Duke Systems

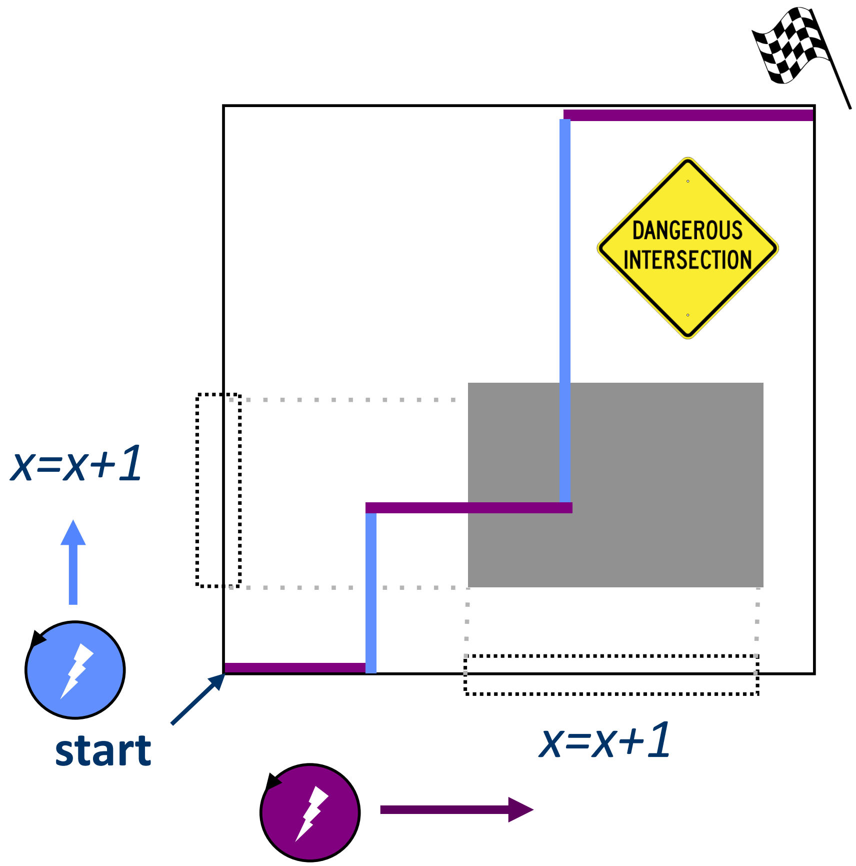
CPS 510

Races and Eraser
Plus a taste of memory models

Jeff Chase

Duke University

A race



- **Concurrent** threads
  - Arbitrarily interleaved
- **Conflicting** accesses
  - Shared variable **x**
  - Write (store) to **x**

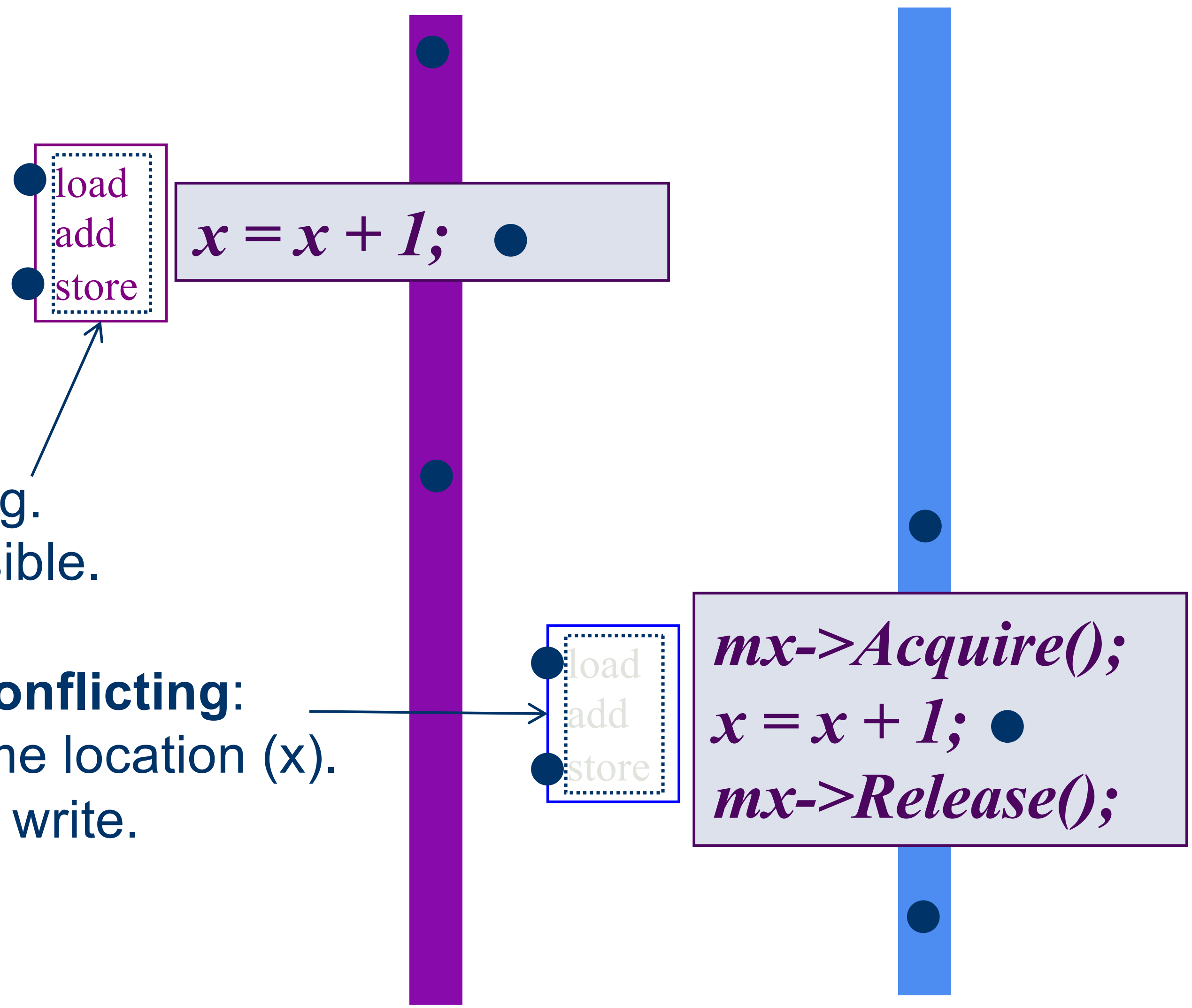
→ **Bug: unsafe!**
  - Depends on schedule
  - E.g., may “lose” counts
- **Solution**: “lock it down”.
  - Lock critical sections

Another race

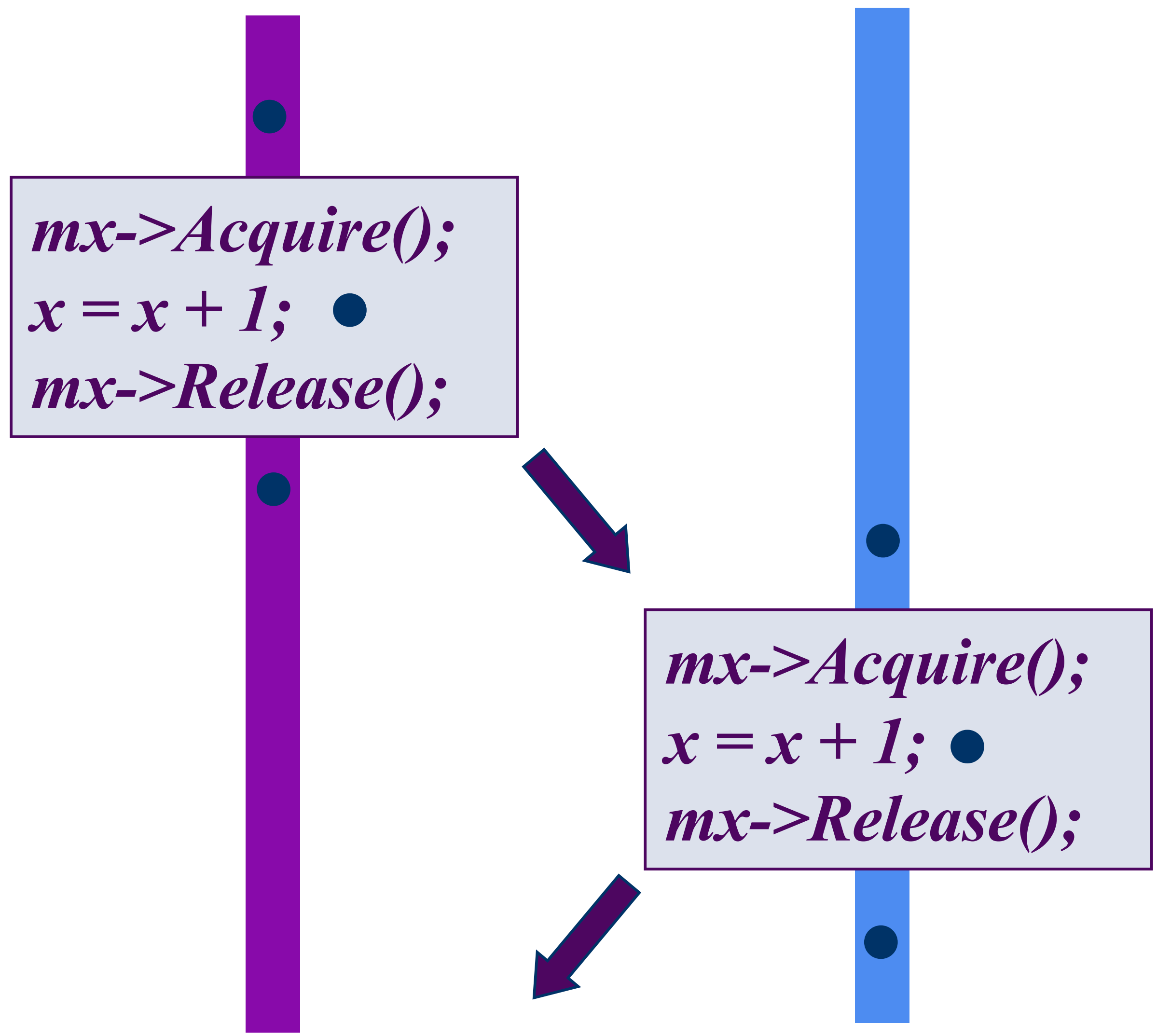
No lock! → No ordering.
Any interleaving is possible.

