ATYPON

CPU Simulator Assignment

Prepared by: Mahmoud Haifawi

Feb-2023

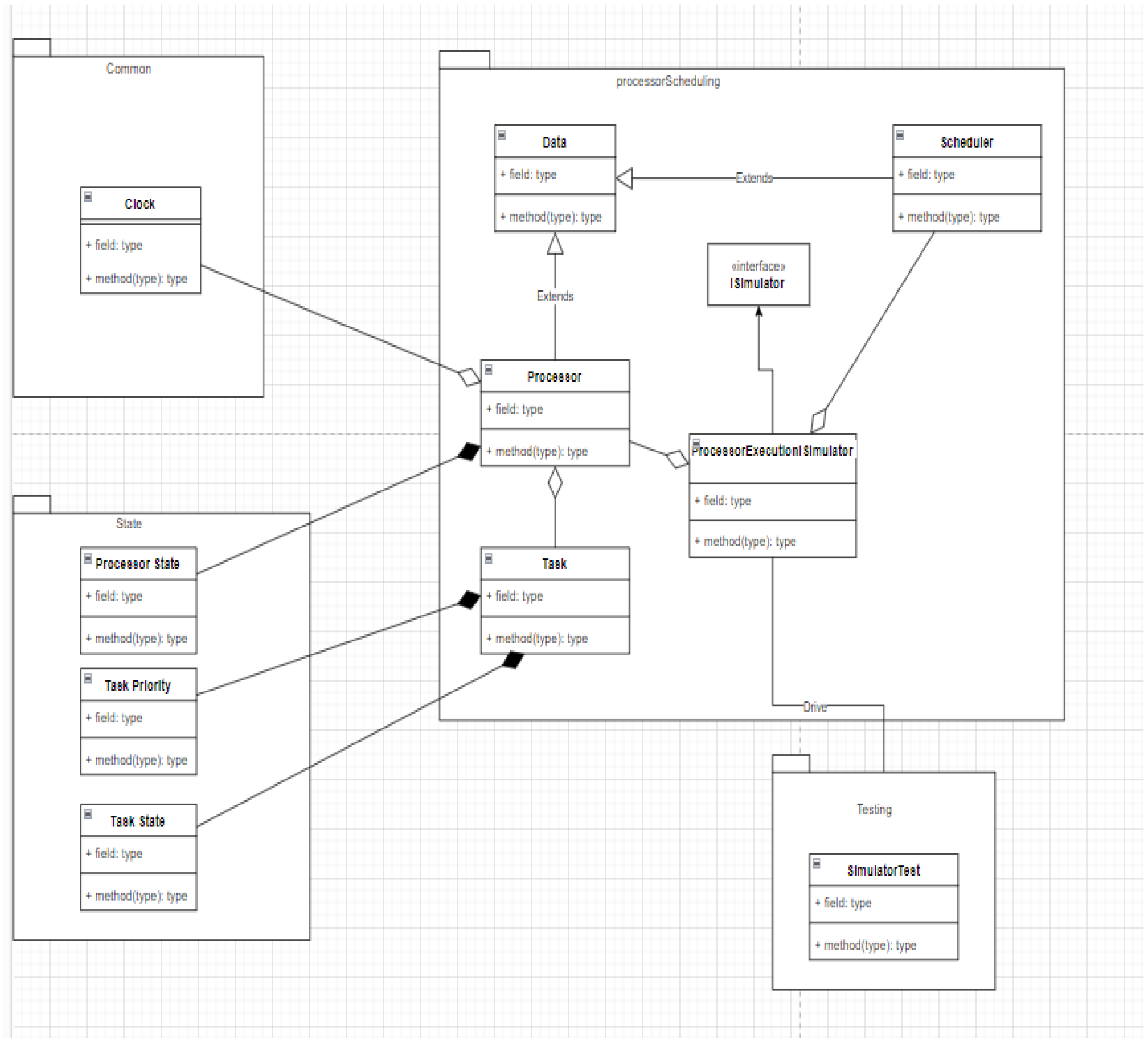
1. Abstract

The objective of this assignment is to build a Processor Execution Simulator that can simulate the execution of tasks in a computer system. The number of processors is fixed throughout the simulation and all processors are synchronized, meaning they all have the same clock. The simulator will receive an input text file that describes the tasks and their information, including their creation time, execution time, and priority (high or low). The simulator will also have a queue for holding tasks, and a scheduler that is responsible for determining which tasks to assign to available processors.

Tasks with high priority will be prioritized over low priority tasks, and if there is a tie, the task with the longest execution time will be assigned first. Task execution cannot be interrupted, and the task creation time is negligible. The simulator takes three arguments as input, the number of processors, the total number of clock cycles, and the path to the input file.

