

# Parallelism: Memory Consistency Models

**Brian Railing, Todd C. Mowry**

## Part 2 of Memory Correctness: Memory Consistency Model

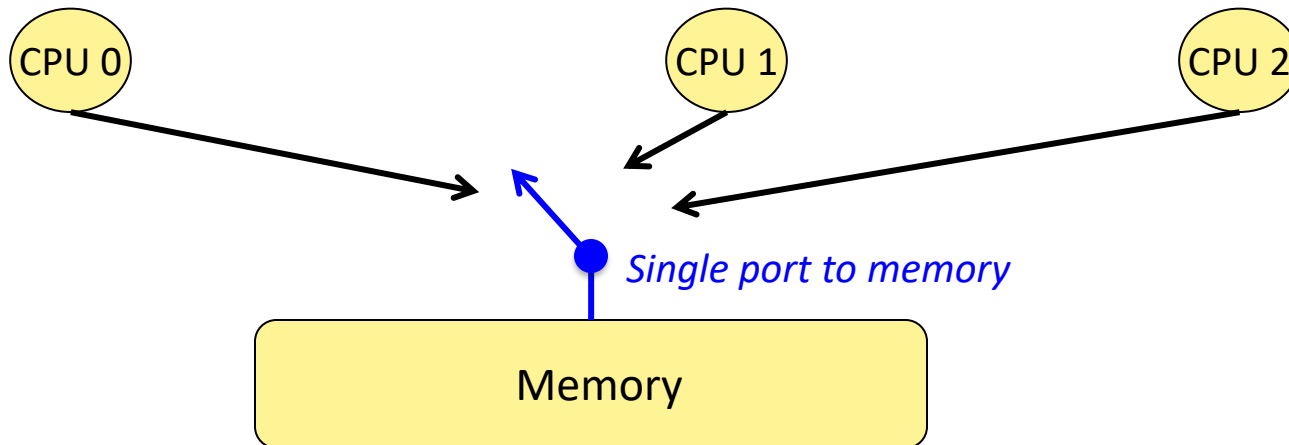
### 1. “Cache Coherence”

- do all loads and stores to a **given cache block** behave correctly?

### 2. “Memory Consistency Model” (sometimes called “Memory Ordering”)

- do all loads and stores, even to **separate cache blocks**, behave correctly?

Recall: our **intuition**



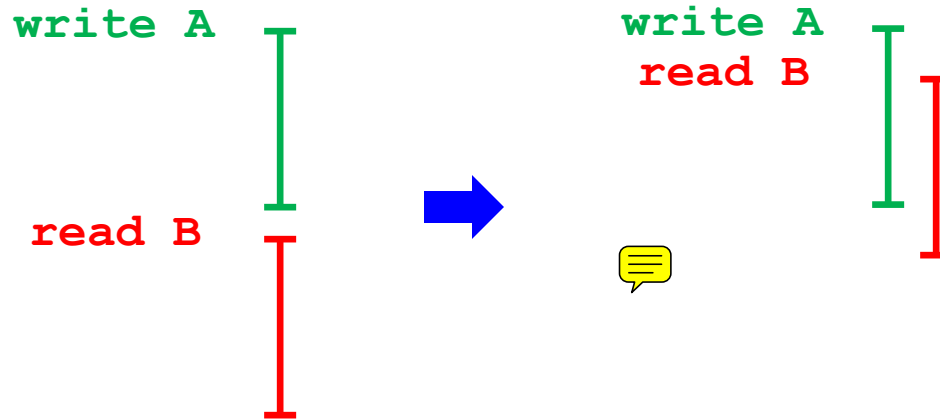
# Why is this so complicated?

- Fundamental issue:
  - loads and stores are very expensive, even on a uniprocessor
    - can easily take 10's to 100's of cycles
- What programmers intuitively expect:
  - processor atomically performs one instruction at a time, in program order
- In reality:
  - if the processor actually operated this way, it would be painfully slow
  - instead, the processor aggressively reorders instructions to hide memory latency
- Upshot:
  - *within a given thread*, the processor preserves the program order illusion
  - but this illusion has nothing to do with what happens in physical time!
  - from the perspective of other threads, all bets are off!

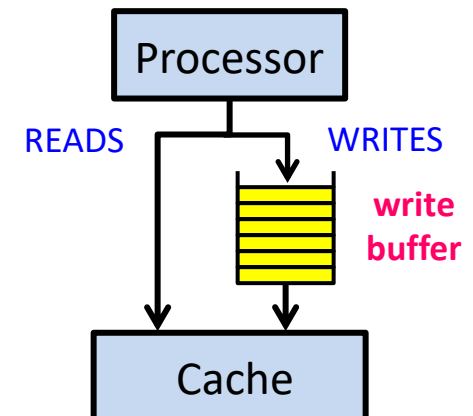


# Hiding Memory Latency is Important for Performance

- Idea: *overlap memory accesses* with *other accesses* and *computation*



- Hiding *write* latency is simple in uniprocessors:
  - add a *write buffer*
- (But this affects *correctness* in multiprocessors)



# How Can We Hide the Latency of Memory Reads?

## “Out of order” pipelining:

- when an instruction is stuck, perhaps there are subsequent instructions that can be executed

```
x = *p;
```

```
y = x + 1;
```

```
z = a + 2;
```

```
b = c / 3;
```

← suffers expensive cache miss

← stuck waiting on true dependence

} ← these do not need to wait

- Implication: memory accesses may be performed out-of-order!!!

## What About Conditional Branches?

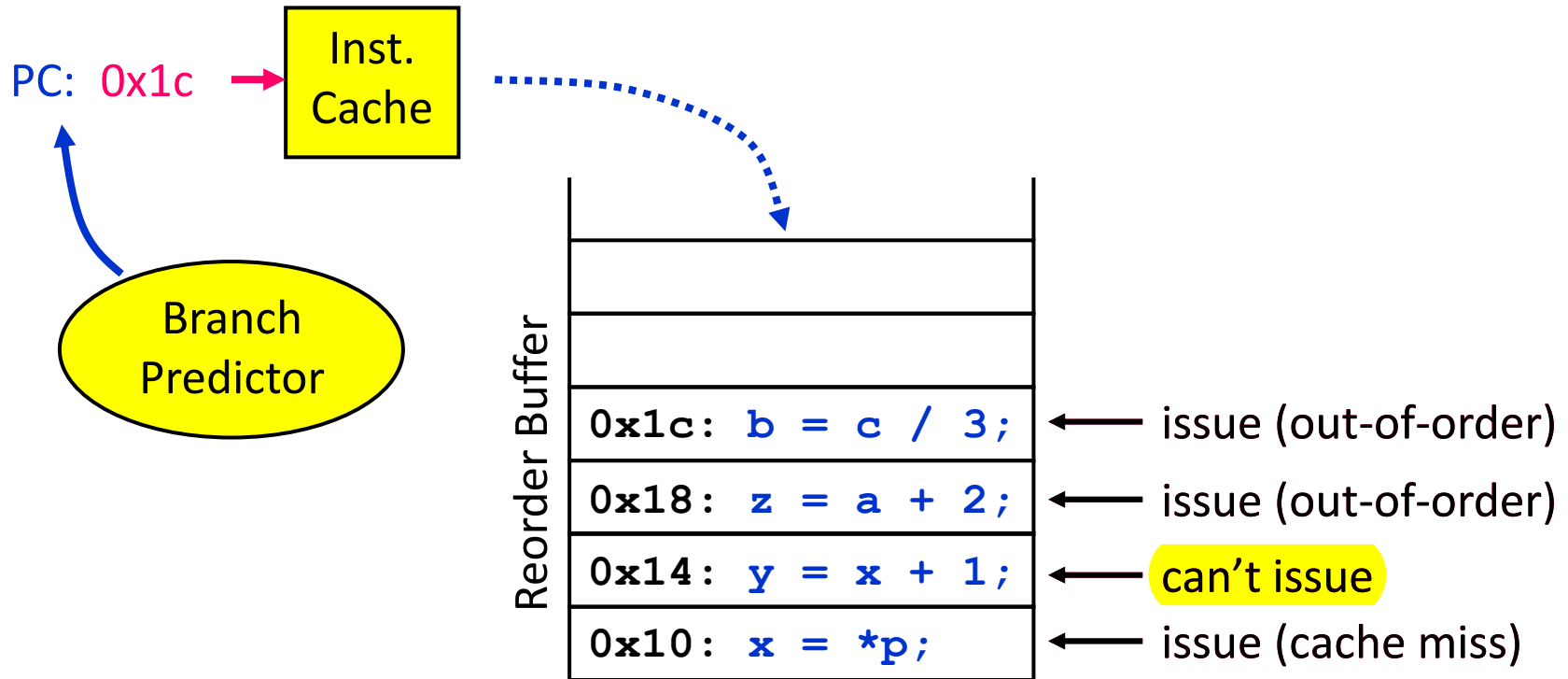
- Do we need to wait for a conditional branch to be resolved before proceeding?
  - No! Just predict the branch outcome and continue executing speculatively.
    - if prediction is wrong, squash any side-effects and restart down correct path

```
x = *p;  
y = x + 1;  
z = a + 2;  
b = c / 3;  
if (x != z)  
    d = e - 7;  
else d = e + 5;  
...
```

