Last Two Lectures: Synchronization Primitives
Today: Concurrent Objects

- Adding threads should not lower throughput
  - Contention effects
  - Mostly fixed by Queue locks
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
Coarse-Grained Synchronization

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

• Each method locks the object
  – Avoid contention using queue locks
  – Easy to reason about
    • In simple cases

• So, are we done?
Coarse-Grained Synchronization

• Sequential bottleneck
  – Threads “stand in line”
Coarse-Grained Synchronization

• Sequential bottleneck
  – Threads “stand in line”

• Adding more threads
  – Does not improve throughput
  – Struggle to keep it from getting worse
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 …
This Lecture

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

• For highly-concurrent objects
  – 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 …
Second: Optimistic Synchronization

- Search without locking …
- If you find it, lock and check …
  - OK: we are done
  -Oops: start over
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 …
Fourth:
Lock-Free Synchronization

• Don’t use locks at all
  – Use compareAndSet() & relatives …

• Advantages
  – No Scheduler Assumptions/Support
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
Set Interface

- Unordered collection of items
- No duplicates
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-Based Sets

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

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

Usually hash code
List Node

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

Reference to next node
The List-Based Set

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

- Invariant
  - Property that always holds
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()`
Specifically …

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

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

• Invariants make sense only if
  – methods considered
  – are the only modifiers
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
Abstract Data Types

- Concrete representation:
- Abstract Type:
  - \{a, b\}
Abstract Data Types

- Meaning of representation given by abstraction map

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

• Which concrete values 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 is incorrect?
Blame Game

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

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

• Which is incorrect?
  – If rep invariant says \textit{no duplicates}
    • \texttt{add()} is incorrect
  – Otherwise
    • \texttt{remove()} is incorrect
Rep Invariant (partly)

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

• \( S(\text{head}) = \)
  
  \[ \{ x \mid \text{there exists a such that} \]
  
  • a reachable from head and
  
  • a.item = x

  \[ \} \]
Sequential List Based Set

add()

remove()
Sequential List Based Set

add()

remove()
Coarse-Grained Locking
Coarse-Grained Locking
Coarse-Grained Locking

Simple but hotspot + bottleneck
Coarse-Grained Locking

• Easy, same as synchronized methods
  – “One lock to rule them all …”
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

[Diagram showing the process of hand-over-hand locking with elements labeled a, b, and c.]
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 hold 2 locks?
Concurrent Removes

\[
\text{remove}(b), \quad \text{remove}(c)
\]
Concurrent Removes

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

remove(b)

remove(c)
Concurrent Removes

remove(b)

remove(c)
Concurrent Removes

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

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

remove(b)

remove(c)
Concurrent Removes

\[ \text{remove}(b) \quad \text{remove}(c) \]
Concurrent Removes

```
remove(b)
```

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

- remove(b)
- remove(c)
Uh, Oh

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

- 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
Hand-Over-Hand Again

remove(b)
Hand-Over-Hand Again

remove(b)
Hand-Over-Hand Again

remove(b)
Hand-Over-Hand Again

remove(b)

Found it!
Hand-Over-Hand Again

remove(b)

Found it!
Hand-Over-Hand Again

remove(b)
Removing a Node

```
remove(b)
```

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

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

Cannot acquire lock for b

remove(c)
Removing a Node

Wait!
Removing a Node

Proceed to remove(b)
Removing a Node

remove(b)
Removing a Node

remove(b)
Removing a Node

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

![Diagram showing the process of removing a node from a linked list, with nodes labeled a and d.](image_url)
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();
    }
}
public boolean remove(Item item) {
    int key = item.hashCode();
    Node pred, curr;
    try {
        ... 
    } finally {
        currNode.unlock();
        predNode.unlock();
    }
}

Predecessor and current nodes
Remove method

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

Make sure locks released
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 { … }
```

lock pred == head
Remove method

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

Lock current
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;
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
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
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

Only one node locked!

```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;
```
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;

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;
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
• So **curr**.item is in the set

Why remove() is linearizable
Why remove() is linearizable

```c
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 if item is present
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;
```
Why remove() is linearizable

```c
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;
```

Item not present
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

```
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**
Adding Nodes

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

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

• $S(\text{head}) =$
  
  \[ \{ x \mid \text{there exists } a \text{ such that} \]
  
  • $a$ reachable from head and
  
  • $a.\text{item} = x$

  \[ \} \]
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
Optimistic Synchronization

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

```
add(c)     a       b       d       e
```

Aha!
Optimistic: Lock and Load

add(c)
Optimistic: Lock and Load

add(c)
What could go wrong?

add(c)
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?

add(c)

Uh-oh
Validate – Part 1

```
add(c)
```

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

add(c)

Aha!
What Else Could Go Wrong?

add(c)

add(b’)

Diagram showing a sequence of nodes connected by arrows, with some nodes marked with a lock symbol.
What Else Could Go Wrong?
What Else Could Go Wrong?

add(c)
What Else Could Go Wrong?

add(c)
Validate Part 2 (while holding locks)

- add(c)
- Yes, b still points to d
Optimistic: Linearization Point

add(c)
Same Abstraction Map

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

Aha!

remove(c)
Validate (1)

Yes, still reachable from head

remove(c)
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;
}

Validation
Predecessor & current nodes
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;
}  

Predecessor reachable
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;
}

Predecessor not reachable
public boolean remove(Item item) {
    int key = item.hashCode();
    retry: while (true) {
        Node pred = this.head;
        Node curr = pred.next;
        while (curr.key <= key) {
            if (item == curr.item)
                break;
            pred = curr;
            curr = curr.next;
        }
    ...
}
public boolean remove(Item item) {
    int key = item.hashCode();
    retry: while (true) {
        Node pred = this.head;
        Node curr = pred.next;
        while (curr.key <= key) {
            if (item == curr.item)
                break;
            pred = curr;
            curr = curr.next;
        } …
    }

