

Processor Execution Simulation Report

ESSAM ABDO

Table of Contents

Table of Figures	2
Problem Statement.....	3
UML Diagram	4
Design Pattern.....	5
Singleton	5
SOLID Principles	5
S : Single Responsibility Principle	5
O : Open / Closed Principle (OCP).....	5
I : Interface Segregation Principle (ISP).....	6
Clean Code	6
Easy To Use	7
Tie Breaking.....	7
Input and Output Sample.....	11

Table of Figures

Figure 1 UML Diagram for Processor Execution Simulator	4
Figure 2 Interface Implementation	6
Figure 3 Main Method	7
Figure 4 Queues of the program	7
Figure 5 Schedule Function	8
Figure 6 assignIdleProcessors function	8
Figure 7 bringJusticeAmongTasks function	9
Figure 8 solveTieBreaking Function.....	10
Figure 9 Simulator run function	11
Figure 10 Input Structure	11
Figure 11 Input Example.....	12
Figure 12 Output Example	12

Problem Statement

In this assignment, you are required to build a simulator that simulates processor execution for processes (i.e., tasks). Assuming that we have P processors .

What should be a class? What interface should each class have? What are the relationships between classes? What data structures should I use? These are some of the questions that you should carefully consider.

Assume that your simulator is maybe extended by others in the future. Therefore, pay careful attention to object-oriented design to make your code: organized, flexible, and reusable.

Your simulator must be configurable using an input configuration file that specifies the number of processors and tasks, as well as the needed descriptions for each task (creation time, requested time, and priority). I will leave it to you to determine the exact nature of the configuration file.

Your simulator must produce meaningful output to the user that contains information about the total simulation time, completion times for all tasks, and other information you think is important. I will also leave it to you to determine the exact nature of the simulator output.

UML Diagram

This is the structure of my implementation of Processor Execution Simulation :

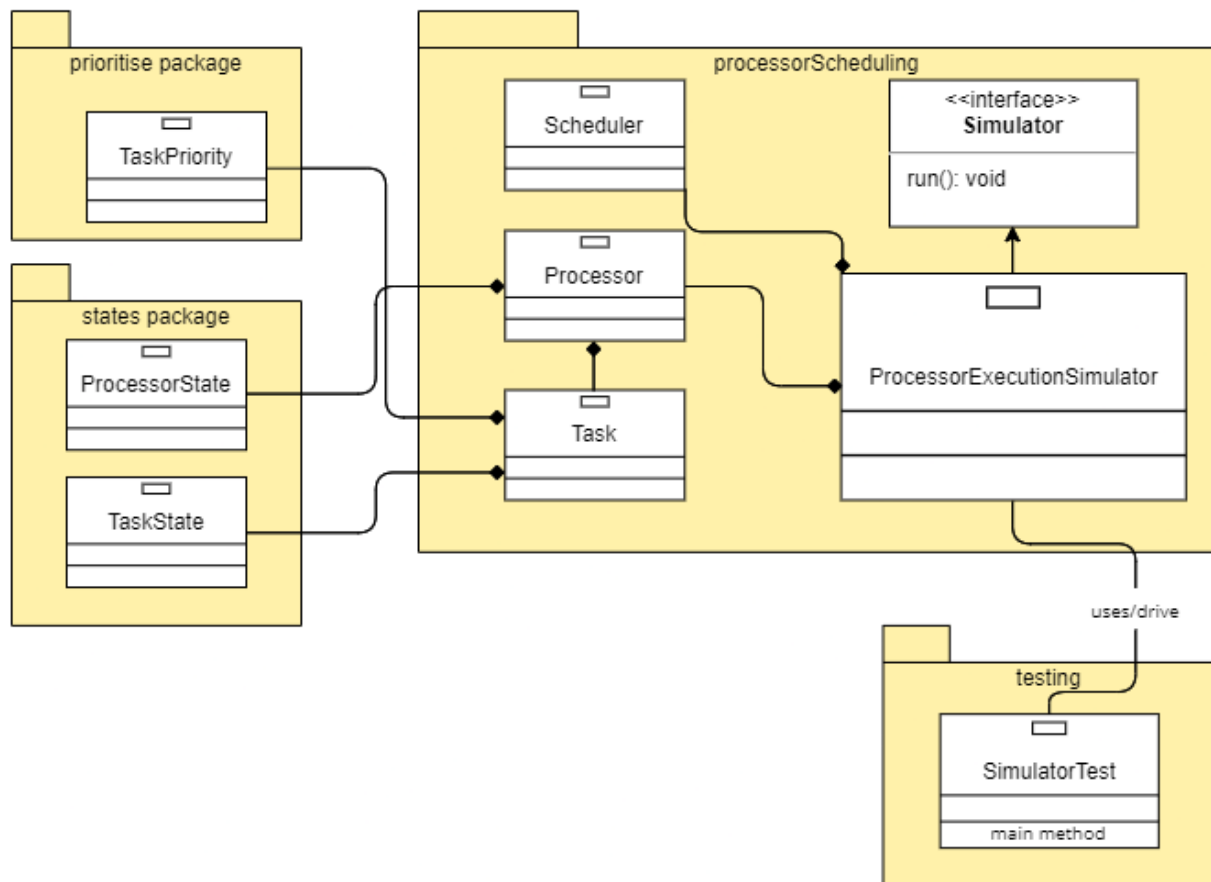


Figure 1 UML Diagram for Processor Execution Simulator

Design Pattern

I used the following design pattern :

