Parallelism: Memory Consistency Models and Scheduling

Todd C. Mowry, Dave Eckhardt & Dave O'Hallaron

I. Memory Consistency Models

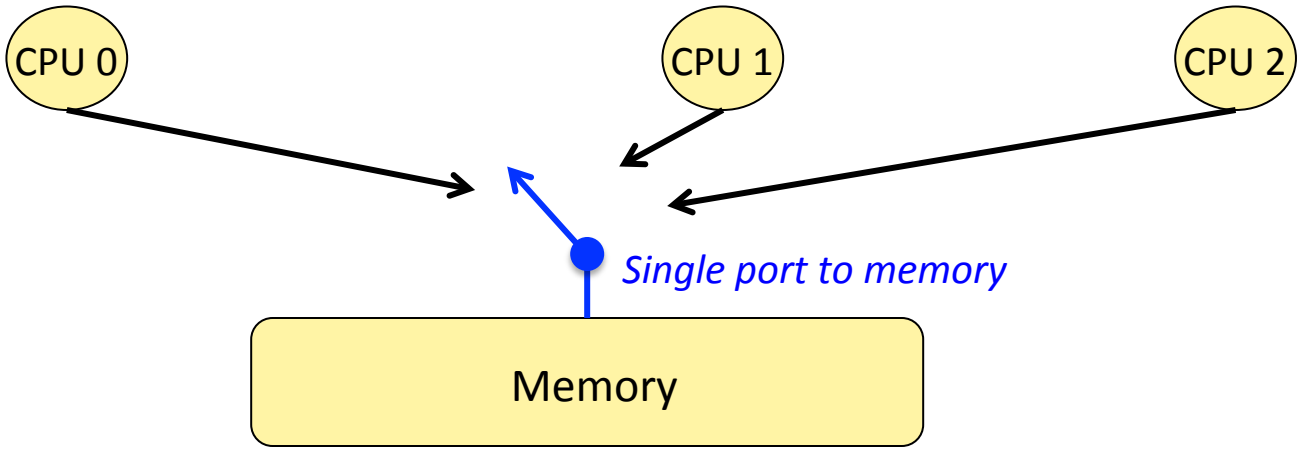
II. Process Scheduling Revisited

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

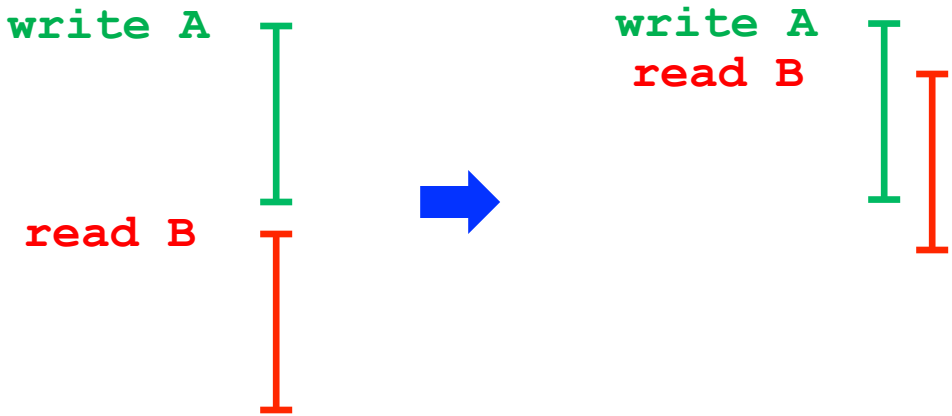
- What programmers intuitively expect:
  - processor atomically performs one instruction at a time, in program order

- In reality:
  - if the processor actually operated this way, it would be painfully slow
  - instead, the processor *aggressively reorders instructions* to hide memory latency

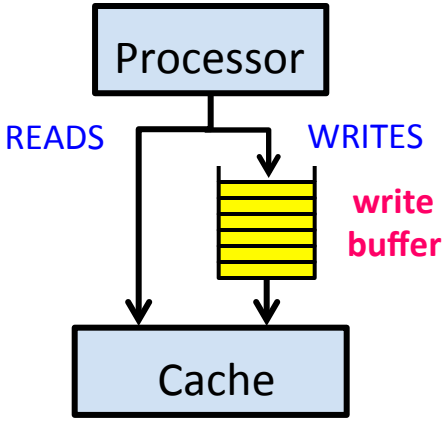
- Upshot:
  - *within a given thread*, the processor preserves the program order illusion
  - but this illusion has nothing to do with what happens in physical time!
  - from the perspective of *other threads*, all bets are off!

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

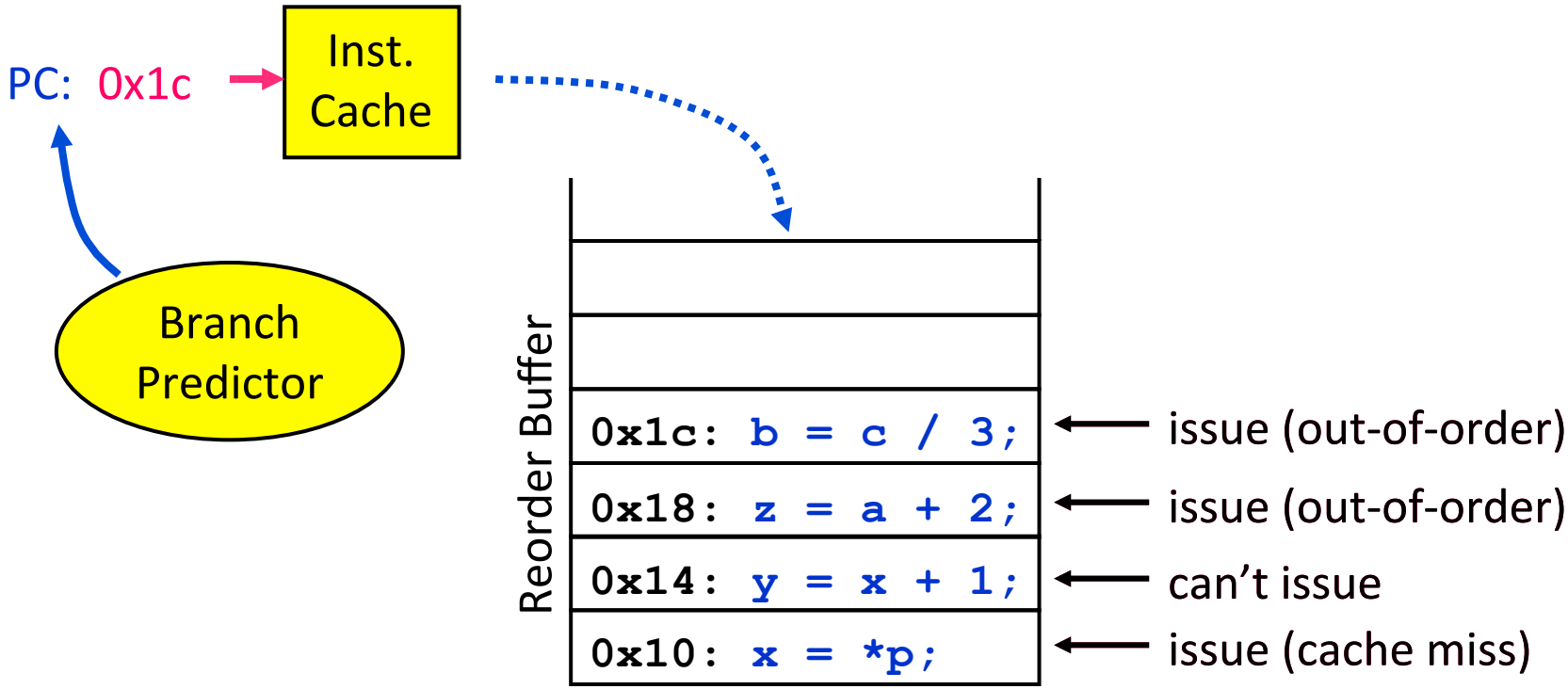
```
x = *p;
y = x + 1;
z = a + 2;
b = c / 3;
if (x != z)
  d = e - 7;
else d = e + 5;
…
```

if hardware guesses that this is true

then execute "then" part (speculatively) (without waiting for **x** or **z**)

