The Java Memory Model

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?
    - A correctly synchronized program has only SC semantics
  - What are the semantics of an incorrectly synchronized program?
    - A program with data races in an SC execution

Proposed Changes

- Make it unambiguous
- 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

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

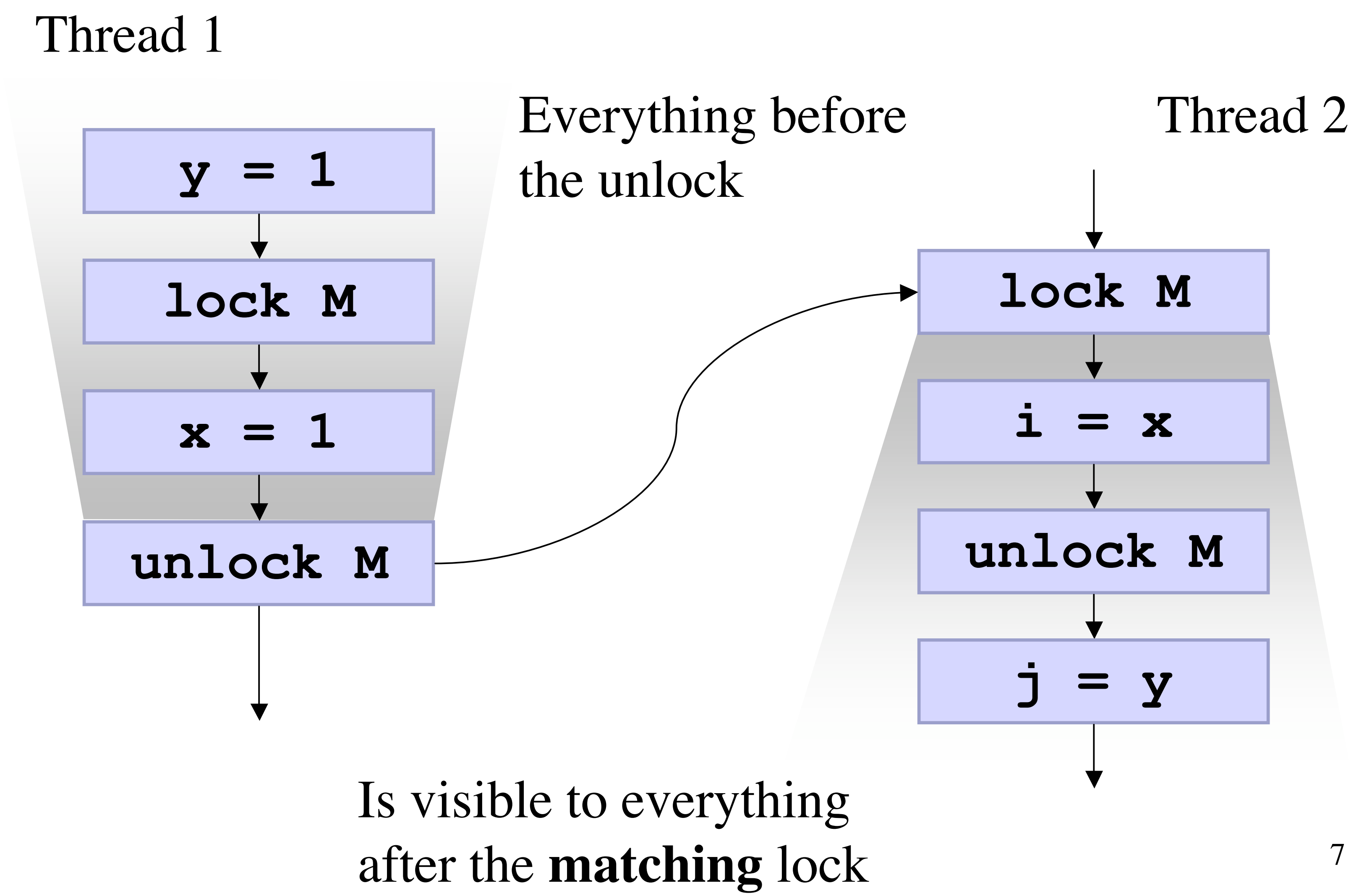
Read/Write atomicity

- All reads and writes are atomic
  - except for non-volatile longs and doubles
- No word tearing

