ECSE 425 Lecture 27: Symmetric Multiprocessors

H&P Chapter 4

Last Time

- Basic multi-processor architecture
- Parallel processing challenges

Today

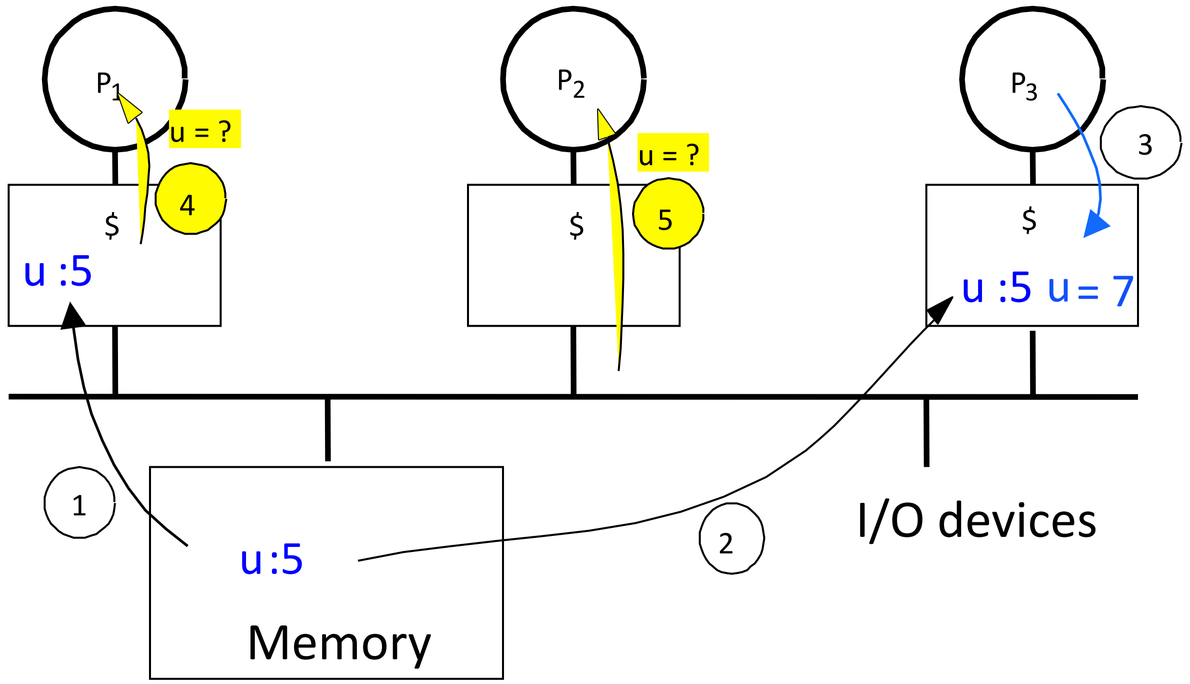
- Symmetric Multiprocessors
  - Cache coherence
  - Memory consistency
- Introduction to Cache Coherence Protocols

Symmetric Shared-Memory

- Reduce memory bandwidth with multilevel caches
  - If bandwidth is reduced enough, multiple processors may be able to share the same memory
  - E.g., multiple processor boards connected by a shared bus
  - E.g., multiple processors inside a single chip
- Private data is used by a single processor
- Shared data is used by multiple processors
  - Shared value may be replicated in many caches
  - Reduces latency and bandwidth requirements, but introduces a *cache coherence* problem

Example Cache Coherence Problem

- Processors see different values for *u* after event 3
- With write-back caches
  - Memory and caches store different values of *u*
  - Value written to memory depends on access ordering
- Unacceptable for programming, and it's frequent!