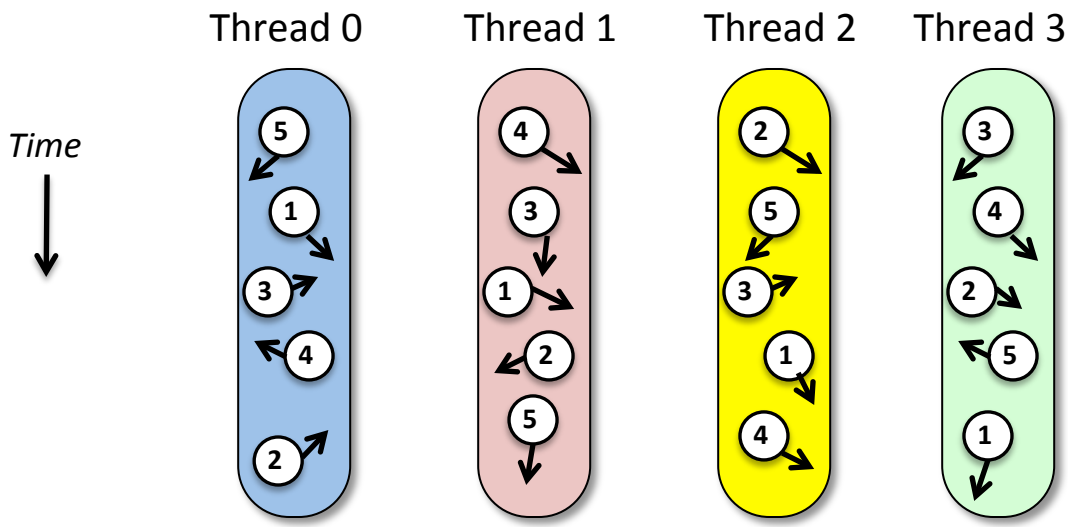
How Out-of-Order Pipelining Works in Modern Processors

- Fetch and graduate instructions in-order, but issue out-of-order



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

Analogy: Gas Particles in Balloons



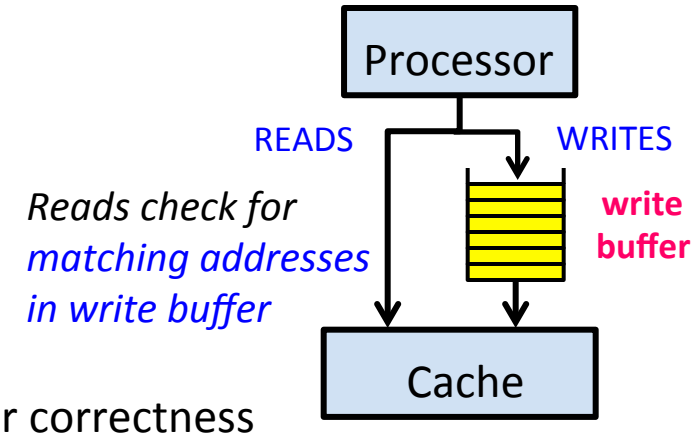
(wikiHow)

- 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

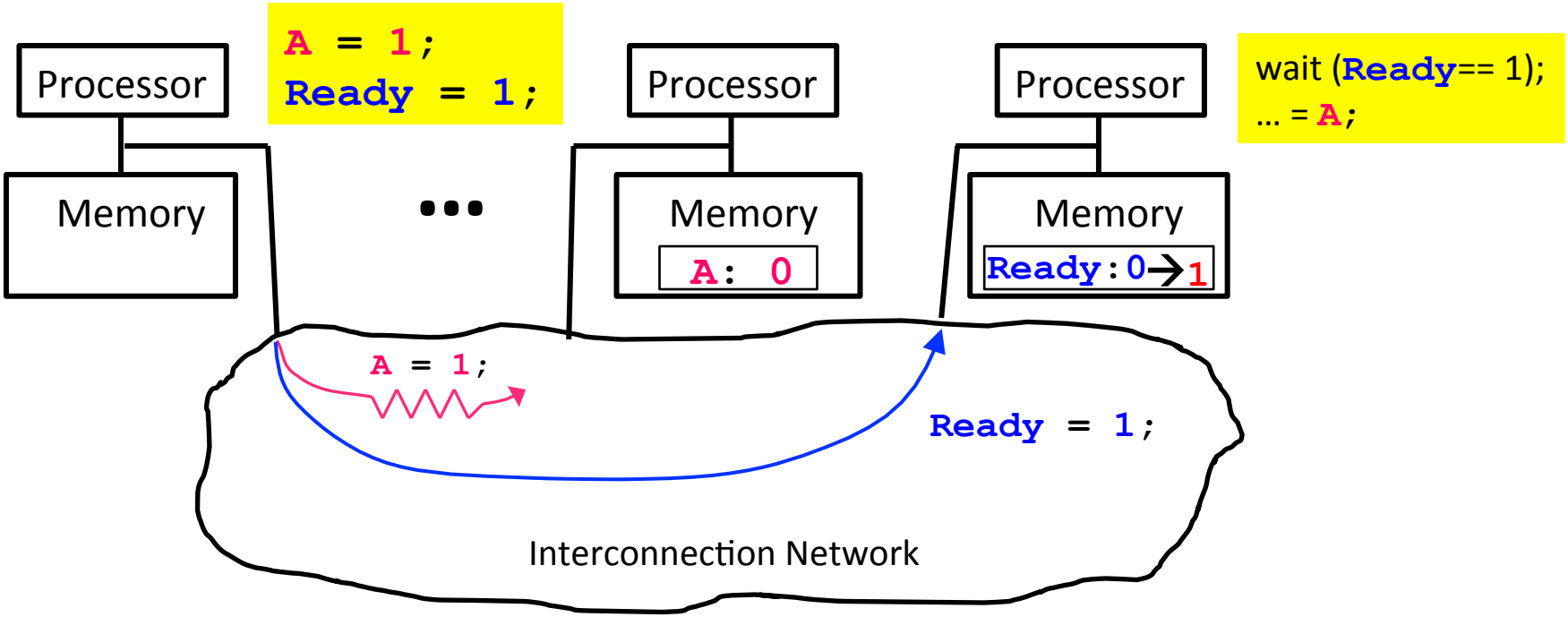
In Parallel Machines (with a Shared Address Space)

- Order between accesses to different locations becomes important

(Initially A and Ready = 0)