Synchronization

- Programming model is similar to 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
  - Some actions have no memory semantics:
    - 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 – more on this later

Old Semantics of Volatile

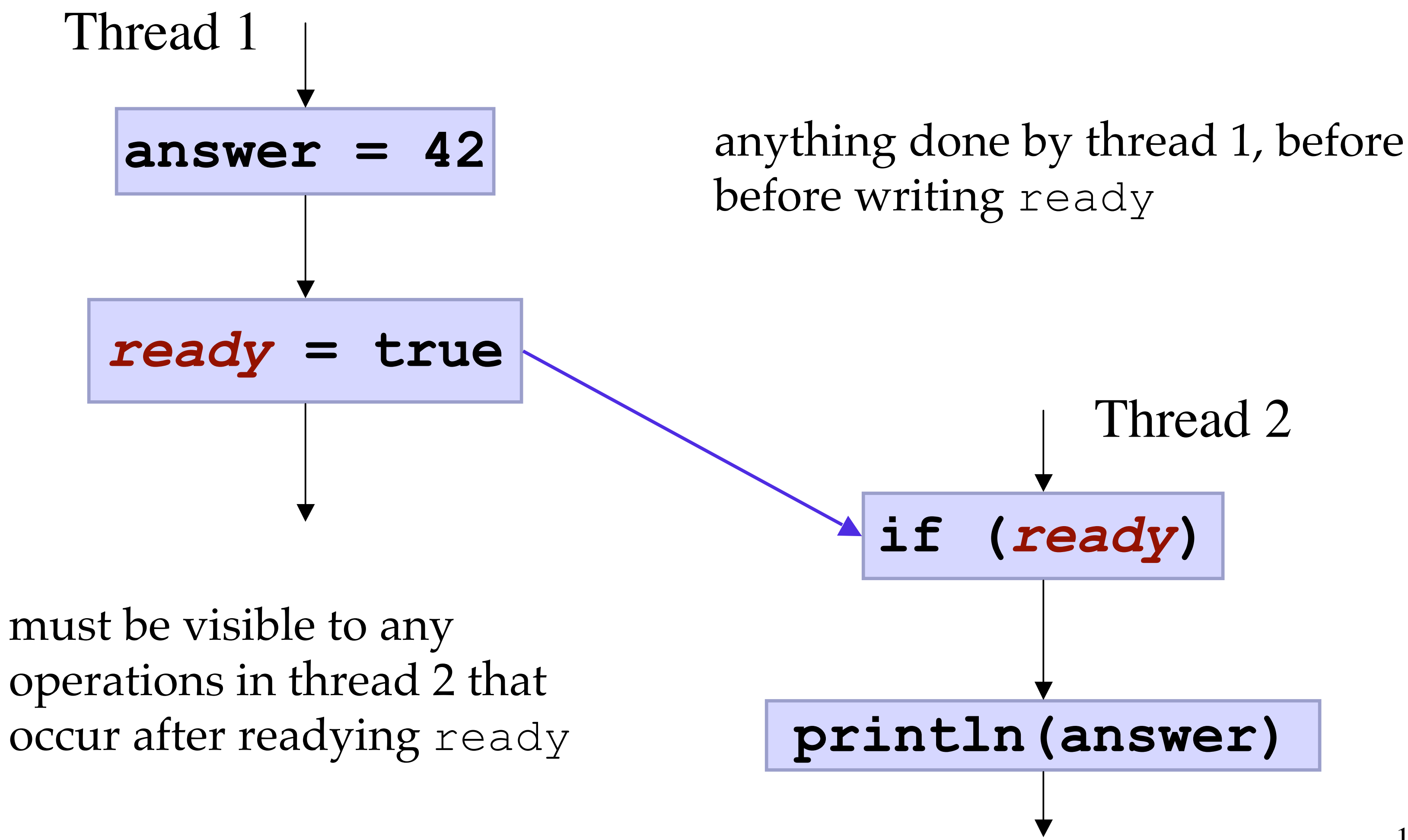
- No compiler optimizations
  - Can't hoist read out of loop
  - reads/writes go directly to memory
- Reads/writes of volatile are sequentially consistent and can not be reordered
  - but access to volatile and non-volatile variables can be reordered – makes volatiles much less useful
- Reads/writes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



Semantics of correctly synchronized programs

Correct Sync => SC behavior

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?

