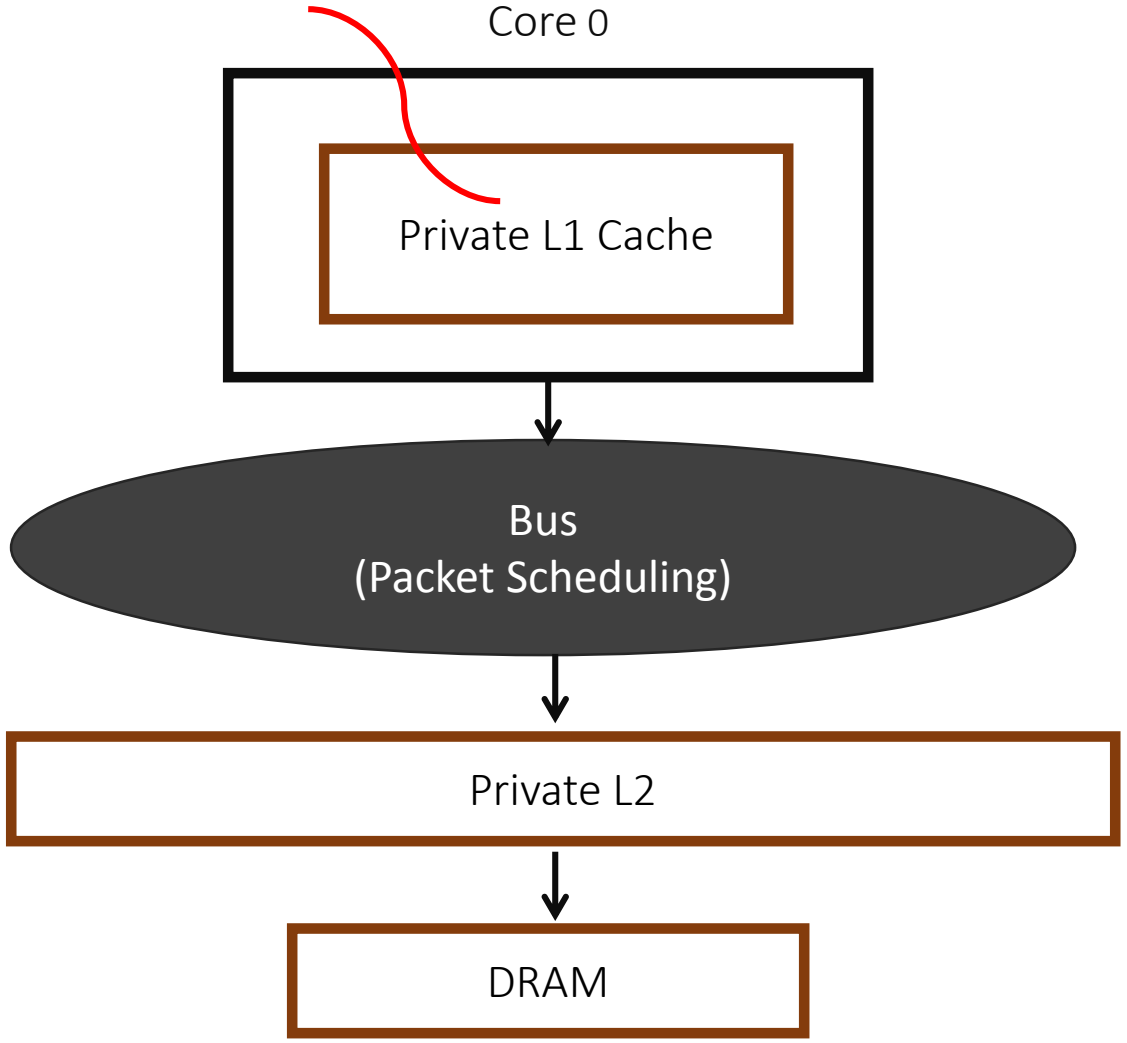
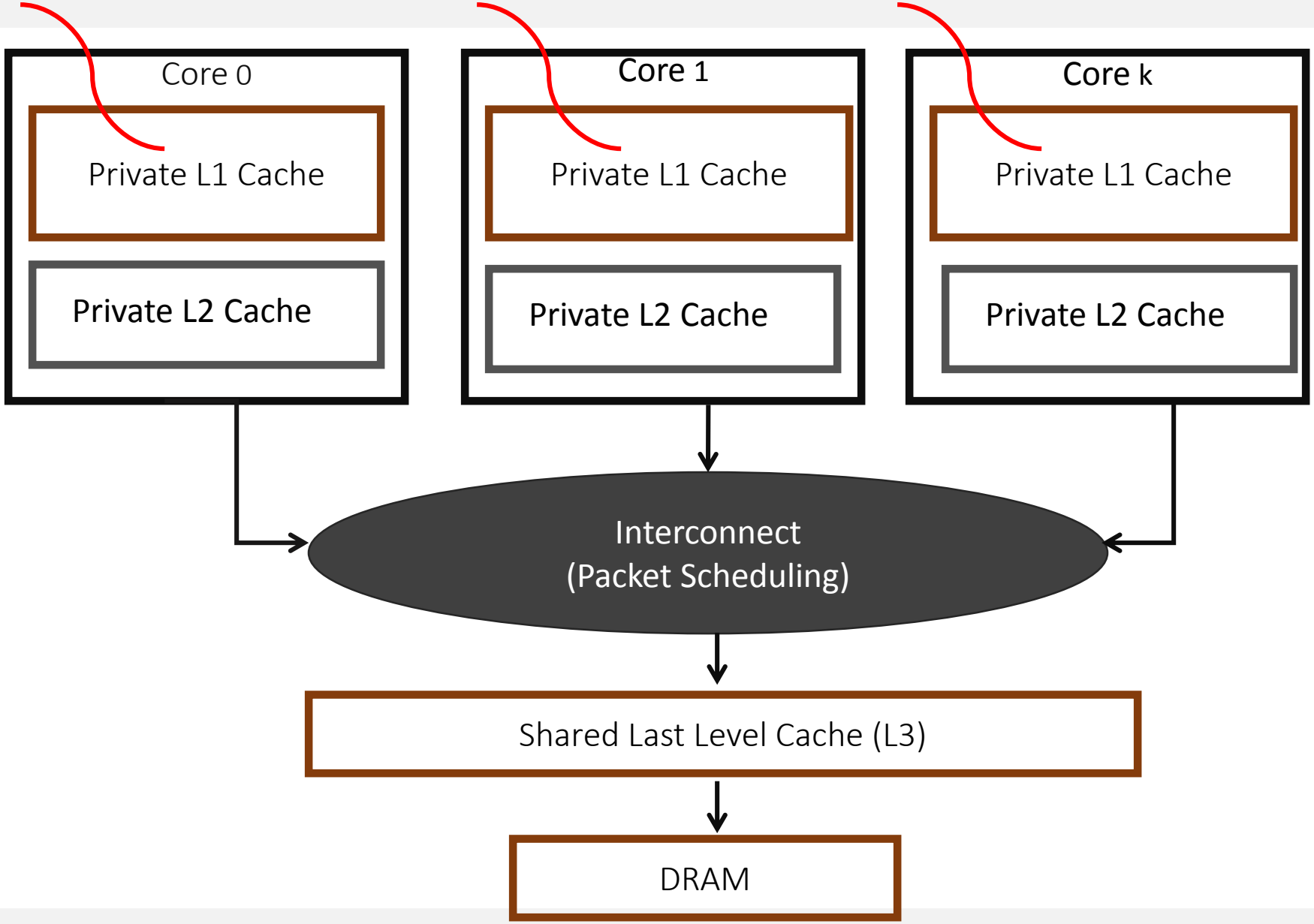
Lecture-22 (Cache Coherence Protocols) CS422-Spring 2018

Biswa@CSE-IITK

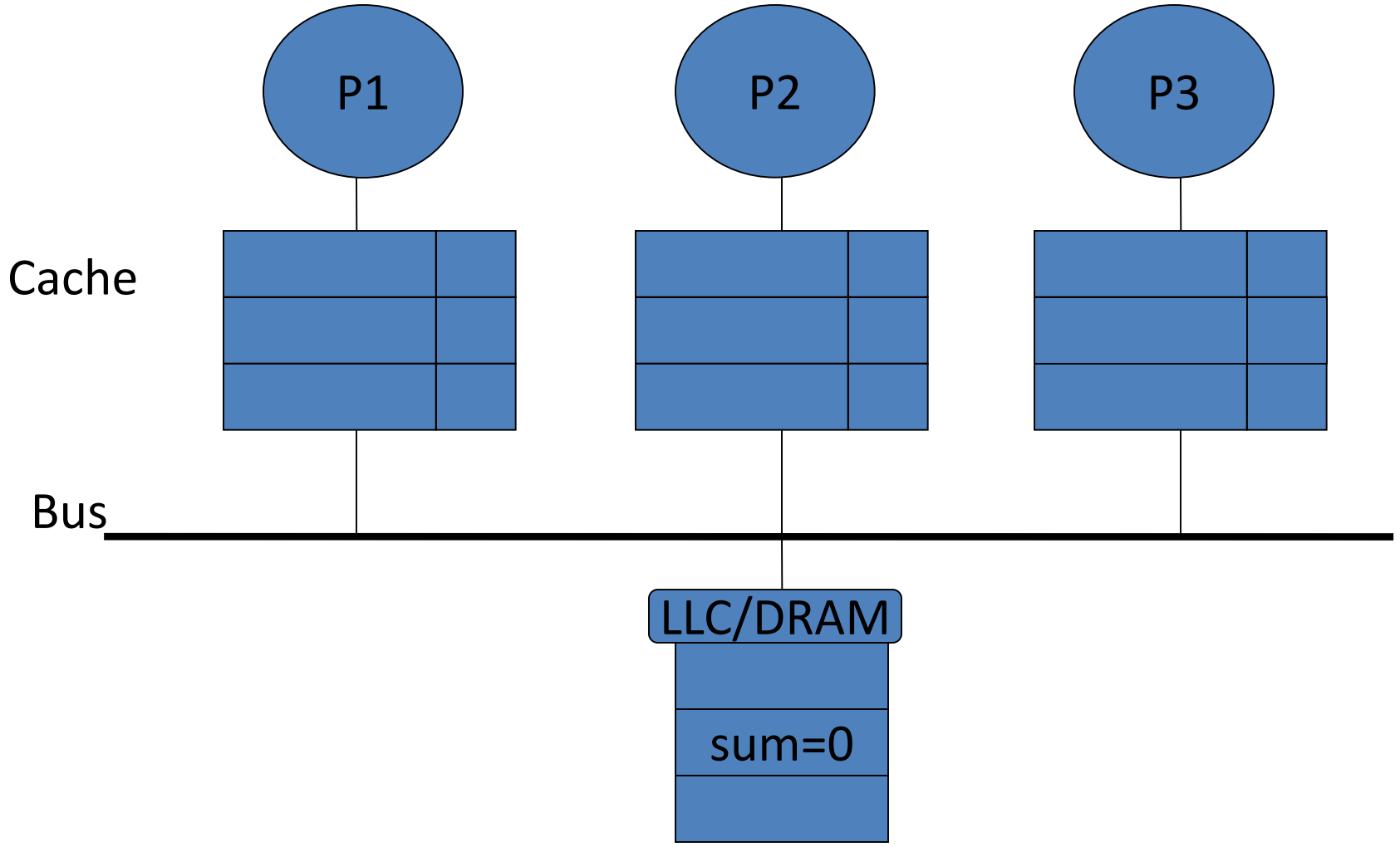
Single Core


Core 0
Private L1 Cache
Bus
(Packet Scheduling)
Private L2
DRAM

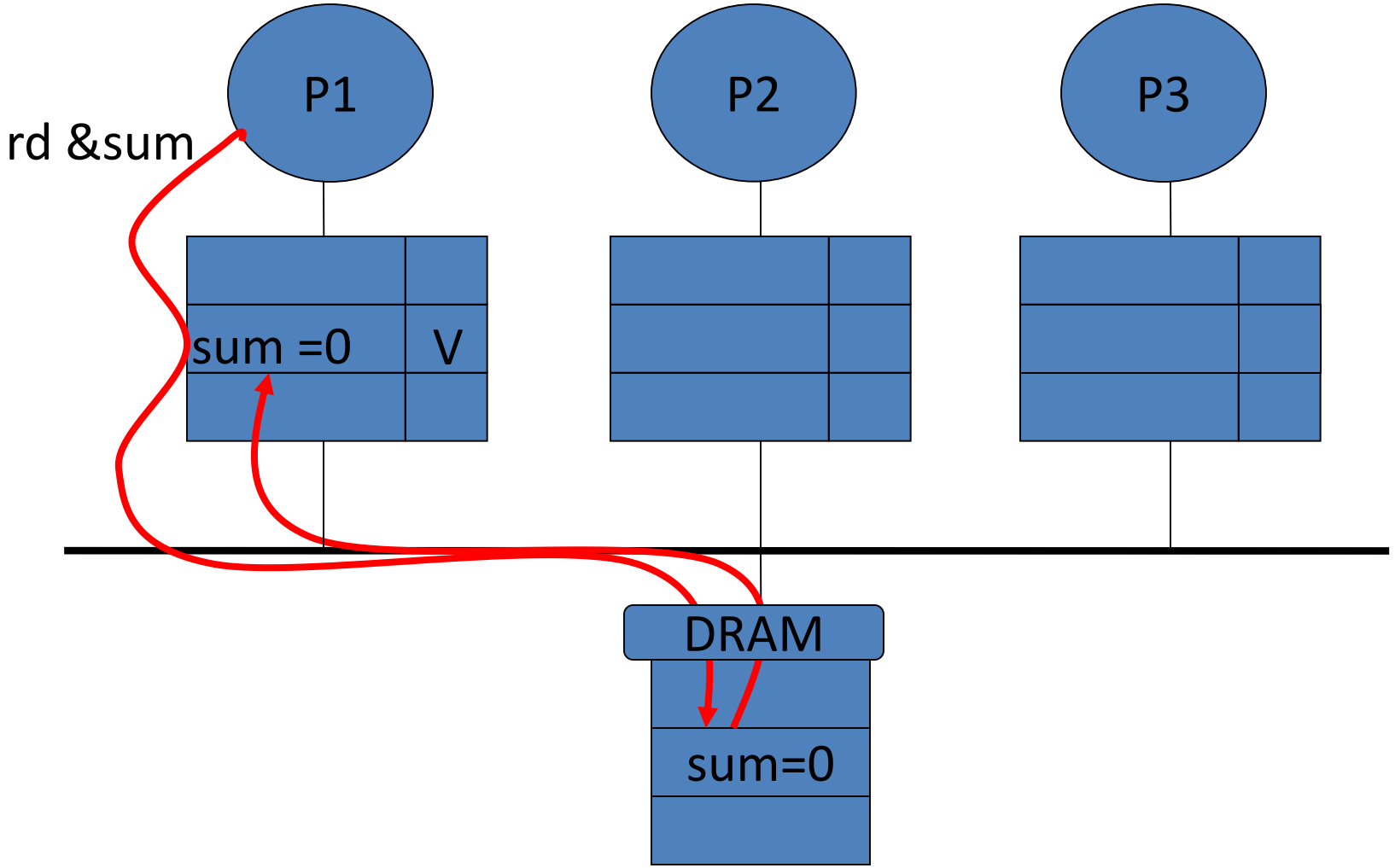
Multicore


Core 0
Private L1 Cache
Private L2 Cache
Core 1
Private L1 Cache
Private L2 Cache
Core k
Private L1 Cache
Private L2 Cache
Interconnect
(Packet Scheduling)
Shared Last Level Cache (L3)
DRAM

