

CS 152 Computer Architecture and Engineering

CS252 Graduate Computer Architecture

Lecture 19 Memory Consistency Models and Synchronization

John Wawrzynek
Electrical Engineering and Computer Sciences
University of California at Berkeley

<http://www.eecs.berkeley.edu/~johnw>
<http://inst.eecs.berkeley.edu/~cs152>

Last Time in Lecture 18

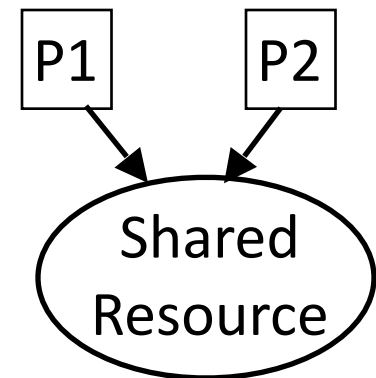
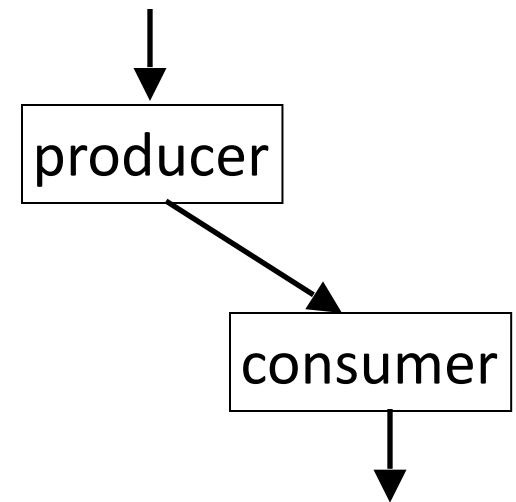
- Cache coherence, making sure every store to memory is eventually visible to any load to same memory address
- Cache line states: M,S,I or M,E,S,I
- Cache miss if tag not present, or line has wrong state
 - Write to a shared line is handled as a miss
- Snoopy coherence:
 - Broadcast updates and probe all cache tags on any miss of any processor, used to be bus connection now often broadcast over point-to-point links
 - 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

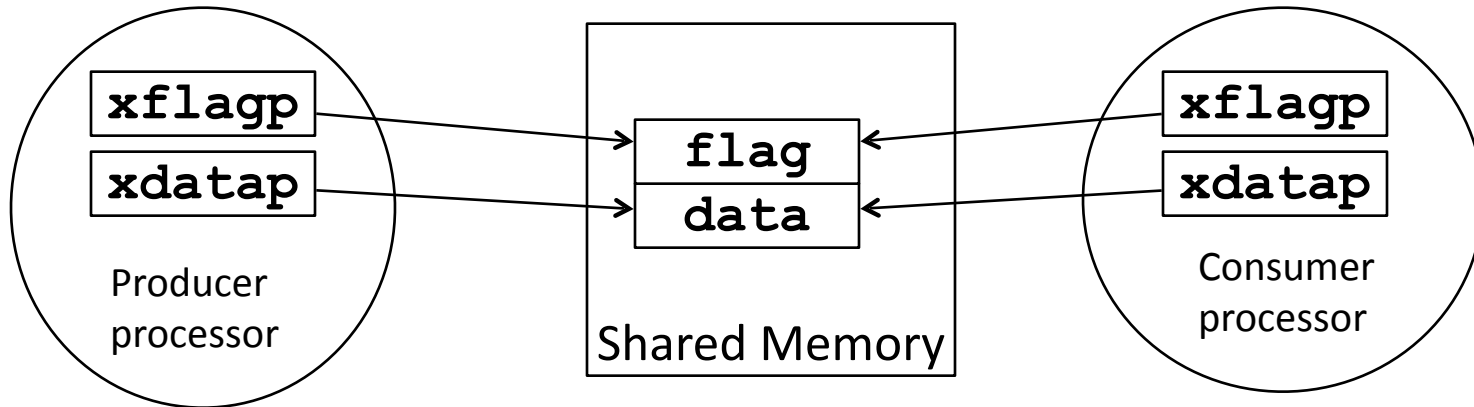
The need for synchronization arises whenever there are concurrent processes in a system (*even in a uniprocessor system*).

