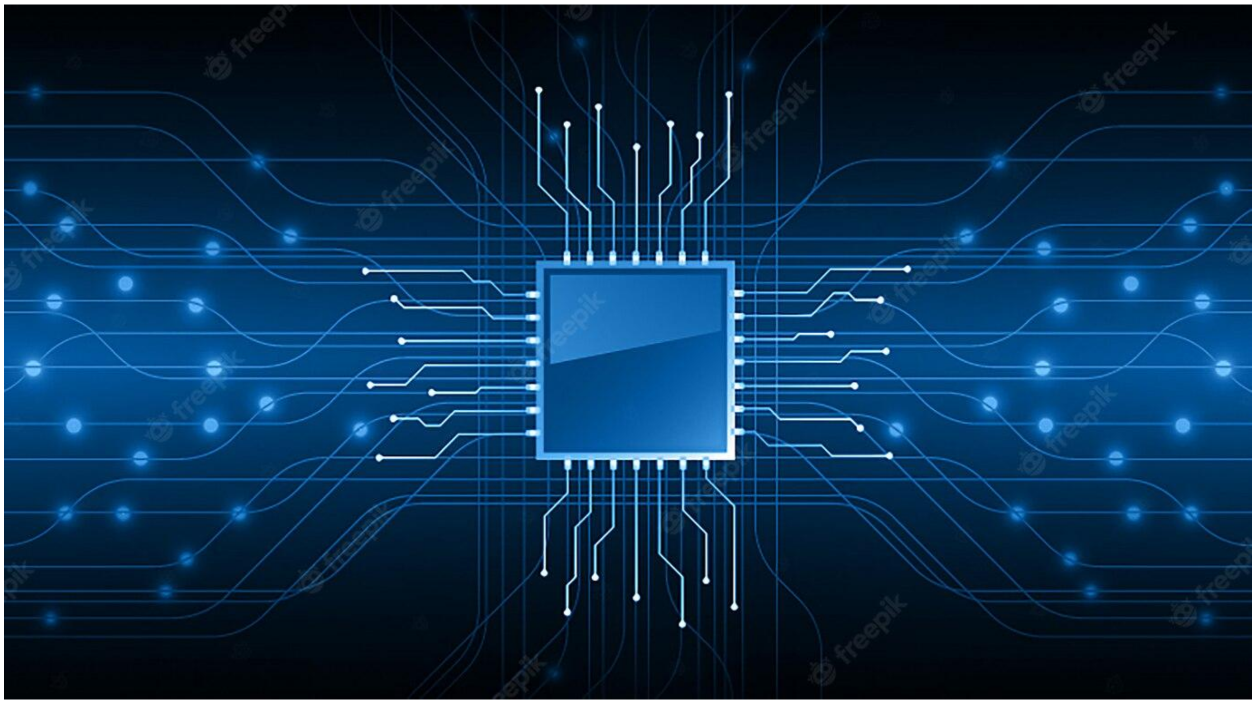
Atypon Training - Summer 2022

Processor Execution Simulator Report

Mohannad Atmeh



Abstract

This report is about a processor simulation to practice Java multithreading and SOLID principles.