The output of the simulation will be a cycle-by-cycle report about important events during the simulation, such as created tasks, running tasks, completed tasks, and processor's state.

2. UML Diagram



3. Design Pattern

I used the following design pattern:

Singleton

I implemented the Singleton design pattern for the design of both the ProcessorExecutionSimulator and Scheduler classes as only one instance of each class should exist during the simulation.

The Singleton pattern ensures that a class has only one instance while providing a global point of access to this instance.
This is important in the current scenario as the ProcessorExecutionSimulator and Scheduler classes need to be accessible globally throughout the simulation and having multiple instances of these classes can result in unexpected behavior.

By implementing the Singleton pattern, we can ensure that only one instance of these classes is created and accessed, which provides consistency and reliability to the simulation.

4. SOLID Principles

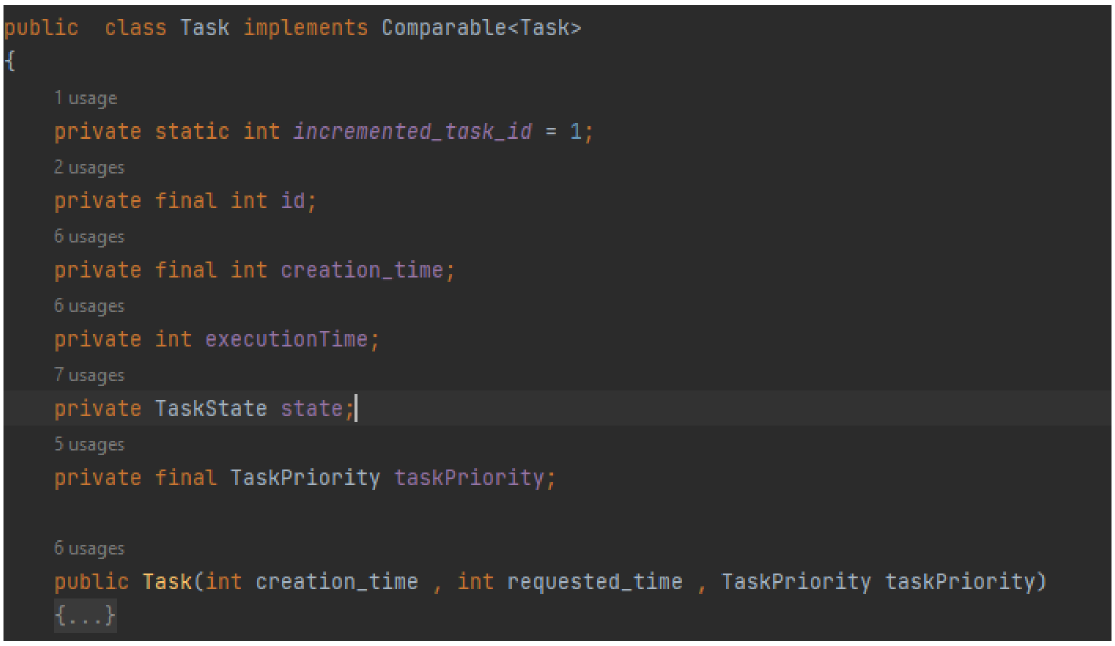
I have applied and taken consideration of the SOLID principle, while programming the simulator.

For each principle I will demonstrate how this principle was applied in my program:

4.1 S: Single Responsibility Principle

"A class should have one and only responsibility over a single part of the functionality provided by the software."

All the classes and methods I've written adhere to this principle, ensuring that none of them deviate from it.

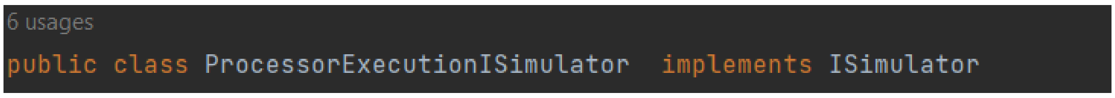


```
public  class Task implements Comparable<Task>
{
    1 usage
    private static int incremented_task_id = 1;
    2 usages
    private final int id;
    6 usages
    private final int creation_time;
    6 usages
    private int executionTime;
    7 usages
    private TaskState state;
    5 usages
    private final TaskPriority taskPriority;

    6 usages
    public Task(int creation_time , int requested_time , TaskPriority taskPriority)
    {...}
```

4.2 O: Open / Closed Principle (OCP)

"Entities should be open for extension , but closed for modification (We can apply this using abstract class , and other classes should inherit from this abstract class) . "

I created a class called ProcessorExecutionSimulator that implement an interface call ISimulator .



```
6 usages
public class ProcessorExecutionISimulator  implements ISimulator
```

4.3 I: Interface Segregation Principle (ISP)

"One fat interface need to be split to many smaller and relevant interfaces ."

