Threads and Java Memory Model

Oleg Šelajev
@shelajev

2017, Tallinn

- Homework is an individual assignment
  - Copying work is forbidden
  - Sharing work is frown upon


What if I never
find out who's a
good boy?

Agenda

- Threads
- Java Memory Model

Concurrency

- Concurrency - several computations are executing simultaneously, potentially interacting with each other

Why do we care?

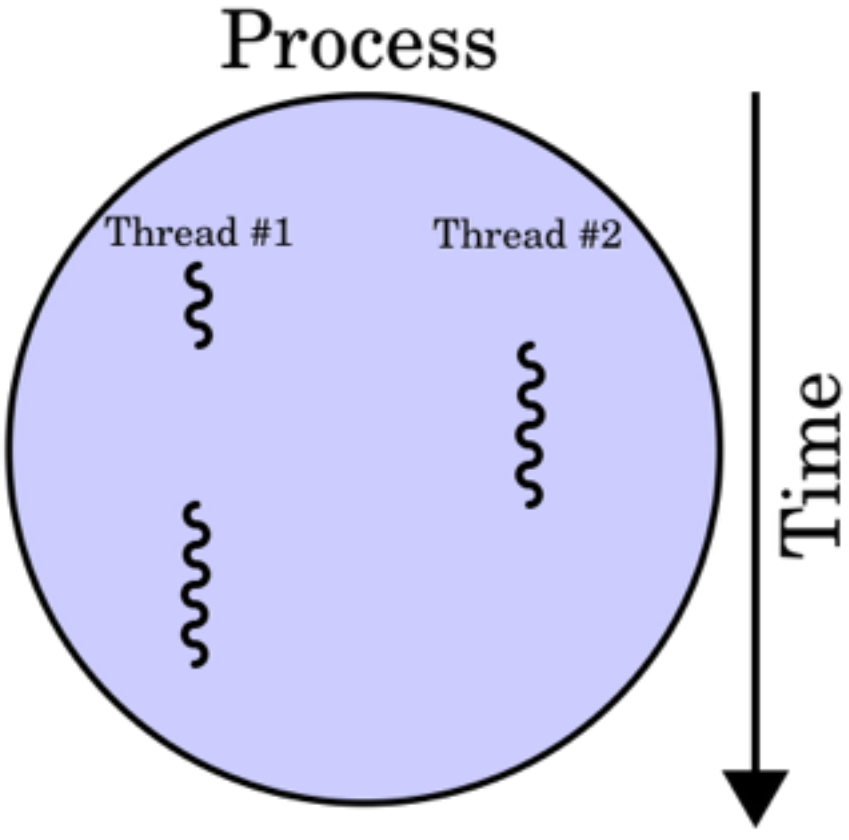
https://twitter.com/reubenbond/status/662061791497744384

Process

- Process - an instance of a computer program that is being executed
- Isolated, independently executing programs
- Communicate through sockets, signals, files, shared memory etc.

Thread

- Lightweight process
- Multiple streams of control flow coexist within a process
- Has its own program counter, stack, and local variables
- Share memory, file handles, sockets etc.


Process
Thread #1
Thread #2
Time

Why threads

- Simplicity of modelling
- Handling asynchronous events
  - via blocking calls
- Resource utilisation

Resource utilisation

Starting a thread

```
new Thread() {
   public void run() {
      System.out.println("foo");
   }
}.start();
```

Starting a thread 2

```
new Thread(
   new Runnable() {
      public void run() {
         System.out.println("bar");
      }
   }
).start();
```

Waiting for a thread to complete

```
Thread t = new Thread() {
  public void run() {
    for (int i = 0; i < 1000000000; i++);
    System.out.println("ready");
  }
};
System.out.println("start");
t.start();
// current thread waits for t to complete
t.join();
System.out.println("end");
```

Pausing execution with sleep()

```
long start = System.currentTimeMillis();
Thread.sleep(100);
long end = System.currentTimeMillis();
System.out.println("done " + (end - start));
```

Daemon threads

```
Thread t = new Thread() {
  public void run() {
    for (int i = 0; i < 1000000000; i++);
    System.out.println("ready"); // never
  printed
   }
};
System.out.println("start");
t.setDaemon(true); // won't prevent stopping
t.start();
System.out.println("end");
```

Stopping a thread

```
final AtomicBoolean ready = new AtomicBoolean();
Thread t = new Thread() {
  public void run() {
    while (!ready.get());
    System.out.println("ready");
  }
};
System.out.println("start");
t.start();
ready.set(true);
t.join();
System.out.println("end");
```

Interrupting a thread

```
Thread t = new Thread() {
  public void run() {
    System.out.println("thread start");
    try {
      Thread.sleep(Long.MAX_VALUE);
    } catch (InterruptedException e) {
      System.out.println("thread interrupted");
    }
  }
};
System.out.println("start");
t.start(); Thread.sleep(100);
t.interrupt(); t.join();
System.out.println("end");
```

Interrupting a thread 2

```
Thread t = new Thread() {
  public void run() {
    System.out.println("thread start");
    while (true);
  }
};
System.out.println("start");
t.start();
Thread.sleep(100);
t.interrupt();
t.join();
System.out.println("end");
```

Interrupting a thread 3

```
Thread t = new Thread() {
  public void run() {
    System.out.println("thread start");
    while (!isInterrupted());
  }
};
System.out.println("start");
t.start();
Thread.sleep(100);
t.interrupt();
t.join();
System.out.println("end");
```

Deprecated methods

- stop()
- stop(Throwable t)
- suspend()
- resume()
- http://docs.oracle.com/javase/1.5.0/docs/guide/misc/threadPrimitiveDeprecation.html

Race condition

- A race condition occurs when the correctness of a computation depends on the relative timing or interleaving of multiple threads by the runtime; in other words, when getting the right answer relies on lucky timing.

Race condition example

```
public class UnsafeSequence {
   private int value;

   public int getNext() {
      return value++;
   }
}
```

Synchronization

- **`synchronized`** keyword
- Every object has an intrinsic lock - „monitor“
- Automatically acquired and released by the executing thread in **`synchronized`** block
- Mutually exclusive
- Reentrant

Synchronization

- Synchronized block locks on object
- Instance method locks on **this** object
- Static method locks on **Class** object

Synchronization

```
synchronized (foo) {
  // at most one thread is
  // executing this block
}
synchronized void foo() {}
static synchronized void bar() {}
```

Thread safety

- In absence of sufficient synchronization the ordering of operations in multiple threads is **unpredictable**
- A class is thread-safe when it continues to behave correctly when accessed from multiple threads with no additional synchronization or other coordination on the part of the calling code

Thread safe counter

```
public class Sequence {
   private int value;

   public synchronized int getNext() {
      return value++;
   }
}
```

Atomicity

- Operations A and B are atomic with respect to each other if, from the perspective of a thread executing A, when another thread executes B, either all of B has executed or none of it has.
- An atomic operation is one that is atomic with respect to all other operations, including itself, that operate on the same state.

Wait & notify

- java.lang.Object methods
- wait – wait for a condition
- notify – signal condition
- notifyAll - signal to all
- can only be invoked when holding object monitor
- spurious wakeup

Wait & notify 2

```
final AtomicBoolean ready = new AtomicBoolean();
Thread t = new Thread() {
  public void run() {
    try {
      synchronized (ready) {
        while (!ready.get()) // check condition
          ready.wait(); // wait to be notified
      }
      System.out.println("ready");
    } catch (InterruptedException e) {
      System.out.println("interrupted");
    }}};
System.out.println("start"); t.start();
synchronized (ready) {
  ready.set(true);
  ready.notifyAll(); // wake up waiting threads
}
t.join(); System.out.println("end");
```

