The Java™ Technology Memory Model: the Building Block of Concurrency

Jeremy Manson, Purdue University
William Pugh, Univ. of Maryland

http://www.cs.umd.edu/~pugh/java/memoryModel/

TS-1630

Goal

Learn the building blocks of concurrency and how to design clever but correct concurrent abstractions and design patterns.

Agenda

Scope

The Fundamentals: Happens-Before Ordering

Using Volatile

Thread Safe Lazy Initialization

Final Fields

Recommendations

Java™ Technology Thread Specification

- Revised as part of JSR 133
  - Part of Tiger and later releases
- Goals
  - Clear and easy to understand
  - Foster reliable multithreaded code
  - Allow for high performance JVM™ machines
- Not all of these ideas are guaranteed to work in previous versions
  - Previous thread spec was broken
    - Forbid optimizations performed by many JVM machines

Safety Issues in Multithreaded Systems

- Many intuitive assumptions do not hold
- Some widely used idioms are not safe
  - Original double-checked locking idiom
  - Checking non-volatile flag for thread termination
- Can’t depend on testing to check for errors
  - Some anomalies will occur only on some platforms
    - e.g., multiprocessors
  - Anomalies will occur rarely and non-repeatedly

This Talk

- Describe building blocks of synchronization and concurrent programming in Java language
  - Both language primitives and util.concurrent abstractions
- Explain what it means for code to be correctly synchronized
- Try to convince you that clever reasoning about unsynchronized multithreaded code is almost certainly wrong
  - And not needed for efficient and reliable programs

This Talk

- We will be talking mostly about
  - Synchronized methods and blocks
  - Volatile fields
- Same principles apply to JSR 166 classes
- Will also talk about final fields and immutability

Taxonomy

- High level concurrency abstractions
  - JSR 166 and java.util.concurrent
- Low level locking
  - `synchronized()` blocks and util.concurrent.locks
- Low level primitives
  - Volatile variables, java.util.concurrent.atomic classes
  - Allows for non-blocking synchronization
- Data races: deliberate undersynchronization
  - Avoid!
  - Not even Doug Lea can get it right

Agenda

Scope

The Fundamentals: Happens-Before Ordering

Using Volatile

Thread Safe Lazy Initialization

Final Fields

Recommendations

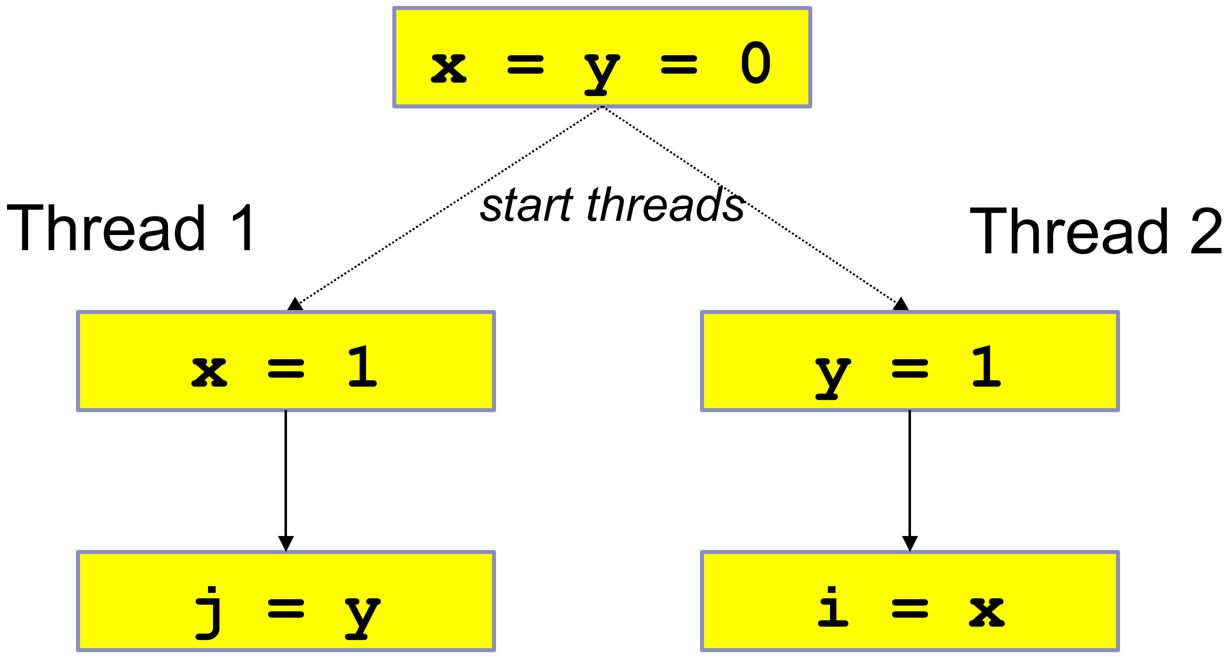
Synchronization Is Needed for Blocking and Visibility

