Batch system simulation

In this lab, you’ll complete a simulation of a computer system that runs programs (called jobs) which users submit ahead of time. This type of system is called a batch system to distinguish it from the (now) more common interactive systems that immediately responds to user requests.

Begin by downloading the given code lab7given.zip from the course webpage and importing it (as an existing project). This code has several classes. The first is Event. An Event object represents something that will happen at a specific time. It has a method happen which will be called when that time is reached. ArrivalEvent is a subclass of this; its objects represent the arrival of a job. The main function is in Simulator. Objects of the Simulator class maintain a priority queue of scheduled events. The priority queue orders the events by the time they happen. The Simulator method run runs the simulation by repeatedly pulling events from the queue and calling their happen method.

After you’ve had a chance to look at the code a bit, compile and run it. You’ll get a series of messages:

0: A job of duration 10 arrives
1: A job of duration 5 arrives
2: A job of duration 4 arrives

The numbers are the times at which each of these events occurs. These are the three jobs added in the main method. Change the order in which they are added, recompile, and rerun the program to verify that the event order is not affected by the order in which the events are added to the priority queue.

For our first substantive change to this program, let’s have it tell when each job finishes. To do this, create a class DepartureEvent that extends Event; use ArrivalEvent as a model. You’ll need a constructor that takes a time and calls the Event constructor (called super since Event is the superclass of your new class). You’ll also need a happen method that prints a message about the job finishing. Then in the happen method of ArrivalEvent, create one of these objects and add it to the priority queue (events). You are simulating a situation where each job starts as soon as it arrives. Each job finishes duration time units after it started.

Once the program is updated, you should have output like the following:

0: A job of duration 10 arrives
1: A job of duration 5 arrives
2: A job of duration 4 arrives
6: A job finishes
6: A job finishes
10: A job finishes

The first two jobs to finish are the ones that arrived at times 1 and 2. The last job to finish is the one that arrived at time 0.

In actual systems, it is often not possible to run each job as soon as it arrives. For example, suppose that our simulated system could only run one job at a time. Then the first job would already be running when the others arrived, causing them to wait. One of those jobs would run after the first job finishes at time 10 and the other would start once it finished.

To simulate this kind of system, you need to store jobs that have arrived but which have not yet been started. In keeping with the theme of the lab, you should use a priority queue. This means that waiting jobs will run in order of increasing duration (an algorithm which is called Shortest Job First or Shortest Processing
Add a `PriorityQueue<Integer>` to the `Simulator` class. Also add a boolean to indicate whether the system is currently running a job. Modify the constructor to initialize both of these. Next, modify the `happen` method of `ArrivalEvent` so that it adds the newly arrived job (represented by just its duration) to the priority queue. Then modify the `happen` methods of both `ArrivalEvent` and `DepartureEvent` to check if the queue is non-empty and start a job if possible (i.e. if the boolean indicates that the system isn’t already running something). You’ll also need to make sure that the boolean is properly updated at all times.

Once all this is done, your output should be

0: A job of duration 10 arrives  
1: A job of duration 5 arrives  
2: A job of duration 4 arrives  
10: A job finishes  
14: A job finishes  
19: A job finishes  

If you have more time, see if you can modify the simulation to run jobs in order of decreasing duration (i.e. run the longest one first). To do this, you’ll want to make a `Comparator` specifying this order and pass it to the `PriorityQueue` constructor. A `comparator` embodies a way to compare objects (integers in our case). In Java, it is a class that implements `Comparator<Integer>`: `implements` is like `extends` but for interfaces rather than classes. Your class just needs a method `int compare(Integer o1, Integer o2)`. The return value of this method should be 0 if the two arguments are equal, negative if `o1` should come first (in our case, if it is larger), and positive if `o2` should come first. To get the `int` values of the `Integer` objects, use the `intValue` method of `Integer`. Once you’ve written your comparator class, create an object of the class and pass it to the `PriorityQueue` constructor:

```java
... = new PriorityQueue<Integer>(10, comp);
```

where `comp` is the name of the comparator object. The 10 is an initial capacity of the priority queue; unfortunately, the `PriorityQueue` class doesn’t have a constructor that just takes the comparator.