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

• Object = linked set of components
• Search without locking ...
• If you find it, lock and check ...
  - OK, we are done
  - Oops, try again
• Evaluation
  - 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
  - Robust against asynchrony

• Disadvantages
  - Complex
  - Sometimes high overhead
Linked List

• Illustrate these patterns ...

• Using a linked-list class
  - Common application
  - Building block for other apps
Set Interface

- Unordered collection of objects
- No duplicates
- Methods
  - Add a new object
  - Remove an object
  - Test if object is present
List-Based Sets

```java
public interface Set {
    public boolean add(Object x);
    public boolean remove(Object x);
    public boolean contains(Object x);
}
```
List-Based Sets

```java
public interface Set {
    public boolean add(Object x);
    public boolean remove(Object x);
    public boolean contains(Object x);
}

Add object to set
```
List-Based Sets

public interface Set {
    public boolean add(Object x);
    public boolean remove(Object x);
    public boolean contains(Object x);
}

Remove object from set
List-Based Sets

public interface Set {
    public boolean add(Object x);
    public boolean remove(Object x);
    public boolean contains(Object x);
}

Is object in set?
List Entry

```java
public class Entry {
    public Object object;
    public int key;
    public Entry next;
}
```
List Entry

```java
public class Entry {
    public Object object;
    public int key;
    public Entry next;
}
```

Object of interest
public class Entry {
    public Object object;
    public int key;
    public Entry next;
}
public class Entry {
    public Object object;
    public int key;
    public Entry next;
}

List Entry

Sorting makes it easy to detect absence
List Entry

```java
public class Entry {
    public Object object;
    public int key;
    public Entry next;
}
```

Reference to next entry
List-Based Set

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

- **Invariant**
  - Property that always holds

- **Established by**
  - 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

• Proof that invariants preserved works only if
  - methods considered
  - are the only modifiers
• Language encapsulation helps
  - List entries not visible outside class
Interference

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

• Garbage-collection helps here
Abstract Data Types

• Concrete representation

• Abstract Type
  - \{a, b\}
Abstract Data Types

- **Meaning of rep given by abstraction map**

  \[ S(\begin{array}{c}
  a \\
  \end{array}) = \{a,b\} \]
Rep Invariant

• Which concrete values are meaningful?
  - Sorted? Duplicates?

• Rep invariant
  - Characterizes legal concrete reps
  - Preserved by methods
  - Relied on by methods
Blame Game

• Rep invariant is a contract
• Suppose
  – add() leaves behind 2 copies of $x$
  – remove() removes only 1
• Which one is incorrect?
Blame Game

• Suppose
  – add() leaves behind 2 copies of \( x \)
  – remove() removes only 1

• Which one is incorrect?
  - If rep invariant says no duplicates
    • add() is incorrect
  - Otherwise
    • remove() is incorrect
Shorthand

• $a \rightarrow b$ means $a\.next = b$
• $a \Rightarrow b$ means $b$ reachable from $a$
  - $a \Rightarrow a$
  - If $a \Rightarrow b$ and $b \rightarrow c$ then $a \Rightarrow c$
Rep Invariant (partly)

- Sentinel nodes
  - head \Rightarrow tail
- Sorted, no duplicates
  - If \ a \rightarrow \ b \ \text{then} \ a.key < b.key
Abstraction Map

- $S(\text{head}) =$
  - $\{ x \mid \text{there exists } a \text{ such that}$
    - head $\Rightarrow a$ and
    - $a.\text{object} = x$
  - $\}$
Adding an Entry
Adding an Entry

a \rightarrow b \rightarrow c
Adding an Entry

Adding an Entry

Adding an Entry

Adding an Entry

Adding an Entry

Adding an Entry

Adding an Entry
Adding an Entry

- a
- b
- c

- b
Removing an Entry
Removing an Entry

\[ \text{a} \rightarrow \text{b} \rightarrow \text{c} \]
Removing an Entry
Removing an Entry
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
Optimistic Synchronization

• Requires very careful thought
  - “Do not meddle in the affairs of dragons, for you are crunchy and taste good with ketchup.”