- Synchronization isn’t just about mutual exclusion and blocking
- It also regulates when other threads must see writes by other threads
  - When writes become visible
- Without synchronization, compiler and processor are allowed to reorder memory accesses in ways that may surprise you
  - And break your code

Don’t Try To Be Too Clever

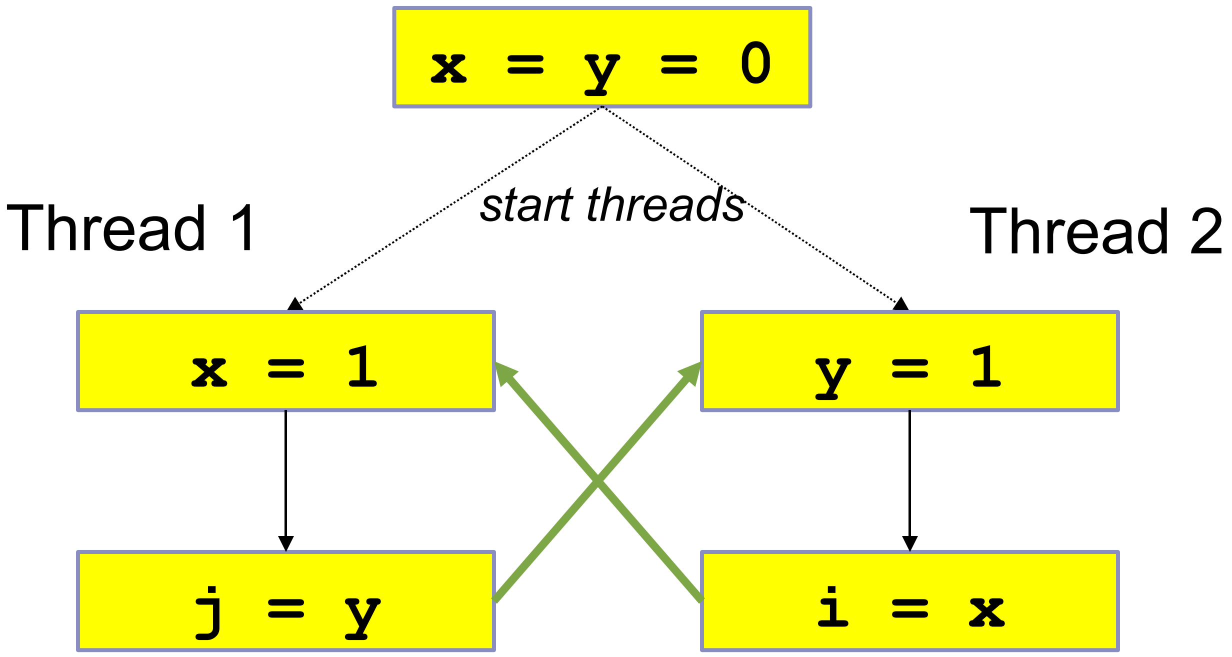
- People worry about the cost of synchronization
  - Try to devise schemes to communicate between threads without using synchronization
    - Locks, volatiles, or other concurrency abstractions
- Nearly impossible to do correctly
  - Interthread communication without synchronization is not intuitive

Quiz Time



Can this result in i = 0 and j = 0?

