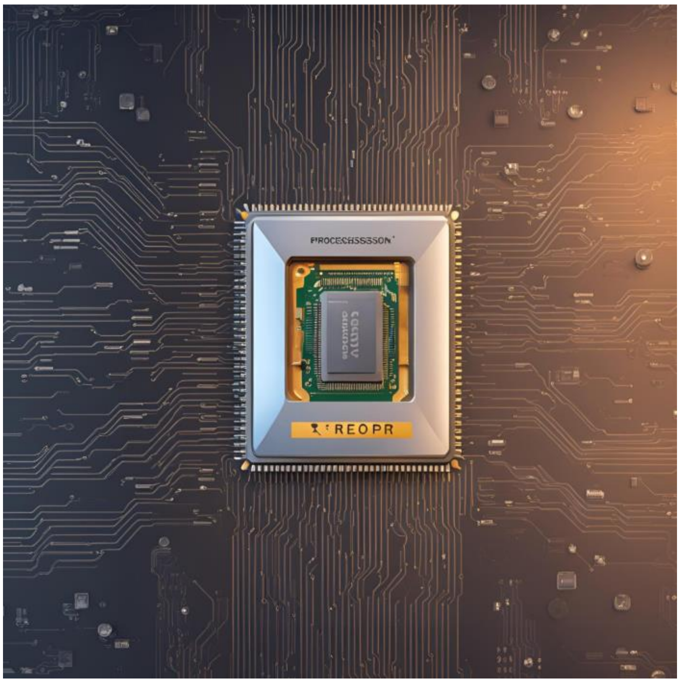

TASK SCHEDULING SIMULATOR

ABSTRACT

The Task Scheduling Simulator is a Java application that simulates task scheduling in a multi-processor environment. It reads tasks from an input file and assigns them to processors based on priority and execution time over a set number of clock cycles.

NOUREDDIN MOHAMMAD AHMAD LUTFI

Atypon Java & DevOps

Outline: Task Scheduling Simulator Report

1. Introduction
2. Software Design Overview
    - Architecture
        - Simulator
        - Scheduler
        - Processor
        - Task
        - TaskComparator
        - TaskReader
        - Clock (Utility Static)
        - Color (Enum)
    - Data Structures Used in the Simulator
        - Priority Queue (Tasks)
        - HashMap (CycleTasks)
        - Lists and Queues (IdleProcessors and BusyProcessors)
        - BufferedReader (br)
3. Simulation Process (Code Logic)
    - Main Class
    - Simulator Class
    - Processor Class
    - TaskReader Class
    - Task Class
    - Clock Class
    - TaskComparator Class
4. UML Diagram
    - Visual representation of class relationships and interactions.
5. Efficient Design: Cohesion and Coupling
    - High Cohesion
    - Low Coupling
    - Reason for Coupling in the Simulation Class
6. Exception Handling
7. Debatable Cases Analysis
    - Queue & ArrayList for Processors vs simple combine list for both
    - Using Observer Design Pattern
8. Future Extensions
    - Modular Design for Extensibility
    - Example: Interrupted Tasks Handling
    - Summary of Changes needed
9. Scenario: Real-time Task Prioritization
10. Conclusion

Introduction

Effective task scheduling is crucial for maximizing efficiency in multi-processor systems. The Task Scheduling Simulator, developed in Java, provides a platform to simulate and analyze different scheduling strategies. By dynamically assigning tasks based on priority and execution time, the simulator aims to optimize resource utilization and system responsiveness. This report explores the simulator's design, algorithms utilized, simulation process, and potential enhancements, offering insights into effective task management strategies in complex computing environments.

