A Promising Semantics for Relaxed-Memory Concurrency by Kang et al

@pwlpdx

Nick Fitzgerald
Papers We Love PDX
October 26th, 2016

@fitzgen

- new memory model
- for relaxed memory
- and concurrency
- not my area of expertise
    - any incorrectness is my own
- foundational -> important
- approachable

Memory model?

Formal semantics for **loads** and **stores**

- formal semantics for reading from and writing to memory
  - what order?
  - what values?
- "relaxed" = normal loads and stores, no atomics/synchronization
- intuition for uniprocessor memory model:
  - map from addresses to values
  - read returns current value associated with address
  - write immediately updates address's associated value

Difficulties

- Multiprocessors

- multiprocessors and concurrency
  - which operation happened first?
  - can we even define an ordering on memory operations?
- SLIDE — Memory is actually hierarchical w/ multiple layers of caches
  - some shared between all cores
  - other layers distinct between cores
- SLIDE — We have those cache layers because memory is slow
  - particularly writes b/c they invalidate other cores' caches
  - write buffers batch writes so that they don't need to happen as often
- SLIDE — Optimizing compilers want to minimize + reorder loads and stores
  - move them out of loops
  - easy to reason about only one thread, but not so w/ concurrency
- Both Intel and AMD repeatedly published incorrect descriptions of their own semantics
  - x86 TSO paper observed behavior Intel/AMD said was impossible
  - and then went further and provided semantics that *did* accurately describe behavior
  - but that Power and ARM have even weaker memory behavior, using TSO on them would require tons of expensive fences
- 2 notable programming language level memory models:
  - 1. java: first language to provide a formal mem model
    - both too weak to reason about and so strong it prevented many basic compiler optimizations
    - and this is Guy Steele — what chance do us mere mortals have?

Difficulties

- Multiprocessors
- Caches

- multiprocessors and concurrency
  - which operation happened first?
  - can we even define an ordering on memory operations?
- SLIDE — Memory is actually hierarchical w/ multiple layers of caches
  - some shared between all cores
  - other layers distinct between cores
- SLIDE — We have those cache layers because memory is slow
  - particularly writes b/c they invalidate other cores' caches
  - write buffers batch writes so that they don't need to happen as often
- SLIDE — Optimizing compilers want to minimize + reorder loads and stores
  - move them out of loops
  - easy to reason about only one thread, but not so w/ concurrency
- Both Intel and AMD repeatedly published incorrect descriptions of their own semantics
  - x86 TSO paper observed behavior Intel/AMD said was impossible
  - and then went further and provided semantics that *did* accurately describe behavior
  - but that Power and ARM have even weaker memory behavior, using TSO on them would require tons of expensive fences
- 2 notable programming language level memory models:
  - 1. java: first language to provide a formal mem model
    - both too weak to reason about and so strong it prevented many basic compiler optimizations
    - and this is Guy Steele — what chance do us mere mortals have?

Difficulties

- Multiprocessors
- Caches
- Buffers

- multiprocessors and concurrency
  - which operation happened first?
  - can we even define an ordering on memory operations?
- SLIDE — Memory is actually hierarchical w/ multiple layers of caches
  - some shared between all cores
  - other layers distinct between cores
- SLIDE — We have those cache layers because memory is slow
  - particularly writes b/c they invalidate other cores' caches
  - write buffers batch writes so that they don't need to happen as often
- SLIDE — Optimizing compilers want to minimize + reorder loads and stores
  - move them out of loops
  - easy to reason about only one thread, but not so w/ concurrency
- Both Intel and AMD repeatedly published incorrect descriptions of their own semantics
  - x86 TSO paper observed behavior Intel/AMD said was impossible
  - and then went further and provided semantics that *did* accurately describe behavior
  - but that Power and ARM have even weaker memory behavior, using TSO on them would require tons of expensive fences
- 2 notable programming language level memory models:
  - 1. java: first language to provide a formal mem model
    - both too weak to reason about and so strong it prevented many basic compiler optimizations
    - and this is Guy Steele — what chance do us mere mortals have?

Difficulties

- Multiprocessors
- Caches
- Buffers
- Compiler optimizations

- multiprocessors and concurrency
    - which operation happened first?
    - can we even define an ordering on memory operations?
- SLIDE — Memory is actually hierarchical w/ multiple layers of caches
    - some shared between all cores
    - other layers distinct between cores
- SLIDE — We have those cache layers because memory is slow
    - particularly writes b/c they invalidate other cores' caches
    - write buffers batch writes so that they don't need to happen as often
- SLIDE — Optimizing compilers want to minimize + reorder loads and stores
    - move them out of loops
    - easy to reason about only one thread, but not so w/ concurrency
- Both Intel and AMD repeatedly published incorrect descriptions of their own semantics
    - x86 TSO paper observed behavior Intel/AMD said was impossible
    - and then went further and provided semantics that *did* accurately describe behavior
    - but that Power and ARM have even weaker memory behavior, using TSO on them would require tons of expensive fences
- 2 notable programming language level memory models:
    - 1. java: first language to provide a formal mem model
        - both too weak to reason about and so strong it prevented many basic compiler optimizations
        - and this is Guy Steele — what chance do us mere mortals have?

“Out of Thin Air” Reads

- The C++ memory model is fundamentally broken in usual way
  - undefined behavior if not DRF
  - out of thin air reads
    - read a value that never had a corresponding previous write
    - circular argument that if we read this value now
    - it could then be written in the future
    - and then travel back to the past and become its own grandfather

"Without a semantics, programmers currently have to program against their **folklore** understanding of what the Java and C/C++ implementations provide, and research on verification, compilation, or testing for such languages is on **shaky foundations**."

–Batty et al in *The Problem of Programming Language Concurrency Semantics*

- we have been writing concurrent programs
  - in a glass house of cards
  - built on foundation of sand
  - where "writing concurrent programs" means "throwing rocks"
  - and all the while we're waiting for the Next Big Earthquake
- can't avoid this by using higher level languages/paradigms
  - guess what they're implemented in?

Goals

- Describe **actual hardware** behavior

- often conflicting
- has to describe behavior of hardware that already exists
  - ships have sailed
  - if you don't, then it won't get used, period
  - at the same time, can't over specify and constrain future hardware design
- SLIDE — if too strong, compilers can't reorder/elide
- SLIDE — if too weak, can't reason about behavior/correctness programs
  - easiest thing to reason about is SC, not practical
- SLIDE — absolutely need to avoid UB
  - nondeterministic behavior ok

Goals

- Describe **actual hardware** behavior
- Weak enough to enable **compiler optimizations**

- often conflicting
- has to describe behavior of hardware that already exists
  - ships have sailed
  - if you don't, then it won't get used, period
  - at the same time, can't over specify and constrain future hardware design
- SLIDE — if too strong, compilers can't reorder/elide
- SLIDE — if too weak, can't reason about behavior/correctness programs
  - easiest thing to reason about is SC, not practical
- SLIDE — absolutely need to avoid UB
  - nondeterministic behavior ok

Goals

- Describe **actual hardware** behavior
- Weak enough to enable **compiler optimizations**
- Strong enough to **reason** on top of

- often conflicting
- has to describe behavior of hardware that already exists
    - ships have sailed
    - if you don’t, then it won’t get used, period
    - at the same time, can’t over specify and constrain future hardware design
- SLIDE — if too strong, compilers can’t reorder/elide
- SLIDE — if too weak, can’t reason about behavior/correctness programs
    - easiest thing to reason about is SC, not practical
- SLIDE — absolutely need to avoid UB
    - nondeterministic behavior ok

Goals

- Describe **actual hardware** behavior
- Weak enough to enable **compiler optimizations**
- Strong enough to **reason** on top of
- No **undefined behavior!**

- often conflicting
- has to describe behavior of hardware that already exists
  - ships have sailed
  - if you don't, then it won't get used, period
  - at the same time, can't over specify and constrain future hardware design
- SLIDE — if too strong, compilers can't reorder/elide
- SLIDE — if too weak, can't reason about behavior/correctness programs
  - easiest thing to reason about is SC, not practical
- SLIDE — absolutely need to avoid UB
  - nondeterministic behavior ok

The *Promising* Semantics

- (Mostly) backwards compatible with the C++11 memory model

- backwards compat = maybe we can get C++ to adopt this model?
- SLIDE — no OOTA = sanity
- SLIDE — allowing some eliding + reordering permits normal nondeterminism of concurrency + compiler optimization
- SLIDE — if a program is single threaded, or uses atomics/locks properly, then it is completely unaffected
- SLIDE — no undefined behavior
    - yes nondeterminism, but all permissible executions are well defined

The *Promising* Semantics

- (Mostly) backwards compatible with the C++11 memory model
- Avoids out-of-thin-air

- backwards compat = maybe we can get C++ to adopt this model?
- SLIDE — no OOTA = sanity
- SLIDE — allowing some eliding + reordering permits normal nondeterminism of concurrency + compiler optimization
- SLIDE — if a program is single threaded, or uses atomics/locks properly, then it is completely unaffected
- SLIDE — no undefined behavior
    - yes nondeterminism, but all permissible executions are well defined

The *Promising* Semantics

- (Mostly) backwards compatible with the C++11 memory model
- Avoids out-of-thin-air
- Permits sane eliding and reordering

- backwards compat = maybe we can get C++ to adopt this model?
- SLIDE — no OOTA = sanity
- SLIDE — allowing some eliding + reordering permits normal nondeterminism of concurrency + compiler optimization
- SLIDE — if a program is single threaded, or uses atomics/locks properly, then it is completely unaffected
- SLIDE — no undefined behavior
    - yes nondeterminism, but all permissible executions are well defined

The *Promising* Semantics

- (Mostly) backwards compatible with the C++11 memory model
- Avoids out-of-thin-air
- Permits sane eliding and reordering
- Not infectious for data-race free programs

- backwards compat = maybe we can get C++ to adopt this model?
- SLIDE — no OOTA = sanity
- SLIDE — allowing some eliding + reordering permits normal nondeterminism of concurrency + compiler optimization
- SLIDE — if a program is single threaded, or uses atomics/locks properly, then it is completely unaffected
- SLIDE — no undefined behavior
    - yes nondeterminism, but all permissible executions are well defined

The *Promising* Semantics

- (Mostly) backwards compatible with the C++11 memory model
- Avoids out-of-thin-air
- Permits sane eliding and reordering
- Not infectious for data-race free programs
- No undefined behavior

- backwards compat = maybe we can get C++ to adopt this model?
- SLIDE — no OOTA = sanity
- SLIDE — allowing some eliding + reordering permits normal nondeterminism of concurrency + compiler optimization
- SLIDE — if a program is single threaded, or uses atomics/locks properly, then it is completely unaffected
- SLIDE — no undefined behavior
    - yes nondeterminism, but all permissible executions are well defined

The *Promising* Semantics

Comprehensible!!

- what I liked most about this paper, and what made me love it, is how easy it is to understand
- especially in comparison to other memory semantics papers
    - operational semantics vs happens-before partial ordering

Example (SB)

```
x := 1           ||  y := 1
a := y // 0      ||  b := x // 0
```

- first, let’s go through examples from the paper
    - whether we want to permit it or not
    - semantics will have to formally describe why it is permissible or not
- 2 threads, separated by ||
- SLIDE — all memory and all registers are initially 0
- SLIDE — a, b, c are registers
- SLIDE — x, y, z are all distinct memory locations
    - x := 1 is writing to memory
    - a := y is reading from memory
- SLIDE — // 0 means we observed 0 for a given read
    - so how do we observe that both x and y are 0 here??