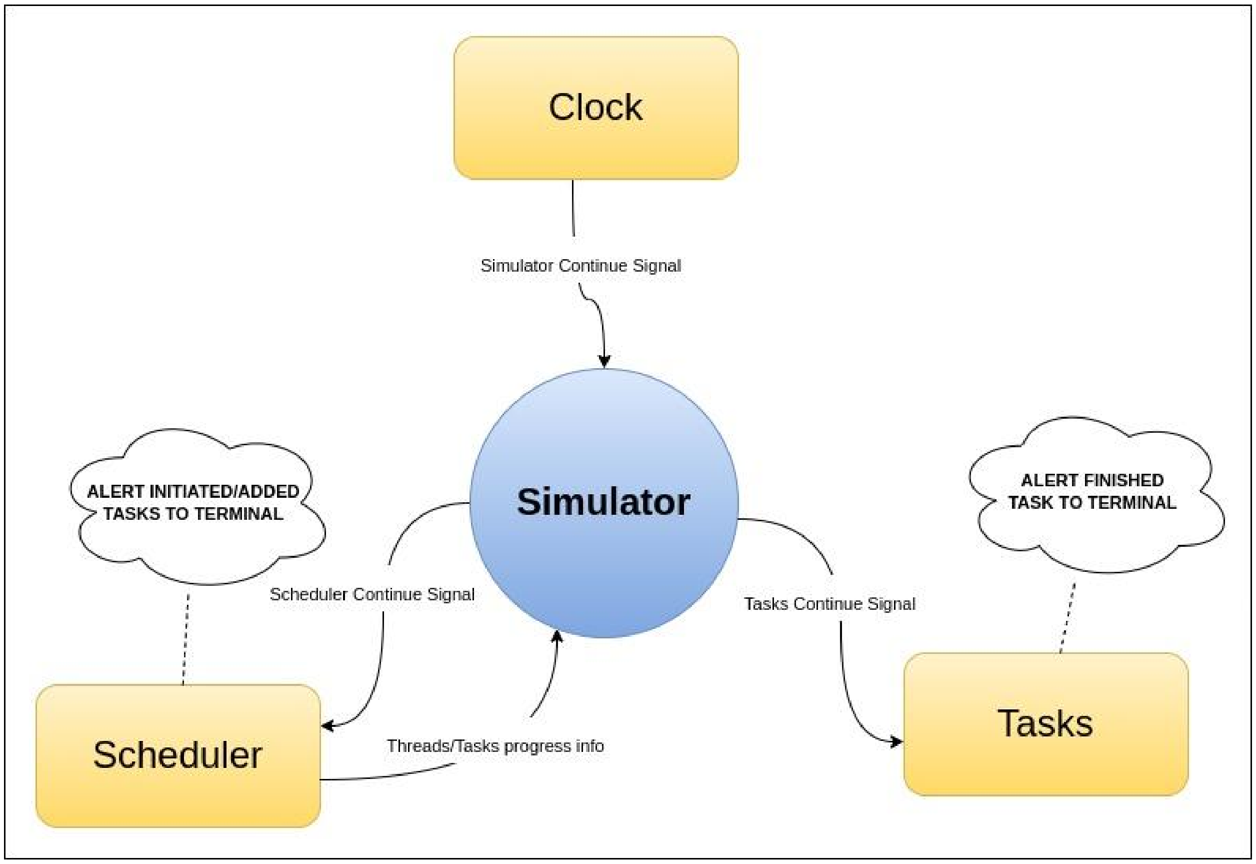
Introduction

We were asked to make a processor simulation as Java code, using Thread and Runnable classes, also we need to make sure that this code is valid upon SOLID principles and Design Patterns.

User will input 3 inputs, Number Of Threads that will be used, Number Of Cycles that the Simulator CPU Clock will use, and the name of the text file that contains dummy tasks written as Initial Time, Duration Time, Priority.

Some challenges are how to make sure that tasks are being put into threads in the correct order, and how to make sure that every component in this simulation is working in a synchronous way; we will learn more in the next sections.

