

CONCURRENCY: SEQUENTIAL CONSISTENCY, DATA RACES, AND DYNAMIC ANALYSES

Lecture by Rohan Padhye

17-355/17-665/17-819: Program Analysis

Material from past lectures by Jonathan Aldrich, based in large part on slides by John Erickson, Stephen Freund, Madan Musuvathi, Mike Bond, and Man Cao

Lecture Goals

- What is sequential consistency and why is it important?
- What is a data race, and what is data-race-free execution?
- Subtleties of data races and memory models
 - Why taking advantage of “harmless races” is almost certainly a bad idea
- Lockset analysis for data race detection
- Happens-before based data race detection

SEQUENTIAL CONSISTENCY

First things First

Assigning Semantics to Concurrent Programs

int X = F = 0;

X = 1;
F = 1;

t = F;
u = X;

- What does this program mean?
- Sequential Consistency [Lamport '79]
Program behavior = set of its thread interleavings

Recall: Semantics of WHILE_{||} from midterm

$$\frac{\langle E, S_1 \rangle \rightarrow \langle E', S'_1 \rangle}{\langle E, S_1; S_2 \rangle \rightarrow \langle E', S'_1; S_2 \rangle} \text{ small-seq-congruence}$$

$$\frac{}{\langle E, \text{skip}; S_2 \rangle \rightarrow \langle E, S_2 \rangle} \text{ small-seq}$$

$$\frac{\langle E, S_1 \rangle \rightarrow \langle E', S'_1 \rangle}{\langle E, S_1 \parallel S_2 \rangle \rightarrow \langle E', S'_1 \parallel S_2 \rangle} \text{ small-par-congruence-1}$$

$$\frac{\langle E, S_2 \rangle \rightarrow \langle E', S'_2 \rangle}{\langle E, S_1 \parallel S_2 \rangle \rightarrow \langle E', S_1 \parallel S'_2 \rangle} \text{ small-par-congruence-2}$$

$$\frac{}{\langle E, \text{skip} \parallel \text{skip} \rangle \rightarrow \langle E, \text{skip} \rangle} \text{ small-par-skip}$$

Exercise 1:

```
int X = F = 0;
```

```
X = 1;  
F = 1;
```

```
t = F;  
u = X;
```

- What are the possible final values for variables `t` and `u` after running this program, assuming sequential consistency?

Sequential Consistency Explained

int X = F = 0; // F = 1 implies X is initialized

X = 1;
F = 1;

t = F;
u = X;

X = 1;

X = 1;

X = 1;

t = F;

t = F;

t = F;

F = 1;

t = F;

t = F;

u = X;

X = 1;

X = 1;

t = F;

F = 1;

u = X;

X = 1;

u = X;

F = 1;

u = X;

u = X;

F = 1;

F = 1;

F = 1;

u = X;

t=1, u=1

t=0, u=1

t=0, u=1

t=0, u=0

t=0, u=1

t=0, u=1

t=1 implies u=1

Naturalness of Sequential Consistency

- Sequential Consistency provides two crucial abstractions
- Program Order Abstraction
 - Instructions execute in the order specified in the program
 $A ; B$
means “Execute A and then B”
- Shared Memory Abstraction
 - Memory behaves as a global array, with reads and writes done immediately
- We implicitly assume these abstractions for sequential programs
 - As we will see, we can only rely on these abstractions under certain conditions in a concurrent context

WHAT IS A DATA RACE ?

- The term “data race” is often overloaded to mean different things
- Precise definition is important in designing a tool

Data Race

- Two accesses *conflict* if
 - they access the same memory location, and
 - at least one of them is a write

Write X – Write X

Write X – Read X

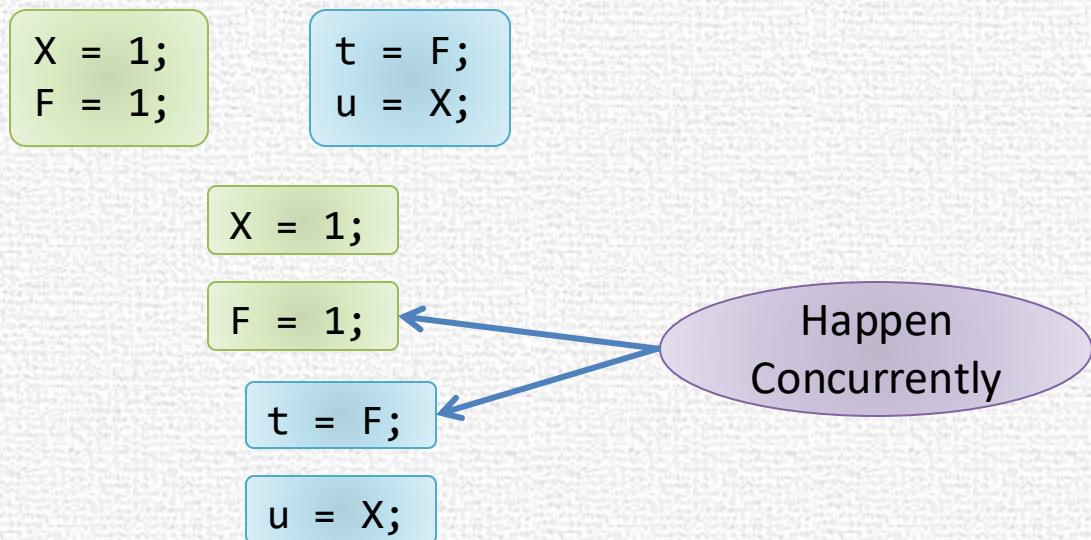
Read X – Write X

Read X – Read X

- A data race is a pair of conflicting accesses **that happen concurrently**

“Happen Concurrently”

- A and B happen concurrently if they occur in different threads, and
- there exists a sequentially consistent execution in which they occur one after the other

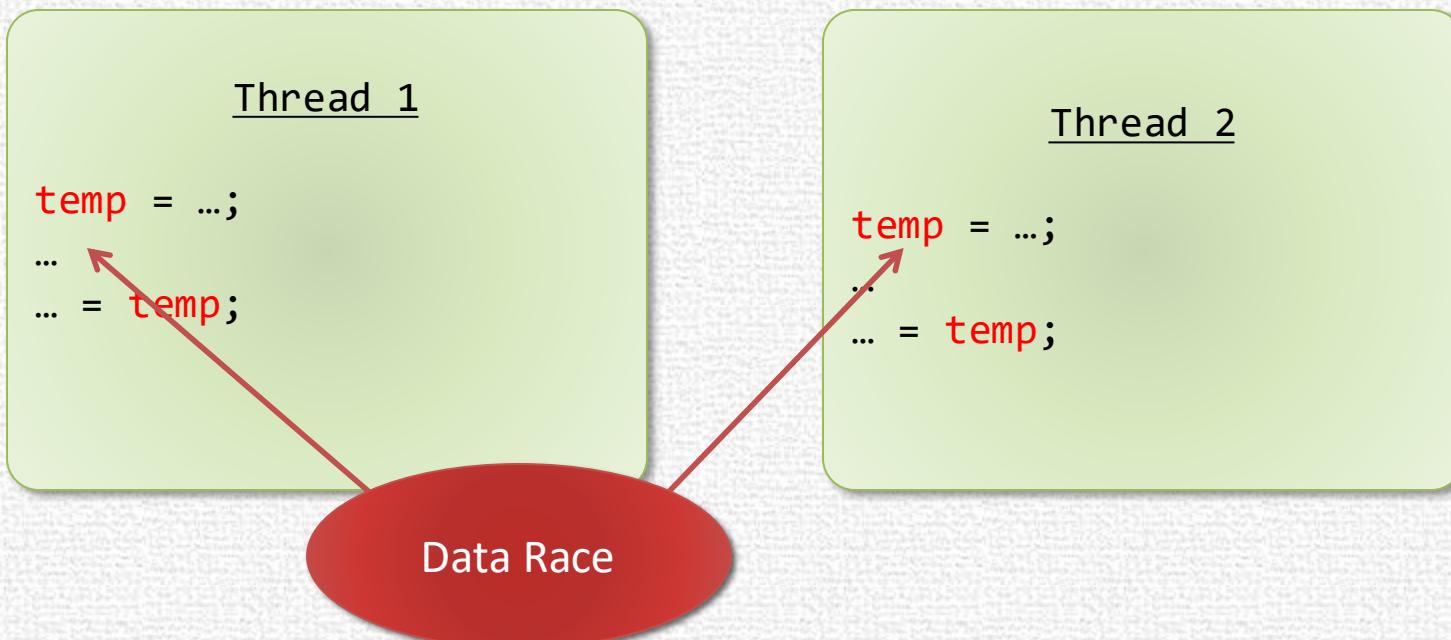


Data races are almost always no good

- What are some consequences of a data race, even when assuming sequential consistency?

Unintended Sharing

- Threads accidentally sharing data that should not be global
- *Solution:* Change allocation (e.g., stack var or static thread-local)



Atomicity Violation

- When code that is meant to execute *atomically* (that is, perform a single undivisible operation) suffers interference from some other thread
- *Solution:* Surround critical sections with locks

Thread 1

```
void Bank::Update(int a)
{
    int t = bal;
    bal = t + a;
}
```

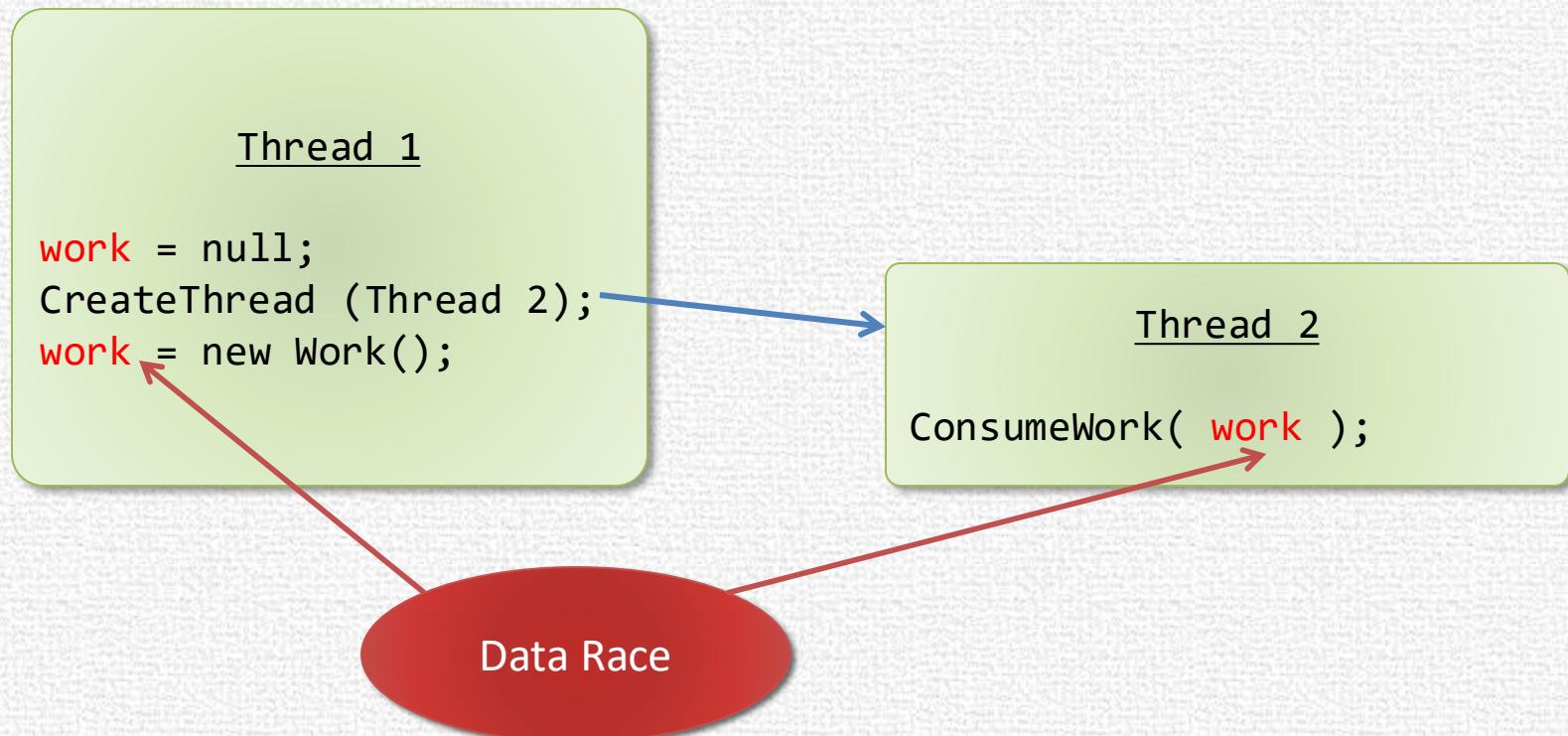
Thread 2

```
void Bank::Withdraw(int a)
{
    int t = bal;
    bal = t - a;
}
```

Data Race

Ordering Violation

- Incorrect signaling between a producer and a consumer
- *Solution:* Reorder operations or use synchronization (e.g., signals)



But,....

- How do you think "locks" are implemented?
- Atomic compare-and-swap (CAS)

```
AcquireLock(lock){  
    while (!CAS (lock, 0, 1)) {}  
}
```

```
ReleaseLock(lock) {  
    lock = 0;  
}
```

Data Race ?

