Parallelism: Scheduling and Memory Consistency Models

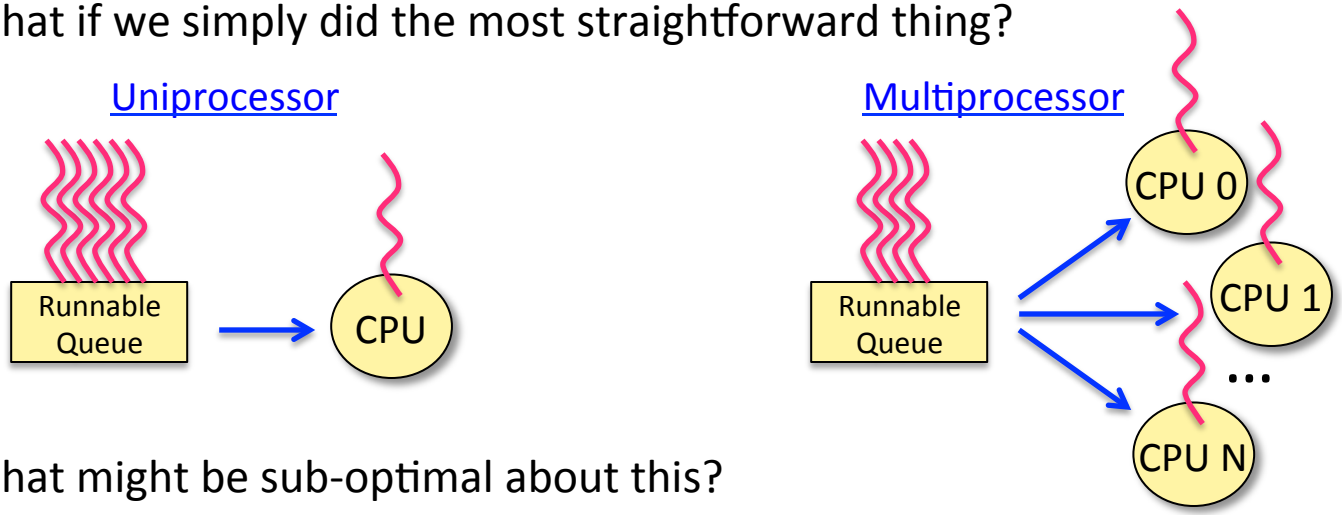
Todd C. Mowry & Dave Eckhardt

I. Process Scheduling Revisited

II. Memory Consistency Models

Process Scheduling Revisited: Scheduling on a Multiprocessor

- What if we simply did the most straightforward thing?



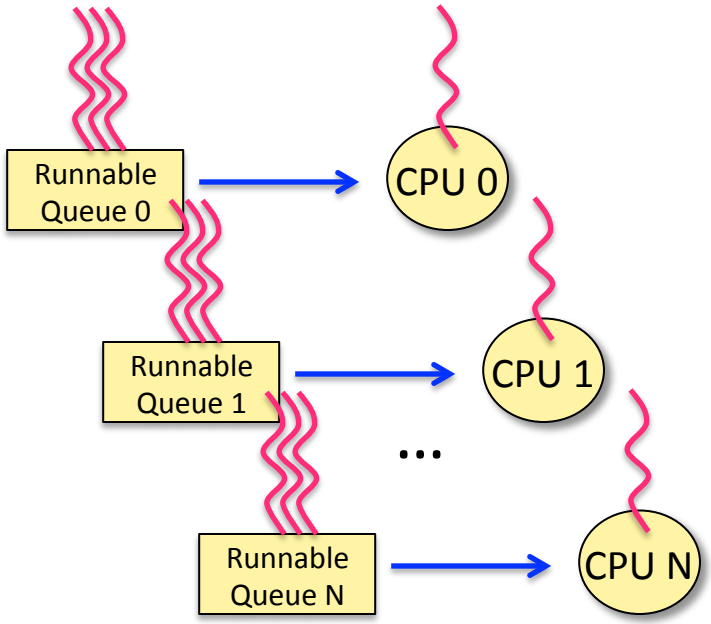
- What might be sub-optimal about this?
  - contention for the (centralized) run queue
  - migrating threads away from their data (disrupting data locality)
    - data in caches, data in nearby NUMA memories
  - disrupting communication locality between groups of threads
  - de-scheduling a thread holding 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

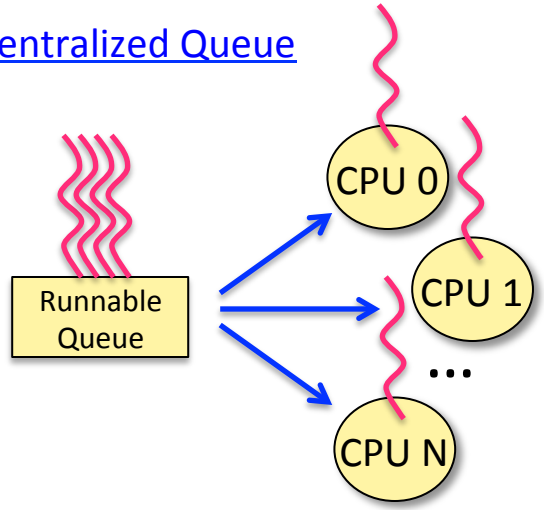
1. Load Balancing
   - try to distribute the work evenly across the processors
     - avoid having processors go idle, or waste time searching for work
2. Affinity
   - try to always restart a task on the same processor where it ran before
     - so that it can still find its data in the cache or in 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)