Context Diagram

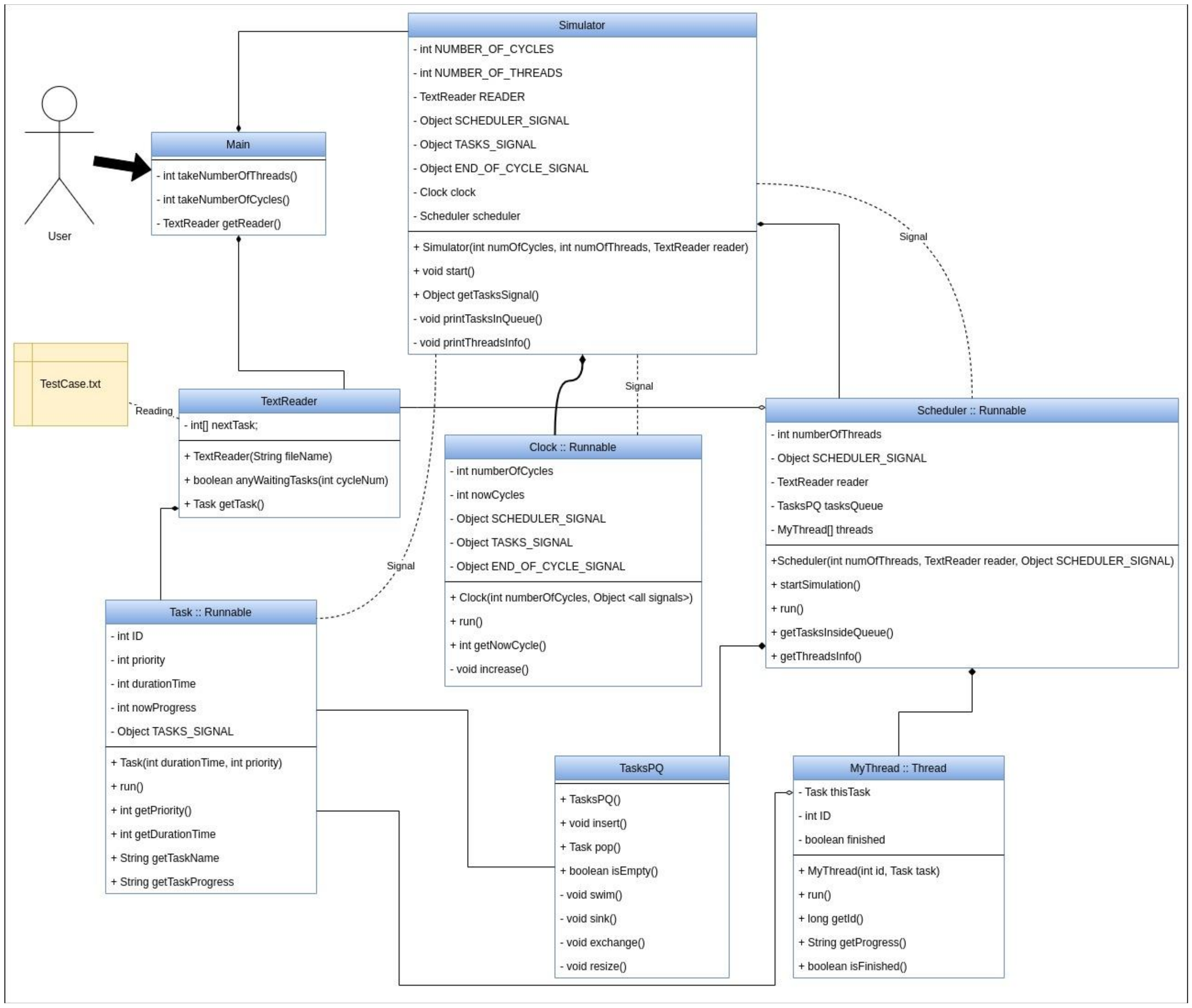

Clock
Simulator Continue Signal
Simulator
ALERT INITIATED/ADDED TASKS TO TERMINAL
ALERT FINISHED TASK TO TERMINAL
Scheduler Continue Signal
Tasks Continue Signal
Scheduler
Threads/Tasks progress info
Tasks

- **Simulator**: The main entity in the system, that will make sure all other entities can synchronize with each other by sending and receiving signals at a perfect time, will also show the progress and infoes of Tasks and Threads (we will learn more in next sections).
- **Clock**: A whole thread is responsible for the timing, the Clock will notify the simulator every second (every one cycle) that a new cycle has begun.
- **Scheduler**: Responsible for reading Tasks from a text file and putting them on empty threads in a correct order, will also send information to the Simulator entity about Threads and Tasks progress.
- **Tasks**: dummy tasks that take some time (in clock cycles) to be finished, Tasks will receive signals from the simulator every one second (one cycle) to make a one unit of progress, once the progress

is finished the task will be completed and will be replaced with a new one in the thread by the Scheduler.

