Bus-Based Coherent Multiprocessors

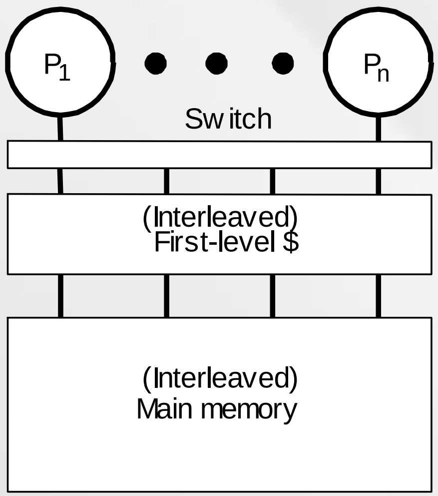
Lecture 13

(Chapter 7)

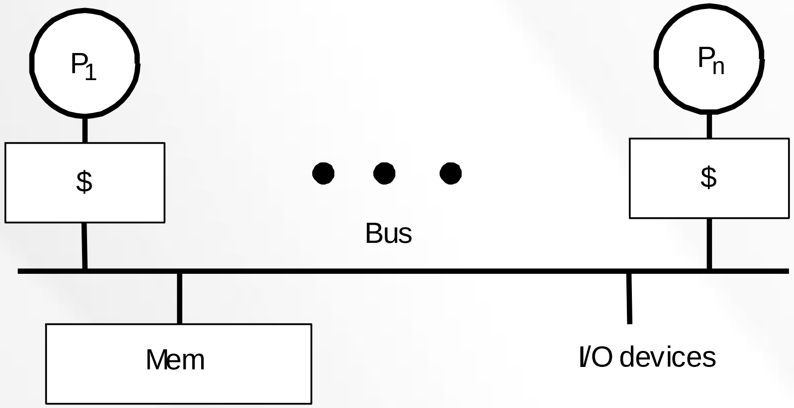
Outline

- **Bus-based coherence**
- Memory consistency
  - Sequential consistency
- Invalidation vs. update coherence protocols

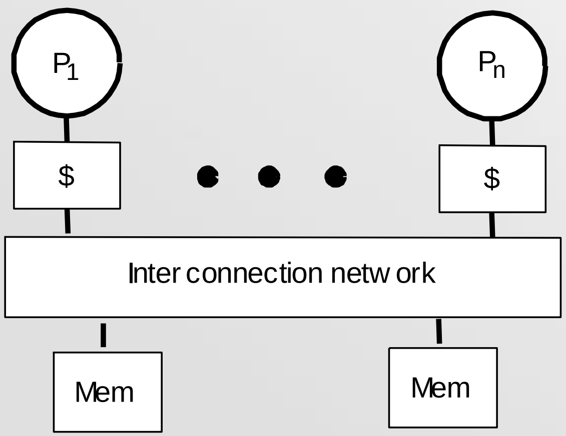
Several Configurations for a Memory System



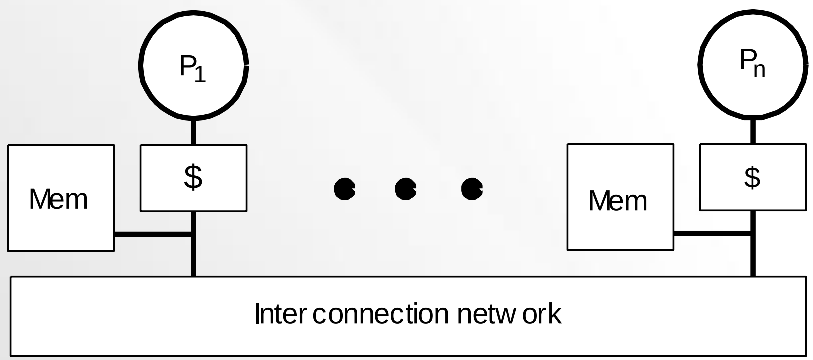
(a) Shared cache



(b) Bus-based shared memory



(c) Dancehall



(d) Distributed-memory

Assume a Bus-Based SMP

- Built on top of two fundamentals of uniprocessor system
  - Bus transactions
  - Cache-line finite-state machine
- Uniprocessor bus transaction:
  - Three phases: arbitration, command/address, data transfer
  - All devices observe addresses, one is responsible
- Uniprocessor cache states:
  - Every cache line has a finite-state machine
  - In WT+write no-allocate: Valid, Invalid states
  - WB: Valid, Invalid, Modified ("Dirty")
- Multiprocessors extend both these somewhat to implement coherence

Snoop-Based Coherence on a Bus

- *Basic Idea*
  - Assign a snooper to each processor so that all bus transactions are visible to all processors ("snooping").
  - Processors (via cache controllers) change line states on relevant events.

