CS131 Homework 3 Report

1. Introduction

The JMM, or Java Memory Model, describes how the Java virtual machine works in RAM. It describes how multithreaded programs should be handled and specifies how programs and processes can access shared memory, while simultaneously preventing race conditions.

In this lab, we examined four different synchronization methods and each ones effects on multithreaded programs in Java. These parallelization methods included using a keyword, not doing any concurrency, using specialized data structures to perform atomic accesses, and using Java's Reentrant Locks.

2. Implementations

Java comes with some built-in synchronization functionality, and they scope in which they perform range from high-level forms that are made for implementations that wants to see concurrency to low-level forms that gives the user more control, although this would lead to a greater chance of mistakes being made.

The implementation of concurrency can be a very challenging aspect of programming. If done carelessly, this can lead to some nasty bugs and result in incorrect behavior. Because of this point, using the higher-level methods provided by Java is a better option for most Java users. However, people are know what they are doing and would put in the time and effort to debug their parallelized applications can use the lower-level functionality. This tradeoff can be worth it in some situations, as the decrease in overhead in lower-level applications would result in better performance.

2.1 java.util.concurrent

This package basically provides some utility classes and the data structures that allows users to make a class thread-safe. It can be used in apps like Synchronizers, Concurrent Collections, and Executors to perform concurrent functions. It offers a very simple, efficient, straightforward method of preventing issues that exists with threads.

3.2 java.util.concurrent.atomic

This package basically provides a small toolkit of classes that allow users to do lock-free, thread-safe programming on single variables, updating the associated values in these variables in an atomic action. This therefore results in programs that were threadsafe.

This package would have resulted in faster performance for BetterSafe because of the lower overhead in performing atomic updates versus using a lock to lock and unlock. However, this doesn't solve all our issues with BetterSafe. If our program didn't have to do reading and writing to locks, this would have been perfect. Because this is necessary, locks were the best way of accomplishing this. A potential situation to demonstrate the issue with atomic operations is if the values in the array are incremented after being checked for validity. If for some reason, the context is changed to another thread right before the increment, the program will update the "old value". This error can result in nasty bugs, and this is why atomic isn't used for BetterSafe.

3.3 java.util.concurrent.locks

This package provides a framework for locking and waiting conditions, and this framework offers greater flexibility in locks and conditions compared to the built-in synchronization and monitors. These locks allow for mutual exclusion in functions, and were relatively simple to use. To use it, all I had to do was add a lock operation right before a critical section, and an unlock operation after the critical section was done executing.

In BetterSafe, I mainly used this package, more specifically the RentrantLock. This made the BetterThis single lock was all that was needed to make the program thread-safe, and this did a good job at protecting any reads and writes to the array. However, this resulted in a decrease in performance because of the locking and unlocking operations. Only one thread can have access to the array at any one time, which prevents the parallelization of array execution.

3.4 java.lang.invoke.VarHandle

This class provides a strongly-typed reference to a variable, and it also supports various access modes, one of which is a volatile read and write access. Like the package java.util.concurrent.atomic, variable values can be updated atomically. However, the same reason why the atomic package didn't work is the reason

why this package would not work well as well for BetterSafe.

4. BetterSafe Implementation

In this section, we will discuss why BetterSafe is faster than just the pure Synchronized class that was provided. Compared to Synchronized, BetterSafe utilized a Reentrant lock from the java.util.concurrent.locks package to protect critical sections of the code. If we were to use a lower-level class in java.util.concurrent to implement concurrency, it would result in faster performance, but at the same time increase chances of mistakes happening. Because of the use of the locks, it is faster than Synchronized, and it basically guarantees that Bettersafe is data race free (DRF).

Atomic methods in java.util.concurrent.atomic and volatile access classes would not work well for BetterSafe because they are still error prone, and don't guarantee DRF, even though it results in better performance. This eliminates java.lang.invoke.VarHandle as a contender for being used in BetterSafe.

All of this means, using locks is the best way of making BetterSafe DRF. Using the keyword synchronized, like in the Synchronized class, also results in a higher overhead than using locks. This is because Java will put a spin-lock (which consumes more CPU cycles) on an entire method that does some update to the values of the array in the critical section. Reentrant locks however offer finer granularity in where we lock the critical section, thus offering increased performance.

5. Testing

To test the different classes I had implemented, I used unsafeMemory to run the class against various number of threads and swaps. (e.g. *java unsafeMemory Synchronized 8 1000000 6 5 6 3 0 3*). The performance output that was received for everything other than Null is compared relative to the output for Null, and has the measurement of average ns / transaction. This Null is basically considered to be a lower bound in this lab for all the other test cases. For testing environments, I used lnxsrv10 on the UCLA SEASnet servers, running OpenJDK 9. To test the reliability and performance for everything, I varied the number of iterations, but tested with 8 threads on the server.

