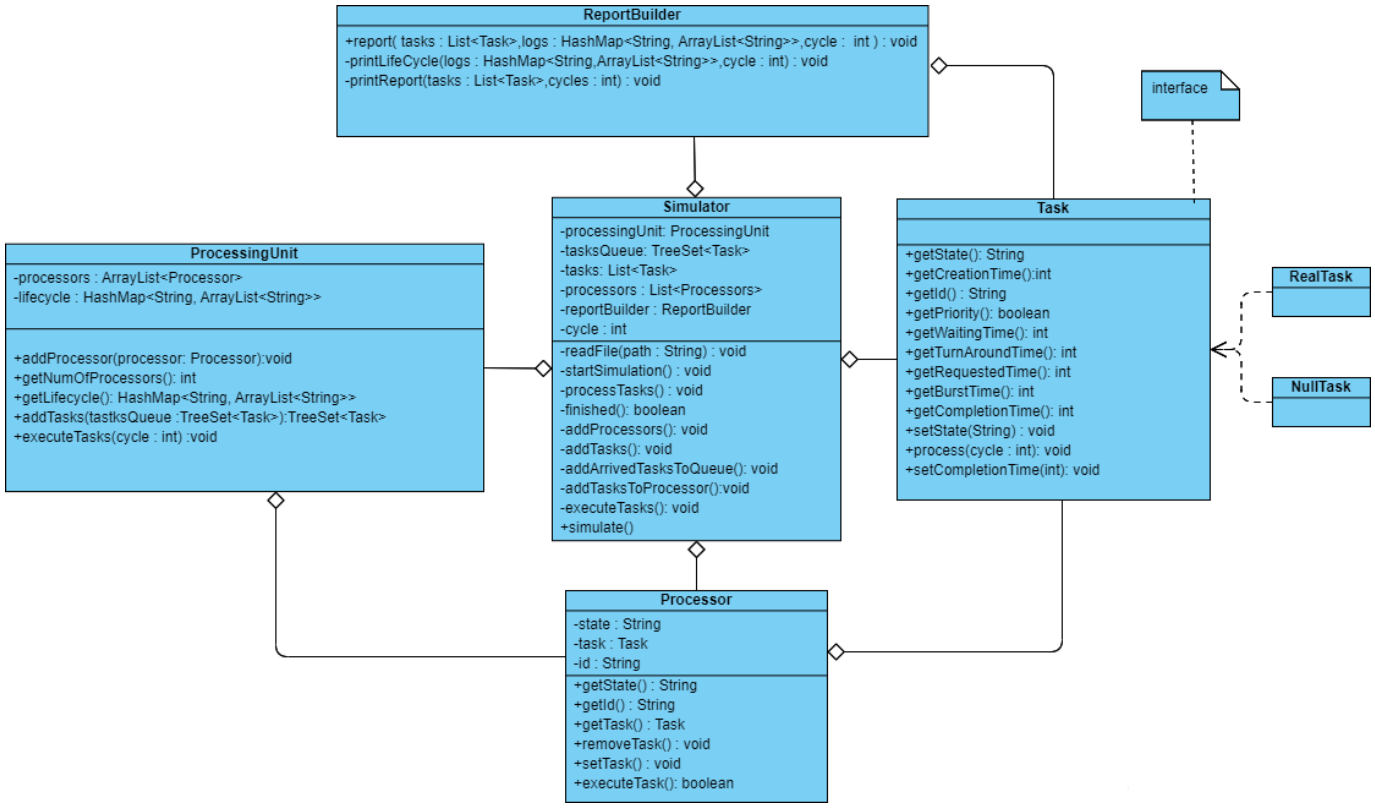
Processor Execution Simulator

In this report I will be showing the task Processor Execution Simulator project, describing and walkingthrough all the algorithms and data sturctures and ideas implemented in this project



Processor execution simulator Class Diagram

As shown in the diagram above, **Simulator** class works as the brain of the simulation, when the **Simulator** class reads the file, it send these tasks to the processors, then the processors executes their tasks in the **ProcessingUnit** and saves the processor operations in a logs data structure that will be sent to the **ReportBuilder**, taking in mind that class **simulator** keeps track or the cycle the last step will be the **ReportBuilder** making use of the logs saved by the processors and display it in the terminal.

