CS 131 Homework 3 Report

Abstract
The project objective of Homework 3: Java shared memory performance races, was to observe differences in speed performance and accuracy of synchronized and non-synchronized Java classes serving the same purpose, as well as gain a more thorough understanding of the Java Memory Model (JMM) and how to develop performant, data-race free (DRF) programs. Through the process we observed that synchronized Java classes tended to have higher accuracy due to alleviation or elimination of effects from data races, but also tended to be less performant due to extra checks being performed to guarantee synchronization of code blocks and mutual exclusion of data accesses and writes to shared memory locations. On the flip side, we observed that non-synchronized Java classes were far more performant, but much less accurate, both a result of unrestricted access to shared memory locations (which allows data races).

Testing Platform Information
Using the command $java –version, I was able to identify that the current version of java on the SEASnet server, on which I tested my Java project, was openjdk version "1.8.0_161", with OpenJDK Runtime Environment (build 1.8.0_161-b14) and OpenJDK 64-bit Server VM (build 25.161-b14, mixed mode). Further, by checking the system files /proc/cpuinfo and /proc/meminfo, I was able to determine a lot of the system information, which would allow future testers to reproduce my results on similar machines using the same version of Java. For example, using /proc/cpuinfo I was able to determine that the SEASnet server on which I tested has 32 Intel Xeon E5-2640 v2 2.00GHz CPUs each with 8 cores, more precisely clocked at 2.299921 GHz, with cache size 20480 KB in L1, and with 46 bit physical addresses, 48 bit virtual addresses, and 64 cache alignment. Similarly, using /proc/meminfo I was able to find more important identifying data, for example, that the system has approximately 64 GB RAM, the size of kernel stacks being 10160 KB and the page tables are approximately 53384 KB, in addition to less relevant information on the current state of my machine regarding pages mapped, active inodes, swapped pages, and more.

Testing Mechanism
In order to test the different mechanisms of handling concurrency I developed, I ran an integrated version of the given test program UnsafeMemory.java on 1000 swaps, 10,000 swaps, and 1,000,000 swaps, using 1, 8, 16, and 32 threads, and using a byte array of 6, 100, and 600 elements ranging from 0 to 127, the range of an unsigned byte, with each data entry in the table being an average of 5 runs.

Performance and Reliability Results
The results I achieved using the testing mechanism described above can be transcribed in a table format as appears below.

Figure 1. Time for Transition with 1,000,000 swaps and 600 elements

Memory Order Model	Time for transition (ns/transition)				DRF
	1 threads	8 threads	16 threads	32 threads	
Null	43.5553	1431.27	4641.45	7958.04	Y
Synchronized	71.8389	2760.19	5700.96	14611.2	Y
Unsynchronized	54.6855	1387.89	3435.40	8733.15	N
GetNSet	77.4668	1530.07	3264.15	10506.6	N
BetterSafe	74.8322	1423.28	2780.35	6949.57	Y

Figure 2. Time for Transition with 8 threads and 600 elements

Memory Order Model	Time for transition (ns/transition)			DRF
	1000 swaps	10, 000 swaps	1,000,000 swaps	
Null	25359.0	6122.57	1431.27	Y
Synchronized	36678.4	11338.2	2760.19	Y
Unsynchronized	38361.2	9145.48	1387.89	N
GetNSet	44213.3	12716.8	4130.07	N
BetterSafe	57324.2	13149.7	1423.28	Y

Figure 3. Time for Transition with 1,000,000 swaps and 8 threads

Memory Order Model	Time for transition (ns/transition)			DRF
	6 elements	100 elements	600 elements	
Null	2286.09	1910.39	1431.27	Y
Synchronized	2399.98	2989.05	2760.19	Y

Unsynchronized	32003.6	1768.56	1387.89	N
GetNSet	*doesn't complete most of time	1467.02	4130.07	N
BetterSafe	1428.36	1296.87	1423.28	Y

// Compare the classes' reliability and performance here. Which is the best to use our purposes? Meaning we want to achieve the best possible performance while maintaining above threshold reliability, meaning consistent results from running the UnsafeMemory tests provided by Eggert.

Comparisons

From the test data we see that on average across all tests, Unsynchronized runs faster than Synchronized. This is not surprising, since there is no synchronization overhead in the Unsynchronized implementation, and this implementation has a much weaker memory order model. However, because of this, Unsynchronized often has a sum mismatch of a large factor, proving that the lack of synchronization takes a reliability hit, whereas the Synchronized implementation is 100% reliable.

