Atomic Variables & Nonblocking Synchronization

CMSC 433
Fall 2014
Michael Hicks
(with some slides due to Rance Cleaveland)
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 & 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
  – Can result in 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 earlier lecture we discussed **optimistic** trying
  - Proceed with the update
  - Check for collision
  - If update fails, retry
- Processor can use atomic operations to support optimistic trying
Compare and Swap (CAS)

- CAS has 3 operands
  - Memory location V, expected 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
  - They can just try again
**Simulated CAS**

```java
public class SimulatedCAS { // not implemented this way!
    private int value;

    public synchronized int get() { return currValue; }

    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));
    }
}
```

Critical observation: uses `==` here to compare old and new values; not `.equals()` method call!
// demonstrates the use of CAS

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
- In general, doesn’t support atomic check-then-act sequences
Atomic Variable Classes

• AtomicInteger
• AtomicLong
• AtomicBoolean
• AtomicReference<T>
  – set(), get() operations
  – getAndSet() atomically sets to new value, returns old value
  – Boolean compareAndSet(int expect, int new)
  – Arithmetic (increment, decrement) as appropriate
    • getAndAdd – return old value
    • addAndGet – return new value
    • getAndIncrement
    • incrementAndGet

• Also AtomicIntegerArray, AtomicLongArray, AtomicReferenceArray
  – Each element in array supports atomic operations
Updating Complex Objects

- Example: Want to manage two related variables
  - Can’t do this with volatiles
- Idiom: turn compound update into single update
// INVARIANT: lower <= upper
// How do you make this thread-safe?

private static class IntPair {
    final int lower, upper;
    public IntPair(int lower, int upper) {....}
    public void setLower(int i) {....}
    public void setUpper(int i) {....}
}
public class CasNumberRange {
    // IntPair is a pair of Integers
    private final AtomicReference<IntPair> values =
        new AtomicReference<IntPair>(new IntPair(0, 0));

    public void setLower(int i) {
        while (true) {
            IntPair oldv = values.get(); // gets the current value atomically
            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?
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 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
Comparing Performance

The diagram compares the performance of atomic and synchronous operations across different numbers of threads. The x-axis represents the number of threads, ranging from 2 to 8, while the y-axis shows the number of updates. The chart illustrates that the atomic operation performs better than the synchronous operation as the number of threads increases. Additional details include:

- **Machine Specifications:** MacPro, OS X Snow Leopard, 8 cores, 8 GB of RAM
Nonblocking Algorithms

• No locks
• Stopping one thread will not prevent global progress
  – Immune to deadlock
  – But: Starvation is possible
• Writing correct nonblocking algorithms is very hard!
Nonblocking Stack

• See: ConcurrentStack.java & SynchStack.java
• Follows the same pattern of optimistic retries that we have seen already
  – Push:
      • create the new head of the stack;
      • try to set it, but
      • fail and retry if the old head has changed
  – Pop:
      • get reference to current head;
      • attempt to set new head to be current.next, but
      • fail and retry if the head has changed
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 ConcurrentStack <E> {

    private static class Node <E> {
        public final E item;   public Node<E> next; …
    }

    AtomicReference<Node<E>> top = …

    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;
    }
}
A Nonblocking Queue

- Rule of thumb—limit change to one variable
- Harder for a Queue because we need to update both head and tail
  - But CAS only works on one item
- See: SynchQueue.java & ConcurrentQueue.java
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
Michael & Scott Nonblocking Queue

- Key observation:
  - An operation is in progress if tail.next is non-null
- If a thread finds an operation in progress, it can try to advance the 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);
}

ConcurrentQueue
public class ConcurrentQueue<E> {
    private static class Node<E> {
        final E item; final AtomicReference<Node<E>> next;
    }
    private final Node<E> dummy = …
    private final AtomicReference<Node<E>> head = …
    private final AtomicReference<Node<E>> tail = …
    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()) { // assuming tail did not change …
                if (tailNext != null) { // If queue in intermediate state, advance tail
                    tail.compareAndSet(curTail, tailNext); // then retry
                } else { // If queue 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; // done!
                    }
                }
            }
        }
    }
}
public E take() {
    for (;;) {
        Node<E> oldHead = head.get();
        Node<E> oldTail = tail.get();
        Node<E> oldHeadNext = oldHead.next.get();
        if (oldHead == head.get()) {
            if (oldHead == oldTail) {
                if (oldHeadNext == null)
                    return null;
                tail.compareAndSet(oldTail, oldHeadNext);
            } else {
                tail.compareAndSet(oldTail, oldHeadNext);
            }
        } else {
            if (head.compareAndSet(oldHead, oldHeadNext))
                return oldHeadNext.item;
        }
    }
    return null;
}
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 int. 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
                }
            }
        } else { // In quiescent state, try inserting new node
            // Insertion succeeded, try advancing tail
            tail.compareAndSet(curTail, newNode);
            // will fail if tail already moved
        }
        return true;
    }
}

