

May 2017  
Shonan Meeting

# A Promising Semantics for Relaxed-Memory Concurrency

**Seoul National University  
(Korea)**



Jeehoon Kang  
**Chung-Kil Hur**

**MPI-SWS  
(Germany)**



MAX-PLANCK-GESELLSCHAFT

Ori Lahav  
Viktor Vafeiadis  
Derek Dreyer

# A Message Passing Example: No Data Race

Initially:  $D = F = 0$

**D = 42**

**LOCK(L)**

**F = 1**

**UNLOCK(L)**

**while (1) {**

**LOCK(L)**

**f = F**

**UNLOCK(L)**

**if (f) break**

**}**

**d = D**

Finally:  $d = 42$

# Sequentially Consistent Concurrency

Initially:  $D = F = 0$

**D = 42**

**F = 1**

```
while (1) {
```

```
    f = F
```

```
    if (f) break
```

```
}
```

```
d = D
```

Finally:  $d = 42$

# Relaxed-Memory Concurrency

Initially: `D = F = 0`

`F = 1`      `D = 42`

 **HW out of order exec**

`D = 42`      `F = 1`

```
while (1) {  
    f = F  
  
    if (f) break  
}  
  
d = D
```

Finally: `d = 42 or 0`

# Release & Acquire

Initially:  $D = F = 0$

$D = 42$

$F = 1$  [rel]

`while (1) {`

`f = F` [acq]

`if (f) break`

`}`

`d = D`

Finally:  $d = 42$

# Release & Acquire

Initially:  $D = F = 0$

$D = 42$

$F = 1$  [rel]

Run as if  
in a single thread

while (1) {

$f = F$  [acq]

if (f) break

}

$d = D$

Finally:  $d = 42$

# Release & Acquire with Tweak

Initially:  $D = F = 0$

$D = 42$

$F = 1$  [rel]

$f = F$  [acq]

```
if (f) {  
    d = D // d = 42?  
}
```

$D = 10$

# Concurrency Models

- **Semantics of multi-threaded programs?**
  - Sequential consistency (SC): simple but **expensive**
- **Relaxed memory model (C/C++, Java)**
  - Many consistency modes (cost vs. consistency tradeoff)
  - **Open problem: what is the “right” semantics?**



# “Right” Concurrency Semantics?

**Conflicting goals** of compilers, hardware & programmers

- **Compiler/hardware:** validating optimizations (e.g. reordering, merging)
- **Programmer:** supporting reasoning principles (e.g. DRF theorem, program logic)

# “Right” Concurrency Semantics?

**Conflicting goals** of compilers, hardware & programmers



**Java memory model**

- **Compiler/hardware:** validating optimizations (e.g. reordering, merging)
- **Programmer:** supporting reasoning principles (e.g. DRF theorem, program logic)

# “Right” Concurrency Semantics?

**Conflicting goals** of compilers, hardware & programmers



**Java memory model**

- **Compiler/hardware:** validating optimizations (e.g. reordering, merging)
- **Programmer:** supporting reasoning principles (e.g. DRF theorem, program logic)



**C/C++ memory model**

# “Right” Concurrency Semantics?

**Conflicting goals** of compilers, hardware & programmers



**Java memory model**

- **Compiler/hardware:** validating optimizations (e.g. reordering, merging)
- **Programmer:** supporting reasoning principles (e.g. DRF theorem, program logic)



**C/C++ memory model**

Key problem: “out-of-thin-air”

# “Out-of-thin-air” problem (1/3)

## Load-Buffering (LB)

<b>Thread 1</b>	<b>Thread 2</b>
$a = X$	$b = Y$
$Y = a$	$X = 42$
(allowed: $a=b=42$ )	

# "Out-of-thin-air" problem (1/3)

## Load-Buffering (LB)

Registers

**Thread 1**

$a = X$

$Y = a$

**Thread 2**

$b = Y$

$X = 42$

(allowed:  $a=b=42$ )

# "Out-of-thin-air" problem (1/3)

## Locking (LB)

Registers

Shared  
Locations

**Thread 1**      **Thread 2**

$a = X$

$b = Y$

$Y = a$

$X = 42$

(allowed:  $a=b=42$ )

# "Out-of-thin-air" problem (1/3)

## Localizing (LB)

Registers

Shared  
Locations

**Thread 1**

**Thread 2**

a = X

b = Y

Y = a

X = 42

C11  
Relaxed

(allowed: a=b=42)



# “Out-of-thin-air” problem (1/3)

## Load-Buffering (LB)

<b>Thread 1</b>	<b>Thread 2</b>
$a = X$	$b = Y$
$Y = a$	$X = 42$
(allowed: $a=b=42$ )	

# "Out-of-thin-air" problem (1/3)

## Load-Buffering (LB)

**Thread 1**    **Thread 2**  
a = X            b = Y  
Y = a            X = 42  
(allowed: a=b=42)

Allowed by reordering  
(Power/ARM)

X = 42  
b = Y

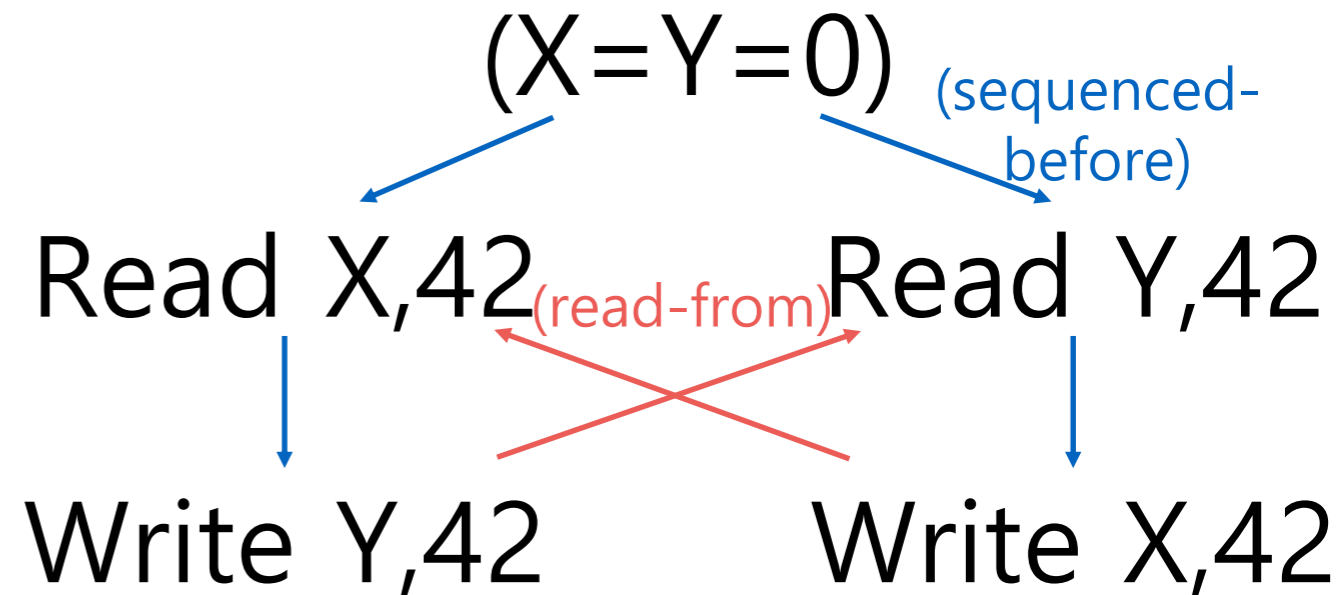
# "Out-of-thin-air" problem (1/3)

## Load-Buffering (LB)

**Thread 1**      **Thread 2**  
a = X              b = Y  
Y = a              X = 42  
(allowed: a=b=42)

Allowed by reordering  
(Power/ARM)

X = 42  
b = Y



Allowed by justification  
(C/C++)

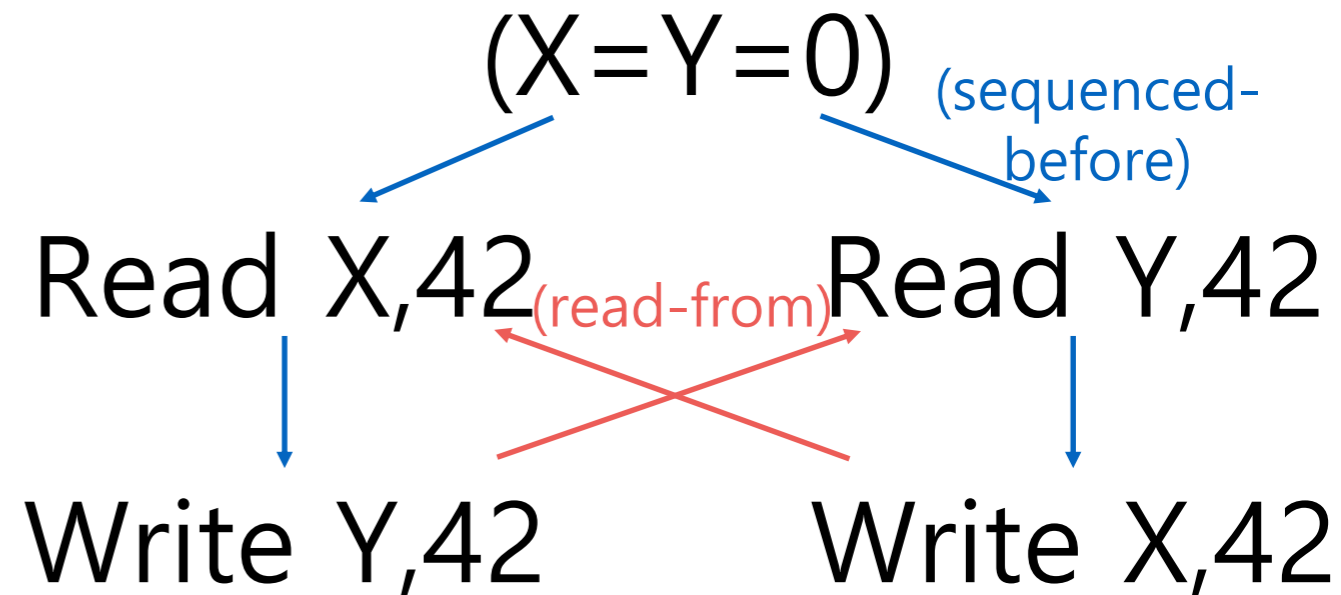
# "Out-of-thin-air" problem (1/3)

## Load-Buffering (LB)

**Thread 1**    **Thread 2**  
a = X            b = Y  
Y = a            X = 42  
(allowed: a=b=42)

Allowed by reordering  
(Power/ARM)

X = 42  
b = Y



Allowed by justification  
(C/C++)

Justification is too loose!

# "Out-of-thin-air" problem (2/3)

## Classical Out-of-thin-air (OOTA)

Thread 1	Thread 2
$a = X$	$b = Y$
$Y = a$	$X = b$
(forbidden: $a=b=42$ )	

# "Out-of-thin-air" problem (2/3)

## Classical Out-of-thin-air (OOTA)

Thread 1	Thread 2
$a = X$	$b = Y$
$Y = a$	$X = b$

(forbidden:  $a=b=42$ )

42 is out-of-thin-air!

Reasoning principles  
(e.g. invariant  $a=b=X=Y=0$ )

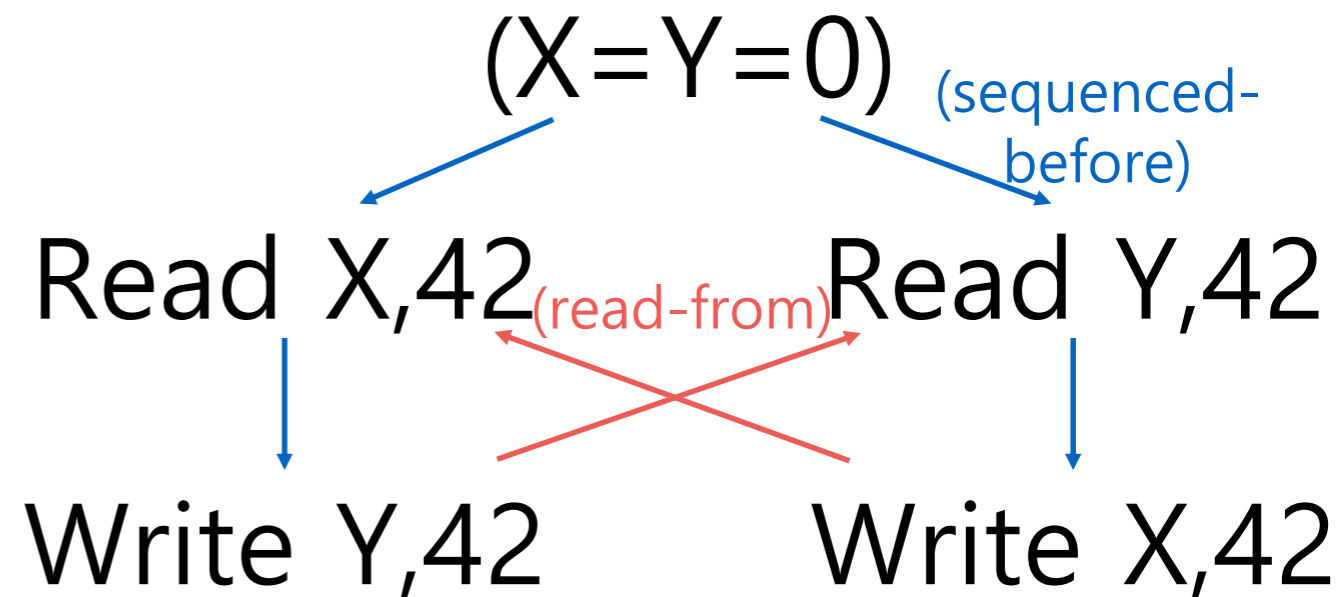
# "Out-of-thin-air" problem (2/3)

## Classical Out-of-thin-air (OOTA)

**Thread 1**      **Thread 2**  
a = X              b = Y  
Y = a              X = b  
(forbidden: a=b=42)

42 is out-of-thin-air!

