

# Multicore Semantics and Programming

Peter Sewell

Tim Harris

University of Cambridge

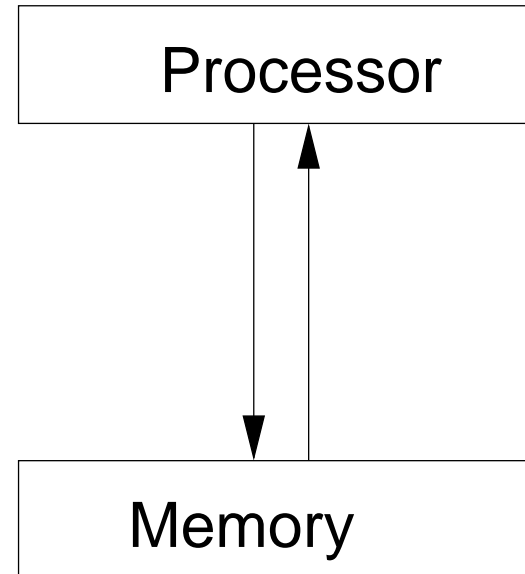
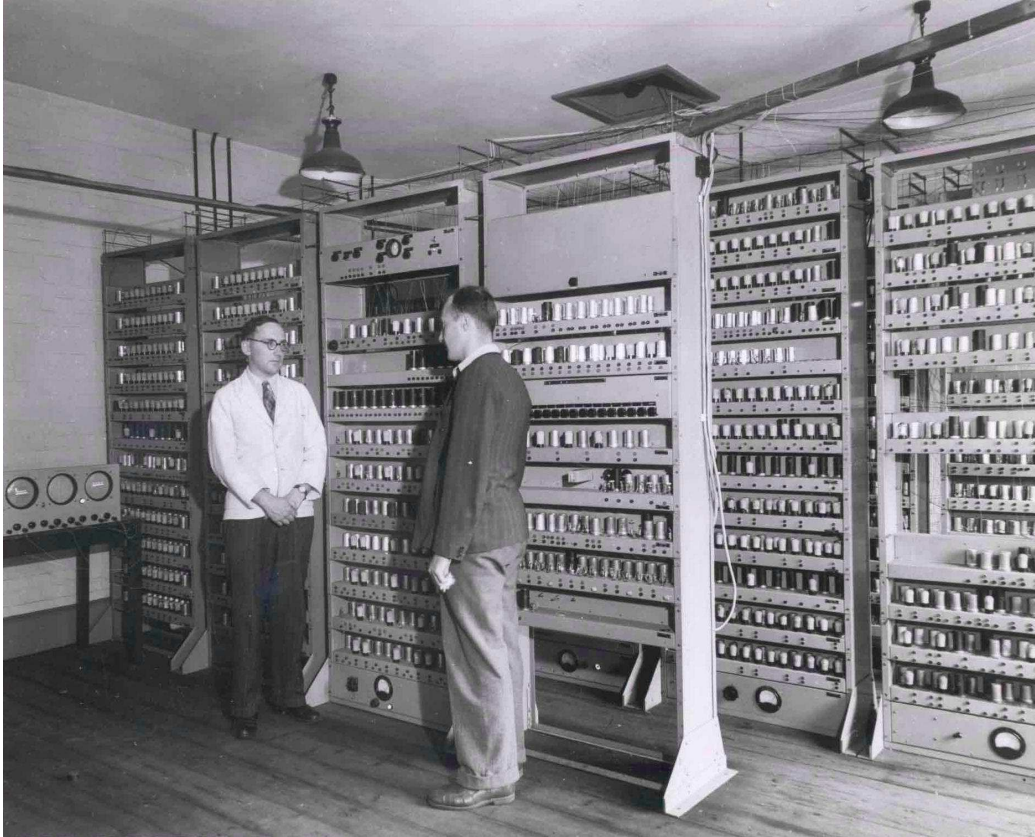
MSR

with thanks to

Francesco Zappa Nardelli, Jaroslav Ševčík, Susmit Sarkar, Tom Ridge,  
Scott Owens, Magnus O. Myreen, Luc Maranget,  
Mark Batty, Jade Alglave

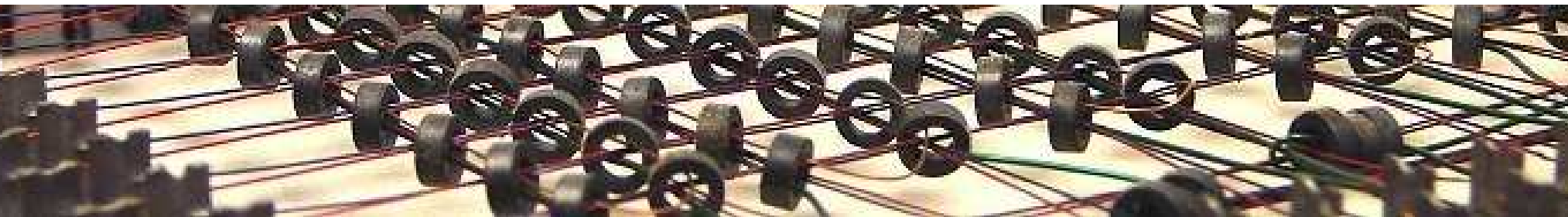
October – November, 2011

# The Golden Age, 1945–1959



# The Golden Age, 1945–1959

Memory = An array of values



# Multiprocessors, 1960–2011

Niche multiprocessors since before 1964 (UNIVAC 1108A)  
IBM System 370/158MP in 1972



Mass-market since 2005 (Intel Core 2 Duo).



Now: dual core ARM phones (HTC, Samsung, iPhone4S),  
12+ core x86 servers, 1024-thread Power 7 servers, Sparc,  
Itanium

# Multiprocessors, 1960–2011

Why now?

Exponential increases in transistor counts continuing — but not per-core performance

- energy efficiency (computation per Watt)
- limits of instruction-level parallelism

Concurrency finally mainstream — but how to understand and design concurrent systems?

# Concurrency Everywhere

In many forms and at many scales:

- intra-core
- GPU
- multicore (/manycore) systems
- datacenter-scale

explicit message-passing vs shared memory

# The First Bizarre Example

At the heart of a mutual exclusion algorithm, e.g. Dekker's, you might find code like this, say on an x86.

Two memory locations  $x$  and  $y$ , initially 0

Thread 0	Thread 1
MOV $[x] \leftarrow 1$ (write $x=1$ )	MOV $[y] \leftarrow 1$ (write $y=1$ )
MOV EAX $\leftarrow [y]$ (read $y$ )	MOV EBX $\leftarrow [x]$ (read $x$ )

What final states are allowed?

# The First Bizarre Example

At the heart of a mutual exclusion algorithm, e.g. Dekker's, you might find code like this, say on an x86.

Two memory locations  $x$  and  $y$ , initially 0

Thread 0	Thread 1
MOV $[x] \leftarrow 1$ (write $x=1$ )	MOV $[y] \leftarrow 1$ (write $y=1$ )
MOV EAX $\leftarrow [y]$ (read $y$ )	MOV EBX $\leftarrow [x]$ (read $x$ )

What final states are allowed?

...an example of *relaxed memory* behaviour



# These Lectures

Part 1: Concurrency in multiprocessors and programming languages (Peter Sewell and others)

Establish a solid basis for thinking about relaxed-memory executions, linking to usage, microarchitecture, experiment, and semantics.

Part 2: Concurrent algorithms (Tim Harris)

Concurrent programming: simple algorithms, correctness criteria, advanced synchronisation patterns, transactional memory.

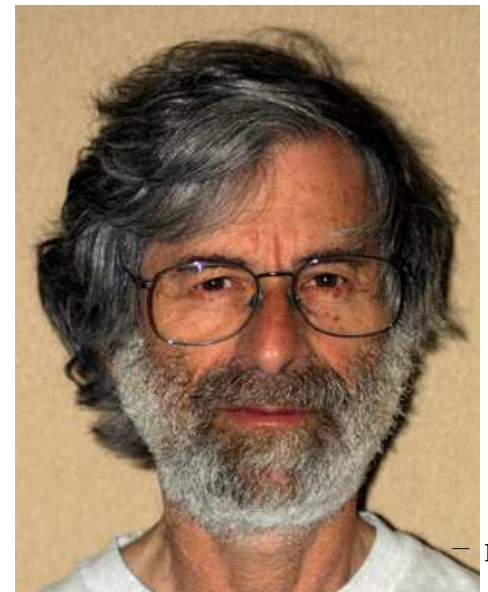
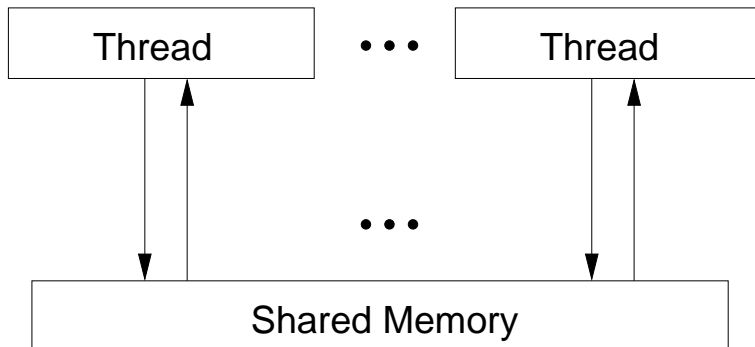
# Start with SC

# In an Ideal World

Multiprocessors would have *sequentially consistent (SC)* shared memory:

*“the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program”.*

Leslie Lamport, 1979



# A Tiny Language

*location, x, m*    address (or pointer value)

*integer, n*        integer

*thread\_id, t*      thread id

*k, i, j*

<i>expression, e</i>	::=	expression
		<i>n</i> integer literal
		<i>*x</i> read from pointer
		<i>*x = e</i> write to pointer
		<i>e; e'</i> sequential composition
		<i>e + e'</i> plus

<i>process, p</i>	::=	process
		<i>t:e</i> thread
		<i>p p'</i> parallel composition

# A Tiny Language

That was just the syntax — how can we be precise about the permitted behaviours of programs?

# An SC Semantics for that Language

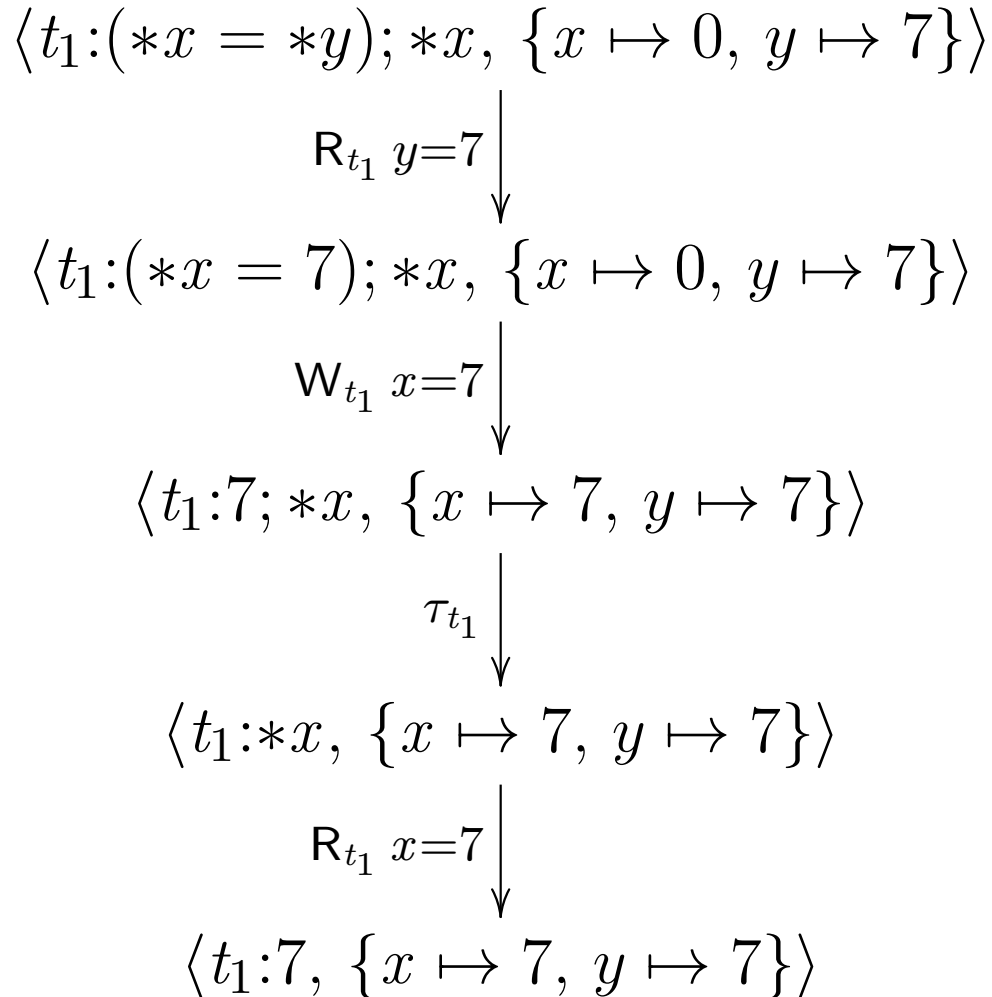
Take a *memory*  $M$  to be a function from addresses to integers, and a *state*  $\langle p, M \rangle$  to be a pair of a process and a memory.

$state, s$	$::=$	<b>state</b>
	$\langle p, M \rangle$	process $p$ and memory $M$
$thread\_label, l_t$	$::=$	<b>thread label</b>
	$W_t x=n$	write
	$R_t x=n$	read
	$\tau_t$	internal action (tau)

We'll define a transition relation  $\langle p, M \rangle \xrightarrow{l_t} \langle p', M' \rangle$

# Example: SC Whole-System Trace

For a thread  $t_1:(*x = *y); *x$  starting in memory  $\{x \mapsto 0, y \mapsto 7\}$ :



# Defining an SC Semantics — threads

What can a thread do in isolation? (without prescribing how the memory behaves)

<i>label, l</i>	$::=$	<b>label</b>
		$W\ x=n$ write
		$R\ x=n$ read
		$\tau$ internal action (tau)



# Defining an SC Semantics: expressions (1)

$e \xrightarrow{l} e'$   $e$  does  $l$  to become  $e'$

$$\frac{}{*x \xrightarrow{R\ x=n} n}$$
 READ

$$\frac{}{*x = n \xrightarrow{W\ x=n} n}$$
 WRITE

$$\frac{e \xrightarrow{l} e'}{*x = e \xrightarrow{l} *x = e'}$$
 WRITE\_CONTEXT

$$\frac{}{n; e \xrightarrow{\tau} e}$$
 SEQ

$$\frac{e_1 \xrightarrow{l} e'_1}{e_1; e_2 \xrightarrow{l} e'_1; e_2}$$
 SEQ\_CONTEXT

# Defining an SC Semantics: expressions (2)

$e \xrightarrow{l} e'$   $e$  does  $l$  to become  $e'$

$$\frac{e_1 \xrightarrow{l} e'_1}{e_1 + e_2 \xrightarrow{l} e'_1 + e_2} \quad \text{PLUS\_CONTEXT\_1}$$

$$\frac{e_2 \xrightarrow{l} e'_2}{n_1 + e_2 \xrightarrow{l} n_1 + e'_2} \quad \text{PLUS\_CONTEXT\_2}$$

$$\frac{n = n_1 + n_2}{n_1 + n_2 \xrightarrow{\tau} n} \quad \text{PLUS}$$

# Example: SC Expression Trace

$(*x = *y); *x$

# Example: SC Expression Trace

$(*x = *y); *x$

$(*x = *y); *x \xrightarrow{R\ y=7} \xrightarrow{W\ x=7} \xrightarrow{\tau} \xrightarrow{R\ x=9} 9$

# Example: SC Expression Trace

$(*x = *y); *x$

$(*x = *y); *x \xrightarrow{Ry=7} \xrightarrow{Wx=7} \xrightarrow{\tau} \xrightarrow{Rx=9} 9$

$*y \xrightarrow{Ry=7} 7$	READ	
$*x = *y \xrightarrow{Ry=7} *x = 7$	WRITE	
$(*x = *y); *x \xrightarrow{Ry=7} (*x = 7); *x$	SEQ_CONTEXT	

# Example: SC Expression Trace

$(*x = *y); *x$

$(*x = *y); *x \xrightarrow{R y=7} \xrightarrow{W x=7} \xrightarrow{\tau} \xrightarrow{R x=9} 9$

$\frac{\frac{}{*x = 7} \xrightarrow{W x=7} 7}{(*)x = 7); *x \xrightarrow{W x=7} 7; *x}$	<b>WRITE</b>	<b>SEQ_CONTEXT</b>
---	--------------	--------------------

# Example: SC Expression Trace

$(*x = *y); *x$

$(*x = *y); *x \xrightarrow{R y=7} \xrightarrow{W x=7} \xrightarrow{\tau} \xrightarrow{R x=9} 9$

