Linked Lists and Synchronization Patterns
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 …
• 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
  – 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)
Invariants

- Sentinel nodes
  - tail reachable from head

- Sorted

- No duplicates
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

- 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

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

remove(b)
Removing a Node

remove(b)
Removing a Node

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

\texttt{remove(b)}
Removing a Node

Why hold 2 locks?

remove(b)
Concurrent Removes

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

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

remove(b)

remove(c)
Concurrent Removes

remove(b)
remove(c)
Concurrent Removes

remove(b)

remove(c)
Concurrent Removes

remove(b)

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)
Uh, Oh

remove(b)

remove(c)
Uh, Oh

Bad news, c not removed
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

```
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 of b

remove(c)
Removing a Node

Cannot acquire lock of b

remove(c)
Removing a Node

Wait!

remove(c)
Removing a Node

Proceed to remove(b)
Removing a Node

remove(b)
Removing a Node

remove(b)
Removing a Node

```python
remove(b)
```
Removing a Node
Remove method

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

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

Key used to order node
public boolean remove(Item item) {
    int key = item.hashCode();
    Node pred, curr;

    try {
        ... 
    } finally {
        currNode.unlock();
        predNode.unlock();
    }
}
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 { ... }
```
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 { ... }
```

Traversing list
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 = curr;
    curr = curr.next;
    curr.lock();
} return false;

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

```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;
```

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;

If node found, remove it
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 = curr;
    curr = curr.next;
    curr.lock();
}
return false;
Remove: searching

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

Otherwise, not present
Why 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**
• So **curr.item** is in the set
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 if item is present
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

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

Linearization point

Linearization point
Adding Nodes

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

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

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')
What Else Could Go Wrong?

add(c)

add(b')
What Else Could Go Wrong?

\[\text{add}(c)\]
What Else Could Go Wrong?

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

Yes, b still points to d
Optimistic: Linearization Point

`add(c)`
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!
Yes, b 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
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
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
  – \texttt{contains}(x) never locks …

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

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

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

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

Diagram showing a sequence of nodes labeled 'a', 'b', 'c', and 'd' connected in a linear fashion.
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
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
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?
Remove Using CAS

Logical Removal = Set Mark Bit

Use CAS to verify pointer is correct

Not enough!
Logical Removal =
Set Mark Bit

Problem:
d not added to list…
Must Prevent
manipulation of
removed node’s pointer
The Solution: Combine Bit and Pointer

Logical Removal =
Set Mark Bit

Mark-Bit and Pointer are CASed together
(AtomicMarkableReference)

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`
Extracting Reference & Mark

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

`Public Object get(boolean[] marked);`

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

public boolean isMarked();

Value of mark
Changing State

```java
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

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

public boolean attemptMark(
    Object expectedRef,
    boolean updateMark);

If this is the current reference …
Changing State

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

.. then change to this new mark.
Removing a Node

a → CAS → c → d

CAS
Removing a Node

remove b

failed

remove c
Removing a Node
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

```java
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

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, or ...

pred curr succ
The Find Method

Window window = find(item);

At some instant,

pred curr = null

item

not in list

succ
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])
}
Performance

On 16 node shared memory machine
Benchmark throughput of Java List-based Set algs. Vary % of Contains() method Calls.
High Contains Ratio

Ops/sec (90% reads/0 load)

Lock-free
Lazy list
Course Grained
Fine Lock-coupling
Low Contains Ratio

Ops/sec (50% reads/0 load)

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

threads
As Contains Ratio Increases

Ops/sec (32 threads/0 load)

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

• Coarse-grained locking
• Fine-grained locking
• Optimistic synchronization
• Lock-free synchronization