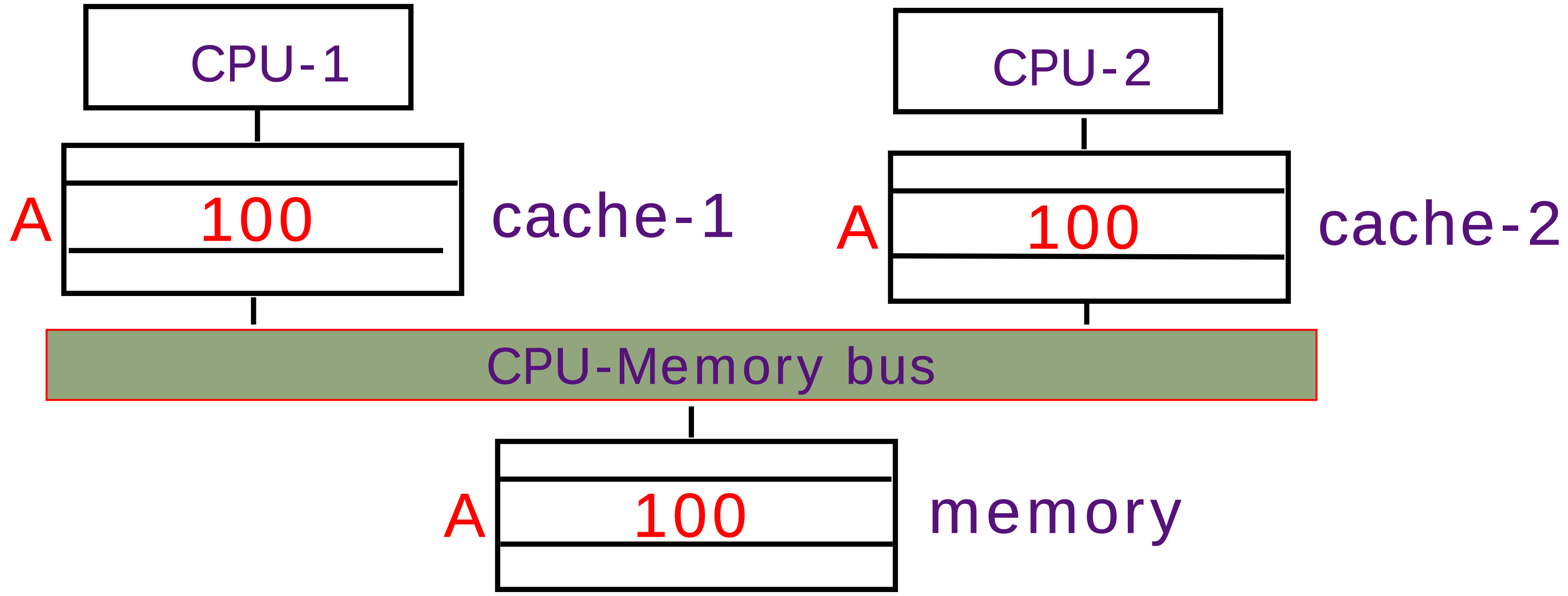
Sequential Consistency and Cache Coherence Protocols

Arvind
Computer Science and Artificial Intelligence Lab
M.I.T.

Based on the material prepared by Arvind and Krste Asanovic

Memory Consistency in SMPs



Suppose CPU-1 updates A to 200.

write-back: memory and cache-2 have stale values

write-through: cache-2 has a stale value

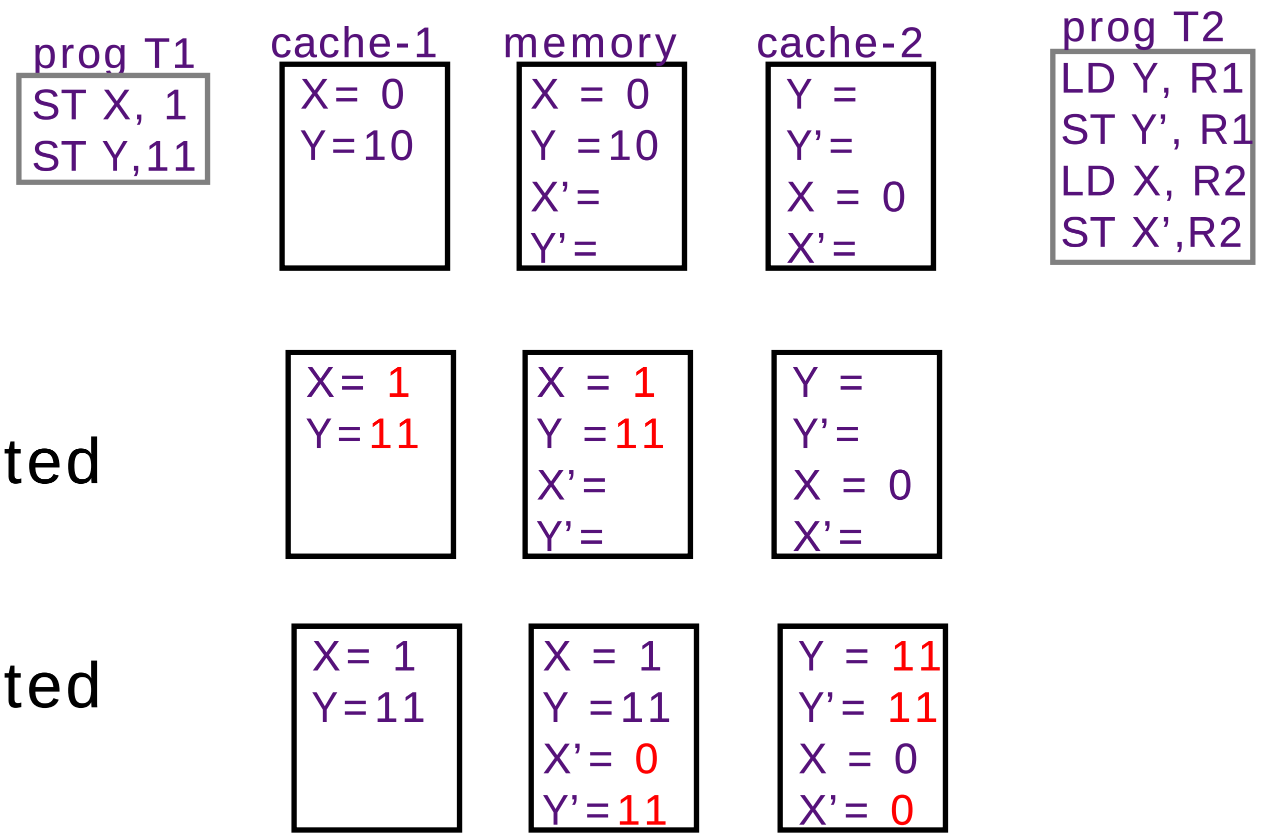
Do these stale values matter?

What is the view of shared memory for programming?

Write-back Caches & SC

	prog T1	cache-1	memory	cache-2	prog T2
• T1 is executed	ST X, 1 ST Y,11	X= 1 Y=11	X = 0 Y =10 X'= Y'=	Y = Y'= X = X'=	LD Y, R1 ST Y', R1 LD X, R2 ST X',R2
• cache-1 writes back Y		X= 1 Y=11	X = 0 Y =11 X'= Y'=	Y = Y'= X = X'=	
• T2 executed		X= 1 Y=11	X = 0 Y =11 X'= Y'=	Y = 11 Y'= 11 X = 0 X'= 0	
• cache-1 writes back X		X= 1 Y=11	X = 1 Y =11 X'= Y'=	Y = 11 Y'= 11 X = 0 X'= 0	
• cache-2 writes back X' & Y'		X= 1 Y=11	X = 1 Y =11 X'= 0 Y'=11	Y =11 Y'=11 X = 0 X'= 0	incoherent

Write-through Caches & SC



- T1 executed
- T2 executed

Write-through caches don't preserve sequential consistency either

Maintaining Sequential Consistency

SC is sufficient for correct producer-consumer and mutual exclusion code (e.g., Dekker)

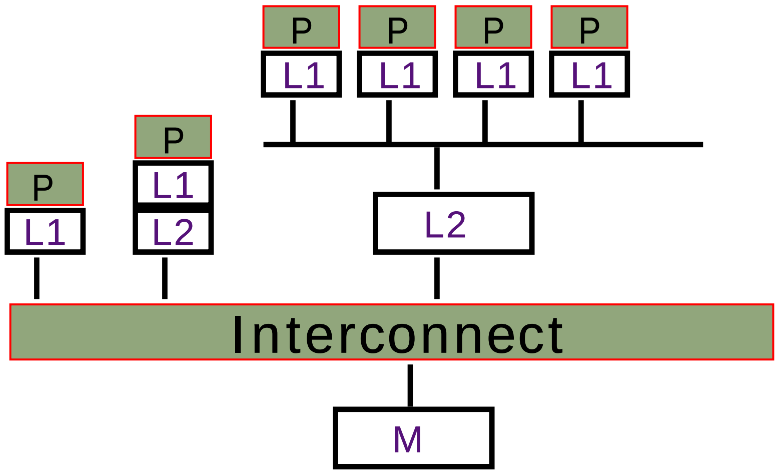
Multiple copies of a location in various caches can cause SC to break down.

