Homework # 7
due Monday, March 20, 10:00 PM

(Due after exam week, but we strongly recommend you finish this before the exam.)

In this assignment we implement an endogenous cyclic doubly-linked queue with a dummy. Check the “Linked List Variations” handout on the class webpage to refresh in your mind what each of these variations entail. When finished, we’ll put this ADT to work in an interesting simulation of what goes on inside a computer.

1 The Data Structure

In a previous assignment (Homework 6), you implemented a generic LinkedCollection class using a cyclic singly-linked list with a dummy node. In lab 7, you will investigate the FIFO property of queues. This assignment will combine aspects of the two with some key differences. Let’s break down all the properties of this new ADT:

endogenous As in Homework 5, the links to the other elements in the list will be stored inside the data structures themselves. This means there will be no separate “node” class to hold the links.
cyclic As in Homework 6, the last element in the list will point back to the head element.
doubly-linked There will be two links per element, one to the next and one to the previous element.
Queue Operates on the First-In-First-Out (FIFO) property. This means the first element to be added should also always be the first element to be removed.

with a dummy The head element of our queue will always be a special ”dummy” that contains no meaningful data. When the queue is empty, the next and prev fields of the dummy element will both point back to itself; after elements are added, the dummy element’s next pointer refers to the first element in the queue, while its prev pointer refers to the last element in the queue.

For example, the following picture shows a Queue with three elements, p1, p2 and p3.
We will be extending the `AbstractQueue` class to lessen the amount of methods we need to implement, however we’ll need to override a number of methods to ensure they are efficient. Before we can delve into the Queue implementation let’s take a look at what each data element will consist of.

2 The Process class

Our queue will consist of endogenous elements representing processes inside a computer. Each process will contain the count of instructions it needs to perform before it’s considered finished (Note: many processes in a computer, however, run continuously and so never really "finish", but for our assignment we will pretend they do). A process object itself will maintain the links to the other processes in its queue, enforcing the endogenous property and removing the need for a node class. The `Process` class will provide various getters and setters, as well as a `performInstruction()` method which will increment it one step closer to finishing.

2.1 How are Processes Organized?

As you may know, modern computers can run many processes (seemingly) simultaneously. However, before the advent of multiple cores for a single processor, many of these processes needed to fight for same space to get their work done: the CPU. How was it possible to balance all these requests in a fair and logical way?

Enter: The Process Scheduler!

The process scheduler was devised as the authority inside a computer as to which process was able to use the processor and when. The scheduler hand-selects a process, feeds it to the CPU, and after some amount of time would take it out and fill the CPU with another process. There are many disciplines by which schedulers decide which process to give to the CPU next. We will be using two:

**Strict First-Come, First-Serve** As you might guess, this discipline aligns perfectly with our ADT. This scheduler will allow processes to be scheduled into its ready queue, and when the CPU is ready will feed it the process at the front of the queue. It will allow the CPU to work on the process until the process is done, then receive it back and place it with other processes that are done (but not back in the ready queue!). This is the simplest discipline in scheduling.

**Round-Robin** This discipline is the polar opposite of the other. It also maintains a ready queue, from which it feeds the front process to the CPU. In Round-Robin, however, this scheduler only allows the CPU to perform a single instruction before it asks for the process back. This has the benefit of allowing all processes equal access to the CPU, however it has its own unique pros/cons to also be illustrated in the driver.

You’ll find when running the driver that both disciplines can achieve greater ”success” in different scenarios (success defined as more processes finishing sooner). Modern schedulers may switch between these and other disciplines depending on the needs of the system at the time, gaining the benefits of both and the drawbacks of neither! Another major theme among scheduling disciplines is scheduling processes by priority, which requires the use of a PriorityQueue (another interesting ADT worthy of investigation).

3 The Queue class

We will implement our Queue class directly inside the Process class, giving it handy access to the links of each process. As mentioned before, we’ll be extending `AbstractQueue`, and the list of methods you’ll need to implement is provided below. Note that queues have a different vocabulary for some common operations: "poll" instead of "remove", "offer" instead of "add". While these serve similar functions, it’s useful to note that "poll" will always be removing from the front of the queue, and "offer" will always be adding to the back.
3.1 The Iterator

We will continue implementing a fail-fast iterator by using version stamps. Because the doubly-linked list provides us the ability to move in both directions, we can simply use a cursor field instead of a precursor. Since it’s only legal to remove from the front of a queue, we cannot add "remove" functionality into our iterator. It will simply be a handy way to look at all the elements in our queue.

4 The Invariant

As always, we suggest implementing the invariant checks before implementing the other methods, as invariant checking will make writing correct methods easier. Because we use a doubly-linked list, the invariants are easier to check and are listed inside the class.

Additionally, we require that there is no dummy data for all the processes except the dummy process. For a Process object, we will consider dummy data to be a null name and 0 total instructions. The iterator has its own set of invariants as well. As before, do not check the iterator invariants if the versions do not match – if the iterator is stale, being well-formed or not is not important.

5 What You Need To Do

5.1 Process.Queue

boolean wellFormed() The queue invariant (details inside class).

Queue() The constructor.

boolean offer(Process) Add the parameter process to the end of the queue. We won’t allow null or a process that’s currently in another queue to be added to this one.

boolean takeAll(Queue) Removes all processes from the parameter queue and puts them at the end of this queue. The parameter queue will be empty after this method.

Process peek() Return (but don’t remove) the process at the front of the queue.

Process poll() Remove and return the process at the front of the queue.

int size() Return the count of real processes in the queue.

Queue clone() Return a deep clone of this queue. Any subsequent changes made to this queue or its processes should not affect the copy or the copy’s processes. Since this queue is endogenous, cloning the queue will entail cloning the processes too.

5.2 Process.Queue.MyIterator

Details inside class.

5.3 Scheduler

void step() Imitates the behavior of a Strict-FIFO or a Round-Robin scheduler during each pulse of the system’s clock. If a Strict-FIFO scheduler, then each process should be left in the CPU until it’s done. If it’s Round-Robin, the CPU should be loaded with a process, called to execute an instruction, and unloaded all in the same Scheduler step. More details inside class.

See the comments and Javadoc in the skeleton files for more details.
6 Files

The git repository for this homework contains the following files:

6.1 JUnit Tests

src/UnlockTest.java Unlock all the tests without running them.

src/TestQueue.java Tests for the Queue data structure.

src/TestInvariant.java Tests for the invariant.

src/TestScheduler.java Tests for the Scheduler class.

src/TestEfficiency.java Tests for the efficiency of all the operations.

6.2 Skeletons/Driver

src/edu/uwm/cs351/Process.java This contains the skeletonized Queue class.

src/edu/uwm/cs351/Scheduler.java The Scheduler class.

src/edu/uwm/cs351/CPU.java A (mostly symbolic) CPU class.

src/edu/uwm/cs351/Driver.java The driver with which to see the fruits of your labor. You may load it with scenarios and watch how the Strict-FIFO and Round-Robin scheduling disciplines perform in different situations.

scenarios Provided scenarios for the driver. Feel free to add your own using the format inside the file and let us know if you notice anything interesting about how they perform!