Sequential Consistency and Cache Coherence Protocols

Joel Emer
Computer Science and Artificial Intelligence Lab
M.I.T.

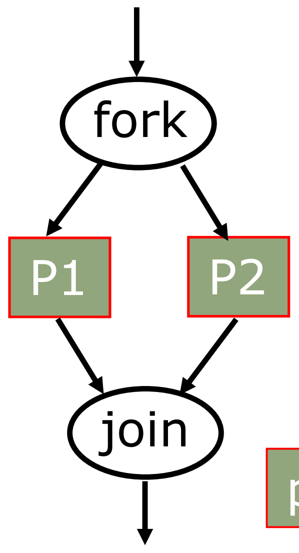
http://www.csg.csail.mit.edu/6.823

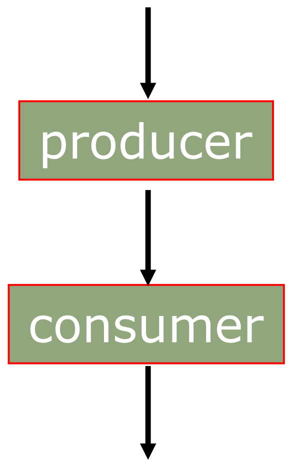
Synchronization

The need for synchronization arises whenever there are parallel processes in a system

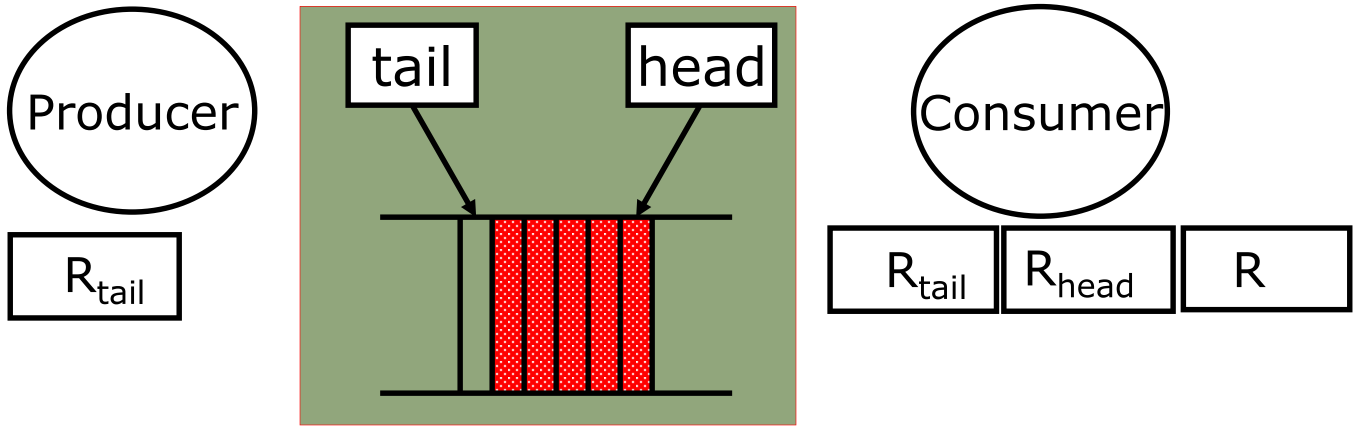
(even in a uniprocessor system)

- *Forks and Joins: A* parallel process may want to wait until several events have occurred
- *Producer-Consumer:* A consumer process must wait until the producer process has produced data
- *Exclusive use of a resource:* Operating system has to ensure that only one process uses a resource at a given time





A Producer-Consumer Example



Producer posting Item x:

```
        Load $R_{tail}$, (tail)
        Store ($R_{tail}$), x
        $R_{tail}$=$R_{tail}$+1
        Store tail, $R_{tail}$
```

Consumer:

```
        Load $R_{head}$, (head)
spin:   Load $R_{tail}$, (tail)
        if $R_{head}$==$R_{tail}$ goto spin
        Load R, ($R_{head}$)
        $R_{head}$=$R_{head}$+1
        Store head, $R_{head}$
        process(R)
```

The program is written assuming instructions are executed in order.

Problems?

A Producer-Consumer Example
continued

Producer posting Item x:

```
        Load $R_{tail}$, (tail)
1       Store ($R_{tail}$), x
        $R_{tail}$=$R_{tail}$+1
2       Store tail, $R_{tail}$
```

Consumer:

```
        Load $R_{head}$, (head)
spin:   Load $R_{tail}$, (tail)                3
        if $R_{head}$==$R_{tail}$ goto spin
        Load R, ($R_{head}$)                   4
        $R_{head}$=$R_{head}$+1
        Store head, $R_{head}$
        process(R)
```

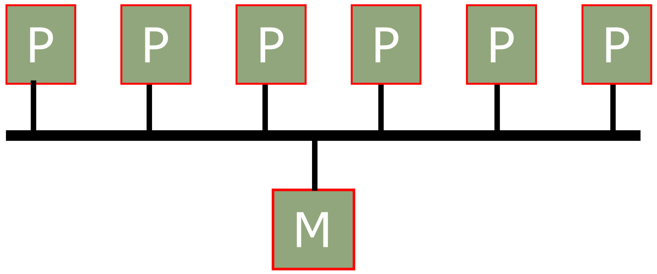
Can the tail pointer get updated before the item x is stored?

Programmer assumes that if 3 happens after 2, then 4 happens after 1.

Problem sequences are:

2, 3, 4, 1
4, 1, 2, 3

Sequential Consistency
A Memory Model

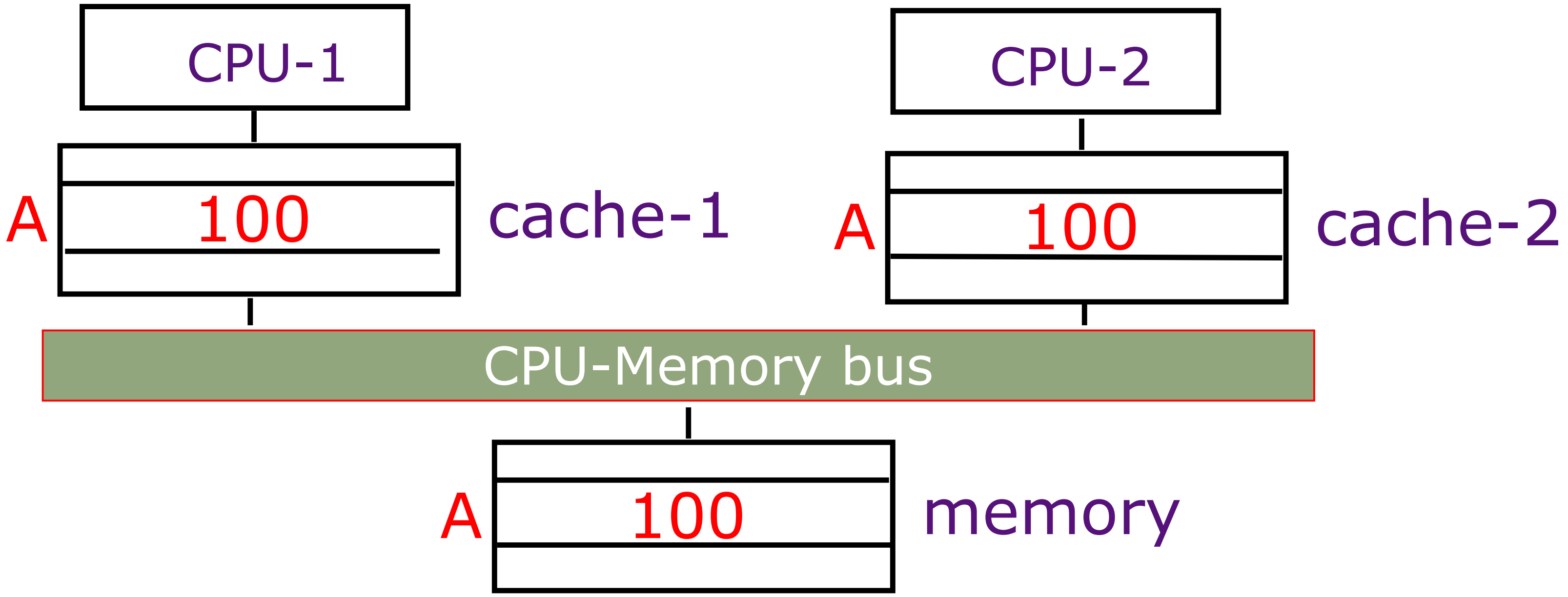


" A system is *sequentially consistent* if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in the order specified by the program"

Leslie Lamport

Sequential Consistency =
arbitrary *order-preserving interleaving*
of memory references of sequential programs

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?

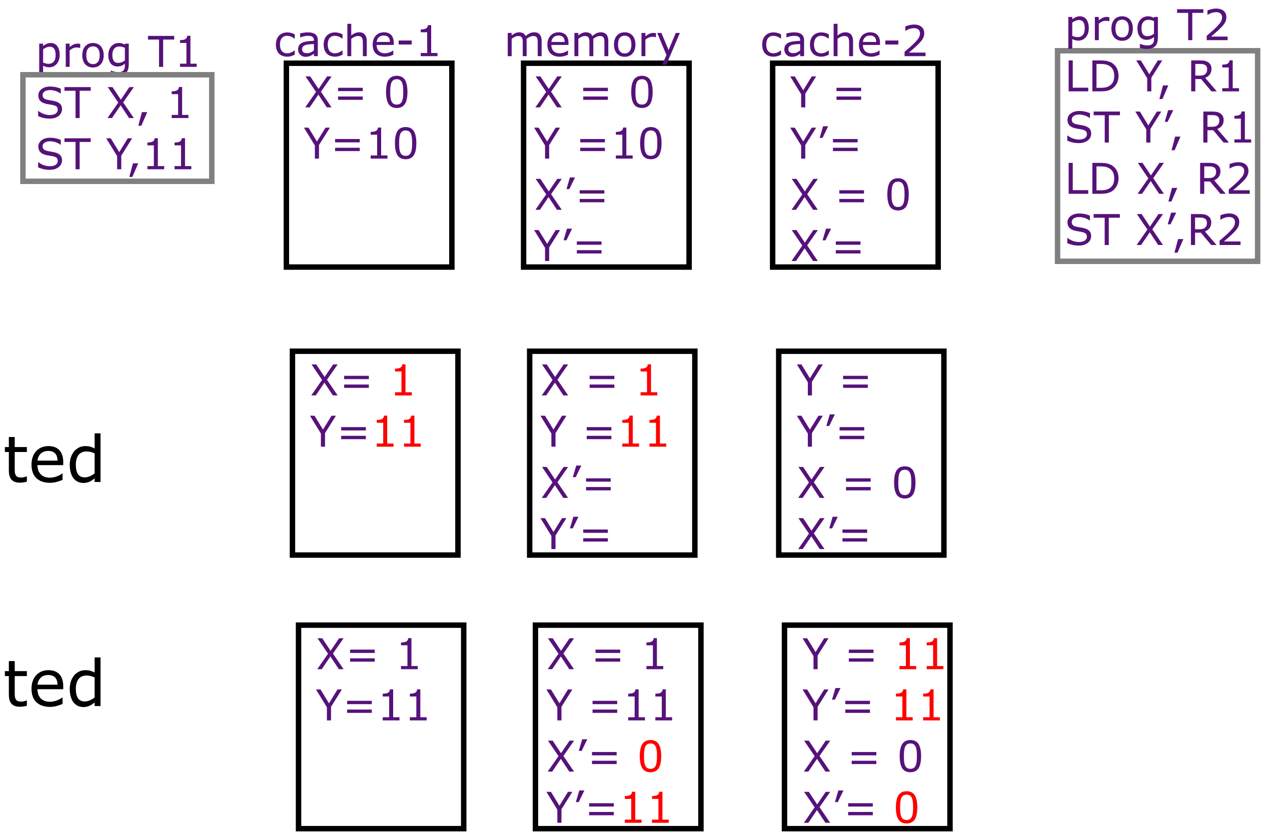
What is the view of shared memory for programming?

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

Motivation: We can do without locks -- SC is sufficient for writing producer-consumer and mutual exclusion codes (e.g., Dekker)

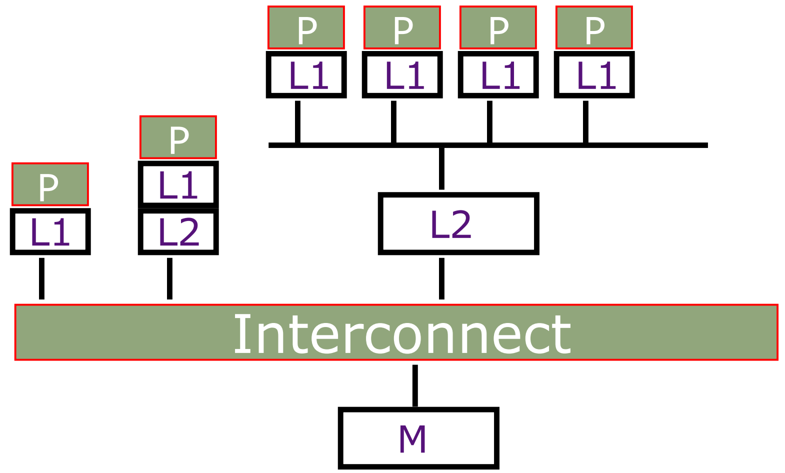
Problem: SC requires all processors to see writes occur in the same order, but multiple copies of a location in various caches can cause this to be violated.

