Practical Task 1

Aim: *Practical experience with scheduling*

Begin Date: *November 07, 2019*

End Date: *November 15, 2018, midnight*

Submission: *an archive file (zip or tar) that includes the source code of your solution should be submitted through MyMoodle. Submissions after the deadline are not accepted; if the submitted program does not run using the given instructions on how to run/test your code, your solution will not be accepted.*

Instructions: *you are allowed and encouraged to use the literature recommended in the course; use of other literature is also encouraged; assignment must be completed independently and without the help of other colleagues or the teacher;*

Problem Definition

Develop a Java program that simulates the First-Come First-Served Scheduling algorithm (see Chapter 5 of the book[1]). First-Come, First-Served (FCFS) is the simplest CPU scheduling algorithm. The basic idea of this algorithm is to dispatch tasks (processes) from a waiting list to the CPU. The criteria of how these tasks are dispatched to the CPU are based on the task's arrival time on the waiting list.
FCFS is a non-preemptive algorithm, which means that once a task is dispatched to the CPU it runs until it has finished.

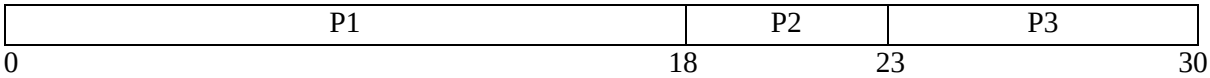
Example: Let's assume that we have a list of processes *P1*, *P2*, and *P3* (see Table 1), where the Arrival Time (AT) and Burst Time (BT) for each process is known. We need to calculate the Completed Time (CT), Turnaround Time (TAT) and Waiting Time (WT) for each task.

Table 1. A list of processes waiting for scheduling

Process Id	Arrival Time	Burst Time
1	0	18
2	2	5
3	4	7

We will use the Gantt chart to demonstrate the scheduling algorithm. Based on the arrival time of each task (that is the scheduling criteria of FCFS algorithm) the processes are dispatched to the CPU as follows: the first process dispatched to the CPU is *P1*, then *P2*, and at the end *P3*.

Figure 1. Gantt chart of the scheduling process



The Completed Time indicates the timeframe since first task has arrived, until the specific task has been completed. As *P1* arrived first, it is dispatched to the CPU. It will run for 18 time-units, therefore its completed time (CT) is 18. As soon as *P1* has finished its execution, *P2* is dispatched and will run

[1] Operating System Concepts, 9th Edition, by Abraham Silberschatz, Peter B. Galvin, Greg Gagne

for 5 time-units. The completed time of *P2* is 23. Finally, when *P2* has finished its execution, *P3* will start running for 7 time-units, and will finish at 30. Table 2 lists the CT, TAT, and WT for each of the processes listed in Table 1.

The Turnaround Time indicates the total time a task has spent since it has arrived until it has finished. We calculate TAT using the following equation: *TAT = CT – AT*. So, the turnaround time of *P1*, *P2*, and *P3* is 18, 21, and 26 time-units, respectively.

The Waiting Time indicates how much time a task spent waiting on the queue before it was allocated. We can calculate WT using the following equation: *WT = TAT – BT*. The waiting time of *P1* is 0, because it was dispatched as soon as it arrived. *P2* arrived at 2, however it can't be dispatched immediately because *P1* is running, therefore *P2* has to wait until *P1* has finished its execution (that is 16 time-units). *P3* arrived at 4, which means that it has to wait for both *P1* and *P2* to finish (that is 19 time-units).

Table 2. Scheduling a list of processes using the FCFS algorithm

Process Id	Arrival Time	Burst Time	Completed Time	Turnaround Time	Waiting Time
1	0	18	18	18	0
2	2	5	23	21	16
3	4	7	30	26	19

Solution

We have provided three Java classes:

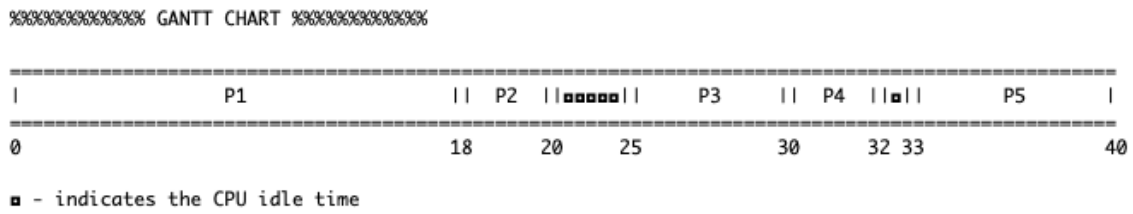
1. **Process.java** – that contains information (such as process id, arrival time, burst time, completed time, turnaround time, and waiting time) for each of the processes.
2. **FCFS.java** – that contains the *run(), printTable(),* and *printGanttChart()* methods, and a list (named processes) to store the processes.
3. **FCFSTest.java** – that contains JUnit test cases to check the correctness of your solution. Note that these programs (methods) test only some cases of scheduling scenarios. Feel free to extend this class with more methods that test different scheduling scenarios.

Your task is to implement the FCFS's methods: *run(), printTable(),* and *printGanttChart()*. The constructor of the FCFS class is already implemented. You are free to add additional methods to this class, but do not rename the existing methods. The implementation of the FCFS algorithm should be in the *run()* method. The *printTable()* is expected to print a table similar to Table 2 and *printGanttChart()* should demonstrate the scheduling process using the Gantt charts (see Figure 1).
An example of the output of *printTable()* method is provided in Figure 2. An example of the *printGanttChart()* function is provided in Figure 3.

Figure 2. An example of the output of printTable() function

```
----------------------------------------
PID     AT      BT      CT      TAT     WT
1       0       18      18      18      0
2       3       2       20      17      15
3       25      5       30      5       0
4       29      2       32      3       1
5       33      7       40      7       0
----------------------------------------
```

Figure 3. An example of the output of printGantChart() function



Hints: The example above shows a scenario when the processes are waiting for the CPU. Think about how to handle CPU idle times (that means CPU is waiting for processes).

Instructions for running and testing the application

Add JUnit4 to the build path in order to use the FCFSTest.java class to check the correctness of your solution.