Homework 3 Report: Java shared memory performance races

Abstract

The objective of this project is to compare the reliability and performance of the implemented classes to figure out the best choice of the multithreading state for our startup company Ginormous Data Inc. During the process of evaluation, we would always need to consider the trade-off between accuracy and speed because the class with higher accuracy would be slower and vice versa. Basically, we would use a simple prototype that manages a data structure that represents an array of integers and implement a state transition. We would try different synchronization strategies in this project and characterize their performance.

1. Testing Platform

We test our classes on the SEASnet Linux server: linxsrv09 and the program operates under Java 11. The output of java -version gives the following information:

openjdk version "11.0.1" 2018-10-16

OpenJDK Runtime Environment 18.9 (build 11.0.1+13)

OpenJDK 64-Bit Server VM 18.9 (build 11.0.1+13, mixed mode)

We look into the file /proc/cpuinfo to find out the CPU information and notice that: the SEASnet Linux server09 has a total of 32 Intel Xeon E5-2640 v2 2.00GHz CPUs. Each processor is 8 cores, with a cache size of 20480 kB, 64 cache alignment, 46-bit physical address and 48-bit virtual address.

We also look into the file /proc/meminfo and find that the memory information for the server: the total memory is 65756948 kB, with 43777396 kB free and 62677536 kB available. The buffers are 478620 kB and the page tables are 64596 kB.

The two files would give us a lot of detailed information related to the system we use for testing.

2. Testing Results

To make the experiment outcome more meaningful, we apply the control-of variables strategy. We are using a fixed size of the state array of 5 bytes in the range form 0 to maxval (maximum of 127) and a fixed sum of value 6 in the state array. We then test the five classes with changes on the number of threads, and number of swap transitions, separately. Our number of threads are 1, 2, 4, 8, 32, respectively and our number of transitions are 1000, 10000, 100000, 1000000, respectively. We would compute the average of 10 runs for the final results of time for transition.

The testing results are as follows:

State	Number of Threads	Swaps				DRF
		10^3	10^4	10^5	10^6	
Null	1	3234.61	551.743	193.041	45.5081	Y
	4	13881.5	3445.96	885.564	421.885	
	8	30996.2	6885.73	1692.14	2367.15	
	16	68404.8	14407.6	3628.41	4267.02	
	32	190185	30165.4	7528.78	7553.12	
Synchronized	1	3752.41	665.546	222.501	66.0598	Y
	4	15403.7	5313.65	2948.67	1375.23	
	8	33812.8	10300.8	6238.66	2812.10	
	16	84831.7	22043.6	12206.6	5884.59	
	32	192488	48248.2	22820.9	13097.7	
Unsynchronized	1	3396.53	642.722	215.344	53.2678	N
	4	N/A	N/A	N/A	N/A	
	8	N/A	N/A	N/A	N/A	
	16	N/A	N/A	N/A	N/A	
	32	N/A	N/A	N/A	N/A	
GetNSet	1	6877.63	1148.99	308.365	79.1346	N
	4	N/A	N/A	N/A	N/A	
	8	N/A	N/A	N/A	N/A	
	16	N/A	N/A	N/A	N/A	
	32	N/A	N/A	N/A	N/A	
BetterSafe	1	6997.78	1084.96	291.904	89.7540	Y
	4	35027.5	7477.85	2651.93	672.42	
	8	76242.2	15740.2	5047.54	1313.06	
	16	183077	33163.3	10695.4	2894.32	
	32	417260	67826.2	20027.9	6221.47	

The results for some of the tests are not available because there exist conditions when we are stuck in infinite loops or mismatch error of the sum due to the problems of data-race freedom for certain methods of implementations. Therefore, we are not able to compute the average of the time transition in this case.

