



University of New York Tirana

Faculty of Engineering & Architecture

Department of Computer Science

Simulation of CPU Scheduling Algorithms

Course: Operating Systems

Instructor: Kristi Gorea, MSc.

Date: 19 January 2024

Semester: Fall 2023

Prepared by: Sofi Çapi

List of Graphics

Graphic 3.1: Average Waiting Time for All Iterations and Algorithms	17
Graphic 3.2: Average Waiting Time for All Iterations and for the FCFS Algorithm	17
Graphic 3.3: Average Waiting Time for All Iterations and for the SJF Algorithm.....	18
Graphic 3.4: Average Waiting Time for All Iterations and for the RR Algorithm	18

List of Figures

Figure 2.1: Method calculateRoundRobin (Part I)	5
Figure 2.2: Method calculateRoundRobin (Part II)	5
Figure 2.3: Method calculateFirstComeFirstServeAndShortestJobFirst	6
Figure 2.4: Method showTurnaroundAndWaitingTime (Part I)	7
Figure 2.5: Method showTurnaroundAndWaitingTime (Part II)	8
Figure 2.6: Method generateInputTask1	9
Figure 2.7: Method generateInputTask2	10
Figure 2.8: Method main (Part I)	11
Figure 2.9: Method main (Part II)	11
Figure 2.10: Output of Task I (Part I)	12
Figure 2.11: Output of Task I (Part II)	13
Figure 2.12: Output of Task I (Part III)	13
Figure 3.1: Output of Task II (Part I)	14
Figure 3.2: Output of Task II (Part II)	15

List of Tables

Table 3.1: Average Waiting Time for 20 th Iterations and for All Algorithms	16
---	----

Table of Content

List of Graphics.....	ii
List of Figures.....	iii
List of Tables.....	iv
Table of Content	v
1. Introduction	1
2. Task I.....	2
2.1. Process Class	2
2.2. CPUScheduling Class	3
2.3. Main Class.....	8
2.4. Output Result of Task I	12
3. Task II.....	14
3.1. Output Result of Task II	14
3.2. Result Interpretation	16

1. Introduction

The project's objectives are divided into two important tasks. The first task requires implementing the three most known CPU scheduling algorithms, which are First Come First Serve (FCFS), Shortest Job First (SJF), and Round Robin (RR) with a quantum of 5-time units. After calculating these algorithms, the console should print out the order of execution of the processes, the turnaround time, and waiting time of every process, and the average turnaround and waiting time.

Furthermore, the second task asks for a simulation of three scheduling algorithms (FCFS, SJF, RR) with randomly generated process data, and then measuring the average waiting times for different burst time ranges. After running multiple simulations, you want to visualize the results in bar charts and analyze if there is any trend as the burst time range widens. Hence, the goal is to systematically vary burst time ranges, run simulations, and analyze the average waiting times for each scheduling algorithm. Afterward, the visualization will help to easily compare the performance of FCFS, SJF, and RR under different conditions and identify any trends related to the burst time range.

Regarding the code implementation, I used Java programming language, and to not have redundant code, I did not use different methods for each task, but I interweaved them in the same methods, and to differentiate them from each other, I used ternary operations to show the order of execution, the waiting and turnaround time for every execution, and the average turnaround time on the screen if the task number is 1, in contrast, it will not show anything.

2. Task I

In the first task, we needed to deal with the implementation of the three most known CPU scheduling algorithms, which are First Come First Serve, Shortest Job First, and Round Robin with a quantum of 5-time units. Attached to its implementation, we needed to display the turnaround time, and waiting time of each process, and at the end, the screen should show the average turnaround, and waiting time of the overall processes for every algorithm that will be implemented.

The programming language used in this project to implement these three algorithms was Java. The total number of classes used for the proper implementation and execution of the FCFS, SJF, and RR was three, which are named as follows: Main, CPUScheduling, and Process Class.

2.1. Process Class

First, the Process class contains five attributes, which are name, burstTime, turnaroundTime, waitingTime, and remainingTime. The first attribute, name, represents the name of the process, and it is of type String. Then, it is followed by the burstTime attribute, which represents the time required for the process to complete, and the data type is integer. Additionally, we have the turnaroundTime attribute of type integer, which refers to the time interval from the time of submission of a process towards the time of the finishing of the process, while the fourth attribute, the waitingTime, is defined as the total time that is spent by the process while staying in a ready queue before it reaches the CPU, and the data type is an integer. Lastly, we have the remainingTime, whose data type is the same as the last three attributes,

integer data type. So, as the name suggests, it represents the remaining time needed for the process to complete.

The other components of this class are constructors and methods. In this code, we have only one constructor, that initializes a *Process* object with a given name and burst time, and sets *turnaroundTime* and *waitingTime* to 0 initially, while the *remainingTime* is set to the initial *burstTime*. To continue with the second component, we have two types of methods, setters and getters, that respectively allow modification of the attributes, and retrieving the value of these attributes.

