Homework 5
Due February 28, 2017 at 10:30 AM

3 Required Problems (85 points), and Style and Tests (15 points)
DO NOT modify methods or method headers that were already provided to you.
DO NOT add public or protected methods.
DO introduce helper methods that are package-private (i.e., do not have a privacy modifier) as you deem necessary.

Motivation

In this assignment, you will be implementing a UNIX-esque scheduler. As you will learn if you choose to take CIS 380, a scheduler is a critical component of a computer’s operating system. You can think of an operating system as a software interface between someone who writes program applications - like your web browser, or a music player - and the many parts that make up your computer, including the processor, memory, hard disks, and I/O devices (e.g. your display, keyboard, mouse). With the details of managing these resources abstracted away, programmers and users have a simpler and cleaner model of a computer to work with. If you’ve ever used a shell, like Terminal on OS X or Windows Command Prompt, you have been interacting directly with your machine’s operating system.

So what is the role of the scheduler? Put simply, a scheduler manages processes, or the programs currently executing on your computer. At any given point in time, out of all viable processes, and a scheduler must decide which to run and for how long. You can see the processes your system is managing by running the top command in Terminal on OS X, or by opening Task Manager on Windows and viewing the Processes tab.

The scheduler’s task is more complex than it might seem at first glance. There are tradeoffs involved such as maximizing throughput (the speed at which processes are completed) versus preventing starvation (the scenario in which a certain process is never scheduled). Furthermore, some processes may be more urgent to run than others: we say that these processes have “low” priority values. You will implement several different scheduling algorithms in this assignment, allowing you to examine their advantages and disadvantages.

Part One: Implementing a Deque (25 points)

Files to submit: ResizingDeque.java, ResizingDequeTest.java

In this part, you will implement a deque to use as the basic data structure backing your scheduler. “Deque”, pronounced like “deck”, stands for double-ended queue, and generalizes both queues and stacks: you can add/enqueue (offer) and delete/dequeue (poll) from the structure at either end front or back. The
usual queues only support enqueueing at the back and dequeueing at the front. Stacks support adding (push) and deleting (pop) at the front. Your deque should also support null entries as elements. You will be implementing the Java Deque interface. You can find the documentation here.

Note that ResizingDeque extends AbstractDeque. We have provided the abstract class AbstractDeque in order to reduce the amount of cruft you have to write to implement the Java Deque interface.

There are some points of difference between your expected ResizingDeque and the Java Deque, however. When implementing the ResizingDeque methods for add*, poll*, and offer*, you do not need to worry about throwing a ClassCastException. You also do not have to throw an IllegalArgumentException for the add* methods. Additionally, the add() method should always return true. Due to the resizing nature of your implementation, your data structure will have no notion of a maximum capacity.

Implementation: Resizing Array

You will implement a deque using array resizing, as discussed in class. Inside ResizingDeque, you should have an array for storing the elements of your deque. The array should start at 2 elements, and it should double in size (resize up) every time it fills up. That is, whenever the addition of a new element would cause the deque to exceed the size of the array. Whenever the removal of an element would cause the number of elements in the deque to fall to less than one quarter of the size of the array, you should cut the size of the array in half (resize down: don’t want to waste space!).

Your implementation will also want to use a concept known as wraparound. That is to say, after a series of pushes and pops, you may have a case where the head of your deque is not at array index 0, but maybe at an index farther along the array (like array index 4). After some more pushes, the index of the tail of your deque will get to the end of the array. The next push should then be at index 0, then index 1, etc.

Note that two of the unimplemented Deque methods in the AbstractDeque return Iterators. This homework assumes that you are familiar with iterators (textbook pp.138–9 as well as the Javadocs for the interface Iterator). For this homework, you do not need to implement remove for your iterators—in Java 8 remove is a default method that throws an UnsupportedOperationException. Moreover, since you cannot submit any additional files, your iterators must be private inner class(es). These two methods should return instances of the respective private inner class.

You should also think about efficiency when you are implementing resizingArray. For instance, add, delete functions should not need to copy the entire array. That is to say, most operations should be in-place, meaning that there’s no need for additional space. You will be taken points off if your implementation is not optimal in terms of space or runtime.

