CMPSCI 691W Parallel and Concurrent Programming **Spring 2006**

Lecture 5: February 13

Lecturer: Emery Berger *Scribe: Dennis Gove*

5.1 Overview

This lecture is the second of two concerning concurrency in Java (specifically concurrency *improvements* in Java 1.5 - also known as Java 5). We begin by discussing some problems with standard locks and then delve into some java implementations meant to solve those problems (including atomic integers).

5.2 Locks

A quick review of locks will remind us that locks provide *safety* for shared data. That is, only one thread at at time will be allowed to access a given piece of memory. You'll recall that data locking is accomplished in java via a special command *synchronized(variable)* - one can also synchronize entire methods.

5.2.1 Problems with Locks

That's right, you can't have your cake and eat it too - there are some problems with java locks. First, you can't attempt to acquire a lock and then stop trying after some timeout. Second, there is no concept of a reader/writer lock. Third, all locks are reentrant-able (allows recursion, but *very* overhead intensive). Fourth, any method can call *synchronized* which leaves us with almost no access control. Fifth, we only have block structured locking (*synchronized{ do what you want.... until the end of the block}*). Sixth, priority inversion can occur if a low-priority process acquires a lock then gets switched out for a higher-priority process and the high-priority process must block because it needs the same lock. Seventh, convoying can occur where everyone who needs a lock must wait for the slowest process to release the lock (think of a convoy of fruit trucks where the speed of the lead truck maxes at 50mph while the final truck flies at 80mps - the final truck must slow down to 50mph) Don't worry, there are some solutions.

5.2.2 Some Solutions

- java.util.concurrent.locks

The package *concurrent.locks* provides us with the ability to use the familiar lock(), unlock() as well as trylock(), trylock(time, timeunit), reentrant locks, and reentrantReadWrite locks. The trylock() will return if it cannot acquire the lock, while the trylock(time, timeunit) will continually try for time timeunits (5, TimeUnit.SECONDS) means it will try for 5 seconds. There is support for *rolling your own* using conditions, though this is discouraged unless you are an expert (discouraged to the tune of *don't do it - ever*).

- java.util.concurrent.atomic
  The package *java.util.concurrent.atomic* provides us with the ability to get at some hardware level atomic operations (*test-and-set, etc...*) which are the building blocks for non-blocking data structures.

  - AtomicInteger
    * set(int)
    * get()
    * addAndGet(int), incrementAndGet(), getAndAdd(int), getAndIncrement()
    * compareAndSet(expected, update)
  - AtomicReference
    * set(V newValue)
    * get(), getAndSet(V newValue)
    * compareAndSet(expected, update)

  The atomicReference can lead to the Stack-ABA problem where we are poping things off a stack whereby we pop(A), (switch process) somebody else pops(B), (switch process back to us) we push(A). This will lead to a situation where the head points to B, but should point to A. (see slides 9-15 of lecture 5: Advanced Java Concurrency for a pictorial rep). What is the solution? We could stamp our references and make sure that the stamp we expect to be there is actually the one that is there. A good analagy is a file versioning system (SVN, CVS)

  - AtomicStampedReference
    * set(V newValue)
    * get(), getAndSet(V newValue)
    * compareAndSet(expected, update)

- java.nio
  The package java.nio provides support for non-blocking i/o operations and other low-level i/o (memory mapped byte buffers, channels, pipes, selectors)
  - Memory Mapped Buffers
    * An array is mapped to a file on disk and uses VM operations to access.
    * Access to buffer in memory equal a call to the disk
    * This is much faster than regular i/o because i/o is the slowest thing in the world and if we can amortize that with VM calls we are much better off.
  - Selectors
    * Same idea as select(). You are able to add any channels of interest and then start io operations (remember that it must be configured as non-blocking (sc.configureblocking(false))
    * When called it returns an iterator with channels ready for i/o operations.