These operations are **conflicting**:

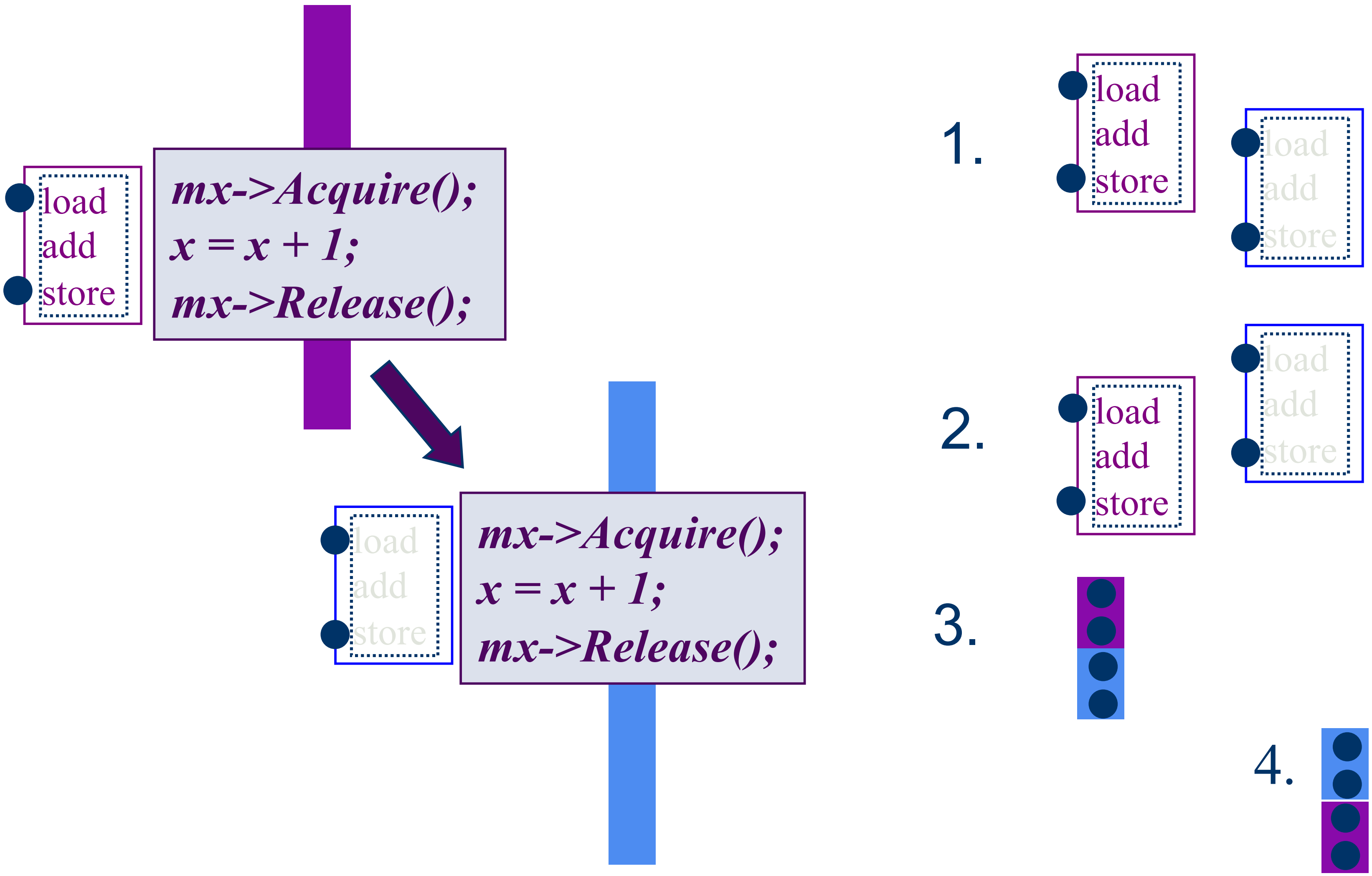
- They access the same location (x).
- And at least one is a write.



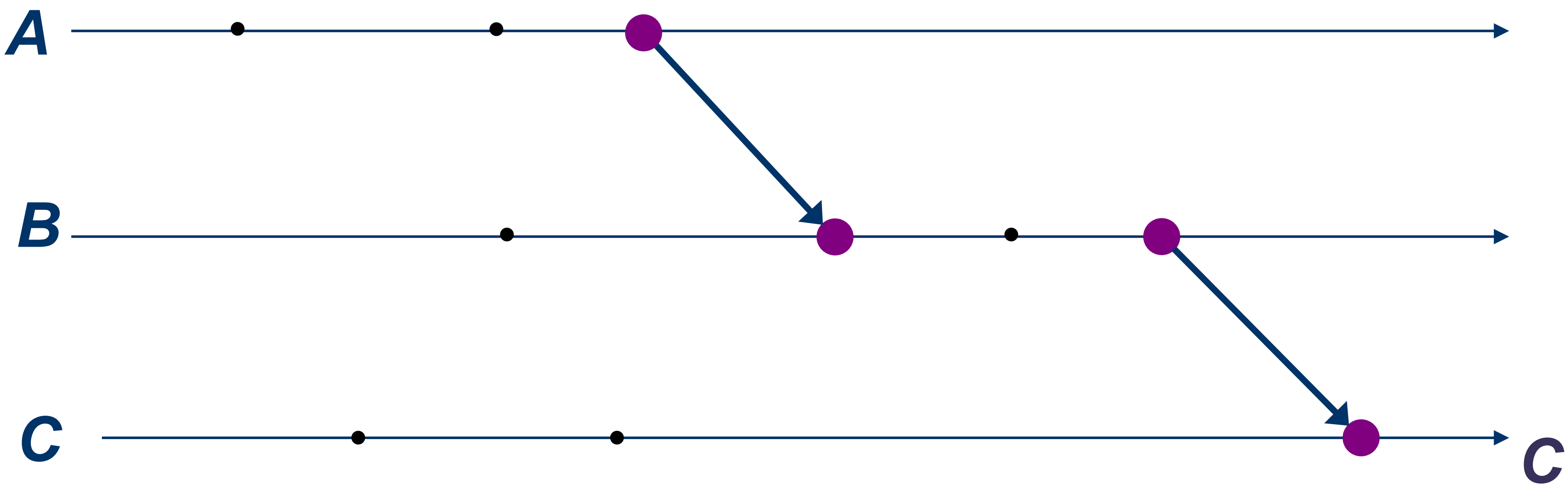
Locks and ordering



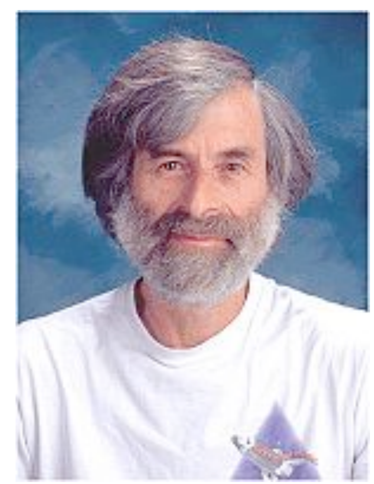
Possible interleavings?



