

Semantics, languages and algorithms for multicore programming

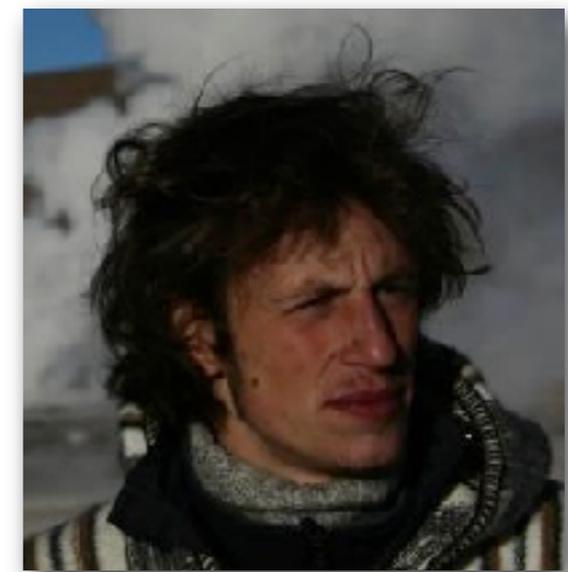
Albert Cohen



Luc Maranget



Francesco Zappa Nardelli



Concurrency, in theory

Example: 2-way Buffers

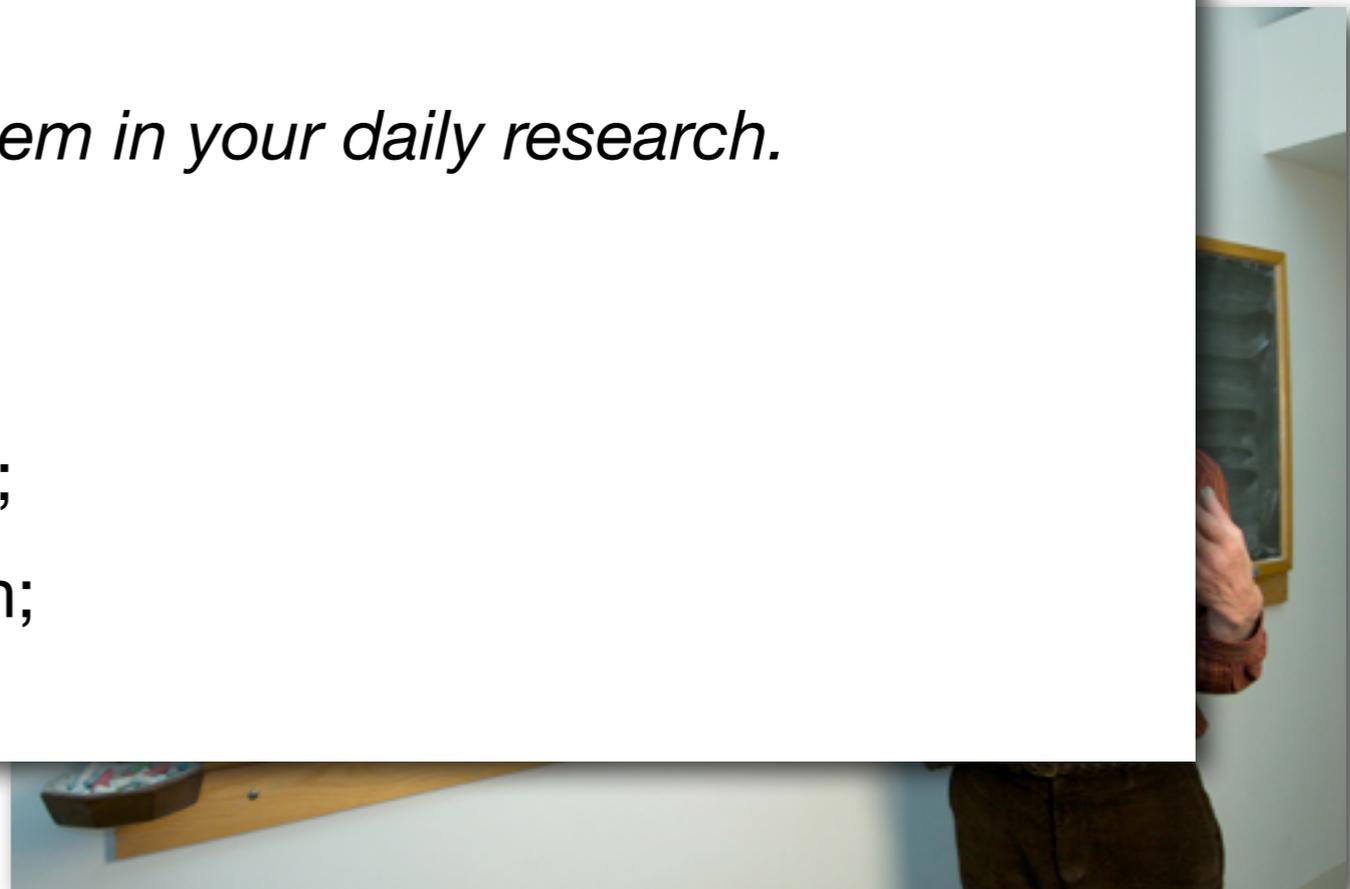
Concurrency theory is fundamental

Many of the concepts and techniques developed in 25 years of study of concurrency theory are fundamental.

You will reuse them in your daily research.

Just some examples:

- labelled transition systems;
- simulation and bisimulation;
- contextual equivalences.



Concurrency, in practice

in practice

sequential code, interaction via shared memory, some OS calls.

Libraries may provide some abstractions (e.g. message passing).
However, somebody must still implement these libraries. And...

Programming is hard:
subtle algorithms, awful corner cases.

Testing is hard:
some behaviours are observed rarely and difficult to reproduce.

Warm-up: let's implement a shared stack.

excerpt from
www.javaconcurrencyinpractice.com

```
return instance;
}
}
class ExpensiveObject { }
```

Setup

A program is composed by *threads* that communicate by writing and reading in a *shared memory*. No assumptions about the relative speed of the threads.

In this example we will use a *mild variant* of the *C programming language*:

- local variables: x, y, \dots (allocated on the stack, local to each thread)
- global variables: Top, H, \dots (allocated on the heap, shared between threads)
- data structures: arrays $H[i]$, records $n = t \rightarrow t1, \dots$
- an atomic *compare-and-swap* operation (e.g. `CMPXCHG` on x86):

```
bool CAS (value_t *addr, value_t exp, value_t new) {
    atomic {
        if (*addr == exp) then { *addr = new; return true; }
        else return false;
    }
}
```

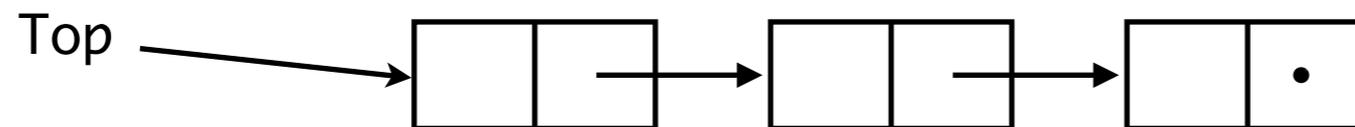
A stack

We implement a stack using a list living in the heap:

- each entry of the stack is a record of two fields:

```
typedef struct entry { value hd; entry *tl } entry
```

- the top of the stack is pointed by Top.



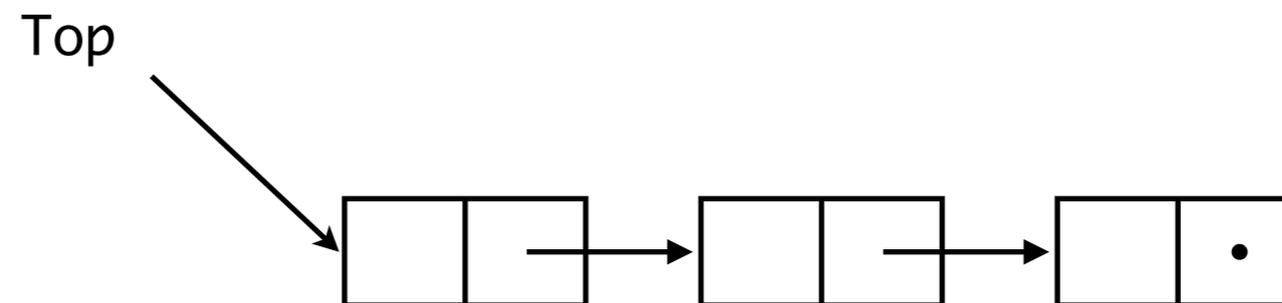
```
pop () {  
    t = Top;  
    if (t != nil)  
        Top = t->tl;  
    return t;  
}
```

```
push (b) {  
    b->tl = Top;  
    Top = b;  
    return true;  
}
```

A sequential stack: demo

```
pop ( ) {  
    t = Top;  
    if (t != nil)  
        Top = t->tl;  
    return t;  
}
```

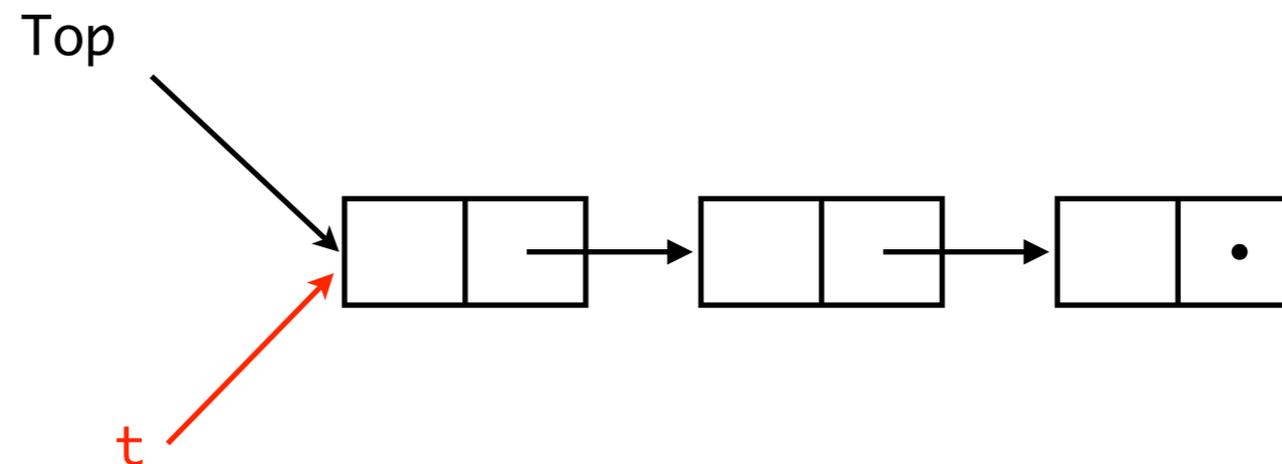
```
push (b) {  
    b->tl = Top;  
    Top = b;  
    return true;  
}
```



A sequential stack: pop ()

```
pop ( ) {  
    t = Top;  
    if (t != nil)  
        Top = t->tl;  
    return t;  
}
```

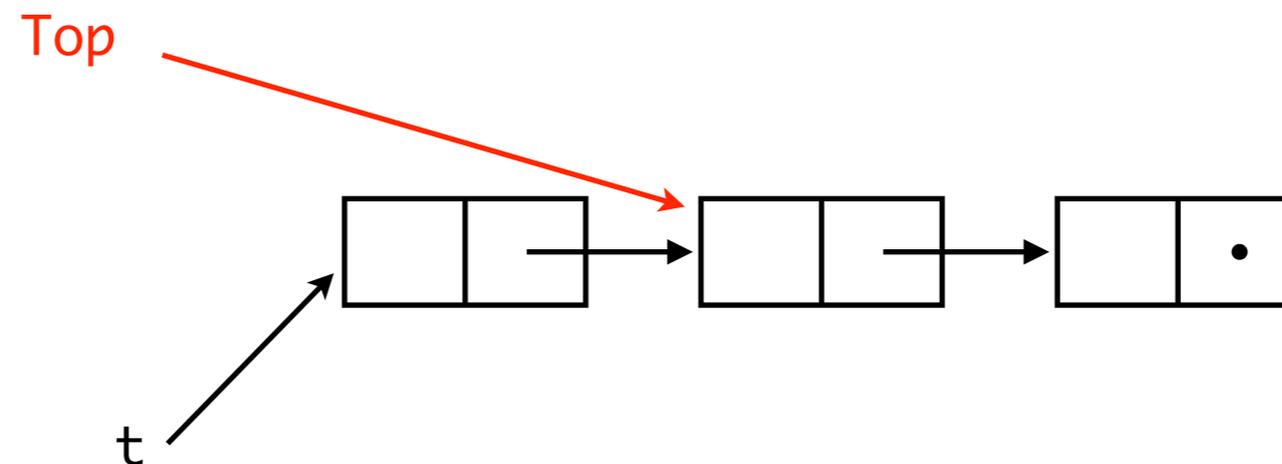
```
push (b) {  
    b->tl = Top;  
    Top = b;  
    return true;  
}
```



A sequential stack: pop ()

```
pop ( ) {  
  t = Top;  
  if (t != nil)  
    Top = t->tl;  
  return t;  
}
```

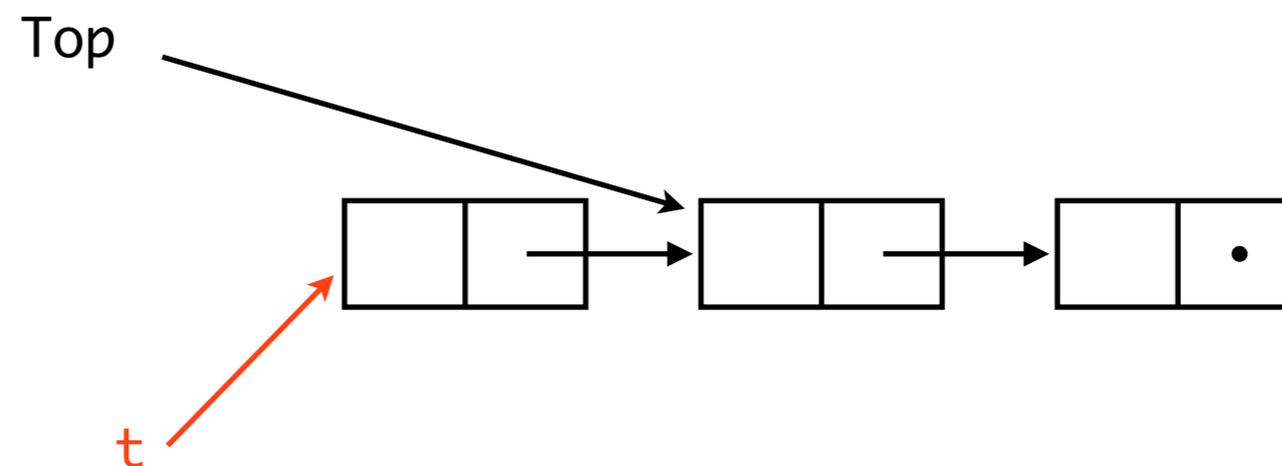
```
push (b) {  
  b->tl = Top;  
  Top = b;  
  return true;  
}
```



A sequential stack: pop ()

```
pop ( ) {  
    t = Top;  
    if (t != nil)  
        Top = t->tl;  
    return t;  
}
```

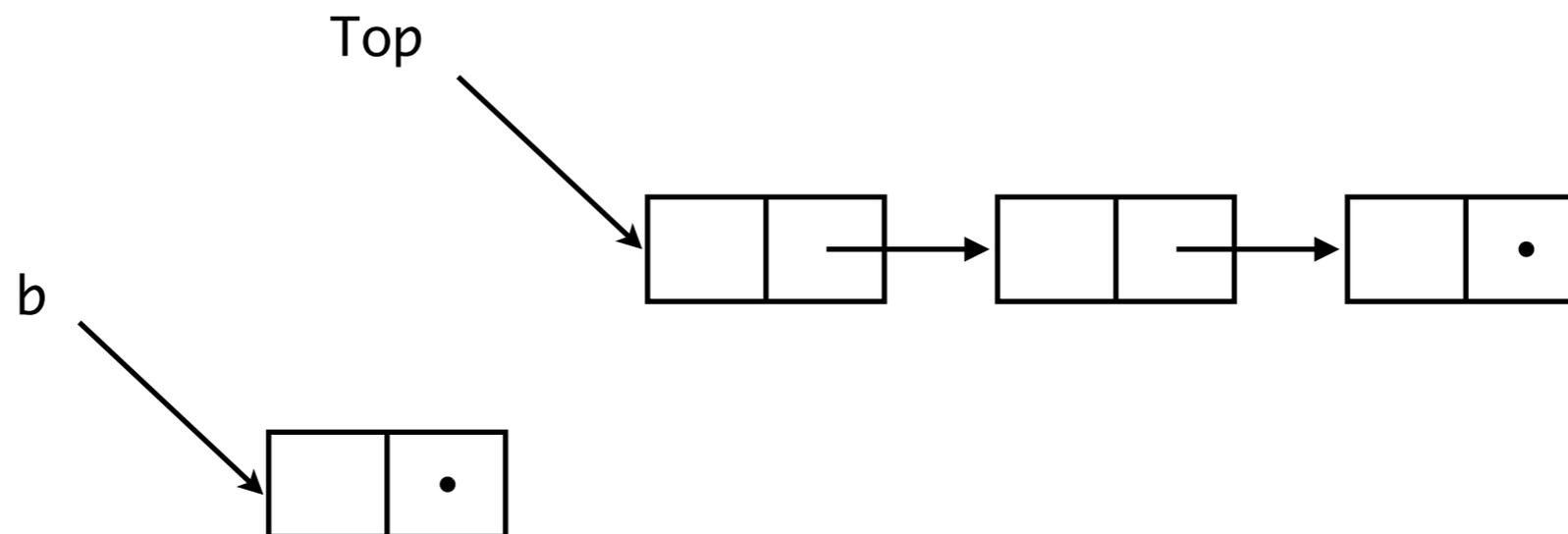
```
push (b) {  
    b->tl = Top;  
    Top = b;  
    return true;  
}
```



A sequential stack: push (b)

```
pop ( ) {  
  t = Top;  
  if (t != nil)  
    Top = t->tl;  
  return t;  
}
```

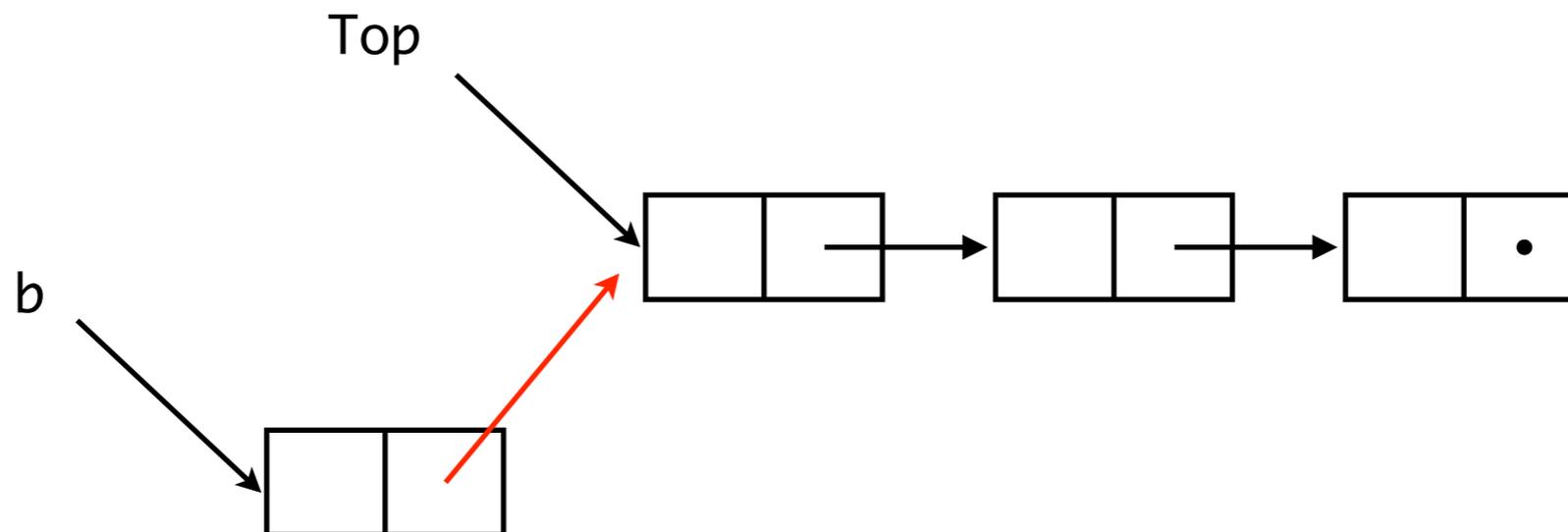
```
push (b) {  
  b->tl = Top;  
  Top = b;  
  return true;  
}
```



A sequential stack: push (b)

```
pop ( ) {  
  t = Top;  
  if (t != nil)  
    Top = t->tl;  
  return t;  
}
```

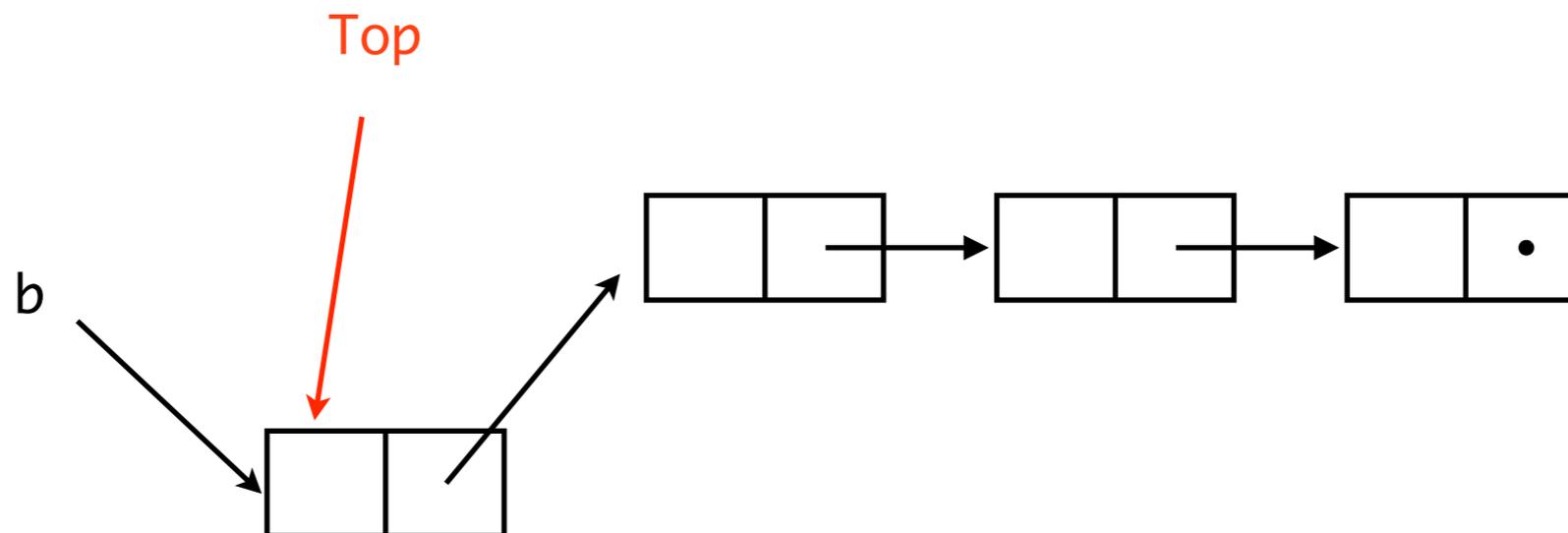
```
push (b) {  
  b->tl = Top;  
  Top = b;  
  return true;  
}
```



A sequential stack: push (b)

```
pop ( ) {  
  t = Top;  
  if (t != nil)  
    Top = t->tl;  
  return t;  
}
```

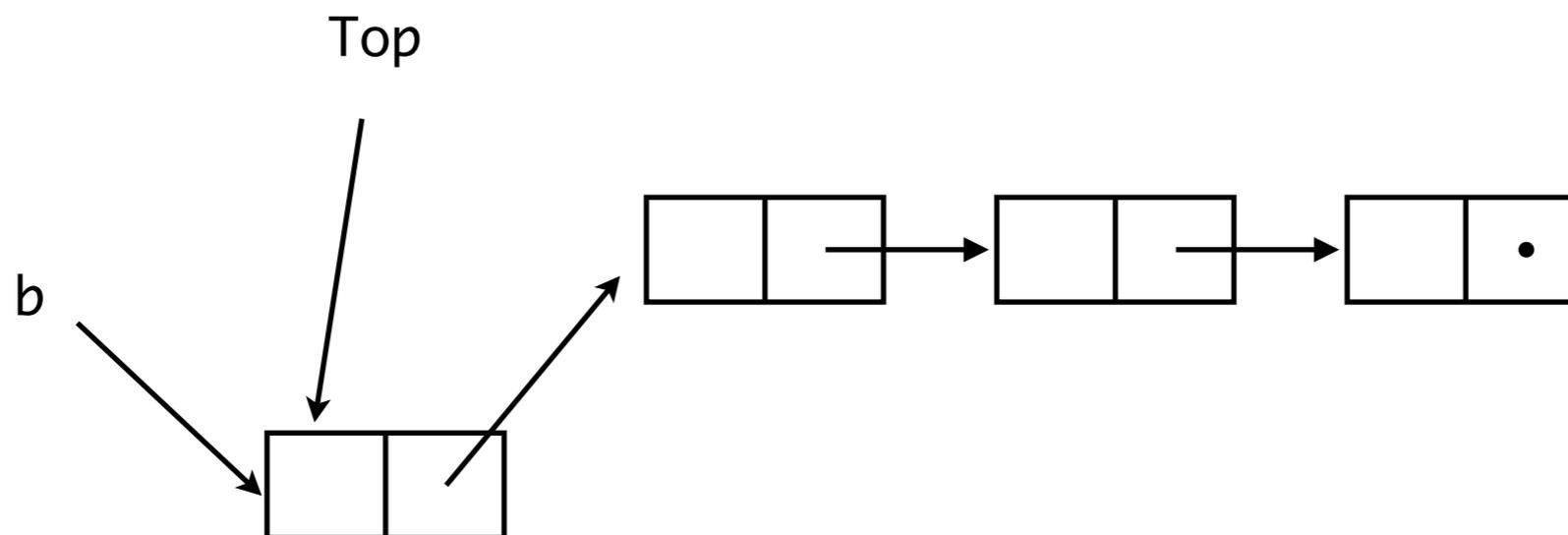
```
push (b) {  
  b->tl = Top;  
  Top = b;  
  return true;  
}
```



A sequential stack: push (b)

```
pop ( ) {  
  t = Top;  
  if (t != nil)  
    Top = t->tl;  
  return t;  
}
```

```
push (b) {  
  b->tl = Top;  
  Top = b;  
  return true;  
}
```

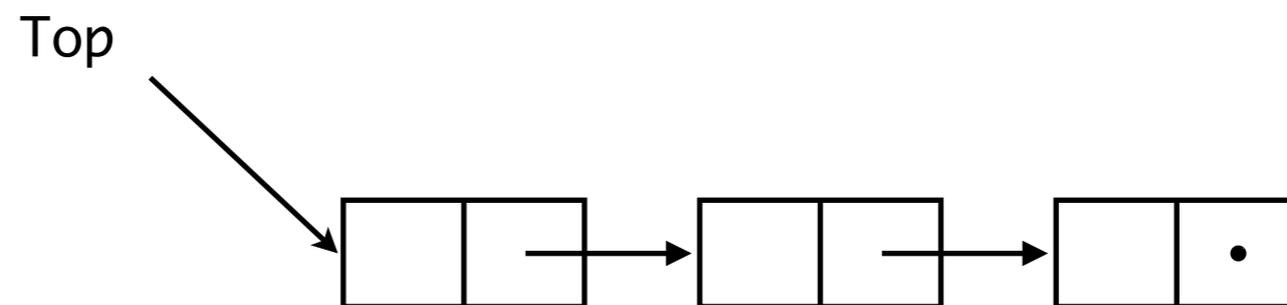


A sequential stack in a concurrent world

```
pop ( ) {  
  t = Top;  
  if (t != nil)  
    Top = t->tl;  
  return t;  
}
```

```
push (b) {  
  b->tl = Top;  
  Top = b;  
  return true;  
}
```

Imagine that two threads invoke `pop()` concurrently...