P1	P2
`A = 1;`	
`Ready = 1;`	
	`while (Ready != 1);`
	`… = A;`

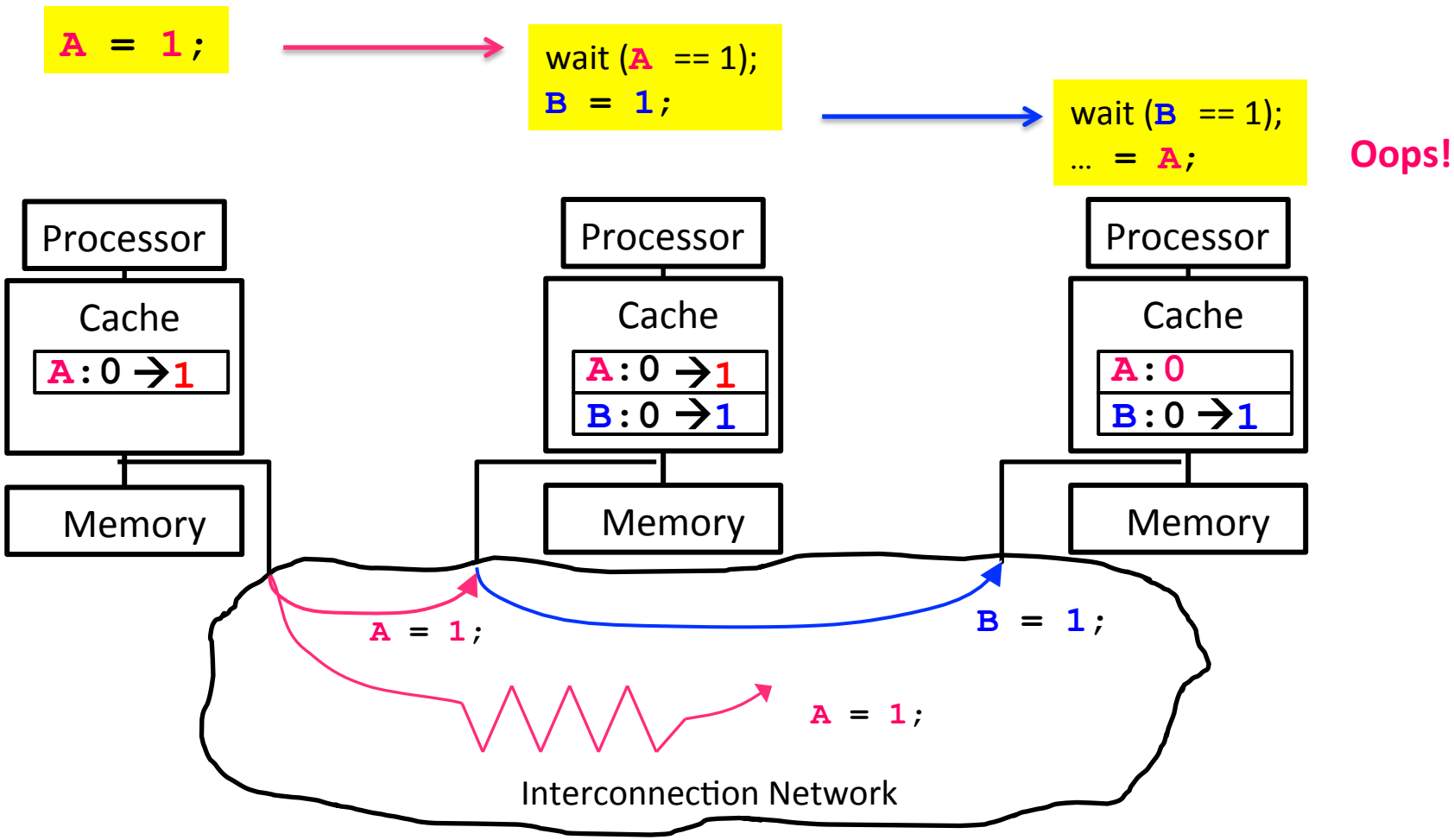
How Unsafe Reordering Can Happen



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

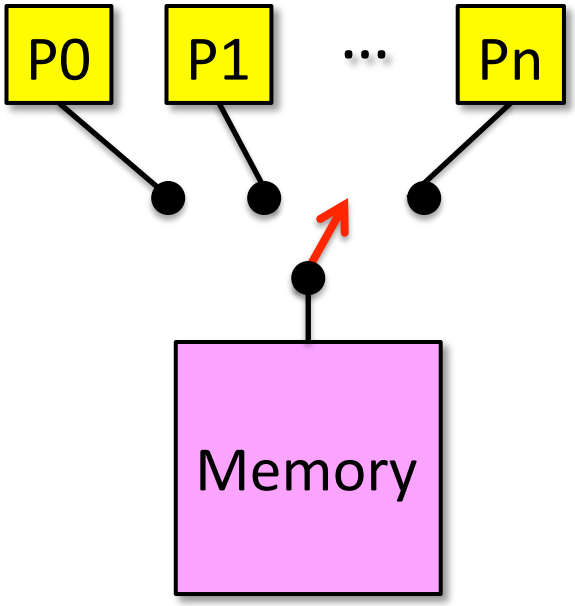
Caches Complicate Things More

- 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		P1	
`A = 1`	(a)		
`Ready = 1`	(b)	`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		P1	
`want[0] = 1`	(a)	`want[1] = 1`	(c)
`x = want[1]`	(b)	`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

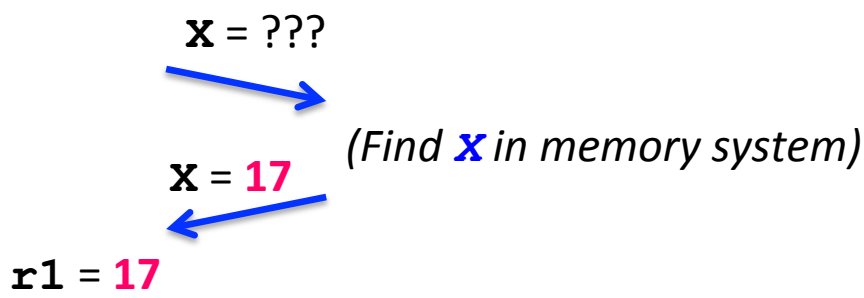
2. For each processor, delay start of memory access until previous one completes
   → each processor has only one outstanding memory access at a time

- 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
```



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
```

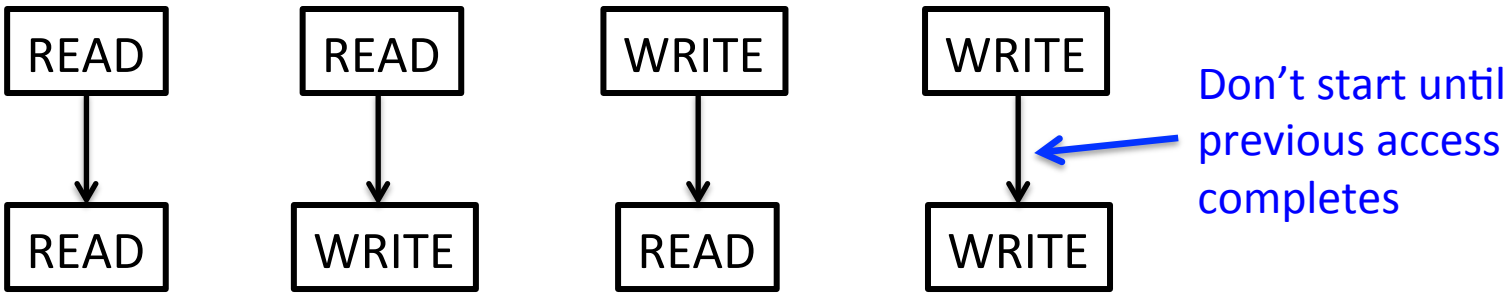
x = 23

(Commit to memory order)
(aka "serialize")

