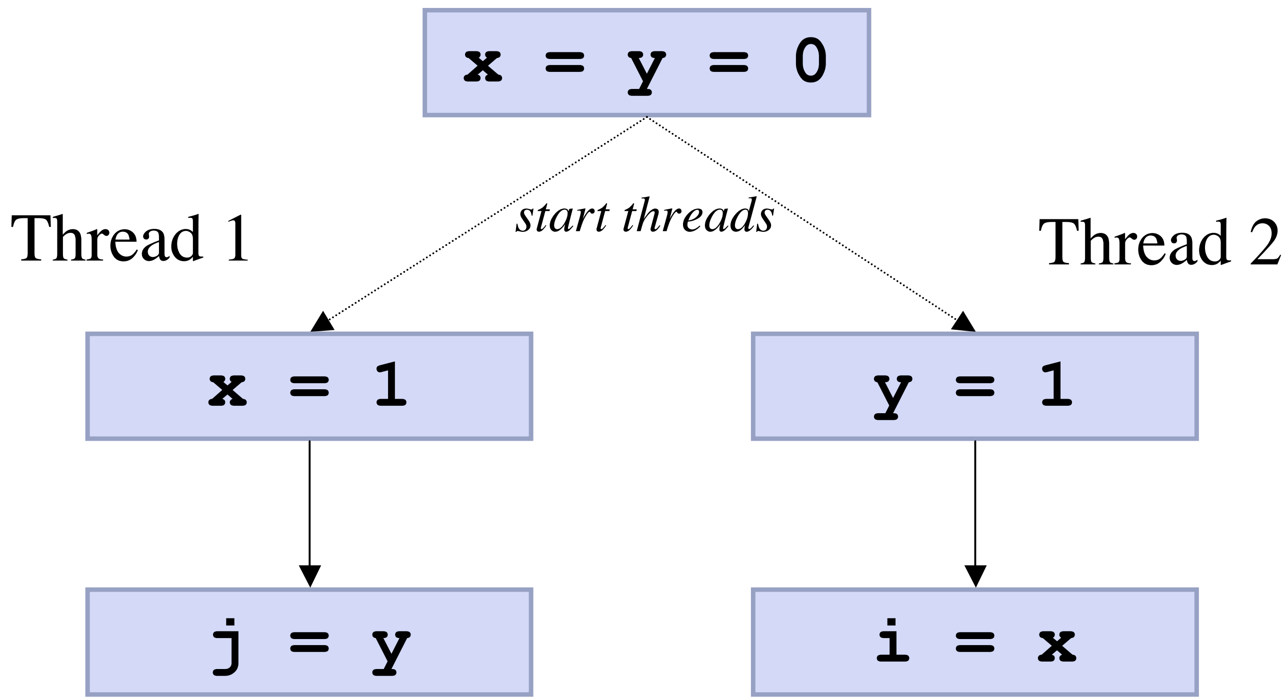
# The Java Memory Model and Simulator

Jeremy Manson, William Pugh
Univ. of Maryland, College Park

# Java Memory Model and Thread Specification

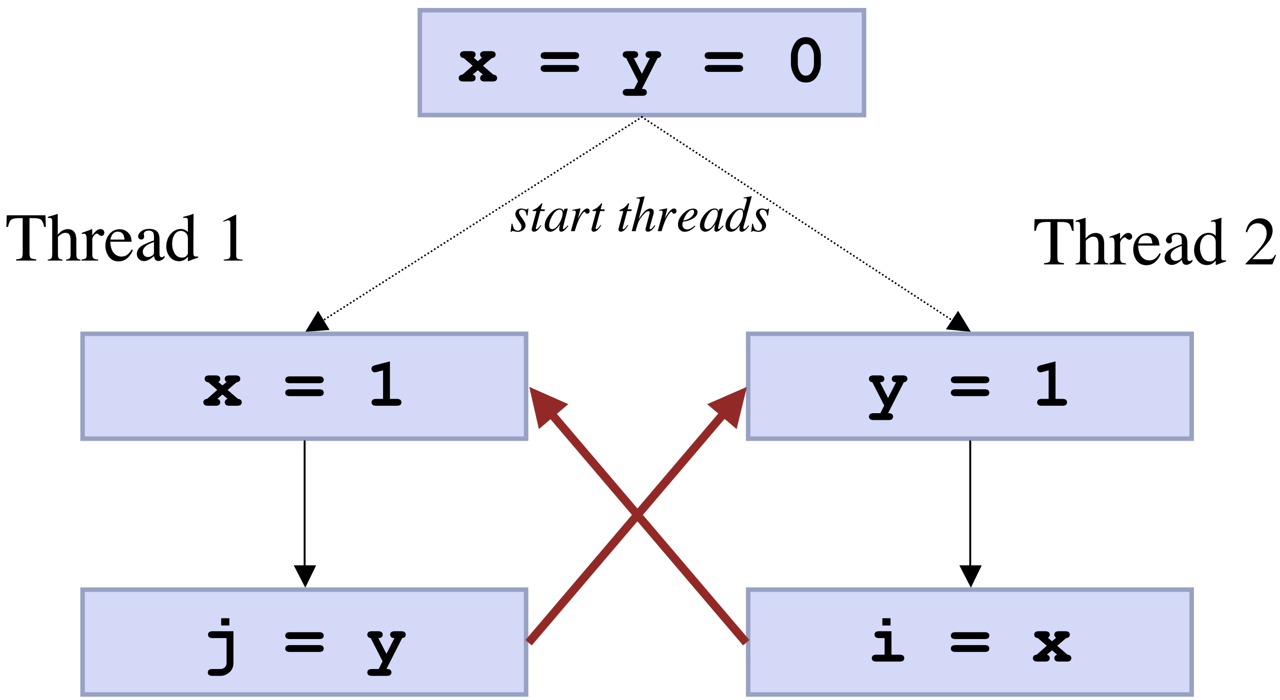
- Defines the semantics of multithreaded programs
  - When is a program correctly synchronized?
  - What are the semantics of an incorrectly synchronized program?
    - e.g., a program with data races