Example (SB)

```
x := 1          ||   y := 1
a := y // 0     ||   b := x // 0
```

- Everything is initially `0`

- first, let's go through examples from the paper
  - whether we want to permit it or not
  - semantics will have to formally describe why it is permissible or not
- 2 threads, separated by ||
- SLIDE — all memory and all registers are initially 0
- SLIDE — a, b, c are registers
- SLIDE — x, y, z are all distinct memory locations
  - x := 1 is writing to memory
  - a := y is reading from memory
- SLIDE — // 0 means we observed 0 for a given read
  - so how do we observe that both x and y are 0 here??

Example (SB)

```
x := 1          ||  y := 1
a := y // 0     ||  b := x // 0
```

- Everything is initially `0`
- `a, b, c` are registers

- first, let's go through examples from the paper
  - whether we want to permit it or not
  - semantics will have to formally describe why it is permissible or not
- 2 threads, separated by ||
- SLIDE — all memory and all registers are initially 0
- SLIDE — a, b, c are registers
- SLIDE — x, y, z are all distinct memory locations
  - x := 1 is writing to memory
  - a := y is reading from memory
- SLIDE — // 0 means we observed 0 for a given read
  - so how do we observe that both x and y are 0 here??

Example (SB)

```
x := 1          ||   y := 1
a := y // 0     ||   b := x // 0
```

- Everything is initially `0`
- `a, b, c` are registers
- `x, y, z` are distinct memory locations

- first, let's go through examples from the paper
  - whether we want to permit it or not
  - semantics will have to formally describe why it is permissible or not
- 2 threads, separated by ||
- SLIDE — all memory and all registers are initially 0
- SLIDE — a, b, c are registers
- SLIDE — x, y, z are all distinct memory locations
  - x := 1 is writing to memory
  - a := y is reading from memory
- SLIDE — // 0 means we observed 0 for a given read
  - so how do we observe that both x and y are 0 here??

Example (SB)

```
x := 1          ||  y := 1
a := y // 0     ||  b := x // 0
```

- Everything is initially `0`
- `a, b, c` are registers
- `x, y, z` are distinct memory locations
- `// 0` means that we observed the value `0`

- first, let's go through examples from the paper
  - whether we want to permit it or not
  - semantics will have to formally describe why it is permissible or not
- 2 threads, separated by ||
- SLIDE — all memory and all registers are initially 0
- SLIDE — a, b, c are registers
- SLIDE — x, y, z are all distinct memory locations
  - x := 1 is writing to memory
  - a := y is reading from memory
- SLIDE — // 0 means we observed 0 for a given read
  - so how do we observe that both x and y are 0 here??

Example (SB)

```
a := y // 0   ||   b := x // 0
x := 1        ||   y := 1
```

Store buffering!

- no syntactic dependency between `x := 1` and `a := y`
- therefore, safe to buffer the write to happen later
- reorders each threads read and write operations so they execute like the above
    - both threads do their reads
    - and then their writes
- want to support
    - x86 does

Example (LB)

`a := x // 1`	‖	`x := y`
`y := 1`	‖	

- should we permit the first thread to observe 1 for its read of x?
  - yes

Example (LB)

```
y := 1            ||   x := y
a := x // 1
```

Power

Load buffering!

ARM

* no dependency between first thread's write to y and read of x
* load buffering allows delaying the read so that the write to y happens first
* to observe a read where x is 1:
    * first thread writes y=1
    * second thread reads y=1 and writes x=y=1
    * first thread reads x=1
* crazy as this behavior may seem, both Power and ARM do this!

Example (LBd)

```
a := x // 1?  ||  x := y
y := a        ||
```

* instead of writing y = 1, writing y = a
* Permit first thread to observe x = 1?
    * nope!

Example (LBd)

a := x // ~~1~~ ‖ x := y
y := a

Nope!

* first, syntactic dependency between first thread's read and write
  * can't reorder them
* second, where did 1 come from?
  * out of thin air!
* this is actually permitted by the C++11 memory model
  * but we should hold ourselves to higher standards!

Example (LBfd)

```
a := x // 1?      ‖  x := y
y := a + 1 - a    ‖
```

* Same thing but replace `y := a` with `y := a + 1 - a`
* Now should we permit observing 1 when reading x?
  * yes!

Example (LBfd)

`a := x // 1?` `y := 1`	‖	`x := y`

Compiler optimizations!

* We should permit an optimizing compiler to reduce `a + 1 - a` to `1`

Example (LBfd)

```
y := 1                ‖   x := y
a := x // 1?
```

Load buffering!

Compiler optimizations!

* and once we've done that, there is no more syntactic dependency between the read and write
* we can do the same load buffering induced reordering as example (LB)

Example (LBfd)

```
y := 1                ║   x := y
a := x // 1!          ║
```

Load buffering!

Compiler optimizations!

Yes!

- To observe x=1:
  - first thread writes y=1
  - second thread reads y=1, then writes x=y=1
  - first thread reads x=1

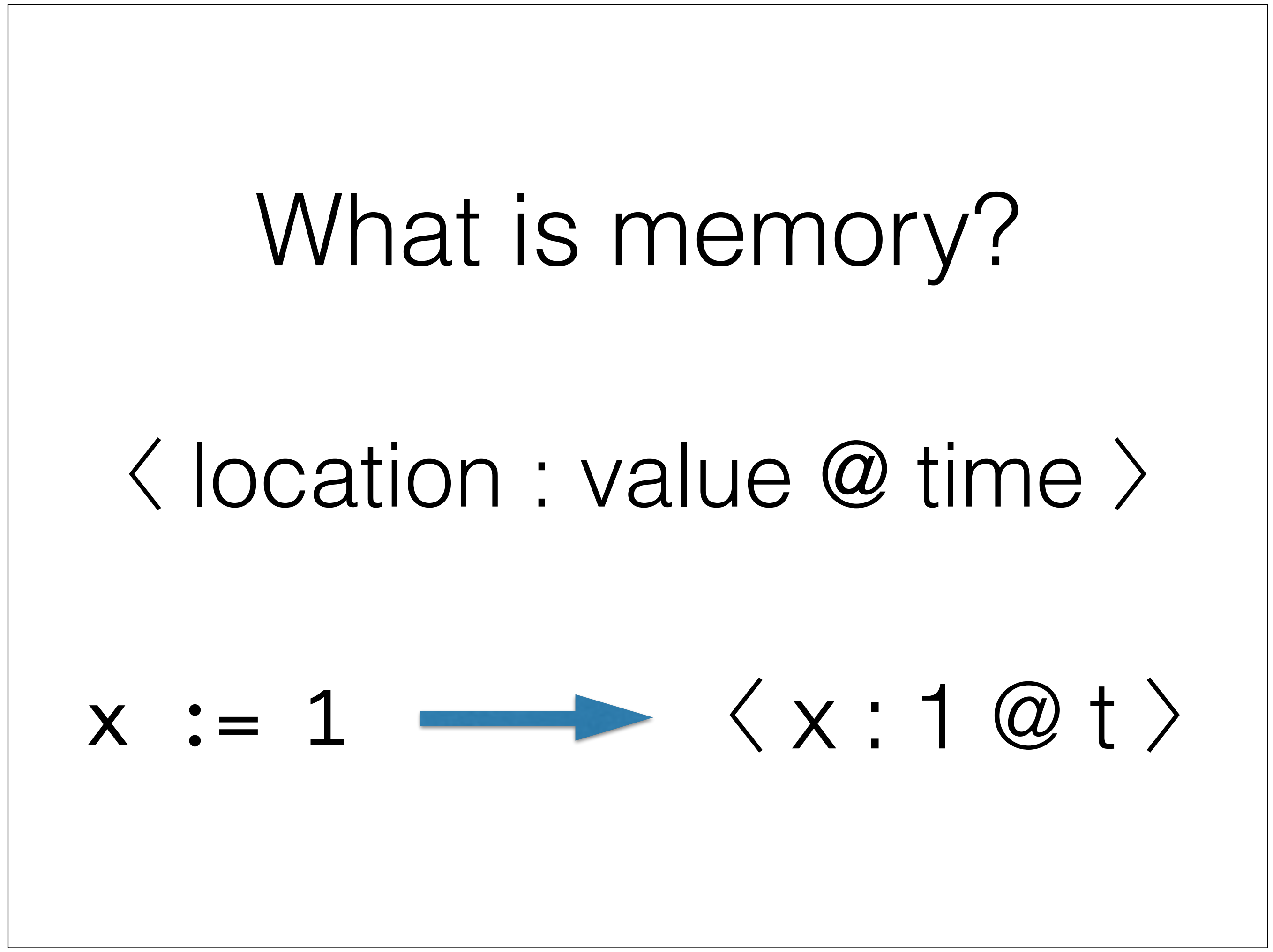
What is memory?

- traditionally think of map from address to value
  - not conducive to formalization
- SLIDE — set of all writes ever occurred in program
- location: x or y or z
- value: 1 or whatever
- time?
  - rational number
  - any number representable by p / q where p,q are integers
  - infinite number of rationals
  - and infinite number of rationals between two distinct rationals == “dense”
  - more on this later…
- SLIDE
  - \`x := 1\` translates to 〈 x : 1 @ t 〉
- initially contains zero messages for all memory at timestamp 0

What is memory?

⟨ location : value @ time ⟩

- traditionally think of map from address to value
  - not conducive to formalization
- SLIDE — set of all writes ever occurred in program
- location: x or y or z
- value: 1 or whatever
- time?
  - rational number
  - any number representable by p / q where p,q are integers
  - infinite number of rationals
  - and infinite number of rationals between two distinct rationals == “dense”
  - more on this later…
- SLIDE
  - `x := 1` translates to ⟨ x : 1 @ t ⟩
- initially contains zero messages for all memory at timestamp 0



- traditionally think of map from address to value
  - not conducive to formalization
- SLIDE — set of all writes ever occurred in program
- location: x or y or z
- value: 1 or whatever
- time?
  - rational number
  - any number representable by p / q where p,q are integers
  - infinite number of rationals
  - and infinite number of rationals between two distinct rationals == “dense”
  - more on this later…
- SLIDE
  - `x := 1` translates to 〈 x : 1 @ t 〉
- initially contains zero messages for all memory at timestamp 0

- Each thread has a local address ⇒ timestamp map
- It is updated by reads and writes

- Thread local maps from memory locations => largest timestamp observed for location
- map updated by thread’s reads/writes
    - reads must be satisfied by a write message w/ timestamp >= thread’s view of address
    - writes allocate new timestamp > thread’s view of address, add write message to global memory set

```
write(T, x, v):
    let t = new unique timestamp > T.view[x]
    T.view[x] = t
    insert 〈 x : v @ t 〉 into the memory set
```

- not going to go into the operational semantics, use pseudocode instead
- T = thread
- x = memory location
- v = value being written
- T.view is thread-local map
    - T.view[x] is thread’s most recent view of memory location x

```
write(T, x, v):
    let t = new unique timestamp > T.view[x]
    T.view[x] = t
    insert 〈 x : v @ t 〉 into the memory set
```

```
write(T, x, v):
    let t = new unique timestamp > T.view[x]
    T.view[x] = t
    insert 〈 x : v @ t 〉 into the memory set
```

```
write(T, x, v):
    let t = new unique timestamp > T.view[x]
    T.view[x] = t
    insert ⟨ x : v @ t ⟩ into the memory set
```

