PCPP **IT University, E2020**

Exercises lesson 12

Last update 2020-11-10

Goal of the exercises

The goal of this week's exercises is to make sure that you understand the challenges of testing concurrent software, that you can nevertheless write a reasonable test suite for such software using recommended techniques, and use mutation to judge the quality of the test suite.

Exercise 12.1 This exercise is based on the exercise 5.1 from lesson 5 where we had a thread un-safe implementation of transferring money from one account to an other. This exercise is similar. It is based on the classes in file PicoBankTest.java. The file contains two classes, PicoBankTest which is used to run Tests of the other class named PicoBankBasic. To make it easier to experiment with different implementations there is a small interface PicoBank.

.

Green

1. Your first task is to write a test program that shows that the PicoBank is **not** thread safe. Your solution should have sufficient number of threads and sufficient number of transfers to show that the race conditions happens or visibility of updates are broken, or both.

   You must also argue that your method of determining success or failure of a test is reasonable.

   It is recommended to use the CyclicBarrier class to start the threads, and restart the main method as done in the book (listing 12.5 and listing 12.6).

2. Next, you should make an other implementation of PicoBank where one or more methods are synchronized. Rerun the test. This time you should be able to use the test program from the previous exercise to show that your new implementation of PicoBank is free of race conditions.

3. As can be seen in the method for testing with a single thread, a time measurement is printed. Add timing to your multithreaded test as well, and print out the results so they can be compared with the single threaded execution. Make sure the single threaded and the multi threaded tests make the same number of transfers.

4. Just like in the original exercise 5.1.2 from lesson 5, one can argue that it can be allowed to move an amount from account1 to account7 at the same time as moving money from account6 to account3. In exercise 5.1.2 you were asked to use locking, and in particular the locking order principles from section 10.1.2 and listing 10.3. As we discussed at that time, there is no need to use hashCode as done in listing 10.3, because all accounts have an id.

   Implement the PicoBank using lock-order. Use the same test program to verify that this implementation also produces the right results.

5. Some students came up with the idea that one do not need any locks at all. If one use an AtomicLong for the balance in the Account class. Implement that solution. What does your test say about that solution?

   If your test accept this solution, is it because the test is broken, or is this truely am acceptable solution to prevention of race conditions and visibility?

Yellow

6. All the mutating operations in AtomicLong is fundamentally build using get() and compareAndSet(). As in the previous subquestion, let the balances be represented by AtomicLong, but implement the transfer using only get() and compareAndSet(). Verify that your solution works using the test program from earlier.

Exercise 12.2 In this exercise you must conduct a functional test of the StripedWriteMap<K,V> implementation of a concurrent hash map presented in lesson 7's lecture, and completed by you in the exercises. Since it was designed for this course, and the implementation was completed by you, nobody knows whether it is correct. We need to test it.

Green

1. First, consider a functional test of the hash map's sequential correctness as attempted in method `testMap` in the file. Does the implementation pass this simple test? Describe any inadequacies in the test, such as lack of method coverage or statement coverage in the hash map implementation. Extend the test to address the deficiencies. Does the implementation still pass the sequential test?

Yellow

2. Now turn to testing of the hash map's functional correctness in a concurrent context, where multiple threads read and modify the hash map at the same time.

   You may draw inspiration from Goetz at al. section 12.1 which shows how to test a blocking queue:

   - Create a single StripedWriteMap<Integer,String> concurrent hash map instance to test, with Integers as keys and Strings as values. To increase the chance that multiple threads will manipulate the same bucket, and the same stripe, at the same time, you should create the map with few stripes, maybe 7, and with few buckets, maybe 77 — remember that the number of buckets must be a multiple of the number of stripes. Also, you should run with a rather small range of random keys to insert into the table, maybe $0 \ldots 99$, to increase the chance of the same key being added or removed at the same time.
   - Create multiple testing threads to manipulate the concurrent hash map. There should be more threads than cores, but not unreasonably many, so 16 testing threads would be a good choice on most current hardware.
   - Each testing thread performs `containsKey`, `put`, `putIfAbsent` and `remove` on the concurrent hash map, on randomly chosen keys.
   - Each testing thread should have its own random number generator. Using a shared random number generator might affect the thread scheduling and hence impair the thread interleaving coverage of the test.
   - Each testing thread maintains the sum of all new keys it puts into the hash map, minus the keys it removes. Note that neither `put(k,v)` nor `putIfAbsent(k,v)` adds a new key if `k` is already present.
   - After all testing threads have completed, the sum of the keys in the hash map should equal the sum of the sums from the testing threads.
   - Use a CyclicBarrier from package java.util.concurrent to make sure that the testing threads run only when all of them are ready; this minimizes the risk that they will run sequentially.

