Lab 10: Java Locks

Instructor: Vivek Sarkar, Due: Friday April 4, 2014

Resource Summary

Course wiki: https://wiki.rice.edu/confluence/display/PARPROG/COMP322

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

Clear Login: ssh *your-netid*@ssh.clear.rice.edu and then login with your password

Important tips and links:

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

Piazza site : https://piazza.com/rice/spring2014/comp322/home

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

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

HJ-lib Jar File : http://www.cs.rice.edu/~vs3/hjlib/habanero-java-lib.jar

HJ-lib API Documentation : https://wiki.rice.edu/confluence/display/PARPROG/API+Documentation

HelloWorld Project : https://wiki.rice.edu/confluence/display/PARPROG/Download+and+Set+Up

Sugar Login: ssh *your-netid*@sugar.rice.edu and then login with your password

Linux Tutorial visit http://www.rcsg.rice.edu/tutorials/

On SUGAR, JDK8 is already available at `/users/COMP322/jdk1.8.0` and HJ-Lib is already installed at `/users/COMP322/habanero-java-lib.jar`. Run the following command to setup the JDK8 path.

```
source /users/COMP322/hjLibSetup.txt
```

When you log on to Sugar, you will be connected to a *login node* along with many other users. To request a dedicated *compute node*, you should use the following command from a SUGAR login node:

```
qsub -q commons -I -V -l nodes=1:ppn=8,walltime=00:30:00
```

We learned that there may be some issues with SUGAR this week. If you're unable to obtain a compute node, please use your laptop for today's lab instead.

Note that all commands below are CaSe-SeNsItIvE. For example, be sure to use "S14" instead of "s14".

IMPORTANT: please refer to the tutorial on Linux and SUGAR, before staring this lab. Also, if you and others experience long waiting times with the "qsub" command, please ask the TAs to announce to everyone that they should use ppn=4 instead of ppn=8 in their qsub command (to request 4 cores instead of 8 cores).

1 Sorted Linked List Example using Java's Synchronized Methods

NOTE: see slides for Lectures 26 and 27 for a recap of Java's synchronized statement and locking libraries respectively.

Download the `lab10.zip` archive from the course web page. It consist of six files: `SyncList.java`, `ListDriver.java`, `ListCounter.java`, `ListSet.java`, `ListTest.java`, `RWMix.java`. Of these, you only need to focus on `SyncList.java`, which contains a thread-safe implementation of a sorted linked list that supports `contains()`, `add()` and `remove()` methods. The default driver options 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, your tasks are as follows:

1. Compile all Java files by issuing the command, `javac *.java`.
2. Execute the `SyncList` class with the default driver options for 1, 2, 4, 8 threads, by issuing the following commands:
   ```
   java ListDriver -t 1 -b ListTest -s SyncList
   java ListDriver -t 2 -b ListTest -s SyncList
   java ListDriver -t 4 -b ListTest -s SyncList
   java ListDriver -t 8 -b ListTest -s SyncList
   ```

   Observe the performance reported next to the text "Operations per seconds:". Since this is a throughput metric, a larger value will indicate better performance. How does the performance vary with number of threads? Can you explain why this happens?

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 explicit locking instead. For this section, your tasks are as follows:

1. Make a copy of `SyncList.java` named `CoarseList.java`.
2. Replace two occurrences of "SyncList" by "CoarseList" in `CoarseList.java`.
3. Allocate a single instance of `ReentrantLock` when creating an instance of `CoarseList`. See slides 12 and 13 in Lecture 27 for this step, and the remaining steps below.
4. Replace the three occurrences of "synchronized" by appropriate calls to `lock()` and `unlock()`. Remember to use a try-finally block as follows to ensure that `unlock()` is always called:

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

5. Compile all Java files by issuing the command, `javac *.java`.
6. Execute the `CoarseList` class with the default driver options for 1, 2, 4, 8 threads, by issuing the following commands:
   ```
   java ListDriver -t 1 -b ListTest -s CoarseList
   ```

```
java ListDriver -t 2 -b ListTest -s CoarseList
java ListDriver -t 4 -b ListTest -s CoarseList
java ListDriver -t 8 -b ListTest -s CoarseList
```

How does the performance compare with the performance observed for `SyncList`?

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. Make a copy of `CoarseList.java` named `CoarseRWList.java`.
2. Replace two occurrences of "CoarseList" by "CoarseRWList" in `CoarseRWList.java`.
3. Replace the instance of `ReentrantLock` by an instance of `ReentrantReadWriteLock`. See slides 19 and 20 in Lecture 27 for this step, and the remaining steps below.
4. Replace the calls to lock() by readLock.lock() or writeLock.lock() where appropriate. Likewise for unlock().
5. Compile all Java files by issuing the command, `javac *.java`.
6. Execute the `CoarseRWList` class with the default driver options for 1, 2, 4, 8 threads, by issuing the following commands:
   ```
   java ListDriver -t 1 -b ListTest -s CoarseRWList
   java ListDriver -t 2 -b ListTest -s CoarseRWList
   java ListDriver -t 4 -b ListTest -s CoarseRWList
   java ListDriver -t 8 -b ListTest -s CoarseRWList
   ```

   How does the performance compare with the performance observed for `CoarseList`?
7. This example also allows for selection of different fractions of read (combine), add, and remove operations. For practicality, it is important to use the same fraction for add and remove operations (otherwise the list will become grow too large or too small). To see an increased benefit with read-write locks, you can add the "-r 100 -a 0 -d 0" options to the driver program to make the fraction of read operations 100% (an extreme case).

   Now execute the following commands to compare the performance of `CoarseList` and `CoarseRWList` on 8 threads with 100% read operations:
   ```
   java ListDriver -t 8 -b ListTest -r 100 -a 0 -d 0 -s CoarseList
   java ListDriver -t 8 -b ListTest -r 100 -a 0 -d 0 -s CoarseRWList
   ```