Processor Execution Simulator

By Hamza Alhalabi

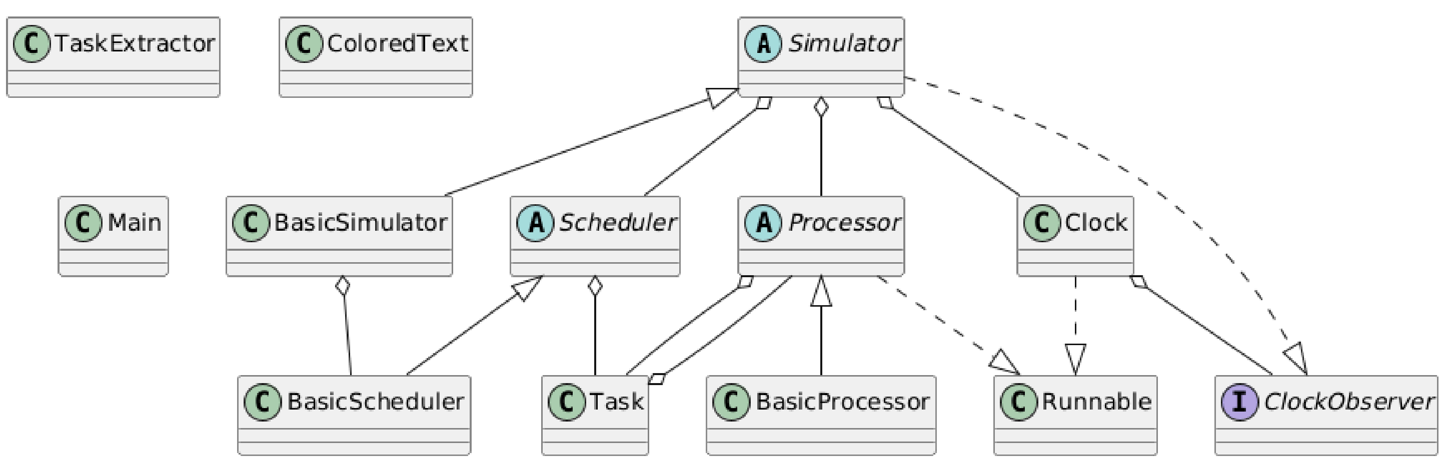
Table of Contents

1. Introduction

In this report, I will present the design and implementation of a Processor Execution Simulator that models how tasks are scheduled and executed across multiple processors simultaneously.

I started with planning the design and relationships between given classes and figuring out how the classes will work together and be capable of extending with different behaviours when needed.

This is the simplified UML diagram that shows the high-level relationships between classes:



2. System Design

2.1 Architecture Overview

The simulator uses six main classes:

- **Main**: Entry point that receives arguments, instansiate an object of concrete `Simulation` class, and call `startSimulation()` to start the whole simulation

```
public class Main {
    public static void main(String[] arguments) {
        Simulator simulatorOne = new BasicSimulator(Integer.parseInt(arguments[0]), Integer.parseInt(arguments[1]), arguments[2]);
        simulatorOne.startSimulation();
    }
}
```

- **Simulator**: Core component that manages the simulation process it is an **abstract** class that need to be extended to implement different simulation algorithms and rules. It has associations with `Clock` and `Scheduler` classes as single instances, and with `Task` and `Processor` as **lists** of intances.

  `Simulator` class implements the interface `ClockObserver` which is part of the observer design pattern that is used in `Clock` to manage its observers.

```
public abstract class Simulator implements ClockObserver{
    protected int numOfProcessors;
    protected ExecutorService processorPool;
    protected List<Processor> processors;
    protected int maxCycle;
    protected int currentCycle;
    protected List<Task> tasks;
    protected Clock clock;
    protected Scheduler scheduler;
```

- **Clock**: A class that provides timing for the simulation so a report can be printed every clock cycle. `Clock` class implements `Runnable` interface, that gives it the ability to run in a distinct thread.

```
public class Clock implements Runnable {
    private static volatile Clock singletonInstance;
    private int currentCycle = 1;
    private final int maxCycle;
    private final List<ClockObserver> observers = new ArrayList<>();
```

