Locking Strategies: Optimistic, Lazy

CS 475, Fall 2019
Concurrent & Distributed Systems

With material from Herlihy & Shavit, Art of Multiprocessor Programming
Sequential Consistency vs Linearizability

• Linearizability can be composed:
  • If p’s execution and q’s execution are both linearizable, then the combination must also be linearizable

• Sequential consistency can not be composed:
  • If p’s execution and q’s execution are both sequential, then the combination MAY also be sequential (but not guaranteed!)

• Why use sequential consistency?
  • Does not require global clock
The List-Based Set

Sorted with Sentinel nodes
(min & max possible keys)
Course Grained Locking

Simple but hotspot + bottleneck
Coarse-Grained Locking

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

• Simple, clearly correct
  – Deserves respect!

• Works poorly with contention
  – Queue locks help
  – But bottleneck still an issue
Fine-grained Locking

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

• Split object into pieces
  - Each piece has own lock
  - Methods that work on disjoint pieces need not exclude each other
Simple Fine-Grained Locking: Remove

```
remove(c)
remove(b)
```
Simple Fine-Grained Locking: Remove

remove(b)
remove(c)
Simple Fine-Grained Locking: Remove

- remove(b)
- remove(c)
Simple Fine-Grained Locking: Remove

- remove(b)
- remove(c)
Simple Fine-Grained Locking: Remove

remove(b)  
remove(c)
Simple Fine-Grained Locking: Remove

- remove(b)
- remove(c)
Simple Fine-Grained Locking: Remove

remove(b)

remove(c)
Uh, Oh

remove(b)
remove(c)
Uh, Oh

Bad news, C not removed

remove(b)

remove(c)
Hand-over-Hand locking
Hand-over-Hand locking
Hand-over-Hand locking
Hand-over-Hand locking
Hand-over-Hand locking
Dining Philosophers

• Give each chopstick a lock
• Is this enough?
• Could deadlock!
• Actual solutions:
  • Pick up one chopstick, wait for the other for N msec, otherwise put down what you have, wait, and try again
  • Only allow 4 philosophers to pick up chopsticks at once
  • Even # seats pick up right chopstick, odd # seats pick up left
Adding Nodes

• To add node e
  – Neither successor nor predecessor can be deleted while add is running
  – Simply locking predecessor will guarantee that
Simple Fine-Grained Locking: Add (Execution Order 1)

- add(b)
- remove(c)
Simple Fine-Grained Locking: Add (Execution Order 2)

add(b)
remove(c)
Today

- Adding threads should not lower throughput - should increase throughput
  - Not possible if inherently sequential
  - How do we structure locks for faster performance?
  - Case study on a data structure
- Reading: H&S 9.6-9.9
Optimistic Synchronization

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

add(c)
Optimistic: Lock and Load
What could go wrong?

add(c) → a

b → d

e

remove(b)

Aha!
Validate – Part 1 (while holding locks)

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

add(c)
What Else Can Go Wrong?

![Diagram showing a process with nodes labeled a, b, d, e, and an arrow from a to b, another from b to d, and a separate node with an arrow labeled \( \text{add}(c) \) and another labeled \( \text{add}(b') \).]
What Else Can Go Wrong?

add(c)

Aha!
Validate Part 2
(while holding locks)

Yes, b still points to d

add(c)
Optimistic: Linearization Point

```
add(c)
```

Diagram showing a sequence of transactions with linearization points.
Same Abstraction Map

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

remove(c) → a → b → d → e

Aha!
Validate (1)

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
Validation

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

```java
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;
}
```

Search range of keys
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 curry) {
    Node node = head;
    while (node.key <= pred.key) {
        if (node == pred)
            return pred.next == curr;
        node = node.next;
    }
    return false;
}
private boolean validate(Node pred, Node curr) {
    Node node = head;
    while (node.key <= pred.key) {
        if (node == pred) {
            return pred.next == curr;
        }
        node = node.next;
    }
    return false;
}
Validation

```java
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
Remove: searching

```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;
        }
    }
```
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;
        } ...
    }

    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;
        } ...
    }
}

Stop if we find item
Remove: searching

```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;
        }
    }
}
```
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
Remove Method

```java
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();
}
```
Remove Method

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

Always unlock
Remove Method

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

```java
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();
}
```

Check for synchronization conflicts
Remove Method

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

```java
try {
pred.lock(); curr.lock();
if (validate(pred, curr) {
    if (curr.item == item) {
        pred.next = curr.next;
        return true;
    } else {
        // target not found
        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 (it’s a set!)
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 pred is not marked
• Check that curr is not marked
• Check that pred points to curr
Business as Usual
Business as Usual
Business as Usual
Business as Usual

remove(b)
Business as Usual

![Diagram showing relationship between a, b, and c with note 'a not marked']
Business as Usual

a still points to b
Business as Usual

![Diagram showing logical delete process]
Business as Usual

physical delete
Business as Usual

Diagram showing connections between nodes labeled 'a', 'b', and 'c'.
New Abstraction Map

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

```java
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;
}

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

Validate as before
Remove

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

Traverse without locking
(nodes may have been removed)
Contains

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

Present and undeleted?
Summary: Wait-free Contains

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

• Next week: wait-free algorithms and data structures!
HW1 Discussion

Go to socrative.com and select “Student Login” Room: CS475; ID is your G-Number

1. How fair do you think this assignment was?
2. How difficult did you think this assignment was?
3. How long did you spend on this assignment?

Reminder: If you are not in class, you may not complete the activity. If you do anyway, this will constitute a violation of the honor code.
HW 2 Discussion

This work is licensed under a Creative Commons Attribution-ShareAlike license

- This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/4.0/
- You are free to:
  - Share — copy and redistribute the material in any medium or format
  - Adapt — remix, transform, and build upon the material
  - for any purpose, even commercially.
- Under the following terms:
  - Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
  - ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.
  - No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.