Java and Locks

ICS432

Concurrent and High-Performance Programming

Henri Casanova (henric@hawaii.edu)

Java and Locks

- Java makes it very simple
- EVERY Java Object has a lock hidden within it!
  - That implements adaptive/hybrid spinning/blocking
    - Spins for a while, then blocks
- *Some* methods can be declared **`synchronized`**
  - A class can have both synchronized and non-synchronized methods
- Synchronized methods are executed in mutual exclusion with implicit calls for lock() and unlock() on the lock hidden within the object
- So you can use locks in Java without calling lock() or unlock()
  - This way you won't forget the unlock()!

Synchronized Methods

```
public class SomeClass {

  public synchronized void SomeMethod() {
    . . .
  }

  public synchronized void SomeOtherMethod() {
    . . .
  }

}
```

- At all times: #threads in SomeMethod() + #threads in SomeOtherMethod() <= 1

- Coarse-grained mutual exclusion
- Implemented internally with a single lock, invisible to you (in the Object class)

Example of `synchronized`

```
public class Counter {
  private int value;
  public Counter() {
    value = 0;
  }

  public synchronized void increment() {
    value++;
  }

  public synchronized void decrement() {
    value--;
  }

  public int getValue() {
    return value;
  }
}
```

```
Counter counter = new Counter();

// Thread 1
. . .
counter.increment();
. . .

// Thread 2
. . .
counter.decrement();
. . .
```

- Methods increment() and decrement() are **thread-safe**

Synchronized Statements

- It is not always good to have an entire method be used in mutual exclusion
  - Perhaps there are only a few “critical” statements in the method
  - And we have seen that shorter critical sections are better for concurrency/performance
  - The solution could be to put the critical statements in their own methods
  - But then we artificially create more method calls, which may add clutter and also harms performance (although we hope the compiler does inlining)
- As a result, Java provides ways to have synchronized statements inside non-synchronized methods
  - And so the weirdness begins…

Synchronized Statements

```
public class Counter {
  private int value;
  public Counter() {
    value = 0;
  }

  public void increment() {
    System.out.println(“hello”);
    synchronized(this) {
      value++;
    }
    System.out.println(“bye”);
  }

  . . .
}
```

- Two threads can print “hello” and/or “bye” at the same time
- But only one can increment the value at a time

- The **`synchronized(this)`** statement makes it possible to make short critical sections and thus maximize concurrency
- **`this`** refers to the current instance, which is an Object, and thus a lock inside it!
- synchronized(x) means“call lock() on the lock that is inside the object referenced by x”

Only One Lock?

- Having only one lock per object can be a problem
- Say you define a class in which not all methods need to be in mutual exclusion
- Example:
  - Methods f1 and f2 should be executed in mutual exclusion
  - Methods f3 and f4 should be executed in mutual exclusion
  - Methods f1 and f3 can be executed concurrently
  - Methods f1 and f4 can be executed concurrently
  - Methods f2 and f3 can be executed concurrently
  - Methods f2 and f4 can be executed concurrently

Example: Two Counters

```
public class TwoCounters {
  private int value1, value2;
  public TwoCounters() {
    value1 = 0; value2 = 0;
  }

  public void increment1() {
    synchronized(this) {
      value1++;
    }
  }

  public void increment2() {
    synchronized(this) {
      value2++;
    }
  }
}
```

- This solution is correct, but overly restrictive
- Two threads should be allowed to update two different counters simultaneously
- In this case, with a single lock there is no way to do this
- Therefore, we need to have multiple locks
- Problem: The TwoCounters object has only one (hidden) lock!