As the assignment was not complex, there was no requirement to strictly adhere to the principle of having one responsibility per class.

4.4 Liskov Substitution

"Super types can be substituted by subtypes without affecting correctness. In other words, a subclass should not change the expected behavior inherited from a superclass"

An example from the project is ISimulator interface and ProcessorExecutionSimulator class, as ProcessorExecutionSimulator class implements ISimulator interface, and they are substituted in a place of each other, and they do not affect the correctness.

5. Clean Code

Throughout my work on this assignment, I made a concerted effort to maintain clean code practices in my code. This can be seen through the use of descriptive names, the absence of flag arguments, clear and readable conditions, proper exception handling, and the removal of unnecessary comments.

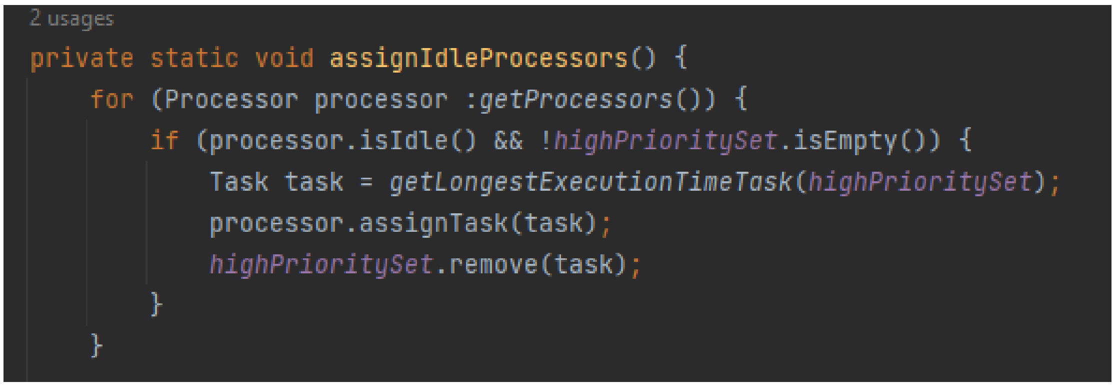
5.1 Code smells

5.1.1 Comments

I focused on writing clear and concise code that speaks for itself, rather than relying on comments to explain the logic. I also made sure to give meaningful names to variables and methods to provide clarity and understanding of their purpose. This approach not only helps in keeping the code clean and readable but also makes it easy to maintain in the future. By breaking down the algorithm into smaller steps, I made the code more approachable and less complex for someone to understand. Overall, I aimed to strike a balance between code readability and commenting to effectively communicate the code's functionality.

5.1.2 Duplication

This principle is applied for all methods as there is no duplication in the code and the code that has been used multiple time it is refactor to method.



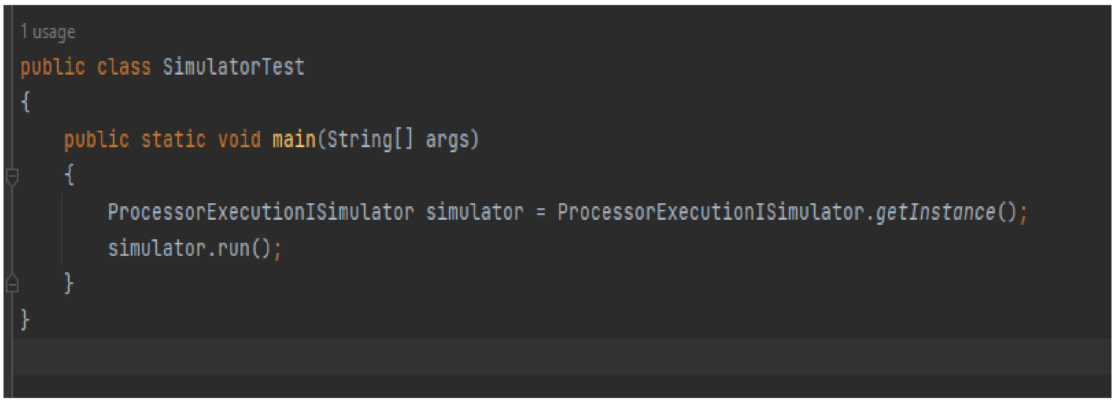
```
2 usages
private static void assignIdleProcessors() {
    for (Processor processor :getProcessors()) {
        if (processor.isIdle() && !highPrioritySet.isEmpty()) {
            Task task = getLongestExecutionTimeTask(highPrioritySet);
            processor.assignTask(task);
            highPrioritySet.remove(task);
        }
    }
```

3.1.3 Dead code

This principle is applied for the whole project as there is no code that never used or in a block that never reached.