Concurrency and time

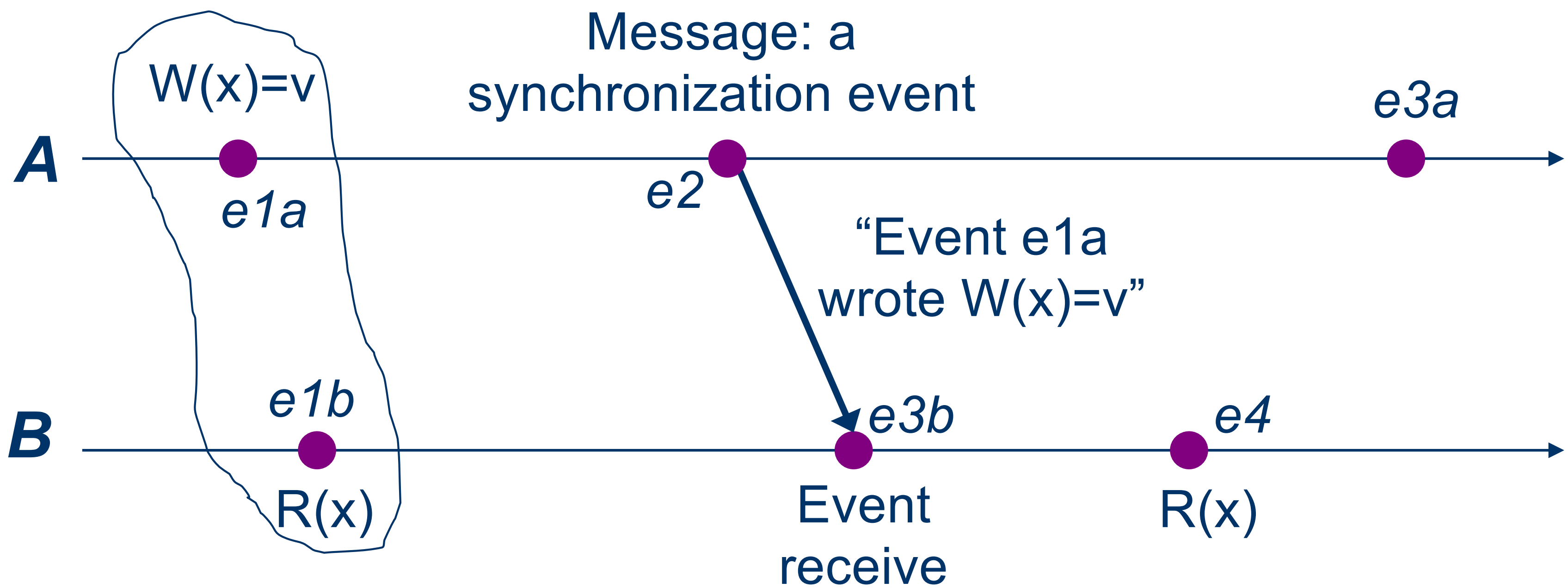


What do *before* and *after* mean in a concurrent world?



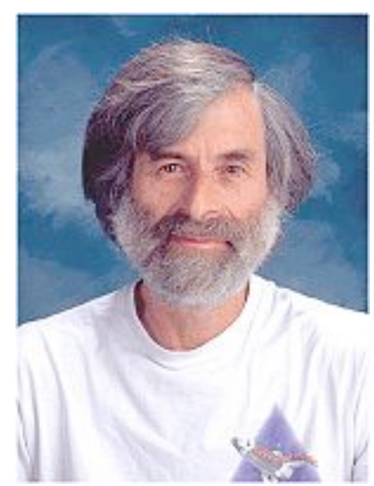
Time, Clocks, and the Ordering of Events in Distributed systems, by Leslie Lamport, CACM 21(7), July 1978

Same world, different timelines

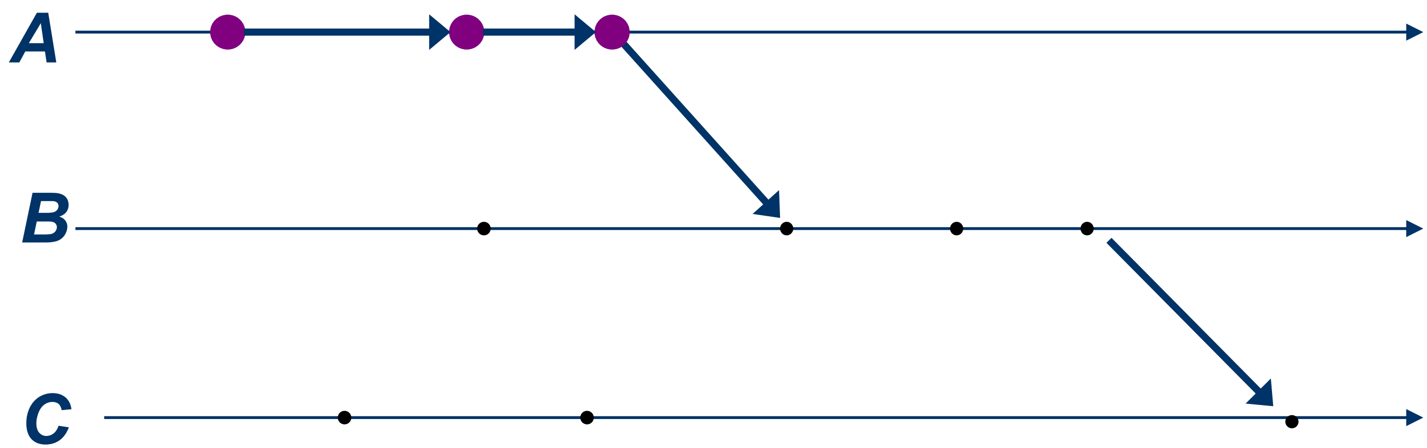


Which of these
happened first?

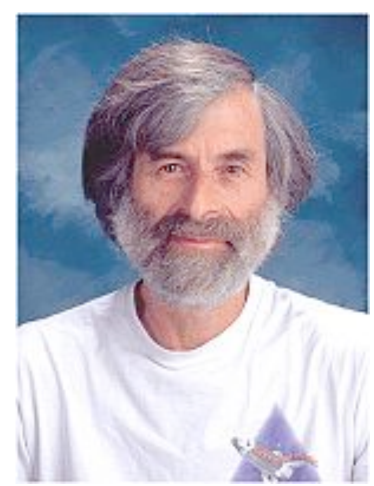
e1a is concurrent with e1b
e3a is concurrent with e3b and e4
What about the others?



Axiom 1: *happened-before* (→)

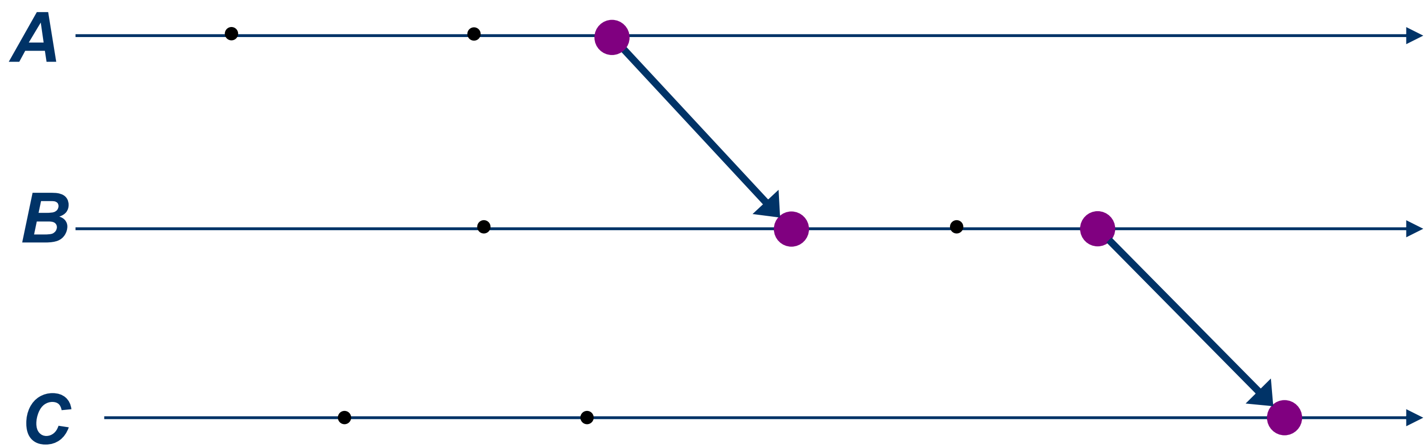


1. If *e1, e2* are in the same process/node, and *e1* comes before *e2*, then *e1*→*e2*.
   - Also called program order

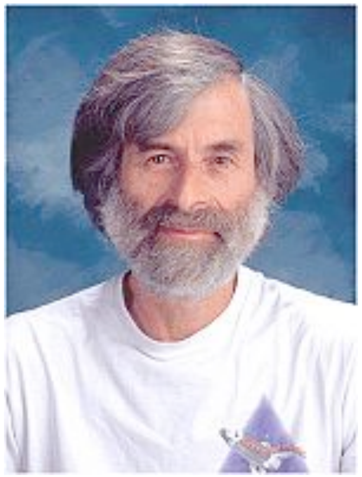


Time, Clocks, and the Ordering of Events in Distributed systems, by Leslie Lamport, CACM 21(7), July 1978

Axiom 2: *happened-before* (→)

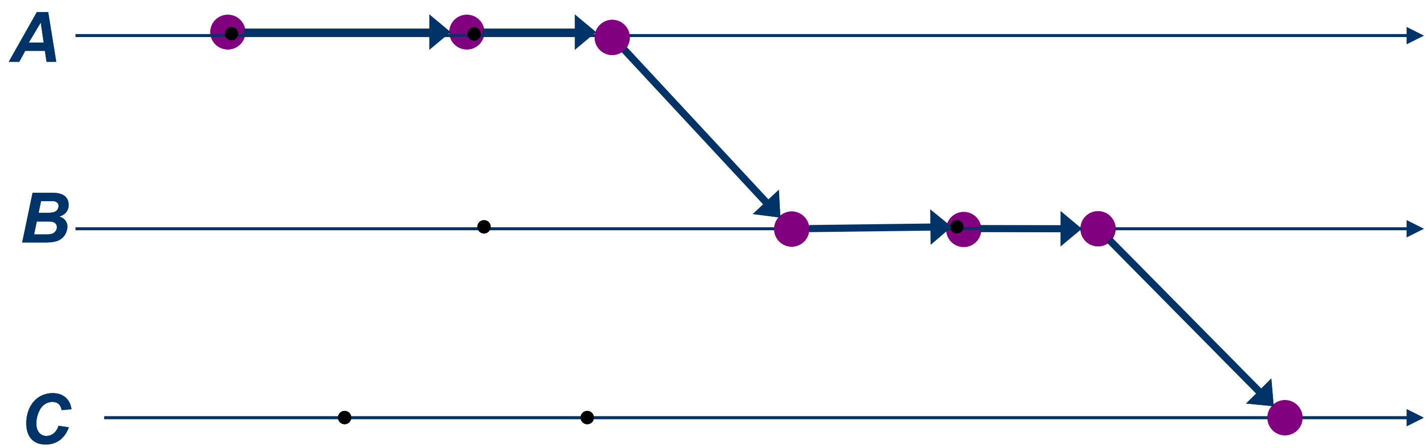


2. If *e1* is a message send, and *e2* is the corresponding receive, then *e1* → *e2.*



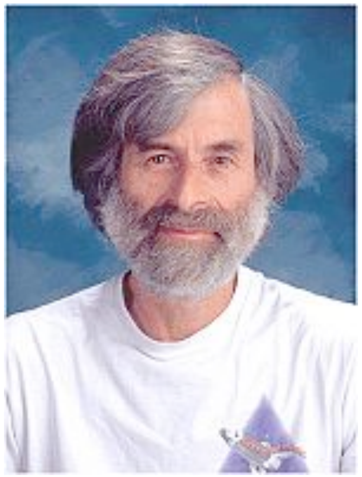
Time, Clocks, and the Ordering of Events in Distributed systems, by Leslie Lamport, CACM 21(7), July 1978

Axiom 3: *happened-before* (→)



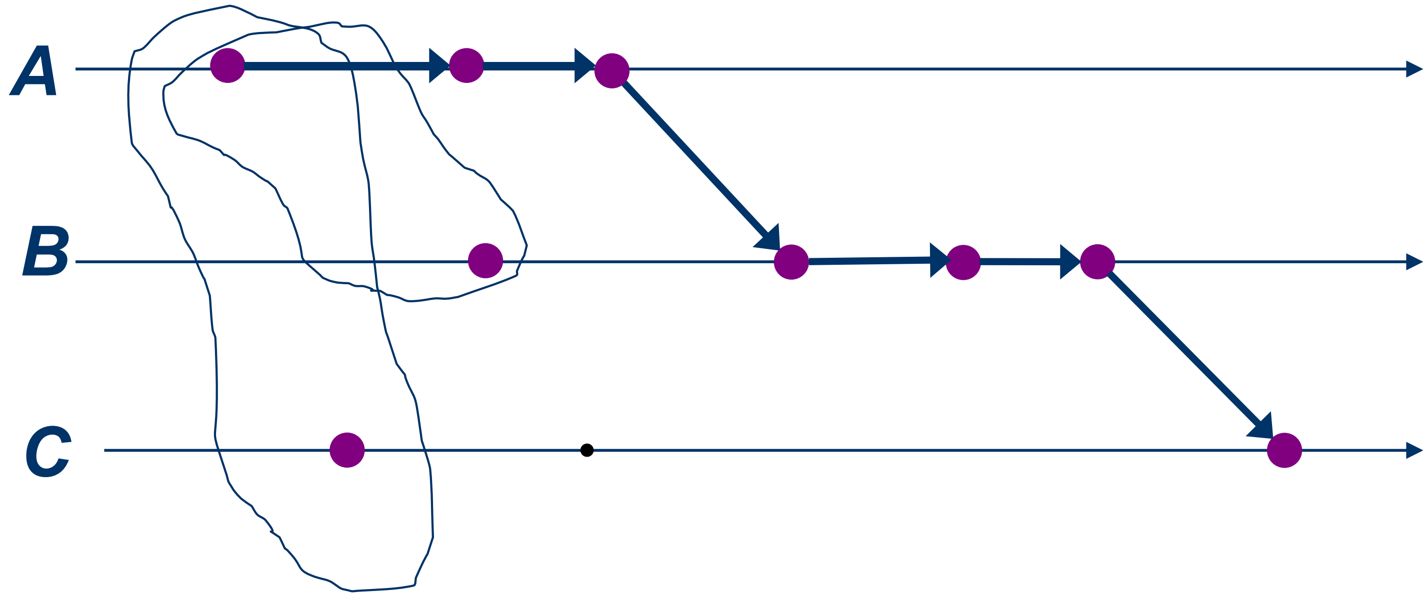
3. → is transitive

happened-before is the transitive closure of the relation defined by #1 and #2



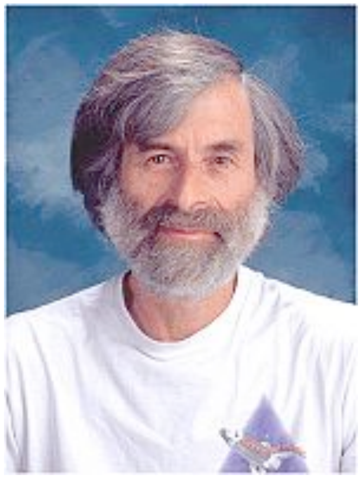
Time, Clocks, and the Ordering of Events in Distributed systems, by Leslie Lamport, CACM 21(7), July 1978

Lamport *happened-before* (→)



Two events are **concurrent** if neither **happens-before** the other.

C

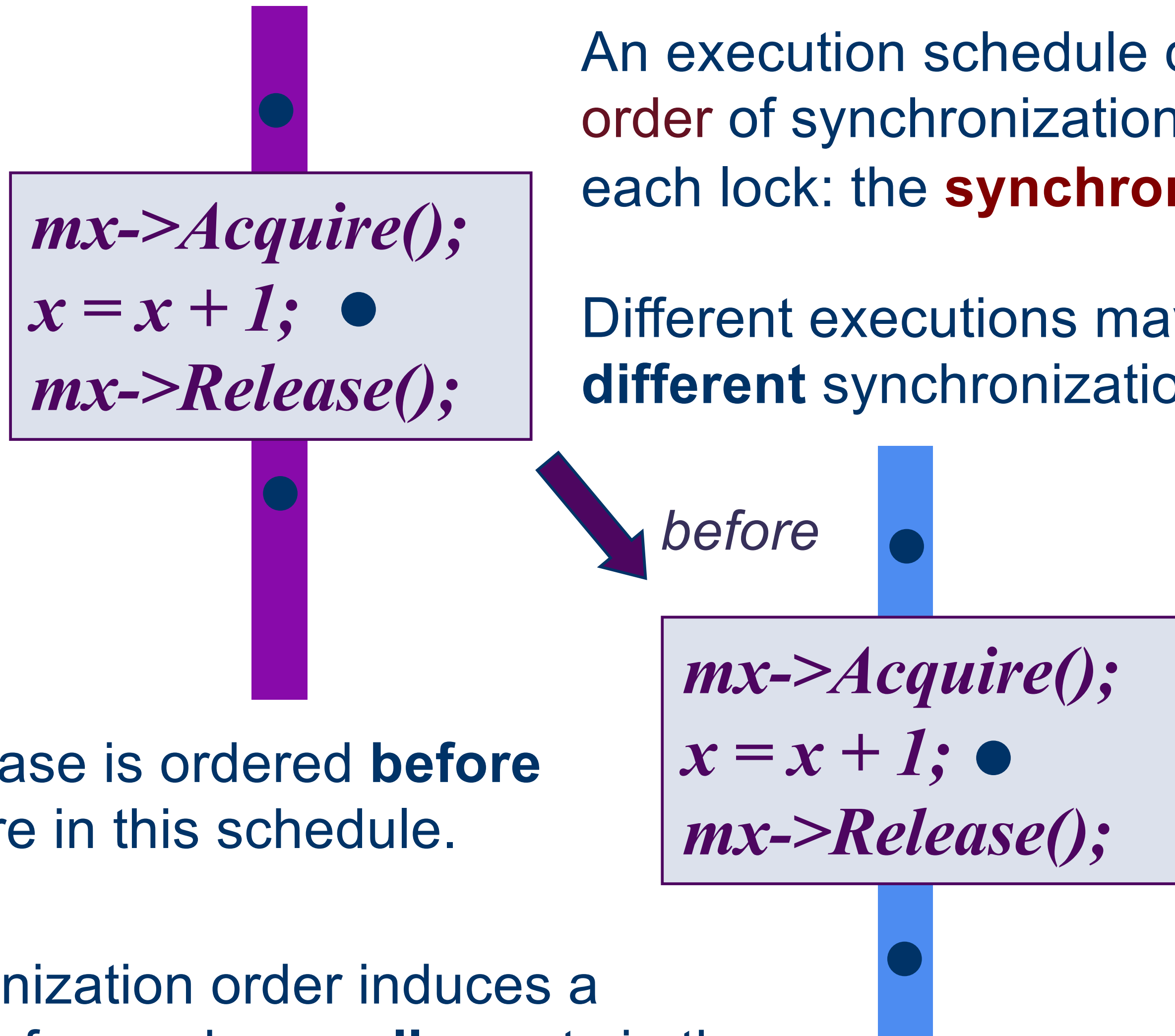


Time, Clocks, and the Ordering of Events in Distributed systems, by Leslie Lamport, CACM 21(7), July 1978

Synchronization order

An execution schedule defines a total order of synchronization events on each lock: the **synchronization order**.

Different executions may have **different** synchronization orders.



Purple's release is ordered **before** Blue's acquire in this schedule.

The synchronization order induces a happened-before order on **all** events in the execution—like message send/receive.

Racy programs: a definition

A program *P*'s *Acquire* events impose a partial order on memory accesses for each execution of *P*.

- Memory access event x_1 *happens-before* x_2 iff the synchronization orders x_1 before x_2 in that execution.
- If neither x_1 nor x_2 *happens-before* the other in that execution, then x_1 and x_2 are *concurrent*.

P has a *race* iff there exists some execution of *P* containing accesses x_1 and x_2 such that:

- Accesses x_1 and x_2 are *conflicting*.
- Accesses x_1 and x_2 are *concurrent*.

Race-free programs

- Consider a program *P*.
- *P* is **data-race-free** if no execution of *P* has a race.
- *P* is said to be **fully/correctly synchronized**.
- *P* does not exclude concurrency/parallelism!
- But *P* constrains the set of allowable schedules so that no schedule has a race.
- We use locking to build race-free programs. But it works only if we use locks **correctly**.
- **How can we tell if *P* has a race?**

Building a data race detector

Challenge: how to build a tool that tells us whether or not any *P* follows a consistent locking discipline?

If we had one, we could **save time and aggravation**.

- Option 1: *static* analysis of the source code?
- Option 2: execute the program and see if it works?
- Option 3: *dynamic* observation of the running program to see what happens and what could have happened?

How good an answer can we get from these approaches?

Dynamic data race detection

Option 1. Use *happens-before*.

– Instrument program to observe all accesses.

– Maintain *happens-before* relation on all accesses.

– Concurrent conflicting accesses?

→ Race! Howl!

– Performance? Accuracy? Generality?

Option 2. Check that locking discipline "looks right".

– Eraser's **lockset algorithm** is the canonical reference for Option 2.

– [SOSP 1997, 1800 cites]

Basic Lockset Algorithm

1. Premise: each shared *v* is covered by at least one lock.
2. Which ones? Refine "candidate" lockset for each *v*.
3. If *P* executes a set of accesses to *v*, and no lock is common to all of them, then (1) is false.

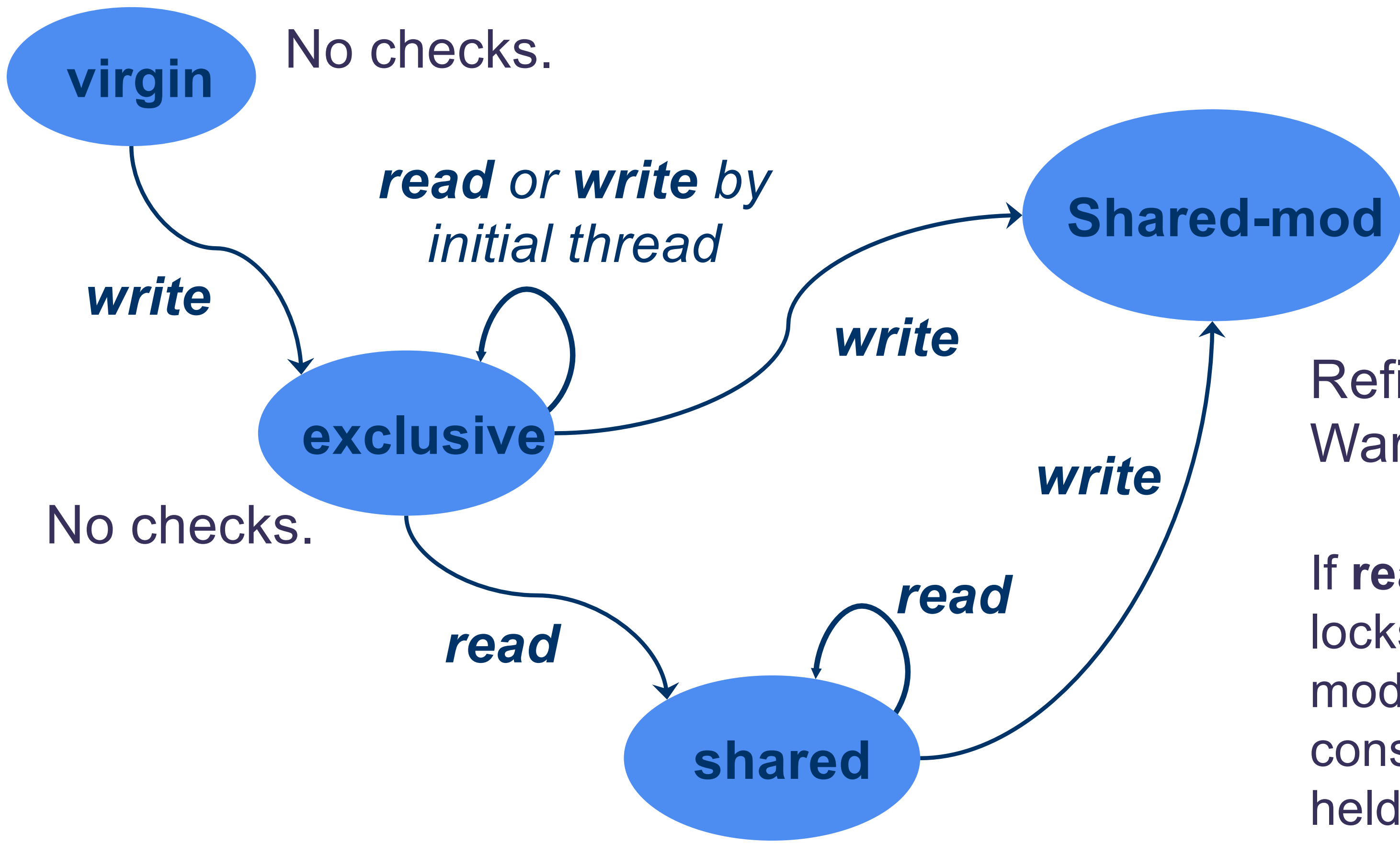
For each variable ***v, C(v) = {all locks}***
When thread ***t*** accesses ***v:***
  C(v) = C(v) $\cap$ ***locks_held(t);***
  if ***C(v) == { }*** then ***howl();***

Complications to Lockset

- “Fast” initialization of *v* before exposed to concurrency.
- WORM data: all accesses after first write are reads.
- Higher-level synchronization, e.g., *SharedLock*
  - *SharedLock* excludes conflicting accesses without holding its “little mutex” or any mutex.
- Heaps!
  - free+malloc may “change the locks”

  → Must instrument heap manager!
  - What about fast recycling above the heap manager?

Modified Lockset Algorithm



Refine $C(v)$ *with* $\cap$
Warn if $C(v) == \{\ \}$.

If **read**, consider only locks held in **read** mode. If **write**, consider only locks held in **write** mode.

Update $C(v)$, but no warnings.

