Computer Architecture

Lecture 21b: Memory Ordering (Memory Consistency)

Prof. Onur Mutlu
ETH Zürich

Fall 2019

5 December 2019

Memory Ordering in Multiprocessors

Readings: Memory Consistency

- Required
  - Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," IEEE Transactions on Computers, 1979

- Recommended
  - Gharachorloo et al., "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors," ISCA 1990.
  - Gharachorloo et al., "Two Techniques to Enhance the Performance of Memory Consistency Models," ICPP 1991.
  - Ceze et al., "BulkSC: bulk enforcement of sequential consistency," ISCA 2007.

Memory Consistency vs. Cache Coherence

- Consistency is about ordering of all memory operations from different processors (i.e., to different memory locations)
  - Global ordering of accesses to *all* memory *locations*
- Coherence is about ordering of operations from different processors to the same memory location
  - Local ordering of accesses to *each* cache *block*

Difficulties of Multiprocessing

- Much of **parallel computer architecture** is about
  - Designing machines that overcome the sequential and parallel bottlenecks to achieve higher performance and efficiency
  - Making programmer's job easier in writing correct and high-performance parallel programs

Ordering of Operations

- Operations: A, B, C, D
  - In what order should the hardware execute (and report the results of) these operations?
- A contract between programmer and microarchitect
  - Specified by the ISA
- Preserving an "expected" (more accurately, "agreed upon") order simplifies programmer's life
  - Ease of debugging; ease of state recovery, exception handling
- Preserving an "expected" order usually makes the hardware designer's life difficult
  - Especially if the goal is to design a high performance processor: Recall load-store queues in out of order execution and their complexity

Memory Ordering in a Single Processor

- Specified by the von Neumann model
- Sequential order
  - Hardware executes the load and store operations in the order specified by the sequential program
- Out-of-order execution does not change the semantics
  - Hardware retires (reports to software the results of) the load and store operations in the order specified by the sequential program
- Advantages: 1) Architectural state is precise within an execution.
  2) Architectural state is consistent across different runs of the program
  → Easier to debug programs
- Disadvantage: Preserving order adds overhead, reduces performance, increases complexity, reduces scalability

Memory Ordering in a Dataflow Processor

- A memory operation executes when its operands are ready
- Ordering specified only by data dependencies
- Two operations can be executed and retired in any order if they have no dependency
- Advantage: Lots of parallelism → high performance
- Disadvantages:
  - Precise state is very hard to maintain (No specified order) → Very hard to debug
  - Order can change across runs of the same program → Very hard to debug

Memory Ordering in a MIMD Processor

- Each processor's memory operations are in sequential order with respect to the "thread" running on that processor (assume each processor obeys the von Neumann model)
- Multiple processors execute memory operations concurrently
- How does the memory see the order of operations from all processors?
  - In other words, what is the ordering of operations across different processors?

Why Does This Even Matter?

- Ease of debugging
  - It is nice to have the same execution done at different times to have the same order of execution → Repeatability
- Correctness
  - Can we have incorrect execution if the order of memory operations is different from the point of view of different processors?
- Performance and overhead
  - Enforcing a strict "sequential ordering" can make life harder for the hardware designer in implementing performance enhancement techniques (e.g., OoO execution, caches)

When Could Order Affect Correctness?

- When protecting shared data

Protecting Shared Data

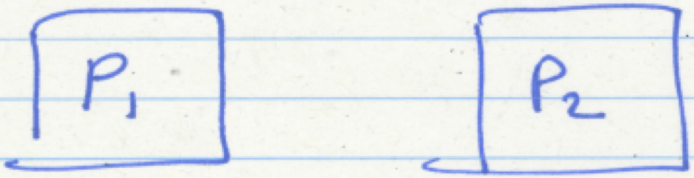
- Threads are not allowed to update shared data concurrently
  - For correctness purposes
- Accesses to shared data are encapsulated inside *critical sections* or protected via *synchronization constructs (locks, semaphores, condition variables)*
- Only one thread can execute a critical section at a given time
  - Mutual exclusion principle
- A multiprocessor should provide the *correct* execution of synchronization primitives to enable the programmer to protect shared data

Supporting Mutual Exclusion

- Programmer needs to make sure mutual exclusion (synchronization) is correctly implemented
  - We will assume this
  - But, correct parallel programming is an important topic
  - Reading: Dijkstra, "Cooperating Sequential Processes," 1965.
    - http://www.cs.utexas.edu/users/EWD/transcriptions/EWD01xx/EWD123.html
    - See Dekker's algorithm for mutual exclusion