if hardware guesses that this is true  
then execute “then” part (speculatively)  
(without waiting for **x** or **z**)

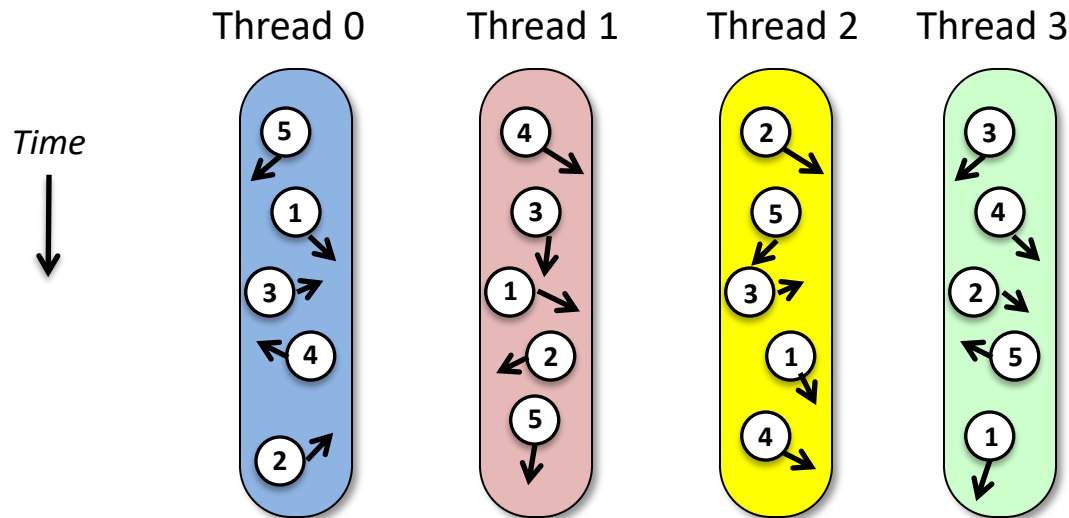
# How Out-of-Order Pipelining Works in Modern Processors

- Fetch and decode instructions in-order, but **issue out-of-order**



- Intra-thread **dependences are preserved**, but **memory accesses get reordered!**

## Analogy: Gas Particles in Balloons



(wikiHow)



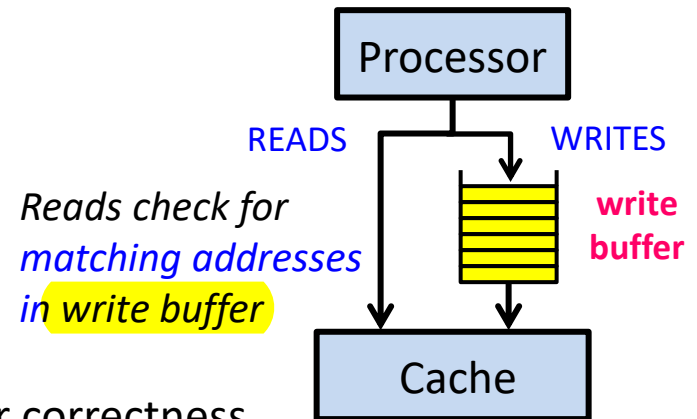
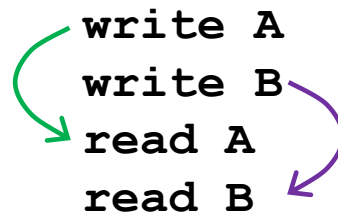
- Imagine that each instruction within a thread is a gas particle inside a twisty balloon
- They were numbered originally, but then they start to move and bounce around
- When a given thread observes memory accesses from a *different* thread:
  - those memory accesses can be (almost) **arbitrarily jumbled around**
    - like trying to locate the position of a particular gas particle in a balloon
- As we'll see later, the only thing that we can do is to put **twists** in the balloon



# Uniprocessor Memory Model

- **Memory model** specifies **ordering constraints** among accesses
- Uniprocessor model: memory accesses **atomic** and **in program order**

**write A**  
**write B**  
**read A**  
**read B**



- Not necessary to maintain sequential order for correctness
  - **hardware**: buffering, pipelining
  - **compiler**: register allocation, code motion
- **Simple** for programmers
- Allows for **high performance**

## In Parallel Machines (with a Shared Address Space)

- Order between **accesses to different locations** becomes important

*(Initially A and Ready = 0)*

P1

**A** = 1;

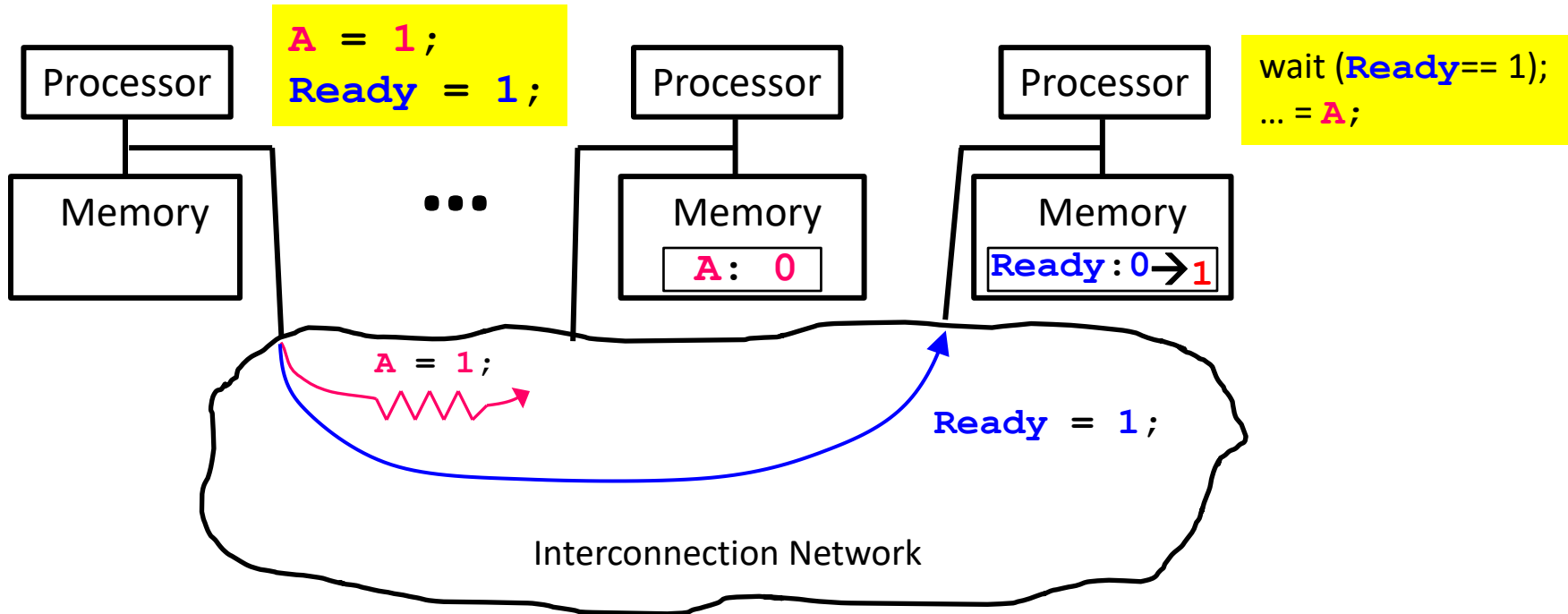
**Ready** = 1;