    // Further logic...
}
```java
public boolean remove(Item item) {
    int key = item.hashCode();

    retry: while (true) {
        Node pred = this.head;
        Node curr = pred.next;
        while (curr.key <= key) {
            if (item == curr.item)
                break;
            pred = curr;
            curr = curr.next;
        } …
    }
    …
    Retry on synchronization conflict
```
public boolean remove(Item item) {
    int key = item.hashCode();
    retry: while (true) {
        Node pred = this.head;
        Node curr = pred.next;
        while (curr.key <= key) {
            if (item == curr.item)
                break;
            pred = curr;
            curr = curr.next;
        }
    }
}
public boolean remove(Item item) {
    int key = item.hashCode();
    retry: while (true) {
        Node pred = this.head;
        Node curr = pred.next;
        while (curr.key <= key) {
            if (item == curr.item)
                break;
            pred = curr;
            curr = curr.next;
        ...
    }
}
public boolean remove(Item item) {
    int key = item.hashCode();
    retry: while (true) {
        Node pred = this.head;
        Node curr = pred.next;
        while (curr.key <= key) {
            if (item == curr.item)
                break;
            pred = curr;
            curr = curr.next;
        }
        if (item == curr.item)
            break;
        pred = curr;
        curr = curr.next;
    }
    Stop if we find item
}
public boolean remove(Item item) {
    int key = item.hashCode();
    retry: while (true) {
        Node pred = this.head;
        Node curr = pred.next;
        while (curr.key <= key) {
            if (item == curr.item)
                break;
            pred = curr;
            curr = curr.next;
        } ...
    }
On Exit from 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
try {
    pred.lock(); curr.lock.lock();
    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 Method

```java
try {
    pred.lock(); curr.lock.lock();
    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
try {
    pred.lock(); curr.lock();
    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();
}
try {
    pred.lock(); curr.lock.lock();
    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();
}
try {
    pred.lock(); curr.lock.lock();
    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();
}
try {
    pred.lock(); curr.lock();
    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
  – Targets of `add()`, `remove()`, `contains()`
  – No contention on traversals

• Moreover
  – Traversals are wait-free
  – Food for thought …
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
  – 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 \texttt{pred} is not marked
- Check that \texttt{curr} is not marked
- Check that \texttt{pred} points to \texttt{curr}
Business as Usual
Business as Usual

Diagram showing a flow from 'a' to 'b' to 'c'.
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

![Diagram showing logical delete process]

Logical delete
Business as Usual

physical delete
Business as Usual
New Abstraction Map

- $S(\text{head}) =$
  - $\{ x \mid \text{there exists node } a \text{ such that}$
    - $a$ reachable from head and
    - $a.\text{item} = x$ and
    - $a$ is unmarked
  - $\}$
Invariant

- If not marked, then item in the set
- and reachable from head
- and if not yet traversed, it is reachable from pred
private boolean validate(Node pred, Node curr) {
    return !pred.marked && !curr.marked && pred.next == curr);
}
List Validate Method

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

```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;
        }
    } 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.lock();
    if (validate(pred, curr) {
        if (curr.key == key) {
            // Logical remove
            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;
        }
    } finally {
        pred.unlock();
        curr.unlock();
    }
}
```

physical remove
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;
}
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;
}
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 + list ordering
1. Not marked $\rightarrow$ in the set
2. Marked or missing $\rightarrow$ not in the set
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
Lock-free Lists

• Next logical step
  – Wait-free contains()
  – lock-free add() and remove()

• Use only compareAndSet()
  – What could go wrong?
Lock-free Lists

Use CAS to verify pointer is correct

Not enough!
Problem...

Logical Removal

Physical Removal

Node added
The Solution: Combine Bit and Pointer

Mark-Bit and Pointer are CASed together (AtomicMarkableReference)

Logical Removal = Set Mark Bit

Physical Removal CAS

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

![Diagram showing a Reference with an address and a mark bit F]
Extracting Reference & Mark

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

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);
Changing State

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

remove b

failed

remove c
Removing a Node

remove b

remove c
Removing a Node

remove b

remove c
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)
The Window Class

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

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

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

Find returns window
Using the Find Method

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

Extract pred and curr
The Find Method

Windows window = find(item);

At some instant, the item may be:

- in the middle (pred, curr, succ)
- at the beginning or end (or ...)

Diagram:

- pred
- curr
- succ
The Find Method

Window window = find(item);

At some instant,

item not in list

pred curr = null 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.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;
        }
    }
}

She’s not there …
public boolean remove(T item) {
    Boolean snip = null;
    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;
        }
    }
}

Try to mark node as deleted
public boolean remove(T item) {
    while (true) {
        Window window = find(head, key);
        Node 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, false, false))
                { return true; }n
    }
}}
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;
    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 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])
}
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])
}
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;
        }
    }
}
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;
        }
    }

    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);
        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);
            ...  
            if (curr.key >= key)  
                return new Window(pred, curr);
            pred = curr;
            curr = succ;
        }
    }
}
Lock-free Find

```java
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

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
• 16-node machine
• Vary percentage of \texttt{contains()} calls
High Contains Ratio

Ops/sec (90% reads/0 load)

Lock-free
Lazy list

Coarse Grained
Fine Lock-coupling
Low Contains Ratio

Ops/sec (50% reads/0 load)

- Lock-free
- Lazy list
- Coarse Grained
- Fine Lock-coupling

threads
As Contains Ratio Increases

Ops/sec (32 threads/0 load)

% Contains()
Summary

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

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

• 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.