Distributed Queues

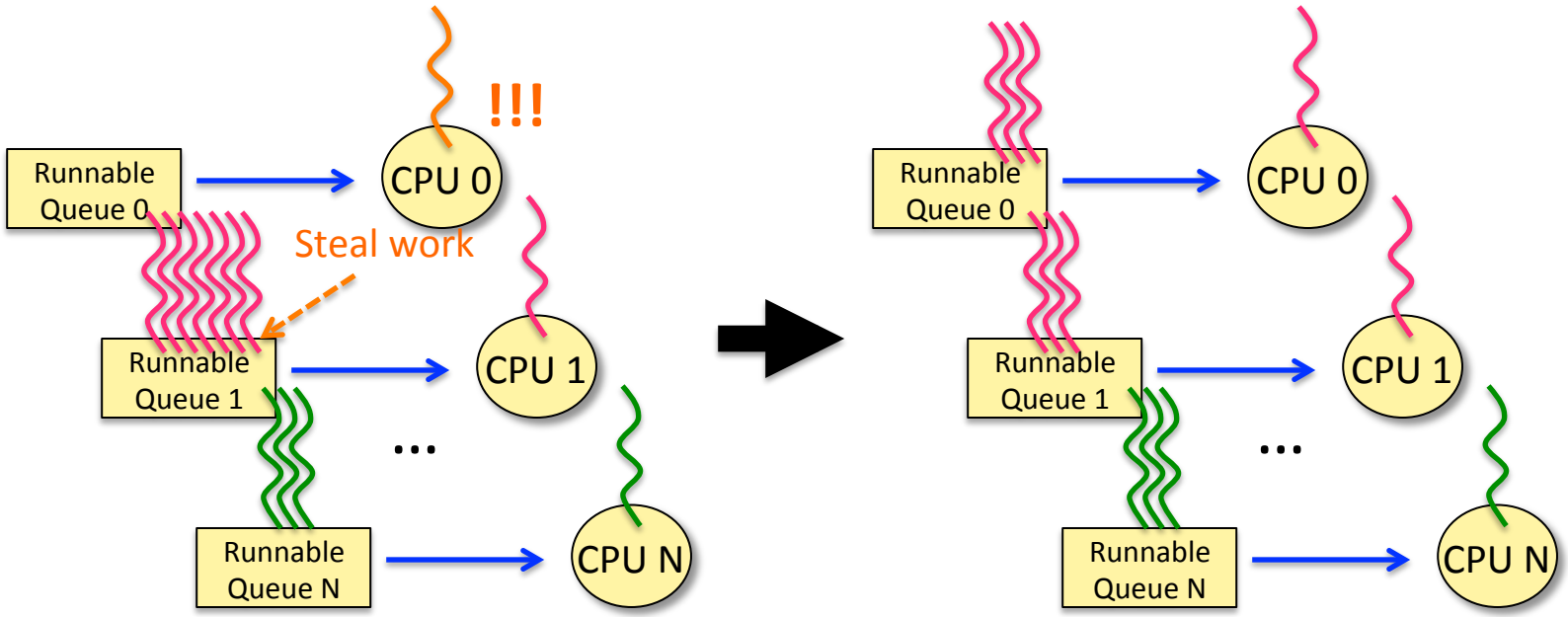


Centralized Queue



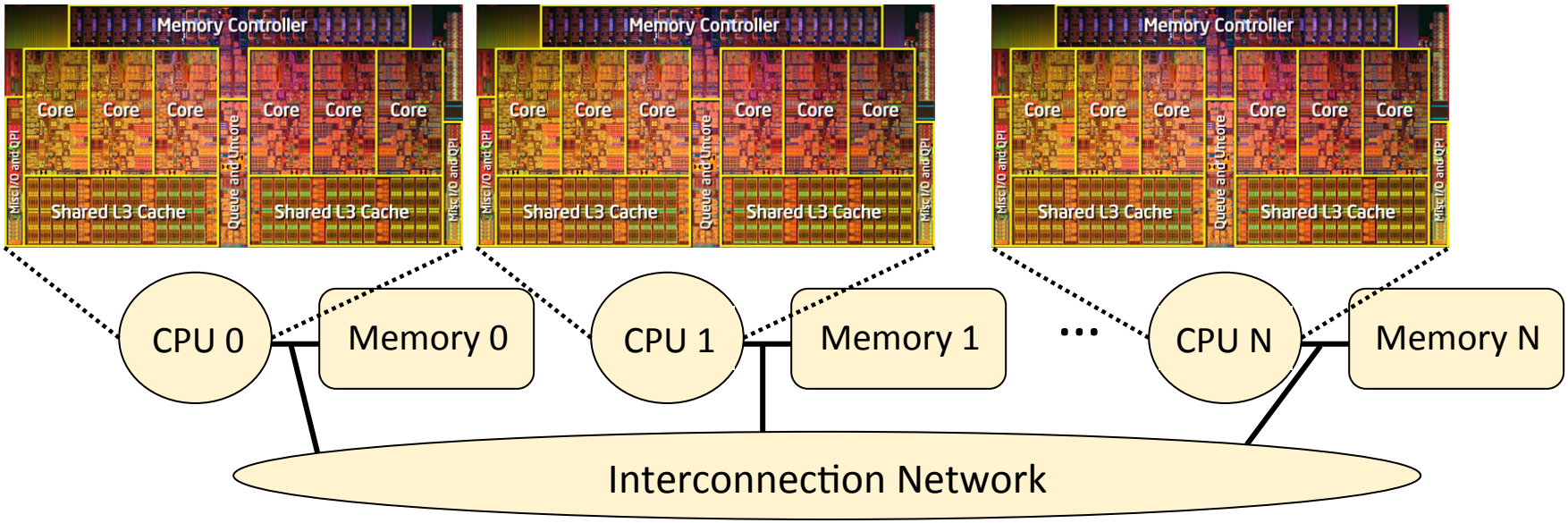
- Advantages of Distributed Queues?
  - easy to maintain affinity, since a blocked process stays in local queue
  - minimal locking contention to access queue
- But what about load balancing?
  - need to steal work from other queues sometimes