UML Diagram

UML Image



High-Level Iteration

To Iterate the UML Class Diagram as a verbal story (no code):

- (main thread) User will execute the **Main class** that will be asking to input three things: numberOfThreads, numberOfCycles, and the .txt file.

- (main thread) A new **TextReader** object will be created and try to locate the .txt file that the user input.

- (main thread) **Main** will create a new **Simulator Class** and will start the simulation.

- (main thread) **Simulator** will create the signal Objects that are needed, new **Clock** in a separate thread, and new **Scheduler** in a separate thread; then **Simulator** is going to put its thread (main thread) on wait, waiting for signals from **Clock.**

- (scheduler thread) once **Scheduler** is created it will create a new **TasksPQ** object that will store new **Task**s in the future, and will create a new **MyThread[**Num Of Threads**]** which are the threads that will handle those **Task**s; then it will put its thread (scheduler thread) on wait, waiting the **Simulator** to notify so it continues its infinite loop.

- (clock thread) **Clock** will send signals to **Simulator** every one second so the (main thread) knows to continue working on its infinite loop, until the **Clock** hits the System.exit(0).

- (main thread) Each cycle the **Simulator Class** (main thread) will notify the **Scheduler**, then will print infos and progress of **MyThread**s and **Task**s, finally will notifyAll **Task**s to make a one unit progress.

- (scheduler thread) Once **Scheduler** (thread) been notified by **Simulator** it will use the **TextReader Class** to read new **Task**(s) if any needs to be initiated on this cycle, **Scheduler** will insert new **Task**(s) to **TasksPQ**, then will check if there are any empty **MyThread** to handle the next **Task** on the **TasksPQ**; then repeat the loop waiting the next signal from **Simulator**.

- (MyThread thread) Each takes one Runnable **Task**, each **MyThread** will be holding this **Task** until it is finished.

- (MyThread thread) **Task** will not make any progress until the **Simulator** (main thread) sends a notifyAll signal to **Task**(s) every one cycle (one second), after notifying, the **Task** will make a one unit of progress until the **Task** duration is reached.

Each Classes Has a Single Responsibility

Simulator Class

- Has a single responsibility which is initializing all the main components and making sure that they work in a synchronized way.

Clock Class

- Has a single responsibility which is notifying the **Simulator Class** each one cycle (one second).

Scheduler Class

- Has a single responsibility which is scheduling the **Task**s to **MyThread**s by the help of **other** classes reading new tasks, and putting tasks in the correct order.

TextReader Class

- Has a single responsibility which is reading/initializing new tasks from the .txt file that was provided from the user.

TasksPQ Class

- Has a single responsibility which is being a data structure that is taking tasks and sorting them in the proper order.

Task class

- Has a single responsibility which is that this class represents a task that was given from the .txt file, and capable of giving information about its attributes.

MyThread Class

- Has a single responsibility which is representing a thread in a cpu to take a task, capable to give infos about its additional attributes.

Technicals

How Is It Synchronized?

All the synchronization was made using the synchronized(Object) method, where I have 3 Objects in my code being created by **Simulator Class:** SIMULATOR_SIGNAL, SCEDULER_SIGNAL, and TASKS_SIGNAL.

SIMULATOR_SIGNAL: Is the object that holds the **Simulator** thread (main thread), the **Clock** thread is responsible to notify the **Simulator** thread every second using this object.

