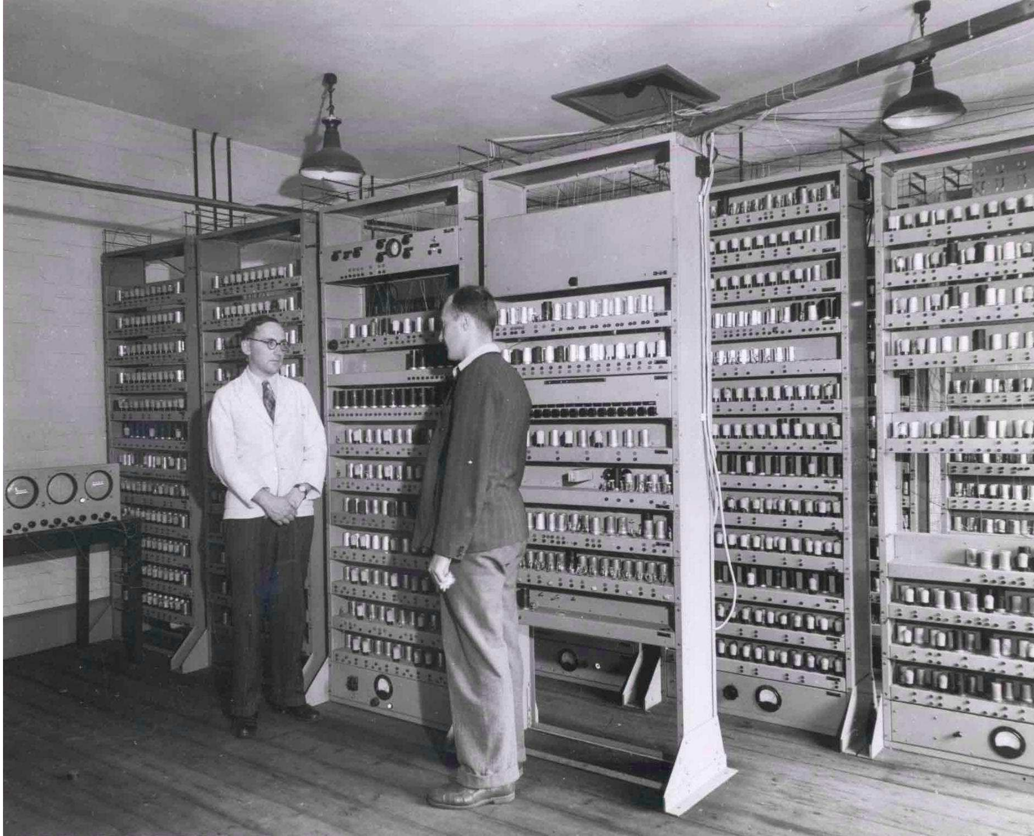
Multicore Semantics and Programming

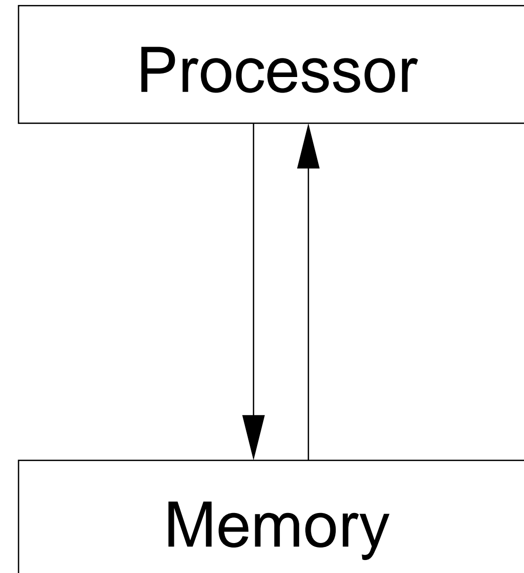
Peter Sewell — Tim Harris
Mark Batty

University of Cambridge — Oracle

October – November, 2012

The Golden Age, 1945–1959





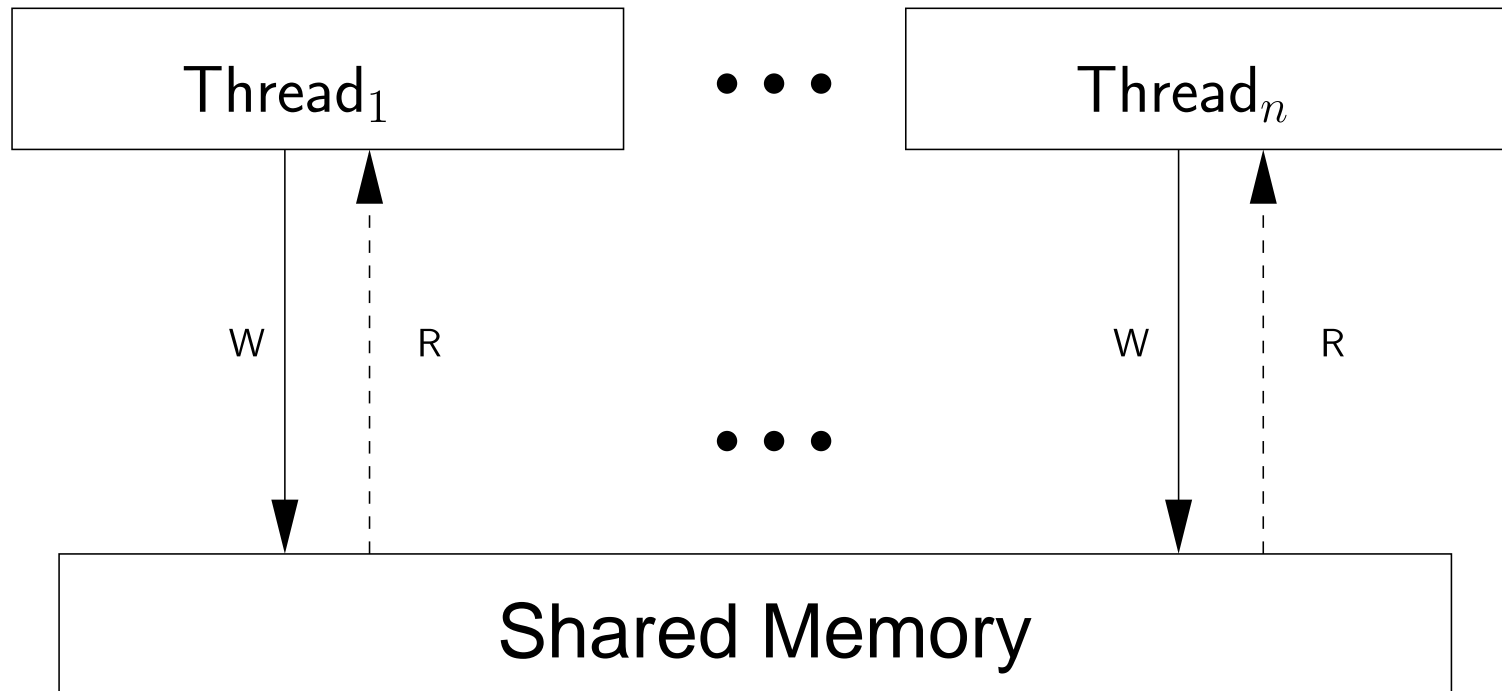
1962: First(?) Multiprocessor

BURROUGHS D825, 1962

‘‘Outstanding features include truly modular hardware with parallel processing throughout’’

FUTURE PLANS
The complement of compiling languages is to be expanded.’’

... with Shared-Memory Concurrency



Multiple threads acting on a *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, respecting the order specified by the program* [Lamport, 1979]

Multiprocessors, 1962–2012

Niche multiprocessors since 1962

IBM System 370/158MP in 1972

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

Multiprocessors, 2012

Intel Xeon E7
(up to 20 hardware threads)
(also AMD, Centaur)

ARM®

MOBILE PHONES NEWS

Best quad core phone: 4 contenders examined

Early View: HTC One X vs ZTE Era vs LG Optimus 4X HD vs Huawei Ascend D Quad

IBM Power 795 server
(up to 1024 hardware threads)

Oracle Sparc, Intel Itanium

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?

Aside: Concurrency everywhere, at many scales:

- intra-core
- GPU
- multicore (/manycore) systems $\leftarrow$ our focus
- datacenter-scale

explicit message-passing vs shared memory

These Lectures

Part 1: Concurrency in multiprocessors and programming languages (Peter Sewell, Susmit Sarkar, Mark Batty)

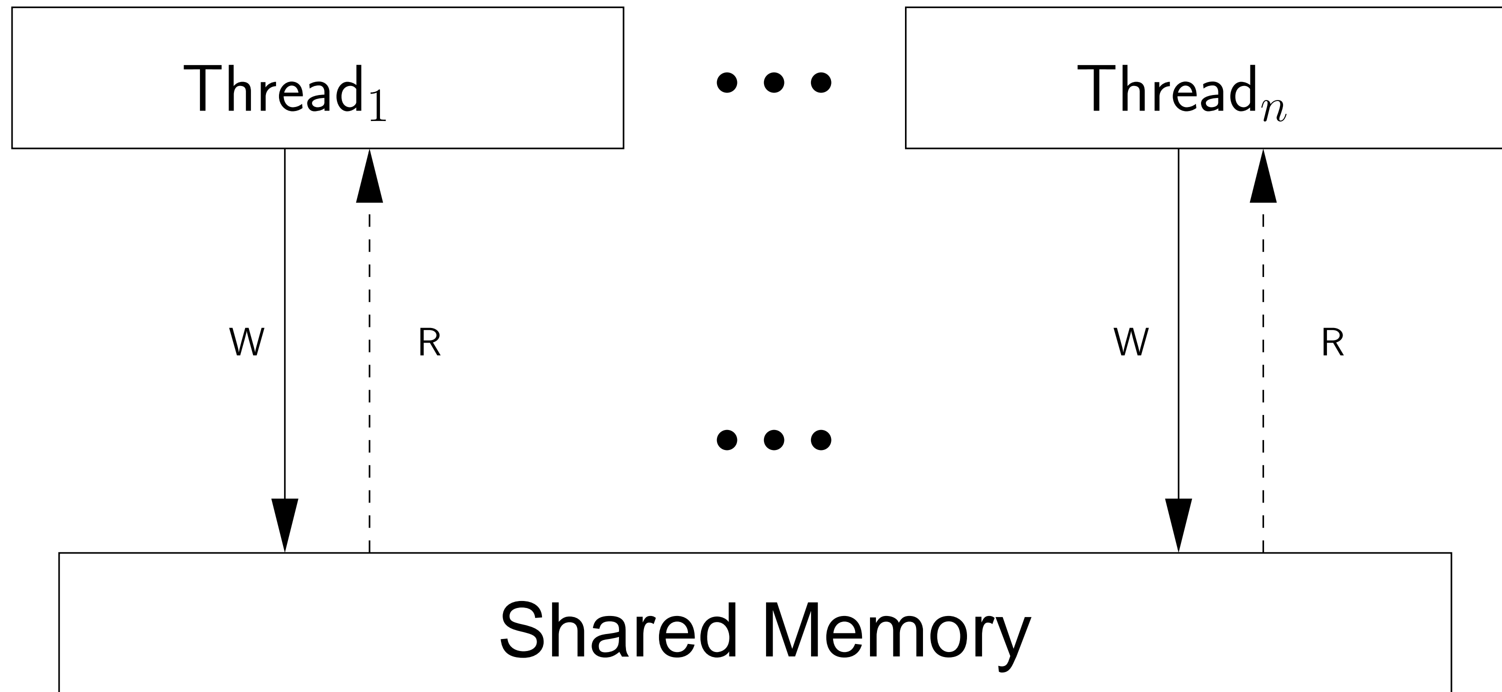
Establish a solid basis for thinking about relaxed-memory executions, linking to usage, microarchitecture, experiment, and semantics. x86, POWER/ARM, problems with Java, C/C++11