• Try it without synchronization
  - If you win, you win
  - If not, try it again with synchronization
Lock-Free Synchronization

• Dump locking altogether ...
  - “You take the red pill and you stay in Wonderland and I show you how deep the rabbit-hole goes”

• No locks, just native atomic methods
  - Usually compareAndSet()
Hand-over-Hand locking

© 2005 Herlihy & Shavit
Hand-over-Hand locking
Hand-over-Hand locking
Hand-over-Hand locking
Hand-over-Hand locking
Removing an Entry

remove b
Removing an Entry

remove b
Removing an Entry

```
(a) -> (b) -> (c) -> (d)
```

remove b
Removing an Entry

remove b
Removing an Entry

- Remove b
- Remove c
Removing an Entry
Removing an Entry

remove b

remove c
Removing an Entry

remove b

remove c
Removing an Entry

remove b

remove c
Removing an Entry

```
| a | b | c | d |
```

- Remove b
- Remove c
Removing an Entry

remove b
remove c
Removing an Entry
Removing an Entry

remove b
remove c
Uh, Oh

remove b

remove c

© 2005 Herlihy & Shavit
Problem

• To delete entry b
  - Swing entry a’s next field to c

• Problem is,
  - Someone could delete c concurrently.
Insight

• If an entry is locked
  - No one can delete entry’s successor

• If a thread locks
  - Entry to be deleted
  - And its predecessor
  - Then it works
Hand-Over-Hand Again

remove b
Hand-Over-Hand Again

remove b
Hand-Over-Hand Again

remove b
Hand-Over-Hand Again

Found it!

remove b
Hand-Over-Hand Again

Found it!

remove b
Hand-Over-Hand Again

remove b

Found it!
Removing an Entry

© 2005 Herlihy & Shavit
Removing an Entry

\[ \text{remove } b \]

\[ \text{remove } c \]

© 2005 Herlihy & Shavit
Removing an Entry
Removing an Entry

remove b

remove c
Removing an Entry

remove b

remove c
Removing an Entry

remove b

remove c
Removing an Entry
Removing an Entry

- Remove b
- Remove c
- d
Removing an Entry

- Remove b
- Remove c

Diagram showing the removal of an entry from a data structure.
Removing an Entry
public boolean remove(Object object) {
    int key = object.hashCode();
    Entry pred, curr;
    try {
        ...
    } finally {
        curr.unlock();
        pred.unlock();
    }
}
public boolean remove(Object object) {
    int key = object.hashCode();
    Entry pred, curr;
    try {
        ...
    } finally {
        curr.unlock();
        pred.unlock();
    }
}
```java
public boolean remove(Object object) {
    int key = object.hashCode();
    Entry pred, curr;
    try {
        ...
    } finally {
        currEntry.unlock();
        predEntry.unlock();
    }
}
```

**Predecessor and current entries**
public boolean remove(Object object) {
    int key = object.hashCode();
    Entry pred, curr;
    try {
        ...
    } finally {
        curr.unlock();
        pred.unlock();
    }
    Make sure locks released
}
public boolean remove(Object object) {
    int key = object.hashCode();
    Entry pred, curr;
    try {
        ...}
    } finally {
        curr.unlock();
        pred.unlock();
    }

Everything else
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

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

try {
    pred = this.head;
    pred.lock();
    curr = pred.next;
    curr.lock();
    ... 
} finally { ... }

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

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

At start of each loop: curr and predy locked
Remove: searching

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

If entry found, remove it
while (curr.key <= key) {
    if (object == curr.object) {
        pred.next = curr.next;
        return true;
    }
    pred.unlock();
    pred = curr;
    curr = curr.next;
    curr.lock();
}

