# 18-447
# Computer Architecture
# Lecture 28: Memory Consistency and Cache Coherence

Prof. Onur Mutlu
Carnegie Mellon University
Spring 2015, 4/8/2015

# My Office Hours Today

- Moved to 3:45-5pm

# Assignments

- Lab 7 out
  - Due April 17
- HW 6
  - Due Friday (April 10)
- Midterm II
  - April 24

# A Note on 740 Next Semester

- If you like 447, 740 is the next course in sequence
- Tentative Time: Lect. MW 7:30-9:20pm, Rect. T 7:30pm
- Content:
  - Lectures: More advanced, with a different perspective
  - Readings: Many fundamental and research readings; will do many critical reviews
  - Recitations: Delving deeper into papers, concepts, advanced topics & discussions
  - Project: Relatively open ended semester-long research project. Proposal → milestones → final poster and presentation
  - Exams: lighter and fewer
  - Homeworks: None
- More relaxed course where you can go deeper and obtain the tools to advance the state of the art

# A Note on Internships & Jobs

- If you are interested in pursuing computer architecture
  - Research
  - Internships
  - Jobs

- Let me know & I can help

# Where We Are in Lecture Schedule

- The memory hierarchy
- Caches, caches, more caches
- Virtualizing the memory hierarchy: Virtual Memory
- Main memory: DRAM
- Main memory control, scheduling
- Memory latency tolerance techniques
- Non-volatile memory

- Multiprocessors
- Coherence and consistency
- Interconnection networks
- Multi-core issues (e.g., heterogeneous multi-core)

# 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.
  - Charachorloo 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
- Disadvantage: 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

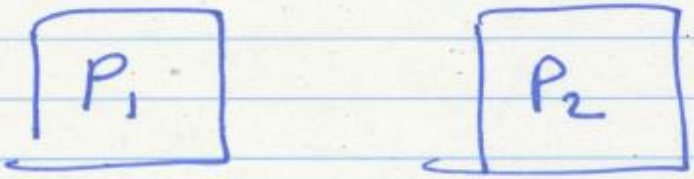
- Threads are not allowed to update shared data concurrently
  - For correctness purposes
- Accesses to shared data are encapsulated inside *critical sections* or protected via *synchronization constructs (locks, semaphores, condition variables)*
- Only one thread can execute a critical section at a 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 $F_2 = 1$

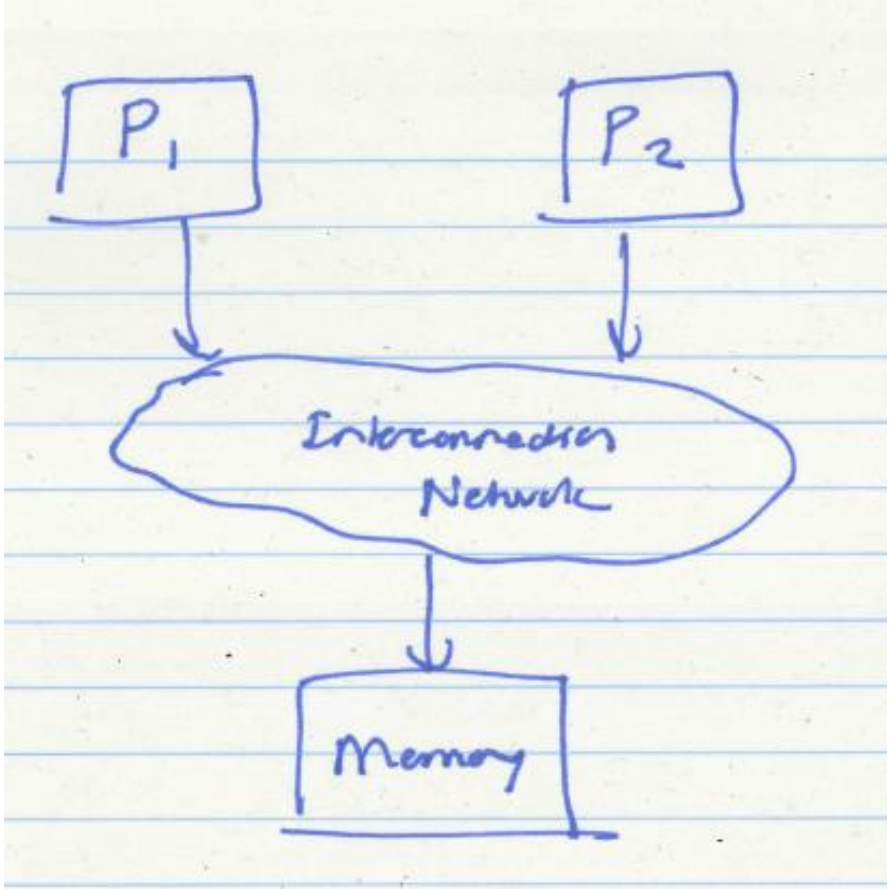
Y IF ($F_1 == \emptyset$) THEN
{Critical section}
$F2 = \emptyset$
ELSE
{ ... }