Reasoning principles  
(e.g. invariant  $a=b=X=Y=0$ )



Allowed by justification  
(C/C++)

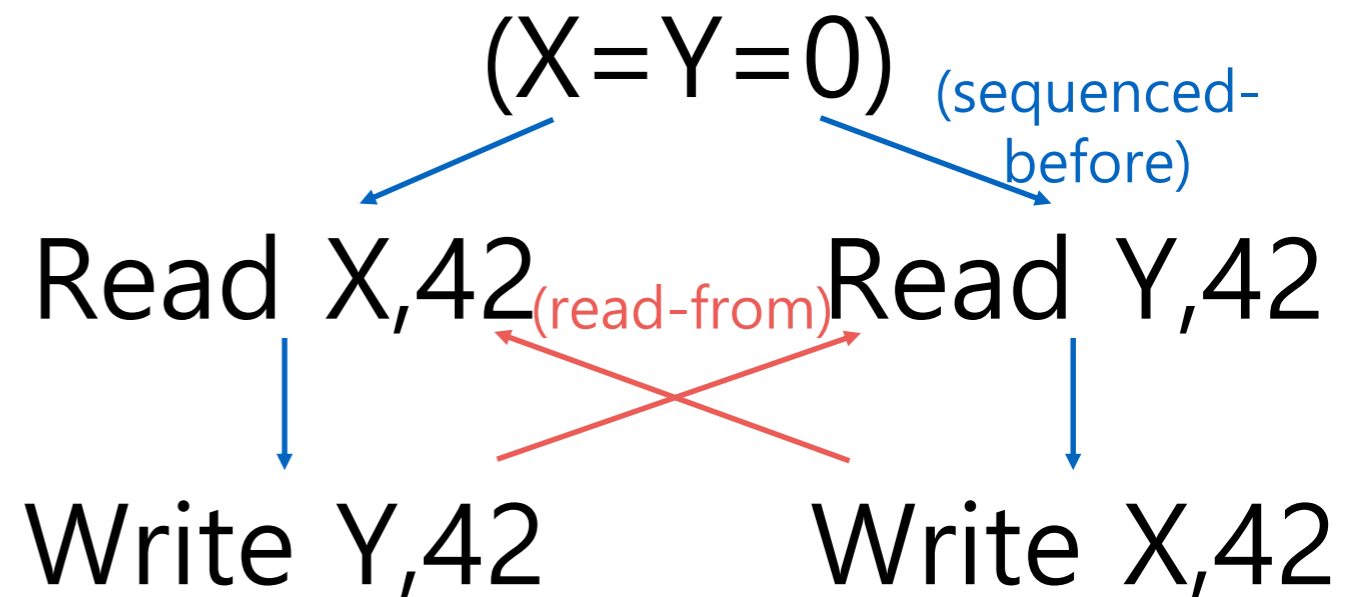
# "Out-of-thin-air" problem (2/3)

## Classical Out-of-thin-air (OOTA)

**Thread 1**      **Thread 2**  
a = X              b = Y  
Y = a              X = b  
(forbidden: a=b=42)

42 is out-of-thin-air!

Reasoning principles  
(e.g. invariant  $a=b=X=Y=0$ )



Allowed by justification  
(C/C++)

What does hardware do?



# “Out-of-thin-air” problem (3/3)

## Tracking Syntactic Dependency?

**Thread 1**      **Thread 2**

$a = X$

$b = Y$

$Y = a$

$X = 42$

( $a=b=42?$ )

**Thread 1**      **Thread 2**

$a = X$

$b = Y$

$Y = a$

$X = b$

( $a=b=42?$ )

# "Out-of-thin-air" problem (3/3)

## Tracking Syntactic Dependency?

**Thread 1**      **Thread 2**

a = X

b = Y

Y = a

X = 42

(a=b=42?)

**Thread 1**      **Thread 2**

a = X

~~b = Y~~

Y = a

~~X = b~~ (dep)

(a=b=42?)

# "Out-of-thin-air" problem (3/3)

## Tracking Syntactic Dependency?

### Thread 1

$a = X$   
 $Y = a$

( $a=b=42$ ?)

### Thread 2

$b = Y$   
 $X = 42$

allowed  
in hardware

### Thread 1

$a = X$   
 $Y = a$

( $a=b=42$ ?)

### Thread 2

~~$b = Y$   
 $X = b$~~  (dep)

forbidden  
in hardware

# "Out-of-thin-air" problem (3/3)

## Tracking Syntactic Dependency?

**Thread 1**

$a = X$

$Y = a$

( $a=b=42?$ )

**Thread 2**

$b = Y$

$X = b+42-b$

**Thread 1**

$a = X$

$Y = a$

( $a=b=42?$ )

**Thread 2**

~~$b = Y$~~

~~$X = b$~~  (dep)

forbidden  
in hardware

# "Out-of-thin-air" problem (3/3)

## Tracking Syntactic Dependency?

**Thread 1**

$a = X$

$Y = a$

( $a=b=42?$ )

**Thread 2**

~~$b = Y$~~

~~$X = b + 42 - b$~~

forbidden  
in hardware

**Thread 1**

$a = X$

$Y = a$

( $a=b=42?$ )

**Thread 2**

~~$b = Y$~~

~~$X = b$~~  (dep)

forbidden  
in hardware

# "Out-of-thin-air" problem (3/3)

## Tracking Syntactic Dependency?

**Thread 1**

$a = X$

$Y = a$

( $a=b=42?$ )

**Thread 2**

~~$b = Y$~~

$X = b + 42 - b$

forbidden  
in hardware

**Thread 1**

$a = X$

$Y = a$

( $a=b=42?$ )

**Thread 2**

~~$b = Y$~~

$X = b$  (dep)

forbidden  
in hardware

could be optimized to "42",  
**should be allowed in PL**

# "Out-of-thin-air" problem (3/3)

## Tracking Syntactic Dependency?

**Thread 1**  
 $a = X$   
 $Y = a$   
( $a=b=42?$ )

**Thread 2**

~~$b = Y$~~   
 $X = b + 42 - b$

forbidden  
in hardware

**Thread 1**

$a = X$   
 $Y = a$

( $a=b=42?$ )

**Thread 2**

~~$b = Y$~~   
 $X = b$  (dep)

forbidden  
in hardware

could be optimized to "42",  
**should be allowed in PL**

Syntactic approach  
doesn't work for PL!

# “Out-of-thin-air” problem (3/3)

“A major open problem for PL semantics”  
(Batty et al. ESOP 2015)

## Thread 1

$a = X$   
 $Y = a$

( $a=b=42?$ )

## Thread 2

~~$b = Y$~~   
 $X = b + 42 - b$

forbidden  
in hardware

## Thread 1

$a = X$   
 $Y = a$

( $a=b=42?$ )

## Thread 2

~~$b = Y$~~   
 $X = b$  (dep)

forbidden  
in hardware

could be optimized to “42”,  
**should be allowed in PL**

Syntactic approach  
doesn't work for PL!



# Promising Semantics

- Solving the **out-of-thin-air problem**
- Supporting **optimizations & reasoning principles**
- Covering most **C/C++ concurrency** features
- **Operational semantics w/o undefined behavior**

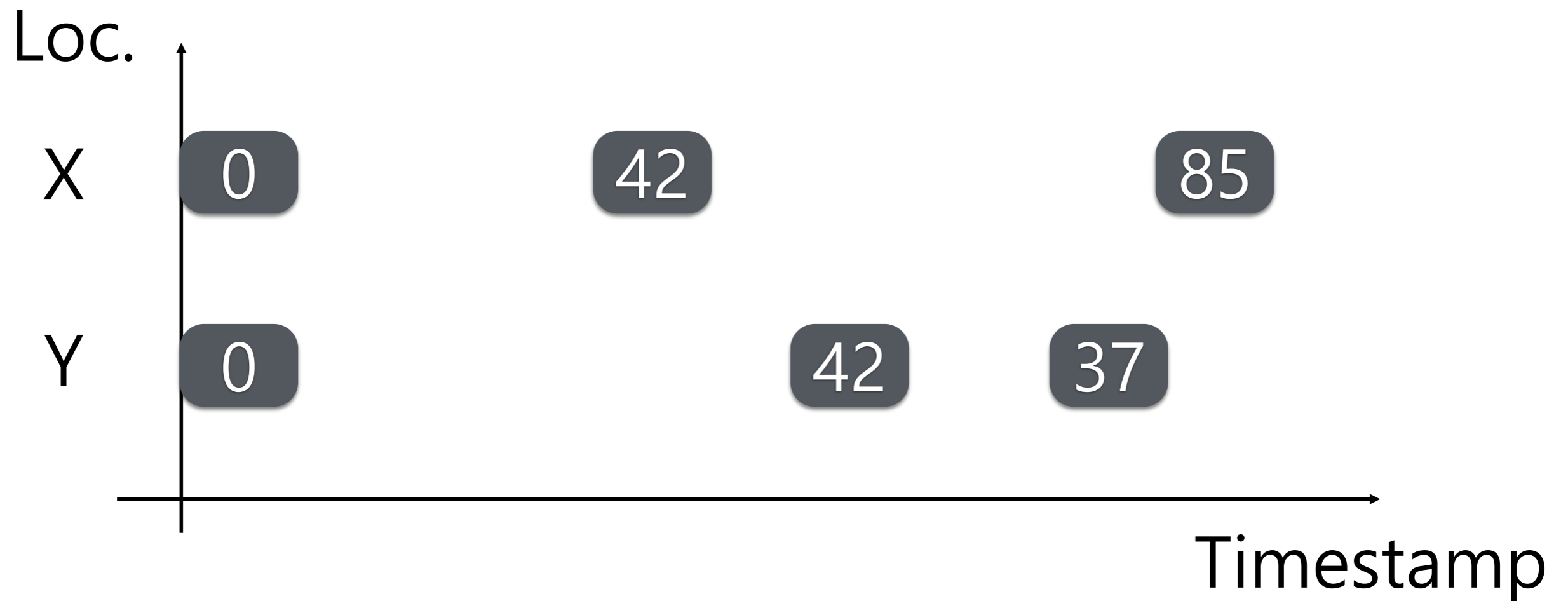
- Most results are **verified in Coq**



<http://sf.snu.ac.kr/promise-concurrency>

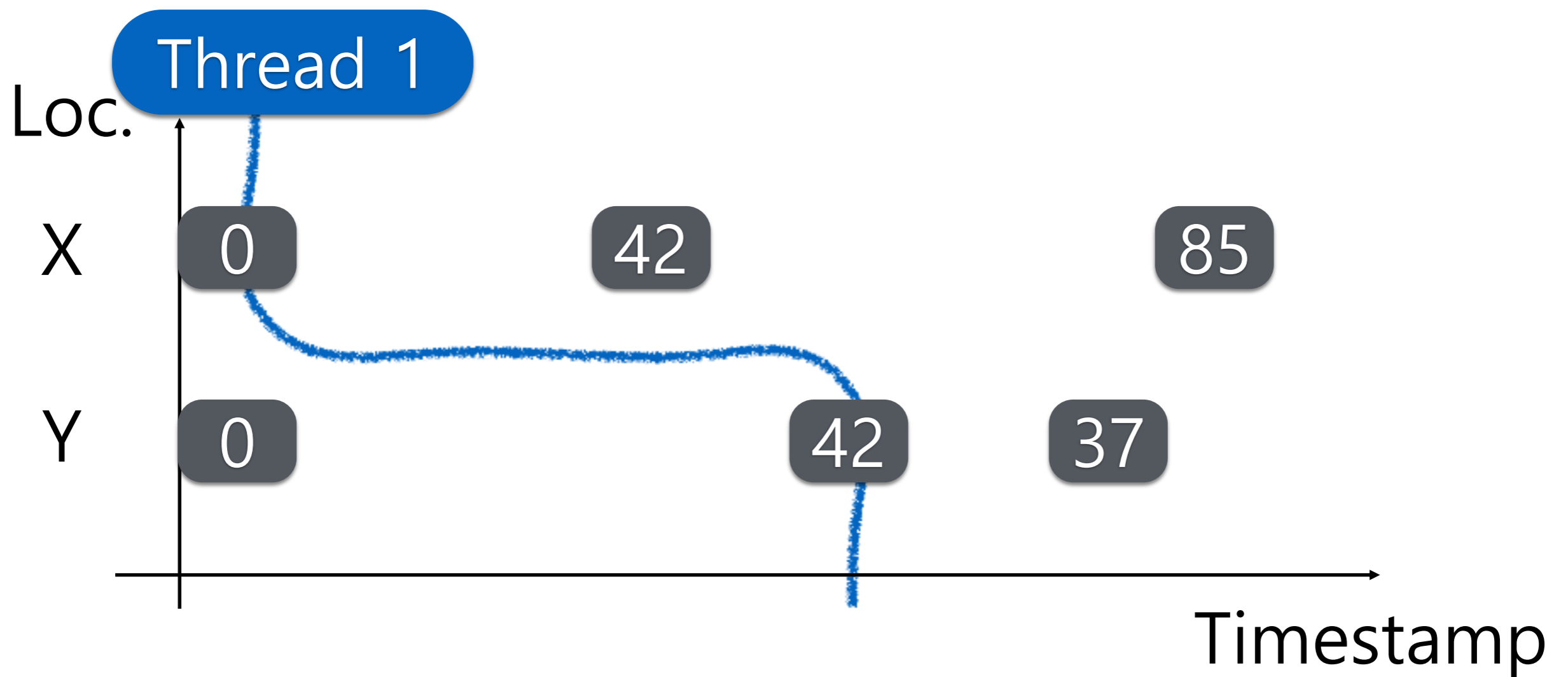
# Key Idea 1: Messages & Views

- Memory: pool of **messages** (loc, val, timestamp)
- Per-thread **view** on the memory