  `Clock` follows the **singleton** design pattern, it ensure that the system contains only one clock instance at a time, this help us organize the operations of clock cycles and protect us of problems.

```java
private Clock(int maxCycle){
    this.maxCycle = maxCycle;
}

public static Clock getInstance(int maxCycle) {
    if (singletonInstance == null) {
        synchronized (Clock.class) {
            if (singletonInstance == null) {
                singletonInstance = new Clock(maxCycle);
            }
        }
    }
    return singletonInstance;
}
```

I also added a reset method to remove the singleton instance in case I want to run different simulation one after another.

```java
public static void reset() {
    synchronized (Clock.class) {
        singletonInstance = null;
    }
}
```

I used the **observer** design pattern to manage the communication between clock and simulations (if many simulations were working together), `Clock` is the **subject** and other observers are from type `ClockObserver`.

```java
public void addObserver(ClockObserver observer) { observers.add(observer); }

public void removeObserver(ClockObserver observer) { observers.remove(observer); }

private void notifyObservers() {
    for (ClockObserver observer : observers) {
        observer.onClockTick(currentCycle);
    }
}
```

In the `run()` method (which is the entry point for threads) I wrote a loop that keep iterating until the specified time is done, I added a one second delay to simulate a human-level cycle of time. At first, I used *1000 milliseconds* but found that it causes inaccurate results because of context switching with many processors who works also in different threads, so I kept it *900 milliseconds* to keep the result organized by time.

```java
@Override
public void run() {
    while (currentCycle <= maxCycle) {
        notifyObservers();
        try {
            Thread.sleep( millis: 900); // Simulate a cycle duration of 1 second
        } catch (InterruptedException e) {
            System.out.println("Clock interrupted: " + e.getMessage());
        }
        currentCycle++;
    }
}
```

- **ClockObserver**: This interface represents classes that need to observe clock cycle from `Clock`.

```java
public interface ClockObserver {
    void onClockTick(int currentCycle);
}
```

- **Scheduler**: Assigns created (ready) tasks to processors based on priority rules and processors availability. I designed it to have 2 queues not one, one for high priority tasks and the other for normal tasks, both queues implementations utilize `PriorityQueue` Java collection so it takes the task with the longest execution time from every queue in `O(log(n))` time.

```java
public abstract class Scheduler {
    protected PriorityQueue<Task> highPriorityQueue =
            new PriorityQueue<>(Comparator.comparingInt((Task t) -> t.getExecutionTime()).reversed());
    protected PriorityQueue<Task> lowPriorityQueue =
            new PriorityQueue<>(Comparator.comparingInt((Task t) -> t.getExecutionTime()).reversed());

    protected Scheduler() {
    }
```

I kept the `Scheduler` class **abstract** so it can be extended with different scheduling rules. I created `BasicScheduler` as a concrete `Scheduler` class and add the assignment rules to it.

```java
public class BasicScheduler extends Scheduler{

    public BasicScheduler(){
        super();
    }

    @Override
    public void scheduleTask(Task task) {...}

    @Override
    public void assignTasks(List<Processor> processors,
                            ExecutorService processorPool) {...}

    @Override
    public void generateQueueReport(){...}
}
```

The Scheduler in my case uses `ExecutorService` (which is passed from `Simulator` class as an argument) to manage processors (as each processor is a `Runnable`) which keep everything organized and managable and gives high flexibility.

```java
public void assignTasks(List<Processor> processors,
                        ExecutorService processorPool) {
    while (true){
        Processor processor = getProcessor(processors);
        if(processor == null){
            break;
        }
        if (!highPriorityQueue.isEmpty()) {
            Task task = highPriorityQueue.poll();
            processor.assignTask(task);
            processorPool.submit(processor);
        } else if (!lowPriorityQueue.isEmpty()) {
            Task task = lowPriorityQueue.poll();
            processor.assignTask(task);
            processorPool.submit(processor);
        }
        else break;
```

- **Task**: Represents individual tasks with their properties (creation time, execution needed time, and high priority or not)

```java
public class Task {
    private final String id;
    private final int creationTime;
    private int executionTime;
    private final boolean highPriority;
    private boolean Completed;
    private Processor assignedProcessor;
```