Only P1 or P2 should be in this section at any given time, not both

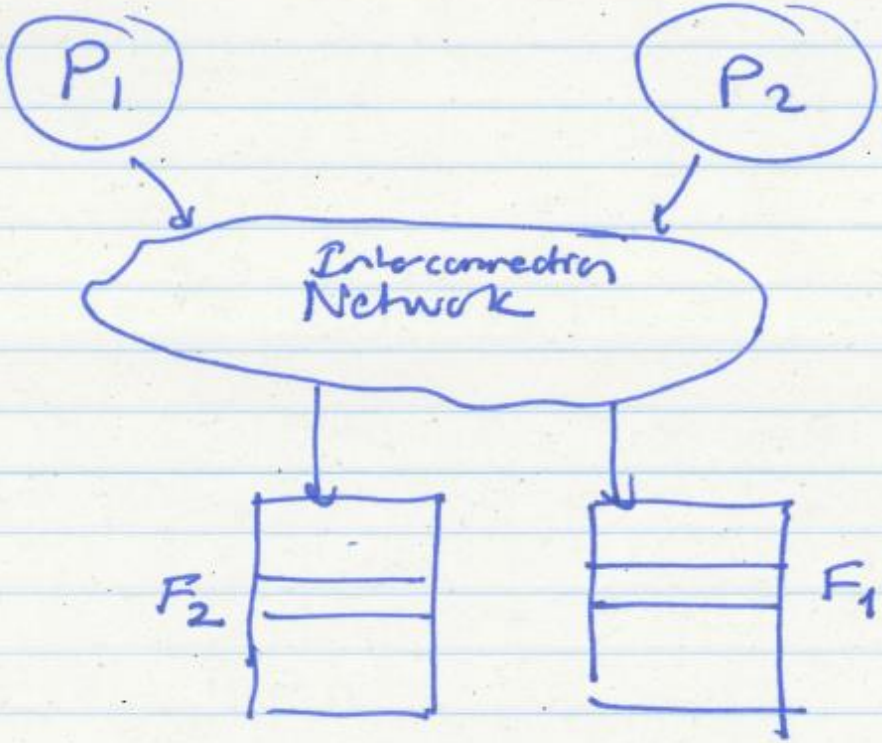
Assume P1 is in critical section.
Intuitively, it must have executed A,
which means F1 must be 1 (as A happens before B),
which means P2 should not enter the critical section.

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

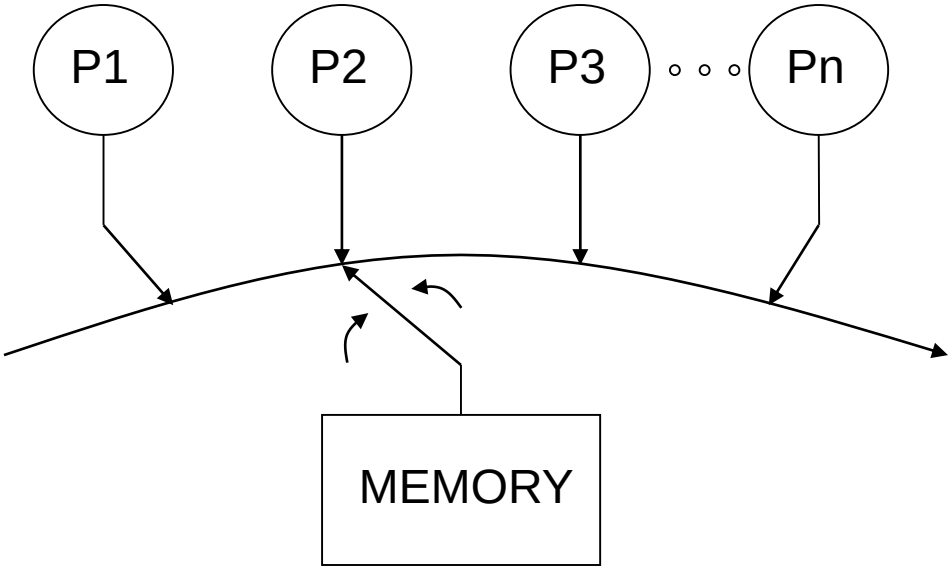
AND

  - 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 a 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

# Related Questions

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

# Caching in Multiprocessors

- Caching not only complicates ordering of all operations…
  - A memory location can be present in multiple caches
  - Prevents the effect of a store or load to be seen by other processors → makes it difficult for all processors to see the same global order of (all) *memory operations*

- … but it also complicates ordering of operations on a single memory location
  - A memory location can be present in multiple caches
  - Makes it difficult for processors that have cached the same location to have the correct value of that location (in the presence of updates to that location)

# Cache Coherence

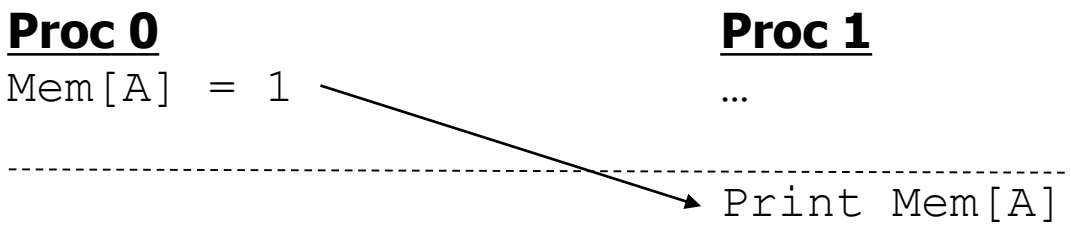
# Readings: Cache Coherence

- Required
  - Culler and Singh, *Parallel Computer Architecture*
    - Chapter 5.1 (pp 269 – 283), Chapter 5.3 (pp 291 – 305)
  - P&H, *Computer Organization and Design*
    - Chapter 5.8 (pp 534 – 538 in 4th and 4th revised eds.)
  - Papamarcos and Patel, "A low-overhead coherence solution for multiprocessors with private cache memories," ISCA 1984.

