CS131 HW3 Report: Java shared memory performance races

1 Introduction

In a theoretical company that uses multithreading to speed up its applications, what would happen if you were told to remove the synchronization methods used in order to avoid race conditions? How would you modify the program in a way to ensure data synchronization but also preserve speed? For this project, our goal was to test the different methods of synchronization, including the absence of synchronization, used in Java concurrent programming in order to observe their effects on an application.This was done by creating four different versions of a simple swap program, each using a different synchronization method. The functions were implemented in these four classes as follows: SynchronizedState, Unsynchronized, GetNSet, and BetterSafe.

2 Classes

2.1 Synchronized

The SynchronizedState class utilizes the built-in java synchronized keyword in order to avoid race conditions. When this keyword is applied to a function, it surrounds the that whole section of code with a lock so that only one thread can access it at a single time. Though this ensures data race free (DRF) conditions, it does so at the cost of speed and efficiency.

2.2 Unsynchronized

The Unsynchronized class is just a modified version of the SynchronizedState class but with the synchronized keyword removed. Due to the lack of locks that can obstruct threads and cause bottlenecks, unsynchronized provides quickness and efficiency. However, since this class contains no method for removing race conditions whatsoever, it is highly susceptible to inconsistencies due to concurrency.

2.3 GetNSet

Instead of blocking entire sections of code that may be susceptible to race conditions as in the Synchronized class, GetNSet utilizes volatile accesses to memory elements defined by the Java memory model (JMM). By using the AtomicIntegerArray, GetNSet forces threads to atomically access its critical elements, thus preventing race conditions. This method is not foolproof, however, since there exists specific cases where race conditions can still happen, but those are few and far between. Overall, GetNSet provides a good middle ground between the reliability of Synchronized and the speed of Unsynchronized.

2.4 BetterSafe

In order to maintain the guarantee of DRF while also preserving speed, we came up with the BetterSafe class. This class was implemented by using the java.util.concurrent reentrant lock due to its simplicity of use and it's guaranteed reliability. Just like any other lock we've seen before, this lock is implemented by surrounding the critical areas of code with its lock and unlock function calls. Even though reentrant locks are used the same way as Synchronized locks, its internal implementation makes it work much more efficiently. To explain, as its name implies, the reentrant lock allows threads to reenter the same locked code more than once while it is still in the code. This is implemented with an incremental lock that is then decremented by lock requests from other threads. This allows for optimized accesses into the critical regions of code via a fairness system. Because of this feature, the use of reentrant locks in BetterSafe helps speed up the program while still guaranteeing DRF like the SynchronizedState class.

3 Test Results

ns/transition					
Threads	**Null**	**Sync**	**Unsync**	**GetNSet**	**Bettersafe**
8	131.139	2509.62	-	849.494	697.188
16	386.727	3977.87	-	1709.07	1377.71
32	919.59	8203.83	-	3878.43	2961.95

Table 1: The time per transition in nanoseconds(ns) for each synchronization method. Each test was performed with 10,000,000 swaps using the indicated number of threads. Null is a program that does nothing and only is used for timing comparisons.

3.1 Performance

As can be seen in the table above, the SynchronizationState class performed the slowest out of all of them but always ended up getting the correct result due to its guaranteed DRF. On the other hand, the Unsynchronized class was unable to complete a single trial due to its lack of protection from race conditions. The threads ended up falling into an infinite loop, preventing them from completing their tasks. The unsynchronized class was only able to regularly complete tests when there were significantly less swaps. In these cases, it performed faster than all of the other synchronized classes, however, the result ended up incorrect most of the time. GetNSet, being the middle ground between sync and unsync, performed twice as efficiently as SynchronizedState yet, never got an incorrect result in any of the tests. Though GetNSet's results were impressive, it has a glaring flaw in its lack of guaranteed DRF. Our last class, BetterSafe, addresses this problem by providing the speed and efficiency of an unsynchronized function while also guaranteeing the DRF conditions of a synchronized function. Based on the results, BetterSafe performed quite closely to GetNSet, but was slightly faster. Out of all of the classes, BetterSafe performed the most reliably and efficiently as predicted due to the features described in section 2.4.

Conclusion

To conclude, from these observations when optimizing for concurrency, the main source of contention is whether to sacrifice efficiency for reliability or vice versa. However, we were able to create a good middle ground that was both reasonably fast and accurate. Along the way, there were some problems that occured in testing. Most were minor, like small accidental bugs in the code that were quickly fixed. The biggest problem during testing was trying to test Unsynchronized with the same test conditions as all the other classes and having it succeed. Though the other classes succeeded in all tests, Unsynchronized failed almost all of them. Testing had to be done with smaller amounts of swaps in order for it to finally succeed. In all successful test cases Unsynchronized performed the fastest out of all the classes, however, it always returned with an incorrect result. In the end, the Synchronized and Unsynchronized classes represented two extremes in the efficiency/reliability spectrum. The best way to deal with this issue was to create a synchronization method that contained the best features of both. We implemented the BetterSafe class to do just this, and due to its speed and guaranteed DRF, BetterSafe ended up being the best choice among all four classes.

Citations

http://gee.cs.oswego.edu/dl/html/j9mm.html

http://winterbe.com/posts/2015/04/30/java8-concurrency-tutorial-synchronized-locks-examples/