$\frac{}{7; *x \xrightarrow{\tau} *x}$  SEQ

$\frac{}{*x \xrightarrow{R x=9} 9}$  READ

# Defining an SC Semantics: lifting to processes

$p \xrightarrow{l_t} p'$   $p$  does  $l_t$  to become  $p'$

$$\frac{e \xrightarrow{l} e'}{t:e \xrightarrow{l_t} t:e'} \quad \text{THREAD}$$
$$\frac{p_1 \xrightarrow{l_t} p'_1}{p_1|p_2 \xrightarrow{l_t} p'_1|p_2} \quad \text{PAR\_CONTEXT\_LEFT}$$
$$\frac{p_2 \xrightarrow{l_t} p'_2}{p_1|p_2 \xrightarrow{l_t} p_1|p'_2} \quad \text{PAR\_CONTEXT\_RIGHT}$$

free interleaving



# Defining an SC Semantics: SC memory

$M \xrightarrow{l} M'$   $M$  does  $l$  to become  $M'$

$$\frac{M(x) = n}{M \xrightarrow{R\ x=n} M} \quad \text{MREAD}$$

$$\frac{}{M \xrightarrow{W\ x=n} M \oplus (x \mapsto n)} \quad \text{MWRITE}$$

# Defining an SC Semantics: whole-system states

$$\boxed{s \xrightarrow{l_t} s'} \quad s \text{ does } l_t \text{ to become } s'$$

$$\frac{\begin{array}{l} p \xrightarrow{R_t x=n} p' \\ M \xrightarrow{R x=n} M' \end{array}}{\langle p, M \rangle \xrightarrow{R_t x=n} \langle p', M' \rangle} \quad \text{SREAD}$$

$$\frac{\begin{array}{l} p \xrightarrow{W_t x=n} p' \\ M \xrightarrow{W x=n} M' \end{array}}{\langle p, M \rangle \xrightarrow{W_t x=n} \langle p', M' \rangle} \quad \text{SWRITE}$$

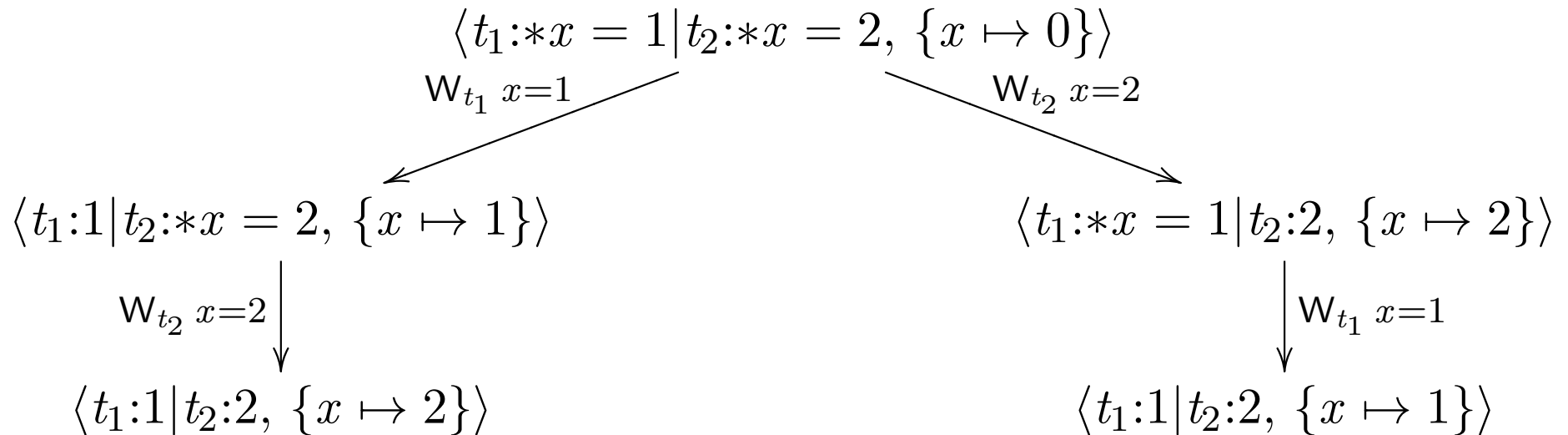
$$\frac{p \xrightarrow{\tau_t} p'}{\langle p, M \rangle \xrightarrow{\tau_t} \langle p', M \rangle} \quad \text{STAU}$$

synchronising between the process and the memory

# Example: SC Interleaving

All threads can read and write the shared memory.

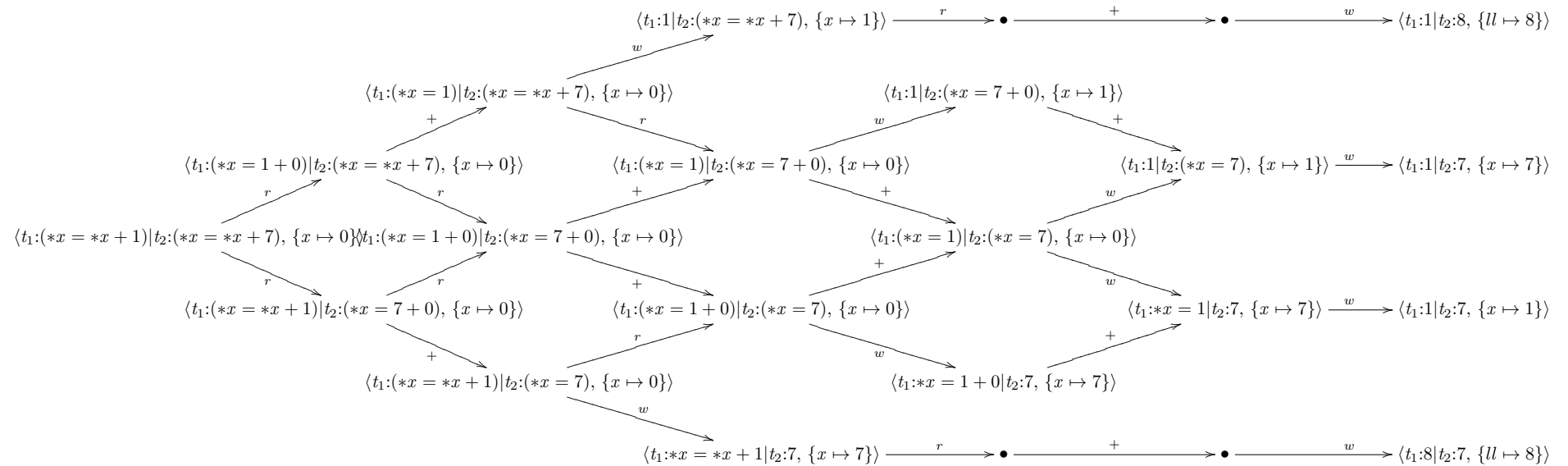
Threads execute asynchronously – the semantics allows any interleaving of the thread transitions. Here there are two:



But each interleaving has a linear order of reads and writes to the memory.

# Combinatorial Explosion

The behaviour of  $t_1: *x = *x + 1 \mid t_2: *x = *x + 7$  for the initial store  $\{x \mapsto 0\}$ :



NB: the labels  $+$ ,  $w$  and  $r$  in this picture are just informal hints as to how those transitions were derived

# Morals

- For free interleaving, number of systems states scales as  $n^t$ , where  $n$  is the threads per state and  $t$  the number of threads.
- Drawing state-space diagrams only works for really tiny examples – we need better techniques for analysis.
- Almost certainly you (as the programmer) didn't want all those 3 outcomes to be possible – need better idioms or constructs for programming.

# Mutual Exclusion

For “simple” concurrency, need some way(s) to synchronise between threads, so can enforce *mutual exclusion* for shared data.

- can code up mutual exclusion library using reads and writes
- but usually, at the language or OS level, there’s built-in support from the scheduler, eg for *mutexes* and *condition variables*
- and at the hardware level, various primitives that we’ll get back to

See this – in the library – for a good discussion of mutexes and condition variables: A. Birrell, J. Guttag, J. Horning, and R. Levin. *Thread synchronization: a Formal Specification*. In *System Programming with Modula-3*, chapter 5, pages 119-129. Prentice-Hall, 1991.

See Herlihy and Shavit’s text, and N. Lynch *Distributed Algorithms*, for many algorithms (and much more).

# Adding Primitive Mutexes

Expressions  $e ::= \dots \mid \text{lock } x \mid \text{unlock } x$

Say lock free if it holds 1, taken otherwise.

Don't mix locations used as locks and other locations.

Semantics (outline):  $\text{lock } x$  has to *atomically* (a) check the mutex is currently free, (b) change its state to taken, and (c) let the thread proceed.

$\text{unlock } x$  has to change its state to free.

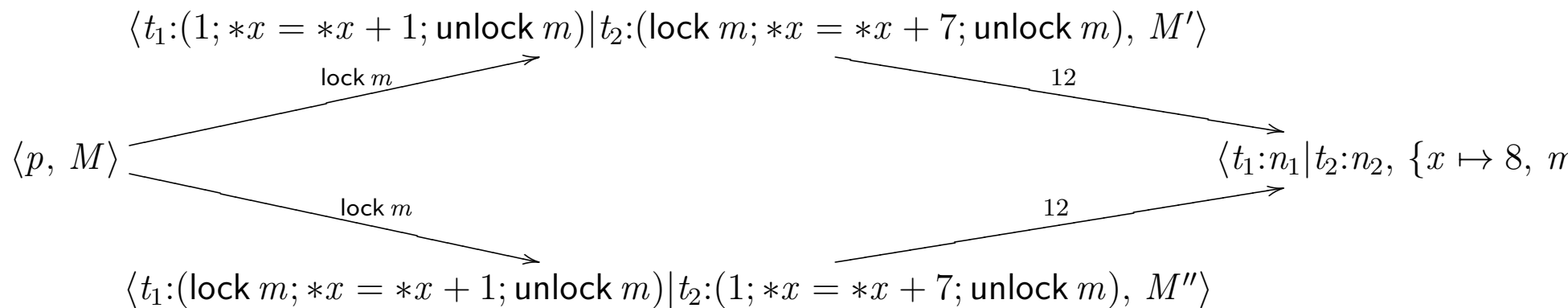
Record of which thread is holding a locked lock? Re-entrancy?

# Using a Mutex

Consider  $p =$

$t_1:(\text{lock } m; *x = *x + 1; \text{unlock } m) | t_2:(\text{lock } m; *x = *x + 7; \text{unlock } m)$

in the initial store  $M = \{x \mapsto 0, m \mapsto 1\}$ :





# Deadlock

lock  $m$  can block (that's the point). Hence, you can *deadlock*.

$$p = \begin{array}{l} t_1: (\text{lock } m_1; \text{lock } m_2; *x = 1; \text{unlock } m_1; \text{unlock } m_2) \\ | \\ t_2: (\text{lock } m_2; \text{lock } m_1; *x = 2; \text{unlock } m_1; \text{unlock } m_2) \end{array}$$

# Locking Disciplines

Suppose we have several programs  $e_1, \dots, e_k$  that we want to execute concurrently without ‘interference’ (whatever that is). You might think of them as transaction bodies.

There are many possible locking disciplines. We’ll describe one, to see how it – and the properties it guarantees – can be made precise and proved.

# An Ordered 2PL Discipline, (too!) Informally

Fix an association between locations and mutexes. For simplicity, make it 1:1 – associate  $x$  with  $m$ ,  $x_1$  with  $m_1$ , etc.  
Fix a lock acquisition order. For simplicity, make it  $m, m_0, m_1,$

...

Require that each  $e_i$

- acquires the lock  $m_j$  for each location  $x_j$  it uses, before it uses it
- acquires and releases each lock in a properly-bracketed way
- does not acquire any lock after it's released any lock (two-phase)
- acquires locks in increasing order

Then, informally,  $\langle t_1:e_1 | \dots | t_k:e_k, M \rangle$  should (a) never deadlock, and (b) be *serialisable* – any execution should be ‘equivalent’ to an execution of  $e_{\pi(1)}; \dots; e_{\pi(k)}$  for some permutation  $\pi$ .

# Now can make the Ordered 2PL Discipline precise

Say  $p$  obeys the discipline if for any (finite or infinite) sequence of transitions, then ...

... and make the guaranteed properties precise

Say  $e_1, \dots, e_k$  are *serialisable* if for any initial memory  $M$ , if  $\langle t_1:e_1 | \dots | t_k:e_k, M \rangle \rightarrow^* \langle t_1:e'_1 | \dots | t_k:e'_k, M' \rangle \neg \rightarrow$  then for some permutation  $\pi$  we have  $\langle t:e_1; \dots; e_k, M \rangle \rightarrow^* \langle t:e', M' \rangle \neg \rightarrow$ .

Say they are *deadlock-free* if ...?

(Warning: there are many subtle variations of these properties!)

# The Theorem

**Conjecture 1** *If each  $e_i$  obeys the discipline, then  $e_1, \dots, e_k$  are serialisable and deadlock-free.*

# Atomicity again

In this toy language, assignments and dereferencing are *atomic*. For example,

$\langle t_1: *x = 3498734590879238429384 \mid t_2: *x = 7, \{x \mapsto 0\} \rangle$

will reduce to a state with  $x$  either 3498734590879238429384 or 7, not something with the first word of one and the second word of the other. Implement?

But in  $t_1: (*x = e) \mid t_2: e'$ , the steps of evaluating  $e$  and  $e'$  can be interleaved.

And the 2PL discipline is enlarging the granularity of atomic reads and writes.

(will come back to this for hardware and real programming languages)

# Data Races

another way to look at 2PL is as a means to exclude *data races*

(and if you've done so, the exact level of atomicity doesn't matter)



# Using Locks

Large-scale lock ordering?

Compositionality?

# Fairness

We've not discussed *fairness* – the semantics allows any interleaving between parallel components, not only fair ones.

Imagine extending the language with conditionals and while loops, and consider

$t_1 : *x = 1 \mid t_2 : \text{while true do } \dots$

starting with  $\{x \mapsto 0\}$ .

# Message Passing

Not everything is mutual exclusion:

$*x_1 = 10;$		$\text{while } 0 == *x \text{ do } ();$
$*x_2 = 20;$		$*x_1 + *x_2$
$*x = 1$		

This is one-shot message passing — a step towards the producer-consumer problems... (c.f. Herlihy and Shavit)

# Recall We're In An Ideal World

Sequentially consistent shared memory has been taken for granted, by almost all:

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

# False, since 1972

## IBM System 370/158MP



Mass-market since 2005.



# The First Bizarre Example

At the heart of a mutual exclusion algorithm, e.g. Dekker's, you might find code like this, say on an x86.

Two memory locations  $x$  and  $y$ , initially 0

Thread 0	Thread 1
MOV $[x] \leftarrow 1$ (write $x=1$ )	MOV $[y] \leftarrow 1$ (write $y=1$ )
MOV EAX $\leftarrow [y]$ (read $y$ )	MOV EBX $\leftarrow [x]$ (read $x$ )

What final states are allowed?

What are the possible sequential orders?

# The First Bizarre Example

At the heart of a mutual exclusion algorithm, e.g. Dekker's, you might find code like this, say on an x86.

Two memory locations  $x$  and  $y$ , initially 0

Thread 0	Thread 1
MOV $[x] \leftarrow 1$ (write $x=1$ ) MOV EAX $\leftarrow [y]$ (read $y=0$ )	MOV $[y] \leftarrow 1$ (write $y=1$ ) MOV EBX $\leftarrow [x]$ (read $x=1$ )

Thread 0:EAX = 0

Thread 1:EBX=1

# The First Bizarre Example

At the heart of a mutual exclusion algorithm, e.g. Dekker's, you might find code like this, say on an x86.

Two memory locations  $x$  and  $y$ , initially 0

Thread 0	Thread 1
MOV $[x] \leftarrow 1$ (write $x=1$ )	MOV $[y] \leftarrow 1$ (write $y=1$ )
MOV EAX $\leftarrow [y]$ (read $y=1$ )	MOV EBX $\leftarrow [x]$ (read $x=1$ )

Thread 0: EAX = 1

Thread 1: EBX = 1



# The First Bizarre Example

At the heart of a mutual exclusion algorithm, e.g. Dekker's, you might find code like this, say on an x86.

Two memory locations  $x$  and  $y$ , initially 0

Thread 0	Thread 1
MOV $[x] \leftarrow 1$ (write $x=1$ )	MOV $[y] \leftarrow 1$ (write $y=1$ )
MOV EAX $\leftarrow [y]$ (read $y=1$ )	MOV EBX $\leftarrow [x]$ (read $x=1$ )

Thread 0: EAX = 1

Thread 1: EBX = 1

# The First Bizarre Example

At the heart of a mutual exclusion algorithm, e.g. Dekker's, you might find code like this, say on an x86.

Two memory locations  $x$  and  $y$ , initially 0

Thread 0	Thread 1
MOV $[x] \leftarrow 1$ (write $x=1$ )	MOV $[y] \leftarrow 1$ (write $y=1$ )
MOV EAX $\leftarrow [y]$ (read $y=1$ )	MOV EBX $\leftarrow [x]$ (read $x=1$ )

Thread 0: EAX = 1

Thread 1: EBX = 1

# The First Bizarre Example

At the heart of a mutual exclusion algorithm, e.g. Dekker's, you might find code like this, say on an x86.

Two memory locations  $x$  and  $y$ , initially 0

Thread 0	Thread 1
MOV $[x] \leftarrow 1$ (write $x=1$ )	MOV $[y] \leftarrow 1$ (write $y=1$ )
MOV EAX $\leftarrow [y]$ (read $y=1$ )	MOV EBX $\leftarrow [x]$ (read $x=1$ )

Thread 0: EAX = 1

Thread 1: EBX = 1

# The First Bizarre Example

At the heart of a mutual exclusion algorithm, e.g. Dekker's, you might find code like this, say on an x86.

Two memory locations  $x$  and  $y$ , initially 0

Thread 0	Thread 1
	MOV $[y] \leftarrow 1$ (write $y=1$ )
	MOV EBX $\leftarrow [x]$ (read $x=0$ )
MOV $[x] \leftarrow 1$ (write $x=1$ )	
MOV EAX $\leftarrow [y]$ (read $y=1$ )	

Thread 0:EAX = 1

Thread 1:EBX=0

# The First Bizarre Example

Conclusion:

0,1 and 1,1 and 1,0 can happen, but 0,0 is impossible

# The First Bizarre Example

Conclusion:

0,1 and 1,1 and 1,0 can happen, but 0,0 is impossible

In fact, in the real world:

we observe 0,0 every 630/100000 runs  
(on an Intel Core Duo x86)

(and so Dekker's algorithm will fail)



# Simple Compiler Optimisation Example

In SC, message passing should work as expected:

Thread 1	Thread 2
<code>data = 1</code> <code>ready = 1</code>	<code>if (ready == 1)</code> <code>    print data</code>

In SC, the program should only print 1.

