ADVANCING AND INNOVATING THE ROUND ROBIN ALGORITHM

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

OPERATING SYSTEMS

It is certified that this is a bona fide project of ‘Scheduling Algorithm’ is done by TEAM ‘Umar ’of Computer Science Engineering branch during the year 2018- 19 at VIT, Vellore-632014 under the guidance of

Professor SANTHI H

Faculty Signature

Date: Prof: SANTHI H

TEAM MEMBERS:
MOHD UMAR 18BCE0196
KEERTHI VASAN 18BCE2133
KAILASHNATH K 16BEC0558
HARISHRAJ 18BCE2381
VISHAL 18BCE2360 **…………………………………………………**

ACKNOWLEDGEMENT

Dedicating this project of ‘Scheduling Algorithm’ to the Almighty whose abundant grace and poise enabled its successful completion, I would like to express my profound gratitude to all the people who have inspired and motivated me to undertake this project.

Then I would like to thank ‘Prof. Santhi H’ who gave me such a platform for studying and doing research on this topic in ‘Vellore Institute of Technology, Vellore’. I would like to sincerely express my gratitude to my mam for providing an opportunity to undertake this project, whose precious guidance in doubts faced during the implementation of this project, Moreover, showed us the right path to modify this project and give it a good shape in coming future.

Finally, A heartfelt gratitude to my Parents who gave moral support and encouragement to accomplish the given task.

ABSTRACT

This project deals with the simulation of CPU scheduling algorithms with C. The following algorithms are simulated:

1.First Come First Serve (FCFS)

2.Shortest Job First

3.SRTF Algorithm

4.Round Robin

5.Our innovative algorithm

The metrics such as finishing time, waiting time, total time taken for the processes to complete, number of rounds, etc are calculated.

SOFTWARE ARCHITECTURE (C CODE) :

There are five files for the five scheduling algorithms. They are made into each functions. This function is imported to main function which has the menu. The menu has the choice of which of the algorithm it want to get executed.

In each function the code for each algorithm is done. The code consists arrival time and burst time for each process. The output has the waiting time and turnaround time for each process.

At the beginning any number of processes can be taken in account. The output of each process with all the algorithms can be found.

Algorithm

1. First Come First Serve (FCFS)

   - ★ First-Come-First-Served algorithm is the simplest scheduling algorithm is the simplest scheduling algorithm. Processes are dispatched according to their arrival time on the ready queue. Being a non preemptive discipline, once a process has a CPU, it runs to completion. The FCFS scheduling is fair in the formal sense or human sense of fairness but it is unfair in the sense that long jobs make short jobs wait and unimportant jobs make important jobs wait.
   - ★ FCFS is more predictable than most of other schemes since it offers time. FCFS scheme is not useful in scheduling interactive users because it cannot guarantee good response time. The code for FCFS scheduling is simple to write and understand. One of the major drawback of this scheme is that the average time is often quite long.
   - ★ The First-Come-First-Served algorithm is rarely used asa master scheme in modern operating systems but it is often embedded within other schemes.

2. Shortest Job First

   - ★ Shortest-Job-First (SJF) is a non-preemptive discipline in which waiting job (or process) with the smallest estimated run -time-to-completion is run next. In other words, when CPU is available, it is assigned to the process that has smallest next CPU burst.

- ★ The SJF scheduling is especially appropriate for batch jobs for which the run times are known in advance. Since the SJF scheduling algorithm gives the minimum average time for a given set of processes, it is probably optimal.
- ★ The SJF algorithm favors short jobs (or processors) at the expense of longer ones.
- ★ The obvious problem with SJF scheme is that it requires precise knowledge of how long a job or process will run, and this information is not usually available.
- ★ The best SJF algorithm can do is to rely on user estimates of run times.

3. SRTF Algorithm

- ★ The SRT is the preemptive counterpart of SJF and useful in time-sharing environment.
- ★ In SRT scheduling, the process with the smallest estimated run-time to completion is run next, including new arrivals.
- ★ In SJF scheme, once a job begin executing, it run to completion.
- ★ In SJF scheme, a running process may be preempted by a new arrival process with shortest estimated run-time.
- ★ The algorithm SRT has higher overhead than its counterpart SJF.

- ★ The SRT must keep track of the elapsed time of the running process and must handle occasional preemptions.

- ★ In this scheme, arrival of small processes will run almost immediately. However, longer jobs have even longer mean waiting time.

4. Round Robin Algorithm

- ★ One of the oldest, simplest, fairest and most widely used algorithm is round robin (RR).
- ★ In the round robin scheduling, processes are dispatched in aFIFO manner but are given a limited amount of CPU time called a time-slice or a quantum.
- ★ If a process does not complete before its CPU-time expires, the CPU is preempted and given to the next process waiting in a queue. The preempted process is then placed at the back of the ready list.
- ★ Round Robin Scheduling is preemptive (at the end of time-slice) therefore it is effective in time-sharing environmentsin which the system needs to guarantee reasonable response times for interactive users.
- ★ The only interesting issue with round robin scheme is the length of the quantum. Setting the quantum too short causes too many context switches and lower the CPU efficiency. On the other hand, setting the quantum too long may cause poor response time and approximates FCFS.

- ★ In any event, the average waiting time under round robin scheduling is often quite long.

5. Our innovative algorithm

- ★ The proposed algorithm focuses on the improvement on CPU scheduling algorithm. The algorithm reduces the waiting time and turnaround time drastically compared to the other Scheduling algorithm and simple Round Robin scheduling algorithm.
- ★ This proposed algorithm works in a similar way as but with some modification.It executes the shortest job having minimum burst time first instead of FCFS simple Round robin algorithm and it also uses Smart time quantum instead ofstatic time quantum. Instead of giving static time quantum in the CPU scheduling algorithms, our algorithm calculates the Smart time quantum itself according to the burst time of all processes.
- ★ The proposed algorithm eliminates the discrepancies of implementing simple round robin architecture.
- ★ In the first stage of the innovative algorithm CPU scheduling algorithms all the processes are arranged in the increasing order of CPU burst time. It means it automatically assign the priority to the processes.
- ★ Process having low burst time has high priority to the process have high burst time.

