



UNIVERSITY MALAYSIA TERENGGANU

CSF3305

OPERATING SYSTEM

Round Robin

Author:

Maharaj Faawwaz A Yusran

Ayu Nabilah Binti Rozani

Nur Hajidah Iffah Binti Abdul Rahim

Subhashini Kannan

Matric Number:

S52500

S51356

S50947

S52014

Contents

| | | |
|----------|---|-----------|
| 1 | Introduction | 2 |
| 2 | Definition, Advantages and Disadvantages | 3 |
| 3 | Source Code | 4 |
| 4 | Output and Discussion | 9 |
| 4.1 | Same Arrival Time | 9 |
| 4.2 | Different Arrival Times | 10 |
| 5 | Conclusion | 11 |

1 Introduction

An operating system needs a *program scheduler* to schedule processes to run on the computer. There are three types of schedulers: long term, mid-term and short term. Moreover, an operating system uses two types of scheduling process executions which are preemptive and non-preemptive. In preemptive scheduling policy, a low priority process has to be suspended during its execution if a higher priority process is waiting in the same queue for its execution. In non-preemptive scheduling policy, processes are executed in first come first serve basis, which means the next process is executed only when the current running process finishes its execution.

There are a variety of ways to schedule processes that should run on the computer, called scheduling algorithms. These scheduling algorithms are: First Come First Serve (FCFS), Priority-based Scheduling, Shortest Job First (SJF), Longest Job First (LJF), Shortest Remaining Time First (SRTF), Highest Response Ratio Next and last but not least, the chosen topic of discussion: Round Robin (RR).

To measure the performance of the algorithms, there are two main variables are taken into account: *turnaround time* and *waiting time*, that are calculated based on each process's *arrival*, *burst* and *completion* time.

Arrival time is time at which the process arrives in the ready queue. Completion time is time at which process complete its execution.

Completion time is time at which process complete its execution.

Burst time is time required by a process for CPU execution.

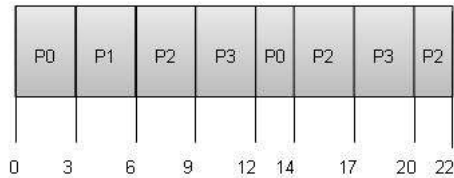
To calculate *turnaround time* and *waiting time*, the following formulas are used:

$$TurnAroundTime = CompletionTime - ArrivalTime$$

$$WaitingTime = TurnAroundTime - BurstTime$$

2 Definition, Advantages and Disadvantages

Quantum = 3



The Round Robin scheduling is simple, easy to implement, and starvation-free as all processes get fair share of CPU. It is particularly effective in a general-purpose time-sharing system or transaction processing system. It is also one of the most commonly used technique in CPU scheduling as a core.

The advantages of Round Robin scheduling is all the processes have the equal priority because of fixed time quantum. Starvation will never occur because each process in every Round Robin scheduling cycle will be schedule for a fixed time slice or time quantum.

The disadvantages of it is more overhead of context switching. In the Round Robin scheduling algorithm, as the time quantum decreases context switching increases. The increases in time quantum value results in time starvation which may put many processes on hold. If the time quantum decreases, it will affect the CPU efficiency. So, time quantum should neither be large nor small. If time quantum becomes infinity, Round Robin scheduling algorithm gradually become *First Come First Serve* (FCFS) scheduling algorithm.

3 Source Code

To understand the Round Robin scheduling algorithm, we have made a small command-line project written in *Java*.

The Java application takes in a list of processes and their respective *burst time* and *arrival time* that is keyed in by the user, and computes the order that processes should be executed in, by using the RR algorithm. Next, the program calculates the *turnarund time* and *waiting time* for each process and displays them in the standard output.