# Simple Compiler Optimisation Example

Thread 1	Thread 2
<code>data = 1</code>	<code>int r1 = data</code>
<code>ready = 1</code>	<code>if (ready == 1)</code>
	<code>    print data</code>

In SC, the program should only print 1.

Regardless of **other reads**.



# Simple Compiler Optimisation Example

Thread 1	Thread 2
<code>data = 1</code> <code>ready = 1</code>	<code>int r1 = data</code> <code>if (ready == 1)</code> <code>    print data</code>

In SC, the program should only print 1.

But **common subexpression elimination** (as in `gcc -O1` and HotSpot) will **rewrite**

`print data`       $\implies$       `print r1`

# Simple Compiler Optimisation Example

Thread 1	Thread 2
<code>data = 1</code> <code>ready = 1</code>	<code>int r1 = data</code> <code>if (ready == 1)</code> <code>    print r1</code>

In SC, the program should only print 1.

But **common subexpression elimination** (as in `gcc -O1` and HotSpot) will **rewrite**

`print data`       $\implies$       `print r1`

So the **compiled program can print 0**

# Weakly Consistent Memory

Multiprocessors and compilers incorporate many performance optimisations

(hierarchies of cache, load and store buffers, speculative execution, cache protocols, common subexpression elimination, etc., etc.)

These are:

- unobservable by single-threaded code
- sometimes observable by concurrent code

Upshot: they provide only various *relaxed* (or *weakly consistent*) memory models, not sequentially consistent memory.

# What About the Specs?

Hardware manufacturers document *architectures*:

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

AMD64 Architecture Programmer's Manual

Power ISA specification

ARM Architecture Reference Manual

and programming languages (at best) are defined by *standards*:

ISO/IEC 9899:1999 Programming languages – C

J2SE 5.0 (September 30, 2004)

- loose specifications,
- claimed to cover a wide range of past and future implementations.

# What About the Specs?

Hardware manufacturers document *architectures*:

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

AMD64 Architecture Programmer's Manual

Power ISA specification

ARM Architecture Reference Manual

and programming languages (at best) are defined by *standards*:

ISO/IEC 9899:1999 Programming languages – C

J2SE 5.0 (September 30, 2004)

- loose specifications,
- claimed to cover a wide range of past and future implementations.

Flawed. Always confusing, sometimes wrong.

# What About the Specs?

*“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

# Hardware Models

## x86 in detail

Why are industrial specs so often flawed?

A usable model: x86-TSO

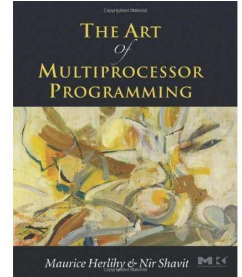
Reasoning about x86-TSO code: races

## Power/ARM

SPARC, Alpha, Itanium

## Programming Language Models (Java/C++)

# Uses



1. how to code low-level concurrent datastructures
2. how to build concurrency testing and verification tools
3. how to specify and test multiprocessors
4. how to design and express high-level language definitions
5. ... and in general, as an example of mathematically rigorous computer engineering



# Hardware Models

## x86 in detail

Why are industrial specs so often flawed?

A usable model: x86-TSO

Reasoning about x86-TSO code: races

Power/ARM

SPARC, Alpha, Itanium

Programming Language Models (Java/C++)

# 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, Nov. 2006, vol 3a, 10-5)

# A Cautionary Tale

Intel 64/IA32 and AMD64 - before August 2007 (Era of Vagueness)

A model called *Processor Ordering*, informal prose

Example: Linux Kernel mailing list, 20 Nov 1999 - 7 Dec 1999 (143 posts)

Keywords: speculation, ordering, cache, retire, causality

A one-instruction programming question, a microarchitectural debate!

## 1. spin\_unlock() Optimization On Intel

20 Nov 1999 - 7 Dec 1999 (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; btr1 \$0,%0" a to 1 tick for a simple "movl \$0,%0" instruction, a huge gain. he reported that Ingo Molnar noticed a 4% speed-up in mark test, making the optimization very valuable. added that the same optimization cropped up in the mailing list a few days previously. But Linus Torvalds poured water on the whole thing, saying:

**It does NOT WORK!**

**Let the FreeBSD people use it, and let them get timings. They will crash, eventually.**

**The window may be small, but if you do this, then suddenly spinlocks aren't reliable any more.**

*Resolved only by appeal to an oracle:*

that the pipelines are no longer invalid and the pipeline should be blown out.

I have seen the behavior Linus describes on a hardware analyzer, BUT ONLY ON SYSTEMS WERE PPRO AND ABOVE. I guess the BSD port must still be on older Pentium hardware and that they don't know this can bite in some cases.

Erich Boleyn, an Architect in an IA32 development group, also replied to Linus, pointing out a possible misconception in his proposed exploit. Regarding the code Linus posted, he replied:

It will always return 0. You don't need "spin\_lock()" to be serializing.

The only thing you need is to make sure there's a store in "spin\_unlock()", and that is kind of true because of the fact that you're changing something to be observable on other processors.

The reason for this is that stores can only be observed when all prior instructions have completed (i.e. the store is not sent outside of the processor until it is committed state, and the earlier instructions are already committed by that time), so the any stores, etc absolutely have to have completed before a cache-miss or not.

He went on:

Since the instructions for the store in the spin\_unlock()

# IWP and AMD64, Aug. 2007/Oct. 2008 (Era of Causality)

Intel published a white paper (IWP) 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 tests* illustrating allowed or forbidden behaviours, e.g.

## Message Passing (MP)

Thread 0	Thread 1
MOV [x]←1 (write x=1)	MOV EAX←[y] (read y=1)
MOV [y]←1 (write y=1)	MOV EBX←[x] (read x=0)
Forbidden Final State: Thread 1:EAX=1 ∧ Thread 1:EBX=0	

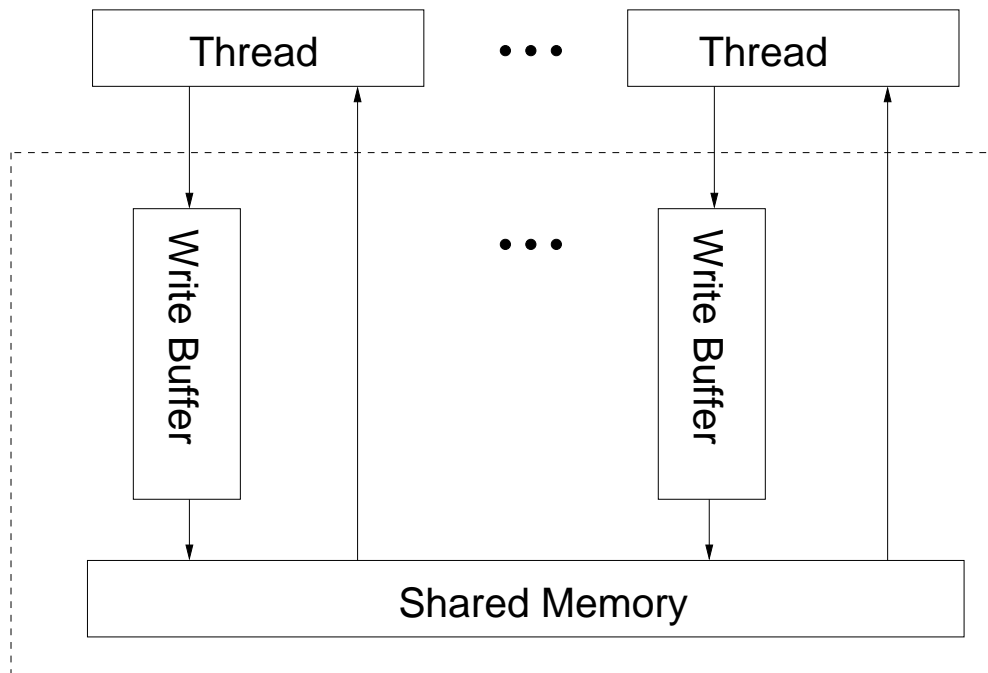
P3. Loads may be reordered with older stores to different locations but not with older stores to the same location

Thread 0	Thread 1
MOV [x] ← 1      (write x=1)	MOV [y] ← 1      (write y=1)
MOV EAX ← [y]    (read y=0)	MOV EBX ← [x]    (read x=0)
Allowed Final State: Thread 0:EAX=0 $\wedge$ Thread 1:EBX=0	

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

## Store Buffer (SB)

Thread 0	Thread 1
MOV [x] ← 1      (write x=1)	MOV [y] ← 1      (write y=1)
MOV EAX ← [y]    (read y=0)	MOV EBX ← [x]    (read x=0)
Allowed Final State: Thread 0:EAX=0 $\wedge$ Thread 1:EBX=0	



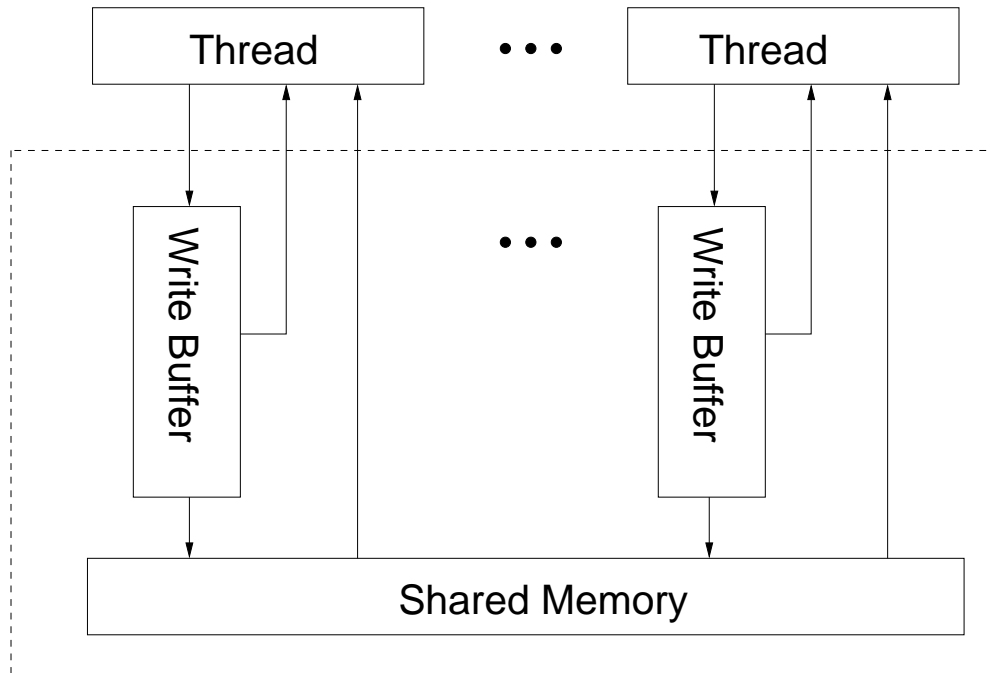
## Litmus Test 2.4. Intra-processor forwarding is allowed

Thread 0	Thread 1
MOV [x]←1 (write x=1)	MOV [y]←1 (write y=1)
MOV EAX←[x] (read x=1)	MOV ECX←[y] (read y=1)
MOV EBX←[y] (read y=0)	MOV EDX←[x] (read x=0)
Allowed Final State: Thread 0:EBX=0 $\wedge$ Thread 1:EDX=0 Thread 0:EAX=1 $\wedge$ Thread 1:ECX=1	



## Litmus Test 2.4. Intra-processor forwarding is allowed

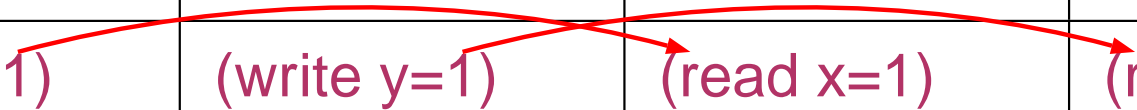
Thread 0	Thread 1
MOV [x]←1 (write x=1)	MOV [y]←1 (write y=1)
MOV EAX←[x] (read x=1)	MOV ECX←[y] (read y=1)
MOV EBX←[y] (read y=0)	MOV EDX←[x] (read x=0)
<b>Allowed Final State:</b> Thread 0:EBX=0 $\wedge$ Thread 1:EDX=0 Thread 0:EAX=1 $\wedge$ Thread 1:ECX=1	



# Problem 1: Weakness

Independent Reads of Independent Writes (IRIW)

Thread 0	Thread 1	Thread 2	Thread 3
(write x=1)	(write y=1)	(read x=1)	(read y=1)
		(read y=0)	(read x=0)
Allowed or Forbidden?			



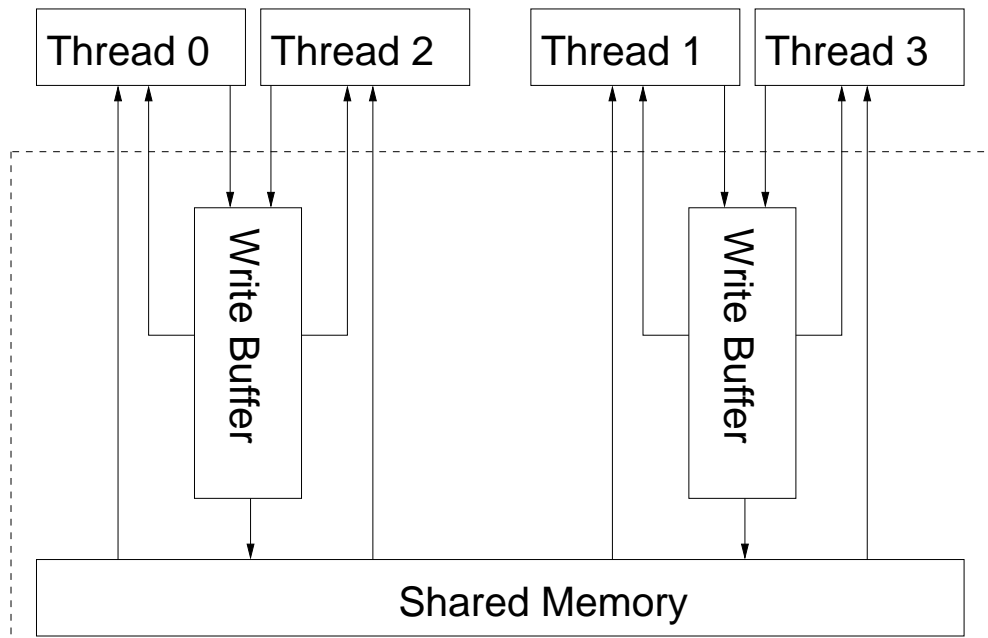
# Problem 1: Weakness

Independent Reads of Independent Writes (IRIW)

Thread 0	Thread 1	Thread 2	Thread 3
(write x=1)	(write y=1)	(read x=1)	(read y=1)
		(read y=0)	(read x=0)

Allowed or Forbidden?

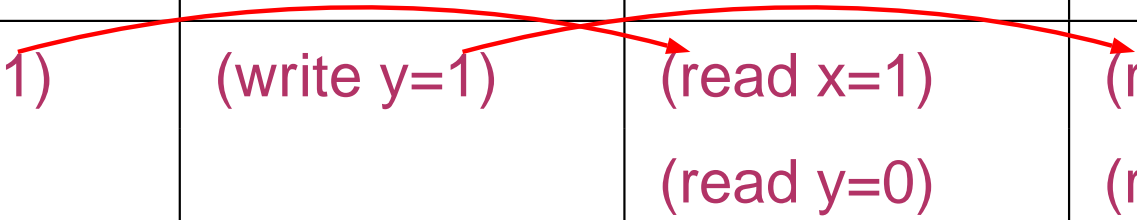
Microarchitecturally plausible? yes, e.g. with shared store buffers



# Problem 1: Weakness

## Independent Reads of Independent Writes (IRIW)

Thread 0	Thread 1	Thread 2	Thread 3
(write x=1)	(write y=1)	(read x=1)	(read y=1)
		(read y=0)	(read x=0)
Allowed or Forbidden?			



- AMD3.14: Allowed
- IWP: ???
- Real hardware: unobserved
- Problem for normal programming: ?

Weakness: adding memory barriers does not recover SC, which was assumed in a Sun implementation of the JMM

# Problem 2: Ambiguity

P1–4. ...may be reordered with...

P5. Intel 64 memory ordering ensures **transitive visibility of stores** — i.e. stores that are **causally related** appear to execute in an order consistent with **the causal relation**

## Write-to-Read Causality (WRC) (Litmus Test 2.5)

Thread 0	Thread 1	Thread 2
MOV [x]←1 (W x=1)	MOV EAX←[x] (R x=1)	MOV EBX←[y] (R y=1)
	MOV [y]←1 (W y=1)	MOV ECX←[x] (R x=0)
Forbidden Final State: Thread 1:EAX=1 $\wedge$ Thread 2:EBX=1 $\wedge$ Thread 2:ECX=0		

# Problem 3: Unsoundness!

Example from Paul Loewenstein:

n6

Thread 0	Thread 1
MOV [x] ← 1 (a:W x=1)	MOV [y] ← 2 (d:W y=2)
MOV EAX ← [x] (b:R x=1)	MOV [x] ← 2 (e:W x=2)
MOV EBX ← [y] (c:R y=0)	
Allowed Final State: Thread 0:EAX=1 ∧ Thread 0:EBX=0 ∧ x=1	

Observed on real hardware, but not allowed by (any interpretation we can make of) the IWP ‘principles’.