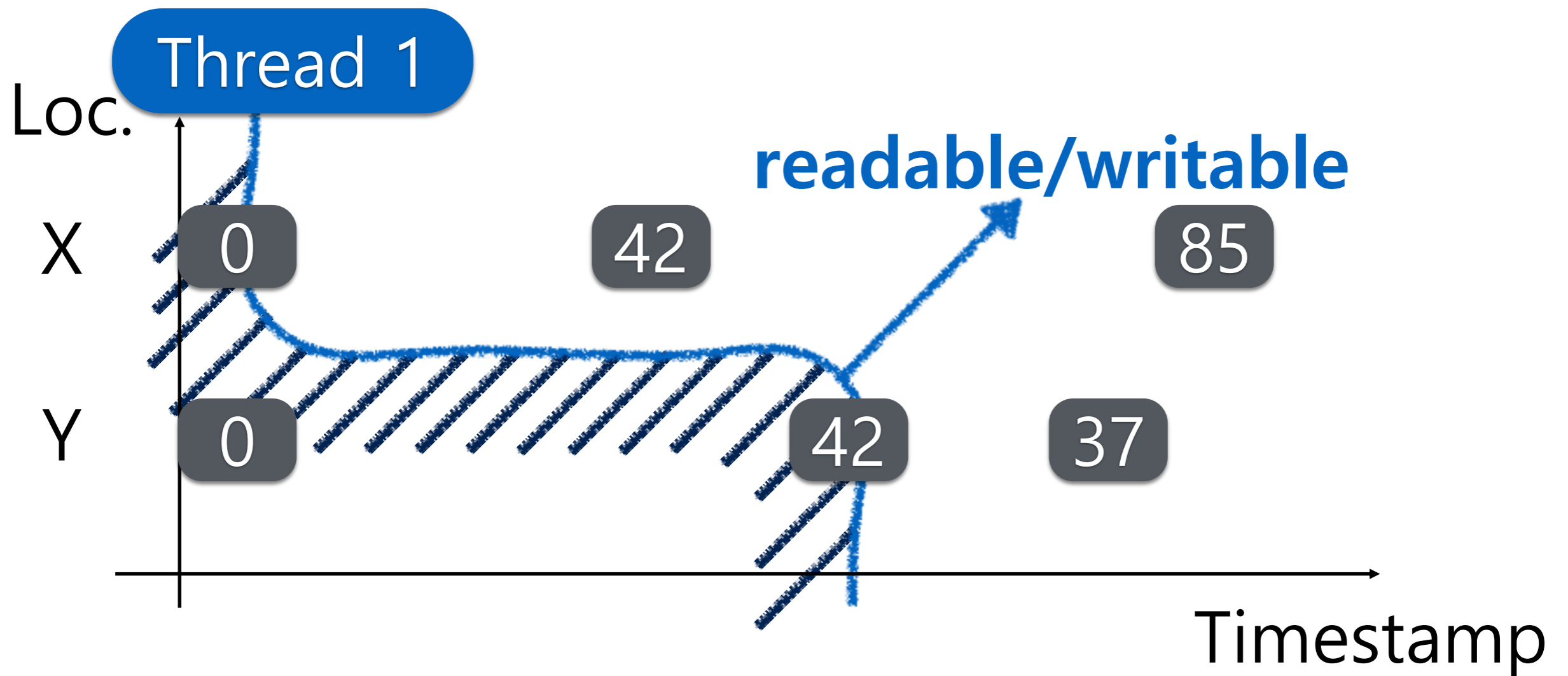
# Key Idea 1: Messages & Views

- Memory: pool of **messages** (loc, val, timestamp)
- Per-thread **view** on the memory



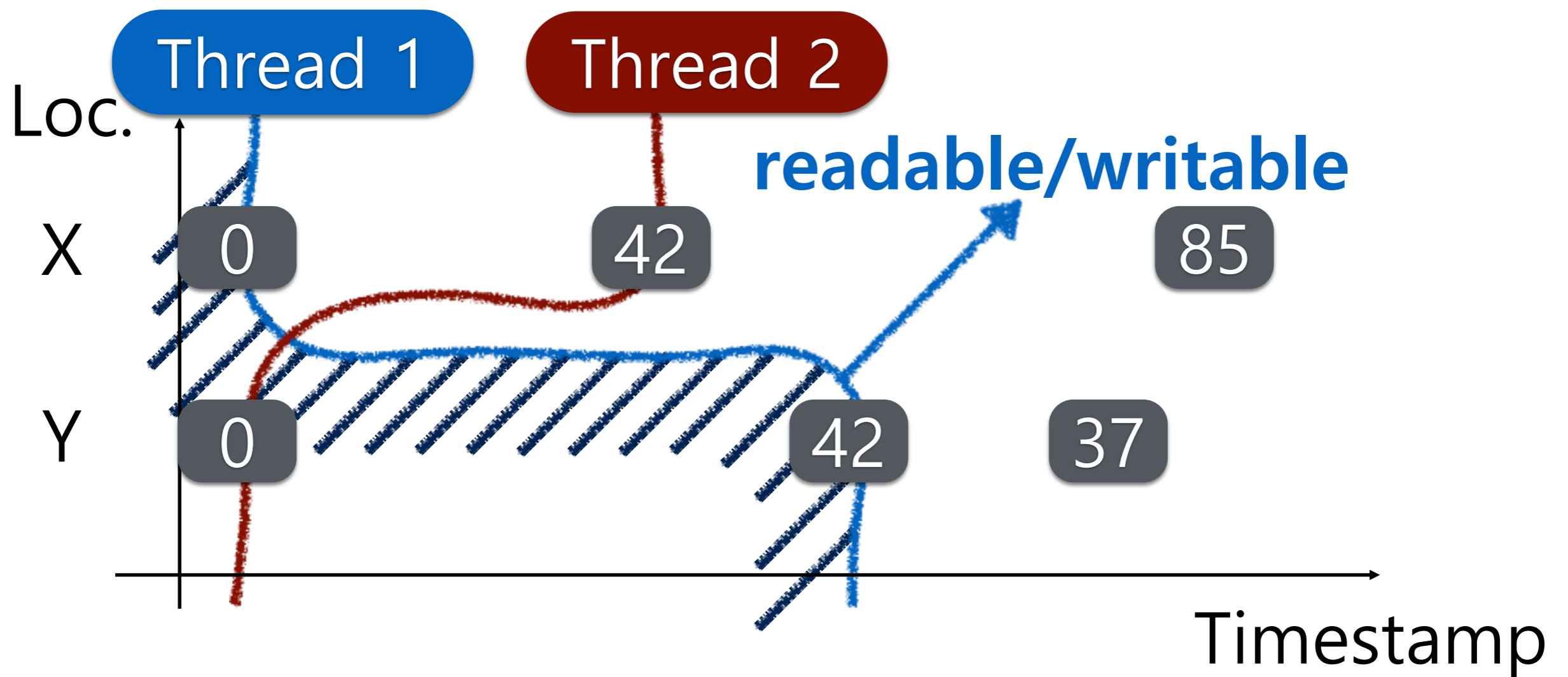
# Key Idea 1: Messages & Views

- Memory: pool of **messages** (loc, val, timestamp)
- Per-thread **view** on the memory



# Key Idea 1: Messages & Views

- Memory: pool of **messages** (loc, val, timestamp)
- Per-thread **view** on the memory



# Example

## Store Buffering

**Thread 1**



$Y = 42$

$a = X$

**Thread 2**



$X = 42$

$b = Y$

(allowed:  $a=b=0$ )



# Example Store Buffering

**Thread 1**



$Y = 42$   
 $a = X$

**Thread 2**



$X = 42$   
 $b = Y$

(allowed:  $a=b=0$ )



**reorderable**

(x86/Power/ARM)

$b = Y$   
 $X = 42$



# Example Store Buffering

## Thread 1

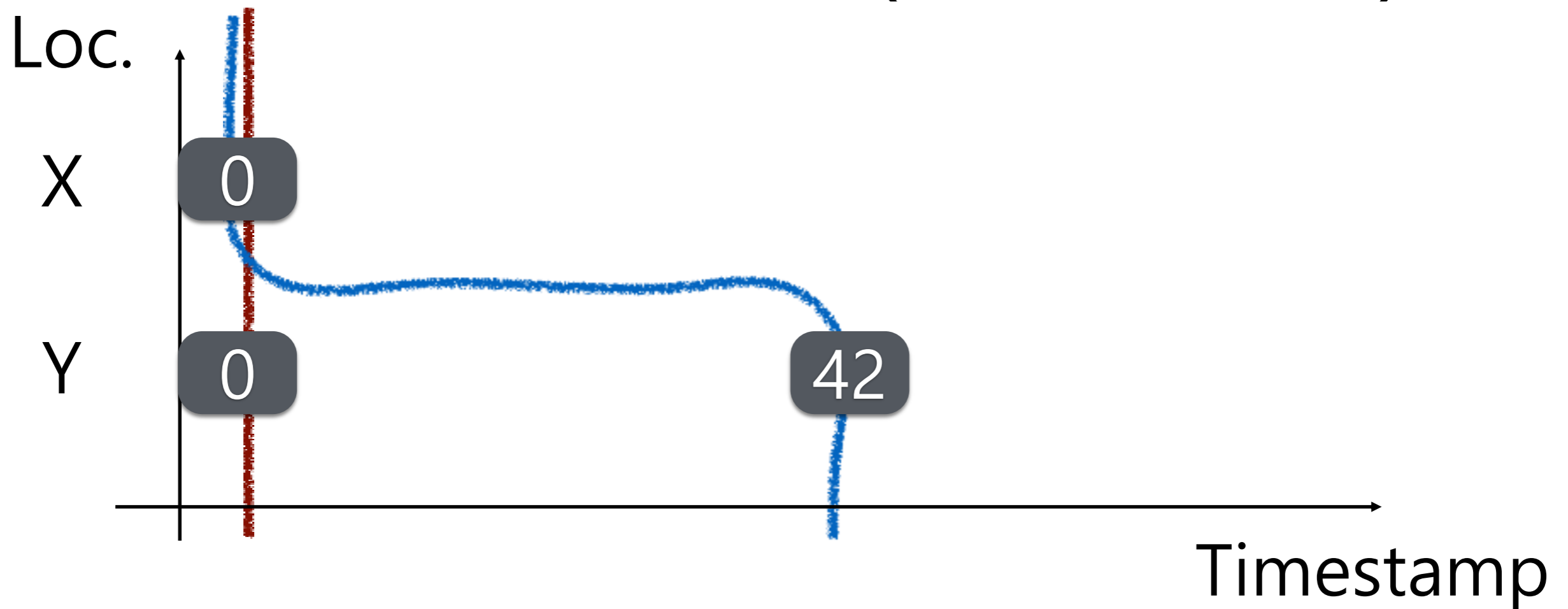
→  $Y = 42$   
 $a = X$

## Thread 2

→  $X = 42$   
 $b = Y$

(allowed:  $a=b=0$ )

→  $b = Y$   
 $X = 42$   
**reorderable**  
(x86/Power/ARM)





# Example Store Buffering

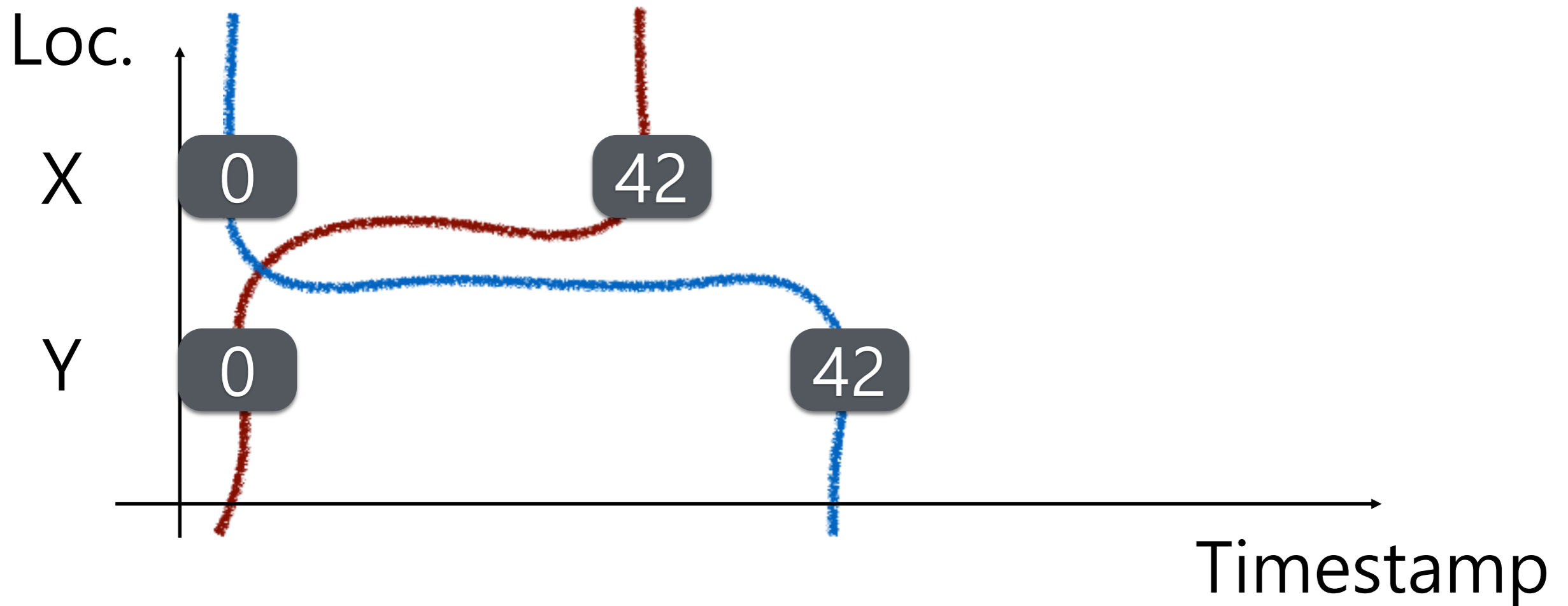
## Thread 1

→  $Y = 42$   
 $a = X$

## Thread 2

→  $X = 42$   
 $b = Y$

(allowed:  $a=b=0$ )  
→  $b = Y$   
 $X = 42$   
**reorderable**  
(x86/Power/ARM)