Answer: Yes!

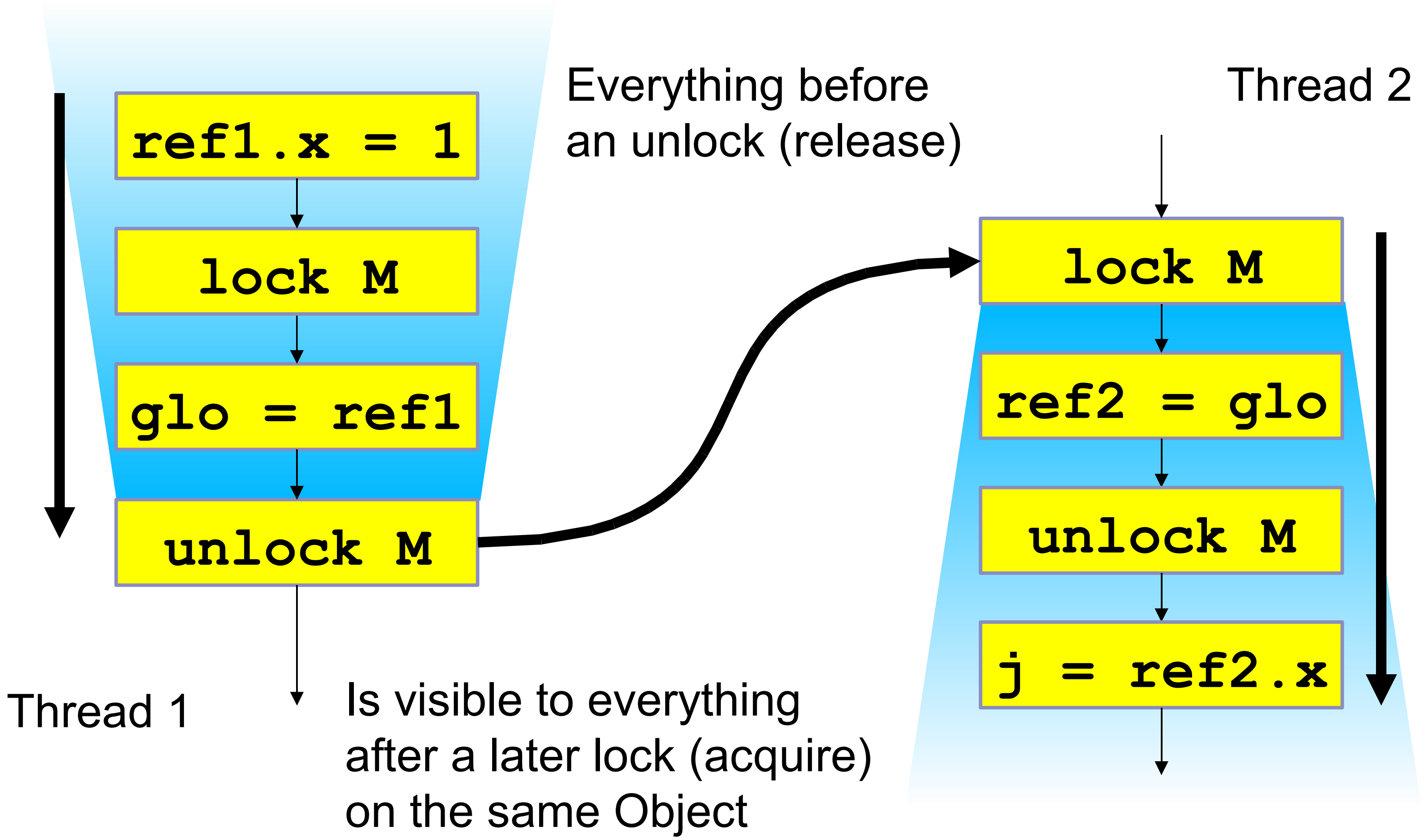


How can i = 0 and j = 0?

