An Application Based Improved Round Robin CPU Scheduling for Real Time Operating System

BRAC UNIVERSITY

Inspiring Excellencet

Supervised By:

Ms. Suraiya Tairin
Lecturer, Department of Computer Science and Engineering
BRAC University

Submitted By:

Nasif Mahmud	13101161
Sadiya Afrin	13101140
Farzana Rahman	14301140
Md.Monirujjaman	13101136

Submitted On

April 2017

ABSTRACT

CPU scheduling is the primary and very important part of any operating system. CPU scheduling criteria is based on multiprogramming operating system. CPU executes one process at a time and other process is in waiting state to be executed. It prioritizes processes to efficiently execute the user requests and help in choosing the appropriate process for execution. Whatever the main goal of CPU scheduling is to minimize the average waiting time, turnaround time and also the context switching in order to make the best use of CPU. In this state, our main goal is to build such model in which Process with the shortest burst time with a dynamic time quantum calculated by using median formula executed first and so on until the ready queue is not empty. Implementing this idea, we can minimize the average turnaround time, waiting time and also reduce the context switch over traditional RR. To implement this idea, we have developed a simulation software to view the experimental result whether it fulfill our requirement or not. However, we hope that in future we can extend it to a more advanced way.

ACKNOWLEDGEMENT

Before starting to write this paper, we would like to express our utmost gratitude to the Almighty Allah who gives us the Strength with intelligence and determination to build up our thesis. We want to thank our parents for their support. At the same time, we would very grateful to our respected faculties and dear classmates for their constant support and motivation. Most importantly we would like to thank our honorable supervisor Suraiya Tairin miss for her consistent supervision, guidance and encouragement in accomplishing our thesis work and help us to execute our idea in a successful way.

Author's Declaration

This thesis work is submitted to the Department of Computer Science & Engineering, BRAC UNIVERSITY, Dhaka. We hereby, declare that this thesis work is based on the results found by ourselves. The other helping materials related with our work are found by other researchers mentioned by reference. This thesis work neither has been previously submitted for any degree.

SIGNATURE OF THE AUTHORS

Nasif

Nasif Mahmud

ID: 13101161

Sadiya Afrin

Sadiya Afrin

ID: 13101140

Farzana Rahman.

Farzana Rahman

ID: 14301140

Monir

Md.Monirujjaman

ID: 13101136

SIGNATURE OF THE SUPERVISOR

[illegible] 20/4/17

Ms. Suraiya Tairin

Lecturer, Department of Computer Science and Engineering

BRAC University.

[illegible] 24-4-17

Contents

Chapter 1
Introduction

1.1 Motivation

CPU scheduling criteria is one of the most important issue in operating system. There are so many traditional criteria of CPU scheduling such as the shortest job first (SJF), First come first serve(FCFS), Priority scheduling and the most important Round Robin(RR) to minimize the response time, waiting time, turnaround time, number of context switching and maximizing the CPU utilization. With these concepts, CPU utilization is not occurring properly. To improve CPU utilization, we are motivated to reduce the turnaround time, average waiting time and the most importantly context switching issues. We have proposed an efficient technique in process scheduling algorithm by using dynamic time quantum in Round Robin Scheduling Algorithm. In order to simulate the behavior of various CPU scheduling algorithm and to improve Round Robin scheduling algorithm using dynamic time slice concept, we proposed improved CPU scheduling Round Robin algorithm. Our approach is based on the calculation of time quantum in single Round Robin cycle. Taking into the consideration of the arrival time, we implement the algorithm. Experimental analysis shows the better performance of this improved Round Robin algorithm. It minimizes the number of context switching, average waiting time, average turn-around time. Consequently the throughput and CPU utilization is better. So, the proposed algorithm is experimentally proven better than conventional Round Robin Algorithm. The simulation results show that the waiting time and turn-around time have been reduced in the proposed algorithm compared to traditional Round Robin.

1.2 Objective

In this present decade due to technology revolution we have to perform a lot of task simultaneously. As a result CPU overhead is increasing. To save this resource (CPU) we have to utilize CPU properly to enhance its performance. Whenever the term we use as multitasking another term Scheduling is come by default which is the main concept of multitasking. From the very beginning till present there are many CPU Scheduling algorithm are proposed to enhance the performance of CPU such as the First-come First serve, Shortest job first, Priority scheduling and the most popular Round Robin (RR)scheduling algorithm. We have deal with this traditional Round Robin scheduling algorithm. In the traditional RR the time quantum TQ is fixed for all the processes where the main problem occurs. If the TQ is too less than the processes burst time the average waiting time the average turnaround time and the most important context switching will be very high which decrease the performance of CPU. On the other hand if the TQ is high then there will be a chance to cause starvation the response time will be very low which also decrease the performance of CPU. So over these limitation of traditional RR we take the initiative to improve this means to reduce the average waiting time turn-around time and also the context switching to increase the performance of CPU. We have planned to use a dynamic TQ for each individual processes based on their burst time which will be optimal. So our main objective is to improved RR by reducing the average waiting time average turn-around time and the context switching to enhance the performance of CPU .

For the purpose of this analysis and testing the user first specifies each process along with information like arrival time, burst time and then the algorithm will produce the output in an appropriate readability.

Chapter 2
Literature Review

2.1 Overview

2.1.1 CPU Scheduling

CPU scheduling is the basis of multiprogramming operating system. By switching CPU among processes the operating system can make the computer more productive. A multiprogramming operating system allows more than one processes at a time into an executable memory and share the CPU using time multiplexing. A reason for using multiprogramming is that the OS itself is implemented as one or more processes so there must be a way to share the CPU between OS and application processes. Another reason is for performing the I/O operation for normal computation. Since I/O ordinarily require more time than perform CPU instruction, so multiprogramming system allocate the CPU to another process whenever a process is invoked for an I/O operation.

Since there are many more available processes than available CPU, so scheduling refers to the processes to run the available CPU.

2.1.2 CPU I/O-Burst Cycle

The success of CPU scheduling depends on the property of the processes. Processes execution is consisting of a cycle between CPU execution and i/o wait. Processes alternates between these two states.

A process begins with a CPU burst time, followed by an I/O, followed by another CPU burst, then again followed by another I/O and goes on. Eventually the last CPU burst ends with a system request to terminate the execution process.

