# CS 152 Computer Architecture and Engineering

## Lecture 18: Snoopy Caches

Dr. George Michelogiannakis
EECS, University of California at Berkeley
CRD, Lawrence Berkeley National Laboratory

**http://inst.eecs.berkeley.edu/~cs152**

# Administrivia

- Lab 4 due now
- PS 5 due next week Wednesday (20th)
- Lecture 20 will be on datacenters
- Quiz 4 and PS 3 will be returned tomorrow

# Last time in Lecture 17

Two kinds of synchronization between processors:

- Producer-Consumer
  - Consumer must wait until producer has produced value
  - Software version of a read-after-write hazard
- Mutual Exclusion
  - Only one processor can be in a critical section at a time
  - Critical section guards shared data that can be written

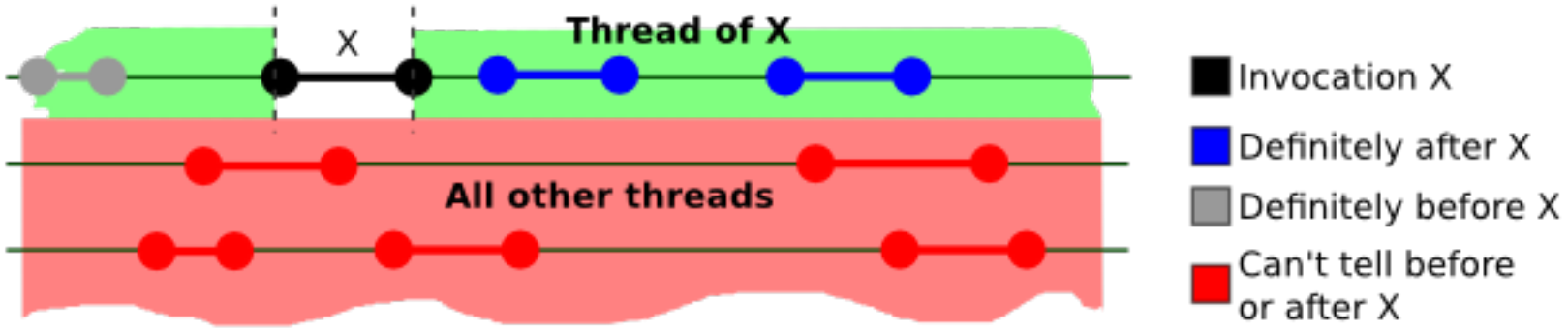
- Producer-consumer synchronization implementable with just loads and stores, but need to know ISA's *memory model!*
- Mutual-exclusion can also be implemented with loads and stores, but tricky and slow, so ISAs add atomic read-modify-write instructions to implement locks

# Sequential Consistency

Cost:

Prevents aggressive compiler reordering optimizations

Constrains hardware utilization (e.g., store buffer)



# Sequential Consistency

Sequential consistency imposes more memory ordering constraints than those imposed by uniprocessor program dependencies (⟶)

*What are these in our example ?*

T1:

Store (X), 1 *(X = 1)*

Store (Y), 11 *(Y = 11)*

T2:

Load $R_1$, (Y)

Store (Y'), $R_1$ *(Y'= Y)*

Load $R_2$, (X)

Store (X'), $R_2$ *(X'= X)*

⟶ additional SC requirements

# Load-reserve & Store-conditional

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

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

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

```
try:    Load-reserve 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-conditional (head), R_head
        if (status==fail) goto try
        process(R)
```

# Performance of Locks

Blocking atomic read-modify-write instructions

*e.g., Test&Set, Fetch&Add, Swap*

vs

Non-blocking atomic read-modify-write instructions

*e.g., Compare&Swap,*

*Load-reserve/Store-conditional*

vs

Protocols based on ordinary Loads and Stores

*Performance depends on several interacting factors:*

degree of contention,

caches,

out-of-order execution of Loads and Stores

*later ...*

# Amdahl's Law

Begins with Simple Software Assumption (Limit Arg.)

Fraction F of execution time perfectly parallelizable

No Overhead for Scheduling Communication, Synchronization, etc.

F is the Parallel Part

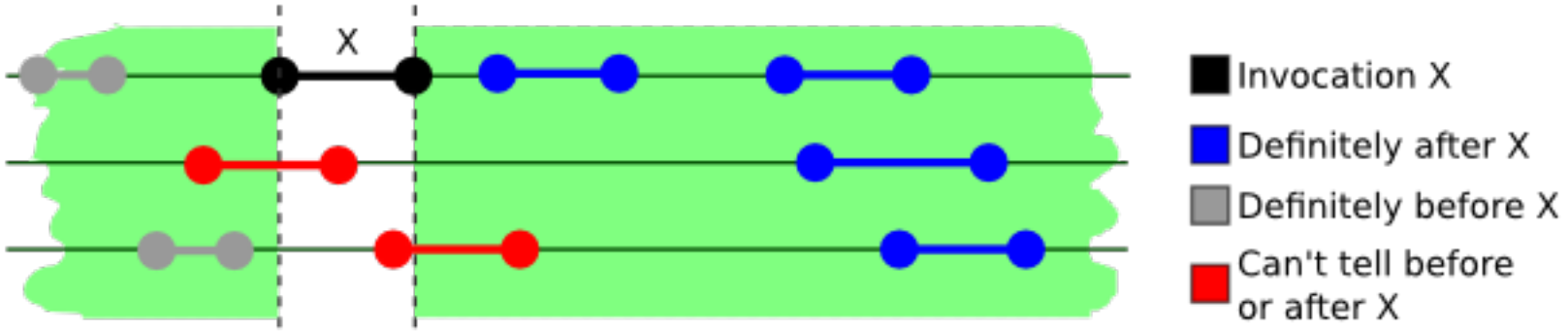
Fraction 1 – F Completely Serial

Time on 1 core = (1 – F) / 1 + F / 1 = 1

**Time on N cores = (1 – F) / 1 + F / N**

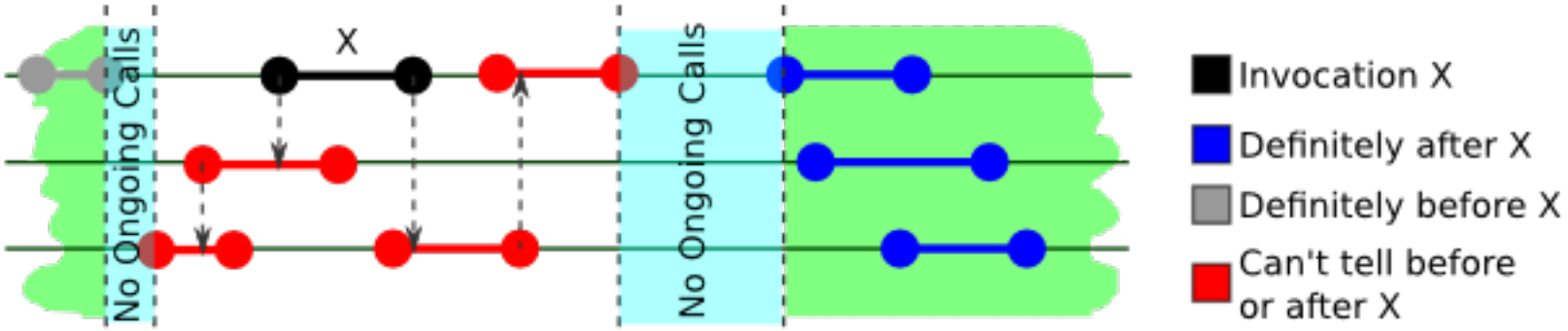
# Strong Consistency

An execution is strongly consistent (linearizable) if the method calls can be correctly arranged retaining the mutual order of calls that do not overlap in time, regardless of what thread calls them.

