Linked Lists: Locking, Lock-Free, and Beyond ...

The Art of Multiprocessor Programming
by Maurice Herlihy & Nir Shavit
chapter 9
Today: Concurrent Objects

- Adding threads should not lower throughput
  - Contention effects
  - Mostly fixed by Queue locks
- Should increase throughput
  - Not possible if inherently sequential
  - Surprising things are parallelizable
Coarse-Grained Synchronization

• Each method locks the object
  - Avoid contention using queue locks
  - Easy to reason about
    • In simple cases
  - Standard Java model
    • Synchronized blocks and methods

• So, are we done?
Coarse-Grained Synchronization

- Sequential bottleneck
  - Threads “stand in line”
- Adding more threads
  - Does not improve throughput
  - Struggle to keep it from getting worse
- So why even use a multiprocessor?
  - Well, some apps inherently parallel ...
This Lecture

• Introduce four “patterns”
  - Bag of tricks ...
  - Methods that work more than once ...
• For highly-concurrent objects
• Goal:
  - Concurrent access
  - More threads, more throughput
First:
Fine-Grained Synchronization

• Instead of using a single lock..
• Split object into
  - Independently-synchronized components
• Methods conflict when they access
  - The same component ...
  - At the same time
Second:
Optimistic Synchronization

• Search without locking ...
• If you find it, lock and check ...
  - OK: we are done
  - Oops: start over
• Evaluation
  - Usually cheaper than locking
  - Mistakes are expensive
Third: Lazy Synchronization

• Postpone hard work
• Removing components is tricky
  – Logical removal
    • Mark component to be deleted
  – Physical removal
    • Do what needs to be done
Fourth: Lock-Free Synchronization

• Don’t use locks at all
  - Use compareAndSet() & relatives ...

• Advantages
  - No Scheduler Assumptions/Support

• Disadvantages
  - Complex
  - Sometimes high overhead
Linked List

• Illustrate these patterns ...
• Using a list-based Set
  – Common application
  – Building block for other apps
Set Interface

- Unordered collection of items
- No duplicates
- Methods
  - add(x) put x in set
  - remove(x) take x out of set
  - contains(x) tests if x in set
List-Based Sets

```java
public interface Set<T> {
    public boolean add(T x);
    public boolean remove(T x);
    public boolean contains(T x);
}
```
List Node

```java
public class Node {
    public T item;
    public int key;
    public Node next;
}
```
The List-Based Set

Sorted with Sentinel nodes (min & max possible keys)
Reasoning about Concurrent Objects

- Invariant
  - Property that always holds

- Established because
  - True when object is created
  - Truth preserved by each method
    - Each step of each method?
      - o importante são passos visíveis externamente...
    - sentinels are neither added nor removed
    - nodes are sorted by unique keys
Specifically ...

• Invariants preserved by
  – add()
  – remove()
  – contains()

• linearizability:
  – o efeito de cada método deve se tornar visível instantaneamente em algum momento entre sua invocação e retorno
    • com locks, seção crítica
Interference

• Invariants make sense only if
  - methods considered are the only modifiers

• Language encapsulation helps
  - List nodes not visible outside class
Interference

- Freedom from interference needed even for removed nodes
  - Some algorithms traverse removed nodes
  - Careful with malloc() & free()!
- Garbage-collection helps here
Sequential List Based Set

Add()

Remove()
Sequential List Based Set

Add()

Remove()
Course Grained Locking
public boolean remove(T item) {
    Node pred, curr;
    int key = item.hashCode();
    lock.lock();
    try {
        pred = head; curr = pred.next;
        while (curr.key < key) {
            pred = curr; curr = curr.next;
        }
        if (key == curr.key) {
            pred.next = curr.next;
            return true;
        } else return false;
    } finally {
        lock.unlock();
    }
}
Course Grained Locking
Course Grained Locking

Simple but hotspot + bottleneck
Coarse-Grained Locking

- Easy, same as synchronized methods
  - “One lock to rule them all …”
- Simple, clearly correct
  - Deserves respect!
- Works poorly with contention
  - Queue locks help
  - But bottleneck still an issue
Fine-grained Locking

• Requires careful thought
  - “Do not meddle in the affairs of wizards, for they are subtle and quick to anger”

• Split object into pieces
  - Each piece has own lock
  - Methods that work on disjoint pieces need not exclude each other
Hand-over-Hand locking