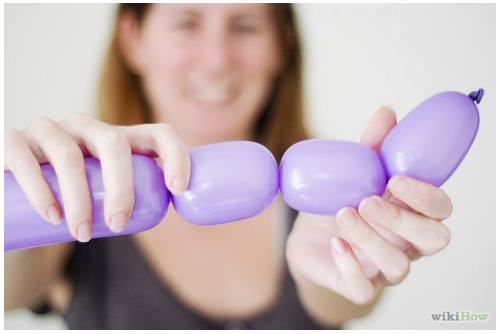
- What does "visible" mean?
  - it does NOT mean that other processors have necessarily seen the value yet
  - it means the new value is committed to the hypothetical serializable order (HSO)
    - a later read of **x** in the HSO will see either this value or a later one
  - (for simplicity, assume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

Summary for Sequential Consistency

- Maintain order between shared accesses in each processor



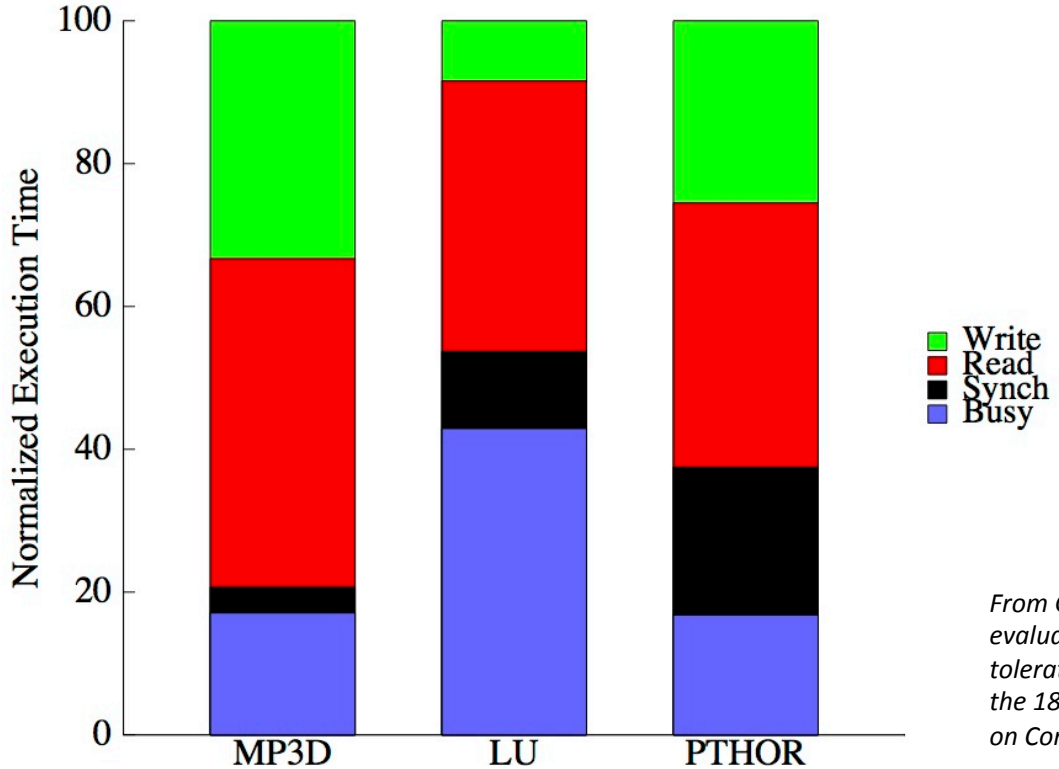
- Balloon analogy:
  - like putting a twist between each individual (ordered) gas particle



- Severely restricts common hardware and compiler optimizations

Performance of Sequential Consistency

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

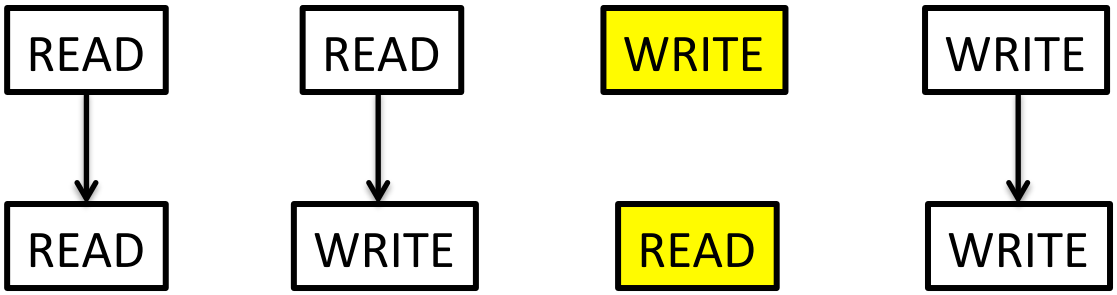


From Gupta et al, "Comparative evaluation of latency reducing and tolerating techniques." In Proceedings of the 18th annual International Symposium on Computer Architecture (ISCA '91)