Summary

- Starting & stopping threads
- Synchronization
- Wait & notify

Java Memory Model

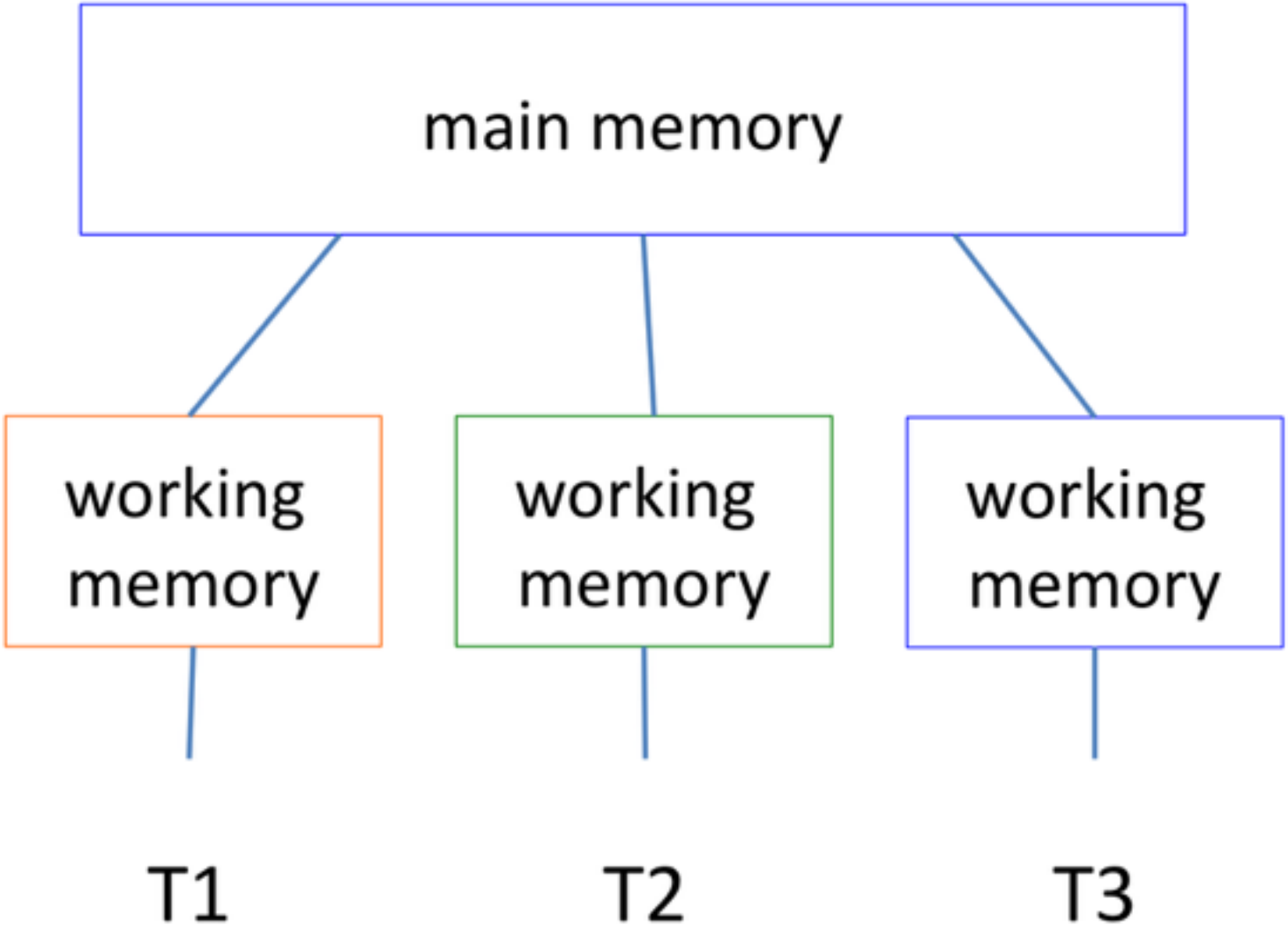
- What? Why?
- Clear and easy to understand
- Specifies minimal guarantees given by JVM
- Reliable multithreaded code
- Allows for high performance JVMs

Java Memory Model

What values can given read instructions see at a given time?

Key principles

- All threads share the main memory
- Each thread uses a local working memory
- Flushing or refreshing working memory to/from main memory must comply to JMM rules


main memory
working memory
working memory
working memory
T1
T2
T3

Safety issues in multithreaded systems

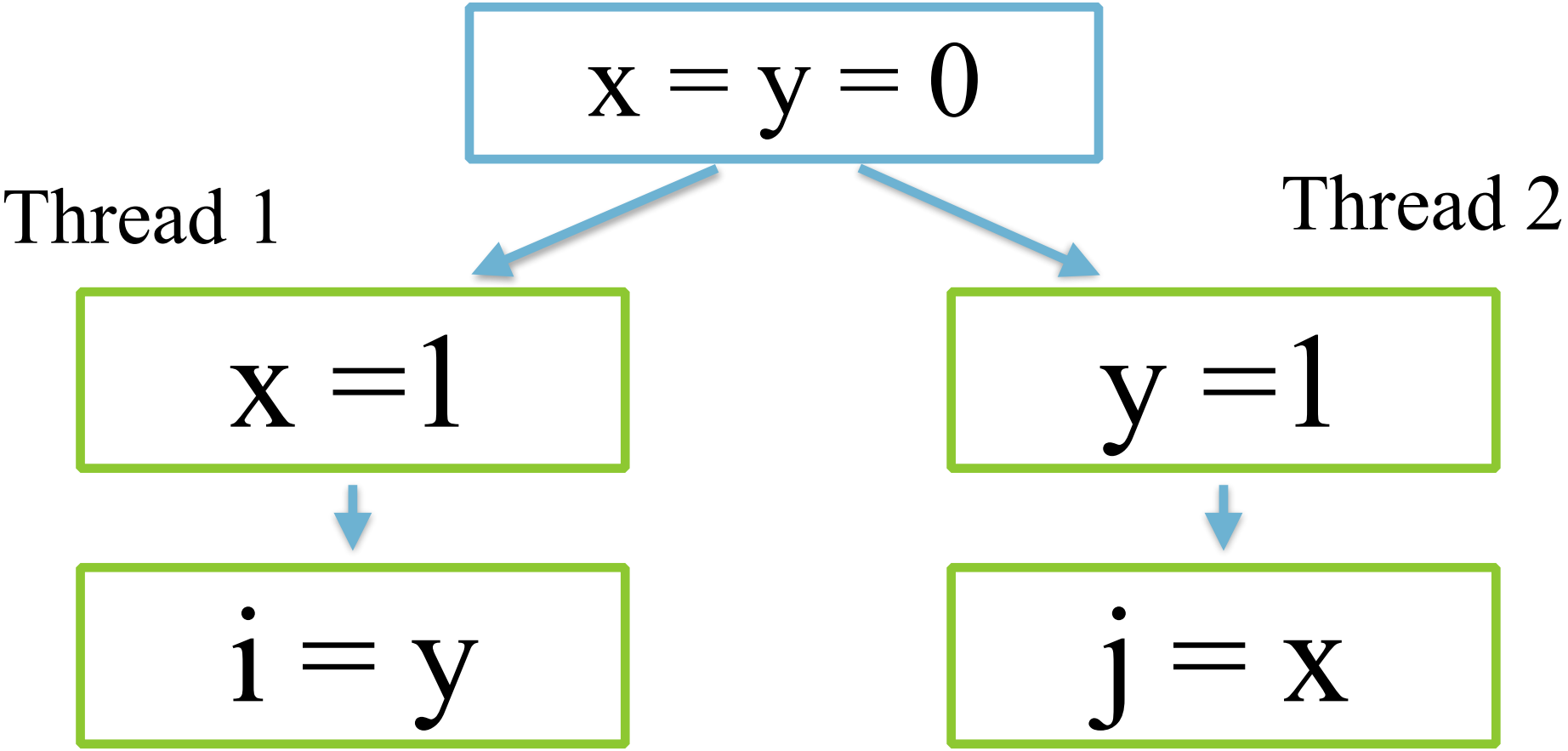
- Many intuitive assumptions do not hold
- Can't depend on testing to check for errors
- Some anomalies will occur only on some platforms
- Anomalies occur rarely and non-repeatedly

Is this code correct?

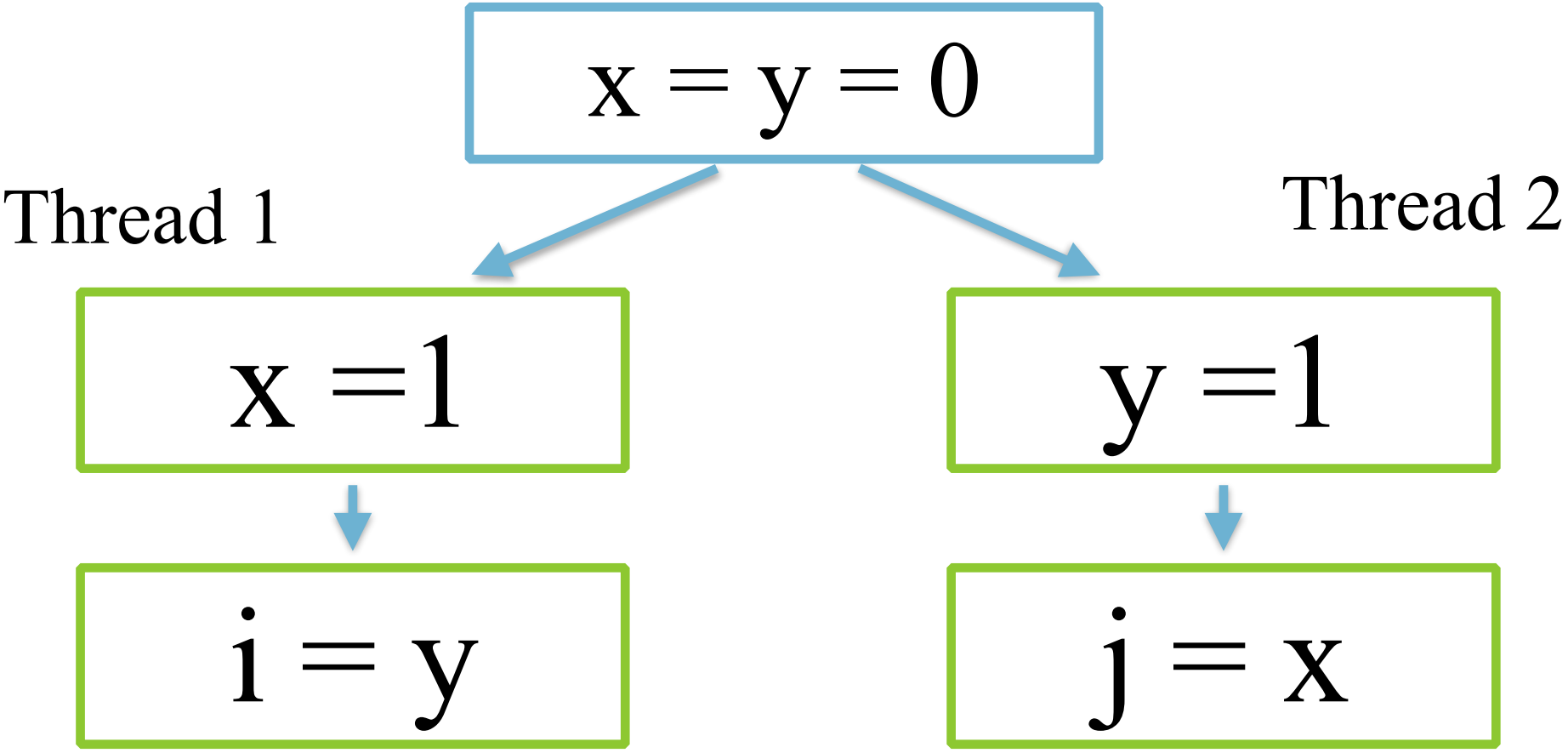
```
boolean ready = false;

Thread t = new Thread() {
  public void run() {
    while (!ready);
    System.out.println("ready");
  }
};
System.out.println("start");
t.start();
ready = true;
t.join();
System.out.println("end");
```