- Recommended
  - Censier and Feautrier, "A new solution to coherence problems in multicache systems," IEEE Trans. Computers, 1978.
  - Goodman, "Using cache memory to reduce processor-memory traffic," ISCA 1983.
  - Laudon and Lenoski, "The SGI Origin: a ccNUMA highly scalable server," ISCA 1997.
  - Martin et al, "Token coherence: decoupling performance and correctness," ISCA 2003.
  - Baer and Wang, "On the inclusion properties for multi-level cache hierarchies," ISCA 1988.

# Shared Memory Model

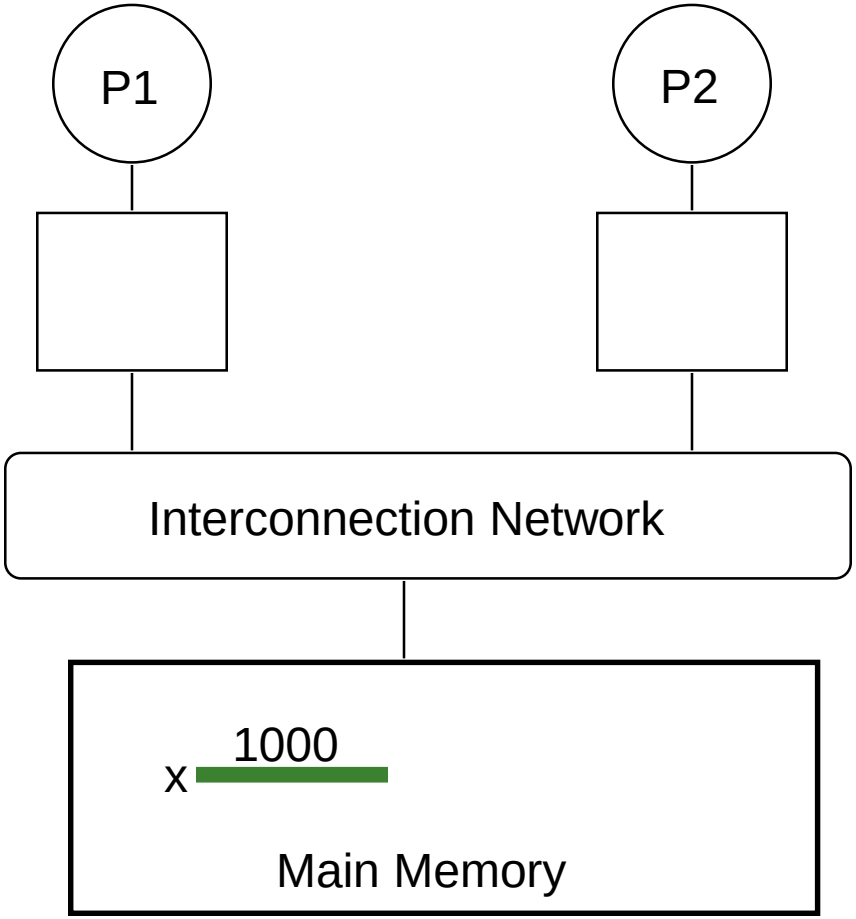
- Many parallel programs communicate through *shared memory*
- Proc 0 writes to an address, followed by Proc 1 reading
  - This implies communication between the two