# Weird Behavior of Improperly Synchronized Code



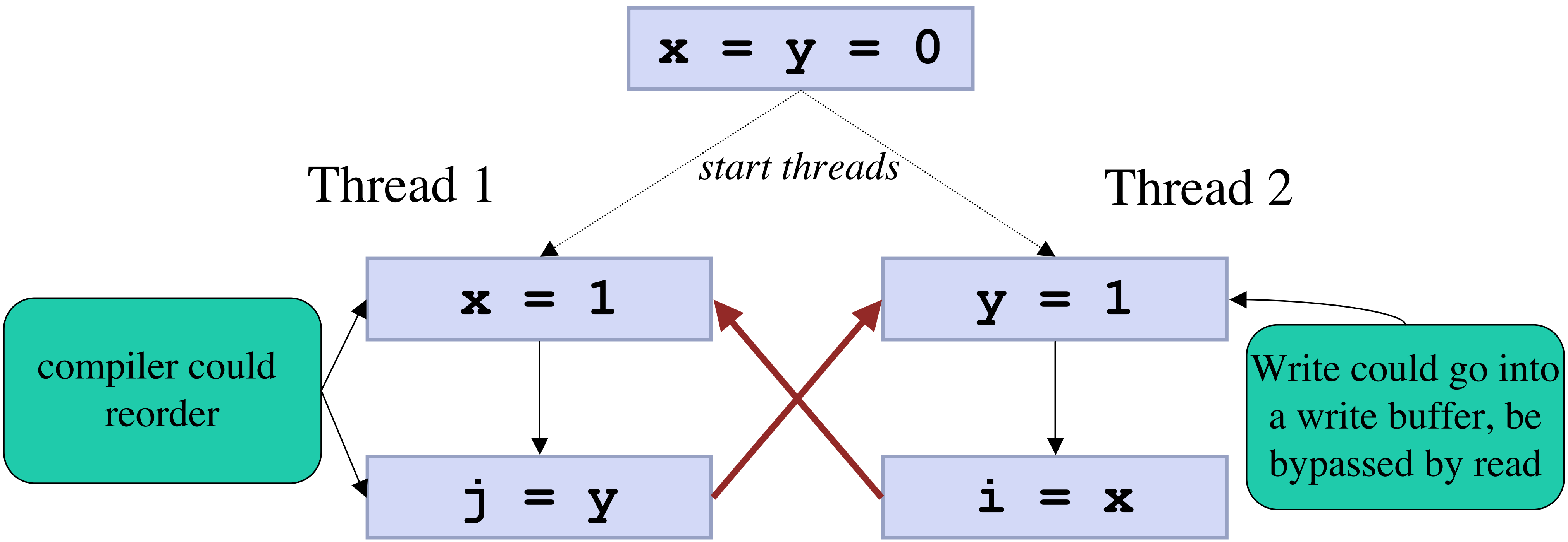

**Can this result in i = 0 and j = 0?**

# No?




**i = 0 and j = 0 implies temporal loop!**

# Answer: Yes!




**How can i = 0 and j = 0?**

# How Can This Happen?

- Compiler can reorder statements
  - or keep values in registers
- On multiprocessors, values not synchronized in global memory
  - Writes go into write buffer
  - Are bypassed by reads
- Must use synchronization to enforce visibility and ordering
  - as well as mutual exclusion

# Java Thread Specification

- Chapter 17 of the Java Language Spec
  - Chapter 8 of the Virtual Machine Spec
- Very, very hard to understand
  - not even the authors understood it
  - doubtful that anyone entirely understands it
  - has subtle implications
    - that forbid standard compiler optimizations
  - all existing JVMs violate the specification
    - some parts should be violated

# Revising the Thread Spec

- JSR 133 will revise the Java Memory Model
  - http://www.cs.umd.edu/~pugh/java/memoryModel
- Goals
  - Clear and easy to understand
  - Foster reliable multithreaded code
  - Allow for high performance JVMs
- Will affect JVMs
  - and badly written existing code
    - including parts of Sun's JDK

# Proposed Changes

- Make it clear
- Allow standard compiler optimizations
- Remove corner cases of synchronization
  - enable additional compiler optimizations
- Strengthen volatile
  - make easier to use
- Strengthen final
  - Enable compiler optimizations
  - Fix security concerns
  - *no time to talk about this in this talk*

# Incorrect synchronization

- Incorrectly synchronized program must have well defined semantics
  - Much other work in the field has avoided defining any semantics for incorrectly synchronized programs
- Synchronization errors might be deliberate
  - to crack security of a system
  - just like buffer overflows

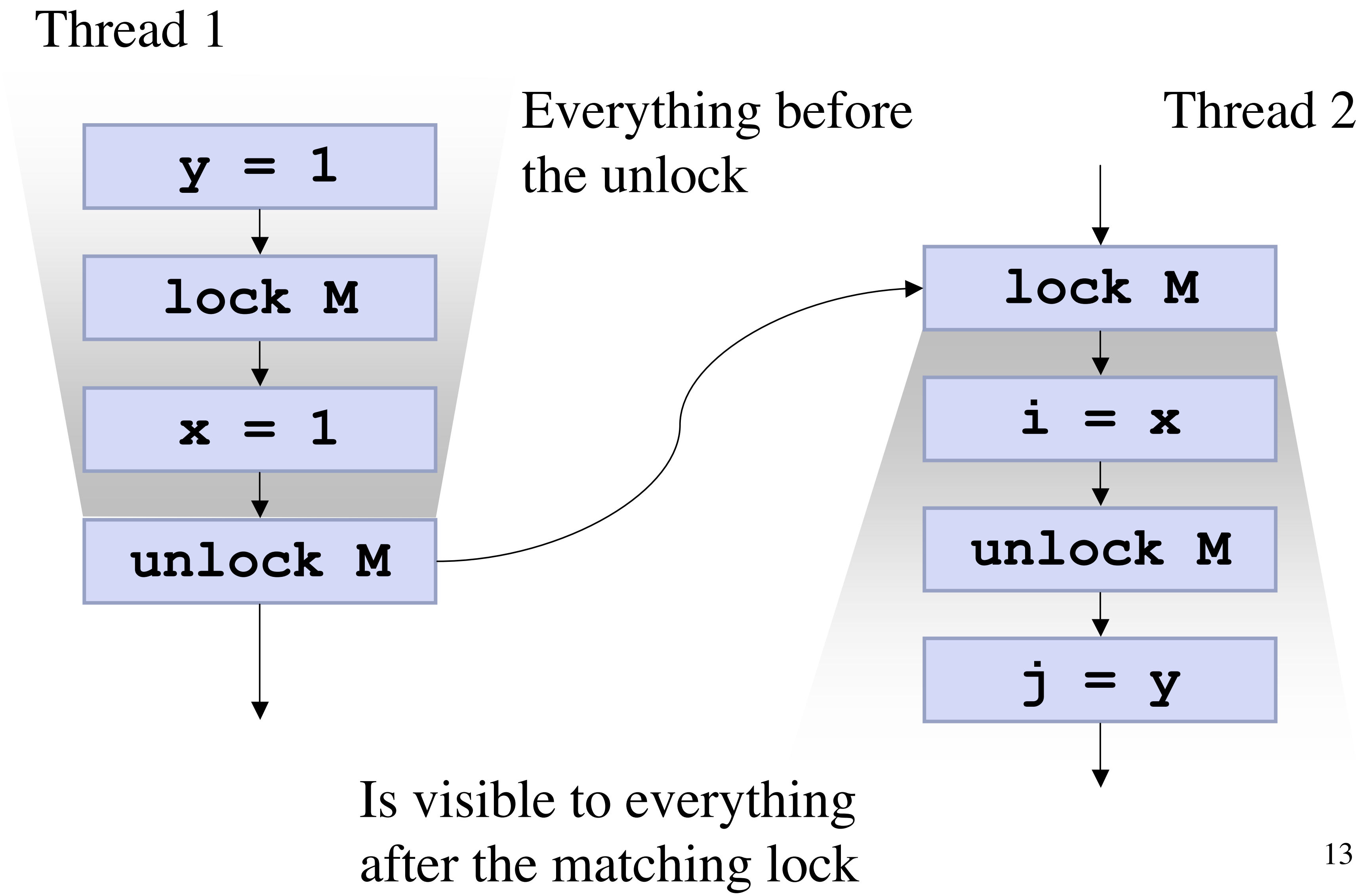
# VM Safety

- Type safety
- Not-out-of-thin-air safety
  - (except for longs and doubles)
- No new VM exceptions
- Only thing lack of synchronization can do is produce surprising values for getfields/getstatics/array loads
  - e.g., arraylength is always correct

# Synchronization

- Programming model is (lazy) release consistency
  - A lock acts like an acquire of data from memory
  - An unlock acts like a release of data to memory

