上海科技大学
ShanghaiTech University

CS211
Advanced Computer Architecture

L16 Memory Consistency

Chundong Wang

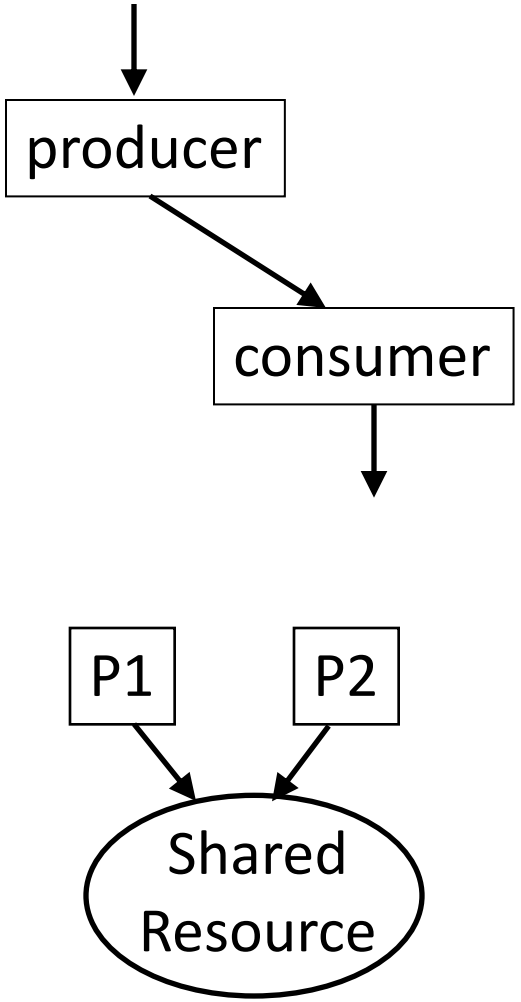
November 21st, 2025

Previously in CS211

- Cache coherence, making sure every store to memory is eventually visible to any load to the same memory address
- Cache line states: {M, S , I} or {M, E, S, I}, or even more
- Cache miss if tag not present, or line has wrong state
  - Write to a shared line is handled as a miss
- Snoopy coherence:
  - Broadcast updates and probe all cache tags on any miss of any processor, used to be bus connection, now often broadcast over point-to-point links
  - Lower latency, but consumes lots of bandwidth on both the communication bus and for probing the cache tags
- Directory coherence:
  - Structure keeps track of which caches can have copies of data, and only send messages/probes to those caches
  - Complicated to get right with all the possible overlapping cache transactions

Synchronization

The need for synchronization arises whenever there are concurrent processes in a system *(even in a uniprocessor system).*

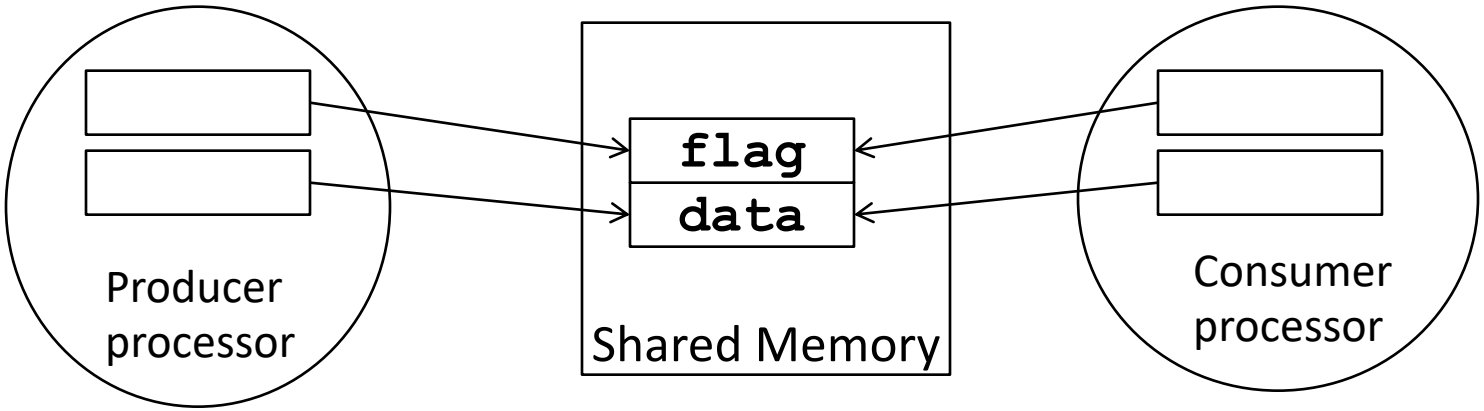


Two classes of synchronization:

- *Producer-Consumer:* A consumer process must wait until the producer process has produced data

- *Mutual Exclusion:* Ensure that only one process uses a resource at a given time

Simple Producer-Consumer Example



Initially **`flag=0, data=0`**

```
sw (data), 10
sw (flag), 0x01
```

```
spin: lw x1, (flag)
      beqz x1, spin
      lw x2, (data)
```

- Is this correct?
  - What value does x2 hold after both processors finish running this code?