Synchronizing on Multiple Objects

- When synchronizing statements one uses the synchronized(this) statement
- The “this” specifies that one uses the lock inside the current object (this)
- It’s standard, but in fact one can synchronize on the lock of **any** object
- This is going to “look weird”, but solves the quandary in the previous slide
  - One of the many examples of “clean design vs. high performance” struggles

Example: Two Counters

```
public class TwoCounters {
  private int value1, value2;
  private Object lock1, lock2;


  public TwoCounters() {
    value1 = 0; value2 = 0;
    lock1 = new Object();
    lock2 = new Object();
  }


  public void increment1() {
    synchronized(lock1) {
      value1++;
    }
  }
  public void increment2() {
    synchronized(lock2) {
      value2++;
    }
  }
}
```

- Now we have a distinct lock for each counter
- Note that these locks are encapsulated within Object objects
- I **name** these objects lock1 and lock2 just to remind myself that they are used exclusively for mutual exclusion
- I could have used *any* object in the program really, but it's typically not very readable
- Code like synchronized(window) would seem to imply "window synchronization" (whatever that means), when really it's just "lock/unlock the lock that happens to be hidden in the window object"
- Many programmers find this confusing, and they're right
- We will see later that there is a Lock class we can use….

Synchronized Class

- So far, we have seen mutual exclusion over objects, i.e., instances of classes
- Sometimes one wants a particular method to be in mutual exclusion over all instances of the class
  - Example: only one thread can update some global sum of all the counters at a given time
- One way to do this is to encapsulate the global sum in its own object, with its own lock
- Another way is to declare a “class method”, i.e., a static method, as synchronized

Example

```
public class Counter {
  private int value;
  static private int sum = 0;

  public Counter() {
    value = 0;
  }

  public void increment() {
    synchronized(this) { value++; }
  }

  public static synchronized void updateSum(int value) {
    sum += value;
  }
}
```

- Only one Counter object can update the class variable at a time
- You guessed it, each class has a lock hidden inside it
- So if you define one Java class and creates 6 instances of that class, in total you've created 7 hidden locks

Summary so Far

- The **`synchronized`** keyword is the way to implement mutual exclusion
  - At the class level, e.g., **`public static synchronized`**
  - At the method level, e.g., **`public synchronized`**
  - At the statement level, e.g., **`synchronized(…){ }`**
- One can create objects just for the purpose of using their locks for mutual exclusion
- The main advantage of the **`synchronized`** keyword: you will never forget to call unlock()
  - Which is of course a common cause of deadlocks

java.util.concurrent.locks

- This package provides explicit lock implementations
- But then you mustn't forget to call unlock:

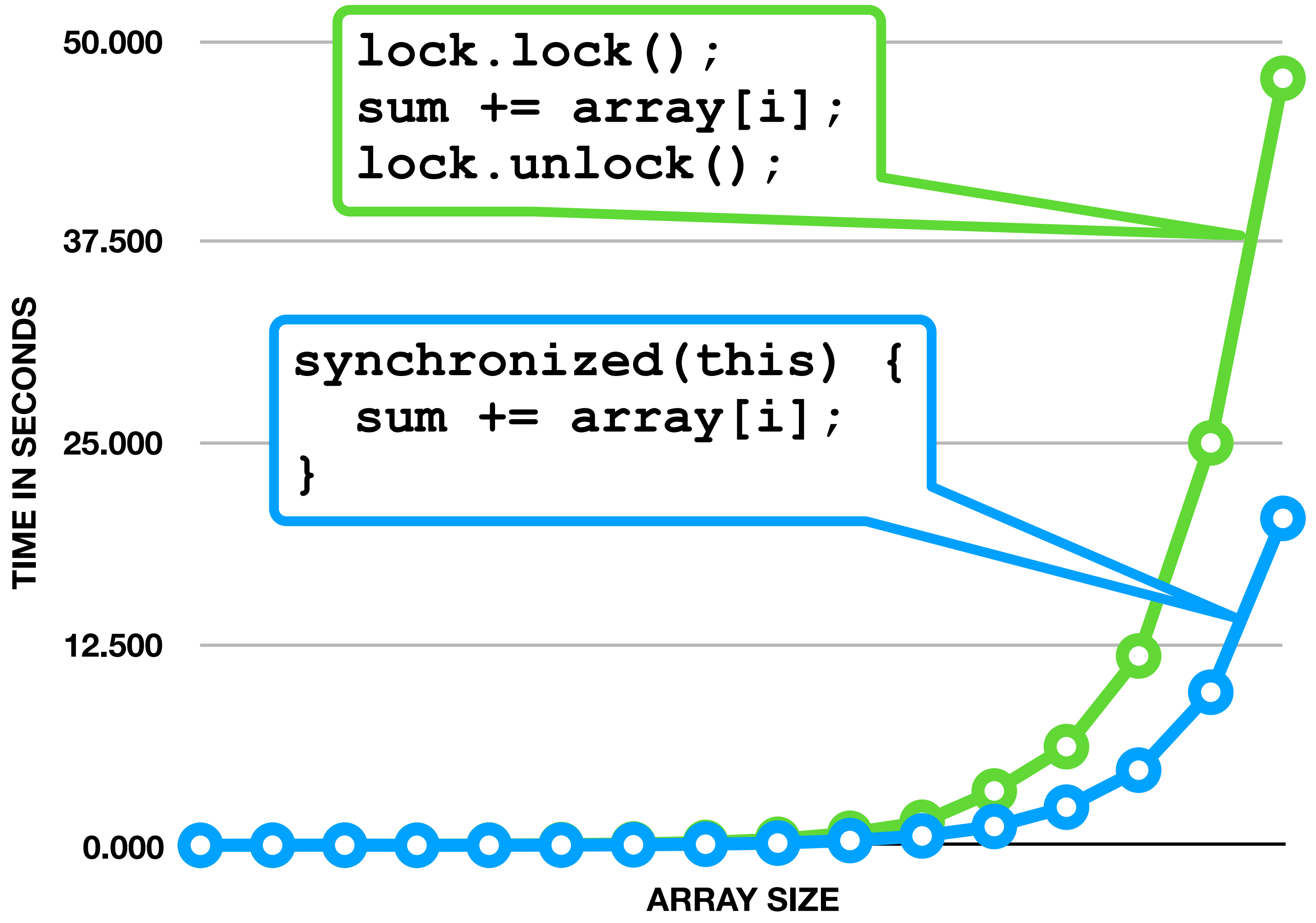
```
lock.lock();
try {
    . . .
} finally {
  lock.unlock()
}
```

- Why would you use these rather than relying on the one-size-fits-all **`synchronized`**?

java.util.concurrent.locks

- **Main motivation:** More functionality/flexibility
- `ReentrantLock` has many useful methods
  - getOwner(), getQueueLength(), isHeldByCurrentThread(), isLocked(), tryLock(), …
- `ReadWriteLock` is useful
  - A special lock that can be held by either one "writer" thread, or by any number of "reader" threads
  - More on this later...
- These locks never spin, which can have lower performance than `synchronized` for short critical sections
  - And may limit compiler optimizations

java.util.concurrent.locks



```
lock.lock();
sum += array[i];
lock.unlock();
```

```
synchronized(this) {
  sum += array[i];
}
```

on my laptop

java.util.concurrent Atomics

- java.util.concurrent provides many simple classes of variable that can be updated atomically
- Say you want to write a program that maintains a shared counter
  - You'll have to create a new class with synchronized methods for increment, decrement, etc.
  - Almost all Java developers who write concurrent programs have done this and will do it again
- java.util.concurrent provides all this
  - Let's look at the documentation for AtomicInteger…

java.util.concurrent Atomics

- Many “atomics” in java.util.concurrent:
  - AtomicBoolean
  - AtomicIntegerArray
  - AtomicLongArray
  - . . .
- So, rather than re-inventing the wheel each time, using these classes from java.util.concurrent may be a better idea
  - Some people do not find them very readable and end up writing wrapper functions around them
  - Still, removes the need to deal with **`synchronized`**

Let's Talk about `volatile` again!

