CS 152 Computer Architecture and Engineering
CS252 Graduate Computer Architecture

Lecture 19 Memory Consistency Models

Chris Fletcher
Electrical Engineering and Computer Sciences
University of California at Berkeley

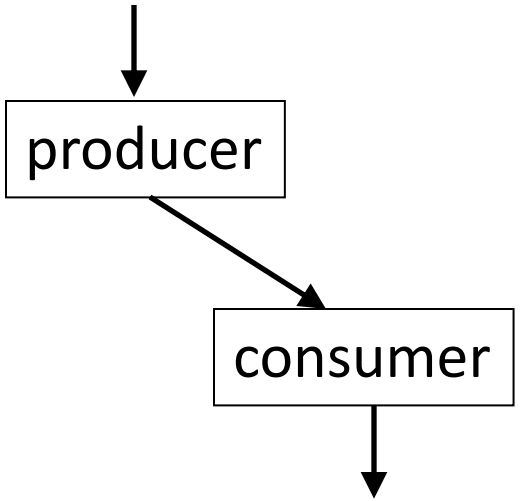
https://cwfletcher.github.io/
http://inst.eecs.berkeley.edu/~cs152

Last Time in Lecture 18

- Cache coherence, making sure every store to memory is eventually visible to any load to same memory address
- Cache line states: M,S,I or M,E,S,I
- 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 iinks
  - 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

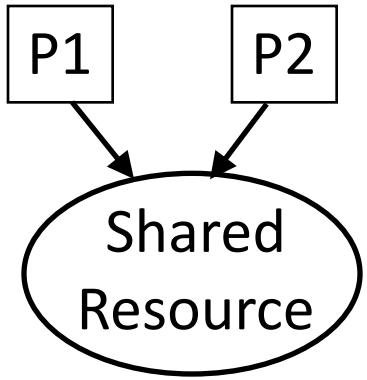
The need for synchronization arises whenever there are concurrent processes in a system *(even in a uniprocessor system)*.



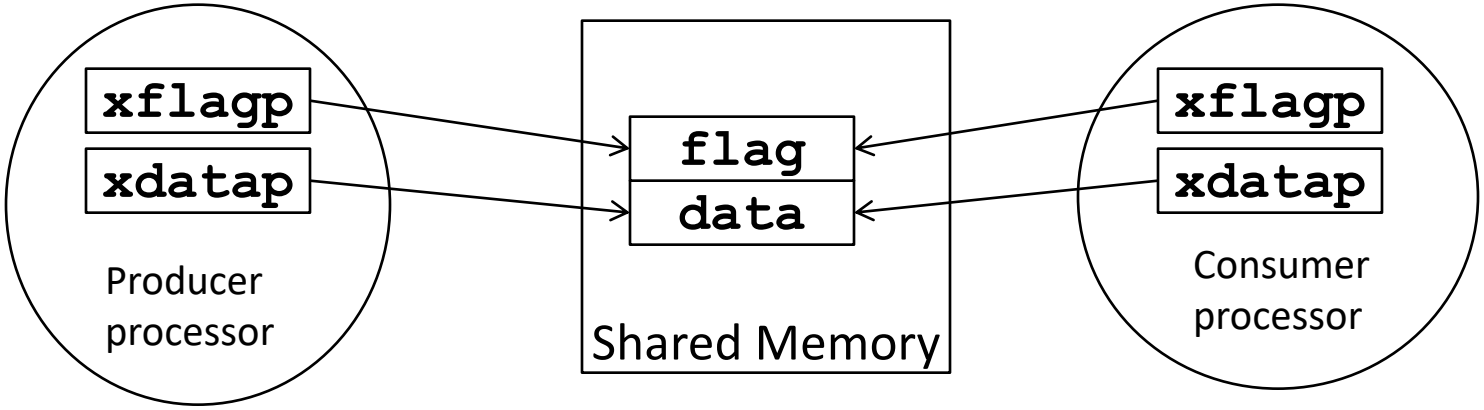
Two classes of synchronization:

- *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`**

```
sw xdata, (xdatap)          spin: lw xflag, (xflagp)
li xflag, 1                       beqz xflag, spin
sw xflag, (xflagp)                lw xdata, (xdatap)
```

Is this correct?