Memory Consistency Model

- Sequential ISA only specifies that each processor sees its own memory operations in program order
- Memory consistency model describes what values can be returned by load instructions across multiple hardware threads

- *Coherence* describes the legal values a *single* memory address should return
- *Consistency* describes properties across *all* memory addresses
  - Order in which memory operations performed by one thread become visible to other threads
  - Coherence only guarantees that writes to address *X* will eventually propagate to other processors
  - Consistency deals with when writes to *X* propagate to other processors, relative to reads and writes to other addresses
  - Memory consistency defines the **allowed** behavior of loads and stores to different addresses in a parallel system

Memory Consistency

- Trailer:
  - Multiprocessors reorder memory operations in ***unintuitive*** and ***strange*** ways
  - This behavior is required for performance
  - Application programmers rarely see this behavior
  - Systems (OS and compiler) developers see it all the time

A memory consistency model is a contract between the hardware and software. The hardware promises to only reorder operations in ways allowed by the model, and in return, the software acknowledges that all such reorderings are possible and that is needs to account for them.

Memory operation ordering

- A program defines a sequence of loads and stores
  - The "program order" of the loads and stores

- Four types of memory operation orderings
  - $W_x \rightarrow R_y$: write to *x* must commit before subsequent read from *y*
  - $R_x \rightarrow R_y$: read from *x* must commit before subsequent read from *y*
  - $R_x \rightarrow W_y$: read to *x* must commit before subsequent write to *y*
  - $W_x \rightarrow W_y$: write to *x* must commit before subsequent write to *y*

"write must commit before subsequent read" means: When a write comes before a read in program order, the write must commit (its results are visible) by the time the read occurs.

Simple Producer-Consumer Example

Initially **flag=0, data=0**

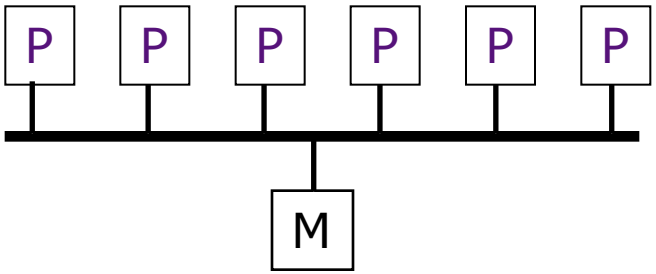
```
sw (data), 10
sw (flag), 0x01
```

```
spin: lw x1, (flag)
      beqz x1, spin
      lw x2, (data)
```

Can consumer read **flag=1** before **data** written by producer visible to consumer?

Sequential Consistency (SC)

A Memory Model



"A system is *sequentially consistent* if the result of any execution is the same as if the operations of all the processors were executed in **some** sequential order, and the operations of each individual processor appear in the order specified by the program"

Leslie Lamport (Turing Award, 2013)

Sequential Consistency = arbitrary *order-preserving interleaving* of memory references of sequential programs over a single shared memory

Simple Producer-Consumer Example

Initially **`flag=0, data=0`**

```
sw (data), 10
sw (flag), 0x01
```

```
spin: lw x1, (flag)
      beqz x1, spin
      lw x2, (data)
```