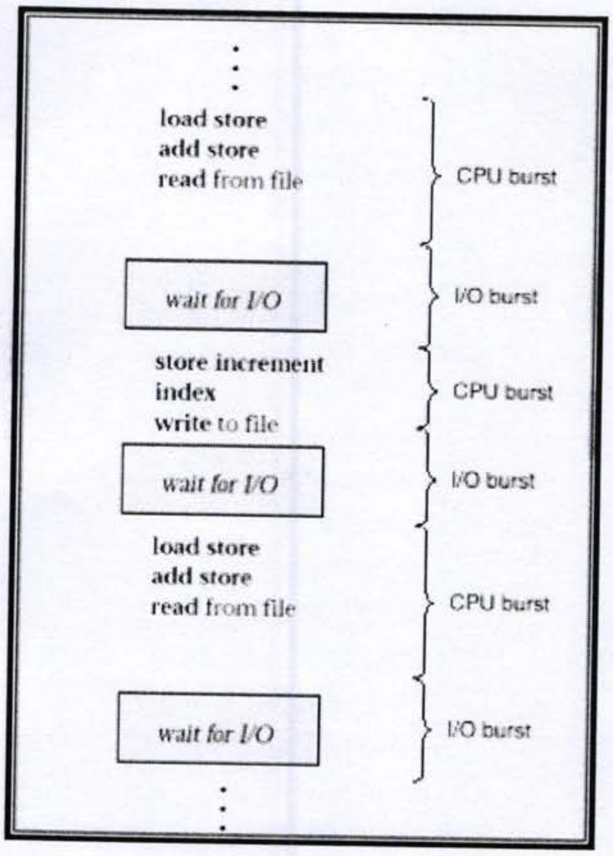


Fig: CPU I/O-Burst Cycle

2.1.3 Process State Diagram

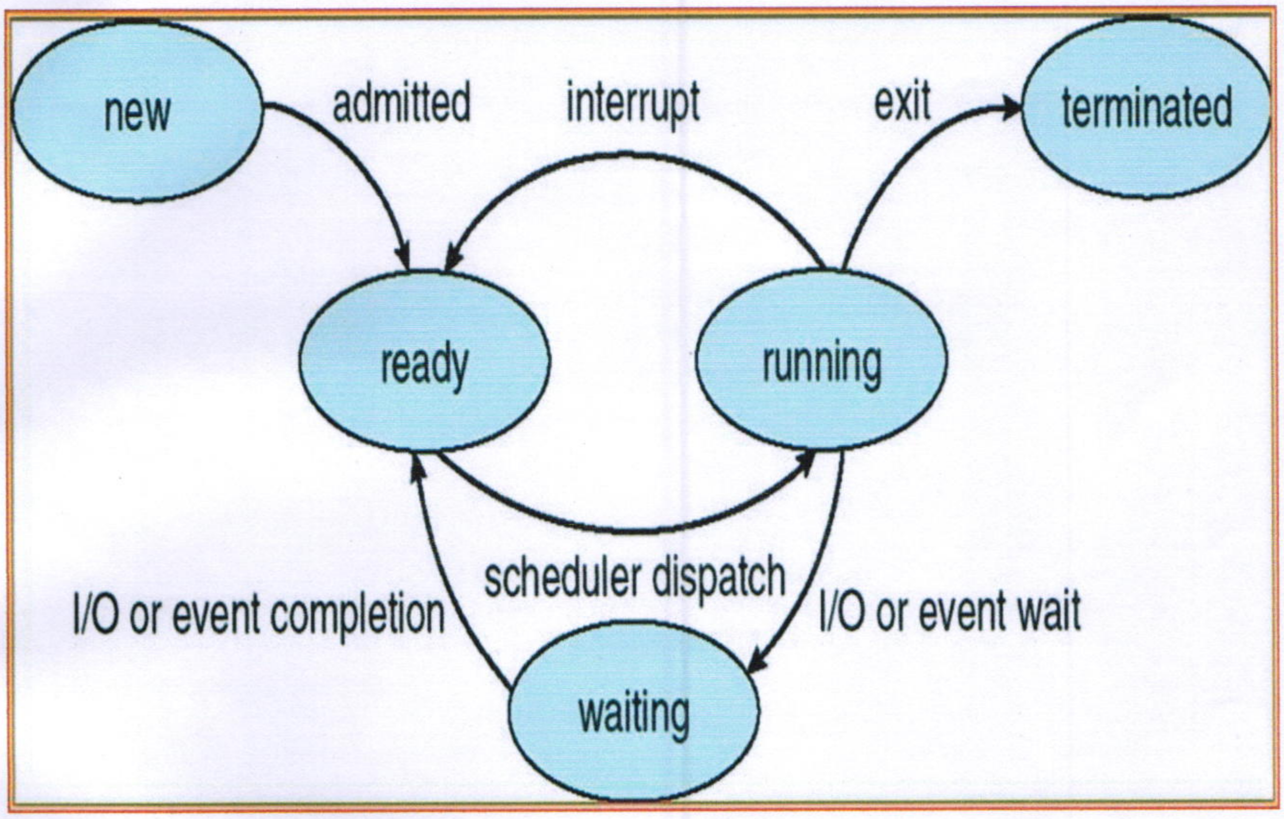


Fig: Process State Diagram

A process which is executed has various states. The states of the processes are called the status of the processes. The status includes whether the process has executed or whether the process is waiting for an I/O from the user or whether the process is waiting for the CPU to run the process after completion of the running process.

New State

When a user request for a Service from the System, then the System will first initialize the process or the System will call it an initial Process. So, every new Operation which is Requested to the System is known as the New Born Process.

Running State

When the Process is Running under the CPU, or When the Program is Executed by the CPU, then this is called as the Running process and when a process is Running then this will also provide us Some Outputs on the Screen.

Waiting State

When a Process is Waiting for Some Input and Output Operations then this is called as the Waiting State. And in this process, is not under the Execution instead the Process is Stored out of Memory and when the user will provide the input then this will Again be on ready State.

Ready State

When the Process is Ready to Execute but he is waiting for the CPU to Execute then this is called as the Ready State. After the Completion of the Input and outputs the Process will be on Ready State means the Process Will Wait for the processor to Execute.

Terminating State

After the Completion of the Process, the Process will be Automatically terminated by the CPU. So, this is also called as the Terminated State of the Process. After Executing the Whole Process the Processor Will Also reallocate the Memory which is allocated to the Process. So, this is called as the Terminated Process.

2.1.4 Scheduling Criteria

Many criteria have been suggested for CPU scheduling algorithms, which characteristics are used for comparison can make a substantial difference in which algorithm is judged to be the best. The criteria include the following:

- **CPU Utilization:** To make out the best use of CPU and not to waste any CPU cycle, CPU would be working most of the time (Ideally 100% of the time). Considering a real system, CPU usage should range from 40% (lightly loaded) to 90% (heavily loaded).

- **Throughput:** It is the total number of processes completed per unit time or rather say total amount of work done in a unit of time. This may range from 10/second to 1/hour depending on the specific processes.

- **Turnaround Time:** It is the amount of time taken to execute a particular process, i.e. the interval from time of submission of the process to the time of completion of the process.

- **Waiting Time:** The sum of the periods spent waiting in the ready queue amount of time a process has been waiting in the ready queue to acquire get control on the CPU.

- **Load Average:** It is the average number of processes residing in the ready queue waiting for their turn to get into the CPU.

- **Response Time:** Amount of time it takes from when a request was submitted until the first response is produced. Remember, it is the time till the first response and not the completion of process execution.

2.1.5 Context Switching

In computing, a context switch is the process of storing and restoring the state (more specifically, the execution context) of a Process or thread so that execution can be resumed from the same point at a later time. This enables multiple processes to share a single CPU and is an essential feature of a multitasking operating system.

2.1.6 Optimization Criteria

- Maximum CPU utilization
- Maximum throughput
- Minimum turnaround time
- Minimum waiting time
- Minimum response time.

2.2 Previous work

2.2.1 Scheduling Algorithms

CPU scheduling deals with the problem of deciding which process in the ready queue is to be allocated to the CPU. There are many CPU scheduling algorithms implemented. In this section, we discuss several of them.

