1].CT+Q3[i].BT;

}

for(i=0;i<r;i++)
    {
        Q3[i].TAT=Q3[i].CT;
        Q3[i].WT=Q3[i].TAT-Q3[i].BT;
        printf("\n%c\t\t%d\t\t%d\t\t%d\t\t",Q3[i].name,Q3[i].BT,Q3[i].WT,Q3[i].TAT);

    }

}
```

SNPSHOT OF THE OUTPUT WINDOW

```
PS C:\Users\USER\javavs> cd "c:\Users\USER\javavs\OS LAB\CPU SCHEDULING\" ; if ($?) { gcc multilevel_que
\multilevel_queue_feedback }
Enter no of processes:6

Enter the arrival time and burst time of process A: 0 30

Enter the arrival time and burst time of process B: 1 35

Enter the arrival time and burst time of process C: 7 15

Enter the arrival time and burst time of process D: 30 5

Enter the arrival time and burst time of process E: 32 10

Enter the arrival time and burst time of process F: 56 30
Process in first queue following RR with qt=5
Process         RT              WT              TAT
D               5               -15             -10
Process in second queue following RR with qt=8
Process         RT              WT              TAT
E               5               49              59
Process in third queue following FCFS
A               17              54              71
B               22              71              93
C               2               93              95
F               17              95              112
```

CONCLUSION

This algorithm has interconnected queues with different time quantams which makes the CPU less ideal and the utilization of CPU is enhanced because everytime a process gets executed its goes on to the next queue and the process gets divided in to many burst times and finally the process gets executed and starvation is avoided. This are the things i can conclude while i was into this assignment.

Assignment 11

Write a C program to implement Multilevel Queue Scheduling Algorithm.

PROGRAM DESCRIPTION:-

Multilevel Queue scheduling is similar to feedback scheduling but one difference is that the 3 queues are not connected to each other ,in other words this means that every process willhave a certain priority to which queue it must go and execute . execution of each queue is separate and non is affected by the other.

VARIABLE DESCRIPTION:

Variable name	Variable type	Variable Description
i , j	int	Loop variables
n	int	Input and store the number of processes
current_time	int	time counter for calculating the current time of the scheduling processes
is_complete[]	int	Array to store the indication that the process is completely executed or not
at[]	int	Array to store the arrival time of the processes
at1[],at2[],at3[]	int	To store the arrival times of the processes getting distributed to different queues (1) for round robin (2) for Non Preemptive priority & (3) for FCFS schedulings
bt1[],bt2[],bt3[]	int	To store the Burst times of the processes getting distributed to different queues (1) for round robin (2) for Non Preemptive priority & (3) for FCFS schedulings
priority2[]	int	To set priority to which queue the process should enter
bt[]	int	Array to store the burst time of the processes
tat1[],tat2[],tat3[]	int	Array to store the turn around time of the processes
comp1[],comp2[], comp3[]	int	Array to store the completion time of the processes
wt1[],wt2[],wt3[]	int	Array to store the waiting time of the processes
awt1,awt2,awt3	float	To store the average waiting of all the processes
atat1,atat2,atat3	float	To store the average turn around time of all the process
art1,art2	float	To store the response average response of the processes
completed	int	Counter variable
idx	int	To store the index of the process of minimum burst time
j,k,l	int	To store the count of the number of processes in a particular queue
burst_remianing[]	int	To store the remaining burst time left before its get preempted.
tq	int	To store the time quantam for the round robin process

CODE FOR MULTILEVEL QUEUE SCHEDULING ALGORITHM:-

```
#include<stdio.h>
#include<String.h>
int at1[30],at2[30],priority2[30],process2[30],at3[30];
int bt1[30],bt2[30],bt3[30];

int Total=0,t1=0,t2=0,t3=0;

int n,i=0,at[30],bt[30],pr[30],j=0,k=0,l=0;

int tq,n,i,idx,j,wt1[30],tat1[30],comp1[30],st1[30],rt1[30],p1[30];
 float atat1=0,awt1=0,art1=0;
int burst_remaining[30];
int q[100],front=-1,rear=-1;

int p,wt3[30],tat3[30],comp3[30];
float awt3=0.0,atat3=0.0;

int wt2[30],tat2[30],comp2[30],st2[30],rt2[30];
float awt2=0.0,atat2=0.0,art2=0.0;

int current_time=0;
void swap(int *a,int f)
{
  int temp=0;
  temp=a[f];
  a[f]=a[f+1];
  a[f+1]=temp;
}
void push(int v,int x)
{
  if(rear==x-1)
  printf("\n");
  else
  {
    if(front==-1)
    front=0;
    rear++;
    q[rear]=v;
  }
}
void pop()
{
  if(front==-1)
  printf("\n");
  else
  {
    front++;
    if(front>rear)
    front=rear=-1;
  }
}
int empty()
{
  if(front==-1)
  return 1;
  else
  return 0;
}
int front1()
{
  if(front==-1)
   printf("\n");
   else
   return q[front];