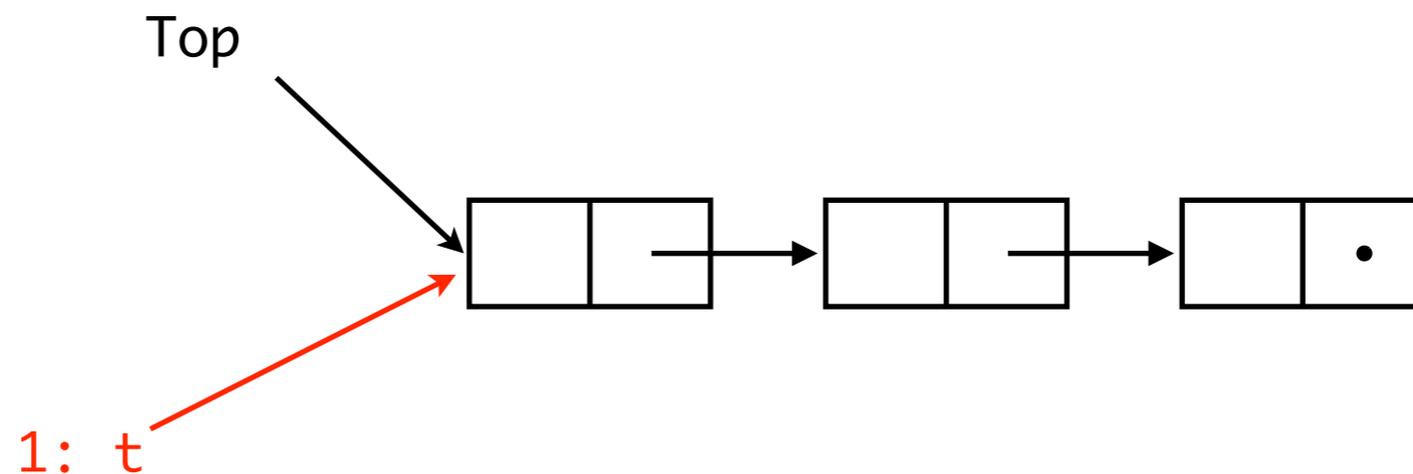


A sequential stack in a concurrent world

```
pop ( ) {  
  t = Top;  
  if (t != nil)  
    Top = t->tl;  
  return t;  
}
```

```
push (b) {  
  b->tl = Top;  
  Top = b;  
  return true;  
}
```

Imagine that two threads invoke `pop()` concurrently...

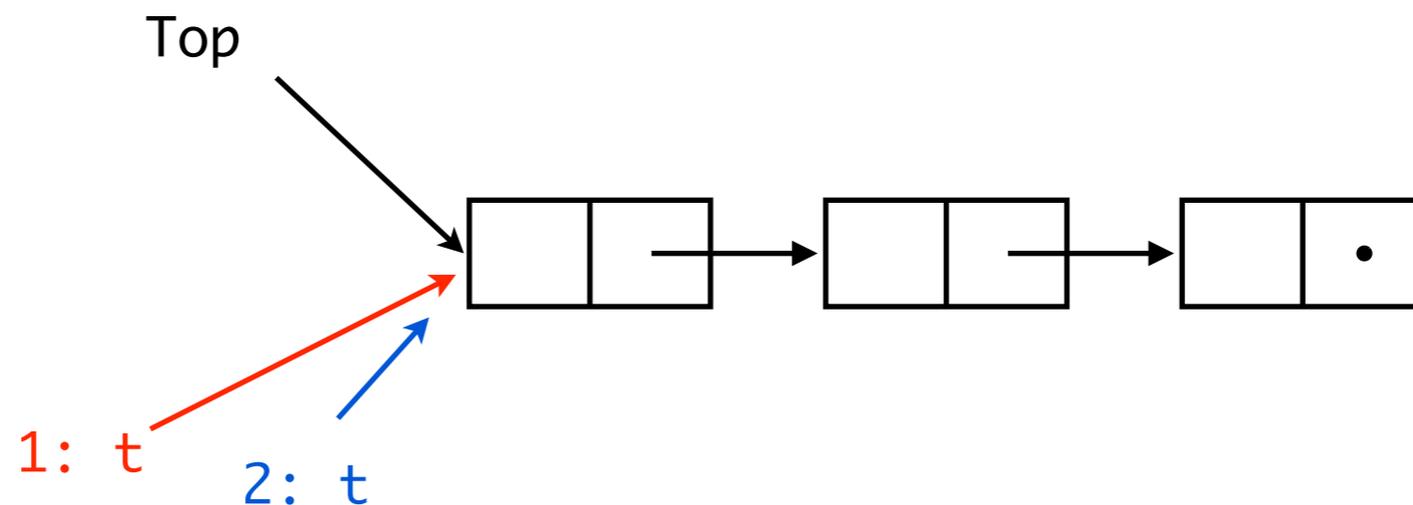


A sequential stack in a concurrent world

```
pop ( ) {  
  t = Top;  
  if (t != nil)  
    Top = t->tl;  
  return t;  
}
```

```
push (b) {  
  b->tl = Top;  
  Top = b;  
  return true;  
}
```

Imagine that two threads invoke pop() concurrently...

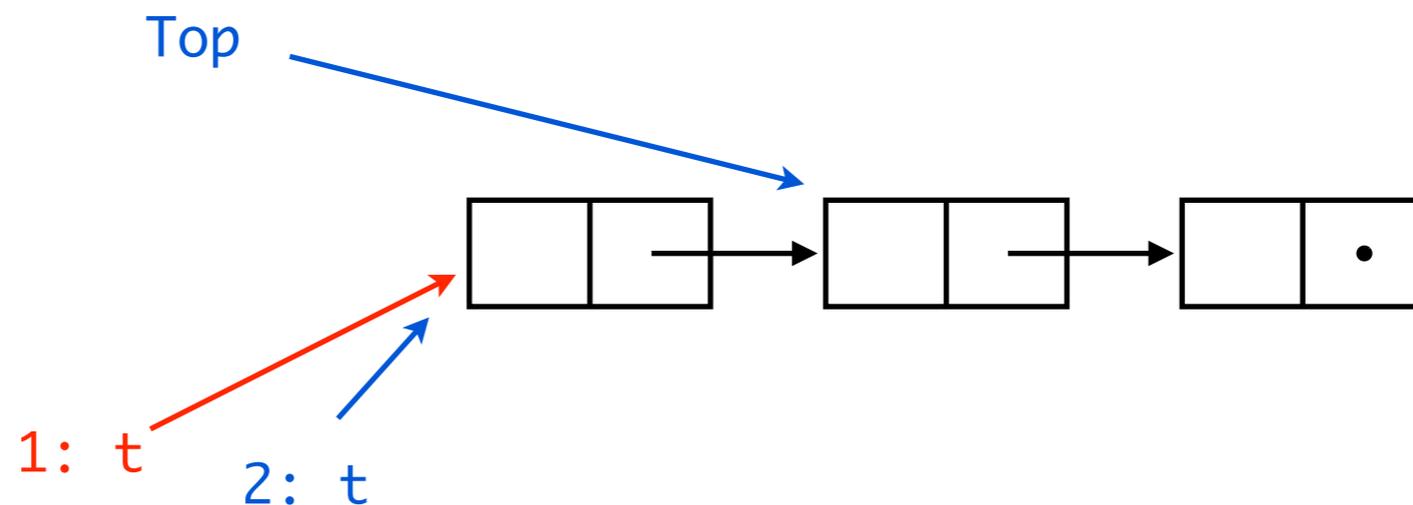


A sequential stack in a concurrent world

```
pop ( ) {  
  t = Top;  
  if (t != nil)  
    Top = t->tl;  
  return t;  
}
```

```
push (b) {  
  b->tl = Top;  
  Top = b;  
  return true;  
}
```

Imagine that two threads invoke `pop()` concurrently...



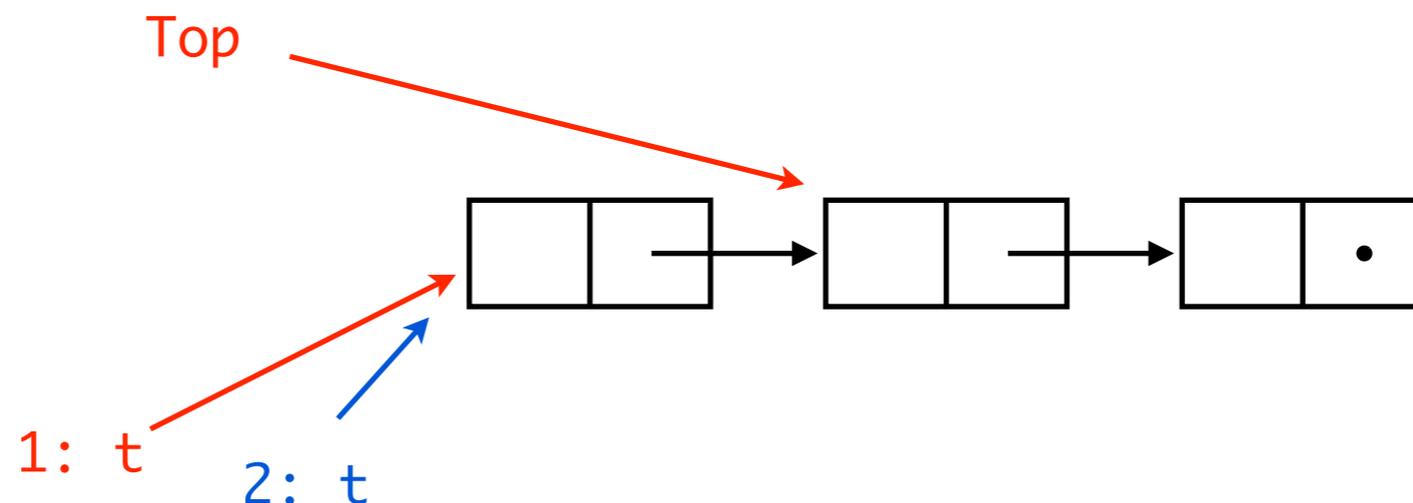
A sequential stack in a concurrent world

```
pop ( ) {  
  t = Top;  
  if (t != nil)  
    Top = t->tl;  
  return t;  
}
```

```
push (b) {  
  b->tl = Top;  
  Top = b;  
  return true;  
}
```

Imagine that two threads invoke `pop()` concurrently...

...the two threads might pop the same entry!



Idea 1: validate the Top pointer using CAS

```
pop ( ) {
  while (true) {
    t = Top;
    if (t == nil) break;
    n = t->tl;
    if CAS(&Top,t,n) break;
  }
  return t;
}
```

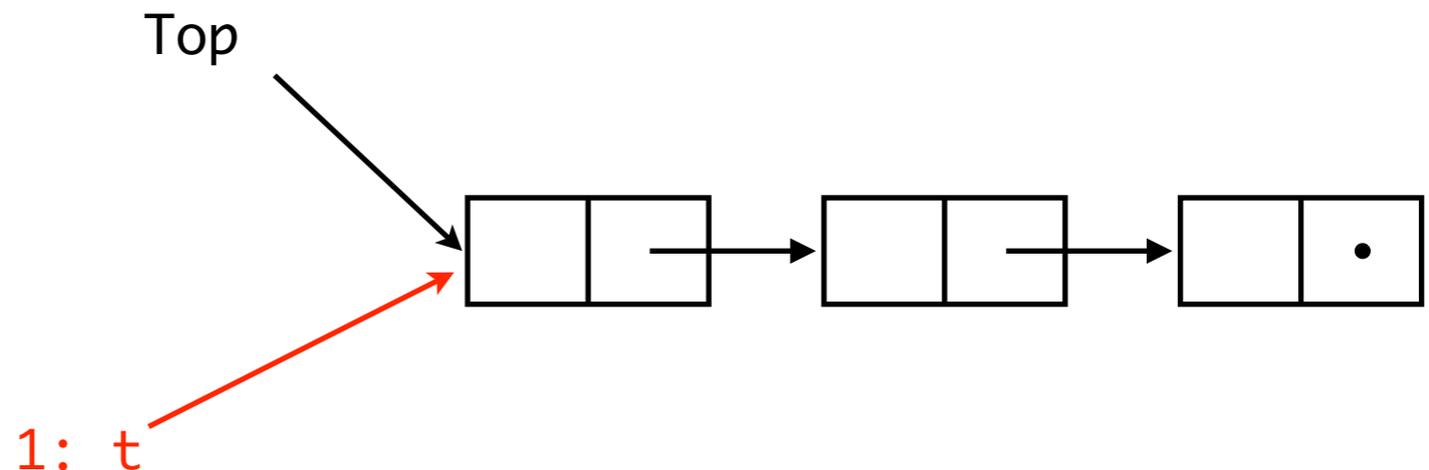
```
push (b) {
  while (true) {
    t = Top;
    b->tl = t;
    if CAS(&Top,t,b) break;
  }
  return true;
}
```

Idea 1: validate the Top pointer using CAS

```
pop ( ) {  
  while (true) {  
    t = Top;  
    if (t == nil) break;  
    n = t->tl;  
    if CAS(&Top,t,n) break;  
  }  
  return t;  
}
```

```
push (b) {  
  while (true) {  
    t = Top;  
    b->tl = t;  
    if CAS(&Top,t,b) break;  
  }  
  return true;  
}
```

Two concurrent pop() now work fine...

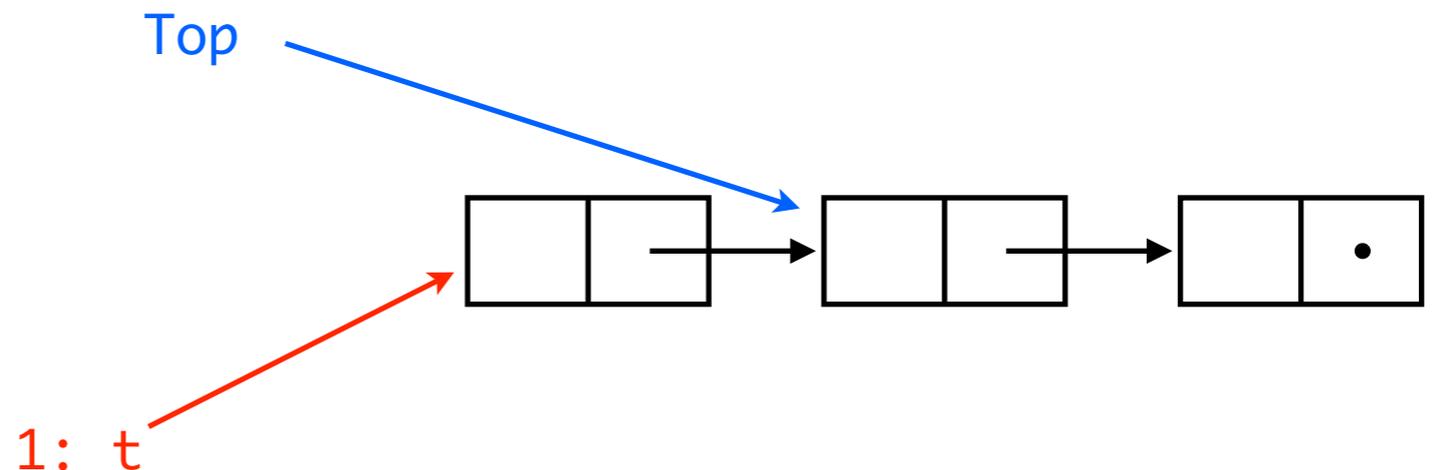


Idea 1: validate the Top pointer using CAS

```
pop ( ) {  
  while (true) {  
    t = Top;  
    if (t == nil) break;  
    n = t->tl;  
    if CAS(&Top,t,n) break;  
  }  
  return t;  
}
```

```
push (b) {  
  while (true) {  
    t = Top;  
    b->tl = t;  
    if CAS(&Top,t,b) break;  
  }  
  return true;  
}
```

Two concurrent pop() now work fine...



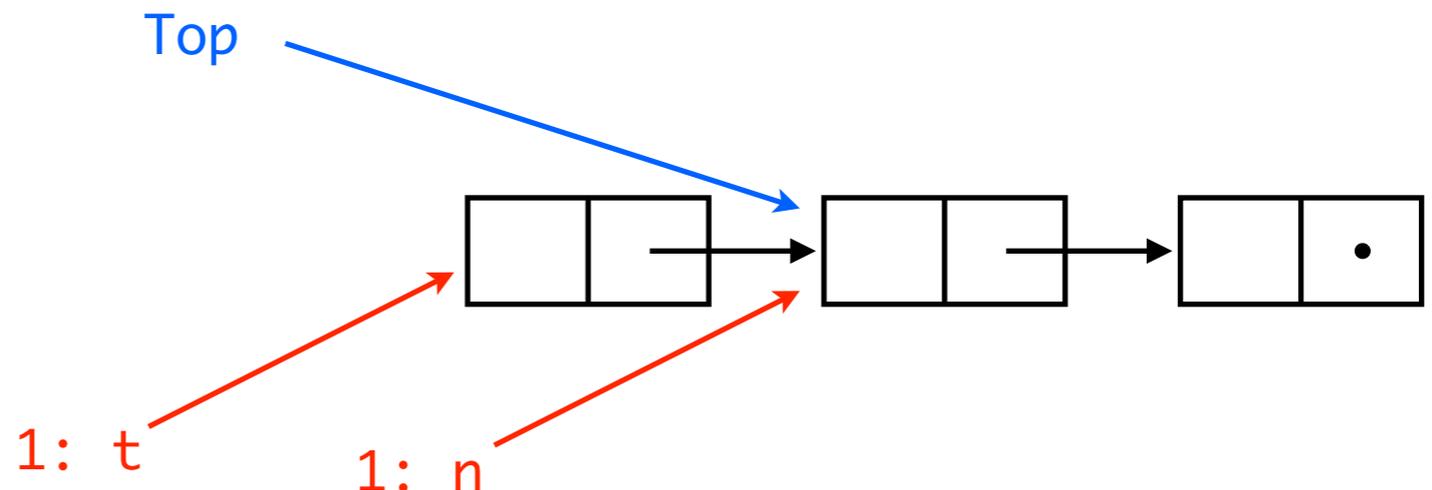
Idea 1: validate the Top pointer using CAS

```
pop ( ) {  
  while (true) {  
    t = Top;  
    if (t == nil) break;  
    n = t->t1;  
    if CAS(&Top,t,n) break;  
  }  
  return t;  
}
```

```
push (b) {  
  while (true) {  
    t = Top;  
    b->t1 = t;  
    if CAS(&Top,t,b) break;  
  }  
  return true;  
}
```

Two concurrent pop() now work fine...

The CAS of Th. 1 fails.

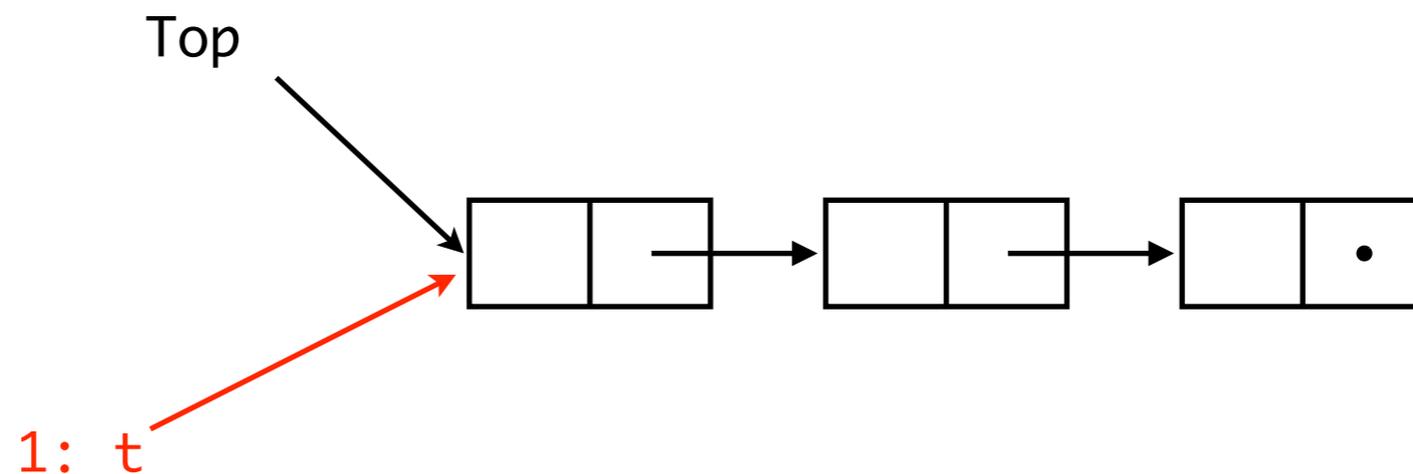


The ABA problem

```
pop ( ) {  
  while (true) {  
    t = Top;  
    if (t == nil) break;  
    n = t->tl;  
    if CAS(&Top,t,n) break;  
  }  
  return t;  
}
```

```
push (b) {  
  while (true) {  
    t = Top;  
    b->tl = t;  
    if CAS(&Top,t,b) break;  
  }  
  return true;  
}
```

Th 1 starts popping...

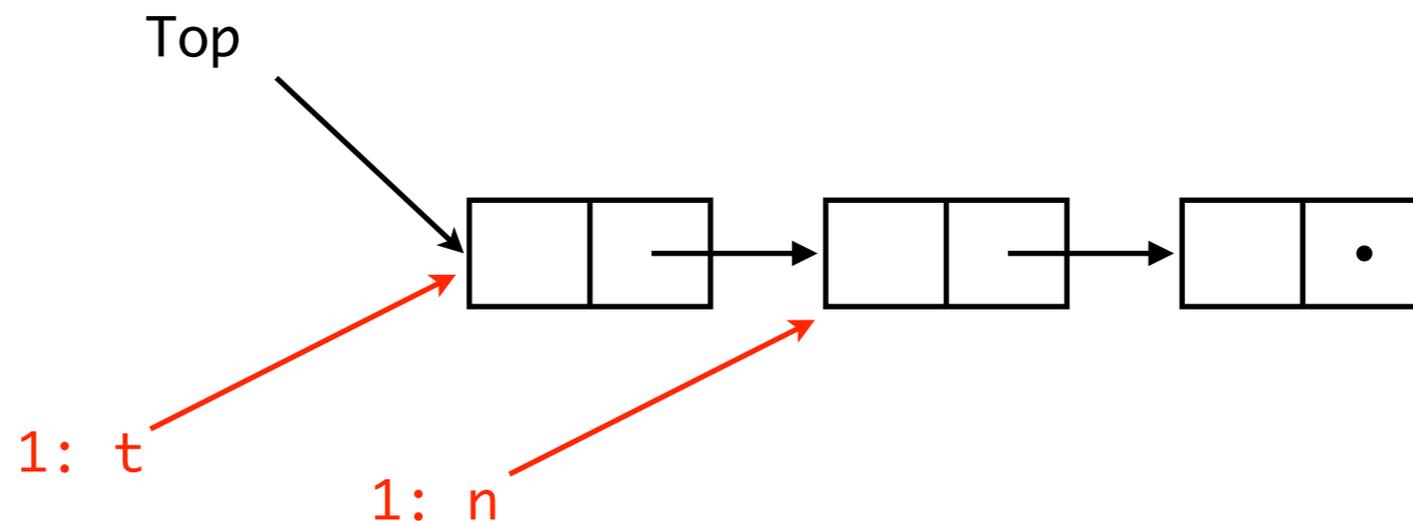


The ABA problem

```
pop ( ) {  
  while (true) {  
    t = Top;  
    if (t == nil) break;  
    n = t->tl;  
    if CAS(&Top,t,n) break;  
  }  
  return t;  
}
```

```
push (b) {  
  while (true) {  
    t = Top;  
    b->tl = t;  
    if CAS(&Top,t,b) break;  
  }  
  return true;  
}
```

Th 1 starts popping...

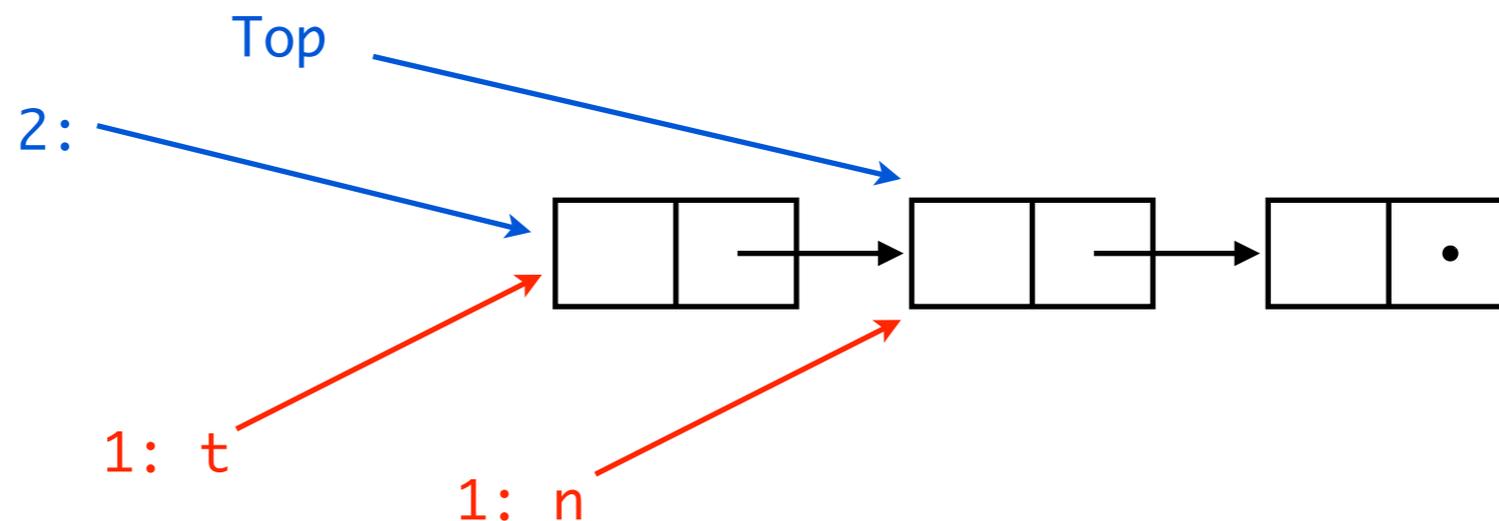


The ABA problem

```
pop ( ) {  
  while (true) {  
    t = Top;  
    if (t == nil) break;  
    n = t->tl;  
    if CAS(&Top,t,n) break;  
  }  
  return t;  
}
```

```
push (b) {  
  while (true) {  
    t = Top;  
    b->tl = t;  
    if CAS(&Top,t,b) break;  
  }  
  return true;  
}
```

Th 2 pops...

