DYNAMIC ANALYSES FOR DATA RACE DETECTION

Jonathan Aldrich

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

Based in large part on slides by John Erickson, Stephen Freund, Madan Musuvathi, Mike Bond, and Man Cao, used by permission

Overview of Data Race Detection Techniques

- Static data race detection
- Dynamic data race detection
  - Lock-set
  - Happen-before
  - DataCollider

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

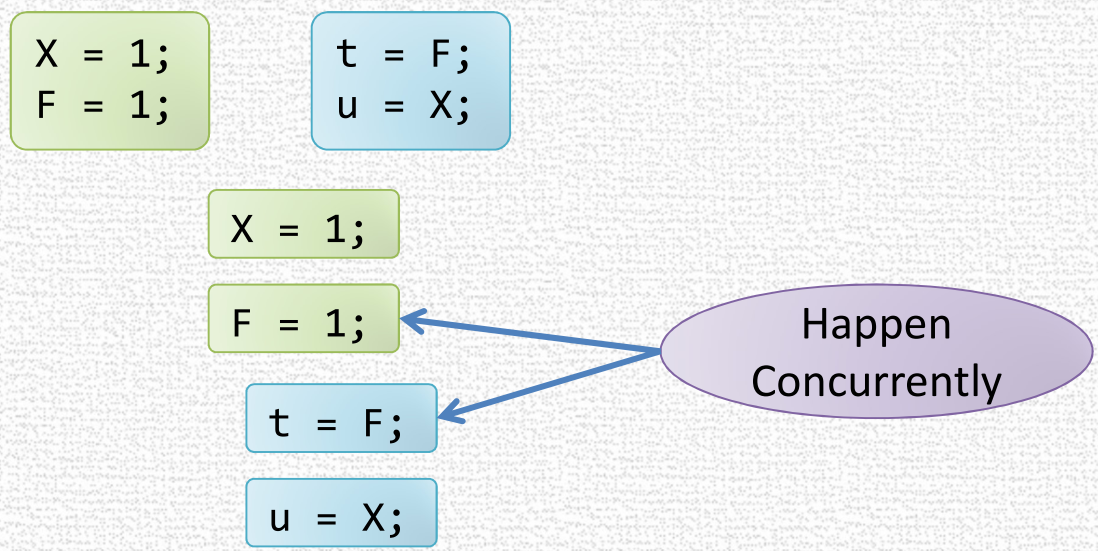
- Disadvantages
  - Run-time overhead (5-20x for best tools)
  - Memory overhead for analysis state
  - Reasons only about observed executions
    - sensitive to test coverage
    - (some generalization possible...)

Tradeoffs: Static vs Dynamic

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

Definition Refresh

- A data race is a pair of concurrent conflicting accesses to unannotated locations



- 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

```
AcquireLock( A );
AcquireLock( B );
x ++;
ReleaseLock( B );
ReleaseLock( A );


AcquireLock( B );
AcquireLock( C );
x ++;
ReleaseLock( C );
ReleaseLock( B );
```

X is protected by A, or B, or both

X is protected by B, or C, or both

X is protected by B

LockSet Algorithm

- Two data structures:
  - LocksHeld(t) = set of locks held currently by thread t
    - Initially set to Empty
  - LockSet(x) = set of locks that could potentially be protecting x
    - Initially set to the universal set

- When thread t acquires lock l
  - $LocksHeld(\ t\ ) = LocksHeld(\ t\ ) \cup \{l\}$
- When thread t releases lock l
  - $LocksHeld(\ t\ ) = LocksHeld(\ t\ )\ - \{l\}$

- When thread t accesses location x
  - $LockSet(\ x\ ) = \ LockSet(\ x\ ) \cap LocksHeld(\ t\ )$
  - Report “data race” when LockSet(x) becomes empty

Algorithm Guarantees

- 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 );
pkt = SendQueue.Top();
ReleaseLock( SendQueueLk );
```

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

LockSet Algorithm Guarantees

- No warnings → no data races on the current execution
  - The program followed consistent locking discipline in this execution

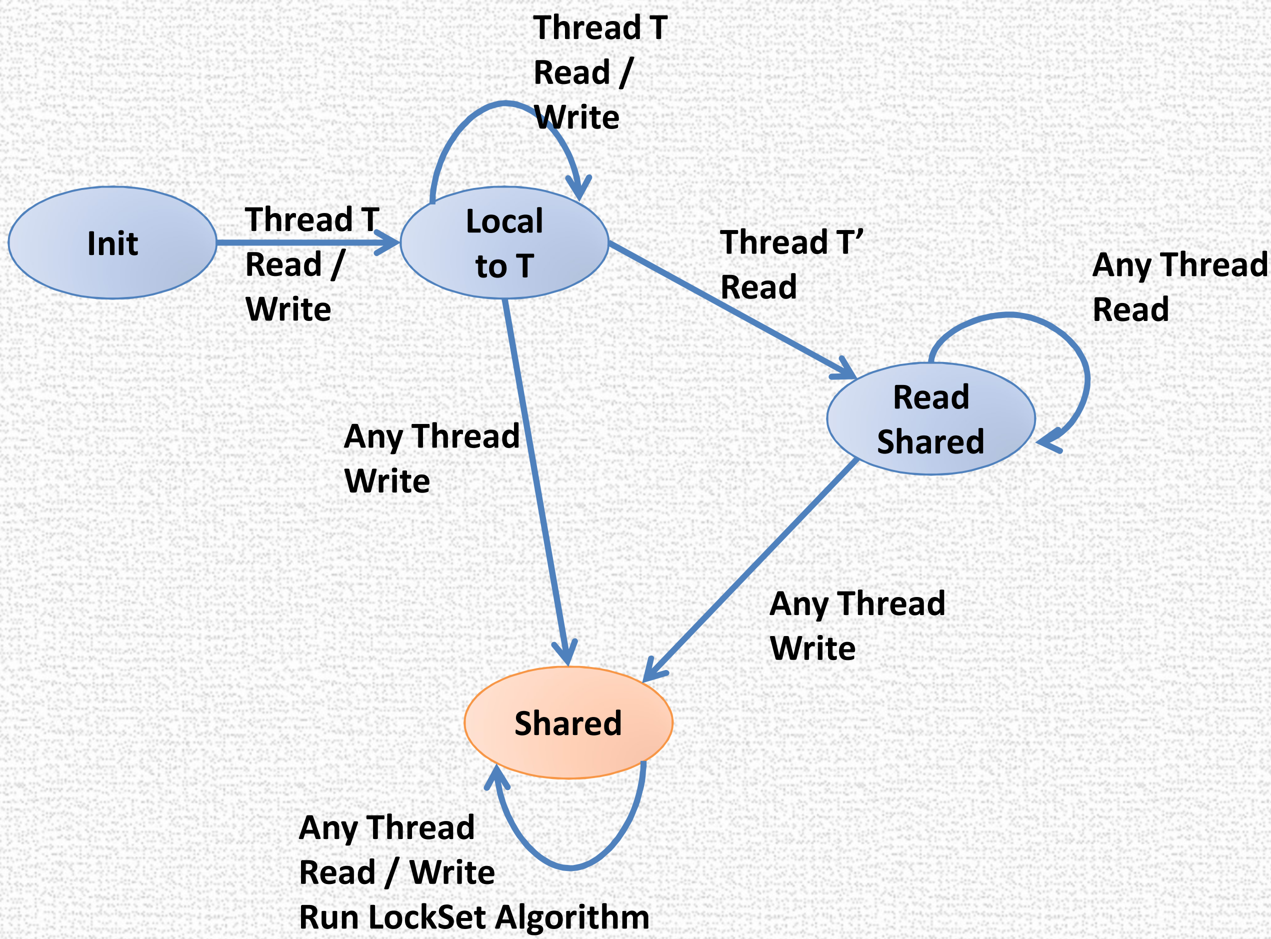
- Warnings does not imply a data race
  - Object read-shared after thread-local initialization

```
A = new A();
A.f = 0;

// publish A
globalA = A;
```

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

Maintain A State Machine Per Location



LockSet Algorithm Guarantees

- 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 Guarantees

- 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

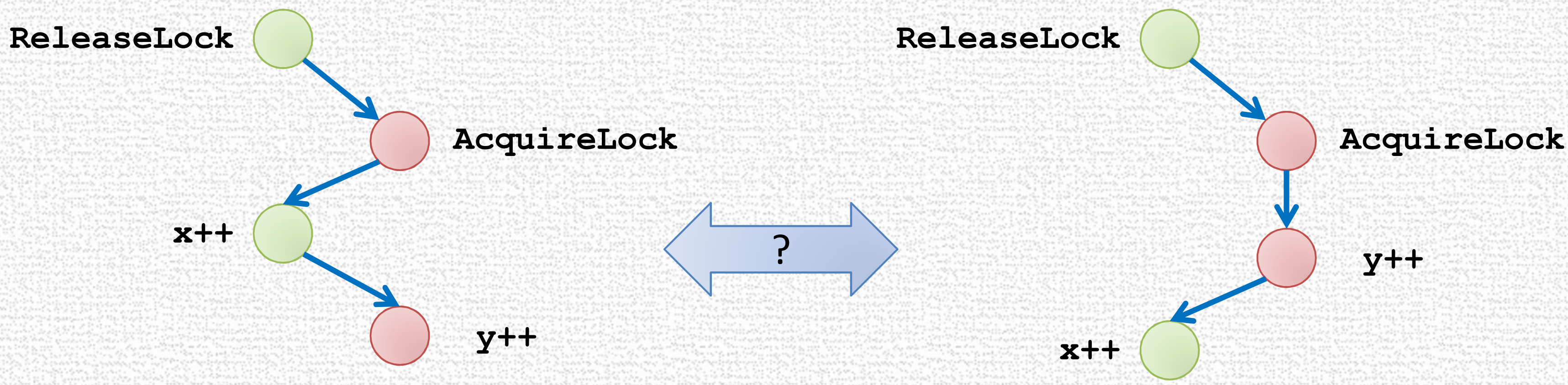
Pkt is thread local

Pkt is protected by ProcQueueLk

HAPPENS-BEFORE

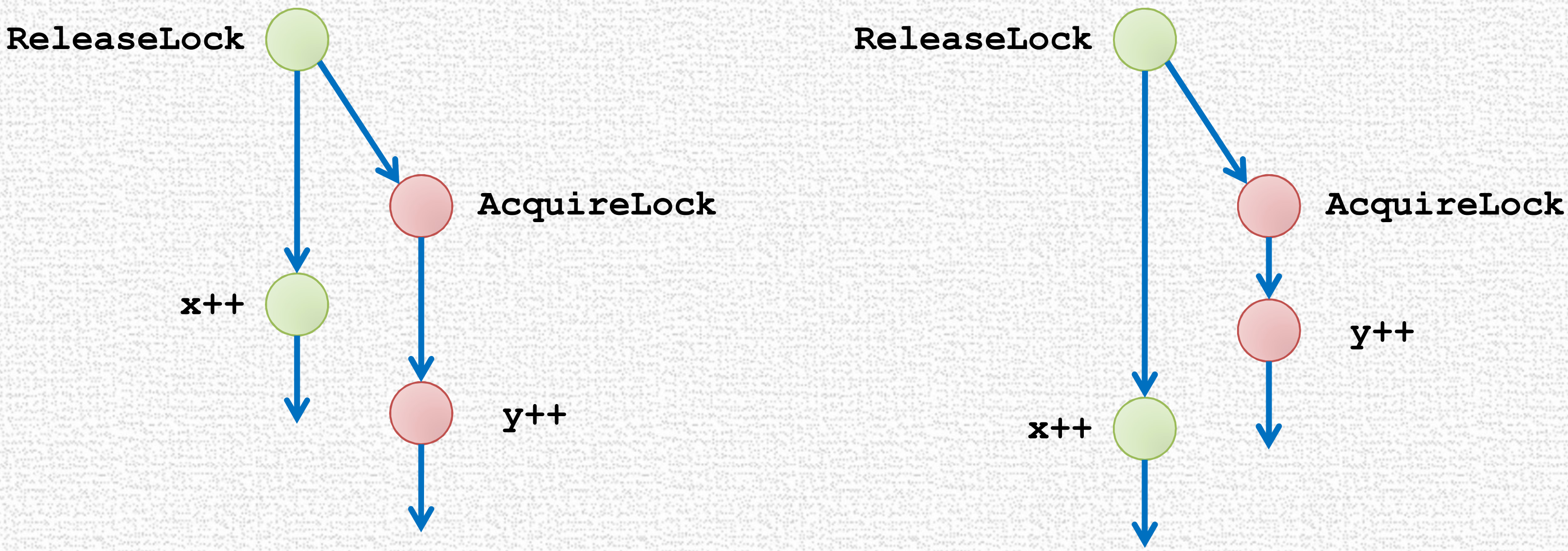
Happens-Before Relation [Lamport '78]

- 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]

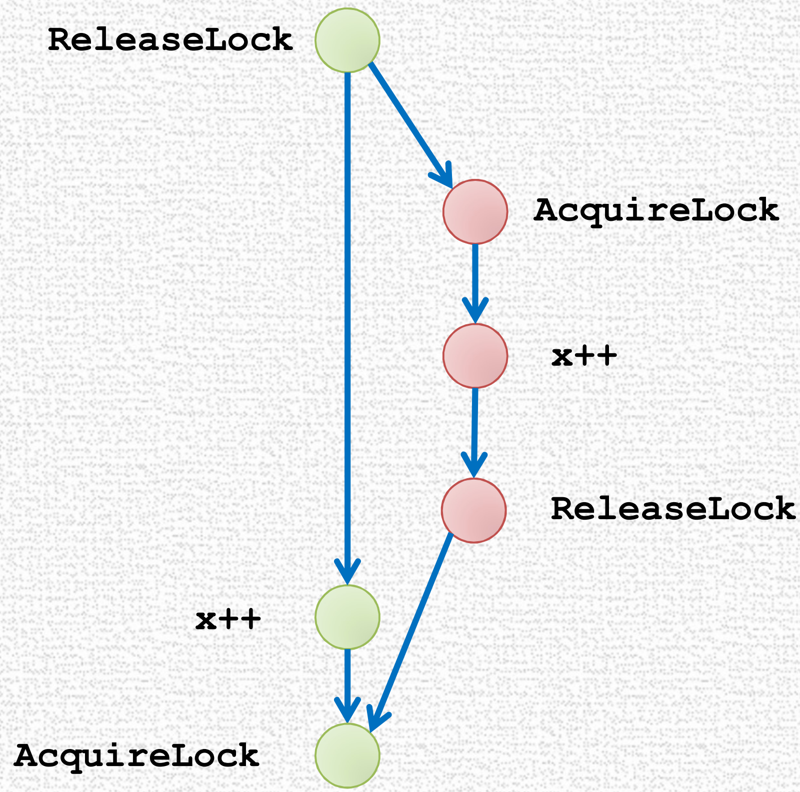
- A concurrent execution is a partial-order determined by communication events
- The program cannot "observe" the order of concurrent non-communicating events



- Both executions form the same happens-before relation

Constructing the Happens-Before Relation

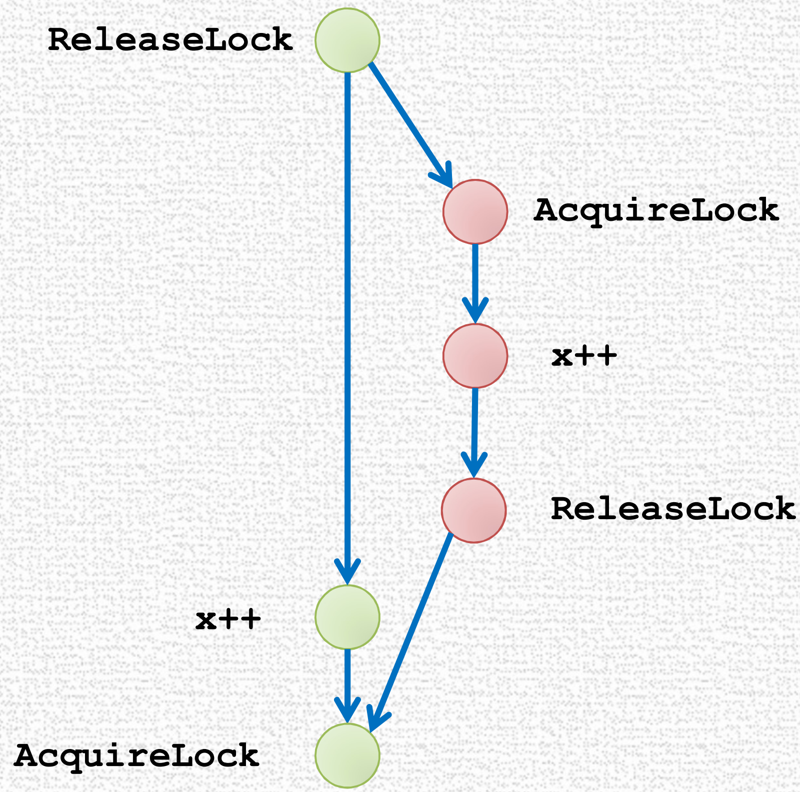
- Program order
  - Total order of thread instructions
- Synchronization order
  - Total order of accesses to the same synchronization



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



Happens-Before and Data-Races

- Not all unordered conflicting accesses are data races

Init: X = Y = 0;

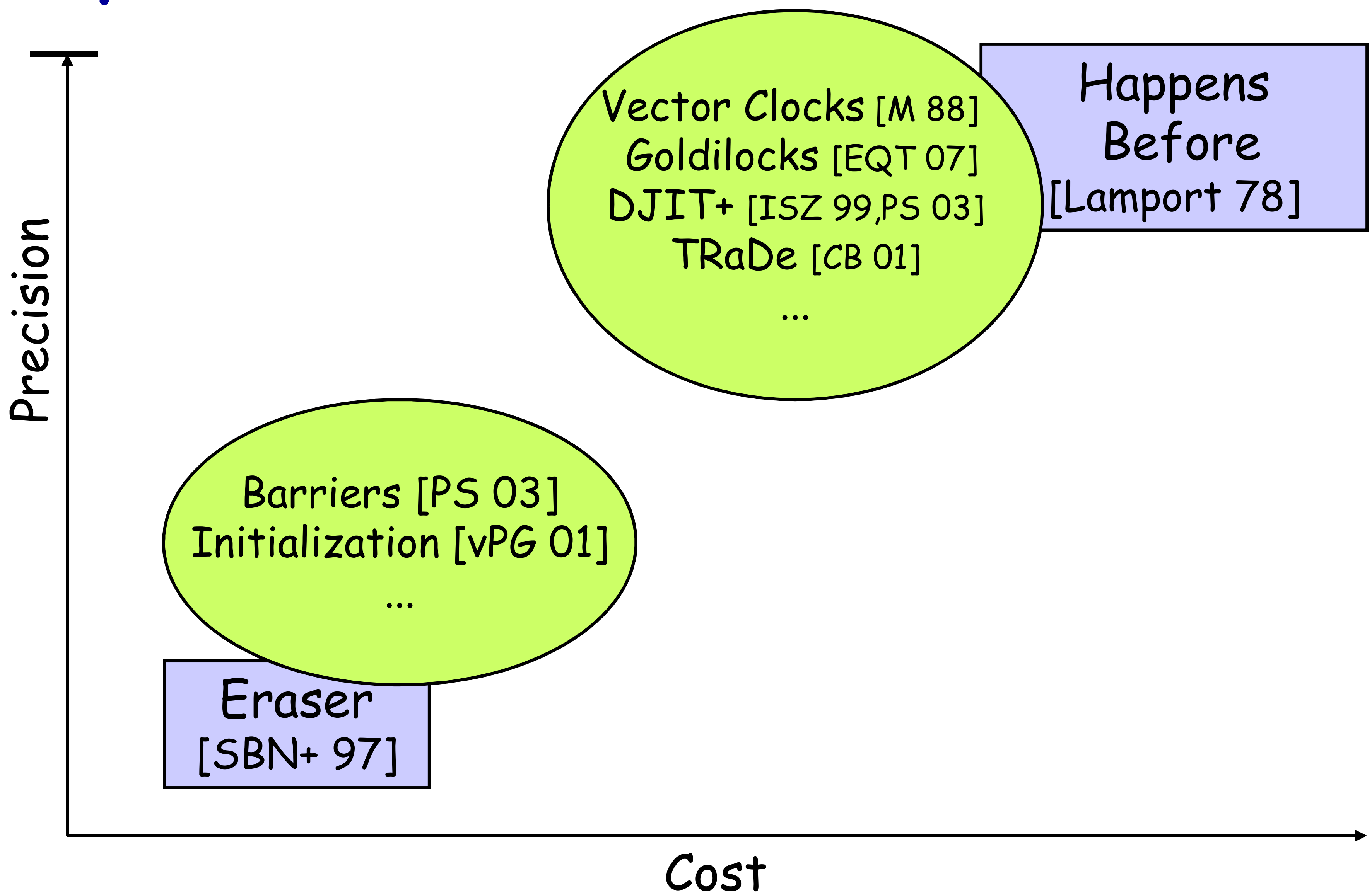
```
X = 1;
Y = 1;
```

