Exercises week 5

Last update: 2024/09/21

Goal of the exercises

The exercises aim to give you practical experience on:

- The use of compare-and-swap (CAS) to solve concurrency problems.
- The use `AtomicXX` to implement a lock-free data structure.
- The design and use a lock-free data structure.

Exercise 5.1 A histogram is a collection of bins, each of which is an integer count. The span of the histogram is the number of bins. In the problems below a span of 30 will be sufficient; in that case the bins are numbered $0 \ldots 29$.

Consider this Histogram interface for creating histograms:

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

Intuitively, these methods work as follows: `increment(7)` will add one to bin 7; method call `getCount(7)` will return the current count in bin 7; method `getSpan()` will return the number of bins; and, finally, the method `getAndClear(7)` returns the current count in bin 7 and reset the count to 0. The class `Histogram1` in `app/src/main/java/exercises05/Histogram1.java` contains a *non*-thread-safe implementation with this behavior.

In this exercise, your task is to implement and test a lock-free Histogram class, `CasHistogram`, that has the behavior explained above also for concurrent execution of method calls. The implementation must use `AtomicInteger`, and ***only*** use the methods `compareAndSet` and `get`; no other methods provided in the class `AtomicInteger` can be used in your implementation.

Mandatory

1. Write a class `CasHistogram` implementing the above interface. In your implementation, ensure that: i) class state does not escape, and ii) safe-publication. Explain why i) and ii) are guaranteed in your implementation, and report any immutable variables.
2. Note that the behavior for the method `getAndClear(int bin)` combines two operations: *get* (obtaining the current count in the specified bin, and *clear* reset the bin count to 0. Does your implementation for this method behave as if the operations were executed atomically? Explain your answer.
3. Your task in this exercise is to write a parallel functional correctness test for `CasHistogram`. In this test, you must use your `CasHistogram` class to concurrently count the number of prime factors for the numbers in the range $(0, 4999)$. For instance, number 2 is a prime number, thus it has one prime factor and the program must increment bin 1. Similarly, number $4 = 2 \cdot 2$, i.e., it is decomposed into 2 prime factors. In this case, the program must increment bin 2. Recall that prime factors are the building blocks of natural numbers—any natural number can be decomposed into prime factors. The skeleton class `TestCasHistogram` in `app/src/test/java/exercises05` contains a static function `countFactors(int p)`, which, given an integer `p`, it returns the number of prime factors. This function might be handy for writing the test. In this parallel test, you must create one thread per number to check, and all threads must share the `CasHistogram` object under test. You must use JUnit 5 and the techniques we covered in week 4. To assert correctness, perform the same computation sequentially using

the class `Histogram1`. 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).

Exercise 5.2 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:

```
interface 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 demonstrates 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.