Part 2: Concurrent algorithms (Tim Harris, Oracle)

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

Guest Lecture: Claudio Russo (MSR)

Let's Pretend... that we live in an SC world



Multiple threads acting on a *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, respecting the order specified by the program* [Lamport, 1979]

A Tiny Language

$location,\ x,\ m$	address		
$integer,\ n$	integer		
$thread_id,\ t$	thread id		
$expression,\ e$	$::=$		expression
	$\vert$	n	integer literal
	$\vert$	x	read from address x
	$\vert$	$x = e$	write value of e to address x
	$\vert$	$e; e'$	sequential composition
	$\vert$	$e + e'$	plus
$process,\ p$	$::=$		process
	$\vert$	$t{:}e$	thread
	$\vert$	$p \vert p'$	parallel composition

A Tiny Language

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

Defining an SC Semantics: SC memory

Take an SC *memory* M to be a function from addresses to integers.

Define the behaviour as a labelled transition system (LTS): the least set of (memory,label,memory) triples satisfying these rules.

$\boxed{M \xrightarrow{t:l} M'}$ M does $t:l$ to become M'

$$\frac{M(x) = n}{M \xrightarrow{t:\mathsf{R}\,x=n} M} \quad \textsc{MRead}$$

$$\frac{}{M \xrightarrow{t:\mathsf{W}\,x=n} M \oplus (x \mapsto n)} \quad \textsc{MWrite}$$

Defining an SC Semantics: expressions

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

$$\frac{}{x \xrightarrow{\mathsf{R}\,x=n} n} \quad \text{READ}$$

$$\frac{}{x = n \xrightarrow{\mathsf{W}\,x=n} n} \quad \text{WRITE}$$

$$\frac{e \xrightarrow{l} e'}{x = e \xrightarrow{l} x = e'} \quad \text{WRITE_CONTEXT}$$

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

$$\frac{e_1 \xrightarrow{l} e_1'}{e_1; e_2 \xrightarrow{l} e_1'; e_2} \quad \text{SEQ_CONTEXT}$$

$\boxed{e \xrightarrow{l} e'}$ 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

$(x = y); x$

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

Example: SC Expression Trace

$$(x = y); x$$

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

$$\dfrac{\dfrac{\dfrac{}{y \xrightarrow{\mathsf{R}\, y=7} 7}\ \text{READ}}{x = y \xrightarrow{\mathsf{R}\, y=7} x = 7}\ \text{WRITE}}{(x = y); x \xrightarrow{\mathsf{R}\, y=7} (x = 7); x}\ \text{SEQ_CONTEXT}$$

Example: SC Expression Trace

$(x = y); x$

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

$$\dfrac{\dfrac{}{x = 7 \xrightarrow{\mathsf{W}\, x=7} 7}\ \text{WRITE}}{(x = 7); x \xrightarrow{\mathsf{W}\, x=7} 7; x}\ \text{SEQ_CONTEXT}$$

Example: SC Expression Trace

$$(x = y); x$$

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

$$\frac{}{7; x \xrightarrow{\tau} x} \quad \text{SEQ}$$

$$\frac{}{x \xrightarrow{\mathsf{R}\, x=9} 9} \quad \text{READ}$$

Defining an SC Semantics: lifting to processes

$\boxed{p \xrightarrow{t:l} p'}$ p does $t : l$ to become p'

$$\frac{e \xrightarrow{l} e'}{t{:}e \xrightarrow{t:l} t{:}e'} \quad \text{THREAD}$$

$$\frac{p_1 \xrightarrow{t:l} p_1'}{p_1|p_2 \xrightarrow{t:l} p_1'|p_2} \quad \text{PAR_CONTEXT_LEFT}$$

$$\frac{p_2 \xrightarrow{t:l} p_2'}{p_1|p_2 \xrightarrow{t:l} p_1|p_2'} \quad \text{PAR_CONTEXT_RIGHT}$$

free interleaving

Defining an SC Semantics: whole-system states

A *system state* $\langle p,\ M\rangle$ is a pair of a process and a memory.

$\boxed{s \xrightarrow{t:l} s'}$ $\quad s$ does $t : l$ to become s'

$$\frac{\begin{array}{c} p \xrightarrow{t:l} p' \\ M \xrightarrow{t:l} M' \end{array}}{\langle p,\ M\rangle \xrightarrow{t:l} \langle p',\ M'\rangle} \quad \text{SSYNC}$$

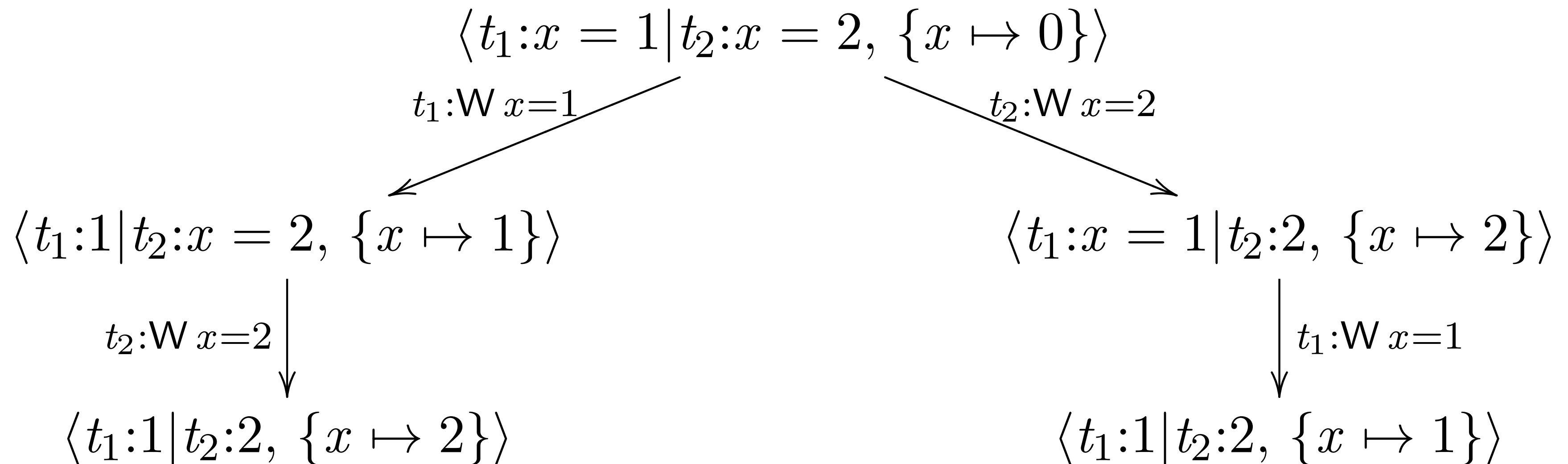
$$\frac{p \xrightarrow{t:\tau} p'}{\langle p,\ M\rangle \xrightarrow{t:\tau} \langle p',\ M\rangle} \quad \text{STAU}$$

synchronising between the process and the memory, and
letting threads do internal transitions

Example: SC Interleaving

All threads can read and write the shared memory.

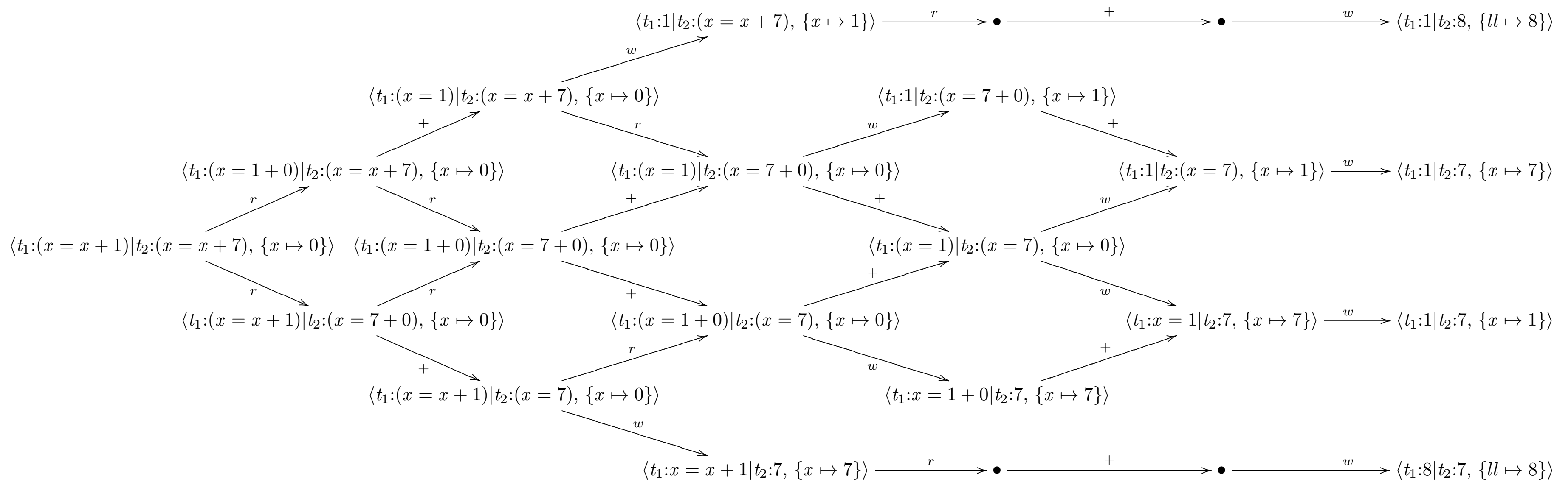
Threads execute asynchronously – the semantics allows any interleaving of the thread transitions. Here there are two:



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

Combinatorial Explosion

The behaviour of $t_1{:}x = x + 1 | t_2{:}x = x + 7$ 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.