Overall, the *Process* class is used to create instances representing individual processes in CPU scheduling, and objects of this class are created in the *Main* class, and are manipulated within the *CPUScheduling* class to calculate the turnaround and waiting times

2.2. CPUScheduling Class

Secondly, we have the **CPUScheduling** class that does not contain any attribute or user-defined constructor. The only component that this class has are three methods, which perform CPU scheduling calculations needed for different algorithms (FCFS, SJF, and RR). This class interacts with a list of *Process* objects, manipulating their attributes to calculate the turnaround and waiting times, and it displays the result on the screen. Overall, this class promotes code modularity by separating the calculation logic for different scheduling algorithms.

Initially, we have the method named *calculateRoundRobin*, which takes two parameters, an *ArrayList* of *Process* objects and the task number. The data structure used here is a queue, which is FIFO (First-In First-Out), and all the objects of the *ArrayList* named processes are

passed to the queue in the order that they are inserted (the first *Process* object in the *ArrayList* is the first object in the queue). Before going to the loop, we declare and initialize an integer variable called *currentTurnaroundTime* with a value of 0, which will be used to modify the turnaround time of the given process.

After the initialization, we use the while loop with the condition that the queue is not empty, and we poll from the queue the first object and save it to a temporary *Process* object named *currentProcess*. Furthermore, we check if the *currentProcess* references null or not. If it references the null value, it goes to the next iteration without executing the other part of the block for the current iteration. Otherwise, if it references an object, we retrieve the minimum quantum time unit between the pre-defined quantum time unit with a value of 5 and the current remaining time of the process. Then, we sum the value of the *currentTurnaroundTime* with the minimum quantum and save the result to *currentTurnaroundTime*, and this result will be added to the current process turnaround time attribute, while the new remaining time of this process will be the difference between the current remaining time and the minimum quantum.

Continuing with the last part of the while loop, we check that there is a simple if condition that checks if the current remaining time of the process is different from 0, if it evaluates to true, then we add the current process to the queue, otherwise we calculate the waiting time of the process with a simple formula that is the difference between the turnaround time and the burst time of the process. This marks the end of the loop, and the next line is the last line of the method, and it calls another method, *showTurnaroundAndWaitingTime*. An illustration of the implementation of the elaborated method is shown in the both figures, *Figure 2-1 & Figure 2-2*.

```

// This method calculates the CPU Scheduling algorithm Round Robin with quantum of 5 with the turnaround and waiting time
2.usages new*
public static void calculateRoundRobin(ArrayList<Process> processes, int taskNumber) {
    // This line of code is creating a new LinkedList and initializing it with the elements of an existing collection,
    // in this case, a Queue named processes.
    Queue<Process> processesInQueue = new LinkedList<>(processes);

    int currentTurnaroundTime = 0;

    // The ternary operation is used to print on the screen the line if the taskNumber is 1
    System.out.print(taskNumber == 1 ? "Order of Execution is: " : "");

    // It will execute the while loop if the queue is not empty
    while (!processesInQueue.isEmpty()) {
        // It retrieved and poll the first element in the queue
        Process currentProcess = processesInQueue.poll();

        // If the currentProcess references to null, it will continue the loop
        if (currentProcess == null)
            continue;

        // It finds the minimum quantum between the remaining time of the process, and number 5
        int minQuantum = Math.min(currentProcess.getRemainingTime(), 5);

        // We set the value of the new turnaround and remaining time
        currentTurnaroundTime += minQuantum;
        currentProcess.setTurnaroundTime(currentTurnaroundTime);
        currentProcess.setRemainingTime(currentProcess.getRemainingTime() - minQuantum);
    }
}

```

Figure 2-1: Method calculateRoundRobin (Part I)

```

// We use ternary operation to show the order of execution of the processes if and only if the taskNumber is 1
System.out.print(taskNumber == 1 ? currentProcess.getName() + "(" + minQuantum + ")" +
    (processesInQueue.isEmpty() && currentProcess.getRemainingTime() == 0 ? " " + "\n": ", ") : "");

// If the remaining time of the currentProcess is different from 0, it will add the process at the end of the queue.
// Otherwise, it will calculate the waiting time.
if (currentProcess.getRemainingTime() != 0)
    processesInQueue.add(currentProcess);
else
    currentProcess.setWaitingTime(currentTurnaroundTime - currentProcess.getBurstTime());
}

// After the loop ends, it will call the method below
showTurnaroundAndWaitingTime(processes, taskNumber);
}

```

Figure 2-2: Method calculateRoundRobin (Part II)

Next, we have the `calculateFirstComeFirstServeAndShortestJobFirst` method, which takes three parameters, which are an `ArrayList` of `Process`, the task number, and the algorithm name that will be implemented, FCFS or SJF. The difference between this method and the previous method is not huge, and there are three major modifications between them.

The first one is that this method does not require the usage of the queue, nevertheless, we create a copy of the `ArrayList` retrieved by the first parameter of the current method, and if the algorithm name is “SJF”, we sort it in ascending order by their burst time. Consequently, the loop that we are using here is the for-each loop, which iterates over all the `Process` objects and we use the `copyOfProcesses`. Secondly, instead of using the minimum quantum, we use the current burst time, and the last point is that the last if condition located at the end of the while loop of the previous method is removed and replaced only by the calculation of the waiting time.