Software Design Overview

❖ Architecture

The simulator employs several **algorithms** and **data structures** to manage and schedule tasks effectively:

- **Simulator:** Orchestrates the simulation by **initializing** the environment and **coordinating interactions** between tasks, processors, and the scheduler.
- **Scheduler:** Handles task allocation among available processors using a **priority queue**. Tasks are prioritized based on **priority level** and **execution time** to optimize system responsiveness and processor utilization.
- **Processor:** Represents computing units capable of executing tasks. Processors manage their workload **dynamically**, transitioning between task execution and idle states based on **scheduling decisions**.
- **Task:** Encapsulates task attributes such as creation time, execution duration, and priority. Tasks are managed in a **priority queue** and processed based on their **priority level** and **execution time**.
- **TaskComparator:** A custom comparator used in the priority queue to prioritize tasks based on priority level and execution time. Ensures **high-priority and shorter tasks are executed sooner**.
- **TaskReader:** The TaskReader class is responsible for reading tasks from a **file** and organizing them into a HashMap for use in the simulation.
- **Clock:** The Clock class is a **utility class** (static class) that contains the **simulation time** and **cycle duration**. It provides methods to control the simulation's time progression and **synchronize** task execution cycles and it **only contains static members and methods**.
- **Color:** The Color enum provides **console color coding** for **output clarity**. It enhances the readability of simulation logs by distinguishing different types of using **ANSI** escape sequences.

❖ Data Structures Used in the Simulator

- **Priority Queue (Tasks):** Organizes tasks awaiting execution based on **priority and estimated execution time**. This structure ensures efficient retrieval and management of tasks, prioritizing **higher priority and shorter execution times** for prompt execution.
- **HashMap (CycleTasks):** Stores **tasks** mapped to their **creation time cycles** after we read them by TaskReader.
- **Lists and Queues (IdleProcessors and BusyProcessors):**
  - **IdleProcessors** is implemented as a **Queue** to efficiently assign tasks by handling processors in a **FIFO** manner. Hence, we **don't care** about mapping **any** idle process to **any** available task.
  - **BusyProcessors** is managed using an **ArrayList** for **direct access** to processors currently executing tasks, facilitating efficient tracking and management of task execution states.
- **BufferedReader (br):** Reads task data from an external file specified at runtime.

Simulation Process (Code Logic)

This process outline emphasizes how tasks are managed and scheduled based on their creation times using a **HashMap**, facilitating efficient organization and retrieval of tasks scheduled to start at specific simulation cycles.

❖ Main Class

The Main class serves as the **entry point** for the application. It **validates** input arguments, **initializes** the number of processors and total clock cycles, and **creates** a Simulator instance to **run** the simulation.

❖ Simulator Class

The Simulator class **orchestrates** the entire simulation process, handling tasks, processors, and scheduling. Key components include:

1. **Initialization**
   - The simulation begins by initializing parameters such as the number of processors, total simulation time, and the file path containing task data.
   - Tasks are read from the file and organized based on their creation times in a **HashMap**, where each key represents a **cycle time** and maps to a **list of tasks scheduled to start at that time**.
2. **Cycle Management**
   - The simulator operates in cycles, starting from the initial cycle up to the defined simulation time.
   - Each cycle represents a unit of simulated time where tasks are scheduled, and processors execute assigned tasks.

3. **Task Scheduling**

   At each cycle we update our **priority queue** with new tasks bases on **HashMap** that contains the **tasks tied to the current cycle**.

4. **Processor Execution**

   Assigned tasks are executed by processors, progressing through their execution cycles until **completion**. Processors transition between idle and busy states as tasks are assigned, executed, and completed according to scheduling priorities.

5. **Cycle Progression**

   The simulation continues until the predefined total simulation time is reached. Throughout the process, the simulator monitors and **logs task execution, processor states, and system performance metrics for analysis and evaluation**, the process continues until the simulation time is reached.

❖ Processor Class

The Processor class represents an **individual processor** capable of **executing tasks**. Key functionalities include:

- **Task Assignment:** Each processor can be assigned a task.
- **Task Execution:** The processor executes the assigned task, progressing through its execution **cycle by cycle**.
- **State Management:** The processor **transitions between idle and busy** states based on **task assignment and completion**. Once a task is completed, the processor returns to the idle pool, ready for the next task.

❖ TaskReader Class

The TaskReader class is responsible for **reading and validating tasks** from an **external file**. Key functionalities include:

- **File Reading:** **Reads** tasks data from the specified file path.
- **Task Validation:** **Ensures** that each task has valid parameters, such as creation time, execution time, and priority.
- **Task Organization:** Stores tasks in a **HashMap**, organizing them by their creation cycle for efficient retrieval during the simulation.

