CS131 Homework 3 Report

Bradley Zhu 304627529

Performance Measurement (ns/transition)

2 Threads

# of Operations	1000	10000	100000	1000000
Synchronized	6394.97	1682.27	846.336	391.014
Unsynchronized	6334.08	1189.23	687.501	97.3504
GetNSet	12723.2	3525.34	1063.25	365.513
BetterSafe	11393.3	2341.54	1077.58	440.952

4 Threads

# of Operations	1000	10000	100000	1000000
Synchronized	12817.5	4280.70	2233.73	1351.97
Unsynchronized	12064.7	3764.35	1344.02	462.514
GetNSet	27762.4	7128.83	2452.72	1380.35
BetterSafe	26269.7	7027.53	2477.14	1223.12

8 Threads

# of Operations	1000	10000	100000	1000000
Synchronized	25786.5	9350.38	4975.74	2794.64
Unsynchronized	22154.9	6201.03	2091.49	1464.16
GetNSet	54265.2	13112.5	4503.34	2729.80
BetterSafe	49294.8	13283.4	4842.11	2694.68

16 Threads

# of Operations	1000	10000	100000	1000000
Synchronized	53879.6	16593.6	9550.92	5276.64
Unsynchronized	55975.7	15759.1	NA	NA
GetNSet	103214	26217.2	8038.09	5592.19
BetterSafe	114969	27166.8	8118.51	4398.21

Abstract

The point of this lab was to explore different locking mechanisms and their relative speeds using different numbers of threads. We tested this using swapping and increments and decrements.

Environment

I used the SEASnet server 6 running Java version 9.0.1, Java(TM) SE Runtime Environment (build 9.0.1+11), Java HotSpot(TM) 64-Bit server VM (build 9.0.1+11, mixed mode). The linux servers have 16 4-core CPUs. The CPU info is
Intel(R) Xeon(R) CPU E5620 @ 2.40 GHz with a cache size of 12288 KB and 32 GB of memory.

Testing

I tested using 2, 4, 8, and 16 threads running 1000, 10000, 100000, and 1000000 operations. I set my max value to be 50 and set the initial values of the array to be 25. I did 3 tests per table entry and averaged the results. I did all the test for Synchronized first, then Unsynchronized,
then GetNSet, then BetterSafe.

Analysis of Results

My BetterSafe achieved better performance than Synchronized when there were more threads and more operations. When there was less operations it did worse I think because of the overhead involved in setting up, locking, and unlocking the locks.

Potential Sources of Error

I think the environment changed significantly as I was doing my tests. I regret testing my stuff in the order of the class instead of by thread/operation number, because when I was testing all my BetterSafe, the server may have had more load than when I was testing all of my Synchronized making my results not as accurate. By the end of my tests, all my stuff was returning longer times. Another potential source of error is low sample size. I only averaged 3 test cases, which is not very many. If I had more time, I would have averaged 20 test cases.

Comparison of Classes

Unsynchronized - This was obviously the fastest throughout my trials as there was nothing to slow it down. All I did was remove the keyword Synchronzied from SynchronizedState. This class would occasionally get stuck due to race conditions.

Synchronized - This did a lot better than I thought it would, and it beat out GetNSet and BetterSafe for low number of operations consistently. This was provided for me.

GetNSet - I used AtomicIntegerArray to implement this using their methods. This was surprisingly slow, slower than Synchronized. This class was not DRF because we only ensure atomicity on the get and set separately, but not in between them. This would also frequently mess up and get stuck occasionally.

BetterSafe - I used the ReentrantLock to implement this in swap, making this DRF. This class barely outperformed Synchronized with large number of threads and operations, and lost out with a few number of threads and operations. I believe this is because Synchronized is already very optimized for Java.

Difficulties

I had difficulty understanding why my program would sometimes get stuck with increased thread numbers and operations. It would solve with a print statement which did not make logical sense to me. Eventually I figured out this was due to if a swap not being

successful, the for loop would occasionally not increment and we would get stuck in an infinite loop.

Best choice for GDI's Applications

I would use BetterSafe for GDI's applications because I assume they will have a large number of threads as well as a large number of operations. I presume BetterSafe's small advantage will only continue to increase in those conditions.

Implementation of Bettersafe

The pros of concurrent are that they are reliable and not too slow. Atomic would be technically slower than the Reentrant lock and also not completely DRF, but errors would be unlikely. Still, over a large number of tests, they inevitably pop up, but GDI doesn't care too many about a small number of errors. I ended up using the reentrant lock because it was reliable and also quick.

According to IBM, ReentrantLock offers better performance than synchronized under heavy contention, and it was faster than using the atomicIntegerArray because the AtomicIntegerArray would require some internal locking mechanism causing overhead with arrays of atomic objects. That is why I chose ReentrantLock over other options.

DRF

My classes that were nonDRF are unsynchronized, and GetNSet.

They are all likely to fail
java UnsafeMemory [Unsynchronized / GetNSet] 16 10000000 50 25 25 25 25 25 25