Acceptable Concurrent Conflicting Accesses

- Implementing synchronization (such as locks) usually requires concurrent conflicting accesses to shared memory
- Innovative uses of shared memory
 - Fast reads
 - Double-checked locking
 - Lazy initialization
 - Setting dirty flag
 - ...
- Need mechanisms to distinguish these from erroneous conflicts

Solution: Programmer Annotation

- Programmer explicitly annotates variables as “synchronization”
 - Java – volatile keyword
 - C++ – std::atomic<> types

Data Race

- Two accesses *conflict* if
 - they access the same memory location, and
 - at least one of them is a write
- A data race is a pair of concurrent conflicting accesses to locations **not annotated as synchronization**
 - Recall: “Concurrent” means there exists a sequentially consistent execution in which they happen one after the other
- Equivalent definition: a pair of conflicting accesses where one doesn’t **happen before** the other
 - Program order
 - Synchronization order
 - Acquire/release, wait-notify, fork-join, volatile read/write

Exercise 2: Is there a data race?

If so, on what variable(s)?

Initially:

```
int data = 0;  
boolean flag = false;
```

T1:

```
data = 42;  
flag = true;
```

T2:

```
if (flag)  
    t = data;
```

Is there a data race?

Initially:

```
int data = 0;  
boolean flag = false;
```

T1:

```
data = 42;  
flag = true;
```

T2:

```
if (flag)  
    t = data;
```

Consider regular compiler transformations/optimizations

Before:

```
data = 42;  
flag = true;
```

After:

```
flag = true;  
data = 42;
```

Possible behavior

Initially:

```
int data = 0;  
boolean flag = false;
```

T1:

```
flag = true;
```

T2:

```
if (flag)  
    t = data;
```

data = 42;

Consider regular compiler transformations/optimizations

Before:

```
if (flag)  
    t = data;
```

After:

```
t2 = data;  
if (flag)  
    t = t2;
```

Possible behavior

Initially:

```
int data = 0;  
boolean flag = false;
```

T1:

```
data = 42;  
flag = true;
```

T2:

```
t2 = data;  
  
if (flag)  
    t = t2;
```

How do we fix this?

Initially:

```
int data = 0;  
boolean flag = false;
```

T1:

```
data = 42;  
flag = true;
```

T2:

```
if (flag)  
    t = data;
```

Using “synchronized” keyword in Java

Initially:

```
int data = 0;  
boolean flag = false;
```

T1:

```
data = ...;  
synchronized (m) {  
    flag = true;  
}
```

T2:

```
boolean f;  
synchronized (m) {  
    f = flag;  
}  
if (f)  
    ... = data;
```

... Implemented via locks

Initially:

```
int data = 0;  
boolean flag = false;
```

T1:

```
data = ...;  
acquire(m);  
flag = true;  
release(m);
```

T2:

```
boolean f;  
acquire(m);  
f = flag;  
release(m);  
if (f)  
    ... = data;
```

Happens-before
relationship

Using “volatile” keyword in Java

Initially:

```
int data = 0;  
volatile boolean flag = false;
```

T1:

```
data = ...;  
flag = true;
```

T2:

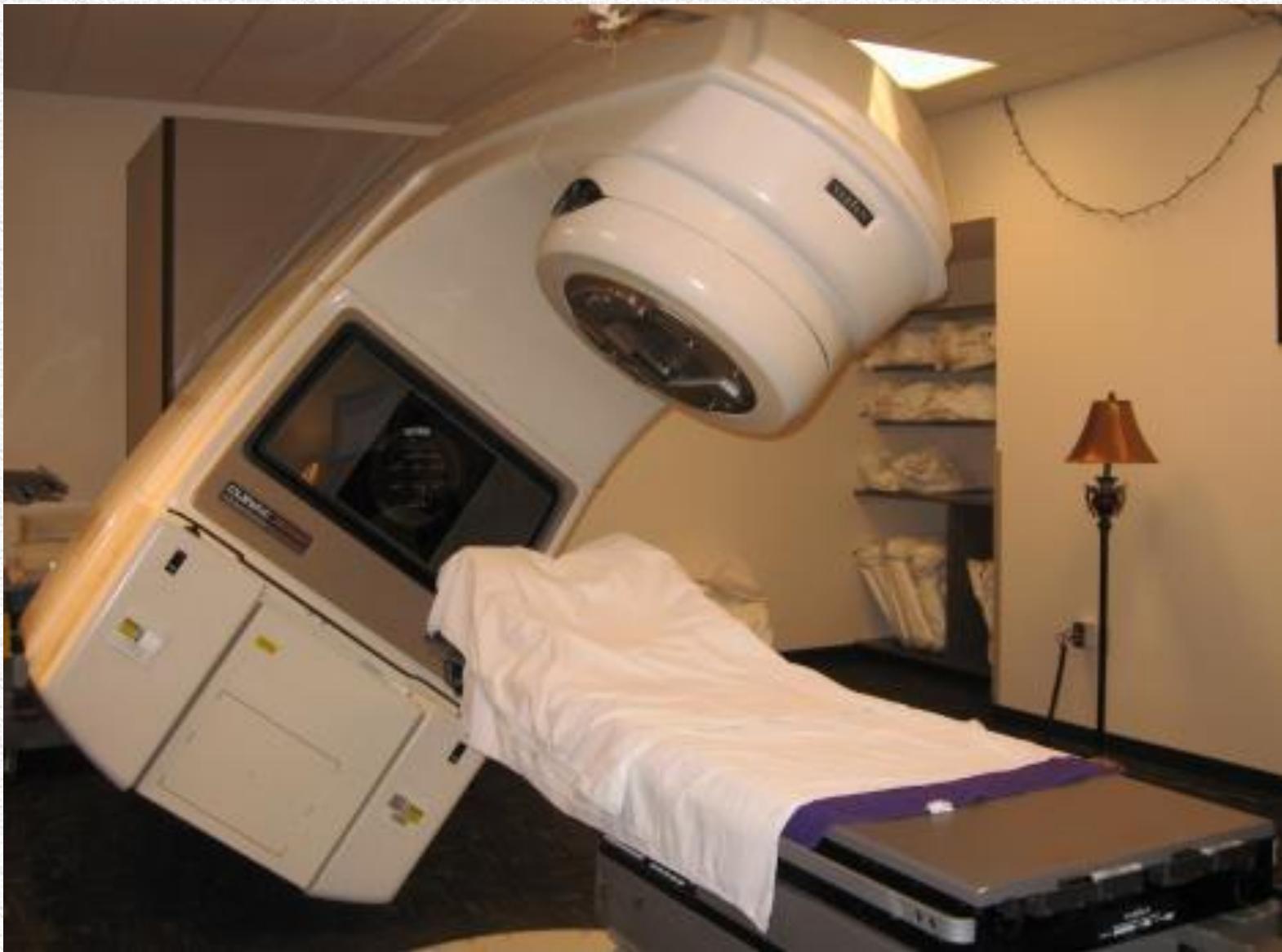
```
if (flag)  
    ... = data;
```

*Happens-before
relationship*

Data Race vs Race Conditions

- Data Races != Race Conditions
 - Confusing terminology
- Race Condition
 - Any timing error in the program
 - Due to events, device interaction, thread interleaving, ...
 - **Race conditions can be very bad!**





Data Race vs Race Conditions

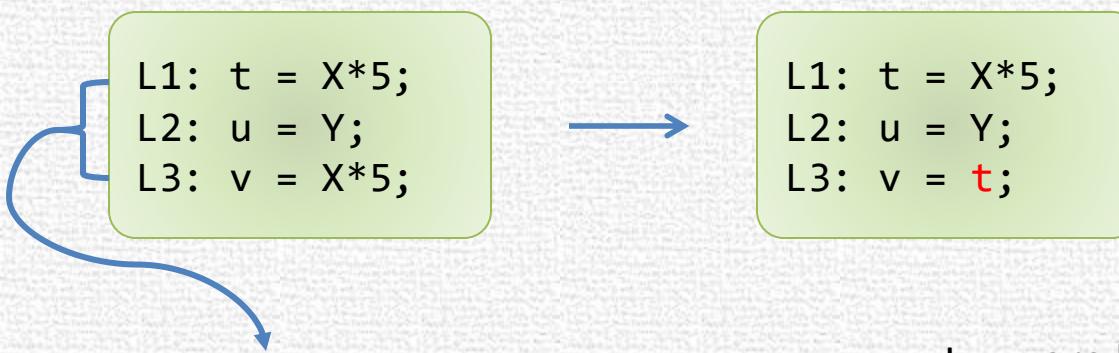
- Data Races != Race Conditions
 - Confusing terminology
- Race Condition
 - Any timing error in the program
 - Due to events, device interaction, thread interleaving, ...
 - **Race conditions can be very bad!**
- Data races are neither sufficient nor necessary for a race condition
 - Data race is a good **symptom** for a race condition

DATA-RACE-FREEDOM SIMPLIFIES LANGUAGE SEMANTICS

Advantage of Eliminating All Data Races

- Defining semantics for concurrent programs becomes surprisingly easy
- In the presence of compiler and hardware optimizations

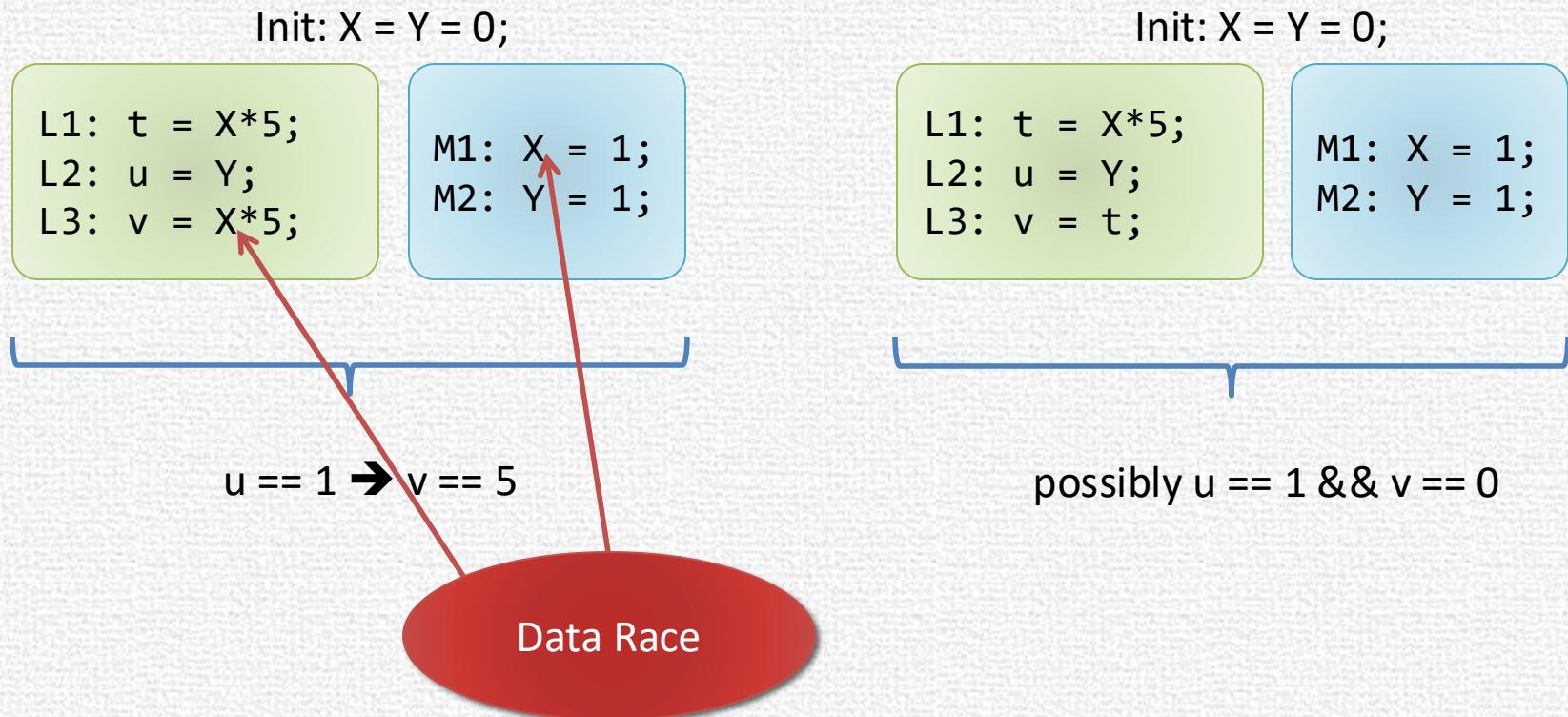
Can A Compiler Do This?



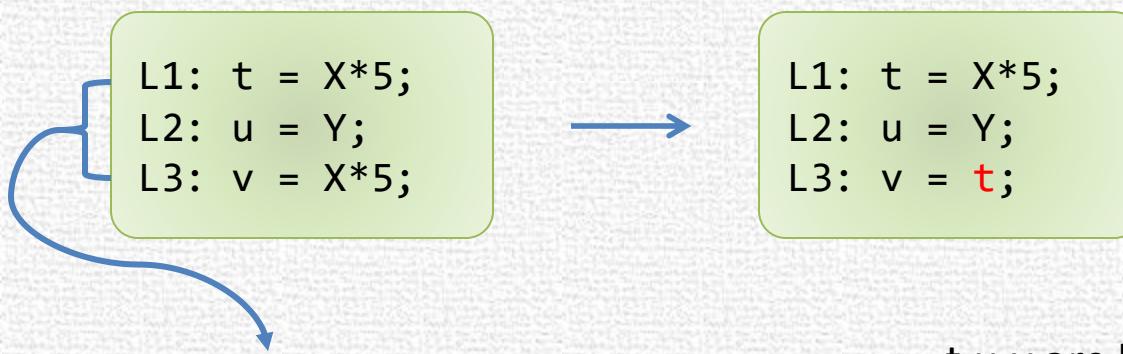
OK for sequential programs
if X is not modified between L1 and L3

t, u, v are local variables
 X, Y are possibly shared

Can Break Sequential Consistent Semantics



Can A Compiler Do This?