(can see allowed in store-buffer microarchitecture)

# Problem 3: Unsoundness!

Example from Paul Loewenstein:

n6

Thread 0	Thread 1
MOV [x]←1 (a:W x=1)	MOV [y]←2 (d:W y=2)
MOV EAX←[x] (b:R x=1)	MOV [x]←2 (e:W x=2)
MOV EBX←[y] (c:R y=0)	
Allowed Final State: Thread 0:EAX=1 ∧ Thread 0:EBX=0 ∧ x=1	

In the view of Thread 0:

a→b by P4: Reads may [...] not be reordered with older writes to the same location.

b→c by P1: Reads are not reordered with other reads.

c→d, otherwise c would read 2 from d

d→e by P3. Writes are not reordered with older reads.

so a:Wx=1 → e:Wx=2

But then that should be respected in the final state, by P6: In a multiprocessor system, stores to the same location have a total order, and it isn't.

(can see allowed in store-buffer microarchitecture)

# Problem 3: Unsoundness!

Example from Paul Loewenstein:

n6

Thread 0	Thread 1
MOV [x]←1 (a:W x=1)	MOV [y]←2 (d:W y=2)
MOV EAX←[x] (b:R x=1)	MOV [x]←2 (e:W x=2)
MOV EBX←[y] (c:R y=0)	
Allowed Final State: Thread 0:EAX=1 $\wedge$ Thread 0:EBX=0 $\wedge$ x=1	

Observed on real hardware, but not allowed by (any interpretation we can make of) the IWP ‘principles’.

(can see allowed in store-buffer microarchitecture)

So spec unsound (and also our POPL09 model based on it).



# Intel SDM and AMD64, Nov. 2008 – now

Intel SDM rev. 29–35 and AMD3.17

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*

Intel:

<http://www.intel.com/products/processor/manuals/index.htm>  
(rev. 35 on 6/10/2010).

See especially SDM Vol. 3A, Ch. 8.

AMD:

<http://developer.amd.com/documentation/guides/Pages/default.aspx>

(rev. 3.17 on 6/10/2010).

See especially APM Vol. 2, Ch. 7.

# Why all these problems?

Recall that the vendor *architectures* are:

- loose specifications;
- claimed to cover a wide range of past and future processor implementations.

Architectures should:

- reveal enough for effective programming;
- without revealing sensitive IP; and
- without unduly constraining future processor design.

There's a big tension between these, compounded by internal politics and inertia.

# Fundamental Problem

Architecture texts: *informal prose* attempts at subtle loose specifications

Fundamental problem: prose specifications cannot be used

- to *test programs against*, or
- to *test processor implementations*, or
- to *prove* properties of either, or even
- to *communicate precisely*.

# Aside: x86 ISA, Locked Instructions

Thread 0	Thread 1
INC x	INC x

# Aside: x86 ISA, Locked Instructions

Thread 0	Thread 1
INC x (read x=0; write x=1)	INC x (read x=0; write x=1)
Allowed Final State: [x]=1	

Non-atomic (even in SC semantics)

# Aside: x86 ISA, Locked Instructions

Thread 0	Thread 1
INC x (read x=0; write x=1)	INC x (read x=0; write x=1)
Allowed Final State: [x]=1	

Non-atomic (even in SC semantics)

Thread 0	Thread 1
LOCK;INC x	LOCK;INC x
Forbidden Final State: [x]=1	

# Aside: x86 ISA, Locked Instructions

Thread 0	Thread 1
INC x (read x=0; write x=1)	INC x (read x=0; write x=1)
Allowed Final State: [x]=1	

Non-atomic (even in SC semantics)

Thread 0	Thread 1
LOCK;INC x	LOCK;INC x
Forbidden Final State: [x]=1	

Also LOCK'd ADD, SUB, XCHG, etc., and CMPXCHG



# Aside: x86 ISA, Locked Instructions

Compare-and-swap (CAS):

`CMPXCHG dest ← src`

compares EAX with dest, then:

- if equal, set ZF=1 and load src into dest,
- otherwise, clear ZF=0 and load dest into EAX

All this is one *atomic* step.

Can use to solve *consensus* problem...

# Aside: x86 ISA, Memory Barriers

MFENCE memory barrier

(also SFENCE and LFENCE)

# Hardware Models

x86 in detail

Why are industrial specs so often flawed?

**A usable model: x86-TSO**

Reasoning about x86-TSO code: races

Power/ARM

SPARC, Alpha, Itanium

Programming Language Models (Java/C++)

# Inventing a Usable Abstraction

Have to be:

- Unambiguous
- Sound w.r.t. experimentally observable behaviour
- Easy to understand
- Consistent with what we know of vendors intentions
- Consistent with expert-programmer reasoning

Key facts:

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

These suggest that x86 is, in practice, like SPARC TSO.

# x86-TSO Abstract Machine

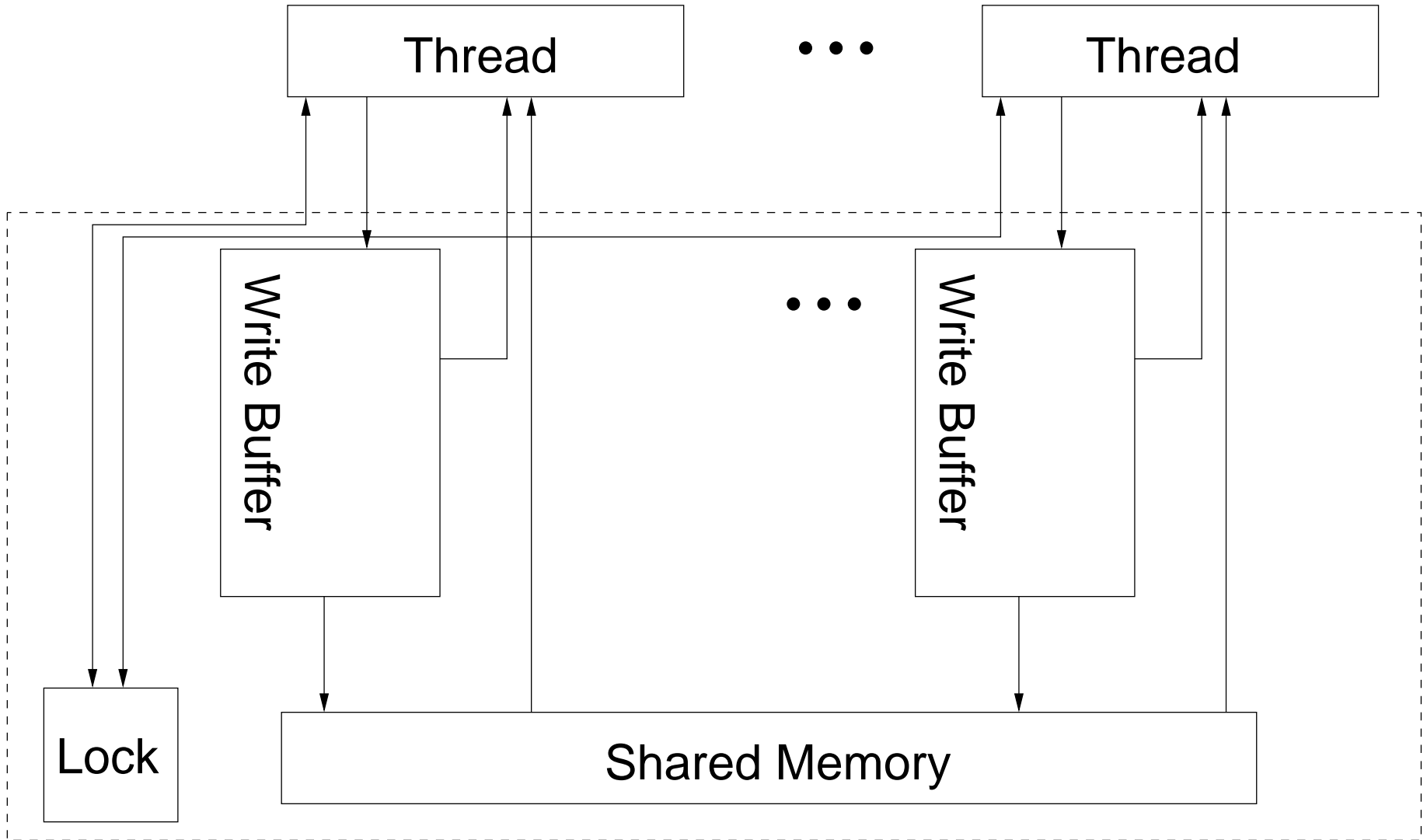
Separate *instruction semantics* and *memory model*

Define the memory model in two (provably equivalent) styles:

- an abstract machine (or operational model)
- an axiomatic model

Put the instruction semantics and abstract machine in parallel, exchanging read and write messages (and lock/unlock messages).

# x86-TSO Abstract Machine



# x86-TSO Abstract Machine: Interface

## Events

$e ::=$	$W_t x=v$	a write of value $v$ to address $x$ by thread $t$
	$R_t x=v$	a read of $v$ from $x$ by $t$
	$B_t$	an MFENCE memory barrier by $t$
	$L_t$	start of an instruction with LOCK prefix by $t$
	$U_t$	end of an instruction with LOCK prefix by $t$
	$\tau_t x=v$	an internal action of the machine, moving $x = v$ from the write buffer on $t$ to shared memory

where

- $t$  is a hardware thread id, of type  $tid$ ,
- $x$  and  $y$  are memory addresses, of type  $addr$
- $v$  and  $w$  are machine words, of type  $value$

# x86-TSO Abstract Machine: Machine States

A *machine state*  $s$  is a record

$$s : \langle \begin{array}{l} M : \text{addr} \rightarrow \text{value}; \\ B : \text{tid} \rightarrow (\text{addr} \times \text{value}) \text{ list}; \\ L : \text{tid option} \end{array} \rangle$$

Here:

- $s.M$  is the shared memory, mapping addresses to values
- $s.B$  gives the store buffer for each thread
- $s.L$  is the global machine lock indicating when a thread has exclusive access to memory



# x86-TSO Abstract Machine: Auxiliary Definitions

Say  $t$  is *not blocked* in machine state  $s$  if either it holds the lock ( $s.L = \text{SOME } t$ ) or the lock is not held ( $s.L = \text{NONE}$ ).

Say there are *no pending* writes in  $t$ 's buffer  $s.B(t)$  for address  $x$  if there are no  $(x, v)$  elements in  $s.B(t)$ .

# x86-TSO Abstract Machine: Behaviour

## RM: Read from memory

$\text{not\_blocked}(s, t)$

$s.M(x) = v$

$\text{no\_pending}(s.B(t), x)$

$$\frac{}{s \xrightarrow{R_t x=v} s}$$

Thread  $t$  can read  $v$  from memory at address  $x$  if  $t$  is not blocked, the memory does contain  $v$  at  $x$ , and there are no writes to  $x$  in  $t$ 's store buffer.

# x86-TSO Abstract Machine: Behaviour

## RB: Read from write buffer

$\text{not\_blocked}(s, t)$

$\exists b_1 b_2. s.B(t) = b_1 \ ++ \ [(x, v)] \ ++ \ b_2$

$\text{no\_pending}(b_1, x)$

---

$$s \xrightarrow{R_t \ x=v} s$$

Thread  $t$  can read  $v$  from its store buffer for address  $x$  if  $t$  is not blocked and has  $v$  as the newest write to  $x$  in its buffer;

# x86-TSO Abstract Machine: Behaviour

## WB: Write to write buffer

---

$$s \xrightarrow{W_t x=v} s \oplus \langle [B := s.B \oplus (t \mapsto ([x, v] ++ s.B(t)))] \rangle$$

Thread  $t$  can write  $v$  to its store buffer for address  $x$  at any time;

# x86-TSO Abstract Machine: Behaviour

**WM: Write from write buffer to memory**

$\text{not\_blocked}(s, t)$

$s.B(t) = b \text{ ++ } [(x, v)]$

---

$s \xrightarrow{\tau_t x=v}$

$s \oplus \langle [M := s.M \oplus (x \mapsto v)] \rangle \oplus \langle [B := s.B \oplus (t \mapsto b)] \rangle$

If  $t$  is not blocked, it can silently dequeue the oldest write from its store buffer and place the value in memory at the given address, without coordinating with any hardware thread

# x86-TSO Abstract Machine: Behaviour

## L: Lock

$$s.L = \text{NONE}$$

$$s.B(t) = []$$

---

$$s \xrightarrow{L_t} s \oplus \langle [L := \text{SOME}(t)] \rangle$$

If the lock is not held and its buffer is empty, thread  $t$  can begin a LOCK'd instruction.

Note that if a hardware thread  $t$  comes to a LOCK'd instruction when its store buffer is not empty, the machine can take one or more  $\tau_t x=v$  steps to empty the buffer and then proceed.

# x86-TSO Abstract Machine: Behaviour

## U: Unlock

$$s.L = \text{SOME}(t)$$

$$s.B(t) = []$$

---

$$s \xrightarrow{U_t} s \oplus \langle [L := \text{NONE}] \rangle$$

If  $t$  holds the lock, and its store buffer is empty, it can end a LOCK'd instruction.

# x86-TSO Abstract Machine: Behaviour

## B: Barrier

$$\frac{s.B(t) = []}{s \xrightarrow{B_t} s}$$

If  $t$ 's store buffer is empty, it can execute an MFENCE.



# Notation Reference

SOME and NONE construct optional values

$(\cdot, \cdot)$  builds tuples

$[\ ]$  builds lists

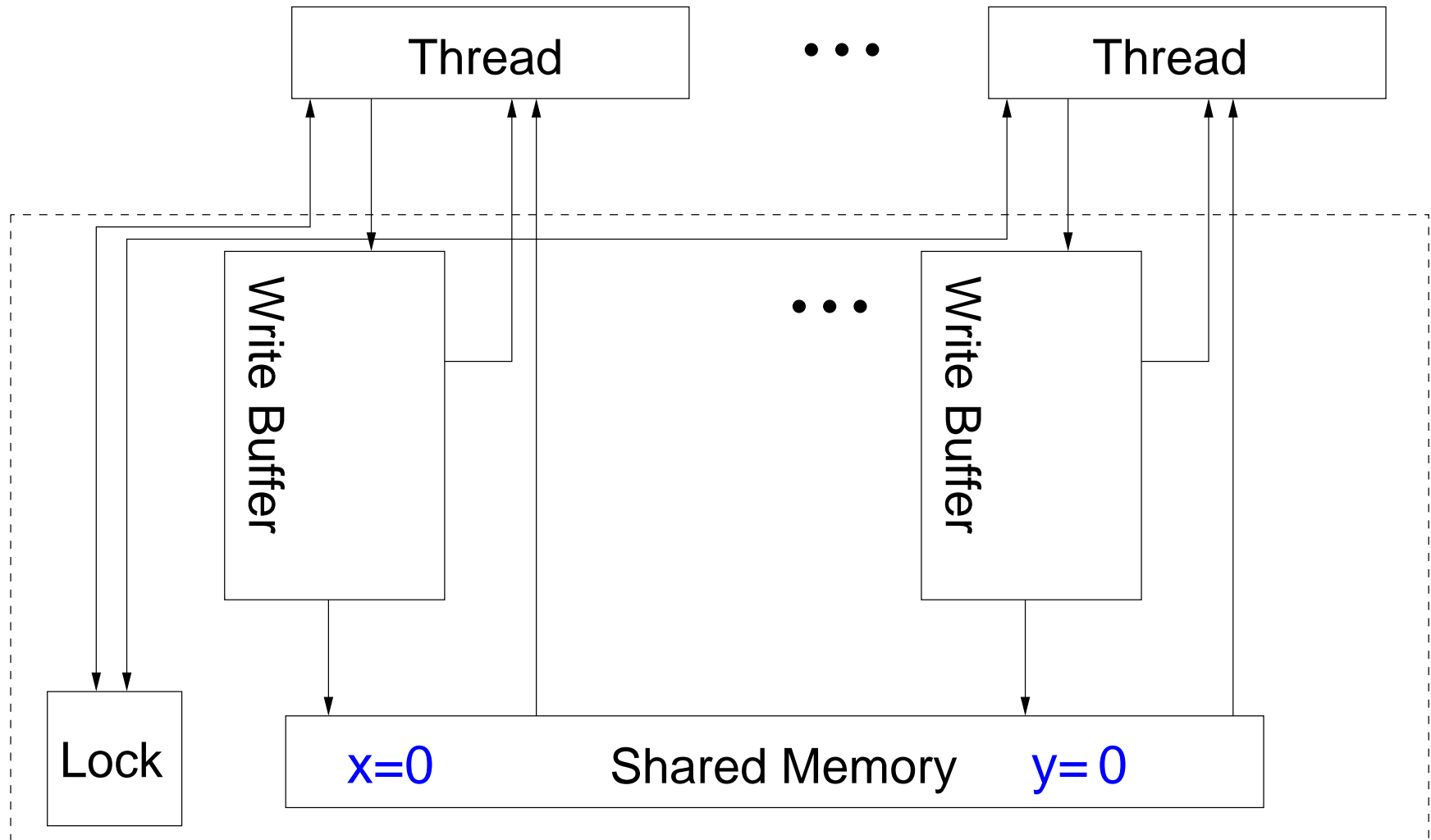
$++$  appends lists

$\cdot \oplus \langle \cdot := \cdot \rangle$  updates records

$\cdot (\cdot \mapsto \cdot)$  updates functions.