Snoop-Based Coherence on a Bus

- *Basic Idea*
  - Assign a snooper to each processor so that all bus transactions are visible to all processors (“snooping”).
  - Processors (via cache controllers) change line states on relevant events.

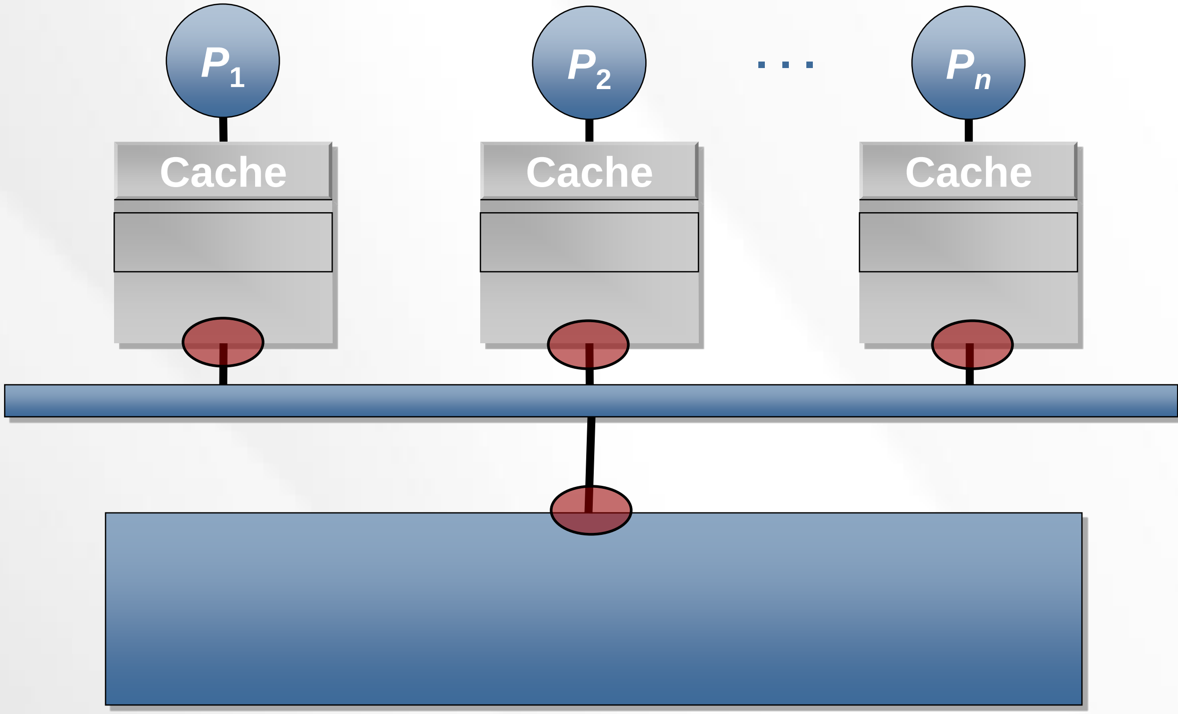
- *Implementing a Protocol*
  - Each **cache controller** reacts to processor and bus events:
    - Takes actions when necessary
      - Updates state, responds with data, generates new bus transactions
  - The **memory controller** also snoops bus transactions and returns data only when needed
  - Granularity of coherence is typically cache line/block
    - Same granularity as in transfer to/from cache

Coherence with Write-Through Caches

```
sum = 0;
begin parallel
for (i=0; i<2; i++) {
  lock(id, myLock);
  sum = sum + a[i];
  unlock(id, myLock);
end parallel
Print sum;

Suppose a[0] = 3 and a[1] = 7
```