# When are actions visible and ordered with other Threads?



# New Optimizations Allowed

- Turning synchronizations into no-ops
  - locks on objects that aren't ever locked by any other threads
  - reentrant locks

- Lock coarsening
  - merging two calls to synchronized methods on same object
    - need to be careful about starvation issues

# Existing Semantics of Volatile

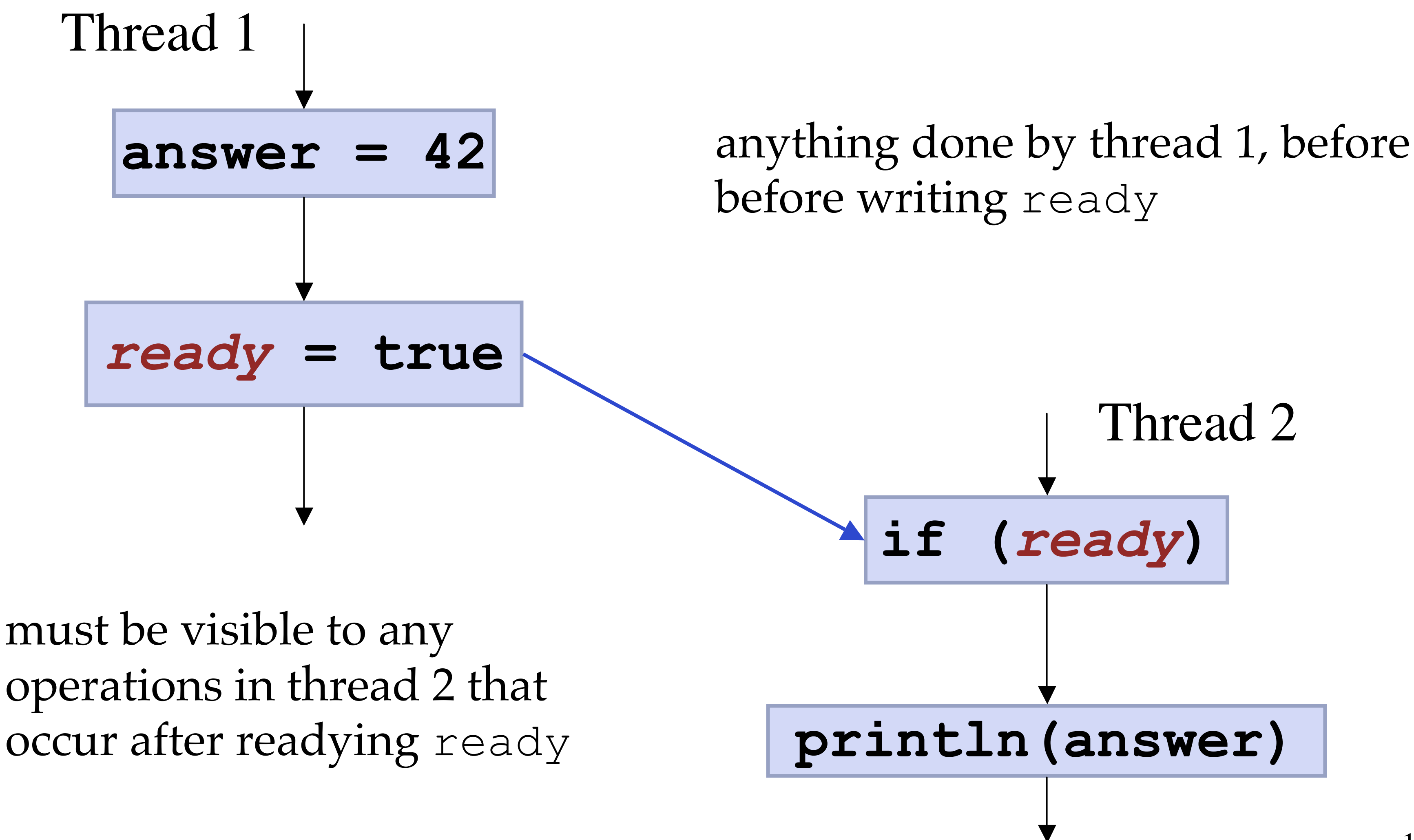
- No compiler optimizations
  - Can't hoist read out of loop
  - reads/writes go directly to memory
- Reads/writes of volatile are sequentially consistent and can not be reordered
  - but access to volatile and non-volatile variables can be reordered – makes volatiles much less useful
- Reads/writes of long/doubles are atomic

# Proposed *New, Additional* Semantics for Volatile

- Write to a volatile acts as a release
- Read of a volatile acts as an acquire

- If a thread reads a volatile
  - all writes done by any other thread,
  - before earlier writes to the same volatile,
  - are guaranteed to be visible

# When Are Actions Visible to Other Threads?



# Non-atomic volatiles?

a and b are volatile and initially 0

| | | | |
|---|---|---|---|
| a = 1 | r1 = a | b = 1 | r3 = b |
| | r2 = b | | r4 = a |

Can we get r1 = 0, r2 = 1, r3 = 0, r4 = 1?

# Conflicting opinions

- Hans Boehm (HP) and Rick Hudson (Intel) say this behavior must be allowed to allow Java to be implemented efficiently on future architectures
- Sarita Adve (UIUC) says nonsense
- I’ll let them fight it out

# Conflicting and unclear goals/constraints

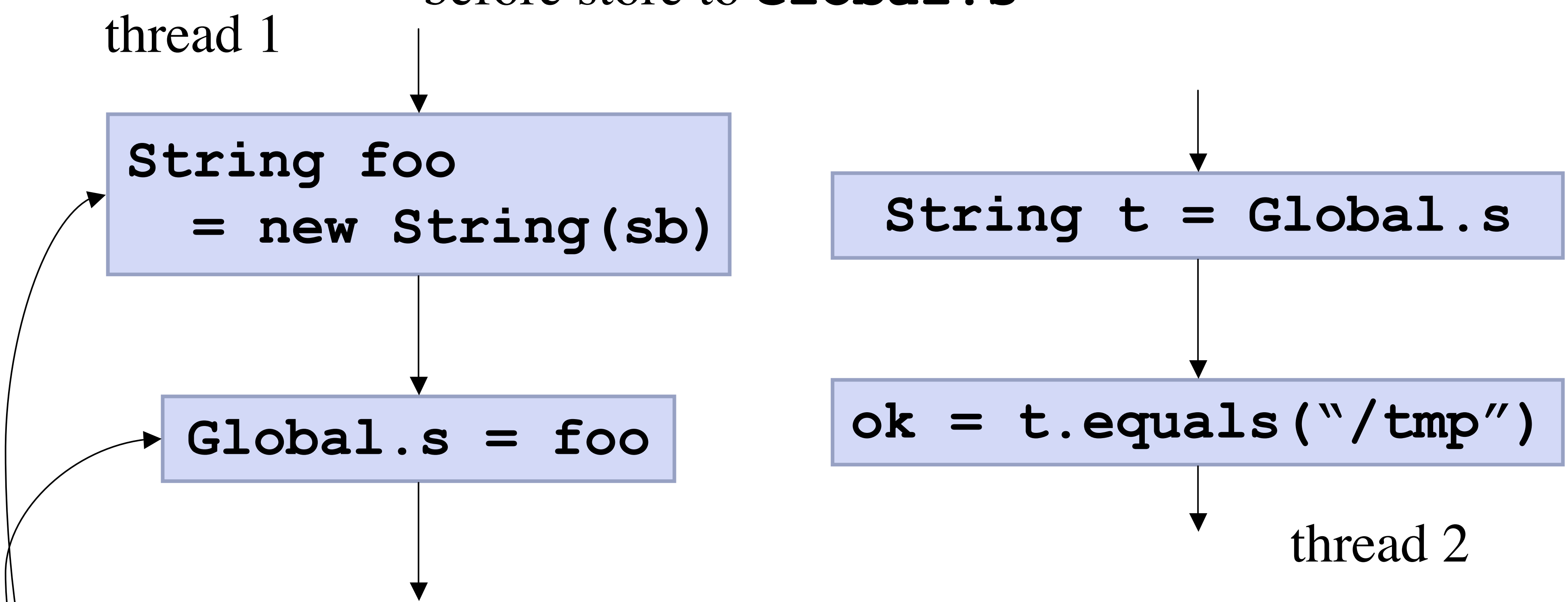
- Three different goals, often in conflict
  - what VM implementers need
  - what Java programmers need
    - for efficient, reliable software
    - for security
  - making the spec clear and simple
- None of these are clearly or formally specified

