More on concurrency

April 22\textsuperscript{nd}, 2010
Earth Day

William Pugh
Housekeeping

- Cookies: times added
- Project 8 questions?
  - Will allow 10 minutes at end of class
  - Get mostly finished by Tuesday, come prepared with questions
Topics

- Thread attributes
- Thread locals
- Fair locking, context switching
- Atomic operations
- Nonblocking synchronization
Thread priority

• Each thread can have a priority
  – Must be set before thread is started
  – 1 (min) to 10 (max)
  – Generally, higher priority threads are given more CPU time
  – But very unspecified
    • Lot more specifics for real time Java
Priority inversion

- Three threads, of low, medium and high priorities
- Low priority thread grabs a lock
- High priority thread tries to grab same lock, can’t because another thread holds it
- Medium priority thread is happy to consume all the CPU cycle
  - Preventing low priority thread from getting enough cycles to make progress and release lock
Priority inheritance

- Done in real time systems
- If you hold a lock, and a higher priority thread wants the lock
- You inherit the other thread’s priority until you release the lock
Daemon threads

• A daemon thread is one that only exists to serve other threads
  – **Must be set before thread is started**

• If you ever get to a point where only daemon threads are running, the JVM shuts down
  – **No other difference**
Thread Locals

- Easy and efficient way to avoid synchronization conflicts
  - `ThreadLocal<Random> myRandom = new ThreadLocal<Random>() {`  
    `protected synchronized Random initialValue() {`  
    `return new Random(); }`  
  
- Call `myRandom.get().nextInt(100)`  
  - Creates a Random for this thread if it doesn’t already have one  
  - Calls `nextInt(100)` on it
Blocking and Locks

- Sometimes you need locks and blocking
- Cost of obtaining uncontended locks generally not an issue
  - If it is, you are doing it wrong
- It is expensive to bench a thread
  - Gets pulled out of the CPU, another thread displaces its information in CPU caches, OS has to get involved to both move the thread to the bench and to move it off the bench
Fair locks

• Many of the java.util.concurrent classes offer fair variants
  – Fair means first come, first served
  – Good for latency, bad for throughput

• With fair locks, if two threads are simultaneously performing lots of updates to a shared structure
  – They have to take turns, getting benched between each turn
  – More efficient to just let a thread sit on the bench for a while
Obtaining locks

- Lock implementations done via compare and swap
- If lock doesn’t succeed, typically lock implementation will just spin for a few microseconds
- If that doesn’t work, set up structures so that the lock knows which threads need to be woken up, and then go sit on the bench and wait to be woken up
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
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
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;
    }
}
A Nonblocking Counter

// 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;
    }
}
Review of 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
- Have to plan for failure, perhaps by retrying
- Like volatiles, can’t perform atomic operations on to separate values
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 Algorithms

- No locks
- Stopping one thread will not prevent global progress
  - Immune to deadlock
  - Starvation is possible
- Writing correct nonblocking algorithms 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
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;
}
public void push(E item) {
    Node<E> nh = new Node<E>(item);
    Node<E> oh;
    do {
        oh = top.get();
        nh.next = oh;
    } while (!top.compareAndSet(oh, nh));
}

public E pop() {
    Node<E> oh;  Node<E> nh;
    do {
        oh = top.get();
        if (oh == null)
            return null;
        nh = oh.next;
    } while (!top.compareAndSet(oh, nh));
    return oh.item;
}
Nonblocking Stack

• See: ConcurrentStack.java & SynchStack.java
A Nonblocking Queue

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

- Observation: if tail.next is non-null, then a put 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
                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 E take() {
    for (;;) {
        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()) {  // has another take happened?
            if (oldHead == oldTail) {  // Queue empty or tail being updated?
                if (oldHeadNext == null) {  // Is queue empty? If yes,
                    return null;
                } else {  // No need to deal with tail
                    tail.compareAndSet(oldTail, oldHeadNext);  // tail needs update. Try to advance it
                    if (head.compareAndSet(oldHead, oldHeadNext))
                        return oldHeadNext.item;
                }
            }
        }
    }
    return null;
}
Execution Traces