- Programmer relies on hardware primitives to support correct synchronization
- If hardware primitives are not correct (or unpredictable), programmer's life is tough
- If hardware primitives are correct but not easy to reason about or use, programmer's life is still tough

Protecting Shared Data



$F_1 = \emptyset$

A $F_1 = 1$

B IF ($F_2 == \emptyset$) THEN
{Critical section}
$F_1 \neq \emptyset$
ELSE
{...}

$F_2 = \emptyset$

X $F_2 = 1$

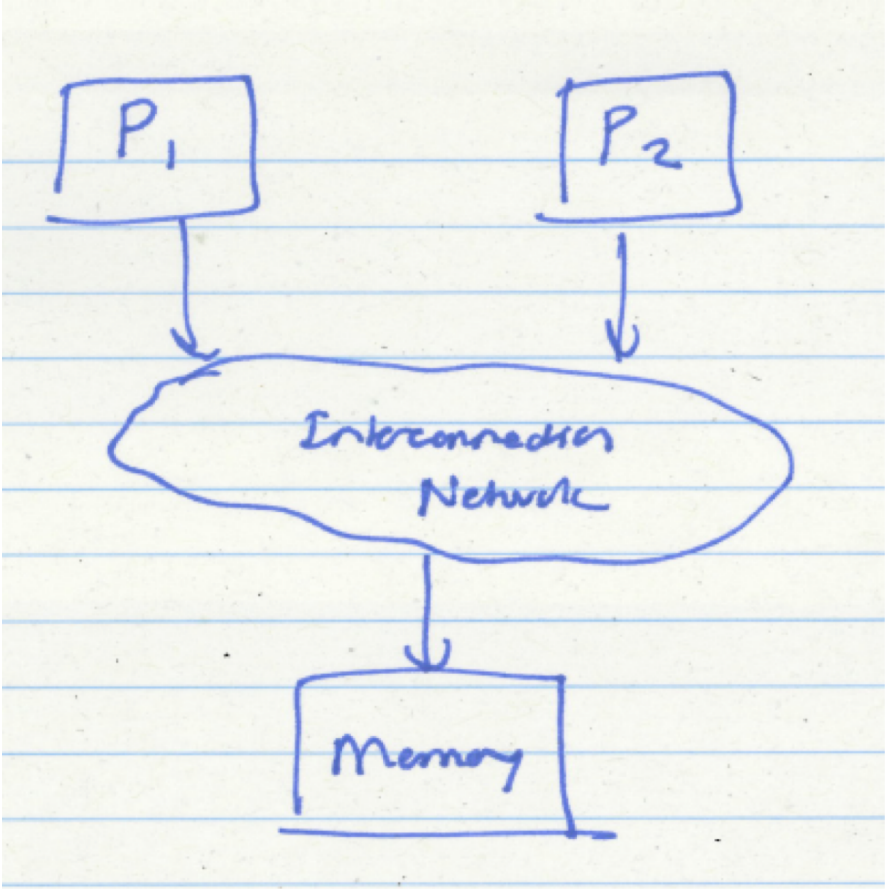
Y IF ($F_1 == \emptyset$) THEN
{Critical section}
$F2 = \emptyset$
ELSE
{....}

Only P_1 or P_2 should be in this section at any given time, not both

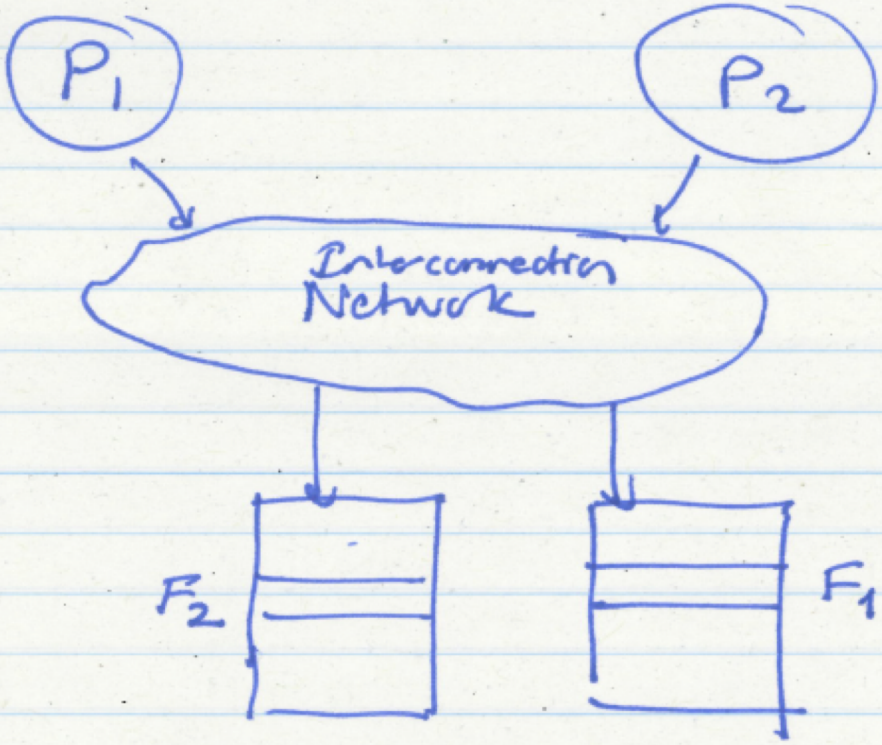
Assume P1 is in critical section.
Intuitively, it must have executed A,
which means F1 must be 1 (as A happens before B),
which means P2 should not enter the critical section.