The implementation of this method is represented in the *Figure 2-3*

```
// It calculates the turnaround and waiting time of the First Come First Serve and Shortest Job First algorithms
4 usages new *
public static void calculateFirstComeFirstServeAndShortestJobFirst(ArrayList<Process> processes, int taskNumber, String algorithm) {
    int currentTurnaroundTime = 0;
    ArrayList<Process> copyOfProcesses = new ArrayList<>(processes); // We create a copy of the arrayList processes, which is named copyOfProcesses

    if(algorithm.equals("SJF"))
        copyOfProcesses.sort(Comparator.comparingInt(Process::getBurstTime)); // We sort the copyOfProcesses in ascending order, based on the Burst Time

    // The ternary operation is used to print on the screen the line if the taskNumber is 1
    System.out.print(taskNumber == 1 ? "Order of Execution is: " : "");
}

for (Process currentProcess : copyOfProcesses) { // It uses a For-Each loop
    if (currentProcess == null) // If the currentProcess references to null, it will continue the loop
        continue;

    int currentBurstTime = currentProcess.getBurstTime(); // It retrieves the current burst time of the current process

    // It calculates the turnaround time, and remaining time
    currentTurnaroundTime += currentBurstTime;
    currentProcess.setTurnaroundTime(currentTurnaroundTime);
    currentProcess.setRemainingTime(currentProcess.getRemainingTime() - currentTurnaroundTime);

    // We use ternary operation to show the order of execution of the processes if and only if the taskNumber is 1
    System.out.print(taskNumber == 1 ? currentProcess.getName() + "(" + currentBurstTime +
        (currentProcess == copyOfProcesses.get(copyOfProcesses.size() - 1) ? ")" : "\n" : ", ") : "");

    currentProcess.setWaitingTime(currentProcess.getTurnaroundTime() - currentBurstTime); // It calculates the waiting time
}
showTurnaroundAndWaitingTime(processes, taskNumber); // It will call this method
}
```

Figure 2-3: Method `calculateFirstComeFirstServeAndShortestJobFirst`

The last method of this class is *showTurnaroundAndWaitingTime*, which also takes two parameters described in the first method. In the following lines, it will be written a summary of its implementation. This method contains two for-each loops that iterate all over the ArrayList, and the first for-each loop shows on the screen the waiting time of every process if the task that the user is running is 1, and it calculates the sum of all waiting times of the processes found in the ArrayList. After finishing the execution of the first for-each loop, it goes to the second for-each loop that displays on the screen the turnaround time of every process in the ArrayList if the task number is equal to 1, and it calculates the sum of all turnaround times of the processes. Furthermore, we have only 2 more calculations left, which are the average turnaround time, and average waiting time that are part of the last line of the code. The implementation is shown in the following figures, *Figure 2-4 & Figure 2-5*.

```
// It shows the turnaround and waiting time of all processes on the screen, and
// it calculates and shows the average turnaround and waiting time
2 usages new *
public static void showTurnaroundAndWaitingTime(ArrayList<Process> processes, int taskNumber) {
    int numberOfProcesses = processes.size();
    int totalTurnaroundTime = 0;
    int totalWaitingTime = 0;

    // The ternary operation is used to print on the screen the line if the taskNumber is 1
    System.out.print(taskNumber == 1 ? ""

        Waiting Times:\s
        "" : "");

    // It uses a For-Each loop
    for (Process currentProcess : processes) {
        // If the currentProcess references to null, it will continue the loop
        if (currentProcess == null)
            continue;

        // We use ternary operation to show the waiting time of the process if and only if the taskNumber is 1
        System.out.print(taskNumber == 1 ? currentProcess.getName() + " = " + currentProcess.getWaitingTime() + "\n" : "");

        // It calculates the total waiting time
        totalWaitingTime += currentProcess.getWaitingTime();
    }
}
```

Figure 2-4: Method showTurnaroundAndWaitingTime (Part I)

```

// The ternary operation is used to print on the screen the line if the taskNumber is 1
System.out.print(taskNumber == 1 ? ""

    Turnaround Times:\s
    "" : "");

// It uses a For-Each loop
for (Process currentProcess : processes) {
    // If the currentProcess references to null, it will continue the loop
    if (currentProcess == null)
        continue;

    // We use ternary operation to show the turnaround time of the process if and only if the taskNumber is 1
    System.out.print(taskNumber == 1 ? currentProcess.getName() + " = " + currentProcess.getTurnaroundTime() + "\n" : "");

    // It calculates the total waiting time
    totalTurnaroundTime += currentProcess.getTurnaroundTime();
}

// It calculates the average waiting time and average turnaround time if the number of process is greater than 0
double averageWaitingTime = numberOfProcesses > 0 ? (double) totalWaitingTime / numberOfProcesses : 0;
double averageTurnaroundTime = numberOfProcesses > 0 ? (double) totalTurnaroundTime / numberOfProcesses : 0;

System.out.print(taskNumber == 1 ? "\n" : "");
// It shows the average waiting time
System.out.println("Average Waiting Time = " + String.format("%.2f", averageWaitingTime));
// We use ternary operation to show the average turnaround time of the process if and only if the taskNumber is 1
System.out.print(taskNumber == 1 ? "Average Turnaround Time = " + String.format("%.2f", averageTurnaroundTime) + "\n\n" : "");
}

```

Figure 2-5: Method showTurnaroundAndWaitingTime (Part II)

2.3. Main Class

The last class is the **Main** class which serves as the entry point for the program, handling user input, executing tasks, and generating processes, and it provides a clear and structured flow for the program. It contains three constants called `reset`, `cyan`, and `green` that are used to add color to the output for better visualization.

