Homework 4. Java shared memory performance races

Testing Platform:

Java version:

Java version "1.8.0_05"
Java(TM) SE Runtime Environment (build 1.8.0_05-b13)
Java HotSpot(TM) 64-Bit Server VM (build 25.5-b02, mixed mode)

CPU info: (from /proc/cpuinfo):

Intel(R) Xeon(R) CPU E5620 @ 2.40GHz

Memory info: (from /proc/meminfo):

Memory Total: 32865604 KB

DRF (Data Race Free)

In this assignment, I implemented several versions of State. I use AtomicInteger for my BetterSorry, AtomicIntegerArray for GetNSet, and ReentrantLock for BetterSafe. As a result, only Synchronized and BetterSafe provide me with 100% reliability. To reach 100% reliability, we have to ensure that every read/write/modify operation has to be atomic, otherwise there will be situations that create inconsistency, such as when one thread writing on a variable while other thread(s) reading it. In most cases the result of data race would be hard to predict because we cannot be sure the order and timing of threads creating such inconsistency. In this homework, Synchronized and BetterSafe reach 100% reliability for the reason that Synchronized ensures only one thread can enter the swap section, and BetterSafe ensures there is only one thread can modify the array at a time. Both implementations, after testing, gives me 100% of correctness. For the other methods like GetNSet and BetterSorry, they did not reach 100% reliability because even though each element in array requires atomic operation, there could be situation that before some threads writing to a variable, some other threads read the value and do the increment/decrement without knowing the latest update on this variable. Therefore those methods can reach better reliability than UnSynchronized but with less than 100% reliability.

Synchronized vs Null

It is apparent that both will reach 100% reliability, and Synchronized will take more time since Null basically does nothing. The resulting time statistics are as follows: (20 trial for each case)

	Time per transition (8 threads, 1M swaps, maxval = 10, initial elements: 0 1 2 3 4 5)	Time per transition (16 threads, 5M swaps, maxval = 20, initial elements: 3 4 8 0 1 2)
Synchronized	3433.11 ns	4709.88 ns
Null	708.42 ns	329.71 ns

UnSynchronized

For Unsynchronized, my implementation is quite simple, just by removing the synchronized keyword.

GetNSet

For this method I used AtomicIntegerArray to help me with the implementation. By using get and set method, it ensures that every element in the array is updated atomically. Apparently this method reach better reliability than UnSynchronized because it ensures atomic operation within each element, but it does not provide 100% reliability because inconsistency may still happen for reading and writing. For instance, if one thread reads the value just before other thread ready to modify it, then the thread reading the value may not be able to get the latest updated value of that variable and may then do an illegal increment/decrement to that variable.

BetterSafe

For this implementation I used ReentrantLock to reach 100% reliability while having better performance than Synchronized. By using the reentrantlock I ensure each time there is only one thread can enter the critical section to read and modify the array values. The reason why BetterSafe and reach 100% reliability but also reach a better performance than Synchronized is that lock is a low level operation that can also provide us with the guarantee that while one thread reading/writing to the array, other threads have to stay outside of the critical section and wait until the current thread unlocks the lock it holds.

BetterSorry

For BetterSorry, I used AtomicInteger to implement. Theoretically I think it should give me better performance since it only ensures atomicity at element level, thus allowing more threads doing work at same time than GetNSet which uses Atomic Array. However, it will also introduce more instability because of more threads are allowed at the same time. My BetterSorry is clearly more reliable than UnSynchronized because of using atomic integers to protect race conditions when while updating single element. But it is also less reliable than Synchronized since race conditions can still occur. The weakness of BetterSorry is obvious, one possible scenario could be right after one thread reads the value into v1 through get(), another thread immediately reads another value into v1 and thus creates inconsistency without the older thread being notified. In the testing, BetterSorry does give out better reliability than UnSynchronized and GetNSet. But I think when sample size becomes larger, results would become uncertain. One situation that can result in a high probability of race conditions of BetterSorry could be a large number of threads each modifying the array elements intensively.

Reliability Comparison

For reliability measurement, I think we cannot simply divide the actual sum by expected sum because the sum can keep increasing or decreasing arbitrarily. Also, from the original settings, we do not know the result of each single swap being successful or failure. Therefore I think up a method to gather the total number of swaps done by using System.out.println inside the testing harness (SwapTest.java). Basically I put the print method inside the loop including the swap operation to print out the loop number into a separate txt file, since the loop will only increment if the swap succeeds, then by counting the total number of loops, we can calculate the reliability by dividing the total loops by expected number loops which is the number of swapping set in the parameter. The resulting statistics for each implementation are as follows:

*testing parameter: 1M swaps, maxval = 10, threads = 8, initial array elemets: 0 1 2 3 4 5
*testing platform: SEASNET

	Synchronized	Null	GetNSet	BetterSafe	BetterSorry	UnSynchronized
Time (ns)	3245.38	609.83	1423.10	1749.06	1180.32	834.13
Reliability (%)	100	100	58.9	100	64.3	54.7

From the above statistics, I think in this particular case or similar, **BetterSorry** would be a better choice for the reason that it gives me a good balance between performance and reliability. However, if provided with different parameters, the reliability percentage would fluctuate largely. (For instance, if provided with very large number of swaps, BetterSorry's reliability would possibly be surpassed by GetNSet.)

DRF Model: Synchronized, Null, BetterSafe

Non-DRF Model: UnSynchronized, BetterSorry, GetNSet

Reliability test that *may* fail the Non-DRF model:

```
java UnsafeMemory BetterSorry 32 10000000 10 0 1 2 3 4 5
java UnsafeMemory GetNSet 32 5000000 4 0 1 2
java UnsafeMemory Unsynchronized 8 1000000 6 0 1 2 3 4
```

Difficulty met/overcame while doing the assignment: (Program Freeze)

One of the major difficulty I met was **"freeze" issue**, particularly for Unsynchronized. To be more specific, when I set the swap number to be something large (> 1 million), my UnSynchronized.java (also for BetterSorry & GetNSet sometimes) is likely to freeze during the run-time without returning. I thought it should not be the deadlock case because I paid particular attention when programming. Also, when I put some debugging code (print message like "we got here") in my file, the freezing behavior disappeared, which makes it even stranger to me. I personally think it definitely has something to do with **thread resource contention**. Considering the following situation: if two threads keep atomically incrementing the same integer, they may experience contention due to cache ping-ponging. In this case no dead locks are involved, but "freeze" may occur.