If entry found, remove it
Remove: searching

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

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

Only one entry locked!
Remove: searching

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

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

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

Lock invariant restored
Remove: searching

```
while (curr.key <= key) {
    if (object == curr.object) {
        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 entry $e$
  - Must lock $e$
  - Must lock $e$'s predecessor

• Therefore, if you lock an entry
  - It can't be removed
  - And neither can its successor
First Invariant

• Different threads have different pred values

• If $A \neq B$, and $\text{pred}_A \neq \text{null}$
  - Then $\text{pred}_A \neq \text{pred}_B$
1st Invariant

- If $A \neq B$, and $\text{pred}_A \neq \text{null}$
  - Then $\text{pred}_A \neq \text{pred}_B$
- Holds initially
- Must show it is preserved
Claim

- **If** \( \text{pred}_A \neq \text{null} \) then \( A \) holds lock
  - True at start when \( \text{pred}_A \) is head
  - \( \text{curr}_A \) locked before assigned to \( \text{pred}_A \)
  - Other statements don’t change \( \text{pred}_A \)
1st Invariant

- If $\text{pred}_A \neq \text{null}$
  - then $A$ holds lock
- If $\text{pred}_B \neq \text{null}$
  - then $B$ holds lock
- Must be distinct
2nd Invariant

• Threads never traverse deleted entries

• If $\text{pred}_A \neq \text{null}$
  - Then $\text{head} \Rightarrow \text{pred}_A \Rightarrow \text{tail}$
2\textsuperscript{nd} Invariant

- True initially
- \( A \) holds lock for \( \text{pred}_A \) throughout traversal
- No other thread can remove it
- So head \( \Rightarrow \) \( \text{pred}_A \) is invariant.
- Same for \( \text{pred}_A \Rightarrow \) tail
Why remove() is linearizable

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

head \Rightarrow \text{pred}_A \Rightarrow \text{curr}_A

so the object is in the set
Why remove() is linearizable

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

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

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

Linearization point
Why remove() is linearizable

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

\[ \text{pred}_A \rightarrow \text{curr}_A \]
\[ \text{pred}_A.key < \text{key} \]
\[ \text{key} < \text{curr}_A.key \]
Why remove() is linearizable

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

Linearization point: when $\text{curr}_A$ set to entry with higher key
Adding Entries

• To add entry $e$
  - Must lock predecessor
  - Must lock successor

• Neither can be deleted
  - (Is successor lock actually required?)
Rep Invariant

• Easy to check that
  - Tail always reachable from head
  - Entries 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 entries without locking
• Lock entries
• Check that everything is OK
Invariants

• Invariants no longer hold
  - OK to scan deleted elements
• But we establish properties by
  - Validation
  - After we lock target entries
Key Property

- **Fine-grained synchronization**
  - head $\Rightarrow$ pred$_A$ $\Rightarrow$ tail
  - Is invariant

- **Optimistic synchronization**
  - Validation checks same property
  - After the fact
  - Must restart if validation fails
Removing an Entry

```
|   |   | a |   | b |   | c |   | d |
```

remove c

return true
What Can Go Wrong?
Check that Entry is Still Accessible
What Can Go Wrong?

- Remove c
- Add b'
What Can Go Wrong?

What can go wrong with removing element c from the linked list?
Check that Entries Still Adjacent

© 2005 Herlihy & Shavit
Correctness

• If
  - Entries b and c both locked
  - Entry b still accessible
  - Entry c still successor to b
• Then
  - Neither will be deleted
  - OK to delete and return true
Removing an Absent Entry

remove c

return false
Correctness

• If
  - Entries $b$ and $d$ both locked
  - Entry $b$ still accessible
  - Entry $d$ still successor to $b$
• Then
  - Neither will be deleted
  - No thread can add $c$ after $b$
  - OK to return $false$
1st Invariant

- Different threads have different \texttt{pred} values if they’re locked
- If $A \neq B$, and $\texttt{pred}_A \neq \texttt{null}$ and locked
  - Then $\texttt{pred}_A \neq \texttt{pred}_B$
2\textsuperscript{nd} Invariant

• An entry will remain reachable from $\text{pred}_A$ as long as it is reachable from the head

• For all reachable $a$,
  - If $\text{pred}_A \neq \text{null}$, $\text{pred}_A.key < a.key$
  - Then $\text{pred}_A \Rightarrow a$
Validation