}



void round_robin()
{
  printf("\n\n Round Robin\n");
  int temp=0,idx=0,tq=2;
  int mark[30];
  for(int m=0;m<n;m++)
  {
    for(int f=0;f<j-m-1;m++)
    {
      if(at1[f]>at1[f+1])
      {
```

```
      swap(at1,f);
      swap(p1,f);
      swap(bt1,f);
      swap(burst_remaining,f);
    }
  }
}

 //int current_time=0;
 push(0,j);
 int completed=0;
 memset(mark,0,sizeof(mark));
 mark[0]=1;
 while(completed!=j)
 {
   idx=front1();
   pop();
   if(burst_remaining[idx]==bt1[idx])
   {
     st1[idx]=(current_time>at1[idx])?current_time:at1[idx];
     current_time=st1[idx];
   }
   if(burst_remaining[idx]-tq>0)
   {
     burst_remaining[idx]-=tq;
     current_time+=tq;
   }
   else{
     current_time+=burst_remaining[idx];
     burst_remaining[idx]=0;
     completed++;
     comp1[idx]=current_time;
     tat1[idx]=comp1[idx]-at1[idx];
     wt1[idx]=tat1[idx]-bt1[idx];
     rt1[idx]=st1[idx]-at1[idx];

     awt1+=wt1[idx];
     atat1+=tat1[idx];
     art1+=rt1[idx];

   }
   for(int m=1;m<n;m++)
   {
     if(burst_remaining[m]>0 && at1[m]<=current_time && mark[m]==0){
       push(m,j);
       mark[m]=1;
     }
   }
   if(burst_remaining[idx]>0)
   push(idx,j);

   if(empty()==1)
   {
     for(int m=1;m<n;m++)
   {
     if(burst_remaining[m]>0){
       push(m,j);
       mark[m]=1;
       break;
     }
   }
   }
 }
   temp=0;
 for(int m=0;m<j;m++)
 {
   for(int f=0;f<j-m-1;f++)
   {
     if(p1[f]>p1[f+1])
     {
       swap(at1,f);
       swap(p1,f);
       swap(bt1,f);
       swap(wt1,f);
       swap(rt1,f);
       swap(tat1,f);
       swap(comp1,f);
     }
   }
 }
 printf("PROCESS\tARRIVAL TIME\tBURST TIME\tCOMPLETION TIME\tWAITING TIME\tTURN AROUND TIME\tRESPONSE TIME\n");
 for(int m=0;m<j;m++)
```

```
   {
      printf("%d\t%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t\t%d\n",p1[m],at1[m],bt1[m],comp1[m],wt1[m],tat1[m],rt1[m]);
   }
   printf("Average waiting time => %f",awt1/j);
   printf("\nAverage Turn around time => %f",atat1/j);
    printf("\nAverage Response time=> %f",art1/j);
}

void priority()
{
  printf("\n\n Priority non preemptive\n");
  //int current_time=0;
  int completed=0;
  int prev=0;
  int is_complete[30];
  memset(is_complete,0,sizeof(is_complete));
  int m=0;
  while(completed!=k)
  {
    int idx=-1;
    int mx=-1;
    for(m=0;m<k;m++)
    {
      if(at2[m]<=current_time && is_complete[m]==0)
      {
        if(priority2[m]>mx)
        {
          mx=priority2[m];
          idx=m;
        }
        if(priority2[m]==mx)
        {
          if(at2[m]<at2[idx])
          {
            mx=priority2[m];
            idx=m;
          }
        }
      }
    }
    if(idx!=-1)
    {
     st2[idx]=current_time;
     comp2[idx]=current_time+bt2[idx];
     tat2[idx]=comp2[idx]-at2[idx];
     wt2[idx]=tat2[idx]-bt2[idx];
     rt2[idx]=st2[idx]-at2[idx];

     awt2+=wt2[idx];
     atat2+=tat2[idx];
     art2+=rt2[idx];
     is_complete[idx]=1;
     completed++;
     current_time=comp2[idx];
     prev=current_time;

    }
    else
    {
      current_time++;
    }
  }
  printf("PROCESS  PRIORITY  AT\t  BT\t  CT\t  WT\t  TAT\t  RT\n");
  for(int i=0;i<k;i++)
   {
      printf("%d\t %d\t   %d\t   %d\t   %d\t   %d\t   %d\t   %d\n",process2[i],priority2[i],at2[i],bt2[i],comp2[i],wt2[
i],tat2[i],rt2[i]);
   }
   printf("Average waiting time => %f",awt2/k);
   printf("\nAverage Turn around time => %f",atat2/k);
   printf("\nAverage Response time=> %f",art2/k);

