CS 131 Programming Languages: HW3 Report

Abstract

This assignment familiarized me with Java programming and to gain great appreciation for its thread interface, especially relative to C. The goal of the assignment was to demonstrate concurrency in Java, using increment and decrement with multithreading. We are to implement various models and compare their performance.

1. Setup

The assignment requires that the program should operate under Java Standard Edition 10. In addition, we are to gather and report statistics about my machine and development environment.

I developed on the SEASnet server lnxsrv07 which runs RedHat. I ran the following command to ensure the Java version on the SEAS server is adequate:

$ java -version

java version "10.0.1" 2018-04-17

Java(TM) SE Runtime Environment 18.3 (build 10.0.1+10)

Java HotSpot(TM) 64-Bit Server VM 18.3 (build 10.0.1+10, mixed mode)

To find info on my machine, I checked the following files:

$ cat /proc/cpuinfo

$ cat /proc/meminfo

My machine is running with 64GB RAM and uses an Intel(R) Xeon(R) CPU E5-2640 v2 @ 2.00GHz for all 32 processors.

2. Descriptions

2.1. Synchronized

The Synchronized class is entirely reliable, such that it is data race free. As the name implies, the implementation is synchronized so multiple threads cannot execute the block of code simultaneously. Therefore, it runs the slowest of the models. This shows the tradeoff of performance and reliability; though this is the safest of the models, it sacrifices the performance capabilities.

2.2. Null

The Null class does essentially nothing, but is a good reference for testing (see section 5).

2.3. Unsynchronized

The implementation of the Unsynchronized class is the same as the Synchronized class, except as the implementation is, again, as the name implies, unsynchronized. This exposes it to data race, and frequent failures as it can entire an infinite loop often with relatively normal arguments. Thus, it is especially unreliable as a result.

2.4. GetNSet

This implementation, as required in the spec, is done with with the get and set methods of java.util.concurrent.atomic.AtomicIntegerArray. Consequently, it is not data race free because the get and set methods force the scheduler to alternate between instructions. This comes relative to the corresponding atomic methods getAndIncrement and getAndDecrement which would introduce a data race free implementation. Though, it is not data race free.

2.5. BetterSafe

The implementation of BetterSafe is data race free, as I used a ReentrantLock from java.util.concurrent.locks, which by definition allows for the lock to prevent read and write to introduce race conditions. The Java Memory Model, with the Reentrant Lock, will prioritize execution, instead of the overhead of scheduling.

3. Testing

I tested each model with 1, 2, 4, 8, 16, 32 threads, and ten-thousand, one hundred-thousand, and a million swaps, maxval 6, and the first five entries of the array: 5 6 3 0 3.

The shell commands I ran was based on the one given on the specification:

$ java UnsafeMemory _____ [1,2,4,8,16,32] 10000[00] 6 5 6 3 0 3

where _____ is Null, Synchronized, Unsynchronized, GetNSet, and BetterSafe.

For the last three, that is, the classes I implemented, I needed to modify UnsafeMemory to allow for those arguments. I note this because I did not realize I had to edit this file and spent two hours attempting to debug a bug that did not exist.

4. Comparisons

Model	**Threads average ___ ns/transition**
Null	834.09
Synchronized	4999.57
**Unsynchronized*	910.85
**GetNSet*	1204.68
BetterSafe	1854.95

* Not DRF

5. Analysis

As mentioned earlier, the Null model does nothing, which just provides a reference point. Considering how the performance for the Unsynchronized model is very close to the Null state, it is the worst implementation considering the frequent failures, it may as well do nothing.

The synchronized model is by far the slowest, as no concurrency happens whatsoever. Though, it is very reliable.

The GetNSet performance is the best, but is less reliable than BetterSafe. Of all the models, it seems that BetterSafe is the one I would choose for GDI as the difference in performance not significantly different, especially considering the alternatives where wither the performance suffers drastically, or the reliability is just the opposite.

Furthermore, because BetterSafe is data race free and GetNSet is not, and dead locks very frequently, the reliability is a lucrative and the tradeoff with performance is minimal.