PCPP **IT University, E2015**

Exercises week 9
Friday 30 October 2015

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 use locks and synchronization to ensure mutual exclusion and thread safety in Java code.

Do this first

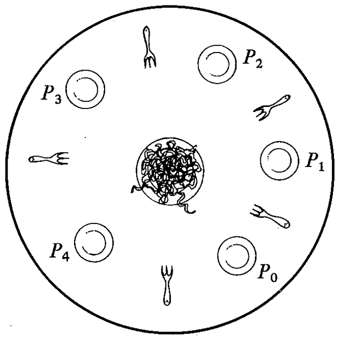
Get and unpack this week's example code in zip file pcpp-week09.zip on the course homepage.

Exercise 9.1 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 9.2 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 9.3 Consider the class in file TestHashedList.java. Class HashedList<T> implements a combined array list and a hash set: It maintains the items in insertion order and allows constant time $O(1)$ indexing into the list, and also allows constant time $O(1)$ test for whether an item is in the list (in contrast to Java's ArrayList).

1. Explain how you would make the class threadsafe, so that methods `contains`, `get`, `add`, `remove` and `set` could be called from any number of concurrent threads. You may ignore `forEach` until further.
2. Explain why it would not be sufficient to wrap each of the fields `itemList` and `itemSet` as a synchronized collection, using methods `synchronizedList` and `synchronizedSet` from class java.util.Collections. Describe an execution scenario that would demonstrate lack of threadsafety.
3. Explain why the `forEach` method is more tricky than for the other methods. Hint: Consider what happens if one implements a method `addAllTo` that uses `forEach` to copy all items to another HashedList, and two threads call `h1.addAllTo(h2)` and `h2.addAllTo(h1)` at the same time.

```
public void addAllTo(HashedList<T> other) {
  forEach(other::add);
}
```

Exercise 9.4 Consider the small artificial program in file TestLocking0.java. In class Mystery, the single mutable field `sum` is private, and all methods are synchronized, so superficially the class seems to be threadsafe.

1. Compile the program and run it several times. Show the results you get. Do they indicate that class Mystery is thread-safe or not?
2. Explain why class Mystery is not thread-safe. Hint: Consider (a) what it means for an instance method to be synchronized, and (b) what it means for a static method to be synchronized. You may consult the *Java Language Specification* (link on the course homepage) or *Java Precisely*.
3. Explain how you could make the class threadsafe. Do it, and rerun the program to see whether it works.

Exercise 9.5 Consider class DoubleArrayList in TestLocking1.java. It implements an array list of numbers, and like Java's ArrayList it dynamically resizes the underlying array when it has become full.

1. Explain the simplest natural way to make class DoubleArrayList threadsafe so it can be used from multiple concurrent threads.

2. Discuss how well the threadsafe version of the class is likely scale if a large number of threads call `get`, `add` and `set` concurrently.

3. Now your renowned colleague Ulrik Funder suggests to improve scalability by introducing a separate lock for each method, roughly as follows:

```
private final Object sizeLock = new Object(), getLock = new Object(),
  addLock = new Object(), setLock = new Object(), toStringLock = ...;
public boolean add(double x) {
  synchronized (addLock) {
    if (size == items.length) {
      ...
    }
    items[size] = x;
    size++;
    return true;
  }
}
public double set(int i, double x) {
  synchronized (setLock) {
    if (0 <= i && i < size) {
      double old = items[i];
      items[i] = x;
      return old;
    } else
      throw new IndexOutOfBoundsException(String.valueOf(i));
  }
}
```

   Does this achieve threadsafety? Explain why not. Does it achieve visibility? Explain why not.

4. Is there some version of Ulrik's proposed locking scheme that actually achieves threadsafety?

Exercise 9.6 Consider the extended class DoubleArrayList in TestLocking2.java. Like the class in the previous exercise it implements an array list of numbers, but now also has a static field `totalSize` that maintains a count of all the items every added to any DoubleArrayList instance.

It also has a static field `allLists` that contains a hashset of all the DoubleArrayList instances created. There are corresponding changes in the `add` method and the constructor.

1. Explain how one can make the class threadsafe enough so that the `totalSize` field is maintained correctly even if multiple concurrent threads work on multiple DoubleArrayList instances at the same time. You may ignore the `allLists` field for now.

2. Explain how one can make the class threadsafe enough so that the `allLists` field is maintained correctly even if multiple concurrent threads create new DoubleArrayList instances at the same time.