To meet the ordering requirement it is sufficient for hardware to ensure:

- 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* in all other caches *before* the write is performed, *or*

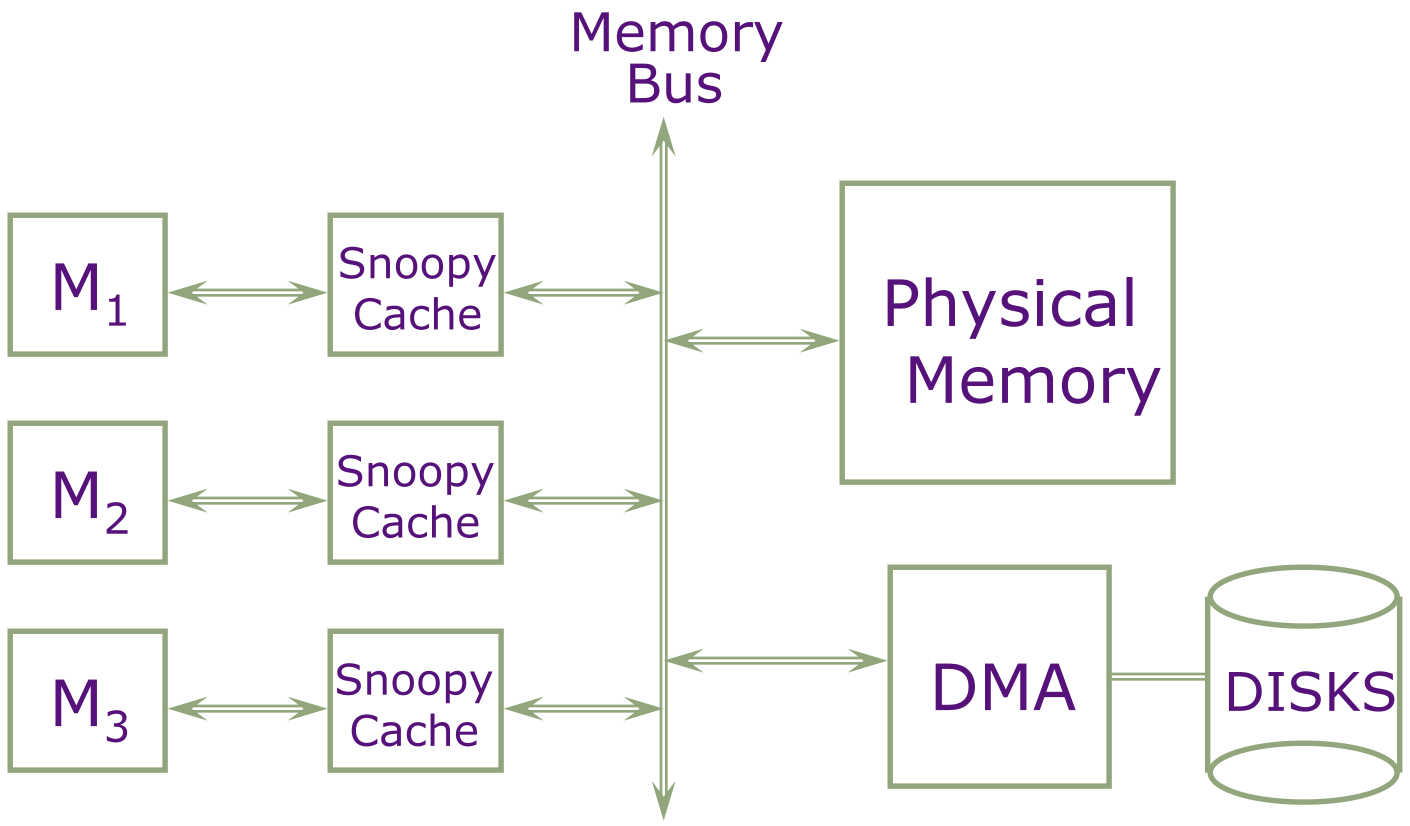
the address is *updated* in all other caches *after* the write is performed

read request:

if a dirty copy is found in some cache, that is the value that must be used, e.g., by doing a write-back and reading the memory or forwarding that dirty value directly to the reader.

