Student Name: Ayush Khandelwal
Student ID: 11715284
Email Address: akhandelwal680@gmail.com
GitHub Link: https://github.com/Gunnu66/OS-project.git

Operating system have many works like memory management, resource allocation and different process execution. My project is based on to make a scheduler that can schedule the processes arriving system at periodical Intervals. Every process is assigned with a fixed time slice t milliseconds a scheduler for controlling different processes for execution. If it is not able to complete its execution within the assigned time quantum, then automated timer generates an interrupt. The scheduler will select the next process in the queue and Dispatcher dispatches the process to processor for execution. There are many scheduling algorithms for processes execution. But assignment is based on scheduling with round robin.

Round Robin Scheduling:

Round robin scheduling algorithm is used to schedule process fairly each job a time slot or quantum and the interrupting the job if it is not completed by then the job come after the other job which are arrived in the quantum time that make these scheduling fairly

Round Robin scheduling has the following characteristics:

- Each process is served by CPU for a fixed time so priority is same for each one
- Starvation does not occur because of its cyclic nature
- Round robin is cyclic in nature so starvation doesn't occur
- Round robin is variant of first come, first served scheduling
- No priority, special importance given to any process or task.

Complexity: - O (n^2)

Constraints:-

1. Executing each student for a fixed time quanta

printf("\nEnter TIME QUANTUM:");

scanf("%d",&tq);

2. If Process is not able to complete its execution within the assigned time quantum, then automated timer generates an interrupt. The scheduler will select the next process in the queue and dispatcher dispatches the process to processor for execution

```
p[i].wt=time - p[i].at-p[i].bt;
// p[i].tt=time - p[i].at;
for(j=0;j<n;j++)                /*dispatcher the processes which have come   while scheduling */
{
if(p[j].at<=time && p[j].completed!=1&& isInQueue(j)!=1)
{
dispatcher(j);
```

Algorithm:

1. Take the process which occurs first and start executing the process.(for quantum time only)

2. Check if any other process request has arrived. If a process request arrives during the quantum time in which another process is executing, then add the new process to the Ready queue
3. After quantum time has passed, check for any processes in Ready queue. If ready queue is empty continue current process. If queue not empty and current process is not complete, then add current process to the end of the ready queue.
4. Take the first process from the Ready queue and start executing it (same rules)
5. Repeat all steps above from 2-5
6. If process is complete and the ready queue is empty then task is complete

After all these we get the two times which are:

1. **Turn Around Time:** total time the process exists in system.(completion time – arrival time).
2. **Waiting Time:** total time the waiting for there complete execution.(turnaround time – burst time).

Code:

Step 1: enqueue Function for Process inserting in queue

```
void enqueue(int i)
{
if(rear==100)
{
printf("overflow");
exit(0);
}                                        //inserting in queue
rear++;
q[rear]=i;
if(front==-1)
{
front=0;
}
}
```

Complexity: O (1)

Step 2: Dequeue Function for removing for Queue

```
int dequeue()
{
if(front==-1)
{
printf("underflow");
exit(0);
}                                          //removing from queue
int temp=q[front];
if(front==rear)
front=rear=-1;
else
{
front++;
}
return temp;
}
```

Complexity: O (1)

Step 3: Function for searching queue

```
int isInQueue(int i)
{int k;
for(k=front;k<=rear;k++)
{
if(q[k]==i)
return 1;
}
return 0;

}
```

Complexity: O (n)

Step 4: dispatcher function for inserting entry

int dispatcher (i)

{

enqueue(i);

}

Complexity: O(1)

Step 5: function for sorting the process

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

Complexity: O(n^2)

Step 6: function for taking user input

```
void input()
{
    printf("\n\nEnter no of processes:");
 scanf("%d",&n);
 for(i=0,c='A';i<n;i++,c++)
 {
 p[i].name=c;
 printf("\n\t------------   PROCESS %c   --------------: ",p[i].name);
 printf("\n\nEnter Arrival Time:");
 scanf("%d",&p[i].at);
 printf("\nEnter Burst Time:");
 scanf("%d",&p[i].bt);
 p[i].rt=p[i].bt;
 p[i].completed=0;
 sum_bt+=p[i].bt;

 }

printf("\nEnter TIME QUANTUM:");
scanf("%d",&tq);
}
```

