Maps performance tips
  ▶ On server: Maintain connections, prepared statements (per thread/request!)
  ▶ On client: Minimize redraws (not on every mouse event!).
  ▶ On client: Let canvas do what it can.
  ▶ Java profiler is jvisualvm, consider YourKit.
  ▶ There is a JS profiler in the Chrome dev tools.

Open to Maps questions before class starts.
Design with Threads

John Jannotti

course/cs0320/www/docs/lectures/

April 6, 2017
A mental model for synchronized blocks

```java
synchronized (bank) {
    double balance = bank.get("John");
    balance += 231.23;
    bank.put("John", balance);
}
```

becomes (CAPITALS == built in “magic”)  

```java
LOCK lock = bank.GETLOCK();
try {
    lock.OBTAIN();
    double balance = bank.get("John");
    balance += 231.23;
    bank.put("John", balance);
} finally {
    lock.RELEASE();
}
```
Java’s synchronized methods

synchronized keyword in front of method.

```java
synchronized void deposit(String name, double amount) {
    double balance = table.get(name);
    balance += amount;
    table.put(name, balance);
}
```

Equivalent to synchronized (this) for body.

```java
void deposit(String name, double amount) {
    synchronized (this) {
        double balance = table.get(name);
        balance += amount;
        table.put(name, balance);
    }
}
```

What about a static synchronized method?
Examples

- **List**
  - add()/remove() in synchronized regions
  - Can you safely do unsynchronized lookups in ArrayList?
  - Iteration?
  - What is the difference between Vector & ArrayList?
    - Prefer Collections.synchronizedList(alist);

- **Maps, Sets, and other collections**
  - Ununsynchronized by default.
  - Map sm = Collections.synchronizedMap(m);
  - Avoids corruption, but { get(); ...; put(); } still isn’t atomic!
  - Consider java.util.concurrent.Concurrent(Hash)Map
Dining Philosophers

- Five diners (threads) at round table.
  - Need 2 forks to eat.
  - Only 5 on table.
- Each picks up left (acquires lock).
  - Then the right (acquires lock).
  - Then eats (does work).
  - Then puts both down (releases locks).
- What (might) go wrong?
- How can you ensure it won’t?
Deadlocks

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock A</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>Lock B</td>
</tr>
<tr>
<td>Lock B</td>
<td>...</td>
</tr>
<tr>
<td>wait</td>
<td>Lock A</td>
</tr>
<tr>
<td>wait</td>
<td>wait</td>
</tr>
</tbody>
</table>

- Doesn’t just affect locks
  - Wait for I/O from another thread/process.
  - Trying to acquire a resource from a “pool”.
  - Think about it when waiting on anything.
Limit the number of locks in your programs
  ▶ By minimizing writable sharing, not by crossing your fingers!

Try to avoid the need to hold multiple locks simultaneously.

If you must hold multiple locks, *order them*.

Hold locks “briefly” (as measured by time *and* code)
  ▶ Keep synchronized regions short.
  ▶ Minimize calls inside locked regions.
  ▶ Avoid I/O inside locked regions.
Using Threads

- Getting the most out of threads is tricky.
- Figuring out where to use threads is hard.
  - Some things are obvious.
  - But most potential uses aren’t.
- Bottom line
  - Use threads where it makes design easier, or you know you have a performance problem.
  - Minimize (mutable) data sharing.
  - Consider “message passing” designs
Read/Write Locks

- java.util.concurrent.locks.ReadWriteLock
- readLock(), writeLock()
- Multiple simultaneous readers allowed.
- Only one writer (with no reader) allowed.
- Useful for read-heavy workloads (a common situation)
- You can not “upgrade” a read lock. Why?
Thread Coordination

- Suppose you build your KD-Tree with multiple threads.
  - At each split, if there’s enough work, “fork” a thread.
  - When can you do a lookup? (That is, when are the threads done?)
- Can be coordinated with synchronized regions.
- There are more direct constructs, and you should use them.
Fork/Join

- Basic operations for run-to-completion threads.
  - Fork: create and start a new thread
  - Join: wait for a thread to exit

- “Structured” in some languages
  - Fork/join are visible in the lexical structure of the program.
  - A forked section might look like a special block.

