Lecture 10:

Cache Coherence: Part I

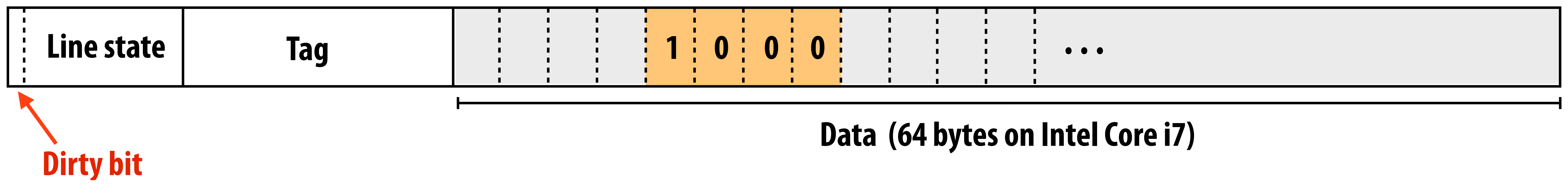
Parallel Computer Architecture and Programming
CMU 15-418, Spring 2013

Cache design review

Let's say your code executes `int x = 1;`

(Assume for simplicity x corresponds to the address 0x12345604 in memory... it's not stored in a register)

One Cache Line:



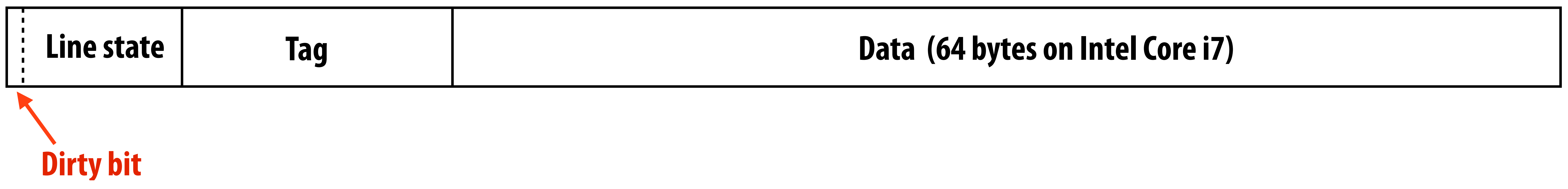
- **Write back vs. write-through cache**

- **Write allocate vs. write-no-allocate cache**

Review: write miss behavior of write-allocate, write-back cache (uniprocessor case)

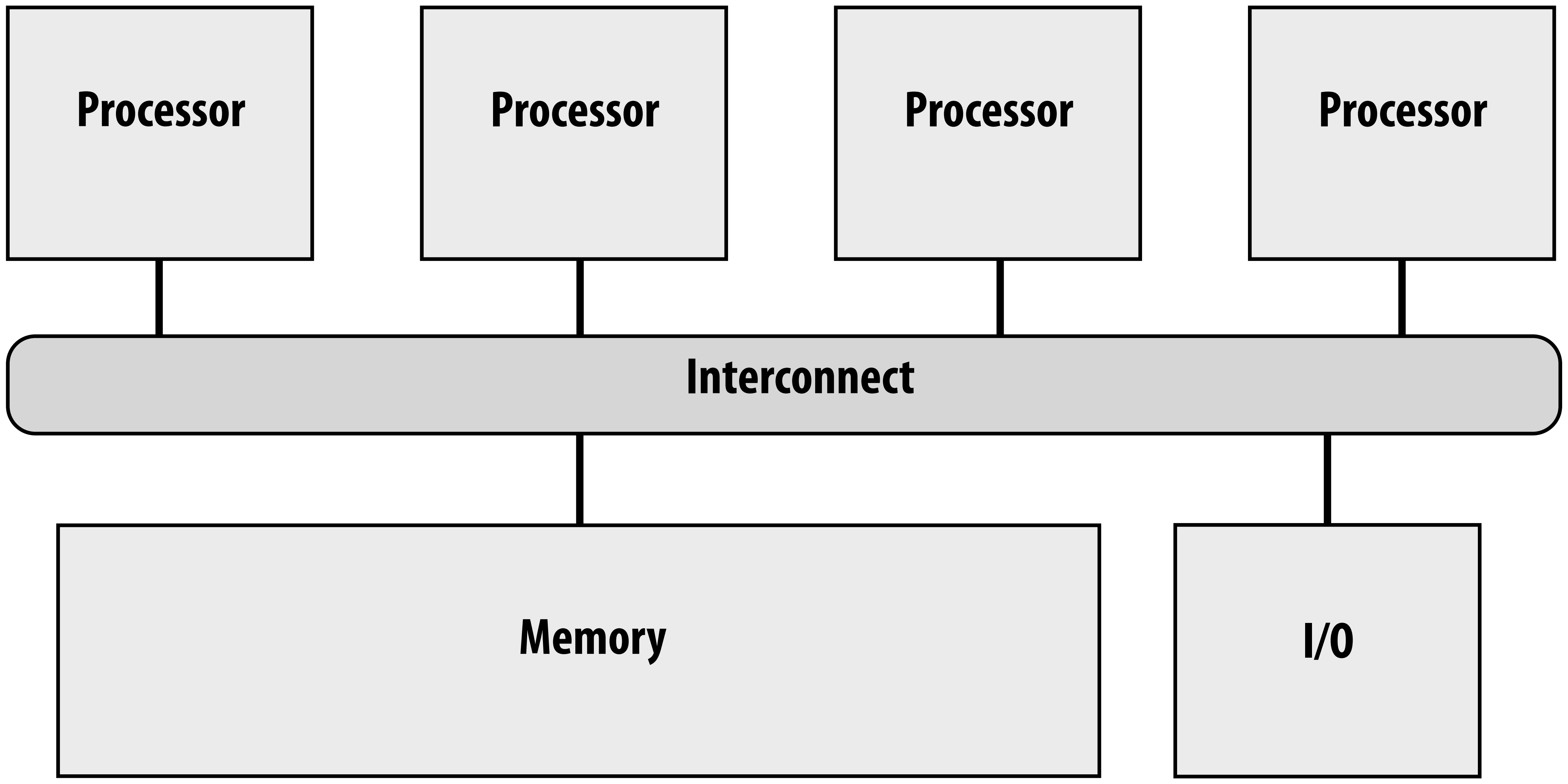
Example: code executes `int x = 1;`

1. Processor performs write to address in line that is not resident in cache
2. Cache loads line from memory
3. One word in cache is updated
4. Cache line is marked as dirty