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

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

Is this correct?

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)
```

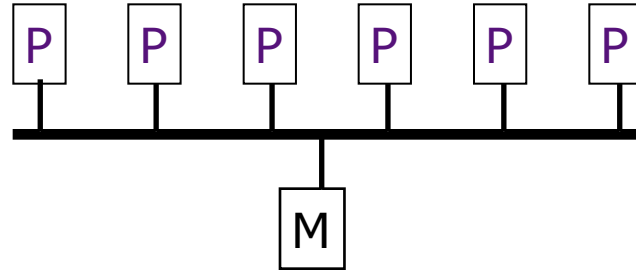
Can consumer read **flag=1** before **data**
written by producer visible to consumer?

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

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

Sequential Consistency (SC)

A Memory Model

- 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 it needs to account for them.
- SC articulated by Leslie Lamport - 2013 Turing award winner
- “Intuitive” model of parallelism
 - Multiple threads running in parallel manipulate a *single main memory*, and so everything must happen in order. There’s no notion that two events can occur “at the same time”. Note that this rule says nothing about what order the events happen in— just that they happen in some order.
 - Events happen in *program order*: the events in a single thread happen in the order in which they were written.
- The problem with this model is that it’s terribly, disastrously slow.

[Thanks to James Bornholt, UT Austin]

SC Example [James Bornholt, UT Austin]

Consistency models deal with how multiple threads (or workers, or nodes, or replicas, etc.) see the world. Consider this simple program, running two threads, and where A and B are initially both 0:

Thread 1

