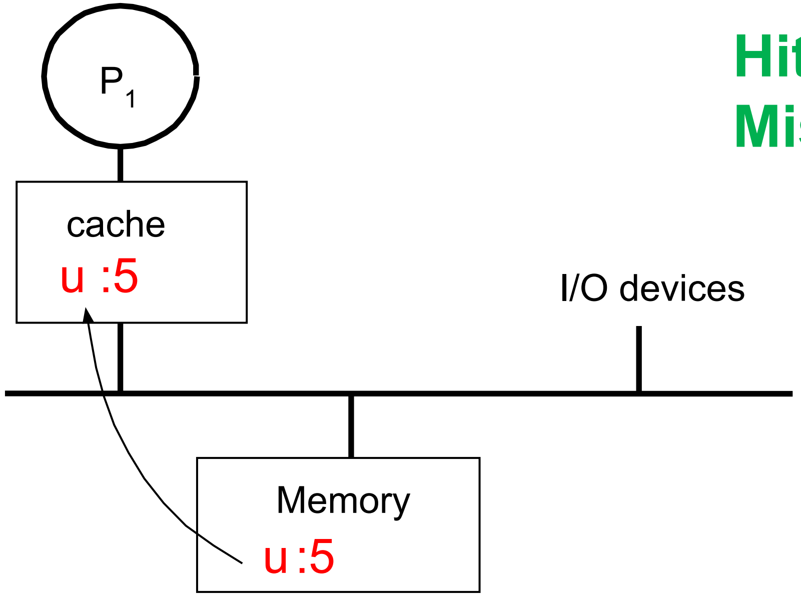
CSC 334 – Parallel and Distributed Computing

Instructor: Dr. M. Hasan Jamal

Lecture# 06: Cache Coherence

Single Processor Caching

P_1 reads u



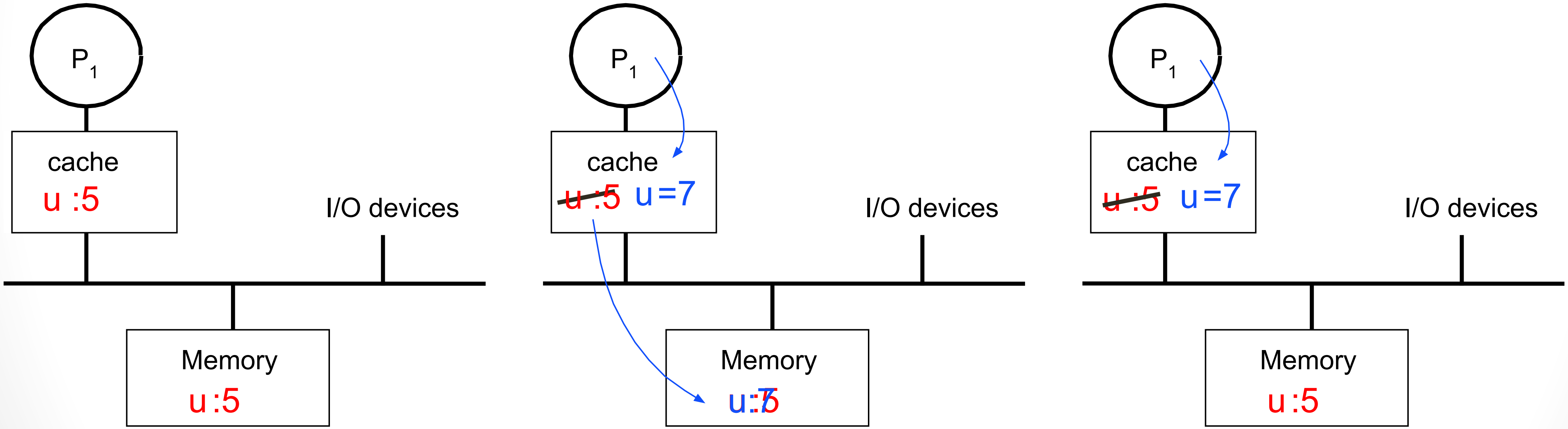
Hit: data in the cache
Miss: data is not in the cache

Hit rate: h
Miss rate: m = (1 – h)

Cache Write Policies

- Writing to Cache in 1 processor case
  - Write Through
  - Write Back

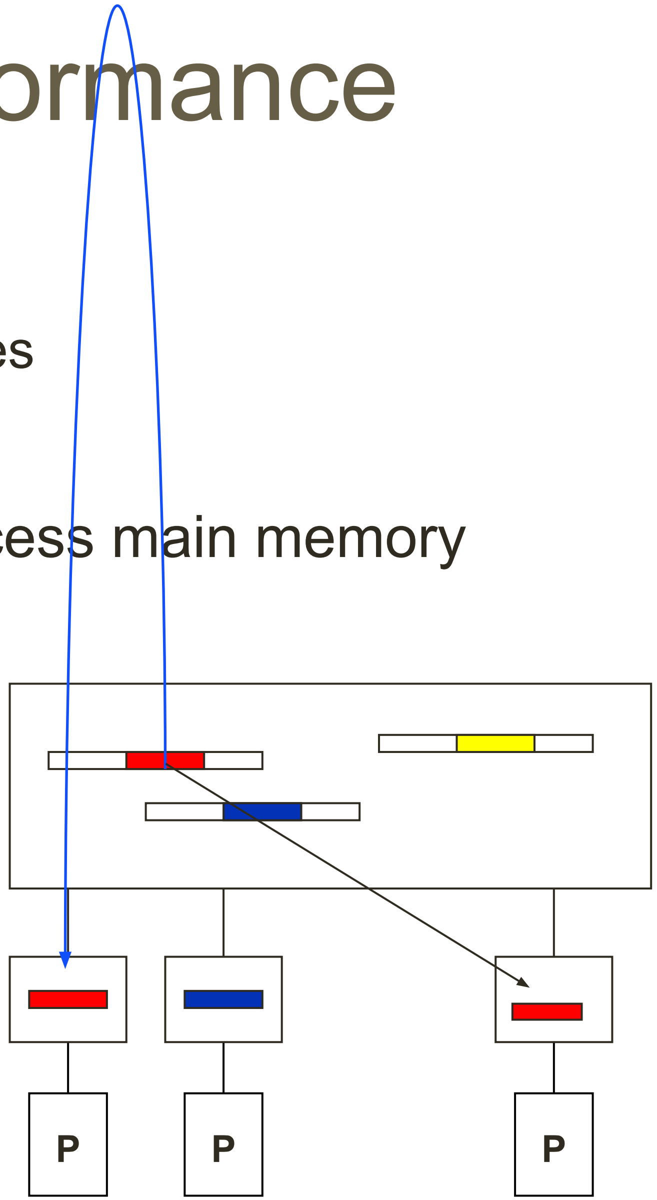
P$_1$ updates u



Before **Write Through** **Write Back**

Caches are Critical for Performance

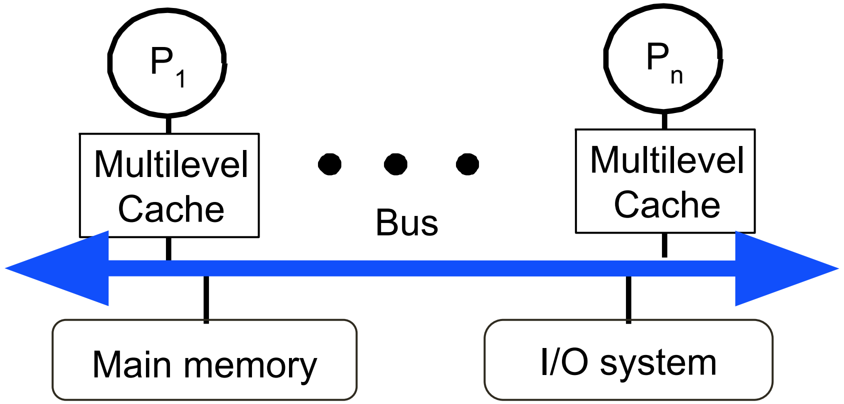
- Reduce average **latency**
  - Main memory access costs from 100 to 1000 cycles
  - Caches can reduce latency to few cycles
- Reduce average **bandwidth** and demand to access main memory
  - Reduce access to shared bus or interconnect
- Automatic **migration** of data
  - Data is moved closer to processor
- Automatic **replication** of data
  - Shared data is replicated upon need
  - Processors can share data efficiently



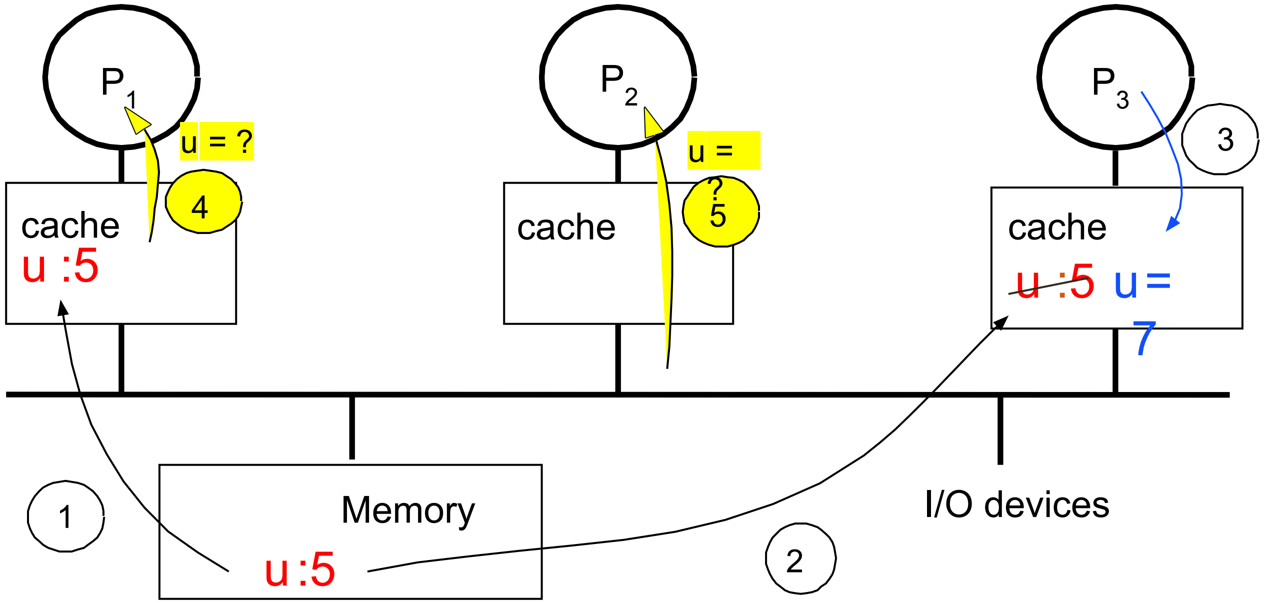
Shared Memory System

- Low-end bus-based multiprocessors dominate the server market
  - Symmetric access to main memory from any processor via loads/stores
  - Cheap and powerful extension to uniprocessors and building blocks for larger systems
  - Automatic data movement and coherent replication in caches
  - Key is extension of memory hierarchy to support multiple processors

- Attractive as throughput servers and for parallel programs that communicate through ***shared memory***



Cache Coherence Problem



- What happens when **different** processors access the **same** memory location?

- Processors see **different** values for ***u*** after event 3

- With private processor caches, multiple copies of ***u*** may exist. A write by one processor may **NOT** become visible to others leading them to keep accessing **stale** value in their caches. This is known as the **cache coherence problem.**

What to do about Cache Coherence?

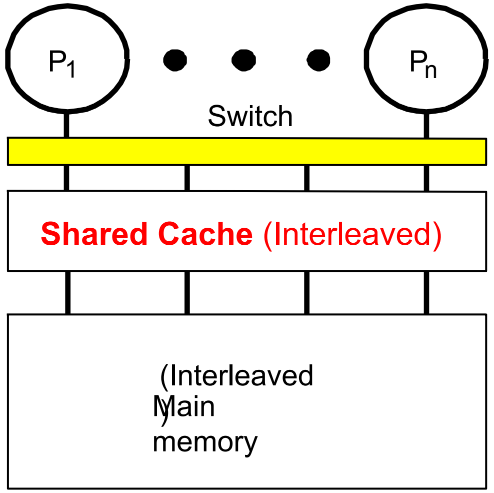
- Organize the memory hierarchy to make it go away
  - Remove private caches and use a shared cache
    - A switch is needed □ added cost and latency
    - Not practical for a large number of processors

- Mark segments of memory as **uncacheable**
  - Shared data or segments used for I/O are not cached
  - Private data is cached only
  - We lose performance

- Detect and take actions to eliminate the problem
  - Can be addressed as a basic hardware design issue
  - Techniques solve both multiprocessor as well as I/O cache coherence

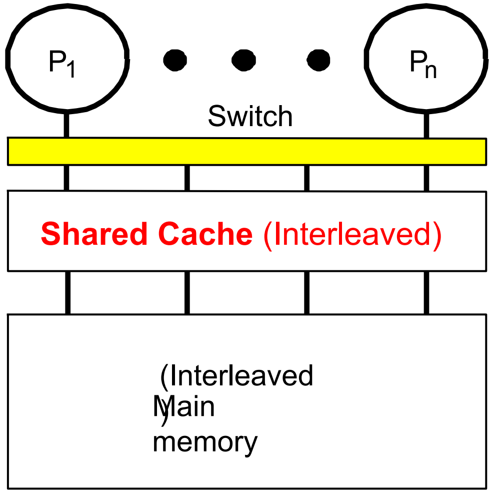
Shared Cache Design: Advantages

- Cache placement identical to single cache
  - Only one copy of any cached block
  - No coherence problem
- Fine-grain sharing
  - Communication latency is reduced when sharing cache
  - Attractive to Chip Multiprocessors (CMP), latency is few cycles
- Potential for positive interference
  - One processor prefetches data for another
- Better utilization of total storage
  - Only one copy of code/data used
- Can share data within a block
  - Long blocks without false sharing



Shared Cache Design: Disadvantages

- Fundamental bandwidth limitation
  - Can connect only a small number of processors
- Increases latency of all accesses
  - Crossbar switch
  - Hit time increases
- Potential for negative interference
  - One processor flushes data needed by another
- Share second-level (L2) cache:
  - Use private L1 caches but make the L2 cache shared
  - Many L2 caches are shared today



Intuitive Coherent Memory Model

- Caches are supposed to be transparent

- What would happen if there were no caches?
  - All reads and writes would go to main memory
  - Reading a location should return **last value written** by any processor

- What does **last value written** mean in a multiprocessor?
  - All operations on a **particular location** would be **serialized**
  - All processors would **see the same access order** to a particular location
    - If they bother to read that location

- Interleaving among memory accesses from different processors
  - Within a processor □ program order on a given memory location
  - Across processors □ only constrained by explicit synchronization

Formal Definition of Memory Coherence

- A memory system is coherent if there exists a serial order of memory operations on each memory location X, such that …

1. A read by any processor P to location X that follows a write by processor Q (or P) to X returns the **last written value** if no other writes to X occur between the two accesses
2. Writes to the same location X are **serialized**; two writes to same location X by any two processors are seen in the same order by all processors

- Two properties
  - **Write propagation:** writes become visible to other processors
  - **Write serialization:** writes are seen in the same order by all processors

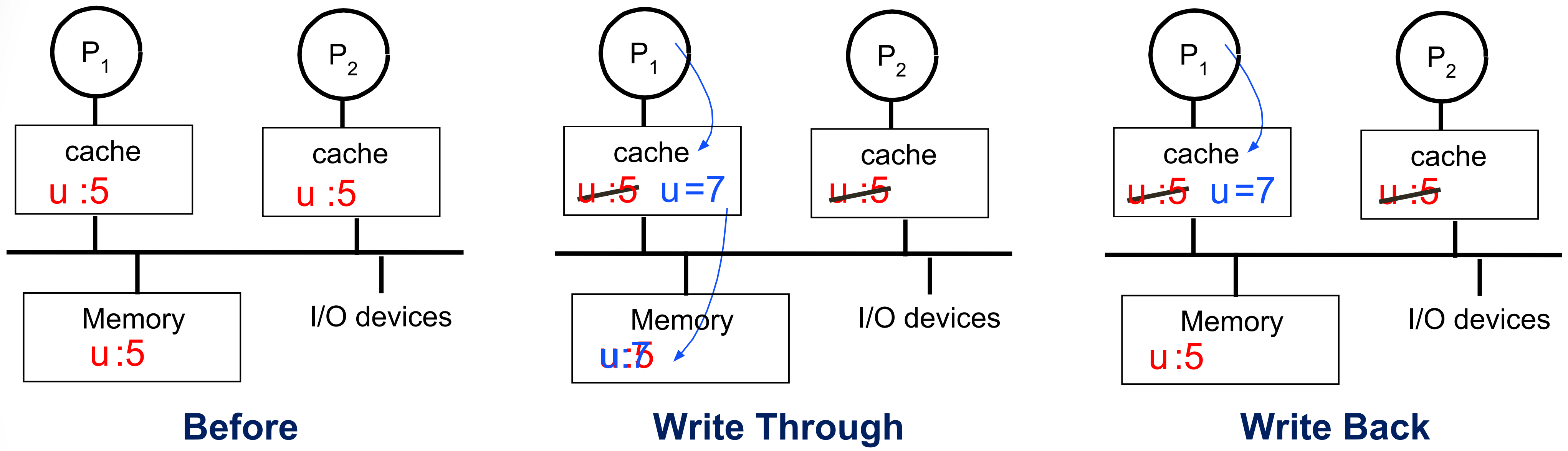
Cache Coherency

- **Cache coherent processors**
  - most current value for an address is the last write
  - all reading processors must get the most current value
- **Cache coherency problem**
  - update from a writing processor is not known to other processors
- **Cache coherency protocols**
  - mechanism for maintaining cache coherency
  - coherency state associated with a cache block of data
  - bus/interconnect operations on shared data change the state
    - for the processor that initiates an operation
    - for other processors that have the data of the operation resident in their caches

Cache Coherence Protocols

- Writing to Cache in ***n*** processor case
  - Write Invalidate – Write Through
  - Write Invalidate – Write Back
  - Write Update – Write Through
  - Write Update – Write Back

Write Invalidate



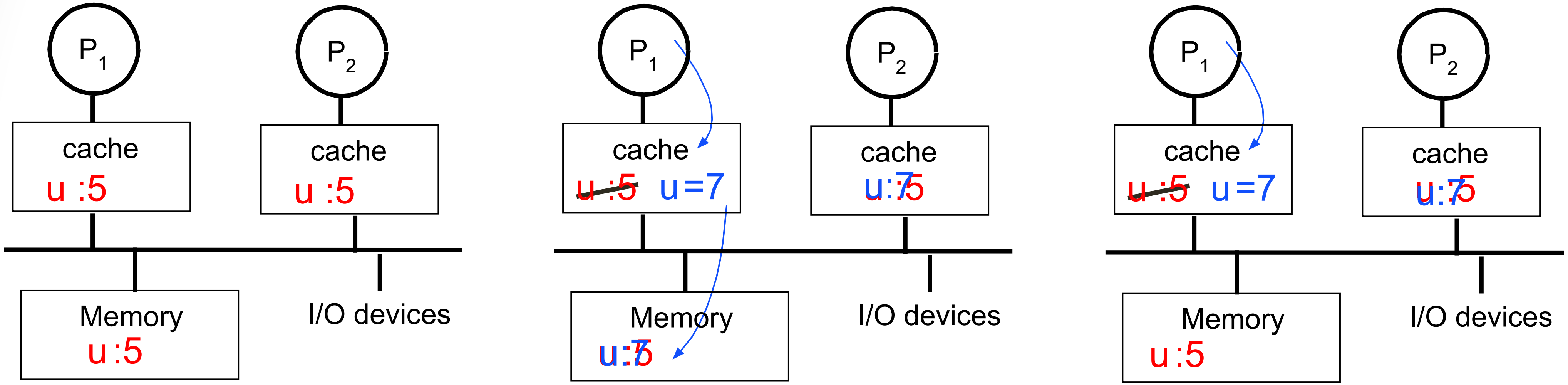
Before

Write Through

Write Back

P_1 updates u

Write Update



Before **Write Through** **Write Back**

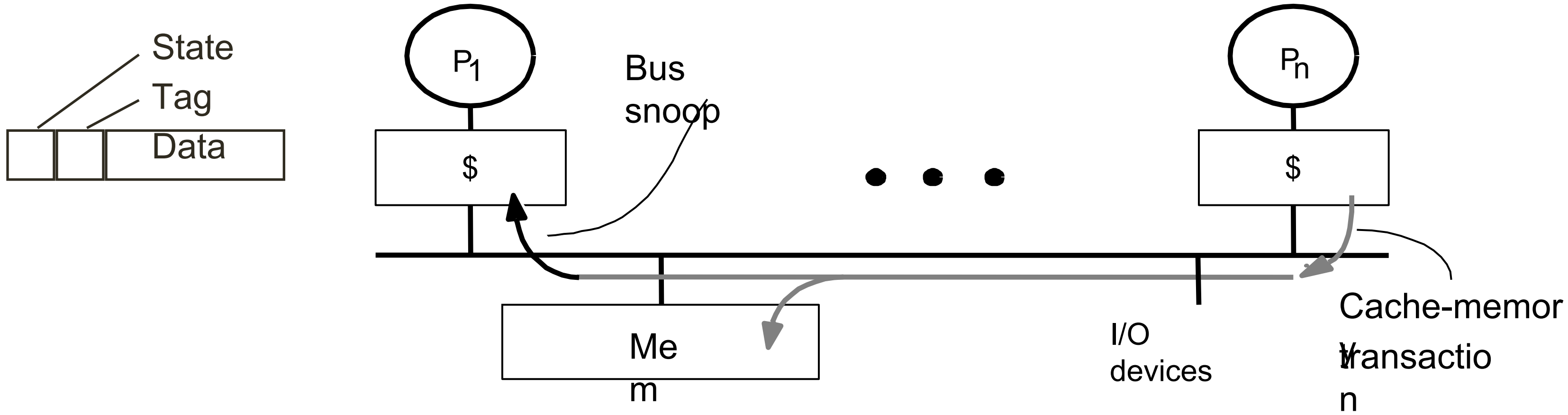
P_1 updates u

Cache Coherence Protocol Implementations

- Bus Snooping
  - Used for low-end bus-based MP’s having few processors and centralized memory
  - Send all requests for data to all processors
  - Processors snoop to see if they have a copy and respond accordingly
  - Requires broadcast, since caching information is in processors
  - Bus is used as the broadcast medium
  - Entire coherency operation is atomic with respect to other processors

- Directory-Based
  - Used for high-end MP’s having more processors, distributed memory and multi-path interconnect
  - Keep track of the sharing status of a block of memory at a single location (directory)
  - Send point-to-point requests to processors via network
  - Scales better than Snooping and avoids bottlenecks

Snoopy Cache-Coherence Protocols



- Snooping protocols are based on watching bus activities and carry out the appropriate coherency commands when necessary.

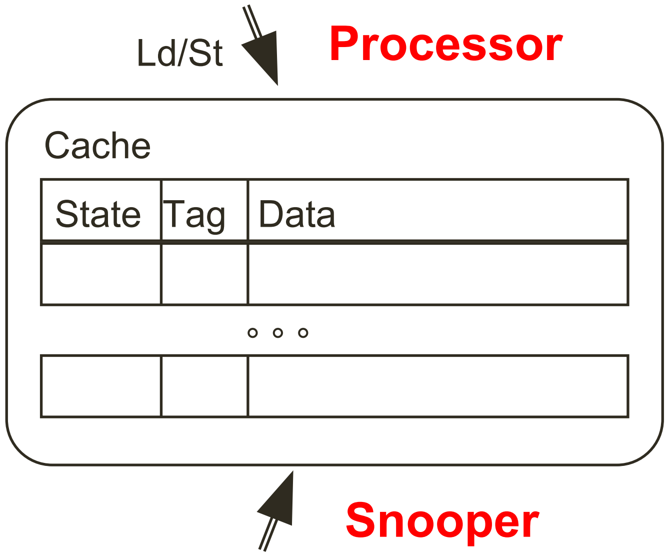
- Global memory is moved in blocks, and each block has a state associated with it, which determines what happens to the entire contents of the block.

- The state of a block might change because of the operations **Read-Miss**, **Read-Hit**, **Write-Miss**, **Write-Hit,** and **Block Replacement**.

Snooping Protocol Implementation

- **A distributed coherency protocol**
  - coherency state associated with each cache block
  - each snoop maintains coherency for its own cache

- Cache controller receives inputs from two sides:
  - Requests from processor (load/store)
  - Bus requests/responses from snooper
- Controller acts in response to both inputs
  - Updates state of blocks
  - Responds with data
  - Generates new bus transactions
- Basic Choices
  - Write-through versus Write-back
  - Invalidate versus Update



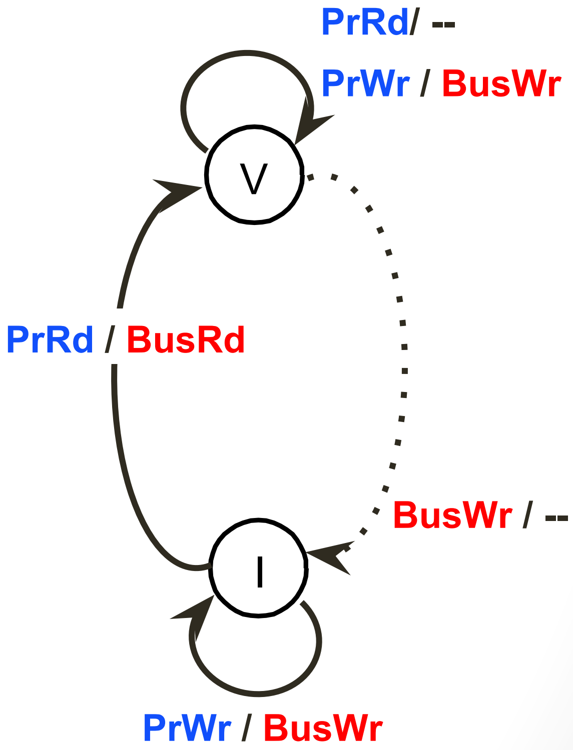
Snooping Protocol Implementation

- Cache line changes state as a function of memory access events. Events may be either:
  - Due to local processor activity (i.e., cache access)
    - **PrRd**: The processor requests to read a cache block.
    - **PrWr**: The processor requests to write a cache block.
    - **Replace**: The processor requests to replace (evict) a block to make space for new one
  - Due to bus activity - as a result of snooping
    - **BusRd**: A read request from another processor.
    - **BusRdX** (Bus Read Exclusive): An exclusive read request from another processor with the intent to write to it.
    - **BusWr**: A write request from another processor.
    - **BusWB**: A bus transaction to write a dirty block back to memory (usually due to eviction).
    - **Flush**: A request to write back an entire cache block to main memory
- Cache line has its own state affected only if address matches

Write-through Invalidate Protocol

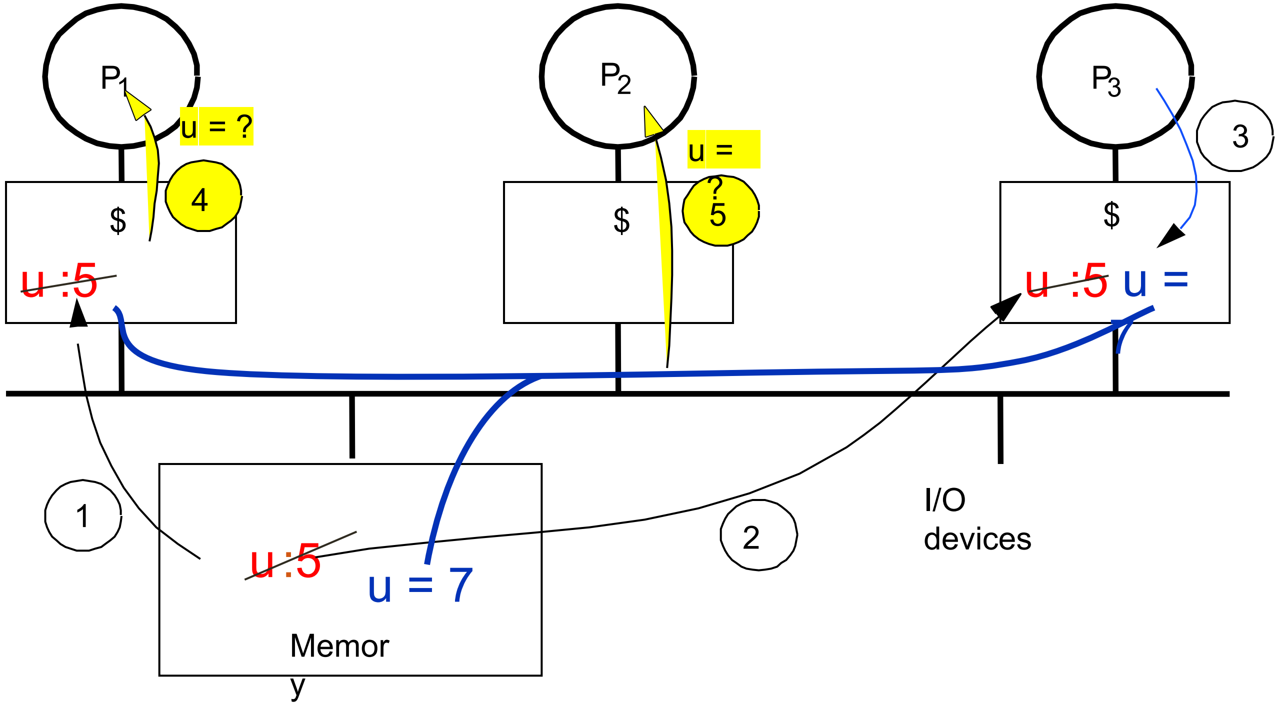
- Two states per block in each cache
  - **Valid (V)** – The copy is consistent with global memory
  - **Invalid (I)** – The copy is inconsistent

Event	Actions
Read Hit	Use the local copy from the cache.
Read Miss	Fetch a copy from global memory. Set the state of this copy to Valid.
Write Hit	Perform the write locally. Broadcast an Invalid command to all caches. Update the global memory.
Write Miss	Using write no-allocate policy, broadcast an invalid command to all caches. Update the global memory. Processor cache remains in invalid state.
Block Replacement	Since memory is always consistent, no write back is needed when a block is replaced.



A / **B** means if A is observed, B is generated.

Example of Write-through Invalidate Protocol



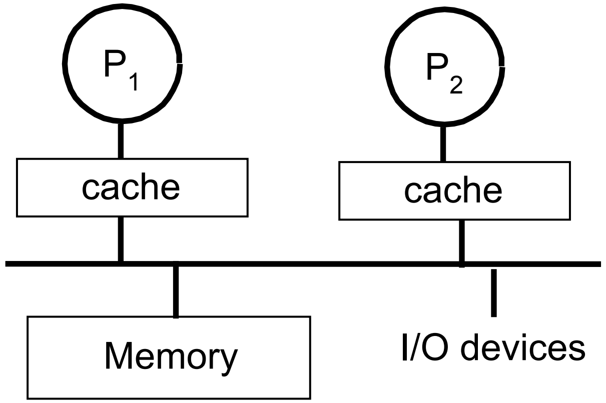
- At step 4, an attempt to read ***u*** by P_1 will result in a cache miss
  - Correct value of *u* is fetched from memory

- Similarly, correct value of *u* is fetched at step 5 by P_2

Example of Write-through Invalidate Protocol

X = 5

1. P_1 reads X
2. P_2 reads X
3. P_2 updates X, X=10
4. P_2 reads X
5. P_2 updates X, X=15
6. P_1 updates X, X=20
7. P_2 reads X

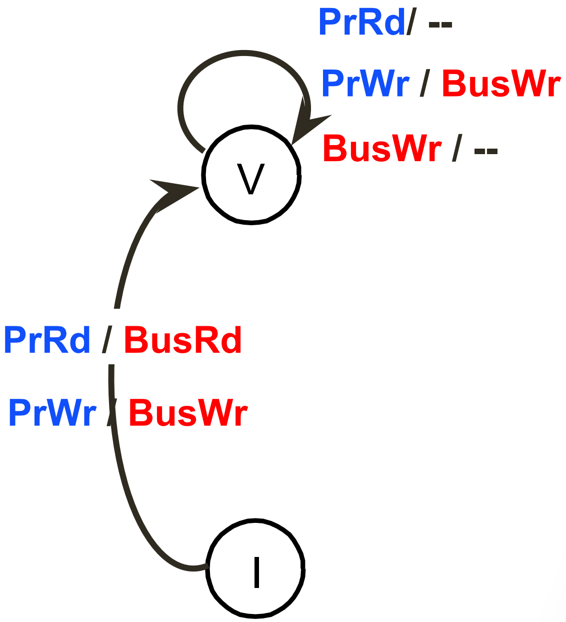


		Memory	Cache P_1		Cache P_2	
	Event	**X**	**X**	**State**	**X**	**State**
0	Original Value	**5**				
1	P_1 reads X	**5**	**5**	**VALID**		
2	P_2 reads X	**5**	**5**	**VALID**	**5**	**VALID**
3	P_2 updates X, X=10	**10**	**5**	**INVALID**	**10**	**VALID**
4	P_2 reads X	**10**	**5**	**INVALID**	**10**	**VALID**
5	P_2 updates X, X=15	**15**	**5**	**INVALID**	**15**	**VALID**
6	P_1 updates X, X=20	**20**	**5**	**INVALID**	**15**	**INVALID**
7	P_2 reads X	**20**	**5**	**INVALID**	**20**	**VALID**

Write-through Update Protocol

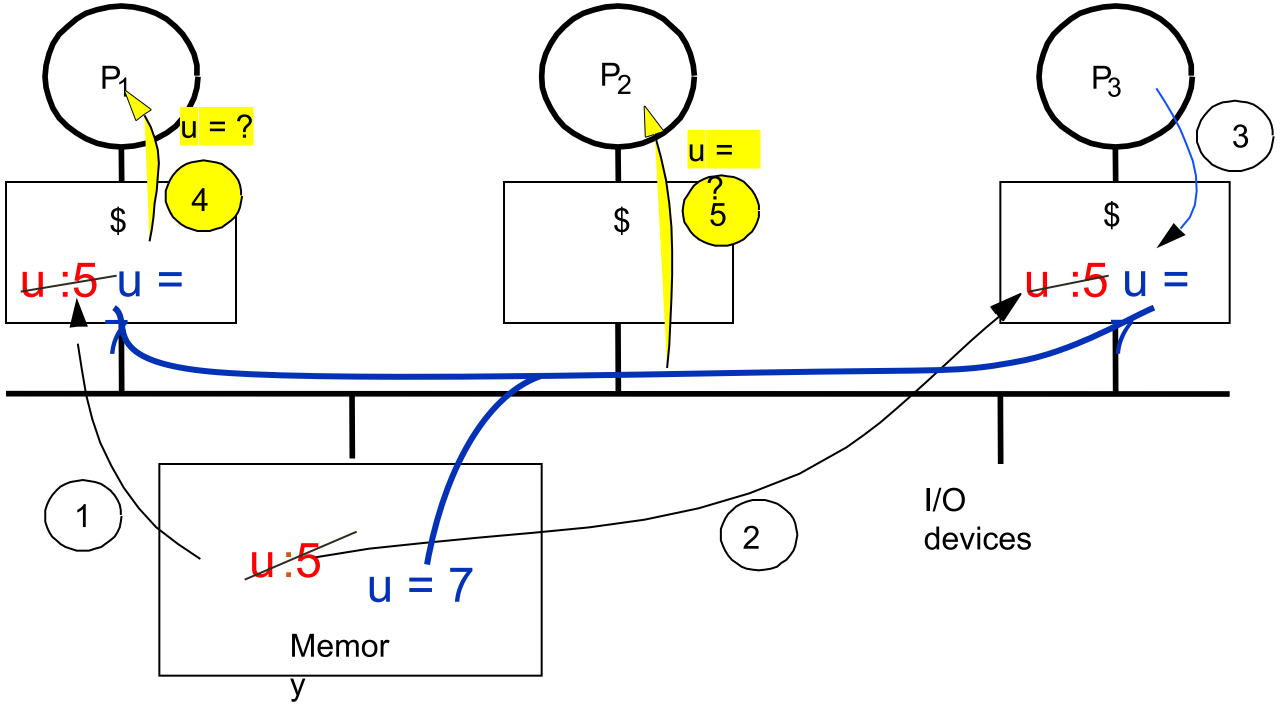
- Two states per block in each cache
  - **Valid (V)** – The copy is consistent with global memory
  - **Invalid (I)** – The copy is inconsistent

Event	Actions
Read Hit	Use the local copy from the cache.
Read Miss	Fetch a copy from global memory. Set the state of this copy to Valid.
Write Hit	Perform the write locally. Broadcast an update command to all caches which update their values. Update the global memory.
Write Miss	Using write allocate policy, fetch a copy from global memory, broadcast an update command to all caches. Update the global memory. Set the state of this copy to vaild.
Replace	Since memory is always consistent, no write back is needed when a block is replaced.



A / **B** means if A is observed, B is generated.

Example of Write-through Update Protocol

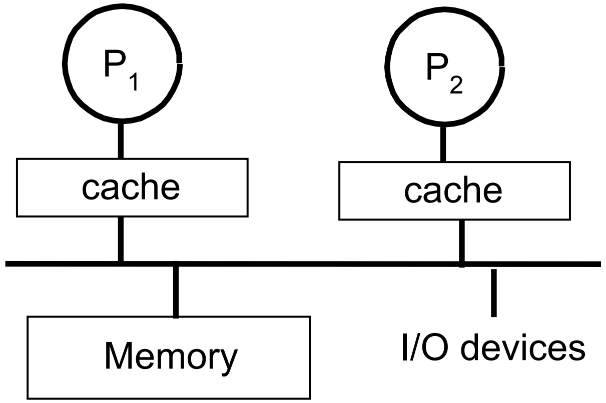


- At step 4, an attempt to read ***u*** by P_1 will result in a cache hit
- Correct value of *u* is fetched at step 5 by P_2

Example of Write-through Update Protocol

X = 5

1. P_1 reads X
2. P_2 reads X
3. P_2 updates X, X=10
4. P_2 reads X
5. P_2 updates X, X=15
6. P_1 updates X, X=20
7. P_2 reads X



		Memory	Cache P_1		Cache P_2	
	Event	**X**	**X**	**State**	**X**	**State**
0	Original Value	**5**				
1	P_1 reads X	**5**	**5**	**VALID**		
2	P_2 reads X	**5**	**5**	**VALID**	**5**	**VALID**
3	P_2 updates X, X=10	**10**	**10**	**VALID**	**10**	**VALID**
4	P_2 reads X	**10**	**10**	**VALID**	**10**	**VALID**
5	P_2 updates X, X=15	**15**	**15**	**VALID**	**15**	**VALID**
6	P_1 updates X, X=20	**20**	**20**	**VALID**	**20**	**VALID**
7	P_2 reads X	**20**	**20**	**VALID**	**20**	**VALID**

2-State Protocol is Coherent

- Assume bus transactions and memory operations are atomic
  - All phases of one bus transaction complete before next one starts
  - Processor waits for memory operation to complete before issuing next
- Assume one-level cache
  - Invalidations applied during bus transaction
- All writes go to bus + atomicity
  - **Writes serialized** by order in which they appear on bus □ **bus order**
  - Invalidations/updates are performed by all cache controllers in bus order
- Read misses are serialized on the bus along with writes
  - Read misses are guaranteed to return the last written value
- Read hits do not go on the bus, however …
  - Read hit returns last written value by processor or by its last read miss

MSI Write-back Invalidate Protocol

- A valid block can be owned by memory and shared in multiple caches that can contain only the shared copies of the block. Multiple processors can safely read these blocks from their caches until one processor updates its copy. At this time, the writer becomes the only owner of the valid block, and all other copies are invalidated.

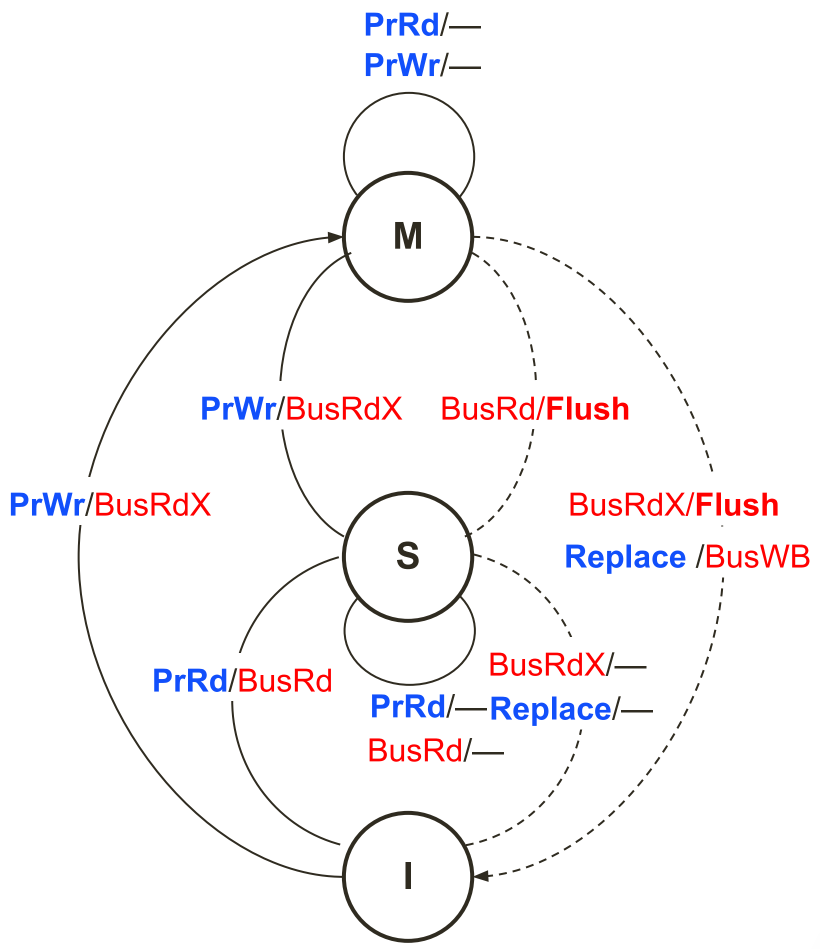
- Three states per block in each cache
  - **Modified (M)**: only this cache has a modified valid copy of this block
  - **Shared (S)**: block is clean and may be cached in more than one cache, memory is up-to-date
  - **Invalid (I)**: block is invalid

MSI Write-back Invalidate Protocol

Event	Actions
Read Hit	Use the local copy from the cache.
Read Miss	If no Exclusive (Read-Write) copy exists, then supply a copy from global memory. Set the state of this copy to Shared (Read-Only). If an Exclusive (Read-Write) copy exists, make a copy from the cache that set the state to Exclusive (Read-Write), update global memory and local cache with the copy. Set the state to Shared (Read-Only) in both caches.
Write Hit	If the copy is Exclusive (Read-Write), perform the write locally. If the state is Shared (Read-Only), then broadcast an Invalid to all caches. Set the state to Exclusive (Read-Write).
Write Miss	Get a copy from either a cache with an Exclusive (Read-Write) copy, or from global memory itself. Broadcast an Invalid command to all caches. Update the local copy and set its state to Exclusive (Read-Write).
Block Replacement	If a copy is in an Exclusive (Read-Write) state, it has to be written back to main memory if the block is being replaced. If the copy is in Invalid or Shared (Read-Only) states, no write back is needed when a block is replaced.

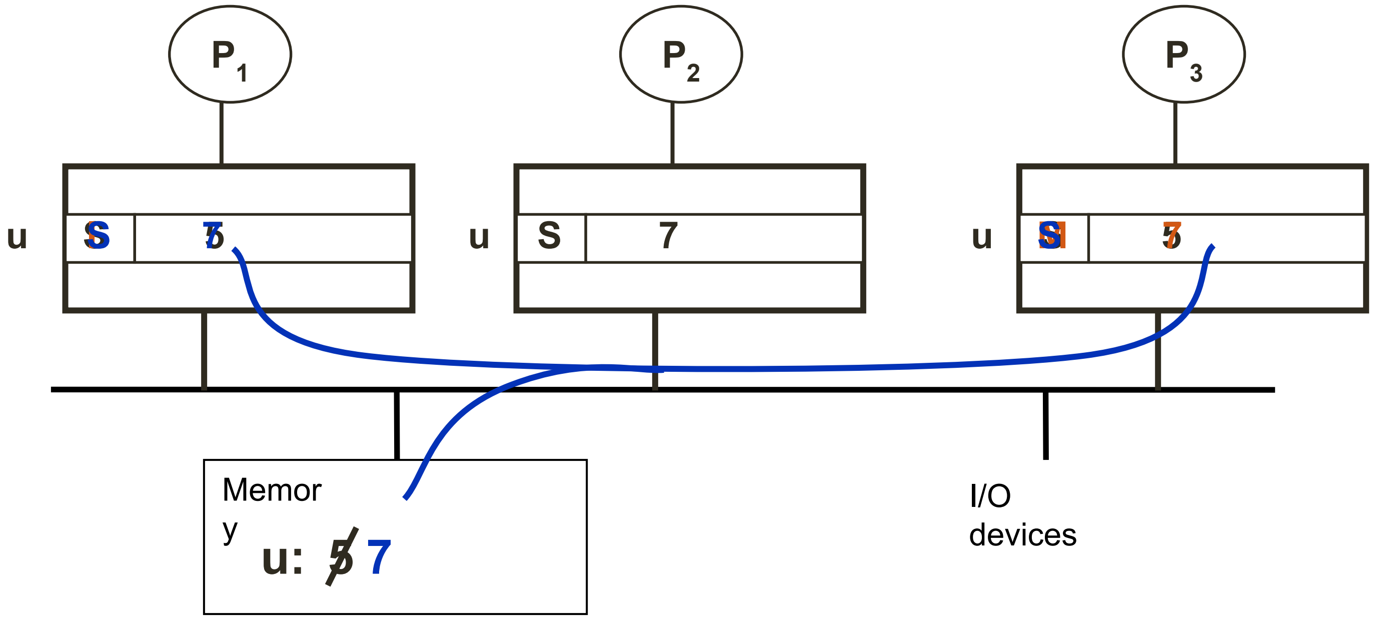
MSI Write-back Invalidate Protocol

- Processor Read
  - Cache miss □ causes a Bus Read
  - Cache hit (S or M) □ no bus activity
- Processor Write
  - No bus activity when Modified block
  - Generates a BusRdX when not Modified
    - BusRdX causes other caches to invalidate
- Observing a Bus Read
  - If Modified, flush block on bus
    - Picked by memory and requesting cache
    - Block is now shared
- Observing a Bus Read Exclusive
  - Invalidate block
  - Flush data on bus if block is modified



A / **B** means if A is observed, B is generated.

Example MSI Write-back Invalidate Protocol

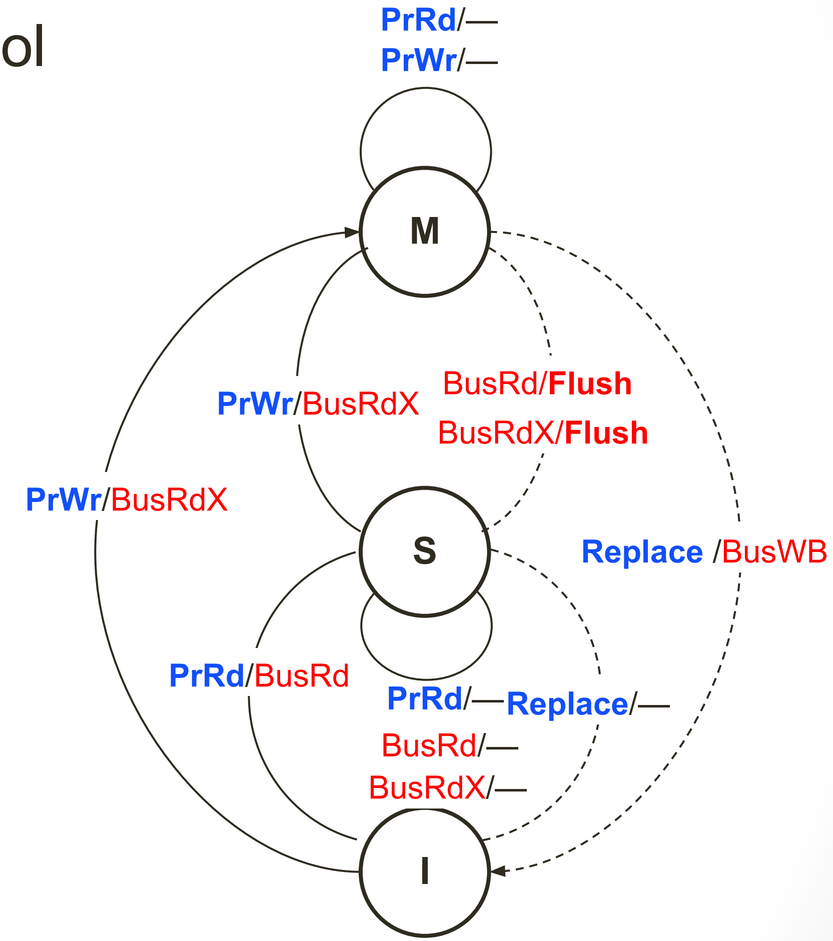


Processor Action	State P1	State P2	State P3	Bus Action	Data from
1. P1 reads u	S			BusRd	Memory
2. P3 reads u	S		S	BusRd	Memory
3. P3 writes u	I		M	BusRdX	Memory
4. P1 reads u	S		S	BusRd, Flush	P3 cache
5. P2 reads u	S	S	S	BusRd	Memory

MSI Write-back Update Protocol

- Mostly similar to MSI Write-back invalidate Protocol

- Actions taken on a BusRdX differs
  - Other’s copies are updated instead of invalidating



A / B means if A is observed, B is generated.

Satisfying Coherence

- Write propagation
  - A write to a shared or invalid block is made visible to all other caches
    - Using the Bus Read-exclusive (BusRdX) transaction
    - Invalidations that the Bus Read-exclusive generates
    - Other processors experience a cache miss before observing the value written
- Write serialization
  - All writes that appear on the bus (BusRdX) are serialized by the bus
    - Ordered in the same way for all processors including the writer
    - Write performed in writer's cache before it handles other transactions
  - However, not all writes appear on the bus
  - Write sequence to modified block must come from same processor, say *P*
  - Serialized within *P*: Reads by *P* will see the write sequence in the serial order
  - Serialized to other processors
    - Read miss by another processor causes a bus transaction
    - Ensures that writes appear to other processors in the same serial order

Update or Invalidate

- Update looks the simplest, most obvious and fastest, but:
  - Multiple writes to same word (no intervening read) need only one invalidate message but would require an update for each
  - Writes to same block in (usual) multi-word cache block require only one invalidate but would require multiple updates.
- Due to both spatial and temporal locality, previous cases occur often.
- Bus bandwidth is a precious commodity in shared memory multi-processors
- Experience has shown that invalidate protocols use significantly less bandwidth.
- Will consider implementation details only of invalidate.

Implementation Issues

- In both schemes, knowing if a cached value is not shared (copy in another cache) can avoid sending any messages.

- Invalidate description assumed that a cache value update was written through to memory. If we used a ‘copy back’ scheme other processors could re-fetch old value on a cache miss.

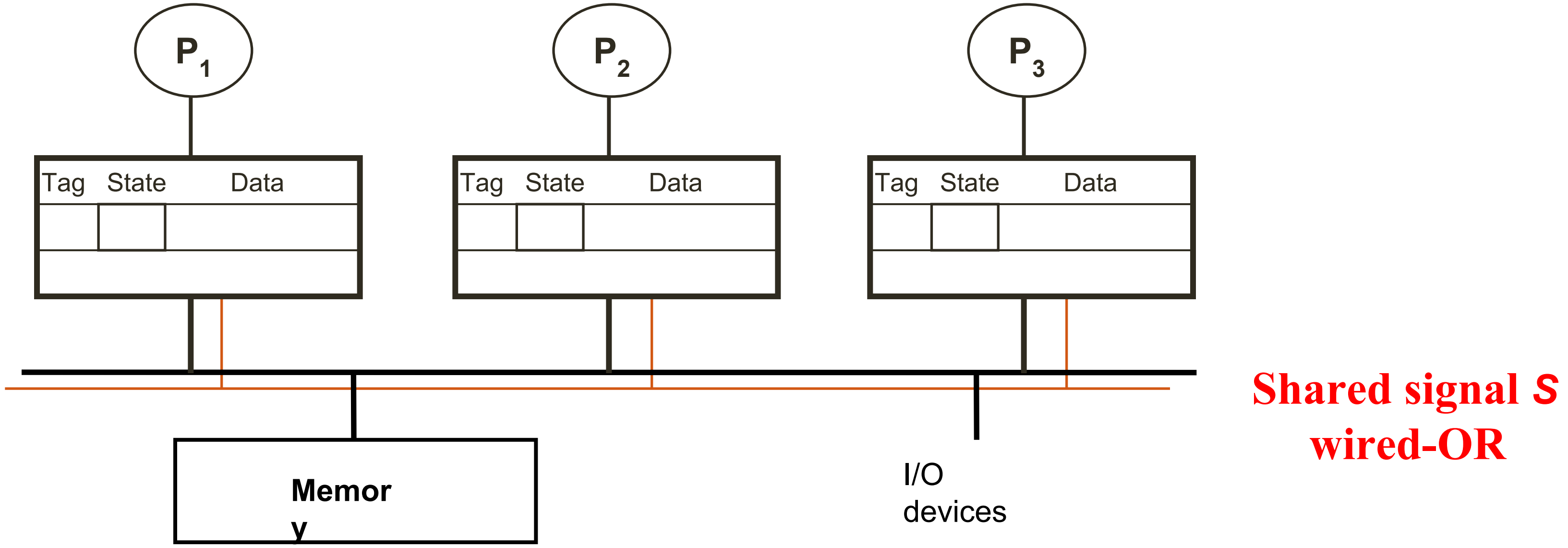
- We need a protocol to handle all this.

- Drawback of the MSI Protocol
  - Read/Write of a block causes 2 bus transactions
    - Read BusRd (I→S) followed by a write BusRdX (S→M)
    - This is the case even when a block is private to a process and not shared
    - Most common when using a multiprogrammed workload

MESI Protocol

- A practical multiprocessor invalidate protocol which attempts to minimize bus usage.
- Allows usage of a ‘write back’ scheme - i.e. main memory not updated until ‘dirty’ cache line is displaced
- Extension of usual cache tags, i.e. invalid tag and ‘dirty’ tag in normal write back cache.
- Any cache line can be in one of the 4 states (2 bits)
  - **Modified (M):** Only this cache has a copy and is modified (dirty). Main memory copy is stale.
  - **Exclusive (E):** Only this cache has a copy which is not modified. Main memory is up-to-date
  - **Shared (S):** Multiple cache may have unmodified copies. Main memory is up-to-date
  - **Invalid (I):** Cache line data is not valid

Hardware Support for MESI

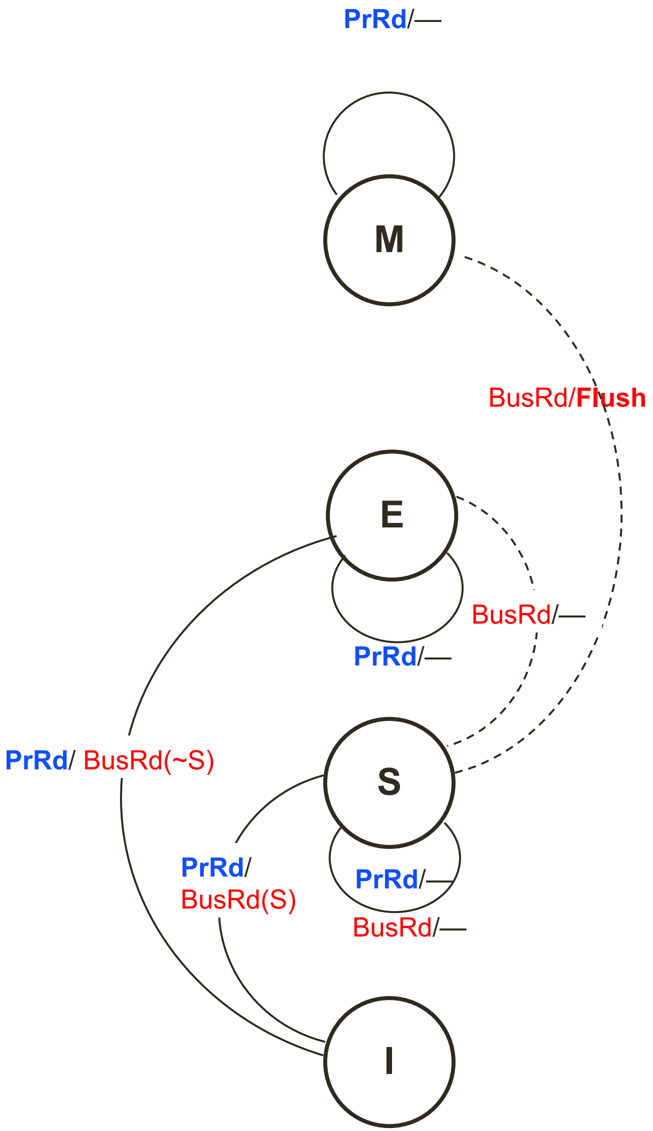


- New requirement on the bus interconnect
  - Additional signal, called the shared signal *S*, must be available to all controllers
  - Implemented as a wired-OR line

- All cache controllers snoop on BusRd
  - Assert shared signal if block is present (state *S*, *E*, or *M*)
  - Requesting cache chooses between *E* and *S* states depending on shared signal

MESI State Transition Diagram

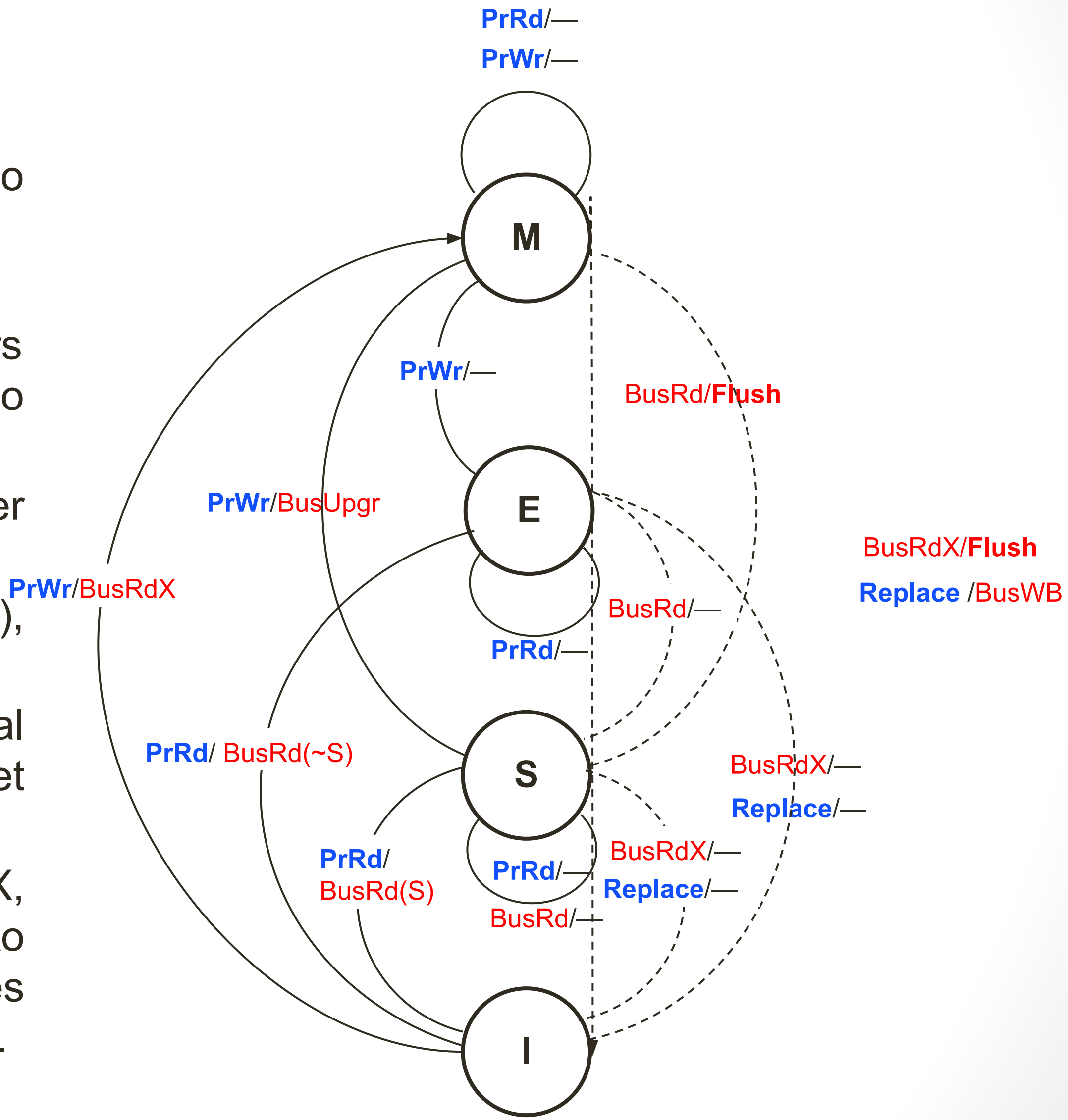
- Read Hit
  - Must be from one of the M, E, S states
  - No state change (if M, it must have been modified locally)
- Read Miss
  - Either no copy, or one cache has E copy, or several S copies, or one cache has M copy
  - If no copy exist, read from memory (I □ E)
  - If E copy exist, snooping cache puts value on bus and both lines set to S
  - If S copies exist, one snooping cache puts value on bus and local copy set to S (others remain S).
  - If M copy exist, snooping cache puts value on bus and local copy set to S. Source (M) value copied back to memory (M □ S)



A / **B** means if A is observed, B is generated.

MESI State Transition Diagram

- Write Hit
  - **M:** Line is exclusive and already ‘dirty’, update with no state change
  - **E:** Update and change state from E to M
  - **S:** Broadcast Invalidate on bus, snooping processors with S copy change to I. Local state change from S to M.
- Write Miss (Detailed action depends on copies in other processors)
  - If no copy exist, read from memory to local cache (?), update value and change state to M.
  - If E or S copies exist, read from memory to local cache with BusRdX. Snooping processors’ copies set to I, local copy set to M.
  - If another M copy exist, processor issues BusRdX, snooping processor blocks BusRdX, write its copy to memory and is set to I. Original processor reissues BusRdX, value read from memory and updated (□M).



A / **B** means if A is observed, B is generated.

MESI: An Example

- Consider a system with 3 processors where all processors are referencing the same cache block.
- Using the MESI protocol fill the table to describe the cache block's state in each processor's cache.

Request	P_1	P_2	P_3	Data Supplier
Initially	–	–	–	–
P_1 Read	E	–	–	Main Memory
P_1 Write	M	–	–	–
P_3 Read	S	–	S	P_1 Cache
P_3 Write	I	–	M	–
P_1 Read	S	–	S	P_3 Cache
P_3 Read	S	–	S	–
P_2 Read	S	S	S	P_1 or P_3 Cache

Directory Schemes

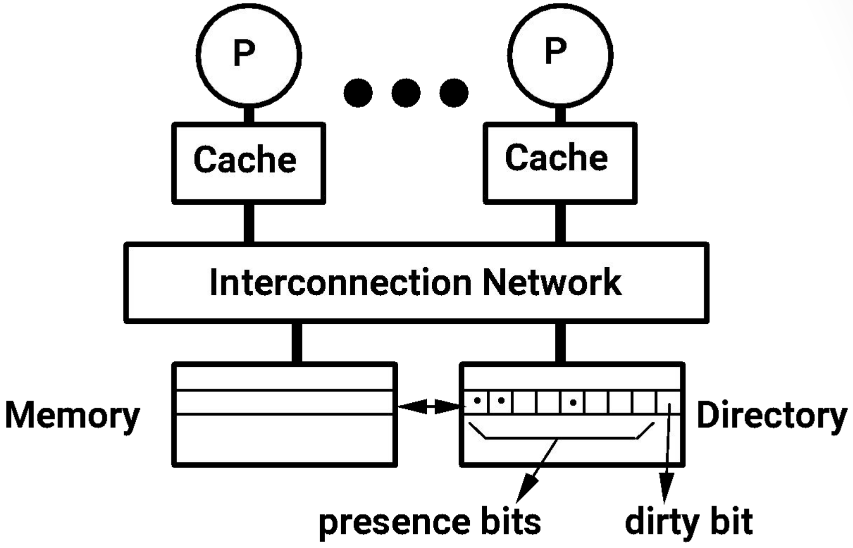
- Snoopy schemes do not scale because they rely on broadcast

- Directory-based schemes allow scaling.
    - avoid broadcasts by keeping track of all PEs caching a memory block, and then using point-to-point messages to maintain coherence
    - they allow the flexibility to use any scalable point-to-point network

Directory Schemes

- Read from main memory by PE-i:
  - If dirty-bit is OFF then { read from main memory; turn p[i] ON; }
  - if dirty-bit is ON then { recall line from dirty PE (cache state to shared); update memory; turn dirty-bit OFF; turn p[i] ON; supply recalled data to PE-i; }

- Write to main memory:
  - If dirty-bit OFF then { send invalidations to all PEs caching that block; turn dirty-bit ON; turn P[i] ON; ... }
  - …



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

Key Issues

- Scaling of memory and directory bandwidth
  - Can not have main memory or directory memory centralized
  - Need a distributed memory and directory structure
- Directory memory requirements do not scale well
  - Number of presence bits grows with number of PEs
  - Many ways to get around this problem
    - limited pointer schemes of many flavors
- Industry standard
  - SCI: Scalable Coherent Interface