Examining Performance in Java Shared Memory Threads

1. Abstract

This report evaluates the performance of different Java synchronization methods. We were provided a State class, which had functions to update and get elements of a private array. This class was implemented with use the Java “synchronized” keyword, volatile access, atomic access, and concurrent locking. To compare the performance, the classes were run with different numbers of threads, swaps, size of arrays, and total array sums. The results indicated locks had the greatest performance and reliability of the methods The Java environments used to test this project is Java(TM) SE Runtime Environment 18.3 (build 10.0.1+10), and Java HotSpot(TM) 64-Bit Server VM 18.3 (build 10.0.1+10, mixed mode).

2. BetterSafe Background

1.1. Concurrent

This package is useful because it separates task scheduling so that programs do not have to write their own thread pools and implement concurrency. Creating and using a thread in Java programming involves creating a new Runnable object, which is a bit unnecessary given the options available and the class to implement.

1.1. Concurrent Atomic

The Concurrent Atomic package extends those of Concurrent with the additional implementation of an atomic (volatile) array, which we test in GetNSet. While this is synchronized, accessing volatile memory takes more swatches, decreasing performance.

1.1. Concurrent Locks

The Conccurent Locks package extends the interface of Concurrent while also implemmetning locking and waiting. These are very simple to use, allowing greater flexibility and less of the awkward syntas of simply Concurrent. A con is that lock is not easy to use as a synchronized method itself, it must be appened by the developer into the function.

1.1. Invoke VarHandle

The VarHandle package provides write and read access under the condition that the object is immutable. VarHandle provides a generic standardized way of invoking Concurrent Atomic. Like Concurent Atomic, VarHandle has a performance issue of accessing older memory.

1.1. Conclusion: Concurrent Locks

After research and testing of the methods, Concurrent Locks appeared to be the most efficient and flexible way of implementing thread-handling for the State class

2. Implementation

1.1. Synchronized

The class implements State using the synchronized keyword on the swap function, which swaps two of the values contained in the private array. They keyword mandates that only one thread at a time should be able to access this function, avoiding a race condition.

The Synchronized and Null classes were compared against each other by running the UnsafeMemory script against datasets that controlled for the following tests: number of threads, number of swap transitions, array size, and sum of values in the array.

1.1. Unsynchronized

This class implements State using the same method of Synchronized but without the synchronized keyword. As a result, multiple threads are able to interleave and access the swap function. This creates a race condition, because on thread may have swapped two elements in the array and another thread may at the same time swap those two elements again, resulting in the wrong results.

1.1. GetNSet

This class implements State by using an atomic array to deal with race conditions. An atomic array uses volatile accesses to array elements. Whenever the swap function is called, the compiler goes all the way to memory and stores the element there. As a result, there are no issue of caches having old versions of the elements. This prevents race conditions, because only one thread will get the most updated version of the elements and switch them in memory too.

1.1. BetterSafe

This class implements State using Reentrant Locks. It has the same behavior of the synchronized method, but with extended capabilities. A lock is owned the by the thread that is currently locking it, and it involves more flexible scheduling because it supports lock polling, unlike the synchronized keyword. The lock was placed in the first line of the swap function, and the lock is released just before swap returns. As a result, the thread holds the lock for only its duration inside the swap function. There are no race conditions of infinite locking.

2. Results

All of the classes were tested with the “number of threads” ranging from 1 to 32, and “number of swaps” from 1000 to 1000000. The Null and Synchronized classes had additional testing, since these were the given base cases in the assignment. These tests were “array size” ranging from 5 to 10, and “sum of values” ranging from 100 to 200.

The table below tests the Null and Synchronized states against each other by Array Size. I controlled for the following variables: 8 threads and 1000000 swaps.

States	**Array Size**		
	5	**10**	**20**
Null	3267.21	2510.78	2080.33
Sync.	2984.28	2446.54	1705.05

The following table tests the Null and Synchronized states against each other by increasing the sum of the elements in the array. I controlled for the variables: 8 threads, 6 as the max element, and a 5 element array.

States	**Sum of Values**			
	100	**150**	**175**	**200**
Null	1674.50	2122.93	1371.6	-
Sync.	2869.21	2956.22	1802.58	2307.55

The next table compares all five of the classes against each other by number of threads and number of swaps. The variables controlled were: 5 as array size, and 6 as max element.

States	**Swap**	**Thread**		
		1	8	32
Null	1000	3275.34	31665.8	276937
	100000	65.4051	1816.49	9872.85
Sync.	1000	3871.01	54258.1	320535
	100000	63.1662	2930.96	9392.74
Unsync.	1000	3136.83	43545.5	3.68E+7
	100000	N/A	N/A	N/A
GetNSet	1000	8608.12	61549.9	451448
	100000	70.2346	5404.46	18618.9
Better	1000	63.1662	2930.96	9392.74
	100000	117.1	1684.96	6370.02

2. Comparisons

We will evaluate performance based on the inverse of ns/switches.

1.1. Base Case: Synchronized versus Null

Both Null and Synchronized appeared to decrease in ns/switches, a negative correlation between number of threads and performance. Synchronized’s ns/switches were consistently lower than those Null’s, indicating that Synchronized had better efficiency than Null. No trends stood out in the tests for Sum of Values. These values ranged within 1000 ns/switches to 2500 ns/switches.

We can reliably expect increased performance with greater threads. Due to the negative relation between swap transitions and class, we can also expect reliably that performance decreases with swaps. Synchronized appears to suffer less than Null with this condition, which is likely because Synchronized applies protection to the threads. Based on the similar results for the array size tests, we can expect that performance decreases with array size. The Synchronized class is reliable.

1.2. Unsynchronized

In all cases of 1000000 swap transitions, Unsychronized failed to terminate so no results were recorded. For 1000 swaps, Unsychronized ns/switches had decreased performance with more threads but also have incorrect sum matchings, so less accuracy. From this, we can conclude that Unsychronized results in an increased chance of race conditions. Since Unsychronized has no way of taking care of race conditions, this was expected. The class is overall unreliable.

1.3. GetNSet

GetNSet had higher ns/switches with more threads for both 1000 and 1000000 swaps. The trend increased at a greater rate than all the classes except for BetterSafe. When comparing the results, we can expect that GetNSet has the worst performance. The class does maintain synchronization, making it reliable.

1.4. BetterSafe

BetterSafe tests indicated a positive correlation between ns/swatches and number of threads for both 1000 and 1000000 swaps. Of all the classes, Better Safe had the least ns/swatches per category, indicating high performance. We can expect that BetterSafe is a reliable class.

3. Challenges

Implementing Unsychronized was simple but tests with Unsychronized originally resulted in hanging. I was unsure about the cause but realized that Unsychronized failed when there were too many swaps occurring and race conditions were creating infinite loops. Unsychronized finally began terminating with a smaller number of swap transitions so I test its performance in that case. Another issue was implementing BetterSafe. I had originally created the lock inside the function and had forgotten to add the unlock before all the return statements. This was an easy fix, so that the scope of locking would only be within the function.

4. Conclusion

After researching and testing multiple packages for concurrent programming, BetterSafe appears to have the best performance and synchronization. Unsychronized fails to terminate since it does not account for race conditions. Null also does not terminate at high sum values. The rest of the classes had lower performance.