```
(1) A = 1  
(2) print(B)
```

Thread 2

```
(3) B = 1  
(4) print(A)
```

To understand what this program can output, we should think about the order in which its events can happen. Intuitively, there are two obvious orders in which this program could run:

- (1) → (2) → (3) → (4): The first thread runs both its events before the second thread, and so the program prints 01.
- (3) → (4) → (1) → (2): The second thread runs both its events before the first thread. The program still prints 01.

There are also some less obvious orders, where the instructions are interleaved with each other:

- (1) → (3) → (2) → (4): The first instruction in each thread runs before the second instruction in either thread, printing 11.
- (1) → (3) → (4) → (2): The first instruction from the first thread runs, then both instructions from the second thread, then the second instruction from the first thread. The program still prints 11.
- and a few others that have the same effect.

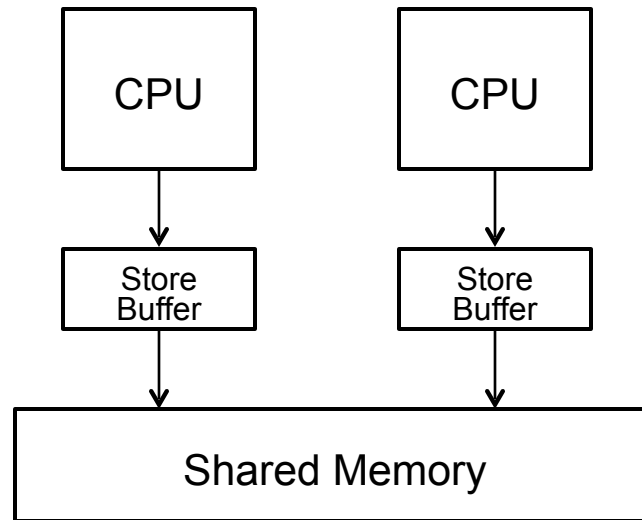
Under SC, 00 can't be printed. Without SC could it? Consider store buffers!

(Cache coherence doesn't kick in until memory system moves values to last level cache.)

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
- Resulted in “weak” memory models with fewer guarantees
- Architects/language designers/applications developers work hard to explain weak memory behavior

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, PC

Write A

Read B

Allowing reads to move ahead of writes

- Total store ordering (TSO)
 - Processor P can read B before its write to A is seen by all processors (processor can move its own reads in front of its own writes)
 - Reads by other processors cannot return new value of A until the write to A is observed by all processors
- Processor consistency (PC)
 - Any processor can read new value of A before the write is observed by all processors
- In TSO and PC, only $W \rightarrow R$ order is relaxed. The $W \rightarrow W$ constraint still exists. Writes by the same thread are not reordered (they occur in program order)

[Stanford CS149, Winter 2019]

A weaker memory model: Total Store Ordering (TSO) - example

- Allows local buffering of stores by processor

X, Y shared, initially Memory[X] = Memory[Y] = 0

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

Possible Outcomes

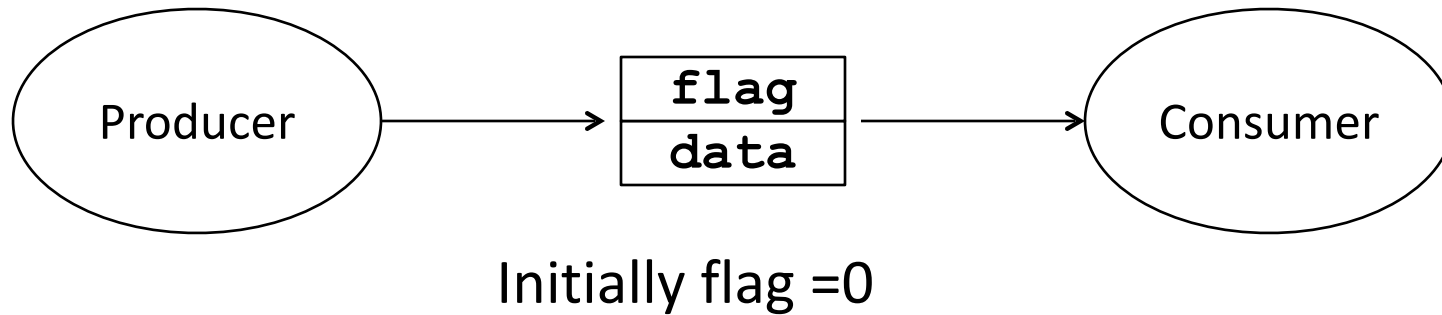
| P1 . x2 | P2 . x2 | SC | TSO |
|---------|---------|----|-----|
| 0 | 0 | N | Y |
| 0 | 1 | Y | Y |
| 1 | 0 | Y | Y |
| 1 | 1 | Y | Y |

- 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 software to impose hardware reorderings
 - Additional instructions (fences) are provided to allow software to specify which orderings are required

Fences in Producer-Consumer Example