❖ Task Class

The Task class **encapsulates** the **properties** and **behaviours** of a task. Key attributes include:

- **Task ID:** A **unique identifier** for the task.
- **Creation Time:** The simulation cycle when the task is **created**.
- **Execution Time:** The total time required to **complete** the task.
- **Priority:** The **priority** level of the task.

- **Remaining Time:** The **time left** to complete the task, decremented with each execution cycle.

❖ Clock Class

The Clock class **manages** the **simulation's timing** and **cycles**. It is a **utility class** with **static methods** and **members**. Key functionalities include:

- **Cycle Tracking:** Keeps track of the **current** simulation cycle.
- **Time Progression:** Provides methods to **increment** the cycle and **retrieve** the current cycle.

❖ TaskComparator Class

The TaskComparator class defines the **priority** logic for tasks in the priority queue. Key functionalities include:

- **Priority Comparison:** **Compares** tasks based on their **priority level** and **execution time**.
- **Queue Management:** Ensures that tasks with higher priority and shorter execution times are prioritized for execution.

UML Diagram

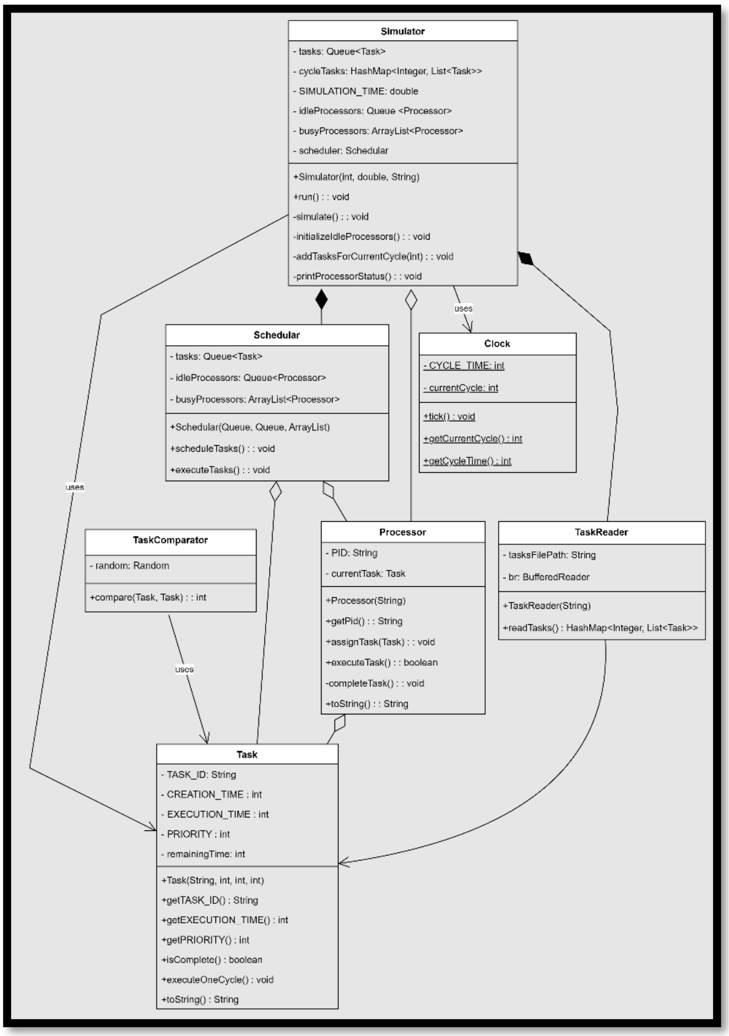

Figure 1. UML Diagram

Efficient Design: Cohesion and Coupling

❖ High Cohesion

High cohesion means that a class or module performs a **single task** or a **group of very related tasks**. High cohesion is evident through the following:

1. **Specialized Classes:**
   - Main**:** Handles **initialization** and **argument parsing**.
   - Simulator**:** Manages the **overall simulation**, including **processor initialization** and **task scheduling**.
   - Schedular**:** Responsible for **task assignment** and **execution**.
   - Processor**:** Represents **processors executing** tasks.
   - Task**:** Encapsulates **task properties** and **behaviours**.
   - TaskReader**: Reads** and **validates** tasks from a file.
   - TaskComparator**:** Defines **task prioritization** logic.
2. **Focused Methods:**
   - Each method performs a **single**, **well-defined function**, such as Simulator.run() for the simulation loop and Schedular.scheduleTasks() for task scheduling.

