Nile University

CSCI 315

Optimizing CPU Scheduling for Real Time Applications Using Mean-Difference Round Robin Algorithm

Presented to Dr. Eman Gawish

Ahmed Khaled Askar	18101381
Hussein El-Shazly	19106038
Mohamed Ahmed	18102714
Darwish Mohamed	18102012

Contents

Abstract:

The Mean-Difference Round Robin Algorithm is a revolutionary approach proposed in this research for optimizing CPU scheduling for real-time applications. The suggested algorithm determines the average burst time for all processes in the ready queue. The difference between a process burst time and the calculated mean burst time is then determined. For each process in theready queue, this phase is repeated. The suggested algorithm then finds the process with the highestDifference value, allocates it to the CPU, and runs it for a single time slice. When the process's time slice expires, the next process in the ready queue with the biggest difference value is selectedand performed for one time slice. All of the processes in the ready queue go through the same procedure. The suggested algorithm's experimental results were compared to the results of other standard scheduling methods, and the proposed Mean Difference Round Robin Algorithm was determined to produce optimal scheduling.

Keywords*: CPU Scheduling, Average Turnaround time, Average Waiting time, Standard Round Robin, Mean difference Round Robin.*

Introduction:

The requirement for most modem systems to perform multitasking (running numerous processes at the same time) and multiplexing necessitates the use of a scheduling algorithm (transmit multiple flows simultaneously). CPU The foundation of multiprogram operating systems is scheduling. The operating system can make the computer system productive by switching the CPU among the processes. Only one process can execute at a time on a single processor system; all others must wait until the CPU is free and can be rescheduled. Multiprogramming's goal is to keep certain processes active at all times in order to maximize CPU utilization. The concept is straightforward. A process is run until it has to wait for something, usually the completion of an I/O request. The CPU then lies idle in a rudimentary computer system. All of this waiting is in vain; no real work is being done. We aim to make the most of this time through multiprogramming. At any given time, several processes are retained in memory. When a process must wait, the operating system diverts the CPU from

that process and gives it to another. This pattern persists. When one process is forced to wait, another process can take over CPU usage. This type of scheduling is a basic operating system function. Almost all computer resources are pre- programmed before they are used. Of course, one of the most important computer resources is theCPU. As a result, it plays a crucial role in operating system development.

Literature Review:

1- Standards for scheduling:

There are several CPU scheduling algorithms with distinct features and choosing one may favor one class of processes over another. When choosing a CPU scheduling method for a specific case,we must evaluate the properties of various algorithms. The following are some of the criteria:

- Context Switch The computer technique of storing and restoring the state of a CPU so that execution can be restarted from the same point at a later time is known as context switching. Context switches are typically computationally costly, resulting in time, memory, and scheduling overhead. As a result, much of the operating system design is focused on optimizing these switches.
- Throughput: Throughput is the number of processes completed in a given amount of time.When round robin scheduling is used, throughput will be slow. Context switching and throughput are inversely proportional.
- CPU Utilization aims to keep the CPU occupied as much as possible.
- Turnaround Time: Turnaround time is the sum of the time spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and performing input/output.It ought to be lower.
- Waiting Time: The amount of time a process has been waiting in the ready queue is knownas waiting time. The CPU scheduling technique has no effect on the amount of time a process spends executing or doing input-output; it only impacts the amount of time it spends waiting in the ready queue.
- Response Time: The time it takes to start responding and the time it takes to output the response is referred to as response time. Large reaction times are a disadvantage of round robin architecture since they degrade system performance.

As a result, we may deduce that a successful real-time and time-sharing scheduling algorithm must have the following characteristics: Maximum CPU utilization. Maximum throughput. Minimum turnaround time. Minimum waiting time. Minimum response time.