}

void fcfs()
{
  printf("\n\n First cum First serve\n");
  int temp=0,m=0,f=0,completed=0;
  int is_complete[30];
  memset(is_complete,0,sizeof(is_complete));
  for(f=0;f<l;f++)
  {
```

```
  for(m=0;j<l-f-1;m++)
  {
    if(at3[m]>at3[m+1])
    {
      temp=at3[m];
      at3[m]=at3[m+1];
      at3[m+1]=temp;
      temp=bt3[m];
      bt3[m]=bt3[m+1];
      bt3[m+1]=temp;

    }
  }
}

printf("PROCESS\tARRIVAL TIME\t BURST TIME\tCOMPLETION TIME\tWAITING TIME\tTURN AROUND TIME\n");
for(m=0;m<l;m++)
{
  wt3[m]=0;
  tat3[m]=0;
  comp3[m]=0;
  if(at3[m]>current_time)
  {
    current_time++;
    m--;
    continue;
   }
   current_time+=bt3[m];
   comp3[m]=current_time;
   tat3[m]=comp3[m]-at3[m];
   wt3[m]=tat3[m]-bt3[m];
   awt3=awt3+wt3[m];
   atat3=atat3+tat3[m];
   printf("%d\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n",m+1,at3[m],bt3[m],comp3[m],wt3[m],tat3[m]);

  }
  printf("Average waiting time => %f",awt3/l);
  printf("\nAverage Turn around time => %f",atat3/l);
  printf("\n");
}



void round_robin1()
{
  printf("Time Quantum between the 3 queues is 10\n");
  for(i=1;i<Total;i=i+10)
  {
    if(t1>10)
    {
      printf("Queue1 is running for 10 units\n");
      t1=t1-10;
    }
    else if(t1<=10&&t1!=0)
    {
      printf("Queue1 is running for %d units\n",t1);
      t1=0;
    }
    if(t2>10)
    {
      printf("Queue2 is running for 10 units\n");
      t2=t2-10;
    }
    else if(t2<=10&&t2!=0)
    {
      printf("Queue2 is running for %d units\n",t2);
      t2=0;
    }
    if(t3>10)
    {
      printf("Queue3 is running for 10 units\n");
      t3=t3-10;
    }
    else if(t3<=10&&t3!=0)
    {
      printf("Queue3 is running for %d units\n",t3);
      t3=0;
    }
  }
}

int main()
```

```
{
  printf("Enter the no. of process you want to enter\n");
  scanf("%d",&n);
  for(i=0;i<n;i++)
  {
    printf("Enter details of process[%d]\n",i+1);
    printf("Arrival Time:");
    scanf("%d",&at[i]);
    printf("Burst Time:");
    scanf("%d",&bt[i]);
    printf("Priority(1 to 15):");
    scanf("%d",&pr[i]);
    Total=Total+bt[i];
  }
  for(i=0;i<n;i++)
  {
    if(pr[i]>=1&&pr[i]<=5)
    {
      printf("Process[%d] belongs to Queue 1\n",i+1);
      at1[j]=at[i];
      bt1[j]=bt[i];
      burst_remaining[j]=bt[i];
      p1[j]=j+1;
      j++;
      t1=t1+bt[i];
    }

    else if(pr[i]>=6&&pr[i]<=10)
    {
      printf("Process[%d] belongs to Queue 2\n",i+1);
      at2[k]=at[i];
      bt2[k]=bt[i];
      priority2[k]=pr[i];
      process2[k]=k+1;
      k++;
      t2=t2+bt[i];
    }

    else if(pr[i]>=11&&pr[i]<=15)
    {
      printf("Process[%d] belongs to Queue 3\n",i+1);
      at3[l]=at[i];
      bt3[l]=bt[i];

      l++;
      t3=t3+bt[i];
    }
  }

  round_robin1();
  round_robin();
  priority();
  fcfs();