No

- Program is correctly synchronized
- Behavior is not SC

Definition of Correct Sync

- If, in all SC executions
  - all conflicting memory accesses
  - are ordered by union of program order
  - and synchronization edges
- program is correctly synchronized

Other issues

- One data race shouldn’t kill semantics in the rest of the program
- Sarita went over this issue, so we won’t repeat it

Semantics of incorrectly synchronized programs

Incorrect synchronization

- Incorrectly synchronized program must have well defined semantics
  - Much other older 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

Consider

Initially, $x = y = 0$

Thread 1	Thread 2
$r1 = x$	$r2 = y$
$y = r1$	$x = r2$

Can this result in $r1 = r2 = 42$?

A reference is a permissions token

- Code should not be able to forge reference to a private object
  - Even in the presence of a data race
- Case less clear for integers, doubles, etc.
  - but still seems compelling
- Values should not come out of thin air

Reasonable transformations and optimizations can lead to very strange behavior

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 $r1 = r2 = 1$?

Yes

- All stores to x and y are of constants 0 or 1
- therefore r1 and r2 are non-negative
- therefore if guards are true
- therefore writes can be moved early

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?

UPC example (old model)

Thread 1

iteration 1

  x = 1

  a[1] = x

iteration 2

  x = 2

  a[2] = x

iteration 3

  x = 3

  a[3] = x

Thread 2

x = 4

x = 5

Not allowed in UPC

  a[1] = 5

  a[2] = 4

  a[3] = 5

UPC requires $<_1$ to be a total order over in thread 1 and all writes by other threads

CRF example

Thread 1

iteration 1

$x = 1$

$a[1] = x$

iteration 2

$x = 2$

$a[2] = x$

$x = 3$

Thread 2

$x = 4$

Thread 3

$x = 5$

Thread 4

iteration 1

$x = 1$

$b[1] = x$

iteration 2

$x = 2$

$c[2] = x$

$x = 3$

Not allowed in CRF

$a[1] = 5$

$a[2] = 4$

$b[1] = 4$

$b[2] = 5$

CRF requires threads 1 and 4 to agree on the order in which $x = 4$ and $x = 5$ occur

Do we care?

- Loop reversal could have produced the behavior seen in UPC/CRF examples
  - in UPC example, reverse all but last iteration
    - last iteration might be peeled to preserve final value of x
  - In CRF, reverse loop in thread 1 but not in thread 4

Formalizing It…

Actions

- Only actions we concern ourselves with are interthread actions
  - actions you would see if standing at the interface between processor and memory
- Actions are labeled with
  - kind of action (read, write, volatile read, volatile write, lock, unlock)
  - thread that performed the action
  - variable accessed
  - value written/read

Execution consists of

- Set of actions
- For each thread, a total order over all actions by that thread (thread sequence order or program order)
- Synchronization order, a total order over all synchronization actions

Consistency checks

- Intrathread semantics
  - For each thread, the program would generate the actions of that thread in given program order
    - taking the value seen by each read as a given
- For each thread *t*, program order of synchronization actions by *t* is consistent with overall synchronization order

Synchronization Edges

- Synchronization edge from each release to each matching acquire that occurs later in synchronization order
  - volatile write matches all later volatile reads of same volatile variable
  - unlock matches all later locks of same monitor

Initial actions

- There are also a set of initial writes that initialize all variables to their default value
- Also synchronization edges from all initial writes to first action in each thread

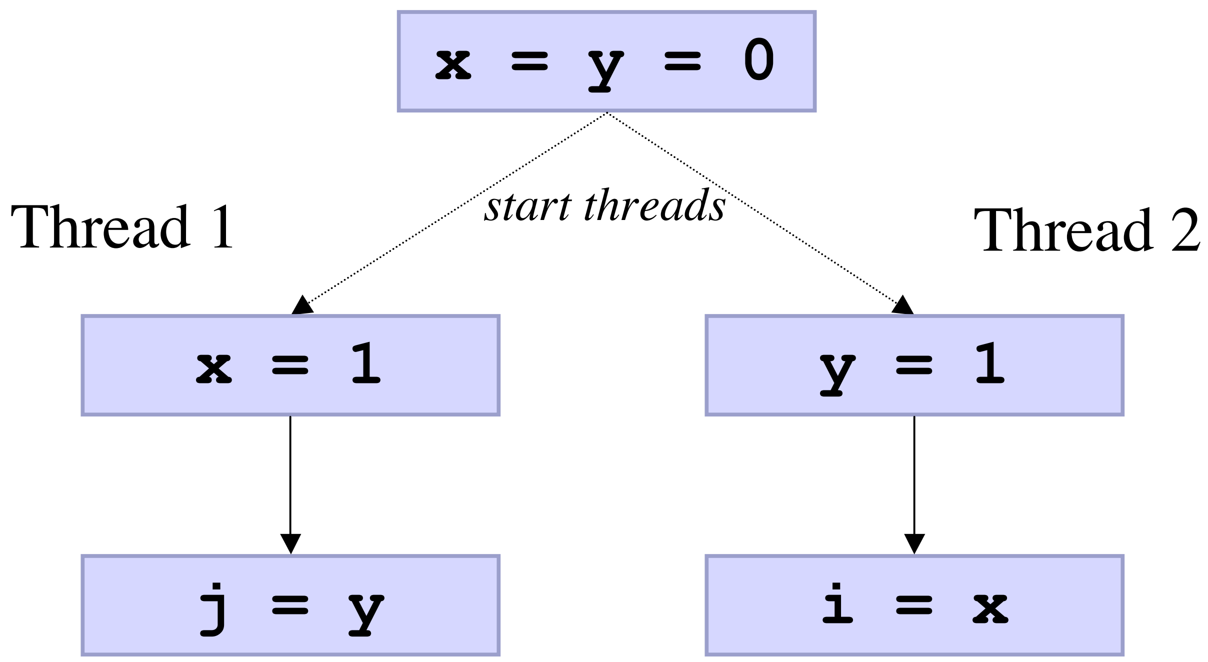
Happens-Before Order

- A partial order over actions
- Happens before order is transitive closure of synchronization edges and program order

Happens-Before Consistency

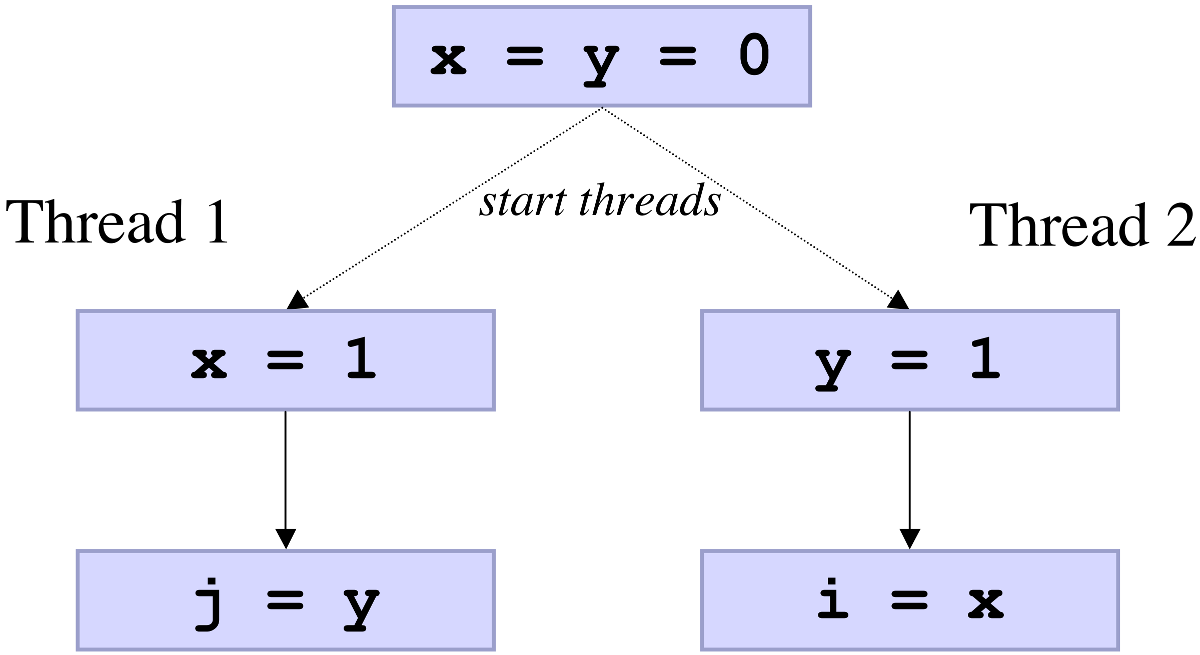
- First pass at a memory model
- A read *r* is *not* allowed to see a write *w* to the same variable *v* if
  - *r* hb w or
  - exists another write *w′* to *v* such that *w* hb *w′* hb *r*
- otherwise, *r* may see *w*

