Atomic Variables & Nonblocking Synchronization

A Locking Counter

```java
public final class Counter {
    private long value = 0;
    public synchronized long getValue() {
        return value;
    }

    public synchronized long increment() {
        return ++value;
    }
}
```
Java.util.concurrent Performance

- Many java.util.concurrent classes perform better then synchronized alternatives. Why?
  - Atomic variables & nonblocking synchronization
- We’ve already talked about atomic variables.
- Nonblocking algorithms are concurrent algorithms that derive their thread safety from low-level atomic hardware primitives (not locks).

Disadvantages of Locking

- When a thread fails to acquire lock it can be suspended
  - Context switching & resumption can be expensive
- When waiting for a lock thread can’t do anything
- If thread holding lock is delayed, no thread that needs that lock can progress
  - Priority inversion: low priority thread has lock needed by a high priority thread
- Caveat: contention, rather than locking, is the real issue. YMMV
Hardware Support

• Locking is pessimistic
  – If contention is infrequent, most locking was unneeded
• In an earlier we talked about optimistic trying
  – Proceed with the update
  – Check for collision
  – If update fails, can retry
• Modern processors support atomic operations

Compare and Swap (CAS)

• CAS has 3 operands
  – Memory location V, Old value A, New value B
• Atomically updates V to value B, but only if current value is A
• If multiple threads try to update V only one succeeds
  – but the losers don’t get punished with suspension
• Losers can just try again
Simulated (CAS)

```java
public class SimulatedCAS { // not implemented this way!
    private int currValue;
    public synchronized int get() {return currValue;}
    public synchronized int compareAndSwap ( int expectedValue, int newValue) {
        if (currValue == expectedValue)
            currValue = newValue;
        return currValue ;
    }
    public synchronized boolean compareAndSet(int expectedValue, int newValue) {
        return (expectedValue == compareAndSwap(expectedValue, newValue));
    }
}
```

A Nonblocking Counter

```java
public class NonblockingCounter {
    private AtomicInteger value;
    public int getValue() {
        return value.get();
    }
    public int increment() {
        int v;
        do {
            v = value.get();
        }while (!value.compareAndSet(v, v + 1));
        return v + 1;
    }
}
```
Atomic Variables

- Generalization of volatile variables
- Allows atomic read-modify-write operations without intrinsic locking
- Scope of contention limited to a single variable
- Faster than locking -- no scheduling impact
- Like volatiles, can’t synchronize two atomic vars
- Doesn’t support atomic check-then-act sequences

Updating Complex Objects

- Example: Want to manage two related variables
- Can’t do that with volatiles
- Idiom: turn compound update into single update
 CasNumberRange

// INARIANT: lower <= upper
private static class IntPair {
    final int lower, upper;
    public IntPair(int lower, int upper) {....}
    public void setLower(int i) {....}
    public void setUpper(int i) {....}
}

 CasNumberRange

public class CasNumberRange {
    private final AtomicReference<IntPair> values =
        new AtomicReference<IntPair>(new IntPair(0, 0));
    public int getLower() {return values.get().lower;}
    public int getUpper() {return values.get().upper;}
    public void setLower(int i) {
        while (true) {
            IntPair oldv = values.get();
            if (i > oldv.upper) throw new IllegalArgumentException();
            IntPair newv = new IntPair(i, oldv.upper);
            if (values.compareAndSet(oldv, newv)) return;
        }
    }
    // setUpper() similar to setLower()
}
Performance Comparison

• Will show two implementations of a pseudo-random number generator (PRNG)
  – One uses locks: ReentrantLockPseudoRandom.java
  – One is nonblocking: AtomicPseudoRandom.java
• PRNG issues
  – Next value based on last value, so you need to remember last value
• How do lock-based and non-lock-based implementations compare?

ReentrantLockPseudoRandom

```java
public class ReentrantLockPseudoRandom extends PseudoRandom {
    private final Lock lock = new ReentrantLock(false);
    private int seed;
    ReentrantLockPseudoRandom(int seed) {this.seed = seed;}
    public int nextInt(int n) {
        lock.lock();
        try {
            int s = seed;  seed = calculateNext(s);  int remainder = s % n;
            return remainder > 0 ? remainder : remainder + n;
        } finally {
            lock.unlock();
        }
    }
}
```
public class AtomicPseudoRandom extends PseudoRandom {
    private AtomicInteger seed;
    AtomicPseudoRandom(int seed) {this.seed = new AtomicInteger(seed);}
    public int nextInt(int n) {
        while (true) {
            int s = seed.get();
            int nextSeed = calculateNext(s);
            if (seed.compareAndSet(s, nextSeed)) {
                int remainder = s % n;
                return remainder > 0 ? remainder : remainder + n;
            }
        }
    }
}

Atomic Updates / Lock Updates

![Graph showing Atomic Updates / Lock Updates](image-url)
Nonblocking Algorithms

- No locks
- Stopping one thread will not prevent global progress
  - Immune to deadlock
  - Starvation possible
- Writing correct nonblocking algs is very hard!