Hardware support is required such that

- only one processor at a time has write permission for a location
- no processor can load a stale copy of the location after a write

$\Rightarrow$ *cache coherence protocols*

A System with Multiple Caches



- Modern systems often have hierarchical caches
- Each cache has exactly one parent but can have zero or more children
- Only a parent and its children can communicate directly
- *Inclusion property* is maintained between a parent and its children, i.e.,

$$a \in L_i \quad \Rightarrow \quad a \in L_{i+1}$$

Cache Coherence Protocols for SC

write request:

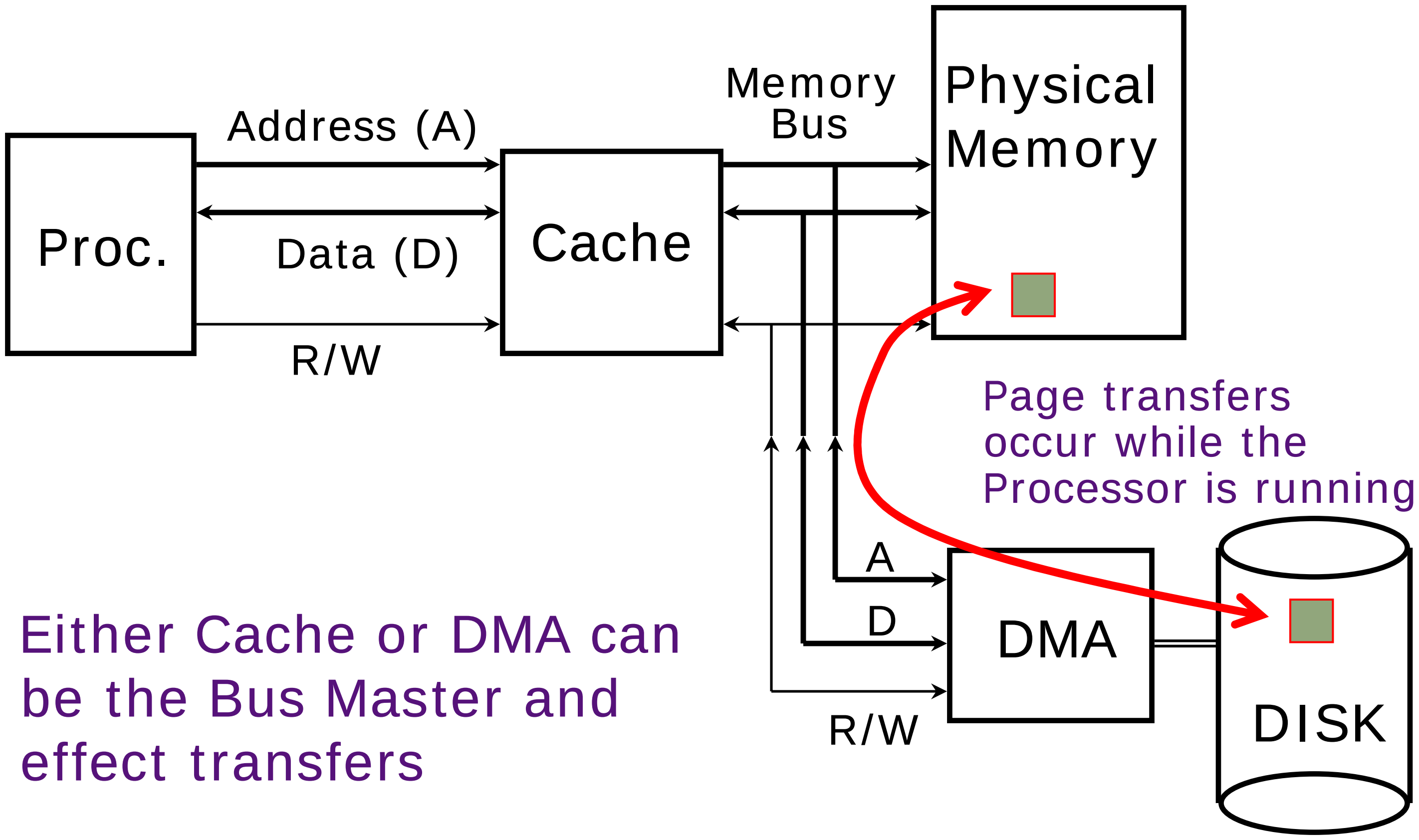
the address is *invalidated* (*updated)* in all other caches *before* (*after*) the write is performed

read request:

if a dirty copy is found in some cache, a write-back is performed before the memory is read