P2

**while** (**Ready** != 1) ;

... = **A**;

## How Unsafe Reordering Can Happen

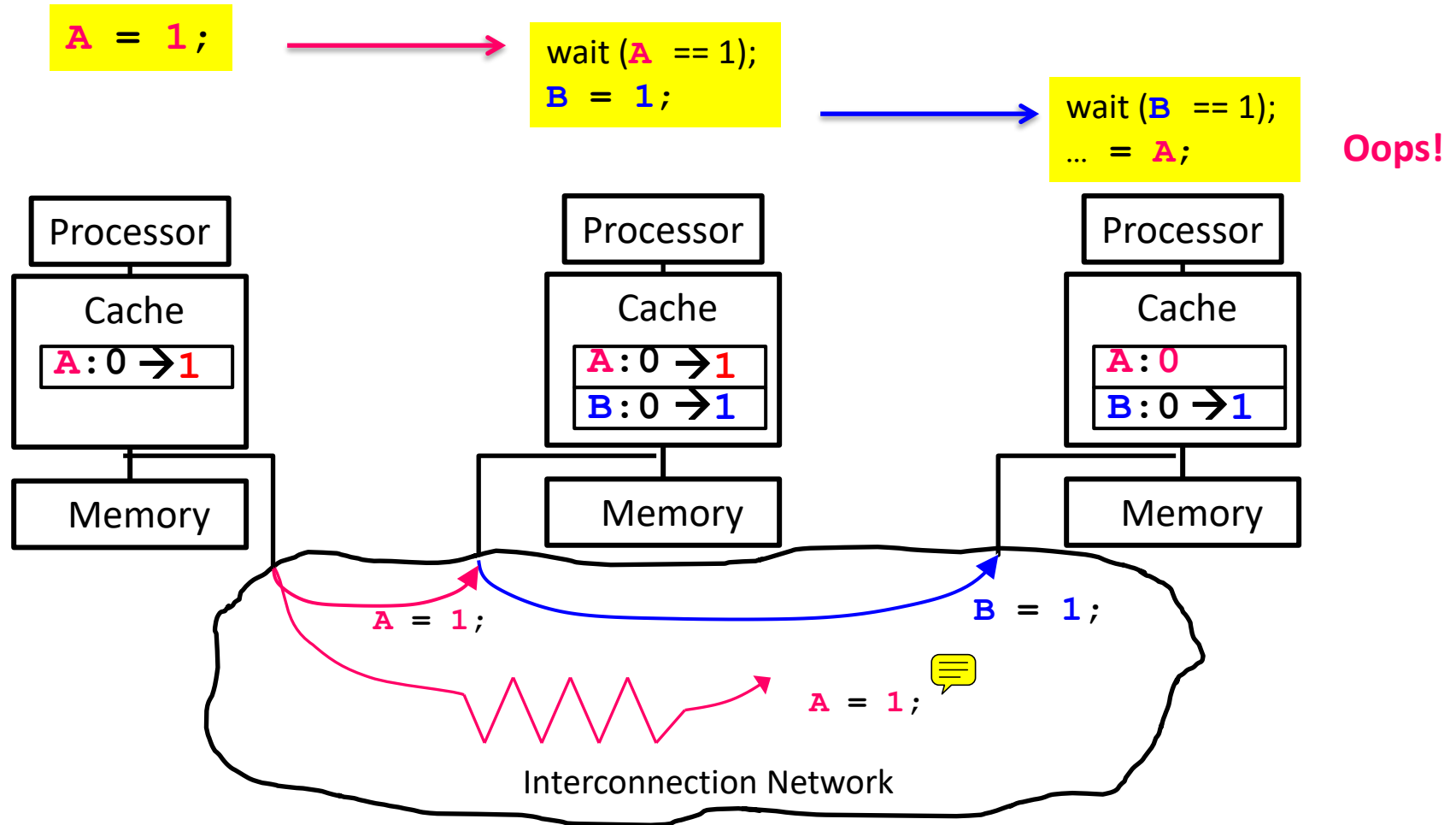


- Distribution of memory resources
  - accesses issued in order may be observed out of order



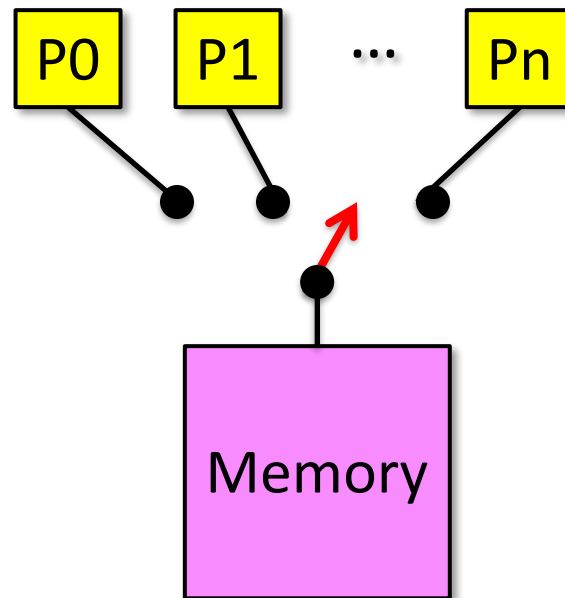
## Caches Complicate Things More

- Multiple copies of the same location



## Our Intuitive Model: “Sequential Consistency” (SC)

- Formalized by Lamport (1979)
  - accesses of each processor in **program order**
  - all accesses appear in **sequential order**



- Any order implicitly assumed by programmer is maintained

## Example with Sequential Consistency

### Simple Synchronization:

P0

**A** = 1 (a)

**Ready** = 1 (b)

P1

**x** = **Ready** (c)

**y** = **A** (d)

- all locations are initialized to 0
- possible outcomes for (x,y):
  - (0,0), (0,1), (1,1)
- (x,y) = (1,0) is **not a possible outcome** (i.e., **Ready** = 1, **A** = 0):
  - we know a->b and c->d by program order
  - b->c implies that a->d
  - y==0 implies d->a which leads to a contradiction

 *but real hardware will do this!*



## Another Example with Sequential Consistency

Stripped-down version of a 2-process mutex (minus the turn-taking):

P0


**want**[0] = 1 (a)

**x** = **want**[1] (b)

P1

**want**[1] = 1 (c)

**y** = **want**[0] (d)

- all locations are initialized to 0
- possible outcomes for (x,y):
  - (0,1), (1,0), (1,1)
- (x,y) = (0,0) is **not a possible outcome** (i.e., **want**[0] = 0, **want**[1] = 0):
  - a->b and c->d implied by program order
  - x = 0 implies b->c which implies a->d
  - a->d says y = 1 which leads to a contradiction
  - similarly, y = 0 implies x = 1 which is also a contradiction
  - *but real hardware will do this!* 

# One Approach to Implementing Sequential Consistency

1. Implement **cache coherence**
    - writes to the **same location** are observed in same order by all processors
  2. For each processor, **delay start of memory access until previous one completes**
    - each processor has only one outstanding memory access at a time
- What does it mean for a memory access to **complete**?