Singleton

I used Singleton design pattern for designing both **ProcessorExecutionSimulator** and **Scheduler** classes , because for both of them , there should not exist more than one instance .

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 :

S : Single Responsibility Principle

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

All classes and methods I wrote , all of them apply this principle , so no method not class break this principle .

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 Simulator .

```
public class ProcessorExecutionSimulator implements Simulator
{
```

Figure 2 Interface Implementation

I : Interface Segregation Principle (ISP)

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

There was no need to apply this principle , due to the lack of complexity of this assignment .

Clean Code

In my coding journey for this assignment , I tried to maintain the clean code principles in my code as much as possible .

You can see that in meaningful names , eliminating flag arguments , beautiful and readable conditions , handling exceptions and eliminating useless comments .

Easy To Use

One of the most important factors that I tried to maintain was hiding unnecessary complexity from the user . This is a snapshot of the driver method :

```
package testing;

import processorScheduling.ProcessorExecutionSimulator;

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

Figure 3 Main Method

Tie Breaking

I have separated the low priority tasks apart from high priority ones :

```
private static final HashSet<Task> high_priority_set = new HashSet<>();
private static final HashSet<Task> low_priority_set = new HashSet<>();
```

Figure 4 Queues of the program

Both of them are **HashSet** of type **Task** . Tasks inside of them are ordered based on :

1. Creating Time of the task .
2. Then the requested time of task .

So the tasks that arrive first will be the first element in the set . If they arrived at the same time , they will be ordered based on the requested time of the tasks (the tasks with low requested time will be placed before the tasks that need more requested time) .

This is how I schedule the tasks :

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

Figure 5 Schedule Function

```
private static void assignIdleProcessors()
{
    for(Processor processor : processors)
    {
        if(processor.isIdle() && !high_priority_set.isEmpty())
        {
            processor.assignTask( getFirstElement( high_priority_set ) );
            high_priority_set.remove( getFirstElement( high_priority_set ) );
        }
    }

    for(Processor processor : processors)
    {
        if(processor.isIdle() && !low_priority_set.isEmpty())
        {
            processor.assignTask( getFirstElement( low_priority_set ) );
            low_priority_set.remove( getFirstElement( low_priority_set ) );
        }
    }
}
```

Figure 6 assignIdleProcessors function

In this function I assign the tasks with high priority to the processors , then if there are any remaining idle processors , I give them the tasks with low priority .


```

private static void bringJusticeAmongTasks()
{
    if(isHighPriorityWaiting() && isLowPriorityAssigned())
    {
        for(Processor processor : processors)
        {
            if(processor.getTask().isLowPriority())
            {
                if(getFirstHighPriorityWaitingTask() != null)
                {
                    replaceTasks( processor , getFirstHighPriorityWaitingTask() );
                }
                else
                {
                    break;
                }
            }
        }
    }
}

```

Figure 7 bringJusticeAmongTasks function

bringJusticAmongTasks method checks if there is any processor that is assigned a task with low priority , and if there is a high priority task that is waiting . If both of those conditions are met , then we assign the high priority waiting tasks to the processors that hold low priority tasks .