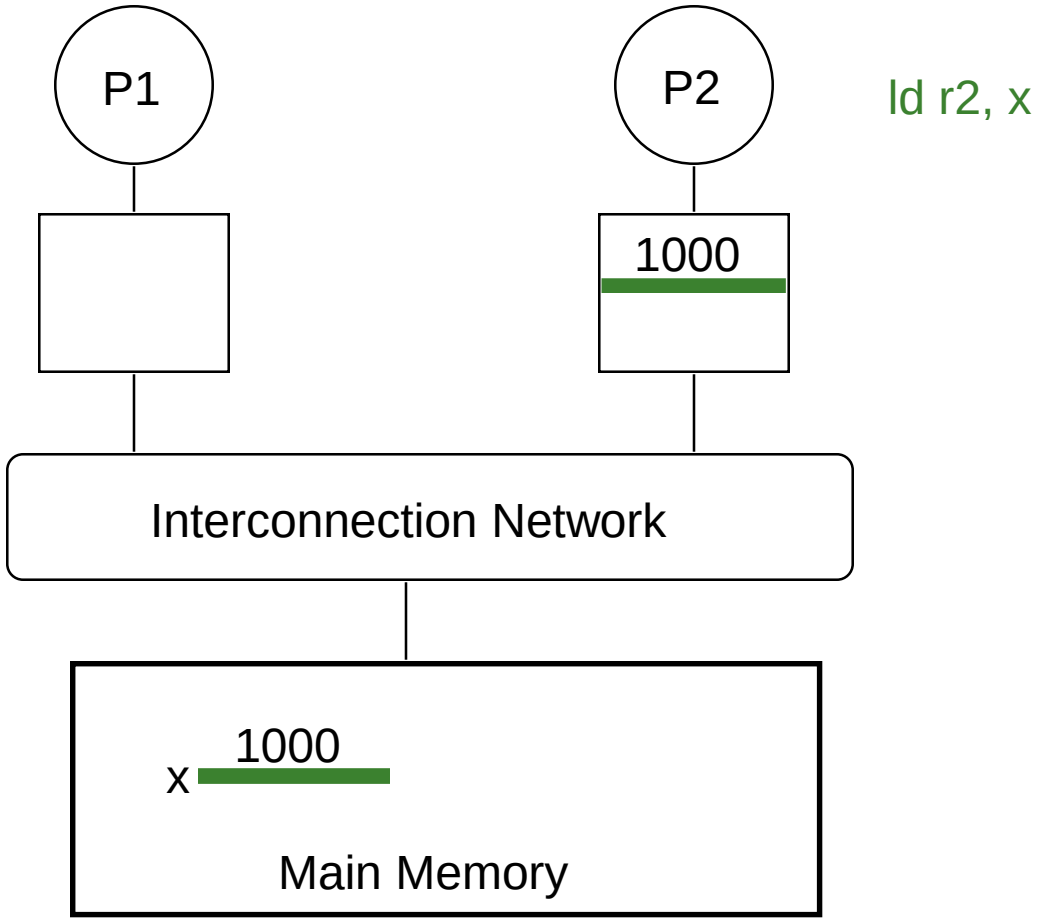

- Each read should receive the value last written by anyone
  - This requires synchronization (what does last written mean?)
- What if Mem[A] is cached (at either end)?

# Cache Coherence

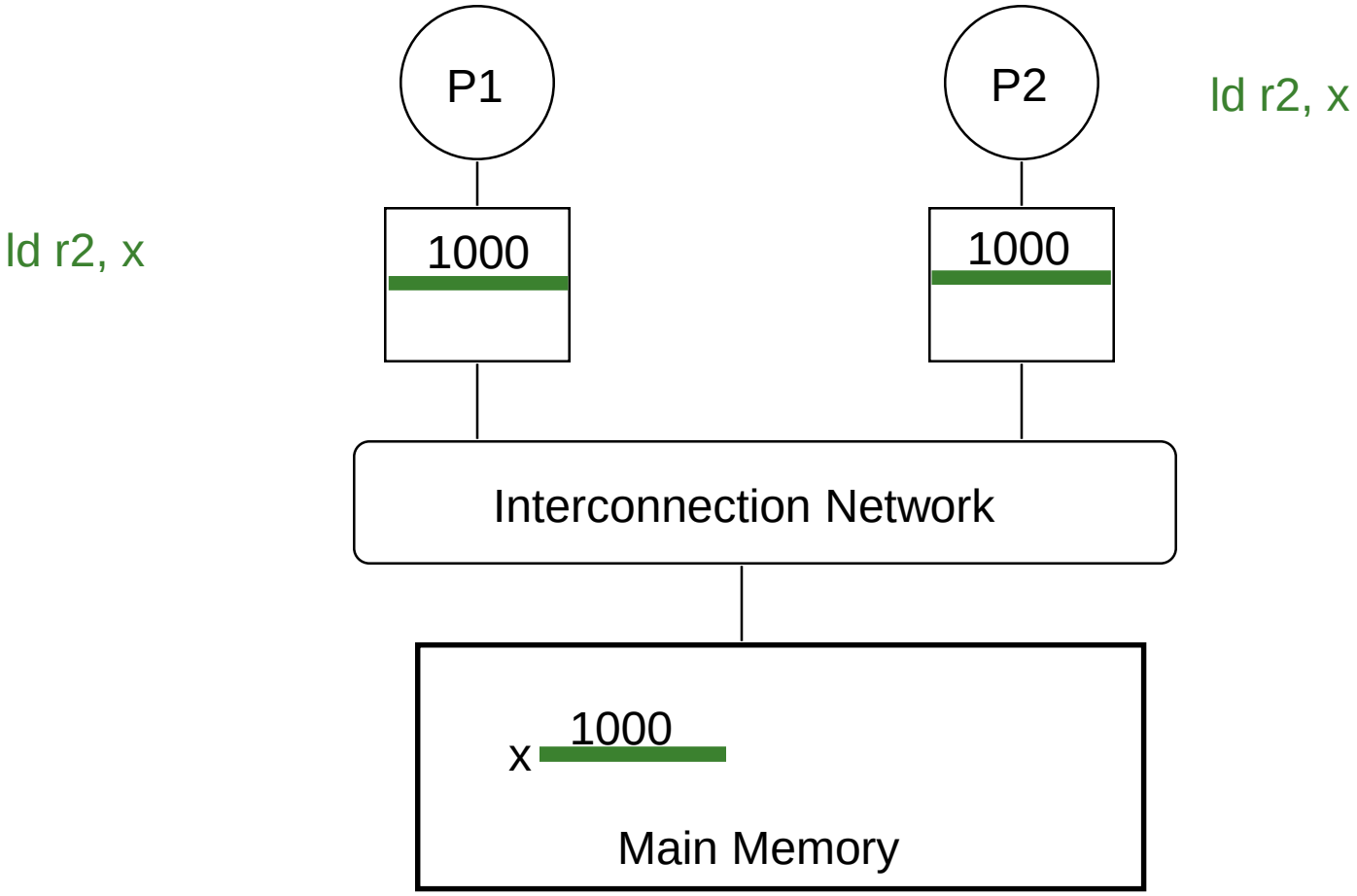
- Basic question: If multiple processors cache the same block, how do they ensure they all see a consistent state?



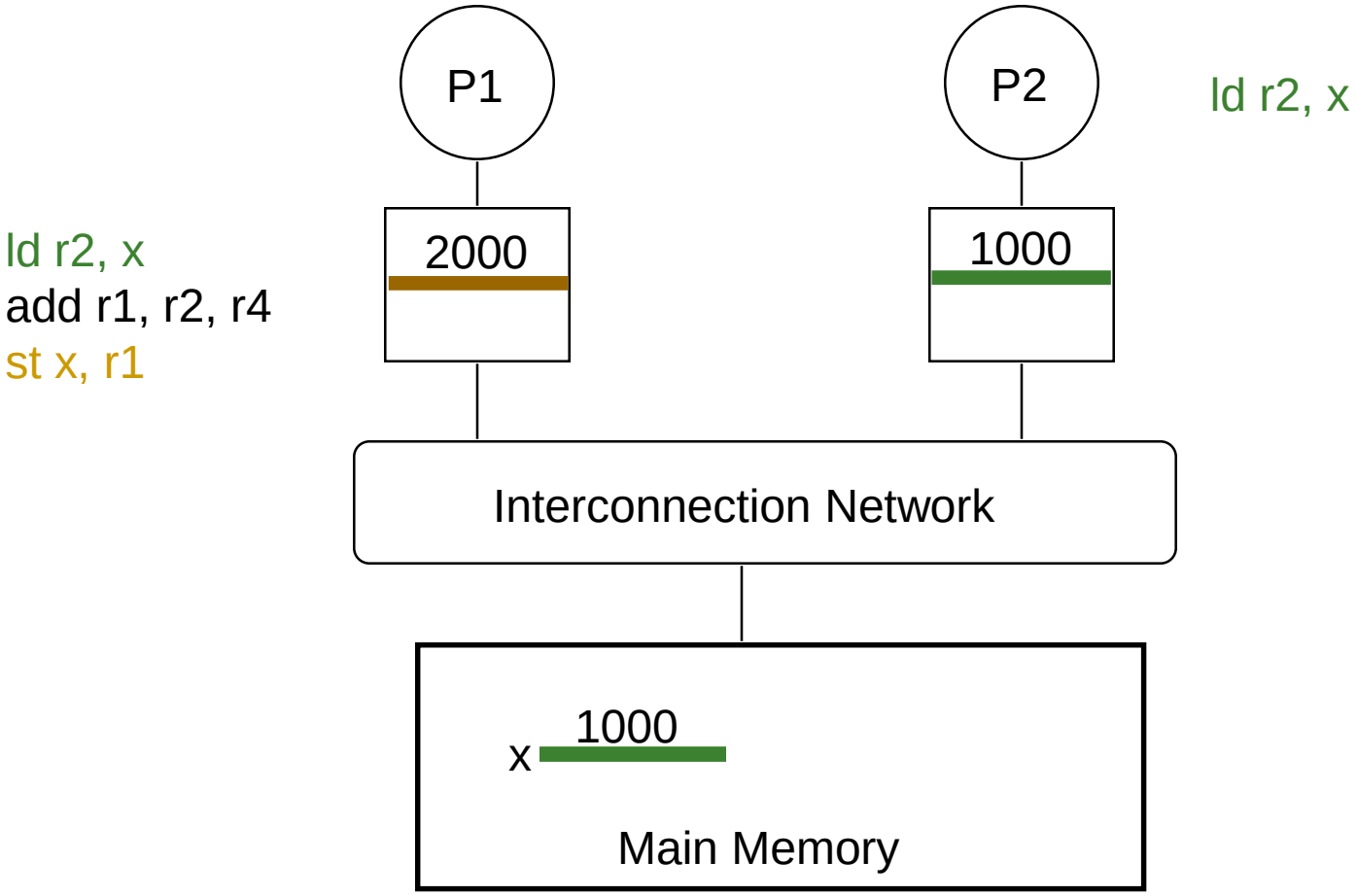
# The Cache Coherence Problem



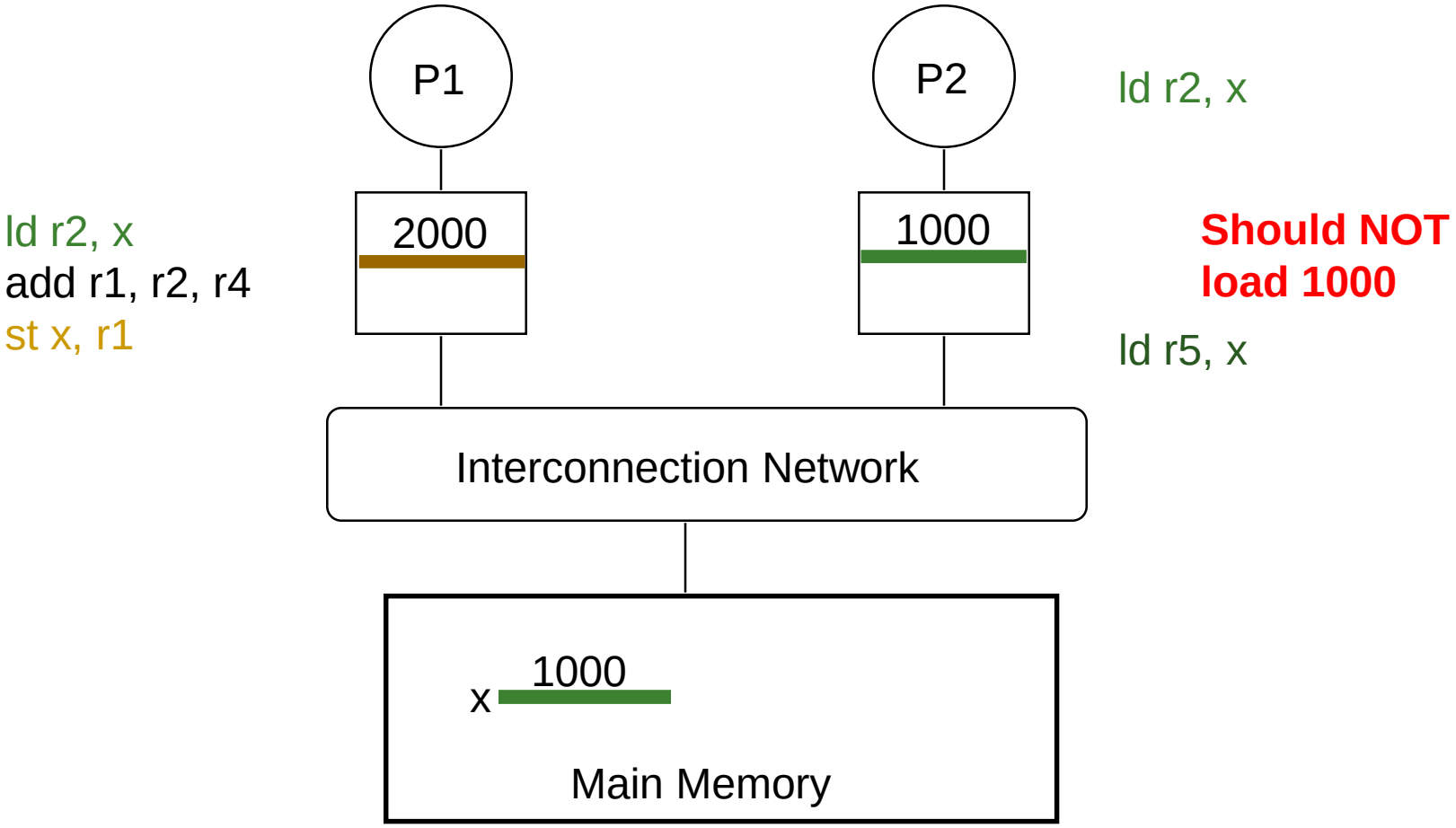
# The Cache Coherence Problem



# The Cache Coherence Problem



# The Cache Coherence Problem



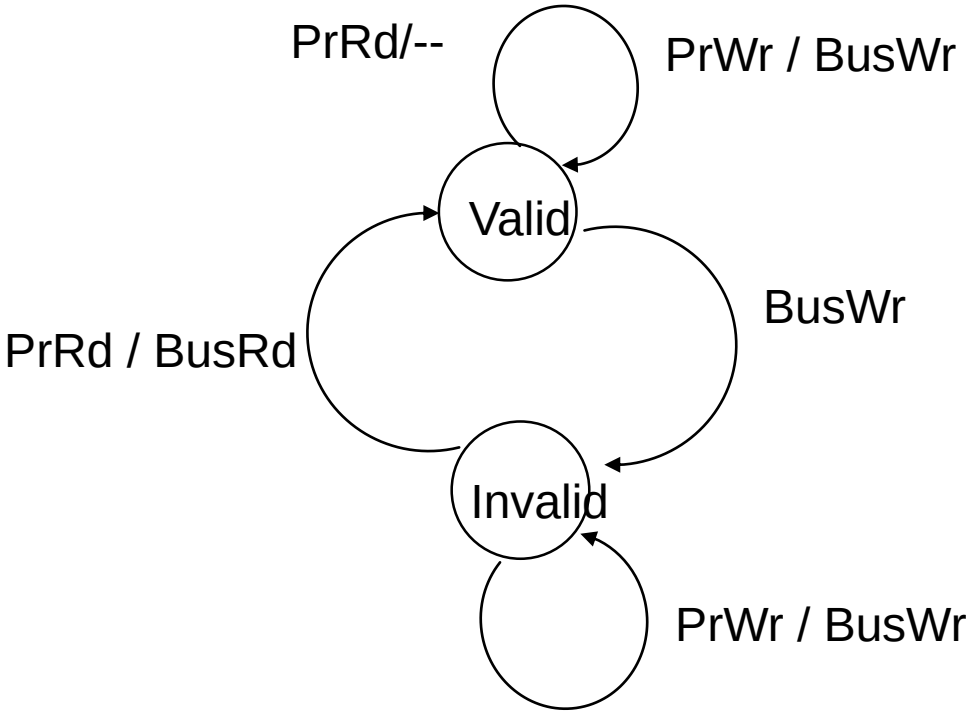
# Cache Coherence: Whose Responsibility?

- Software
  - Can the programmer ensure coherence if caches are invisible to software?
  - What if the ISA provided a cache flush instruction?
    - FLUSH-LOCAL A: Flushes/invalidates the cache block containing address A from a processor's local cache.
    - FLUSH-GLOBAL A: Flushes/invalidates the cache block containing address A from all other processors' caches.
    - FLUSH-CACHE X: Flushes/invalidates all blocks in cache X.

- Hardware
  - Simplifies software's job
  - One idea: Invalidate all other copies of block A when a processor writes to it

# A Very Simple Coherence Scheme (VI)

- Caches "snoop" (observe) each other's write/read operations. If a processor writes to a block, all others invalidate the block.
- A simple protocol:




- Write-through, no-write-allocate cache
- Actions of the local processor on the cache block: PrRd, PrWr,
- Actions that are broadcast on the bus for the block: BusRd, BusWr

# (Non-)Solutions to Cache Coherence

- No hardware based coherence
  - Keeping caches coherent is software's responsibility
  - + Makes microarchitect's life easier
  - -- Makes average programmer's life much harder
    - need to worry about hardware caches to maintain program correctness?
  - -- Overhead in ensuring coherence in software (e.g., page protection and page-based software coherence)

- All caches are shared between all processors
  - + No need for coherence
  - -- Shared cache becomes the bandwidth bottleneck
  - -- Very hard to design a scalable system with low-latency cache access this way

# Maintaining Coherence

- Need to guarantee that all processors see a consistent value (i.e., consistent updates) for the same memory location

- Writes to location A by P0 should be seen by P1 (eventually), and all writes to A should appear in some order

- Coherence needs to provide:
  - **Write propagation**: guarantee that updates will propagate
  - **Write serialization**: provide a consistent global order seen by all processors

- Need a global point of serialization for this store ordering

# Hardware Cache Coherence

- Basic idea:
  - A processor/cache broadcasts its write/update to a memory location to all other processors
  - Another cache that has the location either updates or invalidates its local copy

# Coherence: Update vs. Invalidate

- How can we *safely update replicated data?*
  - Option 1 (Update protocol): push an update to all copies
  - Option 2 (Invalidate protocol): ensure there is only one copy (local), update it

- **On a Read:**
  - If local copy is Invalid, put out request
  - (If another node has a copy, it returns it, otherwise memory does)

# Coherence: Update vs. Invalidate (II)

- **On a Write:**
  - Read block into cache as before

**Update Protocol:**

- Write to block, and simultaneously broadcast written data and address to sharers
- (Other nodes update the data in their caches if block is present)

**Invalidate Protocol:**

- Write to block, and simultaneously broadcast invalidation of address to sharers
- (Other nodes invalidate block in their caches if block is present)

# Update vs. Invalidate Tradeoffs

- Which do we want?
  - Write frequency and sharing behavior are critical
- **Update**
  - \+ If sharer set is constant and updates are infrequent, avoids the cost of invalidate-reacquire (broadcast update pattern)
  - – If data is rewritten without intervening reads by other cores, updates were useless
  - – Write-through cache policy ➔ bus becomes bottleneck
- **Invalidate**
  - \+ After invalidation broadcast, core has exclusive access rights
  - \+ Only cores that keep reading after each write retain a copy
  - – If write contention is high, leads to ping-ponging (rapid mutual invalidation-reacquire)

# Two Cache Coherence Methods

- How do we ensure that the proper caches are updated?

- **Snoopy Bus** [Goodman ISCA 1983, Papamarcos+ ISCA 1984]
  - Bus-based, single point of serialization *for all memory requests*
  - Processors observe other processors' actions
    - E.g.: P1 makes "read-exclusive" request for A on bus, P0 sees this and invalidates its own copy of A

- **Directory** [Censier and Feautrier, IEEE ToC 1978]
  - Single point of serialization *per block*, distributed among nodes
  - Processors make explicit requests for blocks
  - Directory tracks which caches have each block
  - Directory coordinates invalidation and updates
    - E.g.: P1 asks directory for exclusive copy, directory asks P0 to invalidate, waits for ACK, then responds to P1