Synchronization is needed for ***mutual exclusion*** and ***visibility***

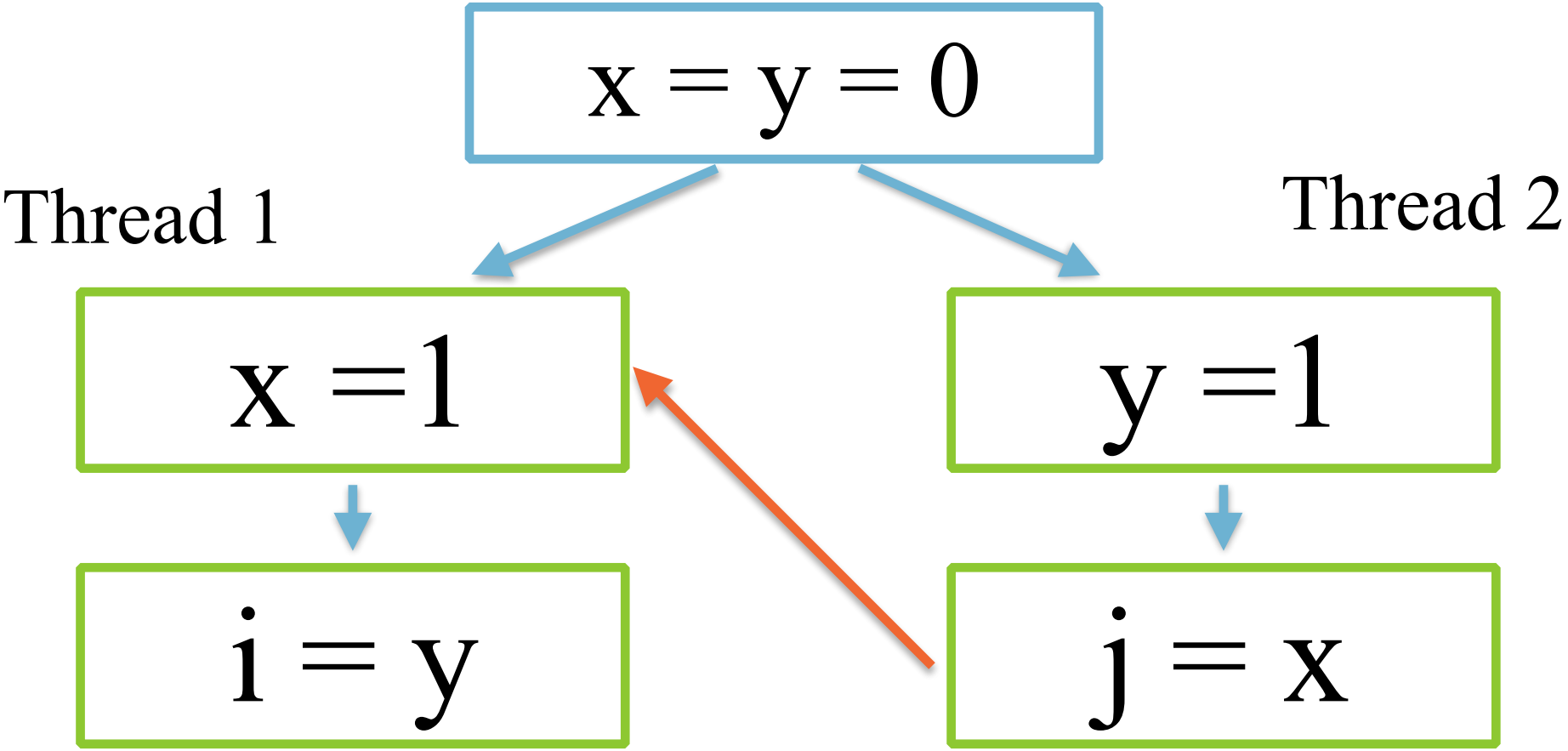
- Synchronization isn't just about mutual exclusion and blocking
- It also regulates when other threads *must* see writes by other threads
  - When writes become visible
- Without synchronization, compiler and processor are allowed to reorder memory accesses in ways that may surprise you
  - And break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



