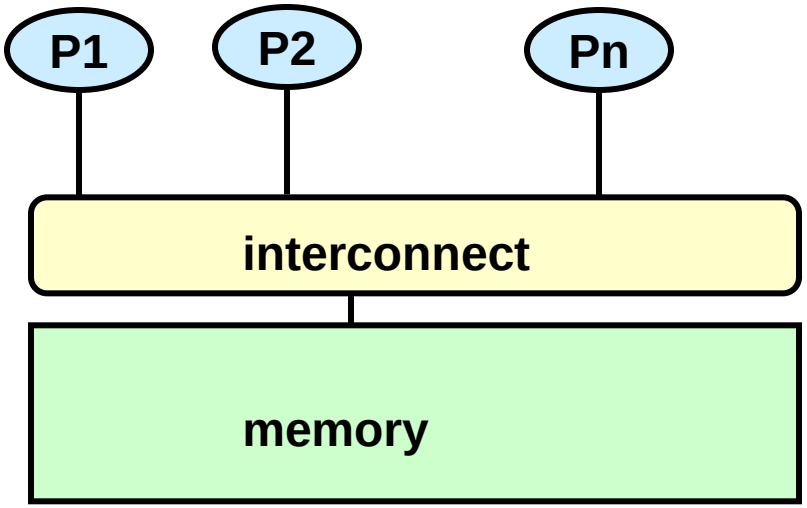
Shared Memory Hardware and Memory Consistency

Modified from J. Demmel and K. Yelick
http://www.cs.berkeley.edu/~demmel/cs267_Spr11/

Basic Shared Memory Architecture

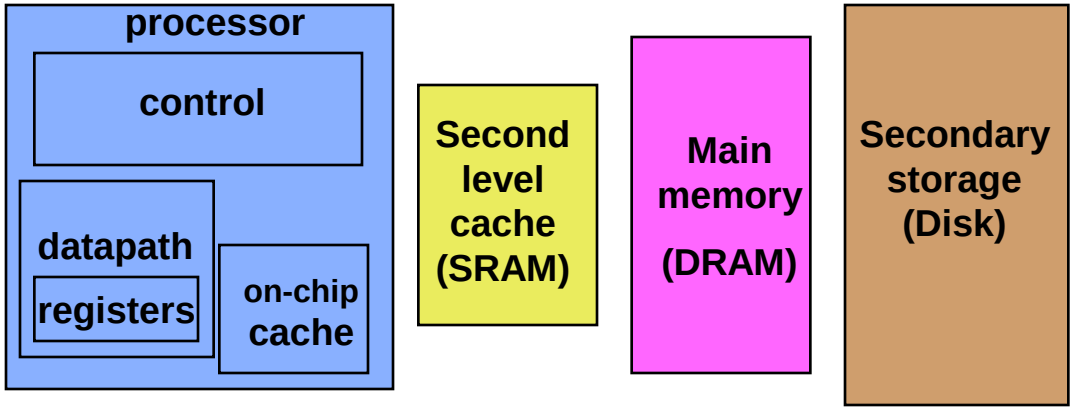
- Processors all connected to a large shared memory
  - Where are caches?



- Now take a closer look at structure, costs, limits, programming

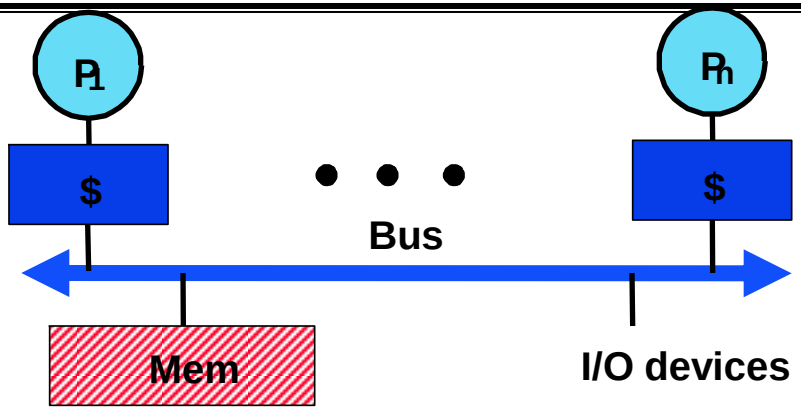
Memory Hierarchy

- Most programs have a high degree of locality in their accesses
  - **spatial locality:** accessing things nearby previous accesses
  - **temporal locality:** reusing an item that was previously accessed
- Memory hierarchy tries to exploit locality to improve average