2.2.1.1 First-Come First-Serve Scheduling

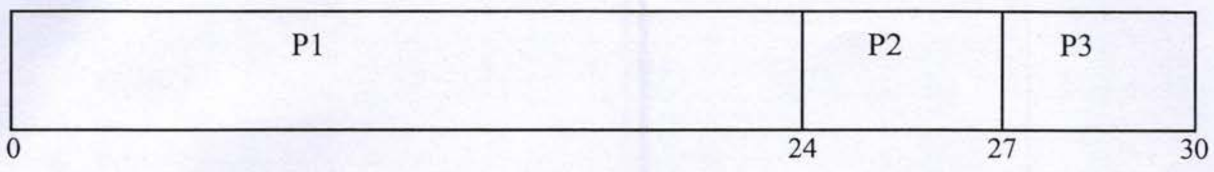
It is the simplest CPU scheduling algorithm. With this scheme, the process that requests the CPU first is allocated the CPU first. The implementation of the FCFS policy is easily managed with FIFO queue. When a process enters into the ready queue its PCB is linked onto the tail of the queue. When the CPU is free it is allocated to the process at the head of the queue. Then the running process is removed from the queue.

The average waiting time for this FCFS is quite long. Consider the following set of processes that arrive at time 0, with length of the CPU burst given in milliseconds:

Process	**Burst time**
P1	24
P2	3
P3	3

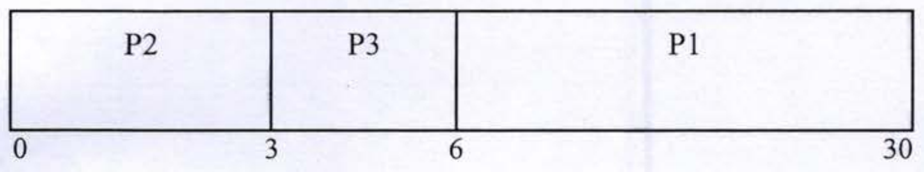
Suppose that the processes arrive in the order: *P1, P2 and P3.*

The Gantt chart for the schedule is:



- Waiting time for P1 =0, P2=24, P3=27
- Average waiting time (0+24+27)/3=17

Suppose that the processes arrive in the order *P2*, *P3*, *P1*.
The Gantt chart for the schedule is:



- Waiting time for P1=6, P2 = 0, P3 = 3
- Average waiting time (6+3+0)/3= 3

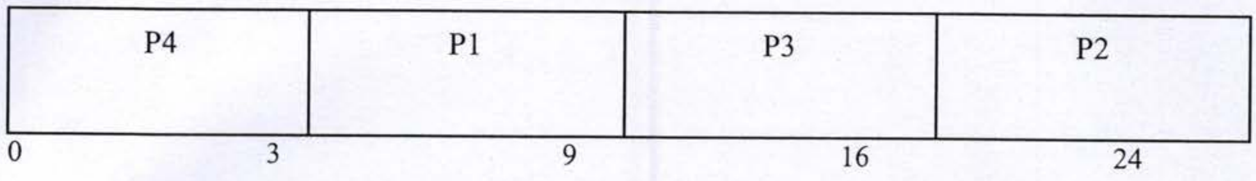
The average waiting time under an FCFS policy is generally not minimal and may vary substantially if the process's CPU burst times vary greatly. In addition, consider the performance of FCFS scheduling in a dynamic situation. Assume we have one CPU-bound process and many I/O-bound processes. As the processes flow around the system, the following scenario may result. The CPU-bound process will get and hold the CPU. During this time, all the other processes will finish their I/O and will move into the ready queue, waiting for the CPU. While the processes wait in the ready queue, the I/O devices are idle. Eventually, the CPU-bound process finishes its CPU burst and moves to an I/O device. All the I/O-bound processes, which have short CPU bursts, execute quickly and move back to the I/O queues. At this point, the CPU sits idle. The CPU-bound process will then move back to the ready queue and be allocated the CPU. Again, all the I/O processes end up waiting in the ready queue until the CPU-bound process is done. There is a convoy effect as all the other processes wait for the one big process to get off the CPU. This effect results in lower CPU and device utilization than might be possible if the shorter processes were allowed to go first. The FCFS scheduling algorithm is no preemptive. Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting I/O. The FCFS algorithm is thus particularly troublesome for time-sharing systems, where it is important that each user get a share of the CPU at regular intervals. It would be disastrous to allow one process to keep the CPU for an extended period.

2.2.1.2 Shortest Job First Scheduling

A different approach to CPU scheduling is the shortest-job-first (SJF) scheduling algorithm. This algorithm associates with each process the length of the process's next CPU burst. When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie. As an example of SJF scheduling, consider the following set of processes, with the length of the CPU burst given in milliseconds:

Process	**Burst time**
P!	**6**
P2	**8**
P3	**7**
P4	**3**

Using SJF scheduling, we would schedule these processes according to the following Gantt chart:

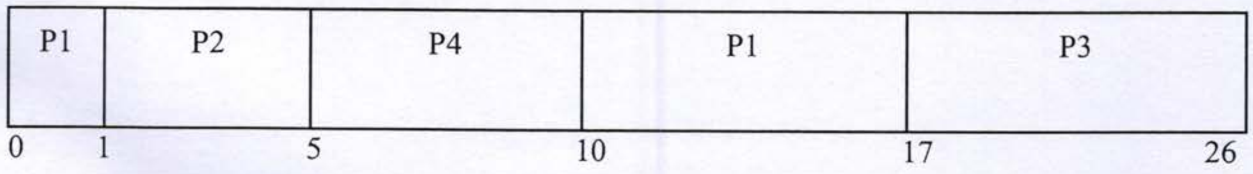


The waiting time is 3 milliseconds for process P1, 16 milliseconds for process *P2, 9* milliseconds for process *P3,* and 0 milliseconds for process *P4.* Thus, the average waiting time is (3 + 16 + 9 + 0)/4 = 7 milliseconds. By comparison, if we were using the FCFS scheduling scheme, the average waiting time would be 10.25 milliseconds. The SJF scheduling algorithm is provably *optimal,* in that it gives the minimum average waiting time for a given set of processes. Moving a short process before a long one decreases the waiting time of the short process more than it increases the waiting time of the long process. Consequently, the *average* waiting time decreases. The SJF algorithm can be either preemptive or non-preemptive. The choice arises when a new process arrives at the ready queue while a previous process is still executing. The next CPU burst of the newly arrived process may be shorter than what is left of the currently executing process. A preemptive SJF algorithm will preempt the currently execute process, whereas a non-preemptive SJF algorithm will allow the currently running process to finish its CPU burst. Sometimes, non-preemptive SJF scheduling is also called **Shortest Process Next (SPN)** scheduling and preemptive SJF is called **Shortest remaining time (SRT)** scheduling.

As an example, consider the following four processes, with the length of the CPU burst given in milliseconds:

Process	**Arrival time**	**Burst time**
P1	**0**	**8**
P2	**1**	**4**
P3	**2**	**9**
P4	**3**	**5**

If the processes arrive at the ready queue at the times shown and need the identical burst times, then the resulting preemptive SJF scheduling is as depicted in the following Gantt chart:



Process *P1* is started at time 0, since it is the only process in the queue. Process *P2* arrives at time 1. The remaining time for process *P1* (7 milliseconds) is larger than the time required by process *P2* (4 milliseconds), so process *P1* is preempted, and process *P2* is scheduled. The average waiting time for this example is ((10 — 1) + (1 — 1) + (17 — 2) + (5 — 3))/4 = 26/4 6.5 milliseconds. Non-preemptive SJF scheduling would result in an average waiting time of 7.75 milliseconds.

