# Java Shared Memory Performance Races

CS 131 – Fall 2016

## Testing Platform

- Java (version 1.8.0_9)
- Java(TM) SE Runtime Environment (build 1.8.0_92-b14)
- Java HotSpot(TM) 64-Bit Server VM (build 25.92-b14, mixed mode)
- Intel(R) Xeon(R) CPU E5620 @ 2.40GHz (4 cores, 8 threads)
- 32 GB of RAM

## BetterSafe vs. Synchronized

My implementation of BetterSafe uses ReentrantLock to ensure 100% reliability. ReentrantLocks have the property of performing better under contention while retaining all of the safety of synchronized. ReentrantLocks are faster and more scalable than synchronized because of its lock polling and interruptible lock waits. Synchronized threads cannot poll, interrupt another waiting thread, or timeout after waiting for a certain period of time, while ReentrantLock can.

## BetterSorry

BetterSorry is implemented using an array of AtomicIntegers (not AtomicIntegerArray),

### 1.1. Faster than BetterSafe

BetterSorry is faster than BetterSafe because BetterSorry does not use any locking mechanism. This means that BetterSorry does not have any overhead that locks do, and that BetterSorry's threads will not wait as long as BetterSafe's threads. The increment and decrement of the race condition variables is volatile, but this does not mean it is race condition free, it means that each memory access to the variable must not be optimized away by the compiler.

### 1.2. More Reliable than Unsynchronized

BetterSorry is more reliable than Unsynchronized because it utilizes volatile behavior. Volatile guarantees that any changes made to a volatile variable are always visible to other threads. Any threads reading the variable will be able to see its most recent change, which allows BetterSorry to better avoid the infinite loop that Unsynchronized faces. This infinite loop can happen in Unsynchronized when multiple threads decrement or increment the same variable at the same time, putting the variable out of bounds (unable to increment or decrement it forever). This makes all later reads to that variable wait indefinitely.

It is possible to make BetterSorry enter the infinite loop. You can do this by setting the maxval to 1 and all of the five initial entries to 1. This forces `value[j]` to be equal to `maxval`, and all threads will always return false, never allowing a swap to happen. This can also be done by setting the five initial entries to be 0.

The program to run in order to cause the infinite loop is the following:

```
$ java UnsafeMemory BetterSorry 1 1 1 1 1 1 1
```

## All Classes

### 2.1. Synchronized

Synchronized is guaranteed as a built-in keyword to be data-race free (DRF) and deadlock free. This means that only one thread can be executing swap at any given time, leading to poor scalability with large number of swaps.

### 2.2. Unynchronized

Unsynchronized is faster than Synchronized because there are is no safety overhead, but suffers from an infinite loop if any values become out of bounds. The chance of looping infinitely increases as the number of threads increases, number of swaps increases, maxval decreases, and initial values are set to numbers that are close to maxval or 0. The chance of this happening was observed to be negligible for less than 10,00 swaps, 20% for 100,000 swaps, and 60% for 1,000,000 swaps (for 2 or more threads).

### 2.3. GetNSet

GetNSet is implemented using AtomicIntegerArray, which guarantees volatile access to values. GetNSet is not DRF and is vulnerable to infinitely looping because there is no guarantee of exclusive access to the variables. The chance of looping infinitely was observed to be negligible for less than 100,000 swaps, 43% for 100,000 swaps, and 80% for 1,000,000 swaps.

### 2.4. BetterSafe

BetterSafe is implemented using ReentrantLocks, which is proven to be more scalable than Synchronized while retaining all safety characteristics. It is DRF, deadlock free, and does not ever enter the infinite loop. BetterSafe was observed to perform faster for swaps over 100,000, increasing in likelihood as the number of swaps increases (nearly 100% for 1,000,000 swaps).

### 2.5 BetterSorry

BetterSorry is implemented using an array of AtomicIntegers along with the getAndIncrement() and getAndDecrement() methods. This always performs faster than BetterSafe for swaps less than 100,000, but is slower for 1,000,000 swaps. BetterSorry is not DRF.

## Results

All tests are run with maxval as 127 and each of the 5 initial entries as 25. The reason for this is to minimize the chance of entering the infinite loop by evenly distributing the total sum over all the variables. Every combination of thread count (1, 2, 4, 8, 16, 32), swaps (100, 1,000, 10,000, 100,000, 1,000,000) is run on each class five times, and the mean of the transition time is taken for each thread count. All reported values are of unit ns/transition. Any reported values of 0 means an infinite loop was entered. If the class is not DRF, an example shell command that is likely to loop infinitely is provided below it.