I used a static factory method for creating new instances of `Task` instead of using the `new` operator.

```java
private Task(String id, int creationTime, int executionTime, boolean highPriority) {
    this.id = id;
    this.creationTime = creationTime;
    this.executionTime = executionTime;
    this.highPriority = highPriority;
    this.Completed = false;
}

public static Task createTask(String id, int creationTime, int executionTime, boolean highPriority){
    return new Task(id, creationTime, executionTime, highPriority);
}
```

- **Processor**: Models processors that execute assigned tasks following a predefined set of rules. I created an **abstract** class called `Processor` so we can extend more prcessing algorithms in the future.

```java
public abstract class Processor implements Runnable{
    protected String id;
    protected Task currentTask;
    protected final AtomicBoolean available;

    protected Processor(String id) {...}

    public boolean isAvailable() { return available.get(); }

    public void assignTask(Task task){...}
```

In the concrete `Processor` class that follows the same rules as the assignment (I named it `BasicProcessor`), I implemented the process operation in `execute()` method by letting the processor iterate for the number of execution cycles and wait for 1 second (*1000 milliseconds*) every time to simulate the process real-world processors.

```java
protected void execute() throws InterruptedException{
    synchronized (this){
        executeCycle();
        while (!currentTask.isCompleted()) {
            printColoredText("Task " + currentTask.getId() + " remaining time: " + currentTask.getExecutionTime()
            executeCycle();
            Thread.sleep(1000);
        }
    }
    currentTask.removeProcessor();
}

private void executeCycle() throws InterruptedException{
    currentTask.executeCycle();
```

- **TaskExtractor**: This class is a **utility** class contains a single **static** method that takes a file path and returns a list of tasks extracted from the input file in a specifec format

```java
public class TaskExtractor {
    public static List<Task> extract(String filePath){
        List<Task> tasks = new ArrayList<>();
        try (BufferedReader br = new BufferedReader(new FileReader(filePath))) {
            int numOfTasks = Integer.parseInt(br.readLine()); // read number of tasks

            for (int task = 1; task <= numOfTasks; task++) {
                String[] taskArguments = br.readLine().split(" ");
                int creationTime = Integer.parseInt(taskArguments[0]);
                int executionTime = Integer.parseInt(taskArguments[1]);
                boolean highPriority = Integer.parseInt(taskArguments[2]) == 1;
                tasks.add(Task.createTask(("T" + task), creationTime, executionTime, highPriority));
            }
        } catch (FileNotFoundException e) {
            System.err.println("Error: File not found - " + filePath);
            return Collections.emptyList();
        } catch (IOException e) {
            System.err.println("Error reading file: " + e.getMessage());
            return Collections.emptyList();
        }

        return tasks;
    }
```

- **ColoredText**: This utility class is used to enable printing on console with colored text, it encapsulates the logic and organizes the process instead of writing alot of code every time.

