Lab 10: Java Locks

Instructor: Vivek Sarkar

1 Turning in your lab assignments — NEW!

We're asking all COMP 322 students to turn in their lab assignments before leaving. You will need to do the following:

1. Create a directory called `lab_10/` in your SUGAR account.
2. Do all your work for today's lab in this directory.
3. Before you leave, create a zip file of your work by changing to the parent directory for `lab_10/` and issuing the following command, " `zip -r lab_10.zip lab_10`".
4. Use the turn-in script to submit the contents of the `lab_10.zip` file as a new `lab_10` directory in your turnin directory. (Transfer the file to your CLEAR account of needed.)

2 Setup on SUGAR

As before, run the following command on SUGAR to setup the environment for executing HJ and Java programs:
source /users/COMP322/hjsetup.txt

To request a dedicated *compute node*, you should use the following command (as usual) from a SUGAR login node:
qsub -q commons -I -V -l nodes=1:ppn=8,walltime=00:30:00
When successful, it will give you a command shell on a dedicated 8-core compute node for your use for 30 minutes at a time. Your home directory is the same on both the login and compute nodes.

3 Sorted Linked List Example using Java's Synchronized Methods

NOTE: see slides for Lectures 29 and 30 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 creates 8 threads that repeatedly calls these three methods with a distribution that aims for 98% read operations (calls to `contains()`), 1% add operations, and 1% remove operations.

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 by issuing the command,
   `java ListDriver -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.

4 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 19 and 20 in Lecture 30 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 by issuing the command,
   `java ListDriver -b ListTest -s CoarseList`
   How does the performance compare with the performance observed for `SyncList`?
7. You can change the number of threads by using the "-t" option in the driver. Re-run the `SyncList` and `CoarseList` classes with 1 thread instead of the default 8 threads by issuing the following commands:
   `java ListDriver -t 1 -b ListTest -s SyncList`
   `java ListDriver -t 1 -b ListTest -s CoarseList`
   Can you explain the performance differences that you observe between 8 threads and 1 thread?

5 Use of Read-Write Locks

The goal of this section is to replace the use of a `ReentrantLock` in `CoarseList.java` by a `ReadWriteReentrantLock`, 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 `ReadWriteReentrantLock`. See slides 26 and 27 in Lecture 30 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 by issuing the command,
   `java ListDriver -b ListTest -s CoarseRWList`
   How does the performance compare with the performance observed for `CoarseList`? Can you explain the difference?