❖ Low Coupling

Low coupling **reduces dependencies between classes**, making the system more **modular** and easier to maintain. This is achieved through:

1. **Encapsulation**:

   Each class hides its implementation details and **exposes only necessary methods**, like Task providing methods to access its properties.
2. **Minimal Dependencies:**

   Classes interact through **well-defined methods**. For instance, Simulator interacts with Schedular and Processor without knowing their **internals**.
3. **Loose Interactions:**
   - TaskComparator is used by PriorityQueue for task prioritization without creating tight coupling.
   - TaskReader **reads tasks** and returns them as a **collection**, allowing Simulator to **use** them without being **tightly coupled** to **file reading logic**.

❖ Reason for Coupling in the Simulation Class

- **Single Responsibility:** The Simulator class handles processor management, task handling, and scheduling directly through dependencies like Scheduler and Task Reader, ensuring **clear control** over simulation operations. In our **virtual simulation context**, after completion, resources like Scheduler and Task Reader **should be discarded** which causes the **composition** relationship.

Exception Handling

Exception handling within the Task Scheduling Simulator is crucial for ensuring smooth operation and reliability in the face of **potential errors**. This section outlines the key areas where exceptions are managed to maintain the **integrity and functionality** of the simulator, especially that some parameters are **explicitly entered by the user**.

- **Main Class (Main.java)**
  - Argument Parsing: Ensure correct usage and validate input arguments. Handle incorrect or insufficient arguments gracefully to **guide users towards correct usage**.
- **Simulator Class (Simulator.java)**
  - Task Validation: Validate task parameters during task reading to ensure they meet expected criteria (e.g., positive creation time, valid priority). **Handle parsing errors and parameter validation exceptions**.
  - Thread Interruptions in Sleep Operations: Catch InterruptedException during **Thread.sleep** to handle interruptions gracefully.
- **Simulator Class (Simulator.java)**
  - TaskReader: Manage file operations such as opening and reading tasks from the specified file. Handle scenarios where the **file might not be found or accessible**.

Debatable Cases Analysis

❖ Single vs Double Collections

In the task scheduling simulator, using a **Queue** for idle processors and an **ArrayList** for busy processors instead of a **single combined list** offers several benefits:

- **Clear Separation**: Maintains **distinct states** between idle and busy processors.
- **Efficiency**: **Queue** for idle processors allows for efficient task assignment using **FIFO**, while **ArrayList** for busy processors enables **direct access** and **quick update**s.
- **Flexibility:** Facilitates flexible state management and improves code maintainability by focusing operations on **relevant processor subsets**. Additionally, combining both into a single list would allow **iterating** over all processors at once for certain operations, providing flexibility where needed.
- **Performance:** Reduces **overhead** by optimizing **task assignment** and **status updates**, potentially enhancing simulation efficiency.

This approach ensures effective task scheduling and streamlines processor management in the simulator, with the added capability to **combine** and **iterate** over processors as **necessary** for **specific operations**.

❖ Using Observer Design Pattern

Our design simulates a task scheduler, but in real-life scenarios, processors work with a real-time clock. For a more **realistic design**, we could use the Observer pattern instead of utility to **synchronize** the **Clock** (if the case applies) with all processors. This would allow the Clock to tick in real-time, **notifying all processors** to **update** their **state simultaneously**.

Future Extensions

Modular Design for Extensibility

The project's **modular** architecture supports future enhancements and new feature integrations. By maintaining **independent** components, the simulator facilitates the addition of capabilities such as real-time task **interruption handling**.

Example: Interrupted Tasks Handling

Introducing a feature for **handling interrupted tasks** could involve enhancing **Task** and **Schedular classes** to support **interruption flags** and **adaptive** scheduling strategies, which means other classes won't be **affected** because the design is **modular** and only **intended classes** should be **affected**.

Summary of Changes needed

- **Task Class Changes:**
  - We need to add some methods like **pauseTask()** and **resumeTask()** to track tasks execution and control its state.
  - State management updates to handle interruption states for example (isInterrupted, isPaused).
- **Scheduler Class Changes:**
  - Modification of **task scheduling logic** to consider **interrupted** task states.
  - Adjustment of **data structures** (e.g., queues, lists) to **accommodate paused and resumed tasks**.