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

Simple Producer-Consumer Example

Initially **flag=0**

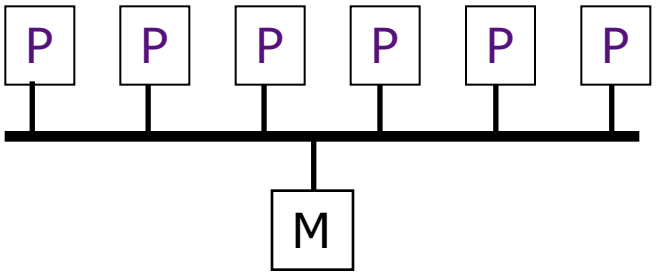
```
sw xdata, (xdatap)
li xflag, 1
sw xflag, (xflagp)
```

```
spin: lw xflag, (xflagp)
      beqz xflag, spin
      lw xdata, (xdatap)
```

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

Sequential Consistency = arbitrary *order-preserving interleaving* of memory references of sequential programs

Simple Producer-Consumer Example

Initially flag =0

```
sw xdata, (xdatap)
li xflag, 1
sw xflag, (xflagp)
```

```
spin: lw xflag, (xflagp)
      beqz xflag, spin
      lw xdata, (xdatap)
```

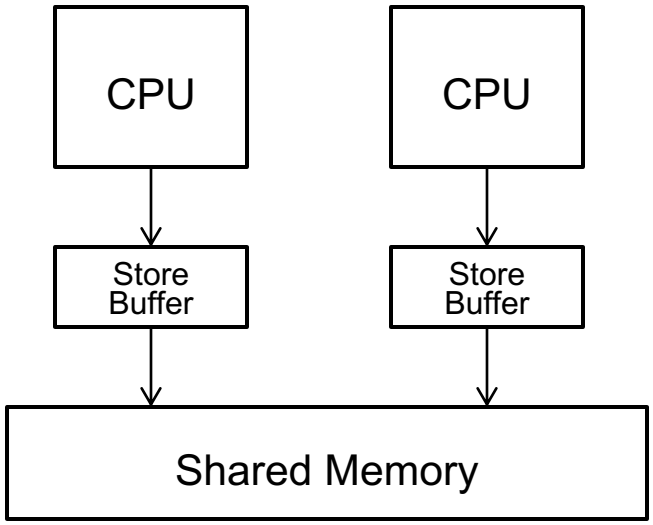
Dependencies from sequential ISA

Dependencies added by sequentially consistent memory model

Most real machines are not SC

- Only a few commercial ISAs require SC
  - Neither IBM 370 nor 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

Store Buffer Optimization



- Common optimization allows stores to be buffered while waiting for access to shared memory
- Load optimizations:
  - Later loads can go ahead of buffered stores if to different address
  - Later loads can bypass value from earlier buffered store if to same address

TSO example

- Allows local buffering of stores by processor

```
Initially M[X] = M[Y] = 0

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

Possible Outcomes

P1.x2	P2.x2	SC	TSO
0	0	N	Y
0	1	Y	Y
1	0	Y	Y
1	1	Y	Y

- TSO is the strongest memory model in common use

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 hardware 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

```
sw xdata, (xdatap)
li xflag, 1
fence w,w //Write-write fence
sw xflag, (xflagp)
```

