# Practical Concurrent and Parallel Programming – Exam 2016

Written by Rasmus Løbner Christensen ([rloc@itu.dk](mailto:rloc@itu.dk))

# Part 1

## Question 1 (5%):

Running the class *TestLocking0.java* 10 times produces the following output:

Sum is 1564637,000000 and should be 2000000,000000

Sum is 1530436,000000 and should be 2000000,000000

Sum is 1623439,000000 and should be 2000000,000000

Sum is 1524630,000000 and should be 2000000,000000

Sum is 1581357,000000 and should be 2000000,000000

Sum is 1599967,000000 and should be 2000000,000000

Sum is 1555741,000000 and should be 2000000,000000

Sum is 1604552,000000 and should be 2000000,000000

Sum is 1640719,000000 and should be 2000000,000000

Sum is 1566290,000000 and should be 2000000,000000

The question now is whether the class is thread-safe or not. First of all, using ***static synchronized*** locks the whole class, while ***synchronized*** (non-static) only locks the method. Since we are discussing whether the \_class\_ is thread safe, the ***synchronized***(non-static) method seems to be lackluster. Surprisingly enough, adding the *static* keyword to the ***addInstance*** method, produces the expected result:

Sum is 2000000,000000 and should be 2000000,000000

The above is my answer to the entire three questions in Question 1.

## Question 2 (5%):

The simplest natural (depends on the interpretation I guess) way would be making the class variables volatile, combined with all the methods being ***synchronized***. While this seems like a feasible solution, the problem is that declaring a volatile array (in this case a double array) does not give volatile access to the fields of the array. Therefore the solution would be to make the double array ***atomic***, but java only support ***AtomicIntegerArray*** which obviously would not solve the problem (we are working with doubles not integers). In order to solve this visibility issue one could re-set the array reference to be itself *every* time we set an element in the array, but this would mean that all writes to the array would involve two writes instead of one.

Moving on the next question about scalability. Beside the above mentioned obnoxious double write solution, using ***synchronized***means that all the threads are using locks, therefore waiting for each other. In lecture 6 (in this course), we saw a slide about the scalability speedup of different solutions, where it was showed that when reaching 16 threads and using ***synchronized***as thread safety – the result would actually give almost no speedup. As already mentioned the whole ***synchronized***approach would only be beneficial while using a limited number of threads.

Question 3 proposes a thread-safe scalable solution. The problem here is that while concurrent ***add*** or ***set*** calls might be safe, the situation with concurrent calls to ***add*** \_and\_ ***set*** might not be, since these could happen at the same time. In order to fix this, a version like the above mentioned ***synchronized*** one would fix this issue (but obviously would not be so scalable).

## Question 3 (5%):

In order to maintain a correct value/count of ***totalSize*** one could simply ***synchronize*** the ***add*** method; since all updates to ***totalSize*** happens through ***add***. To maintain visibility across threads, the ***totalSize*** field also has to be ***volatile***.

Further in order to maintain the correctness of ***allLists***, a solution would be to make the ***allList.add(this)*** call in the constructer synchronized (simply wrapping it in a ***synchronized(this) {…}***  statement). Furthermore making the ***allLists*** field ***volatile***.

# Part 2