- 2 threads attempt to put()}
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 { // Queue in int. state, advance tail
            tail.compareAndSet(curTail, tailNext);
        }
    }
    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
                }
            }
        } 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<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.equals(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<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
                }
            }
        } 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
                }
            }
        } 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;
                }
            }
        }
    }
}
```java
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;
                }
            }
        }
    }
}
```
```java
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
                }
            }
        } 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<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;
                }
            }
        } else {
            // Queue in int. state, advance tail
            tail.compareAndSet(curTail, tailNext);
        }
    }
}
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 { // 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;
                }
            }
        }
    }
    return true;
}

<table>
<thead>
<tr>
<th>Var</th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>tailNext</td>
<td></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
                }
            }
        } else {
            return true;
        }
    }
}
Execution Traces

• 2 threads attempt to take()
public E take() {
    for (; ; ) {
        Node<E> oldHead = head.get(); // current head
        Node<E> oldTail = tail.get(); // current tail
        Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
        if (oldHead == head.get()) {  //head, tail, and next changed?
            if (oldHead == oldTail) {   //Queue empty or tail 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 updating
                    } else {
                    // No need to deal with tail
                if (head.compareAndSet(oldHead, oldHeadNext)) {
                    return oldHeadNext.item;
                }
            }
        }
    }
}
public E take() {
    for (;;) {
        Node<E> oldHead = head.get();                        // current head
        Node<E> oldTail = tail.get();                            // current tail
        Node<E> oldHeadNext = oldHead.next.get(); / / curr head.next
        if (oldHead == head.get()) {      //head, tail, and next changed?
            if (oldHead == oldTail) {       //Queue empty or tail updated?
                if (oldHeadNext == null) { // Is queue empty?
                    return null;                         //Queue is empty, can't take
                }
                return oldHeadNext.item; // Queue is empty, can't take
            } else {                                         // No need to deal with tail
                if (head.compareAndSet(oldHead, oldHeadNext)) {
                    return oldHeadNext.item;
                }
            }
        } else {                                         // No need to deal with tail
            tail.compareAndSet(oldTail, oldHeadNext); // tail updating
        } else {                                         // No need to deal with tail
            return null;                         //Queue is empty, can't take
        }
    }
}
public E take() {
    for (;;) {
        1 Node<E> oldHead = head.get(); // current head
        2 Node<E> oldTail = tail.get(); // current tail
        3 Node<E> oldHeadNext = oldHead.next.get(); / / curr head.next
        4 if (oldHead == head.get()) { //head, tail, and next changed?
            5 if (oldHead == oldTail) { //Queue empty or tail updated?
                6 if (oldHeadNext == null) { // Is queue empty?
                    return null; //Queue is empty, can't take
                }
                7 else {
                    tail.compareAndSet(oldTail, oldHeadNext); // tail updating
                }
            } else { // No need to deal with tail
                8 if (head.compareAndSet(oldHead, oldHeadNext)) {
                    return oldHeadNext.item;
                }
            }
        }
    }
}
public E take() {
    for (;;) {
        Node<E> oldHead = head.get();                        // current head
        Node<E> oldTail = tail.get();                            // current tail
        Node<E> oldHeadNext = oldHead.next.get(); / / curr head.next
        if (oldHead == head.get()) {      //head, tail, and next changed?
            if (oldHead == oldTail) {       //Queue empty or tail updated?
                if (oldHeadNext == null) { // Is queue empty?
                    return null;                         //Queue is empty, can't take
                }
                tail.compareAndSet(oldTail, oldHeadNext); // tail updating
            } else {                                         // No need to deal with tail
                if (head.compareAndSet(oldHead, oldHeadNext)) {
                    return oldHeadNext.item;
                }
            }
        }
    }
}
public E take() {
for (; ; ) {
    Node<E> oldHead = head.get();  // current head
    Node<E> oldTail = tail.get();   // current tail
    Node<E> oldHeadNext = oldHead.next.get();  // curr head.next
    if (oldHead == head.get()) {  //head, tail, and next changed?
        if (oldHead == oldTail) { //Queue empty or tail updated?
            if (oldHeadNext == null) { // Is queue empty?
                return null;  //Queue is empty, can't take
            }
            tail.compareAndSet(oldTail, oldHeadNext);  // tail updating
        } else {                 // No need to deal with tail
            if (head.compareAndSet(oldHead, oldHeadNext)) {
                return oldHeadNext.item;
            }
        }
    }
}
public E take() {
    for (;;) {
1       Node<E> oldHead = head.get();              // current head
2       Node<E> oldTail = tail.get();               // current tail
3       Node<E> oldHeadNext = oldHead.next.get();  // curr head.next
4       if (oldHead == head.get()) {              //head, tail, and next changed?
