1.1 Lecture Summary

1.1 Java Threads

Lecture Summary: In this lecture, we learned the concept of threads as lower-level building blocks for concurrent programs. A unique aspect of Java compared to prior mainstream programming languages is that Java included the notions of threads (as instances of the java.lang.Thread class) in its language definition right from the start.

When an instance of Thread is *created* (via a new operation), it does not start executing right away; instead, it can only start executing when its start() method is invoked. The statement or computation to be executed by the thread is specified as a parameter to the constructor.

The Thread class also includes a *wait* operation in the form of a join() method. If thread t0 performs a t1.join() call, thread t0 will be forced to wait until thread t1 completes, after which point it can safely access any values computed by thread t1. Since there is no restriction on which thread can perform a join on which other thread, it is possible for a programmer to erroneously create a *deadlock cycle* with join operations. (A deadlock occurs when two threads wait for each other indefinitely, so that neither can make any progress.)

Further Reading:

1. Wikipedia article on Threads
2. Tutorial on Java threads
3. Documentation on Thread class in Java 8

1.2 Lecture Summary

1.2 Structured Locks

Lecture Summary: In this lecture, we learned about *structured locks*, and how they can be implemented using synchronized statements and methods in Java. Structured locks can be used to enforce *mutual exclusion* and avoid *data races*, as illustrated by the incr() method in the A.count example, and the insert() and remove() methods in the the Buffer example. A major benefit of structured locks is that their *acquire* and *release* operations are implicit, since these operations are automatically performed by the Java runtime environment when entering and exiting the scope of a synchronized statement or method, even if an exception is thrown in the middle.

We also learned about wait() and notify() operations that can be used to block and resume threads that need to wait for specific conditions. For example, a producer thread performing an insert() operation on a *bounded buffer* can call wait() when the buffer is full, so that it is only unblocked when a consumer

thread performing a remove() operation calls notify(). Likewise, a consumer thread performing a remove() operation on a *bounded buffer* can call wait() when the buffer is empty, so that it is only unblocked when a producer thread performing an insert() operation calls notify(). Structured locks are also referred to as *intrinsic locks* or *monitors*.

Optional Reading:

1. Tutorial on Intrinsic Locks and Synchronization in Java
2. Tutorial on Guarded Blocks in Java
3. Wikipedia article on Monitors

1.3 Lecture Summary

1.3 Unstructured Locks

Lecture Summary: In this lecture, we introduced *unstructured locks* (which can be obtained in Java by creating instances of ReentrantLock()), and used three examples to demonstrate their generality relative to structured locks. The first example showed how explicit lock() and unlock() operations on unstructured locks can be used to support a *hand-over-hand* locking pattern that implements a non-nested pairing of lock/unlock operations which cannot be achieved with synchronized statements/methods. The second example showed how the tryLock() operations in unstructured locks can enable a thread to check the availability of a lock, and thereby acquire it if it is available or do something else if it is not. The third example illustrated the value of *read-write locks* (which can be obtained in Java by creating instances of ReentrantReadWriteLock()), whereby multiple threads are permitted to acquire a lock L in “read mode”, L.readLock().lock(), but only one thread is permitted to acquire the lock in “write mode”, L.writeLock().lock().

However, it is also important to remember that the generality and power of unstructured locks is accompanied by an extra responsibility on the part of the programmer, e.g., ensuring that calls to unlock() are not forgotten, even in the presence of exceptions.

Optional Reading:

1. Tutorial on Lock Objects in Java
2. Documentation on Java’s Lock interfaces

1.4 Lecture Summary

1.4 Liveness and Progress Guarantees

Lecture Summary: In this lecture, we studied three ways in which a parallel program may enter a state in which it stops making forward progress. For sequential programs, an “infinite loop” is a common way for a program to stop making forward progress, but there are other ways to obtain an absence of progress in a parallel program. The first is *deadlock*, in which all threads are blocked indefinitely, thereby preventing any forward progress. The second is *livelock*, in which all threads repeatedly perform an interaction that prevents forward progress, e.g., an infinite “loop” of repeating lock acquire/release patterns. The third is *starvation*, in which at least one thread is prevented from making any forward progress.

The term “liveness” refers to a progress guarantee. The three progress guarantees that correspond to the absence of the conditions listed above are *deadlock freedom*, *livelock freedom*, and *starvation freedom*.