- ★ Then in the second stage the algorithm calculates the mean of the CPU burst time of all the processes. After calculating the mean, it will set the time quantum dynamically i.e. (averageof mean and highest burst time)/2.
- ★ Then in the last stage algorithm pick the first process from the ready queue and allocate the CPU to the process for a time interval of up to 1 Smart time quantum. If the remaining burst time of the current running process is less than 1 Smart time quantum then algorithm again allocate the CPU to the Current process till it execution.
- ★ After execution it will remove the terminated process from the ready queue and again go to the stage 3.

LITERATURE SURVEY :

[1] Abstract- Multiprogramming is an important aspect of operating systems (OS); it requires several processes to be kept simultaneously in the memory, the aim of which is maximum CPU utilization.

Among other CPU scheduling algorithms, like the First Come First Serve (FCFS), Shortest Job First (SJF) and Priority Scheduling (PS); Round Robin is considered the most widely used scheduling algorithm in time sharing and real time OS for allocating the CPU to the processes in the memory in order to achieve the aim mentioned above. This paper proposed a more improvement in the Round Robin CPU scheduling algorithm.

By experimental analysis, this proposed algorithm performs better than the simple Round Robin and the improved Round Robin CPU scheduling algorithms in terms of minimizing average waiting time, average turnaround time and number of context switches.

Keywords: operating system, multiprogramming, CPU utilization, CPU scheduling algorithm, Round Robin.

[2] Abstract - The main objective of this paper is to develop a new approach for round robin scheduling which help to improve the CPU efficiency in real time and time sharing operating system.

There are many algorithms available for CPU scheduling. But we cannot implemented in real time operating system because of high context switch rates, large waiting time, large response

time, large turn around time and less throughput. The proposed algorithm improves all the drawback of simple round robin architecture. The author have also given comparative analysis of proposed with simple round robin scheduling algorithm.

Therefore, the author strongly feel that the proposed architecture solves all the problem encountered in simple round robin architecture by decreasing the performance parameters to desirable extent and thereby increasing the system throughput.

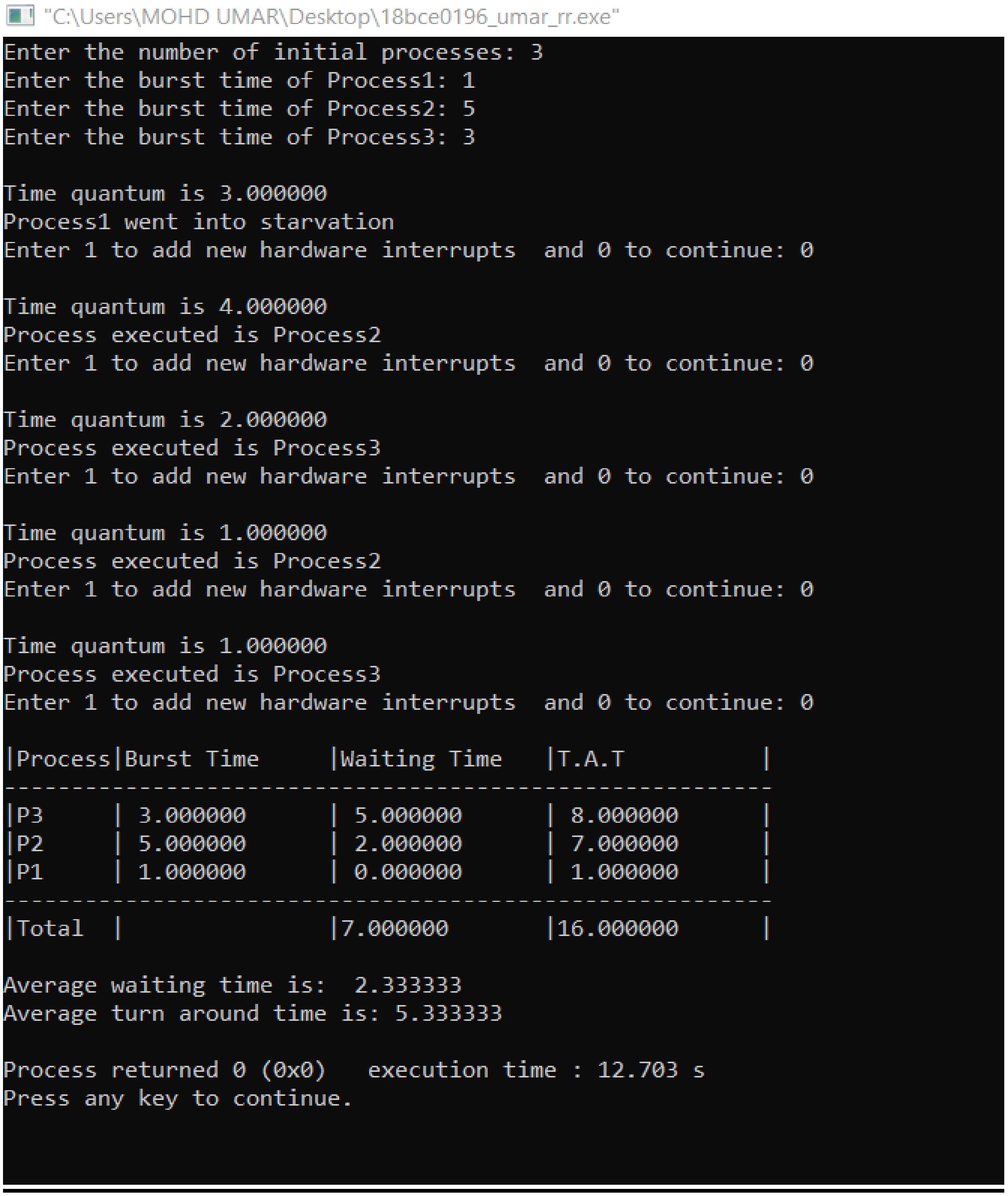
Keywords: CPU scheduling,RR schedulingalgorithm,Turnaround time,Waiting time,Response time,Context switch rate, Gantt chart.

