A Review/Tour of Concurrency & Parallelism

CMPT 886
Automated Software Analysis & Security
Nick Sumner

Seeking out performance

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

Seeking out performance

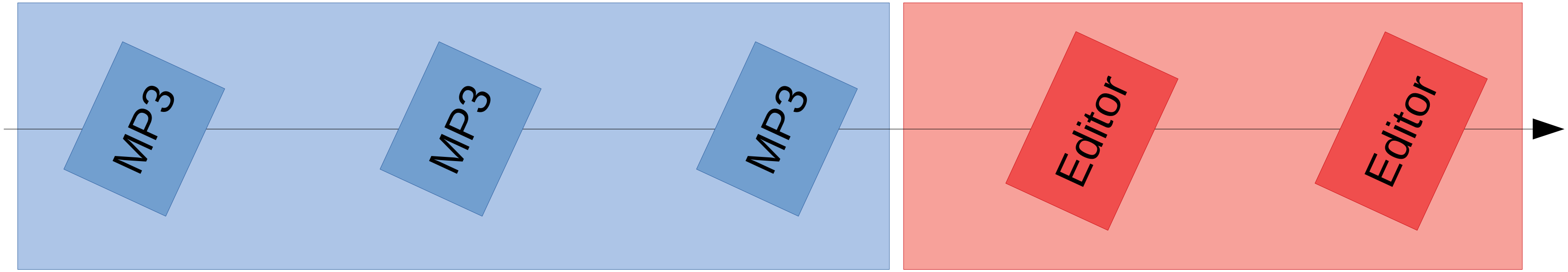
- Improving performance can come from tuning
  - Algorithmic complexity
  - Memory access patterns
  - Concurrency
  - Parallelism
- As processor speeds have slowed increasing, much focus has been placed on the last two

Concurrency & Parallelism

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

Concurrency & Parallelism

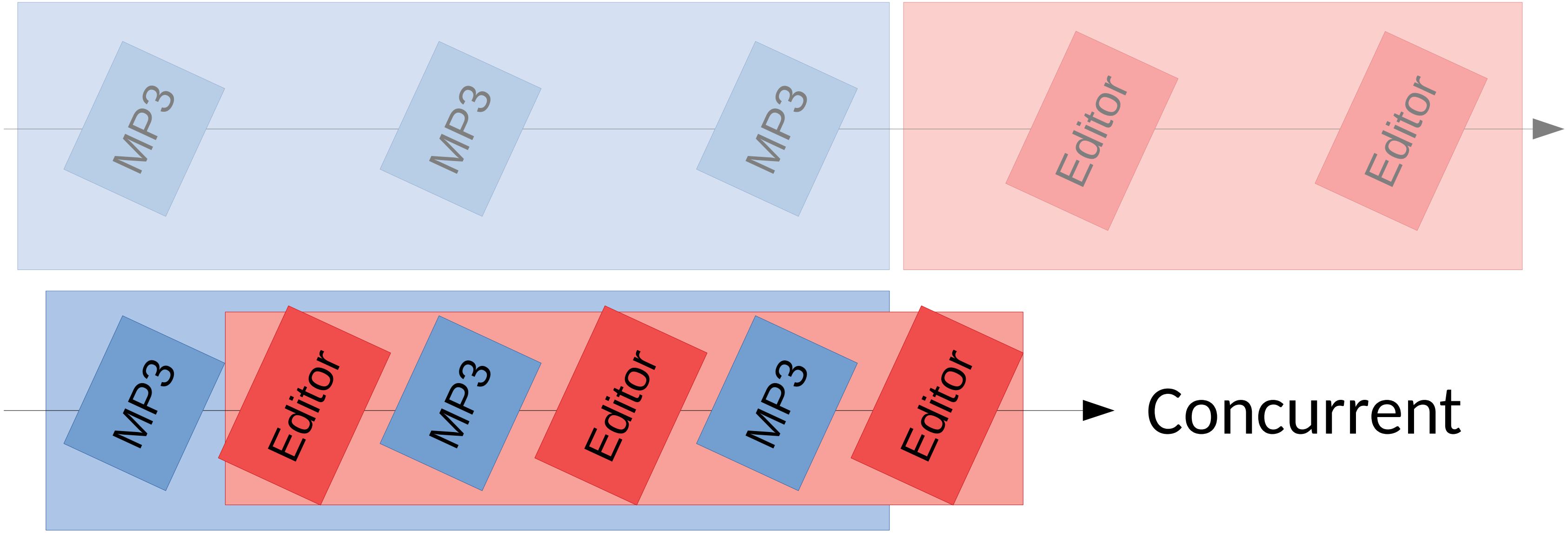
- ***Concurrency*** is the management of multiple tasks at the same time.
  - e.g. Sharing a CPU across multiple processes.



Sequential

Concurrency & Parallelism

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

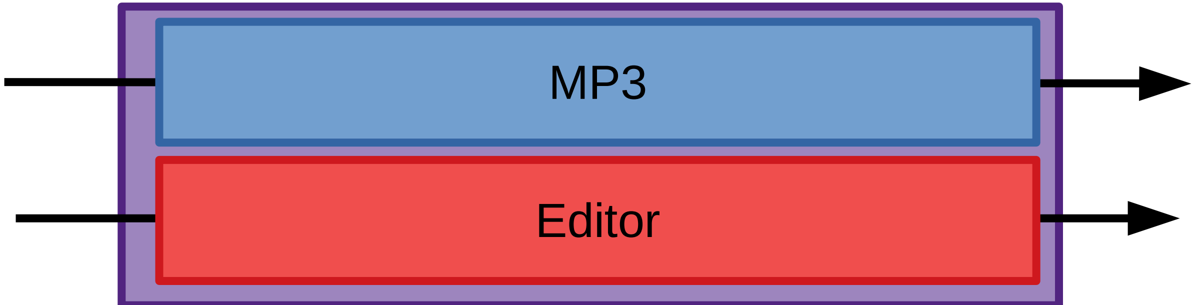


Concurrency & Parallelism

- ***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

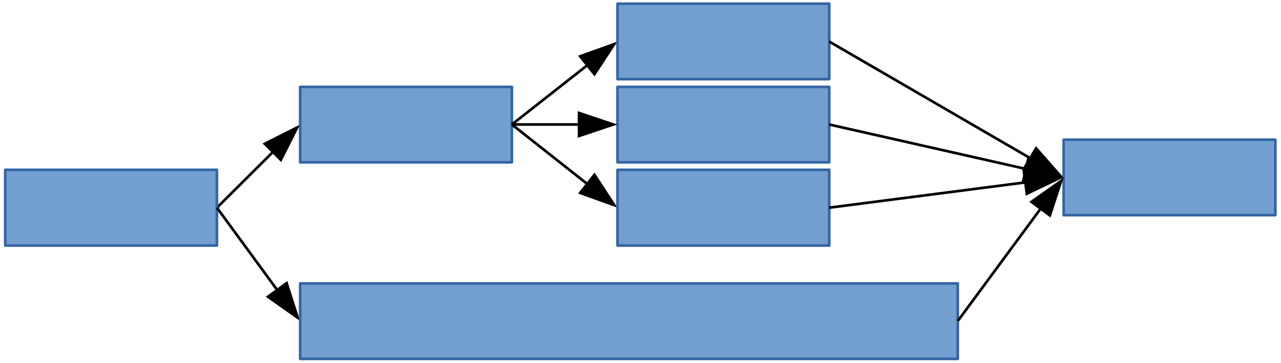
Concurrency & Parallelism

- *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

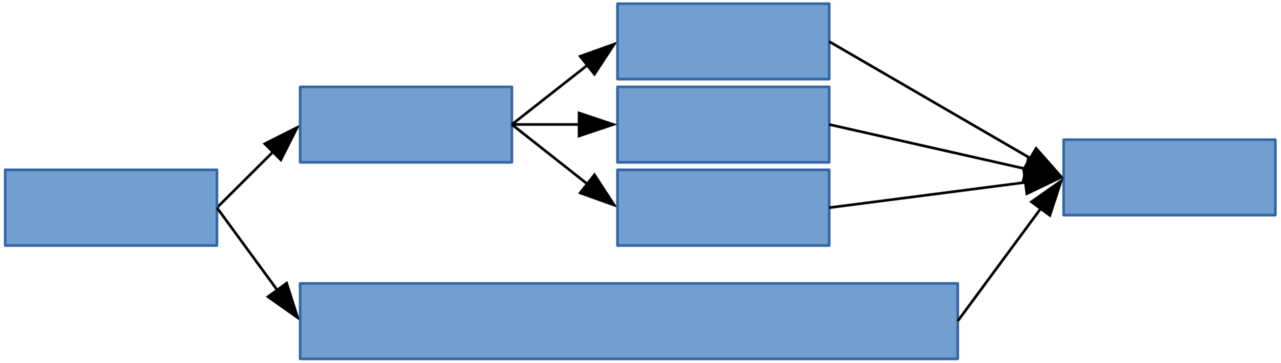


Parallel

Sequential

Using Parallelism

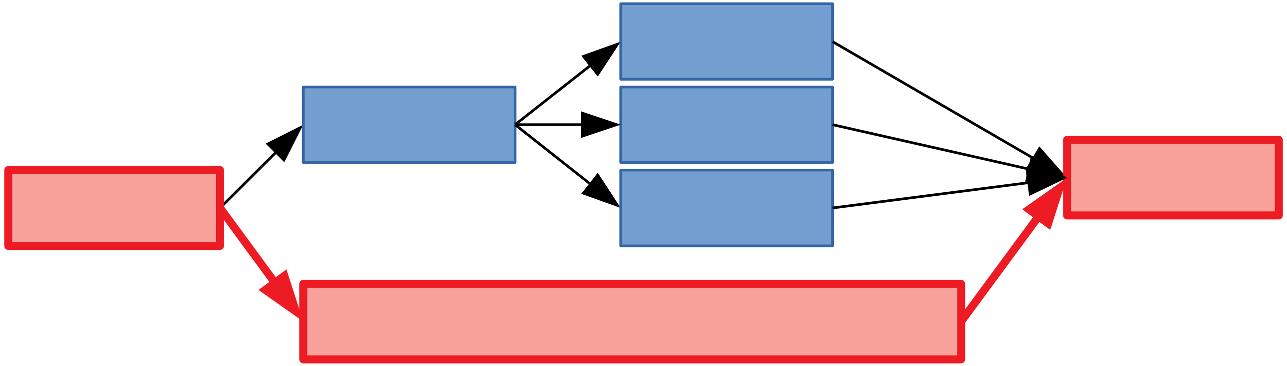
- Large problems can sometimes be split into parallel tasks, and the effects of the parallel tasks combined



- 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

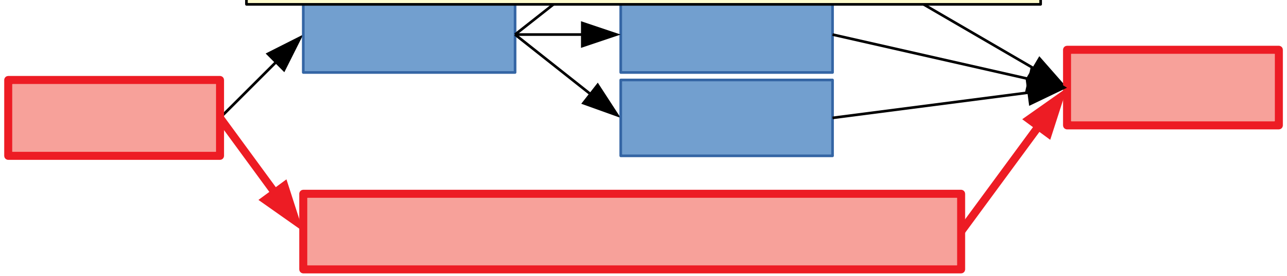


- The best possible running time is determined by the ***critical path*** or ***span*** of dependent tasks through the program.