Dependencies from sequential ISA

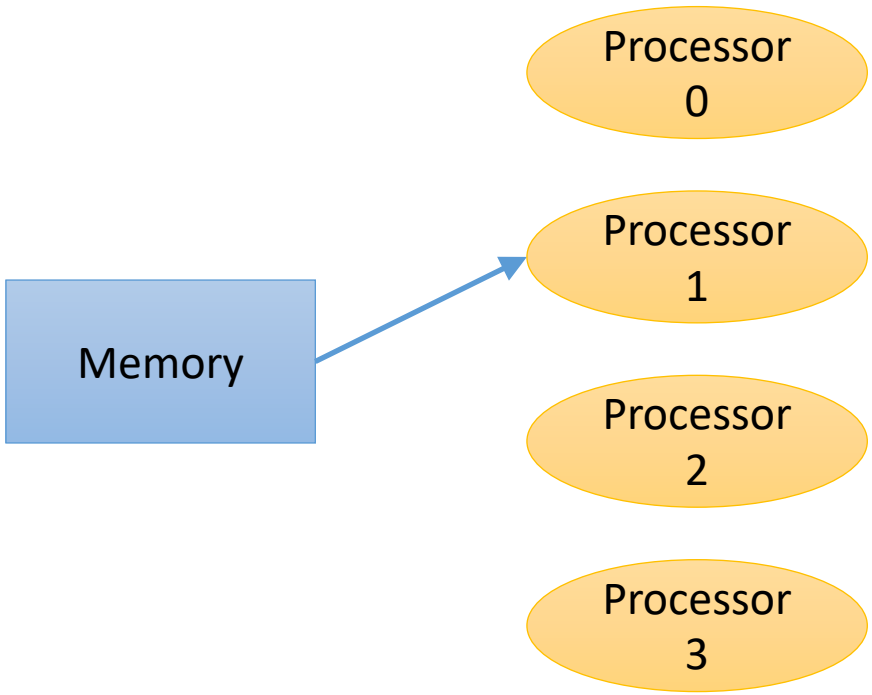
Dependencies added by sequentially consistent memory model

SC

- Each thread's operations happened in program order (*happened-before*)
- All operations were manipulating a single shared memory

Sequential consistency (switch metaphor)

- All processors issue loads and stores in program order
- Memory chooses a processor at random, performs a memory operation to completion, then chooses another processor, …




```
% A == 0          % B == 0
(1) A = 1        (3)B = 1
(2) x1 = B       (4)x2 = A
```

- (1)(2)(3)(4)
- (1)(3)(2)(4)
- (3)(1)(4)(2)
- …

Can we eventually have x1 == 0 && x2 == 0 w.r.t SC?

Most real machines are not SC

Architectural optimizations that are correct for uniprocessors often violate sequential consistency and result in a new memory model for multiprocessors

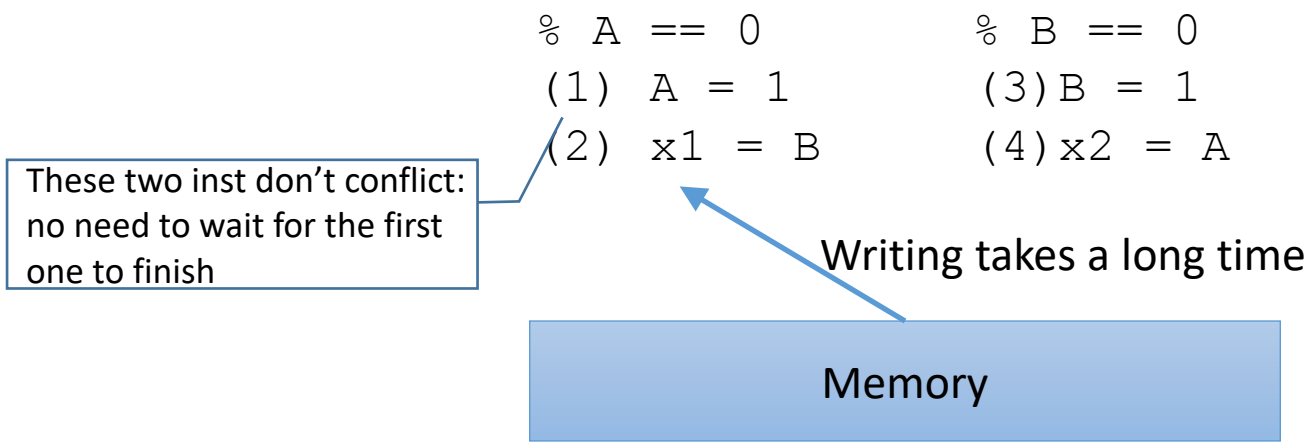
- Only a few commercial ISAs require SC
  - Neither x86 nor ARM nor RISC-V are SC
- Originally, architects developed uniprocessors with optimized memory systems (e.g., store buffer)
- When uniprocessors were lashed together to make multiprocessors, resulting machines were not SC
- Requiring SC would make simpler machines slower, or requires adding complex hardware to retain performance
- Architects/language designers/applications developers work hard to explain weak memory behavior
- Resulted in “weak” memory models with fewer guarantees

Consistency Models

- Sequential Consistency
  - All reads and writes in order
- Relaxed Consistency (one or more of the following)
  - Loads may be reordered after loads
    - e.g., PA-RISC, Power, Alpha
  - Loads may be reordered after stores
    - e.g., PA-RISC, Power, Alpha
  - Stores may be reordered after stores
    - e.g., PA-RISC, Power, Alpha, PSO
  - Stores may be reordered after loads
    - e.g., PA-RISC, Power, Alpha, PSO, TSO, x86
  - Other more esoteric characteristics
    - e.g., Alpha

