Exercises week 10

Last update: 2022/11/10

Goal of the exercises

The goals of this week's exercises are:

- Being able to use compare-and-swap (CAS) to solve concurrency problems.
- Being able to use `AtomicXX` Java classes.
- Design and use a lock-free data structure.

Exercise 10.1 Implement a `CasHistogram` class in the style of week 6 with this interface:

```
interface Histogram {
  void increment(int bin);
  int getCount(int bin);
  int getSpan();
  int getAndClear(int bin);
}
```

The implementation must use `AtomicInteger` (instead of locks), and ***only*** use the methods `compareAndSet` and `get`; no other methods provided in the class `AtomicInteger` are allowed.

The method `getAndClear` returns the current value in the bin and sets it to 0.

Mandatory

1. Write a class `CasHistogram` implementing the above interface. Explain why the methods `increment`, `getBins`, `getSpan` and `getAndClear` are thread-safe.
2. Write a parallel functional correctness test for `CasHistogram` to check that it correctly stores the number of primes in the range $(0, 4999999)$; as you did in exercise 6.3.3 in week 6. You must use JUnit 5 and the techniques we covered in week 4. The test must be executed with 2^n threads where $n \in \{0, \ldots, 4\}$. To assert correctness, perform the same computation sequentially using the class `Histogram1` from week 6. Your test must check that each bin of the resulting `CasHistogram` (executed in parallel) equals the result of the same bin in `Histogram1` (executed sequentially).
3. Measure the overall time to run the program above for `CasHistogram` and the lock-based Histogram week 6, concretely, `Histogram2`. For this task you should not use JUnit 5, as it is does offer good support to measure performance. Instead you can use the code in the file `TestCASLockHistogram.java`. It contains boilerplate code to evaluate the performance of counting prime factors using two Histogram classes. To execute it, simply create two objects named `histogramCAS` and `histogramLock` containing your implementation of Histogram using CAS (`CasHistogram`) and your implementation of Histogram using a single lock from week 6 (`Histogram2`).

   What implementation performs better? The (coarse) lock-based implementation or the CAS-based one?

   Is this result you got expected? Explain why.

Exercise 10.2 Recall read-write locks, in the style of Java's java.util.concurrent.locks.ReentrantReadWriteLock. As we discussed, this type of lock 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 implement the following interface:

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

For convenience, we provide the skeleton of the class in `ReadWriteCASLock.java`.

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 `compareAndSet(...)`, 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).

Mandatory

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 (successfully) trying 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, as 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. You should repeatedly read the `holders` field and, as long as i) it is non-`null` and ii) refers to a ReaderList and iii) 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 JUnit 5 correctness tests that demonstrate that your read-write lock works with a single thread. Your test should check, at least, that:

   - It is not possible to take a read lock while holding a write lock.
   - It is not possible to take a write lock while holding a read lock.
   - It is not possible to unlock a lock that you do not hold (both for read and write unlock).

   You may write other tests to increase your confidence that your lock implementation is correct.

6. Finally, write a parallel functional correctness test that checks that two writers cannot acquire the lock at the same time. You must use JUnit 5 and the techniques we covered in week 4. Note that for this exercise readers are irrelevant. Intuitively, the test should create two or more writer threads that acquire and release the lock. You should instrument the test to check whether there were 2 or more threads holding the lock at the same time. This check must be performed when all threads finished their execution. This test should be performed with enough threads so that race conditions may occur (if the lock has bugs).

Challenging

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.