# In-Class Exercise: Java Threads

Instructor: Mackale Joyner

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

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

## Goals for today's exercise

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

This in-class exercise can be downloaded from the following svn repository:

- `https://svn.rice.edu/r/comp322/turnin/S20/`*`NETID`*`/inclass_ex`

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

In today's exercise, you need to use NOTS to run performance tests.

## 1 Sorted Linked List Example using Java's Synchronized Methods

In today's exercise you will practice using Java Locks. Java Locks were introduced in Lecture 27. **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".

## 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. In `CoarseList.java`, 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(); }
```

2. Compile and run the `testCoarseGrainedLocking` test in `SortedListPerformanceTest.java`. Compare its performance to testing the provided synchronized version using `testSynchronized`.

# 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 (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. Replace the calls to `lock()` by `readLock.lock()` or `writeLock.lock()` where appropriate in `RWCoarseList.java`. Likewise for `unlock()`.
2. Compile and run the `testReadWriteLocks` test in `SortedListPerformanceTest.java`. Compare its performance to the locking and synchronized versions using `testSynchronized` and `testCoarseGrainedLocking`.

# 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 should use the provided myjob.slurm file. As usual, when using the myjob.slurm file please open it to fix any TODO items.