Multicore Semantics and Programming

Mark Batty

University of Cambridge

January, 2013

These Lectures

Semantics of concurrency in multiprocessors and programming languages.

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

Today: x86

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

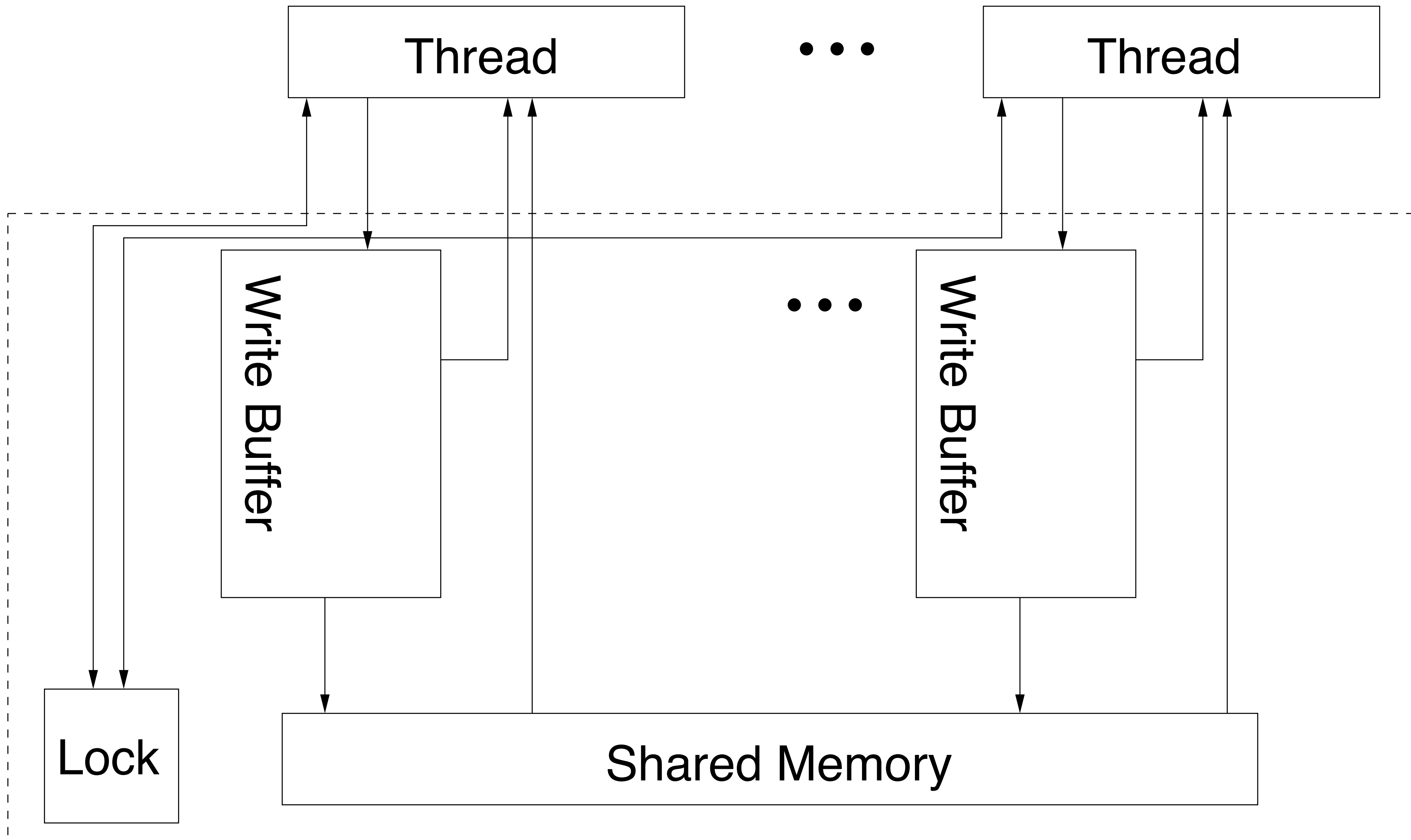
Inventing a Usable Abstraction

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

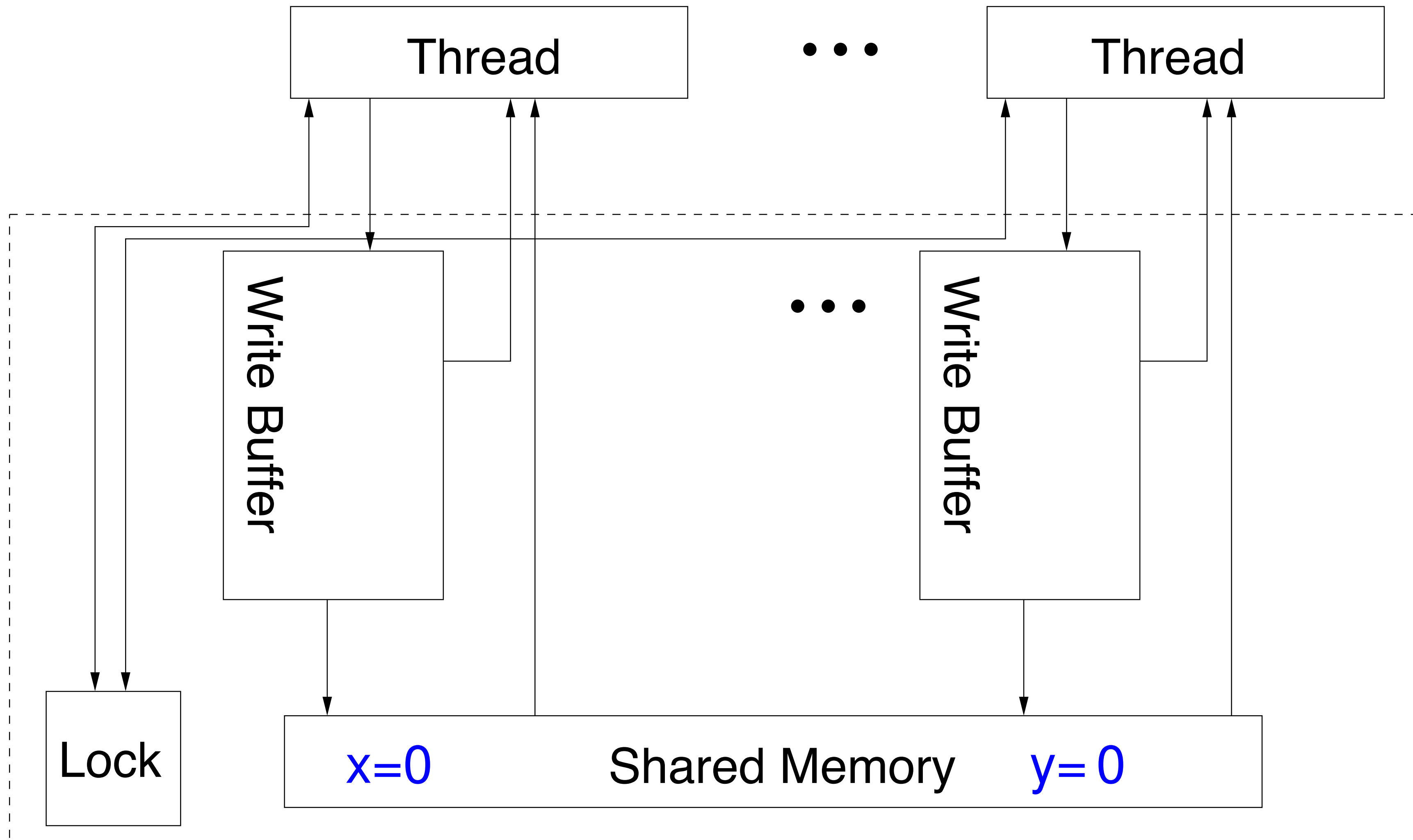
These suggest that x86 is, in practice, like SPARC TSO.

x86-TSO Abstract Machine



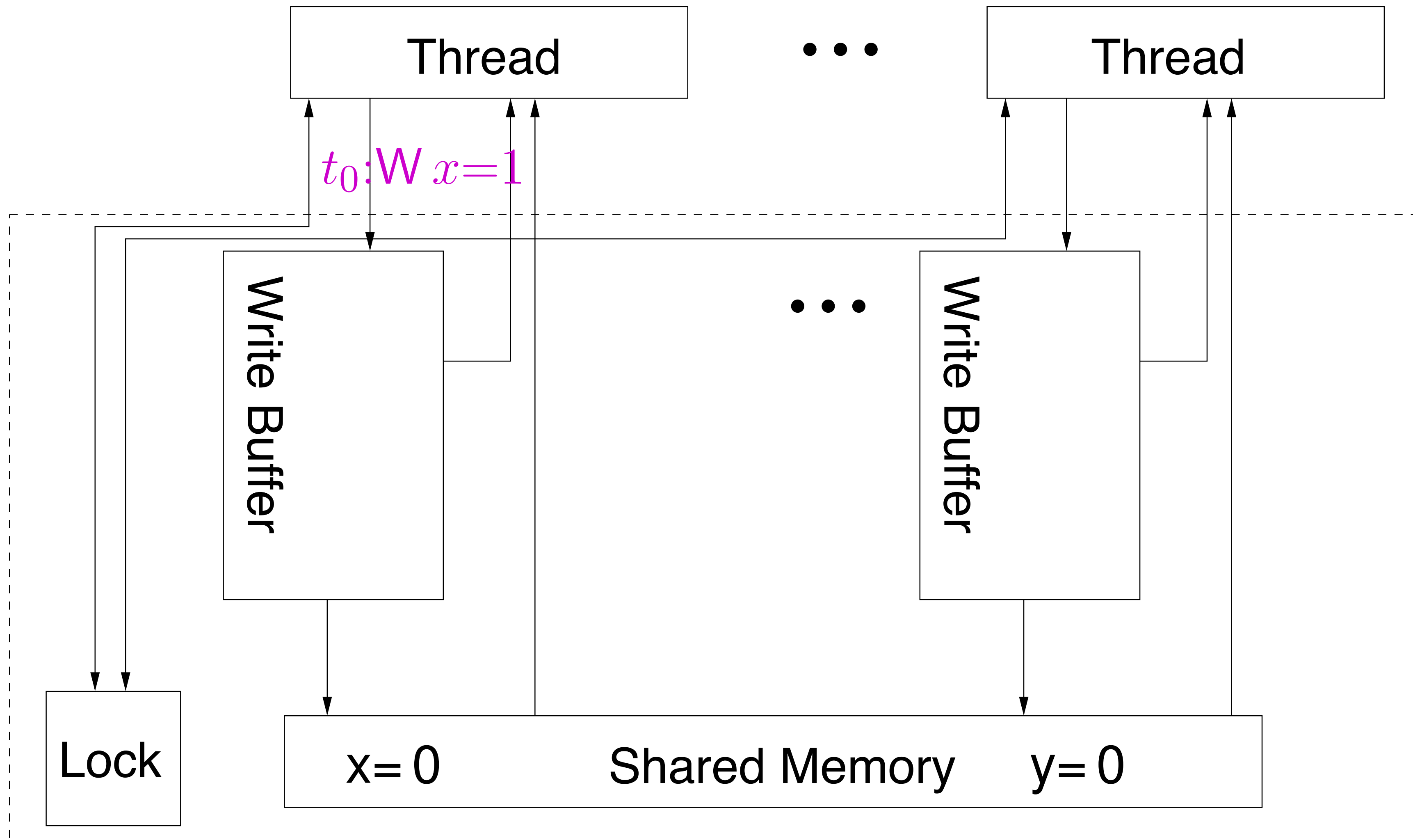
SB, on x86

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)



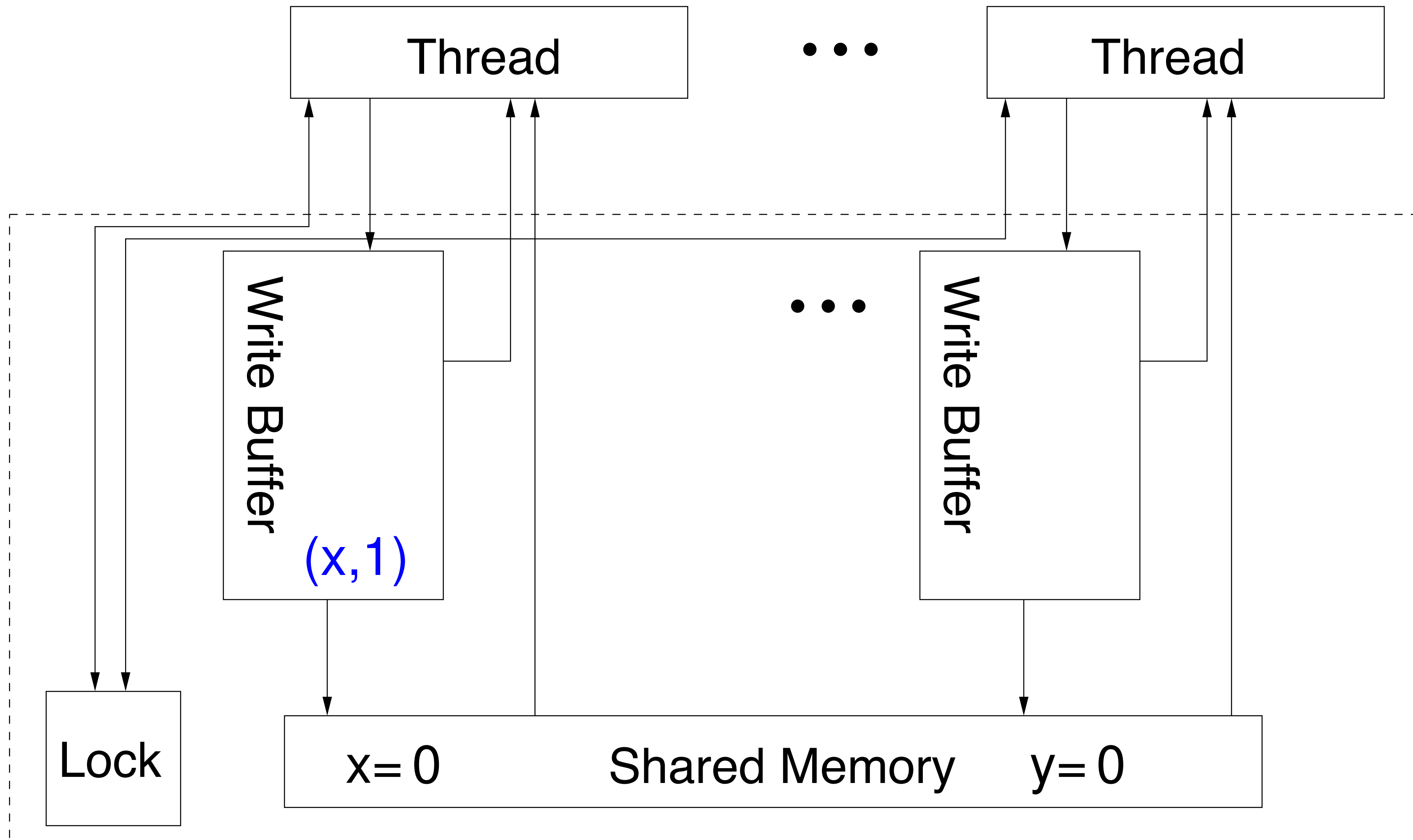
SB, on x86

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)



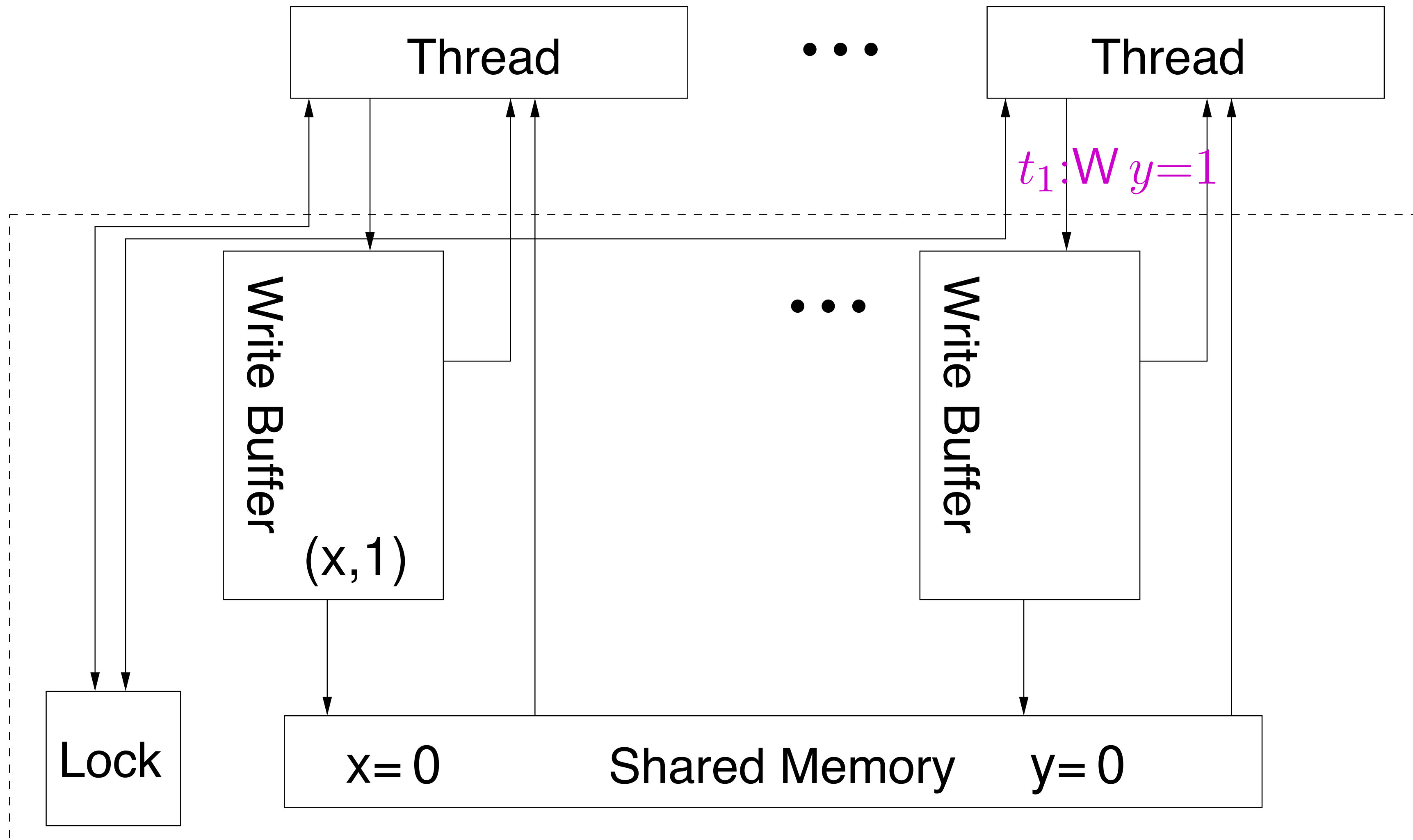
SB, on x86

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)



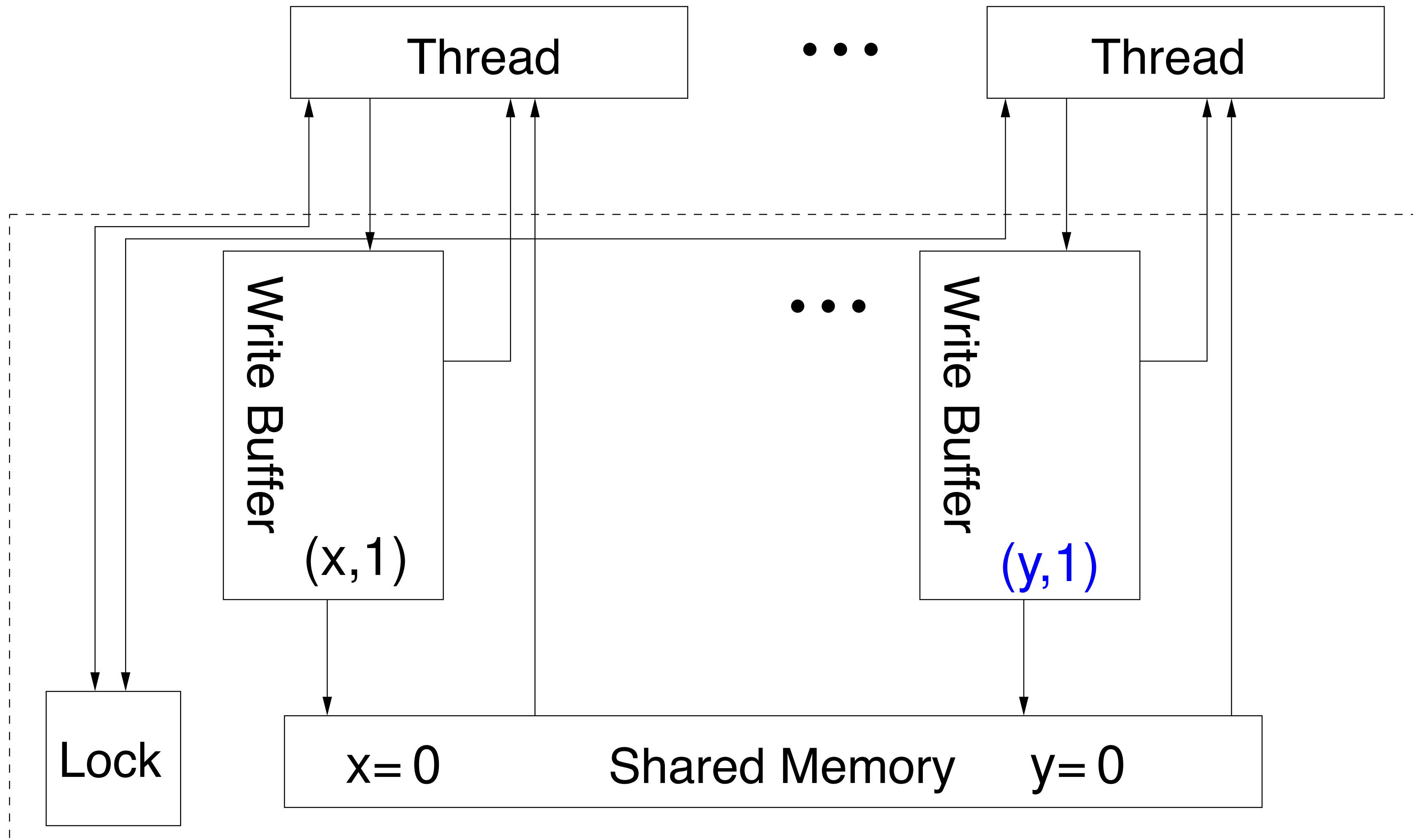
SB, on x86

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)



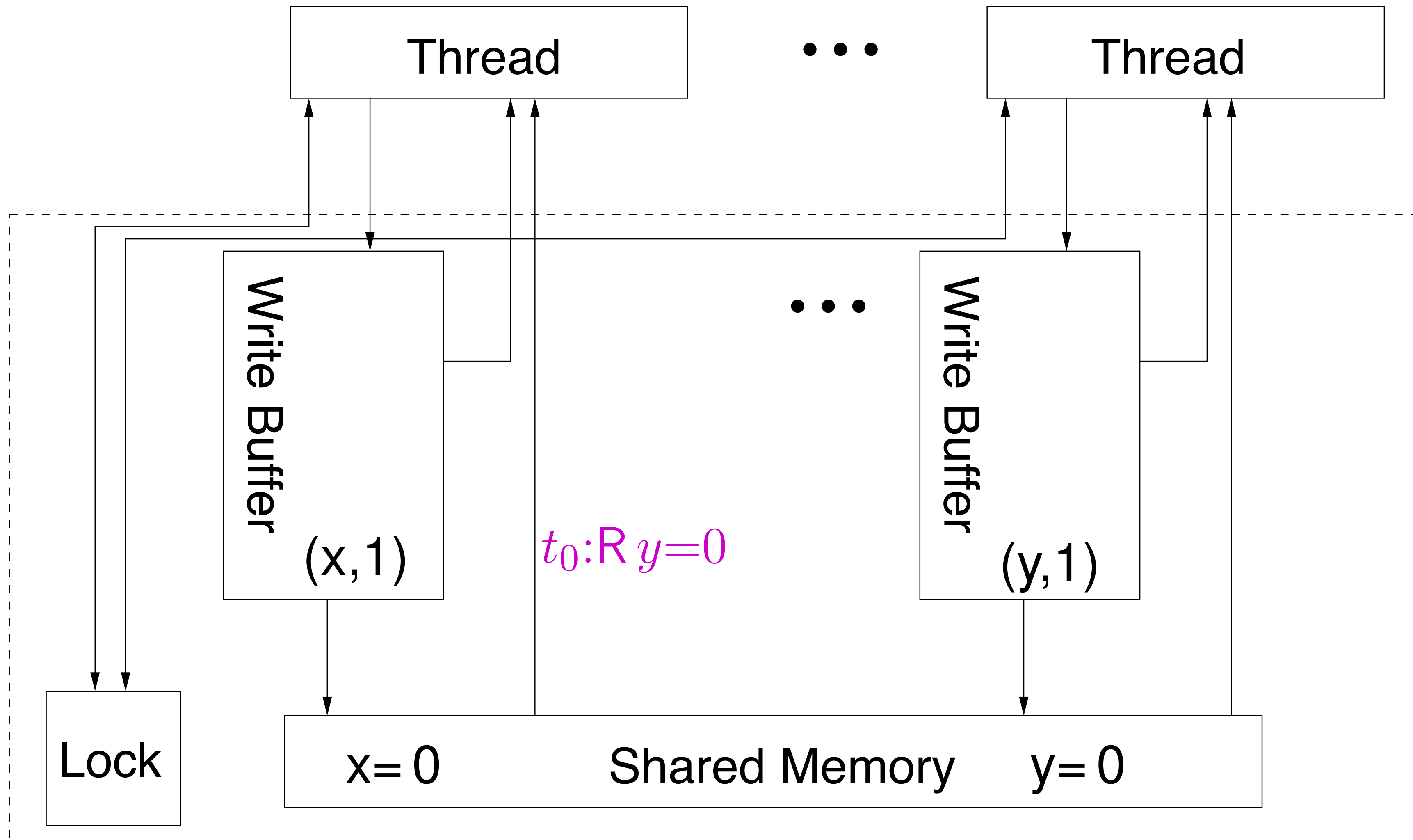
SB, on x86

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)



