

x86

A Cautionary Tale

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

'Processor Ordering' model,
informal prose

Example: Linux Kernel mailing list, Nov–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" asm code, to 1 tick for a simple "mov1 \$0,%0" instruction, a huge gain. Later, he reported that Ingo Molnar noticed a 4% speed-up in a benchmark test, making the optimization very valuable. Ingo also added that the same optimization cropped up in the FreeBSD mailing list a few days previously. But Linus Torvalds poured cold water on the whole thing, saying:

It does NOT WORK!

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

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

The issue is not writes being issued in order (although

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 is 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 \wedge Thread 1:EBX=0	

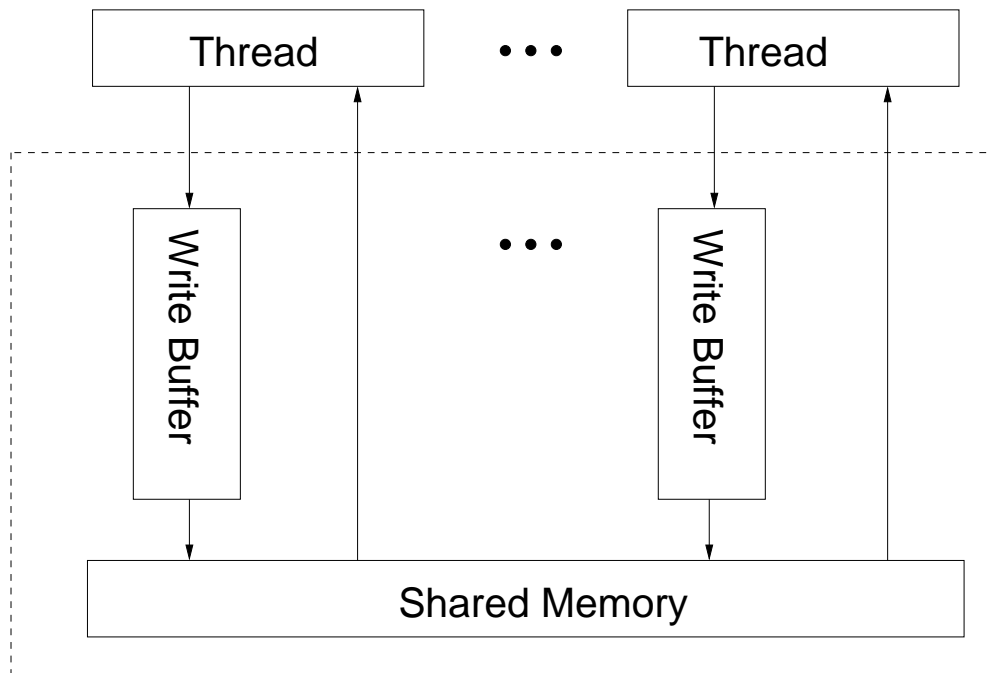
P3. Loads may be reordered with older stores to different locations but not with older stores to the same location

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 ∧ Thread 1:EBX=0	

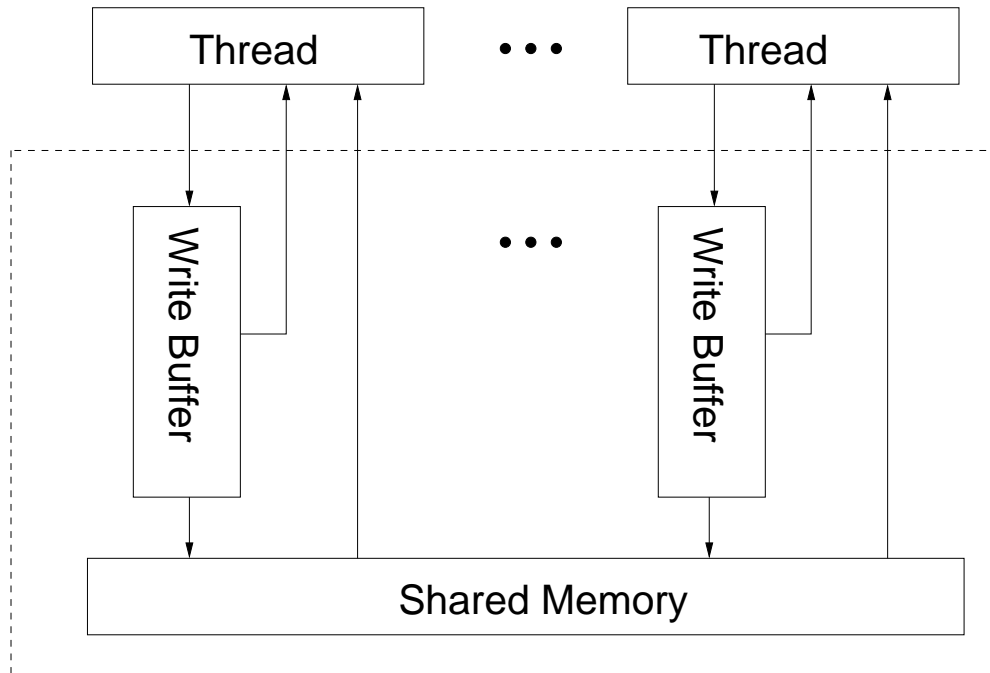


Litmus Test 2.4. Intra-processor forwarding is allowed

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

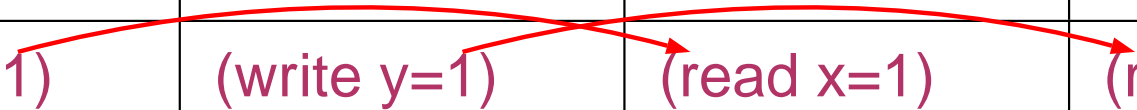
Thread 0	Thread 1
MOV [x]←1 (write x=1)	MOV [y]←1 (write y=1)
MOV EAX←[x] (read x=1)	MOV ECX←[y] (read y=1)
MOV EBX←[y] (read y=0)	MOV EDX←[x] (read x=0)
Allowed Final State: Thread 0:EBX=0 \wedge Thread 1:EDX=0 Thread 0:EAX=1 \wedge Thread 1:ECX=1	



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?			



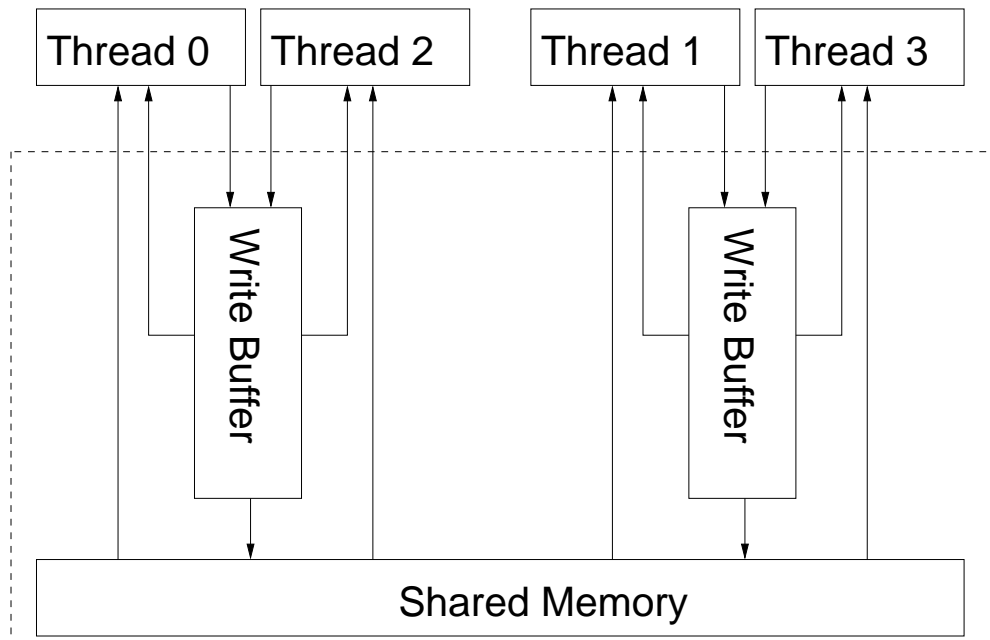
Problem 1: Weakness

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?

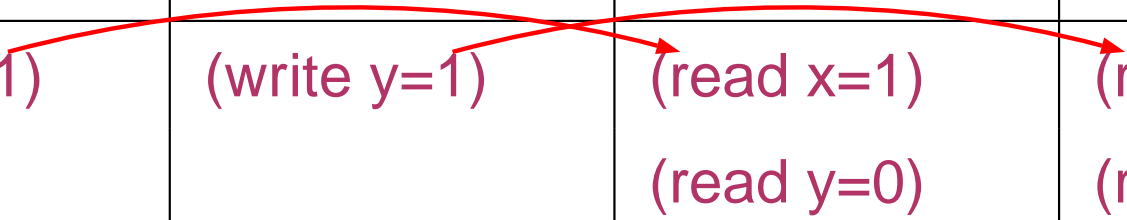
Microarchitecturally plausible? yes, e.g. with shared store buffers



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’, if one reads ‘ordered’ as referring to a single per-execution partial order.