- with built-in support from the scheduler, eg for *mutexes* and *condition variables* (at the language or OS level)
- coding up mutex library implementations just using reads and writes
- ...or with richer hardware primitives (TAS, CAS, LL/SC)

See Part 2 of these lectures.

Here's 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).

Implementing Simple Mutual Exclusion

Initial state: x=0 and y=0	
Thread 0	Thread 1
x=1 if (y==0) { *...critical section...* }	y=1 if (x==0) {*...critical section...* }

Implementing Simple Mutual Exclusion

Initial state: x=0 and y=0	
Thread 0	Thread 1
x=1 if (y==0) { *...critical section...* }	y=1 if (x==0) {*...critical section...* }

repeated use?
thread symmetry (same code on each thread)?
performance?
fairness?
deadlock, global lock ordering, compositionality?

Message Passing

Another basic concurrent idiom:

Initial State: flag=0	

x1=10;
x2=20;
flag=1
	while (0==flag) do {};
	x1+x2

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

Atomicity again

In this toy language, assignments and dereferencing are *atomic*. For example,
$\langle t_1{:}x = 3498734590879238429384 | t_2{:}x = 7,\ \{x \mapsto 0\}\rangle$
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 $t_1{:}(x = e)|t_2{:}e'$, the steps of evaluating e and e' can be interleaved.

Wrapping all subexpressions that access potentially shared variables in lock/unlock pairs enlarges the granularity of atomic reads and writes.

Data Races

...or, said another way, it excludes *data races*

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

(p.s. what *is* a race exactly?)

Let's Try...

SB:

MP

In SC, message passing should work as expected:

Thread 1	Thread 2
`data = 1`	
`ready = 1`	`if (ready == 1)`
	`print data`

In SC, the program should only print 1.

MP

Thread 1	Thread 2
`data = 1`	`int r1 = data`
`ready = 1`	`if (ready == 1)`
	`   print data`

In SC, the program should only print 1.

Regardless of other reads.

MP

Thread 1	Thread 2
`data = 1`	`int r1 = data`
`ready = 1`	`if (ready == 1)`
	`print data`

In SC, the program should only print 1.

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

`print data` $\Longrightarrow$ `print r1`

MP

Thread 1	Thread 2

`data = 1`
`ready = 1`
	`int r1 = data`
	`if (ready == 1)`
	`   print r1`

In SC, the program should only print 1.

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

`print data` $\Longrightarrow$ `print r1`

So the compiled program can print 0

Let's *Not* Pretend... that we live in an SC world

Not since that IBM System 370/158MP in 1972

nor in x86, ARM, POWER, SPARC, or Itanium

or in C, C++, or Java

Relaxed 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

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: "sp
optimization(i386)"

Topics: BSD: FreeBSD, SMP

People: Linus Torvalds, Jeff V. Merkey, Erich Boleyn,
Spraul, Peter Samuelson, Ingo Molnar

Manfred Spraul thought he'd found a way to shave spir
down from about 22 ticks for the "`lock; btrl $0,%0`" a
to 1 tick for a simple "`movl $0,%0`" instruction, a huge ga
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 po
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, the
> denly spinlocks aren't reliable any more.

that the piplines are no longer invalid and the b
should be blown out.

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

Resolved only by appeal to an oracle:

Erich Boleyn, an Architect in an IA32 development grou
also replied to Linus, pointing out a possible miscon
his proposed exploit. Regarding the code Linus post
replied:

It will always return 0. You don't need "`spi`
`lock()`" to be serializing.

The only thing you need is to make sure ther
store in "`spin_unlock()`", and that is kind of tr
the fact that you're changing something to be ol
able on other processors.

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

He went on:

Since the instructions for the store in the spin u

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	

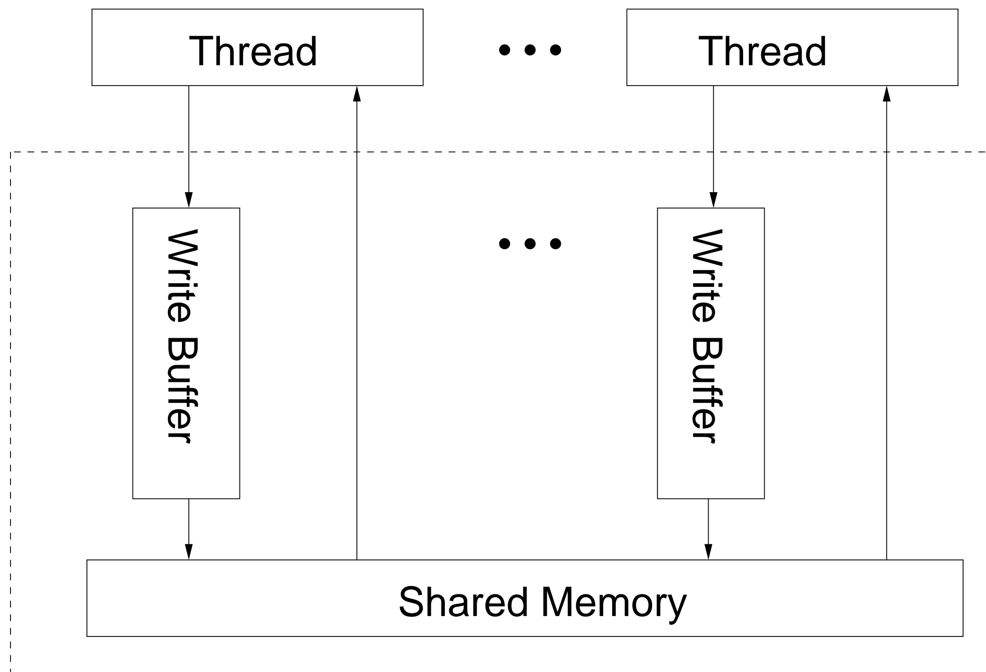
P3. Loads may be reordered with older stores to different locations but not with older stores to the same location

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

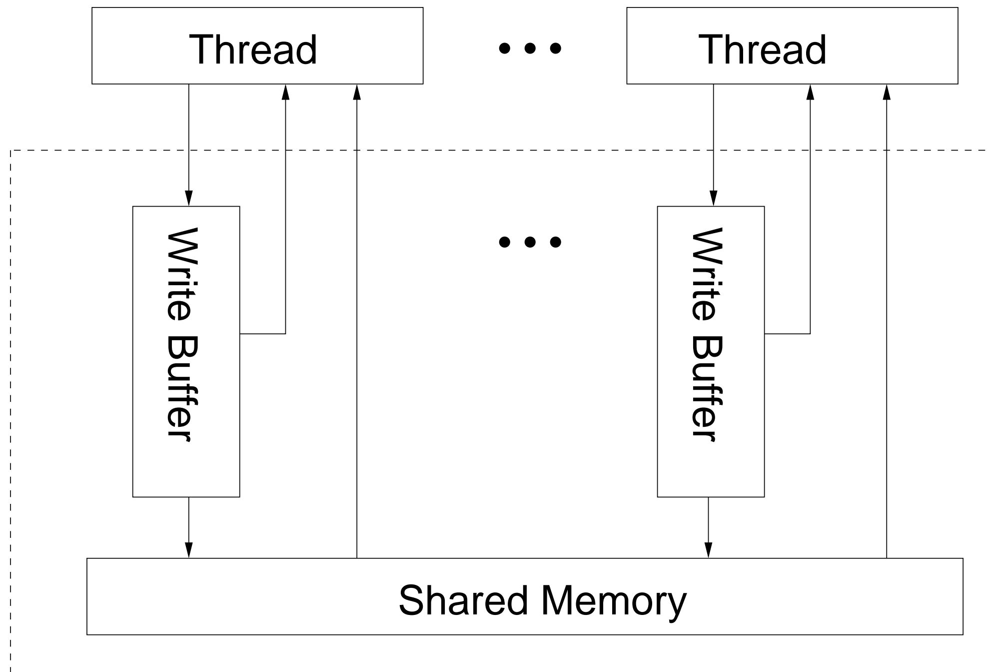


Litmus Test 2.4. Intra-processor forwarding is allowed

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 ∧ Thread 1:EDX=0 Thread 0:EAX=1 ∧ Thread 1:ECX=1			

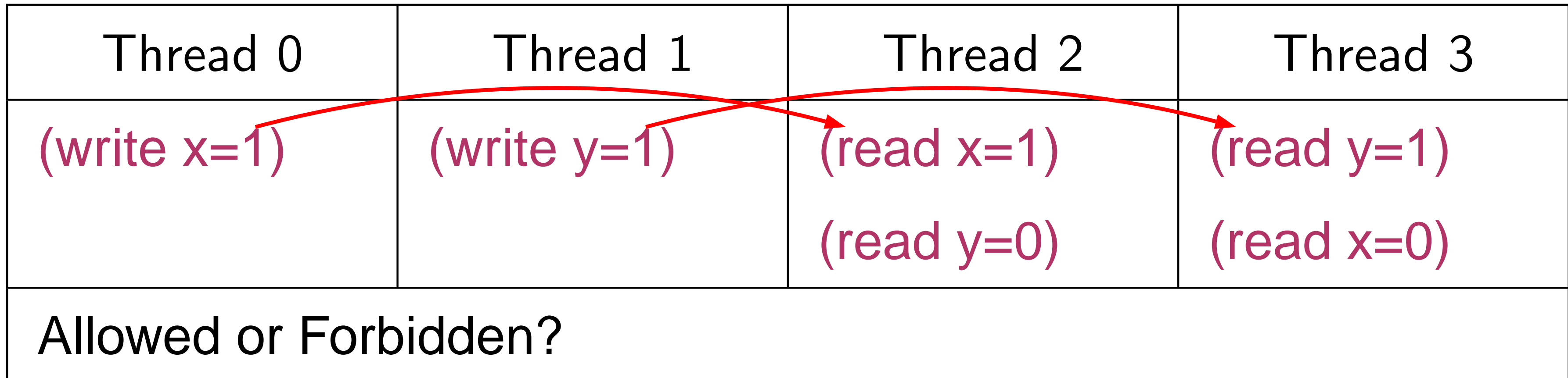
Litmus Test 2.4. Intra-processor forwarding is allowed