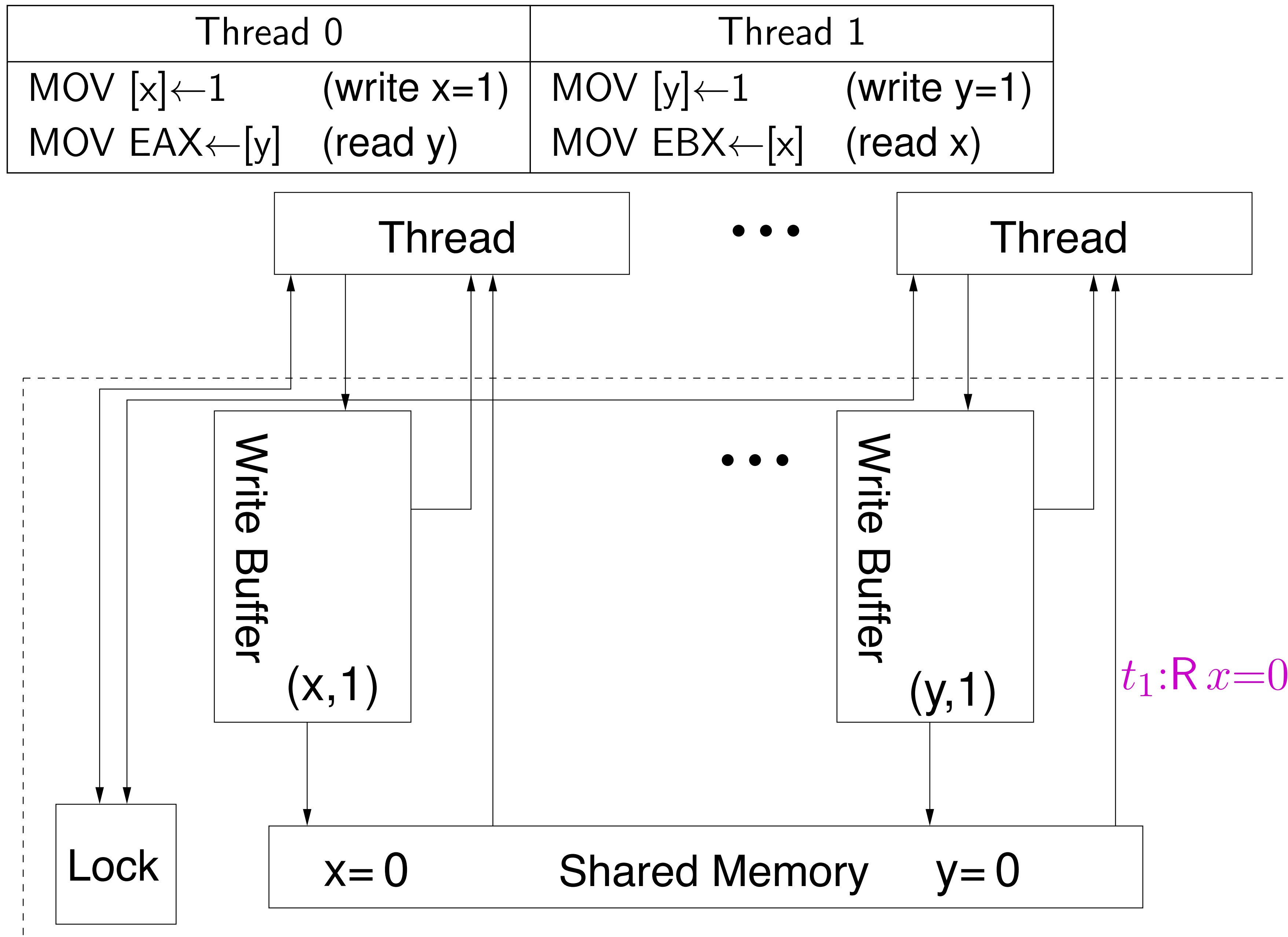
SB, on x86

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)

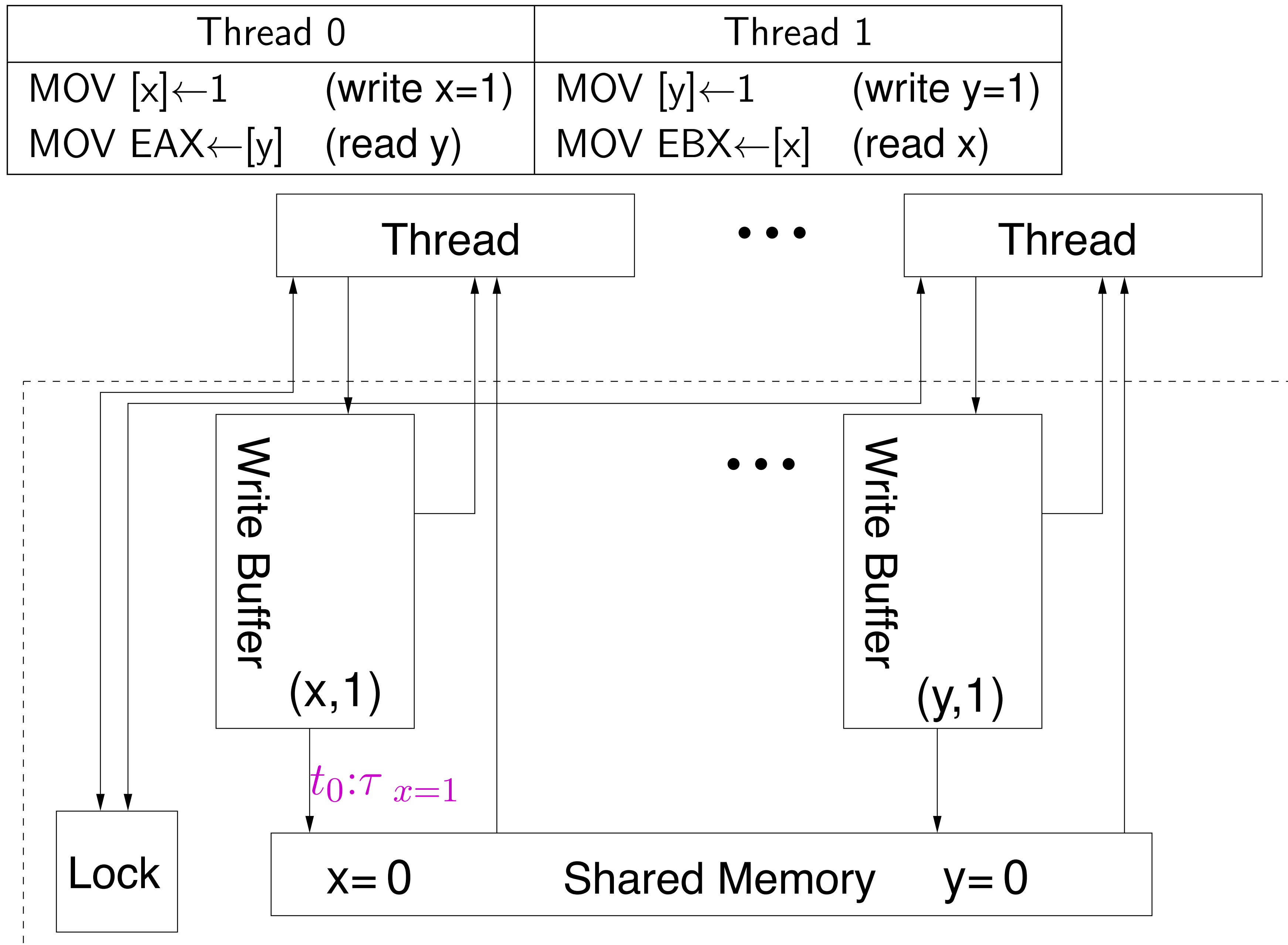


SB, on x86

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)



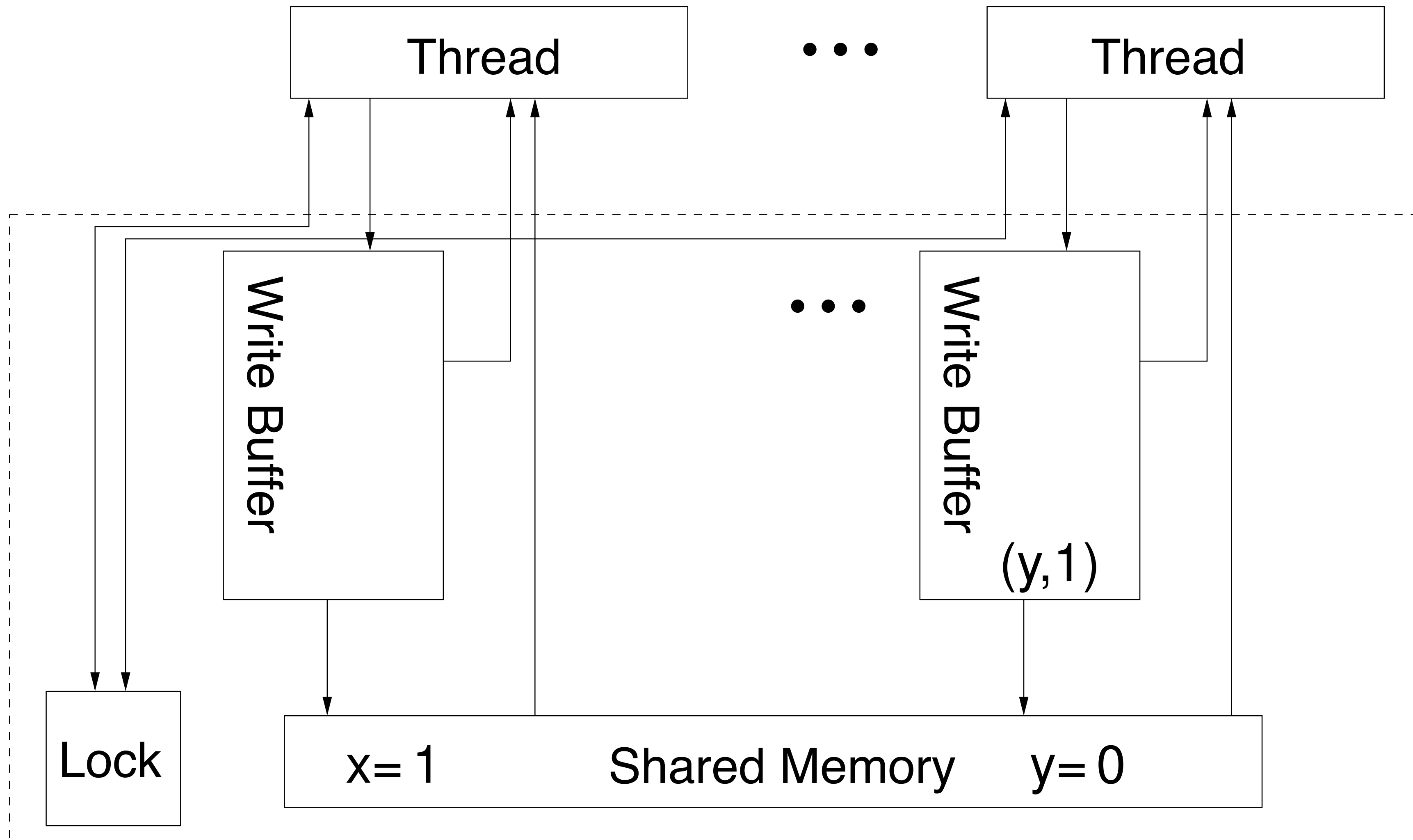
SB, on x86



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)

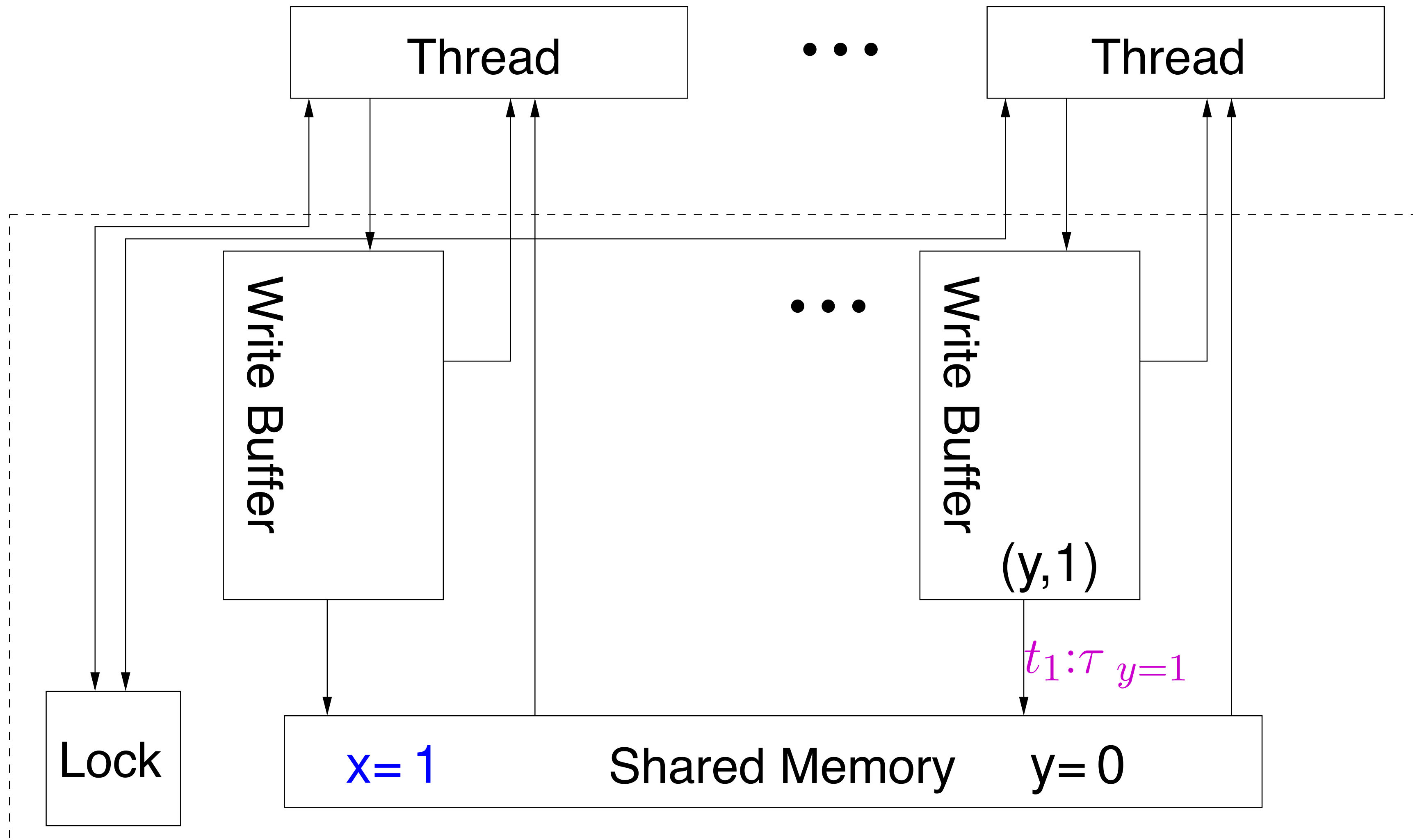
SB, on x86

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)



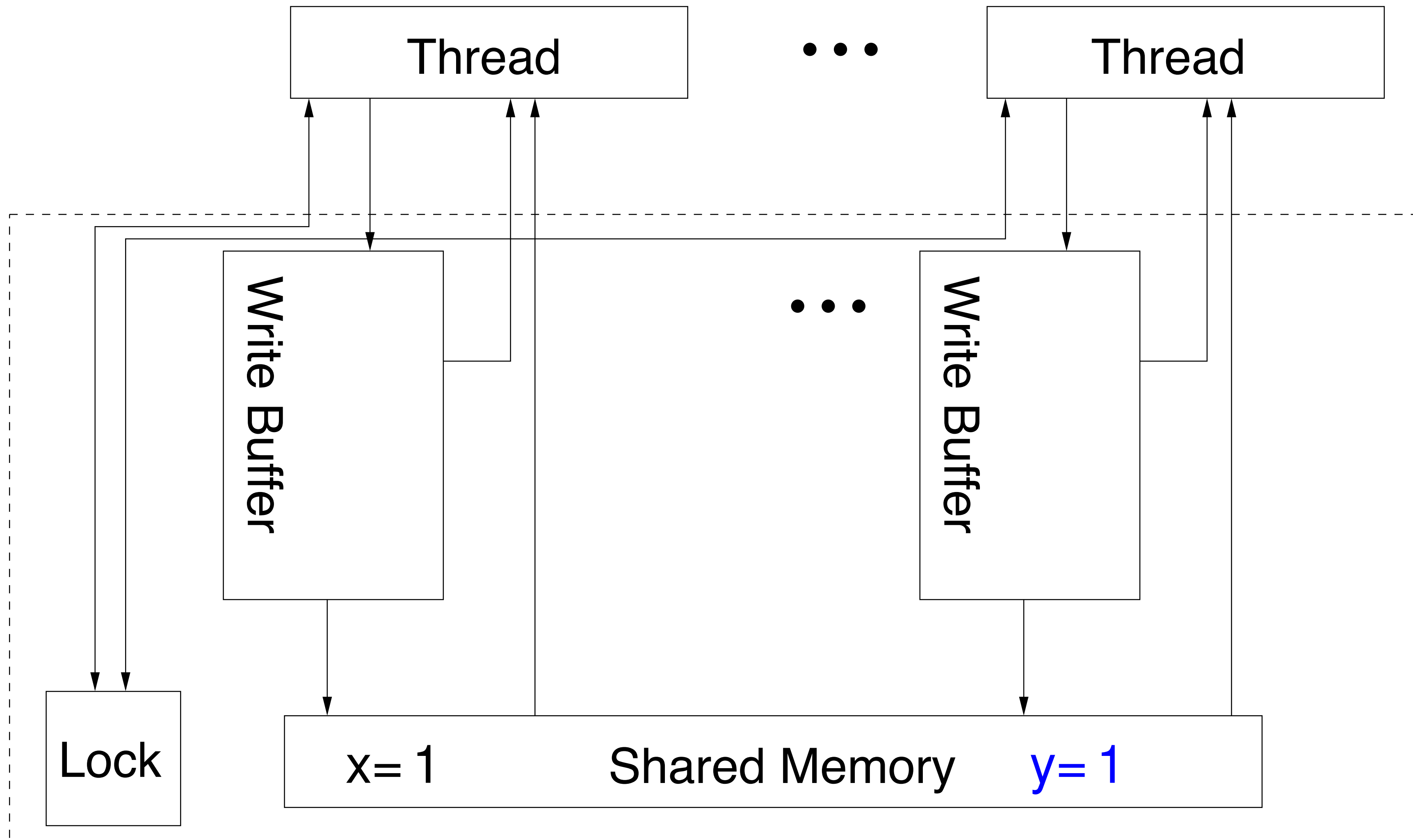
SB, on x86

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)



SB, on x86

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)



How to formally define this?

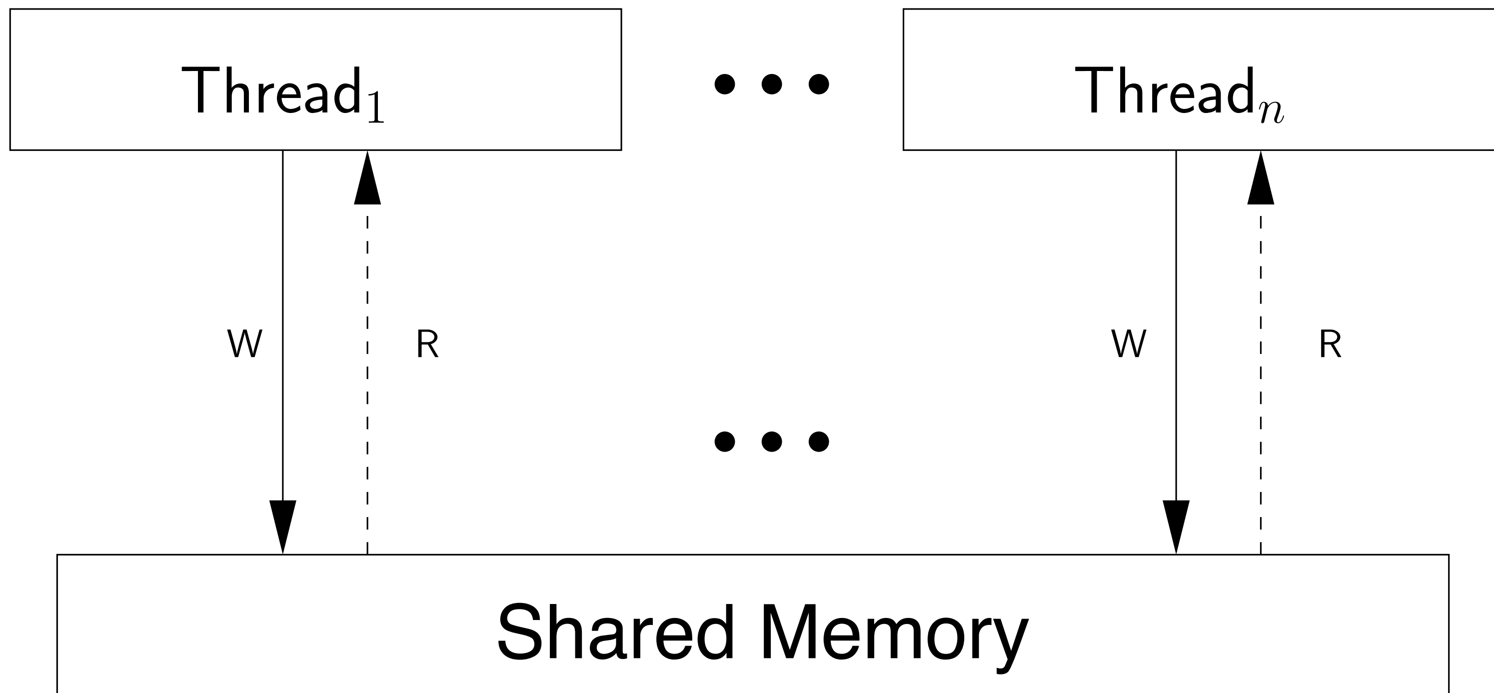
Separate *instruction semantics* and *memory model*

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).