A shared memory multi-processor

- Processors read and write to shared variables
    - More precisely: processors issue load and store instructions
- A reasonable expectation of memory is:
    - Reading a value at address X should return the last value written to address X *by any processor*

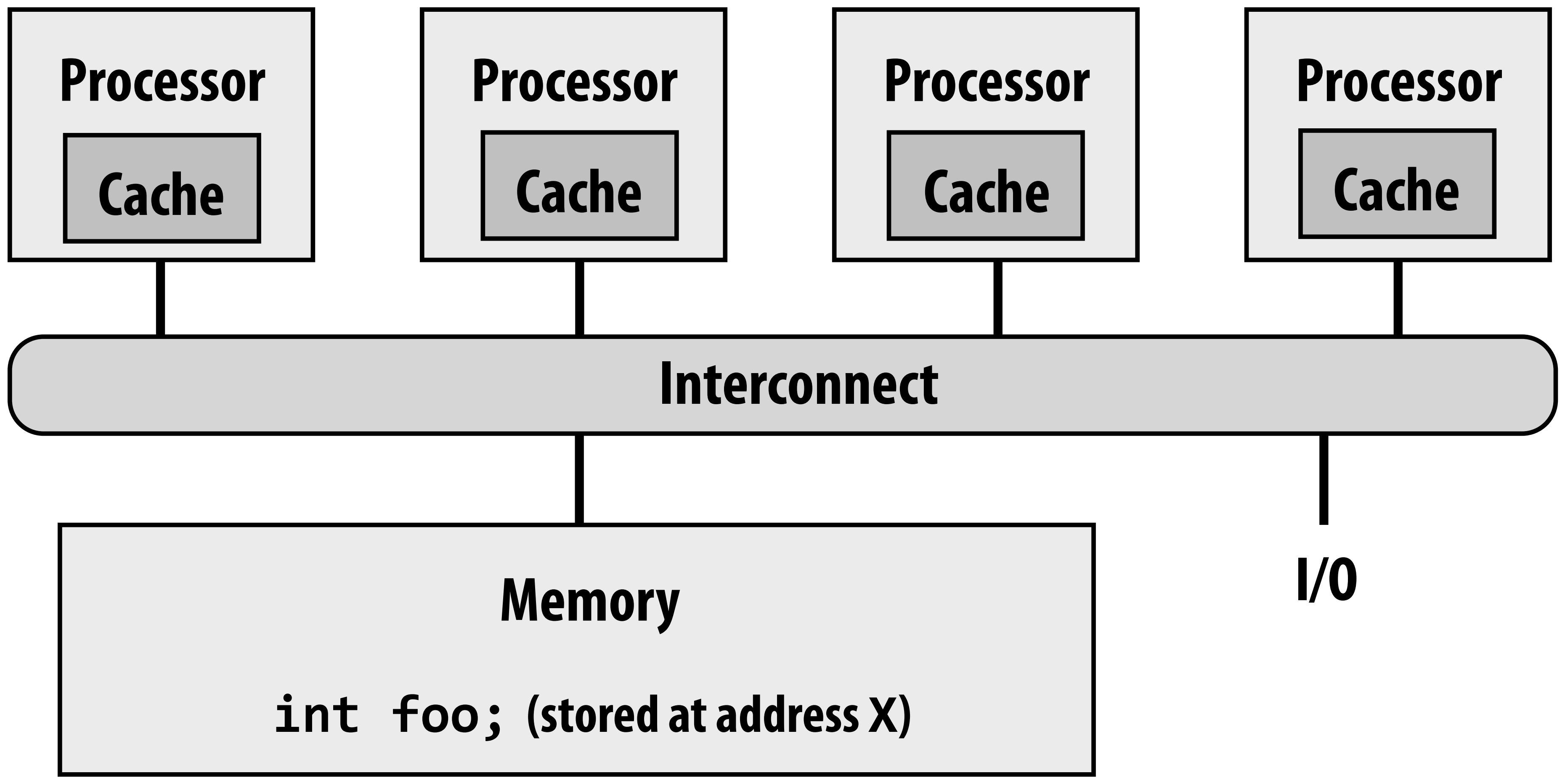


(A nice and simple view of four processors and their shared address space)

The cache coherence problem

Modern processors replicate contents of memory in local caches

Result of writes: processors can have different values for the same memory location

