# CS 131 Homework 3: Java Shared Memory Performance Races


## Abstract

For Homework 3: Java Shared Memory Performance Races, our primary objective was to examine the trade-offs between performance and reliability in a multithreaded Java application. Specifically, we tested the performance and reliability of four different Java classes that each implement synchronization using a different mechanism. From the tests, we were able to observe that as a Java program has increasing synchronization, it is also increasingly reliable because the synchronization prevents data races (multiple threads concurrently accessing shared memory). However, the increase in synchronization also created extra overhead and caused lower performance in synchronized programs in comparison to unsynchronized programs.


## 1. Testing Platforms

All of the performance and reliability tests for the different Java programs were ran on the SEASNet Linux Server 06, specifically running Red Hat Enterprise Linux 7. After examining the `/proc/cpuinfo` file, I found out that lnxsrv06 uses 16 Intel® Xeon® CPU E5-2640 @ 2.40GhHx processors, each having 4 CPU cores. Additionally, the `/proc/meminfo` file gave me the information that lnxsrv06 has approximately 65.79 GB of total memory.

The only difference between my two testing platforms was the version of Java. For the first platform, I used the default version of Java on lnxsrv06. By running the command `java -version`, I saw that the first testing platform used Java OpenJDK version 11.0.2. Then I prepended `"/usr/local/cs/jdk-9/bin:"` to `PATH`, which allowed me to run Java OpenJDK version 9 for the second testing platform.

## 2. Choosing a BetterSafe Implementation

For `BetterSafe`, my goal was to implement a class that maintains the same 100% reliability as `Synchronized`, but also has an increase in performance. I studied the trade-offs between four different packages and classes to use for my `BetterSafe` implementation and chose `java.util.concurrent.locks` as the optimal implementation.

### 2.1. java.util.concurrent

The `java.util.concurrent` package contains several higher-level synchronization libraries such as `Semaphore`, `Exchanger`, `CountDownLatch`, `CyclicBarrier`, and `Phaser` [1]. Additionally, it contains other mechanisms for concurrent programs such as thread pools, blocking and non-blocking queues, and timers. Obviously, this package contains several tools to optimize synchronous Java programs; however, since these higher-level synchronization libraries allow for longer-term locks, they are considerably less light-weight compared to other synchronization options. Since we just need short term locks to do simple operations (incrementing and decrementing array values), this package adds too much unnecessary overhead and complexity to our simple concurrent Java program.

### 2.2. java.util.concurrent.atomic

The `java.util.concurrent.atomic` package contains classes that do not use traditional locking mechanisms, but instead use atomic operations on common array transactions. For example, in our `GetNSet` class, we use an `AtomicIntegerArray` for our value array, so we ensure volatile access to array elements. Although this ensures atomic reads and writes individually, this form of synchronization is not reliable for our Java program. Specifically, our program first does a check to ensure that the current array element is within the range from `0` to `maxval`. Another thread could modify the array element's value between the check and the current thread's update operation, possibly causing the array value to be outside the valid range. Therefore, this package is not reliable enough to be used in our implementation of `BetterSafe`.

### 2.3. java.util.concurrent.locks

The `java.util.concurrent.locks` package contains several classes such as `ReentrantLock` and `ReadWriteLock` that allow flexible locking mechanisms for concurrent programs. After researching all of the classes, I decided to use the `ReentrantLock` class of this package to implement synchronization in `BetterSafe` because it provides exactly what is necessary for this program: a simple, light-weight form of synchronization that prevents all possible data races in the program. Not to mention, the API for this class is extremely simple to use (`lock()` and `unlock()`). Overall, this form of mutual-exclusion locking protects the program against any possibility of multiple threads accessing shared memory at the same time, while also remaining light-weight and minimizing overhead. However, the simplicity does come at a price because we have the risk of a single thread holding onto the lock for an extended period of time, thus starving all of the other threads.