Note that Deque<E> is generic, so your implementation must also be generic; therefore the underlying array of elements must be of type E[]. Unfortunately, Java doesn’t cleanly support generic arrays, so the code to initialize a generic array of type E of size 2 in Java is

\[
E[] \quad \text{elements} = (E[]) \quad \text{new} \quad \text{Object} \quad [2];
\]

While your implementation must be generic, you should also be able to support adding and removing null elements to and from your data structure.
Note on loitering

When you implement any method that removes an element from the array (such as pollLast, removeFirstOccurrence, etc.), you should null out the associated entry in the array. Why should we do this? If we do not do this, then references to that element will “loiter” in the array. Java is a garbage-collected language. As long as we still have references to an object, then those references will prevent the reclamation of the memory used.

Part Two: Scheduler (50 points)

Files to submit: SchedulerImpl.java, SchedulerImplTest.java

In this part, you will be implementing your scheduler using your newly created ResizingDeque. Your scheduler will receive processes and maintain a notion of CPU time to determine how to handle them, beginning at \( t = 0 \).

Processes are passed to your \texttt{Scheduler} as a subclass of \texttt{OSProcess}. Each has a unique non-negative process ID (PID), as well as a (not necessarily unique) number of clock cycles required to complete the process, and priority, which will be used later. You can use \texttt{runProcess()} to simulate the execution of the process, and decrease its remaining run time. You are not required to submit this class, but should create one for testing purposes. Your \texttt{OSProcessImpl} should implement \texttt{OSProcess} and have a constructor that takes in three arguments. (A PID, a Priority, and a Run Time)

Scheduling

An operating system can only run one process at a time. Because of this, we will represent the timeline of which processes we are running at a given time as an integer array: The indices of the array represent the current clock cycle, and the values represent the PID of the process running at that time. If a process takes \( n \) clock cycles to run, it should take up \( n \) indices somewhere within this history after sufficient time has passed. The locations a given process appears at in this history depend on the algorithm we use.

If no process is running at a given time, you should enter the value \(-1\) in the timeline at the corresponding location. You will create this array in \texttt{getProcessTimeline()}. The last index of the array should be the most recently simulated cycle.

The scheduler receives processes through the \texttt{submit()} method, which takes in a \texttt{OSProcess}. The request time of the submitted process is determined by the number of clock cycles the scheduler has already executed via the \texttt{simulateClockCycles()} method. Your scheduler should be able to produce the correct timeline after an arbitrary number and ordering of processes is submitted. This also means your scheduler should handle any way you divide up the simulation time so long as the same processes were submitted at the same times.

In the case where multiple new processes are submitted at the same time \( t \), we consider processes to have arrived in the order in which they are submitted.

Your scheduler class should have a constructor which takes in a \texttt{SchedulingAlgorithm} enum, and uses
it to decide which algorithm to use. It should also take in a `timeQuantum` for the cases that the chosen algorithm is Round Robin or Multilevel Queues.

**FIFO (First In, First Out)**

This simple algorithm runs to completion the processes in the order they were received, regardless of how long any individual process takes. When a process arrives, it is sent to the tail of the queue of processes waiting to be run.

Remember that *request time* is not a field of `OSProcess` and is the number of cycles already simulated by the scheduler. So, if a process begins at $t = 2$, it implies that two cycles had already been simulated before it was submitted.

Consider two processes that were submitted to the scheduler:

- $p_1$ (PID = 1) with request time 2 and run time 3
- $p_2$ (PID = 2) with request time 3 and run time 2

Meaning we simulated 2 cycles, submitted $p_1$, simulated another cycle, submitted $p_2$, then simulated another 4 cycles.

A call to `getProcessTimeline()` will produce the array shown below:

<table>
<thead>
<tr>
<th>Time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process</td>
<td>-1</td>
<td>-1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Observe how $p_2$ waits for $p_1$’s completion before it begins executing. When no process is running or waiting to be run, the array should contain the value $-1$ at that index. You should use your `ResizingDeque` to represent the queue of processes waiting to be run in this and future algorithms.