# When Do Memory Accesses Complete?

- Memory Reads:

- a read completes **when its return value is bound**

```
load r1 ← x
```

x = ???



x = 17 (Find x in memory system)



r1 = 17

# When Do Memory Accesses Complete?

- Memory Reads:
  - a read completes when its return value is bound
- Memory Writes:
  - a write completes when the new value is “visible” to other processors

store 23 → x

x = 23



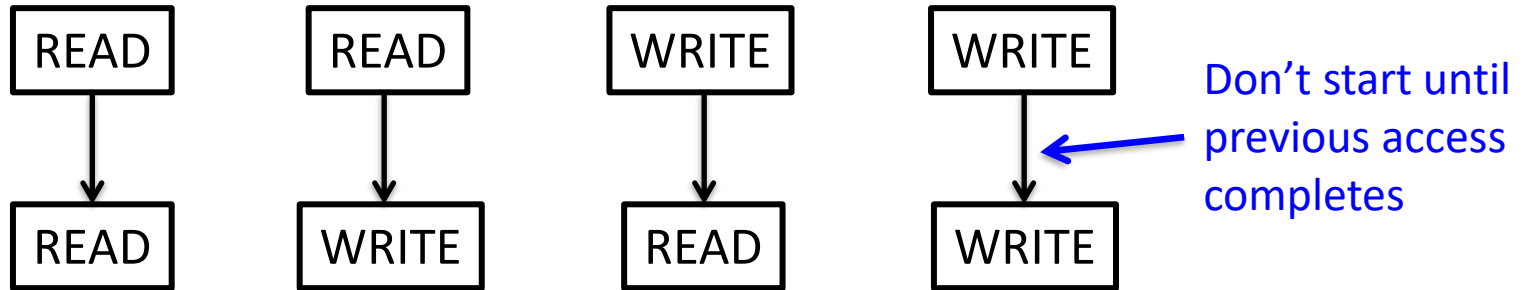
(Commit to memory order)  
(aka “serialize”)

- What does “visible” mean?
  - it does NOT mean that other processors have necessarily seen the value yet
  - it means the new value is committed to the hypothetical serializable order (HSO)
    - a later read of x in the HSO will see either this value or a later one
  - (for simplicity, assume that writes occur atomically)



## Summary for Sequential Consistency

- Maintain order between shared accesses in each processor



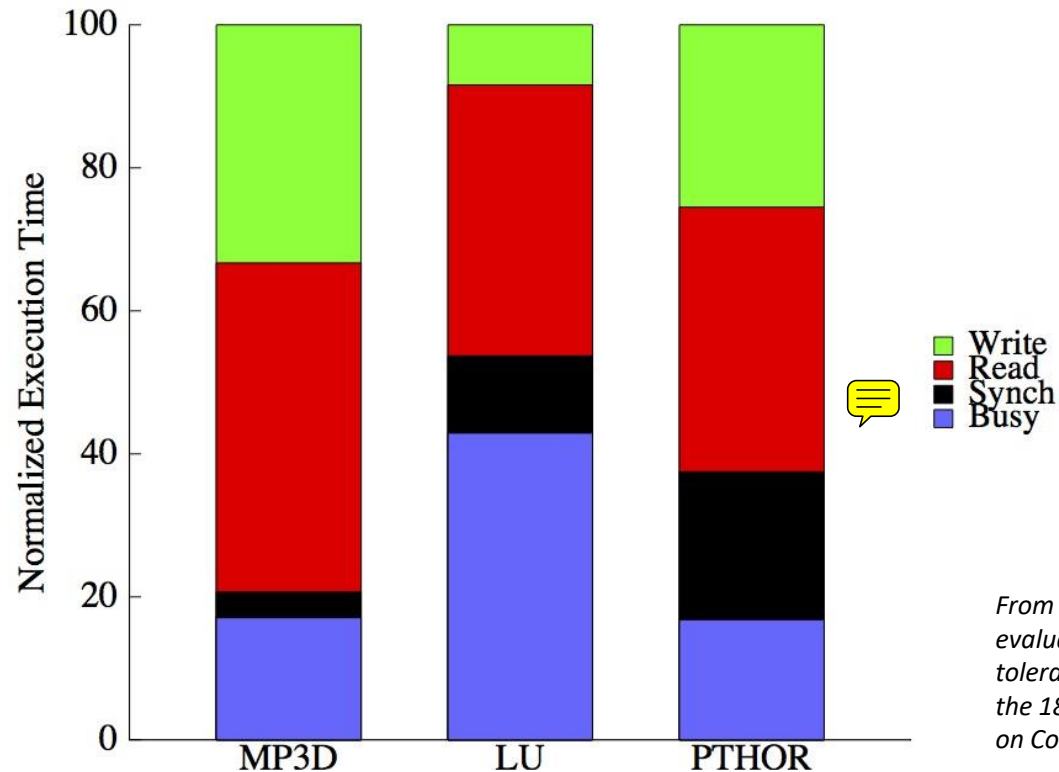
- Balloon analogy:
  - like putting a twist between each individual (ordered) gas particle



- Severely restricts common hardware and compiler optimizations

## Performance of Sequential Consistency

- Processor issues accesses **one-at-a-time** and **stalls for completion**

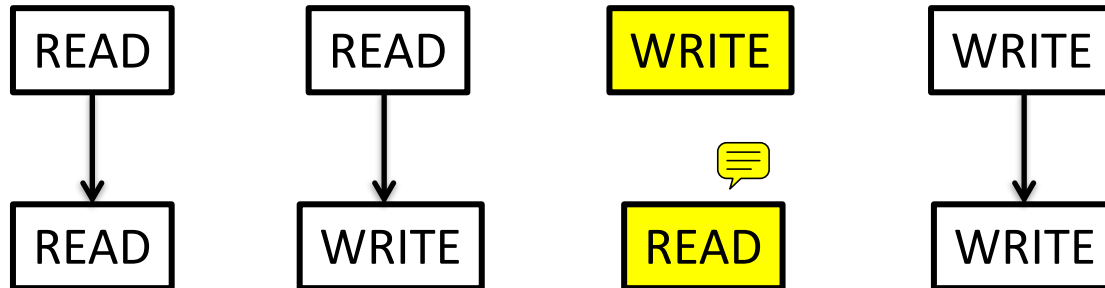


*From Gupta et al, "Comparative evaluation of latency reducing and tolerating techniques." In Proceedings of the 18th annual International Symposium on Computer Architecture (ISCA '91)*

- Low processor utilization** (17% - 42%) **even with caching**

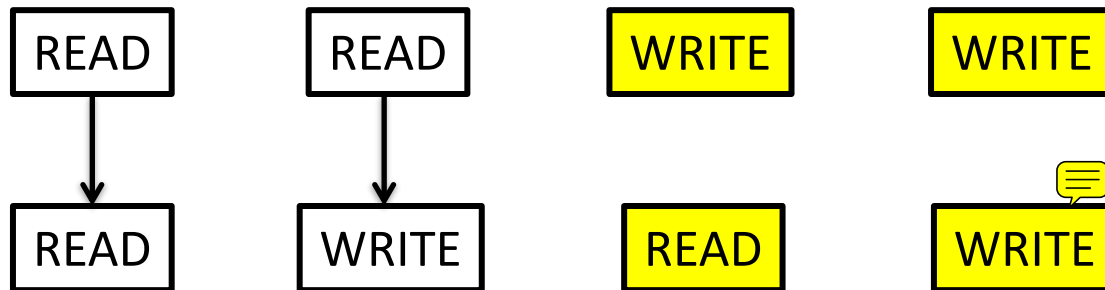
# Alternatives to Sequential Consistency