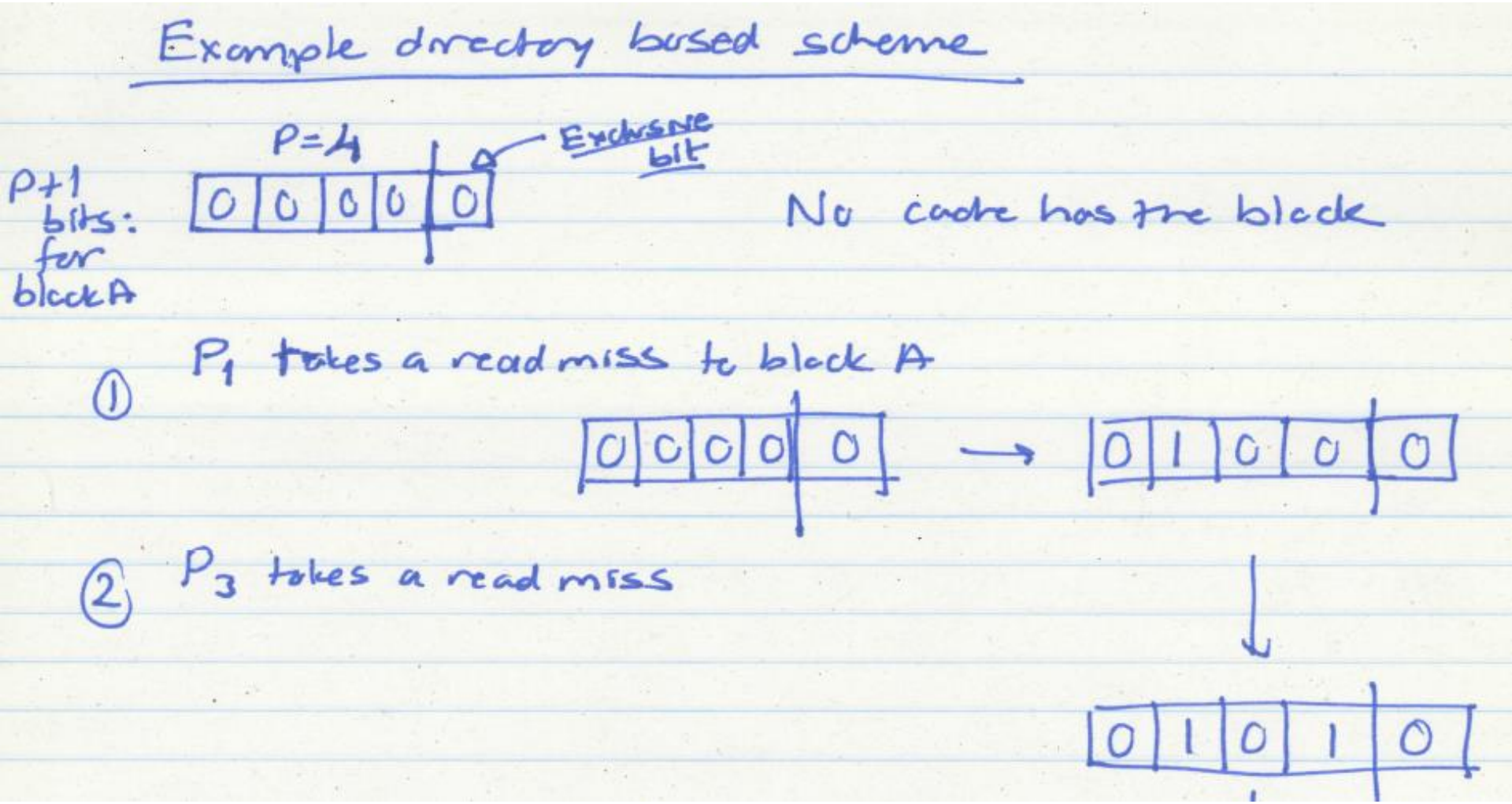
# Directory Based Cache Coherence

# Directory Based Coherence

- Idea: A logically-central directory keeps track of where the copies of each cache block reside. Caches consult this directory to ensure coherence.

- An example mechanism:
  - For each cache block in memory, store P+1 bits in directory
    - One bit for each cache, indicating whether the block is in cache
    - Exclusive bit: indicates that a cache has the only copy of the block and can update it without notifying others
  - On a read: set the cache's bit and arrange the supply of data
  - On a write: invalidate all caches that have the block and reset their bits
  - Have an "exclusive bit" associated with each block in each cache (so that the cache can update the exclusive block silently)

# Directory Based Coherence Example (I)



③ $P_2$ takes a write miss

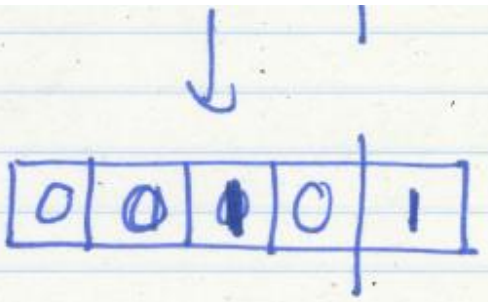
→ Invalidate $P_1$ & $P_3$'s caches

→ write request → $P_2$ has the exclusive copy of the block now. Set the Exclusive bit

→ $P_2$ can now update the block without notifying any other processor or the directory

→ $P_2$ needs to have a bit in its cache indicating it can perform exclusive updates to that block

→ private/exclusive bit per cache block

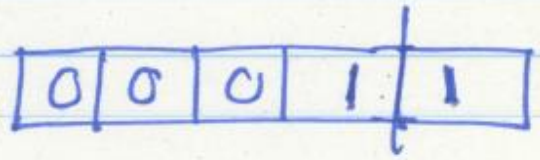



④ $P_3$ takes a write miss

→ Mem ~~Ctrller~~ Controller requests ~~true~~ block from $P_2$

→ Mem Controller gives block to $P_3$

→ P2 invalidates its copy




⑤ $P_2$ takes a read miss

→ $P_3$ supplies it