CS 131 HW3 Report

Environment:

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)
MemTotal: 65758072 kB
server: lnxsrv09
32 processors
Intel(R) Xeon(R) CPU E5-2640 v2 @ 2.00GHz
8 cpu cores

File Description:

UnsynchronizedState.java:
This is based on the SynchronizedState.java but removes the synchronized keyword.
GetNSetState.java:
This class uses the AtomicIntegerArray. Due to the input format of the array, we need to convert the whole array to an integer array and also convert it back. It also calls two atomic functions: get, set.
BetterSafe.java:
4 choices available:
Package 1: java.util.concurrent
Build some advanced tools such as thread pools, concurrent queues, and so on.
The main APIs:
Executor: Executable to a specific task
Future: is the interface to run with Runnable, Callable, such as the result of a thread after the end of the implementation of the results, etc., also provides a cancel termination thread.
BlockingQueue: blockingqueue.
Advantages:
i. Simple to use.
ii. Powerful and standardized library
iii: Performance has been optimized, concurrent with a large number of non-blocking algorithm behaves good.
Disadvantages:
i. Sometimes too complicated to deal with.
Package 2: java.util.concurrent.atomic
This package provides a set of atomic variable classes. The basic characteristic is that in a multithreaded environment, when there are multiple threads at the same time the implementation, when a thread into the method, it will not be interrupted by other threads, and other threads like a spin lock, until is completed, then the JVM will select another thread from the waiting queue to enter.
Advantages:
i. Simple to use.
ii. Efficient.
Disadvantages:
i. Need to do conversion if the input type is not defined.
Package 3: java.util.concurrent.locks
Advantages:
i: Simple to use.
ii: Fairness.
iii. Throughput performance is always good.
Disadvantages:
i. Need to manually handle locks. Will cause troubles if forget to release locks.
Package 4: java.lang.invoke
Behave similar to java.concurrent.atomic.
Advantages:
i. Efficient.
Disadvantages
i. Hard to understand and use.
My Implementation:
I decided to use ReentrantLock provided by java.util.concurrent.locks. The throughput performance is better than "synchronized".ReentrantLock is an implementation with Lock that has the same basic behavior and semantics as implicit monitor locks (using synchronized methods and statement access), but it has the ability to extend. As an additional gain, in the competitive conditions, ReentrantLock implementation than the current synchronized to achieve more scalable. (It is possible to improve the competitive performance of synchronized in future versions of the JVM.) This means that when many threads are competing for the same lock, the throughput of using ReentrantLock is usually better than that of the game. In other words, when many threads try

to access ReentrantLock protected shared resources, the JVM will spend less time scheduling threads, and more time to execute the thread. Although java.lang.invoke.VarHandle is also good but is easy to use and since our program is not that complicate, reentrantlock can provide good enough performance.

Test Result:

1. Based on 8 threads, 10000 transitions.
array value: 11 22 33 44 55 66 77 88
max value: 127

Class	Threads(ns/transition)
Null	7402.32
Unsynchronized	5020.53 (sum mismatch)
Synchronized	8689.57
GetNSet	15845.9 (sum mismatch)
BetterSafe	9351.6

2. Based on 16 threads, 10000 transitions.
array value: 11 22 33 44 55 66 77 88
max value: 127

Class	Threads(ns/transition)
Null	11887.0
Unsynchronized	13364.9 (sum mismatch)
Synchronized	17819.9
GetNSet	29814.1 (sum mismatch)
BetterSafe	21739.3

3. Based on 8 threads, 100000 transitions
array value: 11 22 33 44 55 66 77 88

Class	Threads(ns/transition)
Null	1885.91
Unsynchronized	2582.49 (sum mismatch)
Synchronized	1610.92
GetNSet	infinite (sum mismatch)
BetterSafe	1281.97

Comparisons:

Reliability:
Before test, according to my understanding, I think
both GetNSet and Unsynchronized are not 100% reliable and others are 100% valid. This is because during the time when thread 1 just got the decremented number, maybe another thread 2 interrupts thread 1. In this case, the number that the thread 1 get is higher than it should get and thus the result will be higher than it should be.
In both GetNSet and Unsynchronized, when we decrease the number, since it is not atomic operation, it will have the situation described above which results to sum mismatch.

Speed:
Before test, according to my understanding, I thought the threads average performance from high to low will be: Null, Unsynchronized, BetterSafe, GetNSet, Synchronized. (Ignore the sum mismatch)
As we can see, if the swap times are really big, the result is more likely as what I expected, so I am more interested in it.

Why BetterSafe implementation is faster than Synchronized and why it is still 100% reliable:
BetterSafe is more efficient at a high number of swaps compared to others. Because there is an overhead for running BetterSafe. The

synchronized keyword locks the whole function while BetterSafe just lock the part that we think is important so it runs quickly but maintains reliability meanwhile.

DRF:
Synchronized and BetterSafe are data-race free because they lock the code that do swapping. Unsynchronized is not DRF since there is no lock operation. We could test it using command ”java UnsafeMemory Unsynchronized 8 10000 127 11 22 33 44 55 66 77 88”. GetNSet is not DRF is because no lock prevent interruption and thus if one thread if doing comparison while another thread performs swap, it will also cause problems.