Processor Execution Simulator

Done By : Hamzeh Hatamleh

Instructor : Mostasem Diab

Introduction

IF we took some time searching about the processors , we could notice that it carries out the instructions of a program by performing basic arithmetic, logic, control, and input/output operations

The processor determines how fast and efficiently a computer can run programs, handle calculations, and respond to user inputs. Faster, more powerful processors can handle more complex tasks, like gaming, AI computations, or large data processing.

In this assignment , I worked on building a simulator to execute tasks based on a constant number of processors.

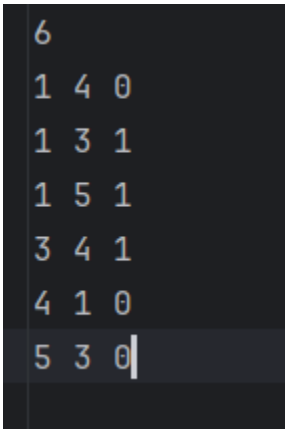
This is a great assignment , where it tested my ability in multithreading and general oop.

TestFile

If we take a look on our test file , we can see that it's all about 3 parts:

Creation Time: The cycle in which the task started.
Execution Time: The number of cycles required for the task.
Priority: High (1) or Low (0).

So we should handle the text file based on the current cycle , so if we are in cycle 1 , then we should add the tasks that start in cycle 1 to the queue, These tasks should be sorted in a way that allows us to dequeue and execute the highest priority task first.



Here , we have 6 tasks , each with the 3 parts mentioned above.

We will talk in the **Simulator** class about how to handle this text file using java

Processor & Task

So , Our main process includes assigning tasks to the available processors, so in the first step I created 2 classes to represent both , **Task** and **Processor.**

Task Class

This class will only be responsible for creating the task , based on the **taskNumber**,**creationTime**,**ExecutionTime**,and **priority.**

```
int taskNumber;  2 usages
int creationTime;  1 usage
int executionTime;  2 usages
int priority;  2 usages

Task(int creationTime, int executionTime, int priority, int TaskNumber) {  1 usage
    this.creationTime = creationTime;
    this.executionTime = executionTime;
    this.priority = priority;
    this.taskNumber = TaskNumber;
}
```

Processor Class

The processor is the core component of our code, so it's crucial to understand its availability and how to represent it. In my opinion, if there are 10 clock cycles, each processor can be represented as an array of 10 addresses. If the processor is busy throughout all cycles, then every position in the array should be set to false. Check variable ***boolean[] available.***

```
public class Processor{  4 usages
    int numberClockCycles;  2 usages
    boolean[] available;  5 usages
    int processorId;  2 usages
```

So based on the number of clock cycles , we will have our processors , and it’s all set to true in the first because all the processes are available before beginning.

The main function here is **executeTask**, which is responsible for executing a given task t at a specific clockCycle. This function checks whether the processor is available at the specified clock cycle. If the processor is available, it proceeds to execute the task and marks the corresponding clock cycles as busy for the duration of the task's execution time. If the processor is not available, it notifies that the task cannot be executed at that cycle due to the processor being busy.

```
public void executeTask(Task t, int clockCycle) {  1 usage
    if (available[clockCycle]) {
        System.out.println("  ✅ Executing Task " + t.getTaskNumber() + " (Exec Time: " + t.getExecutionTime() + ") on cycle: " + (clockCycle + 1) + " [Process
        for (int i = clockCycle; i < clockCycle + t.getExecutionTime(); i++) {
            if (i < numberClockCycles) {
                available[i] = false;
            } else {
                break;
            }
        }
    } else {
        System.out.println("  ❌ Task " + t.getTaskNumber() + " can't execute on cycle: " + (clockCycle + 1) + " (Processor busy)");
    }
}
```

Simulator

This class is the main one in my code , it’s responsible for simulating the whole process.

```
int numberProcessors;  2 usages
int numberClockCycles;  2 usages
String path; // path to text file that contains tasks infos  2 usages
Queue<Task> taskQueue; // Queue to add the tasks to   9 usages
List<Processor> processors; // our processors  5 usages
private static int taskCounter=1;   2 usages


Simulator(int numberProcessors , int numberClockCycles , String path) throws FileNotFoundException {  1 usage
    this.numberProcessors = numberProcessors;
    this.numberClockCycles = numberClockCycles;
    this.path = path;
    this.taskQueue = new LinkedList<>();
    this.processors = new ArrayList<>();

    for (int i = 0; i < numberProcessors; i++) {
        processors.add(new Processor(numberClockCycles, processorId: i+1)); // Initialize processors
    }
}
```

This class got the number of processors, number of clockCycles , and the path of our text file to begin working with.

simulate() function simulate the whole process , so this is how simulator works:

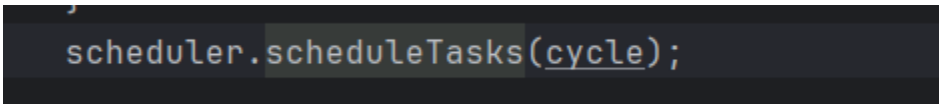
1) First step , we have to deal with the text file , The `HandleTextFile` method reads a text file containing task data and processes tasks based on the current clock cycle. It extracts the creation time, execution time, and priority of each task from the file, and if the task's creation time matches the next clock cycle, it adds the task to the queue for execution. The method ensures that only tasks scheduled for the current cycle are processed, and it assigns a unique identifier to each task before adding it to the queue for further processing.

```java
public void HandleTextFile(int clockCycle) throws IOException {  1 usage
    BufferedReader bf = new BufferedReader(new FileReader(path));
    String line;
    while ((line = bf.readLine()) != null) {



        // Assuming the file has values separated by spaces or commas
        String[] parts = line.split( regex: " ");

        if (parts.length >= 3) { // Ensure at least 3 elements exist
            int creationTime = Integer.parseInt(parts[0]);

            if(creationTime == clockCycle+1){

                int executionTime = Integer.parseInt(parts[1]);
                int priority = Integer.parseInt(parts[2]);
                Task t = new Task(creationTime, executionTime, priority, taskCounter);
                addTaskToTheQueue(t);
                taskCounter++;
            }
        }
    }
}
```

2) After that , we should sort the tasks in the queue in a way we can deque it easily , so the sorting is based on the following:
    - **Sort by priority**: Tasks with a higher priority (value `1`) come before those with lower priority (value `0`).
    - **Sort by execution time**: Among tasks with the same priority, tasks with a longer execution time are placed first, as the sorting is done in descending order (indicated by `.reversed()`).

```java
// this function will sort the queue to dequeue easily
public void sortTasksInQueue() {  1 usage
    List<Task> taskList = new ArrayList<>(taskQueue); // Convert queue to list

    taskList.sort(Comparator
            .comparingInt(Task::getPriority) // 1. Sort by priority (1 before 0)
            .thenComparingInt(Task::getExecutionTime) // 2. Sort by execution time (ascending)
            .reversed() // 3. Reverse the order of execution time (descending)
    );
}
```

3) Now that the queue is sorted, the scheduler can begin dequeuing tasks. Based on the current cycle, the scheduler should schedule tasks to ensure they are ready for execution on the available processor.



Scheduler

The schedular here must pritorize the assignment of the tasks .

One approach is to loop through the processors, identifying the available ones to dequeue tasks. However, this method is inefficient because it requires checking each processor in every cycle.

I came up with an idea: what if we make each processor a thread? This way, each processor operates independently, allowing any available processor to take on tasks without the need for constant looping.

So , In the **first step** we have to create the threads , let's confirm that number of thread = number of processors, then we should start each thread to begin working independently.I

```
// Creates threads for each processor and starts them
private Thread[] createThreads(int clockCycle) {  1 usage
    Thread[] threads = new Thread[sm.processors.size()];

    for (int i = 0; i < sm.processors.size(); i++) {
        final Processor processor = sm.processors.get(i);
        threads[i] = new Thread(() -> executeTaskForProcessor(processor, clockCycle));
        threads[i].start();
    }
    return threads;
}
```

In the **executeTaskForProcessor** function, we will check if the processor is available so we can dequeue the task from the queue.

```java
// Executes the task for a given processor at a specified clock cycle
private void executeTaskForProcessor(Processor processor, int clockCycle) {  1 usage
    if (processor.isAvailable(clockCycle)) {
        Task task = deque(); // Get a task from the queue
        if (task != null) {
            processor.executeTask(task, clockCycle); // Execute the task
        }
    }
}
```

IF the processor is available and the queue has tasks , so the processor should execute the task on the current clock cycle

In the Second Step, we have to wait for all the threads to finish taking the tasks in the cycle (i) , so when they finish we can move to the next cycle.

```java
public void scheduleTasks(int clockCycle) {  1 usage
    // Create and start threads for all processors
    Thread[] threads = createThreads(clockCycle);

    // Wait for all threads to finish executing
    waitForThreads(threads);
}
```

Conclusion

In the end , it’s a great project to work on. I focused on applying multithreading and I did. Multithreading is a great concept to save memory and to optimize the program.

After each assignment , I find myself better. Thank you dr motasem for the great work that you’re giving to us , I truly appreciate the effort and dedication put into the training.