Northeastern University - Seattle

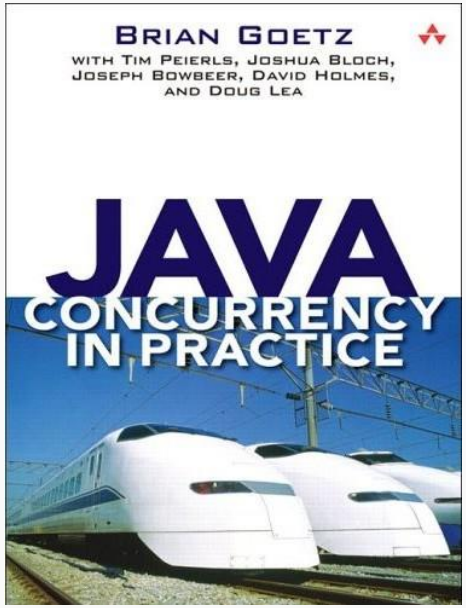
CS5510

Professor Ian Gorton

Week 10

CONCURRENCY II

http://jcip.net/



Overview

- Loops/recursion and threads
- Readers Writers and locking
- Scalability
- Testing

Recap from last week

- If threads share state, then you must use locks to achieve thread-safety
  - Synchronized
  - java.util.concurrent classes
- If threads need to coordinate to order actions
  - Guarded conditions (wait/notify)
  - Beware of deadlocks
- Use executors to manage threads

Loops/Recursion and Threads

- If we have a loop with completely independent iterations:
  - Can use a thread to execute each iteration
- Effects on performance?

```
void processSequentially(List<Element> elements) {
    for (Element e : elements)
        process(e);
}

void processInParallel(Executor exec, List<Element> elements) {
    for (final Element e : elements)
        exec.execute(new Runnable() {
            public void run() {
                process(e);
            }
        });
}
```

JCIP Listing 8.10

Loops/Recursion

- Parallelization of sequential loops works when:
  - Each iteration is completely independent of others
  - Work done in each iteration is enough to offset cost of thread management

Recursion

- Often independent sequential loops in recursive algos
- E.g. each iteration does not require results of recursive iterations it invokes
- Examples?

Recursion (JCIP Fig 8.11) Depth-First Tree Traversal

```
public <T> void sequentialRecursive(List<Node<T>> nodes,
                                    Collection<T> results) {
    for (Node<T> n : nodes) {
        results.add(n.compute());
        sequentialRecursive(n.getChildren(), results);
    }
}
```

Recursion

```
public <T> void parallelRecursive(final Executor exec,
                                  List<Node<T>> nodes,
                                  final Collection<T> results) {
    for (final Node<T> n : nodes) {
        exec.execute(new Runnable() {
            public void run() {
                results.add(n.compute());
            }
        });
        parallelRecursive(exec, n.getChildren(), results);
    }
}
```

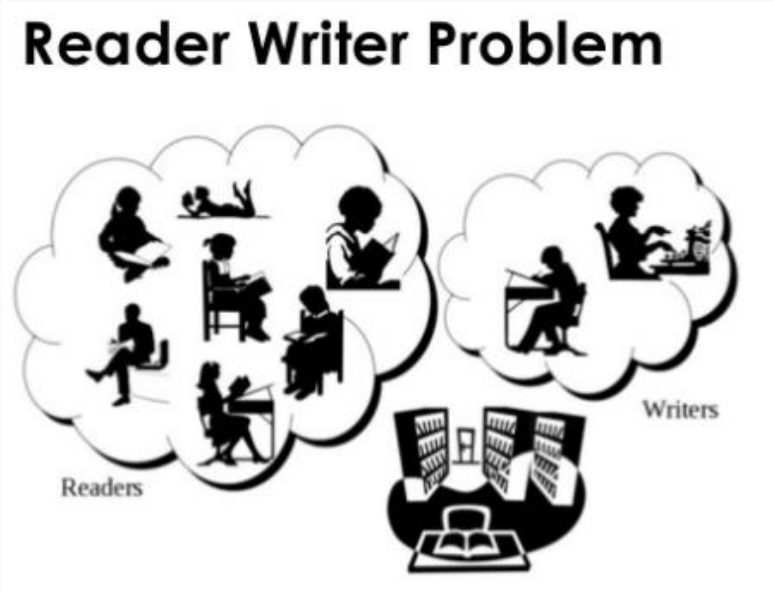
Recursion

```
public <T> Collection<T> getParallelResults(List<Node<T>> nodes)
        throws InterruptedException {

    ExecutorService exec = Executors.newCachedThreadPool();
    Queue<T> resultQueue = new ConcurrentLinkedQueue<T>();

    parallelRecursive(exec, nodes, resultQueue);

    exec.shutdown();
    exec.awaitTermination(Long.MAX_VALUE, TimeUnit.SECONDS);

    return resultQueue;
}
```