The Eraser paper

What makes this a good “systems” paper?

What is interesting about the Experience?

What Validation was required to “sell” the idea?

How does the experience help to show the limitations (and possible future extensions) of the idea?

Why is the choice of applications important?

What are the “real” contributions relative to previous work?

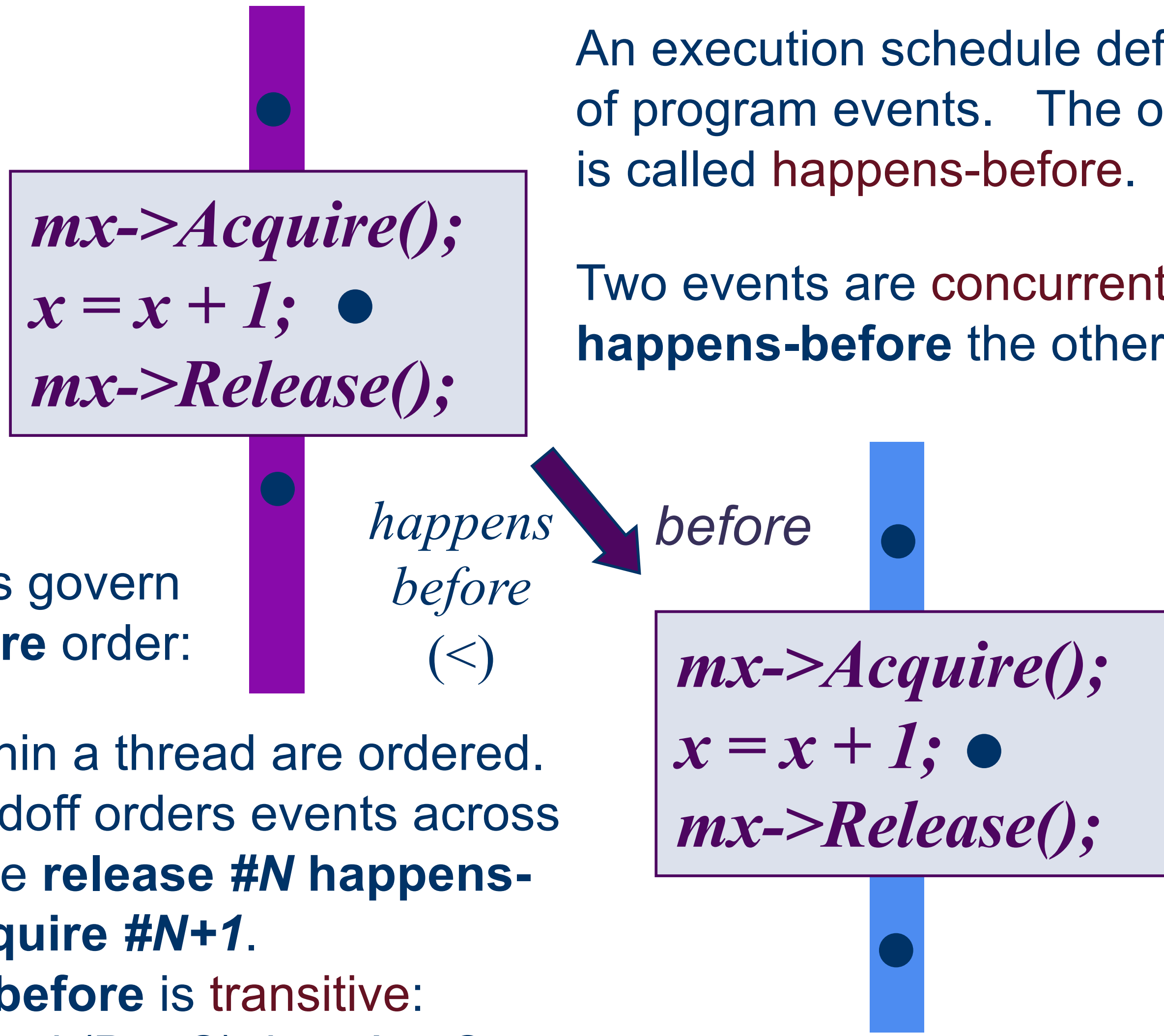
Extra slides

- The recorded lecture does not include these slides.
- The first four offer a summary and thought questions.
- The **memory model** slides preview/complement the material presented in the next lecture in the sequence.
- **Key point**: modern machines run race-free programs correctly (sequentially consistent), else their memory ordering behavior is subtle and often unexpected.

Happens-before revisited

An execution schedule defines a partial order of program events. The ordering relation (<) is called happens-before.

Two events are concurrent if neither **happens-before** the other in the schedule.



Just three rules govern **happens-before** order:

1. Events within a thread are ordered.
2. Mutex handoff orders events across threads: the **release *#N* happens-before acquire *#N+1***.
3. **Happens-before** is transitive:
   if (A < B) and (B < C) then A < C.

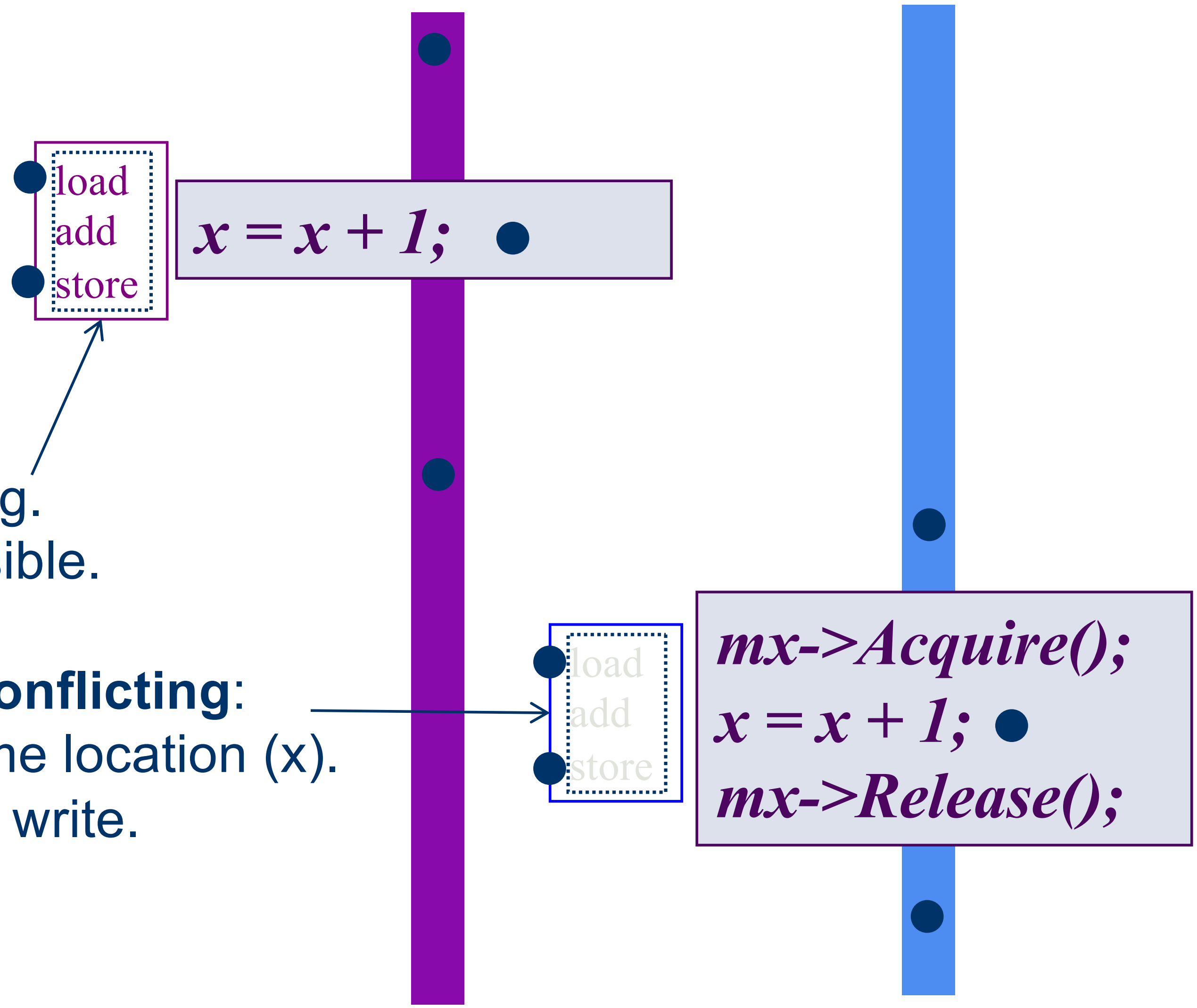
Machines may reorder concurrent events, but they always respect **happens-before** ordering.

Locks and ordering: questions

1. What ordering does *happened-before* define for acquires on a given mutex?
2. What ordering does *happened-before* define for acquires on different mutexes?