}
```

SAMPLE INPUT:-

Enter the no. of process you want to enter

9

Enter details of process[1]

Arrival Time:0

Burst Time:4

Priority(1 to 15):3

Enter details of process[2]

Arrival Time:3

Burst Time:6

Priority(1 to 15):2

Enter details of process[3]

Arrival Time:7

Burst Time:3

Priority(1 to 15):1

Enter details of process[4]

Arrival Time:8

Burst Time:4

Priority(1 to 15):7

Enter details of process[5]

Arrival Time:12

Burst Time:4

Priority(1 to 15):8

Enter details of process[6]

Arrival Time:15

Burst Time:5

Priority(1 to 15):6

Enter details of process[7]

Arrival Time:10

Burst Time:3

Priority(1 to 15):11

Enter details of process[8]

Arrival Time:14

Burst Time:5

Priority(1 to 15):12

Enter details of process[9]

Arrival Time:17

Burst Time:3

Priority(1 to 15):13

SAMPLE OUTPUT:-

Process[1] belongs to Queue 1

Process[2] belongs to Queue 1

Process[3] belongs to Queue 1

Process[4] belongs to Queue 2

Process[5] belongs to Queue 2

Process[6] belongs to Queue 2

Process[7] belongs to Queue 3

Process[8] belongs to Queue 3

Process[9] belongs to Queue 3

Time Quantum between the 3 queues is 10

Queue1 is running for 10 units

Queue2 is running for 10 units

Queue3 is running for 10 units

Queue1 is running for 3 units

Queue2 is running for 3 units

Queue3 is running for 1 units

Round Robin

PROCESS	ARRIVAL TIME	BURST TIME	COMPLETION TIME	WAITING TIME	TURN AROUND TIME	RESPONSE TIME
1	0	4	4	0	4	0
2	3	6	12	3	9	1
3	7	3	13	3	6	1

Average waiting time => 2.000000
Average Turn around time => 6.333333
Average Response time=> 0.666667

Current time-> 13

Priority Non Preemptive

PROCESS	PRIORITY	ARRIVAL TIME	BURST TIME	COMPLETION TIME	WAITING TIME	TURN AROUND TIME	RESPONSE TIME
1	7	8	4	21	9	13	9
2	8	12	4	17	1	5	1
3	6	15	5	26	6	11	6

Average waiting time => 5.333333
Average Turn around time => 9.666667
Average Response time=> 5.333333

Current Time -> 26

First come First Serve

PROCESS	ARRIVAL TIME	BURST TIME	COMPLETION TIME	WAITING TIME	TURN AROUND TIME
1	10	3	29	16	19
2	14	5	34	15	20
3	17	3	37	17	20

Average waiting time => 16.000000

Average Turn around time => 19.666667

Current time -> 37

SNAPSHOT OF THE OUTPUT TERMINAL

```
9
Enter details of process[1]
Arrival Time:0
Burst Time:4
Priority(1 to 15):3
Enter details of process[2]
Arrival Time:3
Burst Time:6
Priority(1 to 15):2
Enter details of process[3]
Arrival Time:7
Burst Time:3
Priority(1 to 15):1
Enter details of process[4]
Arrival Time:8
Burst Time:4
Priority(1 to 15):7
Enter details of process[5]
Arrival Time:12
Burst Time:4
Priority(1 to 15):8
Enter details of process[6]
Arrival Time:15
Burst Time:5
Priority(1 to 15):6
Enter details of process[7]
Arrival Time:10
Burst Time:3
Priority(1 to 15):11
Enter details of process[8]
Arrival Time:14
Burst Time:5
Priority(1 to 15):12
Enter details of process[9]
Arrival Time:17
Burst Time:3
Priority(1 to 15):13
```

```
 Process[1] belongs to Queue 1
 Process[2] belongs to Queue 1
 Process[3] belongs to Queue 1
 Process[4] belongs to Queue 2
 Process[5] belongs to Queue 2
 Process[6] belongs to Queue 2
 Process[7] belongs to Queue 3
 Process[8] belongs to Queue 3
 Process[9] belongs to Queue 3
 Time Quantum between the 3 queues is 10
 Queue1 is running for 10 units
 Queue2 is running for 10 units
 Queue3 is running for 10 units
 Queue1 is running for 3 units
 Queue2 is running for 3 units
 Queue3 is running for 1 units
```

```
 Round Robin
PROCESS ARRIVAL TIME    BURST TIME      COMPLETION TIME WAITING TIME    TURN AROUND TIME        RESPONSE TIME
1       0               4               4               0               4                       0
2       3               6               12              3               9                       1
3       7               3               13              3               6                       1
Average waiting time => 2.000000
Average Turn around time => 6.333333
Average Response time=> 0.666667

 Priority non preemptive
PROCESS  PRIORITY  AT     BT      CT      WT      TAT     RT
1        7         8       4       21      9       13      9
2        8         12      4       17      1       5       1
3        6         15      5       26      6       11      6
Average waiting time => 5.333333
Average Turn around time => 9.666667
Average Response time=> 5.333333