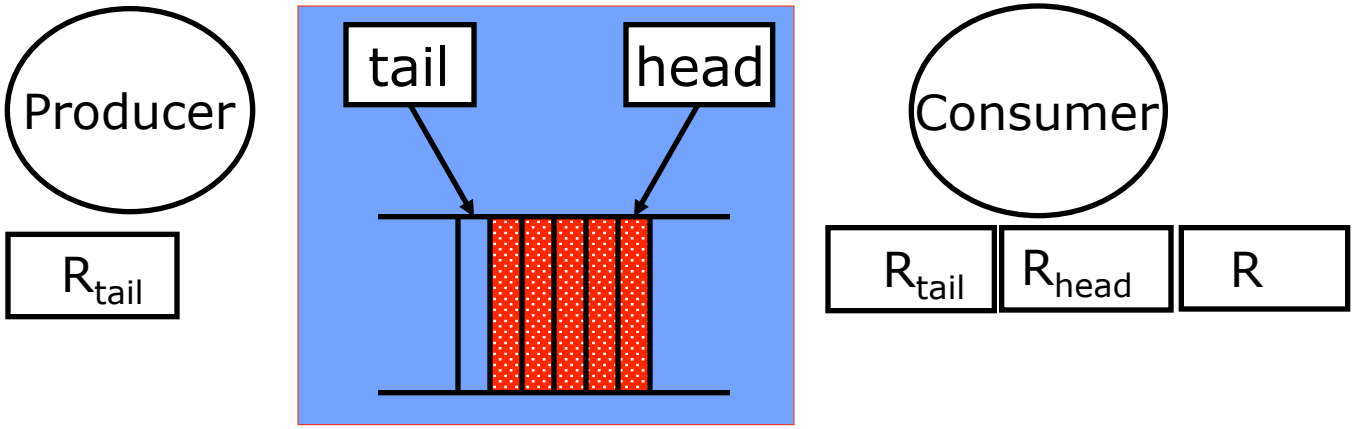


# Quiescent Consistency

An execution is quiescently consistent if the method calls can be correctly arranged retaining the mutual order of calls separated by quiescence, a period of time where no method is being called in any thread.



# Relaxed Memory Models Need Fences




Producer posting Item x:

```
Load R_tail, (tail)
Store (R_tail), x
Membar_SS
R_tail=R_tail+1
Store (tail), R_tail
```

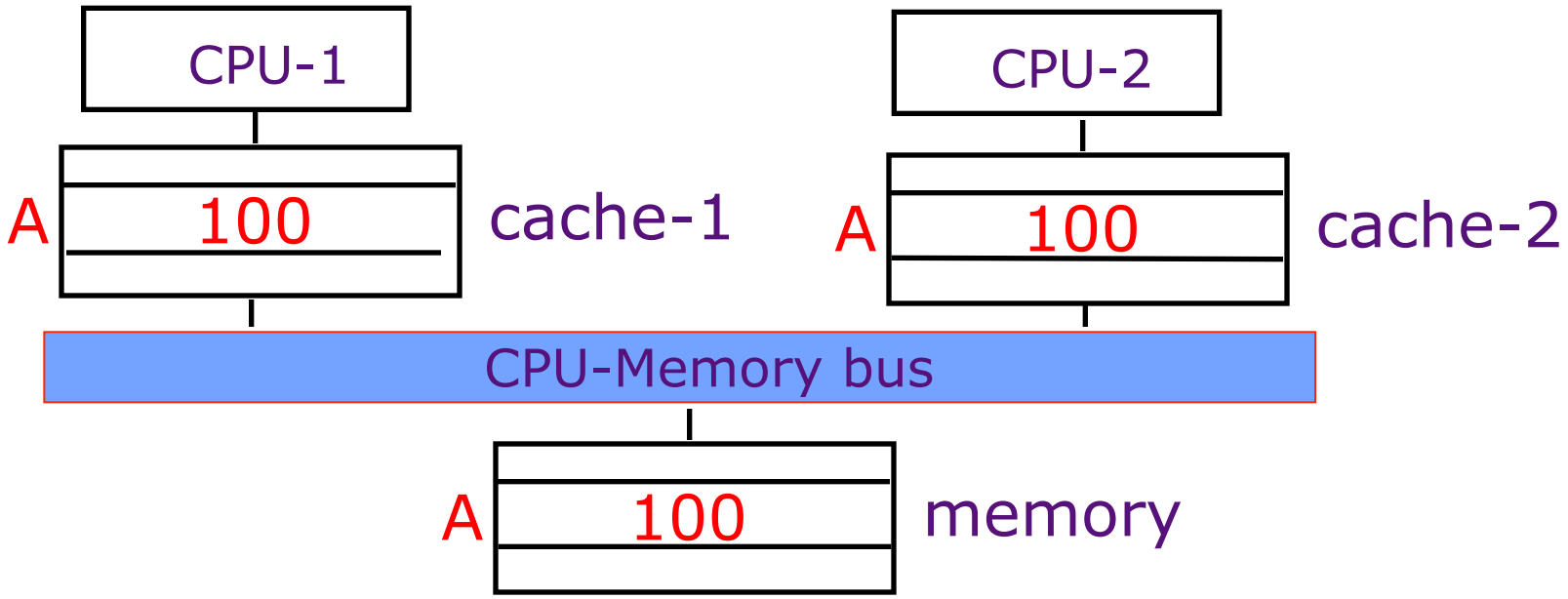
*ensures that tail ptr is not updated before x has been stored*

Consumer:

```
        Load R_head, (head)
spin:   Load R_tail, (tail)
        if R_head==R_tail goto spin
        Membar_LL
        Load R, (R_head)
        R_head=R_head+1
        Store (head), R_head
        process(R)
```

*ensures that R is not loaded before x has been stored*

# Memory Coherence in SMPs




Suppose CPU-1 updates A to 200.

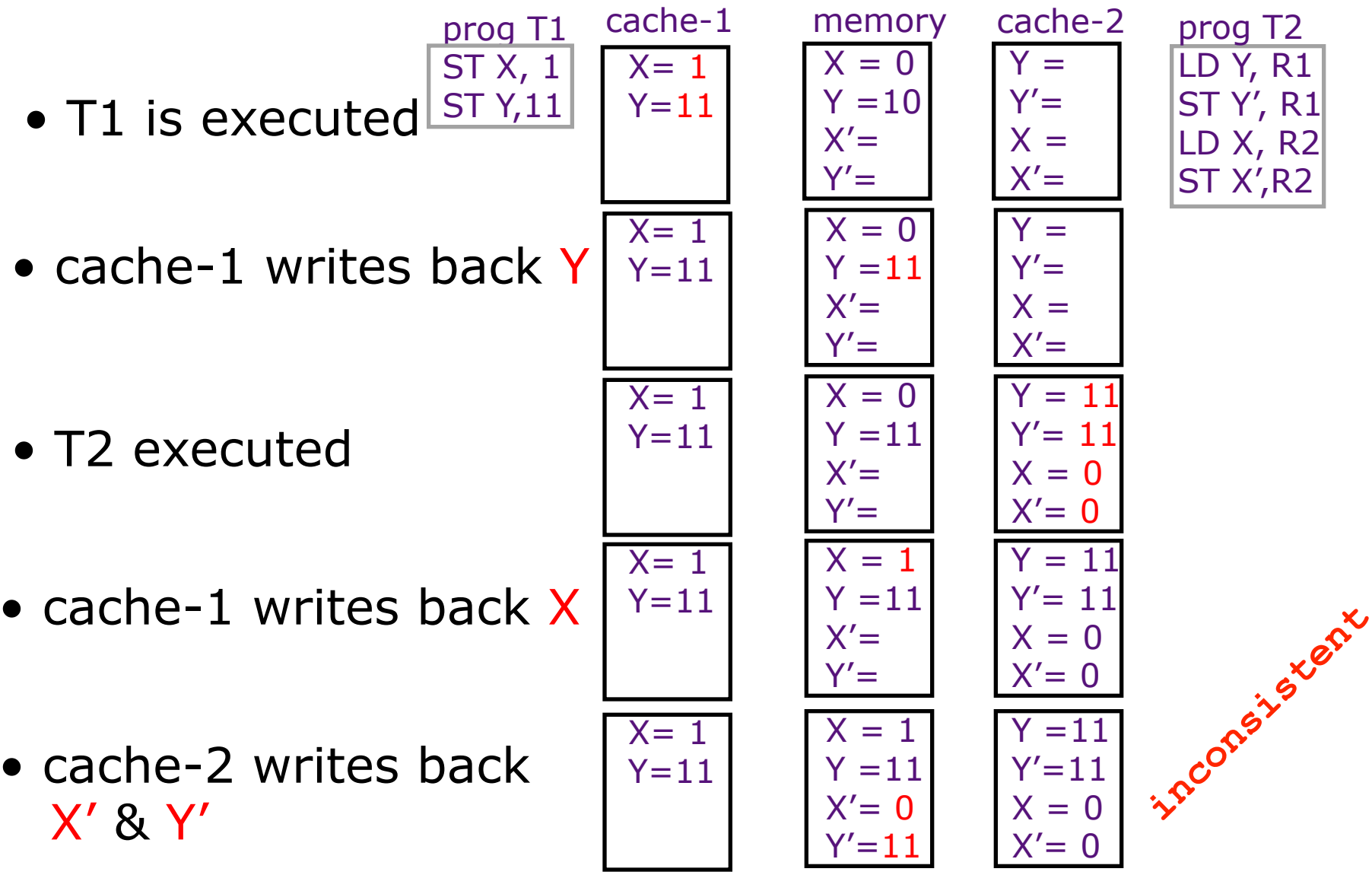
*write-back:* memory and cache-2 have stale values