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



Multiple threads acting on a *sequentially consistent* (SC) shared memory

A Tiny Language

$location,\ x,\ m$	address
$integer,\ n$	integer
$thread_id,\ t$	thread id

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

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

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

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

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

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

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

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

$$\frac{e_1 \xrightarrow{l} e_1'}{e_1 + e_2 \xrightarrow{l} e_1' + e_2} \quad \text{PLUS_CONTEXT_1}$$

$$\frac{e_2 \xrightarrow{l} e_2'}{n_1 + e_2 \xrightarrow{l} n_1 + e_2'} \quad \text{PLUS_CONTEXT_2}$$

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

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

$$(x = y); x$$

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

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

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: 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'

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

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

Defining an SC Semantics: whole-system states

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

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

$$\frac{p \xrightarrow{t:l} p' \qquad M \xrightarrow{t:l} M'}{\langle p,\ M \rangle \xrightarrow{t:l} \langle p',\ M' \rangle} \quad \text{SSYNC}$$

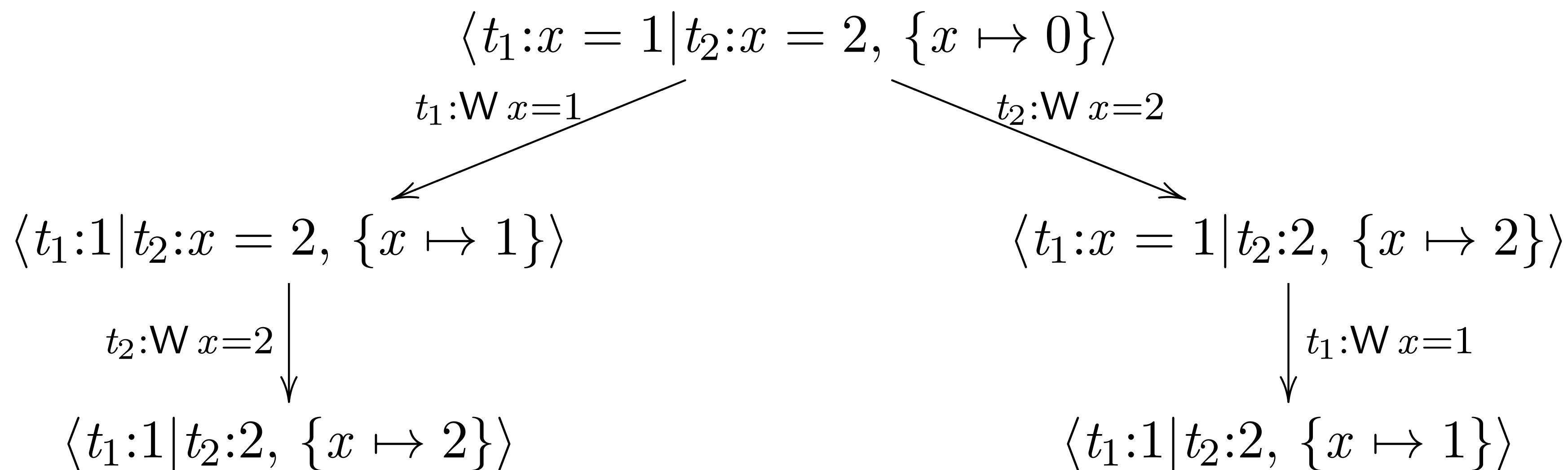
$$\frac{p \xrightarrow{t:\tau} p'}{\langle p,\ M \rangle \xrightarrow{t:\tau} \langle p',\ M \rangle} \quad \text{STAU}$$

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.

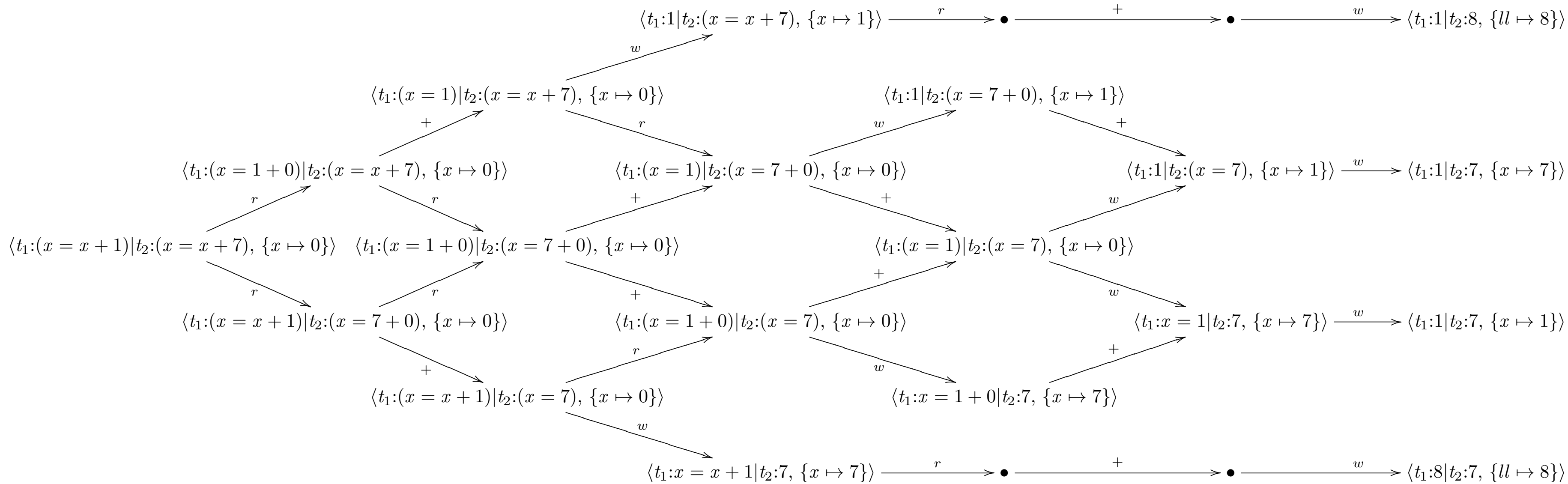
Threads execute asynchronously – the semantics allows any interleaving of the thread transitions. Here there are two:



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.

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

First, more x86 details...

In our 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.

x86 ISA, Locked Instructions

Thread 0	Thread 1
INC x	INC x

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)

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	

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

x86 ISA, Locked Instructions

Compare-and-swap (CAS):

CMPXCHG dest$\leftarrow$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.

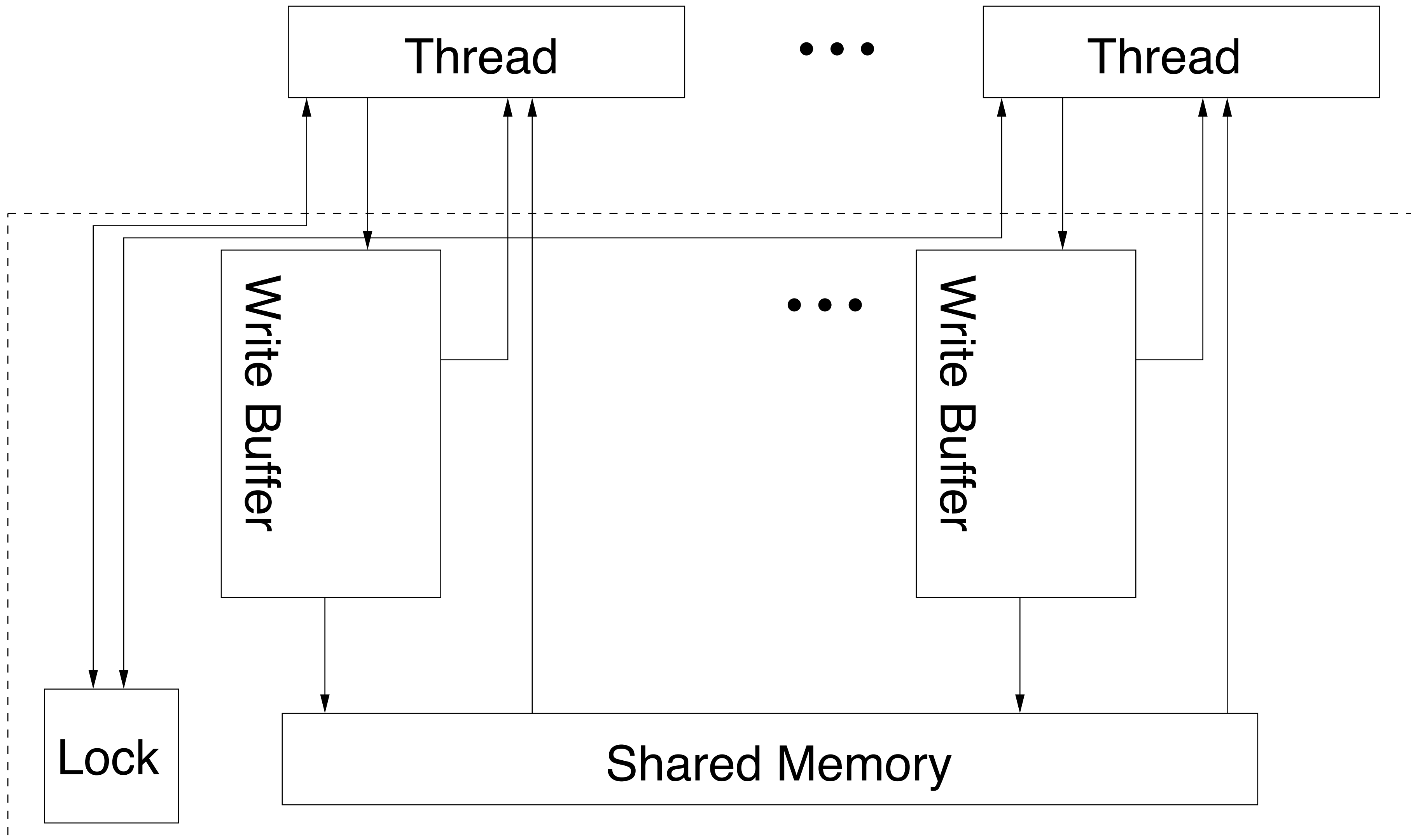
Barriers and LOCK'd Instructions

- 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*

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

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

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 ($s.L = \textsc{Some}\, t$) 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

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

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

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

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

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

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

$$\frac{\begin{array}{l} s.L = \textsf{S}\textsf{OME}(t) \\ s.B(t) = [\,] \end{array}}{s \quad \xrightarrow{t:\textsf{U}} \quad s \oplus \langle\!\![L := \textsf{N}\textsf{ONE}]\!\!\rangle}$$