5           if (oldHead == oldTail) {             //Queue empty or tail updated?
6               if (oldHeadNext == null) {       // Is queue empty?
7                   return null;                   //Queue is empty, can't take
8               }
9               tail.compareAndSet(oldTail, oldHeadNext); // tail updating
10           } else {                            // No need to deal with tail
11               if (head.compareAndSet(oldHead, oldHeadNext)) {
12                   return oldHeadNext.item;
13               }
14           }
15       }
16    }
17}
public E take() {
    for (;;) {
        1 Node<E> oldHead = head.get(); // current head
        2 Node<E> oldTail = tail.get(); // current tail
        3 Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
        4 if (oldHead == head.get()) { //head, tail, and next changed?
            5 if (oldHead == oldTail) { //Queue empty or tail updated?
                6 if (oldHeadNext == null) { // Is queue empty?
                    return null; //Queue is empty, can't take
                } else {
                    // No need to deal with tail
                    9 tail.compareAndSet(oldTail, oldHeadNext); // tail updating
                }
            } else {
                // No need to deal with tail
                9 if (head.compareAndSet(oldHead, oldHeadNext)) {
                    return oldHeadNext.item;
                }
            }
        }
    }
}
public E take() {
    for (;;) {
        Node<E> oldHead = head.get(); // current head
        Node<E> oldTail = tail.get(); // current tail
        Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
        if (oldHead == head.get()) { //head, tail, and next changed?
            if (oldHead == oldTail) { //Queue empty or tail 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 updating
                }
            } else { // No need to deal with tail
                if (head.compareAndSet(oldHead, oldHeadNext)) {
                    return oldHeadNext.item;
                }
            }
        }
    }
}
public E take() {
    for (;;) {
        Node<E> oldHead = head.get(); // current head
        Node<E> oldTail = tail.get(); // current tail
        Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
        if (oldHead == head.get()) { //head, tail, and next changed?
            if (oldHead == oldTail) { //Queue empty or tail updated?
                if (oldHeadNext == null) { // Is queue empty?
                    return null; //Queue is empty, can't take
                }
                tail.compareAndSet(oldTail, oldHeadNext); // tail updating
            } else { // No need to deal with tail
                if (head.compareAndSet(oldHead, oldHeadNext)) {
                    return oldHeadNext.item;
                }
            }
        } else {
            return oldHeadNext.item;
        }
    }
}
public E take() {
    for (; ; ) {
        Node<E> oldHead = head.get();                       // current head
        Node<E> oldTail = tail.get();                        // current tail
        Node<E> oldHeadNext = oldHead.next.get();           // curr head.next
        if (oldHead == head.get()) {                         //head, tail, and next changed?
            if (oldHead == oldTail) {                        //Queue empty or tail updated?
                if (oldHeadNext == null) {                   // Is queue empty?
                    return null;                             //Queue is empty, can't take
                }
            } else {                                        // No need to deal with tail
                if (head.compareAndSet(oldHead, oldHeadNext)) {
                    return oldHeadNext.item;
                }
            }
        }
        tail.compareAndSet(oldTail, oldHeadNext);         // tail updating
    }
}
public E take() {
    for (;;) {
        1 Node<E> oldHead = head.get(); // current head
        2 Node<E> oldTail = tail.get();  // current tail
        3 Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
        4 if (oldHead == head.get()) { //head, tail, and next changed?
            5 if (oldHead == oldTail) { //Queue empty or tail updated?
                6 if (oldHeadNext == null) { // Is queue empty?
                    7 return null;           //Queue is empty, can't take
                } else {                 // No need to deal with tail
                    tail.compareAndSet(oldTail, oldHeadNext); // tail updating
                }
            } else {                   // No need to deal with tail
                if (head.compareAndSet(oldHead, oldHeadNext)) {
                    9 return oldHeadNext.item;
                }
            }
        }
    }
}
public E take() {
    for (;;) {
        Node<E> oldHead = head.get();  // current head
        Node<E> oldTail = tail.get();   // current tail
        Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
        if (oldHead == head.get()) {   //head, tail, and next changed?
            if (oldHead == oldTail) {   //Queue empty or tail updated?
                if (oldHeadNext == null) { // Is queue empty?
                    return null;           //Queue is empty, can't take
                }
                tail.compareAndSet(oldTail, oldHeadNext); // tail updating
            } else { // No need to deal with tail
                if (head.compareAndSet(oldHead, oldHeadNext)) {
                    return oldHeadNext.item;
                }
            }
        }
        return null;
    }
}
public E take() {
    for (;;) {
        1 Node<E> oldHead = head.get(); // current head
        2 Node<E> oldTail = tail.get(); // current tail
        3 Node<E> oldHeadNext = oldHead.next.get(); / / curr head.next
        4 if (oldHead == head.get()) { //head, tail, and next changed?
            5 if (oldHead == oldTail) { //Queue empty or tail updated?
                6 if (oldHeadNext == null) { // Is queue empty?
                    return null; //Queue is empty, can't take
                } else {
                    // No need to deal with tail
                    9 if (head.compareAndSet(oldHead, oldHeadNext)) {
                        return oldHeadNext.item;
                    }
                }
            } else {
                tail.compareAndSet(oldTail, oldHeadNext); // tail updating
            }
        }
    }
}
public E take() {