The class contains three methods, which are *generateInputTask1*, *generateInputTask2*, and *the main* method. The first method listed in the previous sentence is executed only for the first task, and it does not have any parameters. The purpose of this method is to return an `ArrayList` that contains some *Process* objects. A clear representation of the code is shown in *Figure 2-6*.

```

// This method is used to return an arrayList that contains Process objects that will be used for the Task 1
3 usages new *
public static ArrayList<Process> generateInputTask1() {
    ArrayList<Process> arrayList = new ArrayList<>();

    arrayList.add(new Process( name: "P1", burstTime: 7));
    arrayList.add(new Process( name: "P2", burstTime: 8));
    arrayList.add(new Process( name: "P3", burstTime: 2));
    arrayList.add(new Process( name: "P4", burstTime: 6));

    return arrayList;
}

```

Figure 2-6: Method generateInputTask1

Secondly, we have the other method, *generateInputTask2*, that requires three parameters. The parameters that are implemented in this method are *numberOfProcesses*, *minBurstTime*, and *maxBurstTime*. We create three ArrayLists of type *Process* and then in the for loop, we create different processes with random integer burst time between the minimum and maximum burst time, and these processes are added to the ArrayLists. When the execution of the for-loop finishes, it calls three other methods that are part of the *CPUScheduling* class *calculateFirstComeFirstServe*, *calculateShortestJobFirst*, and *calculateRoundRobin*. Figure 2-7 represents the Java implementation of this method.

```

/* This method will generate the input that will be needed for the Task 2, which has three parameters:
   int numberOfProcesses --> The number of processes
   int minBurstTime --> The minimum burst time
   int maxBurstTime --> The maximum burst time
*/
1 usage new *
public static void generateInputTask2(int numberOfProcesses, int minBurstTime, int maxBurstTime) {
    // Created three ArrayList that will be used to store the randomly generated processes. A copy for each of the algorithm.
    ArrayList<Process> processesFCFS = new ArrayList<>();
    ArrayList<Process> processesSJF = new ArrayList<>();
    ArrayList<Process> processesRR = new ArrayList<>();

    // Created an object of the Random class
    Random random = new Random();

    // Creating numberOfProcesses Process objects
    for (int i = 1; i <= numberOfProcesses; i++) {
        int burstTime = random.nextInt( bound: (maxBurstTime - minBurstTime) + 1) + minBurstTime;
        processesFCFS.add(new Process( name: "P" + i, burstTime));
        processesSJF.add(new Process( name: "P" + i, burstTime));
        processesRR.add(new Process( name: "P" + i, burstTime));
    }

    // Calculating three algorithm FCFS, SJF, and RR
    System.out.println(cyan + "***** First Come First Serve *****" + reset);
    CPUScheduling.calculateFirstComeFirstServeAndShortestJobFirst(processesFCFS, taskNumber: 2, algorithm: "FCFS");
    System.out.println(cyan + "***** Shortest Job First *****" + reset);
    CPUScheduling.calculateFirstComeFirstServeAndShortestJobFirst(processesSJF, taskNumber: 2, algorithm: "SJF");
    System.out.println(cyan + "***** Round Robin *****" + reset);
    CPUScheduling.calculateRoundRobin(processesRR, taskNumber: 2);
}

```

Figure 2-7: Method generateInputTask2

Finally, we have the main method, where the code starts to execute. Initially, it asks for the user to enter the task number, 1 or 2. If the task number is neither 1 nor 2, it will keep asking until he/she enters the correct input. After entering either 1 or 2, it will go to the if statement that will verify if the entered task number is 1, and if it evaluates to true, it will start executing the lines inside that if statement. Otherwise, it will execute the else statement, and it starts by asking the user for the number of processes, number of simulations, minimum burst time, and maximum burst time. Lastly, it will calculate the three algorithms. A code implementation is shown in both figures, *Figure 2-8 & Figure 2-9*.

```

public static void main(String[] args) {
    // This line creates a Scanner object named scanner that reads user input from the keyboard.
    Scanner scanner = new Scanner(System.in);

    System.out.print("Which Task Do You Want to Run? (1 or 2): "); // It asks the users which task they want to execute
    int taskNumber = scanner.nextInt();

    // If the taskNumber is not 1, and it is not 2, it will continue asking the user to input the right taskNumber
    while (taskNumber != 1 && taskNumber != 2) {
        System.out.print("Wrong Input! Which Task Do You Want to Run? (1 or 2) ");
        taskNumber = scanner.nextInt();
    }

    // If the task number is 1, it will execute the first part, otherwise it will execute the block of code inside the else statement
    // which refers to the taskNumber 2
    if (taskNumber == 1) {
        // It will call three methods to calculate FCFS, SJF, and RR with quantum of 5
        System.out.println(cyan + "***** First Come First Serve *****" + reset);
        CPUScheduling.calculateFirstComeFirstServeAndShortestJobFirst(generateInputTask1(), taskNumber: 1, algorithm: "FCFS");
        System.out.println(cyan + "***** Shortest Job First *****" + reset);
        CPUScheduling.calculateFirstComeFirstServeAndShortestJobFirst(generateInputTask1(), taskNumber: 1, algorithm: "SJF");
        System.out.println(cyan + "***** Round Robin *****" + reset);
        CPUScheduling.calculateRoundRobin(generateInputTask1(), taskNumber: 1);
    } else {

