HW3 REPORT

Abstract

Java has lots of ways to make sure it is thread safe. However, they all have different performance. This paper aims at comparing the performance of codes that uses different thread safety approaches. The approach is to come up with similar implementations of the same task with different safety measures that ranges from no safe measure to the most conservative one, and then measure their performance and reliability. We will also use two different servers to compare their performance and reliability in different hardware conditions.

1. Hardware Conditions

I am performing my testing on two servers. The first is Lnxsrv09, with CPU model Intel(R) Xeon(R) CPU E5-2640 v2 2.00GHz. It has 32 processors, each having 8 cores. The clock rate is 2.00GHz. The second server I am using is Lnxsrv10, with CPU model Intel(R) Xeon(R) Silver 4116 CPU @ 2.10GHz. It has 4 processors, and each has 4 cores. The clock rate is 2.10GHz.

2. Test Program

The program we use for testing is straightforward. I will use a simple prototype that manages a data structure that represents an array of longs. Each array entry starts at zero. A state transition, called a swap, consists of subtracting 1 from one of the entries, and adding 1 to an entry – typically a different entry although the two entries can be the same in which case a swap does nothing. The sum of all the array entries should therefore remain zero; if it becomes nonzero, that indicates that one or more transitions weren't done correctly. The converse is not true: if the sum is zero it's still possible that some state transitions were done incorrectly. Still, this test is a reasonable way to check for errors in the simulation.

3. Different Class Implementations

3.1 Null State

An implementation of State that does nothing. Swapping has no effect. Since it doesn't do anything, the reliability should be very good. Performance should also be very good. This is used for timing the scaffolding of the simulation.

3.2 Synchronized State

An implementation of State that uses the Synchronized class so that it is safe but slow. The idea is to lock the data structure so that only one call of update could change of the data at a certain time. The reliability should be close to 100%. However, this is a very conservative synchronizing method and the performance could be very bad. It is DRF.

3.3 Unsynchronized State

Its implementation is very much like that of the Synchronized State, except that it doesn't use the Synchronized keyword before the swap method, meaning that all thread could have access to the data structure at the same time. This would make the program extremely unreliable, except when we are not using multithreading. However, the performance should be a lot better than that of the Synchronized State. It is not DRF.

3.4 Acme Safe State

The new class AcmeSafeState of that is safe without using the synchronized keyword. It made use of java.util.concurrent.atomic.AtomicLongArray to ensure thread safety. The goal is to achieve better performance than while retaining safety. According to Lea's definition of DRF in her paper, I think AcmeSafe State is DRF, because the atomic array prevents race condition from happening, therefore it doesn't have "two conflicting accesses occur without synchronization in between".

4. Reliability Test

I tested the 4 implementations above on both machines, with 100 million swap transitions, with varying array size and thread numbers. The result is: Null state, synchronized state, and acme safe state has 100% reliability. Unsynchronized state has 0% reliability when thread number is bigger than 1 while 100% when thread number is 1, no matter what the array size is. This makes sense and is in accordance with my analysis in Section 3 of this paper.

5. Performance Test

5.1 Test Preparation

I choose to use 100 million swap transitions so that there are enough swap transitions so that the results

are dominated by the actual work instead of by startup overhead.

I will test the performance on both servers, with thread numbers 1, 4, 8, and 16, and size of state array 5, 100, and 200.

I pick the average swap time (real) in ns as the benchmark for performance.

