ATYPON

Processor Simulator Report

by Hashim Emad

Table of Contents

Problem Statement

The goal is to build a simulator that simulates processor execution for processes (i.e., tasks). The following notes provide key details about the processor execution:

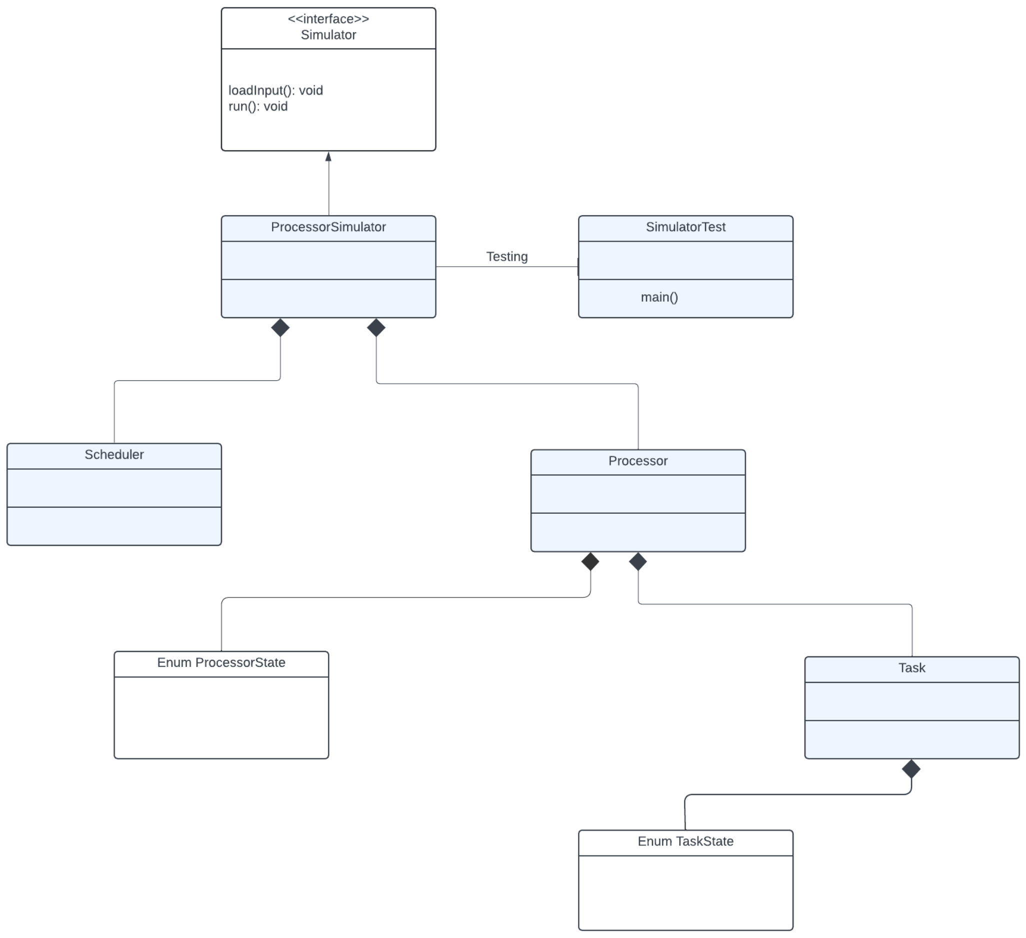
- A processor can only execute one task at a time.
- The number of processors is fixed during the entire simulation.
- All processors are synchronized, meaning they all operate on the same clock cycle.
- Tasks, Processors and clock cycles and are assigned unique IDs sequentially, such as T1, T2, T3, T4, etc.
- **Tie-Breaking:** When multiple tasks are eligible for execution, a tie-breaking mechanism must prioritize tasks based on specific criteria first based on priority, then execution time then requested time.
- **(BONUS) Interruption:** Low-priority tasks can be interrupted if a high-priority task becomes available, ensuring that high-priority tasks are executed as soon as possible.

