# CS131 Homework 3 Report

Shreya Raman, *UCLA*

**Abstract**

The purpose of this project was to assess different ways to prevent data race conditions. The three methods used were synchronization, no synchronization, and a Reentrant lock from java.util.concurrent.locks.

**1. Testing Platform**

I used two UCLA SEASnet servers to test my code: lnxsrv09 and lnxsrv10. Both servers operated with Java 13.0.1, and I confirmed this with the command `java –version`. I inspected the files `/proc/cpuinfo` and `/proc/meminfo` to gather statistics.

Server9 had 32 Intel Xeon E5-2640 v2 2.00GHz CPUs each with 8 cores, with a cache size of 20480 KB, and 46 bit physical address, 48 bit virtual address, and 64 cache alignment. The server memTotal was 65755732 kB and memFree was 34038480. The size of the kernel stacks was 23600 kB and the page tables were 102376 kB.

Server10 had 4 Intel Xeon Silver 4116 2.10GHz CPUs each with 4 cores, with a cache size of 16896 KB, and 44 bit physical address, 48 bit virtual address, and 64 cache alignment. The server memTotal was 65799636 kB and memFree was 733216 kB. The size of the kernel stacks was 4480 kB and the page tables were 19376 kB.

**2. Packages**

The packages given to us are each described below.

**2.1 java.util.concurrent**

This package has allows for more freedom in implementation but seemed to be more complex than necessary for this assignment. I did not use this package in my implementation of AcmeSafe.

**2.2 java.util.concurrent.atomic**

This package allows classes to update values atomically, but introduces the possibility of race conditions because of the mis-ordering of operations. I did not use this package in my implementation of AcmeSafe.

**2.3 java.util.concurrent.locks**

This package allows classes to lock threads and wait for conditions before unlocking. There was more flexibility with this package and I decided to use this in my implementation of AcmeSafe.

**2.4 java.util.concurrent.atomic.AtomicIntegerArray**

This package is part of java.util.concurrent.atomic and has the same pros and cons mentioned in section 2.2. I did not use this in my implementation of AcmeSafe.

**2.5 java.lang.invoke.VarHandle**

This package allows us to create variables that can atomically interact with methods. However, there is still the issue of race conditions so I did not use this in my implementation of AcmeSafe.

**3. Testing**

To compare different concurrency methods, I ran the methods on both server9 and server10 with different values for thread numbers, swap numbers, and items in the arrays. The arrangements are listed below:

(1) 100 swaps; 5 items; server9 (Figure 1)
(2) 100 swaps; 5 items; server10 (Figure 2)
(3) 10,000 swaps; 5 items; server9 (Figure 3)
(4) 10,000 swaps; 5 items; server10 (Figure 4)
(5) 1,000,000 swaps; 5 items; server9 (Figure 5)
(6) 1,000,000 swaps; 5 items; server10 (Figure 6)
(7) 1,000,000 swaps; 10 items; server9 (Figure 7)
(8) 1,000,000 swaps; 10 items; server10 (Figure 8)

**4. Results**

The entries in the tables below show the threads average Ns/transition count.

**Figure 1: Server: 09, Swaps: 100, Array Size: 5**

| State Class | 8 Threads | 15 Threads | 40 threads |
|---|---|---|---|
| *Null* | 313641 | 2.819e+06 | 7.7003e+06 |
| *Synchronized* | 296891 | 878305 | 1.5074e+06 |
| *Unsynchronized* | 296524 (mismatch) | 828801 | 1.6623e+06 |
| *AcmeSafe* | 396086 | 431945 | 1.8008e+06 |

**Figure 2: Server: 10, Swaps: 100, Array Size: 5**

| State Class | 8 Threads | 15 Threads | 40 threads |
|---|---|---|---|
| *Null* | 195575 | 543770 | 2.84631e+06 |
| *Synchronized* | 210048 | 552141 | 2.67398e+06 |
| *Unsynchronized* | 178327 | 518059 | 2.47933e+06 |
| *AcmeSafe* | 220416 | 533210 | 2.68770e+06 |

