CS131, Spring 2018
Homework 3 Report

1 Abstract

Atomic integers and locks implemented by the `synchronous` keyword in Java incur time penalties under various conditions. This report examines the effects of different methods in swapping, incrementing, and decrementing integers in an array, with regards to managing data-races. Tests were conducted on *lnxsrv06* on SEASnet, which allowed up to 40 concurrent threads.

2 Background

Java utilizes a number of memory modes that differ with respect to fault tolerance: plain, opaque, release/acquire, and volatile[1]. Classes in `java.utils .concurrent.locks` and `java.utils .concurrent.atomic` use these modes to achieve levels of granularity suitable for different conditions. We examine at least four methods of integer swapping, and whether they are data-race free (DRF).

The baseline for comparison is `Null State`, which does not swap, such that it times the scaffolding of the simulation.

`SynchronizedState` introduces method-level synchronization, which internally implements a monitor lock[2]. Threads must *acquire* an intrinsic lock to access a member. Since only one thread may own a monitor[3] at any given time, other threads are suspended and queued until the monitor is *released*. This may greatly slow down the transitions, especially when there is high contention, and it is compounded by the coarse granularity of method-level synchronization. However, it is guaranteed to be DRF.

`UnsynchronizedState` eliminates synchronization, returning the mode back to *plain*. The lack of memory management reduces overhead, increasing speed if the test completes. However, data-races not only incur error, but may also cause race conditions that continue infinitely. For testing purposes, we put a hard upper bound on allowable running time. Lack of any synchronization caused errors in all tests with more than 1 thread.

`GetNSetState` uses the class `Atomic IntegerArray` from the `atomic` package, which guarantees atomicity using the compare-and-swap instruction set. However, only the `get()` and `set()` methods are used to simulate an array of *volatile* integers, which checks for updates on each access. This guarantees that the latest values are always read. In this case, incrementing and decrementing still take multiple operations, so this is not necessarily DRF; however, no errors occurred in tests. It can be surmised that an error should eventually occur under higher contention. Compared to synchronization, atomic integers do not implement any locks; therefore, it takes much less time per transition.

Our implementation of `BetterSafe` is uses `ReentrantLock`, although a number of methods have been explored. We determined, after much thought, that it would be either impossible or complicated to achieve atomicity in multiple updates by using `getAndUpdate()` or `compareAndSet()`, or `LongAdder`, of the `atomic` package.

`ReentrantLock` is used by `synchronized` so it provides finer granularity, such that it could be wrapped around specific lines. Its tendency to throw errors on contention forces the use of `try, catch, finally` blocks, which decrease code readability. Although there are many options to fine-tune these locks, such as providing fairness, timeouts, and interrup- tibility[4], they aren't necessarily used in this experiment to maintain low complexity. Although concurrency is concentrated over only a few lines of code, such that the finer granularity cannot fully taken advantage of, its subtle behavior differences does bring about 3 times faster speed under high contention. The primary advantage is that, like `synchronized`, it ensures 100% reliability.

3 Methodology and Results

Tests are performed using a modified test harness. All exiting statements are removed to allow multiple tests to be run at once. For each class, tests are run with $[1, 8, 24, 40]$ threads, $[1e5, 1e6, 5e6]$ transitions, $[31, 63, 127]$ maximum value, and $[5, 50, 500]$ vector sizes. The results are piped into logs, and automated with a Makefile. This streamlines the process and reduces repetition. 6 logs are produced with 144 lines each.

The tests are run usually very early in the morning, at about 3:00AM, on *lnxsrv06*, which allows 40 threads. *lnx srv09* stalls out at 24 threads. There is no particular issue in testing, due to foresight in creating the alternate test and reporting harness. The only issue is perhaps the high variance in output. For example, `GetNSet` under high contention gives nearly 4000ns in one trial, and 774ns in another. Numbers more closely aligned with the average are used.

An abbreviated result is shown below, with these test conditions (in ns):

(A) 40 threads, 5e6 transitions, 127 maximum value, 500 vector size (max setting).

(B) 24 threads, 5e6 transitions, 31 maximum value, 5 vector size (high contention).

(C) 8 threads, 1e5, 127 maximum value, 500 vector size (low contention).

Model	DRF	A	B	C
Null	Yes	136	52	65
Sync	Yes	8260	4933	1990
Unsync	No	1503	178	788
GetNSet	No	1366	774	912
BetterSafe	Yes	2441	1409	2246

Table 1: Model performance.

4 Conclusion

`BetterSafe` is half as fast as `GetNSet`, which almost matches `Unsynchronized`, except in high contention. However, it is more than 3 times faster than `Synchronized` and still 100% reliable. Therefore, it is the recommended solution since it is sufficiently fast without sacrificing functionality.

5 References

[1] Leas, Doug. "Using JDK 9 Memory Order Modes". *State University of New York, Oswego*, 25 Mar. 2018. http://gee.cs.oswego.edu/dl/html/j9mm.html

[2] "Intrinsic Locks and Synchronization". *Oracle*, 2017. https://docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html

[3] "Synchronized in Java". *Geeks for Geeks*. https://www.geeksforgeeks.org/synchronized-in-java/

[4] Javin, Paul. "ReentrantLock Example in Java, Difference between synchronized vs ReentrantLock". *Java Revisited*, 7 Mar, 2017. http://javarevisited.blogspot.com/2013/03/reentrantlock-example-in-java-synchronized-difference-vs-lock.html