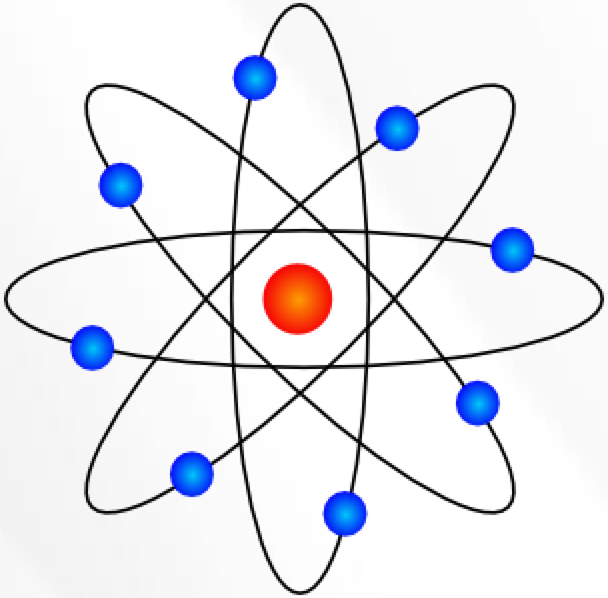
= Snooper



– What happens when we snoop a write?
  - Write-update protocol: write is immediately propagated ***or***
  - Write-invalidation protocol: causes miss on later access, and memory up-to-date via write-through

Snooper Assumptions

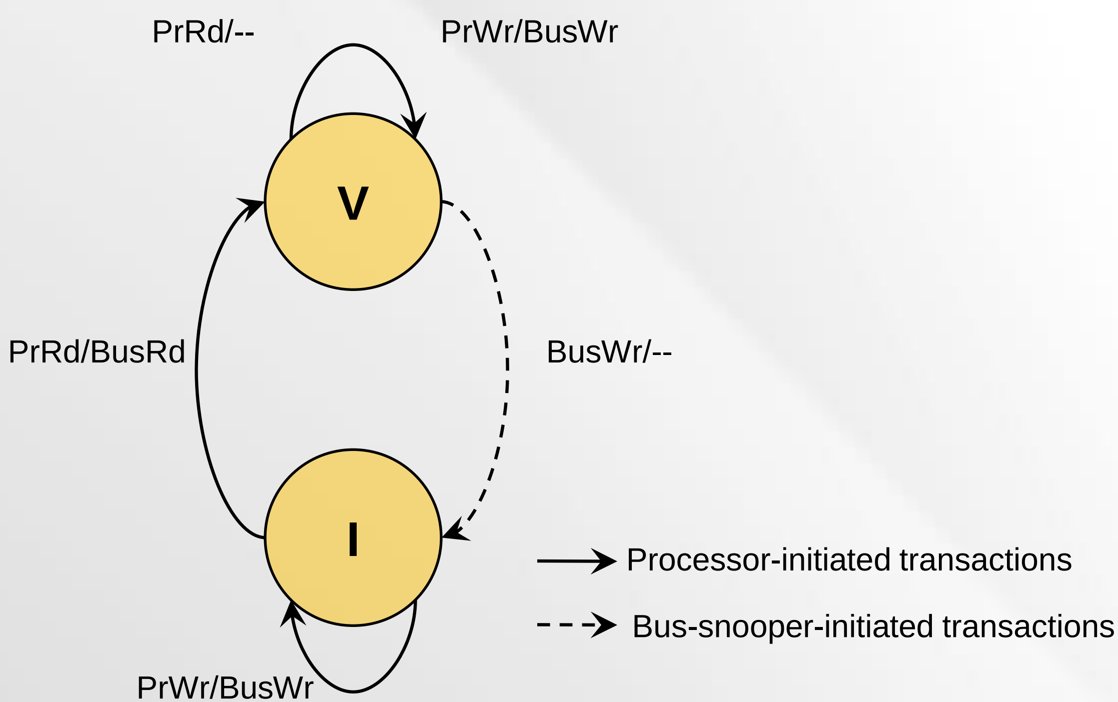
- Atomic bus
- Writes occur in program order



Transactions

- To show what's going on, we will use diagrams involving—
  - Processor transactions
    - PrRd
    - PrWr
  - Snooped bus transactions
    - BusRd
    - BusWr

Write-Through State-Transition Diagram



write-through
no-write-allocate
write invalidate

How does this protocol guarantee write propagation?