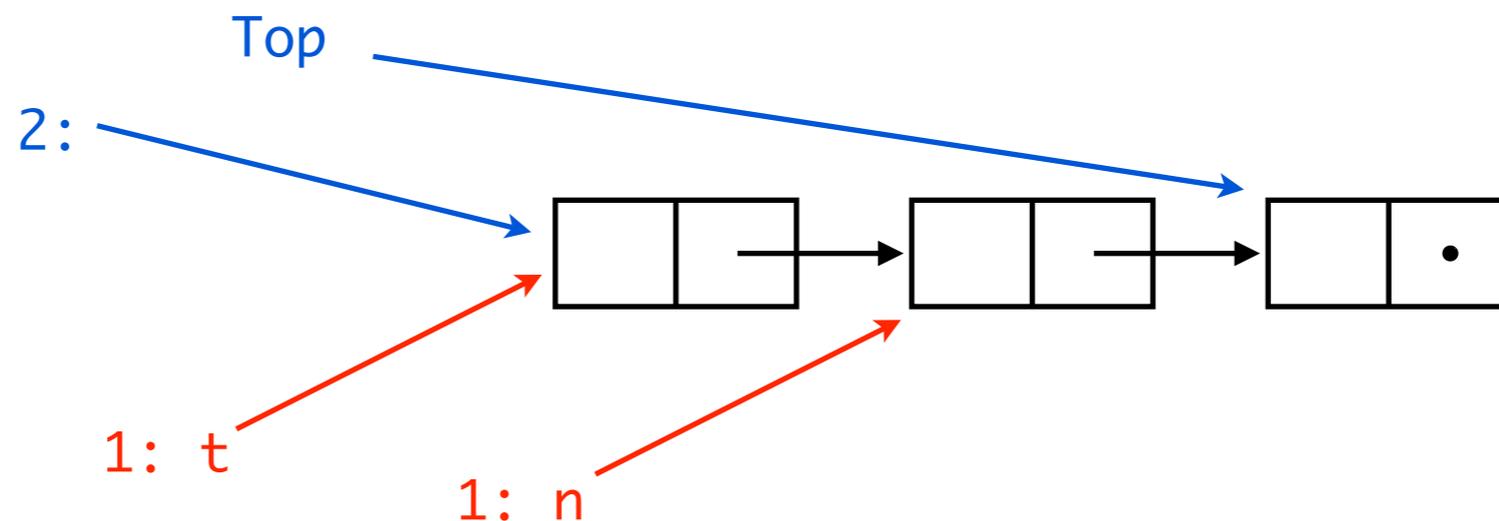


The ABA problem

```
pop ( ) {  
  while (true) {  
    t = Top;  
    if (t == nil) break;  
    n = t->tl;  
    if CAS(&Top,t,n) break;  
  }  
  return t;  
}
```

```
push (b) {  
  while (true) {  
    t = Top;  
    b->tl = t;  
    if CAS(&Top,t,b) break;  
  }  
  return true;  
}
```

Th 2 pops again...

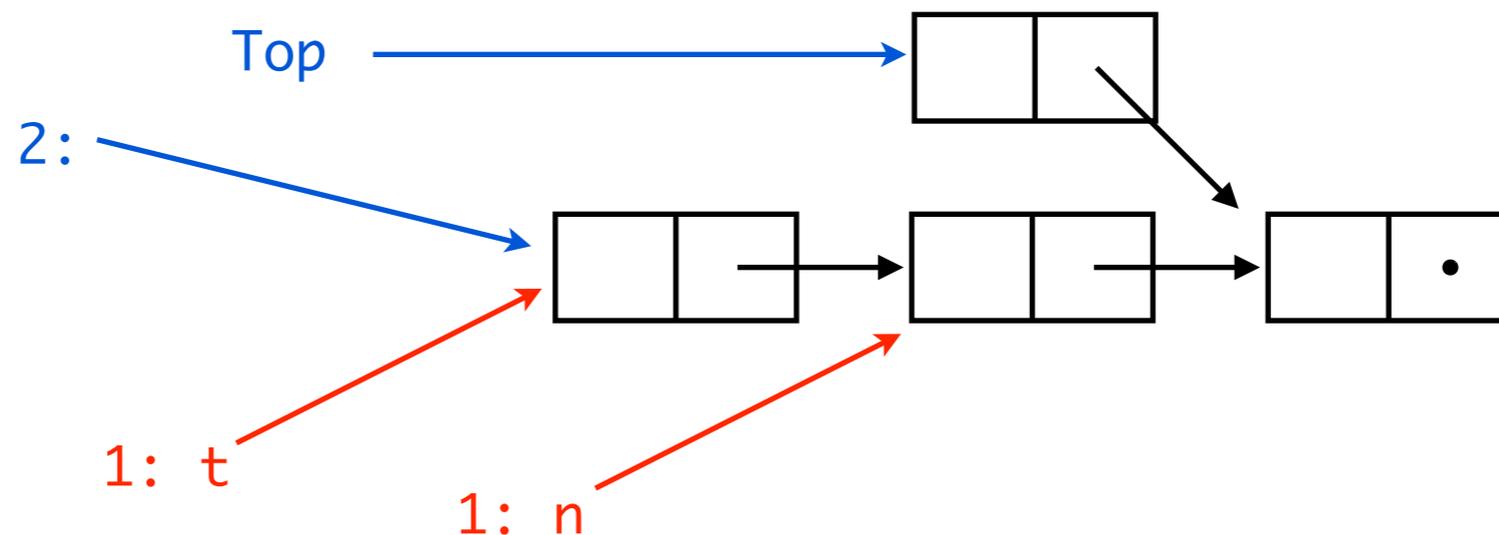


The ABA problem

```
pop ( ) {  
  while (true) {  
    t = Top;  
    if (t == nil) break;  
    n = t->tl;  
    if CAS(&Top,t,n) break;  
  }  
  return t;  
}
```

```
push (b) {  
  while (true) {  
    t = Top;  
    b->tl = t;  
    if CAS(&Top,t,b) break;  
  }  
  return true;  
}
```

Th 2 pushes a new node...

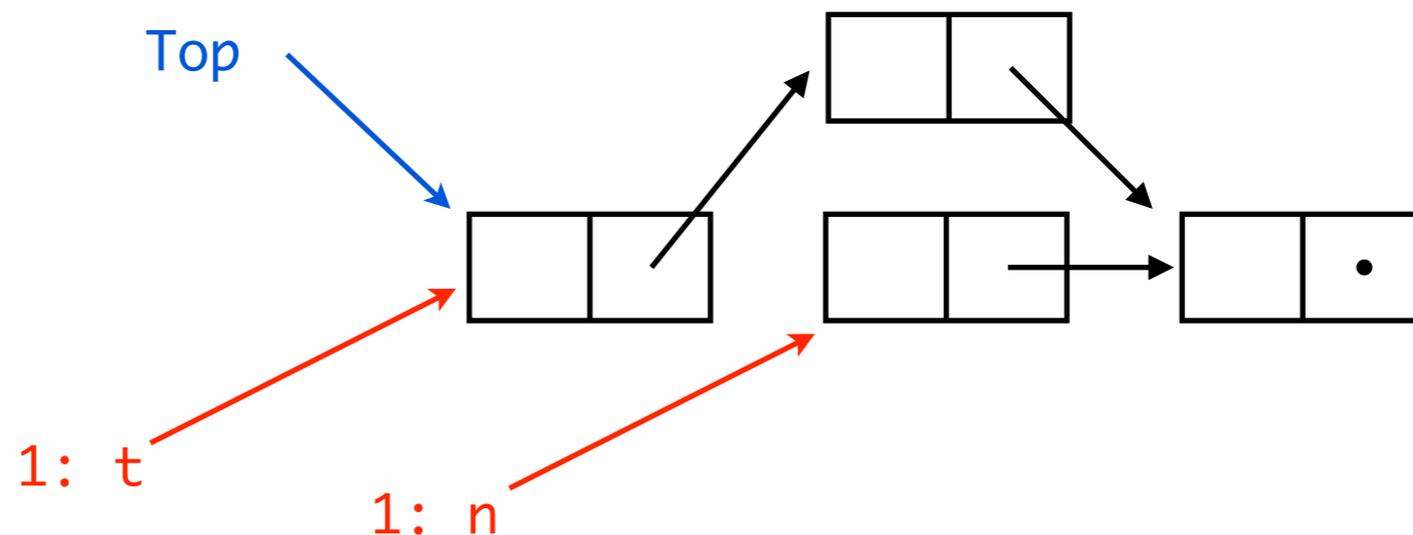


The ABA problem

```
pop ( ) {  
  while (true) {  
    t = Top;  
    if (t == nil) break;  
    n = t->tl;  
    if CAS(&Top,t,n) break;  
  }  
  return t;  
}
```

```
push (b) {  
  while (true) {  
    t = Top;  
    b->tl = t;  
    if CAS(&Top,t,b) break;  
  }  
  return true;  
}
```

Th 2 pushes the old head of the stack...

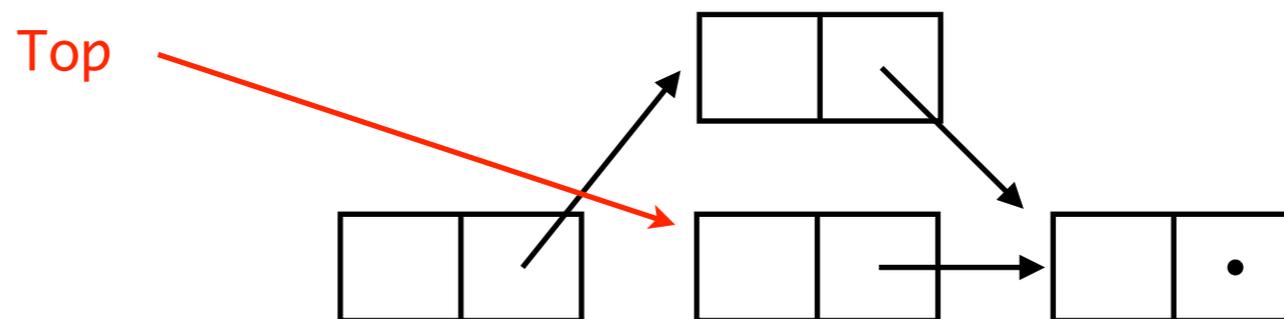


The ABA problem

```
pop ( ) {  
  while (true) {  
    t = Top;  
    if (t == nil) break;  
    n = t->tl;  
    if CAS(&Top,t,n) break;  
  }  
  return t;  
}
```

```
push (b) {  
  while (true) {  
    t = Top;  
    b->tl = t;  
    if CAS(&Top,t,b) break;  
  }  
  return true;  
}
```

Th 1 corrupts the stack...



The hazard pointers methodology

Michael adds to the previous algorithm a *global array H of hazard pointers*:

- thread i alone is allowed to write to element $H[i]$ of the array;
- any thread can read any entry of H .

The algorithm is then modified:

- before popping a cell, a thread puts its address into its own element of H . This entry is cleared only if CAS succeeds or the stack is empty;
- before pushing a cell, a thread checks to see whether it is pointed to from any element of H . If it is, push is delayed.

Michael's algorithm, simplified

```
pop ( ) {  
    while (true) {  
        atomic { t = Top;  
                H[tid] = t; };  
        if (t == nil) break;  
        n = t->tl;  
        if CAS(&Top,t,n) break;  
    }  
    H[tid] = nil;  
    return t;  
}
```

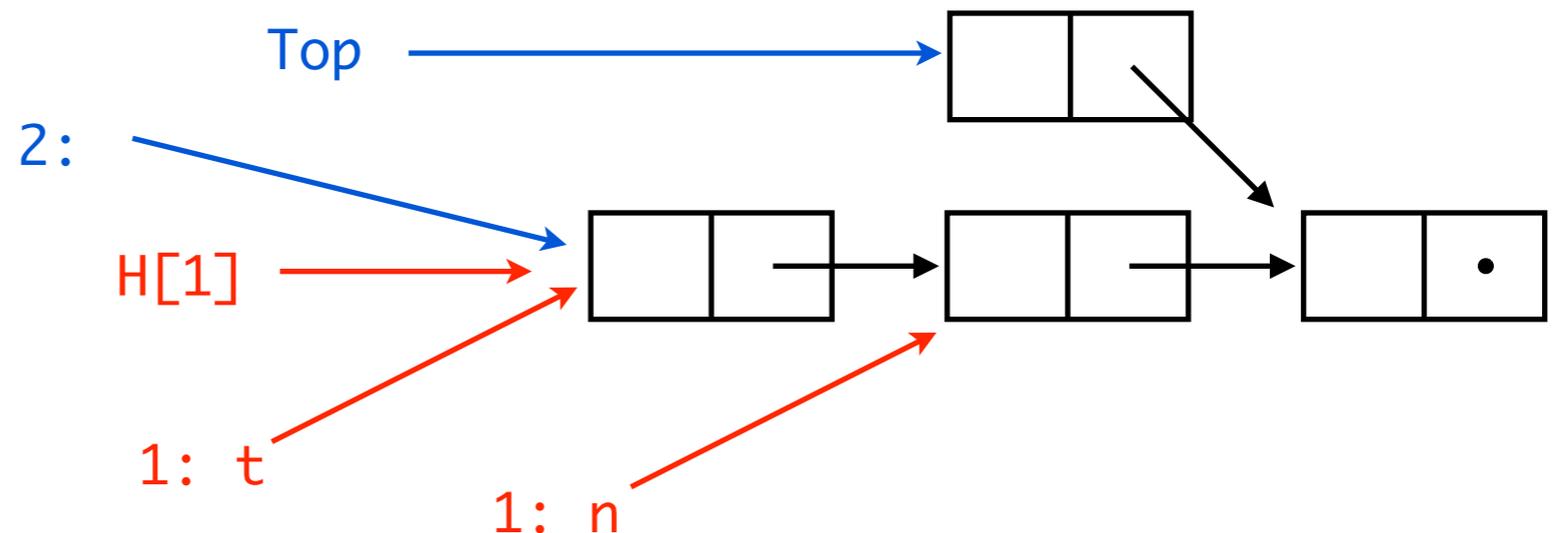
```
push (b) {  
    for (n = 0; n < no_threads, n++)  
        if (H[n] == b) return false;  
    while (true) {  
        t = Top;  
        b->tl = t;  
        if CAS(&Top,t,b) break;  
    }  
    return true;  
}
```

Michael's algorithm, simplified

```
pop ( ) {  
  while (true) {  
    atomic { t = Top;  
            H[tid] = t; };  
    if (t == nil) break;  
    n = t->tl;  
    if CAS(&Top,t,n) break;  
  }  
  H[tid] = nil;  
  return t;  
}
```

Th 2 cannot push the old head, because Th 1 has an hazard pointer on it...

```
push (b) {  
  for (n = 0; n < no_threads, n++)  
    if (H[n] == b) return false;  
  while (true) {  
    t = Top;  
    b->tl = t;  
    if CAS(&Top,t,b) break;  
  }  
  return true;  
}
```



Key properties of Michael's simplified algorithm

- A node can be added to the hazard array only if it is reachable through the stack;
- a node that has been popped is not reachable through the stack;
- a node that is unreachable in the stack and that is in the hazard array cannot be added to the stack;
- while a node is reachable and in the hazard array, it has a constant tail.

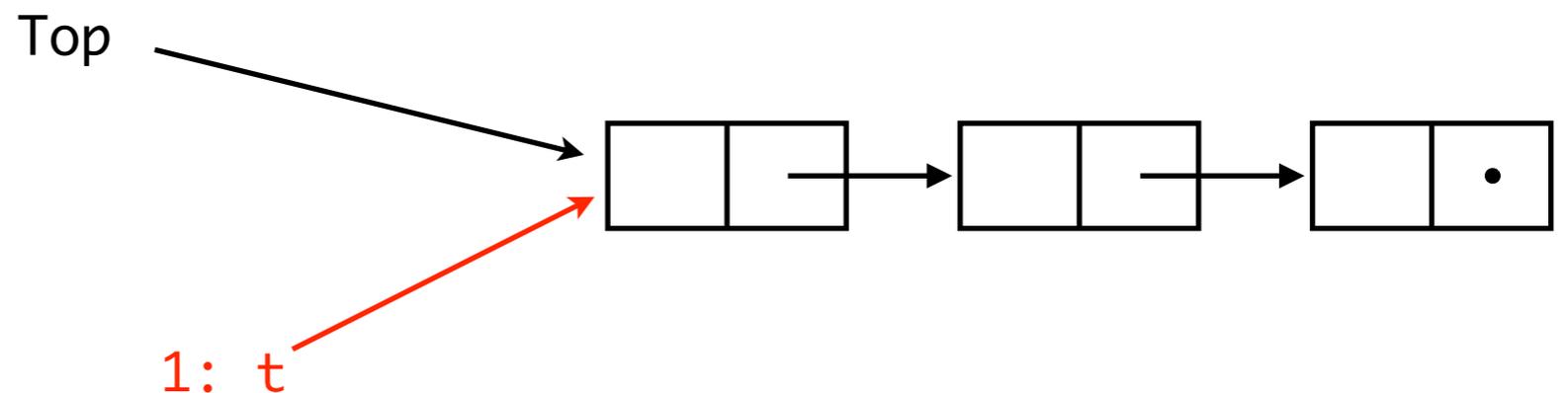
These are a good example of the properties we might want to state and prove about a concurrent algorithm.

The role of *atomic*

```
pop ( ) {  
  while (true) {  
    t = Top;  
    H[tid] = t;  
    if (t == nil) break;  
    n = t->tl;  
    if CAS(&Top,t,n) break;  
  }  
  H[tid] = nil;  
  return t;  
}
```

```
push (b) {  
  for (n = 0; n < no_threads, n++)  
    if (H[n] == b) return false;  
  while (true) {  
    t = Top;  
    b->tl = t;  
    if CAS(&Top,t,b) break;  
  }  
  return true;  
}
```

Th 1 copies Top...

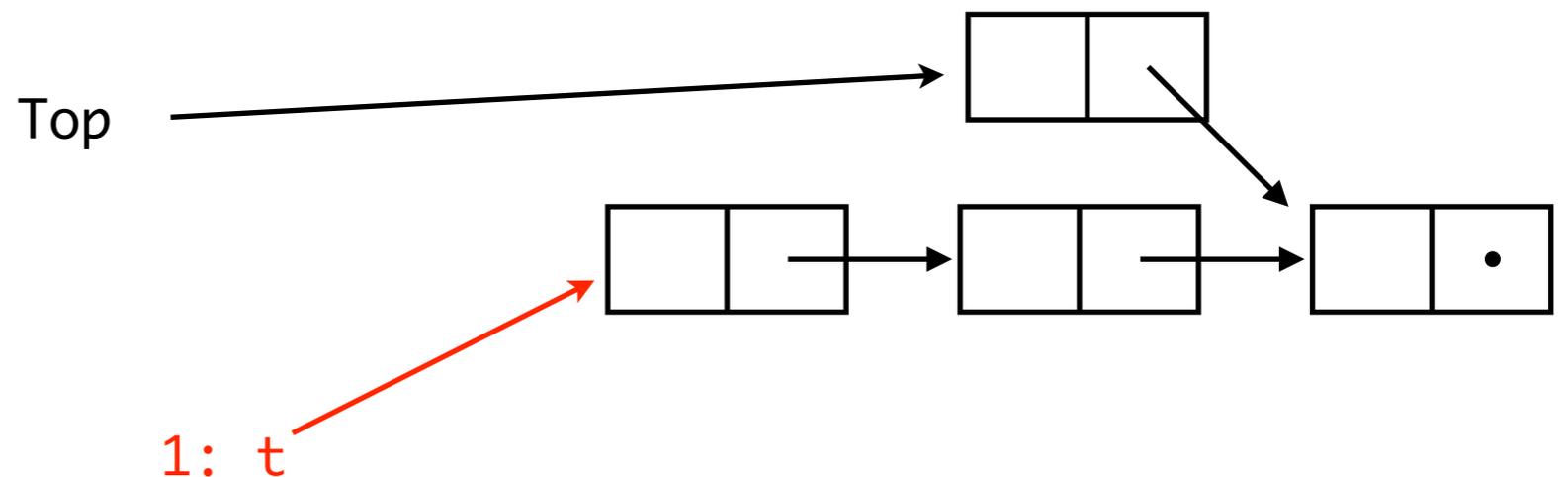


The role of *atomic*

```
pop ( ) {  
  while (true) {  
    t = Top;  
    H[tid] = t;  
    if (t == nil) break;  
    n = t->tl;  
    if CAS(&Top,t,n) break;  
  }  
  H[tid] = nil;  
  return t;  
}
```

Th 2 pops twice, and
pushes a new node...

```
push (b) {  
  for (n = 0; n < no_threads, n++)  
    if (H[n] == b) return false;  
  while (true) {  
    t = Top;  
    b->tl = t;  
    if CAS(&Top,t,b) break;  
  }  
  return true;  
}
```

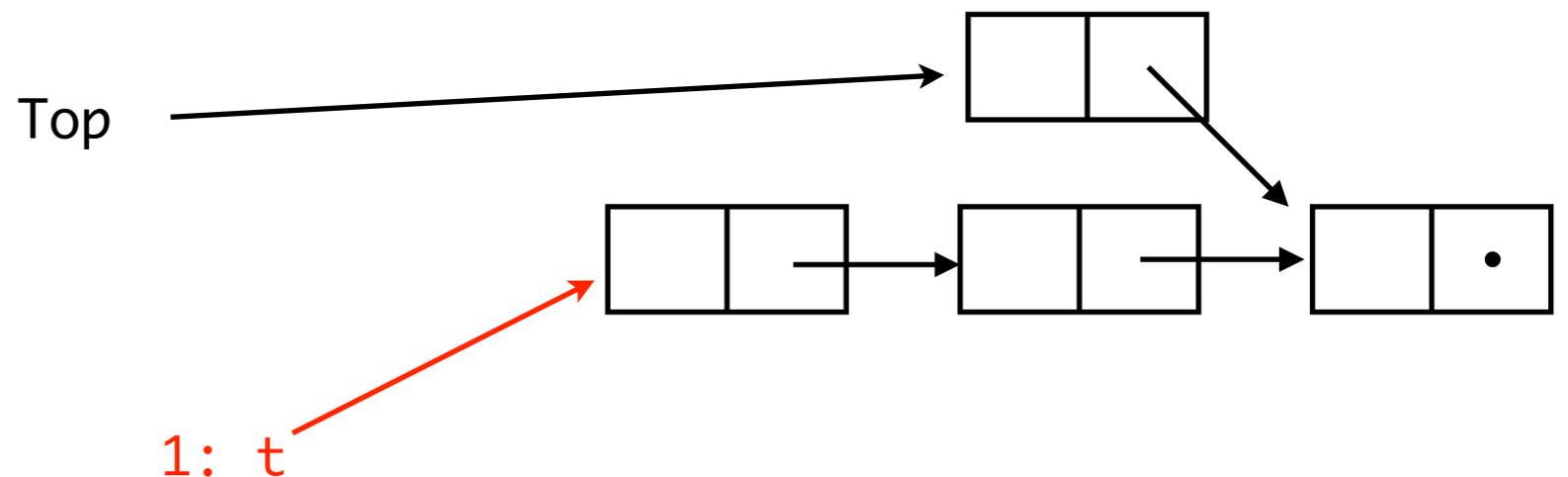


The role of *atomic*

```
pop ( ) {  
  while (true) {  
    t = Top;  
    H[tid] = t;  
    if (t == nil) break;  
    n = t->tl;  
    if CAS(&Top,t,n) break;  
  }  
  H[tid] = nil;  
  return t;  
}
```

Th 2 starts pushing the old head, and is halfway in the for loop...

```
push (b) {  
  for (n = 0; n < no_threads, n++)  
    if (H[n] == b) return false;  
  while (true) {  
    t = Top;  
    b->tl = t;  
    if CAS(&Top,t,b) break;  
  }  
  return true;  
}
```

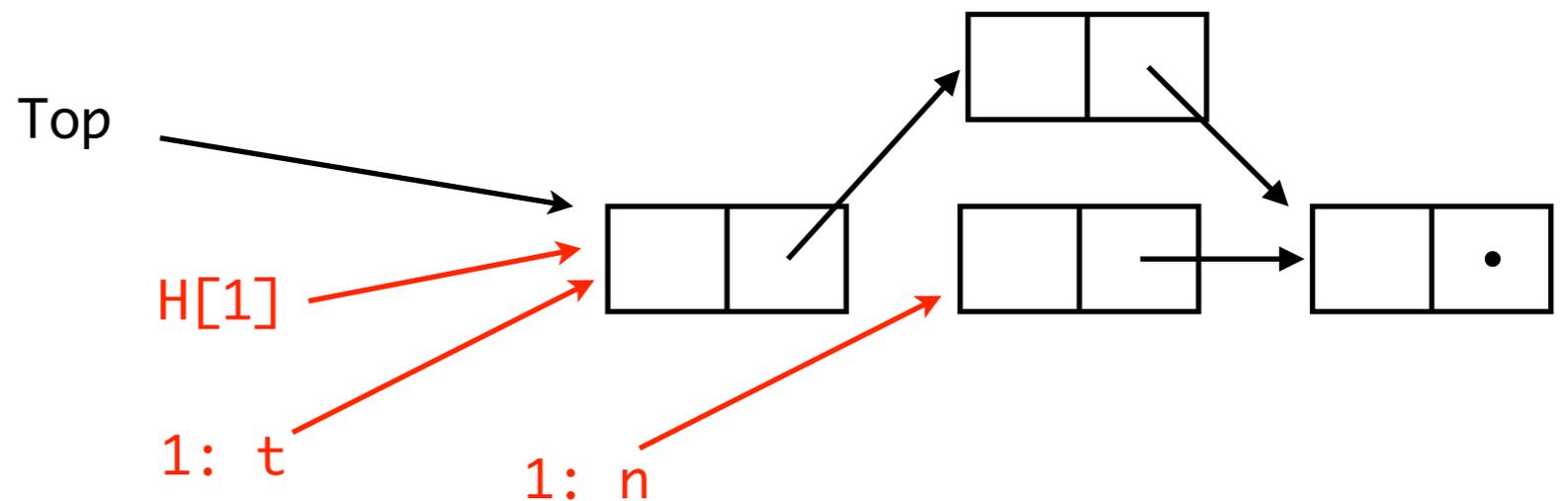


The role of *atomic*

```
pop ( ) {  
  while (true) {  
    t = Top;  
    H[tid] = t;  
    if (t == nil) break;  
    n = t->tl;  
    if CAS(&Top,t,n) break;  
  }  
  H[tid] = nil;  
  return t;  
}
```