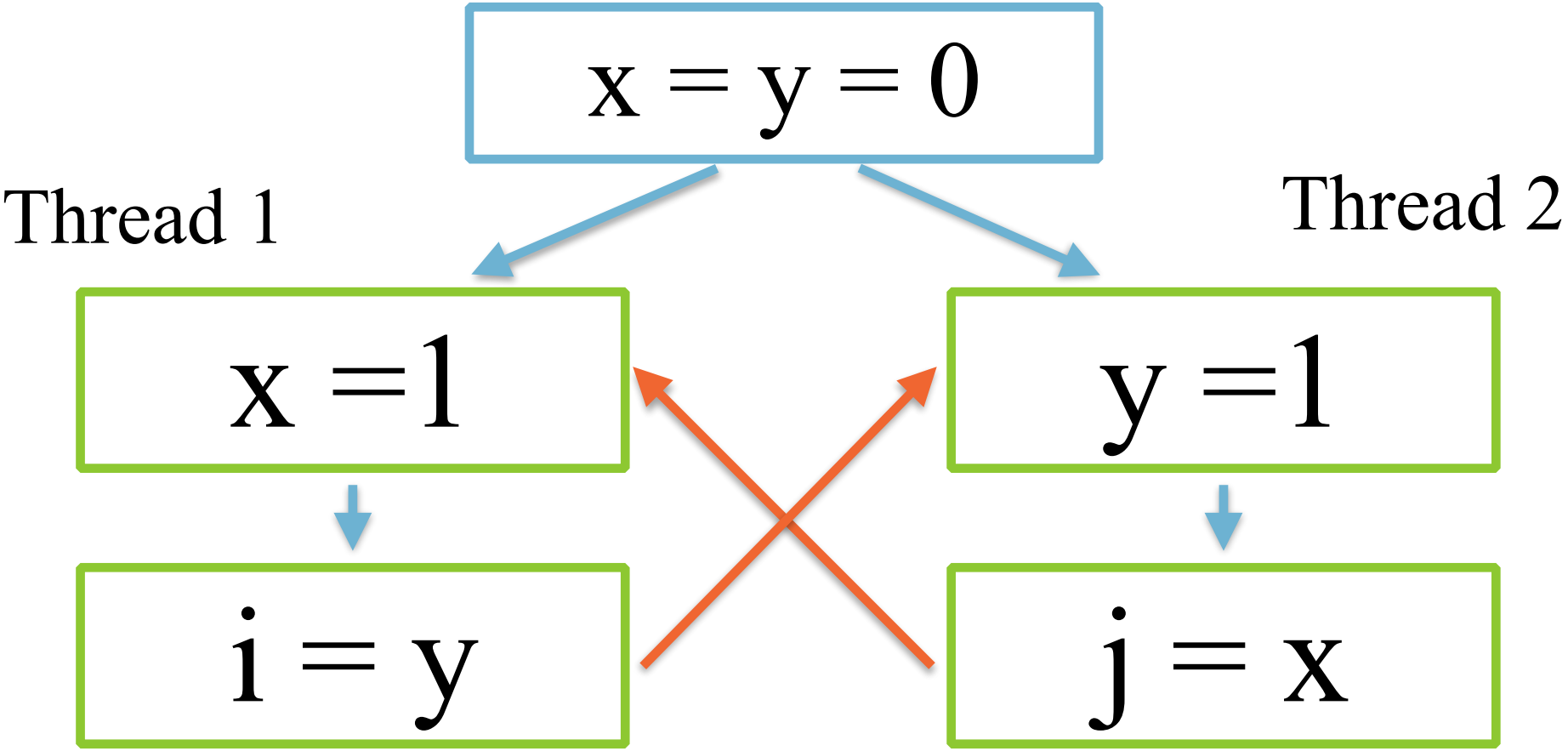
Is values of i and j are



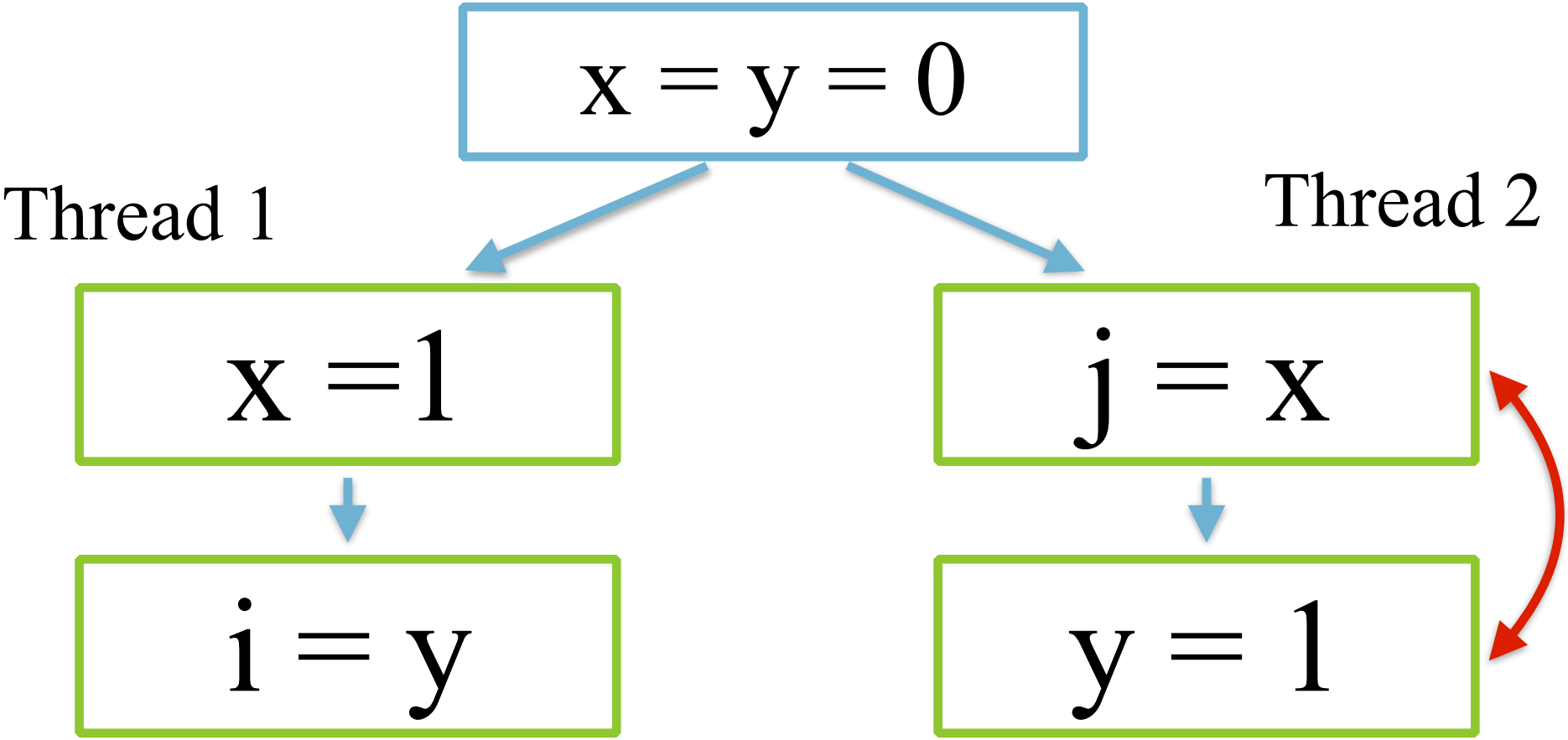
Can we observe **i = 0** and **j =**



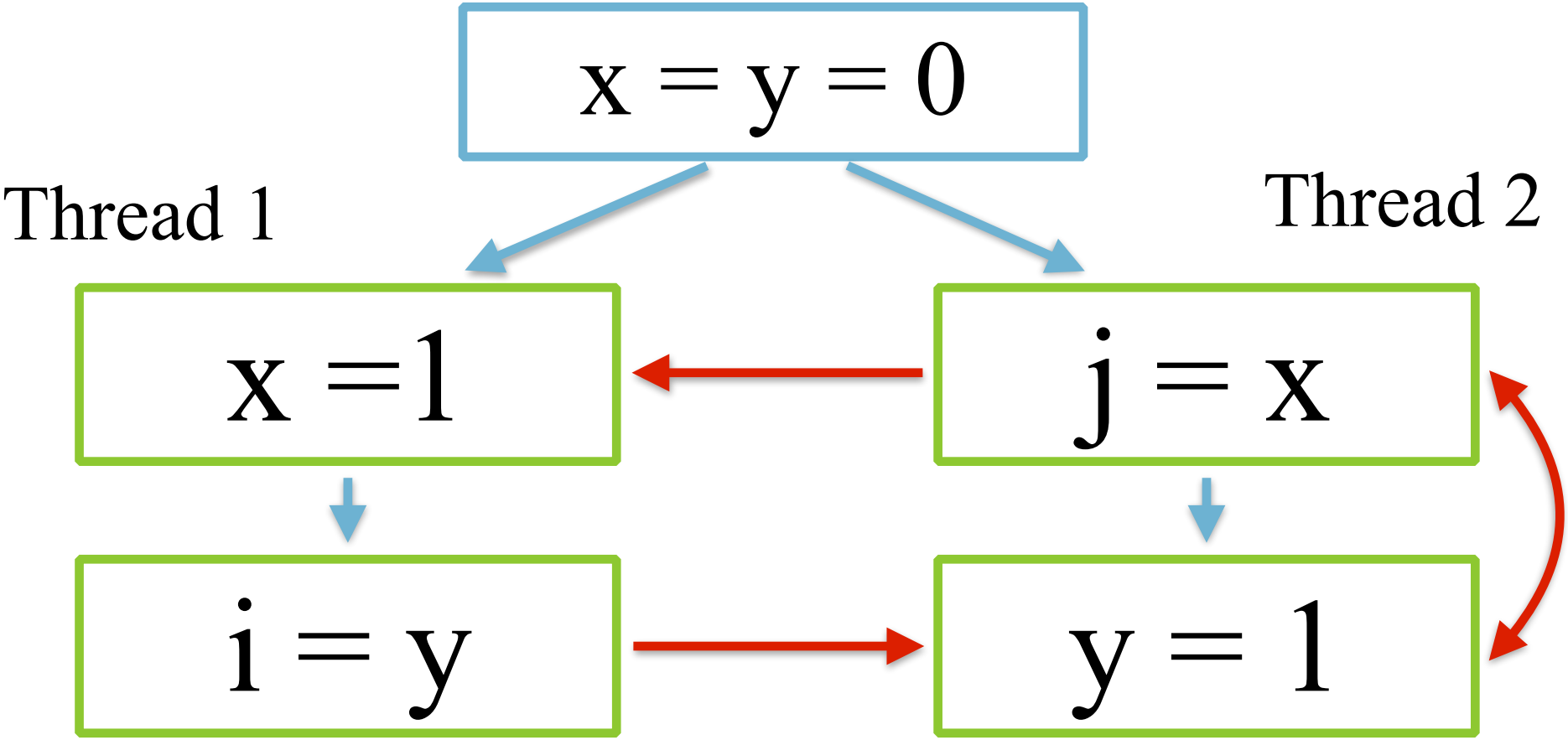
Can we observe **i = 0** and **j =**



Can we observe **i = 0** and **j =**



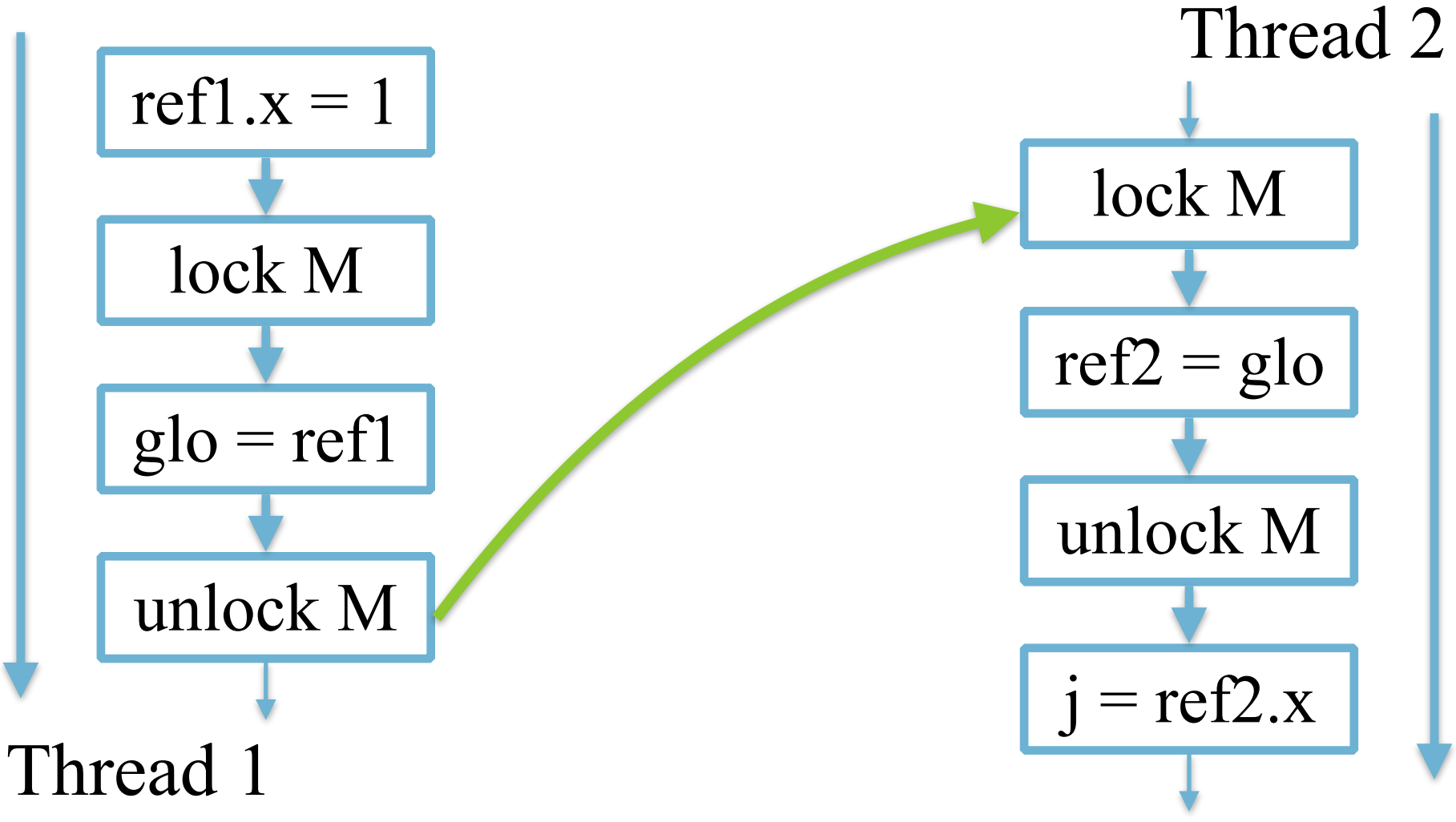
Can we observe **i = 0** and **j =**

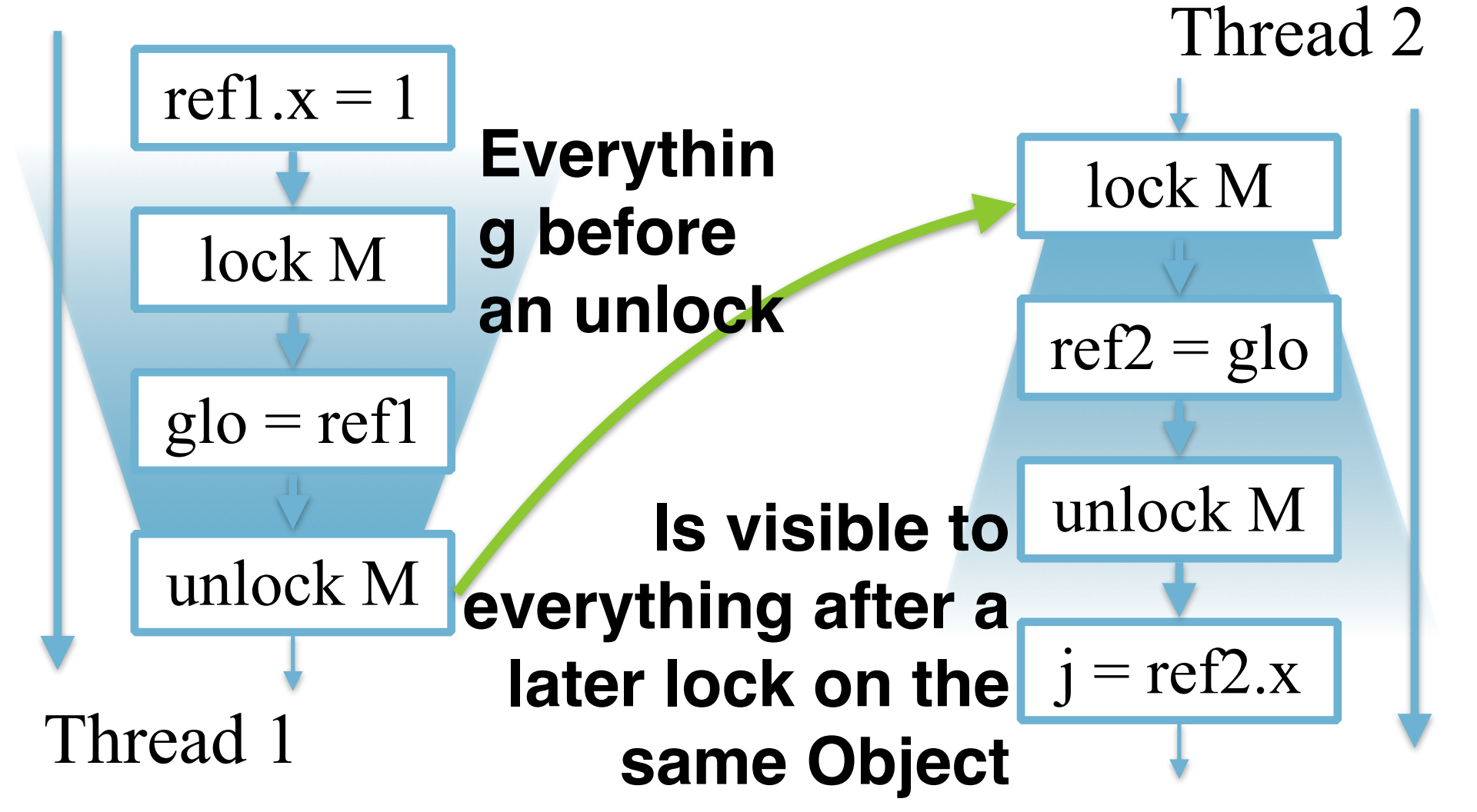


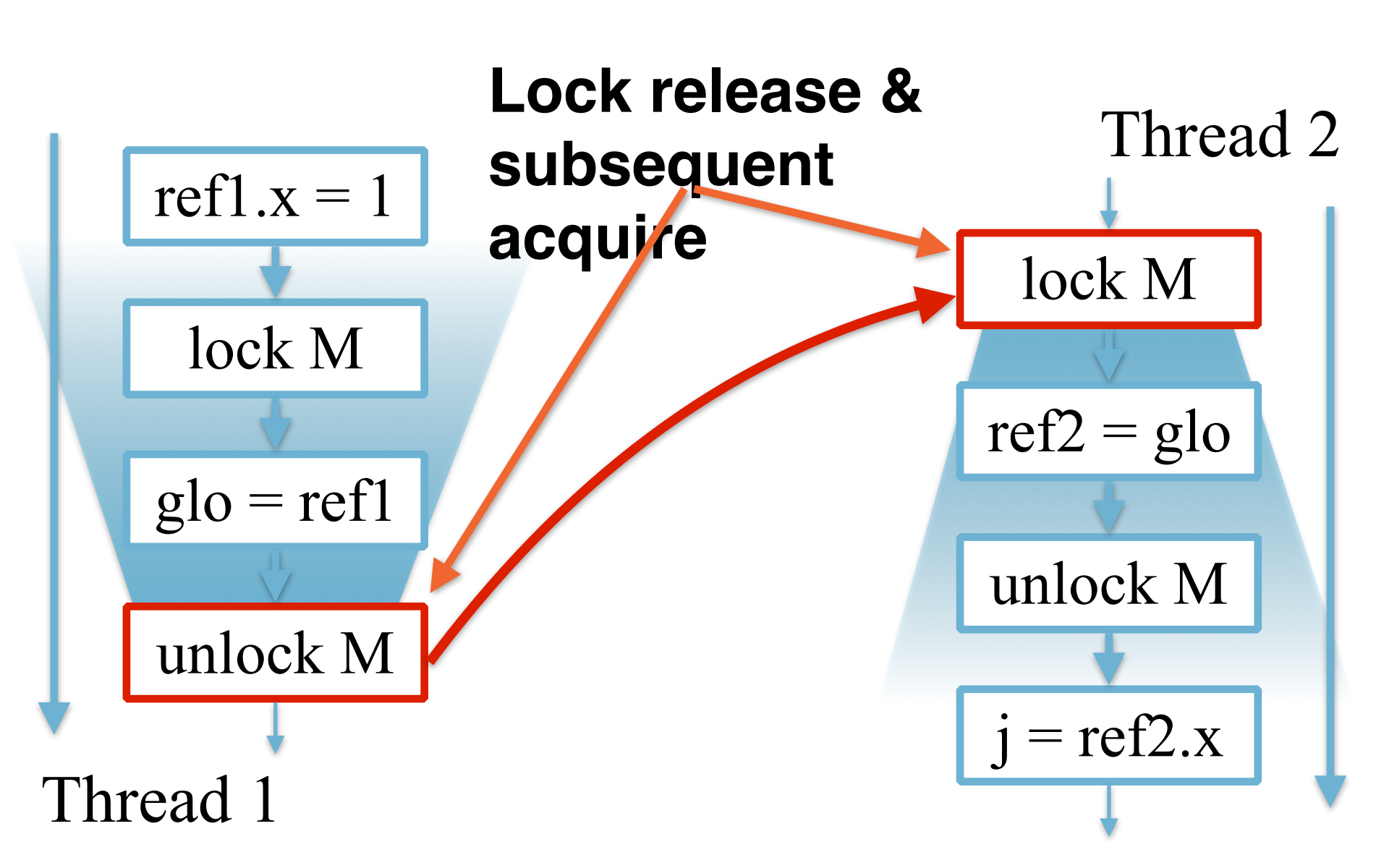
Can we observe **i = 0** and **j =**

How can this happen?

- Compiler can reorder statements
  - Or keep values in registers
- Processor can reorder statements
- On multi-processor, values not synchronized to global memory
- Memory model is designed to allow aggressive optimisation


Thread 2
ref1.x = 1
lock M
glo = ref1
unlock M
lock M
ref2 = glo
unlock M
j = ref2.x
Thread 1


Thread 2
ref1.x = 1
lock M
glo = ref1
unlock M
Everythin
g before
an unlock
lock M
ref2 = glo
unlock M
j = ref2.x
Is visible to
everything after a
later lock on the
same Object
Thread 1


Lock release & subsequent acquire
Thread 2