We will focus on Invalidation protocols as opposed to Update protocols

Warmup: Parallel I/O

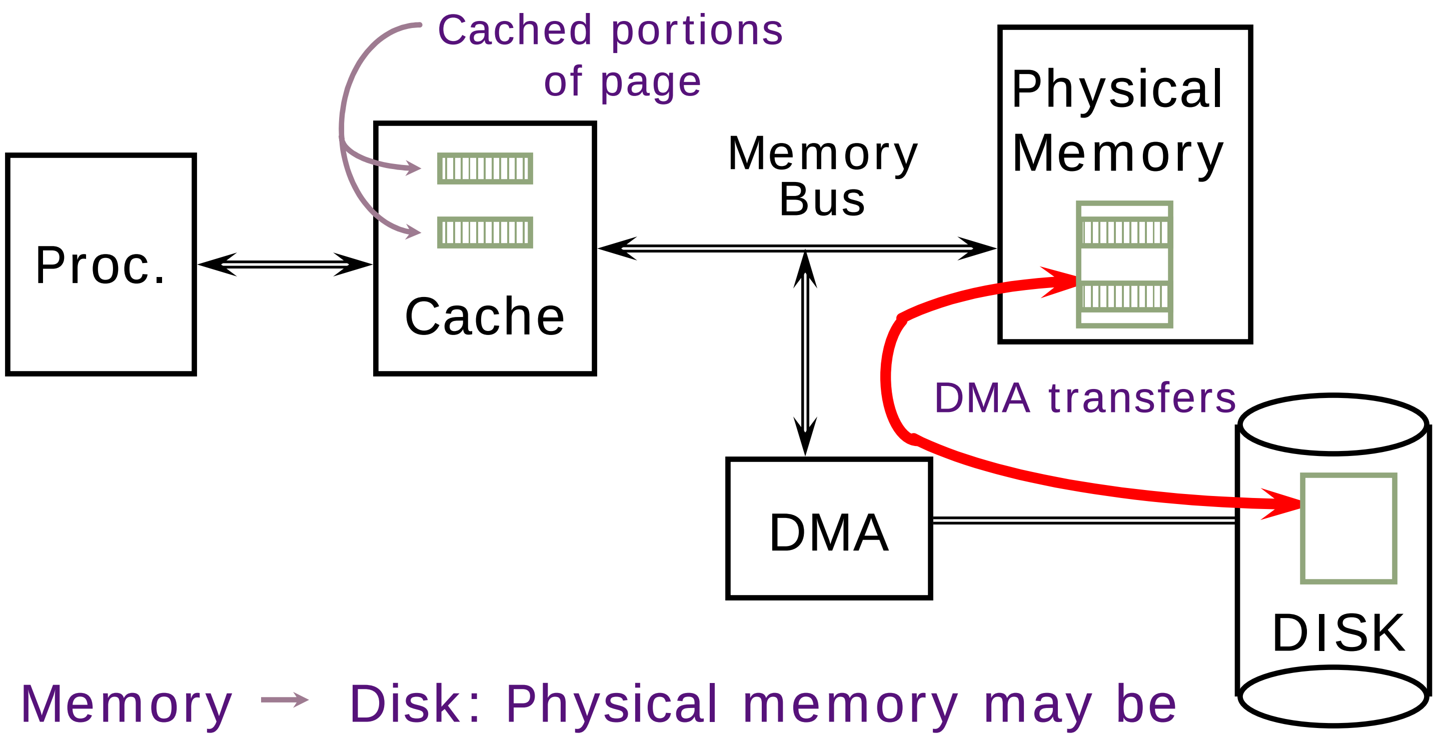


Page transfers occur while the Processor is running

Either Cache or DMA can be the Bus Master and effect transfers

DMA stands for Direct Memory Access

Problems with Parallel I/O

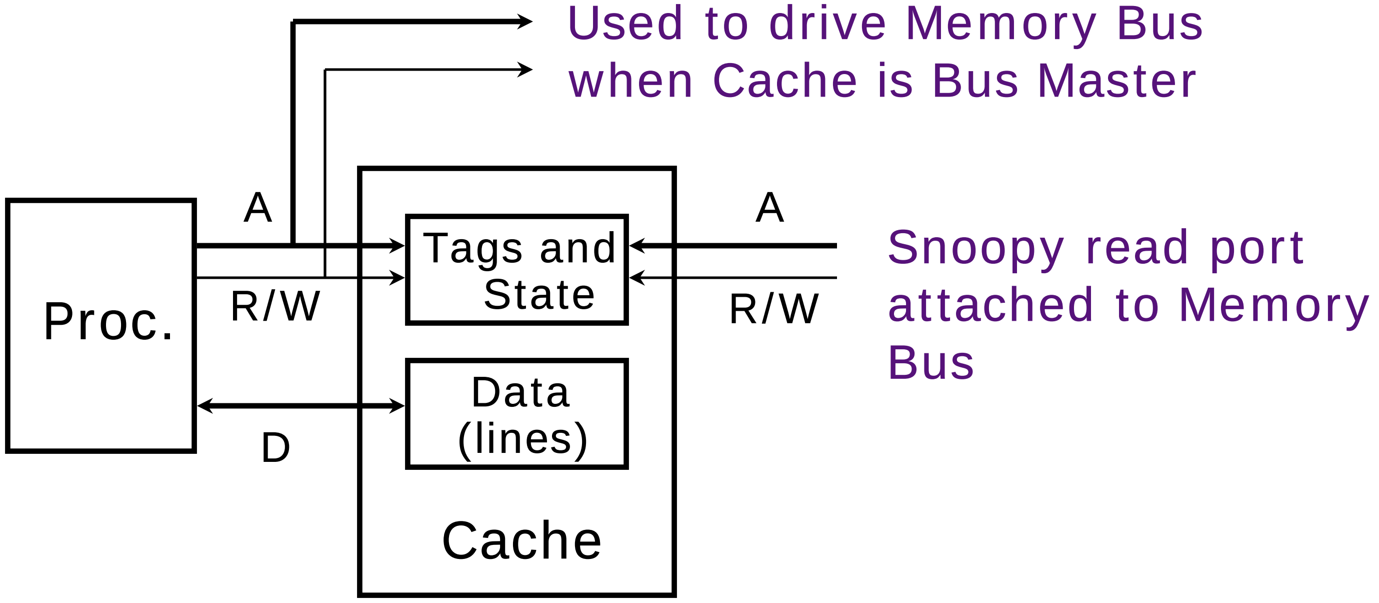


Memory → Disk: Physical memory may be stale if Cache copy is dirty

Disk → Memory: Cache may have data corresponding to the memory

Snoopy Cache *Goodman 1983*

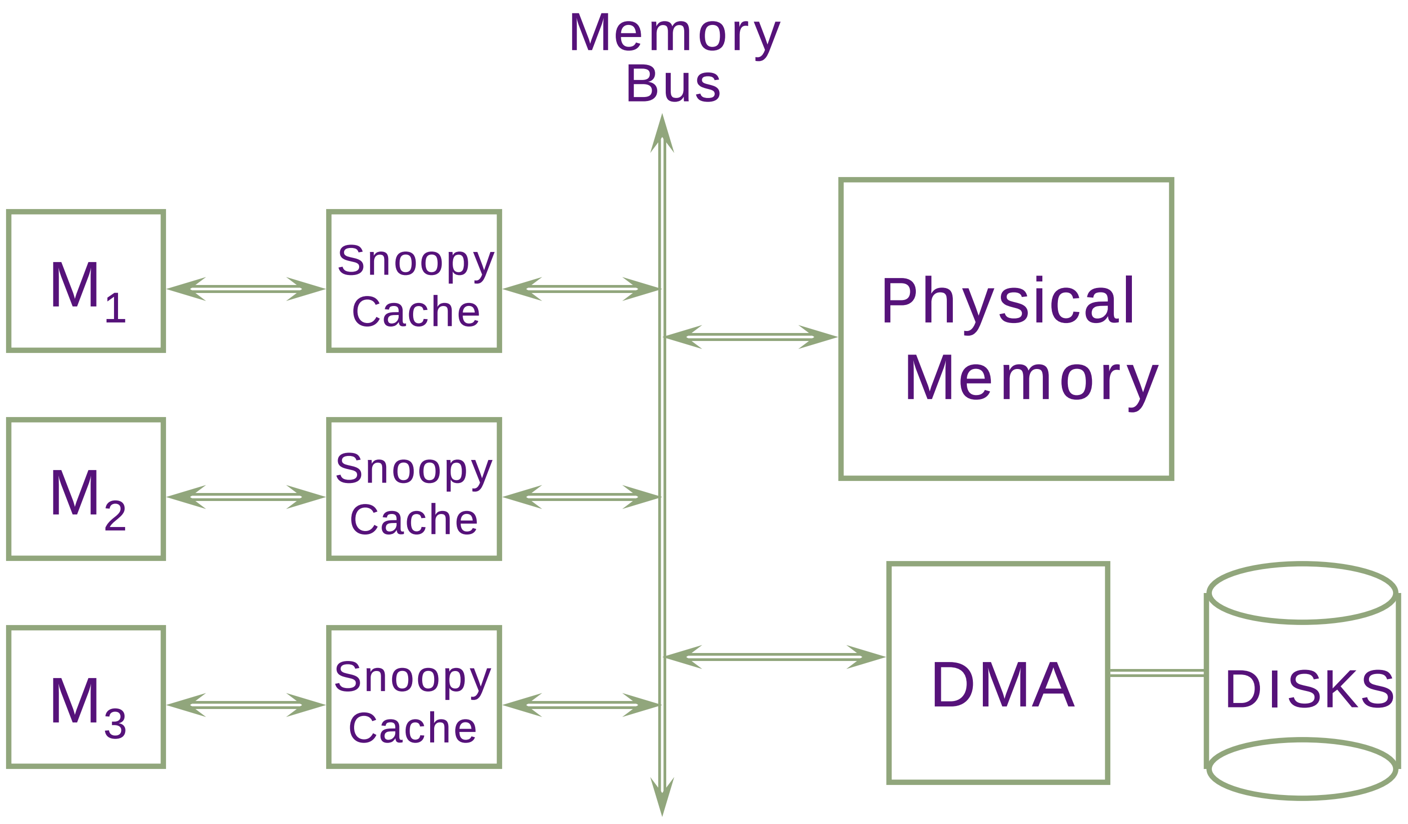
- Idea: Have cache watch (or snoop upon) DMA transfers, and then “do the right thing”
- Snoopy cache tags are dual-ported


Used to drive Memory Bus when Cache is Bus Master
A
R/W
Proc.
D
Tags and State
Data (lines)
Cache
A
R/W
Snoopy read port attached to Memory Bus

Snoopy Cache Actions

Observed Bus Cycle	Cache State	Cache Action
Read Cycle Memory → Disk	Address not cached	No action
	Cached, unmodified	No action
	Cached, modified	Cache intervenes
Write Cycle Disk → Memory	Address not cached	No action
	Cached, unmodified	Cache purges its copy
	Cached, modified	???