How does it guarantee write serialization?

- Key: A write invalidates all other caches
- Therefore, we have:
  - Modified line: exists as V in only 1 cache
  - Clean line: exists as V in at least 1 cache
  - Invalid state represents invalidated line or not present in the cache

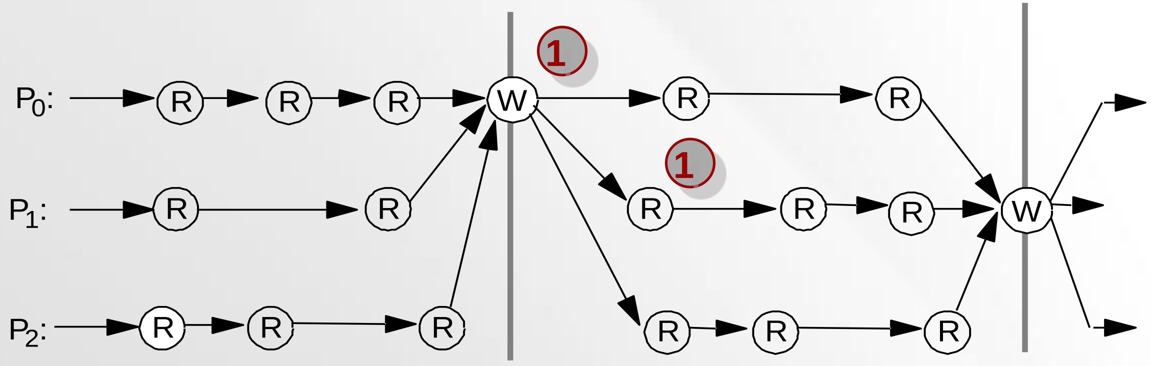
Is It Coherent?

- Write propagation:
  - through ____________
  - then a _____________ , loading the new value
- Write serialization: Assume—
  - atomic bus
  - invalidation happens instantly
  - writes serialized by _______________________
    - So are invalidations
- Do reads see the latest writes?
  - Read misses generate bus transactions, so will get the last write
  - Read hits: do not appear on bus, but are preceded by
    - most recent write by this processor (self), or
    - most recent read miss by this processor
  - Thus, reads hits see latest written values (according to bus order)

Determining Orders More Generally

A memory operation M2 follows a memory operation M1 if the operations are issued by the same processor and M2 follows M1 in program order.

1. Read follows write W if read generates bus transaction that follows W's xaction.

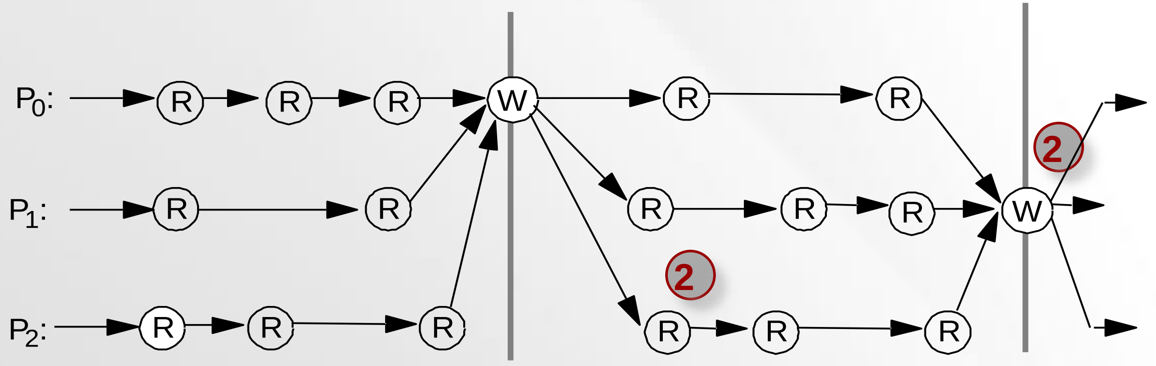


- Writes establish a partial order
- Doesn't constrain ordering of reads, though bus will order read misses too
  - –any order among reads between writes is fine, as long as in program order

Determining Orders More Generally

A memory operation M2 follows a memory operation M1 if the operations are issued by the same processor and M2 follows M1 in program order.

1. Read follows write W if read generates bus transaction that follows W's xaction.
2. Write follows read or write M if M generates bus transaction and the transaction for the write follows that for M.



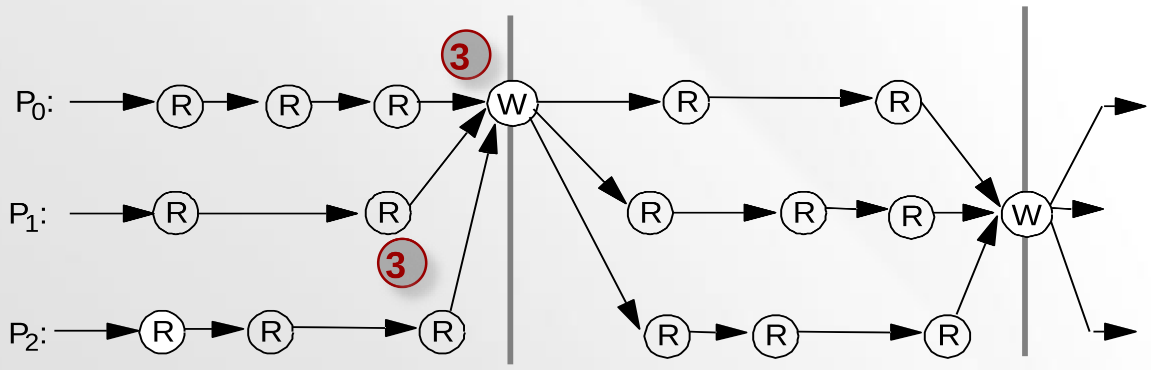
- Writes establish a partial order
- Doesn't constrain ordering of reads, though bus will order read misses too

–any order among reads between writes is fine, as long as in program order

Determining Orders More Generally

A memory operation M2 follows a memory operation M1 if the operations are issued by the same processor and M2 follows M1 in program order.

1. Read follows write W if read generates bus transaction that follows W's xaction.
2. Write follows read or write M if M generates bus transaction and the transaction for the write follows that for M.
3. Write follows read if read does not generate a bus transaction and is not already separated from the write by another bus transaction.



- Writes establish a partial order
- Doesn't constrain ordering of reads, though bus will order read misses too
  – any order among reads between writes is fine, as long as in program order

Problem with Write-Through

- Write-through can guarantee coherence, but needs a lot of bandwidth.
  - Every write goes to the shared bus and memory
  - Example:

    200MHz, 1-CPI processor, and 15% instrs. are 8-byte stores

    Each processor generates 30M stores, or 240MB data, per second

    How many processors could a 1GB/s bus support without saturating?
  - Thus, unpopular for SMPs

- Write-back caches
  - Write hits do not go to the bus ⇒ reduce most write bus transactions
  - But now how do we ensure write propagation and serialization?

Lecture 11 Outline

- Bus-based coherence
- **Memory consistency**
  - Sequential consistency
- Invalidation vs. update coherence protocols

Let’s Switch Gears to Memory Consistency

Coherence: Writes to a single location are visible to all in the same order

Consistency: Writes to multiple locations are visible to all in the same order

- Recall Peterson’s algorithm (**`turn= …; interested[process]=…`**)
- When “multiple” means “all”, we have sequential consistency (SC)

P_1	P_2
/*Assume initial values of A and flag are 0*/	
`A = 1;`	`while (flag == 0);` **/*spin idly*/**
`flag = 1;`	`print A;`

- Sequential consistency (SC) corresponds to our intuition.
- Other memory consistency models do not obey our intuition!
- Coherence doesn’t help; it pertains only to a single location

Another Example of Ordering

P_1	P_2
/*Assume initial values of **`A`** and **`B`** are **`0`** */	
(1a) `A = 1;`	(2a) `print B;`
(1b) `B = 2;`	(2b) `print A;`

- What do you think the results should be? You may think:
  - 1a, 1b, 2a, 2b ⇒ `{A=1, B=2}`
  - 1a, 2a, 2b, 1b ⇒ `{A=1, B=0}`
  - 2a, 2b, 1a, 1b ⇒ `{A=0, B=0}`

  } programmers' intuition: sequential consistency
  - Is `{A=0, B=2}` possible? Yes, suppose P2 sees: 1b, 2a, 2b, 1a e.g. evil compiler, evil interconnection.
- Whatever our intuition is, we need
  - an **ordering model** for clear semantics across different locations
  - as well as **cache coherence**!

so programmers can reason about what results are possible.

