Exercises week 3

Last update 2020-08-29

Goal of the exercises

The goals of these exercises supports the topics of chapter 4 and 5 in *Goetz*. That is, to give you practical experience in designing *efficient* and *thread-safe* objects.

Exercise 3.1 A *Bounded Buffer* is a data structure that can only hold a fixed set of elements. In this exercise you must design such a data structure which allow elements to be inserted into the buffer by one thread, and taken by an other thread.

If a thread tries to insert an element into a full buffer, the thread should be blocked until an element is taken by an other thread. Similarly, if a thread tries to take an element from an empty buffer, it should be blocked until an element is inserted.

The `java.util.concurrent.BlockingQueue` class implements this, but in this exercise we build a simple version our self. Section 5.3 in *Goetz* talks a bit about how to use `BlockingQueue`.

```
interface BoundedBuffer<T> {
  void insert(T elem);
  T take();
}
```

In the buffer you must design, the policy for inserting and taking should be a FIFI Queue.

Green

1. You should not build the internal data structure yourself, but use something already in the java libraries (Though you are not allowed to use the `BlockingQueue`).

   Will you choose a thread safe or regualar collection to implement the buffer? And which one? And why?

2. Section 5.5 in *Goetz* talks about different kinds of *synchronizers*. Which of the mentioned kinds seems to be most suited for this problem (Latches, FutureTask, Semaphores or Barriers)? And why?

Yellow

3. Implement the two methods and the constructor. The constructor should take an argument with the number of items the buffer can hold. The class must be thread-safe.

Red

4. One of the two constructors to Semaphore has an extra parameter named `fair`. Explain what it does, and explain if it matters in this example. If it does not matter in this example, find an example where it does matter.

5. Assume we want to monitor if the buffer is balanced over time. That is, normally there should be room to insert elements, as well as elements to take. How can you extend the buffer with bookkeeping so one can ask the buffer for the number of times it has been full and the number of times it has been empty. *Hint: there are methods in the Semaphore for something like this.*

Exercise 3.2 Consider the potentially computation-intensive problem of counting the number of prime number factors of an integer[1]. This Java method from file TestCountFactors.java finds the number of prime factors of `p`:

[1]The exercises use a lot of primenumbers. It is mostly because they take time to compute and does not depend on libraries or servers. If you want to, you could have substituted a number of other time consuming algorithms - e.g. image recognition, language translation or database access,

```
public static int countFactors(int p) {
  if (p < 2)
    return 0;
  int factorCount = 1, k = 2;
  while (p >= k * k) {
    if (p % k == 0) {
      factorCount++;
      p /= k;
    } else
      k++;
  }
  return factorCount;
}
```

How this method works is not important, only that it may take some time to compute the number of prime factors. Actually the time is bounded by a function proportional to the square root of `p`, in other words $O(\sqrt{\mathtt{p}})$.

Green

1. The file TestCountFactors.java also contains a sequential program to compute the total number of prime factors of the integers in range 0 to 4,999,999. The result should be 18,703,729. How much time does this take? *Hint:* you can use System.nanoTime() to measure the time it takes.
2. For use in the next subquestion you will need a MyAtomicInteger class that represents a thread-safe integer. It must have a method `int addAndGet(int amount)` that atomically adds `amount` to the integer and returns its new value, and a `int get()` method that returns the current value.

   Write such a MyAtomicInteger class, which should not be based on one of the existing java.util.concurrent.atomic classes.
3. Write a parallel program that uses 10 threads to count the total number of prime factors of the integers in range 0 to 4,999,999. Divide the work so that the first thread processes the numbers 0–499,999, the second thread processes the numbers 500,000–999,999, the third thread processes the numbers 1,000,000–1,499,999, and so on, using your MyAtomicInteger class. Do you still get the correct answer? How much time does this take?
4. Could one implement MyAtomicInteger without synchronization, just using a volatile field? Why or why not?