5.2 Test Results

5.2.1 Lnxsrv 09:

5.2.1.1 NullState:

time timeout 3600 java UnsafeMemory Null ThreadNumber 100000000 Arraysize

Thread Number\Array Size	5	100	1000
1	12.78	12.38	12.51
4	14.95	15.50	15.04
8	22.05	25.56	29.77
16	44.39	43.70	43.19

Average swap time(real) (ns)

5.2.1.2 SynchronizedState:

time timeout 3600 java UnsafeMemory Syncrhonized ThreadNumber 100000000 Arraysize

Thread Number\Array Size	5	100	1000
1	20.18	20.71	20.01
4	1157.60	631.94	1183.55
8	1369.98	1170.70	1652.05
16	3989.27	4456.16	5326.34

Average swap time(real) (ns)

5.2.1.3 UnsynchronizedState:

time timeout 3600 java UnsafeMemory Unsyncrhonized ThreadNumber 100000000 Arraysize

Thread Number\Array Size	5	100	1000
1	14.44	15.13	14.05
4	141.08	212.87	147.62
8	195.45	376.80	177.39
16	392.98	670.52	341.86

Average swap time(real) (ns)

5.2.1.4 AcmeSafeState:

time timeout 3600 java UnsafeMemory AcmeSafe ThreadNumber 100000000 Arraysize

Thread Number\Array Size	5	100	1000
1	24.10	24.99	25.60
4	593.89	351.98	178.45
8	426.38	321.57	203.39
16	1485.54	1084.67	805.20

Average swap time(real) (ns)

5.2.2 Lnxsrv 10

5.2.2.1 NullState:

Thread Number\Array Size	5	100	1000
1	11.41	10.76	10.73
4	14.79	13.99	14.32
8	40.50	43.62	36.42
16	98.79	127.35	98.90

Average swap time(real) (ns)

5.2.2.2 SynchronizedState:

Thread Number\Array Size	5	100	1000
1	16.76	16.91	17.04
4	484.38	362.36	363.46
8	407.10	359.23	312.46
16	797.18	728.49	650.62

Average swap time(real) (ns)

5.2.2.3 UnsynchronizedState:

Thread Number\Array Size	5	100	1000
1	12.9	12.3	12.1
4	144.1	138.3	100.53
8	282.58	277.96	180.07
16	304.36	509.87	392.00

Average swap time(real) (ns)

5.2.2.4 AcmeSafeState:

Thread Number\Array Size	5	100	1000
1	25.44	41.03	26.49
4	587.22	158.45	139.69
8	1103.42	382.66	255.56
16	2311.10	1232.50	560.57

Average swap time(real) (ns)

5.3 Performance Analysis

Generally, when thread number and array size is the same, and with the same state, Lnxsrv09 has a better performance than lnxsrv10, which makes sense since

it has a significantly more processors. However, lnxsrv10 has a better performance when we are using the unsynchronized state. Assuming synchronized is implemented by putting locks on that segment of code, I think it's because since lnxsrv09 has more cores, more failing attempts to get the lock would happen, therefore wasting more CPU time on that.

Generally, on the same server and with the same array size and state, the more threads we use, the worse the performance would be, which doesn't make sense. My guess is it's because the transition count is not big enough to even out the extra overhead caused by using multithreading.

Generally, on the same server and with the same thread number, array size doesn't make any impact on the performance. Except when the state we use is AcmeSafe and we are using more than 4 cores, when the performance gets better when the array size is larger, which doesn't make a whole lot of sense either. I am guessing it's because larger array size evens out the reads and writes on each element.

As for the performance of each state, when I am using lnxsrv09, just as I predicted, the Nullstate is the fastest, and Unsynchronized states comes after that, and then AcmeSafeState comes after that, and SynchronizedsState comes last. AcmeSafeState is faster than SynchronizedState because it doesn't lock the whole section of code, therefore more threads could work at the same time.

However, when I am using lnxsrv10, it turns out that AcmeSafeState is actually slower than SynchronizedState, and maybe it has to do with its CPU architecture.

6. Problems

I didn't really have to overcome any major problems when doing this assignment. I would say it's a pretty straightforward homwork. One thing is that in the AcmeSafestate class the current() function should return an array of long although the data structure I used in the class is AtomicLongArray, because it implements the interface of state, which I didn't notice at first. Other than that, it's all good.

7. Conclusion

After careful analysis and testing, we find AcmeTestState to be reliable (as well as DRF), and has the best performance out of all the DRF options.