Th 1 sets its hazard pointer... but Th 2 might not see the hazard pointer of Th 1!

```
push (b) {  
  for (n = 0; n < no_threads, n++)  
    if (H[n] == b) return false;  
  while (true) {  
    t = Top;  
    b->tl = t;  
    if CAS(&Top,t,b) break;  
  }  
  return true;  
}
```



Michael shared stack

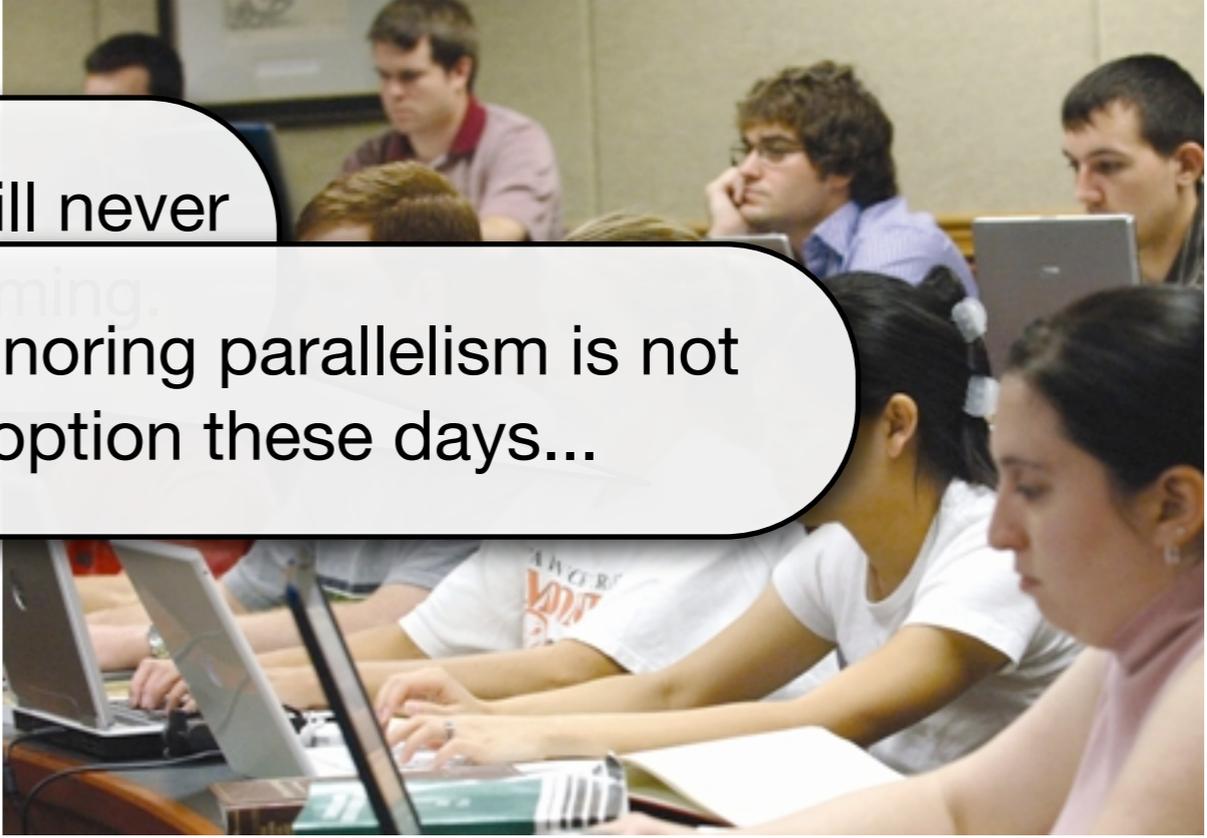
```
pop ( ) {  
    while (true) {  
        t = Top;  
        if (t == nil) break;  
        H[tid] = t;  
        if (t != Top) break;  
        n = t->t1;  
        if CAS(&Top,t,n) break;  
    }  
    H[tid] = nil;  
    return t;  
}
```

```
push (b) {  
    for (n = 0; n < no_threads, n++)  
        if (H[n] == b) return false;  
    while (true) {  
        t = Top;  
        b->t1 = t;  
        if CAS(&Top,t,b) break;  
    }  
    return true;  
}
```

Trust me: if we validate `t` against the `Top` pointer before reading `t->t1`, we get a correct algorithm.



Reaction 1.



That algorithm is insane... I will never use it in my everyday programming.

...and ignoring parallelism is not an option these days...



Yes, you will! Michael algorithms are part of `java.util.concurrent`.

Reaction 2.



The course 2.36.1 gives some hints.

How can we reason about this code?



Re

H
co

Welcome to 2.37.1

1. Relaxed-memory concurrency,
from hardware to programming languages
2. Runtime algorithms
and compilation of parallel programming languages
3. Modern concurrent algorithms

Tell us about the state of the art!





Part 1.

Shared memory: an elusive abstraction

<http://www.di.ens.fr/~zappa/projects/weakmemory>

Based on work done by or with

Peter Sewell, Jaroslav Ševčík, Susmit Sarkar, Tom Ridge, Scott Owens, Viktor Vafeiadis, Magnus O. Myreen, Kayvan Memarian, Luc Maranget, Pankaj Pawan, Thomas Braibant, Mark Batty, Jade Alglave.

The golden age, 1945 - 1972

Memory = Array of Values



Properties

- no thread local reordering

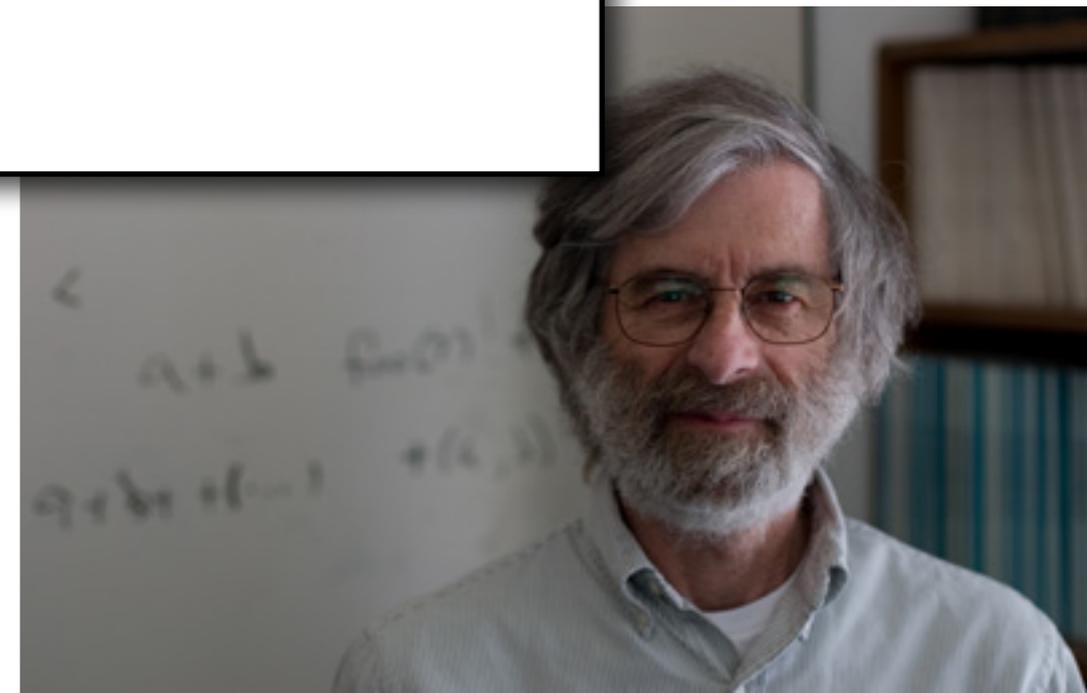
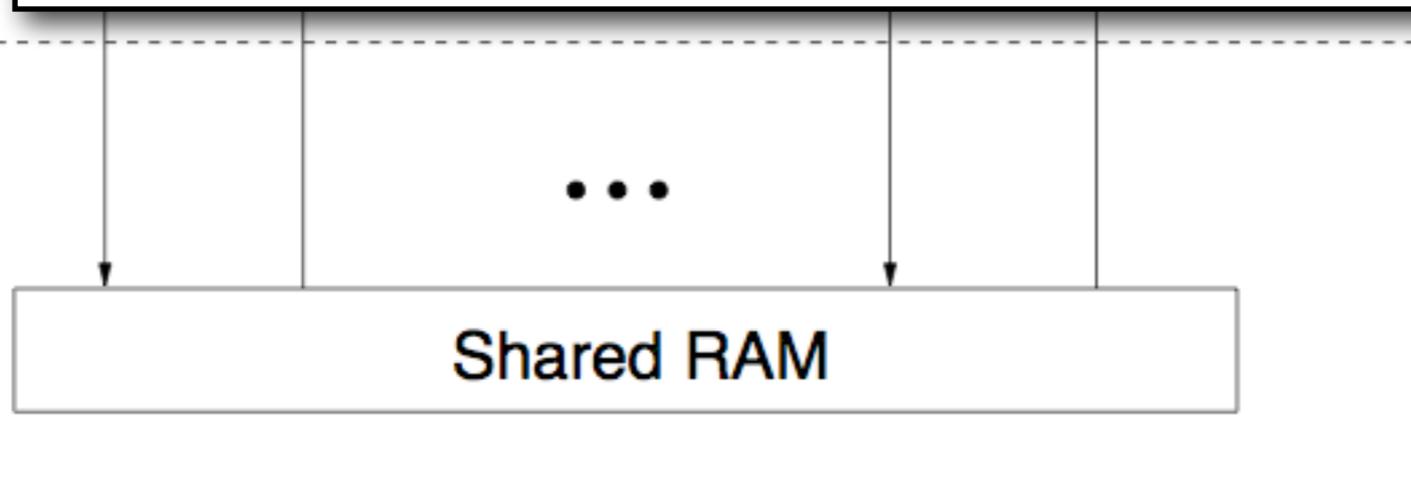
- time

Taken for granted by almost all

- concurrency theorists
- program logics
- concurrent verification tools
- programmers

ions of
der, and
is

Lam



The first shocking example

Consider the following x86 assembler code.

Initial shared memory values: $[x]=0$ $[y]=0$

Per-processor registers: **EAX** **EBX**

Thread 0	Thread 1
<code>MOV [x] ← 1</code>	<code>MOV [y] ← 1</code>
<code>MOV EAX ← [y]</code>	<code>MOV EBX ← [x]</code>

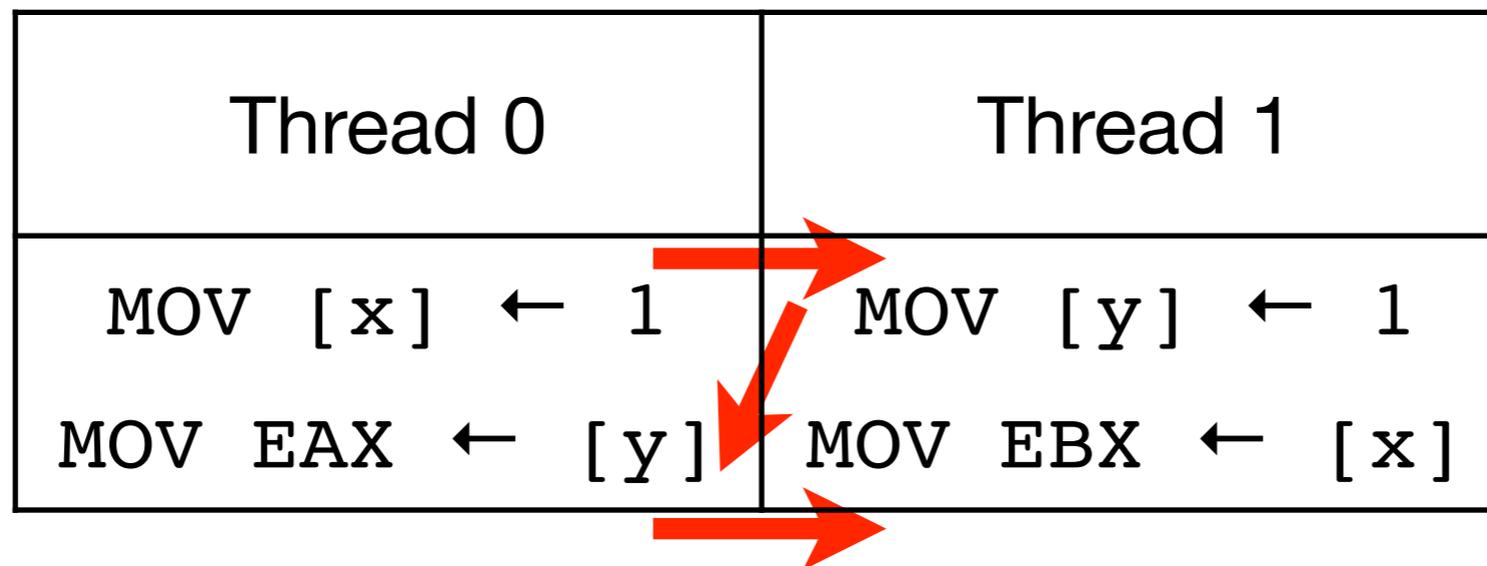
Can you guess the final register values: **EAX = ?** **EBX = ?**

The first shocking example

Consider the following x86 assembler code.

Initial shared memory values: $[x]=0$ $[y]=0$

Per-processor registers: **EAX** **EBX**



Can you guess the final register values: **EAX = 1** **EBX = 1**

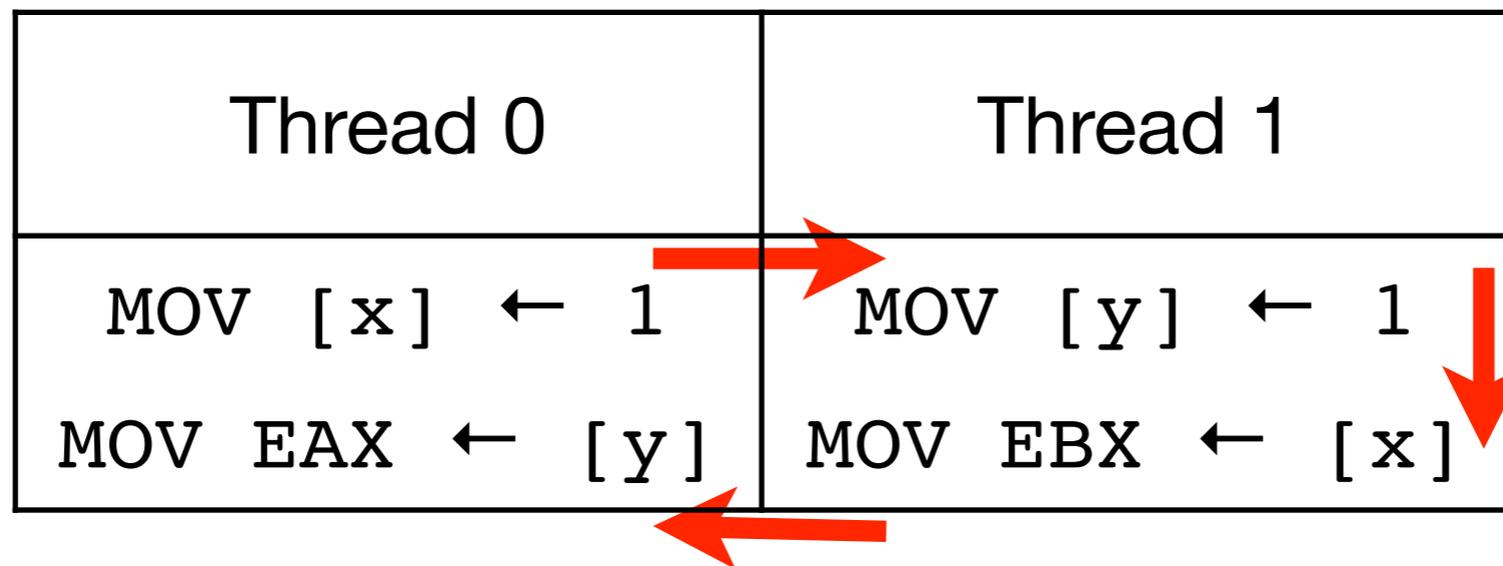
The first shocking example

Consider the following x86 assembler code:

Initial shared memory values: $[x]=0$ $[y]=0$

Per-processor registers: **EAX** **EBX**

Thread 0	Thread 1
<code>MOV [x] ← 1</code>	<code>MOV [y] ← 1</code>
<code>MOV EAX ← [y]</code>	<code>MOV EBX ← [x]</code>



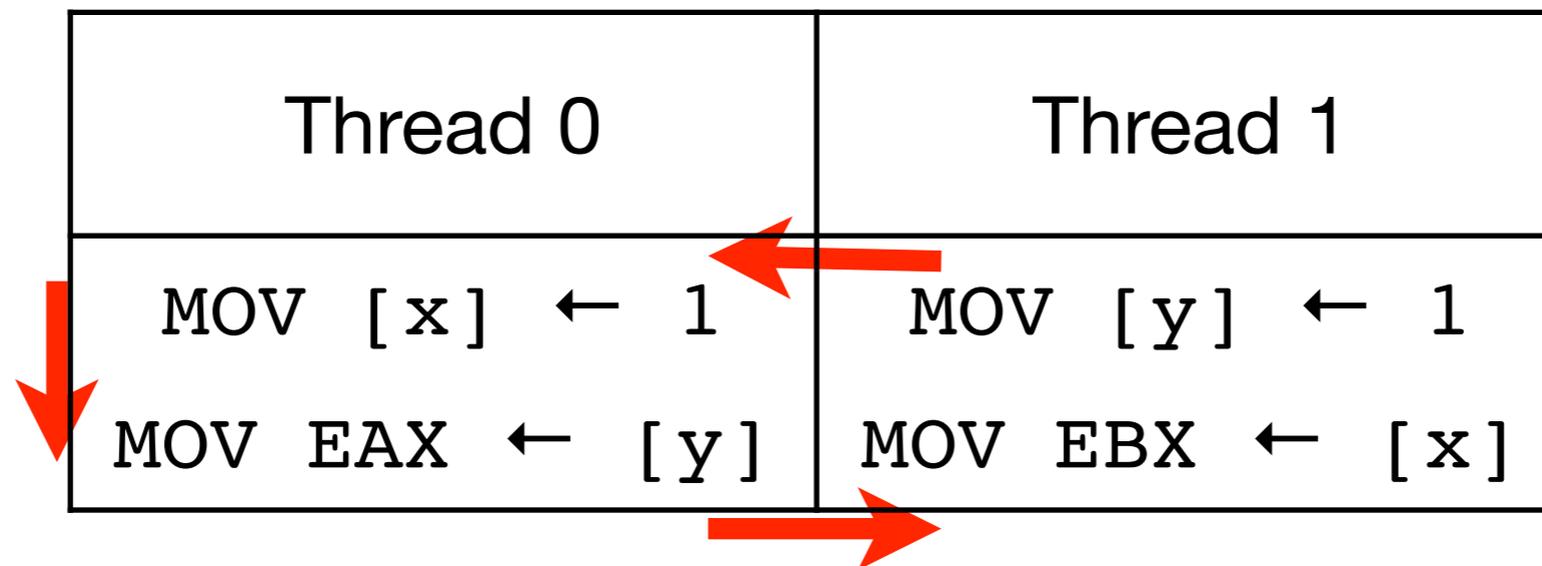
Can you guess the final register values: **EAX = 1** **EBX = 1**

The first shocking example

Consider the following x86 assembler code:

Initial shared memory values: $[x]=0$ $[y]=0$

Per-processor registers: **EAX** **EBX**



Can you guess the final register values: **EAX = 1** **EBX = 1**

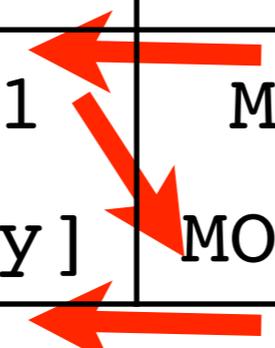
The first shocking example

Consider the following x86 assembler code:

Initial shared memory values: $[x]=0$ $[y]=0$

Per-processor registers: **EAX** **EBX**

Thread 0	Thread 1
<code>MOV [x] ← 1</code>	<code>MOV [y] ← 1</code>
<code>MOV EAX ← [y]</code>	<code>MOV EBX ← [x]</code>



Can you guess the final register values: **EAX = 1** **EBX = 1**

The first shocking example

Consider the following x86 assembler code:

Initial shared memory values: $[x]=0$ $[y]=0$

Per-processor registers: **EAX** **EBX**

Thread 0	Thread 1
 <code>MOV [x] ← 1</code>	<code>MOV [y] ← 1</code>
<code>MOV EAX ← [y]</code>	<code>MOV EBX ← [x]</code> 

Can you guess the final register values: **EAX = 0** **EBX = 1**

The first shocking example

Consider the following x86 assembler code:

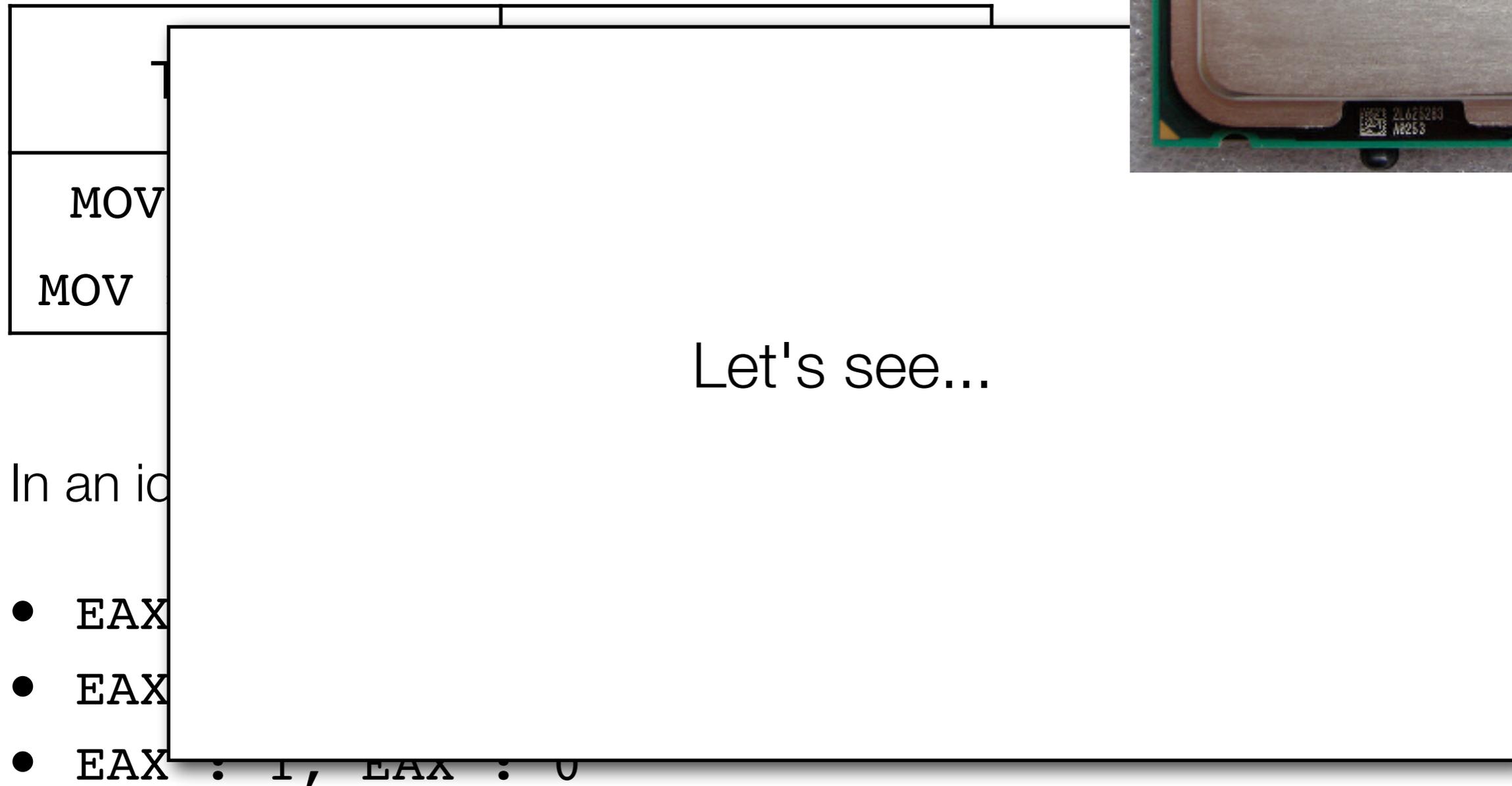
Initial shared memory values: $[x]=0$ $[y]=0$

Per-processor registers: **EAX** **EBX**

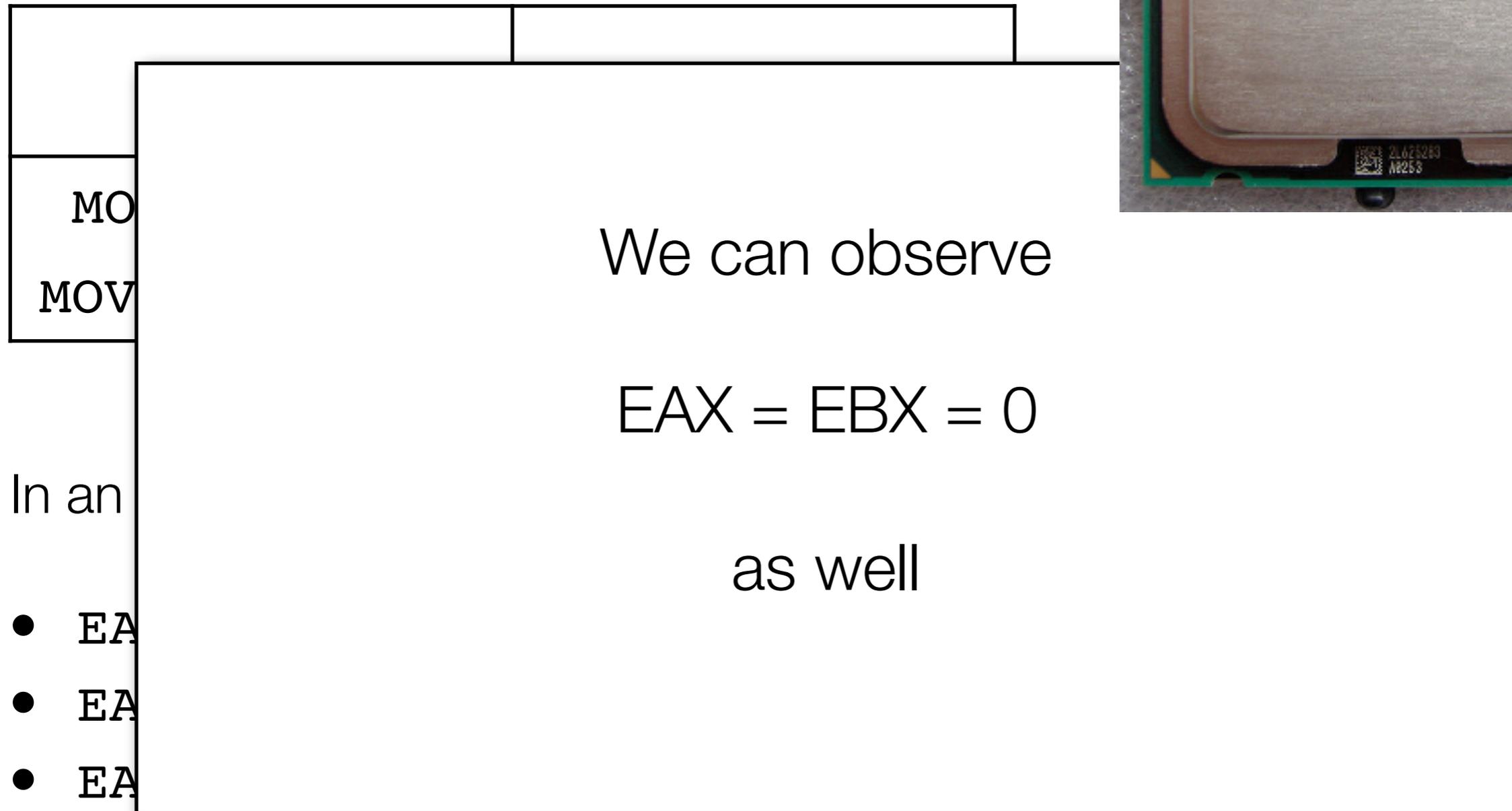
Thread 0	Thread 1
 <code>MOV [x] ← 1</code>	<code>MOV [y] ← 1</code>
<code>MOV EAX ← [y]</code>	<code>MOV EBX ← [x]</code> 

Can you guess the final register values: **EAX = 1** **EBX = 0**

The first shocking example



The first shocking example



According to most programmers

Multiple
threads

Multiple

(local s

These

- und

- sometimes observable by concurrent code.