 First cum First serve
PROCESS ARRIVAL TIME     BURST TIME     COMPLETION TIME WAITING TIME    TURN AROUND TIME
1       10              3               29              16              19
2       14              5               34              15              20
3       17              3               37              17              20
Average waiting time => 16.000000
Average Turn around time => 19.666667
```

CONCLUSION:-

This scheduling is more time taking than feedback but it has some advantages over other scheduling algorithm . This algorithm avoids starvation and idealness of CPU is reduced. So something or the other way this algorithm can be used as one of the prime algorithm to schedule process.

ASSIGNMENT 12

Write a C program to implement Banker's Algorithm

ABOUT BANKER'S ALGORITHM:-

Banker's Algorithm is used to determine whether a process's request for allocation of resources be safely granted immediately.

The grant of request be deferred to a later stage. For the banker's algorithm to operate, each process has to a priori specify its maximum requirement of resources.

A process is admitted for execution only if its maximum requirement of resources is within the system capacity of resources.

The Banker's algorithm is an example of resource allocation policy that avoids deadlock.

CODE:-

```c
#include <stdio.h>
#include <conio.h>
#include <math.h>
#include <stdlib.h>
#include <string.h>
void main()
{
  int n;
  printf("Enter the number of processes\n");
  scanf("%d",&n);
  int i,j,Allocation[n][3],MaxNeed[n][3],Available[3],RemainingNeed[n][3],count=0,c=0;
  int is_complete[n];
  memset(is_complete,0,sizeof(is_complete));
  printf("Enter the allocations\n");
  for(i=0;i<n;i++){
    printf("Process %d\n",(i+1));
    for(j=0;j<3;j++)
    {
      scanf("%d",&Allocation[i][j]);
    }
  }
  printf("Enter the Max Need\n");
  for(i=0;i<n;i++){
    printf("Process %d\n",(i+1));
    for(j=0;j<3;j++){
      scanf("%d",&MaxNeed[i][j]);
    }
  }
  printf("Available resources\n");
  for(i=0;i<3;i++){
    scanf("%d",&Available[i]);
  }

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

    for(j=0;j<3;j++){
     RemainingNeed[i][j]=MaxNeed[i][j]-Allocation[i][j];
    }
  }
  i=0;
  while(count!=n){
    i=i%n;
    if(is_complete[i]==0){
      c=0;
      for(int j=0;j<3;j++){
          if(RemainingNeed[i][j]<=Available[j]){
            c++;
          }
        }
      if(c==3){
        printf("P(%d) --> ",(i+1));
        count++;
        is_complete[i]=1;
        for(int j=0;j<3;j++){
            Available[j]+=Allocation[i][j];
        }}}
    i++;
  }
}
```

SNAPSHOT OF THE OUTPUT WINDOW

```
PS C:\Users\USER\javavs> cd "c:\Users\USER\javavs\OS LAB\" ; if ($?) { gcc b
Enter the number of processes
5
Enter the allocations
Process 1
0 1 0
Process 2
2 0 0
Process 3
3 0 2
Process 4
2 1 1
Process 5
0 0 2
Enter the Max Need
Process 1
7 5 3
Process 2
3 2 2
Process 3
9 0 2
Process 4
4 2 2
Process 5
5 3 3
Available resources
3 3 2
P(2) --> P(4) --> P(5) --> P(1) --> P(3) -->
```

CONCLUSION

Banker's Algorithm has taught me something that in this algorithm as long as the available resources are greater than or equal to the remaining need we will have a safe sequence but if non remaining resource table is less than or equal to the available resources we shall have an unsafe state or deadlock in simple words. This is the conclusion i can give.

Assignment 13

Write a C program to implement Producer Consumer problem

PROGRAM DESCRIPTION:-

Producer consumer problem is a problem where we are provided with a buffer (array in precise). In here producer makes a product and stores it in the buffer while the consumer comsumes the product in a sequence and deletes out the consumed product . this deletion and addition of product is the main concept of producer consumer problem. In here no dead lock should happen so semaphore is used so that only one of the either the producer or the consumer will use the critical section.

CODE FOR PRODUCER CONSUMER PROBLEM

```c
#include<stdio.h>
#include<stdlib.h>

int mutex=1,full=0,empty=3,x=0;

int main()
{
  int n;
  void producer();
  void consumer();
  int wait(int);
  int signal(int);
  printf("\n1.Producer\n2.Consumer\n3.Exit");
  while(1)
  {
    printf("\nEnter your choice:");
    scanf("%d",&n);
    switch(n)
    {
      case 1: if((mutex==1)&&(empty!=0))
            producer();
          else
            printf("Buffer is full!!");
          break;
      case 2: if((mutex==1)&&(full!=0))
            consumer();
          else
            printf("Buffer is empty!!");
          break;
      case 3:
          exit(0);
          break;
    }
  }

  return 0;
}

int wait(int s)
{
  return (--s);
}

int signal(int s)
{
  return(++s);
}

void producer()
{
  mutex=wait(mutex);
  full=signal(full);
  empty=wait(empty);
  x++;
  printf("\nProducer produces the item %d",x);
  mutex=signal(mutex);
}
```