But Coherence is Not Enough

P_1	P_2
/*Assume initial value of A and flag is 0*/	
`A = 1;`	`while (flag == 0);` /* spinning */
`flag = 1;`	`print A;`

- Coherence pertains only to single location
- Memory must respect order between accesses to *different* locations by a process
  - To preserve orders among accesses to same location by different processes
- This is called *consistency*

Memory System Model

- A read request for an address should return the last value written to that address

1. Coherence defines values returned by a read
2. Consistency determines when a written value will be returned by a read

- Coherence defines the behavior of accesses to the same location
- Consistency defines the behavior of accesses to different locations

Defining Coherent Memory System

1. Preserve program order:
   - RAW from the same processor (no intervening write)
   - Read must return the value written by write
2. Coherent view of memory:
   - RAW by different processors (given "sufficient" time)
   - Read must return the value written by write
3. Write serialization:
   - WAW by different processors
   - Two writes to same location by any two processors must be seen in the same order by all processors

Defining Consistency

- When must a written value be seen by a read?
  - RAW by different processors
  - If insufficient time separation, read may not return the value of the most recent write
- For now assume
  - A write does not complete (and allow the next write to occur) until all processors have seen its effect
  - The processor does not change the order of any write with respect to any other memory access

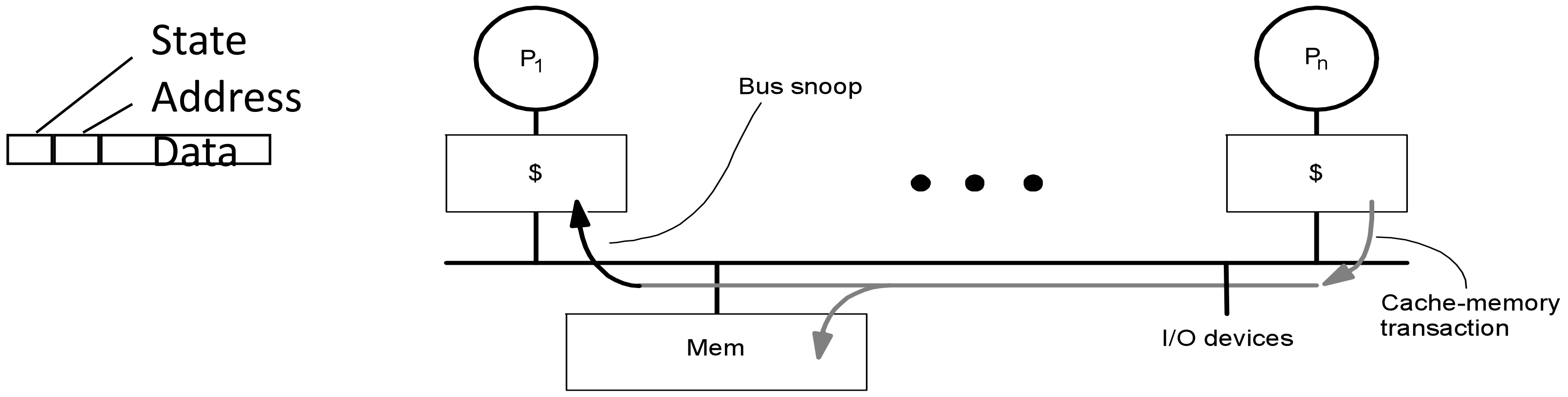
Basic Schemes for Enforcing Coherence

- A program executing on multiple processors will have copies of the same data in several caches
- Coherent caches support *migration* and *replication*
- Migration– data can be moved to a local cache and used there in a transparent fashion
  - Reduces both latency and bandwidth demand
- Replication– for shared data being simultaneously read, since caches make a copy of data in local cache
  - Reduces both latency and contention for read shared data
- Rather than avoiding sharing in SW, SMPs use a HW protocol to maintain coherent caches
  - Cache coherence protocols

Two Classes of Coherence Protocols

1. Directory-based: sharing status of a block of physical memory is kept in the directory
   - Slightly higher implementation overhead
   - Can scale to larger processor counts
2. Snooping: every cache with a copy of data also maintains the sharing status of block
   - All caches must be accessible via some broadcast medium
   - All cache controllers monitor (“snoop”) the medium and respond to requests for data
   - Existing broadcast mechanism makes snooping simple to implement, but also limits its scalability

