CS 521/621
Homework 4
Dynamic Race Detection

Due: **Thursday, April 20, 2017, 9:00 AM EDT** via Moodle. You may work with others on this assignment but each student must submit his or her own write up, clearly specifying the collaborators. The write ups should be individual, not created jointly, and written in the student's own words. Late assignments will not be accepted without **prior** permission.

Overview

The goal of this assignment is to learn to use a dynamic race detection tool. **This assignment requires you to go a bit beyond what you've learned in lecture, and read up on writing multi-threaded programs in Java, what races are, and how to avoid them.**

The assignment consists of:

1. Using a dynamic race detection tool called CHECKSYNC.
2. Answering problem questions about dynamic race detection.

Resources

Download `http://cs.umass.edu/~brun/class/2017Spring/CS521.621/hw4/hw4.zip` with:

- The Java executable (`checkSync.jar` file) for CHECKSYNC, a dynamic race detection tool.
- The Java source code (`.java` files) and executable (`.class` files) for multi-threaded programs POOL and `Harness`, on which we will run CHECKSYNC.

At the end of this document, there is a section that explains how to interpret CHECKSYNC output. You should find this resource very useful.

Setup

POOL is a generic object-pooling library that may be used by a client program to optimize the usage of resources like sockets and database connections. The library, whose source code is provided to you, consists of class `SleepingObjectFactory` that implements an object factory, and class `GenericObjectPool` that uses the object factory to implement an object pool (denoted by the interface `ObjectPool`). Class `GenericObjectPool` extends class `BaseObjectPool` and implements interface `ObjectPool`.

A client program that uses POOL to manage its resources may call the methods declared in interface `ObjectPool`. For instance, methods `.borrowObject` and `.returnObject` enable borrowing objects from the pool and returning them back, respectively, while the `.close` method instructs the pool to close and clean up. The `ObjectPool` interface provides many other methods, but we will focus on just these three methods in this assignment.

The class `Harness`, whose source code is also provided to you, tests the implementation of the `.borrowObject`, `.returnObject`, and `.close` methods in Pool. Specifically, it simulates a multi-threaded client

program that first constructs a pool, then spawns a bunch of threads that simultaneously and repeatedly invoke the `.borrowObject` and `.returnObject` methods on that pool and, finally, when all these threads are done, closes the pool by calling the `.close` method. We wish to determine if the pool can be corrupted by this multi-threaded client program.

CHECKSYNC is a dynamic race detection tool based on the Eraser algorithm. If you are interested, you may read more about Eraser here: http://dl.acm.org/citation.cfm?id=265927

CHECKSYNC takes as input a Java program and produces the file `sync.log` that reports potential races the tool found in the program. For instance, to run the tool on the above test case, you need to execute the following:

```
java -cp ./checkSync.jar:.  edu.umd.cs.pugh.CheckSync Harness
```

- Note that the `.jar` is compiled using Java 6 (v1.6). If you use Java 7 (v1.7) or later, the binaries should execute just fine. You should not run into any compatibility problems, since you are not compiling the code, but if you do, the `-source 6` and `-target 6` arguments to `java` may prove useful.
- Note that the `:` separator in the `-classpath` argument is specific to Linux-based operating systems and that Windows uses `;`

Problems

1. Write a small test case `RaceFreeTest.java` containing a multi-threaded Java program that is race-free but for which the Eraser algorithm, and hence the CheckSync tool, reports a single false race between a pair of accesses (at least one of which is a write, of course).

   For a quick tutorial on how to write multithreaded programs in Java, see http://www.tutorialspoint.com/java/java_multithreading.htm and you may find the `Harness` sample multi-threaded program useful.

   (a) Submit `RaceFreeTest.java`

   (b) In a separate `writeup.txt` or `writeup.pdf`, cut and paste the generated `sync.log` (it must contain a single false race) along with a short explanation why you think the false race was reported.

2. Run CHECKSYNC on the `Harness` test case for the POOL library as described above. Submit the generated `sync.log` and answer the following two problems in the same `writeup.txt` from above.

   (a) For each reported race in `sync.log`, state whether it is a real race or a false race, along with a short explanation why you think it is real or false. You will need to inspect the stack trace generated for each reported race in `sync.log` and the sources of the `Harness` test case and the POOL library. You may group your answers for similar races together instead of describing each race separately. Be careful: A reported race may seem real when in fact it is false!

   (b) For each reported race that is real (as opposed to false), suggest a fix in the POOL library that eliminates it.

Deliverables

You should submit **a single .zip file** containing 3 files:

- `RaceFreeTest.java` from problem 1(a).

- `writeup.txt` or `writeup.pdf` from problems 1(b), 2(a), and 2(b).
- `sync.log` from problem 2.

Interpreting CHECKSYNC output

CHECKSYNC will write the results of tracing the synchronization in a file `sync.log`. Here is an example of a possible error:

```
Possibly broken synchronization on Elevator.myFloor
Previously accessed in Elevator.currentFloor in thread Elevator 0
 with the following locks held: Elevator
current access in thread Elevator 0:
        Elevator.serviceFloor(Elevator.java:108)
        Elevator.lessDumbAlgorithm(Elevator.java:172)
        Elevator.run(Elevator.java:236)
 with no locks held
```

This states that the variable `Elevator.myFloor` could have been accessed by different threads not holding a common lock. In this case, it points out that the last time the variable `Elevator.myFloor` was accessed, in method `Elevator.currentFloor`, a lock named `Elevator` was held. The current access, which occurred in the method `Elevator.serviceFloor` (with a stack trace showing how this was executed), had no locks held.

To use this tool well, you will probably want to give your threads meaningful names. You can do this by using the constructor `Thread(String name);` in the case you are extending `Thread`, just include `super(myname)` as the first line of your constructor.