Optional Reading:

1. Deadlock example with synchronized methods in Java
2. Starvation and Livelock examples in Java
3. Wikipedia article on Deadlock and Livelock

1.5 Lecture Summary

1.5 Dining Philosophers Problem

Lecture Summary: In this lecture, we studied a classical concurrent programming example that is referred to as the *Dining Philosophers Problem*. In this problem, there are five threads, each of which models a “philosopher” that repeatedly performs a sequence of actions which include *think*, *pick up chopsticks*, *eat*, and *put down chopsticks*.

First, we examined a solution to this problem using structured locks, and demonstrated how this solution could lead to a deadlock scenario (but not livelock). Second, we examined a solution using unstructured locks with tryLock() and unlock() operations that never block, and demonstrated how this solution could lead to a livelock scenario (but not deadlock). Finally, we observed how a simple modification to the first solution with structured locks, in which one philosopher picks up their right chopstick and their left, while the others pick up their left chopstick first and then their right, can guarantee an absence of deadlock.

Optional Reading:

1. Wikipedia article on the Dining Philosophers Problem

Mini Project 1: Locking and Synchronization

---

miniproject_1.zip

Project Goals and Outcomes

Threads are the assembly language of parallel computing. They enable programmers to explicitly express parallel computation at a low level, and are supported across a wide range of processors and operating systems. However, one of the dangers of using threads (and concurrency in general) is the risk of data races.

A data race is an unsafe access to the same piece of data from two independently executing threads, without some mechanism in place to ensure that those accesses do not conflict with each other. A data race may only occur when there is no synchronization in place to prevent concurrent access by multiple threads, and when at least one of those accesses is a write to or modification of the data.

This week, you learned about two mechanisms for performing synchronization in concurrent Java programs: Java locks and the Java synchronized statement. In this mini-project, you will gain hands-on experience with writing code using the synchronized statement, Java locks, and Java read-write locks. You will do so by implementing a concurrent, *sorted* list data structure. Using an artificial workload of randomly generated reads and writes, you will also measure the performance benefit of read-write locks over the other synchronization techniques while inserting, removing, and checking for items in the sorted list.

Project Setup

Please refer to Mini-Project 0 for a description of the build and testing process used in this course.

Once you have downloaded and unzipped the project files using the gray button labeled miniproject_1.zip at the top of this description, you should see the project source code file at

miniproject_1/src/main/java/edu/coursera/concurrent/CoarseLists.java

and the project tests in

miniproject_1/src/test/java/edu/coursera/concurrent/ListSetTest.java

It is recommended that you review the demo video for this week before starting this assignment, in which Professor Sarkar works through a similar example.

Project Instructions

Your modifications should be made entirely inside of CoarseLists.java. You should not change the signatures of any public or protected methods inside of CoarseLists.java, but you can edit the method bodies and add any new methods that you choose. We will use our copy of ListSetTest.java in the final grading process, so do not change that file or any other file except CoarseLists .java.

Your main goals for this assignment are as follows:

1. Inside CoarseLists.java, implement the CoarseList class using a ReentrantLock object to protect the add, remove, and contains methods from concurrent accesses. You should refer to SyncList.java as a guide for placing synchronization and understanding the list management logic.
2. Inside CoarseLists.java, implement the RWCoarseList class using a ReentrantReadWriteLock object to protect the add, remove, and contains methods from concurrent accesses. You should refer to SyncList.java as a guide for placing synchronization and understanding the list management logic.

There are helpful TODOs in CoarseLists.java to help guide your implementation.

Project Evaluation

Your assignment submission should consist of only the CoarseLists.java file that you modified to implement this mini-project. As before, you can upload this file through the assignment page for this mini-project. After that, the Coursera autograder will take over and assess your submission, which includes building your code and running it on one or more tests. Your submission will be evaluated on Coursera's auto-grading system using 2 and 4 CPU cores. Note that the performance observed for tests on your local machine may differ from that on Coursera's auto-grading system, but that you will only be evaluated on the measured performance on Coursera. Also note that for all assignments in this course you are free to resubmit as many times as you like. See the Common Pitfalls page under Resources for more details. Please give it a few minutes to complete the grading. Once it has completed, you should see a score appear in the "Score" column of the "My submission" tab based on the following rubric:

- 20% - performance of CoarseList on two cores
- 20% – performance of RWCoarseList on two cores
- 30% – performance of CoarseList on four cores
- 30% – performance of RWCoarseList on four cores