Snooping Protocols



- Cache continuously monitor transactions on the bus
- Respond or take action when address matches a block the cache contains
  - Invalidate, update, or supply value
  - Depending on state of the block and the protocol
- On a write: either get exclusive access before write via write invalidate or update all copies on write

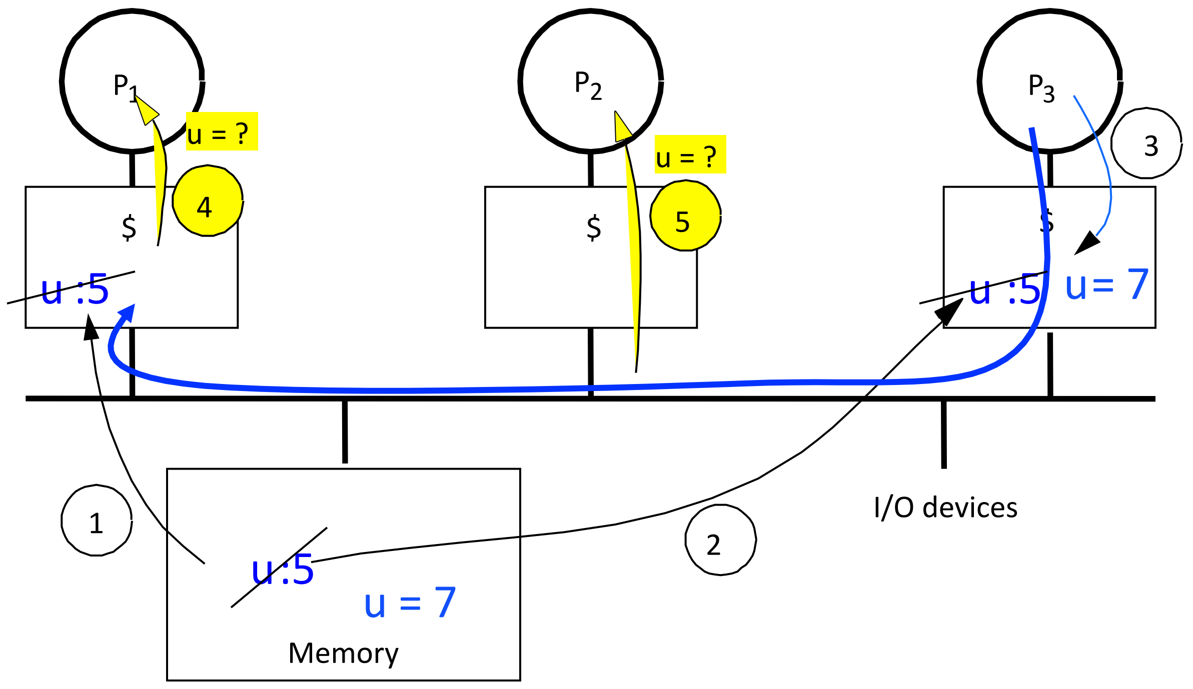
Write Update

- An alternative to write invalidate
- On a write, update all cached copies
  - Broadcast the write value to all shared cache lines
- Consumes considerable bandwidth
  - And as a result, not popular

Write Invalidate

- A writing cache has exclusive access to the data
  - All other copies (with other processors) are invalidated
- If another processor reads after a write
  - The read will miss (the data was marked invalid)
  - The processor will fetch the new copy
- If two writes to the same data at the same time
  - Competition– one succeeds, the other fails
  - The failed processor must obtain a new copy of data to complete its write
  - This is called *write serialization*

Example: Write Invalidate



- Must invalidate before step three
- Write update uses more broadcast medium BW $\Rightarrow$ all recent MPs use write invalidate

Write Invalidate Implementation

- Broadcast on the bus to invalidate shared data
  - First, acquires bus access
  - Second, broadcast the address to be invalidated
- Other processors continuously snoop the bus
  - Compare addresses against cache contents
  - If the invalidated matches, they invalidate their copy
- The bus serializes writes
  - Two simultaneous writes?
  - Only one processor gets bus access