Simple Example



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

Yes



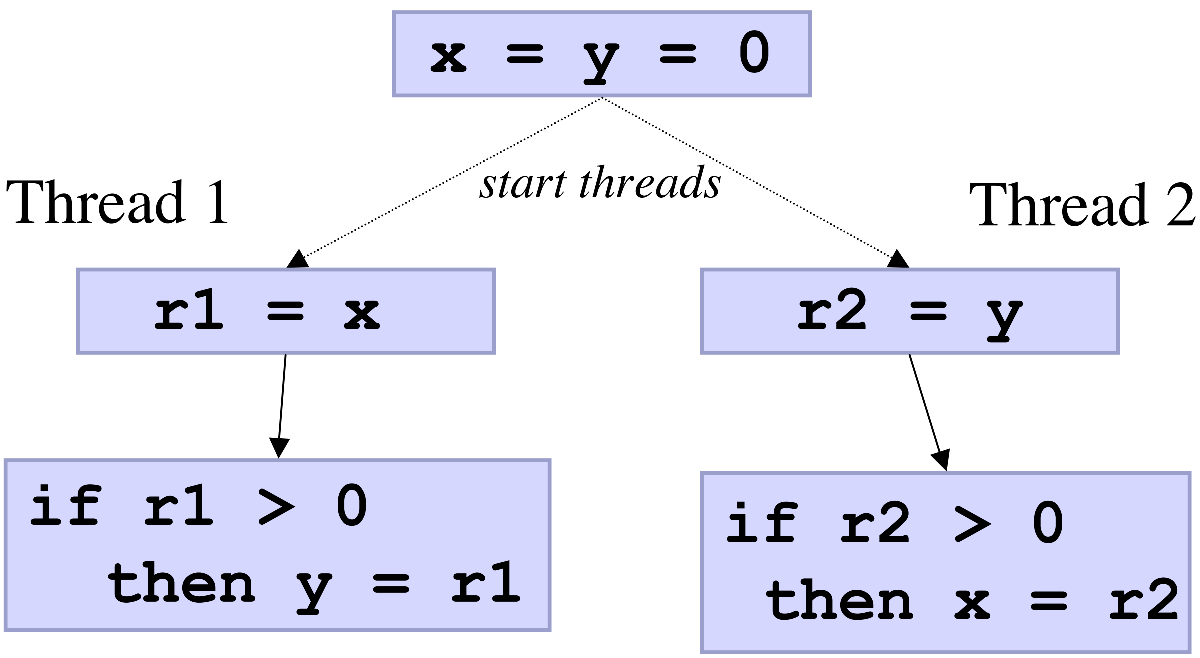
Each Read is *allowed* to see the initial write (as well as the writes of 1)

Not bad as a memory model

- Interesting to compare with UPC model
- Make appropriate adjustments
  - synchronization edges from each synchronization action to all later synchronization actions
- I think this is strictly weaker than UPC model

Problem

- Allows us to violate CS => SC
- *r1* == *r2* == 42 is a possible result of the following program



Problem

- Simply a set of actions
  - an arbitrary fixed point

- Self-consistent
- But no idea of what could have caused them or how they could have been generated
  - not a least fixed point

What is missing?

Causality

- We must be able to understand why each action was allowed to occur
  - was justified
- Need to avoid circularities
  - don't want to justify x via y, and justify y via x
- All actions occur in a *justification order*
  - A total order, not bound by program order
  - But consistent with synchronization order

Alternative names

- We liked the term *causal order*
  - but that name was already taken
- *Execution order* isn't bad
  - but suggests an execution model we don't require

Justification order

- The actions before x in the justification order
  - must ensure that x takes place
  - if x is a read, need not ensure what value is seen by x
- However, a read x can only see writes that come before it in the justification order
  - write seen must also be hb-consistent

Simple case

- What if the justification order is consistent with program order?
- No additional justification needed
- Weaker than SC
  - because a read doesn't have to see most recent write
- But doesn't handle all of the cases we need

Example

Thread 1	Thread 2
r1 = x	r2 = y
y = 1	x = 1

Can we observe r1 == r2 == 1?