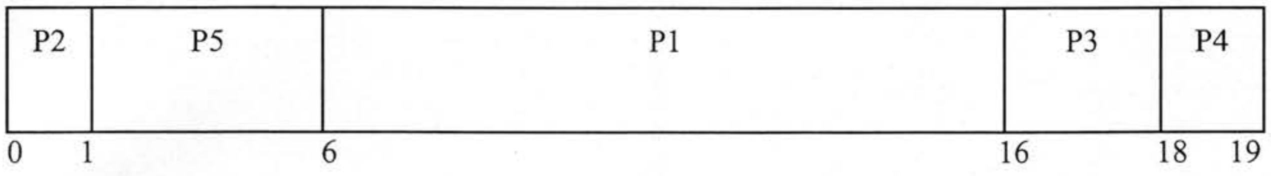
2.2.1.3 Priority Scheduling

The SJF algorithm is a special case of the general priority scheduling algorithm. A priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal-priority processes are scheduled in FCFS order. An SJF algorithm is simply a priority algorithm where the priority (p) is the inverse of the (predicted) next CPU burst. The larger the CPU burst, the lower the priority and vice versa. Note that we discuss scheduling in terms of *high* priority and *low* priority. Some systems use low numbers to represent low priority; others use low numbers for high priority. This difference can lead to confusion. In this text, we assume that low numbers represent high priority

As an example, consider the following set of processes, assumed to have arrived at time 0, in the order *P1, p2... P5.* With the length of the CPU burst given in milliseconds:

process	**Burst time**	**Priority**
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2

Using priority scheduling, we would schedule these processes according to the following Gantt chart



The average waiting time is 8.2 milliseconds.

Priorities can be defined either internally or externally. Internally defined priorities use some measurable quantity or quantities to compute the priority of a process. For example, time limits, memory requirements, the number of open files, and the ratio of average I/O burst to average CPU burst have been used in computing priorities. External priorities are set by criteria outside the operating system, such as the importance of the process, the type and amount of funds being paid for computer use, the department sponsoring the work, and other, often political, factors.

Priority scheduling either preemptive or non-preemptive. When a process arrives at ready queue its priority is compared with the priority of the current running process. A preemptive CPU scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the currently running process. A non-preemptive priority scheduling algorithm will simply put the new process at the head of the ready queue. A major problem of priority scheduling algorithm is indefinite blocking or starvation. A process is ready to run but waiting for the CPU can be considered as blocked. A priority scheduling algorithm can leave some low priority processes waiting indefinitely. In a heavily loaded computer system a steady state of higher priority processes can prevent the low priority process from ever getting the CPU. Generally, one of two things will happen. Either the process will eventually run or the or the computer system will eventually crash or lose all unfinished low priority processes.

A solution to the problem of indefinite blockage of low priority processes is aging. Aging is a technique of gradually increase the priority of that processes that wait in the system for a long time.

2.2.1.4 Round-Robin Scheduling

The round-robin (RR) scheduling algorithm is designed especially for timesharing systems. It is similar to FCFS scheduling, but preemption is added to switch between processes. A small unit of time, called a time quantum or time slice, is defined. A time quantum is generally from 10 to 100 milliseconds. The ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum.

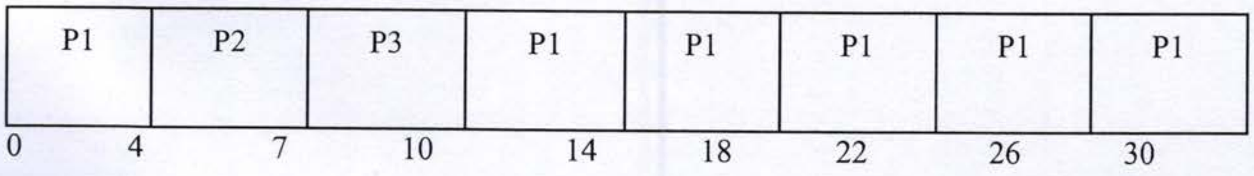
To implement RR scheduling, we keep the ready queue as a FIFO queue of processes. New processes are added to the tail of the ready queue. The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process.

One of two things will then happen. The process may have a CPU burst of less than 1 time quantum. In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue. Otherwise, if the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the operating system. A context switch will be executed, and the process will be put at the tail of the ready queue. The CPU scheduler will then select the next process in the ready queue.

The average waiting time under the RR policy is often long. Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

Processes	**Burst time**
P1	24
P2	3
P3	3

If we use a time quantum of 4 milliseconds, then process *Pi* gets the first 4 milliseconds. Since it requires another 20 milliseconds, it is preempted after the first-time quantum, and the CPU is given to the next process in the queue, process *P2.* Since process *P2* does not need 4 milliseconds, it quits before its time quantum expires. The CPU is then given to the next process, process *P3.* Once each process has received 1 time quantum, the CPU is returned to process *P1* for an additional time quantum. The resulting RR schedule is:

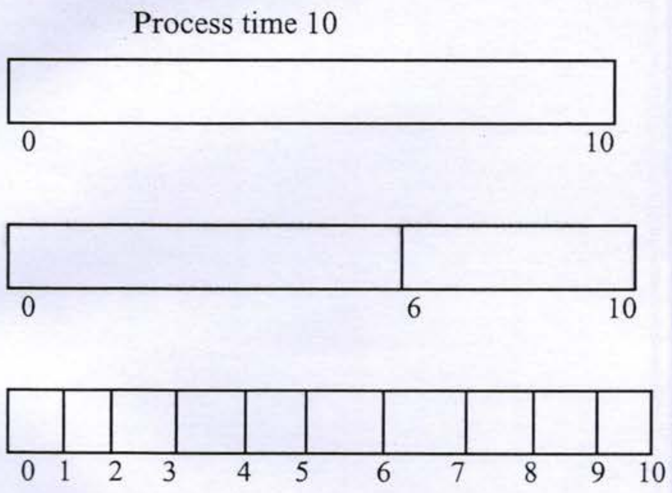


The average waiting time is 17/3 = 5.66 milliseconds.

In the RR scheduling algorithm, no process is allocated the CPU for more than 1 time quantum in a row. If a process's CPU burst exceeds 1 time quantum, that process is *preempted* and is put back in the ready queue. The RR scheduling algorithm is thuspreemptive.

The performance of the RR algorithm depends heavily on the size of the time quantum. At one extreme, if the time quantum is extremely large, the RR policy is the same as the FCFS policy. If the time quantum is extremely small then the RR process is called time sharing and creates the appearance that each of n process has its own processor running at 1/n the speed of real processo

In software, we also need to consider the effect of context switching on the performance of RR Scheduling. Let's assume we have one process of 10 time units, if the quantum is 12 time units the process finishes less than 1 time unit with no overhead. If the quantum is 6 time units however the process requires 2-time quantum resulting in a context switch. If the time quantum is 1 time unit then 9 context switch will occur, slowing the execution process accordingly.