A Question

- Can the two processors be in the critical section at the same time given that they both obey the von Neumann model?
- Answer: yes



An Incorrect Result (due to an implementation that does not provide sequential consistency)



time 0: P_1 executes A (set $F_1 = 1$) st F_1 complete (from P_1's view) A is sent to memory

P_2 executes X (set $F_2 = 1$) st F_2 complete (from P_2's view) X is sent to memory

Both Processors in Critical Section

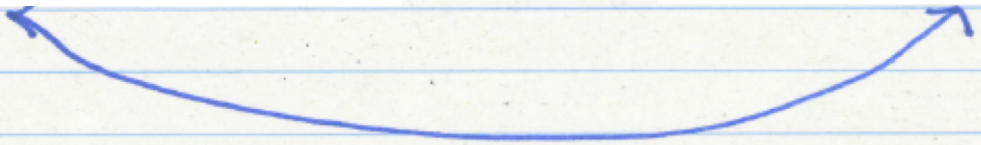
time 0:	P_1 executes A (set $F_1 = 1$) st F_1 complete (from P_1's view) A is sent to memory	P_2 executes X (set $F_2 = 1$) st F_2 complete (from P_2's view) X is sent to memory
time 1:	P_1 executes B (test $F_2 == 0$) ld F_2 started B is sent to memory	P_2 executes Y (test $F_1 == 0$) ld F_1 started Y is sent to memory
time 50:	Memory sends back to P_1 F_2 (0) ld F_2 complete	Memory sends back to P_2 (F_1 (0) ld F_1 complete
time 51:	P_1 is in critical section	P_2 is in critical section
time 100:	Memory completes A $F_1 = 1$ in memory (too late!)	Memory completes X $F_2 = 1$ in memory (too late!)

What happened?

P_1's view of mem. ops		P_2's view	
A	($F_1=1$)	X	($F_2=1$)
B	(test $F_2=0$)	Y	(test $F_1=0$)
X	($F_2=1$)	A	($F_1=1$)

A appeared to happen before X

X appeared to happen before A



Problem!

These two processors did not see the same order of operations on memory

The Problem

- The two processors did **NOT** see the same order of operations to memory
- The "happened before" relationship between multiple updates to memory was inconsistent between the two processors' points of view
- As a result, each processor thought the other was **not** in the critical section

How Can We Solve The Problem?

- Idea: Sequential consistency

- All processors see the same order of operations to memory
- i.e., all memory operations happen in an order (called the global total order) that is consistent across all processors

- Assumption: within this global order, each processor's operations appear in sequential order with respect to its own operations.

Sequential Consistency

- Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," IEEE Transactions on Computers, 1979

