Lab 9: Java Locks

Instructor: Vivek Sarkar, Co-Instructor: Shams Imam

Course Wiki: http://comp322.rice.edu

Staff Email: comp322-staff@mailman.rice.edu

Goals for this lab

- Experimentation with Java Locks (Coarse-grain locks & ReadWrite locks)

Importants tips and links

edX site : https://edge.edx.org/courses/RiceX/COMP322/1T2014R

Piazza site : https://piazza.com/rice/spring2016/comp322/home

Java 8 Download : https://jdk8.java.net/download.html

Maven Download : http://maven.apache.org/download.cgi

IntelliJ IDEA : http://www.jetbrains.com/idea/download/

HJlib Jar File : https://github.com/habanero-maven/hjlib-maven-repo/raw/mvn-repo/edu/rice/hjlib-cooperative/0.1.9/hjlib-cooperative-0.1.9.jar

HJlib API Documentation : http://pasiphae.cs.rice.edu/

HelloWorld Project : https://wiki.rice.edu/confluence/pages/viewpage.action?pageId=14433124

Today's lab is accessible at:

`https://svn.rice.edu/r/comp322/turnin/S16/NETID/lab_9`

Please pull this project down, import it into IntelliJ, and verify you can build it. Feel free to use whatever methods you are most comfortable with to achieve this (e.g. command-line SVN vs. IntelliJ SVN, automatic Maven-based JAR configuration vs. manual JAR imports, etc). **Note that this lab will not have a dependency on HJlib and you will not need the `-javaagent` command line option in the run configurations you use in IntelliJ.**

1 Sorted Linked List Example using Java's Synchronized Methods

In today's lab you will practice using Java Locks. Java Locks were introduced in Lecture 26.

In the provided code there are three files to focus on: `SyncList.java`, `CoarseList.java`, and `RWCoarseList.java`.

`SyncList.java` implements a thread-safe sorted linked list that supports `contains()`, `add()` and `remove()` methods. The provided `testSynchronized` test in `Lab9PerformanceTest.java` repeatedly calls these three methods with a distribution that aims for 98% read operations (calls to `contains()`), 1% add operations, and 1% remove operations. Since all three methods are declared as `synchronized` in SyncList.java, all calls will be serialized on a single `SyncList` object.

For this section, simply verify that you can compile and run the `testSynchronized` test locally using either IntelliJ or Maven. This test (and the others for the following sections of this lab) tests the throughput in operations per second of each concurrent list implementation with varying numbers of threads. The most important metric printed is the "Operations per second".

2 Use of Coarse-Grained Locking instead of Java's Synchronized Methods

The goal of this section is to replace the use of Java's synchronized method in `SyncList.java` by using explicit locking instead. For this section, your tasks are as follows:

1. Transfer the contents of the three functions `contains`, `add`, and `remove` from `SyncList.java` into `CoarseList.java`.
2. Modify `CoarseList.java` to allocate a single instance of a `ReentrantLock` when creating an instance of `CoarseList`.
3. Replace the three occurrences of "synchronized" in `SyncList` by appropriate calls to `lock()` and `unlock()` on the allocated `ReentrantLock`. Remember to use a try-finally block as follows to ensure that `unlock()` is always called:

   ```
   lock.lock();
   try { ... }
   finally { lock.unlock(); }
   ```

4. Compile and run the `testCoarseGrainedLocking` test in `Lab9PerformanceTest.java`. Compare its performance to testing the provided synchronized version using `testSynchronized`. Is there any difference? Do you expect any difference? Note that we are only running local tests at the moment, so small variations in performance are expected.

3 Use of Read-Write Locks

The goal of this section is to replace the use of a `ReentrantLock` in `CoarseList.java` by a `ReentrantReadWriteLock`, so as to leverage the fact that the majority of the operations (98% by default) are calls to `contains()` which are read-only in nature. For this section, your tasks are as follows:

1. Copy the contents of `CoarseList.java` into `RWCoarseList.java`.
2. Replace the instance of `ReentrantLock` by an instance of `ReentrantReadWriteLock`.
3. Replace the calls to `lock()` by `readLock.lock()` or `writeLock.lock()` where appropriate in `RWCoarseList.java`. Likewise for `unlock()`.
4. Compile and run the `testReadWriteLocks` test in `Lab9PerformanceTest.java`. Compare its performance to the locking and synchronized versions using `testSynchronized` and `testCoarseGrainedLocking`. Is there any change? Do you expect any difference? Note that we are only running local tests at the moment, so small variations in performance are expected.

4 Testing on NOTS

Now that we have implementations of a concurrent list using synchronized, locks, and read-write locks we will test their performance on the NOTS cluster to measure the actual performance of each implementation without interference on your laptop.

To do so, you can either use the provided myjob.slurm file or upload to the autograder. As usual, when using the myjob.slurm file please open it to fix any TODO items. If you use the autograder, feel free to ignore the printed grade and simply focus on the output in the section labeled "Performance Tests (16 cores)".

5 Turning in your lab work

For lab 9, you will need to turn in your work before leaving, as follows.

1. Show your work to an instructor or TA to get credit for this lab. In particular, the TAs will want to see the output of `testSynchronized`, `testCoarseGrainedLocking`, and `testReadWriteLocks` running on NOTS through either the autograder or the provided SLURM script.
2. Commit your work to your lab_9 turnin folder. Check that all the work for today's lab is in your lab_9 directory by opening `https://svn.rice.edu/r/comp322/turnin/S16/NETID/lab_9/` in your web browser and checking that your changes have appeared.