Time Quantum	**Context Switch**
12	0
6	1
1	9

Fig: Way in which a smaller time quantum increases context switches

Chapter 3
Proposed Model

3.1 Proposed Approach

Our proposed approach is just a mixture of the traditional FCFS, SJF and the round robin scheduling algorithm. According to the proposed algorithm the process come with burst time and a time quantum allocate to the ready queue according to the FCFS. After that the Process will sort according to their burst time and the dynamic time quantum will be calculated using the median formula. Thus, the process will be executed until queue is empty.

To implement this, we will get a better result as in less no of context switching, minimum turn-around time and the waiting time is also minimum.

Initially Processes come to the queue as follows:

Process Name	**Burst Time**	**Time quantum(ms)**
1. P1	25	5
2. P2	17	5
3. P3	10	5
4. P4	19	5

After sorting the above situation, the Process will be ready for execution in the following order:

Process Name	Burst Time	Time quantum(ms)
1. P3	10	3.5
2. P2	17	7
3. P4	19	10.5
4. P1	25	14

The time quantum is calculated by the median formula. That is as follows:

n = total no of process

If n = odd

Then m = TQ = y * ((n+2)/2)

If n = even

Then m = TQ = ½(y*(n/2) + y * ((1+n)/2)

Where y = serial no of the list

M = median TQ

So, the time quantum of the processes is calculated individually by the above formula.

According to the example for:

P3 = ½ * (1*(4/2)) + 1 * ((1+4)/2) = 3.5

P2 = ½ * (2*(4/2)) + 2 * ((1+4)/2) = 7

P4 = ½ * (3*(4/2)) + 3 * (5/2) = 10.5

P1= ½ * (4*(4/2)) + 4 * (5/2) = 14

3.2 Proposed Algorithm

Step 1: Start

Step 2: Make a ready queue of the processes.

Step 3: Allocate processes following the FCFS into ready queue by default

Step 4: Sort all the processes according to their burst time in ascending order

Step 5: Calculate optimal time quantum as mean dynamic time quantum. Using median formula.

Step 6: Execute processes as default way depending on dynamic time quantum.

Step 7: Repeat step 3 to 6 until ready queue is not empty.

Step 8: If new process come in between running process , the program will restart from step 4.

Step 9: End

3.3 Implementation

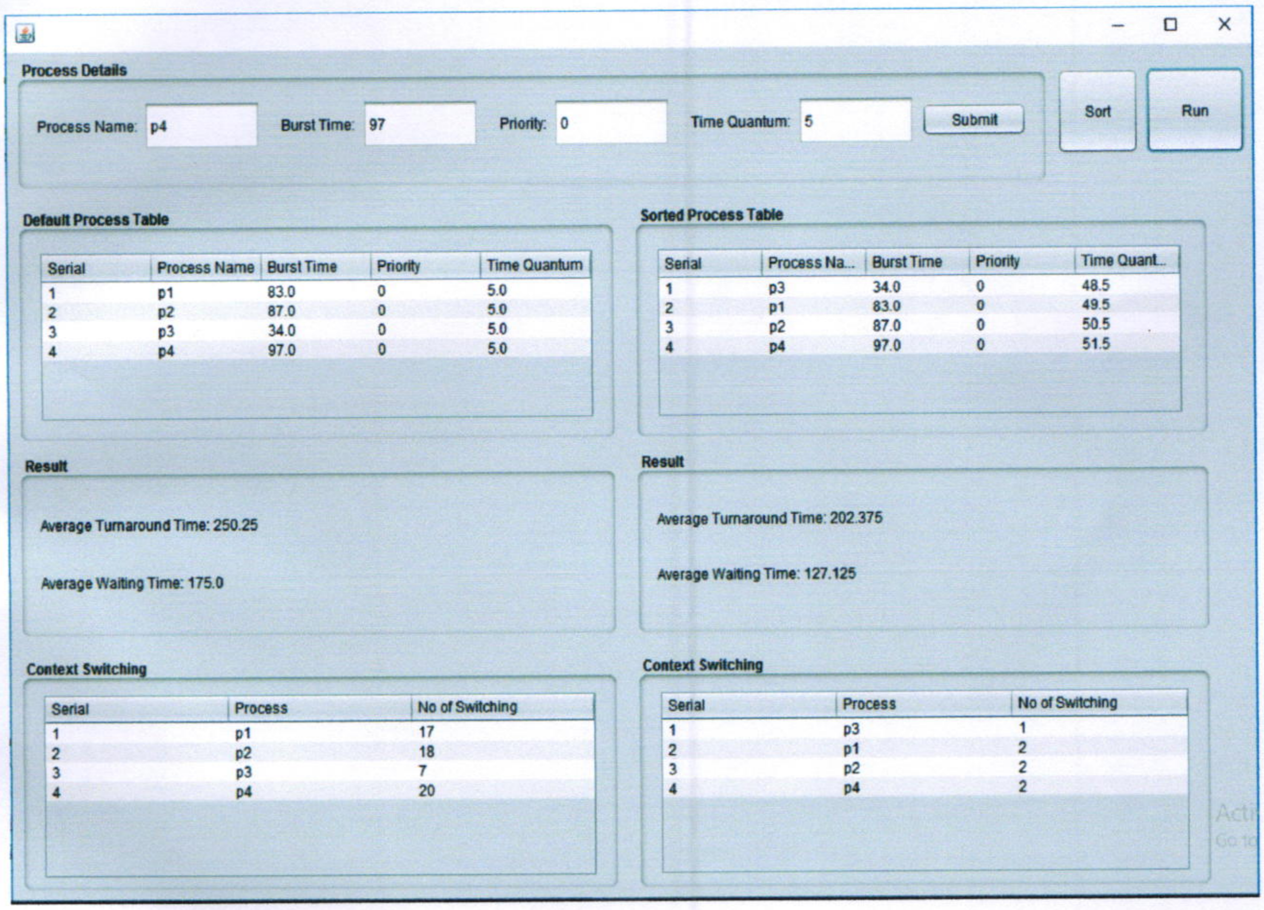
3.3.1 Source Code

```
public void optimalTimeQuantum(){
int n = ProcessName.size();
     double m = 0.0;
if(n%2==0)


{//even
        for(inti=0;i<ProcessName.size();i++)
{
          m = .5*( (i*(n/2)) + ( i* ((1+n)/2) ) );
SortedMedian.add(m);
        }
    }
else
{//odd
        for(inti=0;i<ProcessName.size();i++)
{
          m = i*( (n+2)/2 );
SortedMedian.add(m);
        }
    }

    double highestBT = Collections.max(BurstTime);
    for(inti=0;i<SortedMedian.size();i++){
          double otq = (highestBT + SortedMedian.get(i)) / 2;
dTimeQuantum.add(otq);
        }
  }


        public void NewModelRun()
```

```
{
totalTime = 0;
int size = NewCPU.size();
      while(size>0)
{

for(inti=0;i<newR.length;i++)
{
            double tq = (double) newR[i][2];
            double bt = (double) newR[i][1];


               if(bt>tq&&bt>0)
{
totalTime = totalTime + tq;



bt = bt-tq;
newR[i][1]=bt;
newR [i][3] = totalTime;

int switching = (Integer)newR[i][5] + 1;
newR[i][5] =  switching;
               }
else if(bt<=tq&&bt>0)
{
totalTime = totalTime + bt;
bt = bt-tq;
newR[i][1]=bt;
newR [i][3] = totalTime;
              size=size-1;

int switching = (Integer)newR[i][5] + 1;



newR[i][5] =  switching;
            }
          }
      }
```