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 ∧ Thread 1:EDX=0 Thread 0:EAX=1 ∧ Thread 1:ECX=1			



Problem 1: Weakness

Independent Reads of Independent Writes (IRIW)



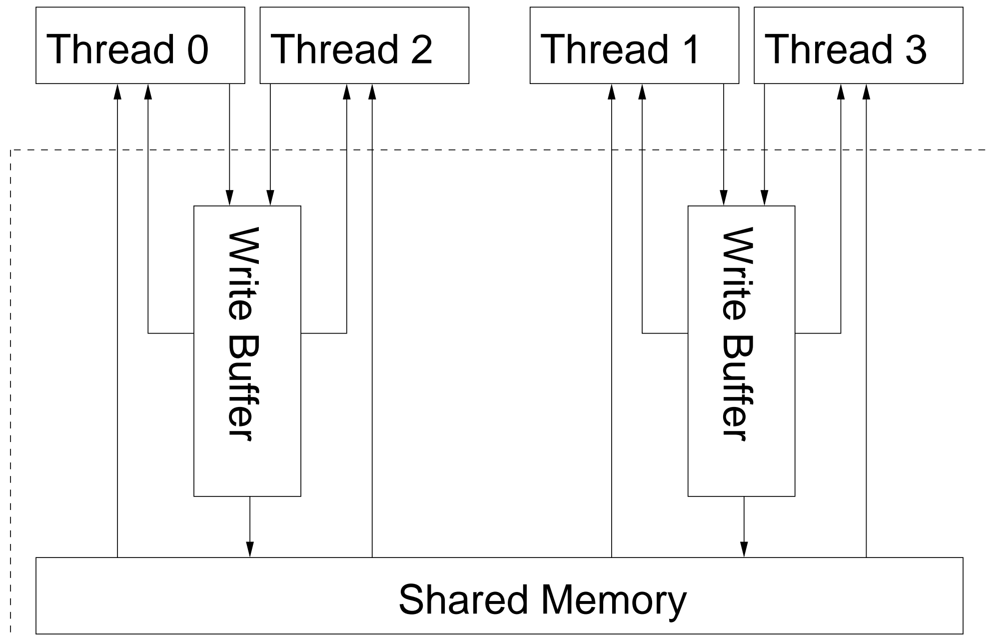
Thread 0	Thread 1	Thread 2	Thread 3
(write x=1)	(write y=1)	(read x=1) (read y=0)	(read y=1) (read x=0)
Allowed or Forbidden?			

Problem 1: Weakness

Independent Reads of Independent Writes (IRIW)

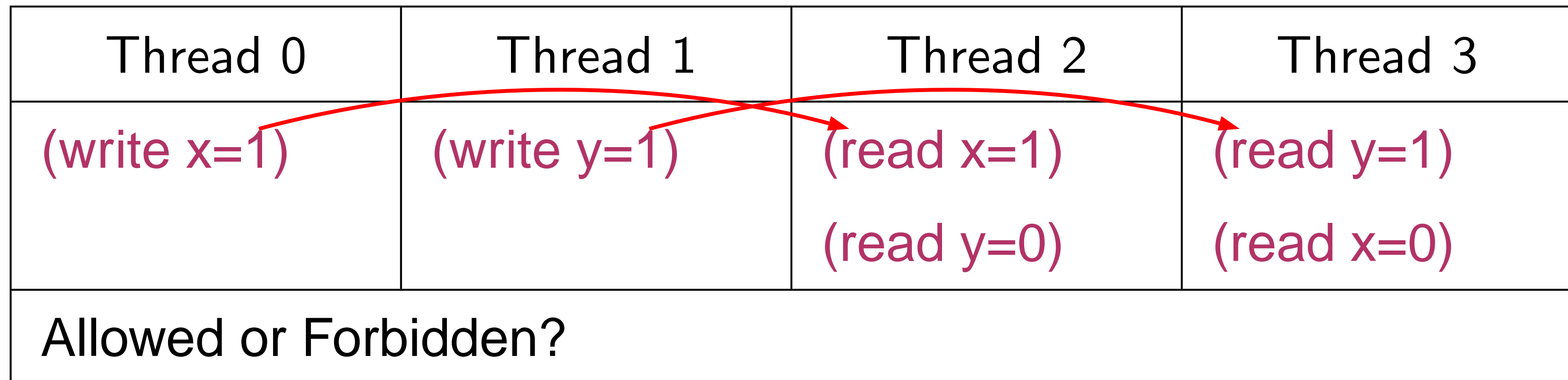
Thread 0	Thread 1	Thread 2	Thread 3
(write x=1)	(write y=1)	(read x=1) (read y=0)	(read y=1) (read x=0)
Allowed or Forbidden?			

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=0)	(read y=1) (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 ∧ Thread 2:EBX=1 ∧ 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/content/www/us/en/processors/architecture
software-developer-manuals.html
(rev. 35 on 6/10/2010).
See especially SDM Vol. 3A, Ch. 8.

AMD:
http://developer.amd.com/Resources/documentation/guides/Pages/
(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

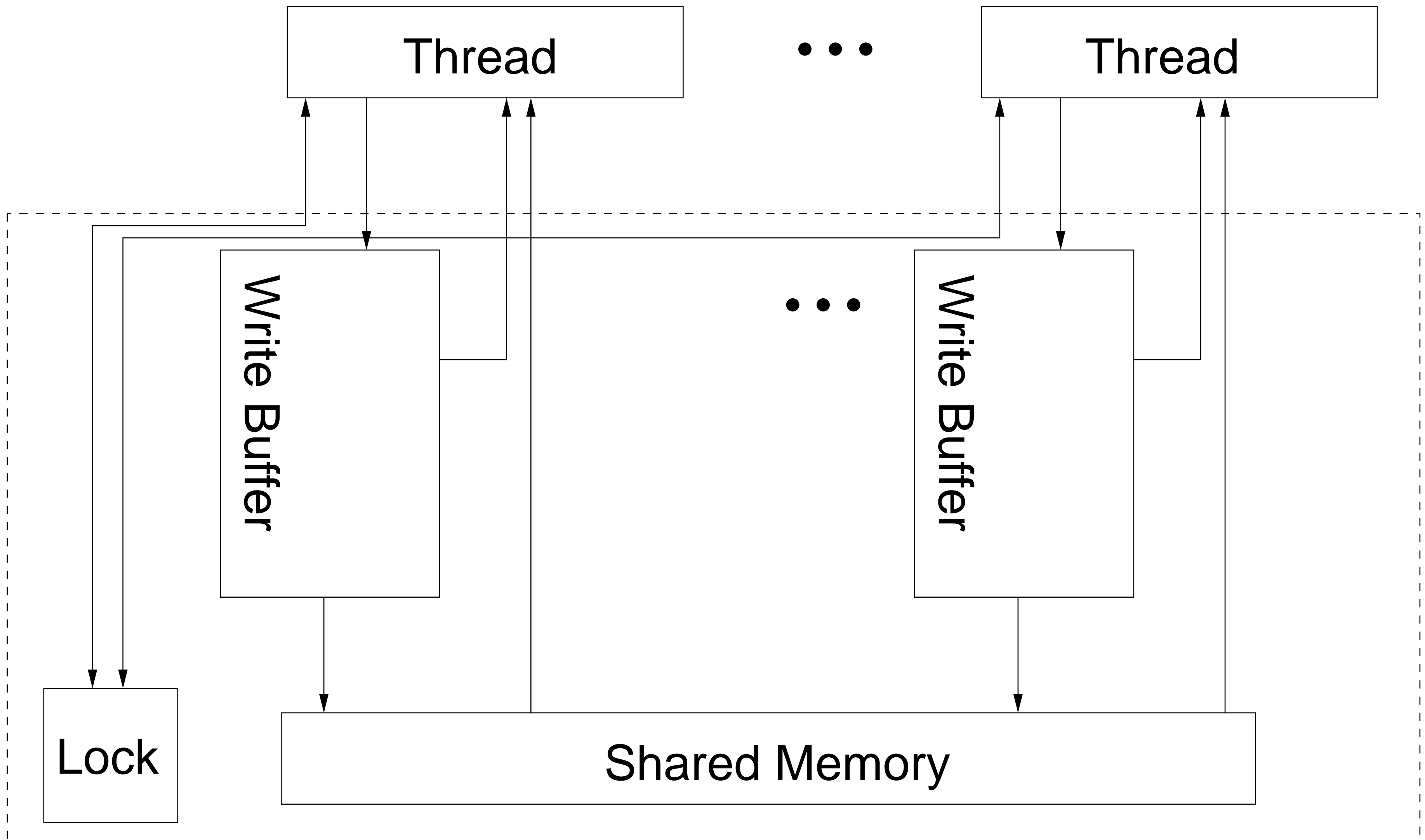
Separate *instruction semantics* and *memory model*

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	::=	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: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
	|	t:$\tau_{x=v}$	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

$$
\begin{aligned}
s : \langle\!| \quad & M : \textit{addr} \rightarrow \textit{value};\\
& B : \textit{tid} \rightarrow (\textit{addr} \times \textit{value})\ \textit{list};\\
& L : \textit{tid option} |\!\rangle
\end{aligned}
$$

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 ($s.L = \textsc{Some}\ t$) or the lock is not held ($s.L = \textsc{None}$).