ref1.x = 1
lock M
glo = ref1
unlock M
lock M
ref2 = glo
unlock M
j = ref2.x
Thread 1

Release and acquire

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

Volatile fields

- If a field could be accessed by multiple threads, and at least one of those is a write, then:
  - Use locking to prevent simultaneous access, or
  - Make the field **volatile**

```
volatile boolean ready;
```

What does volatile do?

- Reads & writes go directly to memory
  - Caching disabled
- Volatile longs & doubles are atomic
- Volatiles reads/writes cannot be reordered
- Reads/writes become acquire/release pairs

http://www.infoq.com/presentations/Do-You-Really-Get-Memory

Stopping a thread 2

```
volatile boolean ready = false;

Thread t = new Thread() {
  public void run() {
    while (!ready);
    System.out.println("ready");
  }
};
System.out.println("start");
t.start();
ready = true;
t.join();
System.out.println("end");
```

Another volatile example

```
private volatile boolean ready;
private Object data;
public Object get() {
  if (!ready) return null;
  return data;
}
public synchronized void set(Object o) {
  if (ready) throw new IllegalStateException();
  data = o;
  ready = true;
}
```

Volatile arrays?

- volatile A[] array;
- volatile – not transitive:
  - … = array; // volatile read
  - array = … // volatile write
  - **array[i] =… // non-volatile write**
- java.util.concurrent:
  - AtomicIntegerArray, AtomicLongArray, AtomicReferenceArray

Happens before ordering

- With a single thread all is simple
- A release and a matching later acquire establish a happens-before relationship

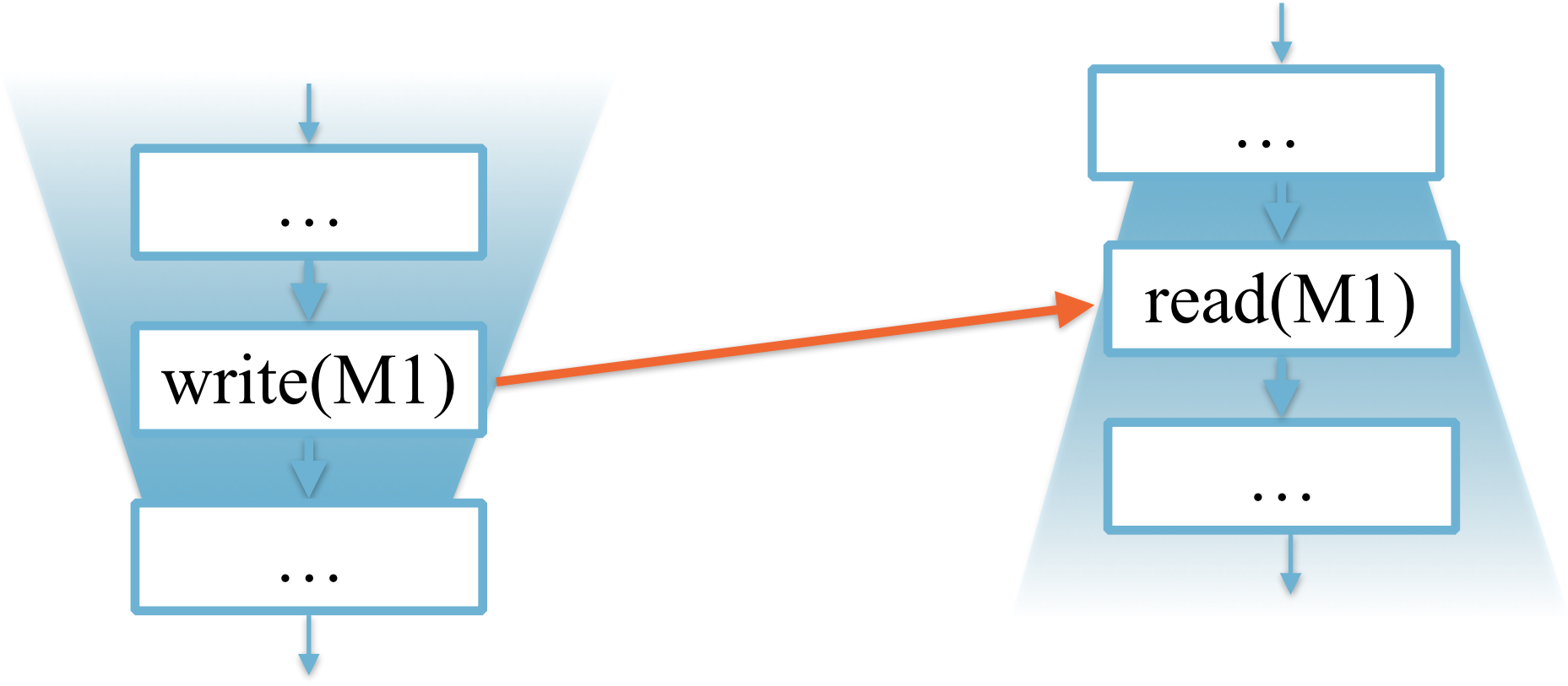
Happens before ordering 2

- **Program order rule**. Each action in a thread *happens-before* every action in that thread that comes later in the program order. (JLS 17.4.3)
- **Monitor lock rule**. An unlock on a monitor lock *happens-before* every subsequent lock on that same monitor lock. (JLS 17.4.4)
- **Volatile variable rule**. A write to a volatile field *happens-before* every subsequent read of that same field. (JLS 17.4.5)