Using Parallelism

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

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

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

- Identifying good opportunities for effective parallelism is open to research

Using Parallelism

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

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

- 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

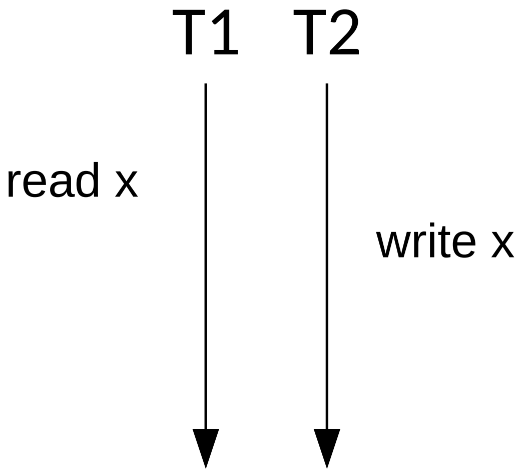
- A data race occurs when:
  1) two threads access the same location
  2) the accesses are *logically simultaneous*

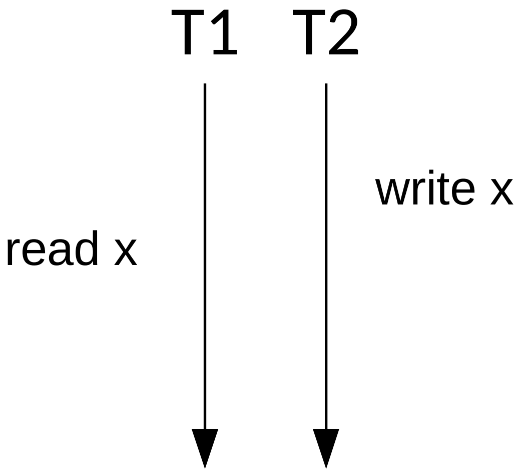
Data Races

- 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, RAR)

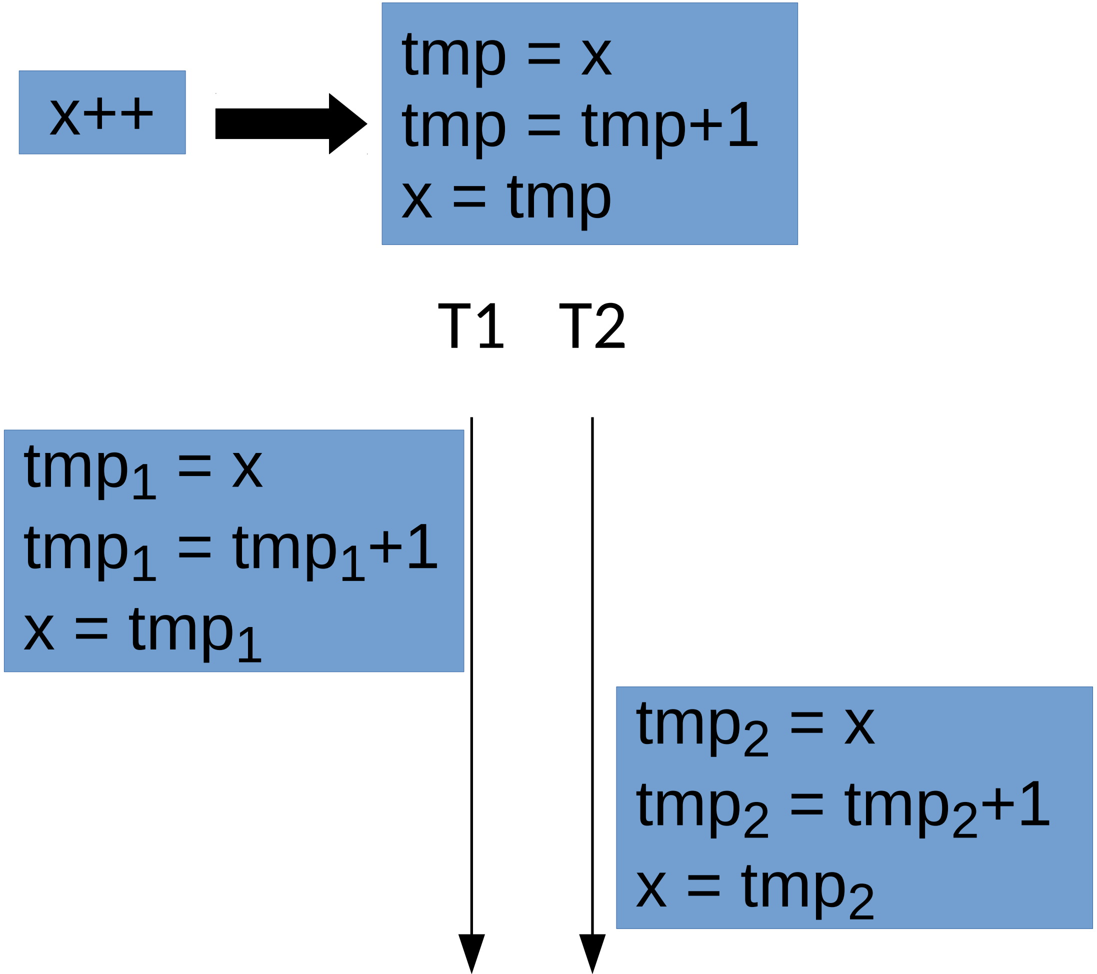
Data Races

- 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, RAR)





Data Races



x++

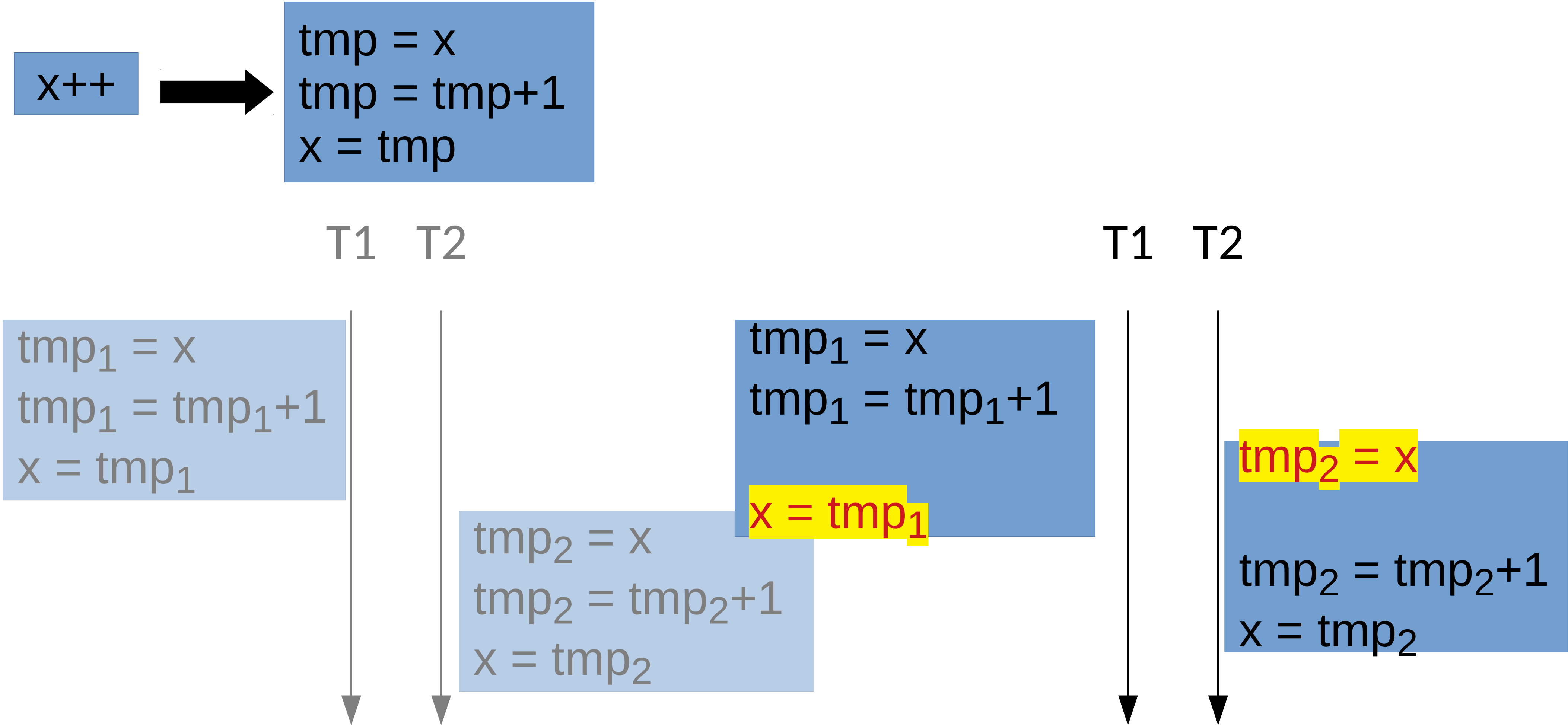
tmp = x
tmp = tmp+1
x = tmp

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

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

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

T1 T2

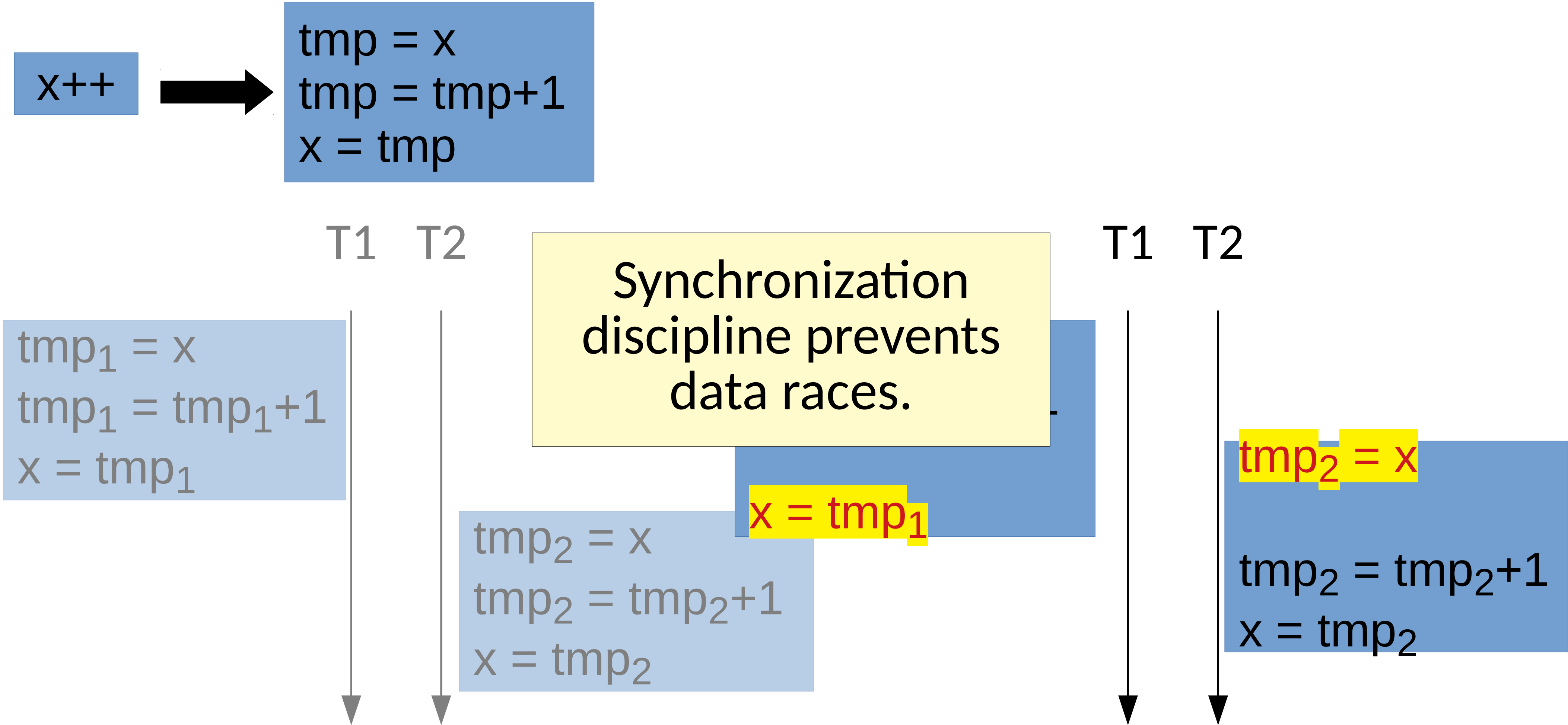
$tmp_1 = x$
$tmp_1 = tmp_1+1$

$x = tmp_1$

$tmp_2 = x$

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

Data Races



x++

tmp = x
tmp = tmp+1
x = tmp

T1 T2

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

“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]

