Ryan Trihernawan

904-063-131

April 27, 2015

Homework 3

Environment:

15 processors, 4 cores, Intel(R) Xeon(R) CPU E5620 @ 2.40GHz, 33GB Ram, 64 bit

Red Hat Enterprise Linux Server release 6.6 (Santiago), java version "1.8.0_45"

My BetterSafe implementation is faster than Synchronized because it uses ReentrantLock, which is a low-level implementation and that I only, locked the section of the code that decrements and increments the values.

My BetterSorry implementation is faster than BetterSafe because it uses AtomicInteger array, which allows for race conditions unlike BetterSafe.

BetterSorry is more reliable than Unsynchronized because it still ensures that the increment and decrement of the value is atomic by converting the byte to AtomicInteger.

BetterSorry will suffer from a race condition that does a large number of swaps because based on the test on SeasNet, the program always hangs when the test is performed on a large number of swaps (> 1000).

The test below will make BetterSorry fail.

java UnsafeMemory BetterSorry 8 10000 6 5 6 3 0 3.

Value1 = 5 6 3 0 3

Performance	NullState	UnynchronizedState	BetterSorry	GetNSet	BetterSafe	SynchronizedState
8 threads 1,000 swaps maxval = 6 Value1	36885.3 ns	41609.1 ns	43278.4 ns	52826.6 ns	62363.3 nss	38787.5 ns

32 threads 1,000 swaps maxval = 6 Value1	258908 ns	252305 ns	268692 ns	304491 ns	375167 ns	293230 ns

Due to SEASnet's disappointing performance, my test cases only work for up to 1000 swaps for all the states except BetterSafe and SynchronizedState. Using swaps over a million will show that BetterSafe is indeed faster than SynchronizedState.

Reliability	NullState	UnynchronizedState	BetterSorry	GetNSett	BetterSafe	SynchronizedState
8 threads 1,000 swaps maxval = 6 Value1	100%	70%	48%	39%	100%	100%
32 threads 1,000 swaps maxval = 6 Value1	100%	65%	47%	41%	100%	100%

Overall, for performance,

NullState > UnsynchronizedState > BetterSorry > GetNSet > BetterSafe > SynchronizedState

For reliability, it is the other way around.