**CS 455: INTRODUCTION TO DISTRIBUTED SYSTEMS [THREAD SAFETY]**

Shrideep Pallickara  
Computer Science  
Colorado State University

February 16, 2017

---

**Topics covered in this lecture**

- Adding functionality to a thread-safe class
- Synchronized & Concurrent Collections
- Locking strategies
  - Lock striping
- Synchronizers

---

**A Vehicle Tracker application**

- Each vehicle
  - Identified by a String
  - Location represented by (x,y) coordinates

- VehicleTracker class
  - Tracks identity and location of all known vehicles

---

**Viewer thread and Updater Thread**

**Viewer**

```java
Map<String, Point> locations = vehicles.getLocations();
for (String key: locations.keySet())
  renderVehicle(key, locations.get(key));
```

**Updater**

```java
public void vehicleMoved(VehicleMovedEvent evt) {
  Point loc = evt.getNewLocation();
  vehicles.setLocation(evt.getVehicleId(), loc.x, loc.y);
}
```
public class MonitorVehicleTracker {
    private final Map<String, MutablePoint> locations;
    public synchronized Map<String, MutablePoint> getLocations() {
        return deepCopy(locations);
    }
    public synchronized MutablePoint getLocation(String id) {
        MutablePoint loc = locations.get(id);
        return loc == null ? null : new MutablePoint(loc);
    }
    public synchronized void setLocation(String id, int x, int y) {
        MutablePoint loc = locations.get(id);
        if (loc == null) {
            throw new IllegalArgumentException(...);
        }
        loc.x = x;
        loc.y = y;
    }
    private deepCopy() {
    }
}

public class BundleVehicleTracker {
    private final Map<String, BundlePoint> locations;
    public synchronized Map<String, BundlePoint> getLocations() {
        return deepCopy(locations);
    }
    public synchronized BundlePoint getLocation(String id) {
        BundlePoint loc = locations.get(id);
        return loc == null ? null : new BundlePoint(loc);
    }
    public synchronized void setLocation(String id, int x, int y) {
        BundlePoint loc = locations.get(id);
        if (loc == null) {
            throw new IllegalArgumentException(...);
        }
        loc.x = x;
        loc.y = y;
    }
    private deepCopy() {
    }
}

The tracker class is thread-safe, even though MutablePoint may not be thread-safe.

public class MonitorVehicleTracker {
    private final Map<String, MutablePoint> locations;
    public synchronized Map<String, MutablePoint> getLocations() {
        return deepCopy(locations);
    }
    public synchronized MutablePoint getLocation(String id) {
        MutablePoint loc = locations.get(id);
        return loc == null ? null : new MutablePoint(loc);
    }
    public synchronized void setLocation(String id, int x, int y) {
        MutablePoint loc = locations.get(id);
        if (loc == null) {
            throw new IllegalArgumentException(...);
        }
        loc.x = x;
        loc.y = y;
    }
    private deepCopy() {
    }
}

The Collections utility class

- List<String> readOnlyList = Collections.unmodifiableList(myList);
- Note:
  - Nothing to differentiate this as a read-only list
  - You have access to the mutator methods
    - But calling them results in an UnsupportedOperationException

public class ImmutablePoint {
    public final int x, y;
    public ImmutablePoint(int x, int y) {
        this.x = x;
        this.y = y;
    }
}

Delegating thread-safety

public class DelegatingVehicleTracker {
    private final ConcurrentMap<String, Point> locations;
    public DelegatingVehicleTracker(Map<String, Point> points) {
        locations = new ConcurrentHashMap<String, Point>(points);
        unmodifiableMap = Collections.unmodifiableMap(locations);
    }
    public Map<String, Point> getLocations() {
        return unmodifiableMap;
    }
    public Point getLocation(String id) {
        return locations.get(id);
    }
    public void setLocation(String id, int x, int y) {
        if (locations.replace(id, new Point(x, y)) == null) {
            throw new IllegalArgumentException("Invalid Vehicle ID");
        }
    }
}

Immutable Bundle

public class ImmutableBundle {
    public final int x, y;
    public ImmutableBundle(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
When delegation fails

```java
public class NumberRange {
    private final AtomicInteger lower = new AtomicInteger(0);
    private final AtomicInteger upper = new AtomicInteger(0);

    public void setLower(int i) {
        if (i > upper.get()) {
            throw IllegalArgumentException("lower > upper!");
        }
    }

    public void setUpper(int i) {
        if (i < lower.get()) {
            throw IllegalArgumentException("upper < lower!");
        }
    }

    public boolean isInRange(int i) {
        return (i >= lower.get() && i <= upper.get());
    }
}
```