```
void consumer()
{
  mutex=wait(mutex);
  full=wait(full);
  empty=signal(empty);
  printf("\nConsumer consumes item %d",x);
  x--;
  mutex=signal(mutex);
```

SNAPSHOT OF THE OUTPUT TERMINAL:

```
PS C:\Users\USER\javavs> cd "c:\Users\USER\javavs\OS LAB\" ; if ($?) { gcc producerConsumer.c -o pro

1.Producer
2.Consumer
3.Exit
Enter your choice:1

Producer produces the item 1
Enter your choice:2

Consumer consumes item 1
Enter your choice:2
Buffer is empty!!
Enter your choice:1

Producer produces the item 1
Enter your choice:2

Consumer consumes item 1
Enter your choice:3
```

CONCLUSION

This problem teaches a lot about process synchronization . This is so because in thus problem there were two function or methods of producer function and the consumer function . to ensure no deadlock should occur certain maeasures were taken and process synchronizing was achieved .So this was the idea which i achieved while i was working on this problem.

Assignment 14

Write a C program to implement Dinning Philosopher Problem

PROGRAM DESCRIPTION:-

The dining philosophers problem states that there are 5 philosophers sharing a circular table and they eat and think alternatively. There is a bowl of rice for each of the philosophers and 5 chopsticks. A philosopher needs both their right and left chopstick to eat. A hungry philosopher may only eat if there are both chopsticks available.Otherwise a philosopher puts down their chopstick and begin thinking again.

The dining philosopher is a classic synchronization problem as it demonstrates a large class of concurrency control problems.

CODE FOR DINNING PHILOSOPHER PROBLEM

```c
#include<stdio.h>

#define n 4

int compltedPhilo = 0,i;

struct fork{
  int taken;
}ForkAvil[n];

struct philosp{
  int left;
  int right;
}Philostatus[n];

void goForDinner(int philID){
  if(Philostatus[philID].left==10 && Philostatus[philID].right==10)
        printf("Philosopher %d completed his dinner\n",philID+1);

  else if(Philostatus[philID].left==1 && Philostatus[philID].right==1){

            printf("Philosopher %d completed his dinner\n",philID+1);

            Philostatus[philID].left = Philostatus[philID].right = 10;
            int otherFork = philID-1;

            if(otherFork== -1)
                otherFork=(n-1);

        ForkAvil[philID].taken = ForkAvil[otherFork].taken = 0;
        printf("Philosopher %d released fork %d and fork %d\n",philID+1,philID+1,otherFork+1);
        compltedPhilo++;
    }
    else if(Philostatus[philID].left==1 && Philostatus[philID].right==0){
            if(philID==(n-1)){
                if(ForkAvil[philID].taken==0){
                    ForkAvil[philID].taken = Philostatus[philID].right = 1;
                    printf("Fork %d taken by philosopher %d\n",philID+1,philID+1);
                }else{
                    printf("Philosopher %d is waiting for fork %d\n",philID+1,philID+1);
                }
            }else{
                int dupphilID = philID;
                philID-=1;

                if(philID== -1)
                    philID=(n-1);

                if(ForkAvil[philID].taken == 0){
                    ForkAvil[philID].taken = Philostatus[dupphilID].right = 1;
                    printf("Fork %d taken by Philosopher %d\n",philID+1,dupphilID+1);
                }else{
                    printf("Philosopher %d is waiting for Fork %d\n",dupphilID+1,philID+1);
                }
            }
```

```
            }
            else if(Philostatus[philID].left==0){
                    if(philID==(n-1)){
                        if(ForkAvil[philID-1].taken==0){
                            ForkAvil[philID-1].taken = Philostatus[philID].left = 1;
                            printf("Fork %d taken by philosopher %d\n",philID,philID+1);
                        }else{
                            printf("Philosopher %d is waiting for fork %d\n",philID+1,philID);
                        }
                    }else{
                        if(ForkAvil[philID].taken == 0){
                            ForkAvil[philID].taken = Philostatus[philID].left = 1;
                            printf("Fork %d taken by Philosopher %d\n",philID+1,philID+1);
                        }else{
                            printf("Philosopher %d is waiting for Fork %d\n",philID+1,philID+1);
                        }
                    }
        }else{}
}

int main(){
  for(i=0;i<n;i++)
        ForkAvil[i].taken=Philostatus[i].left=Philostatus[i].right=0;

  while(compltedPhilo<n){

    for(i=0;i<n;i++)
            goForDinner(i);
    printf("\nTill now num of philosophers completed dinner are %d\n\n",compltedPhilo);
  }

  return 0;
}
```

SNAPSHOT OF THE OUTPUT WINDOW

```
PS C:\Users\USER\javavs\OS LAB> cd "c:\Users\USER\javavs\OS LAB\" ; if ($?) { gcc dinningp
Fork 1 taken by Philosopher 1
Fork 2 taken by Philosopher 2
Fork 3 taken by Philosopher 3
Philosopher 4 is waiting for fork 3

