Further Java Ticklet 3*

In order to gain a star in the mark sheet you must complete this exercise. Completing the exercise does not gain you any credit in the examination. In Workbook 3 you wrote `SafeMessageQueue`, a queue which permitted concurrent access by multiple producers and consumers. In your implementation, each producer and consumer obtained a lock on the `SafeMessageQueue` object before adding or removing items from the queue. When there are more than 2 items in the queue, it's possible for the producers and consumers to work truly concurrently since they should not interfere with one another, but this requires careful use of two locks: one on the first element of the linked list and one on the last. It turns out that, since Java supports an atomic compare-and-set instruction, you can even write a safe implementation with no locks! In this exercise you will provide a two lock and a no lock implementation of a variant of the `MessageQueue` interface.

A two-lock version of `SafeMessageQueue` cannot use the wait-notify paradigm, since the wait-notify paradigm requires any producer or consumer to acquire a single shared lock in Java. (Recall that you must call the `wait` and `notify` methods inside a `synchronized` statement in Java otherwise the JVM will throw a `java.lang.IllegalMonitorStateException` at runtime.) As a consequence you will need to support the following non-blocking API:

```
package uk.ac.cam.your-crsid.fjava.tick3star;

public interface ConcurrentQueue<T> {
 public void offer(T message); //Add "message" to queue
 public T poll();              //Return first item from queue or null if empty
}
```

1. Start the project on Chime (https://www.cl.cam.ac.uk/teaching/current/FJava/ticklet3star) and clone the ticklet3star repository to your local machine.
2. Using `SafeMessageQueue` from Ticklet 3 as a guide, complete your implementation of `OneLockConcurrentQueue` to implement the `ConcurrentQueue` interface instead of the `MessageQueue` interface. (You should keep the methods `offer` and `poll` synchronised just as `take` and `put` are so this implementation uses only a single, shared lock.)
3. Test your implementation of `OneLockConcurrentQueue` by using `ConcurrentQueueTest` in the repository.

It turns out that fine-grained locking strategies and no-locking strategies are very hard to get right, therefore you should base your implementations on the pseudocode in the paper *Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms* by Maged M. Michael and Michael L. Scott.[1] This paper is written for languages which do not have a garbage collector (e.g. C or C++) and therefore there are a couple of points in the paper which you can ignore. Firstly, any reference to `free` in the paper can be safely ignored since the JVM will garbage collect unused memory on your behalf. Secondly, the function called CAS in the paper (an atomic compare-and-set operator) takes four arguments:

```
CAS(a,b,c,d)
```

which means set a to c iff a equals b. (The argument d is used for version counting, and is not necessary in your implementation since Java has a garbage collector.) In your Java implementation you should use the class `java.util.concurrent.atomic.AtomicReference`. You can create a new instance as follows:

[1]This URL contains the paper together with PDF comments which clarify a couple of parts of the original paper: http://www.cs.rochester.edu/u/scott/papers/1996_PODC_queues.pdf; original paper here https://dl.acm.org/citation.cfm?id=248106

```
AtomicReference ar = new AtomicReference(a);
```

You can then do the equivalent of the CAS function as follows:

```
ar.compareAndSet(b,c)
```

which updates `ar` to point to `c` iff `a` equals `b`.

4. Complete the implementation of `TwoLockConcurrentQueue` which implements the `ConcurrentQueue` interface and which supports fine-grained locking by locking on the first and last `Link` items in the queue as suggested in Figure 2 of the paper.
5. Complete the implementation of `NoLockConcurrentQueue` which implements the `ConcurrentQueue` interface and which uses no locks by making use of the `AtomicReference` class in `java.util.concurrent.atomic`. An outline of the code required is given in Figure 1 of the paper.
6. Test your implementation of `TwoLockConcurrentQueue` and `NoLockConcurrentQueue` by using `ConcurrentQueueTest`.

Submission

When you are satisfied you have completed everything, you should commit all outstanding changes and push these to the Chime server. On the Chime server, check that the latest version of your files are in the repository, and once you are happy schedule your code for testing. You can resubmit as many times as you like and there is no penalty for re-submission. If, after waiting one hour, you have not received a final response you should notify `ticks1b-admin@cl.cam.ac.uk` of the problem. You should submit a version of your code which successfully passes the automated checks by the deadline, so don't leave it to the last minute!