OK for sequential programs
if X is not modified between L1 and L3

t, u, v are local variables
 X, Y are possibly shared

OK for concurrent programs
if there is no data race on X or Y

Key Observation [Adve& Hill '90]

- Many sequentially valid (compiler & hardware) transformations also preserve sequential consistency
- Provided the program is data-race free
- Forms the basis for modern C++, Java semantics
 - data-race-free → sequential consistency
 - otherwise → weak/undefined semantics

DATA RACE DETECTION

Overview of Data Race Detection Techniques

- Static data race detection
- Dynamic data race detection
 - Lock-set
 - Happen-before
 - Race Fuzzing

Static Data Race Detection

- Advantages:
 - Reason about all inputs/interleavings
 - No run-time overhead
 - Adapt well-understood static-analysis techniques
 - Annotations to document concurrency invariants
- Example Tools:
 - RCC/Java type-based
 - ESC/Java "functional verification"
(theorem proving-based)

Static Data Race Detection

- Advantages:
 - Reason about all inputs/interleavings
 - No run-time overhead
 - Adapt well-understood static-analysis techniques
 - Annotations to document concurrency invariants
- Disadvantages of static:
 - Undecidable...
 - Tools produce “false positives” or “false negatives”
 - May be slow, require programmer annotations
 - May be hard to interpret results

Dynamic Data Race Detection

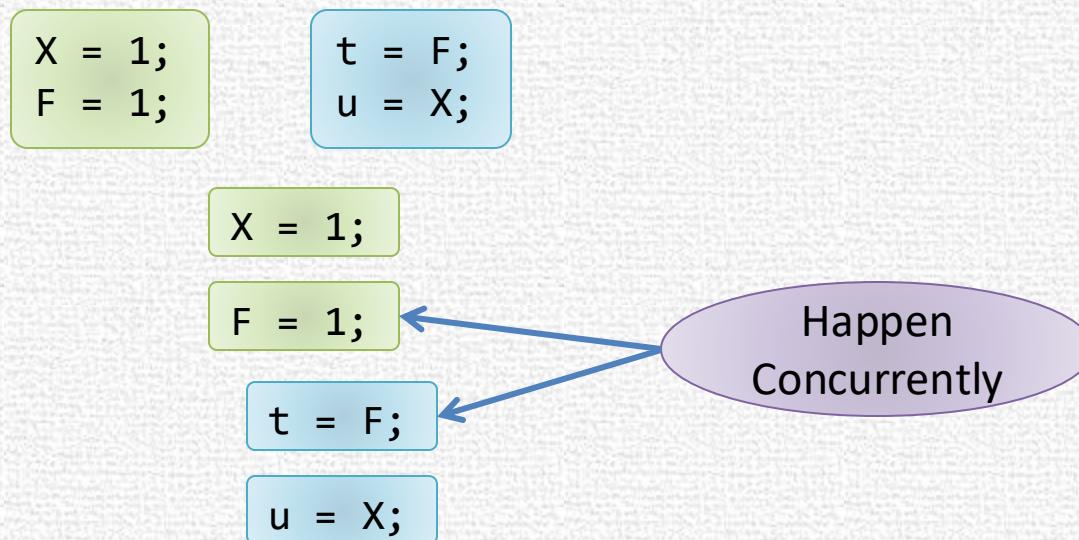
- Advantages
 - Can avoid “false positives”
 - No need for language extensions or sophisticated static analysis
- Disadvantages
 - Run-time overhead (5-20x for best tools)
 - Memory overhead for analysis state
 - Reasons only about observed executions
 - sensitive to test coverage
 - (some generalization possible...)

Tradeoffs: Static vs Dynamic

- Coverage
 - generalize to additional traces?
- Soundness
 - all reported warnings are actually races
- Completeness
 - every actual data race is reported
- Overhead
 - run-time slowdown
 - memory footprint
- Programmer overhead

Definition Refresh

- A data race is a pair of concurrent conflicting accesses to unannotated locations (i.e. not locks or volatile variables)



- Problem for dynamic data race detection
 - Very difficult to catch the two accesses executing concurrently

Solution

- Lockset
 - Infer data races through violation of locking discipline
- Happens-before
 - Infer data races by generalizing a trace to a set of traces with the same happens-before relation

LOCKSET ALGORITHM

Eraser [Savage et.al. '97]

Lockset Algorithm Overview

- Checks a sufficient condition for data-race-freedom
- Consistent locking discipline
 - Every data structure is protected by a single lock
 - All accesses to the data structure made while holding the lock
- Example:

```
// Remove a received packet  
AcquireLock( RecvQueueLk );  
pkt = RecvQueue.RemoveTop();  
ReleaseLock( RecvQueueLk );
```

```
... // process pkt
```

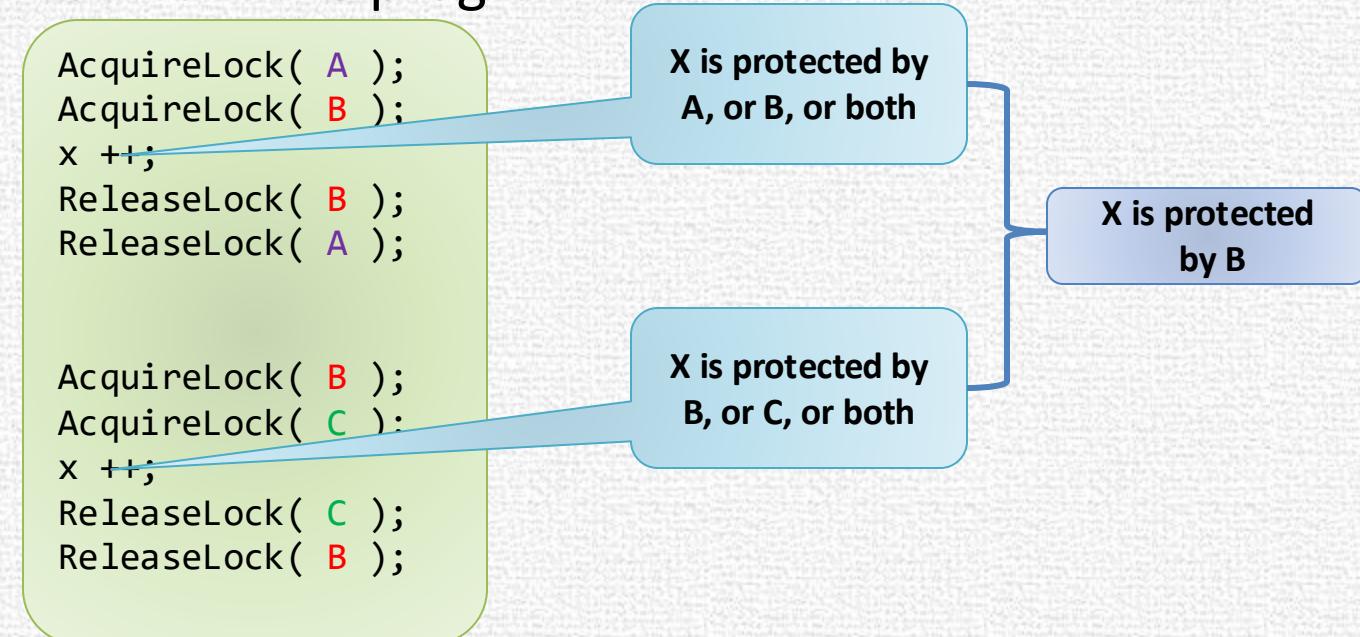
```
// Insert into processed  
AcquireLock( ProcQueueLk );  
ProcQueue.Insert(pkt);  
ReleaseLock( ProcQueueLk );
```

RecvQueue is
consistently protected
by RecvQueueLk

ProcQueue is
consistently protected
by ProcQueueLk

Inferring the Locking Discipline

- How do we know which lock protects what?
 - Asking the programmer is cumbersome
- Solution: Infer from the program



LockSet Algorithm

- Two data structures:
 - $\text{LocksHeld}(t)$ = set of locks held currently by thread t
 - Initially set to Empty
 - $\text{LockSet}(x)$ = set of locks that could potentially be protecting x
 - Initially set to the universal set
- When thread t acquires lock l
 - $\text{LocksHeld}(t) = \text{LocksHeld}(t) \cup \{l\}$
- When thread t releases lock l
 - $\text{LocksHeld}(t) = \text{LocksHeld}(t) - \{l\}$
- When thread t accesses location x
 - $\text{LockSet}(x) = \text{LockSet}(x) \cap \text{LocksHeld}(t)$
 - Report “data race” when $\text{LockSet}(x)$ becomes empty

LockSet Algorithm

- No warnings → no data races on the current execution
 - The program followed consistent locking discipline in this execution
- Warnings does not imply a data race
 - Thread-local initialization

```
// Initialize a packet  
pkt = new Packet();  
pkt.Consumed = 0  
  
AcquireLock( SendQueueLk );  
SendQueue.Enqueue(pkt);  
ReleaseLock( SendQueueLk );
```

```
// Process a packet  
AcquireLock( SendQueueLk );  
pkt = SendQueue.Top();  
pkt.Consumed = 1;  
ReleaseLock( SendQueueLk );
```

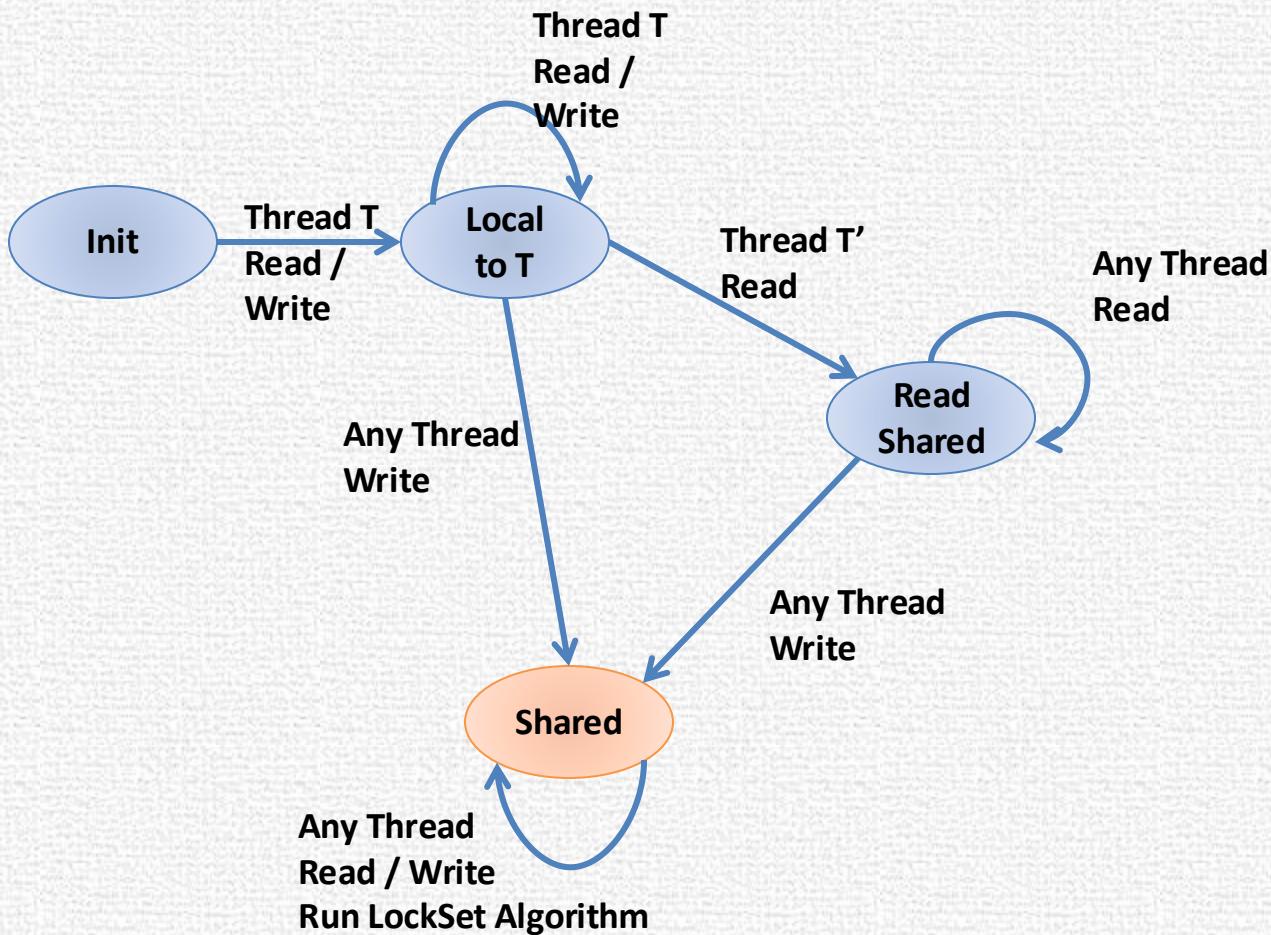
LockSet Algorithm

- No warnings → no data races on the current execution
 - The program followed consistent locking discipline in this execution
- Warnings does not imply a data race
 - Object read-shared after thread-local initialization

```
A = new A();  
A.f = 0;  
  
// publish A  
globalA = A;
```

```
f = globalA.f;
```

Maintain A State Machine Per Location



LockSet Algorithm

- State machine misses some data races

```
// Initialize a packet  
pkt = new Packet();  
pkt.Consumed = 0;
```

```
AcquireLock( WrongLk );  
pkt = SendQueue.Top();  
pkt.Consumed = 1;  
ReleaseLock( WrongLk );
```

```
// Process a packet  
AcquireLock( SendQueueLk );  
pkt = SendQueue.Top();  
pkt.Consumed = 1;  
ReleaseLock( SendQueueLk );
```

