Concurrency

Building thread-safe data structures

In your `recitation/12` directory we have provided a `Queue` interface and an unsynchronized, linked-list-based `Queue` implementation, `UnsynchronizedQueue`. If multiple threads access the same queue concurrently, race conditions can occur and you might obtain unexpected results.

In this recitation you will use primitive Java synchronization to implement an *unbounded blocking queue*. An unbounded blocking queue is a normal queue except if the queue is empty upon a dequeue request, then the request is blocked until an element is enqueued later in another thread. This behavior contrasts with the standard Java queue specification, which either returns `null` or throws a `NoSuchElementException` depending on which dequeue method is called; with an unbounded blocking queue the dequeue method will always return some valid element from the queue, although a thread might need to wait arbitrarily long to dequeue an element from the queue.

To complete this recitation you should:

1. Run the tests in `tests/edu/cmu/cs/cs214/rec12/queue` and understand their expected behavior. These tests will initially fail on all versions of the queue, because the `UnboundedBlockingQueue` and `FineGrainedUnboundedBlockingQueue` are just copies of the `UnsynchronizedQueue` in the code that was distributed. You don't need to understand the details of the tests' implementations — especially the second test, which abuses Java reflection to access a queue's private data.
2. Inspect the `UnsynchronizedQueue` and understand its representation. Discuss which data is subject to race conditions if the queue is accessed by multiple threads and how you can eliminate those race conditions.
3. Using basic Java synchronization and the `wait` and `notify` methods, eliminate the race conditions in the `UnboundedBlockingQueue` and make it a correct unbounded blocking queue. In other words, enqueueing an element should always succeed immediately because the element can always be appended in a new node at the end of the queue. An attempt to dequeue from an empty queue, however, should block until an element has been enqueued by another thread. To simplify your implementation, use a *coarse-grained* locking strategy to eliminate race conditions. In other words, prevent race conditions by allowing only one thread to enqueue or dequeue at a time. Use the provided JUnit tests to evaluate the correctness of your implementation. Look at the sample concurrency code from lecture for more details.
4. Discuss how you could use multiple locks to allow multiple threads to enqueue or dequeue from the queue simultaneously, and discuss the trade-offs of coarse-grained (here, one lock) vs. fine-grained (multiple locks) synchronization strategies. As time permits, implement a fine-grained locking strategy in `FineGrainedUnboundedBlockingQueue`.