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

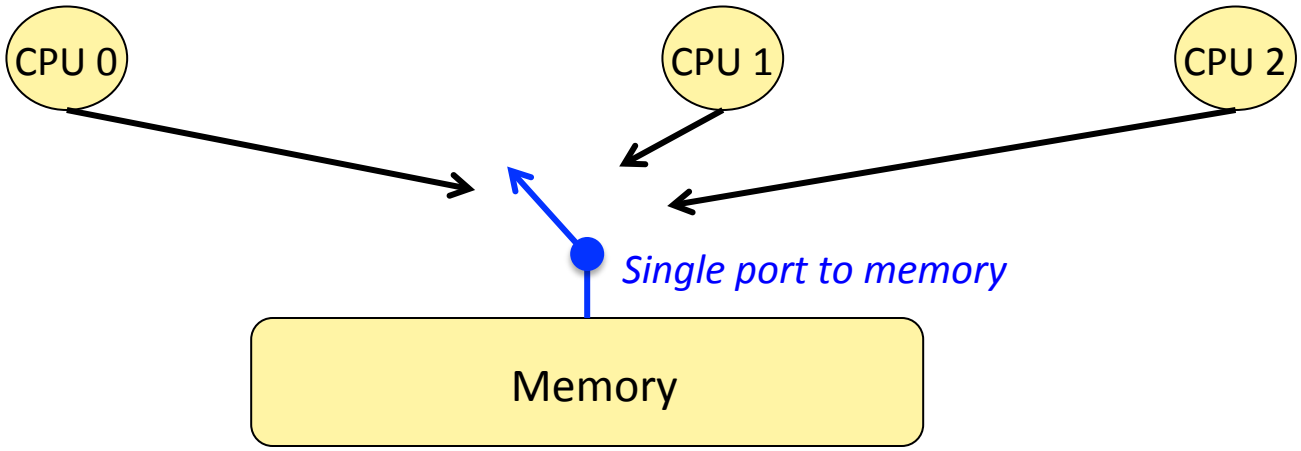
- Affinity Scheduling
  - attempts to preserve cache locality, typically using distributed queues
  - implemented in the Linux O(1) (2.6-2.6.22) and CFS (2.6.3-now) schedulers
- Space Sharing
  - divide processors into groups; jobs wait until required # of CPUs are available
- Time Sharing: "Gang Scheduling" 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

Part 2 of Memory Correctness: Memory Consistency Model

1. "Cache Coherence"
   - do all loads and stores to a given cache block behave correctly?

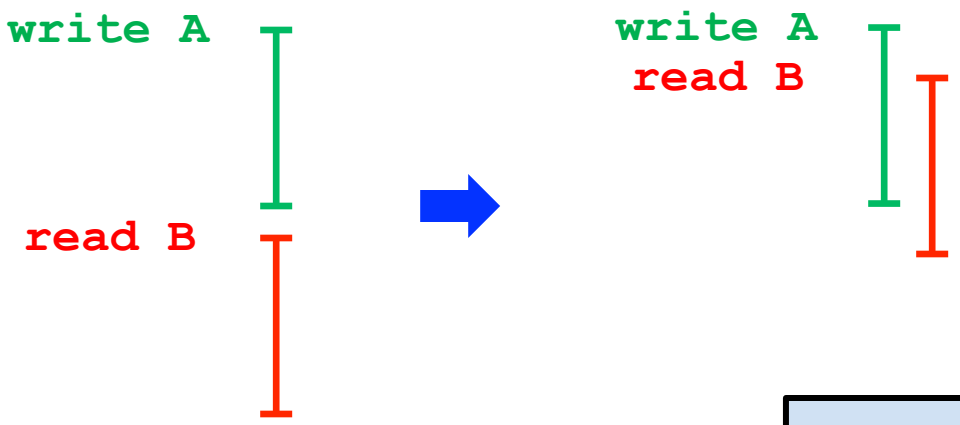
2. "Memory Consistency Model" (sometimes called "Memory Ordering")
   - do all loads and stores, even to separate cache blocks, behave correctly?

Recall: our intuition

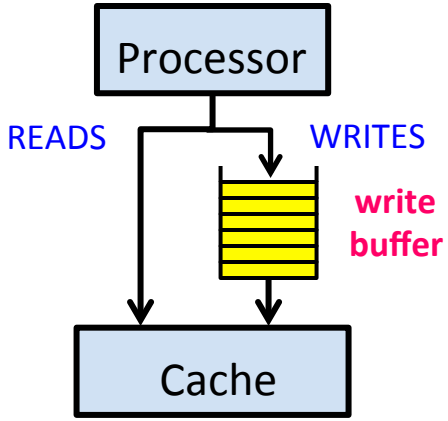


Hiding Memory Latency is Important for Performance

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



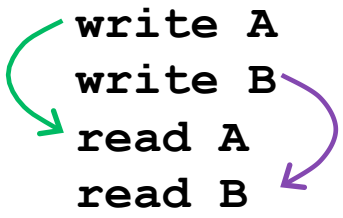
- Simple in uniprocessors:
  - add a write buffer

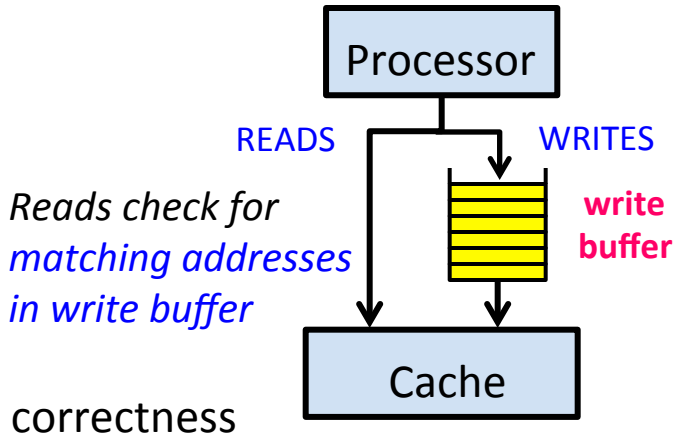


- Can affect correctness in multiprocessors

Uniprocessor Memory Model

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





- 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 Flag = 0)