   Can a data item be safely protected by two locks?
3. When *happened-before* orders x_1 before x_2, does every execution of P preserve that ordering?
4. What can we say about the *happened-before* relation for a single-threaded execution?

Another race



No lock! → No ordering.
Any interleaving is possible.

These operations are **conflicting**:

- They access the same location (x).
- And at least one is a write.

Questions on “Another race”

1. What if the two access pairs were to different variables *x* and *y*?
2. What if the access pairs were protected by different locks?
3. What if the accesses were all reads?
4. What if only one thread modifies the shared variable?
5. What about “variables” consisting of groups of locations?
6. What about “variables” that are fields within locations?
7. What’s a *location*?
8. Is every race an error?

Extra slides

MEMORY MODELS

Needed: a memory model

Challenge: specify memory behavior for a threaded PL.

- **Portable** across multi-core machines.
- Admits full range of optimizations for concurrency.
- **Runs correct (safe) programs correctly.**
- Conforms to Principle of No Unreasonable Surprises for incorrect programs. "No wild shared memory."

  →Easy for programmers to reason about.

Memory model in three steps

1. Define what it means for a program to be **safe.** Also called **data-race-free**. ☑
2. Define memory behavior observed by safe programs.
3. Define the memory model for unsafe programs.

Maybe Step 3 is the hardest and most controversial. We need to understand it later for **lock-free** synchronization.

The Java Memory Model*

Jeremy Manson and William Pugh
Department of Computer Science
University of Maryland, College Park
College Park, MD
{jmanson, pugh}@cs.umd.edu

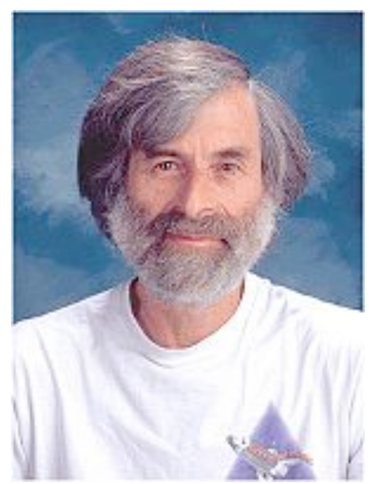
Sarita V. Adve
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana-Champaign, IL
sadve@cs.uiuc.edu

- "This paper describes the new JMM, which…specifies the legal behaviors for a multithreaded program; it defines the semantics of multithreaded Java programs and…legal implementations of JVMs and compilers." [POPL 2005]
- "Happens-before (→) is the transitive closure of program order and synchronization order."
- "A program is said to be **correctly synchronized** or **data-race-free** iff all sequentially consistent executions of the program are free of data races." [Under →]

How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs

LESLIE LAMPORT

Abstract—**Many large sequential computers execute operations in a different order than is specified by the program. A correct execution is achieved if the results produced are the same as would be produced by executing the program steps in order. For a multiprocessor computer, such a correct execution by each processor does not guarantee the correct execution of the entire program. Additional conditions are given which do guarantee that a computer correctly executes multiprocess programs.**

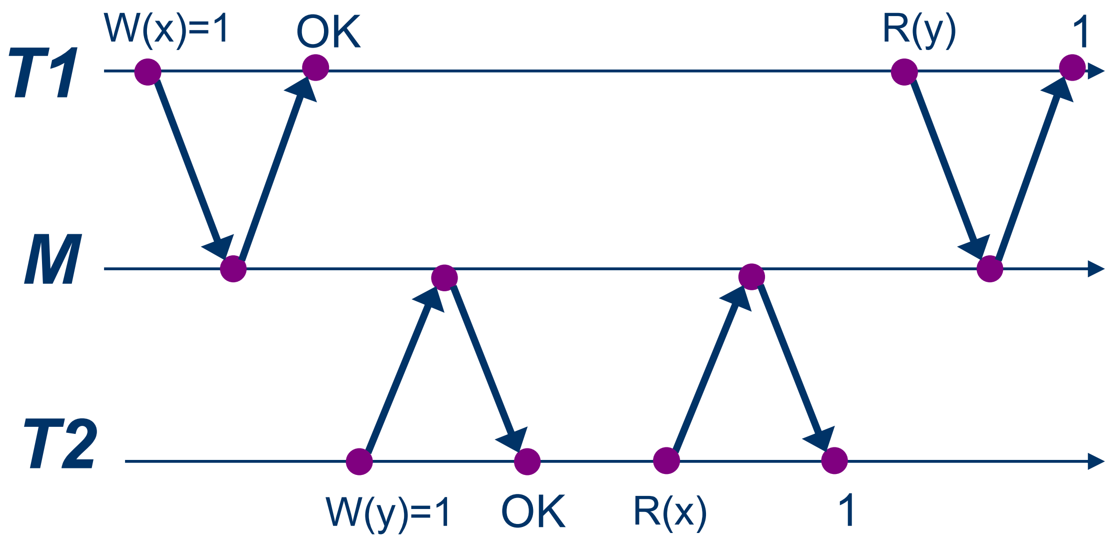


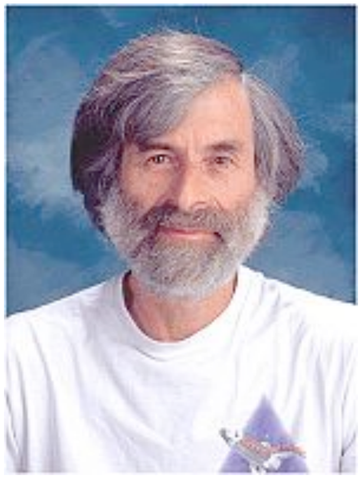
1979: An early understanding of multicore memory model. Proposed **sequential consistency** for machines.

Sequential consistency

A machine is **sequentially consistent** iff:

- Memory operations (loads and stores) appear to execute in some sequential order on the memory;
- Ops from the same core appear to execute in program order;
- The order is identical to all observers.





Sequential consistency is too strong!

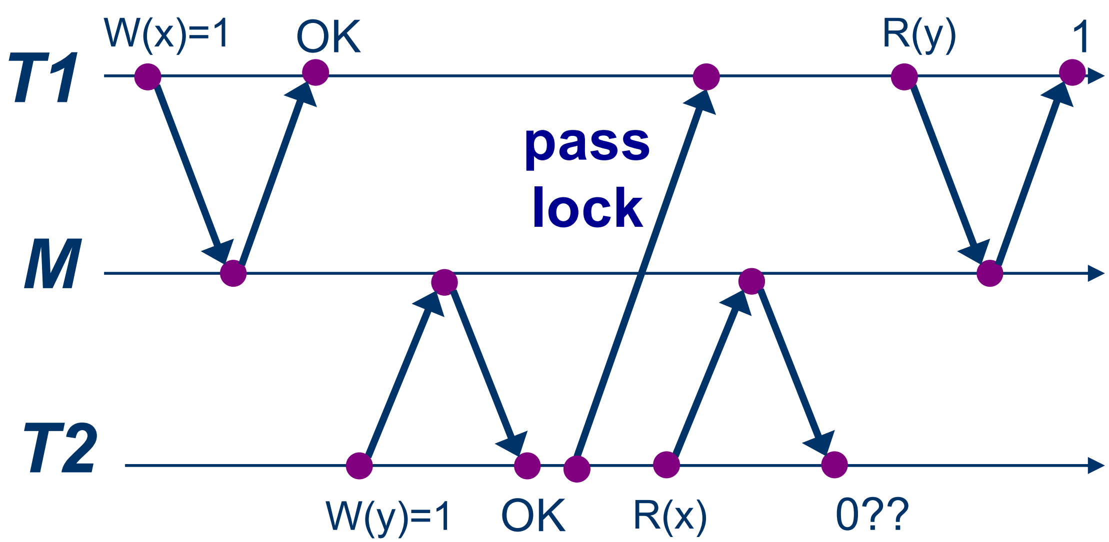
- Sequential consistency requires the machine to do a lot of extra work that might be unnecessary.
- The machine must make memory updates by one core visible to others, even if the program doesn't care.
- The machine must do some of the work even if no other core ever references the updated location!
- Can a multiprocessor with a weaker ordering than sequential consistency still execute programs correctly?
- Answer: yes. Modern multicore systems allow orderings that are weaker, but still respect the happens-before order induced by synchronization (lock/unlock).