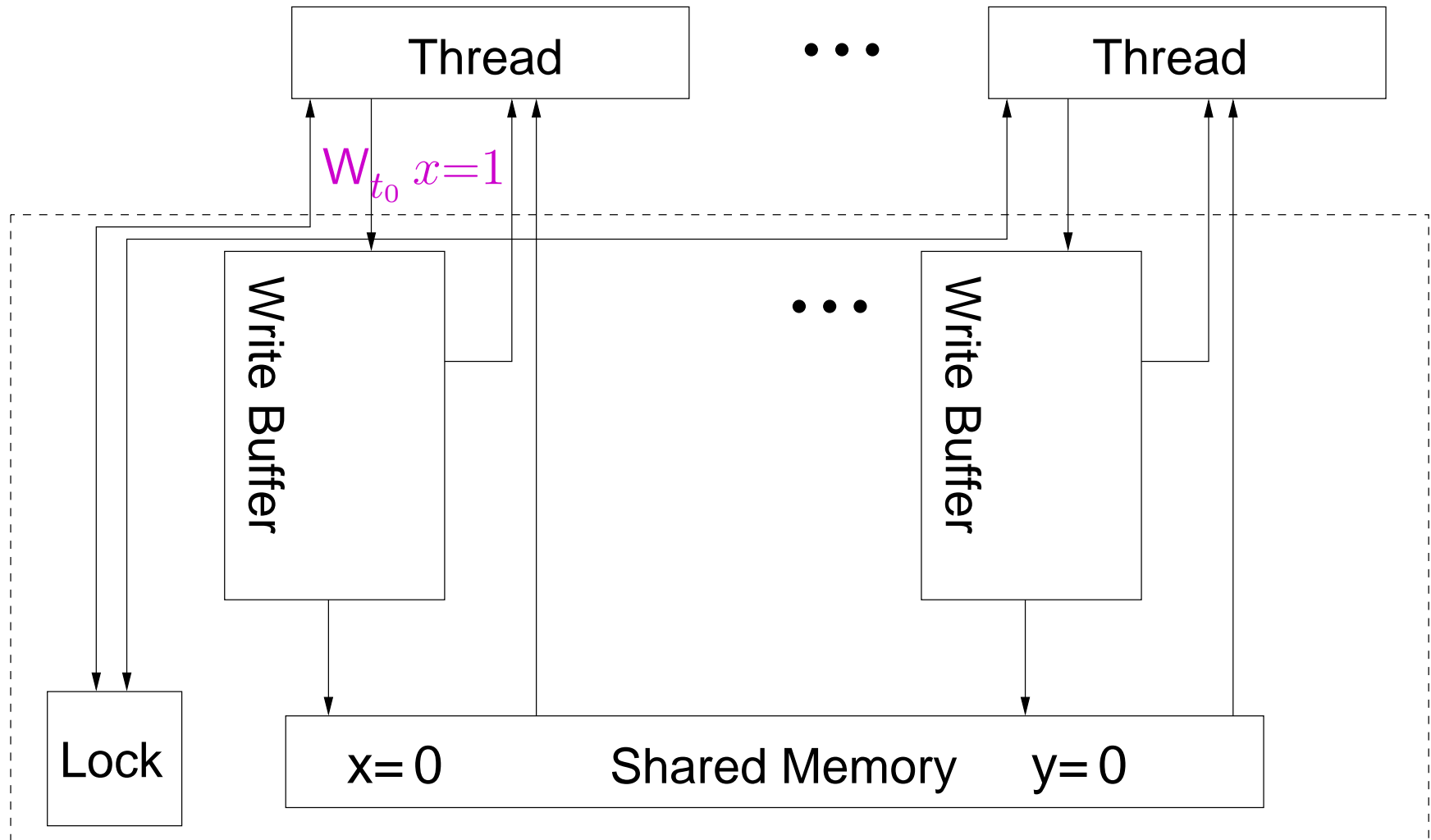
# First Example, Revisited

Thread 0		Thread 1	
MOV [x]←1	(write x=1)	MOV [y]←1	(write y=1)
MOV EAX←[y]	(read y)	MOV EBX←[x]	(read x)



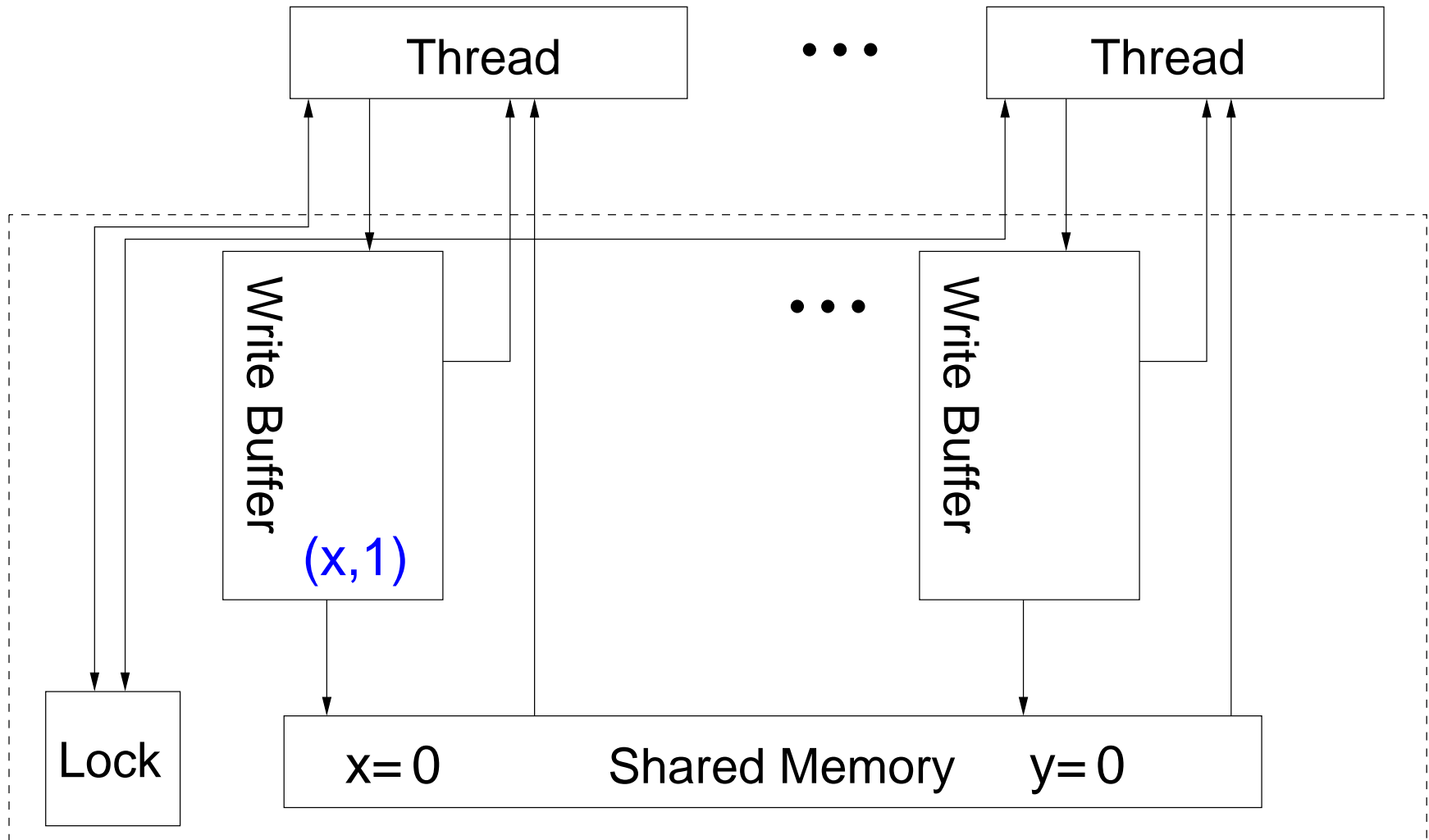
# First Example, Revisited

Thread 0		Thread 1	
MOV [x]←1	(write x=1)	MOV [y]←1	(write y=1)
MOV EAX←[y]	(read y)	MOV EBX←[x]	(read x)



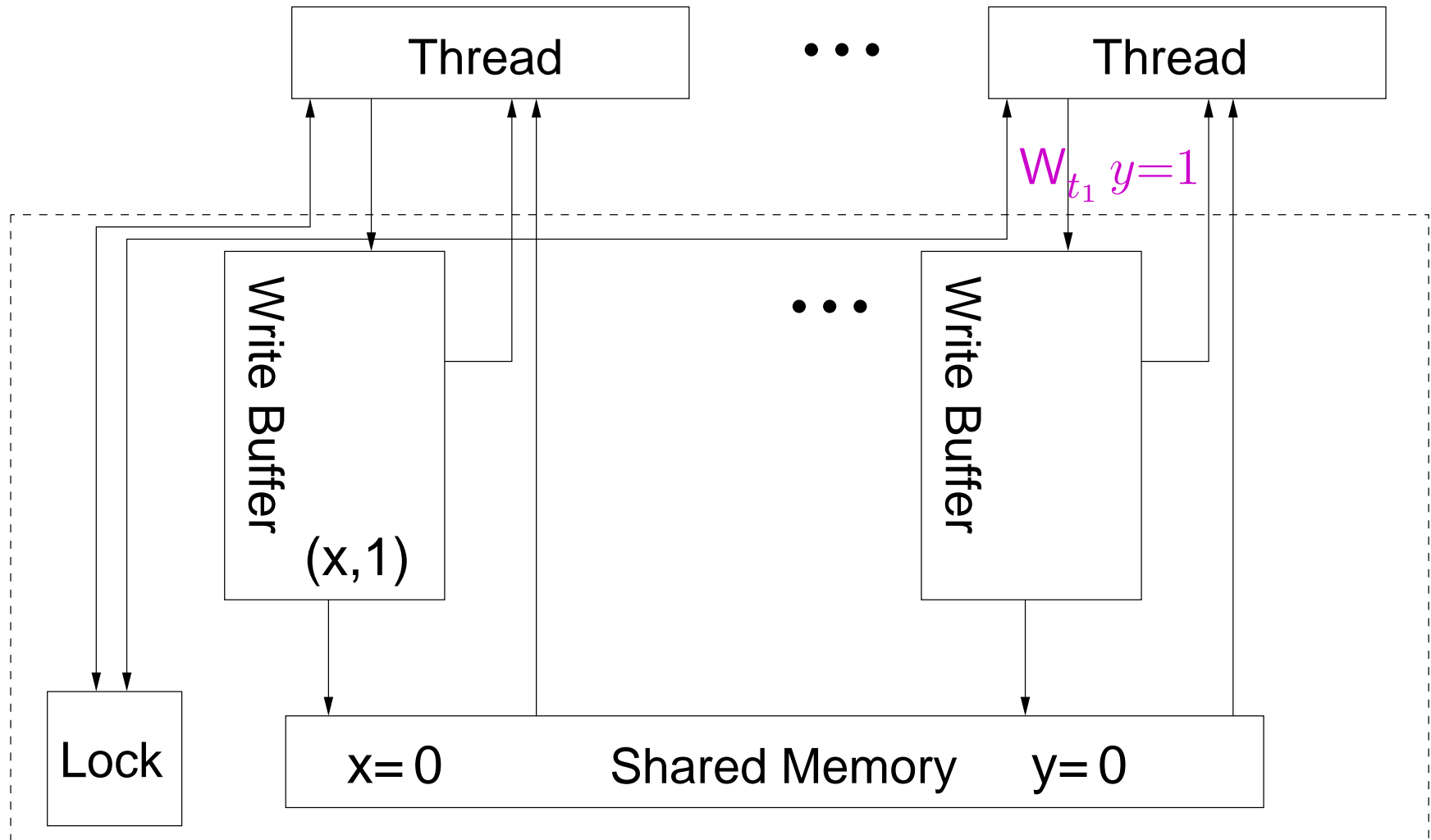
# First Example, Revisited

Thread 0	Thread 1
MOV [x]←1 (write x=1)	MOV [y]←1 (write y=1)
MOV EAX←[y] (read y)	MOV EBX←[x] (read x)



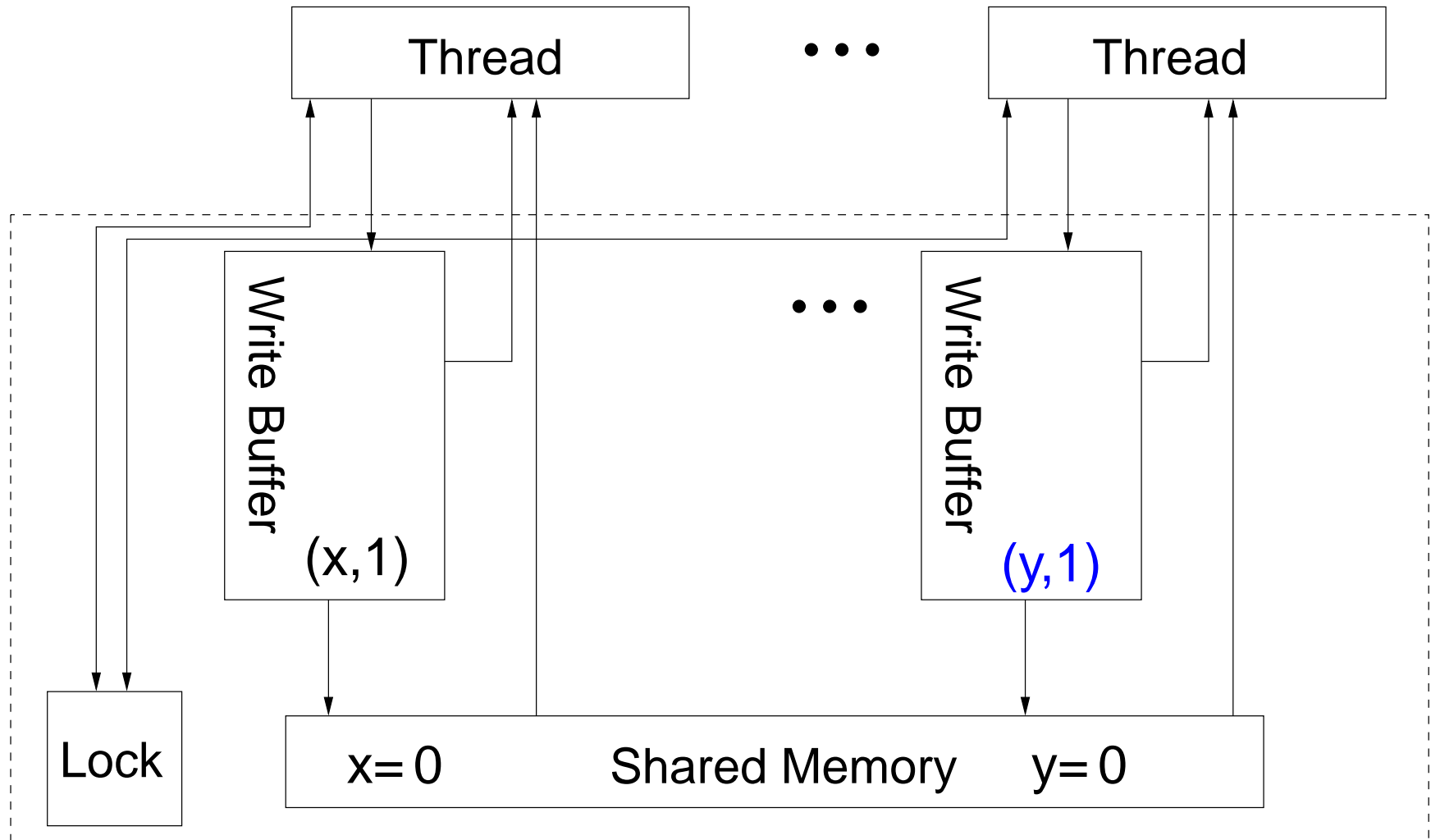
# First Example, Revisited

Thread 0		Thread 1	
MOV [x]←-1	(write x=1)	MOV [y]←-1	(write y=1)
MOV EAX←[y]	(read y)	MOV EBX←[x]	(read x)



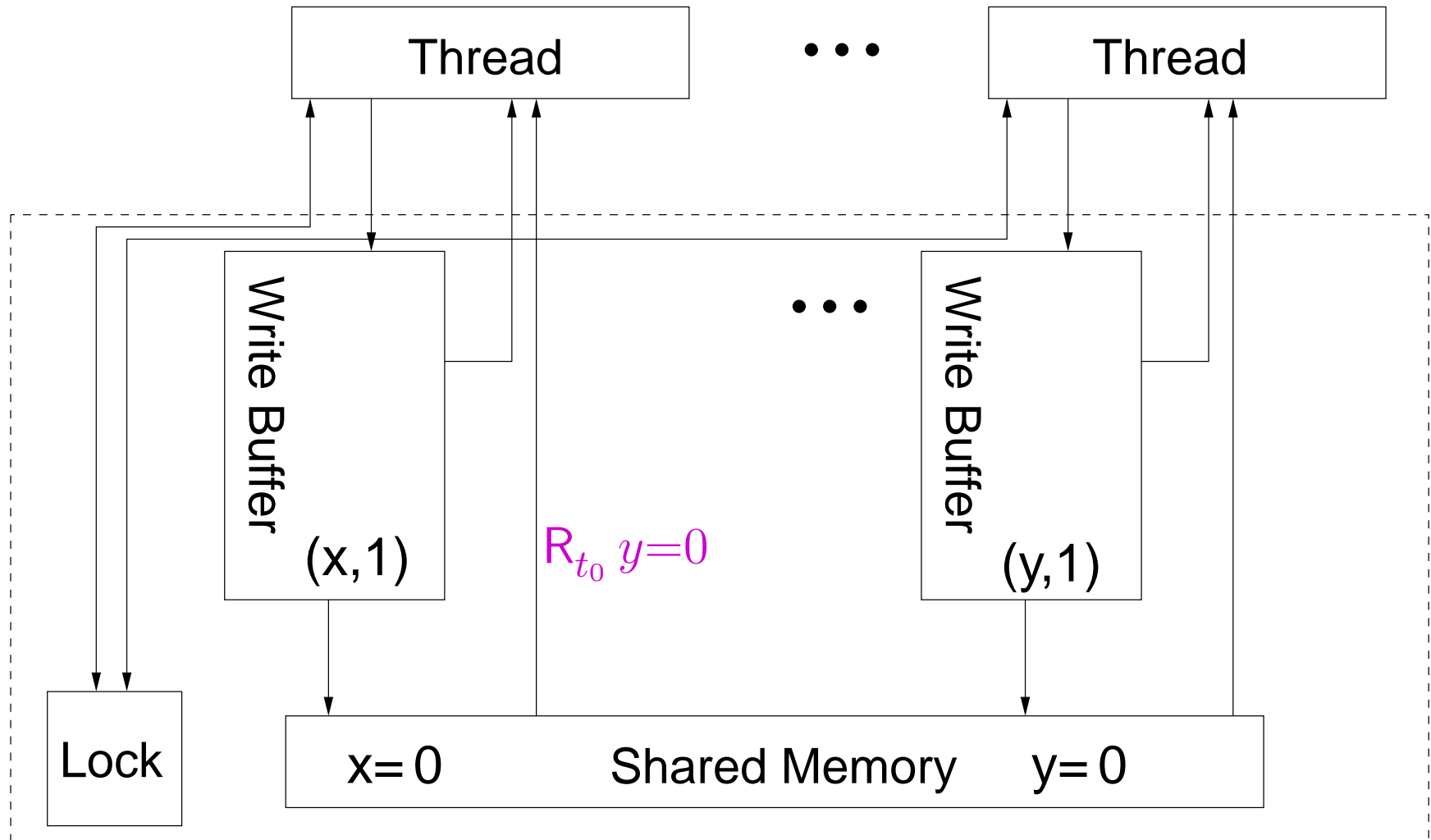
# First Example, Revisited

Thread 0		Thread 1	
MOV [x]←1	(write x=1)	MOV [y]←1	(write y=1)
MOV EAX←[y]	(read y)	MOV EBX←[x]	(read x)