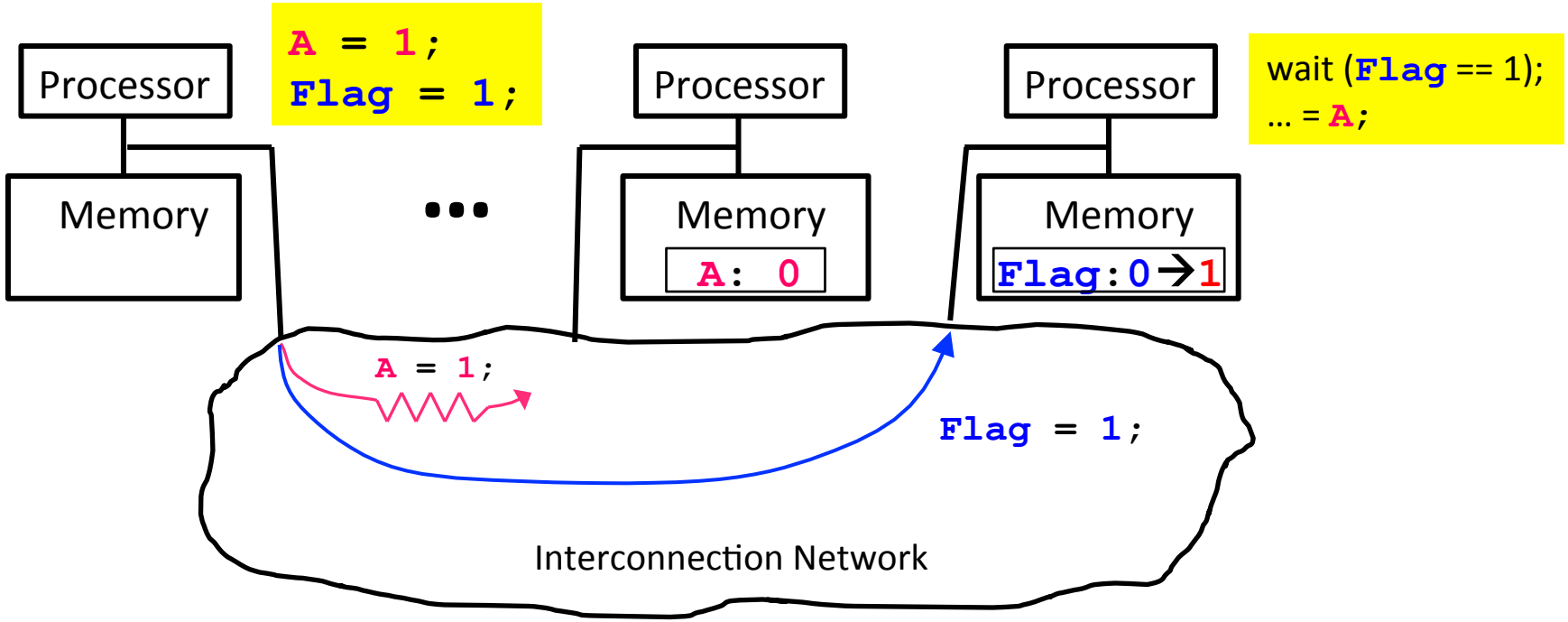
P1

```
A = 1;

Flag = 1;
```

P2

```
while (Flag != 1);

… = A;
```

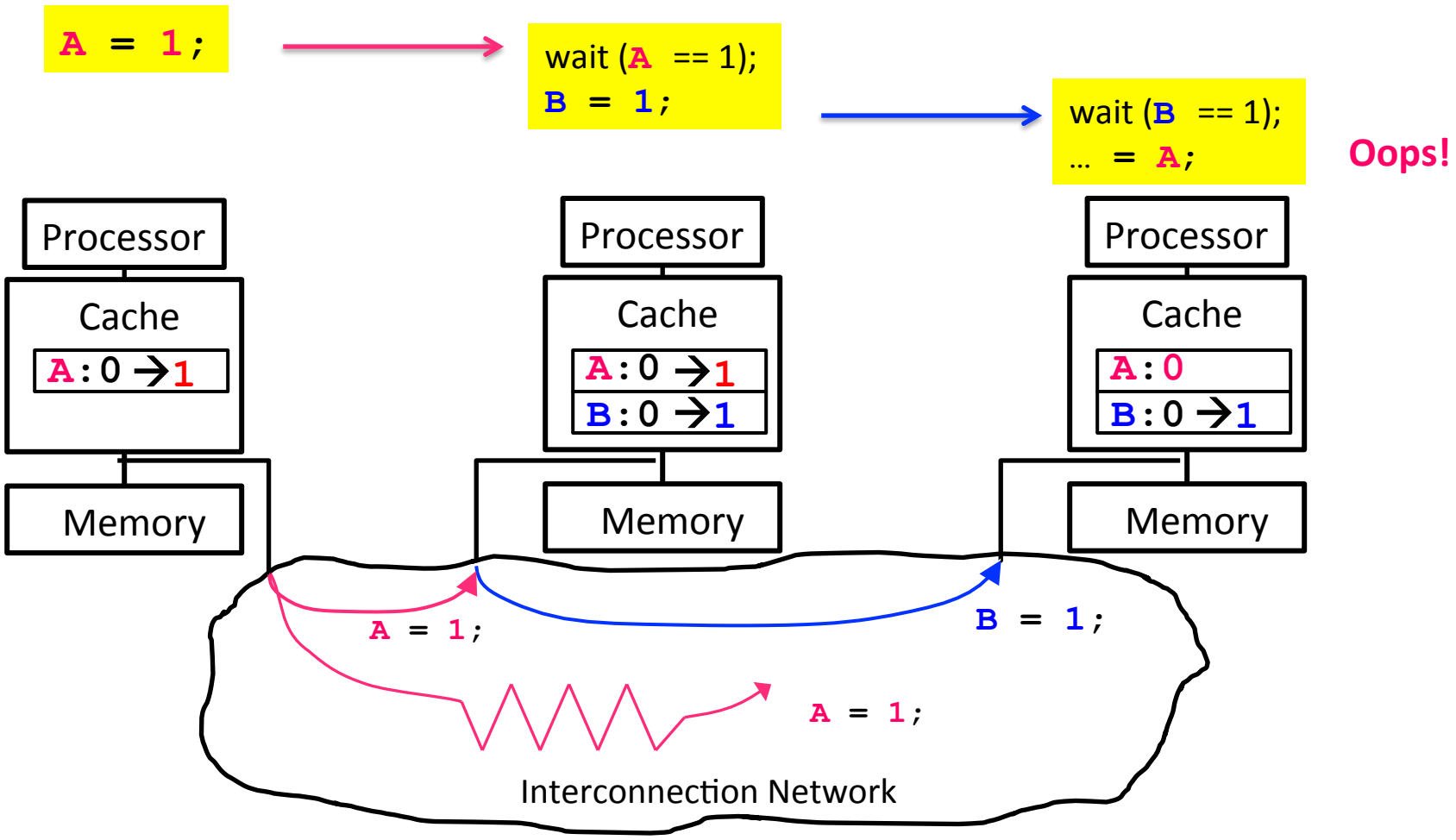
How Unsafe Reordering Can Happen



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

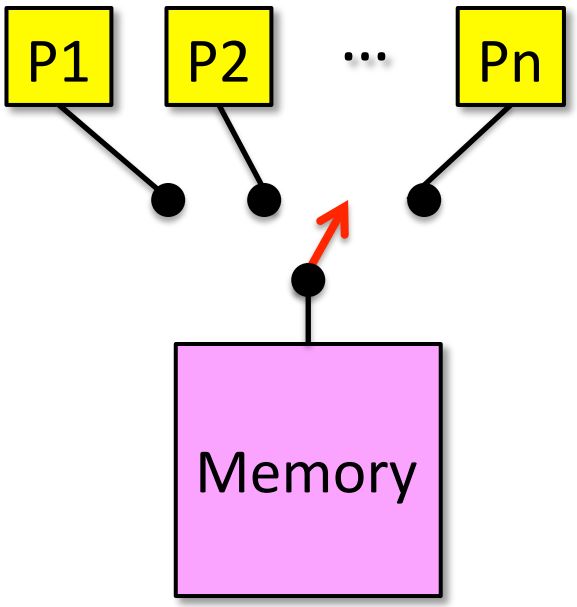
Caches Complicate Things More

- 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:

P1		P2	
`A = 1`	(a)		
`Flag = 1`	(b)	`x = Flag`	(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:
  - 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

Another Example with Sequential Consistency

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

P1		P2	
`A = 1`	(a)	`B = 1`	(c)
`x = B`	(b)	`y = A`	(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:
  - 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

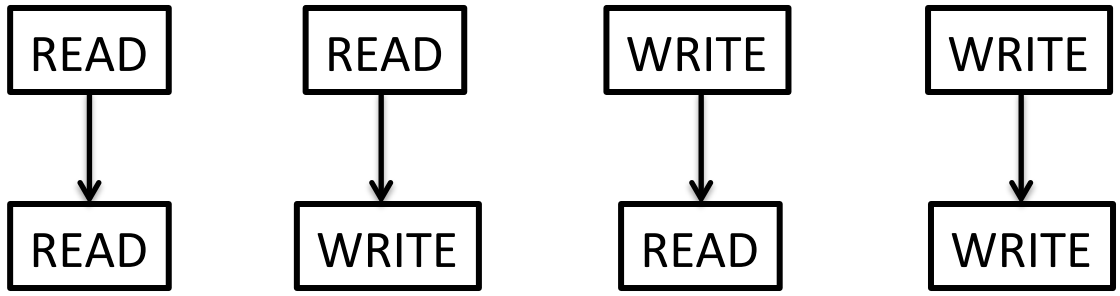
How to Guarantee SC

- Sufficient Conditions for SC (Dubois et al., 1987):
  - assumes general cache coherence (if we have caches):
    - writes to the same location are observed in same order by all P's
  - for each processor, delay issue of access until previous completes
    - a read completes when the return value is back
    - a write completes when new value is visible to all processors
      - for simplicity, we assume writes are atomic

- Important to note that these are not necessary conditions for maintaining SC

Summary for Sequential Consistency

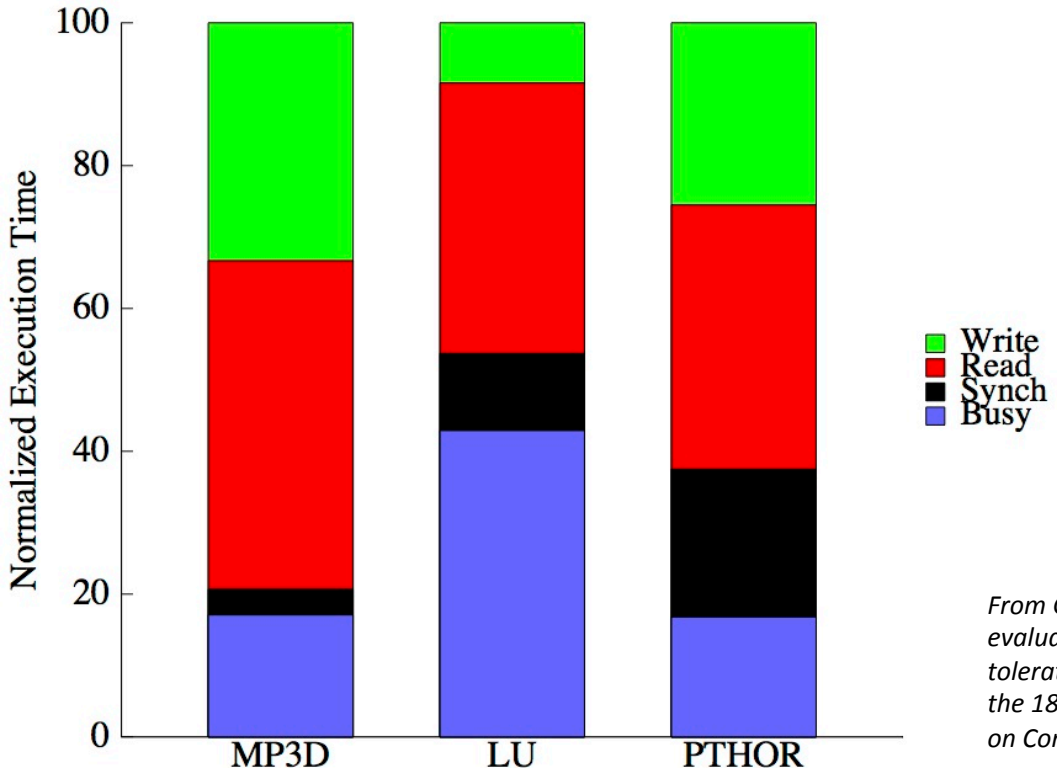
- Maintain order between shared accesses in each process