### 2.4. java.lang.invoke.VarHandle

The `java.lang.invoke.VarHandle` class contains a variable type and a list of coordinate types, as well as several access modes with varying levels of synchronization. Specifically, the access modes fall under five categories: read access, write access, atomic update access, numeric atomic update access, and bitwise atomic update access. The atomic access modes contain the same pros and cons

as the `AtomicIntegerArray` class. One benefit of this class is the fact that it is generic and not restricted to just integers; however, for our purposes, this class contains the same reliability problems as the `AtomicIntegerArray` class, and the performance of the lighter `ReentrantLock` class is still superior.

## 3. Testing Performance and Reliability

For measuring the performance and reliability of the four classes, I ran them with 10,000,000 successful transition swaps, a value array of 15 integers, and a number of threads ranging from 4 to 32. For example, the test case for the upper-left most entry in the first table was ran using the following command: `java UnsafeMemory Null 4 10000000 10 8 4 1 6 0 5 5 9 2 1 9 7 5 4 3`. The transition times (time for a thread to do a successful swap) are shown in the tables below, one for each version of Java.

| Model | Transition Time (ns) | | | | DRF |
|---|---|---|---|---|---|
| | 4 threads | 8 threads | 16 threads | 32 threads | |
| Null | 192.644 | 609.230 | 1195.86 | 3949.29 | Yes |
| Sync | 457.445 | 1146.84 | 2944.63 | 6838.84 | Yes |
| Unsync | 256.488 | 834.045 | 2208.62 | 4173.84 | No |
| GetNSet | 346.068 | 1056.95 | 2516.69 | 5473.43 | No |
| BetterSafe | 395.850 | 1026.04 | 2437.45 | 5827.62 | Yes |

**Table 1: Transition Times for 10,000,000 Transitions Using Java 11.0.2**

| Model | Transition Time (ns) | | | | DRF |
|---|---|---|---|---|---|
| | 4 threads | 8 threads | 16 threads | 32 threads | |
| Null | 144.814 | 372.436 | 944.683 | 3448.82 | Yes |
| Sync | 604.038 | 1063.62 | 2346.83 | 4746.16 | Yes |
| Unsync | 322.603 | 641.250 | 1232.67 | 3600.04 | No |
| GetNSet | 505.848 | 904.572 | 2092.96 | 4693.67 | No |
| BetterSafe | 437.899 | 901.630 | 1842.26 | 4608.49 | Yes |

**Table 2: Transition Times for 10,000,000 Transitions Using Java 9**

In terms of the testing environments, the results followed extremely similar trends except the fact that `BetterSafe` was always faster than `GetNSet` using Java 9, but `GetNSet` slightly outperformed `BetterSafe` in Java 11.0.2. Also, the times for Java 9 turned out to be faster than Java 11.0.2. This came as a surprise because I was expecting the newer version to be more efficient. One possible explanation could be the fact that there was greater server traffic when I was collecting my Java 11.0.2 data. Additionally, if any of the classes failed any of the tests (with a sum mismatch or infinite loop), they were indicated as not DRF (data-race free) in the tables, and they were indicated DRF otherwise. Overall, both versions of Java showed the same trends in the data, which we will now examine.

### 3.1. Synchronized

The `Synchronized` class is implemented by simply using the `synchronized` keyword in the `swap` method, which allows only one thread to enter this synchronized method at a time, thus ensuring that the program is data-race free. This keyword also creates a "happens-before relationship" in which "everything that one thread did while in a synchronized block will be visible to the next thread entering a synchronized block" [2]. Therefore, threads never access any stale values in this class, so it has 100% reliability. However, the keyword also slows down the program significantly because there cannot be any concurrent updates to the array. This is reflected in the data because as shown in Tables 1 and 2, `Synchronized` is DRF, but it also has the worst performance of all of the classes.