**Round Robin**

Imagine you wish to browse the web and listen to music at the same time, while running a MATLAB computation in the background on your computer. As you can see, FIFO falls short here, as you would only be able to do one of these at a time. It is often beneficial to avoid running processes for too much time, and to give other processes a chance to run before the previous process completes. In Round Robin, we initially execute processes in the order we encountered them in, as with FIFO.

This algorithm only allows processes to be run for a set number of cycles dictated by the integer `timeQuantum`. After a process runs for `timeQuantum` cycles, it is moved to the tail of the process queue.

We consider processes that have already been partially run to enter the process queue before new processes arrive, so there’s no need for special handling of this case.

Consider three processes that were submitted to the scheduler, with `timeQuantum = 2`:
p1 (PID = 1) with request time 1 and run time 3
p2 (PID = 2) with request time 2 and run time 2
p3 (PID = 3) with request time 4 and run time 2

After simulating up until \( t = 7 \), a call to \texttt{getProcessTimeline()} will produce the array shown below:

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process</td>
<td>-1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

p1 (with run time 3) cannot be fully executed within the \textit{timeQuantum}. So after \textit{timeQuantum} has elapsed, we insert it at the tail of the process queue, maintaining its remaining run time. In this case, p1 has 1 clock cycle remaining. Then, p2 has a chance to run. Because its run time is no larger than \textit{timeQuantum}, it completes before it is forced to the back of the queue. Since p3 arrives after p1 has been added to the tail of the queue, it runs after p1 gets its second chance to execute.

For this algorithm and Multilevel Queues, \textit{timeQuantum} should reset when the process being executed changes.

**Shortest Job First**

Not all processes are created equal. There are certain user interactions that one will expect to be more responsive than others - for example, you might be okay with having a half-second delay in your file download, but it would be very annoying if there was a half-second lag for every key you typed on your keyboard.

Shortest job first, or SJF, will always attempt to run the process with the smallest run time first, regardless if there is another process currently being run. If a process with a run time of 3 arrives while one with 4 cycles of run time remaining is currently being executed, the new process will replace the current one. When a process completes, the process with the next smallest run time is executed.

If multiple processes have the same remaining run time, we execute the one with the lowest \textit{PID} first.

Consider three processes that were submitted to the scheduler:

p1 (PID = 1) with request time 0 and run time 4
p2 (PID = 2) with request time 1 and run time 2
p3 (PID = 3) with request time 2 and run time 2

After simulating up until \( t = 7 \), a call to \texttt{getProcessTimeline()} will produce the array shown below:

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

p1 arrives at \( t = 0 \), and is executed as it is the only process waiting to be run. At \( t = 1 \), p2, with 2 run time arrives, which is less than p1’s remaining 3 run time, so p2 replaces p1 and begins execution. Now, at
time $t = 2$, a process $p3$ with run time 2 arrives. Since $p2$ only has 1 clock cycle of execution time remaining, less than $p3$’s 2, it continues running. Once $p2$ completes, the next shortest process, $p3$ is fully processed, followed by $p1$.

You may use an instance of `java.util.PriorityQueue` to store your processes in this part. A `PriorityQueue` by default has an implementation as a min-heap, producing efficiently the minimum value it stores. By default, `PriorityQueue` only accepts `Comparable` objects, but our `OSProcess` does not implement `Comparable`. You should be able to find a workaround for this.

**Multilevel Queues**

We have established the notion that some processes are more important to run than others. This notion is encapsulated in the idea of *priorities*. In a real operating system, a process’s priority is a combination of what is assigned to it by the operating system itself as well as an optional priority that the user may assign. For our final algorithm, Multilevel Queues, we aim to generalize Round Robin while also factoring in a process’s priority, resulting in an algorithm that is more flexible than the previous ones in scheduling processes.

You will use the `getPriority()` method of `OSProcess` to get a process’s priority. Somewhat unintuitively, the lower the value of `priority`, the greater the corresponding process’s “importance” and hence the sooner our algorithm will run it. Priority ranges between 0 and `Integer.MAX_VALUE`.