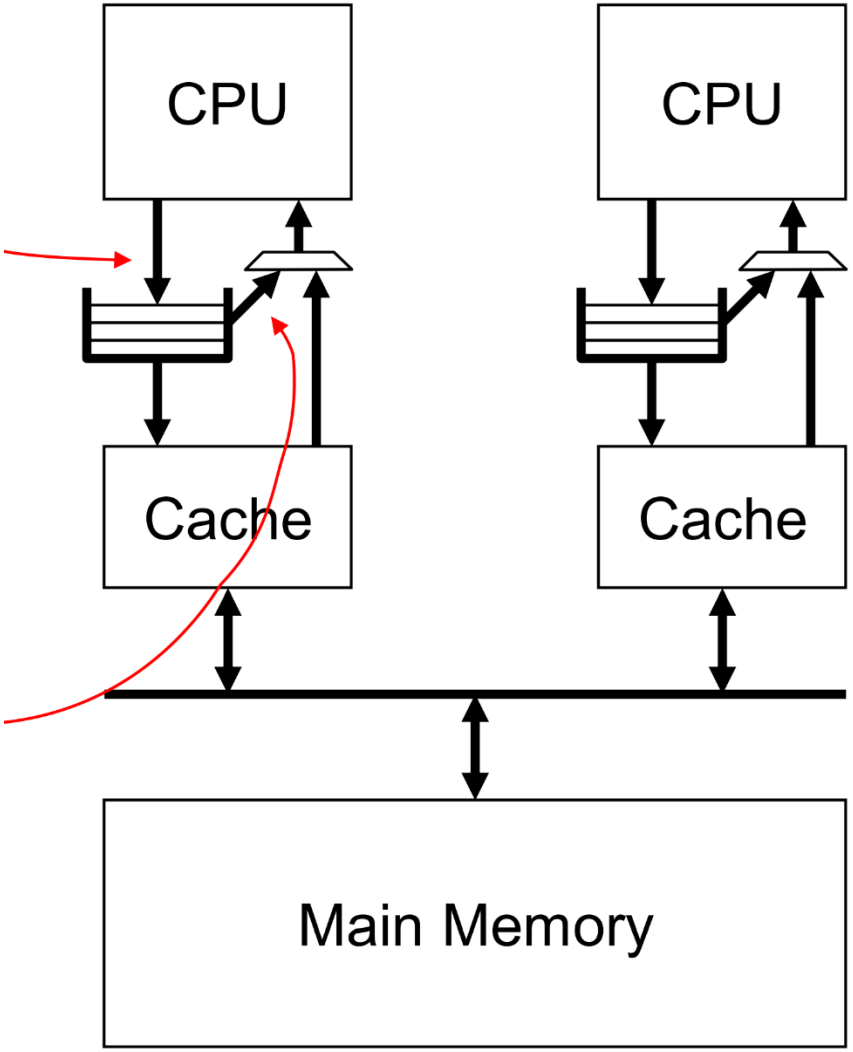
Motivation for relaxed consistency

- Issues with sequential consistency
  - Too conservative ordering requirements
  - Limits the aggressiveness of performance enhancement techniques



Store Buffers

- CPU can continue execution while earlier committed stores are still propagating through memory system
  - Processor can commit other instructions (including loads and stores) while first store is committing to memory
  - Committed store buffer can be combined with speculative store buffer in an out-of-order CPU
- Load optimizations:
  - Local loads can go ahead of buffered stores if to different address
  - Local loads can bypass value from earlier buffered store if to same address



Store Buffers: Who Cares?

- Performance improvement
- Every modern processor uses them
- Intel x86, ARM, SPARC
- Need a weaker memory model
  - TSO: Total Store Ordering
  - Slightly harder to reason about than SC
  - x86 uses an incompletely specified form of TSO

TSO: total store ordering

- TSO is the strongest memory model in common use
- Allows local buffering of stores by processor
- Reads by other processors cannot return new value of A until the write to A is observed by all processors

```
% A == 0          % B == 0
(1) A = 1        (3)B = 1
(2) x1 = B       (4)x2 = A
```

Possible Outcomes

x1	x2	SC	TSO
0	0	N	Y
0	1	Y	Y
1	0	Y	Y
1	1	Y	Y

Allowing reads to move ahead of writes

- Four types of memory operation orderings
  - ~~$W_x \rightarrow R_y$: write to *x* must commit before subsequent read from *y*~~
  - $R_x \rightarrow R_y$: read from *x* must commit before subsequent read from *y*
  - $R_x \rightarrow W_y$: read to *x* must commit before subsequent write to *y*
  - $W_x \rightarrow W_y$: write to *x* must commit before subsequent write to *y*

- In TSO, only $W_x \rightarrow R_y$ order is relaxed. The $W_x \rightarrow W_y$ constraint still exists.
  - Writes by the same thread are not reordered (they occur in program order)

Relaxed memory consistency issues arise from the optimization of reordering memory operations. (Consistency is unrelated to whether or not caches exist in the system.)