*write-through:* cache-2 has a stale value

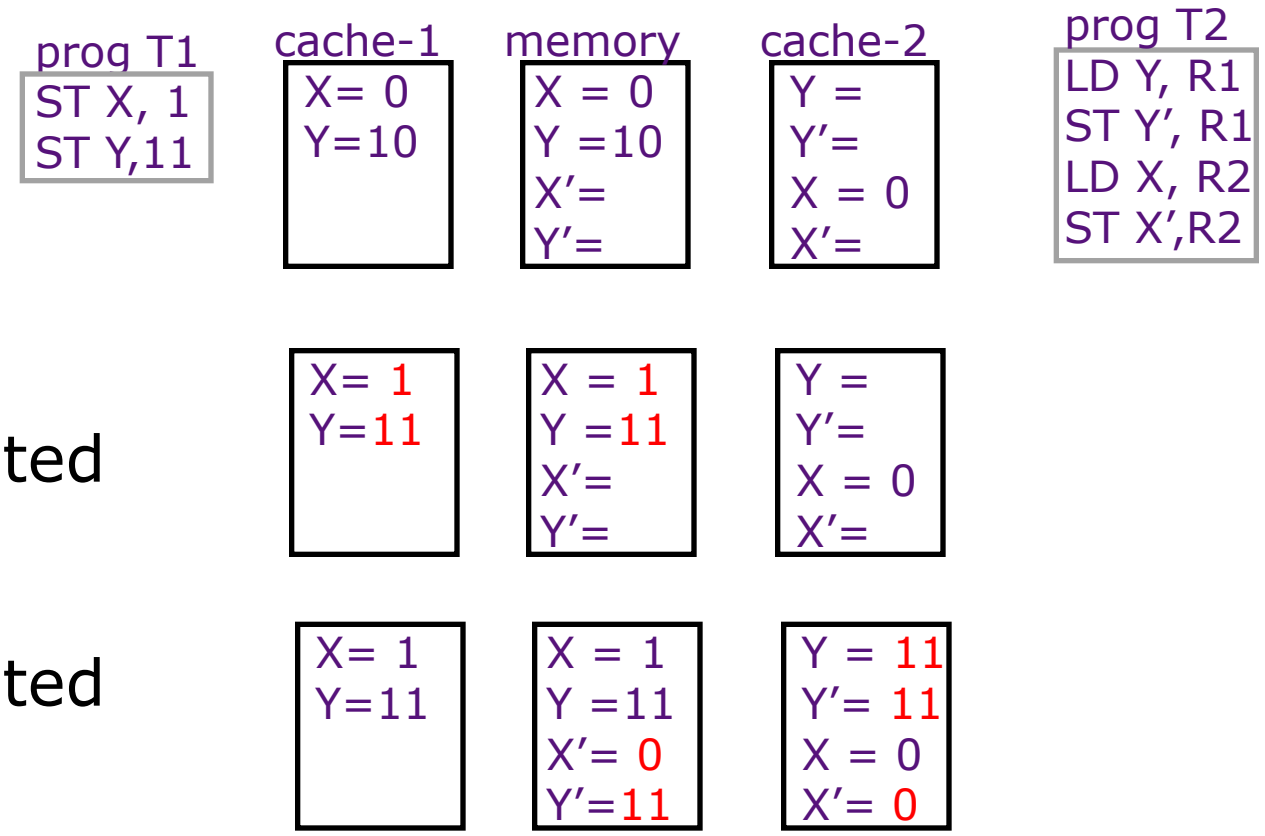
*Do these stale values matter?*

*What is the view of shared memory for programming?*

# Write-back Caches & SC



# Write-through Caches & SC




- T1 executed
- T2 executed

*Write-through caches don't preserve sequential consistency either*

# Maintaining Cache Coherence

- 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

  -> cache coherence protocols

# Cache Coherence vs. Memory Consistency

- A cache coherence protocol ensures that all writes by one processor are *eventually* visible to other processors, **for one memory address**
  - i.e., updates are not lost

- No guarantee of **when** an update should be seen
- No guarantee of what **order** of updates (of different addresses) should be seen

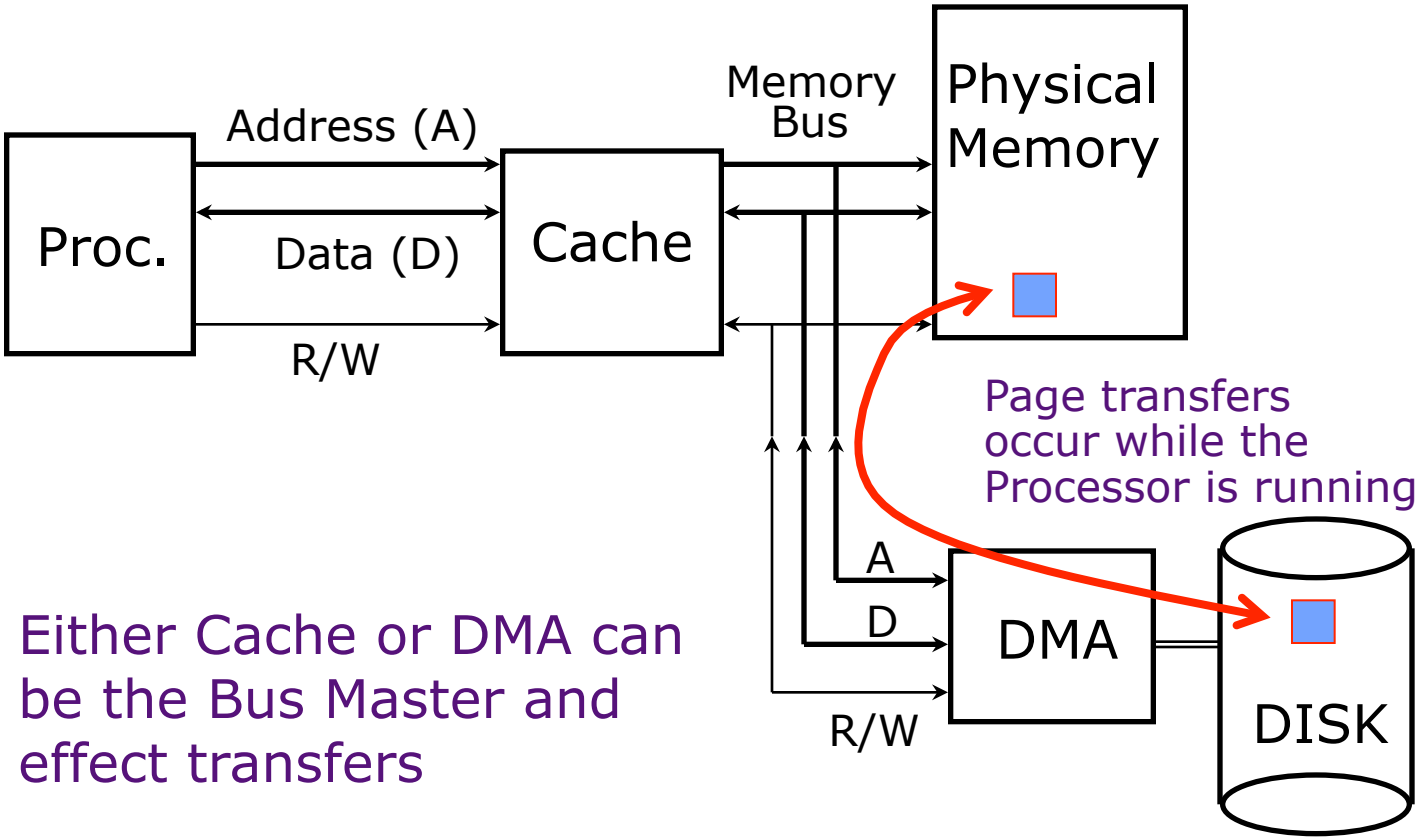
- A cache coherence protocol is not enough to ensure sequential consistency
  - But if sequentially consistent, then caches must be coherent

