Locking - Optimistic and Lazy

CS 475, Spring 2019
Concurrent & Distributed Systems

With material from Herlihy & Shavit, Art of Multiprocessor Programming
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

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

\[ \text{remove(c)} \]
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)
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
Dining Philosophers

- N philosophers seated around a circular table
- One chopstick between each philosopher (N chopsticks)
- A philosopher picks up both chopsticks next to him to eat
- Philosophers may not pick up both chopsticks at the same time
- How do they all eat without deadlocking or starving?
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
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

1,450 grains of rice/sec

5,431,616 grains of rice/sec

12,450,856 grains of rice/sec
Today

• Adding threads should not lower throughput
  • Contention effects
• Should increase throughput
  • Not possible if inherently sequential
  • How do we structure locks for faster performance?
• Reading: H&S 9.6-9.9
HW1 Discussion

HW1 Grades, as of Sat Feb 16 16:06:38 2019

Grade (Out of 100%)

0 20 40 60 80 100
HW1 Discussion

HW1 Submissions per day, as of Sat Feb 16 - Total = 1,194
Drawbacks of Fine Grained Locking for List

- Better than coarse-grained lock
  - Threads can traverse in parallel
- Still not ideal
  - Long chain of acquire/release
  - Inefficient (still can have blocking; two threads traversing list at once are sequential)
Optimistic Synchronization

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

![Diagram showing a sequence of nodes with an event 'add(c)' and an insight 'Aha!' connected by arrows.]
Optimistic: Lock and Load

add(c)
What could go wrong?

```
add(c)
```

```
remove(b)
```

```
Aha!
```
Validate – Part 1
(while holding locks)

add(c)

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

add(c)
What Else Can Go Wrong?

add(c)

add(b')

b'

32
What Else Can Go Wrong?

add(c)

Aha!
Validate Part 2
(while holding locks)

Yes, b still points to d
Optimistic: Linearization Point
Same Abstraction Map

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

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

• If
  – Nodes b and c both locked
  – Node b still accessible
  – Node c still successor to b
• Then
  – Neither will be deleted
  – OK to delete and return true
Unsuccessful Remove

remove(c)

Aha!
Validate (1)

Yes, b still reachable from head

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

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

Predecessor not reachable

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

Examine predecessor and current nodes
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
    }
}
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

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();
    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();
}
```

Lock both nodes
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();
}
```

Check for synchronization conflicts
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;
        }
    } else {
        return false;
    }
} finally {
    pred.unlock();
    curr.unlock();
}
```

target not found
Optimistic List

• Limited hot-spots
  – All methods follow same scheme: traverse (no locks), then lock and validate
  – 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 pred is not marked
- Check that curr is not marked
- Check that pred points to curr
Business as Usual

contains(b)
Business as Usual

contains(b)
Business as Usual

contains(b)
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

physical delete
Business as Usual

Returns false, because b is marked. No need to validate or lock or re-traverse

Intuition: Now can judge if b is in list ONLY by looking at b, don’t also need to look at a
New Abstraction Map

• $S(\text{head}) =$
  - $\{ x \mid$ there exists node $a$ such that
    • $a$ reachable from $\text{head}$ and
    • $a$.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);
}
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;
}}
} 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();
if (validate(pred, curr) {
if (curr.key == key) {
    curr.marked = true;
pred.next = curr.next;
    return true;
} else {
    return false;
}}
```
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;
        }
    } else {
        return false;
    }
} finally {
    pred.unlock();
    curr.unlock();
}

Logical remove
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;
}
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;
}
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 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
• Eliminate locking entirely
• `contains()` wait-free and `add()` and `remove()` lock-free
• Use only `compareAndSet()`
• What could go wrong?
• The textbook works through this
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
As Contains Ratio Increases

![Graph showing Ops/sec (32 threads/0 load) vs % Contains() for Lock-free, Lazy list, Course Grained, Fine Lock-coupling.]
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
  – 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
“To Lock or Not to Lock”

- Locking vs. Non-blocking: **Extremist views on both sides**
- The answer: **nobler to compromise, combine locking and non-blocking**
  - Example: Lazy list combines blocking add() and remove() and a wait-free contains()
  - Remember: Blocking/non-blocking is a property of a method
- Note that compromise makes reasoning even harder...
Plan

• More concurrent data structures in the book if you are interested (Ch 10-15)
• Weds: How to distribute work?
• Mon: Barrier synchronization
• Then we’re off to networks and distributed systems land!
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.