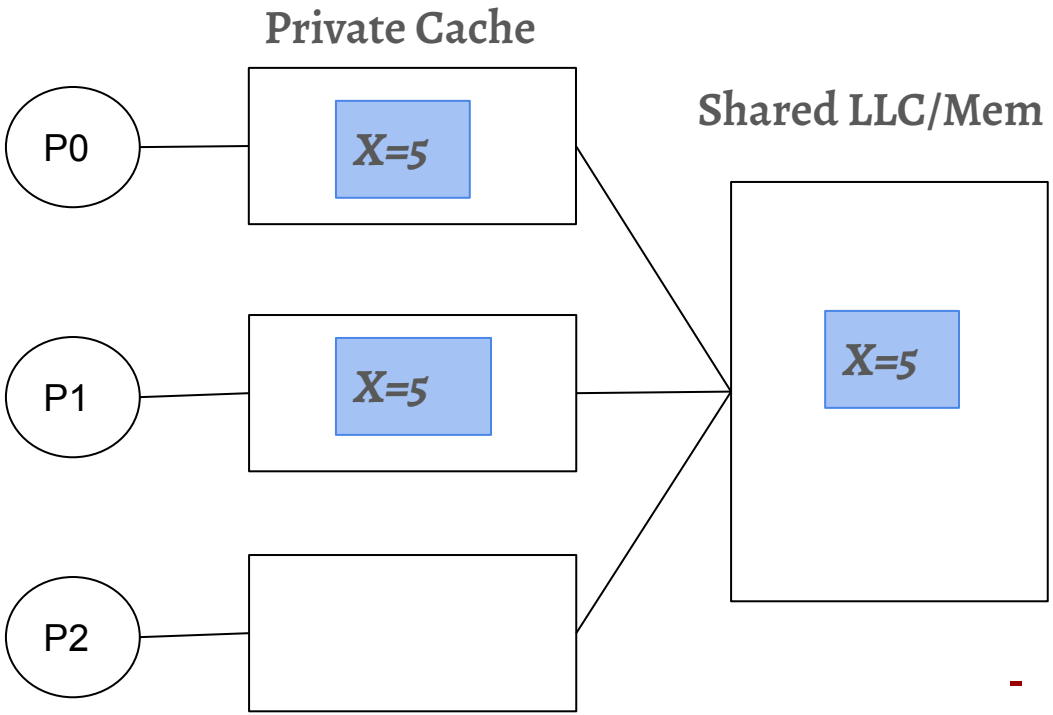
CS622: Advanced Computer Architecture

Cache Coherence

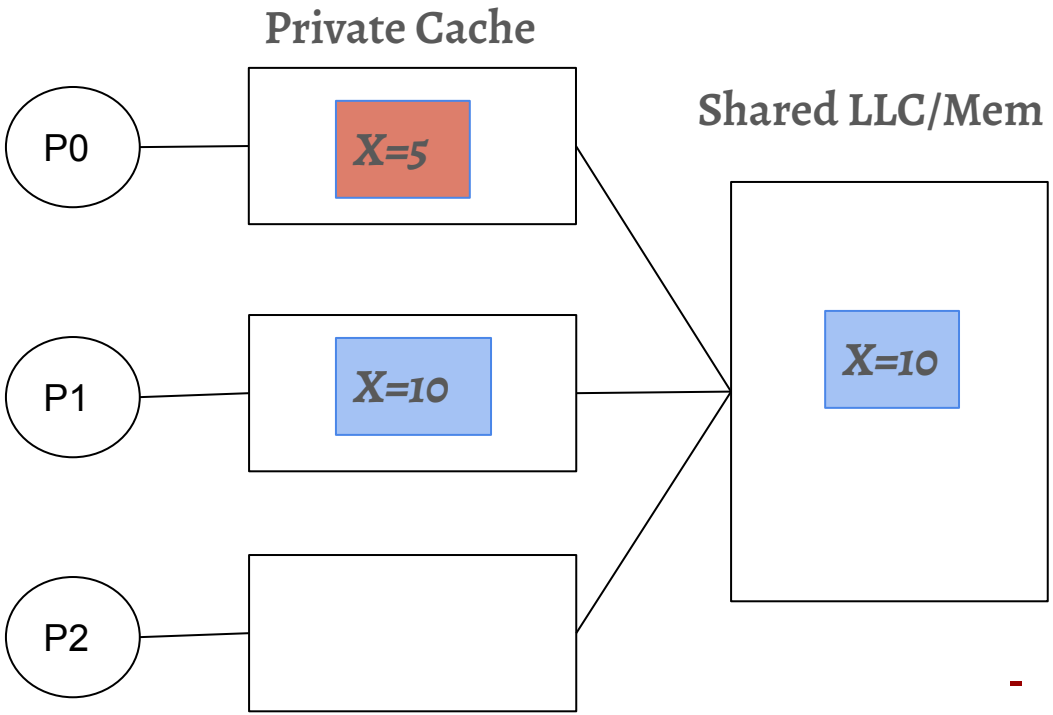
Cache Coherence Example: Write-through Cache



P0: *Read (X)*
P1: *Read(X)*
P1: *Write (X, 10)*
P0: *Read (X)*
P2: *Read (X)*
P2: *Write (X, 20)*

- Assume value of X in memory is 5
- At this point, there are no coherence issues

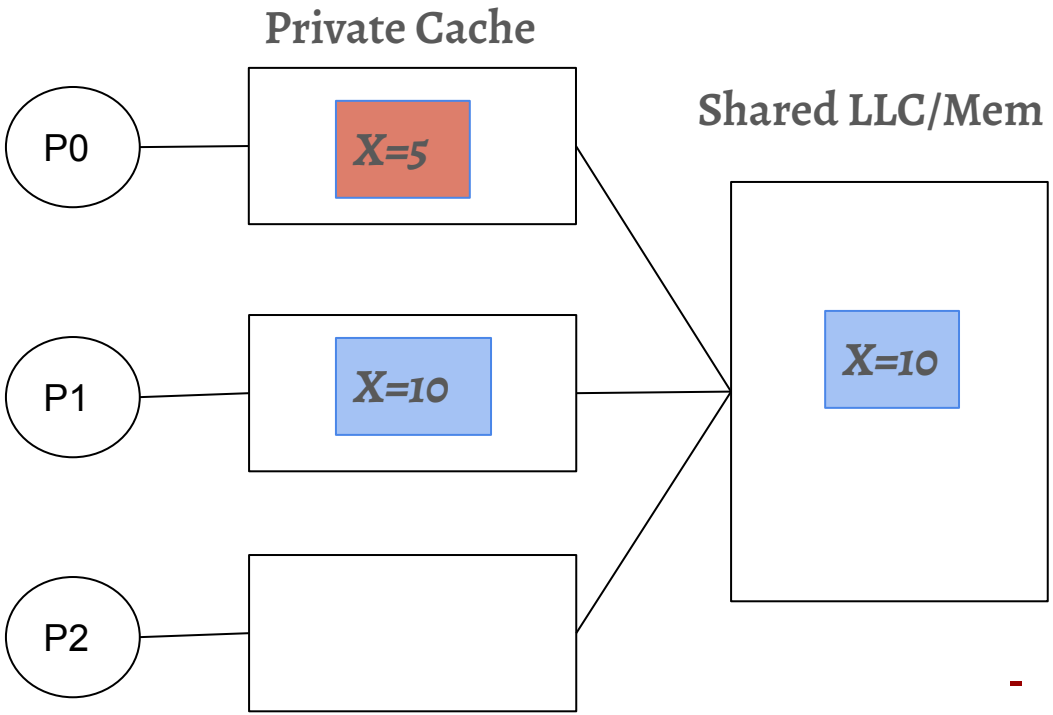
Cache Coherence Example: Write-through Cache



P0: *Read (X)*
P1: *Read(X)*
P1: *Write (X, 10)*
P0: *Read (X)*
P2: *Read (X)*
P2: *Write (X, 20)*

- Write from P1 updates its private cache and memory (WT-cache)
- Private cache of P0 is incoherent

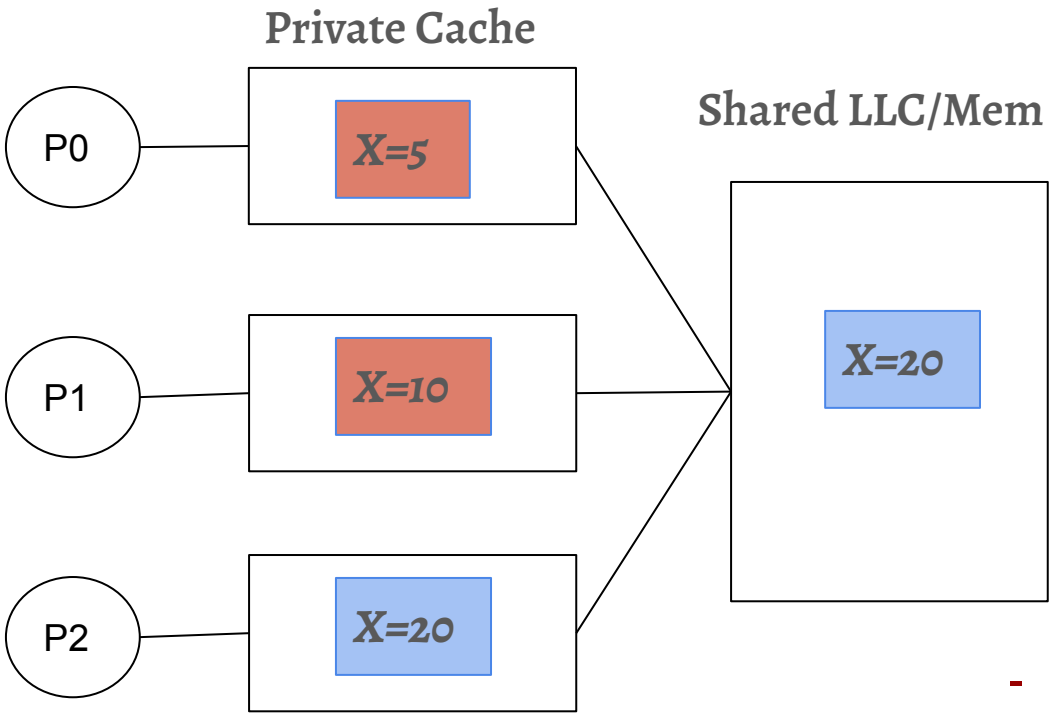
Cache Coherence Example: Write-through Cache



P0: *Read (X)*
P1: *Read(X)*
P1: *Write (X, 10)*
P0: *Read (X)*
P2: *Read (X)*
P2: *Write (X, 20)*

- P0 reads X=5
- P0 may never read the updated X!

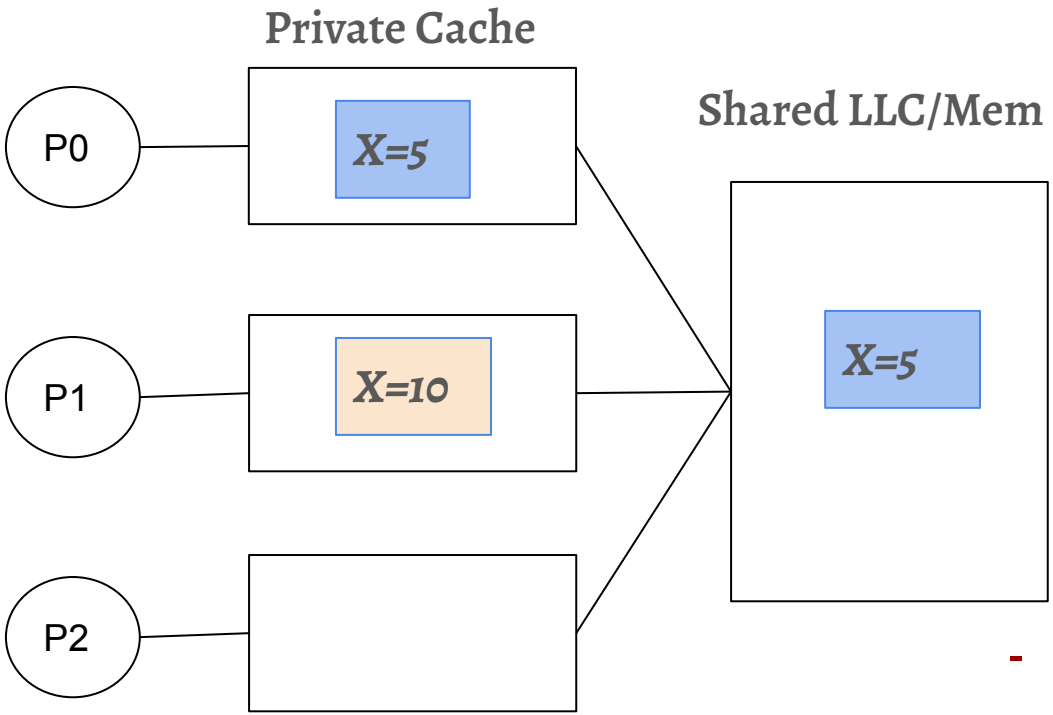
Cache Coherence Example: Write-through Cache



P0: *Read (X)*
P1: *Read(X)*
P1: *Write (X, 10)*
P0: *Read (X)*
P2: *Read (X)*
P2: *Write (X, 20)*

- P2 reads X=10 and writes 20 to X
- Private caches of both P0 and P1 are not coherent now

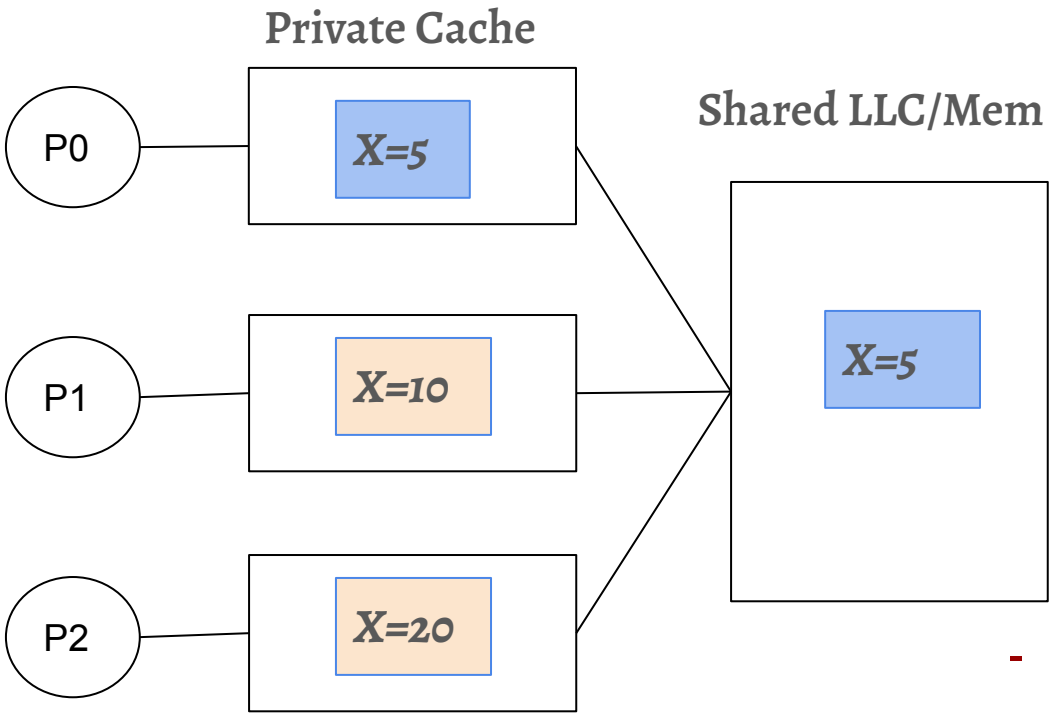
Cache Coherence Example: Write-back Cache



P0: *Read (X)*
P1: *Read(X)*
P1: *Write (X, 10)*
P0: *Read (X)*
P2: *Read (X)*
P2: *Write (X, 20)*

- X=5 in P0, cache line in clean state
- X=10 in P1, cache line in dirty (modified) state

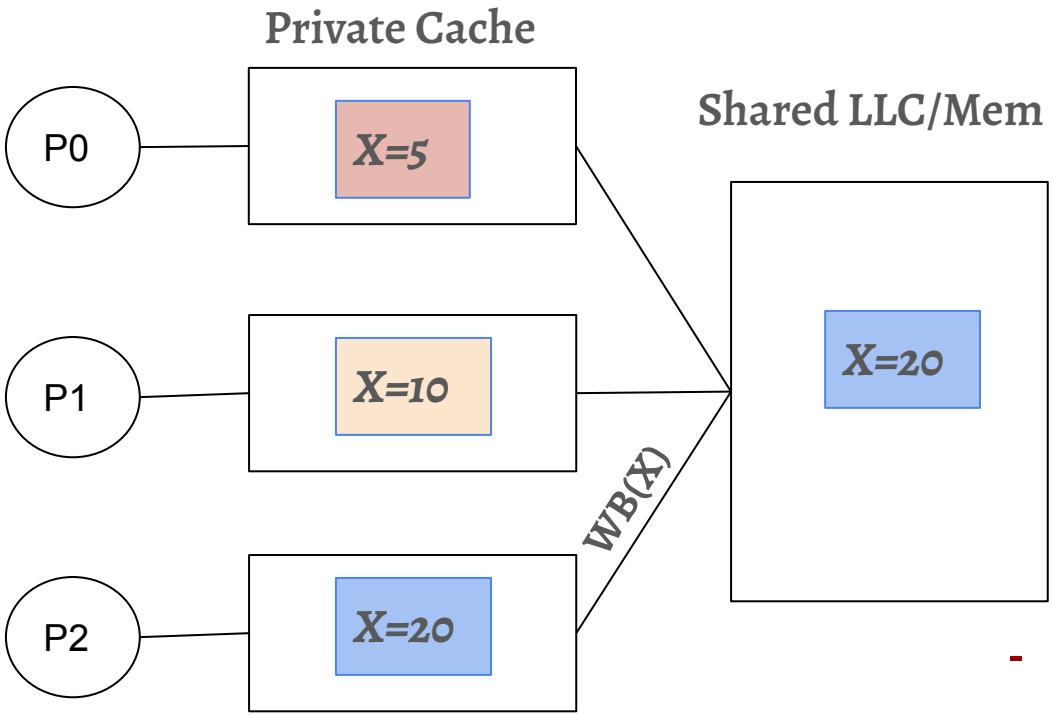
Cache Coherence Example: Write-back Cache



P0: *Read (X)*
P1: *Read(X)*
P1: *Write (X, 10)*
P0: *Read (X)*
P2: *Read (X)*
P2: *Write (X, 20)*

- In P2, cache block is in dirty state
- Write-back can happen in any order

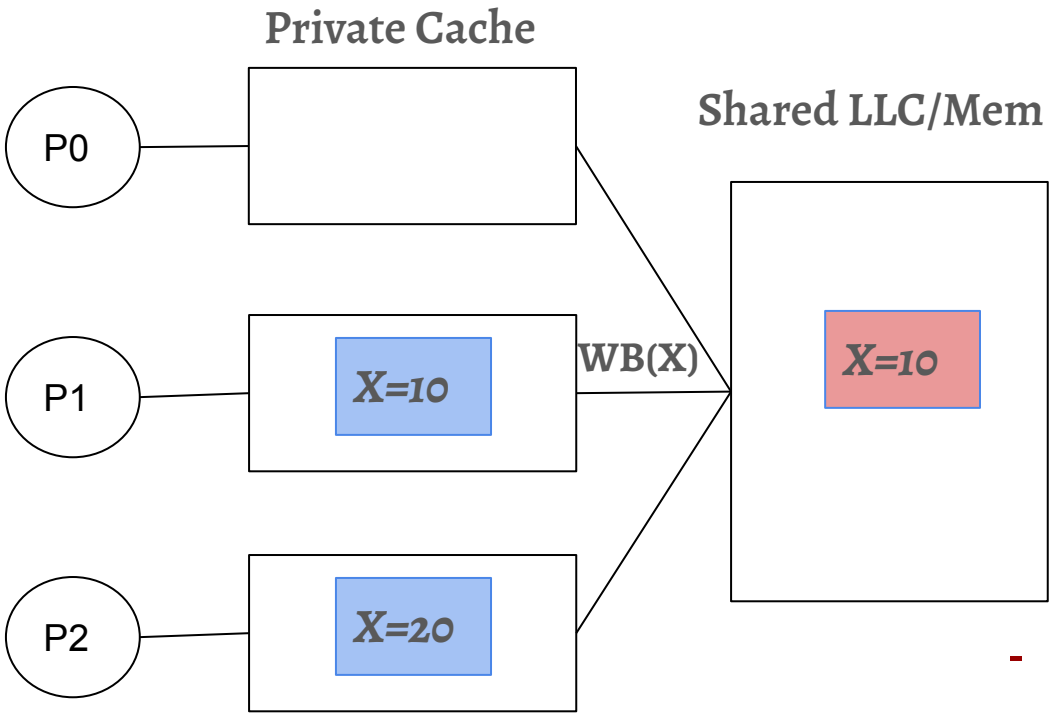
Cache Coherence Example: Write-back Cache



P0: *Read (X)*
P1: *Read(X)*
P1: *Write (X, 10)*
P0: *Read (X)*
P2: *Read (X)*
P2: *Write (X, 20)*

- In memory X=20 (which is correct as per the order of operations for now)

Cache Coherence Example: Write-back Cache



P0: *Read (X)*
P1: *Read(X)*
P1: *Write (X, 10)*
P0: *Read (X)*
P2: *Read (X)*
P2: *Write (X, 20)*

- Value of X is permanently wrong in the memory
- P0 and P1 both will read 10

Cache Coherence: Main issues

- Allowing stores to update cache blocks in private cache of a core (and in LLC/Mem for WT caches) while the same block is present in another core

1. Invalidate all cached copies before allowing a store to proceed

OR

2. Update all cached copies to reflect the modified content

- (Dis)Advantages of each approach?

Cache Coherence: Main issues

- Allowing stores to update cache blocks in private cache of a core (and in LLC/Mem for WT caches) while the same block is present in another core

1. Invalidate all cached copies before allowing a store to proceed

OR

2. Update all cached copies to reflect the modified content

- (Dis)Advantages of each approach?
  - Invalidation can make back-to-back stores to same cache line efficient (No coherence operation till the cache line is dirty)
  - Update allows the following read hits in the private cache but invalidation results in a cache miss

Cache Coherence: Main issues

- Allowing stores to update cache blocks in private cache of a core (and in LLC/Mem for WT caches) while the same block is present in another core

1. Invalidate all cached copies before allowing a store to proceed

OR

2. Update all cached copies to reflect the modified content

Cache Coherence: Main issues

- Allowing stores to update cache blocks in private cache of a core (and in LLC/Mem for WT caches) while the same block is present in another core

1. Invalidate all cached copies before allowing a store to proceed

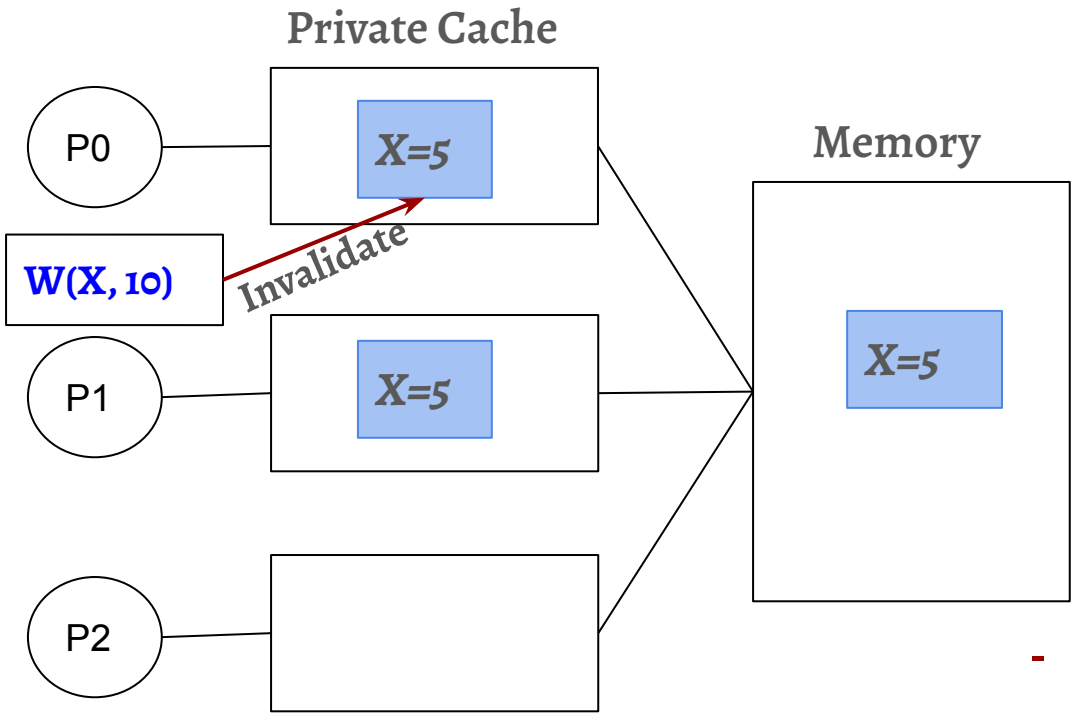
OR

2. Update all cached copies to reflect the modified content

- For write-back caches, the memory state and cache state may not be in sync
- Writeback order must be maintained to ensure correctness

Allow only one copy of the cache line in the dirty state

Analyzing the correct actions



P0: *Read (X)*

P1: *Read(X)*

P1: *Write (X, 10)*

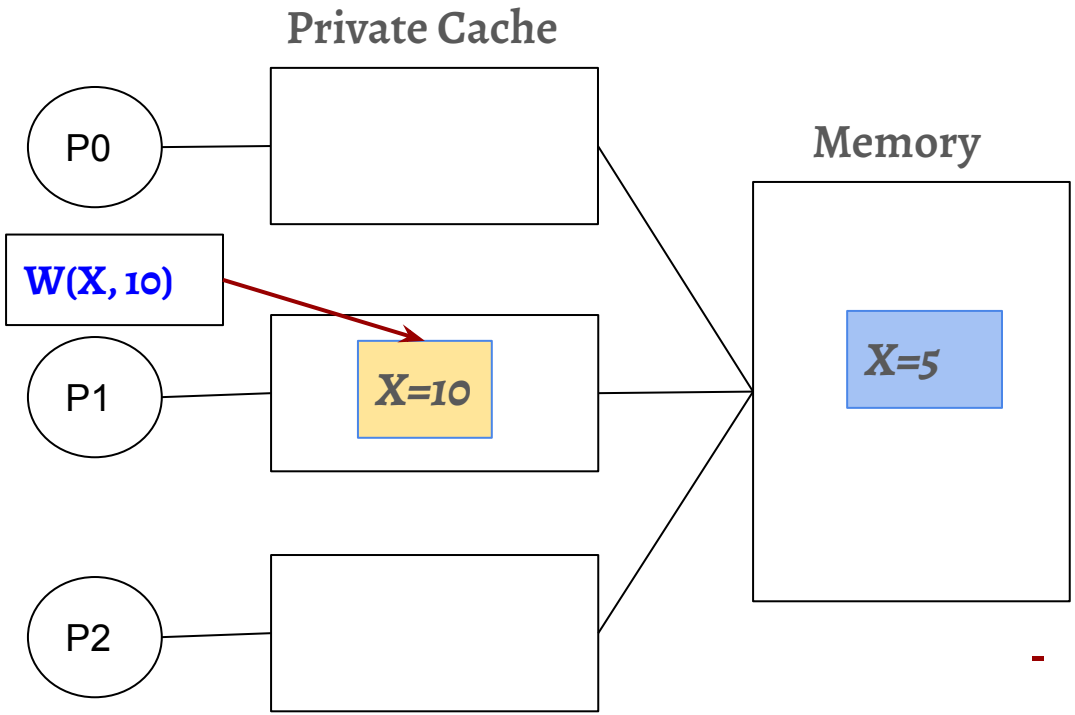
P0: *Read (X)*

P2: *Read (X)*

P2: *Write (X, 20)*

- Invalidate X from P0

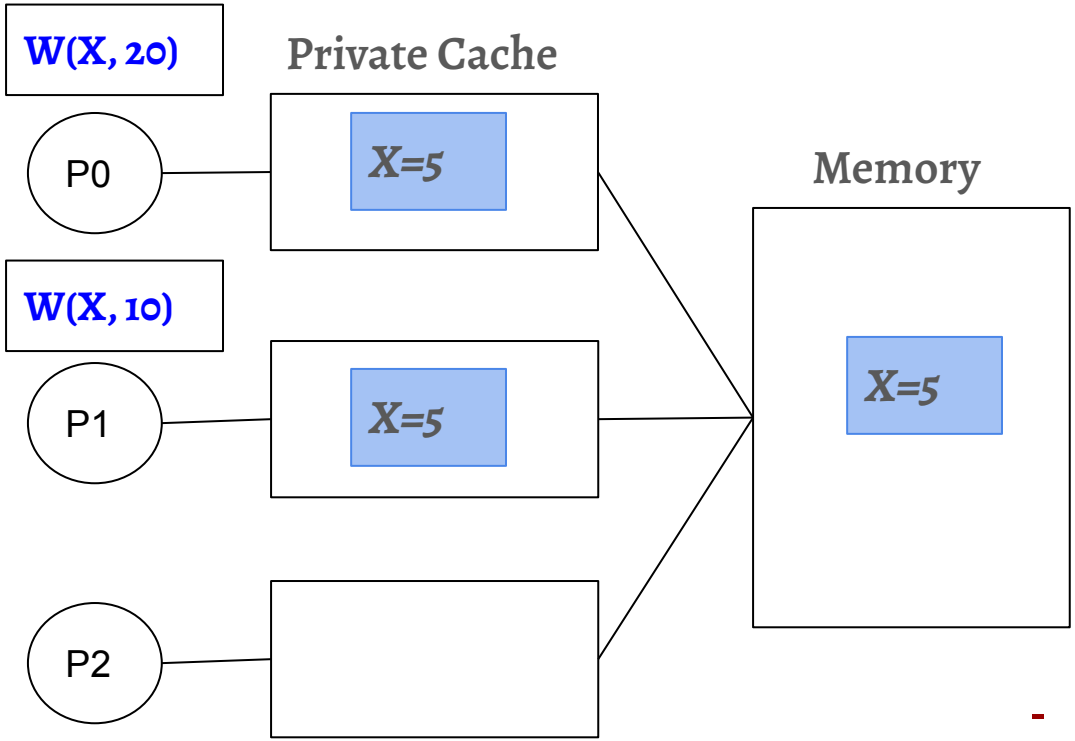
Analyzing the correct actions



P0: *Read (X)*
P1: *Read(X)*
P1: *Write (X, 10)*
P0: *Read (X)*
P2: *Read (X)*
P2: *Write (X, 20)*

- Is there an issues if the write does not wait for invalidation ACK before updating the cache block?

Analyzing the correct actions: concurrent ops



P0: *Read (X)*

P1: *Read(X)*

P1: *Write (X, 10)* P2: *Write (X, 20)*

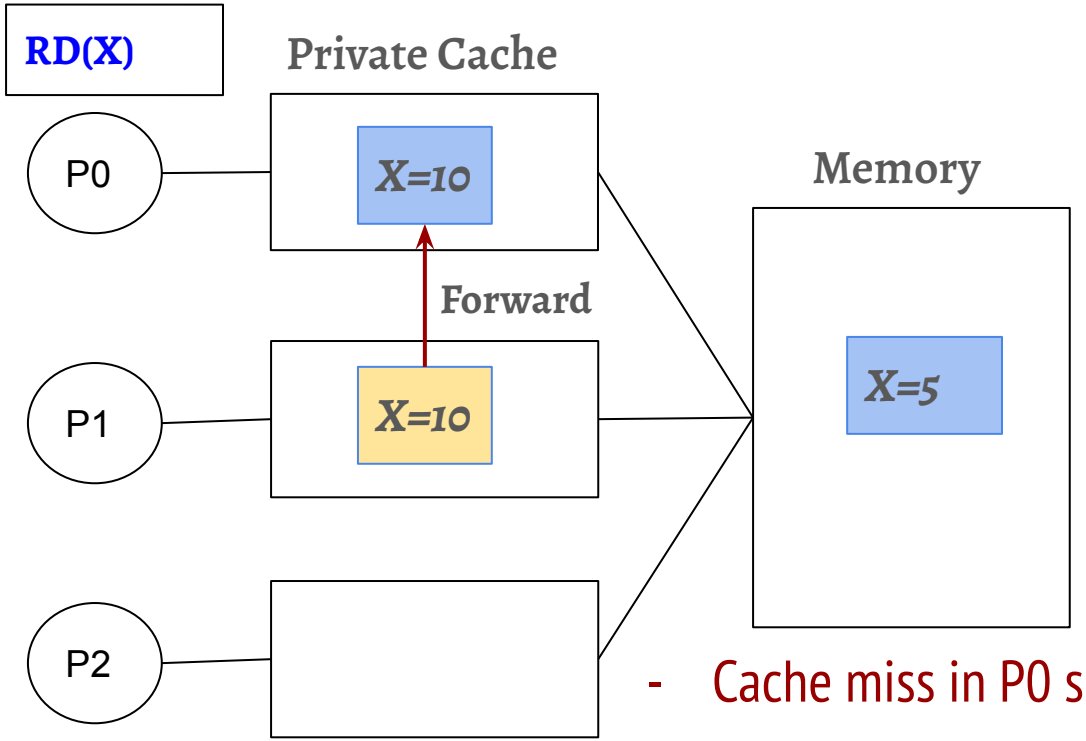
P0: *Read (X)*

P2: *Read (X)*

P2: *Write (X, 20)*

- What could be the issue?

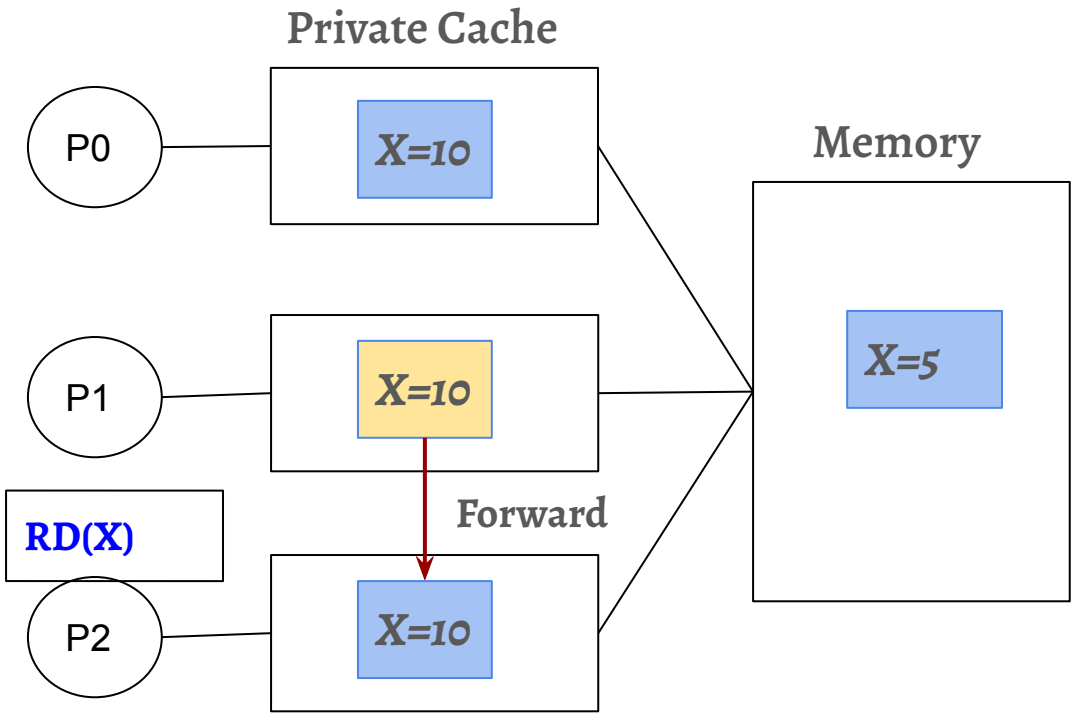
Analyzing the correct actions



P0: *Read (X)*
P1: *Read(X)*
P1: *Write (X, 10)*
P0: *Read (X)*
P2: *Read (X)*
P2: *Write (X, 20)*

- Cache miss in P0 should not be allowed to read from memory
- Force a WB or block forward from P1 (X is clean in P0?)

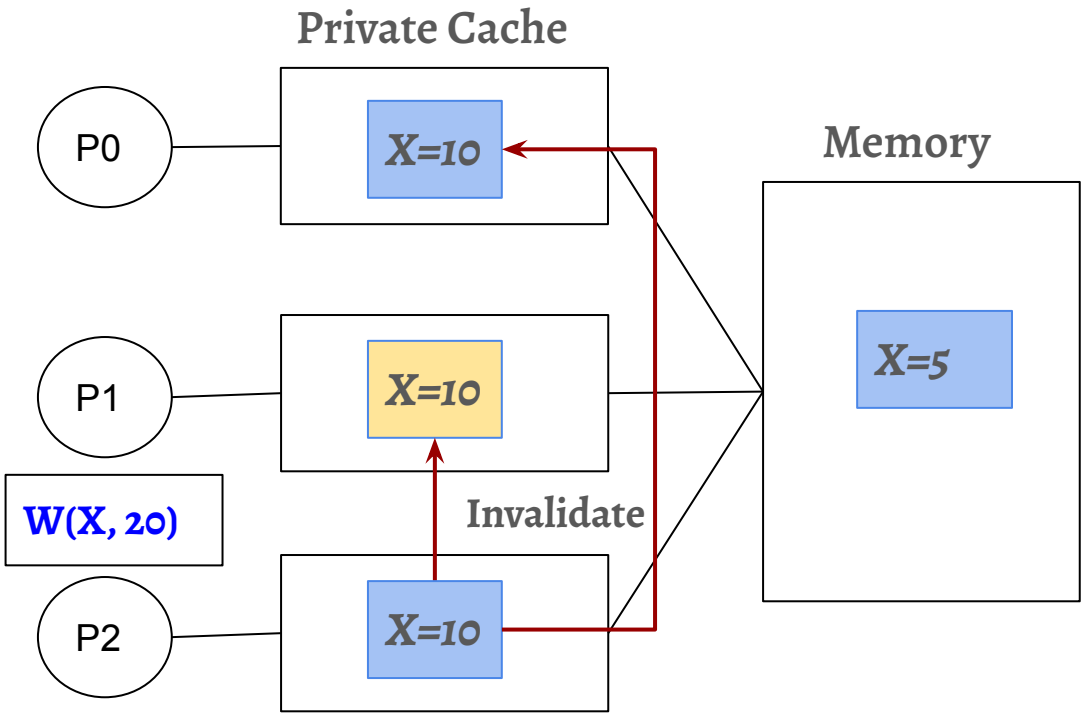
Analyzing the correct actions



P0: *Read (X)*
P1: *Read(X)*
P1: *Write (X, 10)*
P0: *Read (X)*
P2: *Read (X)*
P2: *Write (X, 20)*

- Memory contains a stale value till P1 performs write-back

Analyzing the correct actions



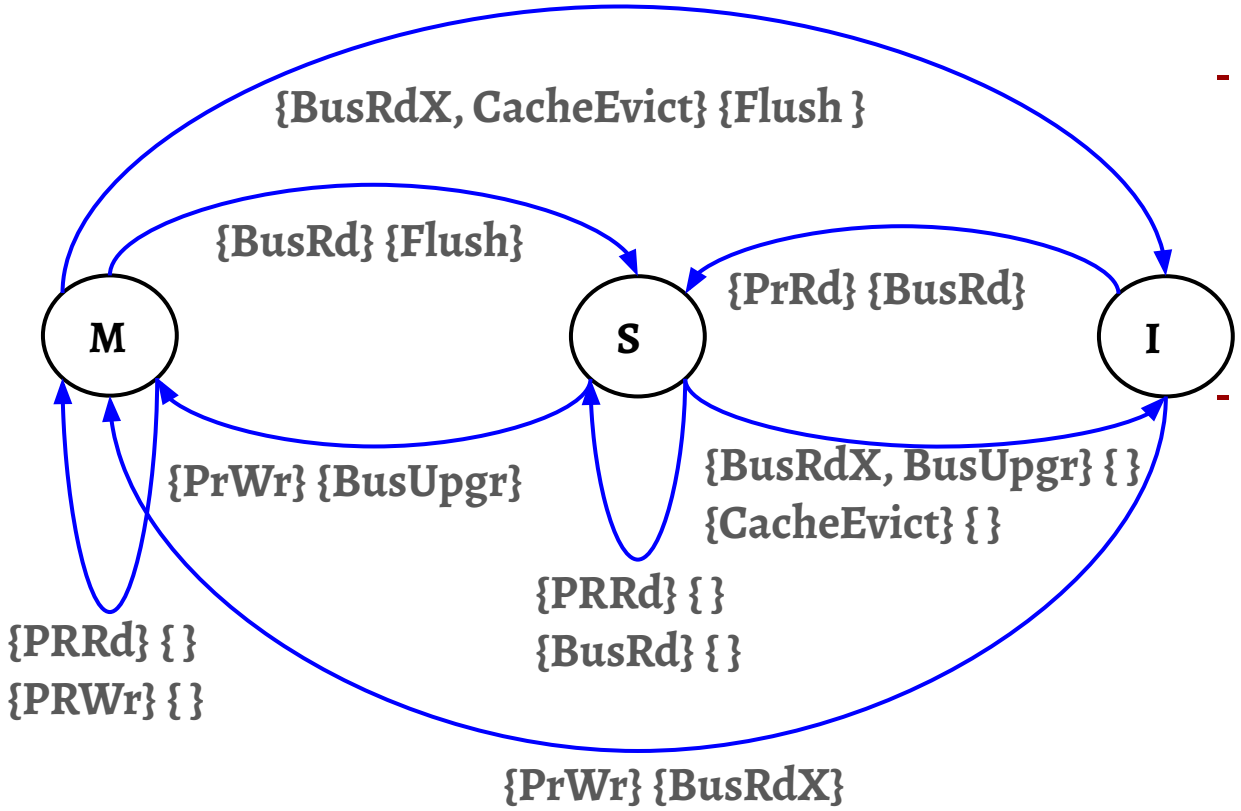
P0: *Read (X)*
P1: *Read(X)*
P1: *Write (X, 10)*
P0: *Read (X)*
P2: *Read (X)*
P2: *Write (X, 20)*

- Does P1 write-back X?
- What if write-back is already in progress?

Cache Coherence using MSI

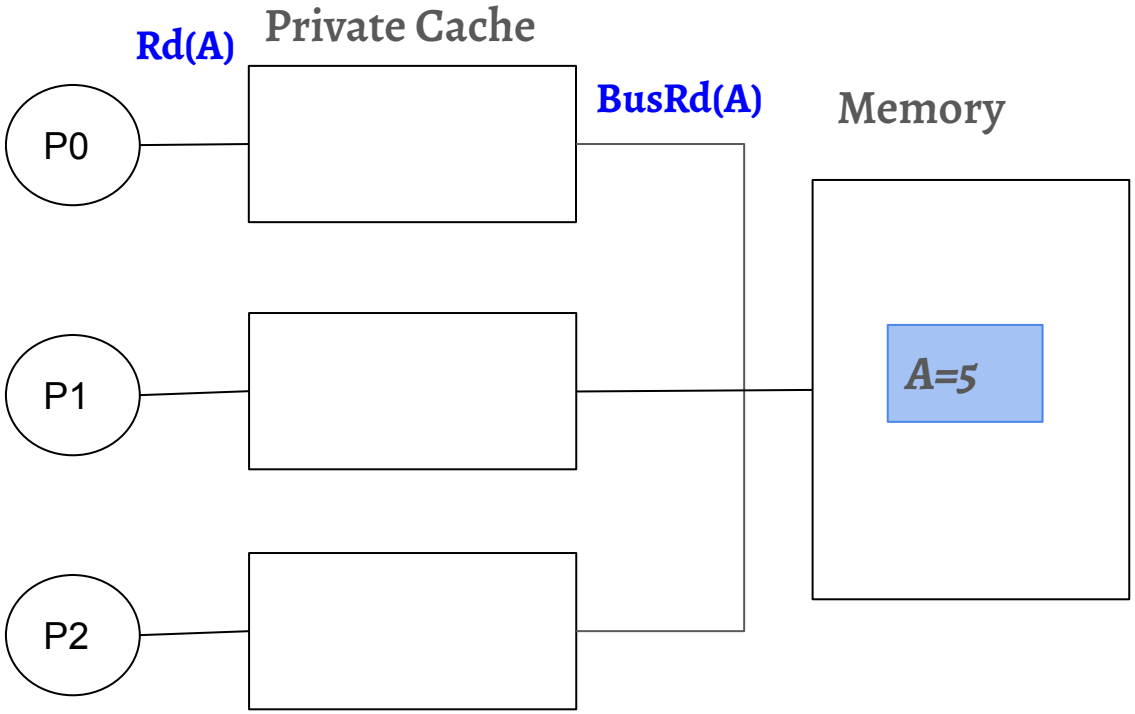
- Every processor maintains the coherence information for every cached block (one way is to maintain along with tag)
- A cached block can be in any of the three states: M, S and I
  - Any cache block can be in S state in more than one processors
  - At most one processor can have a cache block in M state
  - If none of the copies of a block in in private cache, memory must provide a clean copy on a read miss
- There can be types of messages
  - Generated due to an action from the processor
  - Broadcasted over the bus from a core

MSI Protocol - State Transition



- BusRdX: Signifies an exclusive (write) from a processor for an invalid block
- BusUpgr message is sent when a core wants to modify a block that is in shared state

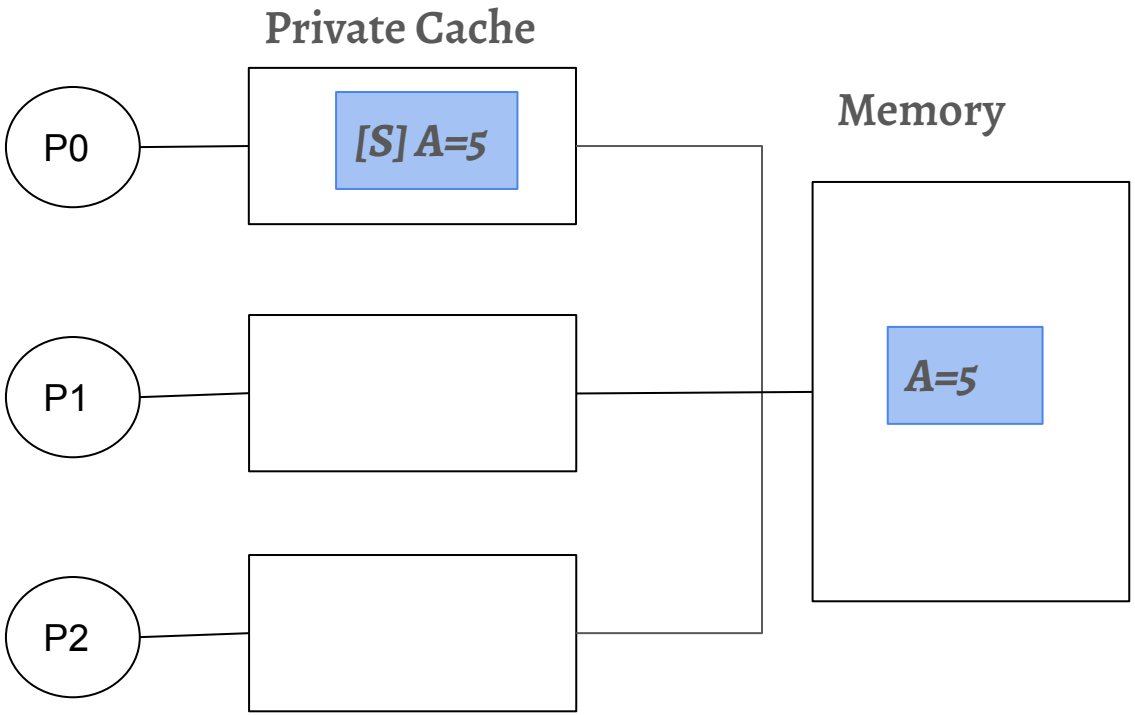
MSI Example Working: Write-back cache



P0: *Read (A)*
P1: *Read(A)*
P1: *Write (A, 10)*
P0: *Read (A)*
P2: *Write (A, 20)*
P1: *Write (A, 30)*

- Assume the cache line is in invalid state in all processors at start
- 'BusRd' message broadcasted in the bus, initiate memory read

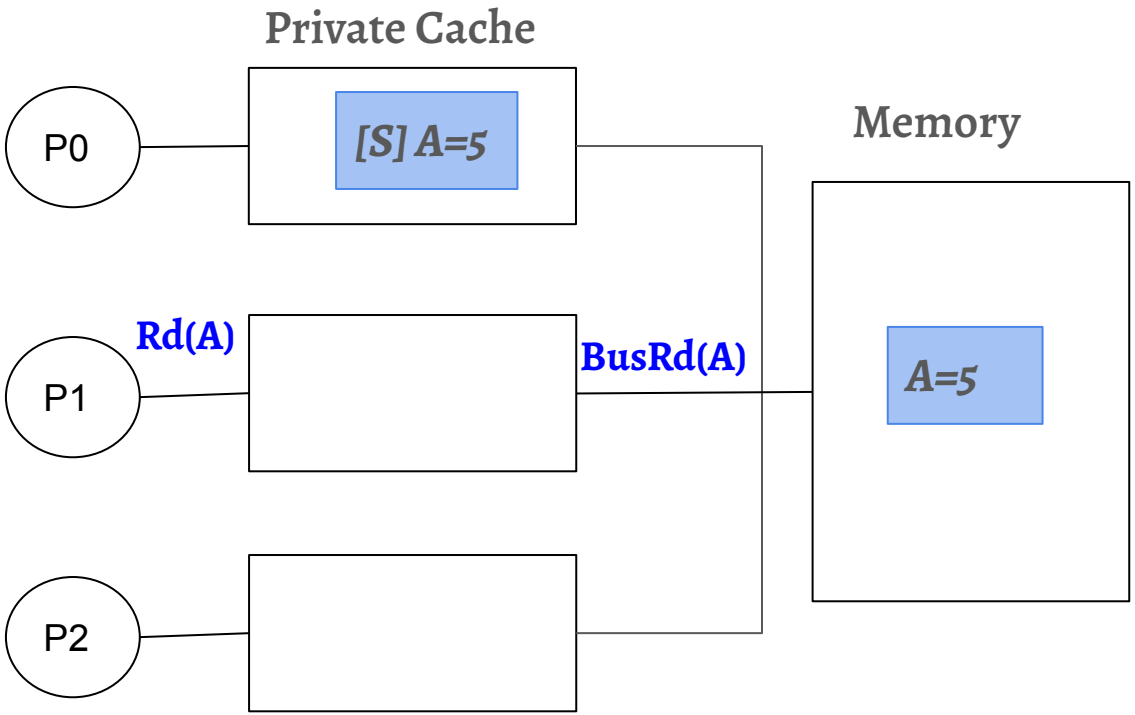
MSI Example Working



P0: *Read (A)*
P1: *Read(A)*
P1: *Write (A, 10)*
P0: *Read (A)*
P2: *Write (A, 20)*
P1: *Write (A, 30)*

- The block is in shared state in P0's private cache

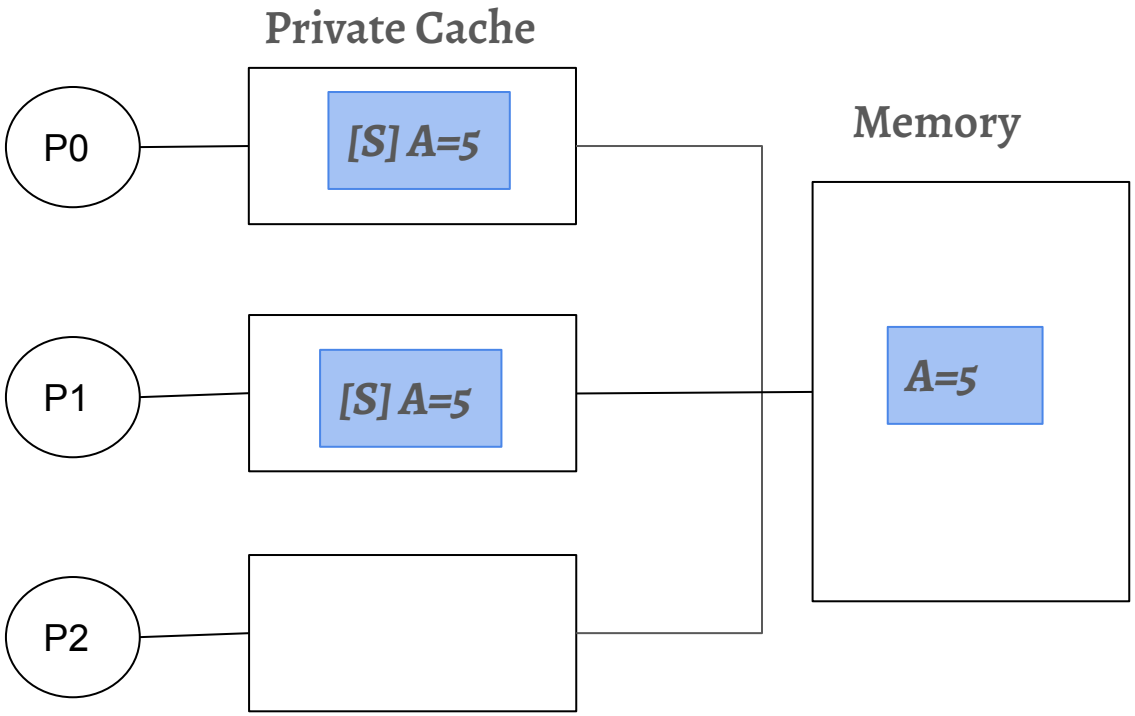
MSI Example Working



P0: *Read (A)*
P1: *Read(A)*
P1: *Write (A, 10)*
P0: *Read (A)*
P2: *Write (A, 20)*
P1: *Write (A, 30)*

- P1 generate 'BusRd', data read from memory
- Cache block in P0 remains in ***S*** state

MSI Example Working



P0: *Read (A)*

P1: *Read(A)*

P1: *Write (A, 10)*

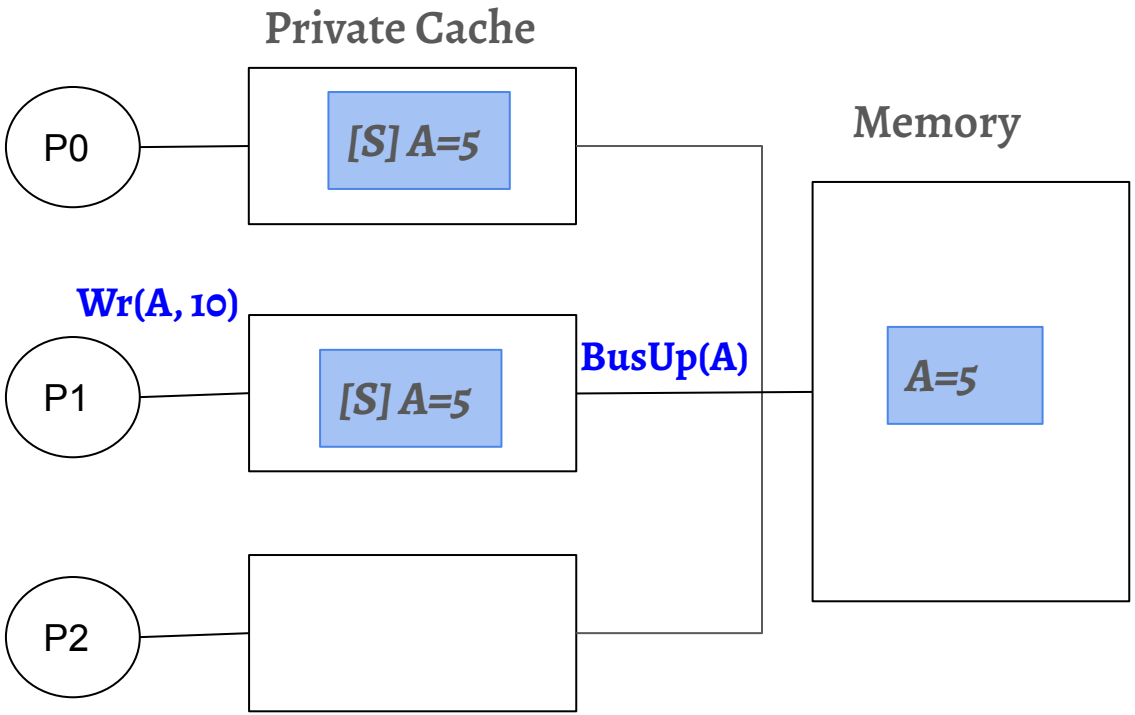
P0: *Read (A)*

P2: *Write (A, 20)*

P1: *Write (A, 30)*

- Cache block state in P1 is ***S*** now

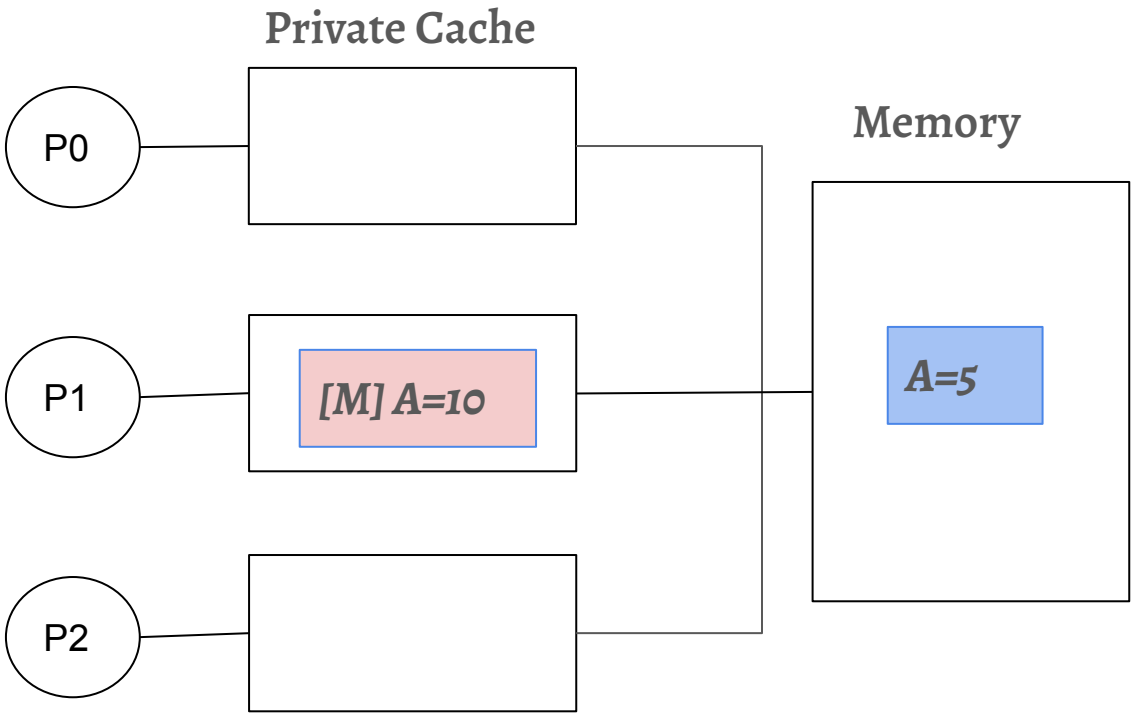
MSI Example Working



P0: *Read (A)*
P1: *Read(A)*
P1: *Write (A, 10)*
P0: *Read (A)*
P2: *Write (A, 20)*
P1: *Write (A, 30)*

- P1 broadcasts bus upgrade message, P0 invalidates the block
- Memory does not respond to this message

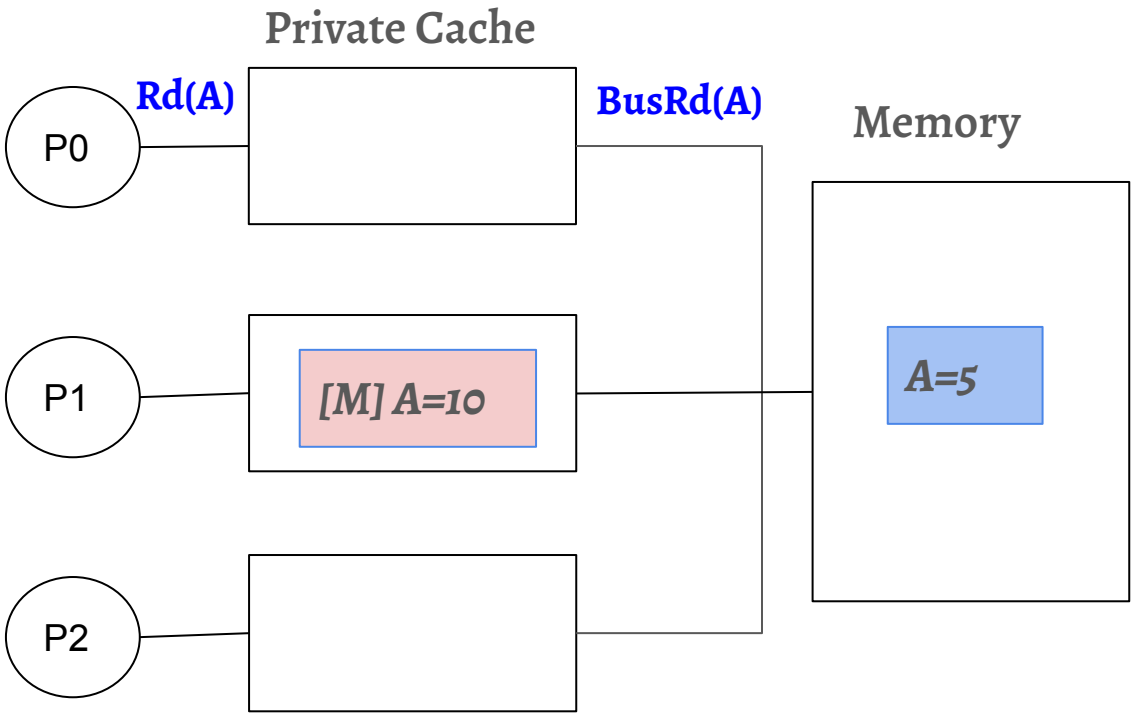
MSI Example Working



P0: *Read (A)*
P1: *Read(A)*
P1: *Write (A, 10)*
P0: *Read (A)*
P2: *Write (A, 20)*
P1: *Write (A, 30)*

- P1 sets the cache block state to ***M***

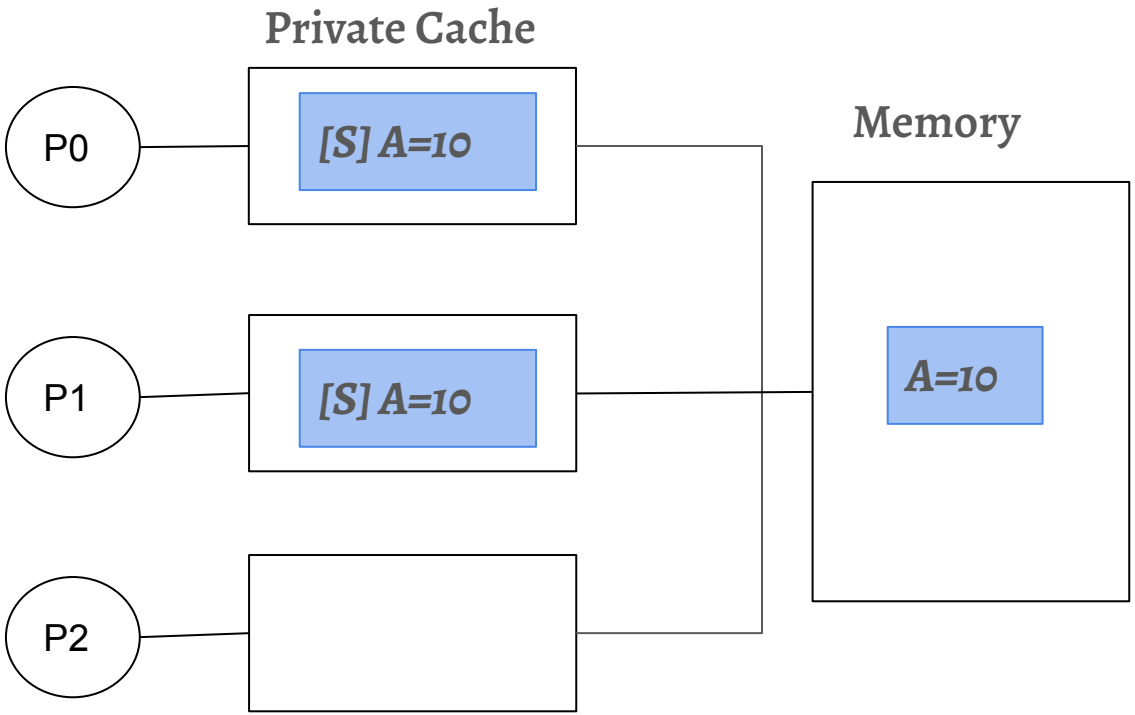
MSI Example Working



P0: *Read (A)*
P1: *Read(A)*
P1: *Write (A, 10)*
P0: *Read (A)*
P2: *Write (A, 20)*
P1: *Write (A, 30)*

- Block is invalid state in P0, generates a BusRd request
- P1 flushes the block for write-back, goes to ***S*** state

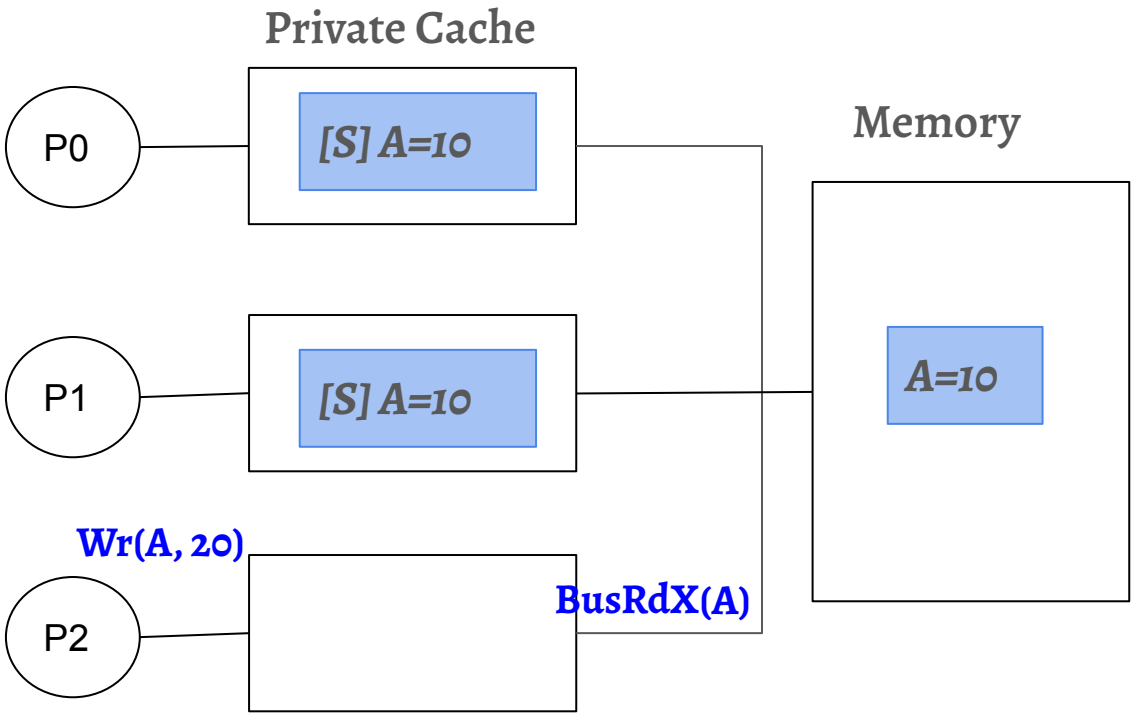
MSI Example Working



P0: *Read (A)*
P1: *Read(A)*
P1: *Write (A, 10)*
P0: *Read (A)*
P2: *Write (A, 20)*
P1: *Write (A, 30)*

- Cache block is in ***S*** state in both P0 and P1, memory is updated with the updated content

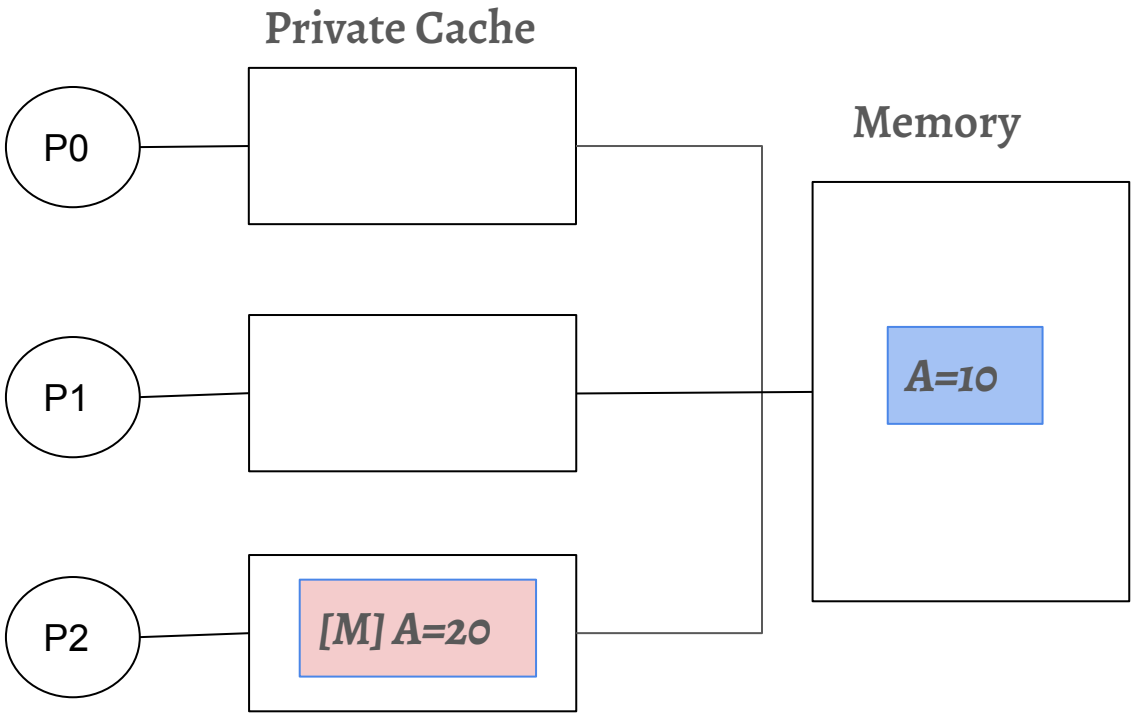
MSI Example Working



P0: *Read (A)*
P1: *Read(A)*
P1: *Write (A, 10)*
P0: *Read (A)*
P2: *Write (A, 20)*
P1: *Write (A, 30)*

- P2 generates BusRdX message, P0 and P1 invalidate, memory provides the cache line

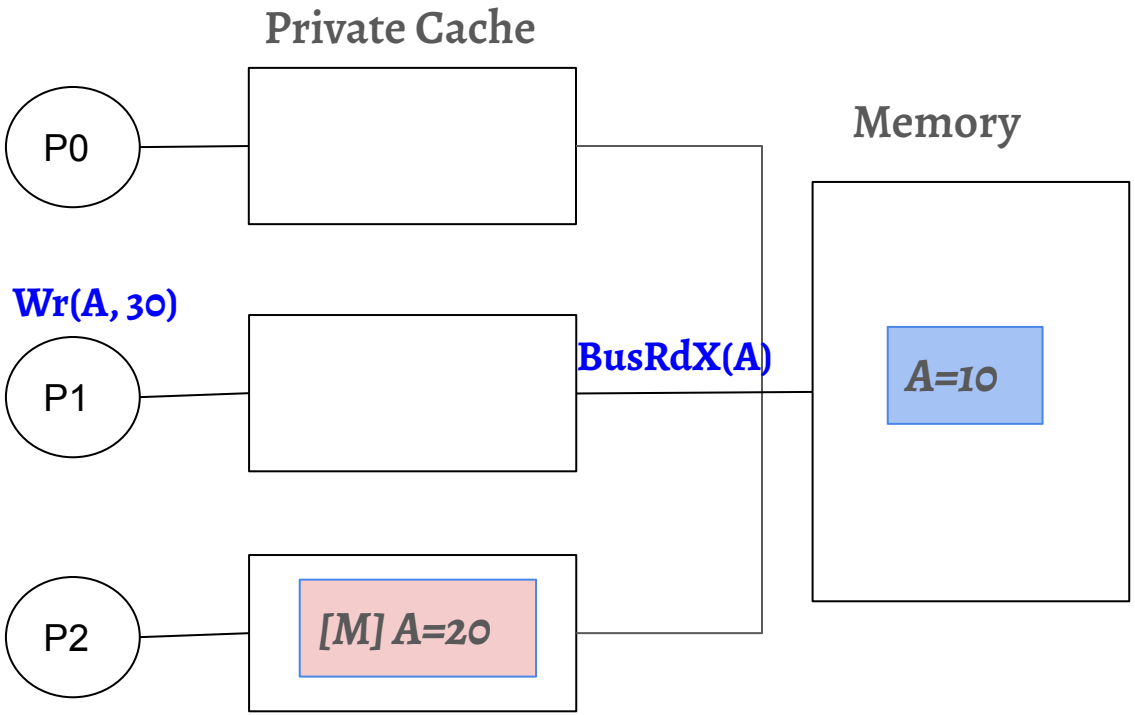
MSI Example Working



P0: *Read (A)*
P1: *Read(A)*
P1: *Write (A, 10)*
P0: *Read (A)*
P2: *Write (A, 20)*
P1: *Write (A, 30)*

- P2 performs write, sets the cache state to ***M***

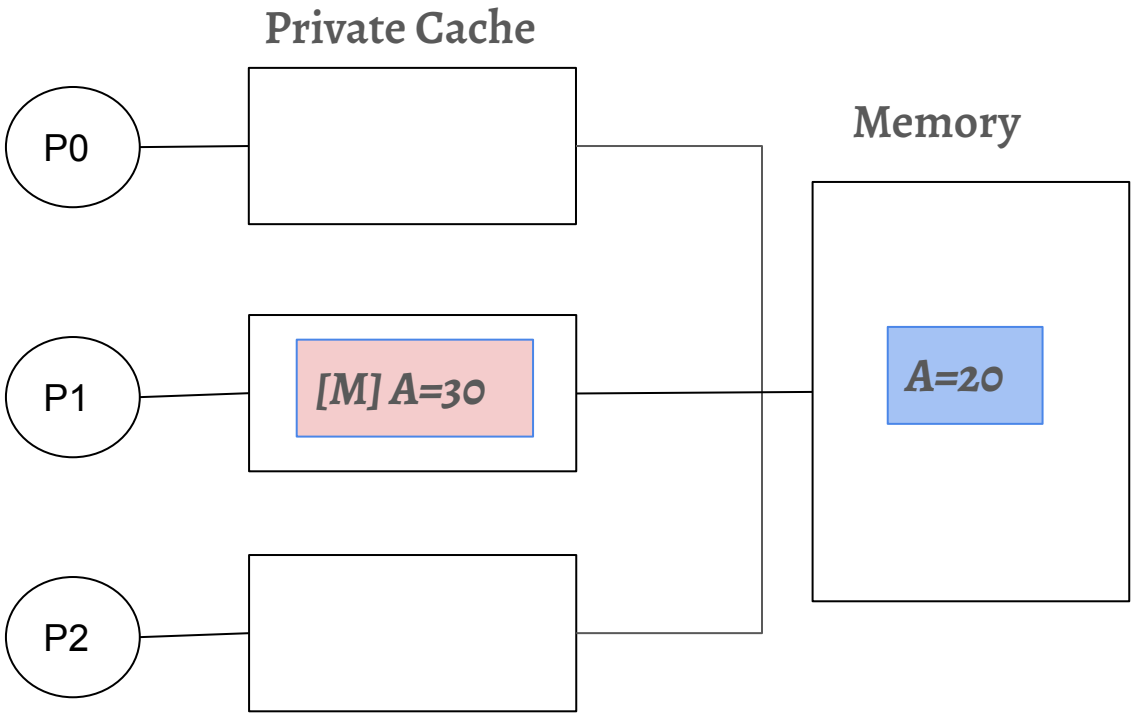
MSI Example Working



P0: *Read (A)*
P1: *Read(A)*
P1: *Write (A, 10)*
P0: *Read (A)*
P2: *Write (A, 20)*
P1: *Write (A, 30)*

- P1 broadcasts BusRdX, P2 flushes its entry for writeback and cache block invalidated

MSI Example Working



P0: *Read (A)*
P1: *Read(A)*
P1: *Write (A, 10)*
P0: *Read (A)*
P2: *Write (A, 20)*
P1: *Write (A, 30)*

- Memory updated with the value written back from P2
- P1 owns the entry in ***M*** state

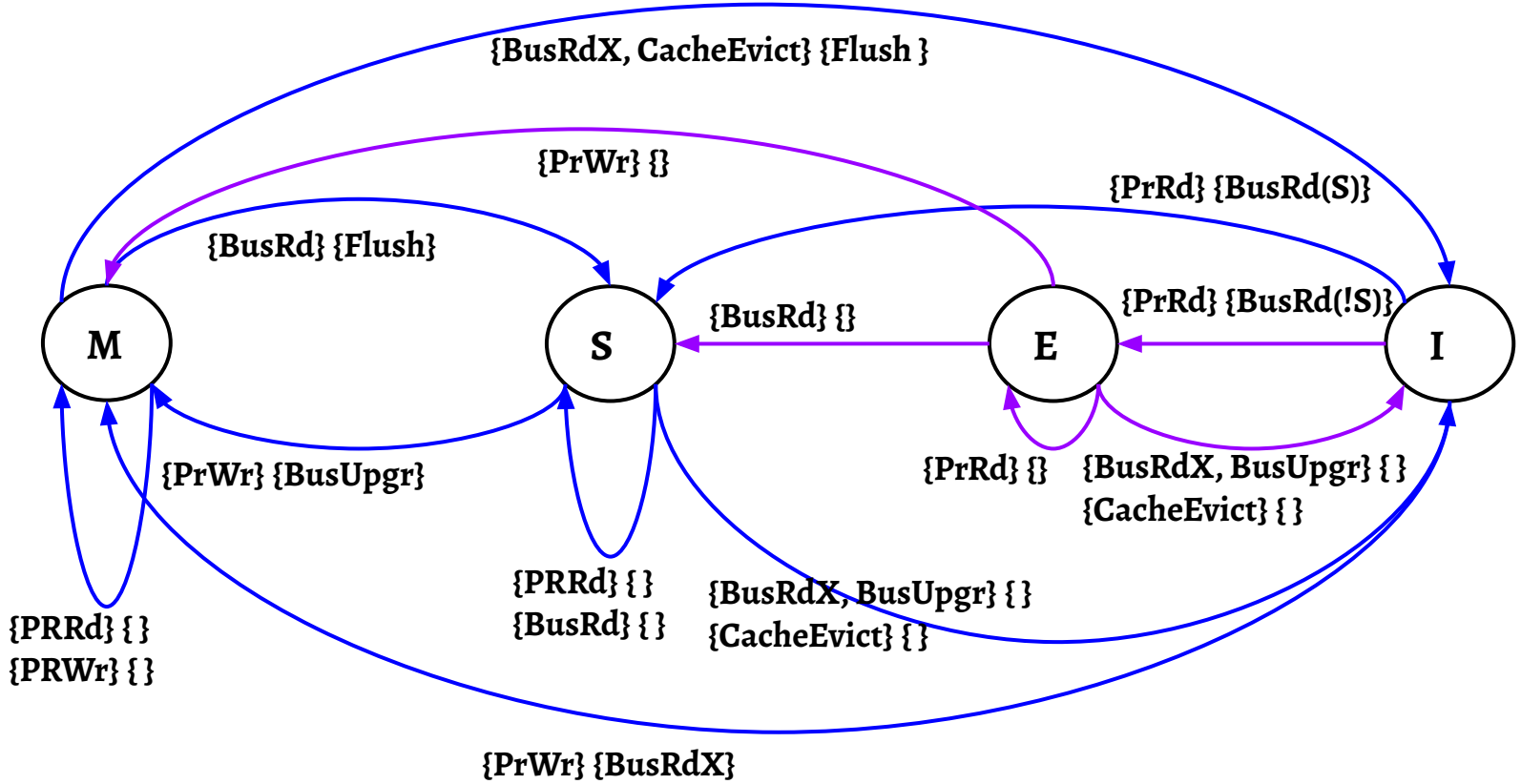
Extending MSI (MESI)

- If the block is not shared across cores, a typical LD followed by ST would require two messages to be broadcasted in MSI
- Solution
  - The first processor reading a block granted “exclusivity” i.e., state ***E***
  - As long as the processor is in E state, it can be written without any bus transaction
- Need extra support from the memory side
  - Check if the block is not in shared state in other cores

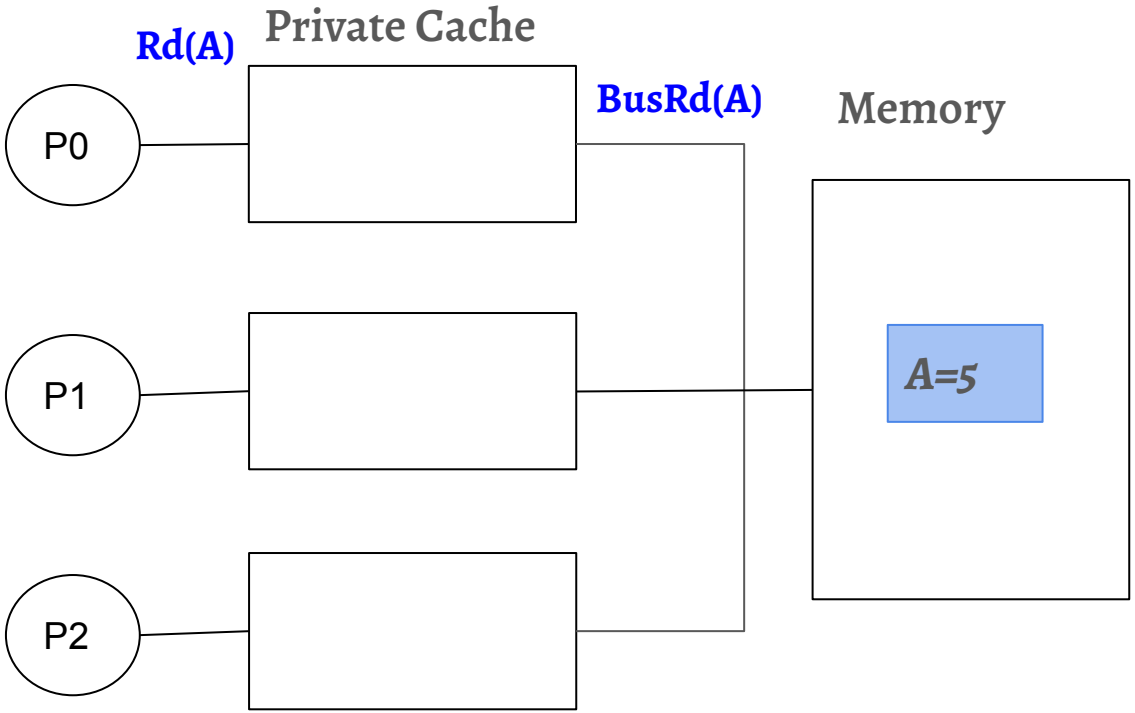
Cache Coherence using MESI

- Reduce the overhead of read followed by write for blocks not shared by others
  - Typical multi-process workload access fits this pattern
- Introduces a new state 'E' → exclusive clean (exclusive unowned)
  - If no other processor cache has a copy, the block is marked 'E' on first read
  - When the block is written, it transitions from E to M (without any bus message)
  - If any other processor reads, the block transitions from E to S
- To determine the state of a block during the first block read, the coherence controller should know if other private caches have the block
  - On a 'BusRead', other processors having the cache generate "shared" signal
  - The requesting processor checks the ORed 'shared' value to decide 'E' or 'S' state for the block

MESI Protocol - State Transition



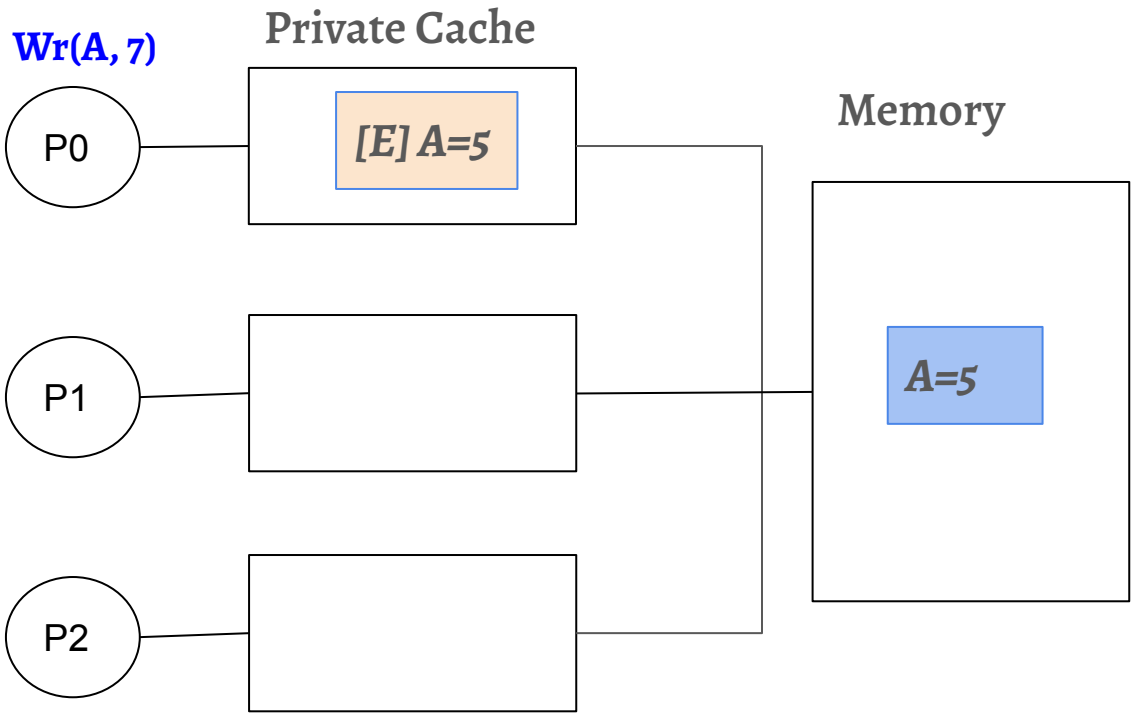
MESI Example Working: Write-back cache



P0: *Read (A)*
P0: *Write(A, 7)*
P1: *Read(A)*
P1: *Write (A, 10)*
P0: *Read (A)*
P2: *Read (A)*

- Assume the cache line is in invalid state in all processors at start
- 'BusRd' message broadcasted in the bus, initiate memory read

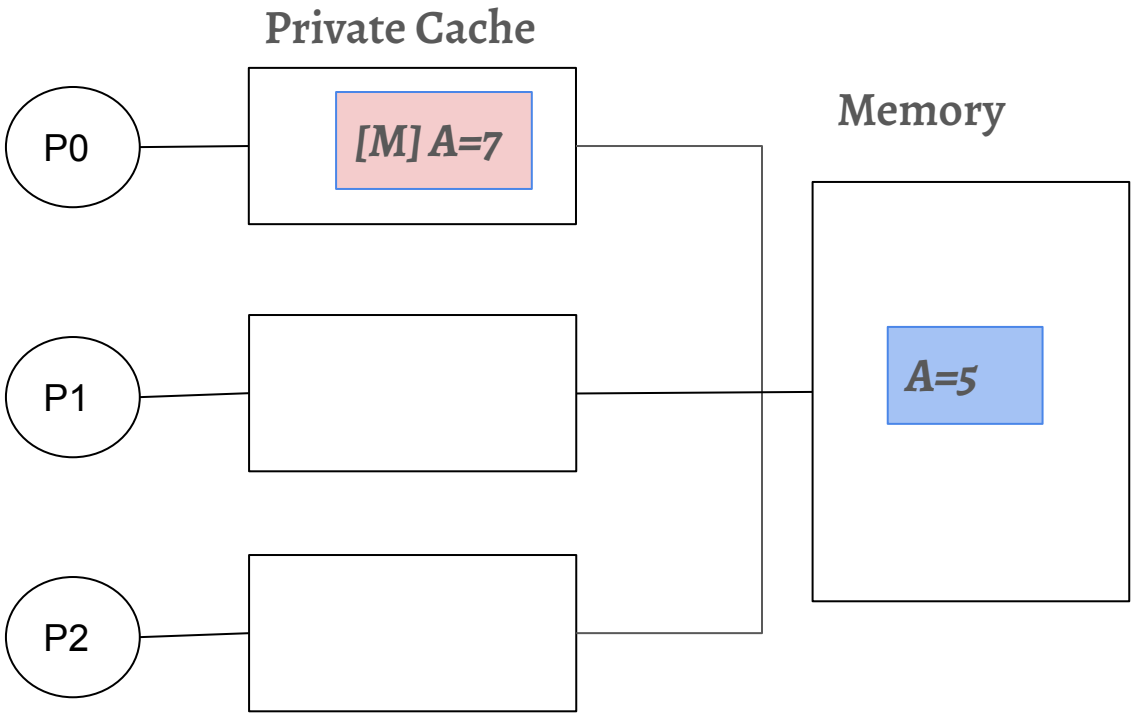
MESI Example Working



P0: *Read (A)*
P0: *Write(A, 7)*
P1: *Read(A)*
P1: *Write (A, 10)*
P0: *Read (A)*
P2: *Read (A)*

- The block is in exclusive clean state at the time of write, it becomes 'M'. No bus message broadcasted.

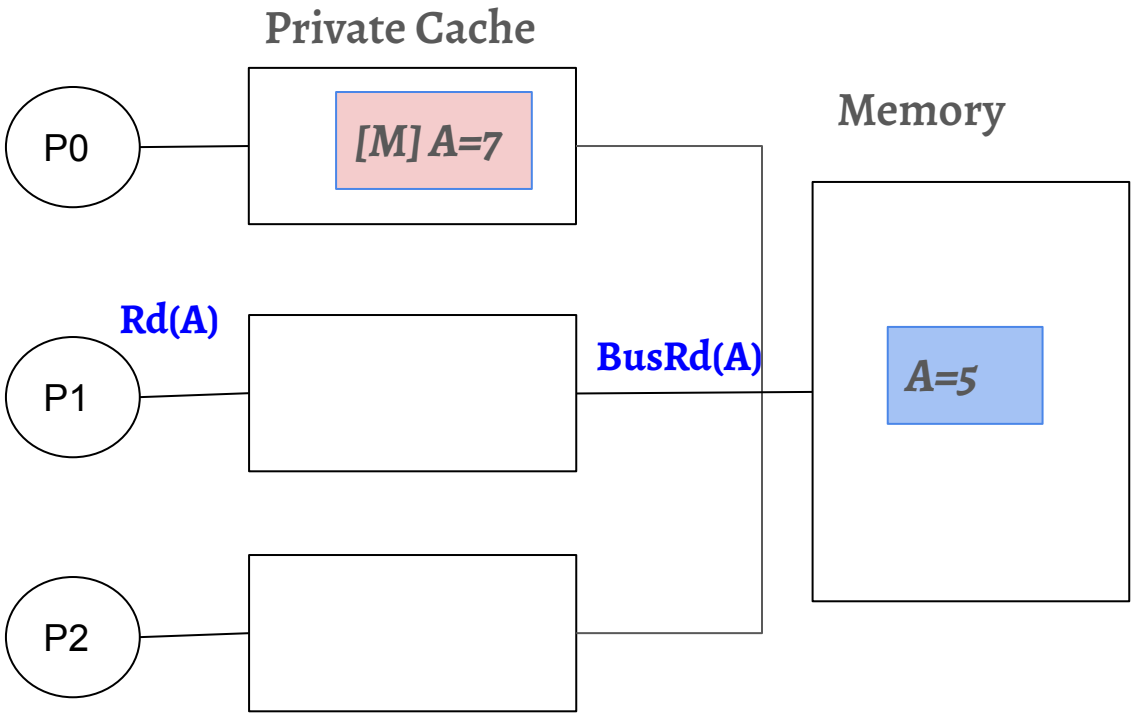
MESI Example Working



P0: *Read (A)*
P0: *Write(A, 7)*
P1: *Read(A)*
P1: *Write (A, 10)*
P0: *Read (A)*
P2: *Read (A)*

- The block is in exclusive clean state at the time of write, it becomes 'M'. No bus message broadcasted.

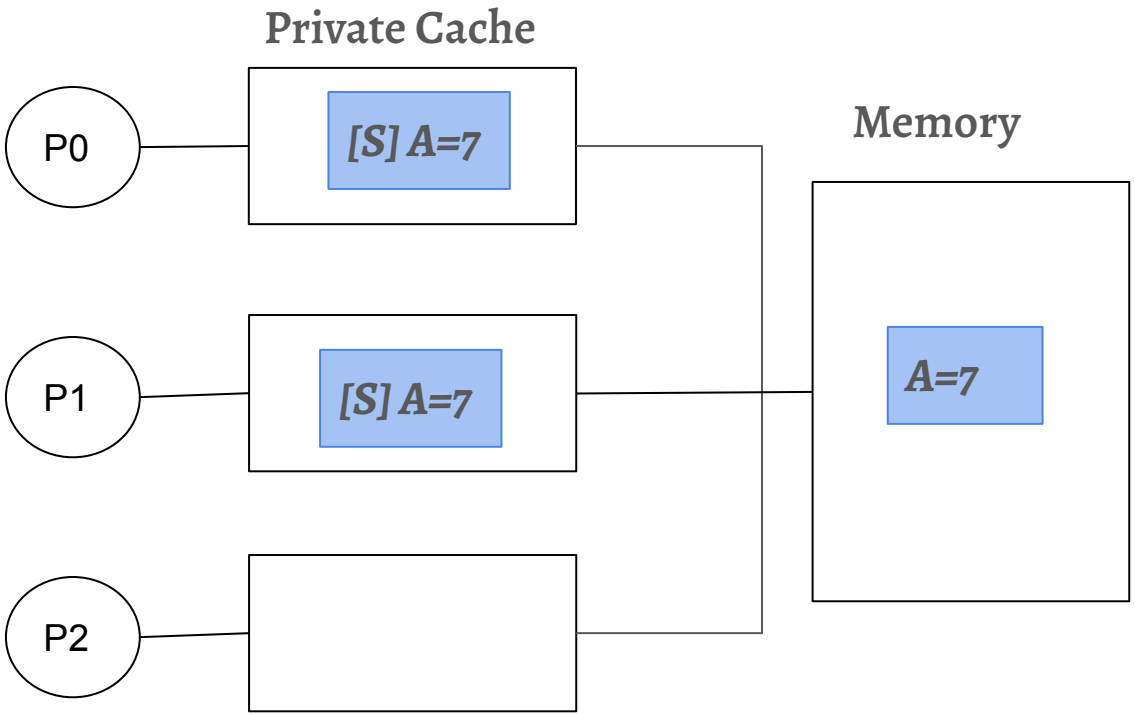
MESI Example Working



P0: *Read (A)*
P0: *Write(A, 7)*
P1: *Read(A)*
P1: *Write (A, 10)*
P0: *Read (A)*
P2: *Read (A)*