- Relax constraints on memory order



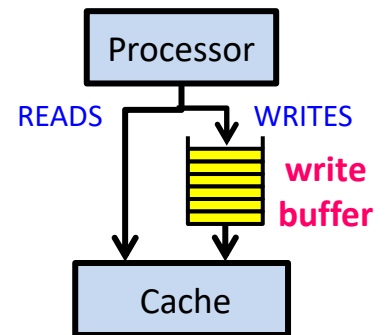
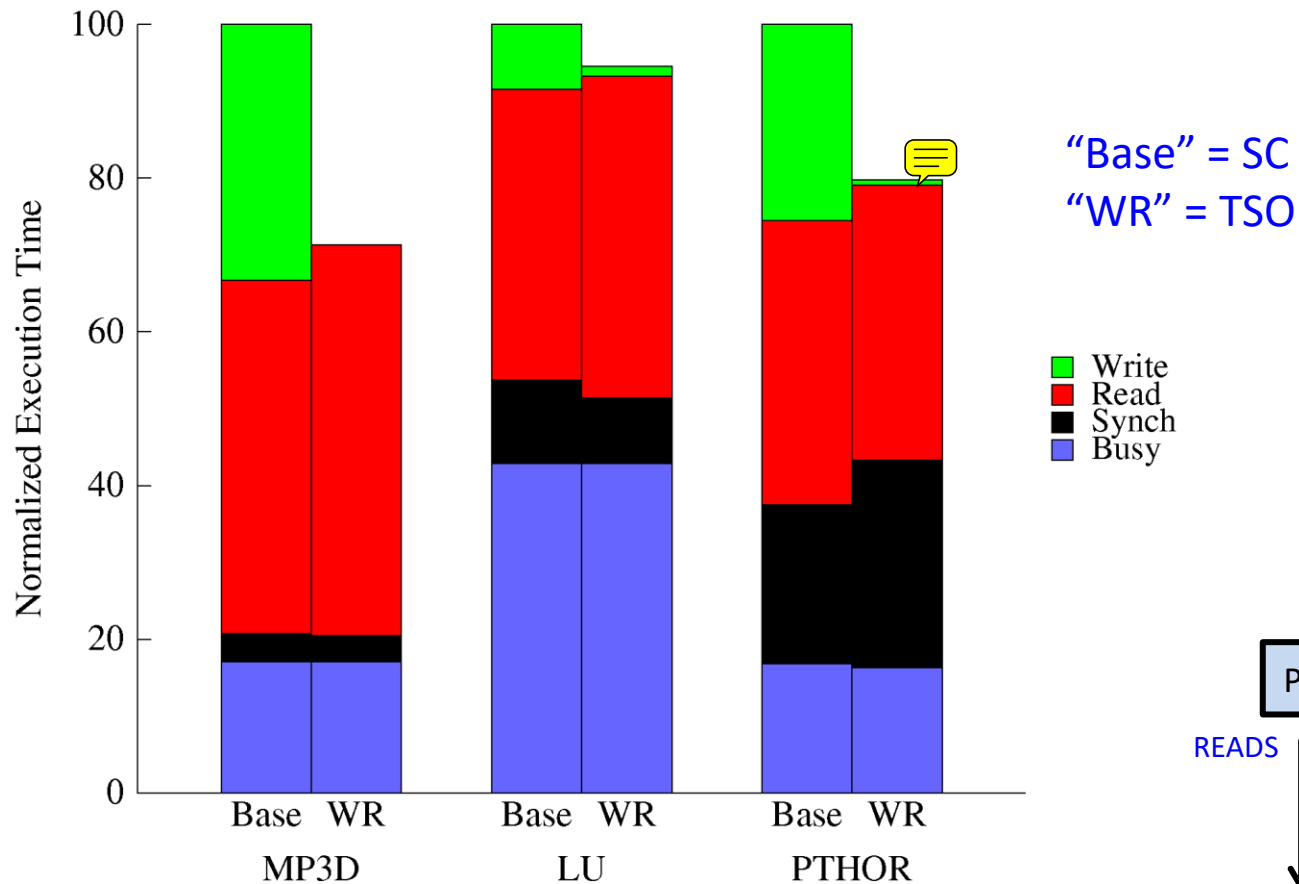
## Total Store Ordering (TSO) (Similar to Intel)

See Section 8.2 of “Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A: System Programming Guide, Part 1”, <http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.pdf>



## Partial Store Ordering (PSO)

## Performance Impact of TSO vs. SC

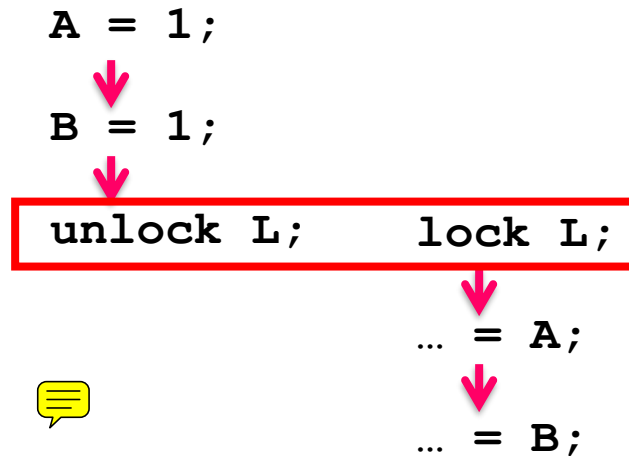


- Can use a **write buffer**
- Write latency is effectively hidden

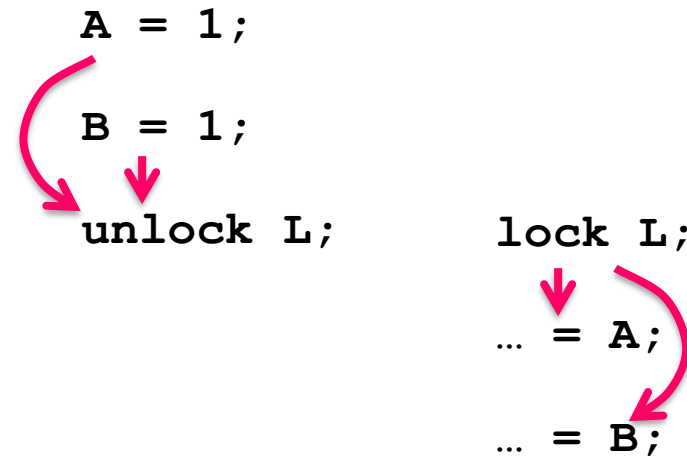
## But Can Programs Live with Weaker Memory Orders?

- “Correctness”: same results as sequential consistency
- Most programs don’t require strict ordering (all of the time) for “correctness”