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

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

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

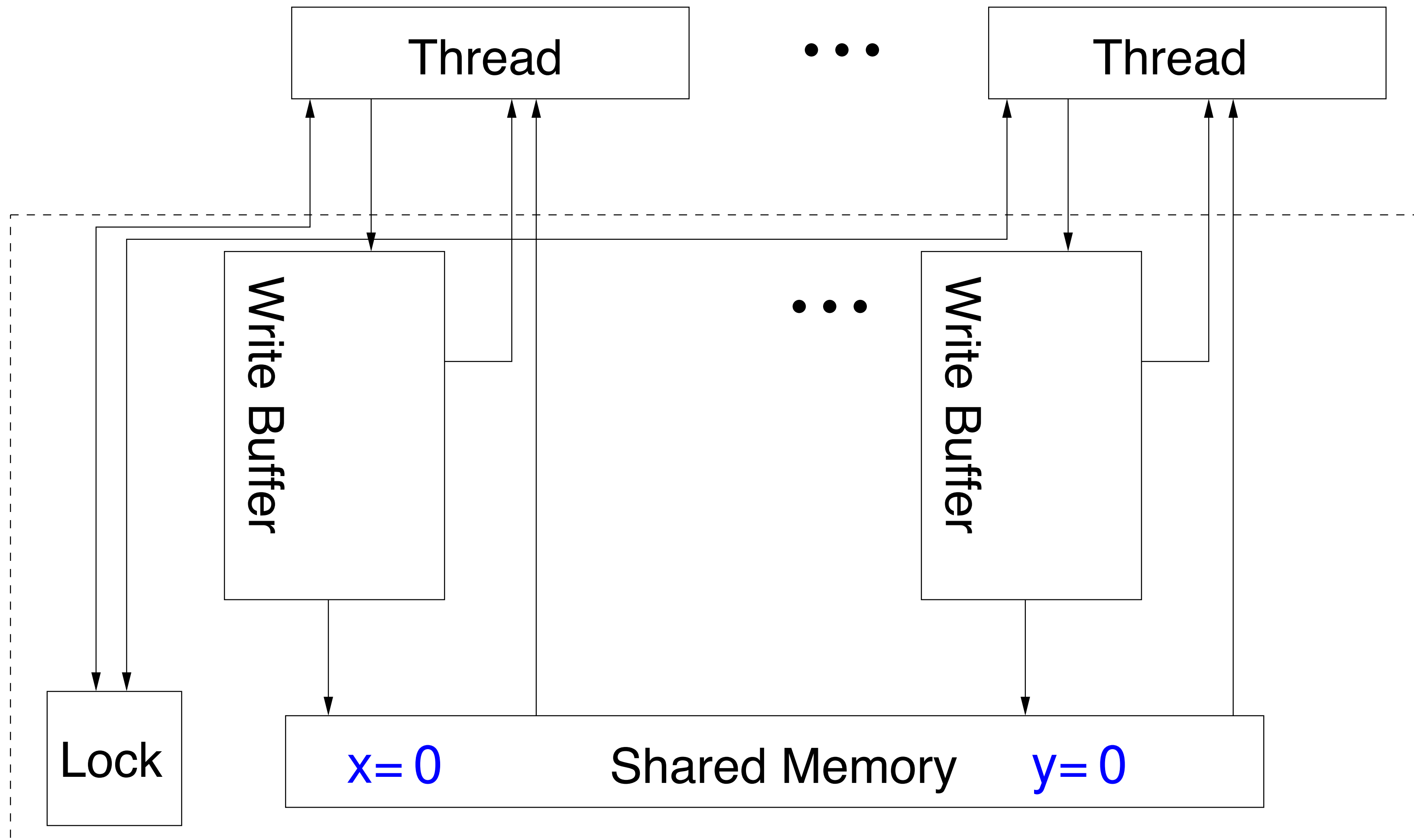
$++$ appends lists

$\cdot \oplus \langle\!\lbrack \cdot := \cdot \rbrack\!\rangle$ updates records

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

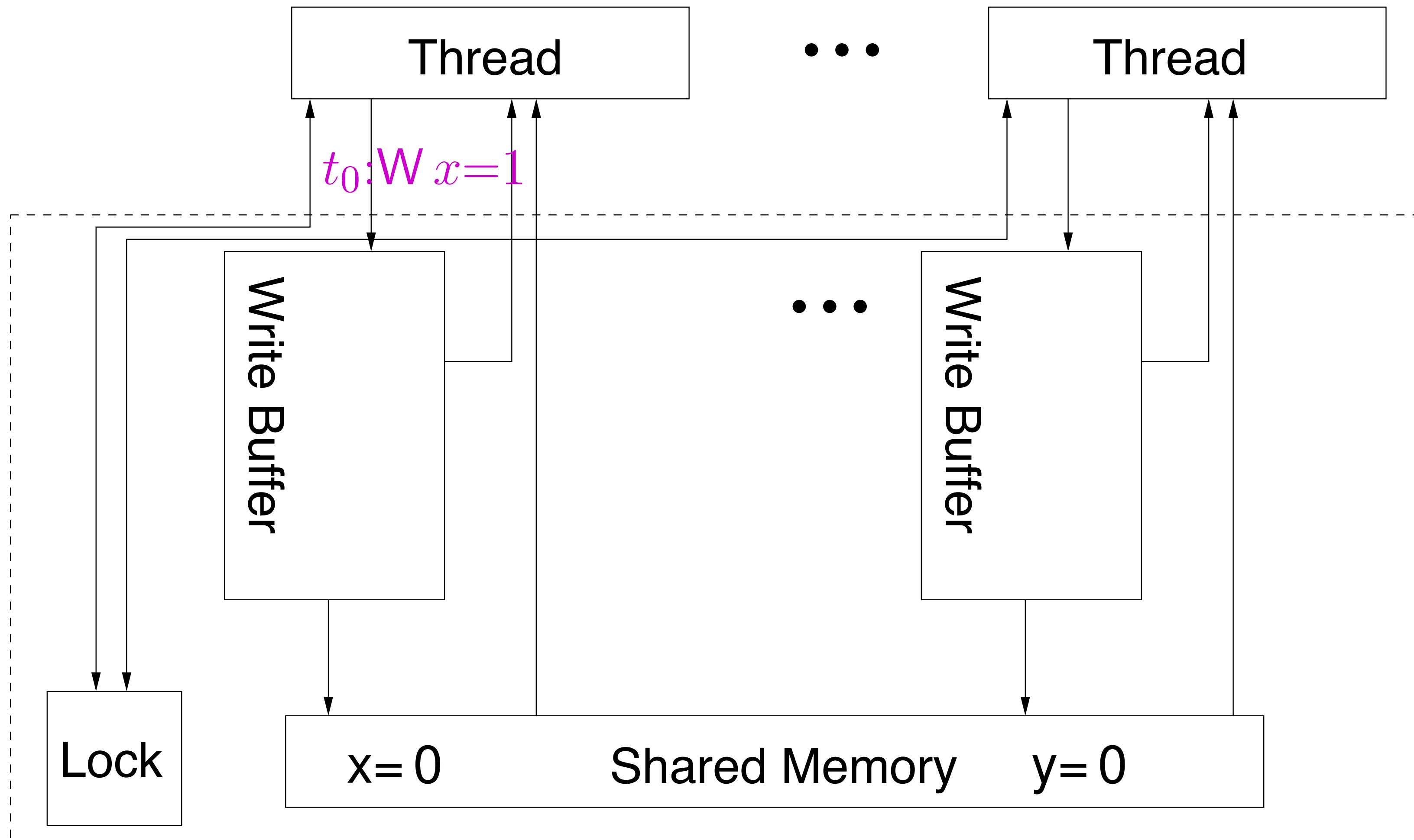
SB, Revisited

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



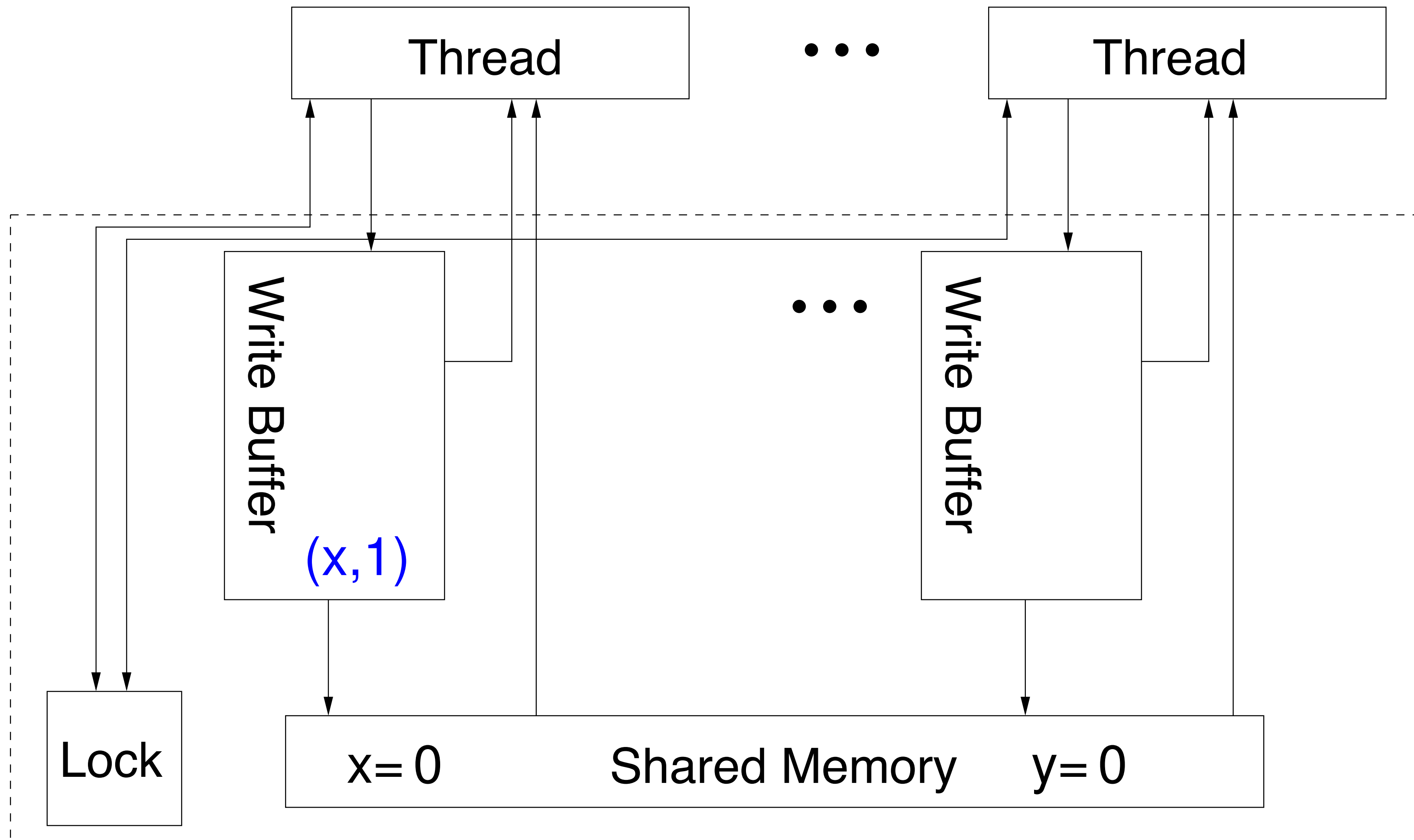
SB, Revisited

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



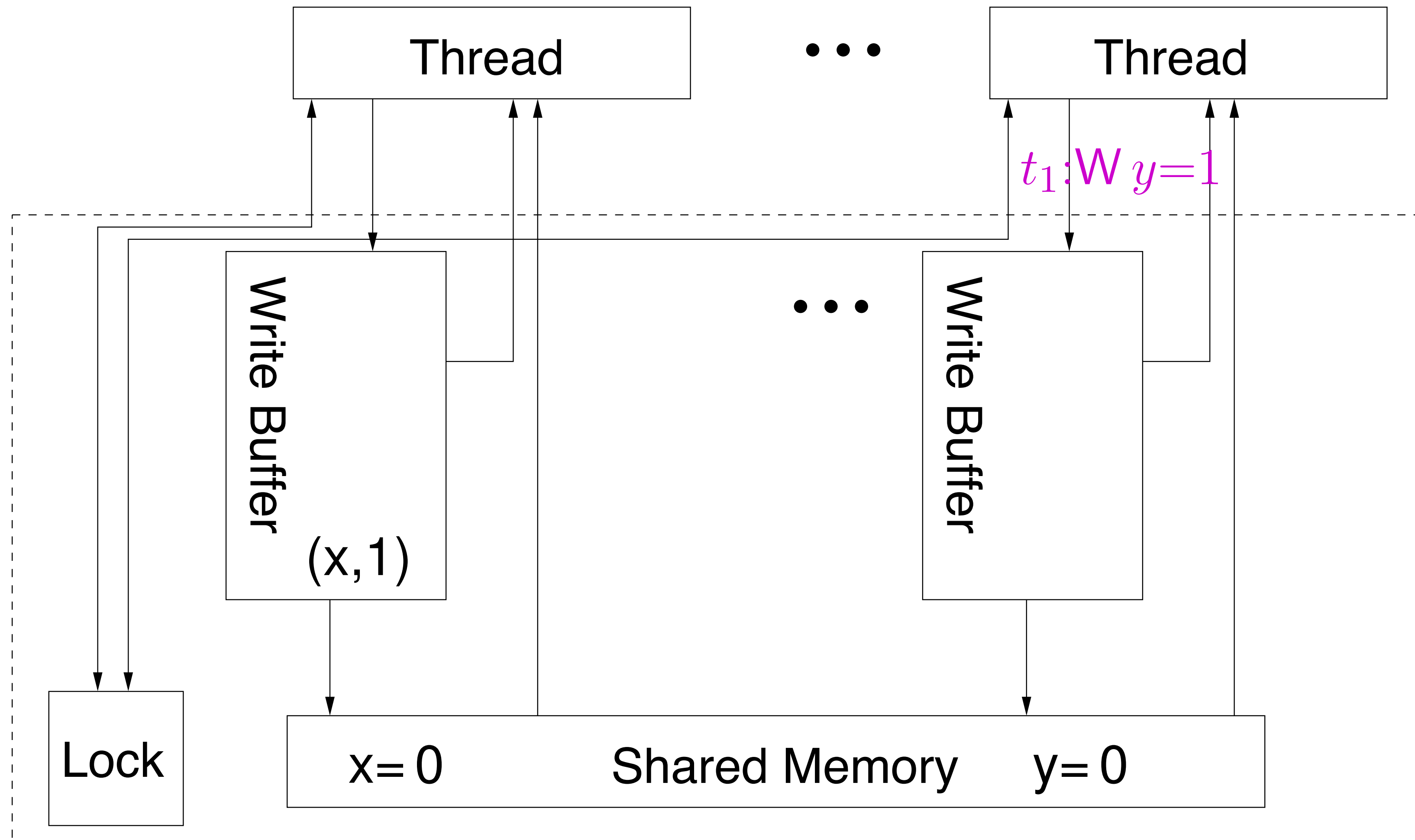
SB, Revisited

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



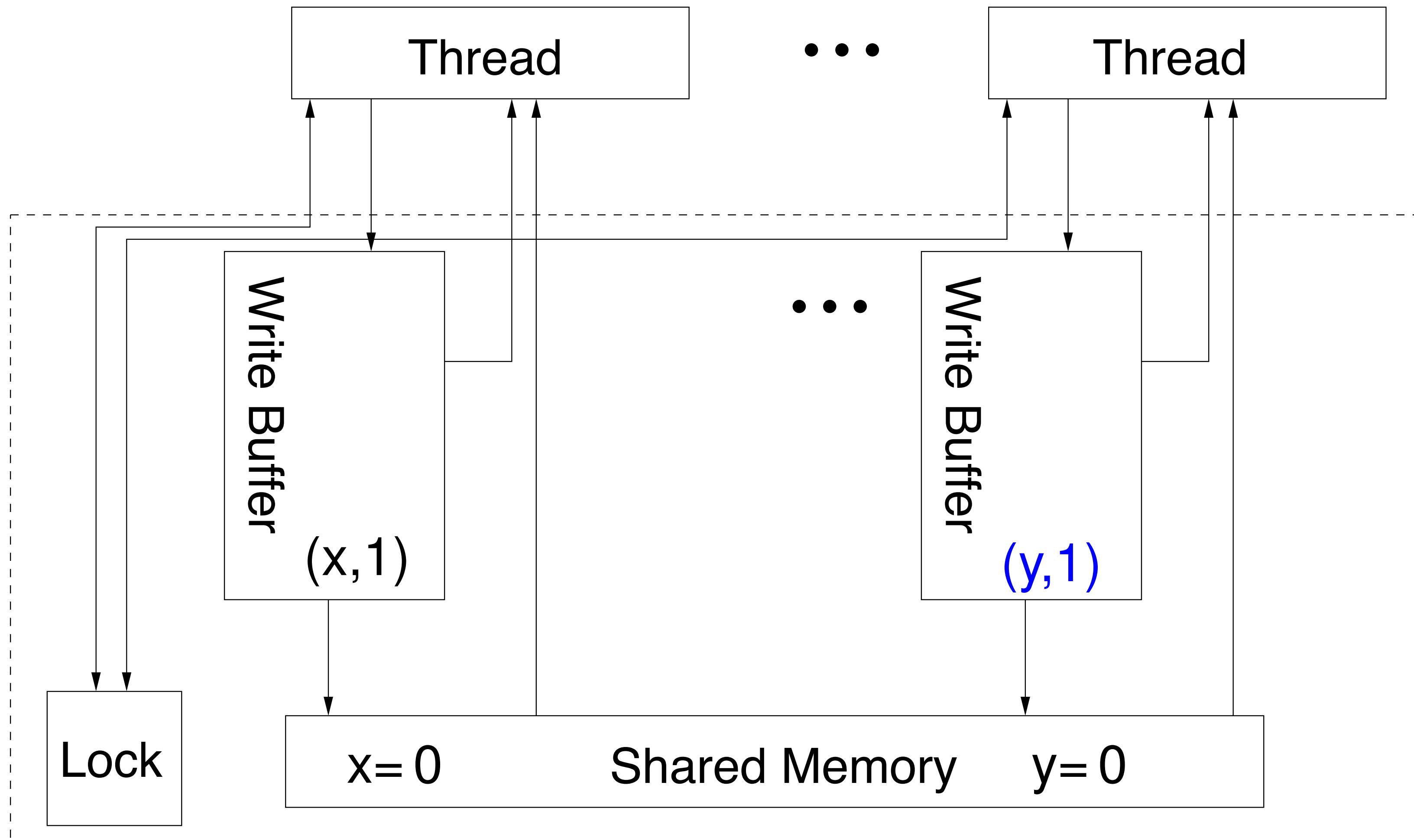
SB, Revisited

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



SB, Revisited

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



SB, Revisited

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

Thread • • • Thread

Write Buffer (x,1) • • • Write Buffer (y,1)

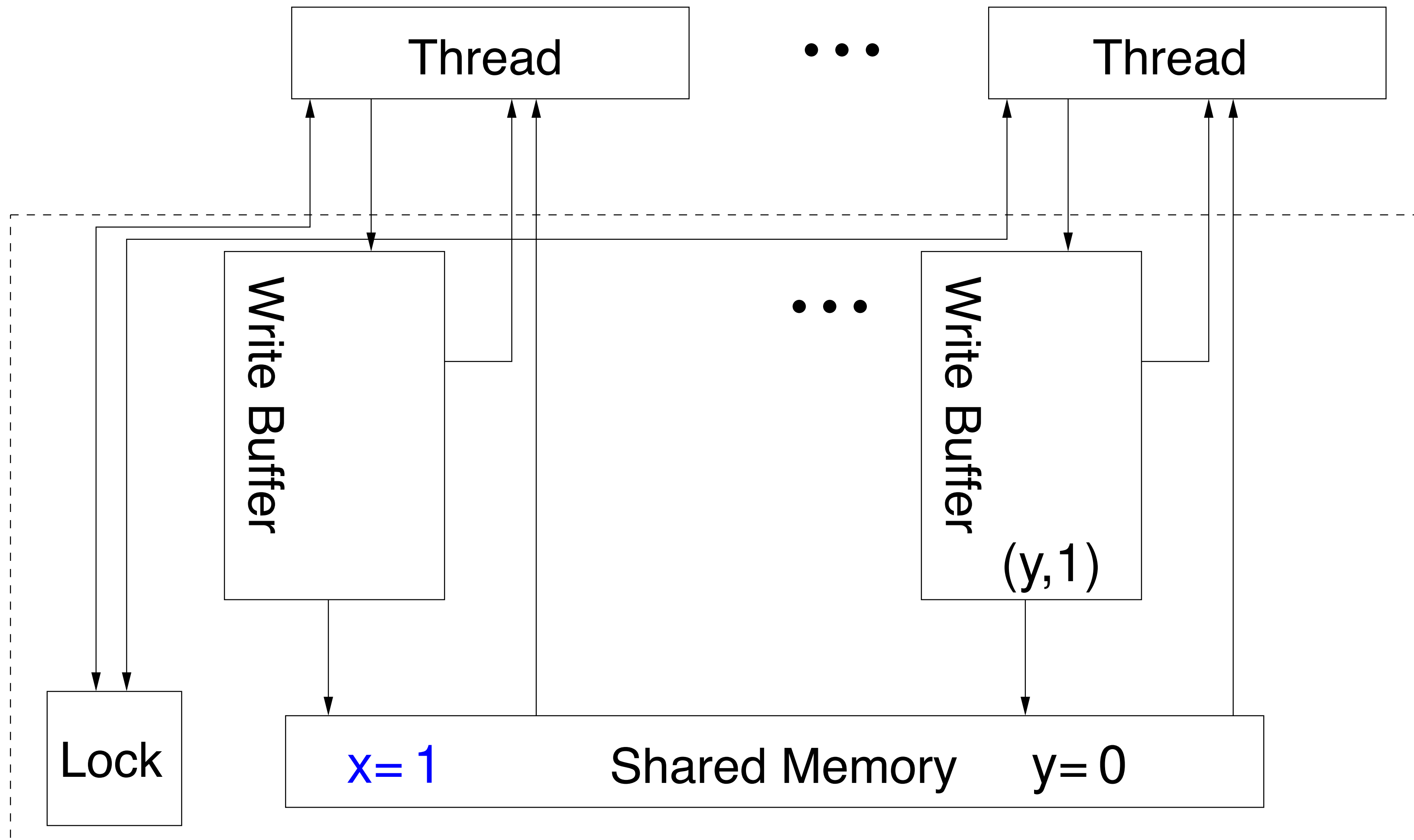
$t_0 : \tau_{x=1}$

Lock

x= 0 Shared Memory y= 0

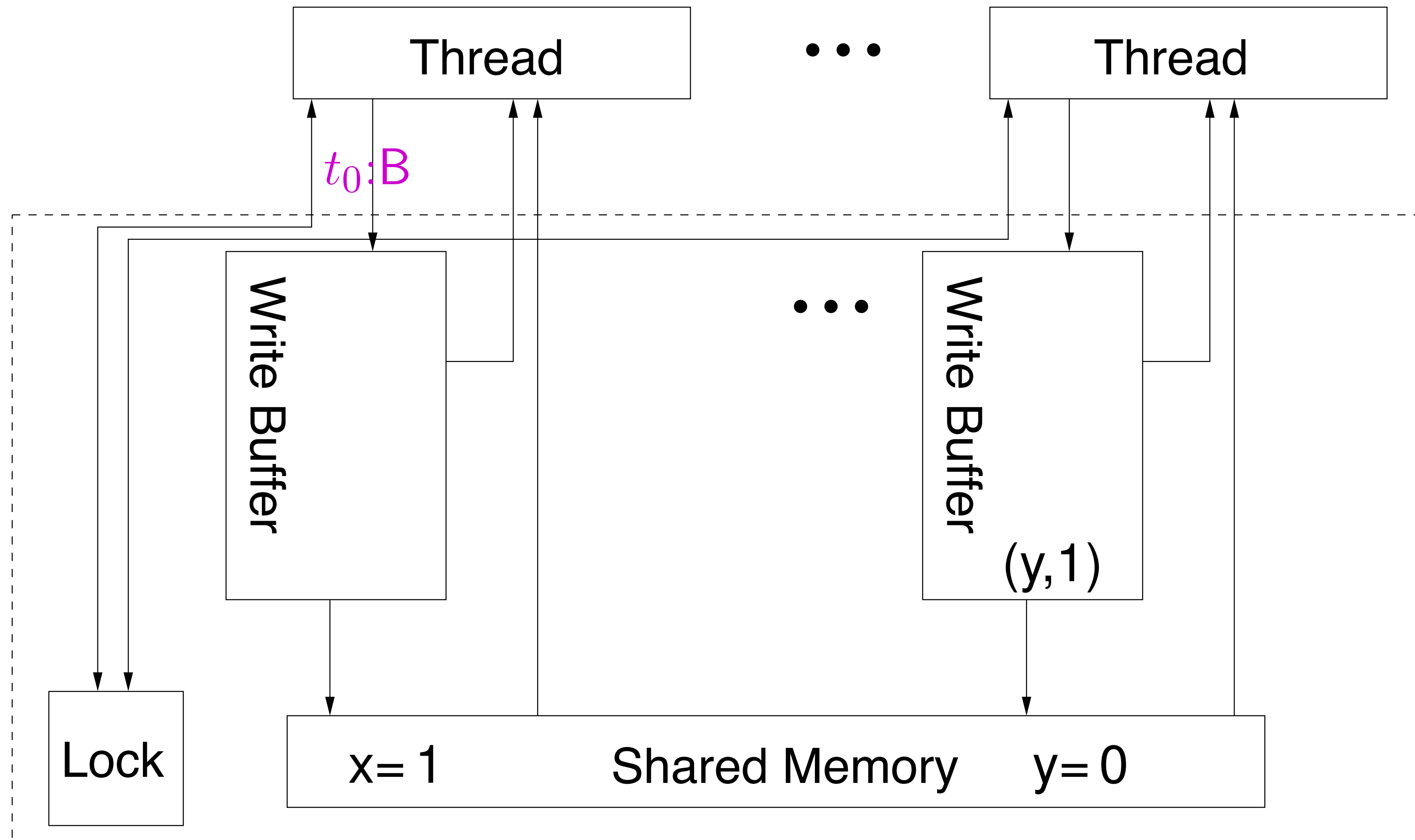
SB, Revisited

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



SB, Revisited

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



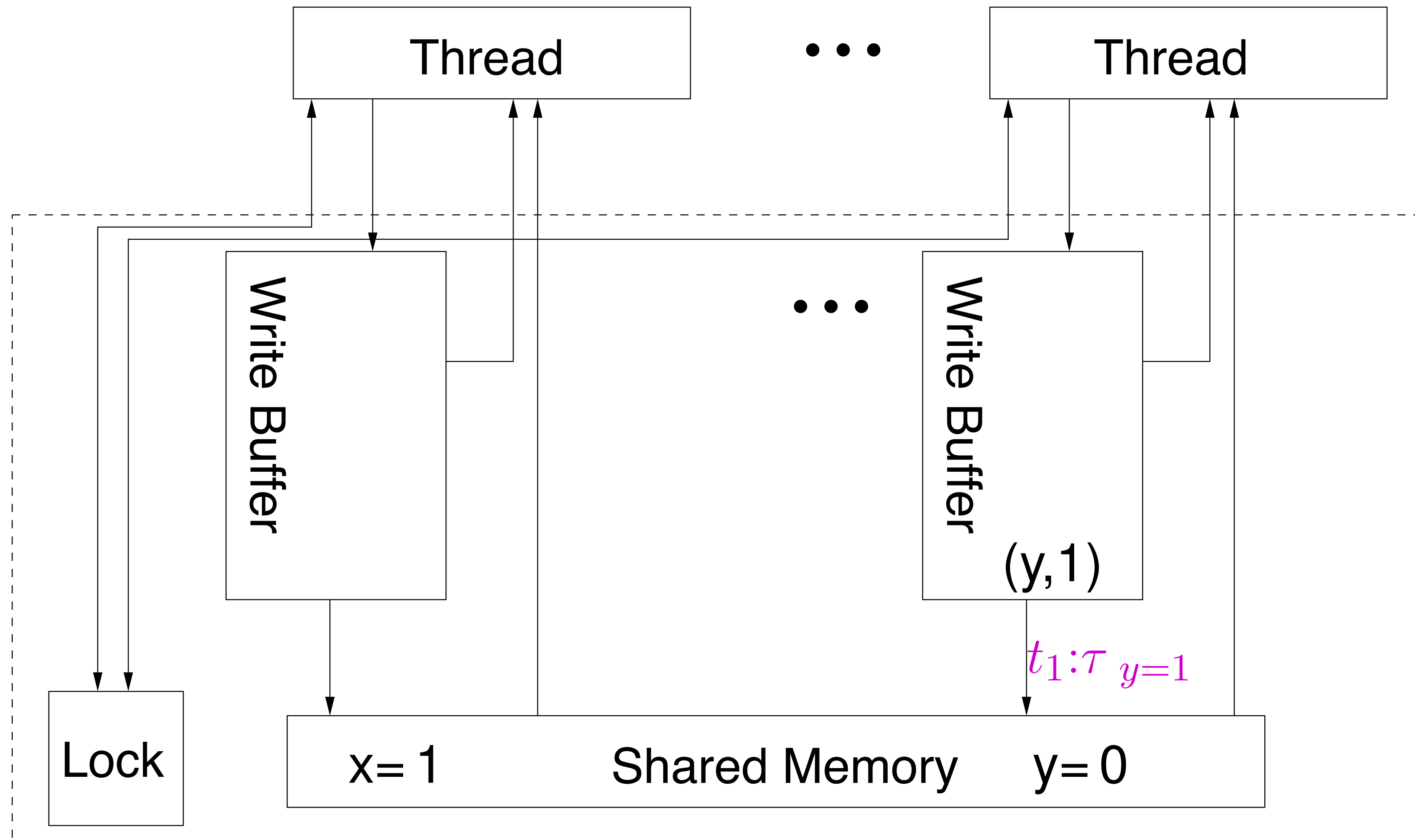
SB, Revisited

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

Thread
• • •
Thread
Write Buffer
• • •
Write Buffer
t_0:R y=0
(y,1)
Lock
x= 1
Shared Memory
y= 0

