Lab 9: Java Threads and Locks

Instructor: Vivek Sarkar, Co-Instructor: Mackale Joyner

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

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

Goals for today's lab

- Experimentation with Java threads
- Experimentation with regular locks and read-write locks in Java

This lab can be downloaded from the following svn repository:

- `https://svn.rice.edu/r/comp322/turnin/S17/`*`NETID`*`/lab_9`

Use the subversion command-line client or IntelliJ to checkout the project into appropriate directories locally.

In today's lab, you need to use NOTS to run performance tests. If you need any guidance on how to submit jobs on NOTS manually or through the autograder, please refer to earlier labs or ask a member of the teaching staff.

1 Conversion to Java threads: Spanning Tree

1. The `SpanningTreeSeq.java` program is an example sequential solution to the spanning tree problem.

   The `SpanningTreeAtomicHjLib.java` program is a provided parallel solution to the minimum spanning tree problem. This version uses finish and async constructs along with an AtomicReference.

2. Your task is to convert SpanningTreeAtomicHjLib.java to a Java program that uses threads instead of HJlib tasks. You should modify the provided `SpanningTreeAtomicThreads.java` file, and use Java thread methods instead of finish/async. (The AtomicReference calls can stay unchanged.) As before, you can include joins within each call to `compute()` for simplicity, or you can use a `ConcurrentLinkedQueue` to collect child Thread objects for a more faithful simulation of a finish construct.

3. You have been provided with tests for your parallel spanning tree implementation in SpanningTreePerformanceTest. To complete this portion of the lab, you should submit these performance tests to NOTS by either modifying the provided myjob.slurm template and submitting manually, or through the autograder. How does HJlib performance compare to using Java Threads? Which version is easier to write and read?

2 Programming Tips and Pitfalls for Java Threads

- Remember to call the *start()* method on any thread that you create. Otherwise, the thread's computation does not get executed.
- Since the *join()* method may potentially throw an InterruptedException, you will either need to enclose each call to *join()* within a *try-catch block*, or add a *throws InterruptedException* clause to the definition of the method that includes the call to *join()*.

3 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. **Note that the sorted list exercises will not have a dependency on HJlib; you will not need the `-javaagent` command line option in the run configurations you use in IntelliJ for these exercises.**

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 `SortedListPerformanceTest.java` repeatedly calls these three methods with a distribution that aims for 99% read operations (calls to `contains()`) and 1% add 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".

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 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`. The term *coarse locking* is used for cases like this when a single lock is used to protect the entire data structure, as opposed to *fine-grained locking* in which different locks may be used to protect different components (e.g., nodes) in a data structure.
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 `SortedListPerformanceTest.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.

5 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 (99% by default) are calls to `contains()` which are read-only in nature and can execute in parallel with each other. 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 `SortedListPerformanceTest.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.

6 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, focus on:

1. Comparing the performance achieved by synchronized, coarse-grain locking, and read-write locks between the two panes called "Performance Tests (1 core)" and "Performance Tests (8 cores)". How does performance change for each when only using 1 core or 8 cores?
2. The speedup achieved in the testSpanningTreeThreads test at the bottom of the "Performance Tests (8 cores)" pane.

7 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`, and `testSpanningTreeThreads` 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/S17/NETID/lab_9/` in your web browser and checking that your changes have appeared.