Exercises week 12
Thursday 14 November 2019

Goal of the exercises

The goal of this week's exercises is to make sure that you can work with lock-free data structures, measure their performance, and test them.

Deliverables

Hand in a self-contained PDF file which include your answers and all relevant code snippets. Submit your source files for documentation. We should be able to correct your assignment solely based on the PDF and only need to refer to the source code if anything is unclear.

Do this first

Get and unpack this week's example code from the course homepage.

Exercise 12.1 (**Optional:**)

This exercise is about testing the lock-free Michael-Scott queue class MSQueue presented in the lecture, and implemented in file TestMSQueue.java. That queue is unbounded, so the `enqueue` method will never block, and symmetrically the `dequeue` method is non-blocking: it just returns `null` if the queue is empty, instead of waiting for an item to arrive in the queue.

1. Write a simple sequential test for the Michael-Scott queue implementation. You may adapt the sequential test from week 8's TestBoundedQueueTest.java file.

2. Write a concurrent test for the Michael-Scott queue implementation. You may adapt the concurrent test from the same file mentioned above. In the original test it is enough for each consumer to perform `nTrials` calls to `take` because each call is guaranteed to return an item, but when testing the non-blocking queue a consumer must loop and call `dequeue` until it has obtained `nTrials` actual non-`null` items.

   Does the MSQueue implementation pass the concurrent test?

3. Inject some faults in the MSQueue implementation and see whether the test detects them. Describe the faults and whether the test detects them, and if it does detect them, how it fails.

Exercise 12.2 (**Optional:**)

In this exercise we take a closer look at the Michael-Scott queue implementation described in Michael and Scott's paper, and implemented by class MSQueue in file TestMSQueue.java.

1. The checks performed at source lines E7 and D5 look reasonable enough, but are they really useful? For instance, it seems that right after `(last == tail.get())` was successfully evaluated to true at E7, another thread could modify `tail`. Hence it seems that the check does not substantially contribute to the correctness of the data structure.

   Do you agree with this argument? Think about it, make some drawings of possible scenarios, perform some computer experiments, or anything else you can think of, and report your findings.

2. Run the sequential and concurrent tests from Exercise 12.1 on a version of the MSQueue class in which you have deleted the check at line E7 in the source code. Does it pass the test?

3. Run the sequential and concurrent tests from Exercise 12.1 on a version of the MSQueue class in which you have deleted the check at line D5 in the source code.

4. If the checks at lines E7 and D5 are indeed unnecessary for correctness, what other reasons could there be to include them in the code? How would you test your hypotheses about such reasons?

5. Describe and conduct an experiment to cast some light on the role of one of E7 and D5.

Exercise 12.3 (**Optional:**)
In this exercise you must measure the scalability of some unbounded queue implementations.

1. Measure performance of the Michael-Scott queue implementation class MSQueue that uses an AtomicReference<Node<T>> in each Node<T> object.

   You may plan for moderate contention, for instance use N threads that call `enqueue` and N that call `dequeue`, and perform some computation in the meantime, and for moderate N such as $1 \ldots 4$. For instance, the producers may produce prime numbers and the consumers check they are prime (using the `isPrime` method from several previous examples).

2. Measure performance of the Michael-Scott queue implementation class MSQueueRefl that instead uses a volatile Node<T> field in each Node<T> object, and the AtomicReferenceFieldUpdater to avoid the allocation of an AtomicReference<Node<T>> for each Node<T> object.

   How much does this change improve the performance?

3. Implement a lock-based unbounded queue, still based on the Node<T> class, but using synchronized `enqueue` and `dequeue` methods, none of which blocks. Measure the performance of this queue implementation and compare with the two Michael-Scott queue implementations.

4. Measure also the performance of a version of the Michael-Scott queue where the E7 and D5 checks have been removed, as discussed in Exercise 12.2.

Exercise 12.4 (**Optional:**)
File TestChaseLevQueue.java contains an implementation of a simplified Chase-Lev work-stealing deque.

1. Write test cases for the ChaseLevDeque<T> implementation, first a precise sequential functional tests (for instance, that if you push an item and then immediately pop, you get the same item back).

2. Write approximate concurrent tests, as discussed in course week 8. In these tests you should create a single ChaseLevDeque<T> instance and then have a single thread that performs `push` and `pop` operations on the instance being tested, and multiple other threads that perform concurrent `steal` operations on it. For example, you may push one million random Integers, and concurrently pop or steal one million random Integers, and afterwards check that the sum of the pushed numbers equals the sum of the popped or stolen numbers.

   Make the queue instance large enough so that you avoid overflowing it; that would throw an exception and ruin the test. Use a CyclicBarrier to make sure all the testing threads are ready to start at the same time, and use a CyclicBarrier to make sure all the testing threads terminate before checking the results.

   Show your test code, explain what it does, explain what parameters (how many threads, how many pushes, and so on) you run it with, and show results of running it.

3. Use mutation of the ChaseLevDeque<T> implementation to find out how good your test cases are. Explain what parts of the implementation you mutate and whether your tests discover the faults that you add to the implementation.

   Compile and run the test cases. Do the mutated implementations pass the tests?

4. As a particular mutation, try to remove the increment of `top` performed in the `pop` method. This may cause two concurrent `steal` and `pop` operations to succeed, so that an item that is put into the queue once may be be taken from the queue twice.

   Does your (concurrent) test detect this?

Exercise 12.5 (**Optional:**) File TestUnionFind.java contains three implementations of the union-find data structure, and basic test sequential and concurrent test cases for them.

1. Compile and run the test cases. Do the implementations pass the tests?

2. Now try to mutate the implementations with faults, starting with the coarse-locking CoarseUnionFind implementation. In particular, make them thread-unsafe, for instance by removing synchronization. Can you provoke the concurrent test to fail at all? This may be hard.

3. Make the concurrent test harder. For instance, you may increase the chance that two threads manipulate the same nodes at the same time. Is it possible to make the concurrent test fail on mutated implementations?