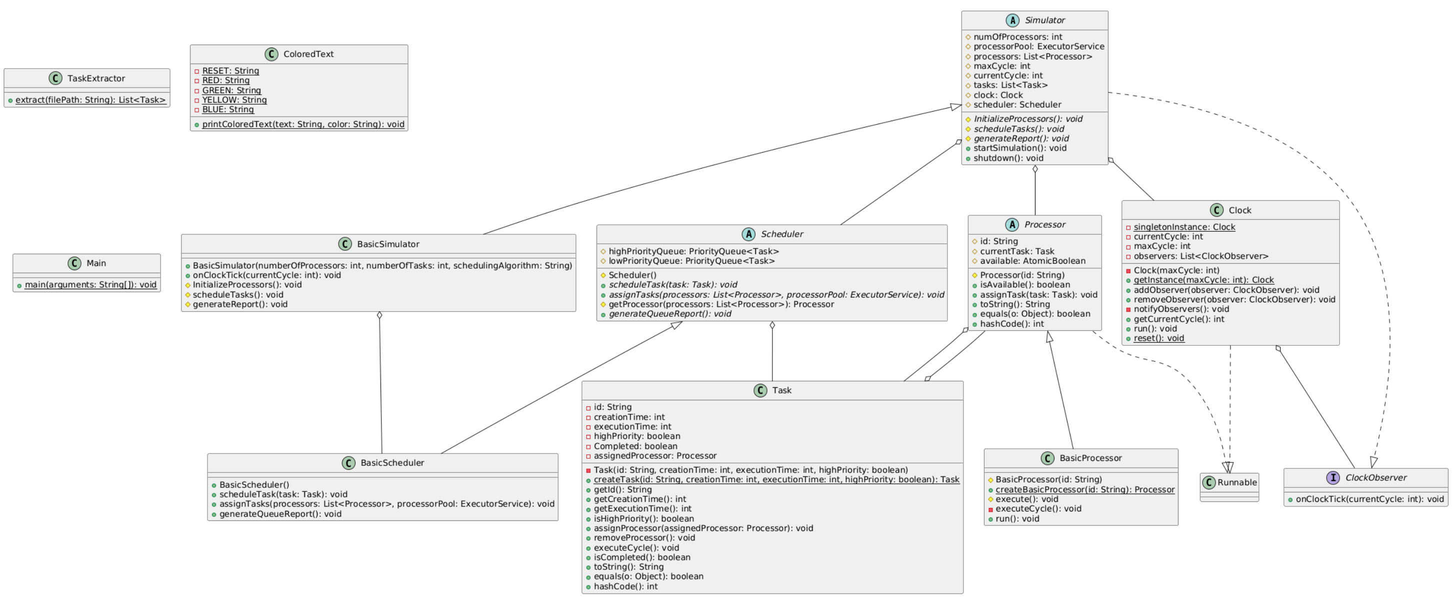
  I import the needed method using `import static Class.method` and use it by its name directly in the class.

```java
public class ColoredText {
    private static final String RESET = "\u001B[0m";
    private static final String RED = "\u001B[31m";
    private static final String GREEN = "\u001B[32m";
    private static final String YELLOW = "\u001B[33m";
    private static final String BLUE = "\u001B[34m";
```

```java
public static void printColoredText(String text, String color) {
    switch (color.toLowerCase()) {
        case "red":
            System.out.println(RED + text + RESET);
            break;
        case "green":
            System.out.println(GREEN + text + RESET);
            break;
        case "yellow":
            System.out.println(YELLOW + text + RESET);
            break;
        case "blue":
            System.out.println(BLUE + text + RESET);
            break;
        default:
            System.out.println(text);
    }
}
```

2.2 UML Class Diagram

This is the UML diagram of the whole project, I drew it using a websit called PlantUML.

