

76% completed



Search Course

Multithreading in Java

- Atomic Assignments
- Thread Safety & Synchronized
- Wait & Notify
- Interrupting Threads
- Volatile
- Reentrant Locks & Condition Variables
- Missed Signals
- Semaphore in Java
- Spurious Wakeups
- Atomic Classes
- More on Atomics
- Non-blocking Synchronization
- Miscellaneous Topics

Java's Memory Model

Interview Practice Problems

Practice Mock Interview

Java Multithreading for Senior Engineering Interviews / ... / Non-blocking Synchronization

Non-blocking Synchronization

Learn about non-blocking synchronization and thread-safe data structures that can be built using them.

Introduction

Much of the performance improvements seen in classes such as `Semaphore` and `ConcurrentLinkedQueue` versus their synchronized equivalents come from the use of atomic variables and non-blocking synchronization. Non-blocking algorithms use machine-level atomic instructions such as compare-and-swap instead of locks to provide data integrity when multiple threads access shared resources. Non-blocking algorithms are harder to design and implement but out perform lock-based alternatives in terms of liveness and scalability. As the name suggests, non-blocking algorithms don't block when multiple threads contend for the same data, and as a consequence greatly reduce scheduling overhead. These algorithms don't suffer from deadlocks, liveness issues and individual thread failures. More formally:

- An algorithm is called non-blocking if the failure or suspension of a thread doesn't cause the failure or suspension of another thread.
- An algorithm is called lock free if at every step of the algorithm some thread participant of the algorithm can make progress.

Nonblocking counter

Designing a thread-safe counter would require using locks so that threads don't step over each other. An increment operation on a `long` variable consists of three steps. Fetching the current value, incrementing it and then writing it back. All three have to be executed atomically to achieve thread-safety. A lock-based implementation of the lock appears below:

```
1 class LockBasedCounter {
2     private long value;
3
4     public synchronized long getValue() {
5         return value;
6     }
7
8     // When multiple threads attempt to invoke the method at the same time,
9     // only one is allowed to do so while the rest are suspended
10    public synchronized void increment() {
11        value++;
12    }
13 }
```

Locking comes with its baggage and we can design a counter that doesn't cause threads to be suspended in the face of contention. To build such a counter, we'll need the hardware to support the CAS instruction. For our purposes we'll write a class `SimulatedCAS` that imitates the CAS instruction. The listing for this class along with comments appears in the widget below:

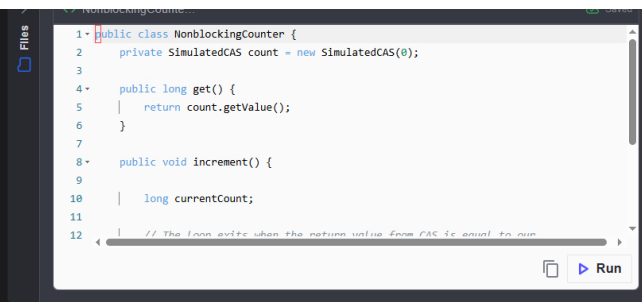
```
1 public class SimulatedCAS {
2
3     // Let's assume for simplicity our value is a long
4     private long value = 0;
5
6     // constructor to initialize the value
7     public SimulatedCAS(long initValue) {
8         value = initValue;
9     }
10
11    synchronized long getValue() {
12        return value;
13    }
14
15    // The synchronized keyword causes all the steps in this method to execute
16    // atomically, which is akin to simulating the compare and swap processor
17    // instruction. The behavior of the function is as follows:
18    //
19    // 1. Return the expectedValue if the CAS instruction completes successfully, i.e.
20    //    the newValue is written.
21    // 2. Return the current value if the CAS instruction doesn't complete successfully
22    //
23    // The method is setup such that when expectedValue equals the return value
24    // the caller can assume success.
25    synchronized long compareAndSwap(long expectedValue, long newValue) {
```

The compare and swap (CAS) functionality is also implemented by some processors as a pair of instructions *loadlinked* and *store-conditional*. Using the `SimulatedCAS` class we can now construct a non-blocking counter that doesn't block threads when multiple of them attempt to manipulate shared state. The listing for the class `NonBlockingCounter` has comments describing the operation of the counter:

Java

Press to interact





```
1- public class NonBlockingCounter {
2-     private SimulatedCAS count = new SimulatedCAS(0);
3-
4-     public long get() {
5-         return count.getValue();
6-     }
7-
8-     public void increment() {
9-
10-         long currentCount;
11-
12-         // The loop exits when the return value from CAS is equal to its own
13-     }
```

CAS-based counter performance

The performance of CAS instruction varies across processors and architectures and though it may seem that a CAS-based counter may perform poorly in comparison to a lock-based counter, the reality is otherwise. In practice CAS-based locks outperform lock-based counters when there is no contention (as the thread doesn't go through the process of acquiring a lock) and often when there is low to moderate contention. Acquiring a lock usually involves at least one CAS operation and peripheral lock-related housekeeping tasks, which implies more work is done by a lock-based counter in the best case of zero contention compared to a CAS-based counter.

Conclusion

There are well-known non-blocking algorithms for commonly used data structures such as hashtables, priority queues, stacks, linked-lists etc. However, designing non-blocking algorithms is much more complex than lock-based alternatives. Generally, non-blocking algorithms skirt many of the vices associated with lock-based approaches such as deadlocks or priority inversion, however, threads participating in a non-blocking algorithm can still experience starvation or livelocks as they may perform repeated retries without success.

[← Back lesson](#)

[Completed](#) [Next →](#)

[More on Atomics](#)

[Miscellaneous Topics](#)