<table>
<thead>
<tr>
<th>Var</th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>tailNext</td>
<td></td>
<td></td>
</tr>
<tr>
<td>curTail</td>
<td></td>
<td></td>
</tr>
<tr>
<td>newNode</td>
<td></td>
<td></td>
</tr>
<tr>
<td>head</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tail</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 int. 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;
                }
            }
        }
    }
}
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 int. 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;
}
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 int. 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;
                }
            }
        }
    }
}
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 int. 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;
                }
            }
        }
    }
}
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 int. 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;
                }
            }
        }
    }
}

<table>
<thead>
<tr>
<th>Var</th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>tailNext</td>
<td>λ</td>
<td>λ</td>
</tr>
<tr>
<td>curTail</td>
<td></td>
<td></td>
</tr>
<tr>
<td>newNode</td>
<td></td>
<td></td>
</tr>
<tr>
<td>head</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tail</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 int. 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;
            }
        }
    }
}
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 int. 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;
                }
            }
        }
    }
}
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 int. 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;
    }
}
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 int. 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;
                }
            }
        }
    }
}
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 int. 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;
                }
            }
        }
    }
}
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 int. 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;
                }
            }
        }
    }
}
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 int. 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;
            }
        }
    }
}
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 int. 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;
                }
            }
        }
    }
}
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 int. 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
                }
            }
        } else {
            tail.compareAndSet(curTail, newNode);
            return true;
        }
    }
}
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 int. 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;
    }
}
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 int. 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;
                }
            }
        }
    }
}
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 int. 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;
                }
            }
        }
    }
}
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 int. 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;
                }
            }
        }
    }
}
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 int. 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;
                }
            }
        }
    }
}
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 int. 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;
                }
            }
        }
    }
}
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 int. 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;
    }
}
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 int. 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;
                }
            }
        }
    }
}
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 int. 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
                }
            }
        } else {
            return true;
        }
    }
}
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 int. 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;
                }
            }
        }
    }
}
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 int. 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;
                }
            }
        }
    }
}
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 int. 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;
}
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 int. 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;
    }
}
public boolean put(E item) {
    Node<E> newNode = new Node<>(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 int. 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;
                }
            }
        }
    }
}
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 int. 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;
                }
            }
        }
    }
}
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 int. 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;
                }
            }
        }
    }
}
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 int. 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;
                }
            }
        }
    }
}
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 int. 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;
            }
        }
    }
}
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 int. 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;
            }
        }
    }
}
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 int. 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;
                }
            }
        }
    }
}
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 int. state, advance tail
                tail.compareAndSet(curTail, tailNext);
                // Insertion succeeded, try advancing tail
                tail.compareAndSet(curTail, newNode);
                // will fail if tail already moved
                return true;
            }
        } 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;
            }
        }
    }
}
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 int. 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;
}
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 int. 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;
}
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 int. 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;
                }
            }
        }
    }
}
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 int. 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;
}
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 int. 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;
}
Trace 2: T2 continues

class Node {
    E item;
    Node next;
}

class Queue {
    Node head;
    Node tail;
}

public boolean put(E item) {
    Node newTail = new Node(item, null);
    while (true) {
        Node curTail = tail.get();
        Node tailNext = curTail.next.get();
        if (curTail == tail.get()) {// did tail change?
            if (tailNext != null) { // Queue in int. 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
                }
            }
        } else {
            return true;
        }
    }
    return true;
}
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 int. 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;
                }
            }
        }
    }
}
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 int. 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;
    }
}
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 int. 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
                } else return true;
            }
        }
    }
}
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 int. 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;
                }
            }
        }
    }
}
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 int. 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;
}