(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 – Oct. 2015

Intel SDM rev. 29–55 and AMD 3.17–3.25

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/content/www/us/en/processors/architecture>
(rev. 35 on 6/10/2010, rev. 55 on 3/10/2015).

See especially SDM Vol. 3A, Ch. 8, Sections 8.1–8.3

AMD:

<http://developer.amd.com/Resources/documentation/guides/Pages/>
(rev. 3.17 on 6/10/2010, rev. 3.25 on 3/10/2015).

See especially APM Vol. 2, Ch. 7, Sections 7.1–7.2

Inventing a Usable Abstraction

Have to be:

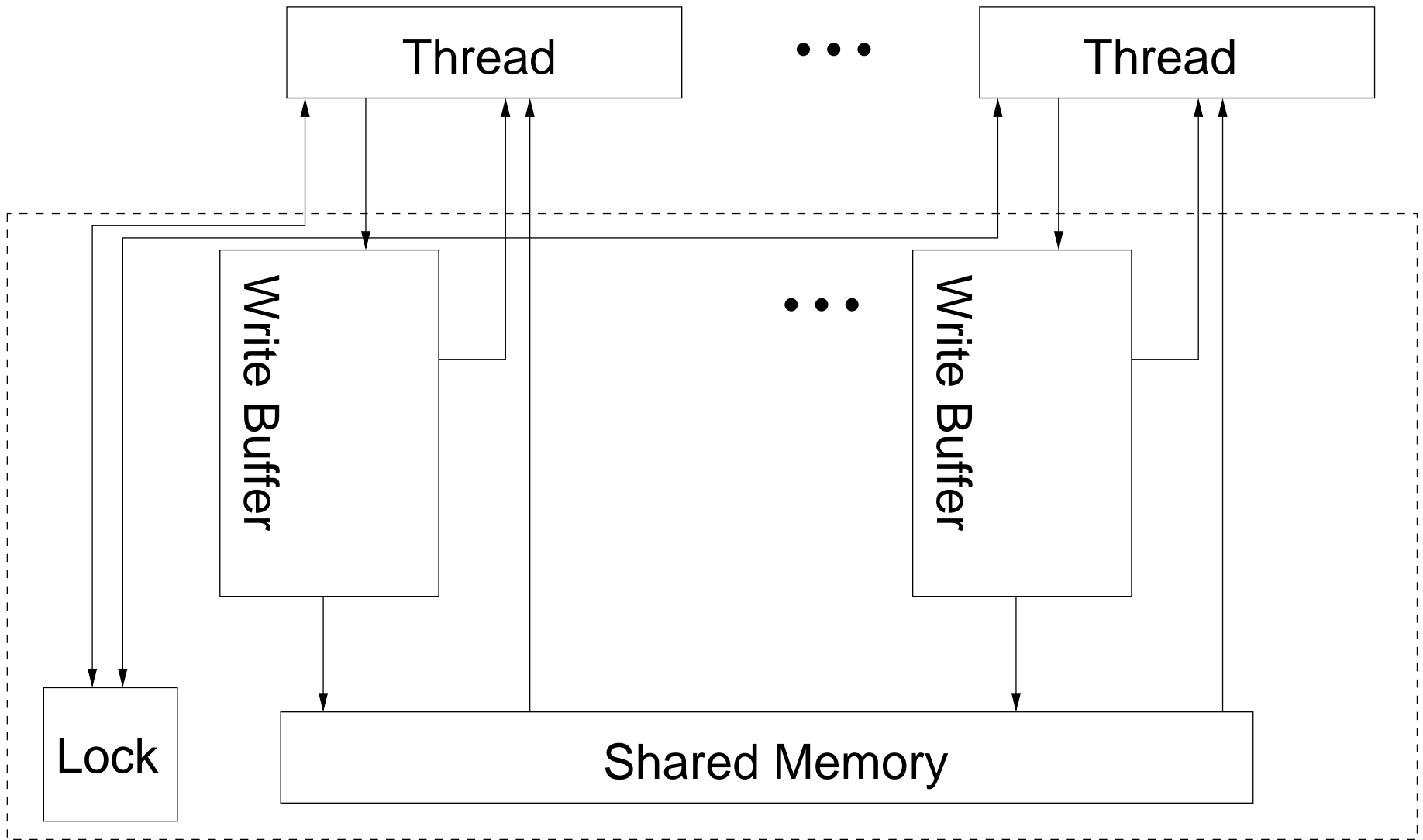
- Unambiguous
- Sound w.r.t. experimentally observable behaviour
- Easy to understand
- Consistent with what we know of vendors intentions
- Consistent with expert-programmer reasoning

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



x86-TSO Abstract Machine

As for Sequential Consistency, we separate the programming language (here, really the *instruction semantics*) and the x86-TSO *memory model*.

(the memory model describes the behaviour of the stuff in the dotted box)

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

x86-TSO Abstract Machine: Interface

Labels

$l ::= t:W \ x=v$	a write of value v to address x by thread t
$t:R \ x=v$	a read of v from x by t
$t:\tau$	an internal action of the thread
$t:\tau \ x=v$	an internal action of the abstract machine, moving $x = v$ from the write buffer on t to shared memory
$t:B$	an MFENCE memory barrier by t
$t:L$	start of an instruction with LOCK prefix by t
$t:U$	end of an instruction with LOCK prefix by t

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

An *x86-TSO abstract machine state* m is a record

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

Here:

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

Write m_0 for the initial state with $m.M = M_0$, $s.B$ empty for all threads, and $m.L = \text{None}$ (lock not taken).

x86-TSO Abstract Machine: Auxiliary Definitions

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

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

x86-TSO Abstract Machine: Behaviour

RM: Read from memory

$\text{not_blocked}(m, t)$

$m.M(x) = v$

$\text{no_pending}(m.B(t), x)$

$m \xrightarrow{t:R\ x=v} m$

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

$\text{not_blocked}(m, t)$

$\exists b_1 b_2. m.B(t) = b_1 \text{ ++ } [(x, v)] \text{ ++ } b_2$

$\text{no_pending}(b_1, x)$

$$m \xrightarrow{t:\text{R } x=v} m$$

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

$$m \quad \frac{t:W \ x=v}{\rightarrow}$$

$$m \oplus \langle [B := m.B \oplus (t \mapsto ([(x, v)] \dashv\vdash m.B(t)))] \rangle$$