READERS WRITERS

Readers-Writers Problem

- Classic concurrency problem
- Multiple readers and writers to a shared database
- Many concurrent readers
- 1 writer
  - No other readers or writers
- Simple solution – writers wait until no readers



Readers Writers – Class Exercise

- Clone the repo at:
  - https://github.com/gortonator/ReadersWriters
- Spend a few minutes to understand the code
- Run with various numbers of readers and writers, eg:
  - 1 1
  - 5 2
  - 10 3
- What do you observe?

Readers Writers – Class Exercise

1. Modify the code to use Executors and Runnables
2. How would you change the code to give writers priority?

ReadWriteLock

- ReadWriteLock maintains a pair of associated locks,
- one for read-only operations
- one for writing
- The read lock may be held simultaneously by multiple reader threads
  - so long as there are no writers.
- The write lock is exclusive

Quick Aside - RentrantLock

- A reentrant mutual exclusion Lock
  - public class ReentrantLock extends Object, implements Lock, Serializable
- same basic semantics as implicit monitor lock using synchronized methods/statements
- A thread invoking lock will acquire the lock, when the lock is not owned by another thread.
- Lock returns immediately if the current thread already owns the lock.
- Some extended capabilities.
  - Fair/unfair acquisition policies,
  - isLocked, getLockQueueLength

Quick Aside - RentrantLock

```
class X {
  private final ReentrantLock lock = new ReentrantLock();
  // ...

  public void m() {
    lock.lock();  // block until condition holds
    try {
      // ... method body
    } finally {
      lock.unlock()
    }
  }
}
```

Thread Safe Dictionary.

```
public class Dictionary {

  private final ReentrantReadWriteLock readWriteLock =
            new ReentrantReadWriteLock();

  private final Lock read  = readWriteLock.readLock();

  private final Lock write = readWriteLock.writeLock();

  private HashMap<String, String> dictionary = new HashMap<String, String>();

  public void set(String key, String value) {
    write.lock();
    try {
      dictionary.put(key, value);
    } finally {
      write.unlock();
    }
  }
```

```
public String get(String key) {
  read.lock();
  try{
    return dictionary.get(key);
  } finally {
    read.unlock();
  }
}

public String[] getKeys(){
  read.lock();
  try{
    String keys[] = new String[dictionary.size()];
    return dictionary.keySet().toArray(keys);
  } finally {
    read.unlock();
  }
}
```