SCEDULER_SIGNAL: Holds the **Scheduler** thread, the **Simulator Class** is the one that is responsible to notify the **Scheduler** using this object.

TASKS_SIGNAL: Holds the Runnable **Task** (MyThread thread), the **Simulator Class** is the one that is responsible to notify the **MyThread** thread using this object.

Tasks In The Proper Order

Data Structure

I used a ***Priority Queue*** that stores and re-orders tasks using ***Binary Tree Heap***.

Once the problem of making sure that tasks must be in particular order, the priority queue popped in my mind as a first thought, and as far as my humble knowledge, representing this Priority Queue using Binary Tree Heap will give us the best complexity.

Complexity

The complexity of this data structure is:
Insert() : O(Log(n))
Pop() : O(1)

Which will let us be able to add as much as we want of tasks to this Queue without being afraid that sorting these tasks will take more than one cycle; that's even if we let the clock cycle be only 1ms it will take an infinite amount of tasks to be afraid of this complexity.

Test Cases

Test Case 1:

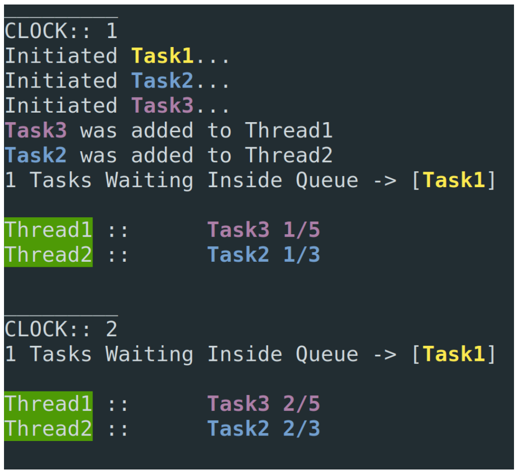
With 2 processors and 9 cycles, and this as a .txt file:

```
5
1 3 0
1 3 1
1 5 1
3 1 1
3 2 0
```

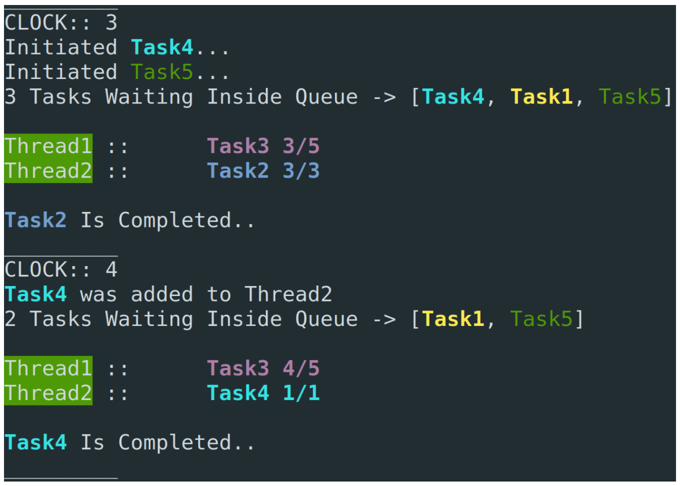
Solution must be like:

	1	2	3	4	5	6	7	8	9
Thread1	T2	T2	T2	T4	T1	T1	T1		
Thread2	T3	T3	T3	T3	T3	T5	T5		

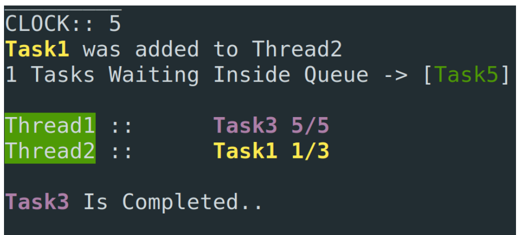
Code output:



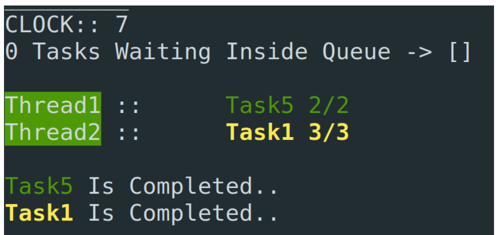
```
CLOCK:: 1
Initiated Task1...
Initiated Task2...
Initiated Task3...
Task3 was added to Thread1
Task2 was added to Thread2
1 Tasks Waiting Inside Queue -> [Task1]

Thread1 ::      Task3 1/5
Thread2 ::      Task2 1/3


CLOCK:: 2
1 Tasks Waiting Inside Queue -> [Task1]

Thread1 ::      Task3 2/5
Thread2 ::      Task2 2/3
```



```
CLOCK:: 3
Initiated Task4...
Initiated Task5...
3 Tasks Waiting Inside Queue -> [Task4, Task1, Task5]

Thread1 ::      Task3 3/5
Thread2 ::      Task2 3/3

Task2 Is Completed..

CLOCK:: 4
Task4 was added to Thread2
2 Tasks Waiting Inside Queue -> [Task1, Task5]

Thread1 ::      Task3 4/5
Thread2 ::      Task4 1/1

Task4 Is Completed..
```



```
CLOCK:: 5
Task1 was added to Thread2
1 Tasks Waiting Inside Queue -> [Task5]

Thread1 ::      Task3 5/5
Thread2 ::      Task1 1/3

Task3 Is Completed..
```



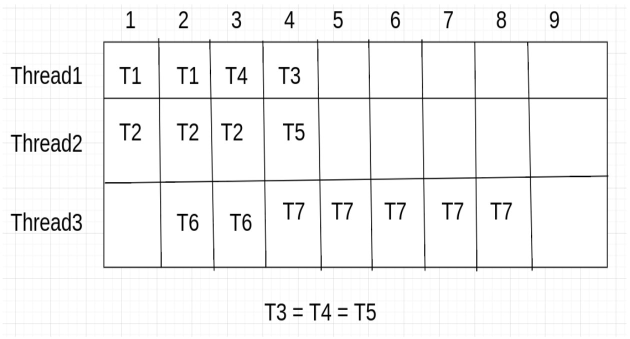
```
CLOCK:: 7
0 Tasks Waiting Inside Queue -> []

Thread1 ::      Task5 2/2
Thread2 ::      Task1 3/3

Task5 Is Completed..
Task1 Is Completed..
```

Test Case 2:

With 2 processors and 9 cycles, and this as a .txt file: —-->

```
7
1 2 1
1 3 1
2 1 0
2 1 0
2 1 0
2 2 0
4 5 0
```

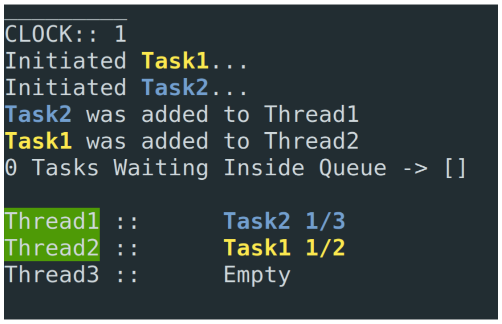
Solution must be like:



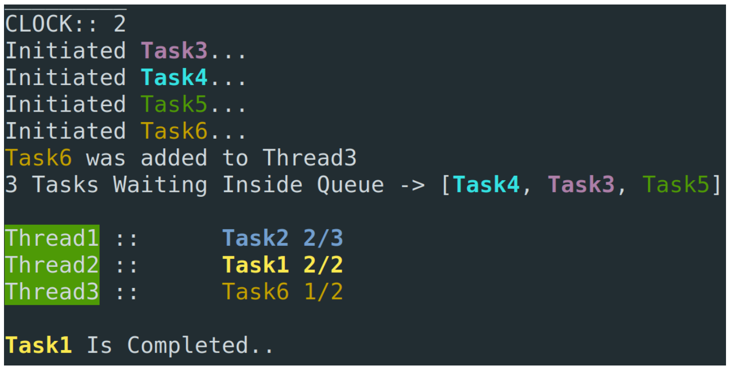
	1	2	3	4	5	6	7	8	9
Thread1	T1	T1	T4	T3					
Thread2	T2	T2	T2	T5					
Thread3		T6	T6	T7	T7	T7	T7	T7	