Till now num of philosophers completed dinner are 0

Fork 4 taken by Philosopher 1
Philosopher 2 is waiting for Fork 1
Philosopher 3 is waiting for Fork 2
Philosopher 4 is waiting for fork 3

Till now num of philosophers completed dinner are 0

Philosopher 1 completed his dinner
Philosopher 1 released fork 1 and fork 4
Fork 1 taken by Philosopher 2
Philosopher 3 is waiting for Fork 2
Philosopher 4 is waiting for fork 3

Till now num of philosophers completed dinner are 1

Philosopher 1 completed his dinner
Philosopher 2 completed his dinner
Philosopher 2 released fork 2 and fork 1
Fork 2 taken by Philosopher 3
Philosopher 4 is waiting for fork 3
```

```
Till now num of philosophers completed dinner are 1

Philosopher 1 completed his dinner
Philosopher 2 completed his dinner
Philosopher 2 released fork 2 and fork 1
Fork 2 taken by Philosopher 3
Philosopher 4 is waiting for fork 3

Till now num of philosophers completed dinner are 2

Philosopher 1 completed his dinner
Philosopher 2 completed his dinner
Philosopher 3 completed his dinner
Philosopher 3 released fork 3 and fork 2
Fork 3 taken by philosopher 4

Till now num of philosophers completed dinner are 3

Philosopher 1 completed his dinner
Philosopher 2 completed his dinner
Philosopher 3 completed his dinner
Fork 4 taken by philosopher 4

Till now num of philosophers completed dinner are 3

Philosopher 1 completed his dinner
Philosopher 2 completed his dinner
Philosopher 3 completed his dinner
Philosopher 4 completed his dinner
Philosopher 4 released fork 4 and fork 3

Till now num of philosophers completed dinner are 4
```

CONCLUSION

The above solution makes sure that no two neighboring philosophers can eat at the same time. But this solution can lead to a deadlock. This may happen if all the philosophers pick their left chopstick simultaneously. Then none of them can eat and deadlock occurs.

Some of the ways to avoid deadlock are as follows

There should be at most four philosophers on the table.

An even philosopher should pick the right chopstick and then the left chopstick while an odd philosopher should pick the left chopstick and then the right chopstick.

A philosopher should only be allowed to pick their chopstick if both are available at the same time.

Assignment 15

Write a C program to implement FCFS Page replacement algorithm

PROGRAM DESCRIPTION:-

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

The main advantage of the FIFO page replacement algorithm is its simplicity. It is easy to understand and implement. It also uses a queue data structure. The number of operations is limited in a queue makes the implementation simple.

Let’s talk about some disadvantages now. When the number of incoming pages is large, it might not provide excellent performance.

When we increase the number of frames or capacity to store pages in the queue, it should give us less number of page faults. Sometimes FIFO may behave abnormally, and it may increase the number of page faults. This behavior of FIFO is called.

.

CODE FOR FCFS PAGE REPLACEMENT ALGORITHM

```c
#include <stdio.h>
#include <conio.h>
#include <math.h>
#include <stdlib.h>
#include <string.h>
#define max 20
#define f 3
void main()
{
  int refstr[max],i,j,k,l,len,farr[f],miss=0,hit=0;
  float hr,mr;
  memset(farr,-1,sizeof(farr));
  printf("Enter the length of the reference string\n");
  scanf("%d",&len);
  printf("Enter the reference string\n");
  for(i=0;i<len;i++)
  {
    scanf("%d ",&refstr[i]);
  }
  j=0;
  int count=0;
  for(i=0;i<len;i++)
  {
    j=j%f;
    count=0;
    for(k=0;k<f;k++)
    {
      if(farr[k]==refstr[i])
      {
        count=1;
        hit=hit+1;
        break;
      }

    }
    if(count!=1)
    {
      miss=miss+1;
      farr[j]=refstr[i];
      j++;
    }
    if(count==1)
    {
      printf("{Hit} -> ");
    }
    else
    printf("Miss ->  ");
    for(l=0;l<f;l++)
    {
      if(farr[l]!=-1)
```