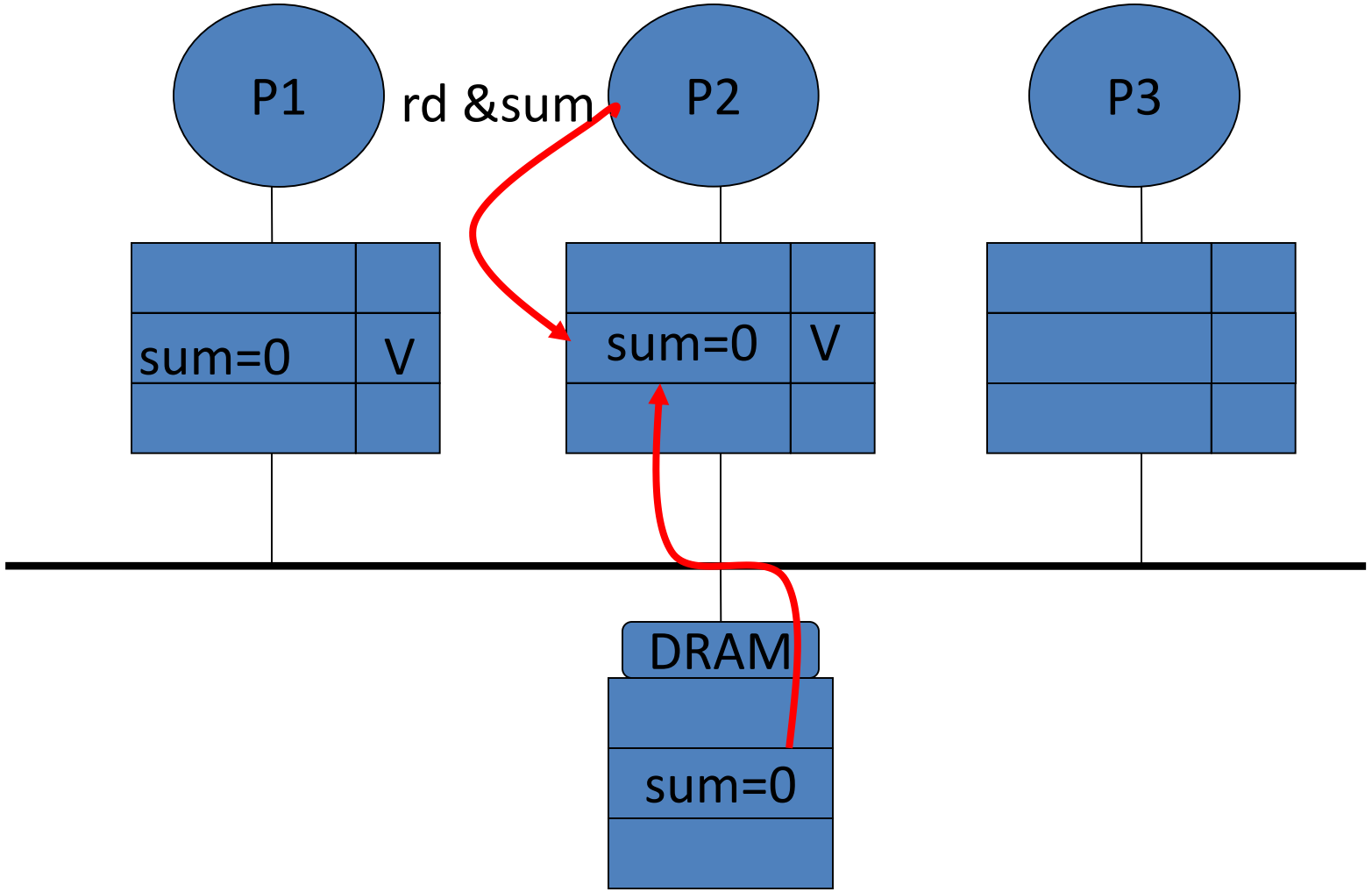
Cache Coherence



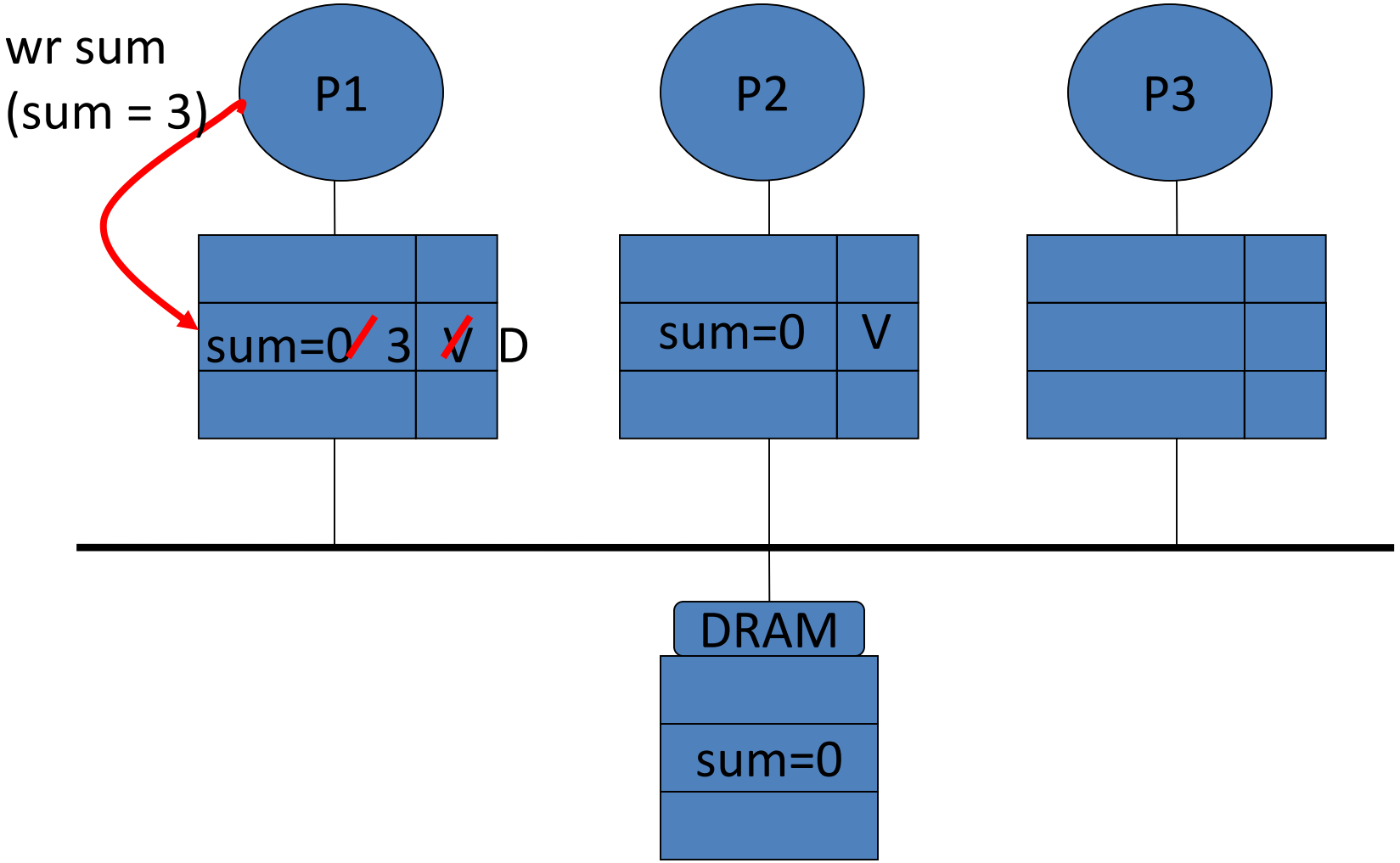
Cache Coherence



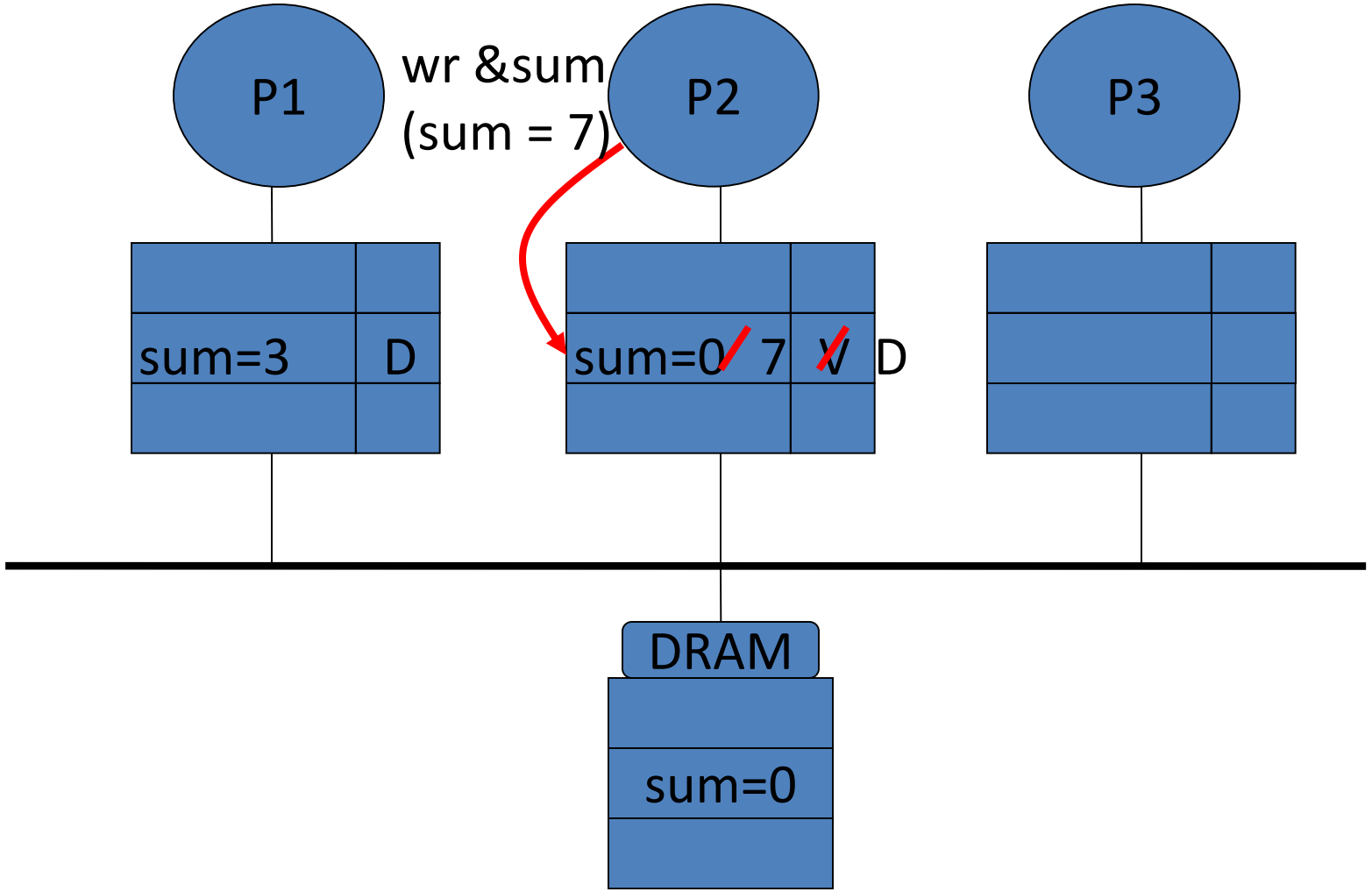
Cache Coherence



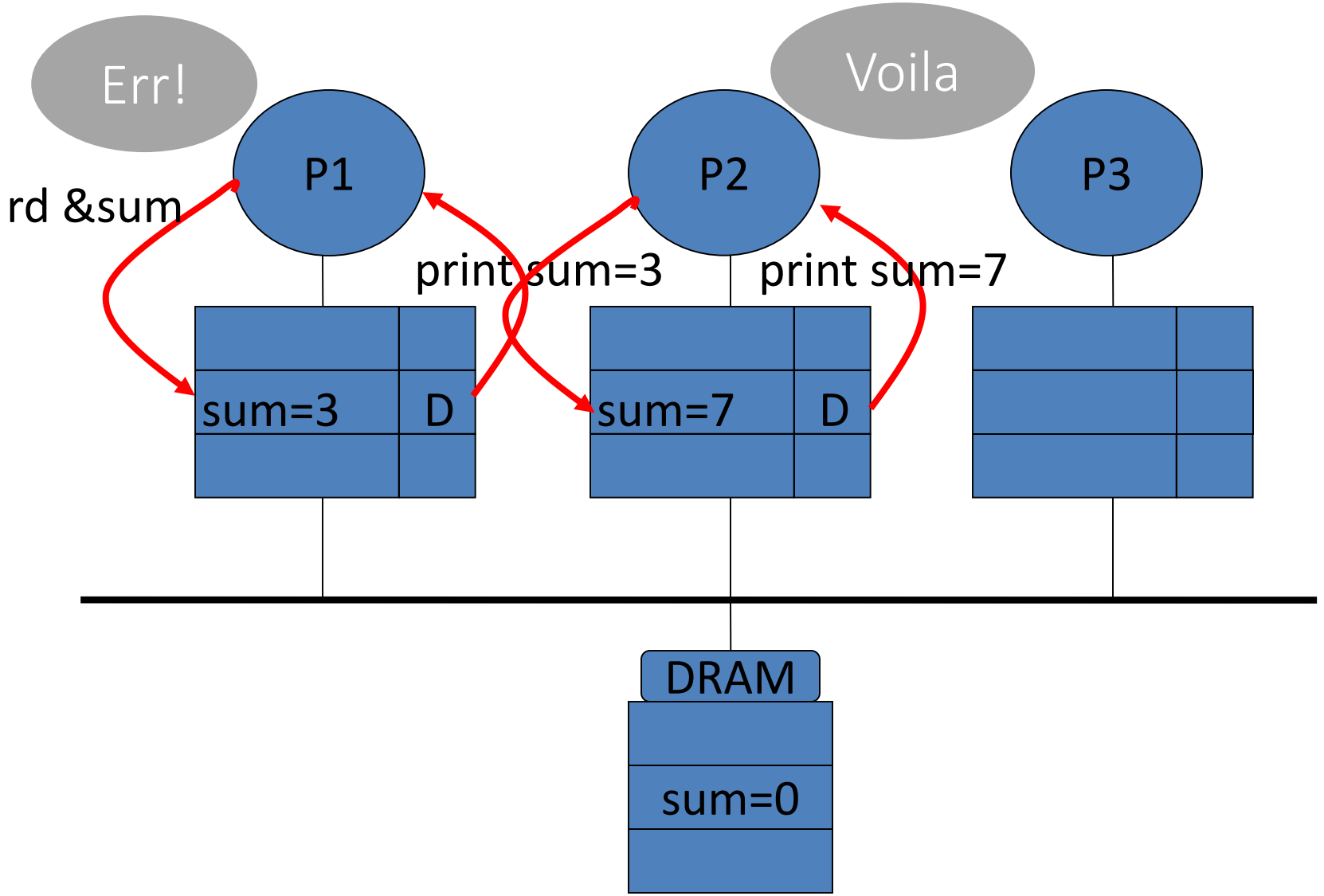
Cache Coherence



Cache Coherence



Cache Coherence



The Problem

If multiple cores cache the same block, how do they ensure they all see a consistent state?

Solution:

1. **Write Propagation**: All writes eventually become visible to other cores.
2. **Write Serialization**: All cores see write to a cache line in same order.

Need a protocol to ensure (1) and (2) called *cache coherence protocol*

Non-solutions

Keeping caches coherent is software's responsibility

(+) Makes microarchitect's life (my) easier

(-) Makes average programmer's life (yours) much harder

Non-solutions

All caches are shared between all processors

+ No need for coherence

-- Shared cache becomes the bandwidth bottleneck

-- Scalability : 100 monkeys – 1 banana

Who Is Responsible?

(1) ISA - What if the ISA provided a cache flush/invalidate instruction?