# Immutable Objects

- Many Java classes represent immutable objects
  - e.g., String
- Creates many serious security holes if Strings are not truly immutable
  - probably other classes as well
  - should do this in String implementation, rather than in all uses of String

# Strings aren't immutable

just because thread 2 sees new value for **`Global.s`** doesn't mean it sees all writes done by thread 1 before store to **`Global.s`**




Compiler, processor or memory system can reorder these writes

*Symantic JIT will do it*

# Why aren't Strings immutable?

- A String object is initialized to have default values for its fields
- *then* the fields are set in the constructor
- Thread 1 could create a String object
- pass it to Thread 2
- which calls a sensitive routine
- which sees the fields change from their default values to their final values

# Final = Immutable?

- Existing Java memory model doesn't mention final
  - no special semantics
- Would be nice if compiler could treat final fields as constant
  - Don't have to reload at memory barrier
  - Don't have to reload over unknown function call

# Proposed Semantics for Final

- Read of a final field always sees the value set in constructor
  - unless object is not constructed properly
    - allows other threads to view object before completely constructed
- Can assume final fields never change
- Makes string immutable?

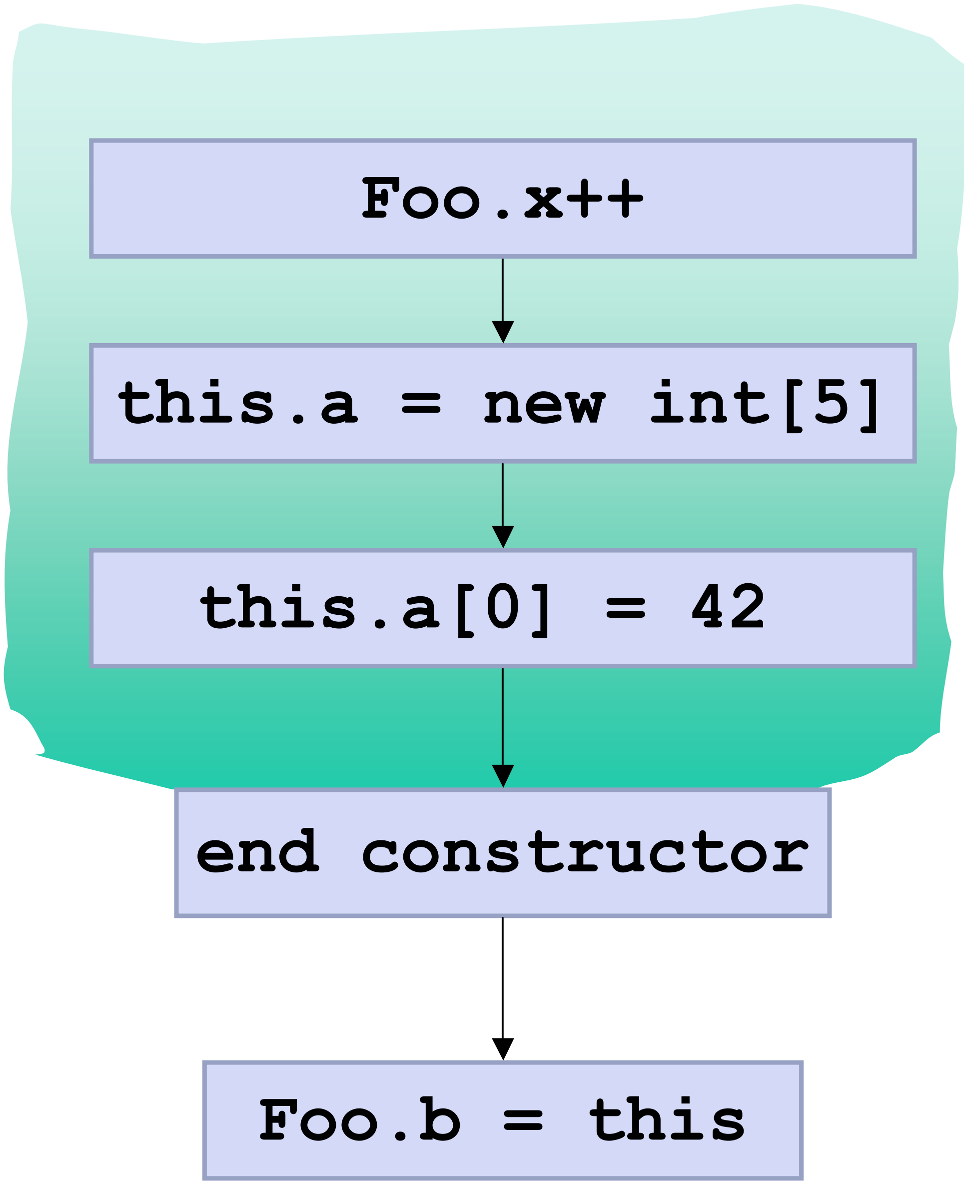
# Problems

- JNI code can change final fields
  - System.setIn, setOut, setErr
  - **Propose to remove this ability**
    - hack for setIn, setOut, setErr
- Objects that can be seen by other threads before constructor is complete
- Doesn't suffice to make strings immutable

