**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 10,000,000 swaps, using 1, 2, 4, 8, 16, and 32 threads, and using a byte array of elements ranging from 0 to 127, the range of an unsigned byte.

**Performance and Reliability Results**

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

<Table goes here>

// 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.

**Analysis of Tools Used**

1. [java.util.concurrent](https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/package-summary.html)

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.

1. [java.util.concurrent.atomic](https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/atomic/package-summary.html)

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.

1. [java.util.concurrent.locks](https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/locks/package-summary.html)

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.

1. [java.lang.invoke.VarHandle](https://docs.oracle.com/javase/9/docs/api/java/lang/invoke/VarHandle.html)

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 Increase in Efficiency 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).

// Discuss any problems you had to overcome to do your measurements properly. Explain whether and why the class is DRF; if it is not DRF (due to a bug) give a reliability test (as a shell command "java UnsafeMemory *model* ...") that the class is extremely likely to fail on the SEASnet GNU/Linux servers.

/// size of the array was an issue, kept getting number format exceptions when attempting to create a longer byte array from the command line. So I had to make a separate function?

The class is data race free (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.