Such as - FLUSH-LOCAL, FLUSH-GLOBAL, FLUSH-CACHE

(2) Hardware - Invalidate all other copies of block say A when a core writes to it

Simplifies yours job

Invalidate vs Update

Invalidate –

- Write to a cache line, and simultaneously broadcast invalidation of address to all others
- Other cores clear/invalidate their cache lines

Update –

- Write to a cache line, and simultaneously broadcast written data to all others
- Other cores update their caches if data was present

MSI Coherence Protocol

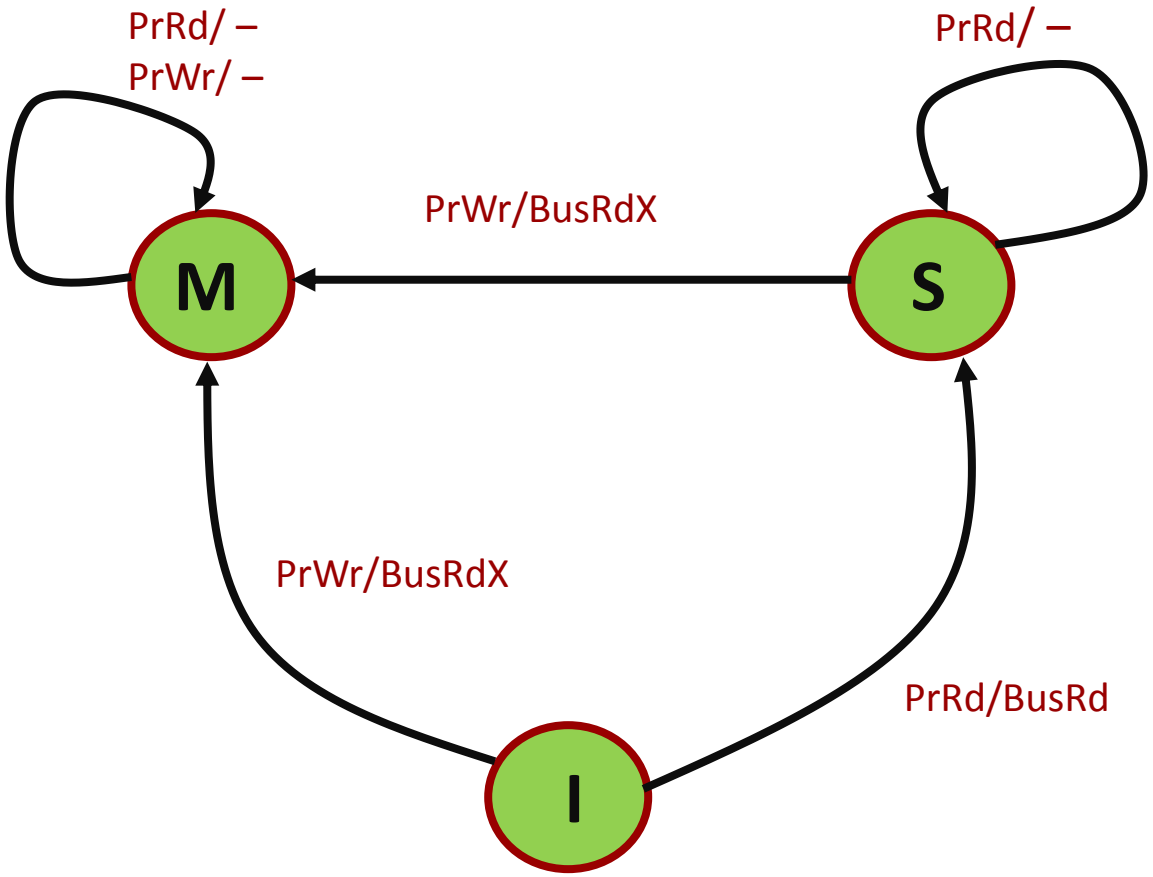
Extend single valid bit per line to three states:

- **M**(odified): only one cache, memory not updated.
- **S**(hared): one or more caches, and memory copy is up-to-date
- **I**(nvalid): not present

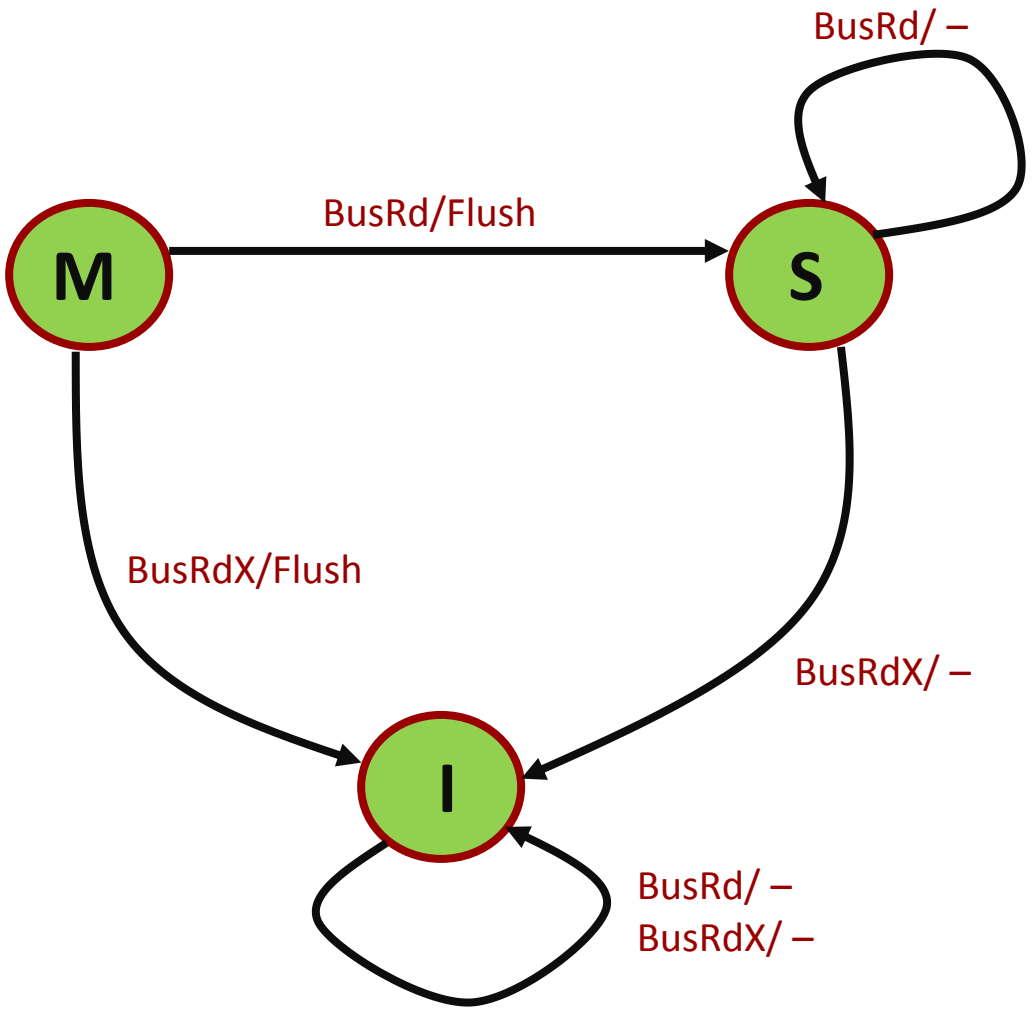
Read - *Read* request on bus, transitions to **S**