Locating Up-to-date Data

- On a R/W miss, need to find up-to-date data
- Write-through: get an up-to-date copy from memory
  - Simple, but high memory BW requirement
- Write-back: up-to-date copy may be in a cache
  - Snoop for both misses and writes
  - Processors with dirty data respond to read misses
  - Can be slower than accessing memory if the processors are on separate chips
- Write-back requires less memory bandwidth
  $\Rightarrow$ Can support larger numbers of faster processors
  $\Rightarrow$ Most multiprocessors use write-back

Cache Behavior and Local Accesses

- Normal cache tags can be used for snooping
  - Compare tag on bus with tag in cache
- Valid bit per block makes invalidation easy
- Read misses are handled by main memory and other snooping processors
- When writing, need to know if the block is shared
  - Maintain a "shared" bit for each cache block
- Block not shared? no need to broadcast the write
- If the block is shared, broadcast an invalidate
  - Then mark block as exclusive (unshared)

Cache Behavior and Remote Requests

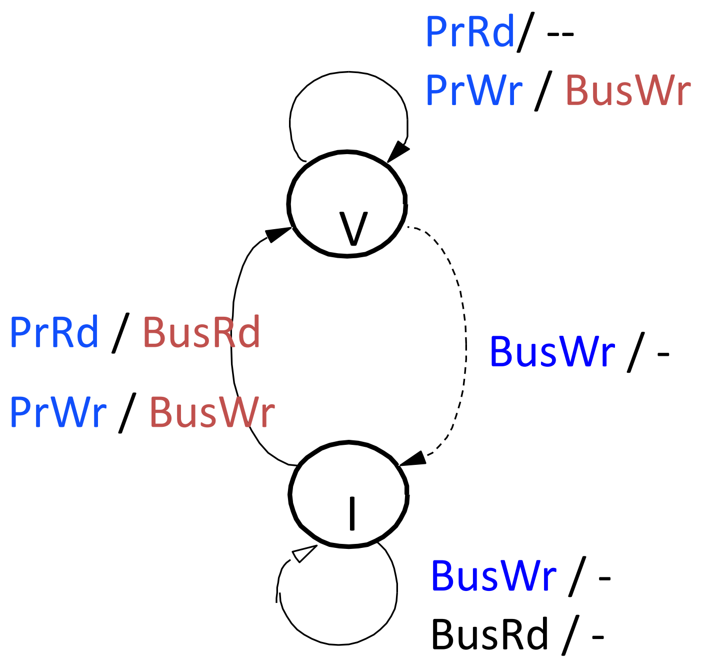
- Must check the cache on every bus transaction
  - Could interfere with processor cache accesses
- Reduce interference by duplicating tags
  - One set for CPU accesses, one set for bus accesses
- Or, reduce interference by using L2 tags
  - L2 is less heavily used than L1
  - Requires L2 inclusion
- If snooping hits in L2
  - Check if the data is dirty in L1
  - May require stalling the processor

Implementing Snooping

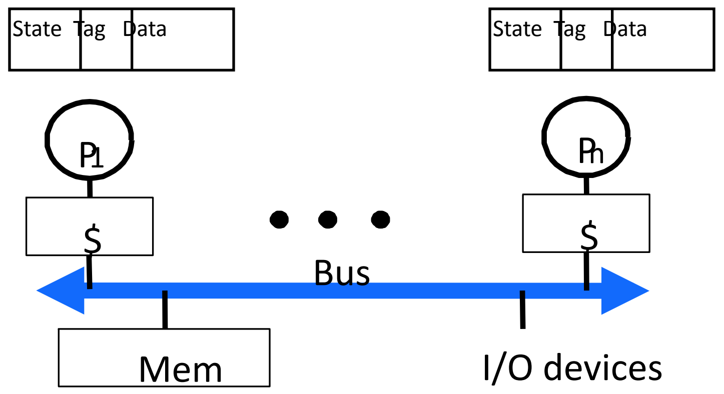
- Implement a cache controller at each node
- Logically, a controller for each cache block
  - Snooping operations or cache requests for different blocks can proceed independently
- Physically, the single controller interleaves multiple operations on distinct blocks
  - Though only one cache access or one bus access is allowed at time,
  - One operation may start before another finishes