    for (;;) {
        Node<E> oldHead = head.get();                        // current head
        Node<E> oldTail = tail.get();                            // current tail
        Node<E> oldHeadNext = oldHead.next.get(); / / curr head.next
        if (oldHead == head.get()) {      //head, tail, and next changed?
            if (oldHead == oldTail) {       //Queue empty or tail 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 updating
                }
            } else {                                         // No need to deal with tail
                if (head.compareAndSet(oldHead, oldHeadNext)) {
                    return oldHeadNext.item;
                }
            }
        }
    }
}
public E take() {
    for (;;) {
        Node<E> oldHead = head.get();                        // current head
        Node<E> oldTail = tail.get();                            // current tail
        Node<E> oldHeadNext = oldHead.next.get(); / / curr head.next
        if (oldHead == head.get()) {      //head, tail, and next changed?
            if (oldHead == oldTail) {       //Queue empty or tail 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 updating
                }
            } else {                                         // No need to deal with tail
                if (head.compareAndSet(oldHead, oldHeadNext)) {
                    return oldHeadNext.item;
                }
            }
        }
    }
}
public E take() {
    for (;;) {
        Node<E> oldHead = head.get(); // current head
        Node<E> oldTail = tail.get(); // current tail
        Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
        if (oldHead == head.get()) { //head, tail, and next changed?
            if (oldHead == oldTail) { //Queue empty or tail 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 updating
                }
            } else { // No need to deal with tail
                head.compareAndSet(oldHead, oldHeadNext); //tail updating
            }
        } else { // No need to deal with tail
            return oldHeadNext.item;
        }
    }
}
public E take() {
    for (;;) {
        Node<E> oldHead = head.get();                        // current head
        Node<E> oldTail = tail.get();                        // current tail
        Node<E> oldHeadNext = oldHead.next.get(); / / curr head.next
        if (oldHead == head.get()) {      //head, tail, and next changed?
            if (oldHead == oldTail) {       //Queue empty or tail 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 updating
                    return oldHeadNext.item;
                }
            }
        
        if (head.compareAndSet(oldHead, oldHeadNext)) {
            return oldHeadNext.item;
        }
    }
}
public E take() {
    for (;;) {
        Node<E> oldHead = head.get(); // current head
        Node<E> oldTail = tail.get(); // current tail
        Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
        if (oldHead == head.get()) { // head, tail, and next changed?
            if (oldHead == oldTail) { // Queue empty or tail updated?
                if (oldHeadNext == null) { // Is queue empty?
                    return null; // Queue is empty, can't take
                } else { // tail updating
                    tail.compareAndSet(oldTail, oldHeadNext);
                }
            } else { // No need to deal with tail
                if (head.compareAndSet(oldHead, oldHeadNext)) {
                    return oldHeadNext.item;
                }
            }
        }
    }
}
public E take() {
    for (; ;) {
        Node<E> oldHead = head.get();                           // current head
        Node<E> oldTail = tail.get();                             // current tail
        Node<E> oldHeadNext = oldHead.next.get();                // curr head.next
        if (oldHead == head.get()) {                             //head, tail, and next changed?
            if (oldHead == oldTail) {                            //Queue empty or tail updated?
                if (oldHeadNext == null) {                        // Is queue empty?
                    return null;                                  //Queue is empty, can't take
                }
                tail.compareAndSet(oldTail, oldHeadNext);       // tail updating
            } else {                                            // No need to deal with tail
                if (head.compareAndSet(oldHead, oldHeadNext)) {
                    return oldHeadNext.item;
                }
            } else {
                return null;
            }
        }
    }
}
public E take() {
    for (;;) {
        Node<E> oldHead = head.get();                        // current head
        Node<E> oldTail = tail.get();                            // current tail
        Node<E> oldHeadNext = oldHead.next.get(); / / curr head.next
        if (oldHead == head.get()) {      //head, tail, and next changed?
            if (oldHead == oldTail) {       //Queue empty or tail updated?
                if (oldHeadNext == null) { // Is queue empty?
                    return null;                         //Queue is empty, can't take
                }
            }
            else {                                         // No need to deal with tail
                if (head.compareAndSet(oldHead, oldHeadNext)) {
                    return oldHeadNext.item;
                }
            }
        }
        tail.compareAndSet(oldTail, oldHeadNext); // tail updating
    }
}
public E take() {
    for (; ; ) {
        Node<E> oldHead = head.get();                       // current head
        Node<E> oldTail = tail.get();                        // current tail
        Node<E> oldHeadNext = oldHead.next.get(); / / curr head.next
        if (oldHead == head.get()) {      //head, tail, and next changed?
            if (oldHead == oldTail) {       //Queue empty or tail updated?
                if (oldHeadNext == null) { // Is queue empty?
                    return null;                         //Queue is empty, can't take
                }
                tail.compareAndSet(oldTail, oldHeadNext); // tail updating
            } else {                                         // No need to deal with tail
                if (head.compareAndSet(oldHead, oldHeadNext)) {
                    return oldHeadNext.item;
                }
            }
        }