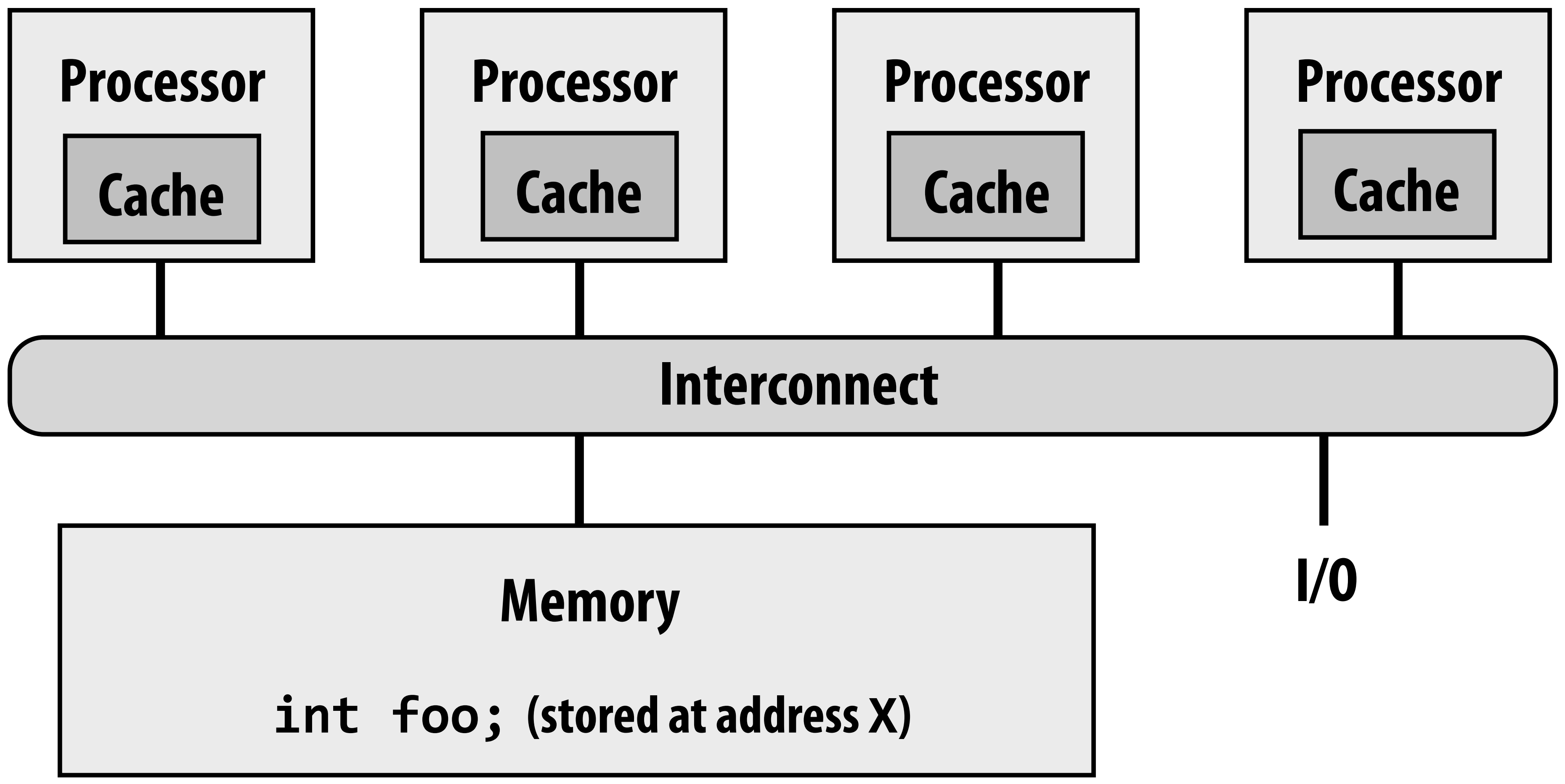


Action	P1 $	P2 $	P3 $	P4 $	mem[X]
					`0`
`P1 load X`	$\boxed{0}$ miss				`0`
`P2 load X`	`0`	$\boxed{0}$ miss			`0`
`P1 store X`	`1`	`0`			`0`
`P3 load X`	`1`	`0`	$\boxed{0}$ miss		`0`
`P3 store X`	`1`	`0`	`2`		`0`
`P2 load X`	`1`	$\boxed{0}$ hit	`2`		`0`
`P1 load Y` `(say this load causes eviction of X)`		`0`	`2`		`1`

Chart shows the value of `foo` (a program variable stored at address X) in main memory, and also in each processor's cache **

** Assumes write-back cache behavior

The cache coherence problem



How is this problem different from the mutual exclusion problem?

Can you fix the problem by adding locks to your program?

Action	P1 $	P2 $	P3 $	P4 $	mem[X]
					0
P1 load X	$\boxed{0}$ miss				0
P2 load X	0	$\boxed{0}$ miss			0
P1 store X	1	0			0
P3 load X	1	0	$\boxed{0}$ miss		0
P3 store X	1	0	2		0
P2 load X	1	$\boxed{0}$ hit	2		0
P1 load Y (say this load causes eviction of X)		0	2		1

Chart shows the value of `foo` (a program variable stored at address X) in main memory, and also in each processor's cache **

