Concurrent Linked Lists

Acknowledgement:
Slides adopted from the companion slides for the book
"The Art of Multiprocessor Programming"
by Maurice Herlihy and Nir Shavit
What We'll Cover Today

Chapter 9 of:

The Art of Multiprocessor Programming

Maurice Herlihy & Nir Shavit

Digital copy can be obtained via WUSTL library:
http://catalog.wustl.edu/search/
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: the Good

• Each method locks the object
  – Avoid contention using queue locks
  – Easy to reason about
    • In simple cases
Coarse-Grained Synchronization: the Bad

- Sequential bottleneck
  - Threads “stand in line”
- Adding more threads
  - Does not improve throughput
  - Struggle to keep it from getting worse
This Lecture

• Introduce four “patterns”
  – Bag of tricks ...
  – Methods that work more than once ...

• For highly-concurrent objects
  – Concurrent access
  – More threads, more throughput
This Lecture

• Coarse-grained locking
• Fine-grained locking
• Optimistic synchronization
• Lazy synchronization
• Lock-free synchronization
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, but
  – 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
  – \texttt{add(x)} put \texttt{x} in set
  – \texttt{remove(x)} take \texttt{x} out of set
  – \texttt{contains(x)} tests if \texttt{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-Based Sets

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

Add item to set
List-Based Sets

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

Remove item from set
List-Based Sets

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

Is item in set?
public class Node {
    public T item;
    public int key;
    public volatile Node next;
}
public class Node {
  public T item;
  public int key;
  public volatile Node next;
}
public class Node {
    public T item;
    public int key;
    public volatile Node next;
}
public class Node {
    public T item;
    public int key;
    public Node next;
}
The List-Based Set

Sorted with Sentinel nodes (min & max possible keys)

Once you find a key larger than the key you are searching for, you are done.
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
Specifically ...

- Invariants preserved by
  - `add()`
  - `remove()`
  - `contains()`

- Most steps are trivial
  - Usually one step tricky
  - Often linearization point
Interference

• Invariants make sense only if we assume **freedom from interference**: methods considered are the only modifiers to the data structure.

• Language encapsulation helps
  – List nodes not visible outside class

• Freedom from interference needed even for removed nodes
  – Some algorithms traverse removed nodes
  – Careful with `malloc()` & `free()`!

• We rely on garbage collection
Abstract Data Types

• Concrete representation:

```
  a  b
```

• Abstract Type:

\{a, b\}
Abstract Data Types

• Meaning of representation given by *abstraction map*, carrying lists that satisfy representation invariant to set.

\[ S( \text{[Diagram]} ) = \{ a, b \} \]
Representation Invariant

• Which concrete values meaningful?
  – Sorted?
  – Duplicates?

• Rep invariant
  – Characterizes legal concrete reps
  – Preserved by methods
  – Relied on by methods
Representation Invariant

- Sentinel nodes
  - tail reachable from head
- Sorted
- No duplicates
Abstraction Map

• $S(\text{head}) =$
  
  \begin{align*}
  &\{ x \mid \text{there exists a such that} \\
  &\quad \text{a reachable from head and} \\
  &\quad \text{a.item } = x \\
  \}
  \end{align*}
Sequential List Based Set

**add()**

```
  □ — □ — □ — □
  a — c — □ — d
```

**remove()**

```
  □ — □ — □ — □
  a — □ — □ — □
```

```
Sequential List Based Set

add()
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”
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
Hand-over-Hand locking
Hand-over-Hand locking
Hand-over-Hand locking
Hand-over-Hand locking
Removing a Node

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

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

remove(b)
Removing a Node

remove(b)
Removing a Node

remove(b)
Removing a Node

Why lock victim node?

remove(b)
Concurrent Removes

remove(b)

remove(c)
Concurrent Removes

remove(b)
remove(c)
Concurrent Removes

- remove(b)
- remove(c)
Concurrent Removes

remove(b)
Concurrent Removes

remove(b)

remove(c)
Concurrent Removes

- remove(b)
- remove(c)
Concurrent Removes

remove(b)

remove(c)
Concurrent Removes

![Diagram showing concurrent removes in a multiprocessor system]
Concurrent Removes

- `remove(b)`
- `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 (c's next field)

• Problem is,
  – Someone deleting b concurrently could
direct a pointer to c
  (reading b's next field)
Insight

• If a node is locked
  – No one can change node’s successor

• If a thread locks
  – Node to be deleted (so its successor don't change)
  – And its predecessor (so you are the only one changing its successor)
  – Then it works
Removing a Node

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

remove(b)

remove(c)
Removing a Node

remove(b)
remove(c)
Removing a Node

- remove(b)
- remove(c)
Removing a Node

- remove(b)
- remove(c)
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 for b

remove(c)
Removing a Node

Waiting to acquire lock for b

remove(c)
Removing a Node

Wait!

remove(c)
Removing a Node

Proceed to remove(b)
Removing a Node

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

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

