Race Detection

cs378

Pro Forma

- Questions?
- Administrivia:
  - Course/Instructor Survey : https://utdirect.utexas.edu/ctl/ecis/
  - Next class: review – send questions!
  - Thoughts on exam
  - Thoughts on project presentation day
- Agenda
  - Linearizability clarification
  - Race Detection
- Acknowledgements:
  - https://ecksit.wordpress.com/2015/09/07/difference-between-sequential-consistency-serializability-and-linearizability/
  - https://www.cl.cam.ac.uk/teaching/1718/R204/slides-tharris-2-lock-free.pptx
  - http://concurrencyfreaks.blogspot.com/2013/05/lock-free-and-wait-free-definition-and.html
  - http://swtv.kaist.ac.kr/courses/cs492b-spring-16/lec6-data-race-bug.pptx
  - https://www.cs.cmu.edu/~clegoues/docs/static-analysis.pptx
  - http://www.cs.sfu.ca/~fedorova/Teaching/CMPT401/Summer2008/Lectures/Lecture8-GlobalClocks.pptx

Race Detection Faux Quiz

- Are linearizable objects composable? Why/why not? Is serializable code composable?
- What is a data race? What kinds of conditions make them difficult to detect automatically?
- What is a consistent cut in a distributed causality interaction graph?
- List some tradeoffs between static and dynamic race detection
- What are some pros and cons of happens-before analysis for race detection? Same for lockset analysis?
- Why might one use a vector clock instead of a logical clock?
- What are some advantages and disadvantages of combined lock-set and happens-before analysis?

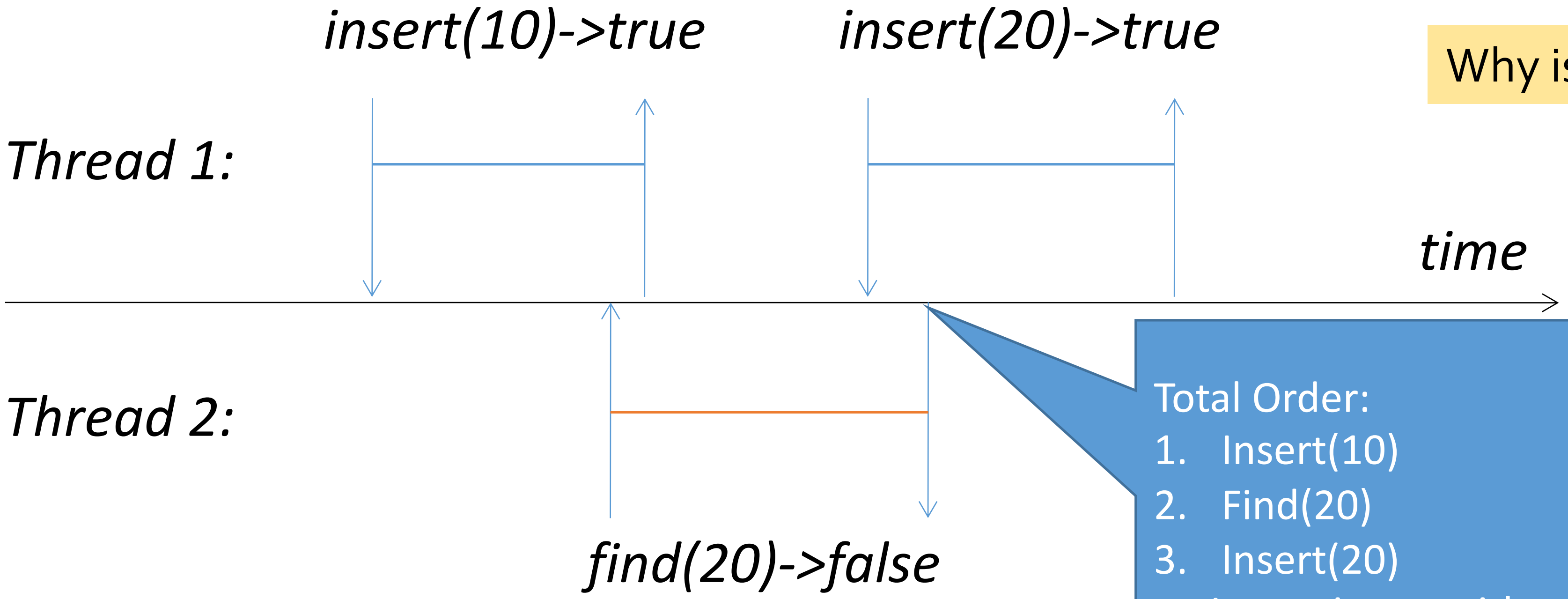
Review: Concurrent history

Linearizability:

- Is there a correct sequential history:
  - Same results as the concurrent one
  - Consistent with the timing of the invocations/responses?
  - Start/end impose ordering constraints

Allow *overlapping* invocations

Why is this one OK?

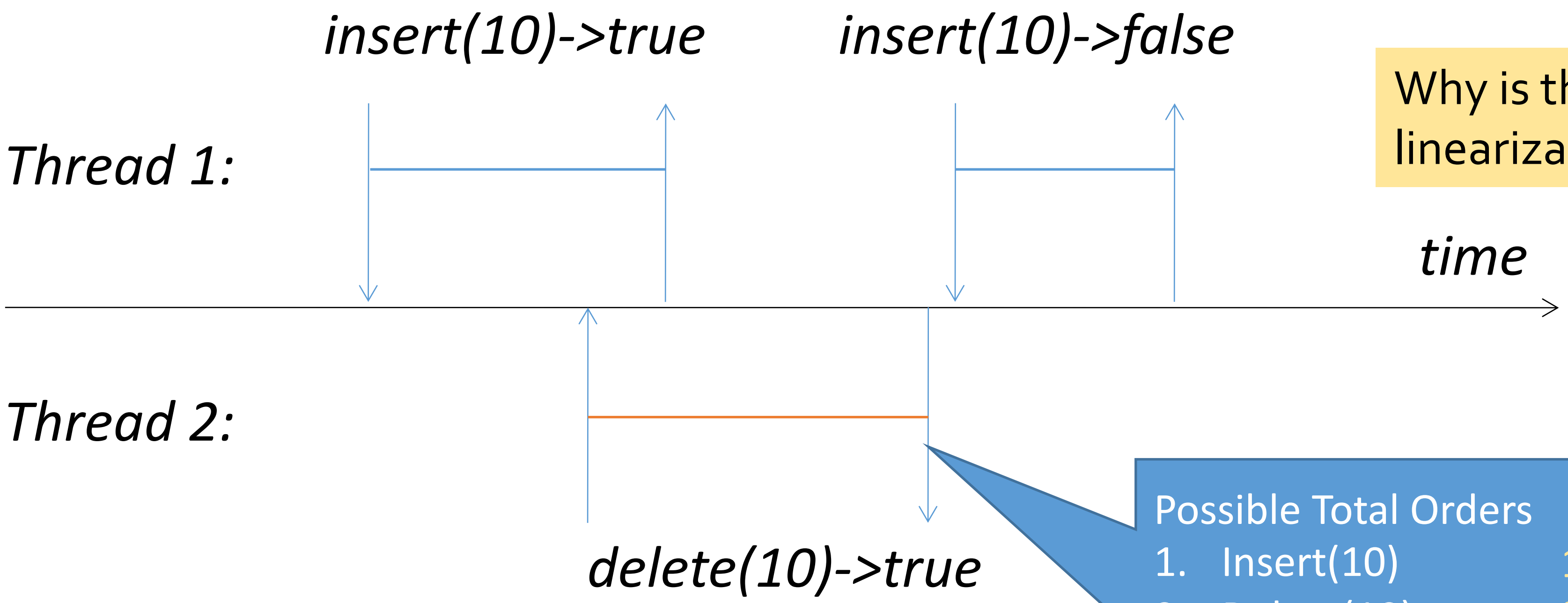


Total Order:

1. Insert(10)
2. Find(20)
3. Insert(20)

- Is consistent with real-time order
- 2, 3 overlap, but return order OK

Review: not linearizable



Why is this one NOT linearizable?

Possible Total Orders

1. Insert(10)
2. Delete(10)
3. Insert(10)

1. Delete(10)
2. Insert(10)
3. Insert(10)

- Both consistent with real-time order
- Neither is consistent w return values
- 1, 2 overlap, but 3 doesn't

Assumptions:

- The set is initially empty
- Return values are meaningful:
  - Insert returns true → *item wasn't present*
  - Insert returns false → *item already present*
  - Delete returns true → *item was present*

Linearizability Properties

- non-blocking
  - one method is never forced to wait to sync with another.
- local property:
  - a system is linearizable iff each individual object is linearizable.
  - gives us **composability**.
- Why is it important?
  - Serializability is not composable.

Composability

```
T * list::remove(Obj key){
  LOCK(this);
  tmp = __do_remove(key);
  UNLOCK(this);
  return tmp;
}

void list::insert(Obj key, T * val){
  LOCK(this);
  __do_insert(key, val);
  UNLOCK(this);
}
```

```
void move(list s, list d, Obj key){
  tmp = s.remove(key);
  d.insert(key, tmp);
}
void move(list s, list d, Obj key){
  LOCK(s);
  LOCK(d);
  tmp = s.remove(key);
  d.insert(key, tmp);
  UNLOCK(d);
  UNLOCK(s);
}
```

Painting with a very broad brush
Composition with linearizability is really about composed schedules

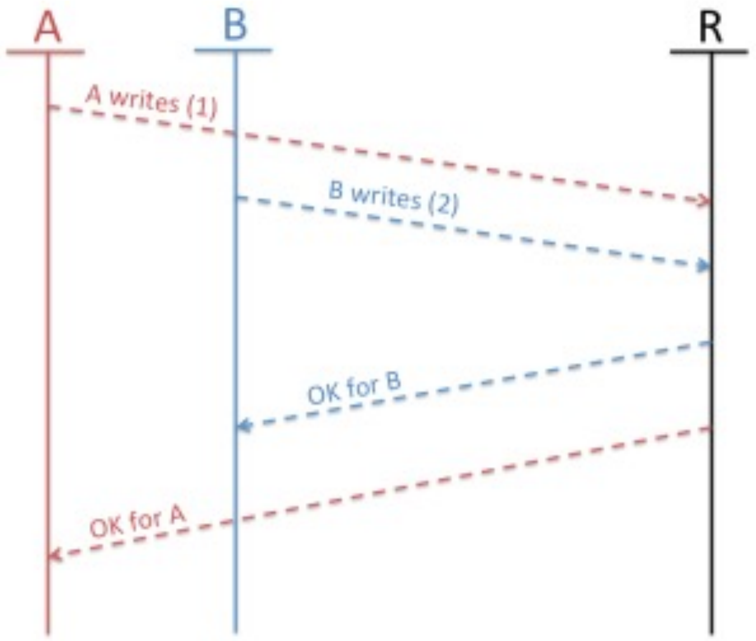
- Lock-based code doesn't compose
- If list were a linearizable concurrent data structure, composition OK?