- P0 initiates flush (writeback) and moves the block to S state

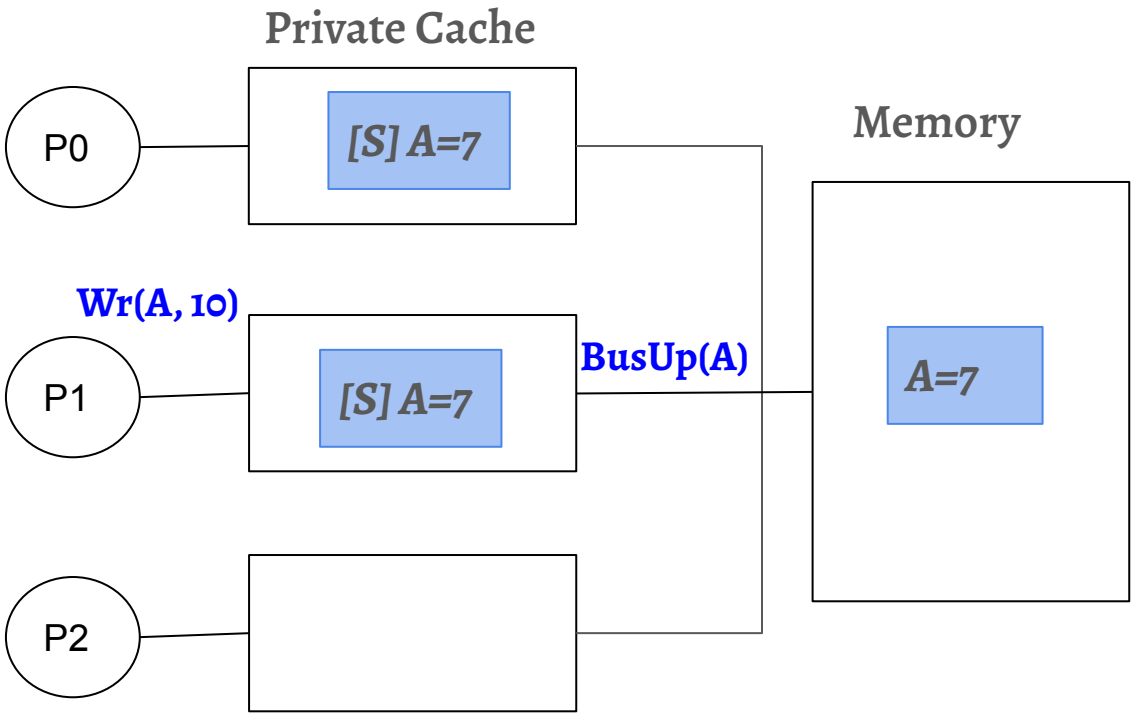
MESI Example Working



P0: *Read (A)*
P0: *Write(A, 7)*
P1: *Read(A)*
P1: *Write (A, 10)*
P0: *Read (A)*
P2: *Read (A)*

- Cache block state in P0 and P1 is ***S*** now

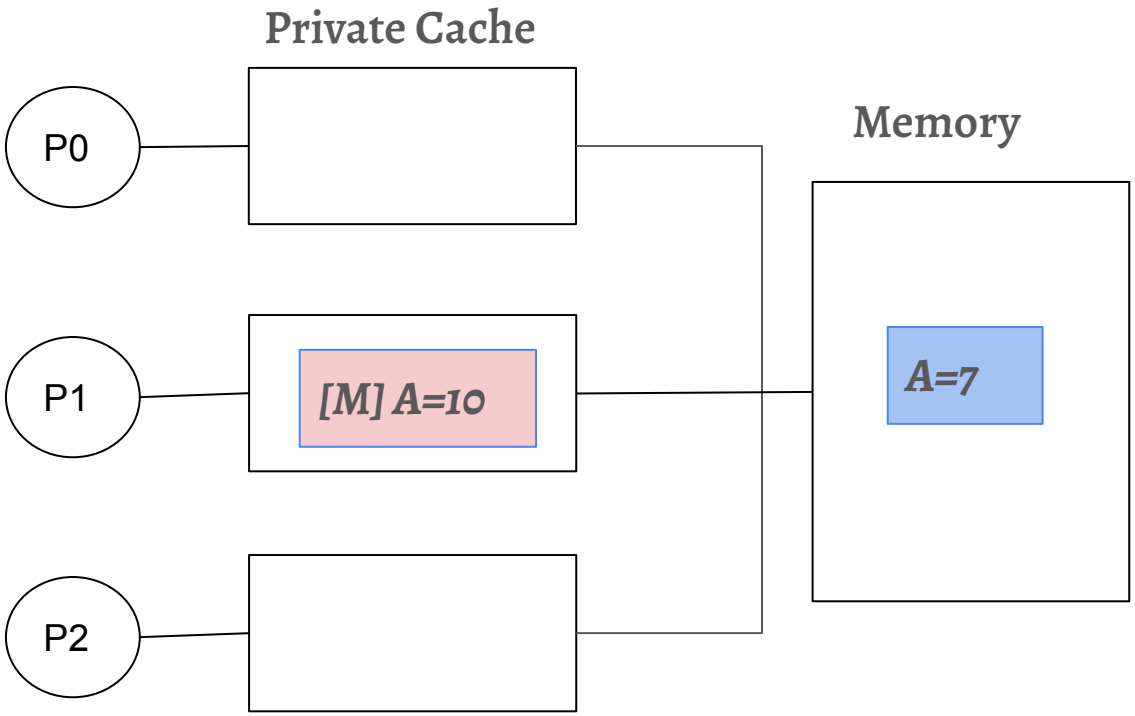
MSI Example Working



P0: *Read (A)*
P0: *Write(A, 7)*
P1: *Read(A)*
P1: *Write (A, 10)*
P0: *Read (A)*
P2: *Read (A)*

- P1 broadcasts bus upgrade message, P0 invalidates the block
- Memory does not respond to this message

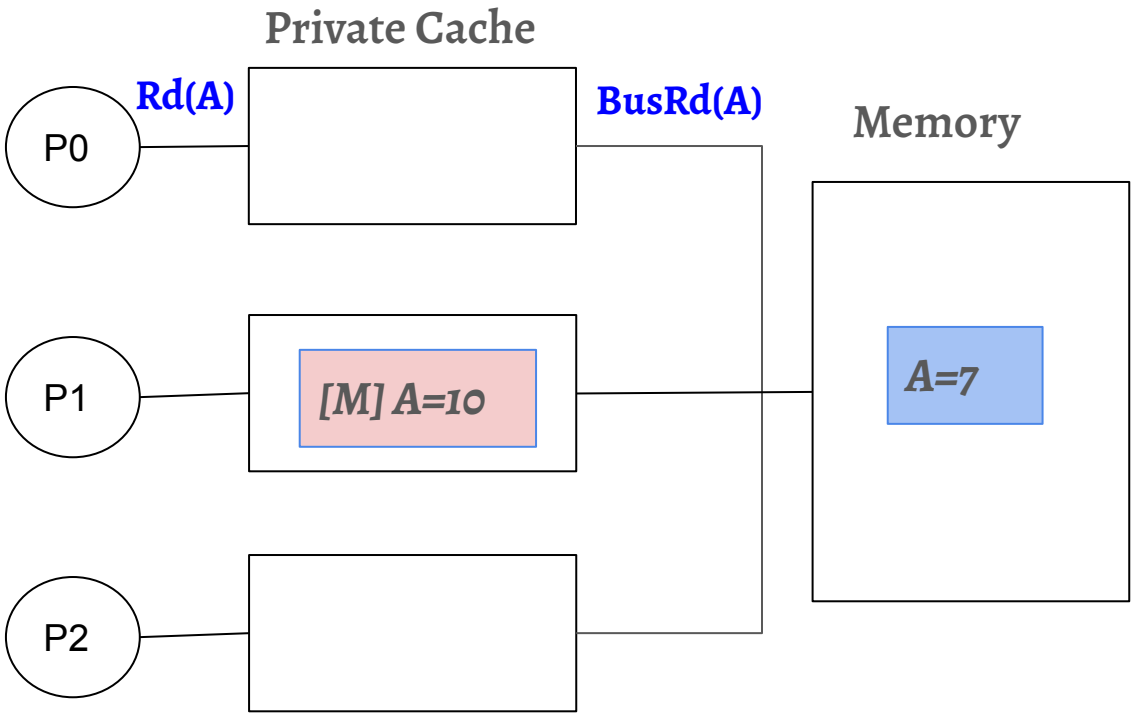
MESI Example Working



P0: *Read (A)*
P0: *Write(A, 7)*
P1: *Read(A)*
P1: *Write (A, 10)*
P0: *Read (A)*
P2: *Read (A)*

- P1 sets the cache block state to ***M***

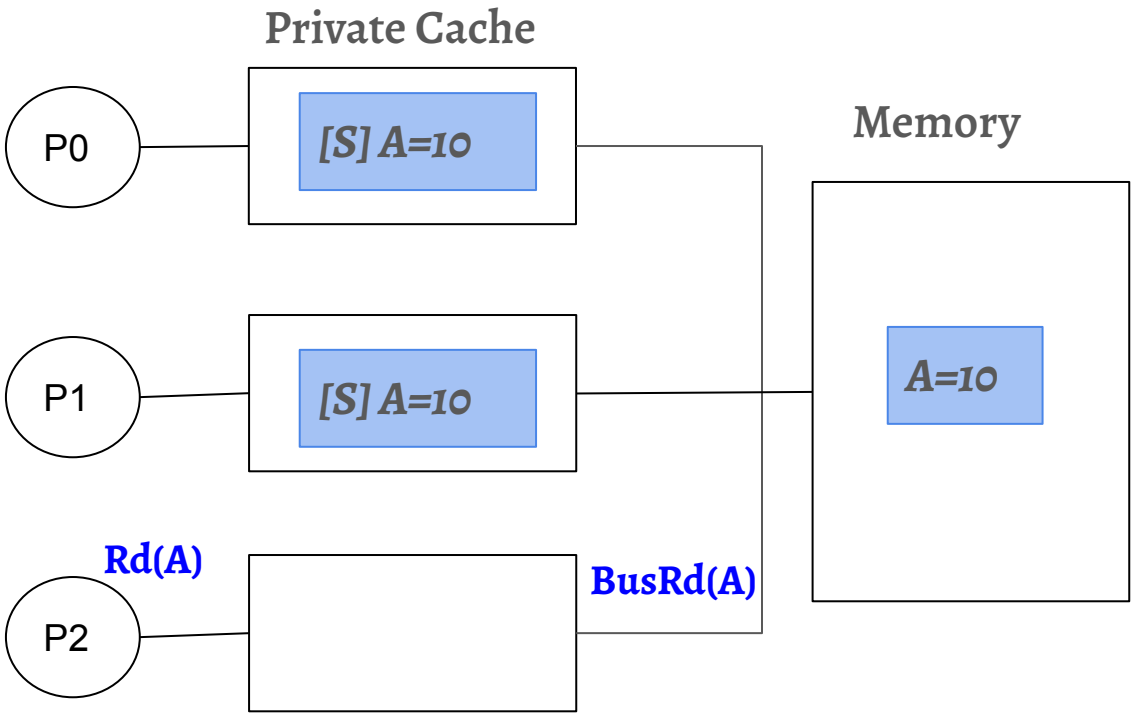
MESI Example Working



P0: *Read (A)*
P0: *Write(A, 7)*
P1: *Read(A)*
P1: *Write (A, 10)*
P0: *Read (A)*
P2: *Read (A)*

- Block is invalid state in P0, generates a BusRd request
- P1 flushes the block for write-back, goes to ***S*** state

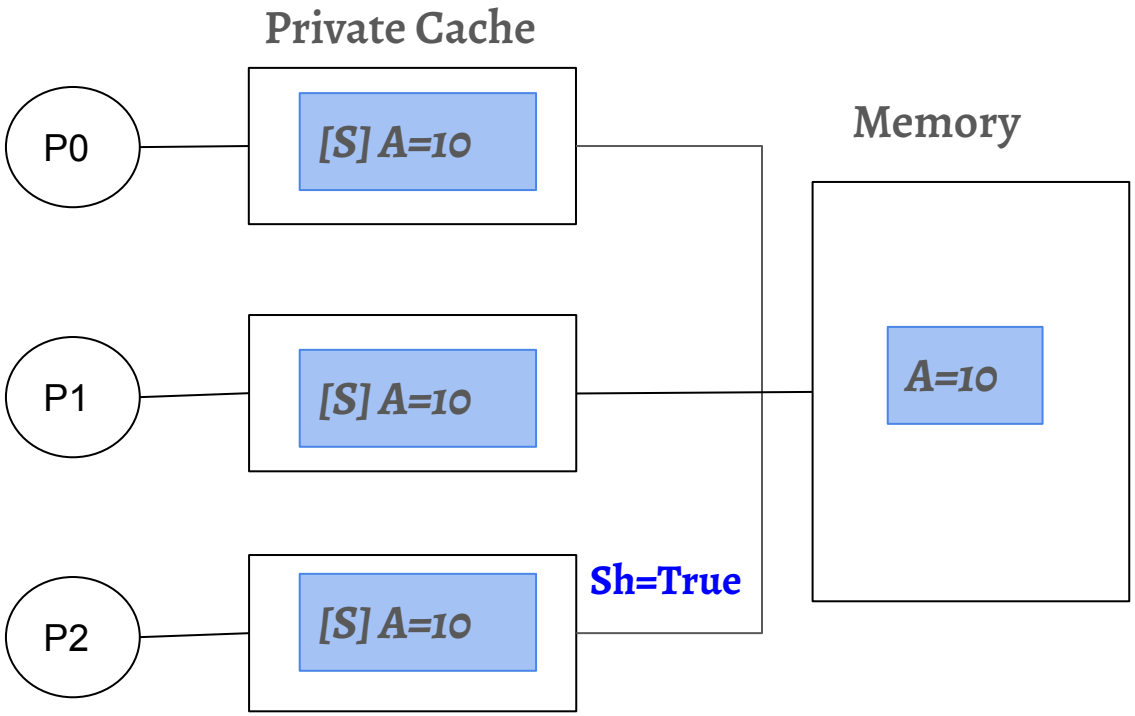
MESI Example Working



P0: *Read (A)*
P0: *Write(A, 7)*
P1: *Read(A)*
P1: *Write (A, 10)*
P0: *Read (A)*
P2: *Read (A)*

- P2 generates BusRead which is in 'S' state in P0 and P1
- P0 and P1 will assert the 'shared' signal

MESI Example Working



P0: *Read (A)*
P0: *Write(A, 7)*
P1: *Read(A)*
P1: *Write (A, 10)*
P0: *Read (A)*
P2: *Read (A)*

- P2 will maintain the block in 'S' state

Limitations of Snooping Coherence Protocols

- Shared bus contention due to broadcasts
  - Memory accesses served from caches generate broadcast messages
  - With increasing number of processors, the contention can lead to increased latency and/or power consumption
- Increased cache access latency due to contention with tag matching for coherence protocol actions. How to reduce the contention?

Limitations of Snooping Coherence Protocols

- Shared bus contention due to broadcasts
  - Memory accesses served from caches generate broadcast messages
  - With increasing number of processors, the contention can lead to increased latency and/or power consumption
- Increased cache access latency due to contention with tag matching for coherence protocol actions. How to reduce the contention?
  - Replicated tag stores
  - Write-through in private cache levels can help to a certain extent