
Lecture: Concurrency Theory and Practise

<http://proglang.informatik.uni-freiburg.de/teaching/concurrency/2014ws/>

Exercise Sheet 4

2014-12-12

I. Santa Claus and his team

Santa Claus sleeps in his shop up at the North Pole, and can only be wakened by either all nine reindeer, Dasher and Dancer, Prancer and Vixen, Comet and Cupid, and Rudolf and Blitzen, being back from their year long vacation on the beaches of some tropical island in the South Pacific, or by some elves who are having some difficulties making the toys.

One elf's problem is never serious enough to wake up Santa (otherwise, he may never get any sleep), therefore, the elves visit Santa in a group of three. When three elves are having their problems solved, any other elf's wishing to visit Santa must wait for those elves to return. If Santa wakes up to find three elves waiting at his shop's door, along with the last reindeer having come back from the tropics, Santa has decided that the elves can wait until after Christmas, because it is more important to get his sleigh ready as soon as possible. (It is assumed that the reindeer don't want to leave the tropics, and therefore they stay there until the last possible moment. They might not even come back, but since Santa is footing the bill for their year in paradise This could also explain the quickness in their delivering of presents, since the reindeer can't wait to get back to where it is warm.) The penalty for the last reindeer to arrive is that it must get Santa while the others wait in a warming hut before being harnessed to the sleigh.

Define an algorithm for Santa and his team using appropriate synchronization and communication primitives (like barriers, see II.2, or conditions, see III.3). Each participant is modeled with his own thread, i.e. there is a Santa thread, several elf threads and the reindeer threads.

1. What is the name of the ninth reindeer?
2. Design and implement the algorithm while waiting for your Christmas presents! You better make sure that Santa is not dead-locked....

Each submitted solution will earn some cookie!



II. Theory

II.1. A Bad CLHLock

Explain how the following implementation of CLHLock can go wrong:

```
1 public class BadCLHLock implements Lock {
2     // most recent lock holder
3     AtomicReference<Qnode> tail;
4     //thread-local variable
5     ThreadLocal<Qnode> myNode;
6
7     public void lock() {
8         Qnode qnode = myNode.get();
9         qnode.locked = true; // I am not done
10        // Make me the new tail, and find my predecessor
11        Qnode pred = tail.getAndSet(qnode);
12        // spin while predecessor holds lock
13        while (pred.locked) {}
14    }
15
16    public void unlock() {
17        // reuse my node next time
18        myNode.get().locked = false;
19    }
20    static class Qnode { // Queue node inner class
21        public boolean locked = false;
22    }
23 }
```

II.2. Barriers

Imagine n threads, each of which executes method `foo()` followed by method `bar()`. Suppose we want to make sure that no thread starts `bar()` until all threads have finished `foo()`. For this kind of synchronization, we place a *barrier* between `foo()` and `bar()`.

First barrier implementation: We have a counter protected by a test-and-test-and-set lock. Each thread locks the counter, increments it, releases the lock, and spins, rereading the counter until it reaches n .

Second barrier implementation: We have an n -element Boolean array `b`, all false. Thread zero sets `b[0]` to true. Every thread i , for $0 < i \leq n$, spins until `b[i - 1]` is true, sets `b[i]` to true, and proceeds. Compare the behavior of these two implementations on a bus-based cache-coherent architecture.

II.3. Have I got the lock?

Design an `isLocked()` method that tests whether a thread is holding a lock (but does not acquire that lock). Give implementations for

- Any `testAndSet()` spin lock
- The CLH queue lock, and
- The MCS queue lock.

III. Practice

III.1. Executing Tasks: Futures and ExecutorServices

Consider the the page renderer `FuturePageRenderer` in <https://proglang.informatik.uni-freiburg.de/svn/progrep/teaching/concurrency/2014ws/exercises/published-solutions/PageRenderer>. Under the assumption that rendering text is much faster than downloading images the resulting performance is not much different from the sequential version.

Modify the `FuturePageRenderer` such that it can concurrently process the homogeneous tasks of downloading the images. You should use `Future` and `ExecutorService` again.

III.2. Adding functionality to collections

Reusing Java libraries is often preferable to creating new ones. But often the best we can find is a class that supports *almost* all the operations we want. In this case we need to add a new operation to it without undermining its thread-safety.

Implement a thread-safe `List` with an atomic `putIfAbsent()` method.

Hint: Synchronising on the `List` implementations that come with Java's collection classes nearly does the job, as they provide `contains` and `add` methods which you can reuse to construct the missing method. You can for example use the `ArrayList` class for the actual list implementation.

III.3. Sharing bathrooms

In the shared bathroom problem, there are two classes of threads, called `MALE` and `FEMALE`. There is a single `Bathroom` resource that must be used in the following way:

1. persons of opposite sex may not occupy the bathroom simultaneously, and
2. everyone who needs to use the bathroom eventually enters.

The protocol is implemented via the following four procedures: `enterMale()` delays the caller until it is ok for a male to enter the bathroom, `leaveMale()` is called when a male leaves the bathroom, while `enterFemale()` and `leaveFemale()` do the same for females.

1. Implement this class using **synchronized**, `wait()`, `notify()`, and `notifyAll()` constructs.
2. Implement this class using `ReentrantLock` and `Condition` variables.

For each implementation, explain why it satisfies mutual exclusion and starvation-freedom.

Hint: A `java.util.concurrent.locks.ReentrantLock` is the explicit version of synchronized methods and statements. A `java.util.concurrent.locks.Condition` replaces the use of the `Object` monitor methods (`wait`, `notify` and `notifyAll`) where an explicit lock is used. Familiarize yourself with these classes by studying their API documentation.

Submission

- Deadline: **2015-01-08, 23:59**
- Submit theory exercises in PDF format via email to `concurrency@informatik.uni-freiburg.de`. Please name your single file with the scheme: `ex4-name(s).pdf`.
- Submit practical exercises as executable jar-files for each exercise. The file name should include the name of the exercise and your name (example: `philosophers-fennell.jar`). Make sure that you include all source files and libraries you use. Sources should always be documented!
- Late submissions may not be corrected.
- Do not forget to write your name(s) on the exercise sheet.
- You may submit in groups up to 2 people.