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

$\text{not_blocked}(m, t)$

$m.B(t) = b ++ [(x, v)]$

$m \xrightarrow{t:\tau \ x=v}$

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

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

...rules for lock, unlock, and mfence later

Notation Reference

SOME and NONE construct optional values

(\cdot, \cdot) builds tuples

$[\]$ builds lists

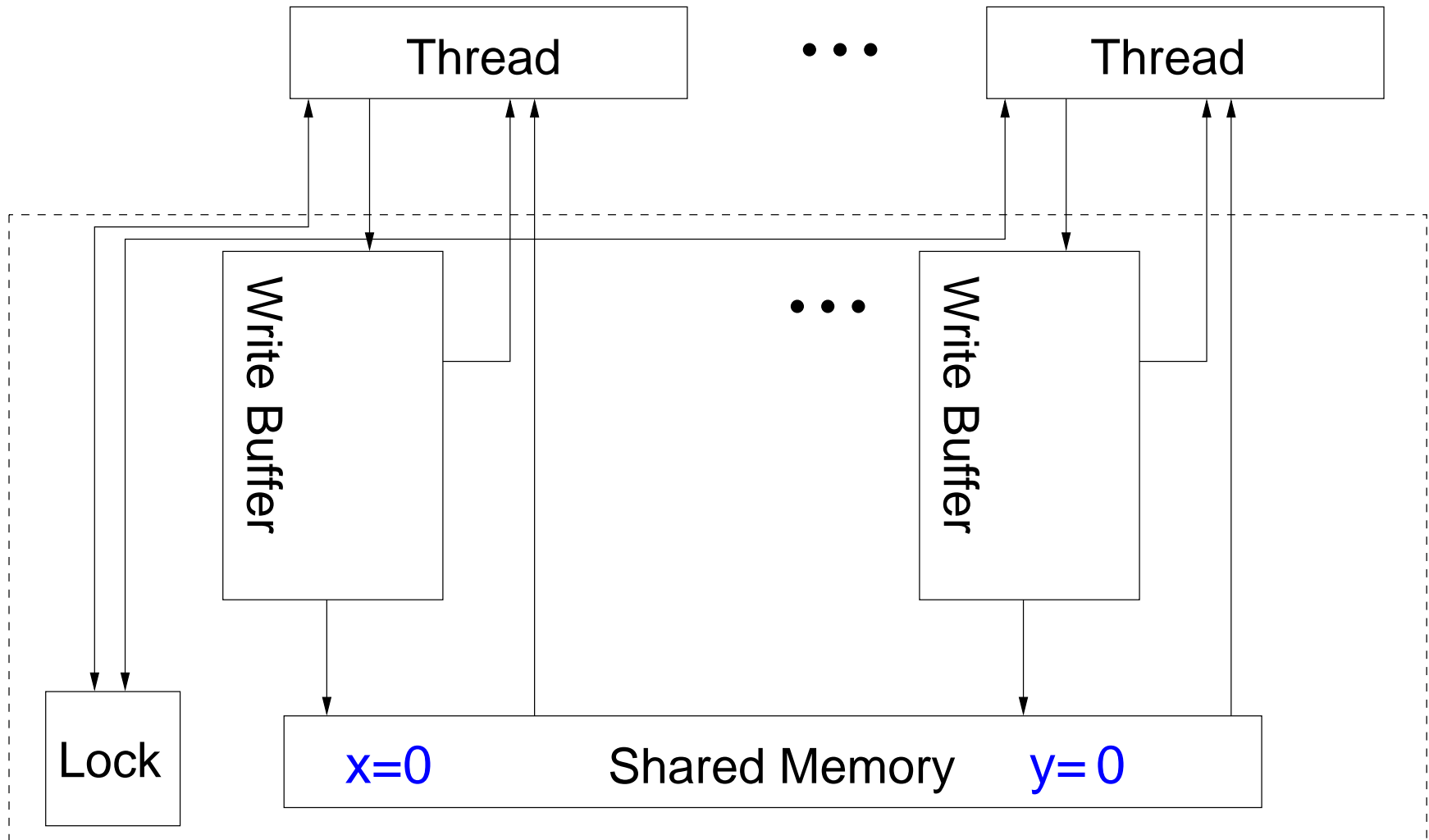
++ appends lists

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

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

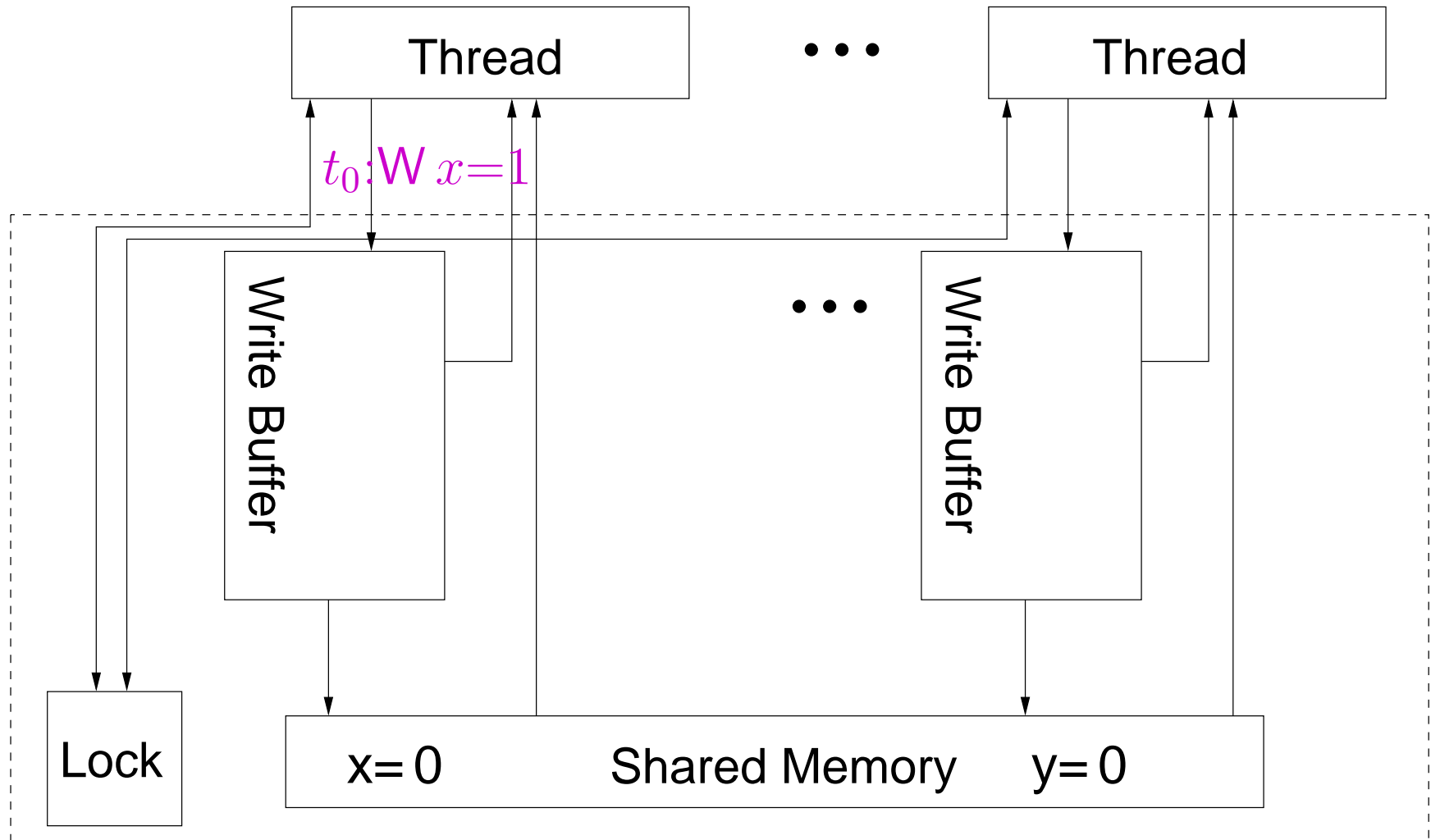
First Example, Revisited

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)



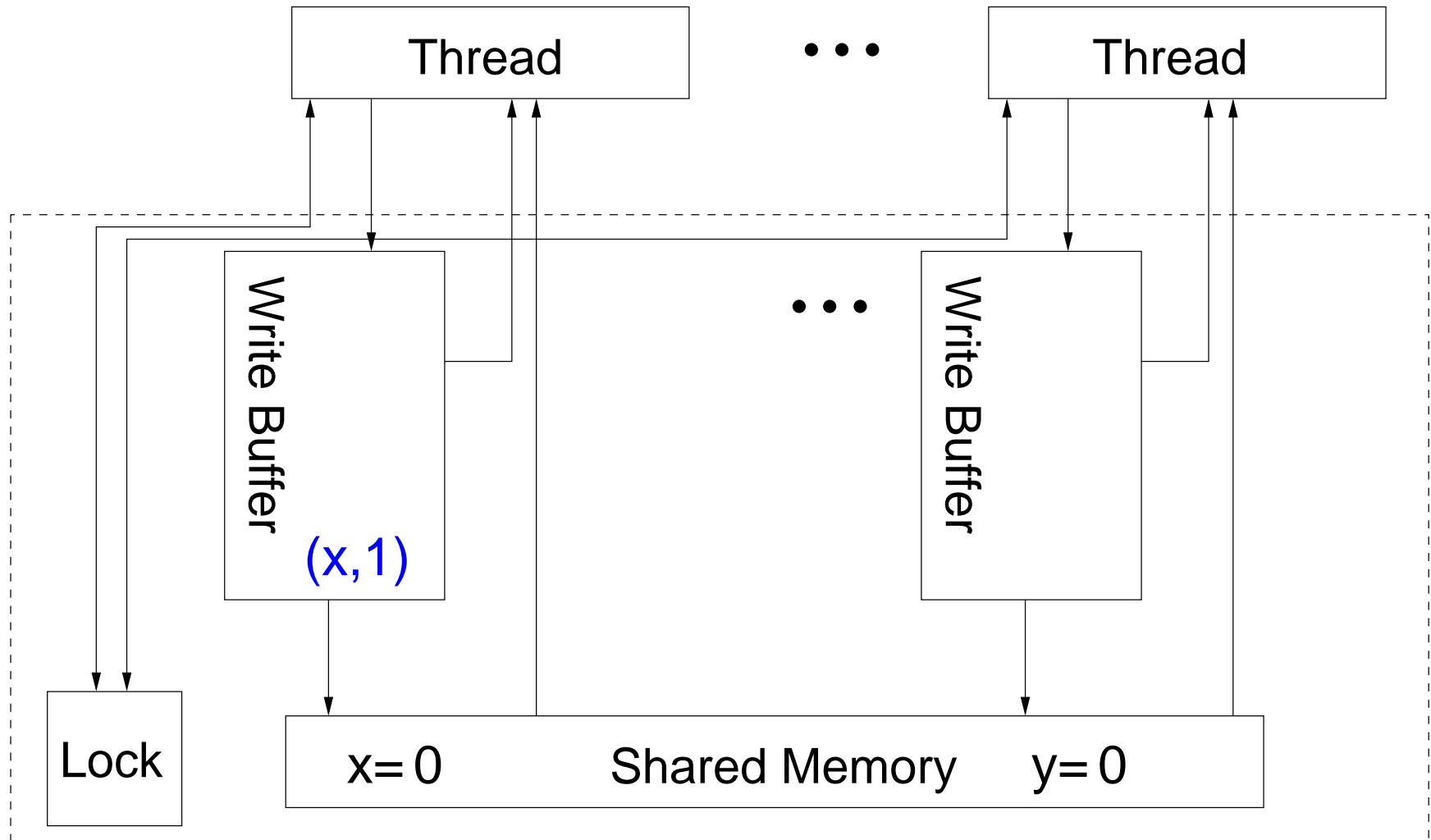
First Example, Revisited

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)