3. Reliability and Performance Analysis

From the table above, it is very clear that the Null state is faster than most of other implementations above. This is due to the fact that the class Null is a dummy implementation and does not do actual work. Null is just an implementation that returns true whenever the swap function is called. So, we do not consider the choice of Null even though it is definitely data-race free. Besides, the Null state tends to perform worse that BetterSafe state when there are more threads and swaps.

The Synchronized state is data-race free because we apply synchronization during the entire implementation by adding the keyword synchronized. The keyword synchronized in Java is responsible for preventing the thread conflict during the implementation but the whole state is not benefit from multi-threading for the swap operation as well. So, the Synchronized state is accurate but very slow.

The Unsynchronized state is not data-race free because it does not use the keywork synchronized in its implementation and generally it has no synchronization at all. Theoretically, it should be the fastest way since it does not have synchronization overhead. However, because several threads can read the same value at the same index of an array, the value written back is likely to be wrong. During our test for this class, it is always stuck in infinite loops or sometimes gives wrong answers of the sum. The command line that makes it fail can be: java UnsafeMemory Unsynchronized 8 1000000 6 5 6 3 0 3.

The GetNSet state is not data-race free because the syllabus requires us to implement it with the get and set methods of instead of using the atomic operation methods getAndDecrement and getAndIncrement. Therefore, even though we use the AtomicIntegerArray in this state, we are using non-atomic operations in the swap function, which would cause a race condition during the process of reading and writing. This class would either run into infinite loops or gives mismatch of the sum like the Unsynchronized class. Theoretically, it should be faster than Synchronized state and slower than Unsynchronized state. It is halfway between Synchronized and Unsynchronized in the way that it does not use the keyword synchronized but uses volatile accesses to array elements. The command line that makes it fail can be: java UnsafeMemory GetNSet 8 1000000 6 5 6 3 0 3.

The BetterSafe state is what we implemented for our own choice according to the syllabus. It should gain better performance than Synchronized state and maintain 100% reliability. We choose to utilize ReentrantLock in the package java.util.concurrent.locks to write this class. The reason is that this package is basically used to optimize the performance of multi-threading and the ReentrantLock we select is mainly guarantee superior performance than the keywork synchronized. Hence, we select the ReentrantLock here to lock the whole function and make it be a critical section so that we only unlock it when we return. The BetterSafe state is data-race free and is 100% reliability due to the utilization of ReentrantLock. The Java memory model also provides the semantics for the multi-threading condition. The BetterSafe method is like a JMM execution consisted of "a sequence of events for each thread, three orders on the events (program order, happens-before order, and synchronization order), and a function that assigns writes to reads". (A:5) During the test, we noticed that the BetterSafe state is more scalable because although it does not perform better than Synchronized state initially, it would gradually beat the Synchronized state when there are more threads and swaps.

4. Difficulties

The most difficult problem we encountered during this project is to figure out the implementation of the BetterSafe state. We have read a lot of API documentations to find out the ReentrantLock. To test the entire program systematically is also very challenging. We use shell scripts to make the process more convenient and apply the control-of variables strategy, which is of great significance in scientific researches to thoroughly test all the classes.

There also has been a problem when we firstly use getAndDecrement with getAndIncrement for the GetNSet method and think that GetNSet is data-race free. Afterwards, we discover that we are not alloed to use these two atomic operations in the swap function and should use get and set instead.

5. Conclusions

After running all the test cases for this assignment, we have characterized all the classes regarded its reliability and performance. We would recommend the BetterSafe method for our company. Considering the trade-off between accuracy and speed, the Unsynchronized method and the GetNSet method fail easily despite of their fast speed. The Synchronized method is accurate but too slow especially when there are a large number of threads and swaps. The BetterSafe is not only data-race free but also scalable and becomes faster when more threads and swaps are involved. Therefore, we think BetterSafe is the best choice for GDI's applications.

References

[1] Lochbihler A. Making the Java memory model safe. ACM TOPLAS 2013 Dec;35(4):12. Doi:10.1145/2518191