Memory behavior in the real world

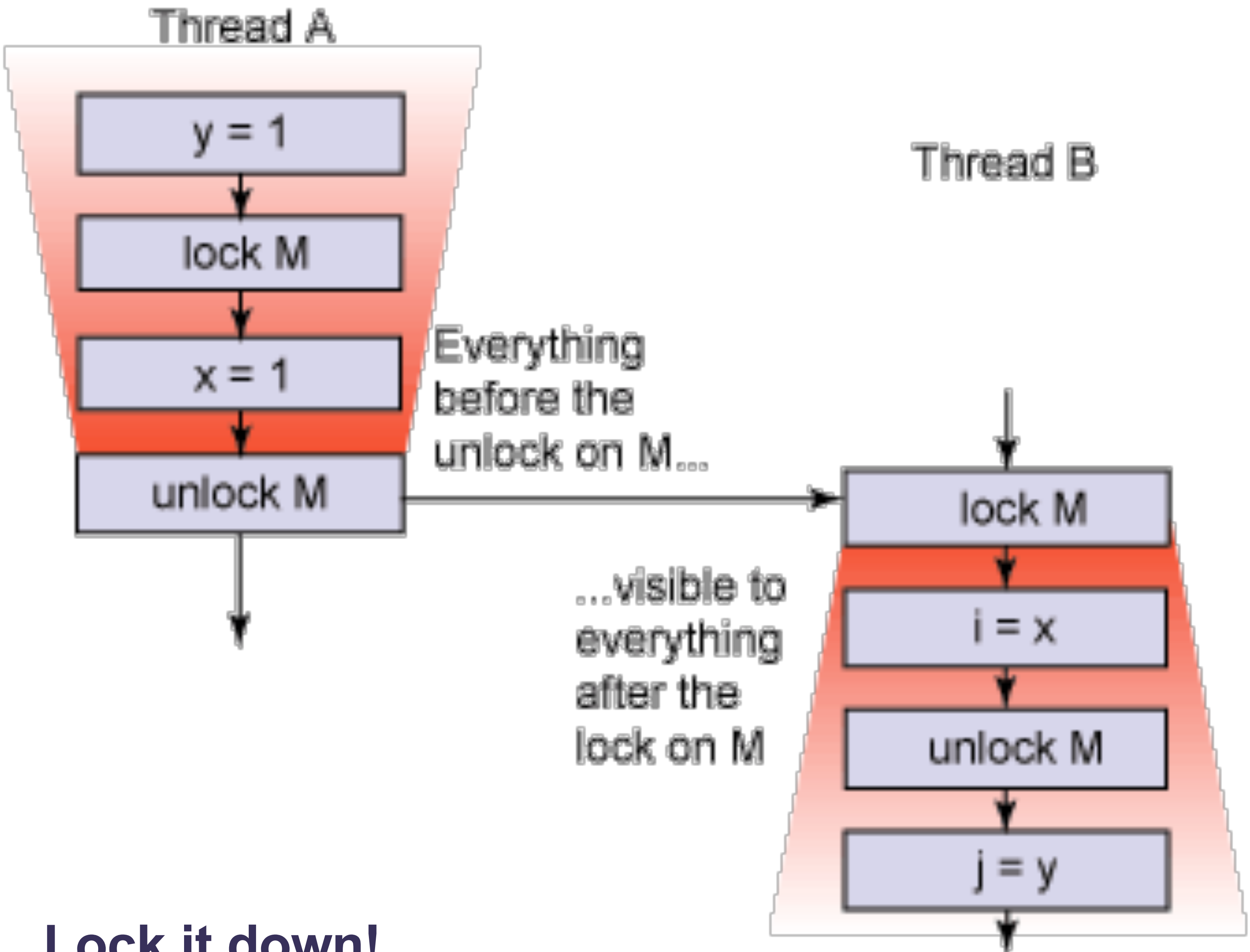
- Synchronization accesses tell the machine that ordering matters: a happens-before relationship exists.
- Machines **always** respect happens-before.

→ sequential consistency for **race-free** programs.

- Otherwise, **all bets are off**. Synchronize!



The most you should assume is that any memory **store** before a lock release is visible to a **load** on a core that has subsequently acquired the same lock.

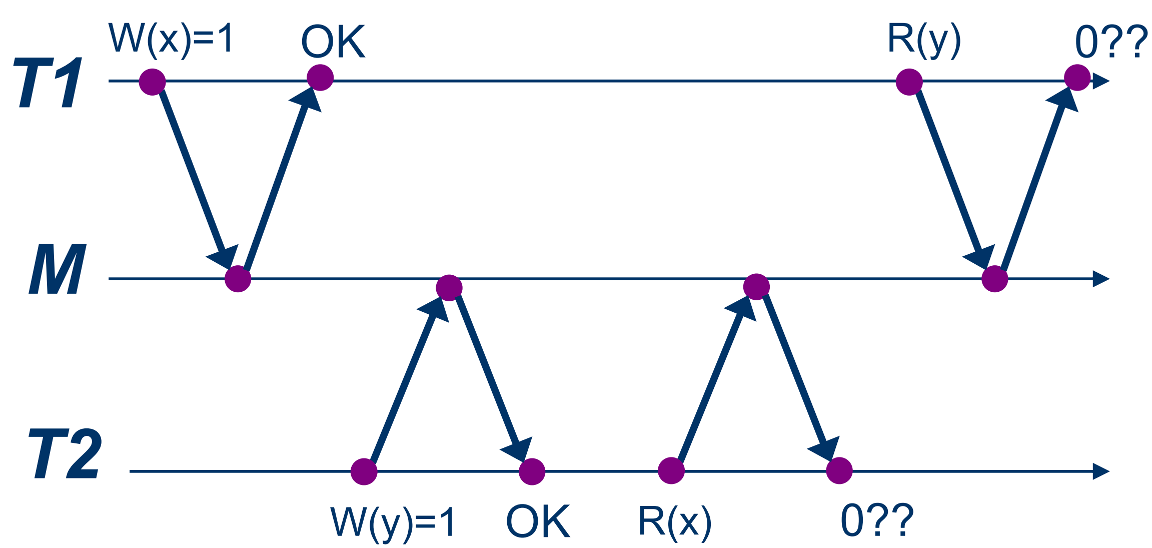


Thread A
y = 1
lock M
x = 1
unlock M
Thread B
Everything
before the
unlock on M...
lock M
...visible to
everything
after the
lock on M
i = x
unlock M
j = y

Lock it down!

Don’t assume sequential consistency

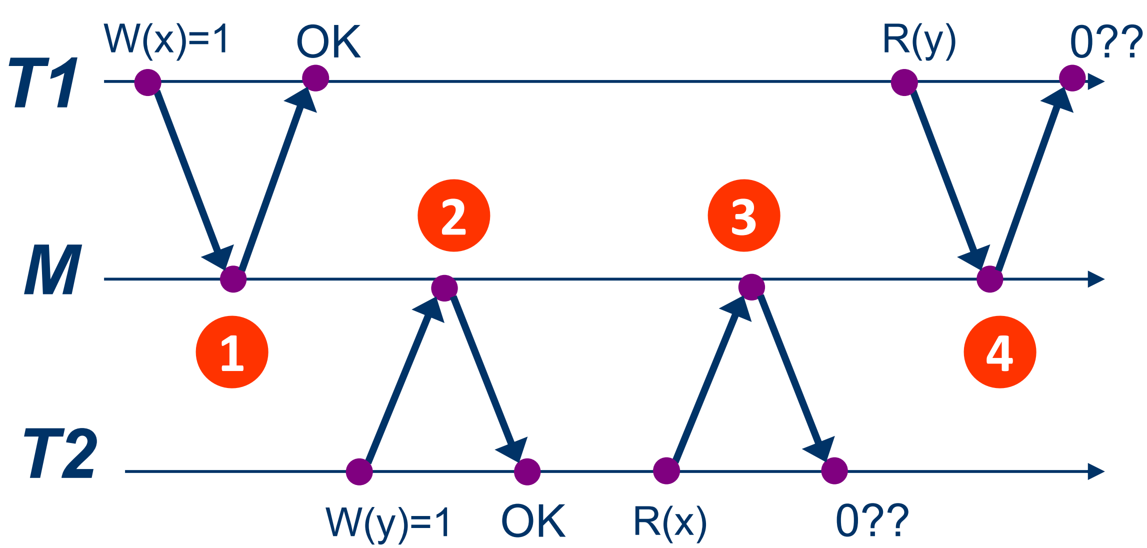
- A **load** might fetch from the local cache and not from memory.
- A **store** may buffer a value in a local cache before draining the value to memory, where other cores can access it.
- Therefore, a **load** from one core does not necessarily return the “latest” value written by a **store** from another core.



A trick called Dekker’s algorithm supports mutual exclusion on multi-core without using atomic instructions. It assumes that **load** and **store** ops execute sequentially.
But they don’t.

Don’t assume sequential consistency

No sequentially consistent execution can produce the result below, yet it can occur on modern machines.



To produce this result: 4<2 (4 **happens-before** 2) and 3<1. No such schedule can exist unless it also reorders the accesses from T1 or T2. Then the reordered accesses are out of program order.

JMM model

The “simple” JMM happens-before model:

- A read cannot see a write that *happens after* it.
- If a read sees a write (to an item) that happens before the read, then the write must be the last write (to that item) that happens before the read.

Augment for sane behavior for unsafe programs (loose):

- Don’t allow an *early* write that “depends on a read returning a value from a data race”.
- An *uncommitted* read must return the value of a write that happens-before the read.