CPU Scheduling

Zina Abohaia, 20194621@ug.buid.ac.ae

Yousef Mamdouh, 20196448@ug.buid.ac.ae

We have implemented FCFS, SJF, Priority, and Round-Robin.

* + Brief description of each with pseudo-code and short explanation

Priority

This scheduling algorithm uses priority of every task to determine which will be completed first. It is usually implemented in the form of integers, with increasing importance with increasing numbers. There are different ways to implement it; An array of tasks can be searched for the highest priority every time, or the array can be sorted by decreasing priority, so the highest priority task would be at index 0, and the lowest priority task would be at index n-1 where n is the size of the array.

Pseudocode:

1- Create input method “NAME”

2- initialize global ArrayList queue to the ArrayList passed as a parameter.

3- Sort queue to have the highest priority task would be at index 0, and the lowest priority task would be at index n-1 where n is the size of the array.

4- Make a method to pick the next task, and return it.

5- Return task and send it to CPU to be executed.

6- Remove Task from queue.

7- While there are still tasks, repeat step 5 and step 6.

8- Calculate average waiting time, average turnaround time, and average response time, by using CPU Burst and arrival time.

9- Output average waiting time, average turnaround time, and average response time.

Round-Robin

The Round-Robin scheduling algorithm uses something called a quantum which is a certain amount of time the CPU has until it will move onto the next task. If its current task CPU burst is less than or equal to the quantum, then it is executed. However, if its CPU burst is more than the quantum, the CPU will work on it for the duration of the quantum, then it will schedule it at the end of the ArrayList, saving its data, and it will move on to the next task.

Pseudocode:

1- Create input method “NAME”

2- initialize global ArrayList queue to the ArrayList passed as a parameter.

3- Initialize global Integer quantum to the quantum value passed as a parameter.

4- Make a method to pick the next task, and return it.

5- Return task and send it to CPU to be executed for the duration of the quantum.

6- If the task’s CPU burst is less than or equal to the quantum, then it is completed, and is removed from queue.

7- if the task’s CPU burst is more than the quantum, subtract the quantum from the CPU burst, and move it to the end of queue.

8- While there are tasks in queue, Repeat step 5 to 7.

9- Calculate sum of waiting time, and sum of turnaround time.

10- Use sum of waiting time, sum of turnaround time, CPU burst, arrival time, and number of tasks to calculate average waiting time, average turnaround time, and average response time.

11- Output average waiting time, average turnaround time, and average response time.

First come first serve

This scheduling algorithm uses arrival time of every task to determine which will be completed first. So, the tasks that arrive first will be executed first, and since in this project the arrival time for all tasks is 0, we can just arrange the tasks in order and execute them in order

Pseudocode:

1- Create input method “NAME”

2- initialize global ArrayList queue to the ArrayList passed as a parameter.

3- Make a method to pick the next task, and return it.

4- Return task and send it to CPU to be executed.

5- Remove Task from queue.

6- While there are still tasks, repeat step 4 and step 5.

7- Calculate average waiting time, average turnaround time, and average response time, by using CPU Burst and arrival time.

8- Output average waiting time, average turnaround time, and average response time.

Shortest job first

This scheduling algorithm uses CPU burst of every task to determine which task have the shortest time to be completed, and it executes tasks in order from the task with the shortest CPU burst to the task with the largest. There are different ways to implement it; An array of tasks can be searched for the lowest CPU burst every time, or the array can be sorted by increasing CPU burst, so the task with the lowest CPU burst would be at index 0, and the task with the highest CPU burst would be at index n-1 where n is the size of the array.

Pseudocode:

1- Create input method “NAME”

2- initialize global ArrayList queue to the ArrayList passed as a parameter.

3- Sort queue to have the task with the lowest CPU at index 0, and the task with the highest CPU burst at index n-1 where n is the size of the array.

4- Make a method to pick the next task, and return it.

5- Return task and send it to CPU to be executed.

6- Remove Task from queue.

7- While there are still tasks, repeat step 5 and step 6.

8- Calculate average waiting time, average turnaround time, and average response time, by using CPU Burst and arrival time.

9- Output average waiting time, average turnaround time, and average response time.

* Implementation Method:
  + Which programming language did you choose and why? Why Java? Or, Why C?

We have chosen to use the Programming Language Java for a number of reasons, including but are not limited to the following:

* + We are proficient in the use of Java.
  + Java is an object-oriented programming language, and we wanted to implement the algorithms as objects.
  + Java is a Data-oriented language; it makes structuring data convenient.
  + Java is a high-level language, and so it can be more easily understood by a user, or our professor during checking, than a low-level language such as C.

Similarly the reasons we didn’t pick the programming language C are, but are not limited to:

* + C is more of a procedural language.
  + C is not optimized to be object-oriented, nor data-centered.