```java
private boolean validate(Entry pred,
    Entry curry) {
    Entry entry = head;
    while (entry.key <= pred.key) {
        if (entry == pred) {
            return pred.next == curr;
        }
        entry = entry.next;
    }
    return false;
}
```
private boolean validate(Entry pred, Entry curr) {
Entry entry = head;
while (entry.key <= pred.key) {
    if (entry == pred) {
        return pred.next == curr;
    }
    entry = entry.next;
}
return false;
}
private boolean validate(Entry pred, Entry curr) {
    Entry entry = head;
    while (entry.key <= pred.key) {
        if (entry == pred) {
            if (entry == pred)
                return pred.next == curr;
            entry = entry.next;
        }
    }
    return false;
}
private boolean validate(Entry pred, Entry curr) {
    Entry entry = head;
    while (entry.key <= pred.key) {
        if (entry == pred) {
            if (entry == pred)
                return pred.next == curr;
            entry = entry.next;
        }
        return false;
    }
}
private boolean validate(Entry pred, Entry curr) {
    Entry entry = head;
    while (entry.key <= pred.key) {
        if (entry == pred) {
            return pred.next == curr;
        }
        entry = entry.next;
    }
    return false;
}
private boolean validate(Entry pred, Entry curry) {
    Entry entry = head;
    while (entry.key <= pred.key) {
        if (entry == pred)
            return pred.next == curr;
        entry = entry.next;
    }
    return false;
}
private boolean validate(Entry pred, Entry curr) {
    Entry entry = head;
    while (entry.key <= pred.key) {
        if (entry == pred) {
            return pred.next == curr;
        } else {
            entry = entry.next;
        }
    }
    return false;
}
private boolean validate(Entry pred, Entry curr) {
    Entry entry = head;
    while (entry.key <= pred.key) {
        if (entry == pred) {
            return pred.next == curr;
        }
        entry = entry.next;
    }
    return false;
}
Remove: searching

public boolean remove(Object object) {
    int key = object.hashCode();
    retry: while (true) {
        Entry pred = this.head;
        Entry curr = pred.next;
        while (curr.key <= key) {
            if (object == curr.object)
                break;
            pred = curr;
            curr = curr.next;
        } ...
    }
public boolean remove(Object object) {
    int key = object.hashCode();
    retry: while (true) {
        Entry pred = this.head;
        Entry curr = pred.next;
        while (curr.key <= key) {
            if (object == curr.object) break;
            pred = curr;
            curr = curr.next;
        } ...
    }
}
public boolean remove(Object object) {
    int key = object.hashCode();
    
    retry: while (true) {
        Entry pred = this.head;
        Entry curr = pred.next;
        while (curr.key <= key) {
            if (object == curr.object)
                break;
            pred = curr;
            curr = curr.next;
        }
    }

    Retry on synchronization conflict
public boolean remove(Object object) {
    int key = object.hashCode();
    retry: while (true) {
        Entry pred = this.head;
        Entry curr = pred.next;
        while (curr.key <= key) {
            if (object == curr.object)
                break;
            pred = curr;
            curr = curr.next;
        }
        Remove:
        Examine predecessor and current entries
    }
}
public boolean remove(Object object) {
    int key = object.hashCode();
    retry: while (true) {
        Entry pred = this.head;
        Entry curr = pred.next;
        while (curr.key <= key) {
            if (object == curr.object)
                break;
            pred = curr;
            curr = curr.next;
        } …
    }

    Search by key
public boolean remove(Object object) {
    int key = object.hashCode();
    retry: while (true) {
        Entry pred = this.head;
        Entry curr = pred.next;
        while (curr.key <= key) {
            if (object == curr.object)
                break;
            pred = curr;
            curr = curr.next;
        } …
    }
    Stop if we find object
    break;
}
public boolean remove(Object object) {
    int key = object.hashCode();
    retry: while (true) {
        Entry pred = this.head;
        Entry curr = pred.next;
        while (curr.key <= key) {
            if (object == curr.object)
                break;
            pred = curr;
            curr = curr.next;
        } ...
    }
}
On Exit from Loop

• If object is present
  - curr holds object
  - pred just before curr

• If object is absent
  - curr has first higher key
  - pred just before curr

• Assuming no synchronization problems
Remove Method

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

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

Lock both entries
Remove Method

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

Check for synchronization conflicts
Remove Method

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

Object found, remove entry
Remove Method

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

Object not found
So Far, So Good

• Much less lock acquisition/release
  - Performance
  - Concurrency

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

• Optimistic works if cost of
  - scanning twice without locks <
  - Scanning once with locks
• Drawback
  - Contains() acquires locks
  - 90% of calls in many apps
Lazy List

• Like optimistic, except
  - Scan once
  - Contains() never locks ...

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

• **Remove Method**
  - Scans list (as before)
  - Locks predecessor & current (as before)

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

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

• All Methods
  - Scan through marked entry
  - Removing an entry doesn’t slow down other method calls ...

• Must still lock $\text{pred}$ and $\text{curr}$ entries.
Validation

• No need to rescan list!
• Check that \textit{pred} is not marked
• Check that \textit{curr} is not marked
• Check that \textit{pred} points to \textit{curr}
Business as Usual

