COMP 322: Fundamentals of Parallel Programming

Lecture 32: Volatile variables, Java memory model

Vivek Sarkar
Department of Computer Science, Rice University
vsarkar@rice.edu

https://wiki.rice.edu/confluence/display/PARPROG/COMP322

Acknowledgments for Today's Lecture

- CMU 15-440 course: Distributed Systems, Fall 2011, David Andersen, Randy Bryant
  - —Lecture on "Time and Synchronization"
    - http://www.cs.cmu.edu/~dga/15-440/F11/lectures/09-time+synch.ppt
- "The Java Memory Model", Jeremy Manson
  - —http://dl.dropbox.com/u/1011627/google_jmm.pdf
- "Introduction to Concurrent Programming in Java", Joe Bowbeer, David Holmes, OOPSLA 2007 tutorial slides
  - —Contributing authors: Doug Lea, Brian Goetz
- "Engineering Fine-Grained Parallelism Support for Java 7", Doug Lea, July 2010
- "Java Concurrency in Practice", Brian Goetz with Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes and Doug Lea. Addison-Wesley, 2006.

Outline

- <u>Volatile variables</u>

- Java Memory Model

Memory Visibility

- <u>Basic question</u>: if a memory location L is written by statement S1 in thread T1, when is that write guaranteed to be visible to a read of L in statement S2 of thread T2?
- <u>HJ answer</u>: whenever there is a directed path of edges from S1 in S2 in the computation graph
  - Computation graph edges are defined by semantics of parallel constructs: async, finish, async-await, futures, phasers, isolated, object-based isolation
- <u>Java answer</u>: whenever there is a "happens-before" relation between S1 and S2

  ==> Should we define "happens-before" using time or ordering?
  - Is there such a thing as universal global time?

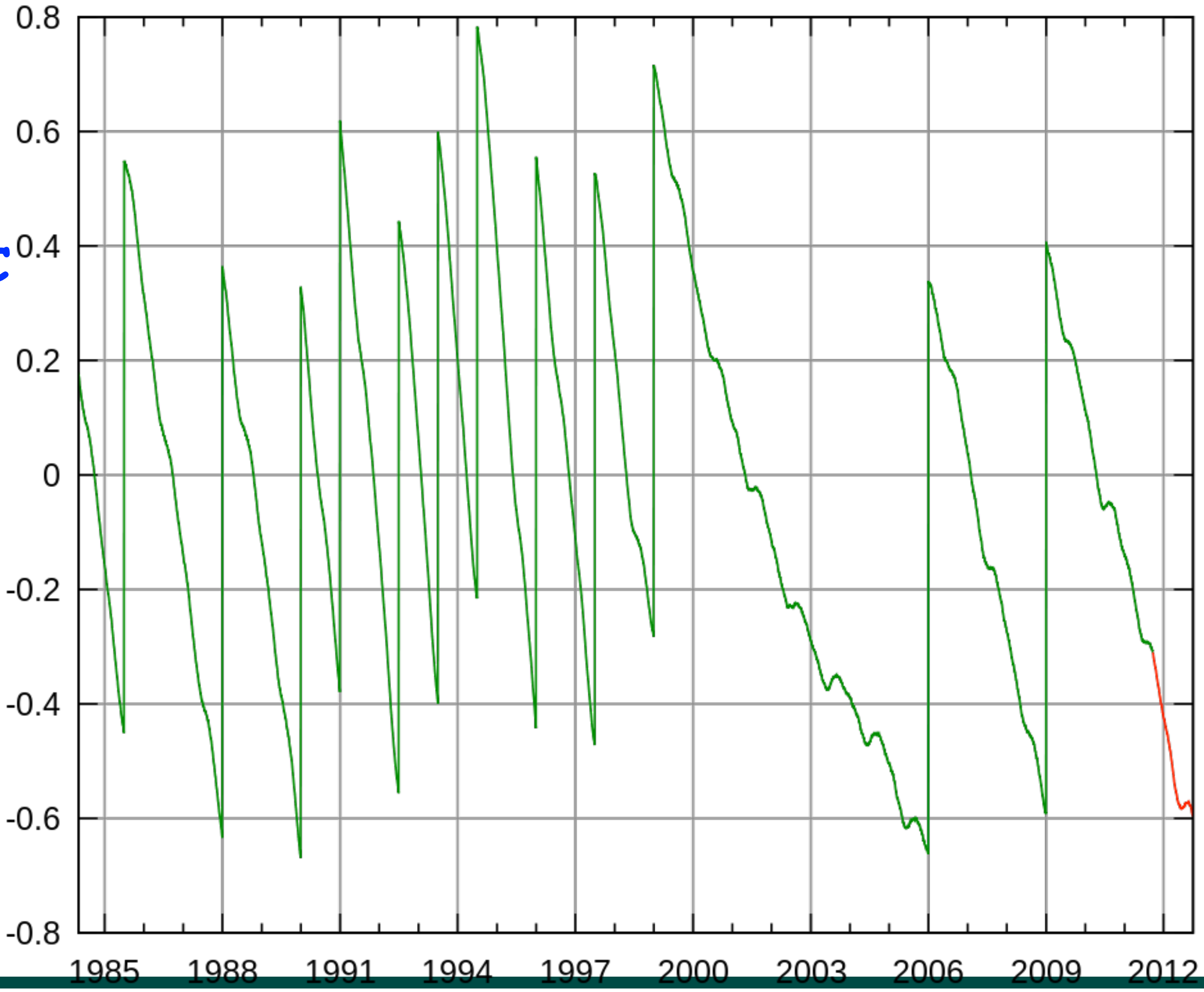
Physical Clocks

Different clocks can be closely synchronized, but never perfect e.g.,

- UT1
  - Based on astronomical observations
  - "Greenwich Mean Time"
- TAI
  - Started Jan 1, 1958
  - Each second is 9,192,631,770 cycles of radiation emitted by Cesium atom
  - Has diverged from UT1 due to slowing of earth's rotation
- UTC
  - TAI + leap seconds to be within 800ms of UT1
  - Currently 34

