Eric Bruhns

Prof. Tannaz Damavandi

CS4310.02

March 26, 2023

Operating Systems Project One Report

For the first Operating Systems Project, I was tasked with writing a program that utilizes multithreading to sort an array of integers, as well as a second program that simulates the scheduling algorithms found within most operating systems: First Come First Serve (FCFS), the algorithm which runs processes based on the order in which they arrive in a ready queue; Shortest Job First (SJF), the algorithm which runs the shortest processes first; Preemptive Priority Scheduling, the algorithm which runs processes based on their given priority number; and Round Robin, the algorithm that uses a Time Quantum in order to switch between processes in the queue.

I. MultiThread Sorting

The first part of the project required some knowledge of how to implement threading in Java. Given the task, it was apparent that two Thread Subclasses will be used: one that takes half of the array and sorts it, while the other then takes two sorted subarrays and, with a while loop, inserts values of both into a single one in ascending order.

Since both threads needed parameters, I could not just create classes that inherited from Java's Thread class. Instead, I had to create a SortingThread Class and a MergingThread Class that implements the Runnable interface, so that way I can create Constructors that take in parameters.

The SortingThread Class takes as arguments an array in which to acquire data, an upper and lower bound index on where to acquire the data from, and an integer ID number as to distinguish the array in a message to the user. The SortingThread Constructor takes the array and the bounds, and places them into the class's private member array, which is half the size of the array passed in. The SortingThread Class also has a run() method which takes its private array and sorts it using an Insertion Sort algorithm. It also has a get-method which returns the private array of the class.

The MergingThread Class takes in three arguments: two arrays and an empty array. The two arrays are the sorted subarrays (hence why the SortingThread class needs a get-method for the subarray) and will be placed in the empty array in ascending order. The class's run() method will loop through the empty array's size and will place an element from either of the given two arrays into the position based on the one that's less than the other.

The main function asks the user to input a string of numbers separated by a space. The string needs to start with a number or else the program will throw an exception. The string is then split into an array of strings using Java's String.split() method, where the numbers–if done correctly–should be an element of an array. Then with a while loop, each string element is converted to a number and passed into a global array of integers. With the data acquired, two SortingThread Classes were created to divide the array into subarrays, the first half and the second half. Those two classes were then passed into Java's Thread Constructors and were activated concurrently. The Thread.join() method, which stops a program from executing until the thread has completed, had to be used, so the displayed results look accurate (some tests had elements repeating, yet the algorithm was correct).

As the threads run, a message is displayed, informing the user of the process working. Thread.sleep(1000) allowed the program to pause every second to allow the user to process the message. The message is displayed at certain parts of the Insertion Algorithm.

Once both threads had finished executing, a MergingThread object was made and executed, again using Thread.join() in order to display accurate results. Once the thread was completed in its task, a function to print the final loop was called.

The results are displayed below, using the example given by the project's instructions:

```
PS C:\Users\bruhe\Documents\os_hw\Threading_HW\MultiThreadSorting> java Main

Welcome to Mutlithread Sorting!!
Please enter your array (separate with spaces)
7 12 19 3 18 4 2 6 15 8
Sorting Sub-Array for thread 1
Sorting Sub-Array for thread 2
Sorting Sub-Array for thread 2
Sorting Sub-Array for thread 1
Sorting Sub-Array for thread 2
Sorting Sub-Array for thread 1
Sorting Sub-Array for thread 2
Sorting Sub-Array for thread 1

The sorted array:
2 3 4 6 7 8 12 15 18 19

Exiting Program...

PS C:\Users\bruhe\Documents\os_hw\Threading_HW\MultiThreadSorting>
```

II. Scheduling Simulator

Part two of the project was more complicated than part one despite the lack of multithreaded programming. The biggest challenge was how to manipulate a queue when

implementing preemptive algorithms like Preemptive Priority Scheduling. The program was to simulate the execution of projects within an Operating System using the four algorithms mentioned before; therefore, five classes were created (excluding the main class): a PCB class which simulates the structure of a real Process Control Block data structure; and classes for each of the four algorithms: FCFS, SJF, PPS, and RR. Upon hindsight, it is questioning why these were made into classes at all. With some reflection, it was assured that the algorithms were made into their own Java files as a way to ease the readability of the code, and Java files are required to be a public class for the compiler.