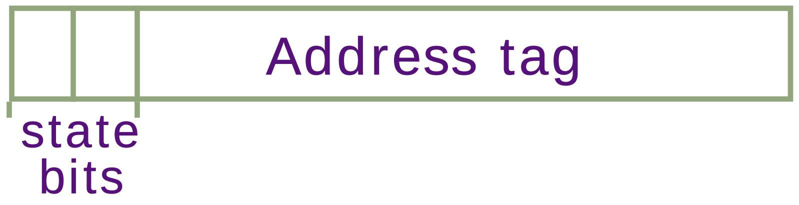
Shared Memory Multiprocessor



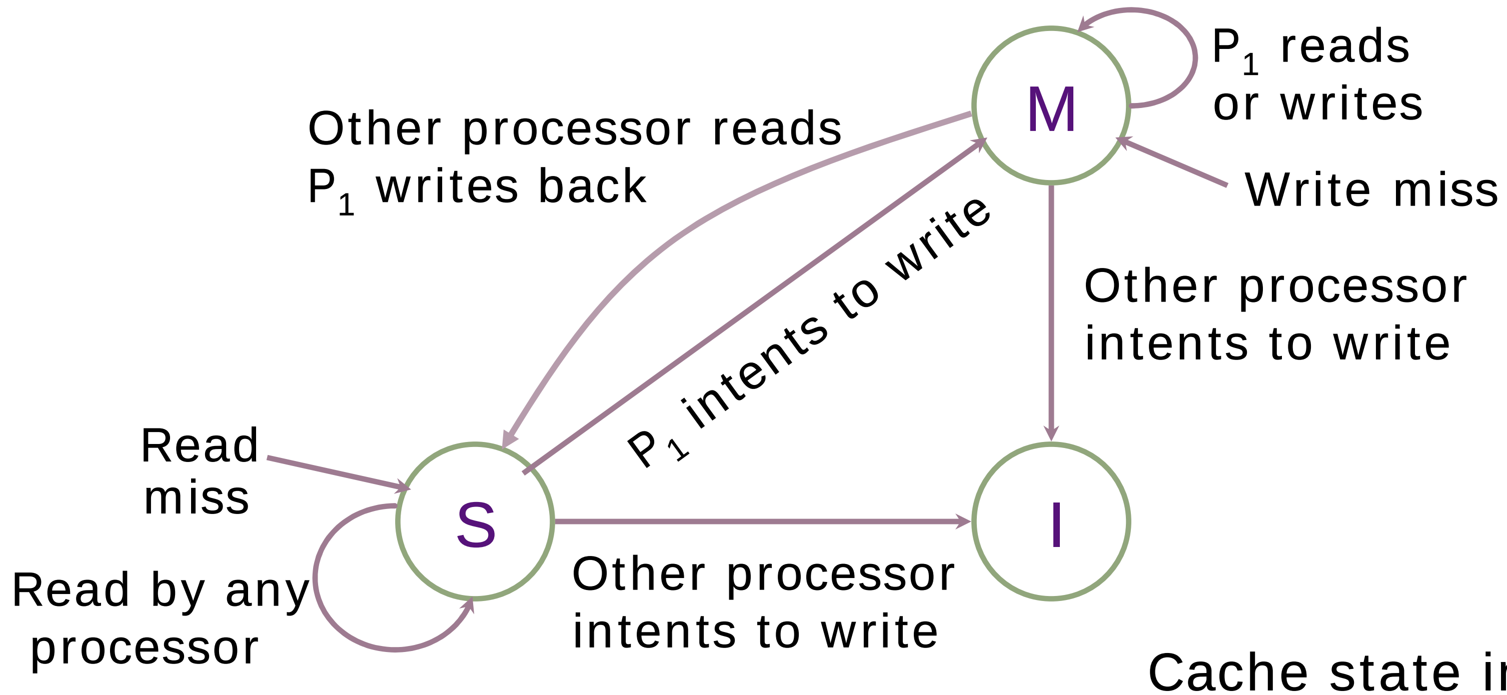
Use snoopy mechanism to keep all processors' view of memory coherent