6. Easy To Use

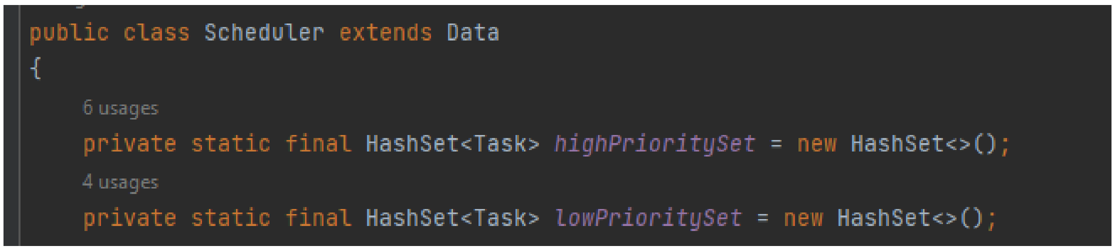
The driver method is designed to provide an easy-to-use interface for the user, by hiding any complex operations behind the scenes. This helps to simplify the overall experience for the user, making the software more accessible and user-friendly. By focusing on this goal, I was able to create a clean and intuitive interface that is easy for users to understand and use effectively.
This is a snapshot of the driver method:



```
1 usage
public class SimulatorTest
{
    public static void main(String[] args)
    {
        ProcessorExecutionISimulator simulator = ProcessorExecutionISimulator.getInstance();
        simulator.run();
    }
}
```

7. Tie Breaking

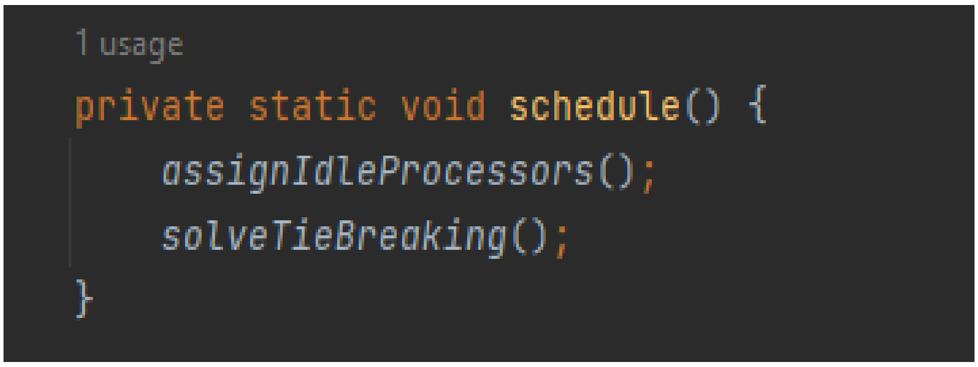
By separating low priority tasks from high priority ones, I was able to prioritize and efficiently manage my workload, making sure that the important tasks were completed first and that the less important tasks did not interfere with them. This helped me to deliver the best results possible and complete the project in a timely manner.



```
public class Scheduler extends Data
{
    6 usages
    private static final HashSet<Task> highPrioritySet = new HashSet<>();
    4 usages
    private static final HashSet<Task> lowPrioritySet = new HashSet<>();
```

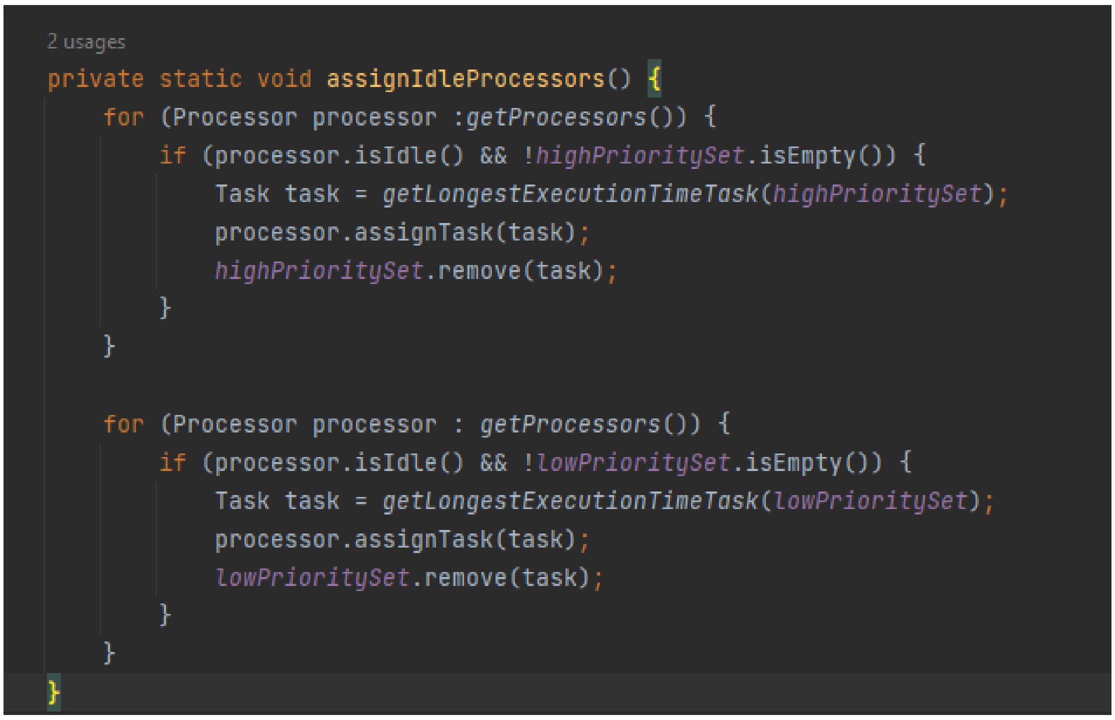
The tasks in the sets are ordered based on their creation time and requested time.
The tasks that arrived first are placed at the front of the set and, in the case of equal creation times, the tasks with lower requested times are prioritized.
This ensures that tasks are processed in an efficient and organized manner.