Write - *ReadEx* request, transitions to **M** state

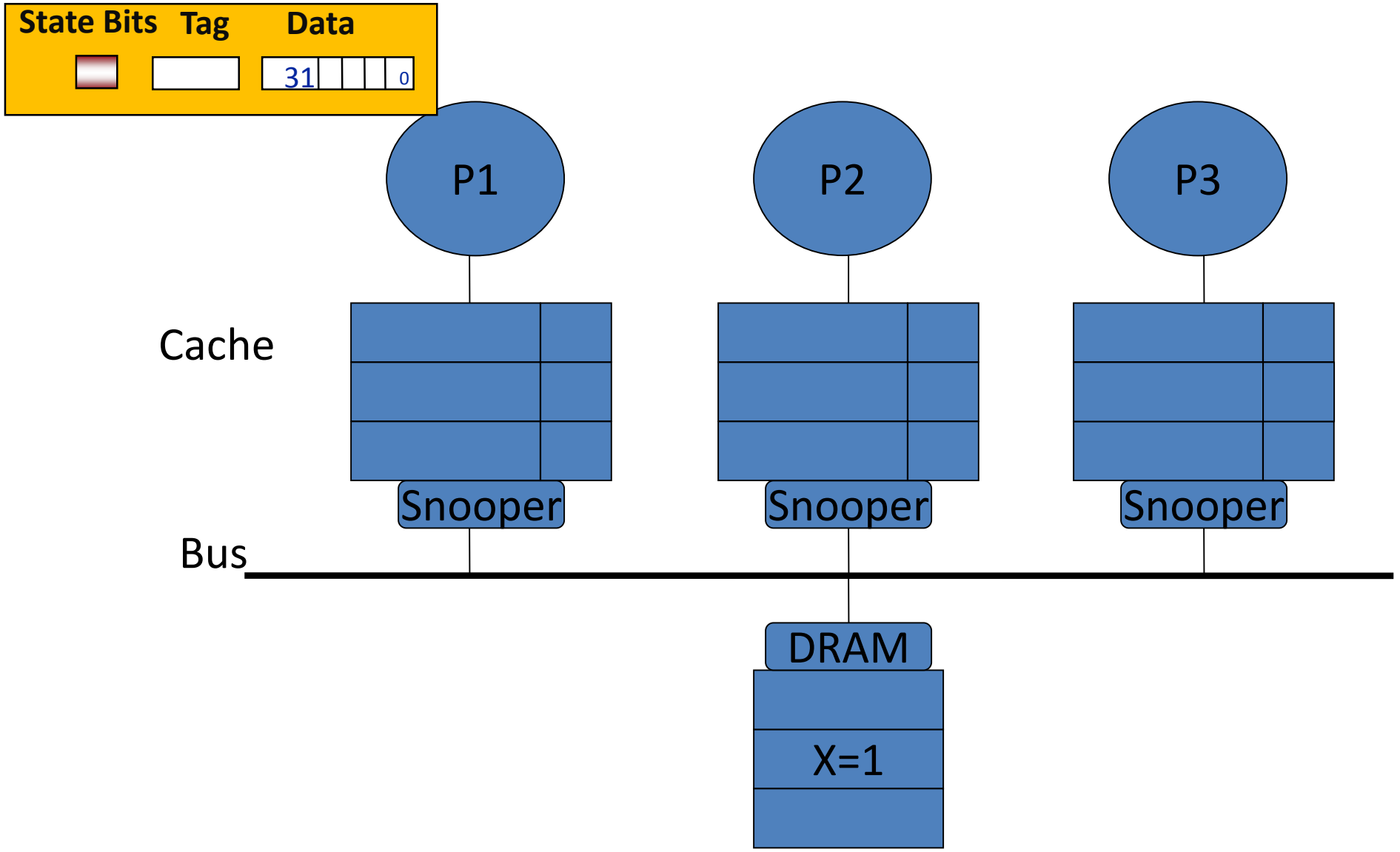
State Transitions – Core Side



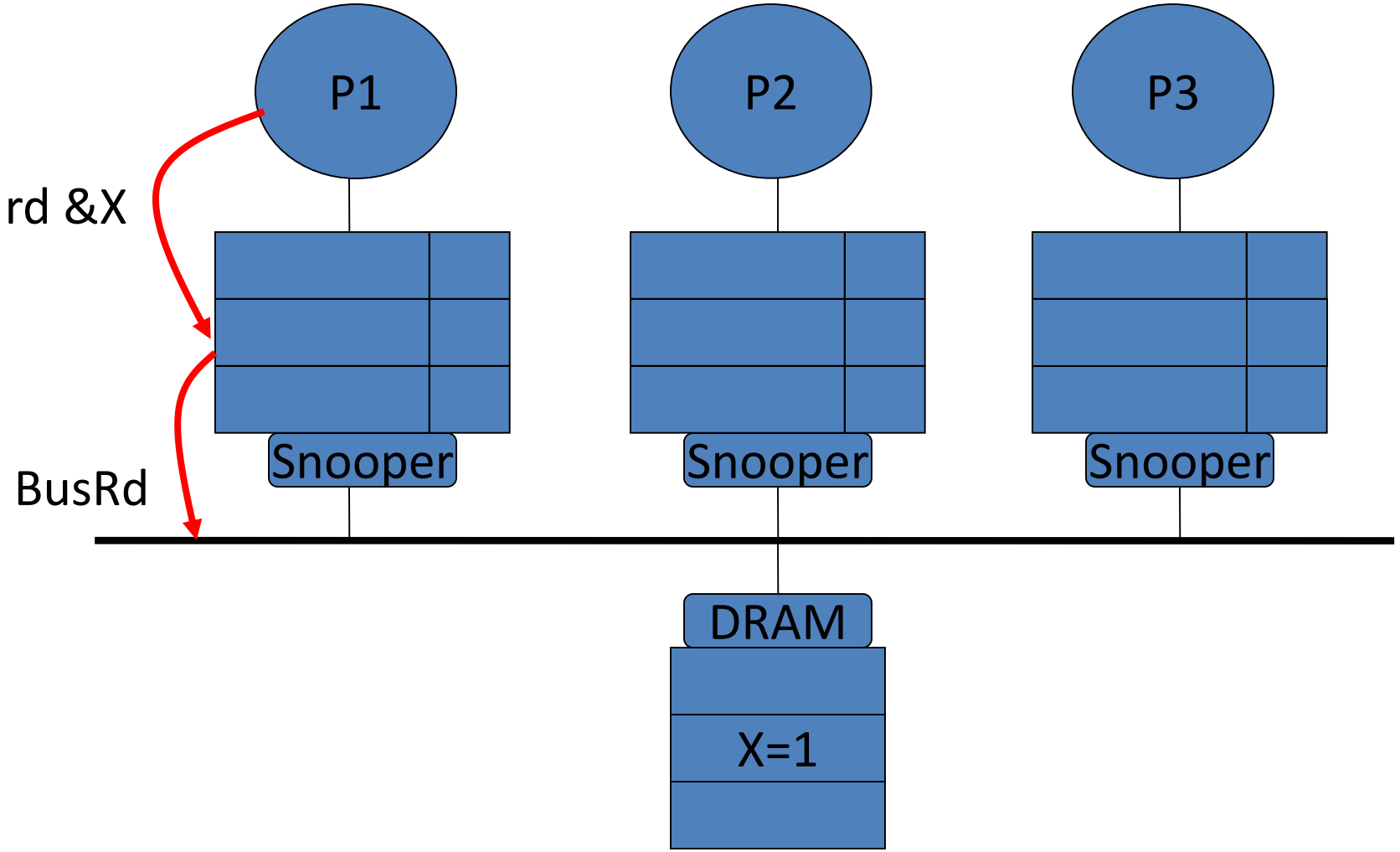
Bus Side



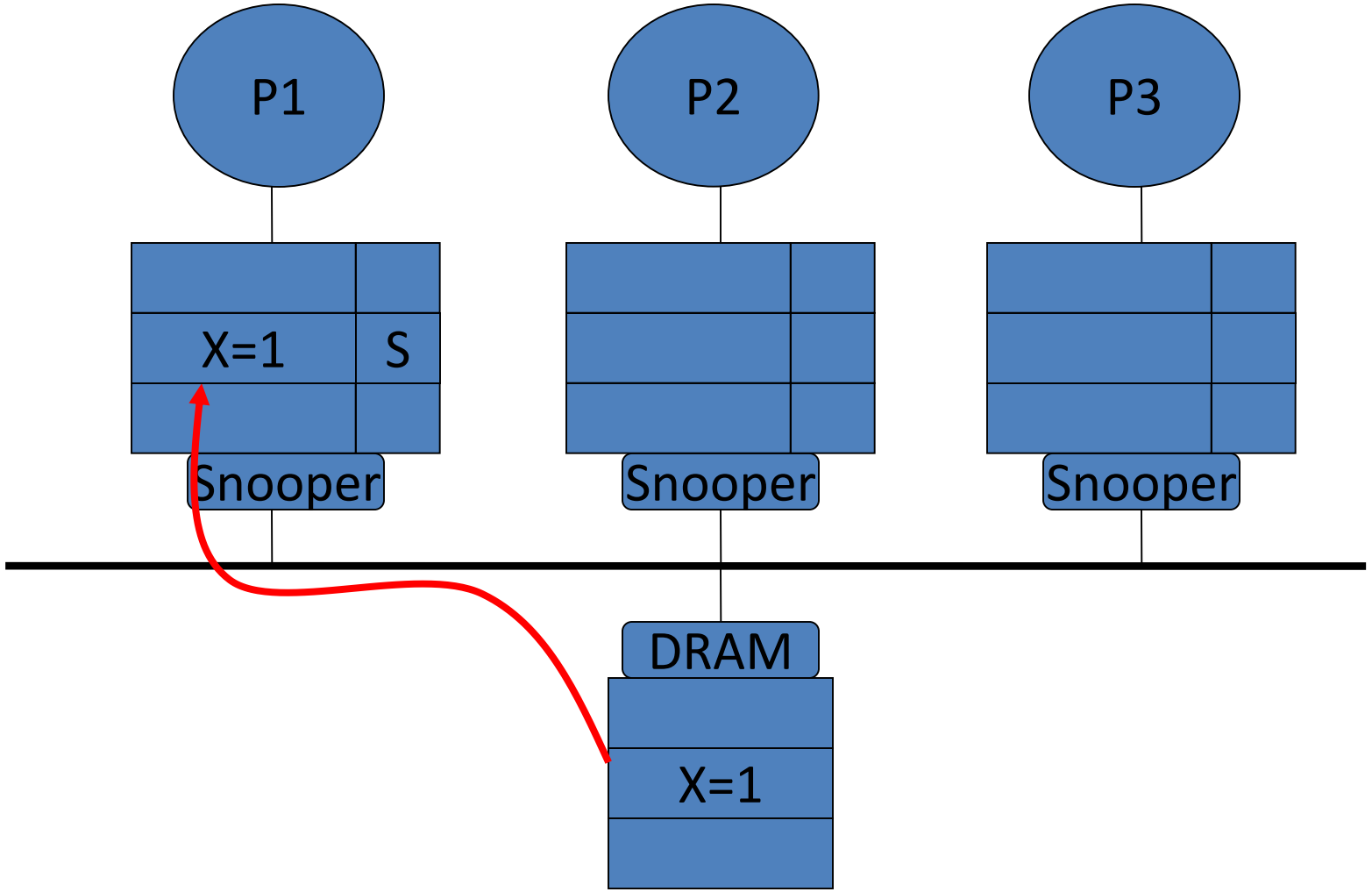
MSI


State Bits
Tag
Data
31
0
P1
P2
P3
Cache
Snooper
Snooper
Snooper
Bus
DRAM
X=1

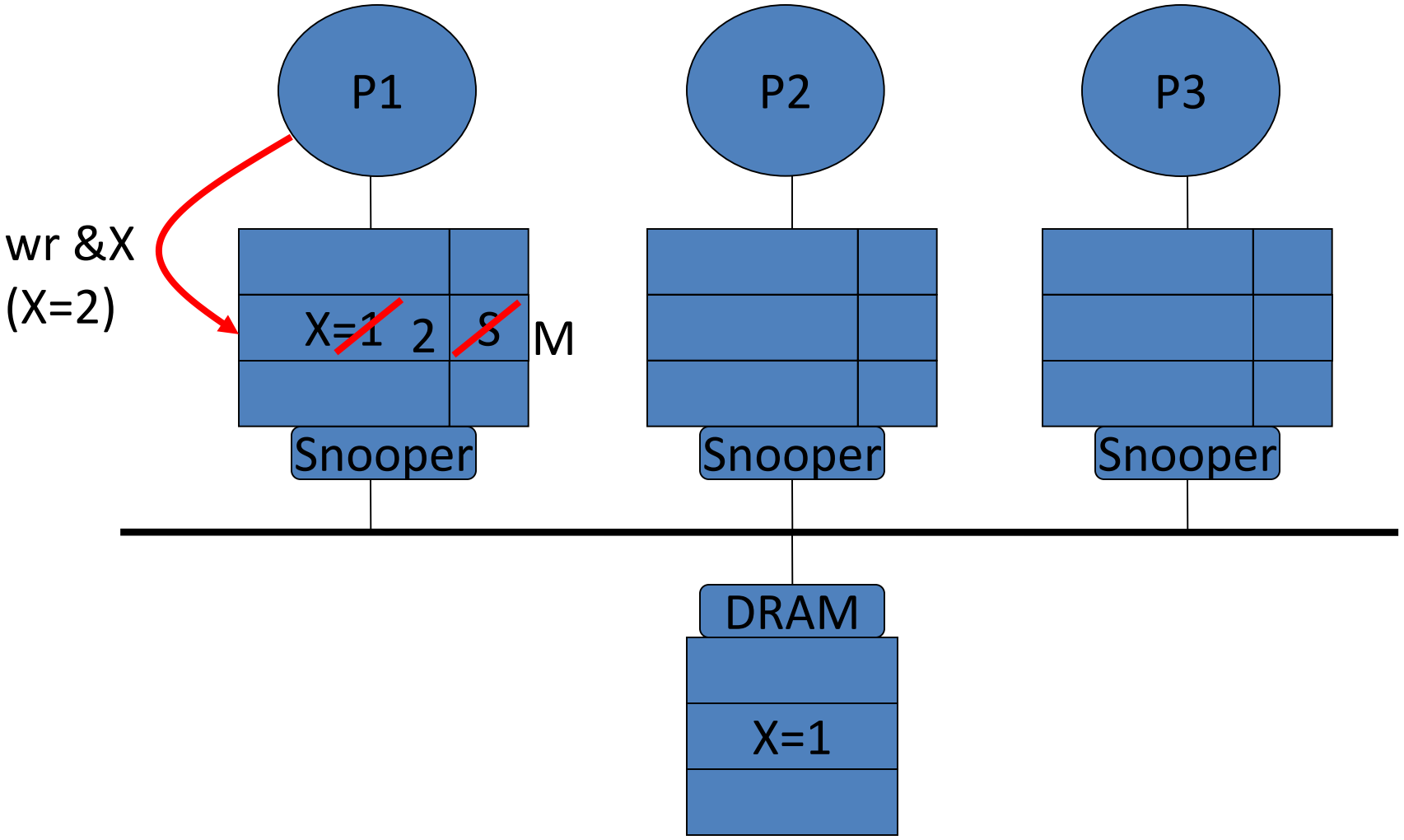
MSI



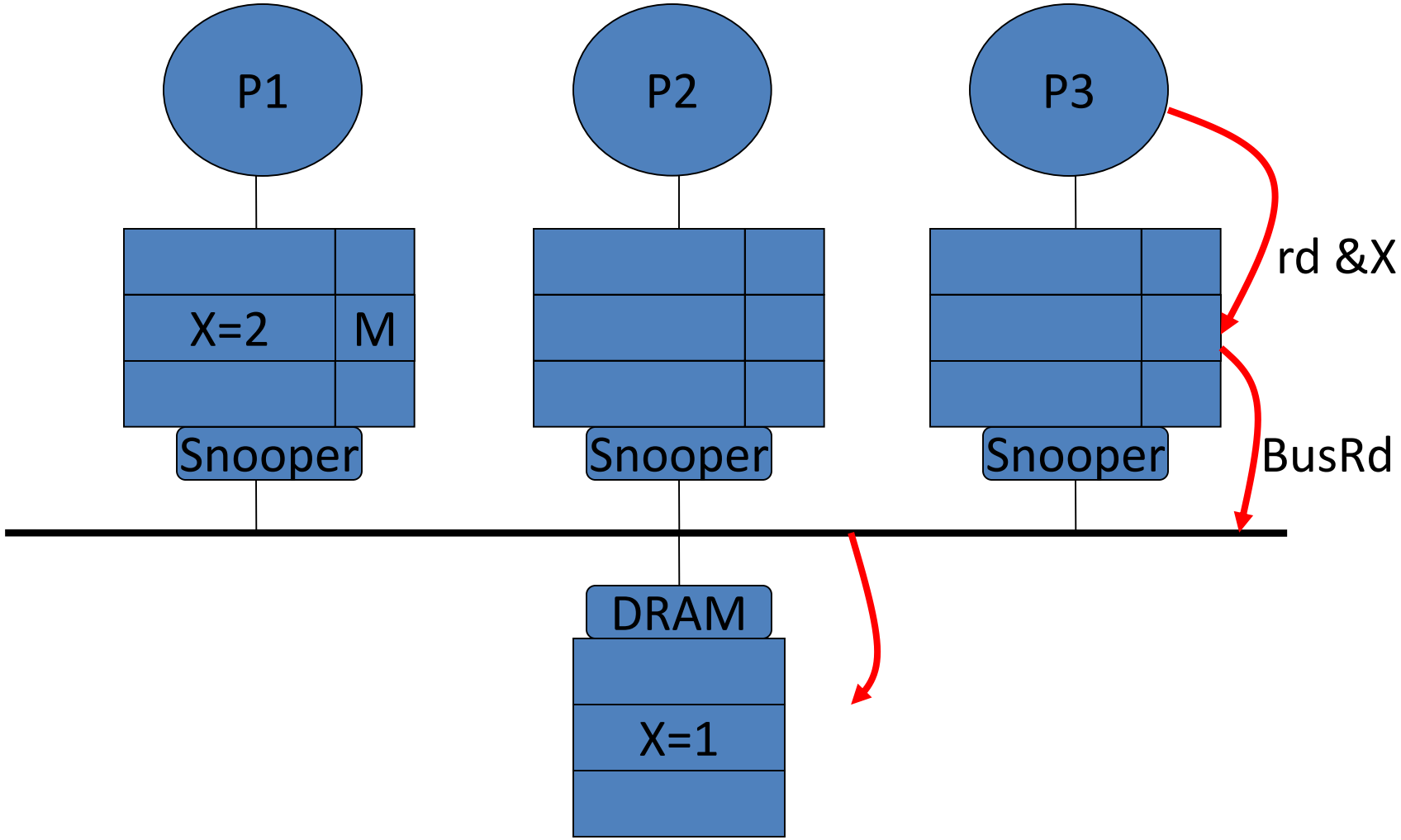
MSI



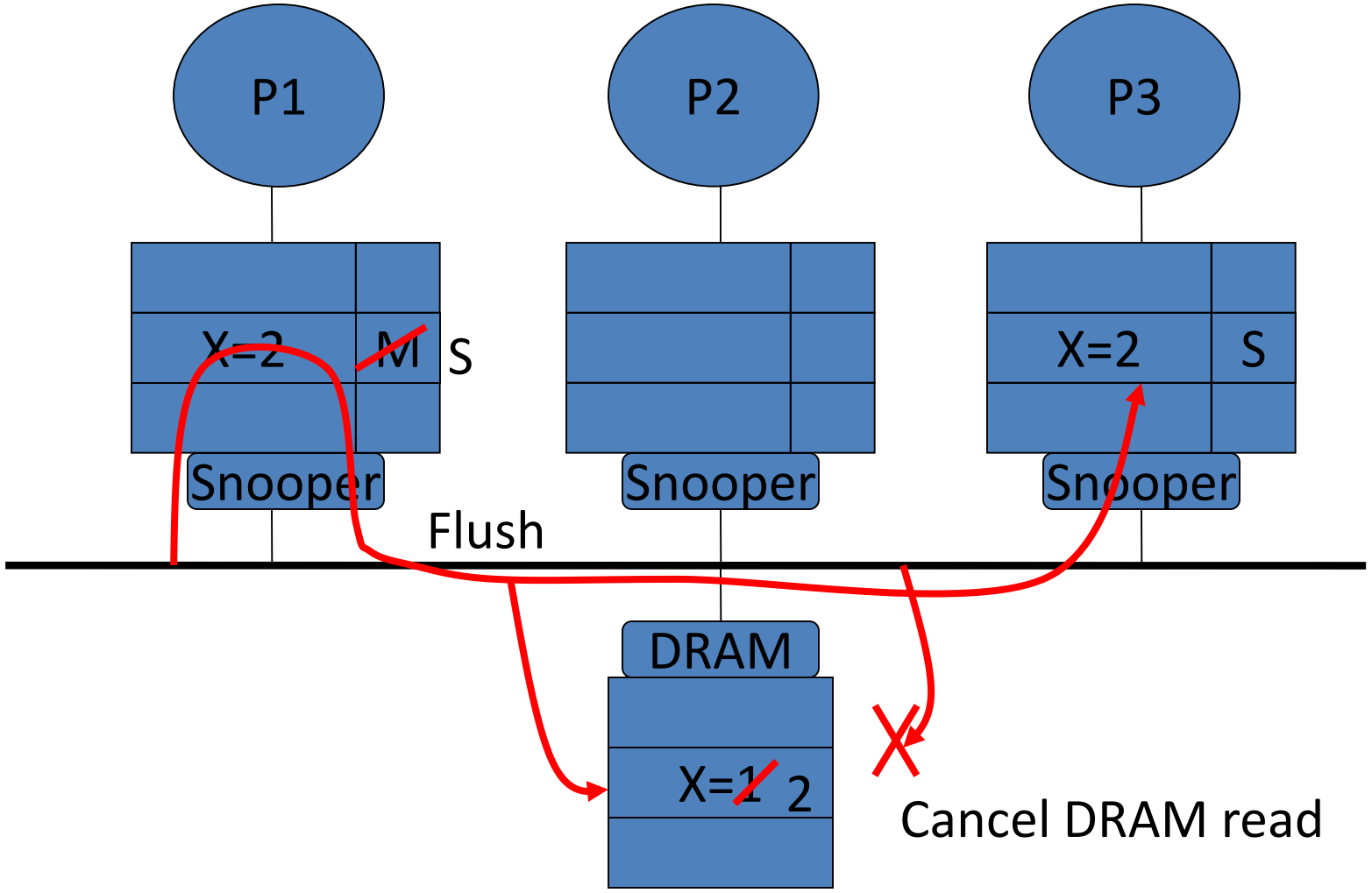
MSI



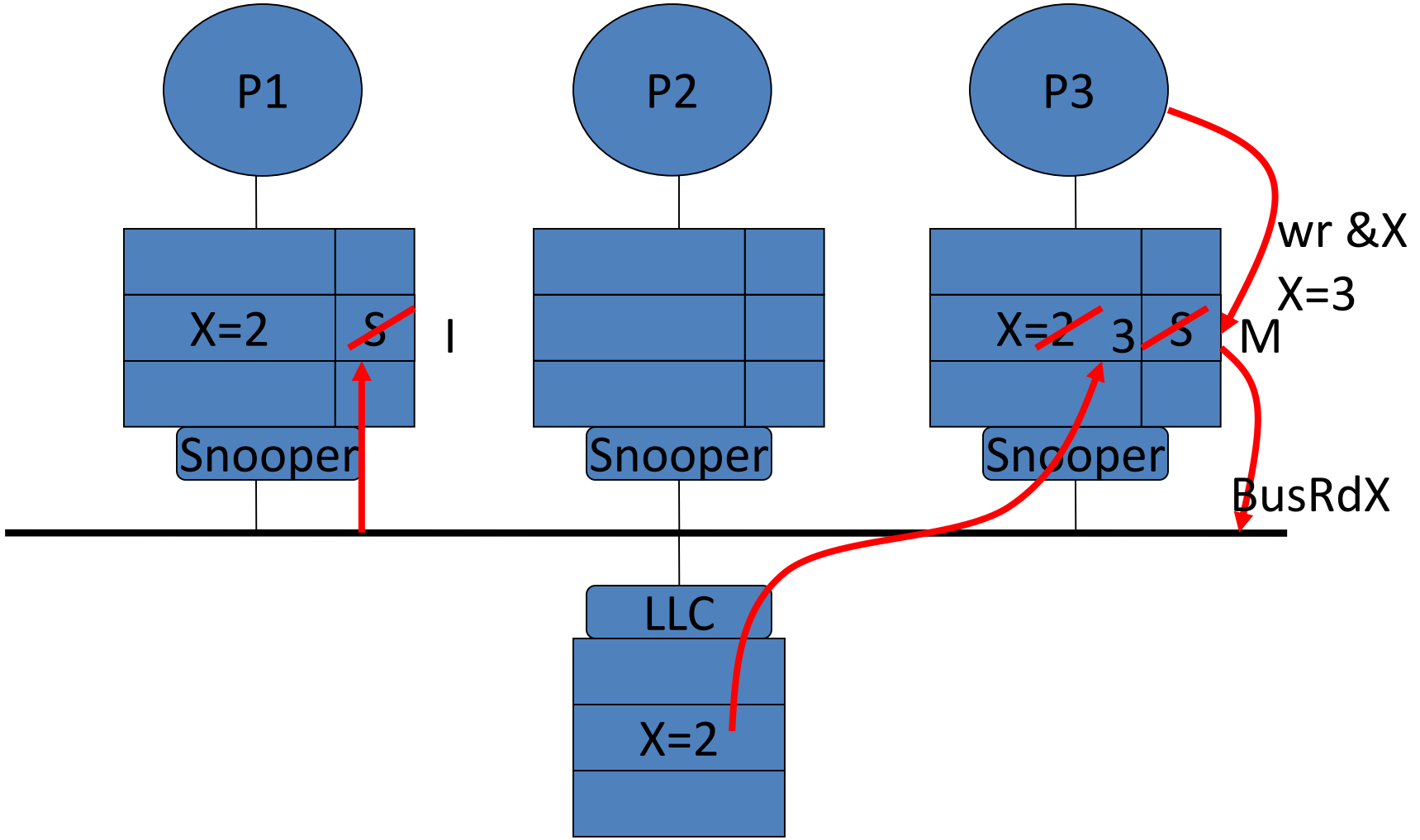
MSI



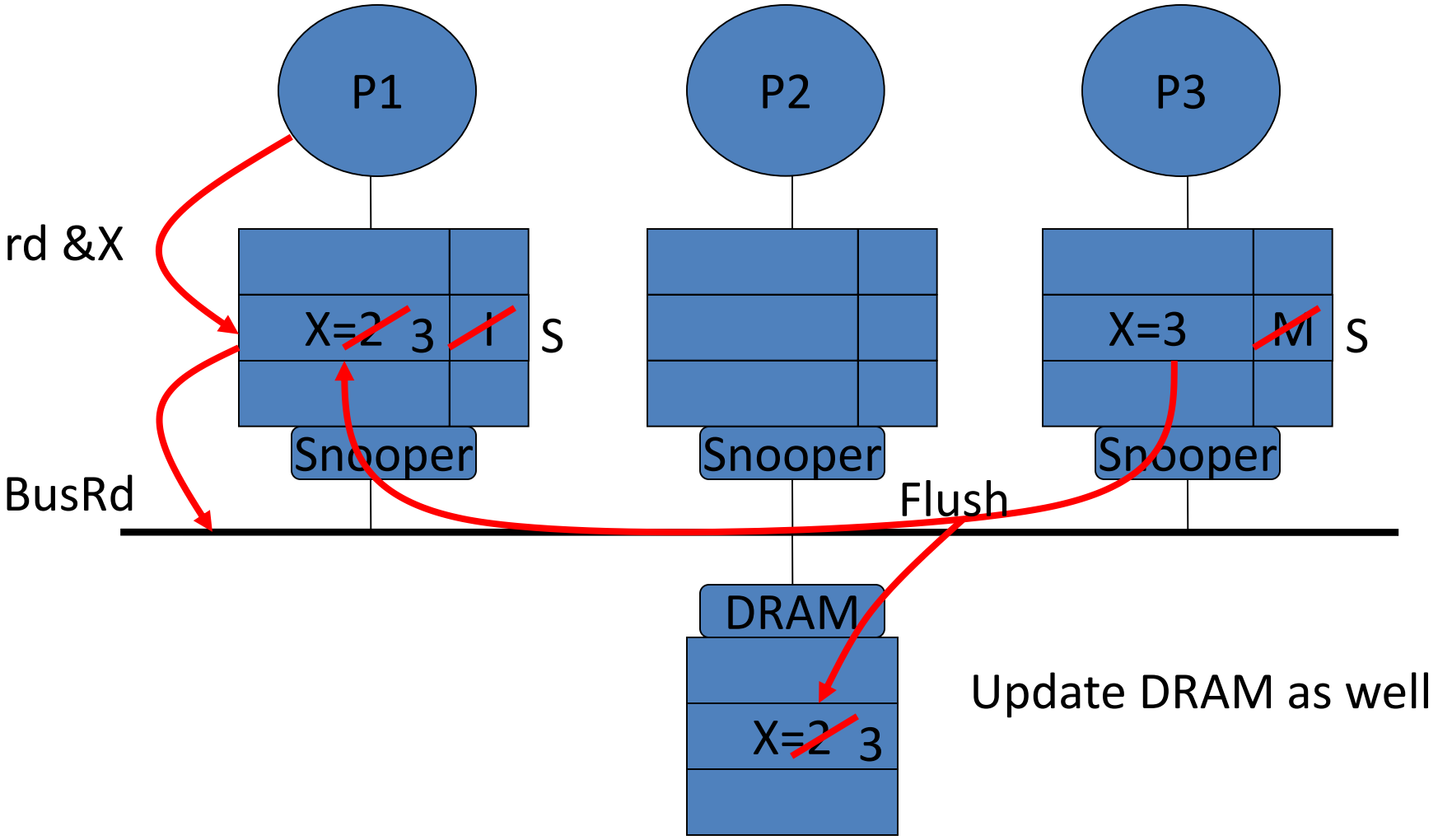
MSI


P1
P2
P3
X=2
M
S
X=2
S
Snooper
Snooper
Snooper
Flush
DRAM
X=1
2
Cancel DRAM read

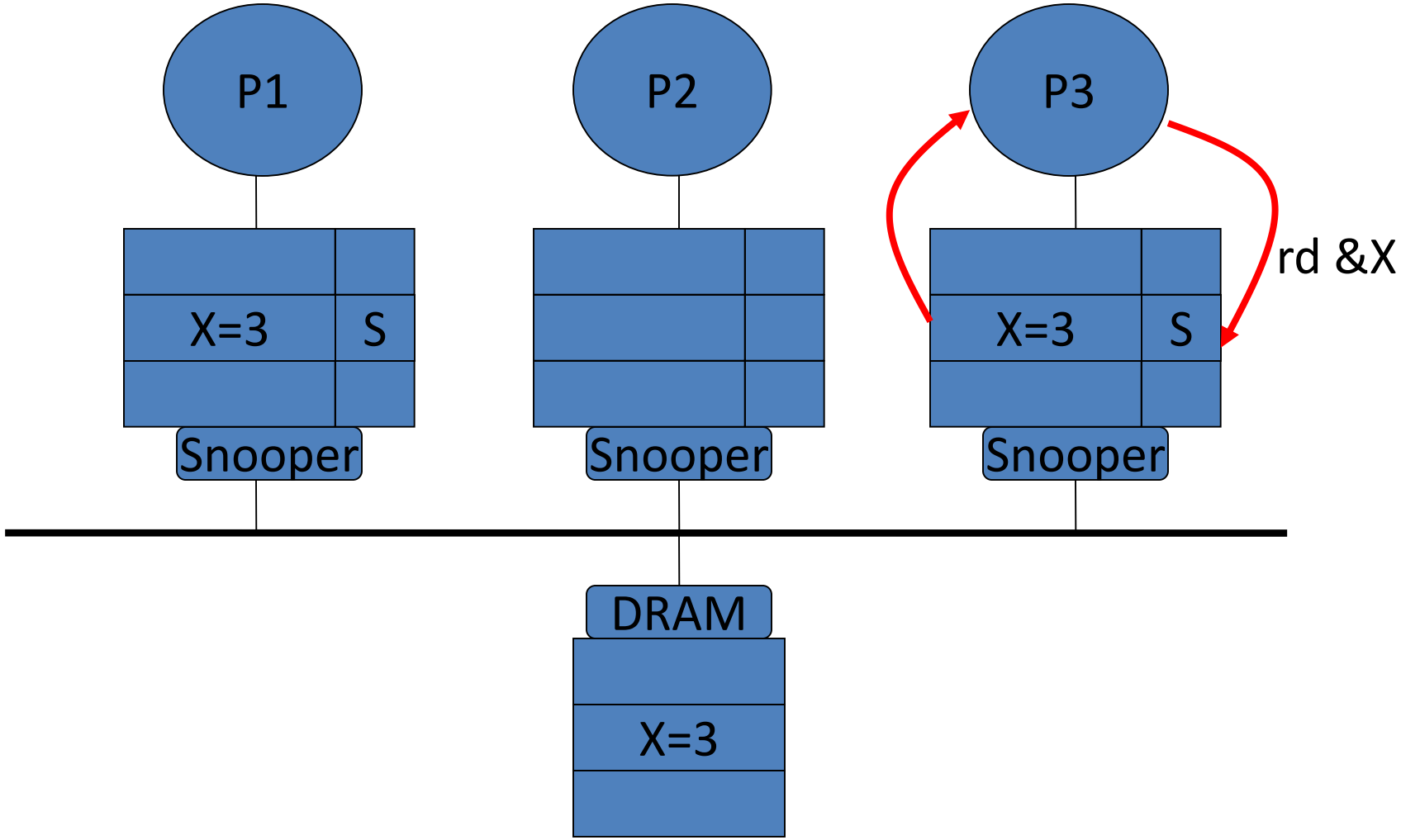
MSI


P1
P2
P3
X=2
S
I
X=2
3
S
M
wr &X
X=3
Snooper
Snooper
Snooper
BusRdX
LLC
X=2

MSI


P1
P2
P3
rd &X
X=2 3
I
S
X=3
M
S
Snooper
Snooper
Snooper
BusRd
Flush
DRAM
X=2 3
Update DRAM as well

MSI



MSI

Proc Action	State P1	State P2	State P3	Bus Action	Data From