Strong versus Weak Memory Consistency Models

- Stronger models provide more guarantees on ordering of loads and stores across different hardware threads
  - Easier ISA-level programming model
  - Can require more hardware to ensure orderings (e.g., MIPS R10K was SC, with hardware to speculate on load/stores and squash when ordering violations detected across cores)
- Weaker models provide fewer guarantees
  - Much more complex ISA-level programming model
    - Extremely difficult to understand, even for experts
  - Simpler to achieve high performance, as weaker models allow many reorderings to be exposed to software
  - Additional instructions (fences) are provided to allow software to specify which orderings are required

Fences in Producer-Consumer Example

Initially **flag=0, data=0**

```
sw (data), 10
fence w,w //Write-write fence
sw (flag), 0x01
```

```
spin: lw x1, (flag)
      beqz x1, spin
      fence r,r // Read-read fence
      lw x2, (data)
```

Partial Store Ordering (PSO)

~~$W_x \rightarrow W_y$: write to *x* must commit before subsequent write to *y*~~

(Consumer may observe change to **flag** before change to **data** w/o fence)

Why might it be useful to allow more aggressive memory operation reorderings?

- ~~$W_x \rightarrow W_y$~~: processor might reorder write operations in a write buffer (e.g., one is a cache miss while the other is a hit)

- ~~$R_x \rightarrow W_y, R_x \rightarrow R_y$~~: processor might reorder independent instructions in an instruction stream (out-of-order execution)

- Motivation is increased performance
  - Overlap multiple reads and writes in the memory system
  - Execute reads as early as possible and writes as late as possible to hide memory latency

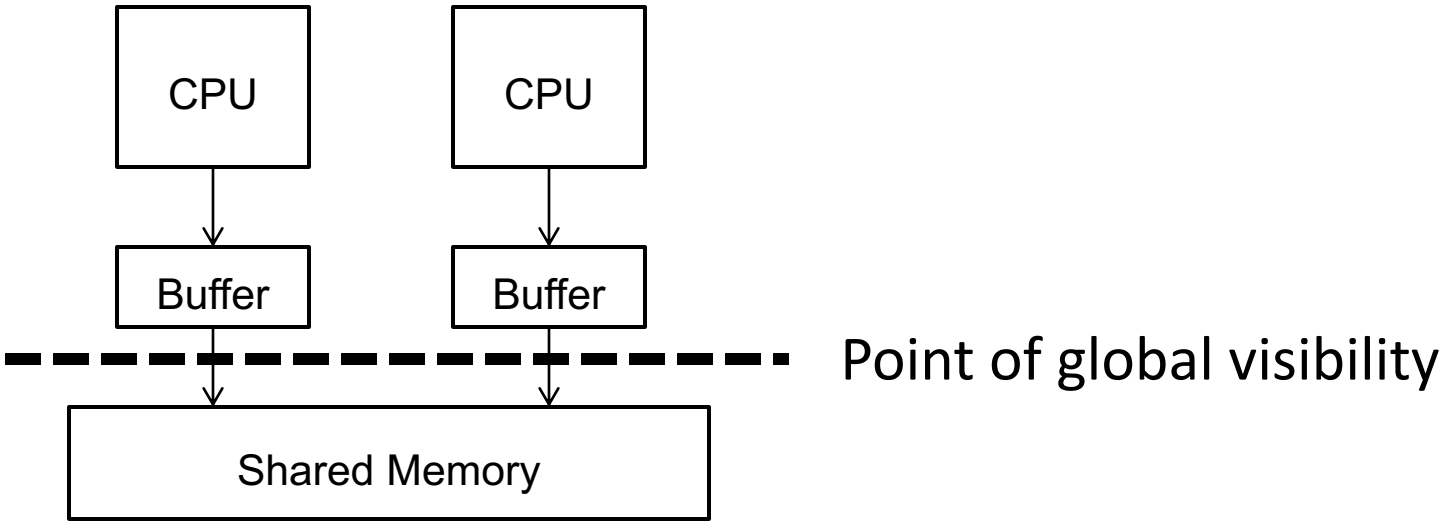
Synchronization to the Rescue

- Memory reordering seems like a nightmare (it is!)
- Every architecture provides synchronization primitives to make memory ordering stricter
- Fence (memory barrier) instructions prevent reorderings, but are expensive
  - All memory operations complete before any memory operation after it can begin
- Other synchronization primitives (per address):
  - read-modify-write/compare-and-swap, transactional memory, …