6. Results

Figure 1 below shows a graph of the result of average ns / transition for each of the StateClasses, with varying amount of threads.

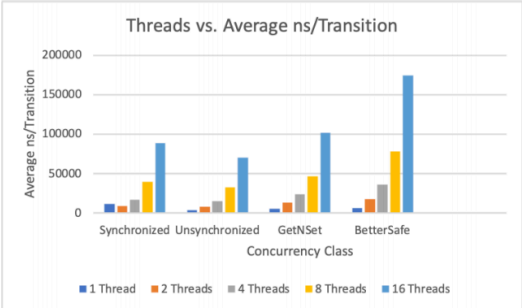

Figure 1: Graph showing the average ns/ transaction for the classes Synchronized, Unsynchronized, GetNSet, and BetterSafe with varying number of threads.

Figure 2 below basically shows what would happen if you change the number of iterations for every Class.

Class	100 it.	1000 it.	10000	100000	1000000
Null	226900	30860	7740	1630	2100
Synchronized	248140	33880	10920	6290	2970
Unsynchronized	208460	32150	Loop	Loop	Loop
GetNSet	249700	43680	Loop	Loop	Loop
BetterSafe	474730	76910	15910	5060	1410

Figure 2: Analysis of increasing the number of iterations for each different State. In this graph, the place where it says "Loop" indicates that there was an infinite loop in the program because of multithreading issues like race conditions.

Now we will cover the features and characteristics of the different classes

6.1 Unsynchronized

Based of what we know about concurrency and synchronization in Java, we know that the Unsynchronized class is not data race free, or DRF. No synchronization attempts are made, with no extra features for concurrency added. Because of this, there is nothing preventing threads from overwriting the same variable in critical sections. Looking at the graph in figure 2, any iterations that didn't have a loop led to results that had gone through a bunch of race conditions, as we got many sum mismatches in the output, indicating that values were changed when they shouldn't have been.

However, because of the lack of any locking or concurrency, the performance was significantly better and the closest to the performance of Null due to this lack of overhead. This doesn't mean

anything in a practical application though because despite good performance, we did not get correct values. All in all, the command *Java UnsafeMemory Unsynchronized 8 1000000 6 5 6 3 0 3* would fail on the SEASnet servers.

6.2 Synchronized

Our Synchronized class is basically DRF. Java basically guaranteed Synchronization with the synchronize keyword, and only allows one thread to access the swap method at any one time, and this is accomplished through a spinlock. This spinlock makes other threads waiting to use the method spin until it becomes available, which prevents any overwriting by any other thread. However, the issue with this class is that Spinlocks and the Synchronization keyword results in lower performance. There is significant overhead with using this, despite the reliability.

6.3 GetNSet

GetNSet doesn't really guarantee DRFs. From our figure 2, we can see that some loops occurred as well as some sum mismatches, meaning some race conditions occurred. This is because GetNSet mainly uses volatile accesses through an AtomicIntegerArray, which doesn't protect against data races in the array. Compared to Unsynchronized, GetNSet performs slightly worse despite lower frequencies of wrong results. It however, does better than Synchronized and BetterSafe because it doesn't use locks, which uses for cycles than atomic accesses.
If we use the command *Java UnsafeMemory GetNSet 8 1000000 6 5 6 3 0 3*, it would fail on the servers.

6.4 BetterSafe

My implementation of BetterSafe was basically DRF. There were no race conditions or sum mismatches in any of the tests I ran. It is almost like Synchronized in the sense that it uses a lock, but the lock in BetterSafe is a Reentrant lock, which is finer in granularity. This Renentrant lock is placed around critical sections with the lock() and unlock() methods. This basically limits only certain parts of the methods to one thread, therefore preventing data races. Because of this, BetterSafe performs better than Synchronized, but at the same time, does not perform as well as GetNSet and Unsynchronized because of the locking. However, in the grand scheme of things, BetterSafe is the best because it is reliable and performs not too badly.

7. Conclusion

In this lab, we explored several ways of implementing concurrency in Java. This included implementations like not using concurrency at all, using atomic accesses, and using locks to lock critical sections. As we saw from our graphs, the classes GetNset (which used atomic volatile accesses) and Unsynchronized (which did no concurrency at all) performed the best. However, they were highly unreliable, as data races occurred frequently, resulting in sum mismatches. On the opposite side, we have Synchronized and BetterSafe. These offered the best reliability and are basically considered DRF. However, there is a tradeoff involved in this, with performance being the cost. Synchronization and BetterSafe uses locks to make the function threadsafe but the process of locking and unlocking takes more cycles than using volatile accesses and no concurrency. As we can see, parallelization is very useful and would benefit GDI greatly.