```
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
```

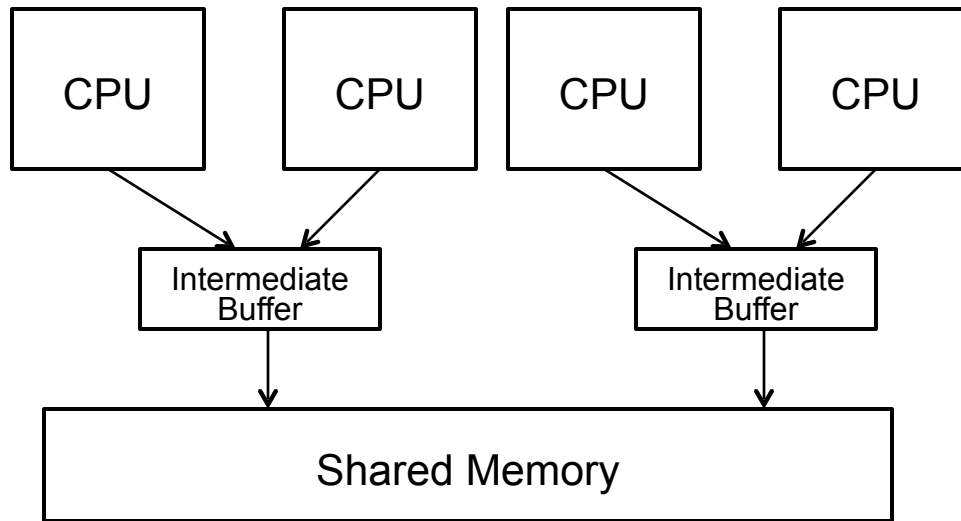
```
lw xdata, (xdatap)
```

A fence instruction forces memory operations before it to complete before memory operation after it can begin.

CS152 Administrivia

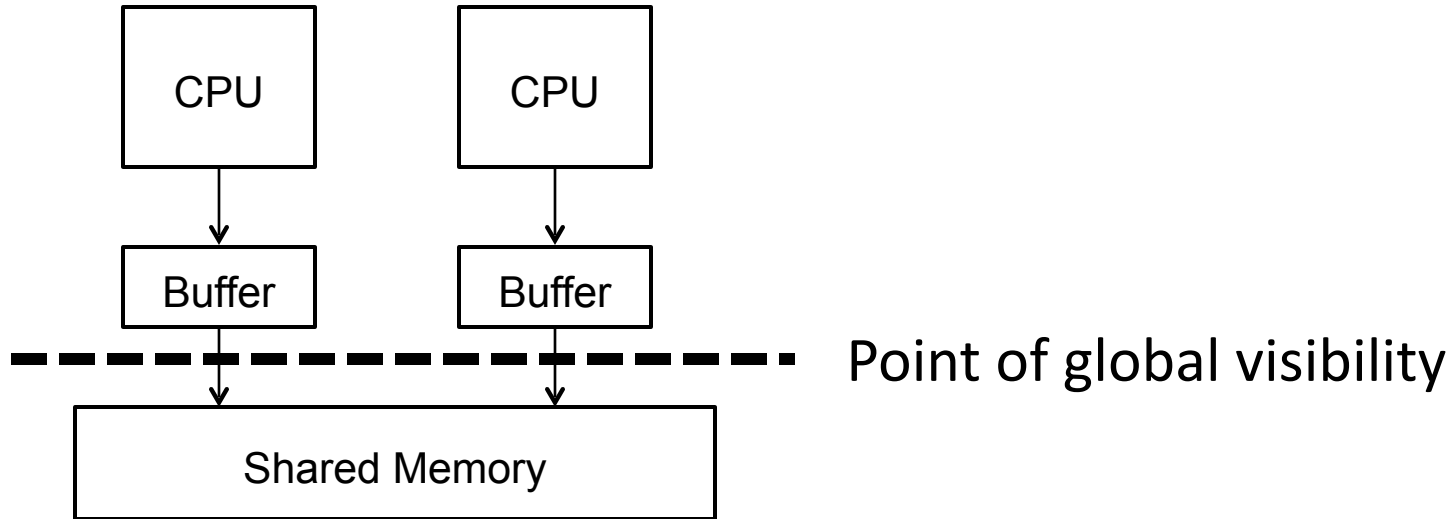
- Lab 4 due Tuesday April 19, Lab 5 out
- PS 5 (final PS) due April 26
- Midterm 2 Comments?
 - Grading underway

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)

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”)

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

Relaxed Memory Models

- 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

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