More on Composability and Compositionality

- High level /informal meaning:
  - Can you compose codes that provide property P
  - …and expect the composition to preserve P?
- More nuanced meanings:
  - Can you compose codes
  - Can you compose schedules
- These are related but differ in subtle ways
- Non-composability of serializability is really about composing schedules

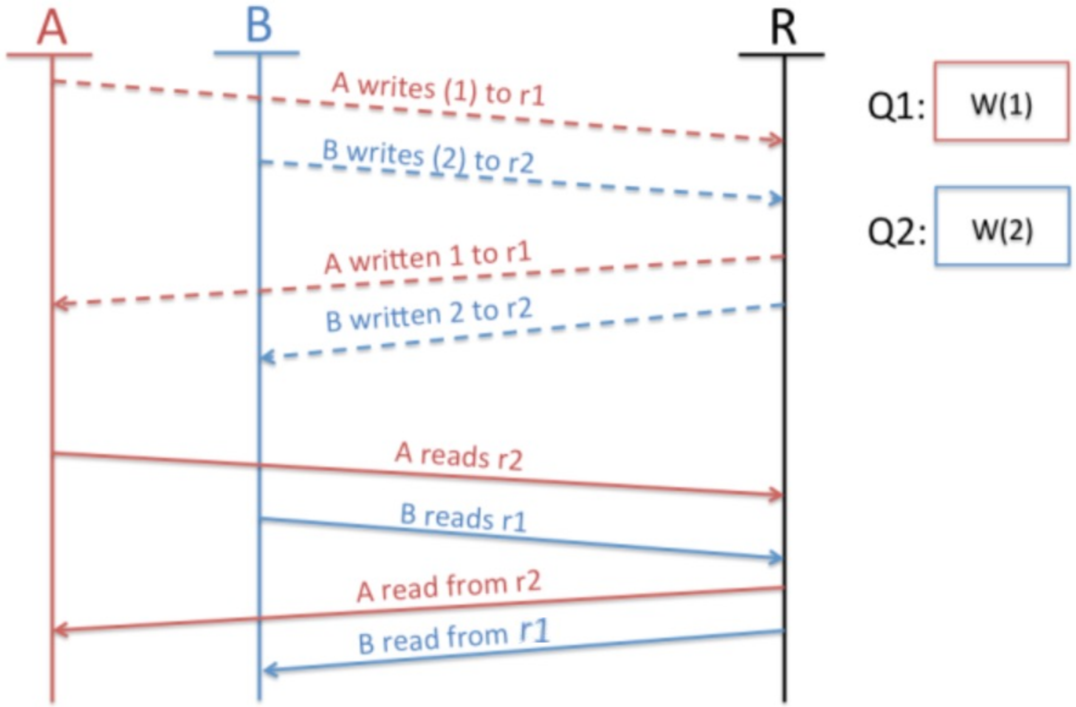
Consider A Concurrent Register

- Threads A, B write integers to a register R
- Because it's concurrent, method invocations overlap



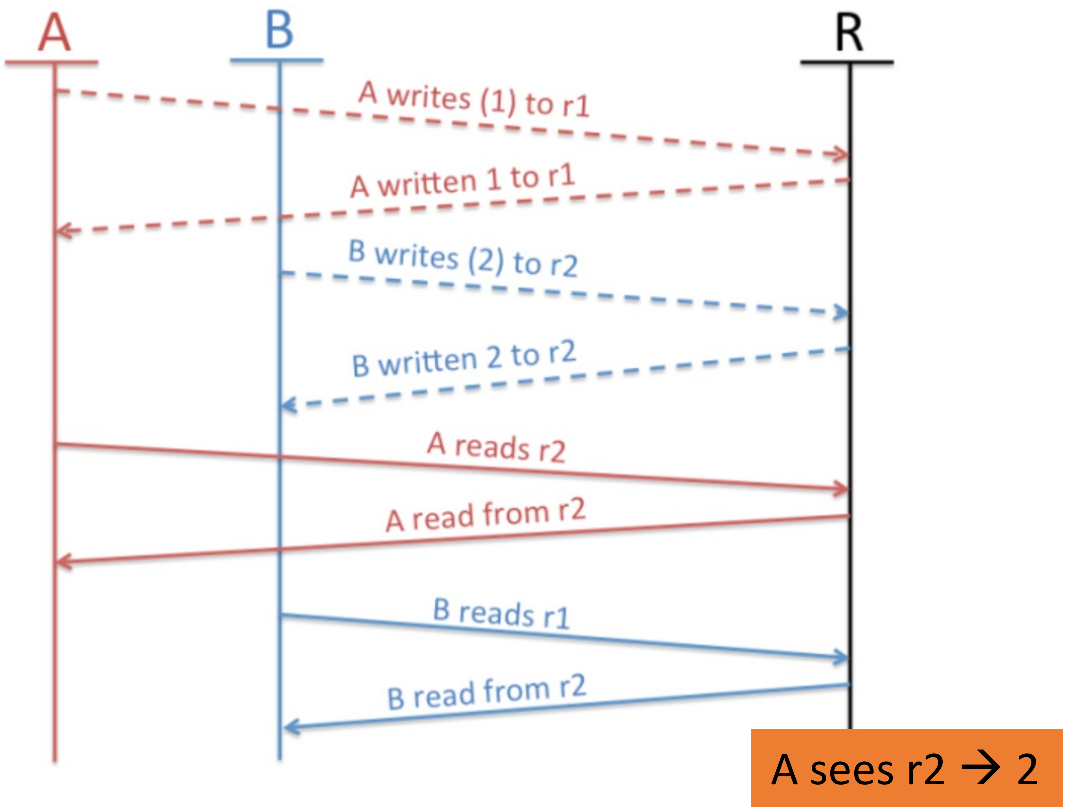
Two Concurrent Registers

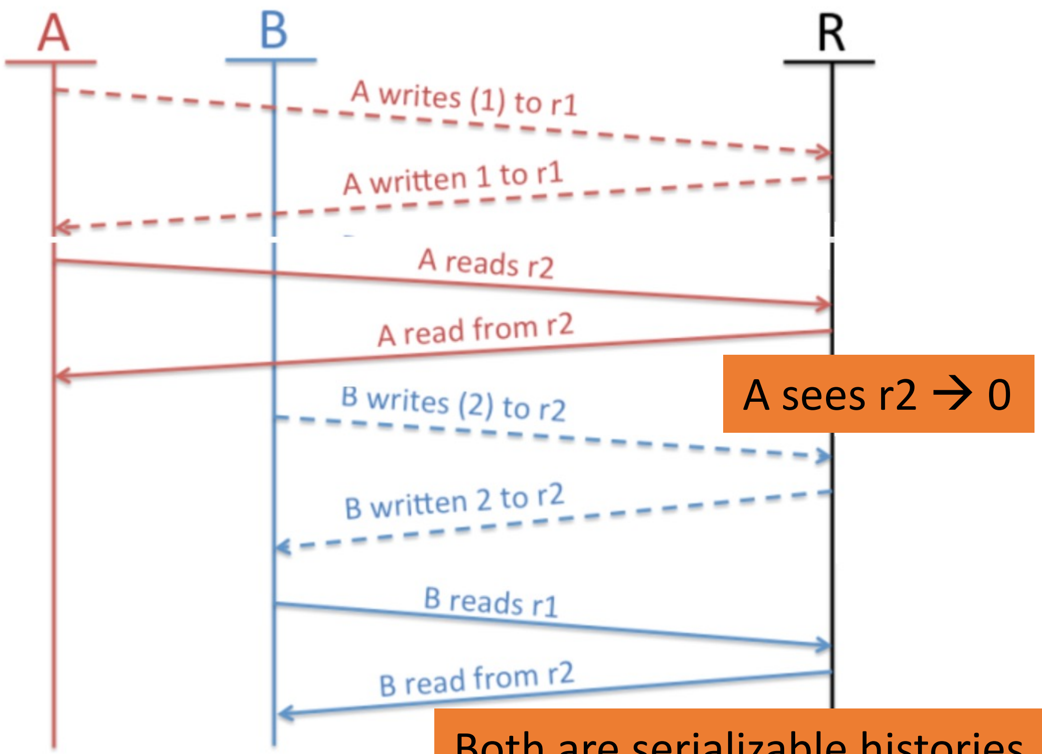
- Register value is initially zero
- The following operations occur:
  - Thread A:
    - write r1 = 1
    - read r2 → ?
  - Thread B:
    - B: write r2 -> 2
    - B: read r1 → ?
- Serializability:
  - Execution equivalent to *some serial order*
  - All see same order



Histories for multiple concurrent registers

- Consider all possible permutations of atomic invocations
  - (That respect program order)

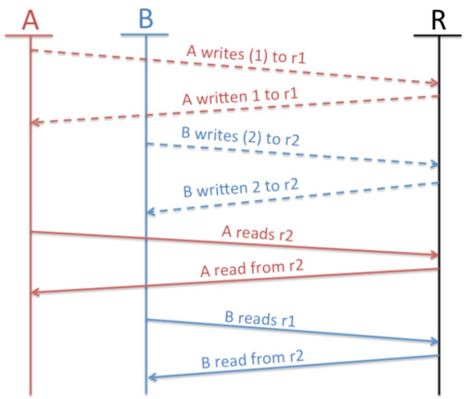




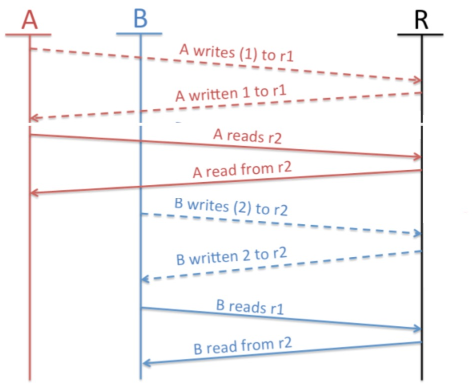
Both are serializable histories from the perspective of A

Histories for multiple concurrent registers

- Consider all possible permutations of atomic invocations
  - (That respect program order)
  - Call them "sub-histories": from A, B "perspective"



Sub-History	Outcome
H1a	A writes r1=1, reads r2 → 0
H2a	A writes r1=1, reads r2 → 2
H1b	B writes r2=2, reads r1 → 0
H2b	B writes r2=2, reads r1 → 1



…

From the perspective threads A, B, all sub-histories are serializable

- They respect program order for each of A, B
- And are equivalent to *some* serial execution
- If we "compose" these histories, some composed histories not serializable

Histories for multiple concurrent registers

- Compose sub-histories to form all possible histories
- Composition of serializable histories → non-serializable histories
- Ex. H1ab is not serializable

Sub-History	Outcome
H1a	A writes r1=1, reads r2 → 0
H2a	A writes r1=1, reads r2 → 2
H1b	B writes r2=2, reads r1 → 0
H2b	B writes r2=2, reads r1 → 1

History	Effect
H1ab	A writes r1=1, B writes r2=2 reads r2 → 0, B reads r1 → 0
H2ab	A writes r1=1, B writes r2=2 reads r2 → 0, B reads r1 → 1
H3ab	A writes r1=1, B writes r2=2 reads r2 → 2, B reads r1 → 0
H4ab	A writes r1=1, B writes r2=2 reads r2 → 2, B reads r1 → 1

4 serializable sub-histories composed
To form 4 complete histories,
Only H4ab is actually serializable

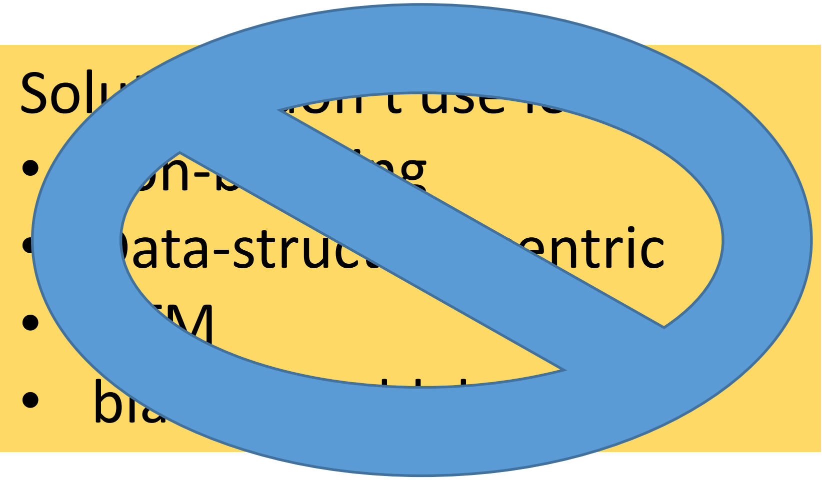
Linearizability Properties

- non-blocking
  - one method is never forced to wait to sync with another.
- local property:
  - a system is linearizable iff each individual object is linearizable.
  - gives us **composability**.
- Why is it important?
  - Serializability is not composable.
  - A system composed of linearizable objects remains linearizable
  - Does this mean you get txn or lock-like composition for free?
    - In general no
    - Serializability is a property of transactions, or groups of updates
    - Linearizability is a property of concurrent objects
    - The two are often conflated (e.g. because txns update only a single object)

Race Detection

Locks: a litany of problems

- Deadlock
- Priority inversion
- Convoys
- Fault Isolation
- Preemption Tolerance
- Performance



Use locks!