Upshot:

only a relaxed (or weakly consistent)
view of the memory.

SE

...)

Not new

Multiprocessors since 1964 (Univac 1108A)

Relaxed Memory since 1972 (IBM System 370/158MP)

Eclipsed for a long time (except in high-end) by advances in performance:

- transistor counts (continuing)
- clock speed (hit power dissipation limit)
- ILP (hit smartness limit?)

Mass-market multiprocessing, since 2005.

Programming multiprocessors no longer just for specialists.



But it's hard!

1. P

2. P

3. P

**Industrial processors and language specs
are often flawed**

We've looked at the specs of x86, Power, ARM, Java, and C++

Alm

cod

They all have problems

Muc

processors and languages (new trend in the last few years).

Hardware models

Architectures

Hardware manufacturers document **architectures**:

loose specifications

covering a **wide range** of past and future **processor implementations**.

Architectures should:

- **reveal enough** for effective programming;
- **without unduly constraining** future processor design.

Examples: Intel 64 and IA-32 Architectures SDM, AMD64 Architecture Programmer's Manual, Power ISA specification, ARM Architecture Reference Manual, ...



Intel[®] 64 and IA-32 Architectures
Software Developer's Manual



VOLUME 3A: System Programming Guide
Part 1



In practice

Architectures described by informal prose:

In a multiprocessor system, maintenance of cache consistency may, in rare circumstances, require intervention by system software.

(Intel SDM, november 2006, vol3a, 10-5)

As we shall see, such descriptions are:

1) vague; 2) incomplete; 3) unsound.

Fundamental problem: prose specifications cannot be used to test programs or to test processor implementations.

Intel 64/IA32 and AMD64 - before Aug. 2007

Era of Vagueness

A model called **Processor Ordering**, informal prose.

Example: Linux kernel mailing list, 20 nov. - 7 déc. 1999 (143 posts).

A one-instruction programming question, a microarchitectural debate!

Keywords: speculation, ordering, causality, retire, cache...

1. spin_unlock() Optimization On Intel

20Nov1999-7Dec1999 (143 posts) Archive Link: "[spin_unlock_optimization\(i386\)](#)"

Topics: [BSD](#), [FreeBSD](#), [SMP](#)

People: [Linus Torvalds](#), [Jeff V. Merkey](#), [Erich Boleyn](#), [Manfred Spraul](#), [Peter Samuelson](#), [Ingo Molnar](#)

Manfred Spraul thought he'd found a way to shave spin_unlock() down from about 22 ticks for the "lock; btrl \$0,%0" asm code, to 1 tick for a simple "movl \$0,%0" instruction, a huge gain. Later, he reported that Ingo Molnar noticed a 4% speed-

Note that I actually thought this was a legal optimization, and for a while I had this in the kernel. It crashed. In random ways.

Note that the fact that it does not crash now is quite possibly because of either

we have a lot less contention on our spinlocks these days. That might hide the

```
spin_lock()
a = 1;
mb();
a = 0;
mb();
b = a;
spin_unlock();
return b;
}
```

"You report that Linus was convinced to do the spinlock optimization on Intel, but apparently someone has since changed his mind back. See <asm-i386/spinlock.h> from 2.3.30pre5 and above:

```
/* Sadly, some early PPro chips require the locked
 * access, otherwise we could just always simply do
 *
 * #define spin_unlock_string \
 * "movb $0,%0"
 *
 * Which is noticeably faster.
 */
#define spin_unlock_string \
"lock ; btrl $0,%0"
```

thing you need is to make sure there is
kind of true by the fact that you're checking
observable on other processors.

Intel 64/IA32 and AMD64 - Aug. 2007 / Oct. 2008

Intel publishes a white paper, defining 8 informal-prose principles, e.g.

P1. Loads are not reordered with older loads.

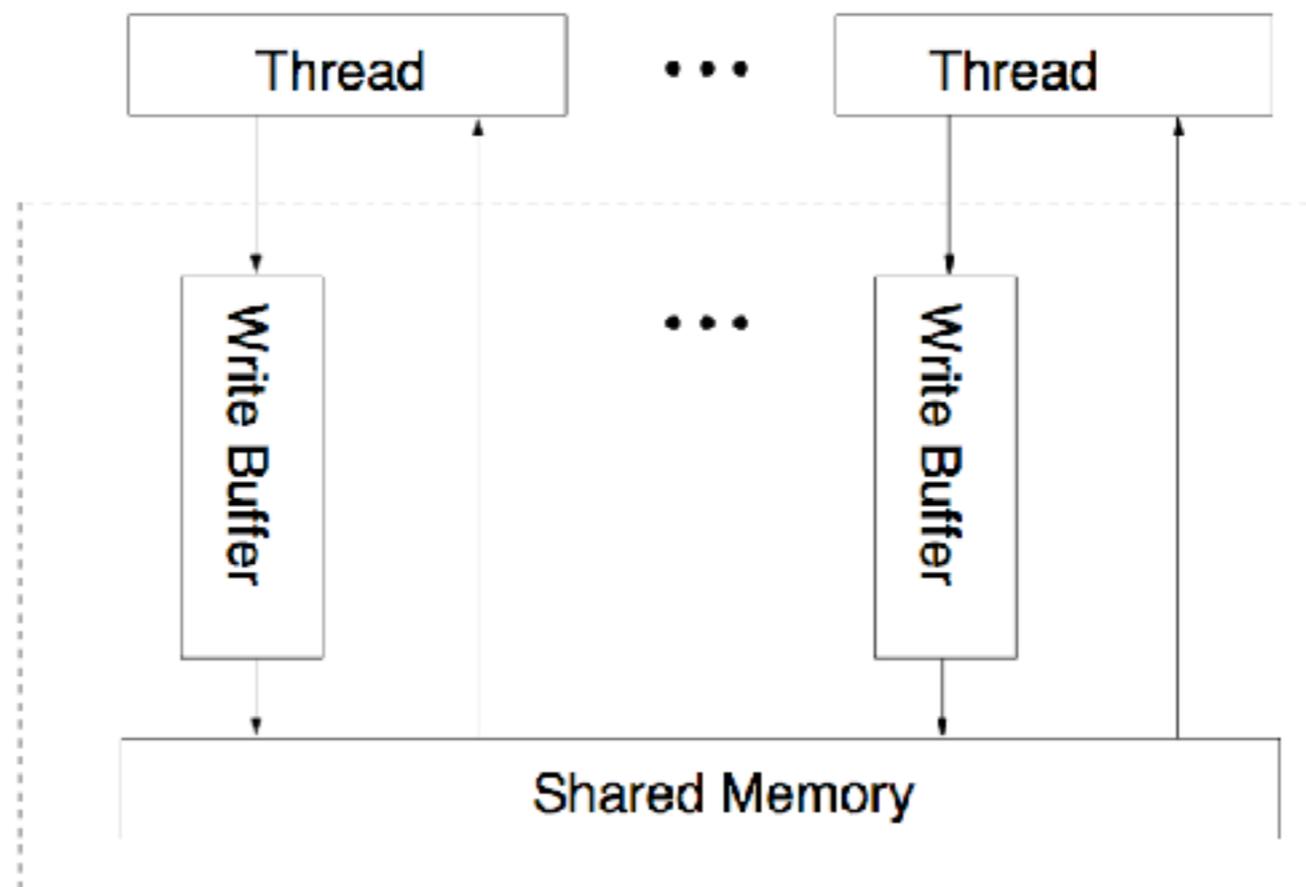
P2. Stores are not reordered with older stores.

supported by 10 litmus test (illustrating allowed or forbidden behaviours), e.g.:

Thread 0	Thread 1
MOV [x] ← 1	MOV EAX ← [y] (1)
MOV [y] ← 1	MOV EBX ← [x] (0)
Forbidden final state: $EAX = 1 \wedge EBX = 0$	

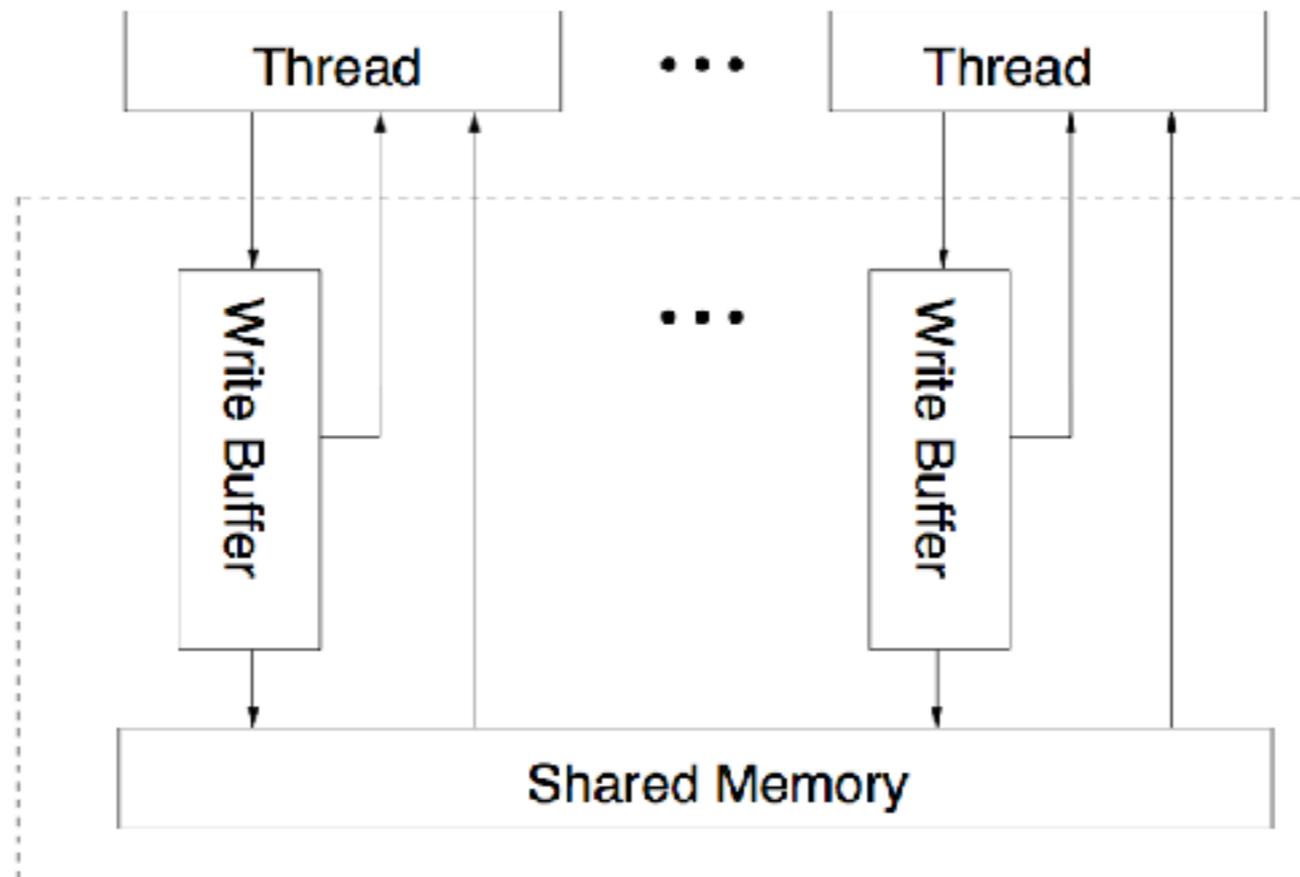
P3. Loads may be reordered with older stores to different locations but not with older stores to the same location.

Thread 0	Thread 1
MOV [x] ← 1	MOV [y] ← 1
MOV EAX ← [y] (0)	MOV EBX ← [x] (0)
Allowed final state: $0:EAX = 0 \wedge 1:EBX = 0$	



Litmus test 2.4: intra-processor forwarding is allowed

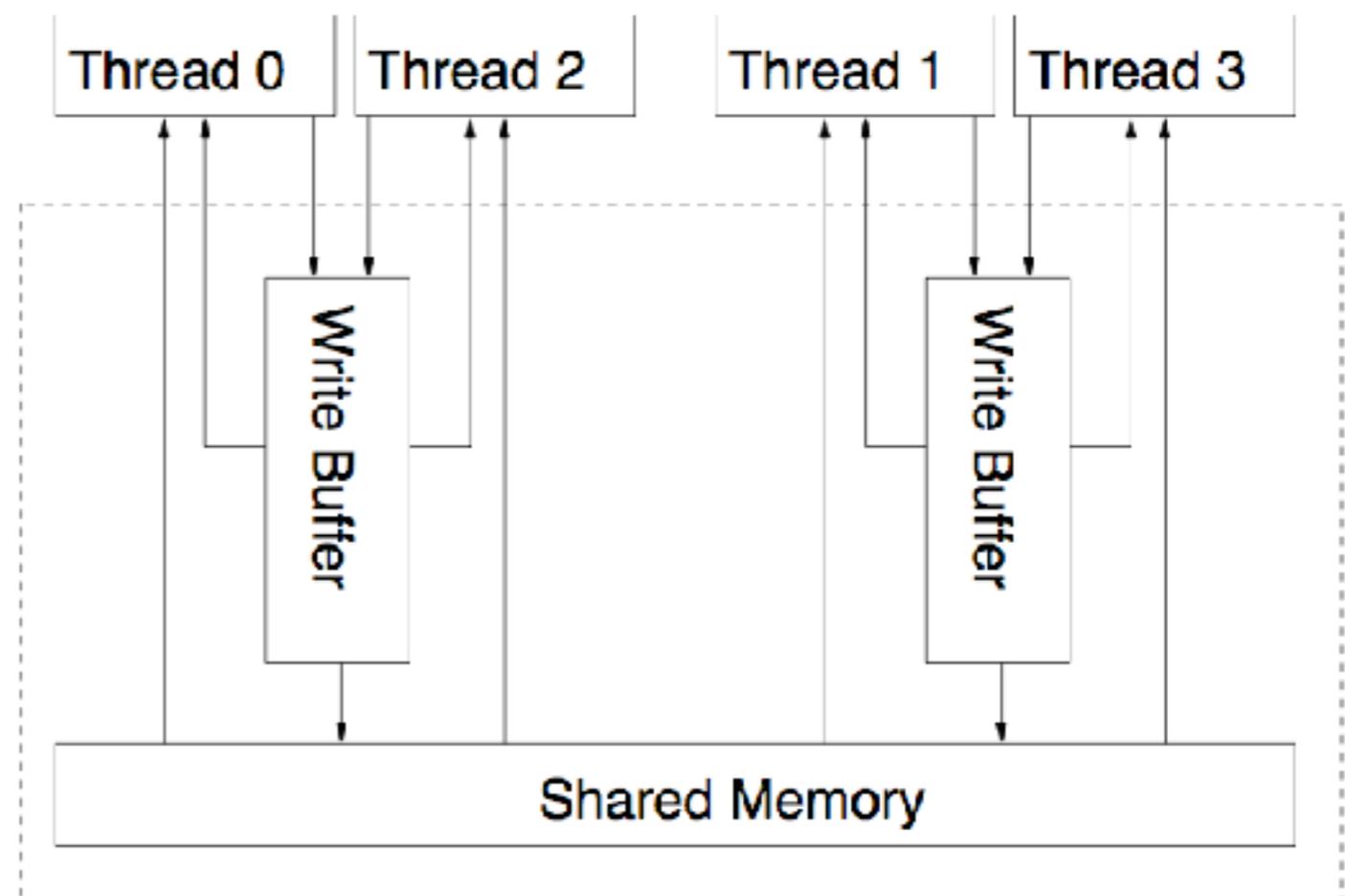
Thread 0	Thread 1
MOV [x] ← 1	MOV [y] ← 1
MOV EAX ← [x] (1)	MOV ECX ← [y] (1)
MOV EBX ← [y] (0)	MOV EDX ← [x] (0)
Allowed final state: $0:EAX = 1 \wedge 0:EBX = 0 \wedge$ $1:ECX = 1 \wedge 1:EDX = 1$	



Thread 0	Thread 1	Thread 2	Thread 3
MOV [x] ← 1	MOV [y] ← 1	MOV EAX ← [x] (1)	MOV ECX ← [y] (1)
		MOV EBX ← [y] (0)	MOV EDX ← [x] (0)
Final state: 2:EAX = 1 ∧ 2:EBX = 0 ∧ 3:ECX = 1 ∧ 3:EDX = 0			

Microarchitecturally plausible?

Yes, with e.g. shared store buffers.



Ambiguity

P1-P4: ... may be reordered with ...

*P5: In
i.e. store
consis*

Ambiguity:

when are two stores casually related?

Thread 0	Thread 1	Thread 2
MOV [x] ← 1	MOV EAX ← [x] MOV [y] ← 1	MOV EBX ← [y] (1) MOV ECX ← [x] (0)
Forbidden final state: $1:EAX = 1 \wedge 2:EBX = 1 \wedge 2:ECX = 0$		

Unsoundness

Example from Paul Loewenstein:

The Intel White Paper specification
is unsound

(and our POPL x86-CC paper too)

Observe

- “Stores to the same location have a total order”

Intel 64/IA32 and AMD64, Nov. 2008 - now

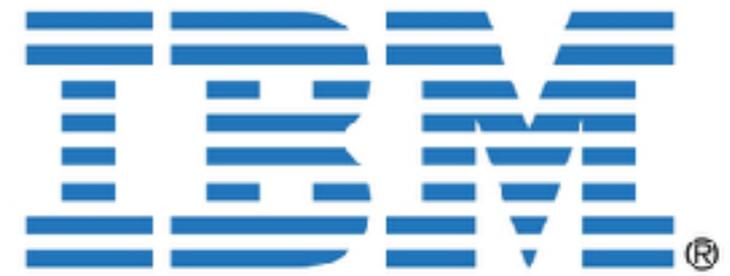
SDM rev 29-31.

- Not unsound in the previous sense
- Explicitly exclude IRIW, so not weak in that sense. New principle:

Any two stores are seen in a consistent order by processors other than those performing the stores.

But... still ambiguous, and the view by those processors is left entirely unspecified!

Thread 0	Thread 1
MOV [x] ← 1	MOV [x] ← 2
MOV EAX ← [x] (2)	MOV EBX ← [x] (1)
0:EAX = 2 ∧ 1:EBX = 1	



Power ISA 2.06 and ARM v7

Key concept: actions being performed.

A load by a processor (P1) is performed with respect to any

"all that horrible horribly incomprehensible and confusing [...] text that no-one can parse or reason with — not even the people who wrote it"

— Anonymous Processor Architect, 2011

A memory model should define if a particular execution is allowed.

It is awkward to make a definition that **explicitly quantifies over all hypothetical variant executions.**

Why all these problems?

Recall

loos

cover

Archite

- reve

- with

There is a big tension between these, with internal politics and inertia.

ations.

Compounded by the informal-prose specification style.

Hardware models:

inventing a usable abstraction for x86

Requirements

- Unambiguous
- Sound w.r.t. experience
- Easy to understand
- Consistent with what we know of vendor intentions
- Consistent with expert-programmer reasoning

Key facts for x86

- Store buffering (with forwarding) is observable
- IRIW is not observable and forbidden by recent docs
- Various other reorderings are not observable and are forbidden

These suggests that x86 is, in practice, like Sparc TSO.

Instructions and events

Initially $[x] = 0$.

Thread 0	Thread 1
INC $[x]$	INC $[x]$

Are we guaranteed that $[x] = 2$ at the end of the execution?

No: $[x] = 1$ is possible.

The instruction `INC $[x]$` is composed by two *atomic events*:

- read the content of the memory location $[x]$;
- write the new content of the memory location $[x]$.

Locked instructions

Thread 0	Thread 1
INC [x]	INC [x]

[x] = 1 is possible

Thread 0	Thread 1
Lock; INC [x]	Lock; INC [x]

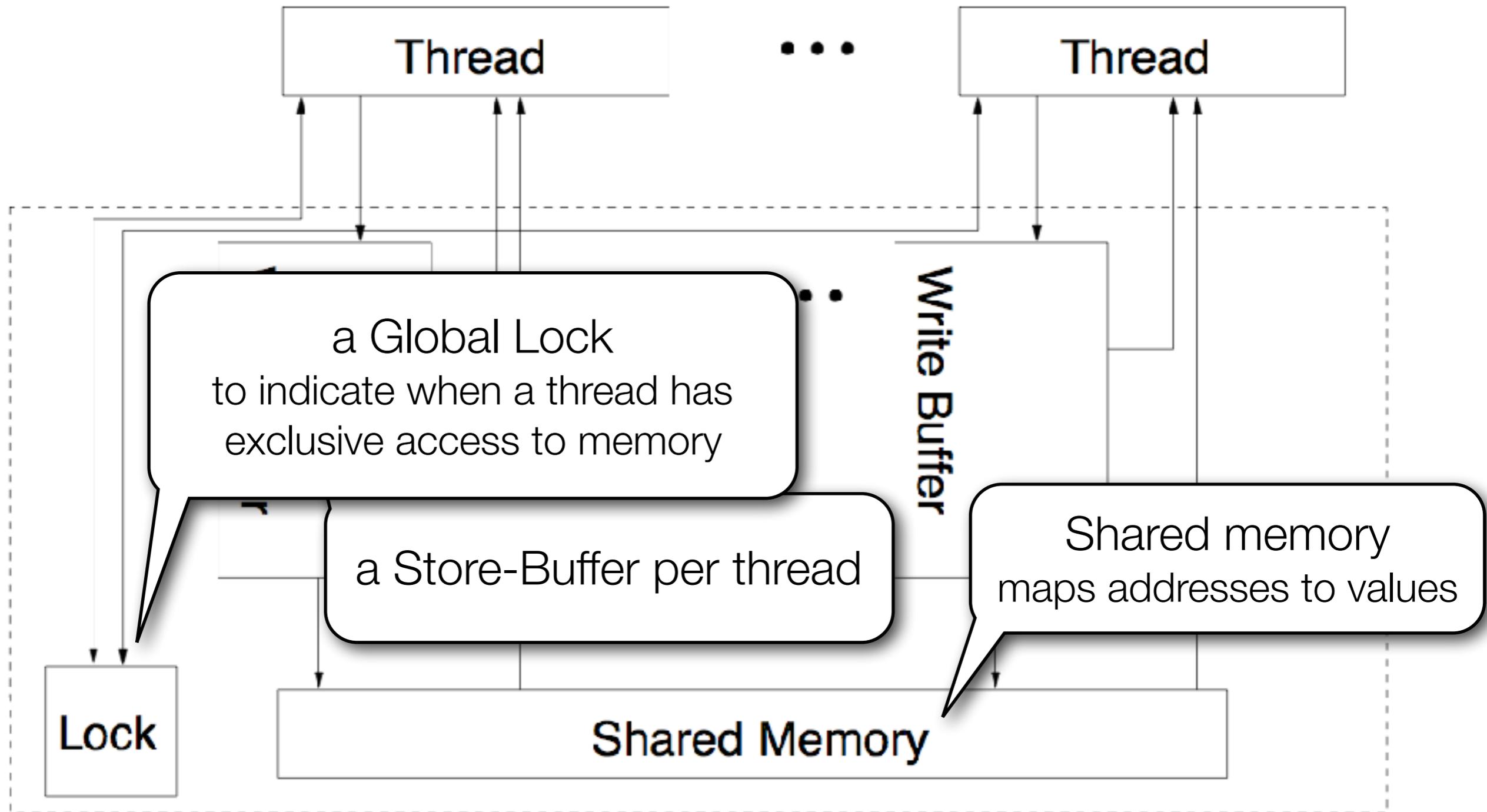
[x] = 1 is forbidden

Also, Lock's ADD, SUB, XCHG, etc., and CMPXCHG

x86-TSO abstract machine

1. Separate instruction semantics and memory model
2. The memory model is defined over *events* rather than *instructions*
3. Define the memory model in two (provably equivalent) styles:
 - an abstract machine (or operational model)
 - an axiomatic model