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

$$\frac{\begin{array}{l}\text{not_blocked}(s, t)\\ s.M(x) = v\\ \text{no_pending}(s.B(t), x)\end{array}}{s \xrightarrow{t:\mathsf{R}\,x=v} s}$$

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

$$\frac{\begin{array}{l}\mathrm{not_blocked}(s, t) \\ \exists b_1\ b_2.\ s.B(t) = b_1 +\!\!+ [(x, v)] +\!\!+ b_2 \\ \mathrm{no_pending}(b_1, x)\end{array}}{s \xrightarrow{t:\mathsf{R}\,x=v} s}$$

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

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

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

$$\frac{\begin{array}{c}\mathrm{not_blocked}(s,t) \\ s.B(t) = b +\!\!+ [(x,v)]\end{array}}{s \xrightarrow{t:\tau_{\ x=v}} s \oplus \langle\![M := s.M \oplus (x \mapsto v)]\!\rangle \oplus \langle\![B := s.B \oplus (t \mapsto b)]\!\rangle}$$

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

$$\frac{s.L = \textsc{None} \qquad s.B(t) = [\,]}{s \xrightarrow{t:\mathsf{L}} s \oplus \langle\!\![L := \textsc{Some}(t)]\!\!\rangle}$$

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

$$\frac{\begin{array}{c} s.L = \textsc{Some}(t) \\ s.B(t) = [\,] \end{array}}{s \xrightarrow{t:\mathsf{U}} s \oplus \langle\!\![L := \textsc{None}]\!\!\rangle}$$

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

$$\frac{s.B(t) = [\,]}{s \quad \xrightarrow{t:\mathsf{B}} \quad s}$$

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

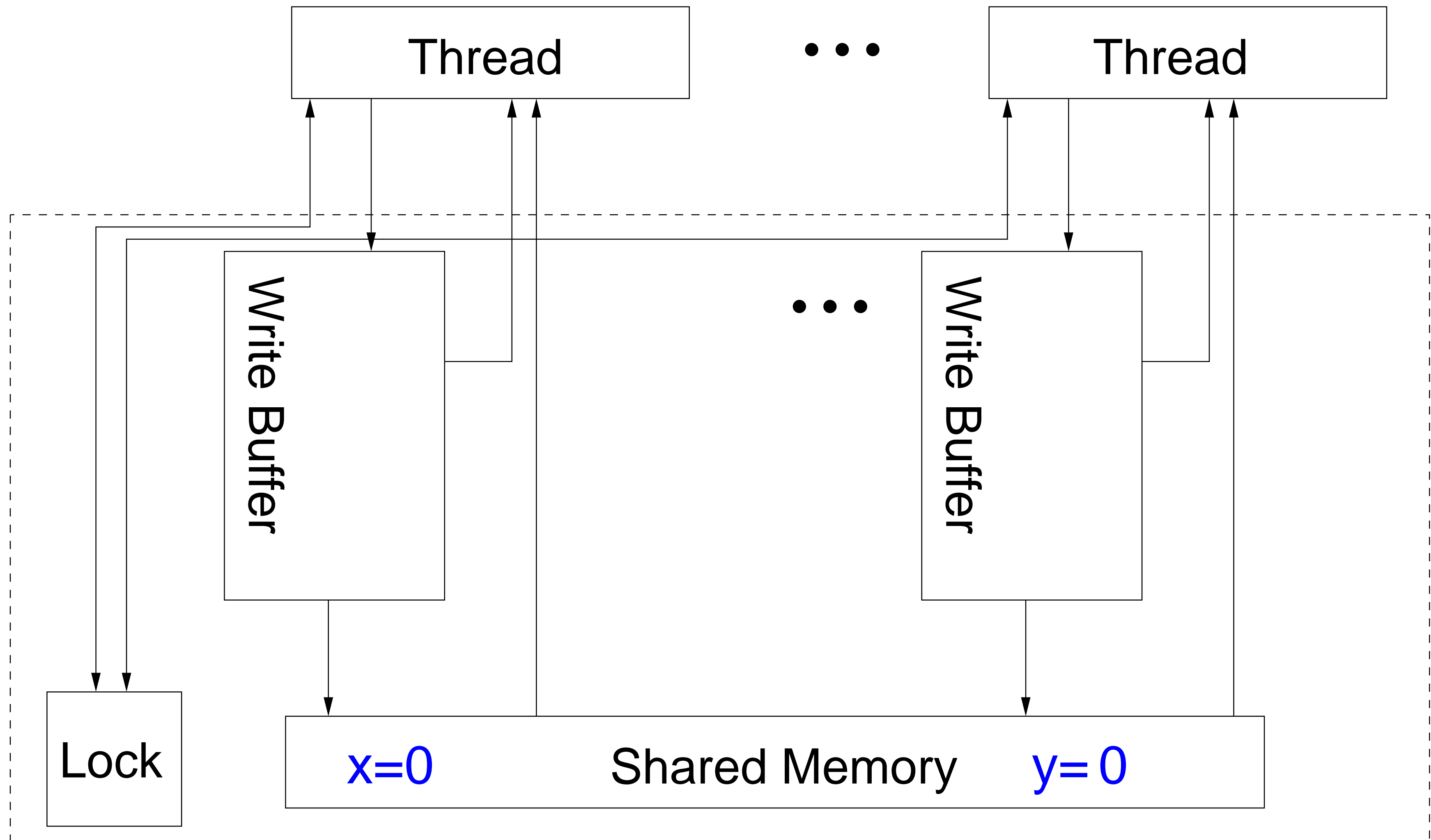
$+\!\!+$ appends lists

$\cdot \oplus \langle\!\![\cdot := \cdot]\!\!\rangle$ updates records

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

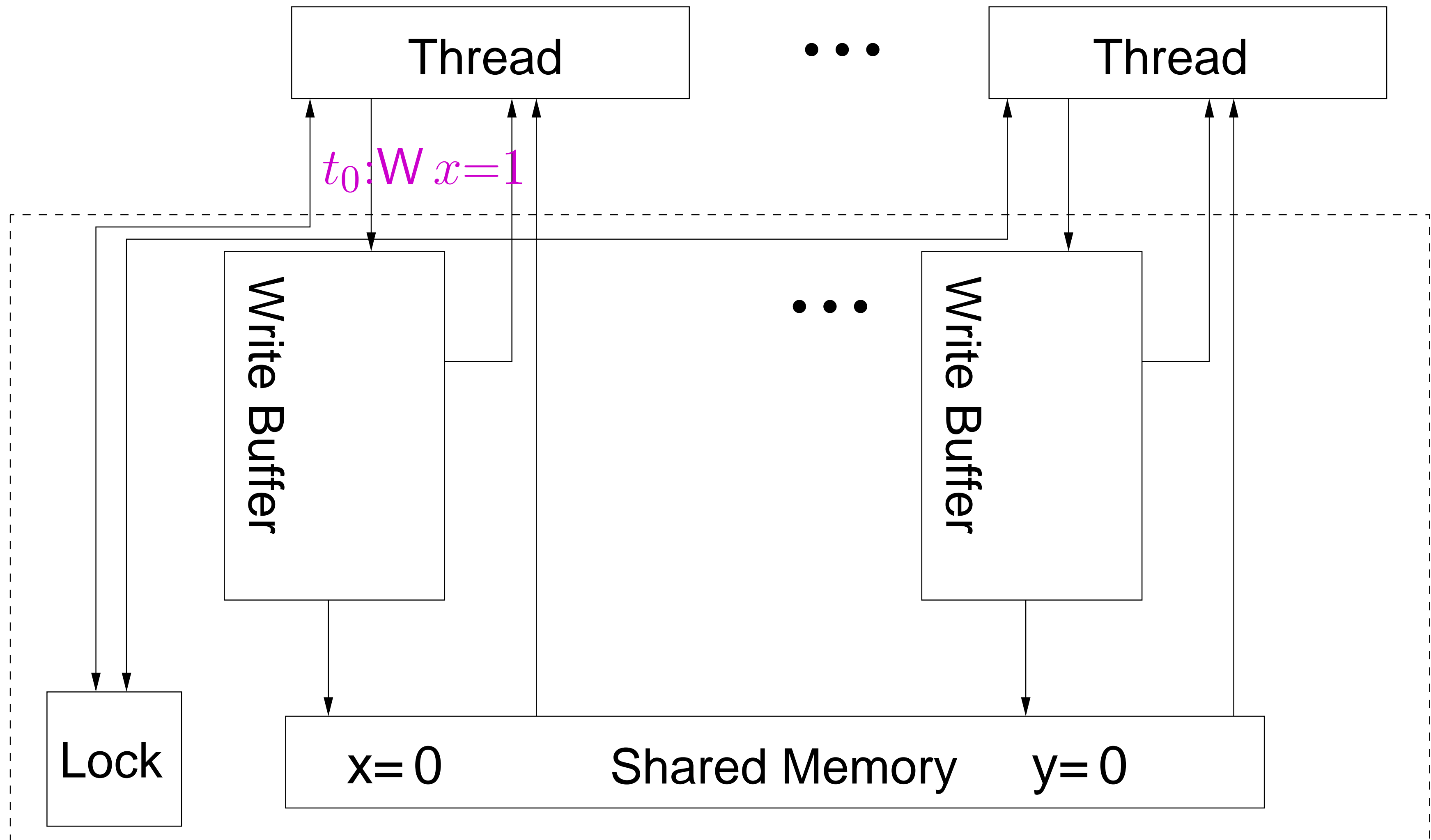
First Example, Revisited

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)