### Program Order



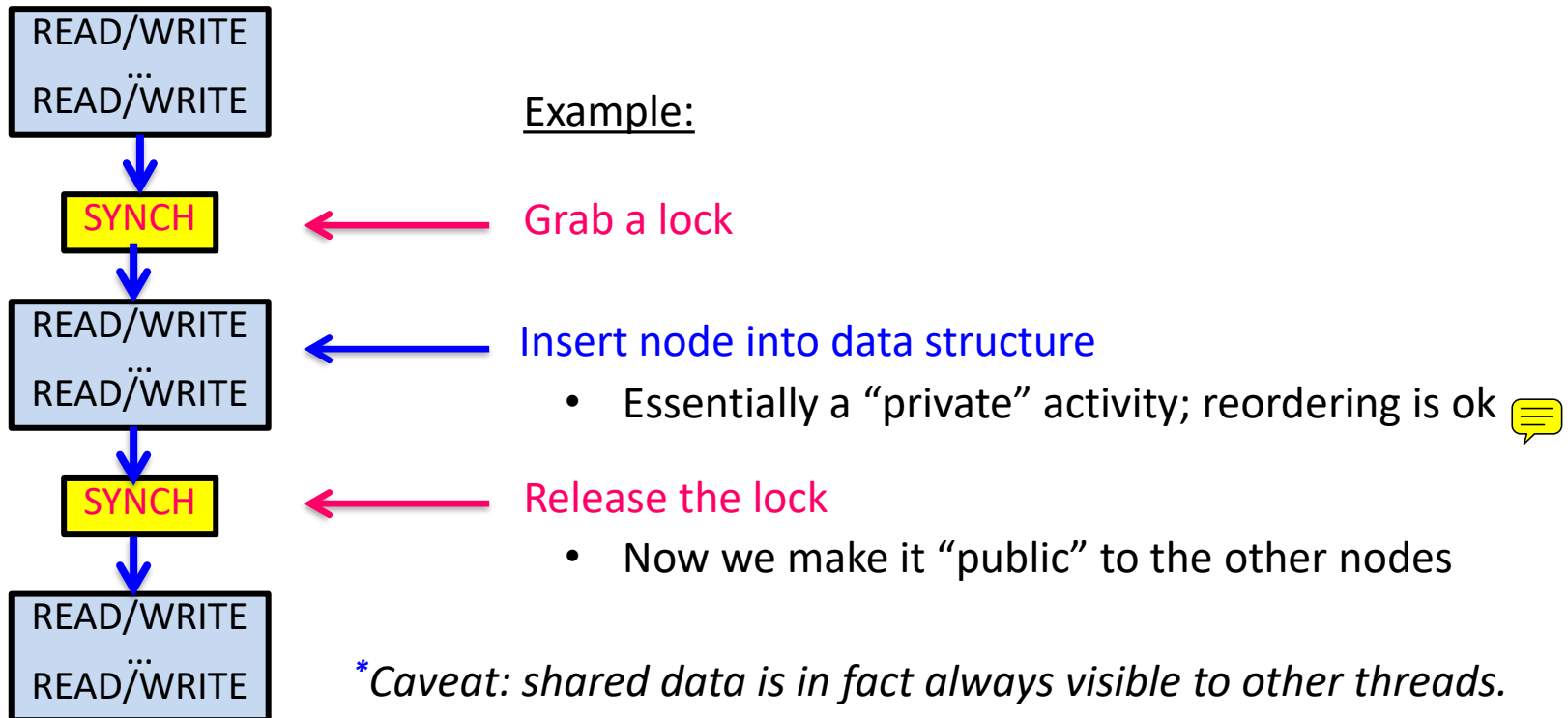
### Sufficient Order



- But how do we know when a program will behave correctly?
  - If all synchronization is explicitly identified
  - All updates of shared data is properly synchronized

# Optimizations for Synchronized Programs

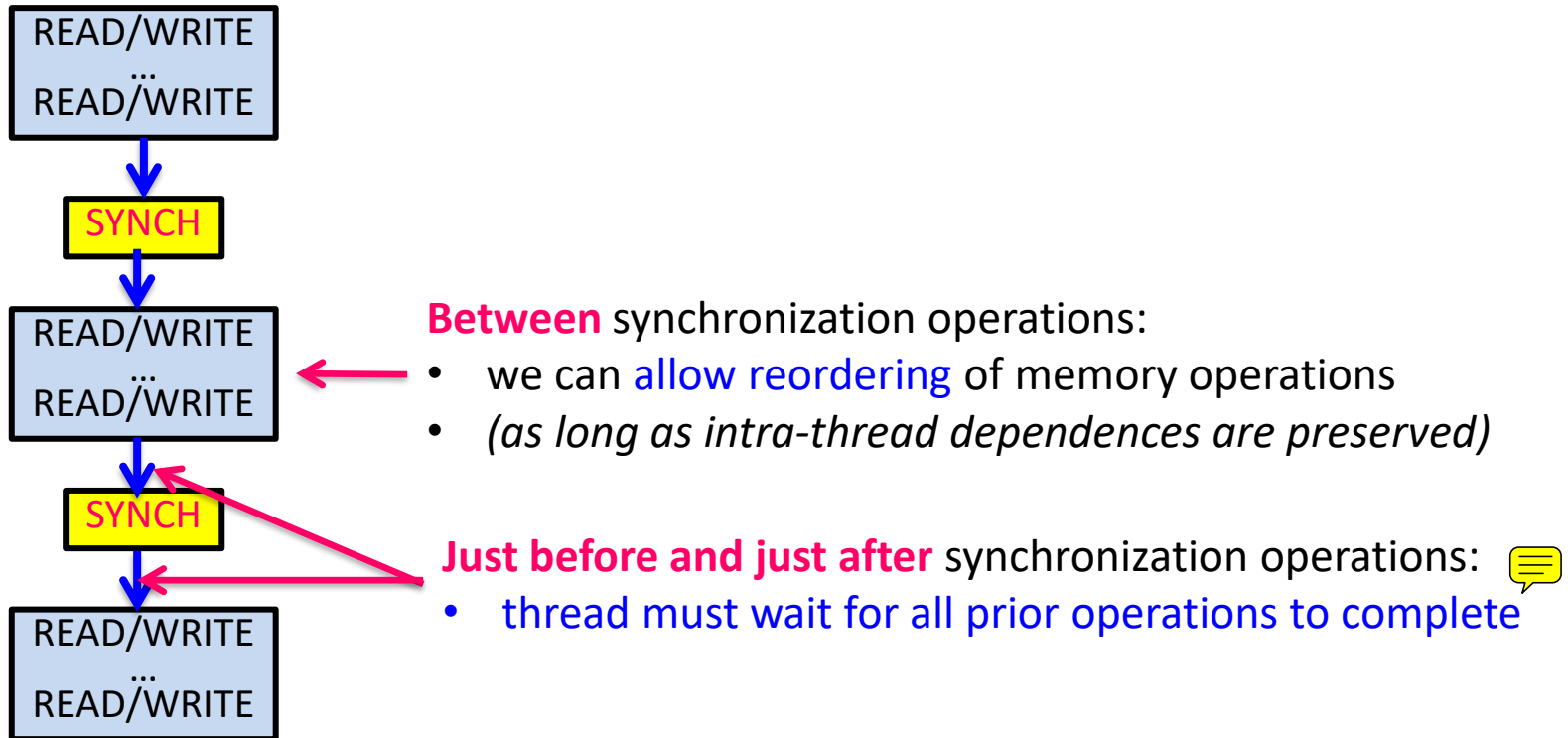
- Intuition: many parallel programs have mixtures of “private” and “public” parts\*
  - the “private” parts must be **protected by synchronization** (e.g., locks)
  - can we **take advantage of synchronization to improve performance?**





# Optimizations for Synchronized Programs

- Exploit information about synchronization

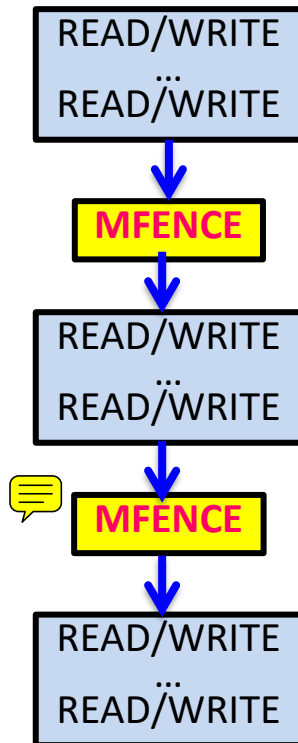


“Weak Ordering” (WO)

- properly synchronized programs should yield the **same result as on an SC machine**

## Intel's MFENCE (Memory Fence) Operation

- An **MFENCE** operation enforces the ordering seen on the previous slide:
  - does not begin until all prior reads & writes from that thread have completed
  - no subsequent read or write from that thread can start until after it finishes



