CS131 Homework 2 Report

Bryan Wong
Discussion 1A

Overview

In this report, we will explore concurrency using different Java synchronization implementations across two different virtual machines. Each of the Java synchronization implementations we will test is based off of the Java Memory Model (JMM), which defines how an application can safely avoid data races when accessing shared memory. The following implementations will be tested: Null, where no operation actually occurs; Synchronized, where the synchronized keyword is used; Unsynchronized, which has a data race; and AcmeSafe, which utilizes an AtomicLongArray. Each implementation is tested by running the UnsafeMemory driver, which creates a fixed size state array and performs a series of increment and decrement operations before checking for synchronization errors. By varying the number of threads, the size of the state array, and the CPU that runs the program, we will analyze the performance of each synchronization implementation.

1. Testing Server Information

Each synchronization test was performed and timed on two of the UCLA SEASnet GNU Linux servers, lnxsrv06 and lnxsrv10. All tests were performed on the same Java version, 13.0.2.

The two server machines use different CPUs. lnxsrv06 runs on a Quad Core Intel Xeon E5620 with 16 processors and a clock rate of 2.40GHz. lnxsrv10 runs on a Quad Core Intel Xeon Silver 4116 with 4 processors and a clock rate of 2.10GHz. The larger number of processors results in a higher expected level of concurrency for the lnxsrv06 server. Its higher clock rate also may result in faster computations.

Upon inspecting memory usage for both servers, both lnxsrv06 and lnxsrv10 have approximately 65.8GB of total memory. However, lnxsrv10 has significantly more available memory, with 13.2GB available compared to lnxsrv06's 6.4GB available. In addition, lnxsrv10 is only caching 7.3GB compared to lnxsrv06's 19.7GB. This excess of memory may result in lnxsrv10 performing faster, as data can be more effectively cached. This also indicates that lnxsrv06 is currently seeing heavier use by other users on the SEAS network.

2. Testing Parameters

In order to test the performance of our synchronization implementations, we will use the Linux command `time` to record the real time, CPU time, and system CPU time cost from the perspective of the Linux kernel. We will also record the total time and average swap time in both real time and CPU time from Java's point of view. These time measurements will be used in our analysis later. We will test the number of threads in increments of 1, 8, 16, 40 and state array size in increments of 5, 100, 500. We will use a static number of transitions, 100000000, for each test case. This will be done for each synchronization implementation and on both lnxsrv06 and lnxsrv10.

The UnsafeMemory driver takes these arguments, delegating the 100000000 transitions between the specified number of threads. The driver also takes the type of implementation as an argument, instantiating a corresponding State object with the specified number of elements in the state array. Each thread is started, then joined. A total time is calculated inside the driver, and the individual thread times are summed to get the resulting total CPU time.

3. Data Analysis

In order to analyze our results, we will use the total time recorded by our Java program as a heuristic. Tests performed on lnxsrv06 will be marked (16P) since 16 processors are used and those performed on lnxsrv10 will be marked (4P).

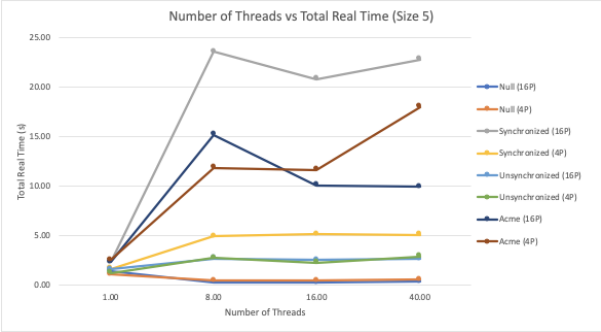

Figure 1: Number of Threads vs Total Real Time for State Array of Size 5.

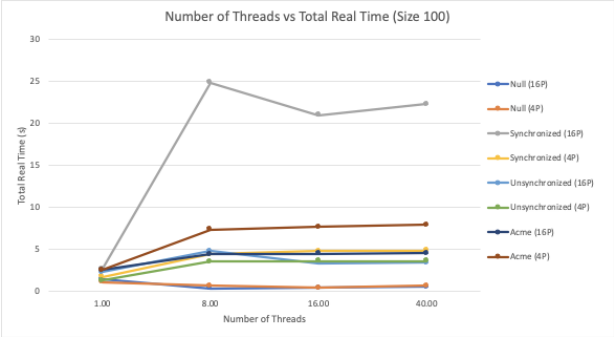

Figure 2: Number of Threads vs Total Real Time for State Array of Size 100.