Comparing Time Standards

UT1 – UTC



Distributed time

- Premise
  - The notion of time is well-defined (and measurable) at each single location
  - But the relationship between time at different locations is unclear
    - Can minimize discrepancies, but never eliminate them

A baseball example

- Four locations: pitcher's mound, first base, home plate, and third base
- Ten events:

  e_1: pitcher throws ball to home

  e_2: ball arrives at home

  e_3: batter hits ball to pitcher

  e_4: batter runs to first base

  e_5: runner runs to home

  e_6: ball arrives at pitcher

  e_7: pitcher throws ball to first base

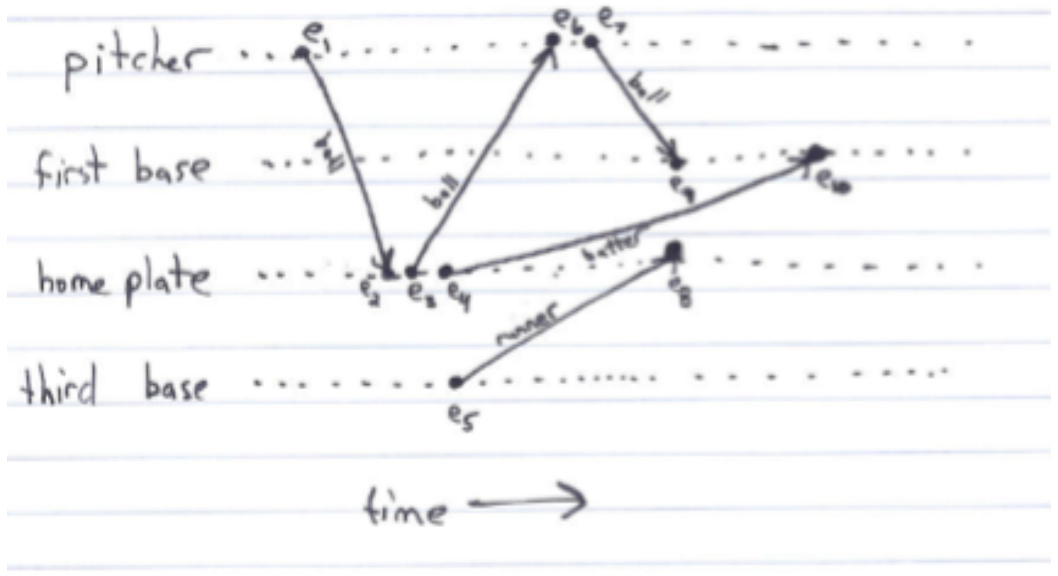
  e_8: runner arrives at home

  e_9: ball arrives at first base

  e_{10}: batter arrives at first base

A baseball example (contd)

- Pitcher knows e_1 happens before e_6, which happens before e_7
- Home plate umpire knows e_2 is before e_3, which is before e_4, which is before e_8, …
- Relationship between e_8 and e_9 is unclear



Logical time

- Capture just the "happens before" relationship between events
  - Discard the infinitesimal granularity of time
  - Akin to dependence edges in computation graph
- Time at each process is well-defined
  - Definition ($\rightarrow_i$): We say $e \rightarrow_i e'$ if e happens before e' at process i
    - Akin to "continue" edges

Global logical time

- **Definition ($\rightarrow$): We define e $\rightarrow$ e' using the following rules:**
  - **Local ordering:** $e \rightarrow e'$ **if** $e \rightarrow_i e'$ **for any process** i
  - **Messages: send(m) $\rightarrow$ receive(m) for any message** m
    - *Messages can encode ordering due to spawn, join, phaser, and isolated-serialization events*
  - **Transitivity:** $e \rightarrow e''$ **if** $e \rightarrow e'$ **and** $e' \rightarrow e''$
- **We say** e **"happens before"** e' **if** $e \rightarrow e'$
- **Definition (concurrency): We say** e **is concurrent with** e' **(written** $e \| e'$**) if neither** $e \rightarrow e'$ **nor** $e' \rightarrow e$
  - *i.e., e and e' "may happen in parallel"*

The baseball example revisited

- $e_1 \rightarrow e_2$
  - by the message rule
- $e_1 \rightarrow e_{10}$, because
  - $e_1 \rightarrow e_2$, by the message rule
  - $e_2 \rightarrow e_4$, by local ordering at home plate
  - $e_4 \rightarrow e_{10}$, by the message rule
  - Repeated transitivity of the above relations
- $e_8 \| e_9$, because
  - No application of the $\rightarrow$ rules yields either $e_8 \rightarrow e_9$ or $e_9 \rightarrow e_8$

Troublesome example

```
public class NoVisibility {
 private static boolean ready;
 private static int number;

 private static class ReaderThread extends Thread {
   public void run() {
     while (!ready) Thread.yield()
     System.out.println(number)
   }
 }

 public static void main(String[] args) {
   new ReaderThread().start();
   number = 42;
   ready = true;
 }
}
```

No happens-before ordering between main thread and ReaderThread
==> ReaderThread may loop forever OR may print 42 OR may print 0 !!