Balloon analogy: it is a twist in the balloon

- no gas particles can pass through it

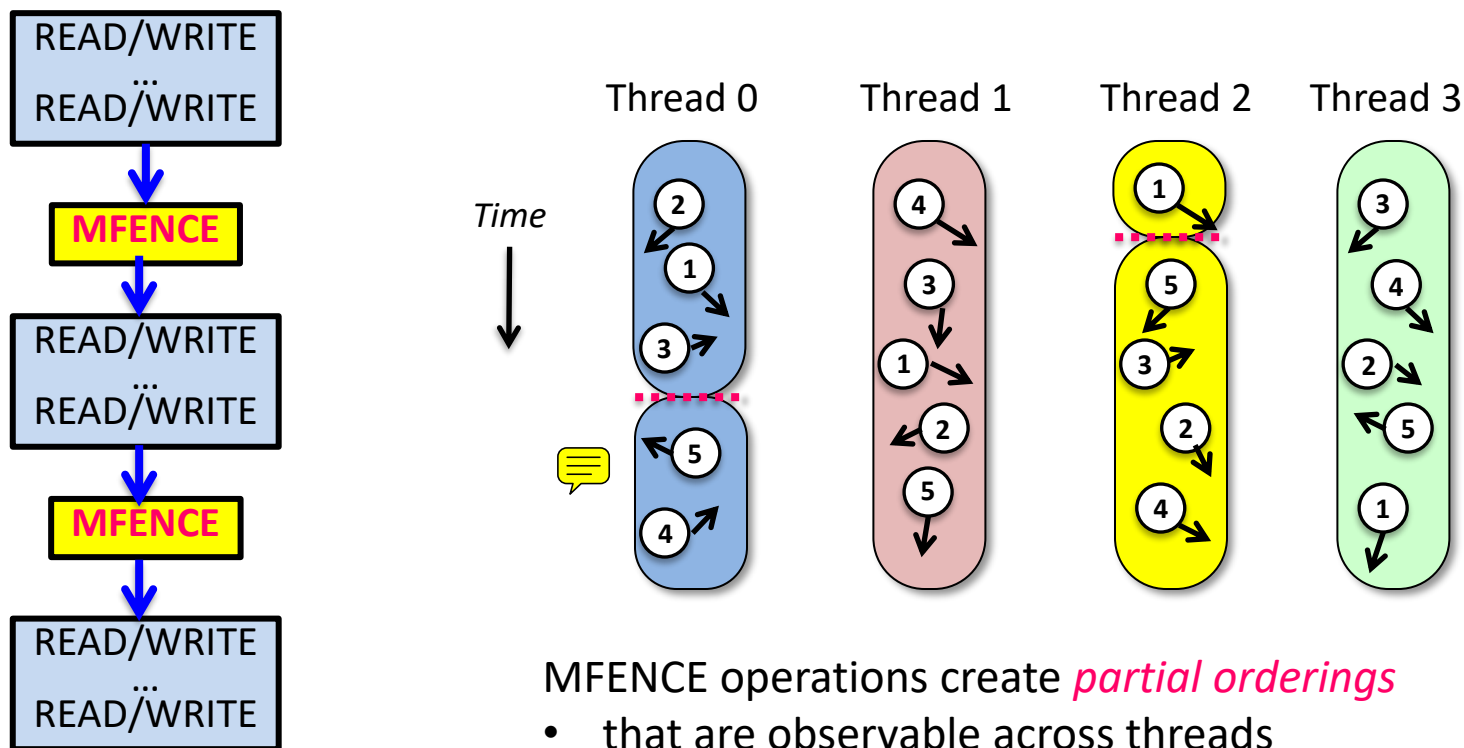


wikiHow  
(wikiHow)

Good news: **xchg** does this implicitly!

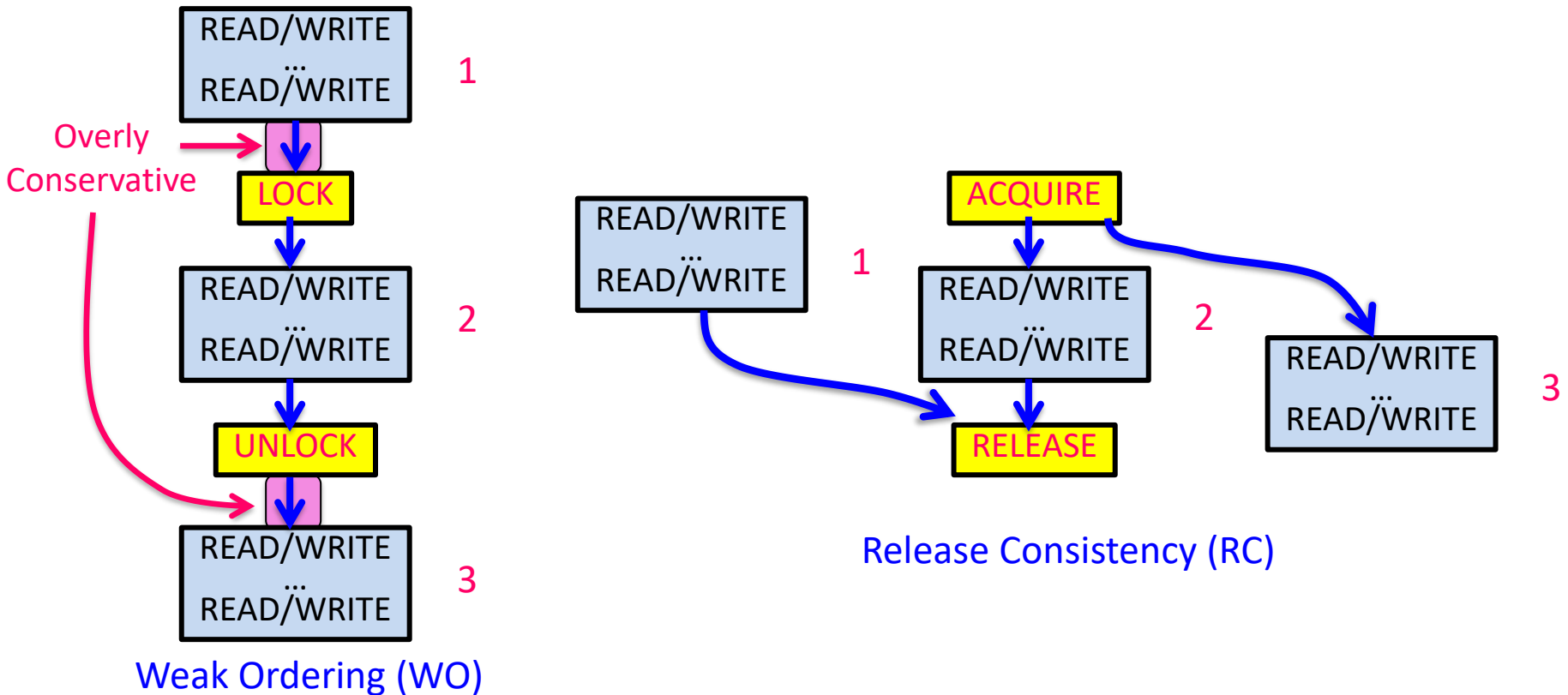
## Common Misconception about MFENCE

- MFENCE operations **do NOT** push values out to other threads
  - it is not a magic “make every thread up-to-date” operation
- Instead, they simply **stall the thread that performs the MFENCE**




# Exploiting Asymmetry in Synchronization: “Release Consistency”

- Lock operation: only gains (“acquires”) permission to access data
- Unlock operation: only gives away (“releases”) permission to access data



## Release Consistency in Pictures

- What does a barrier prevent? 
  - What does a dependency control?
  - How do control flow affect consistency?
  - If there are more than two threads, is consistency transitive?
- 
- For the following pictures, we are working off a release consistency model roughly analogous to ARM
  - <https://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf>

## Initial Example

- Two threads
  - Thread 0 writes to locations X, Y
  - Thread 1 reads from locations X, Y

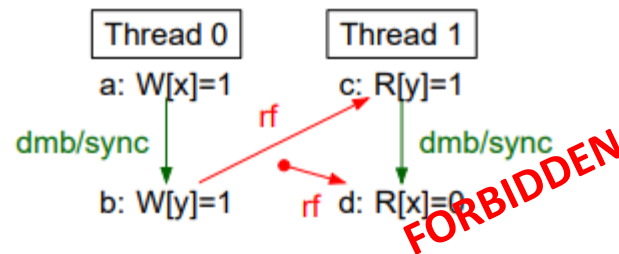
MP	Pseudocode
Thread 0	Thread 1
x=1 y=1	r1=y r2=x
Initial state: $x=0 \wedge y=0$	

- What values can be observed? Can  $r1 = 1$  and  $r2 = 0$ ?
  - SC?
    - NO
  - TSO?
    - NO
  - Release Consistency?
    - Yes?!