```

private static void solveTieBreaking()
{
    //NOW tie breaking between high and low
    if(isHighPriorityWaiting() && allProcessorsAssignedHigh())
    {
        for(Processor processor : processors)
        {
            high_priority_set.add( processor.removeTask() );
        }
        assignIdleProcessors();
    }
}

```

Figure 8 solveTieBreaking Function

solveTiebreaking method is made to break the tie between tasks that has high priority , simply , I take the high priority tasks from processors , then add those tasks to the **high_priority_set** , tasks will be ordered in the set based on the criteria I mentioned above , then I re-assign the tasks again the processors . This way , we ensure that processors that deserve to be executed will be assigned to the processors .

This function will be executed only if there is a high priority task that is waiting , and if all processors hold high priority tasks only .

Input and Output Sample

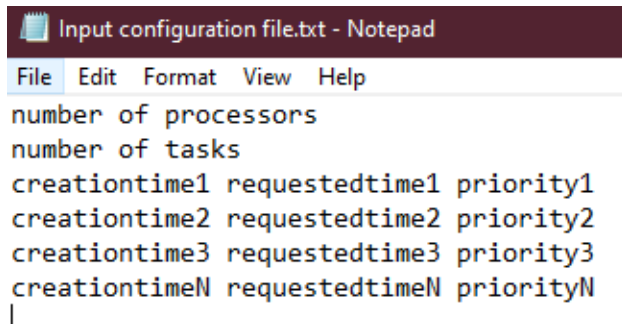
This is how my program handles input and output :

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

Figure 9 Simulator run function

This method is existed inside **Simulator class** , simply we will take the input from the user , using a text file , with specific input structure , then we will make the scheduling process , after that we will print the output using another text file for the user .

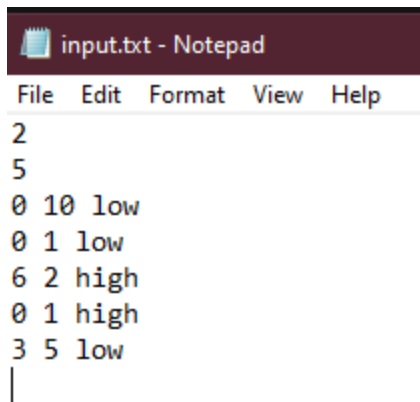
This is the structure that the user will have to follow for entering the input :



```
Input configuration file.txt - Notepad
File Edit Format View Help
number of processors
number of tasks
creationtime1 requestedtime1 priority1
creationtime2 requestedtime2 priority2
creationtime3 requestedtime3 priority3
creationtimeN requestedtimeN priorityN
|
```

Figure 10 Input Structure

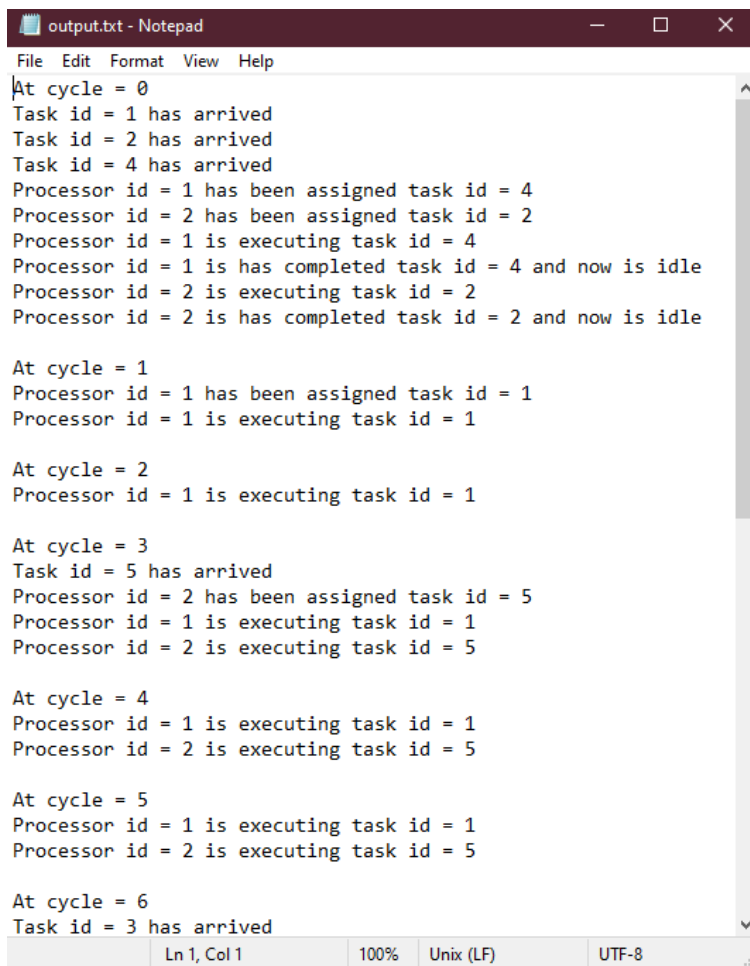
This is an example of how the input should be :



```
input.txt - Notepad
File Edit Format View Help
2
5
0 10 low
0 1 low
6 2 high
0 1 high
3 5 low
|
```

Figure 11 Input Example

This is an example of how the output will be :



```
output.txt - Notepad
File Edit Format View Help
At cycle = 0
Task id = 1 has arrived
Task id = 2 has arrived
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 = 1 is has completed task id = 4 and now is idle
Processor id = 2 is executing task id = 2
Processor id = 2 is has completed task id = 2 and now is idle

At cycle = 1
Processor id = 1 has been assigned task id = 1
Processor id = 1 is executing task id = 1

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

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

At cycle = 4
Processor id = 1 is executing task id = 1
Processor id = 2 is executing task id = 5

At cycle = 5
Processor id = 1 is executing task id = 1
Processor id = 2 is executing task id = 5

At cycle = 6
Task id = 3 has arrived
Ln 1, Col 1 100% Unix (LF) UTF-8
```

Figure 12 Output Example