- use of timestamps provides “coherence”
  - total order on writes to particular location

```
read(T, x):
    let t = T.view[x]
    let possibles = empty set
    for each 〈 x' : v @ t' 〉 in memory:
        if x' == x and t' >= t:
            insert 〈 x' : v @ t' 〉 into possibles
    let 〈 _ : v @ t' 〉 =
        nondeterministically choose one from possibles
    T.view[x] = t'
    return v
```

- again:
  - T = thread
  - x = memory location
  - T.view is thread-local map

```
read(T, x):
    let t = T.view[x]
    let possibles = empty set
    for each 〈 x' : v @ t' 〉 in memory:
        if x' == x and t' >= t:
            insert 〈 x' : v @ t' 〉 into possibles
    let 〈 _ : v @ t' 〉 =
        nondeterministically choose one from possibles
    T.view[x] = t'
    return v
```

- get timestamp of current thread’s view of x

```
read(T, x):
    let t = T.view[x]
    let possibles = empty set
    for each 〈 x' : v @ t' 〉 in memory:
        if x' == x and t' >= t:
            insert 〈 x' : v @ t' 〉 into possibles
    let 〈 _ : v @ t' 〉 =
        nondeterministically choose one from possibles
    T.view[x] = t'
    return v
```

- write messages that can satisfy this read must have time >= that timestamp
- this is NOT the “latest” write to x
    - b/c there is no shared understanding between thread of “latest” write

```
read(T, x):
    let t = T.view[x]
    let possibles = empty set
    for each ⟨ x' : v @ t' ⟩ in memory:
        if x' == x and t' >= t:
            insert ⟨ x' : v @ t' ⟩ into possibles
    let ⟨ _ : v @ t' ⟩ =
        nondeterministically choose one from possibles
    T.view[x] = t'
    return v
```

- have a set of writes that could possibly satisfy this read
- choose one nondeterministically

```
read(T, x):
    let t = T.view[x]
    let possibles = empty set
    for each 〈 x' : v @ t' 〉 in memory:
        if x' == x and t' >= t:
            insert 〈 x' : v @ t' 〉 into possibles
    let 〈 _ : v @ t' 〉 =
        nondeterministically choose one from possibles
    T.view[x] = t'
    return v
```

- update thread’s view to account for observing this new write message
- return the write message’s value to satisfy read

Example (SB)

x := 1	‖	y := 1
a := y		b := x

Memory
⟨ x : 0 @ 0 ⟩
⟨ y : 0 @ 0 ⟩

T_0.view	T_1.view
x @ 0	x @ 0
y @ 0	y @ 0

- revisit store buffering example
  - the paper moves fast, lets move slow
  - like a stepping debugger
- arrow is program counter
  - points to the next instruction to execute
- list of all messages in memory set
- show each thread's view of memory

Example (SB)

```
x := 1    ||    y := 1
a := y    ||    b := x
```

Memory
⟨ x : 0 @ 0 ⟩
⟨ y : 0 @ 0 ⟩

T_0.view	T_1.view
x @ 0	x @ 0
y @ 0	y @ 0

- remember:
    - observed 0 in load of \`y\` into \`a\`
    - observed 0 in load of \`x\` into \`b\`
    - we want our semantics to permit this
- without loss of generality
    - let's say first thread does its write first
- SLIDE — we get new write message in memory
- SLIDE — first thread's view is updated
- SLIDE — advance first thread's program counter
- then the second thread does its write
- SLIDE — we get a new write message in memory
- SLIDE — second thread's view is updated
- SLIDE — advance second thread's pc
- now either read can happen next
- first thread's read of \`y\`
    - its view of \`y\` still at timestamp 0
    - can select either write message to satisfy read
    - choose @0 message to see our desired election
- second thread is similar

Example (SB)

`x := 1`	`y := 1`
`a := y`	`b := x`

Memory
⟨ x : 0 @ 0 ⟩
⟨ y : 0 @ 0 ⟩
⟨ x : 1 @ 1 ⟩

T_0.view	T_1.view
x @ 0	x @ 0
y @ 0	y @ 0

- remember:
  - observed 0 in load of \`y\` into \`a\`
  - observed 0 in load of \`x\` into \`b\`
  - we want our semantics to permit this
- without loss of generality
  - let's say first thread does its write first
- SLIDE — we get new write message in memory
- SLIDE — first thread's view is updated
- SLIDE — advance first thread's program counter
- then the second thread does its write
- SLIDE — we get a new write message in memory
- SLIDE — second thread's view is updated
- SLIDE — advance second thread's pc
- now either read can happen next
- first thread's read of \`y\`
  - its view of \`y\` still at timestamp 0
  - can select either write message to satisfy read
  - choose @0 message to see our desired election
- second thread is similar

Example (SB)

x := 1 a := y	∥	y := 1 b := x

Memory
⟨ x : 0 @ 0 ⟩
⟨ y : 0 @ 0 ⟩
⟨ x : 1 @ 1 ⟩

T_0.view	T_1.view
x @ ~~0~~ 1	x @ 0
y @ 0	y @ 0

- remember:
  - observed 0 in load of `y` into `a`
  - observed 0 in load of `x` into `b`
  - we want our semantics to permit this
- without loss of generality
  - let’s say first thread does its write first
- SLIDE — we get new write message in memory
- SLIDE — first thread’s view is updated
- SLIDE — advance first thread’s program counter
- then the second thread does its write
- SLIDE — we get a new write message in memory
- SLIDE — second thread’s view is updated
- SLIDE — advance second thread’s pc
- now either read can happen next
- first thread’s read of `y`
  - its view of `y` still at timestamp 0
  - can select either write message to satisfy read
  - choose @0 message to see our desired election
- second thread is similar

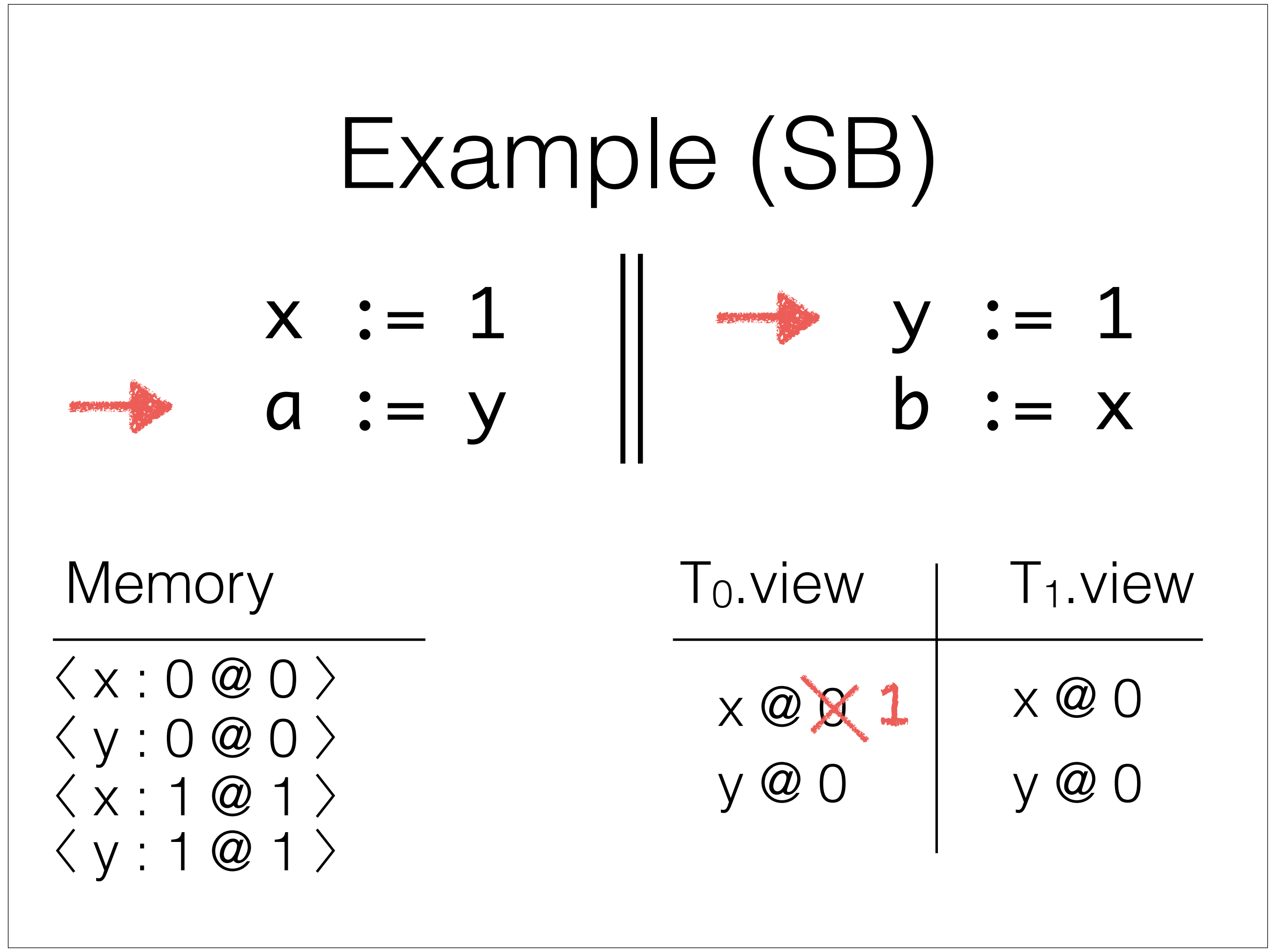
Example (SB)

```
  x := 1    ||  → y := 1
→ a := y    ||    b := x
```