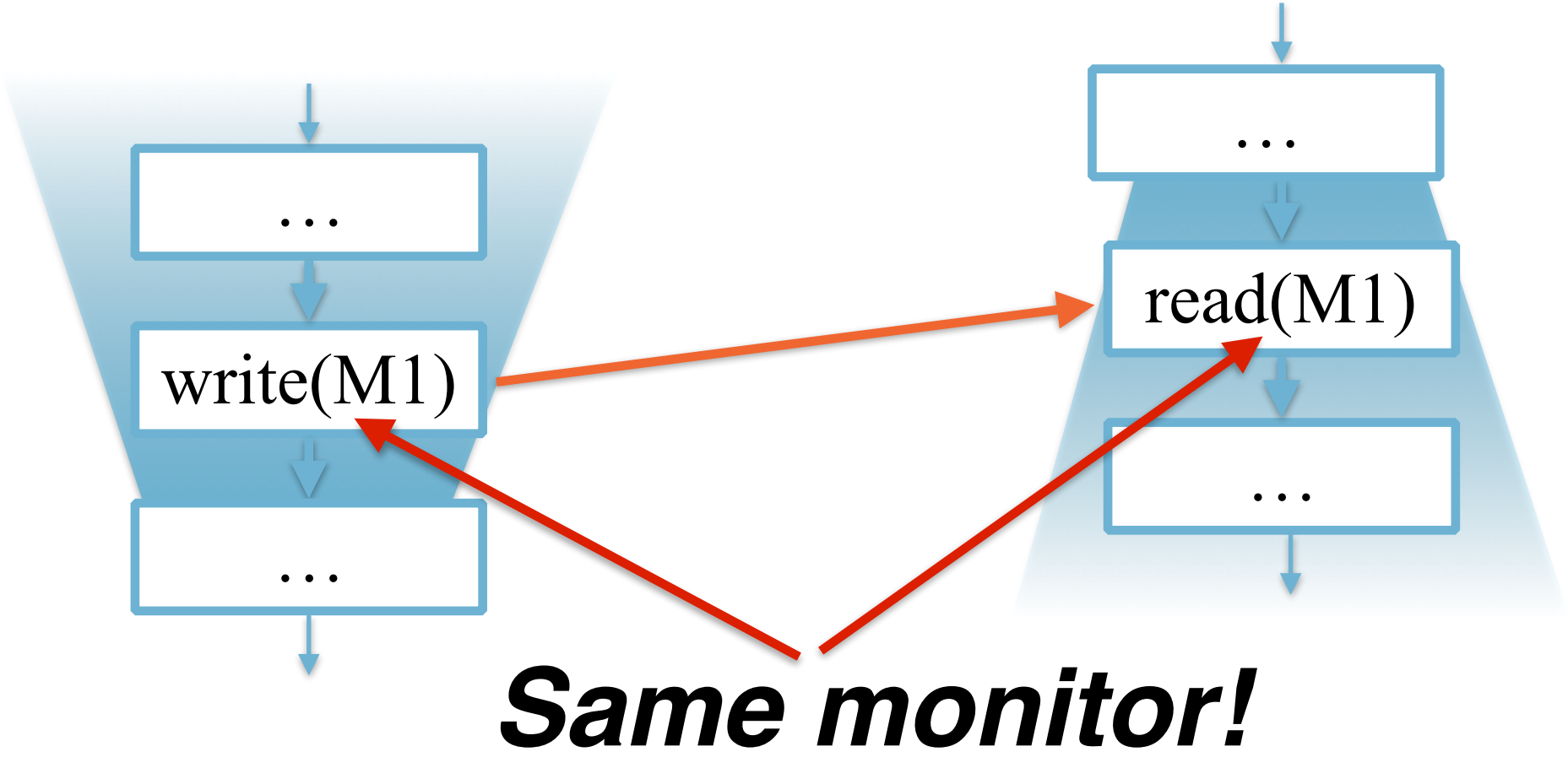
Happens before ordering 3

- **Thread start rule.** A call to Thread.start on a thread *happens-before* any other thread detects that thread has terminated, either by successfully return from Thread.join or by Thread.isAlive returning false.
- **Interruption rule.** A thread calling interrupt on another thread *happens-before* the interrupted thread detects the interrupt (either by having InterruptedException thrown, or invoking isInterrupted() or interrupted()). (JLS 17.2.3)
- **Finalizer rule.** The end of a constructor for an object *happens-before* the start of the finalizer for that object. (JLS 17.4.5)
- **Transitivity.** If A *happens-before* B, and B *happens-before* C, then A *happens-before* C. (JLS 17.4.5)

Visibility between threads



Visibility between threads



Same monitor!

Data race

- If there are two accesses to memory location,
  - At least one of those is a *write*, and
  - The memory location *is not volatile*, then

The access must be ordered by ***happens-before***

Special semantics of *final* fields

```
class A {
  final B ref;
  public A (…) {
     this.ref = … ;
  }
}
```

Special semantics of *final* fields

```
class A {
  final B ref;
  public A (…) {
    this.ref = … ;
  }
}
```

Once constructor completes, any thread can read values written to the *final* field

- and the whole object tree starting from the field

One more time!

Java Memory Model

- Variables: *fields*
- Operations:
  - R/W of instance fields (read/write)
  - R/W of volatile fields (volatile read/write)
  - Synchronization (lock/unlock)

Java Memory Model

- Atomicity
- Visibility
- Final fields semantics
- Reordering

Atomicity

- Read/write operations are atomic
- No out of thin-air values:
  - Any variable read operation should return either a default value, or the value that was assigned to this variable (somewhere else)

Atomicity

- Exception:
  - It is allowed that reads/writes of long/ double type is not atomic, but..
  - … read/write of **volatile** long/double **must** be atomic

Atomicity

- A common mistake:
  - For volatile long/double only the reads and writes are atomic
  - foo++, foo-- ***are not atomic!***
- Solution:
  - *synchronized*
  - *java.util.concurrent.atomic*

Visibility

- Again, the ***happens-before*** relation:
  - If X happens-before Y, then operation X is executed before and Y will will see the result of X

Visibility guarantees

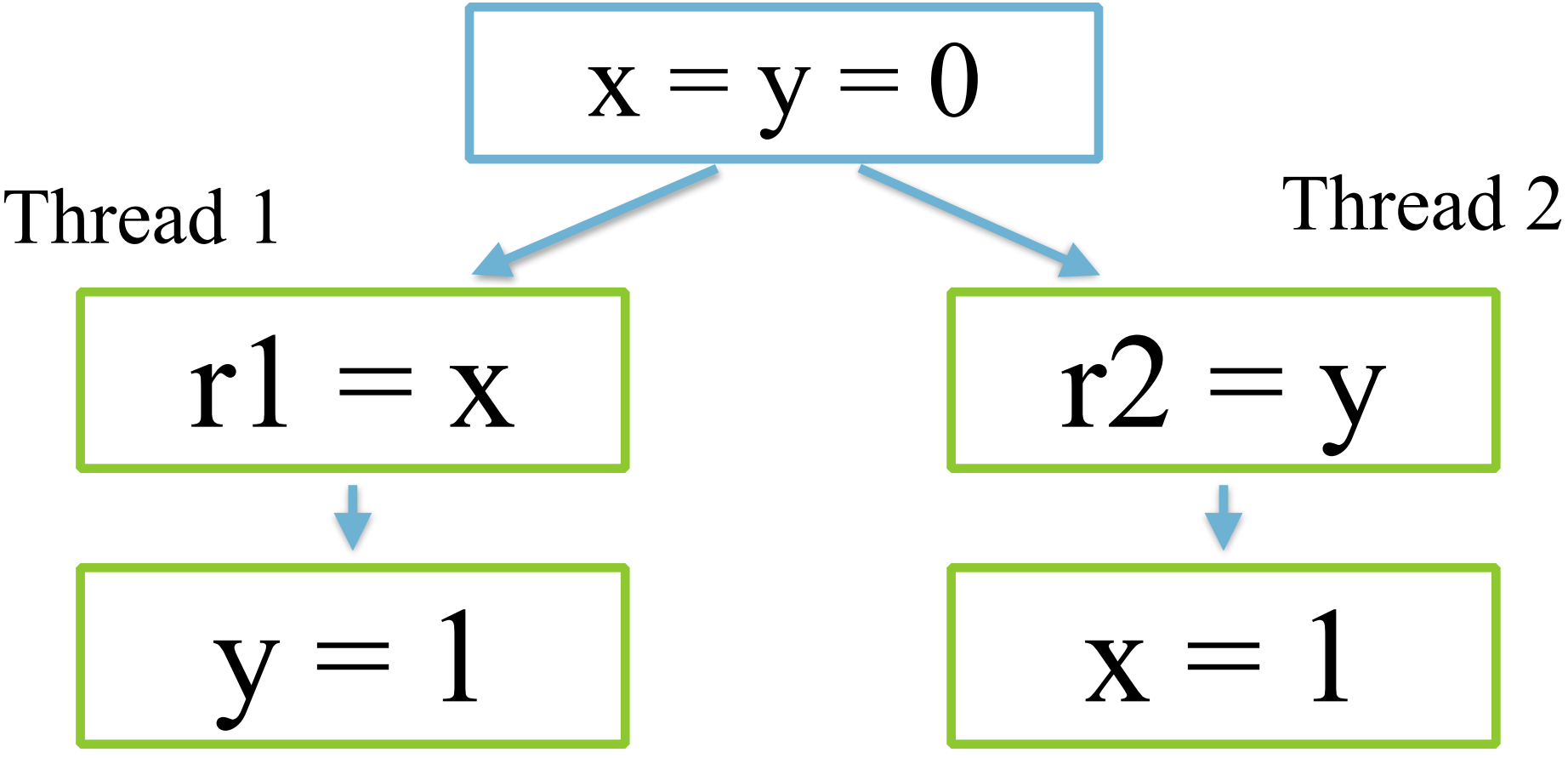
- Changes made in one thread are guaranteed to be visible to other threads under following conditions:
  - Explicit synchronization
  - Thread start and termination
  - Read/write of volatiles
  - First read of finals
- Thus, visibility is an issue in case if the access is *not synchronized*