However, a PCB Class was necessary because it contains private variables such as Process ID, arrival_time, burst_time, priority, used_time, waiting_time, and turnaround_time. Methods for the class were needed to save these variables and update them accordingly when processed through an algorithm or to reset some of their values when restarting a new algorithm. Variables such as used_time are necessary to keep track of the process's lifetime, while wait_time and turnaround_time are needed for average calculations. These variables are automatically initialized to zero.

When the program begins to run, the user is greeted with a welcome message and is prompted to enter the name of a file. This file will contain details of the processes used throughout the program. The file is formatted to contain the following data: PID, Arrival Time, Burst Time and Priority. Each line represents a process while each column represents the four data types. For testing, the file included that contains appropriate data is "input.txt". For each line read in the file, the data is passed to the PCB's Constructor and the newly created object is added to an ArrayList. Given the complexity of manipulating arrays and queues when

implementing the PPS algorithm, an ArrayList was the best data structure that allowed flexibility when popping, pushing and inserting elements into specific positions of a queue.

Once the ArrayList of PCBs is created, the array is sorted based on ascending arrival time, utilizing an overloaded compareTo() method coded within the PCB class. I found sorting the ArrayList beneficial when checking for arrival times within the algorithms.

After the ArrayList of PCBs is created and sorted, the user will then be displayed a menu given options preceded by a numerical value. The options provided are the four scheduling algorithms, and an exit option to end the program. A user will enter a value, which will be passed to a switch statement, which will then call the corresponding algorithm's execution method. Each algorithm is based on the same loop structure, yet with different tweeks to the code that complement their distinct features.

First Come First Serve (FCFS) was the simplest algorithm, since it ran based on the processes by order of their arrival time. The algorithm is also non-preemptive, so the currently running processes will not be interrupted by incoming ones. The algorithm sets up a runtime-counter as well as initializes an idle PCB, using the class's default constructor. Before each loop of the algorithm begins, it will always search the waiting ArrayList (the one passed as a parameter) for processes whose Arrival Time matches the current runtime-counter. If it does, the process is inserted into a Ready Queue also in the form of an ArrayList. If the current PCB is an Idle one (idle PCBs have PIDs of -1), it will check the Ready Queue and if there is an element in the first slot, the current PCB will run through the process. The algorithm will always check if the current PCB object's burst_time is equal to its used_time variable. If not, the used_time variable will be incremented, along with the runtime counter, before the loop repeats the process.

The user is always informed when a process is running based on a message displayed in the terminal: “Process [PID] is CURRENTLY running at time [runtime-counter]”.

Once the process’s burst_time equals its used_time, the process has finished executing. The message “Process [PID] FINISHED running at time [runtime-counter]” is displayed in order to inform the user of the algorithm’s progress. Afterwards, the process’s wait_time and turnaround_time are calculated. Turnaround Time is calculated by subtracting the current runtime-counter with the process’s arrival time, while Wait Time is calculated by subtracting turnaround_time from its burst_time.Once done, the code will then check if both the Ready Queue and Waiting Queue are empty, signifying all processes were executed. If not, the code will check if the Ready Queue is the only one empty, while the Waiting Queue still contains elements. If this condition is true, then the current PCB will return to being idle; but if the condition is false, the next leading PCB will be running and the loop will continue to run. It should be made apparent that each iteration of the loop is a symbolic representation of a millisecond, which is the time unit used to measure the process’s execution. Despite a Thread.sleep(1000) code used to delay the results, the integers being counted are still represented as a millisecond. The delay is only there for the user’s convenience.