“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;
```

“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;
```

```
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]

“Benign” Data Races

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

[Boehm, Hotpar 2011]

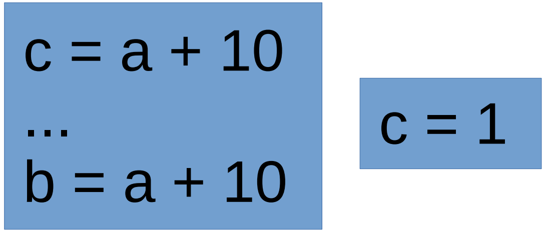
- Data race freedom allows extra reads.

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

False

True

“Benign” Data Races



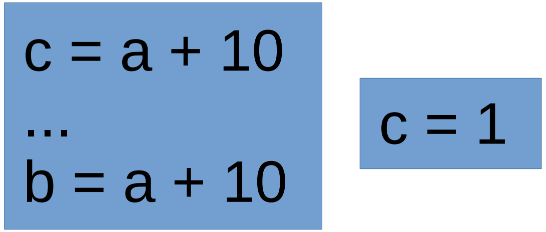
```
c = a + 10
...
b = a + 10
```

```
c = 1
```

[Dolan, PLDI 2018]

- Races can introduce bugs on non-racy variables

“Benign” Data Races

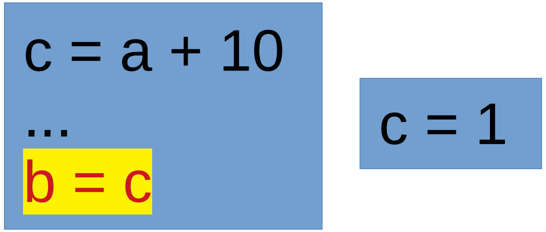


```
c = a + 10
...
b = a + 10
```

```
c = 1
```

[Dolan, PLDI 2018]

- Races can introduce bugs on non-racy variables



```
c = a + 10
...
b = c
```

```
c = 1
```

“Benign” Data Races

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

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

[Dolan, PLDI 2018]

- Races can jump forward and backward in time

“Benign” Data Races

```
a = 1          a = 2
flag = true    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

```
               a = 2
a = 1
flag = true
               f = flag
               b = a
               c = 2
```

“Benign” Data Races

```
a = 1          a = 2
flag = true    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

```
               a = 2
a = 1
flag = true
               f = flag
               b = a
               c = 2
```

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

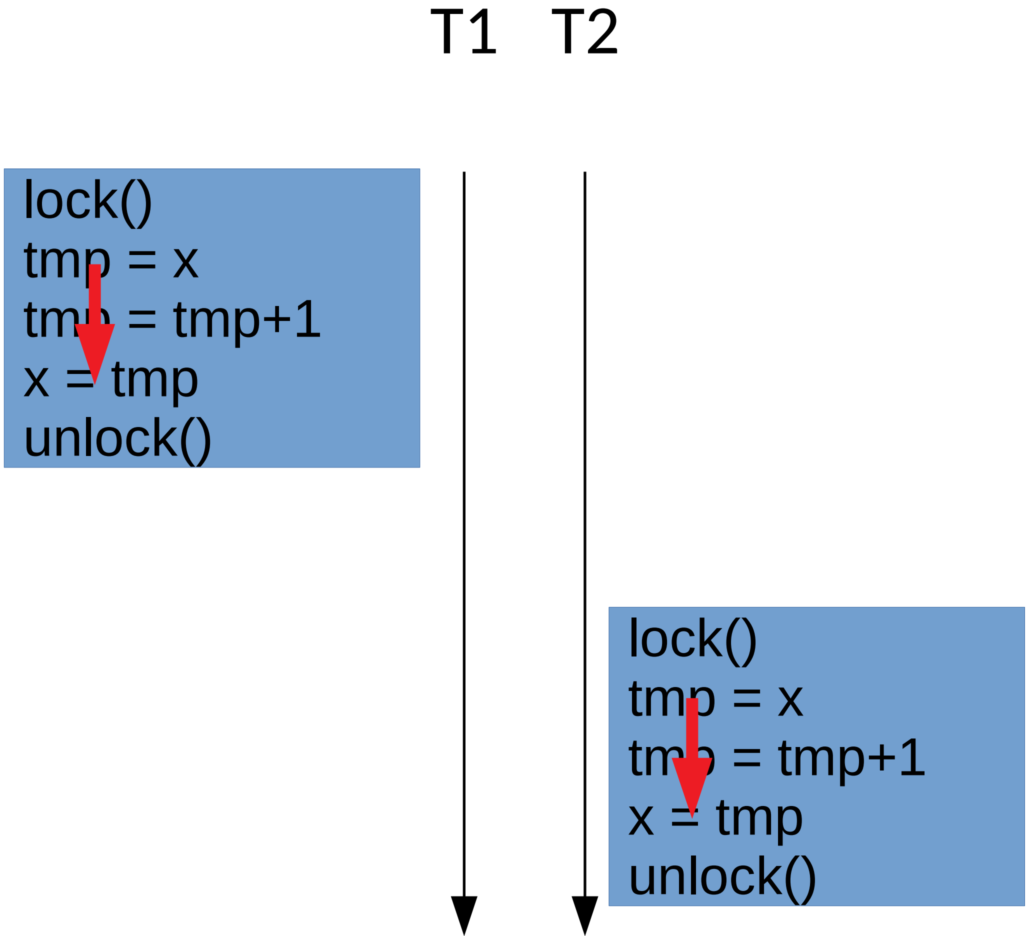
Happens-Before Ordering

- 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 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 variable order
- Happens-Before ordering of a specific execution can be tracked to identify bugs

Happens-Before Ordering

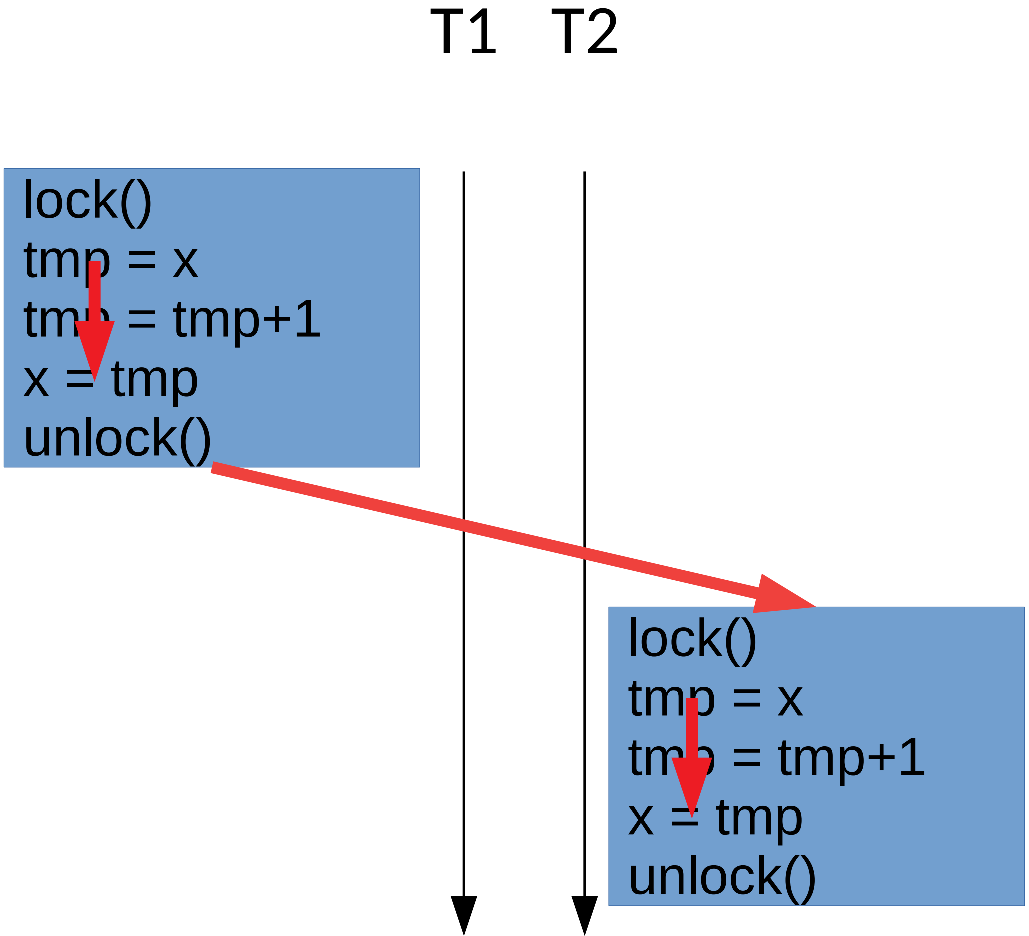


T1 T2

```
lock()
tmp = x
tmp = tmp+1
x = tmp
unlock()
```

```
lock()
tmp = x
tmp = tmp+1
x = tmp
unlock()
```

Happens-Before Ordering

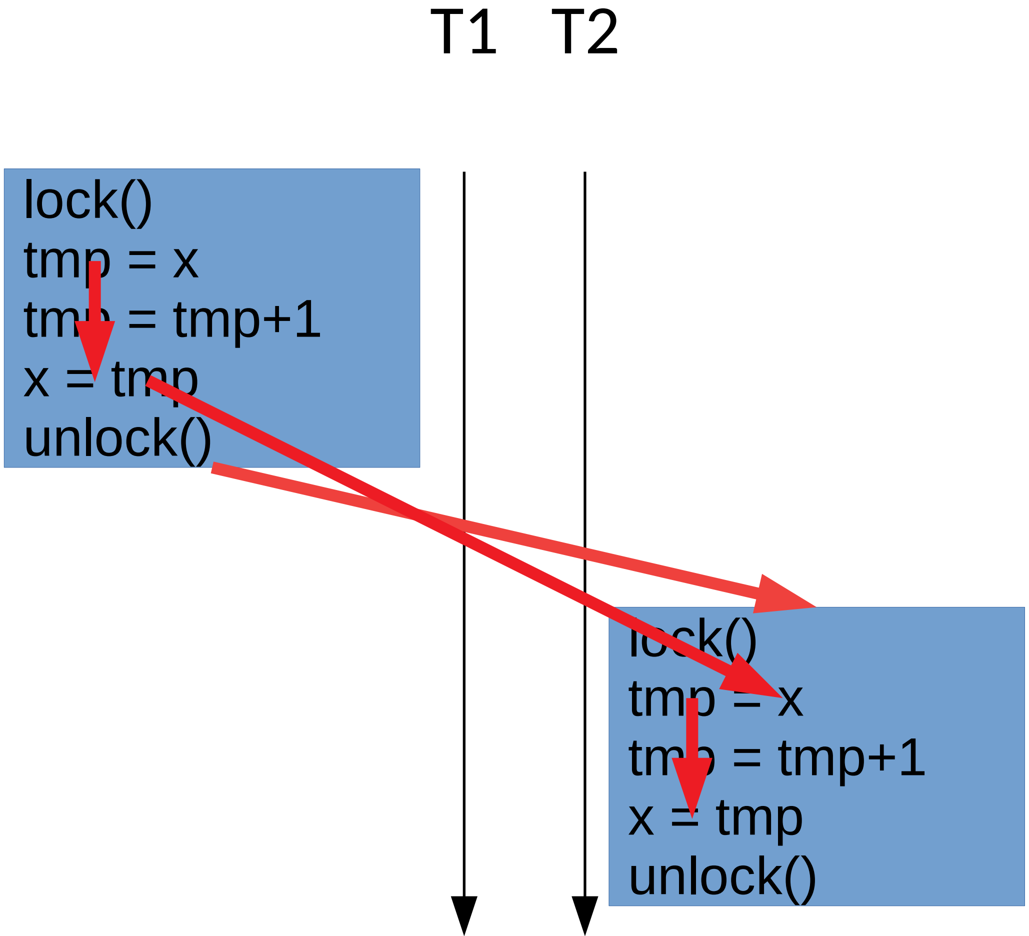


T1 T2

