Locking Strategies: Coarse & Fine

CS 475, Fall 2019
Concurrent & Distributed Systems
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 Flag Example

• Is this behavior really so wrong?
  – We can argue either way …
Volatile Keyword

CPU 1
- thread0()
- CPU 1 Cache
- Main Memory

CPU 2
- thread1()
- CPU 2 Cache
- Main Memory

7ns
100ns

GMU CS 475 Fall 2019
Conditions

• When a thread is waiting for something to happen, it might want to release the lock and be notified when that thing has happened
• Ex: while queue is full, release the lock to let someone else empty it
• This is what a condition does
• Key methods: await, signal

```java
Condition condition = lock.newCondition();
lock.lock();
try{
    while(!property)
        condition.await();
} catch(InterruptedException e){
    //Application-dependent response, property may still be false
}
//at this point, property must be true and we have the lock again

condition.signal(); //wake up one thread await'ing
condition.signalAll(); //wake up all threads await'ing
```
Today

• Adding threads should not lower throughput - should increase throughput
  • Not possible if inherently sequential
  • How do we structure locks for faster performance?
  • First: wrap up discussion of read/write locks
  • Then: case study on a data structure
• Reading: H&S 9.1-9.5
• Note: HW1 coming up: https://www.jonbell.net/gmu-cs-475-fall-2019/homework-1/
Readers and Writers Problem

• Commonly called a read-write lock: data can be read by an unlimited number of threads at a time, written by at most one
• If a thread wants to write, nobody else can be reading (Alice + Bob with the billboards)
• High level approach:
  • Build on top of mutual exclusion locks with condition variables
Readers and Writers Lock

class ReadLock implements Lock{
    public void lock() {
        lock.lock();
        try{
            while(writer){
                condition.await();
            }
            readers++;
        } finally {
            lock.unlock();
        }
    }

    @Override
    public void unlock() {
        lock.unlock();
        try{
            readers--;
            if(readers == 0)
                condition.signalAll();
        } finally{
            condition.signalAll();
        }
    }
}

class WriteLock implements Lock{
    public void lock() {
        lock.lock();
        try{
            while(readers > 0 || writer)
                condition.await();
            writer = true;
        } finally{
            lock.unlock();
        }
    }

    @Override
    public void unlock() {
        writer = false;
        condition.signalAll();
    }
}
Coarse-Grained Synchronization

• Each method locks the object
  – Avoid contention using queue locks
  – Easy to reason about
    • In simple cases
  – Standard Java model
    • Locks, Synchronized blocks and methods

• So, are we done?
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 …
Today:
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
Linked List

- Illustrate this 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

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 contains(T x);
    public boolean remove(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;
}
public class Node {
    public T item;
    public int key;
    public Node next;
}
public class Node {
    public T item;
    public int key;
    public Node next;
}
The List-Based Set

Sorted with Sentinel nodes
(min & max possible keys)
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()`

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

- Invariants make sense only if
  - methods considered are the only way to modify the variables
- Language encapsulation helps
  - List nodes not visible outside class (private)
- Freedom from interference needed even for removed nodes
  - Some algorithms traverse removed nodes
  - Careful with `malloc()` & `free()`!
- Garbage-collection helps here
Abstract Data Types

• Our choice for how we store something concretely doesn’t need to exactly match the abstract type that we expose

• Concrete representation

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

- Meaning of representation given by abstraction map

\[ S(\text{Diagram}) = \{a, b\} \]
Representation Invariants

• Which abstract values are valid?
  – Is the set sorted?
  – Are there duplicates allowed in the set?

• Representation invariant
  – Characterizes legal concrete representation
  – Preserved by methods
  – Relied on by methods
• Rep invariant is a **contract**
• Suppose
  - **add()** leaves behind 2 copies of \( x \) in the concrete (list) type
  - **remove()** removes only 1 from the concrete (list) type
• Which one is incorrect?
Blame Game

• Suppose
  - \texttt{add()} leaves behind 2 copies of }x\text{ in the concrete (list) type
  - \texttt{remove()} removes only 1 from the concrete (list) type

• Which one is incorrect?
  - If rep invariant says no duplicates
    • \texttt{add()} is incorrect
  - Otherwise
    • \texttt{remove()} is incorrect
Representation Invariants (partly)

- For 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 $= x$
  - $\}$
  - (The set made from the head node in the list is the set of all x’s such that the node holding x is reachable from the head)
Sequential List Based Set

Add()

Remove()
Sequential List Based Set

Add()

Remove()
Coarse Grained Locking

```java
public class CoarseLockedSet<T> {
    public synchronized boolean add(T x){ … }
    public synchronized boolean remove(T x){ … }
    public synchronized boolean contains(T x){ … }
}
```
Course Grained Locking

![Diagram of course grained locking]
Course Grained Locking
Course Grained Locking

Simple but hotspot + bottleneck
Coarse-Grained Locking

• Easy, same as the bigLock
  – “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(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)
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
Removing a Node

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

![Diagram showing the removal of node b from a linked list]
Removing a Node

removing a node

remove(b)
Removing a Node

![Diagram showing a linked list with a node marked for removal. The node to be removed is shown as 'b'.]
Removing a Node

![Diagram showing the removal of node 'b' from a linked list.]

- The node 'b' is removed from the list.
- The adjacent nodes 'a' and 'c' are linked directly, bypassing 'b'.
- The code snippet to remove node 'b': `remove(b)`.
Removing a Node

Why do we need to always hold 2 locks?
Concurrent Removes

remove(b)

remove(c)
Concurrent Removes

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

![Diagram showing concurrent removes b and c from a linked list](image-url)
Concurrent Removes

remove(b)

remove(c)
Concurrent Removes

remove(b)

remove(c)
Concurrent Removes

- remove(b)
- remove(c)
- Update A's next to C
- Update B's next to D
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

\textbf{remove}(b)
Removing a Node

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

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

- 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

![Diagram of removing a node](image-url)
Removing a Node

![Diagram showing the removal of a node from a linked list. The node 'a' is to be removed, and the links are adjusted accordingly.]

GMU CS 475 Fall 2019
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();
    }
}

Key used to order node
public boolean remove(Item item) {
    int key = item.hashCode();
    Node pred, curr;
    try {
        ...
    } finally {
        currNode.unlock();
        predNode.unlock();
    }
}

Predecessor and current nodes
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();
    }
}
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 { ... }
```
try {
    pred = this.head;
    pred.lock();
    curr = pred.next;
    curr.lock();
    ...
} finally { ... }

Remove method

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

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;
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;

Unlock predecessor
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 = curr.node;
    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;
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;
```

Otherwise, not present
Why does this work?

• To remove node e
  – Must lock e
  – Must lock e’s predecessor
• Therefore, if you lock a node
  – It can’t be removed
  – And neither can its successor
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;
```

- 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
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;

Node locked, so no other thread can remove it ....
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;
```

Item not 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;
```

- `pred` reachable from `head`
- `curr` is `pred.next`
- `pred.key < key`
- `key < curr.key`
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;
}
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

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

- Easy to check that
  - tail always reachable from head
  - Nodes sorted, no duplicates
Why doesn’t it deadlock?

- Deadlock occurs with at least 2 threads, where each thread wants the lock that the other has.
- HOWEVER: each thread always acquires locks in same order (left to right).
- Hence both can’t be waiting on each other.

![Diagram showing lock acquisition orders]

Must acquire lock of b
remove(c)
Drawbacks

• Better than coarse-grained lock
  – Threads can traverse in parallel

• Still not ideal
  – Long chain of acquire/release
  – Inefficient
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
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.