    }
}
public E take() {
    for (;;) {
        Node<E> oldHead = head.get(); // current head
        Node<E> oldTail = tail.get(); // current tail
        Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
        if (oldHead == head.get()) { //head, tail, and next changed?
            if (oldHead == oldTail) { //Queue empty or tail 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 updating
                }
            } else { // No need to deal with tail
                if (head.compareAndSet(oldHead, oldHeadNext)) {
                    return oldHeadNext.item;
                }
            }
        }
    }
}
public E take() {
    for (;;) {
        Node<E> oldHead = head.get();       // current head
        Node<E> oldTail = tail.get();       // current tail
        Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
        if (oldHead == head.get()) {      //head, tail, and next changed?
            if (oldHead == oldTail) {       //Queue empty or tail 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 updating
                }
            } else { // No need to deal with tail
                if (head.compareAndSet(oldHead, oldHeadNext)) {
                    return oldHeadNext.item;
                }
            }
        }
    }
}
public E take() {
    for (; ; ) {
        1 Node<E> oldHead = head.get();              // current head
        2 Node<E> oldTail = tail.get();               // current tail
        3 Node<E> oldHeadNext = oldHead.next.get();  // curr head.next
        4 if (oldHead == head.get()) {                //head, tail, and next changed?
            5 if (oldHead == oldTail) {                //Queue empty or tail updated?
                6 if (oldHeadNext == null) {          // Is queue empty?
                    7 return null;                   //Queue is empty, can't take
                }
            } else {                                // No need to deal with tail
                tail.compareAndSet(oldTail, oldHeadNext); // tail updating
            }
        } else {                                     // No need to deal with tail
            8 if (head.compareAndSet(oldHead, oldHeadNext)) {
                9 return oldHeadNext.item;
            }
        }
    }
}
public E take() {  
    for (; ;) {  
1     Node<E> oldHead = head.get();  // current head  
2     Node<E> oldTail = tail.get();  // current tail  
3     Node<E> oldHeadNext = oldHead.next.get();  // curr head.next  
4     if (oldHead == head.get()) {  //head, tail, and next changed?  
5         if (oldHead == oldTail) {  //Queue empty or tail updated?  
6             if (oldHeadNext == null) {  // Is queue empty?  
7                 return null;  //Queue is empty, can't take  
8             } else {  // No need to deal with tail  
9                 tail.compareAndSet(oldTail, oldHeadNext);  // tail updating  
10             }  
11         } else {  // No need to deal with tail  
12             if (head.compareAndSet(oldHead, oldHeadNext)) {  
13                 return oldHeadNext.item;  
14             }  
15         }  
16     }  
17    }  
18}
public E take() {
   for (;;) {
      Node<E> oldHead = head.get(); // current head
      Node<E> oldTail = tail.get(); // current tail
      Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
      if (oldHead == head.get()) { //head, tail, and next changed?
         if (oldHead == oldTail) { //Queue empty or tail updated?
            if (oldHeadNext == null) { // Is queue empty?
               return null; //Queue is empty, can't take
            }
            tail.compareAndSet(oldTail, oldHeadNext); // tail updating
         } else { // No need to deal with tail
            if (head.compareAndSet(oldHead, oldHeadNext)) {
               return oldHeadNext.item;
            }
         }
      }
   }
}
public E take() {
    for (; ;) {
        Node<E> oldHead = head.get(); // current head
        Node<E> oldTail = tail.get(); // current tail
        Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
        if (oldHead == head.get()) { //head, tail, and next changed?
            if (oldHead == oldTail) { //Queue empty or tail 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 updating
                }
            } else { // No need to deal with tail
                if (head.compareAndSet(oldHead, oldHeadNext)) {
                    return oldHeadNext.item;
                }
            }
        }
    }
}
public E take() {
  for (;;) {
    // current head
    Node<E> oldHead = head.get();
    // current tail
    Node<E> oldTail = tail.get();
    // curr head.next
    Node<E> oldHeadNext = oldHead.next.get();
    if (oldHead == head.get()) { //head, tail, and next changed?
      if (oldHead == oldTail) { //Queue empty or tail 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 updating
      }
    } else { // No need to deal with tail
      if (head.compareAndSet(oldHead, oldHeadNext)) {
        return oldHeadNext.item;
      }
    }
  }
}
public E take() {
    for (; ; ) {
        Node<E> oldHead = head.get(); // current head
        Node<E> oldTail = tail.get(); // current tail
        Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
        if (oldHead == head.get()) { //head, tail, and next changed?
            if (oldHead == oldTail) { //Queue empty or tail 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 updating
                }
            } else { // No need to deal with tail
                if (head.compareAndSet(oldHead, oldHeadNext)) {
                    return oldHeadNext.item;
                }
            }
        }
    }
}
public E take() {
    for (;;) {
        Node<E> oldHead = head.get();                          // current head
        Node<E> oldTail = tail.get();                            // current tail
        Node<E> oldHeadNext = oldHead.next.get();               // curr head.next
        if (oldHead == head.get()) {                            //head, tail, and next changed?
            if (oldHead == oldTail) {                           //Queue empty or tail 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 updating
            }
        } else {                                               // No need to deal with tail
            if (head.compareAndSet(oldHead, oldHeadNext)) {    //head, tail, and next changed?
                return oldHeadNext.item;