```
if( Y == 1 )
  X = 2;
```

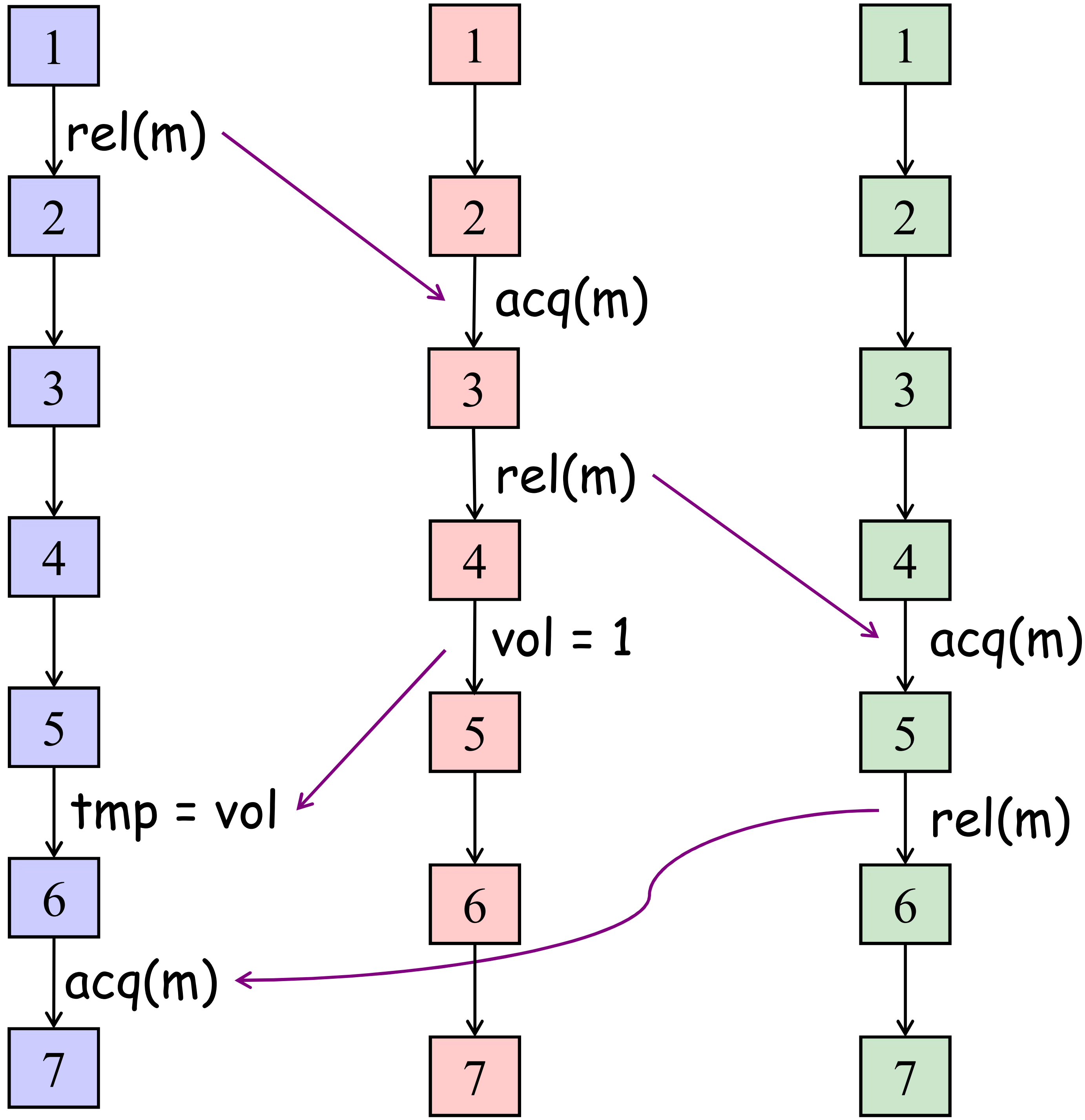
- There is no data race on X
- But, there is a data race on Y
- Remember:
  - Exists unordered conflicting access → Exists data race

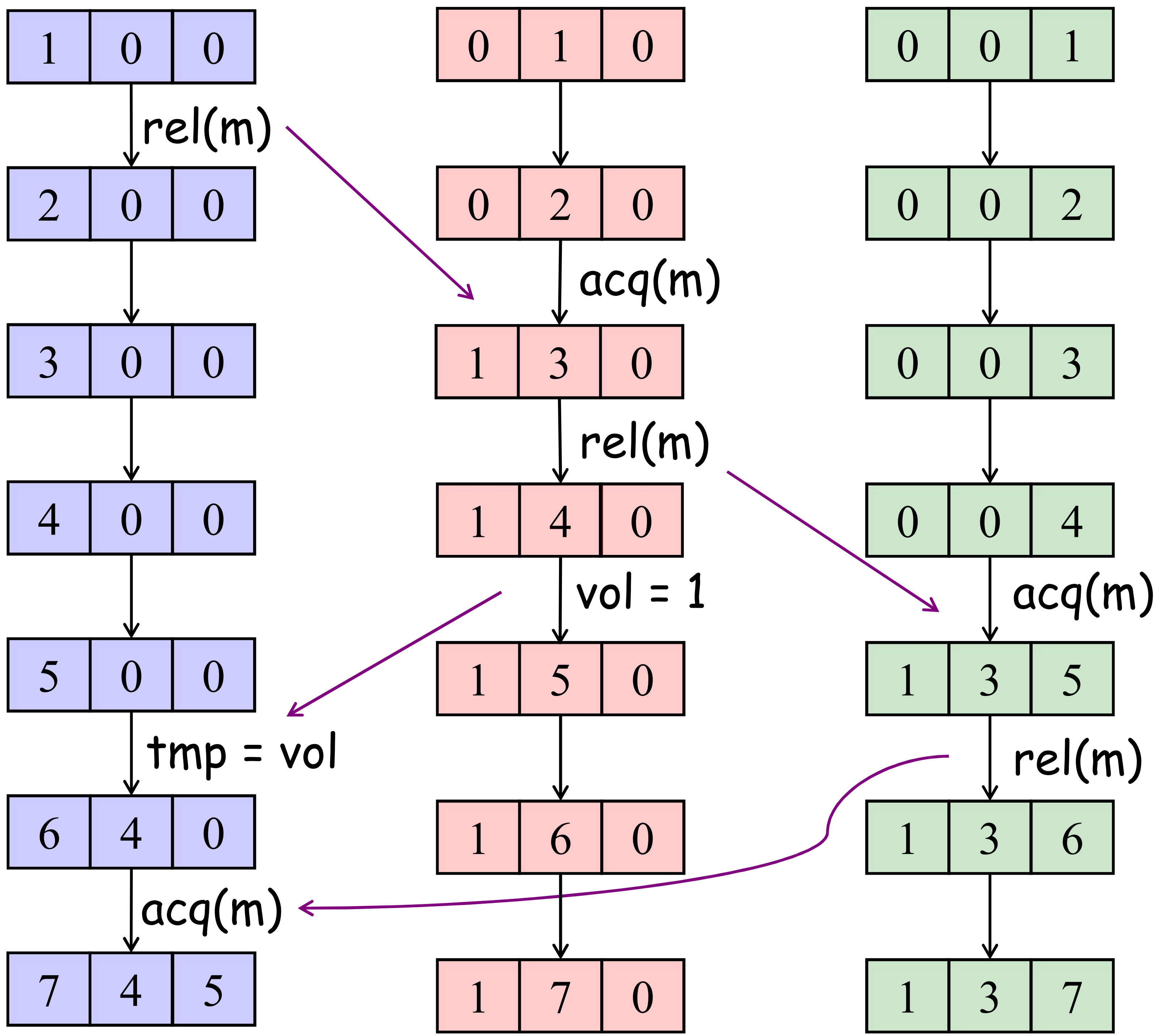
IMPLEMENTING HAPPENS-BEFORE ANALYSES

Dynamic Data-Race Detection

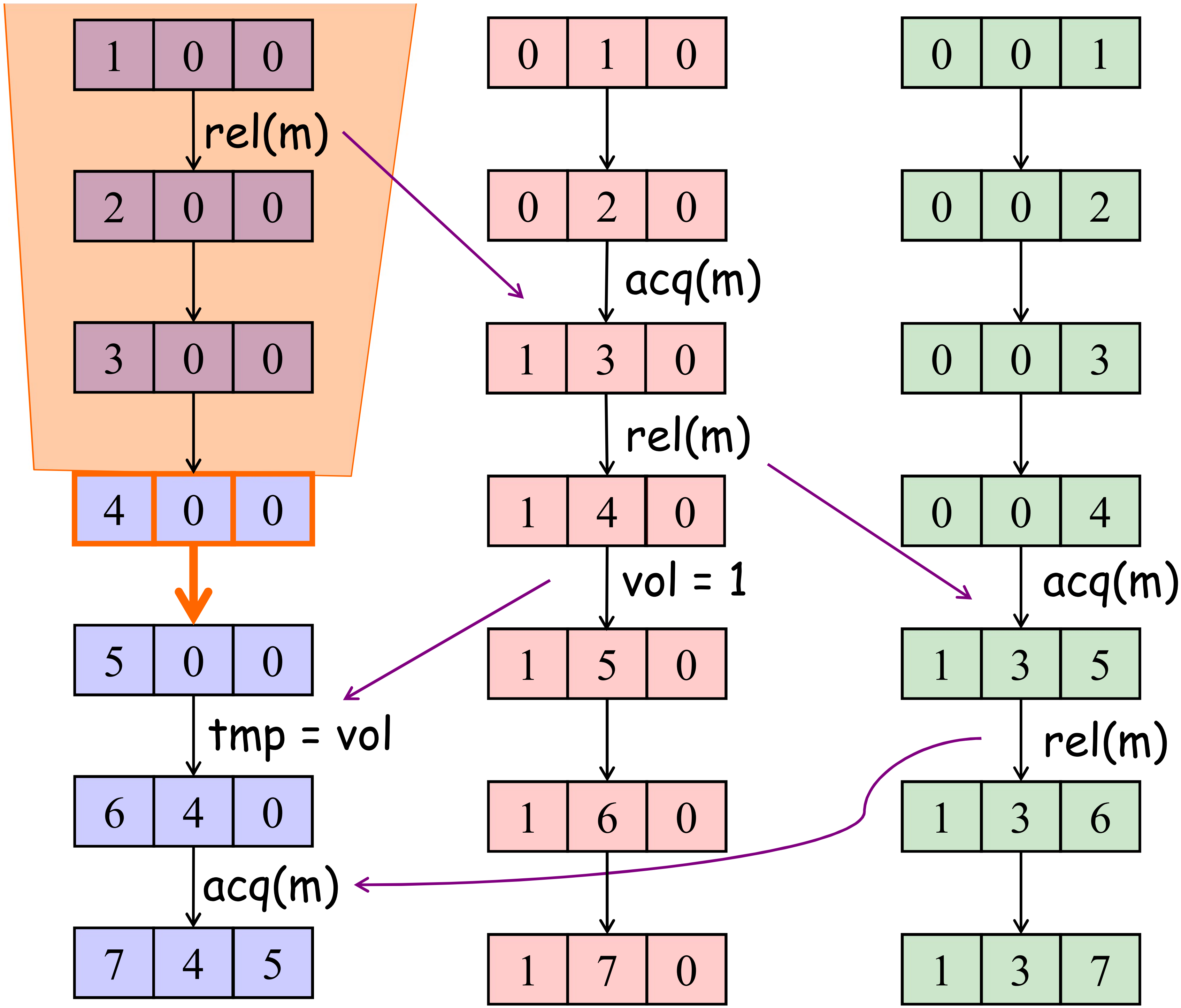


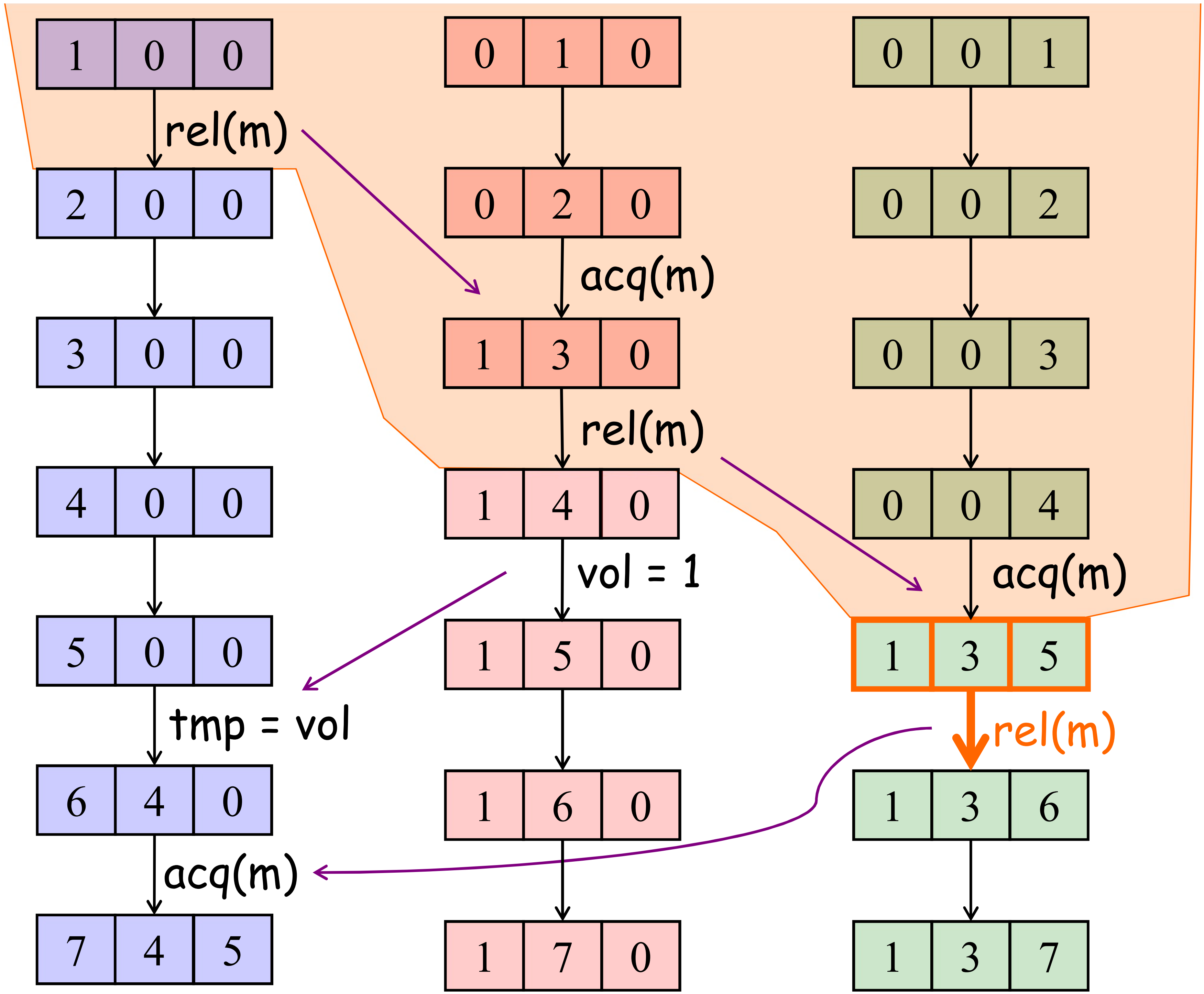
Precise Happens-Before



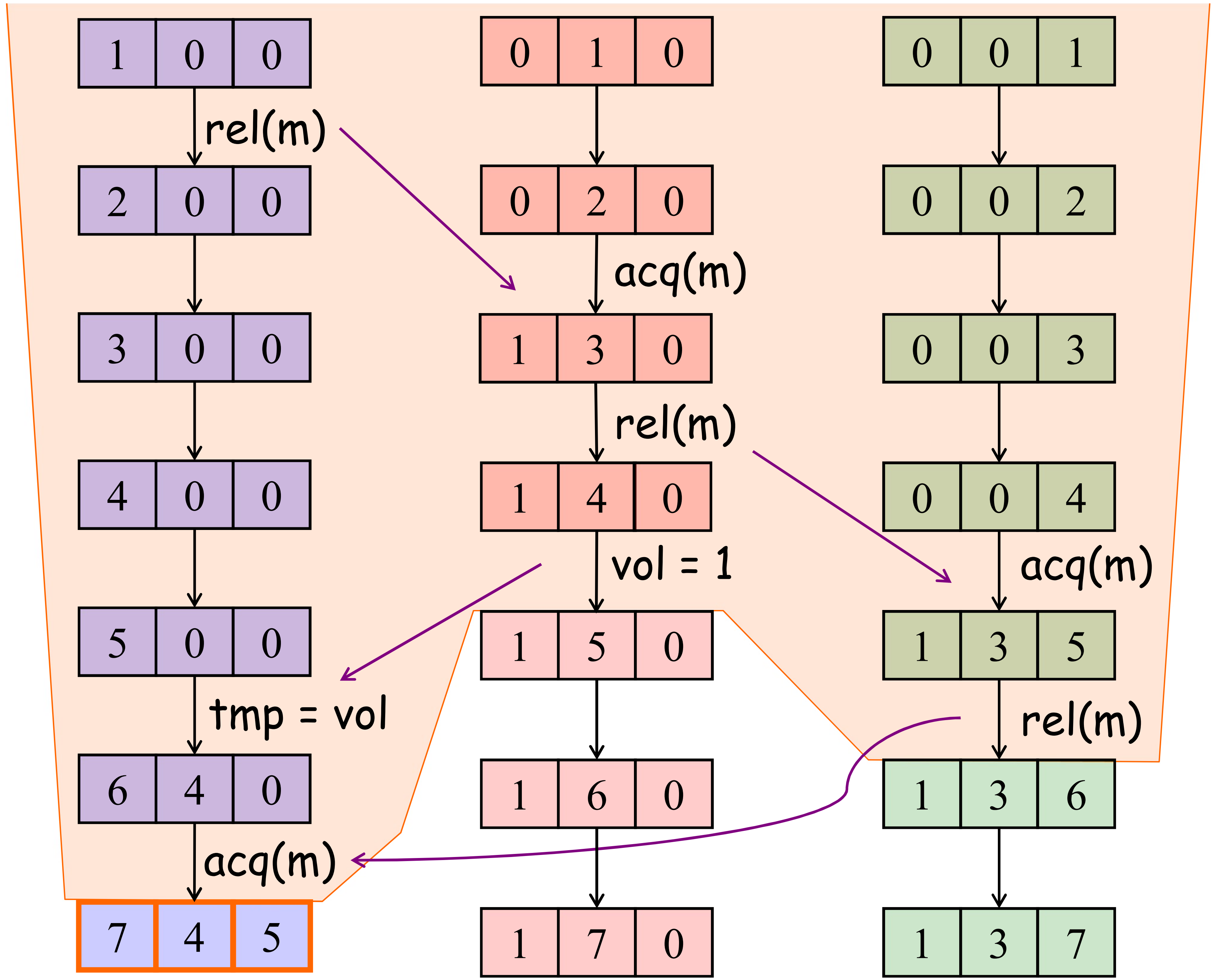


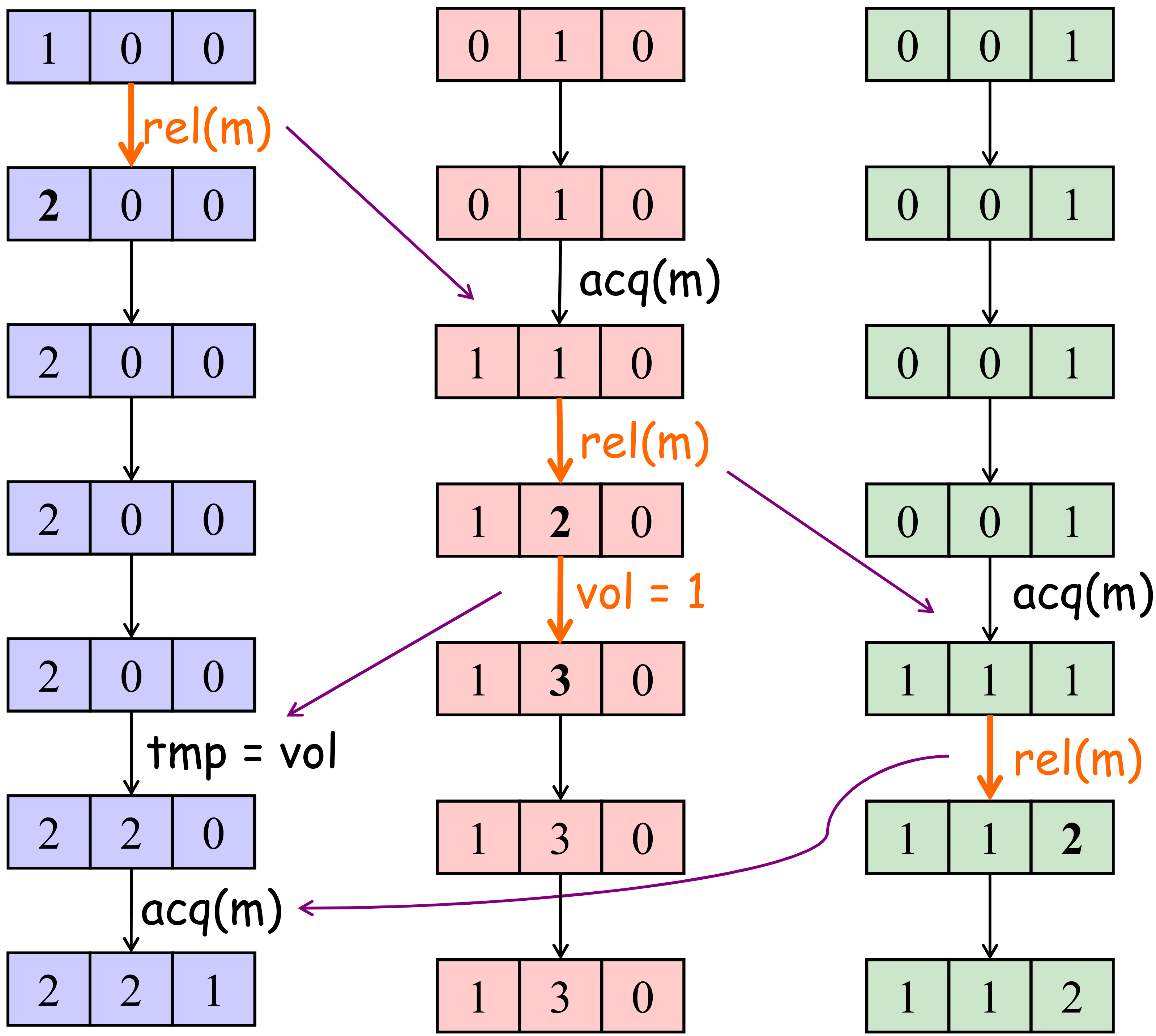
1 0 0
rel(m)
2 0 0
3 0 0
4 0 0
5 0 0
tmp = vol
6 4 0
acq(m)
7 4 5
0 1 0
0 2 0
acq(m)
1 3 0
rel(m)
1 4 0
vol = 1
1 5 0
1 6 0
1 7 0
0 0 1
0 0 2
0 0 3
0 0 4
acq(m)
1 3 5
rel(m)
1 3 6
1 3 7


rel(m)
acq(m)
rel(m)
vol = 1
acq(m)
tmp = vol
rel(m)
acq(m)

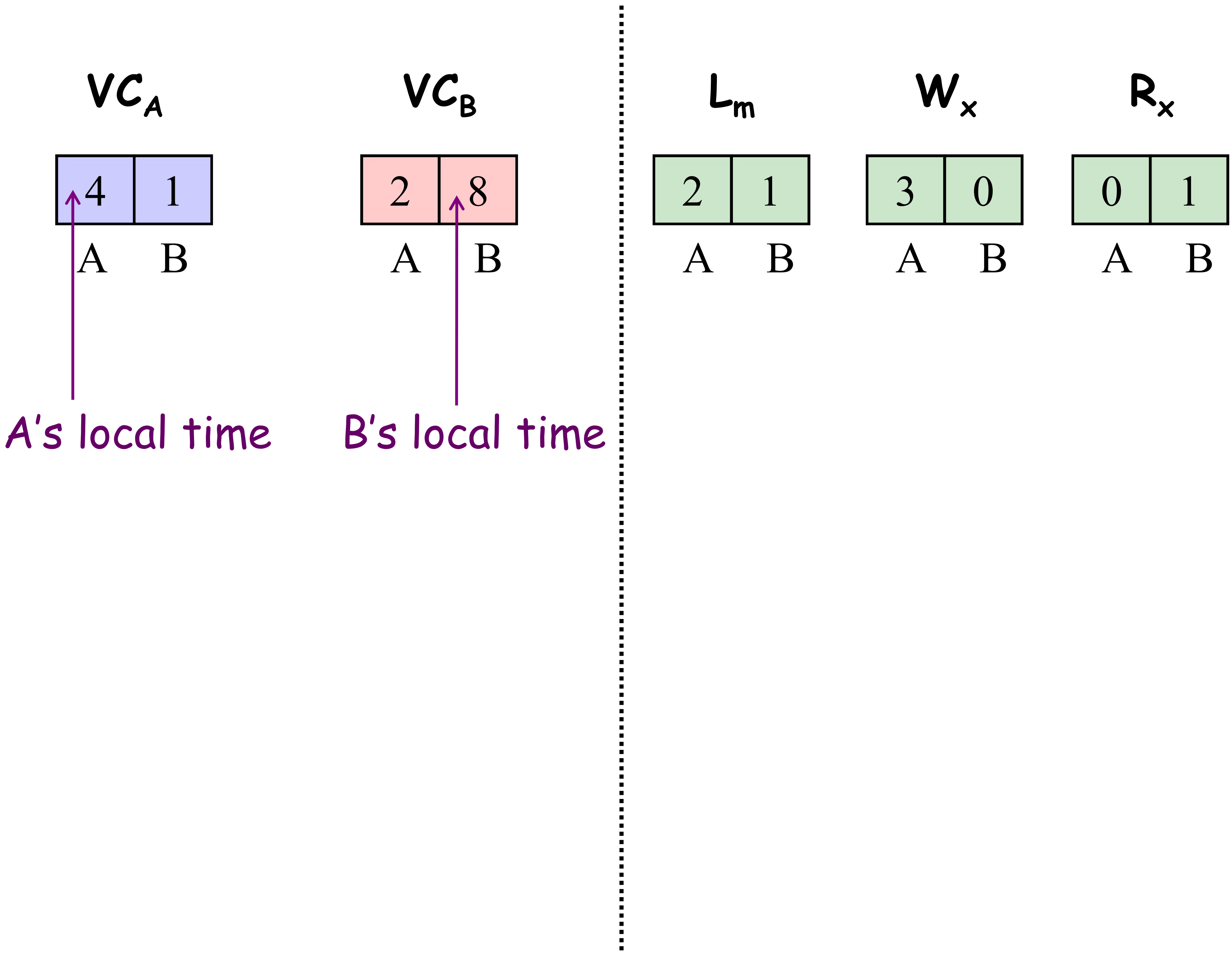


1 0 0
rel(m)
2 0 0