Complexity: O (n)

Step 7: function for displaying the data waiting time turnaround time

```
void display()
{

printf("\n\nPROCESS\t    ARRIVAL TIME\tBURST TIME\tWAITING TIME\tTURNAROUND TIME");
for(i=0;i<n;i++)
{
p[i].tt=p[i].bt+p[i].wt;


avgwt+=p[i].wt;
 avgtt+=p[i].tt;
printf("\n%c\t\t%d\t\t%d\t\t%d\t\t%d",p[i].name,p[i].at,p[i].bt,p[i].wt,p[i].tt);
public int __cdecl printf (const char * __restrict__ _Format, ...)
printf("\nAverage waiting time:%f",avgwt/n);
printf("\nAverage Turnaround time:%f",avgtt/n);
}
```

Complexity : O(N)

Step 7: Main Function

```
int main()
{
input();
sortByArrival();
dispatcher(0);          // dispatcher the first process
printf("\t\t\nProcess execution order: ");

for(time=p[0].at;time<sum_bt;) // run until the total burst time reached
{   i=dequeue();

if(p[i].rt<=tq)
{                        /* for processes having remaining time with less than or  equal  to time quantum  */

time+=p[i].rt;
p[i].rt=0;
p[i].completed=1;
    printf(" %c ",p[i].name);
          p[i].wt=time - p[i].at-p[i].bt;
         // p[i].tt=time - p[i].at;
          for(j=0;j<n;j++)               /*dispatcher the processes which have come   while scheduling */
          {
          if(p[j].at<=time && p[j].completed!=1&& isInQueue(j)!=1)
          {
          dispatcher(j);

          }
         }
     }
    else               // more than time quantum
    {
    time += tq;
    p[i].rt -= tq;
    printf(" %c ",p[i].name);
    for(j=0;j<n;j++)    //first dispatcher the processes which have come while scheduling
          {
          if(p[j].at<=time && p[j].completed!=1&&i!=j&& isInQueue(j)!=1)
          {
          dispatcher(j);

          }
         }
         dispatcher(i);   // then dispatcher the uncompleted process

    }
}
display();
return 0;
```

Complexity: O (n^2)

Test cases: time quantum **=1**

Process	Burst Time	Arrival time
P1	**1**	**0**
P2	**10**	**1**
P3	**8**	**2**
P4	**4**	**5**

Output –

Process execution order: A B C B C B D C B D C B D C B D C B C B C B B

Process	Burst Time	Arrival time	Waiting time	Turn around Time
P1	1	0	0	1
P2	10	1	12	22
P3	8	2	11	19
P4	4	5	7	11

Average Waiting Time= 7.50

Average Turnaround Time=13.25

Correct output by program

Test cases: time quantum=2

Process	Burst time	Arrival time
P1	10	0
P2	20	0
P3	30	0
P4	40	0
P5	50	0

Output:

```
Enter Burst Time:50

Enter TIME QUANTUM:2

Process execution order:  A  B  C  D  E  A  B  C  D  E  A  B  C  D  E  A  B  C  D  E  A  B  C  D  E  B  C  D  E  B  C  D
  E  B  C  D  E  B  C  D  E  B  C  D  E  C  D  E  C  D  E  C  D  E  C  D  E  C  D  E  D  E  D  E  D  E  D  E  D  E  E  E
  E  E  E

PROCESS     ARRIVAL TIME        BURST TIME      WAITING TIME    TURNAROUND TIME
A               0               10              32              42
B               0               20              64              84
C               0               30              86              116
D               0               40              98              138
E               0               50              100             150
Average waiting time:76.000000
Average Turnaround time:106.000000
```

Github Revision: I made 5 times commit on github.