The Shortest Job First algorithm is fairly trickier. It follows the same structure as the FCFS loop structure of the program, but now a second nested loop is required within the initial ArrayList search of arriving processes, for the ones with the shortest job need to be closer to the front of the Ready Queue. The process will traverse the Waiting Queue and be placed in the appropriate position of the Ready Queue based on its burst_time length. It’s not the most efficient way of sorting the processes, but given the time frame, my goal was to have a working demo of the program to present to the class. The aspect of efficiency can be addressed in a future

update. After the Ready Queue is processed, the algorithm runs just the same as FCFS since SJF is non-preemptive; therefore, a process can continue running without interruption from another process.

Unlike the first two, however, Preemptive Priority Scheduling, as its name suggests, is a preemptive algorithm, making it the most difficult algorithm to implement, due to its ability to interrupt a running process. The beginning of the algorithm is practically the same as SJF, but with a nested loop used to sort the Ready Queue by priority rather than burst_time. From there, the execution will run like any of the other algorithms: it will check to see if the current process is idle, and begin executing a new process if true; the program will check if the current PCB's burst_time is equal to its used_time, and will calculate the wait_time and turnaround_time, while defining a new current PCB from the Ready Queue. The only difference being is if the process is not finished, the program must now check to see if the first process in the Ready Queue now has a higher priority than the running one. If true, the current PCB must go back to the Ready Queue in the appropriate position based on its priority, and the higher priority process will take its place. All unfinished changes to the current PCB are marked with dashed-lines for the user to know when a process has changed without completion. The algorithm continues to run and loop as normal.

Round Robin is the most different of the algorithms, but not as tricky, for it does not have preemption as one of its features. Instead, the algorithm introduces a Time-Quantum in the program, which will alternate between processes within the queue once the time-quantum is reached. If a user were to choose the "RR" choice in the menu, they will then be prompted to enter a time-quantum integer, which will be a second parameter to pass into the Round Robin execution method along with the waiting ArrayList of PCBs.

The RR algorithm runs similarly to FCFS as well, considering no preemption or sorting is involved. If a process has not finished executing, then the program will check if the Time-Quantum Counter (which is decremented with every loop iteration) equals zero. If so, then the current PCB is added to the end of the Ready Queue and the first element takes its place. The Time Quantum counter is then reset and the information of the currently running process is displayed. A series of dashed-lines is also used to alert the user when a process has switched to another via the Time Quantum.

All of the algorithms end the same way. Once every process finishes executing, they are placed within a Finished Queue and with another loop, their Average Wait Time (AWT) and their Average Turnaround Time (ATT) are calculated using their private variables. Those results and the result of their Average Response Time, which is the mathematical difference between ATT and AWT, are displayed for the user to see for roughly five seconds, until the menu is displayed again, prompting the user to test another algorithm or to exit the program when finished.

III. Results

The testing results were ran with data below:

	Arrival Time	Burst Time	Priority
P1	0	2	2
P2	0	1	1
P3	0	8	4
P4	0	4	2
P5	0	5	5

First Come First Serve

```
PS C:\Users\bruhe\Documents\os_hw\Threading_HW\SchedulingSimulator> java Main

Welcome to THE SCHEDULER

Please enter a File name:
input.txt
Please select a Scheduling algorithm:
        1. First Come First Serve
        2. Shortest Job First
        3. Preemptive Priority Scheduling
        4. Round Robin
        5. Quit
1
```

```
Process PID 1 is CURRENTLY running at time 0
Process PID 1 is CURRENTLY running at time 1

Process PID 1 FINISHED running at time 2

Process PID 2 is CURRENTLY running at time 2

Process PID 2 FINISHED running at time 3

Process PID 3 is CURRENTLY running at time 3
Process PID 3 is CURRENTLY running at time 4
Process PID 3 is CURRENTLY running at time 5
Process PID 3 is CURRENTLY running at time 6
Process PID 3 is CURRENTLY running at time 7
Process PID 3 is CURRENTLY running at time 8
Process PID 3 is CURRENTLY running at time 9
Process PID 3 is CURRENTLY running at time 10

Process PID 3 FINISHED running at time 11

Process PID 4 is CURRENTLY running at time 11
Process PID 4 is CURRENTLY running at time 12
Process PID 4 is CURRENTLY running at time 13
Process PID 4 is CURRENTLY running at time 14

Process PID 4 FINISHED running at time 15
```

```
Process PID 5 is CURRENTLY running at time 15
Process PID 5 is CURRENTLY running at time 16
Process PID 5 is CURRENTLY running at time 17
Process PID 5 is CURRENTLY running at time 18
Process PID 5 is CURRENTLY running at time 19

Process PID 5 FINISHED running at time 20


The Execution has completed!

The results:
Average Wait Time = 6.20 milliseconds
Average Response Time = 4.00 milliseconds
Average Turnaround Time = 10.20 milliseconds
```