Volatile Variables available in Java (but not in HJ)

- Java provides a "light" form of synchronization/fence operations in the form of `volatile` variables (fields)
- Volatile variables guarantee visibility
  - Reads and writes of volatile variables are assumed to occur in isolated blocks
  - Adds serialization edges to computation graph due to isolated read/write operations on same volatile variable
- Incrementing a volatile variable (++v) is <u>not thread-safe</u>
  - Increment operation looks atomic, but isn't (read and write are two separate operations)
- Volatile variables are best suited for flags that have no dependencies e.g.,

```
volatile boolean asleep;
foo() { ... while (! asleep) ++sheep; ... }
```

— WARNING: In the absence of volatile declaration, the above code can legally be transformed to the following

```
boolean asleep;
foo() { boolean temp = asleep; ... while (! temp) ++sheep; ... }
```

Troublesome example fixed with volatile declaration

```
1. public class NoVisibility {
2.  private static volatile boolean ready;
3.  private static volatile int number;
4.
5.  private static class ReaderThread extends Thread {
6.    public void run() {
7.      while (!ready) Thread.yield()
8.      System.out.println(number)
9.    }
10.  }
11.
12.  public static void main(String[] args) {
13.    new ReaderThread().start();
14.    number = 42;
15.    ready = true;
16.  }
17. }
```

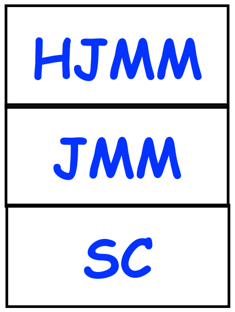
Declaring number and ready as volatile ensures happens-before-edges: 14-->15-->7-->8, thereby ensuring that only 42 will be printed

Outline

- Volatile variables
- <u>Java Memory Model</u>

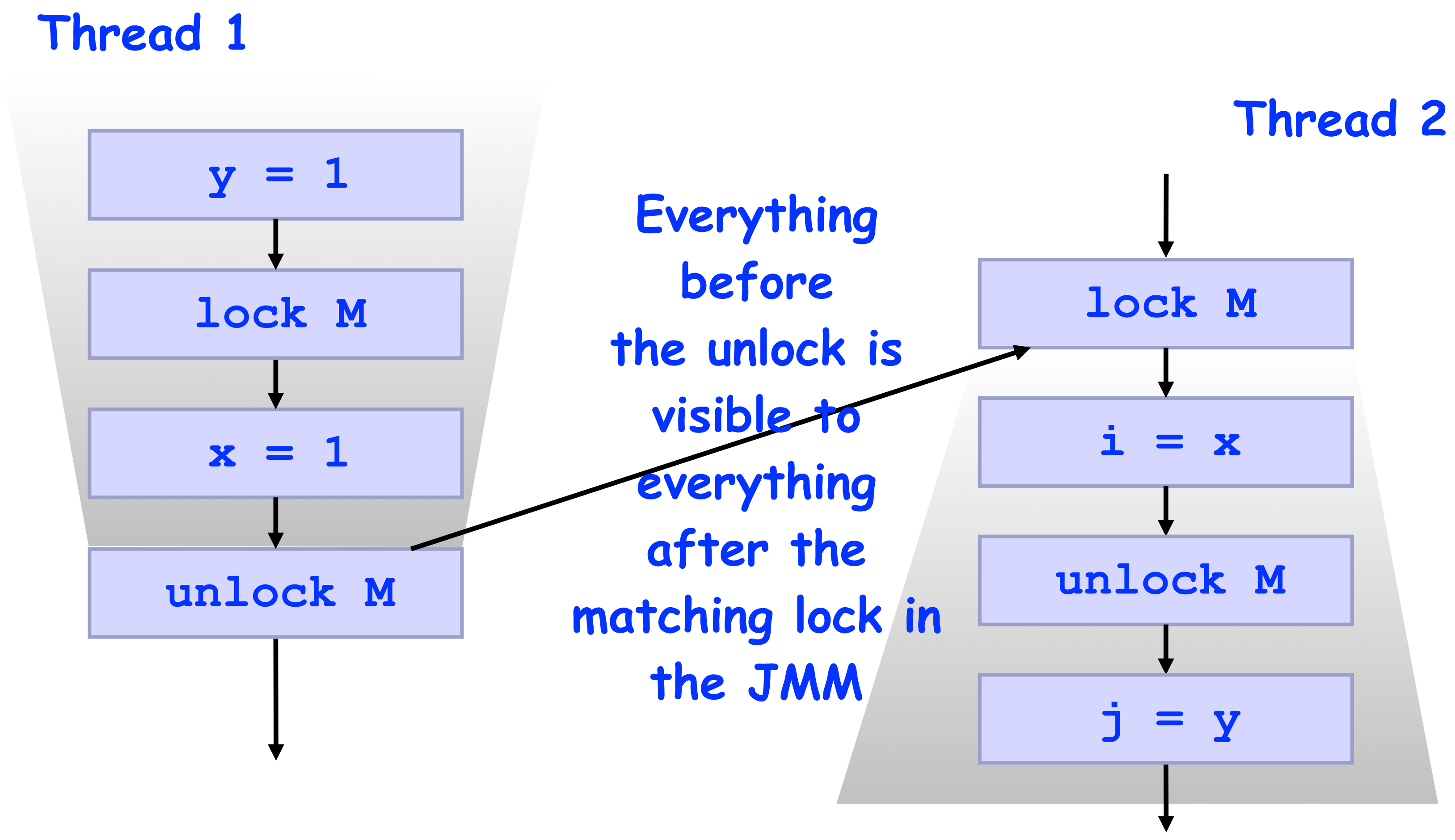
Memory Consistency Models (Recap from Lecture 6)

- A memory consistency model, or memory model, is the part of a programming language specification that defines what write values a read may see in the presence of data races.
- We will briefly introduce three memory models
  - Sequential Consistency (SC)
    - Suitable for specifying semantics at the hardware and OS levels *
  - Java Memory Model (JMM)
    - Suitable for specifying semantics at application thread level *
  - Habanero Java Memory Model (HJMM)
    - Suitable for specifying semantics at application task level *



* This is your instructor's opinion. Memory models are a very controversial topic in parallel programming!