We will focus on Invalidation protocols as opposed to Update protocols

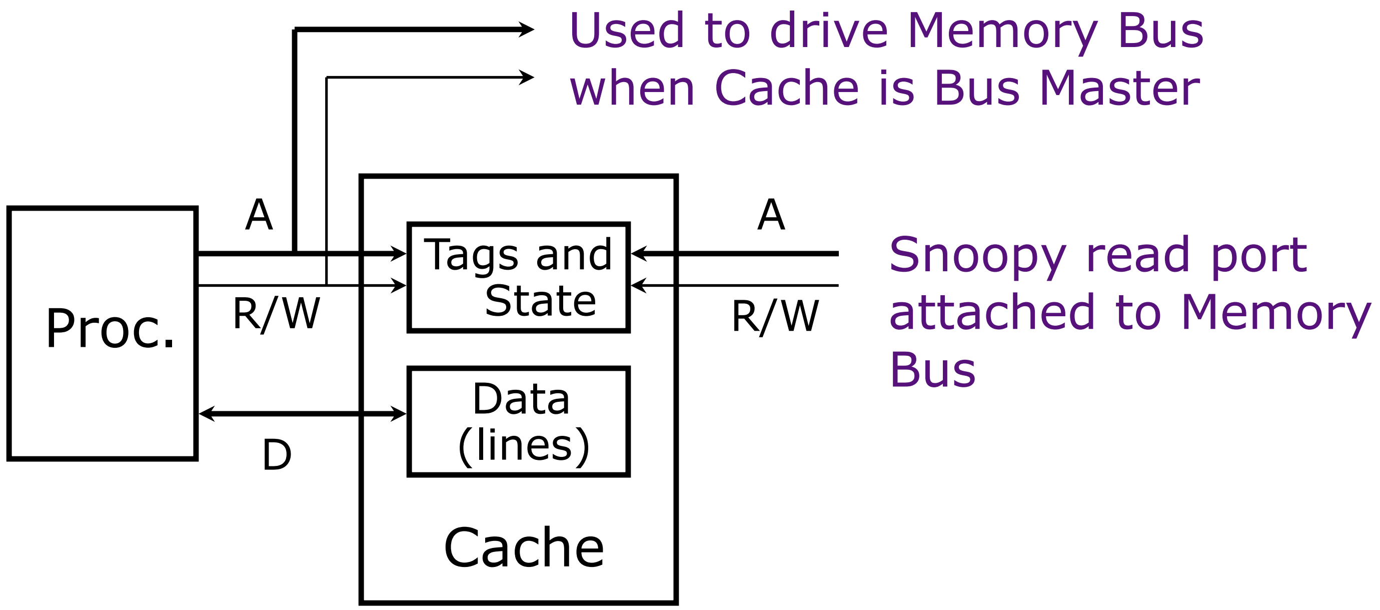
Shared Memory Multiprocessor



Watch (snoop on) bus to keep all processors' view of memory coherent

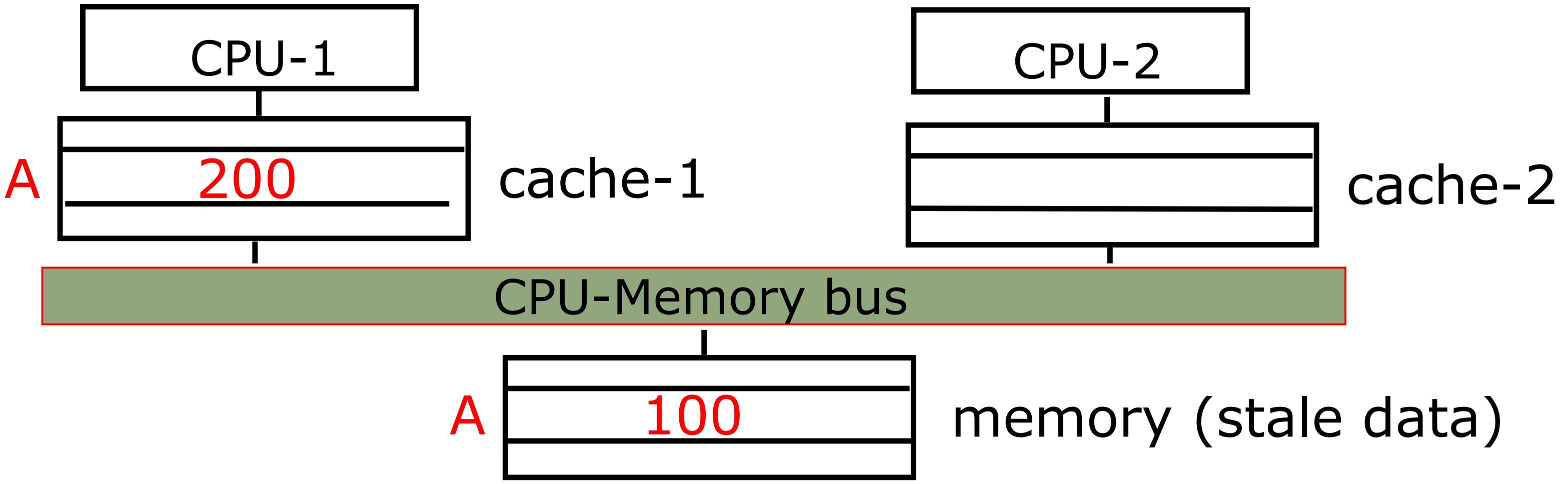
Snoopy Cache *Goodman 1983*

- Idea: Have the cache watch (or snoop upon) data transfers, and then "do the right thing". Thus, memory operations are atomic with respect to all the caches.


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

Note: Snoopy cache tags have increased demand – often they are dual-ported

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 data
- The memory may respond to the request also!

Does memory know it has stale data?

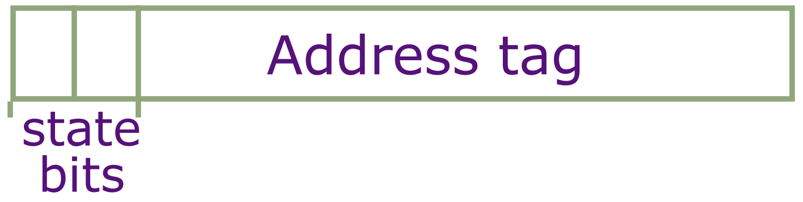
No, Cache-1 needs to intervene through memory controller to supply correct data to cache-2

Snoopy Cache Actions

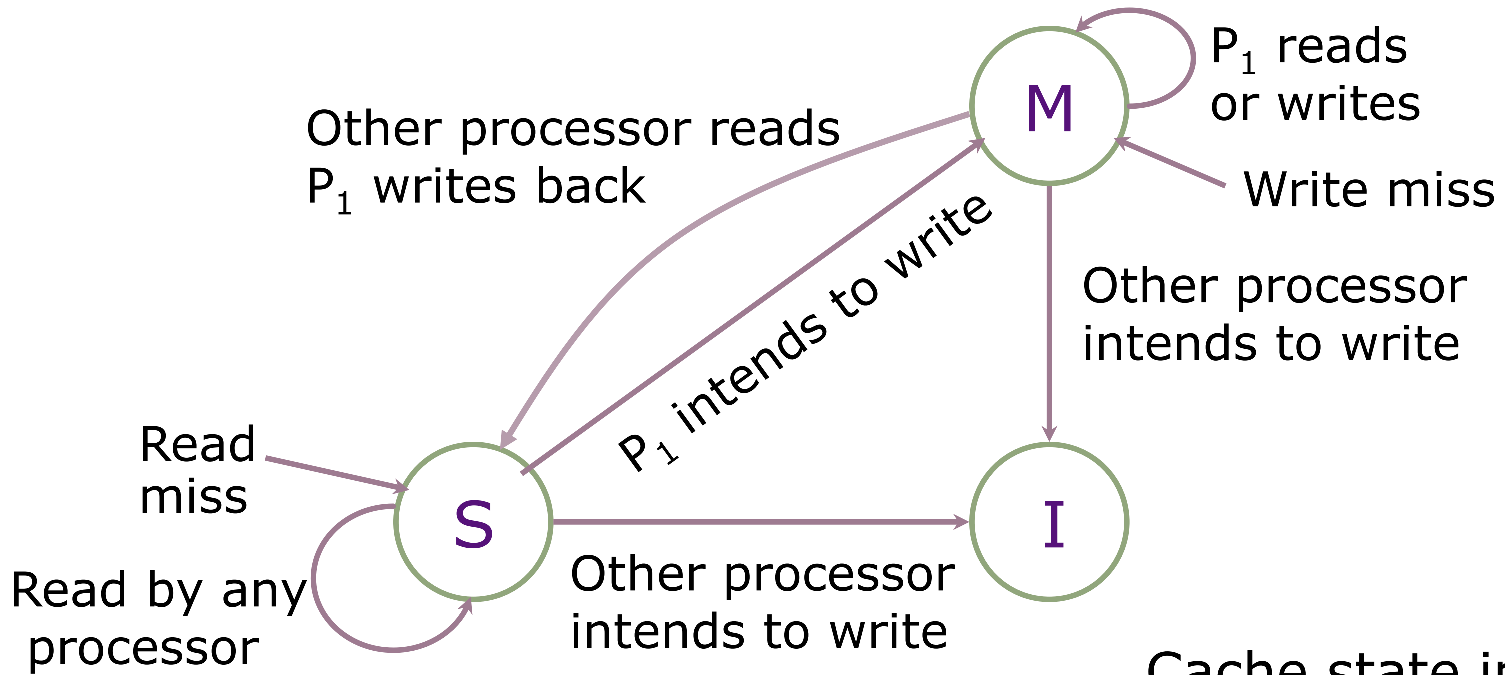
Observed Bus Cycle	Cache State	Cache Action
Remote Read	Address not cached Cached, unmodified Cached, modified	No action No action Cache Intervenes
Remote Write	Address not cached Cached, unmodified Cached, modified	No action Cache Purges Copy ??????