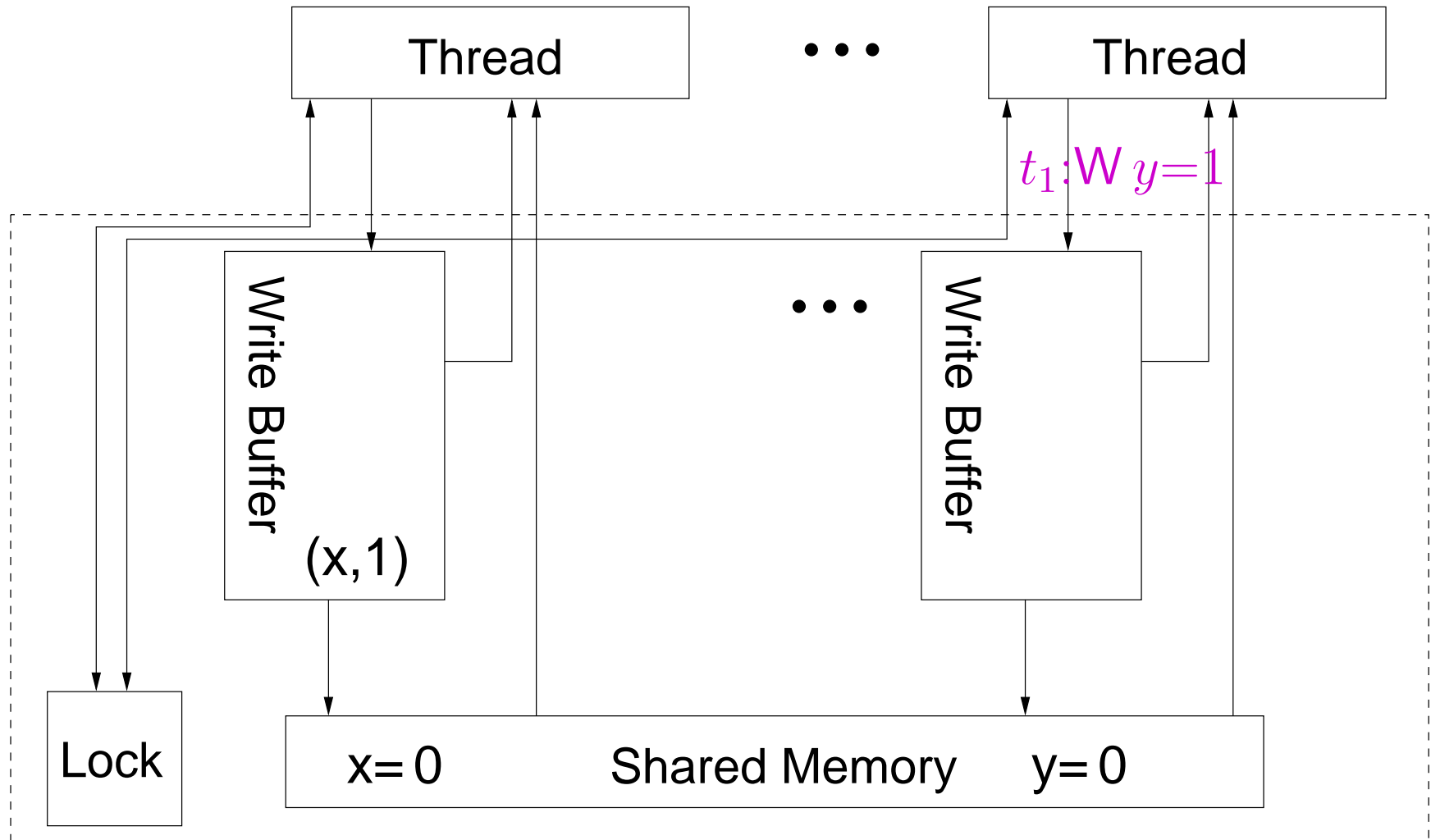
First Example, Revisited

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)



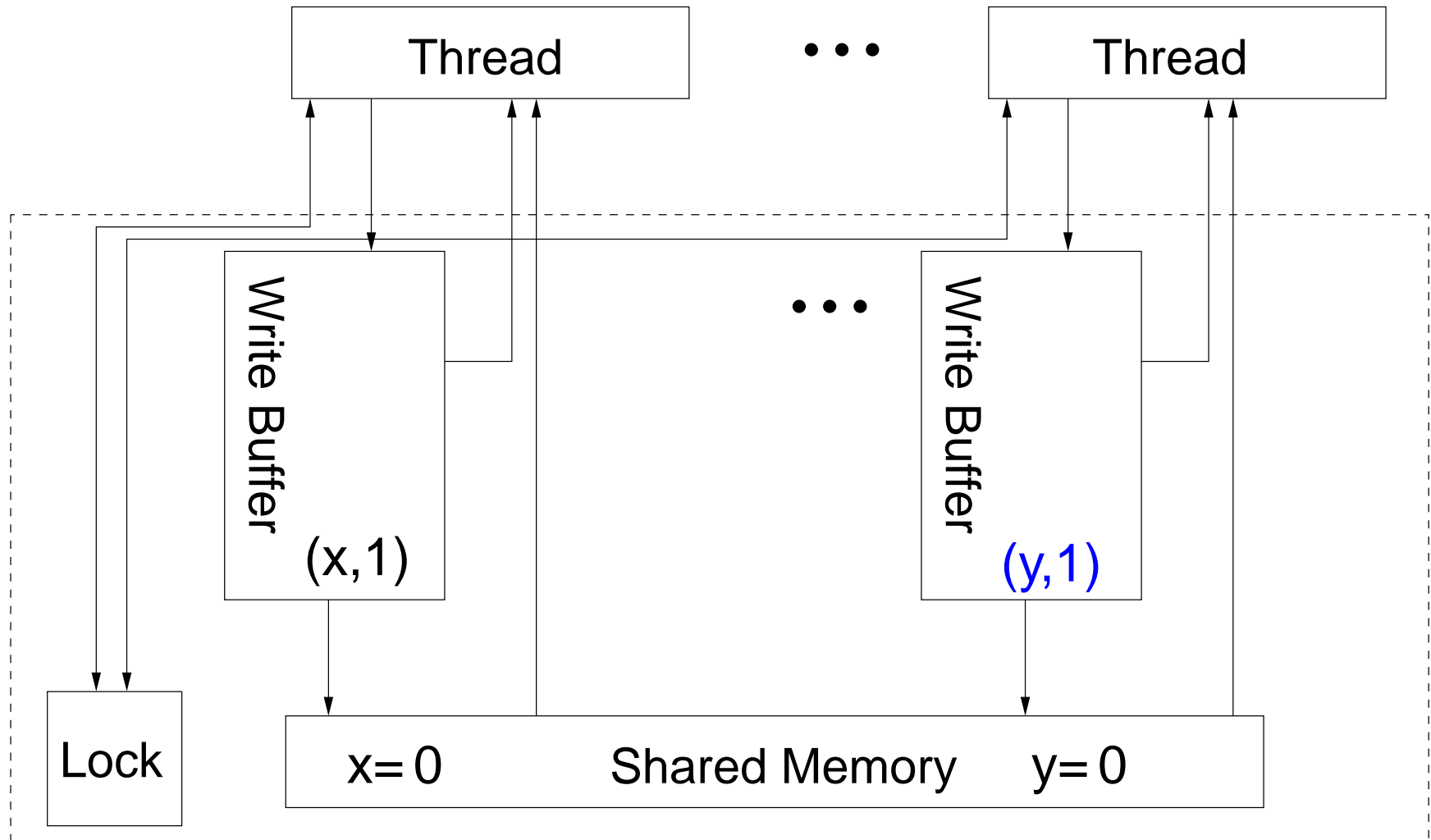
First Example, Revisited

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)