# Cache Coherence vs. Memory Consistency

- A memory consistency model gives the rules on when a write by one processor can be observed by a read on another, **across different addresses**
  - As previously seen with examples

- Combination of cache coherence protocol plus processor memory reorder buffer used to implement a given architecture's memory consistency model

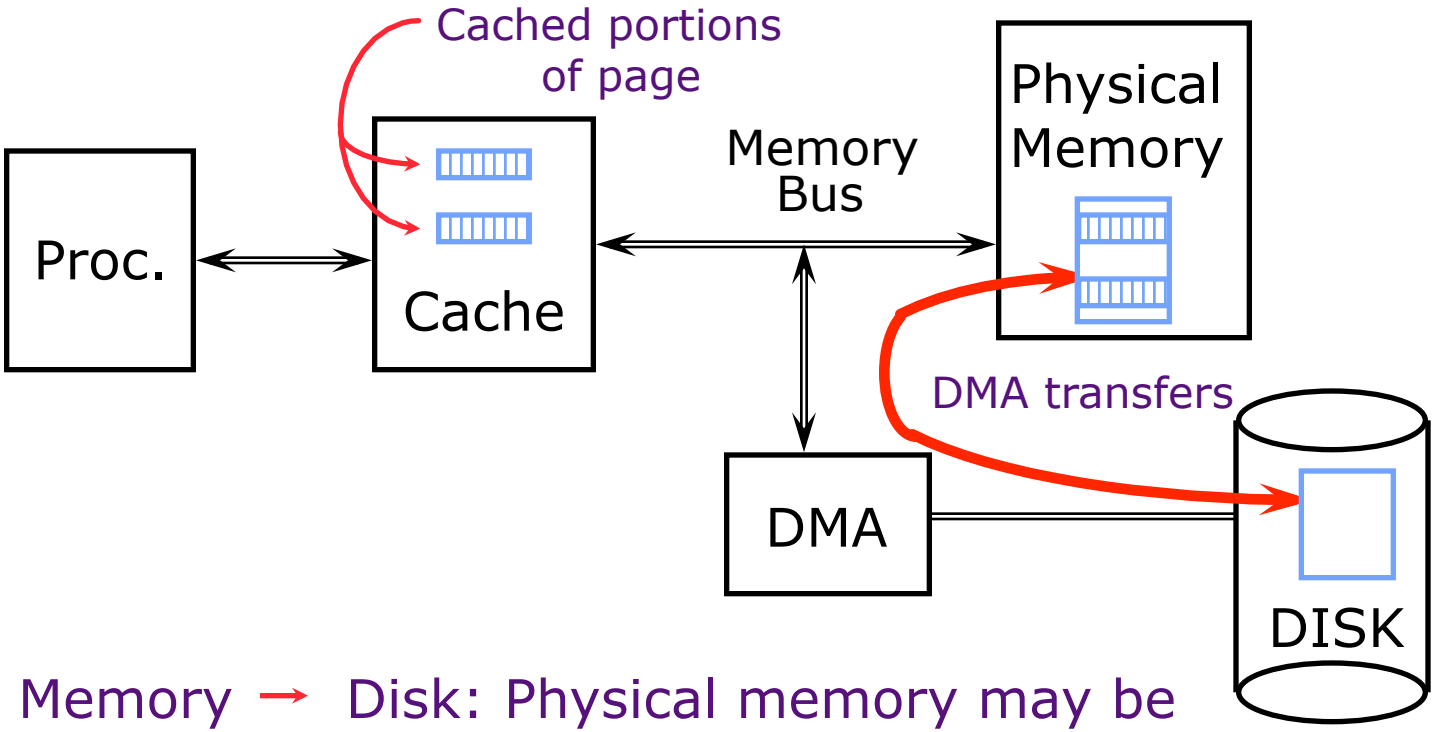
# Warmup: Parallel I/O




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

(DMA stands for "Direct Memory Access", means the I/O device can read/write memory autonomous from the CPU)

# Problems with Parallel I/O

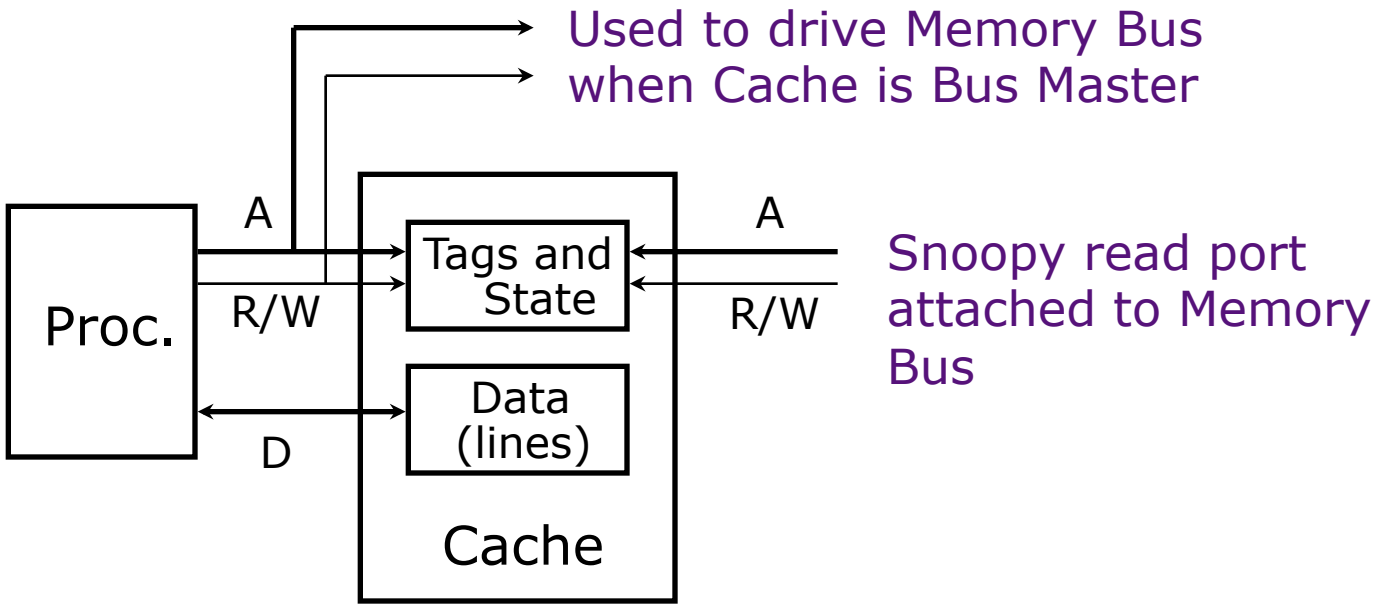



Memory → Disk: Physical memory may be stale if cache copy is dirty

Disk → Memory: Cache may hold stale data and not see memory writes

# Snoopy Cache, *Goodman 1983*

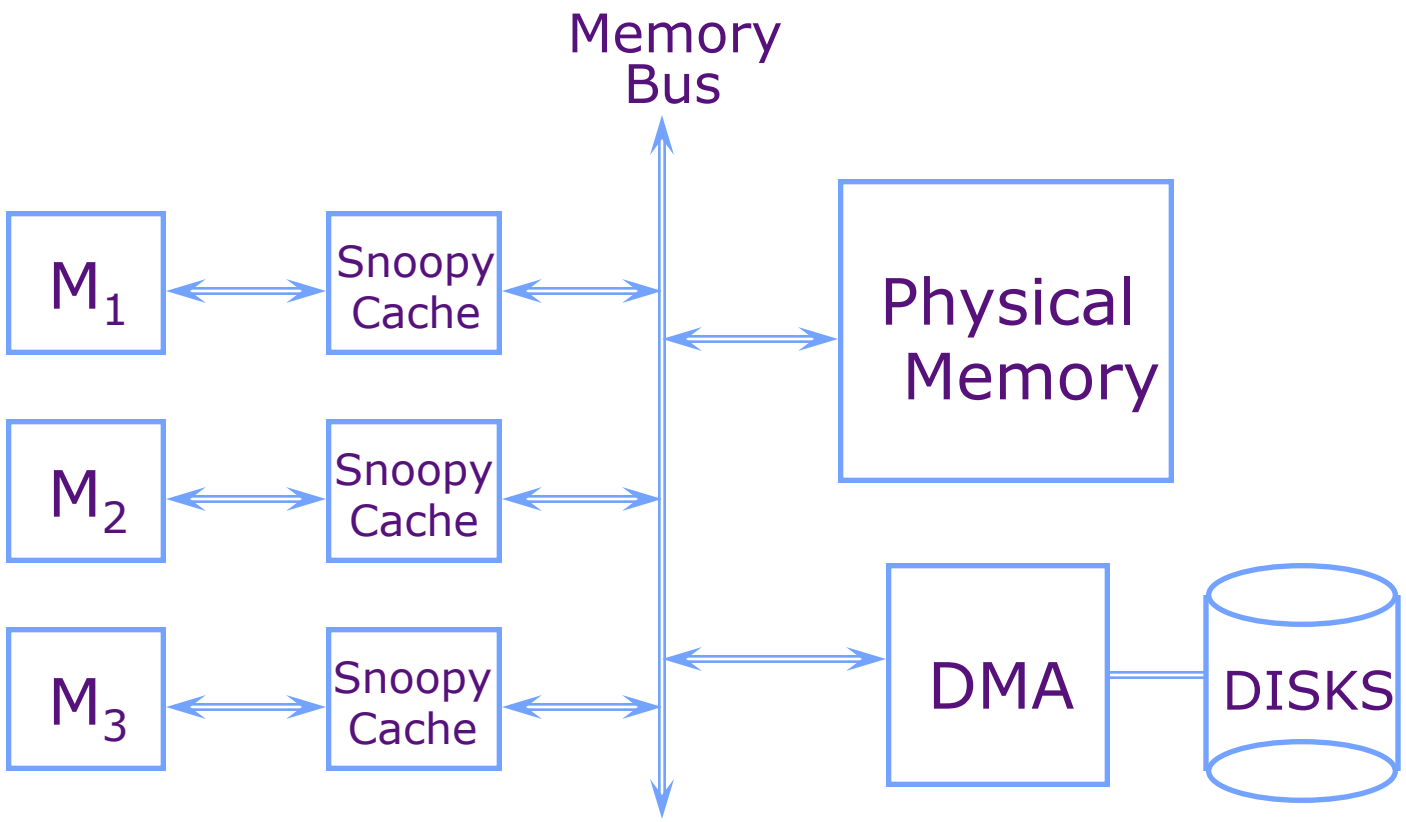
- Idea: Have cache watch (or snoop upon) DMA transfers, and then "do the right thing"
- Snoopy cache tags are dual-ported



# Snoopy Cache Actions for DMA

| Observed Bus Cycle | Cache State | Cache Action |
|---|---|---|
| DMA Read<br>Memory → Disk | Address not cached | No action |
| | Cached, unmodified | No action |
| | Cached, modified | Cache intervenes |
| DMA Write<br>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

# Snoopy Cache Coherence Protocols

*write miss:*

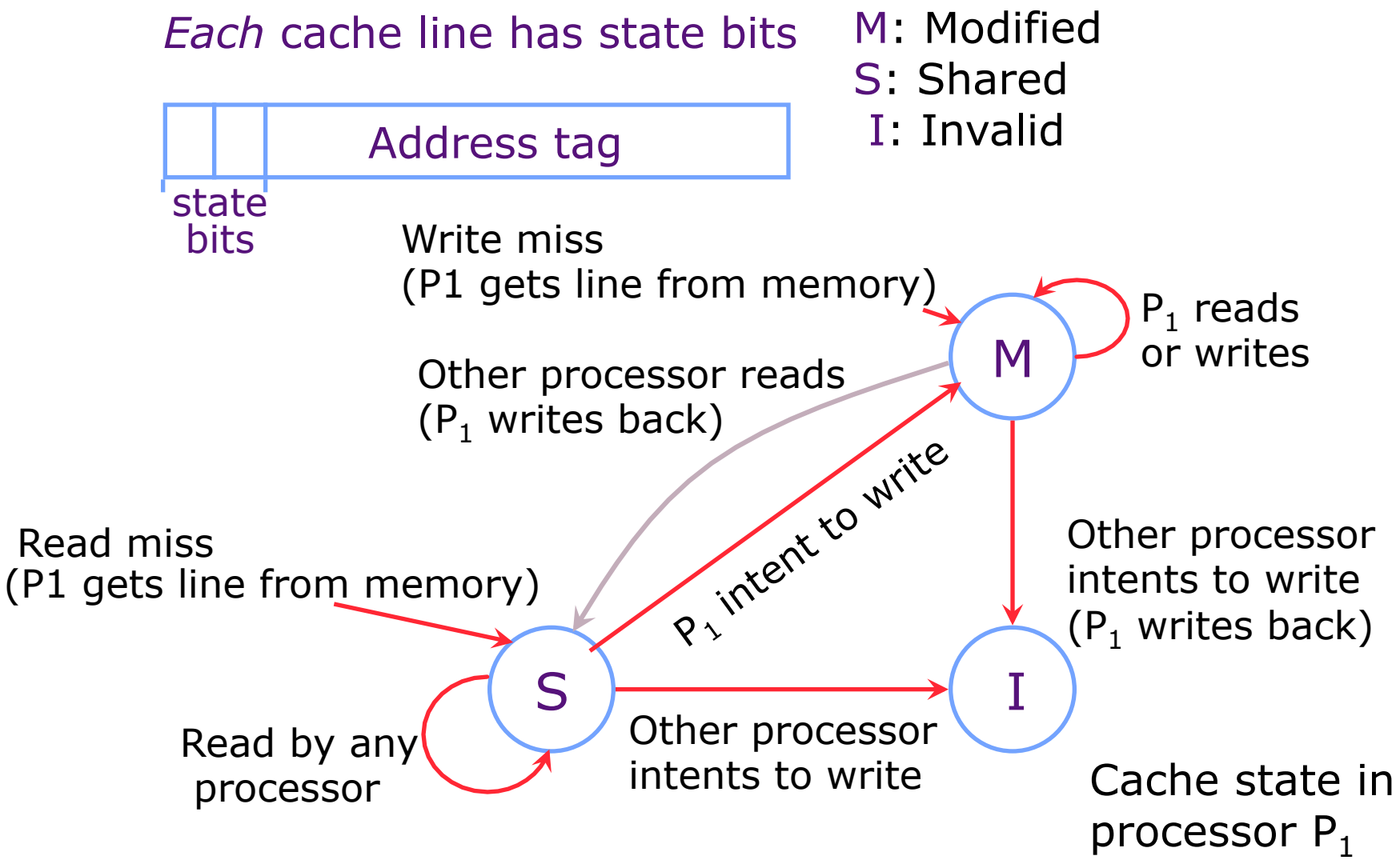
the address is *invalidated* in all other caches *before* the write is performed