3 0 0
4 0 0
5 0 0
tmp = vol
6 4 0
acq(m)
7 4 5
0 1 0
0 2 0
acq(m)
1 3 0
rel(m)
1 4 0
vol = 1
1 5 0
1 6 0
1 7 0
0 0 1
0 0 2
0 0 3
0 0 4
acq(m)
1 3 5
rel(m)
1 3 6
1 3 7

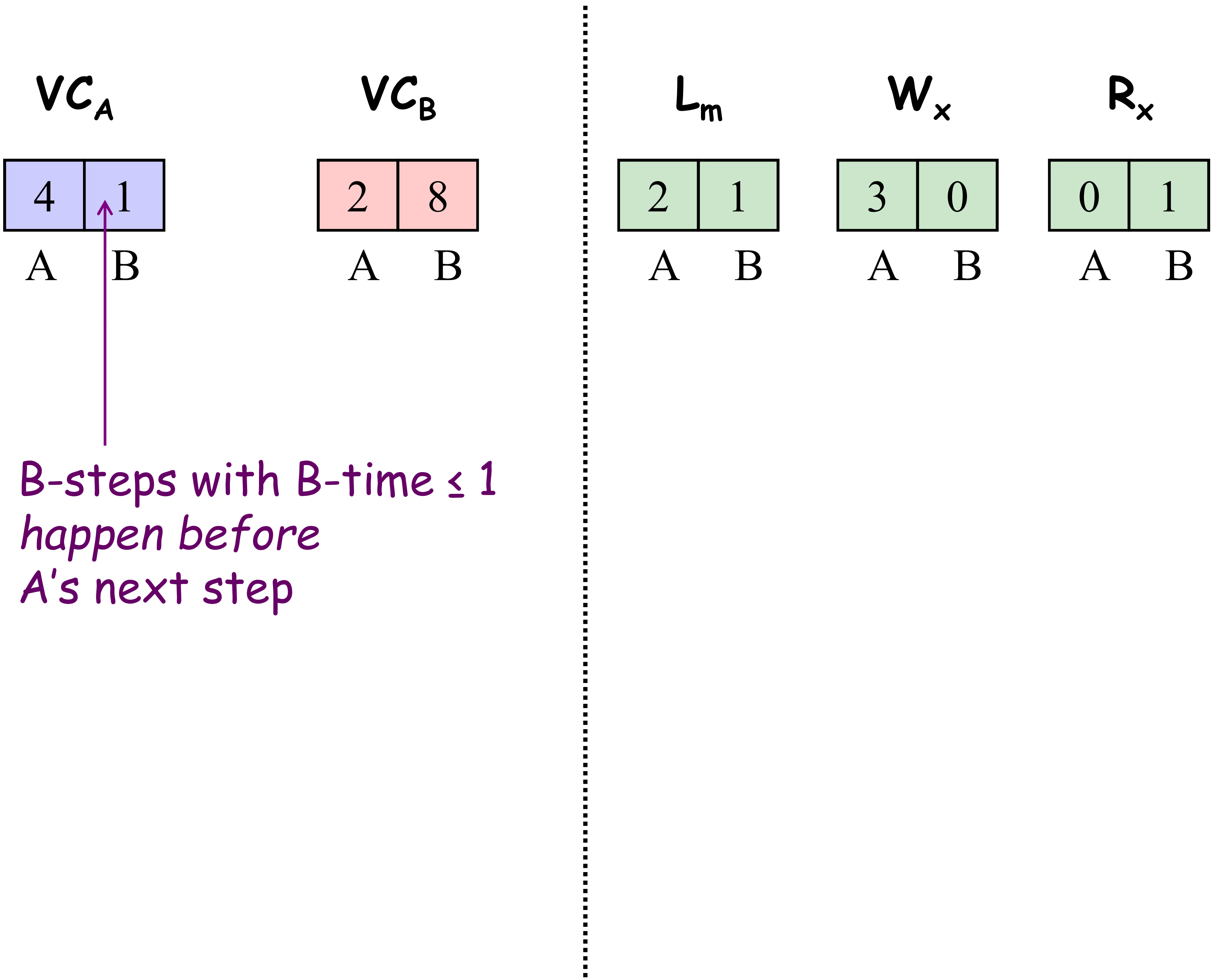

1 0 0
0 1 0
0 0 1
rel(m)
2 0 0
0 2 0
0 0 2
acq(m)
3 0 0
1 3 0
0 0 3
rel(m)
4 0 0
1 4 0
0 0 4
vol = 1
acq(m)
5 0 0
1 5 0
1 3 5
tmp = vol
rel(m)
6 4 0
1 6 0
1 3 6
acq(m)
7 4 5
1 7 0
1 3 7

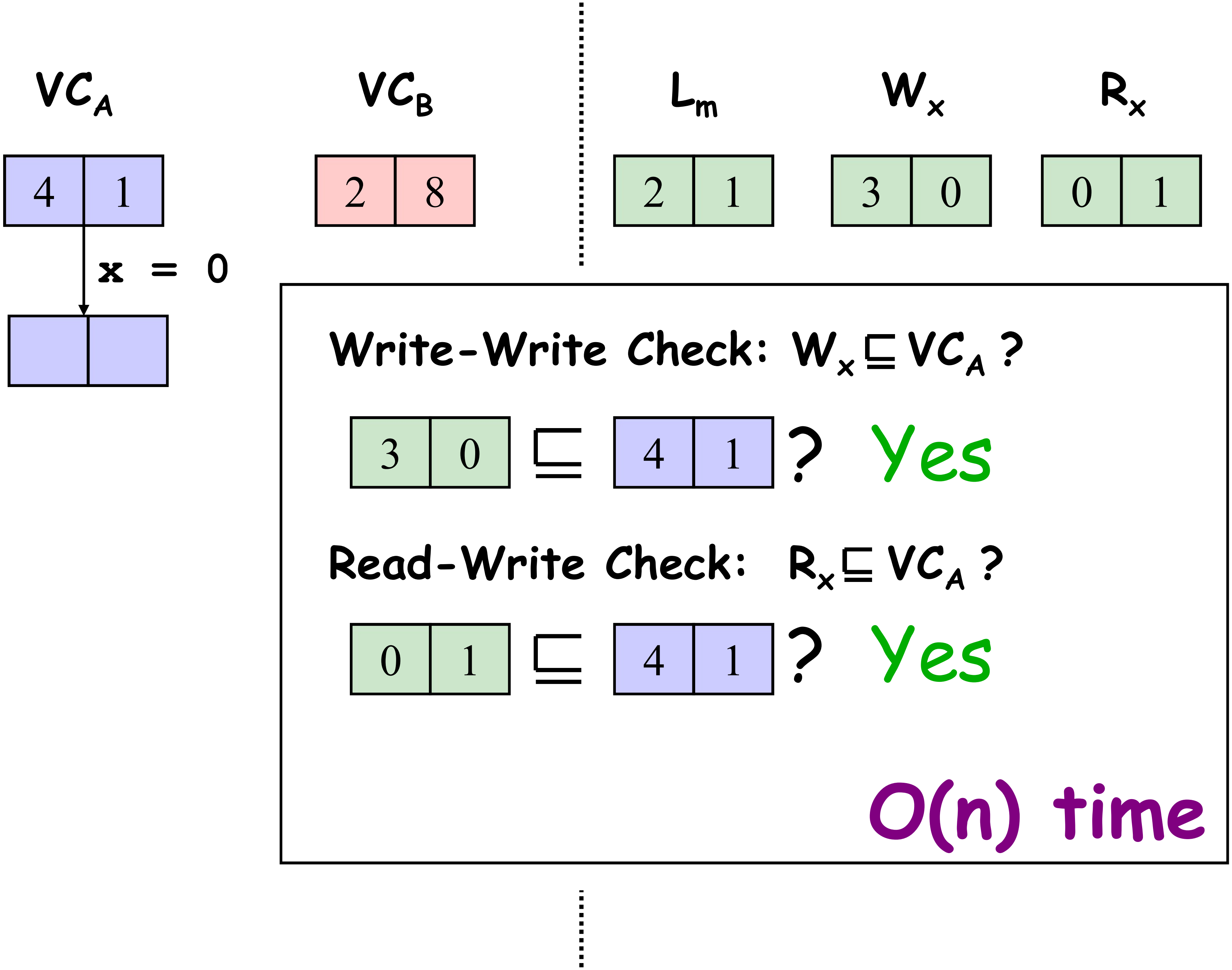


1 0 0
rel(m)
2 0 0
2 0 0
2 0 0
2 0 0
tmp = vol
2 2 0
acq(m)
2 2 1
0 1 0
0 1 0
acq(m)
1 1 0
rel(m)
1 2 0
vol = 1
1 3 0
1 3 0
1 3 0
0 0 1
0 0 1
0 0 1
0 0 1
acq(m)
1 1 1
rel(m)
1 1 2
1 1 2

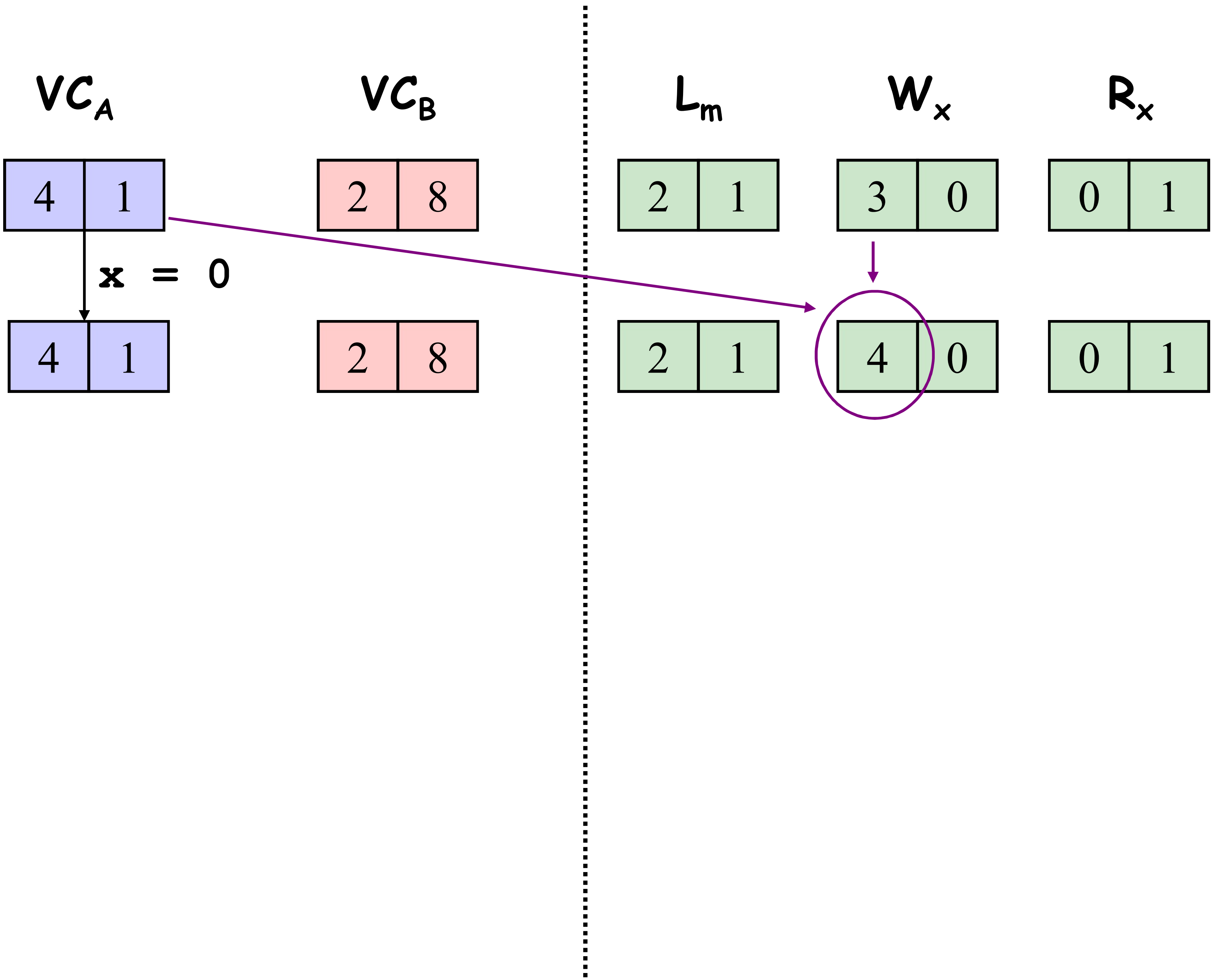


VC_A
4 1
A B
VC_B
2 8
A B
L_m
2 1
A B
W_x
3 0
A B
R_x
0 1
A B
A's local time
B's local time

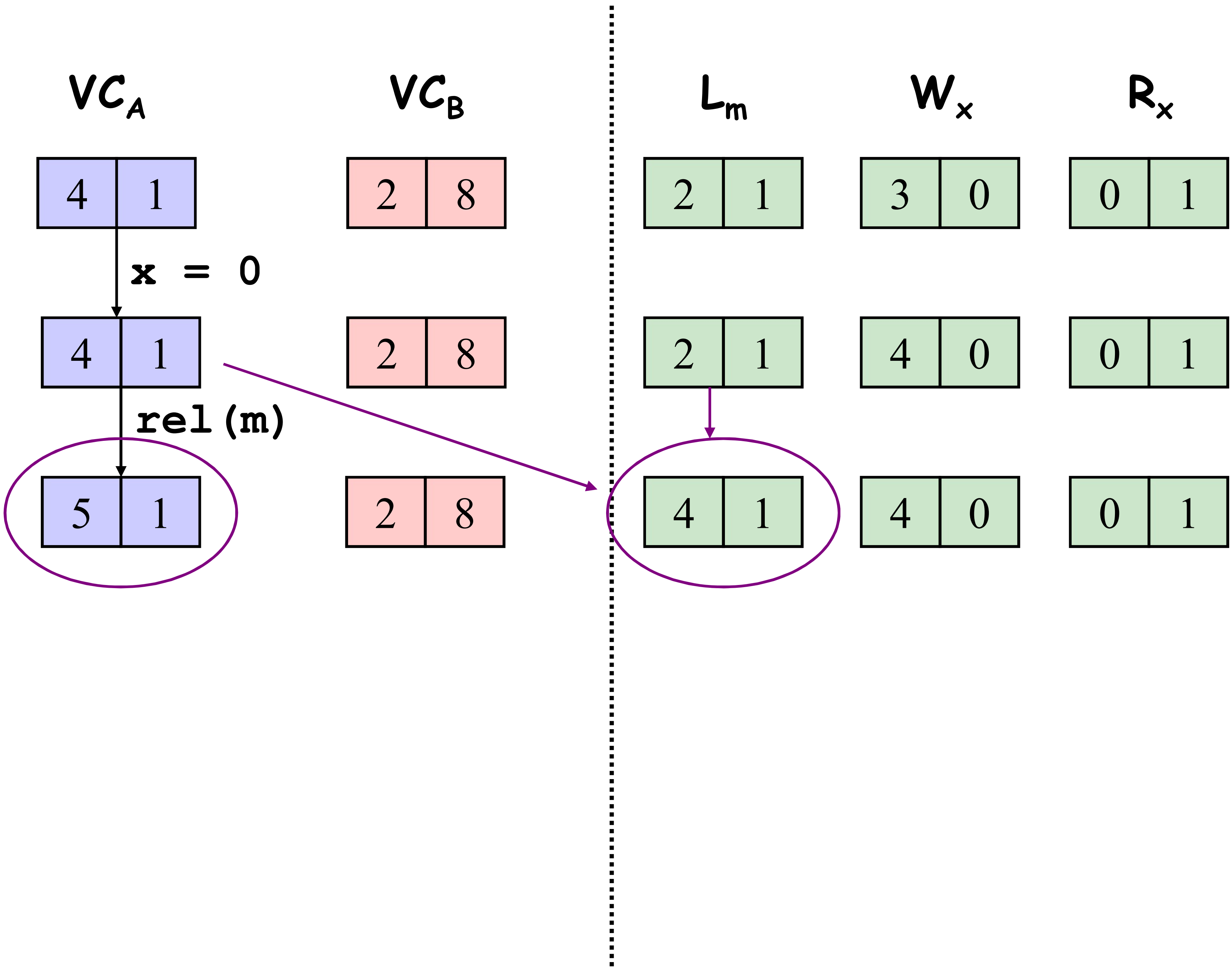

VC_A
4 1