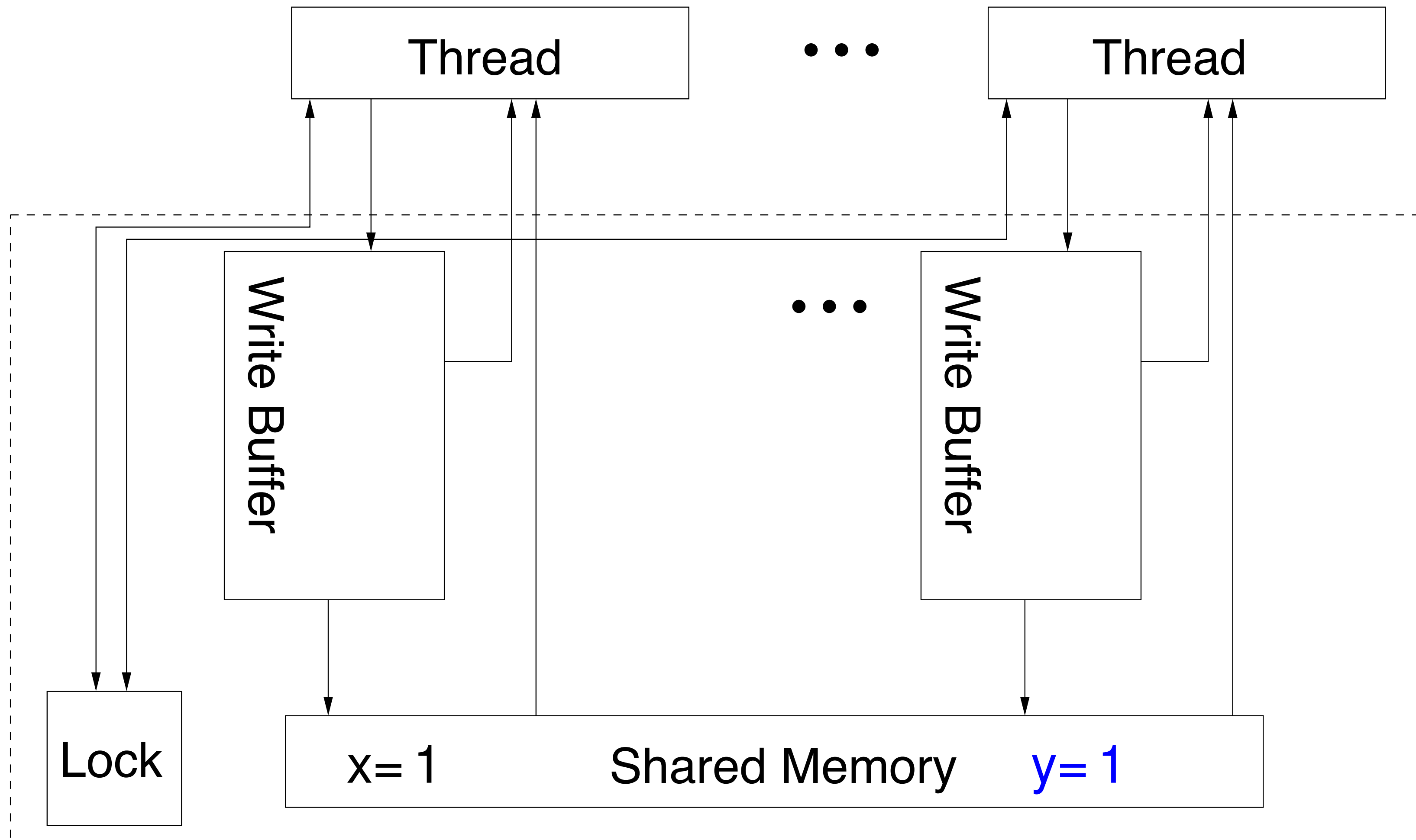
SB, Revisited

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



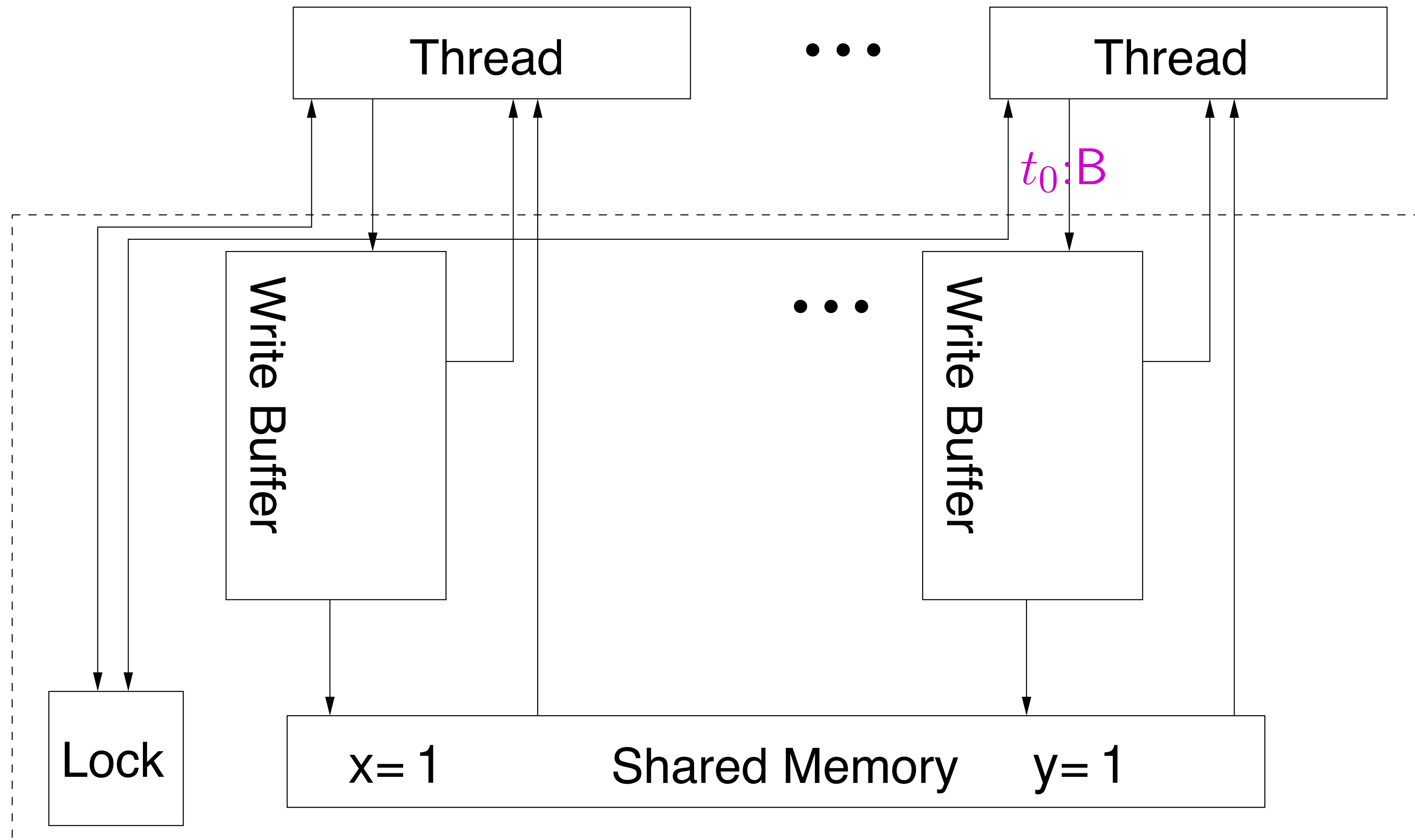
SB, Revisited

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



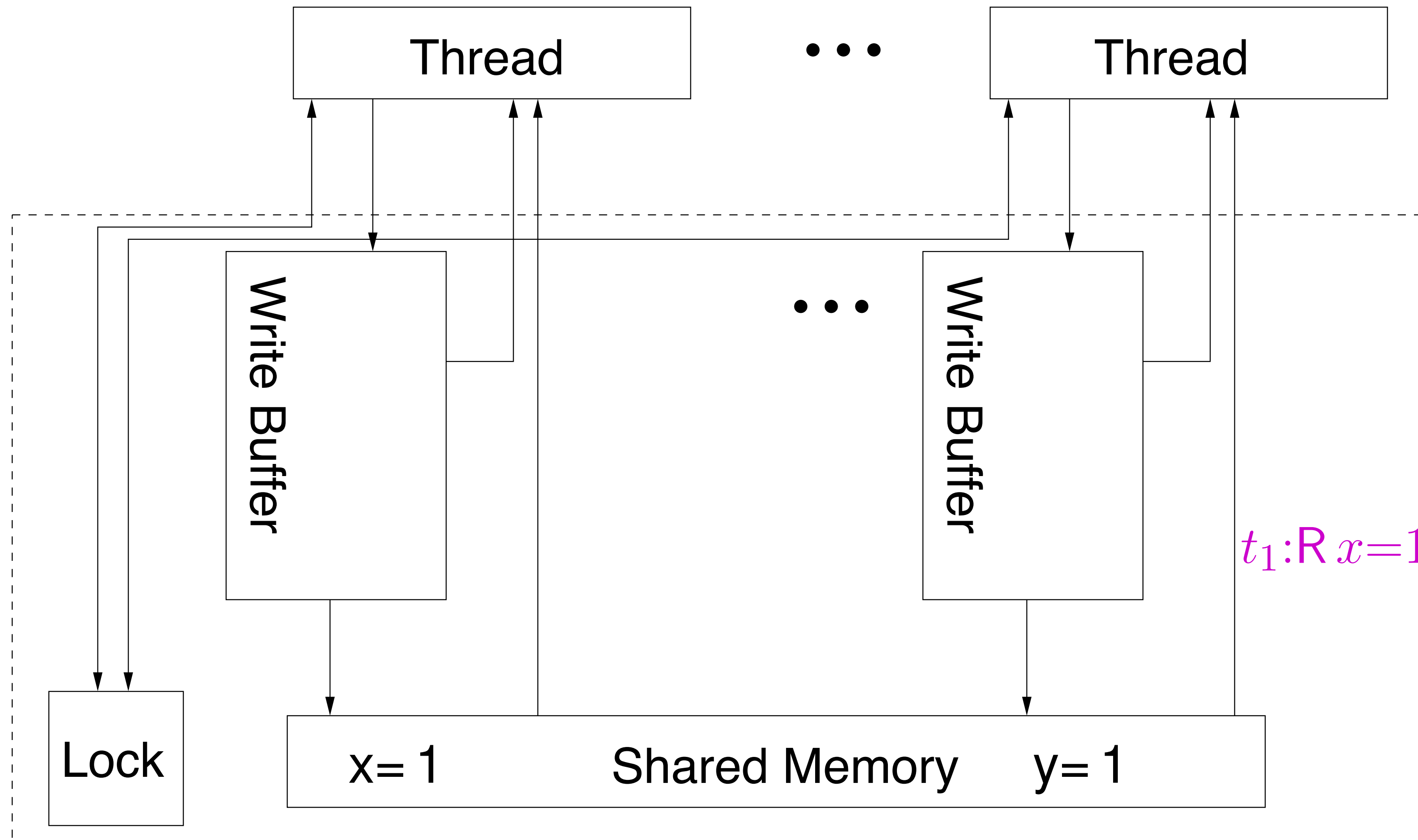
SB, Revisited

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



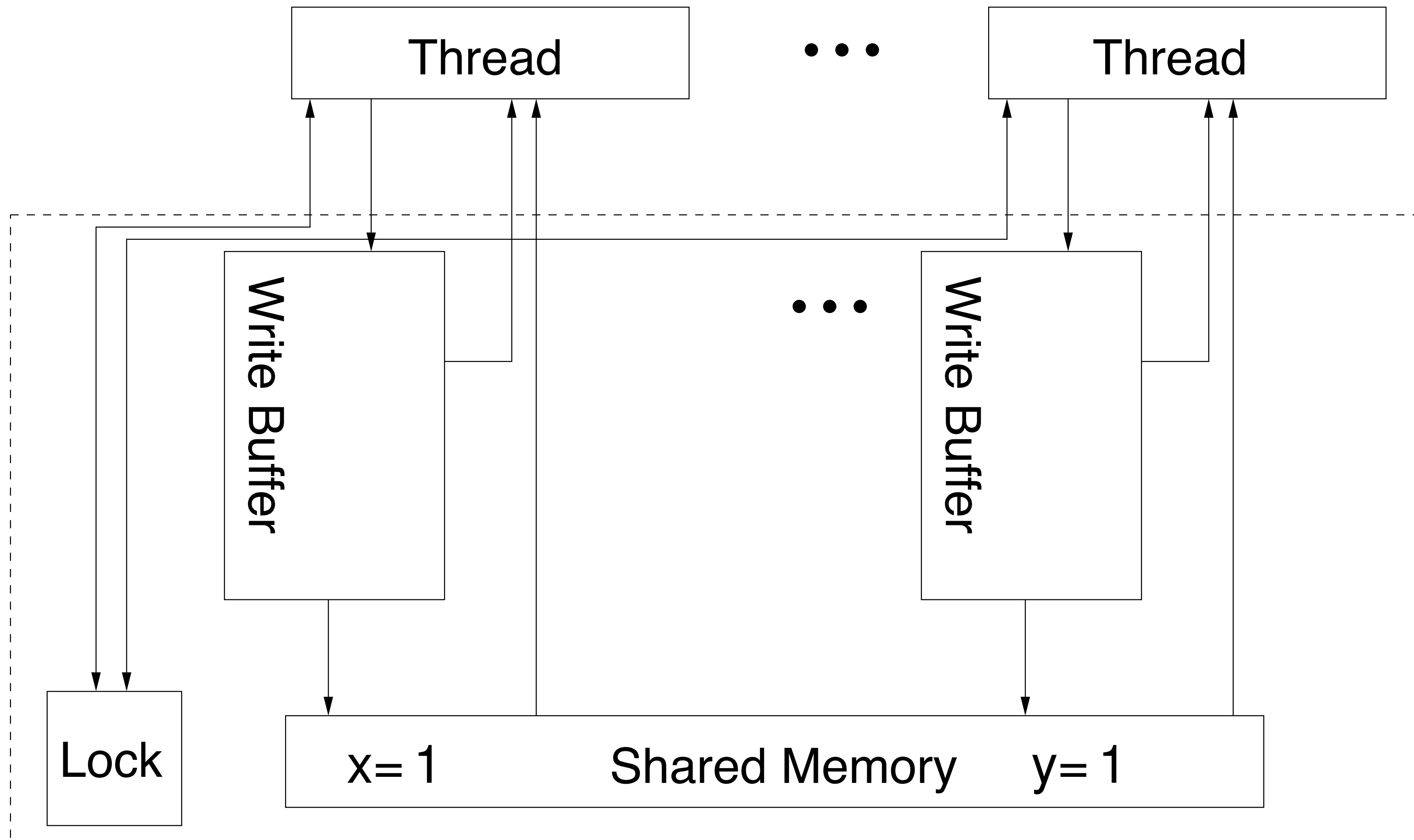
SB, Revisited

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



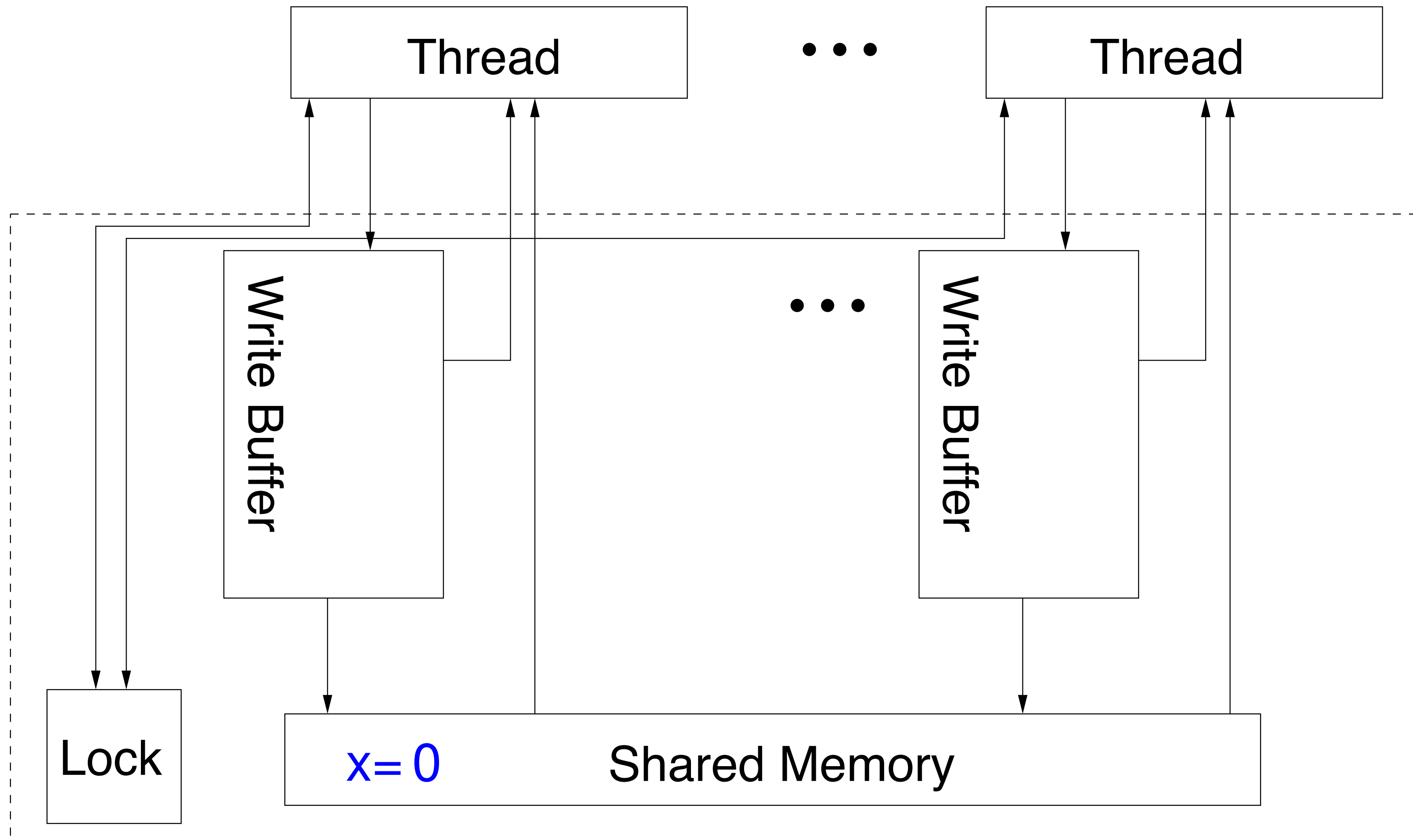
SB, Revisited

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



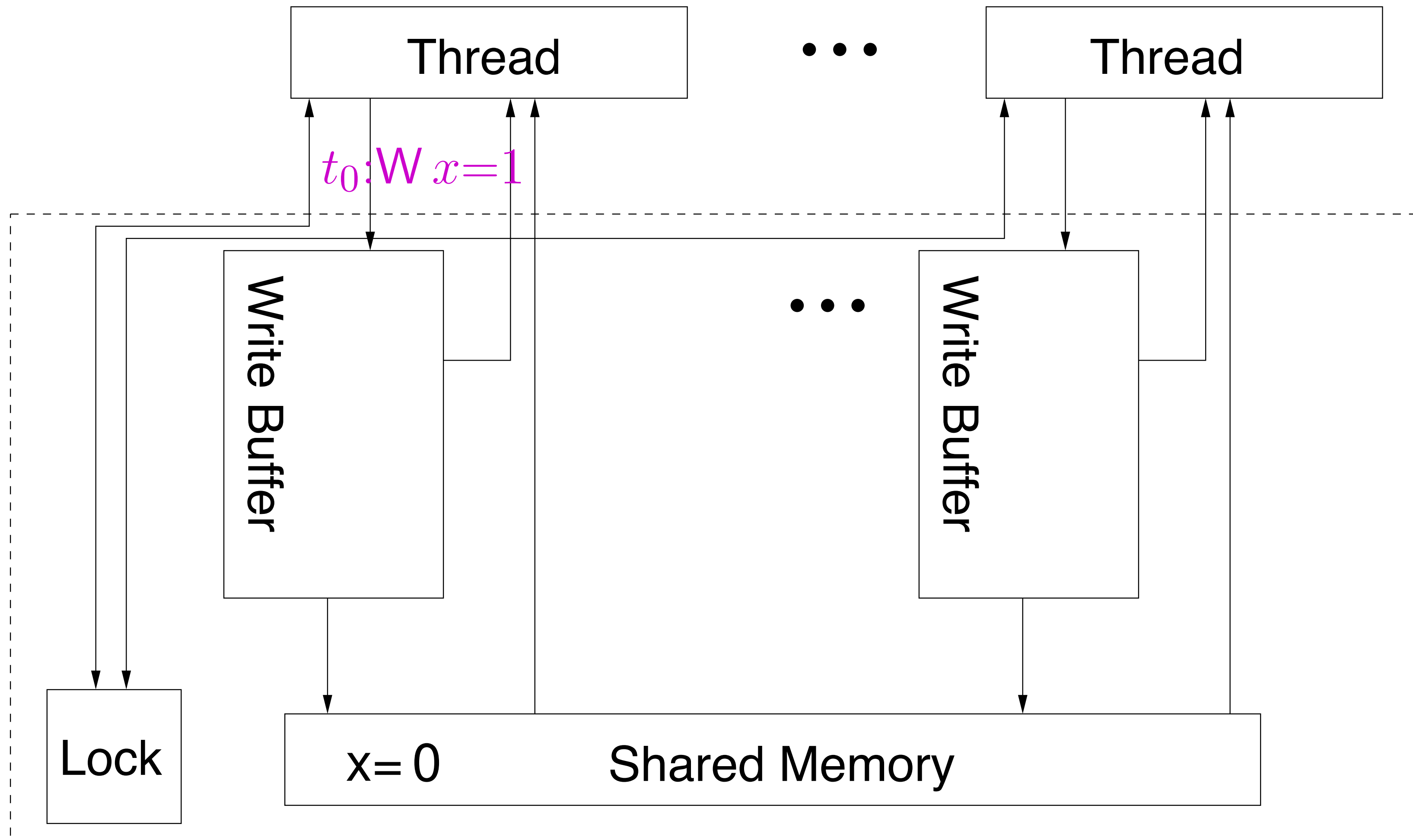
Locked INC

Thread 0	Thread 1
MOV [x]$\leftarrow$1 (write x=1)	
LOCK; INC x	LOCK; INC x



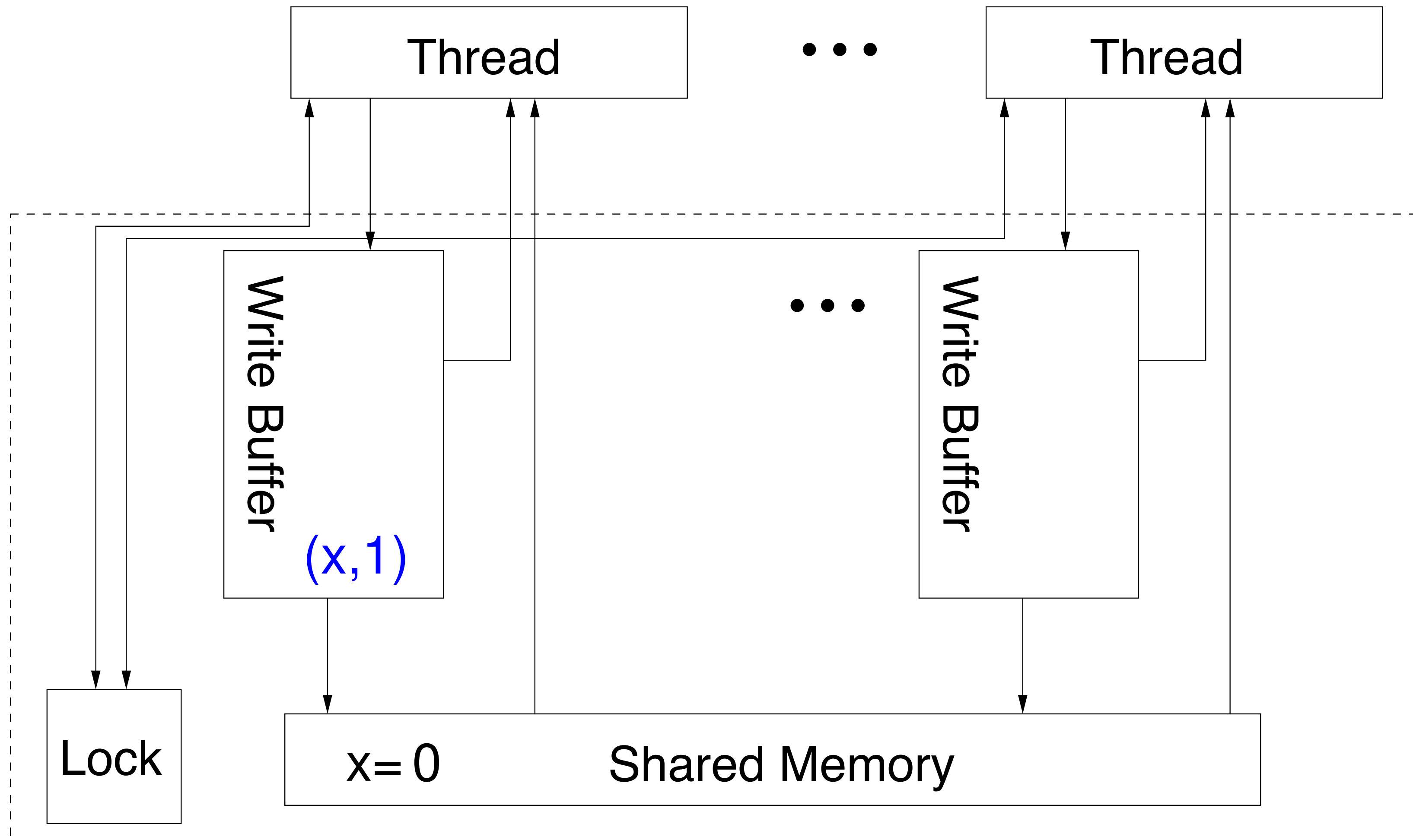
Locked INC

Thread 0	Thread 1
MOV [x]←1 (write x=1)	
LOCK; INC x	LOCK; INC x



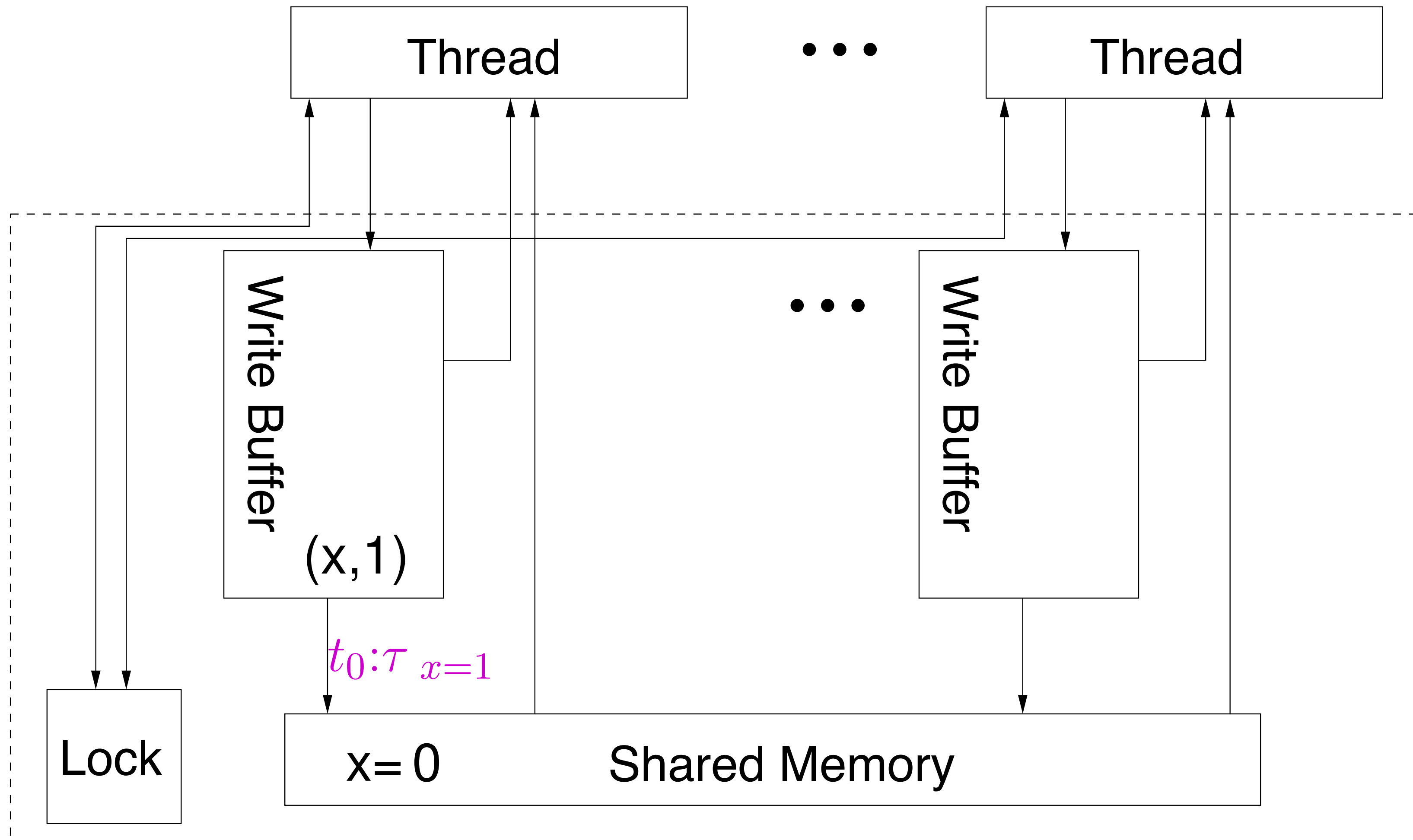
Locked INC

Thread 0	Thread 1
MOV [x]$\leftarrow$1 (write x=1)	
LOCK; INC x	LOCK; INC x



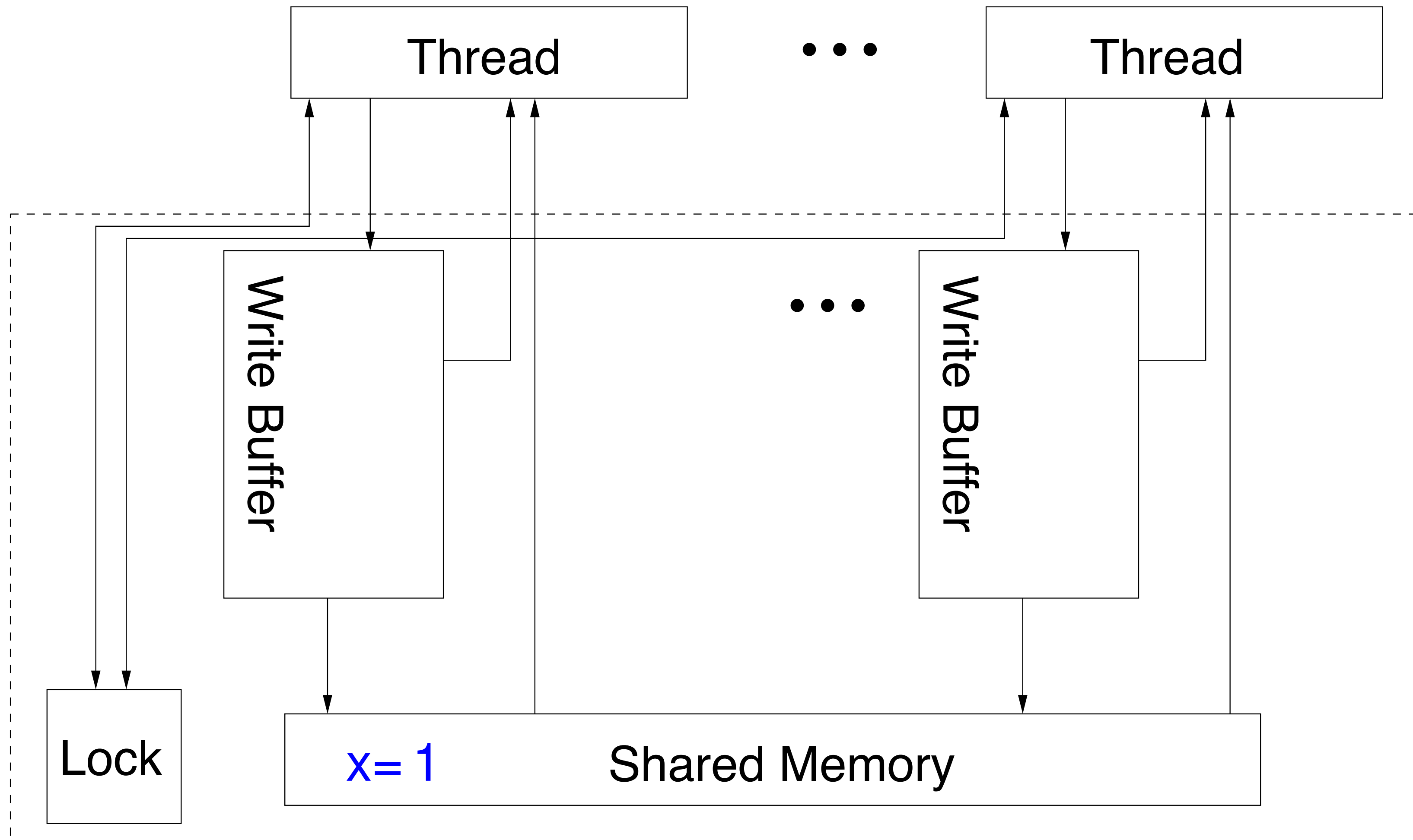
Locked INC

Thread 0	Thread 1
MOV [x]←1 (write x=1) LOCK; INC x	 LOCK; INC x



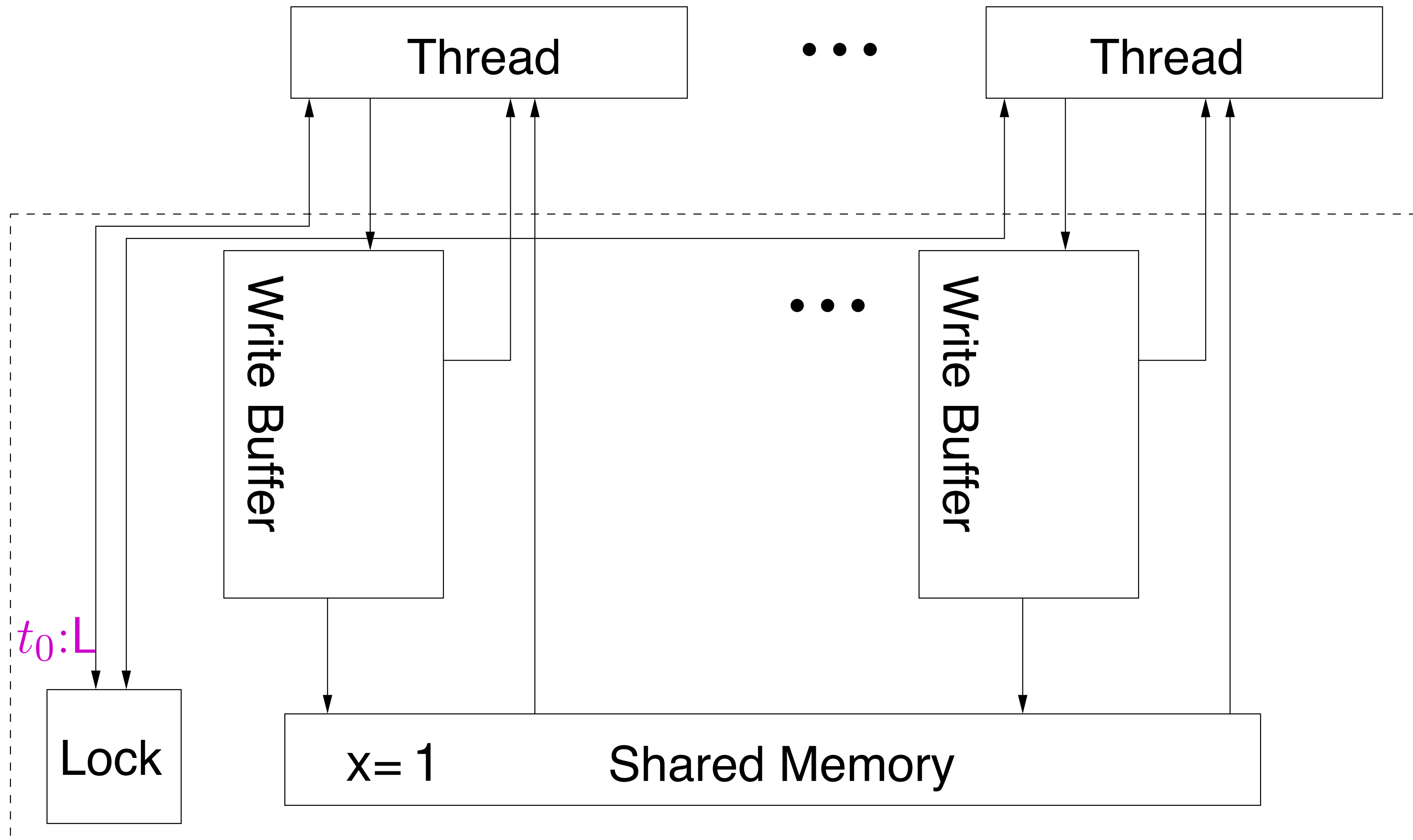
Locked INC

Thread 0	Thread 1
MOV [x]←1 (write x=1) LOCK; INC x	 LOCK; INC x



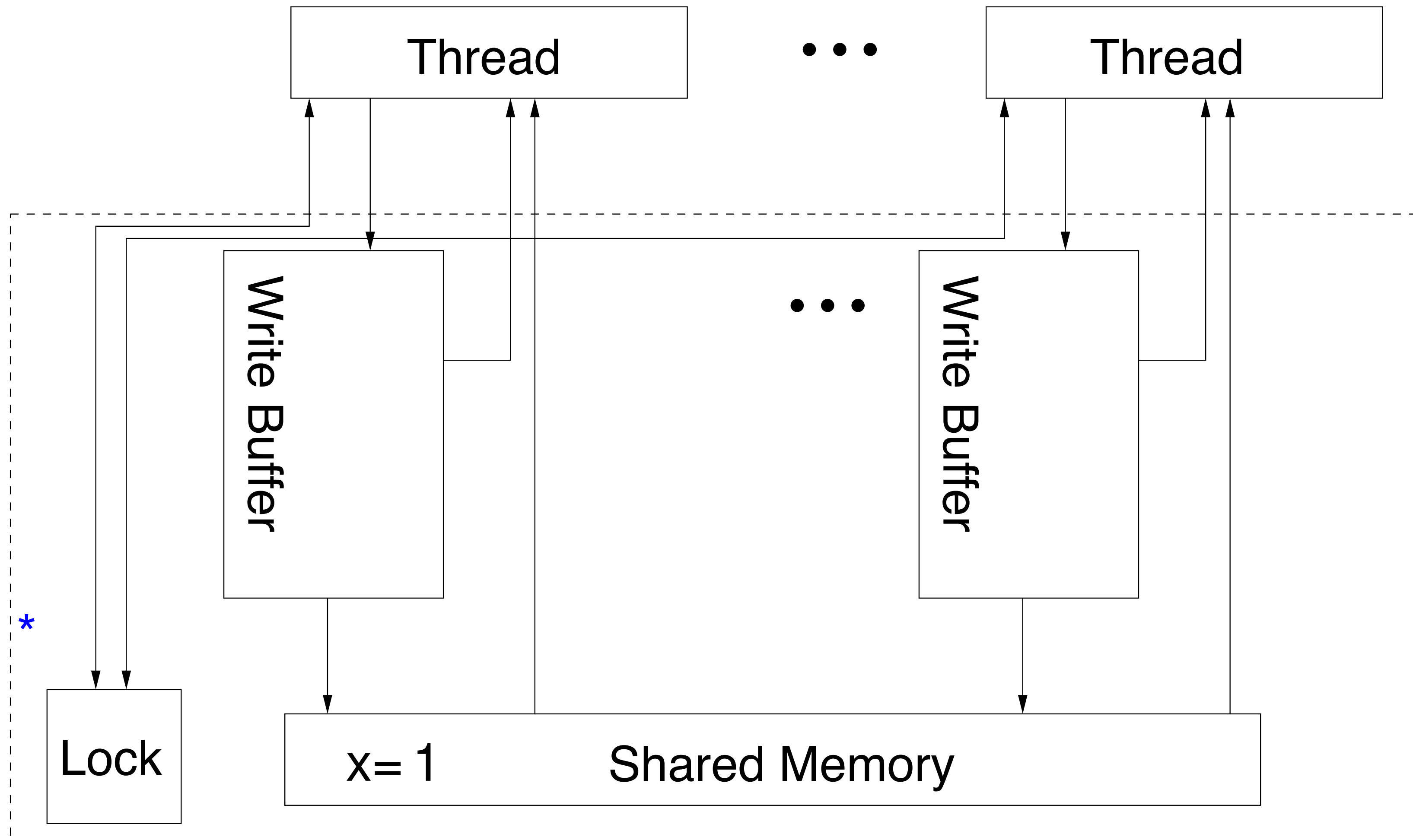
Locked INC

Thread 0	Thread 1
MOV [x]$\leftarrow$1 (write x=1)	
LOCK; INC x	LOCK; INC x



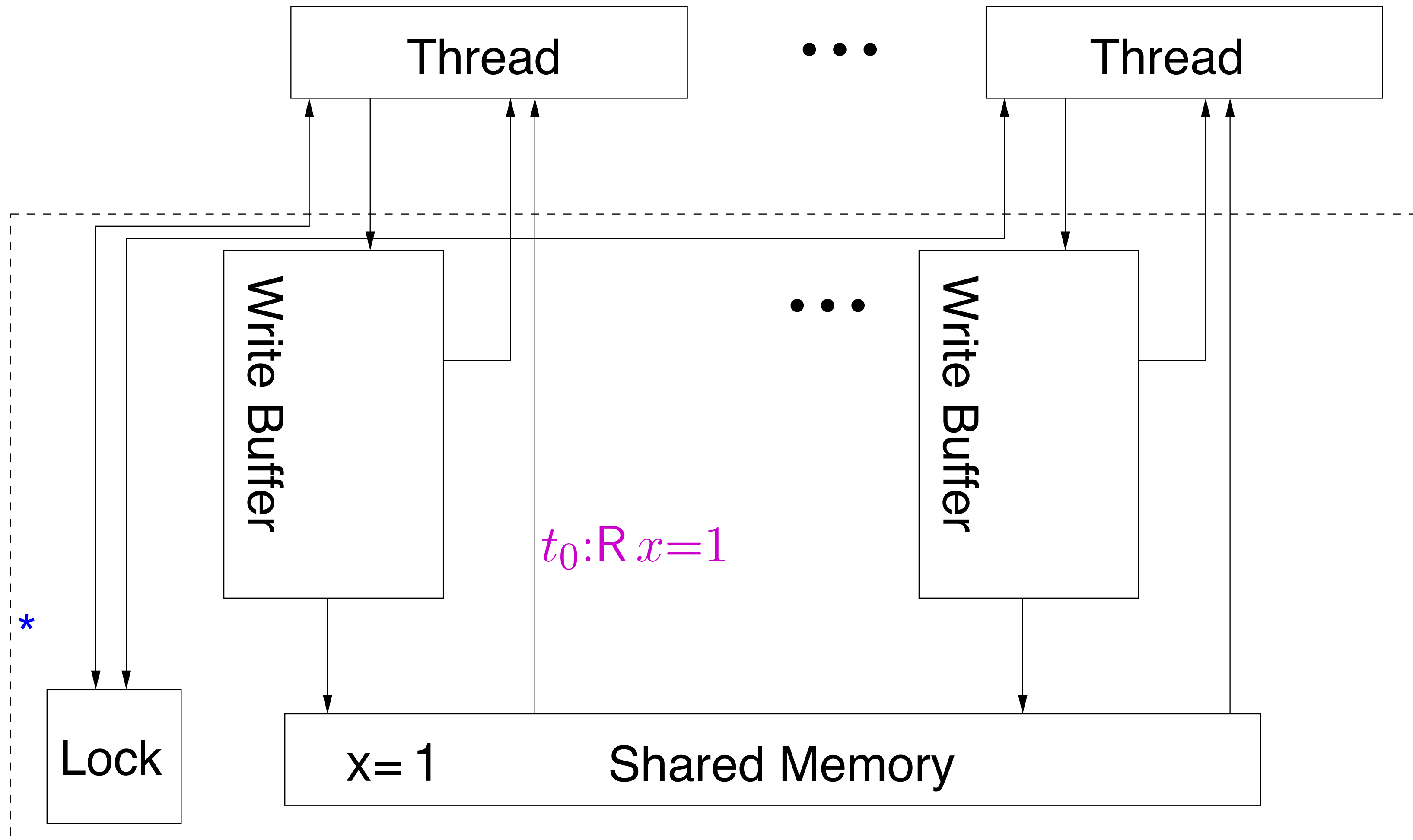
Locked INC

Thread 0	Thread 1
MOV [x]←1 (write x=1)	
LOCK; INC x	LOCK; INC x



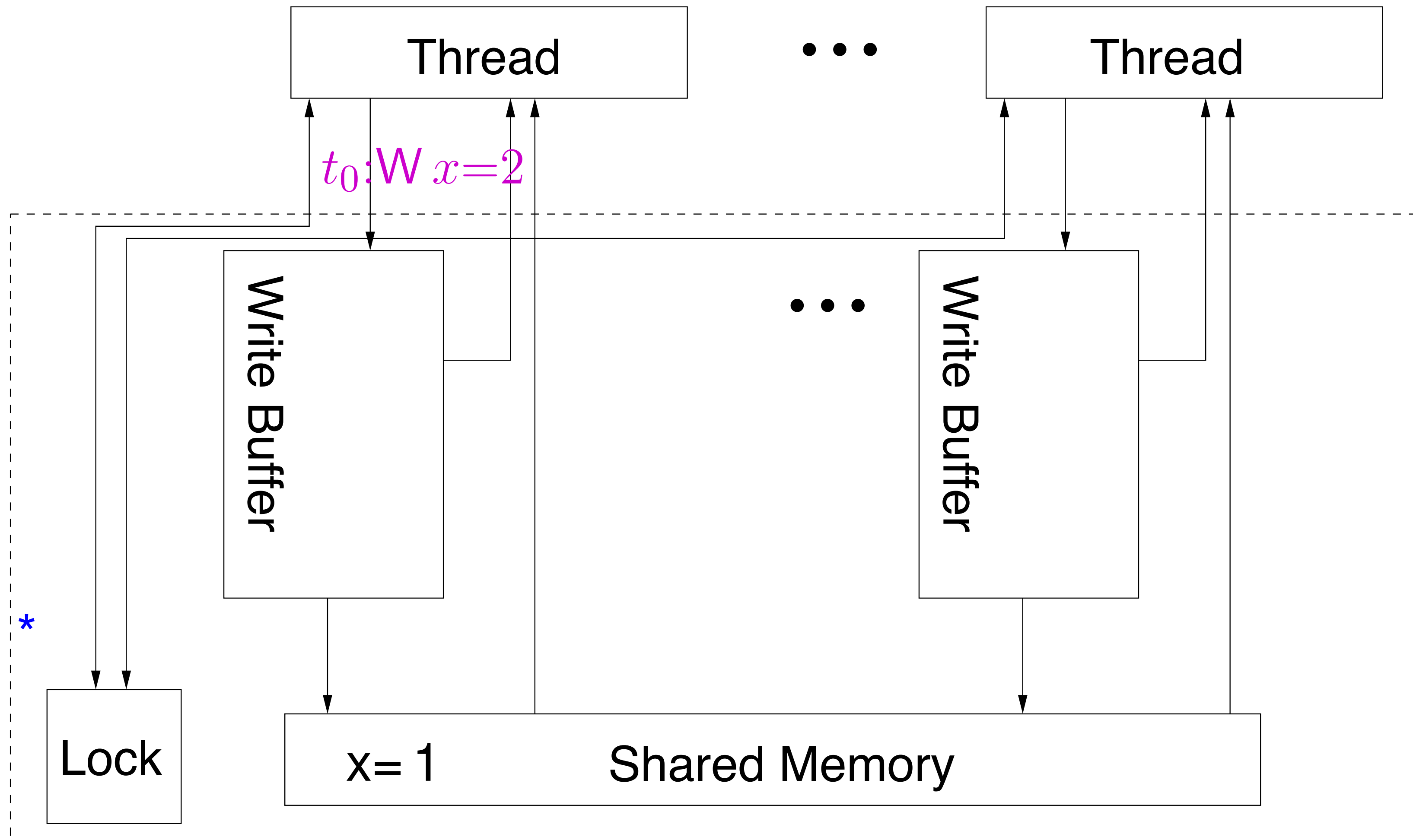
Locked INC

Thread 0	Thread 1
MOV [x]←1 (write x=1) LOCK; INC x	LOCK; INC x



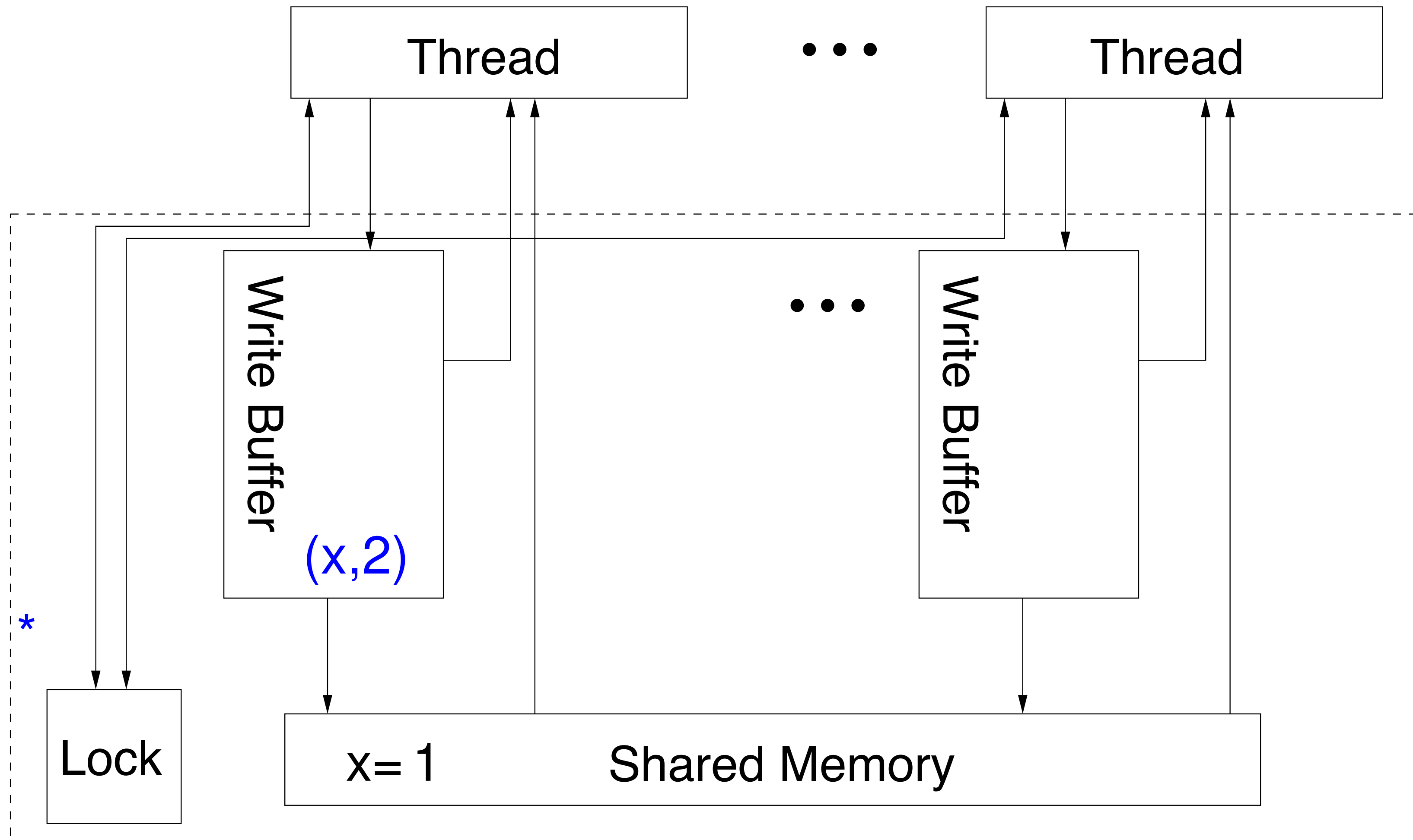
Locked INC

Thread 0	Thread 1
MOV [x]←1 (write x=1) LOCK; INC x	 LOCK; INC x