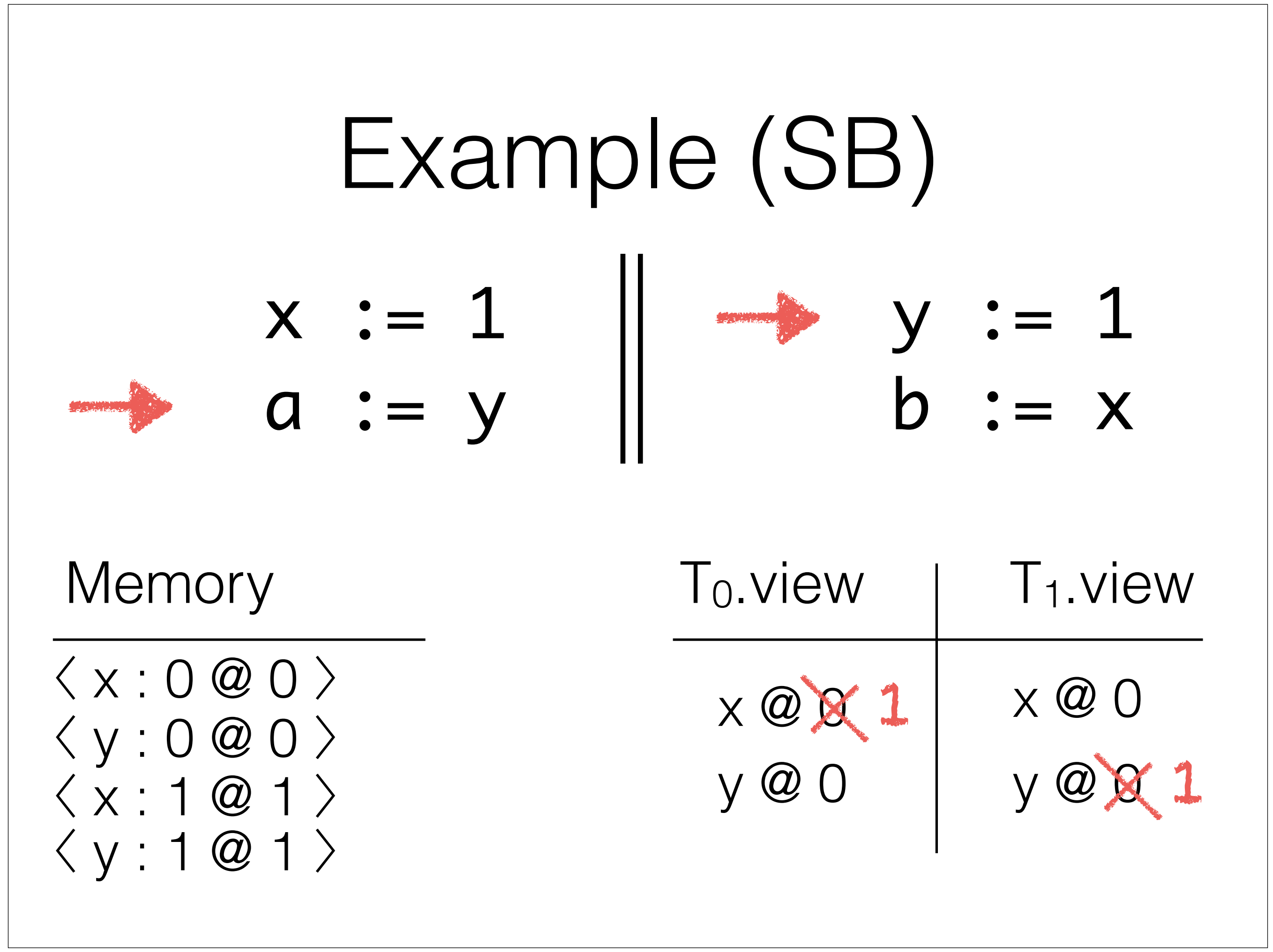
Memory
⟨ x : 0 @ 0 ⟩
⟨ y : 0 @ 0 ⟩
⟨ x : 1 @ 1 ⟩

T_0.view	T_1.view
x @ ~~0~~ 1	x @ 0
y @ 0	y @ 0

- remember:
  - observed 0 in load of `y` into `a`
  - observed 0 in load of `x` into `b`
  - we want our semantics to permit this
- without loss of generality
  - let's say first thread does its write first
- SLIDE — we get new write message in memory
- SLIDE — first thread's view is updated
- SLIDE — advance first thread's program counter
- then the second thread does its write
- SLIDE — we get a new write message in memory
- SLIDE — second thread's view is updated
- SLIDE — advance second thread's pc
- now either read can happen next
- first thread's read of `y`
  - its view of `y` still at timestamp 0
  - can select either write message to satisfy read
  - choose @0 message to see our desired election
- second thread is similar



- remember:
  - observed 0 in load of `y` into `a`
  - observed 0 in load of `x` into `b`
  - we want our semantics to permit this
- without loss of generality
  - let's say first thread does its write first
- SLIDE — we get new write message in memory
- SLIDE — first thread's view is updated
- SLIDE — advance first thread's program counter
- then the second thread does its write
- SLIDE — we get a new write message in memory
- SLIDE — second thread's view is updated
- SLIDE — advance second thread's pc
- now either read can happen next
- first thread's read of `y`
  - its view of `y` still at timestamp 0
  - can select either write message to satisfy read
  - choose @0 message to see our desired election
- second thread is similar



- remember:
    - observed 0 in load of `y` into `a`
    - observed 0 in load of `x` into `b`
    - we want our semantics to permit this
- without loss of generality
    - let's say first thread does its write first
- SLIDE — we get new write message in memory
- SLIDE — first thread's view is updated
- SLIDE — advance first thread's program counter
- then the second thread does its write
- SLIDE — we get a new write message in memory
- SLIDE — second thread's view is updated
- SLIDE — advance second thread's pc
- now either read can happen next
- first thread's read of `y`
    - its view of `y` still at timestamp 0
    - can select either write message to satisfy read
    - choose @0 message to see our desired election
- second thread is similar

Example (SB)

$$
\begin{array}{lcr}
x := 1 & \| & y := 1 \\
\rightarrow a := y & \| & \rightarrow b := x
\end{array}
$$

Memory

⟨ x : 0 @ 0 ⟩
⟨ y : 0 @ 0 ⟩
⟨ x : 1 @ 1 ⟩
⟨ y : 1 @ 1 ⟩

T_0.view	T_1.view
x @ ~~0~~ 1	x @ 0
y @ 0	y @ ~~0~~ 1

- remember:
  - observed 0 in load of `y` into `a`
  - observed 0 in load of `x` into `b`
  - we want our semantics to permit this
- without loss of generality
  - let's say first thread does its write first
- SLIDE — we get new write message in memory
- SLIDE — first thread's view is updated
- SLIDE — advance first thread's program counter
- then the second thread does its write
- SLIDE — we get a new write message in memory
- SLIDE — second thread's view is updated
- SLIDE — advance second thread's pc
- now either read can happen next
- first thread's read of `y`
  - its view of `y` still at timestamp 0
  - can select either write message to satisfy read
  - choose @0 message to see our desired election
- second thread is similar

Example (LB)

```
a := x    ||    x := y
y := 1    ||
```

Memory

⟨ x : 0 @ 0 ⟩
⟨ y : 0 @ 0 ⟩

T_0.view	T_1.view
x @ 0	x @ 0
y @ 0	y @ 0

- revisit load buffering example
    - want to observe `1` when loading `x`
    - b/c this is actual behavior of power & arm
- What if we do the read of `x` first?
    - only message that can satisfy read has value 0
    - won't work
- What if we do the second thread's read+write first?
    - SLIDE — add new write message to memory & update second thread's view
    - SLIDE — advance pc
    - now two messages can satisfy first thread's read
    - but both are `value = 0`
    - SLIDE — our semantics can't describe this yet, need to extend them

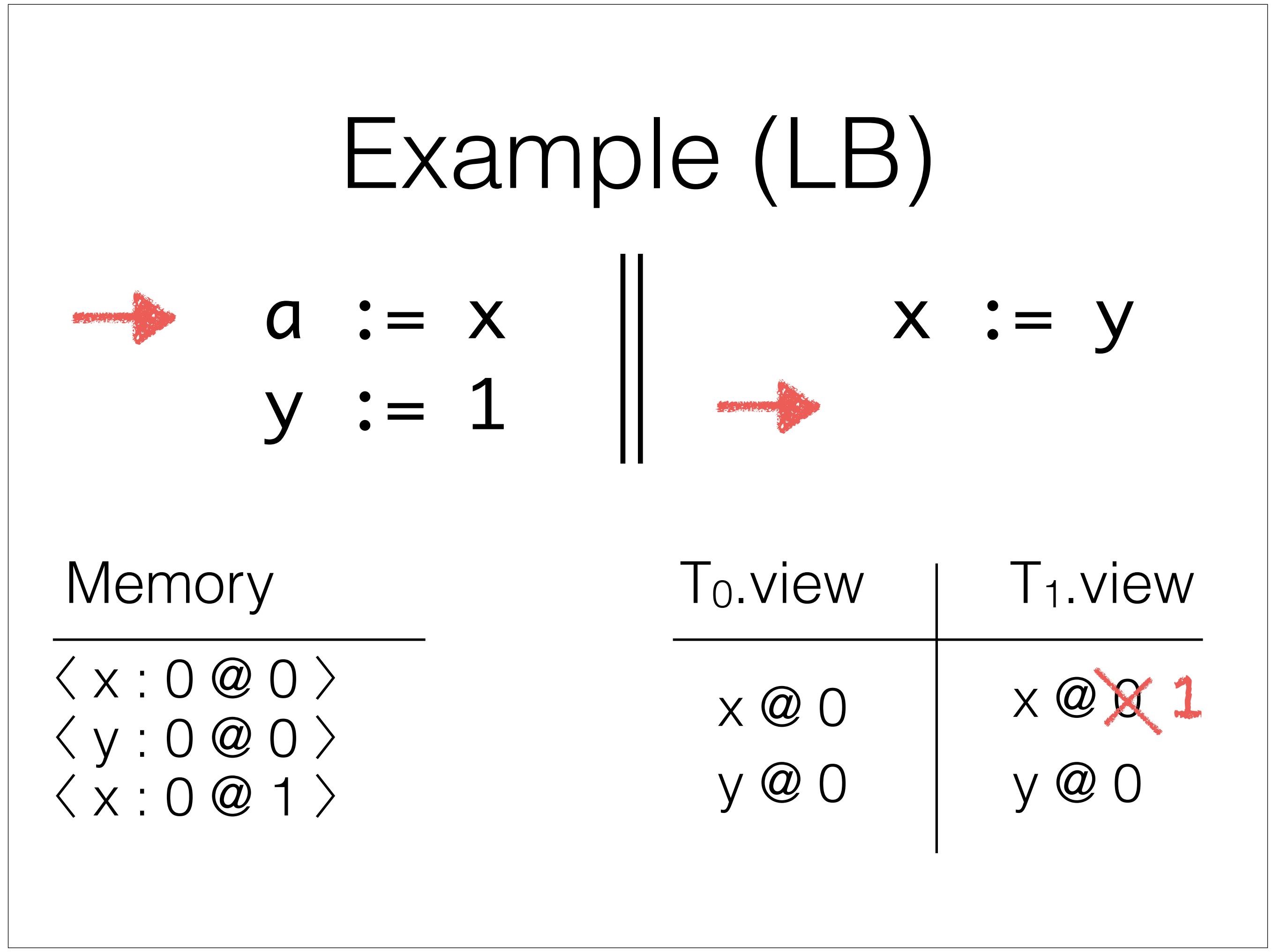
Example (LB)

```
a := x     ||   x := y
y := 1     ||
```