A B
VC_B
2 8
A B
L_m
2 1
A B
W_x
3 0
A B
R_x
0 1
A B
B-steps with B-time ≤ 1
happen before
A's next step



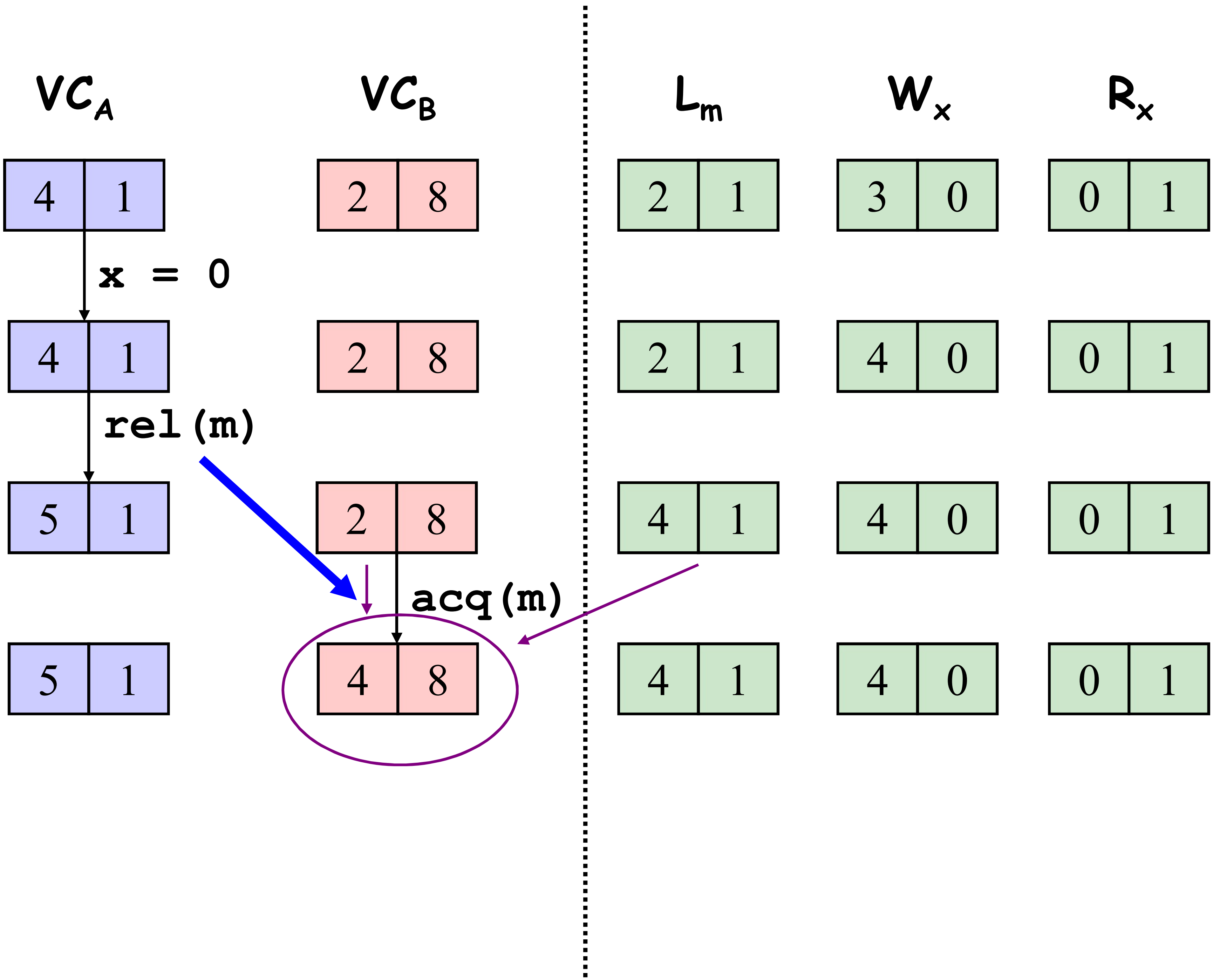
VCA
4 1
x = 0
VCB
2 8
Lm
2 1
Wx
3 0
Rx
0 1
Write-Write Check: Wx ⊑ VCA ?
3 0 ⊑ 4 1 ? Yes
Read-Write Check: Rx ⊑ VCA ?
0 1 ⊑ 4 1 ? Yes
O(n) time



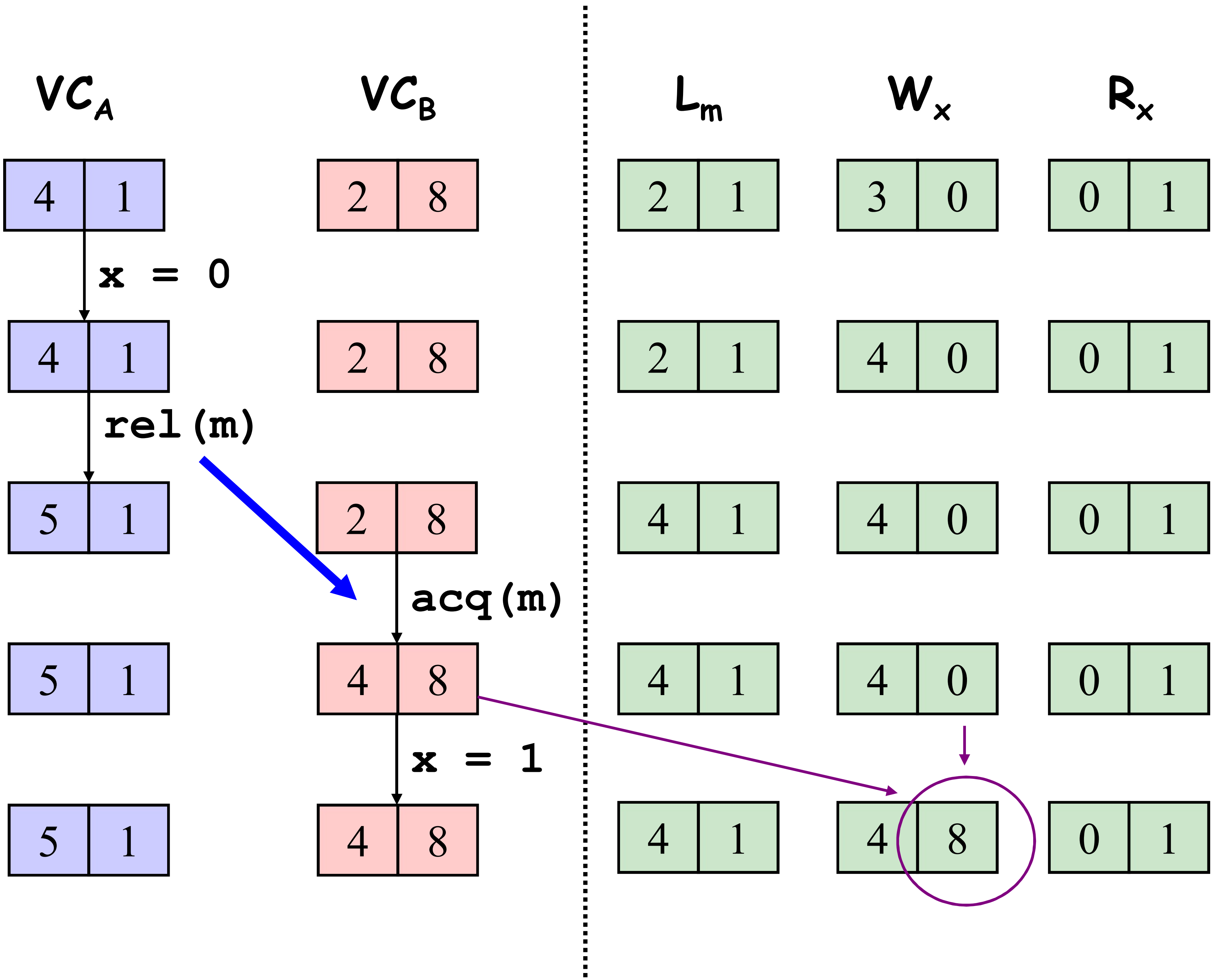
VC_A
VC_B
L_m
W_x
R_x
4 1
2 8
2 1
3 0
0 1
x = 0
4 1
2 8
2 1
4 0
0 1



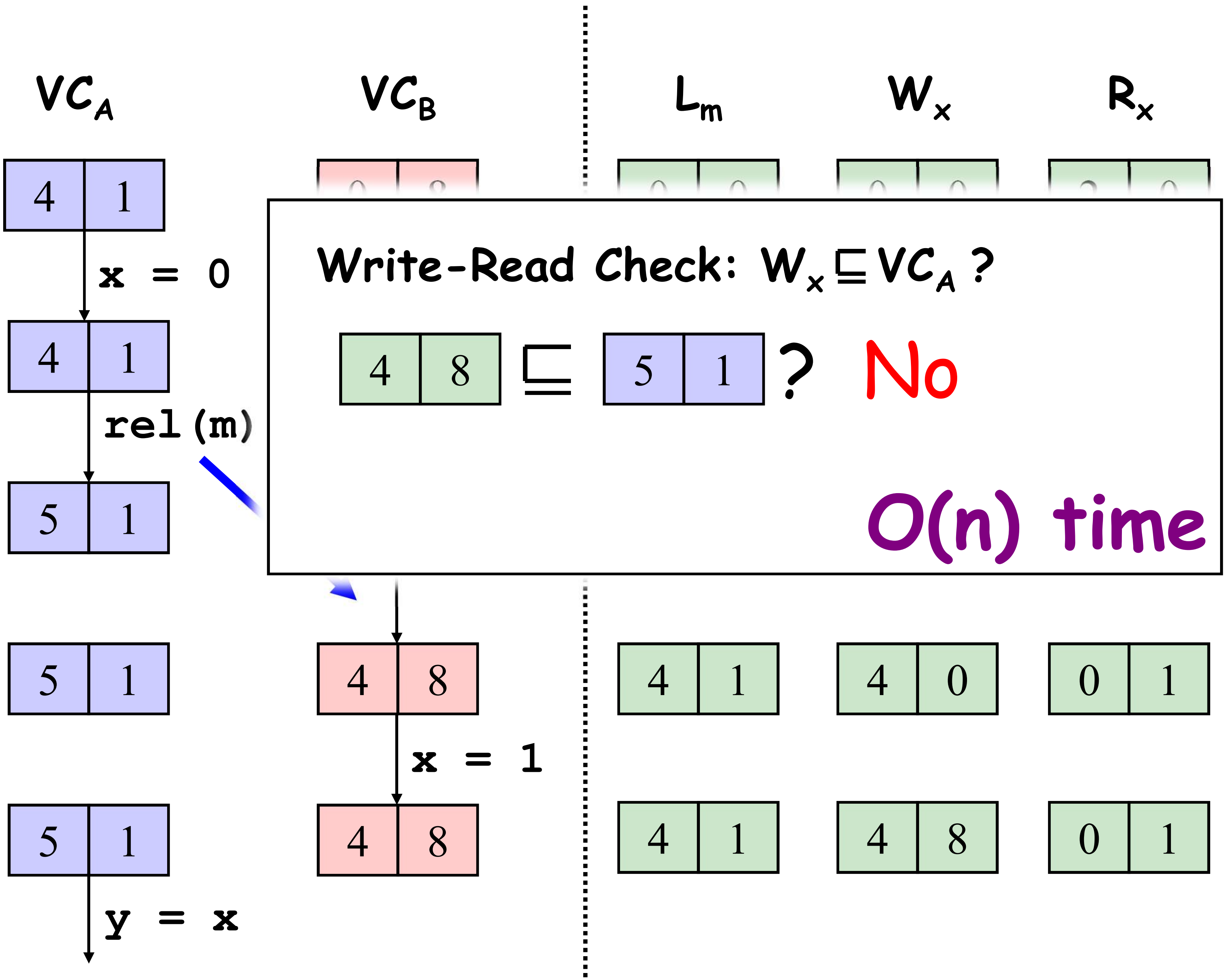
VC_A
VC_B
L_m
W_x
R_x
4 1
2 8
2 1
3 0
0 1
x = 0
4 1
2 8
2 1
4 0
0 1
rel(m)
5 1
2 8
4 1
4 0
0 1



VC_A
VC_B
L_m
W_x
R_x
4 1
2 8
2 1
3 0
0 1
x = 0
4 1
2 8
2 1
4 0
0 1
rel(m)
5 1
2 8
4 1
4 0
0 1
acq(m)
5 1
4 8
4 1
4 0
0 1



VCA
VCB
Lm
Wx
Rx
4 1
2 8
2 1
3 0
0 1
x = 0
4 1
2 8
2 1
4 0
0 1
rel(m)
5 1
2 8
4 1
4 0
0 1
acq(m)
5 1
4 8
4 1
4 0
0 1
x = 1
5 1
4 8
4 1
4 8
0 1