2- Standard Round Robin:

Round Robin scheduling is a preemptive variation of the first come, first served algorithm. The processes are ordered in the ready queue in a first-come, first-served order, and the processor executes the process from the ready queue according to the time slice. If the time slice expires while the process is still running on the processor, the scheduler will forcibly pre-empt the runningprocess and move it to the end of the ready queue, where it will be allocated to the next process inthe queue. The pre-empted process will move to the front of the ready queue and be run by the processor starting at the point of interruption. To implement the round robin architecture, a scheduler requires the time management function as well as a tick timer. The period of clock ticksis proportional to the time slice. In a soft real-time embedded application, the time slice length is crucial because missing deadlines has negligible implications on system performance. The time slice should not be too tiny to cause frequent context switches, and it should be slightly longer thanthe average calculation time for the process.

3- Optimized algorithm for Standard Round Robin:

- Phase 1: Apply RR scheduling with an initial time quantum to allocate every process to the CPU at the same time (say k units).
- Phase 2: After you've completed the first cycle, do the following: a) Increase the beginning time quantum by two (2k units). b) From the waiting queue, choose the shortest process and assign it to the CPU. c) After that, we must choose the next shortest procedure to perform, excluding the one that has already been completed in this phase.
- Phase 3: We must repeat phases 1 and 2 in order to complete all of the processes.

Methodology:

1- Introduction:

The bottleneck is a constraint in CPU Scheduling that results in inefficient CPU performance by failing to maximize throughput (try to service the largest number of processes

per unit of time) and avoid endless postponement and starvation (A process should not experience anunbounded wait time before or while process service).

2- Goals-objectives:

Main Goal: Optimize CPU using Mean-Difference Round Robin Algorithm	
Objectives	Description
Collect all the Processes in the ready queue	The mean burst time of all the processes in the ready queue is calculated.
Finds out the mean difference	Determines the difference between the burst time of a process and the calculated mean burst time. Repeat on all the processes in the ready queue
Finds out the largest difference	Using the provided algorithm, find the greatest difference value and assign it to the CPU for one time slice. When the process's time slice expires, look at the remaining burst time of the running process to see if it's equal to or less than the one time slice. Otherwise, the next process with the biggest difference value is selected from the ready queue and executed for one time slice, after which the remaining burst time is checked to determine whether it should be continued or not.
Repeat	The algorithm will be looped over all the process inside the ready queue.

3- Timeline:

Task	Due Date
Platform & Resources installation	1/6/2022
Prototype	1/6/2022
Adjustment and modification (implementation)	4/6/2022

Tools Used:

- Code Blocks
- Git hub
- Ubuntu

Algorithm of our proposed approach: Mean Difference Round Robin

Inputs:

1- Array of process id's called processes [], we calculate the number of processes called n.2- Array of burst time of all processes called burst_time[]

3- Time quantum called quantum

Outputs:

1- Display processes along with all details (burst time, total waiting time and totalturnaround time)

2- Average waiting time

3- Average Turnaround time

Methodology in Steps:

- Firstly, we implement the Mean Difference Round Robin:

1- The mean burst time of all the processes in the ready queue is calculated.

2- Determines the difference between the burst time of a process and the calculated meanburst time.

3- Repeat on all the processes in the ready queue

4- find the greatest difference value and assign it to the CPU for one time slice

5- the next process with the biggest difference value is selected from the ready queue andexecuted for one time slice. Repeat for all processes

6- Repeat all the steps above until all the processes in the ready queue are finished

7- Average waiting time is calculated.

8- Average Turnaround time is calculated.

Secondly, we implemented the standard Round Robin Algorithm

Thirdly, we will apply our test cases on the two algorithms, and we will compare the two results.

Implementation:

The Mean Difference Round Robin:

```cpp
#include<iostream>
using namespace std;
// Function to find the waiting time for all
// processes
int q[1000];
void rearrange(int processes[],int bt[], int n )
{
    for(int i = 0 ; i < n ; i++ )q[i]=processes[i]-1;
    int burst_sum = 0 , counter= 0;
    for(int i = 0 ; i < n ; i++ ){
        if(bt[i]){
            burst_sum += bt[i];
            counter++;
        }
    }
    if(!counter)return;
    int avg_burst = burst_sum/counter;

    for(int i = 0 ; i < n-1 ; i++ ){
        for(int j = i + 1 ; j < n ; j++ )
        if(avg_burst-bt[i]<avg_burst-bt[j])swap(q[i],q[j]);
    }
}
// Function to calculate turn around time
void findTurnAroundTime(int processes[], int n,
int bt[], int wt[], int tat[]){
    // calculating turnaround time by adding
    // bt[i] + wt[i]
    for (int i = 0; i < n ; i++)tat[i] = bt[i] + wt[i];
}
void findWaitingTime(int processes[], int n,
int bt[], int wt[], int quantum)
{
    // Make a copy of burst times bt[] to store remaining
    // burst times.
    int rem_bt[n];
    for (int i = 0 ; i < n ; i++)rem_bt[i] = bt[i];
    int t = 0; // Current time
    // Keep traversing processes in round robin manner
    // until all of them are not done.
    while (1){
        bool done = true;
        rearrange(processes,rem_bt,n);
        // Traverse all processes one by one repeatedly
        for (int i = 0 ; i < n; i++){
        // If burst time of a process is greater than 0
        // then only need to process further
            if (rem_bt[q[i]] > 0){
                done = false; // There is a pending process
                if (rem_bt[q[i]] > quantum){
                // Increase the value of t i.e. shows
                // how much time a process has been processed
                t += quantum;
                // Decrease the burst_time of current process
                // by quantum
                rem_bt[q[i]] -= quantum;
                }
                // If burst time is smaller than or equal to
                // quantum. Last cycle for this process
                else{
                    // Increase the value of t i.e. shows
                    // how much time a process has been processed
                    t = t + rem_bt[q[i]];
                    // Waiting time is current time minus time
                    // used by this process
                    wt[q[i]] = t - bt[q[i]];
                    // As the process gets fully executed
                    // make its remaining burst time = 0
                    rem_bt[q[i]] = 0;
                }
            }
            rearrange(processes,rem_bt,n);
        }
        // If all processes are done
        if (done == true)
        break;
    }
}
```

```cpp
// Function to calculate average time
void findavgTime(int processes[], int n, int bt[],
int quantum){
    int wt[n], tat[n], total_wt = 0, total_tat = 0;
    // Function to find waiting time of all processes
    findWaitingTime(processes, n, bt, wt, quantum);
    // Function to find turn around time for all processes
    findTurnAroundTime(processes, n, bt, wt, tat);
    // Display processes along with all details
    cout << "Processes "<< " Burst time "
    << " Waiting time " << " Turn around time\n";
    // Calculate total waiting time and total turn
    // around time
    for (int i=0; i<n; i++){
        total_wt = total_wt + wt[i];
        total_tat = total_tat + tat[i];
        cout << " " << i+1 << "\t\t" << bt[i] <<"\t "
        << wt[i] <<"\t\t " << tat[i] <<endl;
    }
    cout << "Average waiting time = "
    << (float)total_wt / (float)n;
    cout << "\nAverage turn around time = "
    << (float)total_tat / (float)n;
}
int main(){
    // process id's
    int processes[] = { 1, 2, 3,4};
    int n = sizeof processes / sizeof processes[0];
    // Burst time of all processes
    int burst_time[] = {53,17,68,24};
    // Time quantum
    int quantum = 10;
    findavgTime(processes, n, burst_time, quantum);
    return 0;
}
```