LockSet Algorithm

- Does not handle locations consistently protected by different locks during a particular execution

```
// Remove a received packet  
AcquireLock( RecvQueueLk );  
pkt = RecvQueue.RemoveTop();  
ReleaseLock( RecvQueueLk );
```

```
... // process pkt
```

```
// Insert into processed  
AcquireLock( ProcQueueLk );  
ProcQueue.Insert(pkt);  
ReleaseLock( ProcQueueLk );
```

Pkt is protected by
RecvQueueLk

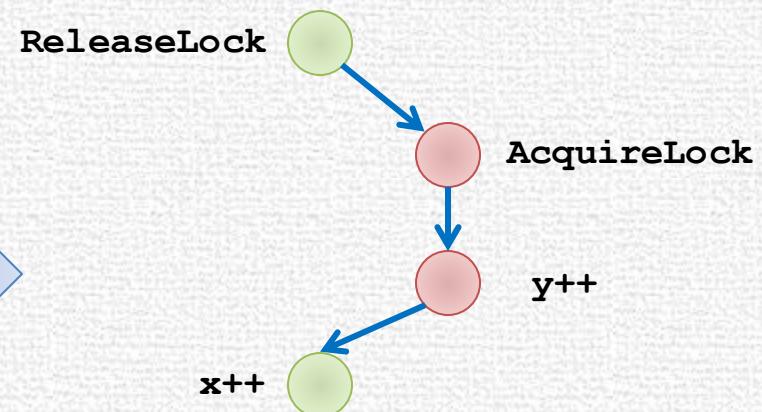
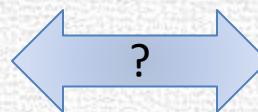
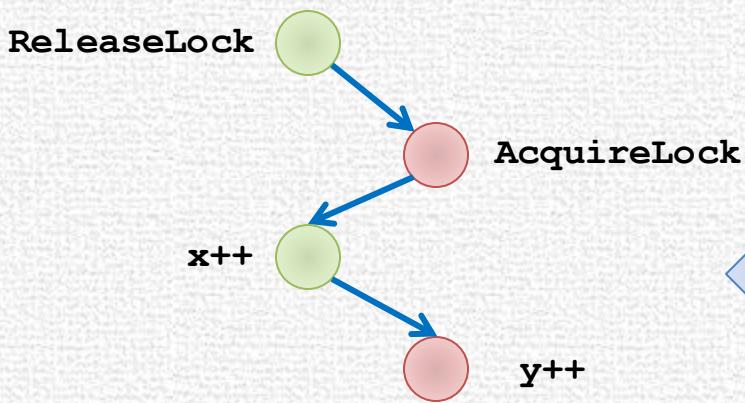
Pkt is thread local

Pkt is protected by
ProcQueueLk

HAPPENS-BEFORE

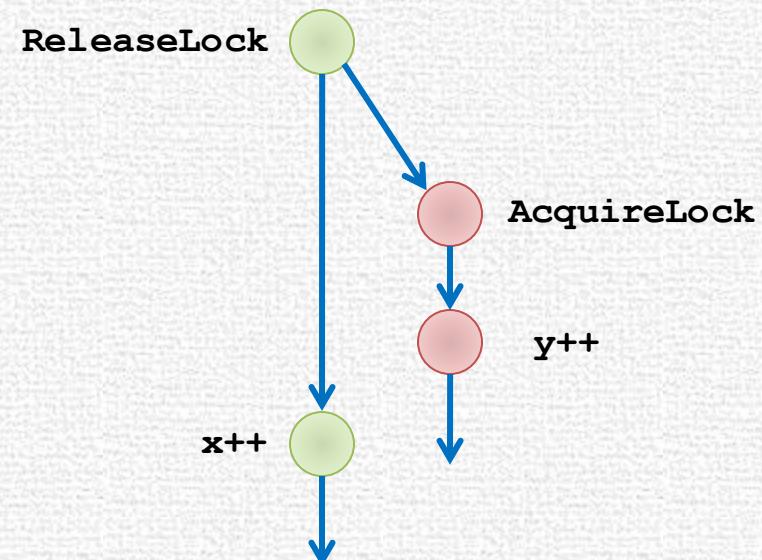
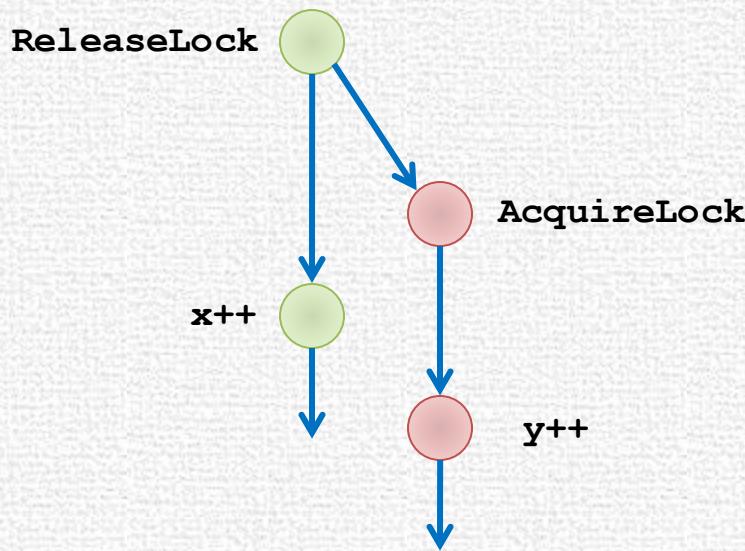
Happens-Before Relation [Lamport '78]

- A concurrent execution is a partial-order determined by communication events
- The program cannot “observe” the order of concurrent non-communicating events



Happens-Before Relation [Lamport '78]

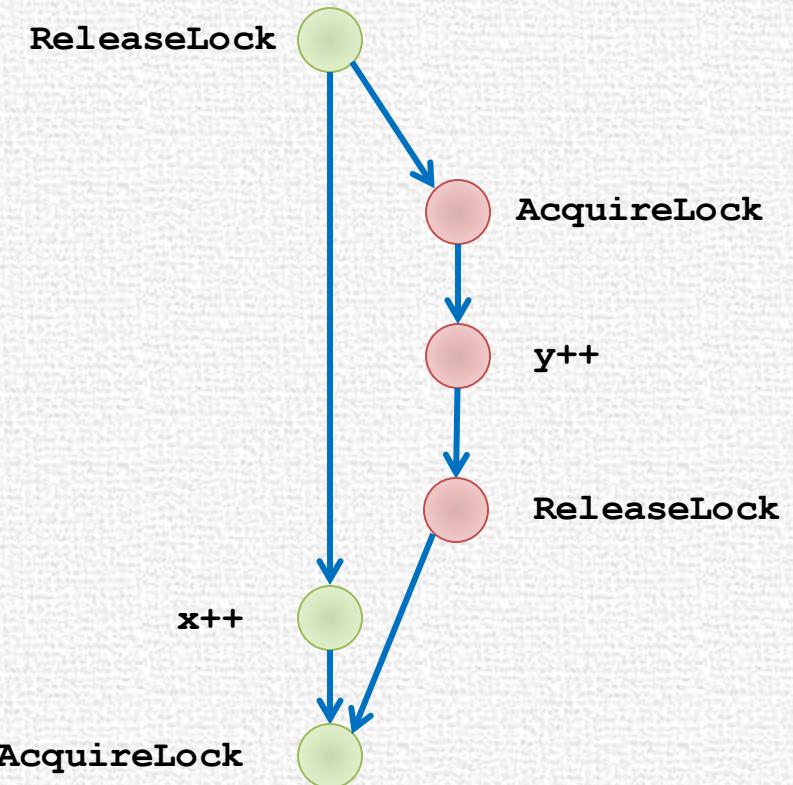
- A concurrent execution is a partial-order determined by communication events
- The program cannot “observe” the order of concurrent non-communicating events



- Both executions form the same happens-before relation

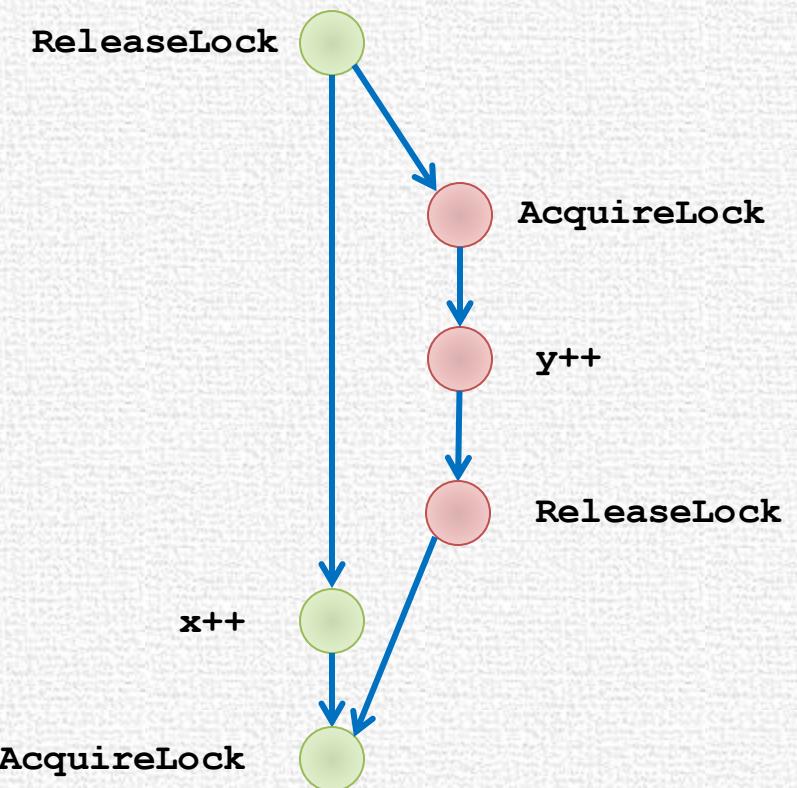
Constructing the Happens-Before Relation

- Program order
 - Total order of thread instructions
- Synchronization order
 - Total order of accesses to the same synchronization