Scenario: Task Prioritization

Let’s consider an example of simulating **six tasks** being executed on a machine with **two processors** for **twelve clock cycles**, the following figures illustrates the main method arguments (number of processors, total simulation time, tasks file path) and the content of tasks.txt (input) which contains the tasks information. The **first line** represents the number of tasks to be executed, **the columns** represent **Task Creation Time, Execution Time, Priority respectively.**

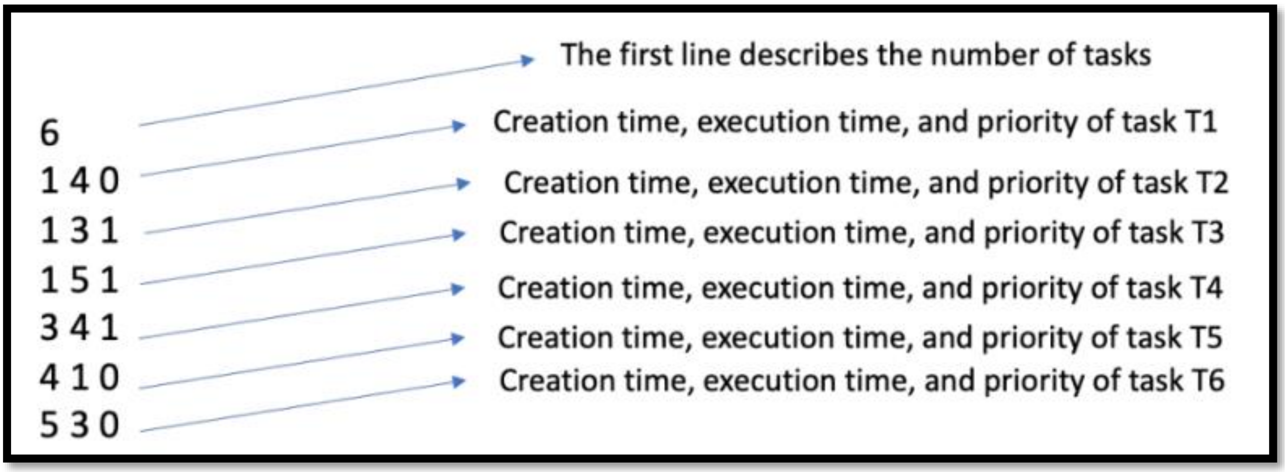

Figure 2. Tasks.txt File content

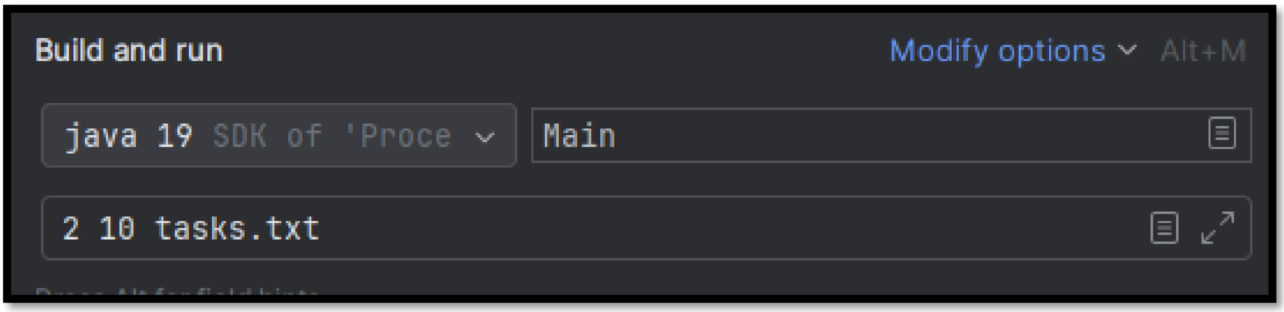

Figure 3. Main Arguments

Now Let’s compare the expected output with our code output, the figures below show that we get the same output but in different format.