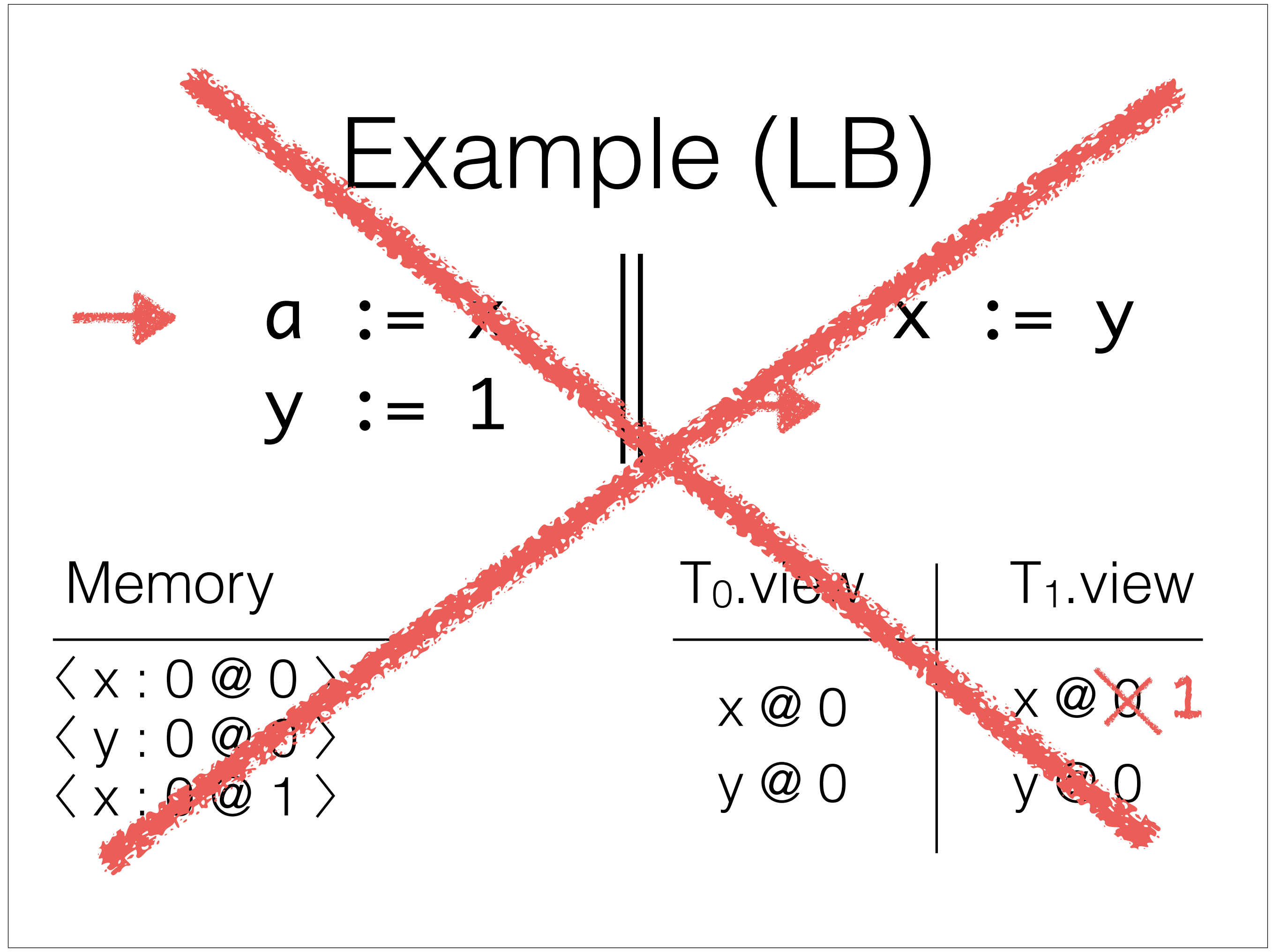
Memory
⟨ x : 0 @ 0 ⟩
⟨ y : 0 @ 0 ⟩
⟨ x : 0 @ 1 ⟩

T_0.view	T_1.view
x @ 0	x @ ~~0~~ 1
y @ 0	y @ 0

- revisit load buffering example
  - want to observe `1` when loading `x`
  - b/c this is actual behavior of power & arm
- What if we do the read of `x` first?
  - only message that can satisfy read has value 0
  - won't work
- What if we do the second thread's read+write first?
  - SLIDE — add new write message to memory & update second thread's view
  - SLIDE — advance pc
  - now two messages can satisfy first thread's read
  - but both are `value = 0`
  - SLIDE — our semantics can't describe this yet, need to extend them

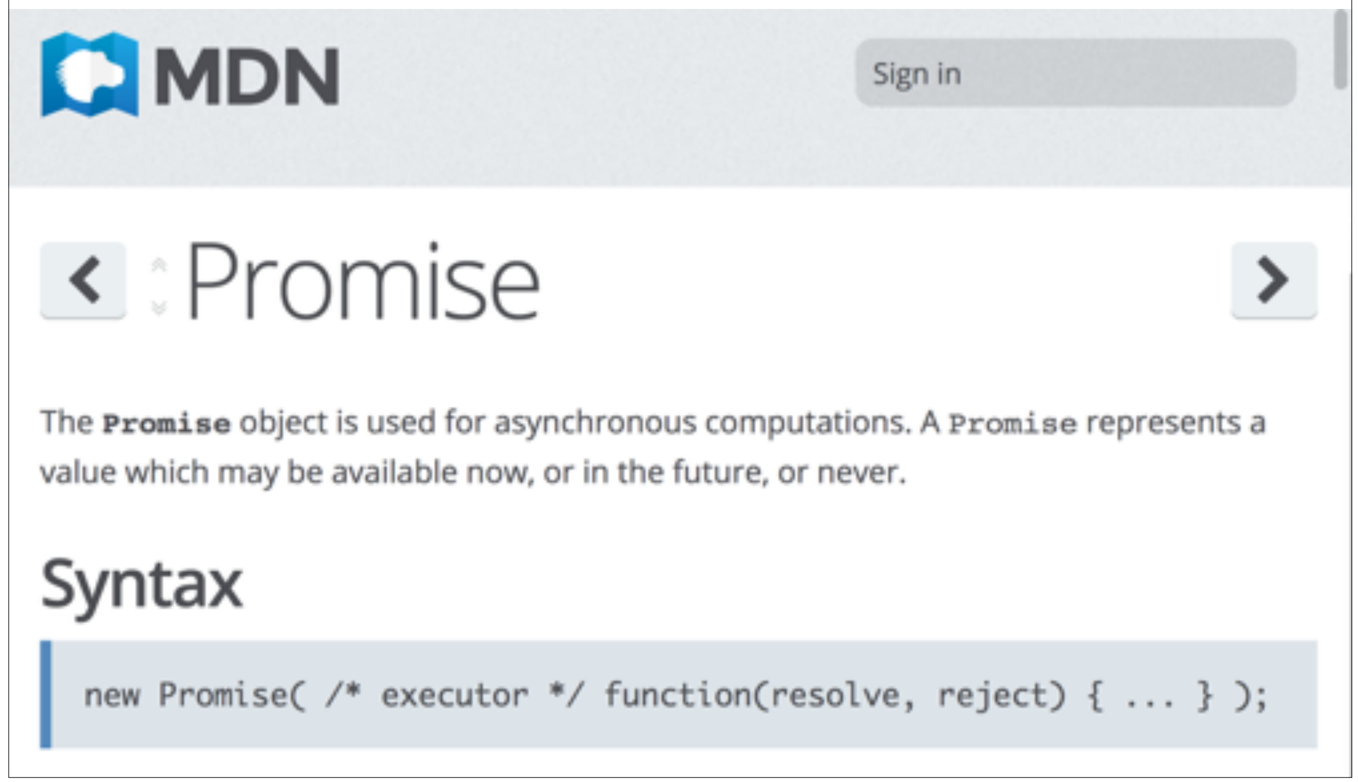


- revisit load buffering example
  - want to observe \`1\` when loading \`x\`
  - b/c this is actual behavior of power & arm
- What if we do the read of \`x\` first?
  - only message that can satisfy read has value 0
  - won't work
- What if we do the second thread's read+write first?
  - SLIDE — add new write message to memory & update second thread's view
  - SLIDE — advance pc
  - now two messages can satisfy first thread's read
  - but both are \`value = 0\`
  - SLIDE — our semantics can't describe this yet, need to extend them



- revisit load buffering example
  - want to observe `1` when loading `x`
  - b/c this is actual behavior of power & arm
- What if we do the read of `x` first?
  - only message that can satisfy read has value 0
  - won't work
- What if we do the second thread's read+write first?
  - SLIDE — add new write message to memory & update second thread's view
  - SLIDE — advance pc
  - now two messages can satisfy first thread's read
  - but both are `value = 0`
  - SLIDE — our semantics can't describe this yet, need to extend them

Introducing Promises



- JavaScript developers had the silver bullet all along! Promises!
- SLIDE — just kidding, not those promises

Introducing Promises

- JavaScript developers had the silver bullet all along! Promises!
- SLIDE — just kidding, not those promises

- thread can promise to write a value in future
  - must guarantee that it will fulfill promise
    - potential infinite loop / early exit / conditional / etc between now and promised write is NOT valid
  - after every step, re-verify it can still fulfill promised write
- SLIDE — promise equivalent to normal write form other threads' POV
  - adds write message to memory set
  - they can observe promised write in reads
- SLIDE — promising thread cannot use its promised write in reads!
  - single threaded program
  - we don't want to promise write `x := 1`
  - and then read that promised write in `a := x`
  - don't need to add special rules for this, existing timestamps are enough
  - if promised write was observed
    - it would update `T.view[x] = timestamp(promise)`
    - which would make fulfilling the write impossible, as write's timestamp must be > than view
- however, *can* indirectly observe promised write via other threads' reads+writes

- A thread can promise to write a value in the future
- Other threads can satisfy reads with that promise

- thread can promise to write a value in future
    - must guarantee that it will fulfill promise
        - potential infinite loop / early exit / conditional / etc between now and promised write is NOT valid
    - after every step, re-verify it can still fulfill promised write
- SLIDE — promise equivalent to normal write form other threads' POV
    - adds write message to memory set
    - they can observe promised write in reads
- SLIDE — promising thread cannot use its promised write in reads!
    - single threaded program
    - we don't want to promise write `x := 1`
    - and then read that promised write in `a := x`
    - don't need to add special rules for this, existing timestamps are enough
    - if promised write was observed
        - it would update `T.view[x] = timestamp(promise)`
        - which would make fulfilling the write impossible, as write's timestamp must be > than view
- however, *can* indirectly observe promised write via other threads' reads+writes

- A thread can promise to write a value in the future
- Other threads can satisfy reads with that promise
- Promising thread *cannot!*

```
a := x // 1? No!!!
x := 1
```

- thread can promise to write a value in future
  - must guarantee that it will fulfill promise
    - potential infinite loop / early exit / conditional / etc between now and promised write is NOT valid
  - after every step, re-verify it can still fulfill promised write
- SLIDE — promise equivalent to normal write form other threads' POV
  - adds write message to memory set
  - they can observe promised write in reads
- SLIDE — promising thread cannot use its promised write in reads!
  - single threaded program
  - we don't want to promise write `x := 1`
  - and then read that promised write in `a := x`
  - don't need to add special rules for this, existing timestamps are enough
  - if promised write was observed
    - it would update `T.view[x] = timestamp(promise)`
    - which would make fulfilling the write impossible, as write's timestamp must be > than view
- however, *can* indirectly observe promised write via other threads' reads+writes

```
promise(T, x, v, t):
    insert ⟨ x : v @ t ⟩ into memory
    insert ⟨ x : v @ t ⟩ into T.prm
```

- Again with the pseudocode
- T = thread making the promise
- x = memory location
- v = value
- t = timestamp

```
promise(T, x, v, t):
    insert ⟨ x : v @ t ⟩ into memory
    insert ⟨ x : v @ t ⟩ into T.prm
```

- just like a “normal” write, message is put into memory

```
promise(T, x, v, t):
    insert ⟨ x : v @ t ⟩ into memory
    insert ⟨ x : v @ t ⟩ into T.prm
```