** Assumes write-back cache behavior

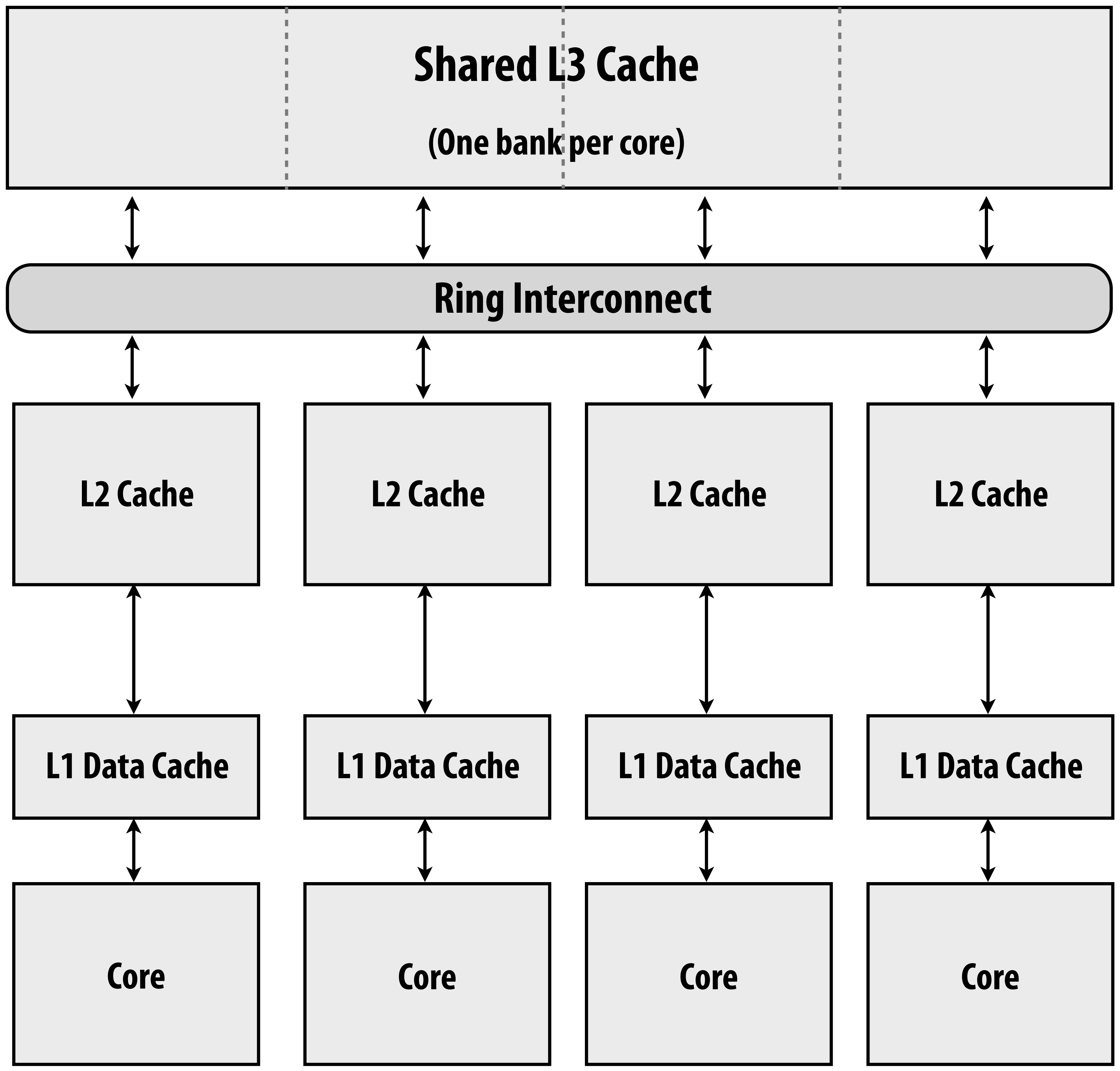
The cache coherence problem

- Intuitive behavior for memory system: reading value at address X should return the last value written at address X *by any processor.*

- Coherence problem exists because there is both global storage (main memory) and per-processor local storage (processor caches) implementing the <u>abstraction</u> of a single shared address space.

Cache hierarchy of Intel Core i7 CPU

64 byte cache line size



L3: (per chip)
8 MB, inclusive
16-way set associative
32B / clock per bank
26-31 cycle latency

L2: (private per core)
256 KB
8-way set associative, write back
32B / clock, 12 cycle latency
Up to 16 outstanding misses

L1: (private per core)
32 KB
8-way set associative, write back
2 x 16B loads + 1 x 16B store per clock
4-6 cycle latency
Up to 10 outstanding misses

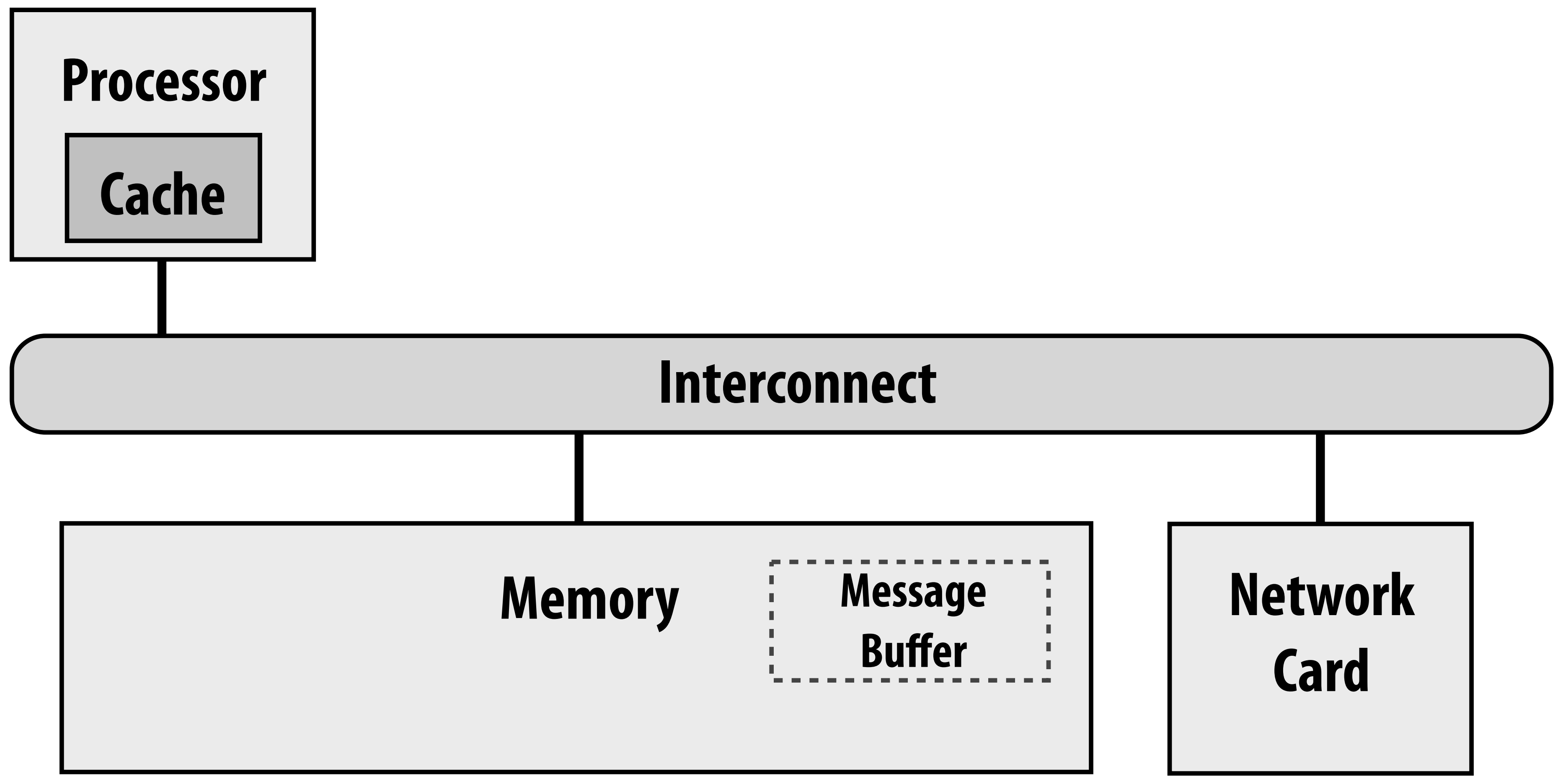
Intuitive expectation of shared memory

- **Intuitive behavior for memory system: reading value at address should return the last value written at the address *by any processor.***

- **Uniprocessor, providing this behavior is fairly simple, since writes typically come from one client: the processor**
    - **Exception: device I/O via direct memory access (DMA)**

Coherence is an issue in a single CPU system

Consider I/O device performing DMA data transfer



Case 1:
Processor writes to buffer in main memory
Processor tells network card to async send buffer
Problem: network card many transfer stale data if processor's writes not flushed to memory

Case 2:
Network card receives message
Network card copies message in buffer in main memory using DMA transfer
Notifies CPU msg received, buffer ready to read
Problem: CPU may read stale data if addresses updated by network card happen to be in cache.