Art of Multiprocessor Programming
Hand-over-Hand locking
Hand-over-Hand locking
Hand-over-Hand locking
Hand-over-Hand locking
Removing a Node

```
remove(b)
```
Removing a Node

\[\text{remove(b)}\]
Removing a Node

remove(b)
Removing a Node

remove(b)
Removing a Node

Art of Multiprocessor Programming
Removing a Node

Why do we need to always hold 2 locks?
Concurrent Removes

remove(b)

remove(c)
Concurrent Removes

Art of Multiprocessor Programming
Concurrent Removes

```
remove(b)
```
Concurrent Removes

```
remove(b)
```

```
remove(c)
```
Concurrent Removes

Art of Multiprocessor Programming
Concurrent Removes

Art of Multiprocessor Programming
Concurrent Removes

Art of Multiprocessor Programming
Concurrent Removes

Art of Multiprocessor Programming
Uh, Oh

![Diagram showing a sequence of operations involving variables a, c, and d with annotations remove(b) and remove(c).]
Uh, Oh

Bad news, C not removed

remove(b)

remove(c)
Problem

- To delete node c
  - Swing node b’s next field to d

- Problem is,
  - Someone deleting b concurrently could direct a pointer to c
Insight

• If a node is locked
  - No one can delete node’s successor

• If a thread locks
  - Node to be deleted
  - And its predecessor
  - Then it works
Removing a Node

\[ \text{remove}(b) \]

\[ \text{remove}(c) \]
Removing a Node

\[ \text{remove}(b) \]

\[ \text{remove}(c) \]
Removing a Node

remove(b)
Removing a Node

remove(b)
remove(c)
Removing a Node

Art of Multiprocessor Programming
Removing a Node

remove(b)

remove(c)
Removing a Node

remove(b)
remove(c)
Removing a Node

remove(b)
remove(c)
Removing a Node

Must acquire Lock of b

remove(c)
Removing a Node

Cannot acquire lock of b

remove(c)
Removing a Node

Wait!

remove(c)
Removing a Node

Proceed to remove(b)
Removing a Node

Art of Multiprocessor Programming
Removing a Node

\[
\text{remove(b)}
\]
Removing a Node

Art of Multiprocessor Programming
Removing a Node
public boolean remove(Item item) {
    int key = item.hashCode();
    Node pred, curr;
    try {
        ...
    } finally {
        curr.unlock();
        pred.unlock();
    }
}
Remove method

```java
public boolean remove(Item item) {
    int key = item.hashCode();
    Node pred, curr;
    try {
        ... 
    } finally {
        curr.unlock();
        pred.unlock();
    }
}
```

Key used to order node
Remove method

```java
public boolean remove(Item item) {
    int key = item.hashCode();
    Node pred, curr;
    try {
        ...
    } finally {
        currNode.unlock();
        predNode.unlock();
    }
}

Predecessor and current nodes
```
public boolean remove(Item item) {
    int key = item.hashCode();
    Node pred, curr;
    try {
        ... 
    } finally {
        curr.unlock();
        pred.unlock();
    }
}
public boolean remove(Item item) {
    int key = item.hashCode();
    Node pred, curr;
    try {
        ...
    } finally {
        curr.unlock();
        pred.unlock();
    }
}
Remove method

```java
try {
    pred = this.head;
    pred.lock();
    curr = pred.next;
    curr.lock();
    ...
} finally {
    ...
}
```
Remove method

```java
try {
    pred = this.head;
pred.lock();
curr = pred.next;
curr.lock();
...}
} finally { ... }
```
Remove method

```java
try {
    pred = this.head;
    pred.lock();
    curr = pred.next;
    curr.lock();
    ...
} finally { ... }
```
Remove method

```java
try {
    pred = this.head;
    pred.lock();
    curr = pred.next;
    curr.lock();
    ...
} finally { ... }
```
while (curr.key <= key) {
    if (item == curr.item) {
        pred.next = curr.next;
        return true;
    }
    pred.unlock();
    pred = curr;
    curr = curr.next;
    curr.lock();
}
return false;
Remove: searching

```cpp
while (curr.key <= key) {
    if (item == curr.item) {
        pred.next = curr.next;
        return true;
    }
    pred.unlock();
    pred = curr;
    curr = curr.next;
    curr.lock();
}
return false;
```
Remove: searching

```java
while (curr.key <= key) {
    if (item == curr.item) {
        pred.next = curr.next;
        return true;
    }
    pred.unlock();
    pred = curr;
    curr = curr.next;
    curr.lock();
}
return false;
```

At start of each loop: `curr` and `pred` locked
Remove: searching

while (curr.key <= key) {
  if (item == curr.item) {
    pred.next = curr.next;
    return true;
  }
  pred.unlock();
  pred = curr;
  curr = curr.next;
  curr.lock();
}