- Severely restricts common hardware and compiler optimizations

Performance of Sequential Consistency

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

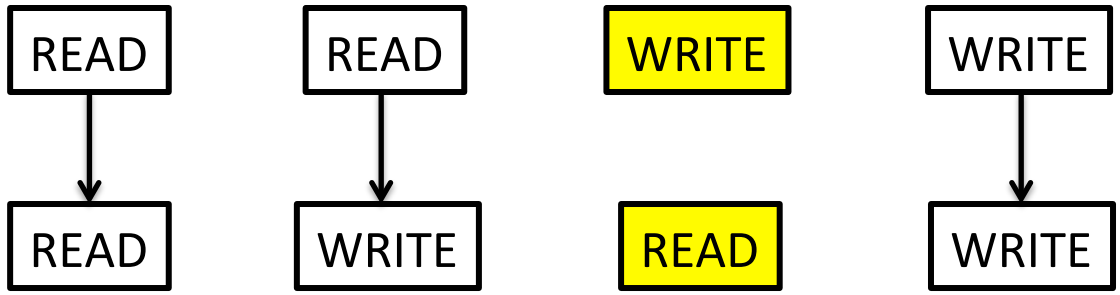


From Gupta et al, "Comparative evaluation of latency reducing and tolerating techniques." In Proceedings of the 18th annual International Symposium on Computer Architecture (ISCA '91)

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

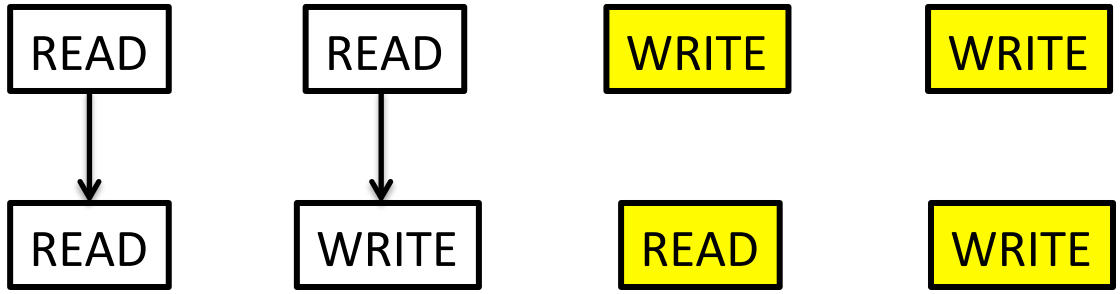
Alternatives to Sequential Consistency

- Relax constraints on memory order



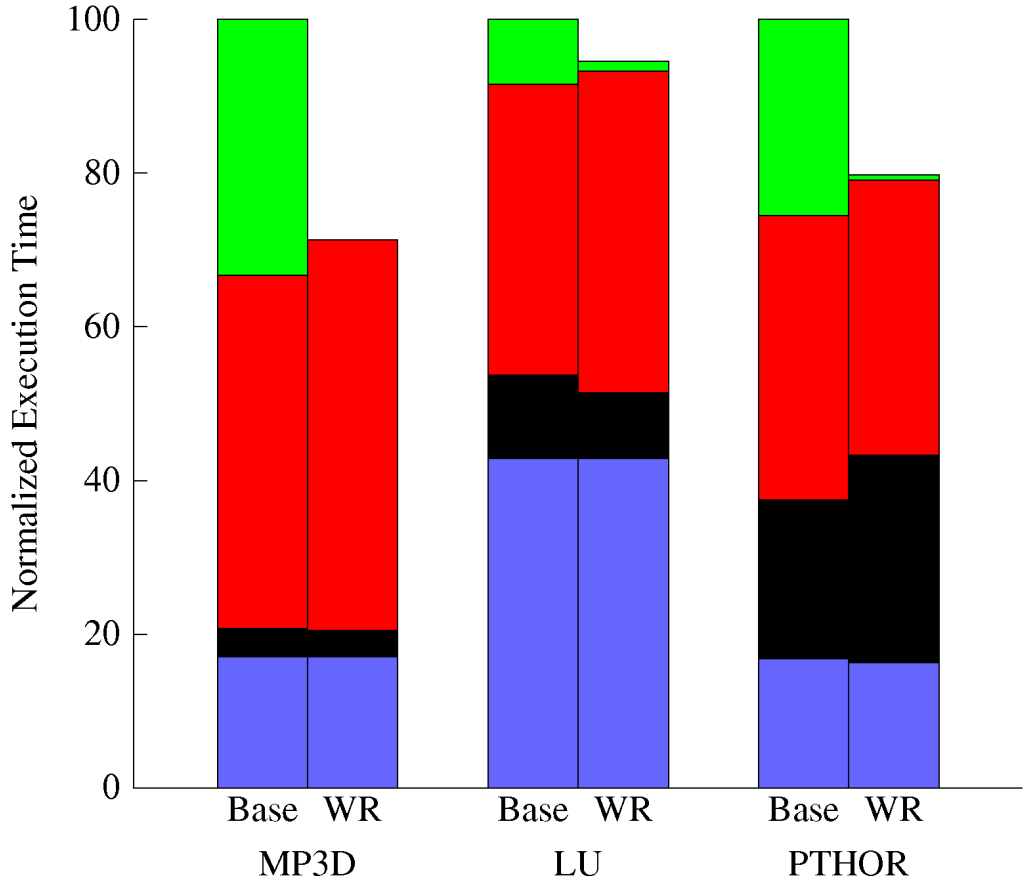
Total Store Ordering (TSO) (Similar to Intel)