- Common solutions:
  - CPU writes to shared buffers using uncached stores (e.g., driver code)
  - OS support:
    - Mark VM pages containing shared buffers as not-cachable
    - Explicitly flush pages from cache when I/O completes
- In practice, DMA transfers are infrequent compared to CPU loads and stores (so these heavyweight solutions are acceptable)

Problems with the intuition

- **Intuitive expectation: reading value at address X should return the last value written to address X *by any processor.***

- **What does "last" mean?**
  - **What if two processors write at the same time?**
  - **What if a write by P1 is followed by a read from P2 so close in time, it's impossible to communicate occurrence to other processors?**

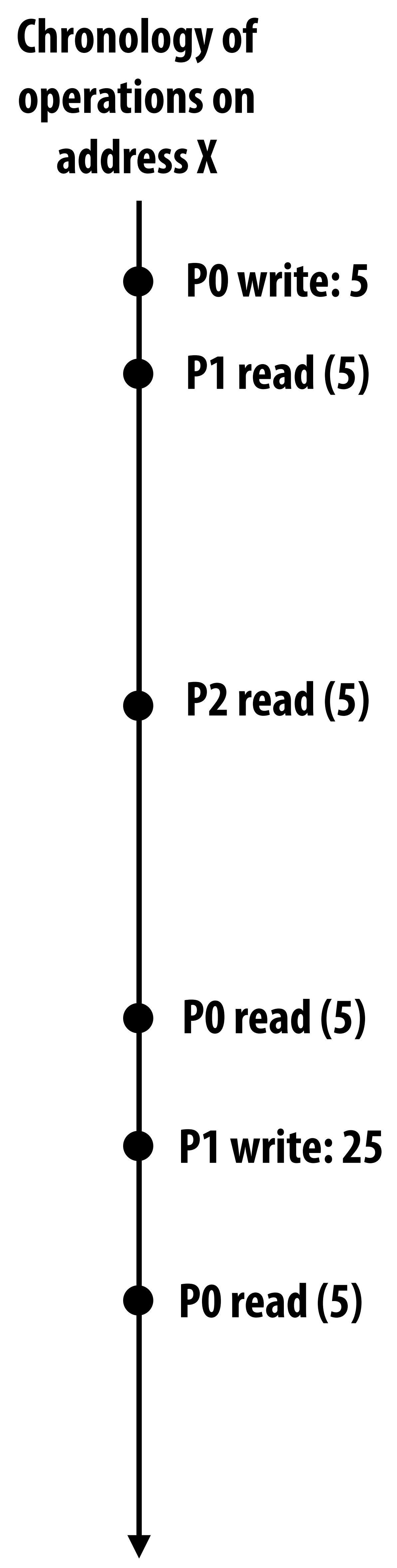
- **In a sequential program, "last" is determined by program order (not time)**
  - **Holds true within one thread of a parallel program**
  - **But need to come up with a meaningful way to describe order across threads**

Definition: coherence

A memory system is coherent if:

The results of a parallel program's execution are such that for <u>each memory location</u>, there is a hypothetical serial order of all program operations (executed by all processors) to the location that is consistent with the results of execution, and:

1. **Memory operations issued by any one processor occur in the order issued by the processor**
2. **The value returned by a read is the value written by the last write to the location in the serial order**



Definition: coherence (said differently)

A memory system is coherent if:

1. **A read by processor P to address X that follows a write by P to address X, should return the value of the write by P** *(assuming no other processor wrote to X in between)*

2. **A read by a processor to address X that follows a write by another processor to X returns the written value... if the read and write are sufficiently separated in time** *(assuming no other write to X occurs in between)*

3. **Writes to the same location are serialized: two writes to the same location by any two processors are seen in the same order by all processors.**
   (Example: if values 1 and then 2 are written to address X, no processor observes 2 before 1)

Condition 1: program order (as expected of a uniprocessor system)

Condition 2: "write propagation": The news of the write has to eventually get to the other processors. Note that precisely when it is propagated is not specified in the definition of coherence.

Condition 3: "write serialization"

Write serialization

Writes to the <u>same location</u> are serialized: two writes to the same location by any two processors are seen in the same order by all processors.
(Example: if values 1 and then 2 are written to address X, no processor observes 2 before 1)

Example: P1 writes value *a* to X. Then P2 writes value *b* to X.

Consider situation where processors observe different order of writes:

Order observed by P1	Order observed by P2
$x \leftarrow a$	$x \leftarrow b$
⋮	⋮
$x \leftarrow b$	$x \leftarrow a$

In terms of the first coherence definition: there is no global ordering of loads and stores to X that is in agreement with results of this parallel program.
(you cannot put the two memory operations on a single timeline and have both processor's observations agree with the timeline)

Coherence vs. consistency

- Coherence defines behavior of reads and writes to the <u>same</u> memory location

- "Memory consistency" (a topic of a future lecture) defines the behavior of reads and writes to <u>different</u> locations
  - Consistency deals with the WHEN of write propagation
  - Coherence only guarantees that it does eventually propagate

- For the purposes of this lecture:
  - If processor writes to address X and then writes to address Y. Then any processor that sees result of write to Y, also observes result of write to X.
  - Why does coherence alone not guarantee this? (hint: first bullet point)