# Doesn't make Strings immutable

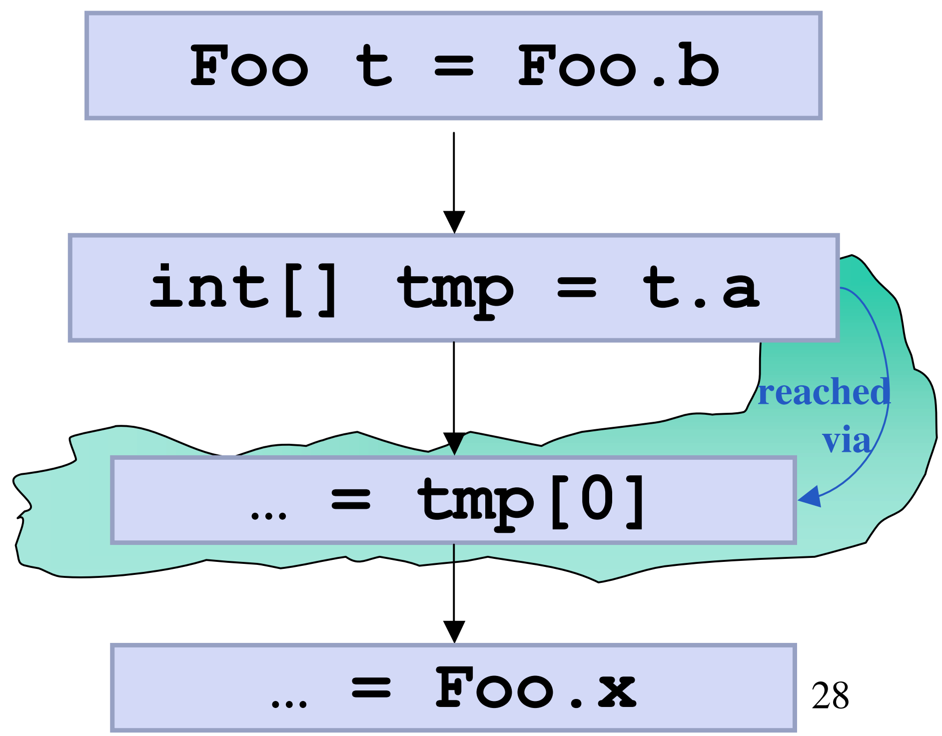
- No way for elements of an array to be final
- For Strings, have to see final values for elements of character array
- So…
  - Read of final field is treated as a weak acquire
    - matching a release done when object is constructed
  - weak in that it only effects things dependent on value read
    - no compiler impact

# Visibility enforced by final field a

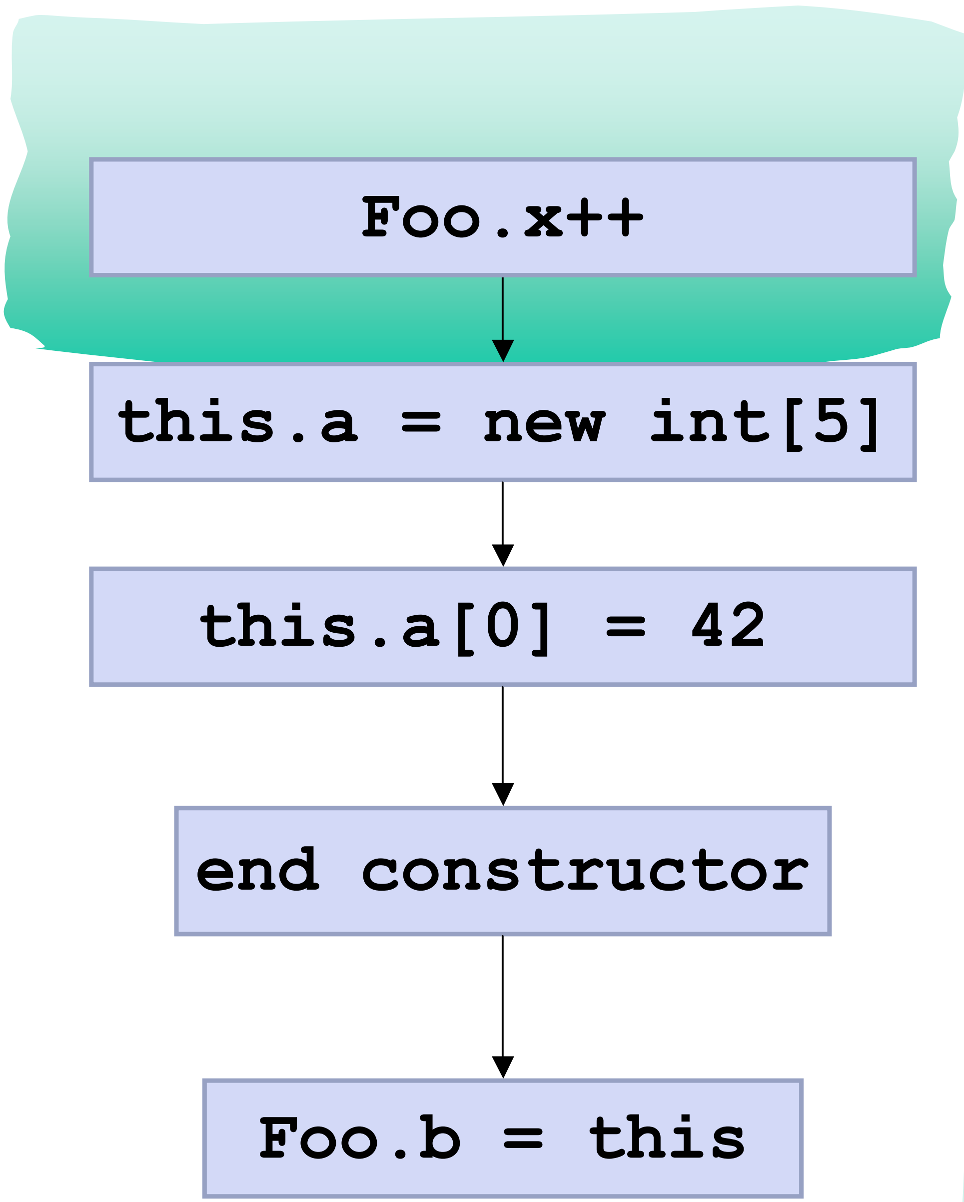



All actions done before completion of constructor

must be visible to any action that is data dependent on the read of a final field set in that constructor