# Example Store Buffering

## Thread 1

$Y = 42$   
 $a = X$



## Thread 2

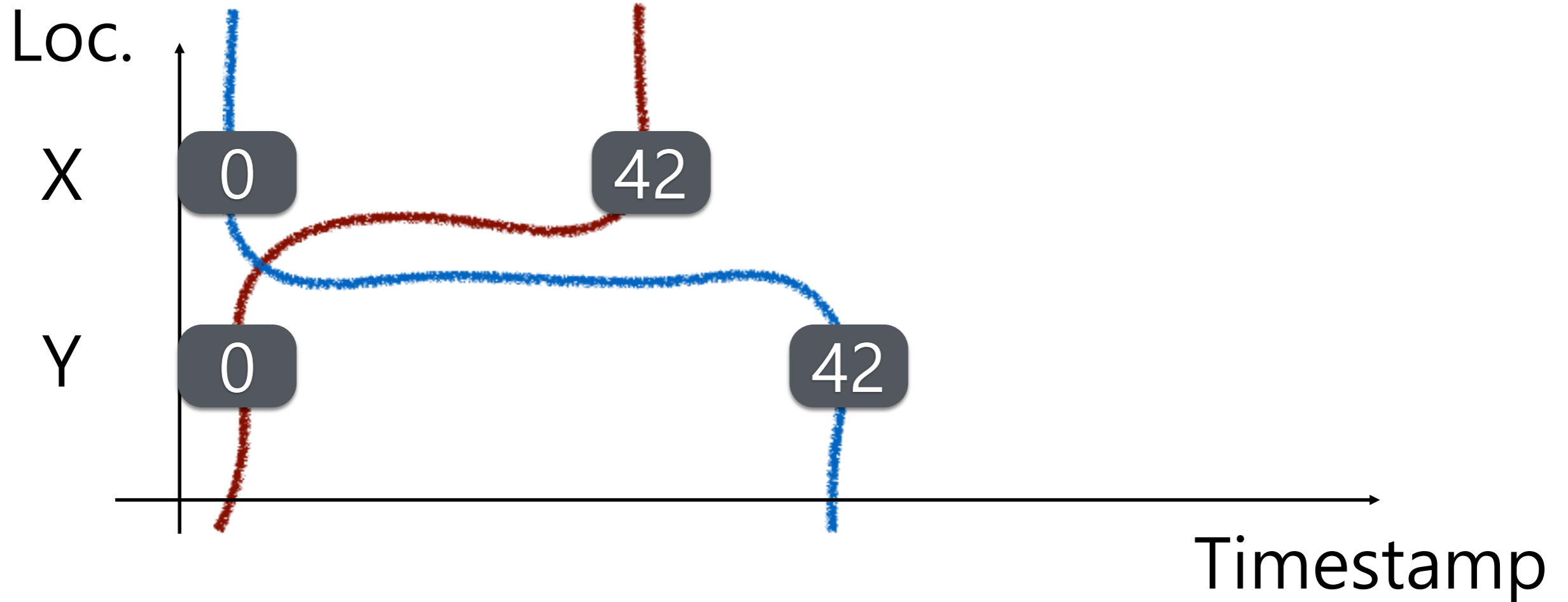
$X = 42$   
 $b = Y$



(allowed:  $a=b=0$ )

**reorderable**  
(x86/Power/ARM)

$b = Y$   
 $X = 42$



# Example Store Buffering

## Thread 1

$Y = 42$

$a = X$



## Thread 2

$X = 42$

$b = Y$



(allowed:  $a=b=0$ )

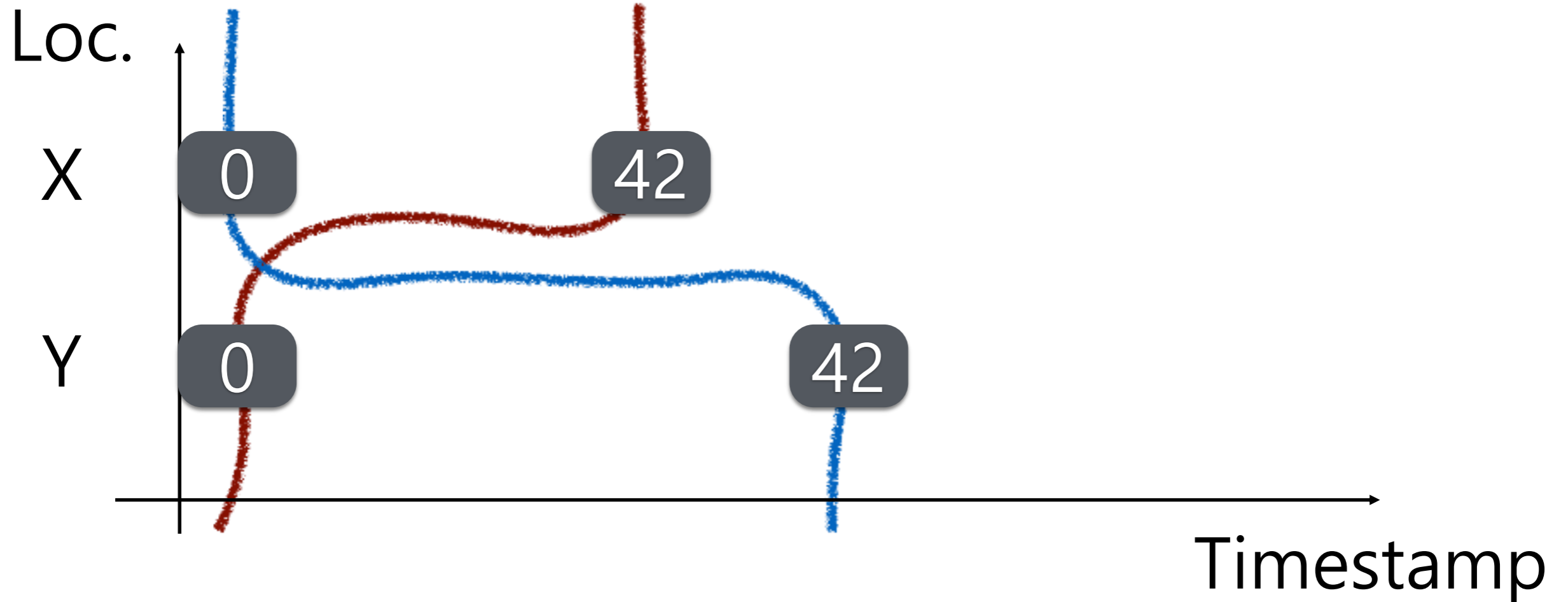
$b = Y$

$X = 42$



**reorderable**

(x86/Power/ARM)



# Example

## Load Buffering (LB)

**Thread 1**



$a = X$

$Y = a$

**Thread 2**



$b = Y$

$X = 42$

(allowed:  $a=b=42$ )



# Key Idea 2: Promises

- A thread can **promise** to write  $X=V$  in the future, after which **other threads can read**  $X=V$ .
- To avoid OOTA, the promising thread must **certify** that it can write  $X=V$  **in isolation**.
- Until all its promises are fulfilled, the thread can take **certifiable steps** only.

# Example Load Buffering (LB)

**Thread 1**



$a = X$

$Y = a$

**Thread 2**



$b = Y$

$X = 42$

(allowed:  $a=b=42$ )



# Example Load Buffering (LB)

**Thread 1**



$a = X$

$Y = a$

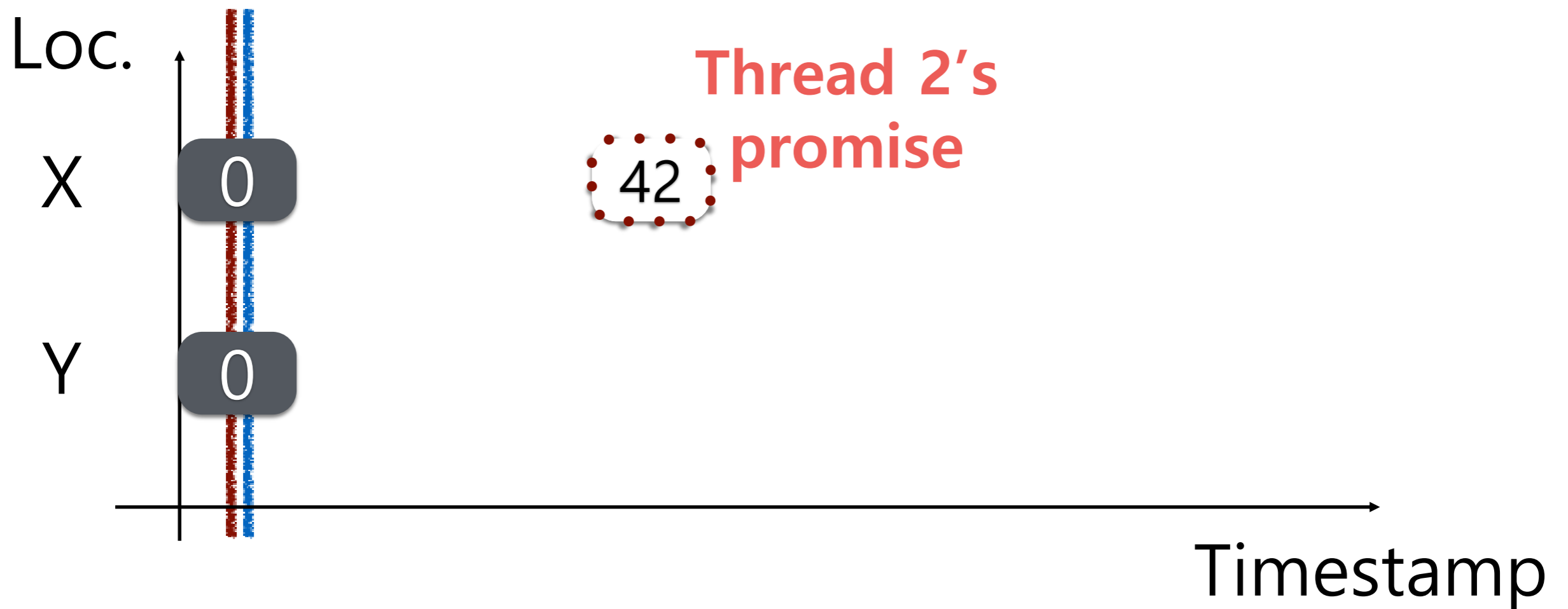
**Thread 2**



$b = Y$

$X = 42$

(allowed:  $a=b=42$ )



# Example Load Buffering (LB)

**Thread 1**



$a = X$

$Y = a$

**Thread 2**



$b = Y$

$X = 42$

(allowed:  $a=b=42$ )

Loc.

X

0

Y

0

Thread 2's  
promise

42

Thread 2 should be  
able to write it  
in isolation

Timestamp



# Example Certification

## Thread 2



$b = Y$

$X = 42$



# Example Certification

## Thread 2

→  $b = Y$   
 $X = 42$



# Example Certification

## Thread 2

$b = Y$

$X = 42$



# Example Certification

## Thread 2

$b = Y$

$X = 42$



# Example Load Buffering (LB)

**Thread 1**



$a = X$

$Y = a$

**Thread 2**



$b = Y$

$X = 42$

(allowed:  $a=b=42$ )

Loc.

X

0

Y

0

Thread 2's  
promise

42

Promise is  
certified!

Timestamp

# Example Load Buffering (LB)

**Thread 1**

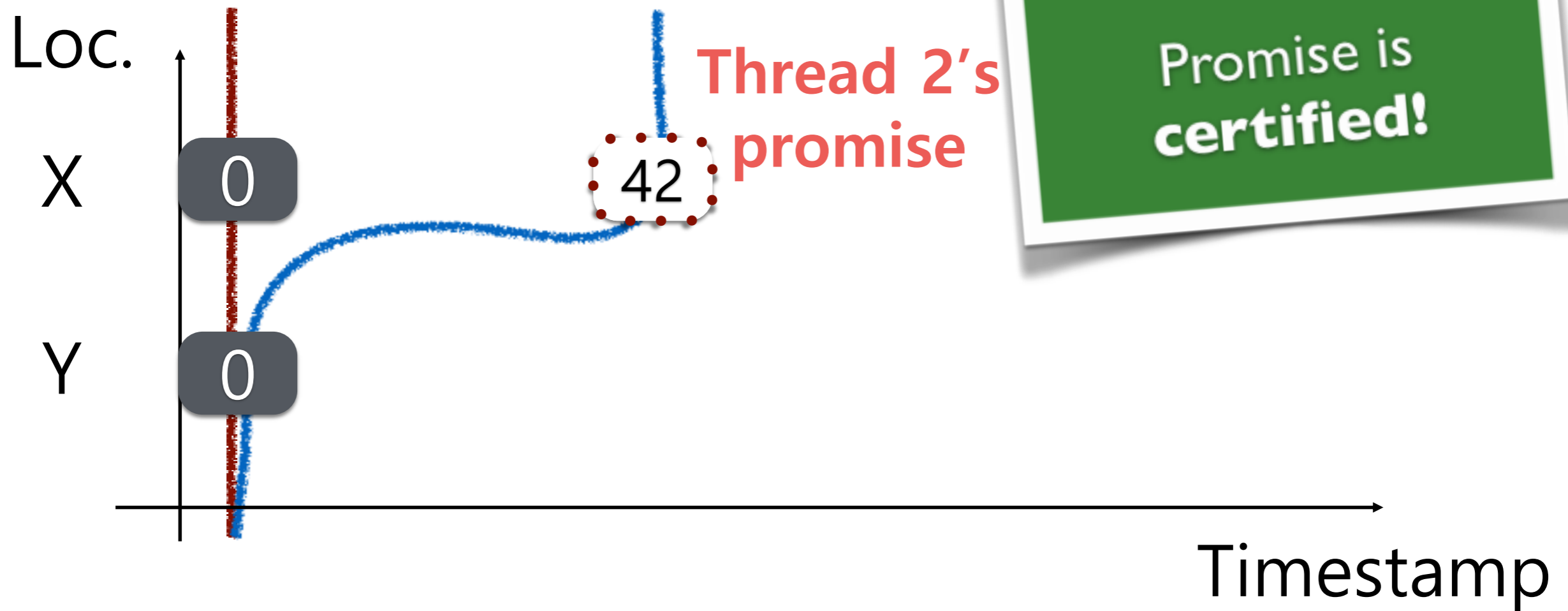
→  $a = X$   
 $Y = a$

**Thread 2**



$b = Y$   
 $X = 42$

(allowed:  $a=b=42$ )



# Example Load Buffering (LB)

**Thread 1**

$a = X$

$\rightarrow Y = a$

**Thread 2**



$b = Y$

$X = 42$

(allowed:  $a=b=42$ )

Loc.

X

0

Y

0

Thread 2's  
promise

42

42

Promise is  
certified!

Timestamp

# Example Load Buffering (LB)

**Thread 1**

$a = X$

$Y = a$



**Thread 2**

$b = Y$

$X = 42$



(allowed:  $a=b=42$ )





# Example Load Buffering (LB)

**Thread 1**

$a = X$

→  $Y = a$

**Thread 2**

$b = Y$

→  $X = 42$

(allowed:  $a=b=42$ )

Loc.

X

0

42

Thread 2's  
promise

Y

0

42

Promise is  
certified!

Timestamp

# Example Load Buffering (LB)

## Thread 1

$a = X$

→  $Y = a$

## Thread 2

$b = Y$

→  $X = b + 42 - b$

false dependency  
makes no difference!

Promise is  
certified!

Loc.