If item found, remove node
Remove: searching

while (curr.key <= key) {
    if (item == curr.item) {
        pred.next = curr.next;
        return true;
    }
    pred.unlock();
    pred = curr;
    curr = curr.next;
    curr.lock();
}

If node found, remove it
Remove: searching

while (curr.key <= key) {
    if (item == curr.item) {
        pred.next = curr.next;
        return true;
    }
    pred.unlock();
    pred = curr;
    curr = curr.next;
    curr.lock();
}
return false;
Remove: searching

while (curr.key <= key) {
    if (item == curr.item) {
        pred.next = curr.next;
        return true;
    }
    pred.unlock();
    pred = curr;
    curr = curr.next;
    curr.lock();
}
return false;

Only one node locked!
while (curr.key <= key) {
  if (item == curr.item) {
    pred.next = curr.next;
    return true;
  }
  pred.unlock();
  pred = curr;
  curr = curr.next;
  curr.lock();
}
return false;
while (curr.key <= key) {
  if (item == curr.item) {
    pred.next = curr.next;
    return true;
  }
  pred.unlock();
  pred = currNode;
  curr = curr.next;
  curr.lock();
}
return false;
while (curr.key <= key) {
    if (item == curr.item) {
        pred.next = curr.next;
        return true;
    }
    pred.unlock();
    pred = currNode;
    curr = curr.next;
    curr.lock();
}
return false;
Remove: searching

while (curr.key <= key) {
    if (item == curr.item) {
        pred.next = curr.next;
        return true;
    }
    pred.unlock();
    pred = curr;
    curr = curr.next;
    curr.lock();
}

return false;

Otherwise, not present
Why does this work?

• To remove node $e$
  - Must lock $e$
  - Must lock $e$’s predecessor

• Therefore, if you lock a node
  - It can’t be removed
  - And neither can its successor

linearization point:
  - if $e$ is present, when $e$’s predecessor is locked
Rep Invariant

- Easy to check that
  - tail always reachable from head
  - Nodes sorted, no duplicates
Drawbacks

• Better than coarse-grained lock
  - Threads can traverse in parallel
• Still not ideal
  - Long chain of acquire/release
  - Inefficient

thread may still be delayed by another using different part of the list...
but if the locks are fair, there will be no starvation
Optimistic Synchronization

• Find nodes without locking
• Lock nodes
• Check that everything is OK
Optimistic: Traverse without Locking

add(c)

Aha!
Optimistic: Lock and Load
What could go wrong?

add(c) → a → b → d → e

remove(b)

Aha!
public boolean remove(T item) {
    int key = item.hashCode();
    while (true) {
        Node pred = head; Node curr = pred.next;
        while (curr.key <= key) {
            pred = curr; curr = curr.next;
            while (curr.key < key) {
                pred = curr; curr = curr.next;
            }
            pred.lock(); curr.lock();
            try {
                if (validate(pred, curr)) {
                    if (curr.key == key) {
                        pred.next = curr.next;
                        return true; }
                    else return false;
                }
                } finally {
                    pred.unlock(); curr.unlock();
                }
            }
        }
    }
}
Optimistic: Linearization Point

remove (d)

locks em pred e curr e validação ok
Invariants

• **Careful:** we may traverse deleted nodes
• **But we establish properties by**
  - Validation
  - After we lock target nodes
Correctness

• If
  - Nodes b and d both locked
  - Node b still accessible
  - Node d still successor to b

• Then
  - Neither will be deleted
  - OK to delete and return true
Unsuccessful Remove

\[ \text{remove}(c) \]

Aha!
Validate (1)

Yes, b still reachable from head
Validate (2)

Yes, b still points to d

remove(c)
OK Computer

```
remove(c)
return false
```
Correctness

• If
  - Nodes b and d both locked
  - Node b still accessible
  - Node d still successor to b
• Then
  - Neither will be deleted
  - No thread can add c after b
  - OK to return false
private boolean validate(Node pred, Node curr) {
    Node node = head;
    while (node.key <= pred.key) {
        if (node == pred)
            return pred.next == curr;
        node = node.next;
    }
    return false;
}
private boolean validate(Node pred, Node curr) {
Node node = head;
while (node.key <= pred.key) {
    if (node == pred)
        return pred.next == curr;
    node = node.next;
}
return false;
}
private boolean validate(Node pred, Node curr) {
    Node node = head;
    while (node.key <= pred.key) {
        if (node == pred)
            return pred.next == curr;
        node = node.next;
    }
    return false;
}
private boolean validate(Node pred, Node curr) {
    Node node = head;
    while (node.key <= pred.key) {
        if (node == pred)
            return pred.next == curr;
        node = node.next;
    }
    return false;
}
private boolean validate(Node pred, Node curr) {
    Node node = head;
    while (node.key <= pred.key) {
        if (node == pred)
            return pred.next == curr;
        node = node.next;
    }
    return false;
}

Validation

Predecessor reachable
private boolean validate(Node pred, Node curry) {
    Node node = head;
    while (node.key <= pred.key) {
        if (node == pred)
            return pred.next == curr;
        node = node.next;
    }
    return false;
}
private boolean validate(Node pred, Node curr) {
    Node node = head;
    while (node.key <= pred.key) {
        if (node == pred)
            return pred.next == curr;
        node = node.next;
    }
    return false;
}
private boolean validate(Node pred, Node curr) {
    Node node = head;
    while (node.key <= pred.key) {
        if (node == pred)
            return pred.next == curr;
        node = node.next;
    }
    return false;
}
possíveis problemas

• nós podem ter saído da lista
  - mas enquanto alguma thread os referenciar, não serão coletados....
public boolean remove(T item) {
    int key = item.hashCode();
    while (true) {
        Node pred = head; Node curr = pred.next;
        while (curr.key <= key) {
            pred = curr; curr = curr.next;
            while (curr.key < key) {
                pred = curr; curr = curr.next;
            }
        }
        pred.lock(); curr.lock();
        try {
            if (validate(pred, curr)) {
                if (curr.key == key) {
                    pred.next = curr.next;
                    return true;
                } else return false;
            }
        } finally {
            pred.unlock(); curr.unlock();
        }
    }
}


nesse caso volta a fazer todo o percurso!
Optimistic List

• Limited hot-spots
  - Targets of `add()`, `remove()`, `contains()`
  - No contention on traversals

• Moreover
  - Traversals are wait-free
  - Food for thought ...

- not starvation-free
So Far, So Good

• Much less lock acquisition/release
  - Performance
  - Concurrency

• Problems
  - Need to traverse list twice
  - contains() method acquires locks
Evaluation

• Optimistic is effective if
  - cost of scanning twice without locks is less than
  - cost of scanning once with locks

• Drawback
  - contains() acquires locks
  - 90% of calls in many apps
Lazy List

- Like optimistic, except
  - Scan once
  - \texttt{contains\(x\)} never locks ...

- Key insight
  - Removing nodes causes trouble
  - Do it “lazily”
Lazy List

• remove()
  - Scans list (as before)
  - Locks predecessor & current (as before)

• Logical delete
  - Marks current node as removed (new!)

• Physical delete
  - Redirects predecessor’s next (as before)
Lazy Removal
Lazy Removal

Present in list
Lazy Removal

Logically deleted
Lazy Removal

Physically deleted
Lazy Removal

Physically deleted
Lazy List

- **All Methods**
  - Scan through locked and marked nodes
  - Removing a node doesn’t slow down other method calls …
- **Must still lock** pred and curr nodes.
Validation

- No need to rescan list!
- Check that \( \text{pred} \) is not marked
- Check that \( \text{curr} \) is not marked
- Check that \( \text{pred} \) points to \( \text{curr} \)

mas não precisa percorrer a lista desde o início
Business as Usual
Business as Usual
Business as Usual

Art of Multiprocessor Programming
Business as Usual

remove(b)
Business as Usual

Art of Multiprocessor Programming
Business as Usual

Art of Multiprocessor Programming
Business as Usual

Art of Multiprocessor Programming
Business as Usual

Art of Multiprocessor Programming
Business as Usual
Validation

```java
private boolean validate(Node pred, Node curr) {
    return !pred.marked && !curr.marked && pred.next == curr);
}
```

objetivo da marca: evitar duplo percurso
private boolean validate(Node pred, Node curr) {
    return !pred.marked && !curr.marked && pred.next == curr);
}

List Validate Method

Predecessor not Logically removed
private boolean validate(Node pred, Node curr) {
    return !pred.marked && !curr.marked && pred.next == curr;
}
private boolean validate(Node pred, Node curr) {
    return 
    !pred.marked && 
    !curr.marked && 
    pred.next == curr);
}
public boolean remove(T item) {
    int key = item.hashCode();
    while (true) {
        Node pred = head; Node curr = pred.next;
        while (curr.key <= key) {
            pred = curr; curr = curr.next;
            while (curr.key < key) {
                pred = curr; curr = curr.next;
            }
            pred.lock(); curr.lock();
            try {
                if (validate(pred, curr)) {
                    ...
                    else return false;
                }
            } finally {
                pred.unlock(); curr.unlock();
            }
        }
    }

    ‣ nesse caso volta a fazer todo o percurso!
try {
    pred.lock(); curr.lock();
    if (validate(pred, curr) {
        if (curr.key == key) {
            curr.marked = true;
            pred.next = curr.next;
            return true;
        } else {
            return false;
        }
    } else {
        return false;
    }}
} finally {
    pred.unlock();
    curr.unlock();
}
try {
    pred.lock(); curr.lock();
    if (validate(pred, curr)) {
        if (curr.key == key) {
            curr.marked = true;
            pred.next = curr.next;
            return true;
        } else {
            return false;
        }
    } else {
        return false;
    }
} finally {
    pred.unlock();
    curr.unlock();
}
try {
    pred.lock(); curr.lock();
    if (validate(pred, curr)) {
        if (curr.key == key) {
            curr.marked = true,
            pred.next = curr.next;
            return true;
        } else {
            return false;
        }
    } else {
        return false;
    }
} finally {
    pred.unlock();
    curr.unlock();
}
Remove

```java
try {
    pred.lock(); curr.lock();
    if (validate(pred, curr)) {
        if (curr.key == key) {
            curr.marked = true;
            pred.next = curr.next;
            return true;
        } else {
            return false;
        }
    } else {
        return false;
    }
} finally {
    pred.unlock();
    curr.unlock();
}}
```

Logical remove
try {
    pred.lock(); curr.lock();
    if (validate(pred, curr) {
        if (curr.key == key) {
            curr.marked = true;
            pred.next = curr.next;
            return true;
        } else {
            return false;
        }
    } else {
        return false;
    }
}} finally {
    pred.unlock();
    curr.unlock();
}}

physical remove
Contains

public boolean contains(Item item) {
    int key = item.hashCode();
    Node curr = this.head;
    while (curr.key < key) {
        curr = curr.next;
    }
    return curr.key == key && !curr.marked;
}
Contains

```java
public boolean contains(Item item) {
    int key = item.hashCode();
    Node curr = this.head;
    while (curr.key < key) {
        curr = curr.next;
    }
    return curr.key == key && !curr.marked;
}
```

Start at the head
public boolean contains(Item item) {
    int key = item.hashCode();
    Node curr = this.head;
    while (curr.key < key) {
        curr = curr.next;
    }
    return curr.key == key && !curr.marked;
}
public boolean contains(Item item) {
    int key = item.hashCode();
    Node curr = this.head;
    while (curr.key < key) {
        curr = curr.next;
    }
    return curr.key == key && !curr.marked;
}
Contains

```java
public boolean contains(Item item) {
    int key = item.hashCode();
    Node curr = this.head;
    while (curr.key < key) {
        curr = curr.next;
    }
    return curr.key == key && !curr.marked;
}
```

Present and undeleted?
Summary: Wait-free Contains

Use Mark bit + Fact that List is ordered
1. Not marked $\rightarrow$ in the set
2. Marked or missing $\rightarrow$ not in the set

- wait-free: every call finishes its execution in a finite number of steps
Lazy List

Lazy add() and remove() + Wait-free contains()
Evaluation

• **Good:**
  - `contains()` doesn’t lock
  - In fact, its wait-free!
  - *Good because typically high % contains()*
  - Uncontended calls don’t re-traverse

• **Bad**
  - *Contended add() and remove() calls do re-traverse*
  - *Traffic jam if one thread delays*
Traffic Jam

• Any concurrent data structure based on mutual exclusion has a weakness

• If one thread
  - Enters critical section
  - And “eats the big muffin”
    • Cache miss, page fault, descheduled ...
  - Everyone else using that lock is stuck!
  - Need to trust the scheduler....
Reminder: Lock-Free Data Structures

• No matter what ...
  - Guarantees minimal progress in any execution
  - i.e. Some thread will always complete a method call
  - Even if others halt at malicious times
  - Implies that implementation can’t use locks

• CAS operations
Lock-free Lists

- Next logical step
- Eliminate locking entirely
- `contains()` wait-free and `add()` and `remove()` lock-free
- Use only `compareAndSet()`
- What could go wrong?
Remove Using CAS

não adianta fazer CAS atômico com as referências!!!
Remove Using CAS

Logical Removal =
Set Mark Bit

Physical
Removal
CAS pointer

tem que levar em
consideração estado do nó!
Problem:
- d not added to list...
- Must Prevent manipulation of removed node’s pointer

Logical Removal = Set Mark Bit

Physical Removal CAS

Node added Before Physical Removal CAS
The Solution: Combine Bit and Pointer

Mark-Bit and Pointer are CASed together (AtomicMarkableReference)