```

Figure 2-8: Method main (Part I)

```

    } else {
        // Input number of processes
        System.out.print("Enter the number of processes: ");
        int numberOfProcesses = scanner.nextInt();
        // Input number of simulations
        System.out.print("Enter the number of simulations: ");
        int numberOfSimulations = scanner.nextInt();
        // Input burst time range
        System.out.print("Enter the minimum burst time: ");
        int minBurstTime = scanner.nextInt();
        System.out.print("Enter the maximum burst time: ");
        int maxBurstTime = scanner.nextInt();

        int i = 1;
        while (i <= numberOfSimulations) {
            System.out.println(green + "***** Iteration Number " + i + " *****" + reset);
            generateInputTask2(numberOfProcesses, minBurstTime, maxBurstTime);
            i++;
        }
    }
    scanner.close();
}

```

Figure 2-9: Method main (Part II)

2.4. Output Result of Task I

When I clicked run, I was asked a question “Which Task Do You Want to Run? (1 or 2):”. Then, I entered 1, which redirected me to task number 1, and the First Come First Serve, Shortest Job First, and Round Robin algorithms were executed. It showed detailed information for every algorithm listed below, including the waiting and turnaround times for each process, but also the average waiting time and turnaround time of the whole process of each algorithm. The figures below represent the output of the first task (*Figure 2-10, Figure 2-11 & Figure 2-12*)

```
Which Task Do You Want to Run? (1 or 2): 1
***** First Come First Serve *****
Order of Execution is: P1(7), P2(8), P3(2), P4(6)

Waiting Times:
P1 = 0
P2 = 7
P3 = 15
P4 = 17

Turnaround Times:
P1 = 7
P2 = 15
P3 = 17
P4 = 23

Average Waiting Time = 9.75
Average Turnaround Time = 15.50
```

Figure 2-10: Output of Task I (Part I)

```
***** Shortest Job First *****
Order of Execution is: P3(2), P4(6), P1(7), P2(8)

Waiting Times:
P1 = 8
P2 = 15
P3 = 0
P4 = 2

Turnaround Times:
P1 = 15
P2 = 23
P3 = 2
P4 = 8

Average Waiting Time = 6.25
Average Turnaround Time = 12.00
```

Figure 2-11: Output of Task I (Part II)

```
***** Round Robin *****
Order of Execution is: P1(5), P2(5), P3(2), P4(5), P1(2), P2(3), P4(1)

Waiting Times:
P1 = 12
P2 = 14
P3 = 10
P4 = 17

Turnaround Times:
P1 = 19
P2 = 22
P3 = 12
P4 = 23

Average Waiting Time = 13.25
Average Turnaround Time = 19.00
```

Figure 2-12: Output of Task I (Part III)

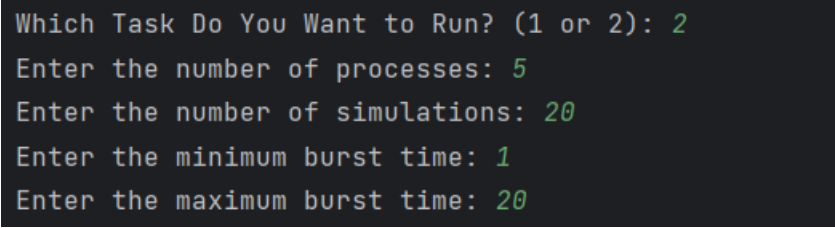
3. Task II

Task II has some minor differences from Task I since the user now is required to enter the number of processes, the number of simulations, and the minimum and maximum of the burst time. As a result, it will randomly create n processes with the burst time between the minimum and maximum values.

As defined in the introduction paragraph, I did not create other methods, but I used the existing one, with the difference that on the screen will not be displayed the order of execution, turnaround, and waiting times of each process, and even the average turnaround time. This was done by using the ternary operation.