Cache State Transition Diagram
The MSI protocol

Each cache line has a tag



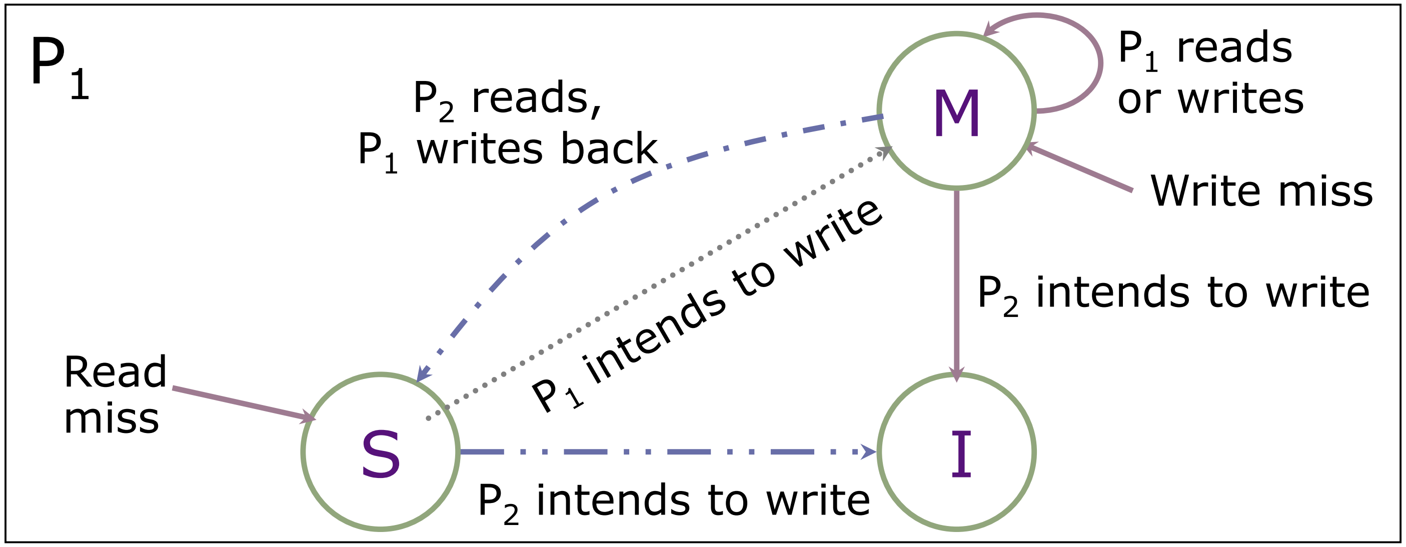
M: Modified
S: Shared
I: Invalid

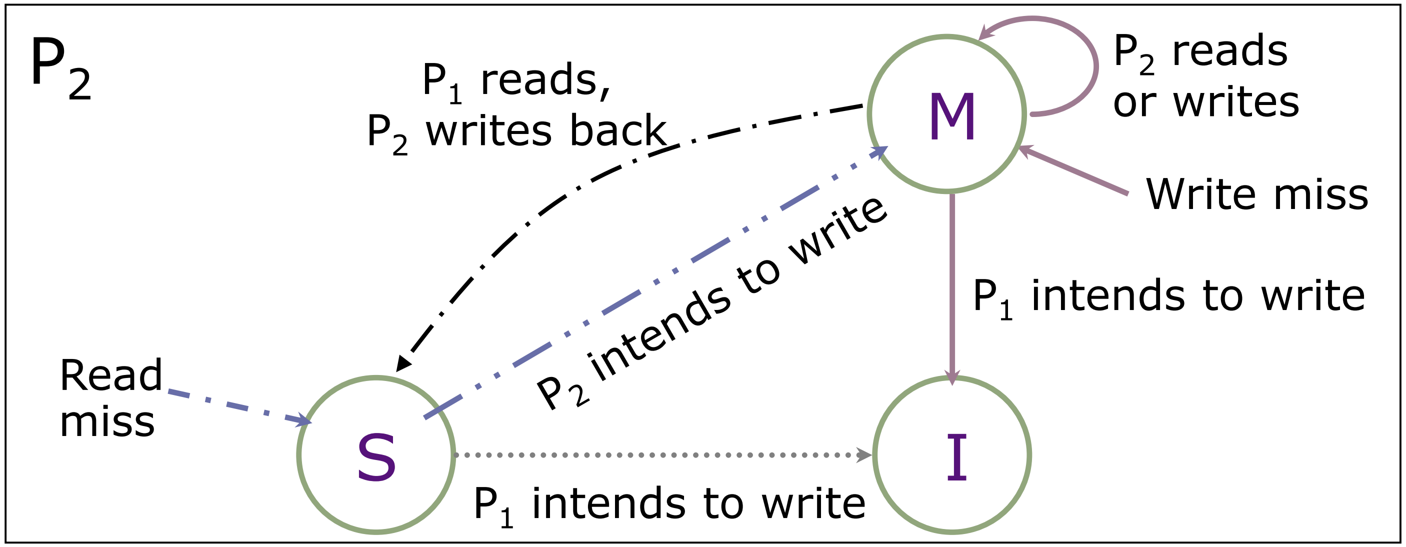


Cache state in processor P_1

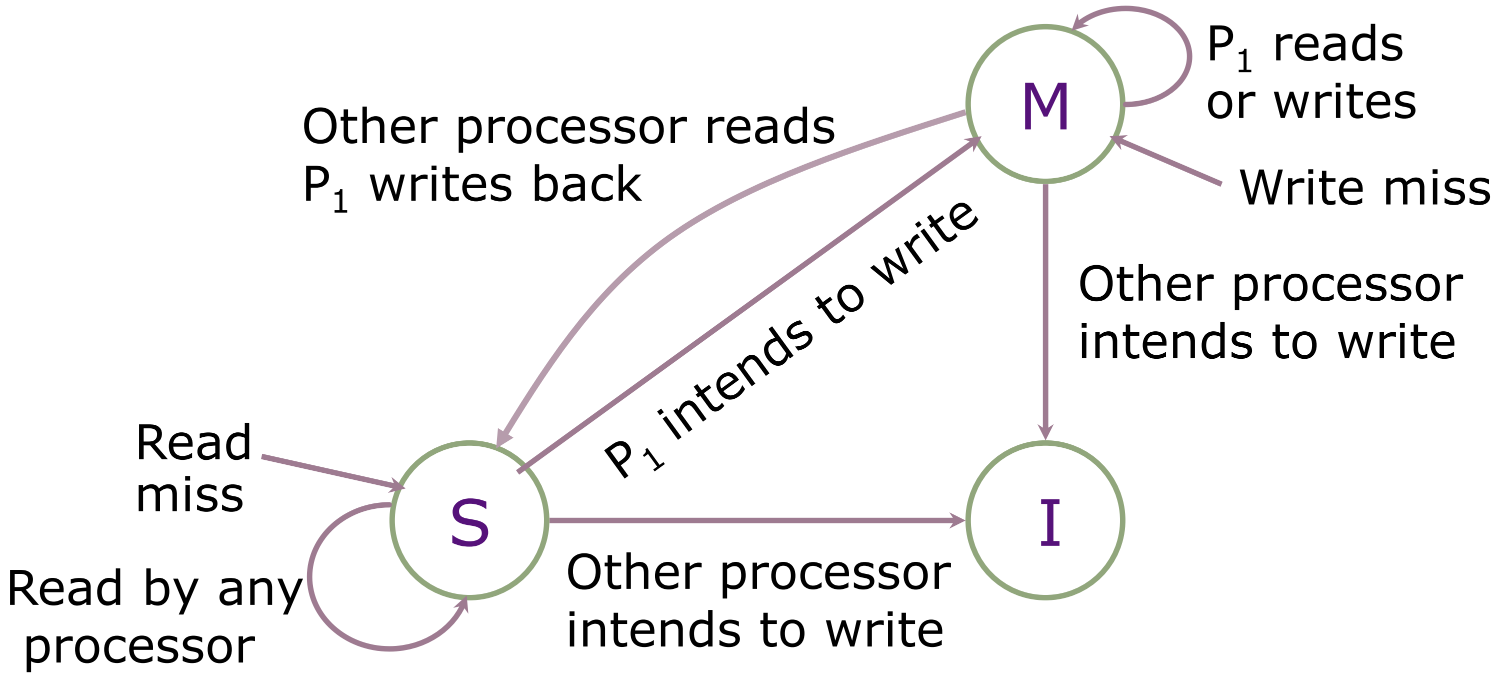
2 Processor Example

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

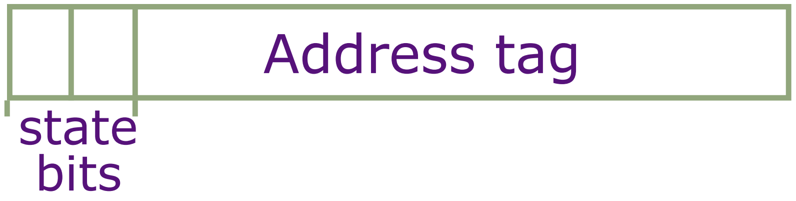

M
S
I
P₁ reads or writes
Write miss
Other processor reads P₁ writes back
P₁ intends to write
Other processor intends to write
Read miss
Read by any processor
Other processor intends to write

- 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

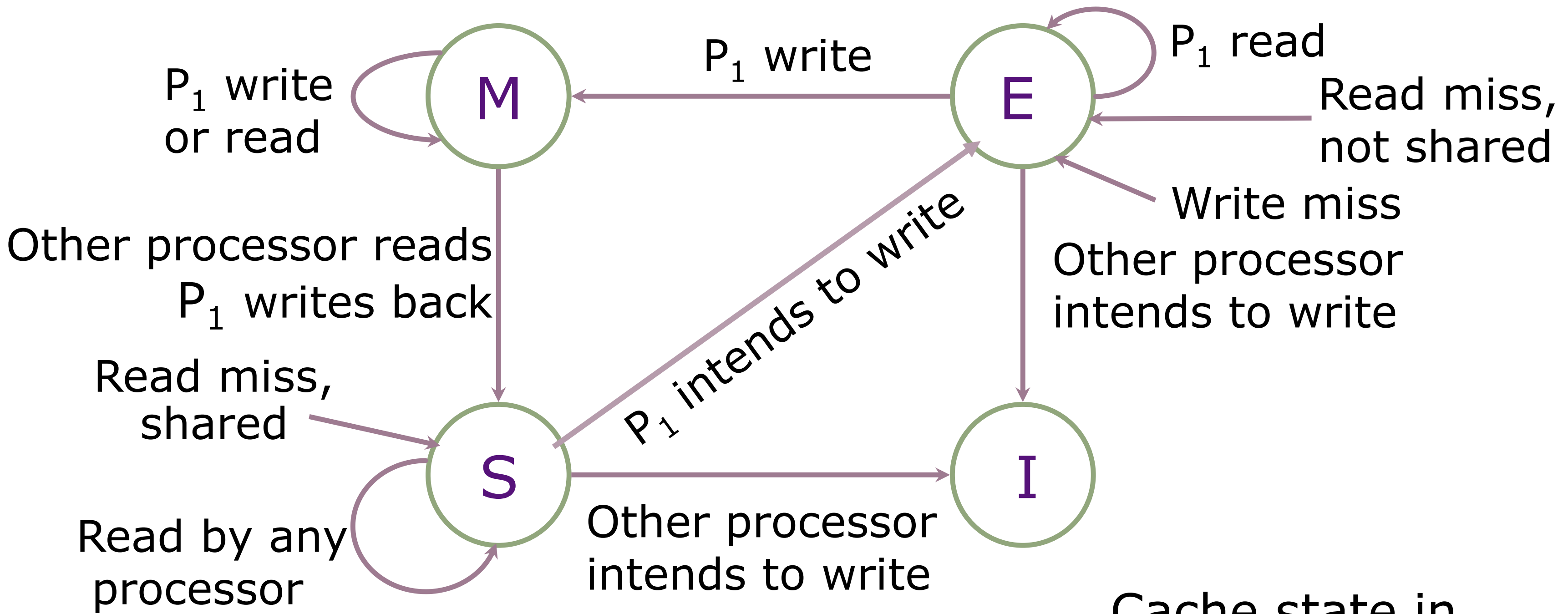
MESI: An Enhanced MSI protocol

increased performance for private data

Each cache line has a tag

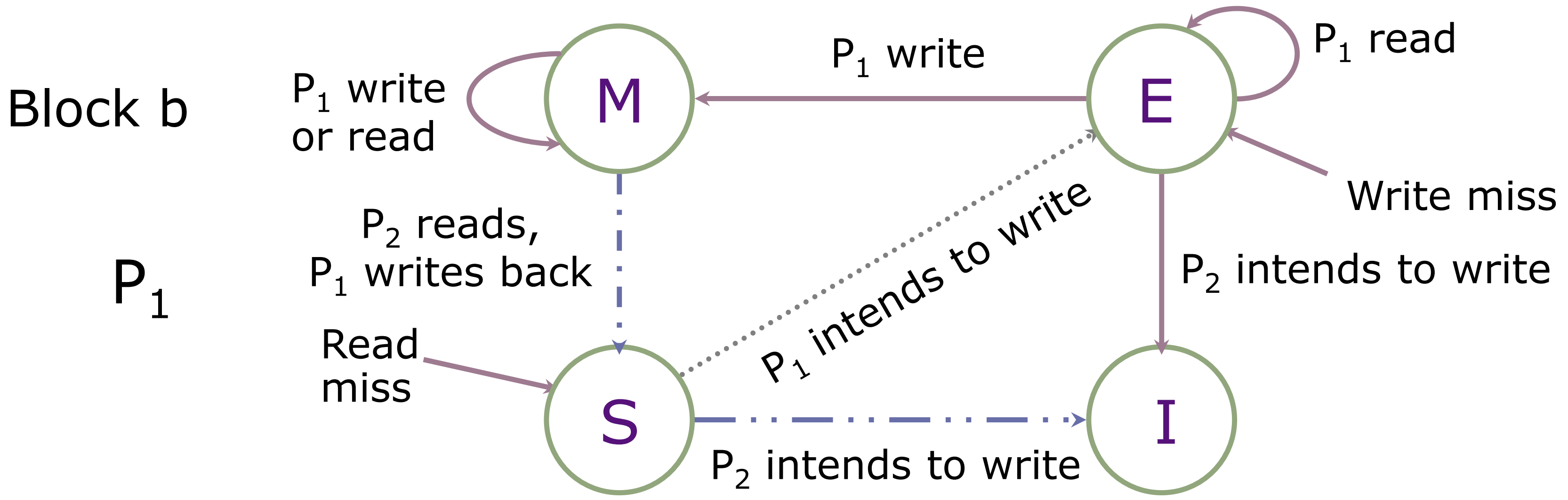


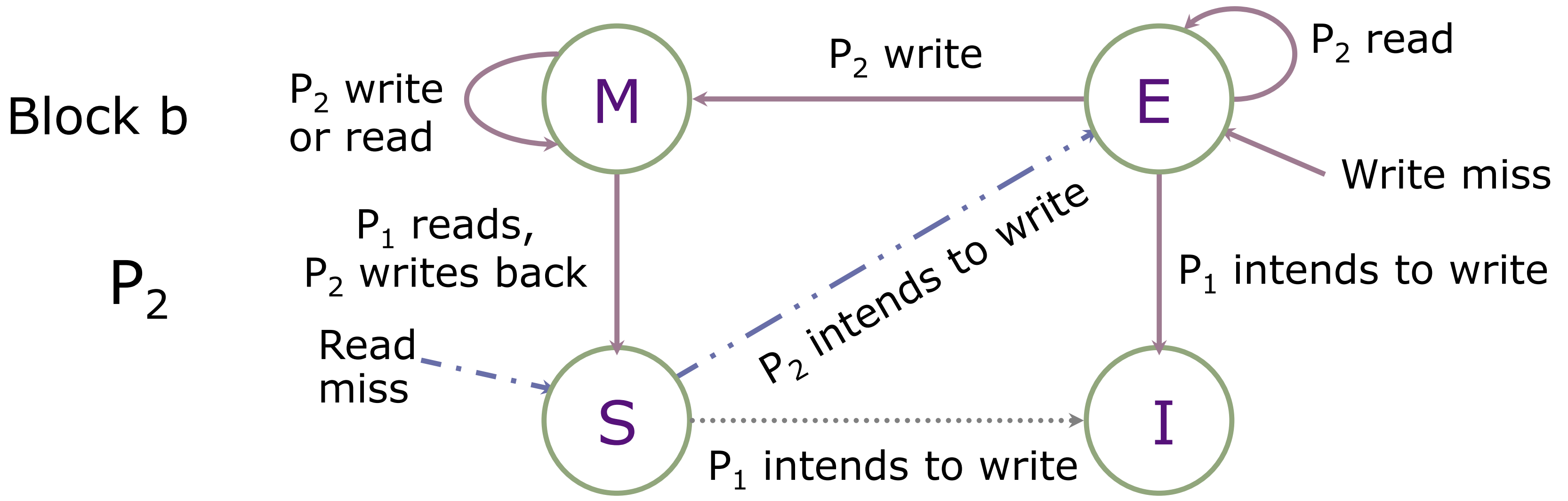
M: Modified Exclusive
E: Exclusive, unmodified
S: Shared
I: Invalid



Cache state in processor P_1

2 Processor Example





CC and False Sharing

Performance Issue - 1

state	blk addr	data0	data1	...	dataN