Methodology:

Turnaround Time = Exit Time - Arrival Time

Waiting Time = Turnaround Time - CPU Burst Time

Response Time = Start Time - Arrival Time

Since Arrival Time = 0:

Turnaround Time = Exit Time

Response Time = Start Time

Substitiuting Turnaround Time with Exit TIme:

Waiting Time = Exit Time - CPU Burst Time

Using Start Time = Exit Time - CPU Burst Time:

Waiting Time = Start Time

and Since Respinse Time = Start Time:

Waiting Time = Response Time

Therefore, whenever we mention Waiting Time, we mean both Waiting Time and Response Time.

With the following Data Example:

Processes: T1, T2, T3, T4, T5

CPU Burst Time: x, y, z, a, b

Calculating the Turnaround Times (Exit Times) for every process:

T1= x

T2= x + y

T3= x + y + z

T4= x + y + z + a

T5= x + y + z + a + b

To calculate the Average of Turnaround Times we use:

Sum of Turnaround Times/Number of Processes

=( T1 + T2 +T3 + T4 + T5)/5

= x + (x+y) + (x + y + z) + ( x + y + z + a) + (x + y + z + a + b)/5

= 5x + 4y + 3z + 2a + b /5

and this was how we calculated the turnaround time.

For Waiting time, we subtracted the CPU Burst Time from the Turnaround Time:

WT1= T1 - x = x-x = 0

WT2= T2 - y = x + y - y = x

WT3= T3 - z = x + y + z - z = x + y

WT4 = T4 - a = x + y + z + a - a = x + y + z

WT5 = T5 - b = x + y + z + a +b - b = x + y + z + a

Average Waiting Time = (Sum of WT) / Number of Processes

= WT1 + WT2 + WT3 + WT4 + WT5/5

= 0 + x + (x + y) + ( x + y + z) + ( x + y + z + a)/5

= 4x + 3y + 2z + a/5

Therefore we can see that we can get the Average Waiting Time by subtracting (x + y + z + a + b/5) from the Average Turnaround Time

and since the response time = waiting time, we get the same results for average response time = average waiting time.

* Results & Analysis:
  + Did your ‘similar test cases’ show different schedules? What can you say about the ‘correctness’ of your output?

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Number of Tasks | Average Turnaround Time | | | | Average Waiting Time | | | |
|  |  |  |  |  |  |  |  |  |
|  | FCFS | SJF | Priority | RR | FCFS | SJF | Priority | RR |
| 10 | 117 | 106 | 117 | 183 | 94 | 84 | 94 | 161 |
| 20 | 288 | 175 | 302 | 334 | 260 | 146 | 274 | 306 |
| 30 | 443 | 232 | 402 | 456 | 415 | 204 | 374 | 428 |
| 40 | 604 | 349 | 577 | 651 | 573 | 319 | 546 | 681 |
| 50 | 782 | 486 | 819 | 938 | 748 | 452 | 785 | 904 |
| 60 | 958 | 547 | 951 | 1072 | 926 | 514 | 919 | 1040 |
| 70 | 1127 | 642 | 1102 | 1252 | 1094 | 609 | 1069 | 1219 |
| 80 | 1292 | 699 | 1204 | 1373 | 1260 | 667 | 1172 | 1341 |
| 90 | 1458 | 818 | 1388 | 1605 | 1426 | 786 | 1355 | 1572 |
| 100 | 1632 | 955 | 1626 | 1861 | 1598 | 921 | 1592 | 1827 |

We tested the same algorithms for different number of tasks, and it stayed consistent for the most part, with SJF algorithm doing the best by far and the other 3 algorithms being similar for the most part, testing the correctness of CPU algorithms results is actually a difficult problem known as the test oracle problem, however there is a solution to it and it’s metamorphic testing, unfortunately it wasn’t available in our disposal, so we tested the algorithms using different number of tasks multiple times, and we got the same results each time.

* + Any other interesting findings?

Up until 30 tasks, SJF had better average waiting time and average turnaround time, while the rest were very similar, but the higher the number of tasks got, Round-Robin algorithm started getting significantly higher average waiting and average turnaround times than the rest which was interesting, so we tried testing different quantum values, as quantum is a constant value we sit, we thought maybe it’s too high so the algorithm is acting like FCFS or maybe it’s too low so that each cycle is not efficient, and we found that having a quantum that is higher than 10 makes it preform worse, and that a quantum of 8 did the best which was also interesting.