CODE

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

int cnt=0;// keeps the count of array level
int count_act;

struct pro
{
    float burst;
    float burst_cpy;
    int pro_no;
    float waiting_time;
};

int add(struct pro *point_arr,int count)
{
    printf("\nEnter 1 to add new hardware interrupts  and 0 to continue: ");
    int choice, new_p,n,higher=0,lower =0;// new_p -> number of new process
    float new_higher[3], new_lower[3];
    float new_pro;
    int choice_prior;
    scanf("%d",&choice);
    switch(choice) // default choice is set to continue
    {
    case 0:
        new_p=0;
        break;
    case 1:
        printf("Enter the number of process to add: ");
        scanf("%d",&new_p);
        int n;
        for(n=0;n<new_p;n++)
    {
        printf("Enter the burst time of process%d: ",n+1);
        scanf("%f",&new_pro);
        printf("Enter the priority: "); // zero for low priority and 1 for high
        scanf("%d",&choice_prior);
        switch(choice_prior)// default choice is set to lower priority
        {
        case 0:
            new_lower[lower]= new_pro;
            lower++;
            break;
        case 1:
            new_higher[higher]= new_pro;
            higher++;
            break;
        default:
            new_lower[lower]= new_pro;
            lower++;
            break;
        }
    }
    break;
default:
    new_p=0;
    break;
}
for (n=0;n<higher;n++)
{
    ((point_arr+40*cnt+n)->burst)= new_higher[n];
    ((point_arr+40*cnt+n)->burst_cpy)=new_higher[n];
    ((point_arr+40*cnt+n)->pro_no)= n+count;
    ((point_arr+40*cnt+n)->waiting_time)=0;
}
for(n=0;n<count_act-1;n++)
```

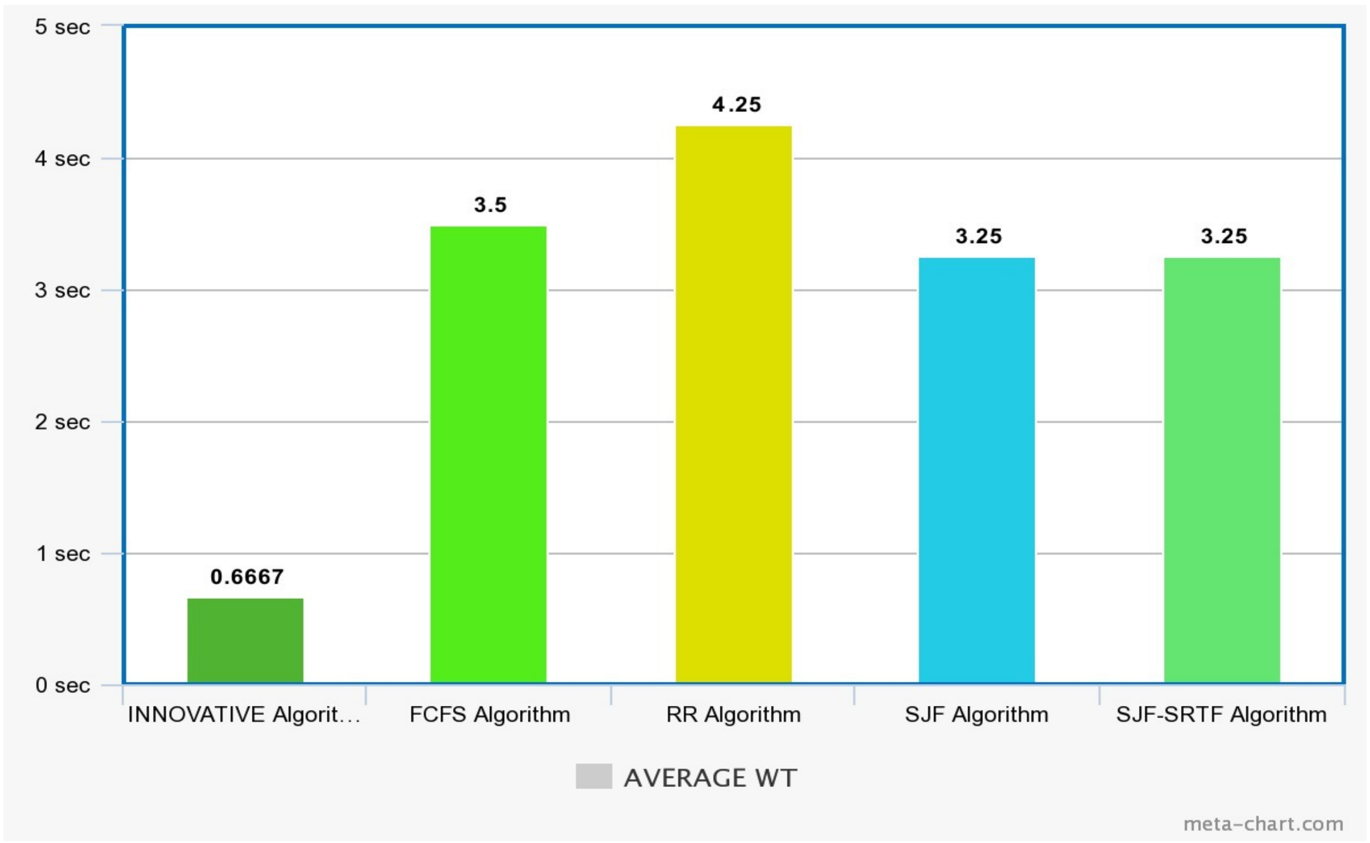
```
    {
        ((point_arr+40*cnt+higher+n)->burst)=((point_arr+40*(cnt-1)+n+1)-
>burst);
        ((point_arr+40*cnt+higher+n)->burst_cpy)=((point_arr+40*(cnt-1)+n+1)-
>burst_cpy);
        ((point_arr+40*cnt+higher+n)->pro_no)= ((point_arr+40*(cnt-1)+n+1)-
>pro_no);
        ((point_arr+40*cnt+higher+n)->waiting_time)=((point_arr+40*(cnt-
1)+n+1)->waiting_time);
    }
    for(n=0;n<=lower;n++)
    {
        ((point_arr+40*cnt+count-1+higher+n)->burst)=new_lower[n];
        ((point_arr+40*cnt+count-1+higher+n)->burst_cpy)=new_lower[n];
        ((point_arr+40*cnt+count-1+higher+n)->pro_no)= count+higher+n;
        ((point_arr+40*cnt+count-1+higher+n)->waiting_time)=0;
    }