*read miss:*

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

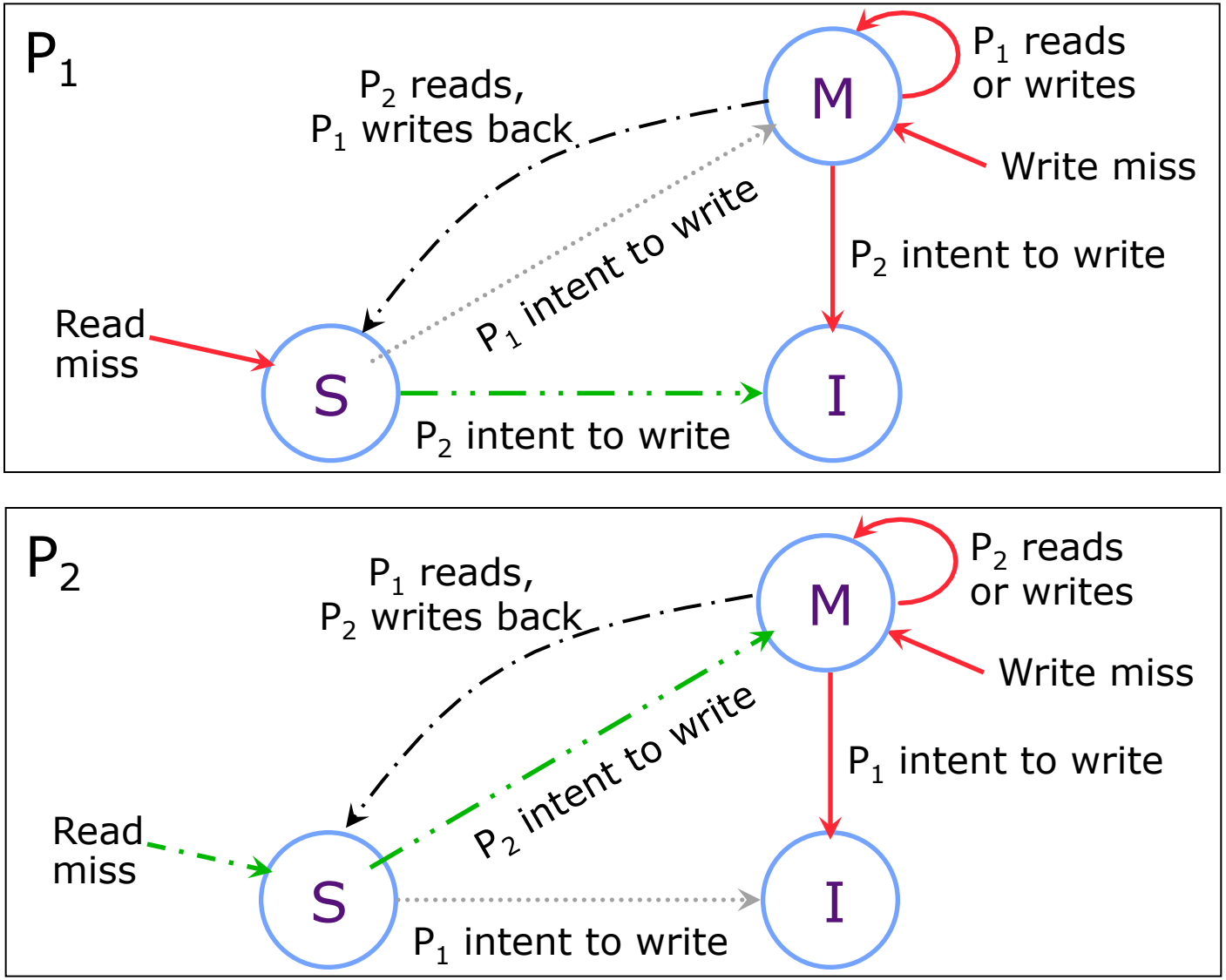
# Cache State Transition Diagram

*The MSI protocol*

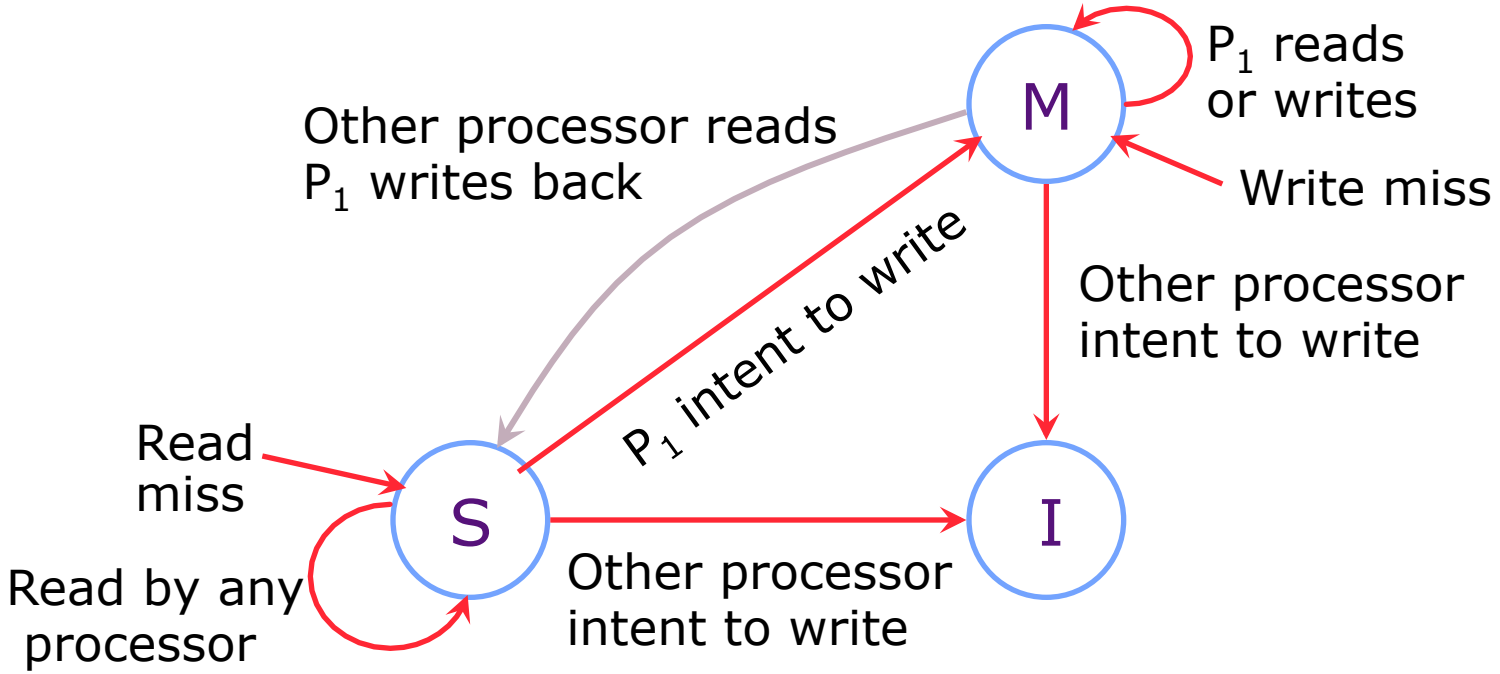


# Two Processor Example
## (Reading and writing the same cache line)

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



# Observation

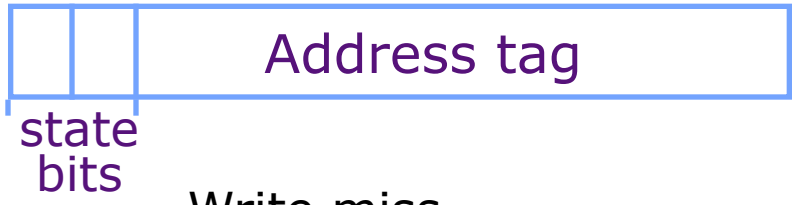



- If a line is in the M state then no other cache can have a copy of the line!
- Memory stays coherent, multiple differing copies cannot exist

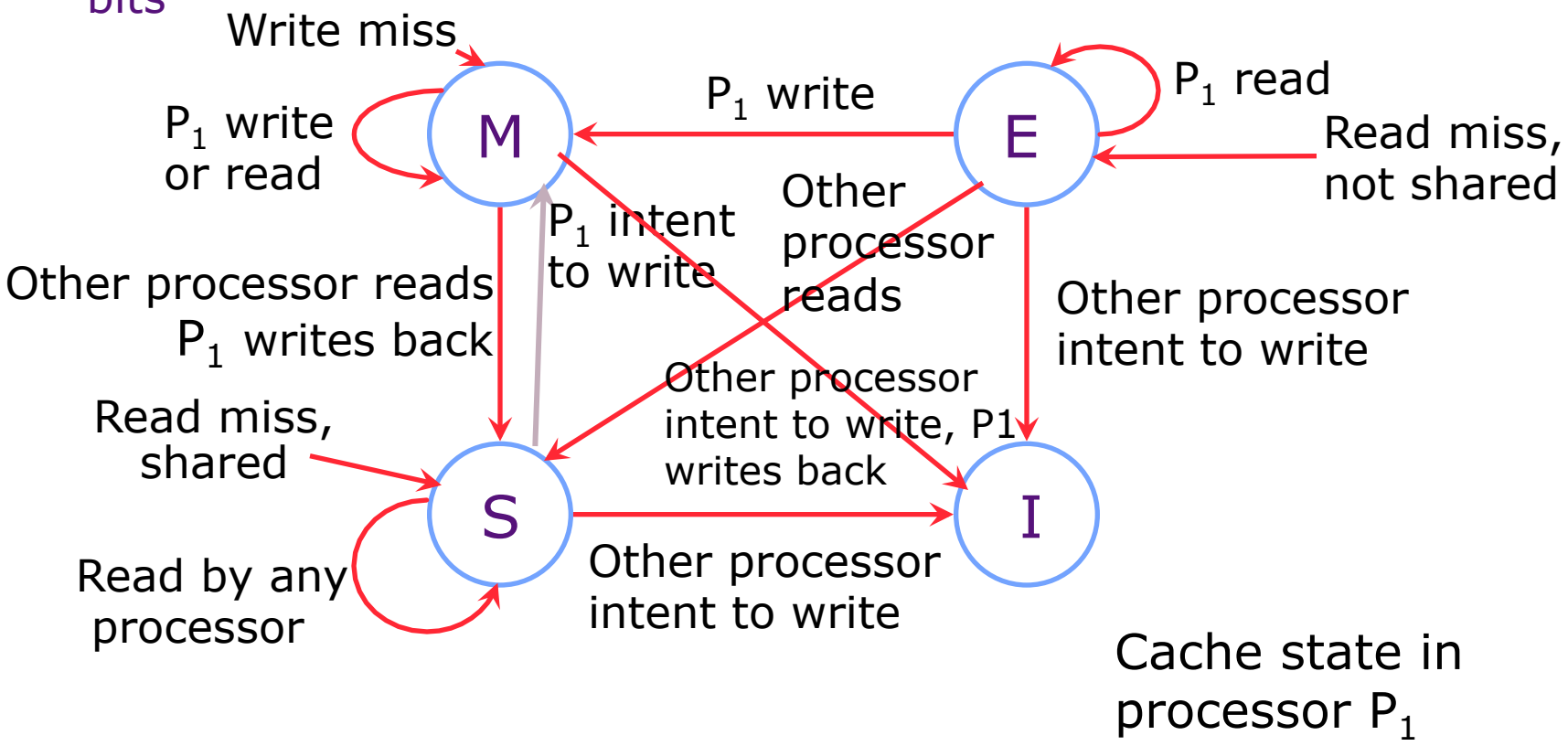
# MESI: An Enhanced MSI protocol

## increased performance for private data

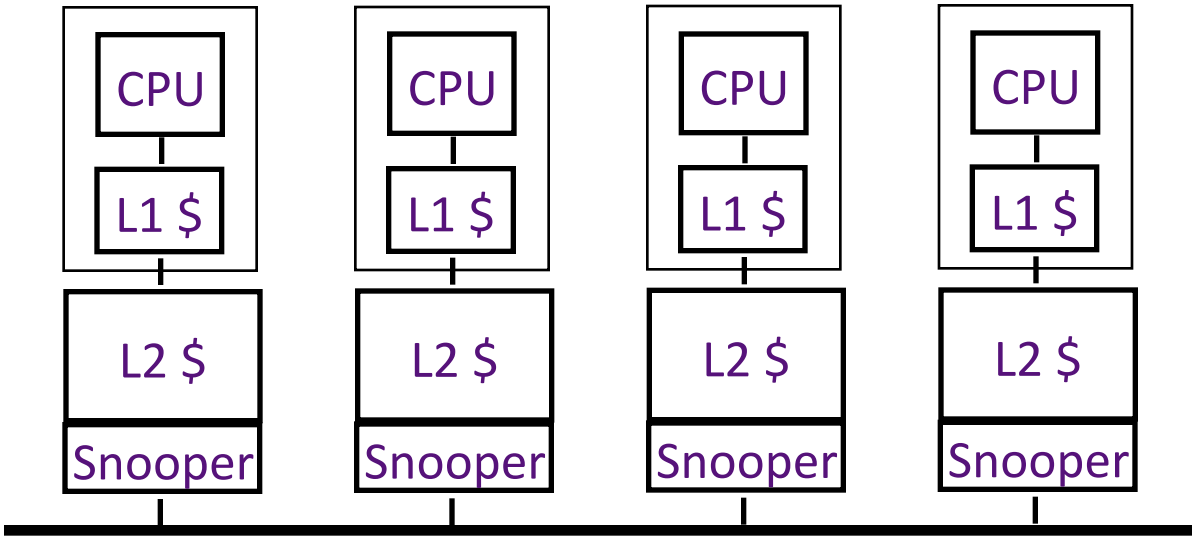
*Each* cache line has a tag




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



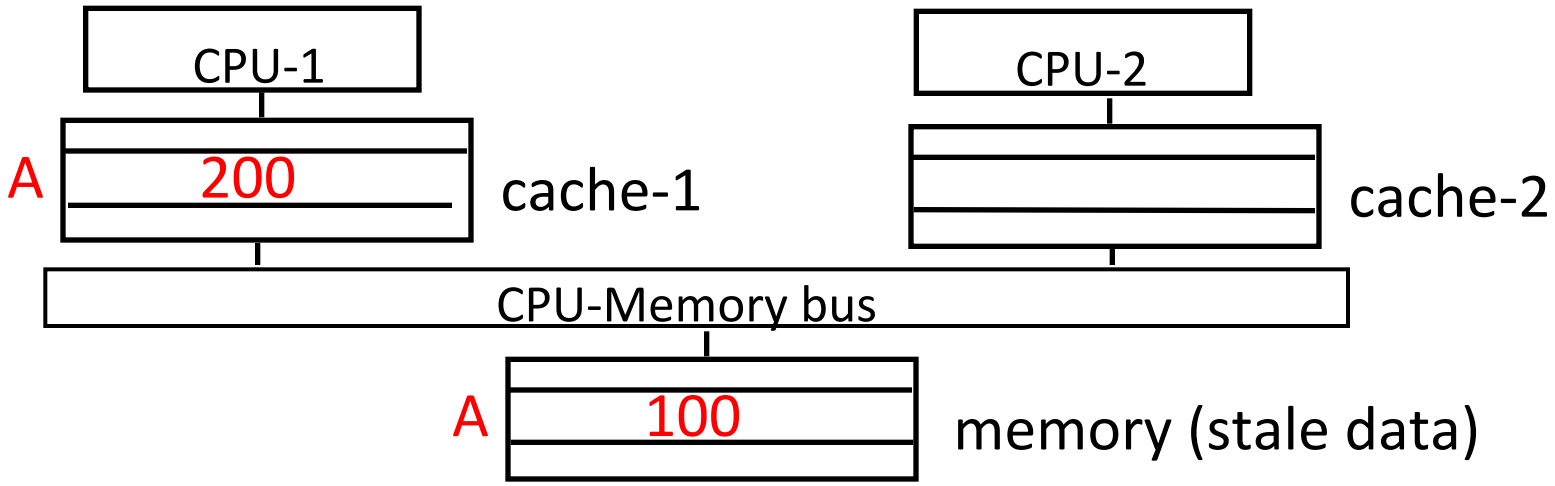
# Optimized Snoop with Level-2 Caches




- Processors often have two-level caches
  - small L1, large L2 (on chip)
- *Inclusion property:* entries in L1 must be in L2
  invalidation in L2 ⇒ 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


A cache line contains more than one word

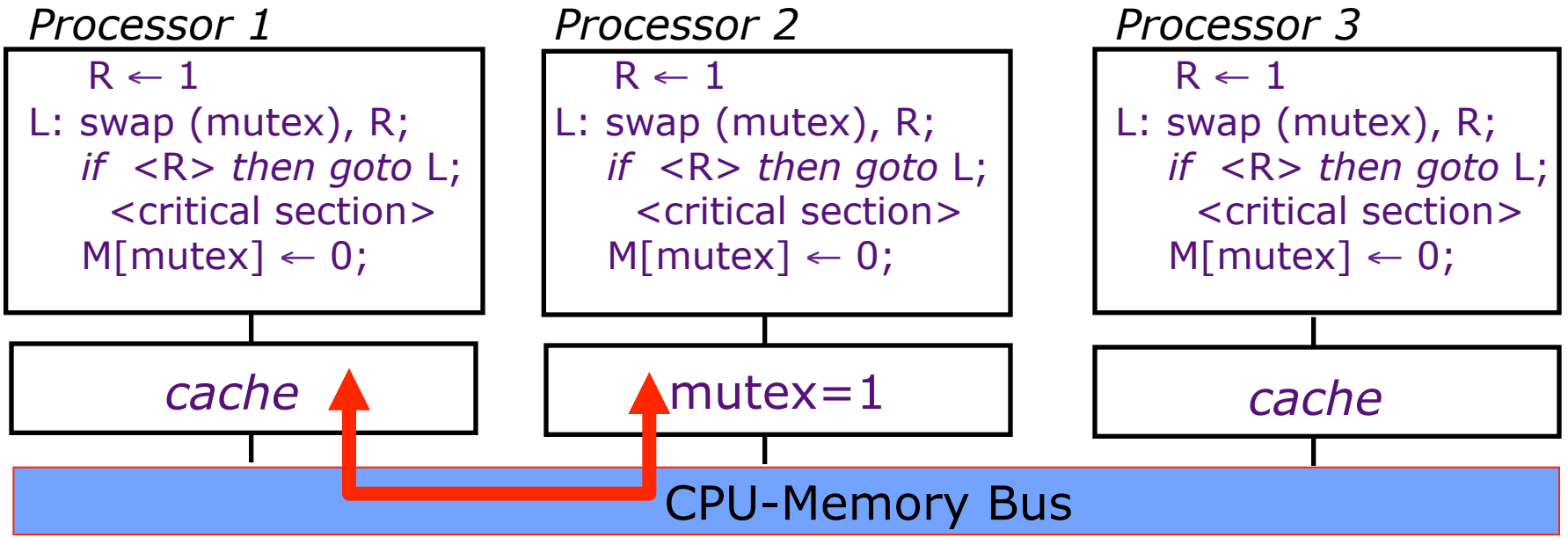
Cache-coherence is done at the line-level and not word-level

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

# Load-reserve & Store-conditional

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