   Implement such a functional test. Does it find defects in the hash map implementation? To what degree does the test convince you that the StripedWriteMap implementation is correct? In particular, does it tell you anything about the correctness of `containsKey`?

3. Run the functional test also on a WrapConcurrentHashMap<Integer,String> instance; here it hopefully finds no defects. In general, if your test finds a defect in the StripedWriteMap implementation, run the test also on WrapConcurrentHashMap to see whether the deficiency is in the StripedWriteMap or in your test (or both).

Red

4. The functional test as proposed above checks only that the hash map contains the expected keys. To check also that the associated values are correct (or at least plausible) you may number the testing threads `t` $= 0 \ldots (N-1)$ and let thread `t` insert a String value of form `"t:k"` for key `k`.

   Then check, when all testing threads have completed, that every entry in the hash map has the form `(k, "t:k")` for some thread number `t`.

5. You can further let each testing thread keep, in an array `int[] counts = new int[`N`]`, a net count of the number of entries "belonging" to thread `t`. That is, if thread `t` adds a new entry `(k,"t:k")` then it increments `counts[t]`. Similarly, if it removes an entry `(k,"u:k")` made by thread `u`, then it decrements `counts[u]`. Note that the latter may happen both as a consequence of `remove(k)` and as a consequence of `put(k,"t:k")`.

   After all testing threads have completed, you should compute the sum of the N threads' `counts` arrays and traverse the hash map to check that each thread `t` has precisely as many values in the table as the sum indicates. For instance, if the sum of the N threads' `count[7]` fields is 426, then there should be 426 entries in the table of the form `(k, "7:k")`.

6. Suggest further ways to improve the test of the concurrent hash map implementations.

Why don't we just compare the results of operations on StripedWriteMap<K,V> with the results of doing the same operations on WrapConcurrentHashMap<K,V>, which presumably is a good reference implementation? The reason is two-fold: (1) While we may control the generation of pseudo-random numbers, we do not control the thread scheduler and hence the interleaving of the threads' method calls, so we can never expect to make two identical test runs, one on StripedWriteMap<K,V> and another on WrapConcurrentHashMap<K,V>. (2) Then we could manipulate both implementations in the same test run, thus exposing them to the exact same sequence of operations. But that would cause any synchronization internally in the reference implementation to interfere with the test thread scheduling, which would make the test much less effective. Probably this is not a big concern in the case of WrapConcurrentHashMap<K,V> which does little internal locking, but often the "reference implementation" will be a fully locking, basically sequential, implementation and that would completely invalidate the test of a new more concurrency-friendly and scalable implementation.

Exercise 12.3 *Red*

If your functional test in Exercise 12.2 finds no defects in the hash map implementation, you may investigate how good the test is by *mutation testing*, by *injecting faults* in the hash map implementation and running the functional test again. For instance, you may:

1. Remove `synchronized` around one or more blocks of code to see whether the functional test "discovers" the lack of synchronization.
2. Change a single occurrence of `synchronized (locks[stripe])` so that it locks on the wrong object, for instance by replacing it with `synchronized (locks[0])` or `synchronized (this)`, to see whether the functional test "discovers" the improper synchronization.
3. Change the representation of the `sizes` array from AtomicIntegerArray to plain `int[]` and the `get` and `getAndAdd` method calls to plain array reads `sized[stripe]` and increments `sized[stripe]++`, to see whether the functional test "discovers" that the sizes are not correctly updated.
4. Remove some of the reads from `sizes[stripe]` to see whether the absence of these atomic reads affects visibility of writes to reads.

   It is probably unlikely that the functional test will discover this particular fault, although it undermines the visibility of writes to subsequent reads. Also, it is not obvious how to devise a test that would reliably reveal this lack of visibility. Maybe it would help to run on an ARM (Raspberry Pi 2B or smartphone) processor, which has a weaker memory model than Intel CPUs.

5. What other ways might there be of injecting faults so as to investigate how good the functional test is? Discuss, and if possible, suggest and try out other faults that may be injected.