We also found that the average waiting time and average turnaround time for each algorithm was very similar with an almost identical pattern

Which makes sense since arrival time for all tasks was 0

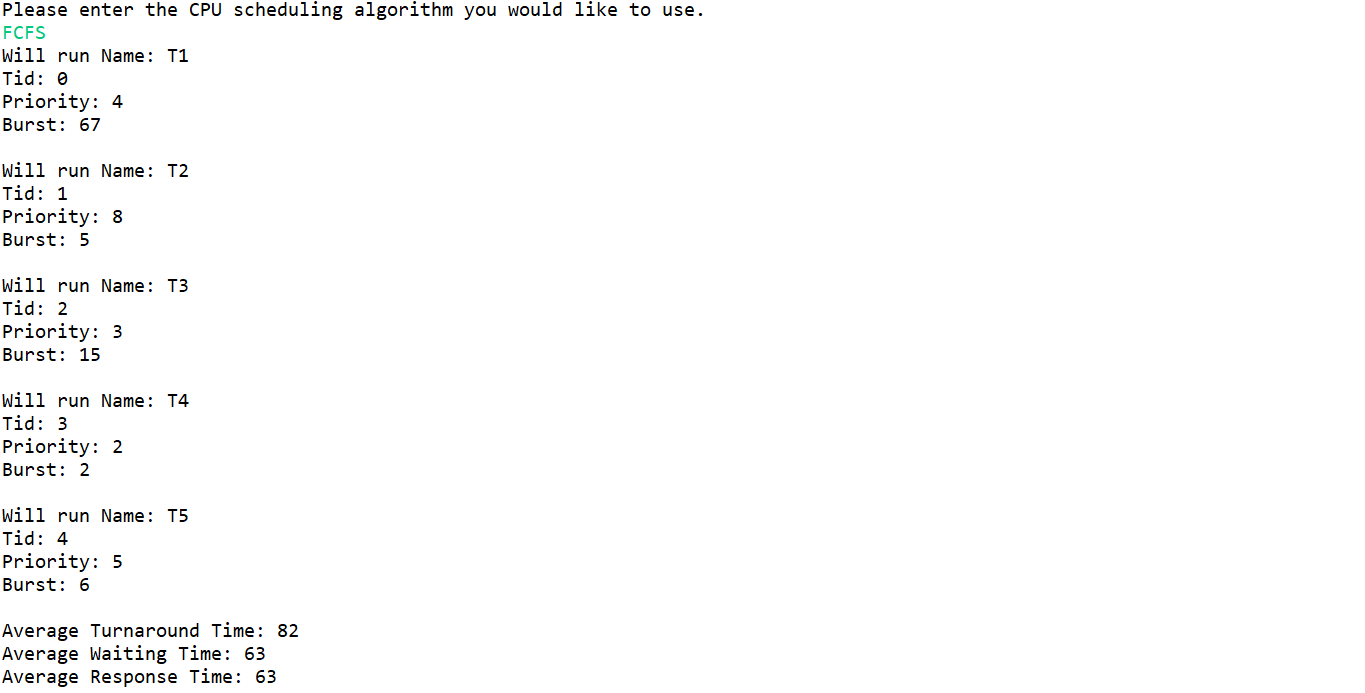
* + Comments on the results and the whole experience (Be scientific and precise; For instance, you may mention some ideas that you investigated but did not turn out to be interesting, etc.).

We initially tried programing the priority algorithm as well as SJF algorithm to search the array for the highest priority and the lowest CPU burst respectively, instead of arranging them in a queue from the beginning, but we found that not to be very interesting or efficient so we went with the arranging method which was more efficient.