	A	B	C	D
1	number of	Processors = 3		
2	Task	CreationTi	Requested	Priority
3	t1	0	10	low
4	t2	2	7	high
5	t3	2	7	high
6	t4	2	7	high
7	t5	4	3	high
8	t6	4	3	high
9	t7	4	3	high

.csv file sample

So, I have decided to read simulation data from a csv file, where the number of processors is given in the first line and the number of tasks, tasks Id, creation time, requested time and priority will be displayed below in a table

Processor Execution Simulator Classes:

Task: is an interface that a **RealTask** and a **NullTask** implements because sometimes the tasks are null, just to be safe from using the null objects.

RealTask: it is used to save all the data about the task which was read from the file, now other than the obvious variables and functions I have implemented some that I believe to be useful:

1- *burstTime*: *RequestedTime* is already defined but *burstTime* will be decremented each cycle and I want to keep track of the *requestedTime* to show in the report.
2- *turnAroundTime*: will be used to get the *waitingTime* and to be displayed in the report.
3- *waitingTime*: is a good stat to be shown in any scheduling simulator.
4- process(): the tasks *burstTime* will be decremented by 1, and it will return nothing.

NullTask: will have the same functions as class **RealTask,** but the functions will do nothing or return "null task ".

Processor: will be saving the id of the processor, state and the task that is attached to it, now other than the obvious methods I have implemented some that I believe to be useful:

1- setTask(): will attach a task to the processor, and will return nothing.
2- removeTask(): will remove a task from the processor, and will return nothing.
3- executeTask(): will use the process() method from the task attached to the processor to decrement the task burst time by 1, and if the task is finished (*burstTime* equals 0) will return true, else it will return false.

ProcessingUnit: this class will save all the processors read from the file in an array list of type **Processor**, and it also uses the observer design pattern to go through the list of processors and use the function executeTask() so they are all updated at the same cycle, and while they are updating the logs of the execution is all saved in data structure called lifecycle, so it can be used in the report. variables and methods used:

1- *ArrayList<Processor> processors*: will save all the processors read from the file.
2- *HashMap<String, ArrayList<String>> lifecycle*: this HashMap will save the processor as a key and the tasks that has been processed as an ArrayList which will be used late in the **ReportBuilder** class.
3- addProcessors (): responsible for adding a processor to the processors ArrayList.
4- addTasks (): will receive TreeSet *tasksQueue* that will have the arrived tasks depending on the cycle, and is ordered in a way that the task with high priority comes first and if there is a tie it will prioritize the task with lower *burstTime,* then it will loop through the processors and give them tasks if processors *state* equals "Idle" or interrupt a task if a task with higher priority arrived in the *tasksQueue* if the task was attached to a processor it will be remove from the *tasksQueue* (to make sure no other processor execute the same task) and if the tasks is interrupted the interrupted task will be added back to the queue, this function will return the *tasksQueue* to maintain the changes that happened to the queue in the **Simulator** class.
5- executeTasks (): this method will loop through all the processors and do:
   a- use the function executeTask () which will decrement the tasks *burstTime* by 1.
   b- Use the *processorID as a key to* add the *taskId* if the **Task** is a **RealTask** or just add the symbol "|" if the **Task** is **NullTask** as a processed task to the *lifecycle*, so it looks something like this.

P1: [T1, T1, T1, T4]
P2: [T2, T2, "|", "|"]

and so the report can be like this:

P1	P2	
T1	T2	
T1	T2	
T1	|	← Idle processor
T4	|	← Idle processor

   If the function executeTask () returns true (the task state is "Finished") it will put in the *lifecycle* the taskId + "★" to show in the report where the task has finished.
   It will look something like this:

P1	P2
T1	T2
T1	T2★
T1★	|
T4★	|

6- getNumOfProcessors (): will return the number of processors.
7- getLifecycle (): will return the lifecycle HashMap.