When are actions visible and ordered with other Threads in the JMM?


Thread 1
y = 1
lock M
x = 1
unlock M
Thread 2
lock M
i = x
unlock M
j = y
Everything before the unlock is visible to everything after the matching lock in the JMM

lock/unlock operations can come from synchronized statement or from explicit calls to locking libraries

Troublesome example fixed with empty synchronized statements instead of volatile (JMM)

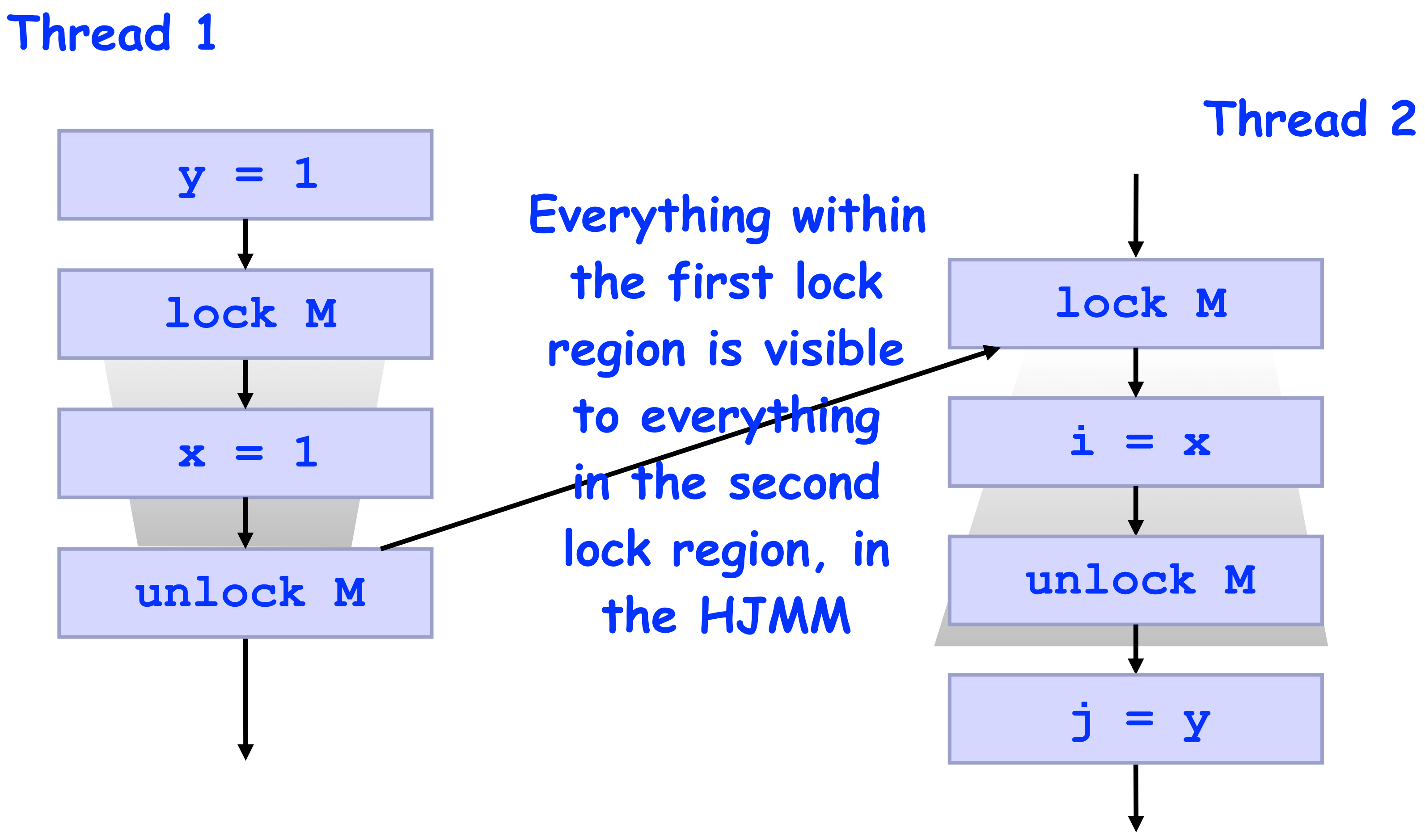
```
public class NoVisibility {
 private static boolean ready;
 private static int number;
 private static final Object a = new Object();

 private static class ReaderThread extends Thread {
   public void run() {
     synchronized(a){}
     while (!ready) { Thread.yield(); synchronized(a){} }
     System.out.println(number);
    }
  }

  public static void main(String[] args) {
    new ReaderThread().start();
    number = 42;
    ready = true; synchronized(a){}
  }
 }
```

Empty synchronized statement is NOT a no-op in Java. It acts as a memory "fence".

When are actions visible and ordered with other Threads in the HJMM?



Everything within the first lock region is visible to everything in the second lock region, in the HJMM

Empty isolated statements are no-ops in HJ

```
public class NoVisibility {
 private static boolean ready;
 private static int number;

 private static class ReaderThread extends Thread {
   public void run() {
     isolated{}
     while (!ready) { Thread.yield(); isolated{} }
     System.out.println(number);
   }
 }

 public static void main(String[] args) {
   new ReaderThread().start();
   number = 42;
   ready = true; isolated {}
 }
}
```

Empty isolated statement is a no-op in HJ. ReaderThread may loop forever OR may print 42 OR may print 0.