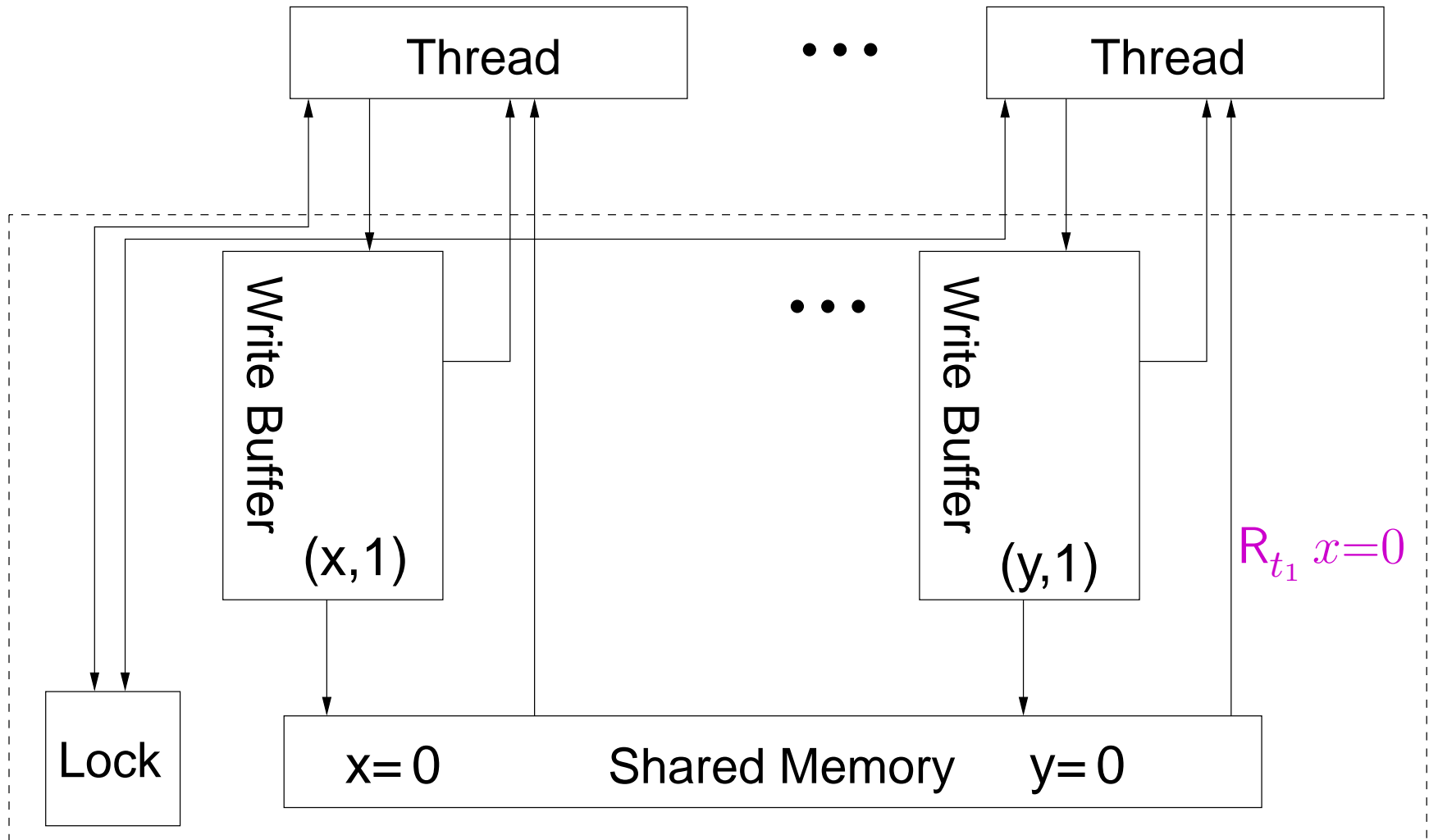
# First Example, Revisited

Thread 0		Thread 1	
MOV [x]←1	(write x=1)	MOV [y]←1	(write y=1)
MOV EAX←[y]	(read y)	MOV EBX←[x]	(read x)



# First Example, Revisited

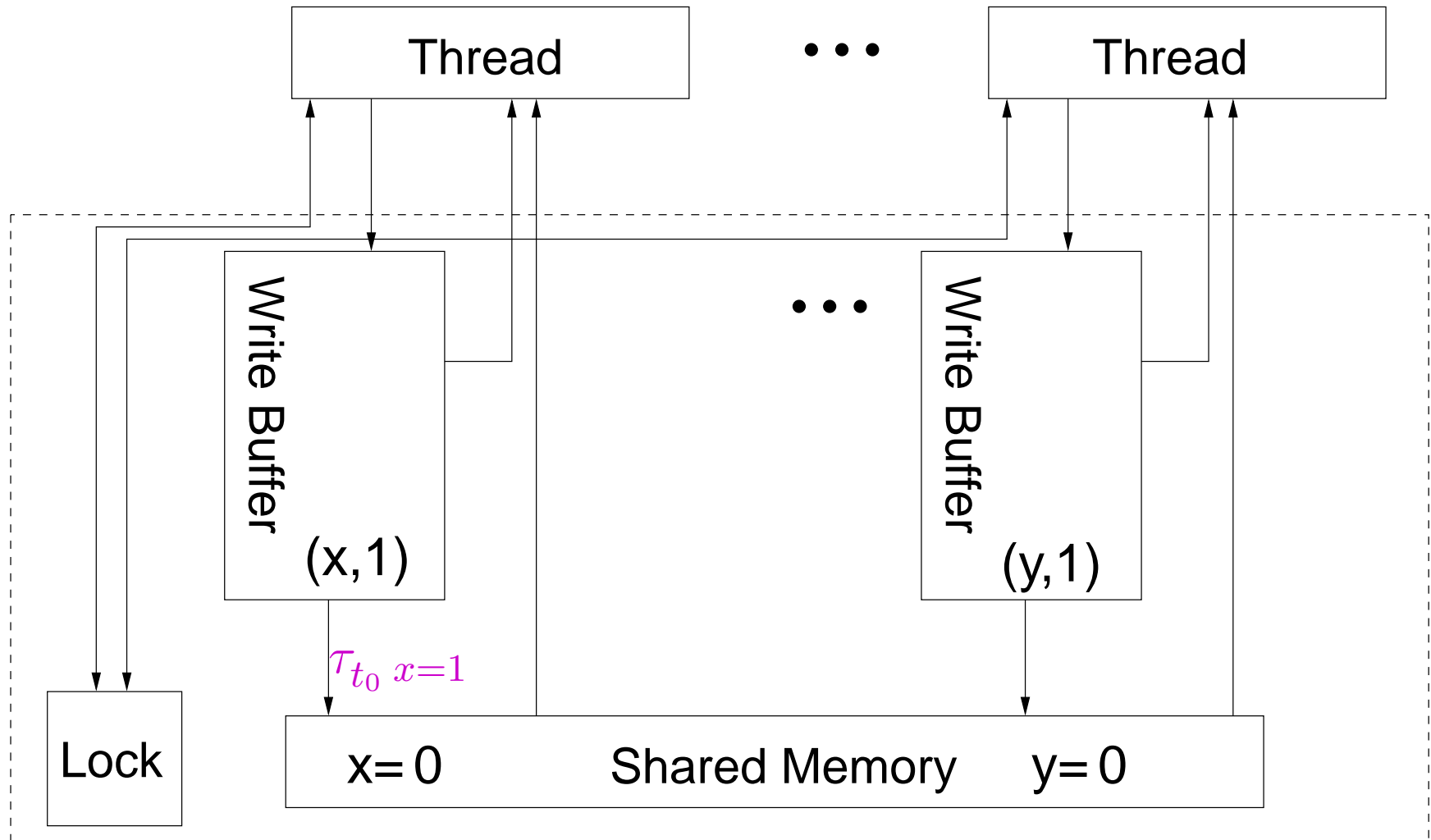
Thread 0		Thread 1	
MOV [x]←1	(write x=1)	MOV [y]←1	(write y=1)
MOV EAX←[y]	(read y)	MOV EBX←[x]	(read x)





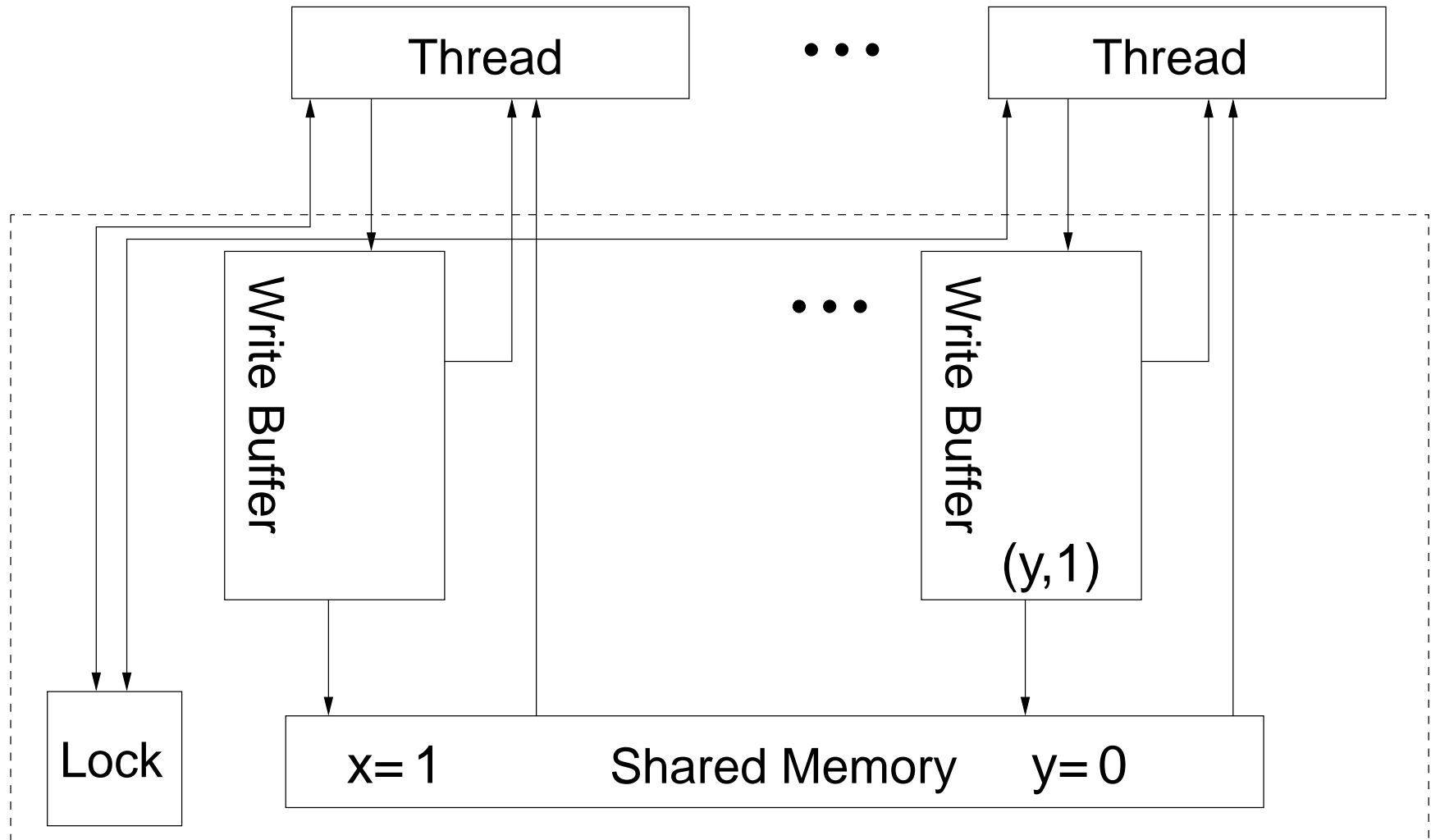
# First Example, Revisited

Thread 0		Thread 1	
MOV [x]←1	(write x=1)	MOV [y]←1	(write y=1)
MOV EAX←[y]	(read y)	MOV EBX←[x]	(read x)



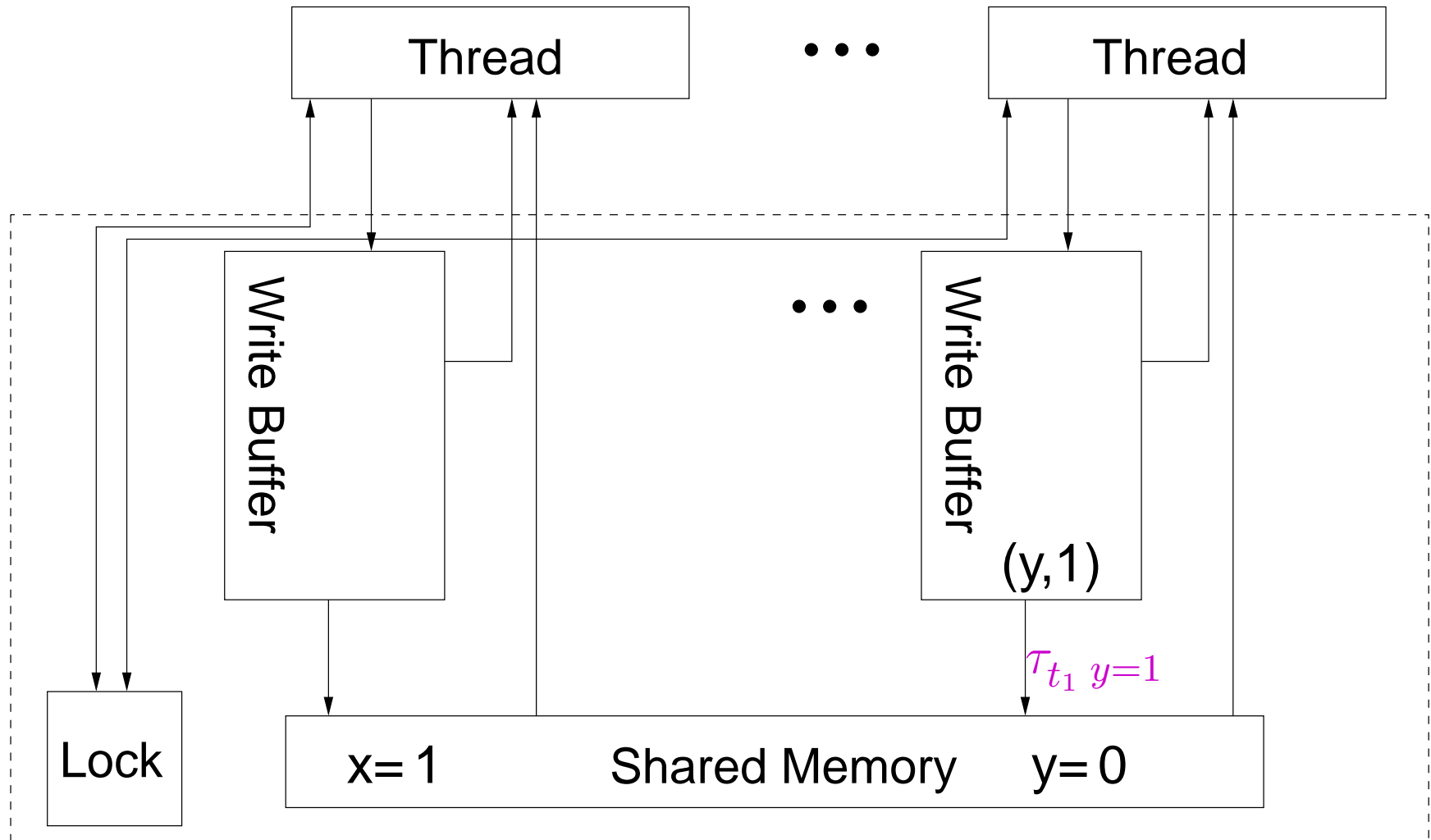
# First Example, Revisited

Thread 0	Thread 1
MOV [x]←1 (write x=1)	MOV [y]←1 (write y=1)
MOV EAX←[y] (read y)	MOV EBX←[x] (read x)



