Exercises week 3
Friday 12 September 2014

Goal of the exercises

The goal of this week's exercises is to make sure that can build a threadsafe class in Java, make effective use of Java's concurrent collection classes in package `java.util.concurrent`, and use the future concept.

Do this first

Get and unpack this week's example code in zip file pcpp-week03.zip on the course homepage.

Exercise 3.1 A histogram is a collection of buckets, each of which is an integer count. The span of the histogram is the number of buckets. In the problems below a span of 30 will be sufficient; in that case the buckets are numbered 0...29.

Consider this Histogram interface for creating histograms:

```
interface Histogram {
  public void increment(int bucket);
  public int getCount(int bucket);
  public int getSpan();
}
```

Method call `increment(7)` will add one to bucket 7; method call `getCount(7)` will return the current count in bucket 7; method `getSpan()` will return the number of buckets.

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.

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 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.

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 buckets. Only when two threads call `increment(7)` at the same time do they need to make sure the increments of bucket 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 `getBuckets` that returns an array of the bucket counts:

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

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

   Note in particular that for instance in the case of Histogram2 it would not be thread-safe to just return a reference to the internal array of integers, since a client who receives that reference could mess with the histogram's bucket counts without any synchronization.

6. (Optional). 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 buckets, and use it to solve the same problem as before.

Exercise 3.2 File TestCache.java contains a version of the prime factorization server example that implements the Computable interface and therefore can be wrapped in a memoizer, as developed in the lecture.

In this exercise you must write a program that creates and starts 16 threads numbered $t = 0 \ldots 15$, each of which computes the factors of 40 000 numbers, and such that their work partially overlaps (to demonstrate that the cache works):

- Every thread t must compute the factors of the 20 000 numbers from 10 000 000 000 to 10 000 019 999.
- Thread t must further compute the factors of the 20 000 numbers from $10\ 000\ 020\ 000 + t \cdot 5\ 000$ to $10\ 000\ 039\ 999 + t \cdot 5\ 000$.

In total the numbers in the range from $10\ 000\ 000\ 000$ to $10\ 000\ 039\ 999 + 15 \cdot 5\ 000 = 10\ 000\ 114\ 999$ will be factorized, that is, $115\ 000$ distinct numbers.

With a view to next week's (mandatory) exercises it is advisable to implement this scheme in terms of two parameters `start` and `range`:

```
final long start = 10_000_000_000L, range = 20_000L;
```

Then thread t considers the two ranges `from1...to1` and `from2...to2`, startpoint included and endpoint excluded, where `from1 = start`, `to1 = from1+range`, `from2 = start+range+t*range/4`, and `to2 = from2+range`.

1. Write a method `exerciseFactorizer` that takes as argument a thread-safe caching factorizer and calls it from 16 threads as specified above. The method outline may be something like this:

   ```
   private static void exerciseFactorizer(Computable<Long, long[]> f) {
     final int threadCount = 16;
     final long start = 10_000_000_000L, range = 20_000L;
     System.out.println(f.getClass());
     ...
   }
   ```

where the purpose of printing `f.getClass()` is just to show which of the cache classes is currently being used.

2. Wrap the given Factorizer in the Memoizer1 class and run the above program on this cached factorizer, then print the number of calls to the underlying Factorizer. You might use code such as this:

   ```
   Factorizer f = new Factorizer();
   exerciseFactorizer(new Memoizer0<Long,long[]>(f));
   System.out.println(f.getCount());
   ```

   The number of calls to the factorizer should be 115 000. Is it?

   If your platform allows it, measure and note the execution time for this activity, using eg. `time java TestCache` on MacOS or Linux. In that case, note both the "real time" which is the wall-clock time, the "user time" which is the total CPU time spent by your code, and the "system time" which is the total CPU time spent in the operating system kernel.

3. Repeat this experiment with Memoizer2. How many times is the factorizer called? How long does the whole process take? Explain both results.

4. Repeat this experiment with Memoizer3. How many times is the factorizer called? How long does it take? Explain both results.

5. Repeat this experiment with Memoizer4. How many times is the factorizer called? How long does it take? Explain both results.

6. (Optional, requires Java 8) Repeat this experiment with Memoizer5. How many times is the factorizer called? How long does it take? Explain both results.

7. (Optional, requires Java 8) Write a caching class Memoizer0 that uses ConcurrentHashMap and its `computeIfAbsent` method to simply compute the given work `c.compute(arg)`, and using no FutureTasks or other fancy features. This can be done in 10 lines of code or less, and the correctness and thread-safety should be obvious. Repeat the above experiment with your Memoizer0. How many times is the factorizer called? How long does it take? Explain both results.

   Note: This is vastly simpler than Goetz's development, yet performs quite well in the present application even though it may violate the advice given in the Java 8 class library documentation for `computeIfAbsent`: "Some attempted update operations on this map by other threads may be blocked while computation is in progress, so the computation should be short and simple". Or maybe prime factorization *is* a "short and simple" computation, whereas for instance an HTTP request to a webserver would not be — such a request might block for many seconds. Hence in general the fancy Memoizer5 cache is probably still much preferable to the simpler Memoizer0.