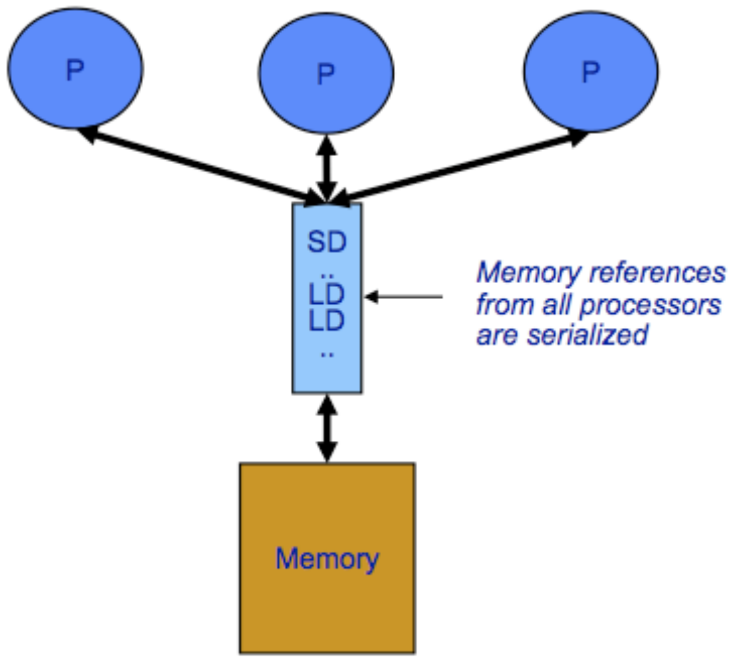
Use explicit synchronization in HJ instead

```
public class NoVisibility {
 private static boolean ready;
 private static int number;
 private static DataDrivenFuture<Boolean>
     readyDDF = new DataDrivenFuture<Boolean>();

 public static void main(String[] args) {
   async await(readyDDF){ System.out.println(number); }
   number = 42;
   readyDDF.put(true);
  }
}
```

REMINDER: HJ does not support volatile variables

Sequential Consistency Memory Model

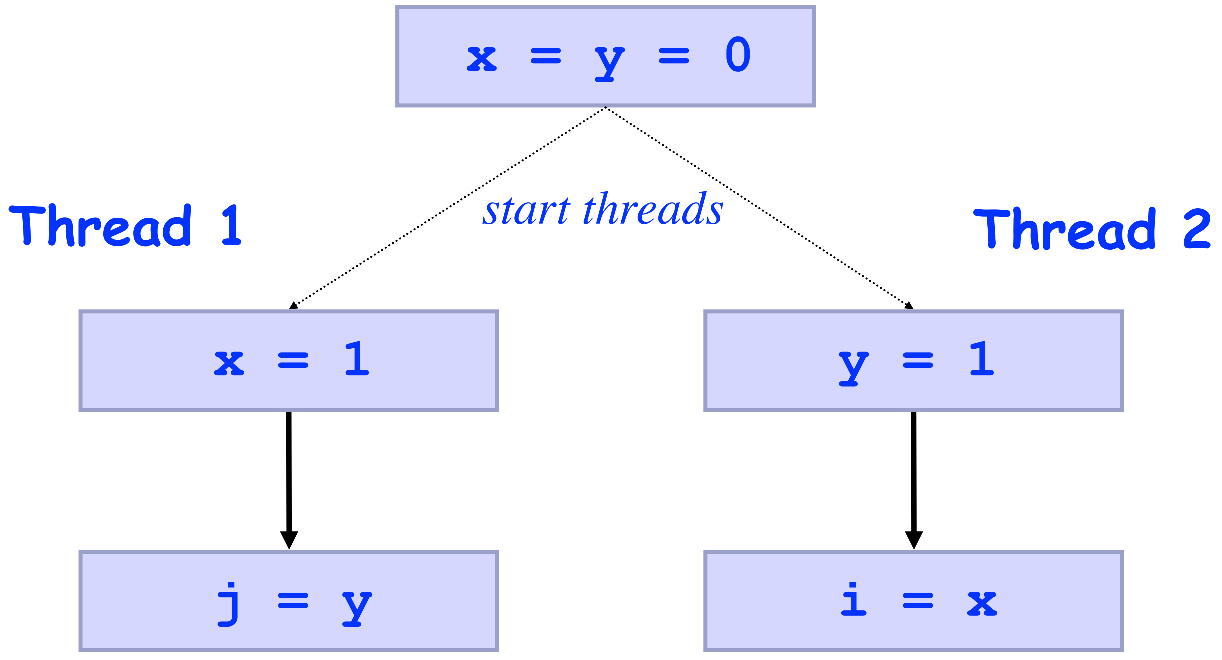


[Lamport] "*A multiprocessor system is sequentially consistent if the result of any execution is the same as if the operations of all processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by the program*"

The Habanero-Java Memory Model (HJMM) and the Java Memory Model (JMM)