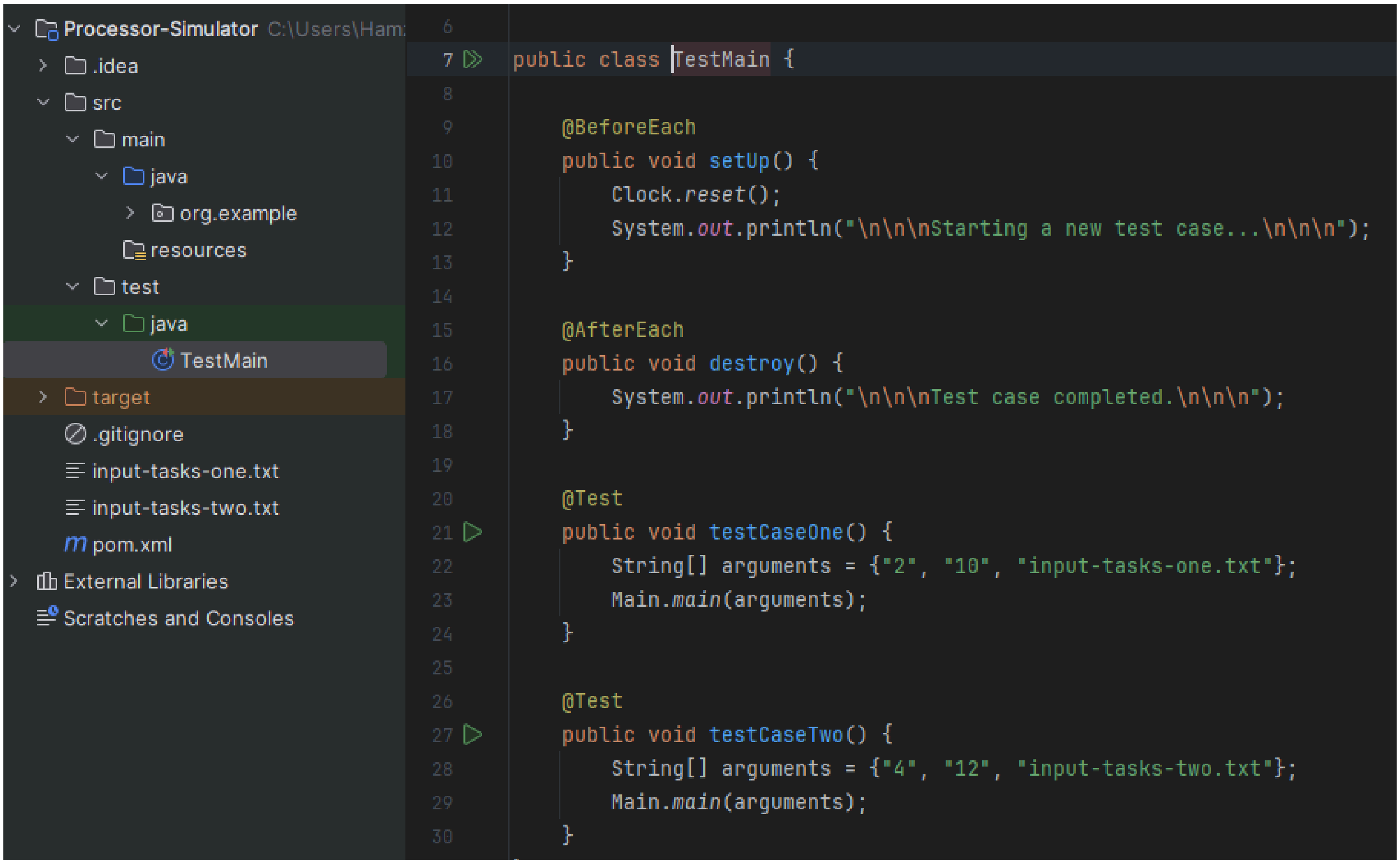


2.3 Design Approach

I tried as possible to build a design that's following object-oriented principles with clear separation of responsibilities, each class handles a specific aspect of the simulation, making the system modular and extensible for future enhancements.

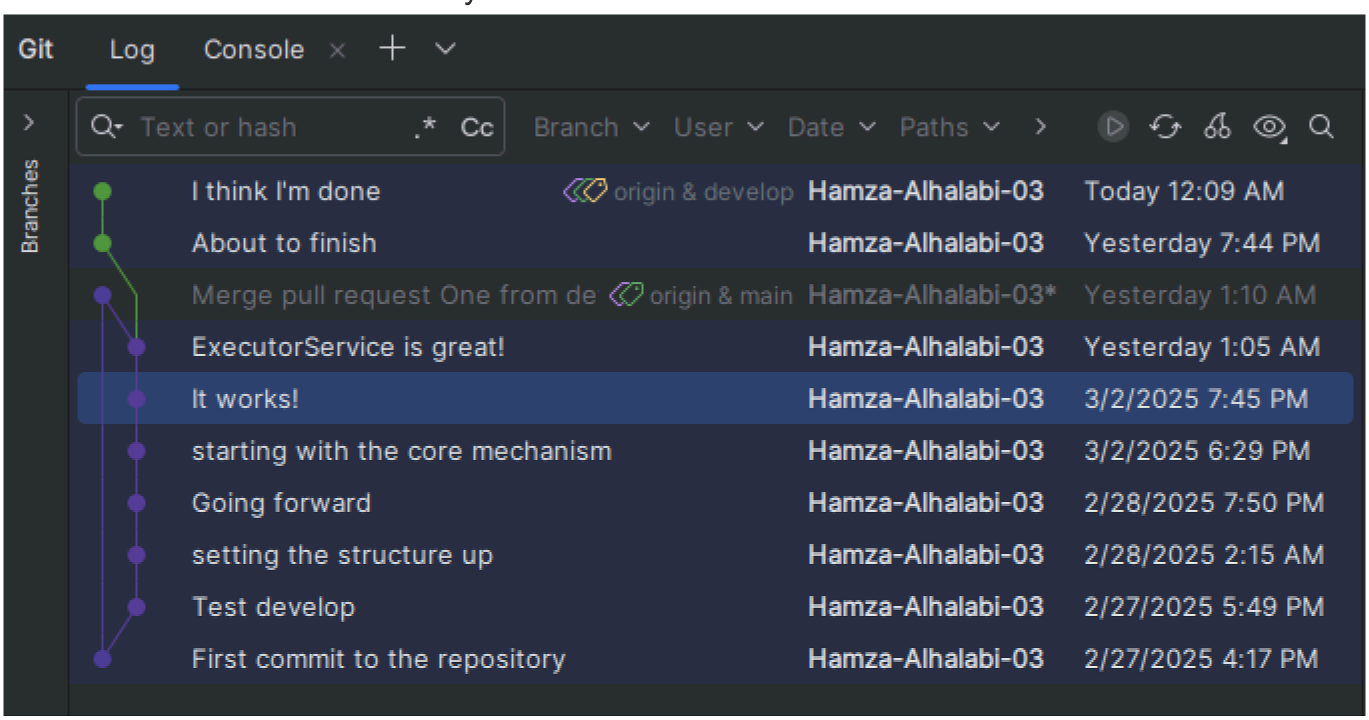
I wrote the classes `Simulator` , `Scheduler` , and `Processor` to be extensible with other solutions and algorithms.