Design Objectives

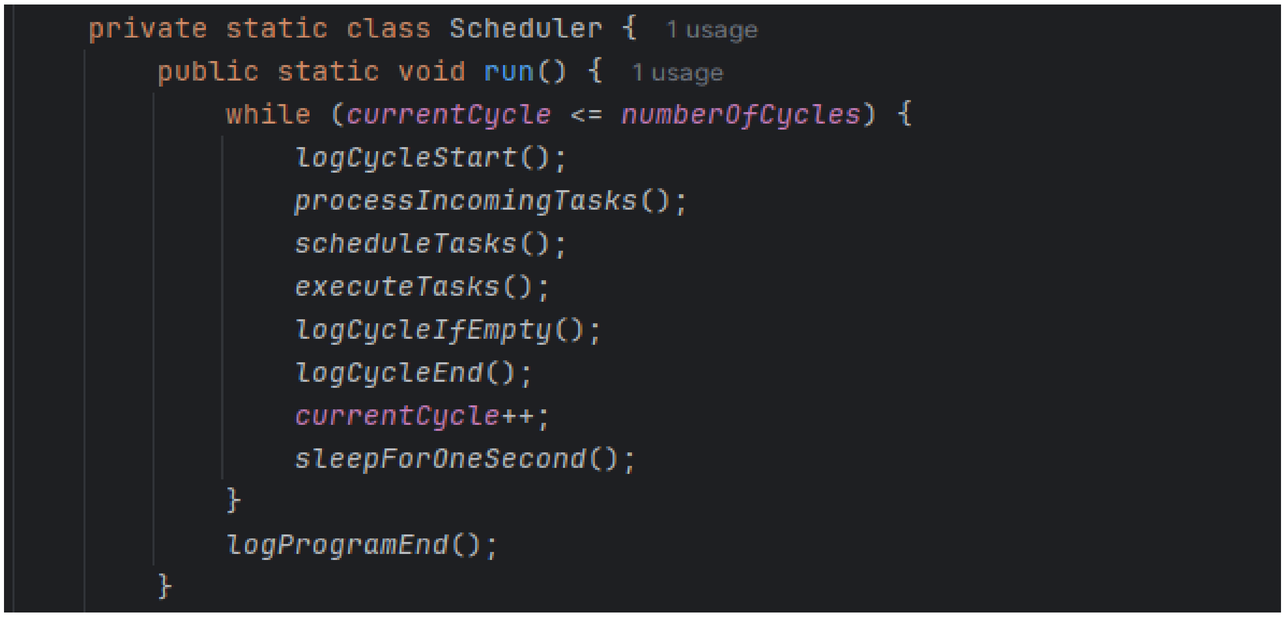
- **Apply SOLID Principles:** The design should adhere to the SOLID principles to ensure a robust and maintainable system architecture.
- **Apply Singleton Design Pattern When Needed:** The Singleton pattern should be applied where a single instance of a class is required, ensuring controlled access to shared resources
- **Flexibility:** The design should allow easy modification and extension of the simulation's functionality.
- **Extensibility:** The system should support the addition of new features, such as different scheduling algorithms or additional task properties, without requiring major changes to the existing codebase.

- **Maintainability:** The code should be organized and written in a way that makes it easy to understand, debug, and update. This includes adhering to best practices in object-oriented design and programming.

UML Diagram



High-Level Explanation of Algorithms



```
    private static class Scheduler {  1 usage
        public static void run() {  1 usage
            while (currentCycle <= numberOfCycles) {
                logCycleStart();
                processIncomingTasks();
                scheduleTasks();
                executeTasks();
                logCycleIfEmpty();
                logCycleEnd();
                currentCycle++;
                sleepForOneSecond();
            }
            logProgramEnd();
        }
```

Scheduler().run() Method:

The Scheduler.run() method orchestrates the entire simulation by coordinating the actions of processors and tasks within each clock cycle. The method follows these steps:

1. A while loop runs until the current cycle exceeds the total number of cycles defined (currentCycle <= numberOfCycles).

2. The logCycleStart() method is called to output the beginning of the current cycle, providing visibility into the simulation's progress.

3. The processIncomingTasks() method identifies and processes tasks that are scheduled to arrive at the current cycle. These tasks are then moved to the waiting queue.

4. The scheduleTasks() method prioritizes tasks in the waiting queue based on predefined criteria (e.g., priority, requested time, and ID). This ensures that tasks are assigned to processors according to their importance and requirements.

5. The executeTasks() method directs processors to execute their assigned tasks, advancing the simulation by one cycle of work for each active task.

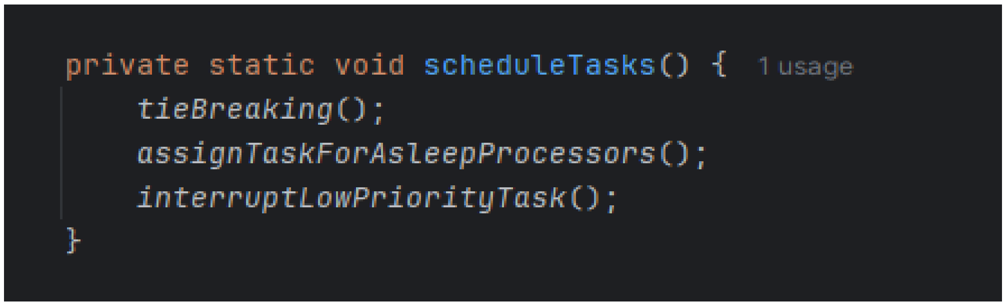
6. The logCycleIfEmpty() method checks if all tasks are completed and if all processors are idle. If so, it logs that all tasks have finished execution.