Schedule Function:



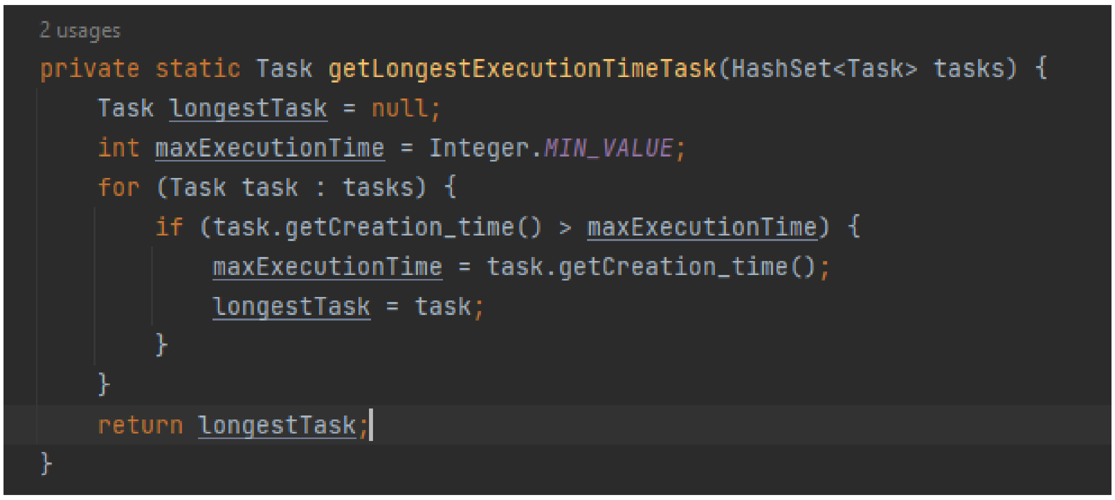
```
1 usage
private static void schedule() {
    assignIdleProcessors();
    solveTieBreaking();
}
```

In this function, I prioritize the allocation of high priority tasks to processors. If there are any idle processors, I then assign low priority tasks to them. This ensures that important tasks are completed first while utilizing available resources efficiently.



```
2 usages
private static void assignIdleProcessors() {
    for (Processor processor :getProcessors()) {
        if (processor.isIdle() && !highPrioritySet.isEmpty()) {
            Task task = getLongestExecutionTimeTask(highPrioritySet);
            processor.assignTask(task);
            highPrioritySet.remove(task);
        }
    }

    for (Processor processor : getProcessors()) {
        if (processor.isIdle() && !lowPrioritySet.isEmpty()) {
            Task task = getLongestExecutionTimeTask(lowPrioritySet);
            processor.assignTask(task);
            lowPrioritySet.remove(task);
        }
    }
}
```