X

0

42

Thread 2's  
promise

Y

0

42

Timestamp

# Example

## Classical Out-of-thin-air (OOTA)

**Thread 1**



$a = X$

$Y = a$

**Thread 2**



$b = Y$

$X = b$

(forbidden:  $a=b=42$ )



# Example

## Classical Out-of-thin-air (OOTA)

**Thread 1**



$a = X$

$Y = a$

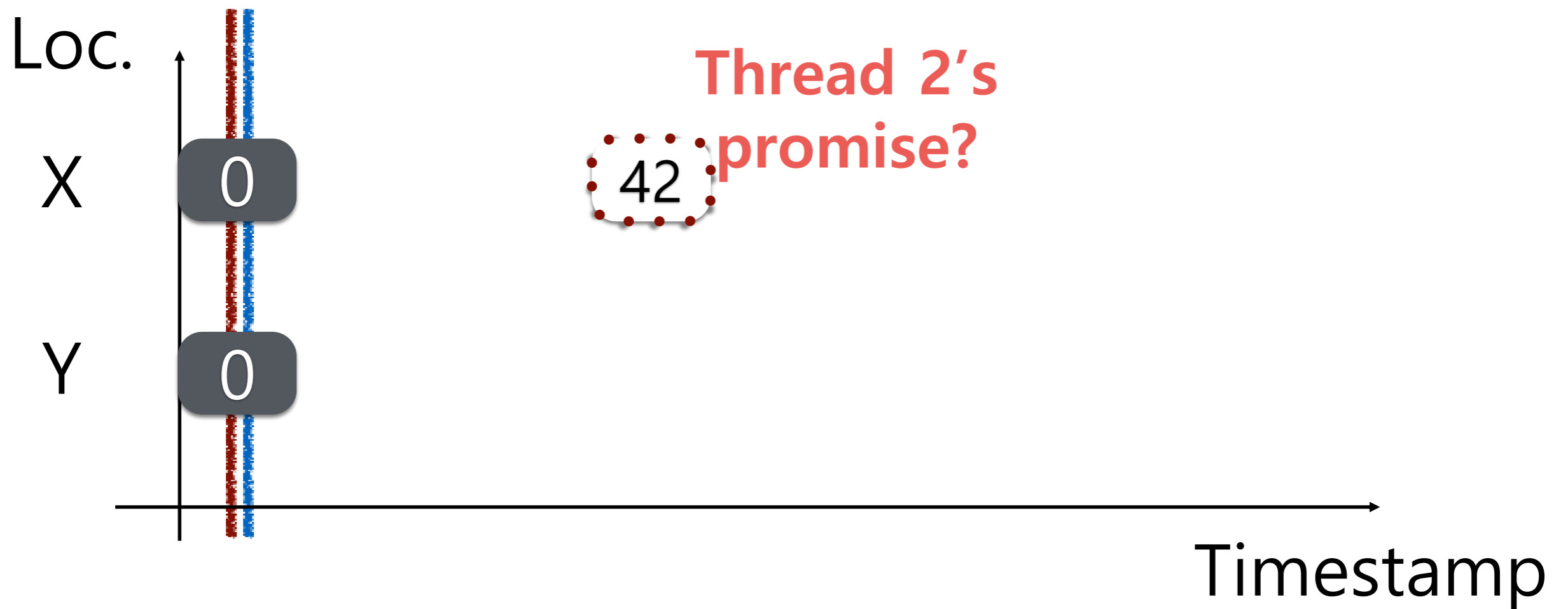
**Thread 2**



$b = Y$

$X = b$

(forbidden:  $a=b=42$ )

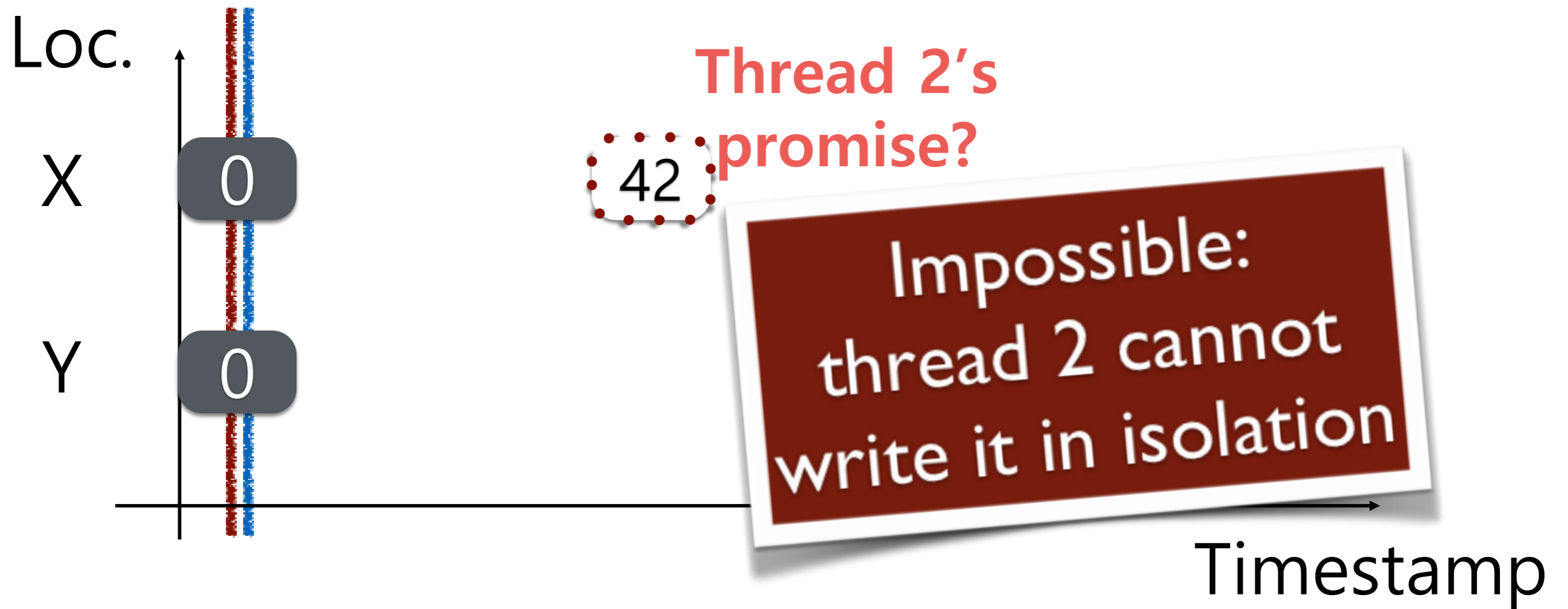


# Example

## Classical Out-of-thin-air (OOTA)

**Thread 1**  
→  
a = X  
Y = a

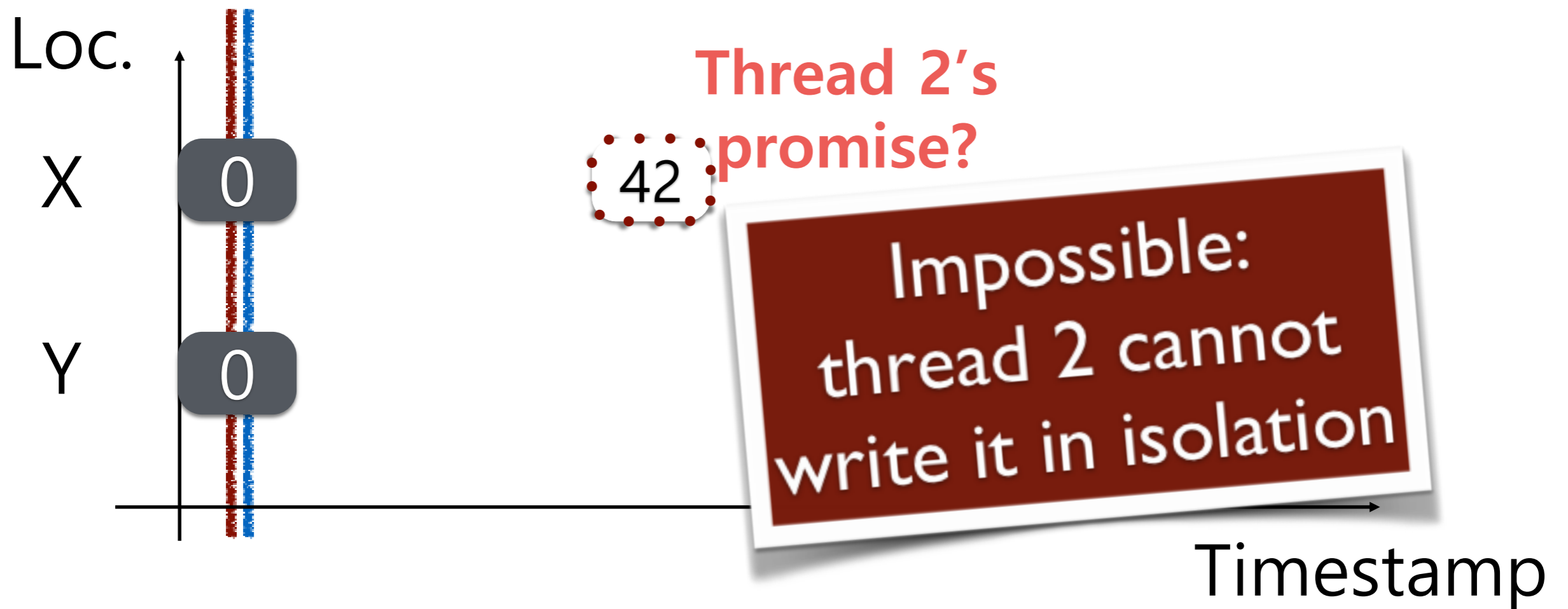
**Thread 2**  
→  
b = Y (forbidden: a=b=42)  
X = b