R1	-	-	-	BusRd	Mem
W1	-	-	-	BusRdX	Mem
R3	-	-	-	BusRd	P1 cache
W3	-	-	-	BusRdX	Mem
R1	-	-	-	BusRd	P3 cache
R3	-	-	-	-	-
R2	-	-	-	BusRd	Mem

MSI

Proc Action	State P1	State P2	State P3	Bus Action	Data From
R1	S	-	-	BusRd	Mem
W1	M	-	-	BusRdX	Mem
R3	S	-	S	BusRd	P1 cache
W3	I	-	M	BusRdX	Mem
R1	S	-	S	BusRd	P3 cache
R3	S	-	S	-	-
R2	S	S	S	BusRd	Mem

1. **Write Propagation**: All writes eventually become visible to other cores.

   In MSI - (Through Invalidation and flush on subsequent BusRds)

2. **Write Serialization**: All cores see write to a cache line in same order.

   In MSI –

   Writes (BusRdX) that go to the bus appear in bus order (and handled by snoopers in bus order!)

Issues

Rd, Wr sequence incurs 2 transactions

- even when no one sharing (e.g., serial program!)
- BusRd (I->S) followed by BusRdX (S->M)

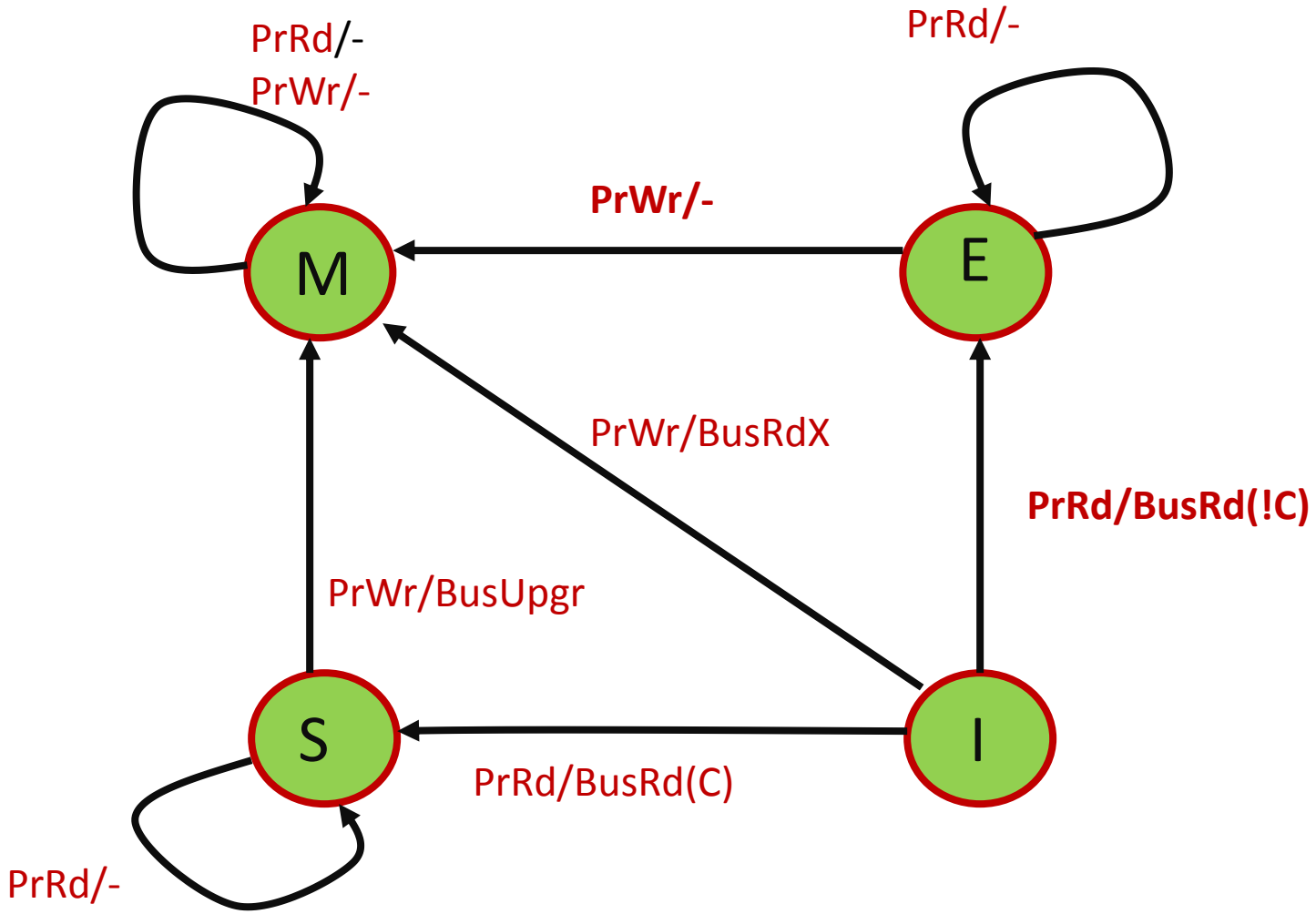
- *In general, penalizing serial programs is unacceptable*