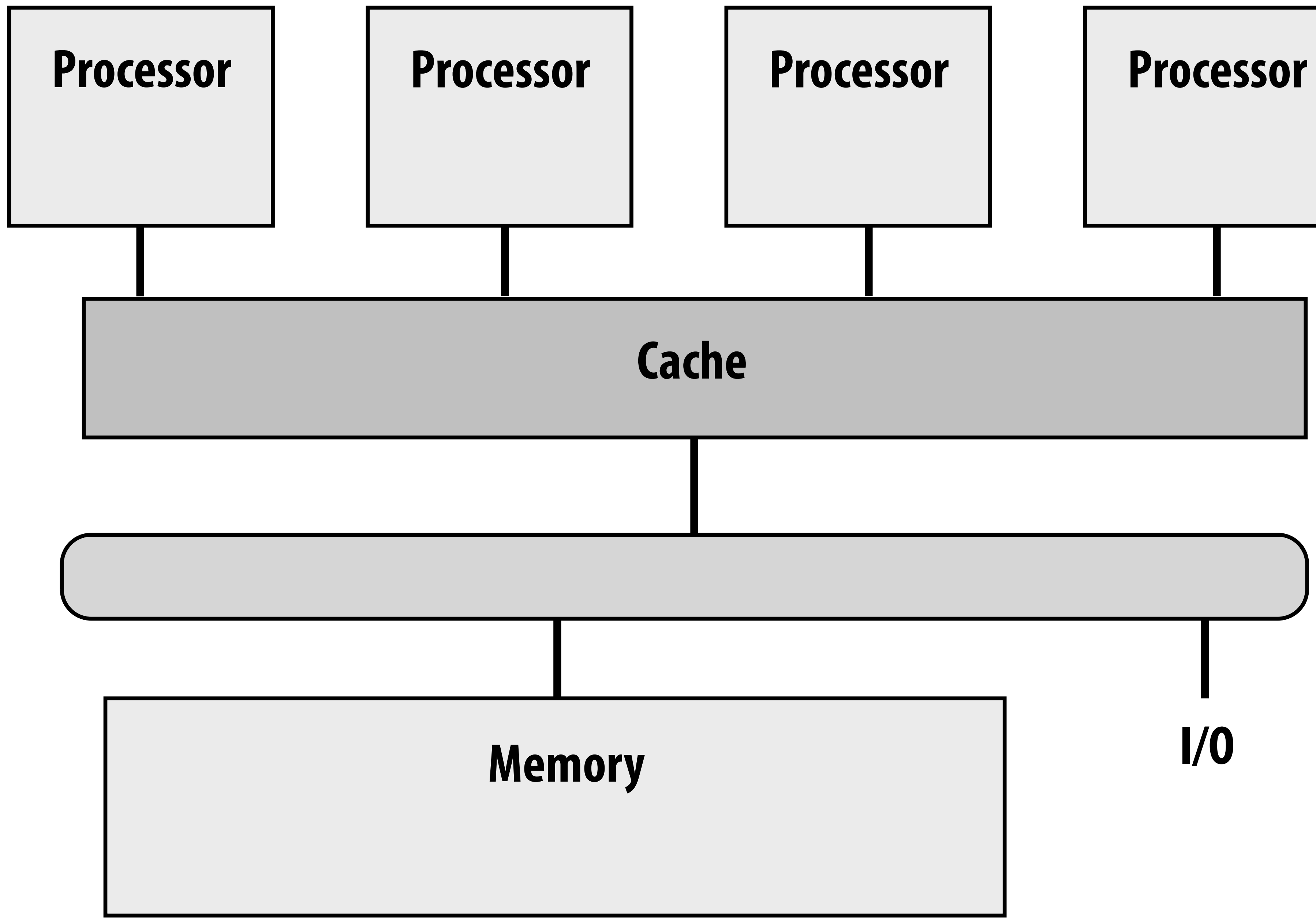
Implementing coherence

- **Software-based solutions**
  - OS uses page fault mechanism to propagate writes
  - Implementations provide memory coherence over clusters of workstations
  - We won't discuss these solutions

- **Hardware-based solutions**
  - "Snooping" based (today and next class)
  - Directory based (next, next class)

Shared caches: coherence made easy

- Obvious scalability problems (the point of a cache is to be local and fast)
  - Interference / contention
- But can have benefits:
  - Fine-grained sharing (overlapping working sets)
  - Actions by one processor might pre-fetch for another