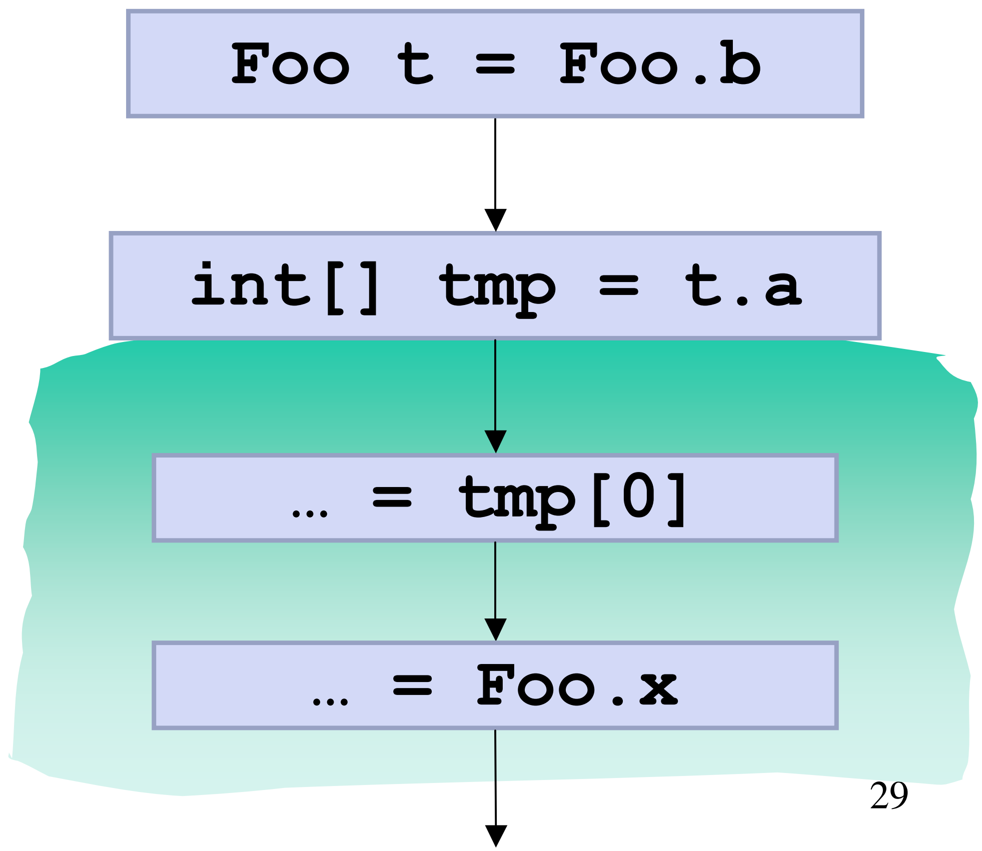


# Contrast with volatile

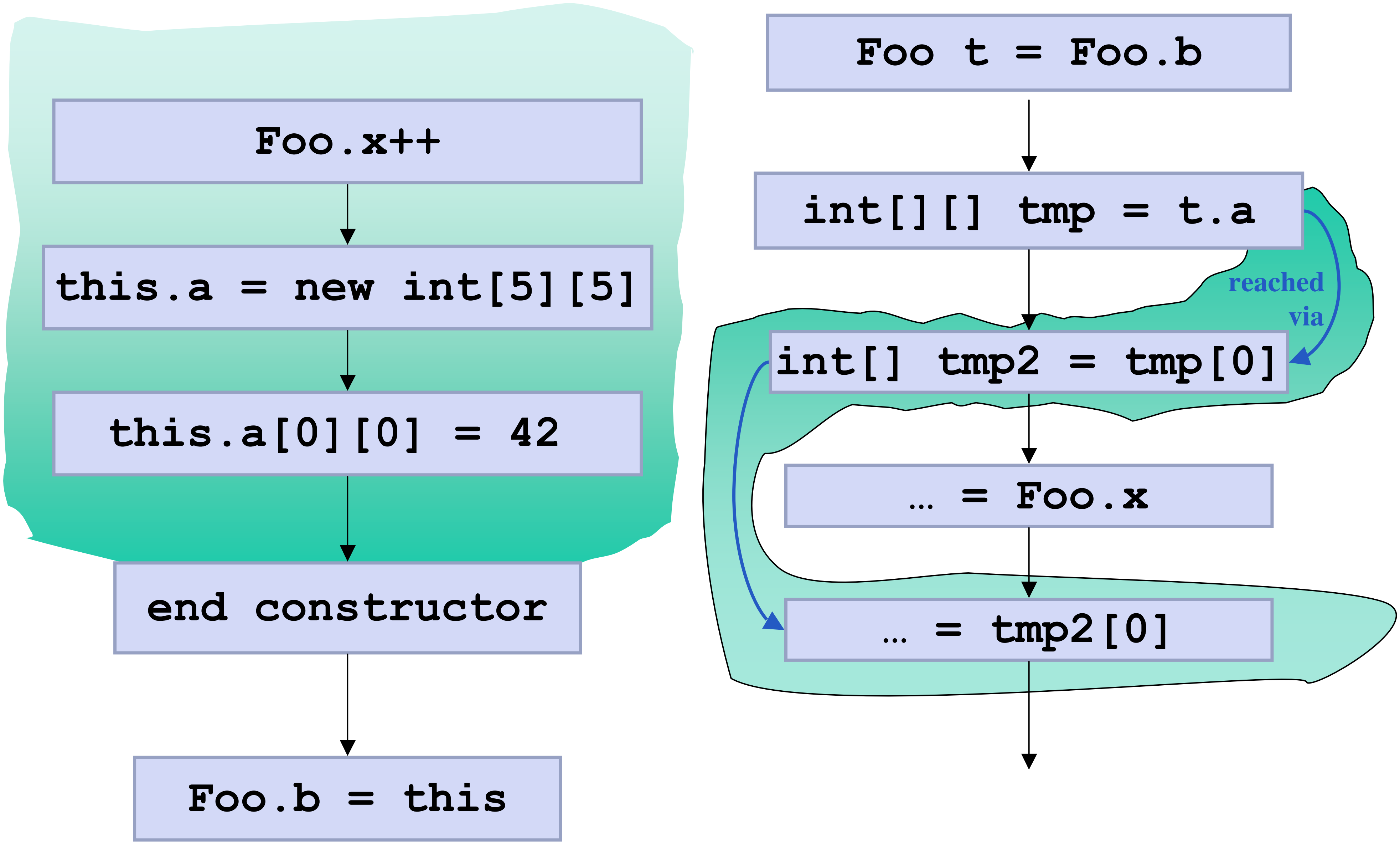



Actions done before assignment to volatile field

must be visible to any action after the read



# Data dependence is transitive



# Complications

- Semantics said that two different references to the same object might have different semantics
  - one reference published “correctly”, one published “prematurely”
- JVM implementers insisted this wasn’t acceptable
- Changing the semantics to accommodate JVM implementers

# Some things to make your brain hurt

Why this is hard…

# Consider

Initially, x = y = 0

Thread 1

```
r1 = x
if r1 >= 0 then
  y = 1
```

Thread 2

```
r2 = y
if r2 >= 0 then
  x = 1
```

Can this result in r1 = r2 = 1?

# Real example

- While not too many systems will do an analysis to determine non-negative integers
- Compilers might want to determine references that are definitely non-null

# Null Pointer example

Initially

```
Foo.p = new Point(1,2);
Foo.q = new Point(3,4);
Foo.r = new Point(5,6);
```

| Thread 1 | Thread 2 |
| --- | --- |
| r1 = Foo.p.x; | r2 = Foo.q.x; |
| Foo.q = Foo.r; | Foo.p = Foo.r; |

Can this result in r1 = r2 = 5?

# A Formalization of the Proposed Semantics for Multithreaded Java

Jeremy Manson & Bill Pugh

# Basic Framework

- Operational semantics
- Actions occur in a global order
  - consistent with original order in each thread
    - except for prescient writes
- If program not correctly synchronized
  - reads non-deterministically choose which value to return from set of candidate writes

# Terms

- Variable
  - a heap allocated field or array element
- Value
  - a primitive type or reference to an object
- Local
  - a value stored in a local or on the stack
- Write
  - a <variable, value, GUID> triplet
  - GUID used to distinguish writes
    - e.g., two writes of 42 to the same variable

# Write Sets

- allWrites: all writes performed so far
- Threads/monitors/volatiles have/know:
  - overwritten: a set of writes known to be overwritten
  - previous: a set of writes known to be in the past
- These are all monotonic sets
  - they only grow

