CS 131 Report Homework 3

Mehar Nallamalli

Abstract— **The main goal for this homework is to test different variations of concurrent classes in Java when working with a decrement/increment swap function from an array of values. Specifically, I used locks to explore the tradeoffs between locking threads and performance.**

Keywords—**OpenCV, facial recognition, eye tracking, yawn detection, Heartrate, alerting.**

I. TESTING PLATFORM

By using the command, java -version, I was able to find out the Java version:

- `java version "1.8.0_181"`
- `Java(TM) SE Runtime Environment (build 1.8.0_181-b13)`
- `Java HotSpot(TM) 64-Bit Server VM (build 25.181-b13, mixed mode)`

Once my environment was set up, I completed developing AcmeSafe on my local machine, and then tested it using the lnxserver09.

II. TESTING PARAMETERS

After I finished implementing AcmeSafe, I wanted to test different input parameters into the command. Using 1 million swaps, I measured the thread average for 8, 16, 32 threads using the range of 0-127.
In my UnsafeMemory.java, I added parameters for Null, Synchronized, Unsynchronized, and AcmeSafe.

III. TESTING RESULTS

Table 1.1: Results for 1 million swaps

Model	**8**	**16**	**32**	**DRF**
Null	670	1695	2529	Y
Synchronized	1280	2530	5477	Y
Unsynchronized	37940	58214	167430	N
AcmeSafe	953	1841	4643	Y

IV. COMPARISONS

According to the table above, it shows that the Unsynchronized is slower than the Synchronized. This is because with a large number of threads, it becomes unreliable. Specifically, an unsuccessful swap is not counted, so if all numbers accidentally go to 0, the program will hang forever.

In fact, I was not able to have a successful result with threads more than 100,000+. I had to test this Unsynchronized model using only 10,000 threads, and the transition speed was much higher than synchronized.

There is no overhead for Unsync and is a much weaker model compared to Synchronized. The downside is that it is not 100% reliable where as AcmeSafe is. Null is the fastest for all threads compared to the rest of the models. But the Null class is not actually swapping elements, it just returns true if the swap function is called. It is the fastest because it has the lowest protection in memory, and also because there is nothing to protect. In fact, the Null model performs better than the safest, AcmeSafe model.

AcmeSafe is better than any other implementation because it utilized Reentrant locks which will make sure we do not run into race conditions. AcmeSafe is thus able to be much more reliable than the other models because it does not report a mismatch. It only synchronizes the critical section using locks rather than locking eh entire function.

AcmeSafe also is better for memory protection than Unsynchronized, but it still is more efficient for higher threads because of its locking for specific critical sections rather than the whole function. AcmeSafe is also better than Synchronized because it is weaker model in memory since it does not lock the entire function. The upside is that it is 100% reliable and is the best implementation for this company dealing with large sets of data.

A. *Java.util.concurrent.locks.**

This package gives a bunch of different locking techniques such as ReentrantLock and LockSupport. I chose to use ReentrantLock for handling critical conditions. For AcmeSafe, I lock the swap method, but instead of the entire function, I used a more granular locking mechanism and it only locks the race condition parts of the code. **AcmeSafe is also DRF**. Because I only lock the racey code, it is more efficient for multithreading. This package also improves performance and reduces starvation in memory.

The one downside to locks is that it prevents other threads to modify different memory blocks than what is currently locked. But in this simple example, that is not an issue. This ReentrantLock will increase performance compared to the Synchronized code because it only locks the critical section, thus allowing more customization and flexibility.

V. AcmeSafe vs Synchronization

As mentioned above, the AcmeSafe implementation is faster than Synchrnoized because the entire swap function is not synchronized, but only the read and write section. This model is 100% reliable since a write can happen before or during a read, which is a collision. Locking only that section results in stronger memory model.

VI. Conclusion

According to my testing (multiple threads numbers, multiple swap operation sizes), I found that the AcmeSafe implementation is the most reliable (DRF), and the small increase in speed is expected due to its overhead.