CS131 Homework 3:
Java Shared Memory Performance Report

I. Overview

In this project, I tested the performance of several classes with different methods of protection from multithreaded race conditions for Ginormous Data Inc. This imaginary company was seeking a balance between having speed and "close enough" calculations. Through this project, I was able to observe race conditions, bottlenecks, and the tradeoffs between performance and reliability.

II. Implementation

The main idea of the classes was to maintain an array that supported a swap() function, in which two indices were specified. The first would have its entry incremented by one, and the second decremented. Therefore at the end of any number of swap() calls, the sum of all array entries should remain the same.

I started with the given Synchronized and Null classes, and based my Unsynchronized and AcmeSafe on those. All four implemented the State class, which had a constructor, size() to calculate array length, current() to return the array of longs, and swap() between two indices.

My Unsynchronized implementation was nearly identical to that of Synchronized; I simply removed the "synchronized" keyword and renamed the class definition. This meant that race conditions would intentionally be left unprotected and would cause issues in multithreaded tests, but would allow GDI to obtain faster calculations.

My AcmeSafe implementation utilized Java's AtomicLongArray library from java.util.concurrent. This ensured reliability, since the class does read-update-write in a single atomic operation therefore guaranteeing thread safety. When implementing the swap() method, I utilized the class methods getAndIncrement and getAndDecrement to update the array atomically. I also rewrote the current() function to traverse the AtomicLongArray and properly compile the array of longs.

III. Data and Analysis

Each test case outputted seven times, including total time (real and CPU) and average swap time (real and CPU). The focus of this assignment was on the swap() method under different threading parameters, so I chose to focus on the swap times. Below we have the data for average swap time (real) in nanoseconds.

I conducted a test on every combination of the four classes (Unsynchronized, AcmeSafe, Synchronized, Null), array sizes (5, 100, 1000), and thread count (1, 8, 20, 40) on two different lnxsrv machines.

Unsynchronized, lnxsrv10:

	Threads			
Array Size	1	8	20	40
5	12.1351	155.952	748.399	842.389
100	12.2235	285.876	727.650	1771.74
1000	12.6204	163.987	475.393	884.797

Unsynchronized, lnxsrv09:

	Threads			
Array Size	1	8	20	40
5	14.3957	181.145	814.016	1309.41
100	14.2844	365.926	724.378	1422.49
1000	13.8368	173.213	354.528	676.846

Because the Unsynchronized implementation was missing the "synchronized" keyword and didn't include any reliability protections, all non-singular thread counts resulted in drastic output sum mismatches. Those results confirmed that this was an undesirable strategy, and would only be accurate enough in single thread cases. However, this implementation resulted in the fastest swap times, which was a primary goal for GDI.

AcmeSafe, lnxsrv10

	Threads			
Array Size	1	8	20	40
5	69.3441	436.589	2607.89	4942.79
100	24.5480	677.196	1297.57	2428.46
1000	24.6805	286.967	746.826	1457.27

Synchronized, lnxsrv10

	Threads			
Array Size	1	8	20	40
5	16.6083	392.989	1002.56	2046.09
100	16.7444	414.011	947.397	2001.23
1000	17.0631	336.184	795.673	1760.90

AcmeSafe, lnxsrv09

	Threads			
Array Size	1	8	20	40
5	26.9157	1365.69	1865.82	3134.59
100	25.1481	325.486	1613.43	2363.66
1000	32.8095	360.492	528.028	1055.13

Synchronized, lnxsrv09

	Threads			
Array Size	1	8	20	40
5	20.9990	2063.33	4790.49	10360.7
100	19.2853	2228.74	3312.09	7110.69
1000	19.6310	2383.05	3496.06	8052.28

As expected, AcmeSafe was perfectly reliable in all single and multithreaded test cases, but was significantly slower than Unsynchronized since it executed all swaps atomically. However, it was also slower than Synchronized. This came as a surprise since the “synchronized” keyword makes the instructions wait to obtain locks and requires much more overhead, originally leading me to predict that AcmeSafe would run faster.

After discussing this puzzling situation with other students on Piazza, I concluded that in order to make AcmeSafe faster, I would’ve had to utilize locks from the java.util.concurrent.ReentrantLock library. However in the context of this project, this would’ve defeated the purpose of creating a new implementation instead of simply using Synchronized, which already relies on locks.

Null, lnxsrv10

	Threads			
Array Size	1	8	20	40
5	11.4170	35.5472	195.340	189.919
100	10.9435	37.4763	87.3077	226.269
1000	10.7331	30.0677	106.119	176.757

Null, lnxsrv09

	Threads			
Array Size	1	8	20	40
5	37.2173	21.4210	58.5492	155.907
100	12.3260	22.7084	62.2604	177.028
1000	12.4423	25.3536	100.794	169.152

For all classes, there was a positive relationship between the thread count and the average swap time. This was expected, as this meant more threads competing and waiting for a turn to execute.

IV. Conclusion

In terms of performance, Unsynchronized ran the fastest as expected, and AcmeSafe surprisingly ran the slowest. In terms of reliability, Unsynchronized was incorrect for all multithreaded test cases, while AcmeSafe maintained correctness across all situations. In theory, AcmeSafe would've been an excellent middle-ground option for GDI, as it would've been faster than Synchronized while maintaining race condition protection.

Something I kept in mind was that results might depend on the CPU architecture, the server load at a given time, the server itself (4 versus 8 cores), and the different scheduling algorithms used to do the load balancing.

The most significant problem I had to overcome to acquire my measurements was waiting for a time when the servers weren't clogged up with other students running their tests as well. Because the level of traffic changed often, I was concerned that not conducting all tests at the same time would result in inaccurate results. Therefore I ended up scrapping and redoing a significant number of tests if I wasn't able to sit down and conduct a full set at the same time.

Overall, I found this to be an interesting experiment, and would advise GDI to continue using Synchronized or incorporate locks into AcmeSafe.