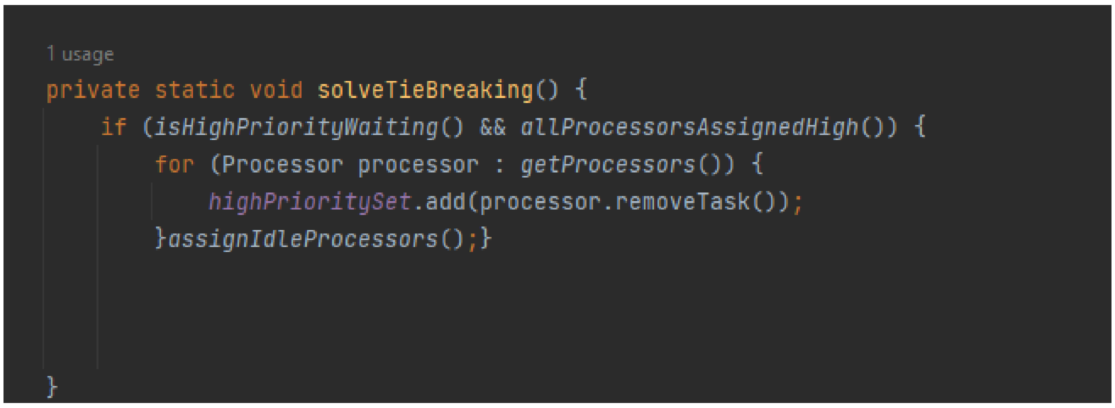
In this Function I made sure that if there is a tie in priority I will assign the task with longest execution Time



```
2 usages
private static Task getLongestExecutionTimeTask(HashSet<Task> tasks) {
    Task longestTask = null;
    int maxExecutionTime = Integer.MIN_VALUE;
    for (Task task : tasks) {
        if (task.getCreation_time() > maxExecutionTime) {
            maxExecutionTime = task.getCreation_time();
            longestTask = task;
        }
    }
    return longestTask;
}
```

In the solveTiebreaking method, I handle the tiebreak between high priority tasks by first retrieving them from the processors and adding them to the high_priority_set. The tasks in the set are ordered based on the creation time and execution time of the task, so that the tasks that arrived first and have a lower execution time are given priority.

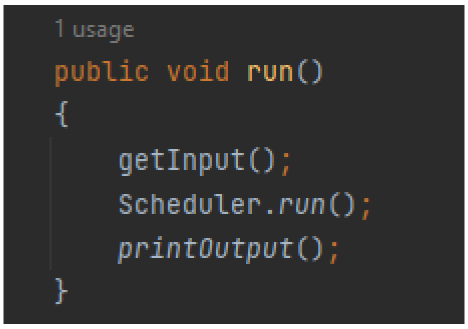
If there are any idle processors, I reassign the tasks to these processors. This function is only executed when there is a waiting high priority task, and all processors are holding only high priority tasks.



```
1 usage
private static void solveTieBreaking() {
    if (isHighPriorityWaiting() && allProcessorsAssignedHigh()) {
        for (Processor processor : getProcessors()) {
            highPrioritySet.add(processor.removeTask());
        }assignIdleProcessors();}



}
```

8. Input and Output Sample

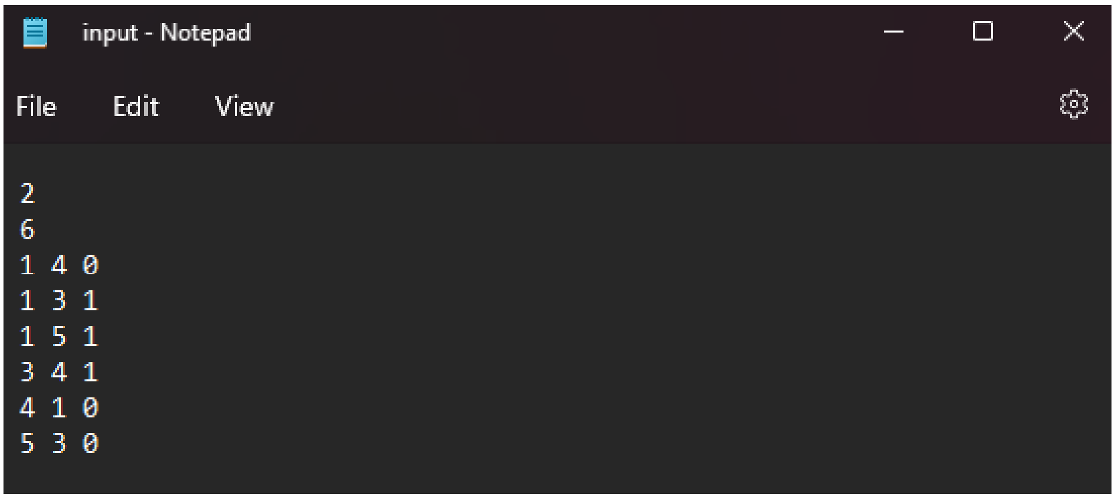
This is how my program handles input and output:



```
1 usage
public void run()
{
    getInput();
    Scheduler.run();
    printOutput();
}
```

This method, located within the Simulator class, takes the input from the user through a text file with a specific structure. After processing the scheduling, the output is then printed for the user on the console.