```

```

© 2005 Herlihy & Shavit
Business as Usual
Business as Usual
Business as Usual
Business as Usual
Interference
Interference
Interference
Interference

© 2005 Herlihy & Shavit
Validation

b not marked
Interference

© 2005 Herlihy & Shavit
Interference

b still points to c
Logical Delete
Scan Through

So what?

Remove c
Physical Deletion

So what?

Remove c
New Abstraction Map

\[ S(\text{head}) = \]
\[ \{ x \mid \text{there exists entry } a \text{ such that} \]
\[ \begin{align*}
& \cdot \text{head } \rightarrow a \text{ and} \\
& \cdot a.\text{object } = x \text{ and} \\
& \cdot a \text{ is unmarked}
\end{align*} \]
\[ \} \]
Modified Invariant

• If A’s pred entry is unmarked, then it is reachable

• If $\text{pred}_A \neq \text{null}$ and is not marked
  - Then $\text{head} \Rightarrow \text{pred}_A \Rightarrow \text{tail}$
Invariant

- Holds initially
- Not modified by `add()` or `contains()`
- Remove()?
  - Marking doesn’t violate invariant
    - No entry made unreachable
  - Physical remove doesn’t violate
    - Entry made unreachable is already marked
Modified Invariant

• If $\text{pred}_A \neq \text{null}$ and is not marked
  - Then $\text{head} \Rightarrow \text{pred}_A \Rightarrow \text{tail}$

• Justifies why $\text{contains}()$ doesn’t need to lock
  - Unmarked reachable entry
  - Remains reachable
  - As long as it remains unmarked
Validation

private boolean validate(Entry pred, Entry curr) {
    return !pred.next.marked &&
           !curr.next.marked &&
           pred.next == curr);
}
List Validate Method

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

Predecessor not
Logically removed
private boolean validate(Entry pred, Entry curr) {
    return !pred.next.marked && !curr.next.marked && curr.next == curr;
}
private boolean validate(Entry pred, Entry curr) {
    return !pred.next.marked && !curr.next.marked &&
           pred.next == curr);
}
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;
}}
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;
    }
}
```

Validate as before
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();
}}
```

Key found
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();
}
```
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;
        }
    }
    return false;
} finally {
    pred.unlock();
    curr.unlock();
}
public boolean contains(Object object) {
    int key = object.hashCode();
    Entry curr = this.head;
    while (curr.key < key) {
        curr = curr.next;
    }
    return curr.key == key && !curr.marked;
}
Contains

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

public boolean contains(Object object) {
    int key = object.hashCode();
    Entry curr = this.head;
    while (curr.key < key) {
        curr = curr.next;
    }

    return curr.key == key && !curr.marked;
}
Evaluation

• **Good:**
  - Contains method doesn’t need to lock
  - Uncontended calls don’t re-traverse

• **Bad**
  - Contended 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” (stops running)
    • Cache miss, page fault, descheduled ...
    • Software error, ...