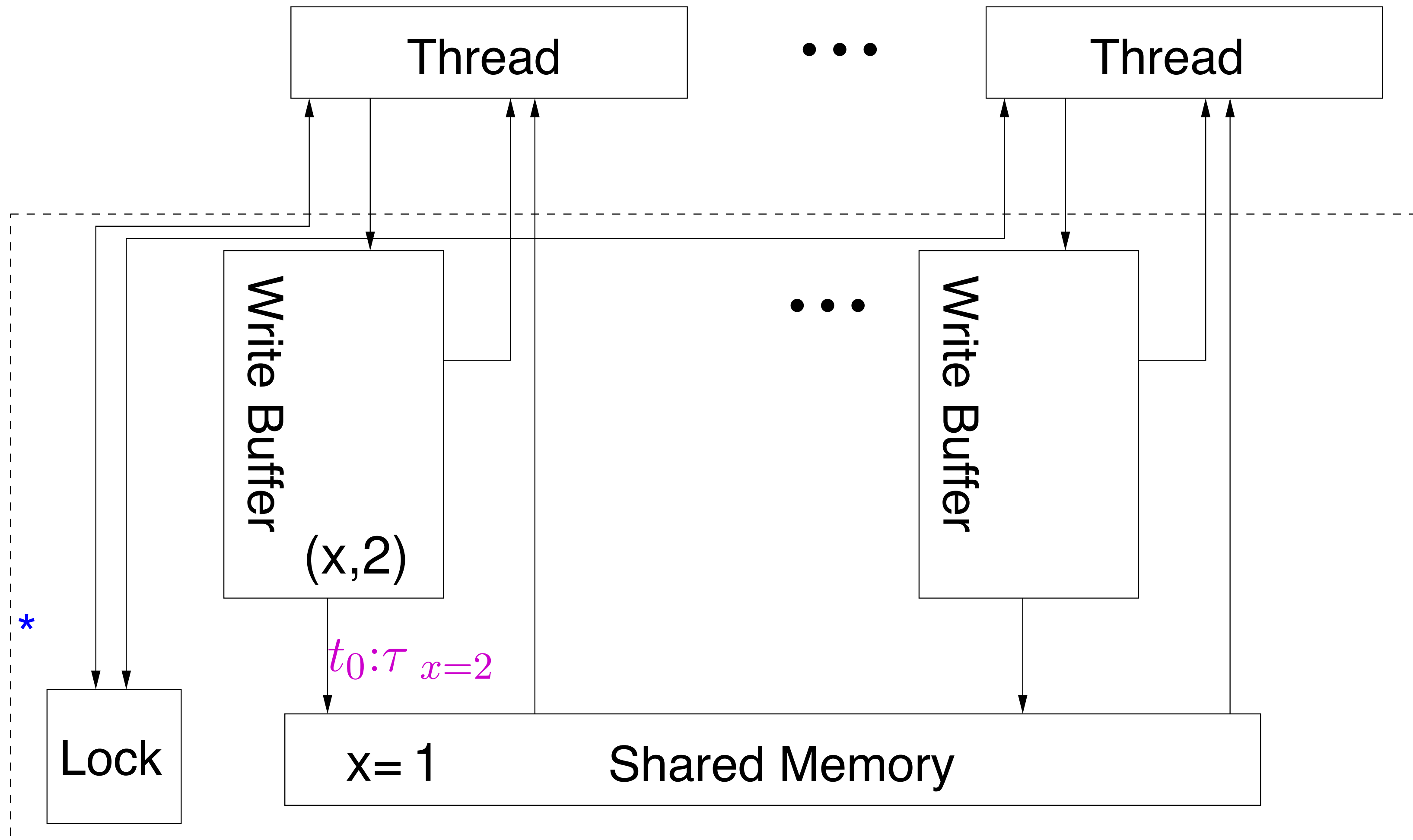
Locked INC

Thread 0	Thread 1
MOV [x]←1 (write x=1) LOCK; INC x	 LOCK; INC x



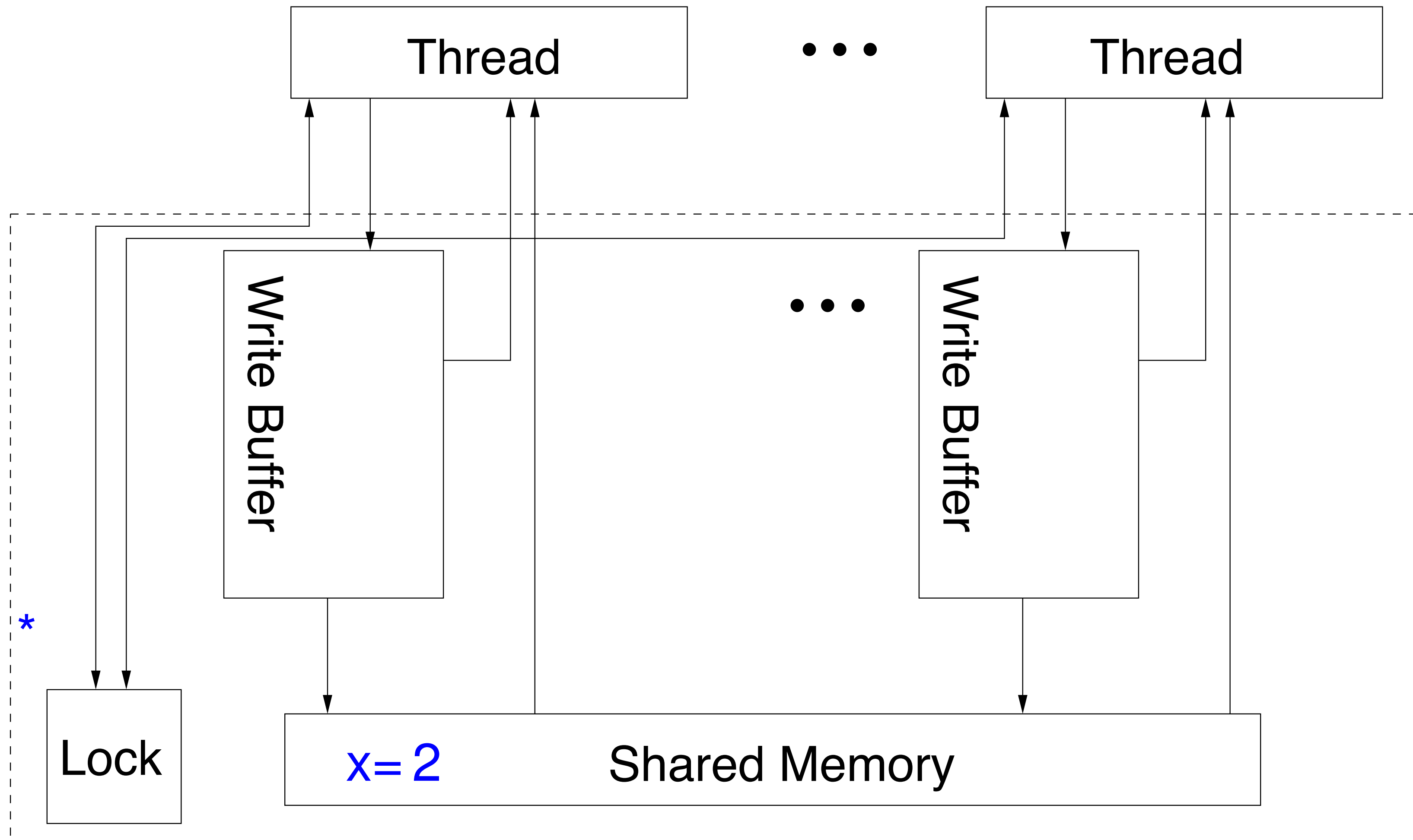
Locked INC

Thread 0	Thread 1
MOV [x]←1 (write x=1)	
LOCK; INC x	LOCK; INC x



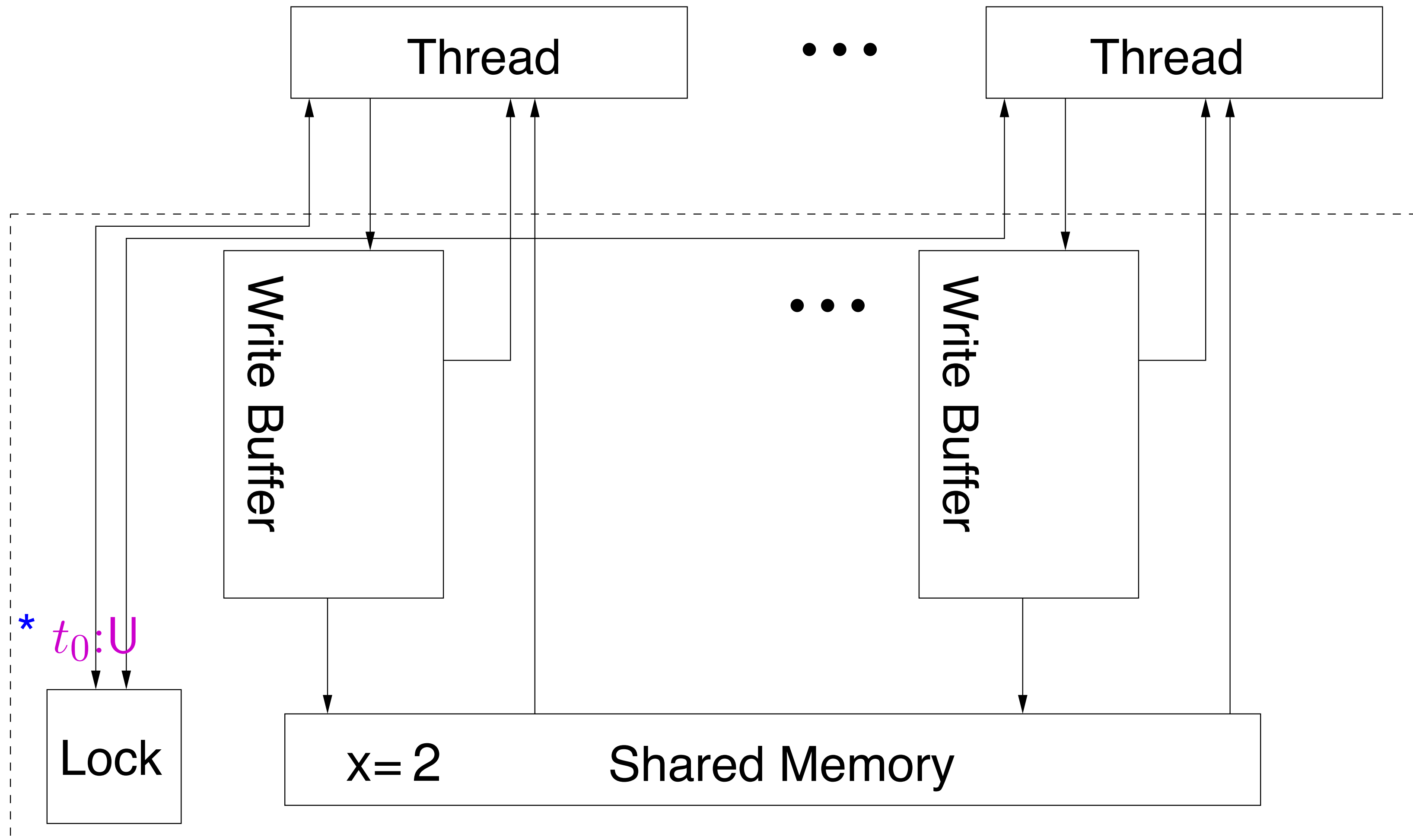
Locked INC

Thread 0	Thread 1
MOV [x]$\leftarrow$1 (write x=1)	
LOCK; INC x	LOCK; INC x



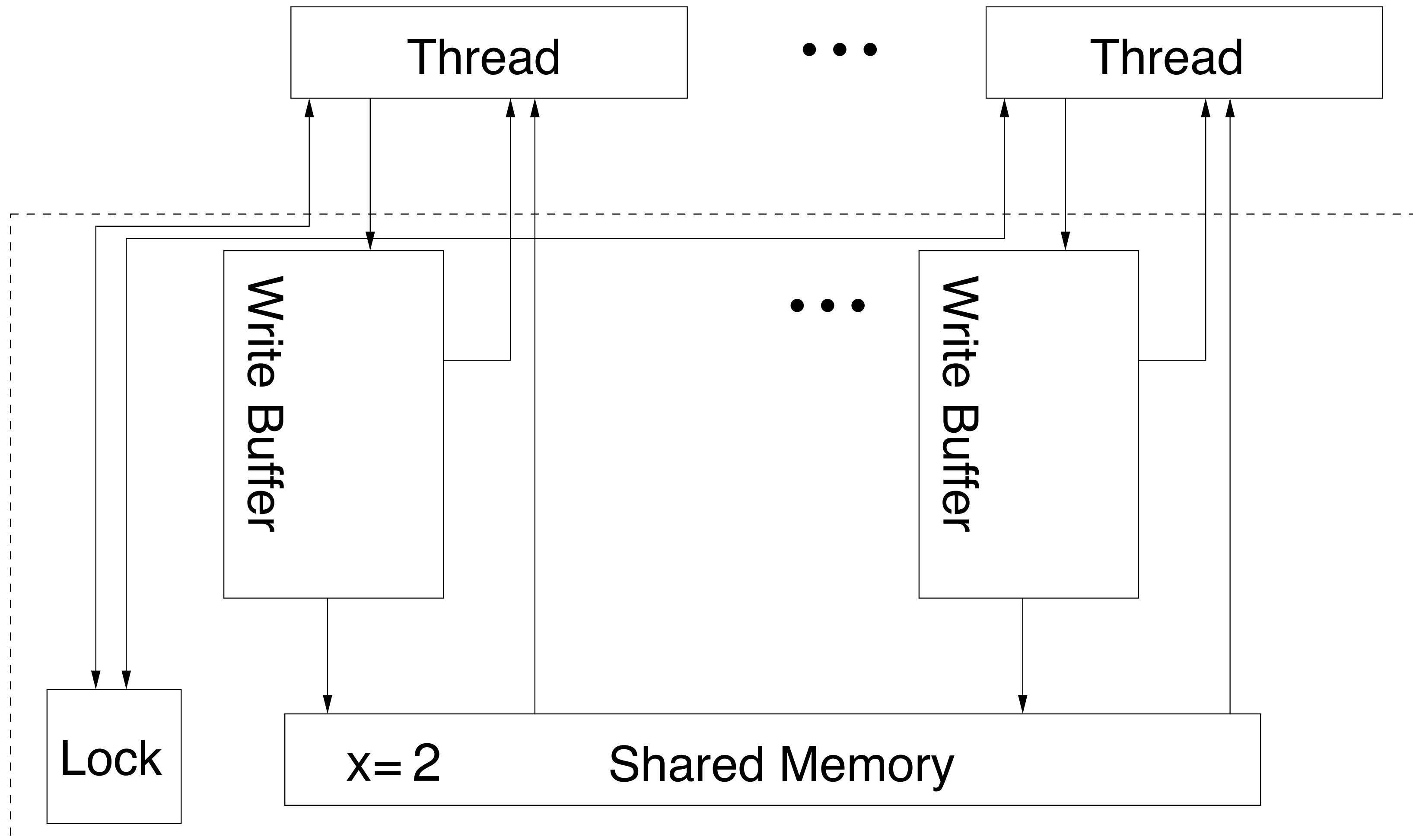
Locked INC

Thread 0	Thread 1
MOV [x]←1 (write x=1)	
LOCK; INC x	LOCK; INC x



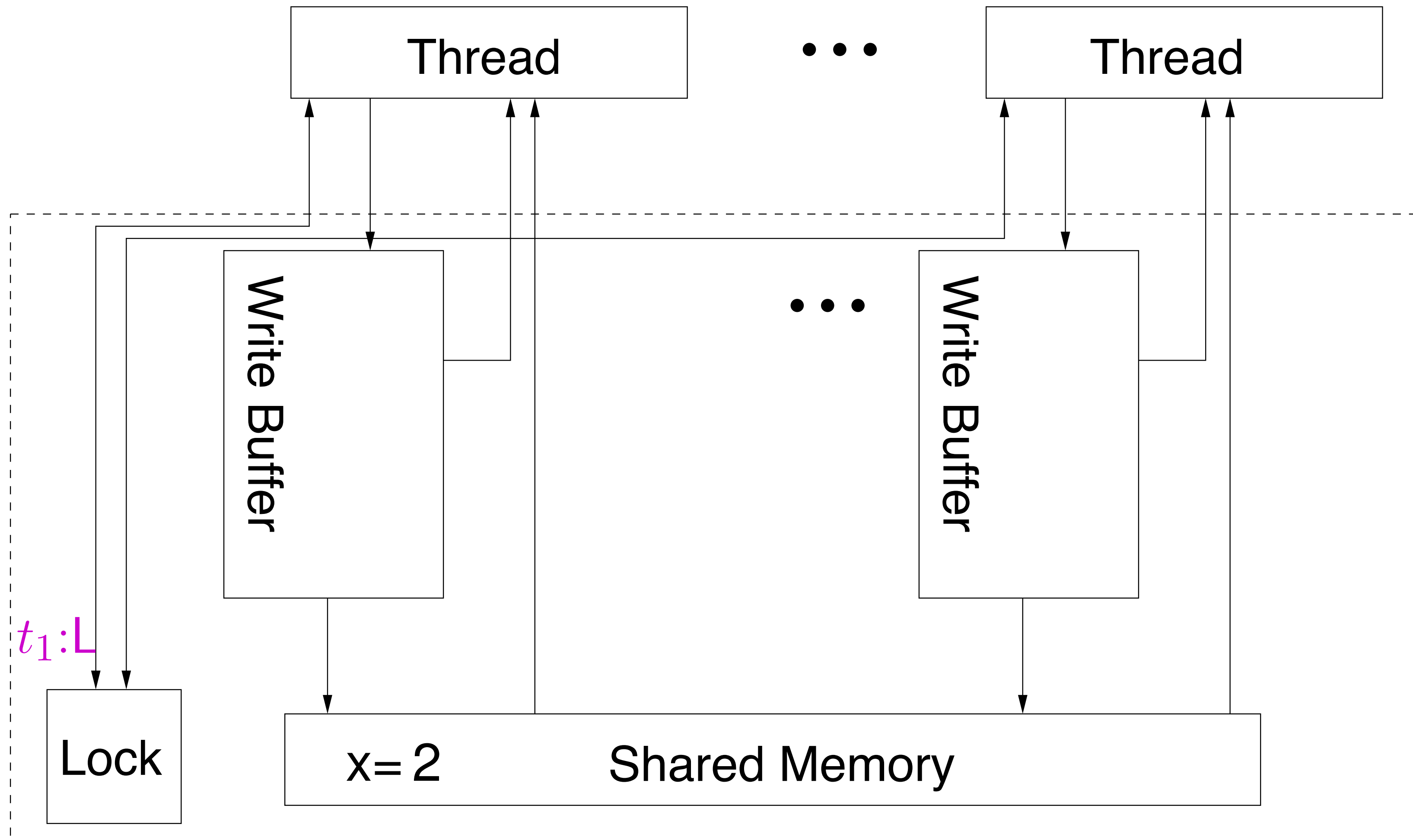
Locked INC

Thread 0	Thread 1
MOV [x]←1 (write x=1)	
LOCK; INC x	LOCK; INC x



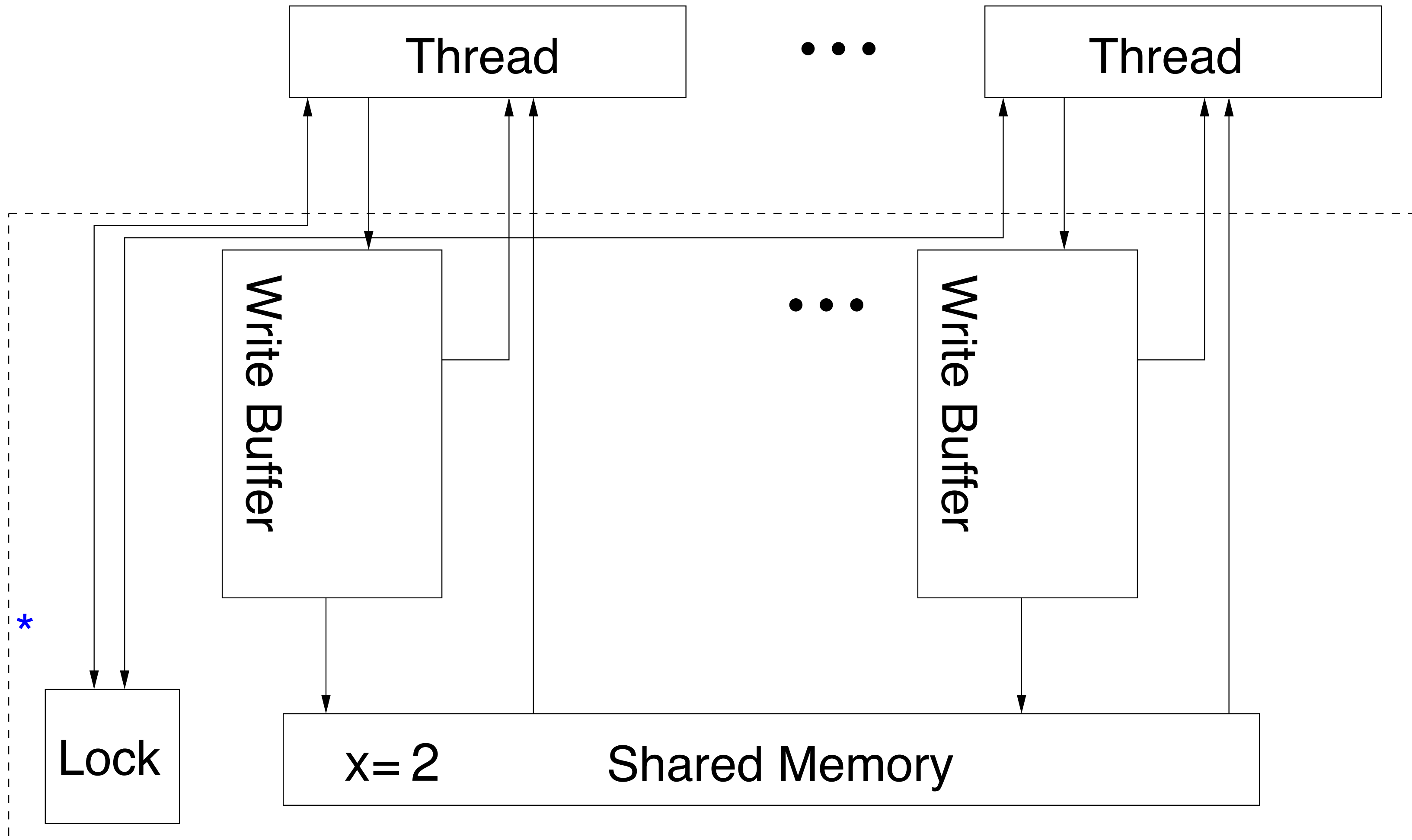
Locked INC

Thread 0	Thread 1
MOV [x]←1 (write x=1) LOCK; INC x	 LOCK; INC x



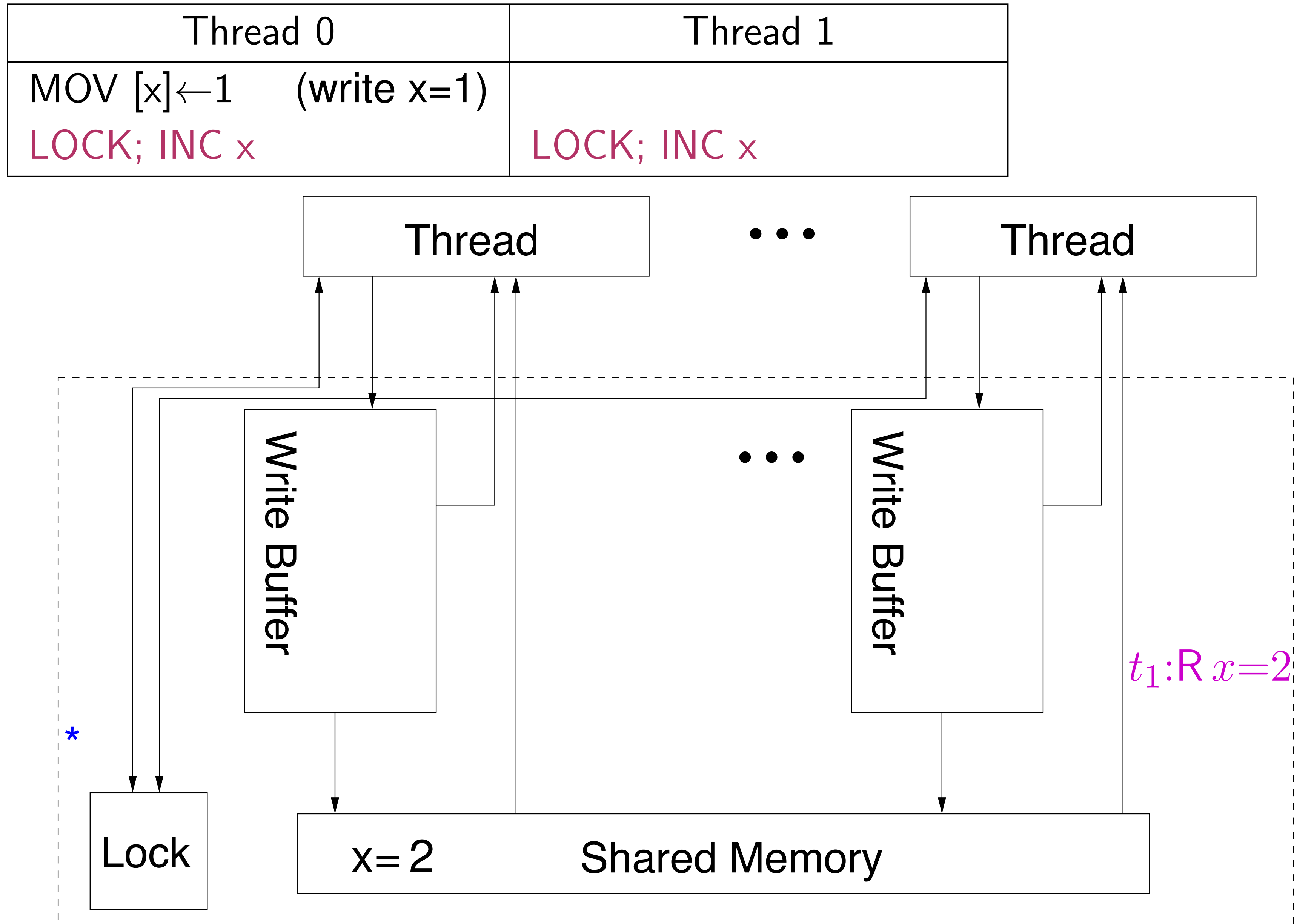
Locked INC

Thread 0	Thread 1
MOV [x]←1 (write x=1) LOCK; INC x	LOCK; INC x



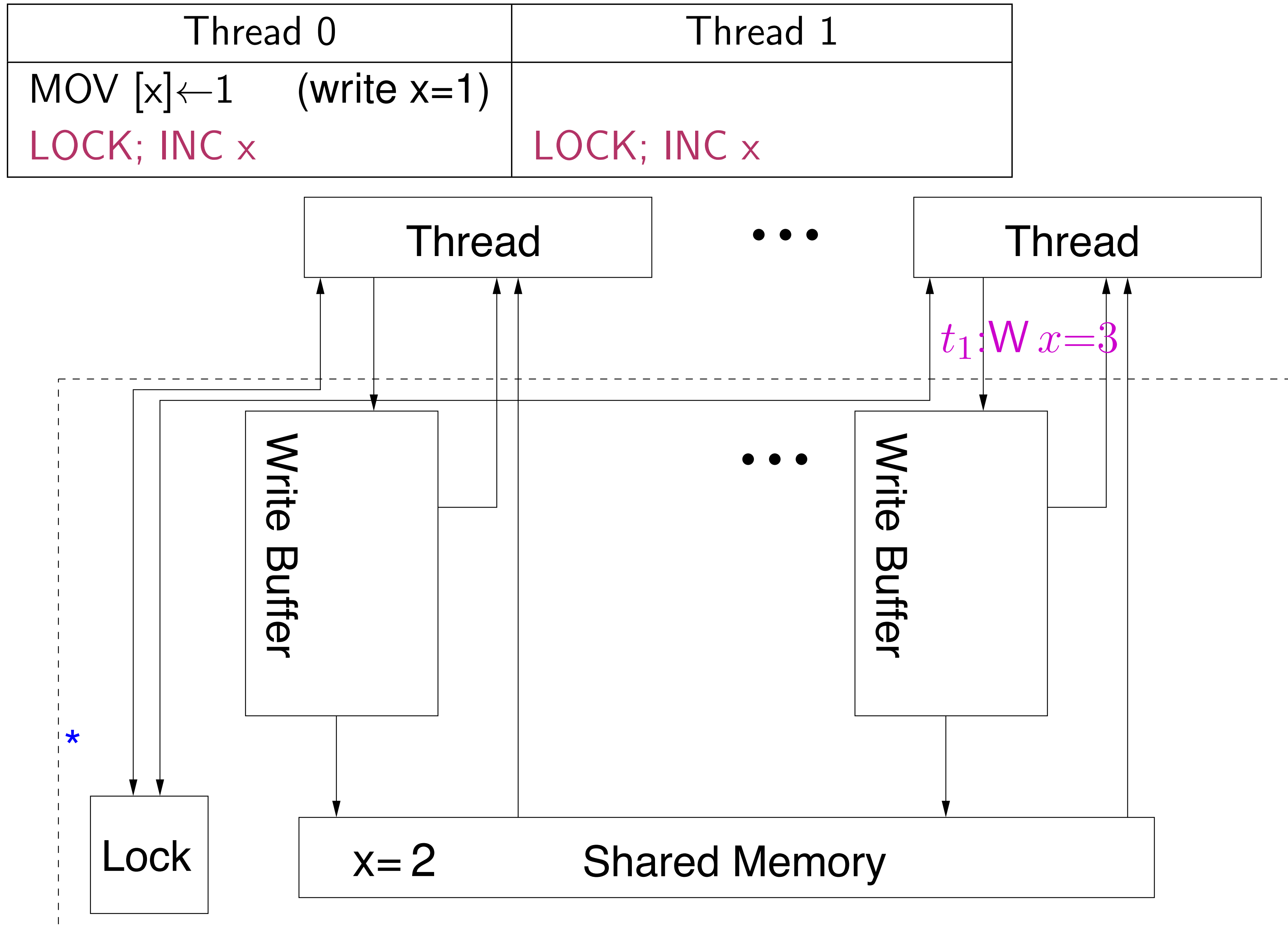
Locked INC

Thread 0	Thread 1
MOV [x]←1 (write x=1)	
LOCK; INC x	LOCK; INC x



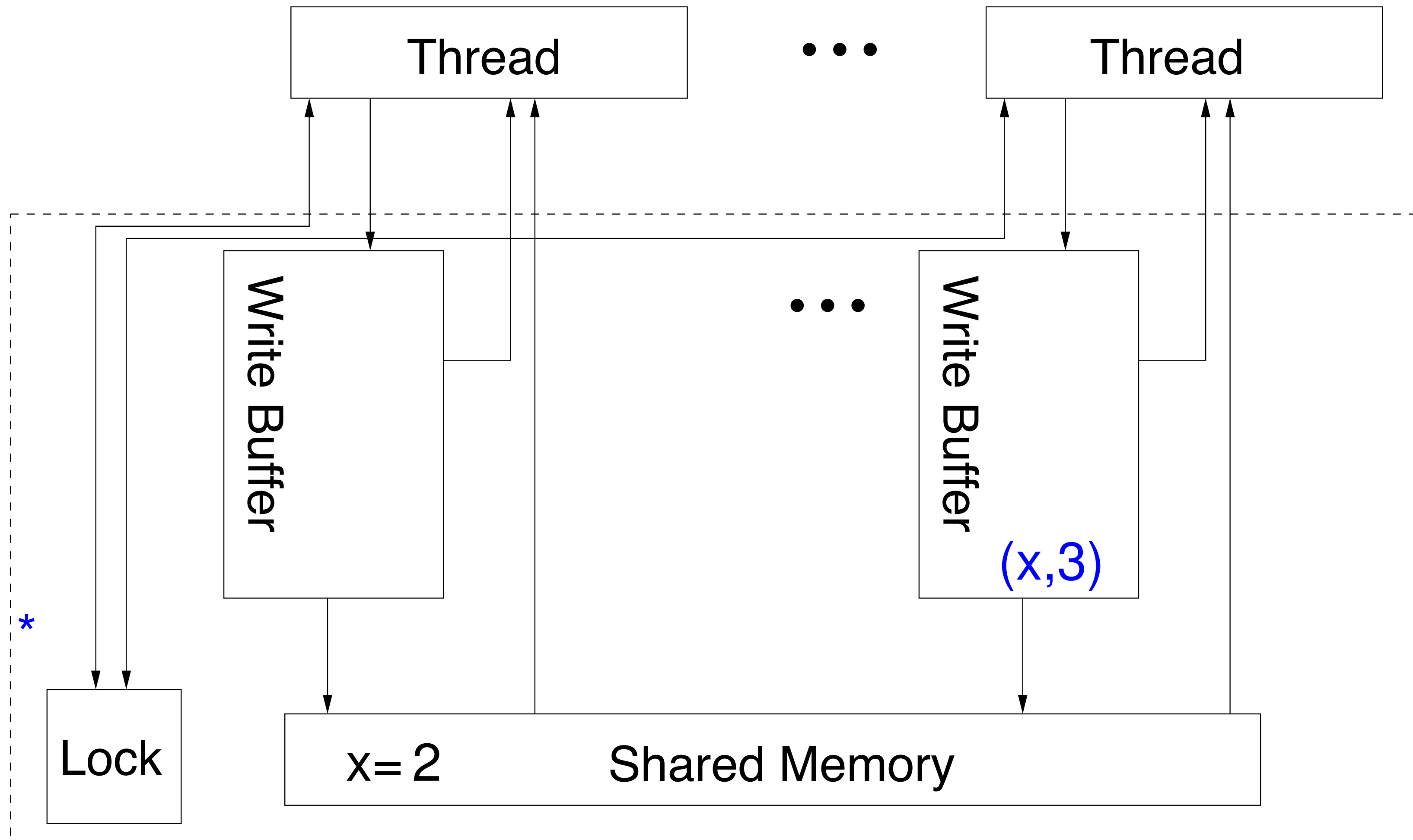
Locked INC

Thread 0	Thread 1
MOV [x]←1 (write x=1)	
LOCK; INC x	LOCK; INC x



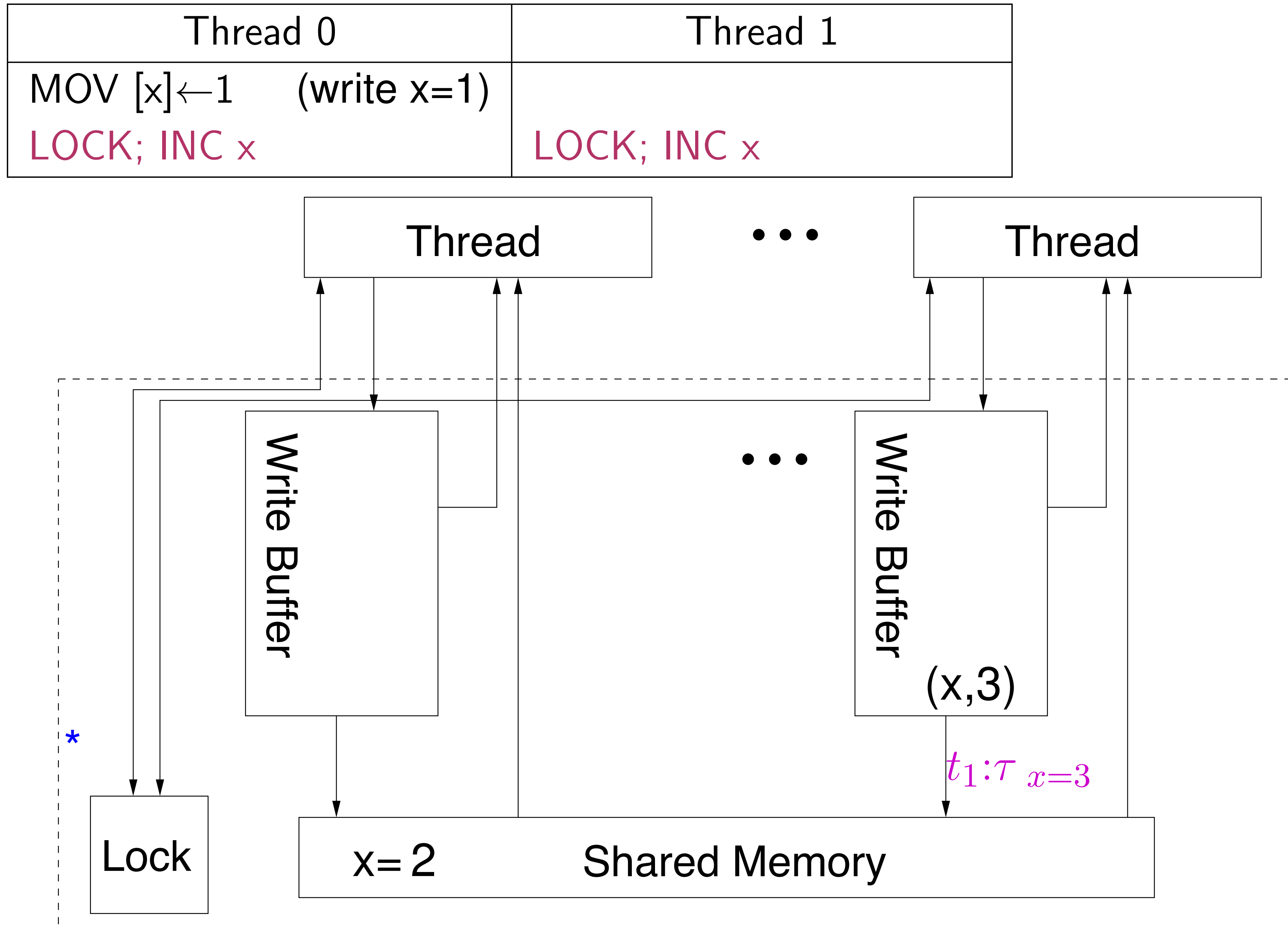
Locked INC

Thread 0	Thread 1
MOV [x]←1 (write x=1) LOCK; INC x	 LOCK; INC x



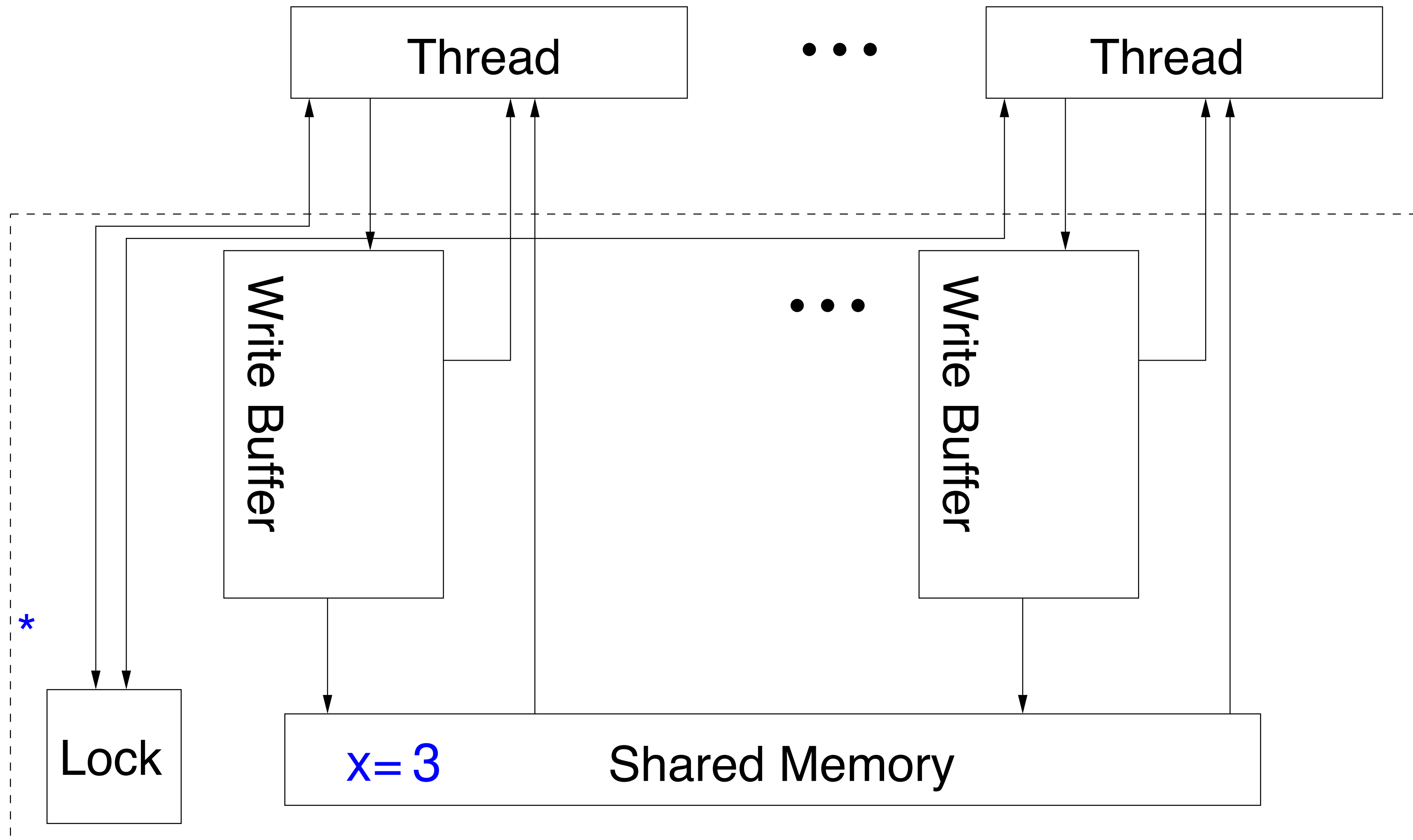
Locked INC

Thread 0	Thread 1
MOV [x]←1 (write x=1)	
LOCK; INC x	LOCK; INC x



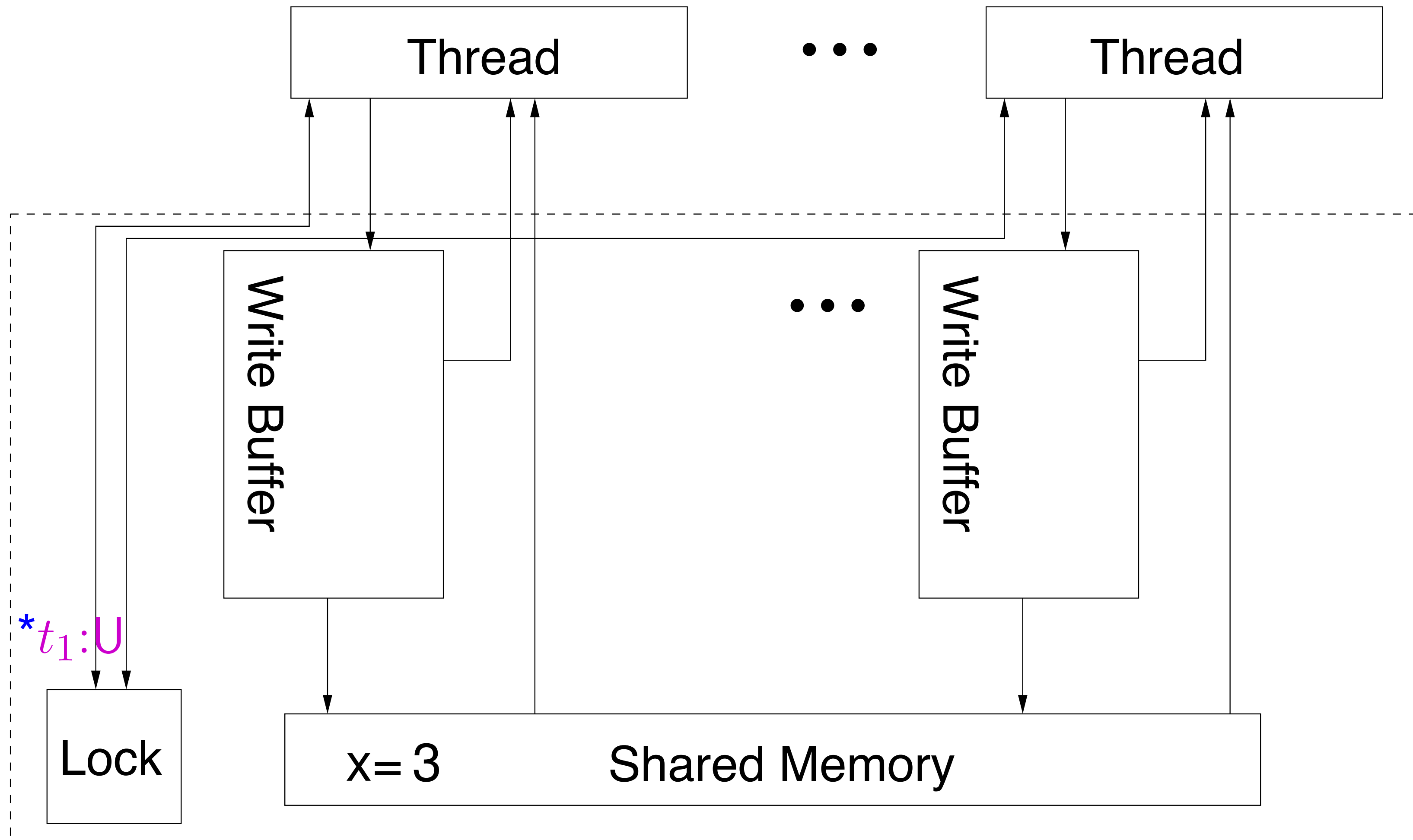
Locked INC

Thread 0	Thread 1
MOV [x]$\leftarrow$1 (write x=1)	
LOCK; INC x	LOCK; INC x



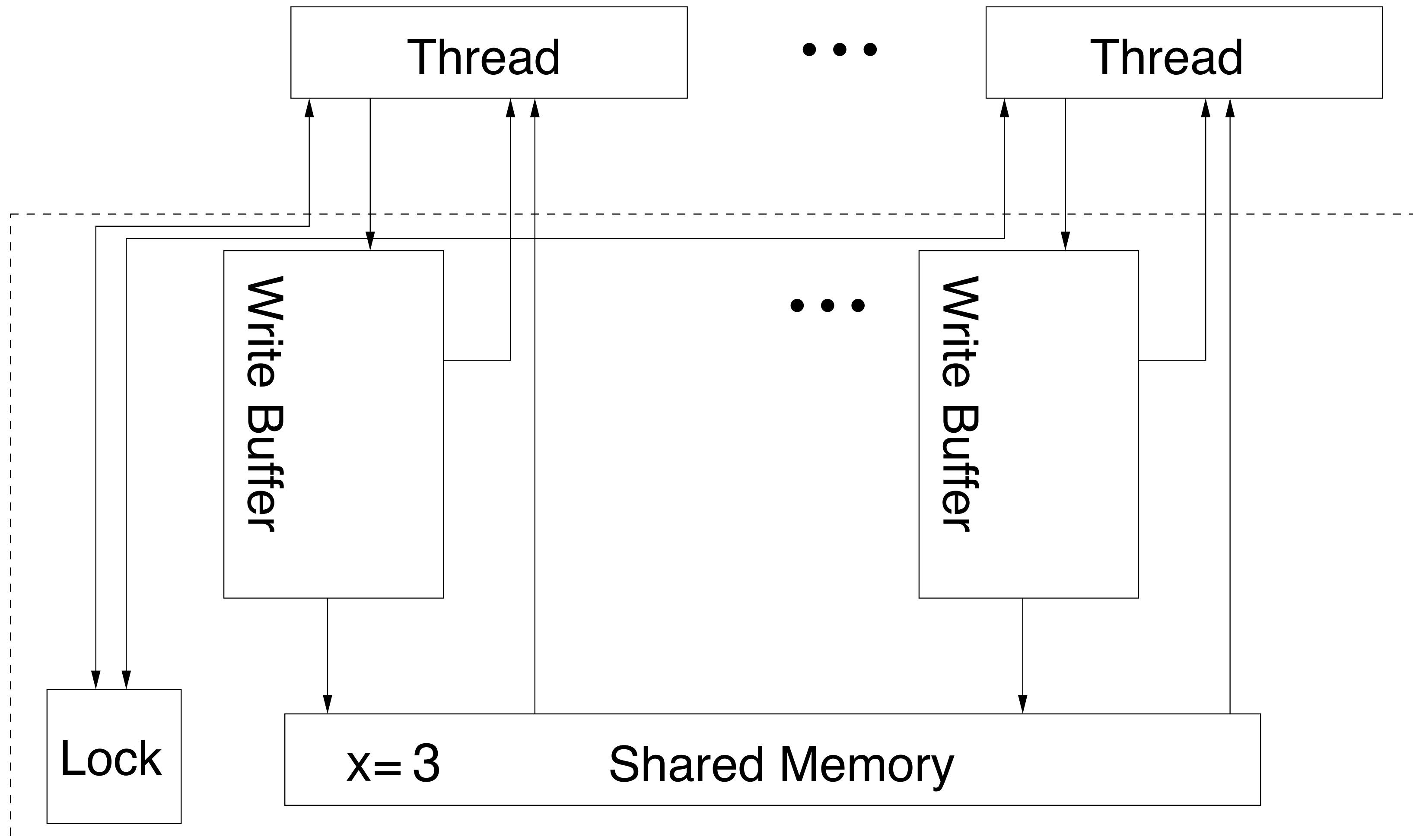
Locked INC

Thread 0	Thread 1
MOV [x]←1 (write x=1) LOCK; INC x	 LOCK; INC x



Locked INC

Thread 0	Thread 1
MOV [x]←1 (write x=1)	
LOCK; INC x	LOCK; INC x



Implementing Mutexes with x86 Spinlocks

