CMPT 745
Software Engineering

Concurrency & Parallelism

Nick Sumner
wsumner@sfu.ca

Concurrency & Parallelism

Seeking out performance

- Improving performance can come from tuning
  - Algorithmic complexity
  - Memory access patterns
  - Concurrency
  - Parallelism

Seeking out performance

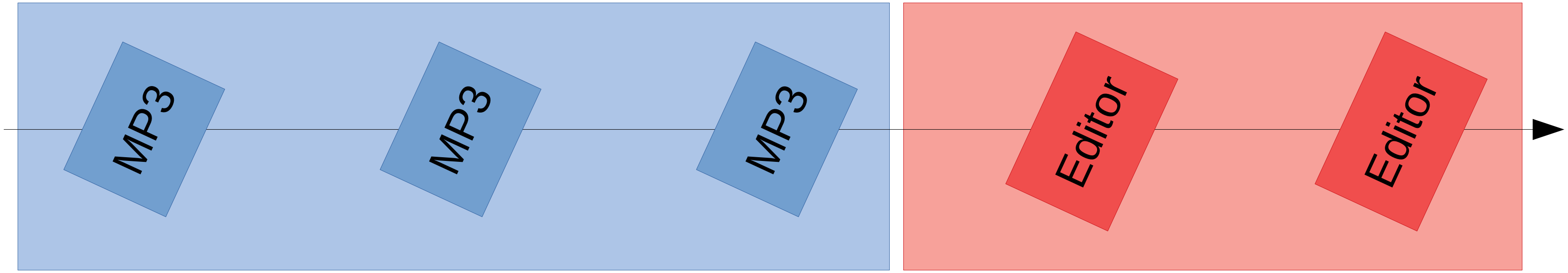
- Improving performance can come from tuning
  - Algorithmic complexity
  - Memory access patterns
  - Concurrency
  - Parallelism
- As processor speeds have slowed increasing, much focus has been placed on the last two

Concurrency & Parallelism

- ***Concurrency*** is the management of multiple tasks at the same time.
  - e.g. Sharing a CPU across multiple processes.

Concurrency & Parallelism

- ***Concurrency*** is the management of multiple tasks at the same time.
  - e.g. Sharing a CPU across multiple processes.



Sequential

Concurrency & Parallelism

- ***Concurrency*** is the management of multiple tasks at the same time.
  - e.g. Sharing a CPU across multiple processes.

Concurrency & Parallelism

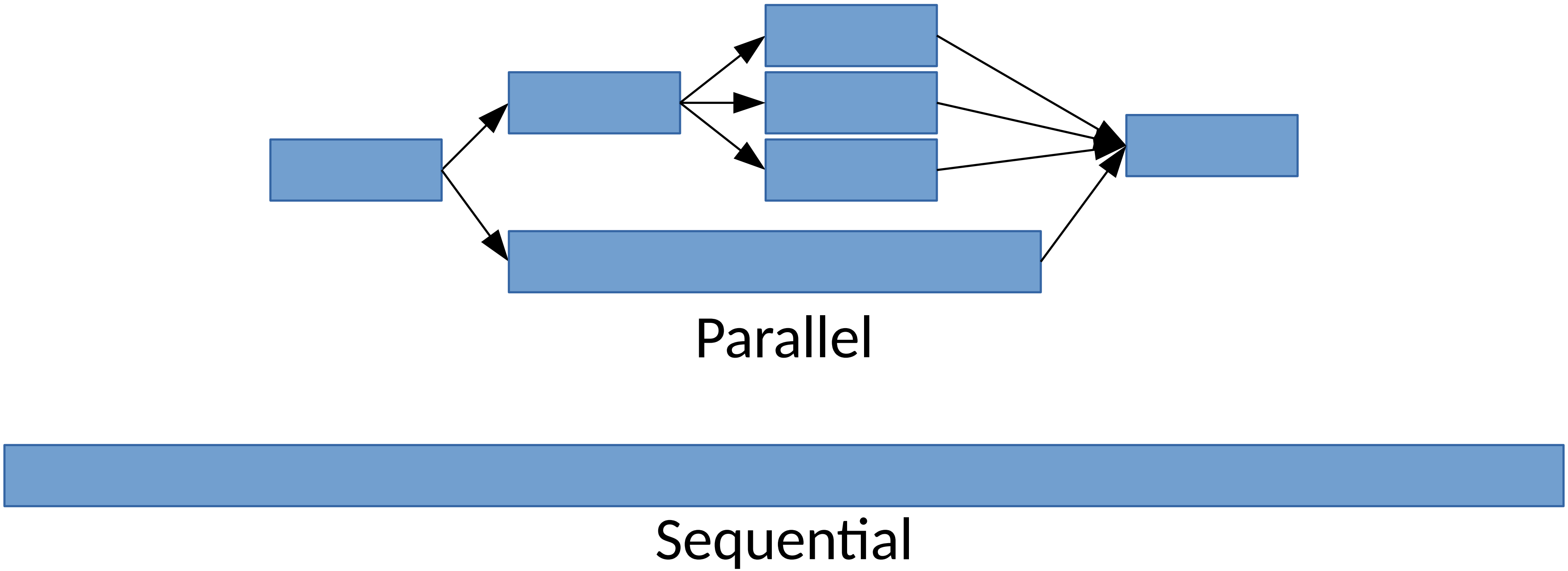
- ***Concurrency*** is the management of multiple tasks at the same time.
  - e.g. Sharing a CPU across multiple processes.
- ***Parallelism*** is using multiple resources to perform multiple tasks at the same time.
  - e.g. multiple cores for tasks, vector instructions

Concurrency & Parallelism

- ***Concurrency*** is the management of multiple tasks at the same time.
  - e.g. Sharing a CPU across multiple processes.
- ***Parallelism*** is using multiple resources to perform multiple tasks at the same time.
  - e.g. multiple cores for tasks, vector instructions

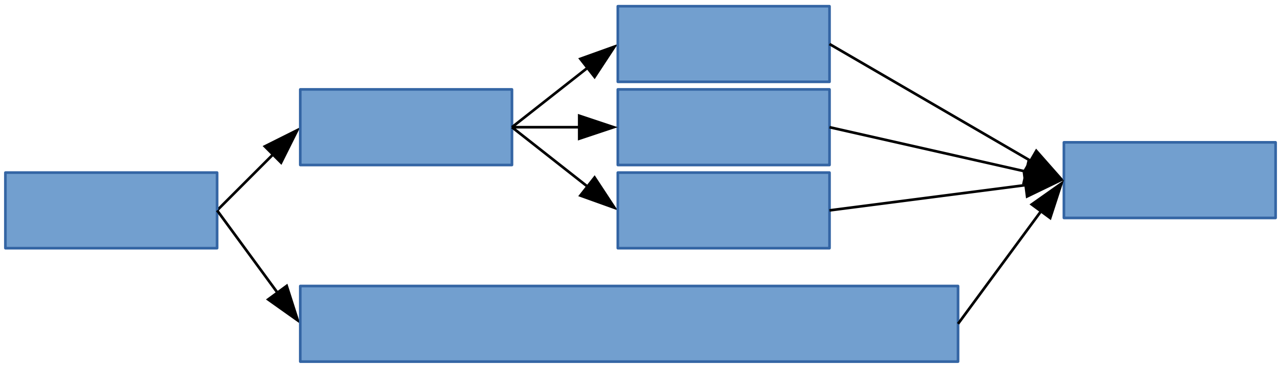
Using Parallelism

- Large problems can sometimes be split into parallel tasks, and the effects of the parallel tasks combined



Using Parallelism

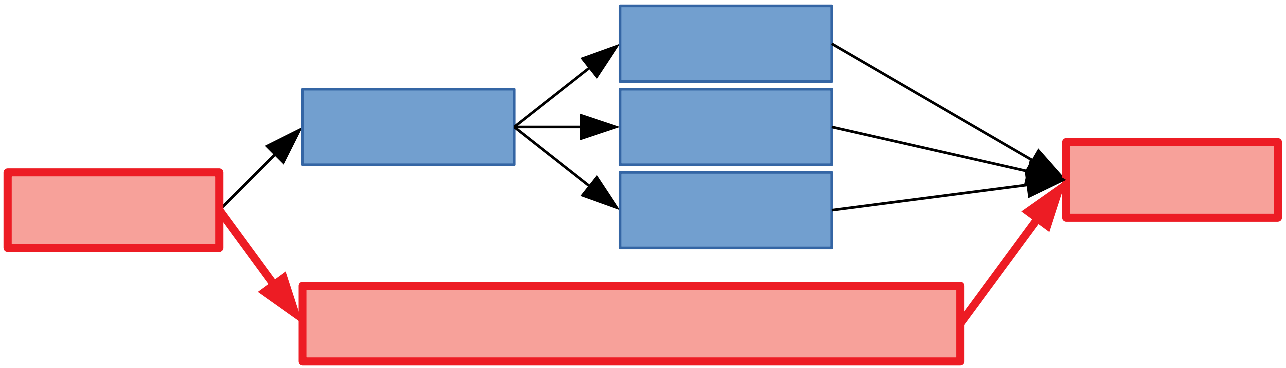
- Large problems can sometimes be split into parallel tasks, and the effects of the parallel tasks combined



- The best possible running time is determined by the ***critical path*** or ***span*** of dependent tasks through the program.

Using Parallelism

- Large problems can sometimes be split into parallel tasks, and the effects of the parallel tasks combined

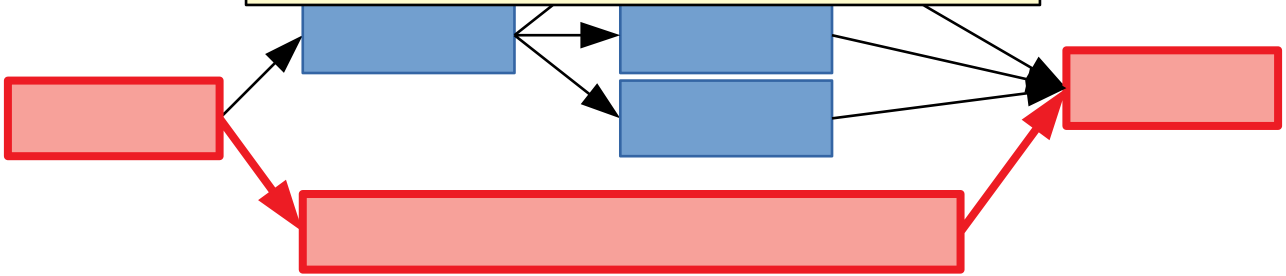


- The best possible running time is determined by the ***critical path*** or ***span*** of dependent tasks through the program.

Using Parallelism

- Large problems can sometimes be split into parallel tasks, and the effects of the para

This is too optimistic!
Why?



- The best possible running time is determined by the ***critical path*** or ***span*** of dependent tasks through the program.

Using Parallelism

- There are often more tasks than compute resources

Using Parallelism

- There are often more tasks than compute resources
  - Brent's Theorem describes the time accounting for limits

$$Given\ p\ processors,\ \frac{Time_1}{p} \leq Time_p \leq \frac{Time_1}{p} + Time_\infty$$

Using Parallelism

- There are often more tasks than compute resources
  - Brent's Theorem describes the time accounting for limits

$$Given\ p\ processors,\ \frac{Time_1}{p} \leq Time_p \leq \frac{Time_1}{p} + Time_\infty$$

- Identifying good opportunities for effective parallelism is open to research

Using Parallelism

- There are often more tasks than compute resources
  - Brent's Theorem describes the time accounting for limits

$$\text{Given } p \text{ processors}, \frac{Time_1}{p} \leq Time_p \leq \frac{Time_1}{p} + Time_\infty$$

- Identifying good opportunities for effective parallelism is open to research
  - Profiling for tasks to extract
  - Understanding the effect of speeding specific tasks
  - …

Correctness issues

- Parallel & concurrent code is challenging to write
  - Nondeterministic timing
  - Actions of one task may subtly affect others

Correctness issues

- Parallel & concurrent code is challenging to write
  - Nondeterministic timing
  - Actions of one task may subtly affect others
- Specifically
  - Deadlock / Livelock
  - Starvation
  - Data races
  - Atomicity violations
  - Order violations
  - …

Correctness issues

- Parallel & concurrent code is challenging to write
  - Nondeterministic timing
  - Actions of one task may subtly affect others
- Specifically
  - Deadlock / Livelock
  - Starvation
  - Data races
  - Atomicity violations
  - Order violations
  - …

} 97% of real world concurrency bugs

[Lu, ASPLOS 2008]

Data Races

- A data race occurs when:

Data Races

- A data race occurs when:
  1) two threads access the same location

Data Races

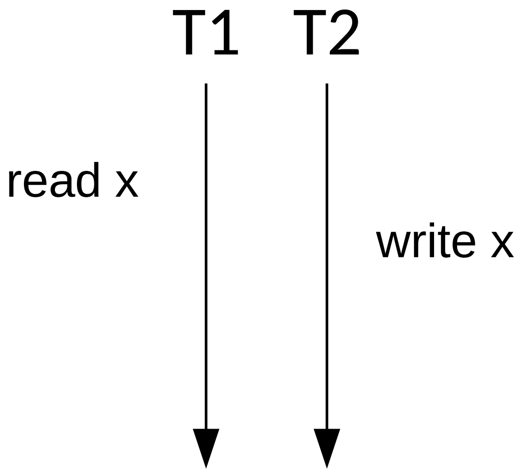
- A data race occurs when:
  1) two threads access the same location
  2) the accesses are *logically simultaneous*

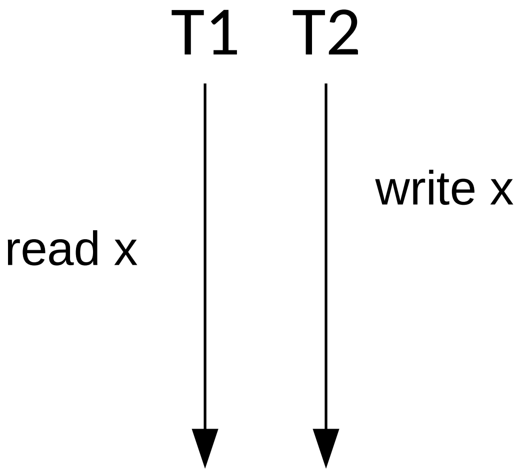
Data Races

- A data race occurs when:
  1) two threads access the same location
  2) the accesses are *logically simultaneous*
  3) at least one access is a write (WAW, WAR, RAW)

Data Races

- A data race occurs when:
  1) two threads access the same location
  2) the accesses are *logically simultaneous*
  3) at least one access is a write (WAW, WAR, RAW)





Data Races

x++ →

tmp = x
tmp = tmp+1
x = tmp

T1 T2

tmp_1 = x
tmp_1 = tmp_1+1
x = tmp_1

tmp_2 = x
tmp_2 = tmp_2+1
x = tmp_2

Data Races

x++ →

```
tmp = x
tmp = tmp+1
x = tmp
```

T1 T2

$x \mapsto 0$

T1:

$tmp_1 = x$
$tmp_1 = tmp_1+1$
$x = tmp_1$

T2:

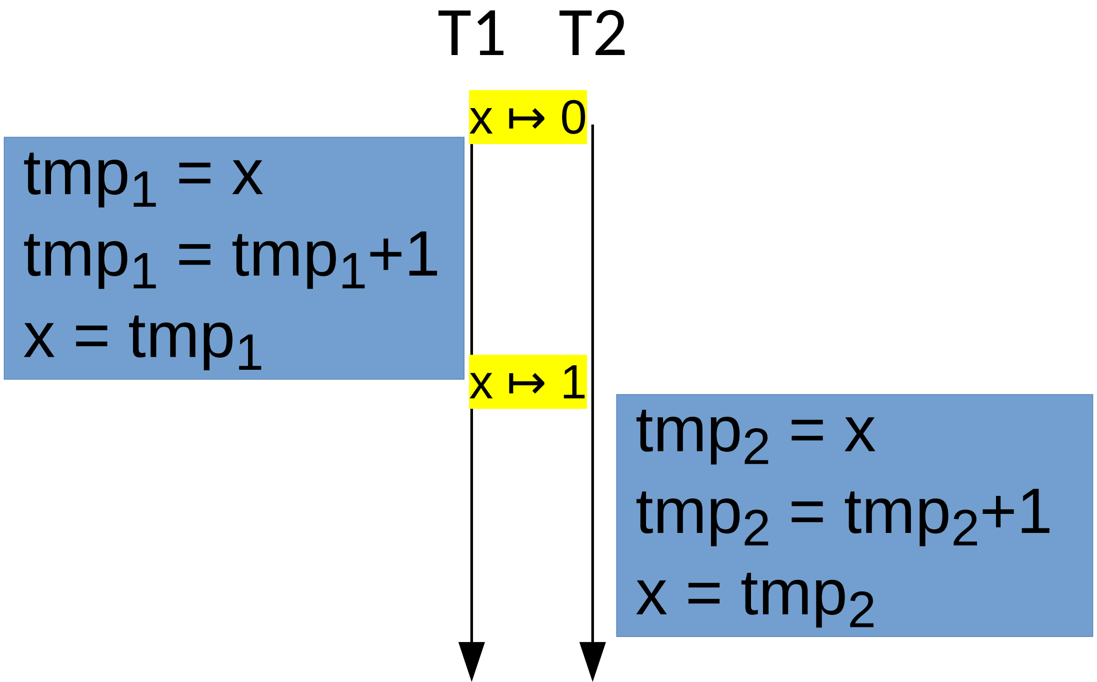
$tmp_2 = x$
$tmp_2 = tmp_2+1$
$x = tmp_2$