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

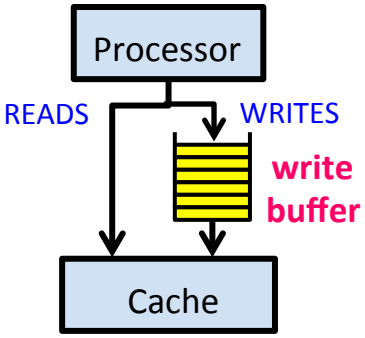


Partial Store Ordering (PSO)

Performance Impact of TSO vs. SC



"Base" = SC
"WR" = TSO

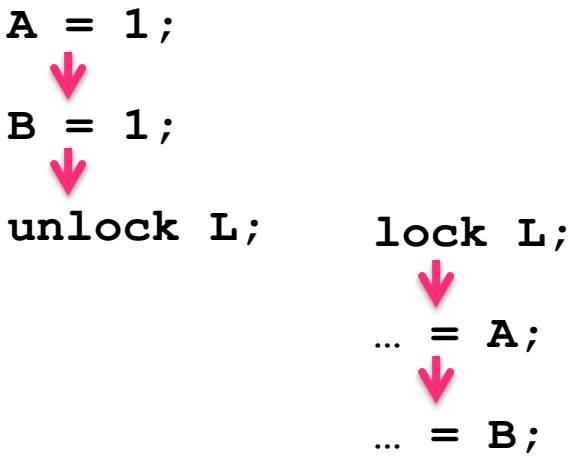


- Can use a write buffer
- Write latency is effectively hidden

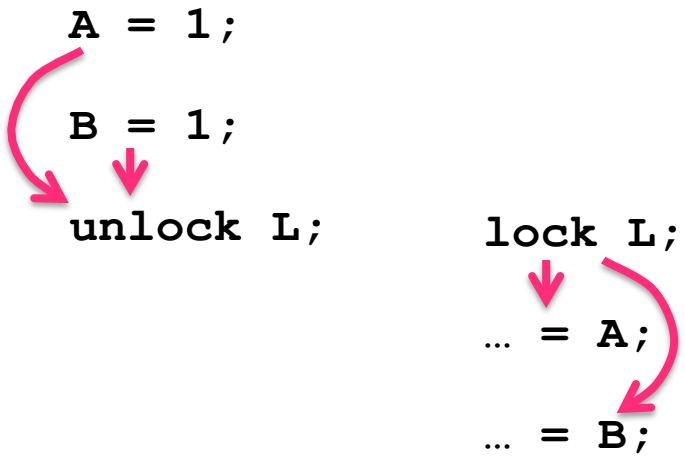
But Can Programs Live with Weaker Memory Orders?

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

Program Order



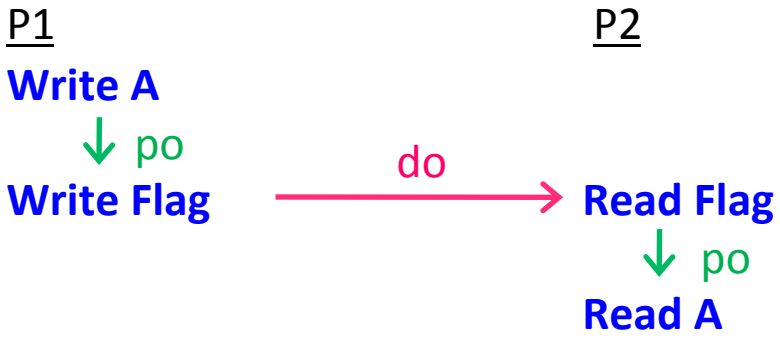
Sufficient Order



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

Identifying Data Races and Synchronization

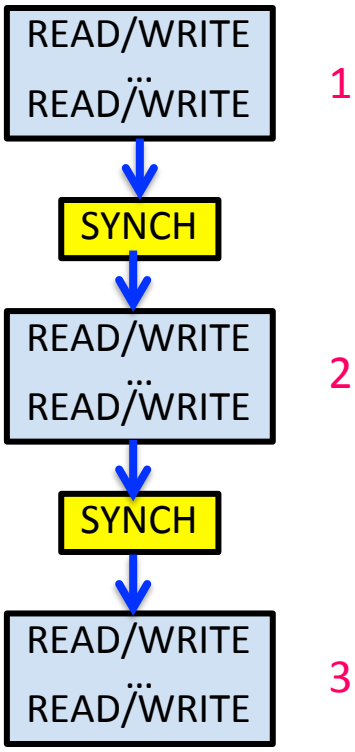
- Two accesses *conflict* if:
  - (i) access same location, and (ii) at least one is a write
- Order accesses by:
  - program order (po)
  - dependence order (do): op1 --> op2 if op2 reads op1