Ordering

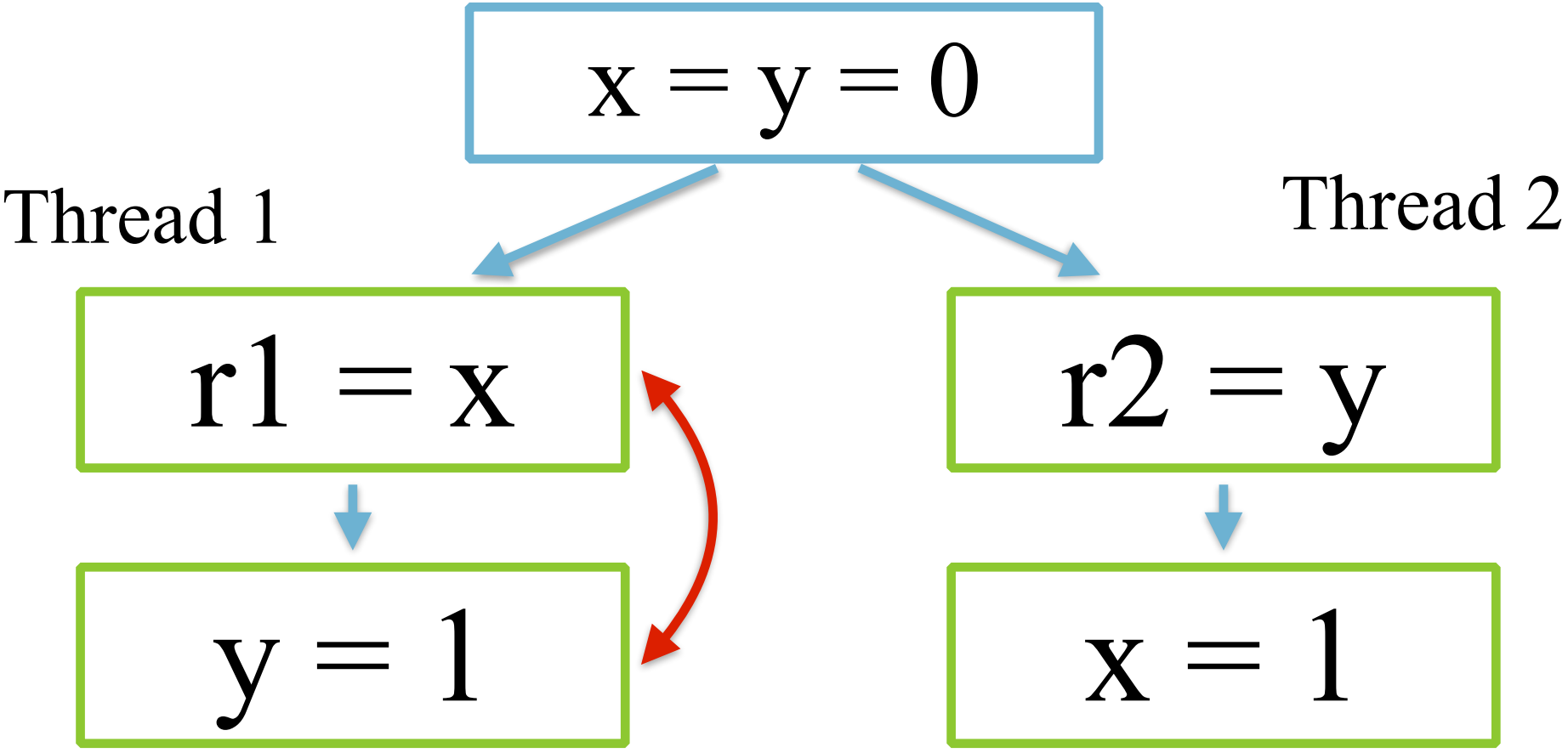
- Within a thread
  - Program order, as-if-sequential execution
  - Reordering is possible as long as currently executing thread cannot tell the difference (data dependencies)

Ordering Guarantees

- Ordering of synchronized blocks is preserved
  - Actions in one synchronized block happen before thread acquires the same lock
- Ordering of read/write of volatile fields is preserved
  - Effect of last write to volatile is visible to all subsequent reads of the same volatile
- Ordering of initialization/access of final fields is preserved
  - All threads will see the correct values of final fields that where set by the constructor



Can we observe **r1 = 1** and **r2 = 1**?



Can we observe **r1 = 1** and **r2 = 1**?

Yes, if T1's actions are reordered

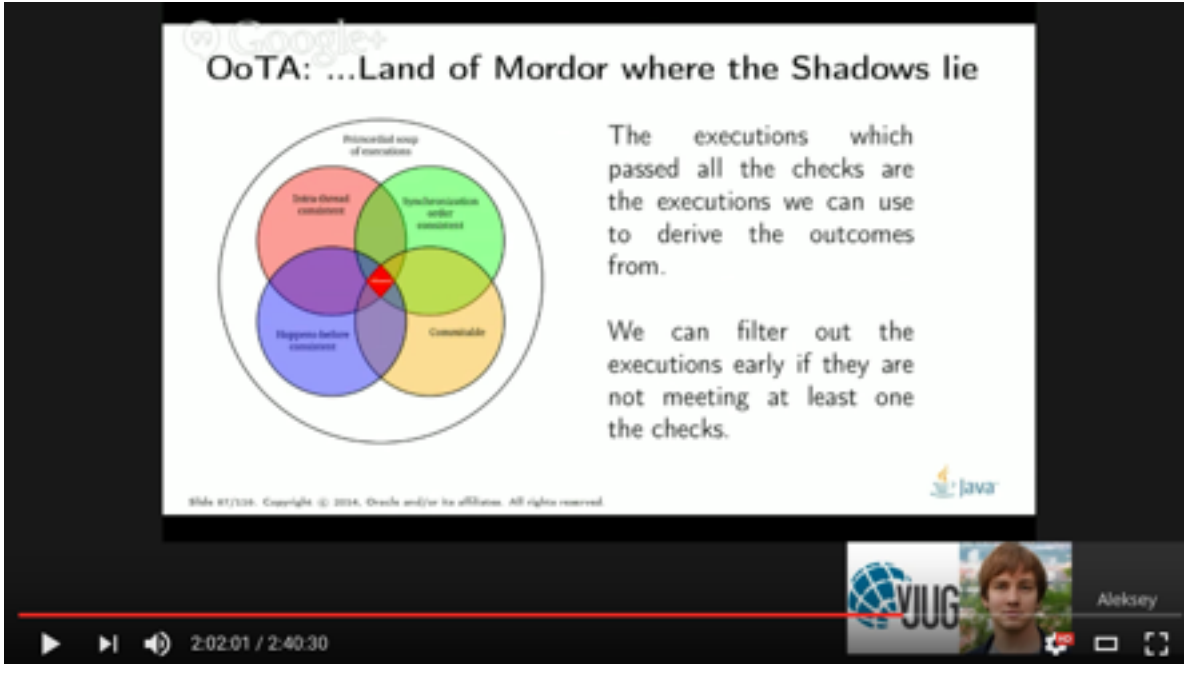
References / Reading

Java Language Specification, Chapter 17:

http://docs.oracle.com/javase/specs/jls/se8/html/jls-17.html

References / Watching

Java Memory Model Pragmatics



https://www.youtube.com/watch?v=TxqsKzxyySo

Reading again

Aleksey Shipilëv
@shipilev FOLLOWS YOU

JVM || Performance Geek, Benchmarking Tzar, Concurrency Underground Dweller. Developing Java and OpenJDK at Red Hat. Opinions are (excellent and) mine only.

- Java Memory Model Pragmatics (transcript)
  - https://shipilev.net/blog/2014/jmm-pragmatics/
- Close Encounters of The Java Memory Model Kind
  - https://shipilev.net/blog/2016/close-encounters-of-jmm-kind/

Homework

Clone
https://github.com/JavaFundamentalsZT/jf-hw-threads-jmm

Create the tests described below

Check the comments in the Java file to get more hints at what you need to change!
Submit using the normal jf-skeleton procedure (see README.md)

Homework: JFHW7E1.java

Create the concurrency test:

Shared memory: int a; int b

Thread 1: b = 1; x = a;
Thread 2: a = 1; y = b;

Question: what values of x, y can be observed at the end?

Homework: JFHW7E2.java

Create the concurrency test:

Shared memory: an instance of java.util.BitSet

Thread 1: sets 0th bit of the BitSet to true
Thread 2: sets 1st bit of the BitSet to true

Question: what values of 0th and 1st bits in the bitset can be observed at the end?

Homework: JFHW7E3.java

Come up with a description of the test that shows non-trivial executions.
If a result of the test shows interesting reorderings, the properties of the volatiles, or something you think is worth showing.

Create a test and explain it.

Homework: Explain the results

Find the results in the “results” folder.
Copy the relevant section into the javadoc of the test class.

Explain why these results happen! Be concise, but show that you understand it.