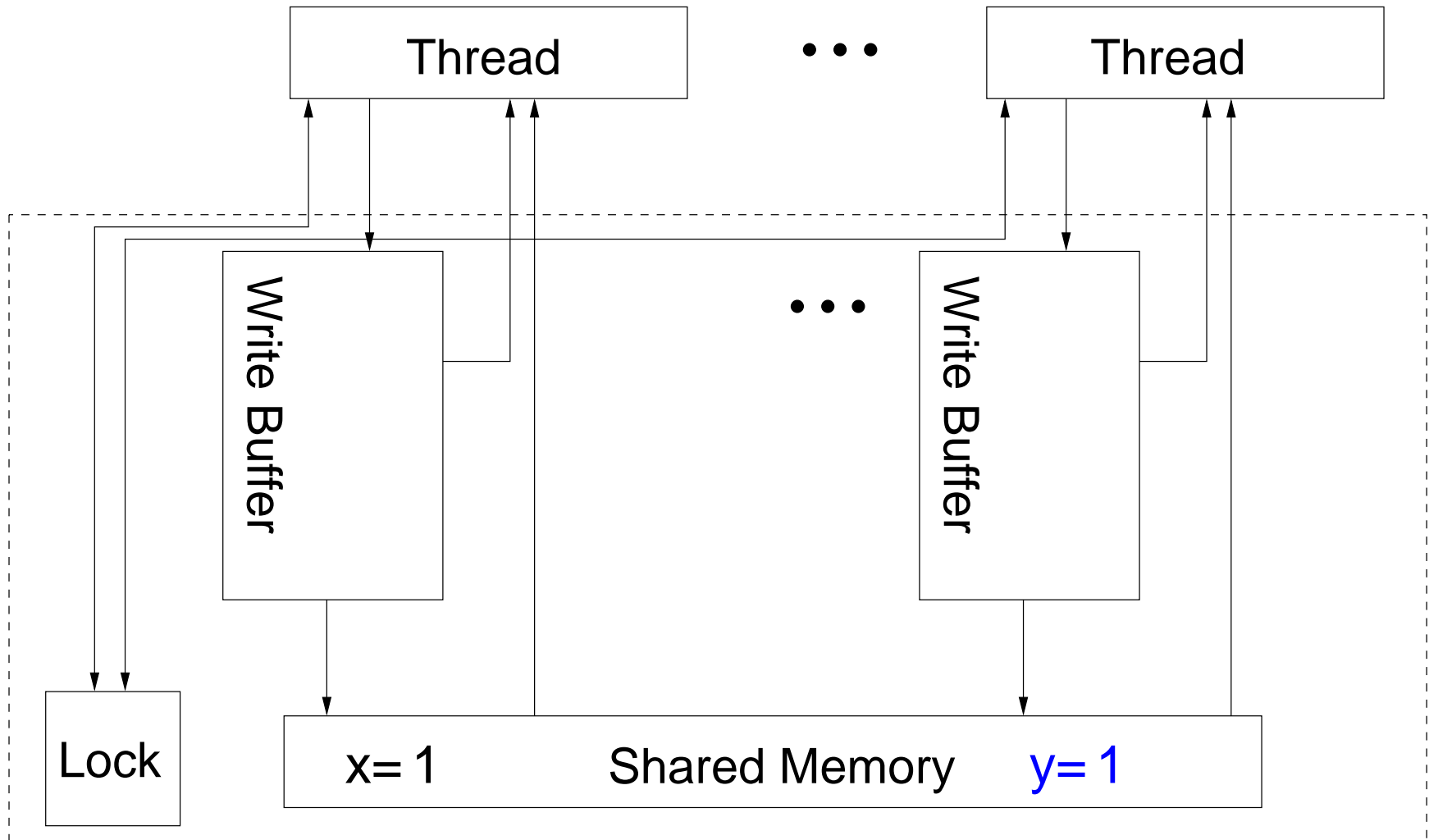
# First Example, Revisited

Thread 0		Thread 1	
MOV [x]←1	(write x=1)	MOV [y]←1	(write y=1)
MOV EAX←[y]	(read y)	MOV EBX←[x]	(read x)



# First Example, Revisited

Thread 0		Thread 1	
MOV [x]←1	(write x=1)	MOV [y]←1	(write y=1)
MOV EAX←[y]	(read y)	MOV EBX←[x]	(read x)



# Barriers and LOCK'd Instructions, recap

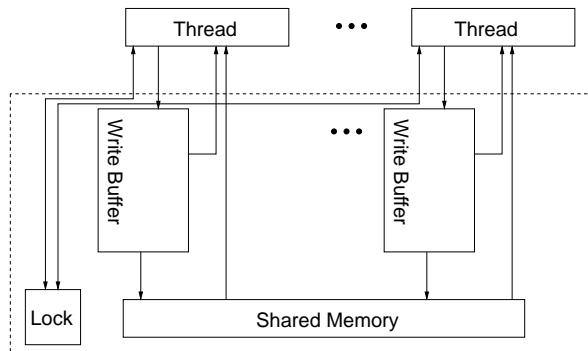
- MFENCE memory barrier
  - flushes local write buffer
- LOCK'd instructions (atomic INC, ADD, CMPXCHG, etc.)
  - flush local write buffer
  - globally locks memory

Thread 0	Thread 1
MOV [x]←1 (write x=1)	MOV [y]←1 (write y=1)
MFENCE	MFENCE
MOV EAX←[y] (read y=0)	MOV EBX←[x] (read x=0)
<b>Forbidden</b> Final State: Thread 0:EAX=0 $\wedge$ Thread 1:EBX=0	

NB: both are *expensive*

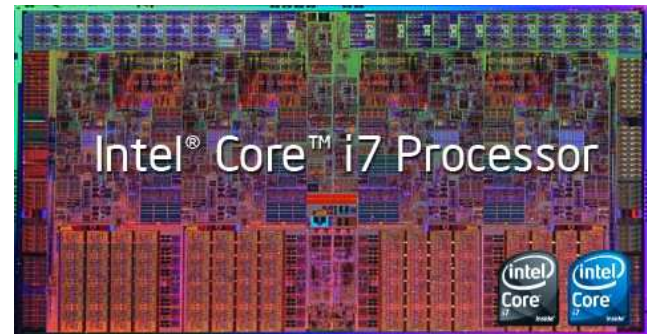
# NB: This is an *Abstract Machine*

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



$\supseteq$  beh

$\neq$  hw



Force: Of 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

# Processors, Hardware Threads, and Threads

Our 'Threads' are hardware threads.

Some processors have *simultaneous multithreading* (Intel: hyperthreading): multiple hardware threads/core sharing resources.

If the OS flushes store buffers on context switch, software threads should have the same semantics.

# Validating the Semantics

Testing tools:

- LITMUS, runs litmus tests on real h/w
- MEMEVENTS, symbolically finds all possible results
- EMUL, daemononic emulator

(Also modelling & testing instruction semantics)

Informal vendor support

Formalized in theorem prover (HOL4)

*One* reasonable model



# Liveness

Question: is every memory write guaranteed to eventually propagate from store buffer to shared memory?

We tentatively assume so (with a progress condition on machine traces).

AMD: yes

Intel: unclear

(ARM: yes)

# NB: Not *All* of x86

Coherent write-back memory (almost all code), but assume

- no exceptions
- no misaligned or mixed-size accesses
- no 'non-temporal' operations
- no device memory
- no self-modifying code
- no page-table changes

# x86-TSO: The Axiomatic Model

The abstract machine generates x86-TSO executions stepwise.

The axiomatic model says whether a complete candidate execution is admitted by x86-TSO.

Events:  $i:W_t x=v$ ,  $i:R_t x=v$  and  $i:B_t$  as before, but with unique ids  $i$ .

Event structures  $E$ :

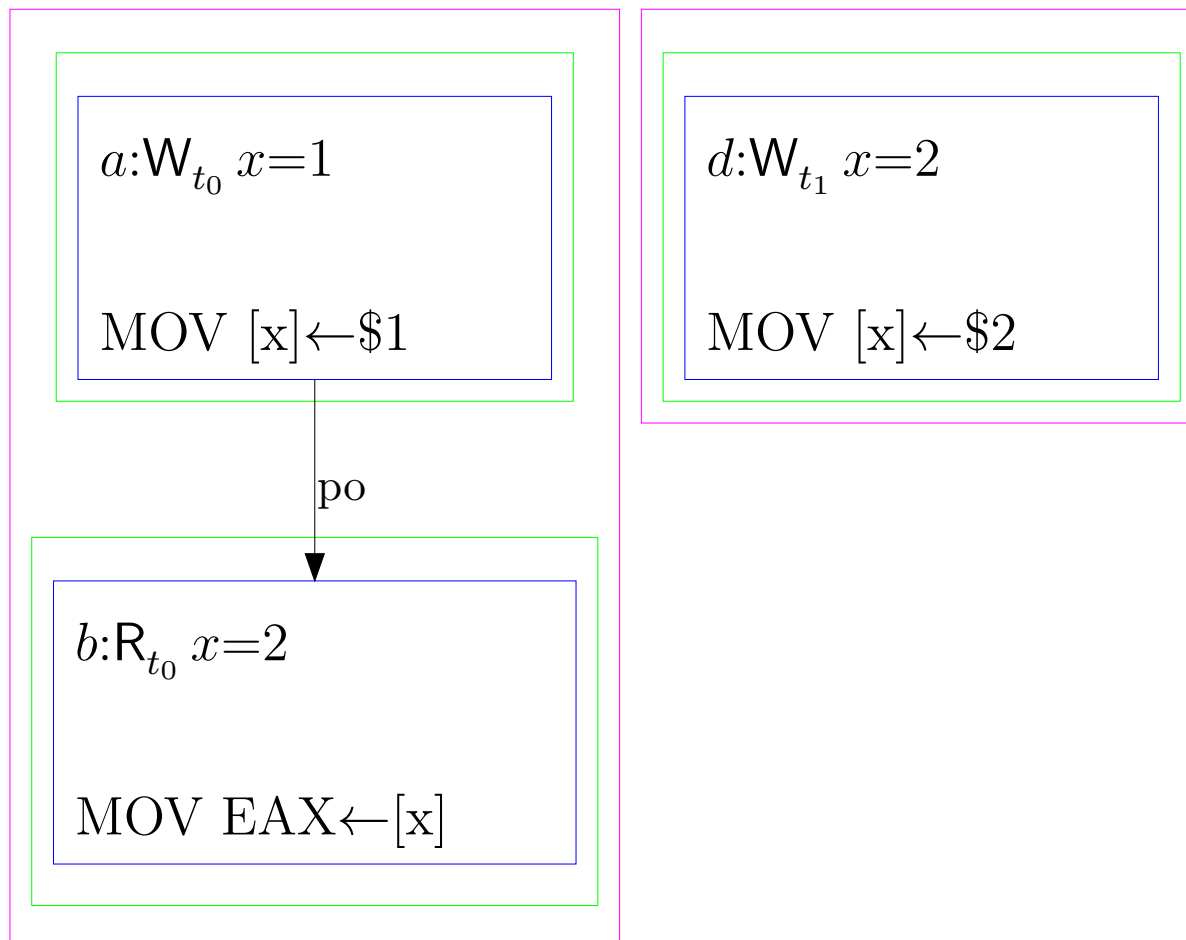
- a set of events
- program order (po) and intra-instruction causality order (iico) over them (strict partial orders)
- an atomicity relation over them (a partial equivalence relation)

Execution witness  $X$ :

execution\_witness =

$\langle \langle$   $memory\_order$  : event reln;  
 $rfmap$  : event reln;  
 $initial\_state$  : addr  $\rightarrow$  value  $\rangle \rangle$

tso1	Thread $t_0$	Thread $t_1$
	MOV [x]←\$1	MOV [x]←\$2
	MOV EAX←[x]	



# Axioms: Memory Order

$X.memory\_order$  is a partial order over memory read and write events of  $E$

$X.memory\_order$ , when restricted to the write events of  $E$ , is a linear order.

# Axioms: Reads-from map

The r fmap only relates such pairs with the same address and value:

$\text{reads\_from\_map\_candidates } E \text{ r fmap} =$

$$\forall (ew, er) \in \text{r fmap}. (er \in \text{mem\_reads } E) \wedge (ew \in \text{mem\_writes } E) \wedge \\ (\text{loc } ew = \text{loc } er) \wedge (\text{value\_of } ew = \text{value\_of } er)$$

**Auxiliary functions over events:**  $\text{loc}$ ,  $\text{value\_of}$

**Auxiliary functions over event structures:**

$\text{mem\_reads}$ ,  $\text{mem\_writes}$ ,  $\text{mem\_accesses}$ ,  $\text{mfences}$

# Axioms: *check\_rfmap\_written*

Let  $po\_iico$   $E$  be the union of (strict) program order and intra-instruction causality.

Check that the *rfmap* relates a read to the most recent preceding write.

$previous\_writes$   $E$   $er$   $<_{order}$  =

$$\{ew' \mid ew' \in mem\_writes\ E \wedge ew' <_{order} er \wedge (loc\ ew' = loc\ er)\}$$

$check\_rfmap\_written$   $E$   $X$  =

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

$$ew \in maximal\_elements (previous\_writes\ E\ er\ (<_{X.memory\_order}) \cup \\ previous\_writes\ E\ er\ (<_{(po\_iico\ E)})) \\ (<_{X.memory\_order})$$



# Axioms: *check\_rfmap\_initial*

And similarly for the initial state:

$\text{check\_rfmap\_initial } E X =$

$\forall er \in (\text{mem\_reads } E \setminus \text{range } X.\text{rfmap}).$

$(\exists l. (\text{loc } er = l) \wedge (\text{value\_of } er = X.\text{initial\_state } l)) \wedge$

$(\text{previous\_writes } E er (<_{X.\text{memory\_order}}) \cup$

$\text{previous\_writes } E er (<_{(\text{po\_iico } E)}) = \{\})$

# Axioms: R/A Program Order

Program order is included in memory order, for a memory read before a memory access (mo\_po\_read\_access) (SPARCV8's **LoadOp**):

$\forall er \in (\text{mem\_reads } E). \forall e \in (\text{mem\_accesses } E).$

$$er <_{(\text{po\_iico } E)} e \implies er <_{X.\text{memory\_order}} e$$

# Axioms: W/W Program Order

Program order is included in memory order, for a memory write before a memory write (mo\_po\_write\_write) (the SPARCv8 **StoreStore**):

$\forall ew_1 ew_2 \in (\text{mem\_writes } E).$

$$ew_1 <_{(\text{po\_iico } E)} ew_2 \implies ew_1 <_{X.\text{memory\_order}} ew_2$$

# Axioms: Fencing

Program order is included in memory order, for a memory write before a memory read, *if* there is an MFENCE between (mo\_po\_mfence).

$$\forall ew \in (\text{mem\_writes } E). \forall er \in (\text{mem\_reads } E). \forall ef \in (\text{mfences } E). \\ (ew <_{(\text{po\_iico } E)} ef \wedge ef <_{(\text{po\_iico } E)} er) \implies ew <_{X.\text{memory\_order}} er$$

# Axioms: Locked Instructions

Program order is included in memory order, for any two memory accesses where at least one is from a LOCK'd instruction (mo\_po\_access/lock):

$$\forall e_1 e_2 \in (\text{mem\_accesses } E). \forall es \in (E.\text{atomicity}). \\ ((e_1 \in es \vee e_2 \in es) \wedge e_1 <_{(\text{po\_iico } E)} e_2) \implies e_1 <_{X.\text{memory\_order}} e_2$$

# Axioms: Atomicity

The memory accesses of a LOCK'd instruction occur atomically in memory order (*mo\_atomicity*), i.e., there must be no intervening memory events.

Further, all program order relationships between the locked memory accesses and other memory accesses are included in the memory order (this is a generalization of the SPARCV8 **Atomicity** axiom):

$$\forall es \in (E.\textit{atomicity}). \forall e \in (\textit{mem\_accesses } E \setminus es).$$
$$(\forall e' \in (es \cap \textit{mem\_accesses } E). e <_{X.\textit{memory\_order}} e') \vee$$
$$(\forall e' \in (es \cap \textit{mem\_accesses } E). e' <_{X.\textit{memory\_order}} e)$$

# Axioms: Infinite Executions

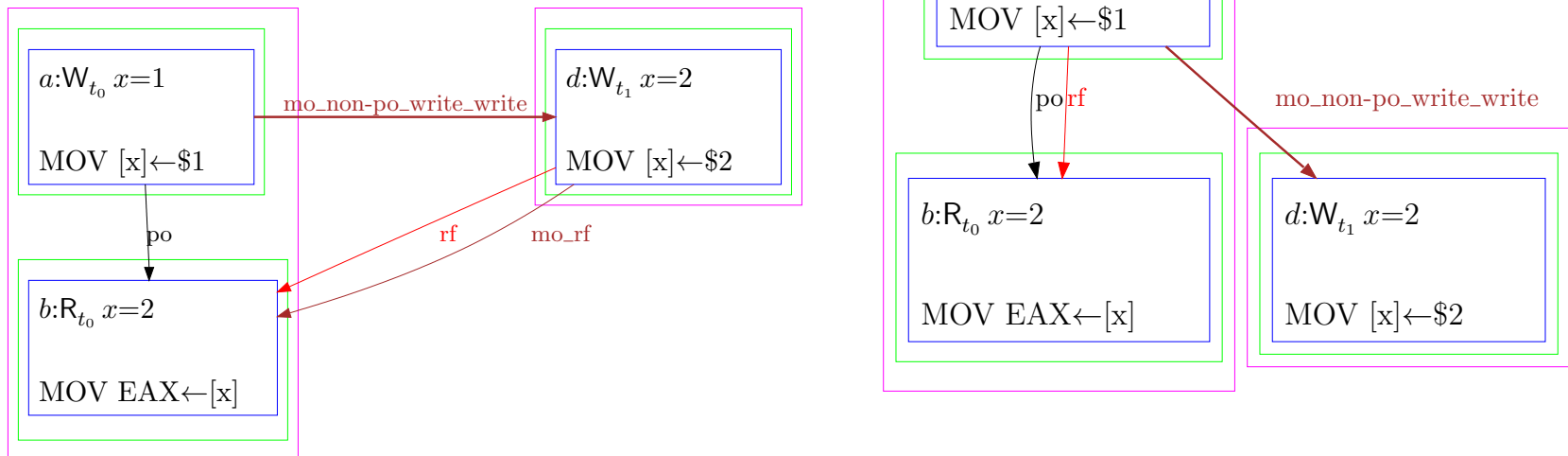
For this course, consider only finite executions ( $E$  with finite sets of events).