Cache State Transition Diagram
The MSI protocol

Each cache line has a tag



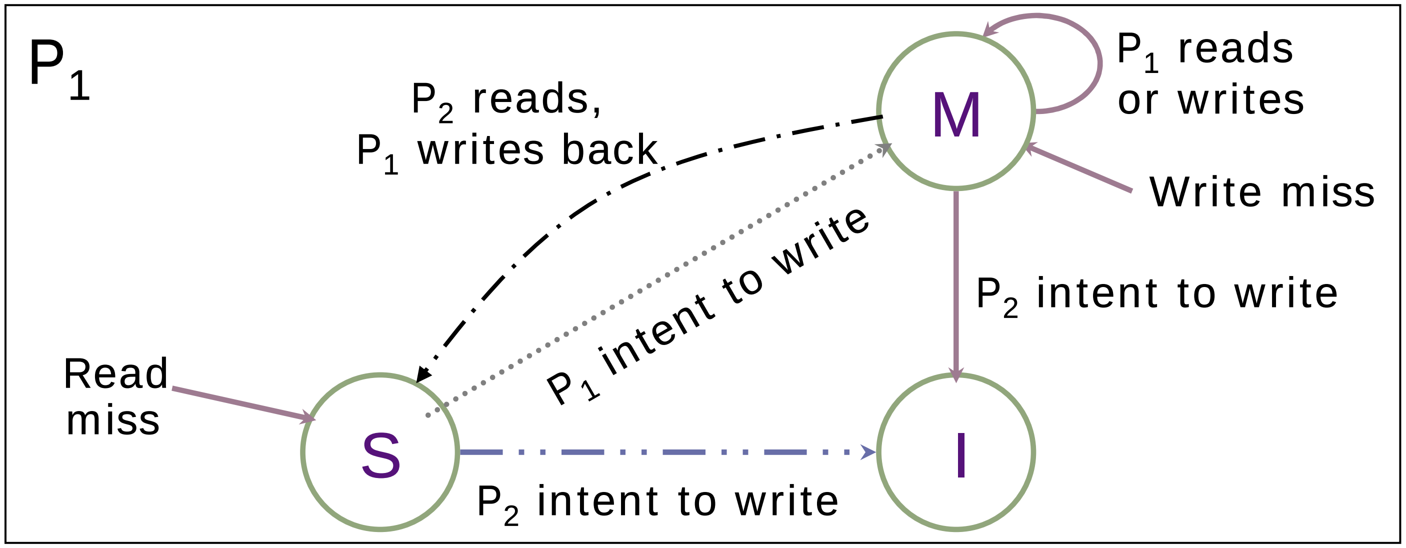
M: Modified
S: Shared
I: Invalid

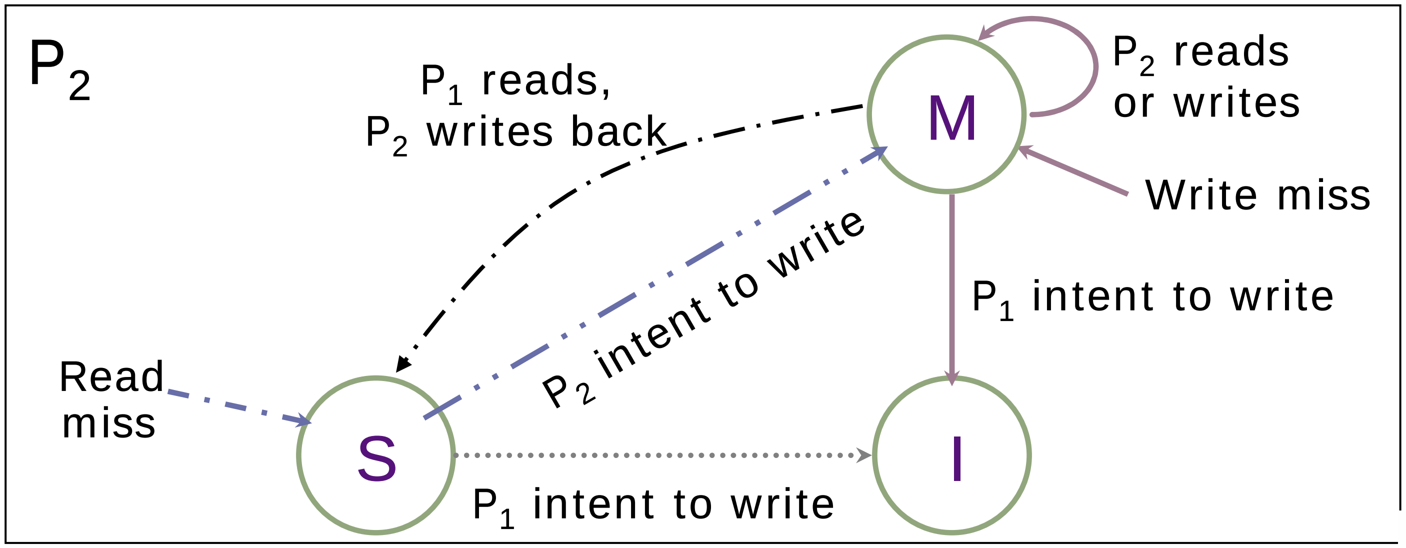

M
S
I
P_1 reads or writes
Write miss
Other processor reads P_1 writes back
P_1 intents to write
Other processor intents to write
Read miss
Read by any processor
Other processor intents to write

Cache state in processor P_1

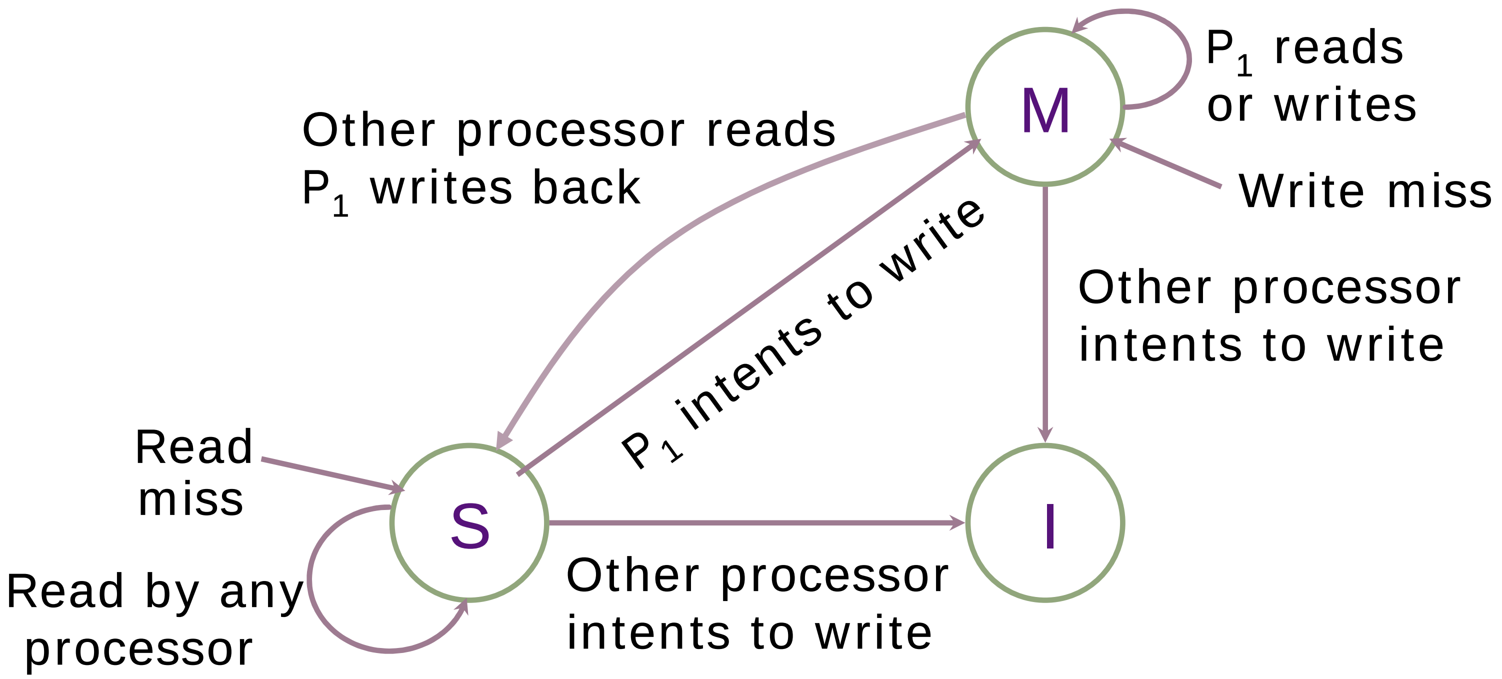
2 Processor Example

P_1 reads
P_1 writes
P_2 reads
P_2 writes
P_1 reads
P_1 writes
P_2 writes
P_1 writes





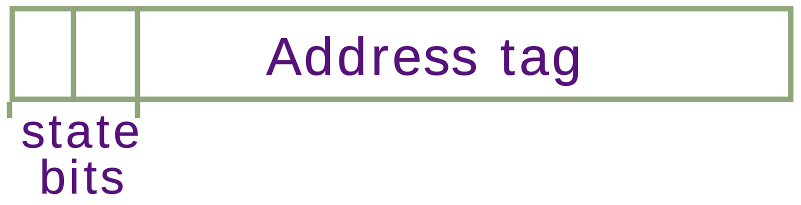
Observation


M
S
I
P1 reads or writes
Write miss
Other processor reads P1 writes back
P1 intents to write
Other processor intents to write
Read miss
Read by any processor
Other processor intents to write

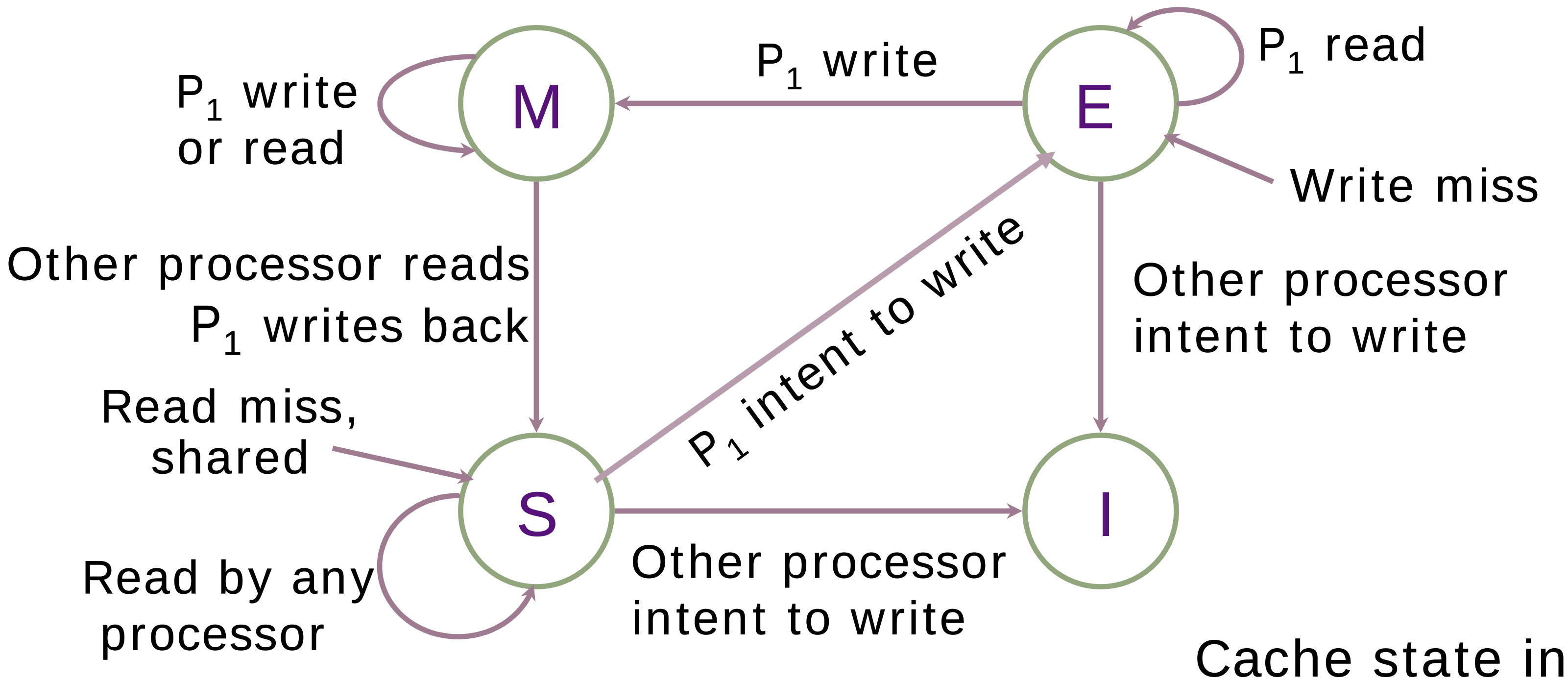
- If a line is in the M state then no other cache can have a copy of the line!
  - Memory stays coherent, multiple differing copies cannot exist

