

Shivam Chhabra

11609367 B47

EMAIL ID: shivam2614@gmail.com

Link :- <https://github.com/Shivam-99/OS-K1618-B47>

Question->

Ques. 16. Design a scheduler that can schedule the processes arriving system at periodical intervals. Every process is assigned with a fixed time slice t milliseconds. 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. Compute the total time for which processes were in the queue waiting for the processor. Take the input for CPU burst, arrival time and time quantum from the user.

EXPLANATION:

Round Robin is a technique used by schedulers for process scheduling and network scheduling. This method has a simple method of assigning equal amount of time to every process to ensure execution of each process. This eliminates the problem of starvation. It is a time sharing concept of OS.

In this scheduling technique, a time quantum is specified. Each process is allotted time equal to the time quantum to execute. When the time is over, regardless of whether the process has completed execution or not, context switching takes place removing the process from execution and allowing the next process to start execution. This process is continued till the last process has been given the time to execute and then the first process is again given the time assuming it hasn't terminated.

Round robin is a pre-emptive technique as it forces the process out of the CPU once its allotted time has expired.

PSUEDOCODE/PROCEDURE OF ALGORITHM:

1. Take input of CPU running time and Time Quantum from the user
2. Take input of each process from the user and store them in an Array of Structure.
3. Apply Selection sort to arrange the processes in the array in ascending order of their arrival times.
4. Initialise an integer representing the current time of the CPU to 0 and loop till the value of the integer becomes equal to the arrival time of the first process.
5. Start a while loop from the current value of CPU time to the maximum CPU running time.
6. Add another loop in the previous loop's body which finds all the processes arriving at the same time as the current CPU running time and adds them to the link list.
7. After the last loop, for each node in the list beginning from the previous node, subtract the time quantum or burst time (whichever is smaller) from the burst time (as it is the time that the process has spent executing).
8. Increment the value of the CPU running time by the value that the last process has spent executing.
9. Delete the node if it has completed execution i.e. has remaining burst time=0 and arrange the nodes in the list to the proper formation.
10. Increment the value of the CPU running time by 1 if no process execution took place in this iteration of the loop.
11. Update the value of the pointer to the next node after receiving processes in the next iteration to the next node in the list that is to be executed.

COMPLEXITY OF THE ALGORITHM:

```
#include<stdio.h>

#include<conio.h>

#include<stdlib.h>

int TimeQuantum;

struct process
{
    int burstTime,arrivalTime,processNumber,totalBurst; //O(1)
}processQueue[30]; //O(1)

struct processList
{
    int burstTime,arrivalTime,processNumber,totalBurst; //O(1)
    struct processList *next; //O(1)
}*head=NULL; //O(1)

int n; //O(1)

void getProcesses();

void AddToList(int);

void delNode(struct processList *);

int main()
{
    int i=0,j=0,k,e=0,AllotedTime=0,TimeQuantum,t; //O(1)
    printf("Enter CPU running time:"); //O(1)
    scanf("%d",&k); //O(1)
    printf("Enter the time Quantum"); //O(1)
    scanf("%d",&TimeQuantum); //O(1)
    getProcesses(); //O(1)
    int min_index; //O(1)
    for (i = 0; i < n-1; i++)
    {
        min_index = i; //O(n)
```

```

for (j = i+1; j < n; j++)
    if (processQueue[j].arrivalTime < processQueue[min_index].arrivalTime)
        min_index = j;                                //O(n*n)
t=processQueue[min_index].arrivalTime;                  //O(n)
processQueue[min_index].arrivalTime=processQueue[i].arrivalTime; //O(n)
processQueue[i].arrivalTime=t;                          //O(n)
t=processQueue[min_index].burstTime;                    //O(n)
processQueue[min_index].burstTime=processQueue[i].burstTime; //O(n)
processQueue[i].burstTime=t;                            //O(n)
t=processQueue[min_index].processNumber;                //O(n)
processQueue[min_index].processNumber=processQueue[i].processNumber; //O(n)
processQueue[i].processNumber=t;                        //O(n)
t=processQueue[min_index].totalBurst;                    //O(n)
processQueue[min_index].totalBurst=processQueue[i].totalBurst; //O(n)
processQueue[i].totalBurst=t;                          //O(n)
}
i=j=0;                                                  //O(1)
while(i<processQueue[0].arrivalTime)
{
    i++;                                                //O(n)
}
struct processList *ptr=head,*save=NULL;              //O(1)
while(i<k)
{
    while(processQueue[j].arrivalTime<=i&& j<n)
    {
        AddToList(j);                                //O(n*n)
        j++;                                          //O(n*n)
        if(ptr==NULL)
        {
            ptr=head;                                //O(n*n)