How Can This Happen?

- Compiler can reorder statements
  - Or keep values in registers
- Processor can reorder them
- On multi-processor, values not synchronized to global memory
- The memory model is designed to allow aggressive optimization
  - Including optimizations no one has implemented yet
- Good for performance
  - Bad for your intuition about insufficiently synchronized code

When Are Actions Visible to Other Threads?



Thread 1
ref1.x = 1
lock M
glo = ref1
unlock M
Everything before an unlock (release)
Thread 2
lock M
ref2 = glo
unlock M
j = ref2.x
Is visible to everything after a later lock (acquire) on the same Object

Release and Acquire

- All memory accesses before a release
  - Are ordered before and visible to
  - Any memory accesses after a matching acquire
- Unlocking a monitor/lock is a release
  - That is acquired by any following lock of that monitor/lock

Happens-Before Ordering

- A release and a matching later acquire establish a happens-before ordering
- Execution order within a thread also establishes a happens-before order
- Happens-before order is transitive

Data Race

- If there are two accesses to a memory location,
  - At least one of those accesses is a write, and
  - The memory location isn’t volatile, then
- The accesses must be ordered by happens-before
- Violate this, and you may need a PhD to figure out what your program can do
  - Not as bad/unspecified as a buffer overflow in C

Need Something More Concrete?

- Okay, perhaps this is a little too abstract
- What does entering/leaving a synchronized block actually do?

Synchronization Actions (Approximately)

He’s lying

```
int z = o.field1;
//  block until obtain lock
synchronized(o) {
   //  get main memory value of field1 and field2
   int x = o.field1;
   int y = o.field2;
   o.field3 = x+y;
   //  commit value of field3 to main memory
}
//  release lock
moreCode();
```

This is a gross oversimplification

Depend on this at your great peril

The figure from five slides earlier is a much better mental image

Ordering

- Roach motel ordering
  - Compiler/processor can move accesses into synchronized blocks
  - Can only move them out under special circumstances, generally not observable
- But a release only matters to a matching acquire
- Some special cases:
  - Locks on thread local objects are a no-op
  - Reentrant locks are a no-op
  - Java SE 6 (Mustang) does optimizations based on this

Agenda

Scope

The Fundamentals: Happens-Before Ordering

Using Volatile

Thread Safe Lazy Initialization

Final Fields

Recommendations

Volatile Fields

- If a field could be simultaneously accessed by multiple threads, and at least one of those accesses is a write
- Two choices:
  - Use synchronization to prevent simultaneous access
  - Make the field volatile
    - Serves as documentation
    - Gives essential JVM machine guarantees
- Can be tricky to get volatile right, but nearly impossible without volatile or synchronization

What Does Volatile Do?

- Reads and writes go directly to memory
    - Not cached in registers
- Volatile longs and doubles are atomic
    - Not true for non-volatile longs and doubles
- Volatile reads/writes cannot be reordered
- Reads/writes become acquire/release pairs

Volatile Happens-Before Edges

- A volatile write happens-before all following reads of the same variable
- A volatile write is similar to a unlock or monitor exit
    - In terms of the happens-before edges it creates
- A volatile read is similar to a lock or monitor enter

Volatile Guarantees Visibility

- `stop` must be declared volatile
  - Otherwise, compiler could keep in register

```
class Animator implements Runnable {
  private volatile boolean stop = false;
  public void stop() { stop = true; }
  public void run() {
    while (!stop)
      oneStep();
      try { Thread.sleep(100); } …;
  }
  private void oneStep() { /*...*/ }
}
```

Volatile Guarantees Ordering

- If a thread reads data, there is a happens-before edge from write to read of **`ready`** that guarantees visibility of **`data`**

```
class Future {
   private volatile boolean ready;
   private Object data;
   public Object get() {
      if (!ready)
        return null;
      return data;
      }
```

```
public synchronized
  void setOnce(Object o) {
    if (ready) throw … ;
    data = o;
    ready = true;
    }
 }
```