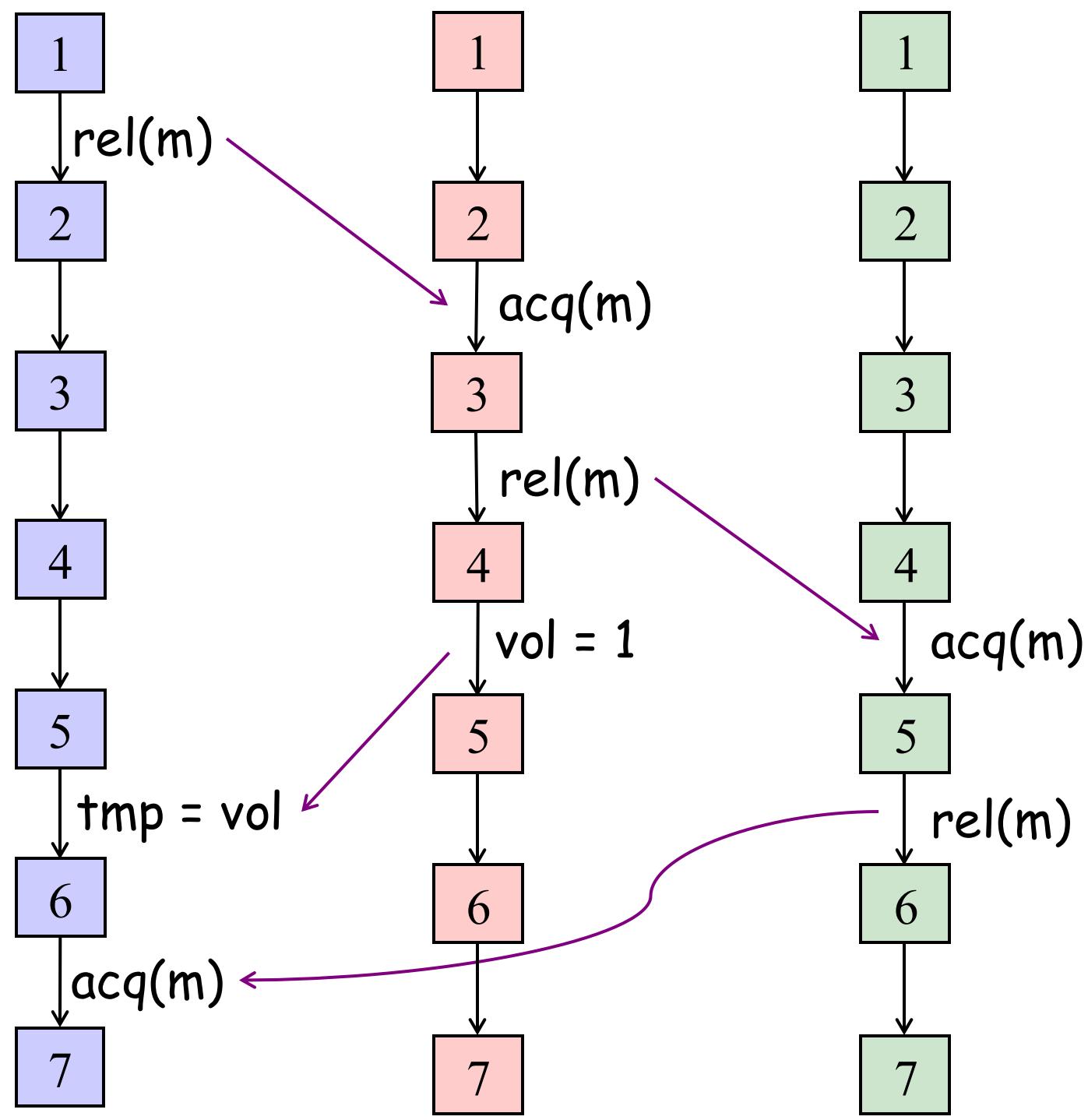
Happens-Before Relation And Data Races

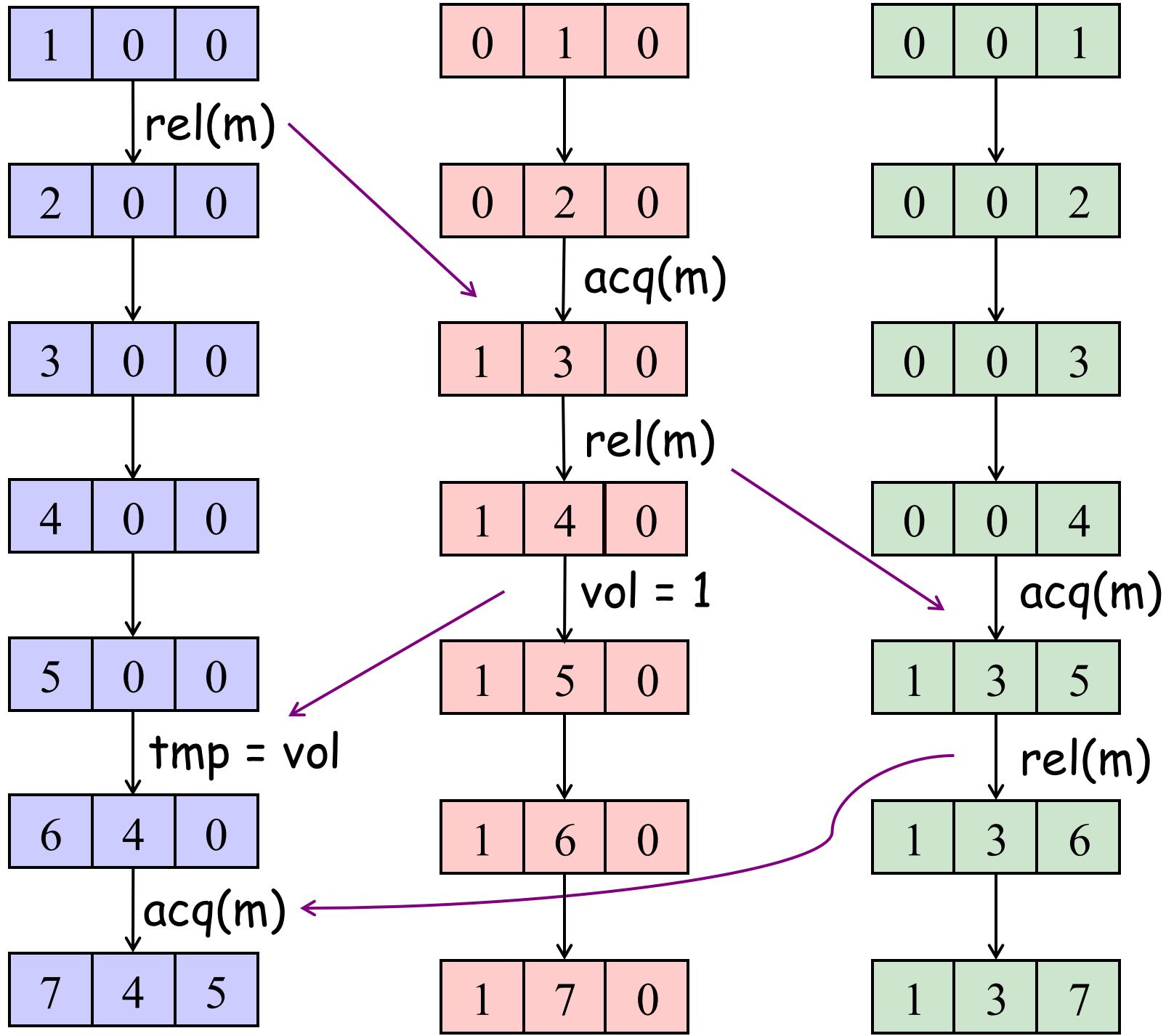
- If all conflicting accesses are ordered by happens-before
 - → data-race-free execution
 - → All linearizations of partial-order are valid program executions
-
- If there exists conflicting accesses not ordered
 - → a data race

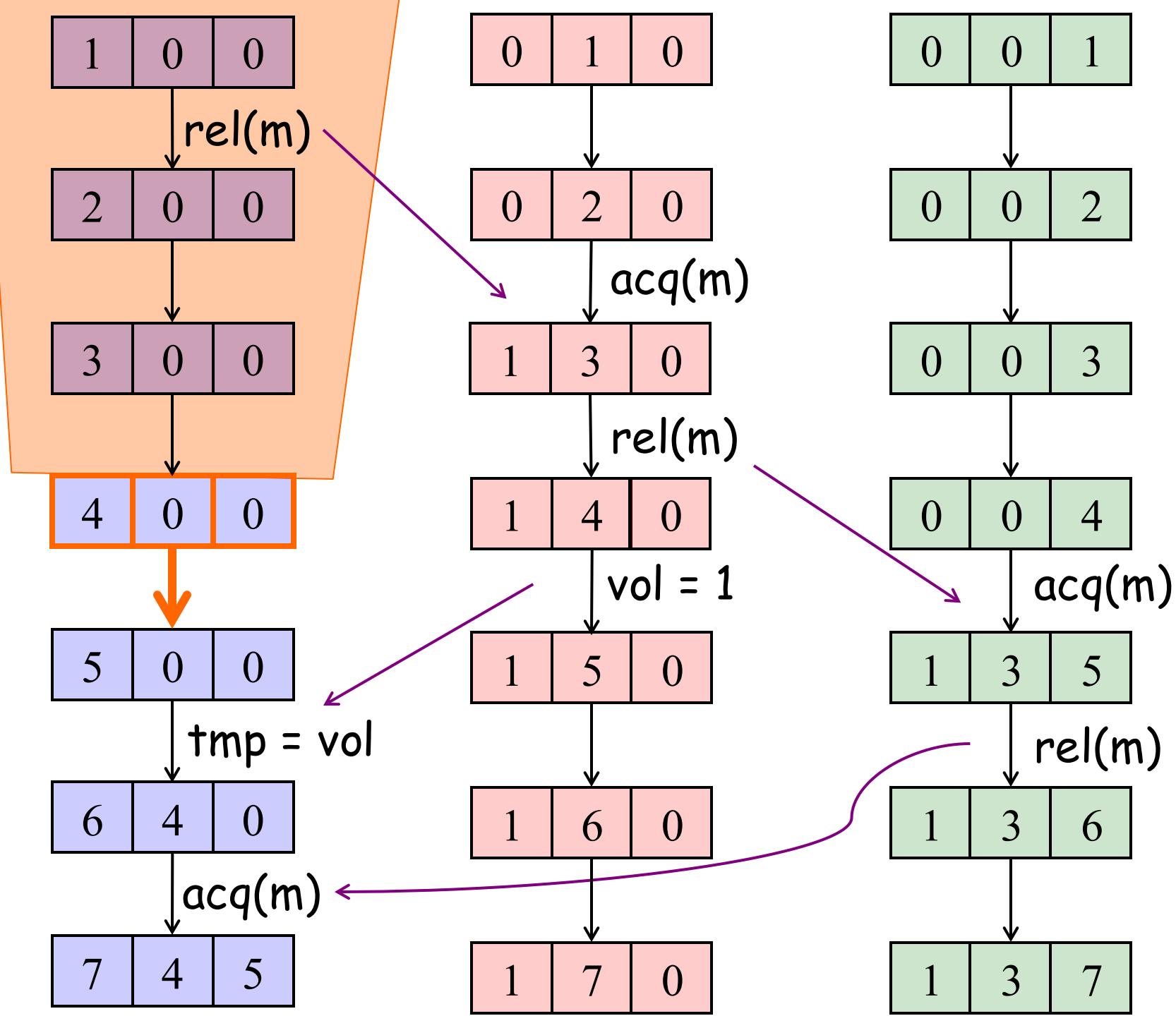


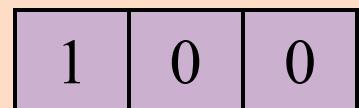
IMPLEMENTING HAPPENS-BEFORE ANALYSES

Precise Happens-Before

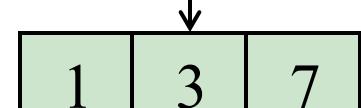
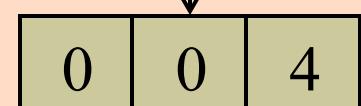
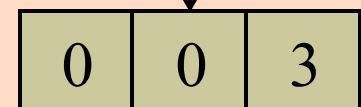
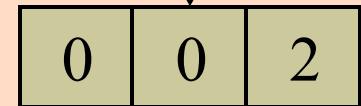
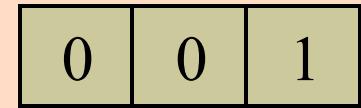
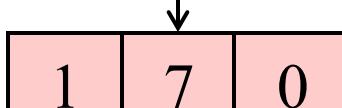
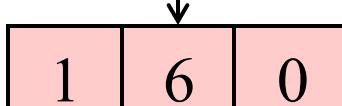
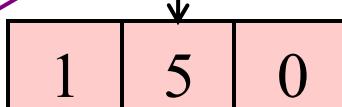
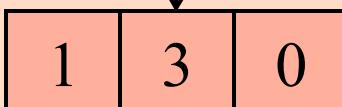
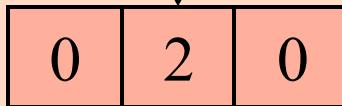
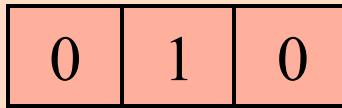
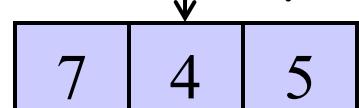
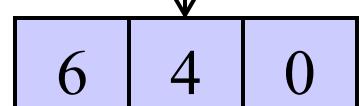
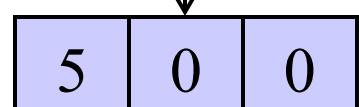
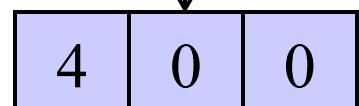
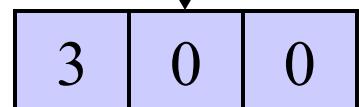
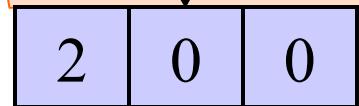








rel(m)



rel(m)

acq(m)

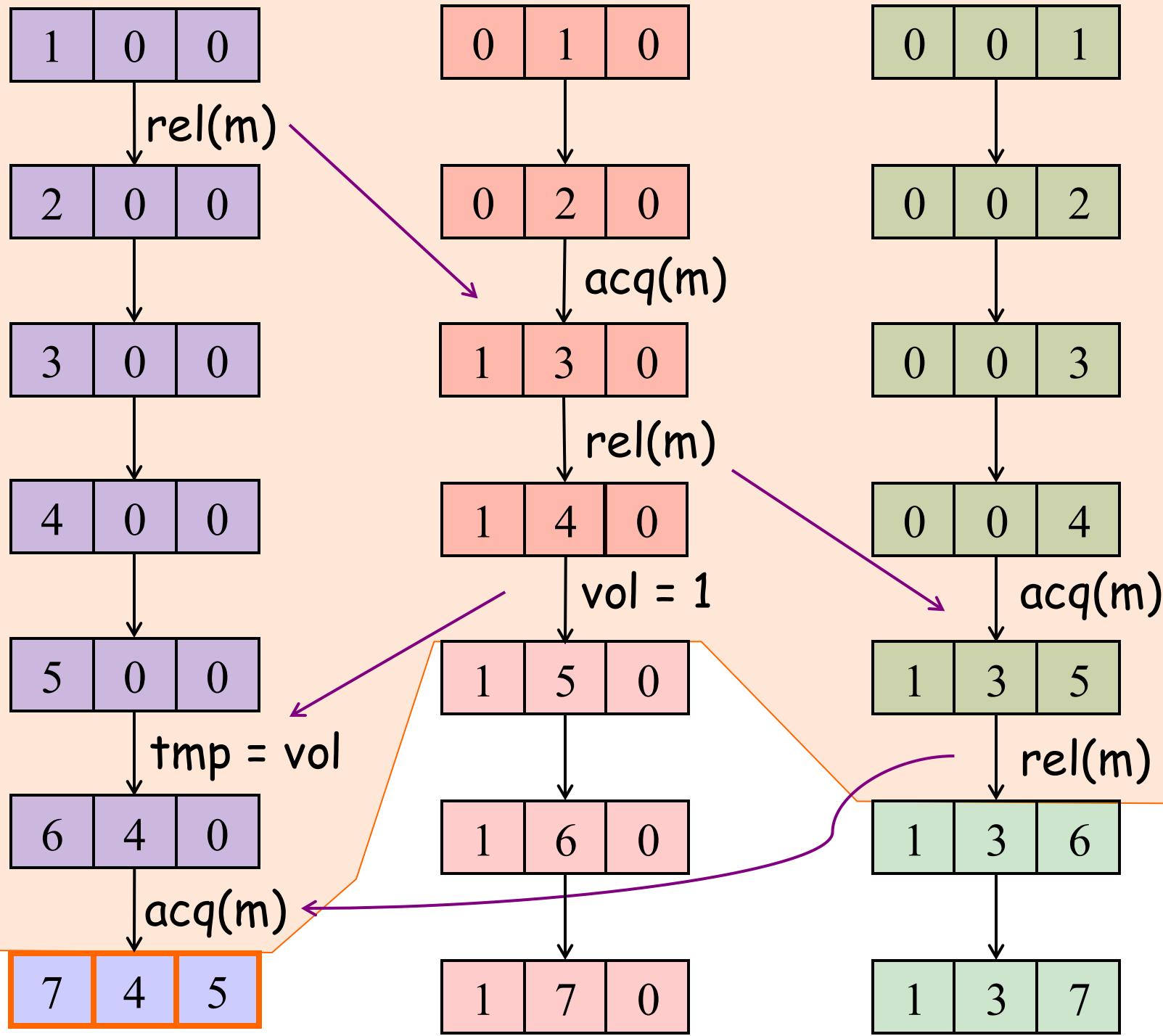
rel(m)