Data Races

```
x++
```

→

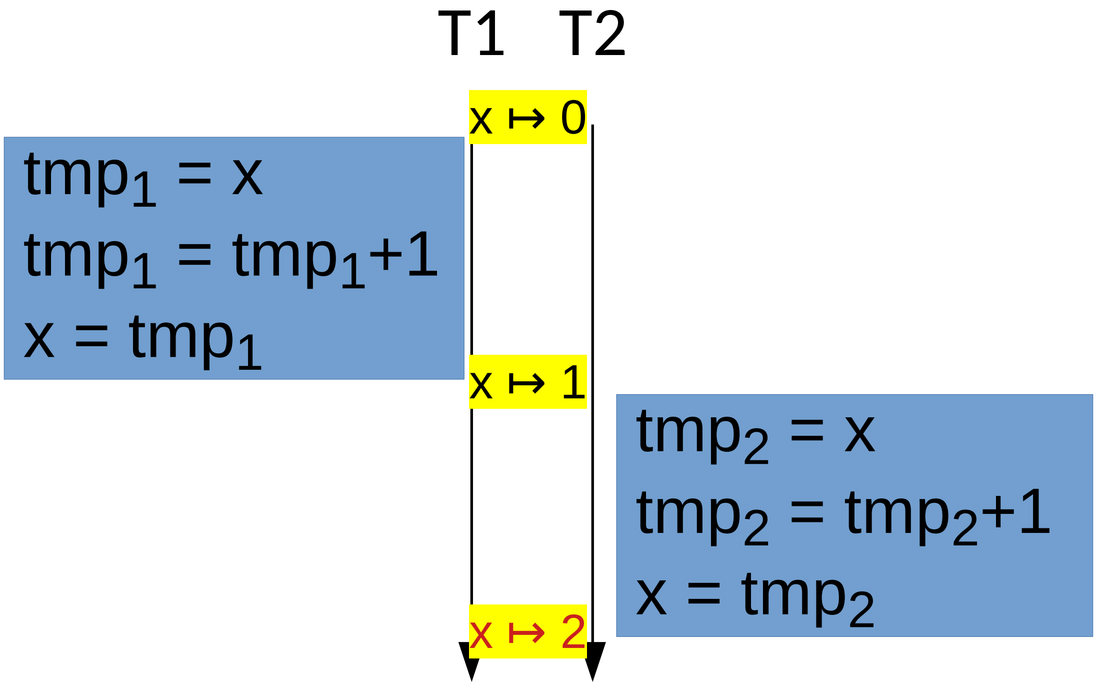
```
tmp = x
tmp = tmp+1
x = tmp
```



Data Races

x++ →

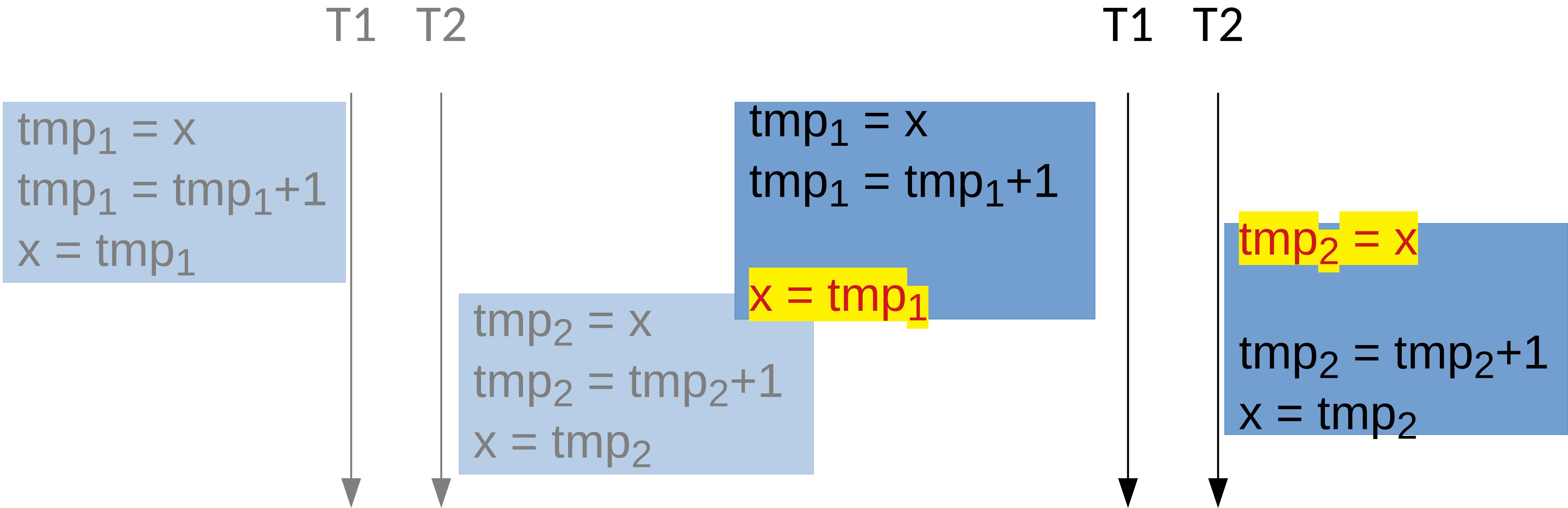
```
tmp = x
tmp = tmp+1
x = tmp
```



Data Races

x++ →

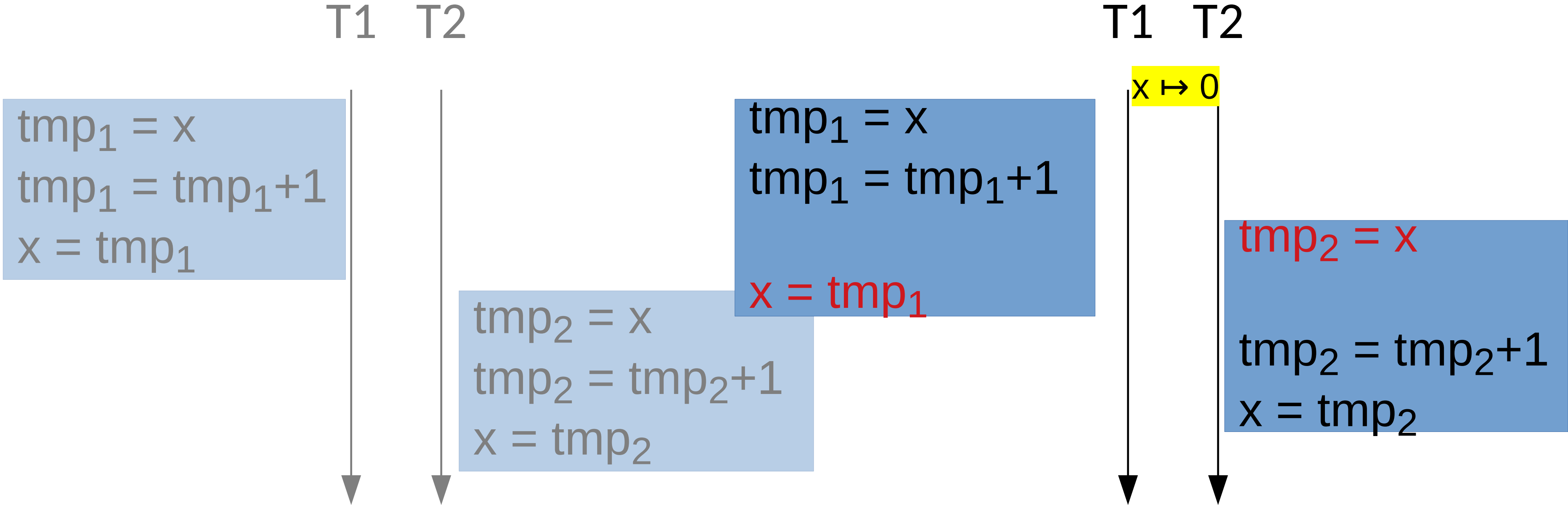
tmp = x
tmp = tmp+1
x = tmp



Data Races

x++ →

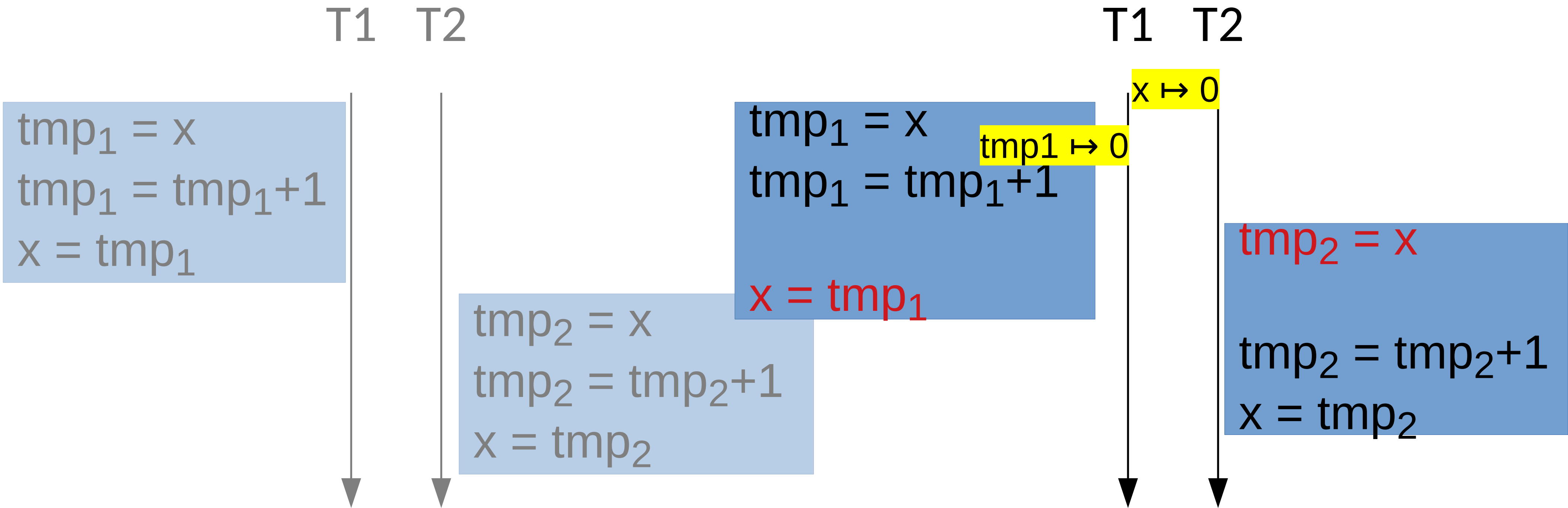
```
tmp = x
tmp = tmp+1
x = tmp
```



Data Races

x++ →

```
tmp = x
tmp = tmp+1
x = tmp
```



Data Races

x++ →

```
tmp = x
tmp = tmp+1
x = tmp
```

T1 T2

$tmp_1 = x$
$tmp_1 = tmp_1+1$
$x = tmp_1$

$tmp_2 = x$
$tmp_2 = tmp_2+1$
$x = tmp_2$

T1 T2

x ↦ 0

$tmp_1 = x$
tmp1 ↦ 0
$tmp_1 = tmp_1+1$

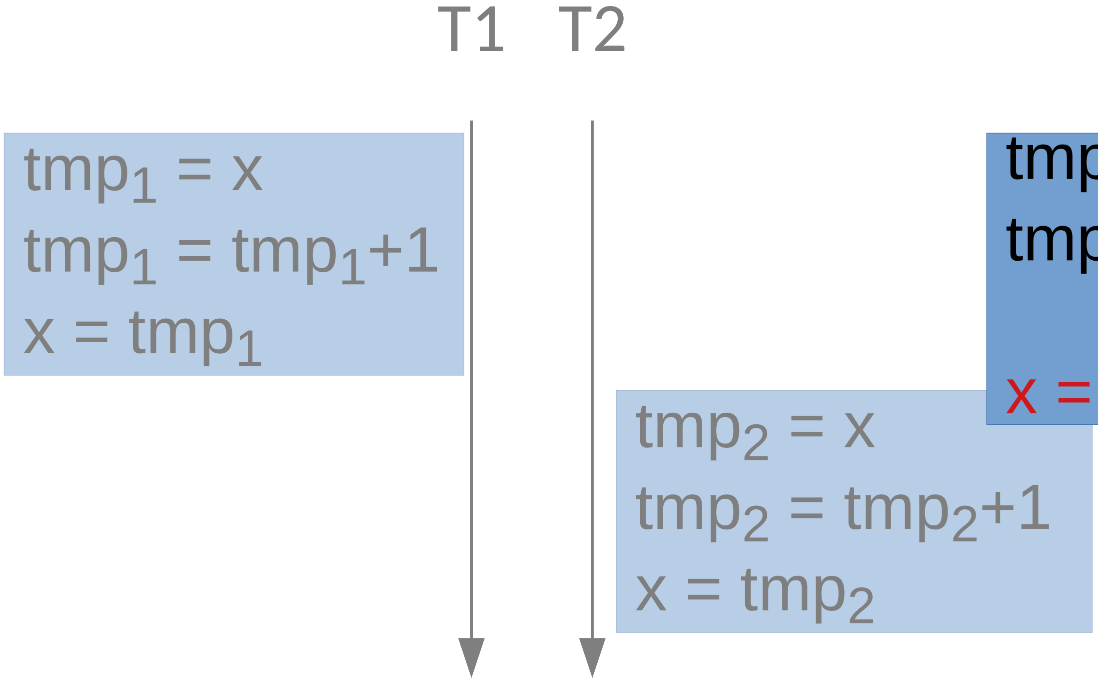
$x = tmp_1$

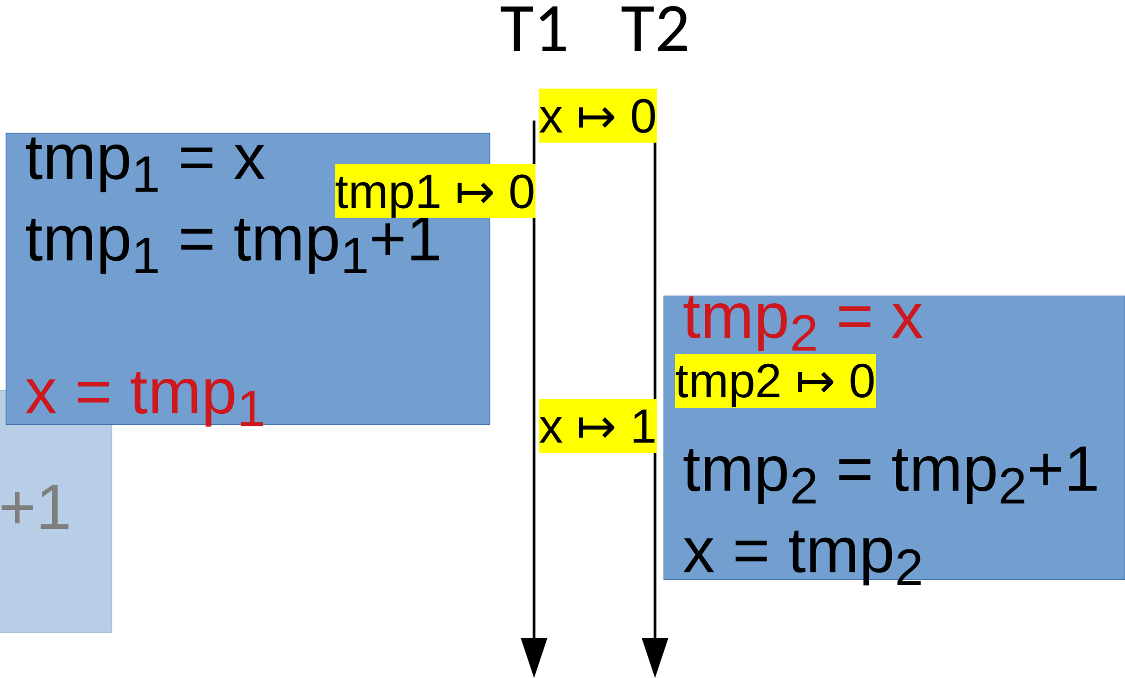
$tmp_2 = x$
tmp2 ↦ 0
$tmp_2 = tmp_2+1$
$x = tmp_2$

Data Races

x++ →

```
tmp = x
tmp = tmp+1
x = tmp
```





Data Races

x++ →

```
tmp = x
tmp = tmp+1
x = tmp
```

T1 T2

tmp_1 = x
tmp_1 = tmp_1+1
x = tmp_1

tmp_2 = x
tmp_2 = tmp_2+1
x = tmp_2

T1 T2

x ↦ 0

tmp_1 = x
tmp1 ↦ 0
tmp_1 = tmp_1+1

tmp_2 = x
tmp2 ↦ 0

x = tmp_1

x ↦ 1

tmp_2 = tmp_2+1
x = tmp_2

x ↦ 1

Data Races

x++ →

$tmp = x$
$tmp = tmp+1$
$x = tmp$

T1 T2

$tmp_1 = x$
$tmp_1 = tmp_1+1$
$x = tmp_1$

$tmp_2 = x$
$tmp_2 = tmp_2+1$
$x = tmp_2$

Synchronization discipline prevents data races.

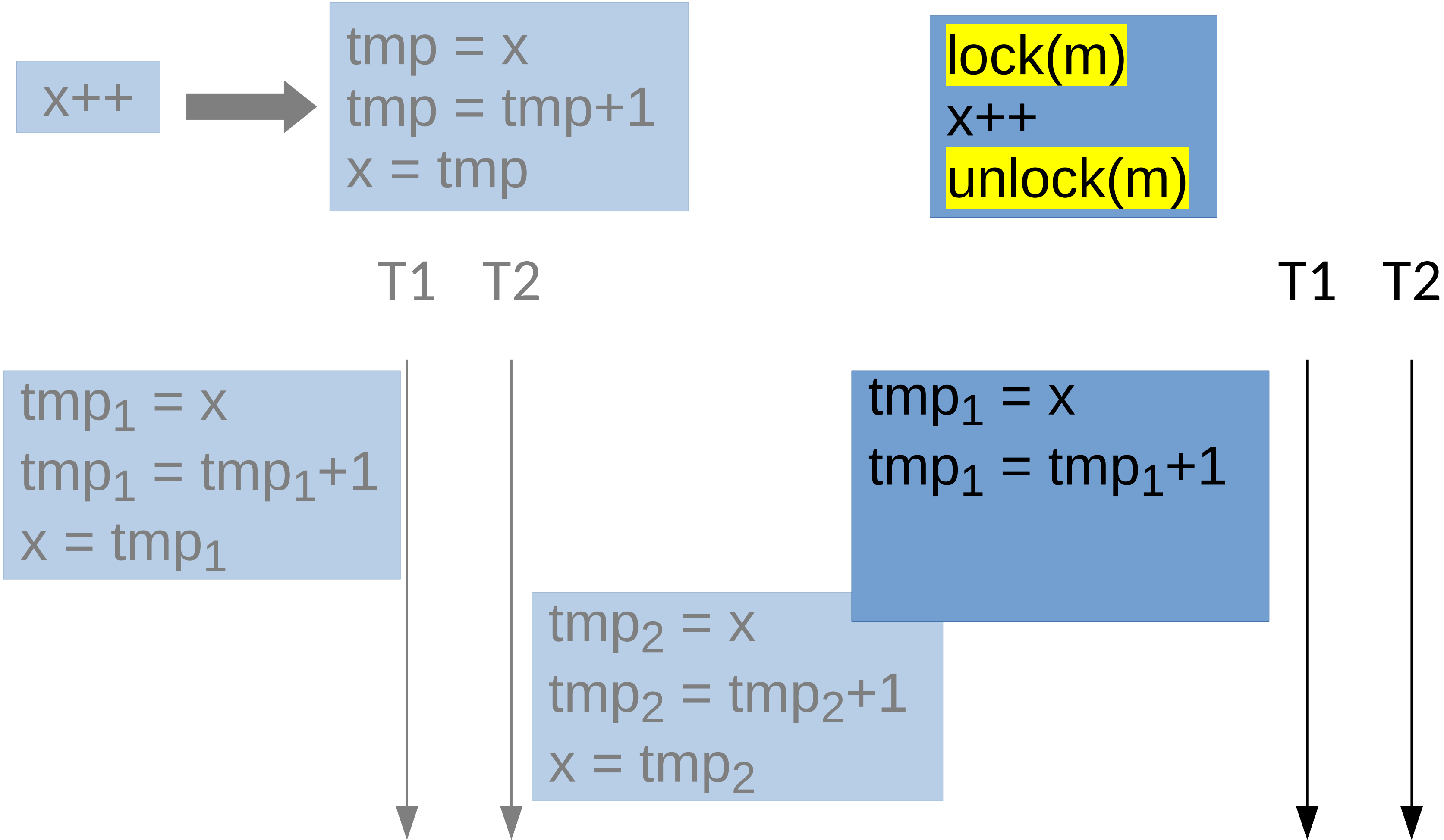
$x = tmp_1$

T1 T2

$tmp_2 = x$

$tmp_2 = tmp_2+1$
$x = tmp_2$

Data Races


x++
tmp = x
tmp = tmp+1
x = tmp
lock(m)
x++
unlock(m)
T1 T2
T1 T2
tmp_1 = x
tmp_1 = tmp_1+1
x = tmp_1
tmp_2 = x
tmp_2 = tmp_2+1
x = tmp_2
tmp_1 = x
tmp_1 = tmp_1+1

Data Races

```
x++  →  tmp = x
        tmp = tmp+1
        x = tmp
```

```
lock(m)
x++
unlock(m)
```

T1 T2

```
tmp1 = x
tmp1 = tmp1+1
x = tmp1
```

```
tmp2 = x
tmp2 = tmp2+1
x = tmp2
```

T1 T2

```
lock(m)
tmp1 = x
tmp1 = tmp1+1
```

Data Races

x++ →

tmp = x
tmp = tmp+1
x = tmp

lock(m)
x++
unlock(m)

T1 T2

tmp_1 = x
tmp_1 = tmp_1+1
x = tmp_1

tmp_2 = x
tmp_2 = tmp_2+1
x = tmp_2

T1 T2

lock(m)
tmp_1 = x
tmp_1 = tmp_1+1

The second task must wait.

Data Races

x++ →

tmp = x
tmp = tmp+1
x = tmp

lock(m)
x++
unlock(m)

T1 T2

tmp_1 = x
tmp_1 = tmp_1+1
x = tmp_1

tmp_2 = x
tmp_2 = tmp_2+1
x = tmp_2

T1 T2

lock(m)
tmp_1 = x
tmp_1 = tmp_1+1
x = tmp_1
unlock(m)

Data Races

x++ →

tmp = x
tmp = tmp+1
x = tmp

lock(m)
x++
unlock(m)

T1 T2

$tmp_1 = x$
$tmp_1 = tmp_1+1$
$x = tmp_1$

$tmp_2 = x$
$tmp_2 = tmp_2+1$
$x = tmp_2$

T1 T2

lock(m)
$tmp_1 = x$
$tmp_1 = tmp_1+1$
$x = tmp_1$
unlock(m)

lock(m)
$tmp_2 = x$
$tmp_2 = tmp_2+1$
$x = tmp_2$

“Benign” Data Races

- Sometimes a developer will make use of a data race
  - Avoid expensive synchronization
  - The race looks “benign” or harmless

“Benign” Data Races

- Sometimes a developer will make use of a data race
  - Avoid expensive synchronization
  - The race looks “benign” or harmless
- Both programming languages and hardware have memory models that determine what is *really* okay

“Benign” Data Races

- Sometimes a developer will make use of a data race
  - Avoid expensive synchronization
  - The race looks “benign” or harmless
- Both programming languages and hardware have memory models that determine what is really okay
  - A ***memory model*** determines what values may be read by a given memory access, esp. w.r.t. previous writes
    [CACM 2010, PLDI 2018]

“Benign” Data Races

```
if (!init) {
    lock();
    if (!init) {
        data = create();
        init = true;
    }
    unlock();
}
tmp = data;
```

[Boehm, Hotpar 2011]

Sometimes developers want to lazily initialize data

even with multiple threads.

“Benign” Data Races

```
if (!init) {
    lock();
    if (!init) {
        data = create();
        init = true;
    }
    unlock();
}
tmp = data;
```

[Boehm, Hotpar 2011]

- Threads race on `init`
- The compiler assumes no races while optimizing

"Benign" Data Races

```
if (!init) {
    lock();
    if (!init) {
        data = create();
        init = true;
    }
    unlock();
}
tmp = data;
```

[Boehm, Hotpar 2011]

- Threads race on `init`
- The compiler assumes no races while optimizing

```
if (!init) {
    lock();
    if (!init) {
        init = true;
        data = create();
    }
    unlock();
}
tmp = data;
```

Accesses can be reordered

"Benign" Data Races

```
if (!init) {
    lock();
    if (!init) {
        data = create();
        init = true;
    }
    unlock();
}
tmp = data;
```

[Boehm, Hotpar 2

- Threads race on `init`
- The compiler assumes no races while optimizing

```
if (!init) {
    lock();
    if (!init) {
        init = true;
    unlock();
}
tmp = data;
```

Data can even be loaded early.

This can even be done just by hardware!

```
tmp = data;
if (!init) {
    lock();
    if (!init) {
        data = create();
        tmp = data;
        init = true;
    }
    unlock();
}
```

“Benign” Data Races

```
local = counter;
if (local > localMax) {
    handler = ...;
}
update = work();
if (local > localMax) {
    handler(update);
}
```

[Boehm, Hotpar 2011]

Sometimes developers want to interleave conditional behavior.

