Atomic Variables
&
Nonblocking Synchronization

Fall 2008

A Locking Counter

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 than synchronized alternatives. Why?
  – Atomic variables and 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
• When a thread is waiting for a lock it can’t do anything else
• If a thread that holds a lock is delay, then no thread that needs that lock can progress
  – Can lead to priority inversion
• Locking can be heavyweight
Hardware Support

• Locking is pessimistic
• 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 in current value is A
• When multiple threads try to update V only one succeeds, but the losers don’t get punished with suspension
• Losers can try again
Simulated (CAS)

```java
public class SimulatedCAS { // not implemented this way!
    private int value;
    public synchronized int get() { return value; }
    public synchronized int compareAndSwap(int expectedValue, int newValue) {
        int oldValue = value;
        if (oldValue == expectedValue) value = newValue;
        return oldValue;
    }
    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 because there’s no scheduling impact
- Like volatiles, can’t synchronize two atomic vars
- Doesn’t support atomic check-then-act sequences

CasNumberRange

// local to CasNumberRange
private static class IntPair { // INVARIANT: lower <= upper
    final int lower;
    final int upper;

    public IntPair(int lower, int upper) {
        this.lower = lower; this.upper = upper;
    }
}
}
`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 psuedo-random number generator (PRNG)
  - One using locks, one nonblocking
- 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?
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;
            }
        }
    }
}

Nonblocking Algorithms

- Nonblocking: failure or suspension of one thread can’t cause failure or suspension of any other thread
  - Immune to deadlock
- Lock-free: at least one thread makes progress at every step
  - Avoids global livelock
- Wait-free: each thread makes progress at every step
  - Avoids thread starvation
- Writing correct nonblocking/lock-free/wait-free algorithms is very hard!
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 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));  
    }  
    }
A Nonblocking Queue

• Rule of thumb– limit change to one variable
• Harder for a Queue because we need to update head and tail

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 arrive 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

![Diagram of a nonblocking queue during insertion]

Michael & Scott Nonblocking Queue

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

![Diagram of a nonblocking queue in quiescent state]
Michael & Scott Nonblocking Queue

- 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

LinkedQueue

```java
public class LinkedQueue<E> {
    private static class Node<E> {
        final E item;
        final AtomicReference<LinkedQueue.Node<E>> next;
        public Node(E item, LinkedQueue.Node<E> next) {
            this.item = item;
            this.next = new AtomicReference<LinkedQueue.Node<E>>(next);
        }
    }
    private final LinkedQueue.Node<E> dummy = new LinkedQueue.Node<E>(null, null);
    private final AtomicReference<LinkedQueue.Node<E>> head
        = new AtomicReference<LinkedQueue.Node<E>>(dummy);
    private final AtomicReference<LinkedQueue.Node<E>> tail
        = new AtomicReference<LinkedQueue.Node<E>>(dummy);
```
public boolean put(E item) {
    LinkedQueue.Node<E> newNode = new LinkedQueue.Node<E>(item, null);
    while (true) {
        LinkedQueue.Node<E> curTail = tail.get();
        LinkedQueue.Node<E> tailNext = curTail.next.get();
        if (curTail == tail.get()) { // did tail change?
            if (tailNext != null) {
                // Queue in intermediate state, advance tail
                tail.compareAndSet(curTail, tailNext);
            } else {
                // In quiescent state, try inserting new node
                if (curTail.next.compareAndSet(null, newNode)) {
                    // Insertion succeeded, try advancing tail
                    tail.compareAndSet(curTail, newNode); // will fail if tail already moved
                    return true;
                }
            }
        }
    }
}