Yellow

5. Solve the same problem (sub-question 3), but use the AtomicInteger class from the java.util.concurrent.atomic package instead of MyAtomicInteger. Is there any noticeable difference in speed or result? Should the AtomicInteger field be declared `final`?

Exercise 3.3 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...29.

Consider this Histogram interface for creating histograms:

```
interface Histogram {
  public void increment(int bin);
  public int getCount(int bin);
  public float getPercentage(int bin);
  public int getSpan();
  public int getTotal();
}
```

Method call `increment(7)` will add one to bin 7; method call `getCount(7)` will return the current count in bin 7;method call `getPercentage(7)` will return the current percentage of total in bin 7; method `getSpan()` will return the number of bins; method call `getTtal()` will return the current total of all bins.

There is a non-threadsafe implementation Histogram1 in file SimpleHistogram.java. You may assume that the `dump` method given there is called only when no other thread manipulates the histogram and therefore does not require locking, and that the span is fixed (immutable) for any given Histogram object.

Green

1. Make a thread-safe implementation, class Histogram2, of interface Histogram by adding suitable modifiers (`final` and `synchronized`) to a copy of the Histogram1 class. Which fields and methods need which modifiers? Why? Does the `getSpan` method need to be synchronized?

2. Now consider again counting the number of prime factors in a number `p`, as in Exercise 3.2.3 and file TestCountFactors.java. Use the Histogram2 class to write a parallel program that counts how many numbers in the range 0. . . 4 999 999 have 0 prime factors, how many have 1 prime factor, how many have 2 prime factors, and so on. You may draw inspiration from the TestCountPrimes.java example.

   The correct result should look like this:

```
   0:         2
   1:    348513
   2:    979274
   3:   1232881
   4:   1015979
   5:    660254
   6:    374791
   7:    197039
   8:     98949
   9:     48400
... and so on
```

   showing that 348 513 numbers in 0. . . 4 999 999 have 1 prime factor (those are the prime numbers), 979 274 numbers have 2 prime factors, and so on. (The 2 numbers that have 0 prime factors are 0 and 1). And of course the numbers in the second column should add up to 5 000 000.

   .

Yellow

3. Define a thread-safe class Histogram3 that uses an array of java.util.concurrent.atomic.AtomicInteger objects instead of an array of integers to hold the counts.

   In principle this solution might perform better, because there is no need to lock the entire histogram object when two threads update distinct bins. Only when two threads call `increment(7)` at the same time do they need to make sure the increments of bin 7 are atomic.

   Can you now remove `synchronized` from all methods? Why? Run your prime factor counter and check that the results are correct.

   .

4. Define a thread-safe class Histogram4 that uses a java.util.concurrent.atomic.AtomicIntegerArray object to hold the counts. Run your prime factor counter and check that the results are correct.

   .

5. Now extend the Histogram interface with a method `getBins` that returns an array of the bin counts:

```
   public int[] getBins();
```

Show how you would implement this method for one of the classes Histogram2, Histogram3 and Histogram4 so that it remain thread-safe. Explain whether it gives a fixed snapshot or a live view of the bin counts, possibly affected by subsequent `increment` calls.

Red

6. In Java 8 there is class java.util.concurrent.atomic.LongAdder that potentially offers even better scalability across multiple threads than AtomicInteger and AtomicLong; see the Java class library documentation. Create a Histogram5 class that uses an array of LongAdder objects for the bins, and use it to solve the same problem as before. Explain the difference between the LongAdder and AtomicLong.

Exercise 3.4 *Red*

Consider the class `TestStaticCounter` provided as source code. It uses one `Long` as the counter, synchronizing on it. Its intended behaviour is that it creates two threads, each incrementing count 20 million times in a synchronized way. Accordingly, the expected output would be that whichever thread finishes last, it produces the same number as the main thread computes.

1. Describe what you observe when you run the program.
2. How can you explain what you observe?
3. Create a version of the program (changing as little as possible) that works as intended.