- If justification order is consistent with program order, either r1 = x or r2 = y must come first
  - can't see r1 == r2 == 1

Use dependences?

- Idea: allow justification order to be reordered, except where prohibited by control and data dependences
  - Doesn’t work
  - Control and data dependences determined by semantics
    - which are determined by the memory model
    - thus using them to define the memory model would result in an ill-defined circular definition
  - Compiler can do dependence-breaking transformations
    - based on the semantics

Prescient actions

- An action x is prescient if there exists a action y that occurs later in the justification order such that y hb x

Back to an Example

Initially, x = y = 0

Thread 1	Thread 2
r1 = x	r2 = y
if r1 >= 0 then	if r2 >= 0 then
y = 1	x = 1

Can this result in r1 = r2 = 1?

Justification order:

y = 1; r2 = y(1); x = 1; r1 = x (1)

Justification of Prescient Actions

- After executing □, we want to perform a prescient action *x*
- Show if you continue execution without performing any (more) prescient actions
- action *x* will always occur

Strictly weaker than dependences

- This approach is strictly weaker than allowing actions to be reordered except where prevented by dependences

Is This too *Strict*?

- Action may only be performed presciently if it happens in *all* executions
- Memory model allows many executions/behaviors
- Compiler transformations and/or VM design may rule out some possible behaviors
- If this guarantees an action will occur
  - that wasn't guaranteed to occur previously
  - we need to be able to perform it early

Transformations that eliminate behaviors

- Redundant Read Elimination
- Compiler Thread Scheduling
- Atomic reads of longs and doubles
- Fairness guarantees

Example

Initially, x = 0, y = 0

Thread 1	Thread 2	Thread 3
r1 = x	r3 = y	x = 2
r2 = x	x = r3	
if r1 == r2 then		
y = 1		

Can we see r1 == r2 == r3 == 1?

- To get this behavior, we need to perform y = 1 presciently
- But y=1 doesn't occur in all executions
  - doesn't occur when r1 == 2 and r2 == 0, or when r1 == 0 and r2 == 2

We need to allow this behavior

Initially, x = 0, y = 0

Thread 1	Thread 2	Thread 3
r1 = x	r3 = y	x = 2
r2 = x	x = r3	
if r1 == r2 then		
y = 1		

Can we see r1 == r2 == r3 == 1?

- Replace r2 = x with r2 = r1
- Replace r1 == r2 with true
  - removing control dependence
- Move write of y early

Resulting Thread 1

```
 y = 1
r1 = x
r2 = r1
```

Forbidden executions

- An execution *E* can be shown legal
  - if there exists a set of forbidden executions
  - that allow justification of all prescient actions in *E*
  - Bunch of consistency constraints to make the forbidden executions sensible
    - an execution can be forbidden only because
      - a read would see a different value
      - a different scheduling decision would be made

Difference between Sarita's model and our model

- Very close agreement on litmus tests
  - formalisms are somewhat close
- One essential difference
  - What is out of thin air?

Agreement on some cases (4)

Initially, $x = y = 0$

Thread 1	Thread 2
$r1 = x$	$r2 = y$
$y = r1$	$x = r2$

Must not result in $r1 = r2 = 42$

Difference on others (5, 10)

Initially, x = y = z = 0

Thread 1	Thread 2	Thread 3	Thread 4
r1 = x	r2 = y	z = 1	r3 = z
y = r1	x = r2		if r3 == 1
			x = 42

Sarita’s model: does allow in

r3 == 0; r1 == r2 == 42

Manson/Pugh: doesn’t allow

r3 == 0; r1 == r2 == 42

Is (6) same as (5,10)?

Initially, x = y = 0

Thread 1	Thread 2
r1 = x	r2 = y

if (r1 == 1)
    y = 1
	if (r2 == 1)
	    x = 1

	else x = 1

Agree: can result in r1 = r2 = 1

Sarita: among statements that execute, seems to be an out-of-thin-air race, just like (5, 10)

Us: model doesn't talk about statements. The actions that occurred can be justified in order.

