Frequently asked questions from the previous class survey

- Are `collections.unmodifiableList` and its variants thread safe?
Topics covered in this lecture

- Concurrent collections
- Synchronizers
- Thread safety summary
- Distributed Servers
  - Performance
  - Amdahl’s Law
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*

Lock striping: How it works

- **ConcurrentHashMap** uses an array of 16 locks
  - Each lock guards 1/16th 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 ConcurrentModificationException

- 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 some 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;
    }
}

Semaphores

- Counting semaphores control the **number of activities** that can:
  - Access a certain resource
  - Perform a given action

- Used to implement resource pools or impose bounds on a collection
Semaphores

- Manage a set of virtual **permits**
  - Initial number passed to the constructor
- Activities **acquire** and **release** permits
- If **no permits** are available?
  - **acquire** blocks until one is available
- The **release** method returns a permit to the semaphore

Semaphores are useful for implementing resource pools

- Block if the pool is empty
  - Unblock if the pool is non-empty
- Initialize a semaphore to the **pool size**
- **acquire** a permit before trying to fetch a resource from pool
- **release** the permit after putting the resource back in pool
- **acquire** **blocks** until the pool is non-empty
Binary semaphores

- Semaphore with an **initial count of 1**
- Can be used as a **mutex** with non-reentrant locking semantics
  - Whoever holds the sole permit holds the mutex

```java
public BoundedHashSet<T> {  
    private final Set<T> set;
    private final Semaphore sem;
    public BoundedHashSet(int bound) {  
        this.set = Collections.synchronizedSet(new HashSet<T>());
        sem = new Semaphore(bound);
    }
    public boolean add(T o) throws InterruptedException {  
        sem.acquire();
        boolean wasAdded = false;
        try {  
            wasAdded = set.add(o);
            return wasAdded;
        } finally {  
            if (!wasAdded) sem.release();
        }
    }
    public boolean remove(Object o) {  
        boolean wasRemoved = set.remove(o);
        if (wasRemoved) sem.release();
        return wasRemoved;
    }
}
```
Barriers

- Barriers are similar to latches in that they block a group of threads till an event has occurred.
- All threads must come together at barrier point at the same time to proceed.
  - Latches wait for events, barriers wait for other threads.

Barriers and dinner ...

- Family rendezvous protocol
- Everyone meet at Panera @ 6:00 pm;
  - Once you get there, stay there ... till everyone shows up
  - Then we'll figure out what we do next
Barriers

- Often used in simulations where work to calculate one step can be done in parallel
  - But all work associated with a given step must complete before advancing to the next step
- All threads complete step $k$, before moving on to step $k+1$

CyclicBarrier

- Allows a fixed number of parties to rendezvous at a fixed point
- Useful in parallel iterative algorithms
  - Break problem into fixed number of independent subproblems
- Creation of a CyclicBarrier
  - Runnable cyclicBarrierAction = ... ;
  - CyclicBarrier cyclicBarrier =
    new CyclicBarrier(2, cyclicBarrierAction);
Using Cyclic Barriers

class Solver {
    final int N;     final CyclicBarrier barrier;
    class Worker implements Runnable {
        int myRow;
        Worker(int row) { myRow = row; }
        public void run() {
            while (!done()) {
                processRow(myRow);
                try {
                    barrier.await();
                } catch (BrokenBarrierException ex) {
                    ...
                }
            }
        }
    }
    public Solver(float[][] matrix) {
        data = matrix;     N = matrix.length;
        barrier = new CyclicBarrier(N, new Runnable() { public void run() {
            mergeRows(...);  } });
        for (int i = 0; i < N; ++i)
            new Thread(new Worker(i)).start(); //DO NOT START THREAD in constructor.
        waitUntilDone();
    }
}

Source: From the Java API

Exchanger

- Another type of barrier
- Two-party barrier
- Parties exchange data at the barrier point
- Useful when asymmetric activities are performed
  - Producer-consumer problem
- When 2 threads exchange objects via Exchanger
  - Safe publication of objects to other party
Thread Safety: Summary

- It’s all about *mutable, shared state*
  - The less mutable state there is, the easier it is to ensure thread-safety
- Make fields **final** unless they need to be mutable
- **Immutable** objects are automatically thread-safe
- **Encapsulation** makes it practical to manage complexity
Thread Safety: Summary

- Guard each mutable variable with a **lock**
- Guard **all variables in an invariant** with the **same lock**
- Hold locks for the **duration** of compound actions

---

Thread Safety: Summary

- Program that accesses mutable variables from multiple threads without synchronization?
  - Broken program

- Include thread-safety in the design process
  - Document if your class is not thread-safe

- Document your synchronization policy
Thread Safety: Summary

- Rather than scattering access to shared state throughout your programs and attempting ad hoc reasoning about interleaved access
  - Structure program to facilitate reasoning about concurrency
  - Use a set of standard synchronization primitives to control access to shared state
Measures of performance

- Service time
- Latency
- Throughput
- Capacity
- Efficiency
- Scalability

How fast?

How much?

Performance and Scalability

- Tuning for performance
  - Do same work with less effort
  - Caching, choice of algorithms $O(n^2)$ to $O(n \log n)$

- Scalability
  - Find ways to parallelize problem
  - Do more work with more resources

How fast?

How much?
HOW FAST and HOW MUCH

- Are separate and can (at times) be at odds with each other
- To scale or for better hardware utilization
  - We often end up increasing the amount of work for each task
  - Divide tasks into multiple pipelined tasks
    - Orchestration overhead

The quest for performance

- What do you mean by faster?
- Under what conditions?
  - Small or large datasets
  - Perform measurements to substantiate arguments
- How often do these conditions arise?
- What are the hidden costs?
  - Development/maintenance risks
  - Tradeoffs
  - Ripple effects of decision
Avoid premature optimizations

- First make it **right, then fast**
- **Measure**, don’t guess
- Quest for performance is one of the biggest source of **bugs**

---

**Amdahl's Law**
How much can we speed things up?

- Harvesting crops
  - The more the number of workers
  - The faster the crop can be harvested

- But some things are fundamentally serial
  - Adding additional workers does not make the crop grow faster

The right tool for the right job: Everything is not a nail

- Make sure that problem is amenable to parallel decomposition

- Most programs have a mix of parallelizable and serial portions
Amdahl’s law describes how much a program can be theoretically sped up

- \( F \): Fraction of components that must be executed serially
- \( N \): Number of available processors

\[
\text{Speedup} \leq \frac{1}{F + \frac{(1 - F)}{N}}
\]

\[
\text{Utilization} = \frac{\text{Speedup}}{N}
\]

As \( N \) approaches infinity; maximum speedup converges to \( 1/F \)

- With 50% serial code
  - Maximum speedup is 2

- With 10% serial code
  - Maximum speedup is 10
  - With \( N = 10 \)
    - Speedup = 5.3 at 53% utilization
  - With \( N = 100 \)
    - Speedup = 9.2 at 9% utilization
Speedups for different parallelization portions

![Amdahl's Law graph](http://en.wikipedia.org/wiki/Amdahl%27s_law)

Source: [en.wikipedia.org/wiki/Amdahl%27s_law](http://en.wikipedia.org/wiki/Amdahl%27s_law)

**Know what to speed up**

Two independent parts A B

Original process

Make B 5x faster

Make A 2x faster

![Diagram showing speedup](http://en.wikipedia.org/wiki/Amdahl%27s_law)
The contents of this slide-set are based on the following references

- https://www.javaspecialists.eu/archive/Issue192b.html