3.4 Results

Result after first test:



Process Details

Process Name: p4 Burst Time: 97 Priority: 0 Time Quantum: 5

Default Process Table

Serial	Process Name	Burst Time	Priority	Time Quantum
1	p1	83.0	0	5.0
2	p2	87.0	0	5.0
3	p3	34.0	0	5.0
4	p4	97.0	0	5.0

Sorted Process Table

Serial	Process Na...	Burst Time	Priority	Time Quant...
1	p3	34.0	0	48.5
2	p1	83.0	0	49.5
3	p2	87.0	0	50.5
4	p4	97.0	0	51.5

Result

Average Turnaround Time: 250.25

Average Waiting Time: 175.0

Result

Average Turnaround Time: 202.375

Average Waiting Time: 127.125

Context Switching

Serial	Process	No of Switching
1	p1	17
2	p2	18
3	p3	7
4	p4	20

Context Switching

Serial	Process	No of Switching
1	p3	1
2	p1	2
3	p2	2
4	p4	2

Fig: Simulation Software

So, we got the better result from traditional round robin scheduling algorithm.

The comparison is as follows:

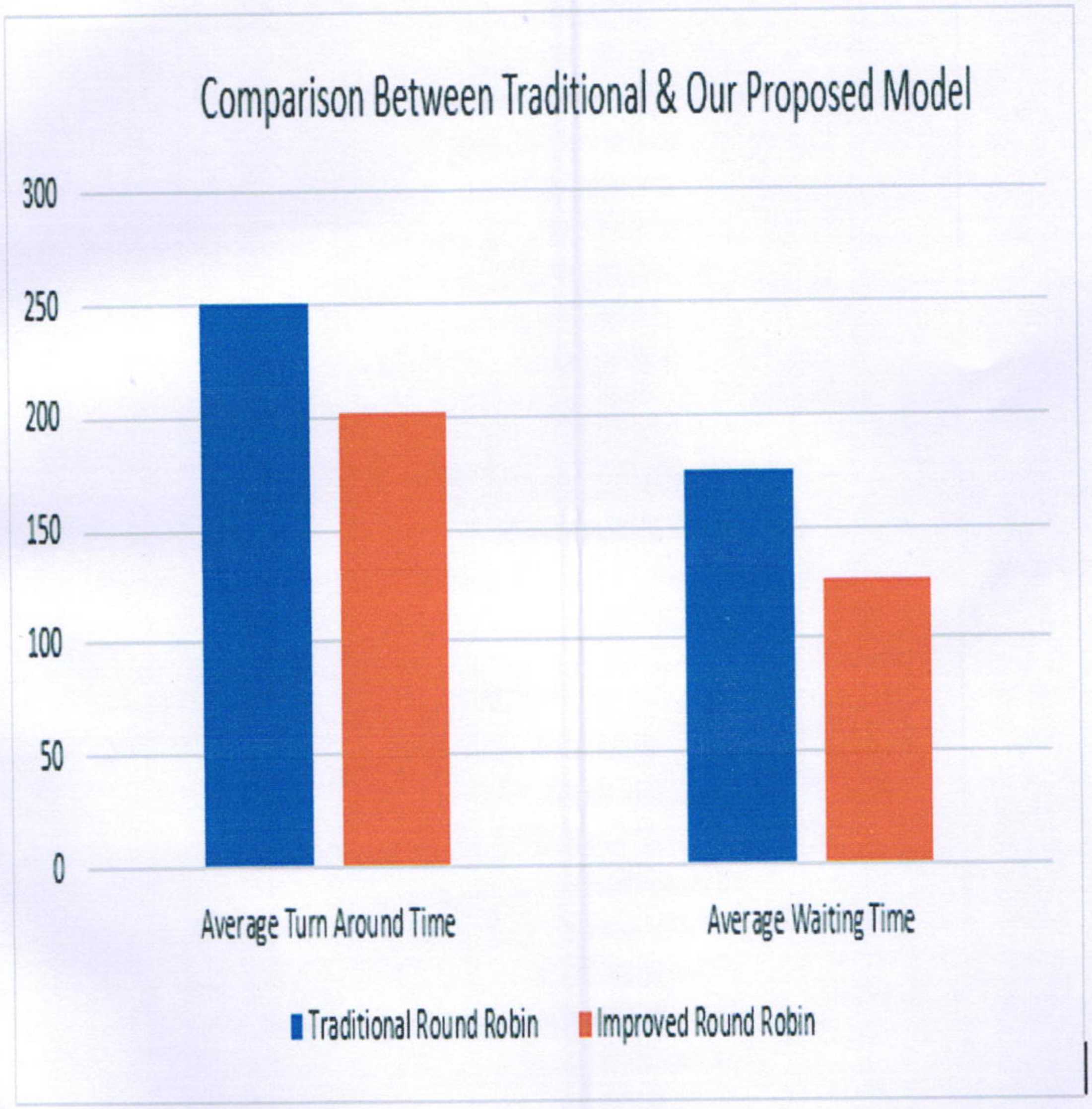
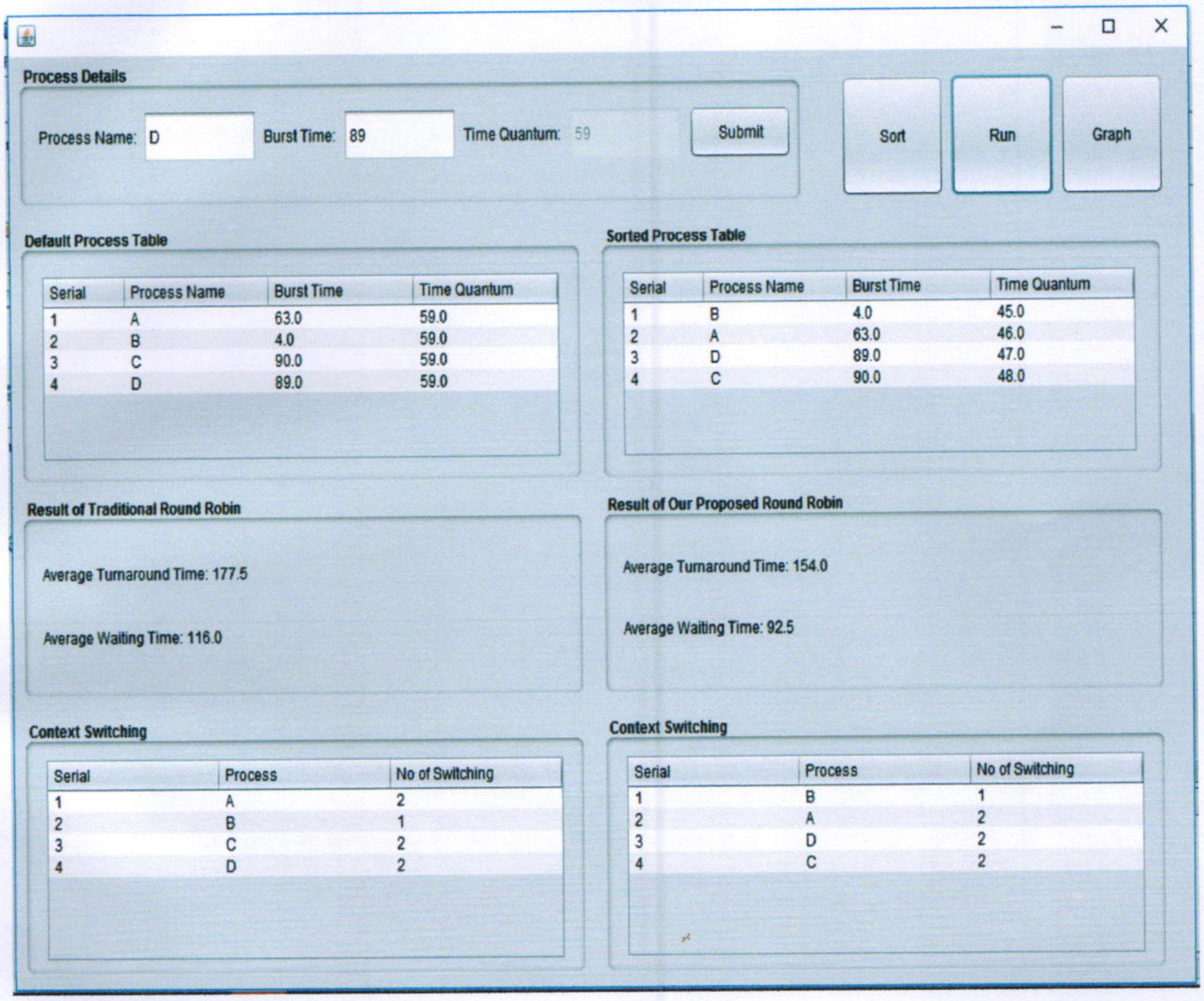