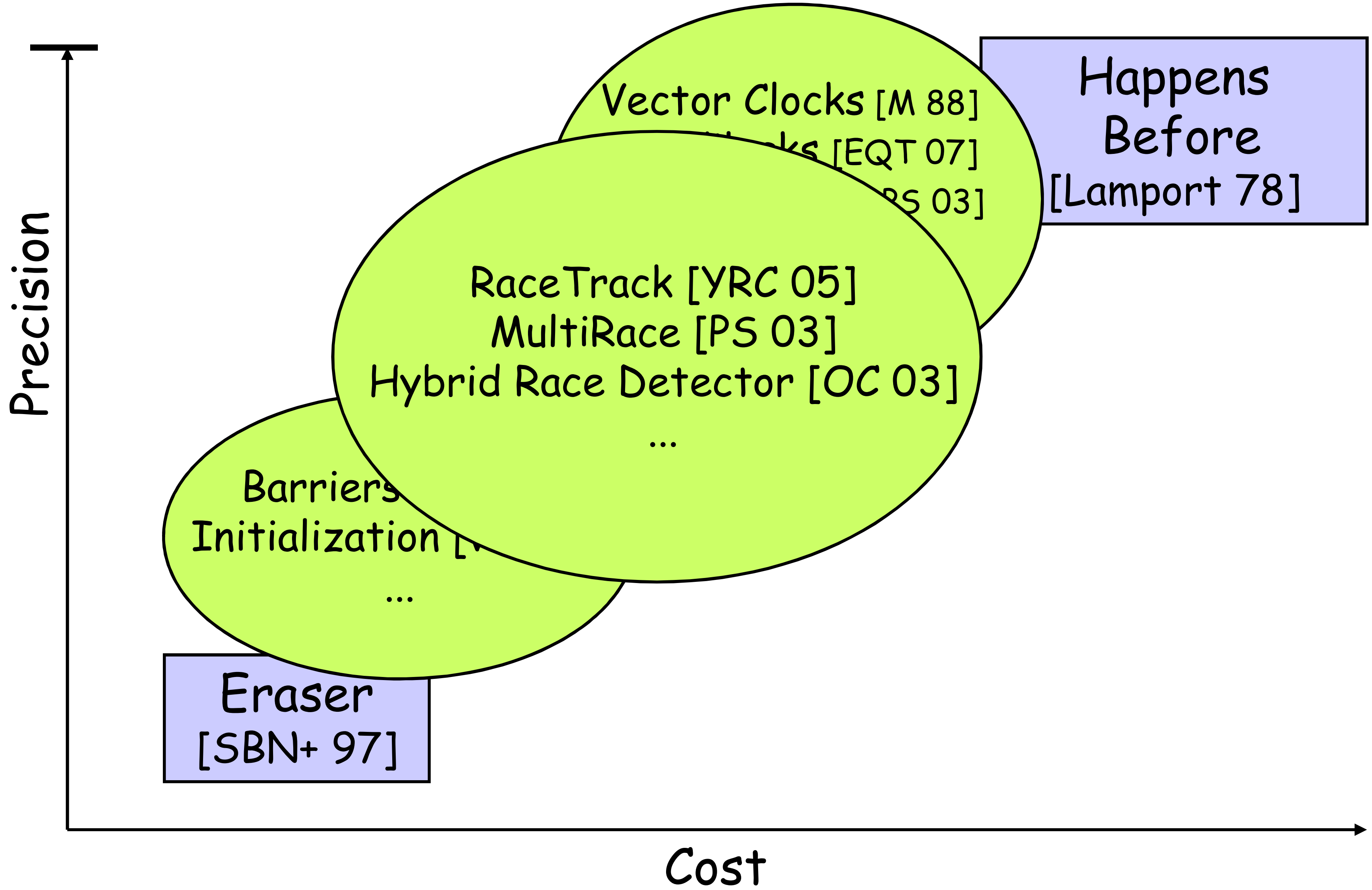
VC_A
VC_B
L_m
W_x
R_x
4 1
x = 0
4 1
rel(m)
5 1
5 1
4 8
4 1
4 0
0 1
x = 1
5 1
4 8
4 1
4 8
0 1
y = x
Write-Read Check: W_x ⊑ VC_A ?
4 8 ⊑ 5 1 ?
No
O(n) time

VectorClocks for Data-Race Detection

- Sound
  - No warnings ➔ data-race-free execution
- Complete
  - Warning ➔ data-race exists

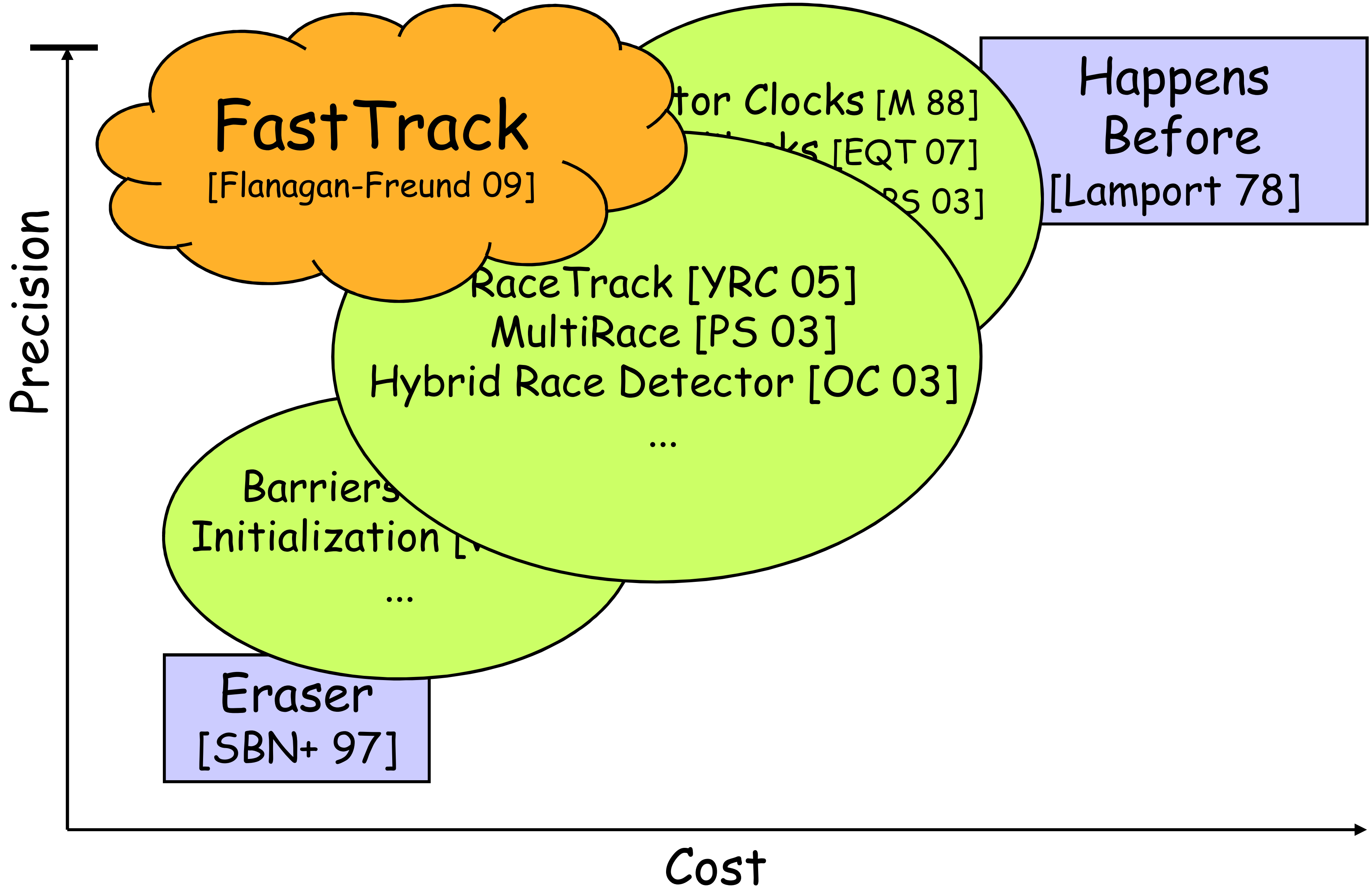
- Performance
  - slowdowns > 50x
  - memory overhead

Dynamic Data-Race Detection

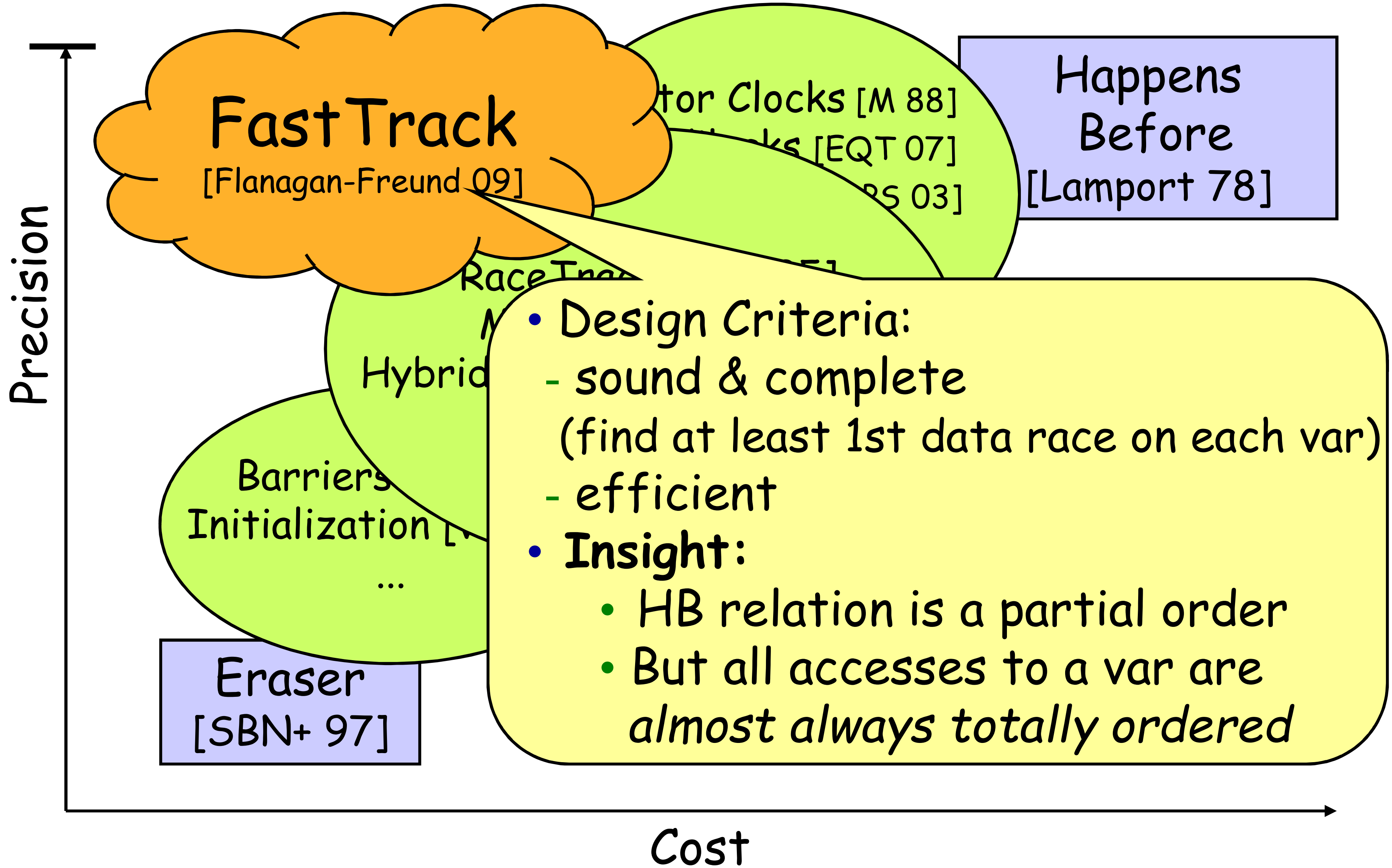


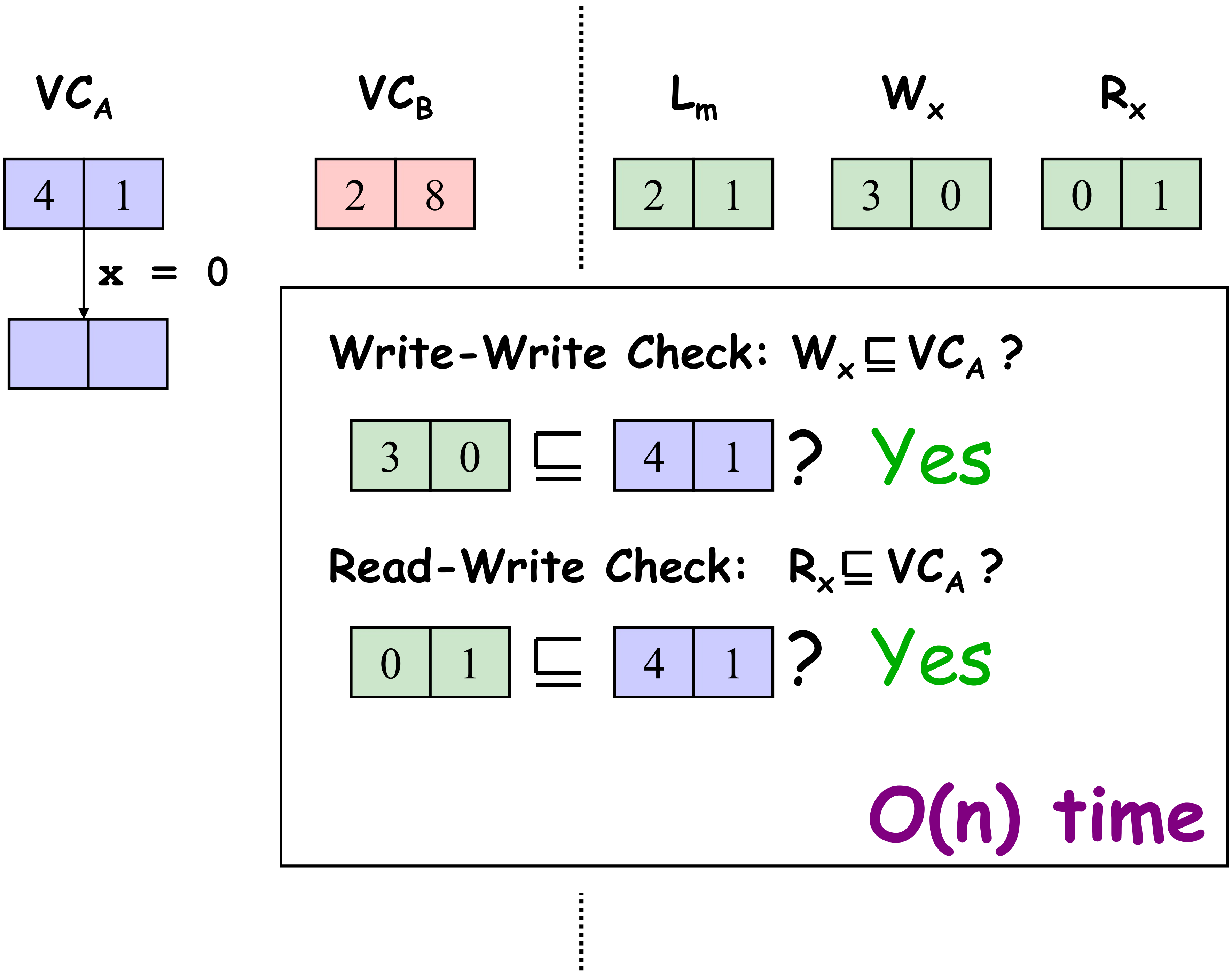
FASTTRACK

Dynamic Data-Race Detection



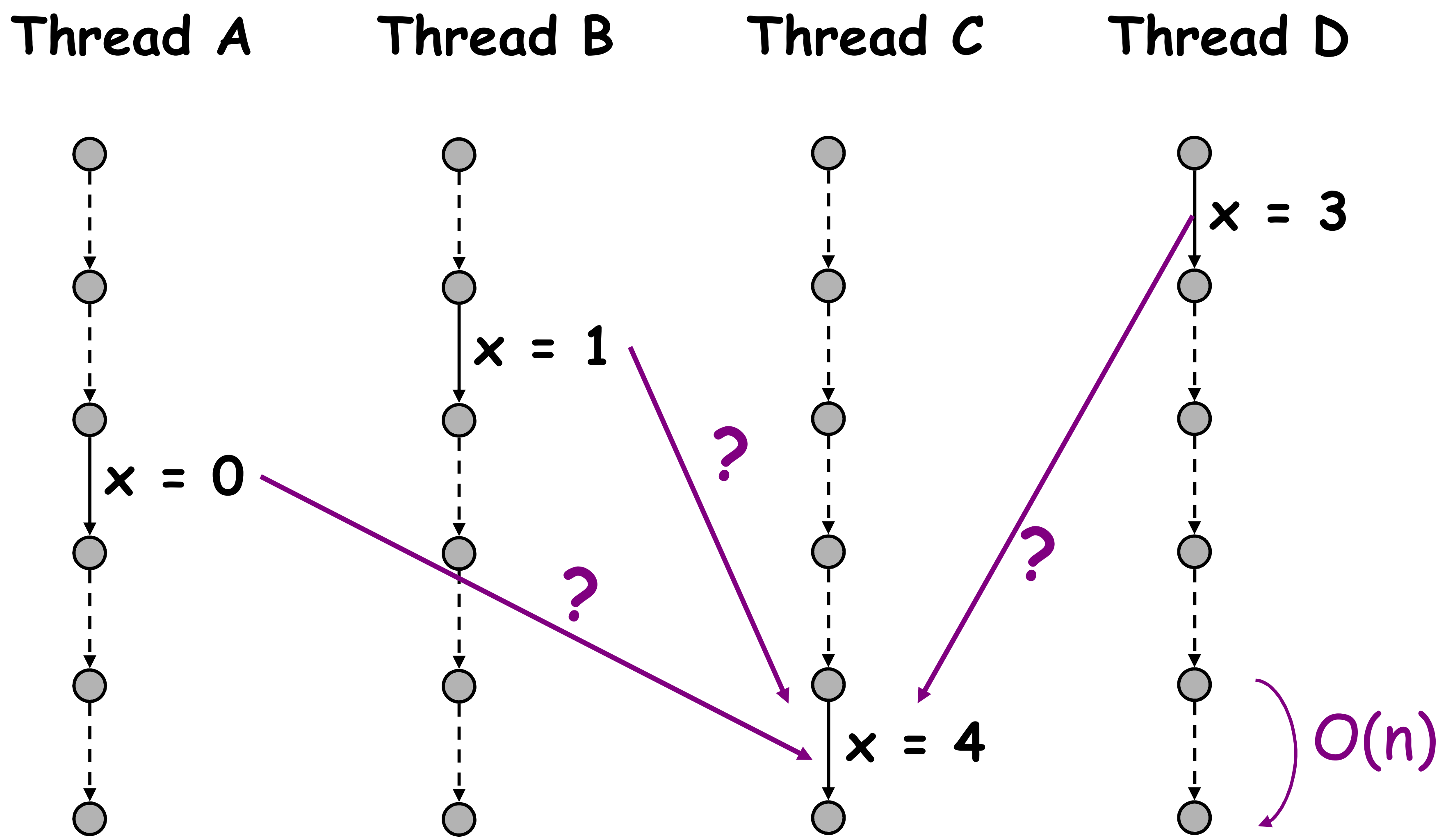
Dynamic Data-Race Detection



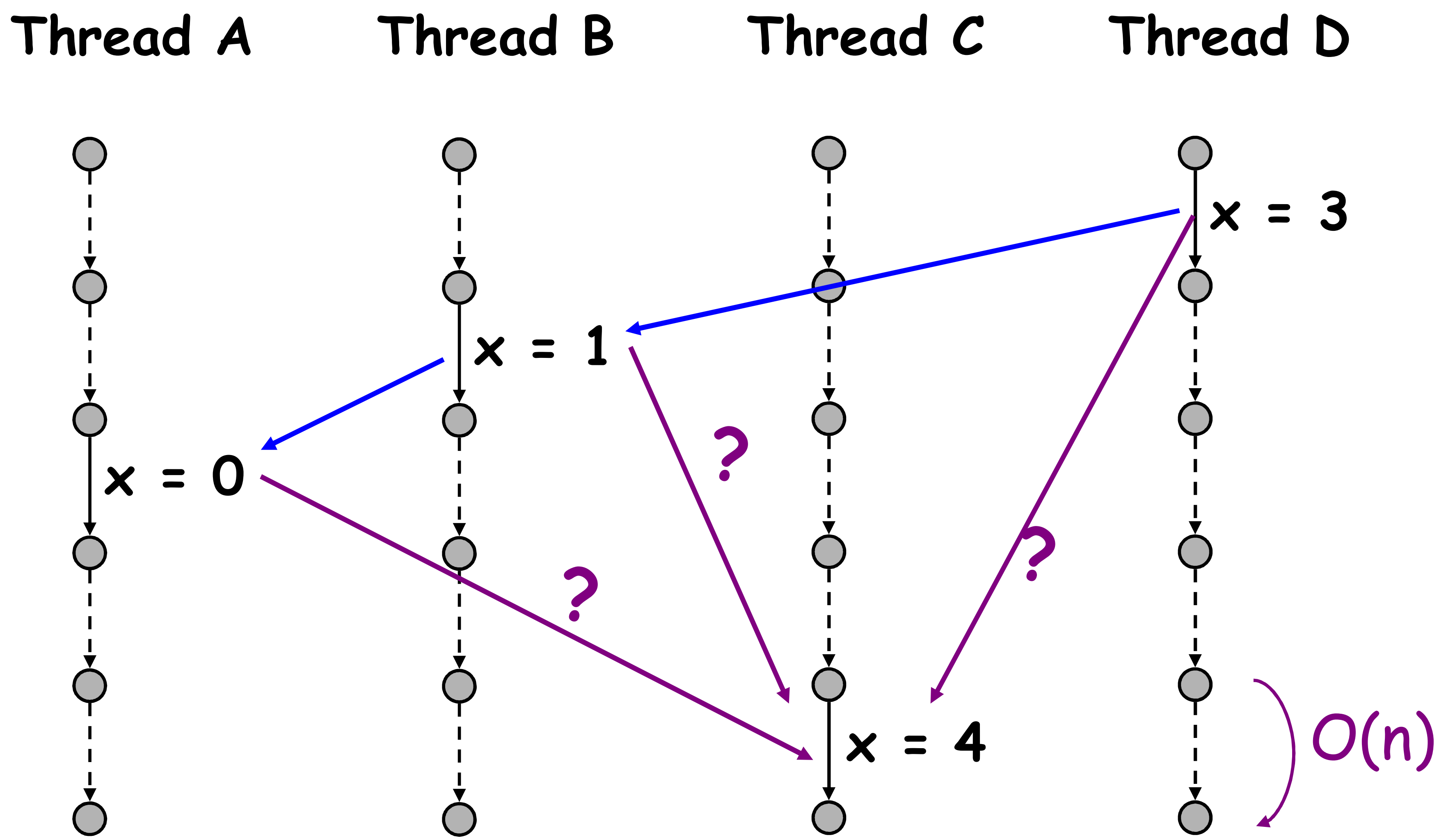


VC_A
4 1
x = 0
VC_B
2 8
L_m
2 1
W_x
3 0
R_x
0 1
Write-Write Check: W_x ⊑ VC_A ?
3 0 ⊑ 4 1 ? Yes
Read-Write Check: R_x ⊑ VC_A ?