```
lock()
tmp = x
tmp = tmp+1
x = tmp
unlock()
```

```
lock()
tmp = x
tmp = tmp+1
x = tmp
unlock()
```

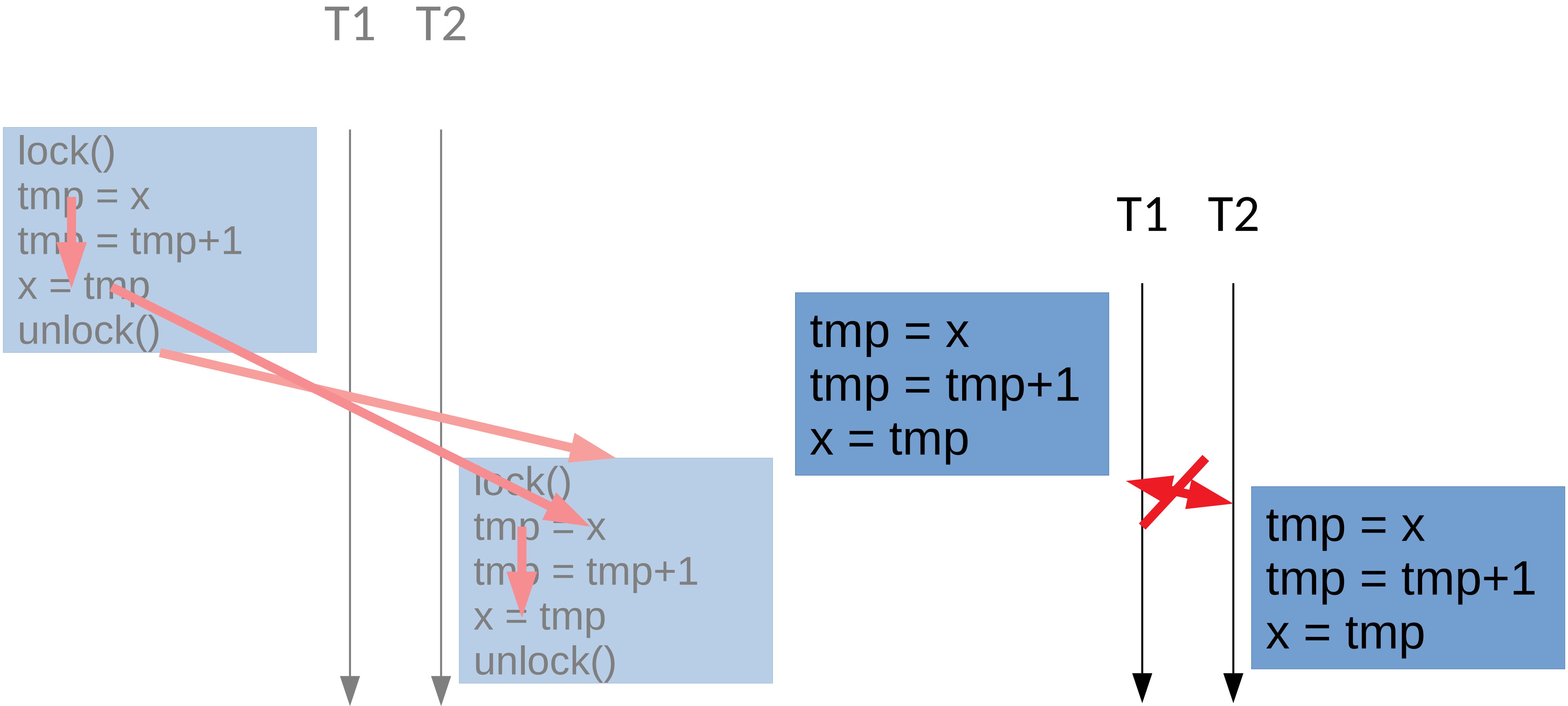
Happens-Before Ordering



```
lock()
tmp = x
tmp = tmp+1
x = tmp
unlock()
```

```
lock()
tmp = x
tmp = tmp+1
x = tmp
unlock()
```

Happens-Before Ordering



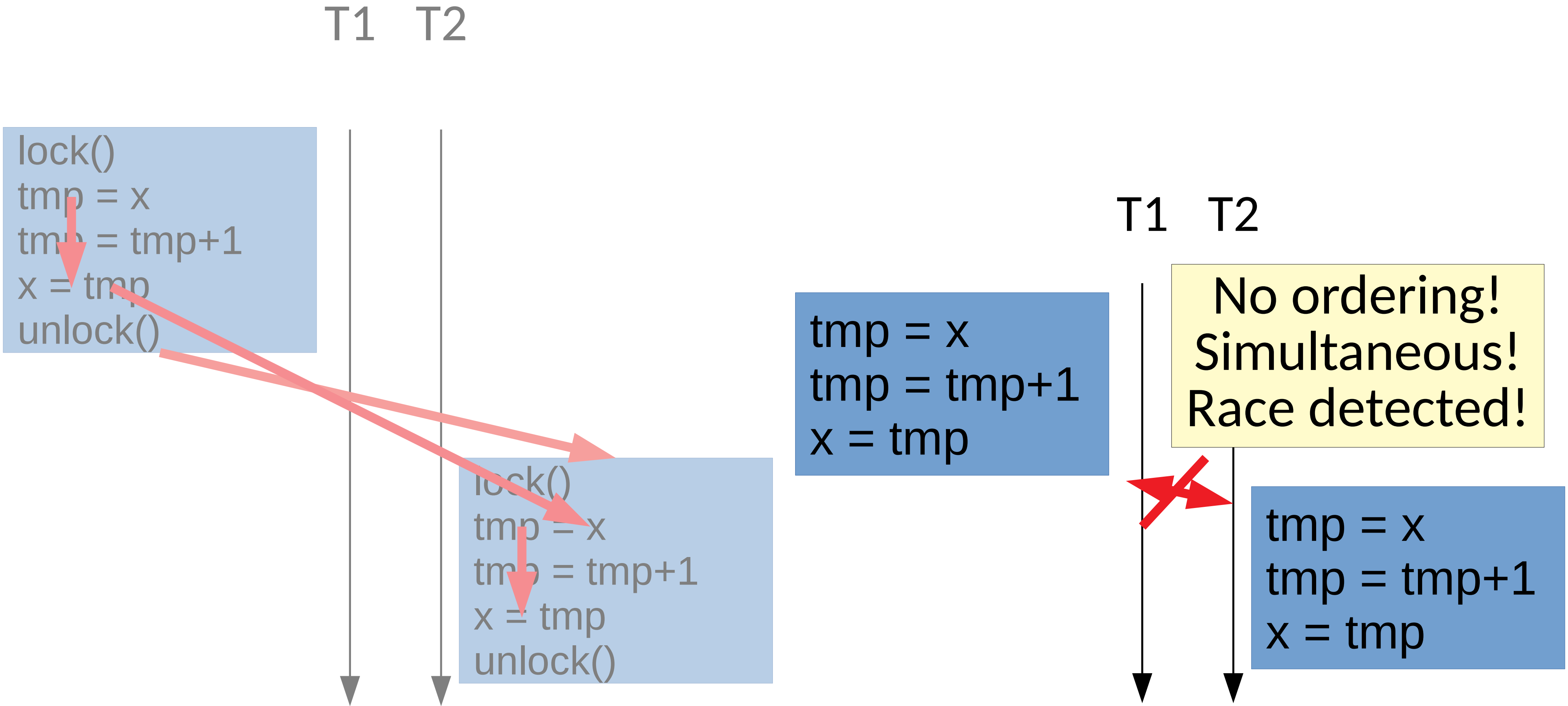
```
lock()
tmp = x
tmp = tmp+1
x = tmp
unlock()
```

```
lock()
tmp = x
tmp = tmp+1
x = tmp
unlock()
```

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

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

Happens-Before Ordering



T1 T2

```
lock()
tmp = x
tmp = tmp+1
x = tmp
unlock()
```

```
lock()
tmp = x
tmp = tmp+1
x = tmp
unlock()
```

T1 T2

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

No ordering!
Simultaneous!
Race detected!

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

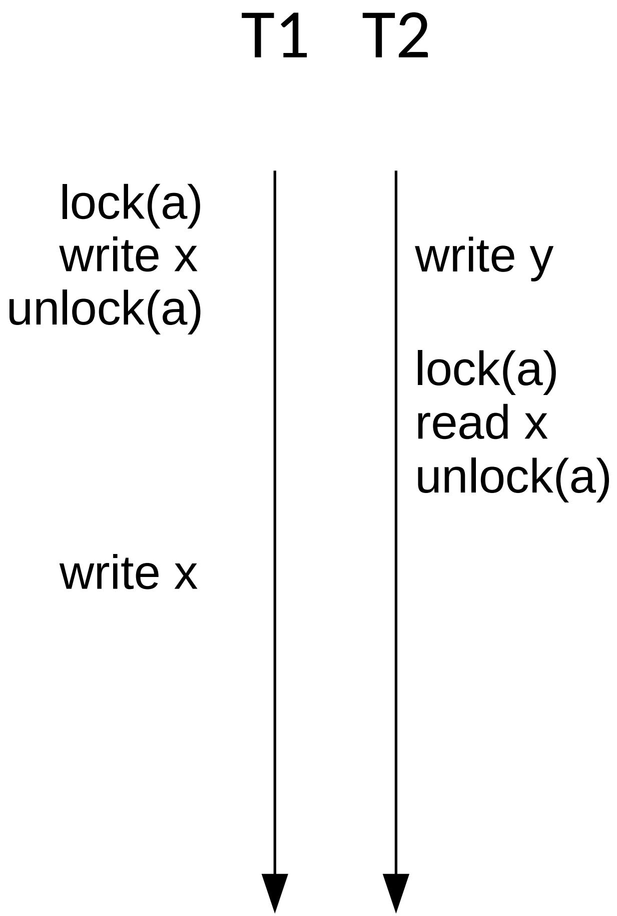
Happens-Before Ordering

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

Happens-Before Ordering

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

Logical Time & Vector Clocks



T1 T2

T1: lock(a)
write x
unlock(a)

T2: write y

T2: lock(a)
read x
unlock(a)

T1: write x

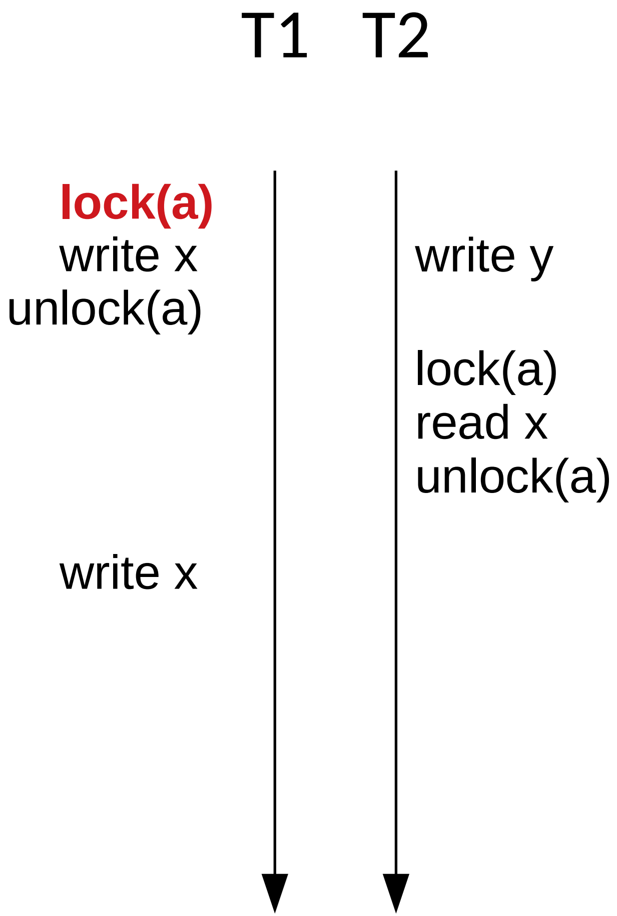
Clocks