  - Everyone else using that lock is stuck!
Lock-Free Data Structures

• No matter what ...
  - Some thread will complete method call
  - Even if others halt at malicious times

• Implies that
  - You can’t use locks (why?)
  - Um, that’s why they call it lock-free
Lock-Free ≠ Wait-Free

- **Wait-free synchronization**
  - Every method call eventually finishes
  - What everyone really wants
- **Lock-free synchronization**
  - Some method call eventually finishes
  - What we are usually willing to pay for
    - Starvation rare in practice ...
Lock-Free Lists

• Next logical step
• Eliminate locking entirely
• Use only `compareAndSet()`
• What could go wrong?
Adding an Entry

a → b → c
Adding an Entry

```
© 2005 Herlihy & Shavit
BROWN
```

- a
- b
- c
- b

Diagram showing the addition of a new entry between a and b.
Adding an Entry

CAS

b

b

c
Adding an Entry

![Diagram showing a linked list with nodes labeled a, b, and c, and an arrow indicating the addition of a new entry labeled b.]
Adding an Entry

```
    a -> b
   /   \
 /     /
b ----> c
```

© 2005 Herlihy & Shavit
Removing an Entry

remove b

remove c
Look Familiar?

remove b

remove c
Problem

- Method updates entry’s next field
- After entry has been removed
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

![Diagram of a node with a reference, address, and mark bit]
Extracting Reference & Mark

Public Object get(boolean[]);
Extracting Reference & Mark

Public Object get(boolean[]);

Returns reference

Returns mark at array index 0!
Extracting Reference Only

public boolean isMarked();
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 ...
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

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

```java
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 an Entry
Removing an Entry

CAS CAS

failed

remove c

remove b

© 2005 Herlihy & Shavit
Removing an Entry

```
  a  b  c  d
  ^--^--^--^--
   ↑    ↑    ↑
   ✡    ✡    ✡
   |    |    |
   v    v    v
```

remove b
remove c
Removing an Entry

```
\[ \text{remove } b \]  
\[ \text{remove } c \]
```
Traversing the List

• **Q:** what do you do when you find a “logically” deleted entry in your path?

• **A:** finish the job.
  - CAS the predecessor’s next field
  - Proceed (repeat as needed)
Lock-Free Traversal
The Find Method

`pred, curr, next = find(object);`
The Find Method

\[ \text{pred, curr, succ} = \text{find(object)}; \]

At some instant, \( \text{pred, curr, succ} \) or ...

\[ \text{pred} \quad \text{curr} \quad \text{succ} \]
The Find Method

\[ \text{pred, curr, succ} = \text{find}(\text{object}); \]

At some instant, \( \text{curr} = \text{null} \), \( \text{object} \) is not in list.
public boolean remove(Object object) {
    while (true) {
        pred, curr, succ = find(object);
        if (curr == null)
            return false;
        if (!curr.next.attemptMark(succ, true))
            continue;
        pred.next.compareAndSet(curr, succ, false, false);
        return true;
    }
}

Remove
public boolean remove(Object object) {
    
    while (true) {
        pred, curr, succ = find(object);
        if (curr == null)
            return false;
        if (!curr.next.attemptMark(succ, true))
            continue;
        pred.next.compareAndSet(curr, succ, false, false);
    }
    return true;
}
public boolean remove(Object object) {
    while (true) {
        pred, curr, succ = find(object);
        if (curr == null)
            return false;
        if (!curr.next.attemptMark(succ, true))
            continue;
        pred.next.compareAndSet(curr, succ, false, false);
        return true;
    }
}
public boolean remove(Object object) {
    while (true) {
        pred, curr, succ = find(object);
        if (curr == null)
            return false;
        if (!curr.next.attemptMark(succ, true))
            continue;
        pred.next.compareAndSet(curr, succ, false, false);
        return true;
    }
}
Remove