More Notes on Volatile

- Incrementing a volatile is not atomic
  - If threads threads try to increment a volatile at the same time, an update might get lost
- Volatile reads are very cheap
  - Volatile writes cheaper than synchronization
- No way to make elements of an array be volatile
- Consider using util.concurrent.atomic package
  - Atomic objects work like volatile fields
  - But support atomic operations such as increment and compare and swap

Other Happens-Before Orderings

- Starting a thread happens-before the run method of the thread
- The termination of a thread happens-before a join with the terminated thread
- Many util.concurrent methods set up happens-before orderings
  - Placing an object into any concurrent collection happens-before the access or removal of that element from the collection

Agenda

Scope

The Fundamentals: Happens-Before Ordering

Using Volatile

Thread Safe Lazy Initialization

Final Fields

Recommendations

Thread Safe Lazy Initialization

- Want to perform lazy initialization of something that will be shared by many threads
- Don’t want to pay for synchronization after object is initialized

Original Double Checked Locking

```
// FIXME: THIS CODE IS BROKEN!

Helper helper;

Helper getHelper() {
  if (helper == null)
    synchronized(this) {
      if (helper == null)
        helper = new Helper();
      }
  return helper;
}
```

Correct Double Checked Locking

```
// THIS CODE WORKS

volatile Helper helper;

Helper getHelper() {
  if (helper == null)
    synchronized(this) {
      if (helper == null)
        helper = new Helper();
      }
  return helper;
}
```

We Don’t Want to Hear Your Solution

- Frankly, we don’t want to hear your solution on how to “fix” double checked locking without using any kind of synchronization or volatile fields
  - Unless a happens-before order is established between the threads, it cannot work
  - We’ve seen hundreds of emails with proposed solutions, none of them work

Even Better Static Lazy Initialization

- If you need to initialize a singleton value
  - Something that will only be initialized once per Java VM
- Just initialize it in the declaration of a static variable
  - Or in a static initialization block
- Spec guarantees it will be initialized in a thread safe way at the first use of that class
  - But not before

Threadsafe Static Lazy Initialization

```
class Helper {
static final Helper helper = new Helper();

public static Helper getHelper() {
   return helper;
}

private Helper() {
   …
   }
}
```

Agenda

Scope

The Fundamentals: Happens-Before Ordering

Using Volatile

Thread Safe Lazy Initialization

Final Fields

Recommendations

Thread Safe Immutable Objects

- Use immutable objects when you can
  - Lots of advantages, including reducing needs for synchronization
- Can make all fields final
  - Don’t allow other threads to see object until construction complete
- Gives added advantage
  - Spec promises immutability, even if malicious code attacks you with data races

Data Race Attack

- Thread 1 creates instance of a class
- Thread 1 hands the instance to thread 2 without using synchronization
- Thread 2 accesses the object
- It is possible, although unlikely, that thread 2 could access an object before all the writes performed by the constructor in thread 1 are visible to thread 2

Strings Could Change

- Without the promises made by final fields, it would be possible for a String to change
  - Created as “/tmp/usr”.substring(4,8)
  - First seen by thread 2 as “/tmp”
  - Later seen by thread 2 as “/usr”
- Since Strings are immutable, they don’t use synchronization
  - Final fields guarantee initialization safety

A Hack to Change Final Fields

- There are times when you may need to change final fields
  - Clone()
  - Deserialization()
- Only do this for newly minted objects
- Use Field.setAccessible(true)
  - Only works in Java version 5.0+
- Be nice to have a better solution in Dolphin

Optimization of Final Fields

- New spec allows aggressive optimization of final fields
  - Hoisting of reads of final fields across synchronization and unknown method calls
  - Still maintains immutability
- Should allow for future JVM machines to obtain performance advantages