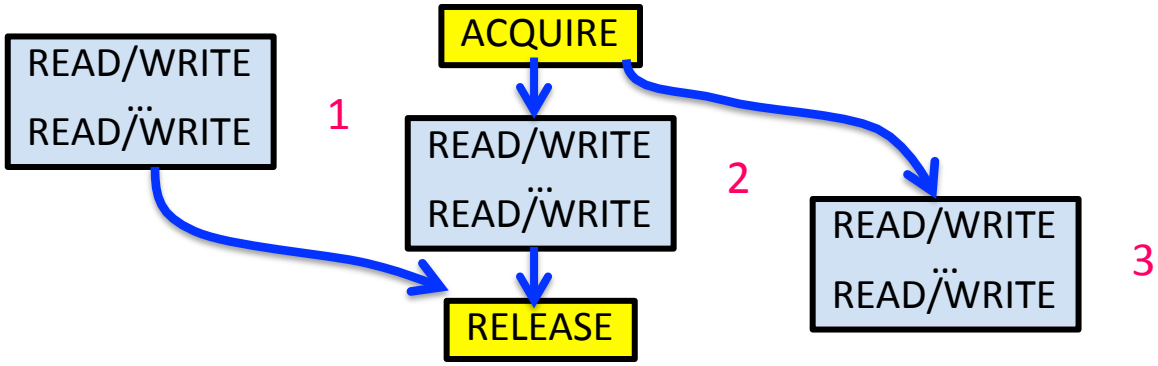
- Data Race:
  - two conflicting accesses on different processors
  - not ordered by intervening accesses
- Properly Synchronized Programs:
  - all synchronizations are explicitly identified
  - all data accesses are ordered through synchronization

Optimizations for Synchronized Programs

- Exploit information about synchronization



Weak Ordering (WO)



Release Consistency (RC)

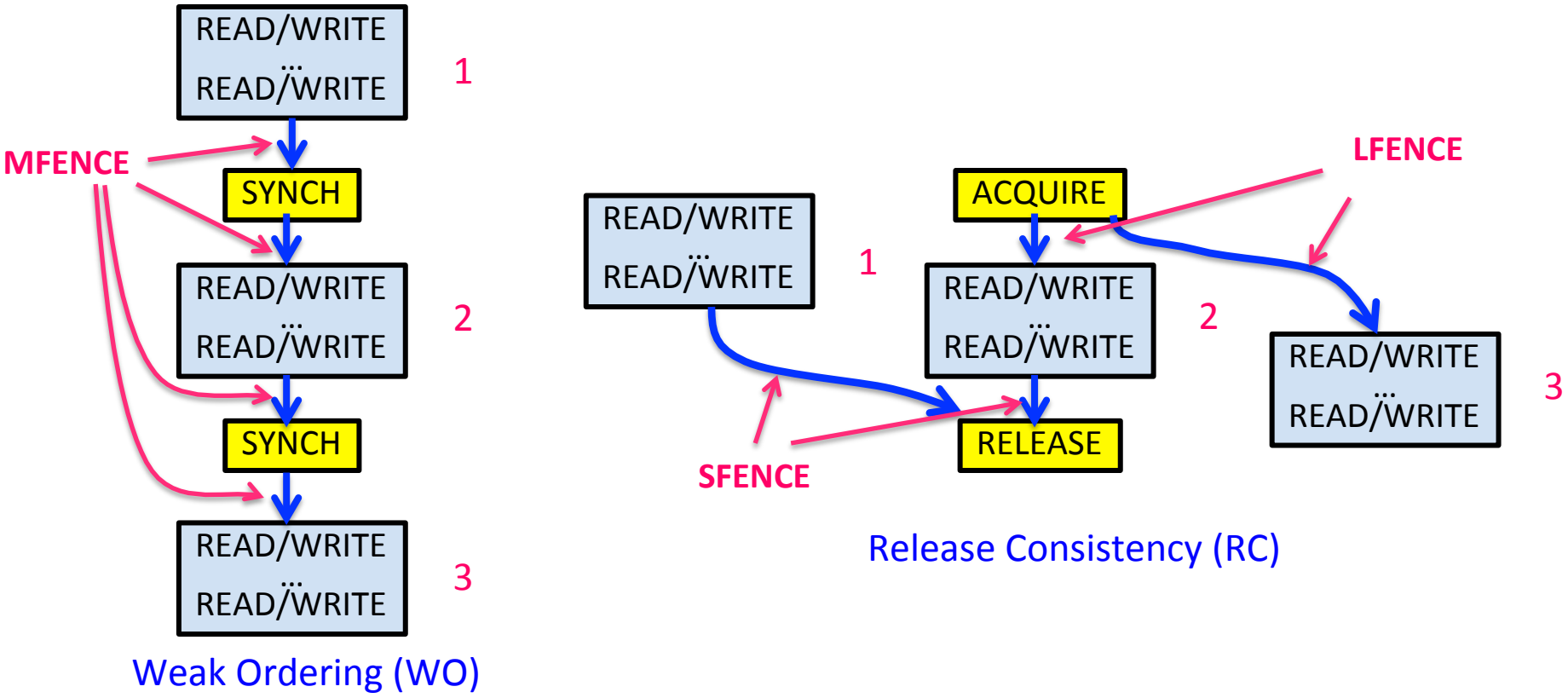
- properly synchronized programs yield SC results on RC systems

Intel's Fence Operations

- Fence operations:
  - enforce the ordering arcs seen on the previous slide
- 3 types of fence operations:
  - **SFENCE** (Store Fence)
    - wait until previous store operations (from same processor) complete
    - used when "releasing" access to data
  - **LFENCE** (Load Fence)
    - stalls future reads until earlier memory accesses (from same processor) complete
    - used when "acquiring" access to data
  - **MFENCE** (Memory Fence)
    - stalls until all previous reads and writes (from same processor) complete
    - corresponds to the "synch" case in weak ordering (i.e. both acquire and release)
- Explicit synchronization operations (xchg, etc.) have an implicit MFENCE

(For more details, see Section 8.2.5 of "Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1)

Using Fence Operations



- Note: fence operations typically have a non-trivial performance overhead
  - use when necessary, but don't use gratuitously

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);
```

Summary

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