Solution - MESI

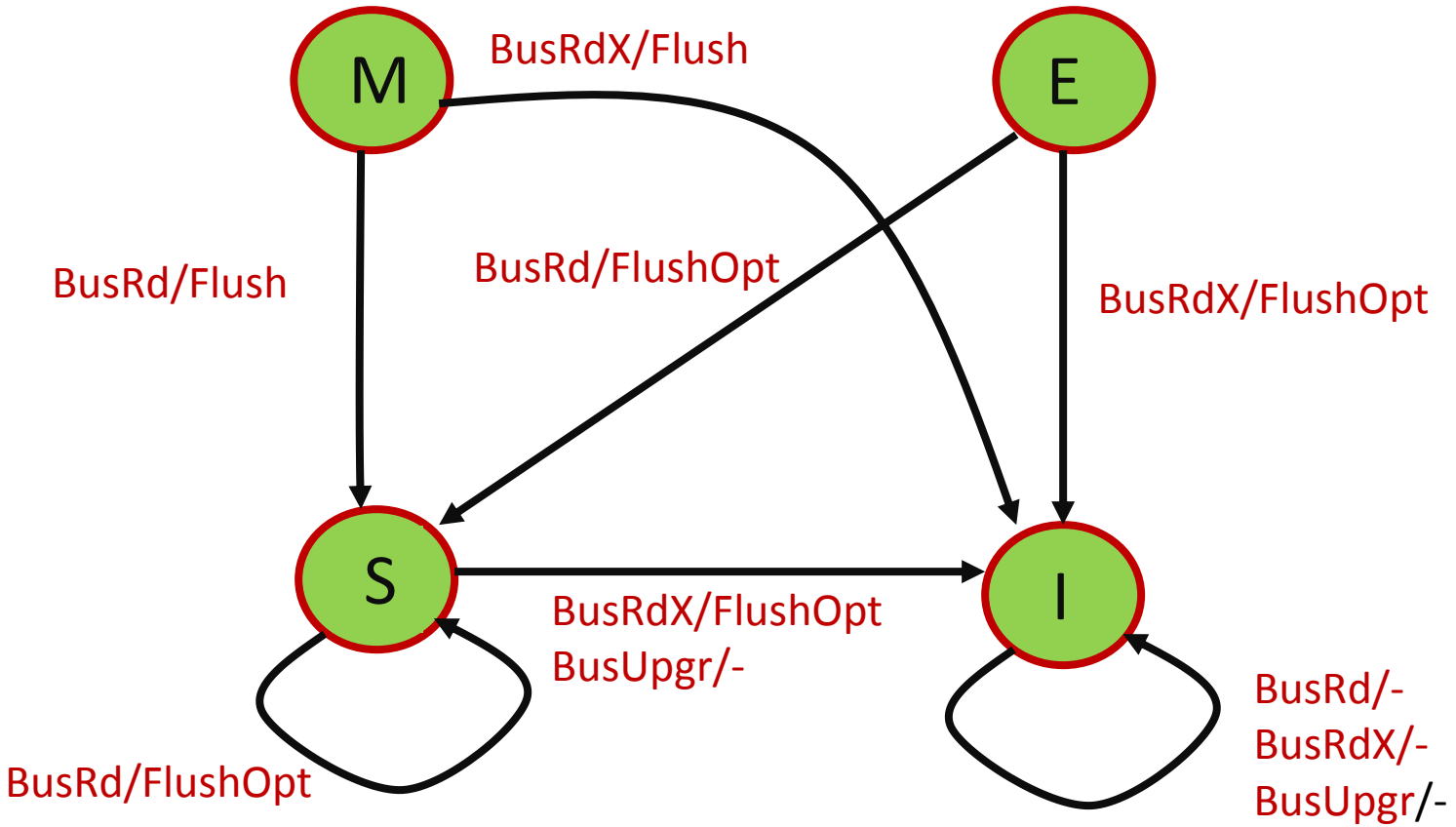
- Add **E***xclusive* state:
  - Invalid
  - modified (dirty)
  - shared (two or more caches may have copies)
  - Exclusive: (only this cache has *clean* copy, same value as in memory)

- How to decide I $\rightarrow$ E or I $\rightarrow$ S?
  - Need to check whether someone else has copy
  - a separate signal on bus: *wired-or* line asserted in response to BusRd. Call it C = "copy exists"