1. Write-through Invalidate Protocol

- Two states per block
  - Valid and Invalid
  - As in an uniprocessor
- Any writes invalidate all other cached copies
  - Can have multiple simultaneous readers
  - Write invalidates them



PrRd: Processor Read
PrWr: Processor Write
BusRd: Bus Read
BusWr: Bus Write



Now for Some Assumptions

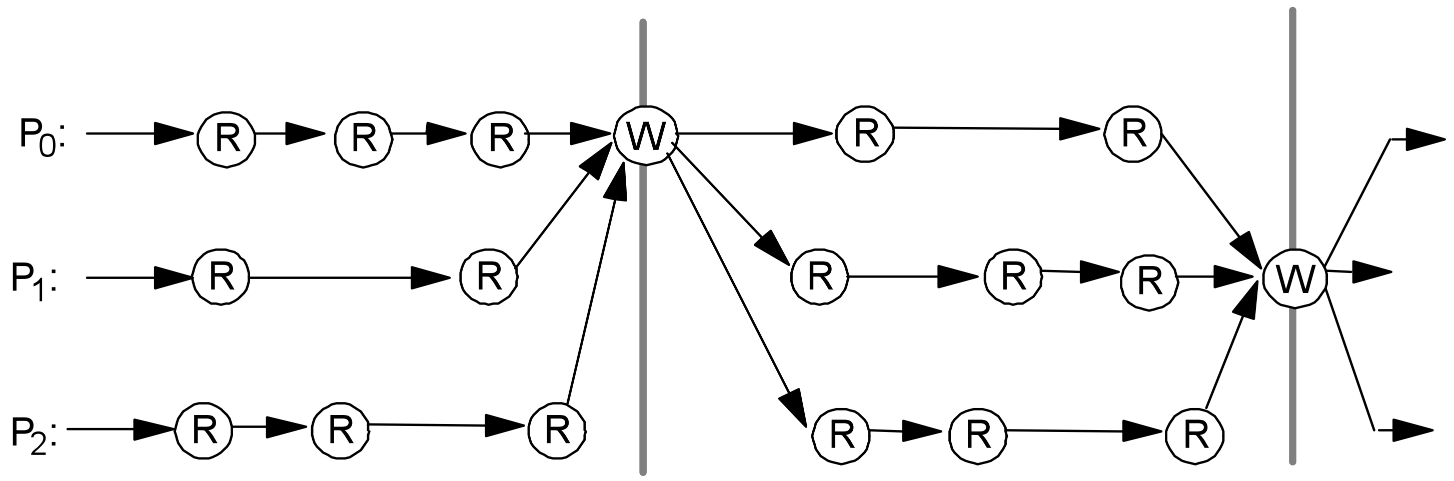
- One-level cache
- Bus transactions and memory operations are atomic
- All phases of a transaction complete before the next starts
- Processor waits for a memory operation to complete before issuing the next
- Invalidations are applied during bus transactions

Is the Two-state Protocol Coherent?

- The processor observes memory state by issuing memory operations
- Writes are serialized in the order in which they appear on the bus
  - All writes go to bus, and are atomic

  => Invalidations are applied to caches in bus order
- How to insert reads in this order?
  - Important, since processors see writes through reads, so determines whether write serialization is satisfied
  - But read hits may happen independently and do not appear on bus or enter directly in bus order

Ordering



- Writes establish a partial order
- Read ordering is unconstrained
  – Though shared-medium (bus) will order read misses
  – Any order among reads between writes is fine, as long as in program order for each processor

Next Time

- More cache coherence!