(In general, we require that the prefixes of the memory order are all finite, ensuring that there are no limit points, and, to ensure that each write eventually takes effect globally, there must not be an infinite set of reads unrelated to any particular write, all on the same memory location (this formalizes the SPARCv8 **Termination** axiom).)

Say  $\text{valid\_execution } E \ X$  iff all the above hold.



# Example



# Equivalence of the two models

Loosely:

**Theorem 1** *For any abstract-machine execution with*

- *atomic sets properly bracketed by lock/unlock pairs*
- *non- $\tau$ /L/U events  $E$*
- *ordered according to  $\text{po\_iico}$*

*there is an  $X$  such that  $\text{valid\_execution } E X$ , with the  $X.\text{memory\_order}$  the order in which machine memory reads and buffer flushes occurred.*

**Theorem 2** *For any axiomatic  $\text{valid\_execution } E X$ , there is some abstract-machine path which when  $\tau$ /L/U-erased has the same events (ordered according to  $\text{po\_iico}$  and with atomic sets properly bracketed by lock/unlock pairs) in which memory reads and buffer flushes respect  $X.\text{memory\_order}$ .*

# Hardware Models

x86 in detail

Why are industrial specs so often flawed?

A usable model: x86-TSO

Reasoning about x86-TSO code: races

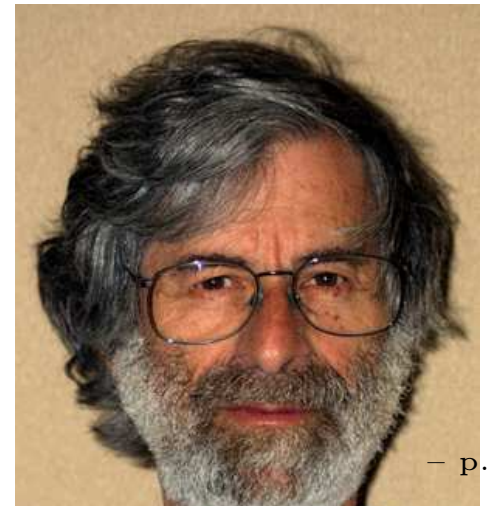
Power/ARM

SPARC, Alpha, Itanium

Programming Language Models (Java/C++)

*For some applications, achieving sequential consistency may not be worth the price of slowing down the processors. In this case, one must be aware that conventional methods for designing multiprocess algorithms cannot be relied upon to produce correctly executing programs. Protocols for synchronizing the processors must be designed at the lowest level of the machine instruction code, and verifying their correctness becomes a monumental task.*

Leslie Lamport, 1979



# Data Race Freedom (DRF)

Basic Principle (you'd hope):

*If a program has no data races in any sequentially consistent (SC) execution, then any relaxed-memory execution is equivalent to some sequentially consistent execution.*

NB: premise only involves SC execution.

# Data Race Freedom (DRF)

Basic Principle (you'd hope):

*If a program has no data races in any sequentially consistent (SC) execution, then any relaxed-memory execution is equivalent to some sequentially consistent execution.*

NB: premise only involves SC execution.

But what *is* a data race?

what does *equivalent* mean?

# What is a data race — first attempt

Suppose SC executions are traces of events

- $R_t x=v$  for thread  $t$  reading value  $v$  from address  $x$
- $W_t x=v$  for thread  $t$  writing value  $v$  to address  $x$

(erase  $\tau$ 's, and ignore lock/unlock/mfence for a moment)

Then say an SC execution has a data race if it contains a pair of adjacent accesses, by different threads, to the same location, that are not both reads:

- $\dots, R_{t_1} x=u, W_{t_2} x=v, \dots$
- $\dots, W_{t_1} x=u, R_{t_2} x=v, \dots$
- $\dots, W_{t_1} x=u, W_{t_2} x=v, \dots$

# What is a data race — for x86

1. Need not consider write/write pairs to be races
2. Have to consider SC semantics for LOCK'd instructions (and MFENCE), with events:
  - $L_t$  at the start of a LOCK'd instruction by  $t$
  - $U_t$  at the end of a LOCK'd instruction by  $t$
  - $B_t$  for an MFENCE by thread  $t$
3. Need not consider a LOCK'd read/any write pair to be a race

Say an *x86 data race* is an execution of one of these shapes:

- $\dots, R_{t_1} x=u, W_{t_2} x=v, \dots$
- $\dots, R_{t_1} x=u, L_{t_2}, \dots, W_{t_2} x=v, \dots$

(or v.v. No unlocks between the  $L_{t_2}$  and  $W_{t_2} x=v$ )



# DRF Principle for x86-TSO

Say a program is *data race free* (DRF) if no SC execution contains an x86 data race.

**Theorem 3 (DRF)** *If a program is DRF then any x86-TSO execution is equivalent to some SC execution.*

(where *equivalent* means that there is an SC execution with the same subsequence of writes and in which each read reads from the corresponding write)

Proof: via the x86-TSO axiomatic model

Scott Owens, ECOOP 2010



# Happens-Before Version

**Here:**

An SC race is two adjacent conflicting actions.

**In the setting of an axiomatic model:**

Often have a *happens before* partial order over events

...and a race is two conflicting actions that aren't ordered by happens-before

# What is a data race, again?

acquire\_mutex(l)

write  $x \leftarrow 1$

release\_mutex(l)

acquire\_mutex(l)

read x

release\_mutex(l)

# Simple Spinlock

acquire\_mutex(x)

*critical section*

release\_mutex(x)

# Simple Spinlock

```
while atomic_decrement(x) < 0 {  
    skip  
}
```

*critical section*

```
release_mutex(x)
```

Invariant:

lock taken if  $x \leq 0$

lock free if  $x=1$

# Simple Spinlock

```
while atomic_decrement(x) < 0 {  
    while x ≤ 0 { skip }  
}
```

*critical section*

```
release_mutex(x)
```

# Simple Spinlock

```
while atomic_decrement(x) < 0 {  
    while x ≤ 0 { skip }  
}
```

*critical section*

$x \leftarrow 1$       *OR*      atomic\_write(x, 1)

# Simple Spinlock

```
while atomic_decrement(x) < 0 {  
    while x ≤ 0 { skip }  
}
```

*critical section*

```
x ← 1
```



# Simple x86 Spinlock

The address of x is stored in register eax.

```
acquire:  LOCK DEC [eax]
```

```
          JNS enter
```

```
spin:    CMP [eax],0
```

```
          JLE spin
```

```
          JMP acquire
```

```
enter:
```

*critical section*

```
release: MOV [eax]←1
```

From Linux v2.6.24.7

NB: don't confuse levels — we're using x86 LOCK'd instructions in implementations of Linux spinlocks.

# Spinlock Example (SC)

```
while atomic_decrement(x) < 0 {  
    while x ≤ 0 { skip } }  
critical section  
x ← 1
```

Shared Memory

Thread 0

Thread 1

---

x = 1

# Spinlock Example (SC)

```
while atomic_decrement(x) < 0 {  
    while x ≤ 0 { skip } }  
critical section  
x ← 1
```

Shared Memory

Thread 0

Thread 1

---

x = 1

x = 0

acquire

# Spinlock Example (SC)

```
while atomic_decrement(x) < 0 {  
    while x ≤ 0 { skip } }  
critical section  
x ← 1
```

Shared Memory

Thread 0

Thread 1

---

x = 1

x = 0

x = 0

acquire

critical

# Spinlock Example (SC)

```
while atomic_decrement(x) < 0 {  
    while x ≤ 0 { skip } }  
critical section  
x ← -1
```

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	acquire	
x = 0	critical	
x = -1	critical	acquire

# Spinlock Example (SC)

```
while atomic_decrement(x) < 0 {  
    while x ≤ 0 { skip } }  
critical section  
x ← -1
```

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	acquire	
x = 0	critical	
x = -1	critical	acquire
x = -1	critical	spin, reading x

# Spinlock Example (SC)

```
while atomic_decrement(x) < 0 {  
    while x ≤ 0 { skip } }  
critical section  
x ← -1
```

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	acquire	
x = 0	critical	
x = -1	critical	acquire
x = -1	critical	spin, reading x
x = 1	release, writing x	

# Spinlock Example (SC)

```
while atomic_decrement(x) < 0 {  
    while x ≤ 0 { skip } }  
critical section  
x ← -1
```

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	acquire	
x = 0	critical	
x = -1	critical	acquire
x = -1	critical	spin, reading x
x = 1	release, writing x	
x = 1		read x



# Spinlock Example (SC)

```
while atomic_decrement(x) < 0 {  
    while x ≤ 0 { skip } }  
critical section  
x ← -1
```

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	acquire	
x = 0	critical	
x = -1	critical	acquire
x = -1	critical	spin, reading x
x = 1	release, writing x	
x = 1		read x
x = 0		acquire

# Spinlock SC Data Race

```
while atomic_decrement(x) < 0 {  
    while x ≤ 0 { skip } }  
critical section  
x ← -1
```

Shared Memory

Thread 0

Thread 1

---

x = 1

x = 0

acquire

x = 0

critical

x = -1

critical

acquire

x = -1

critical

spin, reading x

x = 1

release, writing x

# Spinlock Example (x86-TSO)

```
while atomic_decrement(x) < 0 {  
    while x ≤ 0 { skip } }  
critical section  
x ← 1
```

Shared Memory

Thread 0

Thread 1

---

x = 1

# Spinlock Example (x86-TSO)

```
while atomic_decrement(x) < 0 {  
    while x ≤ 0 { skip } }  
critical section  
x ← 1
```

Shared Memory

Thread 0

Thread 1

---

x = 1

x = 0

acquire

# Spinlock Example (x86-TSO)

```
while atomic_decrement(x) < 0 {  
    while x ≤ 0 { skip } }  
critical section  
x ← -1
```

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	acquire	
x = -1	critical	acquire

# Spinlock Example (x86-TSO)

```
while atomic_decrement(x) < 0 {  
    while x ≤ 0 { skip } }  
critical section  
x ← -1
```

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	acquire	
x = -1	critical	acquire
x = -1	critical	spin, reading x

# Spinlock Example (x86-TSO)

```
while atomic_decrement(x) < 0 {  
    while x ≤ 0 { skip } }  
critical section  
x ← -1
```

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	acquire	
x = -1	critical	acquire
x = -1	critical	spin, reading x
x = -1	release, writing x to buffer	

# Spinlock Example (x86-TSO)

```
while atomic_decrement(x) < 0 {  
    while x ≤ 0 { skip } }  
critical section  
x ← -1
```

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	acquire	
x = -1	critical	acquire
x = -1	critical	spin, reading x
x = -1	release, writing x to buffer	
x = -1	...	spin, reading x



# Spinlock Example (x86-TSO)

```
while atomic_decrement(x) < 0 {  
    while x ≤ 0 { skip } }  
critical section  
x ← 1
```

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	acquire	
x = -1	critical	acquire
x = -1	critical	spin, reading x
x = -1	release, writing x to buffer	
x = -1	...	spin, reading x
x = 1	write x from buffer	

# Spinlock Example (x86-TSO)

```
while atomic_decrement(x) < 0 {  
    while x ≤ 0 { skip } }  
critical section  
x ← 1
```

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	acquire	
x = -1	critical	acquire
x = -1	critical	spin, reading x
x = -1	release, writing x to buffer	
x = -1	...	spin, reading x
x = 1	write x from buffer	
x = 1		read x

# Spinlock Example (x86-TSO)

```
while atomic_decrement(x) < 0 {  
    while x ≤ 0 { skip } }  
critical section  
x ← 1
```

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	acquire	
x = -1	critical	acquire
x = -1	critical	spin, reading x
x = -1	release, writing x to buffer	
x = -1	...	spin, reading x
x = 1	write x from buffer	
x = 1		read x
x = 0		acquire

# Triangular Races (Owens)

- Read/write data race
- Only if there is a bufferable write preceding the read

Triangular race

⋮	$y \leftarrow v_2$
⋮	⋮
$x \leftarrow v_1$	$x$
⋮	⋮

# Triangular Races

- Read/write data race
- Only if there is a bufferable write preceding the read

Triangular race

⋮	$y \leftarrow v_2$
⋮	⋮
$X \leftarrow v_1$	$X$
⋮	⋮

Not triangular race

⋮	$y \leftarrow v_2$
⋮	⋮
$X \leftarrow v_1$	$X \leftarrow w$
⋮	⋮

# Triangular Races

- Read/write data race
- Only if there is a bufferable write preceding the read

Triangular race

⋮	$y \leftarrow v_2$
⋮	⋮
$X \leftarrow v_1$	<b>X</b>
⋮	⋮

Not triangular race

⋮	$y \leftarrow v_2$
⋮	<b>mfence</b>
$X \leftarrow v_1$	<b>X</b>
⋮	⋮

# Triangular Races

- Read/write data race
- Only if there is a bufferable write preceding the read

Triangular race

⋮	$y \leftarrow v_2$
⋮	⋮
$x \leftarrow v_1$	<b>x</b>
⋮	⋮

Not triangular race

⋮	$y \leftarrow v_2$
⋮	⋮
$x \leftarrow v_1$	<b>lock x</b>
⋮	⋮

# Triangular Races

- Read/write data race
- Only if there is a bufferable write preceding the read

Triangular race

⋮	$y \leftarrow v_2$
⋮	⋮
$X \leftarrow v_1$	<b>X</b>
⋮	⋮

Not triangular race

⋮	<b>lock</b> $y \leftarrow v_2$
⋮	⋮
$X \leftarrow v_1$	<b>X</b>
⋮	⋮



# Triangular Races

- Read/write data race
- Only if there is a bufferable write preceding the read

Triangular race

⋮		$y \leftarrow v_2$
⋮		⋮
$X \leftarrow v_1$		$X$
⋮		⋮

Triangular race

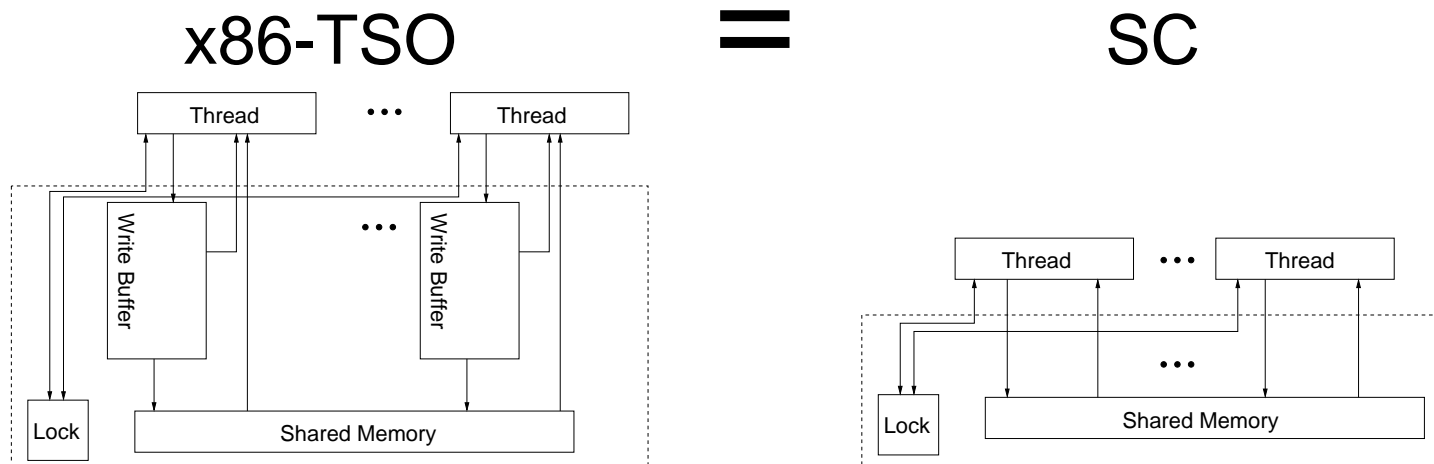
⋮		$y \leftarrow v_2$
⋮		⋮
<b>lock</b> $X \leftarrow v_1$		$X$
⋮		⋮

# TRF Principle for x86-TSO

Say a program is *triangular race free (TRF)* if no SC execution has a triangular race.

**Theorem 4 (TRF)** *If a program is TRF then any x86-TSO execution is equivalent to some SC execution.*

*If a program has no triangular races when run on a sequentially consistent memory, then*



# Spinlock Data Race

```
while atomic_decrement(x) < 0 {  
    while x ≤ 0 { skip }  
    critical section  
    x ← 1
```

---

x = 1

x = 0      acquire

x = -1     critical                  acquire

x = -1     critical                  spin, reading x

x = 1      release, writing x

● acquire's writes are locked

# Program Correctness

**Theorem 5** *Any well-synchronized program that uses the spinlock correctly is TRF.*

**Theorem 6** *Spinlock-enforced critical sections provide mutual exclusion.*

# Other Applications

A concurrency bug in the HotSpot JVM

- Found by Dave Dice (Sun) in Nov. 2009
- `java.util.concurrent.LockSupport` ('Parker')
- Platform specific C++
- Rare hung thread
- Since "day-one" (missing MFENCE)
- Simple explanation in terms of TRF

Also: Ticketed spinlock, Linux SeqLocks, Double-checked locking

# Reflections

We've introduced a plausible model, x86-TSO.

Usable:

- as spec to test h/w against
- to give a solid intuition for systems programmers
- to develop reasoning tools above
- to develop code testing tools above (daemonized emulator)

In terms of that model, we can clearly see why (and indeed *prove*) that that Linux spinlock optimisation is correct.