- But automate bug-finding!

Races

```
Lock(lock);                  
Read-Write(X);               Read-Write(X);
Unlock(lock);                
```

- Is there a race here?
- What is a race?
- Informally: accesses with missing/incorrect sy
- Formally:
  - >1 threads access same item
  - No intervening synchronization
  - At least one access is a write

How to detect races:

```
forall(X) {
  if(not_synchronized(X))
    declare_race()
}
```

Lockset Algorithm

- Locking discipline
  - Every shared mutable variable is protected by some locks
- Core idea
  - Track locks held by thread t

Let $locks\_held(t)$ be the set of locks held by thread t.
For each v, initialize $C(v)$ to the set of all locks.
On each access to v by thread t,
  set $C(v) := C(v) \cap locks\_held(t)$;
  if $C(v) = \{\ \}$, then issue a warning.

Narrow down set of locks maybe protecting v

  - Assume every lock protects every variable
  - On each access, use locks held by thread to narrow that assumption

Lockset Algorithm Example

thread t	locks_held(t)	C(v)
	{}	{lockA, lockB}
lock(lockA);	{lockA}	
v++;		{lockA} $C(v) \cap locks\_held(t)$
unlock(lockA);	{}	
lock(lockB);	{lockB}	
v++;		{} $C(v) \cap locks\_held(t)$ ACK! race
unlock(lockB);	{}	

Pretty clever! Why isn't this a complete solution?

Improving over lockset

thread A

```
1 read-write(X);
2 fork(thread-proc);
3 do_stuff();
4 do_more_stuff();
5 join(thread-proc);
6 read-Write(X);
```

thread B

```
1 thread-proc() {
2
3  read-write(X);
4
5 }
```

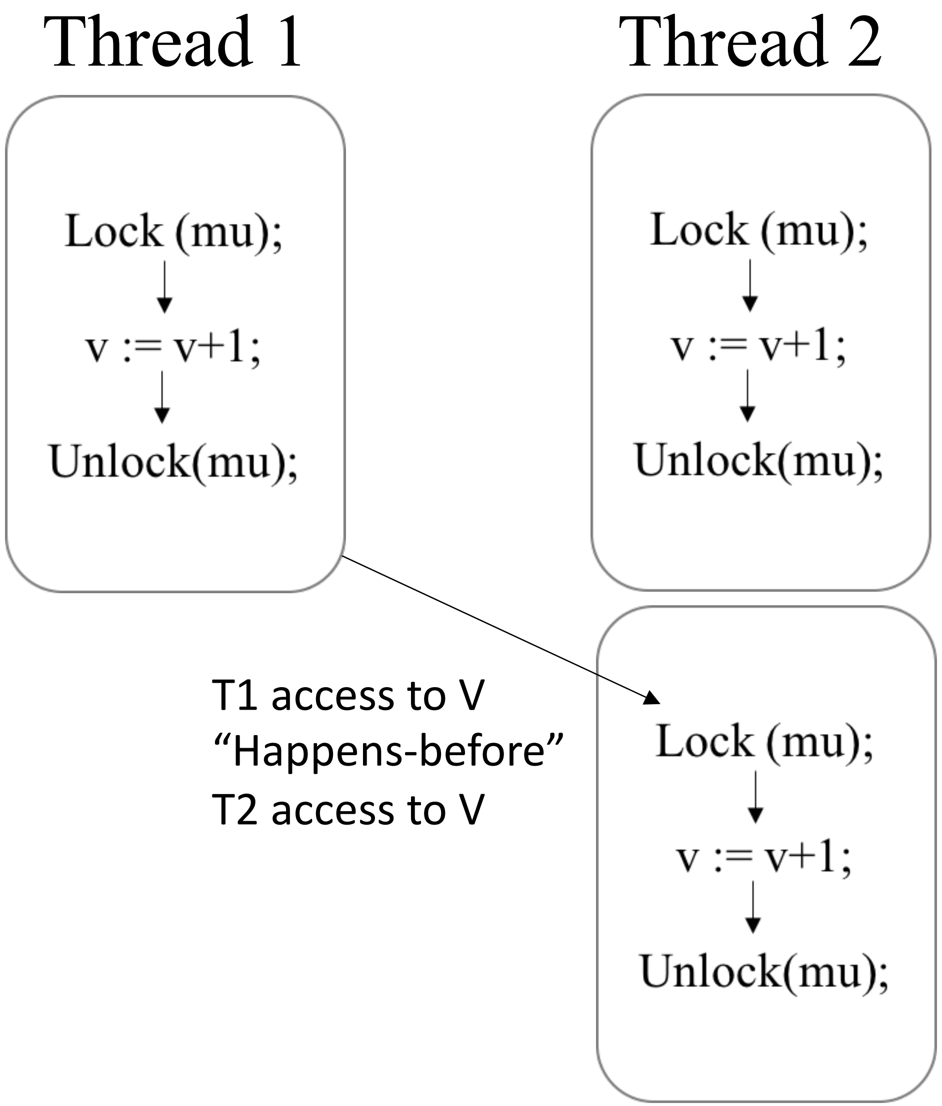
Lockset detects a race

There is no race: why not?

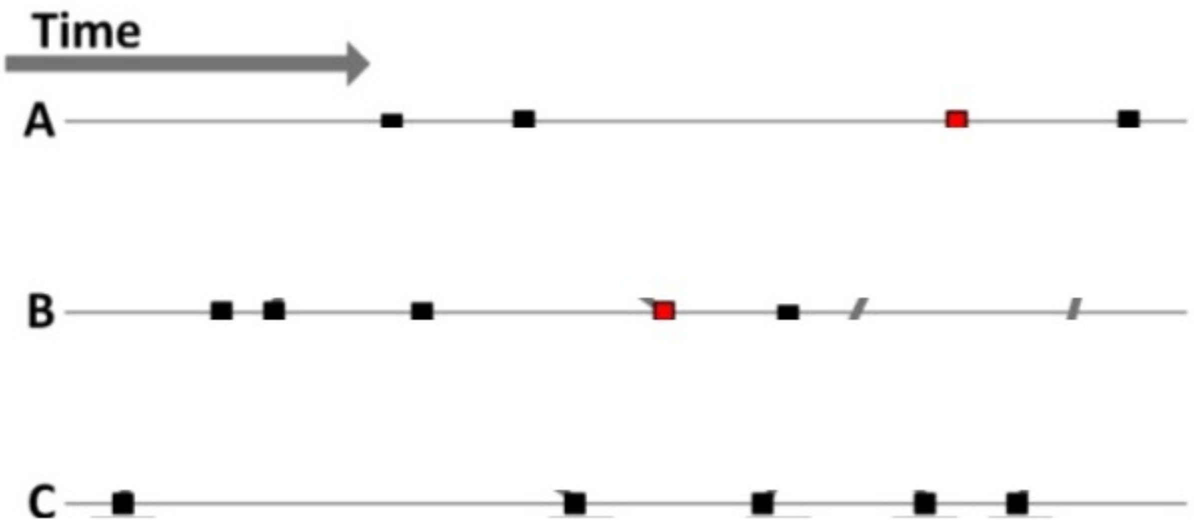
- A-1 happens before B-3
- B-3 happens before A-6
- Insight: races occur when “happens-before” cannot be known

Happens-before

- *Happens-before* relation
  - Within single thread
  - Between threads
- Accessing variables not ordered by “happens-before” is a race
- Captures locks and dynamism
- How to track “happens-before”?
  - Sync objects are ordering events
  - Generalizes to fork/join, etc



Ordering and Causality



A, B, C have local orders

- Want total order
  - But only for causality

Different types of clocks

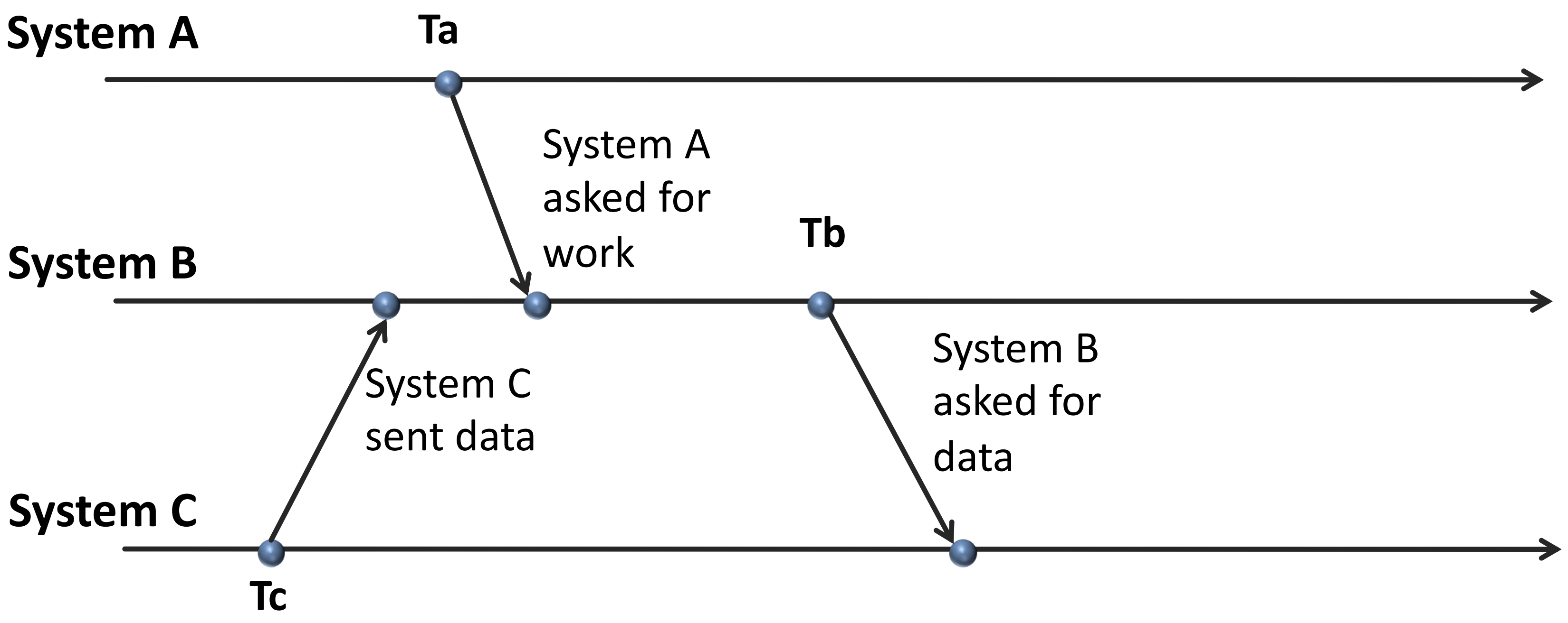
- Physical
- Logical
  - TS(A) later than others A knows about
- **Vector**
  - **TS(A): what A knows about other TS's**
- Matrix
  - TS(A) is N^2 showing pairwise knowledge

A Naïve Approach

- Each system records each event it performed and its timestamp
- Suppose events in the this system happened in this real order:
  - **Time Tc0:** System C sent data to System B (before C stopped responding)
  - **Time Ta0:** System A asked for work from System B
  - **Time Tb0:** System B asked for data from System C