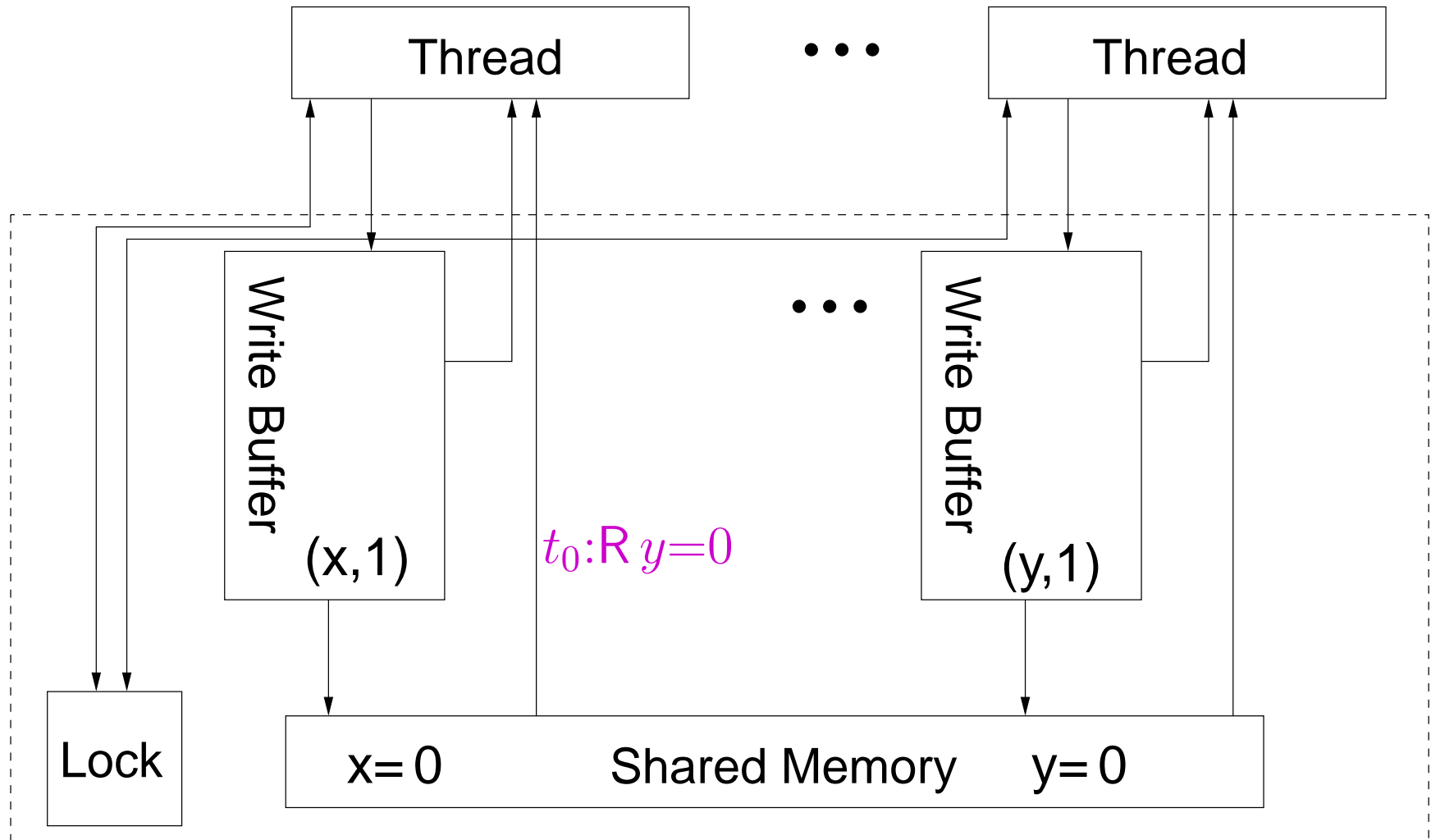
First Example, Revisited

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)



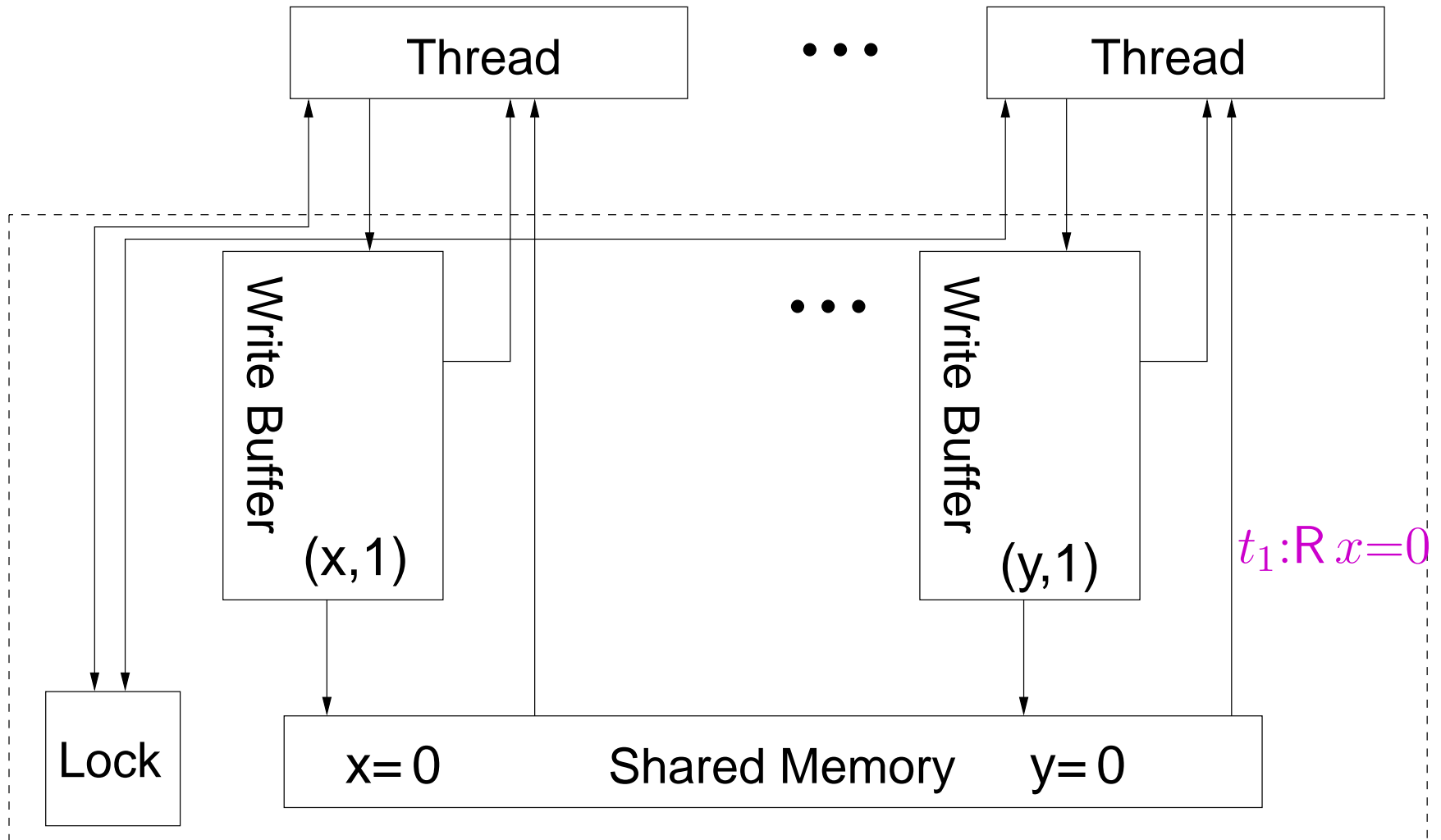
First Example, Revisited

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)



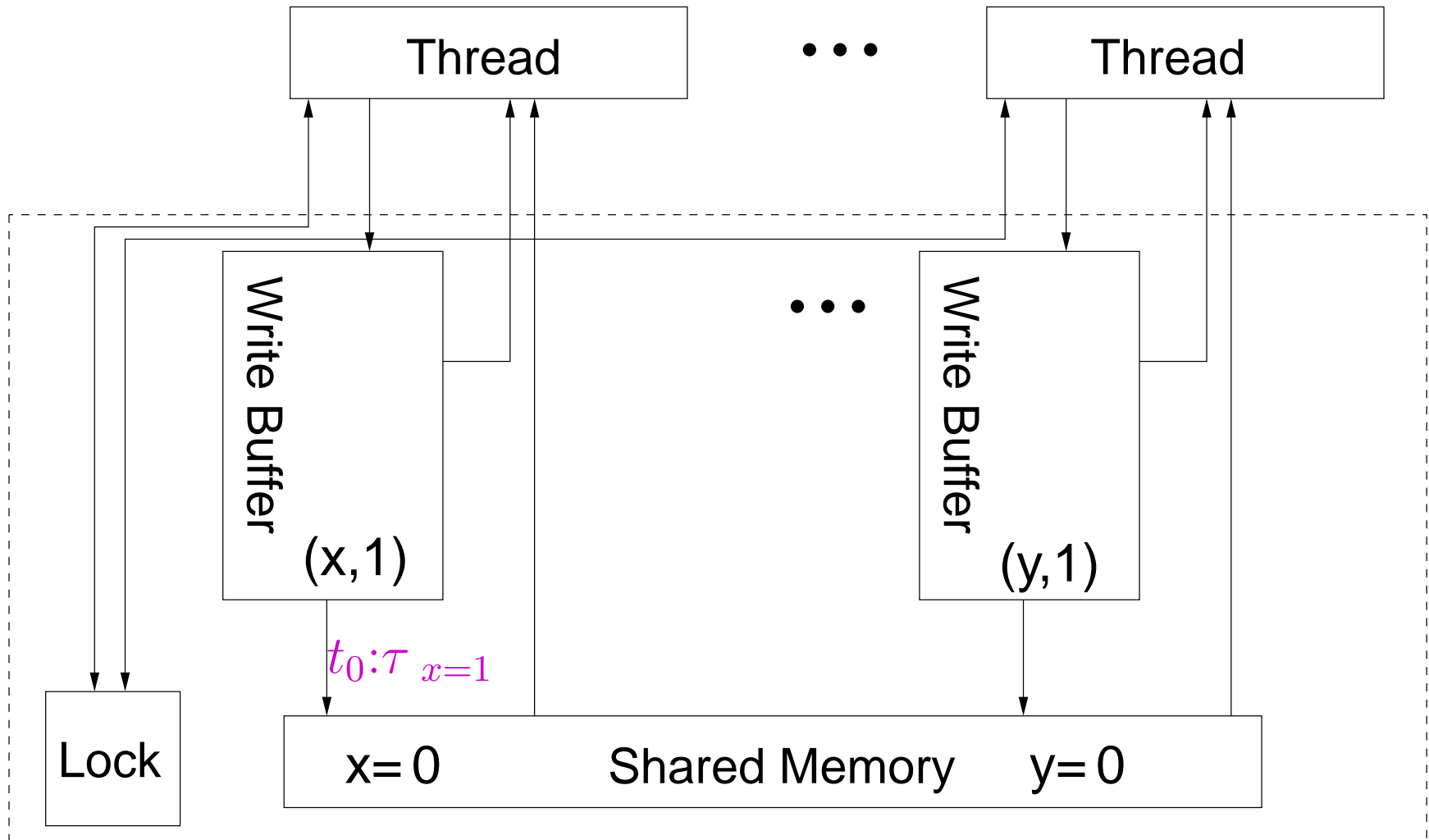
First Example, Revisited

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)



