CS131 Homework 3 Report

MELODY MYAE

May 10, 2025

1 Introduction

This report documents the process and findings of evaluating various synchronization strategies in Java using a multithreaded simulation test. The goal was to analyze the trade-offs between performance and correctness when using different synchronization mechanisms. Specifically, I evaluated three implementations:

- **NullState**: A placeholder that performs no operations, used to measure scaffolding overhead.
- **SynchronizedState**: Uses Java's `synchronized` keyword to ensure thread safety.
- **UnsynchronizedState**: Introduces data races by removing synchronization for potential performance gains.

Each implementation was tested using the `UnsafeMemory` driver, which simulates a series of increment/decrement operations on a shared state array and verifies the result. By varying the thread type (platform vs. virtual), array size, and thread count, we measured both performance and reliability.

2 Challenges Encountered

Before beginning the actual measurements, I encountered issues uploading my Java files to the SEASNet server. Attempts to use `scp` repeatedly failed without clear error messages. After investigation, I discovered that my disk quota had been exceeded. I resolved the issue by identifying and removing large unnecessary files from my home directory. Once sufficient space was freed, the upload succeeded and the compilation proceeded without issue.

3 Server Information

All experiments were conducted on `lnxsrv14.seas.ucla.edu`, ensuring consistent hardware conditions. The environment details are as follows:

- Java version: `java 23.0.2`
- CPU: Intel(R) Xeon(R) Silver 4116 CPU @ 2.10GHz (2 cores)

4 Testing Methodology

Performance was measured using a combination of Java's internal timing and the Linux `time` command. For each test case, the following metrics were recorded: Total real time, Average swap time, Linux `time` output.

The following parameters were varied:

- **Thread type**: Platform vs. Virtual threads
- **Thread count**: 1, 8, and 40
- **State Implementations**: Null, Synchronized, Unsynchronized
- **State array size**: 5 and 100

Each test was run with a fixed number of transitions (`100000000`), evenly divided among threads. The same command format was used throughout:

```
$ time timeout 3600 java UnsafeMemory Thread_type \
State_array_size State_implementation Thread_count \
Transitions
```

5 Data Analysis

5.1 Performance Trends

As shown in Figure 1, **NullState** showed the lowest execution times as expected, since it performs no actual work.

SynchronizedState performed the worst out of the 3 thread types which is reasonable since the threads can only do work one after another.

UnsynchronizedState provided the highest raw speed at lower thread counts. At 1 and 8 threads, it outperformed `SynchronizedState`, particularly for small arrays. However, this came at the cost of correctness. Errors began appearing with 8 threads and worsened significantly at 40 threads, with mismatch values as high as -465100.

5.2 Correctness and Reliability

`SynchronizedState` was reliable under all configurations, consistently producing a final output sum of zero. In contrast, `UnsynchronizedState` frequently failed the

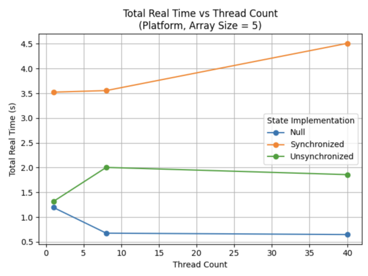

Figure 1: Thread Count vs Total Real Time (s) for array size 5

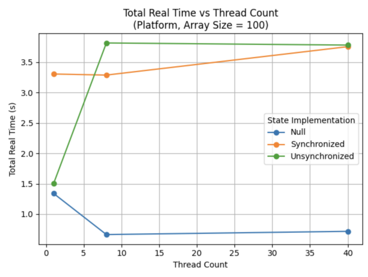

Figure 2: Threadt Count vs Total Real Time (s) for array size 100

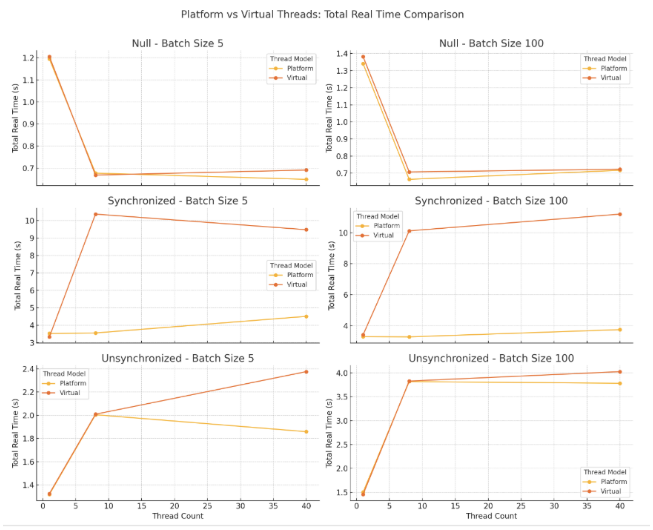

Figure 3: Thread Type Comparison

correctness check under multithreaded conditions, especially with higher thread counts. The magnitude of output mismatch increased with thread count, indicating heightened data races.

Increasing thread count generally improved performance from 1 to 8 threads across all implementations. However, for `SynchronizedState`, going from 8 to 40 threads led to worse performance due to lock contention. `UnsynchronizedState` improved in speed initially but became unreliable and volatile beyond 8 threads.

5.3 Thread Type Comparison

As show in Figure 3, Platform threads consistently performed better than virtual threads under synchronization.

5.4 Array Size Impact

Larger array sizes reduced contention slightly, but did not significantly alter the correctness trend. Execution time increased marginally with larger arrays across all implementations, as expected. However, `UnsynchronizedState` still failed correctness at high thread counts even with larger arrays.

6 Conclusion

The analysis shows a clear trade-off between speed and correctness:

- `NullState` is fast but meaningless, used only to measure overhead.
- `SynchronizedState` is slower but always correct. It performs best with 8 threads and array size 5 on platform threads.
- `UnsynchronizedState` is fast at low thread counts but unreliable at higher ones, especially with virtual threads.

Overall, the best configuration for correctness and good performance was `SynchronizedState` with 8 platform threads. `UnsynchronizedState` only offers value in scenarios where small correctness errors are tolerable and speed is critical. So its optimal use case is with a single thread so that there will be no mismatch error.