Firstly, we create a **Process** class which contains all the attributes of a process running on the computer. Using the object-oriented way, we can store an array of process objects later on in the main program. **Process.java** is as follows:

```
public class Process {

    private final int burstTime;
    private int remainingBurstTime, arrivalTime, timeArrivedInQueue,
        finishedTime;

    public Process(int burstTime, int arrivalTime) {
        this.burstTime = burstTime;
        this.remainingBurstTime = burstTime;
        this.arrivalTime = arrivalTime;
        this.finishedTime = 0;
    }

    public int getBurstTime() {
        return burstTime;
    }

    public int getRmBurstTime() {
        return remainingBurstTime;
    }

    public void setRmBurstTime(int burstTime) {
        this.remainingBurstTime = burstTime;
    }

    public void decreaseBurstTime(int quantum) {
        this.remainingBurstTime -= quantum;
    }
}
```

```
public int getArrivalTime() {
    return arrivalTime;
}

public void setArrivalTime(int arrivalTime) {
    this.arrivalTime = arrivalTime;
}

public void setTimeArrivedInQueue(int t) {
    timeArrivedInQueue = t;
}

public void setFinishedTime(int finishedTime) {
    // only set the completion time if it's not already been set
    if (this.finishedTime == 0) {
        this.finishedTime = finishedTime;
    }
}

public int getTurnaroundTime() {
    // Using the formula
    return (int) Math.abs(finishedTime - timeArrivedInQueue);
}

public int getWaitingTime() {
    // Using the formula
    return getTurnaroundTime() - burstTime;
}
}
```

Now we see the algorithm in action at **RoundRobinJava.java**:

```
import java.util.Scanner;

public class RoundRobinJava {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        // Get input from user
        System.out.print("Number of processes: ");
        int numberOfProcesses = sc.nextInt();
        System.out.print("Quantum: ");
    }
}
```

```
int quantum = sc.nextInt();

// Creates an empty array of processes to be filled in later
Process[] processes = new Process[numberOfProcesses];

// Initialize total time taken for all processes to be executed
int totalTime = 0;

// Get arrival times and burst times from user, loop through each
// process
for (int i = 0; i < numberOfProcesses; i++) {
    System.out.print("Arrival time for P" + i + " (lowest 0): ");
    int arrivalTime = sc.nextInt();
    System.out.print("Burst time for P" + i + " (lowest 0) : ");
    int burstTime = sc.nextInt();

    // Sets the ith process based on user input
    processes[i] = new Process(burstTime, arrivalTime);

    totalTime += processes[i].getBurstTime();
}

sc.close();

// Displays initial data
System.out.println("Process Num\t| Arrival\t| Burst");
for (int i = 0; i < numberOfProcesses; i++) {
    System.out.println(displayProcessDetails(i,
        processes[i].getArrivalTime(),
        processes[i].getRmBurstTime()));
}
System.out.println();

// Store waiting and turnaround times
int[] waitingTimes = new int[numberOfProcesses];
int[] turnaroundTimes = new int[numberOfProcesses];

// CPU begins executing processes
int time = 0;
while (time < totalTime) {
    // loop through each process, check arrival and burst
    for (int num = 0; num < processes.length; num++) {
        // Check to see if the current process has arrived
```

```
if (processes[num].getArrivalTime() <= time) {
    // Check that it still has remaining burst time
    if (processes[num].getRmBurstTime() >= quantum) {
        // Check if the process is executed for the first time
        // (burst time isn't decreased yet)
        if (processes[num].getRmBurstTime() ==
            processes[num].getBurstTime()) {
            // If yes, store it
            processes[num].setTimeArrivedInQueue(time);
        }
        // Add process to timeline
        printProcess(num);

        // Decrease current process's burst time by quantum
        processes[num].decreaseBurstTime(quantum);

        if (processes[num].getRmBurstTime() == 0) {
            // No burst left, set completion time
            processes[num].setFinishedTime(time);
            // calculate waiting and turnaround time
            turnaroundTimes[num] =
                processes[num].getTurnaroundTime();
            waitingTimes[num] = processes[num].getWaitingTime();
        }
    }
    // 1 burst finished, move on
    time += quantum;
}

// Display output
System.out.println("\n\nTotal time: " + totalTime + "s\n");

int totalWaiting = 0, totalTurnaround = 0;
// Display turnaround and waiting times
System.out.println("Process Num\t| Arrival\t| Burst \t| Waiting
    time \t| Turnaround time");
for (int i = 0; i < numberOfProcesses; i++) {
    System.out.println(displayProcessResults(i,
        processes[i].getArrivalTime(), processes[i].getBurstTime(),
        waitingTimes[i], turnaroundTimes[i]));
}
```



```

        // Also sum up the times, to calculate average
        totalWaiting += waitingTimes[i];
        totalTurnaround += turnaroundTimes[i];
    }
    // Calculate and show the averages
    double avgWaitingTime = totalWaiting / numberOfProcesses;
    double avgTurnaroundTime = totalTurnaround / numberOfProcesses;
    System.out.println();
    System.out.printf("Average waiting time: %.2fs\n", avgWaitingTime);
    System.out.printf("Average turnaround time: %.2fs\n",
        avgTurnaroundTime);
}
/**
 * Plot the process to the ordered timeline
 */
private static void printProcess(int processIndex) {
    System.out.print("P" + processIndex + " - ");
}
/**
 * Display a process's details in table format
 *
 * @return process details as a String
 */
private static String displayProcessDetails(int i, int arrival, int
    burst) {
    return "P" + i + "\t\t| " + arrival + "\t\t| " + burst;
}
/**
 * Display a process's waiting and turnaround time in a table format
 *
 * @param i          the process number
 * @param arrival    arrival time of the process
 * @param burst      burst time of the process
 * @param waiting    waiting time of the process
 * @param turnaround turnaround time
 * @return process details as a String
 */
private static String displayProcessResults(int i, int arrival, int
    burst, int waiting, int turnaround) {
    return displayProcessDetails(i, arrival, burst) + "\t\t| " +
        waiting + "\t\t| " + turnaround;
}
}