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

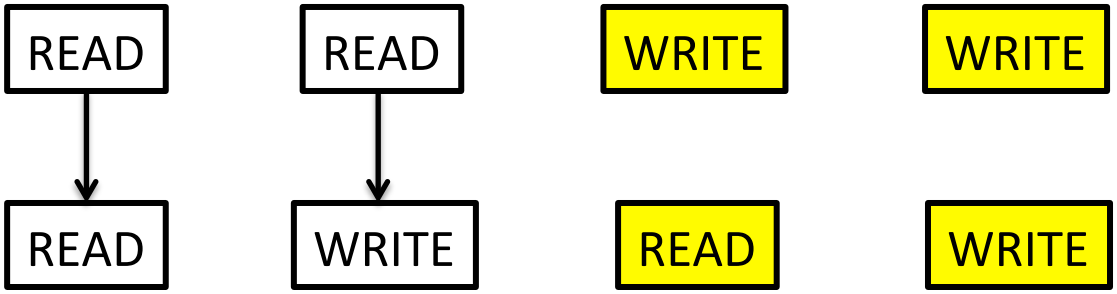
Alternatives to Sequential Consistency

- Relax constraints on memory order



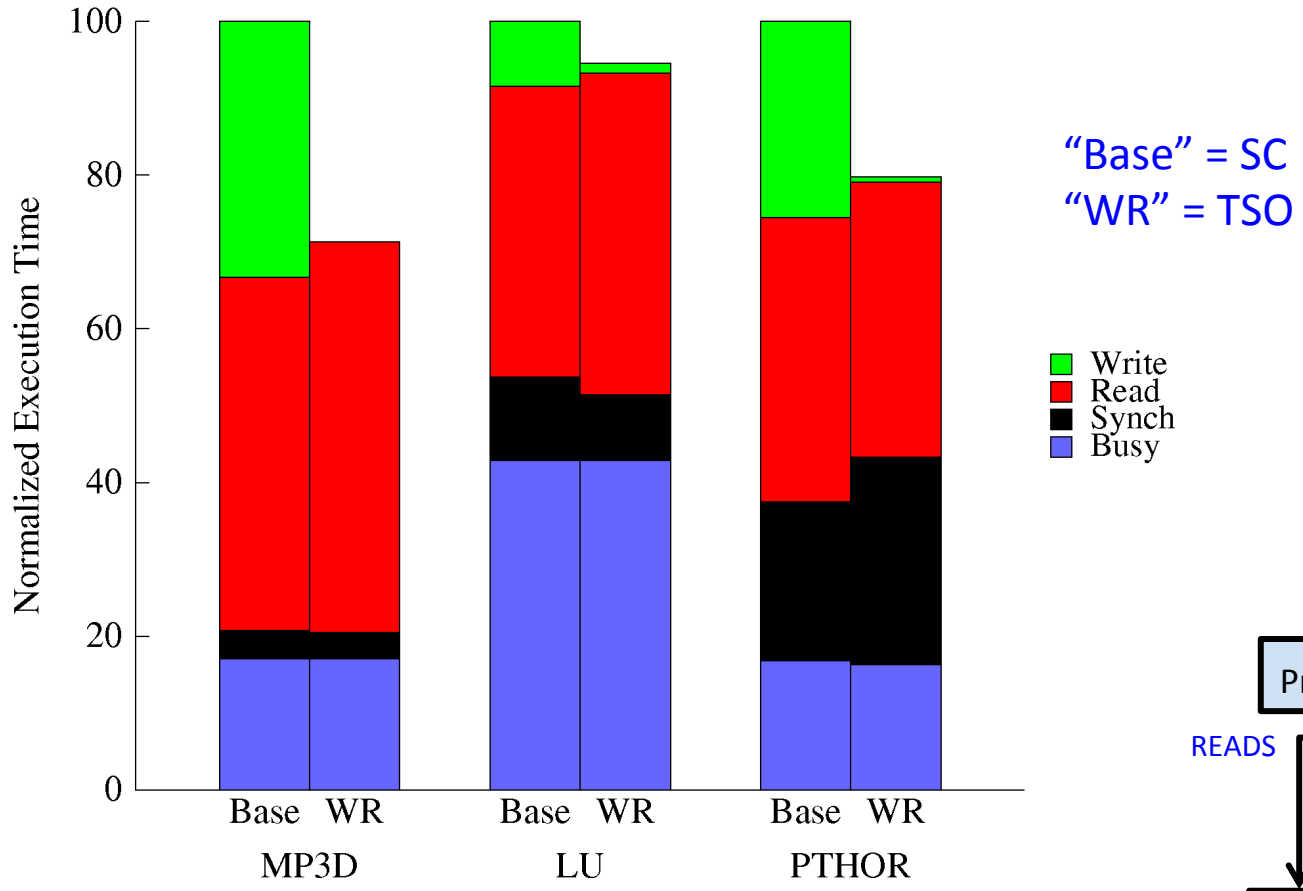
Total Store Ordering (TSO) (Similar to Intel)

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

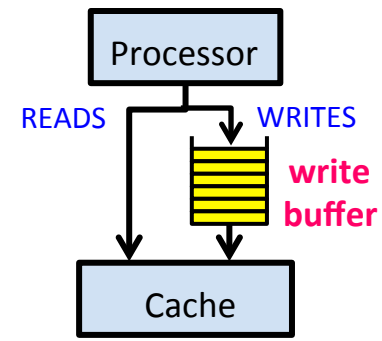


Partial Store Ordering (PSO)

Performance Impact of TSO vs. SC



"Base" = SC
"WR" = TSO



- 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

```
A = 1;
B = 1;
unlock L;

lock L;
… = A;
… = B;
```

Sufficient Order

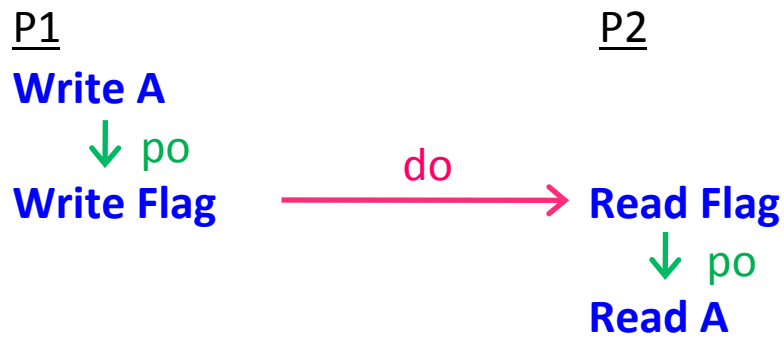
```
A = 1;
B = 1;
unlock L;

lock L;
… = A;
… = B;
```

- But how do we know when a program will behave correctly?

Identifying Data Races and Synchronization

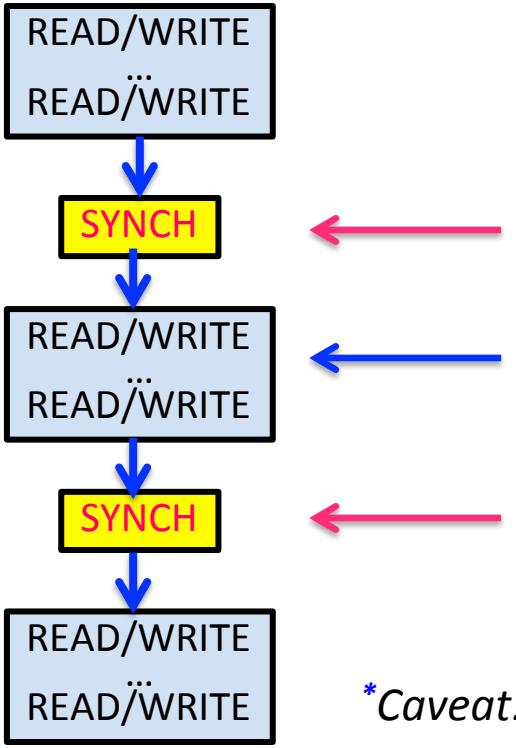
- Two accesses *conflict* if:
  - (i) access same location, and (ii) at least one is a write
- Order accesses by:
  - program order (po)
  - dependence order (do): op1 --> op2 if op2 reads op1



- Data Race:
  - two conflicting accesses on different processors
  - not ordered by intervening accesses
- Properly Synchronized Programs:
  - all synchronizations are explicitly identified
  - all data accesses are ordered through synchronization

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?