Further, Null is faster than almost all options across the board, except for BetterSafe. Null has faster results because the Null class does not actually work, simply returning True whenever the swap operation is called. Here we expect Null to achieve better performance than any other implementation since it has the weakest protections on memory (the accesses to memory are nonexistent, so there is nothing to protect), intuitively, but with more threads Null tends to underperform BetterSafe while outperforming any other implementations.

GetNSet outperforms Synchronized, but like Unsynchronized, that is because there is no use of the synchronized keyword which has a lot of overhead. However, GetNSet performs worse than Unsynchronized because it does have some synchronization when updating the AtomicIntegerArray data structure using atomic writes and reads to access memory from the data structure. Intuitively, GetNSet should perform worse than Unsynchronized because it has stronger memory protections, but it should perform better than Synchronized, because it trades some reliability by not synchronizing the entire critical section and function for speed performance. It should also perform better than BetterSafe, since it has less memory protections and thus less reliability, but we surprisingly see that this is not always the case. Generally, however, this result holds.

BetterSafe generally outperforms every other implementation, except for Null when the number of threads is low and GetNSet when the number of array elements is large. BetterSafe effectively achieves 100% reliability by never reporting a mismatch in sum, while also doubling the speed of Synchronized by only synchronizing the critical section using finer grained locks, rather locking than the entire function. BetterSafe has stronger memory protections than Unsynchronized and GetNSet, but still tends to outperform them at high thread counts due to its finer grained locking which is optimal for high thread counts. However, GetNSet outperforms BetterSafe with a large number of elements because it uses the AtomicIntegerArray, which uses atomic operations rather than locks and thus carries less overhead when modifying or reading each element. BetterSafe also consistently outperforms Synchronized, since it follows a slightly weaker memory order model by using a finer-grained lock than synchronizing an entire function.

Because BetterSafe has 100% reliability and can sometimes outperform GetNSet, while consistently outperforming Unsynchronized and Synchronized, BetterSafe is the best implementation for GDI.

Analysis of Tools Used

1. java.util.concurrent

At first glance, it appears that the concurrent package may be a proper option to implement BetterSafe, since its name implies proper usage with concurrent programs such as the one we want to implement. However, looking through the immediate packages at this level of the concurrent package hierarchy, we see that all the class definitions define low level programming paradigms like Thread Factories and data structures such as Deques to synchronize. The pros of using such low level data structures and classes is that as a programmer, we would have an extremely tight grasp on the ordering of threads and we would be able to enforce a strong memory order mode with a specific ordering each time, by ranking threads and sorting them onto a Thread Deque so that highest ranked threads run first. However, the cons of having such customizability with data structures is that lots of complexity is added with this level of customization, and it is extremely difficult to coordinate lots of threads in an object method that threads will be running rather than a class that is spawning the threads such as UnsafeMemory. It appears that as a result, the concurrent package is not

the correct package to use to implement BetterSafe, but would potentially be a good package to use for an advanced implementation of UnsafeMemory.

2. java.util.concurrent.atomic

The Java Concurrent atomic package provides a repertoire of classes which do not use locks to provide atomicity of executing critical sections or updating data. The pros to this are that we do not need to worry about locking any object before accessing it, the class will take care of it for us, and if we are modifying multiple data structures within an object each data structure will lock itself to allow accesses, without affecting the other data structures, therefore there is a more fine-grained mutual exclusion mechanism. If we were simply performing reads or writes but not both, this package would be extremely helpful because it would provide us a set of data structures that can perform atomic reads and writes and thus protect each of these individually. However, our critical section for state involves performing a read and potentially a write, therefore these data structures in this package are not enough. An interrupt that occurs in the middle of our semi-protected critical section after the read and before the write could mess up the validity of our result, so this package would not work.

3. java.util.concurrent.locks

The Java Concurrent locks package provides a repertoire of lock classes which allow developers to protect a critical section with more customizability. The cons of this package are that because the locks are extremely customizable, there is a complexity when considering how many different types of locks there are and which lock to use for each use case. Locks are also unfortunate because they lock the entire object, similar to synchronization, which disallows any other thread to modify perhaps say a different shared memory space than the one the lock-holding thread is modifying, even though this action is perfectly safe. However, the pros of the lock package are that it is extremely simple to use and fits our use case perfectly. We want to only lock the critical section of our object method and thus decrease the amount of overhead associated with the synchronized keyword in Java. The Reentrant lock is the main implementation for mutual exclusion, as the API states, and thus this is the API class that we use to implement BetterSafe to ensure 100% reliability, as locking the critical section will give us, while also increasing performance from the Synchronized implementation, which decreasing the synchronized area of code to only the critical section will do.

