Assignment 2

Mesk Alhamaideh

Section 1

Process Simulation Execution

Introduction

Assignment2 code does preemptive priority scheduling. The code consists of seven classes (Processors, Tasks, Scheduler, Simulation, Queue, Reader, Writer) we'll go in each one in details.

What is priority Scheduling?

Priority scheduling involves priority assignment to every process, and processes with higher priorities are carried out first, whereas tasks with equal priorities are carried out on a first-come-first-served (FCFS) or round robin basis.

How I implemented it:

- **Queue**

To sort the tasks based on their priority I used a priority queue. Tasks are first read and created in Reader class, when tasks are created, they're automatically added to the priority queue that sorts them using compareTo in the Tasks class based on priority.

Tie breaking strategy:

The Scheduler changes the task inside a processor with another task that is in the queue if the task in the queue is either higher or equal in priority. Meaning higher priority tasks are more important than low priority ones, and equal priority tasks (whether high or low) keep switching together each cycle in a round robin like fashion.

- **Scheduling**

The scheduler does four things for the purpose of scheduling (contextSwitch function):

Firstly, it fills all the idle processors with tasks that are waiting in the queue. Secondly, if there are still tasks remaining in the queue it switches them with the ones inside the processors if their priority is higher. Then we execute the tasks. Finally, the scheduler removes all the tasks that are done into the finishedTasks list.

Data structures used in Scheduler class:

1. A list for idle processors (in order to fill them in the fillIdleProcessors function).
2. A list for busy processors (in order to call execute their tasks).
3. A list for all processors (just for printing output such that processors are ordered by their ID rather than printing processors in a random order).
4. A list for finished tasks (also for output purposes).

Simulation

In the main function we start by asking the user for the number of processors, and then a while loop (with a counter that represents each clock cycle) starts. In each cycle we start by asking the user if there are any new tasks that arrived in that cycle and then we call contextSwitch function in order to distribute tasks and execute them.

Design Patterns

Writer, Reader, Queue classes all follow singleton design pattern, because we only need one reader and writer in order to read from input.txt file and write to output.txt file. We also must have one queue to put all the tasks inside in order to distribute them among processors.

Clean Code

- The code is DRY: "Every piece of knowledge must have a single, unambiguous, authoritative representation within a system".
- Meaningful variable and function names like fillIdleProcessors and removeFinishedTasks.
- Functions are short and don't need many arguments.
- No long comments