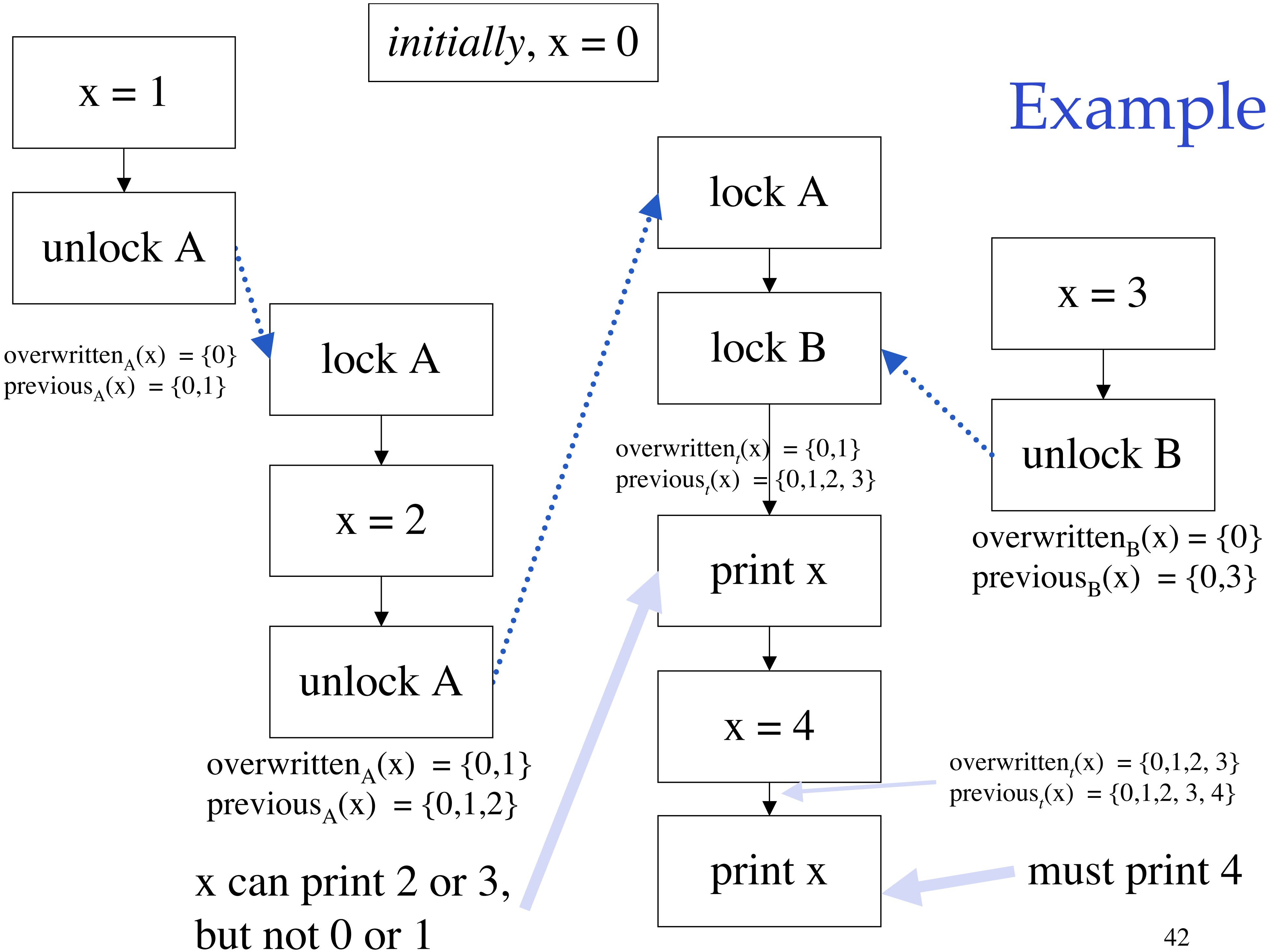
# Normal Reads

- A non-final, non-volatile read
- Nondeterministically returns a write in AllWrites
  - that the thread doesn't know to be overwritten

# Normal Writes

- All writes known to be previous
  - are added to overwritten
- The write performed
  - is added to allWrites and previous

# Example

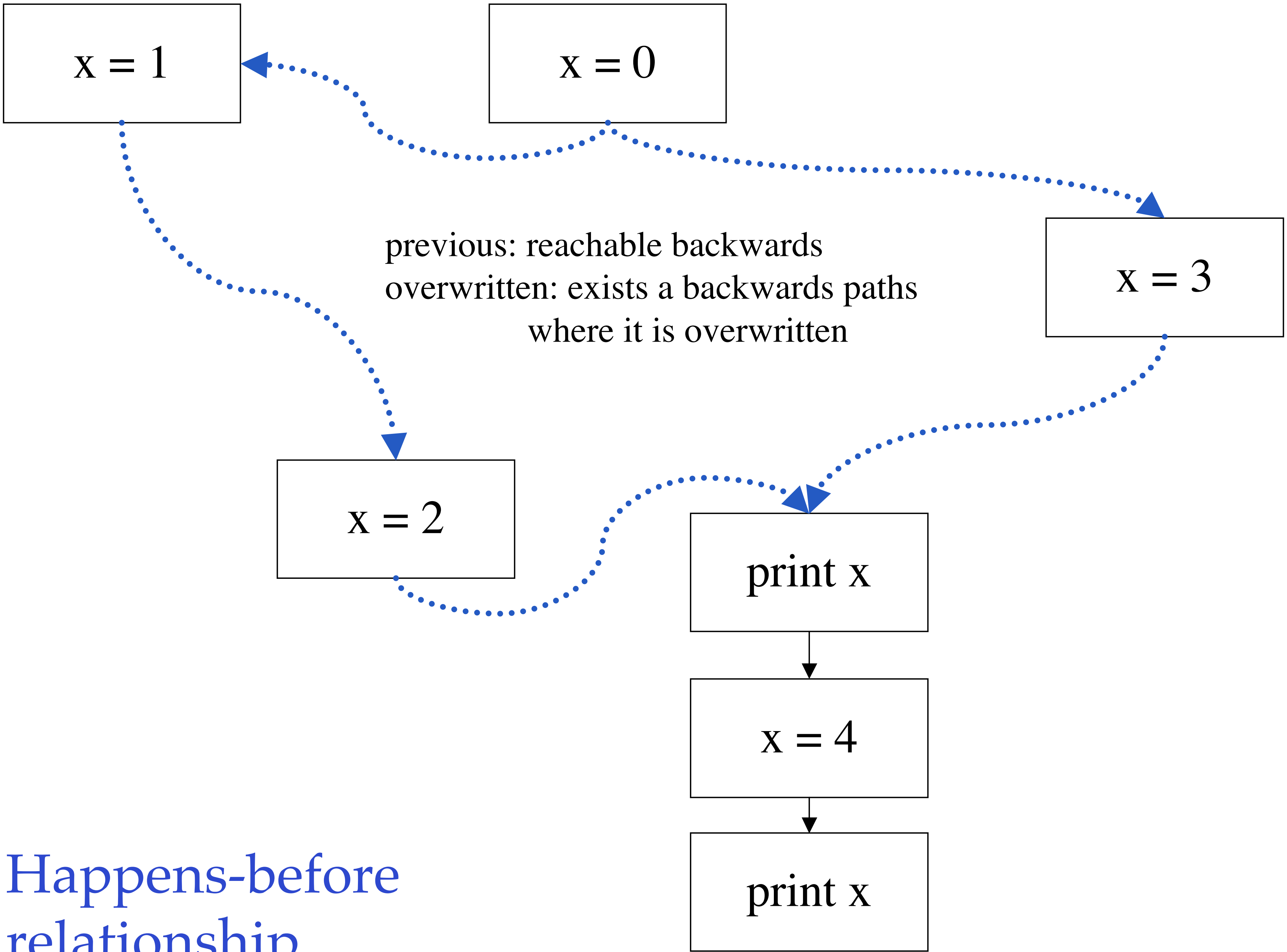


*initially*, x = 0
x = 1
unlock A
overwritten$_A$(x) = {0}
previous$_A$(x) = {0,1}
lock A
x = 2
unlock A
overwritten$_A$(x) = {0,1}
previous$_A$(x) = {0,1,2}
x can print 2 or 3, but not 0 or 1
lock A
lock B
overwritten$_t$(x) = {0,1}
previous$_t$(x) = {0,1,2, 3}
print x
x = 4
overwritten$_t$(x) = {0,1,2, 3}
previous$_t$(x) = {0,1,2, 3, 4}
print x
must print 4
x = 3
unlock B
overwritten$_B$(x) = {0}
previous$_B$(x) = {0,3}