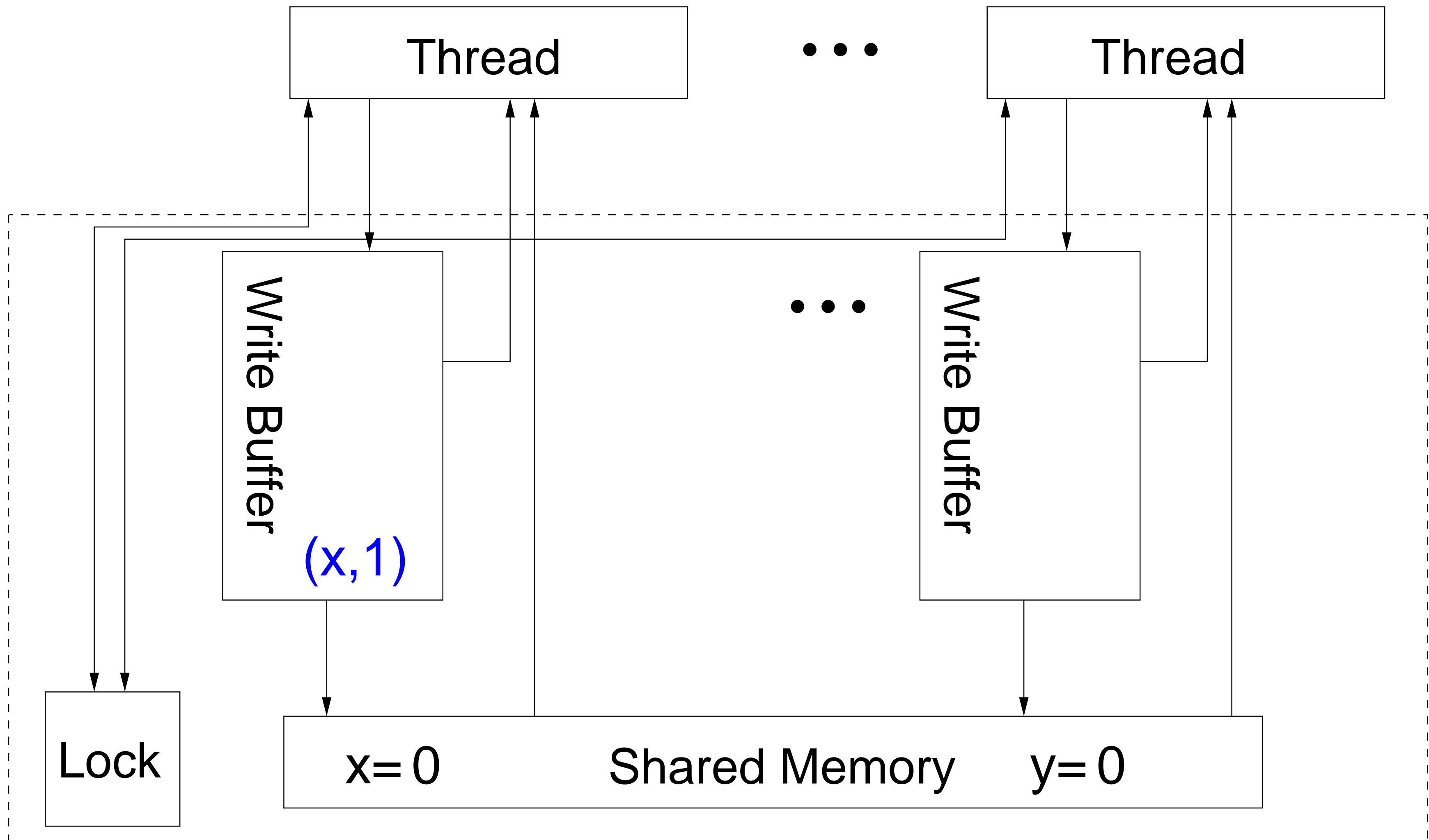
First Example, Revisited

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)



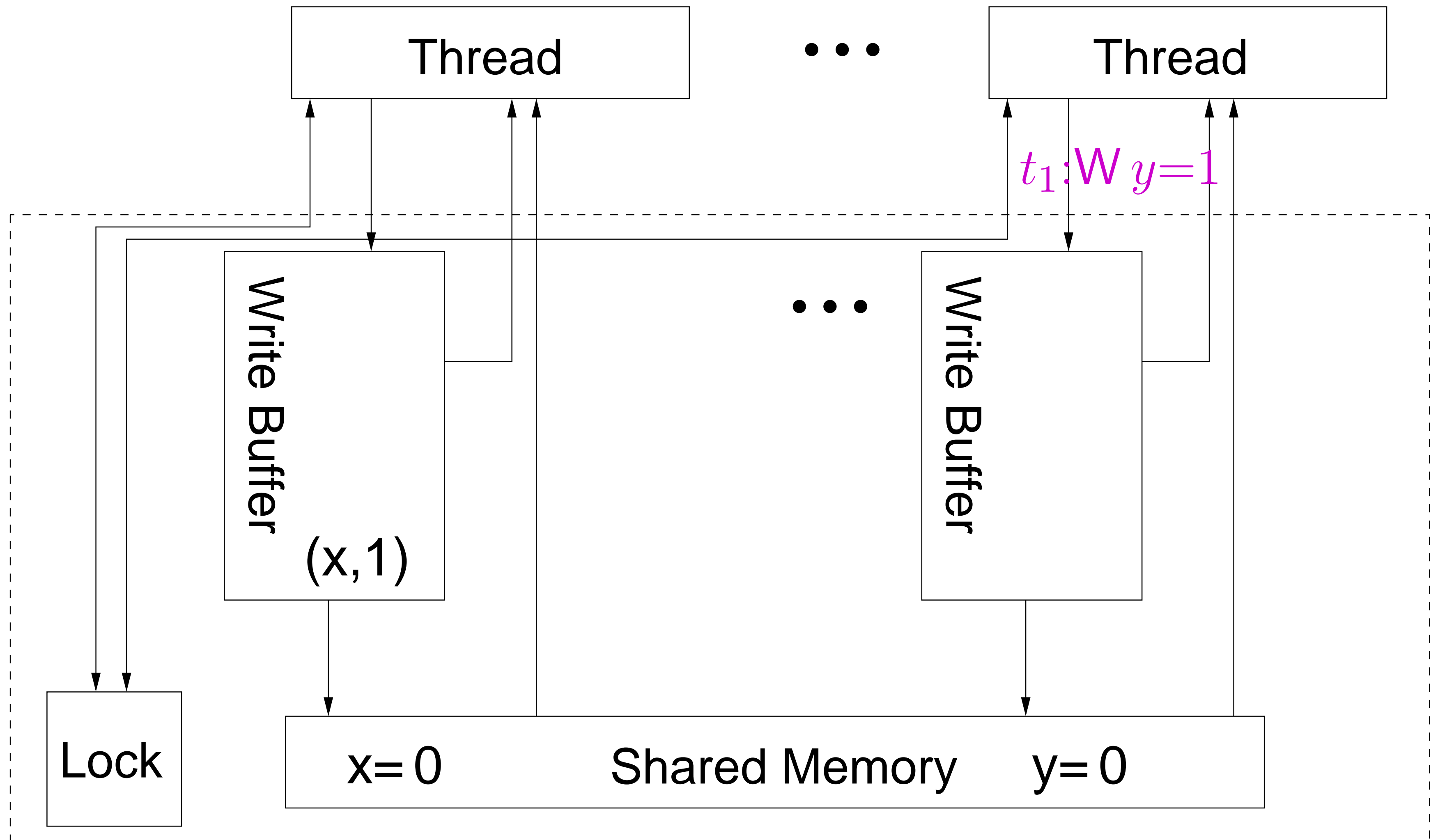
First Example, Revisited

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)



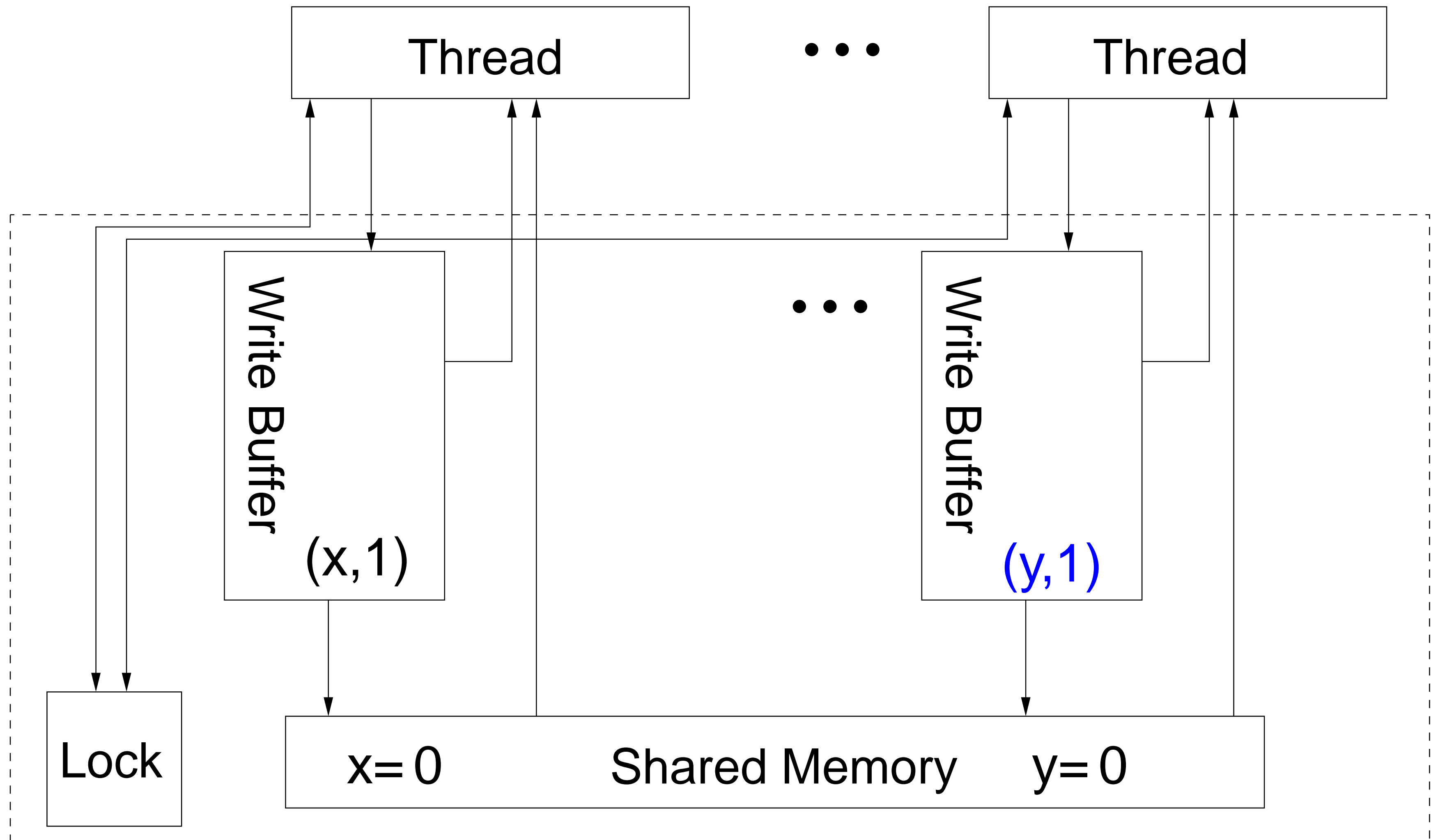
First Example, Revisited

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)