7. The logCycleEnd() method is called to output the end of the current cycle, marking the completion of one simulation cycle.

8. The currentCycle counter is incremented to move to the next simulation cycle.

9. The sleepForOneSecond() method introduces a one-second delay between cycles to simulate the passage of time realistically.

10. After exiting the loop, the logProgramEnd() method is called to indicate the end of the simulation, marking the completion of all defined cycles.



```
private static void scheduleTasks() {  1 usage
    tieBreaking();
    assignTaskForAsleepProcessors();
    interruptLowPriorityTask();
}
```

ScheduleTasks() method:

The scheduleTasks() method is responsible for managing the prioritization and assignment of tasks to processors. It ensures that tasks are scheduled efficiently and that high-priority tasks are executed as soon as possible. The method follows these steps:

1. **Tie-Breaking**:
   - The tieBreaking() method sorts the waiting tasks based on predefined criteria to resolve any conflicts when multiple tasks are eligible for execution. The criteria include:
     - **Priority**: Higher-priority tasks are given precedence.
     - **Requested Time**: Tasks with longer requested times are prioritized if their priorities are the same.
     - **Task ID**: When tasks have the same priority and requested time, the task with the lower ID (i.e., the one that arrived earlier) is prioritized.
2. **Assigning Tasks to Idle Processors**:
   - The assignTaskForAsleepProcessors() method assigns tasks from the waiting queue to processors that are currently idle. This ensures that all available processing power is utilized.
   - The method iterates through the list of processors, and for each idle processor, it assigns the highest-priority task from the waiting queue.
   - Once a task is assigned, it is removed from the waiting queue and marked as executing.

3. **Interrupting Low-Priority Tasks**:
    - The interruptLowPriorityTask() method checks if any processors are executing low-priority tasks when there are high-priority tasks waiting to be executed.
    - If a high-priority task is found, it preempts the low-priority task currently being executed by a processor.
    - The preempted low-priority task is returned to the waiting queue, and the high-priority task is assigned to the processor.
    - This ensures that high-priority tasks are executed as soon as possible, maintaining the system's responsiveness to critical tasks.

These steps ensure that tasks are scheduled efficiently and according to their importance, maximizing processor utilization while respecting task priorities.

Object-Oriented Design Considerations

Flexibility and Reusability

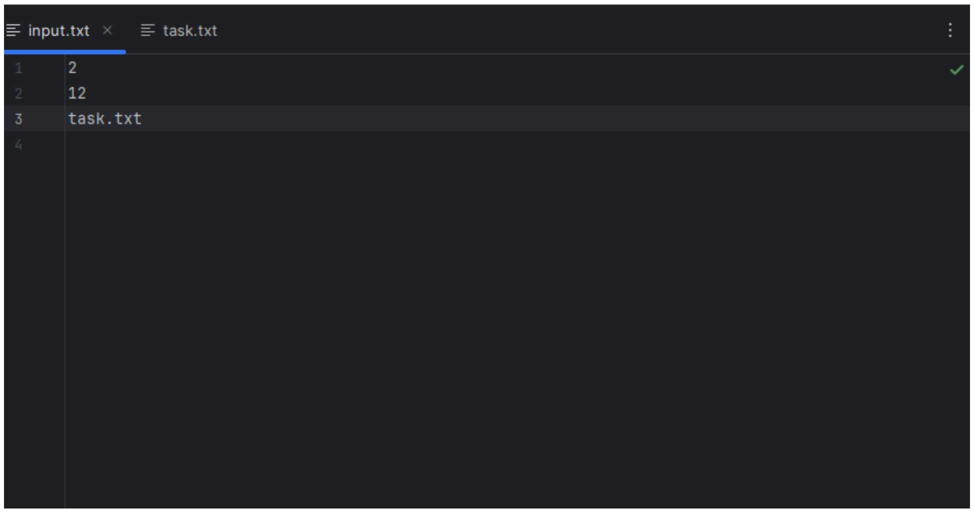
- Singleton Pattern: Used for ProcessorSimulator to ensure a single instance controls the simulation.
- Interfaces: Defined for simulation control to allow easy extension or replacement of simulation logic.
- Enumerations: Used for processor and task states to ensure type safety and readability.
- Modular Design: Each class has a clear responsibility, making it easy to extend or modify individual components without affecting others.

Future Extensions

- Dynamic Processor Management: Add or remove processors during the simulation.
- Advanced Scheduling Algorithms: Implement different scheduling strategies as separate classes.
- Task Dependencies: Handle tasks with dependencies or constraints on execution order.
- Real-Time Simulation: Incorporate real-time constraints and deadlines for tasks.