```
spin: lw xflag, (xflagp)
      beqz xflag, spin
      fence r,r // Read-read fence
      lw xdata, (xdatap)
```

CS152/252a Administrivia

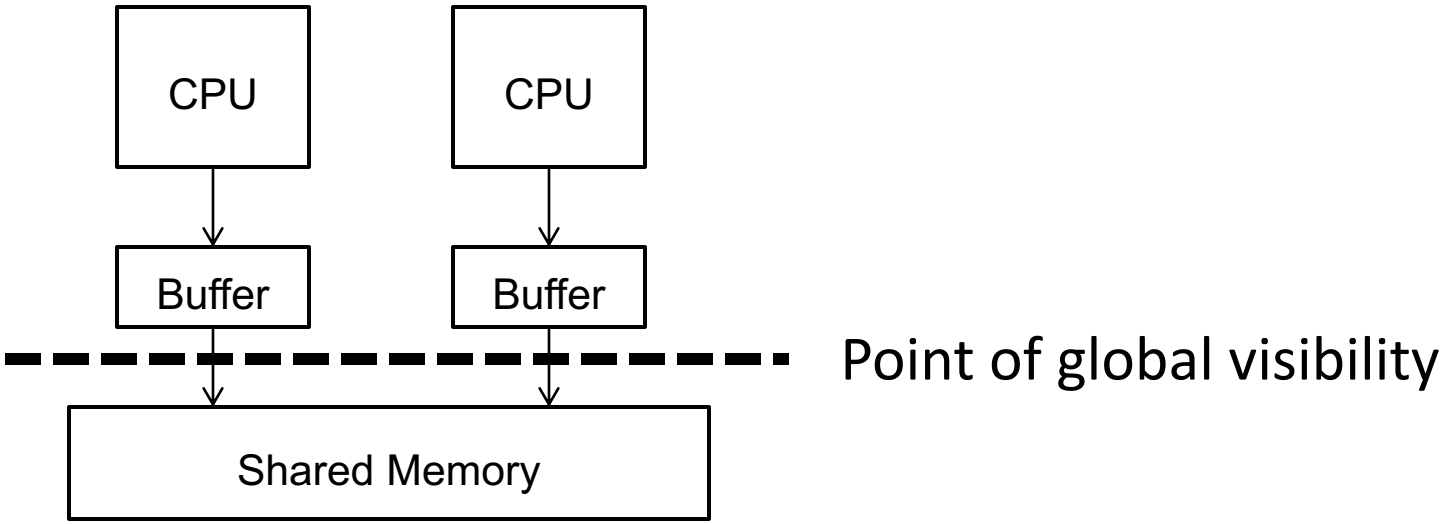
- Today is last day of me teaching
- One last technical topic: Synchronization primitives
  - Staff and I are deciding whether to cover this somehow

- This Thursday: Apple guest lecture on BP + PF.
  - Will be useful for reviewing for the final.
  - Will be recorded, but not sure when recording will be released.
- Next Tuesday: Sagar on hyperscale computing.
- Next Thursday: Annapurna Labs/Amazon ~ Nafea Bshara on Amazon Nitro.
  - Come in person! Ask questions!
  - Lunch w/ Nafea afterwards. Lottery winners by this Thursday.

Range of Memory Consistency Models

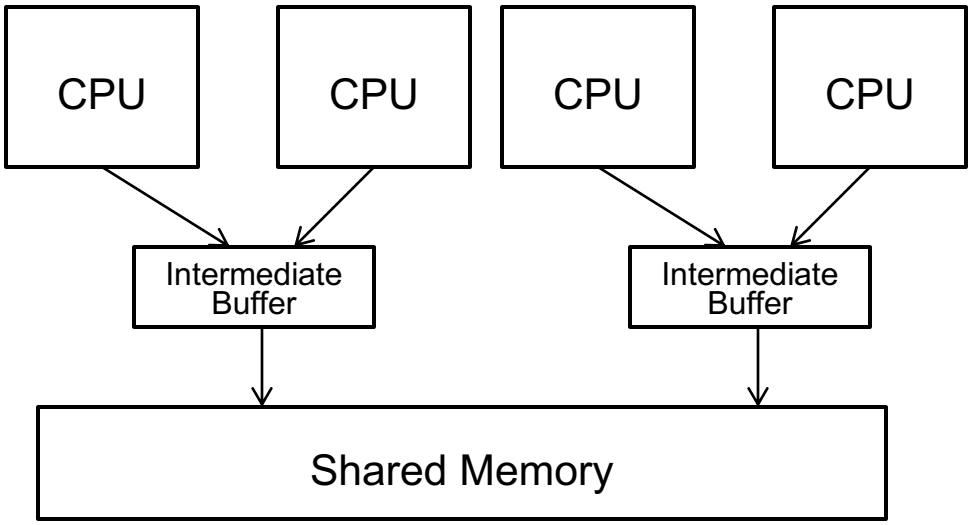
- SC “Sequential Consistency”
  - MIPS R10K
- TSO “Total Store Order”
  - *processor can see its own writes before others do (store buffer)*
  - IBM-370 TSO, x86 TSO, SPARC TSO (default), RISC-V RVTSO (optional)
- Weak, multi-copy-atomic memory models
  - *all processors see writes by another processor in same order*
  - Revised ARM v8 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*
  - ARM v7, original ARM v8
  - 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)

Relaxed Memory Models

- Not all dependencies assumed by SC are supported, and software has to explicitly insert additional dependencies were 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

How on earth are programmers supposed to work with this????

Relaxed Consistency Models, Formalized

- **Key idea:** allow reordering of some memory operations, use synchronization (again) to give illusion of SC

- Reordering rules: X → Y means X must be visible before Y is visible
- Sequential consistency (SC) requires:

  R → R

  R → W

  W → R

  W → W … i.e., all orderings are enforced

- Relaxed models remove a subset of these ordering rules

TSO, Formalized

- Relaxes W → R
- Enforces R → R, R → W, W → W
- In other words:

  Subsequent reads don't have to wait for writes to become visible

- Recall: many of the interesting cases only required W → W, R → R ☺
- E.g., our old friend

P1:

```
Data = 23
Flag = 1
```

P2:

```
while (Flag != 1) { }
x = Data
```

Release Consistency

- Relax all four orderings:

  $R \rightarrow R$, $R \rightarrow W$, $W \rightarrow R$, $W \rightarrow W$

- Explicit synchronization **acquire** S_A and **release** S_R semantics
- Enforced orderings:

  $S_A \rightarrow R$, $S_A \rightarrow W$, $R \rightarrow S_R$, $W \rightarrow S_R$ — Synchronization boundaries

  $S_A \rightarrow S_A$, $S_R \rightarrow S_R$, $S_A \rightarrow S_R$, $S_R \rightarrow S_A$ — Synchronizations are ordered

- Note: $R \rightarrow S_A$, $W \rightarrow S_A$ (vice versa for S_R) not enforced

- Used by ARM, RISC V

Other Memory Consistency Models

Relaxation:	W →R Order	W →W Order	R →RW Order	Read Others' Write Early	Read Own Write Early	Safety Net
IBM 370	✓					serialization instructions
TSO	✓				✓	RMW
PC	✓			✓	✓	RMW
PSO	✓	✓			✓	RMW, STBAR
WO	✓	✓	✓		✓	synchronization
RCsc	✓	✓	✓		✓	release, acquire, nsync, RMW
RCpc	✓	✓	✓	✓	✓	release, acquire, nsync, RMW
Alpha	✓	✓	✓		✓	MB, WMB
RMO	✓	✓	✓		✓	various MEMBARs
PowerPC	✓	✓	✓	✓	✓	SYNC

But compilers reorder too!

```
//Producer code
*datap = x/y;
*flagp = 1;
```

```
//Consumer code
while (!*flagp)
      ;
d = *datap;
```

- 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
  - Adopt release consistency
- 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`

Acknowledgements

- This course is partly inspired by previous MIT 6.823 and Berkeley CS252 computer architecture courses created by my collaborators and colleagues:
  - Arvind (MIT)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)
  - Krste Asanovic (UCB)
  - Sophia Shao (UCB)