CS 346 – Lab 6 – Non-preemptive scheduling

In this lab, we will investigate a couple of simple scheduling algorithms that an operating system may use to figure out when jobs should execute. Specifically, we'll look at the first 2 scheduling methods we discussed in class: "First come, first serve (or FIFO)", and "Shortest Job Next."

Create a new directory called lab06, and copy the 3 Java files from my /home/chealy/www/cs346/lab06 directory.

This simulation program defines a process as follows: The attributes are name, arrival time, execution time, time remaining to execute, start time, done time.
And the methods are

- Initial value constructor
- reset – so that we can restart the simulation of the same set of processes with a different algorithm
- toString
- getName
- getArrivalTime
- getExecTime
- getTimeLeft
- setTimeLeft
- reduceTimeLeft – we won't use this, but would be good for the pre-emptive strategies
- isDone
- setStartTime
- setDoneTime
- findResponseTime – very useful when we want to gague the efficiency of the schedule
- makeDone – sets the "time left" value to 0

Basically, your job involves two phases:

1. Implement a FIFO schedule.
2. Implement a "shortest job next" schedule.

Almost all the modifications you need to make will be in Driver.java. The Process class and the comparator are provided for OO convenience. The Driver.java file already contains an ArrayList containing 5 tasks we want to execute. For each process, we give it a name, and we set its arrival time and its execution time. The code has comments to help you.

As you test your schedule, you may need to watch out for the following issues:

- In the case of FIFO, what happens when the next job in the list has not yet arrived?
- How should we keep track of the response time of each process? (Hint: look at some of the attributes of the Process class.)
- In the case of Shortest Job Next, we also need to sort by shortest execution time, so it will probably be necessary to create a 2nd comparator class that's similar to the one that allows us to sort by arrival times.
- We may have a situation (in Shortest Job Next) where the "shortest" job has not yet arrived. Similarly, there may be no shortest job currently because there are no jobs waiting. We also need to detect when all the jobs are finished.

As a guide, here is the output when I wrote the scheduling program.

```
Let's try FIFO scheduling.
Average response time = 14.2
Process A arrives 0, requires 8, 8 remaining :: startTime = 0, doneTime = 8
Process B arrives 1, requires 4, 4 remaining :: startTime = 8, doneTime = 12
Process C arrives 2, requires 9, 9 remaining :: startTime = 12, doneTime = 21
Process D arrives 3, requires 5, 5 remaining :: startTime = 21, doneTime = 26
Process E arrives 50, requires 10, 10 remaining :: startTime = 50, doneTime =
60

Let's try Shortest Job Next scheduling.
EXECUTING A
EXECUTING B
EXECUTING D
EXECUTING C
need to advance time to 50
EXECUTING E
Average response time = 13.4
Process B arrives 1, requires 4, 0 remaining :: startTime = 8, doneTime = 12
Process D arrives 3, requires 5, 0 remaining :: startTime = 12, doneTime = 17
Process A arrives 0, requires 8, 0 remaining :: startTime = 0, doneTime = 8
Process C arrives 2, requires 9, 0 remaining :: startTime = 17, doneTime = 26
Process E arrives 50, requires 10, 0 remaining :: startTime = 50, doneTime =
60
```

When you are finished with the lab, please send me all of your source files.