Homework 3. Java shared memory performance race

1. Introduction

In the multicores and multithreading program, processes and threads are running in parallel. Multiple threads share the same variables for the faster and efficient use of the system. However, there is possibilities of error when two threads operating on the same variable at the same time. This triggers race condition and it is fatal as it produces thread interference and memory consistency errors. The race condition defines as when two or more operation taking places and operations don't happens in proper order and hence unexpected programs behaviors can be seen. Hence, to avoid this problem, a program synchronization is one of the option. In Java, a key "Synchronized" is used to define method to avoid race condition. However, Synchronized method has drawback as it slows the applications called thread contention. Because one threads tends to wait for other threads during the execution time. Hence, my goal is to change the implementation of the given class so that it can speed up the executing process. Thus I am going to create class Synchronized, Unsynchronized, GetNSet, and BetterSafe and measure the characteristic's and class performance and reliability.

2. Implementation

2.1 Unsynchronized

This class is using the keyword "Synchronized", so the goal is to remove the key and make a different class called Unsynchronized. When remove the key, it is possible that there is uncertainty of Happens-before order that may cause the data race.

2.2 BetterSafeState

Goal of implementing this class is to gain 100% reliability and get better performance without using Synchronized. Towards the way of implementing this class, I had to find the solution for the race condition. Exploring the oracle website, I came across where article states that I can use "Lock" and "Unlock" key to achieve the shared resource by different thread. To use Lock and Unlock, I can use "ReentrantLock" class that give access to the longest-waiting thread. While using Lock(), once a thread get Lock ID, no other thread can use the same resource at the same time. So after the thread done with using resource, it can release the lock by key "unlock()" method. Thus it is believed that this class will be 100% reliable as synchronized because it provides a similar thread safety ability as synchronized does.

2.3 GetNSetState

To implement this class I have used AtomicIntergerArray as spec suggested. To use this array I have imported library from java.util.concurrent.atomic.AtomicIntegerArray. The benefits of using this method is that it does not use the synchronized code. But on the other hand, it does not guarantee the race free condition like synchronized do and thus it is not reliable. Because using the get and set method it is possible that different thread can manipulate the same data at the same time.

2.4 BetterSorryState

The implementation of this class is similar to the GetNSetState. I have used the atomic library for the implementation but used atomic integer library. I have imported from the library java.util.concurrent.atomic.AtomicInteger. It is different with atomic array in a sense that it requires an object per element in contrast to AtomicIntegerArray. The benefits of using AtomicInteger is that it is faster than AtomicArray because it only access the element of the array while other access an object and an array. On the other hand this class is 100% reliable since it is not Data Race Fee. Also it don't have the same mechanism like synchronized and thus will be unreliable but faster.

3. Experiment

I ran test using "java UnsafeMemory CLASS 8 (100 – 1000000) 6 5 6 3 0 3". Each time I ran, I got different results, so decided to run 15 times and averaged out the result. I kept all parameter constant and change the swap transitions from 100 to 1000000 and record the result.

4. Result

Implementation	Swap Transition (ns/transition)						Data Race Free
	10^2	MM	10^4	MM	10^6	MM	
Null	236525.12	-	6921.89	-	2145.24	-	Yes
Synchronized	253229.33	-	9688.48	-	2515.97	-	Yes
Unsynchronized	210728.86	(17!=18)	**No Output**				No
GetNSet	216433.47	-	10798.2		1617.08	-	No
BetterSafe	320176.23	-	11431.10		1281.52	-	Yes
BetterSorry	231568.65	-	8226.03		2165.20	-	No

Table: MM (mismatch) N/A(-)

Above table is the summary of the output I got when I ran each class at least 15 times in different swap transition. The swap transition is measured in ns/transition. It can be seen that as swap transition increases, the ns/transition is decreases. The program gets slower as swap transition increases because output time taken for each class increases. Among all the class, when compare the swap transition Unsynchronized has lower ns/transition at 10^2 swap while BetterSafe has lesser at 10^6 swap transition. It is because there is multiple data trying to access the number of same I and j value in the swap method, but synchronized make access very restricted because of the lock method and hence male program very slower compare to rest of the class. However, the class BeterSafe using ReentrantLock() make program more faster than synchronized because it is always correct and has no data race occur.

From the above table, it can be seen that BetterSorry is faster than BetterSage when number of transition is less than 10^6 because data races seems like lower when swap is low. Hence, BetterSorry uses the AtomicInteger makes it faster. But, BetterSafe is faster when the number of swap is larger as it has reentrantLock(). However, BetterSorry is faster and reliable, its fault rate is very low and is not data race free. But only one time BetterSorry output the "too large(7! =6)".

For Unsynchronised, it just stock when swap is 10^4. Also doing this, I get (17!=18) mismatch an average. It only run when the number of swap is below 10^4. The possibilities of the reason behind this behavior is that it may have trapped in the infinite loop while swapping with the large number of the thread. Because it is unsynchronized the operation taking place is not sequential, and threads are changing the value at the same time, which may results never get correct output for value and hence never meet the right value to get out of the loop.

5. Difficulties

Running program on the linux server in different time of the day seems inconsistent. It gave different output in different time. So, getting precise value as an output was quite challenging

6. Conclusion

Doing this project, it can be seen both class where one is reliable/accuracy but other is faster. It is found that, reliable and the program being fast doesn't come in the same plate. It's one or other, so it is hard to achieve both at the same time. Fastness may not give accurate result always.

However, doing this project there is a couple key takeaway. We can make our program synchronized or fast, reliable, correctness depends on our requirements. Sometimes we need a fast program but correctness is not important but other time other around. For complete unsynchronized method, we can take unsynchronized class. Secondly, if we want to swap less that 10^6, BeterSorry is the good option with the accuracy of 90% while BetterSafe is the best option for the large transition with 100% reliability.

7.Resource

https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReentrantLock.html

https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/atomic/AtomicIntegerArray.html

https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/atomic/AtomicInteger.html