Fig: Simulation Graph

Result after second test:



Process Details

Process Name: D Burst Time: 89 Time Quantum: 59 Submit

Sort Run Graph

Default Process Table

Serial	Process Name	Burst Time	Time Quantum
1	A	63.0	59.0
2	B	4.0	59.0
3	C	90.0	59.0
4	D	89.0	59.0

Sorted Process Table

Serial	Process Name	Burst Time	Time Quantum
1	B	4.0	45.0
2	A	63.0	46.0
3	D	89.0	47.0
4	C	90.0	48.0

Result of Traditional Round Robin

Average Turnaround Time: 177.5

Average Waiting Time: 116.0

Result of Our Proposed Round Robin

Average Turnaround Time: 154.0

Average Waiting Time: 92.5

Context Switching

Serial	Process	No of Switching
1	A	2
2	B	1
3	C	2
4	D	2

Context Switching

Serial	Process	No of Switching
1	B	1
2	A	2
3	D	2
4	C	2

Fig: Simulation Software

Here, we also got the better result from traditional round robin scheduling algorithm.

The comparison is as follows:

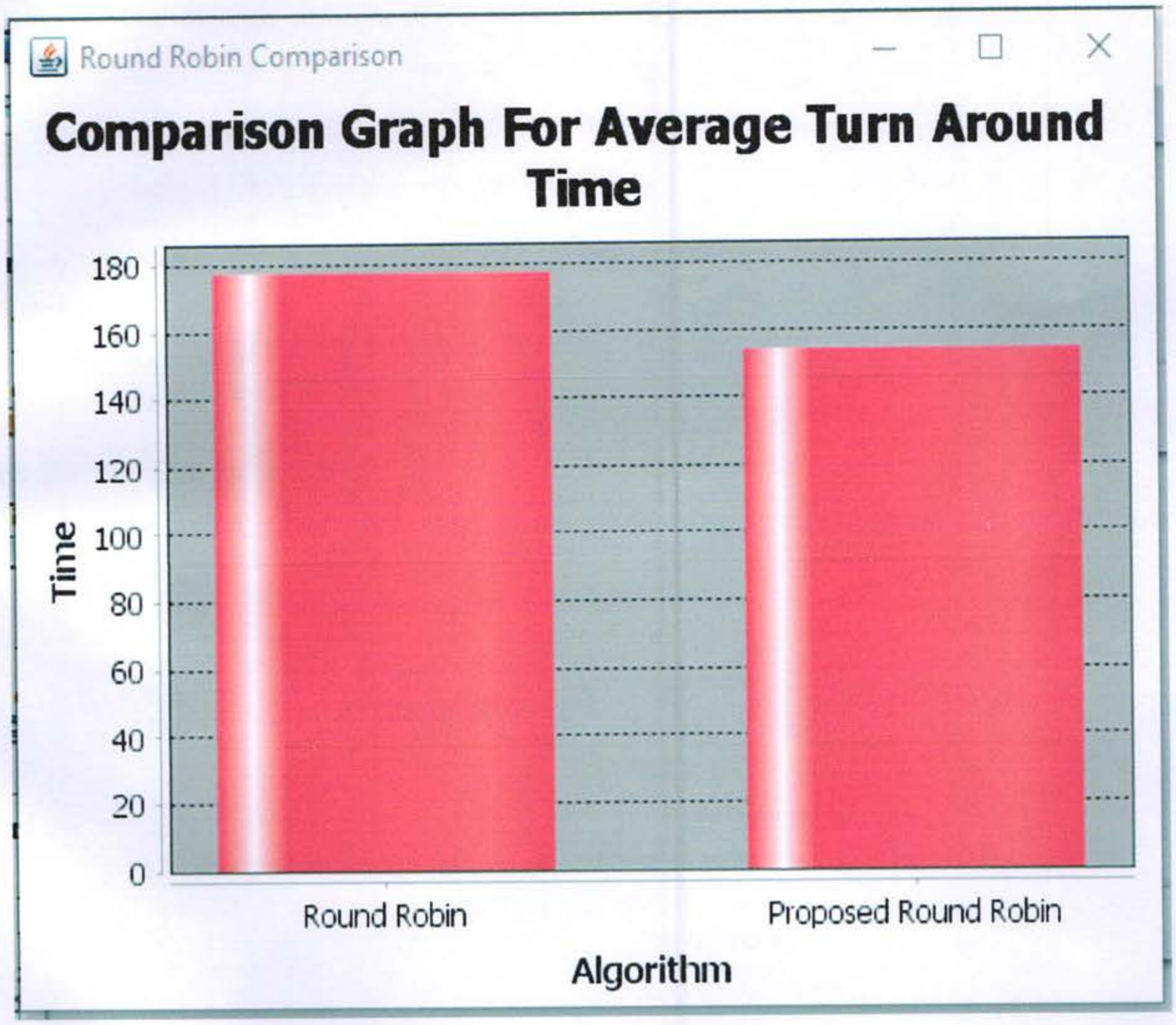


Fig: Simulation Graph

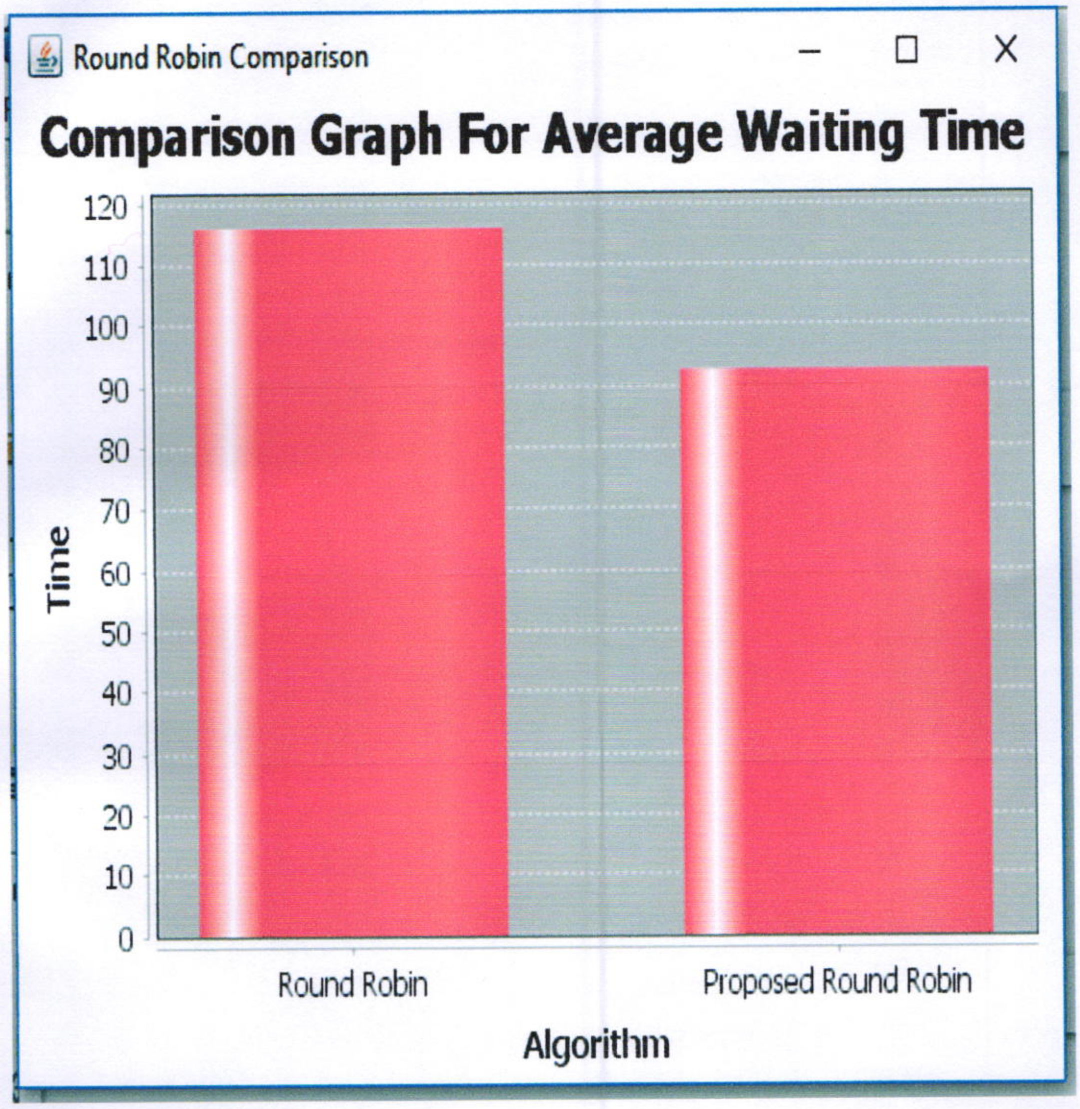


Fig: Simulation Graph

Chapter 4

Conclusion

4.1 Future Scope

The proposed model of our thesis is basically a part of pure computer science and much theoretically analyzed. The accuracy and real life implementation of this model is so challenging. Still we are hopeful extend our work with real life. Our future plans would be the following:

- Implement the proposed model with Linux kernel in real life.
- Perform various test (real life application) to measure accuracy of this model.
- Write paper documentation for future reference.
- Publish this model for public uses.

4.2 Effectiveness of the proposed model

This paper is supposed to be an analytical paper which compares and analyze the traditional scheduling algorithm with the proposed algorithm. Unlike all the traditional algorithm our proposed algorithm we can optimize the processor using the dynamic time quantum of typical round robin scheduling algorithm. Except this, in our proposed model the turnaround time and the average waiting time is minimum and the context switching is also less over traditional RR which are the most crucial part of any scheduling algorithm. We have done till this. Hopefully it will be implemented with the Linux kernel with the real time.

References

1. A. R. Dash. S. K. Sahu and S. K. Samantra, An Optimized Round Robin CPU Scheduling Algorithm with Dynamic Time Quantum,*International Journal of Computer Science, Engineering and Information Technology (IJCSEIT), Vol. 5, No.I,* February 2015.