3.1. Output Result of Task II

In this task, the user is asked for the task number that needs to be executed, the number of processes, the number of simulations, and the minimum & maximum burst time. After that, the output is shown below in *Figure 3-1* & *Figure 3-2*.



```
Which Task Do You Want to Run? (1 or 2): 2
Enter the number of processes: 5
Enter the number of simulations: 20
Enter the minimum burst time: 1
Enter the maximum burst time: 20
```

Figure 3-1: Output of Task II (Part I)

```
***** Iteration Number 1 *****
***** First Come First Serve *****
Average Waiting Time = 18.80
***** Shortest Job First *****
Average Waiting Time = 9.80
***** Round Robin *****
Average Waiting Time = 20.00
***** Iteration Number 2 *****
***** First Come First Serve *****
Average Waiting Time = 26.20
***** Shortest Job First *****
Average Waiting Time = 22.80
***** Round Robin *****
Average Waiting Time = 46.20
***** Iteration Number 3 *****
***** First Come First Serve *****
Average Waiting Time = 24.80
***** Shortest Job First *****
Average Waiting Time = 20.80
***** Round Robin *****
Average Waiting Time = 40.40
```

Figure 3-2: Output of Task II (Part II)

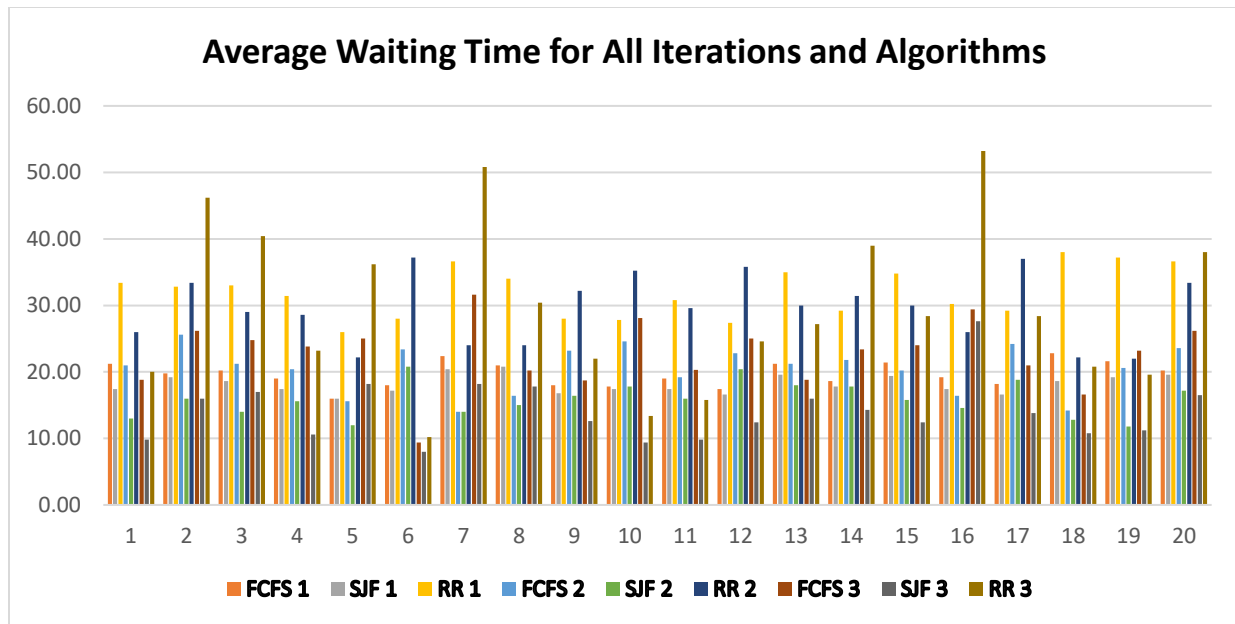
3.2. Result Interpretation

The waiting time of every iteration and algorithm are saved to an excel file, and the table is shown below in *Table 3.1*.