```
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
unlock(a)

write y

lock(a)
read x
unlock(a)

write x

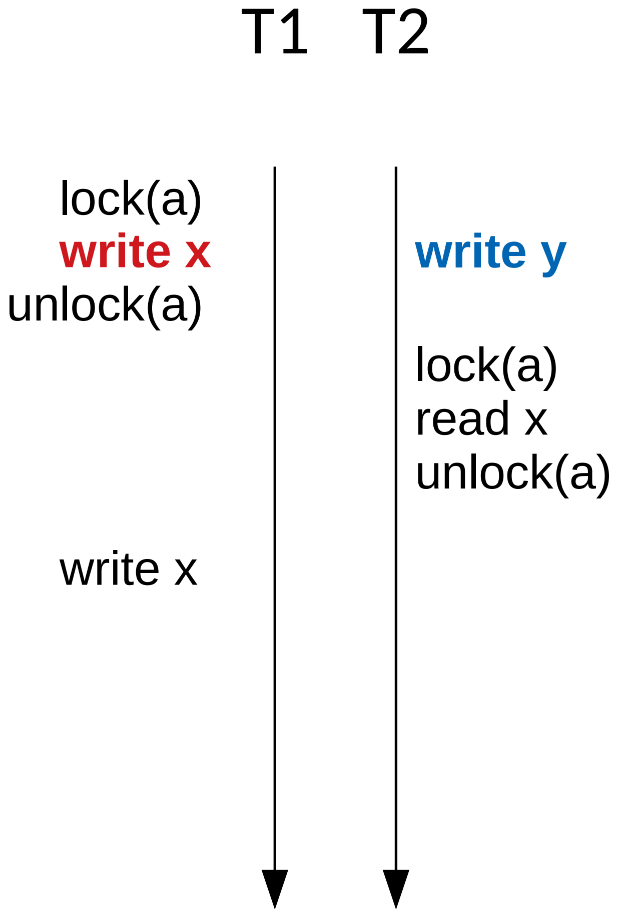
Clocks

```
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



T1 T2

lock(a)
write x
unlock(a)

write y

lock(a)
read x
unlock(a)

write x

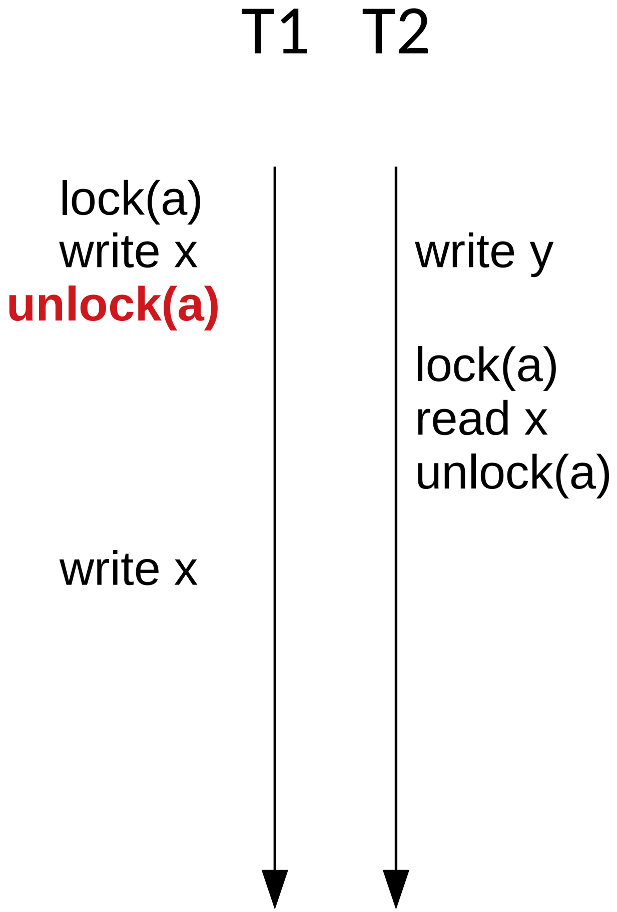
Clocks

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



T1 T2

T1:

lock(a)
write x
unlock(a)

write x

T2:

write y

lock(a)
read x
unlock(a)

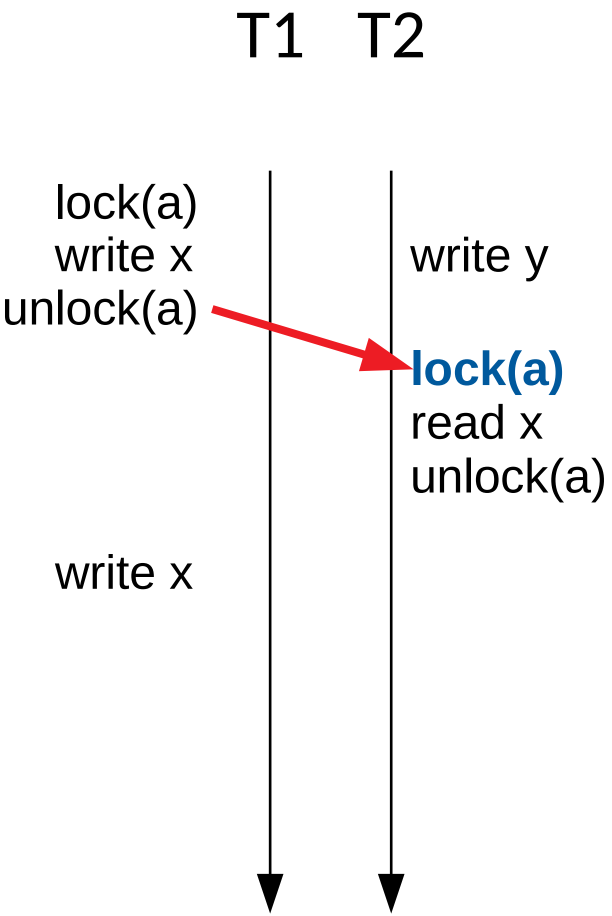
Clocks

```
T1↦[2,0]
T2↦[0,0]
```

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
unlock(a)

write y

lock(a)
read x
unlock(a)

write x

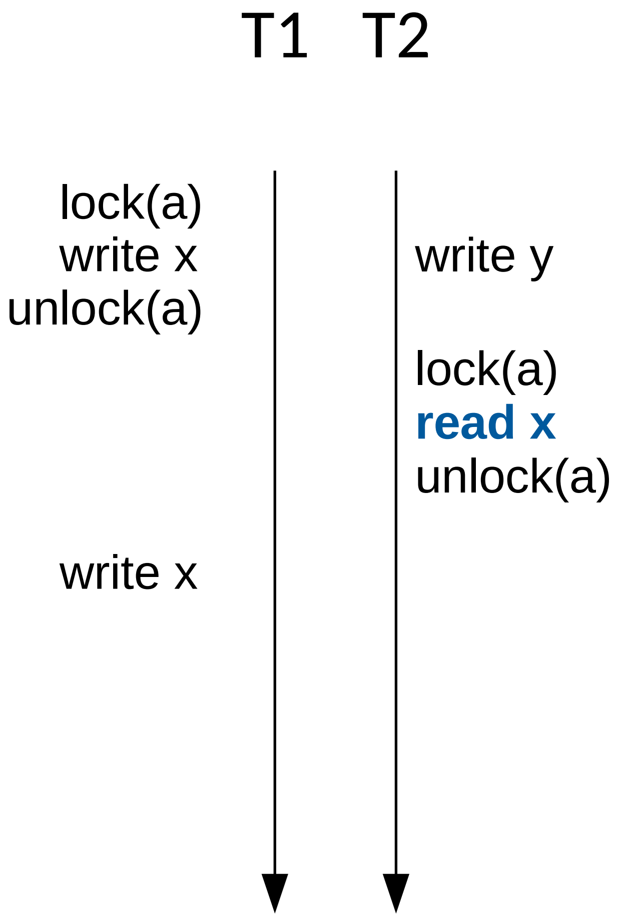
Clocks

```
T1↦[2,0]
T2↦[2,1]
```

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
unlock(a)

write y

lock(a)
read x
unlock(a)

write x

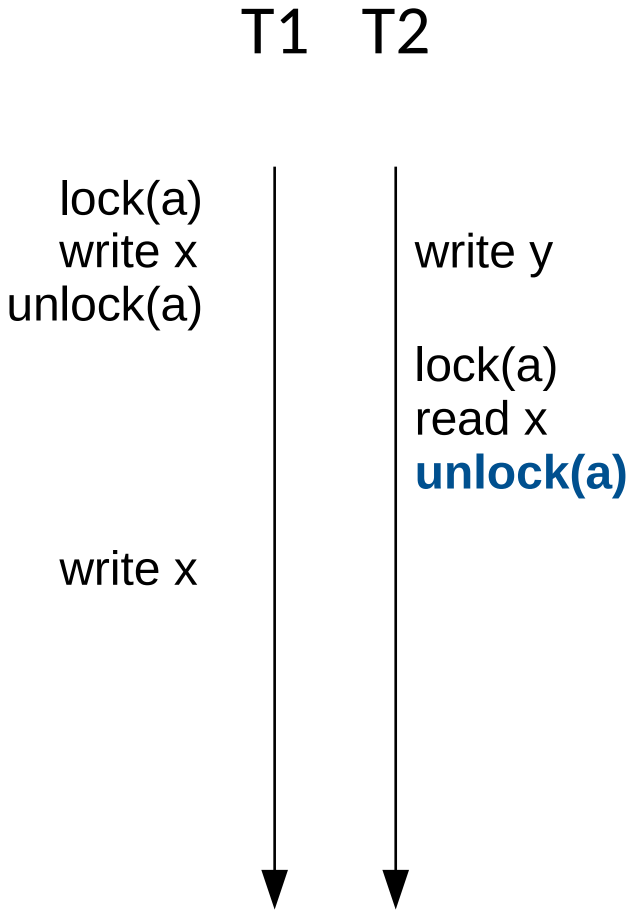
Clocks

```
T1↦[2,0]
T2↦[2,1]
```

Shadow Memory

```
x↦ R[2,1]
   W[1,0]
y↦ R[0,0]
   W[0,0]
```

Logical Time & Vector Clocks



T1 T2

T1:

lock(a)
write x
unlock(a)

write x

T2:

write y

lock(a)
read x
unlock(a)

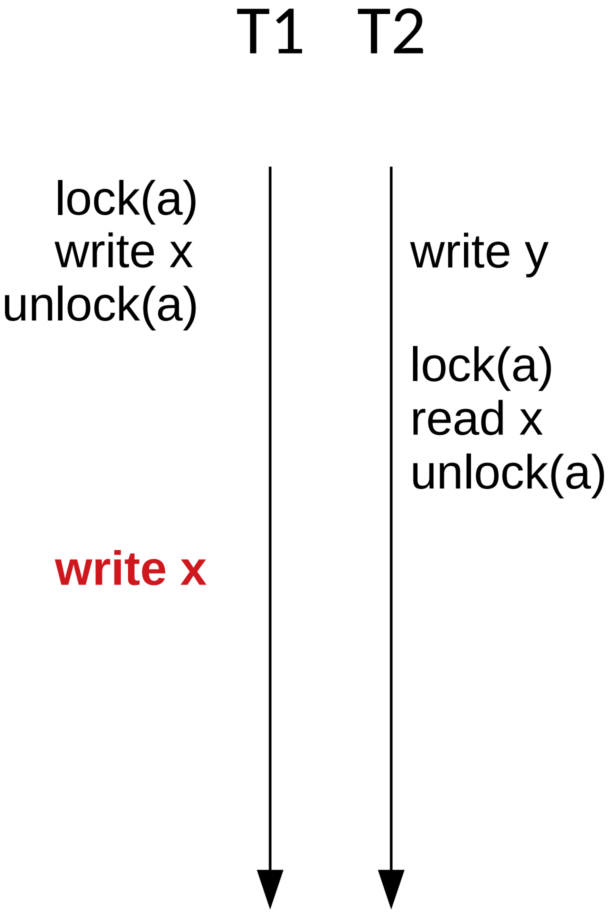
Clocks