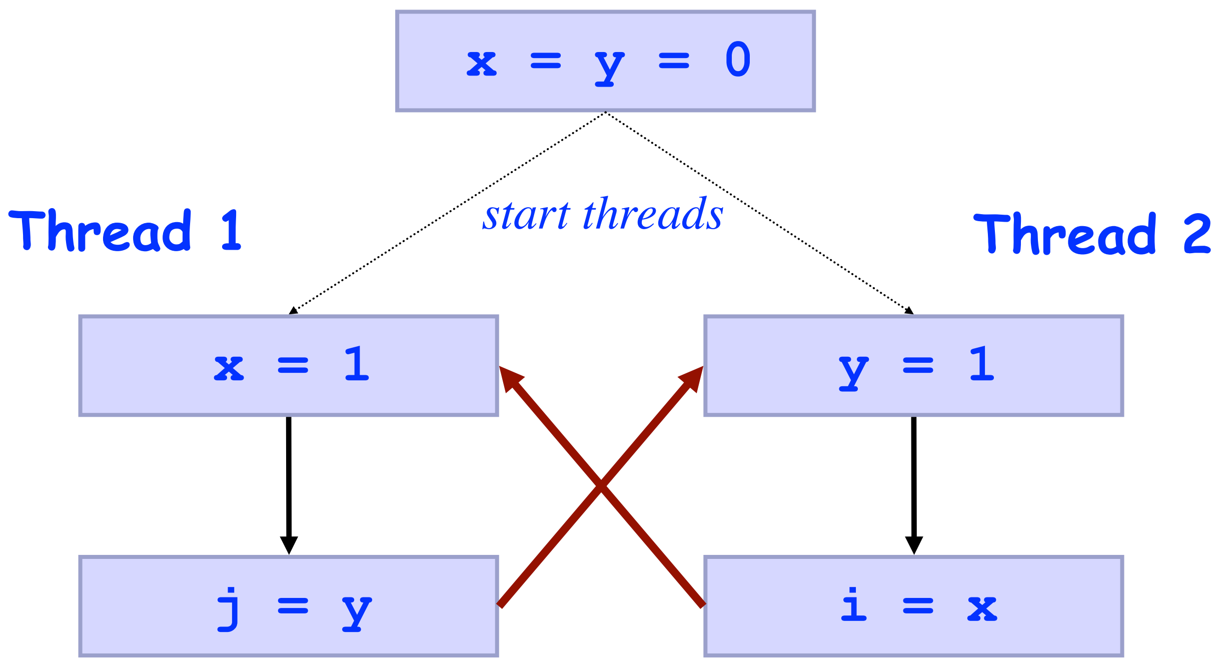
- Conceptually simple:
  - Every time a variable is written, the value is added to the set of "most recent writes" to the variable
  - A read of a variable is allowed to return ANY value from this set
- HJMM has weaker ordering rules for HJ's "isolated" statements, compared to Java's "synchronized" blocks
  - By using ordering relationships ("happens-before") in the Computation Graph to determine when a value must be overwritten
    - Also permit reordering of accesses to different locations within a sequential step
- The JMM defines the rules by which values in the set are removed
  - By using ordering relationships ("happens-before") similar to the Computation Graph to determine when a value must be overwritten
    - More restrictions on reordering of accesses to different locations relative to HJMM
- Programmer's goal: through proper use of synchronization
  - Ensure the absence of data races, in which case this set will never contain more than one value and SC, JMM, HJMM will all have the same semantics

Weird Behavior of Improperly Synchronized Code



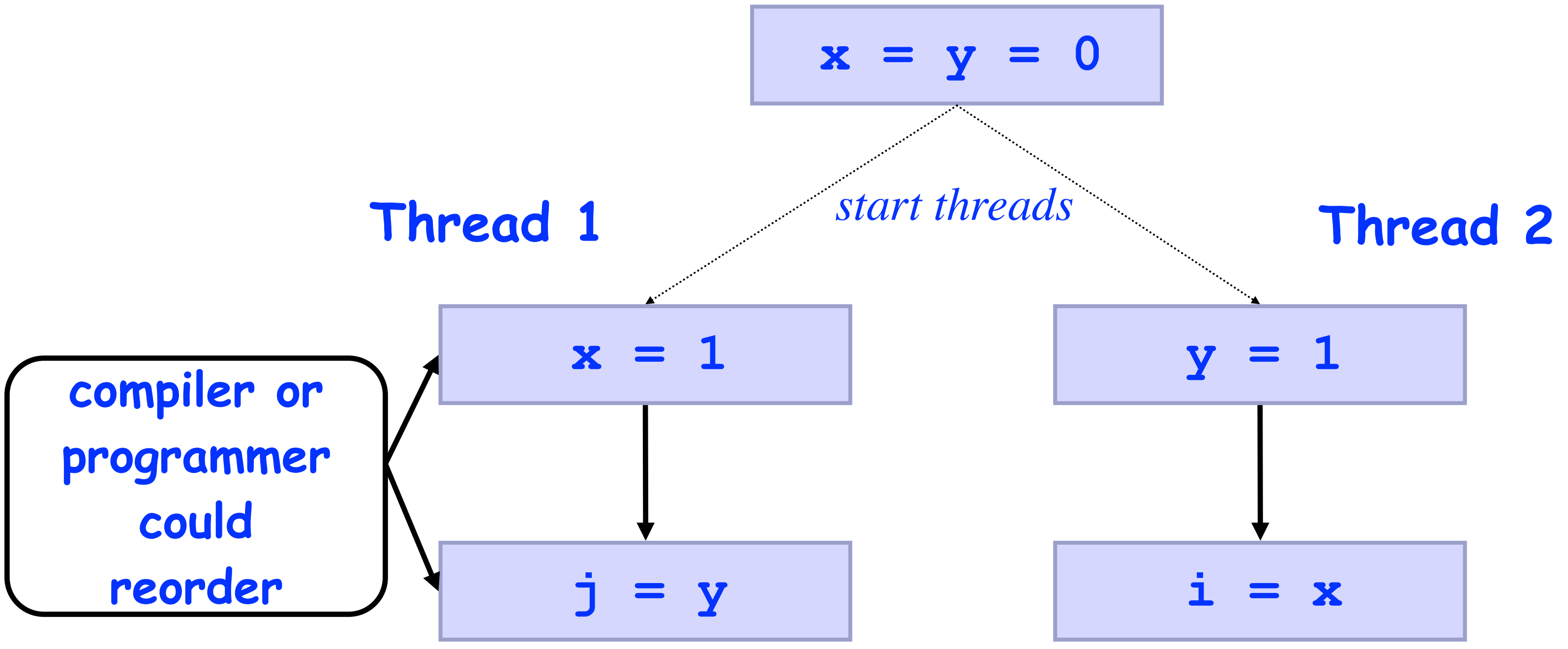
Can this result in i = 0 and j = 0?

No?



i = 0 and j = 0 implies temporal loop!