- also added to thread's local set of unfulfilled promises, T.prm
- T.prm lets us know what we can or can't fulfill
- must be empty at the end of thread's execution
    - if it isn't that means we didn't fulfill all promises

```
fulfill(T, x, v, t):
    remove ⟨ x : v @ t ⟩ from T.prm
    T.view[x] = t
```

- write kind of already happened — fulfill brings thread up to date with its promised write
- removes promised write from unfulfilled promise set
- updates local view of memory location x

```
write(T, x, v):
    let t = new unique timestamp > T.view[x]
    promise(T, x, v, t)
    fulfill(T, x, v, t)
```

- write is no longer “special”
- just a promise + fulfill back-to-back

```
read(T, x):
    let min_t = T.view[x]

    let max_t = infinity
    for each 〈 x' : v @ t 〉 in T.prm:
        if x' == x:
            max_t = minimum(max_t, t)

    let possibles = empty set
    for each 〈 x' : v @ t 〉 in memory:
        if x' == x and min_t <= t and t < max_t:
            insert 〈 x' : v @ t 〉 into possibles

    let 〈 _ : v @ t 〉 =
        nondeterministically choose one from possibles
    T.view[x] = t
    return v
```

- read needs to be revised a bit to play nice with promises
- as before, building up the set of write messages that *could* *possibly* satisfy this read
- but now we have some extra checks
- this pseudocode is *descriptive* only
    - the new checks don't appear in the formal operational semantics

```
read(T, x):
    let min_t = T.view[x]

    let max_t = infinity
    for each 〈 x' : v @ t 〉 in T.prm:
        if x' == x:
            max_t = minimum(max_t, t)

    let possibles = empty set
    for each 〈 x' : v @ t 〉 in memory:
        if x' == x and min_t <= t and t < max_t:
            insert 〈 x' : v @ t 〉 into possibles

    let 〈 _ : v @ t 〉 =
        nondeterministically choose one from possibles
    T.view[x] = t
    return v
```

- minimum timestamp a write message can have to satisfy our read
- this is all we had before

```
read(T, x):
    let min_t = T.view[x]

    let max_t = infinity
    for each 〈 x' : v @ t 〉 in T.prm:
        if x' == x:
            max_t = minimum(max_t, t)

    let possibles = empty set
    for each 〈 x' : v @ t 〉 in memory:
        if x' == x and min_t <= t and t < max_t:
            insert 〈 x' : v @ t 〉 into possibles

    let 〈 _ : v @ t 〉 =
        nondeterministically choose one from possibles
    T.view[x] = t
    return v
```

- unlike before, now defining upper bound on messages' timestamps
- timestamp of message that satisfies read must be less than minimum promised timestamp
    - otherwise promise would be unfulfillable — like single threaded example earlier
- implicitly disallowed in operational semantics
    - failure to obey would make promises unfulfillable

```
read(T, x):
    let min_t = T.view[x]

    let max_t = infinity
    for each 〈 x' : v @ t 〉 in T.prm:
        if x' == x:
            max_t = minimum(max_t, t)

    let possibles = empty set
    for each 〈 x' : v @ t 〉 in memory:
        if x' == x and min_t <= t and t < max_t:
            insert 〈 x' : v @ t 〉 into possibles

    let 〈 _ : v @ t 〉 =
        nondeterministically choose one from possibles
    T.view[x] = t
    return v
```

- have min and max bounds, so get every write message that falls within them
- this is set of write messages that could possibly fulfill this read

```
read(T, x):
    let min_t = T.view[x]

    let max_t = infinity
    for each 〈 x' : v @ t 〉 in T.prm:
        if x' == x:
            max_t = minimum(max_t, t)

    let possibles = empty set
    for each 〈 x' : v @ t 〉 in memory:
        if x' == x and min_t <= t and t < max_t:
            insert 〈 x' : v @ t 〉 into possibles

    let 〈 _ : v @ t 〉 =
        nondeterministically choose one from possibles
    T.view[x] = t
    return v
```

- from here, proceeds as we originally did
- nondeterministically choose one message from our possibilities
- update thread’s local view of memory location x
- return the value

Example (LB)

```
a := x   ||   x := y
y := 1
```

Memory	T_0.view	T_1.view	T_0.prm	T_1.prm
$\langle x : 0 @ 0 \rangle$	x @ 0	x @ 0		
$\langle y : 0 @ 0 \rangle$	y @ 0	y @ 0		

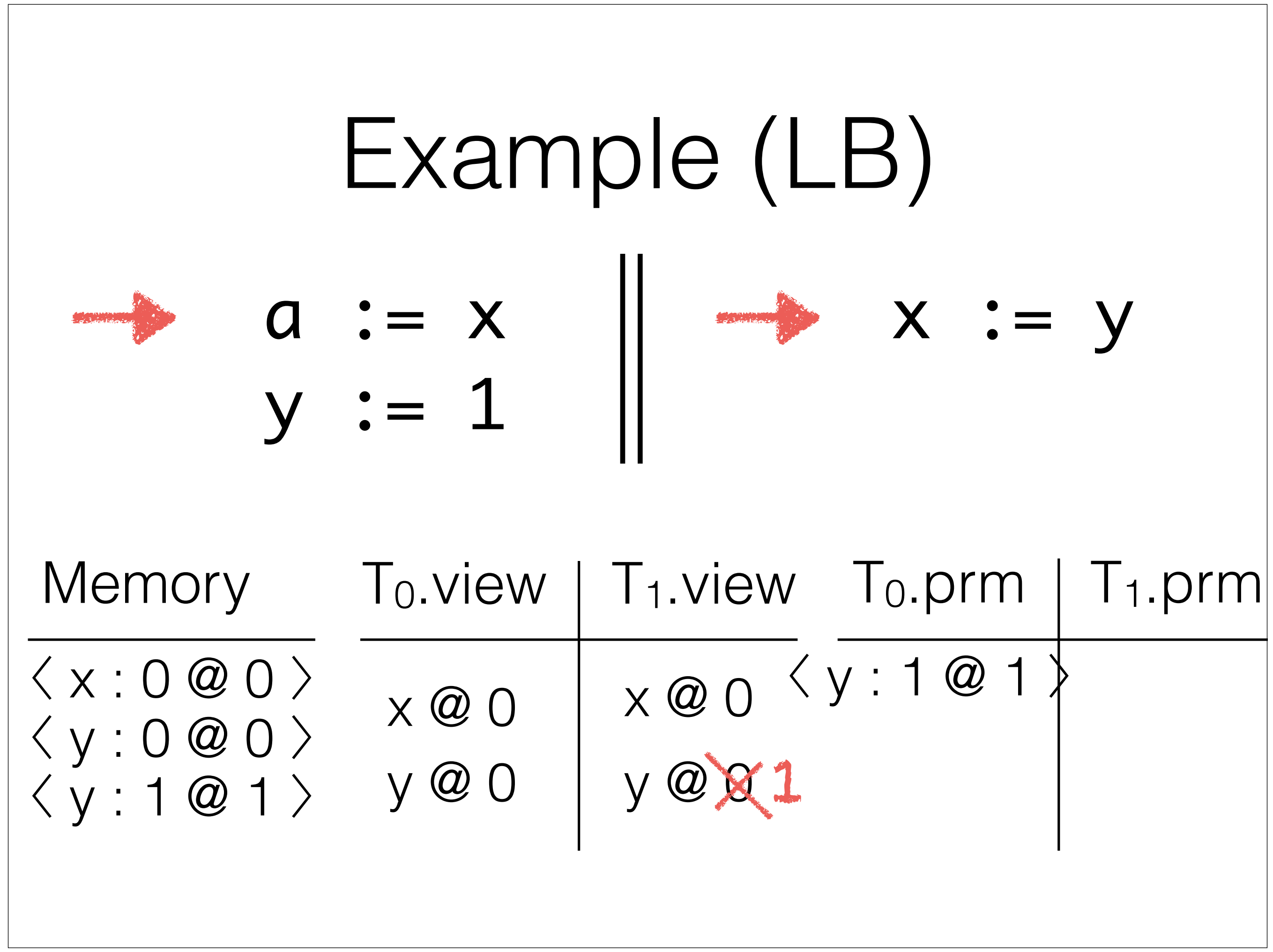
- revisit load buffering example
    - remember: want to <u>permit</u> observing `1` when loading memory location `x` into register `a`
- SLIDE — first thread promises to write `y := 1` at timestamp 1
    - this also adds the promised write to memory
    - but does NOT advance pc!
- SLIDE — second thread reads y
    - possibilities include y @ 0 and y @ 1
    - choose the latter in this execution
    - its view of y is updated
- SLIDE — writes the newly read value back to x with timestamp 1
    - technically this involves promise+fulfill
    - and second thread's view of x is also updated
- SLIDE — and it's pc advances and second thread is finished
- SLIDE — first thread reads x
    - two possibilities: @0 and @1
    - we choose @1 to observe `x = 1`
    - update local view of `x`
- SLIDE
    - not done yet — still have to fulfill promised write
    - local view of `y` is @ 0

Example (LB)

```
a := x    ||    x := y
y := 1
```

Memory	T_0.view	T_1.view	T_0.prm	T_1.prm
⟨ x : 0 @ 0 ⟩	x @ 0	x @ 0	⟨ y : 1 @ 1 ⟩	
⟨ y : 0 @ 0 ⟩	y @ 0	y @ 0		
⟨ y : 1 @ 1 ⟩				

- revisit load buffering example
  - remember: want to permit observing `1` when loading memory location `x` into register `a`
- SLIDE — first thread promises to write `y := 1` at timestamp 1
  - this also adds the promised write to memory
  - but does NOT advance pc!
- SLIDE — second thread reads y
  - possibilities include y @ 0 and y @ 1
  - choose the latter in this execution
  - its view of y is updated
- SLIDE — writes the newly read value back to x with timestamp 1
  - technically this involves promise+fulfill
  - and second thread's view of x is also updated
- SLIDE — and it's pc advances and second thread is finished
- SLIDE — first thread reads x
  - two possibilities: @0 and @1
  - we choose @1 to observe `x = 1`
  - update local view of `x`
- SLIDE
  - not done yet — still have to fulfill promised write
  - local view of `y` is @ 0