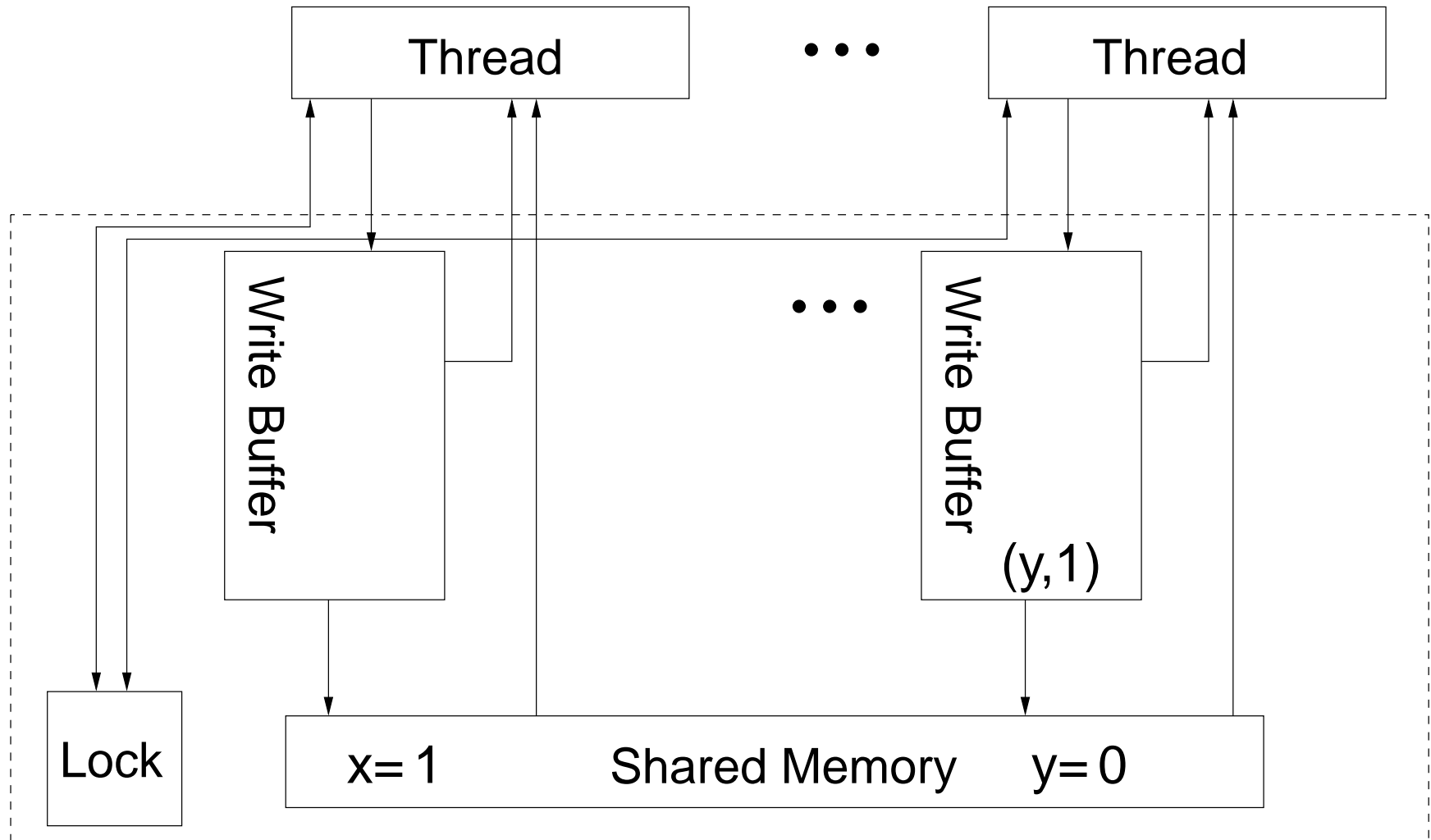
First Example, Revisited

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)



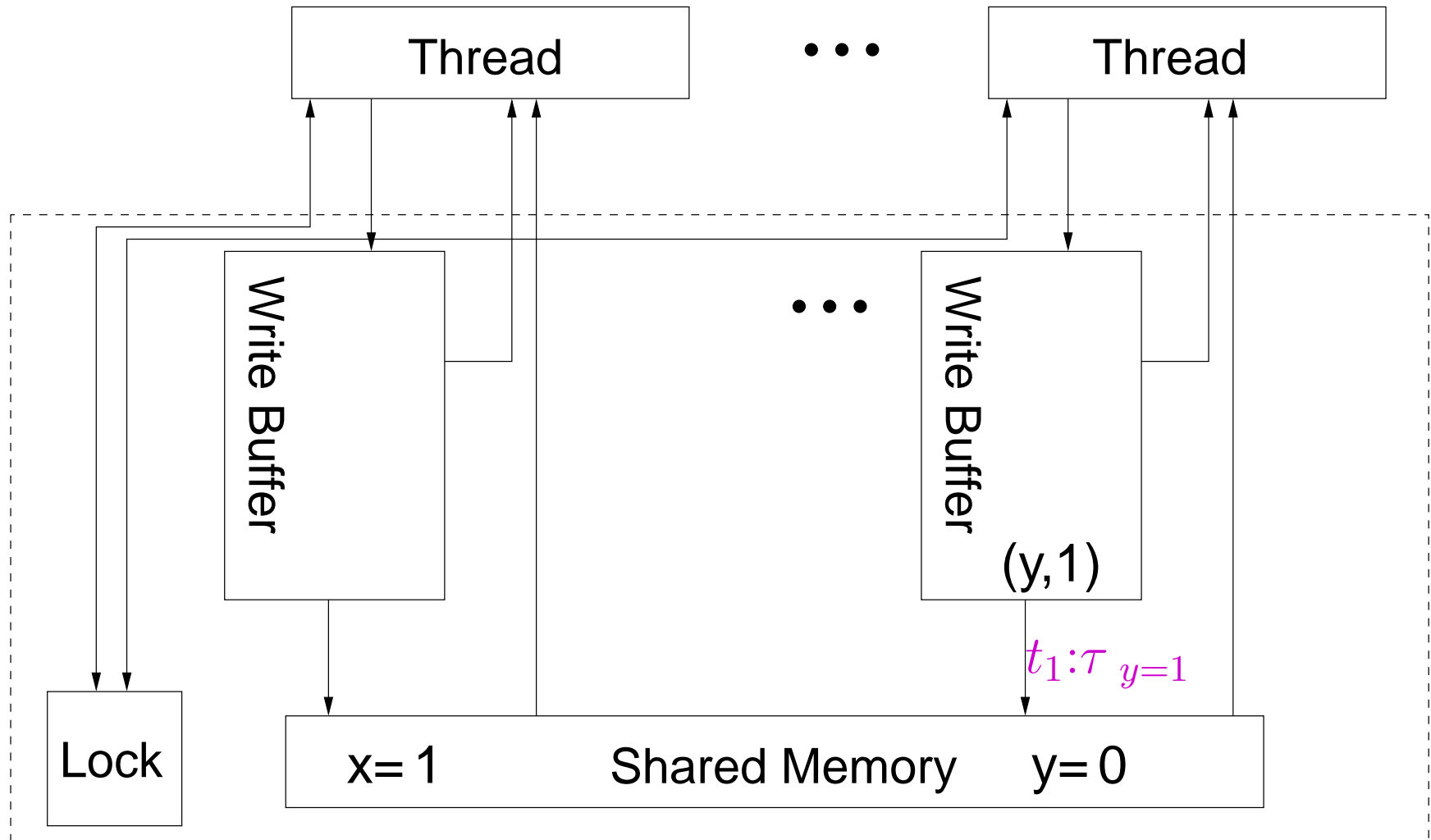
First Example, Revisited

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)