Language-Level Memory Models

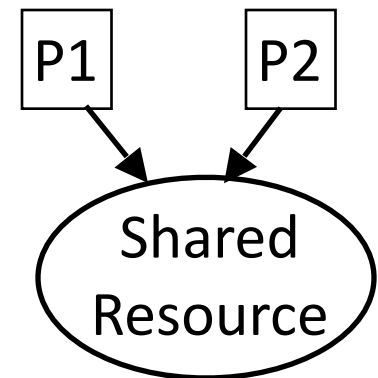
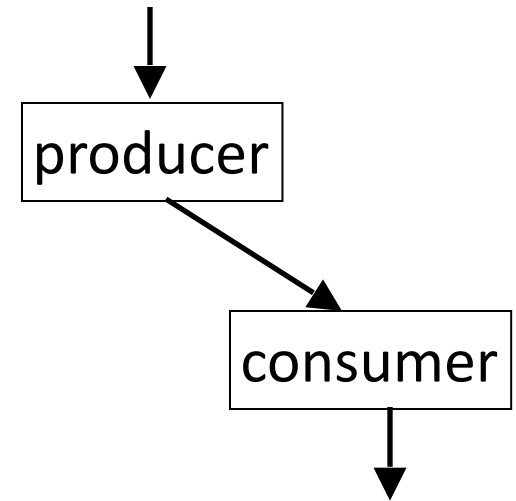
- Programming languages have memory models too
- Hide details of each ISA's memory model underneath language standard
 - c.f. C function declarations versus ISA-specific subroutine linkage convention
- Language memory models: C/C++, Java
- Describe legal behaviors of threaded code in each language and 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`

Synchronization

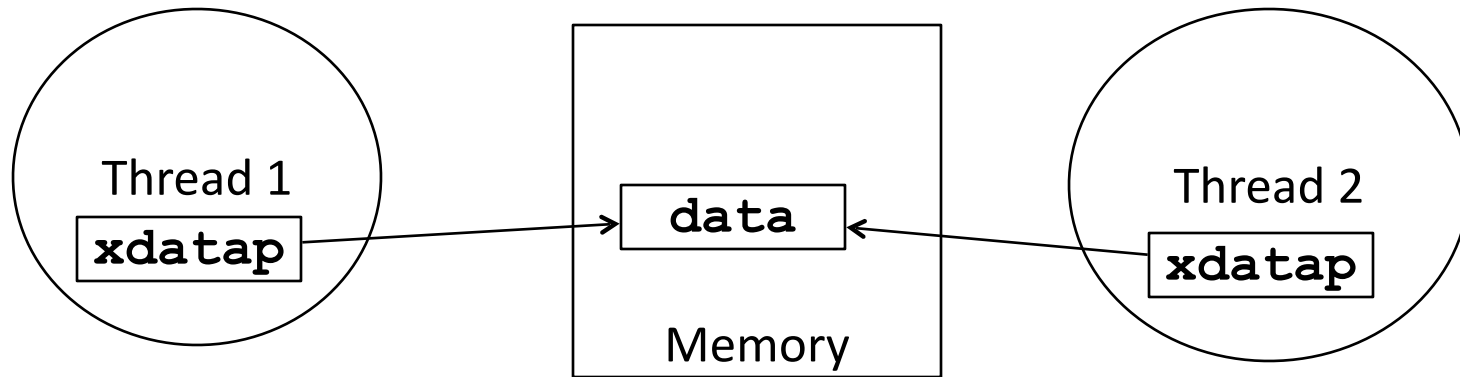
The need for synchronization arises whenever there are concurrent processes in a system (*even in a uniprocessor system*).

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(st) Mutual-Exclusion Example