Adding functionality to existing thread-safe classes

- Sometimes we have a thread-safe class that supports almost all the operations we need
- We should be able to add a new operation to it without undermining its thread safety

Adding a put-if-absent function to a List

- The operation put-if-absent must be atomic
- If List does not have X and we add X twice?
  - It's a problem because the collection should only have one X
- But if put-if-absent is not atomic?
  - Two threads could see that X is absent and the list then has 2 copies of X

Extending Vector to have a put-if-absent method

```java
public class BetterVector<E> extends Vector<E> {
    public synchronized boolean putIfAbsent(E x) {
        boolean absent = !contains(x);
        if (absent) {
            add(x);
        }
        return absent;
    }
}
```
Client side locking

- Sometimes extending a class or adding a method is not possible
- For e.g., if ArrayList is wrapped with a Collections.SynchronizedList wrapper
- Client code does not even know the class of the List object
- In such situations, the 3rd strategy of using a helper class comes in

```
public class ListHelper<E> {
    public List<E> list = Collections.synchronizedList(new ArrayList<E>());
    ...
    public synchronized boolean putIfAbsent(E x) {
        boolean absent = !list.contains(x);
        if (absent) {
            list.add(x);
        }
        return absent;
    }
    }
```

Contrasting extending a class AND client side locking

- Extending a class to add an atomic operation?
  - Distributes locking code over multiple classes in the object hierarchy
- Client side locking is even more fragile
  - We put locking code for a Class C in classes that are completely unrelated to it

Composition: A less fragile alternative to adding an atomic operation

```
public class ImprovedList<T> implements List<T> {
    private final List<T> list = new ArrayList<T>();
    ...
    public synchronized boolean putIfAbsent(T x) {
        boolean absent = !list.contains(x);
        if (absent) {
            list.add(x);
        }
        return absent;
    }
    public synchronized void clear() {list.clear();}
    // delegate other list methods ...
    }
```

More about the ImprovedList

- No worries even if the underlying List is not thread-safe
- ImprovedList uses its intrinsic lock
- Extra layer of synchronization may add small performance penalty
  - But it is much better than attempting to mimic the locking strategy of another object
Synchronized collections

- These include classes such as Vector and Hashtable
- There is also the synchronized wrapper classes
  - Created by Collections.synchronizedX factory methods
    - E.g., Collections.synchronizedList(List list), Collections.synchronizedMap(Map m), Collections.synchronizedSet(Set s)

Problems with synchronized collections

- Thread-safe but additional client-side locking needed to guard compound actions
  - Iteration
  - Navigation
  - Find the next element
  - Conditional operations
  - Put-if-absent

Compound actions producing confusing results

```
public Object getLast(Vector list) {
    int lastIndex = list.size() - 1;
    return list.get(lastIndex);
}
```

```
public void deleteLast(Vector list) {
    int lastIndex = list.size() - 1;
    list.remove(lastIndex);
}
```

Interleaving of getLast and deleteLast

A
size -> 10
remove(9)

B
size -> 10
get(9)
Boom!

Are there problems with this code?

```
for (int i=0; i < vector.size(); i++) {
    doSomething(vector.get(i));
}
```

There is chance that other threads may modify vector between the calls to size() and get()
Iterators

- The standard way to iterate over a Collection is with an iterator
- Using iterators does not mean that you don’t need to lock the collection
- Iterators returned by synchronized collections are not designed for concurrent modification

Iterators in synchronized collections

- Iterators of synchronized collections are fail-fast
- If they detect that the collection has changed since iteration began
  - Unchecked ConcurrentModificationException is thrown

Fail-fast iterators are not designed to be fool proof

- Designed to catch concurrency errors on a good-faith basis
- Associate a modification count with the collection
- If the modification count changes during iteration?
  - hasNext() or next() throws ConcurrentModificationException

Let’s look at this code snippet

List<Widget> widgetList = Collections.synchronizedList(new ArrayList<Widget>());
...
//May throw ConcurrentModificationException
for (Widget w : widgetList) {
  doSomething(w);
}

Internally javac generates code that uses iterator and repeatedly calls hasNext() and next() to iterate the List

How to prevent the ConcurrentModificationException

- Hold the collection lock for the duration of the iteration
- Is this desirable?
Issues with locking a collection during iteration

- Other threads that need to access the collection will block.
- If the collection is large or if the task performed on each element is lengthy, the wait could be really long.

Locking collection and scalability

- The longer a lock is held, the more likely it will be contended.
- If many threads are waiting for a lock, throughput and CPU utilization plummet.
- **ALTERNATIVE:**
  - Deep-copy the collection and iterate over the copy.
  - The copy is thread-confined.