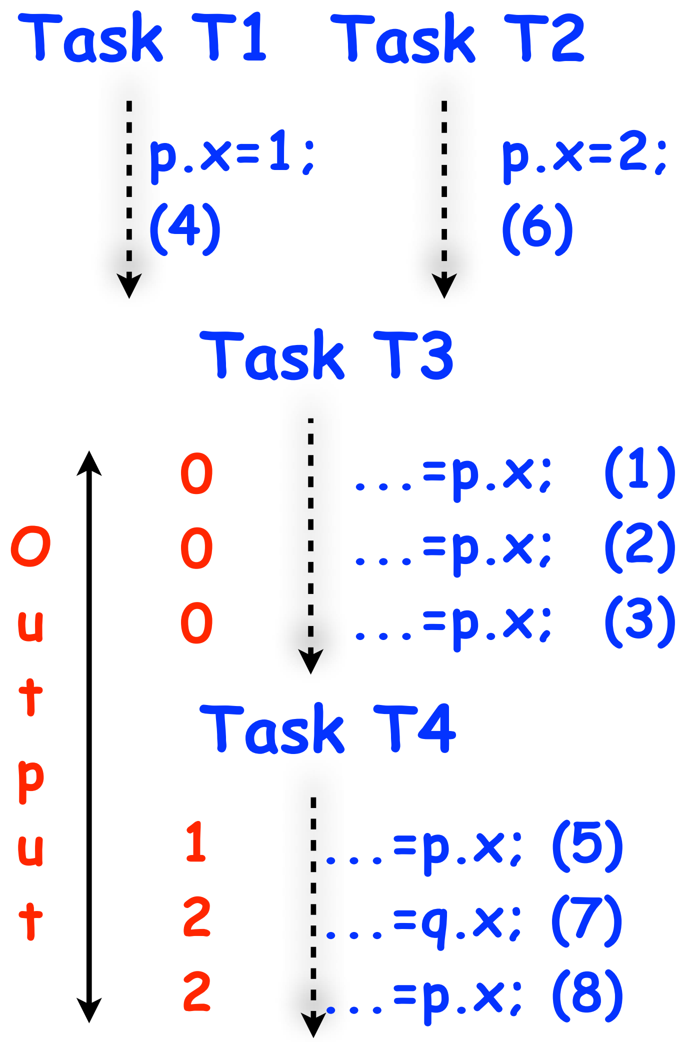
Answer: Yes in JMM and HJMM (but not in SC)



Sequential Consistency (SC) Memory Model

- SC constrains all memory operations across all tasks
  - Write → Read
  - Write → Write
  - Read → Read
  - Read → Write

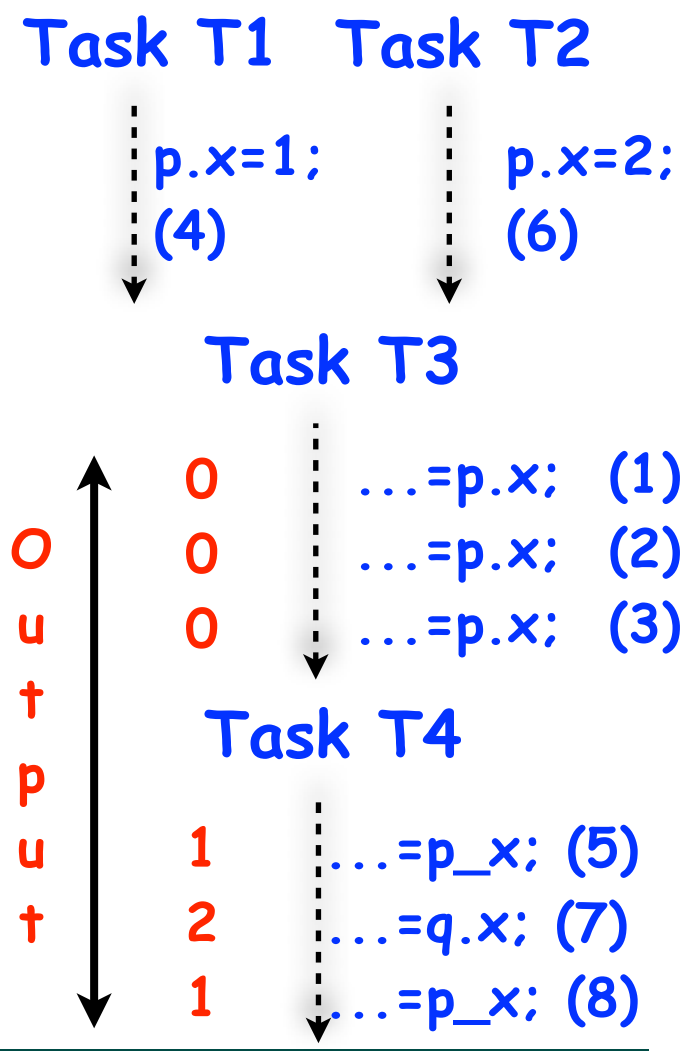
- Simple model for reasoning about data races at the hardware level, but may lead to counter-intuitive behavior at the application level e.g.,
  - A programmer may perform modular code transformations for software engineering reasons without realizing that they are changing the program's semantics



Consider a “reasonable” code transformation performed by a programmer

Example HJ program:

```
p.x = 0; q = p;
async p.x = 1; // Task T1
async p.x = 2; // Task T2
async { // Task T3
  System.out.println("First read = " + p.x);
  System.out.println("Second read = " + p.x);
  System.out.println("Third read = " + p.x)
}
async { // Task T4
  // Assume programmer doesn’t know that p=q
  int p_x = p.x;
  System.out.println("First read = " + p_x);
  System.out.println("Second read = " + q.x);
  System.out.println("Third read = " + p_x);
}
```

