CS131 Homework 3 Report

Abstract

The primary objective of homework 3 was to analyze the relationship and tradeoffs between thread safety and performance. This objective was carried out by comparing different approaches to achieve synchronization between threads in the form of a program that continuously swapped numbers on an array of integers that increment and decrement the values of the array elements.

Testing Platform

java -version

- openjdk version "11.0.2" 2019-01-15
- OpenJDK Runtime Environment 18.9 (build 11.0.2+9)
- OpenJDK 64-Bit Server VM 18.9 (build 11.0.2+9, mixed mode)

/proc/cpuinfo

- model: 62
- model name: Intel(R) Xeon(R) CPU E5-2640 v2 @ 2.00GHz
- cpu MHz: 1927.001
- cache size: 20480 KB

/proc/meminfo

- MemTotal: 65755884 kB

1. Analysis of Class Implementations

1.1 NullState

This is an implementation of the State class that does nothing (i.e. swapping has no effect). This class is only useful for understanding any overhead or bulkiness of the implementation, and for deriving a baseline case for timing the scaffolding of the simulation.

1.2 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 (as opposed to just locking the section of the method that encounters the race condition). By locking the method 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 (i.e. no queue or preference for threads waiting to execute), which leads to potential starvation of threads. Thus, while reliable, performance suffers.

1.3 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; and thus, this is not DRF. Multiple threads can execute swap at the same time, which could result in stale reads and writes of incorrect values to the array. As an example, the test case '*java UnsafeMemory Unsynchronized 8 1000000 53 111 4 18 62 123 31 96 75 16*' fails.

As far as performance is concerned, this is the most efficient implementation as there is no locking or blocking of threads.

1.4 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. This would be fine if we used the atomic operations of the *AtomicIntegerArray* class (i.e. *getAndIncrement* and *getAndDecrement*), which would prevent staleness and leave the array in a reliable state. However, as increment and decrement consist of two operations (i.e. read and write), they need to be handled in a single atomic operation (which *get* and *set* methods don't do). Therefore, race conditions will be met if threads are interleaved in their execution; and therefore, this implementation is not DRF. As an example, the test case '*java UnsafeMemory GetNSet 8 1000000 53 111 4 18 62 123 31 96 75 16*' fails.

Performance wise, this implementation is faster than Synchronized as it only consists of synchronizing the values of the array elements, instead of also locking the entire swap method.

1.5 BetterSafeState

This is an implementation of the State in which I chose to use *ReentrantLocks* from the *java.util.concurrent.locks* package for synchronization purposes. This implementation is similar to Synchronized in that it locks the swap method, but instead of the coarse-grained locking that Synchronized provides (i.e. 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 (as opposed to the entire method), BetterSafe makes better use of multiple threads. In addition, *ReentrantLocks* provides more synchronization flexibility by implementing fairness for lock acquisitions, which improves performance in the face of heavy contention and reduces the chance of starvation.

BetterSafe ended up performing faster than all other implementations (except for Null as expected), while beating Synchronized due to finer granularity and increased flexibility.

2. Performance 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 integers 53, 111, 4, 18, 62, 123, 31, 96, 75, and 16. The results shown below are the averages of running each case 50 times each.

Class Name	Time to completion (ns/transaction)				DRF
	Number of threads				
	4	8	16	32	
Null	354.13	1691.09	4296.40	8746.13	Yes
Sync	980.16	2548.11	4735.83	10508.58	Yes
Unsync	469.23	1420.98	4453.56	6755.59	No
GetNSet	789.56	1576.74	3474.84	8224.21	No
BetterSafe	610.28	1254.94	2394.99	5278.79	Yes

2.1 Performance Analysis

As can be seen from the table above, BetterSafe has the best implementation of the different implementations as well as provides the DRF guarantee. On the opposite end, Synchronized has the worst performance, but also 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 for GDI to choose from.

3. Analysis of Concurrency Packages

From the four Java Concurrency Packages recommended for the implementation of BetterSafe, I ended up choosing *java.util.concurrent.locks*. The pros and cons of each of the options is discussed below.

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 comes at the tradeoff of having increased complexity. For the simplicity of the assignment to synchronize threads for increasing/decreasing values of the elements of an array for a single method, the added complexity was not necessary (and I opted to go for a simpler implementation).

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 (as discussed in section 1.4). 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 (as the swap method only increments or decrements values of elements of the array). Instead, if each element of the array was an AtomicInteger, then it could exhibit the type of fine-grained locking we are looking for, as the entire array wouldn't be locked, but just the individual elements of the array. For similar reasons stated in section 3.1, I opted to go with a simpler implementation.

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. Namely, *ReentrantLocks* are easy to implement and provide the fine granularity necessary to 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 (which is something that we are trying to avoid with this assignment).

4. Problems Encountered

This was a relatively straightforward homework assignment, so not too many problems were encountered. The main problem that I ran into was trying to choose command-line parameters to get the Unsynchronized and GetNSet implementations to not hang up during execution. Using some hints discussed on Piazza, I realized that there were many ways to avoid hang up (e.g. smaller number of swaps, higher maxval, more elements in the array, etc.).

5. Conclusion

After careful analysis and testing of the various implementations, the BetterSafe implementation is the most reliable (i.e. DRF) and has the fastest performance.