In Round Robin, once a process elapses `timeQuantum` cycles of run time, it is sent to the tail of the process queue. In Multilevel Queues, however, it is sent to the tail of the process queue of one lower priority. So, if a process $p1$ with priority 0 runs out of time on its current queue, it is inserted at the tail of the priority 1 queue.

In Multilevel Queues, the priority value associated with each process represents the initial queue the process is sent to. As such, you may have to store many different priority queues simultaneously. Similar to Round Robin and FIFO, when a process arrives, it is inserted at the tail of its corresponding queue. You should always run the highest priority process available. When a queue of a given priority is empty, begin executing processes at the next highest priority queue. Again, if multiple new processes arrive in the same queue at the same time, we consider the one submitted first as having arrived earlier.

For simplicity, we will only consider the queues with priority 0, 1, and 2. If a process elapses `timeQuantum` while in the queue of priority 2, it should be reinserted at the tail of the priority 2 queue.

If a process on a low priority queue is running when a higher priority event arrives, your scheduler should wait until `timeQuantum` elapses executing the current process before executing the higher priority process. You should *not* cut the execution time of any process short, as you did in Shortest Job First.

Consider four processes that were submitted to the scheduler, with `timeQuantum = 2`.

- $p1$ (PID = 1) with request time 0, priority 0, and run time 4
- $p2$ (PID = 2) with request time 2, priority 1, and run time 3
- $p3$ (PID = 3) with request time 3, priority 2, and run time 2.
p4 (PID = 4) with request time 4, priority 0, and run time 2.

After simulating up until \( t = 10 \), a call to `getProcessTimeline()` will produce the array shown below:

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

Process \( p_1 \) arrives and is added to queue of priority 0, where it runs for 2 clock cycles before dropping a priority level. Then it arrives to the priority 1 queue, which it enters ahead of \( p_2 \). So \( p_1 \) runs another two cycles and completes.

While it is running, \( p_3 \) arrives in queue 2. Now, at \( t = 4 \), \( p_4 \) arrives in queue 0, where it becomes the highest priority process available. So, we run \( p_4 \) for \( timeQuantum \), where it completes.

Then, in queue 1, \( p_2 \) runs for \( timeQuantum = 2 \) clock cycles before dropping to the tail of the priority 2 queue. From there, \( p_3 \) runs to completion, and then \( p_2 \) finishes processing.

**Part Three: SchedulerFactory**

Files to submit: `SchedulerFactoryImpl.java`

To finish off your Scheduler, you will build a subclass of `SchedulerFactory`. The factory will have methods for the creation of each the scheduling algorithms. You should instantiate your `Scheduler` according to the method being called.

**Part Three: Asymptotic Analysis (10 points)**

Files to submit: `AnswersToDequeQuestions.java`

You will thus answer some questions about the running time complexity of your code. The questions are specified in the interface `DequeQuestions.java`, and you should provide your answers by filling out the return values in `AnswersToDequeQuestions.java`. The following questions are also specified in the comments to each method, and you can answer each question through code using the `RunningTimeAnalysisAnswer` enum we've provided in the `edu.upenn.cis121.hw5` package.

Consider a new implementation of `Deque` using two singly linked lists as the underlying data structure, called `LinkedListDeque`. Notice that you should perform an amortized running time analysis of the `ResizingDeque` methods. (By amortized, you may assume that any given offer or poll won’t resize the array, since it happens so infrequently.)

1. What is the running time of `LinkedListDeque.contains()`?
2. What is the running time of `LinkedListDeque.offerLast()`?
3. What is the running time of `LinkedListDeque.pollFirst()`?
4. What is the running time of `ResizingDeque.contains()`?
5. What is the running time of `ResizingDeque.offerLast()`?
6. What is the running time of `ResizingDeque.pollFirst()`?

**Style and Tests (15 points)**

The above parts together are worth a total of 85 points. The remaining 15 points are awarded for code style, documentation, and sensible tests. Style is worth 5 of the 15 points, and you will be graded according to the [121 style guide](https://example.com/121_style_guide).

Also, please refer to the [Java testing guide](https://example.com/java_testing_guide) on the course website.