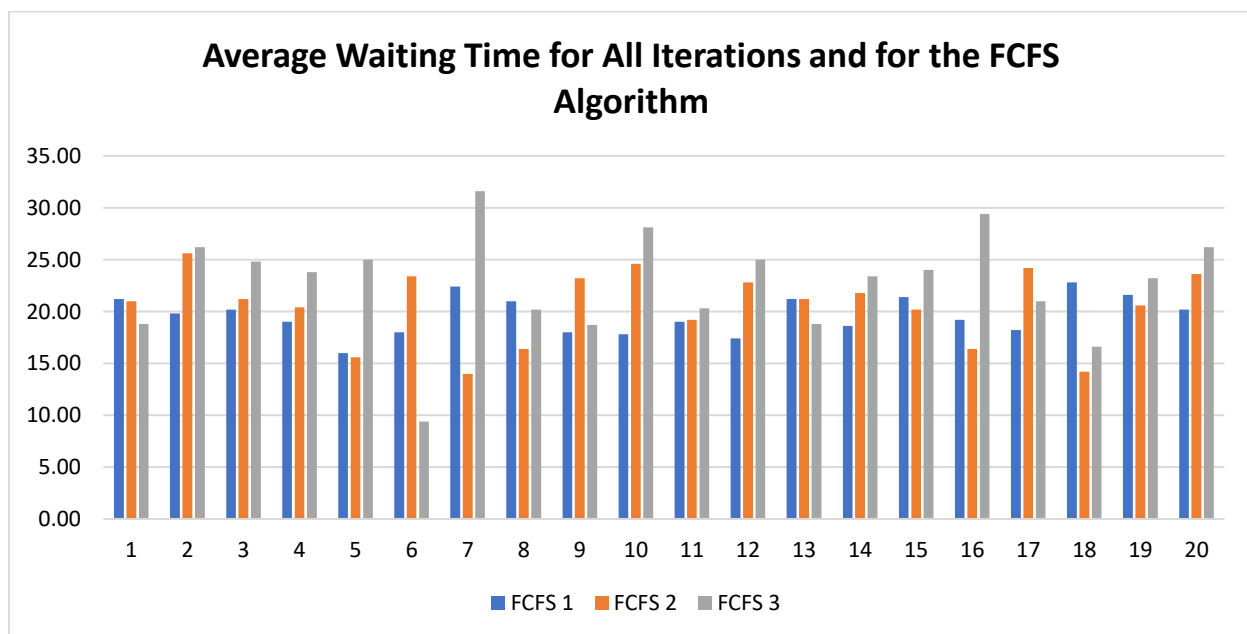
Iterations	Burst Time (8 - 12)			Burst Time (5 - 15)			Burst Time (1 - 20)		
	FCFS	SJF	RR	FCFS	SJF	RR	FCFS	SJF	RR
1	21.20	17.40	33.40	21.00	13.00	26.00	18.80	9.80	20.00
2	19.80	19.20	32.80	25.60	16.00	33.40	26.20	22.80	46.20
3	20.20	18.60	33.00	21.20	14.00	29.00	24.80	20.80	40.40
4	19.00	17.40	31.40	20.40	15.60	28.60	23.80	10.60	23.20
5	16.00	16.00	26.00	15.60	12.00	22.20	25.00	18.20	36.20
6	18.00	17.20	28.00	23.40	20.80	37.20	9.40	8.00	10.20
7	22.40	20.40	36.60	14.00	14.00	24.00	31.60	26.00	50.80
8	21.00	20.80	34.00	16.40	15.00	24.00	20.20	17.80	30.40
9	18.00	16.80	28.00	23.20	16.40	32.20	17.00	12.60	22.00
10	17.80	17.40	27.80	24.60	17.80	35.20	12.80	9.40	13.40
11	19.00	17.40	30.80	19.20	16.00	29.60	12.40	9.80	15.80
12	17.40	16.60	27.40	22.80	20.40	35.80	25.00	12.40	24.60
13	21.20	19.60	35.00	21.20	18.00	30.00	18.80	16.00	27.20
14	18.60	17.80	29.20	21.80	17.80	31.40	23.40	19.60	39.00
15	21.40	19.40	34.80	20.20	15.80	30.00	24.00	12.40	28.40
16	19.20	17.40	30.20	16.40	14.60	26.00	29.40	27.60	53.20
17	18.20	16.60	29.20	24.20	18.80	37.00	21.00	13.80	28.40
18	22.80	18.60	38.00	14.20	12.80	22.20	16.60	10.80	20.80
19	21.60	19.20	37.20	20.60	11.80	22.00	15.60	11.20	19.60
20	20.20	19.60	36.60	23.60	17.20	33.40	26.20	21.00	38.00

Table 3.1: Average Waiting Time for 20th Iterations and for All Algorithms

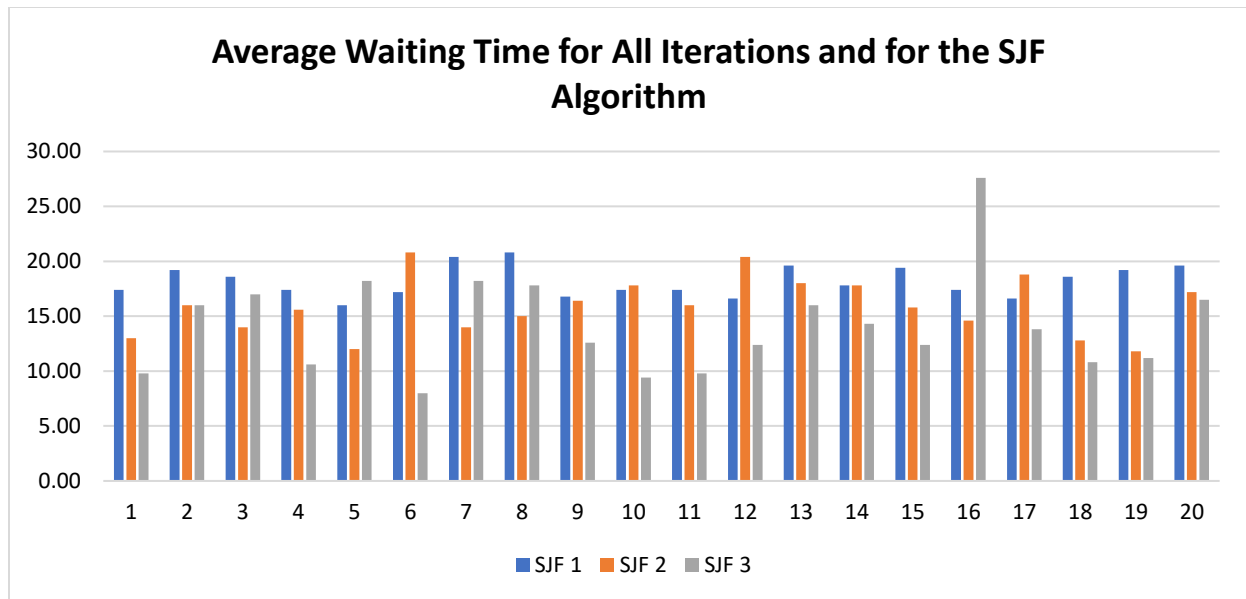
Also, to determine better the trends that come from three ranges of burst times, I have included four bar charts, one is for all iterations and algorithm (*Graphic 3.1*), one is only for the FCFS algorithm (*Graphic 3.2*), one is only for SJF algorithm (*Graphic 3.3*), and the last one is only for RR algorithm with quantum of 5-time units (*Graphic 3.4*).