# Example

## Classical Out-of-thin-air (OOTA)

Promises: "Semantic Solution" to OOTA



# The Message Passing Example

## Thread 1



`D = 42`  
`F = 1`

## Thread 2



```
while (1) {  
    f = F  
    if (f) break  
}  
d = D
```



# The Message Passing Example

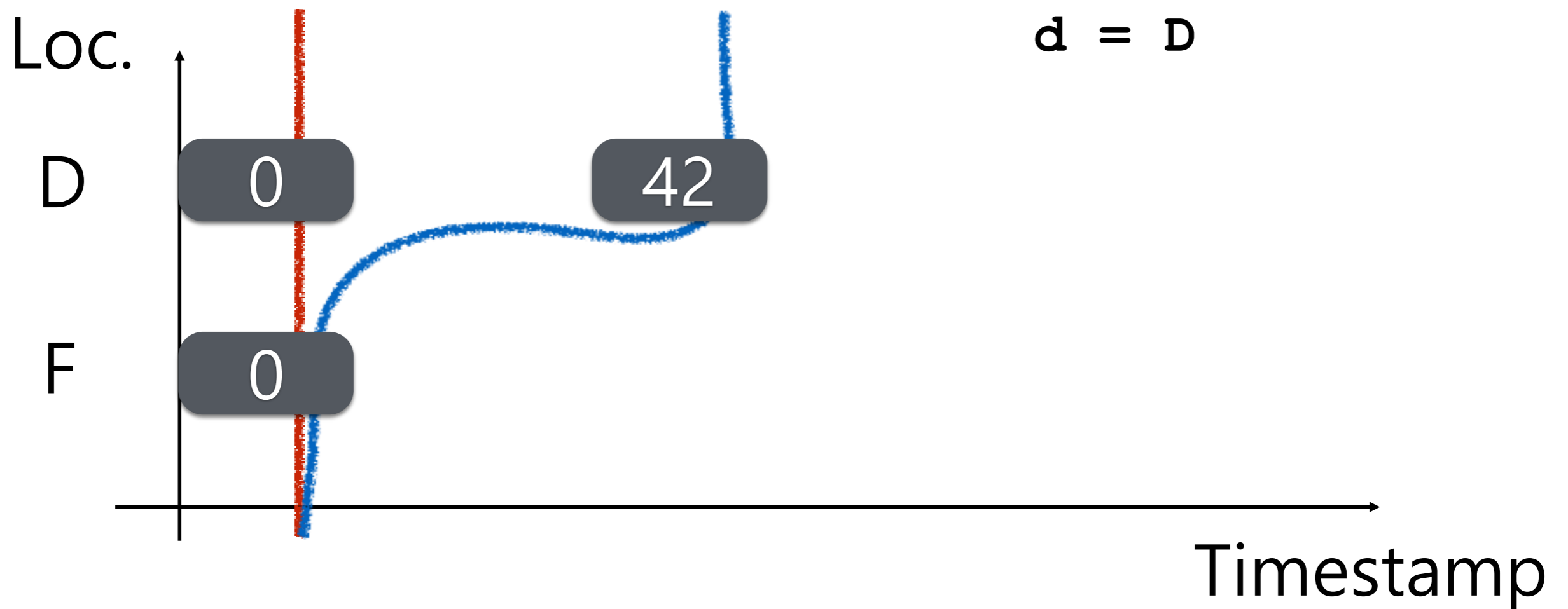
## Thread 1

→ D = 42  
F = 1

## Thread 2



```
while (1) {  
    f = F  
    if (f) break  
}  
d = D
```





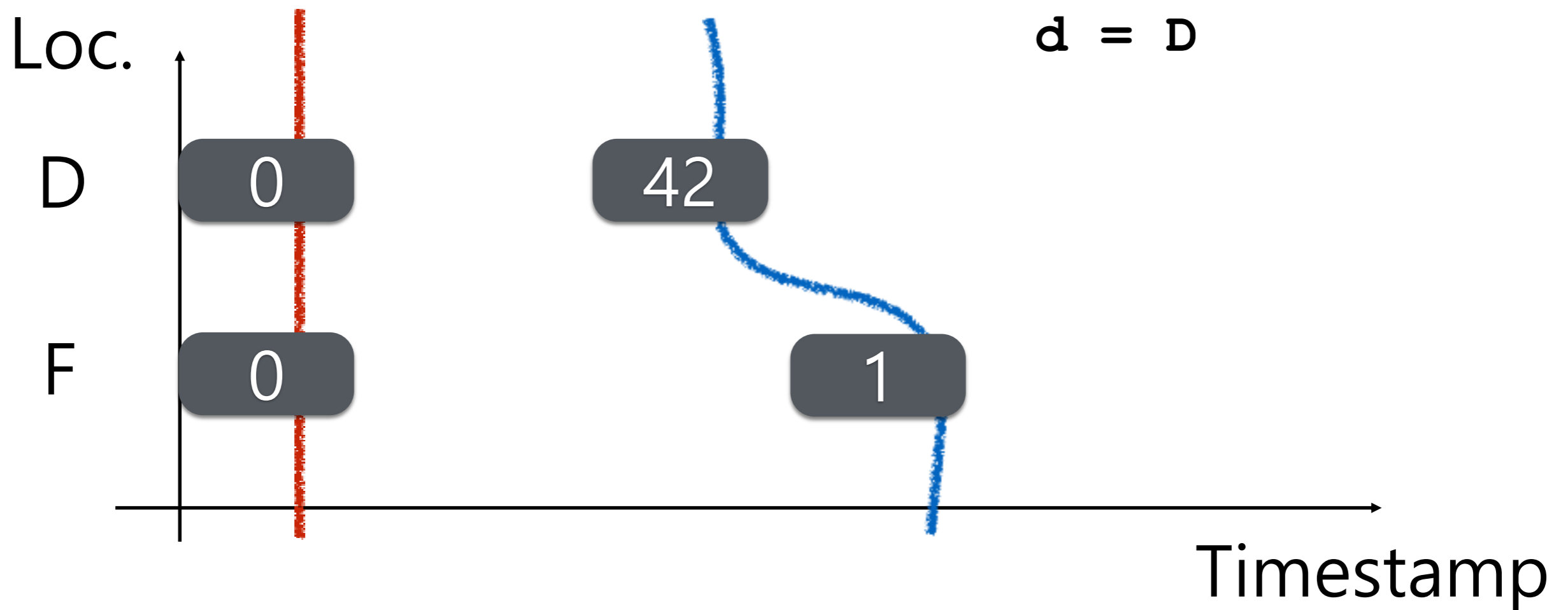
# The Message Passing Example

## Thread 1

$D = 42$   
 $F = 1$

## Thread 2

```
while (1) {  
    f = F  
    if (f) break  
}  
d = D
```



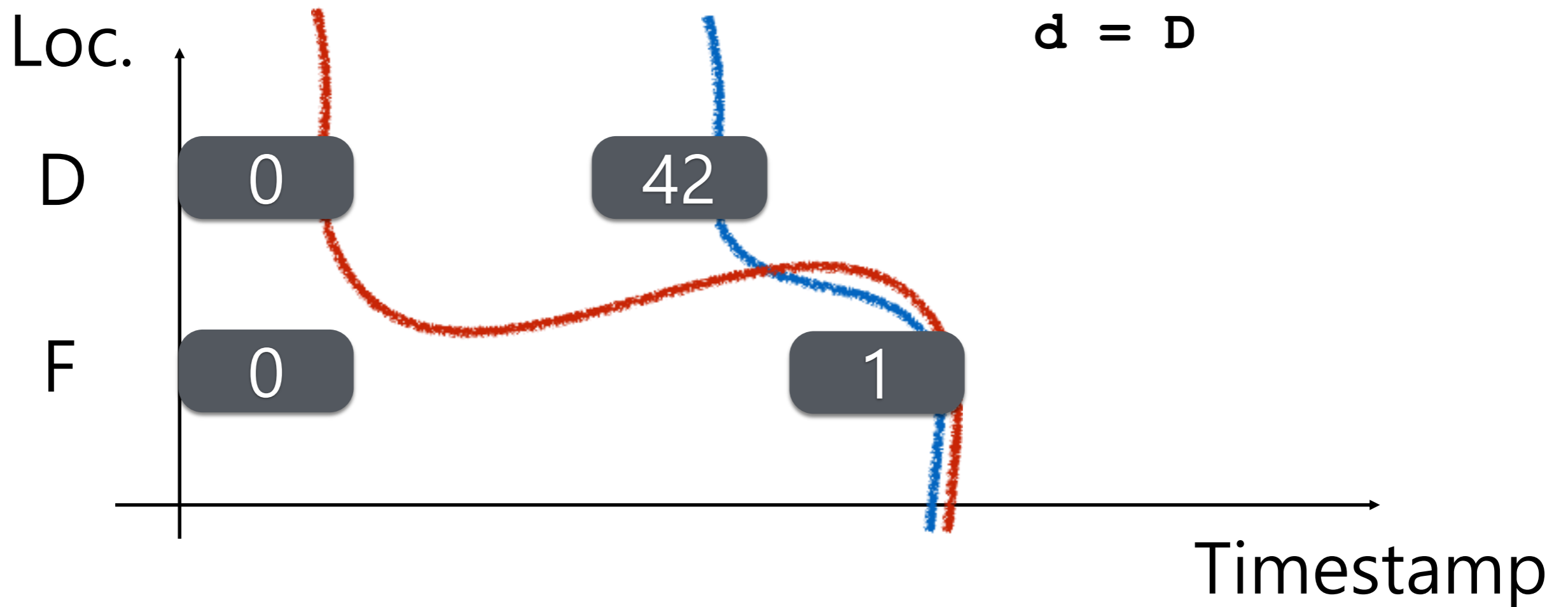
# The Message Passing Example

## Thread 1

$D = 42$   
 $F = 1$

## Thread 2

```
while (1) {  
    f = F  
    if (f) break  
}  
d = D
```



# The Message Passing Example

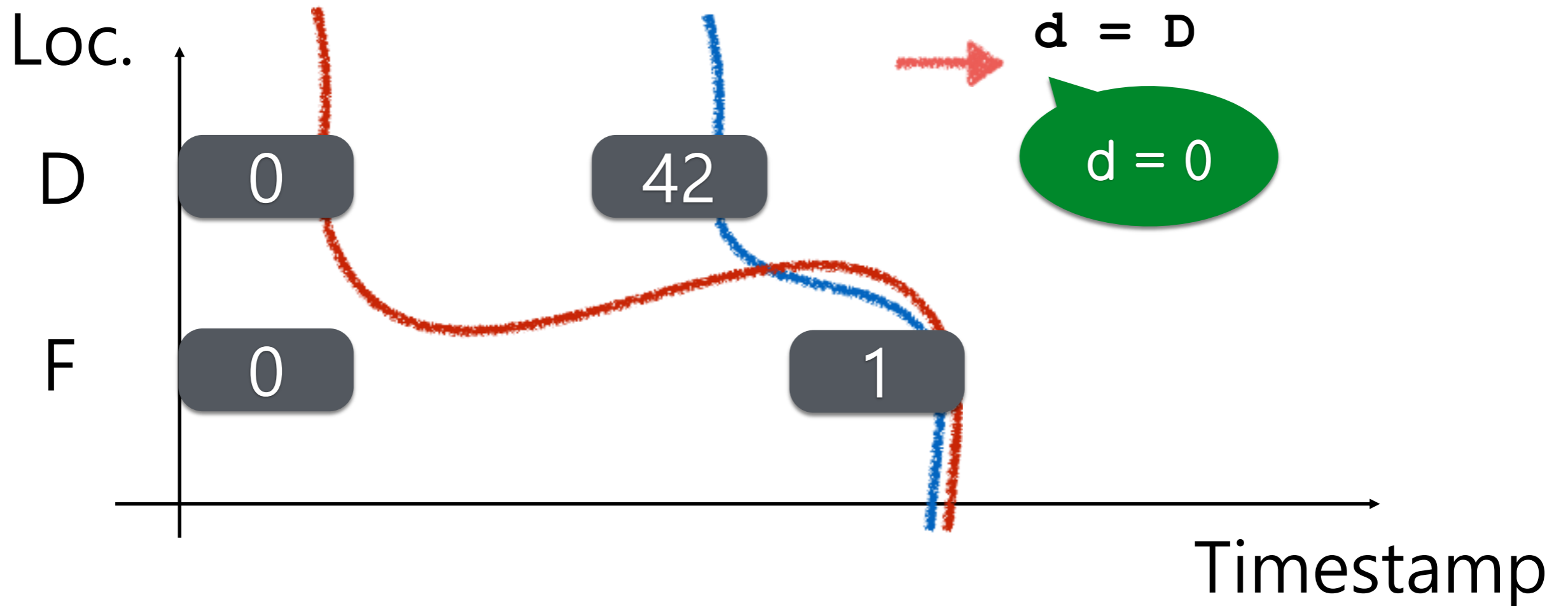
## Thread 1

$D = 42$   
 $\rightarrow F = 1$

## Thread 2

```
while (1) {  
    f = F  
    if (f) break  
}  
d = D
```

$d = 0$



# Release & Acquire

Thread 1



```
D = 42  
F = 1 [rel]
```

Thread 2



```
while (1) {  
    f = F [acq]  
    if (f) break  
}  
d = D
```



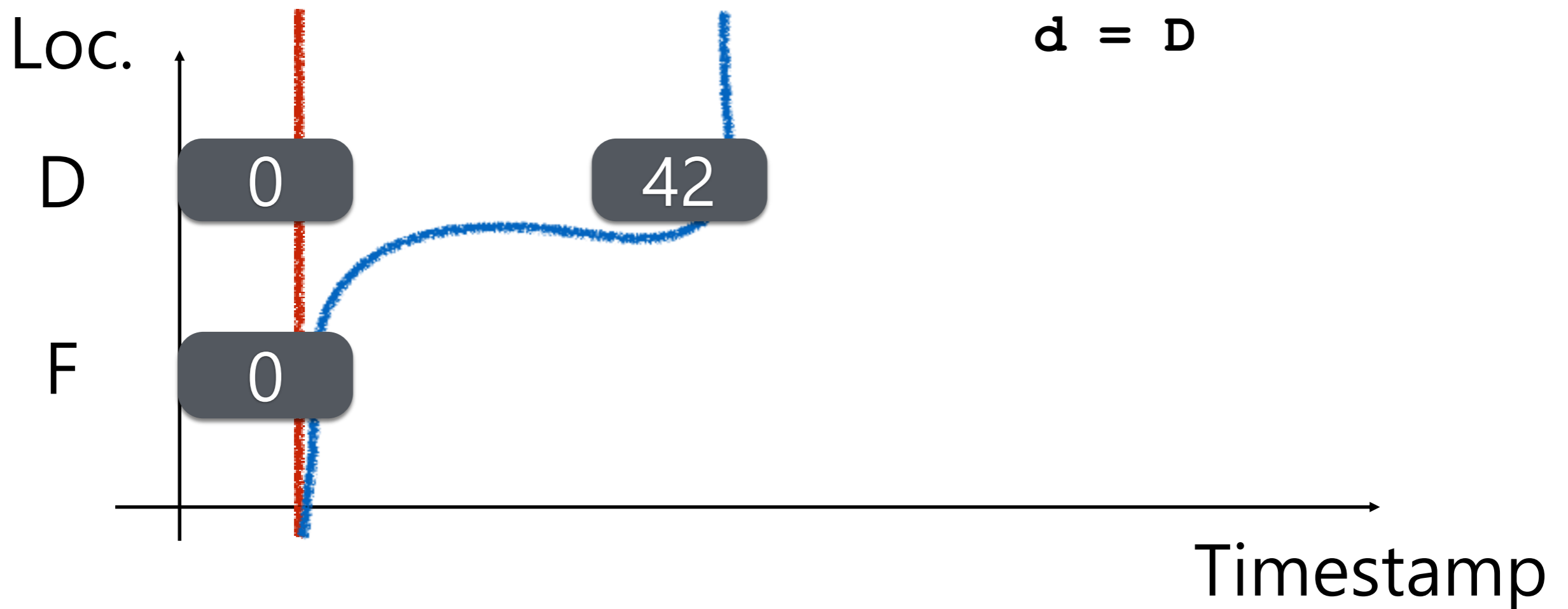
# Release & Acquire

## Thread 1

→ D = 42  
F = 1 [rel]

## Thread 2

→  
while (1) {  
 f = F [acq]  
 if (f) break  
}  
d = D



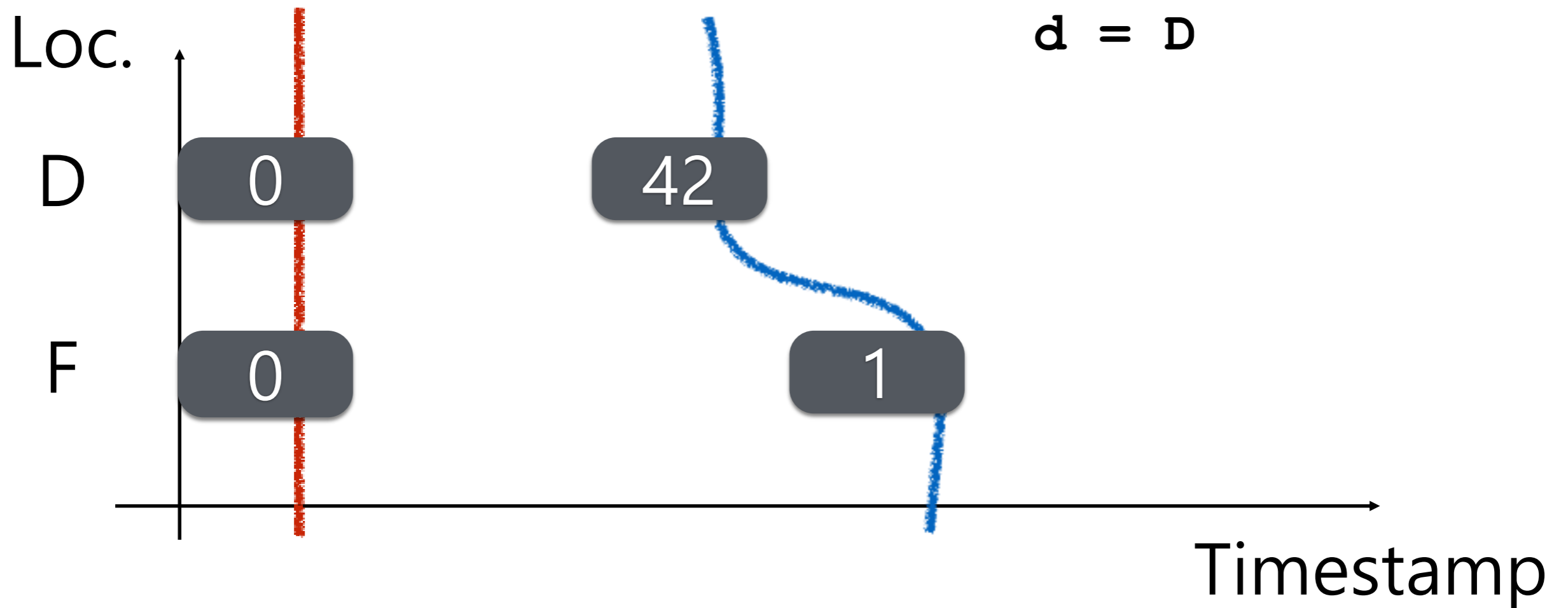
# Release & Acquire

Thread 1

$D = 42$   
 $F = 1$  [rel]