```
remove(b)
```
public boolean remove(T item) {
    int key = item.hashCode();
    Node pred, curr;
    pred = head;
    pred.lock();
    curr = pred.next;
    curr.lock();
    try {
        ...
    } finally {
        curr.unlock();
        pred.unlock();
    }
}
public boolean remove(T item) {
    int key = item.hashCode();
    Node pred, curr;
    pred = head;
    pred.lock();
    curr = pred.next;
    curr.lock();
    try {
        ...
    } finally {
        curr.unlock();
        pred.unlock();
    }
}
public boolean remove(T item) {
    int key = item.hashCode();
    Node pred, curr;
    pred = head;
    pred.lock();
    curr = pred.next;
    curr.lock();
    try {
        ... 
    } finally {
        curr.unlock();
        pred.unlock();
    }
}
public boolean remove(T item) {
    int key = item.hashCode();
    Node pred, curr;
    pred = head;
    pred.lock();
    curr = pred.next;
    curr.lock();
    try {
        ...
    } finally {
        curr.unlock();
        pred.unlock();
    }
}
public boolean remove(T item) {
    int key = item.hashCode();
    Node pred, curr;
    pred = head;
    pred.lock();
    curr = pred.next;
    curr.lock();
    try {
        ...
    } finally {
        curr.unlock();
        pred.unlock();
    }
}
public boolean remove(T item) {
    int key = item.hashCode();
    Node pred, curr;
    pred = head;
    pred.lock();
    curr = pred.next;
    curr.lock();
    try {
        ...
    } finally {
        curr.unlock();
        pred.unlock();
    }
}
public boolean remove(T item) {
    int key = item.hashCode();
    Node pred, curr;
    pred = head;
    pred.lock();
    curr = pred.next;
    curr.lock();
    try {
        ...
    } finally {
        curr.unlock();
        pred.unlock();
    }
}
public boolean remove(T item) {
    int key = item.hashCode();
    Node pred, curr;
    pred = head;
    pred.lock();
    curr = pred.next;
    curr.lock();
    try {
        ...
    } finally {
        curr.unlock();
        pred.unlock();
    }
}
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 = curr;
  curr = curr.next;
  curr.lock();
}
return false;

Remove: searching (Inside the Try Block)
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 = curr;
    curr = curr.next;
    curr.lock();
}

If item found, remove node

Remove: searching (Inside the Try Block)
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 (Inside the Try Block)
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
(Inside the Try Block)

Lock invariant
restored
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 (Inside the Try Block)
Aside: Next Field Must be Volatile!

```java
public class Node {
    public T item;
    public int key;
    public volatile Node next;
}
```

Since we are no longer holding a lock when we read the "next" field, it needs to be volatile to avoid race conditions (more on that in future lecture).
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;

Why remove() is linearizable

• pred reachable from head
• curr is pred.next
• So curr.item is in the set
Why remove() is linearizable

```java
while (curr.key <= key) {
    if (item == curr.item) {
        // Linearization point if item is present
        pred.next = curr.next;
        return true;
    }
    pred.unlock();
    pred = curr;
    curr = curr.next;
    curr.lock();
}
return false;
```
Why remove() is linearizable

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;

Node locked, so no other thread can remove it ....
Why remove() is linearizable

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;
Why remove() is linearizable

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;

• pred reachable from head
• curr is pred.next
• pred.key < key
• key < curr.key
Why remove() is linearizable

```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;
```

Linearization point: the most recent read before return
Adding Nodes

• To add node e
  – Must lock predecessor
  – Must lock successor

• Neither can be deleted
  – (Is successor lock actually required?)
Abstraction Map

• $S(\text{head}) =$

\{
  x \mid \text{there exists a such that}
  
  • a reachable from head and
  
  • a.item = x

\}

Representation 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
Optimistic Synchronization

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

add(c)

Aha!
Optimistic: Lock and Load

add(c)
Optimistic: Lock and Load

add(c)
What could go wrong?

add(c)

Aha!
What could go wrong?

add(c)
What could go wrong?

remove(b)
What could go wrong?

remove(b)
What could go wrong?