```

4 Output and Discussion

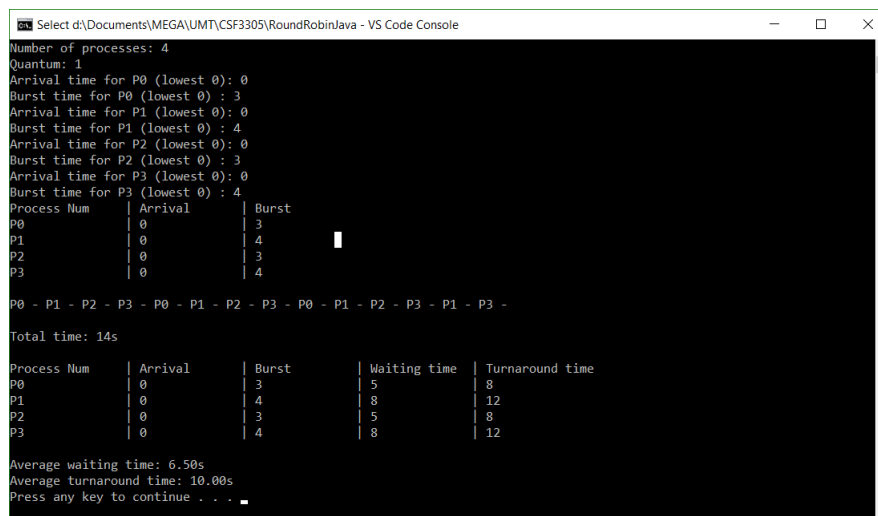
4.1 Same Arrival Time

Input:

Number of processes = 4

quantum = 1

Arrival time = 0



```
Select d:\Documents\MEGA\UMT\CSF3305\RoundRobinJava - VS Code Console
Number of processes: 4
Quantum: 1
Arrival time for P0 (lowest 0): 0
Burst time for P0 (lowest 0): 3
Arrival time for P1 (lowest 0): 0
Burst time for P1 (lowest 0): 4
Arrival time for P2 (lowest 0): 0
Burst time for P2 (lowest 0): 3
Arrival time for P3 (lowest 0): 0
Burst time for P3 (lowest 0): 4
Process Num | Arrival | Burst
P0          | 0       | 3
P1          | 0       | 4
P2          | 0       | 3
P3          | 0       | 4
P0 - P1 - P2 - P3 - P0 - P1 - P2 - P3 - P0 - P1 - P2 - P3 - P1 - P3 -
Total time: 14s
Process Num | Arrival | Burst | Waiting time | Turnaround time
P0          | 0       | 3     | 5            | 8
P1          | 0       | 4     | 8            | 12
P2          | 0       | 3     | 5            | 8
P3          | 0       | 4     | 8            | 12
Average waiting time: 6.50s
Average turnaround time: 10.00s
Press any key to continue . . .
```