```
// Both threads execute:  
ld xdata, (xdatap)  
add xdata, 1  
sd xdata, (xdatap)
```

Is this correct? What are the possible outcomes?

Need to provide *exclusive* access to each thread.

Mutual Exclusion Using Load/Store (assume SC)

A protocol based on two shared variables $c1$ and $c2$. Initially, both $c1$ and $c2$ are 0. $c1$ or $c2$ indicate *intent* to enter the critical section (reservation).

Process 1

```
...  
c1=1;  
L: if c2=1 then go to L  
   < critical section >  
c1=0;
```

Process 2

```
...  
c2=1;  
L: if c1=1 then go to L  
   < critical section >  
c2=0;
```

What is wrong?

Deadlock!

Mutual Exclusion: *second attempt*

To avoid *deadlock*, let a process give up the reservation (i.e. Process 1 sets c_1 to 0) while waiting.

Process 1

```
...
L: c1=1;
   if c2=1 then
       { c1=0; go to L }
   < critical section >
   c1=0
```

Process 2

```
...
L: c2=1;
   if c1=1 then
       { c2=0; go to L }
   < critical section >
   c2=0
```

- Deadlock is not possible but with a low probability a *livelock* may occur.
- An unlucky process may never get to enter the critical section \Rightarrow *starvation*

A Protocol for Mutual Exclusion

T. Dekker, 1966

A protocol based on 3 shared variables c_1 , c_2 and $turn$.
Initially, both c_1 and c_2 are 0 (*not busy*)

Process 1

```
...
c1=1;
turn = 1;
L: if c2=1 & turn=1
           then go to L
   < critical section >
c1=0;
```

Process 2

```
...
c2=1;
turn = 2;
L: if c1=1 & turn=2
           then go to L
   < critical section >
c2=0;
```

- $turn = i$ ensures that only process i can wait
- variables c_1 and c_2 ensure *mutual exclusion*

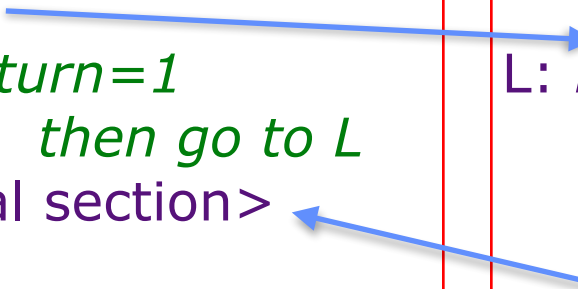
*Solution for n processes was given by Dijkstra
and is quite tricky!*

Analysis of Dekker's Algorithm

Scenario 1

```
... Process 1
c1=1;
turn = 1;
L: if c2=1 & turn=1
    then go to L
    < critical section >
c1=0;
```

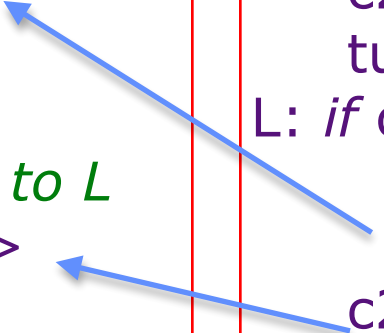
```
... Process 2
c2=1;
turn = 2;
L: if c1=1 & turn=2
    then go to L
    < critical section >
c2=0;
```



Scenario 2