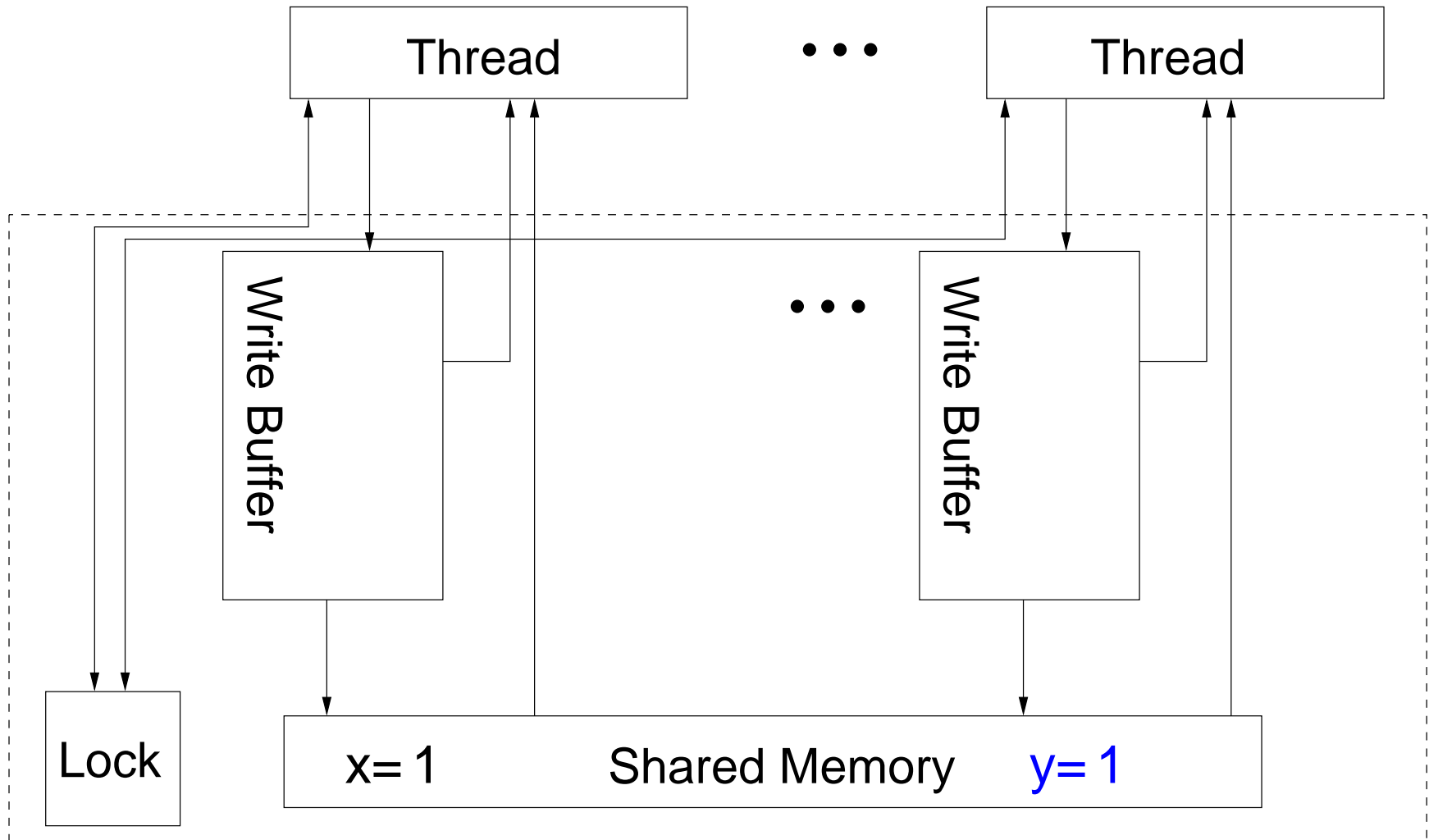
First Example, Revisited

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)



Strengthening the model: the MFENCE memory barrier

MFENCE: an x86 assembly instruction

...waits for local write buffer to drain (or forces it – is that an observable distinction?)

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)
Forbidden Final State: Thread 0:EAX=0 \wedge Thread 1:EBX=0	

NB: no inter-thread synchronisation

x86-TSO Abstract Machine: Behaviour

B: Barrier

$$\frac{m.B(t) = []}{m \xrightarrow{t:B} m}$$

If t 's store buffer is empty, it can execute an MFENCE (otherwise the MFENCE blocks until that becomes true).

Adding MFENCE to our tiny language

Syntax:

$$\begin{array}{l} \textit{statement}, s \quad ::= \quad \text{statement} \\ \quad \quad \quad \quad \quad \quad | \quad \dots \\ \quad \quad \quad \quad \quad \quad | \quad \text{mfence} \quad \quad \text{mfence} \end{array}$$

Threadwise Semantics:

$$\frac{}{t : \langle \text{mfence}, R \rangle \xrightarrow{t:\text{B}} t : \langle \text{skip}, R \rangle} \quad \text{T_MFENCE}$$

Defining a whole-system x86-TSO Semantics

An *x86-TSO system state* $Stso = \langle P, m_{tso} \rangle$ is a pair of a process and an x86-TSO abstract machine state $mtso$.

$Stso \xrightarrow{l} Stso'$ system $Stso$ does l to become $Stso'$

$$\frac{\begin{array}{l} P \xrightarrow{l} P' \\ m_{tso} \xrightarrow{l} m_{tso}' \end{array}}{\langle P, m_{tso} \rangle \xrightarrow{l} \langle P', m_{tso}' \rangle} \quad \text{STSO_ACCESS}$$

$$\frac{P \xrightarrow{t:\tau} P'}{\langle P, m_{tso} \rangle \xrightarrow{t:\tau} \langle P', m_{tso} \rangle} \quad \text{STSO_INTERNAL_PROG}$$

$$\frac{m_{tso} \xrightarrow{t:\tau_{x=v}} m_{tso}'}{\langle P, m_{tso} \rangle \xrightarrow{t:\tau_{x=v}} \langle P, m_{tso}' \rangle} \quad \text{STSO_INTERNAL_MEM}$$

Does MFENCE restore SC?

For any process P , define $\text{insert_fences}(P)$ to be the process with all $s_1 ; s_2$ replaced by $s_1 ; \text{mfence} ; s_2$ (formally define this recursively over statements, threads, and processes).

For any trace l_1, \dots, l_k of an x86-TSO system state, define $\text{erase_flushes}(l_1, \dots, l_k)$ to be the trace with all $t:\tau_{x=v}$ labels erased (formally define this recursively over the list of labels).

Theorem 1 (?) *For all processes P ,*

$$\text{traces}(\langle P, m_0 \rangle) = \text{erase_flushes}(\text{traces}(\langle \text{insert_fences}(P), m_{\text{tso}0} \rangle))$$

Adding Read-Modify-Write instructions

x86 is not RISC – there are many instructions that read and write memory, e.g.

Thread 0	Thread 1
INC x	INC x

Adding Read-Modify-Write 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)

Adding Read-Modify-Write 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	

Adding Read-Modify-Write 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

Being able to do that atomically is important for many low-level algorithms. On x86 can also do for other sizes, including for 8B and 16B adjacent-doublesize quantities

CAS

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...

Adding LOCK'd instructions to the model

1. extend the tiny language syntax
2. extend the tiny language semantics so that whatever represents a LOCK;INC x will (in thread t) do
 - (a) $t:L$
 - (b) $t:R\ x=v$ for an arbitrary v
 - (c) $t:W\ x=(v + 1)$
 - (d) $t:U$
3. extend the x86-TSO abstract machine with rules for the LOCK and UNLOCK transitions

(this lets us reuse the semantics for INC for LOCK;INC, and to do so uniformly for all RMWs)

x86-TSO Abstract Machine: Behaviour

L: Lock

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

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

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

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 $t:\tau_{x=v}$ steps to empty the buffer and then proceed.

x86-TSO Abstract Machine: Behaviour

U: Unlock

$$m.L = \mathbf{SOME}(t)$$

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

$$m \xrightarrow{t:\mathbf{U}} m \oplus \langle [L := \mathbf{NONE}] \rangle$$

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

Restoring SC with RMWs

CAS cost

From Paul McKenney

(<http://www2.rdrop.com/~paulmck/RCU/>):

16-CPU 2.8GHz Intel X5550 (Nehalem) System

Operation	Cost (ns)	Ratio
Clock period	0.4	1
"Best-case" CAS	12.2	33.8
Best-case lock	25.6	71.2
Single cache miss	12.9	35.8
CAS cache miss	7.0	19.4
Single cache miss (off-core)	31.2	86.6
CAS cache miss (off-core)	31.2	86.5
Single cache miss (off-socket)	92.4	256.7
CAS cache miss (off-socket)	95.9	266.4

Want to be here!

Heavily optimized reader-writer lock might get here for readers (but too bad about those poor writers...)

Typical synchronization mechanisms do this a lot, plus suffer from contention

NB: 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.

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

Also no fairness properties: finite executions only, in this course.

x86-TSO vs SPARC TSO

x86-TSO based on SPARC TSO

SPARC defined

- TSO (Total Store Order)
- PSO (Partial Store Order)
- RMO (Relaxed Memory Order)

But as far as we know, only TSO has really been used (implementations have not been as weak as PSO/RMO or software has turned them off).

The SPARC Architecture Manual, Version 8, 1992.

<http://sparc.org/wp-content/uploads/2014/01/v8.pdf.gz> App. K defines TSO and PSO.