“Benign” Data Races

```
local = counter;
if (local > localMax) {
    handler = ...;
}
update = work();
if (local > localMax) {
    handler(update);
}
```

[Boehm, Hotpar 2011]

- Data race freedom allows extra reads.

```
local = counter;
if (local > localMax) {
    handler = ...;
}
update = work();
if (counter > localMax) {
    handler(update);
}
```

"Benign" Data Races

```
local = counter;
if (local > localMax) {
    handler = ...;
}
update = work();
if (local > localMax) {
    handler(update);
}
```

[Boehm, Hotpar 2011]

- Data race freedom allows extra reads.

```
local = counter;
if (local > localMax) {
    handler = ...;
}
update = work();
if (counter > localMax) {
    handler(update);
}
```

Other Thread

```
counter = ...;
```

“Benign” Data Races

```
local = counter;
if (local > localMax) {
    handler = ...;
}
update = work();
if (local > localMax) {
    handler(update);
}
```

[Boehm, Hotpar 2011]

- Data race freedom allows extra reads.

```
local = counter;
if (local > localMax) {
    handler = ...;
}
update = work();
if (counter > localMax) {
    handler(update);
}
```

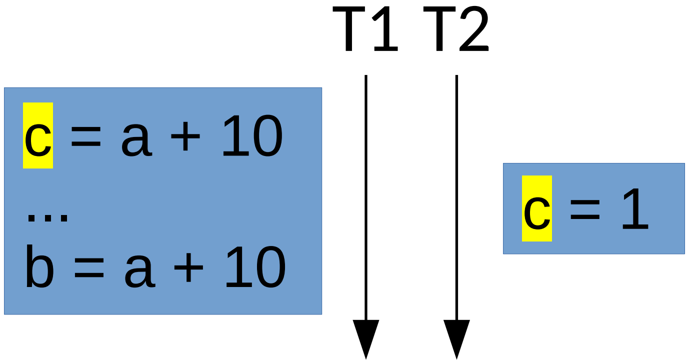
False

True

Other Thread

counter = ...;

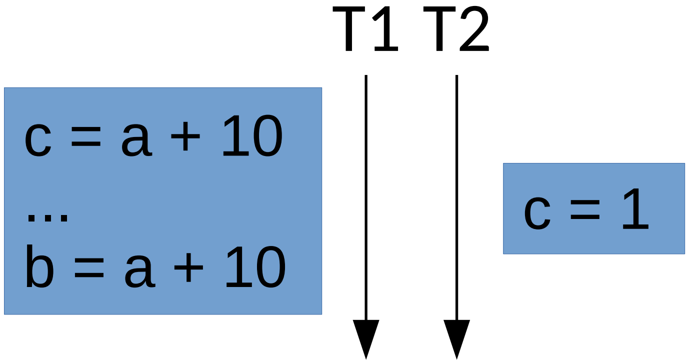
“Benign” Data Races



[Dolan, PLDI 2018]

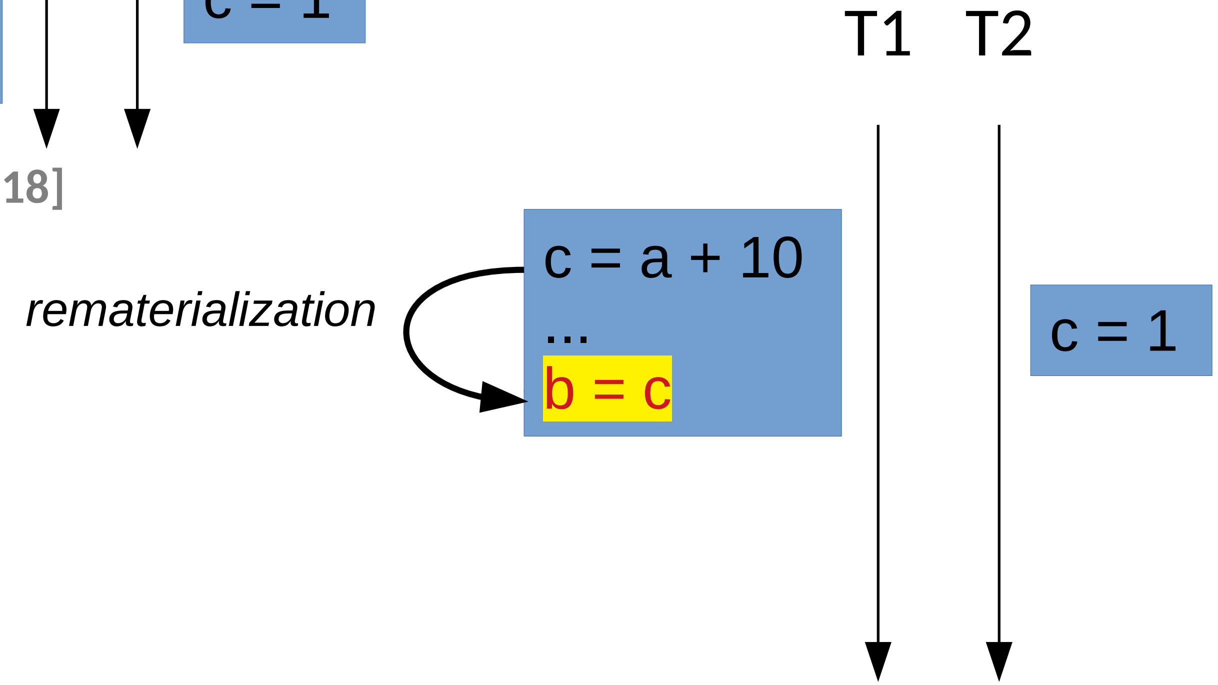
- Races can introduce bugs on *non-racy variables*

“Benign” Data Races



[Dolan, PLDI 2018]

- Races can introduce bugs on non-racy variables



“Benign” Data Races

T1 T2

```
a = 1
flag = true
```

```
a = 2
f = flag
b = a
c = a
```

- Races can jump forward and backward in time

[Dolan, PLDI 2018]

“Benign” Data Races

T1 T2

a = 1
flag = true

a = 2
f = flag
b = a
c = a

[Dolan, PLDI 2018]

- Races can jump forward and backward in time

This can happen in Java when flag is volatile
& b is a complex reference

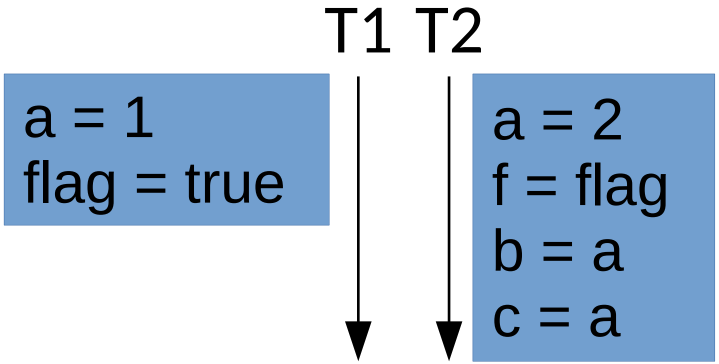
T1 T2

a = 1
flag = true

a = 2

f = flag
b = a
c = 2

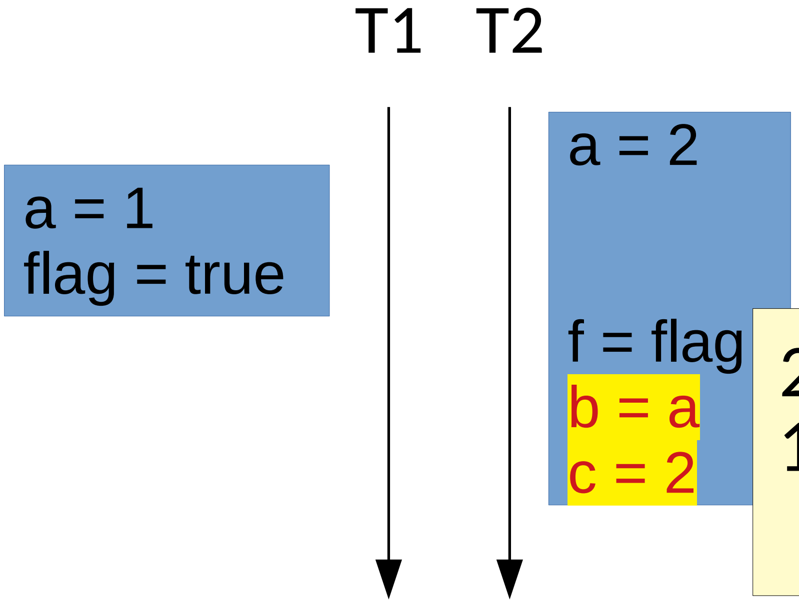
“Benign” Data Races



[Dolan, PLDI 2018]

- Races can jump forward and backward in time

This can happen in Java when flag is volatile
& b is a complex reference



2 can be read after 1 even in the same thread!

Happens-Before Ordering

- Memory models are often specified using ***Happens-Before*** relations.

Happens-Before Ordering

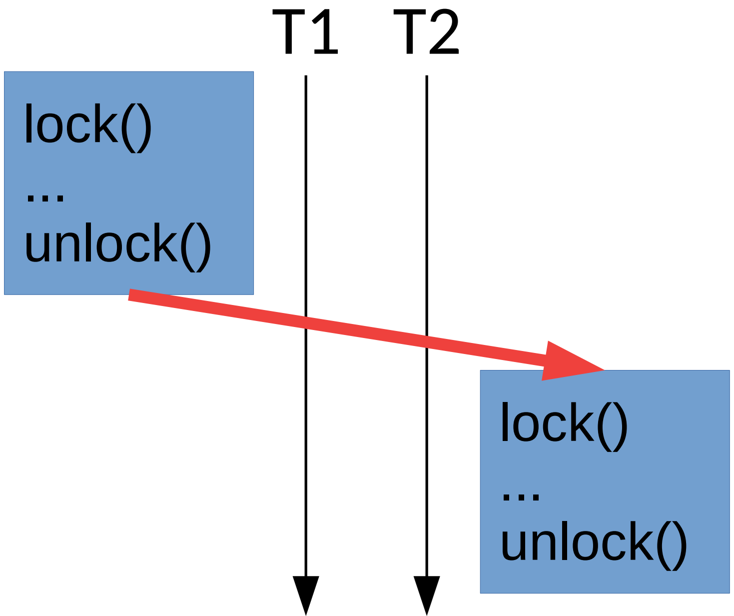
- Memory models are often specified using Happens-Before relations.
  - a partial order over logical time (recall: *simultaneously*)
  - defined behavior occurs when writes & reads are ordered

Happens-Before Ordering

- Memory models are often specified using Happens-Before relations.
  - a partial order over logical time (recall: *simultaneously*)
  - defined behavior occurs when writes & reads are ordered
  - lock/unlock, fork/join constrain order
  - access to volatile variables keeps *per individual variable* order

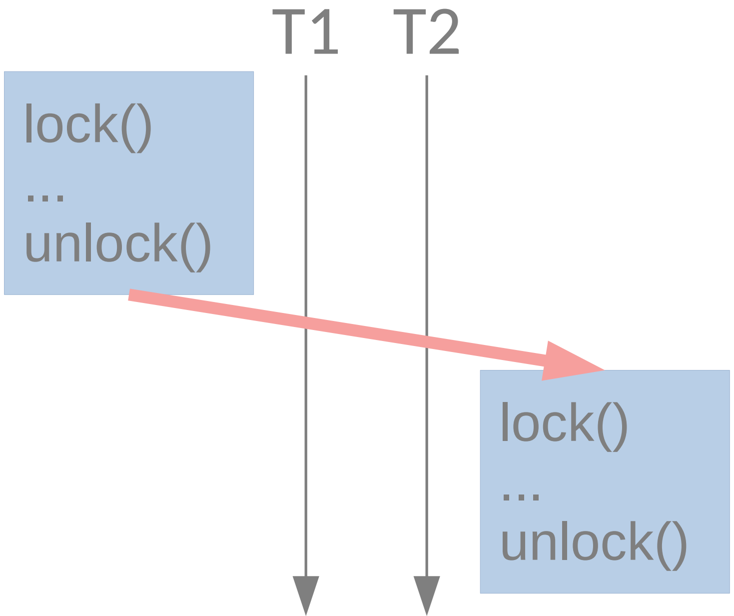
Happens-Before Ordering

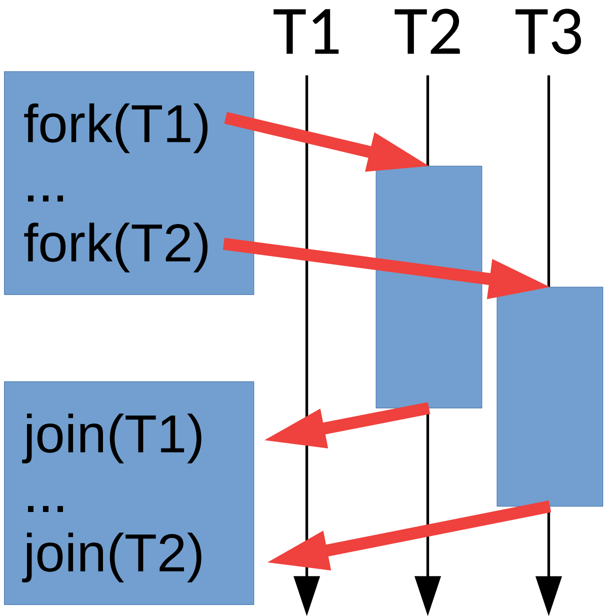
- Memory models are often specified using Happens-Before relations.
  - a partial order over logical time (recall: *simultaneously*)
  - defined behavior occurs when writes & reads are ordered
  - lock/unlock, fork/join constrain order
  - access to volatile variables keeps *per individual variable* order



Happens-Before Ordering

- Memory models are often specified using Happens-Before relations.
  - a partial order over logical time (recall: *simultaneously*)
  - defined behavior occurs when writes & reads are ordered
  - lock/unlock, fork/join constrain order
  - access to volatile variables keeps *per individual variable* order

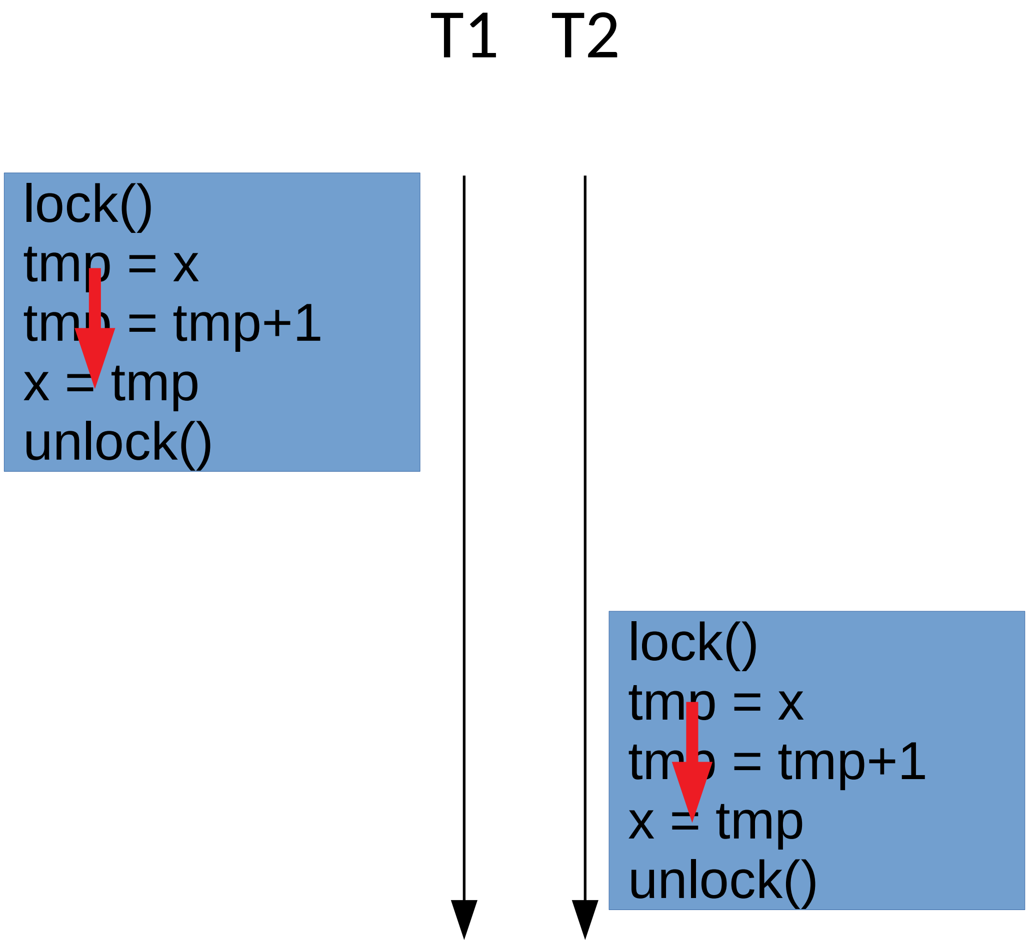




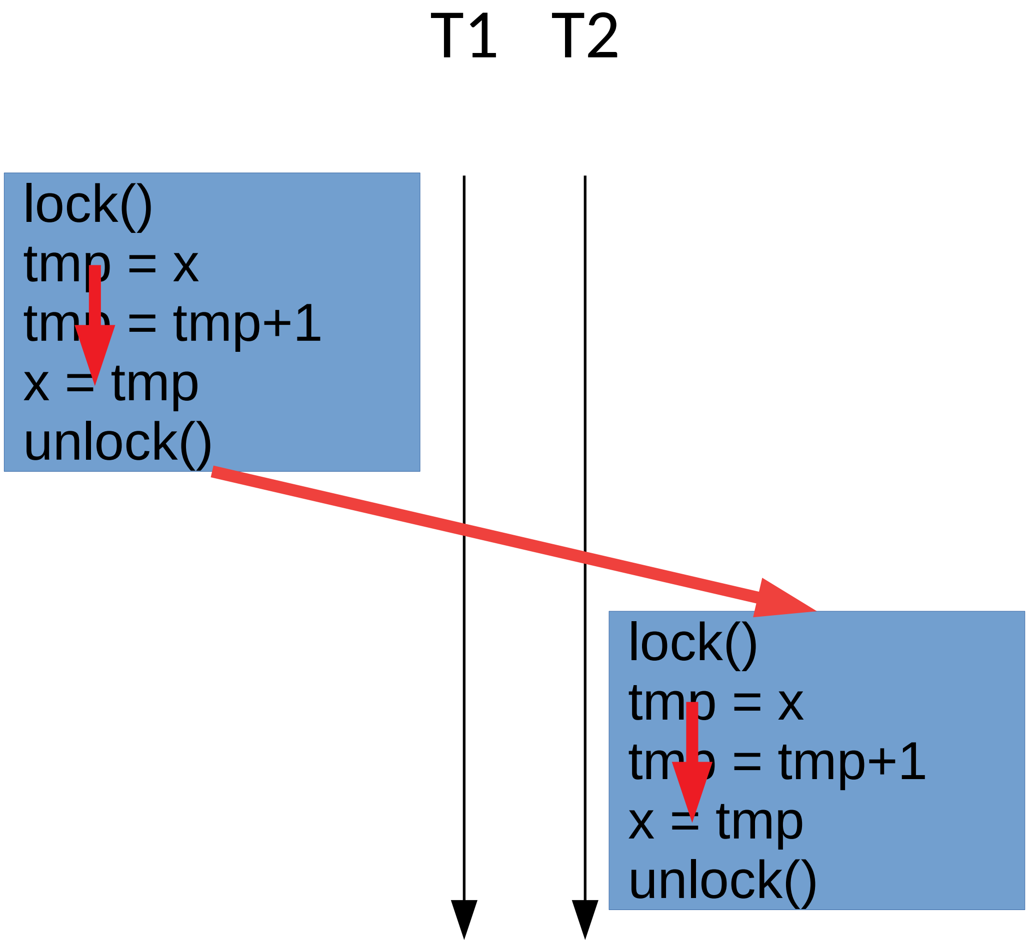
Happens-Before Ordering

- Memory models are often specified using Happens-Before relations.
  - a partial order over logical time (recall: *simultaneously*)
  - defined behavior occurs when writes & reads are ordered
  - lock/unlock, fork/join constrain order
  - access to volatile variables keeps per individual variable order
- Happens-Before ordering of a specific execution can be tracked to identify bugs