Suppose register eax holds the address x, which holds 1 if the lock is free or ≤ 0 if taken.

```
lock:     LOCK DEC [eax]
          JNS enter
spin:     CMP [eax],0
          JLE spin
          JMP lock
enter:

          critical section

unlock:   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

Spinlock Example (SC)

while atomic_decrement(x) $<$ 0 {
  while x $\leq$ 0 { **skip** } }
critical section
x $\leftarrow$1

Shared Memory	Thread 0	Thread 1
x = 1		

Spinlock Example (SC)

while atomic_decrement(x) $<$ 0 {
    while x $\leq$ 0 { **skip** } }
critical section
x $\leftarrow$1

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	lock	

Spinlock Example (SC)

while atomic_decrement(x) $<$ 0 {
  while x $\leq$ 0 { **skip** } }
critical section
x $\leftarrow$1

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	lock	
x = 0	critical	

Spinlock Example (SC)

while atomic_decrement(x) $<$ 0 {
  while x $\leq$ 0 { **skip** } }
critical section
x $\leftarrow$1

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

Spinlock Example (SC)

while atomic_decrement(x) $<$ 0 {
  while x $\leq$ 0 { **skip** } }
critical section
x $\leftarrow$1

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

Spinlock Example (SC)

while atomic_decrement(x) $<$ 0 {
  while x $\leq$ 0 { **skip** } }
critical section
x $\leftarrow$1

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	lock	
x = 0	critical	
x = -1	critical	lock
x = -1	critical	spin, reading x
x = 1	unlock, 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	lock	
x = 0	critical	
x = -1	critical	lock
x = -1	critical	spin, reading x
x = 1	unlock, 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	lock	
x = 0	critical	