- Remember the Java **`volatile`** keyword?
- Many developers don't know about volatile and yet they have written multi-threaded Java for months (years?)
- Some day, they get hit with the "Thread #1 updates something, but Thread #2 never sees it!!" bug
  - This happened with a ICS111 TA years ago who was writing a multi-threaded GUI in which there were long, unexplained lags
- But in almost all multi-threaded programs we need threads to see recent updates to memory at least for some variables!
  - After all we use threads because they share memory, so if they don't "see" memory updates, what's the point?
- How could a Java developer not know **`volatile`**???

`synchronized` is more than it seems

- **It turns out that in Java entering a `synchronized` method / block of code ALSO synchronizes memory**
  - Acquiring a lock forces all variable values to be updated with “main memory” values
  - Releasing a lock forces all written variable values to be written to “main memory”
- In other terms, each time you enter/leave a synchronized section of code, memory fence instructions are executed
  - Which has a performance hit
- So in all the examples we’ve seen in the previous and this module we never needed to worry about “will the thread see the last value?” because accesses to shared variables were always within synchronized methods or blocks!
- This is why **`volatile`** is not well known, and so baffling when you discover it on your own (or worse, when it causes a bug)

`volatile` and Tread Safety?

- Volatile does nothing for atomicity
- Having multiple threads do “var++” on volatile variable var is still a race condition
  - Thread #1 reads the value into a register
  - Thread #2 reads the value into a register
  - Thread #1 writes the value to RAM
    - At this point all threads reading the value see the new value in RAM
  - Thread #2 writes the value to RAM
    - We still have a lost update
    - In spite of Thread #2 always seeing the latest value in RAM (but not in registers!)

Example with a tiny class

- Consider the following simple class:

```
public class SomeValue {
  private float value = 0.0;

  void set(float v) {
    value = v;
  }

  float get() {
    return value;
  }
```

- In a multi-threaded context, the problem is thst threads may not see the latest value

Example with a tiny class

- Consider the following simple class:

```
public class SomeValue {
  private volatile float value = 0.0;

  void set(float v) {
    value = v;
  }

  float get() {
    return value;
  }
```

- A good fix is to make the variable volatile

Example with a tiny class

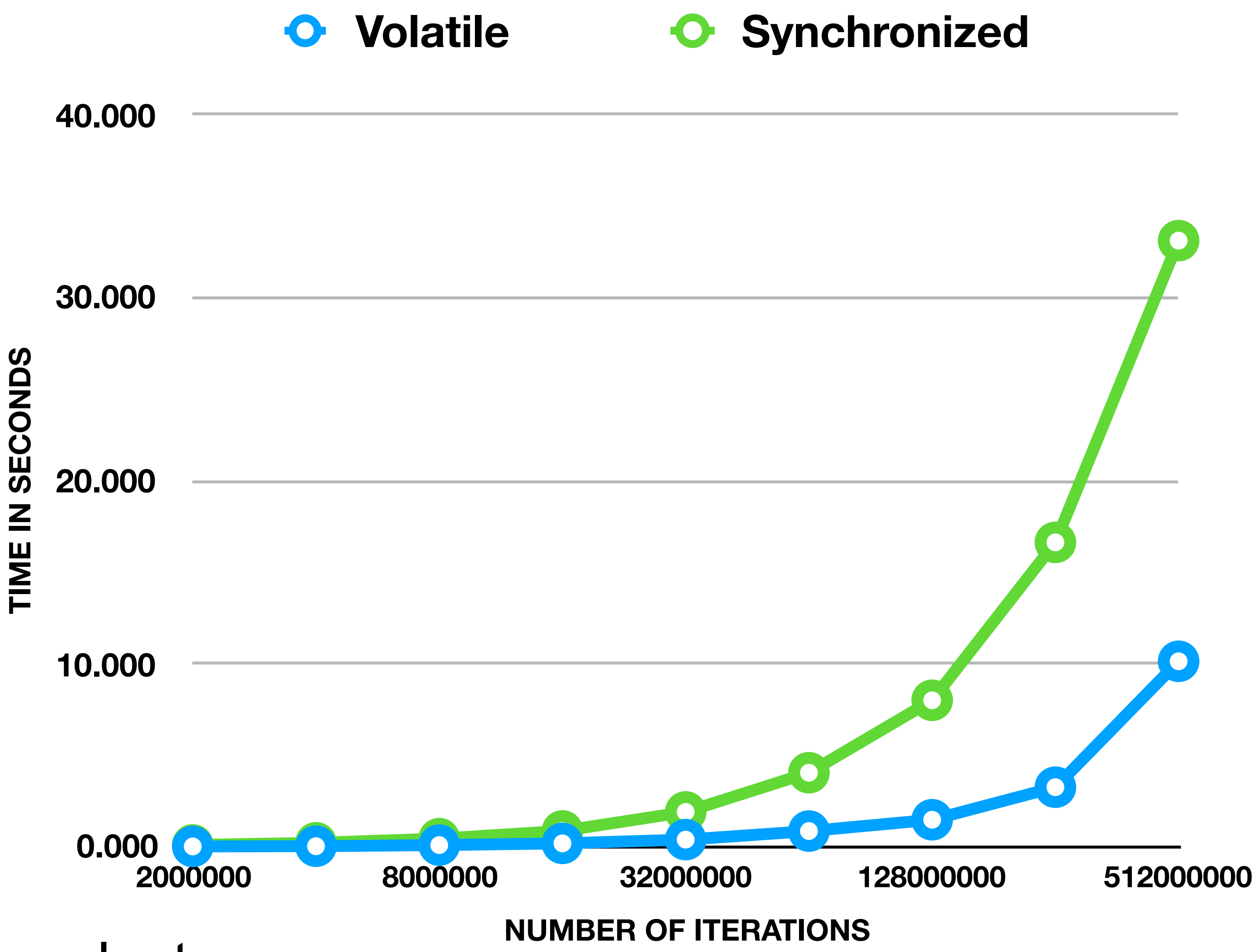
- Consider the following simple class:

```
public class SomeValue {
  private float value = 0.0;

  void synchronized set(float v) {
    value = v;
  }

  float synchronized get() {
    return value;
  }
```