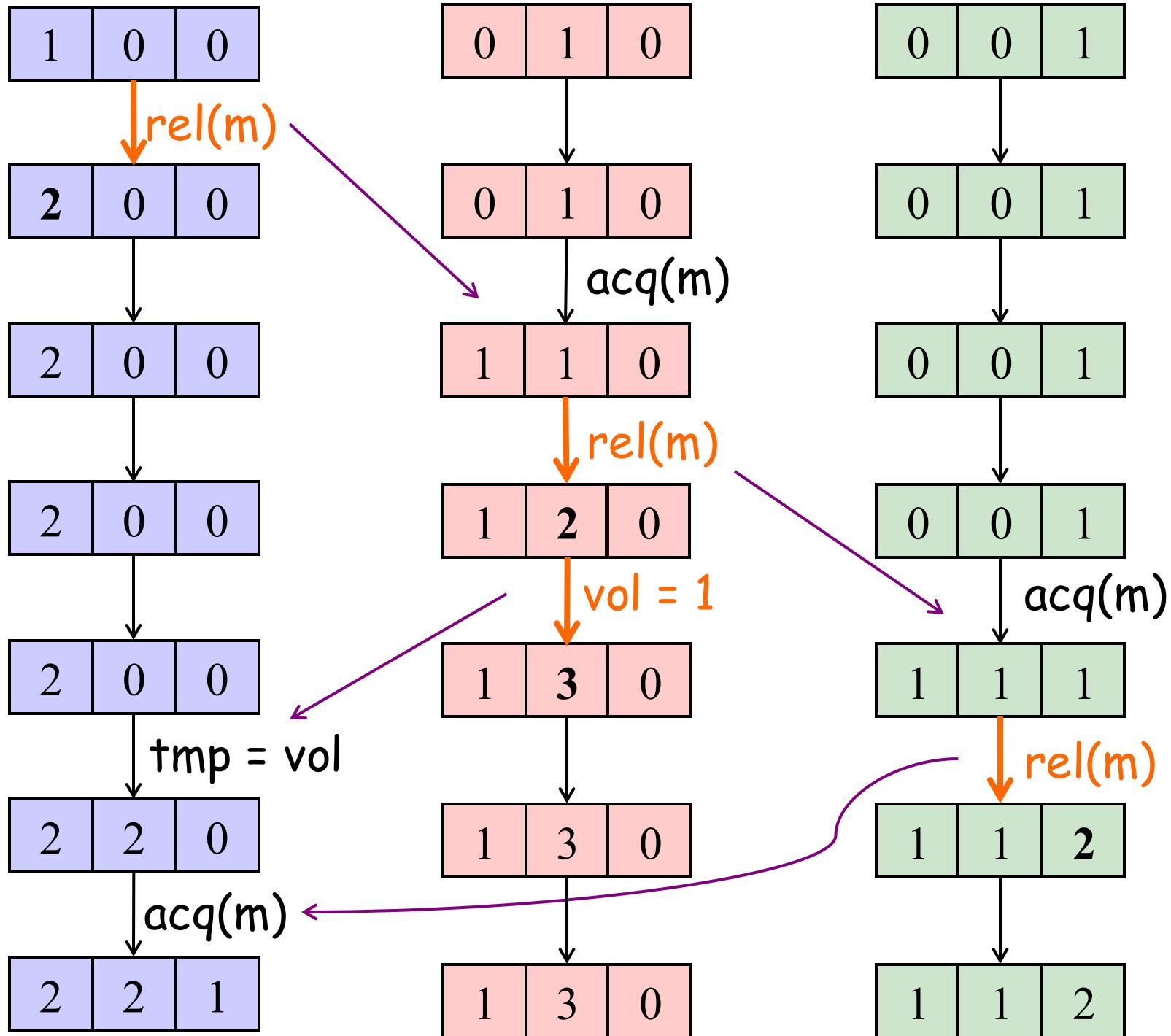
vol = 1

tmp = vol

acq(m)

rel(m)





Exercise on vector clocks and partial ordering

- $VC = [t_1, t_2, \dots, t_N]$
- What is $VC_a \sqsubseteq VC_b$?
- What is $VC_a \sqcup VC_b$?
- What are sufficient and necessary conditions for there to be a data race between two accesses having vector clocks VC_a and VC_b ?

VC_A

4	1
---	---

A B

 VC_B

2	8
---	---

A B

 L_m

2	1
---	---

A B

 W_x

3	0
---	---

A B

 R_x

0	1
---	---

A B

A's local time

B's local time

VC_A

4	1
---	---

A B

 VC_B

2	8
---	---

A B

 L_m

2	1
---	---

A B

 W_x

3	0
---	---

A B

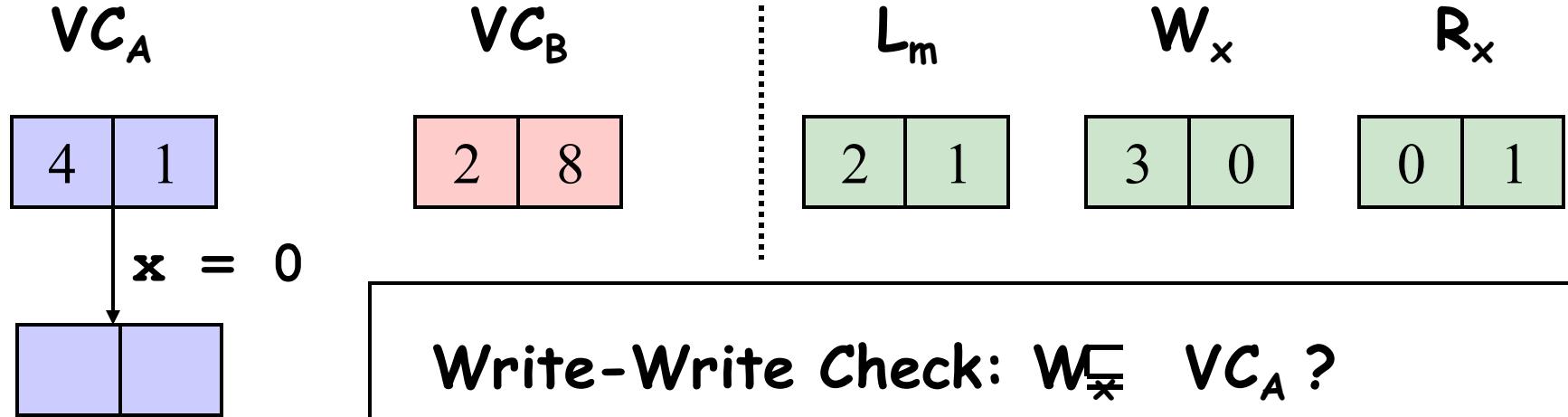
 R_x

0	1
---	---

A B

B-steps with B-time ≤ 1
happen before
A's next step





Write-Write Check: $W_x \sqsubseteq VC_A$?

3	0
---	---

 \sqsubseteq

4	1
---	---

 ? Yes

Read-Write Check: $R_x \sqsubseteq VC_A$?