Range of Memory Consistency Models

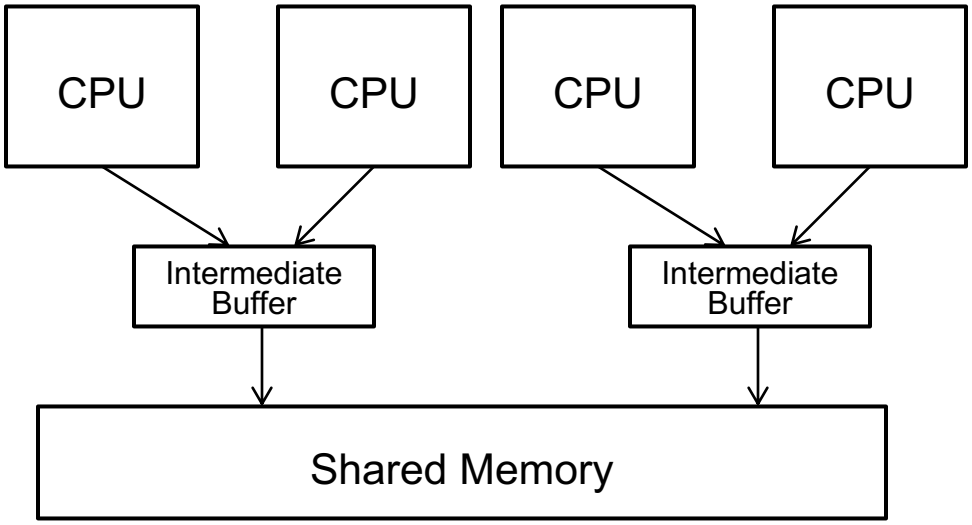
- SC “Sequential Consistency”
  - MIPS R10K
- TSO “Total Store Ordering”
  - *processor can see its own writes before others do (store buffer)*
  - IBM-370 TSO, x86 TSO, SPARC TSO (default), RISC-V RVTSO (optional)
- Weak, multi-copy-atomic memory models
  - *all processors see writes by another processor in same order*
  - Revised ARMv8 memory model
  - RISC-V RVWMO, baseline weak memory model for RISC-V
- Weak, non-multi-copy-atomic memory models
  - *processors can see another’s writes in different orders*
  - ARMv7, original ARMv8
  - IBM POWER
  - Digital Alpha (extremely weak MCM)
  - Recent consensus is that these appear to be too weak for general-purpose processors

Multi-Copy Atomic models



- Each hardware thread must view its own memory operations in program order, but can buffer these locally and reorder accesses around the buffer
- But once a local store is made visible to one other hardware thread in system, all other hardware threads must also be able to observe it (this is what is meant by “atomic”)

Hierarchical Shared Buffering



- Common in large systems to have shared intermediate buffers on path between CPUs and global memory
- Potential optimization is to allow some CPUs see some writes by a CPU before other CPUs
- Shared memory stores are **not** seen to happen atomically by other threads (non multi-copy atomic)

Non-Multi-Copy Atomic

```
          Initially X == Y == 0

P1:           P2:           P3:
li x1, 1      lw x1, X      lw x1, Y
sw x1, X      sw x1, Y      fence r,r
                            lw x2, X
```

Can P3.x1 = 1, and P3.x2 = 0 ?

- In general, Non-MCA is very difficult to reason about
- Software in one thread cannot assume all data it sees is visible to other threads, so how to share data structures?
- Adding local fences to require ordering of each thread's accesses is insufficient – need a more global memory barrier to ensure all writes are made visible

Conflicting data accesses

- Two memory accesses by different processors conflict if
  - They access to the same memory location
  - At least one is a write
  - Unordered by synchronization operations

- Unsynchronized program
  - Conflicting accesses not ordered by synchronization (e.g., a fence, operation with release/acquire semantics, barrier, etc.)
  - Unsynchronized programs contain data races: the output of the program depends on relative speed of processors (non-deterministic program results)

Synchronized programs

- Synchronized programs yield SC results on non-SC systems
  - Synchronized programs are data-race-free
- If there are no data races, reordering behavior doesn't matter
  - Accesses are ordered by synchronization, and synchronization forces sequential consistency
- In practice, most programs you encounter will be synchronized (via locks, barriers, etc. implemented in synchronization libraries)
  - Very few programmers do programming that relies on SC
  - Rather than via ad-hoc reads/writes to shared variables like in the example programs