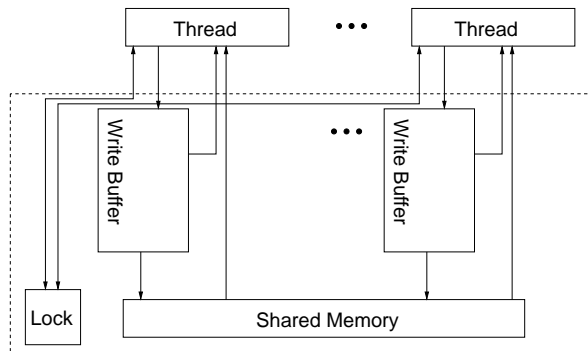
Version 9, Revision SAV09R1459912. 1994

<http://sparc.org/wp-content/uploads/2014/01/SPARCV9.pdf.gz> Ch. 8 and App. D define TSO, PSO, RMO

(in an axiomatic style – see later)

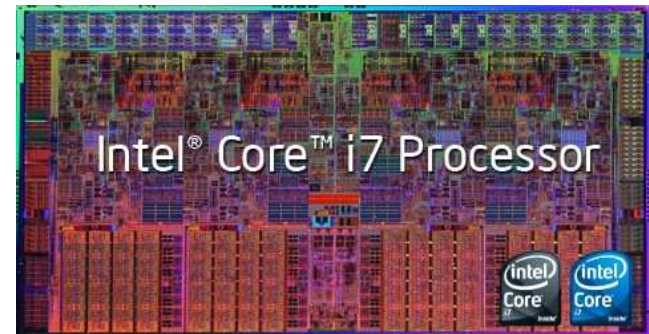
NB: This is an *Abstract Machine*

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

x86 spinlock example

Adding primitive mutexes to our source language

Statements $s ::= \dots \mid \text{lock } x \mid \text{unlock } x$

Say lock free if it holds 0, 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.

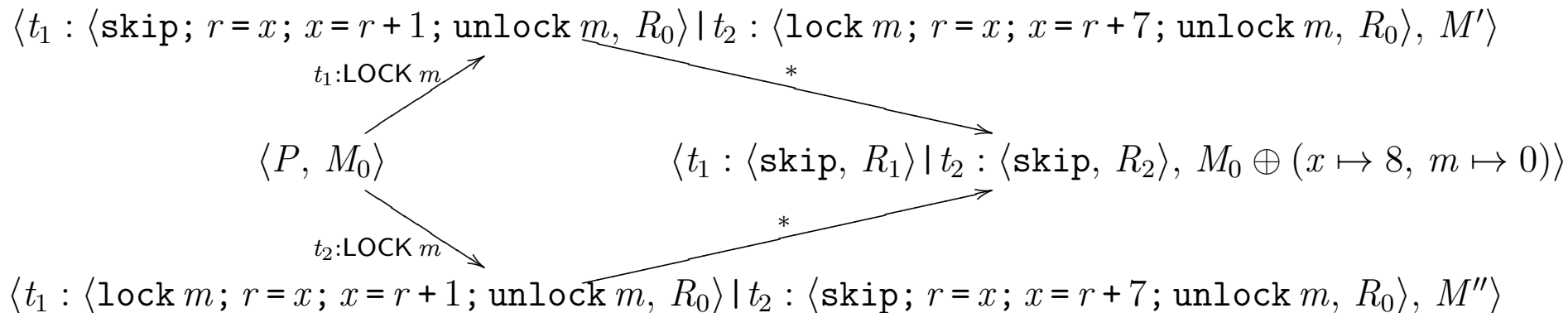
Record of which thread is holding a locked lock? Re-entrancy?

Using a Mutex

Consider

$$P = \begin{array}{l} t_1 : \langle \text{lock } m; r = x; x = r + 1; \text{unlock } m, R_0 \rangle \\ | \\ t_2 : \langle \text{lock } m; r = x; x = r + 7; \text{unlock } m, R_0 \rangle \end{array}$$

in the initial store M_0 :



where $M' = M_0 \oplus (m \mapsto 1)$

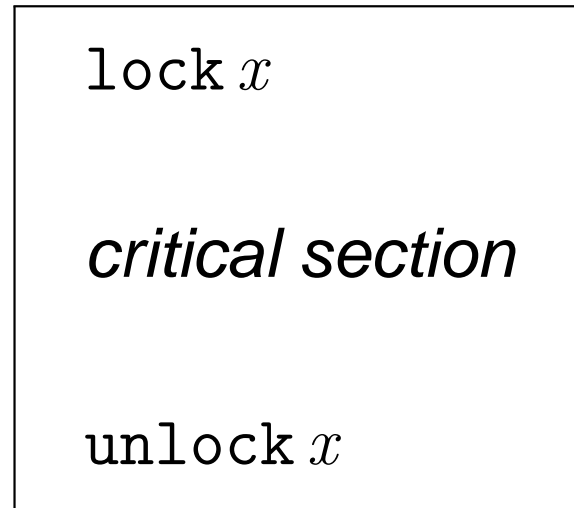
Deadlock

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

$$P = \begin{array}{l} t_1 : \langle \text{lock } m_1; \text{lock } m_2; x = 1; \text{unlock } m_1; \text{unlock } m_2, R_0 \rangle \\ | \\ t_2 : \langle \text{lock } m_2; \text{lock } m_1; x = 2; \text{unlock } m_1; \text{unlock } m_2, R_0 \rangle \end{array}$$

Implementing mutexes with simple x86 spinlocks

Implementing the language-level mutex with x86-level simple spinlocks



Implementing mutexes with simple x86 spinlocks

```
while atomic_decrement(x) < 0 {  
    skip  
}  
  
critical section  
  
unlock(x)
```

Invariant:

lock taken if $x \leq 0$

lock free if $x=1$

(NB: different internal representation from high-level semantics)

Implementing mutexes with simple x86 spinlocks

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

critical section

unlock(x)

Implementing mutexes with simple x86 spinlocks

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

critical section

$x \leftarrow 1$ *OR* atomic_write(x, 1)

Implementing mutexes with simple x86 spinlocks

```
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 atomic (LOCK'd) instructions in a Linux spinlock implementation.

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

x = 0

x = -1

x = -1

x = 1

acquire

critical

critical

critical

release, writing x

acquire

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

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

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

Not triangular race

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

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

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

Triangular race

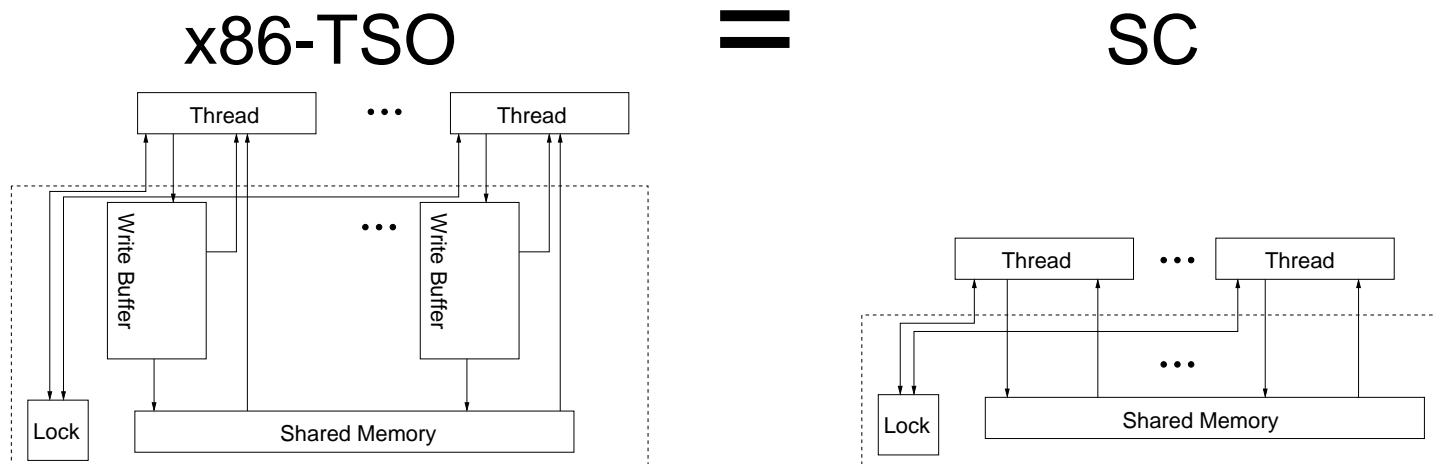
⋮	$y \leftarrow v_2$
⋮	⋮
lock $x \leftarrow v_1$	x
⋮	⋮

TRF Principle for x86-TSO

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

Theorem 2 (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 3 *Any well-synchronized program that uses the spinlock correctly is TRF.*

Theorem 4 *Spinlock-enforced critical sections provide mutual exclusion.*

Other Applications of TRF

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

Architectures

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

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

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)

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*.

(in a real sense, the architectures don't *exist*).

The models we're developing here can be used for all these things. An 'architecture' should be such a precisely defined mathematical artifact.

Validating the models?

We are inventing new abstractions, not just formalising existing clear-but-non-mathematical specs. So why should anyone believe them?

- some aspects of existing arch specs *are* clear (a few concurrency examples, much of ISA spec)
- experimental testing
 - models should be *sound* w.r.t. experimentally observable behaviour of existing h/w (modulo h/w bugs)
 - but the architectural intent may be (often is) looser
- discussion with architects
- consistency with expert-programmer intuition
- formalisation (at least mathematically consistent)
- proofs of metatheory