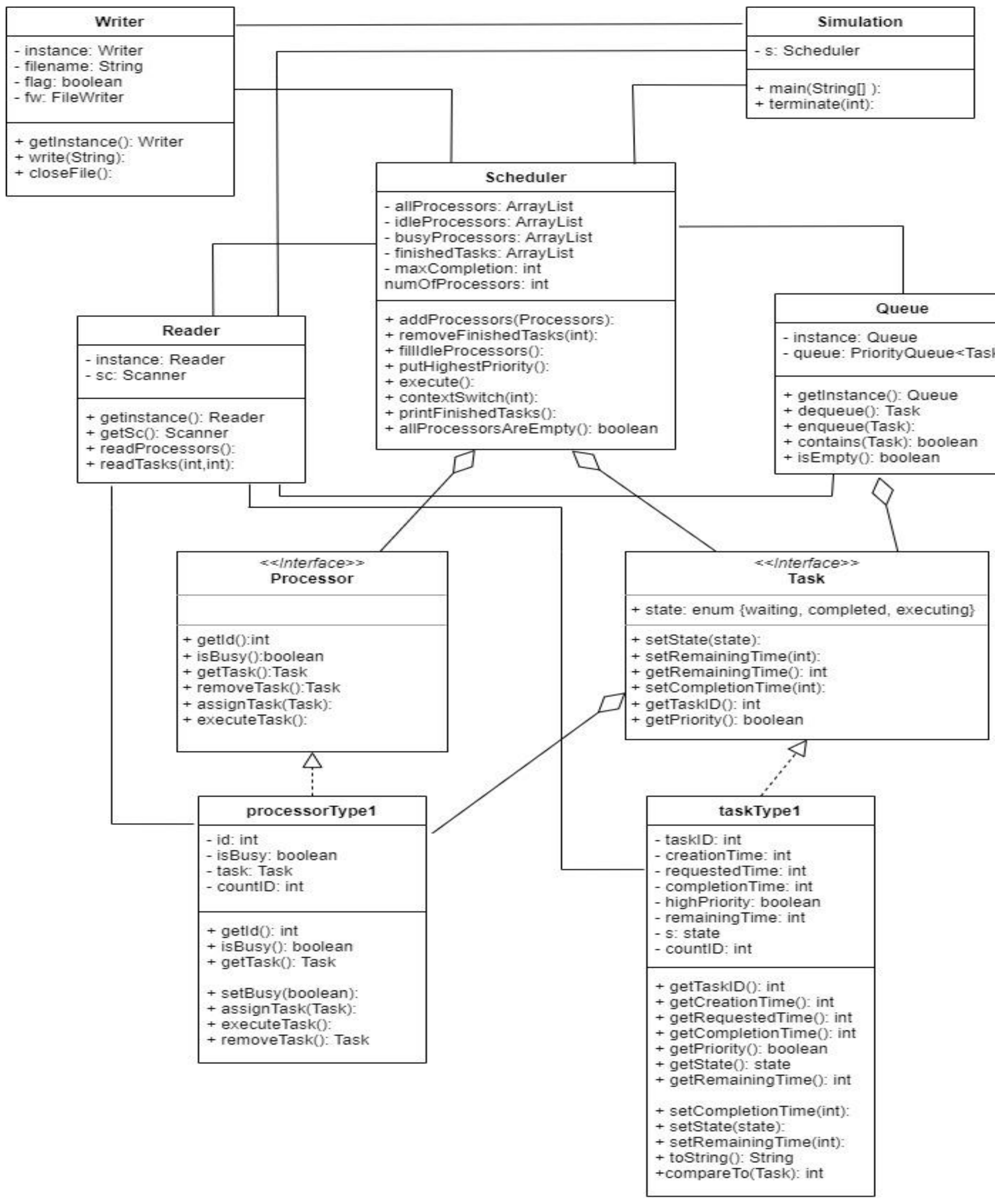
SOLID Principles

- Single responsibility: each module in the code has only one responsibility
- Dependency inversion: "High-level modules should not depend on low-level modules. Both should depend on abstractions"

Which is achieved by making task and processor interfaces

- Open Closed principle: the code is extendable if any changes occur.

UML


Writer
- instance: Writer
- filename: String
- flag: boolean
- fw: FileWriter
+ getInstance(): Writer
+ write(String):
+ closeFile():
Simulation
- s: Scheduler
+ main(String[]):
+ terminate(int):
Scheduler
- allProcessors: ArrayList
- idleProcessors: ArrayList
- busyProcessors: ArrayList
- finishedTasks: ArrayList
- maxCompletion: int
numOfProcessors: int
+ addProcessors(Processors):
+ removeFinishedTasks(int):
+ fillIdleProcessors():
+ putHighestPriority():
+ execute():
+ contextSwitch(int):
+ printFinishedTasks():
+ allProcessorsAreEmpty(): boolean
Reader
- instance: Reader
- sc: Scanner
+ getinstance(): Reader
+ getSc(): Scanner
+ readProcessors():
+ readTasks(int,int):
Queue
- instance: Queue
- queue: PriorityQueue<Task>
+ getInstance(): Queue
+ dequeue(): Task
+ enqueue(Task):
+ contains(Task): boolean
+ isEmpty(): boolean
<<Interface>>
Processor
+ getId():int
+ isBusy():boolean
+ getTask():Task
+ removeTask():Task
+ assignTask(Task):
+ executeTask():
<<Interface>>
Task
+ state: enum {waiting, completed, executing}
+ setState(state):
+ setRemainingTime(int):
+ getRemainingTime(): int
+ setCompletionTime(int):
+ getTaskID(): int
+ getPriority(): boolean
processorType1
- id: int
- isBusy: boolean
- task: Task
- countID: int
+ getId(): int
+ isBusy(): boolean
+ getTask(): Task
+ setBusy(boolean):
+ assignTask(Task):
+ executeTask():
+ removeTask(): Task
taskType1
- taskID: int
- creationTime: int
- requestedTime: int
- completionTime: int
- highPriority: boolean
- remainingTime: int
- s: state
- countID: int
+ getTaskID(): int
+ getCreationTime(): int
+ getRequestedTime(): int
+ getCompletionTime(): int
+ getPriority(): boolean
+ getState(): state
+ getRemainingTime(): int
+ setCompletionTime(int):
+ setState(state):
+ setRemainingTime(int):
+ toString(): String
+compareTo(Task): int