Latency	1ns	10ns	100ns	10ms
Size	KB	MB	GB	TB

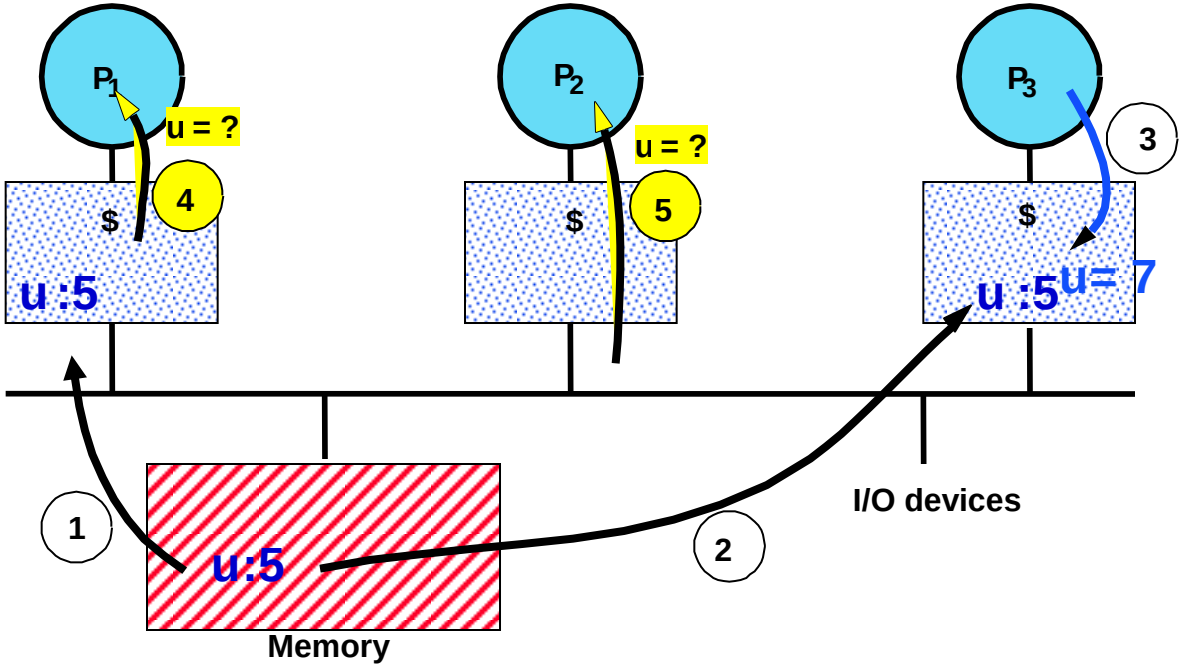
Caching in Shared Memory Multiprocessors



- Want High performance for shared memory: Use Caches!
  - Each processor has its own cache (or multiple caches)
  - Place data from memory into cache
  - Writeback cache: don't send all writes over bus to memory
- Caches Reduce average latency
  - Automatic replication closer to processor
  - *More* important to multiprocessor than uniprocessor: latencies longer
- Normal uniprocessor mechanisms to access data
  - Loads and Stores form very low-overhead communication primitive
- Problem: Cache Coherence!