    for(n=count_act;n<count;n++)
    {
        ((point_arr+40*cnt+higher+lower+n)->burst)=((point_arr+40*(cnt-1)+n)-
>burst);
        ((point_arr+40*cnt+higher+lower+n)->burst_cpy)=((point_arr+40*(cnt-
1)+n)->burst_cpy);
        ((point_arr+40*cnt+higher+lower+n)->pro_no)=((point_arr+40*(cnt-
1)+n)->pro_no);
        ((point_arr+40*cnt+higher+lower+n)-
>waiting_time)=((point_arr+40*(cnt-1)+n)->waiting_time);
    }
    /*for(n=count_act;n<count;n++)
    {
        ((point_arr+40*cnt+higher+lower+n)->burst)=((point_arr+40*(cnt-
1)+count-count_act+n-1)->burst);
        ((point_arr+40*cnt+higher+lower+n)->burst_cpy)=((point_arr+40*(cnt-
1)+count-count_act+n-1)->burst_cpy);
        ((point_arr+40*cnt+higher+lower+n)->pro_no)=((point_arr+40*(cnt-
1)+count-count_act+n-1)->pro_no);
        ((point_arr+40*cnt+higher+lower+n)-
>waiting_time)=((point_arr+40*(cnt-1)+count-count_act+n-1)->waiting_time);
    }*/
    if (((point_arr+40*(cnt-1))->burst)!=0)
    {
        ((point_arr+40*cnt+higher+lower+count_act-1)->burst)=
((point_arr+40*(cnt-1))->burst);
        ((point_arr+40*cnt+higher+lower+count_act-1)-
>burst_cpy)=((point_arr+40*(cnt-1))->burst_cpy);
        ((point_arr+40*cnt+higher+lower+count_act-1)-
>pro_no)=((point_arr+40*(cnt-1))->pro_no);
        ((point_arr+40*cnt+count_act-1+higher+lower)-
>waiting_time)=((point_arr+40*(cnt-1))->waiting_time);
        count_act = count_act +new_p;
    }
    else
    {
        ((point_arr+40*cnt+higher+lower+count_act-1)->burst)=
((point_arr+40*(cnt-1))->burst);
        ((point_arr+40*cnt+higher+lower+count_act-1)-
>burst_cpy)=((point_arr+40*(cnt-1))->burst_cpy);
        ((point_arr+40*cnt+higher+lower+count_act-1)-
>pro_no)=((point_arr+40*(cnt-1))->pro_no);
        ((point_arr+40*cnt+count_act-1+higher+lower)-
>waiting_time)=((point_arr+40*(cnt-1))->waiting_time);
        count_act = count_act +new_p-1;
    }
    count+=new_p;

    return(count);
}
float tq (struct pro *point_arr, int count)//function to calculate time quantum
{
    int n;
    float tm_q=0;
    for(n=0;n<count;n++)
    {
        tm_q += ((point_arr+40*cnt+n)->burst);
    }
    tm_q = tm_q/count_act;
    return(tm_q);
}

int check(struct pro *point_arr, int count)
{
    int n,p;
    for(n=0;n<count;n++)
    {
        if(((point_arr+40*cnt+n)->burst)==0)
        {
            p=1;
            continue;
        }
        else
        {
            p=0;
            break;
        }
    }
    return(p);
}

int main()
{
    struct pro arr[40][40]; // array containing the process info
    struct pro *point_arr = &arr[0][0]; // pointer to the first element of the array
    int np; // initial number of process
    printf("Enter the number of initial processes: ");
    scanf("%d",&np);
    int n;
    for(n=0;n<np;n++)
    {
        printf("Enter the burst time of Process%d: ",n+1);
        scanf("%f",&arr[0][n].burst);
        arr[0][n].burst_cpy= arr[0][n].burst;
        arr[0][n].pro_no = n;
        arr[0][n].waiting_time = 0;
    }
    float tmq;// time- quantum
    int flag=0;
    int count = np;
    count_act= count;
    while (flag==0)
    {
        tmq = tq(point_arr,count);
        printf("\nTime quantum is %f",tmq);
        if(tmq<=((point_arr+40*cnt)->burst))
        {
            ((point_arr+40*cnt)->burst) = ((point_arr+40*cnt)->burst)-tmq;
            printf("\nProcess executed is Process%d",((point_arr+40*cnt)-
>pro_no)+1);
            //((point_arr+40*cnt)->waitnig_time)=((point_arr+40*cnt)-
>waitnig_time)+tmq;
            for(n=1;n<count;n++)
        {
            //printf("\n%f",((point_arr+40*cnt+n)->burst));
            if(((point_arr+40*cnt+n)->burst)!=0)
            {
                ((point_arr+40*cnt+n)->waiting_time)=(((point_arr+40*cnt+n)-
>waiting_time)+tmq) ;
             //  printf("\n%f",((point_arr+40*cnt+n)->waiting_time));
            }
        }