Class Exercise

- Modify the database from the previous example to use a ReadWriteLock

SCALABILITY

Scalability

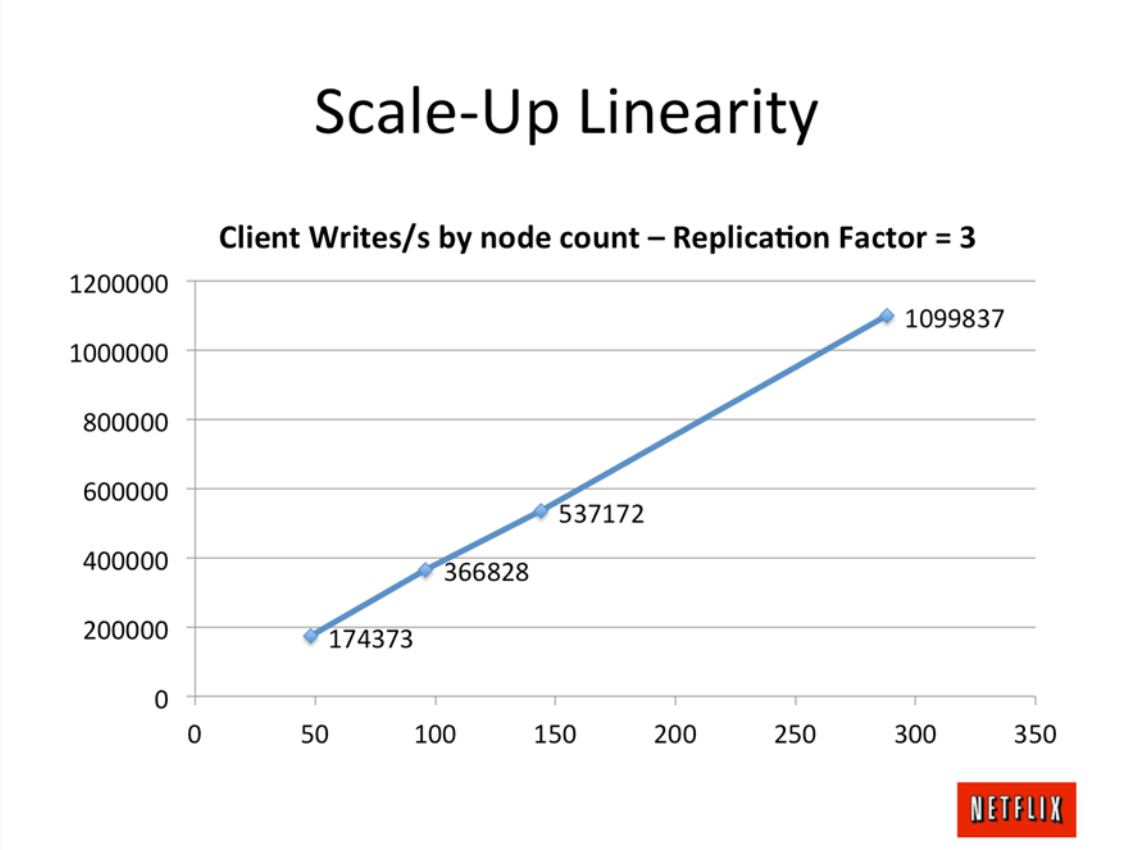
Scalability describes the ability to improve throughput or capacity when additional computing resources (such as additional CPUs, memory, storage, or I/O bandwidth) are added.

Engineering Concerns

- Threads make it possible to better utilize resources
- But also introduce overheads ….
  - Creation, context switching, management, coordination
- Some terminology …
- Service time/latency/response time
  - Measures of how fast a piece of work happens
- Capacity/throughput
  - How much work can be performed with a given quantity of computing resources

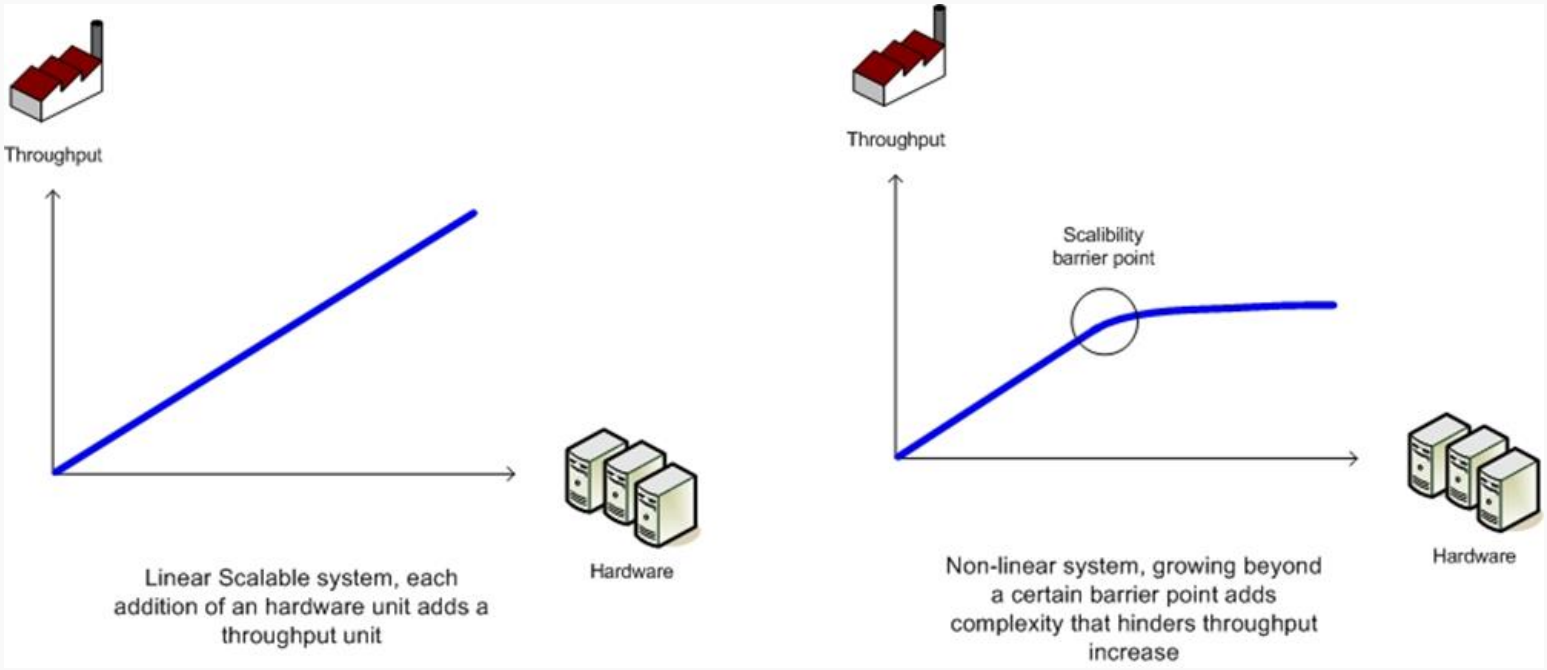
Amdahl's Law

The perfect world ….

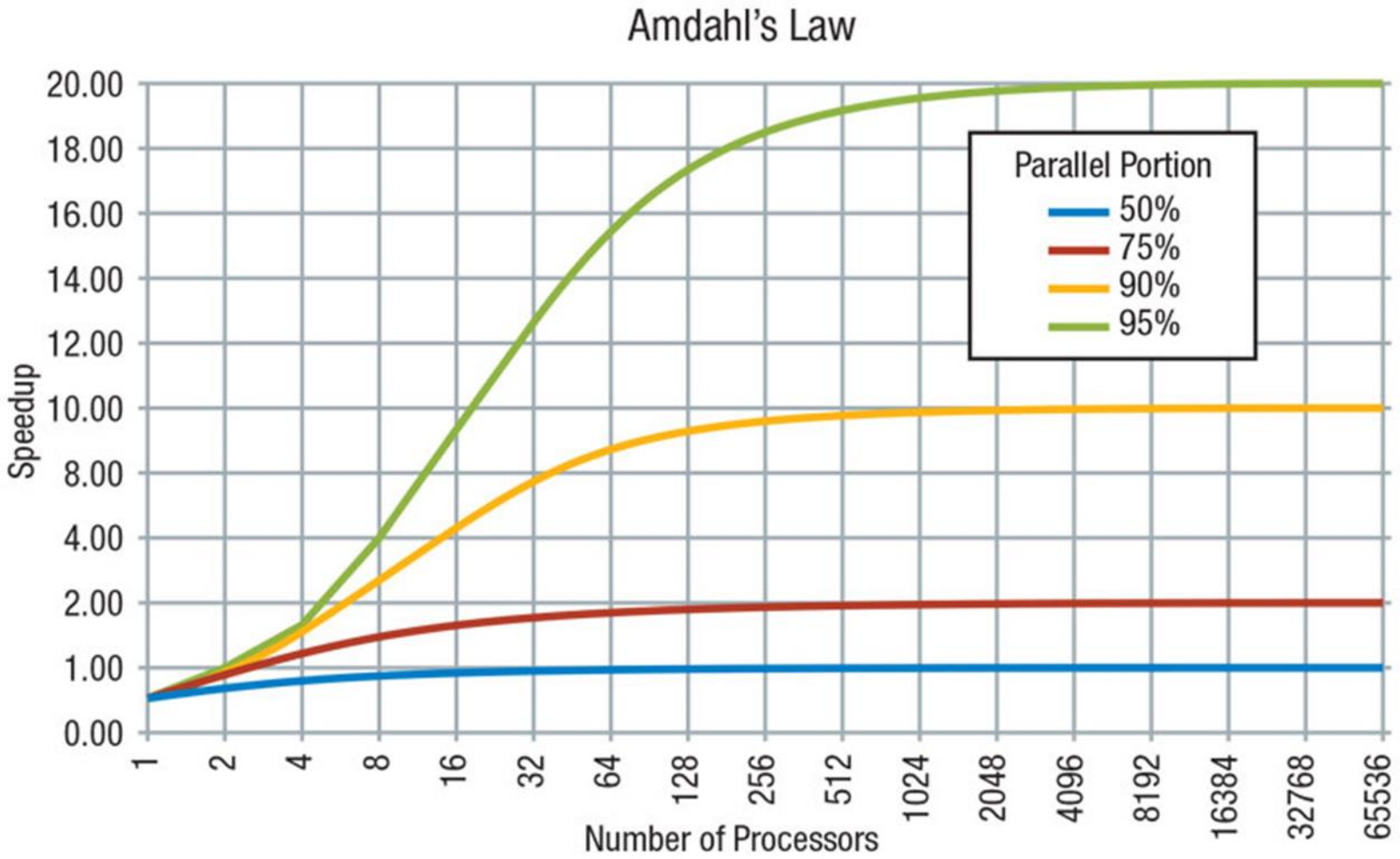


http://www.datastax.com/2012/01/choosing-the-right-architecture-for-big-data-scale

Amdahl's Law


Throughput
Hardware
Linear Scalable system, each addition of an hardware unit adds a throughput unit
Throughput
Scalibility barrier point
Hardware
Non-linear system, growing beyond a certain barrier point adds complexity that hinders throughput increase

Amdahl's Law


Amdahl's Law
Speedup
20.00
18.00
16.00
14.00
12.00
10.00
8.00
4.00
2.00
1.00
0.00
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
65536
Number of Processors
Parallel Portion
50%
75%
90%
95%

Amdahls's Law Example

Serialized Access to a task queue (jcip p141)

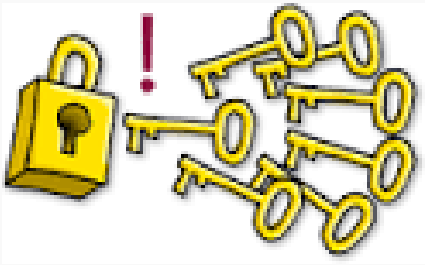
```
public class WorkerThread extends Thread {
    private final BlockingQueue<Runnable> queue;

    public WorkerThread(BlockingQueue<Runnable> queue) {
        this.queue = queue;
    }

    public void run() {
        while (true) {
            try {
                Runnable task = queue.take();
                task.run();    // results generated by thread stored somewhere?
            } catch (InterruptedException e) {
                break; /* Allow thread to exit */
            }
        }
    }
}
```

Thread Overheads

- Context switching has costs
  - Manipulates shared structures in OS/VM
  - For a new thread, local data not likely in cache so higher latencies due to cache misses
    - Cache pollution: newly running threads run slower until cache fills
  - Depends on CPU, but typically a microsecond or so
- Every time a thread blocks it gets switched
  - Blocking IO
  - Contended locks
  - Condition variables
- Frequently blocking threads reduce throughput

Reducing Lock Contention

- Serialization hurts scalability
- Context switching hurts performance
- Lock contention hurts both!!