Example:

Grab a lock

Insert node into data structure

- Essentially a "private" activity; reordering is ok

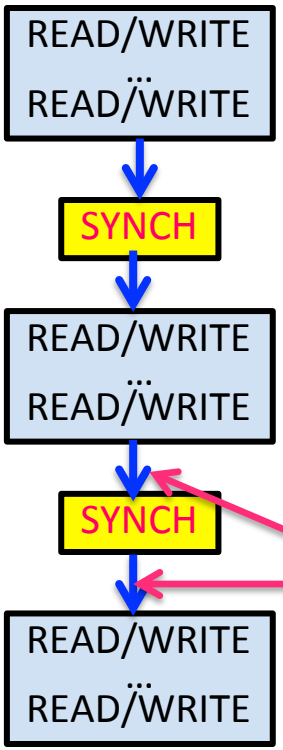
Release the lock

- Now we make it "public" to the other nodes

**Caveat: shared data is in fact always visible to other threads.*

Optimizations for Synchronized Programs

- Exploit information about synchronization



Between synchronization operations:
- we can allow reordering of memory operations
- *(as long as intra-thread dependences are preserved)*

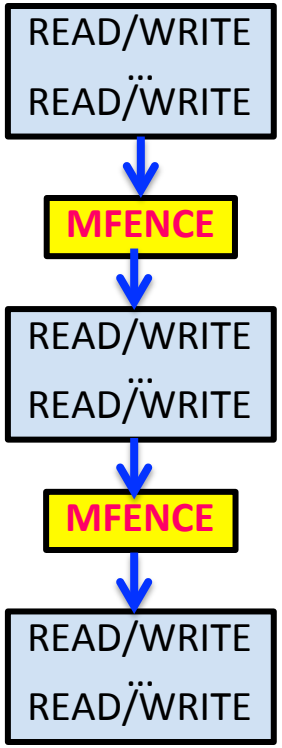
Just before and just after synchronization operations:
- thread must wait for all prior operations to complete

"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

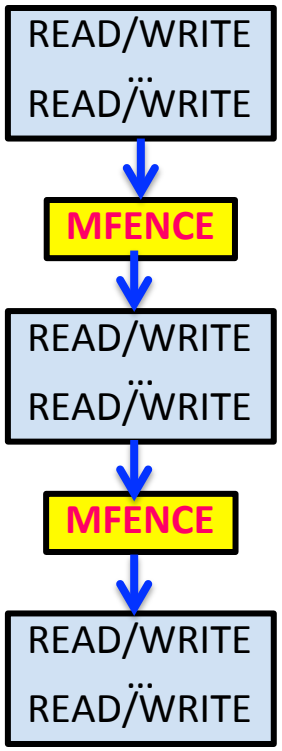
- no gas particles can pass through it

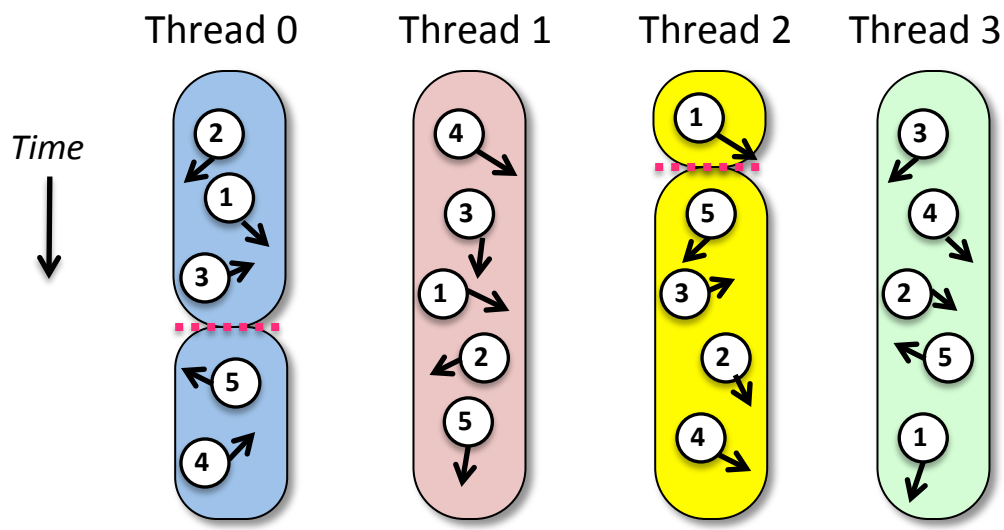
(wikiHow)

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



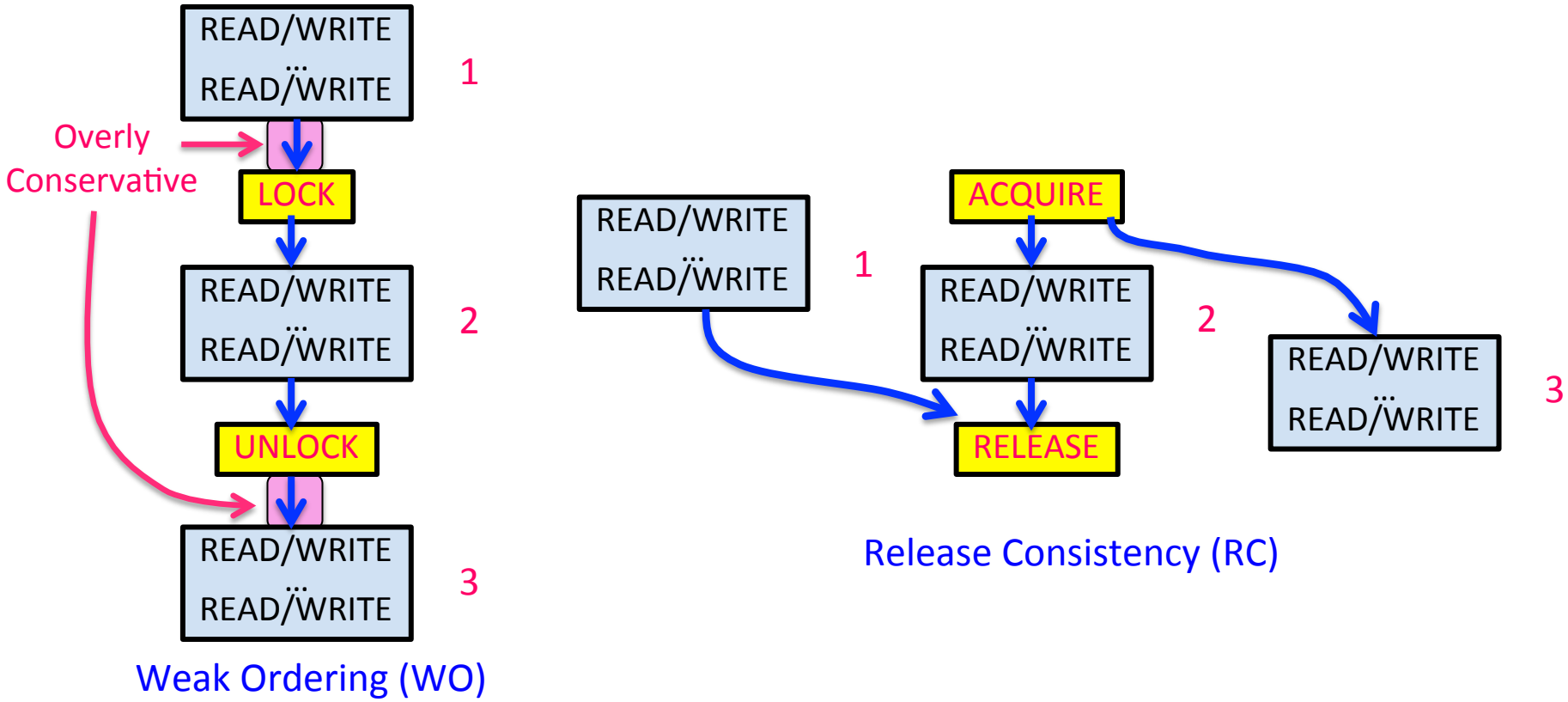


MFENCE operations create *partial orderings*

- that are observable across threads

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



Intel's Full Set of Fence Operations

- In addition to **MFENCE**, Intel also supports two other fence operations:
  - **LFENCE**: serializes only with respect to load operations (not stores!)
  - **SFENCE**: serializes only with respect to store operations (not loads!)
    - Note: It does slightly more than this; see the spec for details:
      - *Section 8.2.5 of "Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1*

