Discussion: HW1 Part 4

```
kvstore.get(key1)
addToList(key1, newValue)
addToList(key2, newValue2)
kvstore.get(key1)
list.add(newValue2)
list.add(newValue)
```
@Test
public void testAddOrDeleteFromDifferentListsShouldBeConcurrent() throws InterruptedException {
    String lk1 = "listKey1";
    String lk2 = "listKey2";
    final ExecutorService executor = Executors.newFixedThreadPool(2);
    // Requirement here is that we MUST see overlap
    kvStore.set(lk1, new ArrayList<String>() {
        @Override
        public boolean add(String s) {
            Thread otherThread = new Thread(() -> {
                kvStore.addToList(lk2, s);
            });
            otherThread.start();
            try {
                otherThread.join(5000);
            } catch (InterruptedException ex) {
            }
            if (otherThread.isAlive())
                testFailed = true;
            return super.add(s);
        }
    });
    kvStore.set(lk2, new ArrayList<String>());
    kvStore.addToList(lk1, "obj" + System.currentTimeMillis);
    if (testFailed)
        Assert.fail("Expected to be able to call addToList on different lists simultaneously");
}
Review: Readers and Writers Lock

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

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

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();
    }
}
Review: Basic Spin-Lock

...lock suffers from contention

Contention -> ???
Review: Test-and-set Lock

class TASlock {
    AtomicBoolean state =
    new AtomicBoolean(false);

    void lock() {
        while (state.getAndSet(true)) {}
    }

    void unlock() {
        state.set(false);
    }
}
Review: Mystery #1

Adding MORE threads makes it SLOWER!
Review: End of the Day ...

Reading OK, no writing
Review: Test-and-test-and-set

- Wait until lock “looks” free
  - Spin on local cache
  - No bus use while lock busy
- Problem: when lock is released
  - Invalidation storm …
Review: Mystery #2

- TAS lock
- TTAS lock
- Ideal
Review: Spin-Waiting Overhead

The diagram illustrates the relative performance of different lock mechanisms over time and number of threads. The curves represent:

- **TAS lock**: This lock mechanism shows a higher overhead as the number of threads increases, making it less ideal for high-concurrency scenarios.

- **Backoff lock**: This mechanism exhibits a more gradual increase in overhead with an increasing number of threads, making it more suitable for situations with moderate concurrency.

- **TTAS Lock**: Demonstrates the least overhead among the three, making it the most ideal for high-threaded applications.

The x-axis represents the number of threads, while the y-axis indicates time. The ideal scenario is where the response time remains constant regardless of the number of threads.
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 9.1-9.5
- Note: HW2 posted
Coarse-Grained Synchronization

• Each method locks the object
  – Avoid contention using queue locks
  – Easy to reason about
    • In simple cases
    – Standard Java model
      • 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
  - \texttt{add(x)} put $x$ in set
  - \texttt{remove(x)} take $x$ out of set
  - \texttt{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);
}

Add item to set
List-Based Sets

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);
}
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;
}
```

item of interest
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

  - \( S(\text{a} \rightarrow \text{b} \rightarrow \text{c}) = \{\text{a, b}\} \)
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
Blame Game

• 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
  - `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?
  - If rep invariant says no duplicates
    - `add()` is incorrect
  - Otherwise
    - `remove()` is incorrect
Representation Invariants (partly)

- For sentinel nodes:
  - tail reachable from head
- Sorted
- No duplicates
Abstraction Map

• S(head) =
  – \{ x | \text{there exists a such that}
    • a \text{ 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

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
Course Grained Locking
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(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

The diagram shows a sequence of operations on a linked list. The nodes are labeled with `a`, `c`, and `d`, and the operations `remove(b)` and `remove(c)` are indicated. The sequence of operations results in the removal of nodes `b` and `c`, leaving the remaining nodes connected as `a` and `c` to `d`. 
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

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

![Diagram of a linked list with a node to be removed highlighted. The node to be removed is labeled as 'b'.]
Removing a Node

remove(b)
Removing a Node

remove(b)
Removing a Node

remove(b)
Removing a Node

Why do we need to always hold 2 locks?

remove(b)
Concurrent Removes

remove(b)
remove(c)
Concurrent Removes

remove(b)
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

```
Concurrent Removes

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

remove(b)

remove(c)
Uh, Oh

Bad news, C not removed

remove(b)
remove(c)
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

\[ \text{remove}(b) \]
Hand-Over-Hand Again

- remove(b)
- Found it!
Hand-Over-Hand Again

Found it!

remove(b)
Hand-Over-Hand Again

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

remove(b)

remove(c)
Removing a Node

- remove(b)
- remove(c)
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

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

```
        a
   ┌───────────────┐
   │                │
   │                │
   │                │
   │                │
   │                │
   └───────────────┘
        b
```

```
    ┌───────┐
    │  d   │
    │      │
    └───────┘
```

remove(b)
Removing a Node

```
remove(b)
```
Removing a Node
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();
    }
}
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
Try
pred = this.head;
pred.lock();
curr = pred.next;
curr.lock();

... finally { ... }

Lock current
Remove method

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
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;

Only one node locked!
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;
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

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;
    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

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