Thread 2

```
while (1) {  
    f = F [acq]  
    if (f) break  
}  
d = D
```



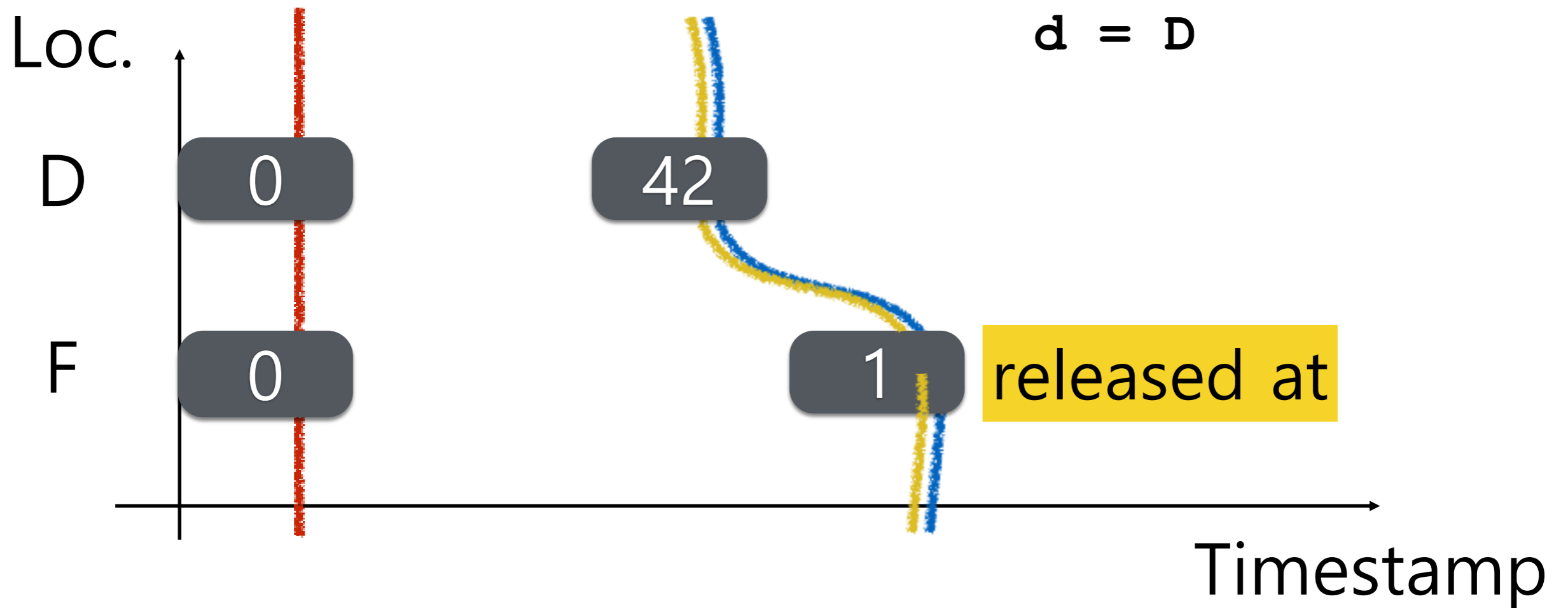
# Release & Acquire

Thread 1

$D = 42$   
 $F = 1$  [rel]

Thread 2

$\rightarrow$   
while (1) {  
    f = F [acq]  
    if (f) break  
}  
d = D



# Release & Acquire

## Thread 1

`D = 42`

`F = 1` [rel]

## Thread 2

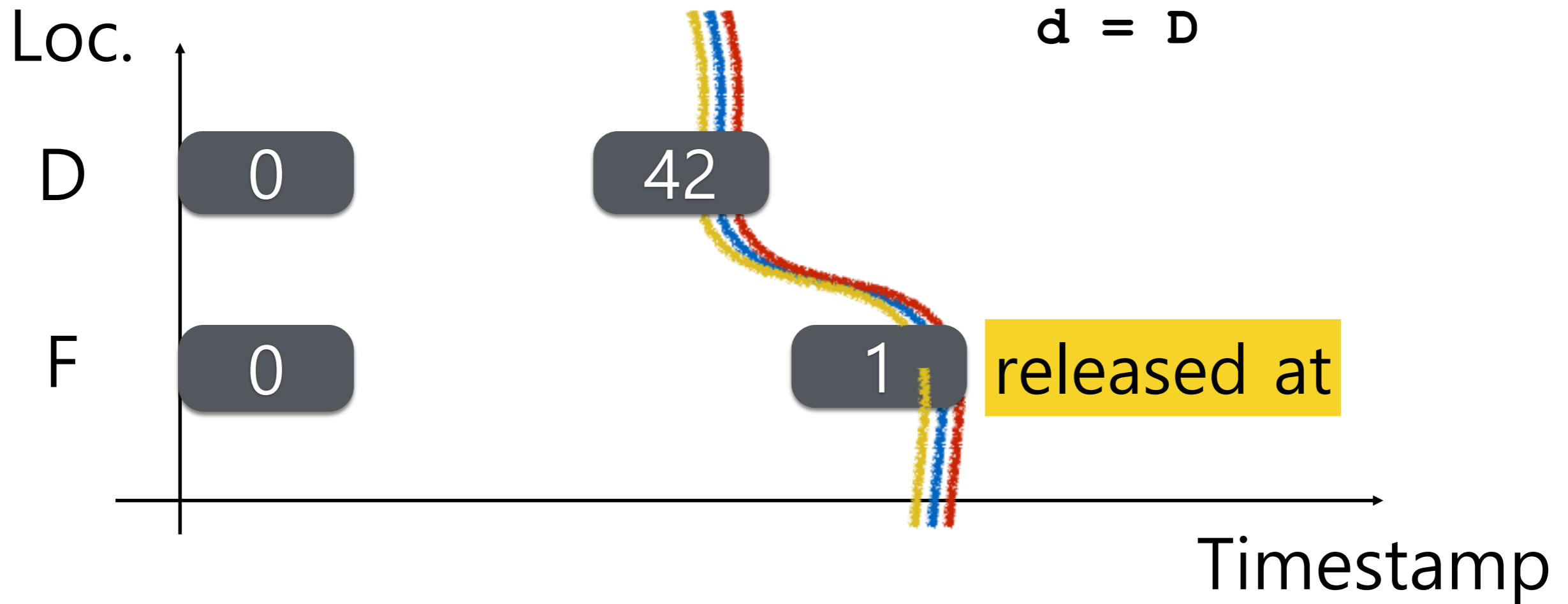
`while (1) {`

`f = F` [acq]

`if (f) break`

`}`

`d = D`





# Release & Acquire

## Thread 1

`D = 42`

`F = 1 [rel]`

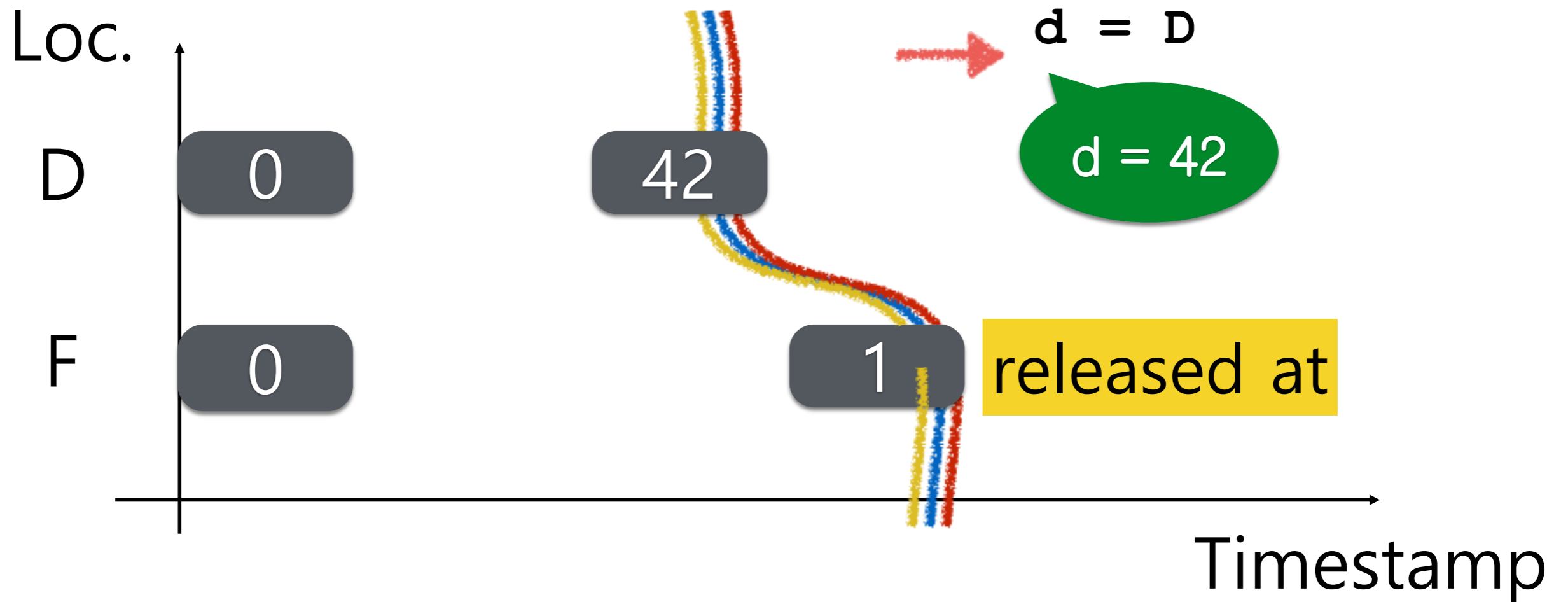
## Thread 2

```
while (1) {  
    f = F [acq]  
    if (f) break  
}
```

`d = D`

`d = 42`

released at



# Release & Acquire with Tweak

Initially:  $D = F = 0$

$D = 42$

$F = 1$  [rel]

$f = F$  [acq]

```
if (f) {  
    d = D // d = 42?  
}
```

$D = 10$

# Re-Certification is Necessary

- A thread must **re-certify during the execution** that it can write all its promises **in isolation**.

## Thread 1

W=1 [rel]

## Thread 2

[acq]

if (W) Y=1 [rel]

else Z=1

[acq]

if (X) Z=1

## Thread 3

[acq]

if (Y && Z) X=1 [rel]

(forbidden: X=1)

# Re-Certification is Necessary

- A thread must **re-certify during the execution** that it can write all its promises **in isolation**.

## Thread 1

W=1 [rel]

## Thread 2

[acq]

if (W) Y=1 [rel]

else Z=1 **Z=1 promised**

[acq]

if (X) Z=1

## Thread 3

[acq]

if (Y && Z) X=1 [rel]

(forbidden: X=1)

# Re-Certification is Necessary

- A thread must **re-certify during the execution** that it can write all its promises **in isolation**.

```
Thread 1  
W=1 [rel]
```

## Thread 2

```
[acq]  
if (W) Y=1 [rel]  
else Z=1  
[acq]  
if (X) Z=1
```

```
Thread 3  
[acq]  
if (Y && Z) X=1 [rel]
```

**Z=1 promised**

Certified:  
T2 in isolation

(forbidden: X=1)

# Re-Certification is Necessary

- A thread must **re-certify during the execution** that it can write all its promises **in isolation**.

## Thread 1

W=1 [rel]

## Thread 2

[acq]

if (W) Y=1 [rel]

else Z=1 **Z=1 promised**

[acq]

if (X) Z=1

## Thread 3

[acq]

if (Y && Z) X=1 [rel]

Certified:  
T2 in isolation

(forbidden: X=1)

# Re-Certification is Necessary

- A thread must **re-certify during the execution** that it can write all its promises **in isolation**.

**Thread 1**

W=1 [rel]

**Thread 2**

[acq]  
if (W) Y=1 [rel]

else Z=1

[acq]  
if (X) Z=1

**Thread 3**

[acq]  
if (Y && Z) X=1 [rel]

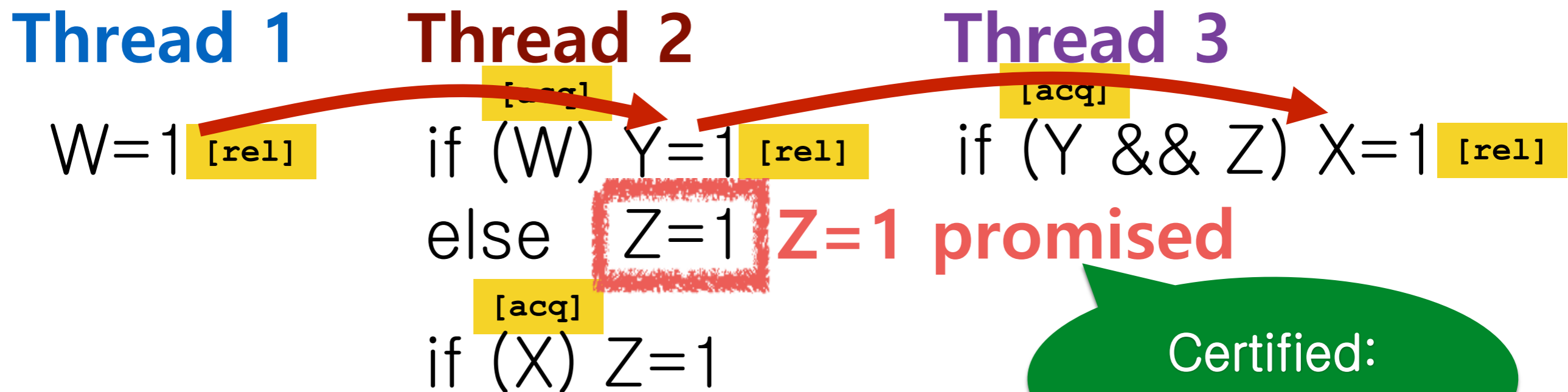
Z=1 promised

Certified:  
T2 in isolation

(forbidden: X=1)

# Re-Certification is Necessary

- A thread must **re-certify during the execution** that it can write all its promises **in isolation**.