** NOTE: A process's Burst Time is equivalent to the number of "CURRENTLY" statements for a process*

Shortest Job First

```
Please select a Scheduling algorithm:
        1. First Come First Serve
        2. Shortest Job First
        3. Preemptive Priority Scheduling
        4. Round Robin
        5. Quit
2
Process PID 2 is CURRENTLY running at time 0

Process PID 2 FINISHED running at time 1

Process PID 1 is CURRENTLY running at time 1
Process PID 1 is CURRENTLY running at time 2

Process PID 1 FINISHED running at time 3

Process PID 4 is CURRENTLY running at time 3
Process PID 4 is CURRENTLY running at time 4
Process PID 4 is CURRENTLY running at time 5
Process PID 4 is CURRENTLY running at time 6

Process PID 4 FINISHED running at time 7

Process PID 5 is CURRENTLY running at time 7
Process PID 5 is CURRENTLY running at time 8
Process PID 5 is CURRENTLY running at time 9
Process PID 5 is CURRENTLY running at time 10
Process PID 5 is CURRENTLY running at time 11

Process PID 5 FINISHED running at time 12
```

```
Process PID 3 is CURRENTLY running at time 12
Process PID 3 is CURRENTLY running at time 13
Process PID 3 is CURRENTLY running at time 14
Process PID 3 is CURRENTLY running at time 15
Process PID 3 is CURRENTLY running at time 16
Process PID 3 is CURRENTLY running at time 17
Process PID 3 is CURRENTLY running at time 18
Process PID 3 is CURRENTLY running at time 19

Process PID 3 FINISHED running at time 20


The Execution has completed!

The results:
Average Wait Time = 4.60 milliseconds
Average Response Time = 4.00 milliseconds
Average Turnaround Time = 8.60 milliseconds
```

Preemptive Priority Scheduling

```
Please select a Scheduling algorithm:
        1. First Come First Serve
        2. Shortest Job First
        3. Preemptive Priority Scheduling
        4. Round Robin
        5. Quit
3
Process PID 2 is CURRENTLY running at time 0

Process PID 2 FINISHED running at time 1

Process PID 1 is CURRENTLY running at time 1
Process PID 1 is CURRENTLY running at time 2

Process PID 1 FINISHED running at time 3

Process PID 4 is CURRENTLY running at time 3
Process PID 4 is CURRENTLY running at time 4
Process PID 4 is CURRENTLY running at time 5
Process PID 4 is CURRENTLY running at time 6

Process PID 4 FINISHED running at time 7
```

```
Process PID 3 is CURRENTLY running at time 7
Process PID 3 is CURRENTLY running at time 8
Process PID 3 is CURRENTLY running at time 9
Process PID 3 is CURRENTLY running at time 10
Process PID 3 is CURRENTLY running at time 11
Process PID 3 is CURRENTLY running at time 12
Process PID 3 is CURRENTLY running at time 13
Process PID 3 is CURRENTLY running at time 14

Process PID 3 FINISHED running at time 15

Process PID 5 is CURRENTLY running at time 15
Process PID 5 is CURRENTLY running at time 16
Process PID 5 is CURRENTLY running at time 17
Process PID 5 is CURRENTLY running at time 18
Process PID 5 is CURRENTLY running at time 19

Process PID 5 FINISHED running at time 20


The Execution has completed!

The results:
Average Wait Time = 5.20 milliseconds
Average Response Time = 4.00 milliseconds
Average Turnaround Time = 9.20 milliseconds
```