| Null | | Threads | | | | | |
|---|---|---|---|---|---|---|---|
| | | **1** | **2** | **4** | **8** | **16** | **32** |
| Swaps | **100** | 28853.54 | 54522.5 | 121971 | 248350.4 | 763489.4 | 2271684 |
| | **1,000** | 4430.91 | 8239.9 | 18533.22 | 34876.14 | 79475 | 253864 |
| | **10,000** | 749.8884 | 1684.166 | 3780.22 | 7384.726 | 16323.94 | 36749.3 |
| | **100,000** | 207.8832 | 527.863 | 1037.2426 | 1313.122 | 2729.912 | 7812.028 |
| | **1,000,000** | 42.18246 | 107.8127 | 341.3864 | 1782.774 | 3831.254 | 7257.532 |

| Synchronized | | Threads | | | | | |
|---|---|---|---|---|---|---|---|
| | | **1** | **2** | **4** | **8** | **16** | **32** |
| Swaps | **100** | 27517.8 | 53589.68 | 114247.8 | 265507 | 789262.6 | 2400660 |
| | **1,000** | 4441.948 | 9010.06 | 16410.24 | 31837.8 | 78332.62 | 253864 |
| | **10,000** | 734.8954 | 1892.274 | 4939.464 | 9456.238 | 18867.12 | 40537.86 |
| | **100,000** | 210.1898 | 660.9526 | 2278.372 | 4489.976 | 8303.444 | 15483.74 |
| | **1,000,000** | 63.34434 | 380.5606 | 1365.652 | 3120.152 | 5316.202 | 8543.648 |

| Unsynchronized | | Threads | | | | | |
|---|---|---|---|---|---|---|---|
| | | **1** | **2** | **4** | **8** | **16** | **32** |
| Swaps | **100** | 28786.72 | 53579.4 | 123232 | 249653.4 | 809725.8 | 2166648 |
| | **1,000** | 4311.662 | 8785.026 | 16819.22 | 33536.98 | 85078.08 | 255960.6 |
| | **10,000** | 754.6728 | 1892.274 | 4939.464 | 9456.238 | 18867.12 | 40537.86 |
| | **100,000** | 213.4112 | 461.2558 | 1035.5235 | 1626.43 | 3380.17 | 6938.77333 |
| | **1,000,000** | 49.70484 | 0 | 728.078 | 1669.27 | 4783.47 | 9261.59 |

```
$ java UnsafeMemory Unsynchronized 32 1000000 127 25 25 25 25 25
```

| GetNSet | Threads |
|---|---|

| | | **1** | **2** | **4** | **8** | **16** | **32** |
|---|---|---|---|---|---|---|---|
| Swaps | **100** | 32770.36 | 64273.66 | 128176.2 | 308381.4 | 819138 | 2699656 |
| | **1,000** | 5518.98 | 11392.28 | 21892.02 | 38377.72 | 107575.04 | 265541.8 |
| | **10,000** | 959.034 | 3046.862 | 5820.154 | 12573.54 | 28767.8 | 59960.06 |
| | **100,000** | 247.5492 | 652.95575 | 2195.07 | 0 | 9243.55 | 28688.2 |
| | **1,000,000** | 66.27978 | 0 | 0 | 0 | 0 | 0 |

```
$ java UnsafeMemo GetNSet 32 1000000 127 25 25 25 25 25
```

| BetterSafe | | Threads | | | | | |
|---|---|---|---|---|---|---|---|
| | | **1** | **2** | **4** | **8** | **16** | **32** |
| Swaps | **100** | 30647.02 | 646248.06 | 167991 | 386853 | 1023911.8 | 2878208 |
| | **1,000** | 5835.436 | 12007.76 | 27957.64 | 57699.7 | 128612 | 316776.4 |
| | **10,000** | 1023.388 | 3027.838 | 6106.792 | 12266.74 | 26737.3 | 54843.5 |
| | **100,000** | 240.5926 | 1165.5144 | 2211.208 | 4284.948 | 7983.65 | 16856.38 |
| | **1,000,000** | 69.09506 | 627.0114 | 680.1482 | 1313.838 | 2762.134 | 5976.1 |

| BetterSorry | | Threads | | | | | |
|---|---|---|---|---|---|---|---|
| | | **1** | **2** | **4** | **8** | **16** | **32** |
| Swaps | **100** | 29729.42 | 53497 | 109485.12 | 259387 | 708584.2 | 2147244 |
| | **1,000** | 4772.068 | 8861.77 | 19849.18 | 37289.8 | 108005.52 | 256953 |
| | **10,000** | 793.501 | 2159.062 | 4080.088 | 8801.13 | 17237.52 | 38716.64 |
| | **100,000** | 228.5002 | 742.1348 | 1242.934 | 1862.938 | 3092.722 | 9462.336 |
| | **1,000,000** | 62.56722 | 248.1914 | 866.2644 | 1375.3012 | 3778.648 | 8474.21 |

```
$ java UnsafeMemo BetterSorry 32 10000000 127 25 25 25 25 25
```

Based on the results, GDI should use BetterSorry, since there is some allowance for error and speed is a priority.

In order to run all of these tests efficiently, I had to write multiple bash scripts that would run test each combination of class, threads, and swaps, and then parse the results to be used in Excel to calculate average transition times.