MESI: An Enhanced MSI protocol

Each cache line has a tag



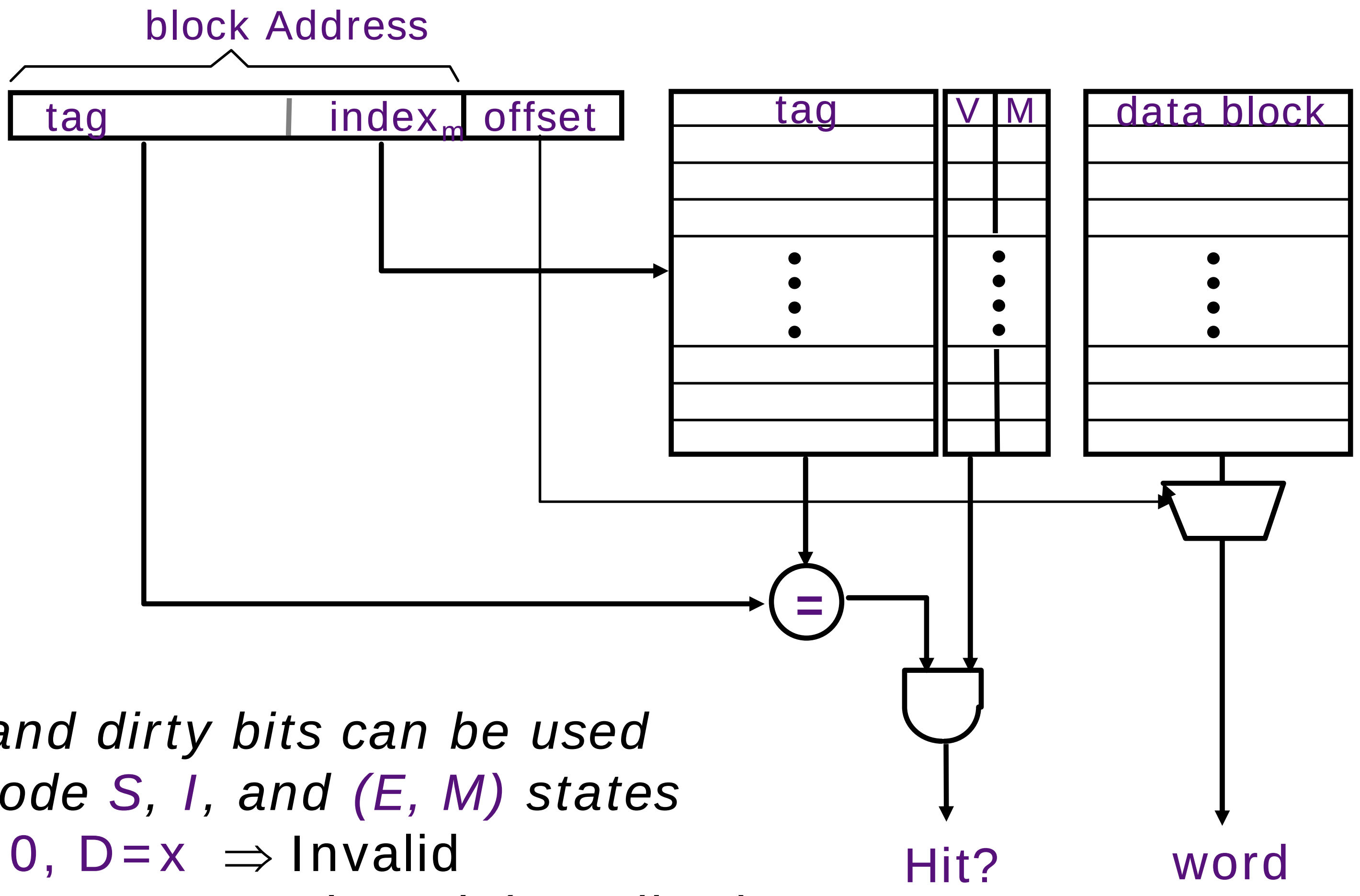
M: Modified Exclusive
E: Exclusive, unmodified
S: Shared
I: Invalid



Cache state in processor P_1

Five-minute break to stretch your legs

Cache Coherence State Encoding



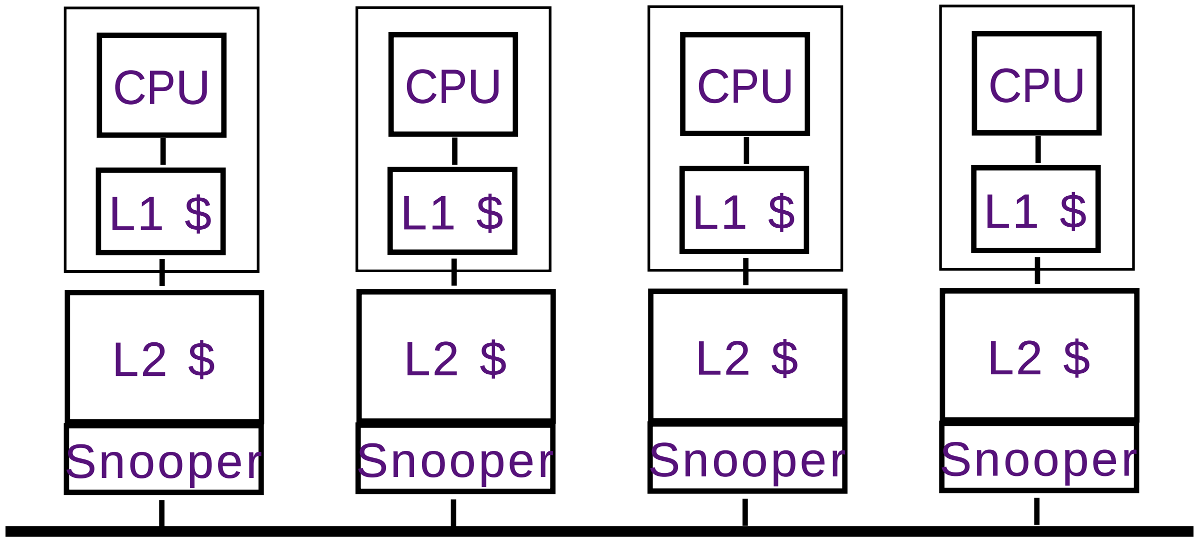
Valid and dirty bits can be used to encode S, I, *and* (E, M) *states*