```

```
    }  
}  
  
if(save!=NULL)  
{  
    if(save->next!=NULL) //O(n)  
    {  
        ptr=save->next; //O(n)  
    }  
}  
  
if(ptr!=NULL)  
{  
    if(ptr->burstTime<=TimeQuantum)  
    {  
        i+=ptr->burstTime; //O(n)  
        ptr->burstTime=0; //O(n)  
        printf("Process %d executed at %d with Waiting Time:%d and Turn Around  
Time:%d\n",ptr->processNumber,i,i-ptr->totalBurst-ptr->arrivalTime,i-ptr->arrivalTime);  
        save=ptr; //O(n)  
        delNode(ptr); //O(n)  
        ptr=ptr->next; //O(n)  
        e++; //O(n)  
    }  
else  
{  
    i+=TimeQuantum; //O(n)  
    ptr->burstTime-=TimeQuantum; //O(n)  
    save=ptr; //O(n)  
    ptr=ptr->next; //O(n)  
}  
  
if(ptr==NULL)  
{
```

```

        ptr=head;                                //O(n)
    }
}
else
    i++;                                          //O(n)
if(e==n)
{
    break;
}
struct processList *pt=head;                    //O(n)
while(pt!=NULL)
{
    pt=pt->next;                                //O(n)
}
}
}
void getProcesses()
{
    int i;                                       //O(1)
    printf("Enter the number of processes:");   //O(1)
    scanf("%d",&n);                             //O(1)
    for(i=0;i<n;i++)
    {
        printf("Enter the Arrival Time of Process %d:",i+1); //O(n)
        scanf("%d",&processQueue[i].arrivalTime); //O(n)
        printf("Enter the Burst Time of Process %d:",i+1); //O(n)
        scanf("%d",&processQueue[i].burstTime); //O(n)
        processQueue[i].totalBurst=processQueue[i].burstTime; //O(n)
        processQueue[i].processNumber=i+1;      //O(n)
    }
}
}

```

```
void AddToList(int j)
{
    struct processList *ptr,*pt; //O(1)

    ptr = (struct processList*)malloc(sizeof(struct processList)); //O(1)

    ptr->arrivalTime=processQueue[j].arrivalTime; //O(1)

    ptr->burstTime=processQueue[j].burstTime; //O(1)

    ptr->processNumber=processQueue[j].processNumber; //O(1)

    ptr->totalBurst=processQueue[j].totalBurst; //O(1)

    ptr->next=NULL; //O(1)

    if(head==NULL)
    {
        head=ptr; //O(1)
    }
    else
    {
        pt=head; //O(1)

        while(pt->next!=NULL)
        {
            pt=pt->next; //O(n)
        }

        pt->next=ptr; //O(1)
    }
}

void delNode(struct processList *ptr)
{
    if(head==ptr)
    {
        head=head->next; //O(1)
    }
    else
    {
```



```
struct processList *pt=head; //O(1)
while(pt->next!=ptr)
{
    pt=pt->next; //O(n)
}
pt->next=ptr->next; //O(1)
}
}
```

Overall Complexity of Algorithm: $O(n*n)$

ADDITIONAL CONCEPTS:

This program uses the concept of linked lists. The processes which were first entered into an array were later shifted to a link list as the program reached the time of arrival of each individual process.

This program also uses the algorithm of selection sort, a sorting technique which has been used to sort the processes in ascending order of their Arrival times to make it easy to read from the array and then later on write in the linked list.