Relaxed Memory Models

- Motivation
  - To obtain higher performance by allowing reordering of memory operations (reordering is not allowed by sequential consistency)
- Not all dependencies assumed by SC are supported, and software has to explicitly insert additional dependencies where needed
- Which dependencies are dropped depends on the particular memory model
  - IBM370, TSO, PSO, WO, PC, Alpha, RMO, …
  - Some ISAs allow several memory models, some machines have switchable memory models
- How to introduce needed dependencies varies by system
  - Explicit FENCE instructions (sometimes called sync or memory barrier instructions)
  - Implicit effects of atomic memory instructions

Do not forget Compiler and Language!

```
//Thread 1          //Thread 2
X = 0               X = 0
For i in (1:100):
        X = 1
        Print X
```

111111111111…

111111011111…

```
//Thread 1          //Thread 2
X = 1               X = 0
For i in (1:100):
        Print X
```

111111111111…

111111000000…

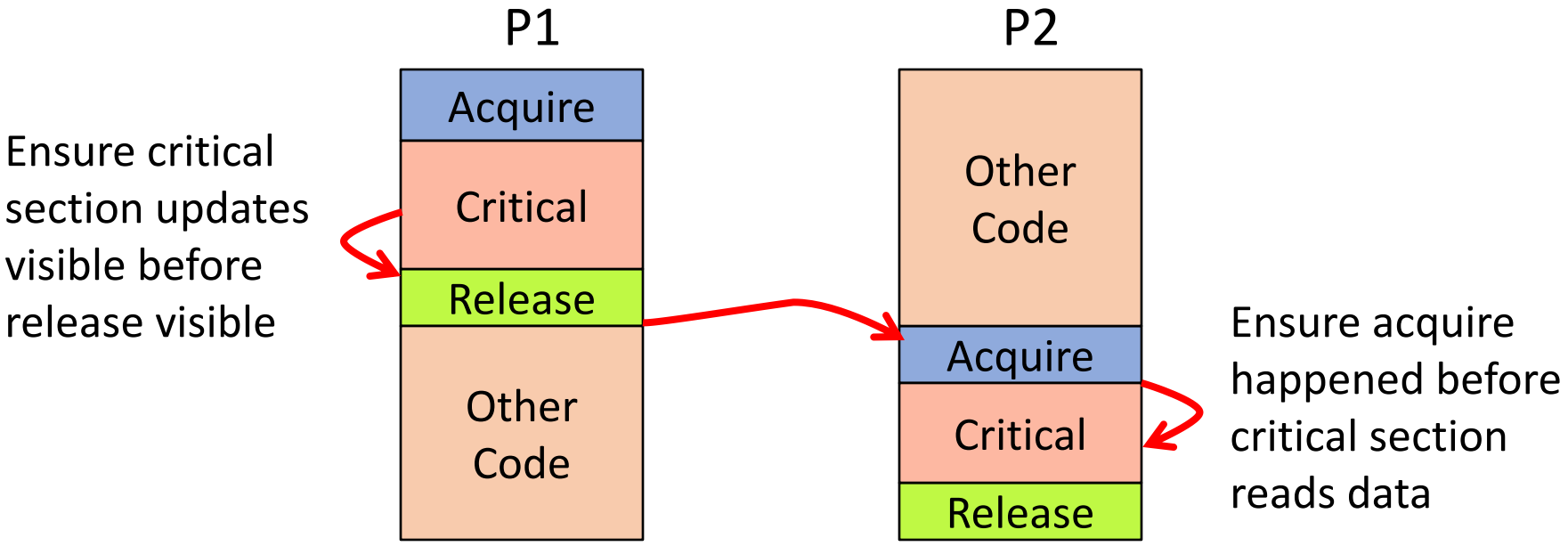
- Compiler can reorder/remove memory operations:
  - Instruction scheduling, move loads before stores if to different address
  - Register allocation, cache load value in register, don't check memory
- Prohibiting these optimizations would result in very poor performance

Language-Level Memory Models

- Programming languages have memory models too
- Hide details of each ISA's memory model underneath language standard
  - c.f. C function declarations versus ISA-specific subroutine linkage convention
- Language memory models: C/C++, Java
  - Modern (C11, C++11) and not-so-modern (Java 5) languages guarantee sequential consistency for data-race-free programs ("SC for DRF")
  - Compilers will insert the necessary synchronization to cope with the hardware memory model
- Describe legal behaviors of threaded code in each language and what optimizations are legal for compiler to make
- E.g., C11/C++11: **`atomic_load(memory_order_seq_cst)`** maps to RISC-V **`fence rw,rw; lw; fence r,rw`**