Physical Removal CAS

Logical Removal = Set Mark Bit

Fail CAS: Node not added after logical Removal
Solution

- **Use** `AtomicMarkableReference`
- **Atomically**
  - Swing reference and
  - Update flag
- **Remove in two steps**
  - Set mark bit in next field
  - Redirect predecessor’s pointer
Marking a Node

- AtomicMarkableReference class
  - Java.util.concurrent.atomic package
Extracting Reference & Mark

```java
public Object get(boolean[] marked);
```
Extracting Reference & Mark

Public Object get();

Returns reference

boolean[] marked;

Returns mark at array index 0!
Extracting Reference Only

```java
public boolean isMarked();
```

Value of mark
Changing State

```java
public boolean compareAndSet(
    Object expectedRef,
    Object updateRef,
    boolean expectedMark,
    boolean updateMark);
```
Changing State

```java
public boolean compareAndSet(
    Object expectedRef,
    Object updateRef,
    boolean expectedMark,
    boolean updateMark);
```

If this is the current reference ...

And this is the current mark ...

Art of Multiprocessor Programming
Changing State

Public boolean compareAndSet(
    Object expectedRef,
    Object updateRef,
    boolean expectedMark,
    boolean updateMark);

...then change to this new reference ...

... and this new mark
Changing State

```java
public boolean attemptMark(
    Object expectedRef,
    boolean updateMark);
```
Changing State

public boolean attemptMark(
    Object expectedRef,
    boolean updateMark);

If this is the current reference ...
public boolean attemptMark(
Object expectedRef,
boolean updateMark);

.. then change to this new mark.
Traversing the List

• Q: what do you do when you find a “logically” deleted node in your path?
• A: finish the job.
  - CAS the predecessor’s next field
  - Proceed (repeat as needed)
Lock-Free Traversal
(only Add and Remove)

Uh-oh
class Window {
    public Node pred;
    public Node curr;
    Window(Node pred, Node curr) {
        this.pred = pred; this.curr = curr;
    }
}
The Window Class

class Window {
    public Node pred;
    public Node curr;

    Window(Node pred, Node curr) {
        this.pred = pred; this.curr = curr;
    }
}

A container for pred and current values
Using the Find Method

```java
window window = find(head, key);
Node pred = window.pred;
curr = window.curr;
```
Using the Find Method

```java
window window = find(head, key);
Node pred = window.pred;
curr = window.curr;
```

Find returns window
Using the Find Method

Window window = find(head, key);

Node pred = window.pred;
curr = window.curr;

Extract pred and curr
The Find Method

At some instant,

\[ \text{window} \] = \text{find(item)};

or ...

pred curr succ

item
The Find Method

At some instant,

window window = find(item);

pred curr = null

item not in list

succ
public boolean remove(T item) {
    Boolean snip;
    while (true) {
        Window window = find(head, key);
        Node pred = window.pred, curr = window.curr;
        if (curr.key != key) {
            return false;
        } else {
            Node succ = curr.next.getReference();
            snip = curr.next.attemptMark(succ, true);
            if (!snip) continue;
            pred.next.compareAndSet(curr, succ, false, false);
            return true;
        }
    }
}
public boolean remove(T item) {
  Boolean snip;
  while (true) {
    Window window = find(head, key);
    Node pred = window.pred, curr = window.curr;
    if (curr.key != key) {
      return false;
    } else {
      Node succ = curr.next.getReference();
      snip = curr.next.attemptMark(succ, true);
      if (!snip) continue;
      pred.next.compareAndSet(curr, succ, false, false);
      return true;
    }
  }
}
public boolean remove(T item) {
    Boolean snip;
    while (true) {
        Window window = find(head, key);
        Node pred = window.pred, curr = window.curr;
        if (curr.key != key) {
            return false;
        } else {
            Node succ = curr.next.getReference();
            snip = curr.next.attemptMark(succ, true);
            if (!snip) continue;
            pred.next.compareAndSet(curr, succ, false, false);
            return true;
        }
    }
}}
public boolean remove(T item) {
    Boolean snip;
    while (true) {
        Window window = find(head, key);
        Node pred = window.pred, curr = window.curr;
        if (curr.key != key) {
            return false;
        } else {
            Node succ = curr.next.getReference();
            snip = curr.next.attemptMark(succ, true);
            if (!snip) continue;
            pred.next.compareAndSet(curr, succ, false, false);
            return true;
        }
    }
}

She's not there …
public boolean remove(T item) {
    Boolean snip;
    while (true) {
        Window window = find(head, key);
        Node pred = window.pred, curr = window.curr;
        if (curr.key != key) {
            return false;
        } else {
            Node succ = curr.next.getReference();
            snip = curr.next.attemptMark(succ, true);
            if (!snip) continue;
            pred.next.compareAndSet(curr, succ, false, false);
            return true;
        }
    }
}