```
... Process 1
c1=1;
turn = 1;
L: if c2=1 & turn=1
    then go to L
    < critical section >
c1=0;
```

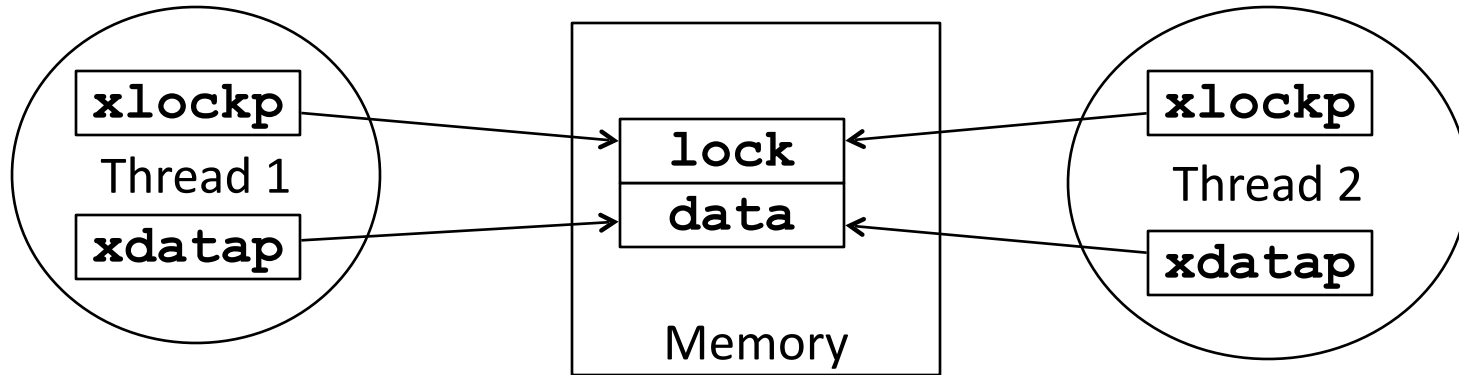
```
... Process 2
c2=1;
turn = 2;
L: if c1=1 & turn=2
    then go to L
    < critical section >
c2=0;
```



ISA Support for Mutual-Exclusion Locks

- Regular loads and stores in SC model (plus fences in weaker model) sufficient to implement mutual exclusion, but code is inefficient and complex
- Therefore, atomic read-modify-write (RMW) instructions added to ISAs to support mutual exclusion
- Many forms of atomic RMW instruction possible, some simple examples:
 - Test and set ($\text{reg_x} = \text{M}[a]; \text{M}[a]=1$)
 - Swap ($\text{reg_x}=\text{M}[a]; \text{M}[a] = \text{reg_y}$)

Lock for Mutual-Exclusion Example



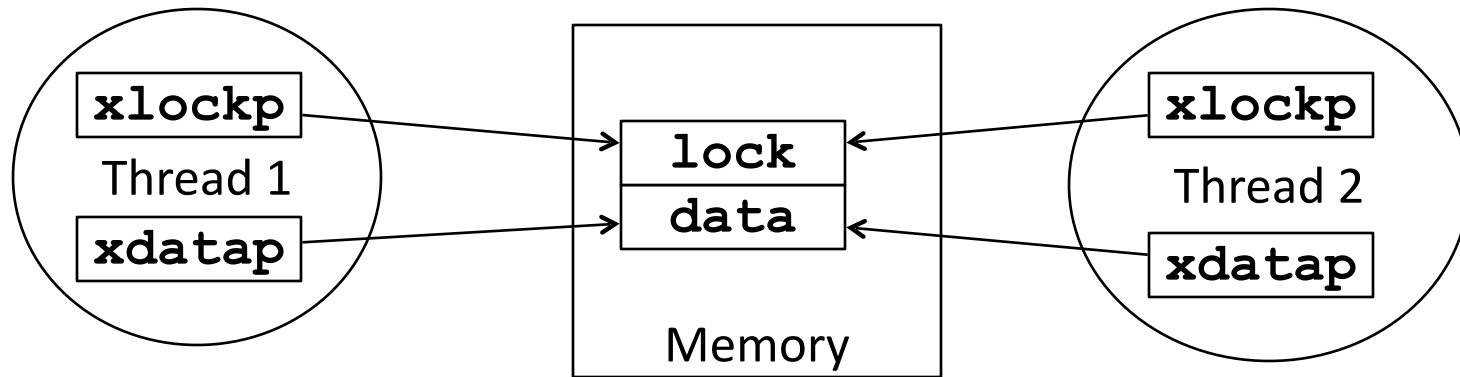
// Both threads execute:

```
li xone, 1
```

```
spin: amoswap xlock, xone, (xlockp)           Acquire Lock
      bnez xlock, spin
      ld xdata, (xdatap)
      add xdata, 1                             Critical Section
      sd xdata, (xdatap)
      sd x0, (xlockp)                           Release Lock
```

Assumes SC memory model

Lock for Mutual-Exclusion with Relaxed MM



// Both threads execute:

```
li xone, 1
```

```
spin: amoswap xlock, xone, (xlockp)
```

```
bnez xlock, spin
```

Acquire Lock

```
fence r,rw
```

```
ld xdata, (xdatap)
```

```
add xdata, 1
```

Critical Section

```
sd xdata, (xdatap)
```

```
fence rw,w
```

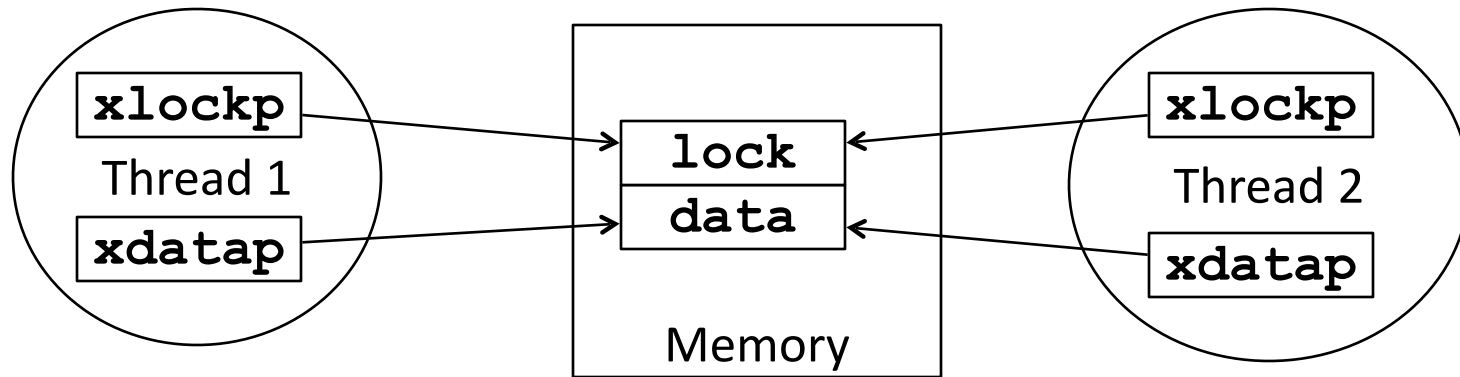
```
sd x0, (xlockp)
```

Release Lock

RISC-V Atomic Memory Operations

- Atomic Memory Operations (AMOs) have two ordering bits:
 - Acquire (aq)
 - Release (rl)
- If both clear, no additional ordering implied
- If aq set, then AMO “happens before” any following loads or stores
- If rl set, then AMO “happens after” any earlier loads or stores
- If both aq and rl set, then AMO happens in program order

Lock for Mutual-Exclusion using RISC-V AMO



// Both threads execute:

```
li xone, 1
```

```
spin: amoswap.w.aq xlock, xone, (xlockp)
```

Acquire Lock

```
bnez xlock, spin
```

```
ld xdata, (xdatap)
```

```
add xdata, 1
```

Critical Section

```
sd xdata, (xdatap)
```

```
amoswap.w.rl x0, x0, (xlockp)
```


Release Lock

RISC-V FENCE versus AMO.aq/rl

```
sd x1, (a1) # Unrelated store
ld x2, (a2) # Unrelated load
li t0, 1
```

again:

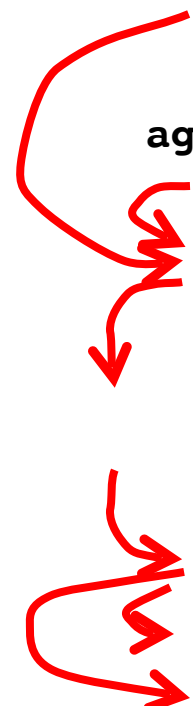
```
amoswap.w.aq t0, t0, (a0)
bnez t0, again
# ...
# critical section
# ...
amoswap.w.rl x0, x0, (a0)
sd x3, (a3) # Unrelated store
ld x4, (a4) # Unrelated load
```



```
sd x1, (a1) # Unrelated store
ld x2, (a2) # Unrelated load
li t0, 1
```

again:

```
amoswap.w t0, t0, (a0)
fence r, rw
bnez t0, again
# ...
# critical section
# ...
fence rw, w
amoswap.w x0, x0, (a0)
sd x3, (a3) # Unrelated store
ld x4, (a4) # Unrelated load
```



AMOs only order the AMO w.r.t. other loads/stores/AMOs

FENCES order every load/store/AMO before/after FENCE

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)