An interesting point is that I used **Maven** to build the project and manage dependencies like `JUnit` .



```
Starting a new test case...




****************************************
Current Cycle: C1
****************************************
Create task: T1
Create task: T2
Create task: T3
Processor P1 started processing: T1
Processor P2 started processing: T2
Task T1 remaining time: 8
Task T2 remaining time: 4
Processor P3 started processing: T3
Task T3 remaining time: 3

****************************************
Current Cycle: C2
****************************************
Task T3 remaining time: 2
Task T2 remaining time: 3
Task T1 remaining time: 7

****************************************
```

I used Git also in an effective way with the IDE GUI.



3. Implementation Details

3.1 Scheduling Algorithm

The scheduler prioritizes tasks based on:

1. Task priority (high over low)
2. Execution time (longer tasks first)
3. When both previous priority are equal choose randomly
4. We assign ready tasks to available processors using `ExecutorService` mechanism
5. I left a space for extending the abstract class and adding different behaviour to the scheduler

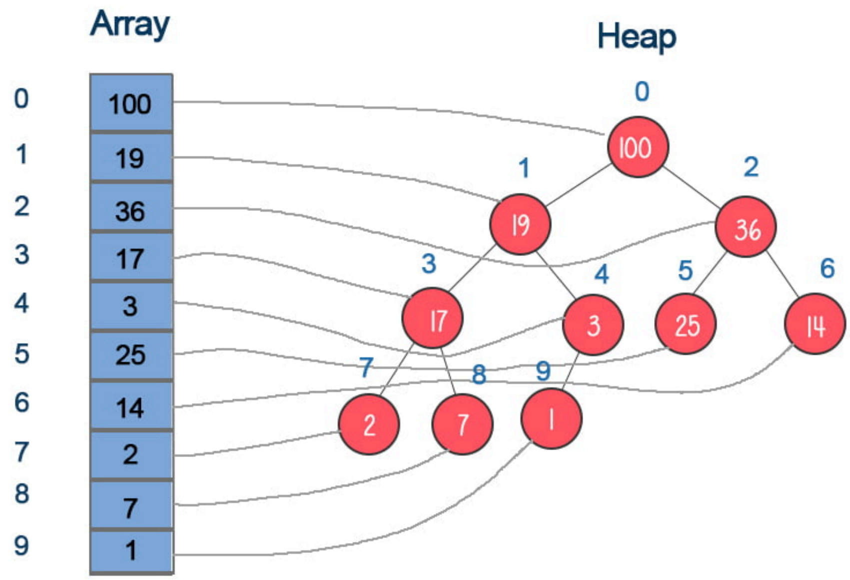
3.2 Simulation Process

For each clock cycle, the simulator:

1. Creates new tasks and send them to the scheduler to add them in queues
2. Assigns waiting tasks to available processors
3. Updates running tasks' progress
4. The printing report was done using decentralized approach, each task or processor printed what is needed in the correct time without contacting with the `Simulator` class everytime

3.3 Key Data Structures

- Task queue for managing waiting tasks using priority queue (max heap)



- Processor array for tracking processor states using array list
- Task list for monitoring all tasks in the system using array list also

4. Conclusion

This assignment was a great experience for me, this was the first time for me building a real application using threads and multithreading, sometimes I felt stuck and overwhelmed with so many details, but at the end it was very helpful.