Release Consistency *[Garachorloo 1990]*

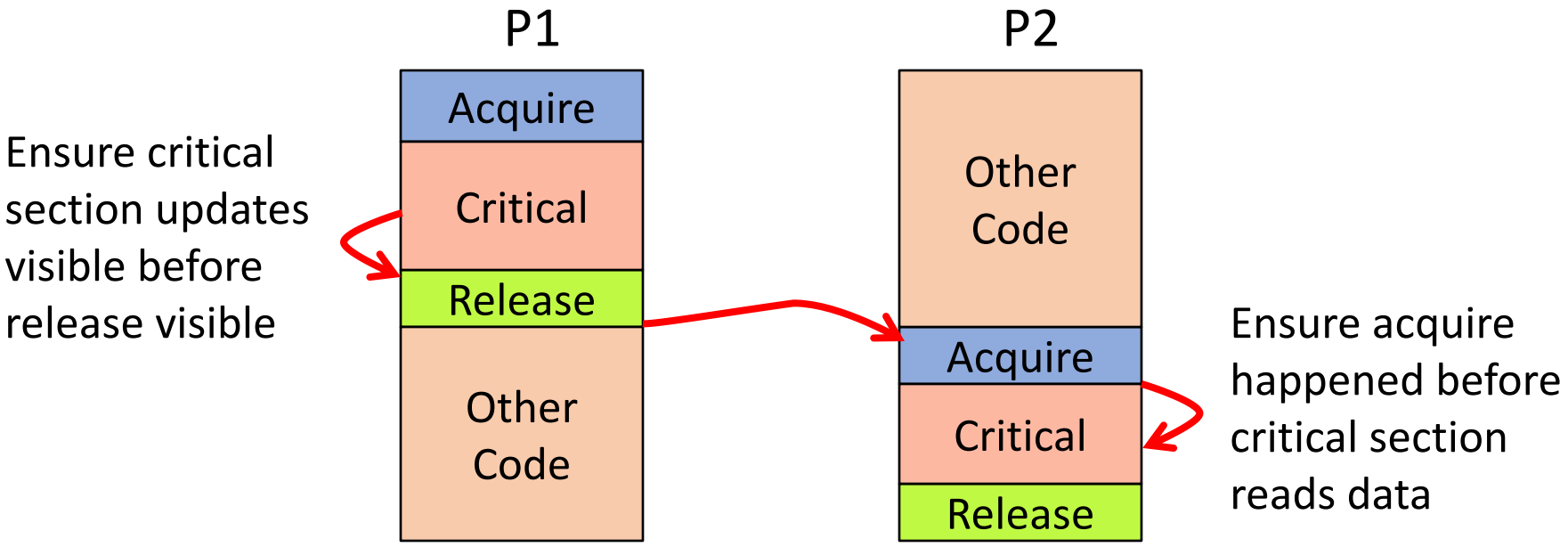
- Observation that consistency only matters when processes communicate data
- Only need to have consistent view when one process shares its updates to other processes
- Other processes only need to ensure they receive updates after they acquire access to shared data



Release Consistency *[Garachorloo 1990]*

- Obs
  co
- Or
  up
- Ot
  acquire access to shared data

Release consistency is essentially the same as weak consistency, but synchronization accesses must only be processor consistent with respect to each other. Synchronization operations are broken down into *acquire* and *release* operations. All pending acquires (e.g., a lock operation) must be done before a release (e.g., an unlock operation) is done. Local dependencies within the same processor must still be respected.



Release Consistency Adopted

- Only care about inter-processor memory ordering at thread synchronization points, not in between
- Can treat all synchronization instructions as the only ordering points

- Memory model for C/C++ and Java uses release consistency
- Programmer has to identify synchronization operations, and if all data accesses are protected by synchronization, appears like SC to programmer

- ARMv8 and RISC-V ISA adopt release consistency semantics on atomic memory operations (AMOs)
  - AMOs, such as compare-and-swap, fetch-and-add, etc. can be used to implement lock- and wait-free algorithms and data structures
  - Lock-free algorithms are supposed to allow an arbitrary number of threads to share a resource without the need for serial execution on a lock

Conclusion

- Sequential Consistency
- Relaxed Consistency

Acknowledgements

- These slides contain materials developed and copyright by:
  - Prof. Krste Asanovic (UC Berkeley)
  - Prof. Daniel Sanchez (MIT)
  - Prof. Kayvon Fatahalian (Stanford)
  - Prof. Kunle Olukotun (Stanford)
  - Prof. James Bornholt (UT Austin)