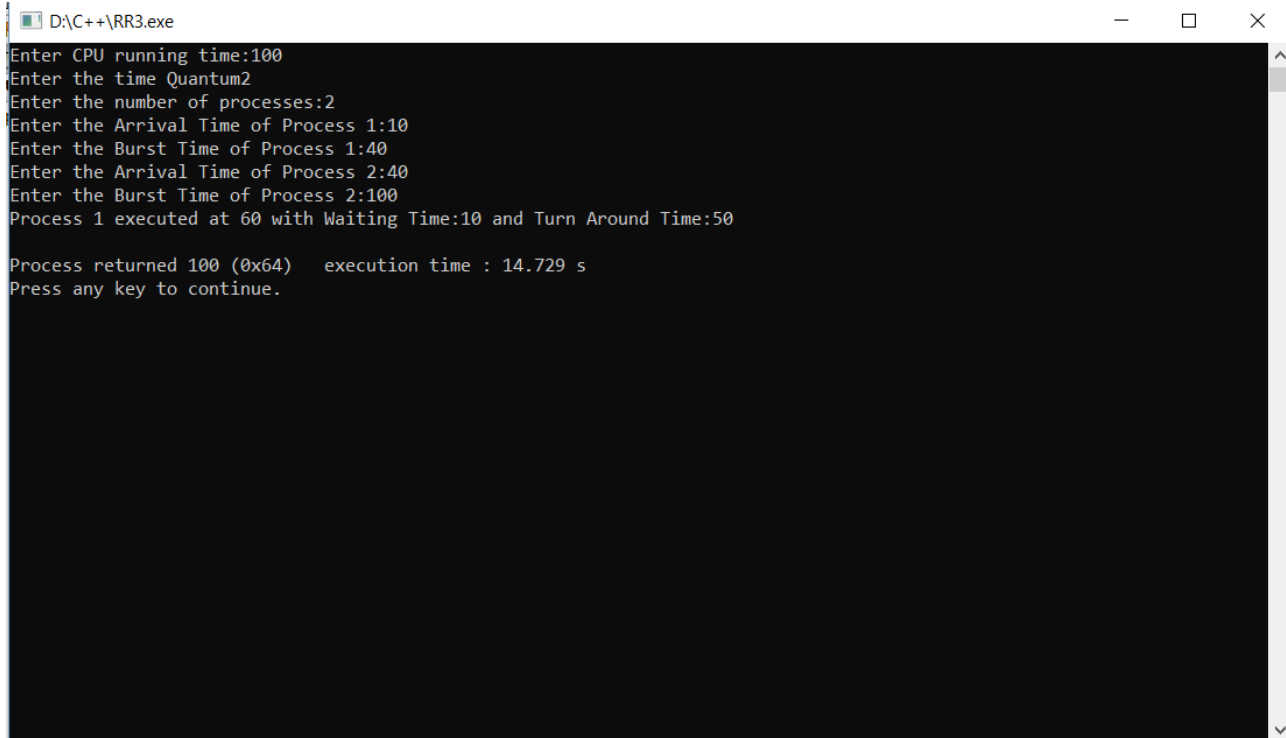
CONSTRAINTS:

The processQueue array has a limit to execute only 30 processes at once so the limit of the variable becomes $0 \leq n \leq 30$.

The CPU time is being entered by the user and so are the burst time and arrival time so their constraint is the limit that an integer holds in C.

TEST CASES :

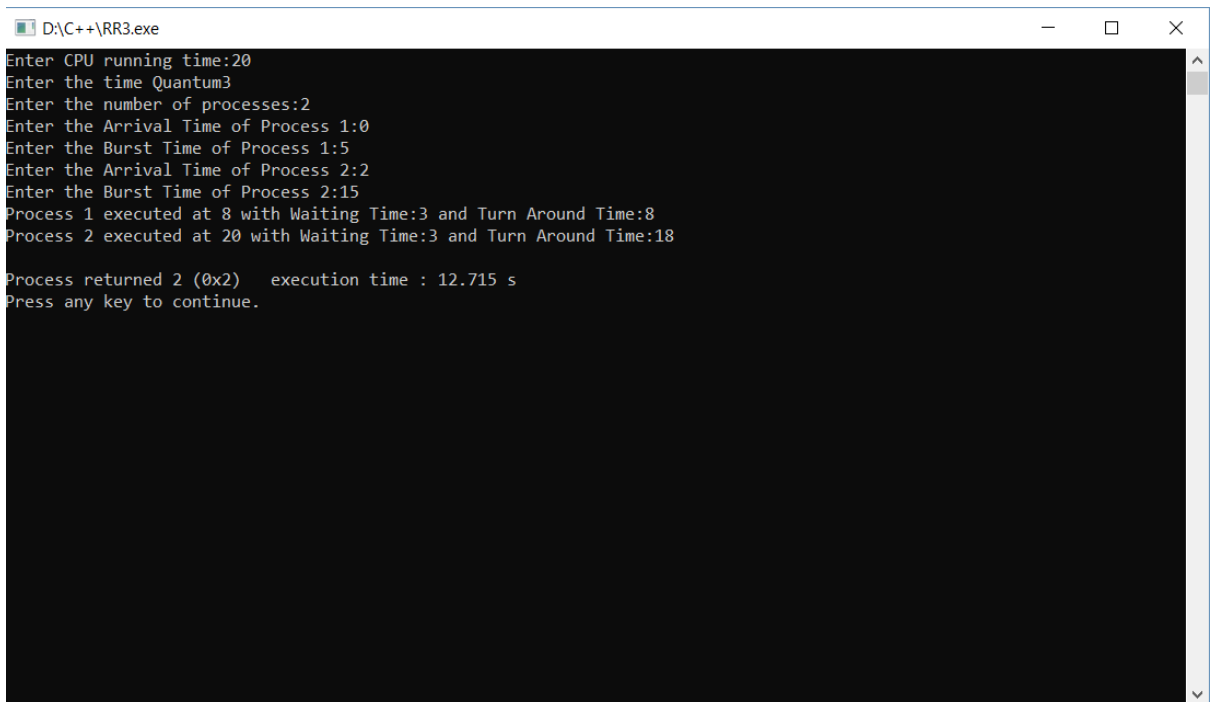
- 1) Boundary Condition: If total time taken by all the processes exceeds the CPU time, the processes that can be executed by then executes and then the execution stops.



```
D:\C++\RR3.exe
Enter CPU running time:100
Enter the time Quantum2
Enter the number of processes:2
Enter the Arrival Time of Process 1:10
Enter the Burst Time of Process 1:40
Enter the Arrival Time of Process 2:40
Enter the Burst Time of Process 2:100
Process 1 executed at 60 with Waiting Time:10 and Turn Around Time:50

Process returned 100 (0x64)   execution time : 14.729 s
Press any key to continue.
```

