2024

PROCESSOR EXECUTION SIMULATOR

PRESENTED TO:
DR. FAHED JUBAIR

PREPARED BY:
AYMAN AL-ATTILI

Introduction

The Processor Execution Simulator is designed to model the behavior of a multi-processor system handling various tasks based on their priority and execution time. The simulation manages task scheduling, processor assignments, and logging events, providing insights into the efficiency of task execution over multiple clock cycles.

Classes Overview

1- Clock

The `Clock` class keeps track of the current cycle in the simulation. It has methods to increment the cycle and retrieve the current cycle value.

2- Event

The `Event` class represents an event in the simulation. Each event has a type, cycle when it occurs, and additional details.

3- Logger

The `Logger` class maintains a log of events. It uses a queue to store events and provides a method to log new events and retrieve the logs.

4- PriorityComparator

The `PriorityComparator` class implements the `Comparator` interface to compare tasks based on their priority and execution time. Higher priority tasks are given precedence, and among tasks with the same priority, those with shorter execution times are prioritized.

5- Processor

The `Processor` class represents a processor in the simulation. Each processor can be assigned a task, and it maintains its state (idle or busy).

6- Scheduler

The `Scheduler` class is responsible for scheduling tasks to processors. It uses a priority queue to manage tasks and assigns tasks to idle processors.

7- Simulator

The `Simulator` class is the core of the simulation. It initializes processors, tasks, and other components, and controls the simulation loop.

8- Task

The `Task` class represents a task to be executed. Each task has a creation time, execution time, priority, and an identifier.

9- TaskQueue

The `TaskQueue` class manages tasks using a priority queue, which sorts tasks based on their priority and execution time.

10- Main.java

The `Main` class initializes the `Simulator` with the specified number of processors, total clock cycles, and task file path, then starts the simulation.

Simulation Logic Explanation

The Processor Execution Simulator simulates the execution of tasks by multiple processors over a series of clock cycles. The simulation involves initializing tasks, scheduling them based on priority, executing them on available processors, and logging events. Here is a step-by-step explanation of the logic:

1. **Initialization**:
    - **Creating Components**:
        - The `Simulator` class initializes the necessary components: `numProcessors` (number of processors), `totalClockCycles` (total number of cycles for the simulation), `taskFilePath` (file path for tasks), and several internal components including `processors`, `taskQueue`, `scheduler`, `clock`, and `logger`.
        - Processors are initialized and added to the `processors` list.
        - `TaskQueue` and a temporary `taskQueueTemp` are created to manage tasks.
        - `Scheduler` is initialized with the `TaskQueue`.
        - `Clock` is initialized to keep track of simulation cycles.
        - `Logger` is initialized to log events.
        -

2. **Starting the Simulation**:
    - The `startSimulation` method is called to begin the simulation.
3. **Task Initialization**:
    - The `initializeTasks` method reads tasks from the specified file and adds them to the `taskQueueTemp`.
        - Tasks are read from the file and parsed into `creationTime`, `executionTime`, and `priority`.
        - Each task is created as an instance of the `Task` class and added to the `taskQueueTemp`.
4. **Simulation Loop**:
    - The simulation runs for the specified number of clock cycles (`totalClockCycles`).
    - For each cycle:
        - **Task Queue Update**:
            - Tasks from `taskQueueTemp` are moved to `taskQueue` if their creation time is less than or equal to the current cycle. This ensures that tasks are only considered for scheduling when they are due to be created.
        - **Simulation Update**:
            - The `updateSimulation` method is called with the current cycle to manage task scheduling and execution.
5. **Updating Simulation**:
    - **Task Scheduling**:
        - The `scheduleTasks` method in the `Scheduler` class assigns tasks to idle processors based on priority.
            - Idle processors are identified.
            - Tasks are dequeued from the `taskQueue` and assigned to idle processors.
            - Events related to task assignment are logged using the `Logger`.
    - **Task Execution**:
        - Processors execute their assigned tasks.
        - The `decrementExecutionTime` method of the `Task` class reduces the remaining execution time of the task by 1.
        - Events related to task execution are logged.
        - If a task's remaining execution time reaches 0, it is completed:
            - The processor releases the task.
            - Events related to task completion are logged.

    - **Clock Update**:
        - The `Clock` increments the cycle count using the `tick` method.
6. **Logging**:
    - The `Logger` logs all events related to task assignment, execution, and completion.
    - After each cycle, the logs are printed to the console.

UML Diagram

Relationships Between Classes

- `Simulator` uses `Scheduler`, `Clock`, `Logger`, `Processor`, and `TaskQueue` to manage the simulation.
- `Scheduler` interacts with `Processor` and `TaskQueue` to schedule tasks.
- `Logger` logs events related to `Task` assignments and executions.
- `Clock` provides the current cycle to `Simulator` and `Scheduler`.
- `PriorityComparator` is used by `TaskQueue` to prioritize tasks.

Data Structures and Their Usage

- **Queue (LinkedList)**: Used in `Logger` to maintain the event log. Chosen for its simplicity in adding and removing events in FIFO order.
- **PriorityQueue**: Used in `TaskQueue` to manage tasks based on priority and execution time. The `PriorityComparator` ensures tasks are sorted correctly.
- **ArrayList**: Used in `Simulator` to maintain the list of processors for easy iteration and random access.

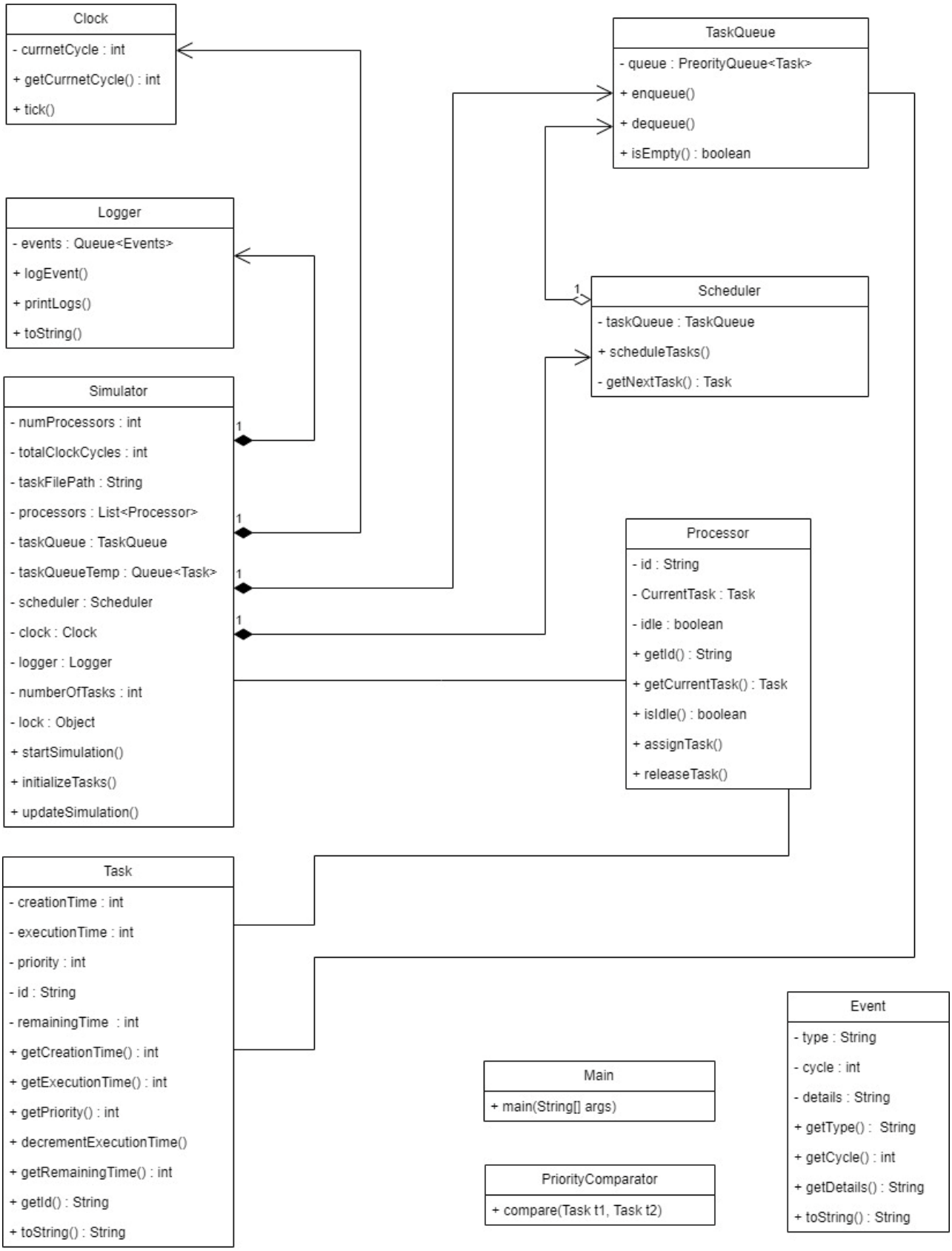
Coupling and Cohesion

Coupling

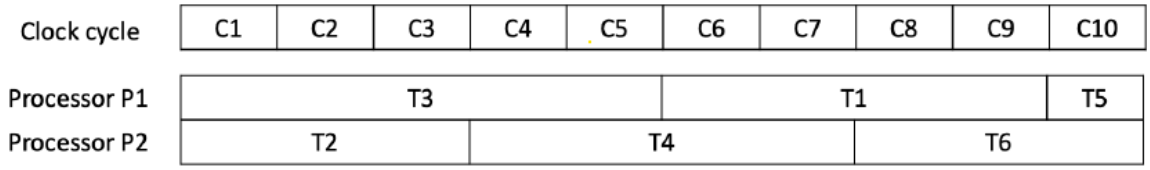
- **Low Coupling**: The system achieves low coupling by ensuring each class has a clear, distinct responsibility. For instance, the `Clock` class only manages the simulation cycle, and the `Logger` only handles event logging.
- **Interaction through Interfaces**: Classes interact through well-defined methods, reducing dependencies. For example, `Scheduler` interacts with `TaskQueue` through enqueue and dequeue methods without knowing the internal implementation.

Cohesion

- **High Cohesion**: Each class in the system has a single, well-defined purpose. For example, `Task` only manages task attributes and behavior, and `Processor` manages task assignments and processor state.
- **Focused Responsibilities**: Methods within each class are closely related to the class's main function, enhancing clarity and maintainability. For example, all methods in `Logger` are related to logging events.


Clock
- currnetCycle : int
+ getCurrnetCycle() : int
+ tick()
TaskQueue
- queue : PreorityQueue<Task>
+ enqueue()
+ dequeue()
+ isEmpty() : boolean
Logger
- events : Queue<Events>
+ logEvent()
+ printLogs()
+ toString()
Scheduler
- taskQueue : TaskQueue
+ scheduleTasks()
- getNextTask() : Task
Simulator
- numProcessors : int
- totalClockCycles : int
- taskFilePath : String
- processors : List<Processor>
- taskQueue : TaskQueue
- taskQueueTemp : Queue<Task>
- scheduler : Scheduler
- clock : Clock
- logger : Logger
- numberOfTasks : int
- lock : Object
+ startSimulation()
+ initializeTasks()
+ updateSimulation()
Processor
- id : String
- CurrentTask : Task
- idle : boolean
+ getId() : String
+ getCurrentTask() : Task
+ isIdle() : boolean
+ assignTask()
+ releaseTask()
Task
- creationTime : int
- executionTime : int
- priority : int
- id : String
- remainingTime : int
+ getCreationTime() : int
+ getExecutionTime() : int
+ getPriority() : int
+ decrementExecutionTime()
+ getRemainingTime() : int
+ getId() : String
+ toString() : String
Main
+ main(String[] args)
PriorityComparator
+ compare(Task t1, Task t2)
Event
- type : String
- cycle : int
- details : String
+ getType() : String
+ getCycle() : int
+ getDetails() : String
+ toString() : String
1
1
1
1
1

Test case #1 :



My simulation report for case#1:

Event{type='TaskExecution', cycle=1, details='Executing task T3 on P1'}

Event{type='TaskExecution', cycle=1, details='Executing task T2 on P2'}

Event{type='TaskExecution', cycle=2, details='Executing task T3 on P1'}

Event{type='TaskExecution', cycle=2, details='Executing task T2 on P2'}

Event{type='TaskExecution', cycle=3, details='Executing task T3 on P1'}

Event{type='TaskExecution', cycle=3, details='Executing task T2 on P2'}

Event{type='TaskCompleted', cycle=3, details='Task T2 completed on P2'}

Event{type='TaskExecution', cycle=4, details='Executing task T3 on P1'}

Event{type='TaskExecution', cycle=4, details='Executing task T4 on P2'}

Event{type='TaskExecution', cycle=5, details='Executing task T3 on P1'}

Event{type='TaskCompleted', cycle=5, details='Task T3 completed on P1'}

Event{type='TaskExecution', cycle=5, details='Executing task T4 on P2'}

Event{type='TaskExecution', cycle=6, details='Executing task T1 on P1'}

Event{type='TaskExecution', cycle=6, details='Executing task T4 on P2'}

Event{type='TaskExecution', cycle=7, details='Executing task T1 on P1'}

Event{type='TaskExecution', cycle=7, details='Executing task T4 on P2'}

Event{type='TaskCompleted', cycle=7, details='Task T4 completed on P2'}

Event{type='TaskExecution', cycle=8, details='Executing task T1 on P1'}

Event{type='TaskExecution', cycle=8, details='Executing task T6 on P2'}

Event{type='TaskExecution', cycle=9, details='Executing task T1 on P1'}

Event{type='TaskCompleted', cycle=9, details='Task T1 completed on P1'}

Event{type='TaskExecution', cycle=9, details='Executing task T6 on P2'}

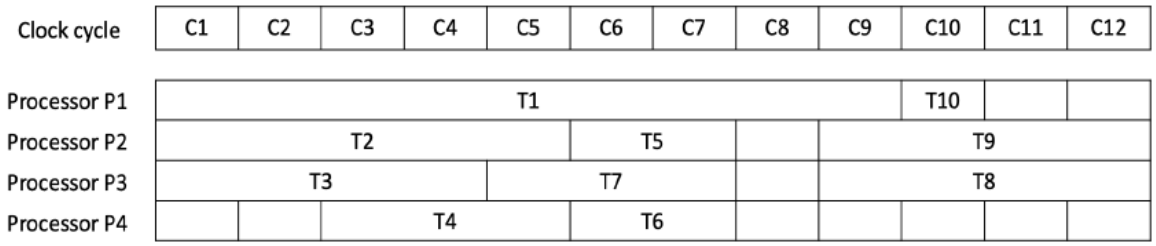
Event{type='TaskExecution', cycle=10, details='Executing task T5 on P1'}

Event{type='TaskCompleted', cycle=10, details='Task T5 completed on P1'}

Event{type='TaskExecution', cycle=10, details='Executing task T6 on P2'}

Event{type='TaskCompleted', cycle=10, details='Task T6 completed on P2'}

Test case #2:



My simulation report for case #2:

Event{type='TaskExecution', cycle=1, details='Executing task T1 on P1'}

Event{type='TaskExecution', cycle=1, details='Executing task T2 on P2'}

Event{type='TaskExecution', cycle=1, details='Executing task T3 on P3'}

Event{type='TaskExecution', cycle=2, details='Executing task T1 on P1'}

Event{type='TaskExecution', cycle=2, details='Executing task T2 on P2'}

Event{type='TaskExecution', cycle=2, details='Executing task T3 on P3'}

Event{type='TaskExecution', cycle=3, details='Executing task T1 on P1'}

Event{type='TaskExecution', cycle=3, details='Executing task T2 on P2'}

Event{type='TaskExecution', cycle=3, details='Executing task T3 on P3'}

Event{type='TaskExecution', cycle=3, details='Executing task T4 on P4'}

Event{type='TaskExecution', cycle=4, details='Executing task T1 on P1'}

Event{type='TaskExecution', cycle=4, details='Executing task T2 on P2'}

Event{type='TaskExecution', cycle=4, details='Executing task T3 on P3'}

Event{type='TaskCompleted', cycle=4, details='Task T3 completed on P3'}

Event{type='TaskExecution', cycle=4, details='Executing task T4 on P4'}

Event{type='TaskExecution', cycle=5, details='Executing task T1 on P1'}

Event{type='TaskExecution', cycle=5, details='Executing task T2 on P2'}

Event{type='TaskCompleted', cycle=5, details='Task T2 completed on P2'}

Event{type='TaskExecution', cycle=5, details='Executing task T7 on P3'}

Event{type='TaskExecution', cycle=5, details='Executing task T4 on P4'}

Event{type='TaskCompleted', cycle=5, details='Task T4 completed on P4'}

Event{type='TaskExecution', cycle=6, details='Executing task T1 on P1'}

Event{type='TaskExecution', cycle=6, details='Executing task T5 on P2'}

Event{type='TaskExecution', cycle=6, details='Executing task T7 on P3'}

Event{type='TaskExecution', cycle=6, details='Executing task T6 on P4'}

Event{type='TaskExecution', cycle=7, details='Executing task T1 on P1'}

Event{type='TaskExecution', cycle=7, details='Executing task T5 on P2'}

Event{type='TaskCompleted', cycle=7, details='Task T5 completed on P2'}

Event{type='TaskExecution', cycle=7, details='Executing task T7 on P3'}

Event{type='TaskCompleted', cycle=7, details='Task T7 completed on P3'}

Event{type='TaskExecution', cycle=7, details='Executing task T6 on P4'}

Event{type='TaskCompleted', cycle=7, details='Task T6 completed on P4'}

Event{type='TaskExecution', cycle=8, details='Executing task T1 on P1'}

Event{type='TaskExecution', cycle=9, details='Executing task T1 on P1'}

Event{type='TaskCompleted', cycle=9, details='Task T1 completed on P1'}

Event{type='TaskExecution', cycle=9, details='Executing task T9 on P2'}

Event{type='TaskExecution', cycle=9, details='Executing task T8 on P3'}

Event{type='TaskExecution', cycle=10, details='Executing task T10 on P1'}

Event{type='TaskCompleted', cycle=10, details='Task T10 completed on P1'}

Event{type='TaskExecution', cycle=10, details='Executing task T9 on P2'}

Event{type='TaskExecution', cycle=10, details='Executing task T8 on P3'}

Event{type='TaskExecution', cycle=11, details='Executing task T9 on P2'}

Event{type='TaskExecution', cycle=11, details='Executing task T8 on P3'}

Event{type='TaskExecution', cycle=12, details='Executing task T9 on P2'}

Event{type='TaskCompleted', cycle=12, details='Task T9 completed on P2'}

Event{type='TaskExecution', cycle=12, details='Executing task T8 on P3'}

Event{type='TaskCompleted', cycle=12, details='Task T8 completed on P3'}