Abstract

The goal of this assignment is to observe how different synchronization or concurrency methods impact the overall performance as the program swaps numbers and both increment and decrement elements.

Through this experiment with different classes implementing or not implementing synchronization, we will observe both the speed performance and the reliability of the program.

Background

Throughout the assignment, the program was executed in SEASnet GNU/Linux lnxsrv10 server. With the command "$ java --version", the openjdk version was openjdk 11.0.1, and other environment includes OpenJDK Runtime Environment 18.9 (build 11.0.1+13) and OpenJDK 64-Bit Server VM 18.9 (build 11.0.1+13, mixed mode).

Several classes tested in this environment had a race condition. A race condition is when two or more threads access the same data and tries to change the data simultaneously. The data-race free(DRF) classes are classes that guarantees the absence of race condition, so it will allow reliable results.

Measurements

The measurements were made with 4 different swap numbers: 10,000, 100,000, 1 million, and 10 million. Each swap was operated under 3 different thread numbers: 8, 16, and 32 threads. Each measurement is an average of 50 different results to get better precision on the result from the program. Addition to using 4 different number of swaps and 3 different number of threads, 3 different maximum values were used for better analysis. Followings are the maximum value used for testing purpose: 50, 100, and 127.

Shell scripting was used to obtain raw data that result in the analyzed result shown in Figure 1, Figure 2, Figure 3, and Figure 4. With Excel software, results were averaged to come to the value shown in the figures. Additionally, each array contains random numbers that were generated through shell scripting.

Null Class

Null class doesn't have much importance other than seeing the overhead of basic operation. However, when the number of swaps is small, it seems that the overhead is big. As the number of swaps increases, as shown in Figure 4, we could see that Null class is the fastest among all classes. This is the expected result, since Null class does not increment or decrement any elements in the array. Therefore, this class could be DRF.

Although it may be fast in speed with very large number of swaps, Null class would be useless for very obvious reason.

Synchronized State Class

Synchronized class is expected to be the slowest among all the classes. Instead of guarding only the important section of code, *synchronized* keyword is used on the whole function. *Synchronized* keyword allows to perform the operation one thread at a time, so the advantage of using multithreads does not apply in this situation. The speed performance must be worse than other classes because of macro protection it provides.

Despite its slow performance, Synchronized class is DRF. It does not allow a situation where two or more threads to enter into a section of code that changes the elements of array. Thus, one could simply use *synchronized* keyword to obtain full reliability.

Unsynchronized State Class

Unsynchronized class is a class without *synchronized* keyword and provides no protection on the code. It allows two or more threads to simultaneously change the same data, thus causing a race condition. Therefore, this is not DRF. It should be slightly faster than the synchronized class because it allows multiple thread to operate.

Test case such as *java UnsafeMemory Unsynchronized 32 1000 6 2 6 3 0 3* would result in a race condition.

GetNSet Class

GetNSet Class uses atomic array where operations can occur atomically. However, the source code is using *set()* and *get*() operations, which allows multiple threads to work on the same data. Therefore, it is not DRF since it allows race condition.

Test case such as *UnsafeMemory GetNSet 32 1000 6 2 6 3 0 3* would also fail with the race condition most of the time. Additionally, Unsynchronized State and GetNSet may result in infinite loop because of

interference from other thread when evaluating the if statement.

BetterSafe Class

BetterSafe class utilizes reentrant lock mechanism to protect the critical sections. It allows threads to grab a lock one at a time, so it is reliable. Thus, this is DRF since each thread must wait for other thread to release the lock. This class provides better speed performance than Synchronized since it does the finer grain protection rather than course grain protection the *synchronized* keyword provides. Although it may be slower than Unsynchronized method, BetterSafe provides reliability and optimized speed performance compared to Synchronized class. Thus, this class would be ideal to use especially with very large number of swap as you can see in Figure 3 and Figure 4.

In this class, the following four packages were taken in consideration: *java.util.concurrent*, *java.util.concurrent.atomic*, *java.util.concurrent.locks*, and *java.lang.invoke.VarHandle*. The first package is the parent of the second and third, so this will not be in consideration. We have seen in GetNSet that atomic integer array still allows race condition, so this is already taken out of consideration to come up with reliable program. Thus, there are two packages that were taken into consideration.

While searching what *VarHandle* package, I have found out that "The goal of VarHandle is to define a standard for invoking equivalents of java.util.concurrent.atomic and sun.misc.Unsafe operations on fields and array elements…" according to the article from *https://www.baeldung.com/java-variable-handles.* This will may result in similar situation as the *atomic* package as seen from GetNSet class. Additionally, *lock* package allows 100 percent reliability since it strictly implements the security of section of the code. Code section in between lock() and unlock() would be secured by a thread.

There would be a danger of dead locks when using locks, but the code does not contain circular dependent codes.

Challenges

There were many challenges in this assignment, but it is mainly divided into code and testing.

While coding for GetNSet and BetterSafe, I used the Synchronized class as a reference in terms of what should be in those classes. Writing the codes required some researches on the packages that I used.

While testing for the code, I came across with the unexpected results. Figure 1 and Figure 2 are the most problematic results that frustrated me the most. I was expecting Null and BetterSafe to be low. Thus, I tried changing all the variable to test it out. After many trials, I found that having a very large number of swaps would give you expected values for those classes. The reason could have been the overhead that may have to do with cache miss, but it is still not too clear. Another challenge was evaluating Unsynchronized and GetNSet. These too did not result in expected values; I was expecting values lower than of Synchronized, but always come out to be higher for all instances.

10,000 swaps (nsec/trans)				
Class\thread #	**32**	**16**	**8**	
Null	1990000	669000	184000	
Synchronized	55400	26100	10800	
Unsynchronized	53500	25900	10900	
GetNSet	90300	48100	17500	
BetterSafe	2890000	796768.3	234292.5	

Figure 1. Average time for 10,000 swaps with 4, 8, 16, and 32 threads for each class.

100,000 swaps (nsec/trans)				
Class\thread #	**32**	**16**	**8**	
Null	248057.1	81927.8	23461.2	
Synchronized	15527.9	6772.6	3191.2	
Unsynchronized	13849.8	7042.7	3697	
GetNSet	33496.5	13103.6	4183.8	
BetterSafe	299173	88710.3	25582.7	

Figure 2. Average time for 100,000 swaps with 4, 8, 16, and 32 threads for each class.

1,000,000 swaps (nsec/trans)				
Class\thread #	**32**	**16**	**8**	
Null	25002	8845.9	3209	
Synchronized	4412.7	1930.3	883	
Unsynchronized	5141.7	3684.3	1441.3	
GetNSet	14193.9	5700.3	1591.8	
BetterSafe	33015	10097.7	3565.5	

Figure 3. Average time for 1,000,000 swaps with 4, 8, 16, and 32 threads for each class.

10,000,000 swaps (nsec/trans)				
Class\thread #	**32**	**16**	**8**	
Null	1245.2	439.2	125	
Synchronized	1897.6	890.9	471	
Unsynchronized	2792.1	1046..2	462	
GetNSet	4301	1798.2	793	
BetterSafe	1447.73	699.9	417	

Figure 4. Average time for 10,000,000 swaps with 4, 8, 16, and 32 threads for each class.

References

(2018, April 04). Java 9 Variable Handles Demystified. Retrieved from https://www.baeldung.com/java-variable-handles