Analysis of Java shared memory performance races

Abstract

The objective of this report is compare 4 different classes used to test concurrency on a simple prototype based on performance and reliability. This report also analyzes 4 different packages and classes that can be used to achieve synchronization and compares them against each other. The measurements are conducted on OpenJDK version 11.0.2. It is found that the BetterSafe class is the most reliable and has the best performance and while the different concurrent packages have their pros and cons, for this implementation, the locks package seems more suitable.

1. Comparison of Classes

For this project, we tested 4 different classes that implement a State class which is the API for a simulation state that contains a swap method. The 4 classes are: SynchronizedState, UnsychronizedState, GetNSetState and BetterSafeState. The NullState is used as a baseline classifier. These classes are measured in terms of performance and reliability and there are 3 main parameters that are varied – number of threads, number of swaps and number of elements. The maxval is the same for all of them – 127.

1.1. Number of threads

The program was invoked with 1,000,000 swap transitions, 600 elements and varying number of threads – 1, 8, 16 and 32.

	Time for transition (ns)				
Class	**1 thread**	**8 threads**	**16 threads**	**32 threads**	**DRF?**
Null State	37.0630	2022.09	4043.90	6765.31	Y
Synchronized State	45.1744	2689.55	5667.80	14042.8	Y
Unsynch Ronized State	41.7453	1352.87	4807.68	5401.65	N
GetNSet State	59.9580	979.253	2720.57	7458.01	N
Better Safe State	65.6886	1280.87	2632.77	4779.01	Y

We observe that as the number of threads increases, the performance decreases. This could be because the thread creation overhead outweighs performance benefits from using multiple threads in this implementation.

1.2. Number of swaps

The program was invoked with 8 threads, 600 elements and varying number of swaps – 100,000, 500,000, 1,000,000, 5,000,000.

	Time for transition (ns)				
Class	**100,000 swaps**	**500,000 swaps**	**1m swaps**	**5m swaps**	**DRF?**
Null State	1488.04	2335.62	2062.87	263.280	Y
Synchronized State	4166.11	3324.58	2673.75	2106.71	Y
Unsynch Ronized State	1566.35	2518.41	1427.33	622.498	N
GetNSet State	2460.90	2144.21	1041.78	528.568	N
Better Safe State	4975.24	2121.98	1190.51	759.908	Y

We observe that for a higher number of swaps, the performance generally increases.

1.3. Number of elements

The program was invoked with 1,000,000 swap transitions, 8 threads and varying number of elements – 6, 100, 600, 1000.

	Time for transition (ns)				
Class	**6 elements**	**100 elements**	**600 elements**	**1000 elements**	**DRF?**
Null State	2118.13	1986.99	1982.07	1954.95	Y
Synchronized State	2617.08	2500.14	2623.43	2368.17	Y
Unsynch Ronized State	1566.35	1858.93	1438.42	1373.24	N
GetNSet State	1762.75	1477.63	1292.08	1179.65	N
Better Safe	1110.60	1086.55	1329.10	1360.22	Y

State					

There seems to be no significant difference in performance as the number of elements increase apart from a small improvement in performance.

1.4. Comparison across classes

In general, for most of the measurements, we observe that while both BetterSafeState and SynchronizedState are data race free and 100% reliable, BetterSafeState is better in terms of performance than SynchronizedState and takes a smaller time per transition.

Additionally, both GetNSetState and UnsychronizedState and consistently faster than SynchronizedState for all measurements. This is expected because the use of synchronization results in controlled access to shared memory and the program is not able to maximize the benefits of using multithreading. Hence performance is sacrificed for reliability.

One would expect UnsynchronizedState to be faster than GetNSetState because UnsynchronizedState is completely lock free while GetNSetState is halfway between synchronized and unsynchronized as it uses volatile accesses to array elements. However, there is no clear distinction in the performance of the two classes.

1.4. Best choice for GDI's applications

GDI prioritizes speed over completely accurate results and does not require 100% reliability. Hence, I believe that either BetterSafeState or GetNSetState should be used for GDI's applications. SynchronizedState is the slowest in terms of performance among the 4 and hence is not suitable. While GetNSetState is not 100% reliable, it is significantly better than UnsychronizedState in terms of reliability and the occasional lapse in reliability would be permissible for this use case. From the above measurements, both GetNSetState and BetterSafeState are similar in terms of performance and more testing would have to be done to determine which is the better choice.

2. Comparison of Packages

2.1. Java Concurrency Packages

There are 4 different packages that can be used that retains 100% reliability.

1. java.util.concurrent: This package consists of Synchronizers such as Semaphores, CountDownLatches, CyclicBarriers, Phaser and Exchangers. This package. Semaphores seemed to me to be the most intuitive out of the 5 with its acquire() and release() methods but it seemed like using a Semaphore entails writing a more thoroughly tested application as the release() method does not do exception handling on its own and was more useful for programs that require more fine-grained control.
2. java.util.concurrent.atomic: This packages consists of Atomic classes that provide access and updates to a single variable of a certain type and provide atomic increment methods and methods such as which could be useful for our implementation. However, upon looking at the API, it seems like this package is not useful for both read and writes within the same critical section and from GetNSet, which uses AtomicIntegerArray, atomic locking does not produce a data race free program.
3. java.util.concurrent.locks: This package consists of Lock and Condition interfaces and provides flexibility in their use at the expense of awkward syntax. The main implementation of the Lock interface is ReentrantLock which is what was used in my implementation of BetterSafe, with its distinguishing factor being the ease of usage. For more complicated programs, however, Locks have the drawback of requiring to be explicitly released by the programmer which opens the program to potential bugs when locking and unlocking occurs in different scopes.
4. java.lang.invoke.VarHandle: A VarHandle is a dynamically strongly typed reference to a variable where access is supported under various access modes. It offers low-level manipulation, which seems overkill for a simple program and would reduce readability.

2.2. Comparison of BetterSafe and Synchronized

As seen from the previous section, the BetterSafe implementation has a faster performance than Synchronized. This is due to several reasons:

1) In Synchronized, a thread can be blocked infinitely while waiting for the lock and it is not possible to interrupt a thread waiting to acquire a lock, while in BetterSafe, the thread can be interrupted with timeout while waiting for thelock.
2) Synchronized also does not support fairness. Once a lock is released, any thread can acquire it and no preference can be specified. Whereas with BetterSafe, the lock is provided to the longest waiting thread.

2.3. Reliability of BetterSafe

BetterSafe is still 100% reliable because the lock contains all the operations being performed on the shared memory structure. Hence only one thread can enter the contained

code block at a time, preventing two conflicting accesses and ensuring that the program is data race free.