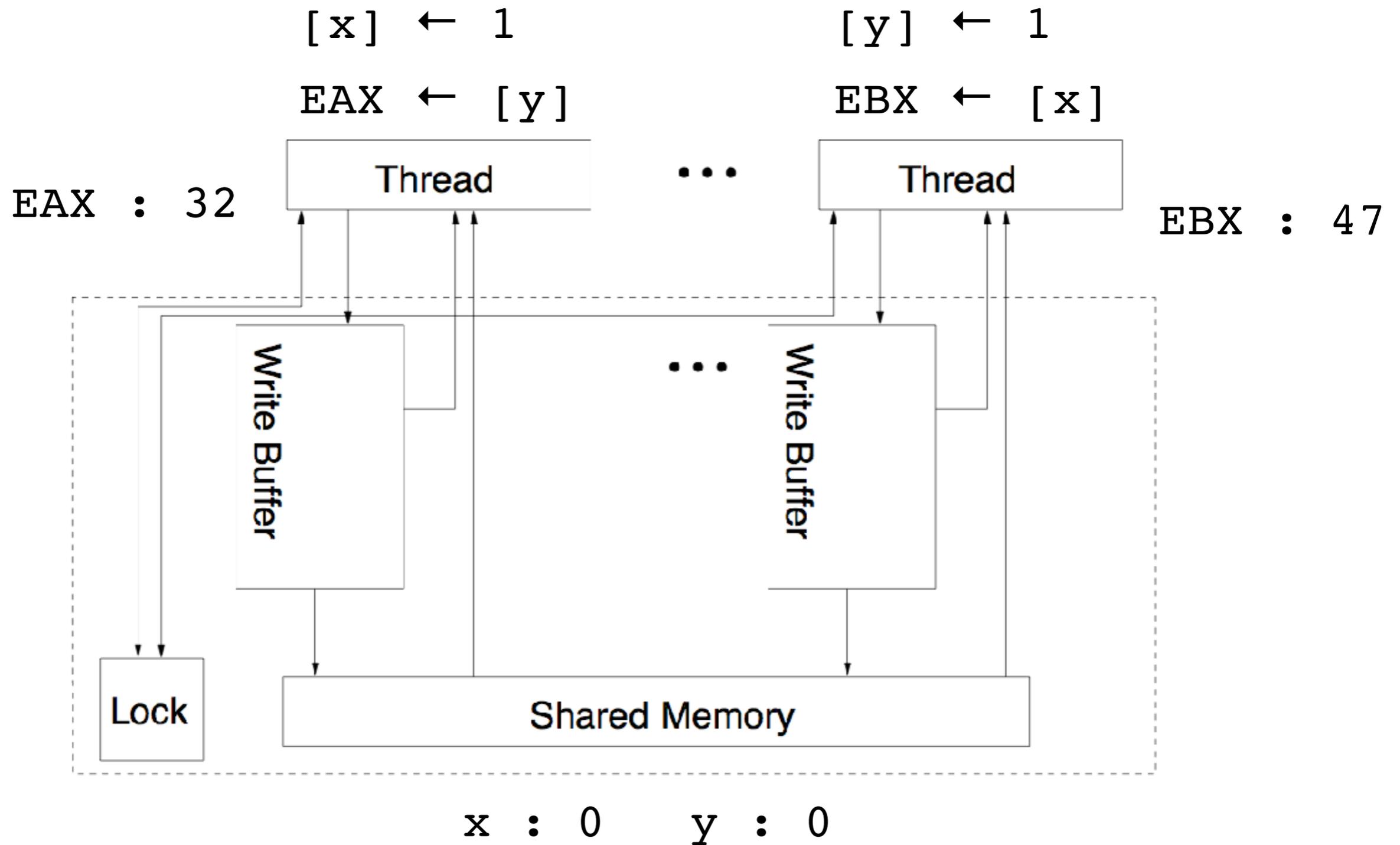
x86-TSO abstract machine



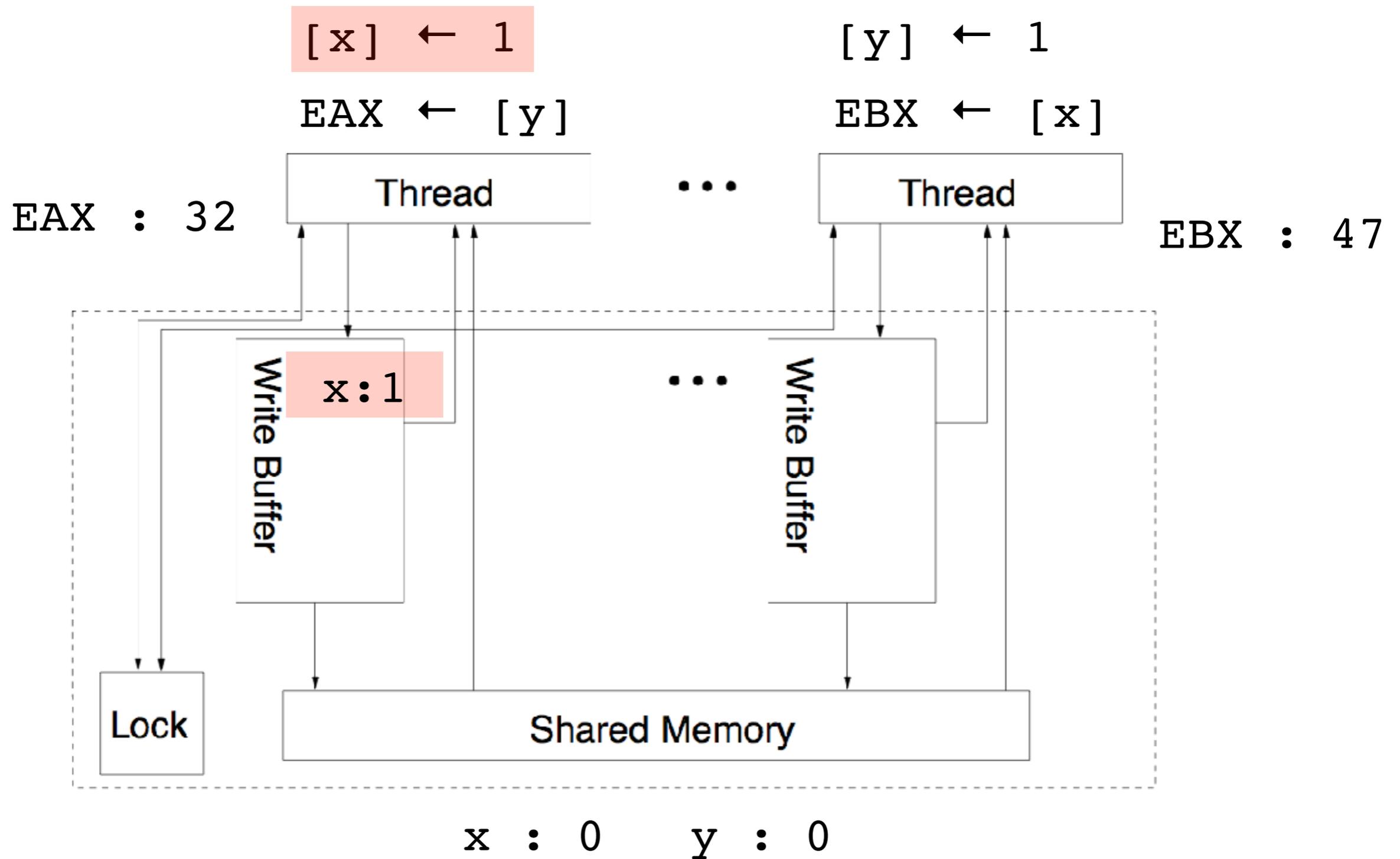
x86-TSO abstract machine

- The store buffers are FIFO. A reading thread must read its most recent buffered write, if there is one, to that address; otherwise reads are satisfied from shared memory.
- To execute a LOCK'd instruction, a thread must first obtain the global lock. At the end of the instruction, it flushes its store buffer and relinquishes the lock. While the lock is held by one thread, no other thread can read.
- A buffered write from a thread can propagate to the shared memory at any time except when some other thread holds the lock.