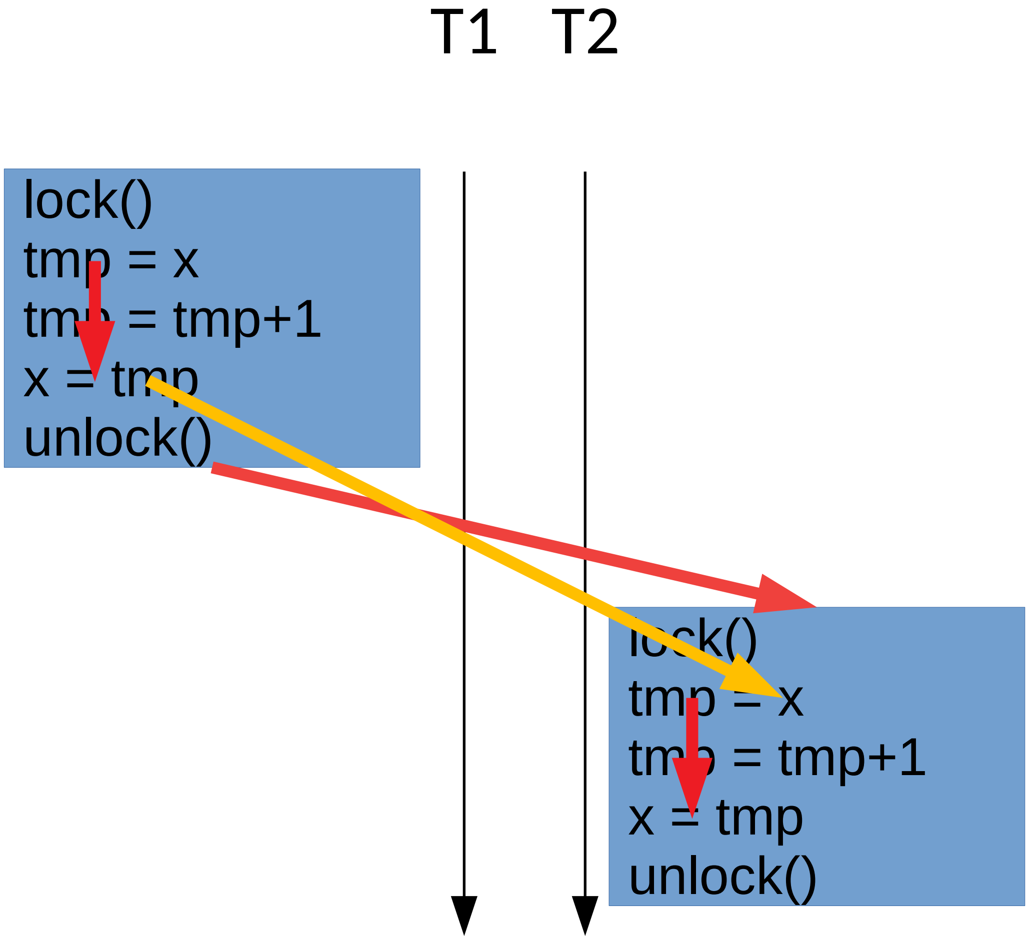
Happens-Before Ordering



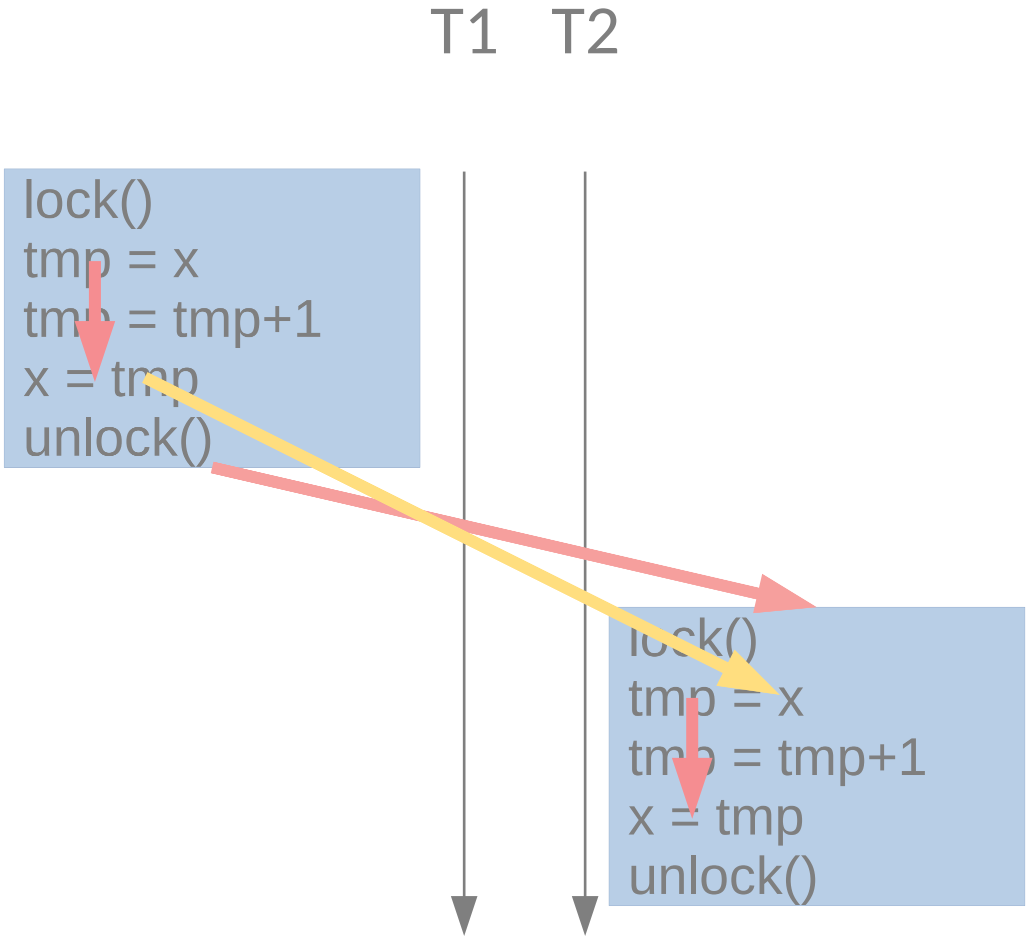
Happens-Before Ordering

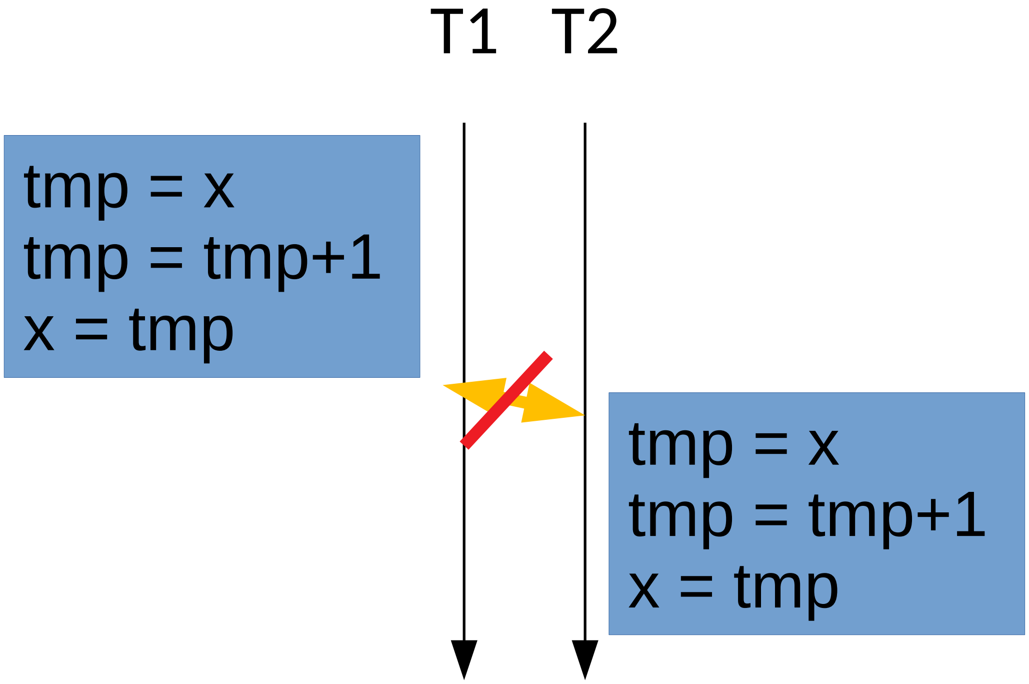


Happens-Before Ordering

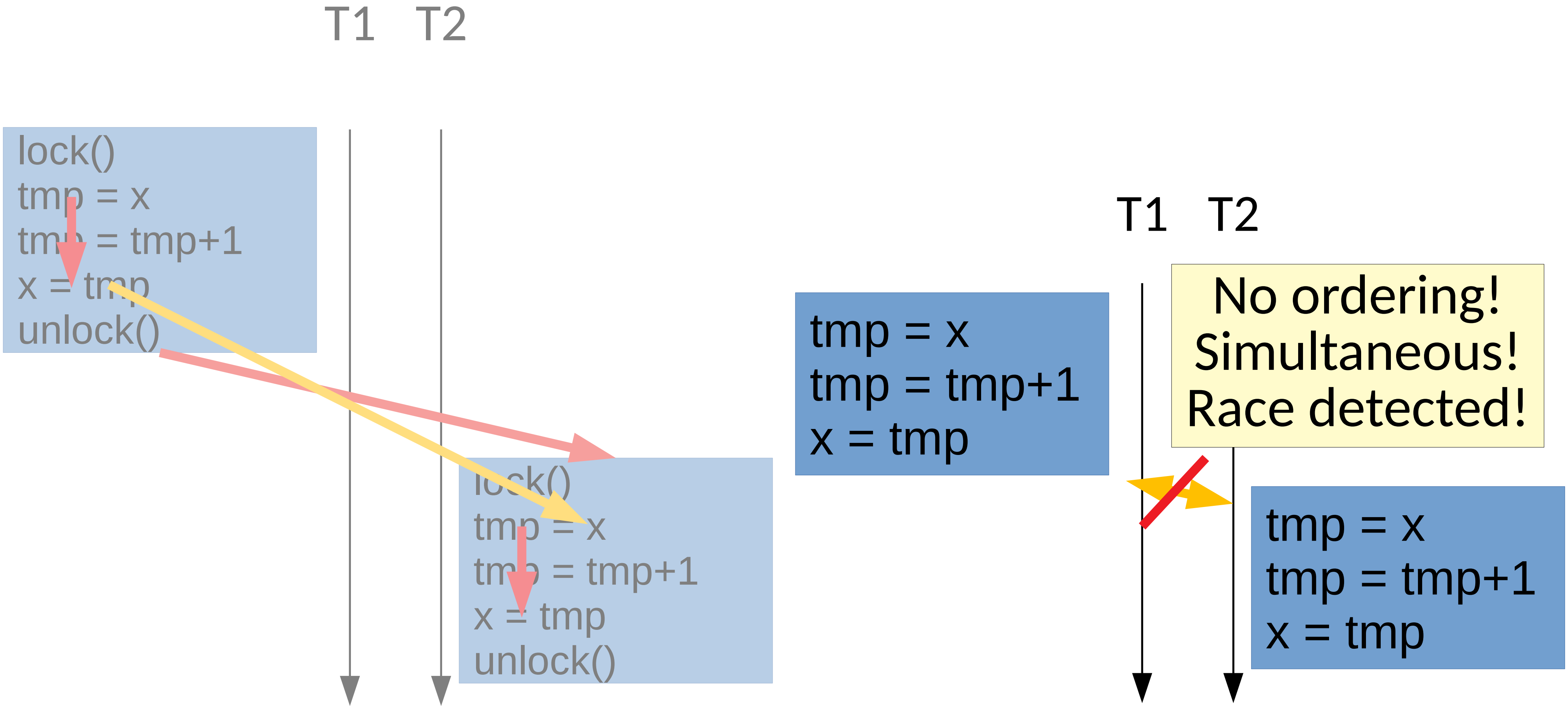


Happens-Before Ordering





Happens-Before Ordering



Happens-Before Ordering

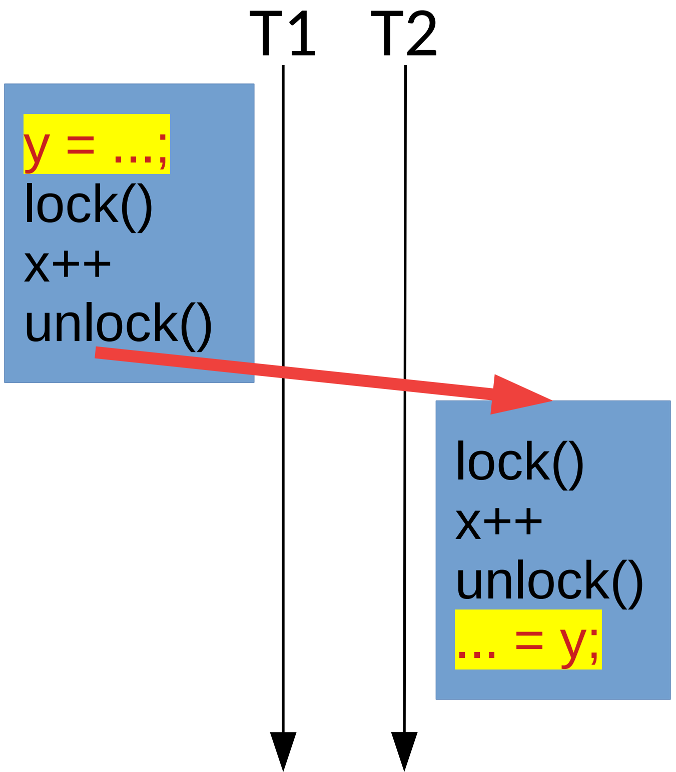
- Note, this only detects races in the current execution & schedule!

Why?

Happens-Before Ordering

- Note, this only detects races in the current execution & schedule!

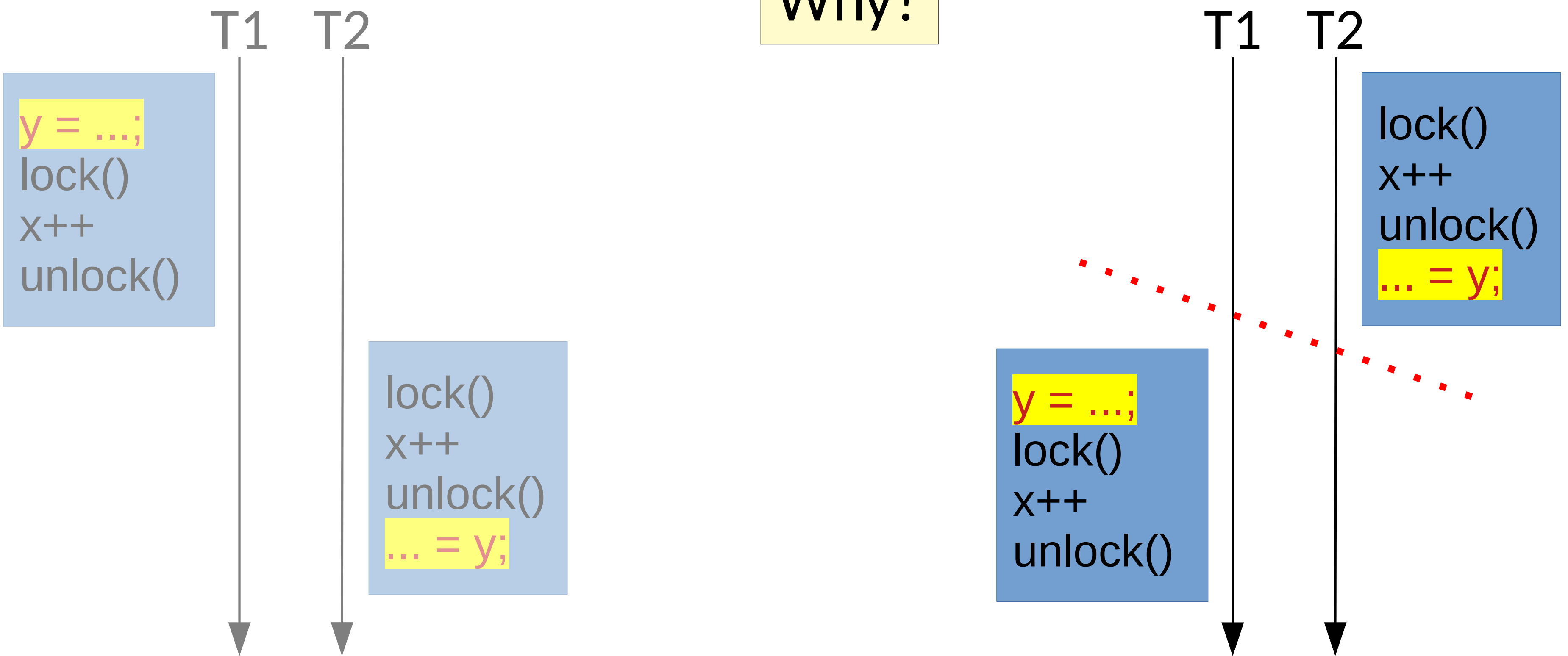
Why?



Happens-Before Ordering

- Note, this only detects races in the current execution & schedule!

Why?



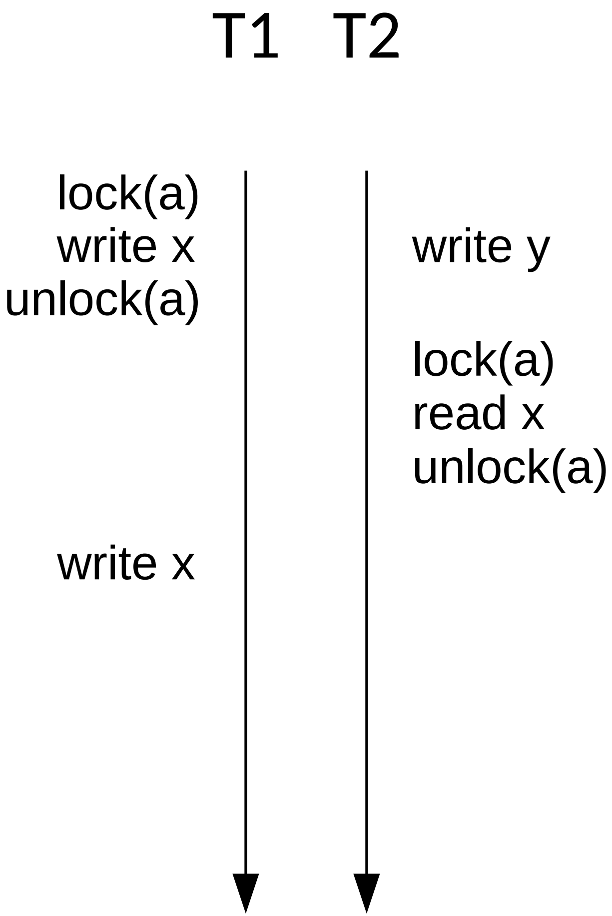
Happens-Before Ordering

- Note, this only detects races in the current execution & schedule!
  - *Sound predictive* data race detection can extend it *across other executions* **[PLDI 2017/2018, 2020]**

Happens-Before Ordering

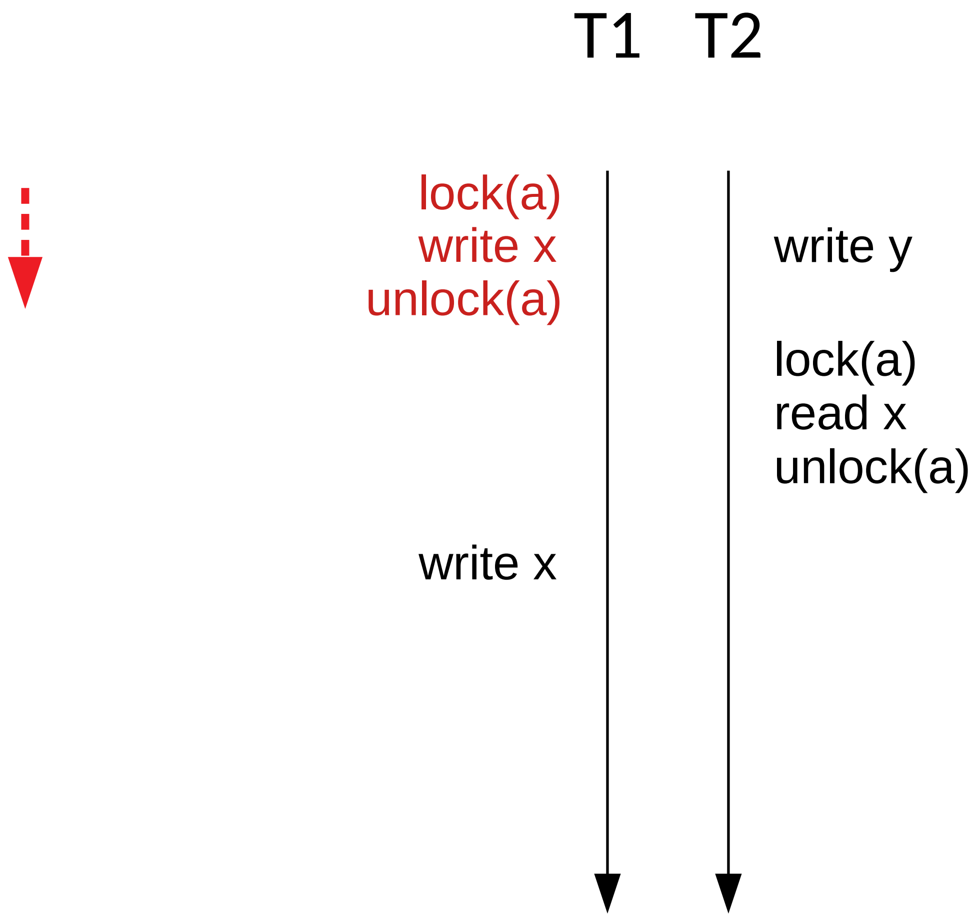
- Note, this only detects races in the current execution & schedule!
  - *Sound predictive* data race detection can extend it across other executions **[PLDI 2017/2018, 2020]**
- Requires careful tracking of dependences
  - Careful construction of logical time using ***vector clocks*** **[JVM 2001, PLDI 2009]**

Logical Time & Vector Clocks



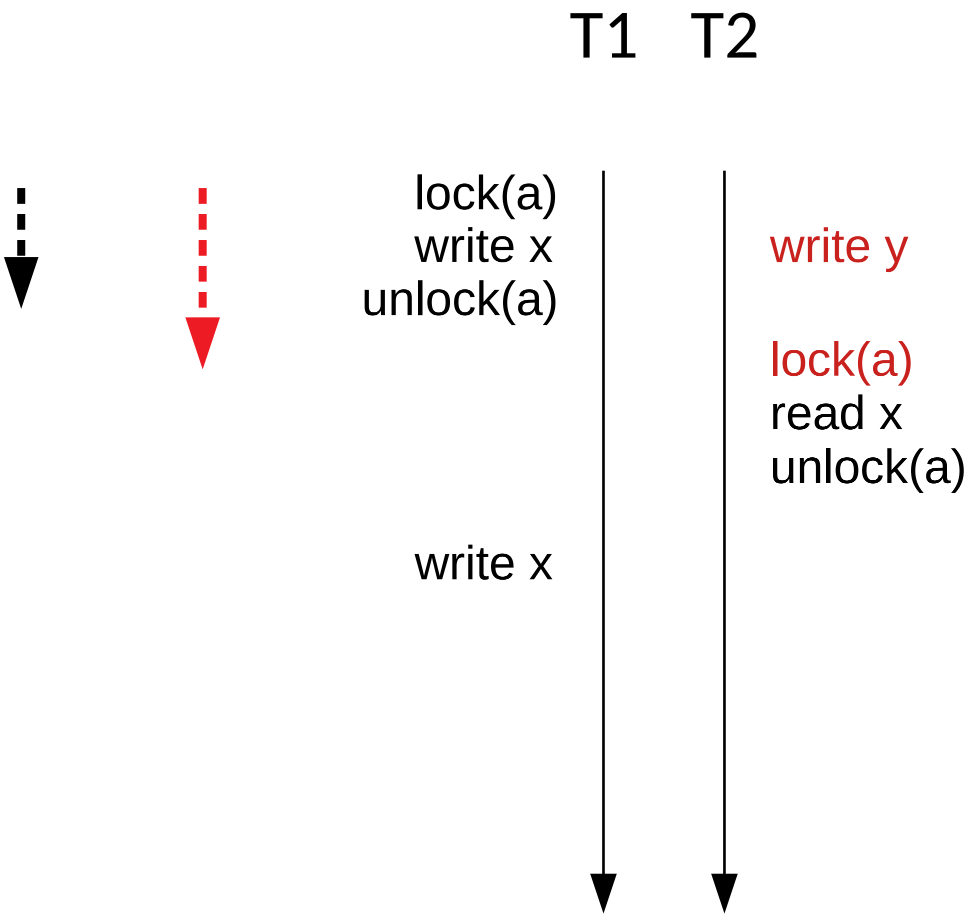
What is the happens-before relation?

