Processor Execution Simulator

Atypon Software Engineering Internship

Yazan Yousef

A report to clarify the Processor Execution Simulator Assignment, defending against the clean code principles, and pointing to the design patterns and data structures used in the project.

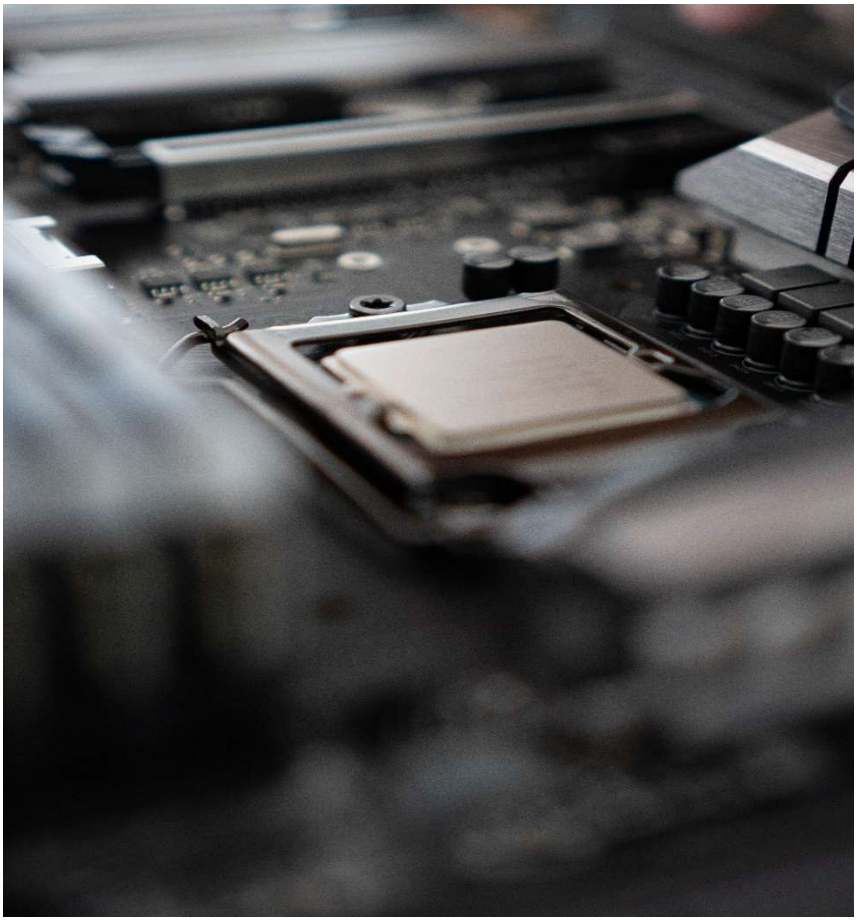


Table Of Contents

1 Introduction

The purpose of this project is to simulate the execution of tasks in a multi-processor system. The simulation utilizes a priority-based scheduling algorithm to assign tasks to processors, with the goal of executing tasks with the highest priority first. In case of a tie, the execution time of the task is used as a tie-breaker.

Each processor has a task assigned to it, with the task having a creation time (or clock cycle), execution time, and priority. The simulation will start considering each task after its creation time and will have a predefined number of clock cycles, processors, and tasks. The end result of the simulation will be a report showing the state of each processor at each clock cycle.

The Full UML Diagram :http://bitly.ws/A79x

2 Work Flow

The program starts by reading the input from the user, and initializing the tasks and processors.
Then, we start the simulation, and we run the simulation for predefined number of clock cycles.

In each clock cycle, we add tasks that has been created at that clock cycle to our priority queue, and then we assign the tasks to the available processors (which is not working on any task).
After that, we print the current state of the simulation, and then we go to the next cycle, we keep repeating till the end of clock cycles, after that we show the end result as a full time line that represents the state of each processor at each clock cycle.