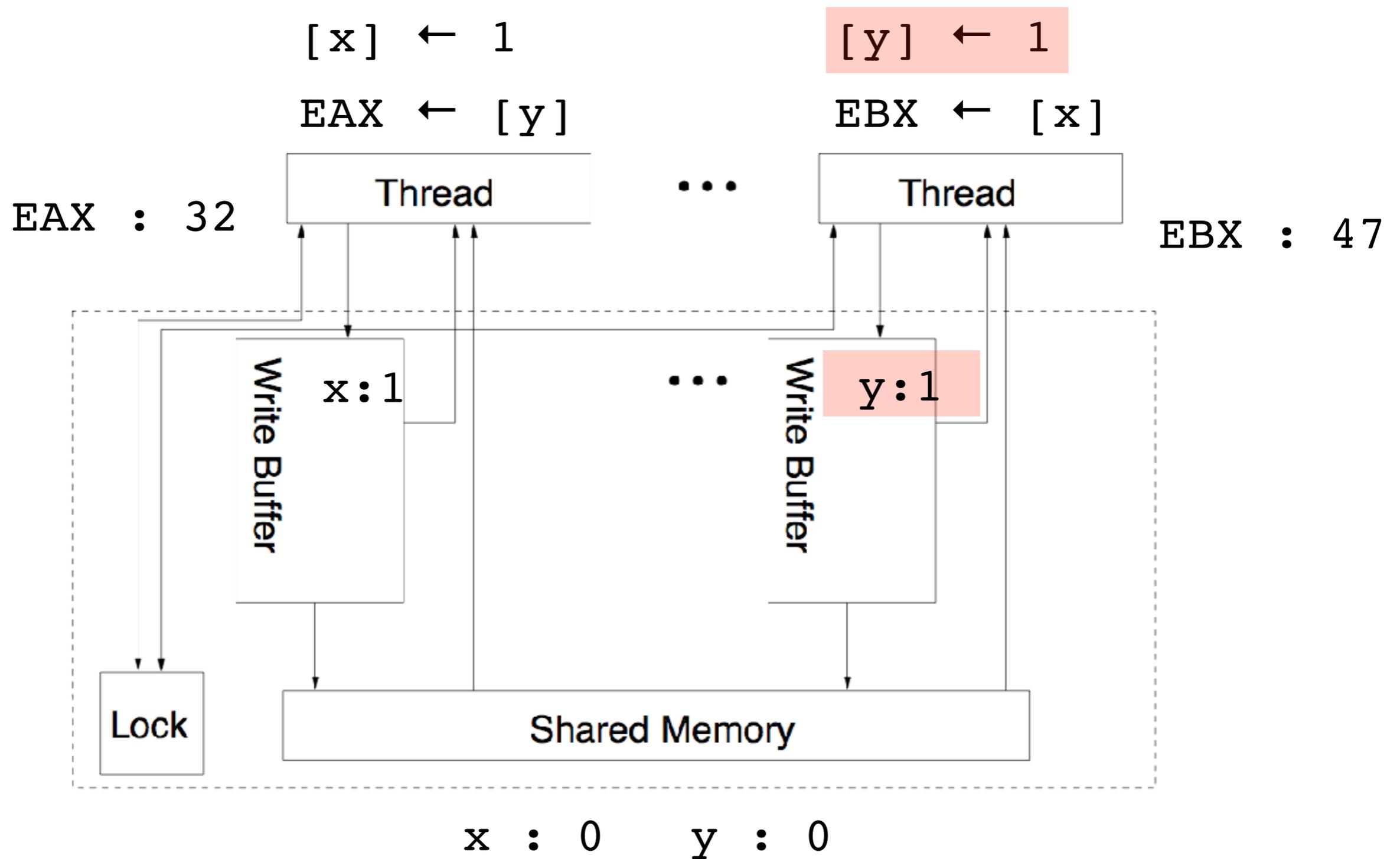
The *not-so shocking* first example



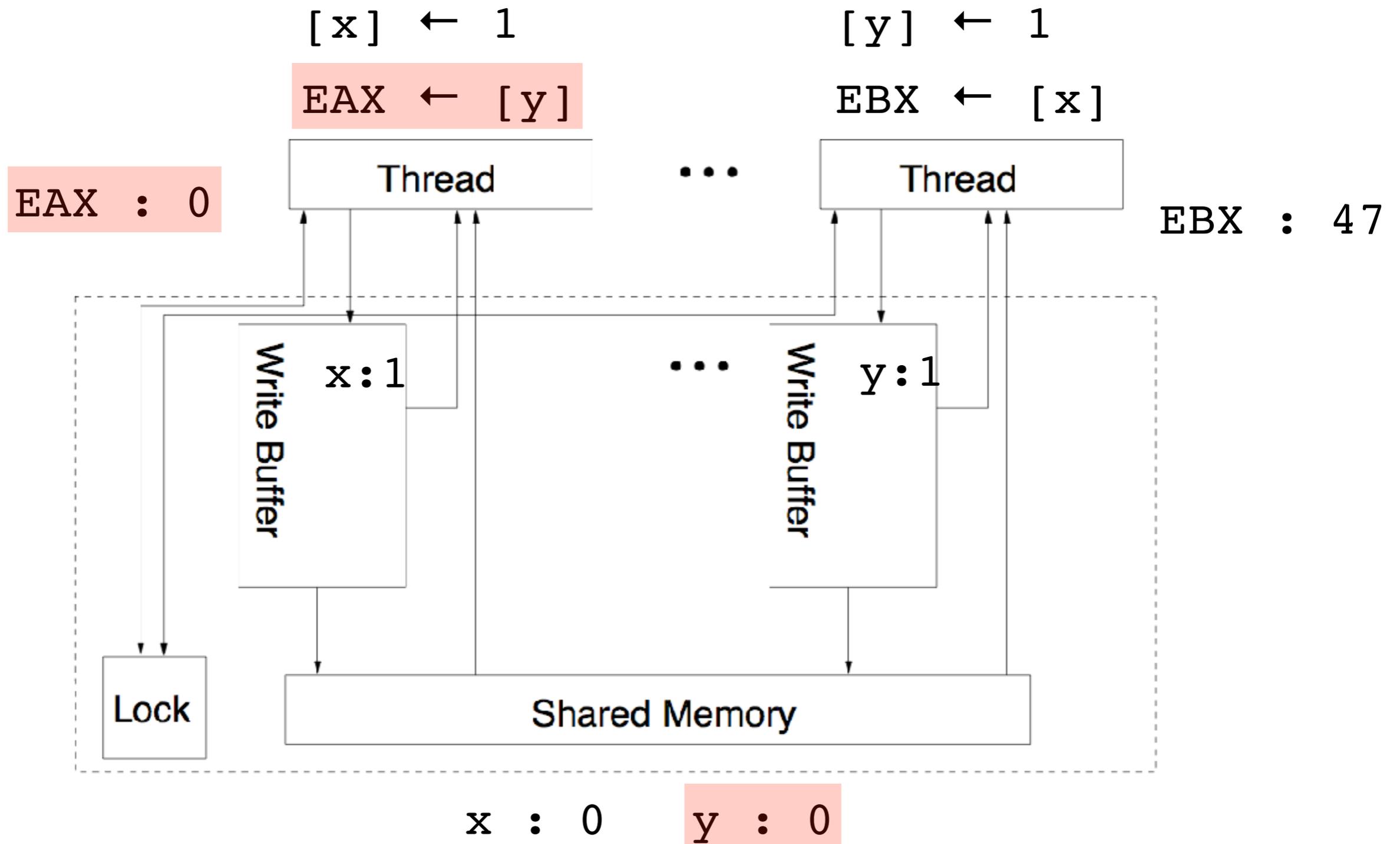
The *not-so shocking* first example



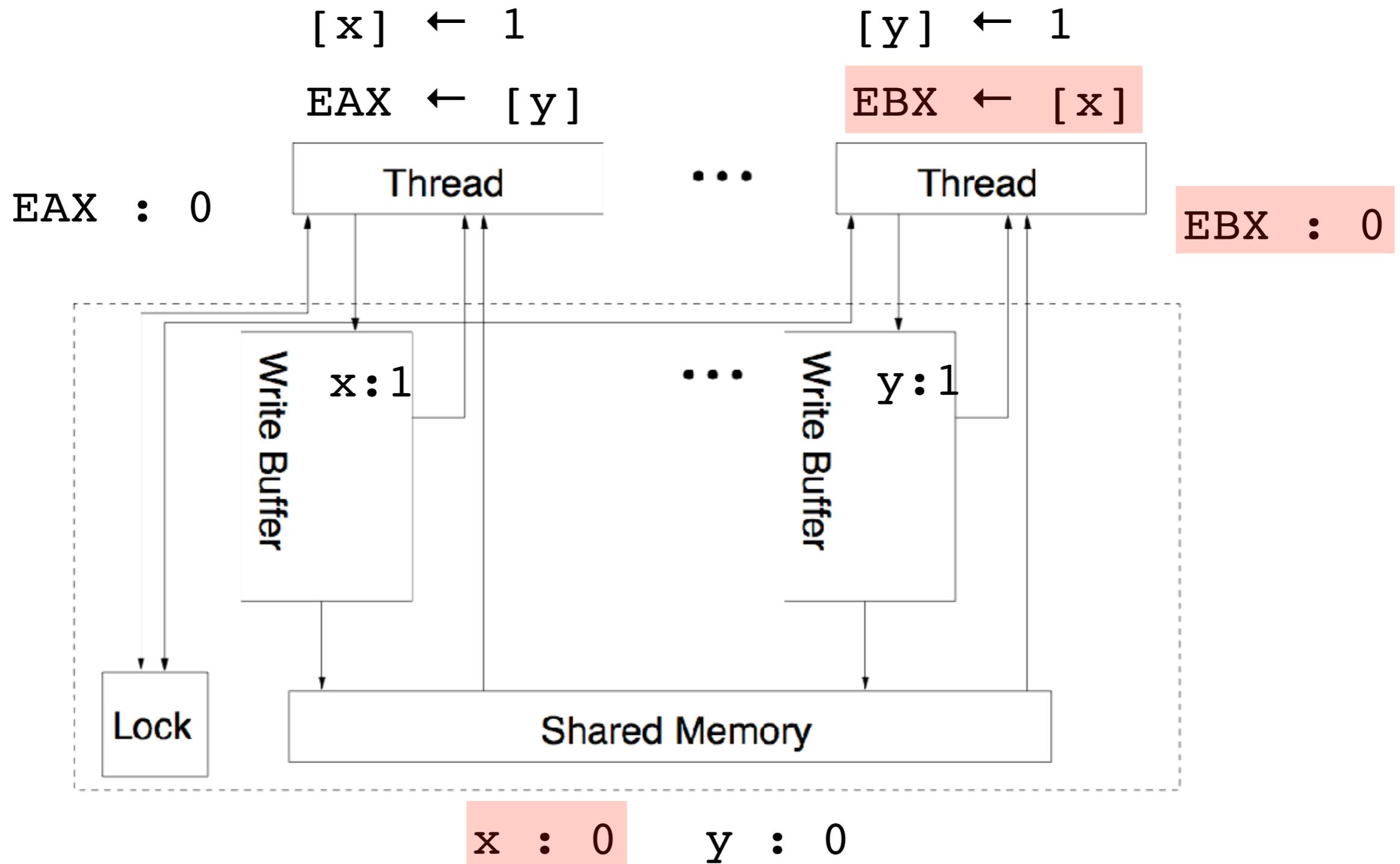
The *not-so shocking* first example



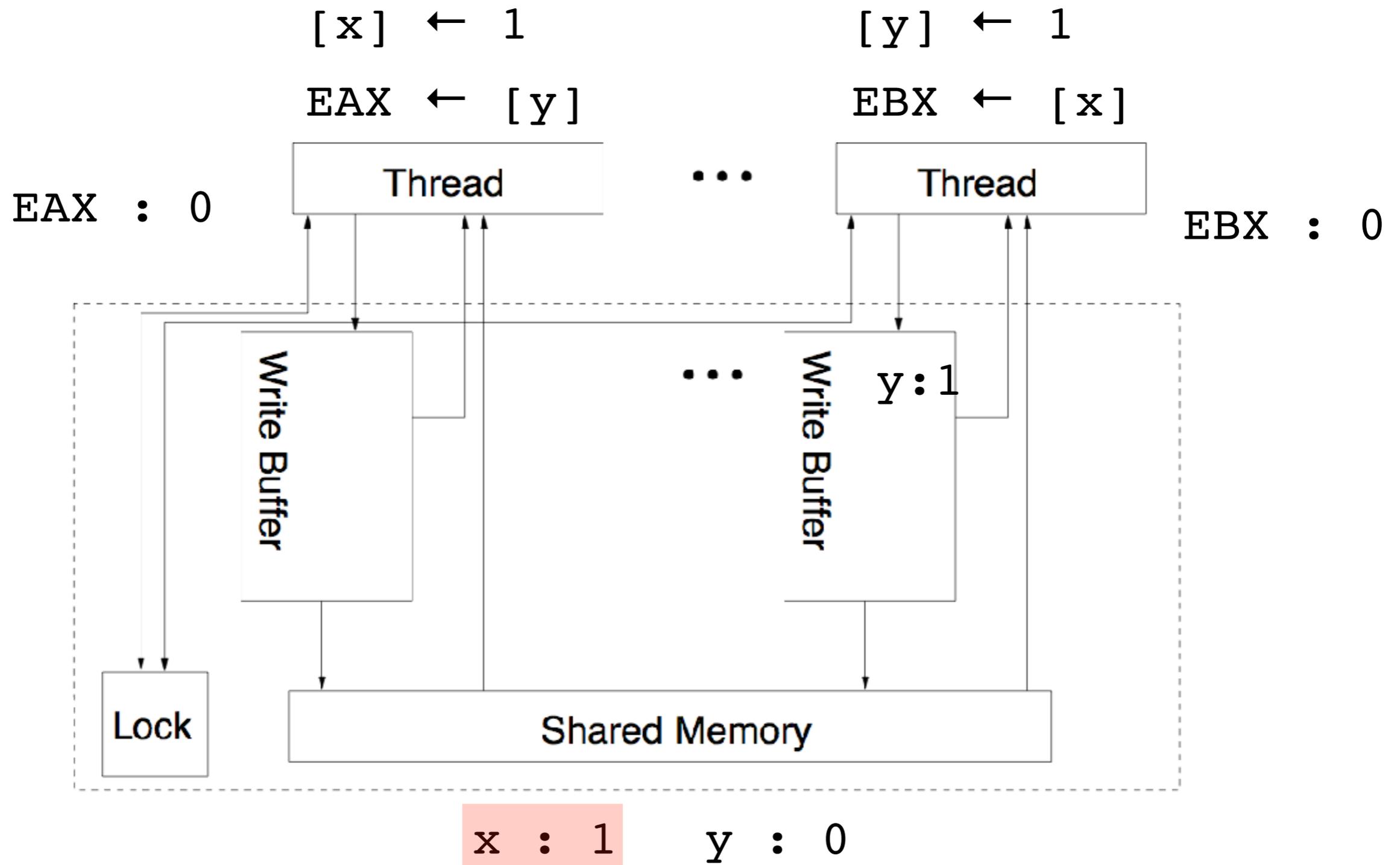
The *not-so shocking* first example



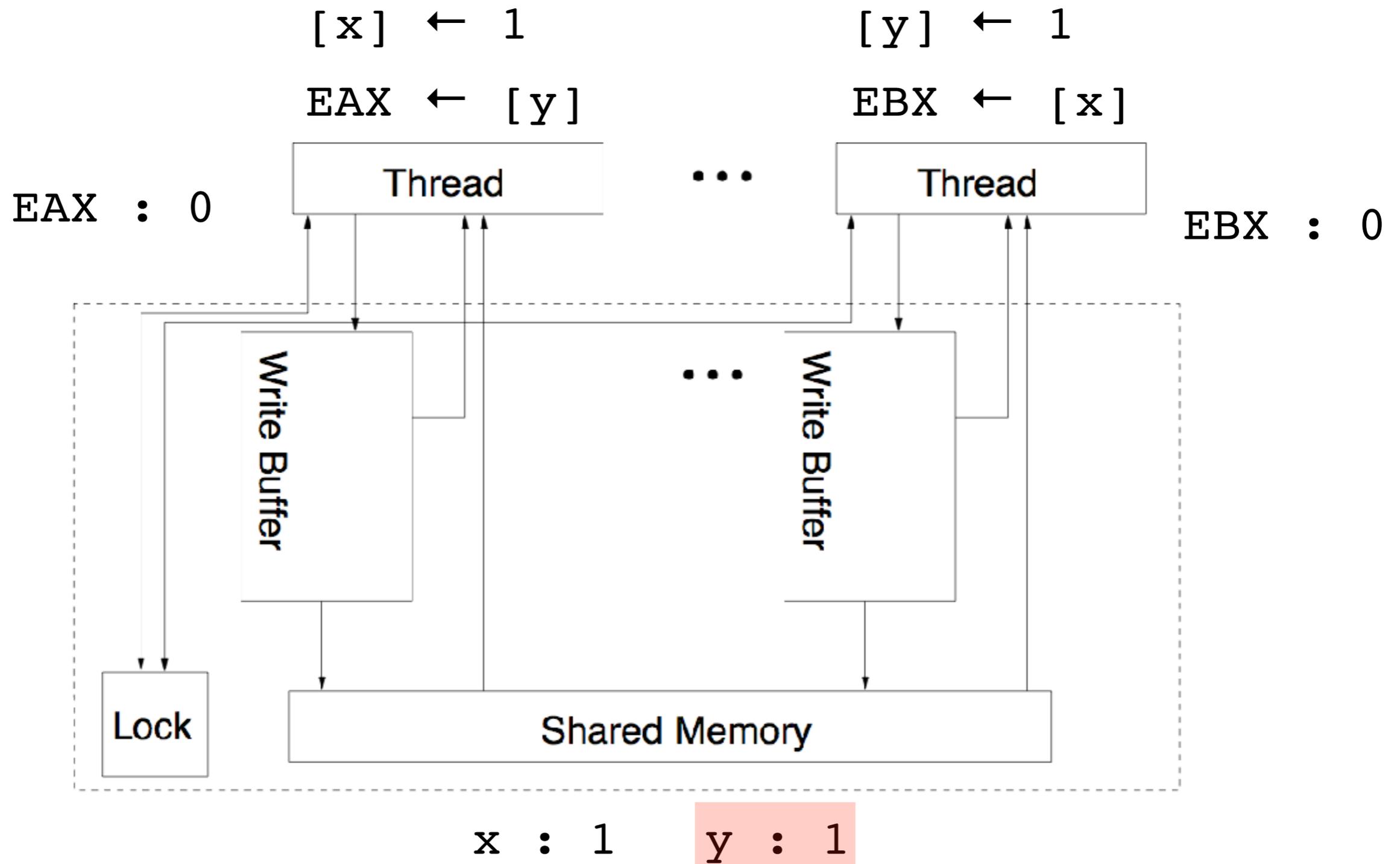
The *not-so shocking* first example



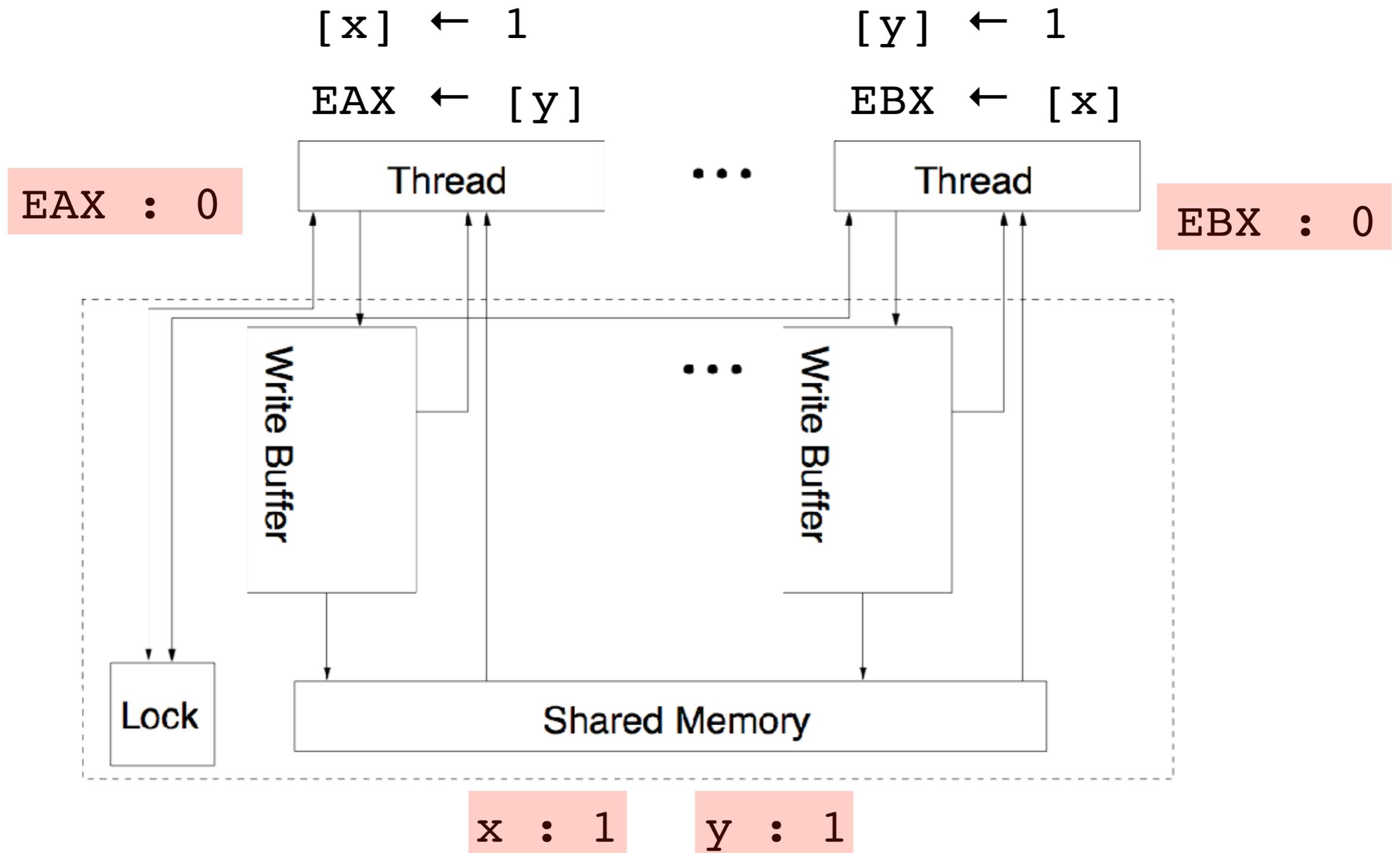
The *not-so shocking* first example



The *not-so shocking* first example



The *not-so shocking* first example



Linux Spinlock Optimisation

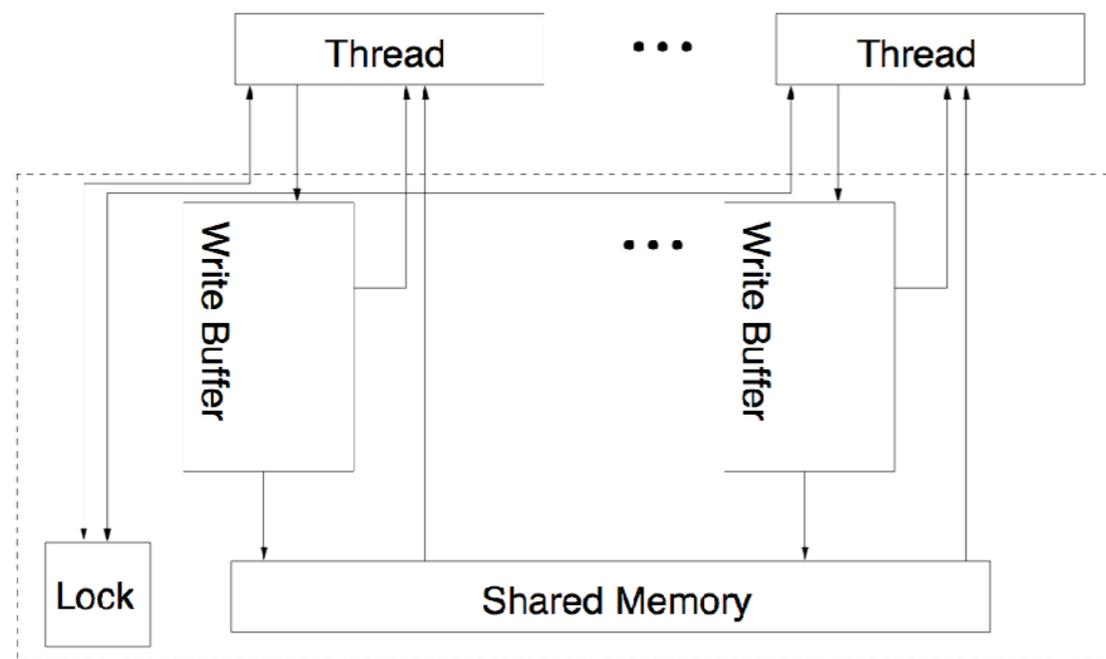
On entry the address of spinlock is in register EAX and the spinlock is unlocked iff its value is 1	
acquire:	LOCK;DEC [EAX] ; LOCK'd decrement of [EAX]
	JNS enter ; branch if [EAX] was ≥ 1
spin:	CMP [EAX],0 ; test [EAX]
	JLE spin ; branch if [EAX] was ≤ 0
	JMP acquire ; try again
enter:	; the critical section starts here
release:	MOV [EAX] \leftarrow 1

Sample properties:

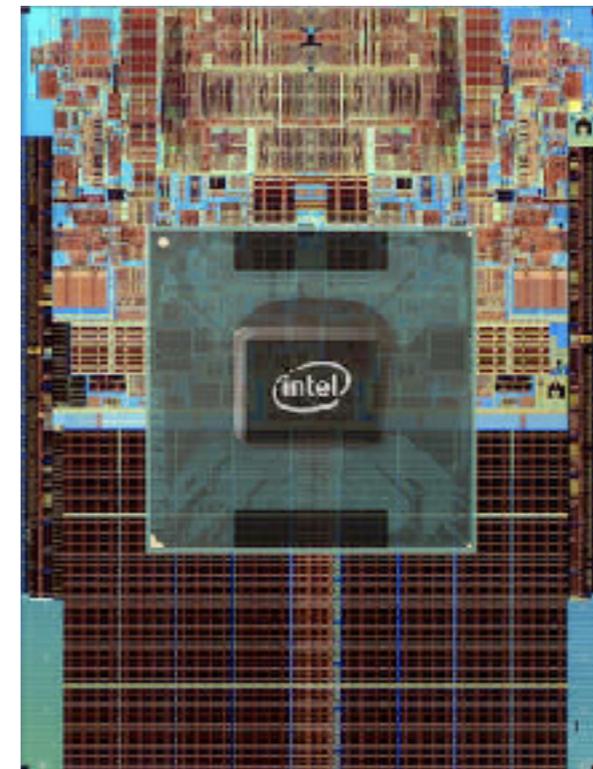
1. only one thread can acquire the spinlock at a time;
2. all writes performed inside a critical section must have been propagated to main memory before another thread can acquire the spinlock.

NB: this is an abstract machine

A tool to specify exactly and only the programmer-visible behaviour, not a description of the implementation internals.



\supseteq
be
 \neq
h



Force: of the the internal optimizations of processors, only per-thread FIFO write buffers are visible to programmers.

Still quite a loose spec: unbounded buffers, nondeterministic unbuffering, arbitrary interleaving

Read from memory

$$\frac{\text{not_blocked } s \ p \wedge (s.M \ a = \text{SOME } v) \wedge \text{no_pending } (s.B \ p) \ a}{s \xrightarrow{\text{EVT } p \ (\text{ACCESS } R \ (\text{LOCATION_MEM } a) \ v)} s}$$

Read from write buffer

$$\frac{\text{not_blocked } s \ p \wedge (\exists b_1 \ b_2. (s.B \ p = b_1 \ ++ \ [(a, v)] \ ++ \ b_2) \wedge \text{no_pending } b_1 \ a)}{s \xrightarrow{\text{EVT } p \ (\text{ACCESS } R \ (\text{LOCATION_MEM } a) \ v)} s}$$

Read from register

$$\frac{(s.R \ p \ r)}{s \xrightarrow{\text{EVT } p \ (\text{ACCESS } R \ r)} s}$$

Write to write buffer

$$\frac{\text{EVT } p \ (\text{ACCESS } W \ a \ v)}{s \oplus (B := s.B \oplus (p \ \vdash \ a \ v))}$$

Write from write buffer

$$\frac{\text{not_block}}{s \xrightarrow{\text{TAU}} s \oplus (M := s.M \oplus (p \ \vdash \ a \ v))}$$

Write to register

$$\frac{\text{EVT } p \ (\text{ACCESS } W \ a \ v)}{s \oplus (R := s.R \oplus (p \ \vdash \ a \ v))}$$

Barrier

$$\frac{(b = \text{MFENCE}) \implies (b = \text{MFENCE})}{s \xrightarrow{\text{EVT } p \ (\text{BARRIER } b)} s}$$

Lock

$$\frac{(s.L = \text{NONE}) \wedge (s.B \ p = [])}{s \xrightarrow{\text{LOCK } p} s \oplus (L := \text{SOME } p)}$$

Unlock

$$\frac{(s.L = \text{SOME } p) \wedge (s.B \ p = [])}{s \xrightarrow{\text{UNLOCK } p} s \oplus (L := \text{NONE})}$$

reads_from_map_candidates =

$$\forall (ew, er) \in \text{rfmap}. (er \in \text{reads } E) \wedge (ew \in \text{writes } E) \wedge (\text{loc } ew = \text{loc } er) \wedge (\text{value_of } ew = \text{value_of } er)$$

check_rfmap_written =

$$\forall (ew, er) \in (X.\text{rfmap}).$$

if $ew \in \text{mem_accesses } E$ **then**

$$ew \in \text{maximal_elements } (\text{previous_writes } E \ er \ X.\text{memory_order} \cup \text{previous_writes } E \ er \ (\text{po_iico } E)) \ X.\text{memory_order}$$

else

Mathematics (in HOL4)
rather than informal prose.

$$(ew, ej) \in \text{po_iico } E \wedge (ej, er) \in \text{po_iico } E \implies$$

$$(ew, er) \in X.\text{memory_order} \wedge$$

$$(\forall e_1 \ e_2 \in (\text{mem_accesses } E). \forall es \in (E.\text{atomicity}).$$

$$(e_1 \in es \vee e_2 \in es) \wedge (e_1, e_2) \in \text{po_iico } E$$

$$\implies$$

$$(e_1, e_2) \in X.\text{memory_order} \wedge$$

$$(\forall es \in (E.\text{atomicity}). \forall e \in (\text{mem_accesses } E \setminus es).$$

$$(\forall e' \in (es \cap \text{mem_accesses } E). (e, e') \in X.\text{memory_order}) \vee$$

$$(\forall e' \in (es \cap \text{mem_accesses } E). (e', e) \in X.\text{memory_order})) \wedge$$

$$X.\text{rfmap} \in \text{reads_from_map_candidates } E \wedge$$

$$\text{check_rfmap_written } E \ X \wedge$$

$$\text{check_rfmap_initial } E \ X$$



Hardware models:

inventing a usable abstraction for Power/ARM

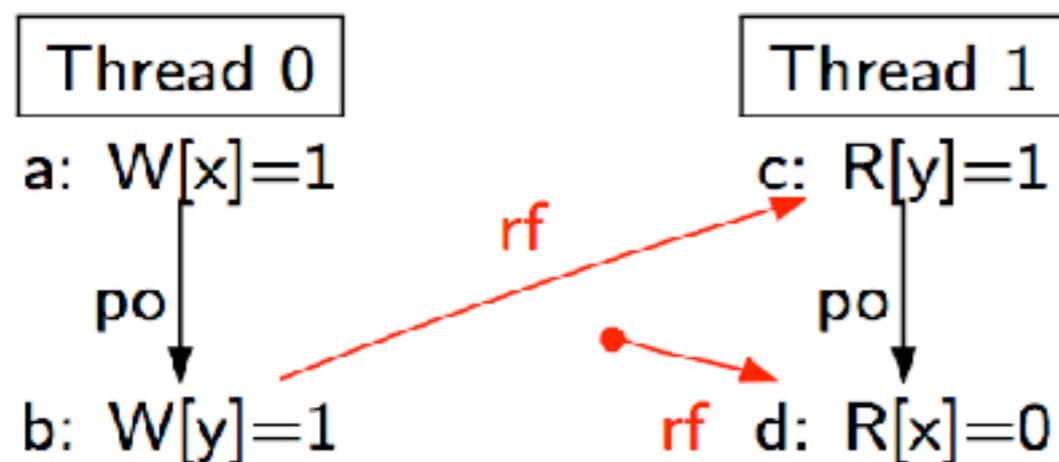
Disclaimer:

1. ARM MM is analogous to Power MM... all this is your phone!
2. The model I will present is (as far as we know) accurate for ARM if barriers weaker than DMB are not used.
...but ARM chips seem to have bugs - ask Luc for details.

Power: much more relaxed than x86

Thread 0	Thread 1
<code>x = 1</code>	<code>while (y==0) {};</code>
<code>y = 1</code>	<code>r = x</code>

Observable behaviour: `r = 0`



Forbidden on SC and x86-TSO

Allowed and observed on Power

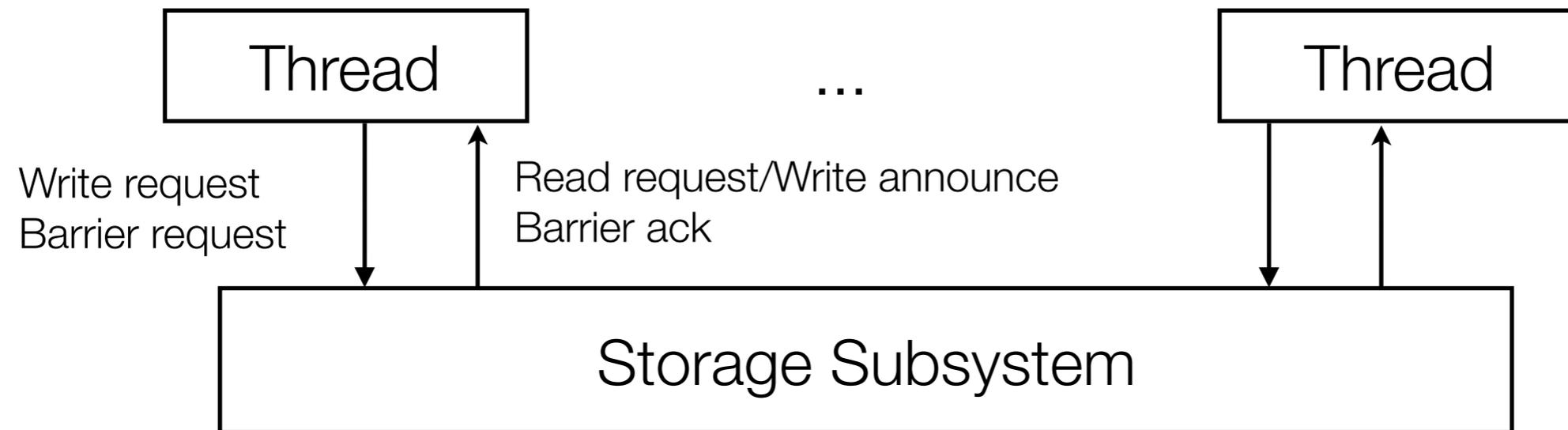
Power: much more relaxed than x86

Three pos

Power has all three!

1. the two writes are performed in opposite order
reordering store buffers
2. the two reads are performed in opposite order
load reorder buffers / speculation
3. propagation of writes ignores order in which they are presented
interconnects partitioned by address (cache lines)

The model overall structure



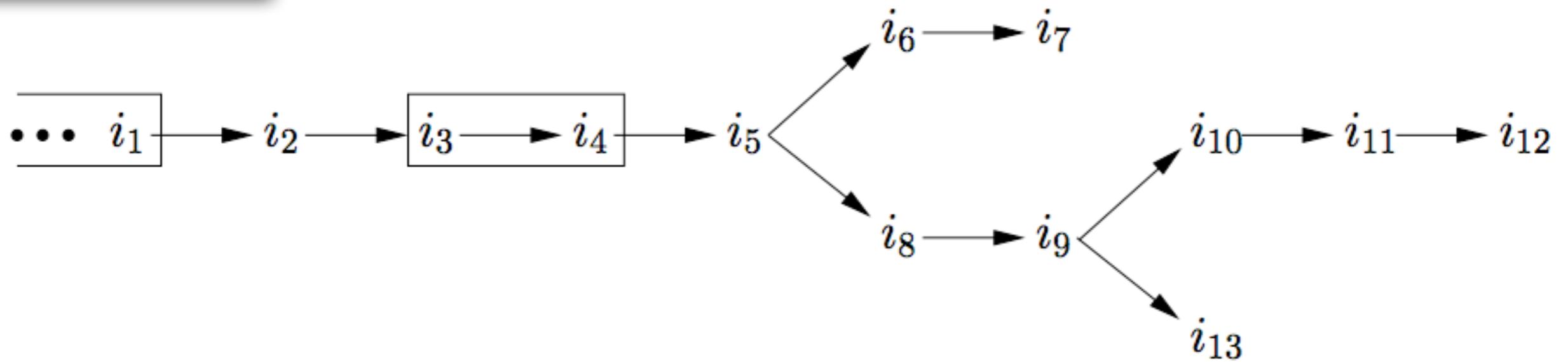
Some aspects are thread-only, some storage-only, some both.

Threads and storage subsystem are abstract state machines.

Speculative execution in Threads; topology-independent Storage.

*Much more complicated than x86-TSO.
Are you ready?*

Thread



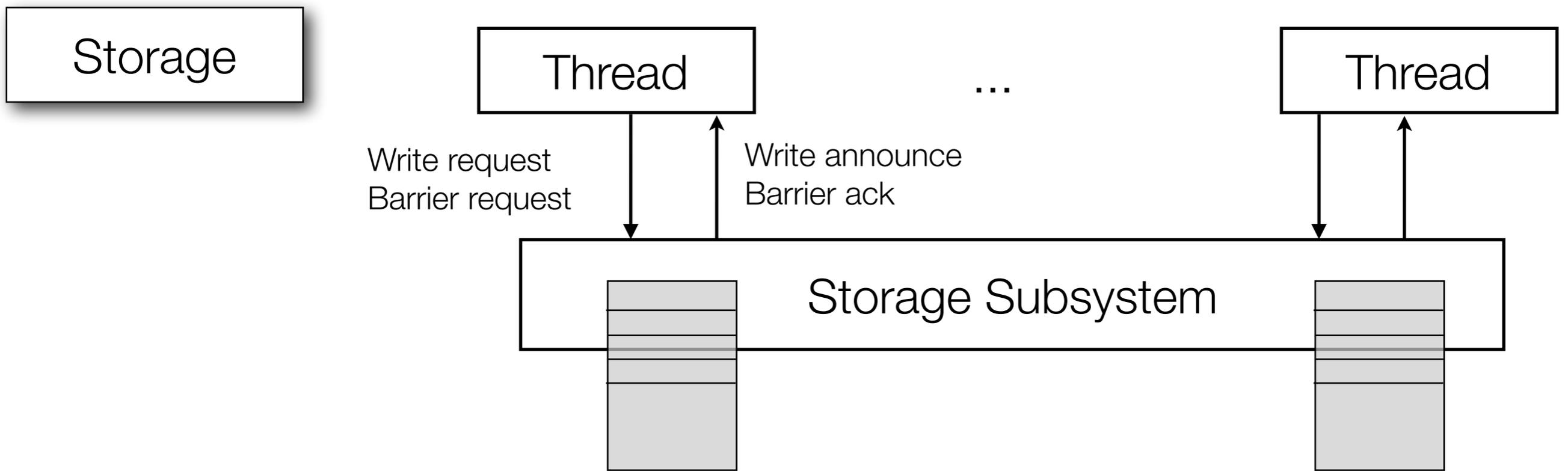
Each thread loads its code, instructions instances are initially marked *in-flight*.

In-flight instructions can be *committed*, not necessarily in program order.

When a branch is committed, the un-taken alternatives are discarded.

Instructions that follow an uncommitted branch cannot be committed.

In-flight instructions can be processed even before being committed (e.g. to speculate reads from memory, perform computation, ...).



The storage keeps (among other things):

1. for each thread, a list of the events propagated to the thread.

When receiving a write request, the storage adds the write event to the list of the events propagated to the thread who issued the request.

The storage can propagate an observed event to a thread list at any time
(unless barriers / coherence / ... conditions).

Threads can commit writes at any time
(unless dependency / synch / pending / ... conditions).

Storage

Thread

...

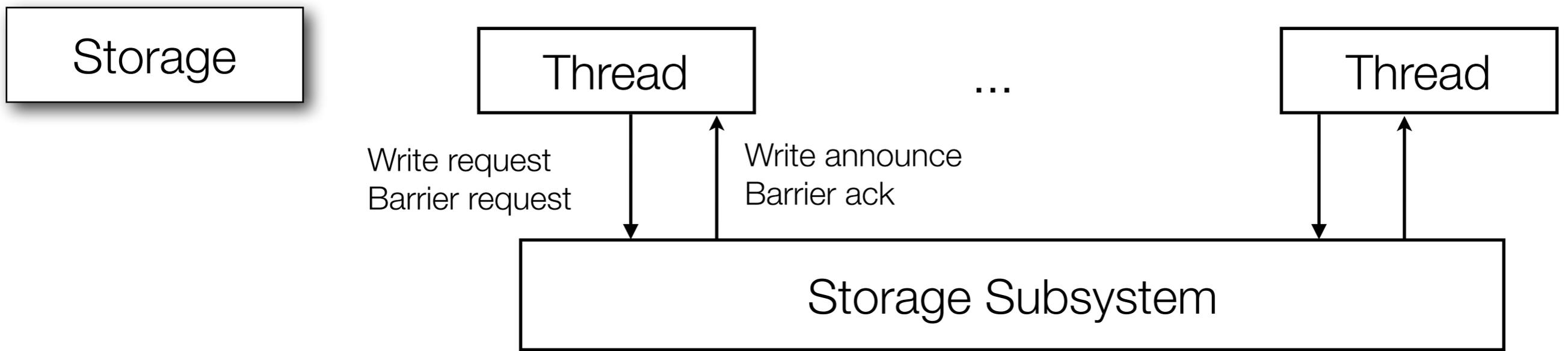
Thread

Simulation: 1. write_propagation

Thread 0	Thread 1	Thread 2
x = 1	x = 2	
y = 1		

The storage can propagate an observed event to a thread list at any time
(unless barriers / coherence / ... conditions).

Threads can commit writes at any time
(unless dependency / synch / pending / ... conditions).



The storage keeps: ...

2. for each location, a partial order of *coherence commitments*

Idea 1: at the end of the execution, writes to each location are totally ordered.

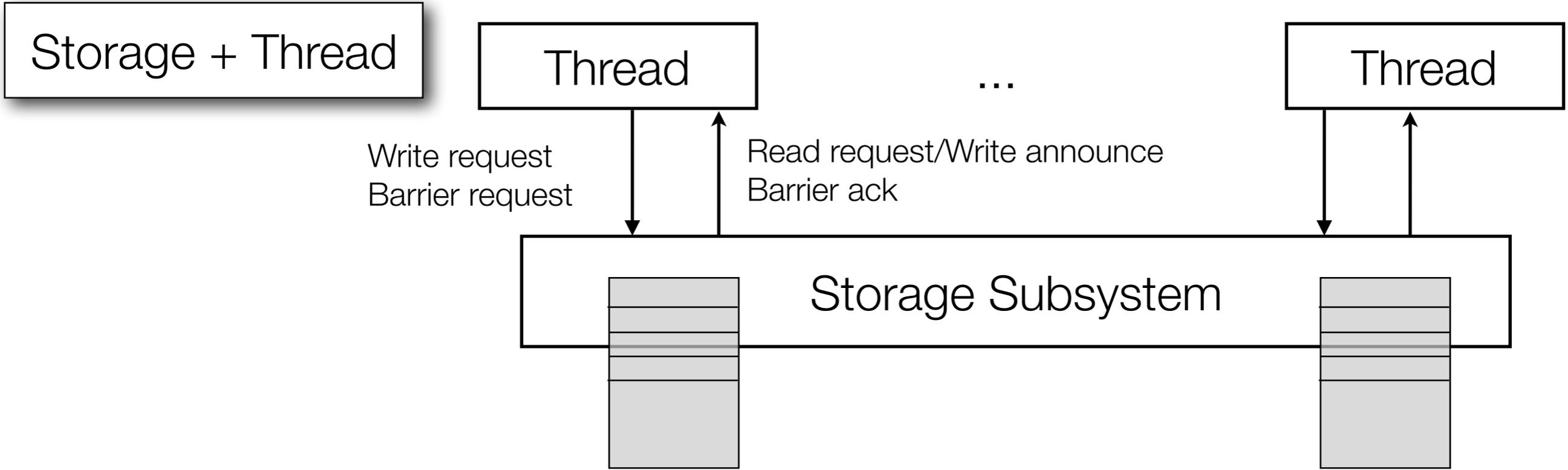
Idea 2: during computation, reads and propagation of writes must respect the coherence order (*reduce non-determinism of previous rules*).

Intuition: if a thread executes $x=1$ and then $x=2$, another thread cannot first read 2 and then 1.

Simulation: 2. coherence_propagation

Thread 0	Thread 1
$x = 1$	
$x = 2$	

Intuition: if a thread executes $x=1$ and then $x=2$, another thread cannot first read 2 and then 1.



Threads can issue *read-requests* at any time (*unless dependency / synch / ...*).

Issuing a read-request and committing a read are **different actions**.

Storage can accept a read-request by a thread at any time, and reply with the **most recent write** to the same address **that has been propagated** to the thread.

Remark: receiving a write-announce is not enough to commit a read instruction.

Write-announces can be invalidated, and read-requests can be re-issued.

Storage + Thread

Thread

...