How can we reduce lock contention?

Narrowing lock scope

- Busy (hot) locks limit scalability.
- Example:
  - Operation holds a lock for 2 mSecs
  - All threads must acquire this lock
- What is the ***maximum*** throughput we can attain?
  - No matter how many processors we have

Get In, Get Out (jcip p145)

```
public class AttributeStore {
     private final Map<String, String>
     attributes = new HashMap<String, String>();

    public synchronized boolean userLocationMatches(String name,
                                                    String regexp) {
        String key = "users." + name + ".location";  // construct key
        String location = attributes.get(key);          // search hashmap
        if (location == null)
            return false;
        else
            return Pattern.matches(regexp, location);  // process results
    }
}
```

Get in, Get out (jcip p145)

```
public class BetterAttributeStore {
        private final Map<String, String>
        attributes = new HashMap<String, String>();

    public boolean userLocationMatches(String name, String regexp) {
        String key = "users." + name + ".location";
        String location;
        synchronized (this) {
            location = attributes.get(key);
        }
        if (location == null)
            return false;
        else
            return Pattern.matches(regexp, location);
    }
}
```

Easier – use thread safe collections

```
public class BetterAttributeStore {
        private final ConcurrentHashMap<String, String>
        attributes = new ConcurrentHashMap<String, String>();

    public boolean userLocationMatches(String name, String regexp) {
        String key = "users." + name + ".location";
        String location;

        location = attributes.get(key);

        if (location == null)
            return false;
        else
            return Pattern.matches(regexp, location);
    }
}
```

Lock Splitting (jcip p146)

```
public class ServerStatusBeforeSplit {
        public final Set<String> users;
        public final Set<String> queries;
    public ServerStatusBeforeSplit() {
        users = new HashSet<String>();
        queries = new HashSet<String>();
    }
    public synchronized void addUser(String u) {
        users.add(u);
    }
    public synchronized void addQuery(String q) {
        queries.add(q);
    }
    public synchronized void removeUser(String u) {
        users.remove(u);
    }
    public synchronized void removeQuery(String q) {
        queries.remove(q);
    }
}
```

Lock Splitting (jcip p146)

```
public class ServerStatusAfterSplit {
     public final Set<String> users;
     public final Set<String> queries;

    public ServerStatusAfterSplit() {
        users = new HashSet<String>();
        queries = new HashSet<String>();
    }
    public void addUser(String u) {
        synchronized (users) {
            users.add(u);
        }
    }
    public void addQuery(String q) {
        synchronized (queries) {
            queries.add(q);
        }
    }
       // other refactored methods omitted
}
```

Lock Striping

- Basic approach is to partition a data structure and use a different lock for each partition
  - ConcurrentHashMap uses 16 locks
  - Each lock guards 1/16th of the hash buckets
- How many concurrent threads can be accessed in this example?
- What if we need to grow the hash map?

Lock Striping (jcip p148)

```
public class StripedMap {
    // Synchronization policy: buckets[n] guarded by locks[n%N_LOCKS]
    private static final int N_LOCKS = 16;
    private final Node[] buckets;
    private final Object[] locks;

    private static class Node { ….} // stuff missing

    public StripedMap(int numBuckets) {
        buckets = new Node[numBuckets];
        locks = new Object[N_LOCKS];
        for (int i = 0; i < N_LOCKS; i++)
            locks[i] = new Object();
    }
```

Lock Striping (jcip p148)

```
private final int hash(Object key) {
        return Math.abs(key.hashCode() % buckets.length);
    }
    public Object get(Object key) {
        int hash = hash(key);
        synchronized (locks[hash % N_LOCKS]) {
            for (Node m = buckets[hash]; m != null; m = m.next)
                if (m.key.equals(key))
                    return m.value;
        }
        return null;
    }
    public void clear() { // non atomic clear
        for (int i = 0; i < buckets.length; i++) {
            synchronized (locks[i % N_LOCKS]) {
                buckets[i] = null;
            }
        }
    } }
```