            }
        }
    }
}

<table>
<thead>
<tr>
<th>Var</th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>oldHead</td>
<td></td>
<td>T2</td>
</tr>
<tr>
<td>oldTail</td>
<td></td>
<td>T2</td>
</tr>
<tr>
<td>oldHeadNext</td>
<td></td>
<td>T2</td>
</tr>
</tbody>
</table>

1 2 \lambda
public E take() {
    for (;;) {
        Node<E> oldHead = head.get(); // current head
        Node<E> oldTail = tail.get(); // current tail
        Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
        if (oldHead == head.get()) { //head, tail, and next changed?
            if (oldHead == oldTail) { //Queue empty or tail 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 updating
                    break; // At the head of the queue
                }
            } else { // No need to deal with tail
                if (head.compareAndSet(oldHead, oldHeadNext)) {
                    return oldHeadNext.item;
                }
            }
        } else { // No need to deal with tail
            tail.compareAndSet(oldTail, oldHeadNext); // tail updating
            break; // At the head of the queue
        }
    }
    return null;
}
public E take() {
    for (;;) {
        Node<E> oldHead = head.get(); // current head
        Node<E> oldTail = tail.get(); // current tail
        Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
        if (oldHead == head.get()) { //head, tail, and next changed?
            if (oldHead == oldTail) { //Queue empty or tail 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 updating
                }
            } else { // No need to deal with tail
                if (head.compareAndSet(oldHead, oldHeadNext)) {
                    return oldHeadNext.item;
                }
            }
        }
    }
}
public E take() {
    for (; ;) {
        Node<E> oldHead = head.get(); // current head
        Node<E> oldTail = tail.get(); // current tail
        Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
        if (oldHead == head.get()) { // head, tail, and next changed?
            if (oldHead == oldTail) { // Queue empty or tail 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 updating
                }
            } else { // No need to deal with tail
                if (head.compareAndSet(oldHead, oldHeadNext)) {
                    return oldHeadNext.item;
                }
            }
        }
    }
}
public E take() {
    for (;;) {
        1 Node<E> oldHead = head.get(); // current head
        2 Node<E> oldTail = tail.get();  // current tail
        3 Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
        4 if (oldHead == head.get()) { //head, tail, and next changed?
            5 if (oldHead == oldTail) { //Queue empty or tail updated?
                6 if (oldHeadNext == null) { // Is queue empty?
                    7 return null; //Queue is empty, can't take
                }
                8 tail.compareAndSet(oldTail, oldHeadNext); // tail updating
            } else { // No need to deal with tail
                9 if (head.compareAndSet(oldHead, oldHeadNext)) {
                    10 return oldHeadNext.item;
                }
            }
        }
    }
}
public E take() {
    for (;;) {
        Node<E> oldHead = head.get(); // current head
        Node<E> oldTail = tail.get(); // current tail
        Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
        if (oldHead == head.get()) { //head, tail, and next changed?
            if (oldHead == oldTail) { //Queue empty or tail 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 updating
                }
            } else { // No need to deal with tail
                if (head.compareAndSet(oldHead, oldHeadNext)) {
                    return oldHeadNext.item;
                }
            }
        }
    }
}
Execution Traces

- 1 thread calls take()
- 1 thread calls put()
public E take() {
    for (;;) {
        Node<E> oldHead = head.get();       // current head
        Node<E> oldTail = tail.get();       // current tail
        Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
        if (oldHead == head.get()) {        // head, tail, and next changed?
            if (oldHead == oldTail) {        // Queue empty or tail updated?
                if (oldHeadNext == null) { // Is queue empty?
                    return null;              // Queue is empty, can't take
                }
                tail.compareAndSet(oldTail, oldHeadNext); // tail updating
                else {                         // No need to deal with tail
                    if (head.compareAndSet(oldHead, oldHeadNext)) {
                        return oldHeadNext.item;
                    }
                }
            }
        }
    }
}
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 E take() {
    for (;;) {
        Node<E> oldHead = head.get(); // current head
        Node<E> oldTail = tail.get(); // current tail
        Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
        if (oldHead == head.get()) { //head, tail, and next changed?
            if (oldHead == oldTail) { //Queue empty or tail updated?
                if (oldHeadNext == null) { // Is queue empty?
                    return null; //Queue is empty, can't take
                } else { // No need to deal with tail
                    return oldHeadNext.item;
                }
            } else { // No need to deal with tail
                tail.compareAndSet(oldTail, oldHeadNext); // tail updating
            }
        }
        else { // No need to deal with tail
            if (head.compareAndSet(oldHead, oldHeadNext)) {
                return oldHeadNext.item;
            }
        }
    }
}
public E take() {
    for (;;) {
        Node<E> oldHead = head.get(); // current head
        Node<E> oldTail = tail.get(); // current tail
        Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
        if (oldHead == head.get()) { //head, tail, and next changed?
            if (oldHead == oldTail) { //Queue empty or tail 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 updating
                }
            } else { // No need to deal with tail
                if (head.compareAndSet(oldHead, oldHeadNext)) {
                    return oldHeadNext.item;
                }
            }
        }
    }
}
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 1: CAS succeeds