More expensive, and it looks strange to put `synchronized` around atomic statements!

- A **not-as-good** fix is to make methods synchronized
  - We're using synchronized not because we want to prevent race conditions, but just because it has the side effect of having threads see the latest value
  - It works, but performance is much lower than just using volatile

Performance Comparison



on my laptop

In a NutShell

- If your variables are read/written/ updated in synchronized blocks, you don’t need `volatile` at all
- You should use `volatile` when
  - One thread writes the variable
  - One or more threads read the variable
- In this case you don’t need mutual exclusion, and `volatile` is much cheaper than `synchronized`

Since we're Talking `volatile`…

- How many of you have heard of the Singleton design pattern? (How about design patterns in general?)

```
public class Whatever {
  private SomeObject instance = null;
  public SomeObject getInstance() {
    if (instance == null) {
      instance = new SomeObject();
    }
    return instance;
  }
}
```

- Useful when there must be a single instance of one class
- The first call to the getInstance() method creates that instance
- Every subsequent call just gets a reference to the instance

Multi-threaded Singleton

- Of course, with threads, we must make it synchronized:

```
public class Whatever {
  private SomeObject instance = null;
  public synchronized SomeObject getInstance() {
    if (instance == null) {
      instance = new SomeObject();
    }
    return instance;
  }
}
```

- We have to make it thread-safe because “testing and doing” isn’t atomic
- So far, so good

Multi-threaded Singleton

- If you are a performance person, you hate this code

```
public class Whatever {
  private SomeObject instance = null;
  public synchronized SomeObject getInstance() {
    if (instance == null) {
      instance = new SomeObject();
    }
    return instance;
  }
}
```

- “if (instance == null)” is useless 99.99999% of the time (after the first call it always returns false), but makes the critical section longer!
- And your ICS432 professor told you to make critical sections short!