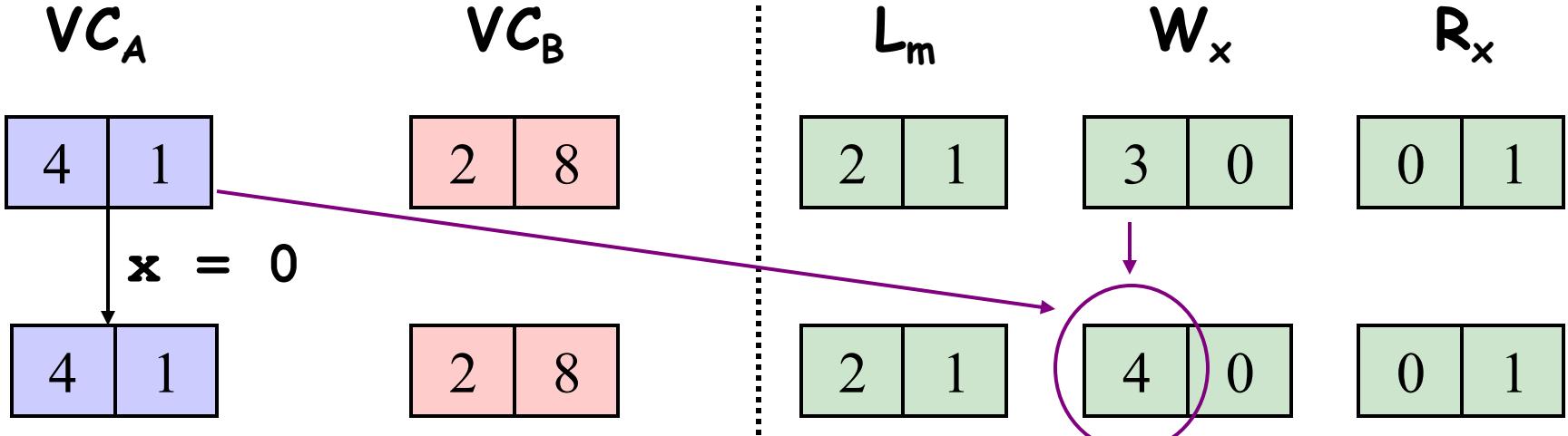
0	1
---	---

 \sqsubseteq

4	1
---	---

 ? Yes

$O(n)$ time



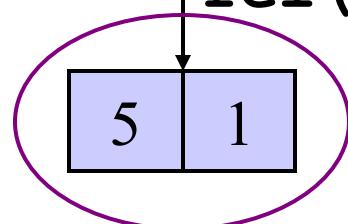
VC_A

4	1
---	---

$$x = 0$$

4	1
---	---

$rel(m)$



VC_B

2	8
---	---

L_m

2	1
---	---

W_x

3	0
---	---

R_x

0	1
---	---

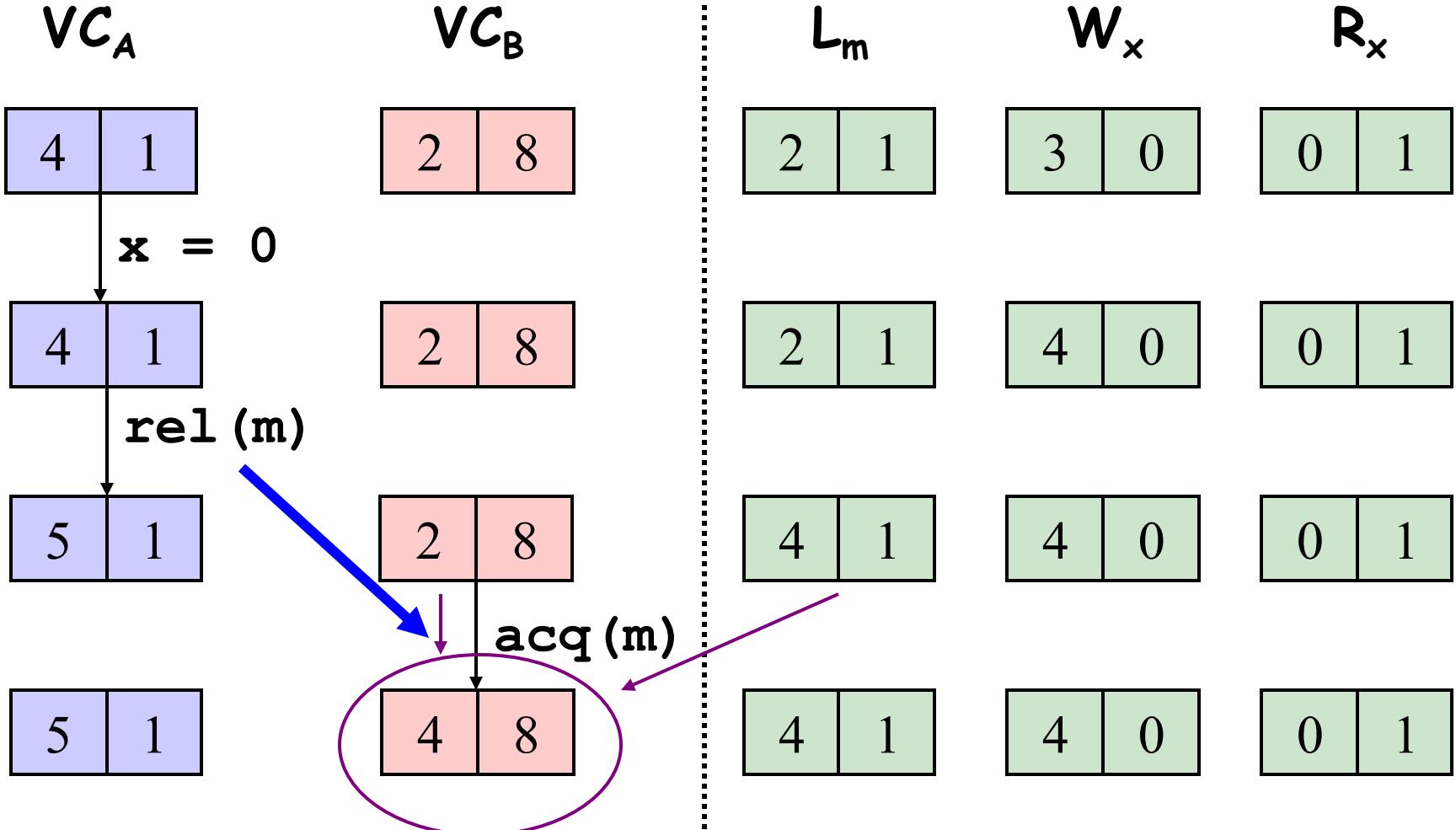
2	8
---	---

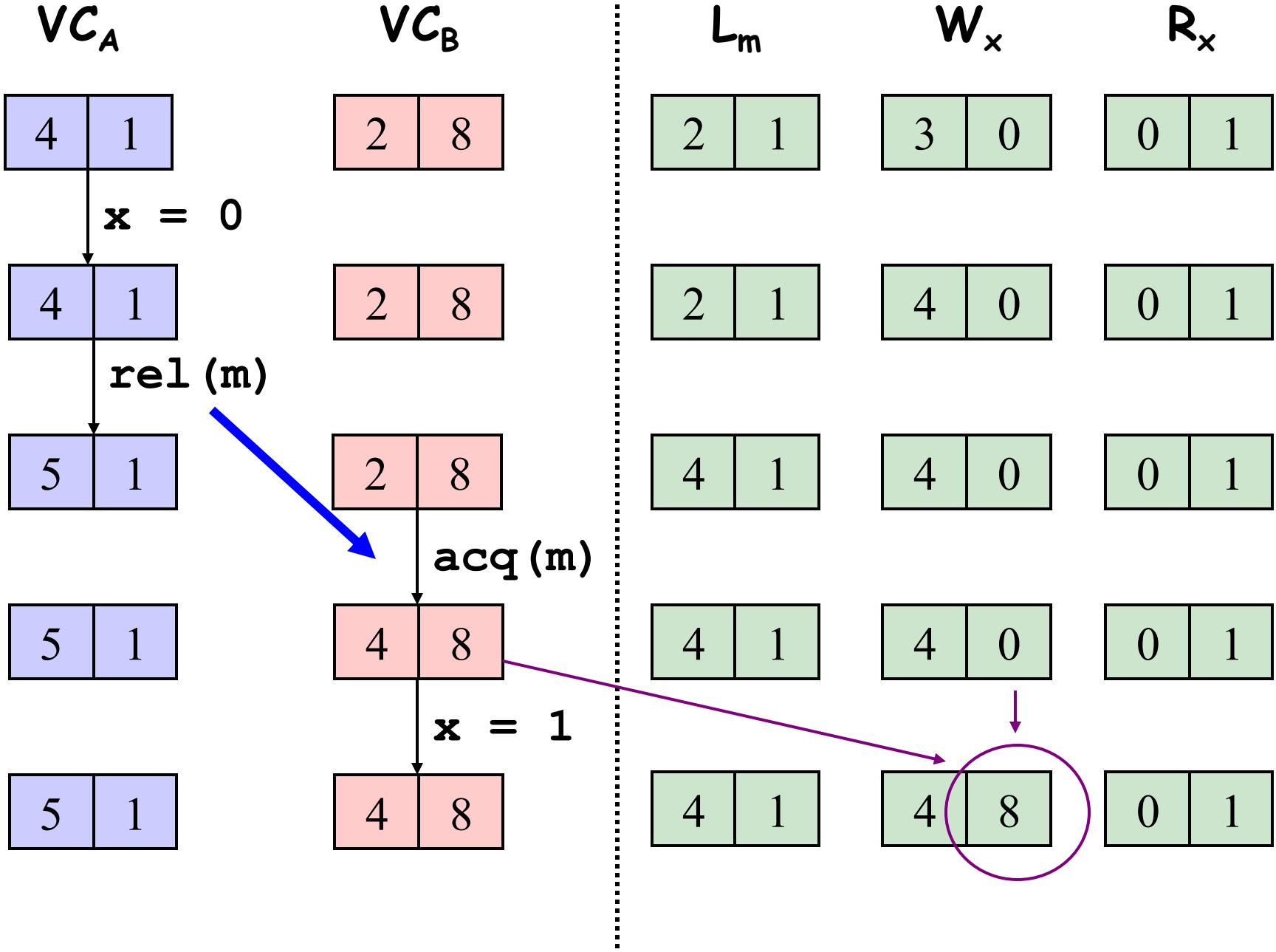
2	8
---	---

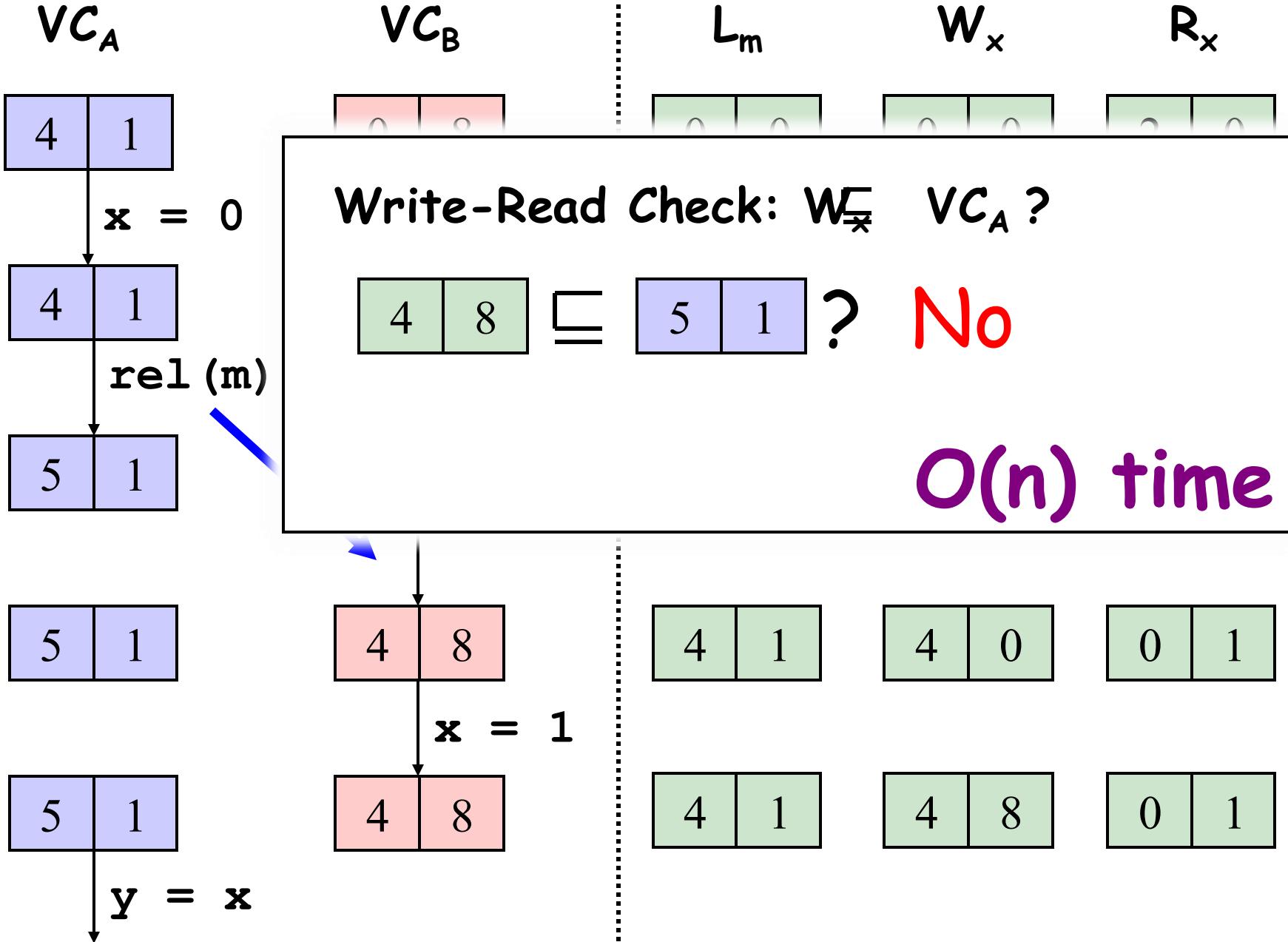
2	1
---	---

4	0
---	---

0	1
---	---







VectorClocks for Data-Race Detection

- Sound
 - Warning → data-race exists
- Complete
 - No warnings → data-race-free execution
- Performance
 - slowdowns > 50x
 - memory overhead

FUZZING TECHNIQUES

Fuzzing can also find data races

- Idea: Catch races “red handed”. Loosely,
 - Pause thread execution when writing to X
 - If another thread reaches a statement that reads or writes X then we have observed concurrent conflicting accesses!
- Analysis does not care about locks or other synchronization primitives.
 - Consistent locking will make the above condition impossible to trigger.

Race Fuzzer

- Run-time Overhead
 - No overhead of tracking synchronization, locks, or vector clocks (hey, that rhymes!)
 - But pausing threads forever can lead to deadlocks
 - Pausing threads for a short while (e.g. `sleep(1000)`) adds overhead for every write access, though this approach is very effective.
- Solution idea:
 - Instead of “pausing” thread, just deprioritize it in the OS scheduler

Race Fuzzing

- Randomized scheduling still depends on luck
- Can do systematic schedule exploration with a bounded number of context switches
- Sophisticated randomized algorithms like PCT can give probabilistic guarantees of uncovering concurrency bugs with a bounded number of “ordering constraints”.
- Or use heuristics, e.g. TSVD uses an initial run to infer “likely” happens-before relationships based on wall-clock timestamps to select candidate “racing pairs”.

Lecture Takeaways

- Data race: two accesses, one of which is a write, with no happens-before relation
- Data races are subtle
 - Compiler optimizations, hardware reordering make racy program behavior hard to predict
 - Better to synchronize consistently
- Lockset analysis: intuitive, fast
 - But many false warnings
- Happens-before data race detection
 - Sound; OK speed if carefully implemented
- Stress testing
 - Sound and fast; Can catch data races red handed
 - Needs assumptions to prune the space of possible races

Key References

- Hans-J. Boehm and Sarita V. Adve, "You Don't Know Jack About Shared Variables or Memory Models", CACM 2012.
- Leslie Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System", CACM 1978.
- Martin Abadi, Cormac Flanagan, and Stephen N. Freund, "Types for Safe Locking: Static Race Detection for Java", TOPLAS 2006.
- Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu, "Finding and Reproducing Heisenbugs in Concurrent Programs", OSDI 2008.
- Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. "Extended static checking for Java", PLDI 2002.
- S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson, "Eraser: A dynamic data race detector for multi-threaded programs", TOCS 1997.

Key References

- Friedemann Mattern, "Virtual Time and Global States of Distributed Systems", Workshop on Parallel and Distributed Algorithms 1989.
- Yuan Yu, Tom Rodeheffer, and Wei Chen, "RaceTrack: Efficient detection of data race conditions via adaptive tracking", SOSP 2005.
- Eli Pozniansky and Assaf Schuster, "MultiRace: Efficient on-the-fly data race detection in multithreaded C++ programs", Concurrency and Computation: Practice and Experience 2007.
- Robert O'Callahan and Jong-Deok Choi, "Hybrid Dynamic Data Race Detection", PPOPP 2003.
- Cormac Flanagan and Stephen N. Freund, "FastTrack: efficient and precise dynamic race detection", CACM 2010.
- Cormac Flanagan and Stephen N. Freund, "The RoadRunner dynamic analysis framework for concurrent programs", PASTE 2010.

Key References

- John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, Kirk Olynyk, "Effective Data-Race Detection for the Kernel", OSDI 2010.
- Madanlal Musuvathi, Sebastian Burckhardt, Pravesh Kothari, and Santosh Nagarakatte, "A Randomized Scheduler with Probabilistic Guarantees of Finding Bugs", ASPLOS 2010.
- Michael D. Bond, Katherine E. Coons, Kathryn S. McKinley, "PACER: proportional detection of data races", PLDI 2010.
- Cormac Flanagan and Stephen N. Freund, "Adversarial memory for detecting destructive races", PLDI 2010.
- Koushik Sen. "Race directed random testing of concurrent programs". PLDI 2010.
- Guangpu Li, Shan Lu, Madanlal Musuvathi, Suman Nath, and Rohan Padhye. "Efficient scalable thread-safety-violation detection: finding thousands of concurrency bugs during testing", SOSP 2019.