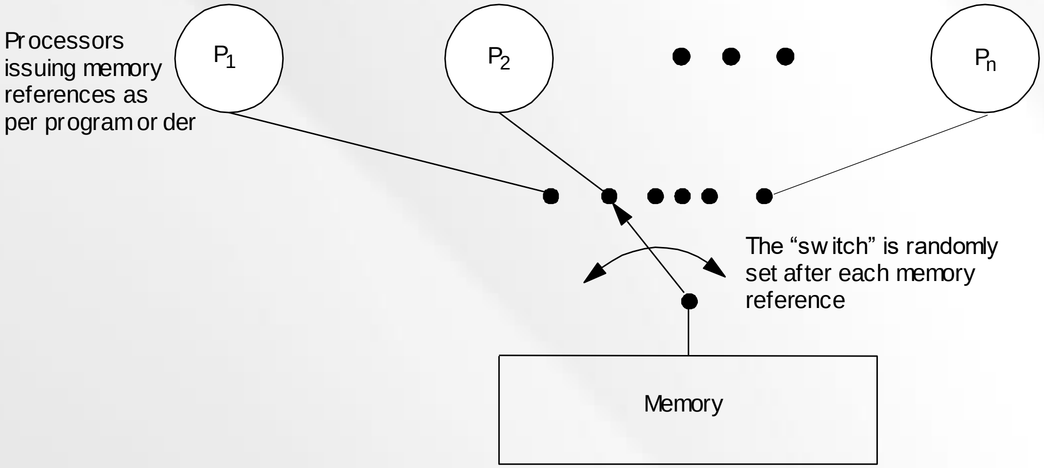
A Memory-Consistency Model …

- Is a contract between programmer and system
  - Necessary to reason about correctness of shared-memory programs
- Specifies constraints on the order in which memory operations (from any process) can *appear to execute* with respect to one another
  - Given a load, constrains the possible values returned by it
- Implications for programmers
  - Restricts algorithms that can be used
  - e.g., Peterson’s algorithm, home-brew synchronization will be incorrect in machines that do not guarantee SC
- Implications for compiler writers and computer architects
  - Determines how much accesses can be reordered.

Lecture 11 Outline

- Bus-based coherence
- Memory consistency
  - **Sequential consistency**
- Invalidation vs. update coherence protocols

Sequential Consistency



"A multiprocessor 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." [Lamport, 1979]

- (as if there were no caches, and a single memory)
- Total order achieved by *interleaving* accesses from different processes
- Maintains *program order*, and memory operations, from all processes, appear to [issue, execute, complete] atomically w.r.t. others

What Really Is Program Order?

- Intuitively, the order in which operations appear in source code

- *Thus, we assume order as seen by programmer,*
  - *the compiler is prohibited from reordering memory accesses to shared variables.*
- Note that this is one reason parallel programs are less efficient than serial programs.

What Reordering Is Safe in SC?

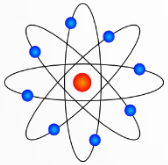
What matters is the order in which code *appears to execute*, not the order in which it actually *executes.*

P_1	P_2
/*Assume initial values of **A** and **B** are **0** */	
(1a) `A = 1;`	(2a) `print B;`
(1b) `B = 2;`	(2b) `print A;`

- Possible outcomes for `(A,B)`: `(0,0)`, `(1,0)`, `(1,2)`; impossible under SC: `(0,2)`
- *Proof:* By program order we know 1a → 1b and 2a → 2b
  - `A = 0` implies 2b → 1a, which implies 2a → 1b
  - `B = 2` implies 1b → 2a, which leads to a contradiction
- BUT, actual execution 1b →1a → 2b → 2a is SC, despite not being in program order
  - It produces the same result as 1a → 1b → 2a → 2b.
  - Actual execution 1b → 2a → 2b → 1a is not SC, as shown above
  - Thus, some reordering is possible, but difficult to reason that it ensures SC

Conditions for SC

- Two kinds of requirements
  - **Program order**
    - Memory operations issued by a process must appear to become visible (to others and itself) in program order.
  - **Global order**
    - Atomicity: One memory operation should appear to complete with respect to all processes before the next one is issued.
    - Global order: The same order of operations is seen by all processes.