Slide source: John Kubiatowicz

Example Cache Coherence Problem

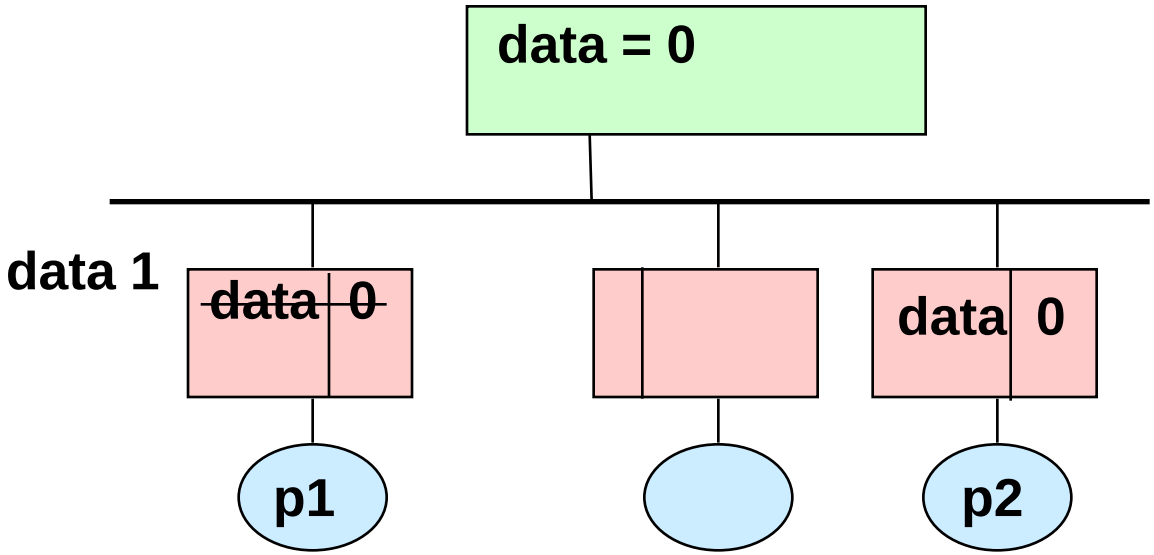


- Processors could see different values for u after event 3
- How to fix with a bus: Coherence Protocol
  - Use bus to broadcast writes or invalidations
- Bus not scalable beyond about 64 processors (max)
  - Capacity, bandwidth limitations

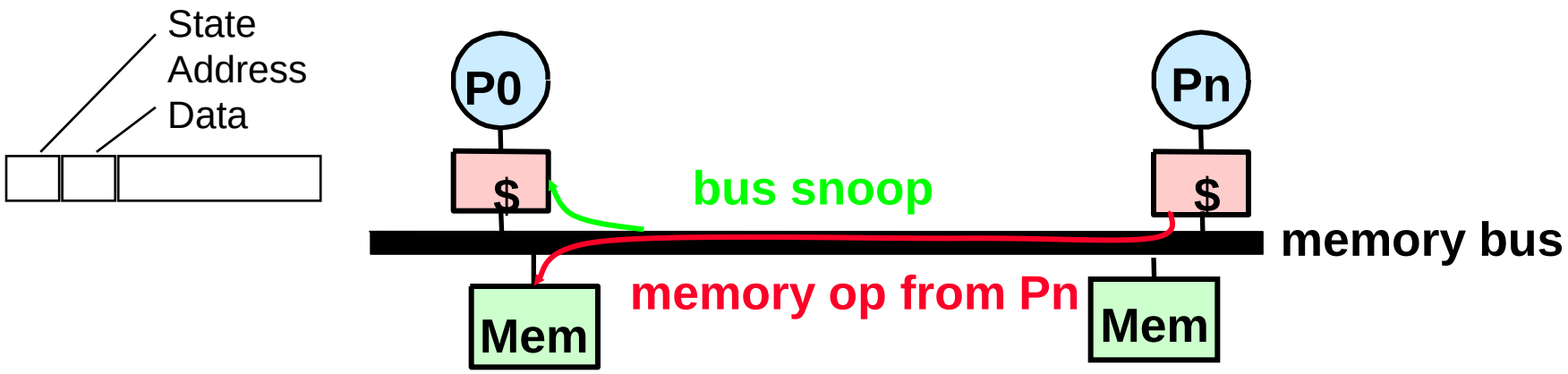
Slide source: John Kubiatowicz

Another example: Cache Coherence

- Coherence means different copies of same location have same value, incoherent otherwise:
- p1 and p2 both have cached copies of data (= 0)
- p1 writes data=1
  - May "write through" to memory
- p2 reads data, but gets the "stale" cached copy
  - This may happen even if it read an updated value of another variable, flag, that came from memory