```
add(c)
```
What could go wrong?

add(c)
What could go wrong?

Uh-oh

add(c)
Validate – Part 1

Yes, b still reachable from head
What Else Could Go Wrong?

add(c)  Aha!
What Else Could Go Wrong?

add(c)

add(b')
What Else Could Go Wrong?

add(c)

add(b')

Art of Multiprocessor Programming 116
What Else Could Go Wrong?

add(c)
What Else Could Go Wrong?

Art of Multiprocessor Programming
Validate Part 2
(while holding locks)

add(c)

Yes, b still points to d
Optimistic: Linearization Point

add(c)
Same Abstraction Map

- $S(\text{head}) = \{ x \mid \text{there exists a such that}
  \begin{itemize}
    \item a reachable from head and
    \item a.item = x
  \end{itemize}
\}
Invariants

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

• If
  – Nodes b and c both locked
  – Node b still accessible
  – Node c still successor to b

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

remove(c)

Aha!
Validate (1)

Yes, b still reachable from head

remove(c)
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 curry) {
    Node node = head;
    while (node.key <= pred.key) {
        if (node == pred) {
            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) {
            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) {
            if (node == pred) {
                return pred.next == curr;
            } else {
                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

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;
}
public boolean remove(T item) {
    int key = item.hashCode();
    retry: while (true) {
        Node pred = head;
        Node curr = pred.next;
        while (curr.key <= key) {
            if (item == curr.item)
                break;
            pred = curr;
            curr = curr.next;
        }
    }
    ...
public boolean remove(T item) {
    int key = item.hashCode();
    retry: while (true) {
        Node pred = head;
        Node curr = pred.next;
        while (curr.key <= key) {
            if (item == curr.item) {
                break;
            }
            pred = curr;
            curr = curr.next;
        }
    } ...
}
public boolean remove(T item) {
    int key = item.hashCode();
    retry: while (true) {
        Node pred = head;
        Node curr = pred.next;
        while (curr.key <= key) {
            if (item == curr.item)
                break;
            pred = curr;
            curr = curr.next;
        } ...
    }
    // Retry on synchronization conflict
    // (If validation fails, we come back here.)
public boolean remove(T item) {
    int key = item.hashCode();
    retry: while (true) {
        Node pred = head;
        Node curr = pred.next;
        while (curr.key <= key) {
            if (item == curr.item) break;
            pred = curr;
            curr = curr.next;
        }
    }
}

Remove: the Traversal

Examine predecessor and current nodes
public boolean remove(T item) {
    int key = item.hashCode();
    retry: while (true) {
        Node pred = head;
        Node curr = pred.next;
        while (curr.key <= key) {
            if (item == curr.item)
                break;
            pred = curr;
            curr = curr.next;
        }
        ...
public boolean remove(T item) {
    int key = item.hashCode();
    retry: while (true) {
        Node pred = head;
        Node curr = pred.next;
        while (curr.key <= key) {
            if (item == curr.item)
                break;
            pred = curr;
            curr = curr.next;
        }
    }
    return false;
}

Stop if we find item
public boolean remove(T item) {
    int key = item.hashCode();
    retry: while (true) {
        Node pred = head;
        Node curr = pred.next;
        while (curr.key <= key) {
            if (item == curr.item)
                break;
            pred = curr;
            curr = curr.next;
        }
    } ...

On Exit from Inner Loop

• If item is present
  – curr holds item
  – pred just before curr

• If item is absent
  – curr has first higher key
  – pred just before curr

• Assuming no synchronization problems
Remove Continue: the Deletion (After Existing Inner Loop)

```java
pred.lock(); curr.lock();
try {
    if (validate(pred, curr) {
        if (curr.item == item) {
            pred.next = curr.next;
            return true;
        } else {
            return false;
        }
    } else {
        return false;
    }
}} finally {
    pred.unlock();
    curr.unlock();
}}
```
Remove Continue: the Deletion
(After Existing Inner Loop)

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

Always unlock
pred.lock(); curr.lock();
try {
  if (validate(pred, curr) {
    if (curr.item == item) {
      pred.next = curr.next;
      return true;
    } else {
      return false;
    }
  } else {
    return false;
  }
} finally {
  pred.unlock();
  curr.unlock();
}

Check for synchronization conflicts
Remove Continue: the Deletion (After Existing Inner Loop)