```
T1↦[2,0]
T2↦[2,2]
```

Shadow Memory

```
x↦ R[2,1]
   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

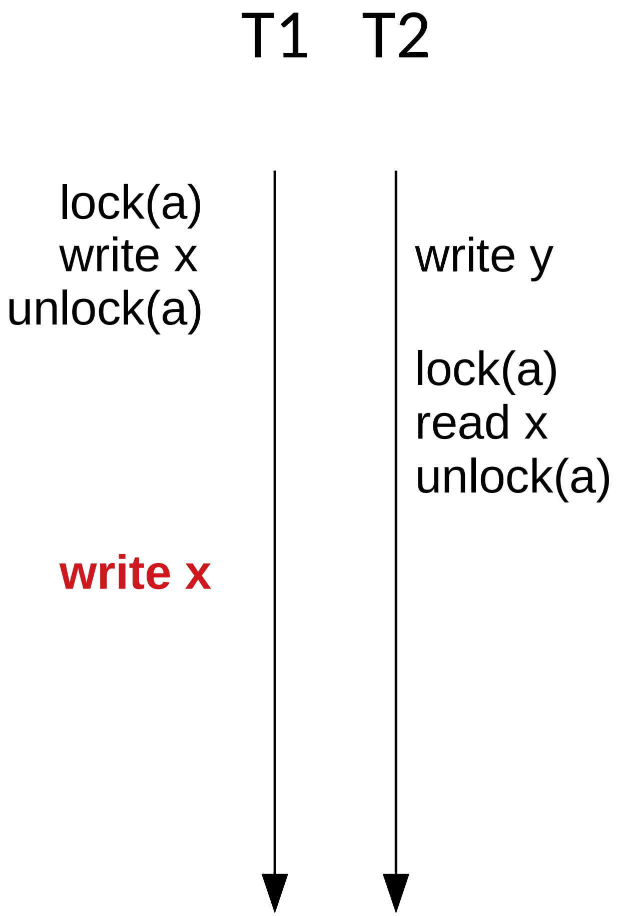
Clocks

```
T1↦[2,0]
T2↦[2,2]
```

Shadow Memory

```
x↦ R[2,1]
   W[2,0]
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↦[2,2]
```

Shadow Memory

```
x↦ R[2,1]
   W[2,0]
y↦ R[0,0]
   W[0,0]
```

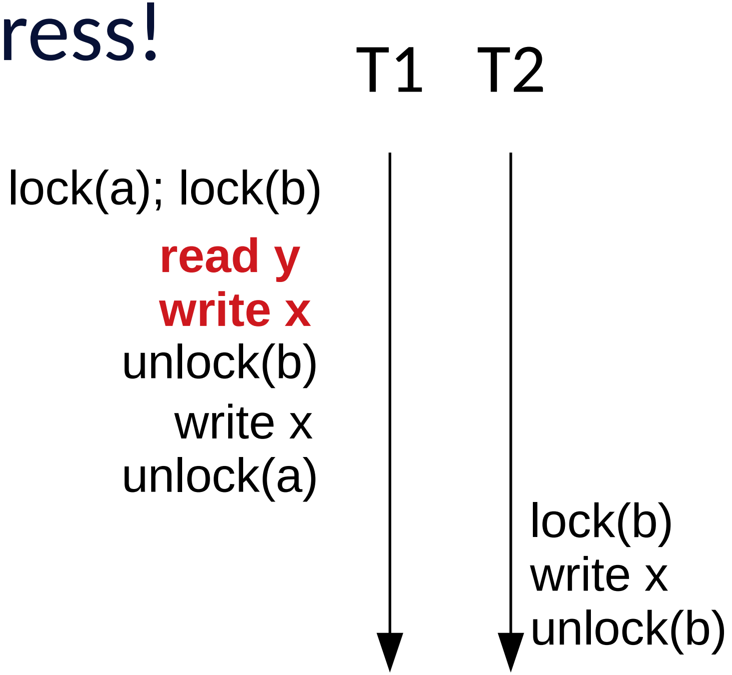
C(R,X) ⋢ C(W,X) → race!
(simplified for this case)

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!



T1 T2

lock(a); lock(b)

read y

write x

unlock(b)

write x

unlock(a)

lock(b)

write x

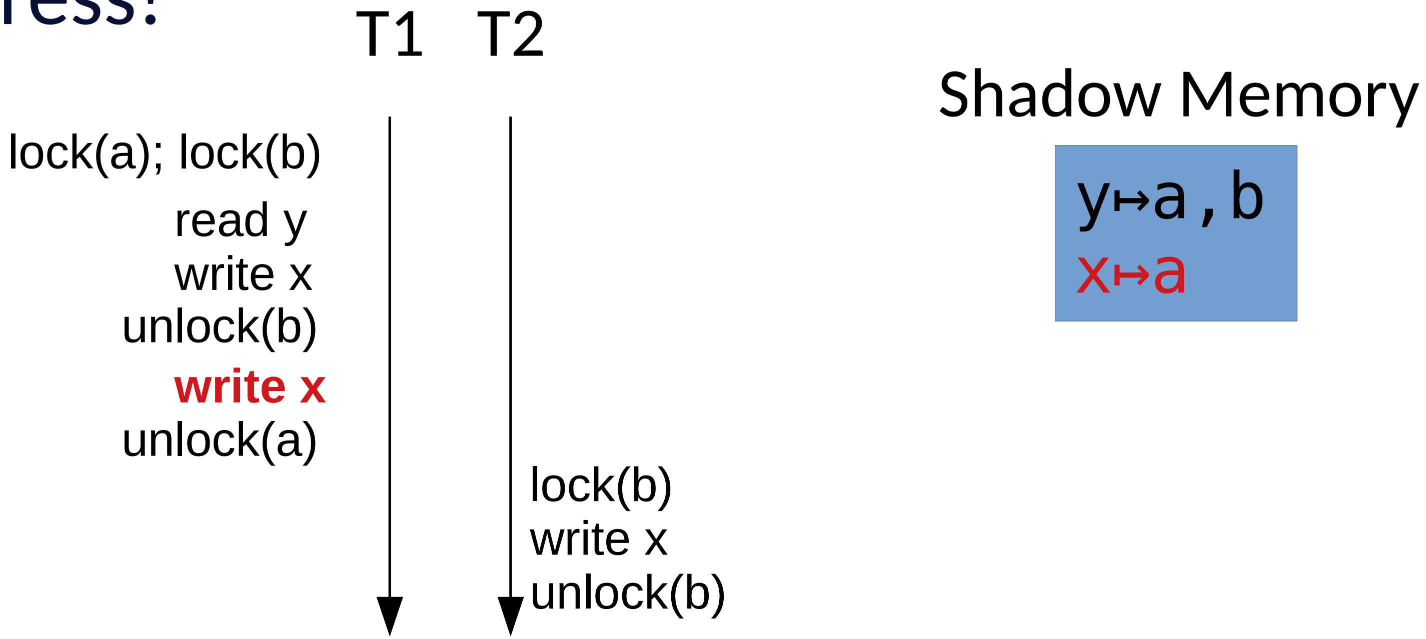
unlock(b)

Shadow Memory

y↦a,b
x↦a,b

Data Race Detection - Locksets

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



T1 T2

lock(a); lock(b)
read y
write x
unlock(b)
write x
unlock(a)

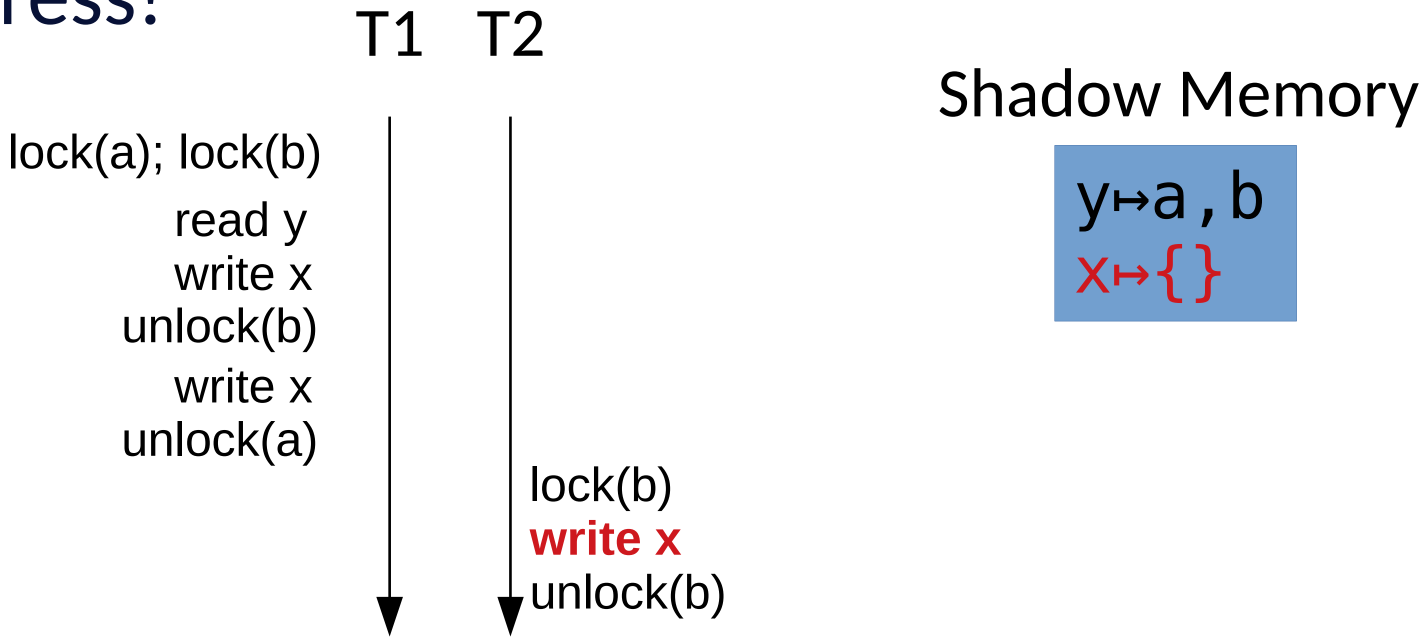
lock(b)
write x
unlock(b)

Shadow Memory

y↦a,b
x↦a

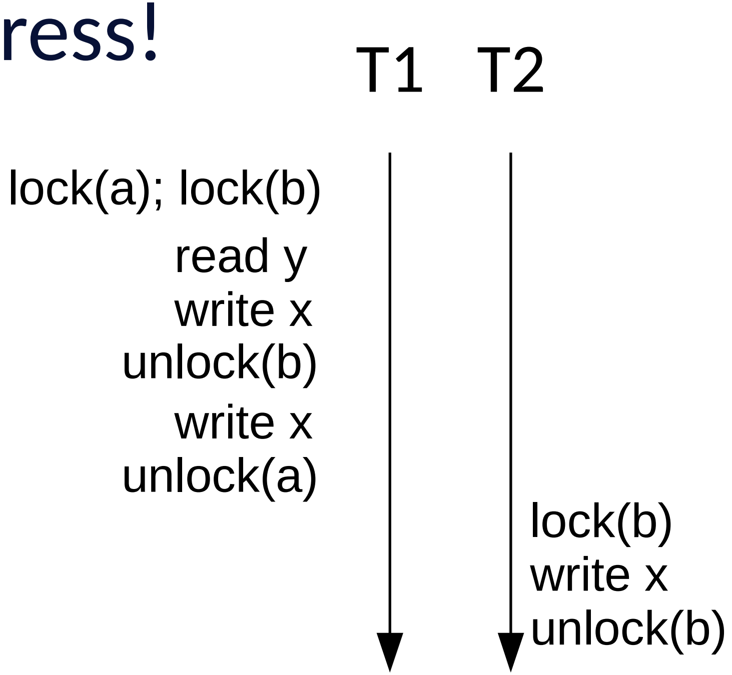
Data Race Detection - Locksets

- 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!



