Exercises week 11
Thursday 7 November 2019

Goal of the exercises

The goal of this week's exercises is to make sure that you can use lock-less approaches to mutable shared memory, and that you can use the compare-and-swap primitive to implement simple lock-less data structures.

Do this first

Get and unpack this week's example code in code.tar.gz on the course homepage.

Exercise 11.1 Implement a CasHistogram class in the style of week 9 with this interface:

```
interface Histogram {
  void increment(int bin);
  int getCount(int bin);
  int getSpan();
  int[] getBins();
  int getAndClear(int bin);
  void transferBins(Histogram hist);
}
```

The implementation should use AtomicInteger instead of transactions or locks, and use *only* methods `get` and `compareAndSet`, not the other methods provided on AtomicInteger. This should be quite easy if you take some hints from the lecture.

1. Write a class CasHistogram so that it implements the above interface. Explain why the methods `increment`, `getBins`, `getAndClear` and `transferBins` work.

2. Use your new CasHistogram class for the parallel prime counting example; you can take most of the code from week 9's stm/TestStmHistogram.java example file. Does it produce the right results (see Exercise 10.2.2) when run on this example?

3. Measure the overall time to run the above-mentioned test, on week 9's StmHistogram implementation as well as on this week's CasHistogram implementation. Use the simple Timer class directly to measure the time from right after passing the start CyclicBarrier to right after passing the end CyclicBarrier; you do not need the Mark6 or Mark7 timing infrastructure. Report the measured running times. Which implementations is fastest? Reflect on the possible reasons.

4. (Optional) Measure the overall time to run the above-mentioned test also on week 2's lock-based Histogram2 implementation. As before, use the simple Timer class directly to measure the time from right after passing the start CyclicBarrier to right after passing the end CyclicBarrier; you do not need the Mark6 or Mark7 timing infrastructure. How does the performance of that (coarse) lock-based implementation compare to the CAS-based one in this application?

Exercise 11.2 A read-write lock, in the style of Java's java.util.concurrent.locks.ReentrantReadWriteLock, can be held either by any number of readers, or by a single writer.

In this exercise you must implement a simple read-write lock class SimpleRWTryLock that is **not** reentrant and that does **not** block. It should have the following four public methods:

```
class SimpleRWTryLock {
  public boolean readerTryLock() { ... }
  public void readerUnlock() { ... }
  public boolean writerTryLock() { ... }
  public void writerUnlock() { ... }
}
```

Method `writerTryLock` is called by a thread that tries to obtain a write lock. It must succeed and return true if the lock is not already held by any thread, and return false if the lock is held by at least one reader or by a writer.

Method `writerUnlock` is called to release the write lock, and must throw an exception if the calling thread does not hold a write lock.

Method `readerTryLock` is called by a thread that tries to obtain a read lock. It must succeed and return true if the lock is held only by readers (or nobody), and return false if the lock is held by a writer.

Method `readerUnlock` is called to release a read lock, and must throw an exception if the calling thread does not hold a read lock.

The class can be implemented using AtomicReference and compare-and-swap, by maintaining a single field `holders` which is an atomic reference of type Holders, an abstract class that has two concrete subclasses:

```
private static abstract class Holders { }

private static class ReaderList extends Holders {
  private final Thread thread;
  private final ReaderList next;
  ...
}

private static class Writer extends Holders {
  public final Thread thread;
  ...
}
```

The ReaderList class is used to represent an immutable linked list of the threads that hold read locks. The Writer class is used to represent a thread that holds the write lock. When `holders` is `null` the lock is unheld.

(Representing the holders of read locks by a linked list is very inefficient, but simple and adequate for illustration. The real Java ReentrantReadWriteLock essential has a shared atomic integer count of the number of locks held, supplemented with a ThreadLocal integer for reentrancy of each thread and for checking that only lock holders unlock anything. But this would complicate the exercise. Incidentally, the design used here allows the read locks to be reentrant, since a thread can be in the reader list multiple times, but this is inefficient too).

1. Implement the `writerTryLock` method. It must check that the lock is currently unheld and then atomically set `holders` to an appropriate Writer object.

2. Implement the `writerUnlock` method. It must check that the lock is currently held and that the holder is the calling thread, and then release the lock by setting `holders` to `null`; or else throw an exception.

3. Implement the `readerTryLock` method. This is marginally more complicated because multiple other threads may be trying (successfully) to lock at the same time, or may be unlocking read locks at the same time. Hence you need to repeatedly read the `holders` field, and so long as it is either `null` or a ReaderList, attempt to update the field with an extended reader list, containing also the current thread.

   (Although the SimpleRWTryLock is not intended to be reentrant, for the purposes of this exercise you need not prevent a thread from taking the same lock more than once).

4. Implement the `readerUnlock` method. This also requires a loop and for the same reason as above. You should repeatedly read the `holders` field and so long as it is non-`null` and refers to a ReaderList and the calling thread is on the reader list, create a new reader list where the thread has been removed, and try to atomically store that in the `holders` field; if this succeeds, it should return. If `holders` is `null` or does not refer to a ReaderList or the current thread is not on the reader list, then it must throw an exception.

   For the `readerUnlock` method it is useful to implement a couple of auxiliary methods on the immutable ReaderList:

   ```
   public boolean contains(Thread t) { ... }
   public ReaderList remove(Thread t) { ... }
   ```

5. Write simple sequential test cases that demonstrate that your read-write lock works with a single thread. For instance, it should not be able to take a read lock while holding a write lock, and vice versa, and should not be allowed to unlock a read lock or write lock that it does not already hold.

6. Write slightly more advanced test cases that use at least two threads to test basic lock functionality.

7. Improve the `readerTryLock` method so that it prevents a thread from taking the same lock more than once, instead an exception if it tries. For instance, the calling thread may use the `contains` method to check whether it is not on the readers list, and add itself to the list only if it is not. Explain why such a solution would work in this particular case, even if the test-then-set sequence is not atomic.

Exercise 11.3 This exercise concerns the scalability of five different pseudo-random number generators.

1. Run the scalability test in file TestPseudoRandom.java on your own computer and preferably also a different one for comparison. By default the scalability test runs with 1 to 32 threads; if your computer has 2 or 4 cores you may reduce the 32 threads to 16 or 8 threads. Hand in the results in a table or graphical form and reflect on them. Which random number generator is fastest in absolute terms, and which one scales best with more threads?