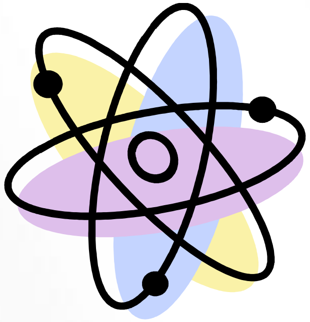
- Tricky part: how to make writes atomic?
  - ➔ Necessary to detect write completion
  - Read completion is easy: a read completes when the data returns
- Who should enforce SC?
  - Compiler should not change program order
  - Hardware should ensure program order and atomicity

Write Atomicity

- *Write Atomicity* ensures same write ordering is seen by all procs.
  - In effect, extends write serialization to writes from multiple processes

P_1	P_2	P_3
`A=1;` →	`while (A==0);`	
	`B=1;` →	`while (B==0);`
		`print A;`

- Under SC, transitivity implies that **`A`** should print as 1. Without SC, why might it not?



Is the Write-Through Example SC?

- Assume no write buffers, or load-store bypassing
- Yes, it is SC, because of the atomic bus:
  - Any write and read misses (to *all locations)* are serialized by the bus into bus order.
  - If a read obtains value of write W, W is guaranteed to have completed since it caused a bus transaction
  - When write W is performed *with respect to any processor*, all previous writes in bus order have completed

Lecture 11 Outline

- Bus-based coherence
- Memory consistency
  - Sequential consistency
- **Invalidation vs. update coherence protocols**

Dealing with "Dirty" Lines

- What does it mean to say a cache line is "dirty"?
  - That at least one of its words has been changed since it was brought in from main memory.
- Dirty in a uniprocessor vs. a multiprocessor
  - Uniprocessor:
    - Only need to keep track of *whether* a line has been modified.
  - Multiprocessor:
    - Keep track of *whether* line is modified.
    - Keep track of which cache *owns* the line.
  - Thus, a cache line must know whether it is—
    - **Exclusive**: "I'm the only one that has it, other than possibly main memory."
    - The **Owner**: "I'm responsible for supplying the block upon a request for it."

Invalidation vs. Update Protocols

- Question: What happens to a line if *another* processor changes one of its words?
  - It can be *invalidated.*

  - It can be *updated.*

Invalidation-Based Protocols

- Idea: When I write the block, invalidate everybody else
  ⇒ I get exclusive state.
- "Exclusive" means …
  - Can modify without notifying anyone else (i.e., without a bus transaction)
- But, before writing to it,
  - Must first get block in exclusive state
  - Even if block is already in state V, a bus transaction (Read Exclusive = RdX) is needed to invalidate others.
- What happens on a writeback
  - if the block is not dirty?
  - if the block *is* dirty?

UPDATE!-Based Protocols

- Idea: If this block is written, send the new word to all other caches.
  - New bus transaction: Update
- Compared to invalidate, what are advs. and disads.?
- Advantages
  - Other processors don’t miss on next access
  - Saves refetch: In invalidation protocols, they would miss & bus transaction.
  - Saves bandwidth: A single bus transaction updates several caches
- Disadvantages
  - Multiple writes by same processor cause multiple update transactions
    - In invalidation, first write gets exclusive ownership, other writes local

Invalidate versus Update

- Is a block written by one processor read by other processors before it is rewritten?
- Invalidation:
  - Yes → Readers will take a miss.
  - No → Multiple writes can occur without additional traffic.
    - Copies that won't be used again get cleared out.
- Update:
  - Yes → Readers will not miss if they had a copy previously
    - A single bus transaction will update all copies
  - No → Multiple useless updates, even to dead copies
- Invalidation protocols are much more popular.
  - Some systems provide both, or even hybrid

Summary

- One solution for small-scale multiprocessors is a shared bus.
- State-transition diagrams can be used to show how a cache-coherence *protocol* operates.
  - The simplest protocol is write-through, but it has performance problems.
- Sequential consistency guarantees that memory operations are seen in order throughout the system.
  - It is fairly easy to show whether a result is or is not sequentially consistent.
- The two main types of coherence protocols are invalidate and update.
  - Invalidate usually works better, because it frees up cache lines more quickly.