Try to mark node as deleted

se curr.next
ainda referencia
succ, marca curr
como eliminado
public boolean remove(T item) {
  Boolean snip;
  while (true) {
    Window window = find(head, pred, curr = window.curr;
    if (curr.key != key) {
      return false;
    } else {
      Node succ = curr.next.getReference();
      snip = curr.next.attemptMark(succ, true);
      if (!snip) continue;
      pred.next.compareAndSet(curr, succ, false, false);
      return true;
    }}
  }}

If it doesn’t work, just retry, if it does, job essentially done
public boolean remove(T item) {
    Boolean snip;
    while (true) {
        Window window = find(head, key);
        Node pred = window.pred, curr = window.curr;
        if (curr.key != key) {
            return false;
        } else {
            Node succ = curr.next.getReference();
            snip = curr.next.attemptMark(succ, true);
            if (!snip) continue;
            pred.next.compareAndSet(curr, succ,
                                    false,
                                    false);
            return true;
        }
    }
}
public boolean add(T item) {
    boolean splice;
    while (true) {
        Window window = find(head, key);
        Node pred = window.pred, curr = window.curr;
        if (curr.key == key) {
            return false;
        } else {
            Node node = new Node(item);
            node.next = new AtomicMarkableRef(curr, false);
            if (pred.next.compareAndSet(curr, node, false, false)) {return true;}
        }
    }
}
public boolean add(T item) {
    boolean splice;
    while (true) {
        Window window = find(head, key);
        Node pred = window.pred, curr = window.curr;
        if (curr.key == key) {
            return false;
        } else {
            Node node = new Node(item);
            node.next = new AtomicMarkableRef(curr, false);
            if (pred.next.compareAndSet(curr, node, false, false)) {
                return true;
            }
        }
    }
}

Item already there.
public boolean add(T item) {
    boolean splice = false;
    while (true) {
        Window window = find(head, item);
        Node pred = window.pred, curr = window.curr;
        if (curr.key == key) {
            return false;
        } else {
            Node node = new Node(item);
            node.next = new AtomicMarkableRef(curr, false);
            if (pred.next.compareAndSet(curr, node, false, false)) {
                return true;
            }
        }
    }
}
public boolean add(T item) {
    boolean splice;
    while (true) {
        Window window = find(head, key);
        Node pred = window.pred, curr = window.curr;
        if (curr.key == key) {
            return false;
        } else {
            Node node = new Node(item);
            node.next = new AtomicMarkableRef(curr, false);
            if (pred.next.compareAndSet(curr, node, false, false)) {
                return true;
            }
        }
    }
}
Wait-free Contains

```java
class Node {
    int key;
    Node next;
}

public boolean contains(Tt item) {
    boolean marked;
    int key = item.hashCode();
    Node curr = this.head;
    while (curr.key < key) {
        curr = curr.next;
        Node succ = curr.next.get(marked);
    }
    return (curr.key == key && !marked[0])
}
```
Wait-free Contains

```java
public boolean contains(T item) {
    boolean marked;
    int key = item.hashCode();
    Node curr = this.head;
    while (curr.key < key)
        curr = curr.next;
    Node succ = curr.next.get(marked);
    return (curr.key == key && !marked[0])
}
```

Only diff is that we get and check marked
Lock-free Find

```java
public Window find(Node head, int key) {
    Node pred = null, curr = null, succ = null;
    boolean[] marked = {false};
    boolean snip;
    retry: while (true) {
        pred = head;
        curr = pred.next.getReference();
        while (true) {
            succ = curr.next.get(marked);
            while (marked[0]) {
                ...
            }
            if (curr.key >= key)
                return new Window(pred, curr);
            pred = curr;
            curr = succ;
        }
    }
}
```
Lock-free Find

```java
public Window find(Node head, int key) {
    Node pred = null, curr = null, succ = null;
    boolean[] marked = {false};
    boolean snip;

    retry:
    while (true) {
        pred = head;
        curr = pred.next.getReference();
        while (true) {
            succ = curr.next.get(marked);
            while (marked[0]) {
                ...
            }
            if (curr.key >= key)
                return new Window(pred, curr);
            pred = curr;
            curr = succ;
        }
    }
}
```