Logical Time & Vector Clocks



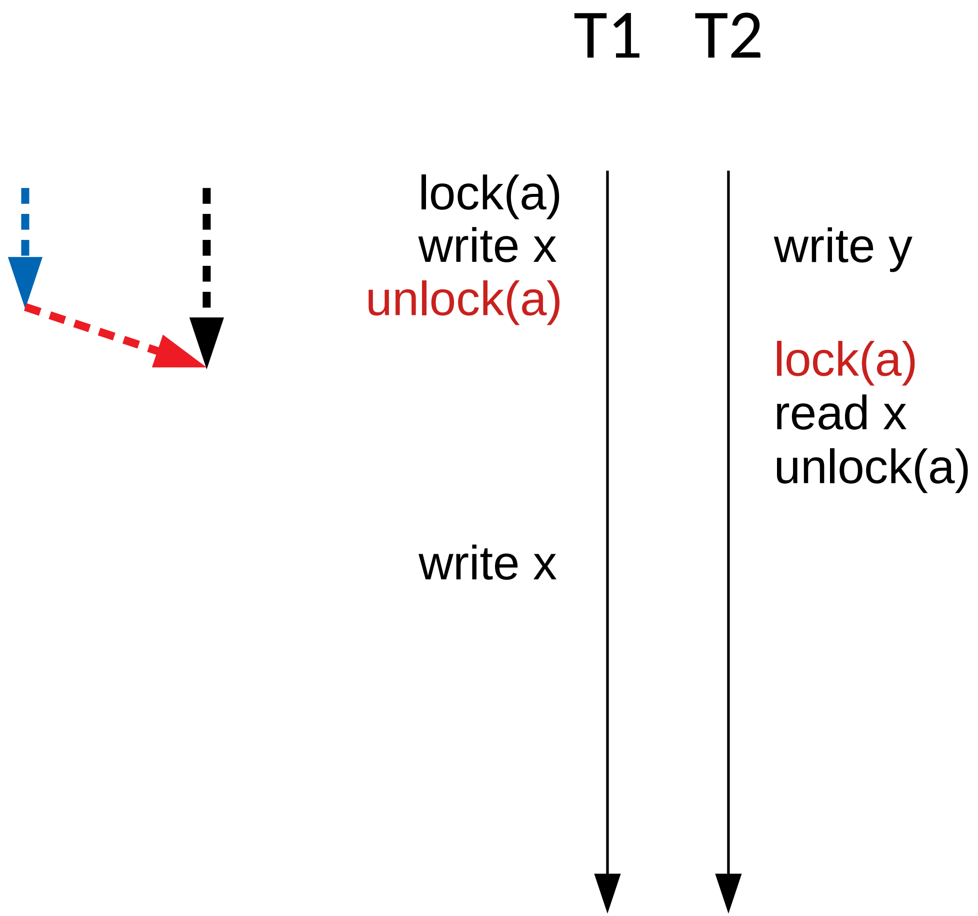
What is the happens-before relation?

Logical Time & Vector Clocks



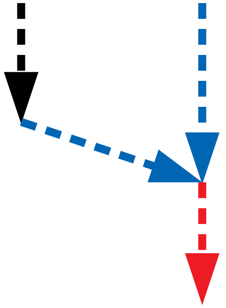
What is the happens-before relation?

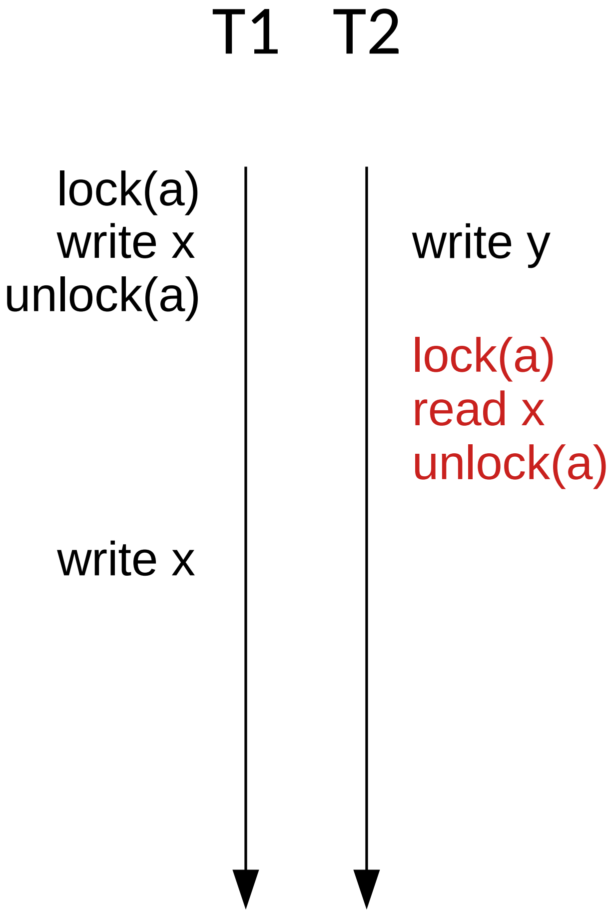
Logical Time & Vector Clocks



What is the happens-before relation?

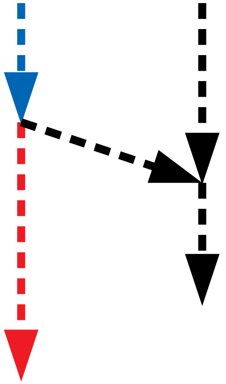
Logical Time & Vector Clocks

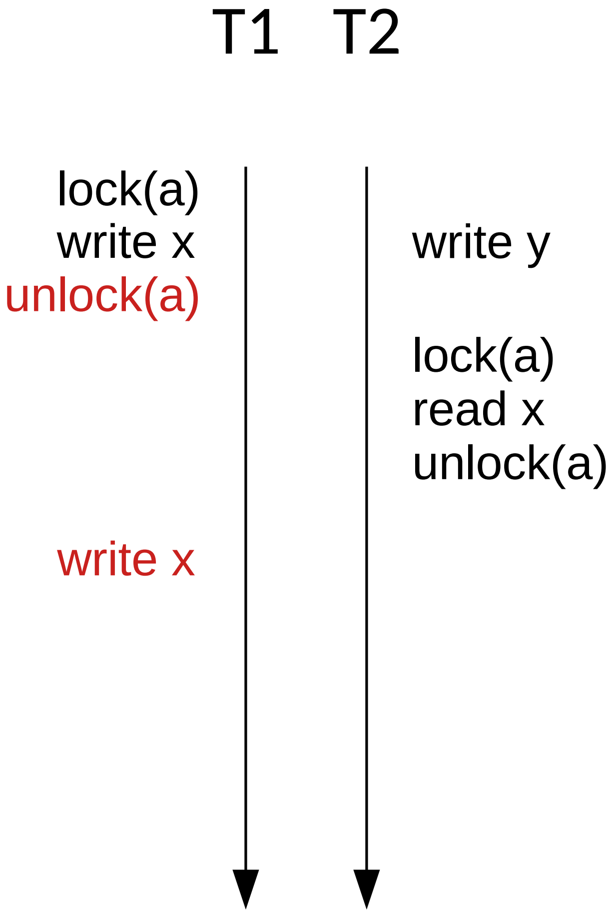




What is the happens-before relation?

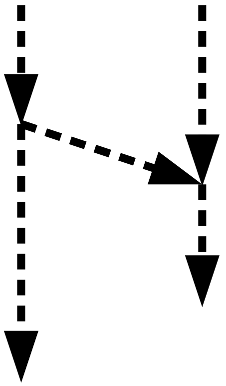
Logical Time & Vector Clocks

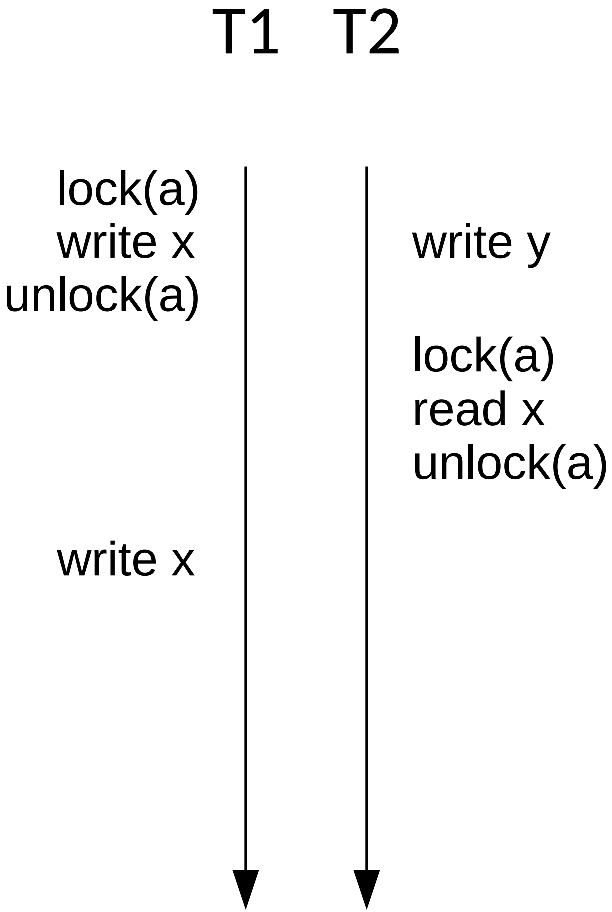




What is the happens-before relation?

Logical Time & Vector Clocks





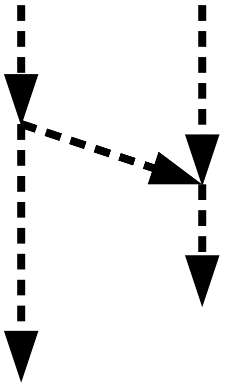
Clocks

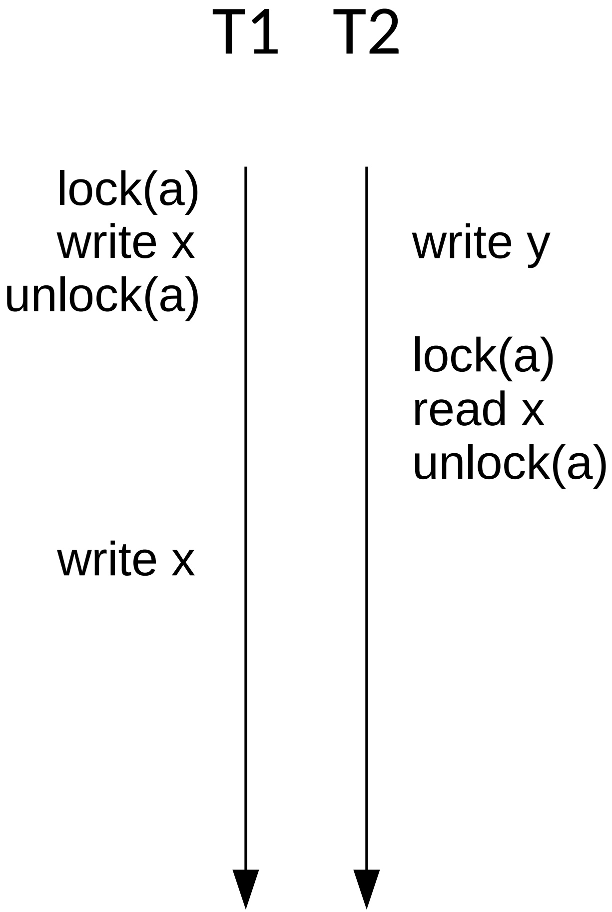
T1↦[0,0]
T2↦[0,0]

Shadow Memory

x↦ R[0,0]
W[0,0]
y↦ R[0,0]
W[0,0]

Logical Time & Vector Clocks





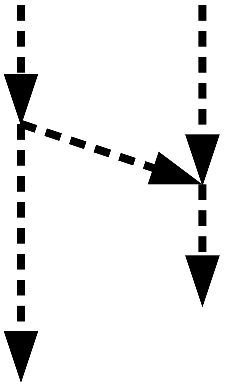
Clocks

T1↦[0,0]
T2↦[0,0]

Shadow Memory

x↦ R[0,0]
W[0,0]
y↦ R[0,0]
W[0,0]

Logical Time & Vector Clocks



	T1	T2
lock(a) write x unlock(a)		
		write y
		lock(a) read x unlock(a)
write x		

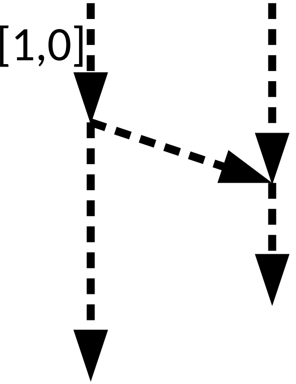
Clocks

```
T1↦[0,0]
T2↦[0,0]
```

Shadow Memory

```
x↦ R[0,0]
   W[0,0]
y↦ R[0,0]
   W[0,0]
```

Logical Time & Vector Clocks



	T1	T2
lock(a)		
write x		write y
unlock(a)		
		lock(a)
		read x
		unlock(a)
write x		

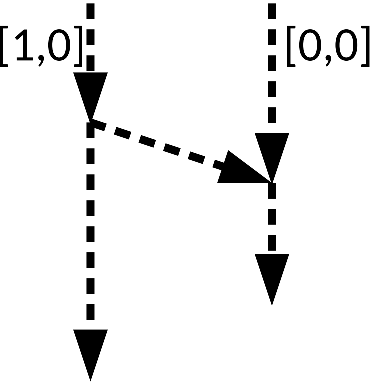
Clocks

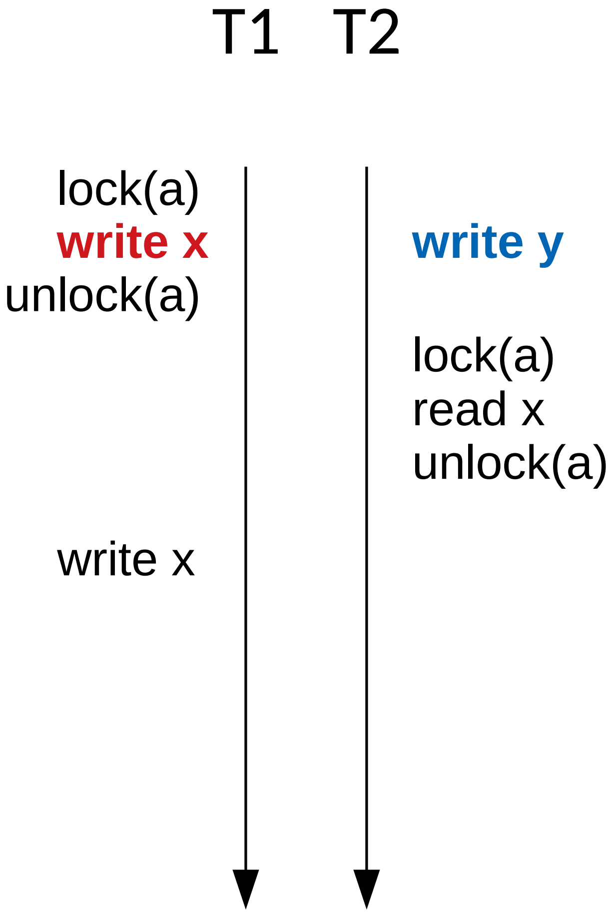
T1↦[**1**,0]
T2↦[0,0]

Shadow Memory

x↦ R[0,0]
 W[0,0]
y↦ R[0,0]
 W[0,0]

Logical Time & Vector Clocks





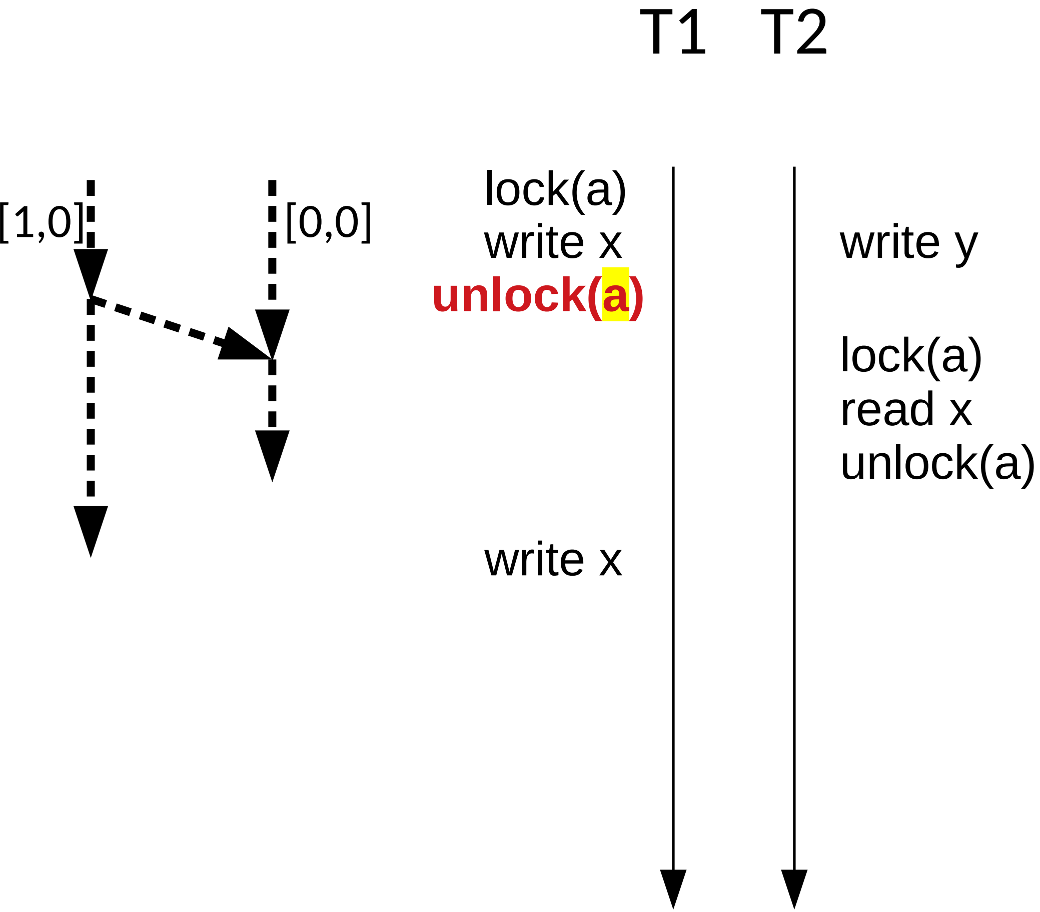
Clocks

T1↦[1,0]
T2↦[0,0]

Shadow Memory

x↦ R[0,0]
W[1,0]
y↦ R[0,0]
W[0,0]

Logical Time & Vector Clocks



Clocks

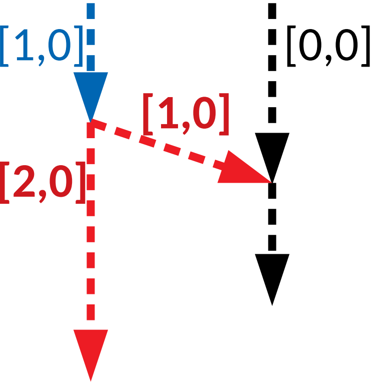
T1↦[2,0]
T2↦[0,0]

a↦[1,0]

Shadow Memory

x↦ R[0,0]
W[1,0]
y↦ R[0,0]
W[0,0]