- In practice, you are most likely to use:
  - **MFENCE**
  - **xchg**

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;
```

Exercise for the reader: Where should we add fences (and which type) to fix this?

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

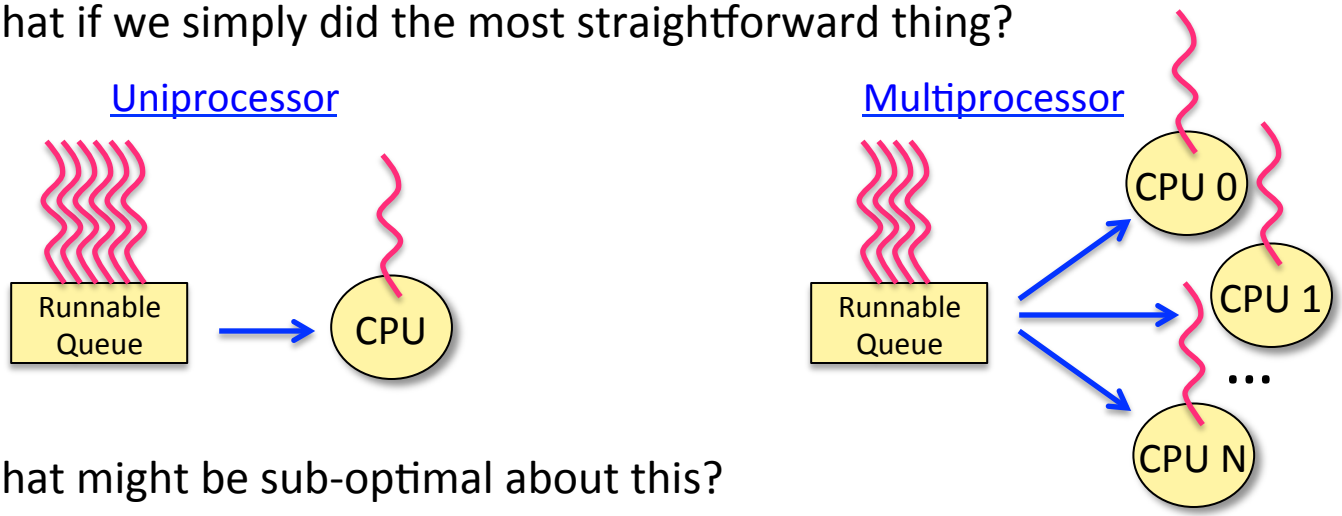
```
while (!xchg(&lock_available, 0)
  continue;
... critical section ...
xchg(&lock_available, 1);
```

Outline

- Memory Consistency Models
- Process Scheduling Revisited

Process Scheduling Revisited: Scheduling on a Multiprocessor

- What if we simply did the most straightforward thing?

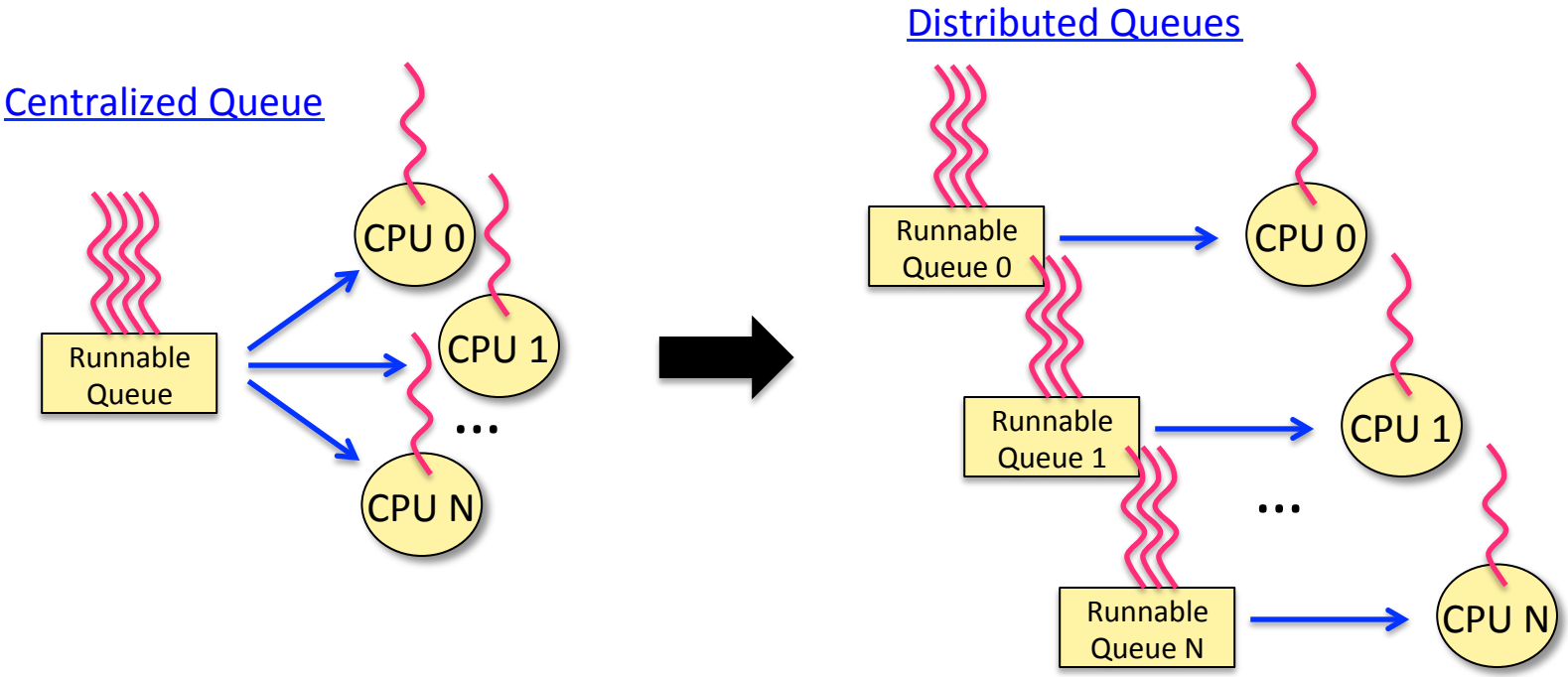


- What might be sub-optimal about this?
  - contention for the (centralized) run queue
  - migrating threads away from their data (disrupting data locality)
    - data in caches, data in nearby NUMA memories
  - disrupting communication locality between groups of threads
  - de-scheduling a thread holding a lock that other threads are waiting for
- Not just a question of when something runs, but also where it runs
  - need to optimize for *space* as well as time

Scheduling Goals for a Parallel OS

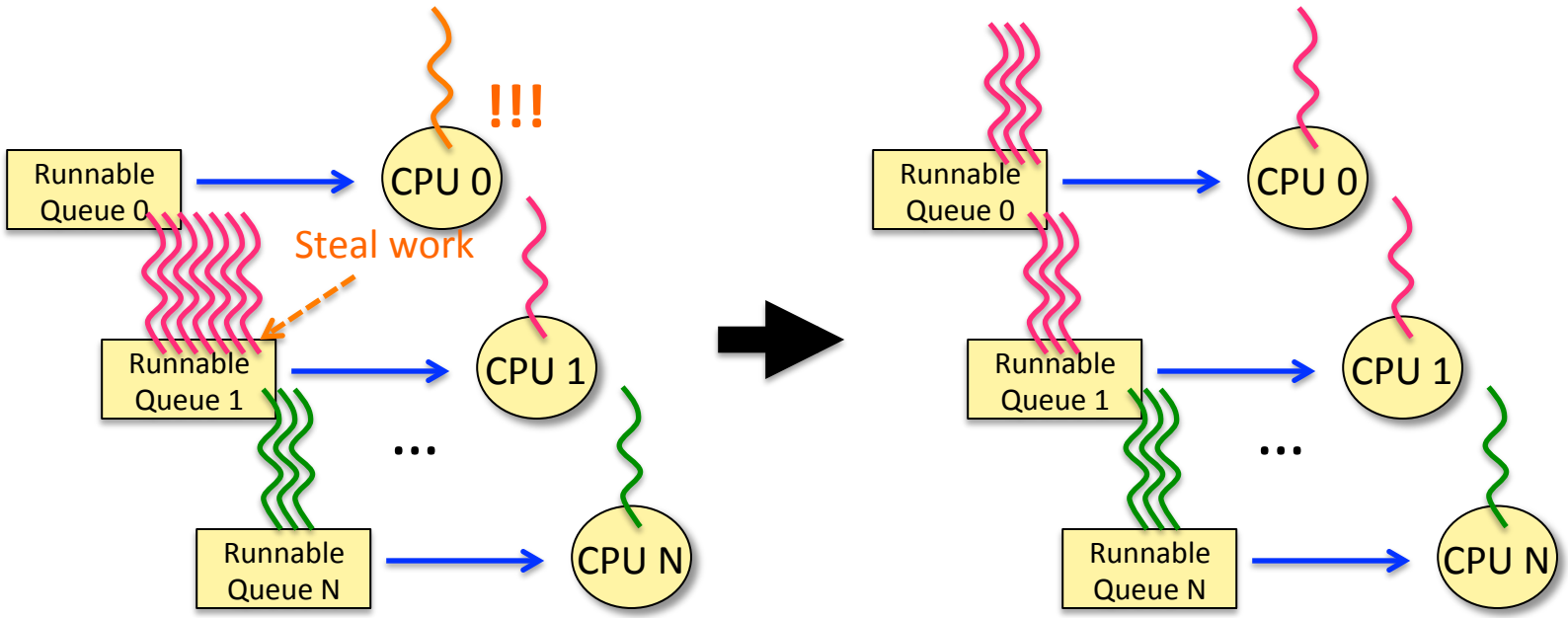
1. Load Balancing
   - try to distribute the work evenly across the processors
     - avoid having processors go idle, or waste time searching for work
2. Affinity
   - try to always restart a task on the same processor where it ran before
     - so that it can still find its data in the cache or in local NUMA memory
3. Power conservation (?)
   - perhaps we can power down parts of the machine if we aren't using them
     - how does this interact with load balancing? Hmm…
4. Dealing with heterogeneity (?)
   - what if some processors are slower than others?
     - because they were built that way, or
     - because they are running slower to save power

Alternative Designs for the Runnable Queue(s)



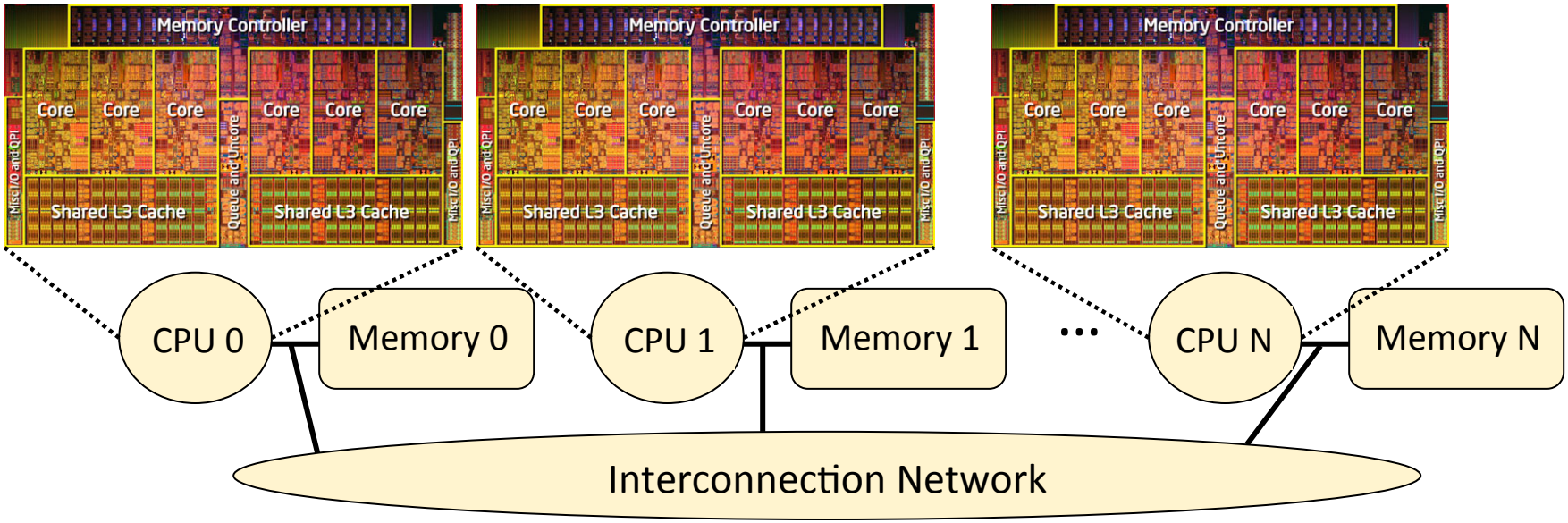
- Advantages of Distributed Queues?