```java
pred.lock(); curr.lock();
try {
    if (validate(pred, curr)) {
        if (curr.item == item) {
            pred.next = curr.next;
            return true;
        } else {
            return false;
        }
    } else {
        return false;
    }
} finally {
    pred.unlock();
    curr.unlock();
}
```
Remove Continue: the Deletion (After Existing Inner Loop)

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

• Limited hot-spots
  – Holding locks only on the targets of add(), remove(), contains()
  – No contention on traversals
  – Traversals are "wait-free"
    (What's wait free?)
Progress Conditions

- **Deadlock-free**: some thread trying to acquire the lock eventually succeeds.
- **Starvation-free**: every thread trying to acquire the lock eventually succeeds.
- **Lock-free**: some thread calling a method eventually returns.
- **Wait-free**: every thread calling a method eventually returns.
## Progress Conditions

<table>
<thead>
<tr>
<th>Everyone makes progress</th>
<th>Non-Blocking</th>
<th>Blocking</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Wait-free</td>
<td>Starvation-free</td>
</tr>
<tr>
<td>Someone makes progress</td>
<td>Lock-free</td>
<td>Deadlock-free</td>
</tr>
</tbody>
</table>
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
  – `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
  – add and remove still locks pred and curr, but not contain
  – Adding / removing a node doesn’t slow down contain() …
Lazy List Validation

• No need to rescan list!
• Check that pred is not marked
• Check that curr is not marked
• Check that pred points to curr
New Abstraction Map

- $S(\text{head}) =$
  
  $\{ x | \text{there exists node } a \text{ such that}
  \begin{array}{l}
  \bullet a \text{ reachable from head and} \\
  \bullet a.\text{item} = x \text{ and} \\
  \bullet a \text{ is unmarked}
  \end{array}
  \}$
Invariant

• If an item is not marked, it is reachable from head and still in the set.
• Any unmarked reachable node remains reachable even if its predecessor is logically or physically 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);
}
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);
}
private boolean validate(Node pred, Node curr) {
    return !pred.marked && !curr.marked && pred.next == curr);
}
Remove: the Deletion

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

Validate as before
... // the traversal
pred.lock(); curr.lock();
try {
  if (validate(pred, curr) {
    if (curr.key == key) {
      curr.marked = true;
      pred.next = curr.next;
      return true;
    }
  else {
      return false;
  }
}}
finally {
  pred.unlock();
  curr.unlock();
}
Remove: the Deletion

... // the traversal
pred.lock(); curr.lock();
try {
    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();
}
... // the traversal
pred.lock(); curr.lock();
try {
    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();
}}
public boolean contains(T item) {
    int key = item.hashCode();
    Node curr = head;
    while (curr.key < key) {
        curr = curr.next;
    }
    return curr.key == key && !curr.marked;
}
public boolean contains(T item) {
    int key = item.hashCode();
    Node curr = head;
    while (curr.key < key) {
        curr = curr.next;
    }
    return curr.key == key && !curr.marked;
}
public boolean contains(T item) {
    int key = item.hashCode();
    Node curr = head;
    while (curr.key < key) {
        curr = curr.next;
    }
    return curr.key == key && !curr.marked;
}
Contains

public boolean contains(T item) {
    int key = item.hashCode();
    Node curr = head;
    while (curr.key < key) {
        curr = curr.next;
    }
    return curr.key == key && !curr.marked;
}
public boolean contains(T item) {
    int key = item.hashCode();
    Node curr = head;
    while (curr.key < key) {
        curr = curr.next;
    }
    return curr.key == key && !curr.marked;
}
Summary: Wait-free Contains

Use Mark bit + list ordering
1. Not marked → in the set
2. Marked or missing → not in the set
3. Traverse the list only once!
Evaluation

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

• Bad
  – Contended `add()` and `remove()` calls must 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…. 
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
Lock-free Lists

- Next logical step
  - Wait-free `contains()`
  - lock-free `add()` and `remove()`
- Use only `compareAndSet()`
  - What could go wrong?
public abstract class CASObject {
  private int value;
  public boolean synchronized compareAndSet(int expected, int update) {
    if (value == expected) {
      value = update; return true;
    }
    return false;
  }
  ...  
}
public abstract class CASObject {
    private int value;
    public boolean synchronized compareAndSet(int expected, int update) {
        if (value == expected) {
            value = update; return true;
        }
        return false;
    }
    ... }

If value is as expected, ...
public abstract class CASOBJECT{
    private int value;
    public boolean synchronized compareAndSet(int expected, int update) {
        if (value == expected) {
            value = update;
            return true;
        }
        return false;
    }
    ... }

... replace it
public abstract class RMWRegister {
    private int value;
    public boolean synchronized compareAndSet(int expected, int update) {
        if (value == expected) {
            value = update;
            return true;
        }
        return false;
    }
}  … }

Report success
public abstract class RMWRegister {
    private int value;
    public boolean synchronized compareAndSet(int expected, int update) {
        if (value == expected) {
            value = update; return true;
        }
        return false;
    }
    ... }

Otherwise report failure

return false;
Lock-free Lists

Use CAS to verify pointer is correct

Not enough!
Problem...

T1: Logical Removal

T1: Physical Removal

T2: Node added

Lost Update!
The Solution: Combine Bit and Pointer

Logical Removal = Set Mark Bit

Mark-Bit and Pointer are CASed together
(AtomicMarkableReference)

Physical Removal

Fail CAS: Node not added after logical Removal
Marking a Node

- **AtomicMarkableReference** class
  - Java.util.concurrent.atomic package
public Object get(boolean[] marked);
public Object get(boolean[] marked);

Returns reference
Returns mark at array index 0!
Extracting Mark Only

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

Value of mark
public boolean compareAndSet(
    Object expectedRef,
    Object updateRef,
    boolean expectedMark,
    boolean updateMark);
public boolean compareAndSet(
    Object expectedRef,
    Object updateRef,
    boolean expectedMark,
    boolean updateMark);

If this is the current reference ...

And this is the current mark ...
Changing State

```java
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);
```
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.
Removing a Node

remove(c)
Removing a Node

```
remove(b)
CAS
CAS
remove(c)
failed
```
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`
The Window Class

class Window {
    public Node pred;
    public Node curr;
    Window(Node pred, Node curr) {
        pred = pred; curr = curr;
    }
}
The Window Class

class Window {
    public Node pred;
    public Node curr;

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

A container for pred and current values
Using the Find Method

```python
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

Window window = find(item);

At some instant,

pred ——— item ——— curr ——— succ

or ...
The Find Method

Window window = find(item);

At some instant,
pred curr = null succ

item

not in list
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.compareAndSet(succ, succ, false, 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.compareAndSet(succ, succ, false, 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.compareAndSet(succ, succ, false, 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.compareAndSet(succ, succ, false, 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.compareAndSet(succ, succ, false, true);
            if (!snip) continue;
            pred.next.compareAndSet(curr, succ, false, false);
            return true;
        }
    }
}
public boolean remove(T item) {
    Node pred, curr = find(head, key);
    return false;
    if (curr.key != key) {
        Node succ = curr.next.getReference();
        snip = curr.next.compareAndSet(succ, succ, false, 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.compareAndSet(succ, succ, false, 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.compareAndSet(succ, succ, false, 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.compareAndSet(succ, succ, false, 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;}
        }
    }
}
public boolean add(T item) {
    boolean splice;
    while (true) {
        Window window = find(head);
        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;}
        }
    }
}
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; }
        }
    }
}
Wait-free Contains

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

Get ref to successor and current deleted bit
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;
        }
    }
}

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();
        if (curr.key >= key) return new Window(pred, curr);
        succ = curr.next.get(marked);
        while (marked[0]) {
            ...}
        }
        return new Window(pred, curr);
    }
}
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;
        }
    }
}
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);
  }
  ...
}
Lock-free Find

if predecessor’s next field changed, 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);
  }
  ...
}
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);
    }
    ...
}

Otherwise move on to check if next node deleted
Performance

- Different list-based set implementations
- SunFire 6800 (bus based cache coherence)
- 16-node machine, each 1.2 GHz
- Vary percentage of `contains()` calls
High Contains Ratio

Ops/sec (90% contain, 9% add, 1% remove)
As Contains Ratio Increases

Ops/sec using 32 threads

% Contains()

Lock-free
Lazy list
Coarse Grained
Fine Lock-coupling
“To Lock or Not to Lock”

• Locking vs. Non-blocking:
  – Depending on the application usage

• The answer: nobler to compromise
  – Example: Lazy list combines blocking \texttt{add()} and \texttt{remove()} and a wait-free \texttt{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.
Business as Usual
Business as Usual
Business as Usual
Business as Usual

remove(b)
Business as Usual

a not marked
Business as Usual

a still points to b
Business as Usual

Logical delete
Business as Usual
Business as Usual