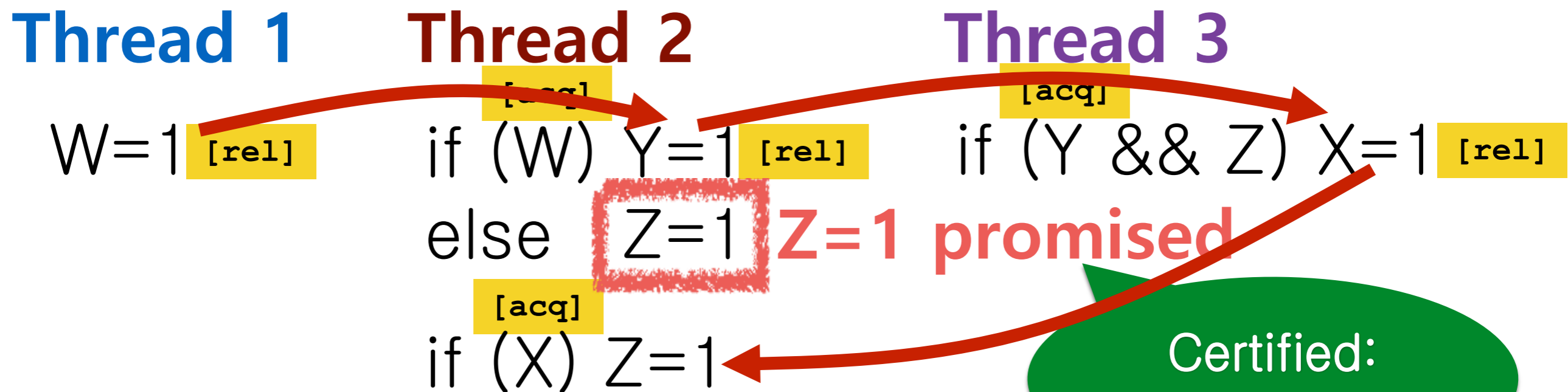


(forbidden: X=1)



# Re-Certification is Necessary

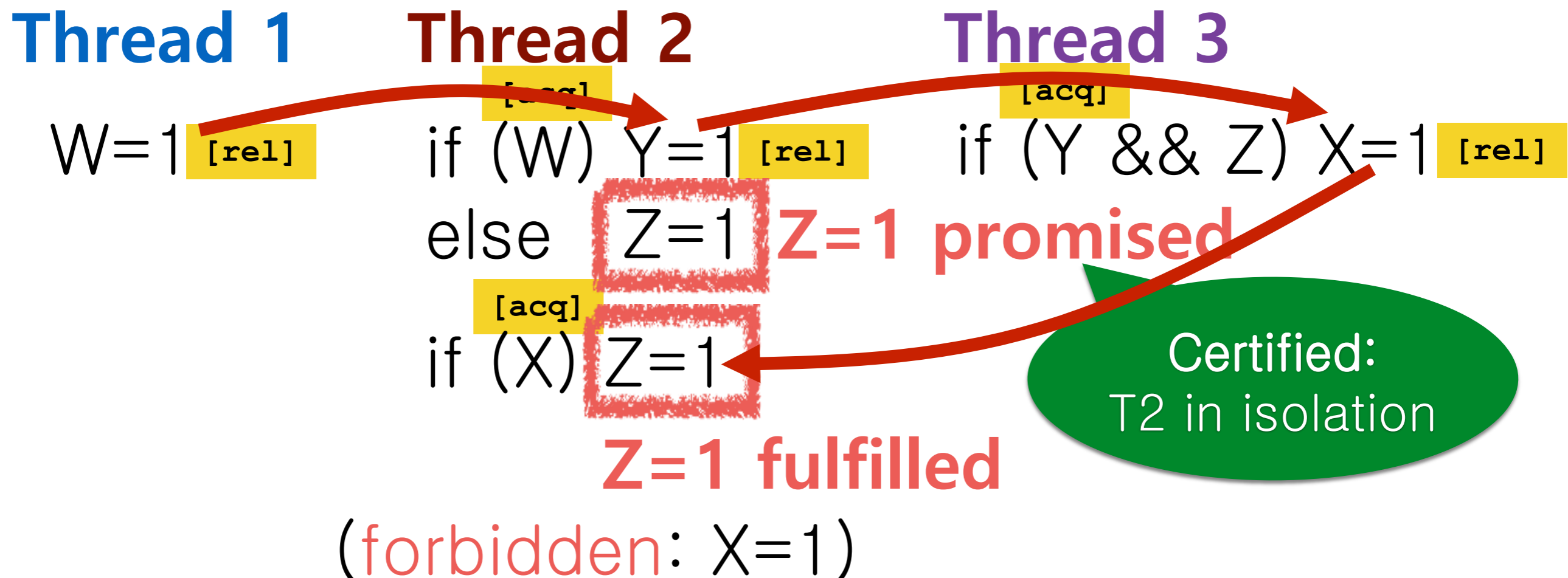
- A thread must **re-certify during the execution** that it can write all its promises **in isolation**.



(forbidden: X=1)

# Re-Certification is Necessary

- A thread must **re-certify during the execution** that it can write all its promises **in isolation**.



# Re-Certification is Necessary

- A thread must **re-certify during the execution** that it can write all its promises **in isolation**.

**Thread 1**

W=1 [rel]

**Thread 2**

[acq]  
if (W) Y=1 [rel]

else Z=1

[acq]  
if (X) Z=1

**Thread 3**

[acq]  
if (Y && Z) X=1 [rel]

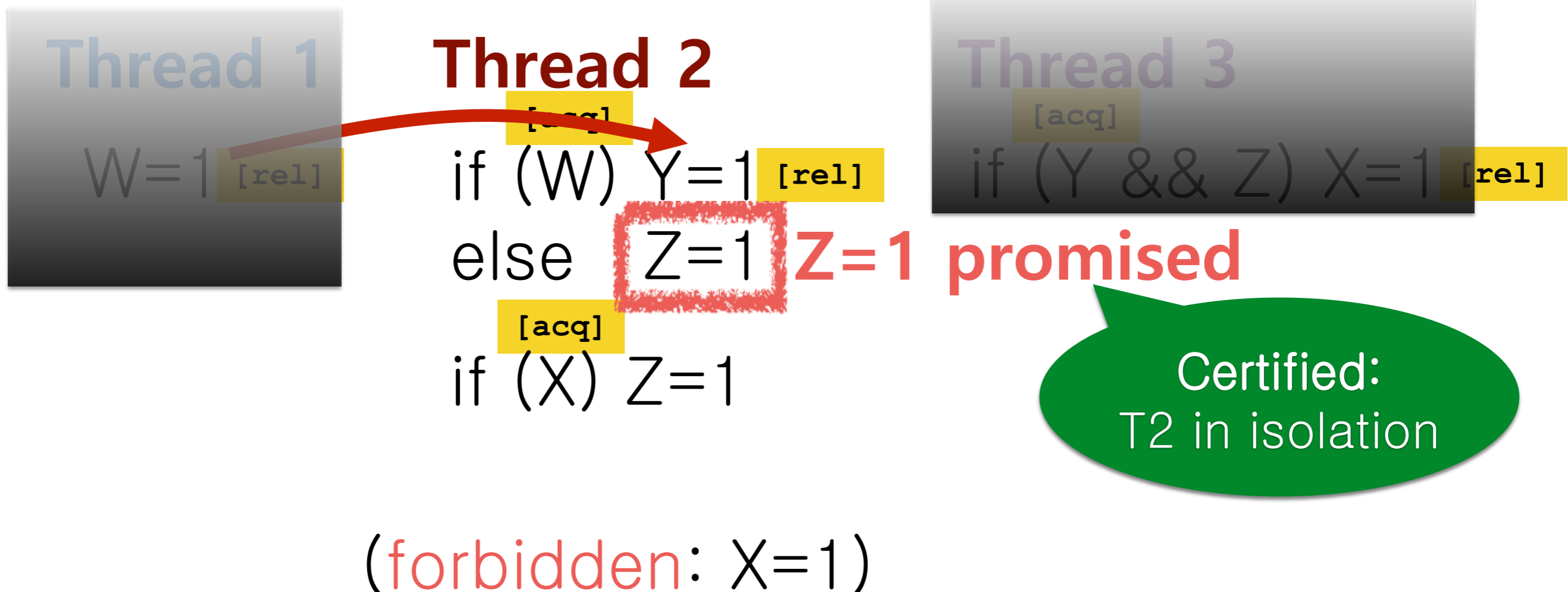
Z=1 promised

Certified:  
T2 in isolation

(forbidden: X=1)

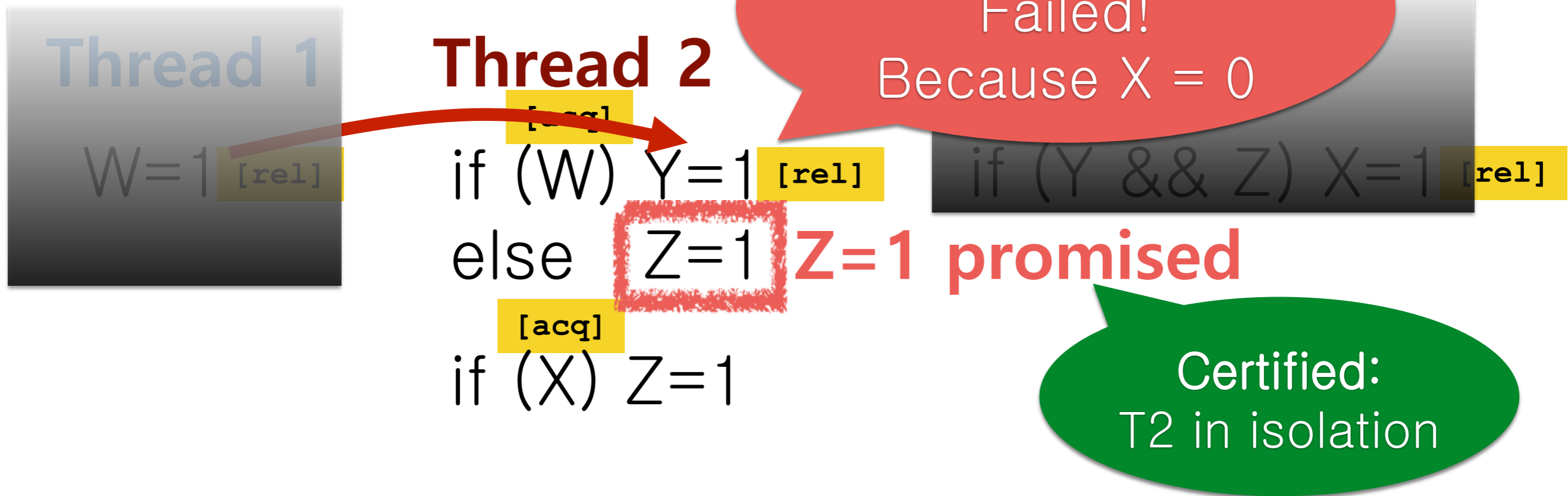
# Re-Certification is Necessary

- A thread must **re-certify during the execution** that it can write all its promises **in isolation**.



# Re-Certification is Necessary

- A thread must **re-certify during the execution** that it can write all its promises.



(forbidden: X=1)

# Example Counter

Core 1

Core 2

$r1 = \text{fetch-add } X$

$r2 = \text{fetch-add } X$

Loc.

X

0

(forbidden:  
 $r1=1, r2=1$ )

Timestamp

# Example Counter

Core 1

Core 2

$r1 = \text{fetch-add } X$

$r2 = \text{fetch-add } X$

(forbidden:  
 $r1=1, r2=1$ )

Loc.

X

0

1

Timestamp

# Example Counter

Core 1

Core 2

$r1 = \text{fetch-add } X$

$r2 = \text{fetch-add } X$

(forbidden:  
 $r1=1, r2=1$ )

Loc.

X

0

1

2

Timestamp



# Example

## Write-Read Coherence

**Thread 1**



$X = 1$

$a = X$

**Thread 2**



$X = 2$

$b = X$

(forbidden:  $a=2, b=1$ )

# Example

## Write-Read Coherence

**Thread 1**



$X = 1$   
 $a = X$

**Thread 2**




$X = 2$   
 $b = X$

(forbidden:  $a=2, b=1$ )


# Example

## Write-Read Coherence

**Thread 1**

  $X = 1$   
 $a = X$

**Thread 2**

  $X = 2$   
 $b = X$

(forbidden:  $a=2, b=1$ )

# Example

## Write-Read Coherence

**Thread 1**

$X = 1$

  $a = X$

**Thread 2**

  $X = 2$

$b = X$

(forbidden:  $a=2, b=1$ )

# Example

## Write-Read Coherence

**Thread 1**

$X = 1$

  $a = X$

**Thread 2**

$X = 2$

  $b = X$

(forbidden:  $a=2, b=1$ )

# Example

## Read-Write Coherence

**Thread 1**



$X = 1$

$X = 2$

**Thread 2**



$a = X$

$b = X$

(forbidden:  $a=2, b=1$ )

# Example

## Read-Write Coherence

**Thread 1**



$X = 1$   
 $X = 2$

**Thread 2**



$a = X$   
 $b = X$

(forbidden:  $a=2, b=1$ )

# Example

## Read-Write Coherence

**Thread 1**

$X = 1$

$X = 2$



**Thread 2**



$a = X$

$b = X$

(forbidden:  $a=2, b=1$ )



# Example

## Read-Write Coherence

**Thread 1**

$X = 1$

$X = 2$



**Thread 2**

$a = X$

$b = X$



(forbidden:  $a=2, b=1$ )

# Example

## Read-Write Coherence

### Thread 1

$X = 1$

$X = 2$



### Thread 2

$a = X$

$b = X$



(forbidden:  $a=2, b=1$ )




## Results (1/2)

# Compiler/HW Optimizations

- **Operational semantics** for C/C++ concurrency: plain/relaxed/release/acquire r/w/u/fence, SC fence
- **Compiler optimizations** 🍄  
(reordering, merge, dead load elim., ...)
- **Compilation** to x86 🍄 & Power 📄




# Results (2/2)

## Reasoning Principles

- **DRF-SC: Data Race Freedom  $\Rightarrow$  SC** 
  - DRF-PromiseFree:  
DRF  $\Rightarrow$  semantics w/o promises 
- **Invariant-based logic:**   
soundness of global invariant (e.g.  $a=b=X=Y=0$ )
- <http://sf.snu.ac.kr/promise-concurrency>



# More comprehensive semantics for C/C++ concurrency

- **DRF-SC: Data Race Freedom  $\Rightarrow$  SC** 
  - DRF-PromiseFree:  
DRF  $\Rightarrow$  semantics w/o promises 
- **Invariant-based logic:**   
soundness of global invariant (e.g.  $a=b=X=Y=0$ )
- <http://sf.snu.ac.kr/promise-concurrency>



# Future Work

- Supporting **SC reads & writes**  
(We found a **flaw in C/C++11 on SC**)
- Supporting **consume** reads
- Developing a rich **program logic** &  
**Verifying** fine-grained concurrent programs