x = -1	critical	lock
x = -1	critical	spin, reading x
x = 1	unlock, writing x	
x = 1		read x
x = 0		lock

Spinlock Example (x86-TSO)

while atomic_decrement(x) $<$ 0 {
  while x $\leq$ 0 { **skip** } }
critical section
x $\leftarrow$1

Shared Memory	Thread 0	Thread 1
x = 1		

Spinlock Example (x86-TSO)

while atomic_decrement(x) $<$ 0 {
  while x $\leq$ 0 { **skip** } }
critical section
x $\leftarrow$1

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	lock	

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	lock	
x = -1	critical	lock

Spinlock Example (x86-TSO)

while atomic_decrement(x) $<$ 0 {
  while x $\leq$ 0 { **skip** } }
critical section
x $\leftarrow$1

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	lock	
x = -1	critical	lock
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	lock	
x = -1	critical	lock
x = -1	critical	spin, reading x
x = -1	unlock, writing x to buffer	

Spinlock Example (x86-TSO)

while atomic_decrement(x) $<$ 0 {
  while x $\leq$ 0 { **skip** } }
critical section
x $\leftarrow$1

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

Spinlock Example (x86-TSO)

while atomic_decrement(x) $<$ 0 {
  while x $\leq$ 0 { **skip** } }
critical section
x $\leftarrow$1

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

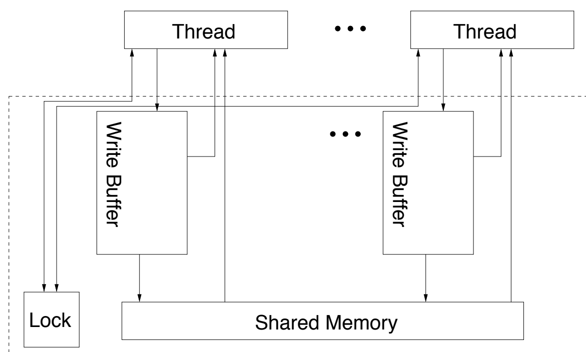
Spinlock Example (x86-TSO)

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

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

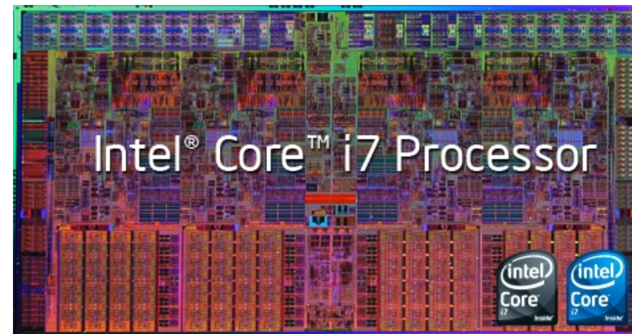
NB: This is an *Abstract* Machine

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.

That's x86

Next slot, 3:30PM: TD session, x86

Tomorrow, 9:00AM: The more relaxed Power and ARM architectures.