2. Manish Kumar Mishra1 and Dr. Faizur Rashid," An Improved Round Robin CPU Scheduling Algorithm with Varying Time Quantum", *International Journal of Computer Science, Engineering and Applications, Vol.4, No.4*, August 2014
3. Goel N., Garg R. B., "Improvised Optimum Multilevel Dynamic Round Robin Algorithm for Optimizing CPU Scheduling", International Journal of Computer Applications , ISSN 0975 – 8887, Volume – No., August 2015.
4. Rajput I., Gupta D., "A Priority based Round Robin CPU Scheduling Algorithm for Real Time Systems", IJIET, Vol. 1 Issue 3 Oct. 2012
5. Abbas Noon1, Ali Kalakech2, SeifedineKadry, "A New Round Robin Based Scheduling Algorithm for Operating Systems:Dynamic Quantum Using the Mean Average", *IJCSI International Journal of Computer Science Issues, Vol. 8, Issue 3, No. 1,* May 2011 ISSN (Online) 1694-0814.
6. Rakesh Mohanty, H.S. Behera and et. al, Design and Performance Evaluation of a new proposed Shortest Remaining Burst Round Robin(SRBRR) scheduling algorithm, Proceedings of the International Symposium on Computer Engineering and Technology(ISCET), March, 2010.

7. Rami J. Matarneh, "Self-Adjustment Time Quantum inRound Robin Algorithm Depending on Burst Time of theNow Running Processes", American Journal of AppliedSciences, Vol 6, No. 10, 2009.
8. Abdulrazak Abdulrahim, Saleh E. Abdullahi & Junaidu B. Sahalu, (2014) "A New Improved Round Robin (NIRR) CPU Scheduling Algorithm", International Journal of Computer Applications, Vol. 90, No. 4, pp 27-33.
9. M Lavanya & S. Saravanan, (2013) "Robust Quantum Based Low-power Switching Technique to improve System Performance", International Journal of Engineering and Technology, Vol. 5, No. 4, pp 3634-3638.
10. Mehdi Neshat, Mehdi Sargolzaei, Adel Najaran & Ali Adeli, (2012) "The New method of Adaptive CPU Scheduling using Fonseca and Fleming's Genetic Algorithm", Journal of Theoretical and Applied Information Technology, Vol. 37, No. 1, pp 1-16.
11. Debashree Nayak, Sanjeev Kumar Malla & Debashree Debadarshini, (2012) "Improved Round Robin Scheduling using Dynamic Time Quantum", International Journal of Computer Applications, Vol. 38, No. 5, pp 34-38. International Journal of Computer Science, Engineering and Applications (IJCSEA) Vol.4, No.4, August 2014.
12. M.K. Srivastav, Sanjay Pandey, Indresh Gahoi & Neelesh Kumar Namdev, (2012) "Fair Priority Round Robin with Dynamic Time Quantum", International Journal of Modern Engineering Research, Vol. 2, Issue 3, pp 876-881.