# Barriers

- Placing a full barrier between the operations in threads 0 and 1
  - All examples are release consistency now

MP+dmb/syncs	Pseudocode
Thread 0	Thread 1
x=1 dmb/sync y=1	r1=y dmb/sync r2=x
Initial state: $x=0 \wedge y=0$	

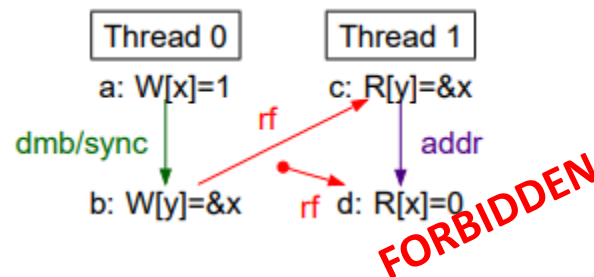


## Adding a dependency between reads



- Replacing the barrier with an address dependency
  - Can thread 1 read \*r1 and get 0?

MP+dmb/sync+addr'	Pseudocode
Thread 0	Thread 1
x=1 dmb/sync y=&x	r1=y  r2=*r1
Initial state: $x=0 \wedge y=0$	

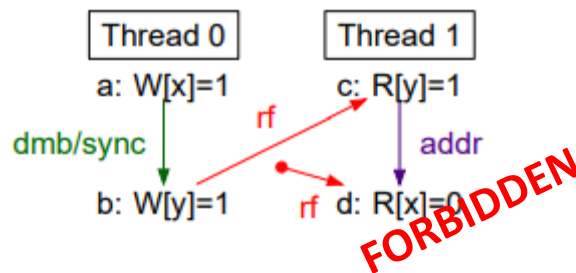




## Extending dependencies

- Programmers can force the value to still be a dependency **without \*using\* it**
  - The read of x “depends” on the value of y

MP+dmb/sync+addr	Pseudocode
Thread 0	Thread 1
x=1 dmb/sync y=1	r1=y r3=(r1 xor r1) r2=*&x + r3
Initial state: $x=0 \wedge y=0$	

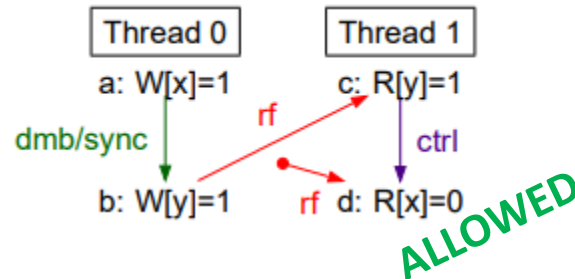


## Control Dependencies

- Rather than using the value in the address, can we control the order using branches?
  - If there is a branch or loop between the two operations, does that order them?

MP+dmb/sync+ctrl Pseudocode

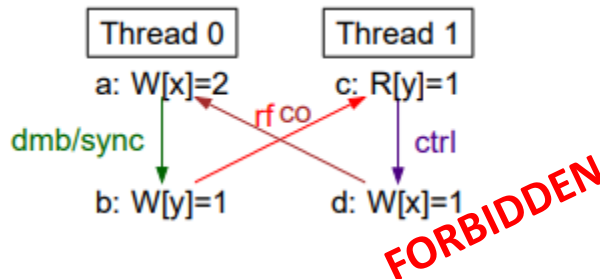
Thread 0	Thread 1
x=1 dmb/sync y=1	r1=y if (r1 == r1) {} <b>r2=x</b>
Initial state: $x=0 \wedge y=0$	



## Control Dependencies Matter

- Writes respect the branch
  - Without the branch, thread 1 can write before its read
  - With the branch (or a direct value dependence), the write ordering is forced

S+dmb/sync+ctrl	Pseudocode
Thread 0	Thread 1
x=2 dmb/sync y=1	r1=y if (r1==r1) { } x=1
Initial state: $x=0 \wedge y=0$	



## Are these orderings common?

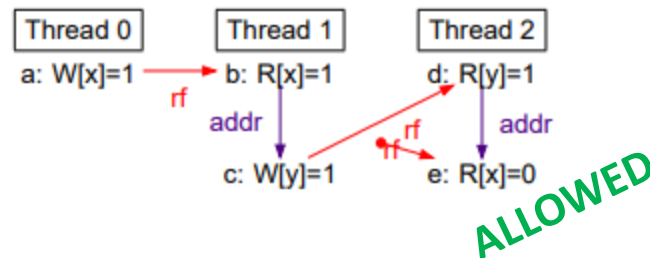
- Testing on ARM verifies that the orderings occur as expected.

		ARM			
	Kind	Tegra2	Tegra3	APQ8060	A5X
MP	Allow	40M/3.8G	138k/16M	61k/552M	437k/185M
MP+dmb/sync+po	Allow	3.1M/3.9G	50/28M	69k/743M	249k/195M
MP+dmb/sync+addr	Forbid	0/29G	0/39G	0/26G	0/2.2G
MP+dmb/sync+ctrl	Allow	5.7M/3.9G	1.5k/53M	556/748M	1.5M/207M
MP+dmb/sync+ctrlsib/isync	Forbid	0/29G	0/39G	0/26G	0/2.2G
S+dmb/sync+po	Allow	271k/4.0G	84/58M	357/1.8G	211k/202M
S+dmb/sync+ctrl	Forbid	0/24G	0/39G	0/26G	0/2.2G
S+dmb/sync+data	Forbid	0/24G	0/39G	0/26G	0/2.2G

## Three threads interact

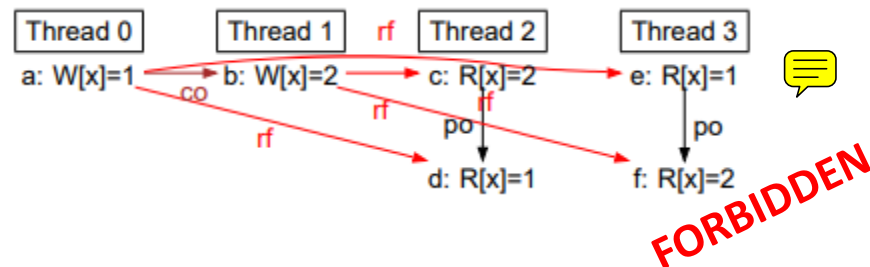
- How do written values propagate to multiple threads?
  - Is it possible for  $r3 = 0$ , while  $r1$  and  $r2$  are 1?
  - Adding a barrier in thread 1 makes this result forbidden

WRC+addr		Pseudocode
Thread 0	Thread 1	Thread 2
$x=1$	$r1=x$ $*(&y+r1-r1) = 1$	$r2=y$ $r3 = *(&x + r2 - r2)$
Initial state: $x=0 \wedge y=0$		



## Is there anything left?

- Yes! Cache coherence must still work.
- Threads must agree on an order of writes.
- And remember that coherence is per-block



# Take-Away Messages on Memory Consistency Models

- **DON'T** use only normal memory operations for synchronization
  - e.g., Peterson's solution (from Synchronization #1 lecture)

```
boolean want[2] = {false, false};  
int turn = 0;
```

```
want[i] = true;
```

```
turn = j;
```

```
while (want[j] && turn == j)
```

```
    continue;
```

```
... critical section ...
```

```
want[i] = false;
```



- **DON'T** use synchronization operations except when necessary
  - **Recall:** you have likely never seen this issue before today

## Take-Away Messages on Memory Consistency Models

- **DO** use either **explicit synchronization operations** (e.g., **xchg**) and/or\* **fences**

```
while (!xchg(&lock_available, 0)  
    continue;  
... critical section ...  
xchg(&lock_available, 1);
```



- **DO** utilize the capabilities provided by your language
  - C has (optionally) `stdatomic.h`
  - Can also use volatile and hardware fences

\*Not all ISAs treat synchronization operations as fences



# Summary

- Memory Consistency Models
  - Be sure to use fences or explicit synchronization operations when ordering matters
    - don't synchronize through normal memory operations!

## Appendix

- <https://www.hpl.hp.com/techreports/Compaq-DEC/WRL-95-7.pdf>
- <https://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf>