ECSE 425 Lecture 28: Snoopy Coherence Protocols

H&P Chapter 4

Last Time

- Cache coherence
- Memory consistency

Today

- Cache Coherence Protocols
  - Write Update
  - Write-Through Invalidate
  - Write-Back Invalidate

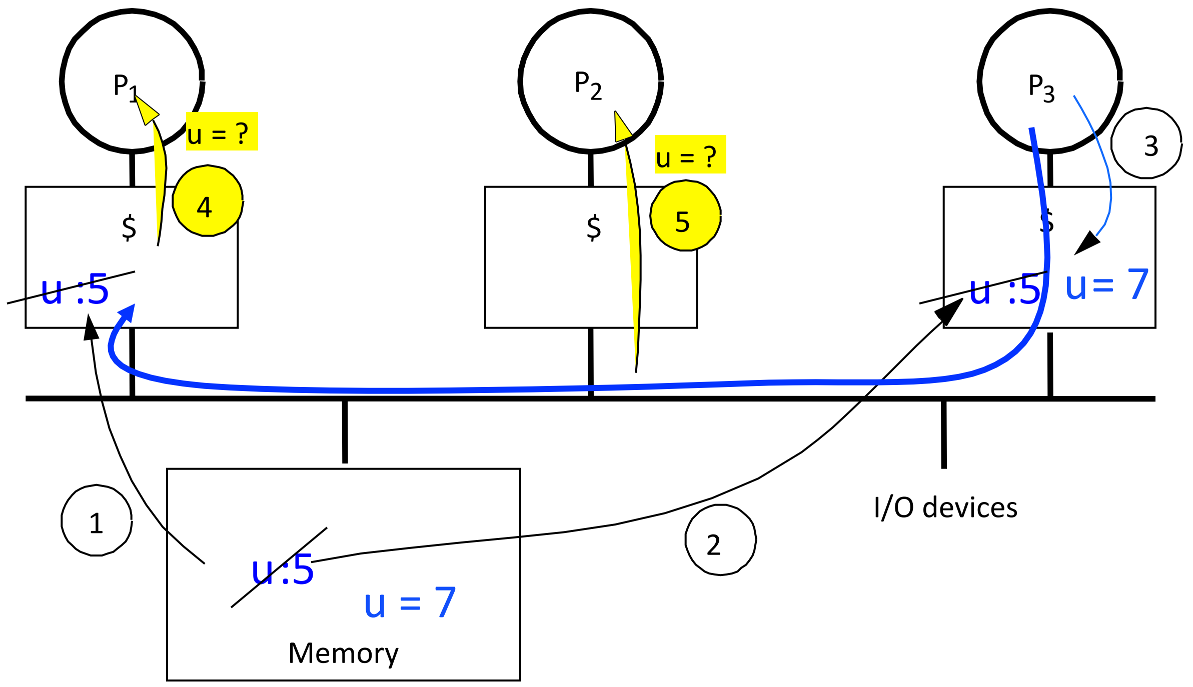
Write Update

- 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 ⇒ 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

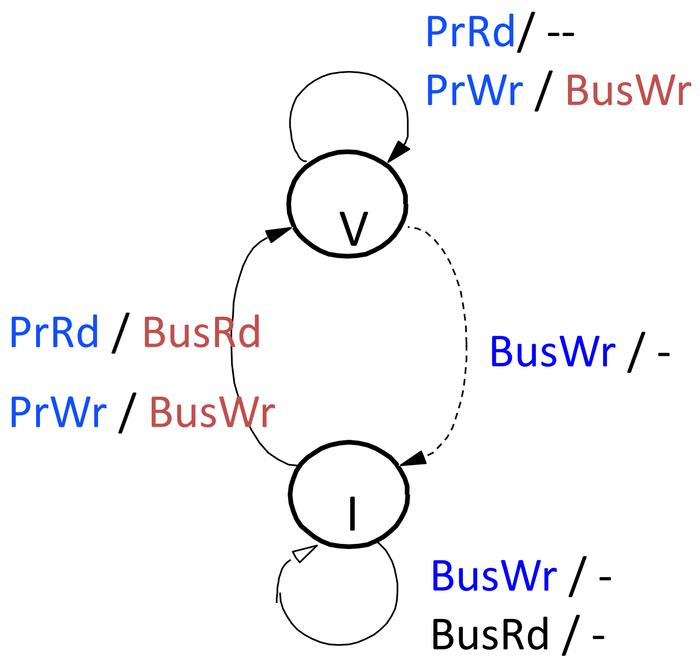
- Must check the cache on every bus transaction
  - Could interfere with processor cache accesses
- Reduce interference by duplicating tags
  - One set for CPU accesses, one set for bus accesses
- Or, reduce interference by using L2 tags
  - L2 is less heavily used than L1
  - Requires L2 inclusion
-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

Implementing Snooping

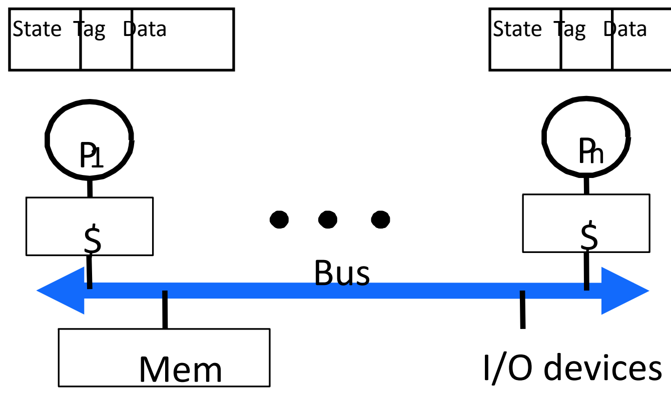
- Implement a cache controller at each node
- Logically, a controller for each cache block
  - Snooping operations or cache requests for different blocks can proceed independently
- Physically, the single controller interleaves multiple operations on distinct blocks
  - Though only one cache access or 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

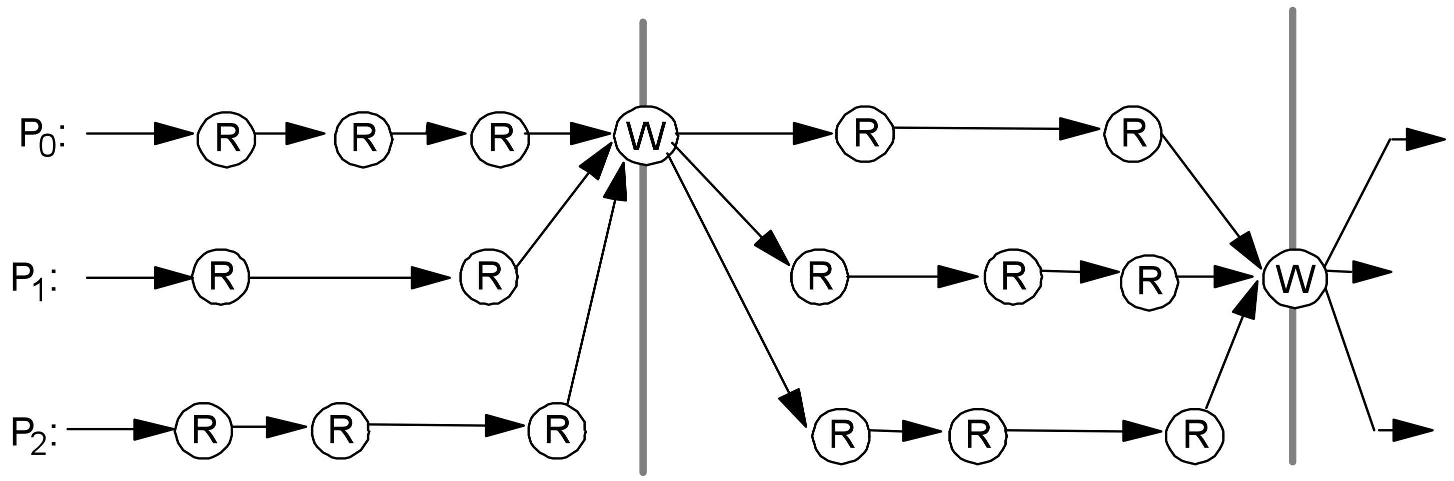
- One-level cache
- Bus transactions and memory operations are atomic
- All phases of a transaction complete before the next starts
- Processor waits for a memory operation to complete before issuing the next
- Invalidations are applied during bus transactions

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



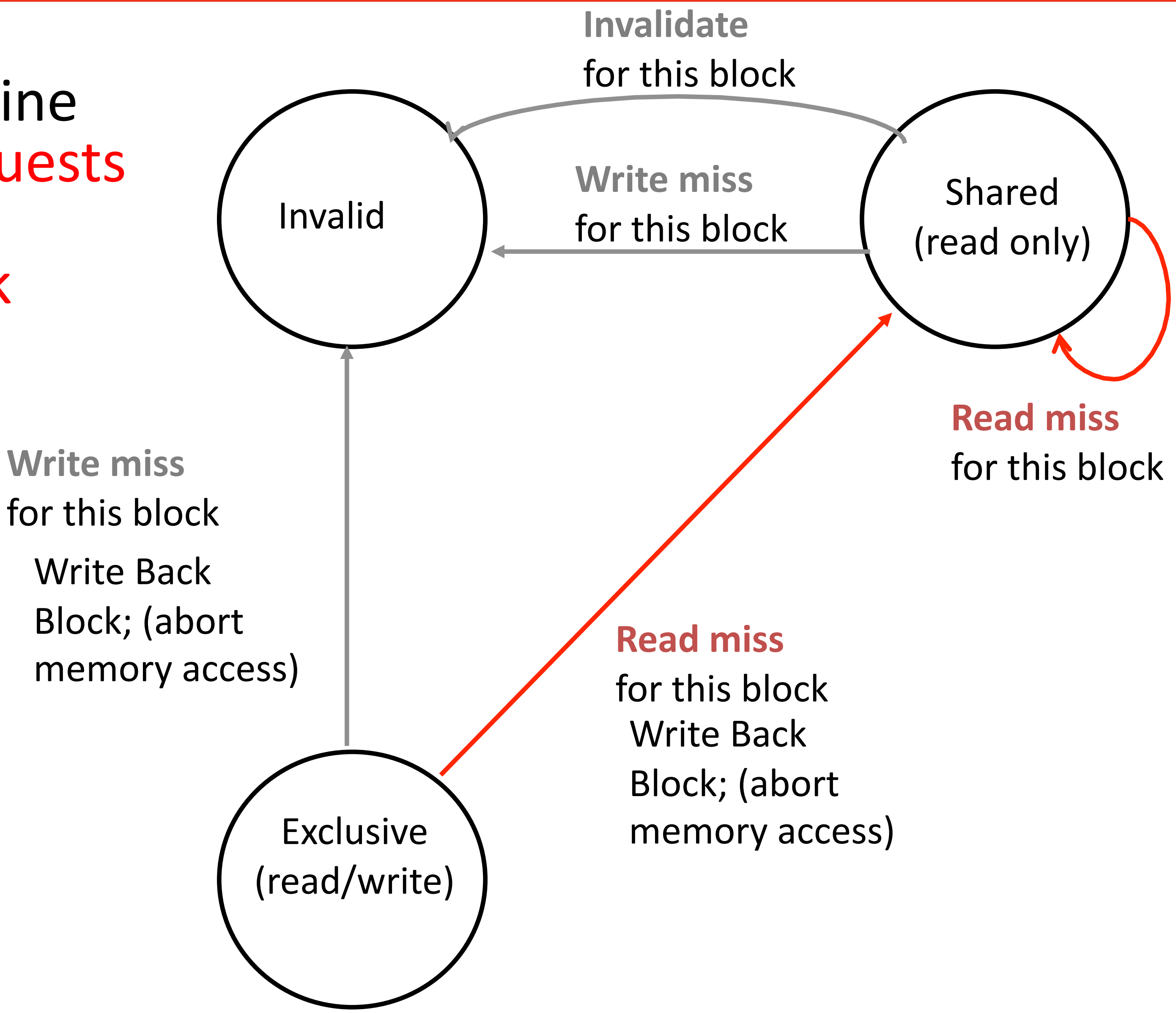
- Writes establish a partial order
- Read ordering is unconstrained
  - Though shared-medium (bus) will order read misses
  - Any order among reads between writes is fine, as long as in program order for each processor

2. Write-Back Invalidate Protocol

- Each memory block is in one state:
  - **Uncached**
  - OR **Clean** in all caches and up-to-date in memory (**Shared**)
  - OR **Dirty** in exactly one cache (**Exclusive**)
- Each cache block is in one state (track these):
  - **Invalid**: block contains no data (as in uni-processor cache)
  - OR **Shared**: block can be read (clean)
  - OR **Exclusive** (or Modified): cache has only copy, it is writeable and dirty
- All caches snoop read and write misses
  - Write-back on read misses
  - Write-back and Invalidate on write misses

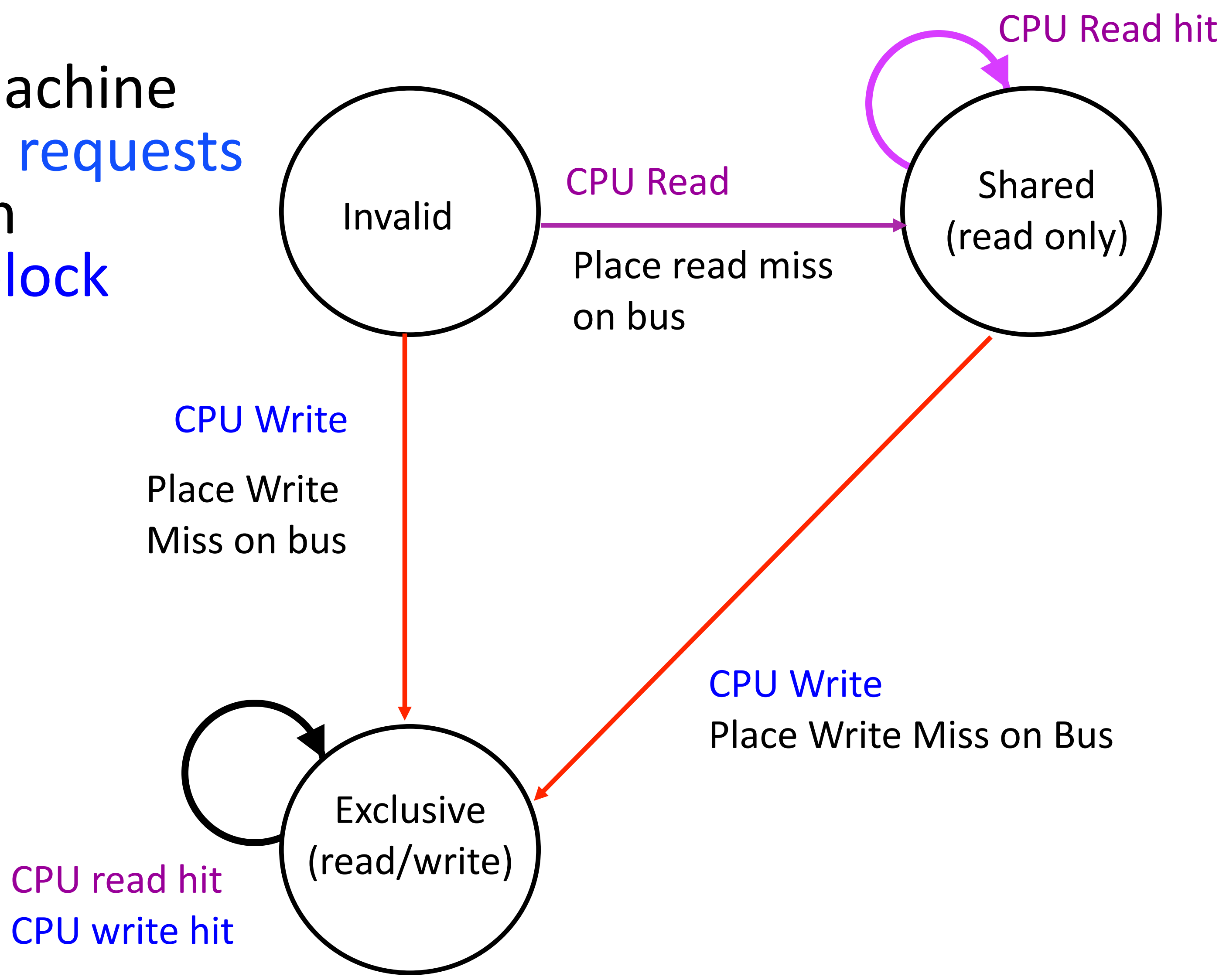
Write-Back State Machine–Bus requests

- State machine for bus requests for each cache block



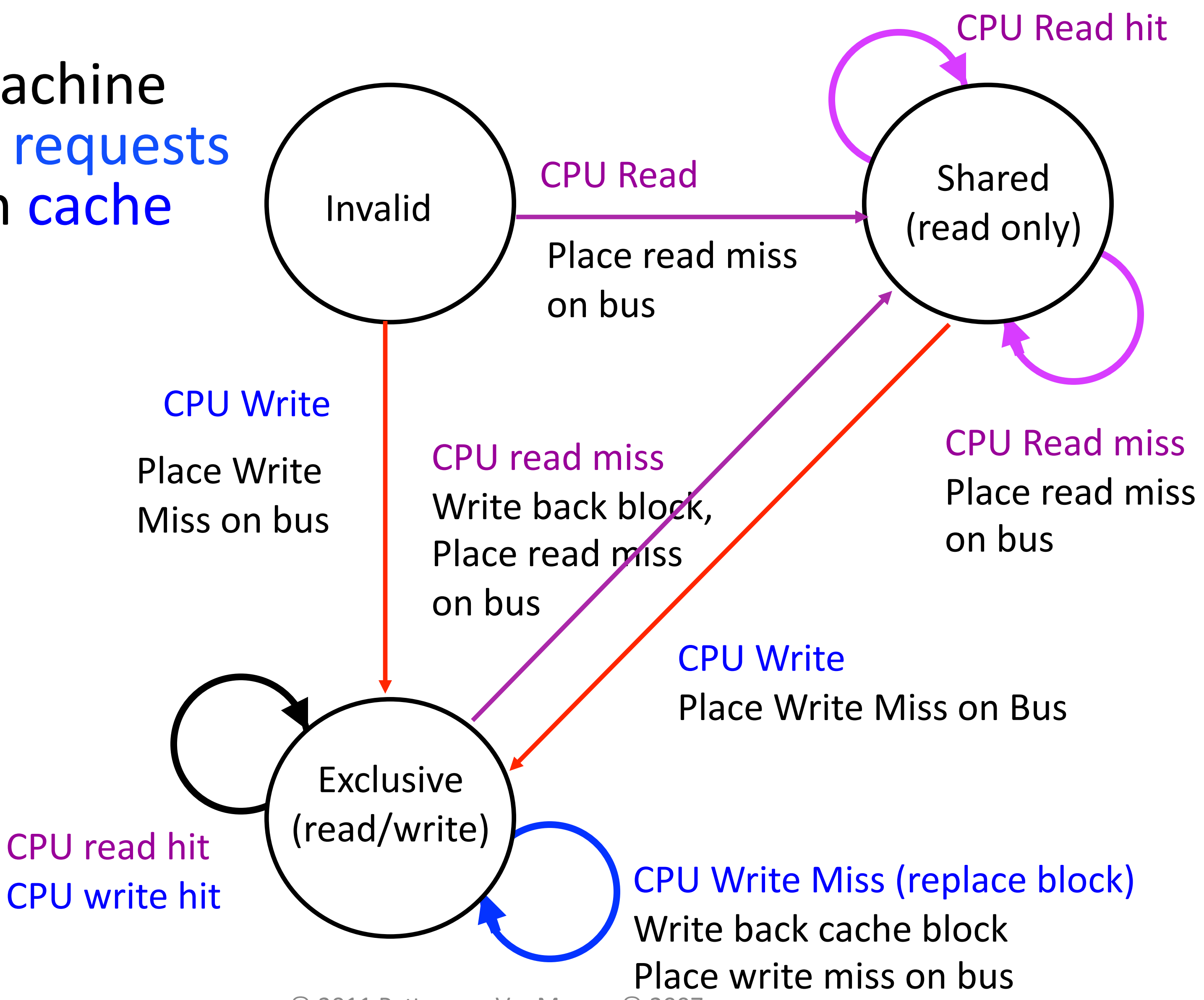
Write-Back State Machine–CPU Requests

- State machine for CPU requests for each cache block



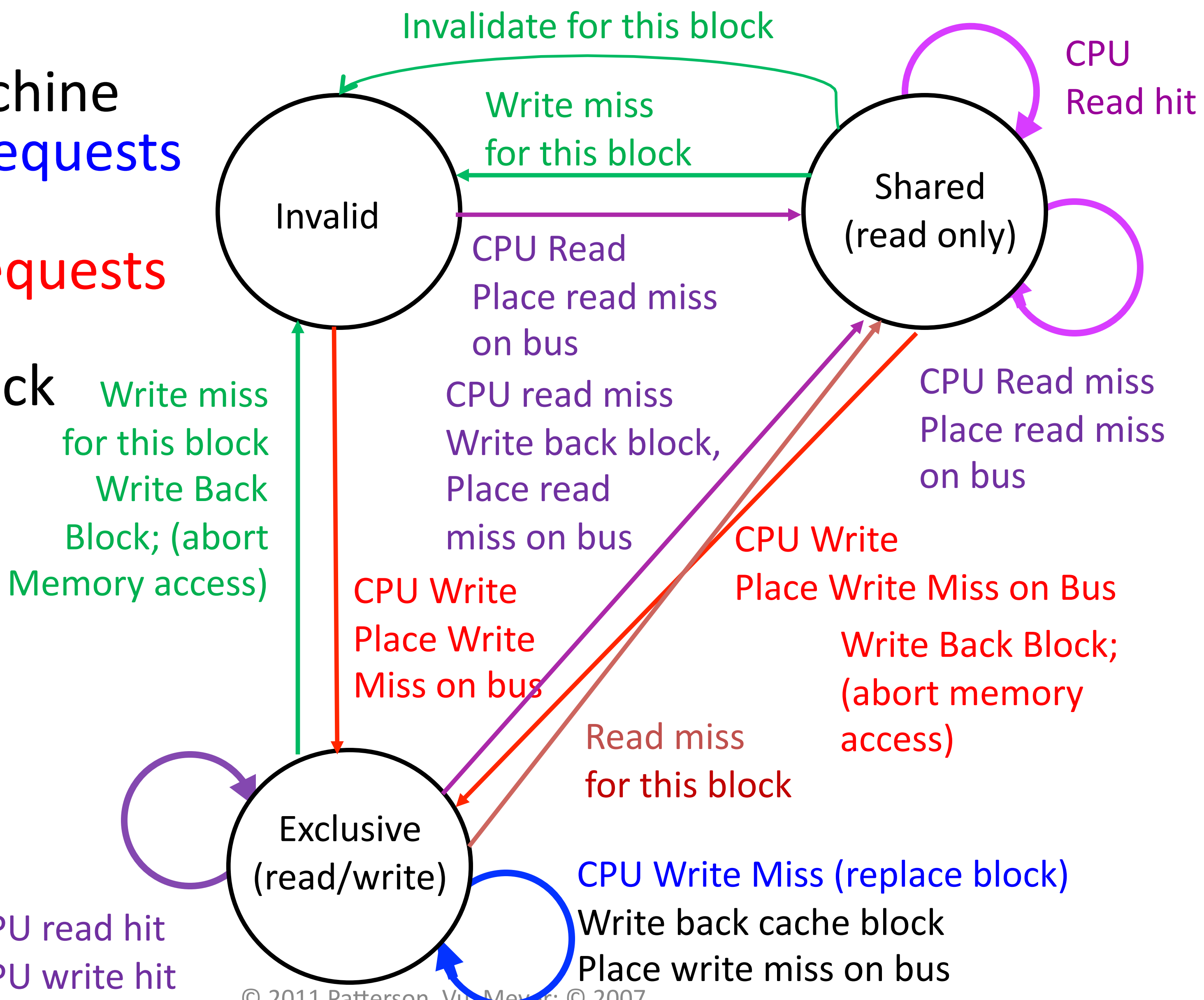
Block Replacement

- State machine for CPU requests for each cache block



Write-back State Machine-III

- State machine for CPU requests and for bus requests for each cache block



Next Time

- Write-back Invalidate Example
- Symmetric Shared-Memory Multiprocessor Performance