Bonus slides on the Java Memory Model (JMM)

Behaviors Allowed in JMM

```
int data = flag = 0;
```

T1

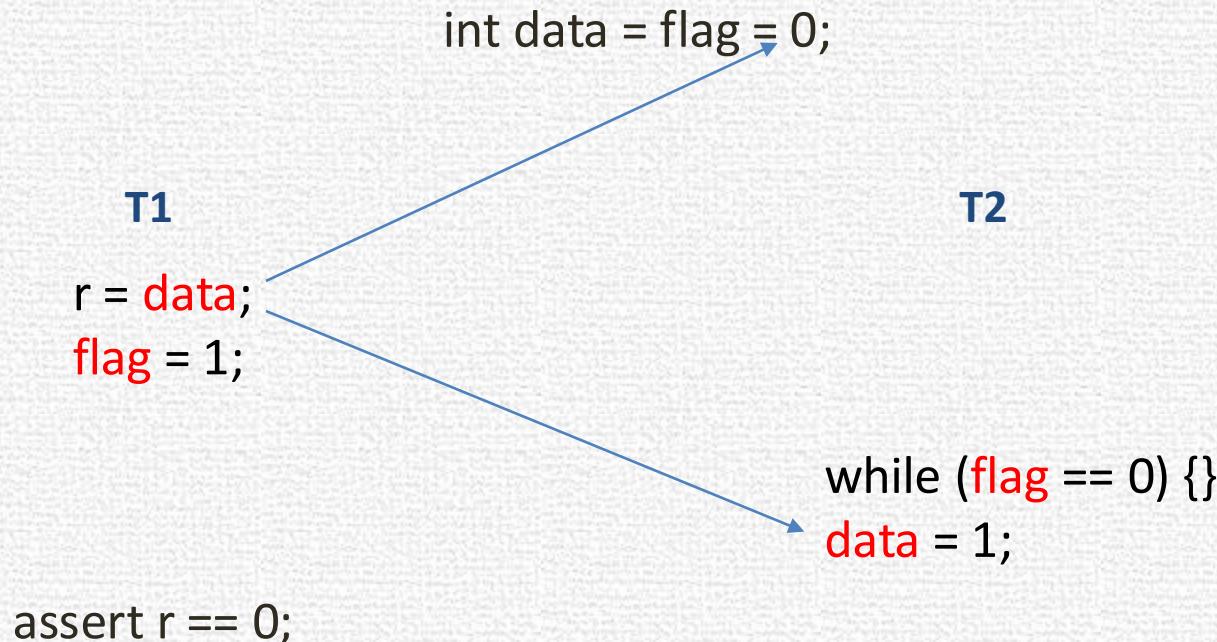
```
r = data;  
flag = 1;
```

```
assert r == 0;
```

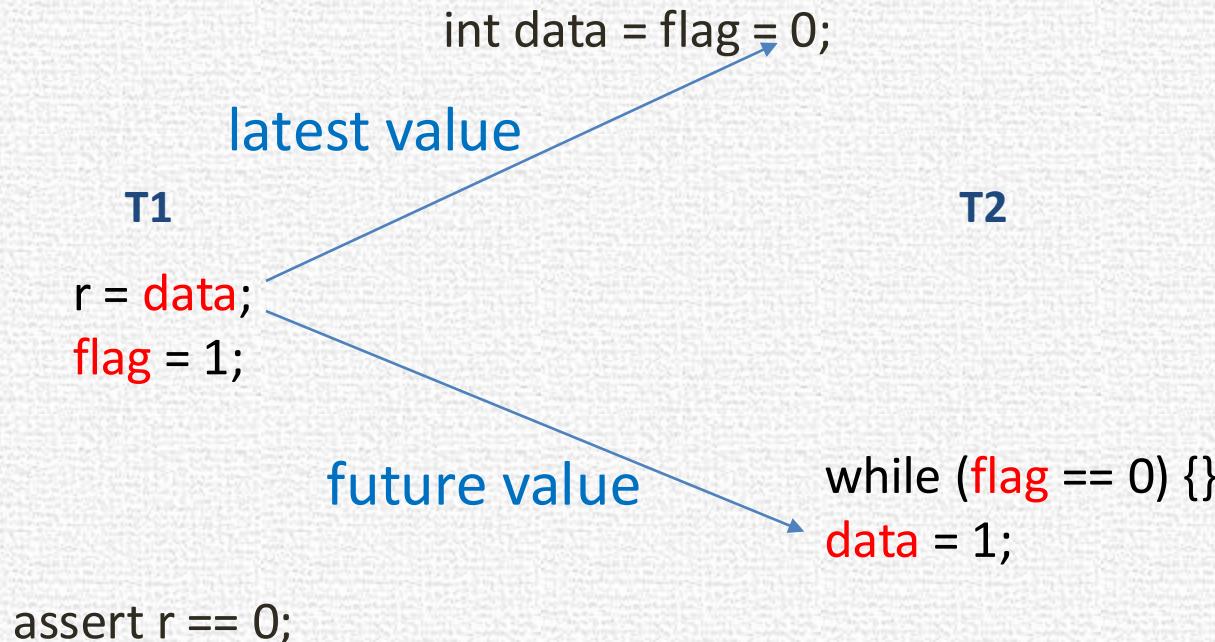
T2

```
while (flag == 0) {}  
data = 1;
```

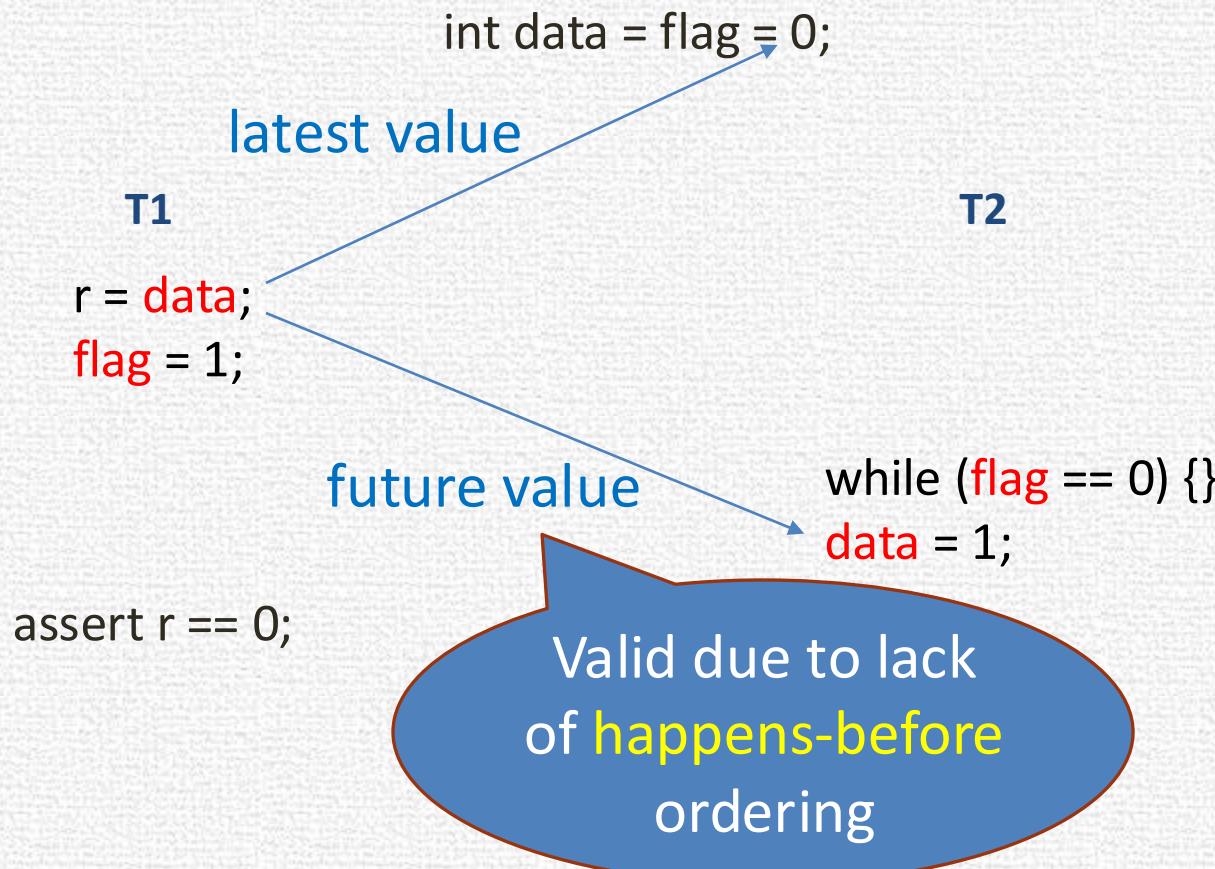
Behaviors Allowed in JMM



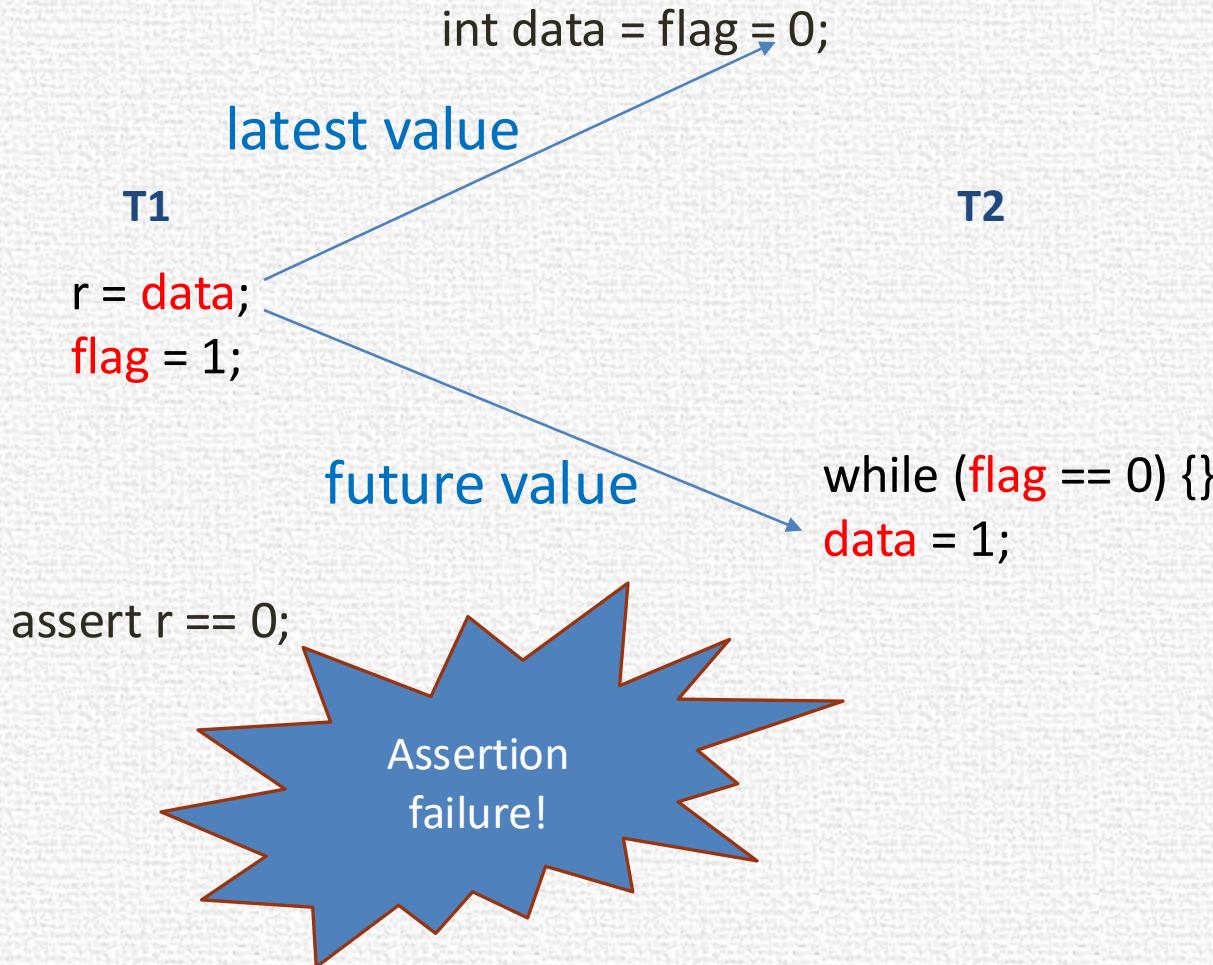
Behaviors Allowed in JMM



Behaviors Allowed in JMM



Behaviors Allowed in JMM



Behaviors Allowed in JMM

```
int data = flag = 0;
```

T1

```
r = data;  
flag = 1;
```

T2

```
while (flag == 0) {}  
data = 1;
```

```
assert r == 0;
```

Assertion
failure!

Behaviors Allowed in JMM

```
int data = flag = 0;
```

T1

```
r = data;  
flag = 1;  
assert r == 0;
```

T2

```
while (flag == 0) {}  
data = 1;
```

Requires returning future value or
reordering to trigger the assertion failure

Can this assert trigger in JVMs? Do you think the JMM allows it?

```
int x = y = 0;
```

T1

```
r1 = x;  
y = r1;
```

T2

```
r2 = y;  
if (r2 == 1) {  
    r3 = y;  
    x = r3;  
} else x = 1;
```

```
assert r2 == 0;
```

The JVM and the JMM

```
int x = y = 0;
```

T1

```
r1 = x;  
y = r1;
```

T2

```
r2 = y;  
if (r2 == 1) {  
    r3 = y;  
    x = r3;  
} else x = 1;
```

JMM disallows
r2 == 1 because
of causality
requirements

```
assert r2 == 0;
```

– Ševčík and Aspinall, ECOOP, 2008

The JVM and the JMM

```
int x = y = 0;
```

T1

```
r1 = x;  
y = r1;
```

However, in a
JVM, after
redundant read
elimination

T2

```
r2 = y;  
if (r2 == 1) {  
    r3 = r2;  
    x = r3;  
} else x = 1;
```

```
assert r2 == 0;
```

The JVM and the JME

```
int x = y = 0;
```

T1

```
r1 = x;  
y = r1;
```

However, in a
JVM, after
redundant read
elimination

T2

```
r2 = y;  
if (r2 == 1) {  
    r3 = r2;  
    x = r3;  
} else x = 1;
```

```
assert r2 == 0;
```

The JVM and the JME

```
int x = y = 0;
```

T1

```
r1 = x;  
y = r1;
```

However, in a
JVM, after
redundant read
elimination

T2

```
r2 = y;  
if (r2 == 1) {  
    r3 = r2;  
    x = r3;  
} else x = 1;  
  
assert r2 == 0;      r2 = y;  
                      If (r2 == 1)  
                      x = r2;  
                      else x = 1;  
  
                      ↓  
                      r2 = y;  
                      x = 1;
```

The JVM and the JME

```
int x = y = 0;
```

T1

```
r1 = x;  
y = r1;
```

However, in a
JVM, after
redundant read
elimination

T2

```
r2 = y;  
if (r2 == 1) {  
    r3 = r2;  
    x = r3;  
} else x = 1;
```

```
r2 = y;  
If (r2 == 1)  
x = r2;  
else x = 1;
```



```
r2 = y;  
x = 1;
```

```
assert r2 == 0;
```

Assertion
failure
possible!

Moral: Just say no to data races

Don't try hacks based on the memory model

- Unless you are as good as Doug Lea



Author of `java.util.concurrent`

- Or you have formalized the memory model rules in a tool
 - And even then, are the rules right?