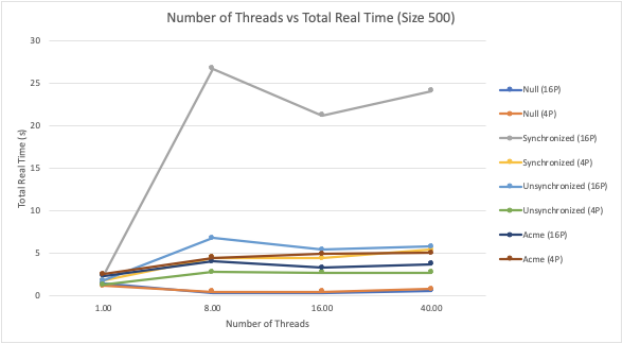

Figure 3: Number of Threads vs Total Real Time for State Array of Size 500.

3.1 Synchronization Implementation

Analyzing the 3 above graphs, we can see that the Null implementation behaves the fastest in each test, as no operation is actually being performed. The next fastest implementation is the Unsynchronized state, since no additional locking or synchronization mechanisms need to be performed. The last two implementations, Synchronized and AcmeSafe, had very different results on the two test servers. On the 16-processor server, the Synchronized implementation performed very poorly compared to the AcmeSafe implementation. On the 4-processor server, the Synchronized implementation performed slightly better than the AcmeSafe implementation. Since the Synchronized implementation relies on a naive mechanism where only one thread can execute the function at a time, this results in a major slowdown for the 16-processor server while impacting the 4-processor server less so. In the 4-processor server, it even outperforms the AcmeSafe implementation, which requires the overhead of maintaining an AtomicLongArray object. Ultimately, it seems that the AcmeSafe implementation is much more effective on servers that utilize high levels of concurrency.

3.2 Number of Threads

We can see that for all non-null synchronization implementations, the single thread version runs faster than any multithreaded versions. This may be due to the start-up times incurred from thread creation and scheduling. Since our performed task is quite simple (increment/decrementing an array element), the additional incurred costs may make the use of multithreading unnecessary. In addition, increasing the number of threads from 8 to 16 or 40 results in a mostly negligible change in performance. This is likely due to the fact that the test servers have 4 processors and 16 processors. By adding additional threads past what the CPU can run concurrently, no additional increase in performance is gained.

3.3 Server CPU

As discussed earlier, the 16-processor lnxsrv06 server handled the AcmeSafe implementation much better than the naive Synchronized implementation. The 4-processor lnxsrv10 server saw negligible differences between the two implementations, sometimes even performing better with the naive Synchronized implementation. This may be due to the fact that the smaller number of processors results in less slowdown from the naive implementation of the Synchronized state. For the other two implementations, Null and Unsynchronized, the two servers had a negligible impact on test results.

3.4 State Array Size

The size of the state array seems to have a negligible effect on performance in the cases of the Null, Unsynchronized, and Synchronized implementations. However, increasing the state array size greatly improves performance for the AcmeSafe implementation. This makes sense, as improving the size of the state array decreases the chances of two threads attempting to update the same array index at the same time. This does not improve the Synchronized implementation, since the naive implementation prevents any index from being modified, even if it is safe to do so.

3.3 Synchronization Errors

In terms of detecting synchronization errors, all implementations worked as expected. The Null implementation resulted in no errors, as no operation is performed. The AcmeSafe and Synchronized implementations also did not result in any errors, performing their synchronization tasks successfully. Only the Unsynchronized implementation with multiple threads resulted in mismatch errors where the sum of the state array elements was not zero. In every Unsynchronized implementation test case with more than one thread, a mismatch error was detected. Generally, the more threads there were and the smaller the state array was, the greater the degree of mismatch. This is because the odds of a data race increase if we increase the number of concurrent threads while reducing the number of possible data elements that can be randomly modified.

3.5 Optimal Use Cases

For the Unsynchronized implementation, the optimal use case is with a single thread, so that there is no risk of a mismatch error. As the single thread test runs consistently outperformed multithreaded tests, a single-threaded Unsynchronized state is likely the best option.

For the Synchronized implementation, the best performances occur on systems that do not have too many processors. Since the synchronized keyword results in only one thread being able to access a function at a time, using too many concurrent executing threads will bloat the program and result in major slowdown.

For the AcmeSafe implementation, the best performances occur on systems that do have a high level of concurrency that can take full advantage of the AtomicLongArray implementation. Additionally, a state array size that is relatively larger compared to the number of concurrently running threads will result in a better performance.

4. Conclusion

In this report, we have seen that the CPU architecture of the server can have significant effects on the effectiveness of different Java concurrency models. By using a number of threads that is well suited to the running system, efficiency can be optimized. In addition, we learned that naive synchronization implementations like the synchronized keyword can result in large slowdowns if used improperly. Additional measures like increasing the size of a state array to reduce the number of collisions when accessing array elements can be highly effective in optimizing performance.