3 Abbreviations And colors

Before we start the report I want to clarify some abbreviations and common colors meanings:

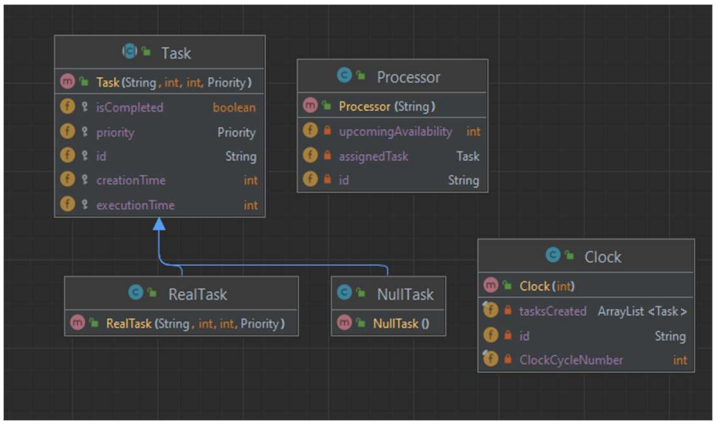
- Each **processor** has an id (P + the number of the processor) , and its color is blue.
- Each **task** has an id (T + the number of the task) , and its color is Green.
- Each **clock cycle** has an id (C + the number of the clock cycle) , and its color is yellow.
- The **null task** has an id always (X) , and its color is red.

- “Time” means clock cycle.

4 Defining Packages & Classes

In order to maintain a clear and organized structure, the project is separated into packages. Each package contains classes that are related to one another and support the principle of encapsulation.

4.1 MainComponents Package



This package contains the main classes of the project, which is used a lot in the project.
The classes in this package are:

4.1.1 Processor Class

This class represents a processor, it contains an Id as a string, and a task assigned to it.
It also have an upcoming availability, which is the next clock cycle that the processor will be available to work on a task on it.

4.1.2 Clock Cycle

The Clock class represents a clock cycle in the simulation of processor execution. This class is an important component in organizing the simulation and ensuring its accuracy. It contains a unique ID, represented as a string, that identifies each clock cycle.

One of the most crucial aspects of the Clock class is its ability to **store a list of tasks that are created at each clock cycle**.

This list is critical in determining which tasks should be considered for execution in a given cycle

For example, when the simulation reaches the third clock cycle (C3), the Clock class will provide access to the list of tasks that were created at the third cycle, which can be found within the C3 object. These tasks are added to the list after they are read and assigned to the corresponding Clock object.

By utilizing the Clock class, the simulation is able to efficiently and accurately keep track of the tasks created in each clock cycle, instead of iterating over the whole list of tasks after each clock cycle.

4.2 Task

This class is part from the **MainComponent** package, but I separated it for simplicity.

4.2.1 Task Abstract Class

This class represents a task, it contains a unique ID for each Task.
Every task has a creation time, an execution time and a priority (high or low).

4.2.2 RealTask Class

Its constructor takes as input an Id as a string, creation time and execution time and priority.

This is just a normal task, it's color is green (when we print it or when we do getID() method).

4.2.3 NullTask Class

This class is placeholder for null tasks objects, it means the processor doesn't have any task assigned to it.

Its constructor doesn't take anything, as it will hold default values for null tasks, for example the id = X.

4.3 Secondary Package

This package contains:

4.3.1 Priority Enumeration

It has two values HIGH and LOW, referring to the priority of the tasks.

4.3.2 ConsoleColors Class

This class contains the colors that are used in the project, and it is used to make the output more readable.

4.3.3 Recorder Class

This class is used to record the simulation, as a final report at the end of the simulation.

It uses a **HashMap** to store for each clock cycle, which task each processor was working on during that cycle, so it looks like this: **HashMap<Clock, HashMap<Processor, Task>>.** I used the hashmap because it's a very fast data structure, and it's easy to use.

Also, the recorder was used with the Scheduler class which will be discussed later, in short the scheduler will be responsible for iterating on all processors after each clock cycle, so it's convenient to put the Recorder object inside of it.

4.4 Collections Package

In this package we have two classes representing collection of clocks and Processor as follows:

4.4.1 TimeLine Class

This class is used to store all clocks objects, for example if the user entered 5 clock cycles, then we will have 5 clock objects stored in this class inside of an ArrayList data structure.

Why we need this class?
As we find ourselves looking to iterate over all clock cycles multiple times in our project, so to make it easier and shared
among the whole project **(Singleton)** I created this class.

4.4.2 ProcessorCollection Class

The same idea for processors we need to iterate over them multiple times in the project, so that I created this **singleton** class as well.