4. java.lang.invoke.VarHandle

The VarHandle class acts like the AtomicIntegerArray class but generalizes to multiple types besides integers. The pros of this are that we are able to synchronize an array of different objects that aren't just integers, which the byte array may contain, while maintaining atomic operations. However, VarHandle does not work well for our BetterSafe implementation because we are handling just Integers, and VarHandle does not protect the entire critical section, allowing for interrupts to break sequential consistency.

BetterSafe's Improvements from Synchronized

My BetterSafe implementation is faster than the Synchronized option because the entire swap function is not synchronized, but rather only the critical section within the function (the portion performing the reads and writes from shared memory) is protected by a reentrant lock, which was described above in the section analyzing the four packages suggested by Professor Eggert and describing my choice of which packages to utilize in my implementation. This use of the reentrant lock also provides 100% reliability, since the critical section where each thread reads and writes is the only part of code that requires protection from data races, since a write can happen at any point before, after, or even during a read, depending on preemption policy regarding threads and thread scheduling, and locking only the critical section provides a lightweight and reliable counter-option to synchronizing the entire function.

As a result, my implementation of the BetterSafe class achieves a stronger memory order mode than just plain ordering relations, but not quite a total sequential ordering. Instead, the BetterSafe class is able to achieve a state of partial ordering, in which there are many linear extension topological sort paths of possible orderings, but the critical sections of each thread are atomically executed and so all of the possible linear extension topological sort paths will be valid orderings that will not corrupt memory, and each local execution of a critical section will be valid, meaning that if there is an interrupt in the middle of a critical section, the thread that was interrupted will still have a lock and thus no other threads will modify the shared memory the thread is acting on, and thus the completion of the interrupted thread's critical section will be as if the thread were not interrupted at all (the execution is valid).

Challenges

One of the problems I ran into during testing that I had to overcome to get consistent results was increasing the number of elements in the array to see

the effect of larger arrays on the times of transition. It was fairly impossible to set up 500 elements in the command line arguments for the main method call, especially since having more than 8 elements lined up would cause my program problems and print the Usage message error. To fix this, I used a random number generator to generate a large array of set size to complete the tests. This random number generator also allowed random inputs to create a more random distribution of timing results, which made my test data more inclusive of a sample space of tests.

Another problem I ran into was running all the tests to populate my test data tables. Running everything by hand would be extremely toiling, so I was able to write a bash script that would run my program with separate arguments using a for loop to vary each the number of threads, the maxval argument, and the number of elements in the array.

DRF Class Analysis

In my analyses below, I interpret the homework specification instruction "Give a reliability test … that the class is extremely likely to fail" as "Provide a test that the class will not achieve 100% reliability on, and will thus have a sum mismatch error that the UnsafeMemory class will catch and display on the standard output of the terminal".

The Null class is data race free (DRF) because it does not modify any shared memory. It is easy to see by observing the code that shared memory locations between threads are never modified, and it is also easy to show by running many test cases that a mismatch in sum will never occur.

The Synchronized class is DRF because when it modifies shared memory, it locks the entire object at the start of the method and unlocks after the function returns using the synchronized keyword. There are also no tests that the Synchronized class can fail due to its DRF status.

The Unsynchronized class is NOT DRF because it does modify shared memory without any memory protections. There are no locks or atomic operations that occur when an Unsynchronized class object executes the swap method, and thus a test case like `java UnsafeMemory Unsynchronized 32 1000000 5` with a large number of threads, a large number of swaps, and a large number of elements in the array will almost definitely return a sum mismatch when executed on the SEASnet server.

Similarly, the GetNSet class is NOT DRF because its atomic operations do not protect the entire critical section, but instead only protect portions of the critical section. The reads and writes to shared memory are independently protected accesses, therefore when a thread preempts or is interrupted in between a read and a write, there will be a data race condition. A test case with a high number of threads, swaps, and elements in the array will greatly increase the chances of an interrupt occurring during a thread's execution of the semi-protected critical section, and thus a test case like `java UnsafeMemory GetNSet 32 1000000 5` will likely cause a sum mismatch and cause the class to fail on the SEASnet server.

The BetterSafe class is DRF because the entire critical section in each thread is locked. This means that any interruptions such as thread preemptions or thread scheduling side effects will not affect the validity of the execution of the critical section in which each thread modifies a shared piece of memory. This therefore guarantees the reentrancy of the execution of each thread's critical section, and guarantees 100% reliability in regards to data races for each thread reading or updating a BetterSafe implementation of the State interface.

Conclusion

For the reasons mentioned in the above paragraphs, especially at the end of the comparison section and DRF Class Analysis section, GDI should choose to use the BetterSafe State implementation.