Hidden Iterators

```java
public class HiddenIterator {
    private final Set<Integer> set = new HashSet<Integer>();
    public synchronized void add(Integer i) {set.add(i);}
    public synchronized void remove(Integer i) {set.remove(i);}
    public void diagnostics() {
        System.out.println("DEBUG: Elements in set: "+set);
    }
}
```

- Lock should have been acquired for the `System.out`
- Iterators are also invoked for `hashCode` and `equals`.

Locking strategies: 
Hashtable & ConcurrentHashMap

- **Hashtable**
  - Lock held for the **duration of each operation**
  - Restricting access to a single thread at a time

- **ConcurrentHashMap**
  - Finer-grained locking mechanism
  - Lock striping

**CONCURRENT COLLECTIONS**

Lock striping: How it works

- **ConcurrentHashMap uses an array of 16 locks**
  - Each lock guards 1/16\(^{th}\) of the hash buckets
  - Bucket \(N\) guarded by lock \(N \mod 16\)

- Assuming hash functions provide reasonable spreading characteristics
- Demand for a given lock should reduce by 1/16

- Enables ConcurrentHashMap to support up to 16 (default) concurrent writers
- A constructor that allows you to specify the concurrency level
**Downsides of lock striping**

- Locking the collection for exclusive access
  - More difficult and costly than a single lock
  - Done by acquiring locks in the stripe set
- When does ConcurrentHashMap need to do this?
  - If the map needs to be expanded, values need to be rehashed into a larger set of buckets

**Concurrent collections and iterators**

- Iterators are weakly consistent instead of fail-safe
  - Do not throw ConcurrentHashMapException
- Weakly consistent iterator
  - Tolerates concurrent modification
  - Traverses elements as they existed when the iterator was created
  - May (no guarantees) reflect modifications after construction

**But what are the trade-offs?**

- Semantics of methods that operate on the entire Map have been weakened to reflect nature of collection
  - size() is allowed to return an approximation
  - size() and isEmpty(): These are far less useful in concurrent environments
  - This allows performance improvements for the most important operations
    - get, put, containsKey, and remove

**One feature offered by synchronized Map implementations?**

- Lock the map for exclusive access
  - With Hashtable and synchronizedMap, acquiring the Map lock prevents other threads from accessing it
  - In most cases replacing Hashtable and synchronizedMap with ConcurrentHashMap?
    - Gives you getter scalability
    - If you need to lock Map for exclusive access?
      - Don't use the ConcurrentHashMap!

**Support for additional atomic Map operations**

- Put-if-absent
- Remove-if-equal
- Replace-if-equal

**ConcurrentMap interface**

```java
public interface ConcurrentMap<K,V> extends Map<K,V> {
    //Insert if no value is mapped from K
    V putIfAbsent(K key, V value);
    //Remove only if K is mapped to V
    boolean remove(K key, V value);
    //Replace value only if K is mapped to oldValue
    boolean replace(K key, V oldValue, V newValue);
    //Replace value only if K is mapped to a same value
    V replace(K key, V newValue)
}
```
Synchronizers

- Are objects that coordinate control flow of threads based on its state
- Examples
  - Latches
  - Semaphores
    - Counting and binary
  - Barriers
    - Cyclic and Exchangers

Synchronizer: Structural properties

- Encapsulate state that determines whether threads arriving at the synchronizer should:
  - Be allowed to pass or wait
- Provide methods to manipulate state
- Provide methods to wait for the synchronizer to enter desired state

Latches

- Latch acts as a gate
  - Until latch reaches terminal state; gate is closed and no threads can pass
  - In the terminal state; gate opens and allows all threads to pass
- Once the latch reaches terminal state?
  - Cannot change state again
  - Remains open forever

When to use latches

- Ensure that a computation does not proceed until all resources that it needs are initialized
- Service does not start until other services that it depends on have started
- Waiting until all parties in an activity are ready to proceed
- Multiplayer gaming

CountDownLatch

- Allows one or more threads to wait for a set of events to occur
- Latch state has a counter initialized to positive number
  - This is the number of events to wait for
  - countDown() decrements the counter indicating that an event has occurred
  - await() method waits for the counter to reach 0
public class TestHarness {
    public long timeTasks(int nThreads, final Runnable task)
        throws InterruptedException {
        final CountDownLatch startGate = new CountDownLatch(1);
        final CountDownLatch endGate = new CountDownLatch(nThreads);
        for (int i = 0; i < nThreads; i++) {
            Thread t = new Thread() {
                public void run() {
                    try {
                        startGate.await();
                        task.run();
                    } finally {
                        endGate.countDown();
                    }
                }
            };
            t.start();
        }
        long start = System.nanoTime();
        startGate.countDown();
        endGate.await();
        long end = System.nanoTime();
        return end - start;
    }
}

The contents of this slide set are based on the following references