

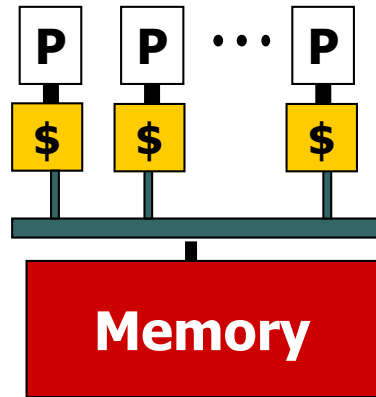
COM515 Advanced Computer Architecture

Lecture 7. Multiprocessor and Memory Coherence

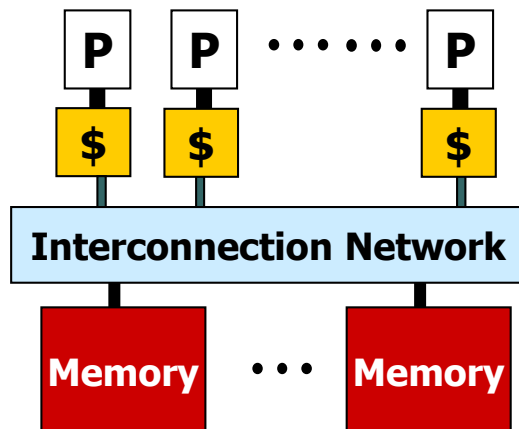
Prof. Taeweon Suh
Computer Science Education
Korea University

Memory Hierarchy in a Multiprocessor

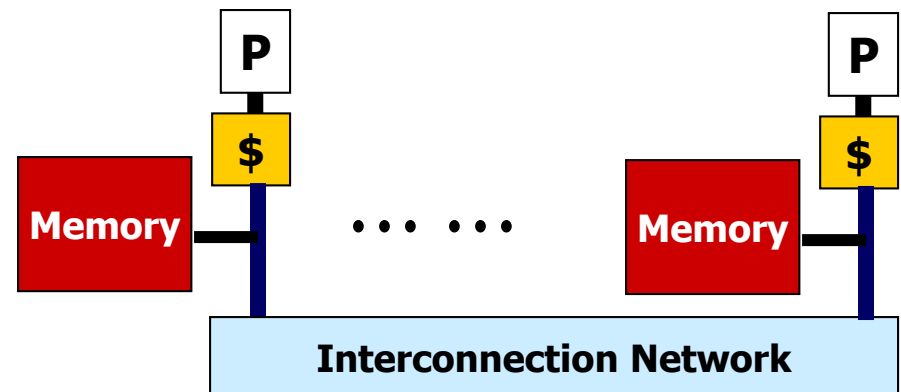
Bus-based shared memory



**Fully-connected shared memory
(Dancehall)**



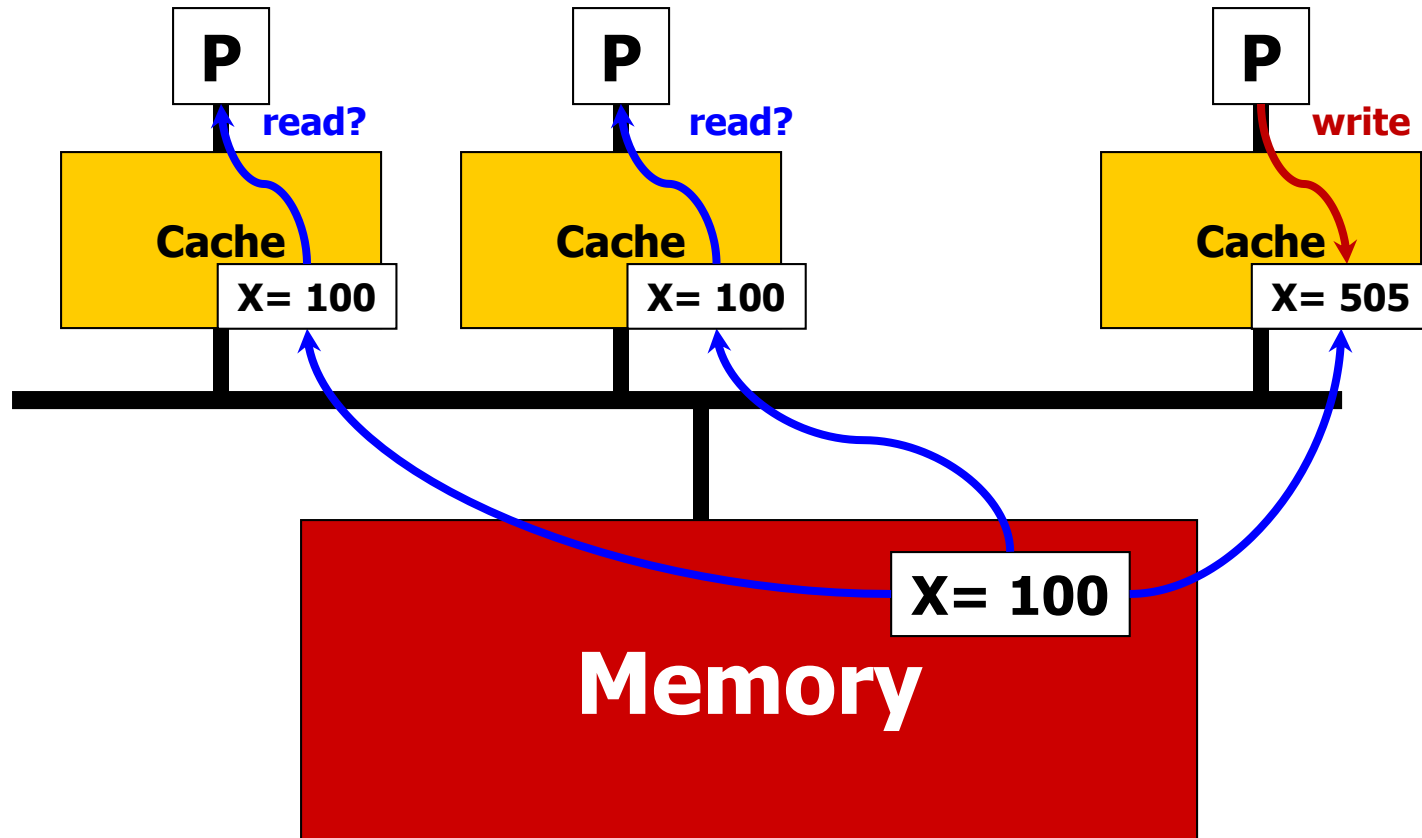
Distributed shared memory



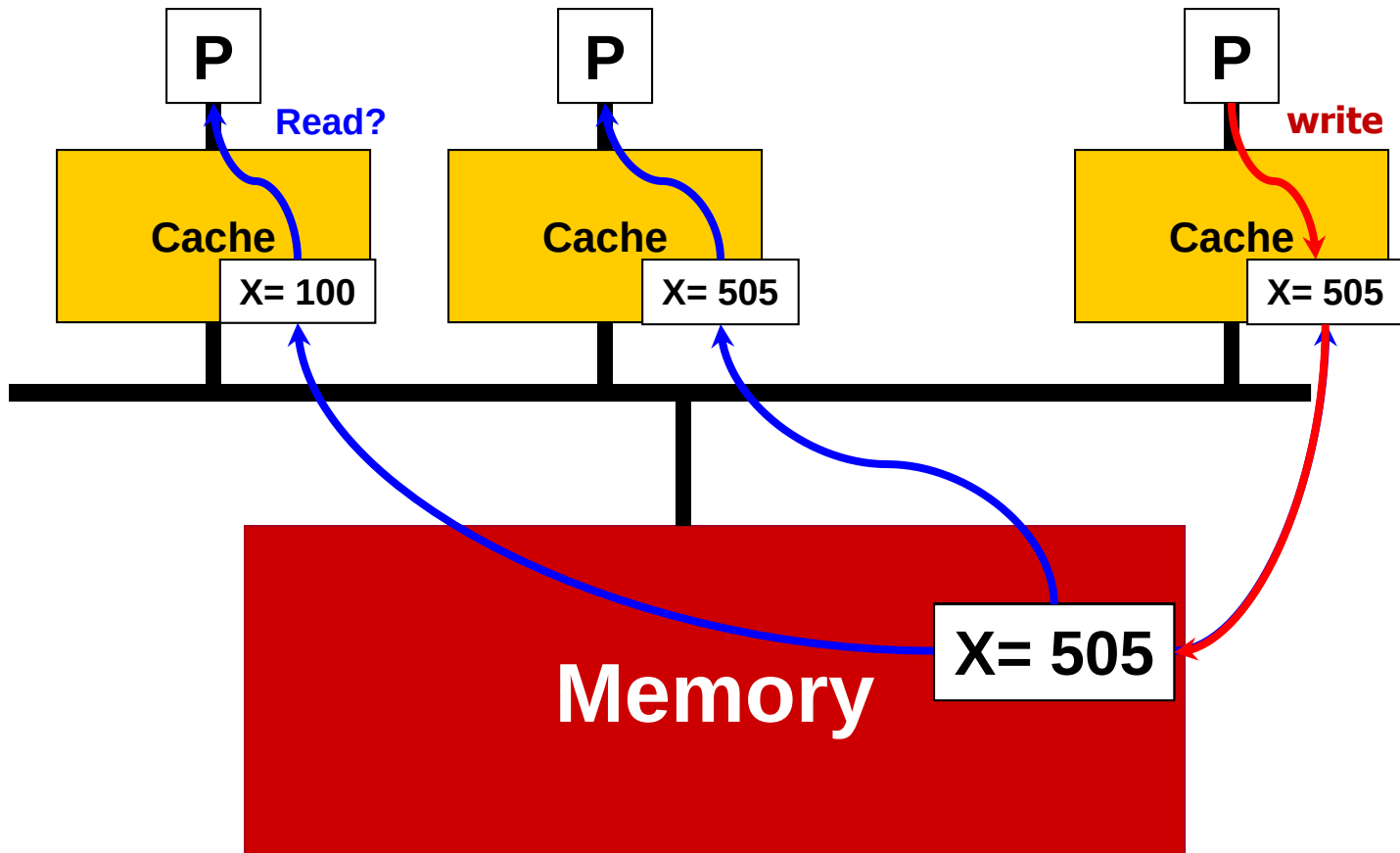
Why Cache Coherency?

- Closest cache level is private
- Multiple copies of cache line can be present across different processor nodes
- Local updates (writes) leads to incoherent state
 - Problem exhibits in both write-through and writeback caches

Writeback Cache w/o Coherence



Writethrough Cache w/o Coherence

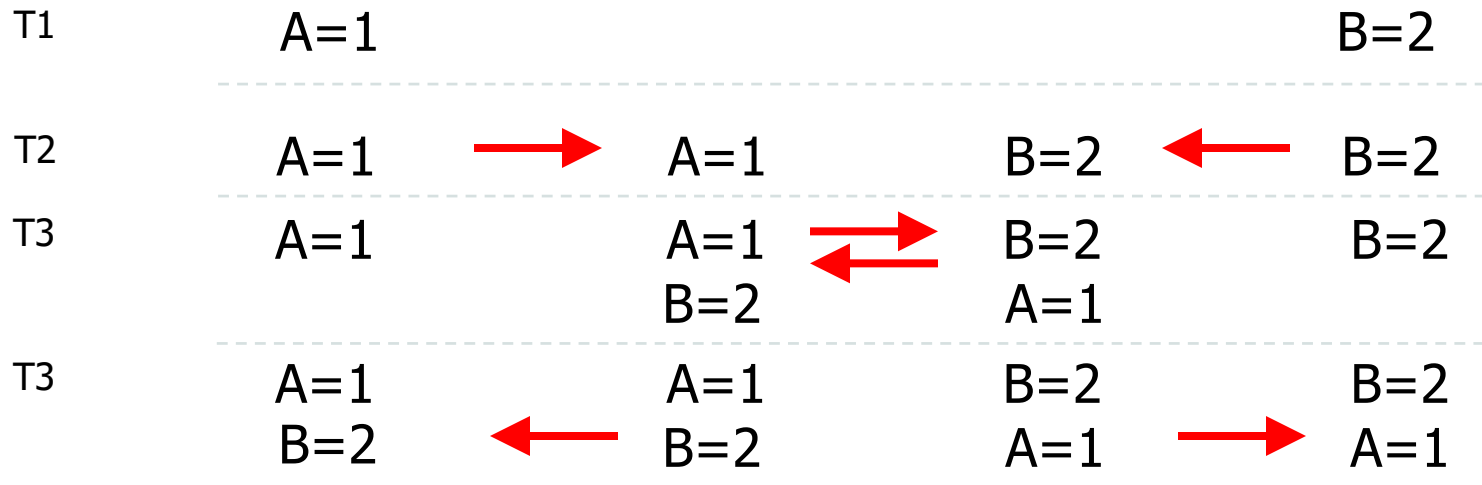
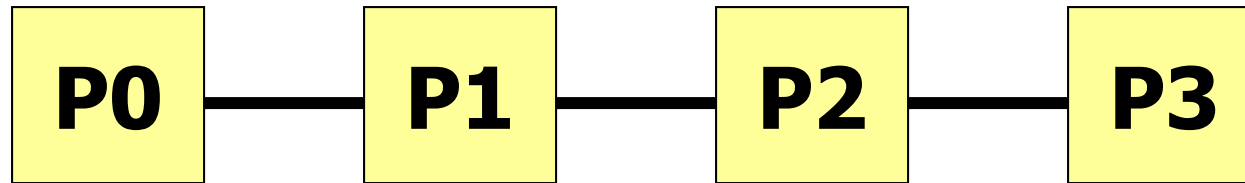


Definition of Coherence

- A multiprocessor memory system is coherent if the results of any execution of a program can be reconstructed by a hypothetical serial order
- Implicit definition of coherence
 - **Write propagation**
 - Writes are visible to other processes
 - **Write serialization**
 - All writes to the same location are seen in the same order by all processes
 - For example, if read operations by P_1 to a location see the value produced by write w_1 (say, from P_2) before the value produced by write w_2 (say, from P_3), then reads by another process P_4 (or P_2 or P_3) also should not be able to see w_2 before w_1

Sounds Easy?

A=0 B=0



See A's update before B's

See B's update before A's

Cache Coherence Protocols According to Caching Policies

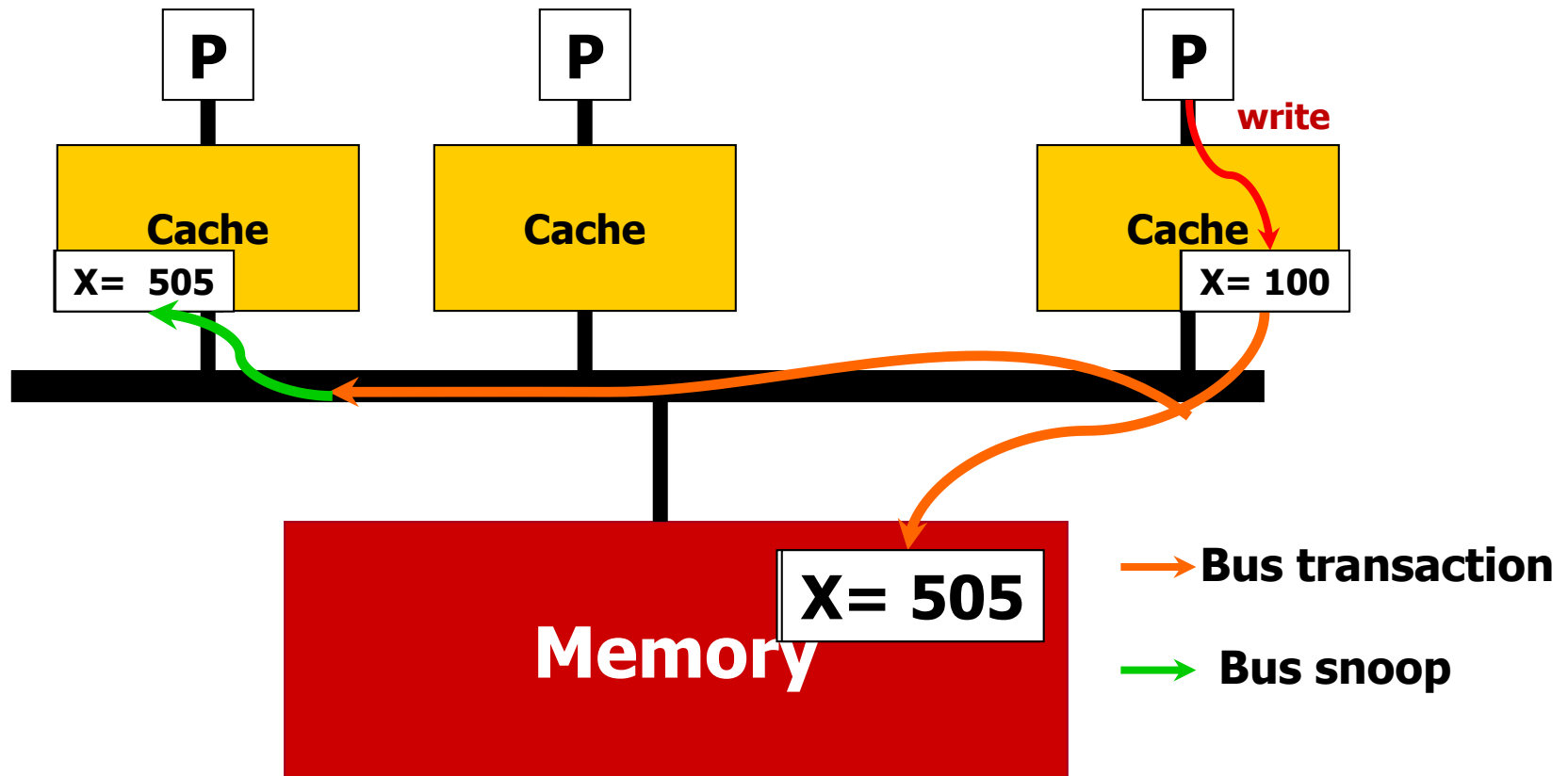
- Write-through cache
 - Update-based protocol
 - Invalidation-based protocol
- Writeback cache
 - Update-based protocol
 - Invalidation-based protocol

Bus Snooping based on Write-Through Cache

- All the writes will be shown as a transaction on the shared bus to memory
- Two protocols
 - Update-based Protocol
 - Invalidation-based Protocol

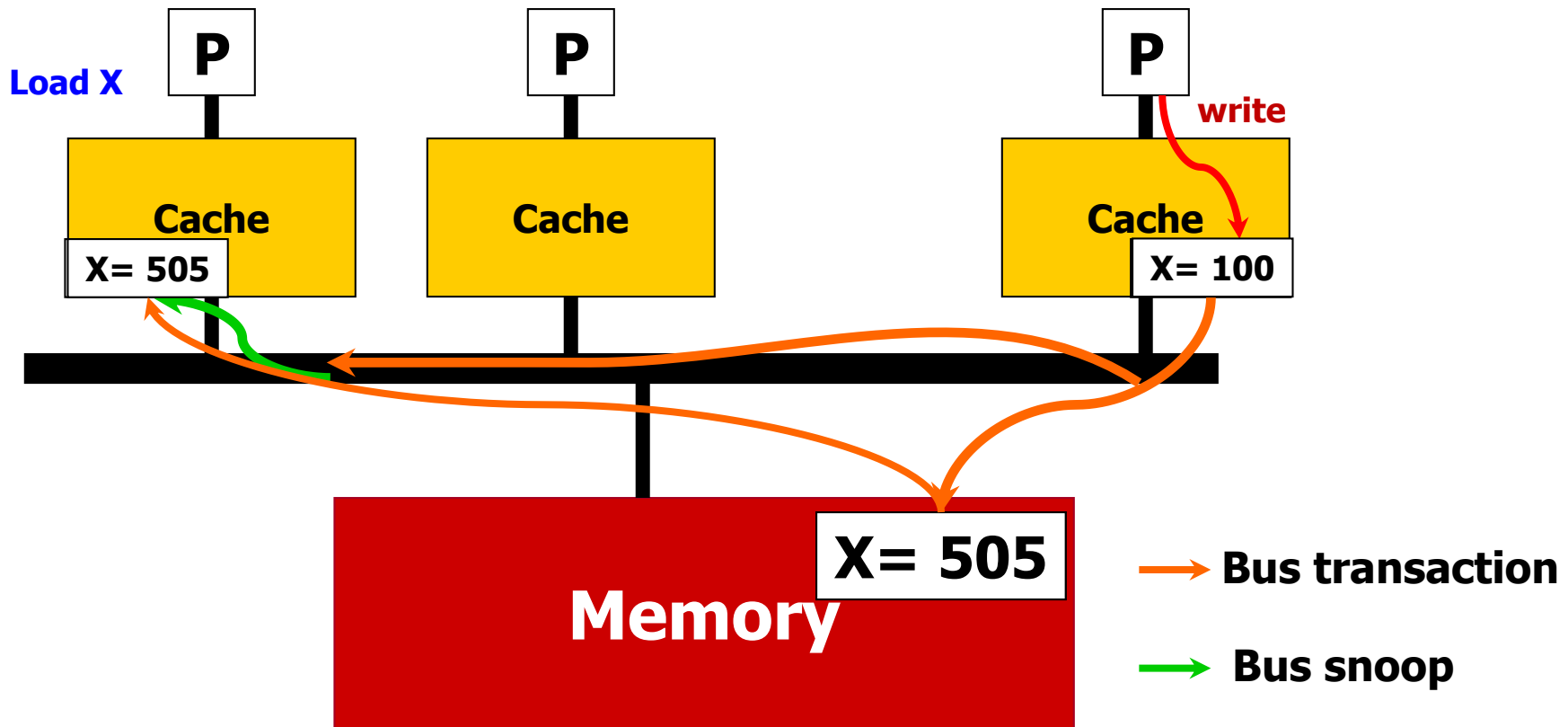
Bus Snooping

- **Update-based** Protocol on **Write-Through** cache

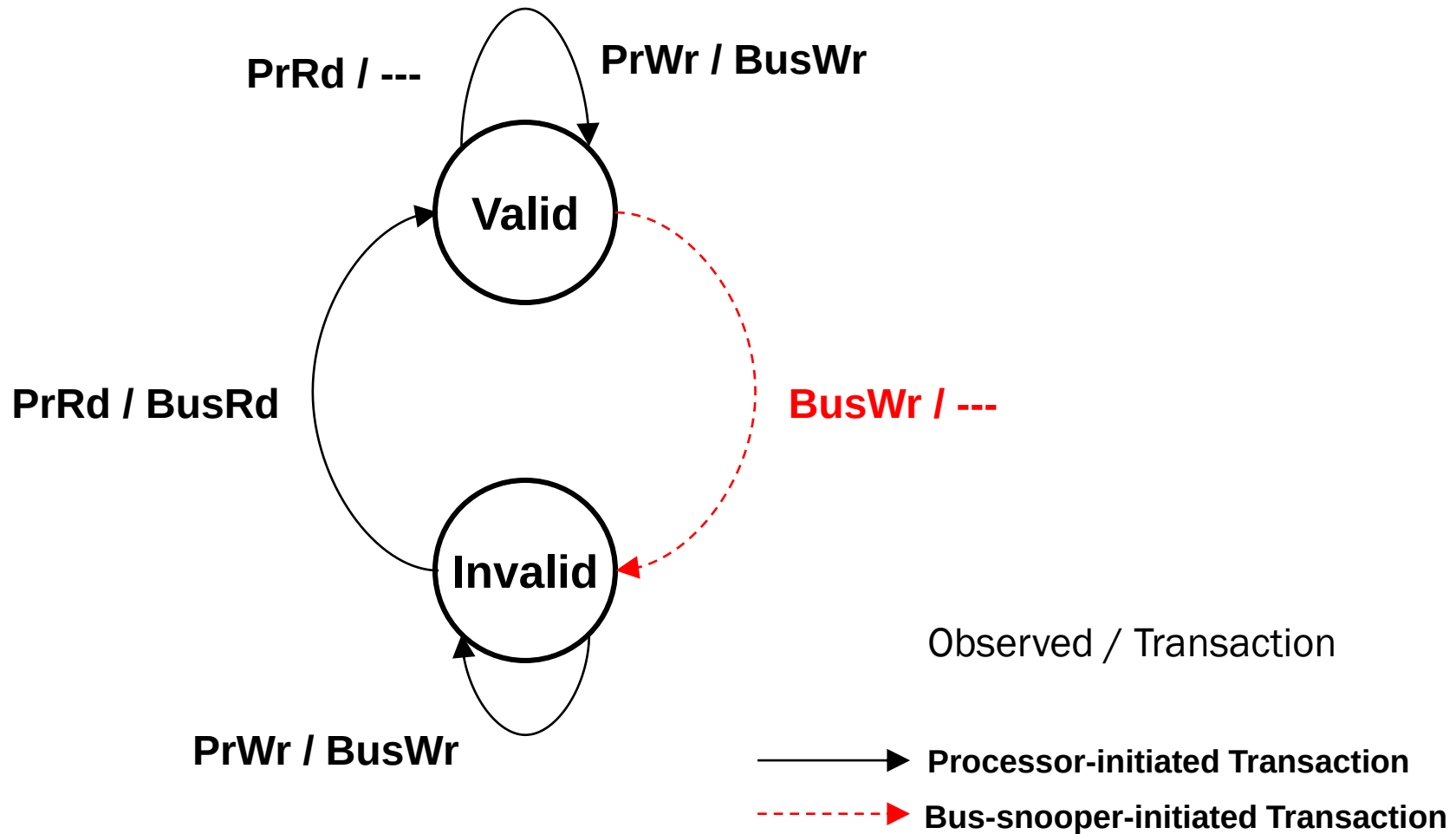


Bus Snooping

- **Invalidation-based** Protocol on **Write-Through** cache



A Simple Snoopy Coherence Protocol for a WT, No Write-Allocate Cache



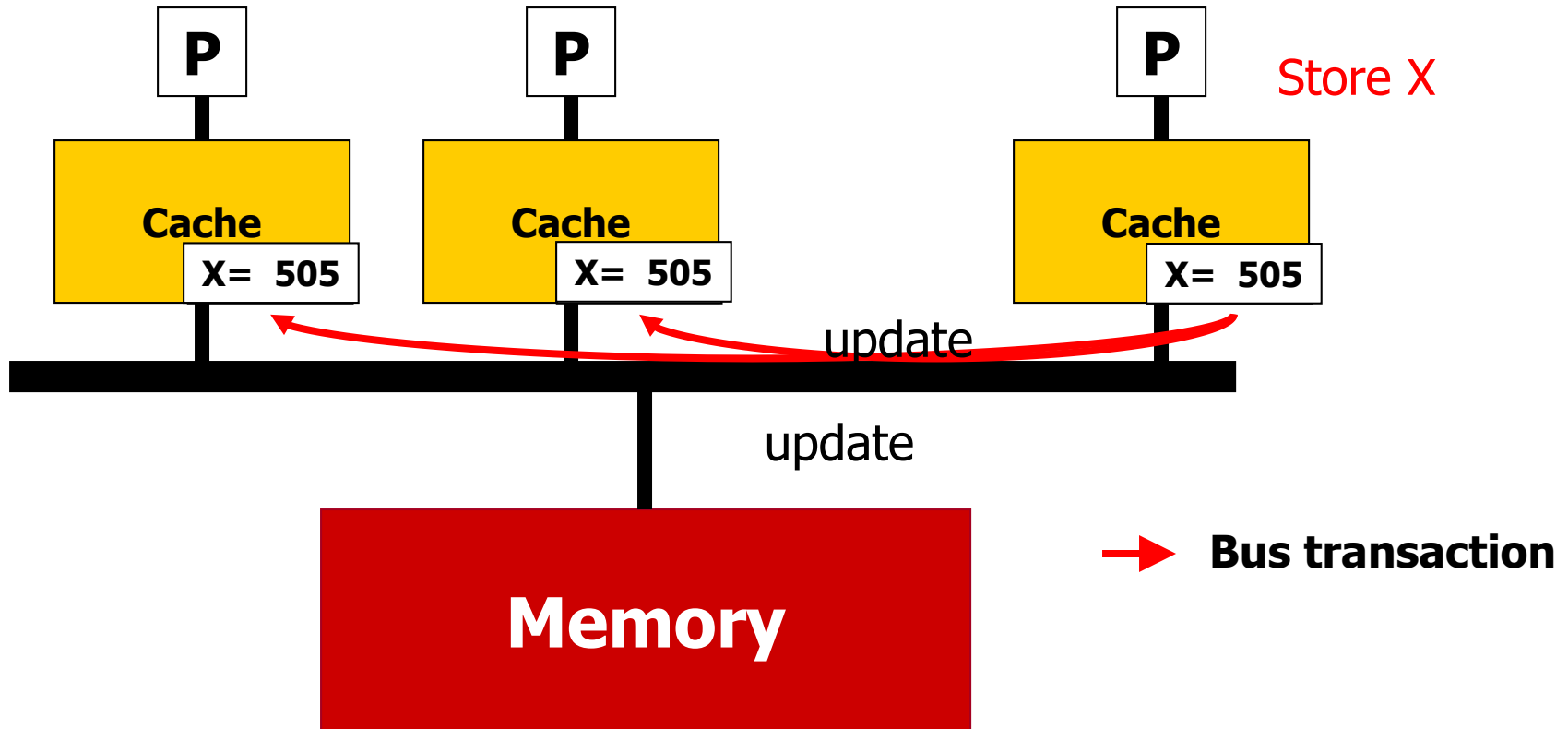
How about Writeback Cache?

- WB cache to reduce bandwidth requirement
- The majority of local writes are hidden behind the processor nodes
- How to snoop?
- Write Ordering

Cache Coherence Protocols for WB Caches

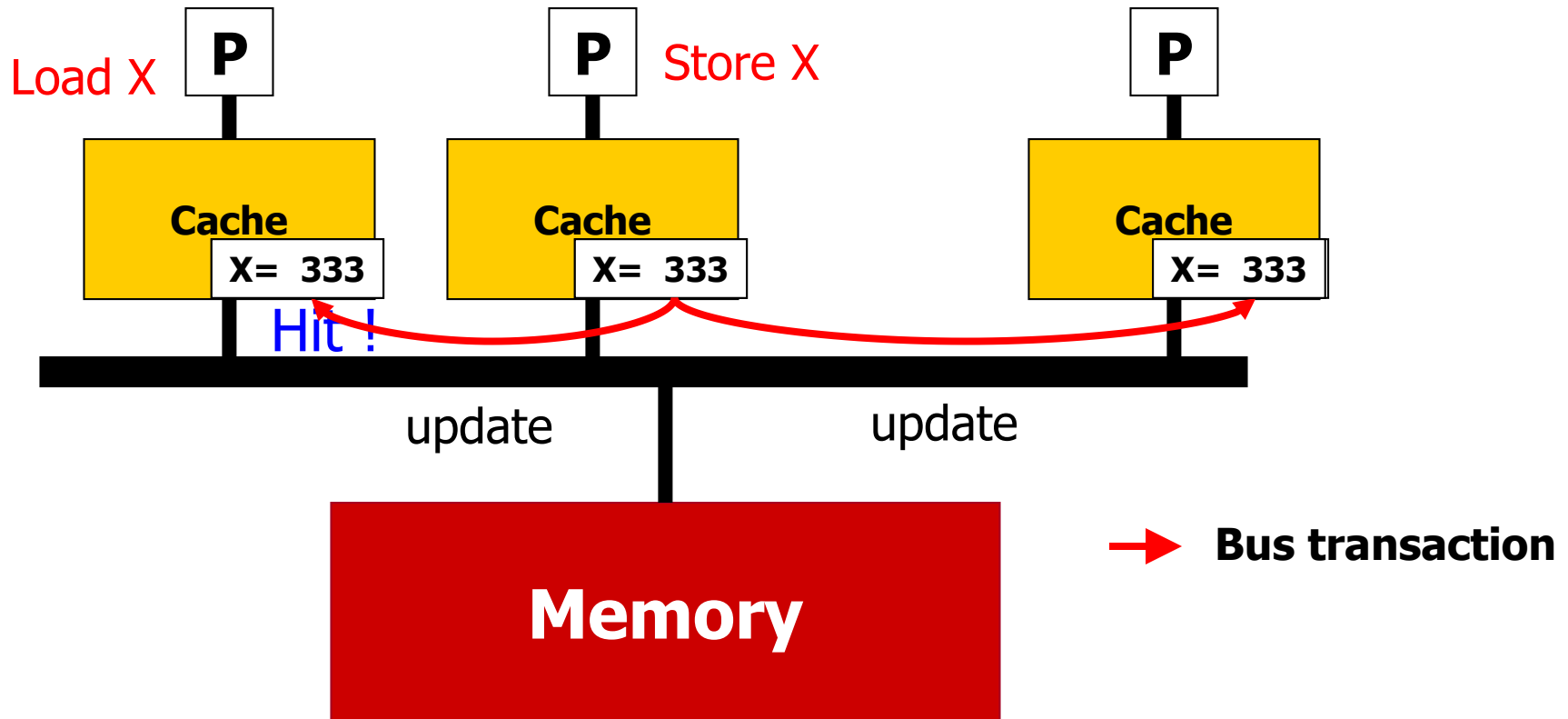
- A cache has an *exclusive* copy of a line if
 - It is the only cache having a valid copy
 - Memory may or may not have it
- Modified (dirty) cache line
 - The cache having the line is the *owner* of the line, because it must supply the block

Update-based Protocol on WB Cache



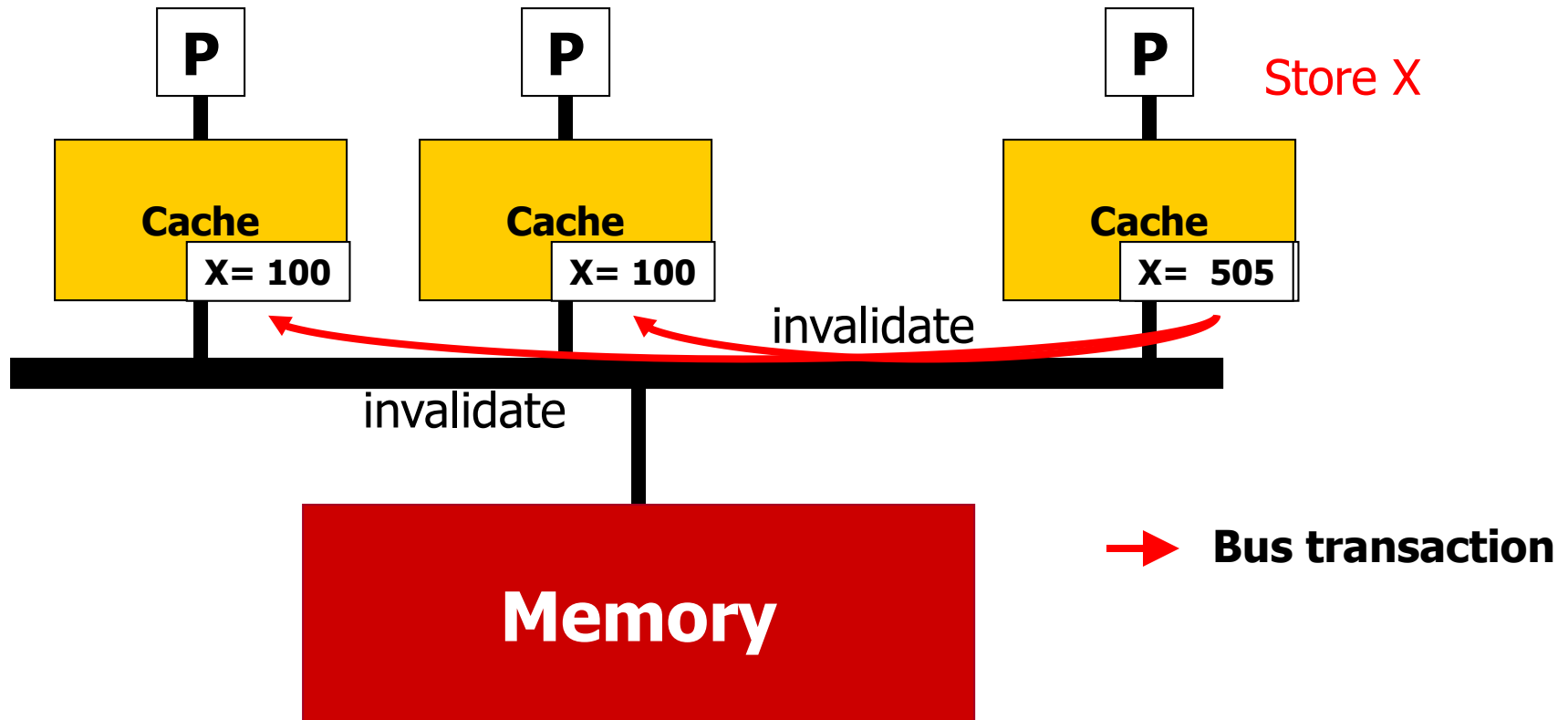
- Update data for all processor nodes who share the same data
- Because a processor node keeps updating the memory location, a lot of traffic will be incurred

Update-based Protocol on WB Cache



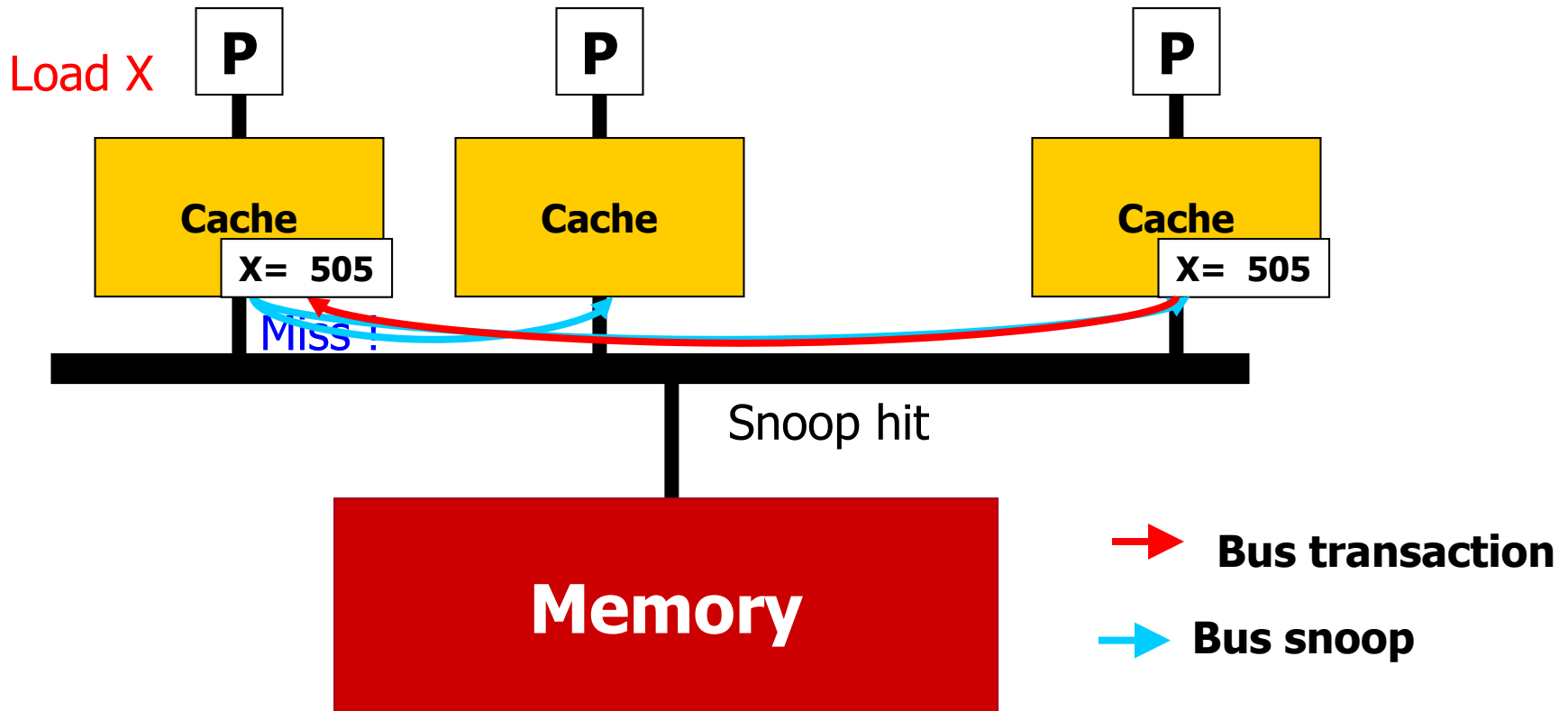
- Update data for all processor nodes who share the same data
- Because a processor node keeps updating the memory location, a lot of traffic will be incurred

Invalidation-based Protocol on WB Cache



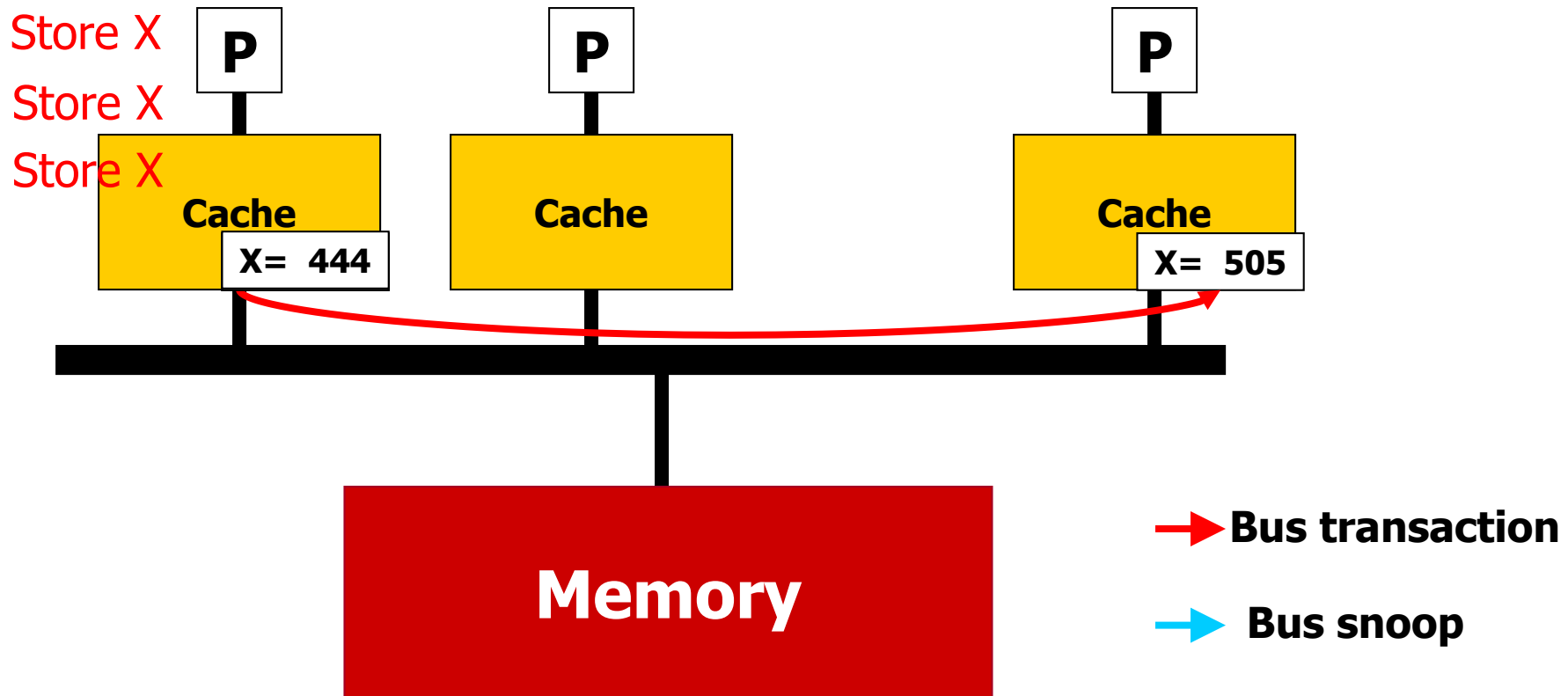
- Invalidate the data copies for the sharing processor nodes
- Reduced traffic when a processor node keeps updating the same memory location

Invalidation-based Protocol on WB Cache



- Invalidate the data copies for the sharing processor nodes
- Reduced traffic when a processor node keeps updating the same memory location

Invalidation-based Protocol on WB Cache



- Invalidate the data copies for the sharing processor nodes
- Reduced traffic when a processor node keeps updating the same memory location

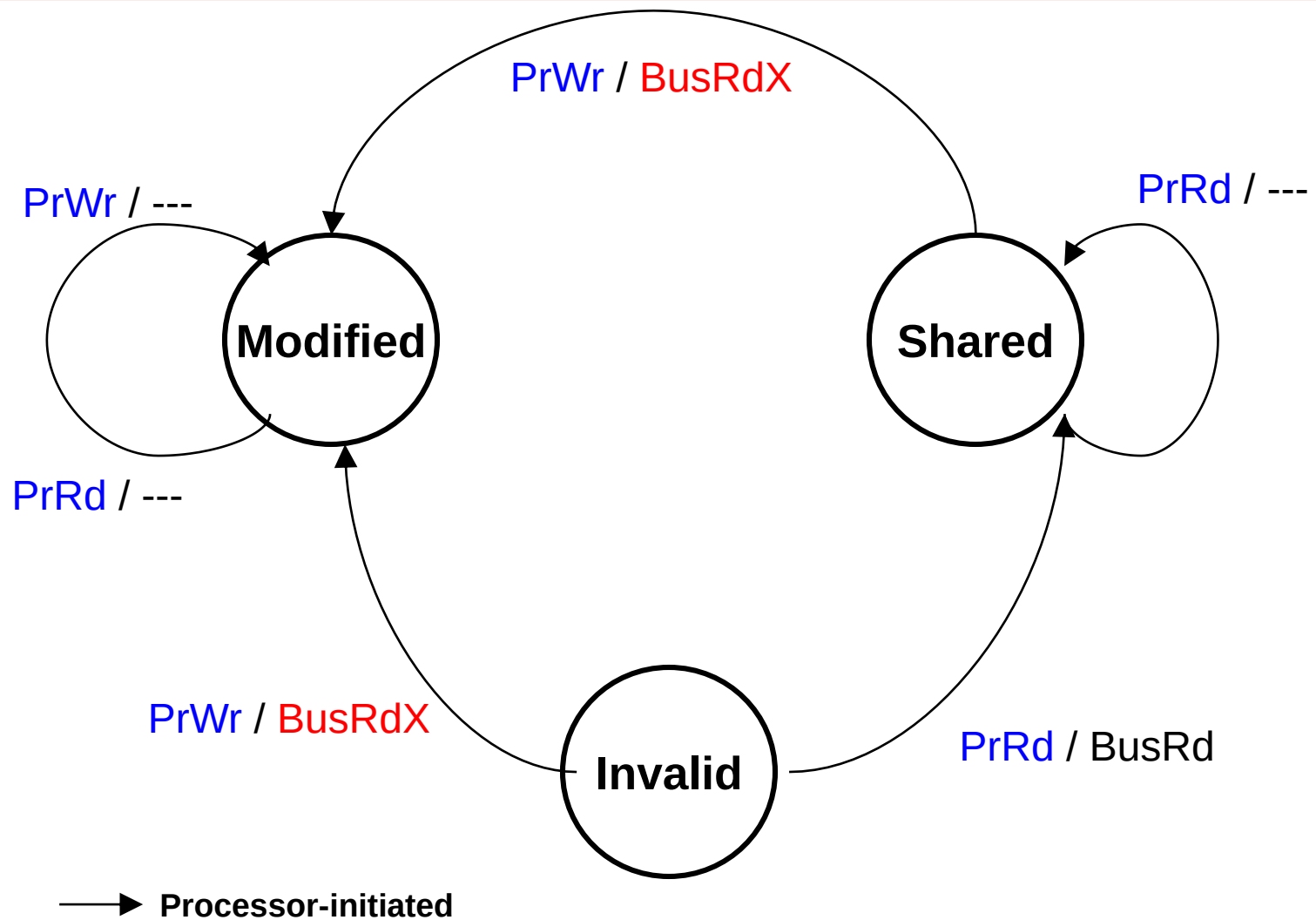
MSI Writeback Invalidation Protocol

- **M**odified
 - Dirty
 - Only this cache has a valid copy
- **S**hared
 - Memory is consistent
 - One or more caches have a valid copy
- **I**nvalid
- Writeback protocol: A cache line can be written multiple times before the memory is updated

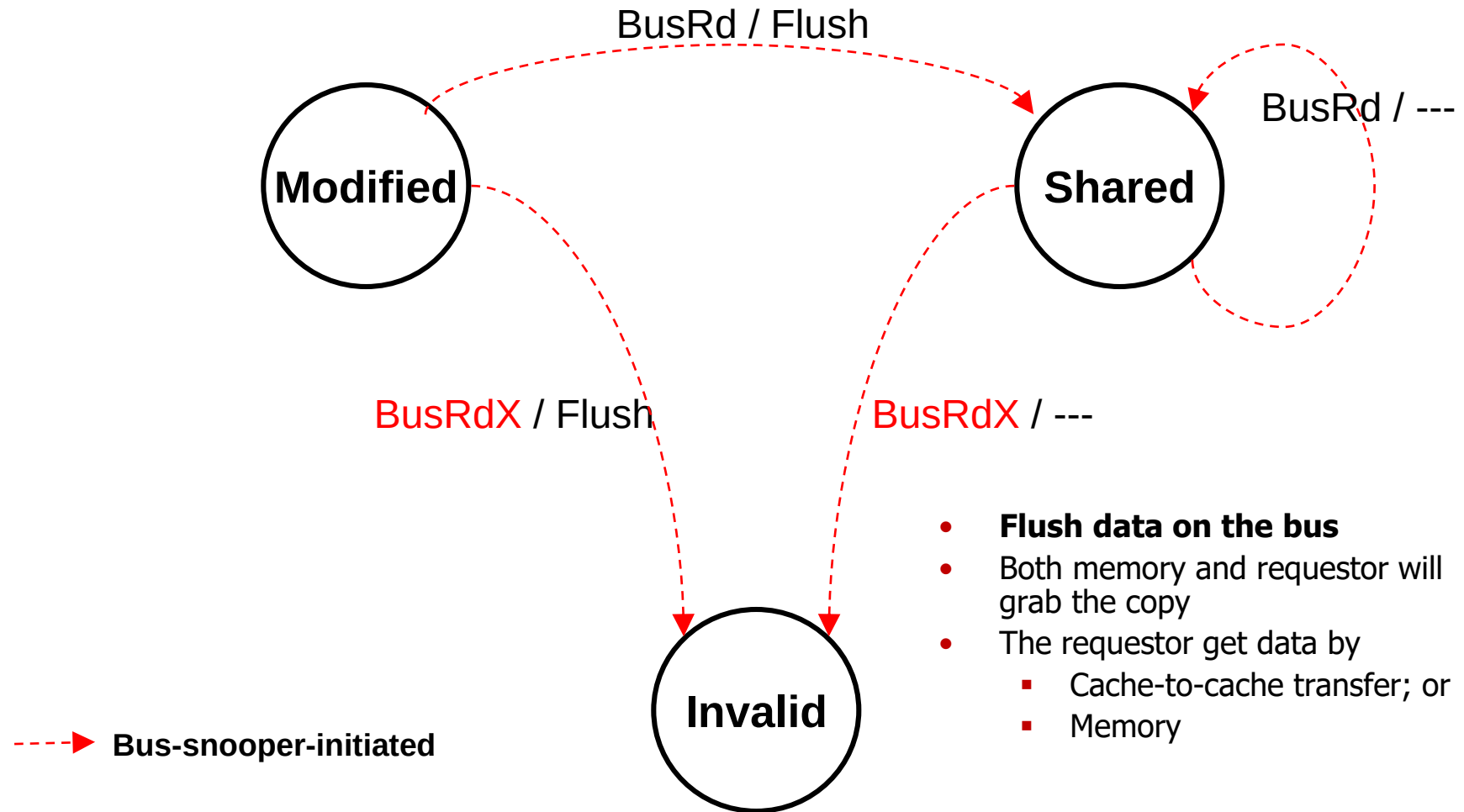
MSI Writeback Invalidation Protocol

- Two types of request from the **processor**
 - PrRd
 - PrWr
- Three types of **bus transactions** post by cache controller
 - **BusRd**
 - PrRd misses the cache
 - Memory or another cache supplies the line
 - **BusRd eXclusive** (Read-to-own)
 - PrWr is issued to a line which is **not** in the **Modified** state
 - **BusWB**
 - Writeback due to replacement
 - Processor does not directly involve in initiating this operation

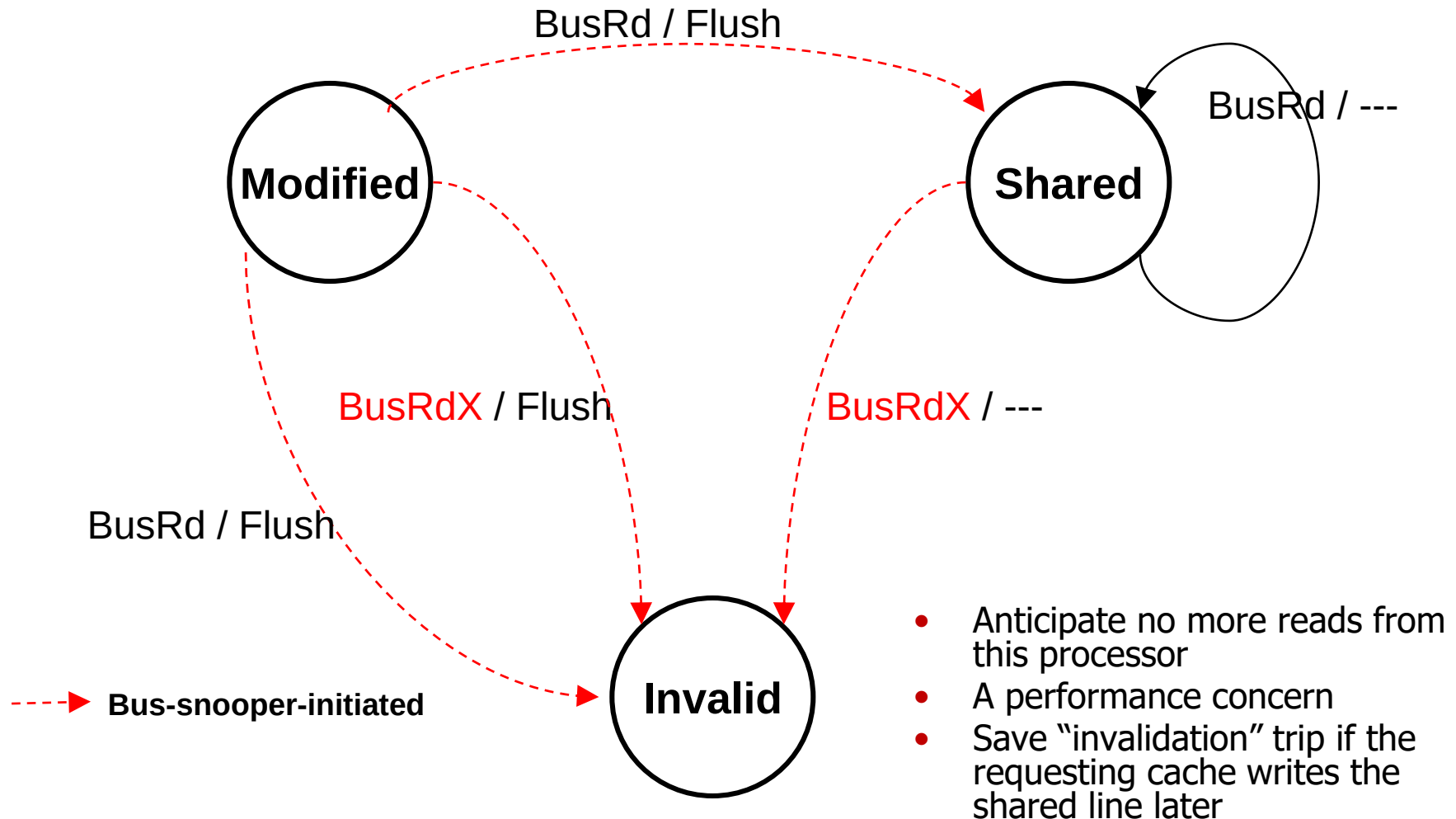
MSI Writeback Invalidation Protocol (Processor Request)



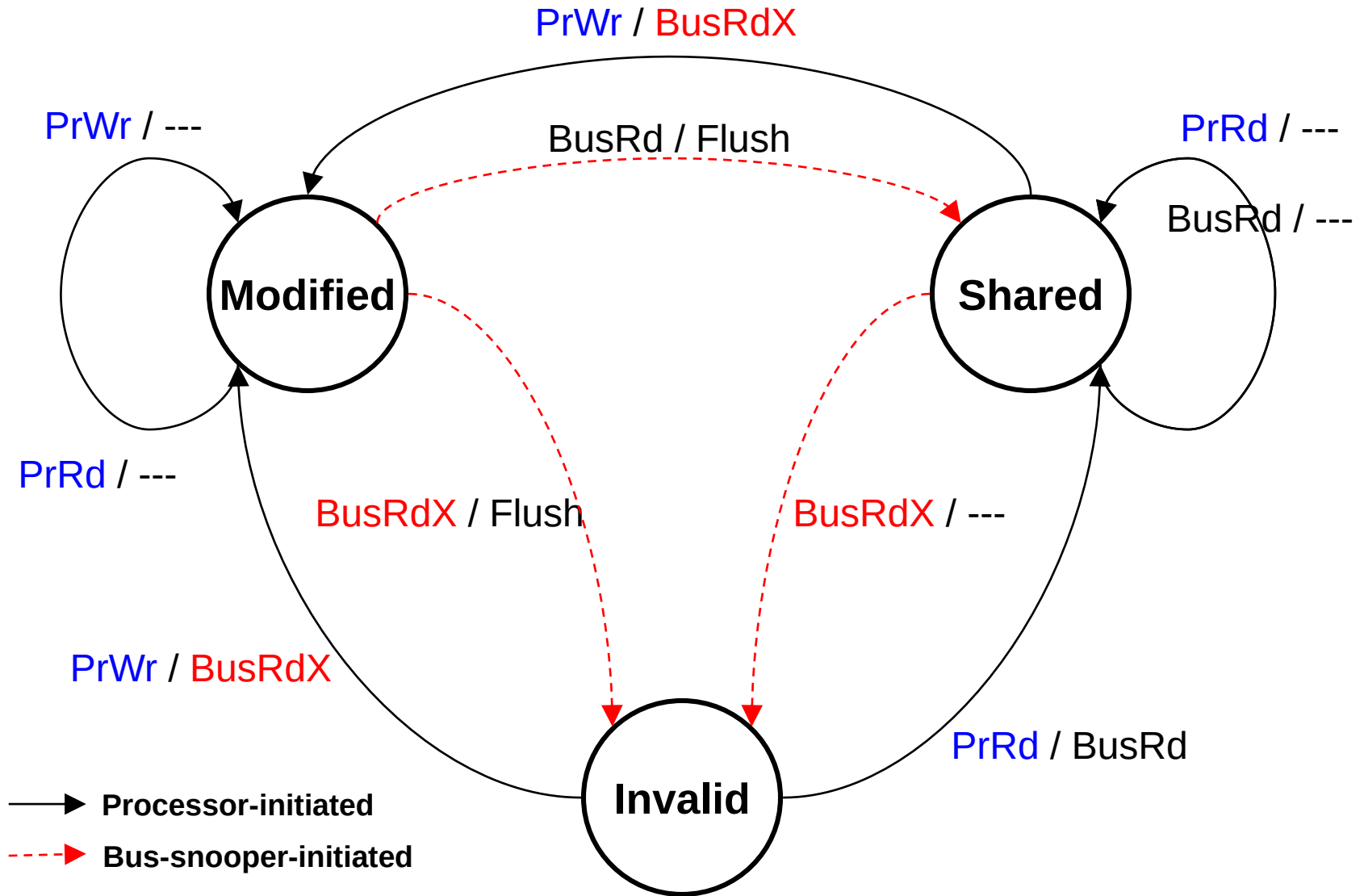
MSI Writeback Invalidation Protocol (Bus Transaction)



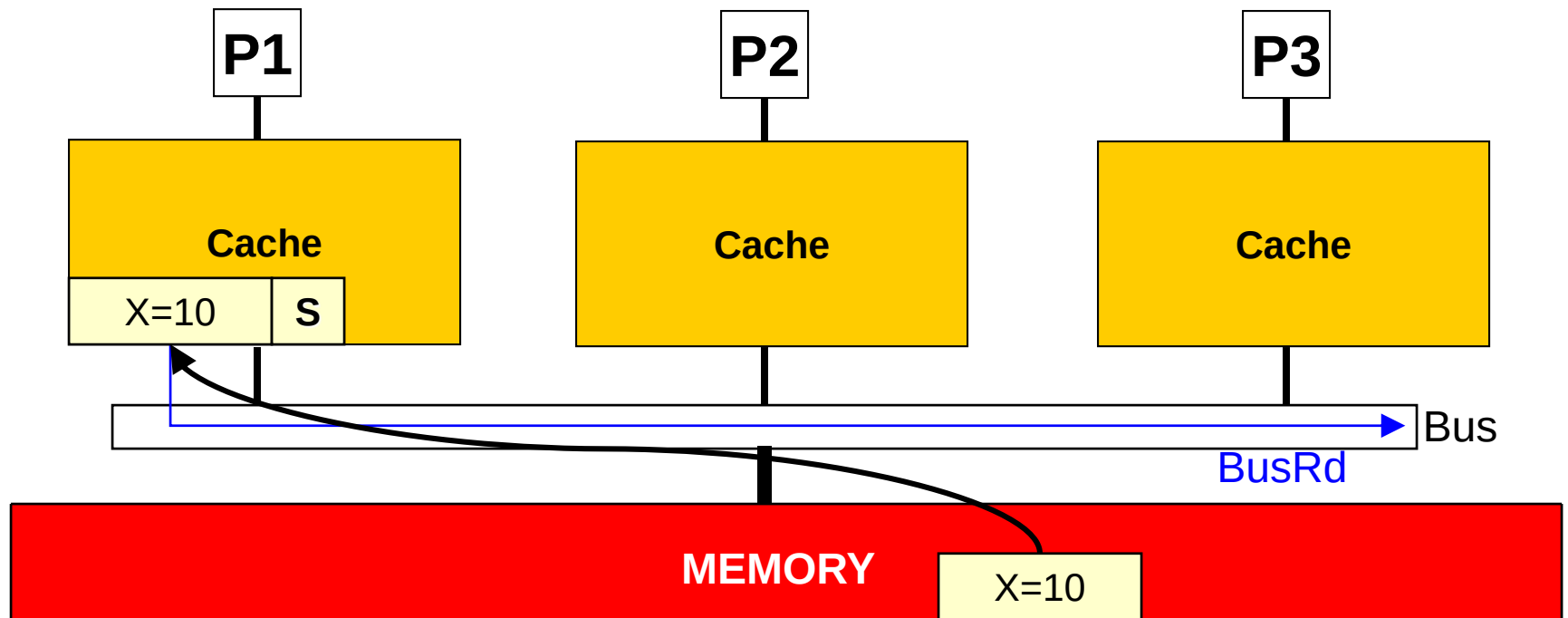
MSI Writeback Invalidation Protocol (Bus transaction) Another possible Implementation



MSI Writeback Invalidation Protocol

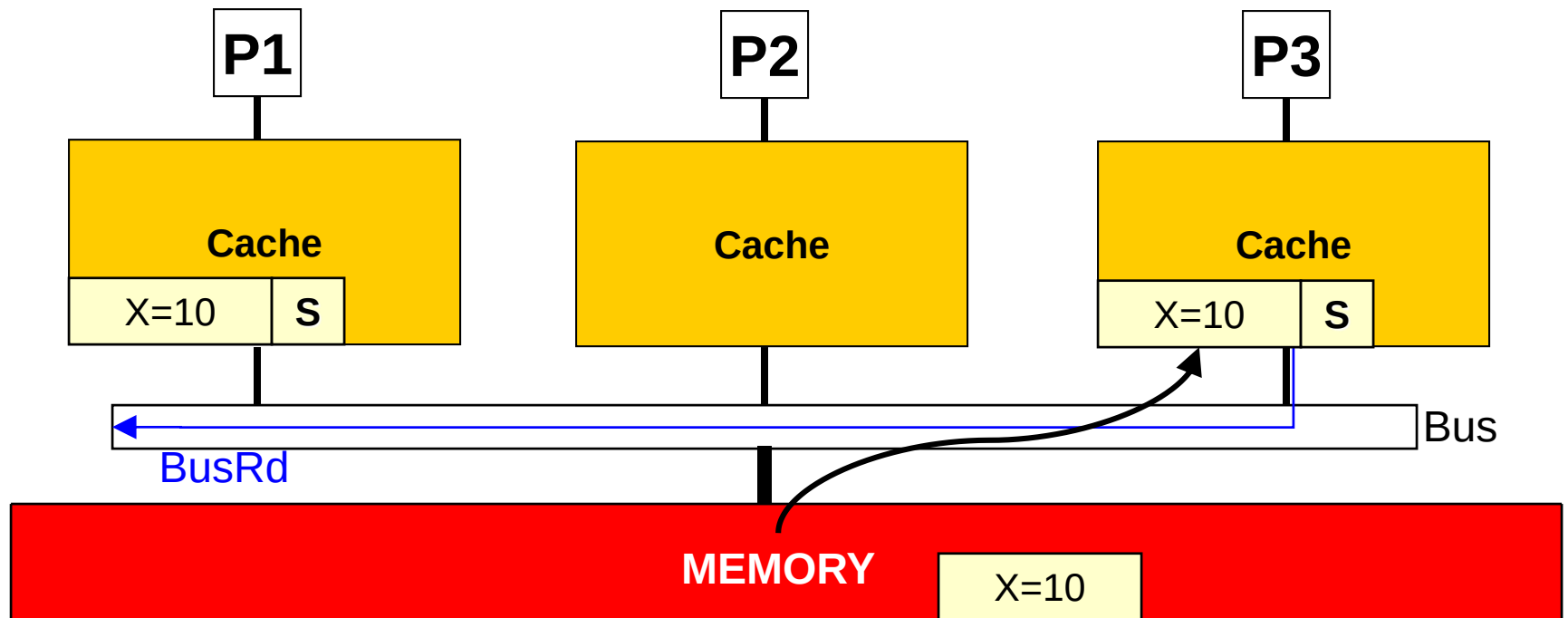


MSI Example



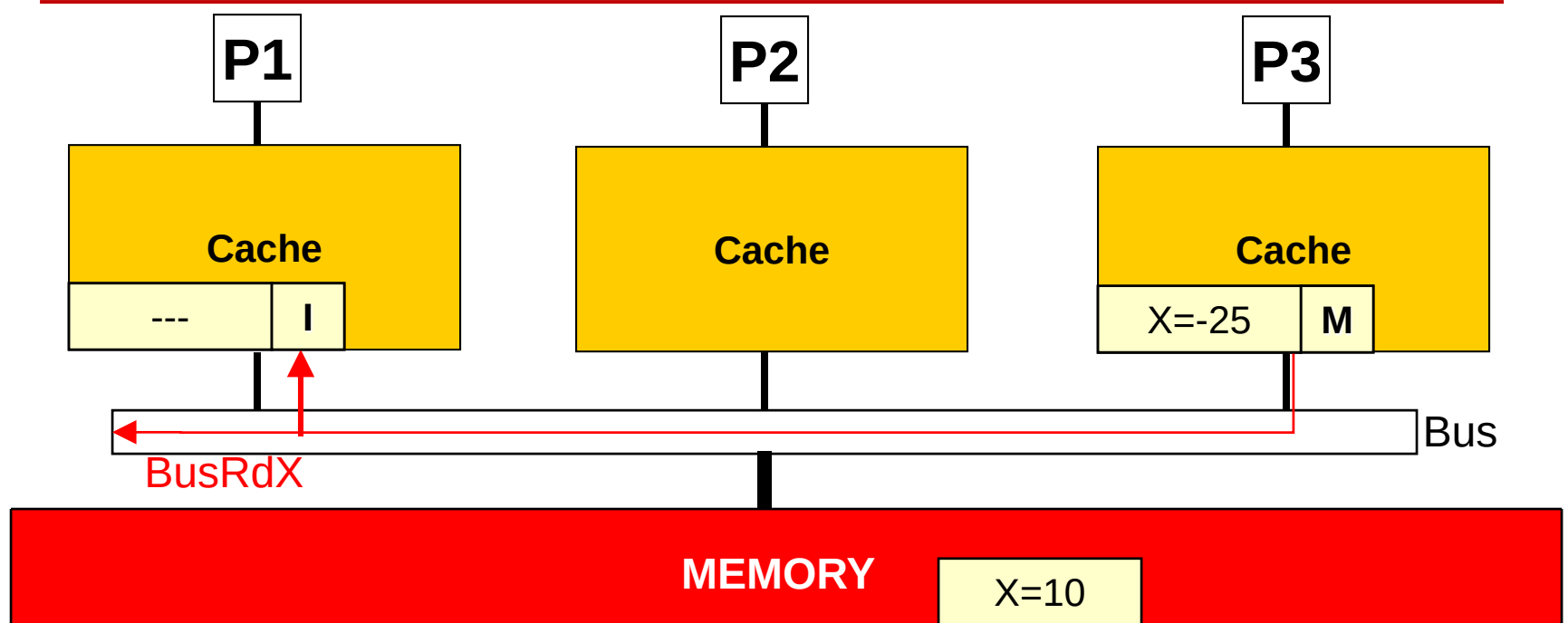
Processor Action	State in P1	State in P2	State in P3	Bus Transaction	Data Supplier
P1 reads X	S	---	---	BusRd	Memory

MSI Example



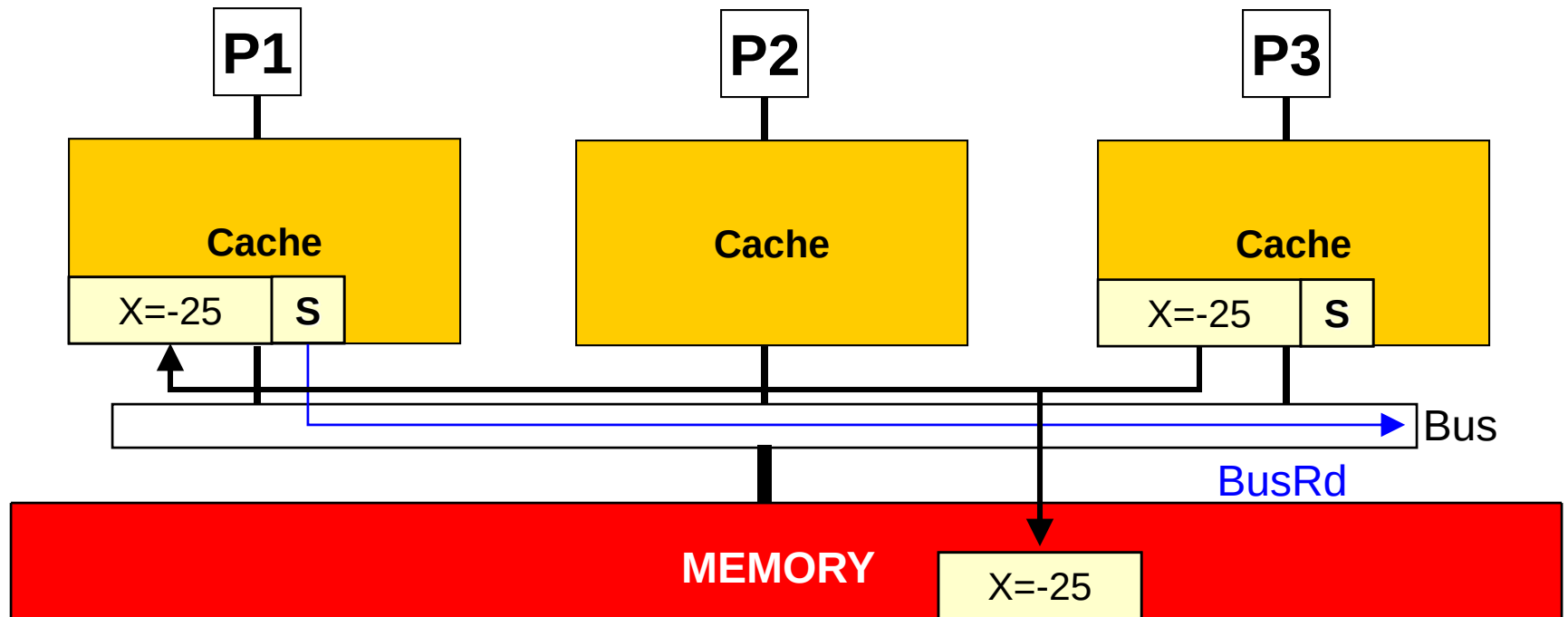
Processor Action	State in P1	State in P2	State in P3	Bus Transaction	Data Supplier
P1 reads X	S	---	---	BusRd	Memory
P3 reads X	S	---	S	BusRd	Memory

MSI Example



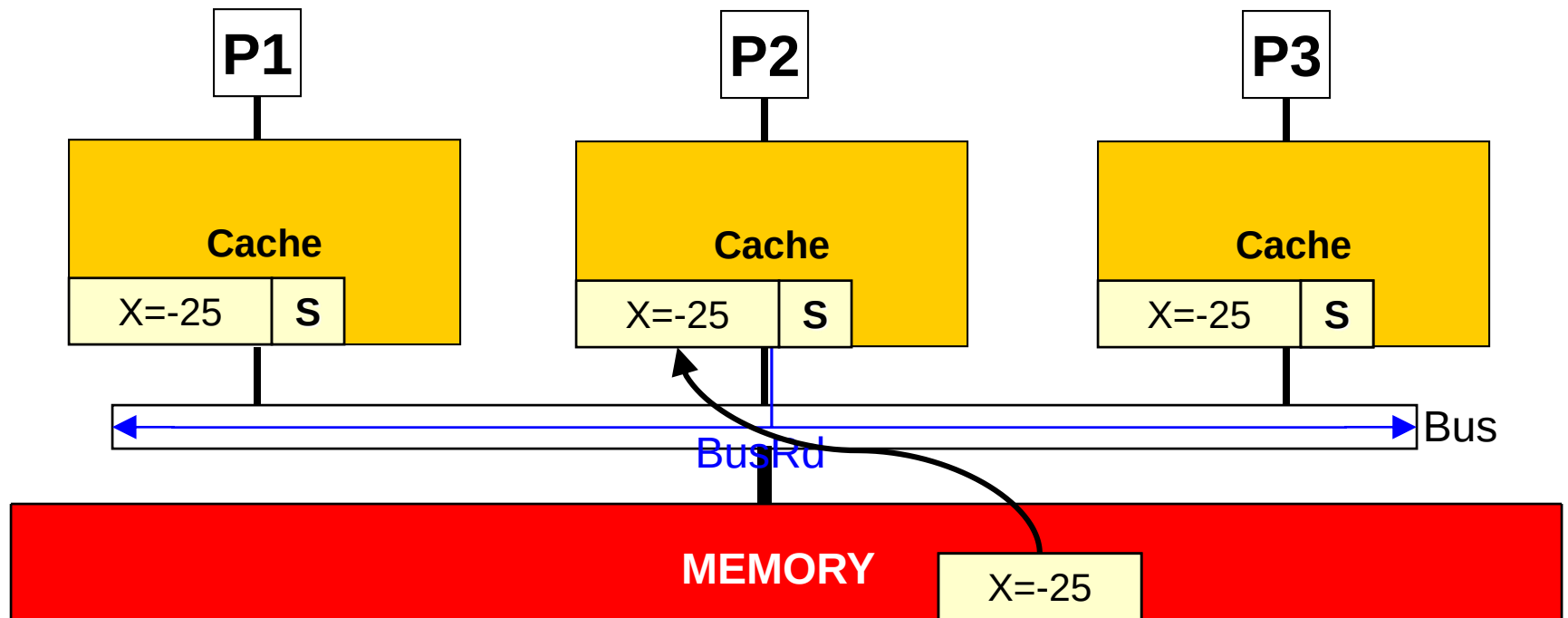
Processor Action	State in P1	State in P2	State in P3	Bus Transaction	Data Supplier
P1 reads X	S	---	---	BusRd	Memory
P3 reads X	S	---	S	BusRd	Memory
P3 writes X	I	---	M	BusRdX	

MSI Example



Processor Action	State in P1	State in P2	State in P3	Bus Transaction	Data Supplier
P1 reads X	S	---	---	BusRd	Memory
P3 reads X	S	---	S	BusRd	Memory
P3 writes X	I	---	M	BusRdX	
P1 reads X	S	---	S	BusRd	P3 Cache

MSI Example

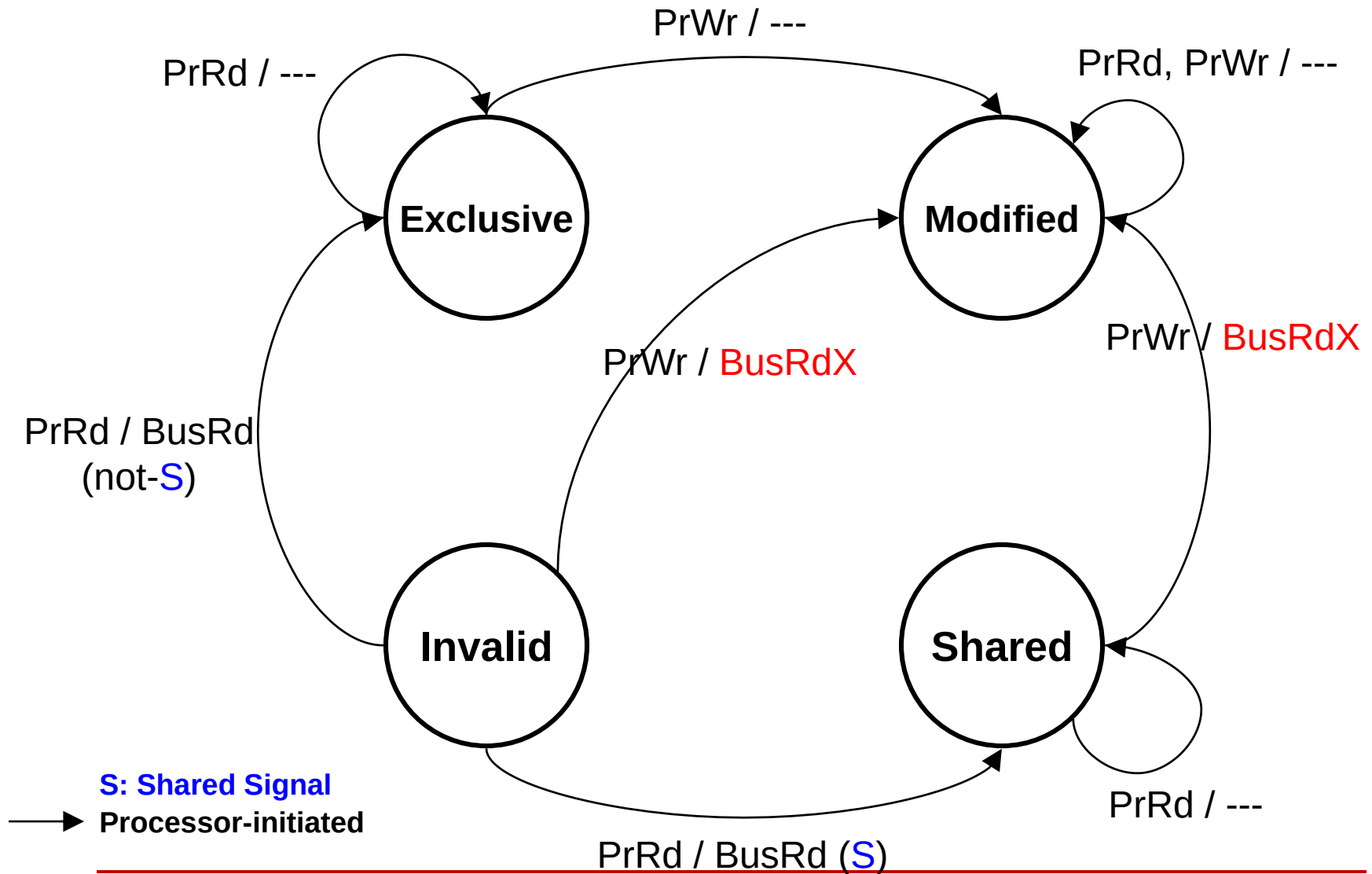


Processor Action	State in P1	State in P2	State in P3	Bus Transaction	Data Supplier
P1 reads X	S	---	---	BusRd	Memory
P3 reads X	S	---	S	BusRd	Memory
P3 writes X	I	---	M	BusRdX	
P1 reads X	S	---	S	BusRd	P3 Cache
P2 reads X	S	S	S	BusRd	Memory

MESI Writeback Invalidation Protocol

- To reduce two types of unnecessary bus transactions
 - BusRdX that snoops and converts the block from S to M when only you are the sole owner of the block
 - BusRd that gets the line in S state when there is *no sharers* (that lead to the overhead above)
- Introduce the **E**xclusive state
 - One can write to the copy without generating BusRdX
- Illinois Protocol: Proposed by Pamarcos and Patel in 1984
- Employed in Intel, PowerPC, MIPS

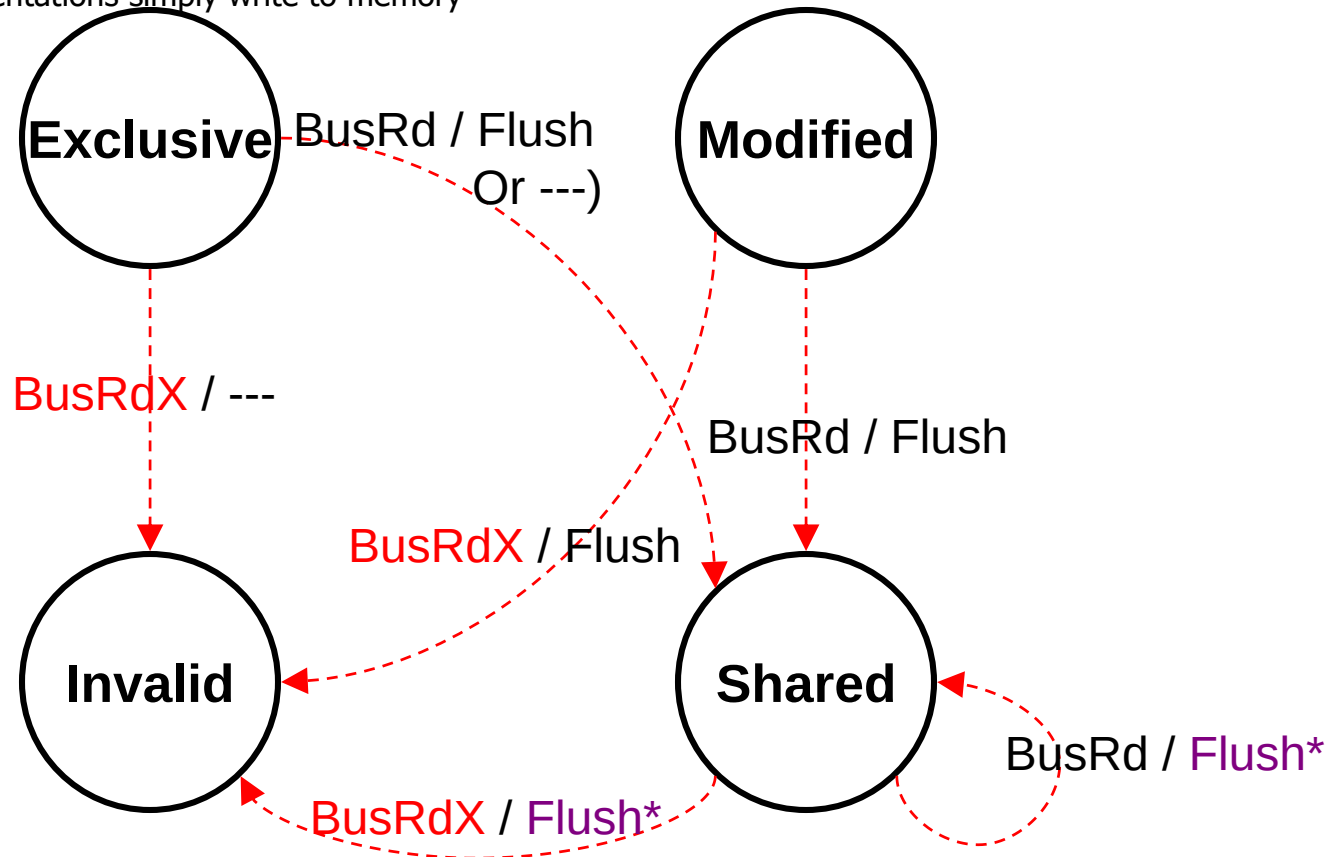
MESI Writeback Invalidation Protocol Processor Request (Illinois Protocol)



MESI Writeback Invalidation Protocol

Bus Transactions (Illinois Protocol)

- Whenever possible, Illinois protocol performs \$-to-\$ transfer rather than having memory to supply the data
- Use a Selection algorithm if there are multiple suppliers (Alternative: add an O state or force update memory)
- Most of the MESI implementations simply write to memory

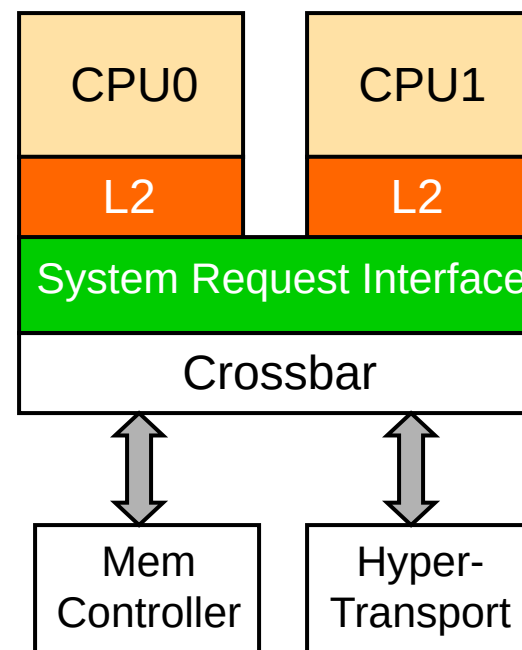


-----> Bus-snooper-initiated

Flush*: Flush for data supplier; no action for other sharers

MOESI Protocol

- Add one additional state — **O**wner state
- Similar to **S**hared state
- The **O** state processor will be responsible for supplying data (copy in memory may be stale)
- Employed by
 - Sun UltraSparc
 - AMD Opteron
 - In dual-core Opteron, cache-to-cache transfer is done through a system request interface (SRI) running at full CPU speed



Implication on Multi-Level Caches

- How to guarantee coherence in a multi-level cache hierarchy
 - Snoop all cache levels?
 - Intel's 8870 chipset has a "snoop filter" for quad-core
- Maintaining **inclusion property**
 - Ensure data in the outer level must be present in the inner level
 - Only snoop the outermost level (e.g. L2)
 - L2 needs to know L1 has write hits
 - Use Write-Through cache
 - Use Write-back but maintain another "modified-but-stale" bit in L2

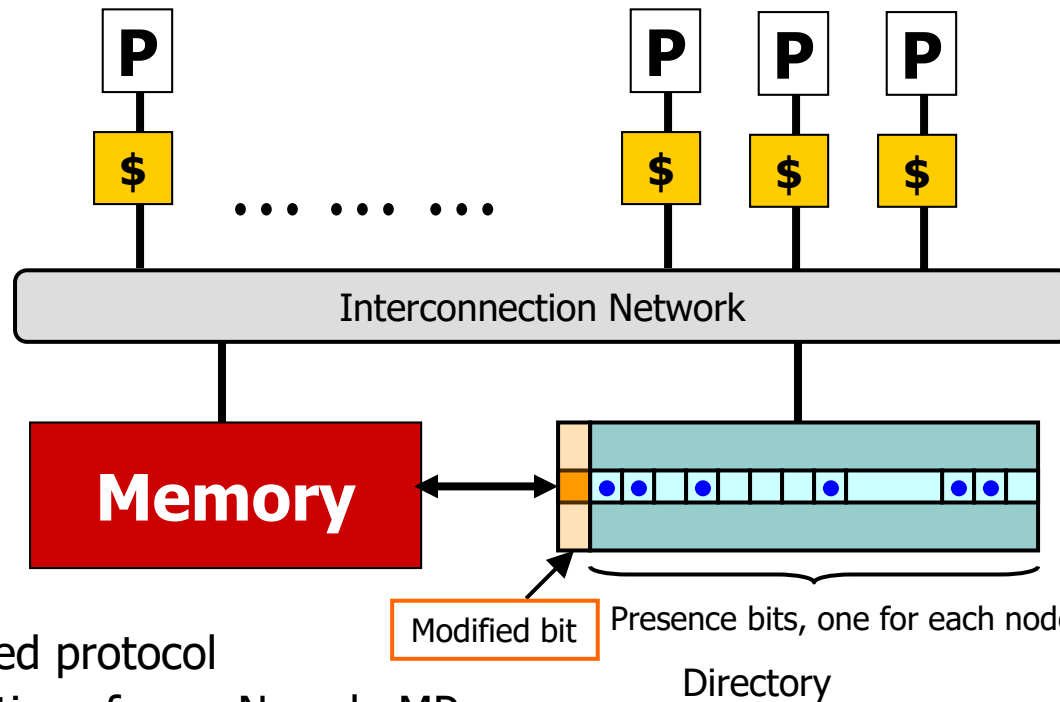
Inclusion Property

- Not so easy ...
 - Replacement
 - Different bus observes different access activities
 - e.g. L2 may replace a line frequently accessed in L1
 - L1 and L2 are 2-way set associative
 - Blocks m1 and m2 go to the same set in L1 and L2
 - A new block m3 mapped to the same entry replaces m1 in L1 and m2 in L2 due to the LRU scheme
 - Split L1 caches and Unified L2
 - Imagine all caches are direct-mapped.
 - m1 (instruction block) and m2 (data block) mapped to the same entry in L2
 - Different cache line sizes
 - What happens if L1's block size is smaller than L2's?

Inclusion Property

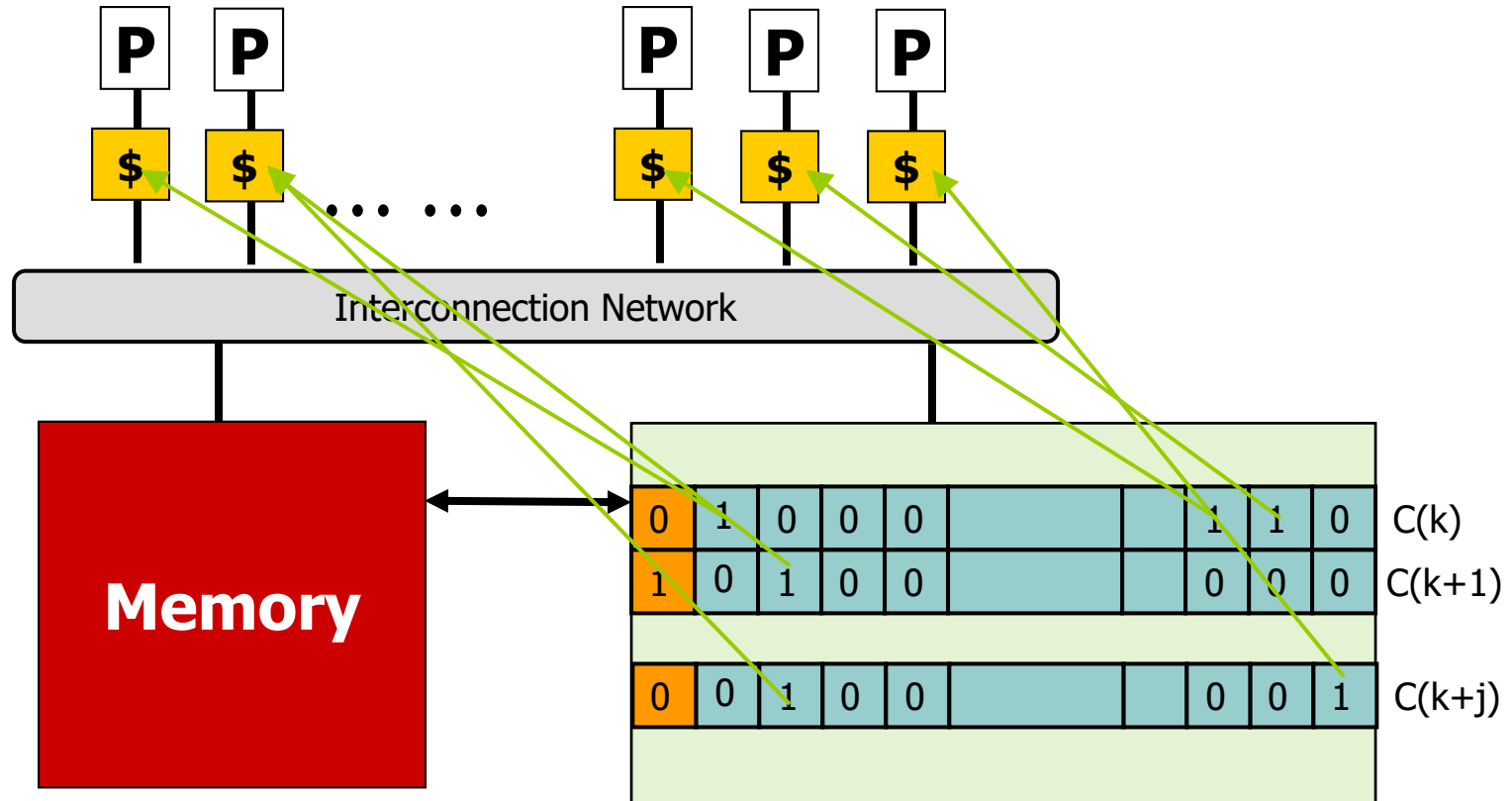
- Use specific cache configurations to maintain the inclusion property automatically
 - E.g., DM L1 + bigger DM or set-associative L2 with the same cache line size ($\#sets\ in\ L2 \geq \#sets\ in\ L1$)
- Explicitly propagate L2 action to L1
 - L2 replacement will flush the corresponding L1 line
 - Observed **BusRdX** bus transaction will invalidate the corresponding L1 line
 - To avoid excess traffic, L2 maintains *an Inclusion bit* for filtering (to indicate in L1 or not)

Directory-based Coherence Protocol



- Snooping-based protocol
 - N transactions for an N-node MP
 - All caches need to watch every memory request from each processor
 - Not a scalable solution for maintaining coherence in large shared memory systems
- Directory protocol
 - Directory-based control of who has what;
 - HW overheads to keep the directory ($\sim \# \text{ lines} * \# \text{ processors}$)

Directory-based Coherence Protocol

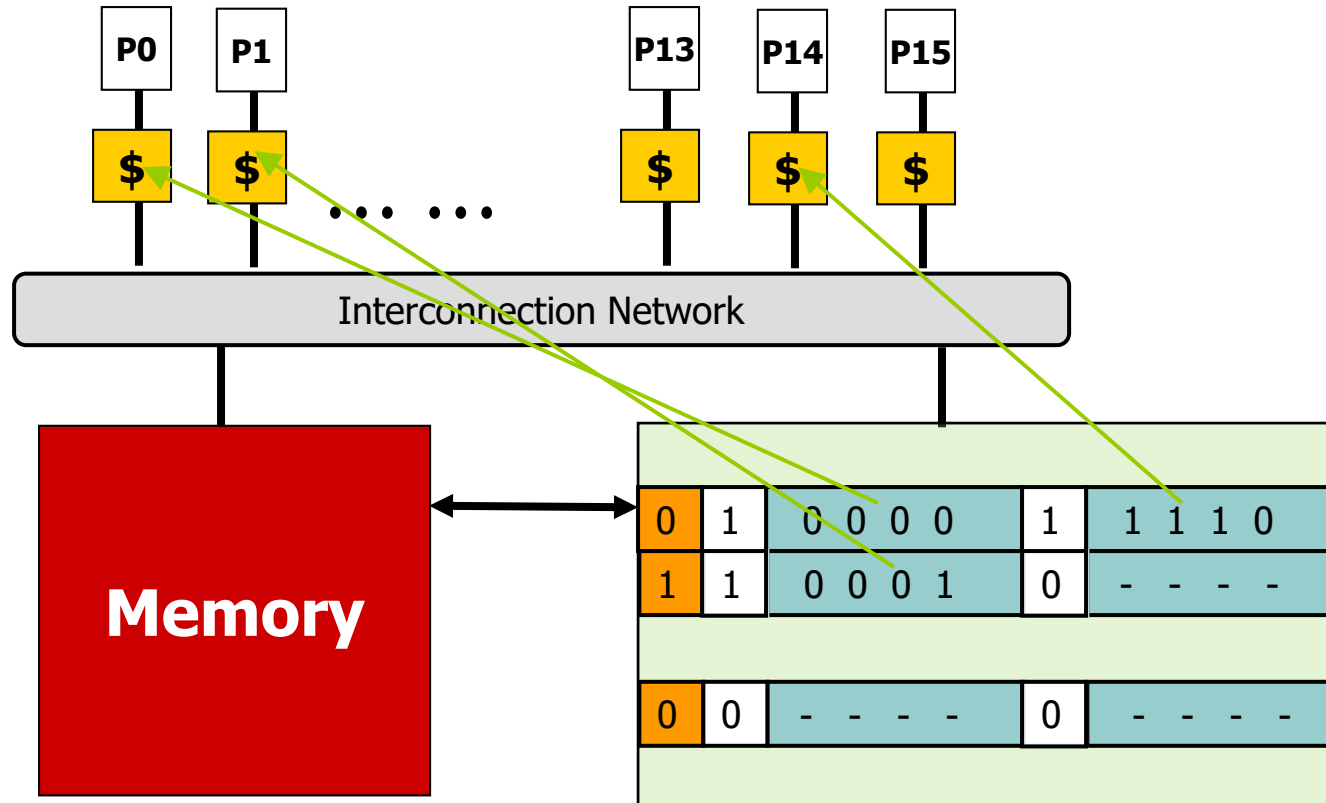


1 modified bit for each cache block in memory



1 presence bit for each processor, each cache block in memory

Directory-based Coherence Protocol (Limited Dir)



1 modified bit for each cache block in memory



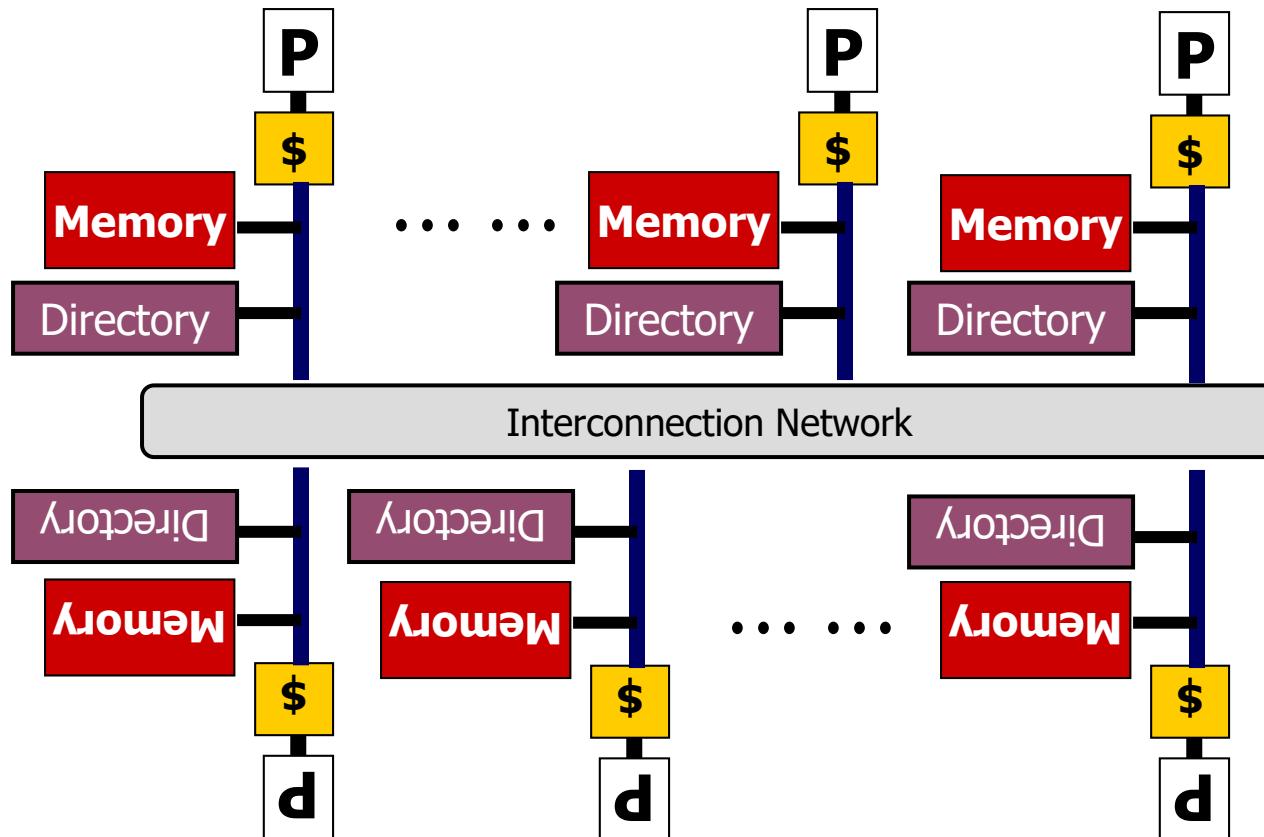
Presence encoding is NULL or not



Encoded Present bits ($\log_2 N$),

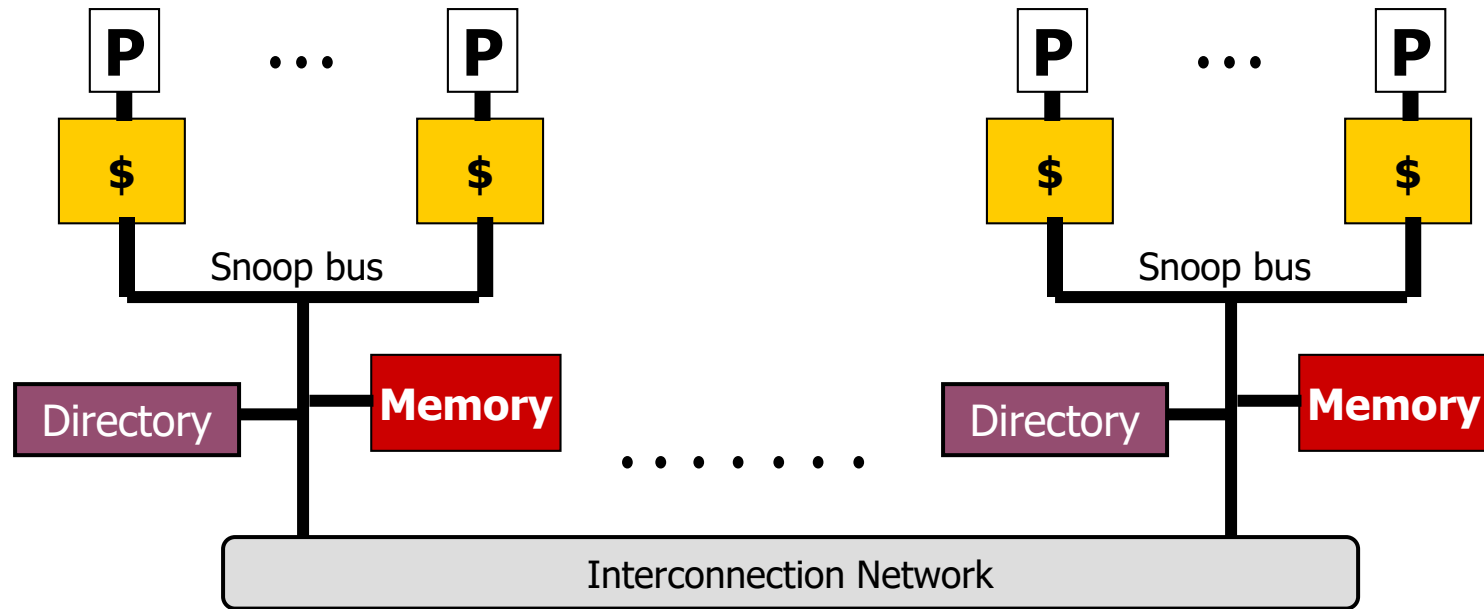
each cache line can reside in **2 processors in this example**

Distributed Directory Coherence Protocol



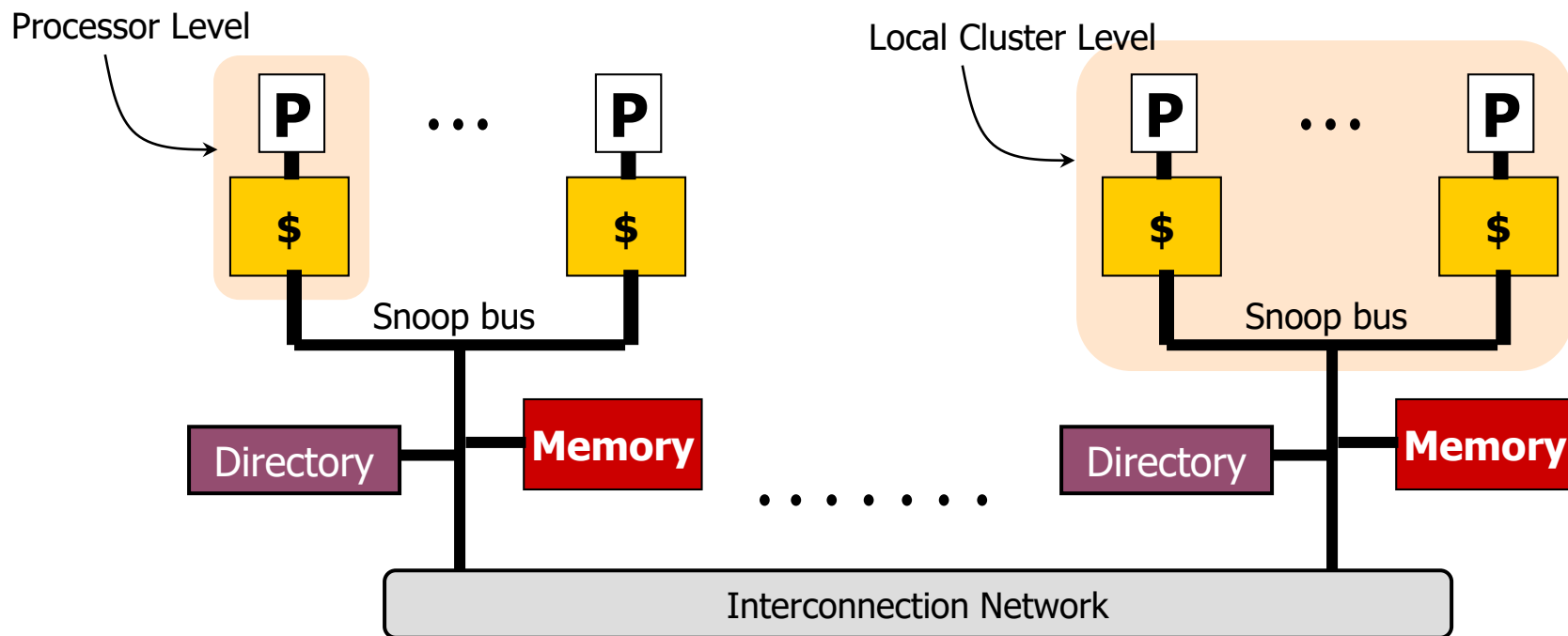
- Centralized directory is less scalable (contention)
- Distributed shared memory (DSM) for a large MP system
- Interconnection network is no longer a shared bus
- Maintain cache coherence (CC-NUMA)
- Each address has a "home"

Stanford DASH



- Stanford DASH: 4 CPUs in each cluster, total 16 clusters (1992)
 - Invalidation-based cache coherence
 - Directory keeps one of the 3 status of a cache block at its home node
 - Uncached
 - Shared (unmodified state)
 - Dirty

DASH Memory Hierarchy (1992)



- Processor Level
- Local Cluster Level
- Home Cluster Level (address is at home)

If dirty, needs to get it from remote node which owns it

- Remote Cluster Level

Stanford DASH

- MIPS R3000
- 33MHz
- 64KB L1 I\$
- 64KB L1 D\$
- 256KB L2
- MESI

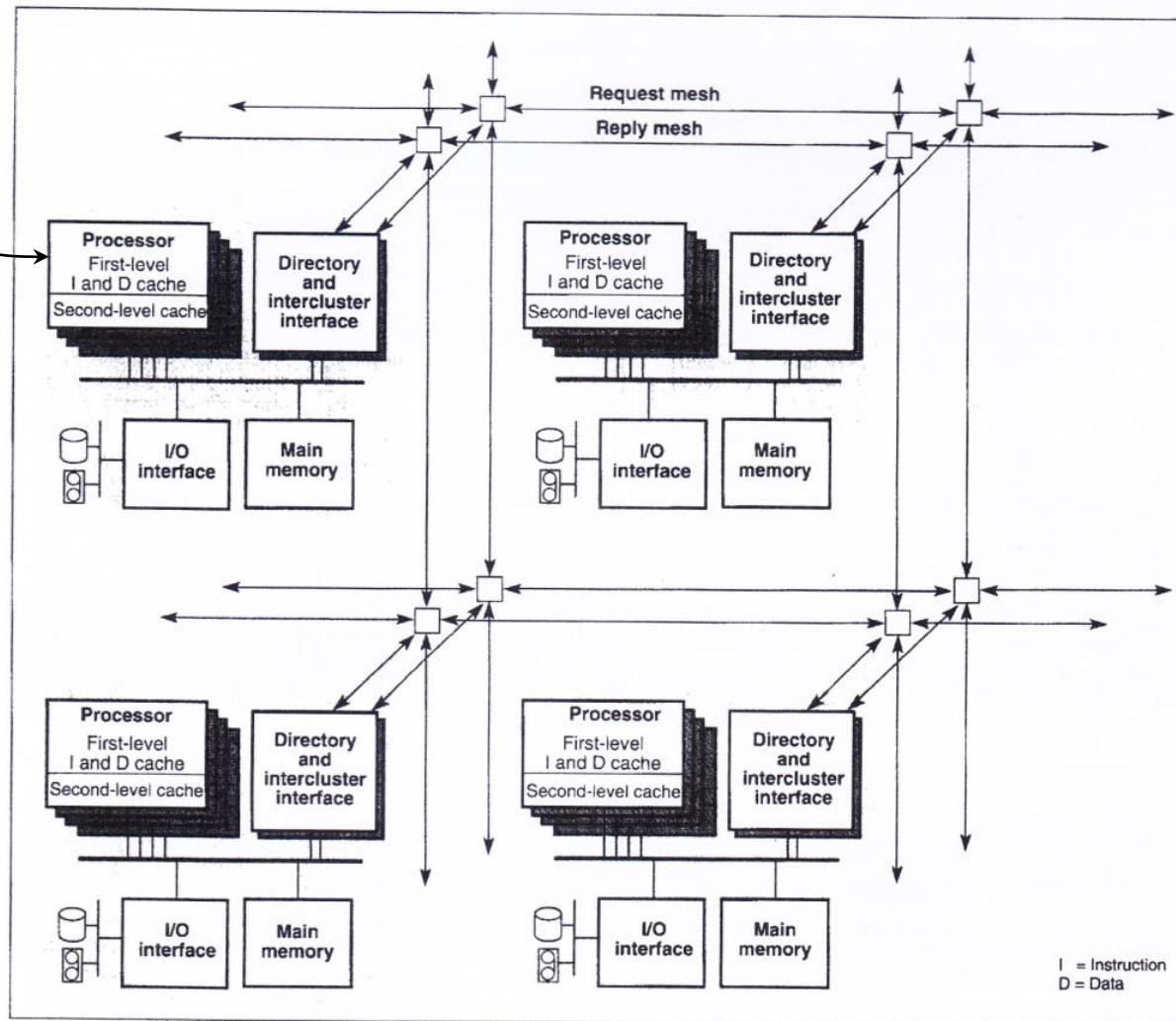
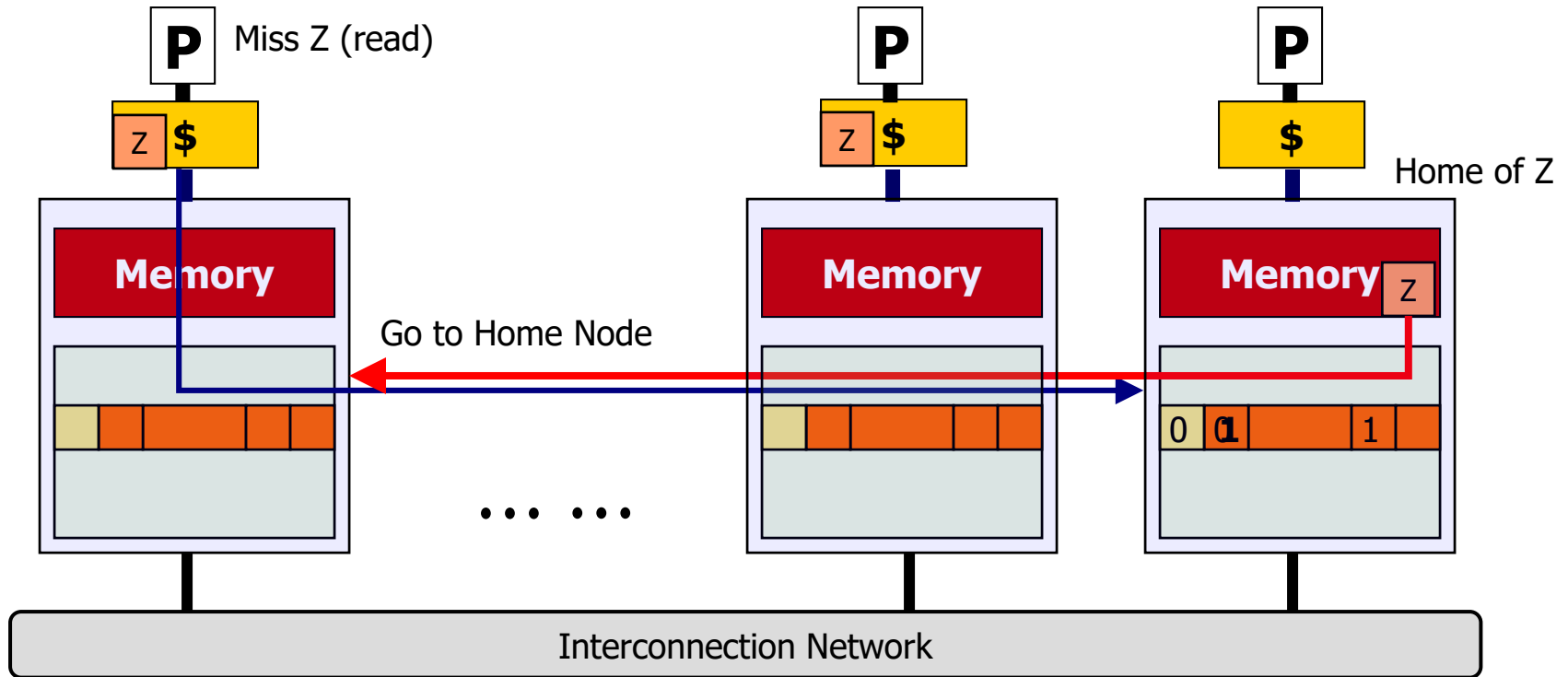


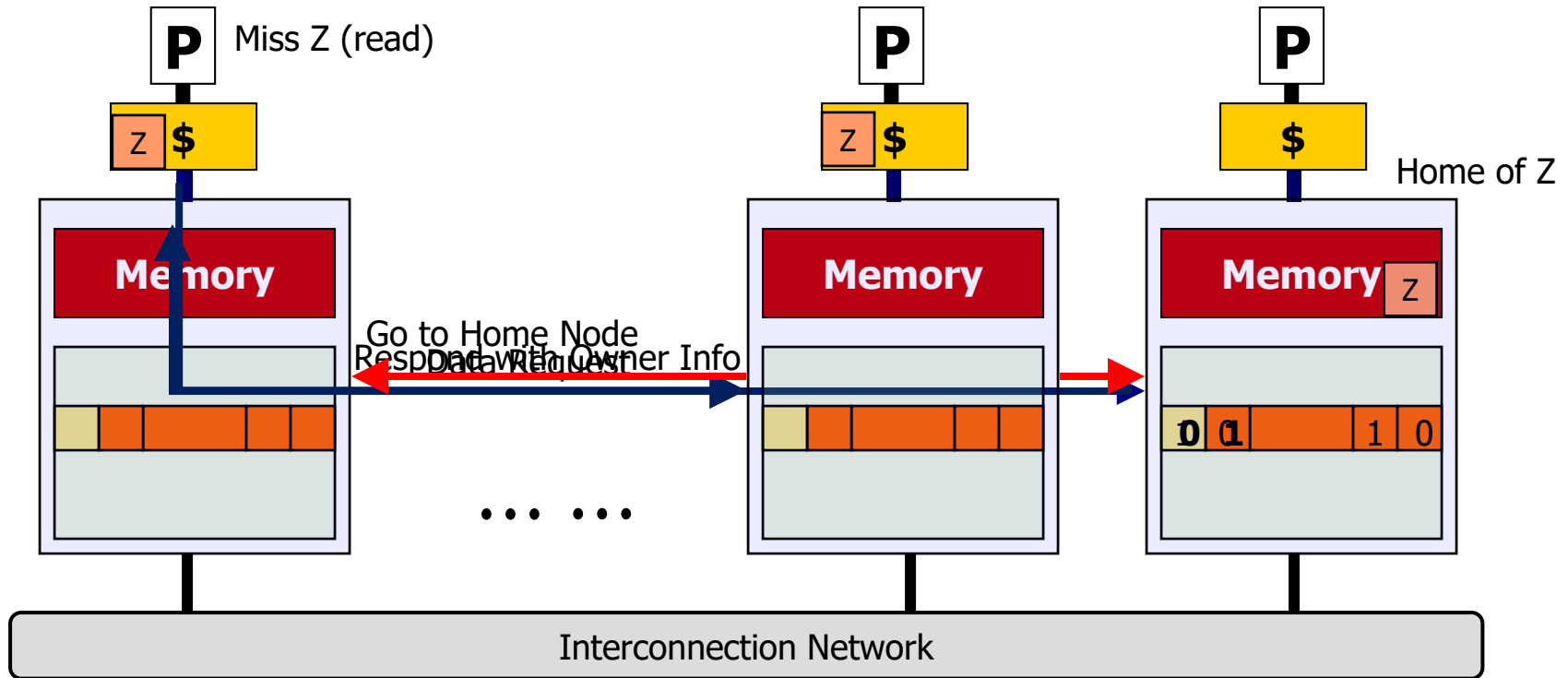
Figure 4. Block diagram of a 2×2 Dash system.

Directory Coherence Protocol: Read Miss



Data Z is shared (clean)

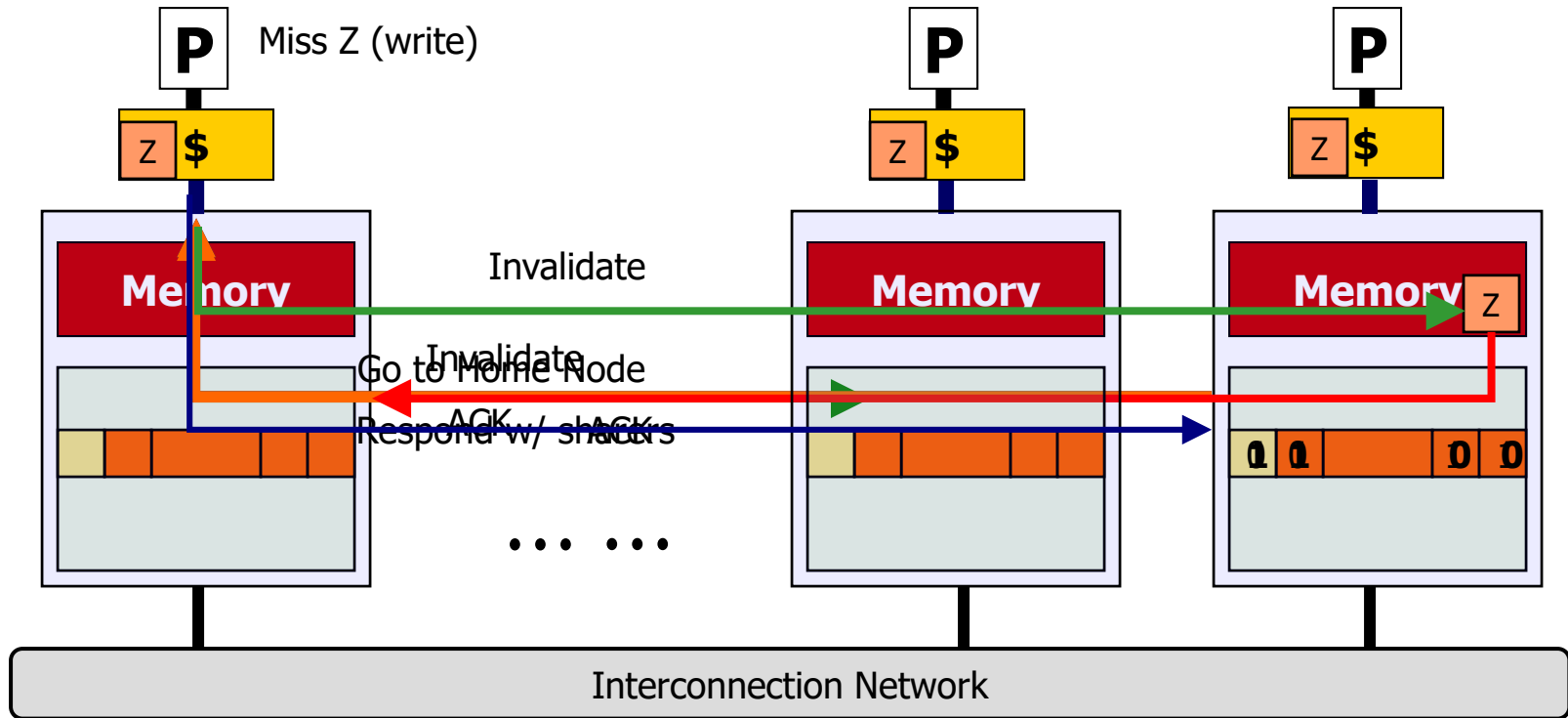
Directory Coherence Protocol: Read Miss



Data Z is **Dirty**

Data Z is **Clean, Shared by 2 nodes**

Directory Coherence Protocol: Write Miss



Write Z can proceed in P0

Memory Consistency Issue

- What do you expect for the following codes?

Initial values

A=0

B=0

P1

P2

A=1;

Flag = 1;

while (Flag==0) {};

print A;

Is it possible P2 prints A=0?

P1

P2

A=1;

B=1;

print B;

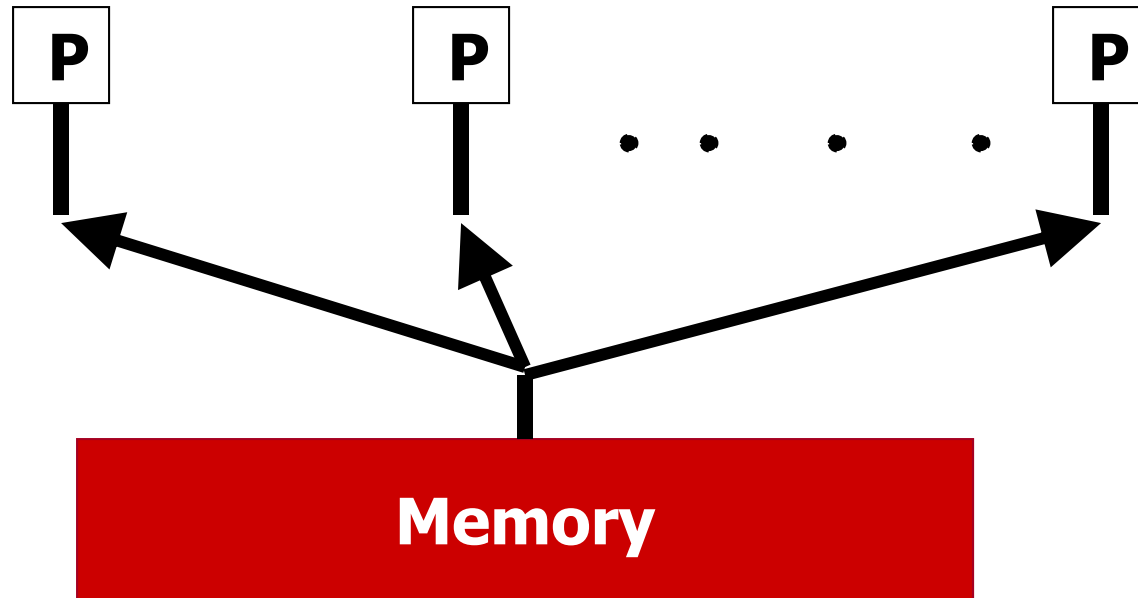
print A;

Is it possible P2 prints B=1, A=0?

Memory Consistency Model

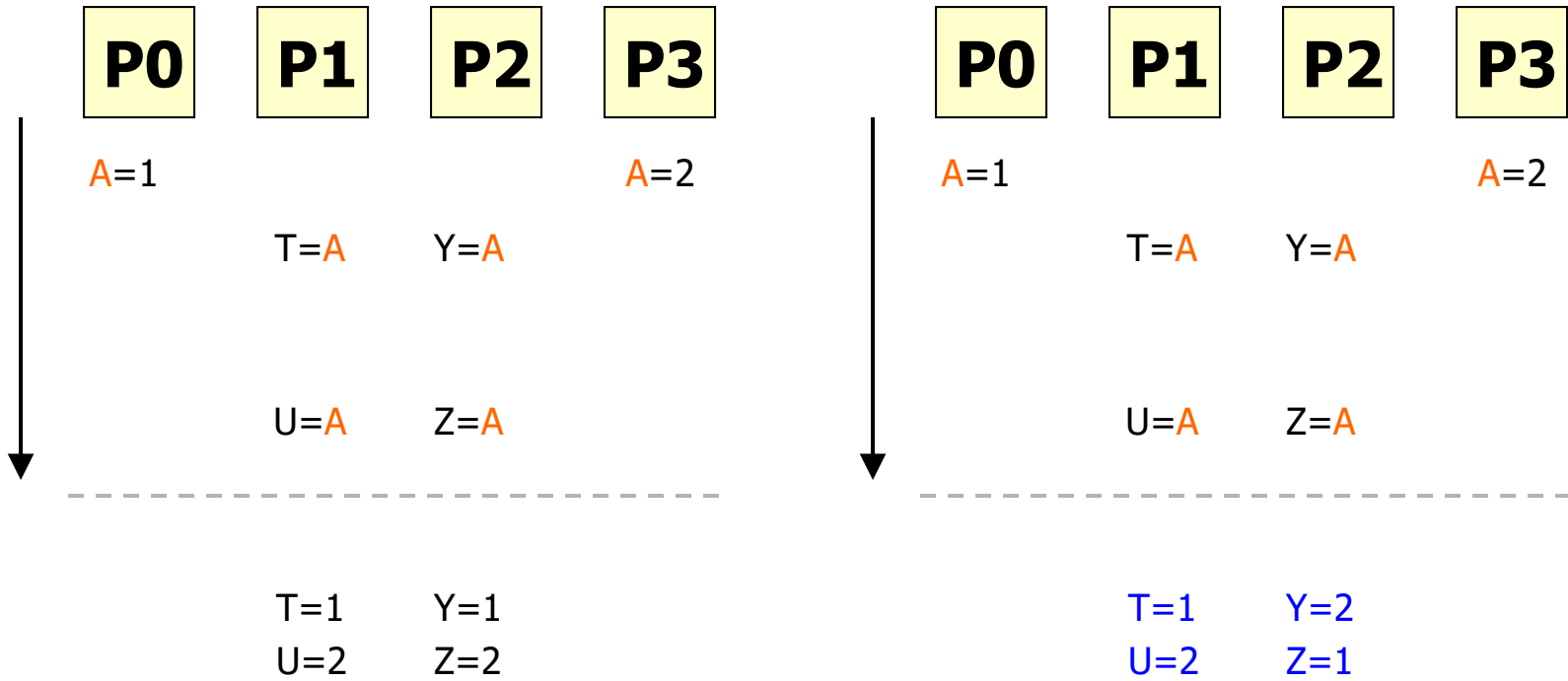
- Programmers anticipate certain memory ordering and program behavior
- Become very complex When
 - Running shared-memory programs
 - A processor supports out-of-order execution
- A memory consistency model specifies the legal ordering of memory events when several processors access the shared memory locations

Sequential Consistency (SC) [Leslie Lamport]



- An MP 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.
- Two properties
 - Program ordering
 - Write atomicity (**All writes to any location** should appear to all processors in the same order)
- Intuitive to programmers

SC Example



Maintain Program Ordering (SC)

Flag1 = Flag2 = 0

P1

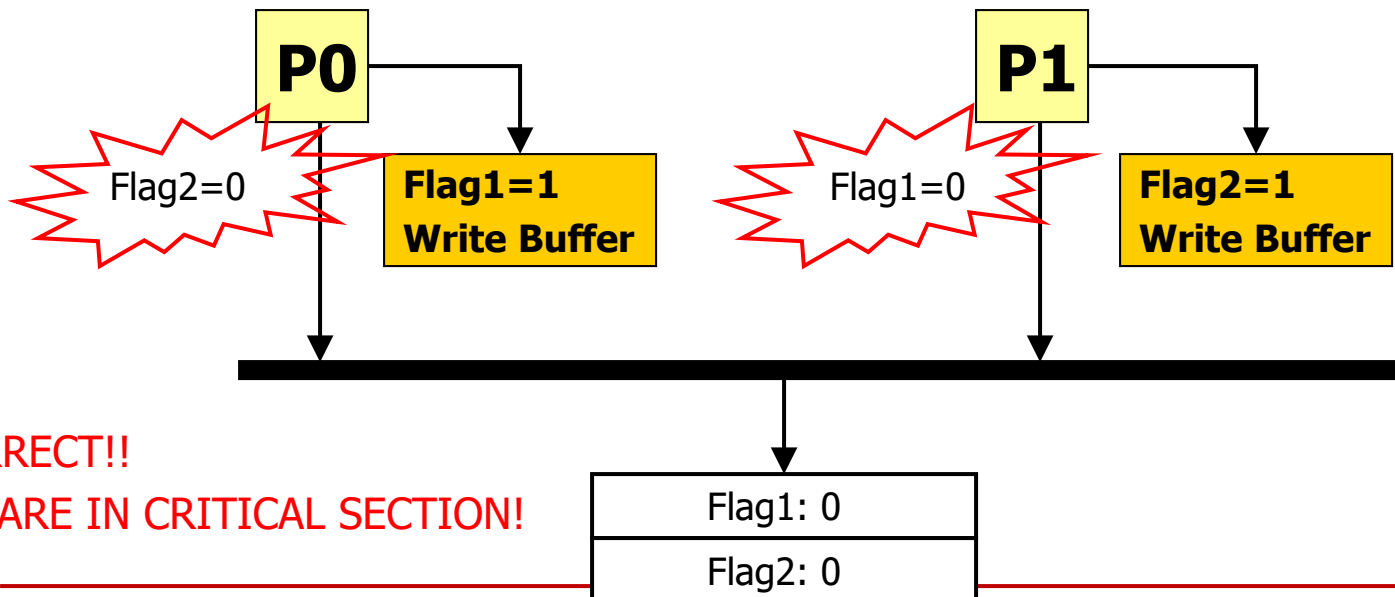
Flag1 = 1
if (Flag2 == 0)
enter Critical Section

P2

Flag2 = 1
if (Flag1 == 0)
enter Critical Section

- Dekker's algorithm
- Only one processor is allowed to enter the CS

Caveat: implementation fine with uni-processor, but violate the ordering of the above



Atomic and Instantaneous Update (SC)

A = B = 0

P1

A = 1

P2

if (A==1)
 B =1

P3

if (B==1)
 R1=A

- Update (of A) must take place atomically to all processors
- A read cannot return the value of another processor's write until the write is made visible by "all" processors

Atomic and Instantaneous Update (SC)

$A = B = 0$

P1

$A = 1$

P2

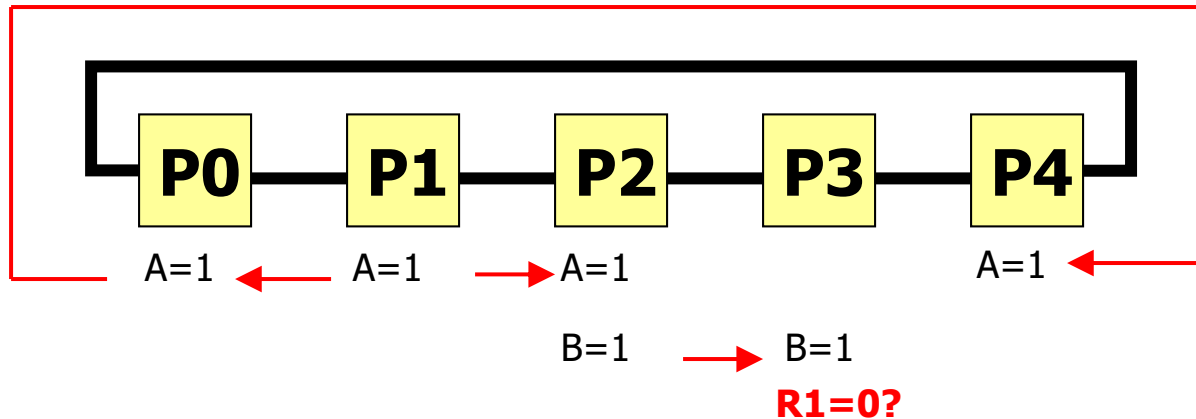
if ($A == 1$)
 $B = 1$

P3

if ($B == 1$)
 $R1 = A$

- Update (of A) must take place atomically to all processors
- A read cannot return the value of another processor's write until the write is made visible by "all" processors

Caveat when an update is not atomic to all ...



Relaxed Memory Models

- Relax program order requirement (its applies only to operation pairs accessing different locations)
 - Load bypass store
 - Load bypass load
 - Store bypass store
 - Store bypass load
- Relax write atomicity requirement?
 - Read others' write early
 - Allow a read to return the value of another processor's write before all cached copies of the accessed location receive the invalidation or update messages generated by the write
 - Read own write early
 - Allow a read to return the value of its own previous write before the write is made visible to other processors

Relaxed Memory Models

Relaxation	W $\rightarrow$ R Order	W $\rightarrow$ W Order	R $\rightarrow$ RW Order	Read Others' Write Early	Read Own Write Early	Safety net
SC [16]					✓	
IBM 370 [14]	✓					serialization instructions
TSO [20]	✓				✓	RMW
PC [13, 12]	✓			✓	✓	RMW
PSO [20]	✓	✓			✓	RMW, STBAR
WO [5]	✓	✓	✓		✓	synchronization
RCsc [13, 12]	✓	✓	✓		✓	release, acquire, nsync, RMW
RCpc [13, 12]	✓	✓	✓	✓	✓	release, acquire, nsync, RMW
Alpha [19]	✓	✓	✓		✓	MB, WMB
RMO [21]	✓	✓	✓		✓	various MEMBAR's
PowerPC [17, 4]	✓	✓	✓	✓	✓	SYNC

Figure 8: Simple categorization of relaxed models. A ✓ indicates that the corresponding relaxation is allowed by straightforward implementations of the corresponding model. It also indicates that the relaxation can be detected by the programmer (by affecting the results of the program) except for the following cases. The “Read Own Write Early” relaxation is not detectable with the SC, WO, Alpha, and PowerPC models. The “Read Others’ Write Early” relaxation is possible and detectable with complex implementations of RCsc.

Relaxation	Example Commercial Systems Providing the Relaxation
W $\rightarrow$ R Order	AlphaServer 8200/8400, Cray T3D, Sequent Balance, SparcCenter1000/2000
W $\rightarrow$ W Order	AlphaServer 8200/8400, Cray T3D
R $\rightarrow$ RW Order	AlphaServer 8200/8400, Cray T3D
Read Others’ Write Early	Cray T3D
Read Own Write Early	AlphaServer 8200/8400, Cray T3D, SparcCenter1000/2000

Figure 9: Some commercial systems that relax sequential consistency.

Relaxed Consistency

- Processor Consistency
 - Used in P6
 - Write visibility could be in different orders of different processors (not guarantee write atomicity)
 - Allow loads to bypass independent stores in each individual processor
 - To achieve SC, **explicit synchronization operations need to be substituted or inserted**
 - Read-modify-write instructions
 - Memory fence instructions

Processor Consistency

Load bypassing Stores

F1=1

A=1

R1=**A**

R2=**F2**

F2=1

A=2

R3=**A**

R4=**F1**

R1=1; R3=2; **R2=0; R4=0** is a possible outcome

Processor Consistency

Flag1 = Flag2 = 0

P1

Flag1 = 1
if (Flag2 == 0)
enter Critical Section

P2

Flag2 = 1
if (Flag1 == 0)
enter Critical Section

- Allow load bypassing store to a different address
- Unlike SC, cannot guarantee mutual exclusion in the critical section

Processor Consistency

$A = B = 0$

P1

P2

P3

$A = 1$

if ($A == 1$)

$B = 1$

if ($B == 1$)

$R1 = A$

$B=1; R1=0$ is a possible outcome
since PC allows $A=1$ to be visible in P2 prior to P3

Backup Slides

Dekker's Algorithm

- Dekker's algorithm guarantees mutual exclusion, freedom from deadlock, and freedom from starvation

Pseudocode

```
flag[0] := false
flag[1] := false
turn := 0 // or 1
```

p0:

```
flag[0] := true
while flag[1] = true {
  if turn ≠ 0 {
    flag[0] := false
    while turn ≠ 0 {
    }
    flag[0] := true
  }
}

// critical section
...
turn := 1
flag[0] := false
// remainder section
```

p1:

```
flag[1] := true
while flag[0] = true {
  if turn ≠ 1 {
    flag[1] := false
    while turn ≠ 1 {
    }
    flag[1] := true
  }
}

// critical section
...
turn := 0
flag[1] := false
// remainder section
```