Java Shared Memory Performance Races

Reinaldo Daniswara

1. Introduction

Race condition is a condition that every multithreading program need to avoid. In fact, the definition of race condition itself is a type of concurrency bug that could occur into a program that runs on a parallel execution by multiple threads at the same time.[1] In order to avoid this problem, programmers tend to insert synchronization. However, synchronization sometimes sacrifices the speed of the applications because it will wait until one threads to finish before executing other threads. Thus, in this project, I will experiment with the synchronization method by creating multiple implementation of synchronization, such as Unsynchronized, GetNSet, BetterSafe, and BetterSorry.

2. Implementation

2.1. Unsynchronized

This class is fairly similar with synchronized class. The only different implementation is I removed every synchronized words in it class. In contrast, the main reason of synchronization is to avoid any data races condition by limiting the access of the threads. Thus, if you remove the synchronize block, it is obvious that it will not consider as Data Race Free or DRF. As a result, even though it run faster than synchronized state method, it will produce a wrong result.

2.2. GetNSet

In order to implement GetNSet method, I does not use synchronized code, but I implement it with get and set method of AtomicIntegerArray. Theoretically, this implementation will also consider as no Data Race Free since it does not has a lock and unlock mechanism like synchronize method has. In addition, AtomicIntegerArray could cause a data racing because they may share the same get() value with other threads, which make the implementation of GetNSet is not fully reliable.

2.3BetterSafe

In order to implement a BetterSafe class, I am using a Reentrant Lock method. According to the explanation from oracle website that explains about the ReentrantLock class, this class has the same basic behavior and semantics as the synchronized methods and statement with additional feature like fairness. Which it will give access to the longest-waiting thread. The Lock will be acquired by the lock() method and will be hold in the thread until unlock() method called. Since the idea of the implementation of this class is quite similar with the Synchronized method, plus some additional benefits, it will be fair to say that this class is Data Race Free. In addition to that, this class is also 100% as reliable as synchronized because it provides a similar thread safety ability as Synchronized class.

2.4. BetterSorry

The implementation of BetterSorry is a little bit similar with the GetNSet class. However, in BetterSorry, I am using AtomicInteger which will require an object per element, instead of AtomicIntegerArray. Theoretically, this implementation is faster than GetNSet because instead of accessing an object and an array, it will just access the element of the array. However, this class is 100% reliable as well since it is not DRF. It does not has such a lock mechanism like synchronized method, thus it may shares some value like the implementation of GetNSet.

3. Experiment

In order to determine the performance and reliability, I run a test harshness using 8 threads and play around with the number of swap transitions. I run the test harshness for 21 times for each class and each threshold. Then, take the average of the result, plus record any mismatch that occur.

For example:

if the threshold is 1000000 transitions, I run command

“java UnsafeMemory Synchronized 8 1000000 6 5 6 3 0 3” (21 times)

and do the same for the other classes.

If the threshold is 100 transitions than I run,

“java UnsafeMemory Synchronized 8 100 6 5 6 3 0 3” (21 times)

4. Result

Implementation	Transition						DRF
	10^2	FR(%)	10^4	FR(%)	10^6	FR(%)	
Null	191,742.8	0	8,781.51	0	2,225.97	0	Yes
Synchronized	245,371	0	11,683.31	0	2,334.37	0	Yes
Unsynchronized	350,134.54	63.63	Hanging				No
GetNSet	439,898.18	72.72					No
BetterSafe	510,752.45	0	16,519.35	0	1,498.78	0	Yes
BetterSorry	270,942.27	9.09	12,770.80	0	2,138.85	0	No

Table 1.1 indicate the time taken for number of swap transitions in nanoseconds/threads. FR is a fault rate of the experiment such as the mismatch or output too large.

From the result of the table, we can see that as the number of transition increase from 10^2 to 10^6, every implementation of the class is becomes faster and faster (at least for synchronized, bettersafe, and bettersorry). If we compare each classes by the number of swap transitions, synchronized seems faster than any other class when the number of swap transition is quite low because the number of transition is still small. However, as it increases, then there will be multiple data trying to access the number of same i and j value in swap method, and here synchronize will make the program control the access of the data by applying the locking method, which make a program run slower. However, the implementation of ReentrantLock() by BetterSafe is indeed make the program way way faster than the synchronized, yet reliable as well since it always correct, plus it has no data race occur.

We could also see that BetterSorry will be faster than BetterSafe as long as the number of transition < 10^6. The reason is because when the number of swap still low, the probability of data races that may occur also lower. Thus, BetterSorry which uses an implementation of AtomicInteger will be faster. However, BetterSafe is faster when the number of swap is large because it has the powerful tools of ReentrantLock() that BetterSorry doesn't have. Even though BetterSorry is quite fast and reliable (by looking at the fact that the fault rate is super low), it is not DRF. It may has a data race condition. It is not easy to determine what command that constantly failed the BetterSorry, but I did it a couple time when I put command
"java UnsafeMemory BetterSorry 8 100 6 5 6 3 0 3"
"output too large (7 != 6)"

For Unsynchronized, and GetNSet, it has a weird behavior where it could run as long as the number of swap transition is low. As I tried using a different threads other than 8, it will also fail and hanging if the number of swap transition is high. I think the reason is because it is trapped in the infinite loop while swapping. On the other hand, null is faster that synchronized because it does not really do any swapping activities.

5. Difficulties

The main problem that I encounter to measure the performance is the inconsistent processing time from Linux server 9. If there are a lot of people who are accessing the server, it will impact the performance of the running time. Thus, I need to take a numerous try to get more precise, valid, and accurate data.

6. Conclusion

In conclusion, it is hard to create a super perfect implementation. Sometimes, you need to sacrifice the performance while maintaining the accuracy and reliability of the program. In addition, we may also need to sacrifice the correctness and in order to get the fast result. However, we still have a couple key takeaway from the experiment. First of all, it is not a good idea to completely neglect synchronization method like in unsynchronized class because the result is not reliable. Secondly, if we want to have a swap transition that less than 10^6, BetterSorry is good because it offers speed with 90% reliability. Lastly, BetterSafe is the best method for a very very large transition because it offers speed with 100% reliability.

7. Bibliography

[1] http://javarevisited.blogspot.com/2012/02/what-is-race-condition-in.html
[2] https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReentrantLock.html