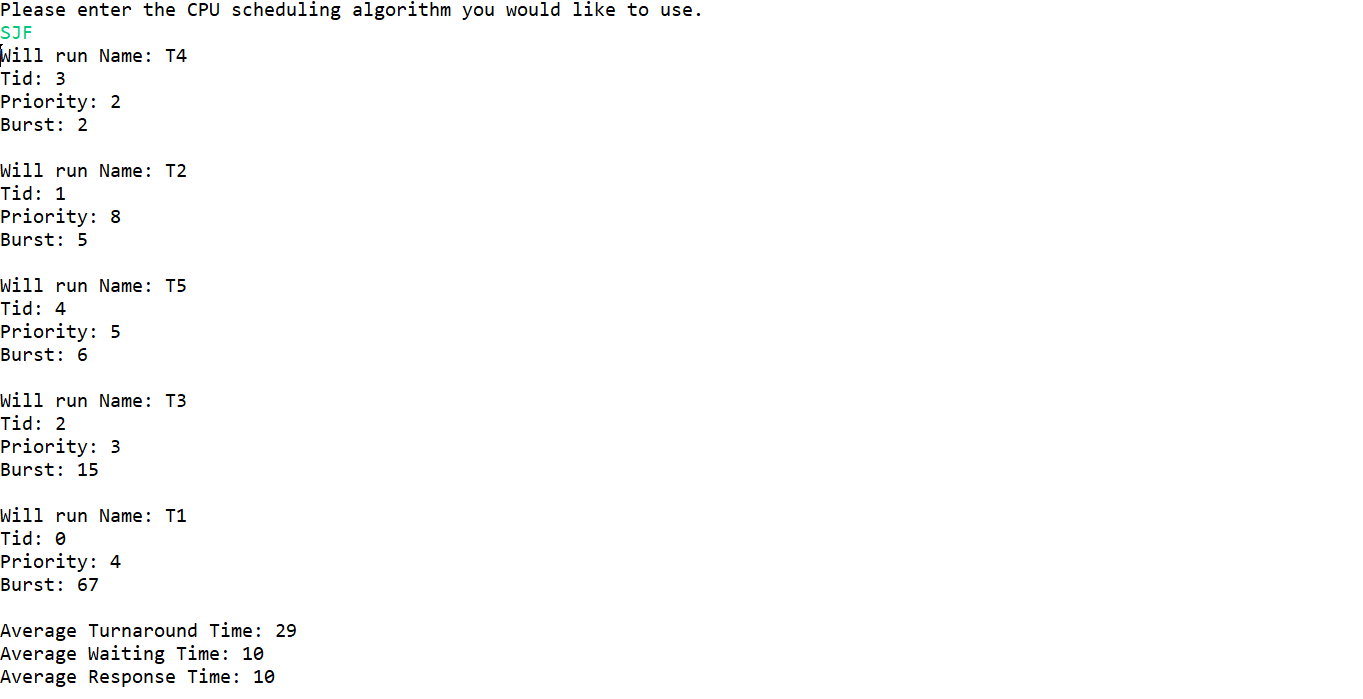
* Appendix:
  + Screenshots of your final output/run for each algorithm on a sample data set

We have chosen to use five tasks in this sample data set to be able to show the order clearly.

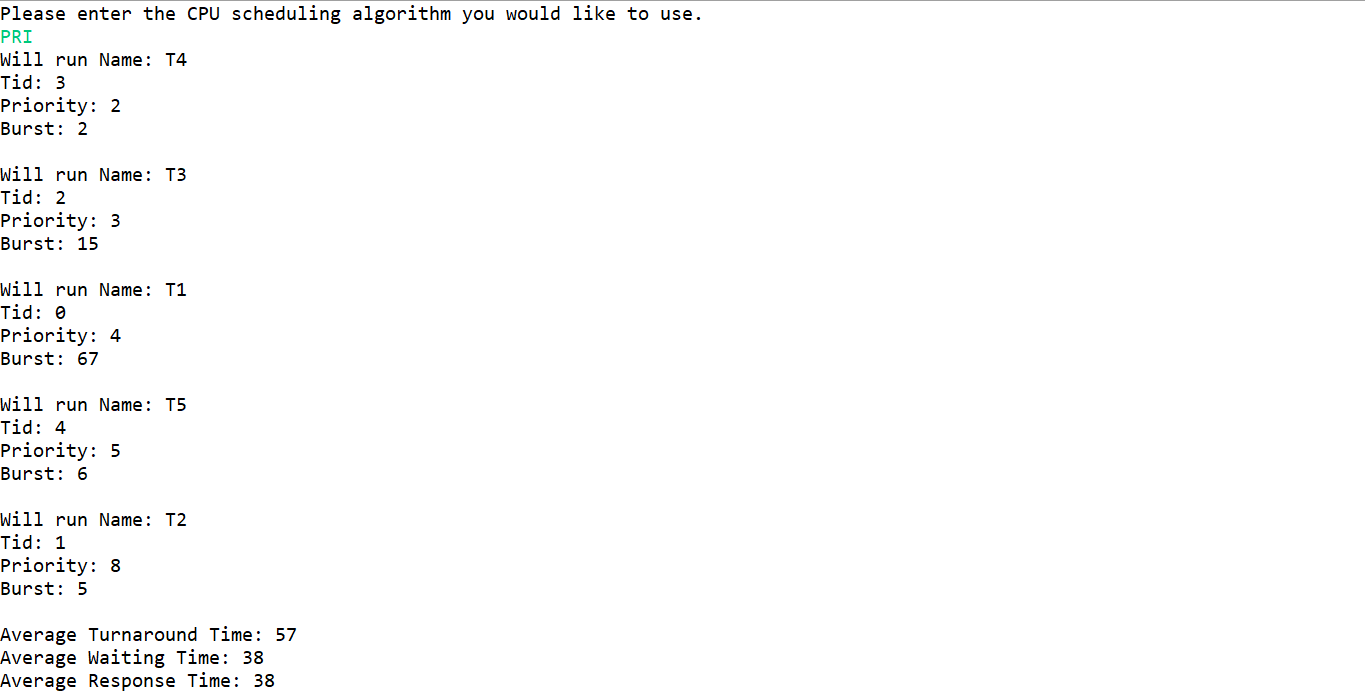
FCFS



SJF



PRIORITY



RR