The standard Round Robin:

```cpp
#include<iostream>
using namespace std;
// Function to find the waiting time for all
// processes
void findWaitingTime(int processes[], int n,
int bt[], int wt[], int quantum)
{
    // Make a copy of burst times bt[] to store remaining
    // burst times.
    int rem_bt[n];
    for (int i = 0 ; i < n ; i++)rem_bt[i] = bt[i];
    int t = 0; // Current time
    // Keep traversing processes in round robin manner
    // until all of them are not done.
    while (1){
        bool done = true;
        // Traverse all processes one by one repeatedly
        for (int i = 0 ; i < n; i++){
            // If burst time of a process is greater than 0
            // then only need to process further
            if (rem_bt[i] > 0){
                done = false; // There is a pending process
                if (rem_bt[i] > quantum){
                    // Increase the value of t i.e. shows
                    // how much time a process has been processed
                    t += quantum;
                    // Decrease the burst_time of current process
                    // by quantum
                    rem_bt[i] -= quantum;
                }
                // If burst time is smaller than or equal to
                // quantum. Last cycle for this process
                else{
                    // Increase the value of t i.e. shows
                    // how much time a process has been processed
                    t += rem_bt[i];
                    // Waiting time is current time minus brust time
                    // used by this process
                    wt[i] = t - bt[i];
                    // As the process gets fully executed
                    // make its remaining burst time = 0
                    rem_bt[i] = 0;
                }
            }
        }
        // If all processes are done
        if (done)break;
    }
}
// Function to calculate turn around time
void findTurnAroundTime(int processes[], int n,
int bt[], int wt[], int tat[]){
    // calculating turnaround time by adding
    //From Eq  (Waiting time = Turn around time - brust time)
    // bt[i] + wt[i]
    for (int i = 0; i < n ; i++)tat[i] = bt[i] + wt[i];
}
```

```cpp
// Function to calculate average time
void findavgTime(int processes[], int n, int bt[],
int quantum)
{
    int wt[n], tat[n], total_wt = 0, total_tat = 0;
    // Function to find waiting time of all processes
    findWaitingTime(processes, n, bt, wt, quantum);
    // Function to find turn around time for all processes
    findTurnAroundTime(processes, n, bt, wt, tat);
    // Display processes along with all details
    cout << "Processes "<< " Burst time "
    << " Waiting time " << " Turn around time\n";
    // Calculate total waiting time and total turn
    // around time
    for (int i=0; i<n; i++){
        total_wt = total_wt + wt[i];
        total_tat = total_tat + tat[i];
        cout << " " << i+1 << "\t\t" << bt[i] <<"\t "
        << wt[i] <<"\t\t " << tat[i] <<endl;
    }
    cout << "Average waiting time = "
    << (float)total_wt / (float)n;
    cout << "\nAverage turn around time = "
    << (float)total_tat / (float)n;
}
int main(){
    // process id's
    int processes[] = { 1, 2, 3,4};
    int n = sizeof processes / sizeof processes[0];
    // Burst time of all processes
    int burst_time[] = {53,17,68,24};
    // Time quantum
    int quantum = 10;
    findavgTime(processes, n, burst_time, quantum);
    return 0;
}
```

Results:

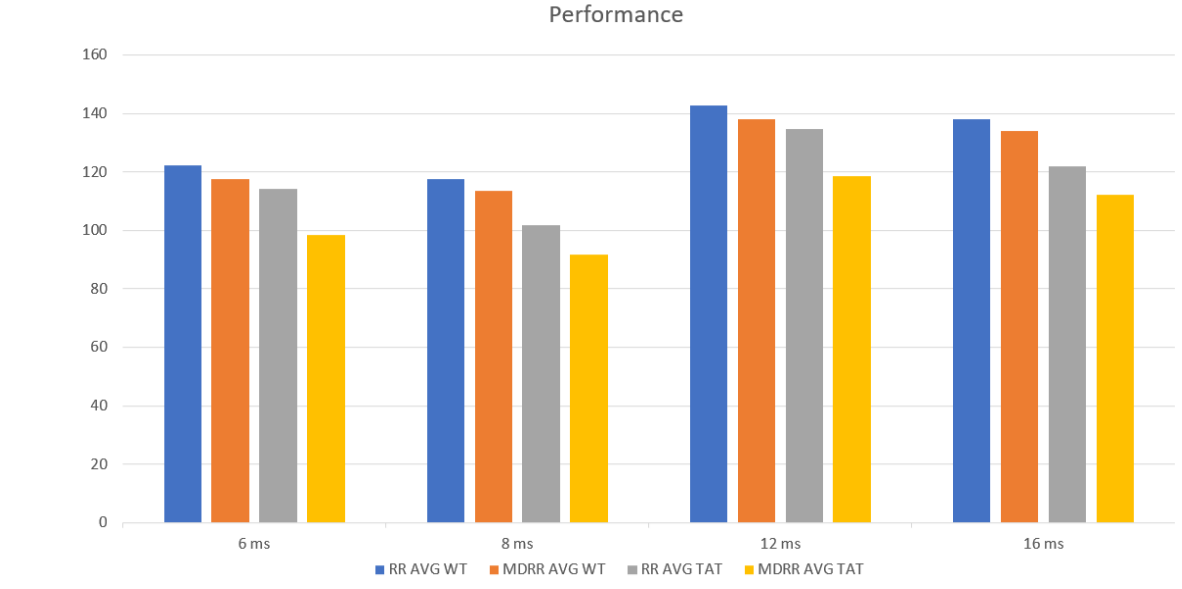
We will compare between the two algorithms using many test cases. We will discuss Four of them in this report:

Test case 1:

Quantum is 6, 8, 12, 16 milliseconds

Process Name	Burst Time
P1	5
P2	13
P3	12
P4	28
P5	19
P6	16
P7	21
P8	30
P9	33
P10	27

Result of test case one: Quantum	6 ms	8 ms	12 ms	16 ms
RR AVG WT	122.4	117.5	114.3	98.3
MDRR AVG WT	117.7	113.7	101.7	91.8
RR AVG TAT	142.8	137.9	134.7	118.7
MDRR AVG TAT	138.1	134.1	122.1	112.2

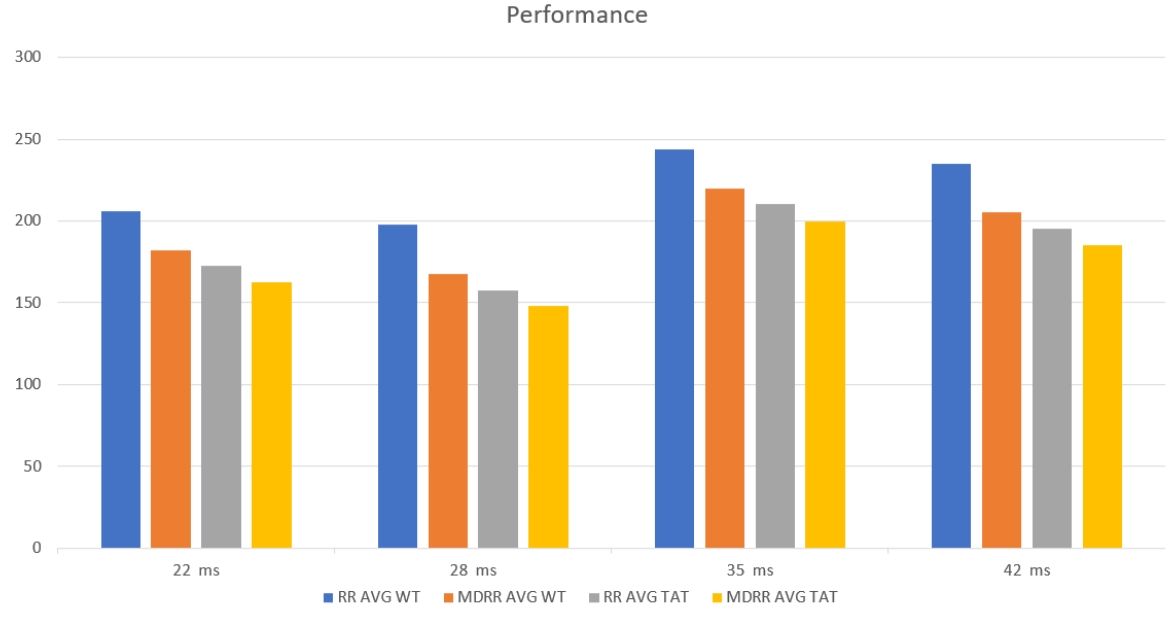


Test Case 2:

Quantum is 22, 28, 35, 42 milliseconds

Process Name	Burst Time
P1	20
P2	25
P3	15
P4	30
P5	37
P6	45
P7	51
P8	60
P9	27
P10	66

Result of test case one: Quantum	22 ms	28 ms	35 ms	42 ms
RR AVG WT	205.9	182	172.5	162.3
MDRR AVG WT	197.5	167.5	157.7	147.9
RR AVG TAT	243.5	219.6	210.1	199.9
MDRR AVG TAT	235.1	205.1	195.3	185.5



Test case 3:
Quantum is 20, 50, 80, 100 milliseconds

Process Name	Burst Time
P1	550
P2	1250
P3	1950
P4	50
P5	500
P6	1200
P7	100

Result of test case one: Quantum	20 ms	50 ms	80 ms	100 ms
RR AVG WT	2305.71	2257.14	2330	2271.43
MDRR AVG WT	2291.43	2221.43	2208.57	2121.43
RR AVG TAT	3105.71	3057.14	3130	3071.43
MDRR AVG TAT	3091.43	3021.43	3008.57	2921.43

The Proposed algorithm performance compared to the Standard Round Robin algorithm using a variety of cases, four of which are described in this report. When the Proposed algorithm experimental results were compared to the existing standard scheduling algorithm, it was discovered that the proposed Mean-Difference Round Robin Algorithm achieved more optimal scheduling.

Conclusion:

In this research, we introduce the Mean-Difference Round Robin Technique, a unique CPU scheduling algorithm aimed at optimizing CPU scheduling for real-time applications. The suggested algorithm determines the average burst time for all processes in the ready queue. The difference between a process burst time and the computed average burst time is then determined. This procedure is carried out for all processes in the ready queue. The suggested technique then determines which process has the biggest difference value and gives it to the CPU to run for one time slice. When the process's time slice expires, the next process in the ready queue with the biggest difference value is selected and performed for one time slice. All of the processes in the ready queue go through the same procedure. The suggested approach is compared to the Standard Round Robin algorithm using a variety of instances, two of which are described in this work. When the suggested algorithm's experimental results were compared to those of existing standard scheduling algorithms, it was discovered that the proposed Mean-Difference Round Robin Algorithm achieved more optimal scheduling.

References:

[1] Harary, F., & Moser, L. (1966). The theory of round robin tournaments. *The American Mathematical Monthly, 73*(3), 231–246. https://doi.org/10.1080/00029890.1966.11970749

[2] Silberchatz, Galvin and Gagne, 2003. Operating systems concepts.

[3] G Siva Nageswara Rao, A NEW PROPOSED DYNAMIC DUAL PROCESSOR BASEDCPU SCHEDULING ALGORITHM, Journal of Theoretical and Applied Information Technology 20th March 2015. Vol.73 No.2,ISSN: 1992-8645.

[4] S. C. Satapathy, P. S. Avadhani, S. K. Udgata, and S. Lakshminarayana, ICT and Critical Infrastructure: Proceedings of the 48th annual Convention of Computer Society of India-Vol I hosted by CSI vishakapatnam chapter. Cham: Springer International Publishing, 2014.

[5] Wikimedia Foundation. (2014, February 12). *Scheduling*. Wikipedia. Retrieved January 2,2022, from http://en.wikipedia.org./wiki/scheduling