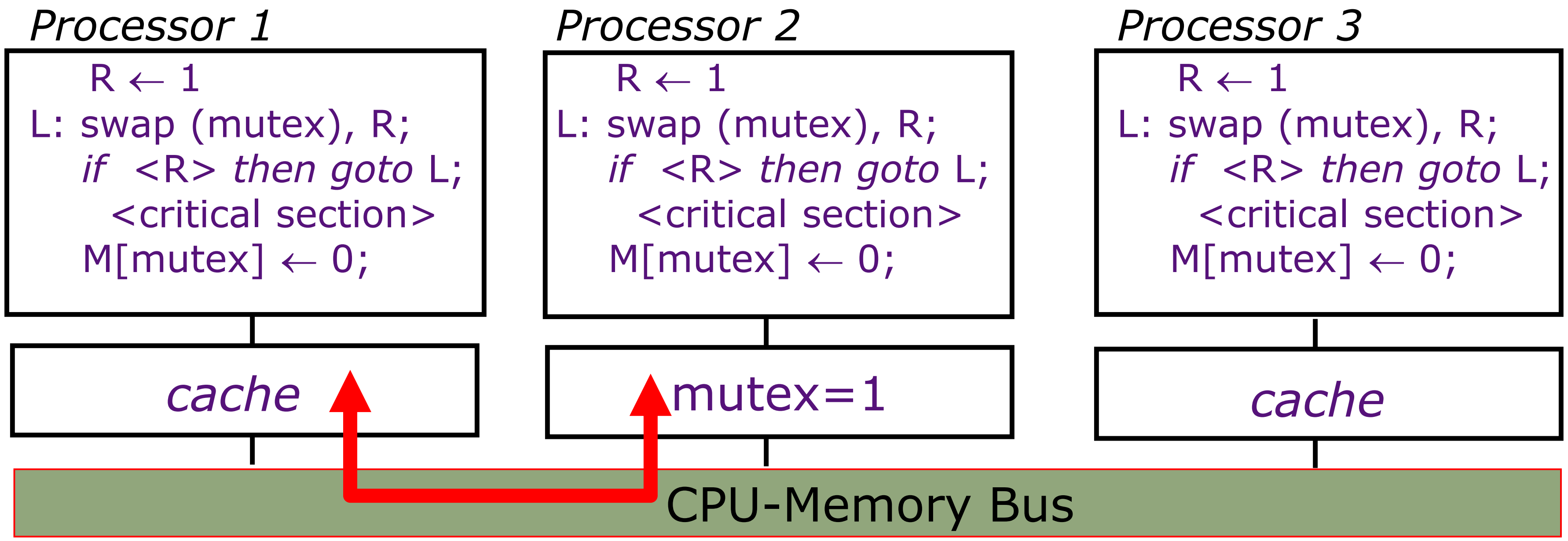
A cache block contains more than one word and cache-coherence is done at the block-level and not word-level

Suppose P_1 writes word_i and P_2 writes word_k and both words have the same block address.

What can happen? The block may be invalidated (ping pong) many times unnecessarily because the addresses are in same block.

CC and Synchronization

Performance Issue - 2



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.