Based on the results above, we can see that the Round Robin scheduling is preemptive, assigning the CPU a part of a process's burst time for a fixed amount of time. The fixed amount of time may also be called *time slice* or *time quantum*. It is essentially a First Come First Serve algorithm, with a fixed *time quantum*, and after the time quantum finishes, the assigned process is preempted and sent back to the queue. Each iteration, the process is moved up in the queue, and ready to be called again after 1 round (hence the name of the algorithm).

With the same arrival time, the order that the processes are executed is circular, starting from P0 to P3 if there are 4 processes to run. When a process finishes its current time quantum, the remaining burst time is decreased. When the process does not have anymore remaning burst time (or the process is finished), the algorithm skips that process.

The average waiting time for processes that have same arrival time is consistent. But also see that the order is heavily based on the time quantum. So if the processes have much higher burst time, it will lead to starvation because they have to wait longer.

4.2 Different Arrival Times

Input:

Number of processes = 3

quantum = 1

Arrival time P0 = 0

Arrival time P1 = 2

Arrival time P2 = 1

Burst time P0 = 4

Burst time P1 = 3

Burst time P2 = 3

```
d:\Documents\MEGA\UMT\CSF3305\RoundRobinJava - VS Code Console
Number of processes: 3
Quantum: 1
Arrival time for P0 (lowest 0): 0
Burst time for P0 (lowest 0): 4
Arrival time for P1 (lowest 0): 2
Burst time for P1 (lowest 0): 3
Arrival time for P2 (lowest 0): 1
Burst time for P2 (lowest 0): 3
Process Num | Arrival | Burst
P0          | 0       | 4
P1          | 2       | 3
P2          | 1       | 3
P0 - P2 - P0 - P1 - P2 - P0 - P1 - P2 - P0 - P1 -
Total time: 10s
Process Num | Arrival | Burst | Waiting time | Turnaround time
P0          | 0       | 4     | 4             | 8
P1          | 2       | 3     | 3             | 6
P2          | 1       | 3     | 3             | 6
Average waiting time: 3.33s
Average turnaround time: 6.67s
Press any key to continue . . .
```

With different arrival times, the processes continue to circulate. At quantum = 1, it skips P1 as P1 has not arrived to the queue. After completing 1 round, the flow restarts and the processor is assigned P0 again.

The waiting times for the processes are consistent also, like the previous test.

5 Conclusion

To sum up, in Round Robin Scheduling the time quantum is fixed and then processes are scheduled such that no process get CPU time more than one time quantum in one go. If time quantum is too large, the response time of the processes gets way higher, which may not be tolerated in interactive environment, and it will become a FCFS scheduling instead. If the time quantum is too small, it causes too many context switching, leading to more overhead. Overall, Round Robin algorithm is best used in time-sharing systems and interactive systems.

References

Tutorialspoint.com (2019). Operating System Scheduling algorithms. Retrieved 5 May, 2019, from <https://www.tutorialspoint.com>

W3schools (2019). Scheduling Algorithms of Operating System. Retrieved 12 May, 2019, from <https://www.w3schools.in>

Gatevidyalay.com (2019). Round Robin — Round Robin Scheduling — Examples. Retrieved 11 May, 2019, from <https://www.gatevidyalay.com>

GeeksforGeeks (2019). Program for Round Robin scheduling — Set 1. Retrieved 3 May, 2019, from <https://www.geeksforgeeks.org>

GeeksforGeeks (2019). Round Robin Scheduling with different arrival times. Retrieved 3 May, 2019, from <https://www.geeksforgeeks.org>