```
      printf("%d ",farr[l]);
      else
      printf(" ");
    }
    printf("\n\n");

  }
  float x=len;
   hr=(hit/x)*100;
   mr=(miss/x)*100;

  printf("\nHit ratio --> %f\n",hr);
  printf("\nMiss ratio --> %f\n",mr);



}
```

SNAPSHOT OF THE OUTPUT WINDOW

```
Enter the reference string
7 0 1 2 0 3 0 4 2 3 0 3 1 2 0
1
Miss ->  7

Miss ->  7 0

Miss ->  7 0 1

Miss ->  2 0 1

{Hit} -> 2 0 1

Miss ->  2 3 1

Miss ->  2 3 0

Miss ->  4 3 0

Miss ->  4 2 0

Miss ->  4 2 3

Miss ->  0 2 3

{Hit} -> 0 2 3

Miss ->  0 1 3

Miss ->  0 1 2

{Hit} -> 0 1 2


Hit ratio --> 20.000000

Miss ratio --> 80.000000
```

CONCLUSION

FIFO page replacement algorithm is definitely not the best page replacement algorithm to use practically. When the number of incoming pages is less, and a user is looking for a simple approach, FIFO might be a reasonable choice.

Assignment 16

Write a C program to implement Optimal page Replacement problem

PROGRAM DESCRIPTION:-

In this algorithm, pages are replaced which would not be used for the longest duration of time in the future. Optimal page replacement is perfect, but not possible in practice as the operating system cannot know future requests. The use of Optimal Page replacement is to set up a benchmark so that other replacement algorithms can be analyzed against it.

.

CODE FOR OPTIMAL PAGE REPLACEMENT ALGORITHM

```c
#include<stdio.h>

int main()
{
    int no_of_frames, no_of_pages, frames[10], pages[30], temp[10], flag1, flag2, flag3, i, j, k, pos, max, faults = 0;
    printf("Enter number of frames: ");
    scanf("%d", &no_of_frames);

    printf("Enter number of pages: ");
    scanf("%d", &no_of_pages);

    printf("Enter page reference string: ");

    for(i = 0; i < no_of_pages; ++i){
        scanf("%d", &pages[i]);
    }

    for(i = 0; i < no_of_frames; ++i){
        frames[i] = -1;
    }

    for(i = 0; i < no_of_pages; ++i){
        flag1 = flag2 = 0;

        for(j = 0; j < no_of_frames; ++j){
            if(frames[j] == pages[i]){
                    flag1 = flag2 = 1;
                    break;
                }
        }

        if(flag1 == 0){
            for(j = 0; j < no_of_frames; ++j){
                if(frames[j] == -1){
                    faults++;
                    frames[j] = pages[i];
                    flag2 = 1;
                    break;
                }
            }
        }

        if(flag2 == 0){
            flag3 =0;

            for(j = 0; j < no_of_frames; ++j){
                temp[j] = -1;

                for(k = i + 1; k < no_of_pages; ++k){
                    if(frames[j] == pages[k]){
                        temp[j] = k;
                        break;
                    }
                }
            }

            for(j = 0; j < no_of_frames; ++j){
                if(temp[j] == -1){
                    pos = j;
```

```
                flag3 = 1;
                break;
            }
        }

        if(flag3 ==0){
            max = temp[0];
            pos = 0;

            for(j = 1; j < no_of_frames; ++j){
                if(temp[j] > max){
                    max = temp[j];
                    pos = j;
                }
            }
        }

        frames[pos] = pages[i];
        faults++;
    }

    printf("\n");

    for(j = 0; j < no_of_frames; ++j){
        printf("%d\t", frames[j]);
        }
    }

    printf("\n\nTotal Page Faults = %d", faults);

    return 0;
}
```

SNAPSHOT OF THE OUTPUT WINDOW

```
PS C:\Users\USER\javavs> cd "c:\Users\USER\javavs\OS LAB\" ; if ($?) { gcc optimalpagereplacem
nt }
Enter number of frames: 3
Enter number of pages: 15
Enter page reference string: 7 0 1 2 0 3 0 4 2 3 0 3 1 2 0

7       -1      -1
7       0       -1
7       0       1
2       0       1
2       0       1
2       0       3
2       0       3
2       4       3
2       4       3
2       4       3
2       0       3
2       0       3
2       0       1
2       0       1
2       0       1

Total Page Faults = 8
```

CONCLUSION

An optimal page- replacement algorithm has the lowest page-fault rate of all algorithms, and will never suffer from Belady's anomaly. Such an algorithm does exist and has been called OPT or MIN. It simply replaces the page that will not be used for the longest period of time.