*Note: executeTasks() is a method in class **ProcessingUnit** & executeTask() is a function in class **Processor***

ReportBuilder: this class will be responsible from displaying the report and the simulation lifecycle in the terminal with these two functions:

1- printLifecycle (): this method will print the lifecycle; it will take the lifecycle HashMap as a parameter and print them in the terminal with a small delay between each cycle

C	p1	p2	p3
	⊤	⊤	⊤
1	t1	|	|
2	t1	|	|
3	t2	t3	t4
4	t2	t3	t4
5	t2	t3	t4
6	t2	t3	t4
7	t2	t3	t4
8	t2	t3	t4
9	t2★	t3★	t4★
10	t5	t6	t7
11	t5	t6	t7
12	t5★	t6★	t7★
13	t1	|	|
14	t1	|	|
15	t1	|	|
16	t1	|	|
17	t1	|	|
18	t1	|	|
19	t1	|	|
20	t1★	|	|

2- printReport (): this will take an array of tasks which was read from the file in their original order, then it will loop around them and display the *taskId, creationTime, requestedTime, completionTime, priority, turnAroundTime and waitingTime,* and will also calculate and show the cycles it took to finish the simulation, average waiting time and average turnaround time.

TaskId	CreationTime	RequestedTime	CompletionTime	Priority	TurnAroundTime	WaitingTime
t1	0	10	20	low	20	10
t2	2	7	9	high	7	0
t3	2	7	9	high	7	0
t4	2	7	9	high	7	0
t5	4	3	12	high	8	5
t6	4	3	12	high	8	5
t7	4	3	12	high	8	5

```
finished in 20 cycles
average turn around time time = 9.285714285714286
average waiting time = 3.5714285714285716
```

PriorityArrivalQueueComp: this is a class that implements comparator which will be used to prioritize tasks over the other in the tasksQueue TreeSet

1- prioritize high priority over low priority.
2- If priority is equal compare burstTime the lower is prioritized.

Simulator: the simulator class works like the brain of this simulation project, it all starts by reading the data from the file, then starting the simulation and executing methods each cycle, the methods and variables used to make this work.

1- *TreeSet<**Task**> tasksQueue:* is a TreeSet of type **Task** used to prioritize tasks over the other by implementing **PriorityArrivalQueueComp**
2- **ProcessingUnit** *processingUnit*: to use its methods
3- will be used to add processors to the **ProcessingUnit** by using method addProcessor (), adding tasks to the processors by using method addTasks (tasksQueue) passing the queue
4- *ArrayList<**Task**> tasks: will save the tasks read from the file and maintain the order the tasks where formed in.*
5- **ReportBuilder** *reportBuilder:* to build the report after the processing is done.
6- **int** *cycle:* to keep track of the cycle which will be initialized to 0;
7- readFile (): will read the file and:

    a- add processors to the **ProcessingUnit** using function addProcessors (new Processor ()) In a loop which will loop depending on how many processors the file informed.
    b- add all the tasks read from the file to the *ArrayList<**Task**>* tasks.
8- finished (): is a Boolean function which will return true if the tasks are all finished else it will return false.
9- startSimulation (): is a private function that will keep looping until finished () is true, inside this loop four procedures will be done:
    a- addArrivedTasksToQueue (): add all the tasks from tasks *ArrayList<**Task**>* that their arrival number matches the cycle to the tasksQueue.
    b- addTasksToProcessors (): Add these tasks to the processors in the **ProcessingUnit**.
    c- executeTasks (): Execute these tasks that are in the **ProcessingUnit.**
    d- Increment *cycle* by 1;

After the loop is finished send the *lifecycle* HashMap (from the **ProcessingUnit**), *tasks ArrayList<**Task**>* and *cycle* to the *reportBuilder* to display the results in the terminal.

Processor Execution Simulator Lifecycle:

1- make N processors in the **ProcessingUnit**
2- add all the tasks to an ArrayList maintaining the original order.
3- while not every task has finished:
    a- add the tasks that have arrived to the TreeSet that will prioritize which one should be executed first.
    b- attach tasks to the processors depending on the Queue, if the processor is idle attach the highest priority task directly to it and remove the task from the queue, else if the processors tasks priority is low and the queue contains a higher priority task add the higher priority task to the processor and the add the detached task to the queue.
    c- loop around the processors and execute each task.
    d- increment the cycle by 1.
4- use the report builder to display the results in the terminal.