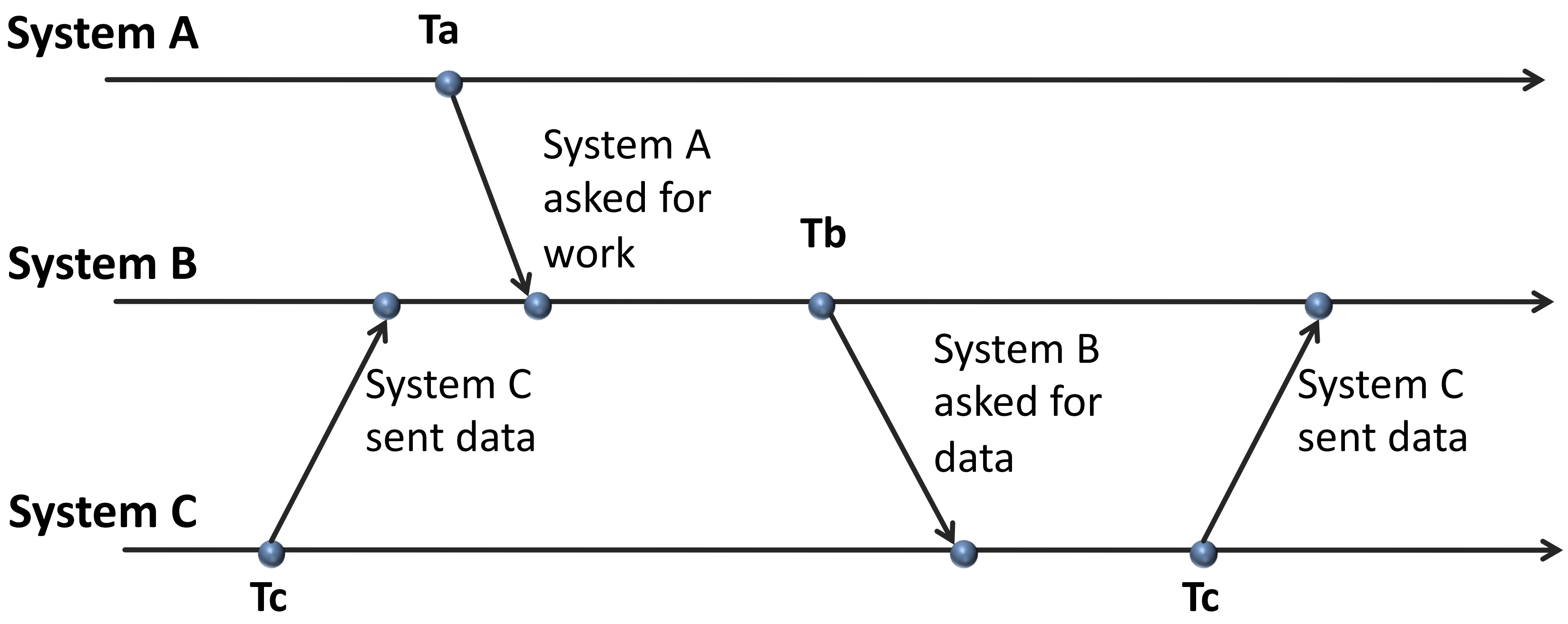
A Naïve Approach (cont)

- Ideally, we will construct real order of events from local timestamps and detect this dependency chain:



A Naïve Approach (cont)

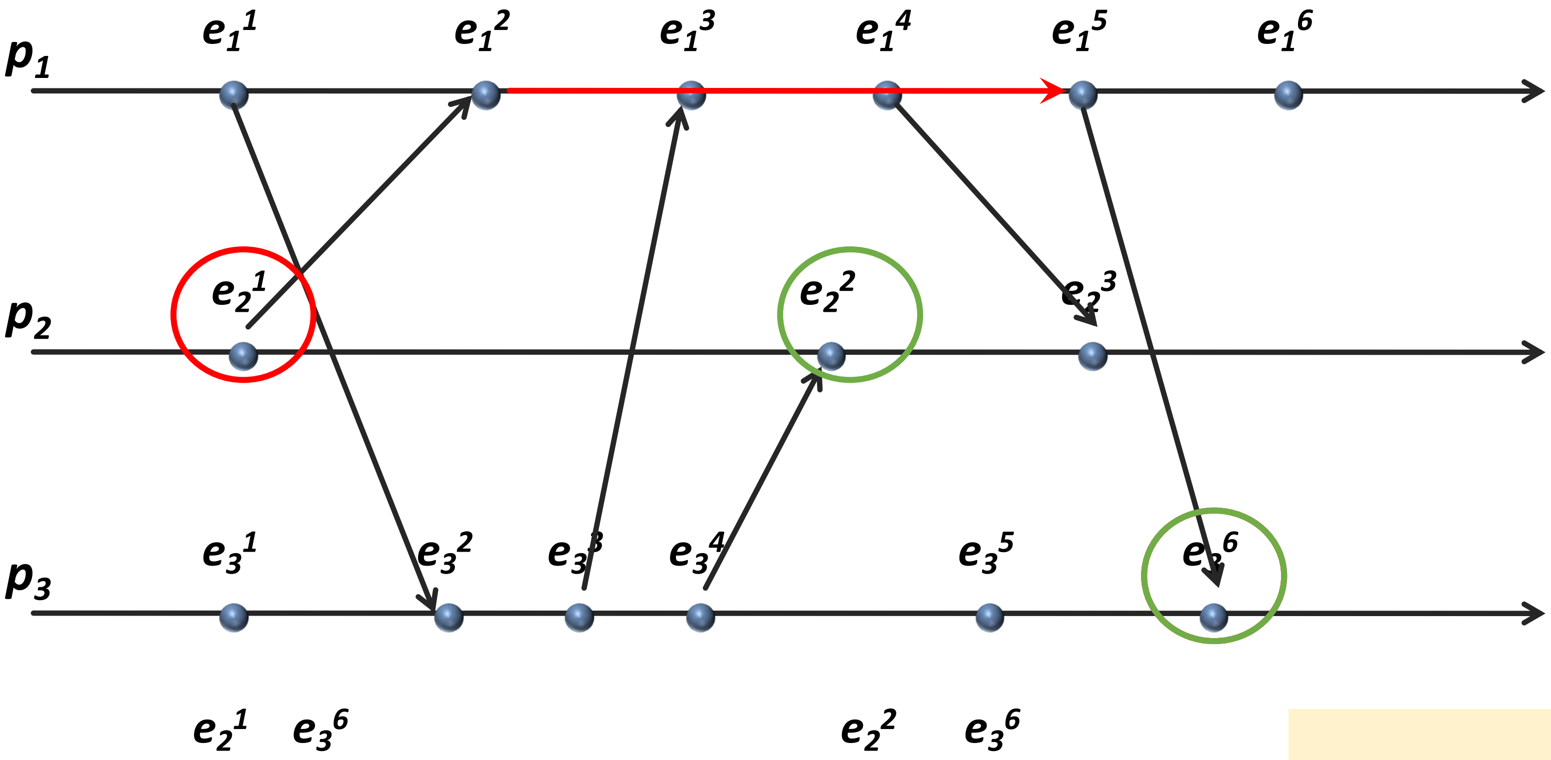
- But in reality, we do not know if Tc occurred **before** Ta and Tb, because in an asynchronous distributed system **clocks are not synchronized**!



Rules for Ordering of Events

- local events precede one another → precede one another globally:
  - If $e_i^k, e_i^m \in h_i$ and $k < m$, then $e_i^k \rightarrow e_i^m$
- Sending a message always precedes receipt of that message:
  - If $e_i = send(m)$ and $e_j = receive(m)$, then $e_i \rightarrow e_j$
- Event ordering is transitive:
  - If $e \rightarrow e'$ and $e' \rightarrow e''$, then $e \rightarrow e''$

Space-time Diagram for Distributed Computation



local events precede one another → precede one another globally:

If $e_i^k, e_i^m \in h_i$ *and* $k < m$, *then* $e_i^k \rightarrow e_i^m$

Sending a message always precedes receipt of that message:

If $e_i = send(m)$ *and* $e_j = receive(m)$, *then* $e_i \rightarrow e_j$

Event ordering is associative:

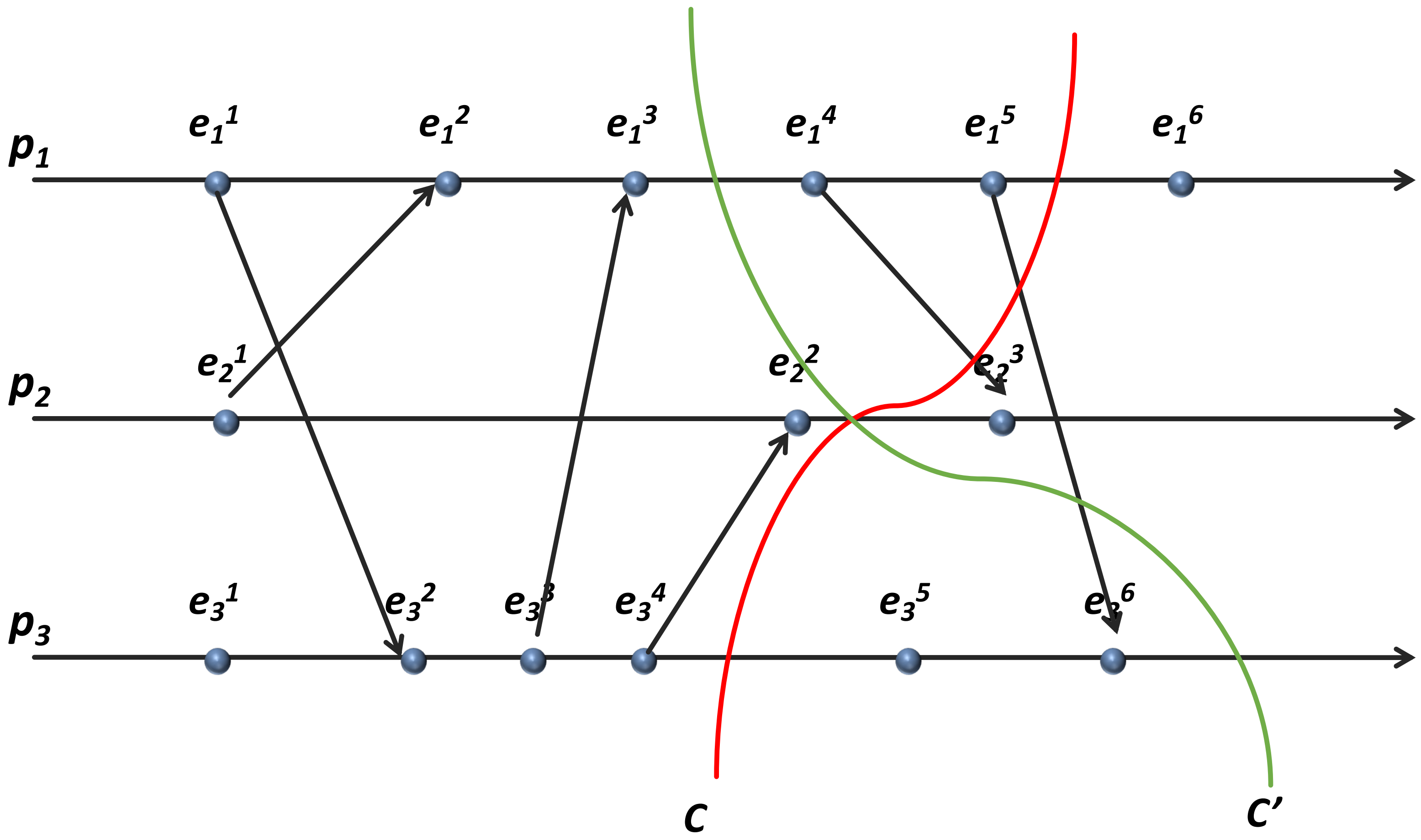
If $e \rightarrow e'$ *and* $e' \rightarrow e''$, *then* $e \rightarrow e''$

Cuts of a Distributed Computation

- Suppose there is an ***external monitor*** process
- External monitor constructs a global state:
  - Asks processes to send it local history
- Global state constructed from these local histories is:

a ***cut of a distributed computation***

Example Cuts

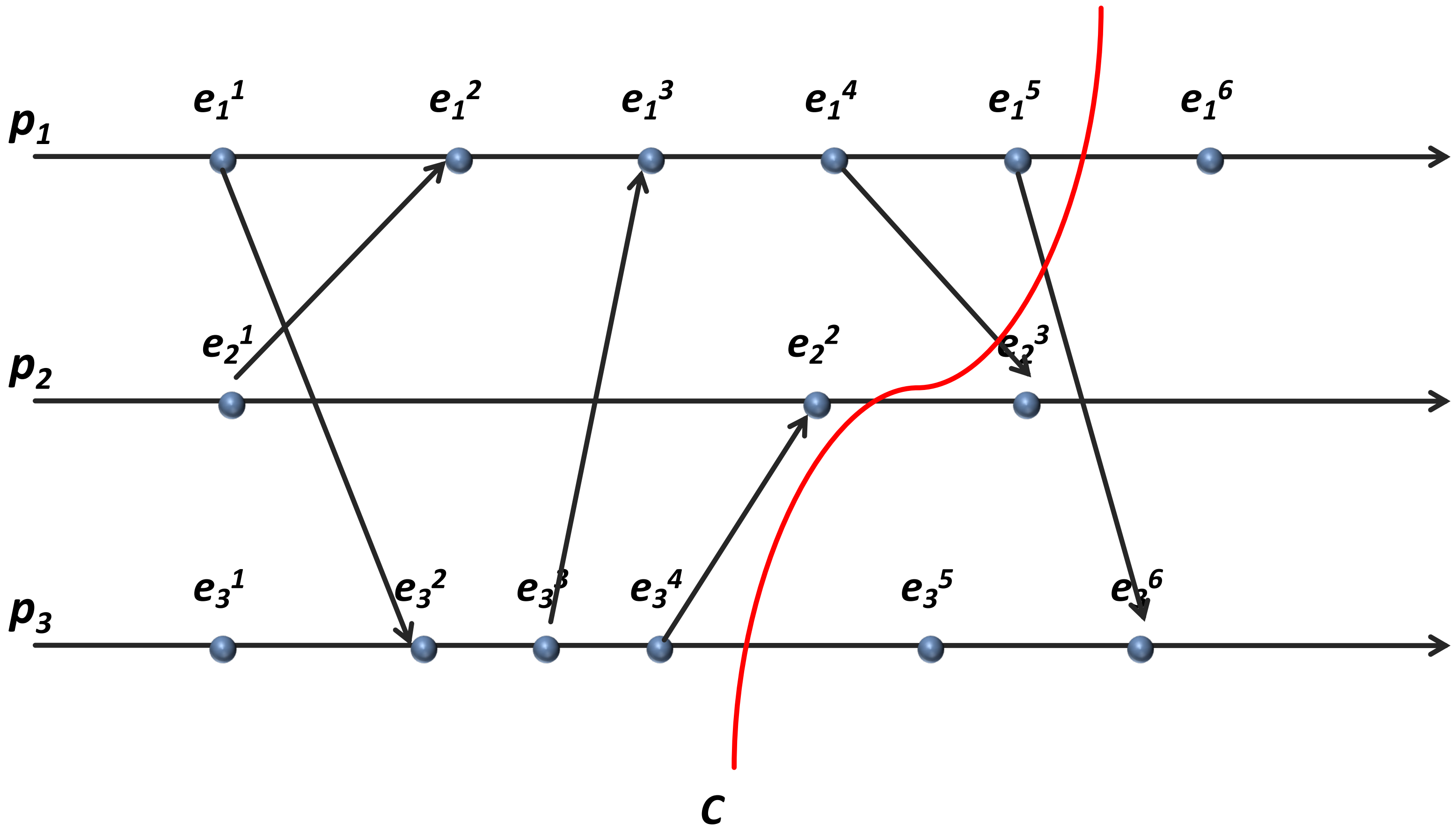


Consistent vs. Inconsistent Cuts

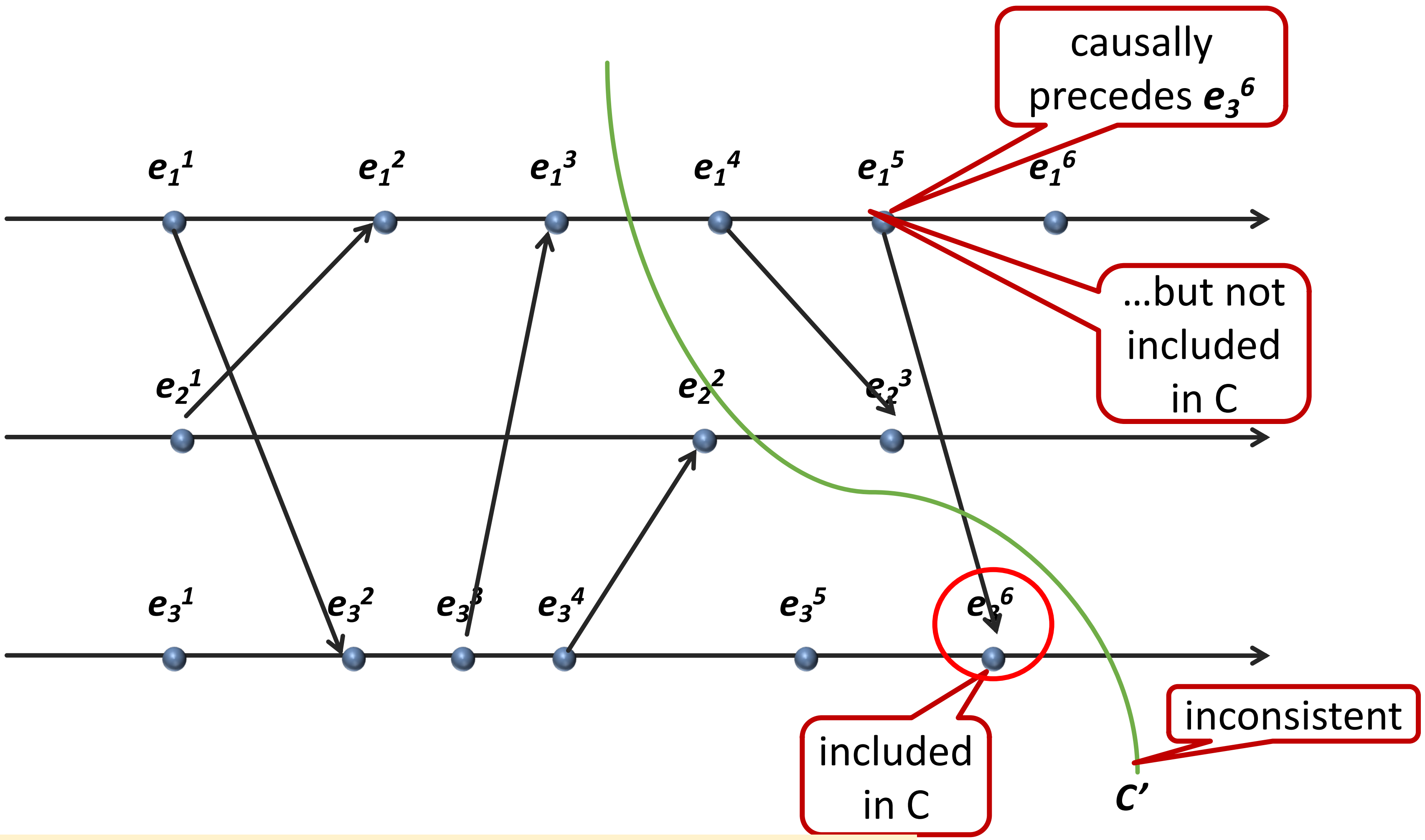
- A cut is consistent if
  - for any event *e* included in the cut
  - any event *e'* that causally precedes *e* is also included in that cut
- For cut *C:*

  $$(e \in C) \wedge (e' \rightarrow e) \Rightarrow e' \in C$$

Are These Cuts Consistent?

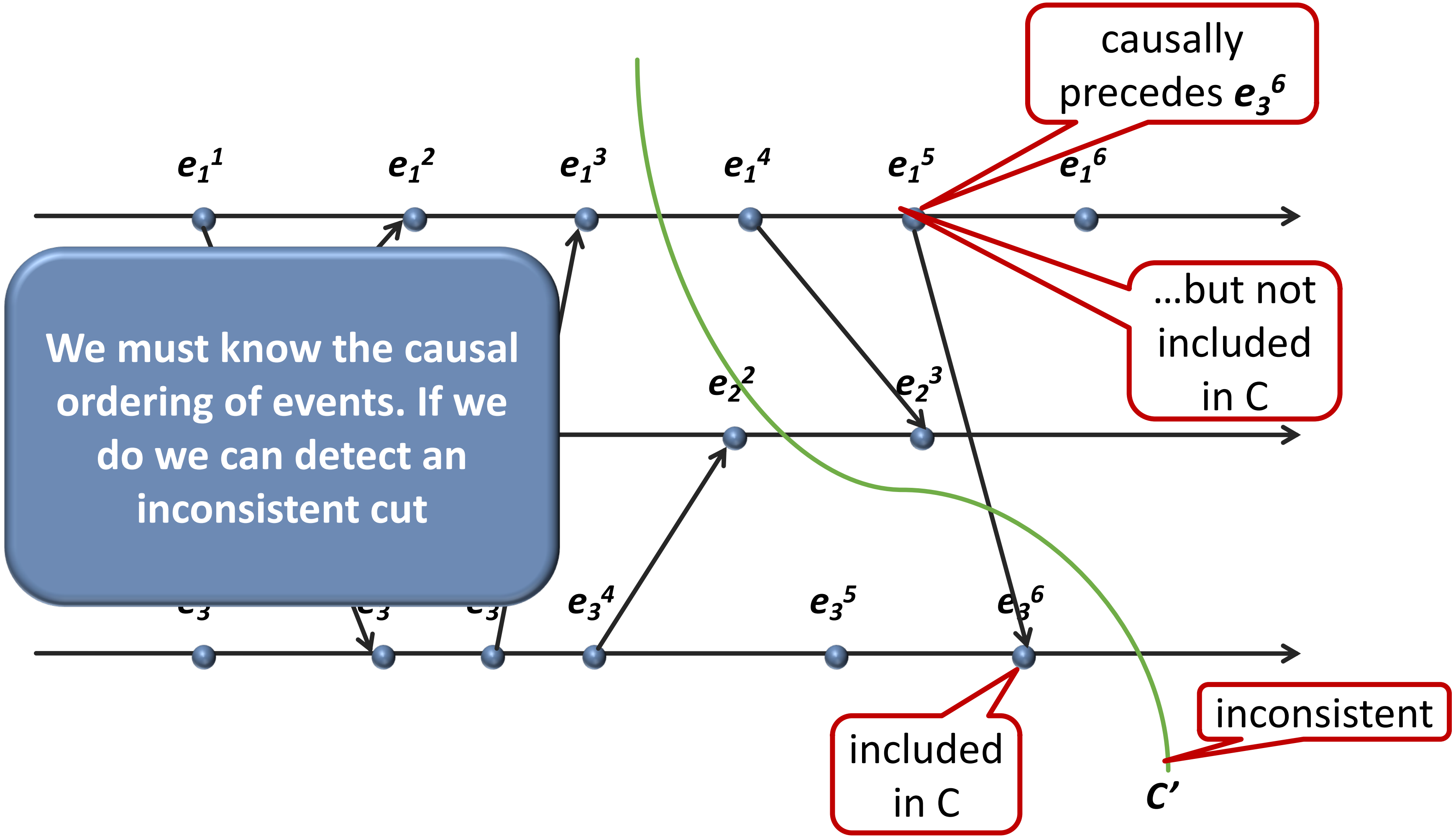


Are These Cuts Consistent?



A consistent cut corresponds to a consistent global state

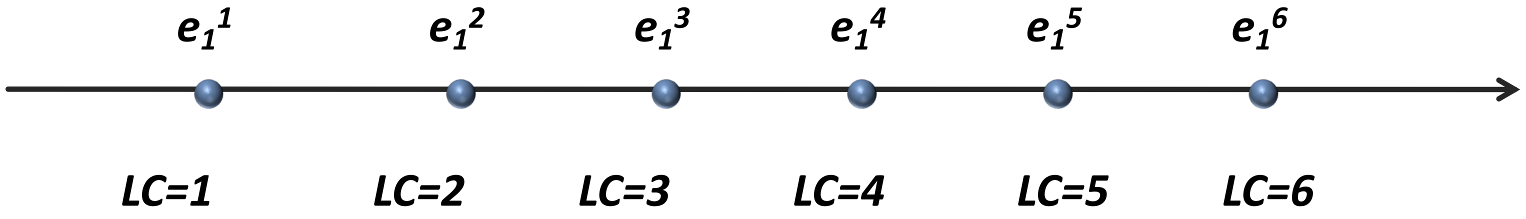
What Do We Need to Know to Construct a Consistent Cut?