T1 T2

lock(a); lock(b)
read y
write x
unlock(b)
write x
unlock(a)

lock(b)
write x
unlock(b)

Shadow Memory

y↦a,b
x↦{}

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

Data Race Detection - Locksets

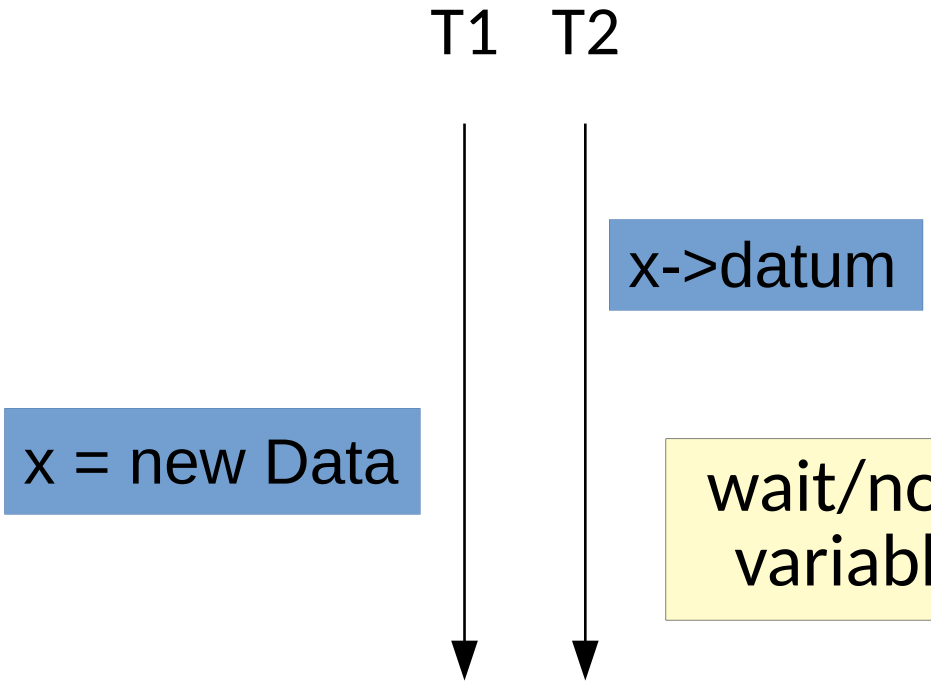
- Lockset based data race detection has many issues
  - Synchronization may be fork/join, wait/notify based
  - Initialization --> Process in Parallel --> Combine
  - Richer parallel designs