- Java is unstructured
  - Fork: Thread t = new Thread(work); t.start()
  - Join: t.join() or t.join(timeout)

- How would you join multiple threads?

- PS. Java 7 introduced a Fork/Join Framework that you might consider.
Monitors

- Used when *conditions* control access, not just mutual exclusion.
- We’ll describe what the Java *language* provides.
- But also be aware of what the standard *libraries* offer.
  - java.util.concurrent
  - com.google.common.util.concurrent
- Java provides basic control primitives
  - Waiting (for condition to hold)
  - Notifying (of change in condition)
  - Java does *not* provide the parenthetical semantics on these operations.
Wait-Notify

- A thread wants to yield CPU until an event occurs
- Coordination without termination (unlike join())
  - Wait for another thread to provide needed data.
  - Wait for user to push a button.
- Java provides this in a limited way.
  - Event must be explicitly noticed by another thread.
  - That other thread needs to “tell” the waiting thread about it.

Java Implementation

- Wait by calling o.wait() or o.wait(timeout)
- Another thread must call o.notifyAll() (on the same object).
  - Or o.notify() (in any doubt? use notifyAll)
  - stackoverflow.com/questions/37026/java-notify-vs-notifyall-all-over-again
Wait-Notify to implement a Monitor

- Details are slightly tricky
  - Java allows `wait()` on an object, not a “condition”
    - `obj.wait()` must occur while synchronized on `obj`.
  - Notify must occur on the same object.
    - And also while synchronized by that object.

- Simplified code pattern

```java
synchronized (obj) {
    while (!condition) {
        obj.wait();
    }
}
```
### Producer-Consumer

- **Consumer**
  - Gets next item from queue.
  - Processes that item.

```java
1 while (true) {
2    synchronized (queue) {
3        while (queue.isEmpty()) {
4            try {
5                queue.wait();
6            } catch (InterruptedException e) {} 
7        }
8        next = queue.remove();
9    }
10   next.process();
11 }
```
Producer

- Adds items to queue

```java
while (true) {
    item = <get next item>
    synchronized (queue) {
        queue.add(item);
        queue.notify();
    }
}
```
Building up from Primitives

- Now you know all the primitives you need.
- But actually, you should rarely use them!
- Get to know the java.util.concurrent package.
  - Executors — more flexible than raw Threads
  - Concurrent Data Structures
  - A variety of lock types / synchronization
  - Written by smart, focused people.
- We’ll motivate some of these utilities
  - and implement some (perhaps poorly)
  - so you never should!
Raw threads can be inflexible

- Consider a Web Server or Ray Tracing Engine.
- Both are “embarassingly parallel”
  - Just start one thread per client connection!
  - Just break the image into tiles!
- And yet...
  - Too many clients... nobody served.
  - How many tiles?
Separate specification of work from thread management

- You already wrote your code as Runnable.
- Instead of (new Thread(r)).start();
- Generalize with an Executor.

```java
e = <Get Executor Implementation>;
e.execute(r);
```

- Allows flexible execution.
  - Serial, Pools, Scheduled...

How would you implement a ThreadPoolExecutor?
- What does ThreadPoolExecutor.execute(Runnable) do?
- What will be shared among threads, and how will you protect it?
WorkerThread run()

```java
public void run() {
    while (true) {
        Runnable task = null;
        synchronized (work_queue) {
            while (work_queue.isEmpty()) {
                try {
                    work_queue.wait();
                } catch (InterruptedException e) { }
            }
            task = work_queue.remove();
        }
        task.run(); /* Why not two lines up? */
    }
}
```
Handling Errors

- What if the internal `run()` throws an exception?
  - Do you want to stop the worker?
  - Do you want to record the report/problem?

```java
try {
    task.run();
} catch (Throwable t) {
    // log an error (and stack trace?)
}
```

- Be sure to catch `Throwable`.
- Runnable doesn’t support returning values (or errors).
Adding to the queue()

```java
public void execute(Runnable task) {
    synchronized (work_queue) {
        work_queue.add(task);
        work_queue.notifyAll();
    }
}
```