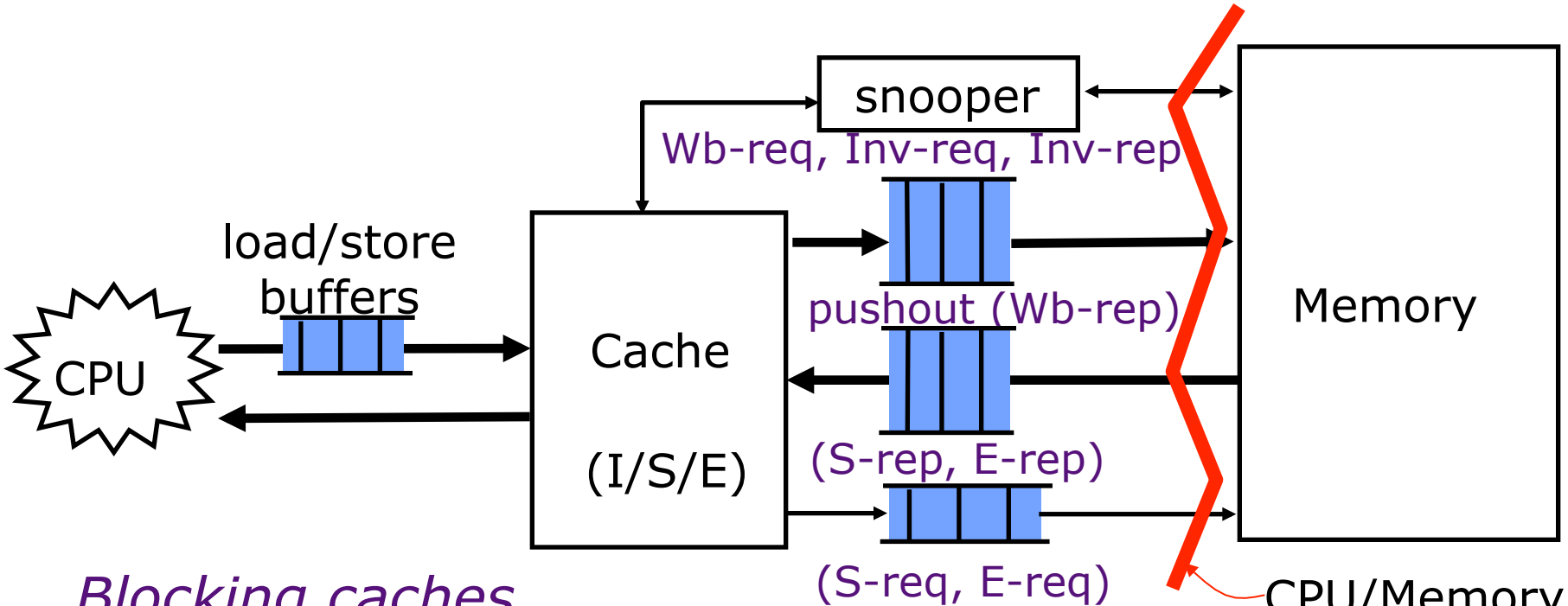
```
Load-reserve R, (a):
    <flag, adr> ← <1, a>;
    R ← M[a];
```

```
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

# Out-of-Order Loads/Stores & CC




*Blocking caches*

One request at a time + CC ⇒ SC

*Non-blocking caches*

Multiple requests (different addresses) concurrently + CC

⇒ Relaxed memory models

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

# Acknowledgements


- These slides contain material developed and copyright by:
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)
- MIT material derived from course 6.823
- UCB material derived from course CS252
- “New microprocessor claims 10x energy improvement”, from extremetech
- “Exploring the diverse world of programming” by Pavel Shved
- “Amdahl’s Law in the Multicore Era” b Mark D. Hill and Michael R. Marty