CC and Bus Occupancy
Performance Issue - 3

In general, an atomic *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

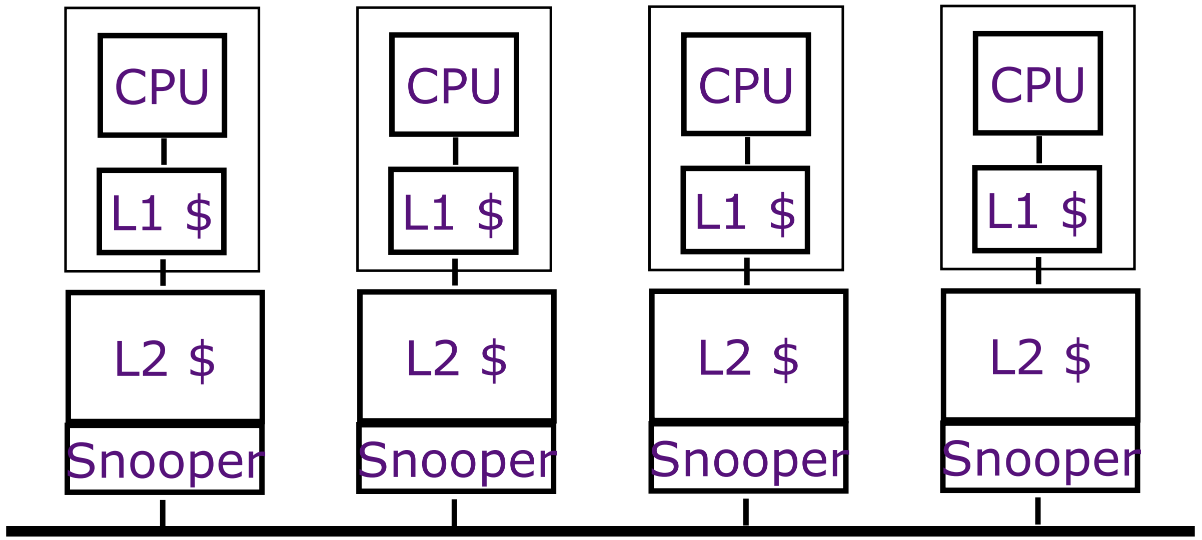
- Several processors may reserve 'a' simultaneously
- These instructions are like ordinary loads and stores with respect to the bus traffic

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 stores each time

2-Level On-chip Caches



Typically
L1 << L2

- *Inclusion property:* entries in L1 must be in L2
  invalidation in L2 ⇒ invalidation in L1

- Does snooping on L2 affect CPU-L1 bandwidth?
  - yes -- to check if a dirty copy is stored in L1
- How can this be avoided?
  - Write-through L1 cache

Implementing SC

1. The memory operations of each individual processor appear to all processors in the order the requests are made to the memory.
   - *Provided by cache coherence*, which ensures that all processors observe the same order of loads and stores to an address

2. Any execution is the same as if the operations of all the processors were executed in some sequential order
   - Provided by enforcing a dependence between each memory operation and the following one.

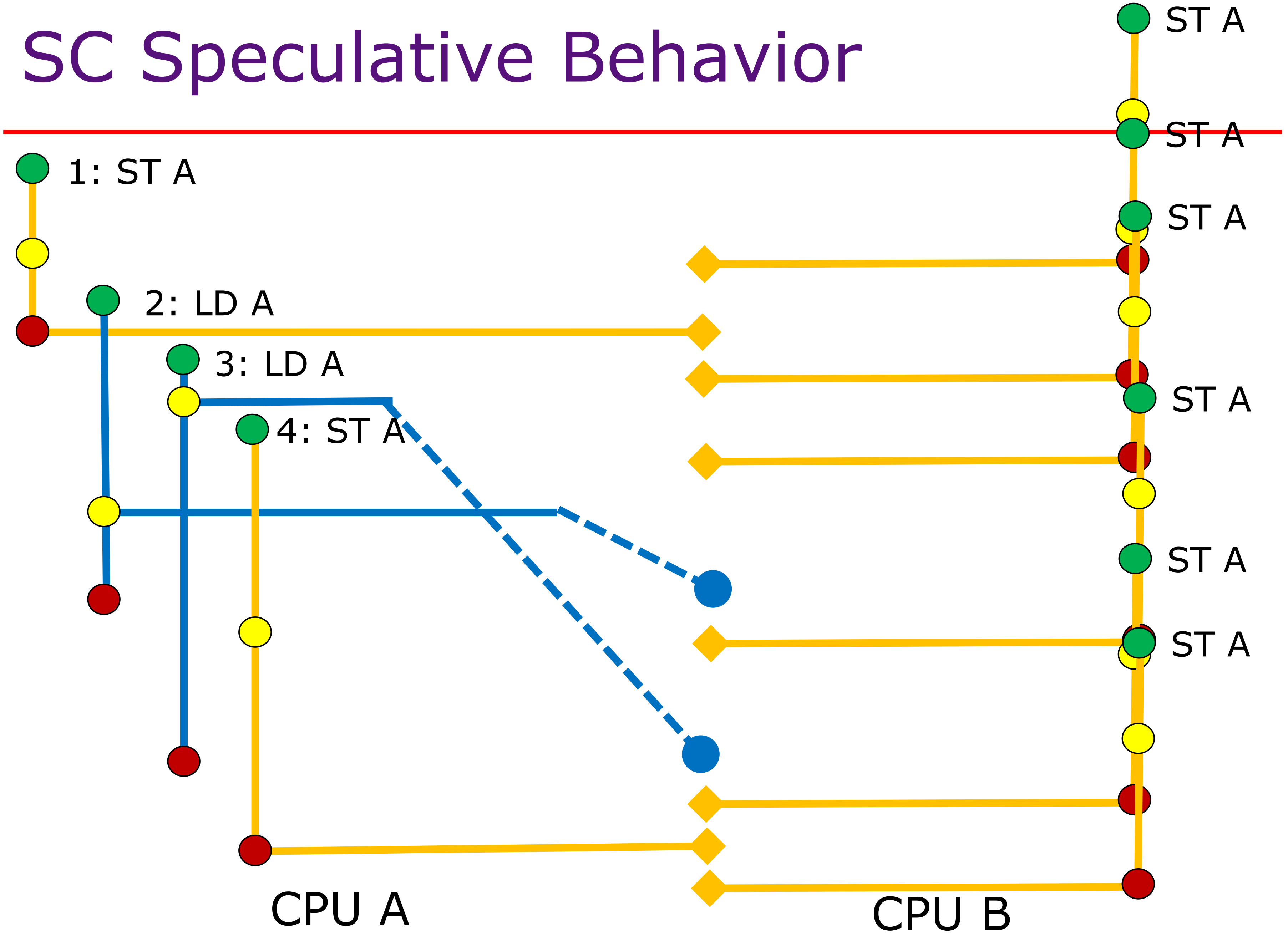
SC Data Dependence

- *Stall*
  - *Use in-order execution with blocking cache*
    - *Cache coherence plus allowing a processor to have only one request in flight at a time will provide SC*
- *Change architecture $\Rightarrow$ Relaxed memory models*
  - *Use OOO and non-blocking caches*
    - *Cache coherence and allowing multiple requests (different addresses) concurrently gives high performance, then add fence operations to force ordering when needed*
- *Speculate…*

Sequential Consistency Speculation

- Local load-store ordering uses standard OOO mechanism
- Globally <u>non-speculative</u> stores
  - Stores execute at commit -> stores are in-order!
- Globally <u>speculative</u> loads
  - **Guess** at issue that the memory location used by a load will not change between issue and commit of the instruction
    - this is equivalent to loads happening in-order at commit
  - **Check** at commit by remembering all loads addresses starting at issue and watching for writes to that location.
  - **Data Management** for rollback relies on the basic out-of-order speculative data management used for uni-processor rollback and instruction re-execution.

SC Speculative Behavior



Next lecture:

How to design a cache coherence protocol