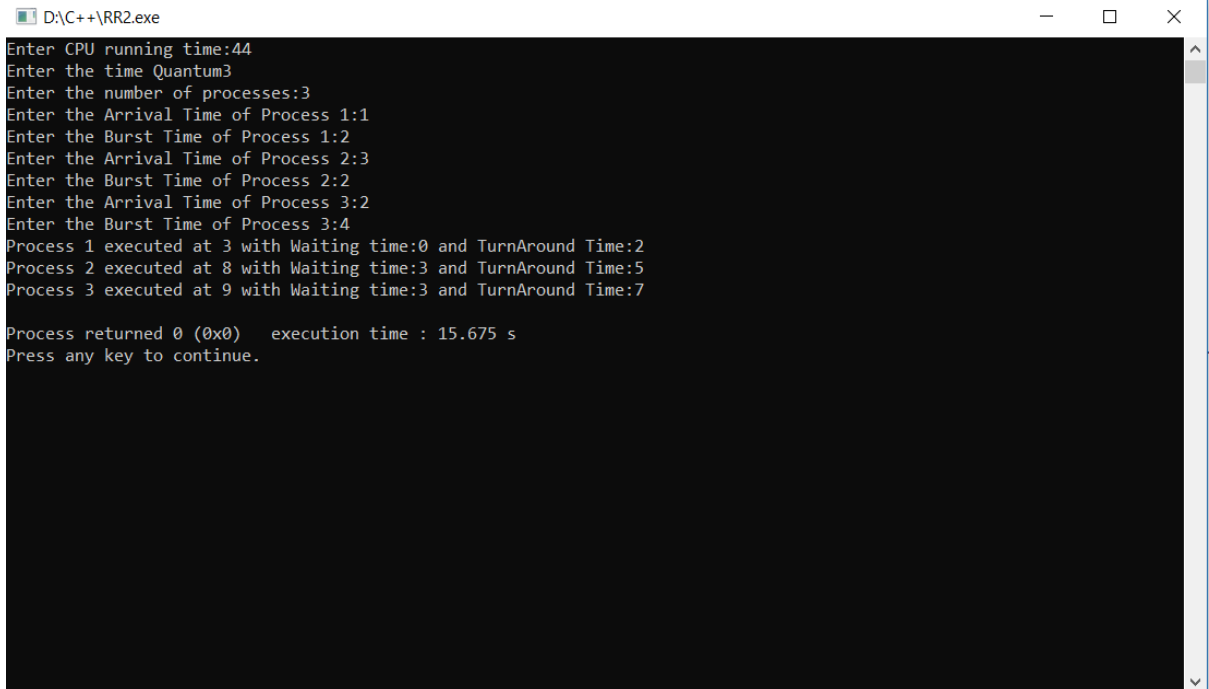
- 2) Boundary Condition : If Total time taken by every process= Available CPU time:



```
D:\C++\RR3.exe
Enter CPU running time:20
Enter the time Quantum3
Enter the number of processes:2
Enter the Arrival Time of Process 1:0
Enter the Burst Time of Process 1:5
Enter the Arrival Time of Process 2:2
Enter the Burst Time of Process 2:15
Process 1 executed at 8 with Waiting Time:3 and Turn Around Time:8
Process 2 executed at 20 with Waiting Time:3 and Turn Around Time:18

Process returned 2 (0x2)   execution time : 12.715 s
Press any key to continue.
```