We must know the causal ordering of events. If we do we can detect an inconsistent cut

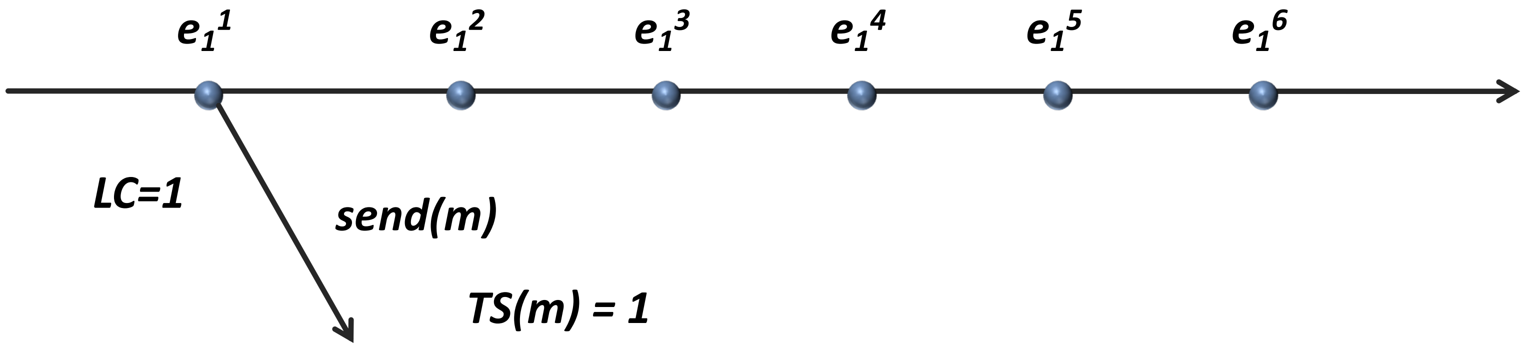
Logical Clocks

- Each process maintains a local value of a logical clock ***LC***
- Logical clock of process *p* counts **how many events in a distributed computation causally preceded the current event at *p*** (including the current event).
- $LC(e_i)$ – the logical clock value at process p_i at event e_i
- Suppose we had a distributed system with only a single process



Logical Clocks (cont.)

- In a system with more than one process logical clocks are updated as follows:
- Each message m that is sent contains a timestamp TS(m)
- TS(m) is the logical clock value associated with sending event at the sending process



Logical Clocks (cont)

- When the receiving process receives message m, it updates its logical clock to:

$$max\{LC, TS(m)\} + 1$$

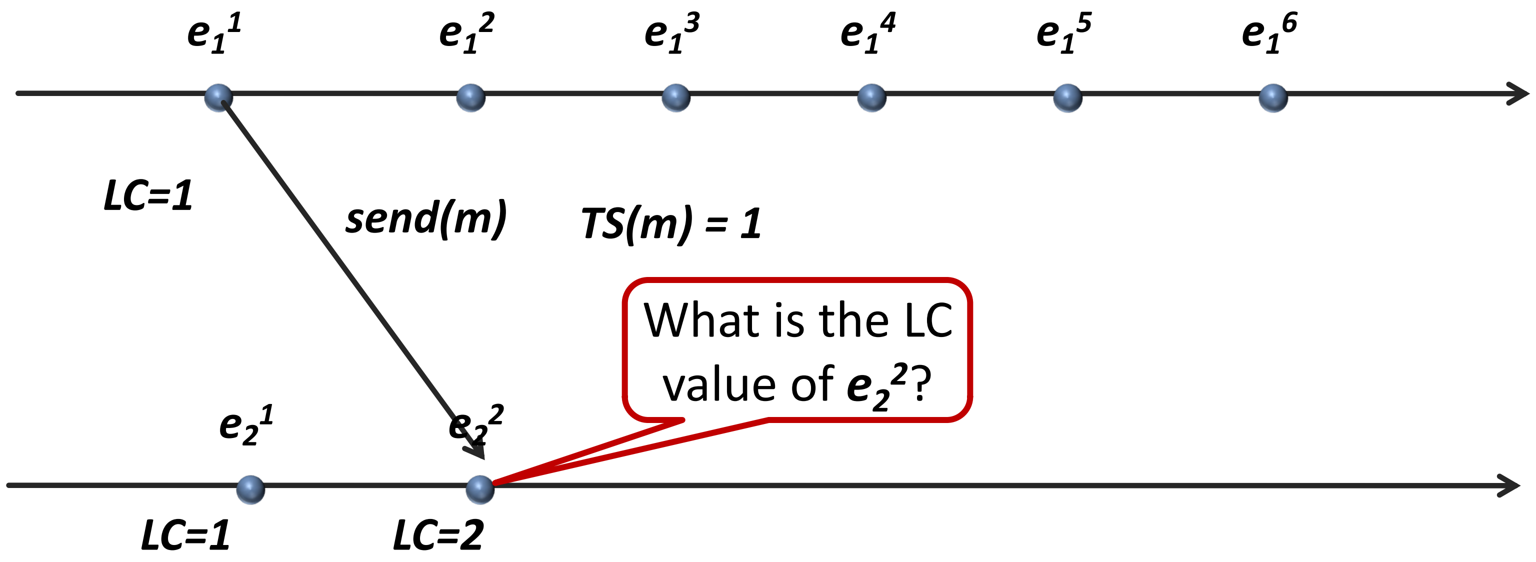
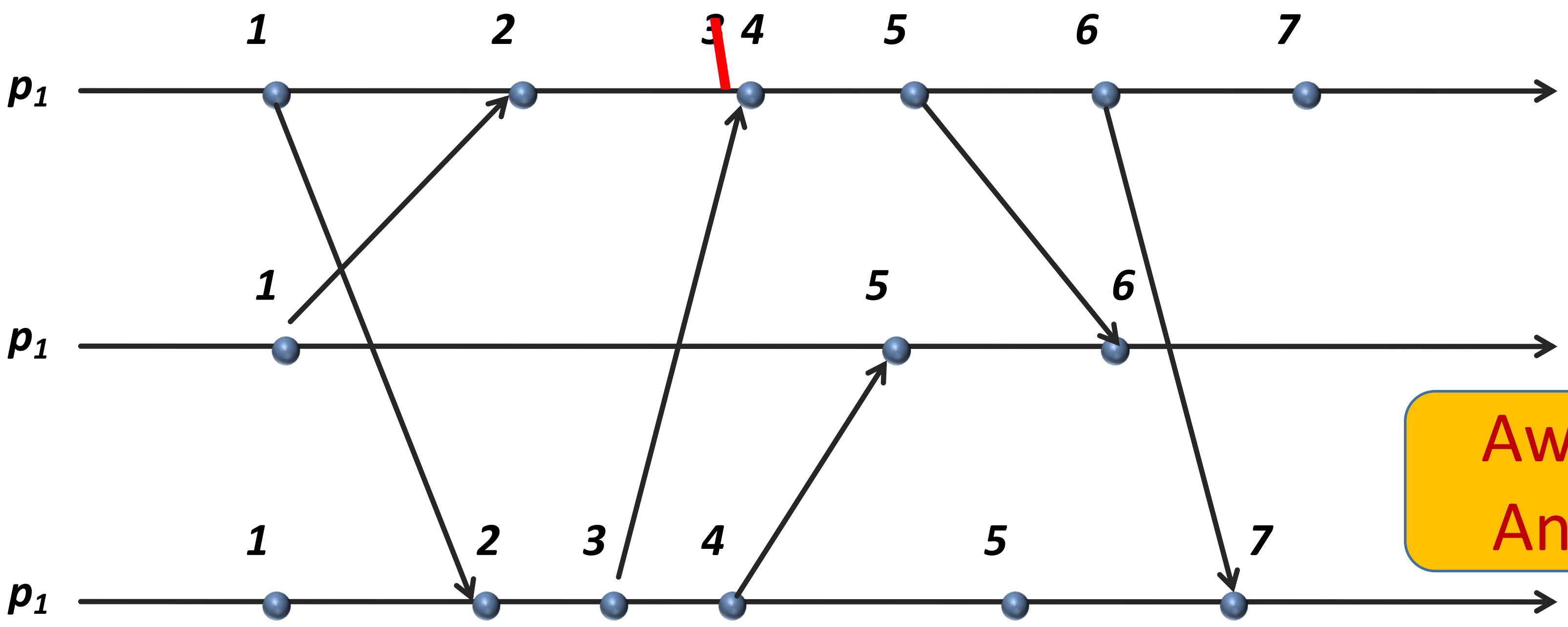


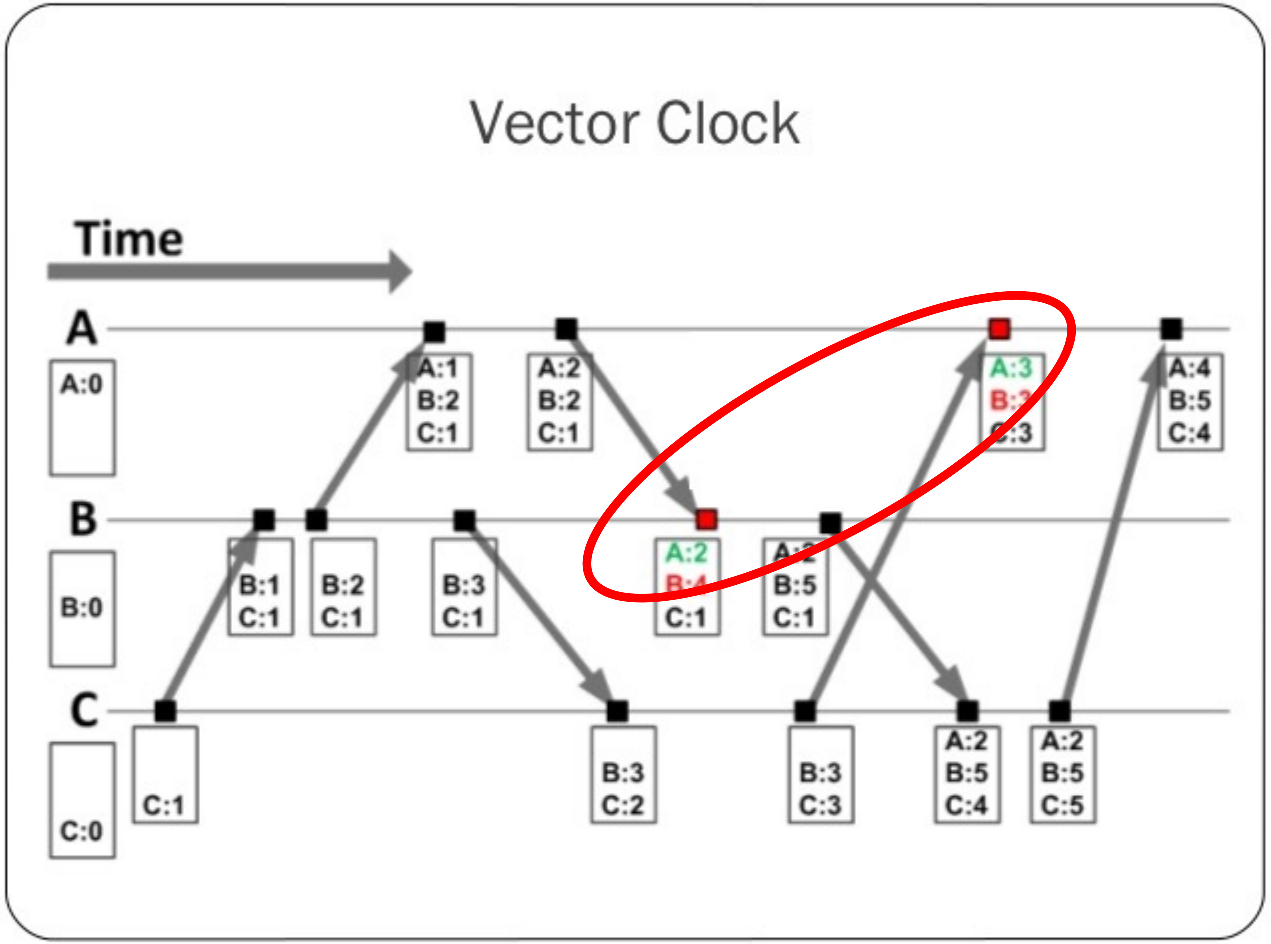
Illustration of a Logical Clock



Awesome, right?
Any drawbacks?

e_x < e_y → TS(e_x) < TS(e_y), but
TS(e_x) < TS(e_y) doesn't guarantee e_x < e_y

Vector Clock



Replace Single Logical value with Vector!

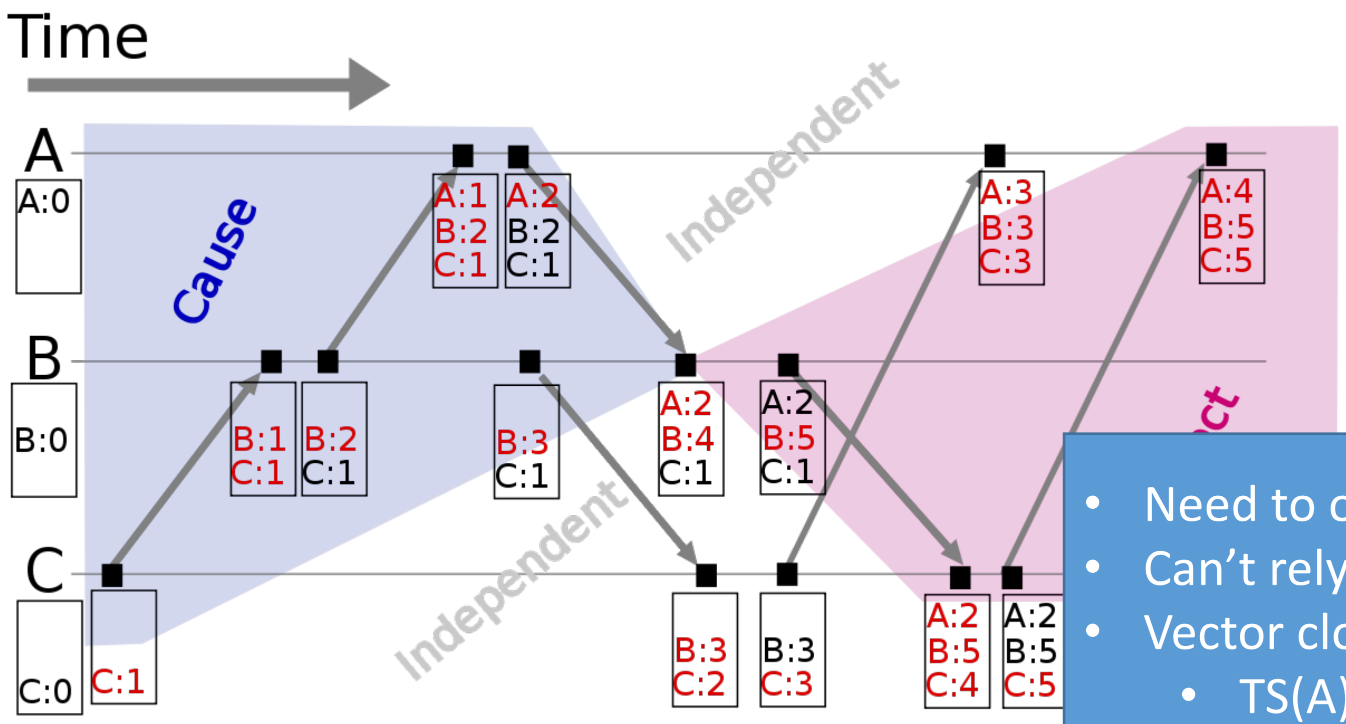
$V_i[i]$: #events occurred at i

$V_i[j]$: #events i knows occurred at j

Update

- On local-event: increment $V_i[I]$
- On send-message: increment, piggyback entire local vector V
- On recv-message: *$V_j[k]$* = max(*$V_j[k], V_i[k]$*)
  - *$V_j[i] = V_j[i]+1$ (increment local clock)*
  - Receiver learns about number of events sender knows occurred elsewhere

Vector Clock Example



Each process *i* maintains a vector V_i

- $V_i[i]$: number of events that have occurred at i
- $V_i[j]$: number of events I knows have occurred at process j

Update

- Local event: increment $V_i[I]$

r V

$V_i[k]$)

s the

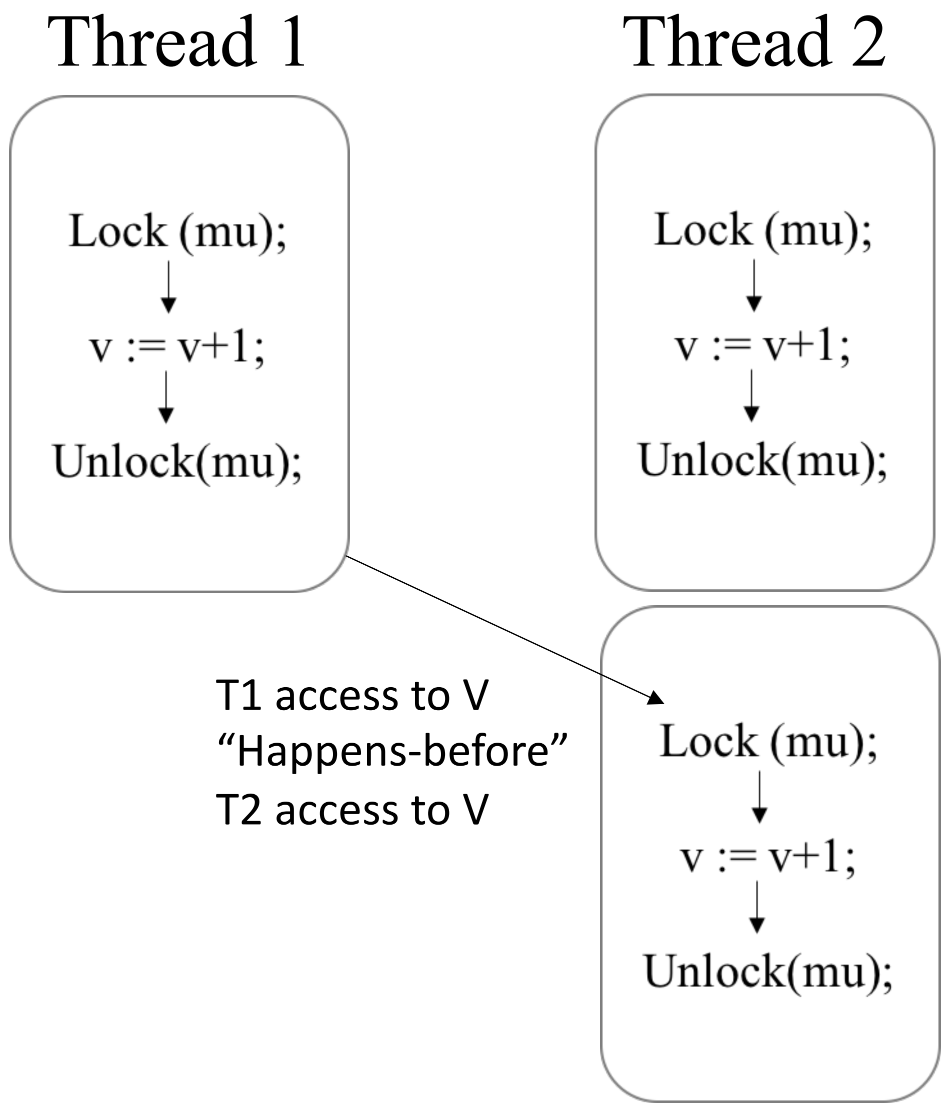
cess *k*

- Need to order operations
- Can’t rely on real-time
- Vector clock: timestamping algorithm s.t.
  - TS(A) < TS(B) → A happens before B
  - Independent ops remain unordered

See any drawbacks?

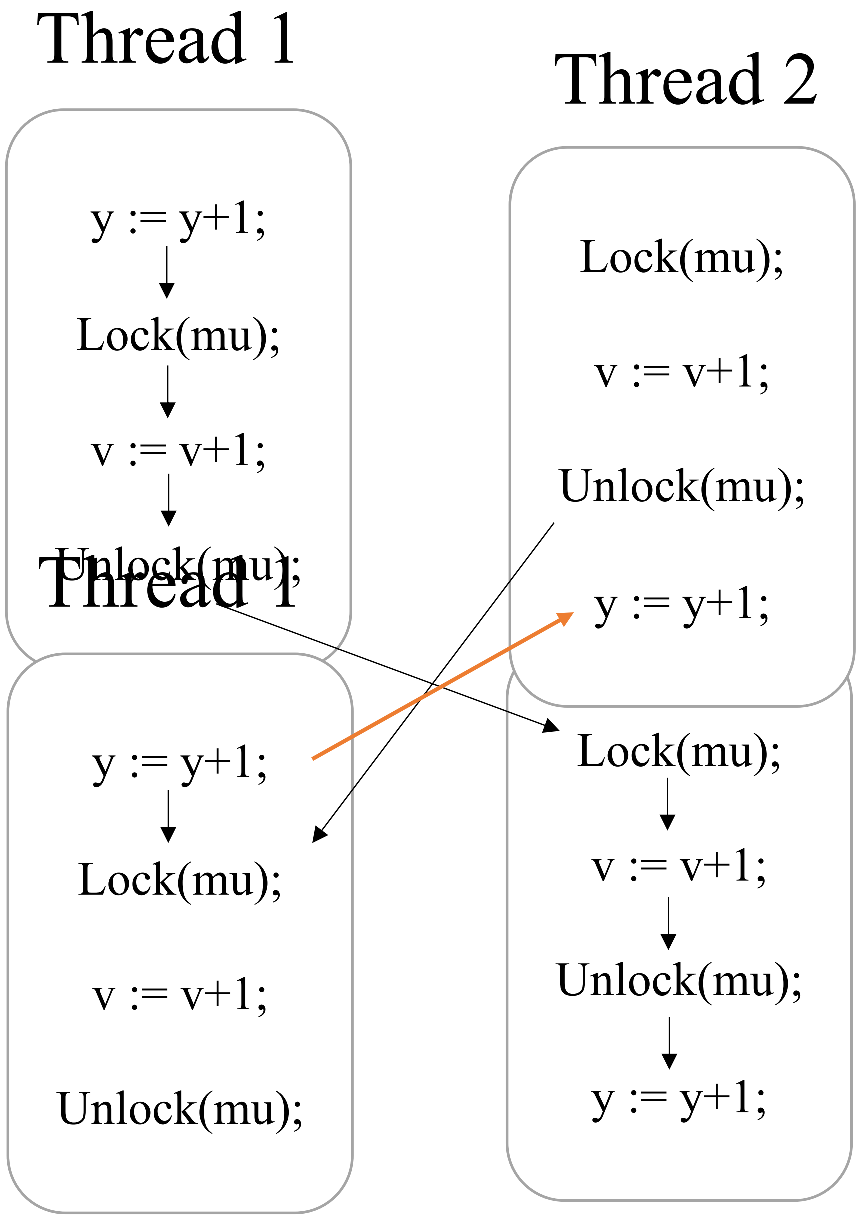
Happens-before

- *Happens-before* relation
  - Within single thread
  - Between threads
- Accessing variables not ordered by "happens-before" is a race
- Captures locks and dynamism
- How to track "happens-before"?
  - Sync objects are ordering events
  - Generalizes to fork/join, etc



Flaws of *Happens-before*

- Difficult to implement
  - Requires per-thread information
- Dependent on the interleaving produced by the scheduler
- Example
  - T1-acc(v) happens before T2-acc(v)
  - T1-acc(y) happens before T1-acc(v)
  - T2-acc(v) happens before T2-acc(y)
  - Conclusion: no race on Y!
  - Finding doesn't generalize

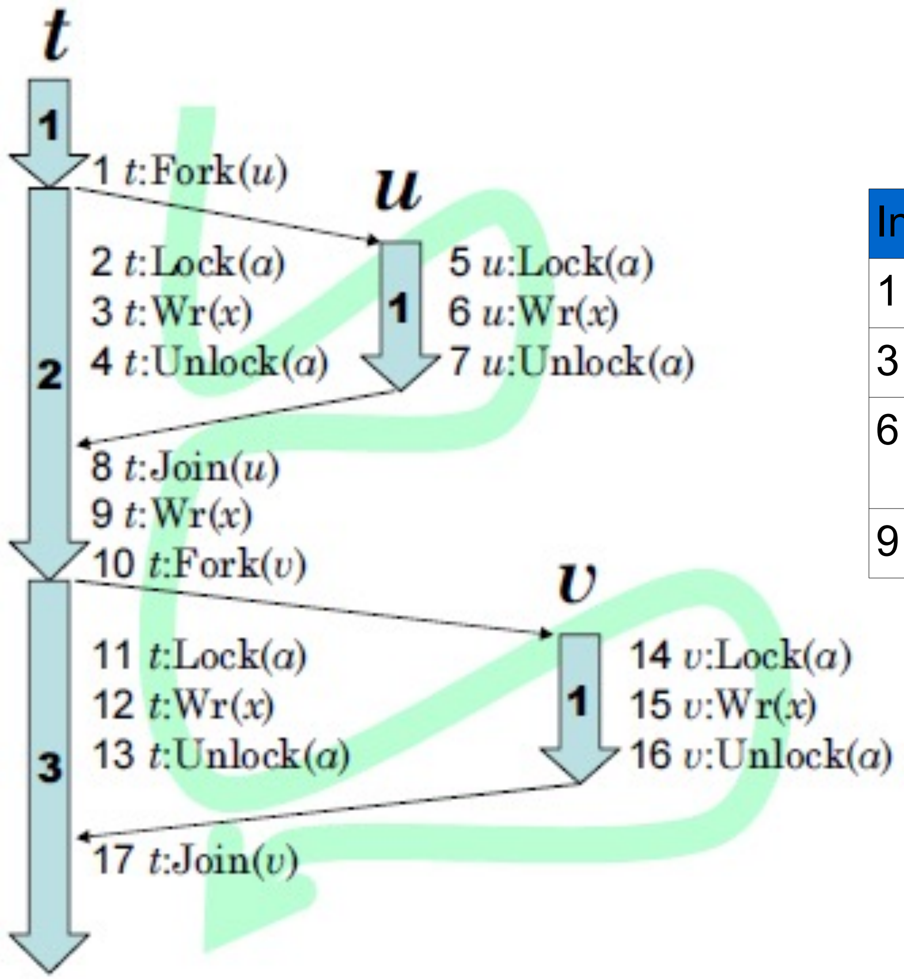


Dynamic Race Detection Summary

- Lockset: verify locking discipline for shared memory
  - ✓ Detect race regardless of thread scheduling
  - ✗ False positives because other synchronization primitives (fork/join, signal/wait) not supported
- Happens-before: track partial order of program events
  - ✓ Supports general synchronization primitives
  - ✗ Higher overhead compared to lockset
  - ✗ False negatives due to sensitivity to thread scheduling

RaceTrack = Lockset + Happens-before

False positive using Lockset



Tracking accesses to X

Inst	State	Lockset
1	Virgin	{ }
3	Exclusive:**t**	{ }
6	Shared Modified	{**a**}
9	Report race	{ }

RaceTrack Notations

Notation	Meaning
L_t	Lockset of thread **t**
C_x	Lockset of memory **x**
B_u	Vector clock of thread **u**
S_x	Threadset of memory **x**
$\mathbf{t}_i$	Thread **t** at clock time **i**

$$
\begin{aligned}
|V| &\triangleq |\{t \in T : V(t) > 0\}| \\
Inc(V,t) &\triangleq u \mapsto \text{if } u = t \text{ then } V(u) + 1 \text{ else } V(u) \\
Merge(V,W) &\triangleq u \mapsto max(V(u), W(u)) \\
Remove(V,W) &\triangleq u \mapsto \text{if } V(u) \leq W(u) \text{ then } 0 \text{ else } V(u)
\end{aligned}
$$

RaceTrack Algorithm

Notation	Meaning
L_t	Lockset of thread **t**
C_x	Lockset of memory **x**
B_t	Vector clock of thread **t**
S_x	Threadset of memory **x**
$\mathbf{t_1}$	Thread **t** at clock time 1

$$
\begin{aligned}
|V| &\triangleq |\{t \in T : V(t) > 0\}| \\
Inc(V,t) &\triangleq u \mapsto \text{if } u = t \text{ then } V(u) + 1 \text{ else } V(u) \\
Merge(V,W) &\triangleq u \mapsto max(V(u), W(u)) \\
Remove(V,W) &\triangleq u \mapsto \text{if } V(u) \leq W(u) \text{ then } 0 \text{ else } V(u)
\end{aligned}
$$

At t:Lock(l):
    $L_t \leftarrow L_t \cup \{l\}$

At t:Unlock(l):
    $L_t \leftarrow L_t - \{l\}$

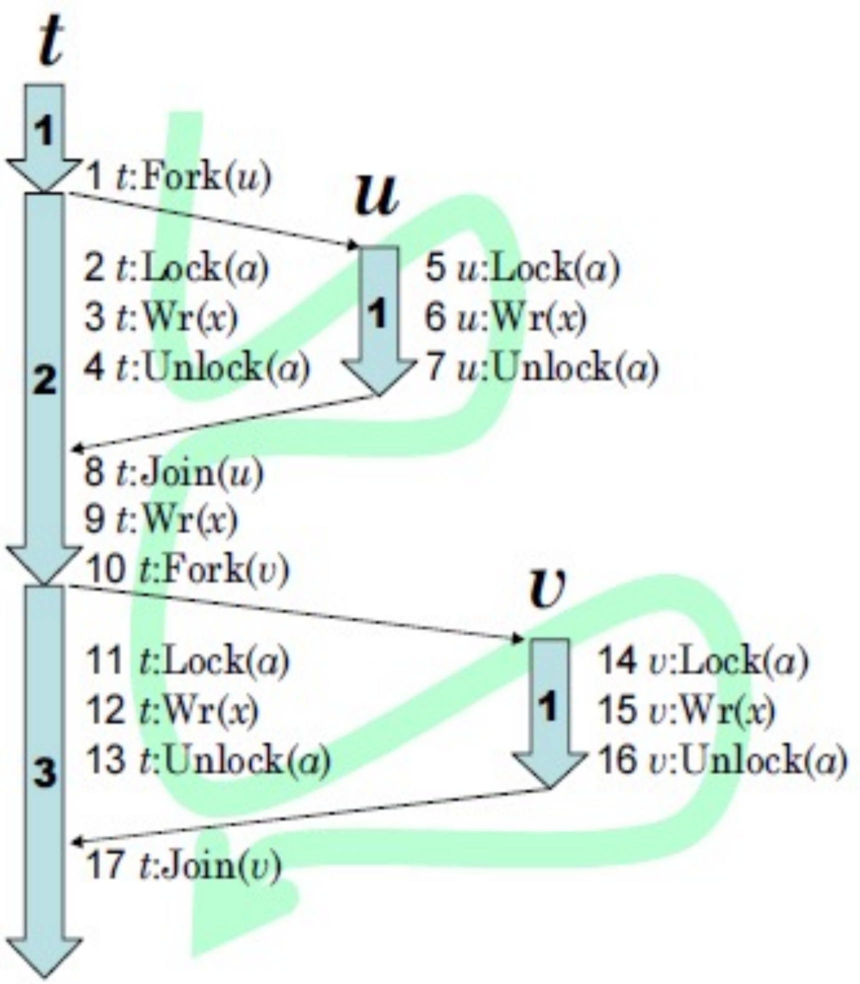
At t:Fork(u):
    $L_u \leftarrow \{\}$
    $B_u \leftarrow Merge(\{\langle u, 1\rangle\}, B_t)$
    $B_t \leftarrow Inc(B_t, t)$

At t:Join(u):
    $B_t \leftarrow Merge(B_t, B_u)$

At t:Rd(x) or t:Wr(x):
    $S_x \leftarrow Merge(Remove(S_x, B_t), \{\langle t, B_t(t)\rangle\})$
    if $|S_x| > 1$
        then $C_x \leftarrow C_x \cap L_t$
        else $C_x \leftarrow L_t$
    if $|S_x| > 1 \wedge C_x = \{\}$ then report race

Avoiding Lockset's false positive (1)

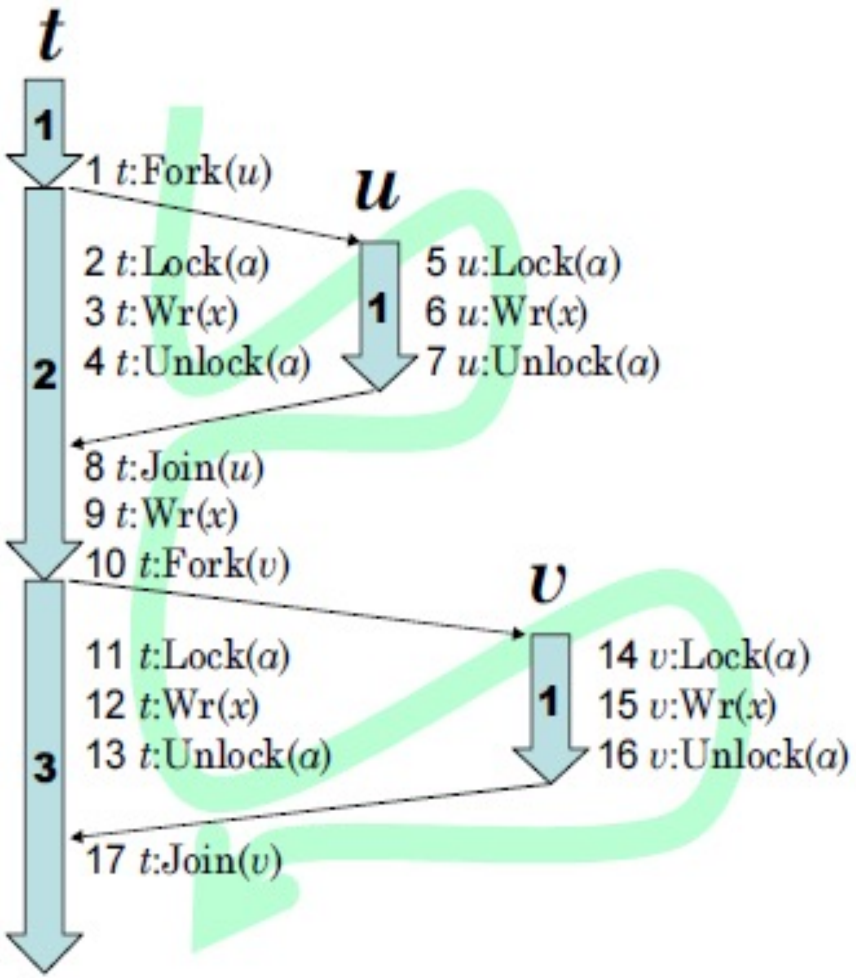
Notation	Meaning
L_t	Lockset of thread **t**
C_x	Lockset of memory **x**
B_t	Vector clock of thread **t**
S_x	Threadset of memory **x**
$\mathbf{t_1}$	Thread **t** at clock time 1



Inst	C_x	S_x	L_t	B_t	L_u	B_u
0	All	{ }	{ }	$\{t_1\}$	-	-
1				$\{t_2\}$	{ }	$\{ t_1, u_1 \}$
2			{a}			
3	{a}	$\{t_2\}$				
4			{ }			
5					{a}	
6		$\{t_2, u_1\}$				
7					{ }	
8				$\{t_2, u_1\}$	-	-

Notation	Meaning
L_t	Lockset of thread **t**
C_x	Lockset of memory **x**
B_t	Vector clock of thread **t**
S_x	Threadset of memory **x**
t_1	Thread **t** at clock time 1

Avoiding Lockset's false positive (2)



Inst	C_x	S_x	L_t	B_t	L_v	B_v
8	{a}	$\{t_2,u_1\}$	{ }	$\{t_2,u_1\}$	-	-
9	{ }	$\{t_2\}$				
10				$\{t_3,u_1\}$	{ }	$\{t_2,v_1\}$
11			{a}			
12	{a}	$\{t_3\}$				
13			{ }			
14					{a}	
15		$\{t_3,v_1\}$				
16					{ }	

Only one thread!

Are we done?