Note:

Both of the mentioned classes implements the **Iterable** interface, so we can iterate over them using **for each loop**, which means both of them will contain inner class defining its own iterator.

4.5 Simulation Package

This package will contain the classes that holds the simulation package, which are the Simulator and the Scheduler classes.

4.5.1 Scheduler Class

This class, the Scheduler, plays a crucial role in the Processor Execution Simulator. It's responsible for assigning tasks to processors at each clock cycle. The scheduling algorithm ensures that tasks with higher priority and longer execution time are executed first by using a priority queue.

Moreover, the Scheduler class is also equipped with the Recorder class, which records all the information about the task assignments, including assigning a new task, leaving the processor with the old task, or even a null task. The Recorder class then prints this information at the end of the simulation.

4.5.2 Simulator Class

This class is the main class of the project, it is responsible for running the simulation.
It collects all the project together and gives a nice way to run the simulation which follows the facade design pattern.

And it will work as follows:

1. It will prompt the user to enter the number of processors, and the number of clock cycles, and the path of the file that contains the list of tasks and their number.

2. It will create the TimeLine object by the number of clock cycles, and it will be filled by tasks, as each task will go to the clock cycle object which corresponds to its creation time (we mentioned previously that the clock class has a list of tasks that are created at that clock cycle).
3. It will create the Processors collection by the number of processors.
4. It will create the Scheduler object.
5. It will start the simulation, and it will run the simulation for predefined number of clock cycles (this is done using a for-each loop over the TimeLine object which holds all the clocks).
6. In each clock cycle, the Scheduler will first add tasks that has been created at that clock cycle to the priority queue, and then it will assign the tasks to the available processors, starting from the tasks with the highest priority and highest execution time.
7. Then, it will print the current state of the simulation, and then it will go to the next cycle.
8. After the simulation is done, it will print the final report using the recorder object.
9. It will print the final report which is inside the recorder object.

5 Used Data Structures & Time Analysis

The challenge was to keep tasks sorted by their priority and execution time during the simulation. Sorting the entire list of tasks one time was not feasible as new tasks would be added at each clock cycle. To solve this, I used a priority queue which can add and remove elements efficiently **(O(log T))** while keeping the tasks sorted.

I also used a **hashMap** inside the recorder, and **arrayList** inside of the collection package.

The simulation's time complexity is **O(C * P * log T)**, where C represents the number of clock cycles, P is the number of processors, and T is the number of tasks, and it's because we should iterate over **every clock cycle** and for each clock we should **check all processors** to check their state, and the **log T** is coming from adding and removing from the priority queue, as at the end we will be adding all **T** tasks.

6 Used Design Patterns:

1) Singleton Pattern:
The singleton design pattern was used in the **Collection package**, and the **Recorder** class, to make sure that there is only one instance of them in the project.
It also makes the code more organized by enforcing a clear and consistent way to access and manipulate those classes.

2) Iterator Pattern:
The Iterator Design Pattern is a behavioral design pattern that provides a standardized way of accessing elements within a collection of objects. It helps to make the code more clean and organized by abstracting the implementation details of the underlying data structure,
making it easier to iterate over the elements within a collection. The pattern was used in the **TimeLine** and **ProcessorsCollection** classes.

3) Null Object Pattern:
I used the null object pattern in the **NullTask** class.
This class serves as a placeholder for Null tasks, it means the processor doesn't have any task assigned to it, so it will return
X always, and it has some default values assigned to it.

4) Facade Pattern:
I used the facade design pattern in the Simulator class, to make the simulation easier to use, and to
hide the complexity of the project from the user, and to make the code more organized.

7 Definding Against SOLID principles

1) Single Responsibility Principle
I made sure that each class, package and/or method has only one responsibility, and it does it well.
2) Open/Closed Principle
I made sure that the code is open for extension, but closed for modification, for example to add a new task type we should only create a
new concrete class and implement the abstract methods without affecting the existing code.
3) Liskov Substitution Principle
I made sure that the
subclasses of the classes in the project can be used in place of the parent class without affecting the correctness of the program.
4) Interface Segregation Principle
There is no interface or
abstract class that forces it’s subclasses to implement methods they can’t hold.
5) Dependency Inversion Principle
I made sure that the high-level modules do not depend on the low-level modules, but they both depend on abstractions.
So in each we don't depend on the concrete task class instead we depend on the abstract task class.