### 3.2. Unsynchronized

The `Unsynchronized` class has the exact same implementation as the `Synchronized` class, minus the `synchronized` keyword. Obviously, `Unsynchronized` performed extremely fast, the fastest of all the classes besides `Null`, because there are no restrictions on when threads could execute. However, without any synchronization mechanisms, this class is extremely susceptible to data races. For example, another thread could change a value between the check and the actual array update. Also, another thread could execute in between the time that a certain thread loads a value and the time that the thread stores the updated value. Therefore, `Unsynchronized` is extremely fast, but not DRF at all due to its lack of synchronization.

### 3.3. GetNSet

The `GetNSet` class uses an `AtomicIntegerArray` as the value array, which ensures that all array operations are atomic. This prevents certain conditions for data races such as when a second thread updates a value in between the loading and storing operations of another thread. However, this still does not fix data races such as a second thread executing after the bounds check, but before the array value update for a thread; therefore, `GetNSet` is not 100% reliable and thus not DRF. Also, its performance is a little faster than `Synchronized` due to the fact that it only synchronizes the individual array operations, not the entire method.

### 3.4. BetterSafe

The `BetterSafe` class utilizes the `ReentrantLock` class to synchronize only the critical section of the `swap` method, instead of the entire `swap` method like in `Synchronized`. This class locks the `ReentrantLock` right

before checking the values of the array, and then it unlocks the lock right after incrementing and decrementing the values. This locking mechanism ensures that `BetterSafe` is 100% reliable because when a thread gains the lock, no other threads can access the critical section until that thread gives up the lock; therefore, no threads can currently access shared memory, so `BetterSafe` is DRF. Additionally, since `BetterSafe` synchronizes only the critical section, it performs better than `Synchronized`, but still worse than `Unsynchronized` because `Unsynchronized` has no synchronization to slow it down.

## 4. Comparing BetterSafe to Synchronized

Based solely on Tables 1 and 2, my `BetterSafe` implementation outperforms `Synchronized`; both classes are DRF, but `BetterSafe` has consistently faster transition times. So based on the data, `BetterSafe` is faster.

First off, `Synchronized` is DRF because the `synchronized` keyword only allows one thread to execute `swap` at a time, which definitely removes any possibility of data races and makes it DRF. Additionally, `BetterSafe` uses a `ReentrantLock` to only allow one thread to execute the critical section of `swap` at a time, thus removing data races and making `BetterSafe` DRF as well. Therefore, both classes are 100% reliable.

However, my implementation for `BetterSafe` outperforms `Synchronized` in terms of transition times because `BetterSafe` only synchronizes the critical section and allows everything else to be executed concurrently, while `Synchronized` synchronizes the entire method. Therefore, `BetterSafe` is superior because it maintains the 100% reliability of `Synchronized` and provides a lighter, faster implementation that has less overhead from synchronization.

## 5. Problems with Measuring Performance

My main problems with measuring performance arose from the non-DRF classes, `Unsynchronized` and `GetNSet`. Since these classes have data races, it was possible for another thread to step in and change the array value after a thread's check, but before its array operations. This made it possible for every value in the array to either become non-positive or greater than or equal to `maxval`. If this occurred, then swap always failed, thus resulting in an infinite loop. In these cases, I had to terminate the running process and try again. Additionally, there was a lot of traffic on the Linux server during testing which could have resulted in slow or inaccurate measurements.

## 6. Conclusion

After analyzing the performance of all of the different classes, we can now see that `BetterSafe` has the most advantages of all of the classes. In terms of reliability, there are only two 100% reliable classes, `Synchronized` and `BetterSafe`, due to previously stated reasons. Among these two, `BetterSafe` is more efficient because its `ReentrantLock` implementation is lighter and contains less overhead than the `synchronized` keyword used in the `Synchronized` class. Therefore, `BetterSafe` is the optimal class to use for GDI Java programs.

## References


[1] Java SE 11 and JDK 11 Package java.util.concurrentAPI: https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/package-summary.html

[2] TA Kimmo Karkkainen's Slides on The Java Memory Model: https://piazza.com/ucla/spring2019/cs131/resources