        }
        else
        {
           ((point_arr+40*cnt)->burst)=0;
           printf("\nProcess%d went into starvation",((point_arr+40*cnt)-
>pro_no)+1);
        }

        cnt++;
        count = add(point_arr,count);
        flag = check(point_arr,count);
    }
    float twt,ttat;//total waitng time, totl turn around time
    //printf("count max = %d",cnt_max);
    printf("\n|Process|Burst Time     |Waiting Time\t|T.A.T\t\t|");
    printf("\n-------------------------------------------------------");
    for(n=0;n<count;n++)
    {
        printf("\n|P%d\t| %f\t| %f\t| %f\t|",(((point_arr+40*cnt+n)-
>pro_no)+1),((point_arr+40*cnt+n)->burst_cpy),((point_arr+40*cnt+n)-
>waiting_time),(((point_arr+40*cnt+n)->waiting_time)+((point_arr+40*cnt+n)-
>burst_cpy)));
        twt+=((point_arr+40*cnt+n)->waiting_time);
        ttat+=((point_arr+40*cnt+n)->waiting_time)+((point_arr+40*cnt+n)-
>burst_cpy);
    }
    printf("\n-------------------------------------------------------");
    printf("\n|Total  |\t\t|%f\t|%f\t|",twt,ttat);
    printf("\n\nAverage waiting time is:  %f",twt/count);
    printf("\nAverage turn around time is: %f",ttat/count);
    printf("\n");
    return 0;
}
```

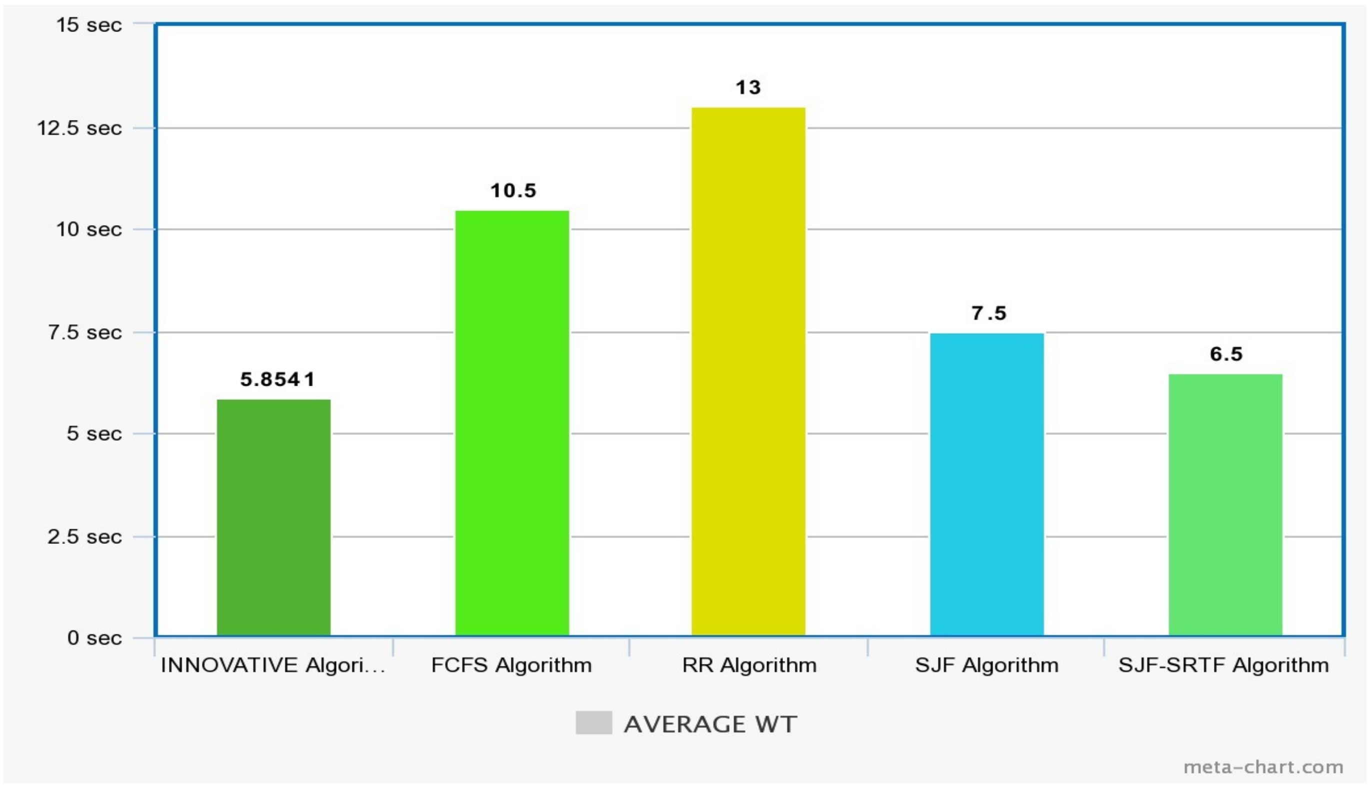
OUTPUT:

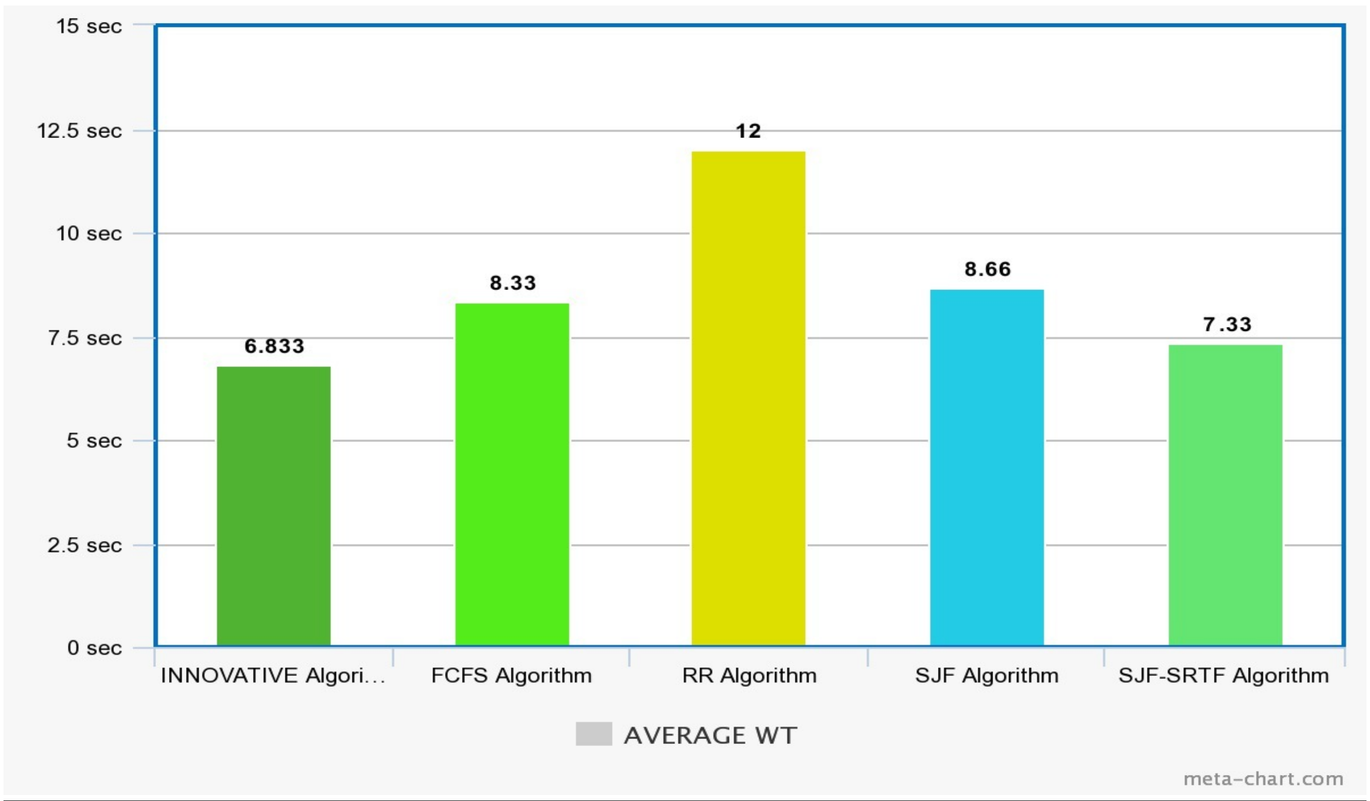

"C:\Users\MOHD UMAR\Desktop\18bce0196_umar_rr.exe"