Logical Time & Vector Clocks



T1	T2
lock(a)	
write x	write y
unlock(a)	
	lock(a)
	read x
	unlock(a)
write x	

Clocks

```
T1↦[2,0]
T2↦[0,0]
```

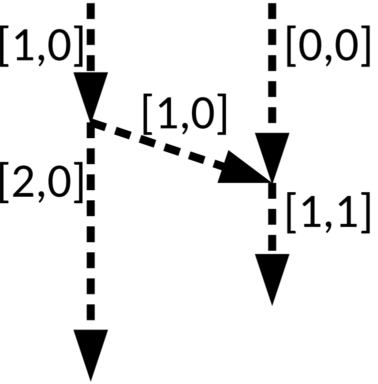
```
a↦[1,0]
```

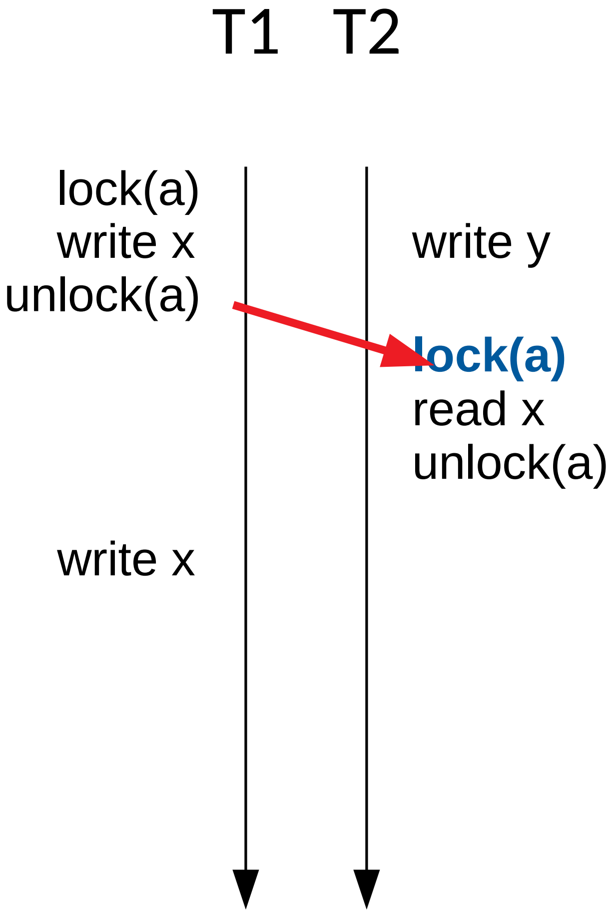
Shadow Memory

```
x↦ R[0,0]
   W[1,0]
y↦ R[0,0]
   W[0,0]
```

```
sendProgress(T1, a)
advanceLocal(T1)
```

Logical Time & Vector Clocks





Clocks

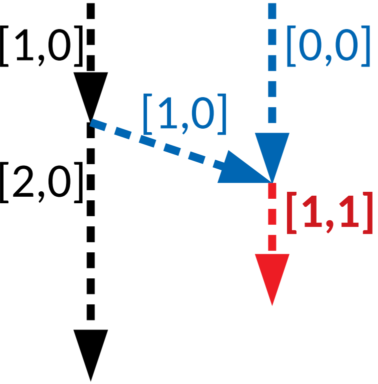
T1↦[2,0]
T2↦[1,1]

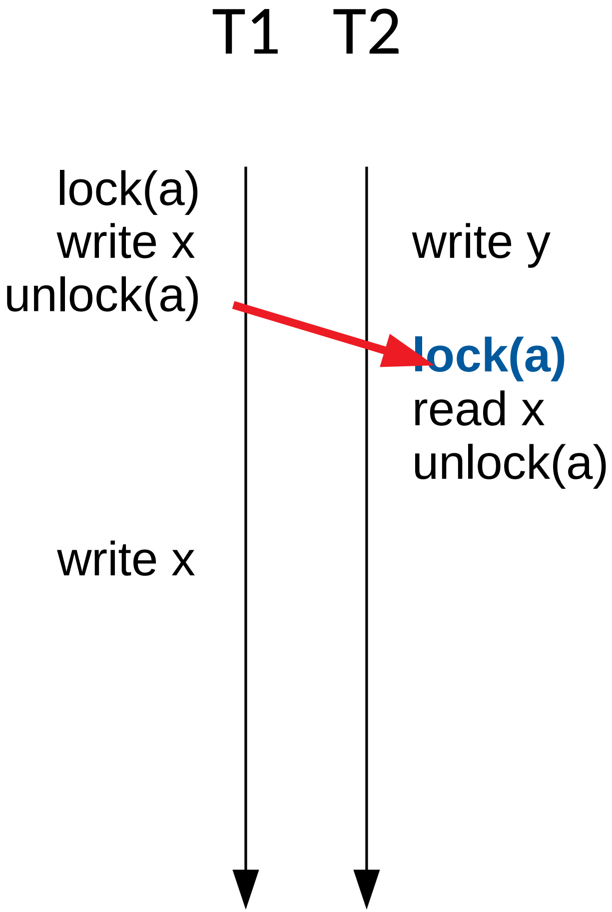
a↦[1,0]

Shadow Memory

x↦ R[0,0]
W[1,0]
y↦ R[0,0]
W[0,0]

Logical Time & Vector Clocks





Clocks

T1↦[2,0]
T2↦[1,1]

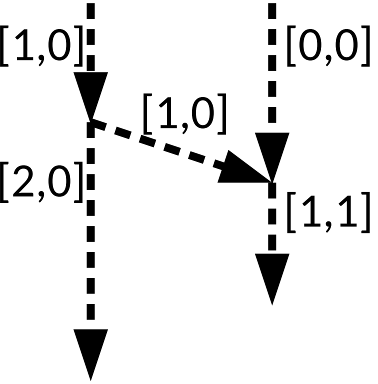
a↦[1,0]

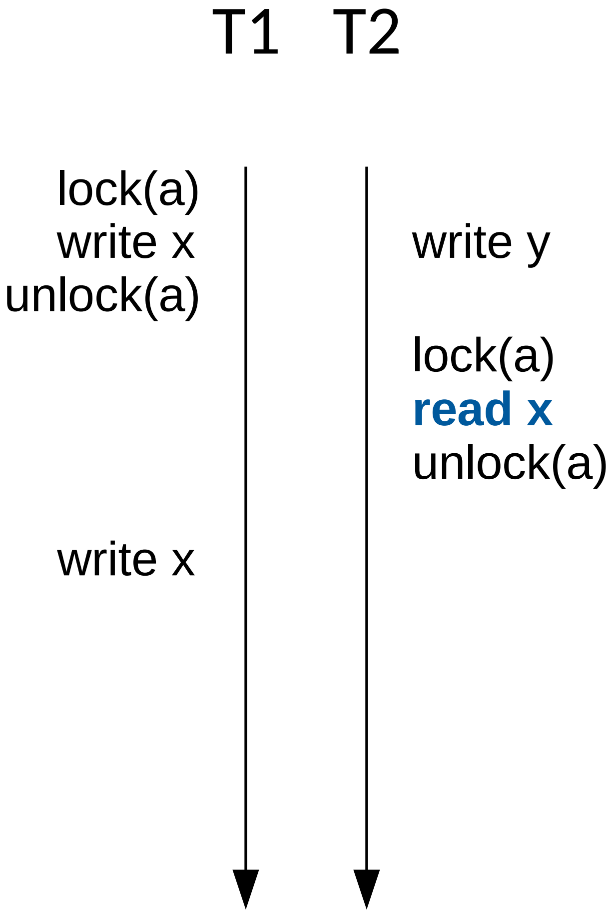
Shadow Memory

x↦ R[0,0]
W[1,0]
y↦ R[0,0]
W[0,0]

```
receiveProgress(T2, a)
advanceLocal(T2)
```

Logical Time & Vector Clocks





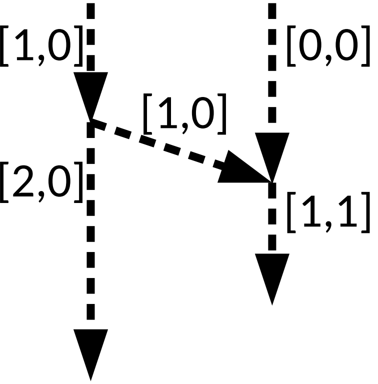
Clocks

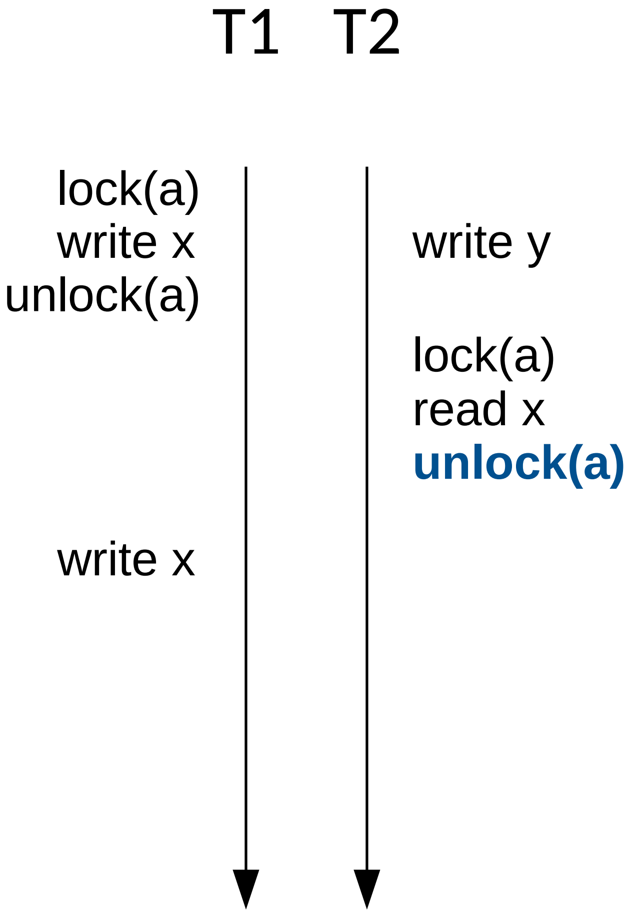
```
T1↦[2,0]
T2↦[1,1]
```

Shadow Memory

```
x↦ R[1,1]
   W[1,0]
y↦ R[0,0]
   W[0,0]
```

Logical Time & Vector Clocks





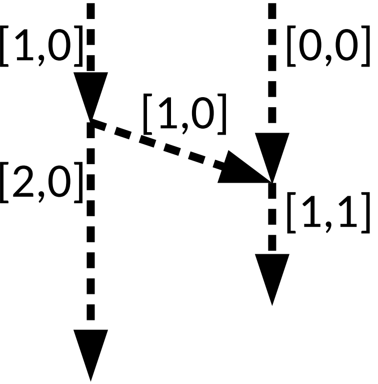
Clocks

T1↦[2,0]
T2↦[1,2]

Shadow Memory

x↦ R[1,1]
W[1,0]
y↦ R[0,0]
W[0,0]

Logical Time & Vector Clocks



T1	T2
lock(a) write x unlock(a)	
	write y
	lock(a) read x unlock(a)
write x	

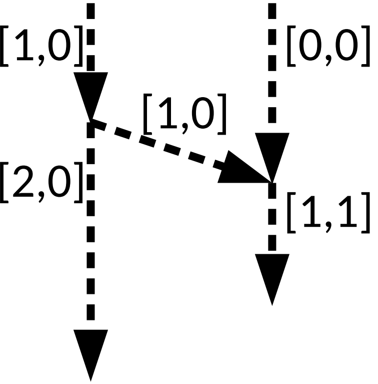
Clocks

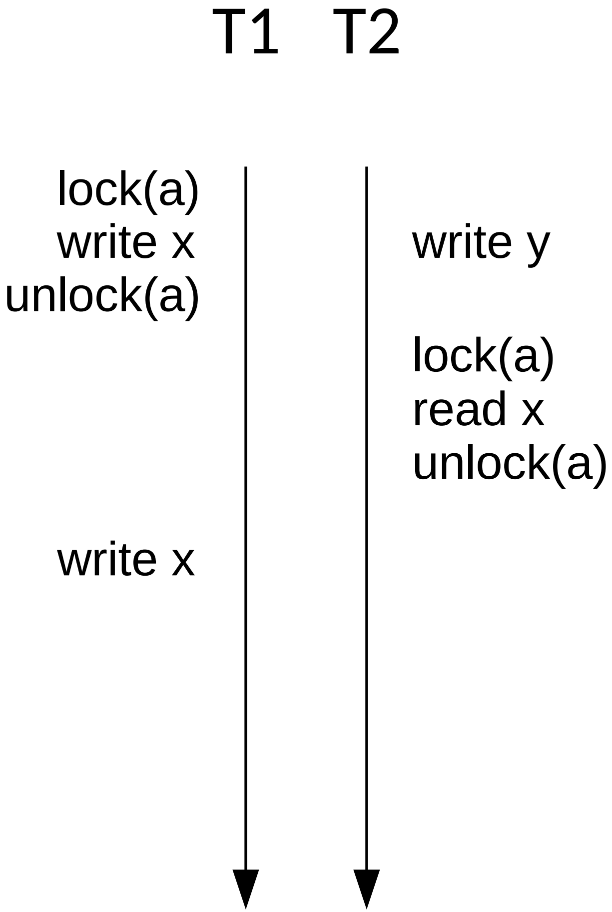
T1↦[2,0]
T2↦[1,2]

Shadow Memory

x↦ R[1,1]
W[2,0]
y↦ R[0,0]
W[0,0]

Logical Time & Vector Clocks





Clocks

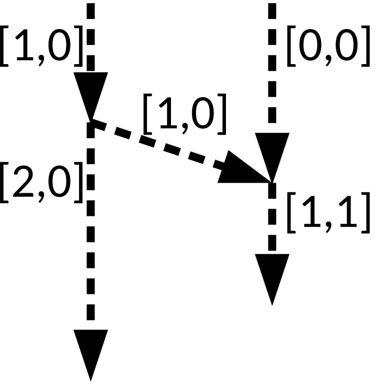
T1↦[2,0]
T2↦[1,2]

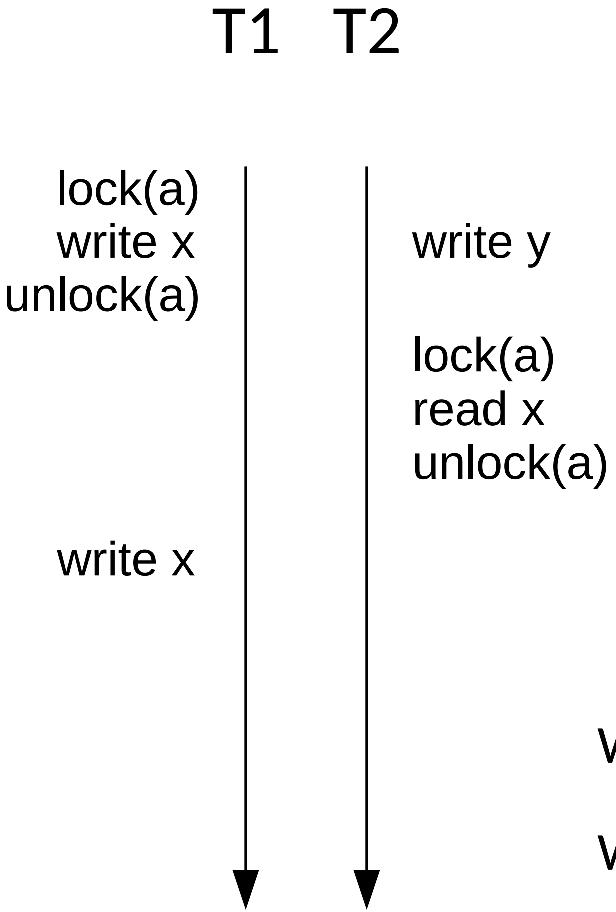
Shadow Memory

x↦ R[1,1]
W[2,0]
y↦ R[0,0]
W[0,0]

C(R,X) ⋢ C(W,X)
& C(W,X) ⋢ C(R,X) → race!

Logical Time & Vector Clocks





Clocks

T1↦[2,0]
T2↦[1,2]

Shadow Memory

x↦ R[1,1]
W[2,0]
y↦ R[0,0]
W[0,0]

$C(R,X) \not\sqsubseteq C(W,X)$
$\& \; C(W,X) \not\sqsubseteq C(R,X) \rightarrow$ race!

When reading:
check against the last write
When writing:
check against the last read and write

Vector Clocks

- Vector clocks have a long history in reasoning about distributed systems and concurrency

Vector Clocks

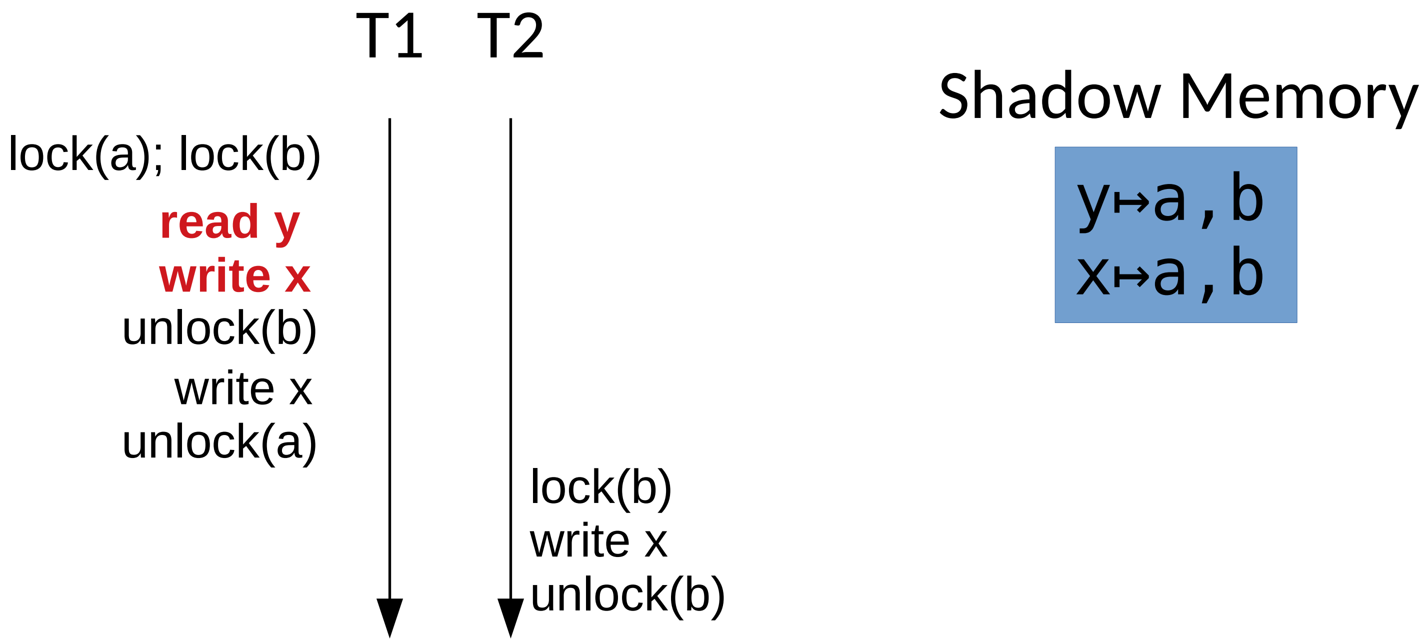
- Vector clocks have a long history in reasoning about distributed systems and concurrency
- With many threads/parallel tasks, clock overheads increase
  - With effort, newer approaches optimize and replace vector clocks [Mathur 2021]

Data Race Detection - Locksets

- Lack of synchronization arises with complex locking
- We can dynamically track the locks guarding an address!

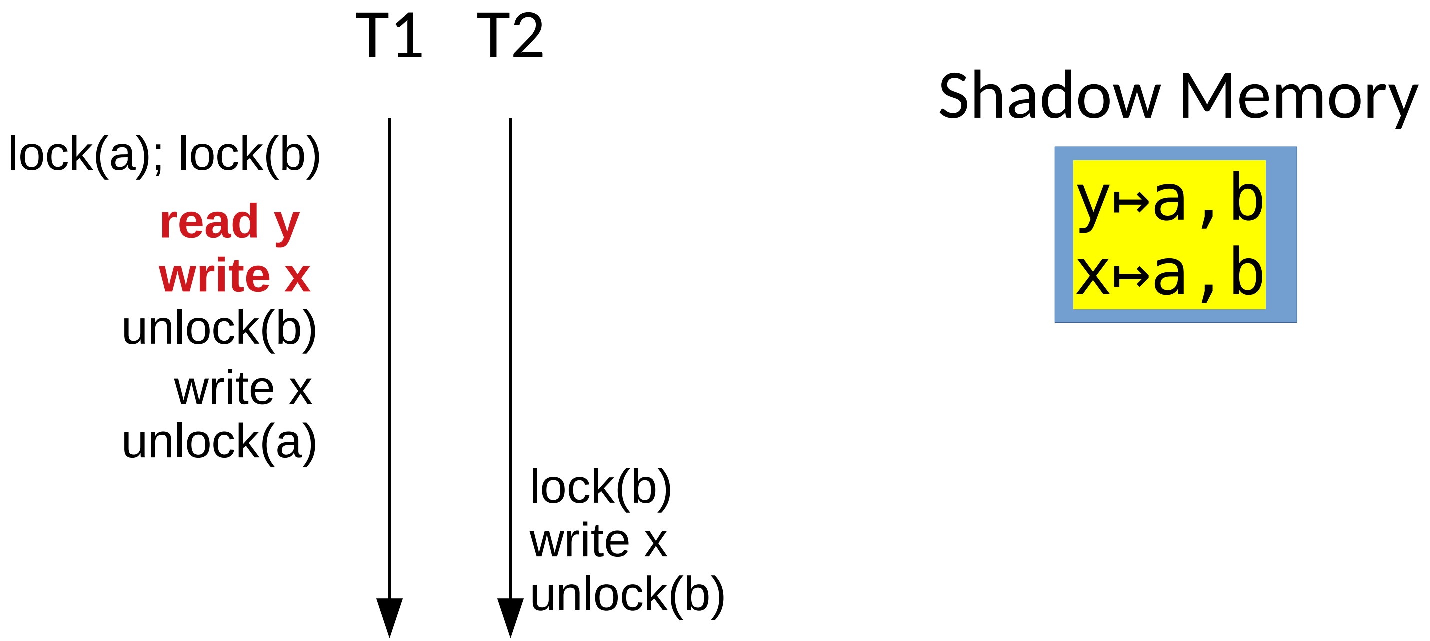
Data Race Detection - Locksets

- Lack of synchronization arises with complex locking
- We can dynamically track the locks guarding an address!



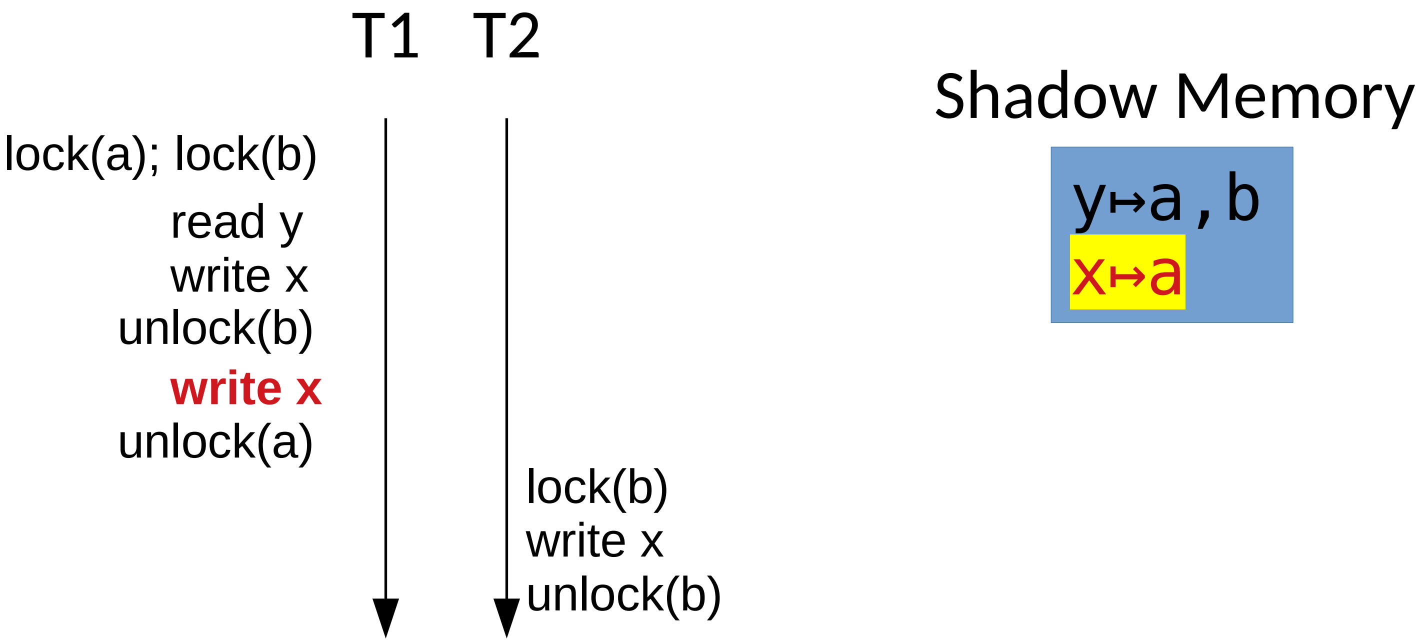
Data Race Detection - Locksets

- Lack of synchronization arises with complex locking
- We can dynamically track the locks guarding an address!



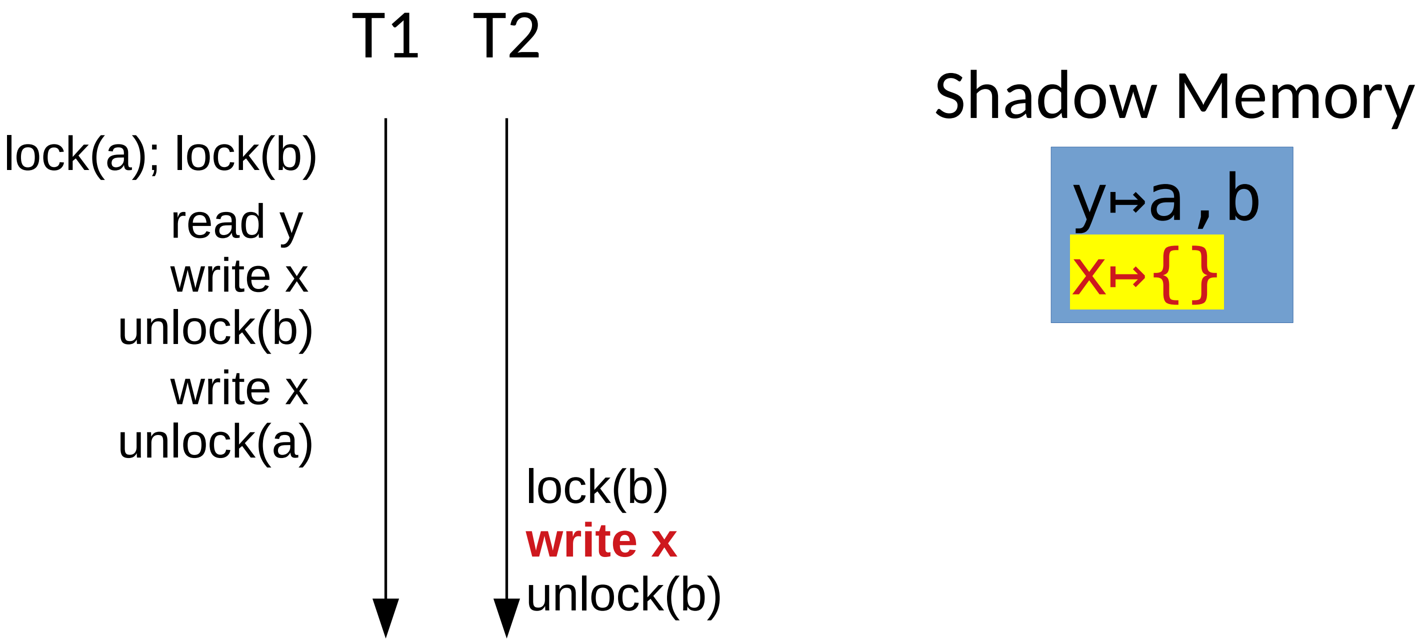
Data Race Detection - Locksets

- Lack of synchronization arises with complex locking
- We can dynamically track the locks guarding an address!



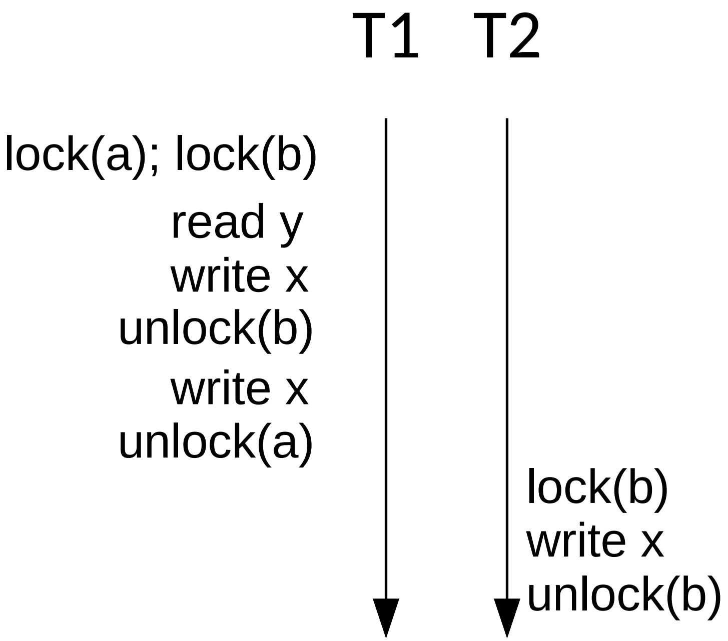
Data Race Detection - Locksets

- Lack of synchronization arises with complex locking
- We can dynamically track the locks guarding an address!



Data Race Detection - Locksets

- Lack of synchronization arises with complex locking
- We can dynamically track the locks guarding an address!



Shadow Memory

y↦a,b
x↦{}

Note: Both x and y are always protected by locks. x *still* races.

Data Race Detection - Locksets

- Lockset based data race detection has many issues
  - Synchronization may be fork/join, wait/notify based
  - Initialization --> Process in Parallel --> Combine
  - Richer parallel designs

Data Race Detection - Locksets

- Lockset based data race detection has many issues
  - Synchronization may be fork/join, wait/notify based
  - Initialization --> Process in Parallel --> Combine
  - Richer parallel designs
- Tends to have many false positives

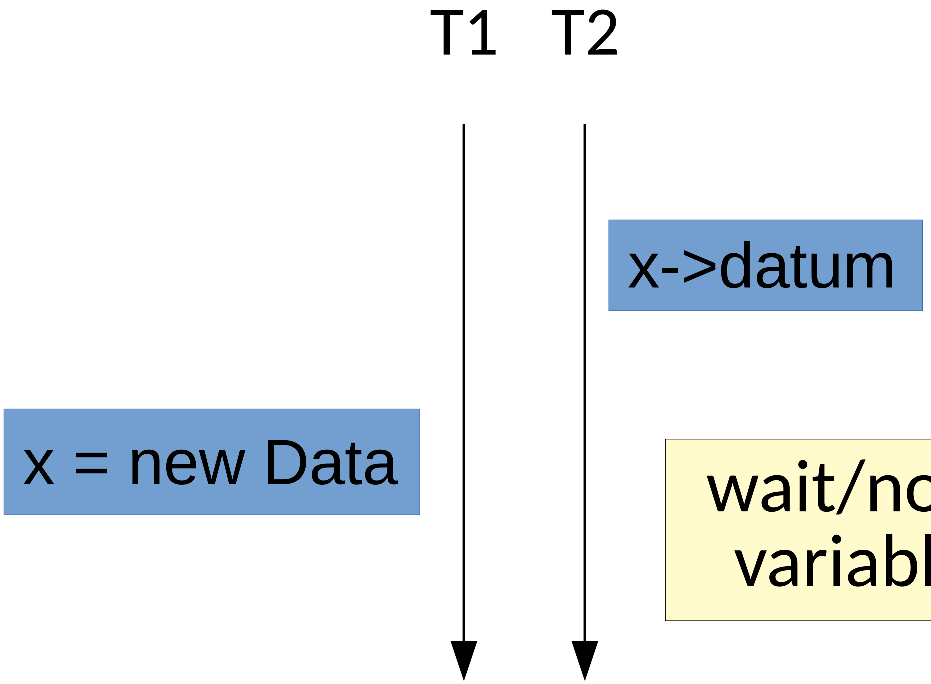
Data Race Detection - Locksets

- Lockset based data race detection has many issues
  - Synchronization may be fork/join, wait/notify based
  - Initialization --> Process in Parallel --> Combine
  - Richer parallel designs
- Tends to have many false positives

Can you think of concurrency bugs it will miss, too?

Order Violations

- Some accesses are wrongly assumed to occur before others



wait/notify or condition variables can fix these

Atomicity Violations

- Data races are a matter of perspective
  - Fine grained locking doesn't solve much.

```
tmp = x
tmp = tmp+1
x = tmp
```

Atomicity Violations

- Data races are a matter of perspective
  - Fine grained locking doesn't solve much.

```
tmp = x
tmp = tmp+1
x = tmp
```

vs

```
lock()
tmp = x
unlock()

tmp = tmp+1

lock()
x = tmp
unlock()
```

No race,
similar effect!

Atomicity Violations

- Data races are a matter of perspective
  - Fine grained locking doesn't solve much.

```
tmp = x
tmp = tmp+1
x = tmp
```

vs

```
lock()
tmp = x
unlock()

tmp = tmp+1

lock()
x = tmp
unlock()
```

What do we really want?

Atomicity Violations

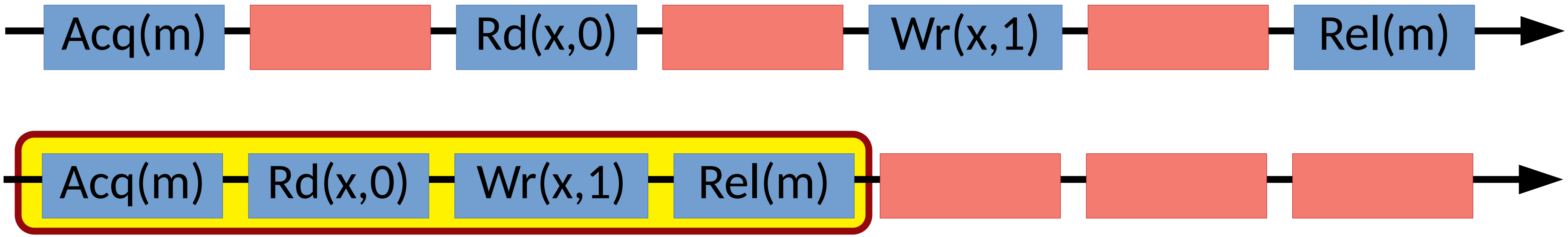
- Data races are a matter of perspective
  - Fine grained locking doesn't solve much.
- An execution (or fragment thereof) is ***atomic*** if it is equivalent to a sequentially executed one.

Atomicity Violations

- Data races are a matter of perspective
  - Fine grained locking doesn't solve much.
- An execution (or fragment thereof) is ***atomic*** if it is equivalent to a sequentially executed one.

Atomicity Violations

- Data races are a matter of perspective
  - Fine grained locking doesn't solve much.
- An execution (or fragment thereof) is ***atomic*** if it is equivalent to a sequentially executed one.



Atomicity Violations

- Data races are a matter of perspective
  - Fine grained locking doesn't solve much.
- An execution (or fragment thereof) is ***atomic*** if it is equivalent to a sequentially executed one.
  - This also takes care of data races
  - Similar to notions from databases (serializability & linearizability)

Atomicity Violations

- How can we find atomicity violations (or correctness)?

Atomicity Violations

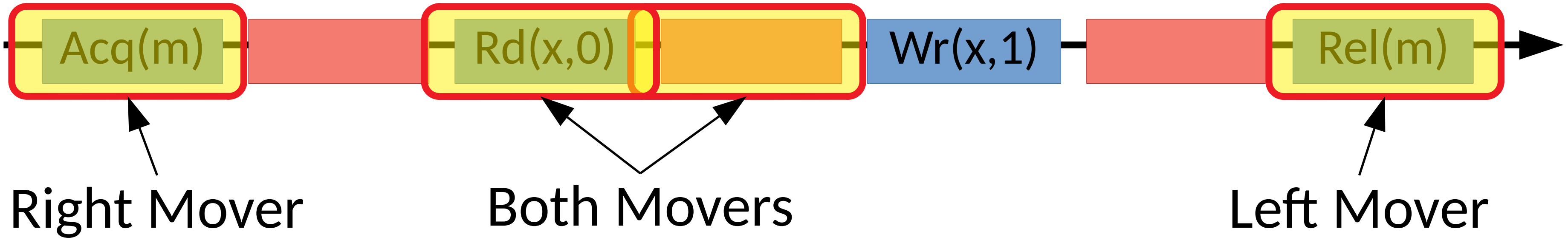
- How can we find atomicity violations (or correctness)?
  - Lipton's Theory of Reduction **[CACM '75, POPL '04]**

Atomicity Violations

- How can we find atomicity violations (or correctness)?
  - Lipton's Theory of Reduction **[CACM '75, POPL '04]**

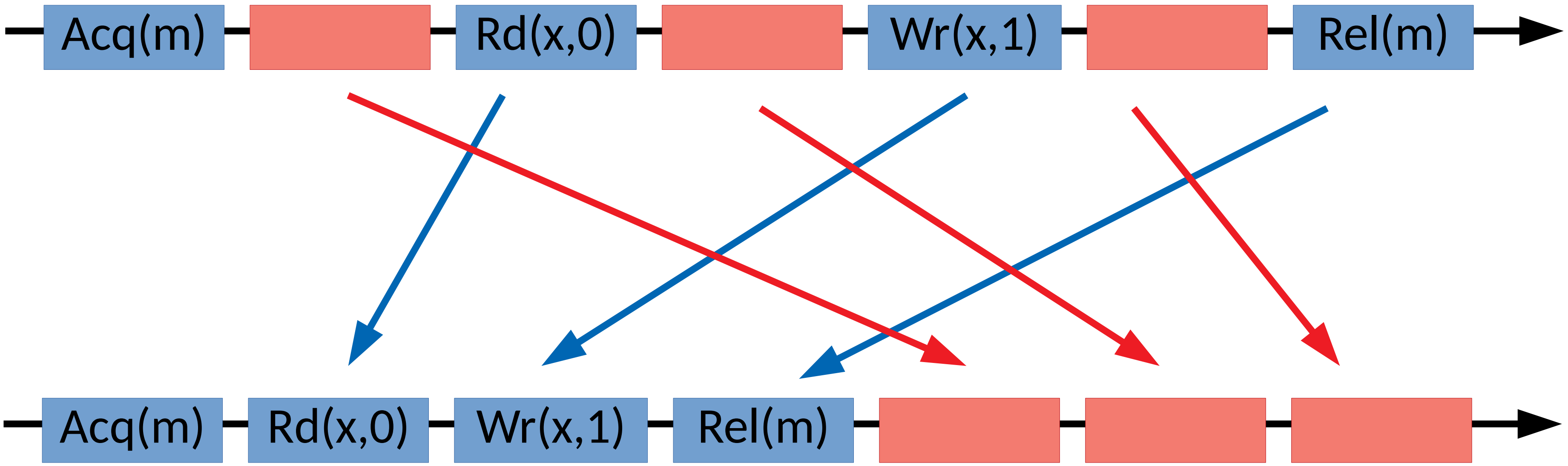
Atomicity Violations

- How can we find atomicity violations (or correctness)?
  - Lipton's Theory of Reduction [CACM '75, POPL '04]



Atomicity Violations

- How can we find atomicity violations (or correctness)?
  - Lipton's Theory of Reduction [CACM '75, POPL '04]



Atomicity Violations

- How can we find atomicity violations (or correctness)?
  - Lipton's Theory of Reduction **[CACM '75, POPL '04]**
  - 2 thread atomicity patterns **[Lu ASPLOS '06]**

Atomicity Violations

- How can we find atomicity violations (or correctness)?
  - Lipton's Theory of Reduction [CACM '75, POPL '04]
  - 2 thread atomicity patterns [Lu ASPLOS '06]

Time →

T1	T2	T1
R	R	R
W	R	R
R	W	R
W	W	R
R	R	W
W	R	W
R	W	W
W	W	W

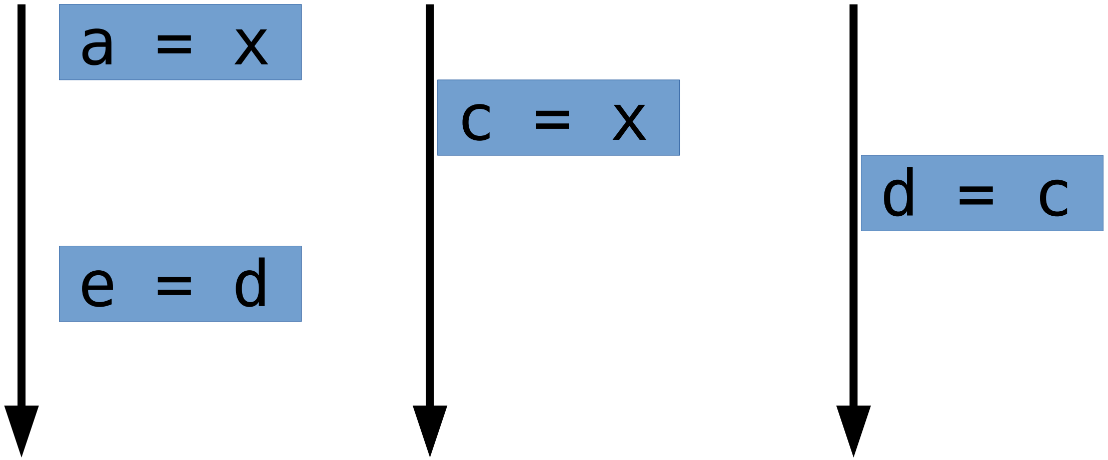
Only some patterns are unserializable.
Detect unlikely issues via training.

Atomicity Violations

- How can we find atomicity violations (or correctness)?
  - Lipton’s Theory of Reduction **[CACM ‘75, POPL ‘04]**
  - 2 thread atomicity patterns **[Lu ASPLOS ‘06]**
  - Conflict graphs **[PLDI ‘08, RV ‘11]**

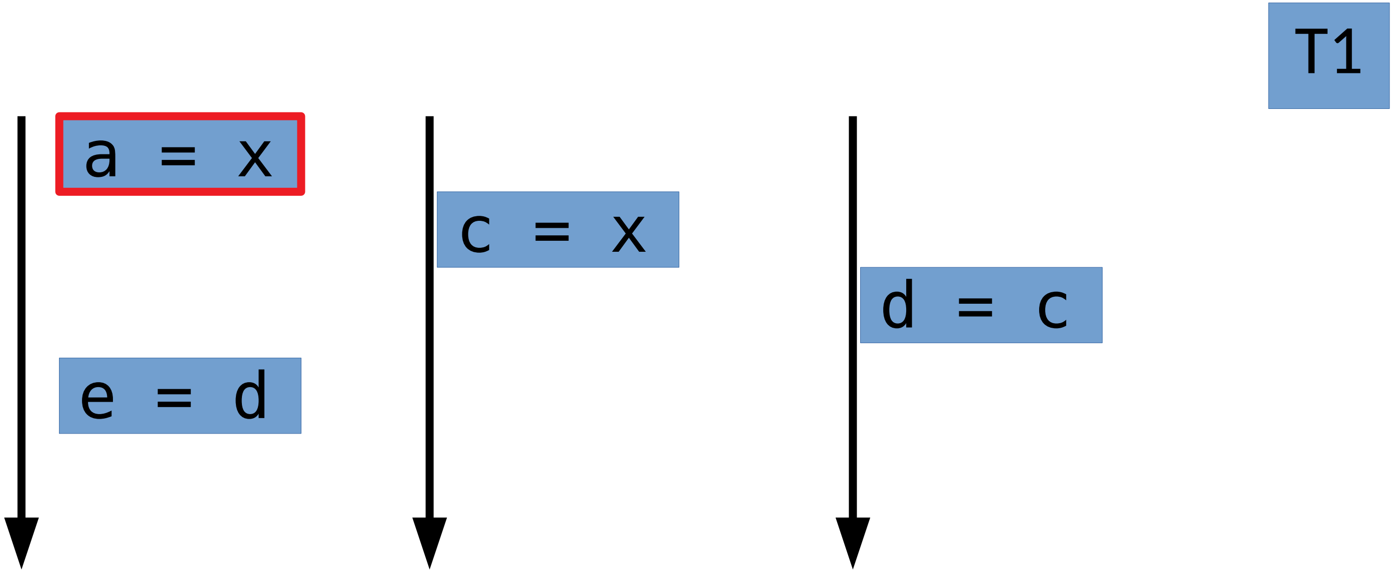
Atomicity Violations

- How can we find atomicity violations (or correctness)?
  - Lipton's Theory of Reduction [CACM '75, POPL '04]
  - 2 thread atomicity patterns [Lu ASPLOS '06]
  - Conflict graphs [PLDI '08, RV '11]



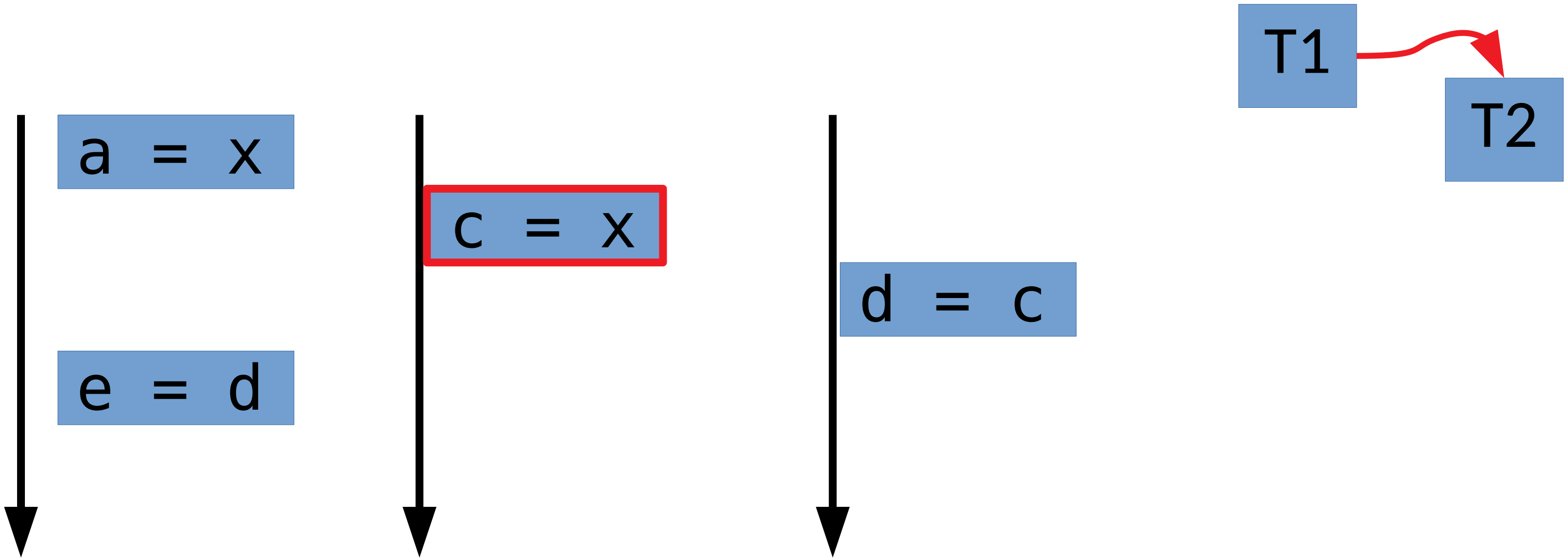
Atomicity Violations

- How can we find atomicity violations (or correctness)?
  - Lipton's Theory of Reduction [CACM '75, POPL '04]
  - 2 thread atomicity patterns [Lu ASPLOS '06]
  - Conflict graphs [PLDI '08, RV '11]



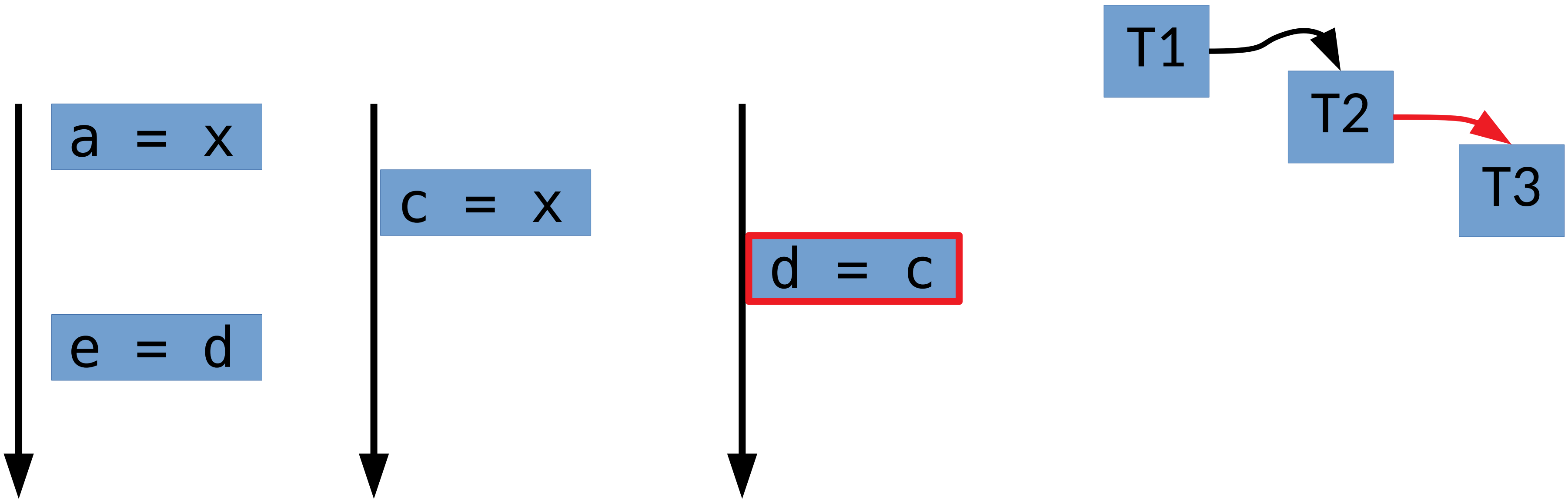
Atomicity Violations

- How can we find atomicity violations (or correctness)?
  - Lipton's Theory of Reduction [CACM '75, POPL '04]
  - 2 thread atomicity patterns [Lu ASPLOS '06]
  - Conflict graphs [PLDI '08, RV '11]



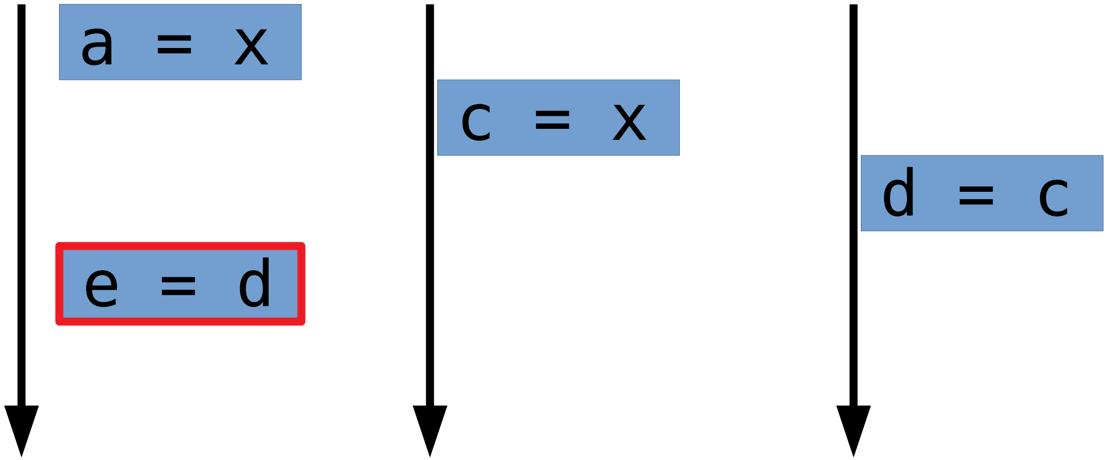
Atomicity Violations

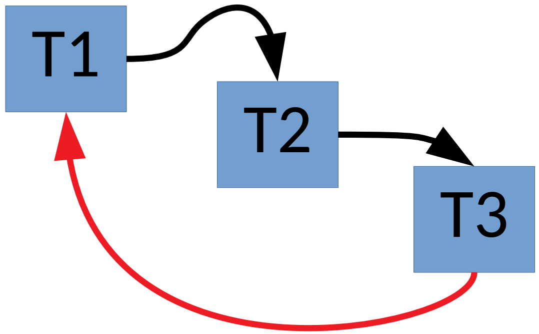
- How can we find atomicity violations (or correctness)?
  - Lipton's Theory of Reduction [CACM '75, POPL '04]
  - 2 thread atomicity patterns [Lu ASPLOS '06]
  - Conflict graphs [PLDI '08, RV '11]



Atomicity Violations

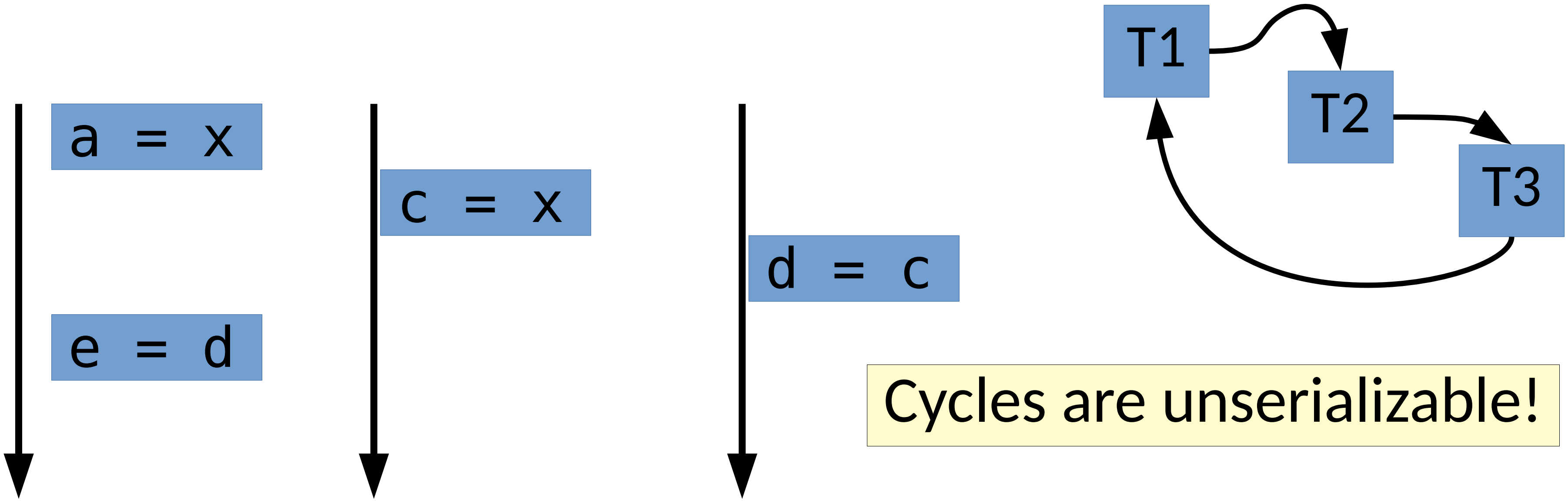
- How can we find atomicity violations (or correctness)?
  - Lipton's Theory of Reduction [CACM '75, POPL '04]
  - 2 thread atomicity patterns [Lu ASPLOS '06]
  - Conflict graphs [PLDI '08, RV '11]





Atomicity Violations

- How can we find atomicity violations (or correctness)?
  - Lipton’s Theory of Reduction [CACM ‘75, POPL ‘04]
  - 2 thread atomicity patterns [Lu ASPLOS ‘06]
  - Conflict graphs [PLDI ‘08, RV ‘11]



Cycles are unserializable!

Atomicity Violations

- How can we find atomicity violations (or correctness)?
  - Lipton’s Theory of Reduction **[CACM ‘75, POPL ‘04]**
  - 2 thread atomicity patterns **[Lu ASPLOS ‘06]**
  - Conflict graphs **[PLDI ‘08, RV ‘11]**
- How do we know what regions should be atomic?

Concurrency in the land of Single Threads

- Even programs without threads face concurrency bugs
  - Does your program have events?
  - Can one event lead to another event?

Concurrency in the land of Single Threads

- Even programs without threads face concurrency bugs
  - Does your program have events?
  - Can one event lead to another event?
  - The order of events can lead to bugs

[Hong 2014]

```
<html><body>
  <button id="b1"onclick="fn()">b1</button>
  <script>
    function fn(){
      m=null;
    };
  </script>

  <script src="lib.js"></script>
  <script>
    m={data:""};
  </script>

  <script src="extn.js"></script>
  <script>
    m.data ="text";
  </script>
</body></html>
```

Concurrency in the land of Single Threads

- Even programs without threads face concurrency bugs
  - Does your program have events?
  - Can one event lead to another event?
  - The order of events can lead to bugs

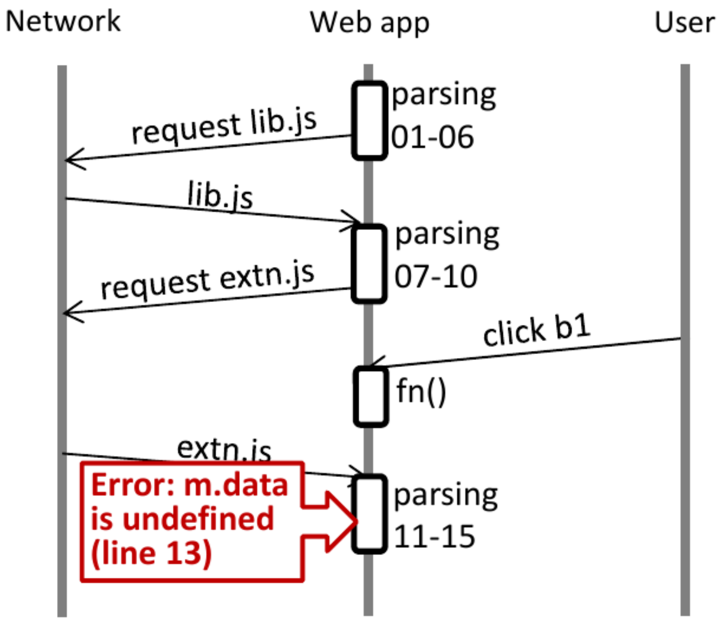
[Hong 2014]

```
<html><body>
  <button id="b1"onclick="fn()">b1</button>
  <script>
    function fn(){
      m=null;
    };
  </script>

  <script src="lib.js"></script>
  <script>
    m={data:""};
  </script>

  <script src="extn.js"></script>
  <script>
    m.data ="text";
  </script>
</body></html>
```

Concurrency in the land of Single Threads

- Even programs without threads face concurrency bugs
  - Does your program have events?
  - Can one event lead to another event?
  - The order of events can lead to bugs

[Hong 2014]




```
<html><body>
  <button id=“b1”onclick=“fn()”>b1</button>
  <script>
    function fn(){
      m=null;
    };
  </script>

  <script src=“lib.js”></script>
  <script>
    m={data:“”};
  </script>
  <script src=“extn.js”></script>
  <script>
    m.data =“text”;
  </script>
</body></html>
```

A user can click “here”

Concurrency in the land of Single Threads

- Even programs without threads face concurrency bugs
  - Does your program have events?
  - Can one event lead to another event?
  - The order of events can lead to bugs
  - While the meaning may be well defined, it can be unexpected & confusing.

Concurrency in the land of Single Threads

- Even programs without threads face concurrency bugs
  - Does your program have events?
  - Can one event lead to another event?
  - The order of events can lead to bugs
  - While the meaning may be well defined, it can be unexpected & confusing.
- In practice, semantic concurrency bugs exist in Javascript!

Concurrency in the land of Single Threads

- Even programs without threads face concurrency bugs
  - Does your program have events?
  - Can one event lead to another event?
  - The order of events can lead to bugs
  - While the meaning may be well defined, it can be unexpected & confusing.
- In practice, semantic concurrency bugs exist in Javascript!
  - Both in client side & server side [Raychev 2013, Hong 2014, Endo 2020]

Concurrency in the land of Single Threads

- Even programs without threads face concurrency bugs

```
var doneTasks = 0;
var entries = 0;

function processFile(filePath) {
  entries++;
  fs.lstat(filePath, function stat(err, stats) {
    if (err) {
      entries--;
      return;
    }
    useStatData(stats);
    fs.readFile(filePath, function read(err, data) {
      performTask(data);
      doneTasks++;
      if (doneTasks === entries)
        finalize();
    });
  });
}
```

[Endo 2020]

it can be unexpected & confusing.

exist in Javascript!

2013, Hong 2014, Endo 2020]

Concurrency in the land of Single Threads

- Even programs without threads face concurrency bugs

```
var doneTasks = 0;
var entries = 0;

function processFile(filePath) {
  entries++;
  fs.lstat(filePath, function stat(err, stats) {
    if (err) {
      entries--;
      return;
    }
    useStatData(stats);
    fs.readFile(filePath, function read(err, data) {
      performTask(data);
      doneTasks++;
      if (doneTasks === entries)
        finalize();
    });
  });
}
```

[Endo 2020]

it can be unexpected & confusing.

exist in Javascript!

2013, Hong 2014, Endo 2020]

Concurrency in the land of Single Threads

- Even programs without threads face concurrency bugs

[Endo 2020]

```
var doneTasks = 0;
var entries = 0;

function processFile(filePath) {
  entries++;
  fs.lstat(filePath, function stat(err, stats) {
    if (err) {
      entries--;
      return;
    }
    useStatData(stats);
    fs.readFile(filePath, function read(err, data) {
      performTask(data);
      doneTasks++;
      if (doneTasks === entries)
        finalize();
    });
  });
}
```

it can be unexpected & confusing.

exist in Javascript!

2013, Hong 2014, Endo 2020]

Concurrency in the land of Single Threads

- Even programs without threads face concurrency bugs

```
var doneTasks = 0;
var entries = 0;

function processFile(filePath) {
  entries++;
  fs.lstat(filePath, function stat(err, stats) {
    if (err) {
      entries--;
      return;
    }
    useStatData(stats);
    fs.readFile(filePath, function read(err, data) {
      performTask(data);
      doneTasks++;
      if (doneTasks === entries)
        finalize();
    });
  });
}
```

[Endo 2020]

it can be unexpected & confusing.

exist in Javascript!

2013, Hong 2014, Endo 2020]

```
processFile('existing-file.txt');
processFile('missing-file.txt');
processFile('empty-file.txt');
```

Concurrency in the land of Single Threads

- Even programs without threads face concurrency bugs

[Endo 2020]

...it can be unexpected & confusing.

...exist in Javascript!

...2013, Hong 2014, Endo 2020]

```
var doneTasks = 0;
var entries = 0;

function processFile(filePath) {
  entries++;
  fs.lstat(filePath, function stat(err, stats) {
    if (err) {
      entries--;
      return;
    }
    useStatData(stats);
    fs.readFile(filePath, function read(err, data) {
      performTask(data);
      doneTasks++;
      if (doneTasks === entries)
        finalize();
    });
  });
}
```

```
processFile('existing-file.txt');
processFile('missing-file.txt');
processFile('empty-file.txt');
```

When the missing file stats last,
finalize is not called,
and Node hangs.

Concurrency in the land of Single Threads

- Even programs without threads face concurrency bugs
  - Does your program have events?
  - Can one event lead to another event?
  - The order of events can lead to bugs
  - While the meaning may be well defined, it can be unexpected & confusing.
- In practice, semantic concurrency bugs exist in Javascript!
  - Both in client side & server side [Raychev 2013, Hong 2014, Endo 2020]
  - In fact, locks are simulated to control interleavings
- Careful (1) monitoring, (2) happens-before analysis, & (3) test generation can automatically find these issues

Concurrent Test Generation

- What if we don't already have a buggy execution?

Concurrent Test Generation

- What if we don't already have a buggy execution?
- Explore bounded schedules
  - 2 threads and few pre-emptions finds most bugs

Concurrent Test Generation

- What if we don't already have a buggy execution?
- Explore bounded schedules
  - 2 threads and few pre-emptions finds most bugs
- Careful schedule generation & selection

Concurrent Test Generation

- What if we don't already have a buggy execution?
- Explore bounded schedules
  - 2 threads and few pre-emptions finds most bugs
- Careful schedule generation & selection
- Generate API unit tests targeting concurrency
  - Small enough for exhaustive schedule exploration

Other Directions

- Shepherding toward good behaviors
- Tolerating & avoiding on the fly
- Static analysis

Summary

- Parallelism is important for modern performance
- Choosing what to parallelize can be hard
- Parallelizing correctly can be very hard

Summary

- Parallelism is important for modern performance
- Choosing what to parallelize can be hard
- Parallelizing correctly can be very hard

And the hard problems
are interesting to study.