Consistency vs. Coherence

[§5.2] In the preceding two lectures, we have studied coherence and consistency. What's the difference?

- Coherence assures that values written by one processor are read by other processors.
- However, coherence says nothing about *when* writes will become visible.

Another way of looking at it:

- Coherence insures that writes *to a particular location* will be seen in order.
- Consistency insures that writes *to different locations* will be seen in an order that makes sense, given the source code.

Example: Two processors are synchronizing on a variable called **`flag`**. Assume **`A`** and **`flag`** are both initialized to 0.

P_1	P_2
`A = 1;` `flag = 1;`	`while (flag == 0); /* spin */` `print A;`

What value does the programmer expect to be printed?

What could go wrong?

Is this a coherence problem?

Is this a consistency problem?

Note that even barrier synchronization can't guarantee the right order.

P_1	P_2
A = 1;	
*barrier***(b1)**	*barrier***(b1)**
	print A;

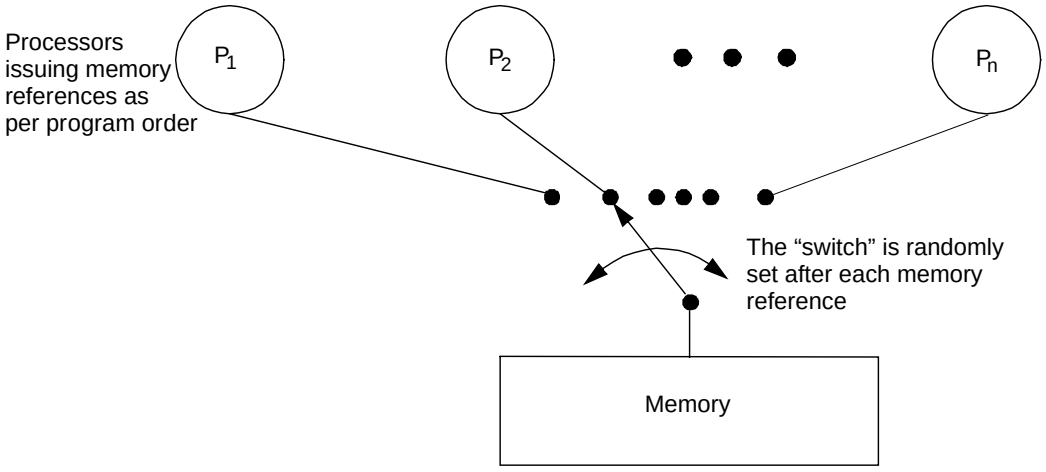
There are two problems with using this approach to guarantee that **A** is printed as 1.

- A barrier just guarantees that processes will wait there until all have arrived. It doesn't guarantee that processes wait

- A barrier may be implemented with reads and writes to shared variables. Nothing makes all processors wait until these reads and writes are "performed everywhere."

Requirements for sequential consistency

Sequential consistency assures that each process appears to issue and complete memory operations one at a time, and in sequential order.



Sequential consistency requires—

- memory operations must become visible to all processes in program order, and
- operations must appear *atomic*—one operation must complete before the next one is issued.

It is tricky to make writes seem atomic, especially when multiple copies of a word need to be updated on each write.

If writes aren't atomic, errors may result.

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

If writes aren't atomic, what might happen with the **`print`** statement?

Bus-based serialization of transactions provides not only coherence, but also consistency. With the two-state write-through invalidation protocol discussed in Lecture 12, writes and read misses to *all* locations are serialized in bus order.

- If a read obtains the value of write *W*, *W* is guaranteed to have completed—since it caused a bus transaction.
- When write *W* is performed wrt. any processor, all previous writes in bus order have completed.

Fortunately, not all operations must be performed in order. Why fortunately?

Of these operations, which must be done in order?

Write *A*
Write *B*
Read *A*
Read *B*

Snooping coherence protocols

[§5.3] Snooping protocols are simple to implement.

- No need to change processor, main memory, or cache. Only the cache controller need be extended.
- They are usable with write-back caches.
- Processors can write to their caches concurrently without any bus transactions.

If a block in a cache is dirty, how many caches may that block be cached in?

A cache is said to *own* a block if it must supply the data upon a request for the block.

We will study three protocols.

- A three-state write-back invalidation protocol (MSI).
- A four-state write-back invalidation protocol (Illinois).
- A four-state write-back update protocol (Dragon).

A three-state invalidation protocol

[§5.3.1] Our write-through protocol had two states,

In a write-back cache, we must distinguish “clean” (unmodified) blocks from “dirty” (modified) blocks. This means we need at least three states.

	Is block in cache modified?	Main memory up to date?	Other caches have copies of block?
Modified			

Shared			
Invalid			

What has to happen before a shared or invalid block can be written and placed in the modified state?

In this system, five different types of transactions are possible.

- Processor read (PrRd).
- Processor write (PrWr)

Either of these might miss. In case of a miss, a block needs to be replaced.

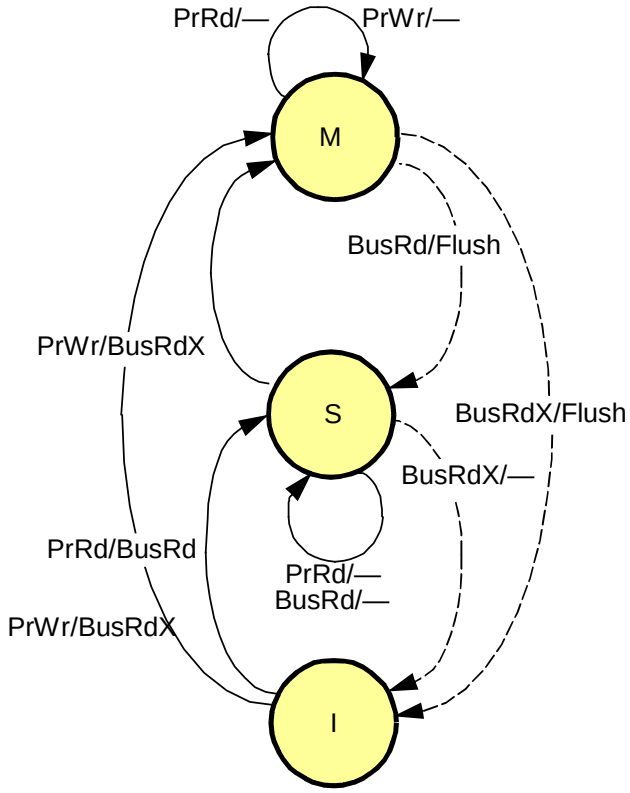
In addition to processor transactions, there are three kinds of bus transactions:

- Bus read (BusRd). This occurs when a PrRd misses in the cache. A request goes out over the bus to get the value of the block.
- Bus read exclusive (BusRdX).

- Bus Write-Back (BusWB). This is generated by

Which of these transactions is required only because of cache coherence?

Let us take a look at the state-transition diagram for this protocol.



There are three states.

The states closer to the top of the diagram are bound more tightly to the processor.

Each transition is labeled with the transaction that *causes* it and, if applicable, the transaction that it *generates*.

♦ Let us first consider the transitions out of the Invalid state.

- What happens if the processor reads a word?
- What happens if this processor writes a word?

♦ How many transitions are there out of the Shared state?

One of these transactions is a write transaction. What happens then?

Since the cache already has a copy of the block, the only effect is to invalidate any copies in other caches. The data coming back on the bus can be ignored, since it is already in the cache. (In fact, a BusUpgr transaction could be introduced to reduce bus traffic.)

The rest of the transactions are read transactions.

- If _____ reads the line, the block stays in the Shared state.
- If _____ , the block is invalidated (moves into the Invalid state).

♦ How many transitions are there out of the Modified state?

One of these transactions is a write transaction. What happens then?

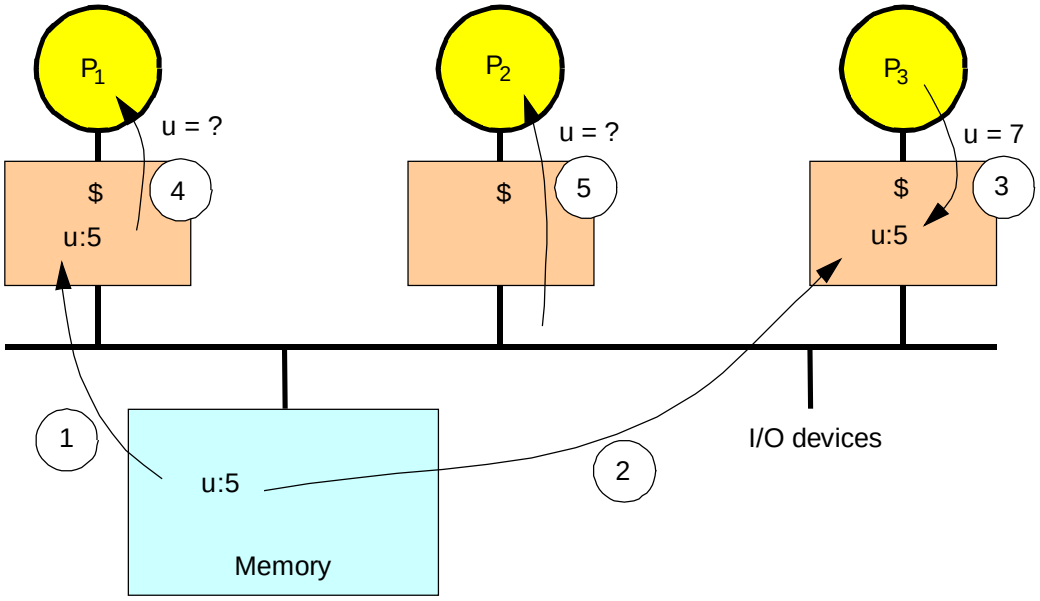
What is the simplest read transaction?

What about bus reads (reads coming from other processors)?

When a block transitions from the Modified to the Shared state, it is said to have been *demoted*. In this case, the memory and the requesting cache both grab a copy of the block from the bus.

Actually, for the sake of completeness, we should have a transistion from each state for each observable transaction. Sometimes we specify that nothing happens for these transactions (as in the _____ state); sometimes we just omit the transaction (as in the _____ state).

Given this diagram from Lecture 12, what would be the action and the state of the data in each cache?



Processor action	State in P_1	State in P_2	State in P_3	Bus action	Data supplied by
P_1 reads **u**		—	—		
P_3 reads **u**		—	—		
P_3 writes **u**					
P_1 reads **u**					
P_2 reads **u**					

To achieve coherence, this protocol must—

- Propagate all writes so they become visible by other processes.
- Serialize all writes, so they are seen in the same order by other processes.

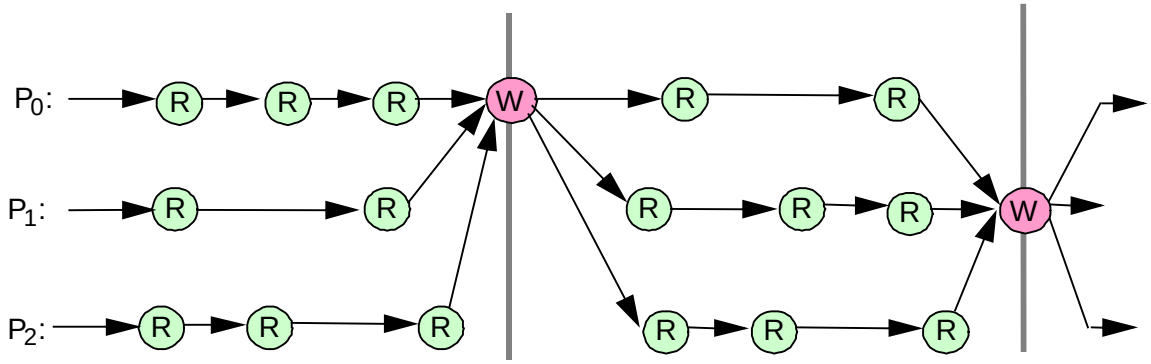
To achieve sequential consistency, this protocol must

- make memory operations visible to all processes in program order, and
- make operations appear *atomic*—one operation must complete before the next one is issued.

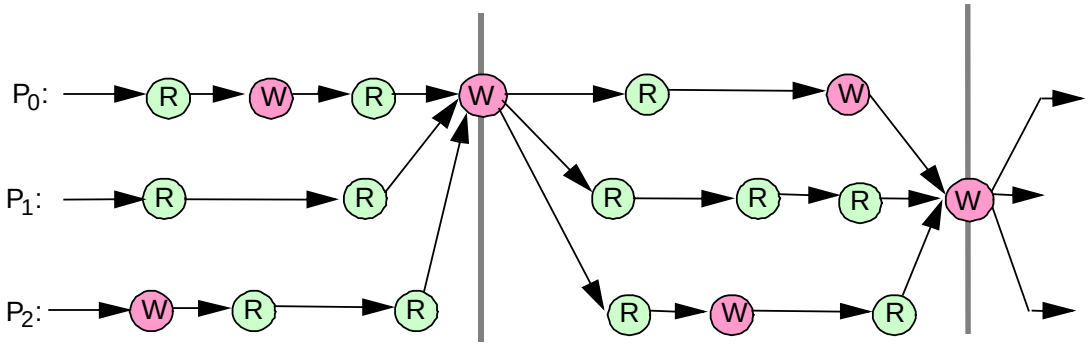
Obviously, bus transactions are totally ordered.

Between bus transactions, processors perform reads and writes in program order.

This looks the condition for merged partial orders that we considered in Lecture 12—



except that writes may be interspersed among the reads between bus transactions.



However, this is not a problem, because those writes are not observed by other processors (if they were, a bus transaction would be generated). See CS&G, p. 298 for a more complete discussion.

The four-state invalidation protocol

[§5.3.2] The three-state protocol generates more bus transactions than necessary.

If data is read, then modified, two bus transactions are generated.

- A _____ transaction (when the block moves from state _____ to state ___).
- A _____ transaction (when the block moves from state _____ to state ___).

The second transaction is generated even by serial programs, is needless, and wastes bus bandwidth.

How can we avoid it?

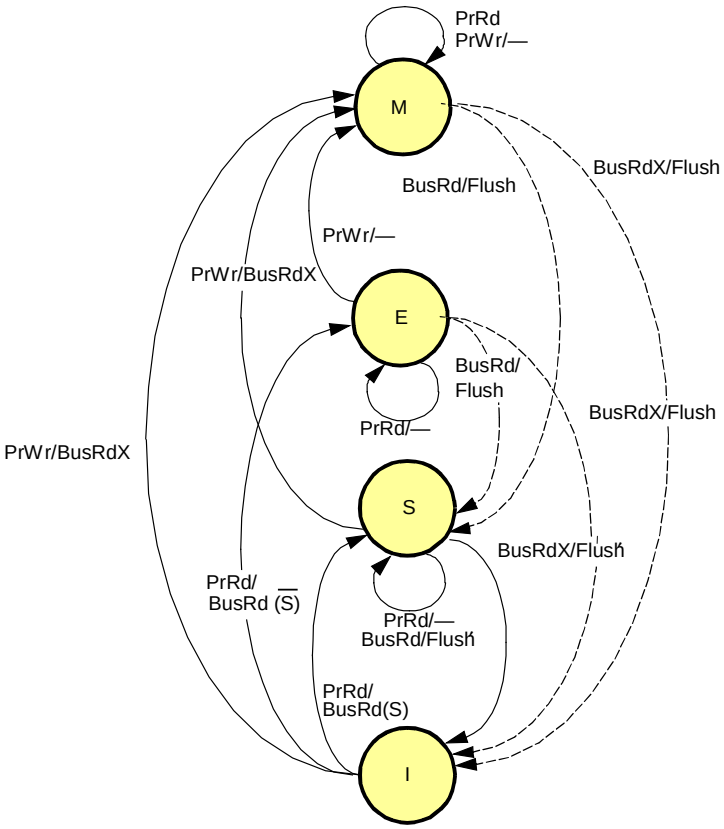
Let's call this state E, for "exclusive." It indicates that a block is in use by a single processor, but has not been modified.

Then if the block is modified, no bus transaction is necessary.

If the block is read by another processor, main memory can supply the value (the block is *unowned*).

The resulting protocol is used by several modern processors.

Here is a state-transition diagram of this protocol.



When a block is first read by a processor, it enters E or S state, depending on whether another cache has a copy.

When a processor writes a block that is in state E, it immediately transitions to state _____ without a bus transaction.

What new requirement does this protocol place on the bus?

If a block is in state S, does that mean it is actually being shared?

Who should provide the block for a BusRd transaction when both main memory and another cache have a copy of it?

Originally, in the Illinois protocol, the cache supplied it. Since caches were constructed out of SRAM, they were assumed to be faster than DRAM.

However, in modern machines, it may be more expensive to access another processor's cache.

- How does main memory know it should supply data?

- If the data resides in multiple caches,

But, on a distributed-memory machine, it may still be faster to get it from the cache.

If it is faster to transfer data from the cache, it may be a good idea to add a fifth “owned” (O) state identifying the owner, who is responsible for supplying requests for the data.