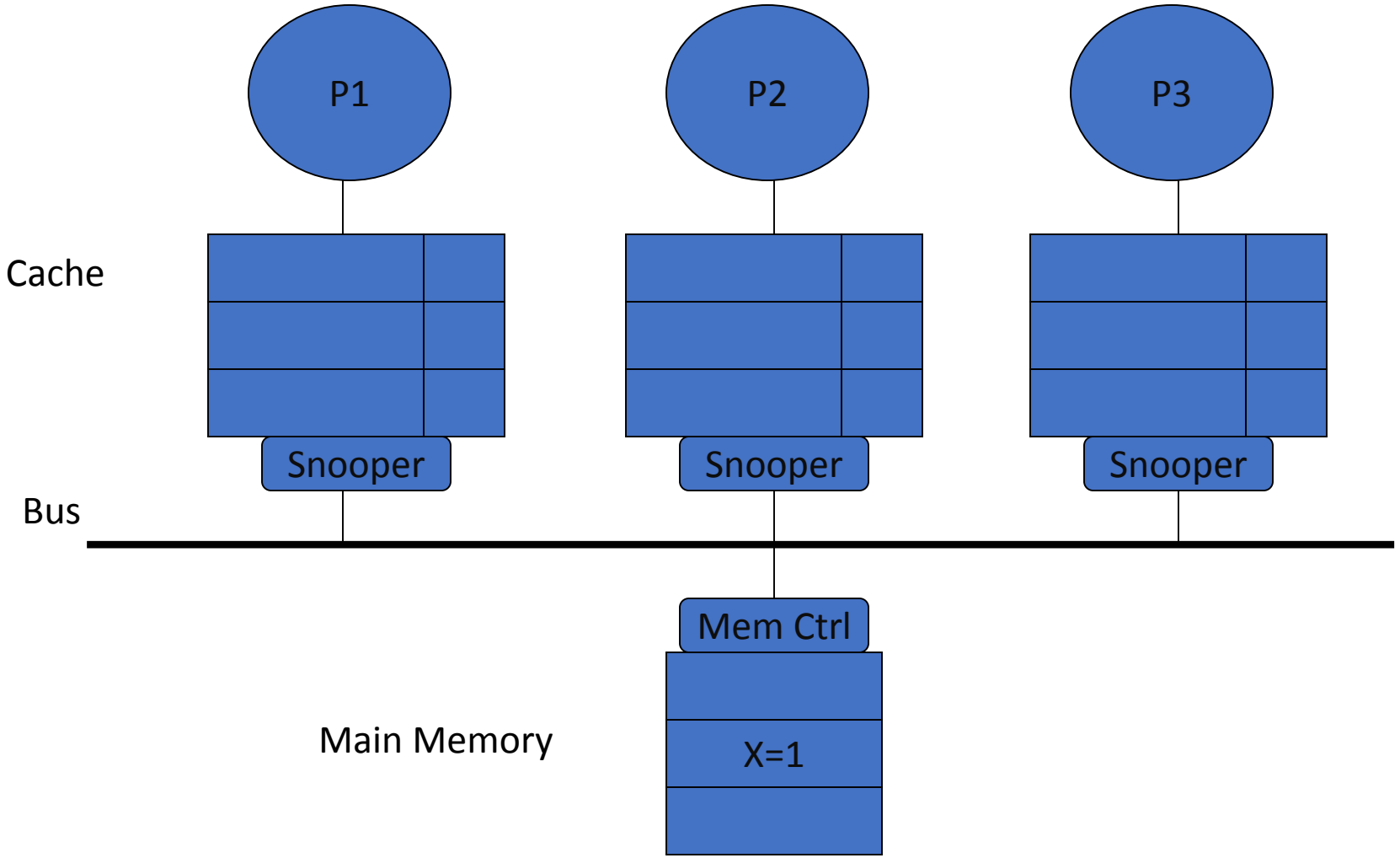
State Transitions – Processor level

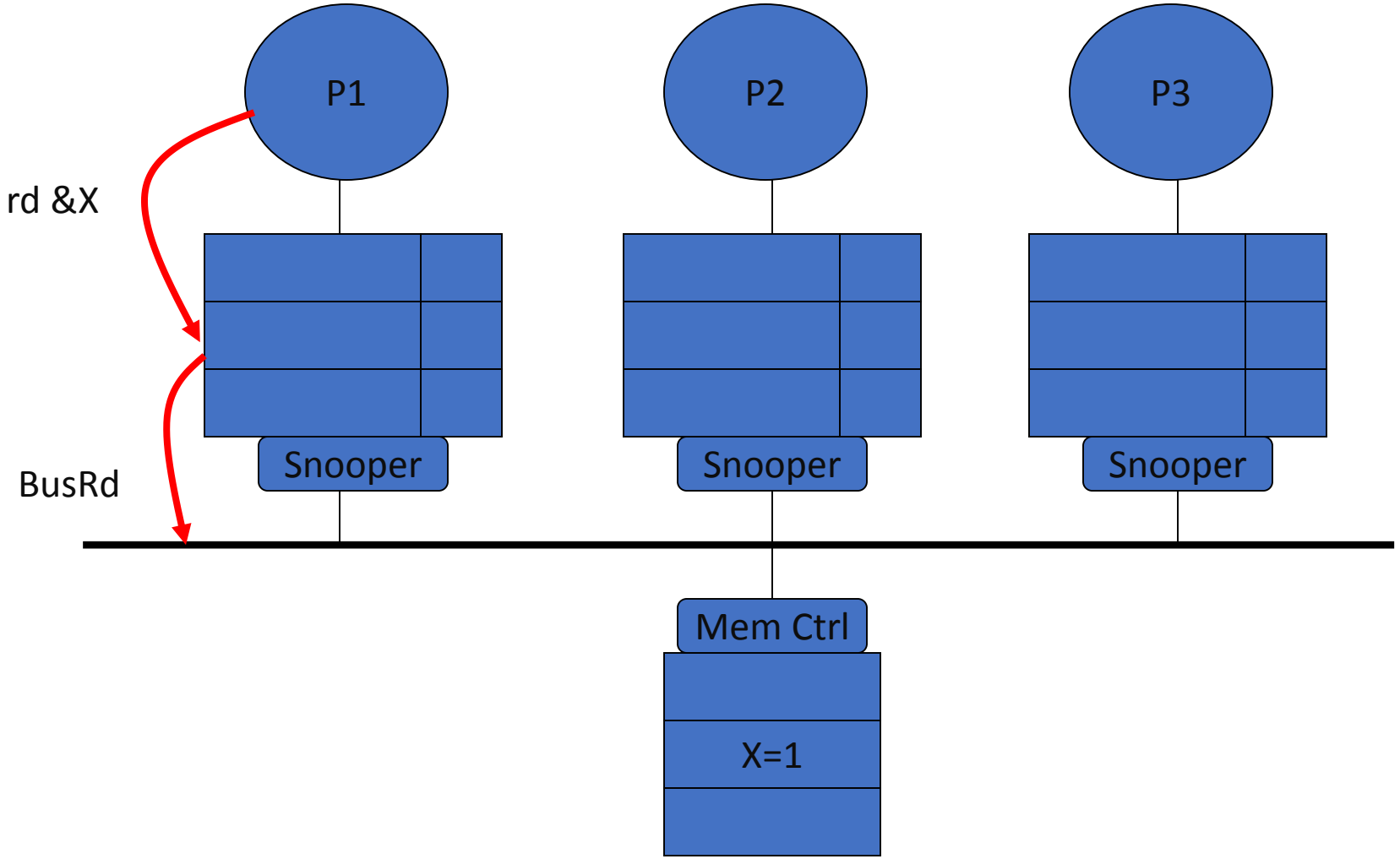


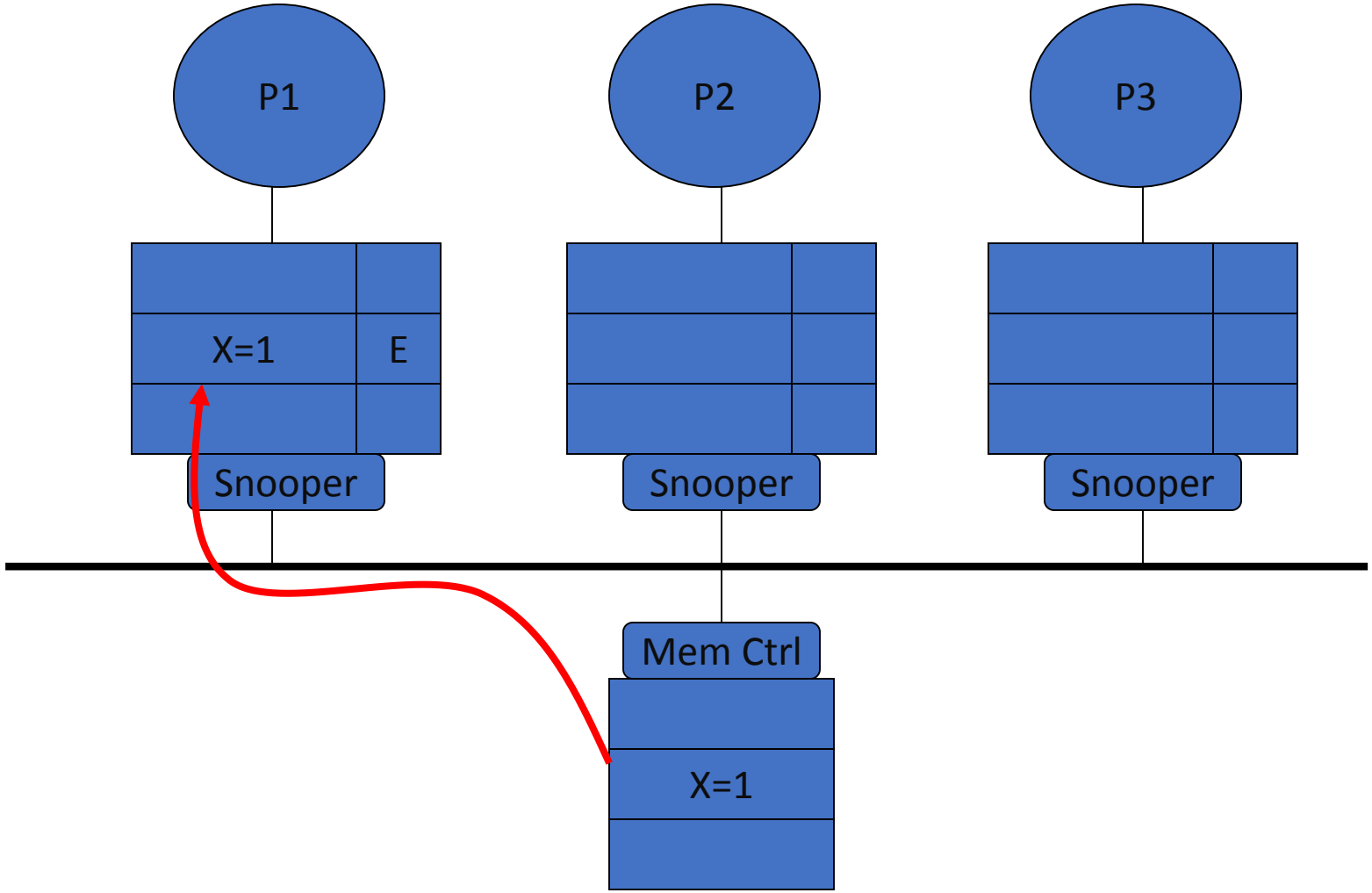
Bus Level

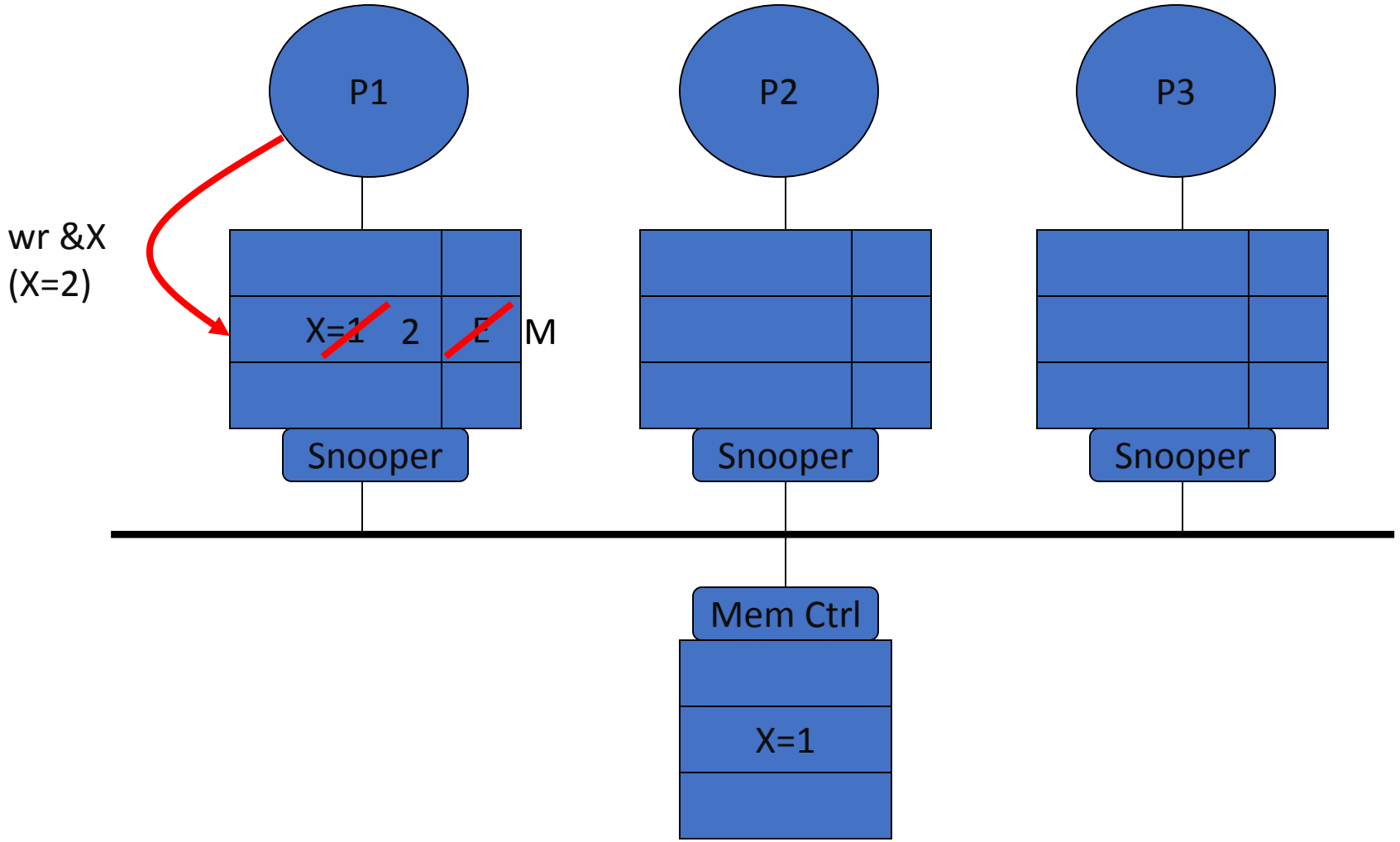

M
E
S
I
BusRdX/Flush
BusRd/Flush
BusRd/FlushOpt
BusRdX/FlushOpt
BusRdX/FlushOpt
BusUpgr/-
BusRd/FlushOpt
BusRd/-
BusRdX/-
BusUpgr/-