Argument against (5, 10)

- Profoundly disturbing (to us)
- No causality means no audit trail
  - don't buy argument that 6 is the same
- Hard to imagine debugging or trying to ensure security without causality
- Consider method that always returns a key, but also always logs it

```
Key getKey() {
      auditLog.record("Gave out key");
      return privateKey;
      }
```

Attacker writes

Initially, x = y = null, z = 0

Thread 1	Thread 2	Thread 3	Thread 4
r1 = x	r2 = y	sleep(1000);	r3 = z
y = r1	x = r2	z = 1	if r3 == 1 x = getKey()

Allows r1 == r2 == key, r3 = 0,
no log in audit trail

Core Memory Model Summary

- If you correctly synchronize your code, you get SC behavior.
- If you don't, you can get surprising results, but such results must always stem from a *causal* sequence of actions
  - may or may not be consistent with program order

Immutability and Final Field Semantics

Immutability in Java

- `final` fields are written once by bytecode, in an object's constructor, and never changed.
- This provides immutability, right?

- **Caution**: much of this is ugly. We cannot break backwards compatibility.
  - Yes, we would design it differently if we were starting over

`String` Class Example

Thread 1	Thread 2
Global.s = "/tmp/usr".substring(4);	String myS = Global.s; if (myS.equals("/tmp")) System.out.println(myS);

```
Thread 1                          Thread 2
Global.s =                        String myS = Global.s;
  "/tmp/usr".substring(4);        if (myS.equals("/tmp"))
                                    System.out.println(myS);
```

- Implementation can
  - Create `final` char array as "/tmp/usr", `final` start index of 4, `final` string length of 4
- But `offset` might not be perceived correctly or consistent by thread 2
  - offset of 0 in myS.equals("/tmp"),
  - offset of 4 during print, so it prints "/usr"
- Massive potential security hole

Goals for `final` fields (What do we need to fix?)

- Value is not intended to change
  - Compiler should never have to reload the value of it, if possible
  - In general, the semantics of `final` should impose a minimal architectural cost
- Objects that have only final fields should appear immutable, even if passed by a data race after construction

Indirect guarantees

- If a final field references write-once but non-final data
  - e.g., a final reference to an array of characters
- Reads of write-once data via final field should see correctly initialized values

Require correct construction and publication

- Lots of issues arise if object is made visible to other threads before final fields are set or construction is complete
- Programmers should strive to avoid these cases
- Fair bit of hair in model to deal with such cases
  - make sure semantics are defined
  - but don't impose implementation cost

Implementation goals

- Want additional barriers only at construction time
  - except on Alpha
- Don't want to treat them as volatile
- Keep finals in registers across synchronization and unknown function calls

Pretty Close

- At the end of a constructor, have a conceptual “freeze” of the state of the final fields
- A reference to an object is “correctly published” if it is written after the freeze.
- Writes in constructor are ordered before reads of final field done by other threads from that reference
  - as are reads transitively reached via final field

`String` Example Revisited

Thread 1	Thread 2
`Global.s =`	`String myS = Global.s;`
`  "/tmp/usr".substring(4);`	`if (myS.equals("/tmp"))`
	`  System.out.println(myS);`

- Thread 2 only accesses string length and offset after correct publication, so is guaranteed to see correct value
- Since guarantee applies transitively, char array is correctly seen, too

Complications

- Several ways to ensure that an object is correctly published
  - write during construction, use Java synchronization to ensure no other thread sees until after construction
  - write reference after construction
- If thread T1 sees an incorrectly published version of an object, thread T2 can still see a correctly published version
- Final fields set multiple times
  - e.g., via deserialization, after construction
- Can hoist reads of final fields

Can hoist reads of final fields

- If a thread sees an incorrectly published reference to *x*
- All other references to *x* are spoiled as well

```
r1 = p // incorrectly published
r2 = r1.x
r3 = q // correctly published
if (r1 == r3 && r2 == r3.x)
   // compiler should be able to eliminate r2 = r3.x
```

Implementation

- *May* need a memory barrier at end of constructor
  - e.g., don't need them for thread local objects or objects with no final fields
- No memory barriers or reordering constraints for reads of final fields
  - except on Alpha
- Can perform aggressive optimizations of final fields
  - compiler can treat them as constant

What to Take Away

- Don't allow other threads to see an object until it is fully constructed/initialized
  - including deserialization, which occurs after construction
- If you do this, final fields will appear immutable to other threads