- A multiprocessor 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

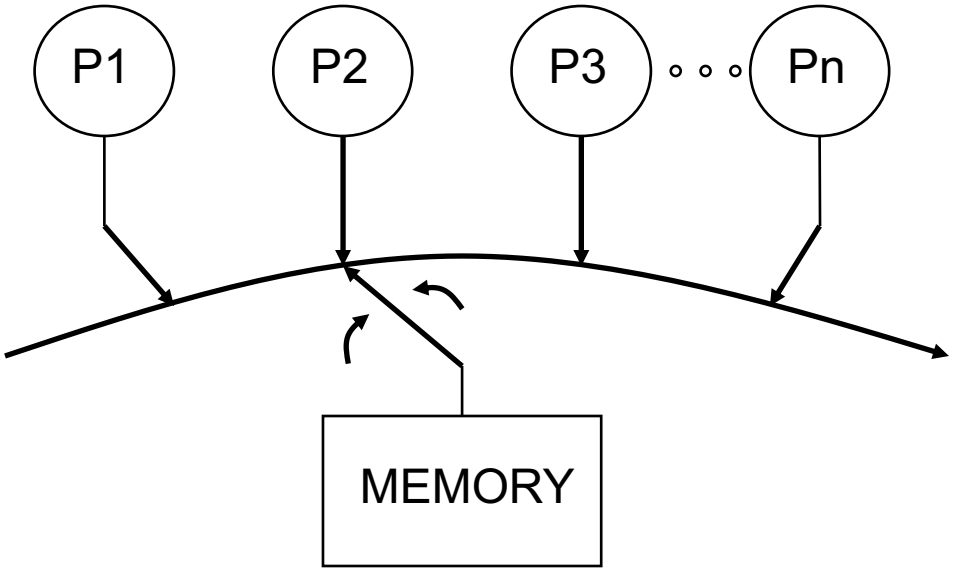
AND

  - the operations of each individual processor appear in this sequence in the order specified by its program

- This is a memory ordering model, or memory model
  - Specified by the ISA

Programmer's Abstraction

- Memory is a switch that services one load or store at a time from any processor
- All processors see the currently serviced load or store at the same time
- Each processor's operations are serviced in program order



Sequentially Consistent Operation Orders

- Potential correct global orders (all are correct):

- A B X Y
- A X B Y
- A X Y B
- X A B Y
- X A Y B
- X Y A B

- Which order (interleaving) is observed depends on implementation and dynamic latencies

Consequences of Sequential Consistency

- Corollaries

1. Within the same execution, all processors see the same global order of operations to memory
   → No correctness issue
   → Satisfies the "happened before" intuition

2. Across different executions, different global orders can be observed (each of which is sequentially consistent)
   → Debugging is still difficult (as order changes across runs)

Issues with Sequential Consistency?

- Nice abstraction for programming, but two issues:
  - Too conservative ordering requirements
  - Limits the aggressiveness of performance enhancement techniques

- Is the total global order requirement too strong?
  - Do we need a global order across all operations and all processors?
  - How about a global order only across all stores?
    - Total store order memory model; unique store order model
  - How about enforcing a global order only at the boundaries of synchronization?
    - Relaxed memory models
    - Acquire-release consistency model

Issues with Sequential Consistency?

- Performance enhancement techniques that could make SC implementation difficult

- Out-of-order execution
  - Loads happen out-of-order with respect to each other and with respect to independent stores → makes it difficult for all processors to see the same global order of all memory operations

- Caching
  - A memory location is now present in multiple places
  - Prevents the effect of a store to be seen by other processors → makes it difficult for all processors to see the same global order of all memory operations

Weaker Memory Consistency

- The ordering of operations is important when the order affects operations on shared data → i.e., when processors need to synchronize to execute a "program region"

- Weak consistency
  - Idea: Programmer specifies regions in which memory operations do not need to be ordered
  - "Memory fence" instructions delineate those regions
    - All memory operations before a fence must complete before fence is executed
    - All memory operations after the fence must wait for the fence to complete
    - Fences complete in program order
  - All synchronization operations act like a fence

Tradeoffs: Weaker Consistency

- Advantage
  - No need to guarantee a very strict order of memory operations
    - → Enables the hardware implementation of performance enhancement techniques to be simpler
    - → Can be higher performance than stricter ordering
- Disadvantage
  - More burden on the programmer or software (need to get the "fences" correct)
- Another example of the programmer-microarchitect tradeoff

Example Question (I)

- Question 4 in
  - http://www.ece.cmu.edu/~ece447/s13/lib/exe/fetch.php?media=final.pdf

4. Sequential Consistency [30 points]

Two threads (A and B) are concurrently running on a dual-core processor that implements a *sequentially consistent* memory model. Assume that the value at address 0x1000 is initialized to 0.

Thread A

```
X1: st 0x1, (0x1000)
X2: ld $r1, (0x1000)
X3: st 0x2, (0x1000)
X4: ld $r2, (0x1000)
```

Thread B

```
Y1: st 0x3, (0x1000)
Y2: ld $r3, (0x1000)
Y3: st 0x4, (0x1000)
Y4: ld $r4, (0x1000)
```

(a) List all possible values that can be stored in `$r3` after both threads have finished executing.

Example Question (II)

(b) After both threads have finished executing, you find that (`$r1`, `$r2`, `$r3`, `$r4`) = (1, 2, 3, 4). How many different *instruction interleavings* of the two threads produce this result?

(c) What is the total number of all possible instruction interleavings? You need not expand factorials.

(d) On a *non-sequentially consistent* processor, is the total number of all possible instruction interleavings less than, equal to, or greater than your answer to question (c)?

Computer Architecture

Lecture 21b: Memory Ordering (Memory Consistency)

Prof. Onur Mutlu
ETH Zürich

Fall 2019

5 December 2019