If list changes while traversed, start over Lock-Free because we start over only if someone else makes progress.
public Window find(Node head, int key) {
    Node pred = null, curr = null, succ = null;
    boolean[] marked = {false}; boolean snip;
    retry: while (true) {
        pred = head;
        curr = pred.next.getReference();
        while (true) {
            succ = curr.next.get(marked);
            while (marked[0]) {
                \...
            }
            if (curr.key >= key) {
                return new Window(pred, curr);
                pred = curr;
                curr = succ;
            }
        }
    }
}
public Window find(Node head, int key) {
    Node pred = null, curr = null, succ = null;
    boolean[] marked = {false}; boolean snip;
    retry: while (true) {
        pred = head;
        curr = pred.next.getReference();
        while (true) {
            succ = curr.next.get(marked);
            while (marked[0]) {
                ...
            }
            if (curr.key >= key)
                return new Window(pred, curr);
            pred = curr;
            curr = succ;
        }
    }
}
public Window find(Node head, int key) {
    Node pred = null, curr = null, succ = null;
    boolean[] marked = {false}; boolean snip;
    retry: while (true) {
        pred = head;
        curr = pred.next.getReference();
        while (true) {
            succ = curr.next.get(marked);
            while (marked[0]) {
                ...
            }
            if (curr.key >= key)
                return new Window(pred, curr);
            pred = curr;
            curr = succ;
        }
    }
}
public Window find(Node head, int key) {
    Node pred = null, curr = null, succ = null;
    boolean[] marked = {false};
    boolean snip;
    retry: while (true) {
        pred = head;
        curr = pred.next.getReference();
        while (true) {
            succ = curr.next.get(marked);
            while (marked[0]) {
                ...
            }
            if (curr.key >= key)
                return new Window(pred, curr);
            pred = curr;
            curr = succ;
        }
    }
}

Try to remove deleted nodes in path...code details soon
public Window find(Node head, int key) {
    Node pred = null, curr = null, succ = null;
    boolean[] marked = {false}; boolean snip;
    retry: while (true) {
        pred = head;
        curr = pred.next.getReference();
        for (int i = 0; i < marked.length; i++) {
            marked[i] = false;
        }
        if (curr.key >= key)
            return new Window(pred, curr);
        pred = curr;
        curr = succ;
    }
}
public Window find(Node head, int key) {
    Node pred = null, curr = null, succ = null;
    boolean[] marked = {false};
    boolean snip;
    retry: while (true) {
        pred = head;
        curr = pred.next.getReference();
        while (true) {
            succ = curr.next.get(marked);
            while (marked[0]) {
                ...}
            if (curr.key >= key)
                return new Window(pred, curr);
            pred = curr;
            curr = succ;
        }
    }
}
Lock-free Find

retry: while (true) {
    ... 
    while (marked[0]) {
        snip = pred.next.compareAndSet(curr, succ, false, false);
        if (!snip) continue retry;
        curr = succ;
        succ = curr.next.get(marked);
    }
    ...
Lock-free Find

Try to snip out node

retry: while (true) {
  ...
  while (marked[0]) {
    snip = pred.next.compareAndSet(curr, succ, false, false);
    if (!snip) continue retry;
    curr = succ;
    succ = curr.next.get(marked);
  }
  ...

Art of Multiprocessor Programming
Lock-free Find

if predecessor’s next field changed must retry whole traversal

retry: while (true) {
    ...
    while (marked[0]) {
        snip = pred.next.compareAndSet(curr, succ, false, false);
        if (!snip) continue retry;
        curr = succ;
        succ = curr.next.get(marked);
    }
    ...
}

Art of Multiprocessor Programming
Lock-free Find

retry: while (true) {
    ... 
    while (marked[0]) {
        snip = pred.next.compareAndSet(curr, succ, false, false);
        if (!snip) continue retry;
        curr = succ;
        succ = curr.next.get(marked);
    }
    ...
As Contains Ratio Increases

Ops/sec (32 threads/0 load)

Lock-free
Lazy list
Course Grained
Fine Lock-coupling
Summary

• Coarse-grained locking
• Fine-grained locking
• Optimistic synchronization
• Lock-free synchronization
“To Lock or Not to Lock”

• Locking vs. Non-blocking: Extremist views on both sides

• The answer: nobler to compromise, combine locking and non-blocking
  - Example: Lazy list combines blocking add() and remove() and a wait-free contains()
  - Remember: Blocking/non-blocking is a property of a method
This work is licensed under a Creative Commons Attribution-ShareAlike 2.5 License.

- You are free:
  - to Share — to copy, distribute and transmit the work
  - to Remix — to adapt the work
- Under the following conditions:
  - Attribution. You must attribute the work to “The Art of Multiprocessor Programming” (but not in any way that suggests that the authors endorse you or your use of the work).
  - Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.
- For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to
  - http://creativecommons.org/licenses/by-sa/3.0/.
- Any of the above conditions can be waived if you get permission from the copyright holder.
- Nothing in this license impairs or restricts the author’s moral rights.