Nonblocking Algorithm Flavors

- Wait-Free
  - All threads complete in finite count of steps
  - Low priority threads cannot block high priority threads
- Lock-Free
  - Every successful step makes global progress
  - Individual threads may starve; priority inversion possible
  - No live-lock
- Obstruction-Free
  - A single thread in isolation completes in finite count of steps
  - Threads may block each other; live-lock possible
  - Example: optimistic retry
Nonblocking Stack

- See: ConcurrentStack.java & SynchStack.java

```java
public class ConcurrentStack <E> {

    private static class Node <E> {
        public final E item;
        public Node <E> next;
        public Node (E item) {
            this.item = item;
        }
    }

    AtomicReference<Node <E>> top = new AtomicReference<Node <E>>();

    public void push(E item) {
        Node <E> newHead = new Node <E> (item);
        Node <E> oldHead;
        do {
            oldHead = top.get();
            newHead.next = oldHead;
        } while (!top.compareAndSet(oldHead, newHead));
    }
}
```
public E pop() {
    Node<E> oldHead; Node<E> newHead;
    do {
        oldHead = top.get();
        if (oldHead == null)
            return null;
        newHead = oldHead.next;
    } while (!top.compareAndSet(oldHead, newHead));
    return oldHead.item;
}
Overview of Michael & Scott Approach

- Make sure queue is always in consistent state
- Threads should know whether another operation is already in progress
  - Thread B can wait for thread A to finish before starting
- Prevents corruption, but late thread can fail if early thread fails

Overview of Michael & Scott Approach

- If thread B arrives while operation in progress for thread A, let B finish update for A
  - Then B can progress without waiting for A
  - If A finds some of its work done, it doesn’t repeat. It just skips doing it itself
Michael & Scott Nonblocking Queue

• Queue with two elements in quiescent state

Michael & Scott Nonblocking Queue

• Queue in intermediate state during insertion
  – After the new element is added but before the tail pointer is updated
Michael & Scott Nonblocking Queue

- Queue in quiescent state again after the tail pointer is updated

- Observation: if tail.next is non-null, then an operation is in progress
- If a thread finds an operation in progress, it will try to advance tail to return queue to stable state
  - Then it will reload tail and repeat process
public class ConcurrentQueue <E> {
    private static class Node <E> {
        final E item;
        final AtomicReference<Node<E>> next;
        public Node(E item, Node<E> next) {
            this.item = item;
            this.next = new AtomicReference<Node<E>>(next);
        }
    }
    private final Node<E> dummy = new Node<E>(null, null);
    private final AtomicReference<Node<E>> head = new AtomicReference<Node<E>>(dummy);
    private final AtomicReference<Node<E>> tail = new AtomicReference<Node<E>>(dummy);
}

public boolean put(E item) {
    Node<E> newNode = new Node<E>(item, null);
    while (true) {
        Node<E> curTail = tail.get();
        Node<E> tailNext = curTail.next.get();
        if (curTail == tail.get()) { // did tail change?
            if (tailNext != null) { // Queue in intermediate state, advance tail
                curTail.next.compareAndSet(null, newNode);
            } else { // In quiescent state, try inserting new node
                if (curTail.next.compareAndSet(null, newNode)) {
                    // Insertion succeeded, try advancing tail
                    curTail.next.compareAndSet(null, newNode);
                    return true;
                }
            }
        }
    }
}
ConcurrentQueue

public E take() {
    for (;;) {  // Keep trying until take is done
        Node<E> oldHead = head.get();  // get current head
        Node<E> oldTail = tail.get();  // get current tail
        Node<E> oldHeadNext = oldHead.next.get();  // get current head.next
        if (oldHead == head.get()) {  // Are head, tail, and next consistent?
            if (oldHead == oldTail) {  // Queue empty or tail being updated?
                if (oldHeadNext == null) {  // Is queue empty?
                    return null;  // Queue is empty, can't take
                } else {  // No need to deal with tail
                    tail.compareAndSet(oldTail, oldHeadNext);  // tail updated, try to advance it
                    if (head.compareAndSet(oldHead, oldHeadNext))
                        return oldHeadNext.item;
                }
            } else {  // No need to deal with tail
                return null;  // Queue is empty, can't take
            }
        }
    }
}