Thread

Write request
Barrier request

Read request/Write announce
Barrier ack

Simulation: 3. read_satisfy

Thread 0	Thread 1
$x = 1$	$r = x$
$x = 2$	

Simulation: 4. invalidate_read

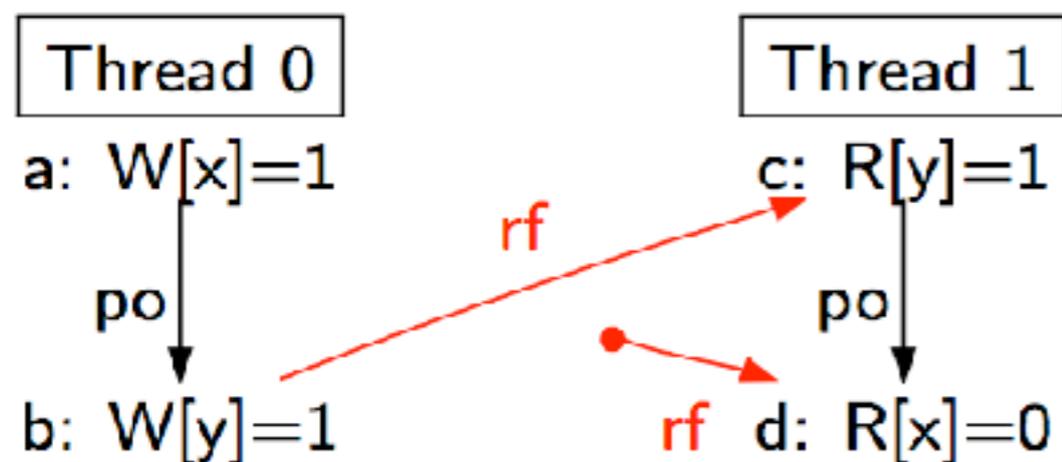
Thread 0	Thread 1
$x = 1$	$r1 = x$
	$r2 = x$

Remarks: loads can be speculated; difference between read/write transitions

Naïve message passing

Thread 0	Thread 1
<code>x = 1</code> <code>y = 1</code>	<code>while (y==0) {};</code> <code>r = x</code>

Observable behaviour: $r = 0$



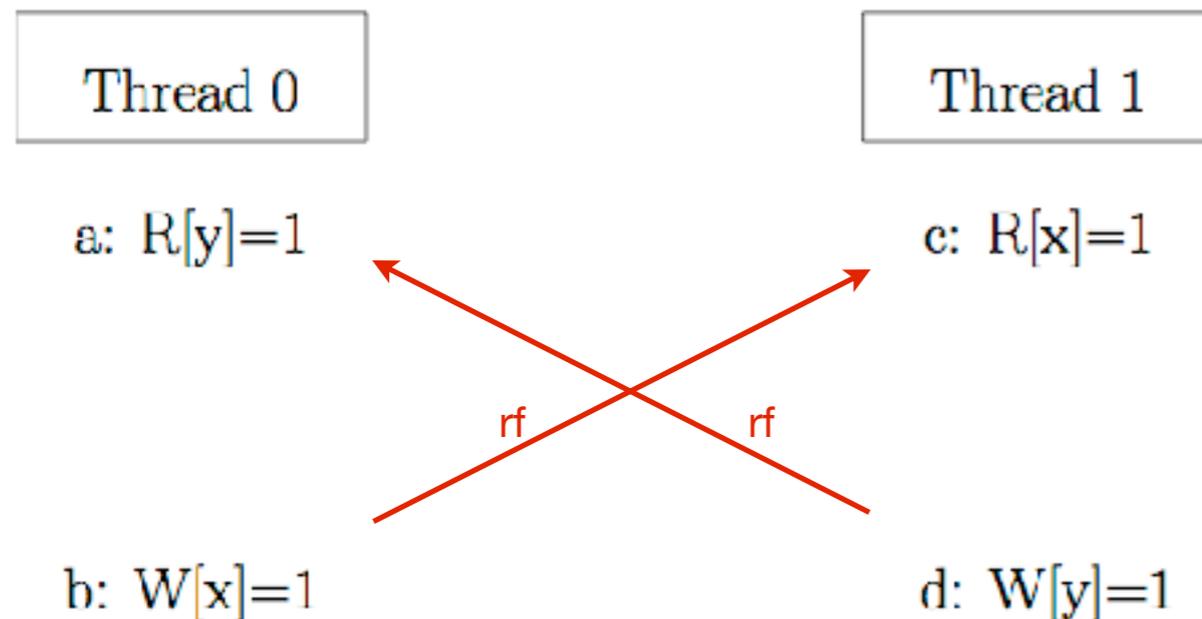
Allowed and observed on *Power*

Simulation: 5. message_passing

Load buffering

Thread 0	Thread 1
$r1 = x$ $y = 1$	$r2 = y$ $x = 1$

Observable behaviour: $r1 = r2 = 1$



Test LB (d1): Allowed (basic data)

Forbidden on SC and x86-TSO

Allowed and observed on Power

Simulation: 6. load_buffering

Power ISA 2.06 and ARM v7

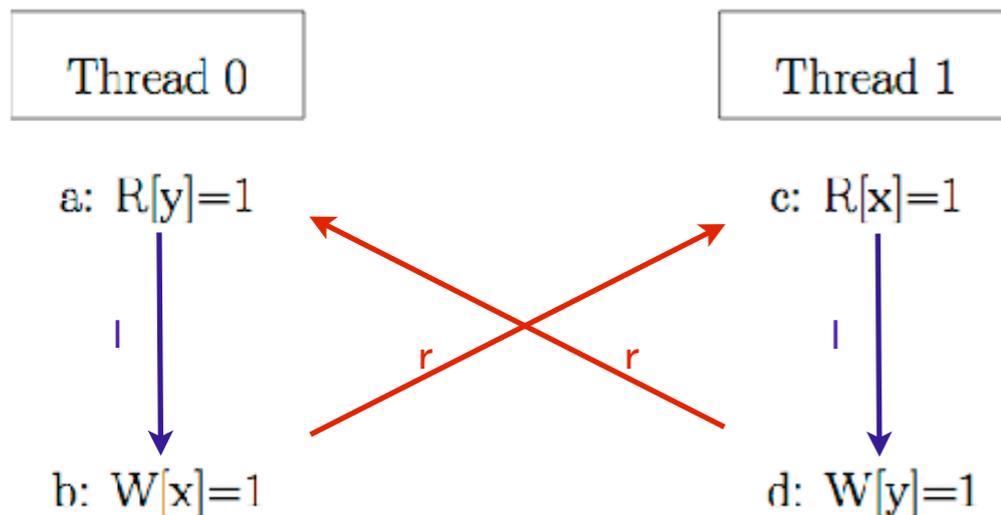
Visible behaviour much weaker and subtle than x86.

Basically, program order is **not preserved** unless:

- writes to the *same* memory location (coherence)
- there is an *address dependency* between two loads
data-flow path through registers and arith/logical operations from the value of the first load to the address of the second
- there is an *address or data or control dependency* between a load and a store
as above, or to the value store, or data flow to the test of an intermediate conditional branch
- you use a *synchronisation instruction* (ptesync, hwsync, lwsync, eieio, mbar, isync).

Load buffering with dependencies

LB+deps	ARM
Thread 0	Thread 1
LDR R2, [R5] AND R3, R2, #0 STR R1, [R3,R4]	LDR R2, [R4] AND R3, R2, #0 STR R1, [R3,R5]
Initial state: $0:R1=1 \wedge 0:R4=x \wedge 0:R5=y$ $\wedge 1:R1=1 \wedge 1:R4=x \wedge 1:R5=y$	
Forbidden: $0:R2=1 \wedge 1:R2=1$	



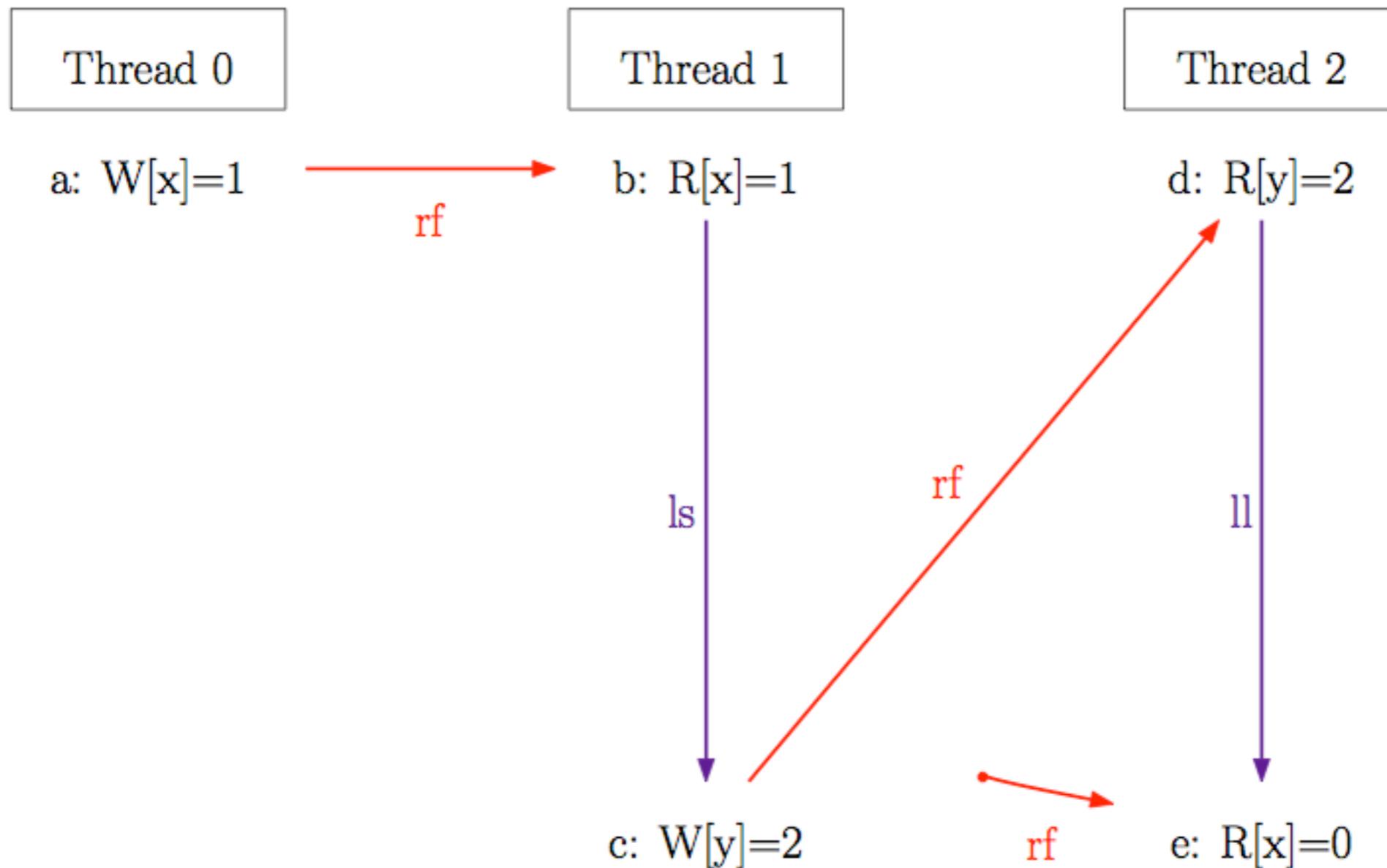
Test LB+deps (d5): Forbidden (basic data)

Simulation: 7. load_buffering_data_deps

Similarly with control dependencies, e.g.:

Play with examples in the LB directory

However dependencies might not be enough



Test WRC+deps (isa1v2): Allowed (basic data)

Exercise: WRC/WRC+addrs

Memory barrier

Power: ptesync, hv

ARM: DSB, **DMB**

For each a_i
will be pe
to the exte
quired att
cessor or

• A in
proc
resp

• B in
proc
instr

returned the value stored by a store that is in B .



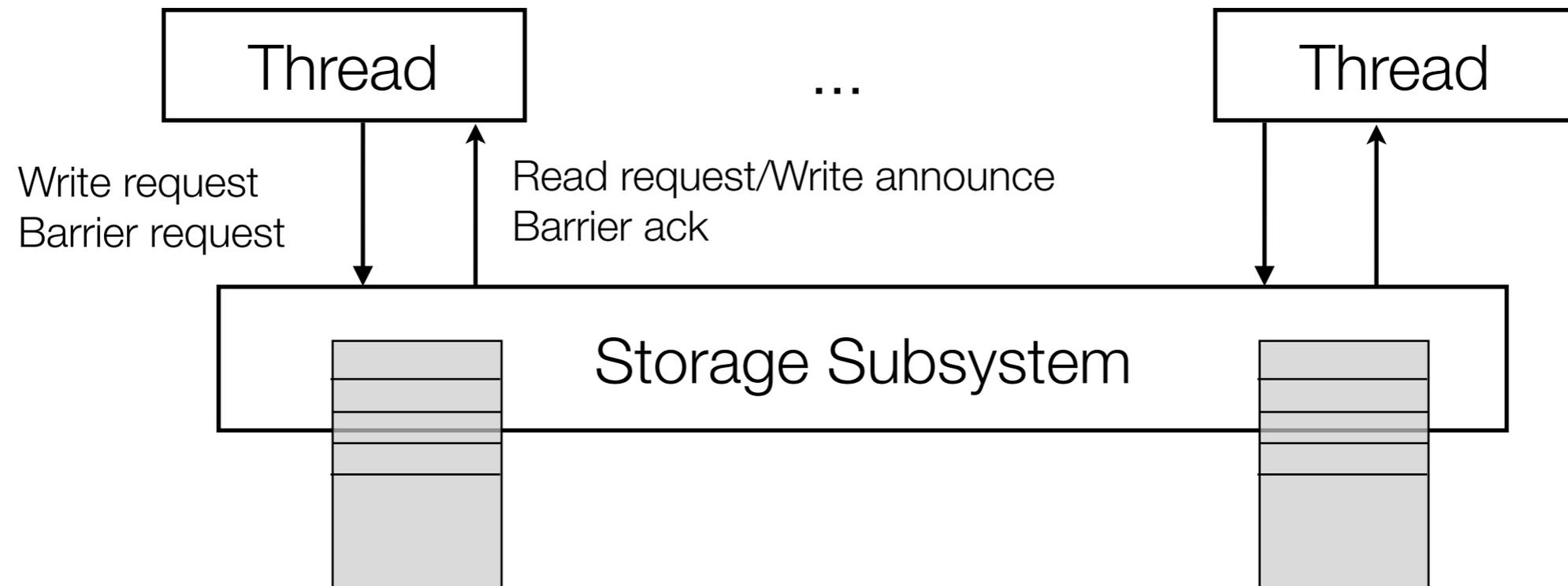
Caution
Mind your head

es that a_i
mechanism,
rence Re-
that pro-

y such
ed with
ed.

any such
er a *Load*
anism has

HWSYNC and LWSYNC



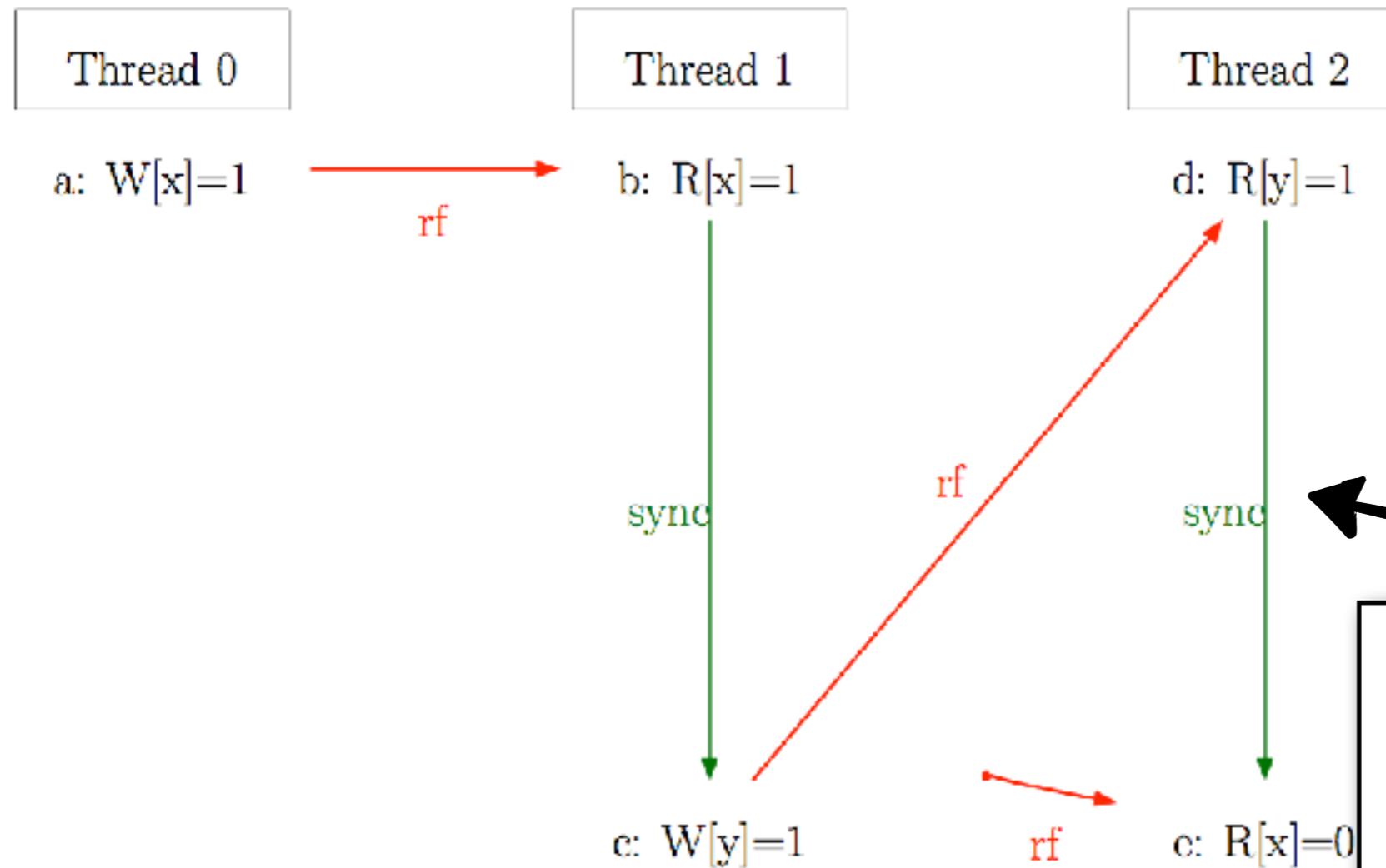
The storage accepts a barrier request (HWSYNC) from a thread.

The barrier request is added to the list of event propagated to that thread.

The thread cannot *execute* instructions following the barrier instructions without first receiving the barrier ack.

The storage sends the barrier ack only once all the preceding events have been propagated to all other threads.

RWC with HWSYNC



Test $WRC+syncs$ (m3s): Forbidden (basic data)

actually, a dependency here is enough...

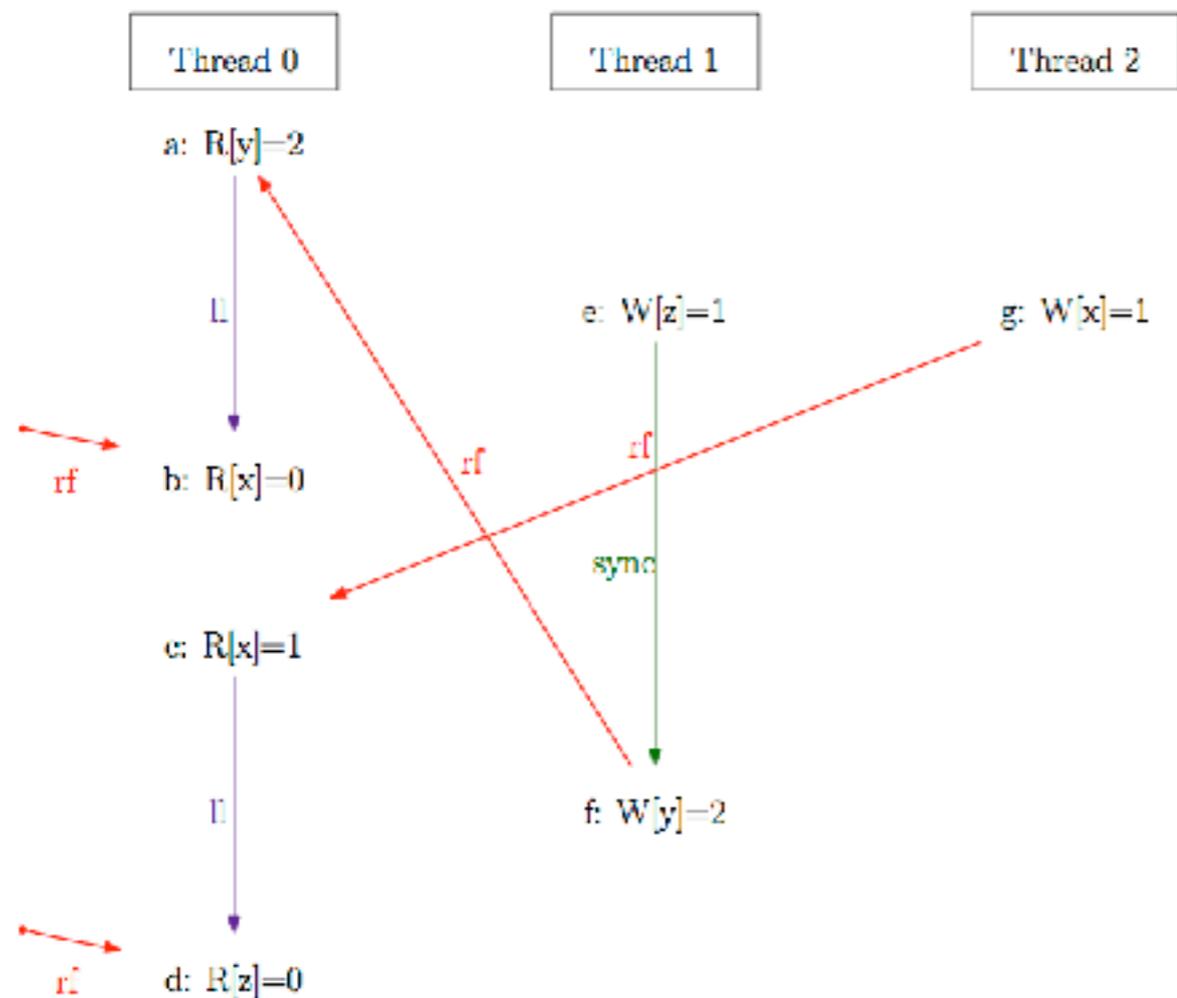
$WRC/WRC+sync+addr$

Simulation: $WRC/WRC+syncs$

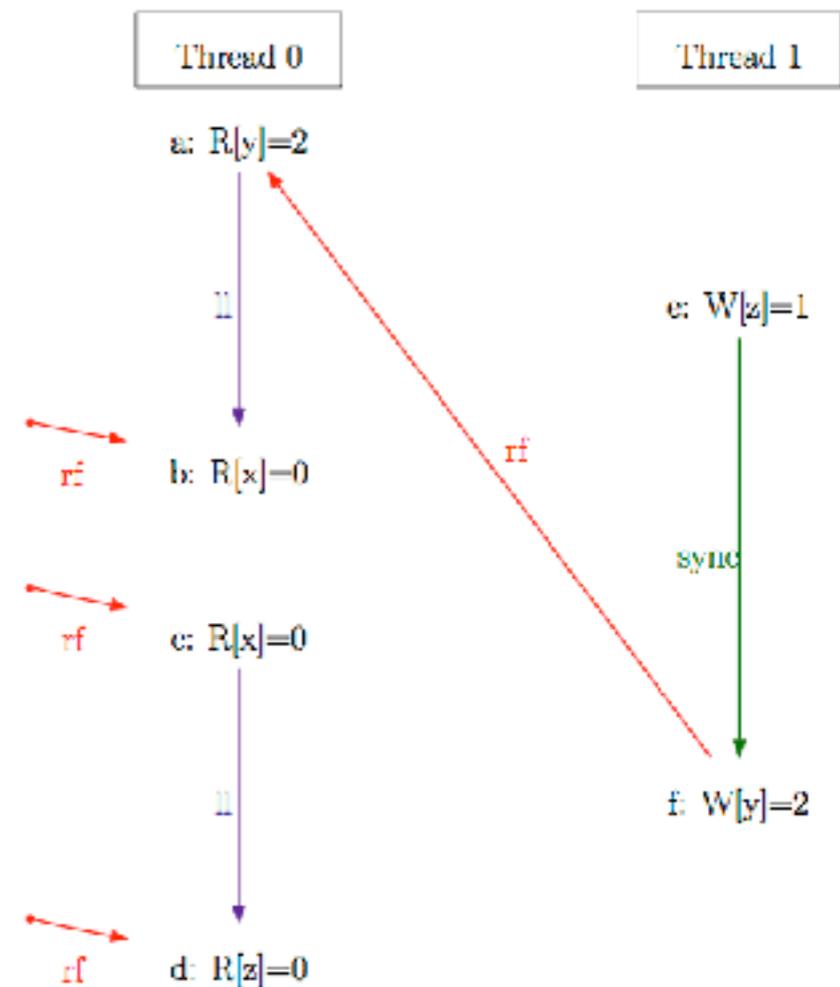
If you want more...

Go to <http://www.cl.cam.ac.uk/~pes20/ppcmem/>

For each test, either find a trace that leads to the final state, or convince yourself that such trace does not exist. *Some tests are complicated...*



Test RDW (ppo3): Forbidden (basic data)



Test RSW (ppo1): Allowed (basic data)



Summary

Concurrent programming
is hard!

programming
d!

Concurrent programming is *even harder* than
what I was taught at university!

We can't ignore it anymore:

we'll see that precise semantics, formal methods,
appropriate language design, clever algorithms,
are needed to put concurrent programming on solid basis.

2.37.1 Plan

- Hardware makes programming hard: let's use DSL/runtimes
 - Shared-memory parallel programming models and runtime systems
 - Data parallelism, programming and implementation, focus on OpenMP
 - Task parallelism, programming and implementation, focus on Cilk
 - Functional parallelism and asynchronous I/O with futures and streams
 - Dependent task parallelism, focus on OMPs, OpenMP, OpenStream
 - GPU programming models
- Cool, but what about general purpose languages?
 - Data-race freedom and Posix threads, locks, conditional barriers
 - Lock-free programming, CAS, the memory model and the programming practice
 - Axiomatic approach to memory models

Resources



<http://www.cl.cam.ac.uk/~pes20/weakmemory/index.html>

P. Sewell, S. Sarkar, S. Owens, F. Zappa Nardelli, M. Myreen

x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors

[Communications of the ACM](#), Vol. 53, 2010

S. Sarkar, P. Sewell, J. Alglave, L. Maranget, D. Williams

Understanding POWER multiprocessors

PLDI 2011

L. Maranget, S. Sarkar, P. Sewell

A tutorial introduction to the ARM and POWER relaxed memory model

Draft: <http://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf>