# CS 131: HW 3 Report

Fulati Aizihaer
*University of California, Los Angeles*

## Abstract

This report evaluates the performance and reliability of synchronization in Java, which is based on Java memory model (JMM). We analyze four different approaches: SynchronizedState, UnsynchronizedState, AcmeSafeState, and NullState, tested across different thread models (platform and virtual), array sizes (5 and 100), and thread counts (1, 8, 40). The performance is measured using real execution time and average swap time, which provides insight into how each of these classes performs and how they compare in speed and reliability under different conditions.

## 1 Testing Platform

The program operates under Java 23 and all of the testing were performed on the SEASnet GNU/Linux servers `lnxsrv13` with Java version 23.0.2. Additionally, I documented all of the statistics about my testing platform, so that others can reproduce these results utilizing similar hardware.

I first began by verifying the correct path starts with "/usr/local/cs/bin:" and my Java version using:

```
echo $PATH
java -version
```

Java version outputted:

```
openjdk version "23.0.2" 2025-01-21
OpenJDK Runtime Environment (build 23.0.2+7-58)
OpenJDK 64-Bit Server VM (build 23.0.2+7-58,
mixed mode, sharing)
```

To gather hardware specifications, I used:

```
cat /proc/cpuinfo
cat /proc/meminfo
```

The machine has an **Intel(R) Xeon(R) Silver 4116 CPU @ 2.10GHz** with 8 cores and 32 GB RAM (32534256 kB).

## 2 Implementation

**SynchronizedState:** Has 100% reliability, but it is the slowest implementation overall. It is the safest option to choose for best correctness.

**UnsynchronizedState:** Identical to SynchronizedState, except it doesn't use the "synchronized" keyword in its implementation. It is significantly faster than Synchronized, but it becomes unreliable as it fails frequently for tests with 8+ threads.

**AcmeSafeState:** Uses the `AtomicLongArray` class from the `java.util.concurrent.atomic`, which I found while reading the Java documentation. To implement the `current()` method, I converted the `AtomicLongArray` into a regular `long[]` by looping through the elements. For the `swap()` method, I used `getAndDecrement()` and `getAndIncrement()` to perform atomic swaps instead of using `value[i]-` and `value[j]++`. The AcmeSafeState class is both correct and it is faster than SynchronizedState.

**NullState:** Performs no operations, however it is good to confirm the correctness by passing the tests.

## 3 Testing

Each class was tested using the variation of the command:

```
time timeout 3600 java UnsafeMemory [Thread Type]
[Array Size] [Class] [Thread Count] 100000000
```

Where below are the corresponding inputs:

- Thread Types: Platform, Virtual
- Array Sizes: 5, 100
- Thread Counts: 1, 8, 40

While testing, I initially encountered an error in `UnsafeMemory.java`, where it checked `args[1]` instead of `args[2]` when selecting the class. After fixing the issue, the testing worked fine.

## 4 Comparison

The tables below contain the average swap time it took for different threads and the different array sizes for each implementation.

### 4.1 SynchronizedState Performance (avg ns)

| Threads | Array Size 5 | Array Size 100 |
|---|---|---|
| 1 | 34.99 | 33.54 |
| 8 | 494.01 | 484.80 |
| 40 | 2095.25 | 2337.27 |

### 4.2 UnsynchronizedState Performance (avg ns)

| Threads | Array Size 5 | Array Size 100 |
|---|---|---|
| 1 | 14.53 | 14.89 |
| 8 | 216.70 | 290.69 |
| 40 | 1056.75 | 1291.28 |

### 4.3 AcmeSafeState Performance (avg ns)

| Threads | Array Size 5 | Array Size 100 |
|---|---|---|
| 1 | 25.38 | 25.26 |
| 8 | 1072.11 | 373.52 |
| 40 | 5280.79 | 1793.78 |

## 5 Analysis

**Platform vs Virtual Threads:** Platform threads performed better than virtual threads under higher thread counts, where virtual threads virtual threads performed similarly at lower thread counts.

**Array Size (5 vs 100):** Using larger array sizes (100) slightly improved performance for synchronized and atomic implementations, but the difference was small.

**Thread Count (1 vs 8 vs 40):**

- **1 thread:** All implementations were fast and produced correct results.
- **8 threads:** Performance decreased overall and Unsynchronized began producing incorrect results.
- **40 threads:** Performance decreased further, and Unsynchronized continued to produce incorrect results.

**Class Comparison:**

- **SynchronizedState** The most reliable but slowest, especially under high thread counts. Best for correctness only.
- **UnsynchronizedState** The fastest for single-threaded execution but failed correctness with 8 or more threads.
- **AcmeSafeState** The best of both performance and reliability, achieving correctness with better performance than synchronized, especially at multiple thread counts.
- **NullState** Useful as for testing, since it was always fast and reliable because it doesn't do anything.

## 6 Conclusion

In conclusion, `SynchronizedState` guarantees the correctness but performs the slowest, especially with many threads. `UnsynchronizedState` is significantly faster in speed when using a single thread, but correctness decreases with multiple threads. `AcmeSafeState` offers both correctness and speed, while outperforming `SynchronizedState`. Across all implementations, using a single thread consistently gave the best results in terms of both speed and correctness.

## References

Java Platform SE 25 - Class AtomicLongArray. https://download.java.net/java/early_access/jdk25/docs/api/java.base/java/util/concurrent/atomic/AtomicLongArray.html