Round Robin with Time Quantum of Four

```
Please select a Scheduling algorithm:
        1. First Come First Serve
        2. Shortest Job First
        3. Preemptive Priority Scheduling
        4. Round Robin
        5. Quit
4
Please input Time Quantum!
4
Process PID 1 is CURRENTLY running at time 0
Process PID 1 is CURRENTLY running at time 1

Process PID 1 FINISHED running at time 2
----------------------------------------
Process PID 2 is CURRENTLY running at time 2

Process PID 2 FINISHED running at time 3
----------------------------------------
Process PID 3 is CURRENTLY running at time 3
Process PID 3 is CURRENTLY running at time 4
Process PID 3 is CURRENTLY running at time 5
Process PID 3 is CURRENTLY running at time 6
----------------------------------------
Process PID 4 is CURRENTLY running at time 7
Process PID 4 is CURRENTLY running at time 8
Process PID 4 is CURRENTLY running at time 9
Process PID 4 is CURRENTLY running at time 10

Process PID 4 FINISHED running at time 11
----------------------------------------
```

```
Process PID 5 is CURRENTLY running at time 11
Process PID 5 is CURRENTLY running at time 12
Process PID 5 is CURRENTLY running at time 13
Process PID 5 is CURRENTLY running at time 14
----------------------------------------
Process PID 3 is CURRENTLY running at time 15
Process PID 3 is CURRENTLY running at time 16
Process PID 3 is CURRENTLY running at time 17
Process PID 3 is CURRENTLY running at time 18

Process PID 3 FINISHED running at time 19
----------------------------------------
Process PID 5 is CURRENTLY running at time 19

Process PID 5 FINISHED running at time 20

The Execution has completed!

The results:
Average Wait Time = 7.00 milliseconds
Average Response Time = 4.00 milliseconds
Average Turnaround Time = 11.00 milliseconds
```

To show preemption with PPS, the same data was run through but with random arrival times. The data is revised as follows:

	Arrival Time	Burst Time	Priority
P1	0	2	2
P2	1	1	1
P3	4	8	4
P4	5	4	2
P5	3	5	5

The results for running the PPS with the current data are given here:

```
Please select a Scheduling algorithm:
        1. First Come First Serve
        2. Shortest Job First
        3. Preemptive Priority Scheduling
        4. Round Robin
        5. Quit
3
Process PID 1 is CURRENTLY running at time 0
-----------------------------------------
Process PID 2 is CURRENTLY running at time 1

Process PID 2 FINISHED running at time 2

Process PID 1 is CURRENTLY running at time 2

Process PID 1 FINISHED running at time 3

Process PID 5 is CURRENTLY running at time 3
-----------------------------------------
Process PID 3 is CURRENTLY running at time 4
-----------------------------------------
Process PID 4 is CURRENTLY running at time 5
Process PID 4 is CURRENTLY running at time 6
Process PID 4 is CURRENTLY running at time 7
Process PID 4 is CURRENTLY running at time 8

Process PID 4 FINISHED running at time 9
```

```
Process PID 3 is CURRENTLY running at time 9
Process PID 3 is CURRENTLY running at time 10
Process PID 3 is CURRENTLY running at time 11
Process PID 3 is CURRENTLY running at time 12
Process PID 3 is CURRENTLY running at time 13
Process PID 3 is CURRENTLY running at time 14
Process PID 3 is CURRENTLY running at time 15

Process PID 3 FINISHED running at time 16

Process PID 5 is CURRENTLY running at time 16
Process PID 5 is CURRENTLY running at time 17
Process PID 5 is CURRENTLY running at time 18
Process PID 5 is CURRENTLY running at time 19

Process PID 5 FINISHED running at time 20


The Execution has completed!

The results:
Average Wait Time = 3.40 milliseconds
Average Response Time = 4.00 milliseconds
Average Turnaround Time = 7.40 milliseconds
```