```
Enter the number of initial processes: 3
Enter the burst time of Process1: 1
Enter the burst time of Process2: 5
Enter the burst time of Process3: 3

Time quantum is 3.000000
Process1 went into starvation
Enter 1 to add new hardware interrupts  and 0 to continue: 0

Time quantum is 4.000000
Process executed is Process2
Enter 1 to add new hardware interrupts  and 0 to continue: 0

Time quantum is 2.000000
Process executed is Process3
Enter 1 to add new hardware interrupts  and 0 to continue: 0

Time quantum is 1.000000
Process executed is Process2
Enter 1 to add new hardware interrupts  and 0 to continue: 0

Time quantum is 1.000000
Process executed is Process3
Enter 1 to add new hardware interrupts  and 0 to continue: 0

|Process|Burst Time     |Waiting Time   |T.A.T          |
---------------------------------------------------------
|P3     | 3.000000      | 5.000000      | 8.000000      |
|P2     | 5.000000      | 2.000000      | 7.000000      |
|P1     | 1.000000      | 0.000000      | 1.000000      |
---------------------------------------------------------
|Total  |               |7.000000       |16.000000      |

Average waiting time is:  2.333333
Average turn around time is: 5.333333

Process returned 0 (0x0)   execution time : 12.703 s
Press any key to continue.
```

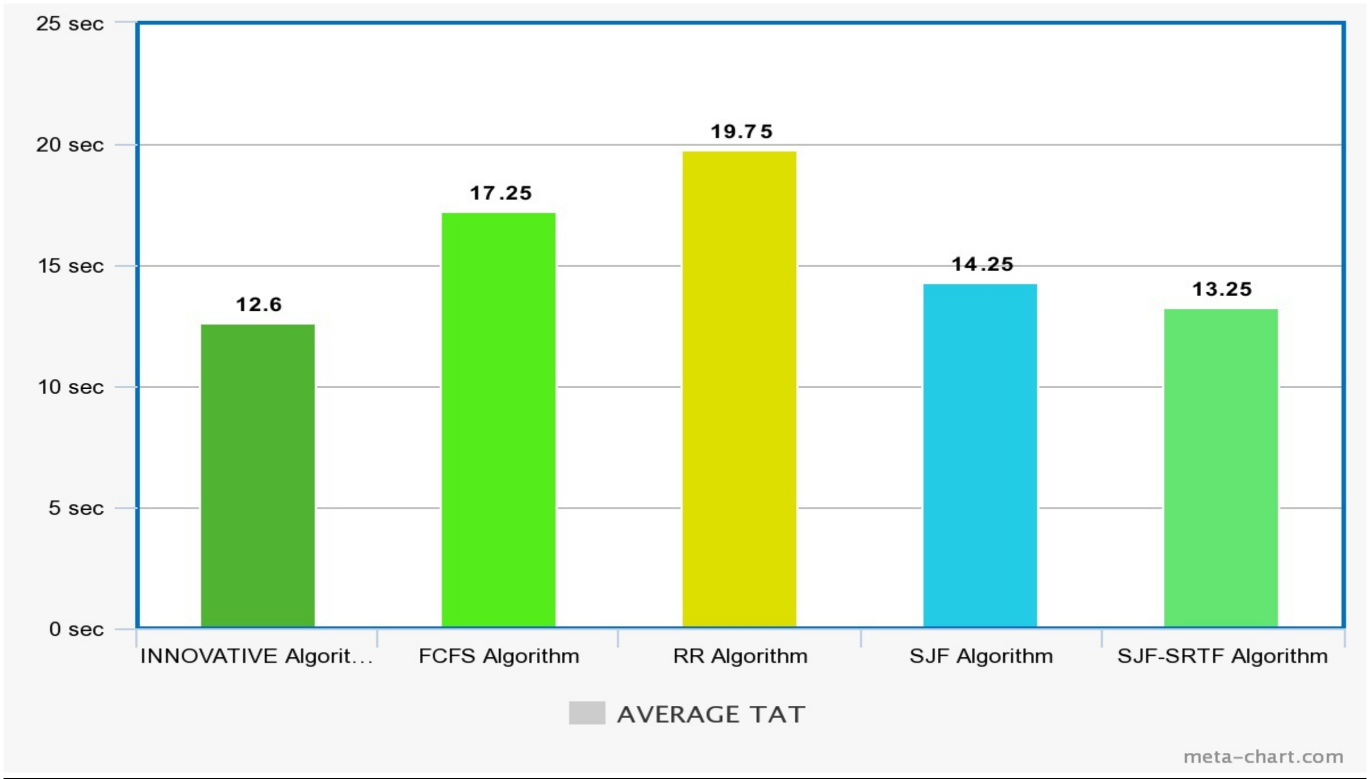
COMPARISON GRAPH BETWEEN AVERAGE WT

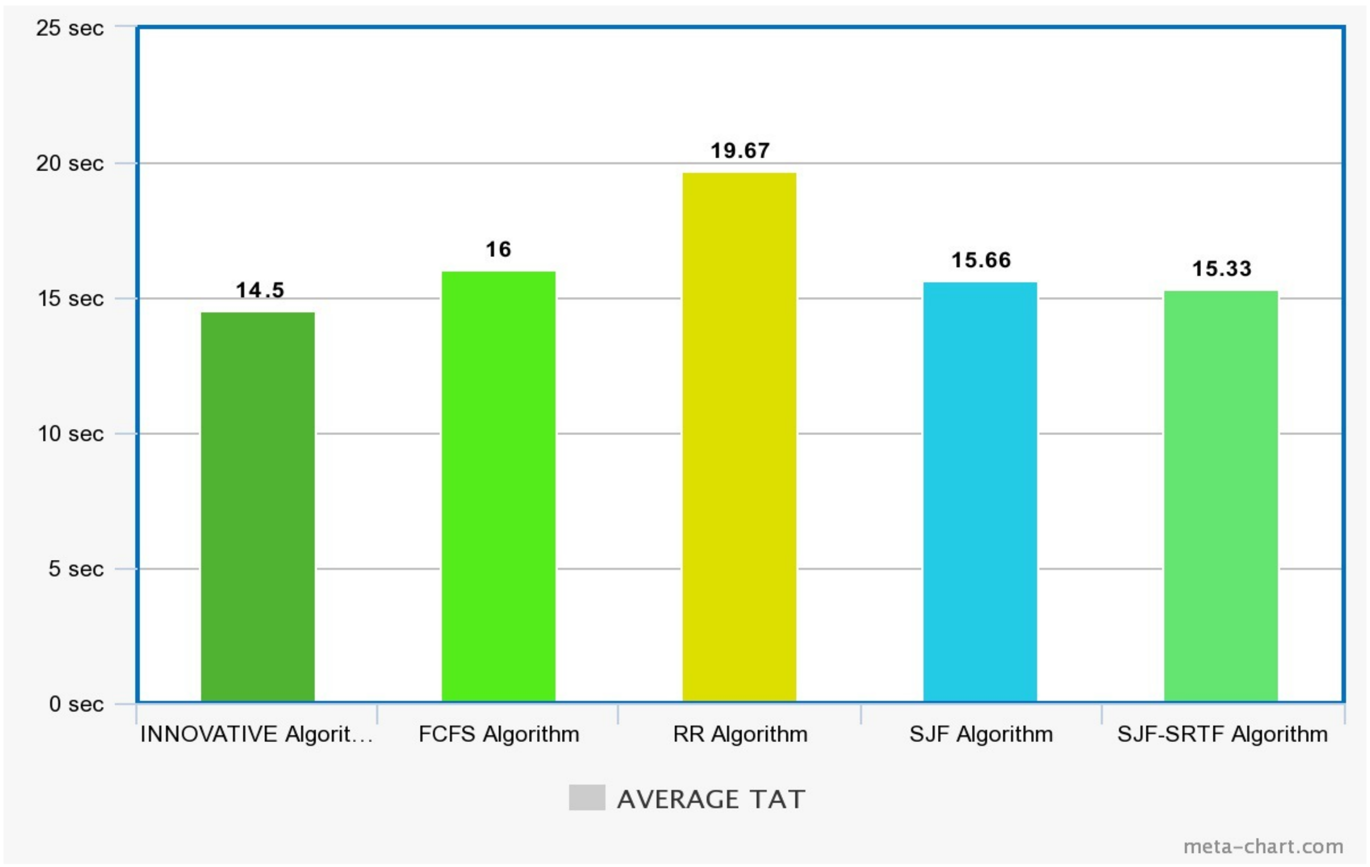


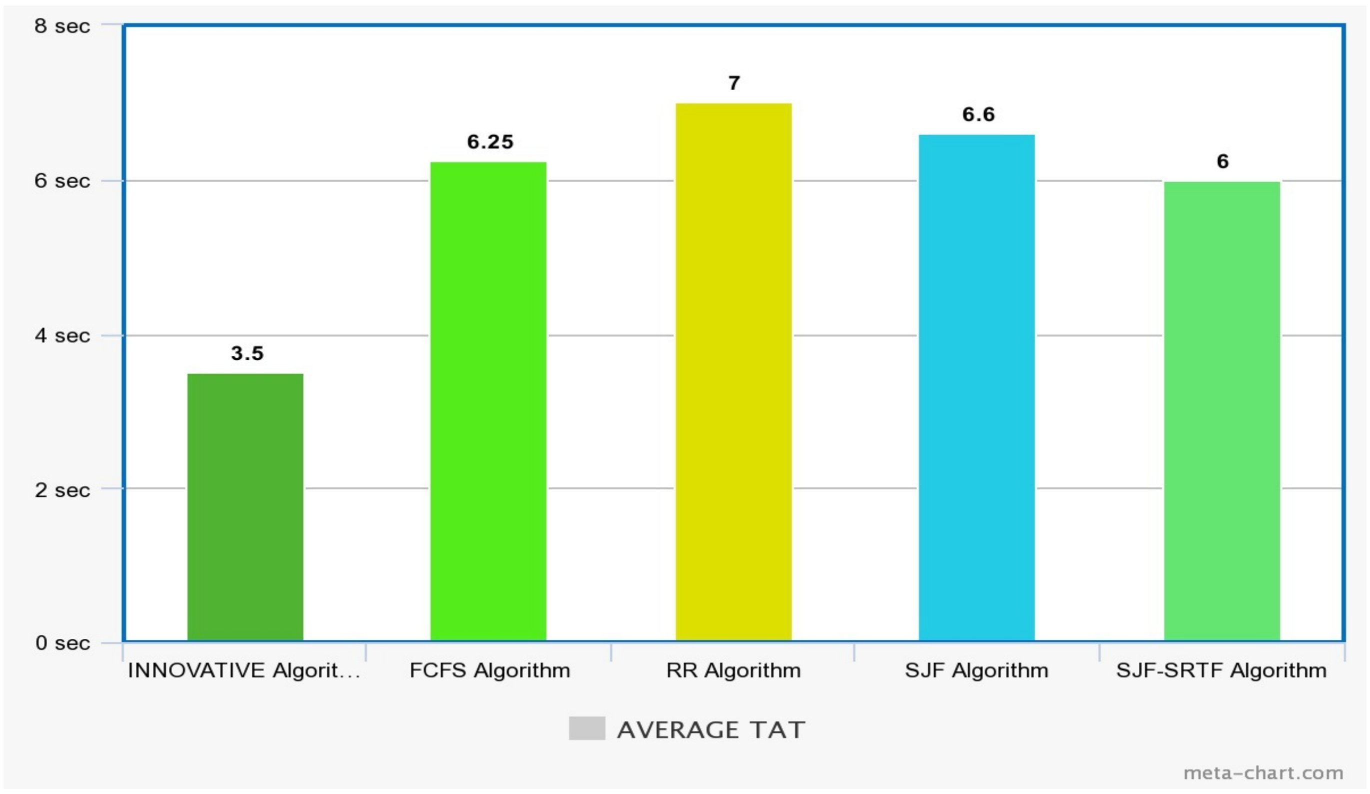




COMPARISON GRAPH BETWEEN AVERAGE TAT