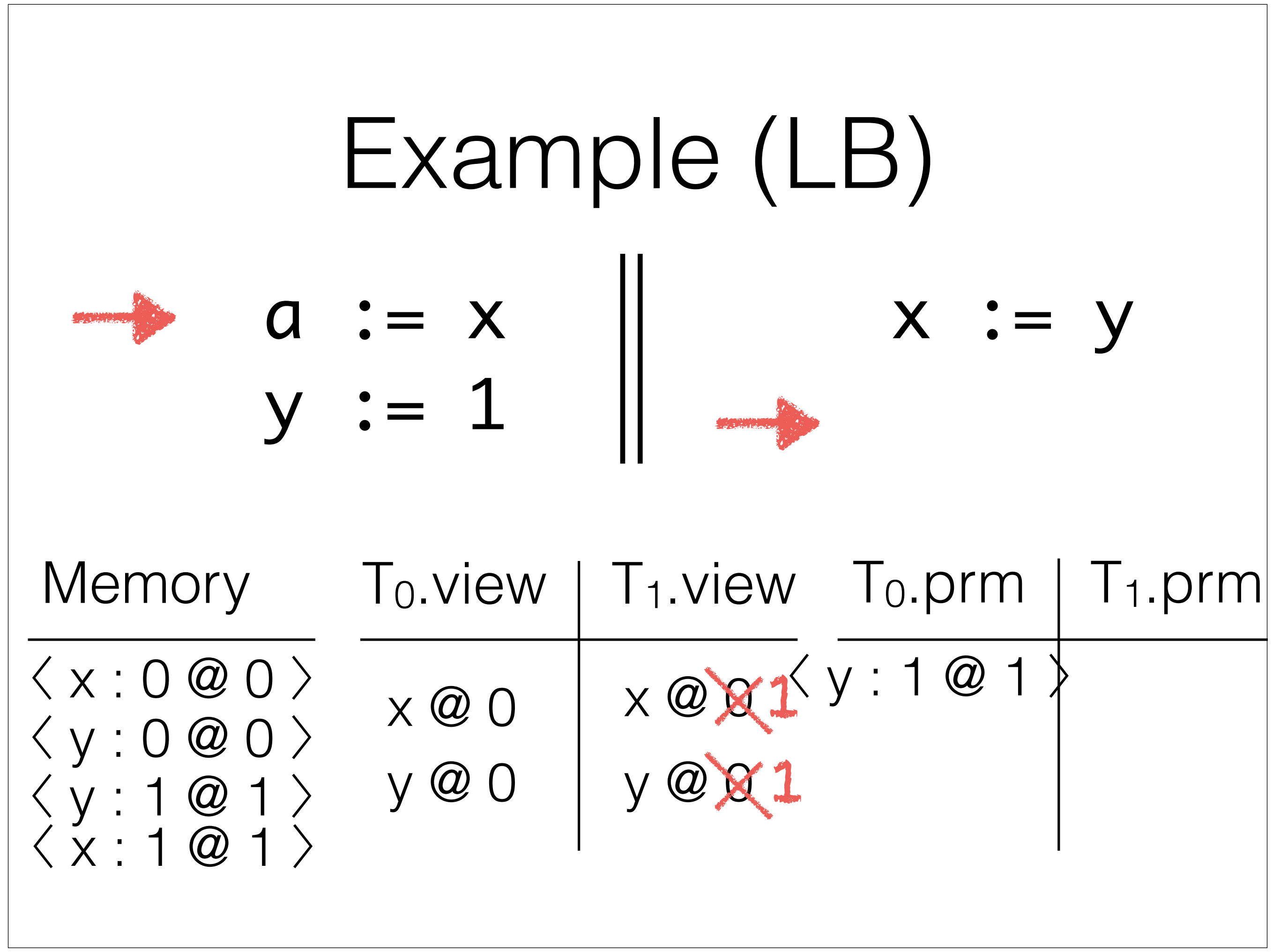
- revisit load buffering example
  - remember: want to <u>permit</u> observing \`1\` when loading memory location \`x\` into register \`a\`
- SLIDE — first thread promises to write \`y := 1\` at timestamp 1
  - this also adds the promised write to memory
  - but does NOT advance pc!
- SLIDE — second thread reads y
  - possibilities include y @ 0 and y @ 1
  - choose the latter in this execution
  - its view of y is updated
- SLIDE — writes the newly read value back to x with timestamp 1
  - technically this involves promise+fulfill
  - and second thread's view of x is also updated
- SLIDE — and it's pc advances and second thread is finished
- SLIDE — first thread reads x
  - two possibilities: @0 and @1
  - we choose @1 to observe \`x = 1\`
  - update local view of \`x\`
- SLIDE
  - not done yet — still have to fulfill promised write
  - local view of \`y\` is @ 0

Example (LB)

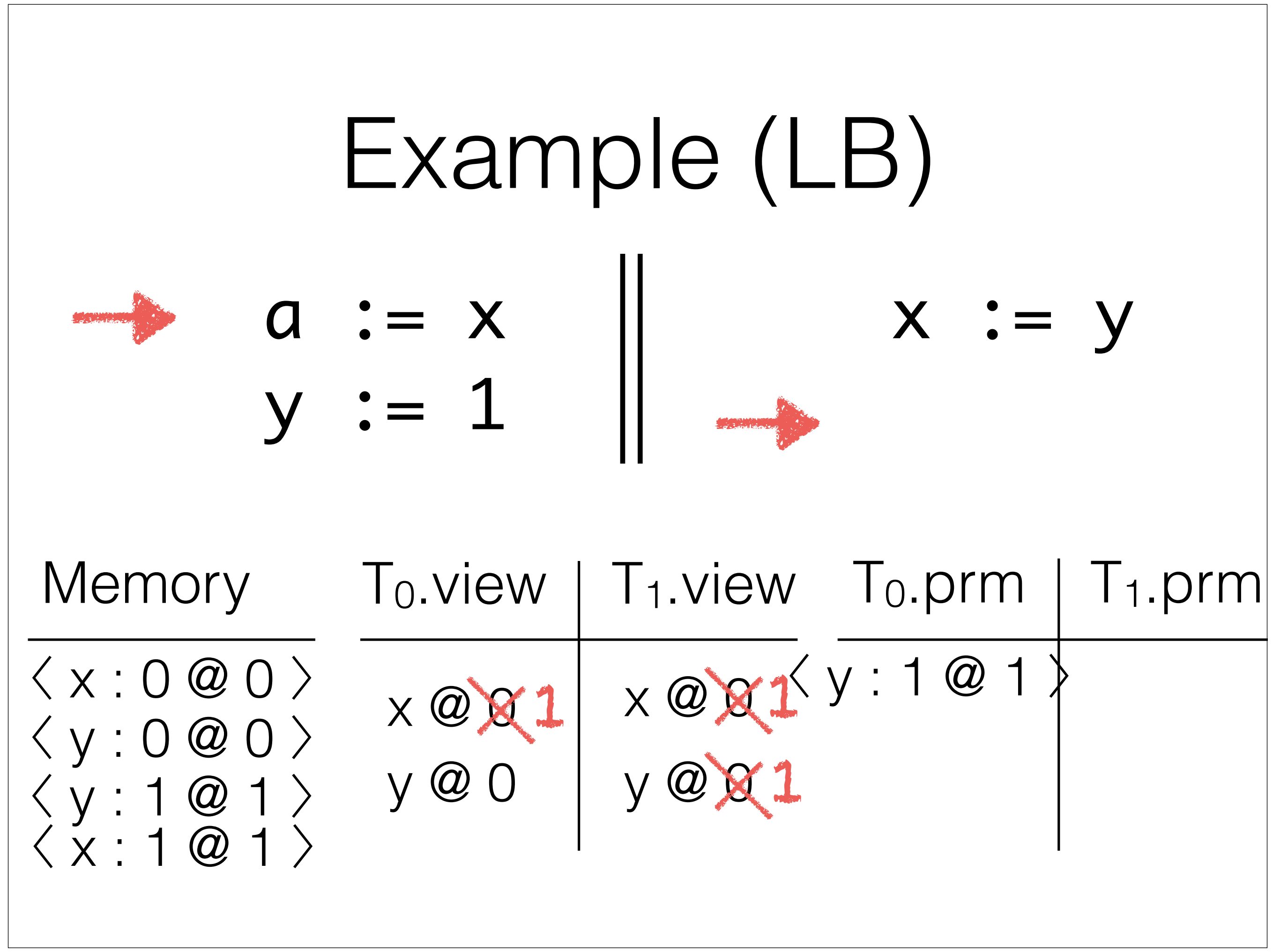
a := x	‖	x := y
y := 1	‖	

Memory	T_0.view	T_1.view	T_0.prm	T_1.prm
⟨ x : 0 @ 0 ⟩	x @ 0	x @ ~~0~~ 1	⟨ y : 1 @ 1 ⟩	
⟨ y : 0 @ 0 ⟩	y @ 0	y @ ~~0~~ 1		
⟨ y : 1 @ 1 ⟩				
⟨ x : 1 @ 1 ⟩				

- revisit load buffering example
    - remember: want to permit observing `1` when loading memory location `x` into register `a`
- SLIDE — first thread promises to write `y := 1` at timestamp 1
    - this also adds the promised write to memory
    - but does NOT advance pc!
- SLIDE — second thread reads y
    - possibilities include y @ 0 and y @ 1
    - choose the latter in this execution
    - its view of y is updated
- SLIDE — writes the newly read value back to x with timestamp 1
    - technically this involves promise+fulfill
    - and second thread's view of x is also updated
- SLIDE — and it's pc advances and second thread is finished
- SLIDE — first thread reads x
    - two possibilities: @0 and @1
    - we choose @1 to observe `x = 1`
    - update local view of `x`
- SLIDE
    - not done yet — still have to fulfill promised write
    - local view of `y` is @ 0



- revisit load buffering example
    - remember: want to permit observing `1` when loading memory location `x` into register `a`
- SLIDE — first thread promises to write `y := 1` at timestamp 1
    - this also adds the promised write to memory
    - but does NOT advance pc!
- SLIDE — second thread reads y
    - possibilities include y @ 0 and y @ 1
    - choose the latter in this execution
    - its view of y is updated
- SLIDE — writes the newly read value back to x with timestamp 1
    - technically this involves promise+fulfill
    - and second thread’s view of x is also updated
- SLIDE — and it’s pc advances and second thread is finished
- SLIDE — first thread reads x
    - two possibilities: @0 and @1
    - we choose @1 to observe `x = 1`
    - update local view of `x`
- SLIDE
    - not done yet — still have to fulfill promised write
    - local view of `y` is @ 0