Double-Checked Locking (DCL)

- A popular performance fix:

```
public class Whatever {
  private SomeObject instance = null;
  public SomeObject getInstance() {
    if (instance == null) { // first check
      synchronized(this){
        if (instance == null) { // second check
          instance = new SomeObject();
        }
      }
    }
    return instance;
  }
}
```

- Enter the synchronized section only if needed!

Double-Checked Locking (DCL)

- A popular performance fix:

```
public class Whatever {
  private SomeObject instance = null;
  public SomeObject getInstance() {
    if (instance == null) { //
      synchronized(thi
        if (i



         rn instance;
    }
}
```

There is an insanely subtle problem with this code

- Enter the synchronized section only if needed!

The Problem with DCL

- The problem takes us (again) down the path of correctness problems posed by compiler optimization
- A very common-place optimization is inlining: replacing a method call by the code of the method
  - This saves a method call, which saves on stack operations (ICS312 anyone?)
- So a Java compiler may inline the constructor call
- Say SomeObject has the following constructor:

```
public SomeObject() {
   this.x = 12;
   this.y = 42;
}
```

The “real” Constructor Code

- The constructor really does this:

```
// Constructor  code
SomeObject *ptr = (SomeObject *)malloc(...);
ptr->x = 12;
ptr->y = 42;
return ptr;
```

- Let’s inline this code in the main program…

Constructor Inlining

```
public class Whatever {
  private SomeObject instance = null;
  public SomeObject getInstance() {
    if (instance == null) { // first check
      synchronized(this){
        if (instance == null) { // second check
          instance = (SomeObject *)malloc(…);
          instance->x = 12;
          instance->y = 42;
        }
      }
    }
    return instance;
  }
}
```

Constructor Inlining

```
public class Whatever {
  private SomeObject instance = null;
  public SomeObject getInstance() {
    if (instance == null) { // first check
      synchronized(this){
        if (instance == null) { // second check
          instance = (SomeObject *)malloc(…);
          instance->x = 12;
          instance->y = 42;
        }
      }
    }
    return instance;
  }
}
```

anybody sees a problem?

Constructor Inlining

```
public class Whatever {
  private SomeObject instance = null;
  public SomeObject getInstance() {
    if (instance == null) { // first check
      synchronized(this){
        if (instance == null) { // second check
          instance = (SomeObject *)malloc(…);
          instance->x = 12;
          instance->y = 42;
        }
      }
    }
    return instance;
  }
}
```

Thread #1 has just called malloc(), and is context-switched out

Thread #2 arrives, sees the instance a NOT NULL, retrieves the reference to it, and accesses uninitialized fields!!!

Solution: make instance volatile

```
public class Whatever {
  private volatile SomeObject instance = null;
  public SomeObject getInstance() {
    if (instance == null) { // first check
      synchronized(this){
        if (instance == null) { // second check
          instance = (SomeObject *)malloc(…);
          instance->x = 12;
          instance->y = 42;
        }
      }
    }
    return instance;
  }
}
```

`volatile` will guarantee the read/writes are in program order and prevent the compiler from doing some optimization

In this case, it prevents a partially initialized object from being read

Double-Checked Locking

- For a ~~fun~~ scary time, do a Web search on “double-checked locking”, “java”, “harmful”, “volatile”
- Safe DCL with Java using volatile started with Java 5 (i.e., the Java people fixed DCL)
- There is a lot of confusion out there
- Many think that a Singleton pattern is not a good idea in the first place?
  - Can’t you use a static variable???
- And often DCL only saves a bit of performance… do you really have thousands of threads all calling getInstance() like crazy?
- The goal here was to expose you to yet-another-example-that-shows-that-abstractions-are-not-perfect, especially because we’re so performance-driven…
- This could be a constant theme in this course, but we will just see a few examples here and there

Conclusions

- Two ways to do locks in Java:
  - The **`synchronized`** keyword
  - **`java.util.concurrent.locks`**
- Each has drawbacks and advantages
- Synchronization implies memory fences
  - Google “Java Memory Model” to go down a fascinating but almost bottomless rabbit hole
- Double-checked locking is weird

- Onward to Homework Assignment #4…