Input Sample

Inside a file called input.txt:



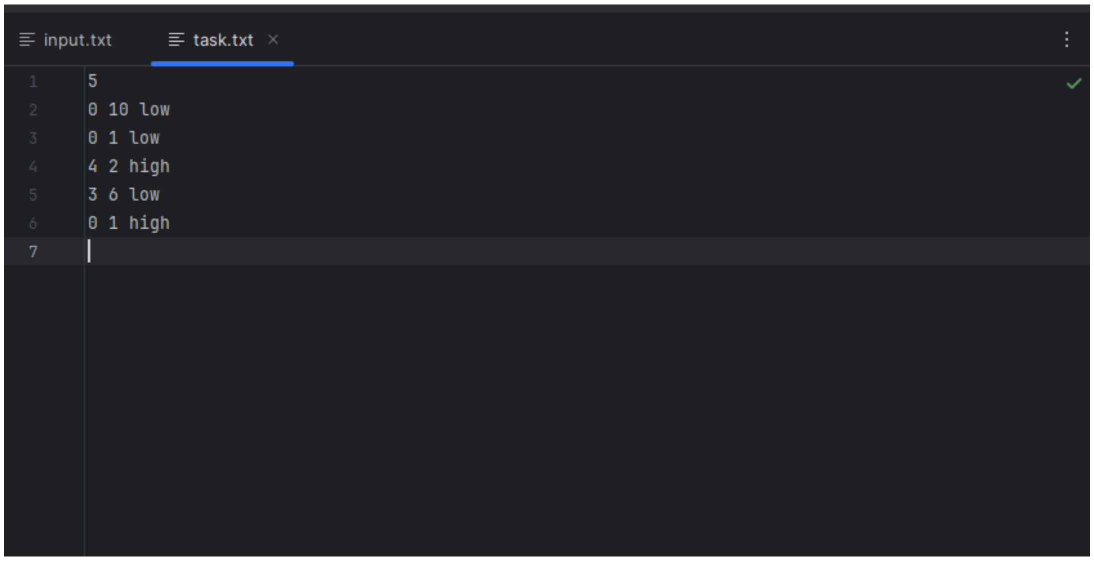
```
2
12
task.txt
```

Number of processors

Number of cycles

Task file path

Inside your task file:



```
5
0 10 low
0 1 low
4 2 high
3 6 low
0 1 high
```

Number of tasks

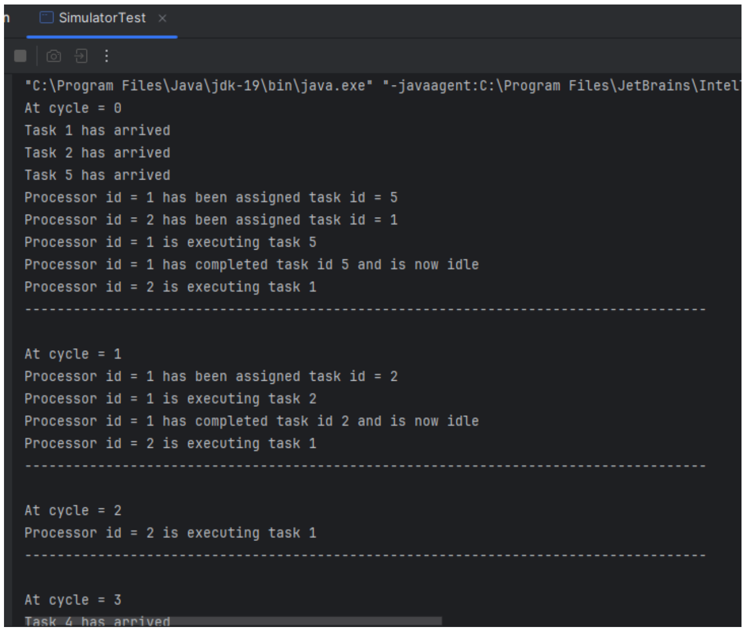
creationtime1 requestedtime1 priority1

creationtime2 requestedtime2 priority2

creationtime3 requestedtime3 priority3

creationtimeN requestedtimeN priority

output will be in console:



```
"C:\Program Files\Java\jdk-19\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\Intel
At cycle = 0
Task 1 has arrived
Task 2 has arrived
Task 5 has arrived
Processor id = 1 has been assigned task id = 5
Processor id = 2 has been assigned task id = 1
Processor id = 1 is executing task 5
Processor id = 1 has completed task id 5 and is now idle
Processor id = 2 is executing task 1
------------------------------------------------------------------------------------

At cycle = 1
Processor id = 1 has been assigned task id = 2
Processor id = 1 is executing task 2
Processor id = 1 has completed task id 2 and is now idle
Processor id = 2 is executing task 1
------------------------------------------------------------------------------------

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

At cycle = 3
Task 4 has arrived
```