Input Sample:

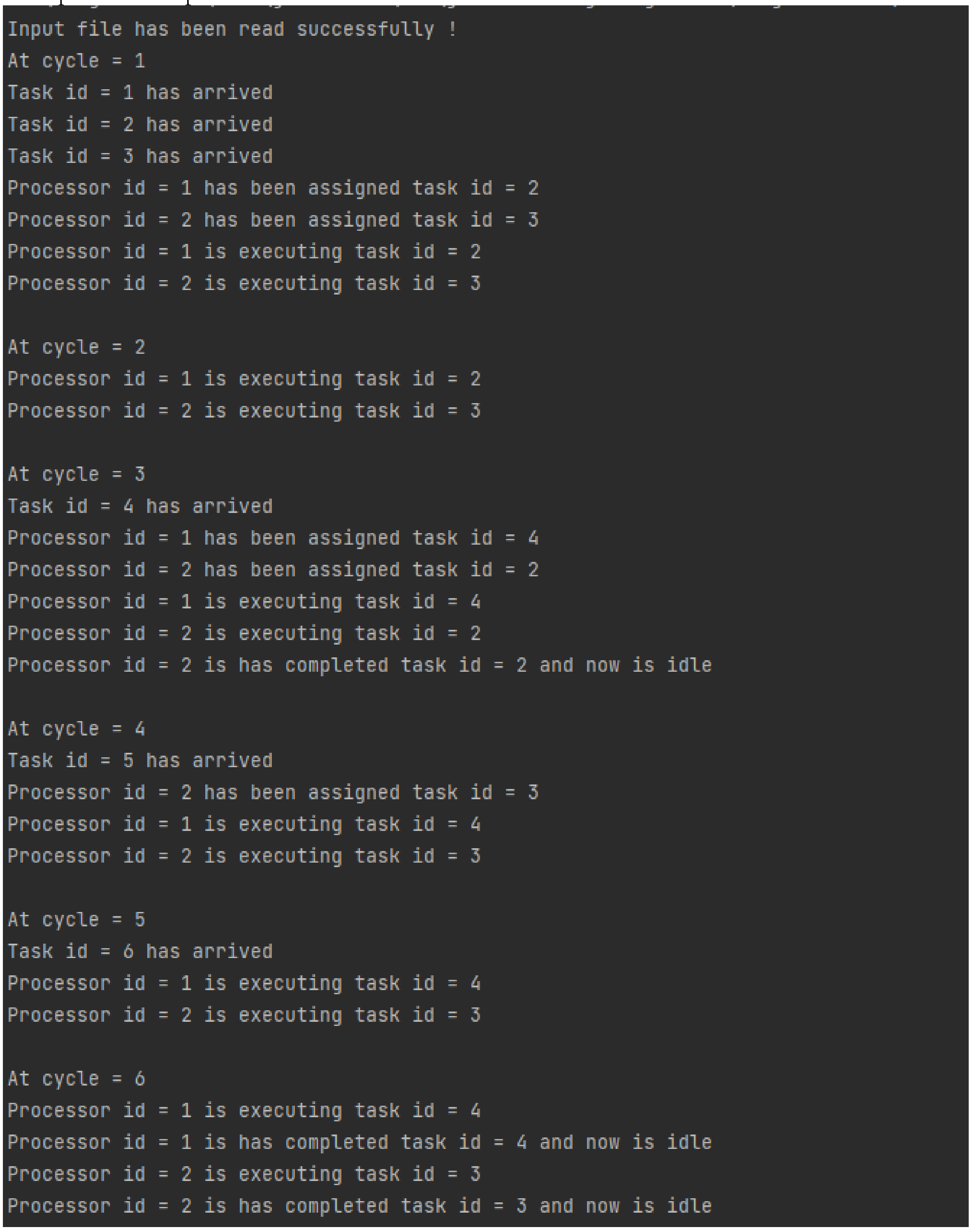


```
2
6
1 4 0
1 3 1
1 5 1
3 4 1
4 1 0
5 3 0
```

Note:

- the first line represents the number of processors.
- the second line represent the number of tasks.