25 sec
20 sec
15 sec
10 sec
5 sec
0 sec
14.5
16
19.67
15.66
15.33
INNOVATIVE Algorit…
FCFS Algorithm
RR Algorithm
SJF Algorithm
SJF-SRTF Algorithm
AVERAGE TAT
meta-chart.com


8 sec
6 sec
4 sec
2 sec
0 sec
3.5
6.25
7
6.6
6
INNOVATIVE Algorit…
FCFS Algorithm
RR Algorithm
SJF Algorithm
SJF-SRTF Algorithm
AVERAGE TAT
meta-chart.com

CONCLUSION

Results have shown that the proposed algorithm gives better results in terms of average waiting time, average turnaround time and number of context switches in all cases of process categories than the simple Round Robin CPU scheduling algorithm.

In all these proposed algorithms time quantum is static due to which in these cases the number of context switches, average waiting time and average turnaround time will be very high and in our proposed algorithm, time quantum is calculated dynamically according to the burst time of all processes and it will find out a smart time quantum for all processes which gives good performance as compared to FCFS, RR, SJF and XX.

REFERENCES

[1]. Abdulraza, Abdulrahim,Salisu Aliyu, Ahmad M Mustapha, Saleh E Abdullahi, “ An Additional Improvement in Round Robin (AAIRR) CPU Scheduling Algorithm,”International Journal of Advanced Research in Computer Science and Software EngineeringVolume 4, Issue 2, February 2014 ISSN: 2277 128X.

[2]. Singh, A., Goyal, P., & Batra, S. (2010). An optimized round robin scheduling algorithm for CPU scheduling. International Journal on Computer Science and Engineering, 2(07), 2383-2385