Homework 4. Java shared memory performance races

Jingyuan Fan
904186250

Data-race Free

The only method to avoid such collision is to make sure that the whole read/modify/write process is atomic. Otherwise, operations executed by one thread may mix up with other operations by other threads, and the result is hard to predict and the program is hard to debug since end result is nondeterministic and depends on the relative timing between interfering threads. In this project, only Synchronized and BetterSafe achieve data-race free, since the whole operations of load the value of i, increment its value and write it back to the array are locked/synchronized, so that when one thread is doing such operations, other threads have to wait (the same for element j). Other methods only have atomically action for single operation. That's why they can achieve better reliability than UnSynchronized method while not have 100% reliability.

Performance Comparison

I used volatile for BetterSorry, AtomicIntegerArray for GetNSet and ReentrantLock for BetterSafe. The reason why BetterSafe can be much faster than Synchronized while realizing 100% reliability is that lock is a low-level operation while is able to prevent other threads reading or writing to the critical section at the same time the thread, which got the lock, takes operations. On the other hand, although both volatile and AtomicIntegerArray uses volatile access to the array, there operations are not mutual exclusive, e.g., between one read and write operation executed by one thread, other threads can also take read and write operations and probably updates the same elements, causing errors. Here sometimes the programs halts, it's not due to deadlock, which I already paid attention to during programming. The reason is that sometimes some elements in the array might be zero, and the transitions cannot continue.

	Null	UnSynchronized	BetterSorry	GetNSet	BetterSafe	Synchronized
Avg. time (ns/thread)	151.119	1,095	1,130	3,070	1,810	6,075

(16 threads, 10,000,000 nTransitions, and the array equals [10 20 30 40 50 60])

Reliability Comparison:

Since the original swap operation consists of subtracting 1 from one of the positive integers in the array, and adding 1 to an integer in that array, we cannot possibly understand whether a specific read and store operation is successful, especially when multiple threads run at the same time, because these operations are not mutually exclusive. And what's worse, someone cannot measure the reliability by taking the ratio of actual result over expected sum, since the sum can keep increasing or decreasing arbitrarily. So I changed the code to add 1 to both element i, j in swap operation, so the

final result would be the number of correct operations that are not influenced by other operations, then get the reliability by dividing the correct operations over the expected sum. By using this method, I also avoid element being subtracted to zero, so the program would not halt. From the chart below, we can see that BetterSafe and Synchronized can achieve 100% reliability, and GetNSet is between Unsynchronized and Synchronized while BetterSorry is much better than GetNSet when there are 32 threads and 100,000,000 transitions, and almost the same when there are 16 threads and 10,000,000 threads.

		Null	UnSynchronized	BetterSorry	GetNSet	BetterSafe	Synchronized
1st	Avg. time (ns/thread)	152.58	1,041.10	1,098.92	2,659.83	1,775.47	5,088.76
	Reliability		56.2%	58.4%	57.8%	100%	100%
2nd	Avg. time (ns/thread)	158.083	1,543.49	1,573.73	4,075.78	2,766.60	9,705.5
	Reliability		31.7%	82.7%	37.3%	100%	100%

(the 1st test: 16 threads, 10,000,000 nTransitions, and the array equals [10 20 30 40 50 60], the 2nd test: 32 threads, 100,000,000 nTransitions, and the array equals [10 20 30 40 50 60])

The operations are performed on SEASnet GNU/Linux servers with 16 processors, each of which has Intel(R) Xeon(R) CPU E5620@2.40GHz with 4 cores, total memory is 3GB. JVM: Java HotSpot(TM) 64-Bit Server VM (build 24.51-b03, mixed mode) Java: Java(TM) SE Runtime Environment (build 1.7.0_51-b13).

Interestingly, the results are different from what my laptop gives.

		Null	UnSynchronized	BetterSorry	GetNSet	BetterSafe	Synchronized
1st	Avg. time (ns/thread)	204.618	308.043	442.749	993.882	1,202.63	1,515.50
	Reliability		58.6%	78.1%	61.3%	100%	100%
2nd	Avg. time (ns/thread)	305.226	573.318	712.644	1,600.02	2,100.40	2,788.72
	Reliability		54.6%	79.2%	57.3%	100%	100%

The processer is 1.7GHz Intel Core i7, memory is 8GB 1600 MHZ DDR3, and JVM is the same as the SEASnet servers.

Suggestion for GDI

BetterSorry would be the best choice, since it gives a good balance between reliability and average time per transition.