T3 = T4 = T5

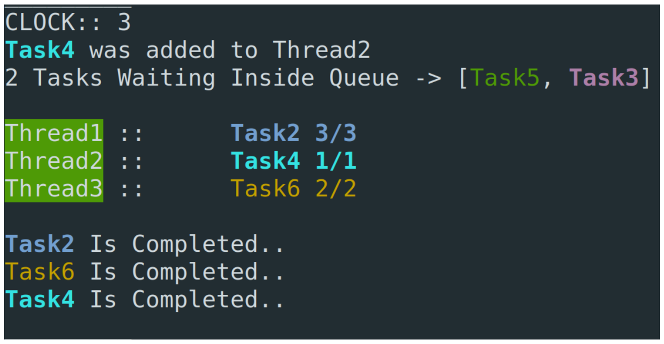
The Code:



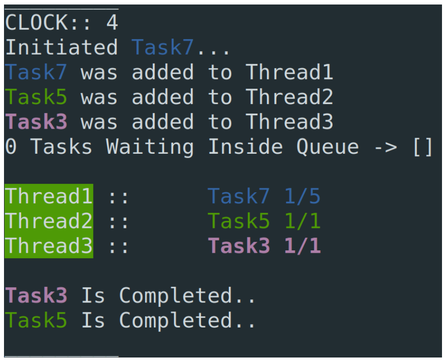
```
CLOCK:: 1
Initiated Task1...
Initiated Task2...
Task2 was added to Thread1
Task1 was added to Thread2
0 Tasks Waiting Inside Queue -> []

Thread1 ::      Task2 1/3
Thread2 ::      Task1 1/2
Thread3 ::      Empty
```



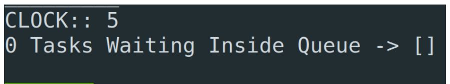
```
CLOCK:: 2
Initiated Task3...
Initiated Task4...
Initiated Task5...
Initiated Task6...
Task6 was added to Thread3
3 Tasks Waiting Inside Queue -> [Task4, Task3, Task5]

Thread1 ::      Task2 2/3
Thread2 ::      Task1 2/2
Thread3 ::      Task6 1/2

Task1 Is Completed..
```



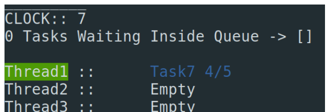
```
CLOCK:: 3
Task4 was added to Thread2
2 Tasks Waiting Inside Queue -> [Task5, Task3]

Thread1 ::      Task2 3/3
Thread2 ::      Task4 1/1
Thread3 ::      Task6 2/2

Task2 Is Completed..
Task6 Is Completed..
Task4 Is Completed..
```



```
CLOCK:: 4
Initiated Task7...
Task7 was added to Thread1
Task5 was added to Thread2
Task3 was added to Thread3
0 Tasks Waiting Inside Queue -> []

Thread1 ::      Task7 1/5
Thread2 ::      Task5 1/1
Thread3 ::      Task3 1/1

Task3 Is Completed..
Task5 Is Completed..
```



```
CLOCK:: 5
0 Tasks Waiting Inside Queue -> []
```



```
CLOCK:: 7
0 Tasks Waiting Inside Queue -> []

Thread1 ::      Task7 4/5
Thread2 ::      Empty
Thread3 ::      Empty
```