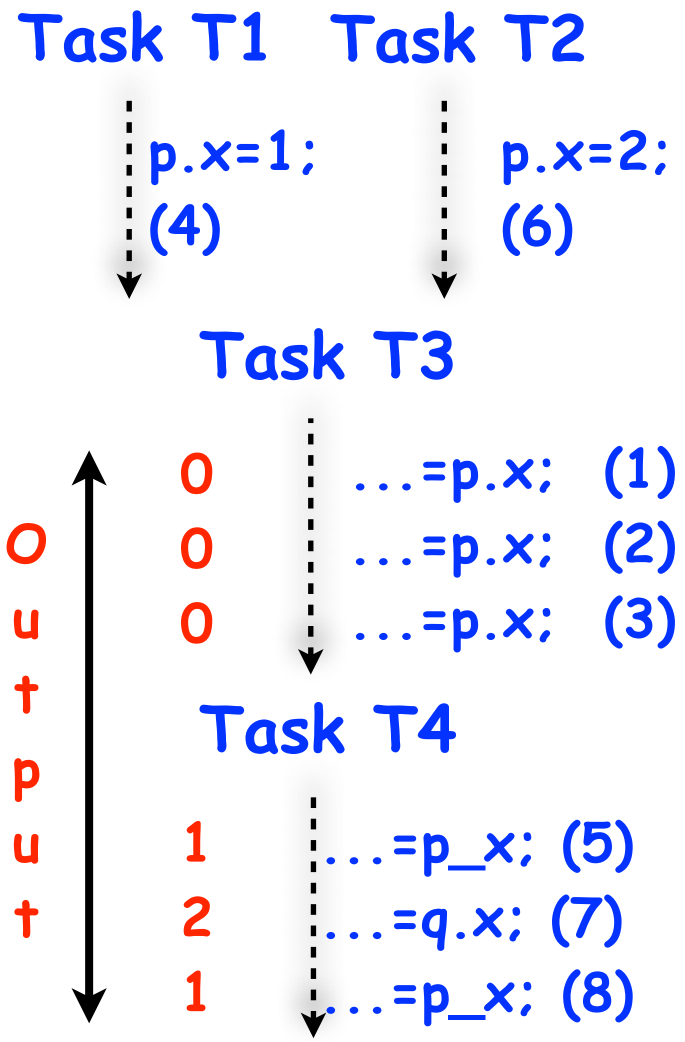


Consider a "reasonable" code transformation performed by a programmer

Example HJ program:

```
p.x = 0; q = p;
async p.x = 1; // Task T1
async p.x =
async {
  Sys                                              ;
  Sy                                              );
  Syst                                            x)
}
async { // Task T4
  // Assume programmer doesn't know that p=q
  int p_x = p.x;
  System.out.println("First read = " + p_x);
  System.out.println("Second read = " + q.x);
  System.out.println("Third read = " + p_x);
}
```

This reasonable code transformation resulted in an illegal output, under the SC model!

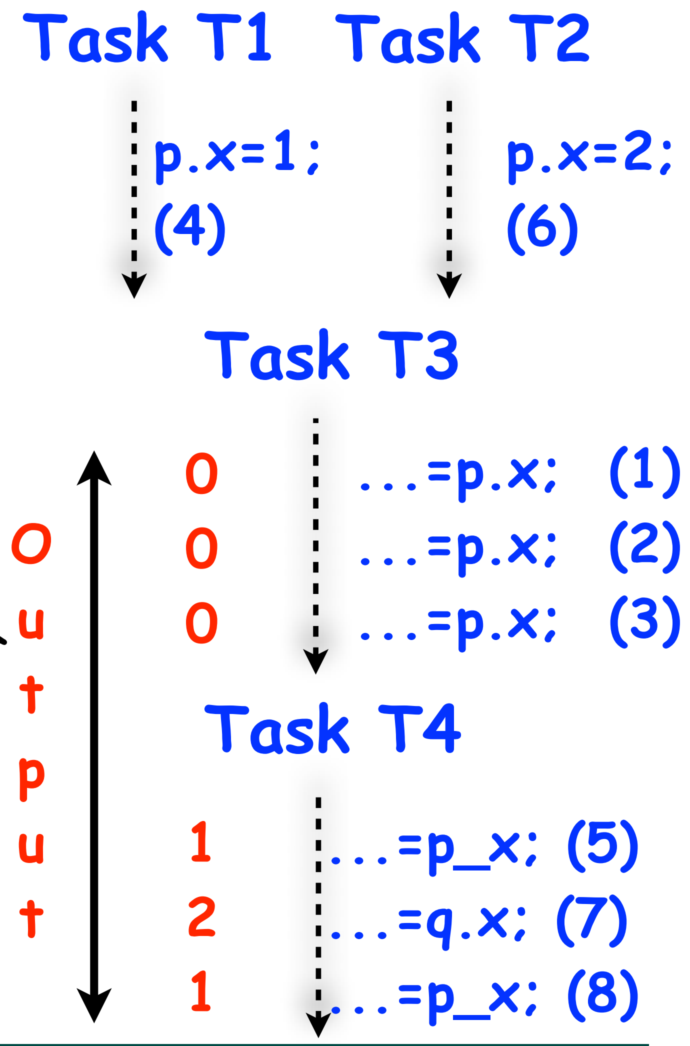


Code Transformation Example

Example HJ program:

```
p.x = 0; q = p;
async p.x = 1; // Task T1
async p.x =
async {
  Sy                                          ;
  Sy                                          );
  Syst                                        x)
}
async { // Task T4
  // Assume programmer doesn’t know that p=q
  int p_x = p.x;
  System.out.println("First read = " + p_x);
  System.out.println("Second read = " + q.x);
  System.out.println("Third read = " + p_x);
}
```

This output is legal under the JMM and HJMM!



Reminders

- Graded midterms can be picked up from Amanda Nokleby in Duncan Hall 3137
- Homework 5 due by 5pm on Friday, April 6th