Snoopy Cache-Coherence Protocols

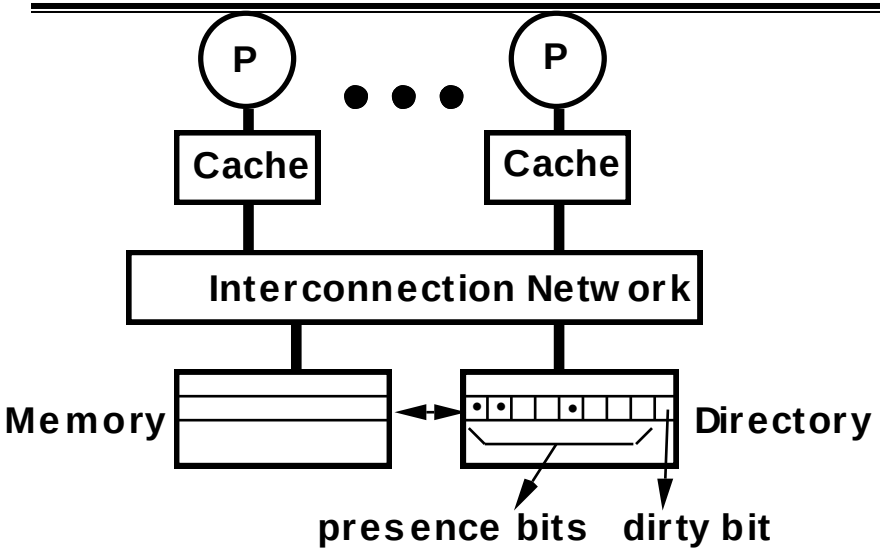


- Memory bus is a broadcast medium
- Caches contain information on which addresses they store
- Cache Controller "snoops" all transactions on the bus
  - A transaction is a relevant transaction if it involves a cache block currently contained in this cache
  - Take action to ensure coherence
    - invalidate, update, or supply value
  - Many possible designs
- Not scalable for a large number of processors

Directory Based Memory/Cache Coherence

- Keep Directory to keep track of which memory stores latest copy of data
- Directory, like cache, may keep information such as:
  - Valid/invalid
  - Dirty (inconsistent with memory)
  - Shared (in another caches)
- When a processor executes a write operation to shared data, basic design choices are:
  - With respect to memory:
    - Write through cache: do the write in memory as well as cache
    - Write back cache: wait and do the write later, when the item is flushed
  - With respect to other cached copies
    - Update: give all other processors the new value
    - Invalidate: all other processors remove from cache

Scalable Shared Memory: Directories



- **k processors.**
- **With each cache-block in memory: k presence-bits, 1 dirty-bit**
- **With each cache-block in cache: 1 valid bit, and 1 dirty (owner) bit**

- Every memory block has associated directory information
  - keeps track of copies of cached blocks and their states
  - on a miss, find directory entry, look it up, and communicate only with the nodes that have copies if necessary
  - in scalable networks, communication with directory and copies is through network transactions
- Each Reader recorded in directory
- Processor asks permission of memory before writing:
  - Send invalidation to each cache with read-only copy
  - Wait for acknowledgements before returning permission for writes

