Exercises week 6
Mandatory handin 3
Friday 3 October 2014

Goal of the exercises

The goal of this week's exercises is to make sure that you can write deadlock-free synchronization code, diagnose deadlocks using the `jvisualvm` tool, and check `@GuardedBy` annotations with the ThreadSafe tool. Note that Exercise 6.3 is optional and need not be answered.

Do this first

Get and unpack this week's example code in zip file pcpp-week06.zip on the course homepage. Also download the ThreadSafe tool, either

- Eclipse plugin, from http://download.contemplateltd.com/threadsafe/threadsafe-eclipse-1.3.3.zip, or
- Command line interface, from http://download.contemplateltd.com/threadsafe/threadsafe-cli-1.3.3.zip

Unpack and install as indicated in the guide at http://www.contemplateltd.com/threadsafe-solo-quick-start.

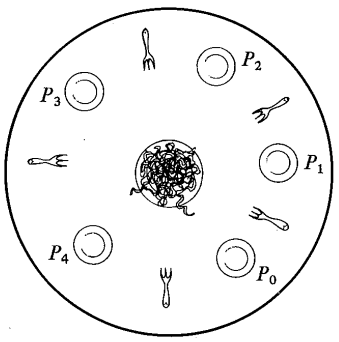
ThreadSafe is commercial software and you must replace the file `threadsafe.properties` with the one containing PCPP's license key; you find that in LearnIT under week 6. Do not share the license key with people outside the IT University.

Exercise 6.1 In this exercise you must experiment with and modify run the lecture's accounts transfer example.

1. Run TestAccountDeadlock.java on your computer. Does it deadlock? If not, how could that be?
2. Modify TestAccountLockOrder.java to use the `transferE` and `balanceSumE` methods, both of which use hashcodes to determine locking order. As said in the lecture and the code comments, this may still deadlock in the rare case two distinct Accounts get the same hashcode. Run it a couple of times on your computer. Does it actually deadlock? (Probably not).
3. Now make `transferE` and `balanceSumE` guaranteed deadlock-free by implementing the Goetz idea (section 10.1.2, code on page 209) that deals with identical hashcodes by taking a third lock that is used only for this purpose. Compile and run it. Does it still work and not deadlock?
4. Would it be safe and deadlock-free to just ignore the hashcodes and *always* use the last `else`-branch in the Goetz page 209 code, taking all three locks whenever a transfer is made? Discuss. What is the reason for not just doing that?

Exercise 6.2 In this exercise you should use the `jvisualvm` tool (distributed with the Java Software Development Kit) to investigate the famous Dining Philosopher's problem, due to E. W. Dijkstra.

Five philosophers (threads) sit at a round table on which there are five forks (shared resources), placed between the philosophers. A philosopher alternatingly thinks and eats spaghetti. To eat, the philosopher needs exclusive use of the two forks placed to his left and right, so he tries to lock them.
Both the places and the forks are numbered 0 to 5. The fork to the left of place `p` has number `p`, and the fork to the right has number `(p+1)%5`.
(Drawing from Ben-Ari: *Principles of Concurrent Programming*, 1982).



1. Consider the Dining Philosophers program in file TestPhilosophers.java. Explain why it may deadlock.

2. Compile the program, and run it until it deadlocks. Do this a few times. Does the time to deadlock vary much?

3. Again, run the program till it deadlocks and leave it there. Start `jvisualvm`, attach it to the TestPhilosophers Java process, and find what it says about the reason for the deadlock. Copy the relevant message to your answer and explain in your own words what it says.

4. Rewrite the philosopher program to avoid deadlock. The solution (as in the lecture) is to impose an ordering on the locks (forks) and then every philosopher (thread) should take the locks in that order. For instance, when a philosopher needs to take locks numbered `i` and `j`, always take the lowest-numbered one first. Implement this small change, and run the program for as long as you care. It should not deadlock.

5. Rewrite the philosopher program to use ReentrantLock on the five forks, and so that every philosopher first attempts to pick up the left fork, then the right one, leaving both on the table if any one of them is in use. You can simply make class Fork a subclass of java.util.concurrent.locks.ReentrantLock and call `tryLock()` on the Fork, as in the lecture's TestAccountTryLock.java example. Try to run the program. Does *any* philosopher get to eat at all?

6. Now is there any *fairness*, that is, at every point does a philosopher who is trying to eat eventually get to do it (also expressed as, does every philosopher get to eat infinitely often)? Use an array of threadsafe counters, for instance AtomicIntegers, to count how many times each philosopher has eaten, and make a for-loop on the "main" thread that prints these numbers every 10,000 milliseconds. What do you observe?

   It is quite possible that the philosophers (threads) get to eat roughly equally often, but this is by now means guaranteed, and the correct functioning of a program should not depend on the thread scheduler's fairness. It may vary between Java versions and operating systems, and be different on Sunday than Monday.

Exercise 6.3 (**Optional**) In this exercise you must apply the ThreadSafe tool to week 3's FirstBadListHelper and SecondBadListHelper classes in file TestListHelper.java.

1. First run ThreadSafe on the example in file ts/guardedby/TestGuardedBy.java.

2. Run ThreadSafe on week 3's FirstBadListHelper and SecondBadListHelper classes. Which of the thread-safety problems does ThreadSafe discover, and which ones does it overlook? Show the messages from ThreadSafe, explain them in your own words, and say whether you agree with them.

3. Now, add a `@GuardedBy("list")` annotation SecondBadListHelper's list field. Does this help? What does ThreadSafe say, and do you agree with the message? Explain.

Exercise 6.4 In this exercise you must apply the ThreadSafe tool to your threadsafe `int[]`-based Histogram2 class from week 3.

1. Add relevant `@GuardedBy` annotations to your threadsafe Histogram2 class from week 3. Compile it. Then use ThreadSafe to check it. Does it pass? Now delete synchronized from one of the public `increment` and `getCount` methods. What does ThreadSafe say? Does ThreadSafe expect the `getSpan()` method to be synchronized?

2. Add a new method `void addAll(Histogram hist)` to the Histogram interface and Histogram2 class. The new method should throw a RuntimeException if this histogram and `hist` have different spans. Otherwise it should add the counts of `hist` to this histogram. What does the method need to lock on? Explain. (Hint: There should be no risk of deadlocks in this question). Compile and run ThreadSafe on the code. Does it agree with you?

3. Now pretend that you are Mort Madcap, who cannot be bothered with interfaces, encapsulation, and other object-oriented dogmas. So despite what the exercise says, he has implemented `addAll` so that (1) it takes an argument of type Histogram2, not Histogram, and (2) it accesses the `hist.counts` array directly instead of using the `getCount` method. Does ThreadSafe help Mort spot any errors he may have made? Explain in your own words what ThreadSafe tries to say, and whether there is any reason to worry.