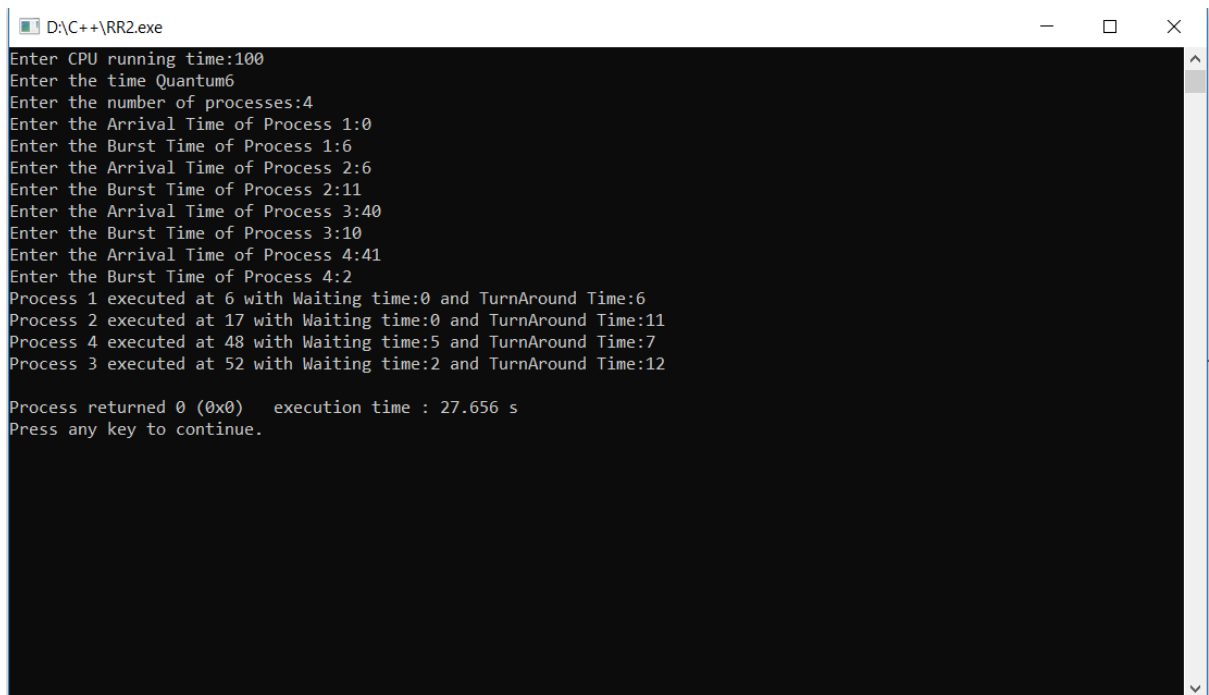
- 3) Normal case: CPU time is more than what is needed by all the processes combined. Round robin is applied and each process executes returning waiting time and turnaround time.



```
D:\C++\RR2.exe
Enter CPU running time:44
Enter the time Quantum3
Enter the number of processes:3
Enter the Arrival Time of Process 1:1
Enter the Burst Time of Process 1:2
Enter the Arrival Time of Process 2:3
Enter the Burst Time of Process 2:2
Enter the Arrival Time of Process 3:2
Enter the Burst Time of Process 3:4
Process 1 executed at 3 with Waiting time:0 and TurnAround Time:2
Process 2 executed at 8 with Waiting time:3 and TurnAround Time:5
Process 3 executed at 9 with Waiting time:3 and TurnAround Time:7

Process returned 0 (0x0)   execution time : 15.675 s
Press any key to continue.
```

- 4) Checks if the program runs into errors when the CPU is left in IDLE state.



```
D:\C++\RR2.exe
Enter CPU running time:100
Enter the time Quantum6
Enter the number of processes:4
Enter the Arrival Time of Process 1:0
Enter the Burst Time of Process 1:6
Enter the Arrival Time of Process 2:6
Enter the Burst Time of Process 2:11
Enter the Arrival Time of Process 3:40
Enter the Burst Time of Process 3:10
Enter the Arrival Time of Process 4:41
Enter the Burst Time of Process 4:2
Process 1 executed at 6 with Waiting time:0 and TurnAround Time:6
Process 2 executed at 17 with Waiting time:0 and TurnAround Time:11
Process 4 executed at 48 with Waiting time:5 and TurnAround Time:7
Process 3 executed at 52 with Waiting time:2 and TurnAround Time:12

Process returned 0 (0x0)   execution time : 27.656 s
Press any key to continue.
```