```java
public boolean remove(Object object) {
    while (true) {
        pred, curr, succ = find(object);
        if (curr == null)
            return false;
        if (!curr.next.attemptMark(succ, true))
            continue;
        pred.next.compareAndSet(curr, succ, false, false);
        return true;
    }
}
```

Try to mark entry as deleted
Remove

If it doesn't work, just retry

```java
public boolean remove(Object object) {
    while (true) {
        pred, curr, succ = find(object);
        if (curr == null)
            return false;
        if (!curr.next.attemptMark(succ, true))
            continue;
        pred.next.compareAndSet(curr, succ, false, false);
        return true;
    }
}
```
public boolean remove(Object object) {
    while (true) {
        pred, curr, succ = find(object);
        if (curr == null)
            return false;
        if (!curr.next.attemptMark(succ, true))
            continue;
        pred.next.compareAndSet(curr, succ, false, false);
        return true;
    }
}

If it works, our job is (essentially) done
public boolean remove(Object object) {
    while (true) {
        pred, curr, succ = find(object);
        if (curr == null)
            return false;
        if (!curr.next.attemptMark(succ, true))
            continue;
        pred.next.compareAndSet(curr, succ, false, false);
        return true;
    }
}

Try to advance reference (if we don't succeed, someone else did).
if (curr == null)
    return false;
if (!curr.next.attemptMark(succ, true))
    continue;
pred.next.compareAndSet(curr, succ, false, false);
return true;
Add

```java
public boolean add(Object object) {
    while (true) {
        pred, curr, succ = find(object);
        if (curr != null)
            return false;
        Entry entry = new Entry(object);
        entry.next = new AMR(succ, false);
        if (pred.next.CAS(succ, entry, false, false))
            return true;
    }
}
```
Add

```java
public boolean add(Object object) {
    while (true) {
        pred, curr, succ = find(object);
        if (curr != null)
            return false;
        Entry entry = new Entry(object);
        entry.next = new AMR(succ, false);
        if (pred.next.CAS(succ, entry, false, false))
            return true;
    }
    return false;  // Object already there.
```
public boolean add(Object object) {
    while (true) {
        pred, curr, succ = find(object);
        if (curr != null)
            return false;
        Entry entry = new Entry(object);
        entry.next = new AMR(succ, false);
        if (pred.next.CAS(succ, entry, false, false))
            return true;
    }
}
public boolean add(Object object) {
    Entry entry = new Entry(object);
    entry.next = new AMR(succ, false);
    if (pred.next.CAS(succ, entry, false, false))
        return true;
    while (true) {
        pred, curr, succ = find(object);
        if (curr != null)
            return false;
        }
    return true;
}
Contains

```java
public boolean contains(Object obj){
    while (true) {
        prev, curr, succ = find(object);
        return (curr != null);
    }
}
```
public boolean contains(Object obj) {
    while (true) {
        prev, curr, succ = find(object);
        return (curr != null);
    }
}
private Entry, Entry, Entry
  find(Object object) {
    Entry pred, curr, succ;
    boolean[] pmark = new boolean[1];
    boolean[] cmark = new boolean[1];
    int key = object.hashCode();
    tryAgain: while (true) {
      ...
    }}
}
private Entry, Entry, Entry
    find(Object object) {
        Entry pred, curr, succ;
        boolean[] pmark = new boolean[1];
        boolean[] cmark = new boolean[1];
        int key = object.hashCode();
        tryAgain: while (true) {
            ...
        }
    }
Find

private Entry, Entry, Entry
find(Object object) {
Entry pred, curr, succ;

boolean[] pmark = new boolean[1];
boolean[] cmark = new boolean[1];

int key = object.hashCode();
tryAgain: while (true) {
    ...
}}

Deleted bits for pred and curr
Find