Slide source: John Kubiatowicz

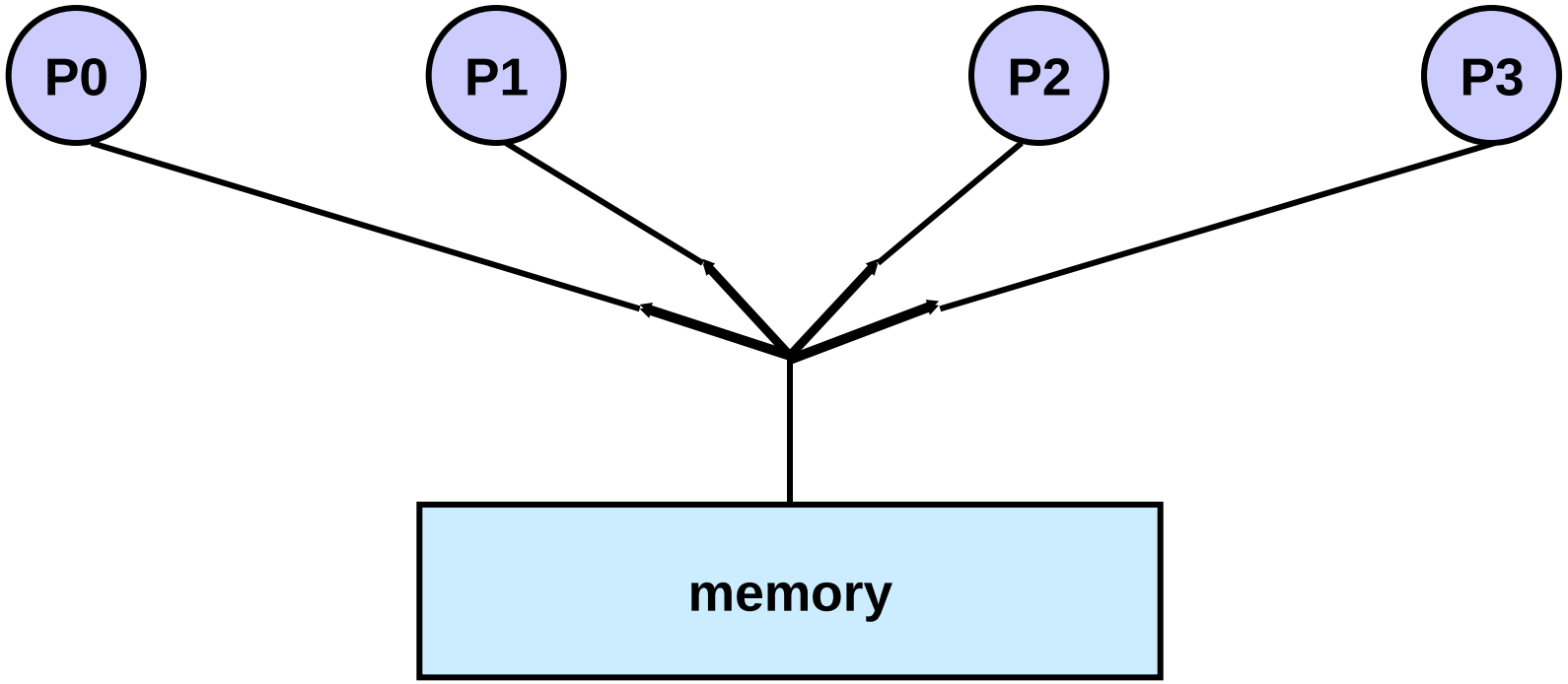
Intuitive Memory Model

- Reading an address should return the last value written to that address
- Easy in uniprocessors
  - except for I/O
- Cache coherence problem in MPs is more pervasive and more performance critical
- More formally, this is called sequential consistency:

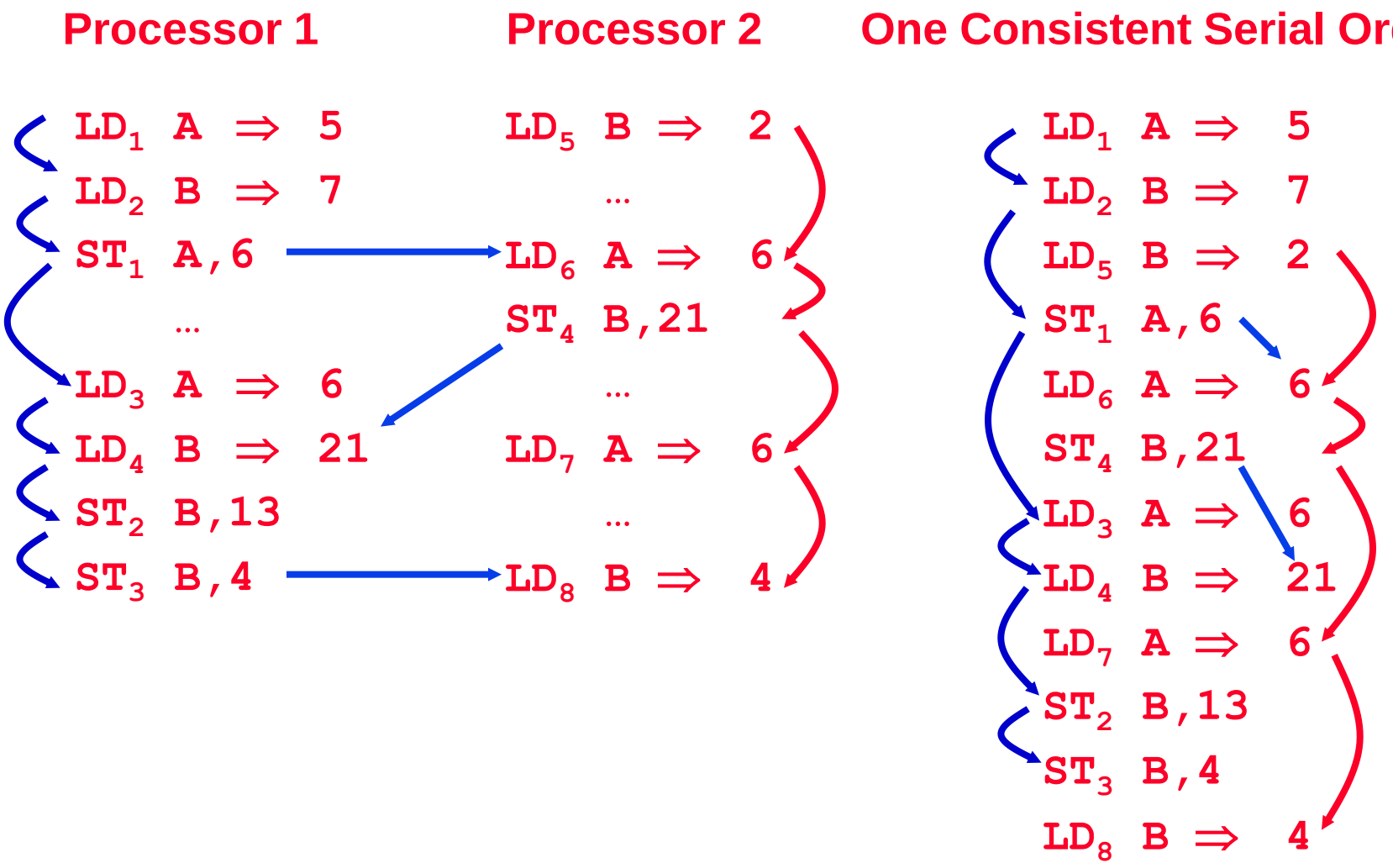
  "A multiprocessor is *sequentially consistent* if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program." [Lamport, 1979]

Sequential Consistency Intuition

- Sequential consistency says the machine *behaves as if* it does the following



Sequential Consistency Example



Slide source: John Kubiatowicz

Memory Consistency Semantics

What does this imply about program behavior?

- **No process ever sees "garbage" values, i.e., average of 2 values**
- **Processors always see values written by some processor**
- **The value seen is constrained by program order on all processors**
  - **Time always moves forward**
- **Example: *spin lock***
  - **P1 writes data=1, then writes flag=1**
  - **P2 waits until flag=1, then reads data**

If P2 sees the new value of flag (=1), it must see the new value of data (=1)

initially:	flag=0 data=0

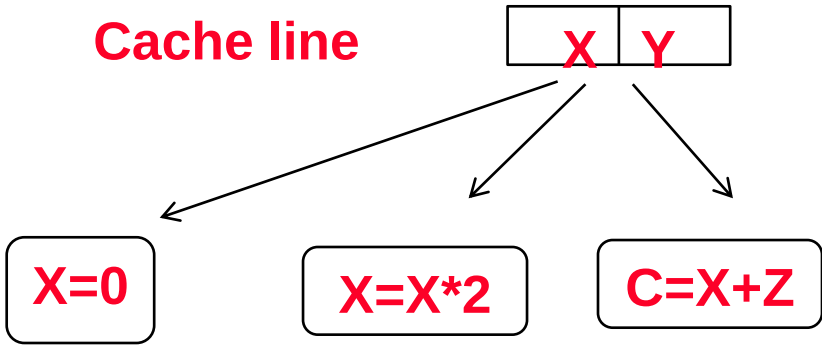
P1	P2
data = 1 flag = 1	While flag=0; print data