Agenda

Scope

The Fundamentals: Happens-Before Ordering

Using Volatile

Thread Safe Lazy Initialization

Final Fields

Recommendations

These Are Building Blocks

- If you can solve your problems using the high level concurrency abstractions provided by util.concurrent
  - Do so
- Understanding the memory model, and what release/acquire means in that context, can help you devise and implement your own concurrency abstractions
  - And learn what not to do

Mostly, It Just Works

- If you aren’t trying to be clever, the memory model just works and doesn’t surprise
    - No change from previous generally recommended programming practice
- Knowing the details can
    - Reassure those whose obsess over details
    - Clarify the fine line between clever and stupid

Synchronize When Needed

- Places where threads interact
  - Need synchronization
  - May need careful thought
  - Don’t need clever hacks
  - May need documentation
  - Cost of required synchronization not significant
    - For most applications
    - No need to get tricky
- Performance of the util.concurrent abstractions is amazing and getting better

Watch Out for Useless Synchronization

- Using a concurrent class in a single threaded context can generate measurable overhead
  - Synchronization on each access to a Vector, or on each IO operation
- Substitute unsynchronized classes when appropriate
  - ArrayList for Vector
- Perform bulk I/O or use java.nio

Sometimes Synchronization Isn’t Enough

- Even if you use a concurrent class, your code may not be thread safe

// THIS CODE WILL NOT WORK

```
ConcurrentHashMap<String,ID> h;
ID getID(String name) {
    ID x = h.get(name);
    if (x == null) {
        x = new ID();
        h.put(name, x);
    }
    return x;
}
```

- Watch out for failures of atomicity

Documenting Concurrency

- Often the concurrency properties of a class are poorly documented
  - Is an IO stream thread safe?
- Not as simple as “this class is thread safe”
- Look at util.concurrent documentation
- Look at annotations described in *Java Concurrency in Practice*
  - Some of which are checked by FindBugs

Designing Fast Concurrent Code

- Make it right before you make it fast
- Reduce synchronization costs
  - Avoid sharing mutable objects across threads
  - Avoid old Collection classes (Vector, Hashtable)
  - Use bulk I/O (or, even better, java.nio classes)
- Use java.util.concurrent classes
  - Designed for speed, scalability and correctness
- Avoid lock contention
  - Reduce lock scopes
  - Reduce lock durations

Wrap-Up

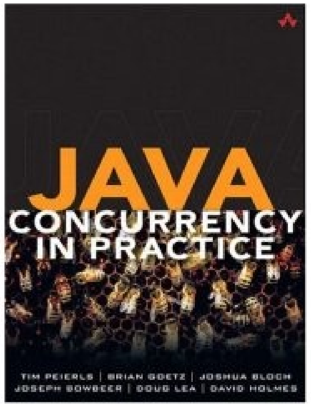
- Cost of synchronization operations can be significant
  - But cost of needed synchronization rarely is
- Thread interaction needs careful thought
  - But not too clever
  - Don’t want to have to think to hard about reordering
    - If you don’t have data races, you don’t have to think about the weird things the compiler is allowed to do

Wrap-Up—Communication

- Communication between threads
  - Requires a happens-before edge/ordering
  - Both threads must participate
  - No way for one thread to push information into other threads
    - Final fields allow some guaranteed communications without a normal happens-before edge, but don't write code that depends on this for normal operations

For More Information

- http://www.cs.umd.edu/~pugh/java/memoryModel/
- Concurrency Utilities (JSR 166) Interest mailing list
- TS-4915: Concurrency Utilities
- Java Concurrency in Practice
  - By Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, Doug Lea


JAVA
CONCURRENCY
IN PRACTICE
TIM PEIERLS | BRIAN GOETZ | JOSHUA BLOCH
JOSEPH BOWBEER | DOUG LEA | DAVID HOLMES

Q&A

Jeremy Manson

Purdue University

William Pugh

Univ. of Maryland

The Java™ Technology Memory Model: the Building Block of Concurrency

Jeremy Manson, Purdue University
William Pugh, Univ. of Maryland

http://www.cs.umd.edu/~pugh/java/memoryModel/

TS-1630