```java
private Entry,Entry,Entry
    find(Object object) { 
    Entry pred, curr, succ;
    boolean[] pmark = new boolean[1];
    boolean[] cmark = new boolean[1]; 
    int key = object.hashCode(); 

    tryAgain: while (true) {
        ...
    }
}
```

If list changes while traversed, start over
private Entry, Entry, Entry
    find(Object object) {
        Entry pred, curr, succ;
        boolean[] pmark = new boolean[1];
        boolean[] cmark = new boolean[1];
        int key = object.hashCode();
        tryAgain:
            while (true) {
                ...
            }
    }

Lock-Free because we start over only if someone else makes progress
Find

tyAgain: while (true) {
    pred = this.head.getReference();
    curr = pred.next.get(pmark);
    while (true) {
        ...
    }
}}

Start with first two entries
Find

tryAgain: while (true) {
    pred = this.head.getReference();
    curr = pred.next.get(pmark);
    while (true) {
        ...
    }
}
Find

... while (true) {
    if (curr == null)
        return pred, null, succ;
    succ = curr.next.get(cmark);
    int ckey = curr.key;
    if (isChanged(pred.next))
        continue tryAgain;
}}

Fell off the end of the list
Find

... while (true) {
    if (curr == null)
        return pred, null, succ;
    succ = curr.next.get(cmark);
    int ckey = curr.key;
    if (isChanged(pred.next))
        continue tryAgain;
}}

Get ref to successor and current deleted bit
Find

... 
while (true) { 
  if (curr == null) 
    return pred, null, succ; 
  succ = curr.next.get(cmark); 
  int ckey = curr.key; 
  if (isChanged(pred.next)) 
    continue tryAgain; 
}

Panic if predecessor's next field changed
Find

while (true) {

if (!cmark[0]) {
    if (curr.object == object)
        return pred, curr, succ;
    else if (ckey <= key) {
        pred = curr;
    } else
        return prev, null, curr;
} else {
    ...}
}}

If current node is not deleted
while (true) {
    …
    if (!cmark[0]) {
        if (curr.object == object)
            return pred, curr, succ;
        else if (ckey <= key) {
            pred = curr;
        } else
            return prev, null, curr;
    } else {
        …
    }}

Object found
while (true) {
    ...
    if (!cmark[0]) {
        if (curr.object == object)
            return pred, curr, succ;
        else if (ckey <= key) {
            pred = curr;
        } else
            return prev, null, curr;
    } else {
        ...
    }}
while (true) {
    ...
    if (!cmark[0]) {
        if (curr.object == object)
            return pred, curr, succ;
        else if (ckey <= key) {
            pred = curr;
        } else {
            return prev, null, curr;
        } else {
            ...
        }
    } else {
    ...
    }}

Not there, give up
Find

... while (true) {
    ...
    if (!cmark[0]) {
        ...
    } else {
        if (pred.next.compareAndSet(curr, succ, false, false))
            continue;
        else
            continue tryAgain;
    }
    
Current entry is logically deleted
Find

Try to redirect predecessor's next reference

```java
while (true) {
    ...
    if (!cmark[0]) {
        ...
    } else {
        if (pred.next.compareAndSet(curr, succ, false, false))
            continue;
        else
            continue tryAgain;
    }
}
while (true) {
    if (!cmark[0]) {
        
    } else {
        if (pred.next.compareAndSet(
            curr, succ, false, false))
            continue;
        else
            continue tryAgain;
    }
}

On success, keep going, on failure, start over
Summary

- Coarse-grained locking
- Fine-grained locking
- Optimistic synchronization
- Lock-free synchronization
Scratch
Scratch
Scratch
Scratch
Scratch
Scratch
Removing an Entry

```
oc b c
```

**CAS**