0 1 ⊑ 4 1 ? Yes
O(n) time

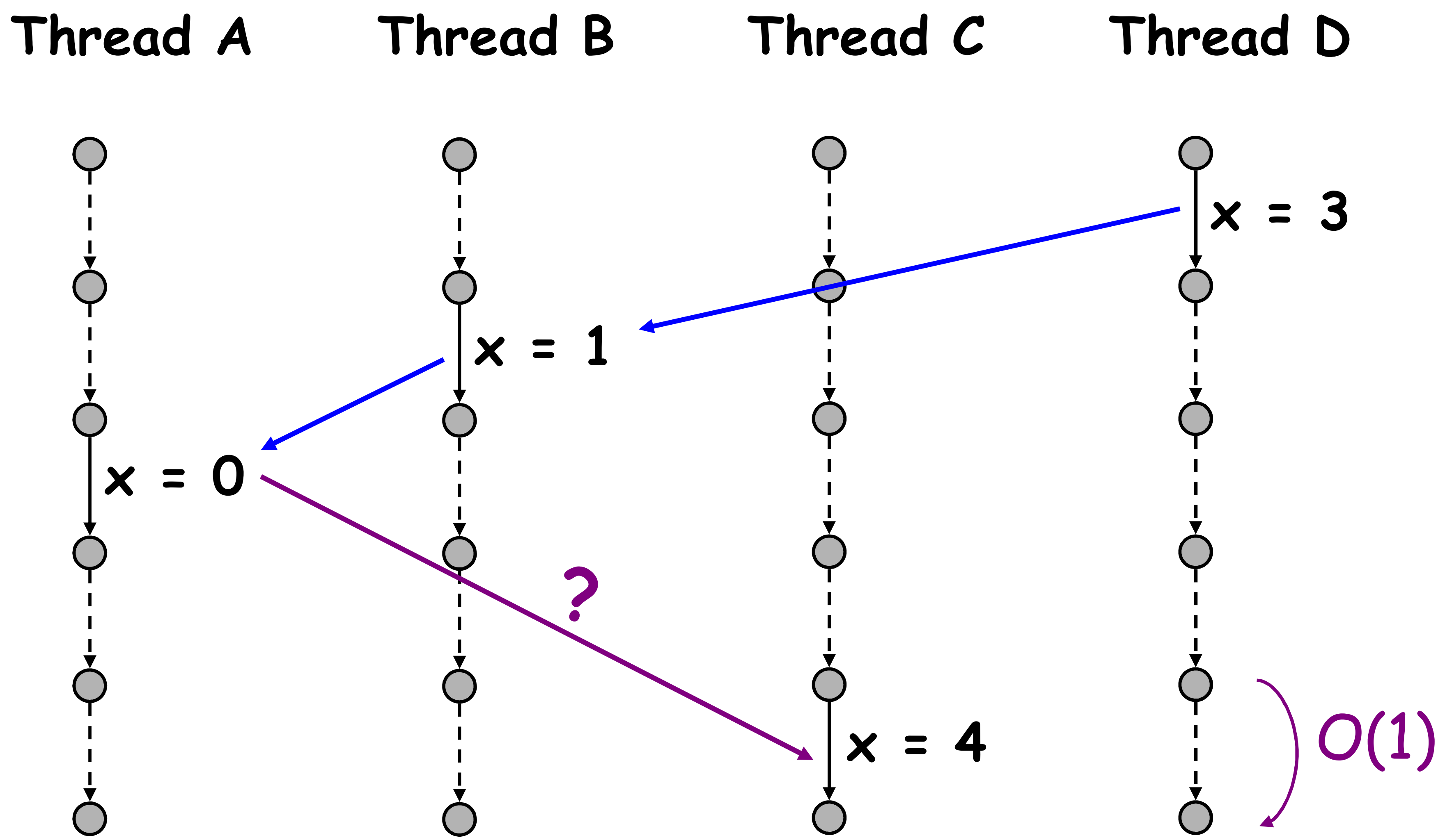
Write-Write and Write-Read Data Races

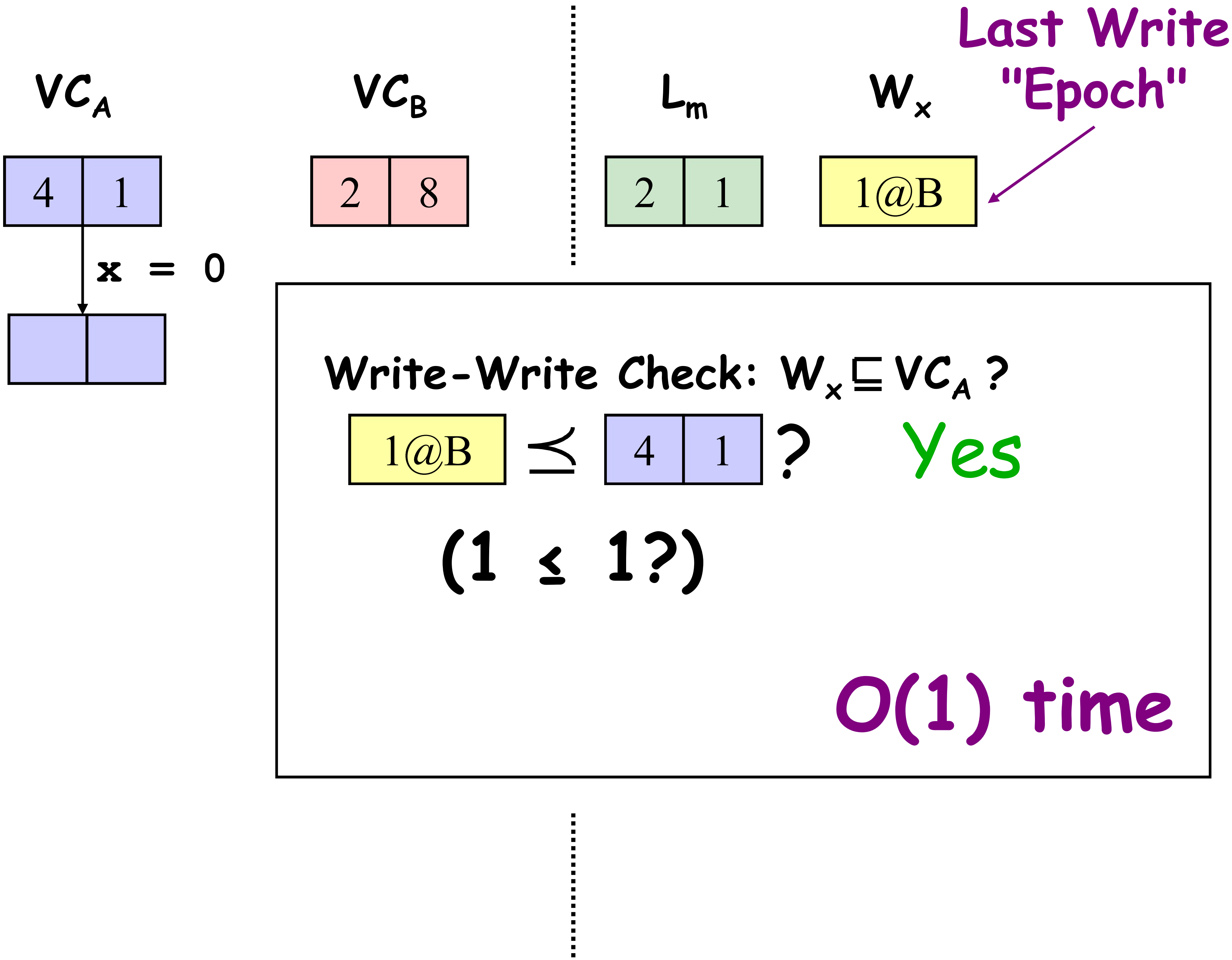


No Data Races Yet: Writes Totally Ordered

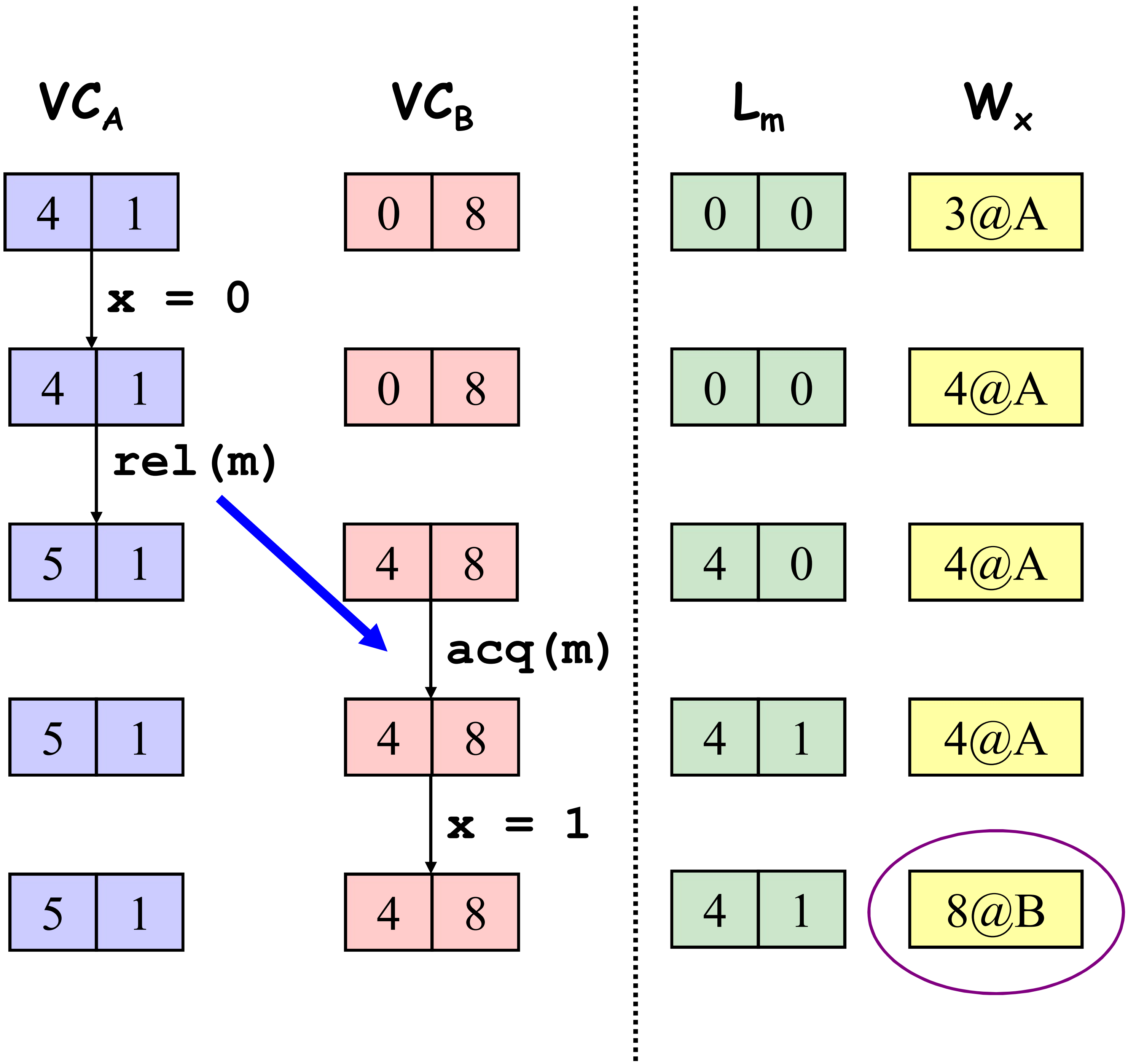


No Data Races Yet: Writes Totally Ordered

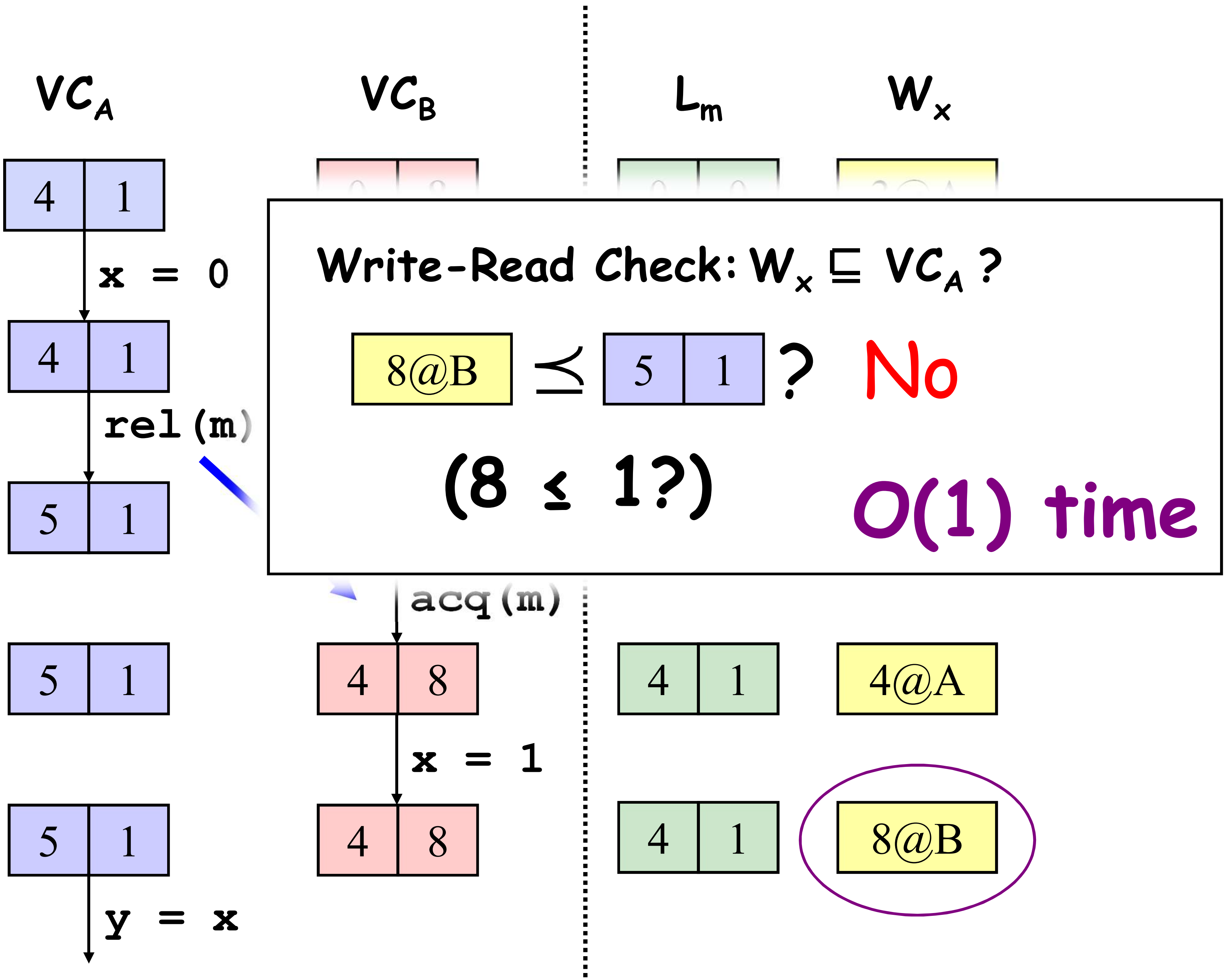




Last Write "Epoch"
VC_A
4 1
x = 0
VC_B
2 8
L_m
2 1
W_x
1@B
Write-Write Check: W_x ⊑ VC_A ?
1@B ≼ 4 1 ?
Yes
(1 ≤ 1?)
O(1) time

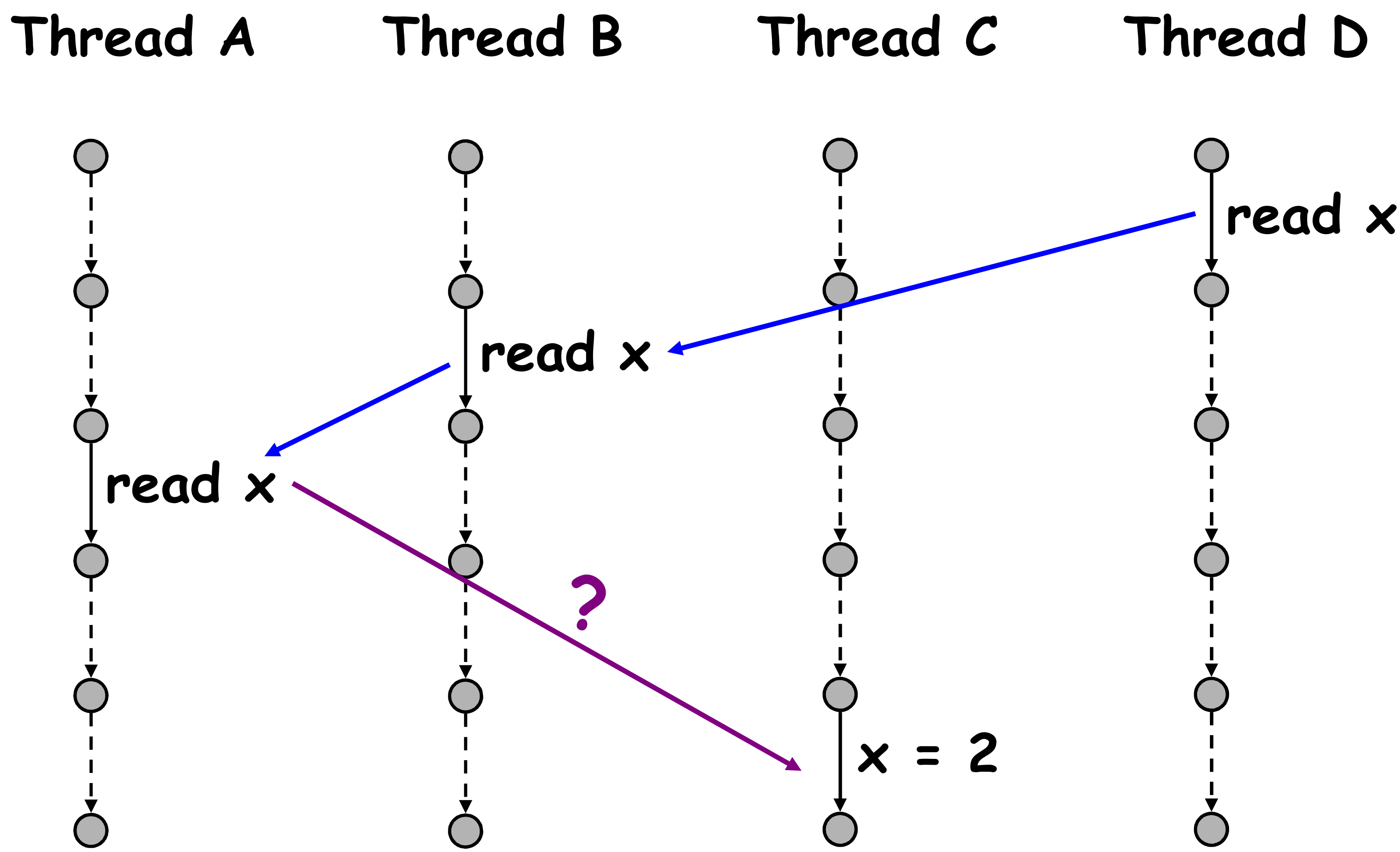


VC_A
VC_B
L_m
W_x
4 1
0 8
0 0
3@A
x = 0
4 1
0 8
0 0
4@A
rel(m)
5 1
4 8
4 0
4@A
acq(m)
5 1
4 8
4 1
4@A
x = 1
5 1
4 8
4 1
8@B



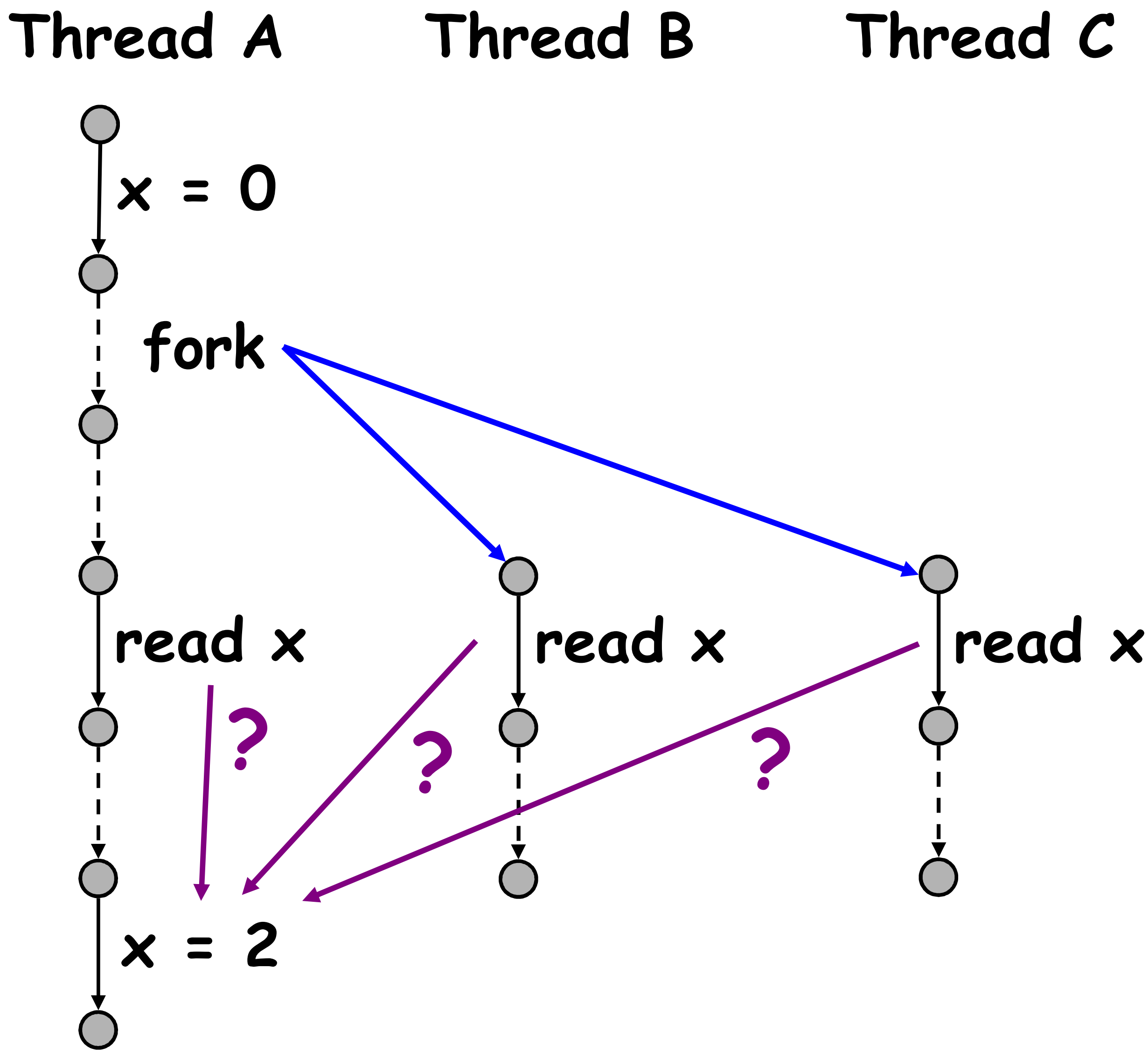
VC_A
VC_B
L_m
W_x
4 1
x = 0
4 1
rel(m)
5 1
Write-Read Check: $W_x \sqsubseteq VC_A$?
8@B $\preceq$ 5 1 ? No
(8 ≤ 1?)
O(1) time
acq(m)
5 1
4 8
4 1
4@A
x = 1
5 1
4 8
4 1
8@B
y = x

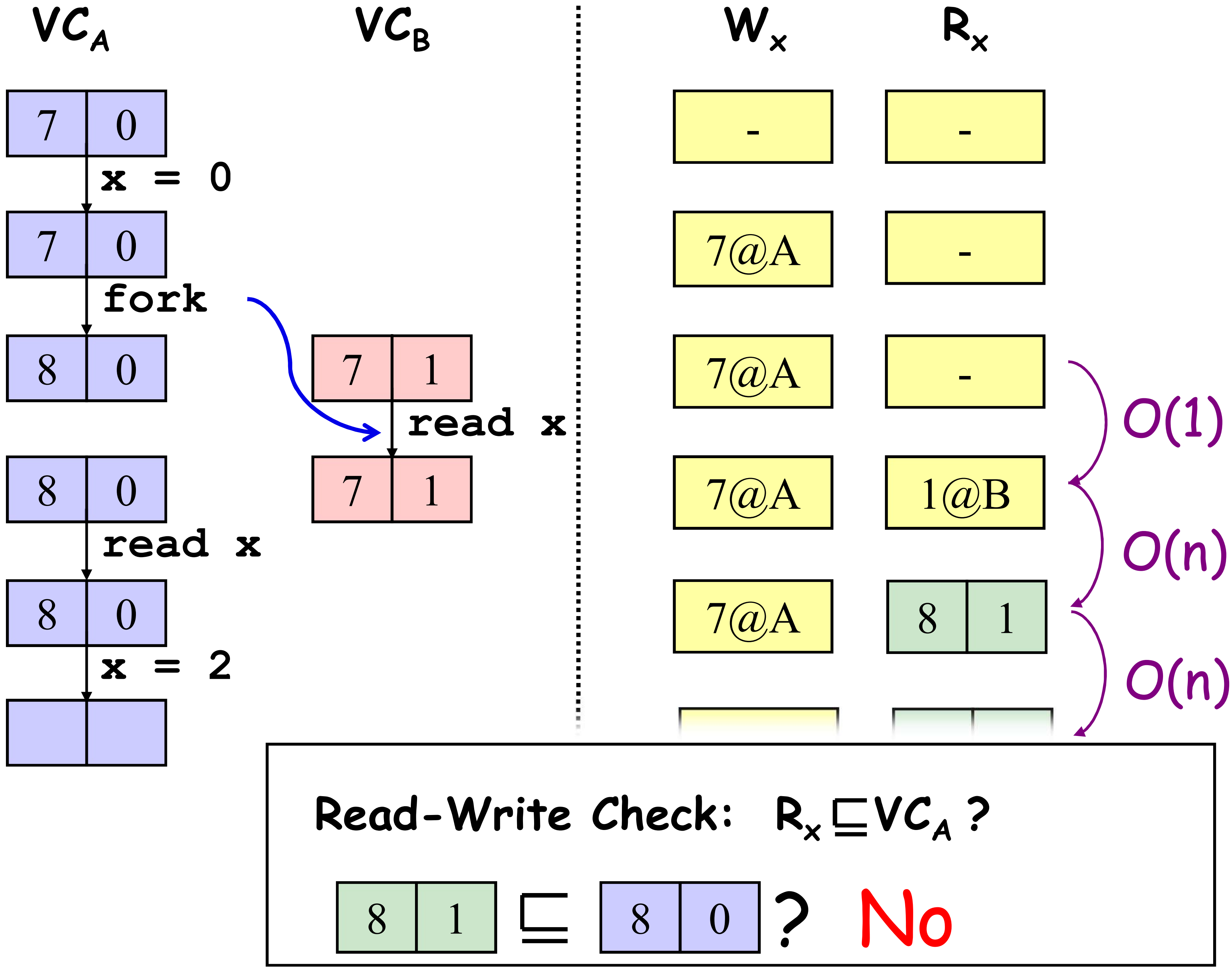
Read-Write Data Races -- Ordered Reads



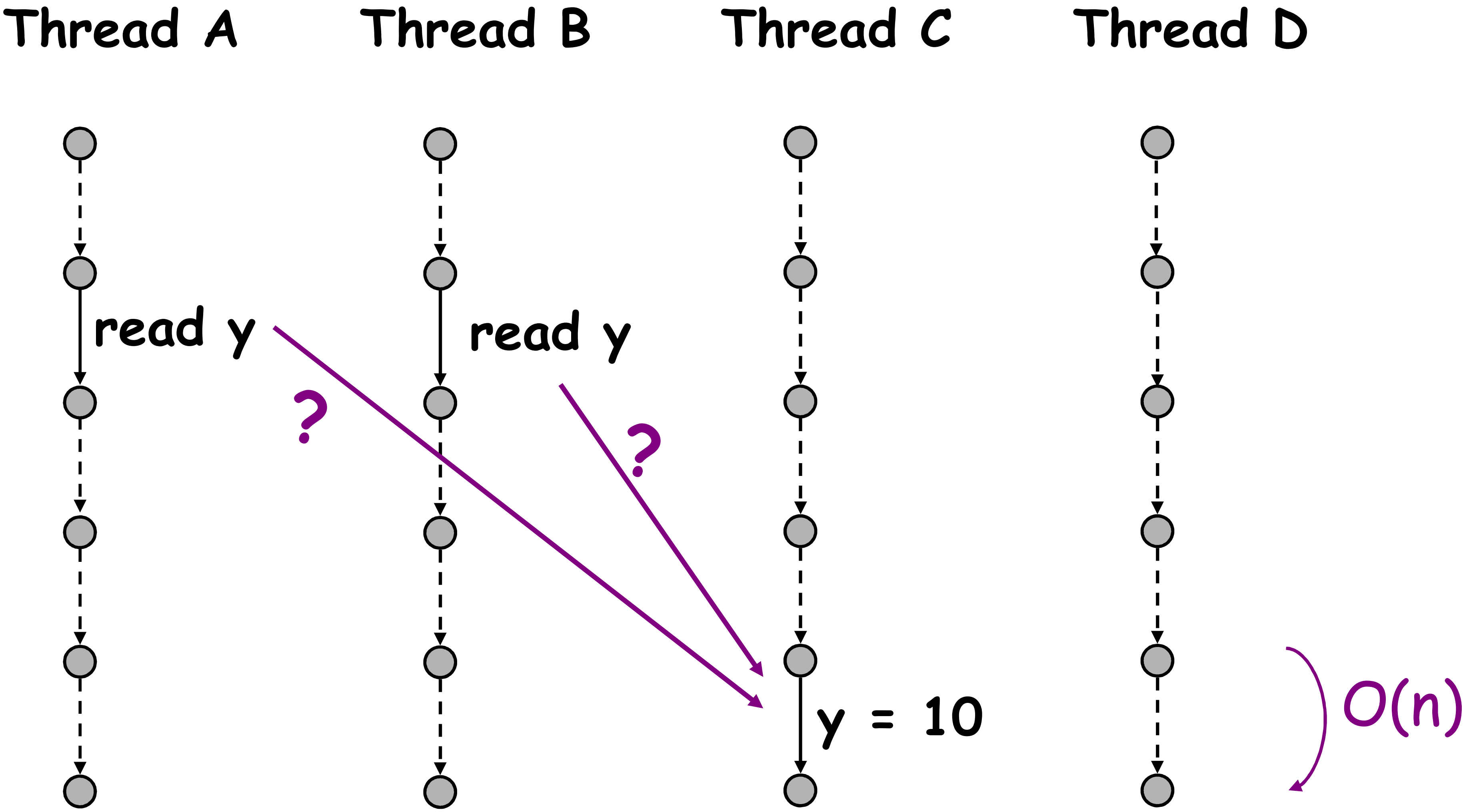
Most common case: thread-local, lock-protected, ...

Read-Write Data Races -- Unordered Reads

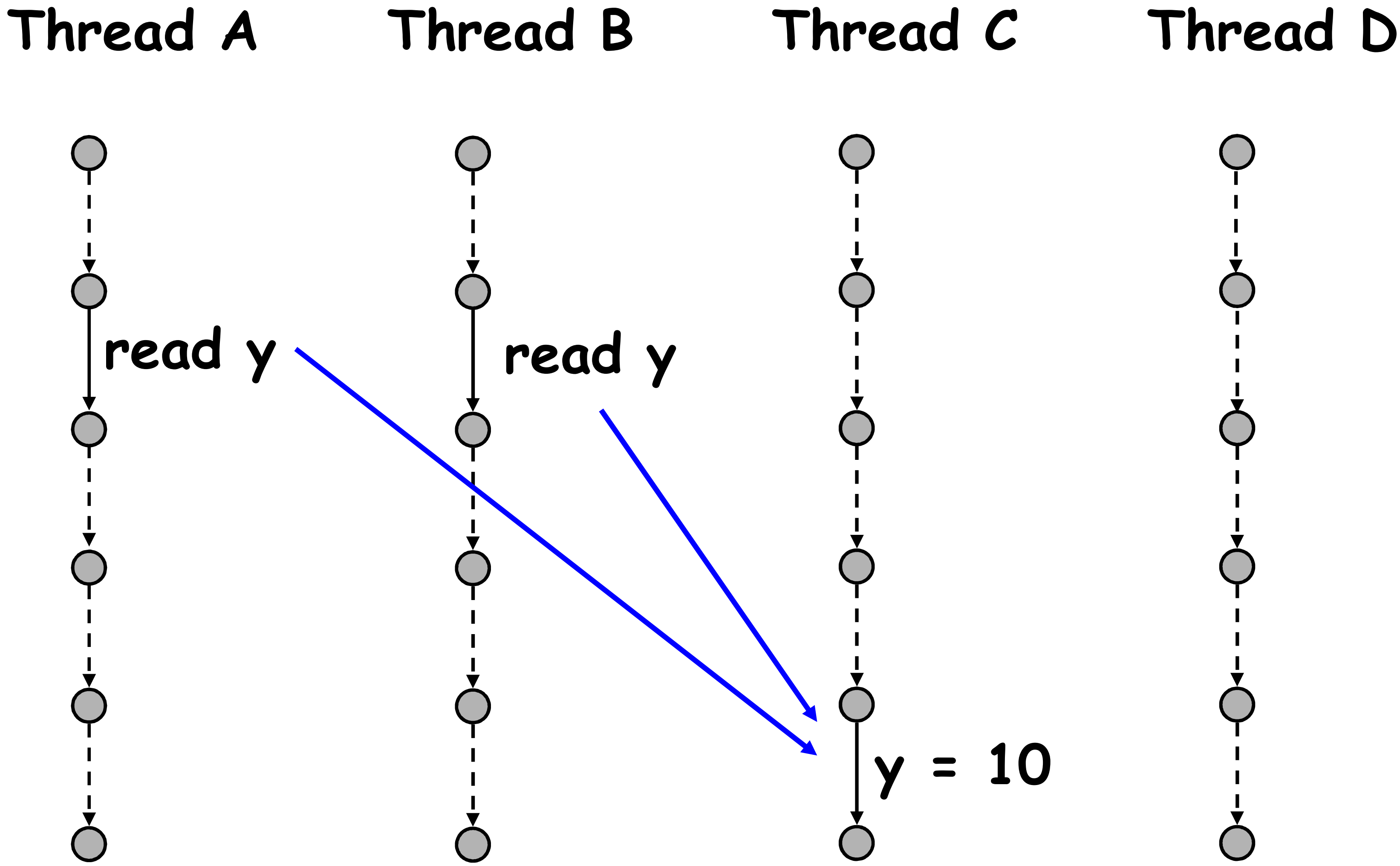




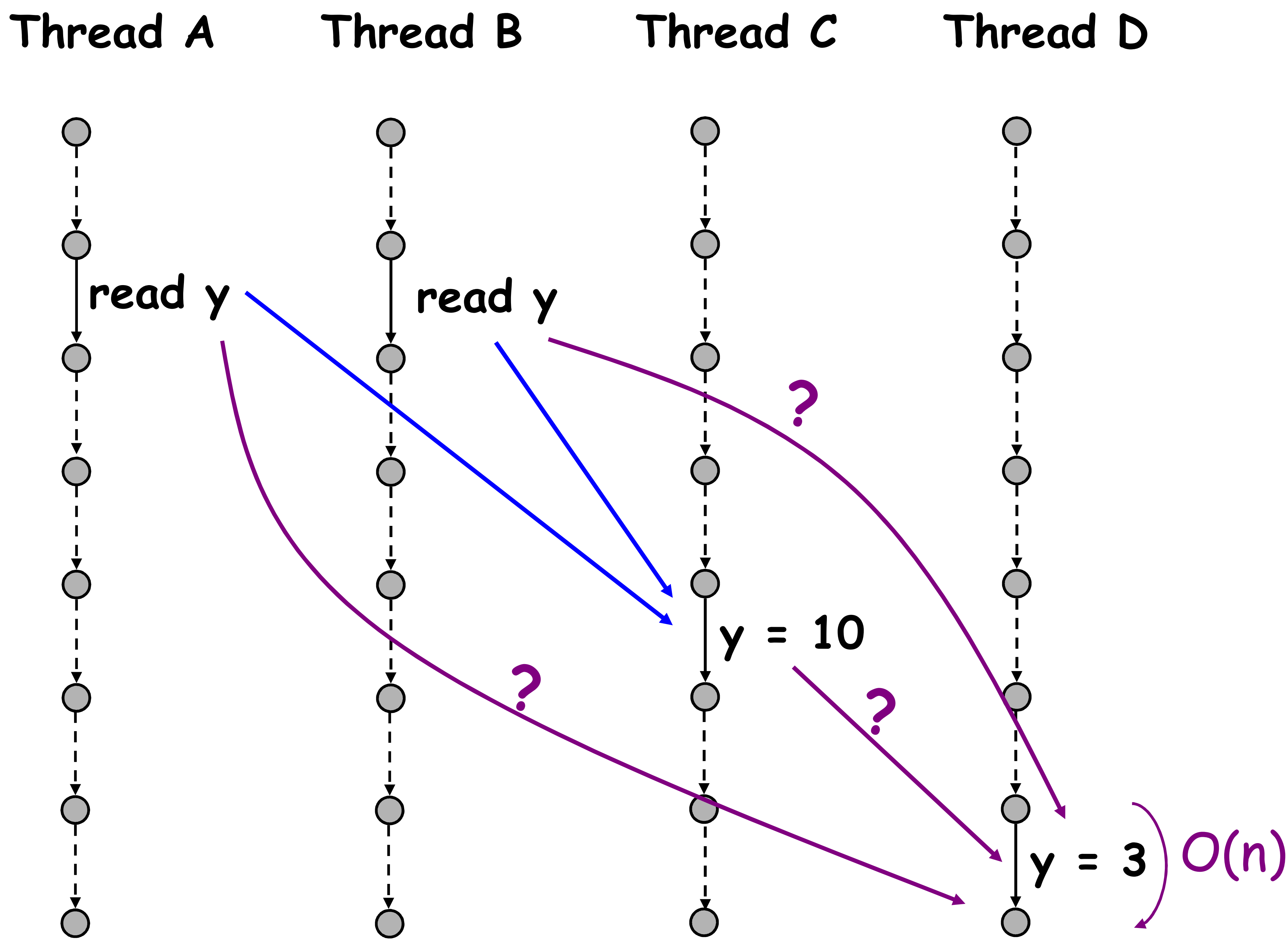
VC_A
VC_B
W_x
R_x
7 0
x = 0
7 0
fork
8 0
7 1
read x
7 1
8 0
read x
8 0
x = 2
-
-
7@A
-
7@A
-
7@A
1@B
7@A
8 1
O(1)
O(n)
O(n)
Read-Write Check: R_x ⊑ VC_A ?
8 1 ⊑ 8 0 ? No

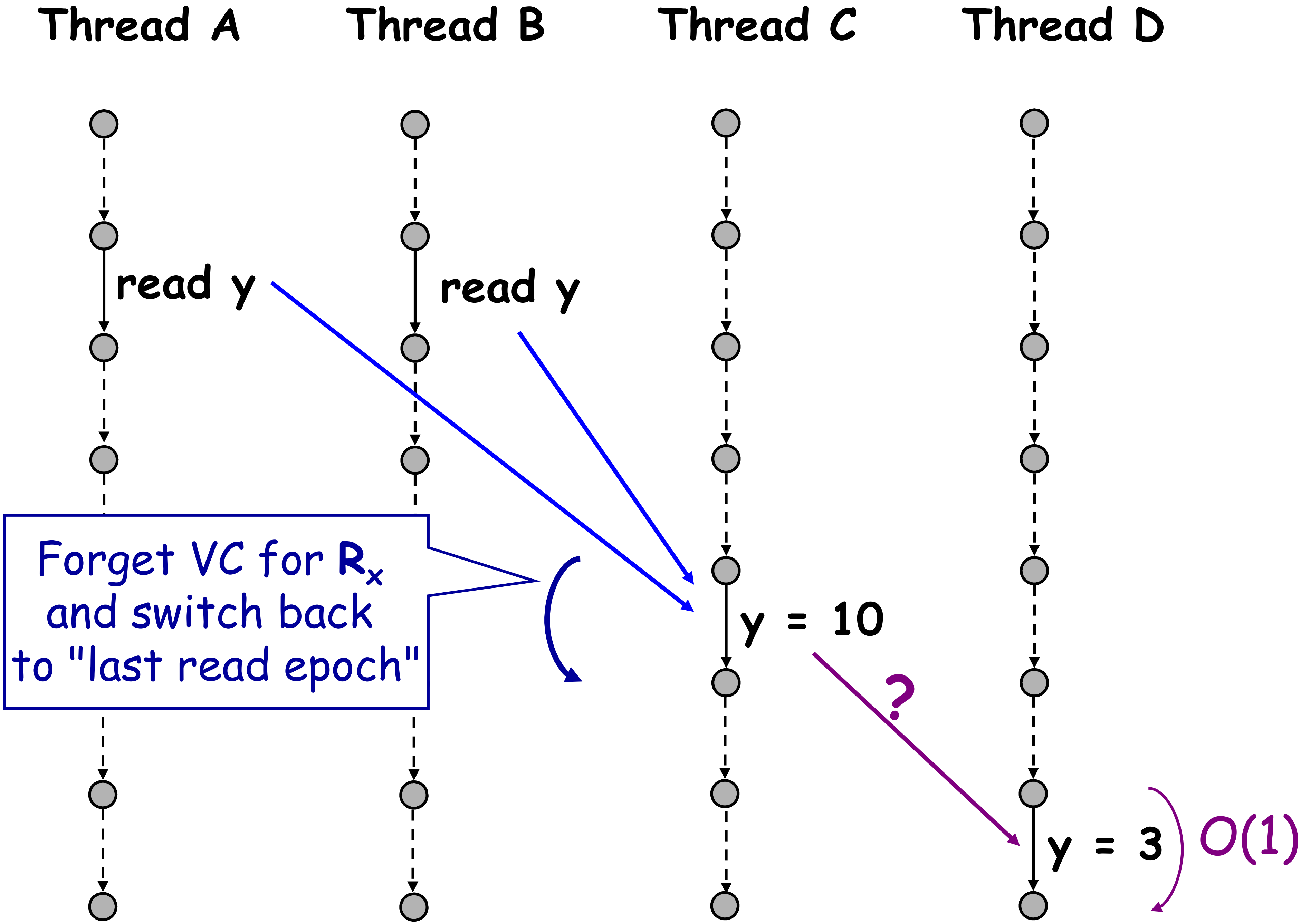


Thread A
Thread B
Thread C
Thread D
read y
read y
?
?
y = 10
O(n)



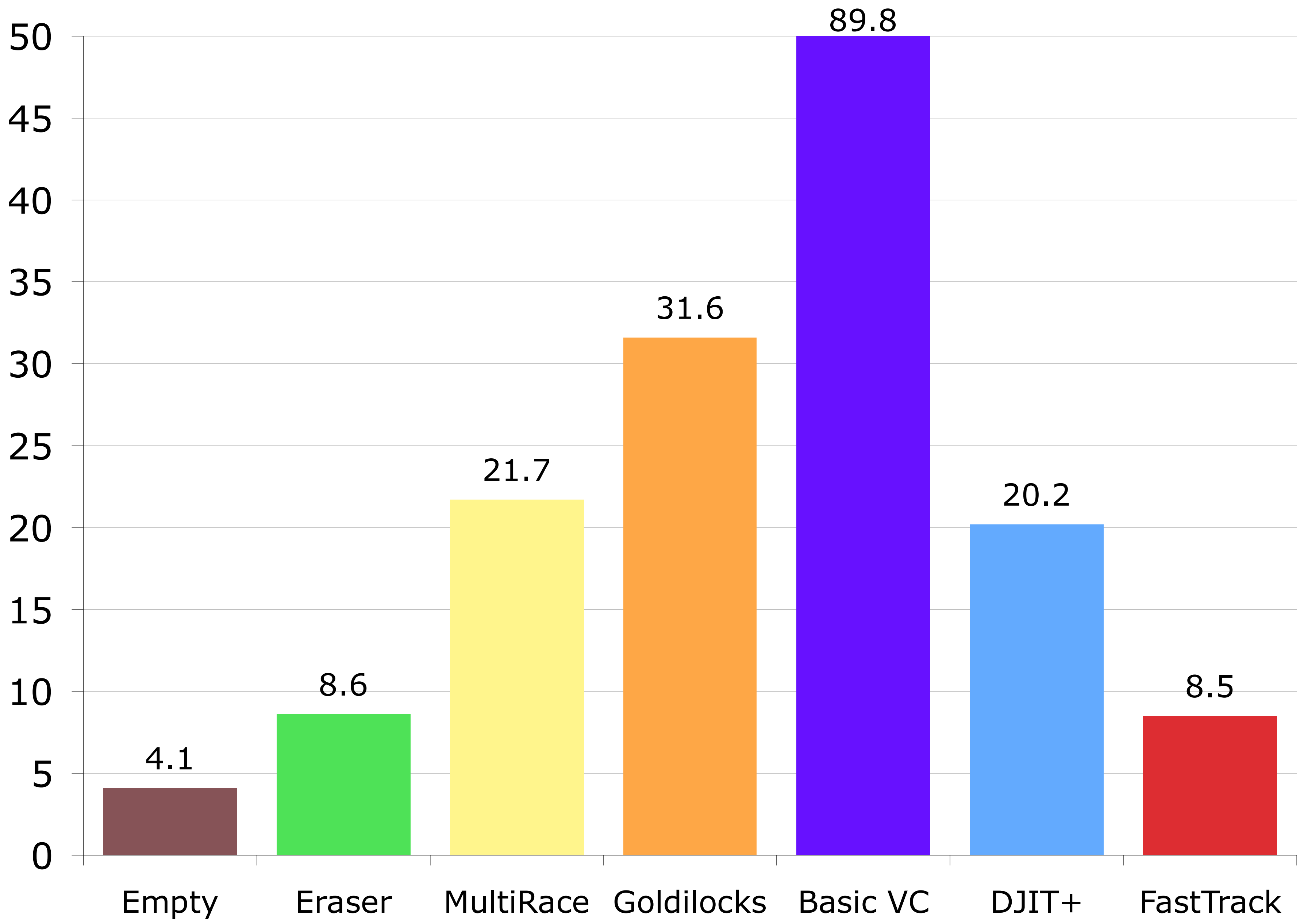
Thread A
Thread B
Thread C
Thread D
read y
read y
y = 10


Thread A
Thread B
Thread C
Thread D
read y
read y
y = 10
y = 3
?
?
?
O(n)



Thread A
Thread B