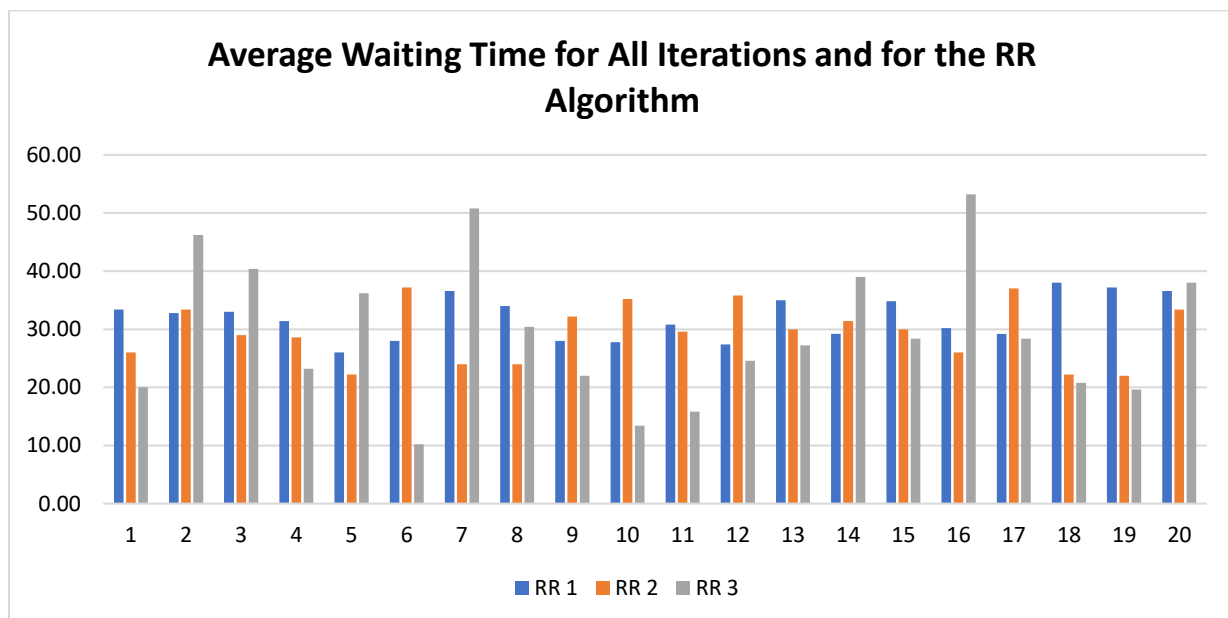
Graphic 3.1: Average Waiting Time for All Iterations and Algorithms



Graphic 3.2: Average Waiting Time for All Iterations and for the FCFS Algorithm



Graphic 3.3: Average Waiting Time for All Iterations and for the SJF Algorithm



Graphic 3.4: Average Waiting Time for All Iterations and for the RR Algorithm

Yes, there are observable trends in the average waiting time as the range of process burst times becomes wider for FCFS, SJF, and RR (quantum = 5). In the First Come First Serve algorithm, the processes are executed in the order they arrive, so by analyzing the bar charts we see that as the range of burst times widens, the average waiting time tends to increase significantly, and the reason behind this stands that processes with shorter burst times might get stuck behind longer processes, leading to increased waiting times, and inefficient utilization of CPU time.

Shortest Job First is a CPU scheduling algorithm used in operating systems, which prioritizes the execution of processes based on their burst time. The purpose of this algorithm is to schedule the process with the shortest burst time first. According to the table and the bar charts, the average waiting time is generally lower and less affected by a wider burst time range compared to the FCFS algorithm. This happens because this algorithm prioritizes shorter processes, and by doing so it minimizes the overall waiting time, even when burst time varies.

The Round-Robin algorithm is another important CPU scheduling algorithm, every process is assigned a fixed time quantum, and the CPU switches between processes in a circular order. In our case, the quantum is 5-time units. The data analyzed by the table and bar charts make it obvious that the average waiting time tends to increase slightly with wider burst time ranges, but less so than FCFS. There is a genuine explanation behind these results that the quantum time limit prevents any process from monopolizing the CPU for too long, reducing overall waiting times.

To sum up, Shortest Job First (SJF) typically demonstrates lower performance in minimizing average waiting times when compared to both First Come First Serve (FCFS) and Round Robin (RR), particularly in scenarios with wider burst time ranges. SJF's ability to

prioritize shorter jobs inherently leads to reduced waiting times. On the other hand, RR provides a balance between fairness and efficiency, mitigating the impact of wide burst time ranges compared to FCFS.