TESTING

Testing concurrent programs

- Tricky in the face of non-determinism
- Larger number of potential interactions and failure cases
- Test suites therefore have to be more extensive and run for longer

Testing concurrent programs

- Most tests can be classified as testing:
  - Safety
    - Nothing bad ever happens
    - Typically testing invariants hold
  - Liveness
    - Something good eventually happens
    - Trickier – eg testing for deadlocks
    - Also includes throughput, response times, scalability

Testing for Correctness

- Similar as for testing sequential code
- Identify post conditions and invariants
- Let's look at an example:
  - Testing a bounded buffer (JCIP section 12.1)

```
public class SemaphoreBoundedBuffer <E> {
    private final Semaphore availableItems, availableSpaces;
    private final E[] items;
    private int putPosition = 0, takePosition = 0;

    public SemaphoreBoundedBuffer(int capacity) {
        if (capacity <= 0)
            throw new IllegalArgumentException();
        availableItems = new Semaphore(0);
        availableSpaces = new Semaphore(capacity);
        items = (E[]) new Object[capacity];
    }

    public void put(E x) throws InterruptedException {
        availableSpaces.acquire();
        doInsert(x);
        availableItems.release();
    }

    public E take() throws InterruptedException {
        availableItems.acquire();
        E item = doExtract();
        availableSpaces.release();
        return item;
    }
```

```
 public boolean isEmpty() {
        return availableItems.availablePermits() == 0;
    }

    public boolean isFull() {
        return availableSpaces.availablePermits() == 0;
    }

private synchronized void doInsert(E x) {
        int i = putPosition;
        items[i] = x;
        putPosition = (++i == items.length) ? 0 : i;
    }

    private synchronized E doExtract() {
        int i = takePosition;
        E x = items[i];
        items[i] = null;
        takePosition = (++i == items.length) ? 0 : i;
        return x;
    }
```

Basic Unit Tests

- Test post-conditions and invariants, eg:
  - New buffer should identify itself as empty
  - New buffer should identify itself not full
  - Insert N elements into a buffer with capacity N and check it is full
  - Insert N elements into a buffer with capacity N and check it is not empty

Basic Unit Tests Examples

```
public class TestBoundedBuffer extends TestCase {

    void testIsEmptyWhenConstructed() {
        SemaphoreBoundedBuffer<Integer> bb = new SemaphoreBoundedBuffer<Integer>(10);
        assertTrue(bb.isEmpty());
        assertFalse(bb.isFull());
    }

    void testIsFullAfterPuts() throws InterruptedException {
        SemaphoreBoundedBuffer<Integer> bb = new SemaphoreBoundedBuffer<Integer>(10);
        for (int i = 0; i < 10; i++)
            bb.put(i);
        assertTrue(bb.isFull());
        assertFalse(bb.isEmpty());
    }
```

Testing Blocking Operations

- If an operation is meant to block, test only passes if thread does not proceed
- But for how long?
- Need to make an educated guess, e.g.:
  - private static final long LOCKUP_DETECT_TIMEOUT = 1000;
- Example:
  - Try to take an element from an empty buffer
  - Create a taker thread, wait, and interrupt it if still blocked
  - If take() succeeds, then test fails
  - If take() successfully interrupted and exists, then test succeeds

```
void testTakeBlocksWhenEmpty() {
        final SemaphoreBoundedBuffer<Integer> bb = new SemaphoreBoundedBuffer<Integer>(10);
        Thread taker = new Thread() {
            public void run() {
                try {
                    int unused = bb.take();
                    fail(); // if we get here, it's an error
                } catch (InterruptedException success) { // thread exits
                }
            }
        };
        try {
            taker.start();
            Thread.sleep(LOCKUP_DETECT_TIMEOUT);
            taker.interrupt();
            taker.join(LOCKUP_DETECT_TIMEOUT);
            assertFalse(taker.isAlive());  // verify join returned successfully as thread will be dead
        } catch (Exception unexpected) {
            fail();
        }
    }
```