Thread C
Thread D
read y
read y
Forget VC for R_x and switch back to "last read epoch"
y = 10
?
y = 3
O(1)

Slowdown (x Base Time)



Memory Usage

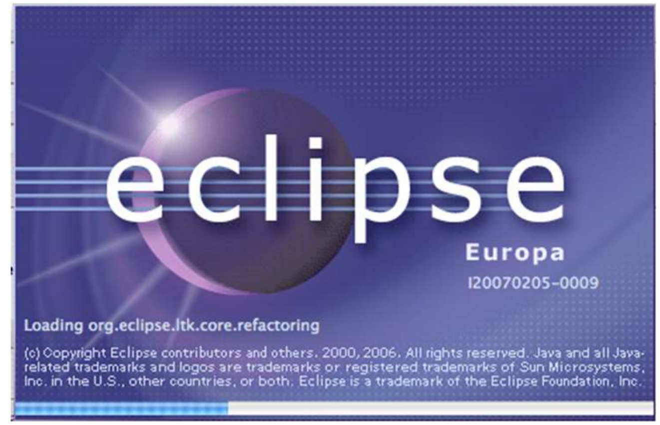
- FastTrack allocated ~200x fewer VCs

Checker	Memory Overhead
Basic VC, DJIT+	7.9x
FastTrack	2.8x
Empty	2.0x

(Note: VCs for dead objects are garbage collected)

- Improvements
  - accordion clocks [CB 01]
  - analysis granularity [PS 03, YRC 05]

Eclipse 3.4



- Scale
  - > 6,000 classes
  - 24 threads
  - custom sync. idioms
- Precision (tested 5 common tasks)
  - Eraser: ~1000 warnings
  - FastTrack: ~30 warnings
- Performance on compute-bound tasks
  - > 2x speed of other precise checkers
  - same as Eraser

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

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.