- Together, these are the Monitor code pattern.
- Specifically Producer/Consumer.
The “monitor” code fragment is not very clear.
A BlockingQueue is a nicer abstraction.

```java
public void run() {
    while (true) {
        try {
            Runnable task = work_queue.take();
            task.run(); /* No worries about being slow. */
        } catch (InterruptedException e) { }
    }
}

class Runnable {
    public void run() { /* Code to be executed by the thread */ }
}

public void execute(Runnable task) {
    try {
        work_queue.put(task);
    } catch (InterruptedException e) { }
}
```
Task completion and returning values

- There’s no longer a Thread to join().
- How can we tell if our tasks are complete?
  - Do you care about individual tasks?
  - Or maybe just that all of them are done.
- Ideas?
Several designs

- Query the queue size (all).
  - Only if you know all tasks submitted.
  - Queue would need to “lie” about the currently running task.

- Use a CountDownLatch (n).

- Provide a “better” Runnable implementation that wraps Runnables.
  - Make a class that “knows” it has been run.
  - `void execute(DoneRunnable task)`
  - `task.isDone()`
  - JDK: `void execute(RunnableFuture<T> task)`

- Use a richer execute() interface.
  - `Future<T> submit(Callable<T> task)`
  - Future and Callable are real. Speculate on their interface?
  - Available from an ExecutorService
Suppose you want to use a Map or Queue in several threads.
You could use Collections.synchronizedMap (et al.)
But
  ▶ Those make every method synchronized
  ▶ And still require external synchronization for common ops.

```java
synchronized (map) {
  if (!map.containsKey(key))
    value = <some lengthy computation>;
  return map.put(key, value);
  else
    return map.get(key);
}
```

ConcurrentHashMap offers putIfAbsent and even computeIfAbsent.
ConcurrentMap, ConcurrentQueue interfaces.
ConcurrentHashMap, ConcurrentLinkedQueue... classes.
Allow multiple readers (and sometimes writers)
oldVal = map.putIfAbsent(key, value);
newVal = map.computeIfAbsent(key, Tile::new)
Simple atomic ooperations

- A common shared variable is a statistics counter.
- Multiple threads need to increment atomically.
- `java.util.concurrent.atomic.AtomicInteger`
Immutable Objects

Consider a Complex number package.

```java
1 public void add(other) {
2   real += other.real;
3   imag += other.imag;
4 }
```

vs

```java
1 public Complex add(Complex other) {
2   return new Complex(real + other.real,
3                        imag + other.imag);
4 }
```
Files are not inherently thread-safe.
  - I/O may not be atomic.
  - POSIX specifies some guarantees, but libraries obfuscate details.
  - And file position is shared across threads

Try to access files with only one thread
Or use synchronization (on the same object!)
  - Not like this: synchronized (new File("safe.txt")) { ... }

Don’t worry about pure-read workloads.
Log files are problematic (maybe you don’t mind).
  - OS supports opening in “append” mode.
  - new FileWriter(..., true)
Lazy initialization is surprisingly hard

In a multithreaded application, this is not safe!

```java
class Geocoder {

    private static Service svc = null;

    public static Service getSvc() {
        if (svc == null)
            svc = new Service();

        return svc;
    }
}
```

Could double create or worse!
And this is unnecessarily slow

```java
class Geocoder {
    private static Service svc = null;
    public static synchronized Service getSvc() {
        if (svc == null)
            svc = new Service();
        return svc;
    }
}
```

You pay synchronization cost every time you use the singleton.
And this is wrong!

class Geocoder {
    private static Service svc = null;
    public static Service getSvc() {
        if (svc == null) {
            synchronized (Geocoder.class) {
                if (svc == null) {
                    svc = new Service();
                }
            }
            return svc;
        }
    }
}

Do you want to know about the volatile keyword? Start here:
Leverage Java’s initialization rules.

If you don’t care how soon svc is initialized.

```java
private static Service svc = new Service();
public static Service getSvc() {
    return svc;
}
```

If you want to wait as long as possible.

```java
class Geocoder {
    private static class ServiceHolder {
        public static Service svc = new Service();
    }
    public static Service getSvc() {
        return ServiceHolder.svc;
    }
}
```