V=0, D=x ⇒ Invalid

V=1, D=0 ⇒ Shared *(not dirty)*

V=1, D=1 ⇒ Exclusive *(dirty)*

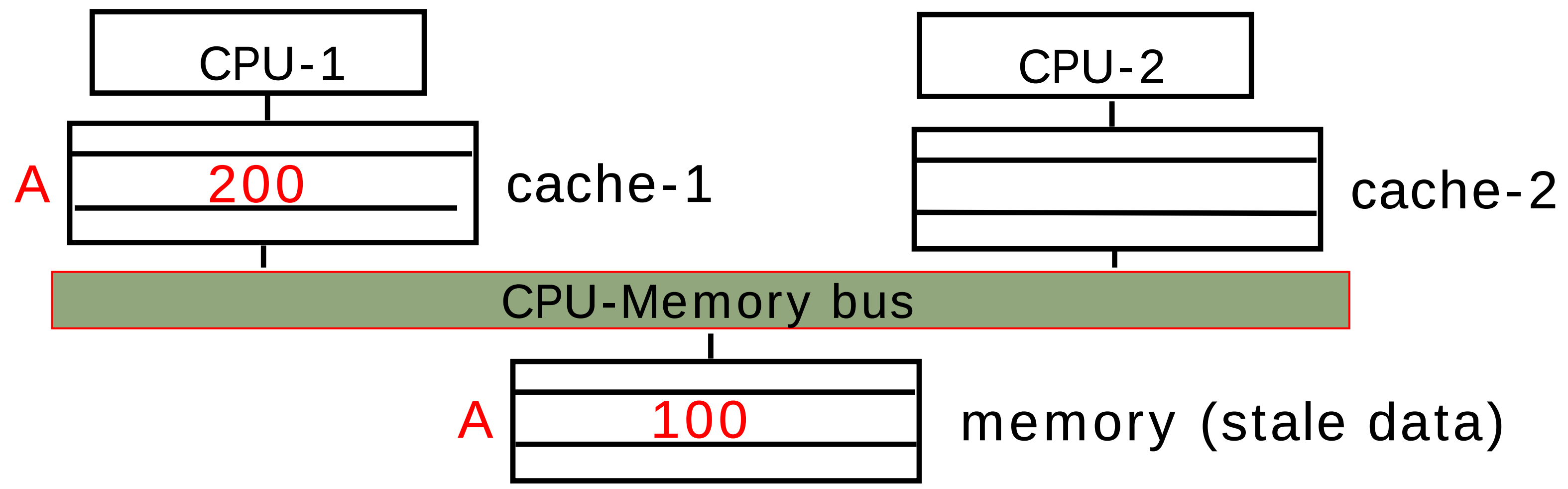
2-Level Caches



- Processors often have two-level caches
  - Small L1 on chip, large L2 off chip
- *Inclusion property:* entries in L1 must be in L2
  invalidation in L2 $\Rightarrow$ invalidation in L1
- Snooping on L2 does not affect CPU-L1 bandwidth

What problem could occur?

Intervention



When a read-miss for A occurs in cache-2, a read request for A is placed on the bus

- Cache-1 needs to supply & change its state to shared
- The memory may respond to the request also!

Does memory know it has stale data?

Cache-1 needs to intervene through memory controller to supply correct data to cache-2

False Sharing

state	blk addr	data0	data1	...	dataN

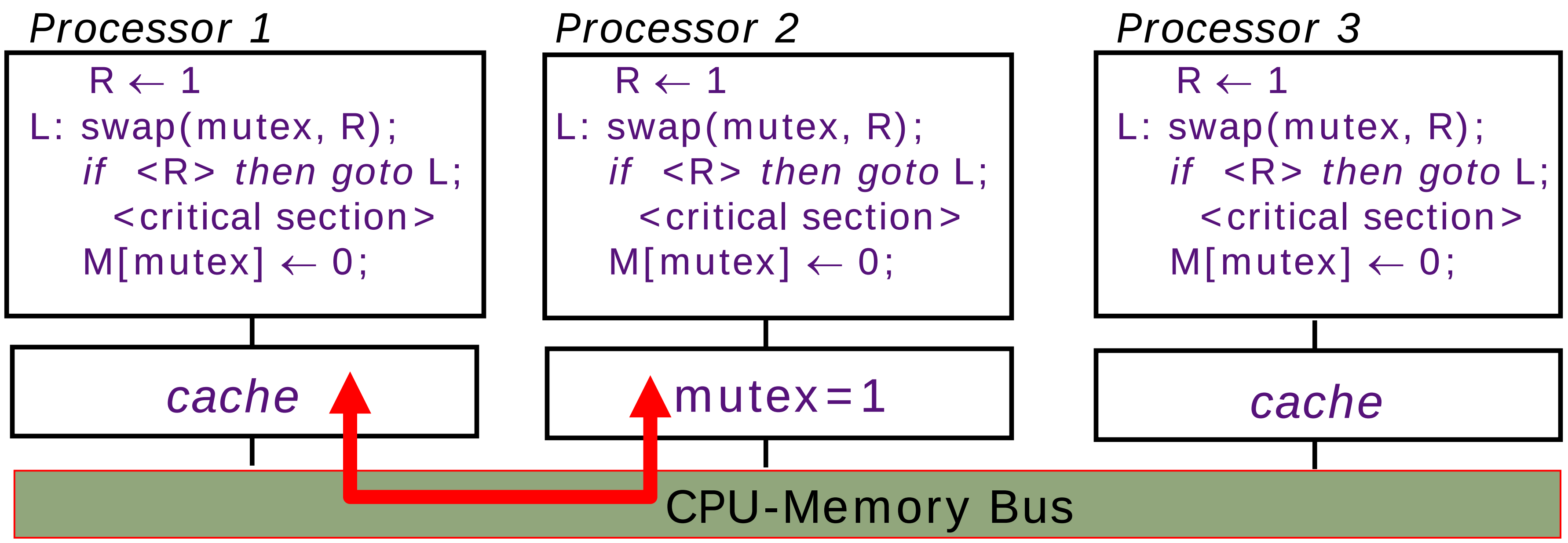
A cache block contains more than one word

Cache-coherence is done at the block-level and not word-level

Suppose M_1 writes $word_i$ and M_2 writes $word_k$ and both words have the same block address.

What can happen?

Synchronization and Caches: *Performance Issues*



Cache-coherence protocols will cause mutex to *ping-pong* between P1's and P2's caches.

Ping-ponging can be reduced by first reading the mutex location *(non-atomically)* and executing a swap only if it is found to be zero.

Performance Related to Bus occupancy

In general, a *read-modify-write* instruction requires two memory (bus) operations without intervening memory operations by other processors

In a multiprocessor setting, bus needs to be locked for the entire duration of the atomic read and write operation

$\Rightarrow$ expensive for simple buses

$\Rightarrow$ *very expensive* for split-transaction buses

modern processors use

load-reserve

store-conditional

Load-reserve & Store-conditional

Special register(s) to hold reservation flag and address, and the outcome of store-conditional

```
Load-reserve(R, a):
    <flag, adr> ← <1, a>;
    R ← M[a];
```

```
Store-conditional(a, R):
    if <flag, adr> == <1, a>
    then  cancel other procs'
          reservation on a;
          M[a] ← <R>;
          status ← succeed;
    else  status ← fail;
```

If the snooper sees a store transaction to the address in the reserve register, the reserve bit is set to 0

- Several processors may reserve 'a' simultaneously
- These instructions are like ordinary loads and stores with respect to the bus traffic