Testing Safety

- Race conditions are trickier to test
- Tests need to be multi-threaded and can be complex
- Want tests to effect non-determinism as little as possible
  - Test code may obscure deadlocks in really evil cases
- **Ideally checking the test properties does not require any synchronization**
  - **And hence effects order of execution as minimally as possible**

Testing Bounded Buffer Example

- Check everything put into a queue/buffer comes out
  - And nothing else!
- Basic approach:
  - Each producer calculates a checksum for all the messages it produces
  - Each consumer calculates a checksum for all the messages it receives
  - When all producers/consumers complete, checksum are combined
    - If they are equal, test passes

Testing Bounded Buffer Example

- Should generate test data randomly
  - Minimize chances of tests accidentally passing
- Roll-your-own simple random number generator
  - As RNGs are thread-safe and hence effect thread synchronization
  - Each thread has own RNG so doesn't need to be thread-safe
  - Seeded with values based on time ensures different values every test
- We'll see this in the example soon …

Testing Bounded Buffer Example

- To introduce more randomness, coordinate starting and termination of threads
  - Ensure sequential thread operation doesn't introduce an element of determinism
  - Ensure testing of checksums is done after all threads finished
  - Use CyclicBarrier to coordinate start and end behavior
- Termination condition doesn't require thread interactions
  - Producers and consumers create and remove known numbers of items

Spend a few minutes looking at

http://jcip.net/listings/PutTakeTest.java

See if you can understand how it works?

```
public class PutTakeTest extends TestCase {
    protected static final ExecutorService pool = Executors.newCachedThreadPool();
    protected CyclicBarrier barrier;
    protected final SemaphoreBoundedBuffer<Integer> bb;
    protected final int nTrials, nPairs;
    protected final AtomicInteger putSum = new AtomicInteger(0);
    protected final AtomicInteger takeSum = new AtomicInteger(0);

    public static void main(String[] args) throws Exception {
        new PutTakeTest(10, 10, 100000).test(); // sample parameters
        pool.shutdown();
    }

    public PutTakeTest(int capacity, int npairs, int ntrials) {
        this.bb = new SemaphoreBoundedBuffer<Integer>(capacity);
        this.nTrials = ntrials;
        this.nPairs = npairs;
        this.barrier = new CyclicBarrier(npairs * 2 + 1); // initialize the barrier  +1 for main thread
    }
```

```
void test() {
        try {
            for (int i = 0; i < nPairs; i++) {   //  create the threads
                pool.execute(new Producer());
                pool.execute(new Consumer());
            }
            barrier.await(); // wait for all threads to be ready
            barrier.await(); // wait for all threads to finish
            assertEquals(putSum.get(), takeSum.get());
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
```

```
class Producer implements Runnable {
        public void run() {
            try {
                int seed = (this.hashCode() ^ (int) System.nanoTime());
                int sum = 0;
                barrier.await();
                for (int i = nTrials; i > 0; --i) {
                    bb.put(seed);
                    sum += seed;
                    seed = xorShift(seed);
                }
                putSum.getAndAdd(sum);
                barrier.await();
            } catch (Exception e) {
                throw new RuntimeException(e);
            }
        }
    }
```

```
class Consumer implements Runnable {
        public void run() {
            try {
                barrier.await();
                int sum = 0;
                for (int i = nTrials; i > 0; --i) {
                    sum += bb.take();
                }
                takeSum.getAndAdd(sum);
                barrier.await();
            } catch (Exception e) {
                throw new RuntimeException(e);
            }
        }
    }
```

Testing Concurrent Code

- Good example of the complexity of test cases can be higher than code complexity
- Other things to test:
- Resource management
  - See testLeak example in http://jcip.net/listings/TestBoundedBuffer.java
- Performance
  - Requires adding timing information and monitoring
- Scalability
  - Requires large volume tests and coordination of more resources
- These are central topics covered in Building Scalable Distributed Systems
  - ☺

Summary

- Threads can be used to parallelize independent loop iterations and independent recursive invocations.
- Readers-Writers is a classic concurrency problem
- Scalability requires careful design is always limited by serialization
- Testing concurrent systems is tricky due to large amount of failure modes and non-determinism


I ASK QUESTIONS.