Java Shared Memory Performance Races Project Report

CSCI 2110 Programming Languages
Name: Fredrick Li

1. Class Analysis

1.1 SynchronizedState

This is an implementation of the State class that uses the synchronized keyword to provide thread synchronization to the swap method. This makes it so that only one thread can execute the logic in the swap method at any one time, while other threads block until the thread executing the method finishes. This provides locking on the entire swap method, which avoids conflict and thus makes this class DRF.

In terms of performance, this is the slowest implementation of the State class as it implements coarse-grained locking on the entire swap method. In this way, it doesn't take advantage of multithreading (as all threads but one will be blocked while waiting to execute). In addition, the synchronized keyword doesn't support fairness, which leads to potential starvation of threads. While reliable, performance is not idealized.

1.2 UnsynchronizedState

This is an implementation of the State class that doesn't use the synchronized keyword. As there is no synchronization among threads, this leads to race conditions. Multiple threads can execute swap at the same time, which could result in writes of incorrect values to the array. In performance-wise, this is the most efficient implementation since there is no locking or blocking of threads.

1.3 GetNSetState

This is an implementation of the State class that uses volatile accesses of the array elements using the java.util.concurrent.atomic.AtomicIntegerArray class (and the associated get and set methods as opposed to using the synchronized keyword). As there is no synchronization keyword, multiple threads can execute the swap method simultaneously. As increment and decrement consist of two operations, they need to be handled in a single atomic operation. Therefore, race conditions will be met if threads are interleaved in their execution; and therefore, this implementation is not DRF. Performance-wise, this implementation is faster than Synchronized as it only consists of synchronizing the values of the array elements, instead of locking the entire swap method.

1.4 BetterSafeState

This is an implementation of the State in which I chose the ReentrantLocks from the java.util.concurrent.locks package. It is similar to Synchronized that it locks the swap method, but instead of the coarse-grained locking that Synchronized provides (locking the entire swap method), BetterSafe implements more finer-grained locking that only locks the portions of the method where the race condition occurs. Therefore, BetterSafe is also DRF. By only locking the necessary portion of the method, BetterSafe makes better use of multiple threads.

2. Performance Test & Results

To test the performance of each of the implementations of the State class, I used 4, 8, 16, and 32 threads to test 1,000,000 swaps for an integer array of size 10 consisting of the random integers. The results shown below are the averages of running each case 50 times each.

Class Name	Time to completion				DRF
	Number of threads				
	4	8	16	32	
NullState	353	1682	4296	8746	Yes
Synchronized	980	2249	4736	10592	Yes
Unsynchronized	469	1421	4454	6756	No
GetNSetState	789	1575	3475	8224	No
BetterSafeState	610	1265	2395	5279	Yes

From the table above, BetterSafe has the best implementation of the different implementations as well as provides the DRF guarantee. Synchronized has the worst performance but provides the DRF guarantee. In the middle are the Unsynchronized and GetNSet implementations, which are both faster than Synchronized; however, they aren't DRF and are not reliable options to choose from.

3. Package Analysis

3.1 java.util.concurrent

This package contains many different methods to handle concurrent programs and coordinate thread synchronization. The options provided in this package allow for stronger memory consistency guarantees but at increased complexity. For the simplicity to synchronize threads for increasing/decreasing values of the elements of an array for a single method, the added complexity was not necessary.

3.2 java.util.concurrent.atomic

This package provides synchronization of threads without the explicit use of locks and opts for volatile accesses of the array elements. It is used in the GetNSet implementation, but the atomic operations weren't used. The getAndIncrement and getAndDecrement operations would be a good choice to use, but with just an AtomicIntegerArray, the entire array would be effectively locked when one thread goes to make a change to it. This is effectively the same as locking the entire swap method. If each element of the array was an AtomicInteger, then it could exhibit the type of fine-grained locking: just the individual elements of the array.

3.3 java.util.concurrent.locks

This package provides synchronization of threads via various types of locking and waiting conditions. I chose to use this package to implement BetterSafe as it provided a simple and reliable lock to synchronize threads for the swap method that performed faster than the reliable Synchronized implementation. ReentrantLocks are easy to implement and could only lock the race condition section of the swap method as opposed to locking the entire method.

3.4 java.lang.invoke.VarHandle

This class provides read/write access to variables in an atomic way similar to java.util.concurrent.atomic. However, on top of the added complexity to implement BetterSafe using this option, it also allows for interrupts, which would slow down performance.