# x = 0

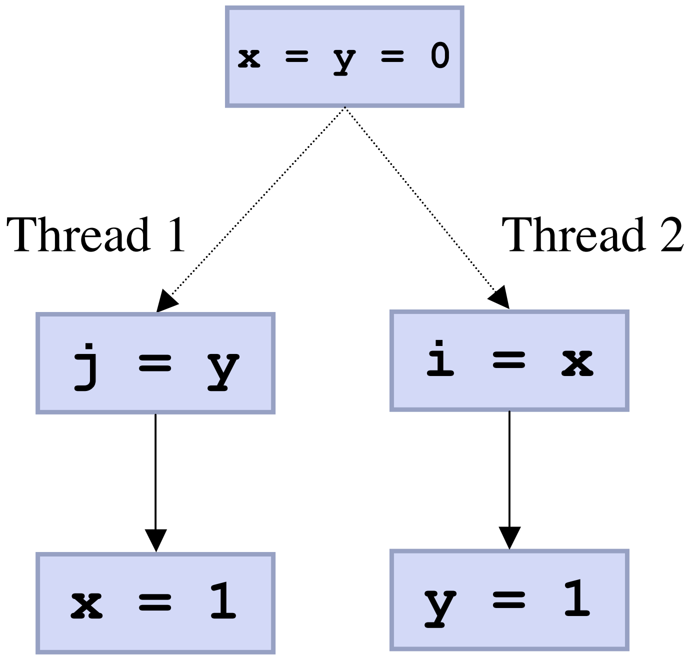
- x = 1
- x = 2
- x: {2,3,4}
- print x

- x = 3
- x: {1,2,3}
- allWrites = {0,1,2,3}
  previous_2 = {0,3}
  overwritten_2 = {0}
- x = 4
- allWrites = {0,1,2,3,4}
  previous_2 = {0,3,4}
  overwritten_2 = {0,3}
- print x




# Happens-before relationship

# Prescient Writes




- In original order, some write instruction must go first
  - Neither can
- Use prescient write instead

**Can this result in i = 1 and j = 1?**

# The Java Memory Model Simulator

# Motivation

- Memory model is complicated
  - Want to ensure it does what we want it to do
- Proof techniques are costly and complicated
  - Often informal or applied to a subset of the semantics
  - Needs to be performed again every time semantics are changed
  - Doesn't mean we don't want to do them!

# Simulator

- Allows us to take small programs, and produce all of their possible results.
- Compiler writers plug in examples and possible optimizations
  - Reveals all of the possible outcomes.
  - If optimization is illegal, introduces new behavior

# Transmogrifier


- Given a program
  - applies "standard compiler transformations"
    - e.g., can reorder two independent memory accesses
    - or move a memory access inside a synchronized block
    - doesn't try to optimize, just generates legal transformations
  - For each resulting program
    - check that simulator produces no new behaviors

# Implementation

- Two implementations – Haskell and Java
  - Haskell for rapid prototyping
    - The rules translate easily into Haskell
  - Java for efficiency
    - Much easier to write efficient Java code
- Helps to ensure understanding of semantics
  - conflicts are sometimes broken implementation, sometimes because semantics are unclear

# Input Language – Closing the Semantic Gap

- Wanted something intuitive, similar to what programs look like
- Very similar to Java, but optimized for small examples – ex:

| | |
|---|---|
| Begin_Thread<br>  Local i;<br>  i = this.x;<br>  this.y = 1;<br>End_Thread | Begin_Thread<br>  Local j;<br>  j = this.y;<br>  this.x = 1;<br>End_Thread |

# Control Flow

- Full control flow would be nice, but is unimplemented
  - Also would cause a lot more states
- `if ... else ... endif` construct
- "Spin wait" statement
  - Thread does not proceed until condition is true.
  - Captures some interesting cases

# More About the Input Language

- Also has other language features
  - Objects and references
  - Final and volatile fields
  - More planned features
    - Dynamic allocation
    - More Control flow
- But we need to support features inimical to the model, not just to languages...

# Prescient Writes

- Prescient writes can be placed in some places, not in others
  - semantics will verify correct placement
  - but can't generate all legal placements
    - except through exhaustive testing
- Automatically place of prescient writes of constant values within the same basic block as original write
- Other prescient writes can be placed by hand

# Efficiency

- For each thread with its instruction set, there are a lot of possible interleavings
- Going through them all would be very expensive

# Worklist-based approach

- Keep list of
  - states seen but not yet explored
    - worklist
  - states seen
- Don’t add to worklist states already seen
  - If we see a state through more than one program path, it doesn’t get explored separately

# Timing Environment

- Dual 350 MHz Pentium II, 1 GB RAM
- Sun JDK 1.4.0
- 57 Litmus Tests
  - 2 – 5 Threads, 2 – 17 Instructions each

# Results

| | Optimized | | | Unoptimized | | |
|---|---|---|---|---|---|---|
| Test Name | States | CPU | Total Time | States | CPU | Total Time |
| coherence | 23 | 0:02 | 0:02 | 67 | 0:02 | 0:02 |
| alpha-3 | 77 | 0:03 | 0:03 | 364 | 0:04 | 0:02 |
| final-2 | 77 | 0:04 | 0:04 | 379 | 0:05 | 0:04 |
| navolatile | 209 | 0:11 | 0:10 | 2720 | 1:01 | 0:51 |
| PC-5 | 2277 | 2:25 | 1:42 | dnf | dnf | dnf |
| All 57 | 6148 | 6:19 | 5:23 | dnf | dnf | dnf |

Times are MM:SS – All done in Java (for performance)
dnf – Simulator took more than 24 hours

# Live Demo

# Related Work

- Original Specification is Ch. 17 of Java Language Spec.
  - Lots of people have studied it
  - Model still broken, doesn't meet needs of Java programmers and of VM implementers
- Maessen, Arvind, Shen: Improving the Java Memory Model using CRF
  - Useful in understanding core issues
  - Formalization was only able to handle some requirements of the new Java MM

# Related Work

- Yang, Gopalakrishnan, Lindstrom; Analyzing the CRF Java Memory Model
  - A simulator for CRF memory model, using Mur□
- *Ibid*, Formalizing the Java Memory Model for Multithreaded Program Correctness and Optimization
  - Attempt to build simulator for our semantics
  - semantics are not the same as our model
    - treats all potential dependences as strict ordering constraints
    - doesn’t handle references

# More related work

- Moore, Porter:An Executable Formal Virtual Machine Thread Model
  - Specifies Java using an operational semantics
  - Assumes Sequential Consistency for multithreaded semantics

# Future work

- Finish Memory Model
  - Still needs some work (mostly polish), for which the simulator helps
- Continue work on Simulator
  - Support full looping, dynamic allocation
  - Support other memory models (SC)
  - Support more realistic programs
  - Explaining results to users

# Conclusions

- PL memory models
  - more complicated than architecture models
    - Have to consider compiler and architecture optimizations
  - balance usability, security and implementability
    - understandable (limited) model for programmers
      - this is how you should program
    - full details understandable by VM implementers and authors of thread tutorials

# Conclusions

- Simulator helps us with these problems
  - Different Haskell & Java versions helpful
  - Simply going through the process of writing simulator helps refine the semantics
- Ease of use is valuable
  - VM Builders and those creating new libraries can use tool to see possible legal results