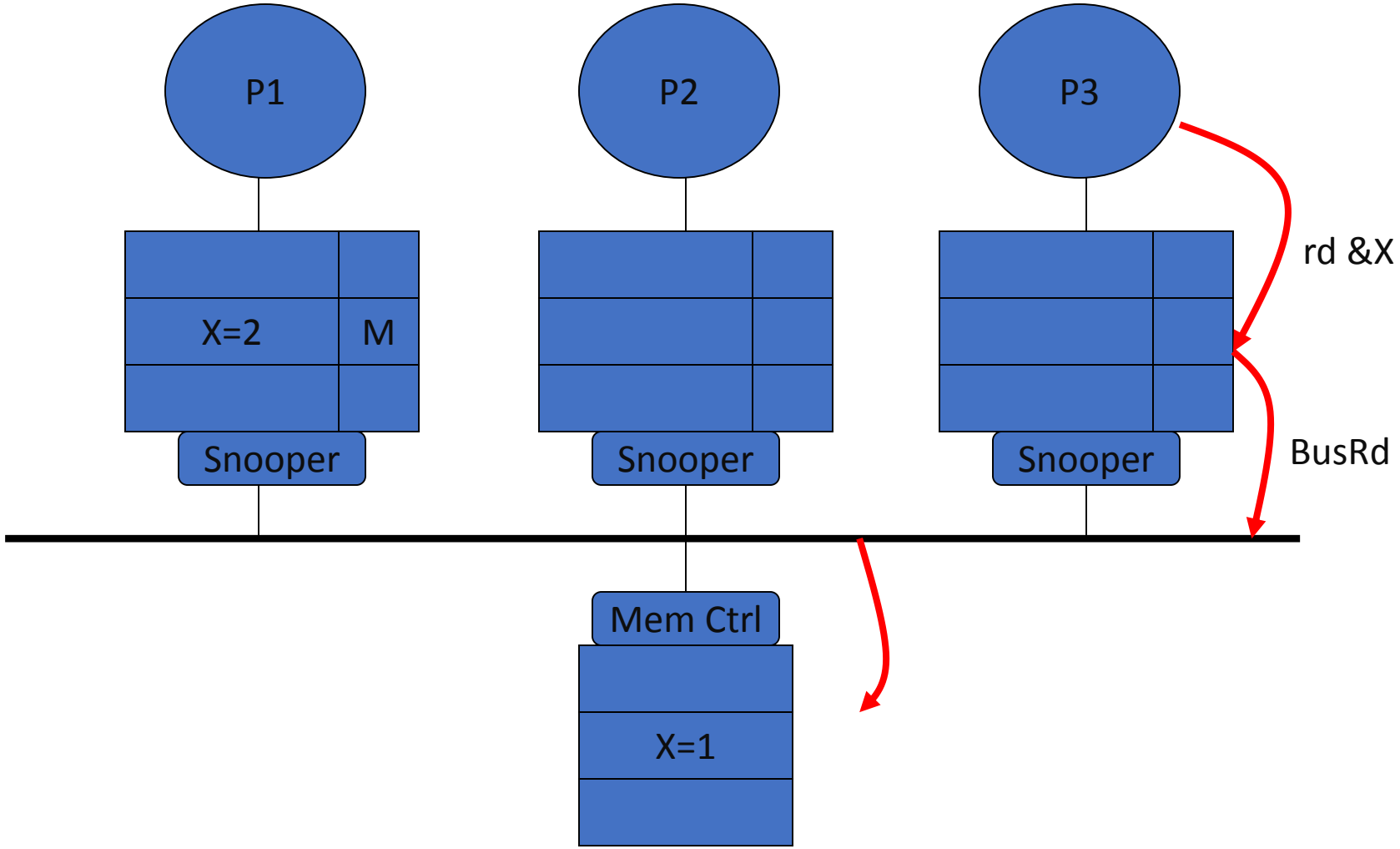
MESI: An Example

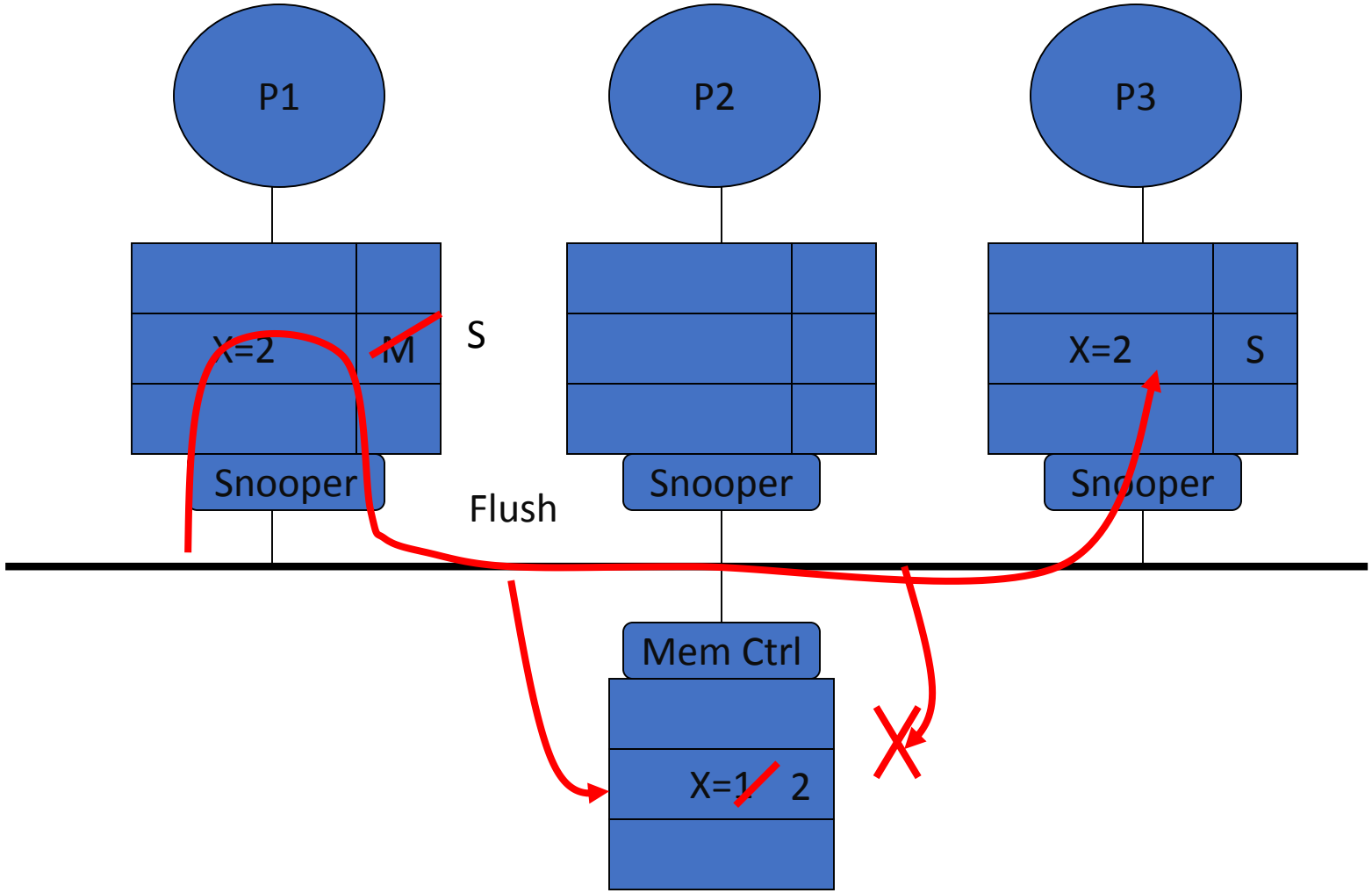


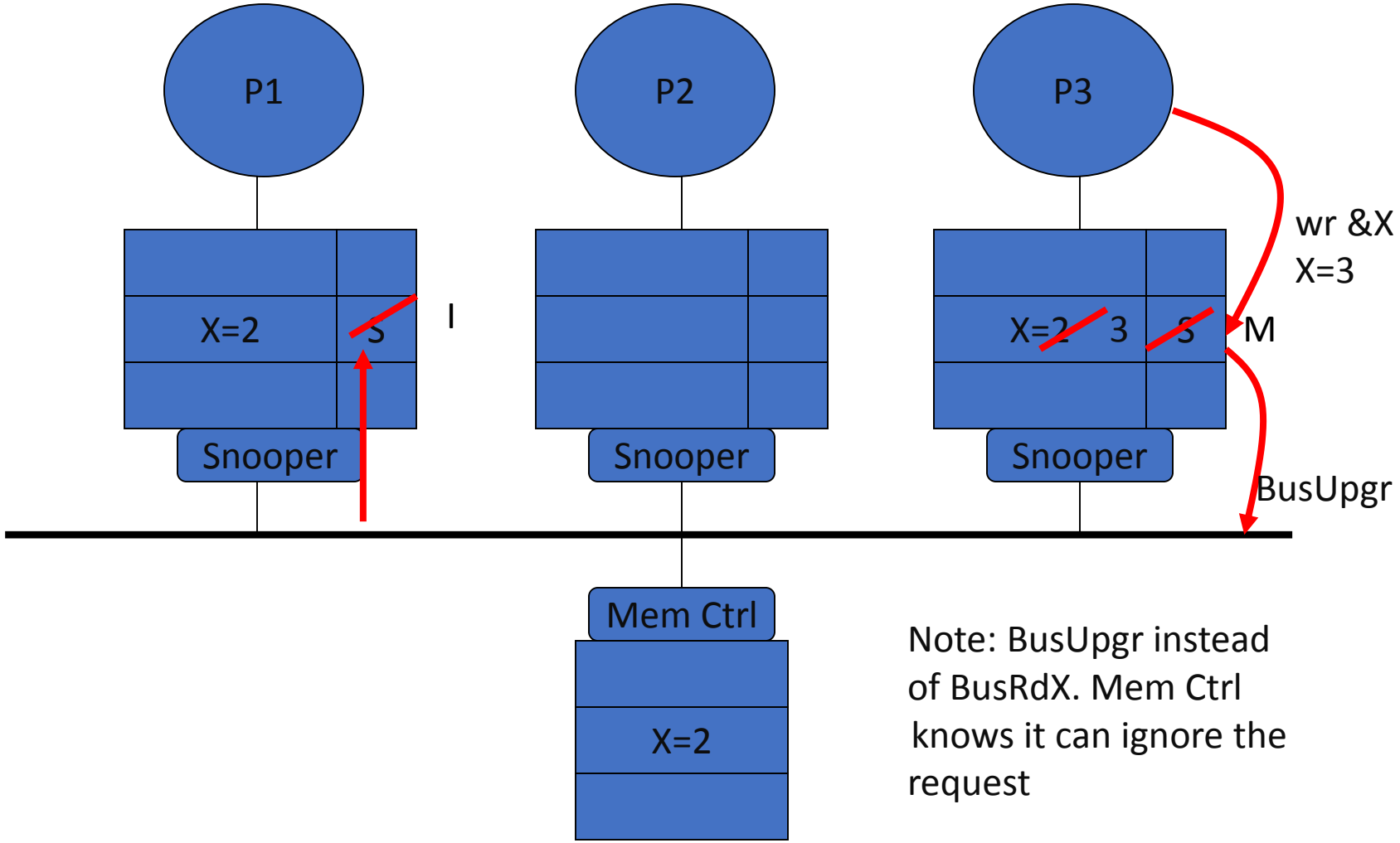

P1
P2
P3
rd &X
BusRd
Snooper
Snooper
Snooper
Mem Ctrl
X=1

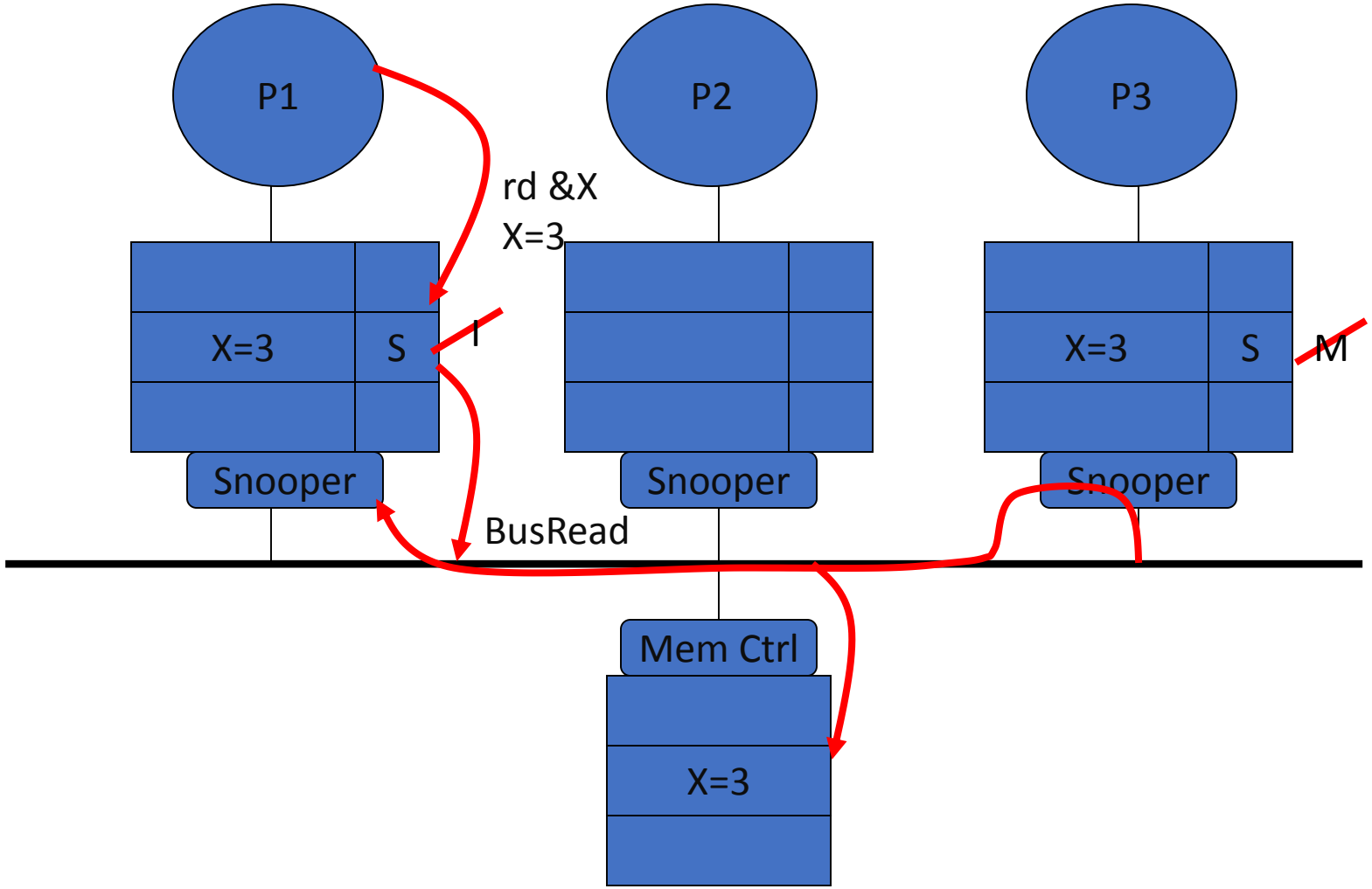

P1
P2
P3
X=1
E
Snooper
Snooper
Snooper
Mem Ctrl
X=1

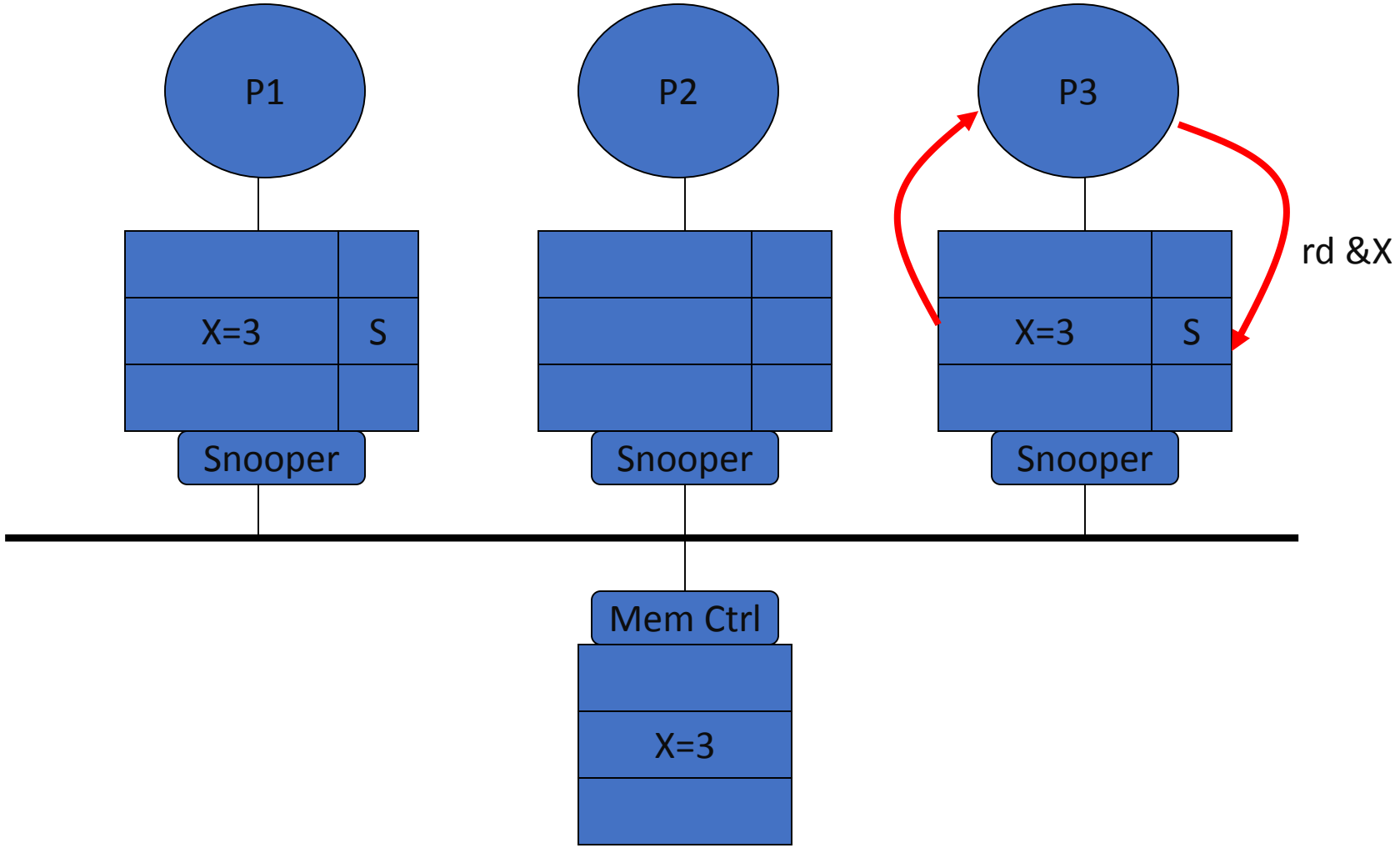

P1
P2
P3
wr &X
(X=2)
X=1
2
E
M
Snooper
Snooper
Snooper
Mem Ctrl
X=1

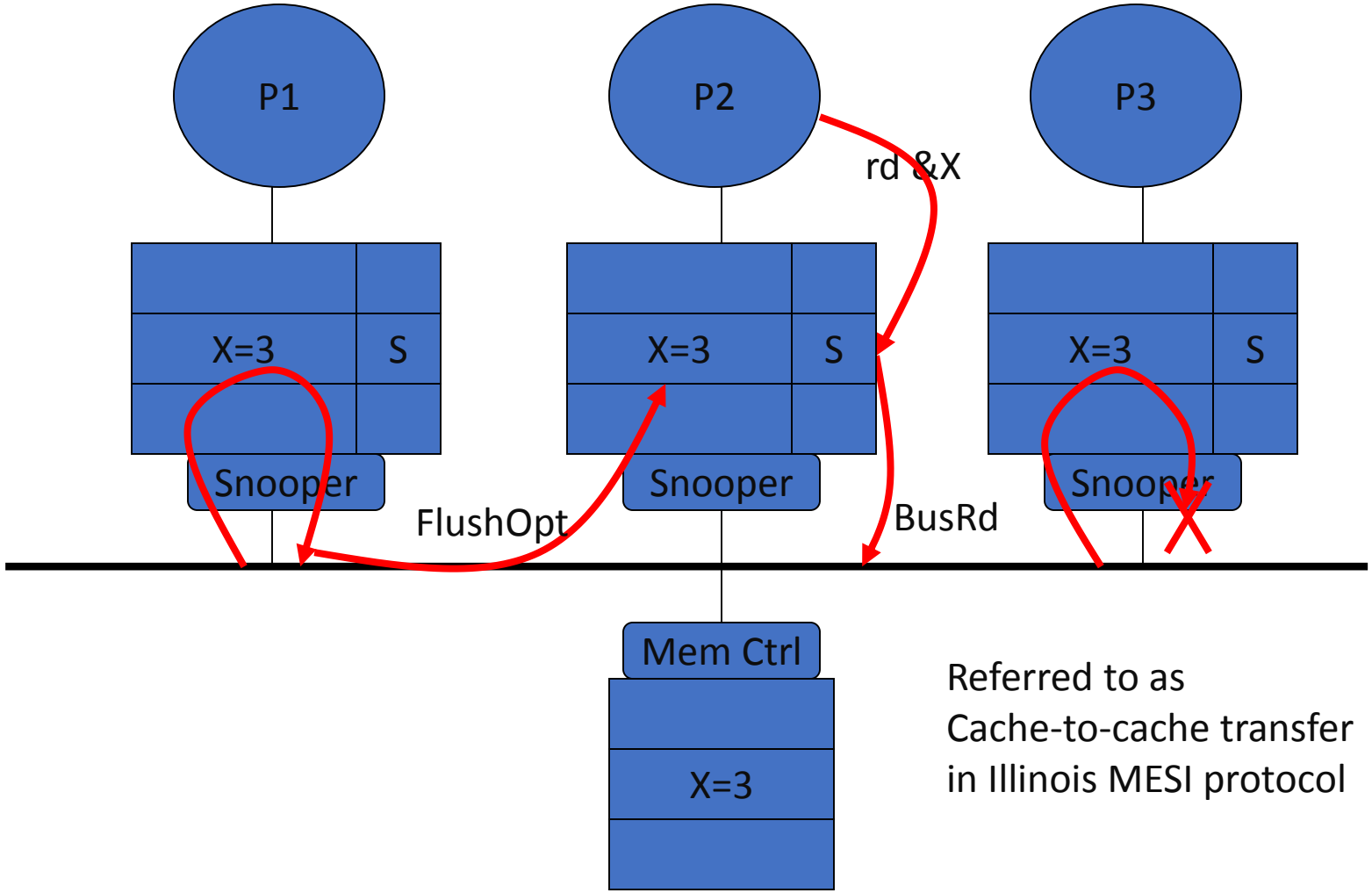

P1
P2
P3
X=2
M
Snooper
Snooper
Snooper
rd &X
BusRd
Mem Ctrl
X=1


P1
P2
P3
X=2
M
S
X=2
S
Snooper
Snooper
Snooper
Flush
Mem Ctrl
X=1
2


P1
P2
P3
X=2
S
I
X=2
3
S
M
wr &X
X=3
Snooper
Snooper
Snooper
BusUpgr
Mem Ctrl
X=2
Note: BusUpgr instead of BusRdX. Mem Ctrl knows it can ignore the request


P1
P2
P3
rd &X
X=3
X=3
S
I
X=3
S
M
Snooper
Snooper
Snooper
BusRead
Mem Ctrl
X=3


P1
P2
P3
X=3
S
X=3
S
rd &X
Snooper
Snooper
Snooper
Mem Ctrl
X=3



Referred to as
Cache-to-cache transfer
in Illinois MESI protocol

Proc Action	State P1	State P2	State P3	Bus Action	Data From
R1	**E**	-	-	BusRd	Mem
W1	M	-	-	-	-
R3	S	-	S	BusRd/Flush	P1's cache
W3	I	-	M	BusUpgr	-
R1	S	-	S	BusRd/Flush	P3's cache
R3	S	-	S	-	-
R2	S	S	S	BusRd/Flush Opt	**P1/P3's cache**

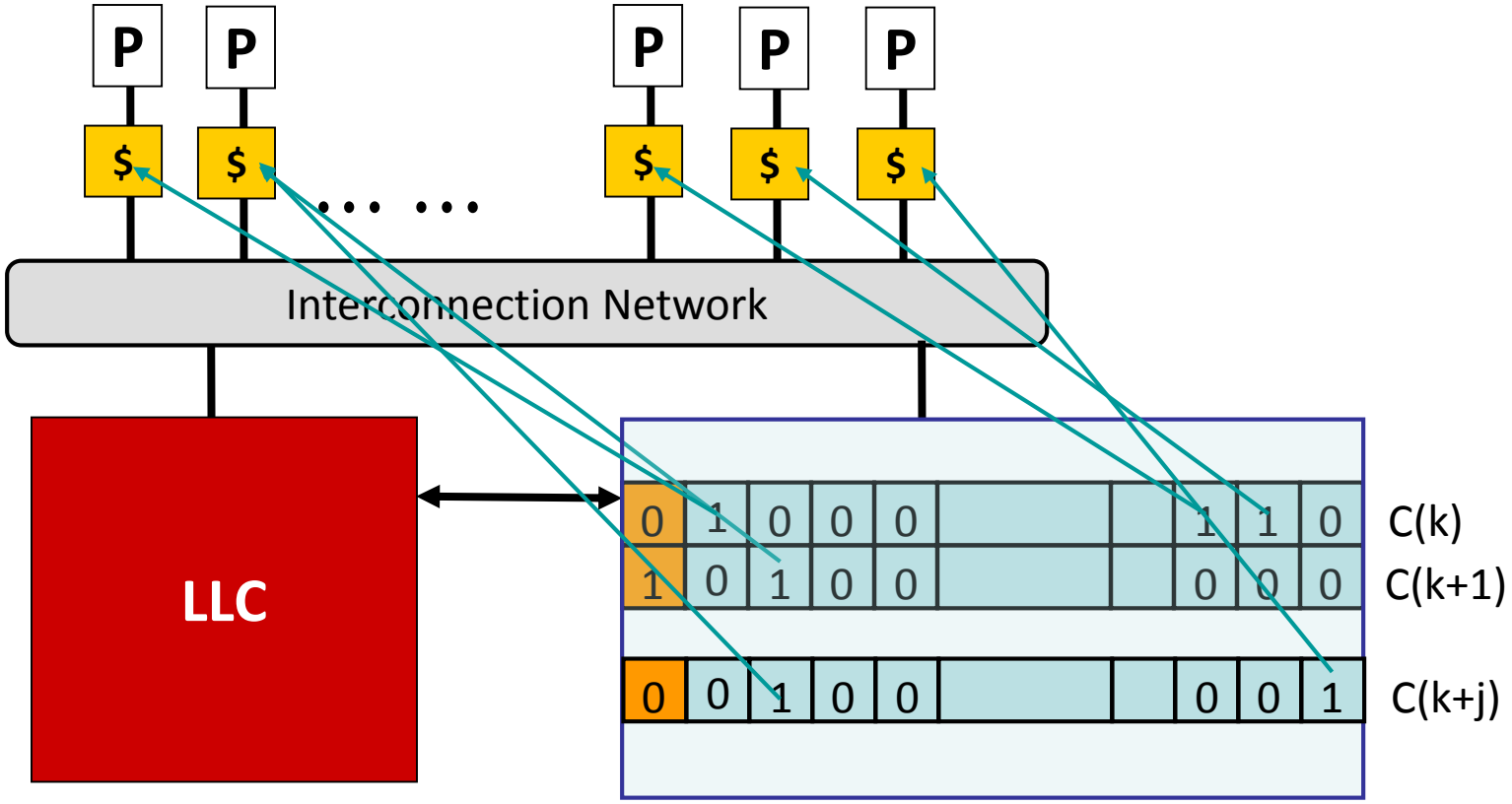
Coherence misses: misses due to accesses to blocks that were invalidated due to coherence events

True sharing: misses that occur when multiple threads share the same data item (byte/word)

False sharing: misses that occur when multiple threads share different data items located in a single cache block

Parameters	True Sharing Misses	False Sharing Misses
Larger cache size	increased	increased
Larger block/line size	decreased	increased
Larger associativity	unclear	unclear

Directory Based



1 modified bit for each cache block in LLC/ memory

1 presence bit for each core, each cache block in LLC/ memory