If P2 reads flag	Then P2 may read data
0	1
0	0
1	1

Cache Coherence and Sequential Consistency

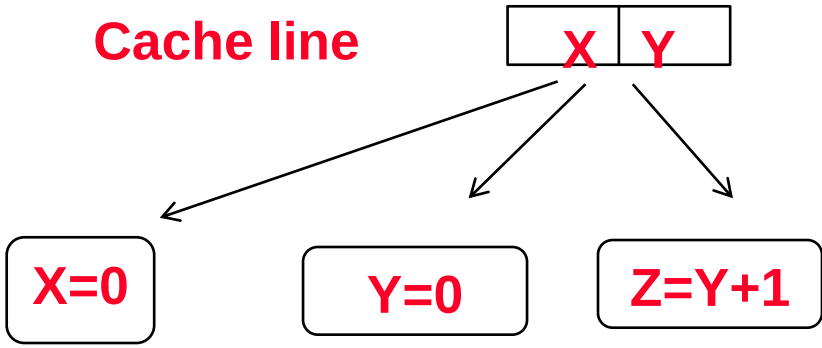
- There is a lot of hardware/work to ensure coherent caches
  - Never more than 1 version of data for a given address in caches
- But other HW/SW features may break sequential consistency (SC):
  - The compiler reorders/removes code (e.g., your spin lock, see previous slide)
  - Write buffers (place to store writes while waiting to complete)
    - Processors may reorder writes to merge addresses (not FIFO)
    - Write X=1, Y=1, X=2 (second write to X may happen before Y's)
  - Prefetch instructions cause read reordering (read data before flag)
  - The network reorders the two write messages.
    - The write to flag is nearby, whereas data is far away.
- Some commercial systems give up SC
  - A correct program on a SC processor may be incorrect on one that is not

Performance Issue in True Sharing



- True sharing
  - Frequent writes to a variable can create a bottleneck
  - OK for read-only or infrequently written data
  - Example problem: the data structure that stores the freelist/heap for malloc/free
- Technique:
  - Make copies of the value, one per processor, if this is possible in the algorithm

Performance Issue with False Sharing



- False sharing
  - Cache block may also introduce artifacts
  - Two distinct variables in the same cache block
  - Example problem: an array of ints, one written frequently by each processor (many ints per cache line)
- Technique:
  - allocate data used by each processor contiguously, or at least avoid interleaving in memory

Programming with Weaker Memory Models than SC

- Possible to reason about machines with fewer properties, but difficult
- Some rules for programming with these models
  - Avoid race conditions
  - Use system-provided synchronization primitives
  - At the assembly level, may use "fences" (memory barrier) directly
- The high level language support for these differs
  - Built-in synchronization primitives normally include the necessary fence operations
    - lock (), … only one thread at a time allowed here…. unlock()
    - Region between lock/unlock called **critical region**
  - For performance, need to keep critical region short

What to Take Away?

- Programming shared memory machines
  - May allocate data in large shared region without too many worries about where
  - Memory hierarchy is critical to performance
    - Even more so than on uniprocessors, due to coherence traffic
  - For performance tuning, watch sharing (both true and false)
- Semantics
  - Need to lock access to shared variable for read-modify-write
  - Sequential consistency is the natural semantics
    - Write race-free programs to get this
  - Architects worked hard to make this work
    - Caches are coherent with buses or directories
    - No caching of remote data on shared address space machines
  - But compiler and processor may still get in the way
    - Non-blocking writes, read prefetching, code motion…
    - Avoid races or use machine-specific fences carefully