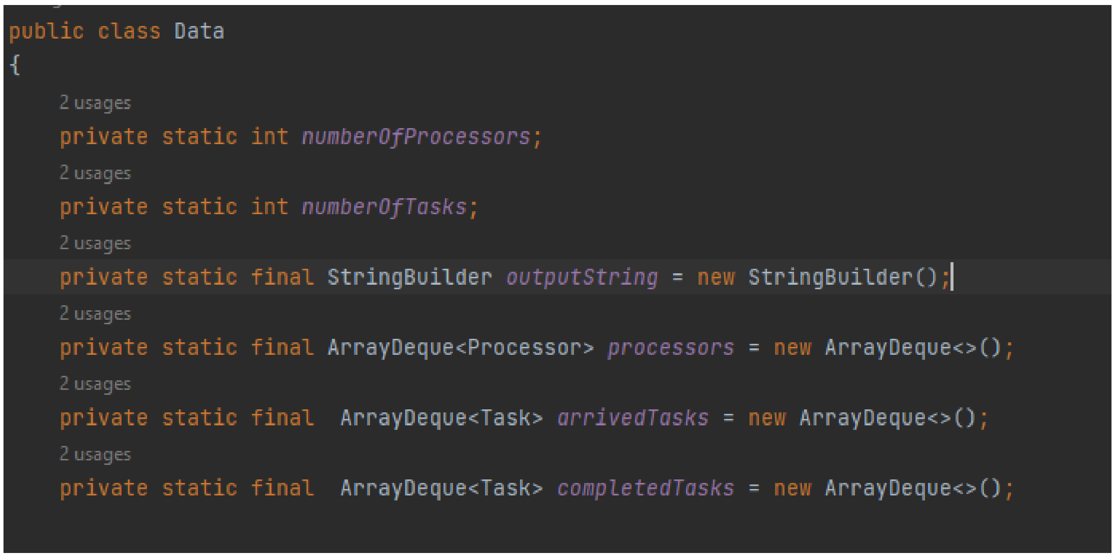
Example of the output:



```
Input file has been read successfully !
At cycle = 1
Task id = 1 has arrived
Task id = 2 has arrived
Task id = 3 has arrived
Processor id = 1 has been assigned task id = 2
Processor id = 2 has been assigned task id = 3
Processor id = 1 is executing task id = 2
Processor id = 2 is executing task id = 3

At cycle = 2
Processor id = 1 is executing task id = 2
Processor id = 2 is executing task id = 3

At cycle = 3
Task id = 4 has arrived
Processor id = 1 has been assigned task id = 4
Processor id = 2 has been assigned task id = 2
Processor id = 1 is executing task id = 4
Processor id = 2 is executing task id = 2
Processor id = 2 is has completed task id = 2 and now is idle

At cycle = 4
Task id = 5 has arrived
Processor id = 2 has been assigned task id = 3
Processor id = 1 is executing task id = 4
Processor id = 2 is executing task id = 3

At cycle = 5
Task id = 6 has arrived
Processor id = 1 is executing task id = 4
Processor id = 2 is executing task id = 3

At cycle = 6
Processor id = 1 is executing task id = 4
Processor id = 1 is has completed task id = 4 and now is idle
Processor id = 2 is executing task id = 3
Processor id = 2 is has completed task id = 3 and now is idle
```

9. Data Structure and Algorithms

In the Data class, I have included all of the data structures used in my application. This helps to organize the data into one unit, making it easier to deploy and maintain the shared data.



```
public class Data
{
    2 usages
    private static int numberOfProcessors;
    2 usages
    private static int numberOfTasks;
    2 usages
    private static final StringBuilder outputString = new StringBuilder();
    2 usages
    private static final ArrayDeque<Processor> processors = new ArrayDeque<>();
    2 usages
    private static final  ArrayDeque<Task> arrivedTasks = new ArrayDeque<>();
    2 usages
    private static final  ArrayDeque<Task> completedTasks = new ArrayDeque<>();
```

9.1.1 Array Dequeue

using an Array Deque also makes the code more readable and efficient, as it can perform all the operations in constant time.

The ability to efficiently add and remove elements from both the front and back of the queue, as well as access the first element without removing it, makes it an ideal data structure for implementing task scheduling in my application.

The only operation that does not have a constant time complexity is the remove operation, but this is still much more efficient than using a traditional array.

9.1.2 StringBuilder

The use of StringBuilder in Java offers several advantages over the use of normal strings. One key advantage is its mutability. StringBuilder allows for the efficient modification of strings by appending, inserting, or deleting characters, while strings are immutable and therefore any modification results in the creation of a new string. Another advantage is improved performance. StringBuilder has a much better time complexity for string concatenation operations compared to normal strings, making it faster and more efficient in cases where multiple string operations are performed. Additionally, StringBuilder has a larger capacity, which reduces the frequency of memory reallocations, further improving its performance. The use of StringBuilder is recommended in cases where frequent string modifications are required, such as in the construction of dynamic strings or the concatenation of large amounts of data.

Summary

In my program, the input is taken in the form of task details such as creating time, requested time, and priority.

These inputs are stored in objects of type "Task" and then added to either the high priority set or the low priority set. The program then schedules the tasks to processors by assigning high priority tasks first, and then if there are any remaining idle processors, the low priority tasks are assigned.

The solveTiebreaking method is used to break the tie between high priority tasks and ensure that the processors that deserve to be executed are assigned the tasks. Finally, the output is generated in the form of the processing time for each task.