  - easy to maintain affinity, since a blocked process stays in local queue
  - minimal locking contention to access queue
- But what about load balancing?
  - one solution: need to steal work from other queues sometimes

Work Stealing with Distributed Runnable Queues



- Pull model:
  - CPU notices its queue is empty (or below threshold), and steals work
- Push model:
  - kernel daemon periodically checks queue lengths, moves work to balance
- Many systems use both push and pull

How Far Should We Migrate Threads?



- If a thread must migrate, hopefully it can still have some data locality
  - e.g., different Hyper-Thread on same core, different core on same chip, etc.
- Linux models this through hierarchical "scheduling domains"
  - balance load at the granularity of these domains
- Related question: when is it good for two threads to be near each other?

Alternative Multiprocessor Scheduling Policies

- Affinity Scheduling
  - attempts to preserve cache locality, typically using distributed queues
  - implemented in the Linux O(1) (2.6-2.6.22) and CFS (2.6.3-now) schedulers
- Space Sharing
  - divide processors into groups; jobs wait until required # of CPUs are available
- Time Sharing: "Gang Scheduling" and "Co-Scheduling"
  - time slice such that all threads in a job always run at the same time
- Knowing about Spinlocks
  - kernel delays de-scheduling a thread if it is holding a lock
    - acquiring/releasing lock sets/clears a kernel-visible flag
- Process control/scheduler activations:
  - application adjusts its number of active threads to match # of CPUs given to it by the OS

Summary

- Memory Consistency Models
  - Be sure to use fences or explicit synchronization operations when ordering matters
    - don't synchronize through normal memory operations!
- Process scheduling for a parallel machine
  - goals: load balancing and processor affinity
  - Affinity scheduling often implemented with distributed runnable queues
    - steal work to balance load