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

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()
```

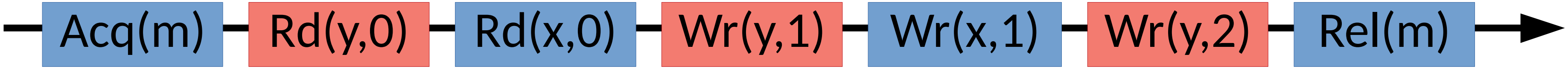
What do we really want?

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

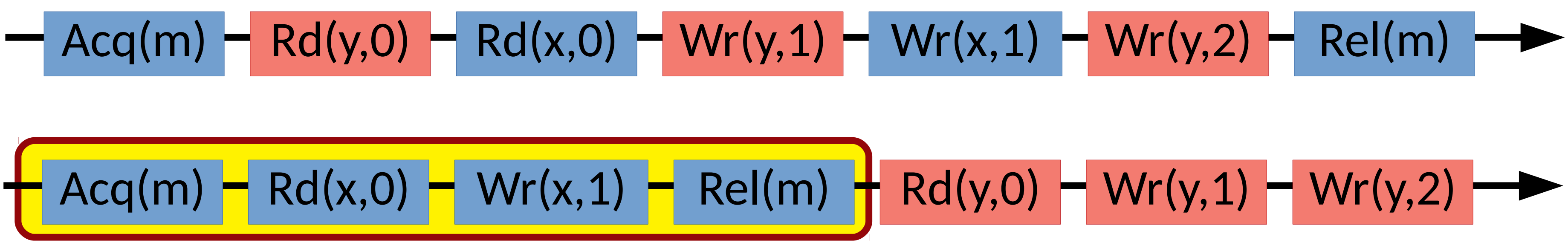
- Data races are a matter of perspective
  - Fine grained locking doesn't solve much.
- An execution (or fragment thereof) is ***atomic*** if it is equivalent to a sequentially executed one.



Acq(m) Rd(y,0) Rd(x,0) Wr(y,1) Wr(x,1) Wr(y,2) Rel(m)

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.



Acq(m) Rd(y,0) Rd(x,0) Wr(y,1) Wr(x,1) Wr(y,2) Rel(m)

Acq(m) Rd(x,0) Wr(x,1) Rel(m) Rd(y,0) Wr(y,1) Wr(y,2)

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

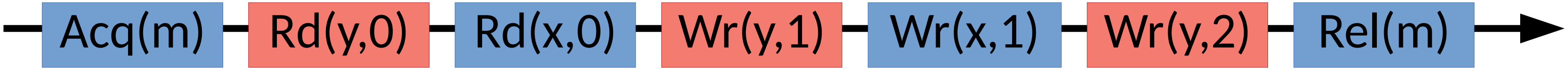
- How can we find atomicity violations?

Atomicity Violations

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

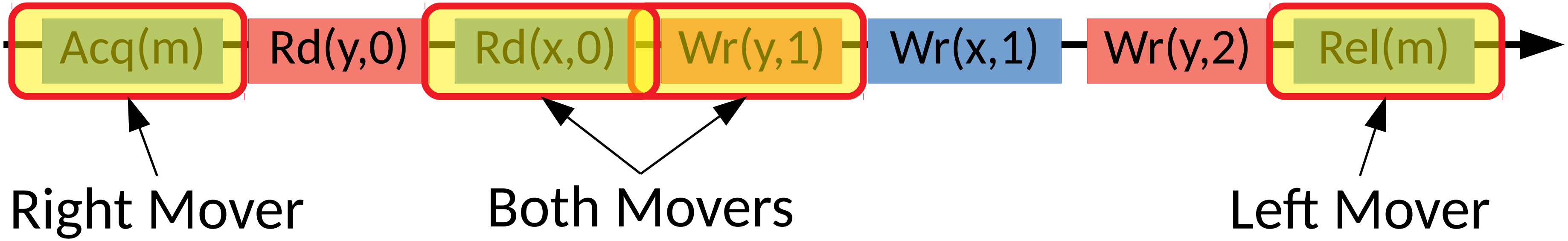
Atomicity Violations

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



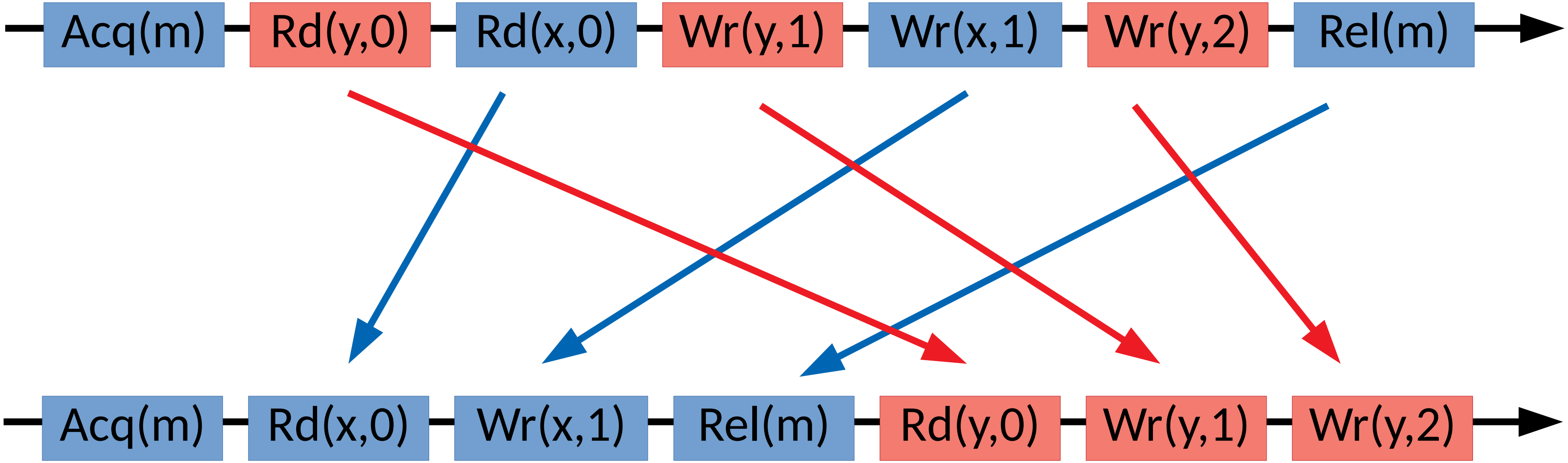
Atomicity Violations

- 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

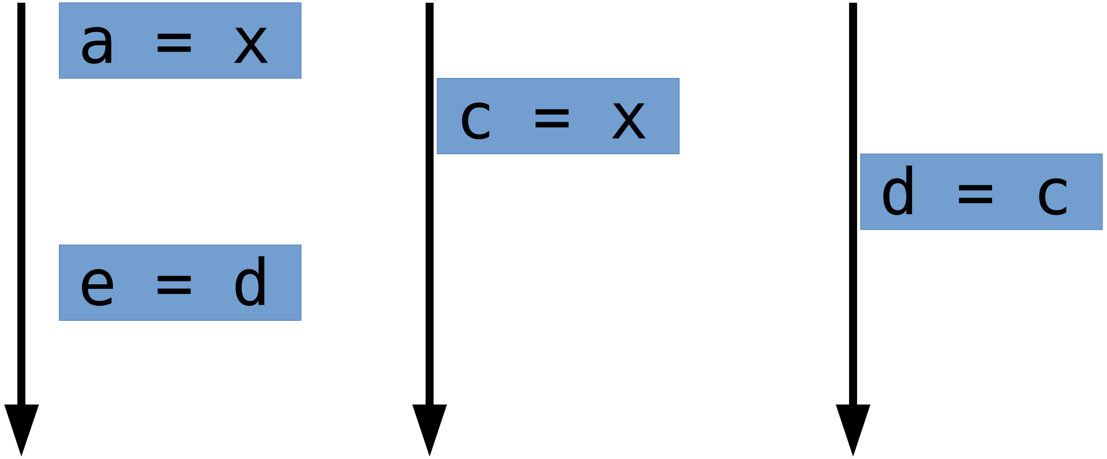
Only some patterns are unserializable.
Detect unlikely issues via training.

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]**

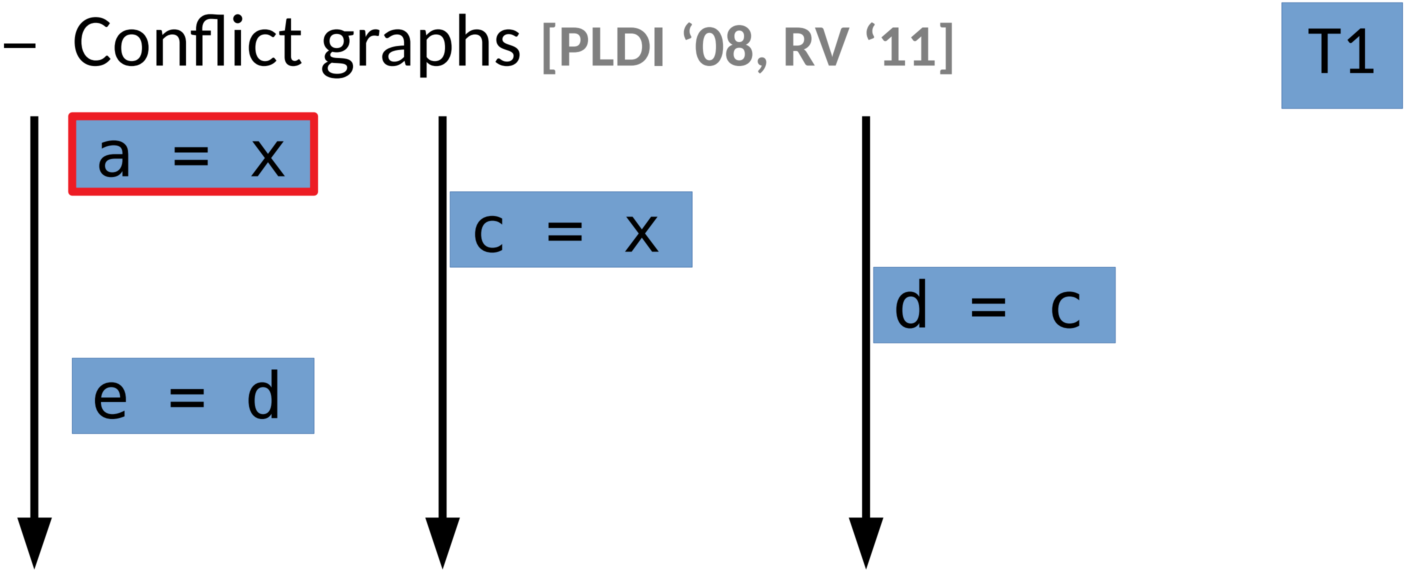
Atomicity Violations

- 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]



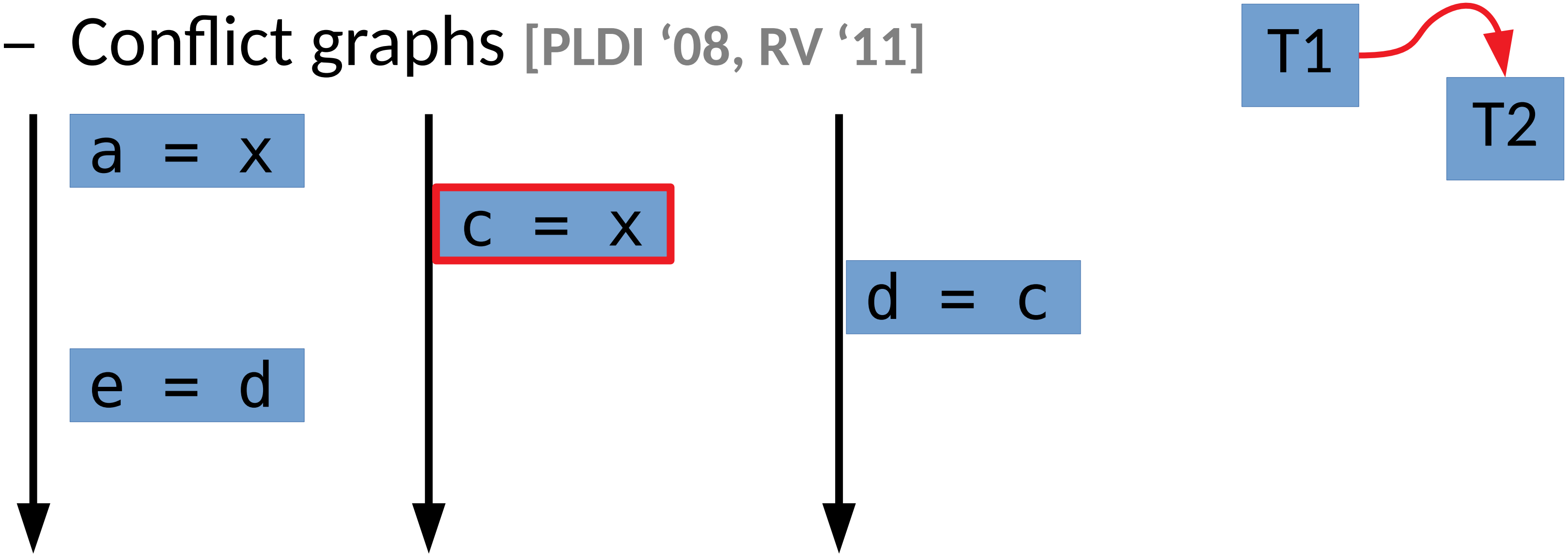
Atomicity Violations

- 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]



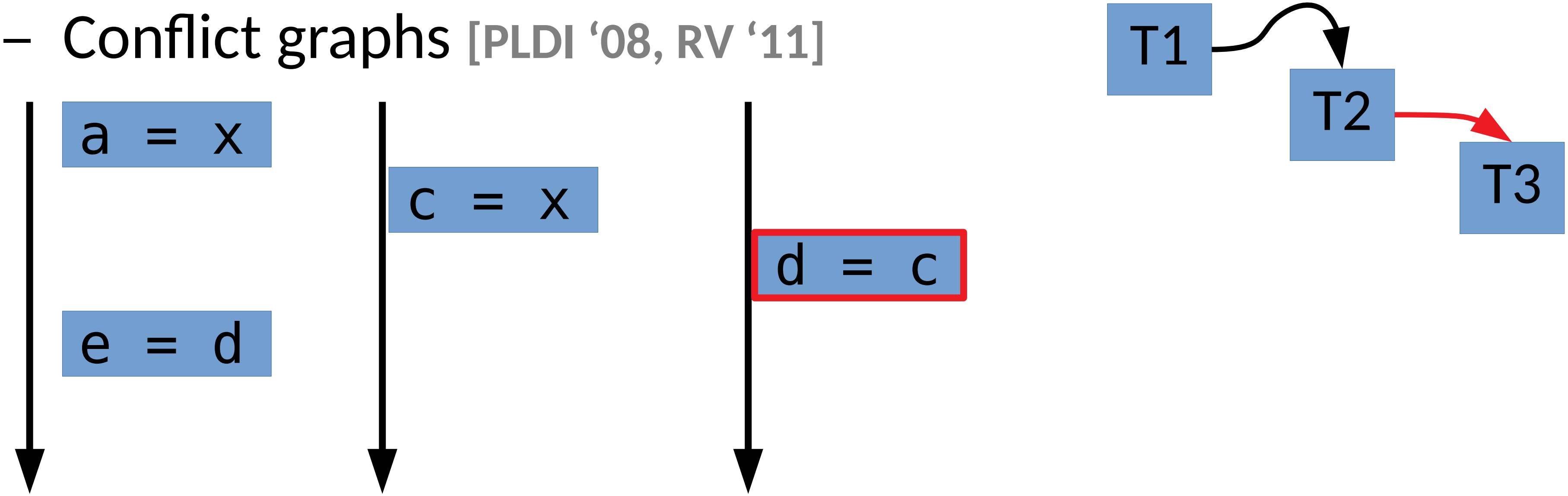
Atomicity Violations

- 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]



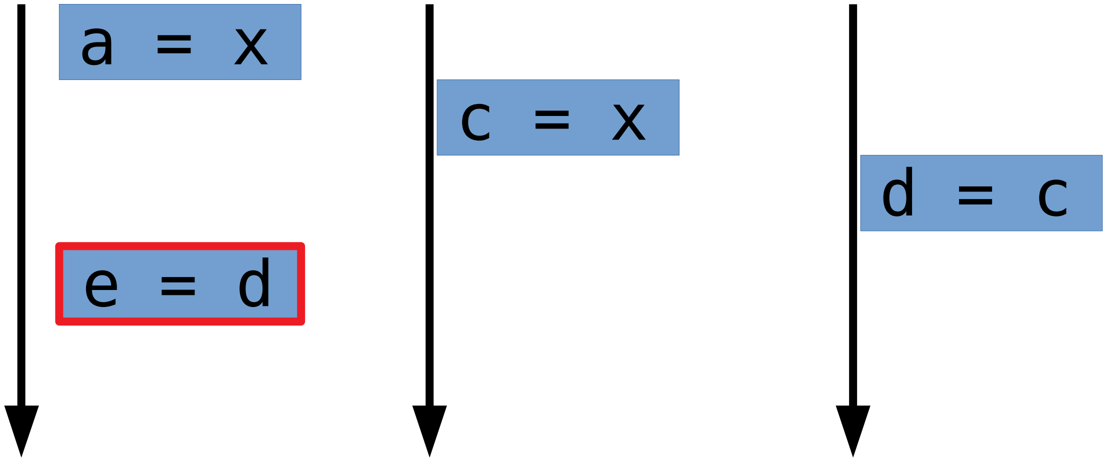
Atomicity Violations

- 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]**

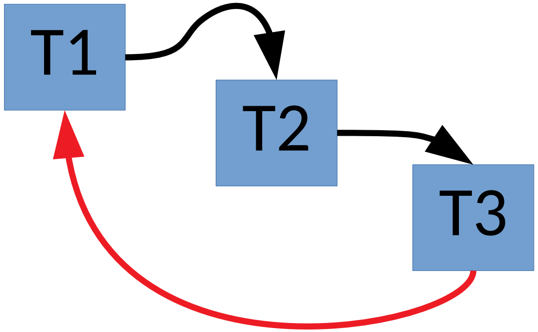


a = x

c = x

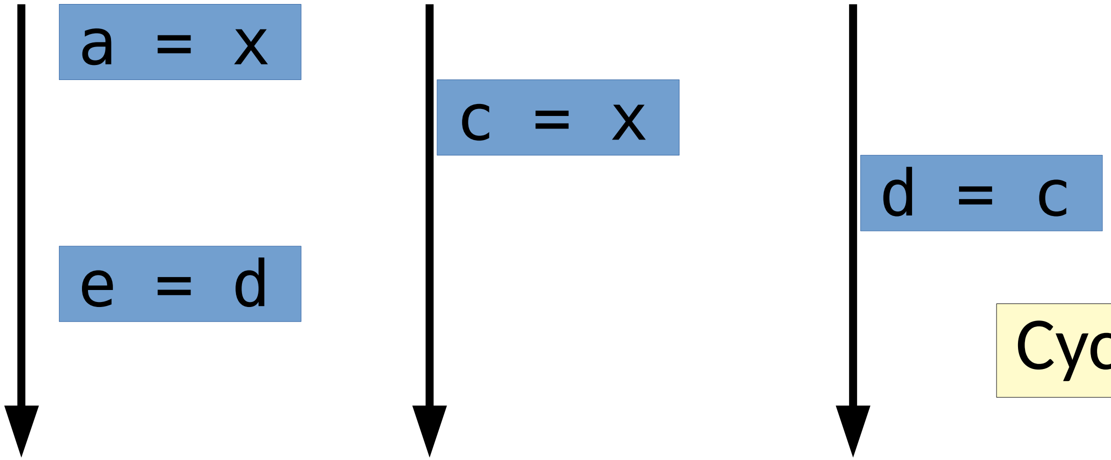
d = c

e = d



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]**

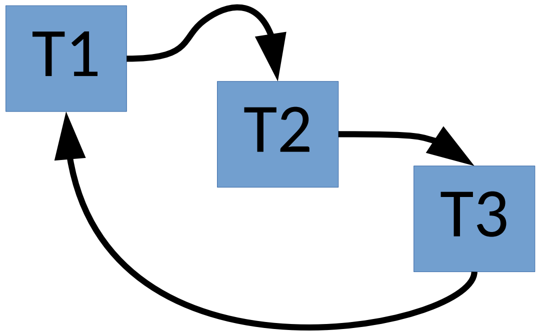


a = x

c = x

d = c

e = d



T1

T2

T3

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?

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.