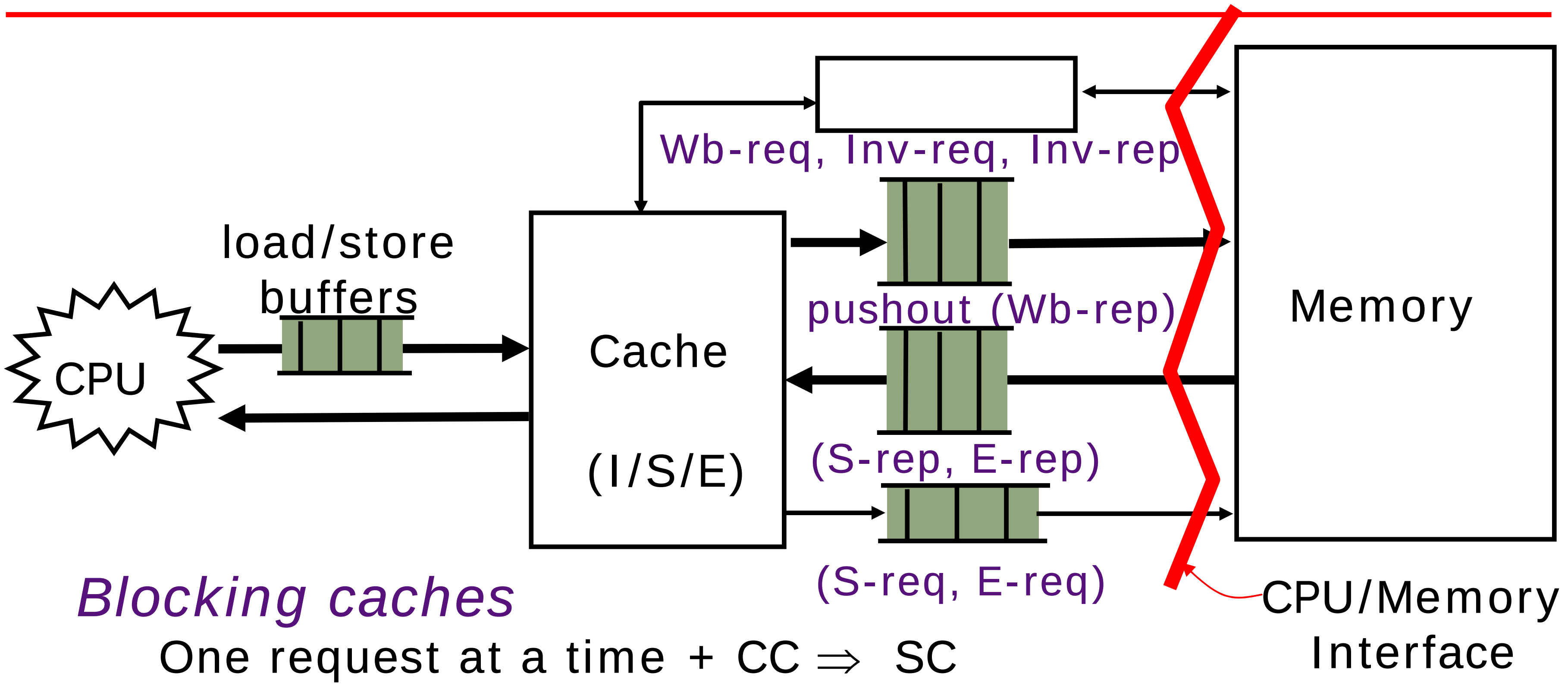
Performance:
Load-reserve & Store-conditional

The total number of memory (bus) transactions is not necessarily reduced, but splitting an atomic instruction into load-reserve & store-conditional:

- *increases bus utilization* (and reduces processor stall time), especially in split-transaction buses

- *reduces cache ping-pong effect* because processors trying to acquire a semaphore do not have to perform a store each time

Out-of-Order Loads/Stores & CC



Blocking caches

One request at a time + CC $\Rightarrow$ SC

Non-blocking caches

Multiple requests (different addresses) concurrently + CC

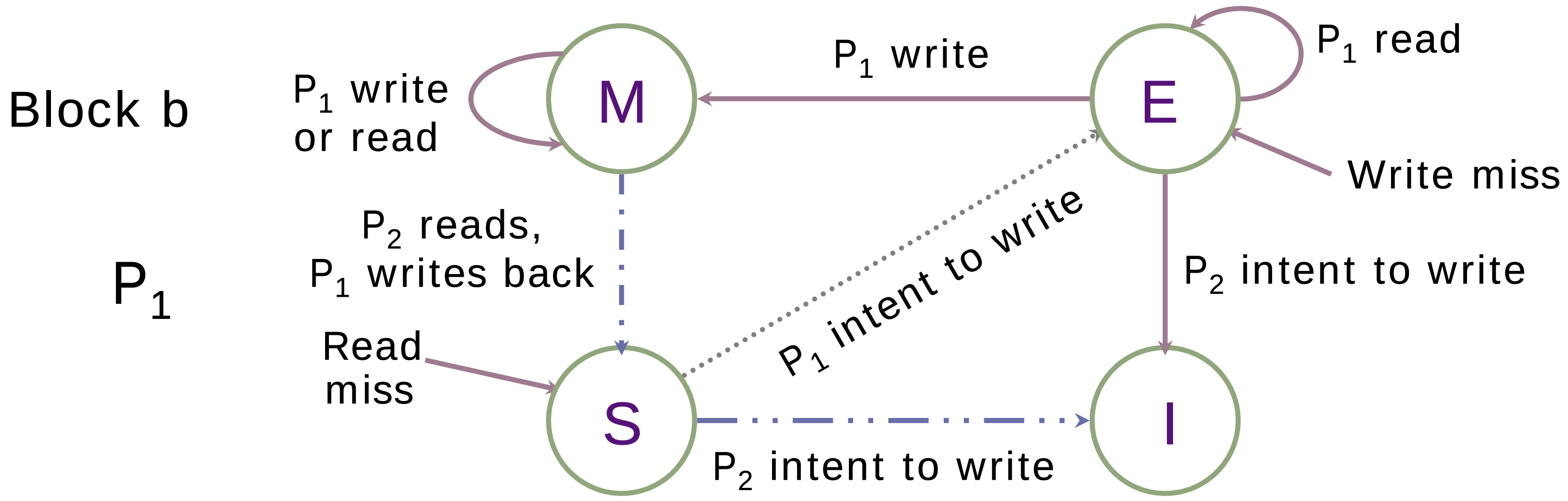
$\Rightarrow$ Relaxed memory models

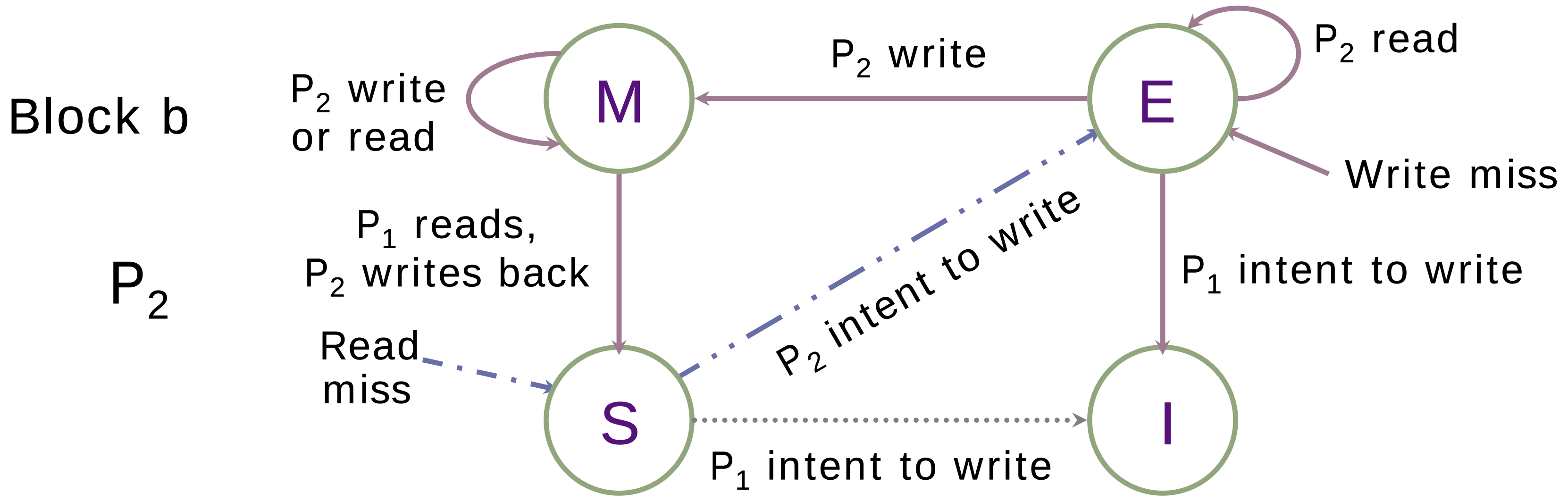
CC ensures that all processors observe the same order of loads and stores to an address *next time*

Designing a Cache Coherence Protocol

Thank you !

2 Processor Example


Block b
P_1
P_1 write or read
M
P_1 write
E
P_1 read
Write miss
P_2 reads, P_1 writes back
P_1 intent to write
P_2 intent to write
Read miss
S
I
P_2 intent to write


Block b
P_2
P_2 write or read
M
P_2 write
E
P_2 read
Write miss
P_1 reads, P_2 writes back
P_2 intent to write
P_1 intent to write
Read miss
S
I
P_1 intent to write