```java
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.equals(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 E take() {
    for (; ; ) {
        Node<E> oldHead = head.get(); // current head
        Node<E> oldTail = tail.get(); // current tail
        Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
        if (oldHead == head.get()) {  // head, tail, and next changed?
            if (oldHead == oldTail) {   // Queue empty or tail updated?
                if (oldHeadNext == null) { // Is queue empty?
                    return null;           // Queue is empty, can't take
                }
                tail.compareAndSet(oldTail, oldHeadNext); // tail updating
            } else {                     // No need to deal with tail
                if (head.compareAndSet(oldHead, oldHeadNext)) {
                    return oldHeadNext.item;
                }
            }
        }
    }
}
public E take() {
    for (;;) {
        Node<E> oldHead = head.get();  // current head
        Node<E> oldTail = tail.get();   // current tail
        Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
        if (oldHead == head.get()) {    // head, tail, and next changed?
            if (oldHead == oldTail) {    // Queue empty or tail 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 updating
                }
            } else {                   // No need to deal with tail
                if (head.compareAndSet(oldHead, oldHeadNext)) {
                    return oldHeadNext.item;
                }
            }
        }
    }
}
public E take() {
  for (;;) {
    Node<E> oldHead = head.get(); // current head
    Node<E> oldTail = tail.get(); // current tail
    Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
    if (oldHead == head.get()) { //head, tail, and next changed?
      if (oldHead == oldTail) { //Queue empty or tail updated?
        if (oldHeadNext == null) { // Is queue empty?
          return null; //Queue is empty, can't take
        }
      }
      tail.compareAndSet(oldTail, oldHeadNext); // tail updating
    } else { // No need to deal with tail
      if (head.compareAndSet(oldHead, oldHeadNext)) {
        return oldHeadNext.item;
      }
    }
  }
}
public E take() {
    for (;;) {
        Node<E> oldHead = head.get(); // current head
        Node<E> oldTail = tail.get(); // current tail
        Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
        if (oldHead == head.get()) { //head, tail, and next changed?
            if (oldHead == oldTail) { //Queue empty or tail updated?
                if (oldHeadNext == null) { // Is queue empty?
                    return null; //Queue is empty, can't take
                }
            }
            tail.compareAndSet(oldTail, oldHeadNext); // tail updating
        } else { // No need to deal with tail
            if (head.compareAndSet(oldHead, oldHeadNext)) {
                return oldHeadNext.item;
            }
        }
    }
}
public E take() {
    for (;;) {
        Node<E> oldHead = head.get(); // current head
        Node<E> oldTail = tail.get();  // current tail
        Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
        if (oldHead == head.get()) {  //head, tail, and next changed?
            if (oldHead == oldTail) {  //Queue empty or tail 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 updating
                }
            } else { // No need to deal with tail
                if (head.compareAndSet(oldHead, oldHeadNext)) {
                    return oldHeadNext.item;
                }
            }
        }
    }
}
public E take() {
    for (;;) {
        Node<E> oldHead = head.get();       // current head
        Node<E> oldTail = tail.get();       // current tail
        Node<E> oldHeadNext = oldHead.next.get();   // curr head.next
        if (oldHead == head.get()) {    // head, tail, and next changed?
            if (oldHead == oldTail) {   //Queue empty or tail updated?
                if (oldHeadNext == null) { // Is queue empty?
                    return null;          //Queue is empty, can't take
                }
            } else {  // No need to deal with tail
                if (head.compareAndSet(oldHead, oldHeadNext)) {
                    return oldHeadNext.item;
                }
            }
        }
        tail.compareAndSet(oldTail, oldHeadNext);  // tail updating
    }
}
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 E take() {
    for (;;) {
        Node<E> oldHead = head.get(); // current head
        Node<E> oldTail = tail.get(); // current tail
        Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
        if (oldHead == head.get()) { //head, tail, and next changed?
            if (oldHead == oldTail) { //Queue empty or tail updated?
                if (oldHeadNext == null) { // Is queue empty?
                    return null; //Queue is empty, can't take
                }
                tail.compareAndSet(oldTail, oldHeadNext); // tail updating