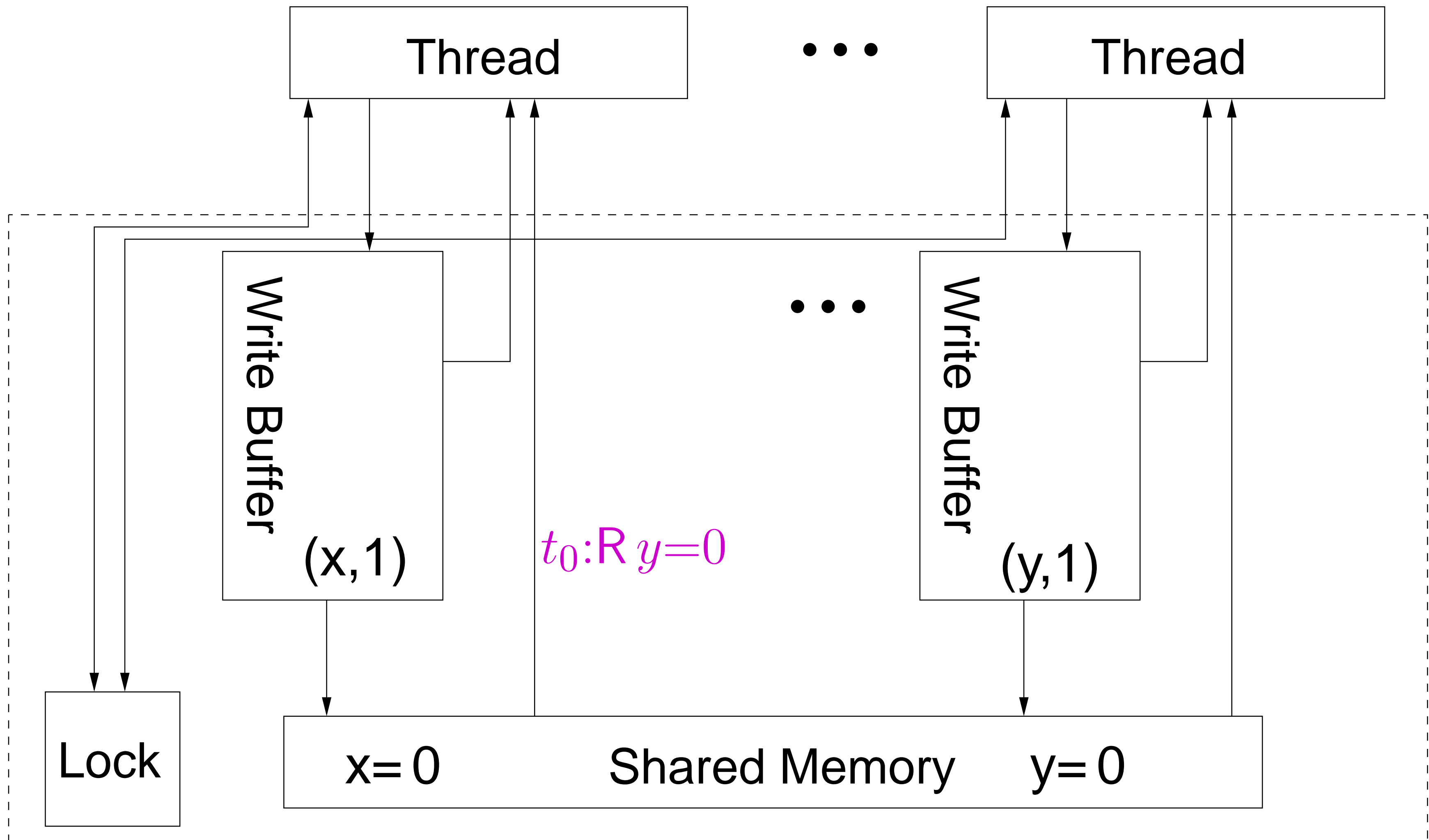
First Example, Revisited

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



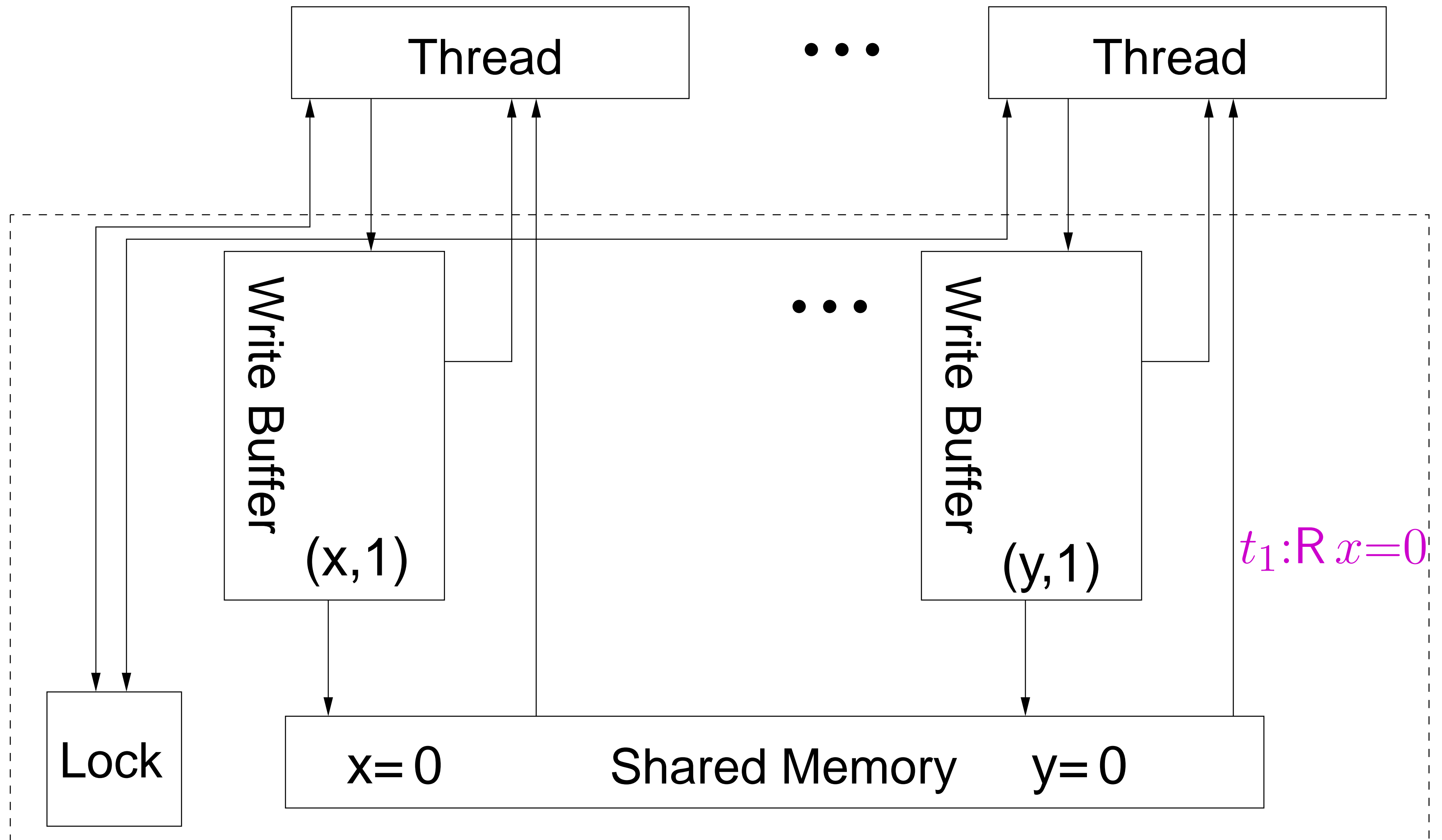
First Example, Revisited

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)



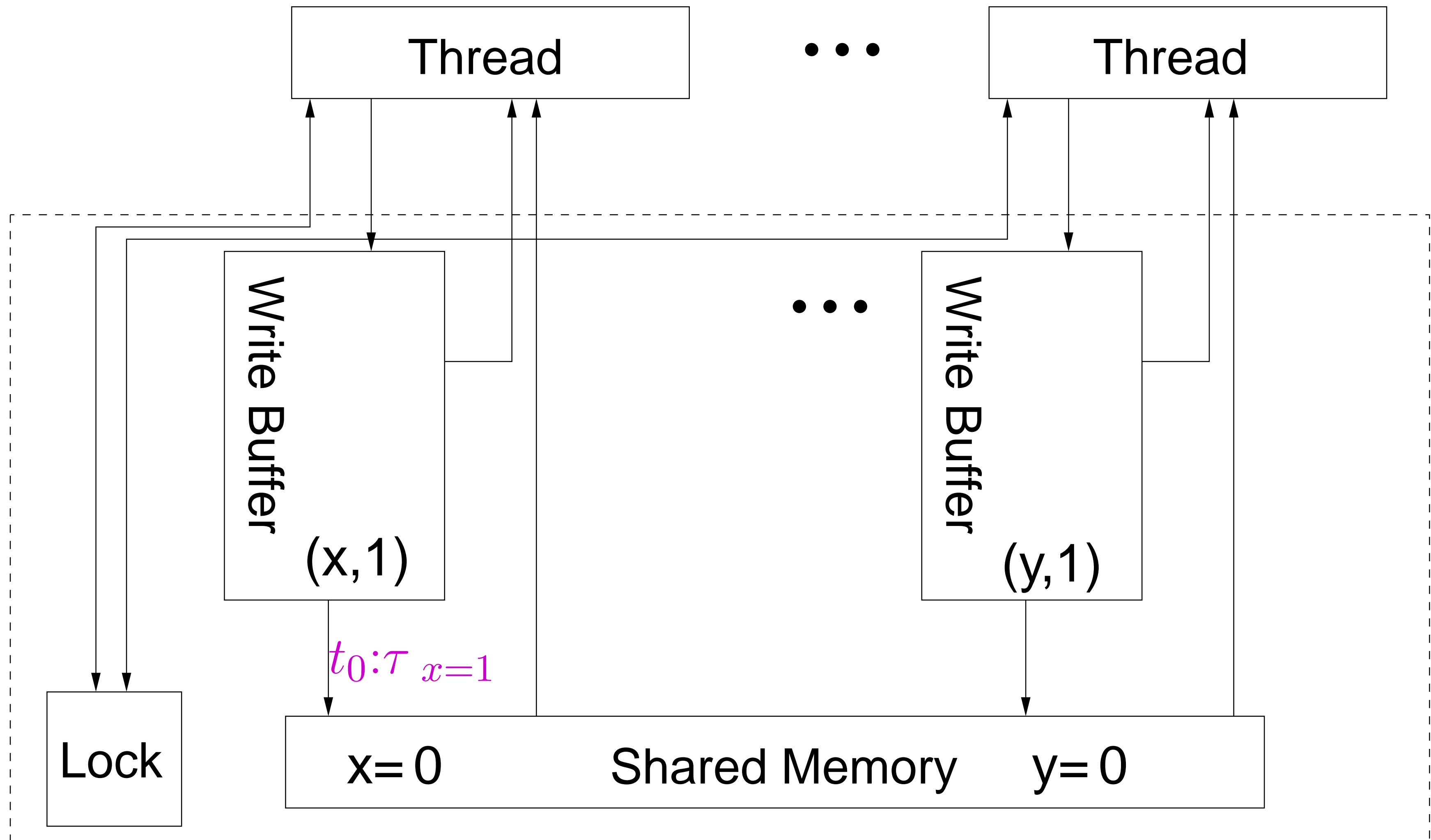
First Example, Revisited

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)