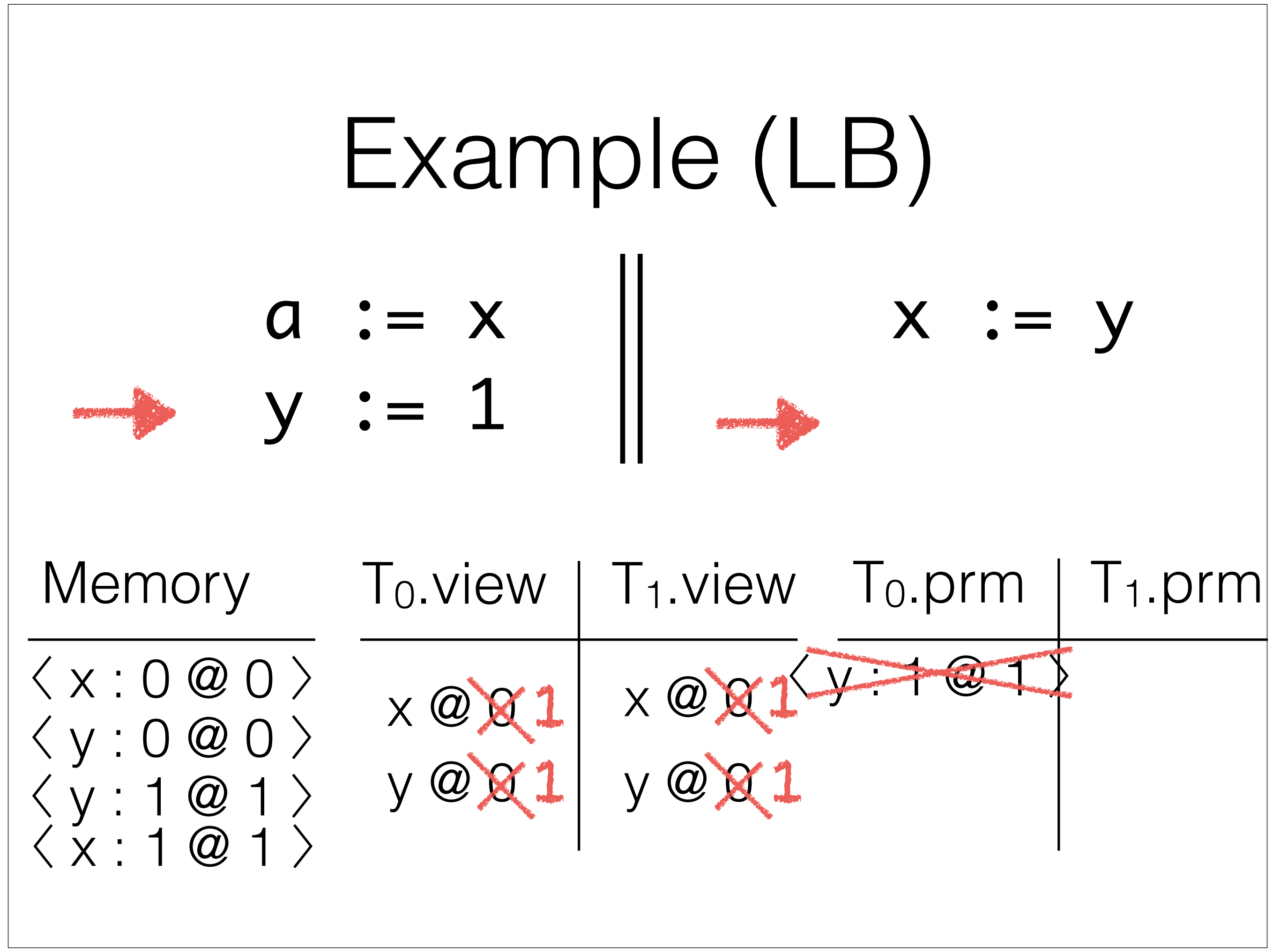
- revisit load buffering example
  - remember: want to permit observing `1` when loading memory location `x` into register `a`
- SLIDE — first thread promises to write `y := 1` at timestamp 1
  - this also adds the promised write to memory
  - but does NOT advance pc!
- SLIDE — second thread reads y
  - possibilities include y @ 0 and y @ 1
  - choose the latter in this execution
  - its view of y is updated
- SLIDE — writes the newly read value back to x with timestamp 1
  - technically this involves promise+fulfill
  - and second thread’s view of x is also updated
- SLIDE — and it’s pc advances and second thread is finished
- SLIDE — first thread reads x
  - two possibilities: @0 and @1
  - we choose @1 to observe `x = 1`
  - update local view of `x`
- SLIDE
  - not done yet — still have to fulfill promised write
  - local view of `y` is @ 0

Example (LB)

```
a := x   ||   x := y
y := 1   ||
```

Memory	T_0.view	T_1.view	T_0.prm	T_1.prm
⟨ x : 0 @ 0 ⟩	x @ ~~0~~ 1	x @ ~~0~~ 1	⟨ y : 1 @ 1 ⟩	
⟨ y : 0 @ 0 ⟩	y @ 0	y @ ~~0~~ 1		
⟨ y : 1 @ 1 ⟩				
⟨ x : 1 @ 1 ⟩				

- revisit load buffering example
  - remember: want to <u>permit</u> observing `1` when loading memory location `x` into register `a`
- SLIDE — first thread promises to write `y := 1` at timestamp 1
  - this also adds the promised write to memory
  - but does NOT advance pc!
- SLIDE — second thread reads y
  - possibilities include y @ 0 and y @ 1
  - choose the latter in this execution
  - its view of y is updated
- SLIDE — writes the newly read value back to x with timestamp 1
  - technically this involves promise+fulfill
  - and second thread's view of x is also updated
- SLIDE — and it's pc advances and second thread is finished
- SLIDE — first thread reads x
  - two possibilities: @0 and @1
  - we choose @1 to observe `x = 1`
  - update local view of `x`
- SLIDE
  - not done yet — still have to fulfill promised write
  - local view of `y` is @ 0



- revisit load buffering example
  - remember: want to permit observing `1` when loading memory location `x` into register `a`
- SLIDE — first thread promises to write `y := 1` at timestamp 1
  - this also adds the promised write to memory
  - but does NOT advance pc!
- SLIDE — second thread reads y
  - possibilities include y @ 0 and y @ 1
  - choose the latter in this execution
  - its view of y is updated
- SLIDE — writes the newly read value back to x with timestamp 1
  - technically this involves promise+fulfill
  - and second thread's view of x is also updated
- SLIDE — and it's pc advances and second thread is finished
- SLIDE — first thread reads x
  - two possibilities: @0 and @1
  - we choose @1 to observe `x = 1`
  - update local view of `x`
- SLIDE
  - not done yet — still have to fulfill promised write
  - local view of `y` is @ 0

Example (LB)

```
a := x     ||     x := y
y := 1     ||
```

Memory	T_0.view	T_1.view	T_0.prm	T_1.prm
$\langle x : 0 @ 0 \rangle$	x @ ~~0~~ 1	x @ ~~0~~ 1	~~$\langle y : 1 @ 1 \rangle$~~	
$\langle y : 0 @ 0 \rangle$	y @ ~~0~~ 1	y @ ~~0~~ 1		
$\langle y : 1 @ 1 \rangle$				
$\langle x : 1 @ 1 \rangle$				

- revisit load buffering example
    - remember: want to <u>permit</u> observing `1` when loading memory location `x` into register `a`
- SLIDE — first thread promises to write `y := 1` at timestamp 1
    - this also adds the promised write to memory
    - but does NOT advance pc!
- SLIDE — second thread reads y
    - possibilities include y @ 0 and y @ 1
    - choose the latter in this execution
    - its view of y is updated
- SLIDE — writes the newly read value back to x with timestamp 1
    - technically this involves promise+fulfill
    - and second thread's view of x is also updated
- SLIDE — and it's pc advances and second thread is finished
- SLIDE — first thread reads x
    - two possibilities: @0 and @1
    - we choose @1 to observe `x = 1`
    - update local view of `x`
- SLIDE
    - not done yet — still have to fulfill promised write
    - local view of `y` is @ 0

Why are time stamps dense?

```
x := 1   ||   x := 3
x := 2   ||
```

* "dense" = infinite timestamps between t_1 and t_2 where t_1 != t_2
* SLIDE — first thread promises ⟨ x : 2 @ 2 ⟩
    * before it can fulfill that promise, it must write ⟨ x : 1 @ 1 ⟩
* SLIDE — but what if second thread writes ⟨ x : 3 @ 1 ⟩ before first thread writes ⟨ x : 1 @ 1 ⟩?
    * now first thread can't write ⟨ x : 1 @ 1 ⟩ because timestamp 1 is already taken
    * but also can't use timestamp 2 because it needs to fulfill its promise at timestamp 2
* SLIDE — choose some timestamp t where 1 < t < 2 — eg 1.5
    * there ALWAYS exists such a `t` because of dense property

Why are time stamps dense?

```
x := 1    ||    x := 3
x := 2    ||
```

- T_0 promises ⟨ x : 2 @ 2 ⟩

- “dense” = infinite timestamps between t_1 and t_2 where t_1 != t_2
- SLIDE — first thread promises ⟨ x : 2 @ 2 ⟩
  - before it can fulfill that promise, it must write ⟨ x : 1 @ 1 ⟩
- SLIDE — but what if second thread writes ⟨ x : 3 @ 1 ⟩ before first thread writes ⟨ x : 1 @ 1 ⟩?
  - now first thread can’t write ⟨ x : 1 @ 1 ⟩ because timestamp 1 is already taken
  - but also can’t use timestamp 2 because it needs to fulfill its promise at timestamp 2
- SLIDE — choose some timestamp t where 1 < t < 2 — eg 1.5
  - there ALWAYS exists such a `t` because of dense property

Why are time stamps dense?

```
x := 1   ||   x := 3
x := 2   ||
```

- T_0 promises ⟨ x : 2 @ 2 ⟩
- T_1 writes ⟨ x : 3 @ 1 ⟩

- “dense” = infinite timestamps between t_1 and t_2 where $t_1 \neq t_2$
- SLIDE — first thread promises ⟨ x : 2 @ 2 ⟩
  - before it can fulfill that promise, it must write ⟨ x : 1 @ 1 ⟩
- SLIDE — but what if second thread writes ⟨ x : 3 @ 1 ⟩ before first thread writes ⟨ x : 1 @ 1 ⟩?
  - now first thread can’t write ⟨ x : 1 @ 1 ⟩ because timestamp 1 is already taken
  - but also can’t use timestamp 2 because it needs to fulfill its promise at timestamp 2
- SLIDE — choose some timestamp t where 1 < t < 2 — eg 1.5
  - there ALWAYS exists such a `t` because of dense property

Why are time stamps dense?

```
x := 1   ||   x := 3
x := 2
```

- T_0 promises ⟨ x : 2 @ 2 ⟩
- T_1 writes ⟨ x : 3 @ 1 ⟩
- T_0 writes ⟨ x : 1 @ t ⟩ where $1 < t < 2$

* "dense" = infinite timestamps between t_1 and t_2 where t_1 != t_2
* SLIDE — first thread promises ⟨ x : 2 @ 2 ⟩
    * before it can fulfill that promise, it must write ⟨ x : 1 @ 1 ⟩
* SLIDE — but what if second thread writes ⟨ x : 3 @ 1 ⟩ before first thread writes ⟨ x : 1 @ 1 ⟩?
    * now first thread can't write ⟨ x : 1 @ 1 ⟩ because timestamp 1 is already taken
    * but also can't use timestamp 2 because it needs to fulfill its promise at timestamp 2
* SLIDE — choose some timestamp t where 1 < t < 2 — eg 1.5
    * there ALWAYS exists such a `t` because of dense property

Wait — There's More!

- Atomics and fences
- Mechanized proofs
- DRF guarantees
- Compilation to TSO and Power

* more good stuff in paper, we don't have time
* extend the model with semantics for release/acquire and SC atomics and fences
* machine checked proofs of correctness implemented in Coq
    * this is an area where we *know* we need formal proofs and we need them checked correct
* details on how data-race free programs (using correct locks/atomics/synchronization) are unaffected
* talk about how they've compiled these semantics into x86 TSO and Power
    * they left ARM for future work, and intend to do that soon

Is *Promising* Perfect?

- No thread inlining
- Limited code motion

* disclaimer: I'm not really qualified to critique this model and discuss its limitations
* thread inlining is where one thread executes all of another thread's work
    * this can make promises unfulfillable
    * maybe has implications for work stealing? honestly not sure
* the per-location SC makes optimizations like LICM difficult (maybe impossible?)
* SLIDE

Is *Promising* Perfect?

- No thread inlining
- Limited code motion
- … No, but it sure is promising!

* disclaimer: I'm not really qualified to critique this model and discuss its limitations
* thread inlining is where one thread executes all of another thread's work
    * this can make promises unfulfillable
    * maybe has implications for work stealing? honestly not sure
* the per-location SC makes optimizations like LICM difficult (maybe impossible?)
* SLIDE

@pwlpdx

THANK YOU!!

@fitzgen