            } else { // No need to deal with tail
                if (head.compareAndSet(oldHead, oldHeadNext)) {
                    return oldHeadNext.item;
                }
            }
        }
    }
}
public E take() {
    for (;;) {
        1 Node<E> oldHead = head.get(); // current head
        2 Node<E> oldTail = tail.get(); // current tail
        3 Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
        4 if (oldHead == head.get()) { //head, tail, and next changed?
            5 if (oldHead == oldTail) { //Queue empty or tail updated?
                6 if (oldHeadNext == null) { // Is queue empty?
                    7 return null; //Queue is empty, can't take
                }
                8 tail.compareAndSet(oldTail, oldHeadNext); // tail updating
            } else { // No need to deal with tail
                9 if (head.compareAndSet(oldHead, oldHeadNext)) {
                    10 return oldHeadNext.item;
                }
            }
        }
    }
}
public E take() {
    for (; ; ) {
    Node<E> oldHead = head.get(); // current head
    Node<E> oldTail = tail.get(); // current tail
    Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
    if (oldHead == head.get()) { //head, tail, and next changed?
        if (oldHead == oldTail) { //Queue empty or tail updated?
            if (oldHeadNext == null) { // Is queue empty?
                return null; //Queue is empty, can't take
            }
            tail.compareAndSet(oldTail, oldHeadNext); // tail updating
        } else { // No need to deal with tail
            if (head.compareAndSet(oldHead, oldHeadNext)) {
                return oldHeadNext.item;
            }
        }
    }
    }
}
public E take() {
    for (;;) {
        Node<E> oldHead = head.get(); // current head
        Node<E> oldTail = tail.get(); // current tail
        Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
        if (oldHead == head.get()) { //head, tail, and next changed?
            if (oldHead == oldTail) { //Queue empty or tail updated?
                if (oldHeadNext == null) { // Is queue empty?
                    return null; //Queue is empty, can't take
                }
            }
            tail.compareAndSet(oldTail, oldHeadNext); // tail updating
        } else { // No need to deal with tail
            if (head.compareAndSet(oldHead, oldHeadNext)) {
                return oldHeadNext.item;
            }
        }
    }
}
public E take() {
    for (;;) {
        Node<E> oldHead = head.get(); // current head
        Node<E> oldTail = tail.get(); // current tail
        Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
        if (oldHead == head.get()) { // head, tail, and next changed?
            if (oldHead == oldTail) { // Queue empty or tail updated?
                if (oldHeadNext == null) { // Is queue empty?
                    return null; // Queue is empty, can't take
                }
                tail.compareAndSet(oldTail, oldHeadNext); // tail updating
            } else { // No need to deal with tail
                if (head.compareAndSet(oldHead, oldHeadNext)) {
                    return oldHeadNext.item;
                }
            }
        }
    }
}
public E take() {
  for (;;) {
    Node<E> oldHead = head.get();                        // current head
    Node<E> oldTail = tail.get();                            // current tail
    Node<E> oldHeadNext = oldHead.next.get(); / / curr head.next
    if (oldHead == head.get()) {      //head, tail, and next changed?
      if (oldHead == oldTail) {       //Queue empty or tail 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 updating
        }
      }
    else {
      return oldHeadNext.item;
    }
  }
}
public E take() {
    for (;;) {
        Node<E> oldHead = head.get(); // current head
        Node<E> oldTail = tail.get(); // current tail
        Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
        if (oldHead == head.get()) { //head, tail, and next changed?
            if (oldHead == oldTail) { //Queue empty or tail 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 updating
                }
            }
        } else { // No need to deal with tail
            if (head.compareAndSet(oldHead, oldHeadNext)) {
                return oldHeadNext.item;
            }
        }
    }
}
public E take() {
    for (;;) {
        Node<E> oldHead = head.get(); // current head
        Node<E> oldTail = tail.get(); // current tail
        Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
        if (oldHead == head.get()) { //head, tail, and next changed?
            if (oldHead == oldTail) { //Queue empty or tail 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 updating
                }
            } else { // No need to deal with tail
                if (head.compareAndSet(oldHead, oldHeadNext)) {
                    return oldHeadNext.item;
                }
            }
        }
    }
}