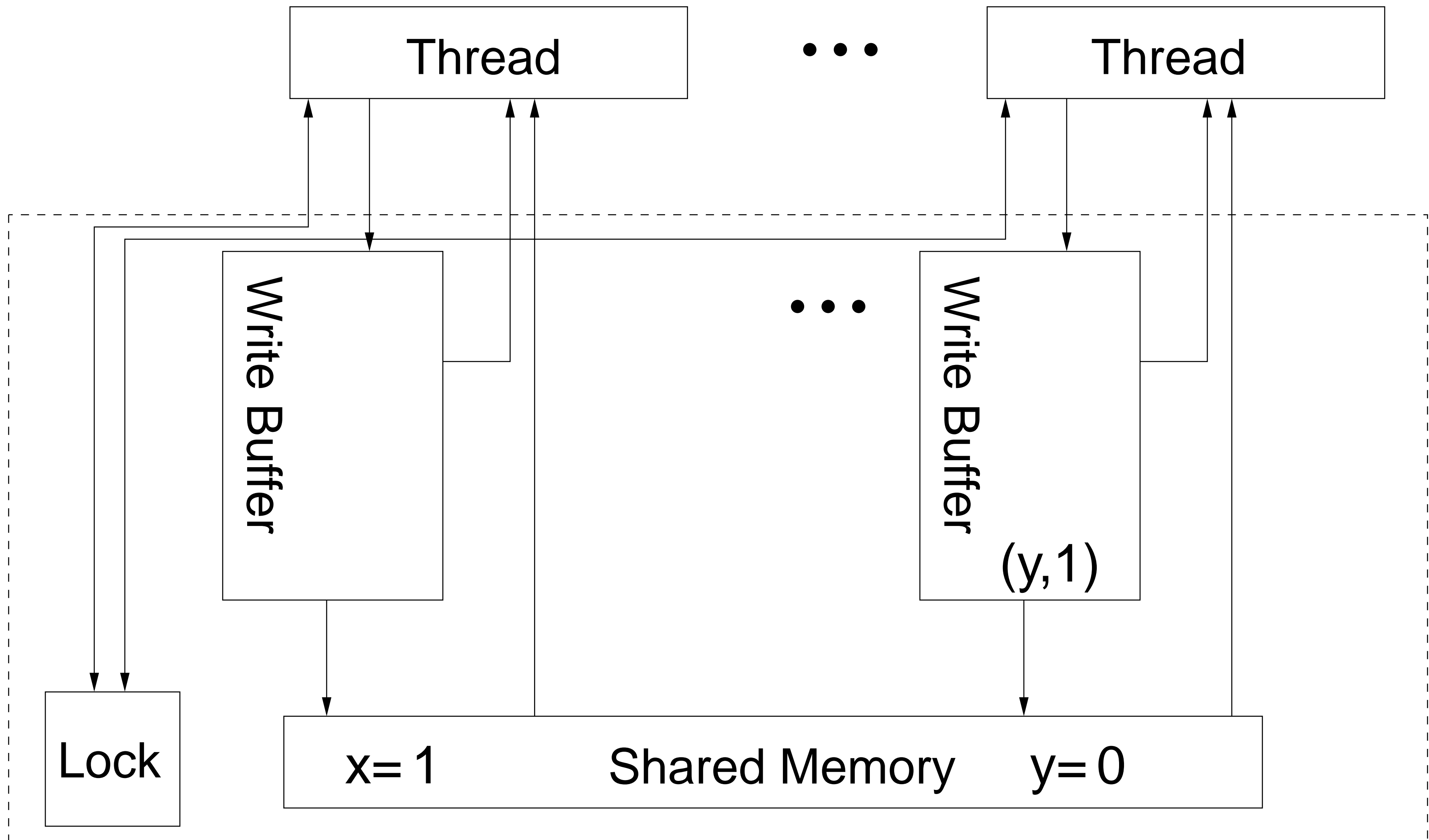
First Example, Revisited

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)



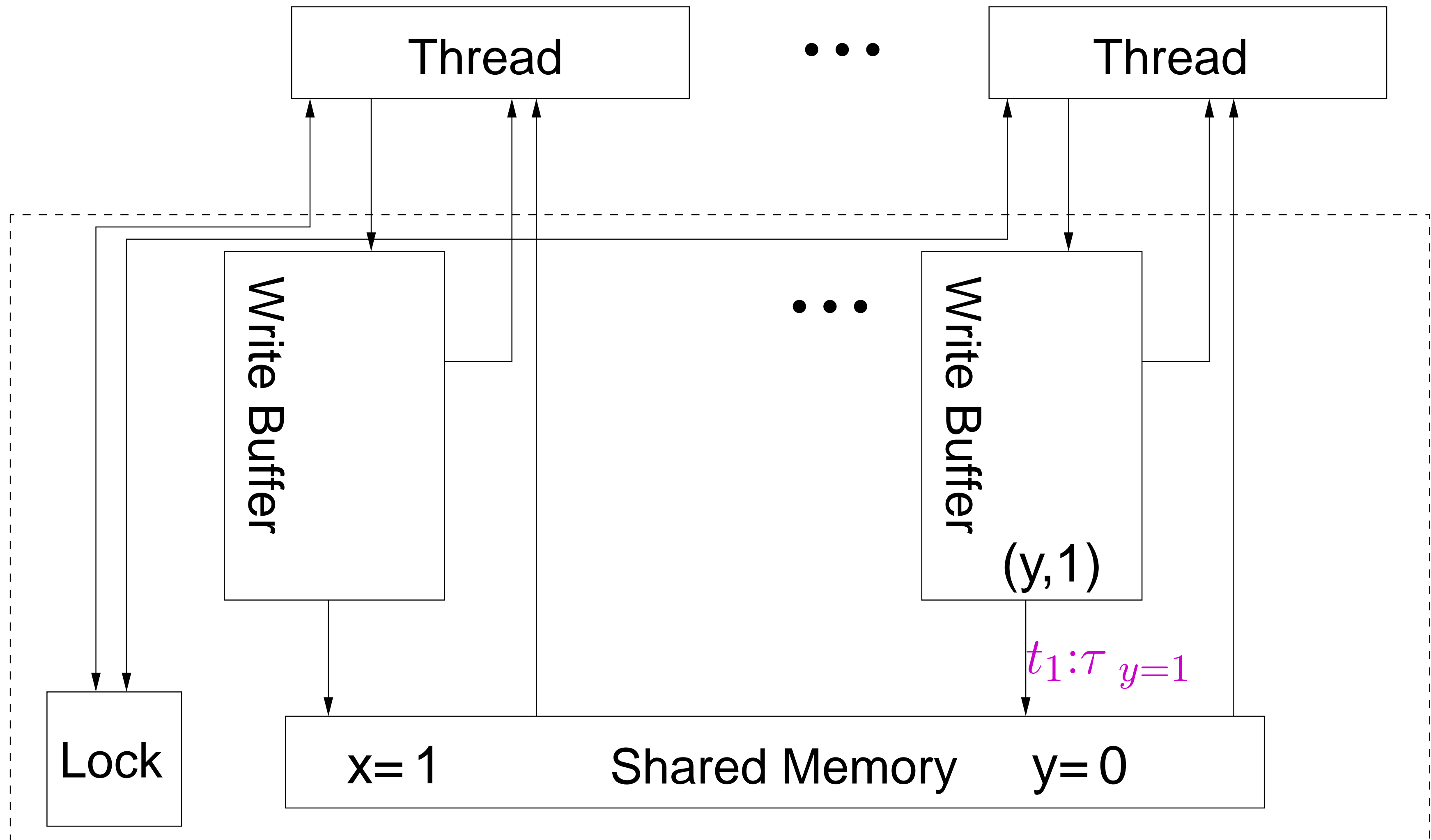
First Example, Revisited

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

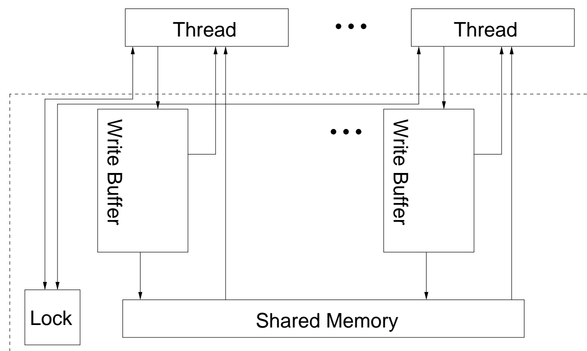
- MFENCE memory barrier
  - flushes local write buffer
- LOCK'd instructions (atomic INC, ADD, CMPXCHG, etc.)
  - flush local write buffer
  - globally locks memory

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)
Forbidden Final State: Thread 0:EAX=0 ∧ Thread 1:EBX=0	

NB: both are *expensive*

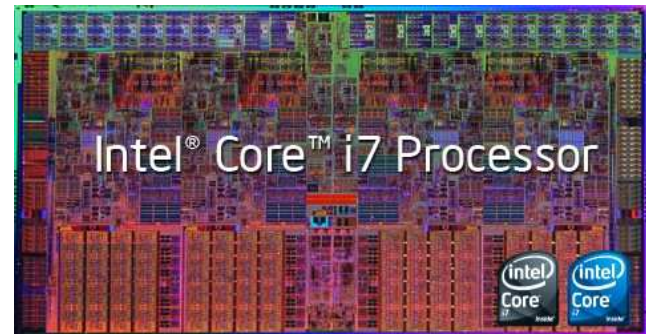
NB: This is an *Abstract* Machine

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.