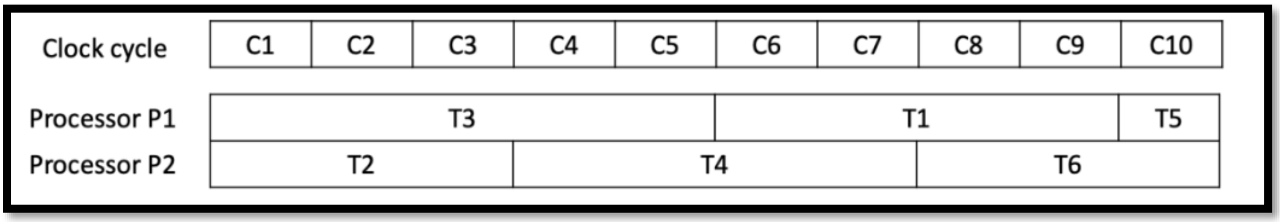

Figure 4. Expected Output

```
Scheduling 6 Tasks
Simulation started...
                                          Cycle 1
---------------------------------------------------------------------------------------------------------------------
Task T1 {creationTime=1, executionTime=4, priority=0, remainingTime=4} created and added to the queue
Task T2 {creationTime=1, executionTime=3, priority=1, remainingTime=3} created and added to the queue
Task T3 {creationTime=1, executionTime=5, priority=1, remainingTime=5} created and added to the queue
Task T3 assigned to P1
Task T2 assigned to P2
P1 {currentTask=T3}
P2 {currentTask=T2}
---------------------------------------------------------------------------------------------------------------------
                                          Cycle 2
---------------------------------------------------------------------------------------------------------------------
P1 {currentTask=T3}
P2 {currentTask=T2}
---------------------------------------------------------------------------------------------------------------------
                                          Cycle 3
---------------------------------------------------------------------------------------------------------------------
Task T4 {creationTime=3, executionTime=4, priority=1, remainingTime=4} created and added to the queue
P1 {currentTask=T3}
P2 {currentTask=T2}
T2 completed on Processor P2
---------------------------------------------------------------------------------------------------------------------
                                          Cycle 4
---------------------------------------------------------------------------------------------------------------------
Task T5 {creationTime=4, executionTime=1, priority=0, remainingTime=1} created and added to the queue
Task T4 assigned to P2
P1 {currentTask=T3}
P2 {currentTask=T4}
---------------------------------------------------------------------------------------------------------------------
                                          Cycle 5
---------------------------------------------------------------------------------------------------------------------
Task T6 {creationTime=5, executionTime=3, priority=0, remainingTime=3} created and added to the queue
P1 {currentTask=T3}
P2 {currentTask=T4}
T3 completed on Processor P1
---------------------------------------------------------------------------------------------------------------------
```

Figure 5. Simulation First Five Cycles

```
                                          Cycle 6
---------------------------------------------------------------------------------------------------------------------
Task T1 assigned to P1
P2 {currentTask=T4}
P1 {currentTask=T1}
---------------------------------------------------------------------------------------------------------------------
                                          Cycle 7
---------------------------------------------------------------------------------------------------------------------
P2 {currentTask=T4}
P1 {currentTask=T1}
T4 completed on Processor P2
---------------------------------------------------------------------------------------------------------------------
                                          Cycle 8
---------------------------------------------------------------------------------------------------------------------
Task T6 assigned to P2
P1 {currentTask=T1}
P2 {currentTask=T6}
---------------------------------------------------------------------------------------------------------------------
                                          Cycle 9
---------------------------------------------------------------------------------------------------------------------
P1 {currentTask=T1}
P2 {currentTask=T6}
T1 completed on Processor P1
---------------------------------------------------------------------------------------------------------------------
                                          Cycle 10
---------------------------------------------------------------------------------------------------------------------
Task T5 assigned to P1
P2 {currentTask=T6}
P1 {currentTask=T5}
T6 completed on Processor P2
T5 completed on Processor P1
---------------------------------------------------------------------------------------------------------------------
Simulation ended.
```

Figure 6. Simulation Second Five Cycles

Conclusion

The task scheduling simulator employs a structured approach to optimize task allocation, workload management, and system performance evaluation. Through strategic use of data structures and a modular design, the simulator facilitates comprehensive analysis of task scheduling strategies in multi-processor environments.