**Figure 3: Server: 09, Swaps: 10,000, Array Size: 5**

| State Class | 8 Threads | 15 Threads | 40 threads |
|---|---|---|---|
| *Null* | 6404.94 | 11441.3 | 31095.7 |
| *Synchronized* | 8359.94 | 15562.3 | 43975.9 |
| *Unsynchronized* | Timeout | 13817.7 (mismatch) | 37631.9 |
| *AcmeSafe* | 11036.6 | 20013.8 | 51977.8 |

**Figure 4: Server: 10, Swaps: 10,000, Array Size: 5**

| State Class | 8 Threads | 15 Threads | 40 threads |
|---|---|---|---|
| *Null* | 5029.15 | 11047.3 | 42296.8 |
| *Synchronized* | 9818.61 | 19382.2 | 85310.0 |
| *Unsynchronized* | 8343.99 (mismatch) | 19083.7 (mismatch) | 64577.2 |
| *AcmeSafe* | 8558.08 | 19057.2 | 75039.2 |

**Figure 5: Server: 09, Swaps: 1,000,000, Array Size: 5**

| State Class | 8 Threads | 15 Threads | 40 threads |
|---|---|---|---|
| *Null* | 2406 | 4130.39 | 8611.21 |
| *Synchronized* | 2612.43 | 5247.71 | 17303.1 |
| *Unsynchronized* | Timeout | Timeout | Timeout |
| *AcmeSafe* | 1231.89 | 2488.20 | 7345.41 |

**Figure 6: Server: 10, Swaps: 1,000,000, Array Size: 5**

| State Class | 8 Threads | 15 Threads | 40 threads |
|---|---|---|---|
| *Null* | 1234.08 | 4252.40 | 3785.72 |
| *Synchronized* | 1423.08 | 2037.89 | 9173.69 |
| *Unsynchronized* | Timeout | Timeout | Timeout |
| *AcmeSafe* | 873.368 | 1577.95 | 3614.55 |

**Figure 7: Server: 09, Swaps: 1,000,000, Array Size: 10**

| State Class | 8 Threads | 15 Threads | 40 threads |
|---|---|---|---|
| *Null* | 1771.48 | 4372.26 | 6488.17 |
| *Synchronized* | 2995.29 | 5411.26 | 13840.5 |
| *Unsynchronized* | 1487.24 (mismatch) | 5000.83 (mismatch) | 8144.02 (mismatch) |
| *AcmeSafe* | 1322.65 | 2609.60 | 7527.84 |

**Figure 8: Server: 10, Swaps: 1,000,000, Array Size: 10**

| State Class | 8 Threads | 15 Threads | 40 threads |
|---|---|---|---|
| *Null* | 693.725 | 1700.98 | 4467.63 |
| *Synchronized* | 1441.09 | 2798.87 | 5798.95 |
| *Unsynchronized* | 1547.37 (mismatch) | 2992.24 (mismatch) | 5132.59 (mismatch) |
| *AcmeSafe* | 754.705 | 1800.71 | 4463.85 |

## 5. Analysis

Sometimes AcmeSafe ran slower than Synchronized and Unsychronized, but it was overall better in terms of speed and safety. Unsynchronized often resulted in a timeout or a mismatch of sums, and AcmeSafe never had this result. It is likely that Unsynchronized had this problem because there was the possibility of race conditions. Additionally, AcmeSafe uses the lock in the critical section of the program and is Data Race Free.

I saw negligible difference in my results between server09 and server10. There was also negligible difference when doubling the array size. However, the highest array size I tested was 10 which is not large at all. If I had used a larger size, I may have seen different results.

## 6. Challenges

I didn't run into too many challenges in this project. The biggest step was to look through the given documentation and figure out the best way to implement AcmeSafe.

## 7. Conclusion

GDI should implement AcmeSafe because of its reliability and scalability in comparison to Synchronized and Unsynchronized. There were no race conditions and is more scalable than Synchronized.