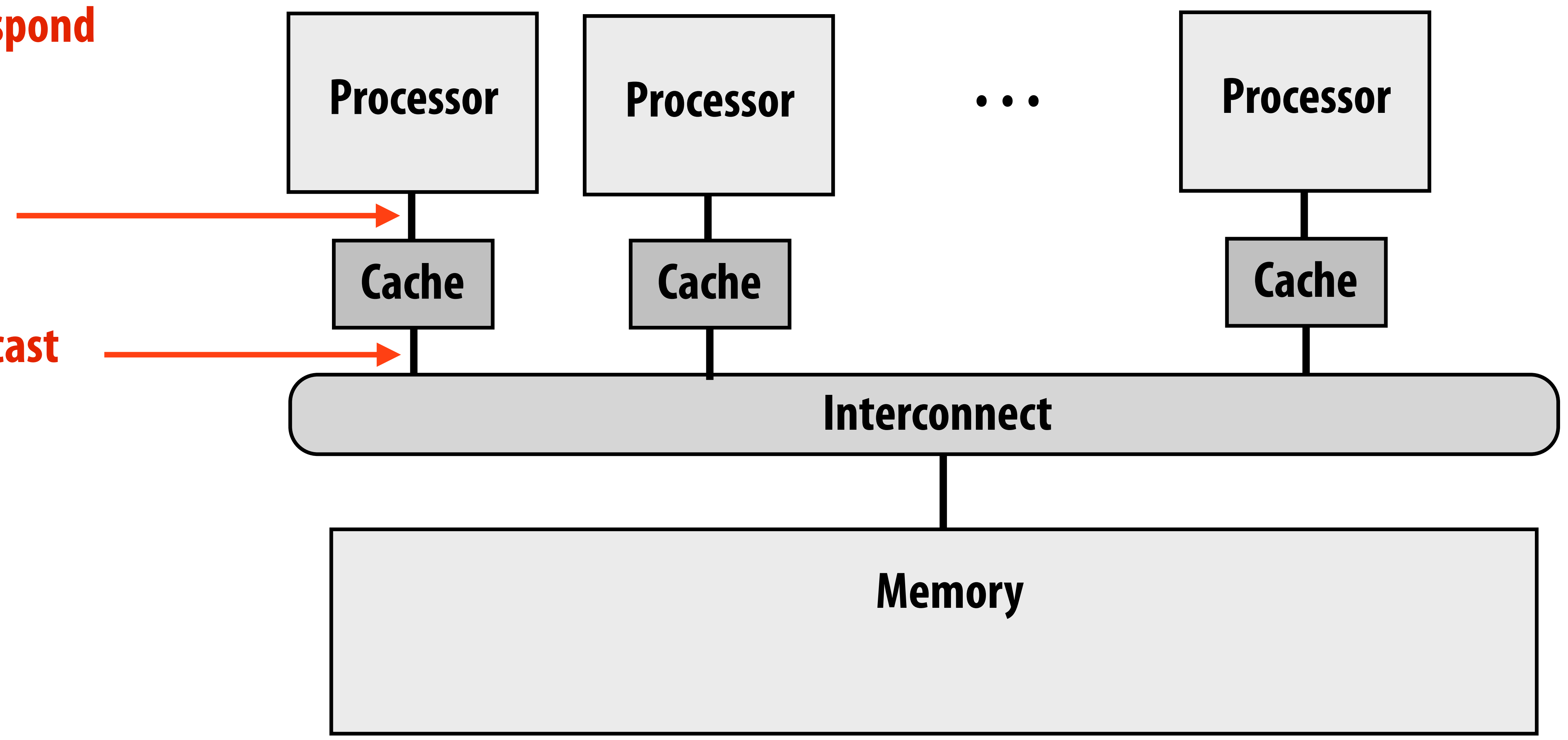
Snooping cache-coherence schemes

- Main idea: all coherence-related activity is broadcast to all processors in the system (actually, the processor's cache controllers)

- Cache controllers monitor ("they snoop") memory operations, and react accordingly to maintain memory coherence

Notice: now cache controller must respond to actions from "both ends":

1. LD/ST requests from its processor
2. Coherence-related activity broadcast over-interconnect



Very simple coherence implementation

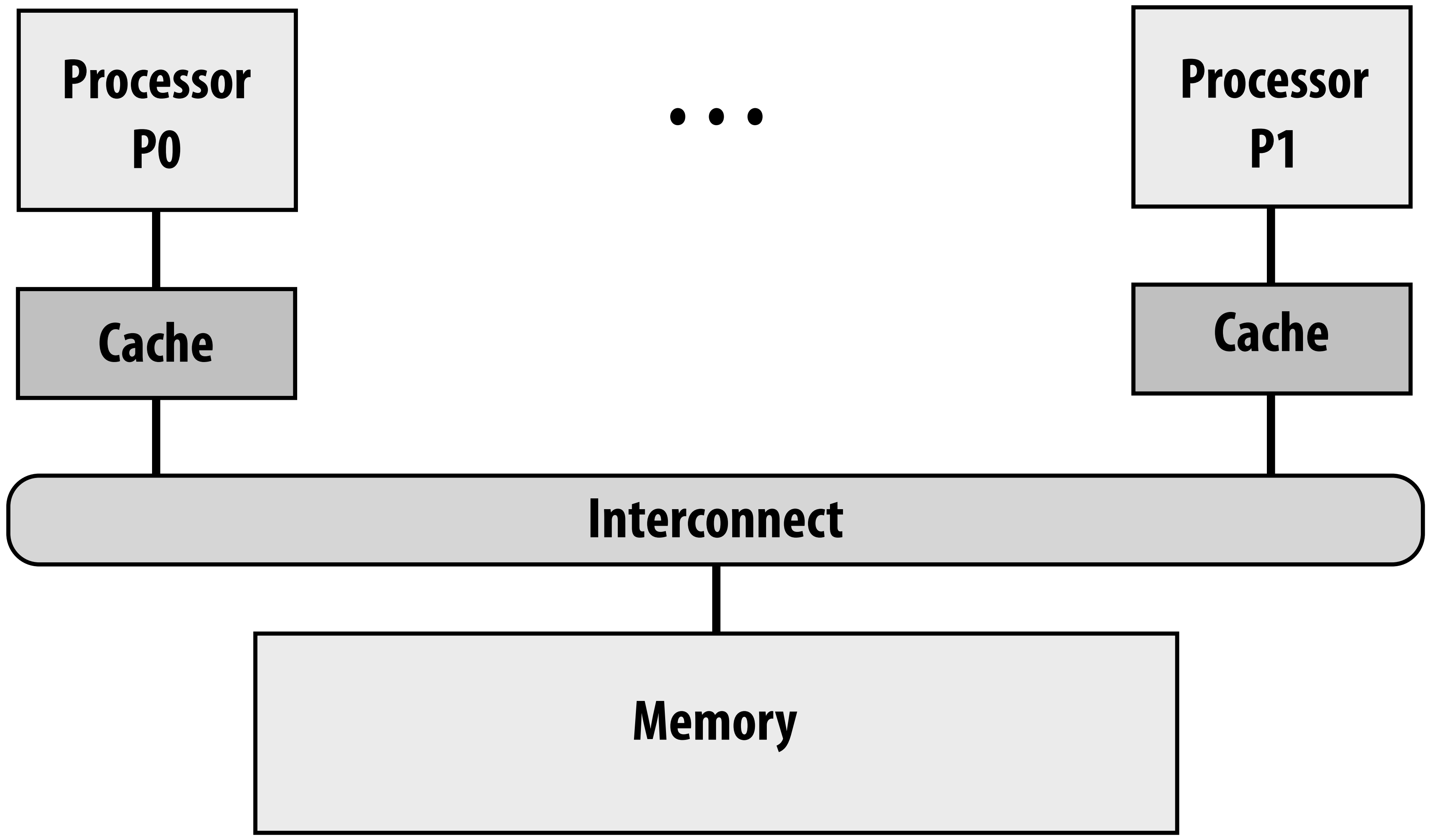
Let's assume:

1. Write-through caches
2. Granularity of coherence is cache block

Upon write, cache controller broadcasts invalidation message

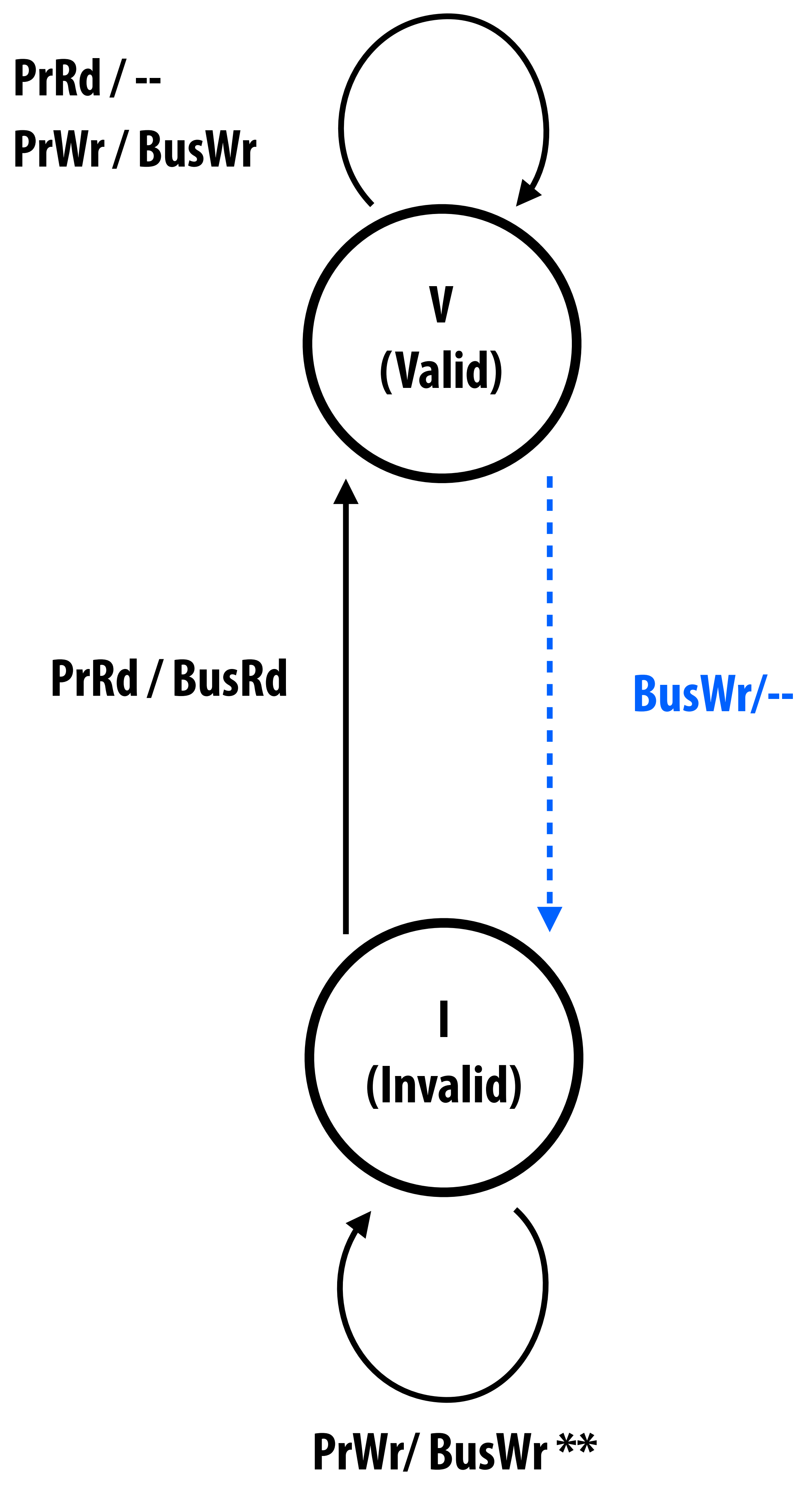
As a result, the next read from other processors will trigger cache miss

(processor retrieves updated value from memory due to write-through policy)



Action	Bus activity	P0 $	P1 $	mem location X
				0
`P0 load X`	`cache miss for X`	0		0
`P1 load X`	`cache miss for X`	0	0	0
`P0 write 100 to X`	`invalidation for X`	100		100
`P1 load X`	`cache miss for X`	100	100	100

Write-through invalidation: state diagram



A / B: if action A is observed by cache controller, action B is taken

- - - -▶	Remote processor (coherence) initiated transaction
⟶	Local processor initiated transaction

Requirements of the interconnect:

1. All write transactions visible to all cache controllers
2. All write transactions visible to all cache controllers in the same order

Simplifying assumptions here:

1. Interconnect and memory transactions are atomic
2. Processor waits until previous memory operations is complete before issuing next memory operation
3. Invalidation applied immediately as part of receiving invalidation broadcast

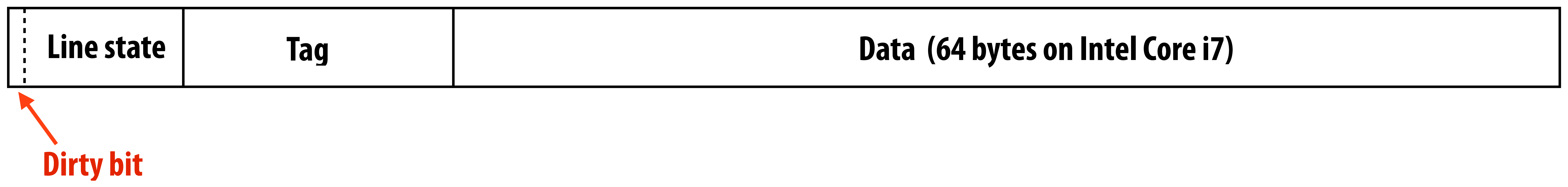
** Assumes write no-allocate policy (for simplicity)

Write-through policy is inefficient

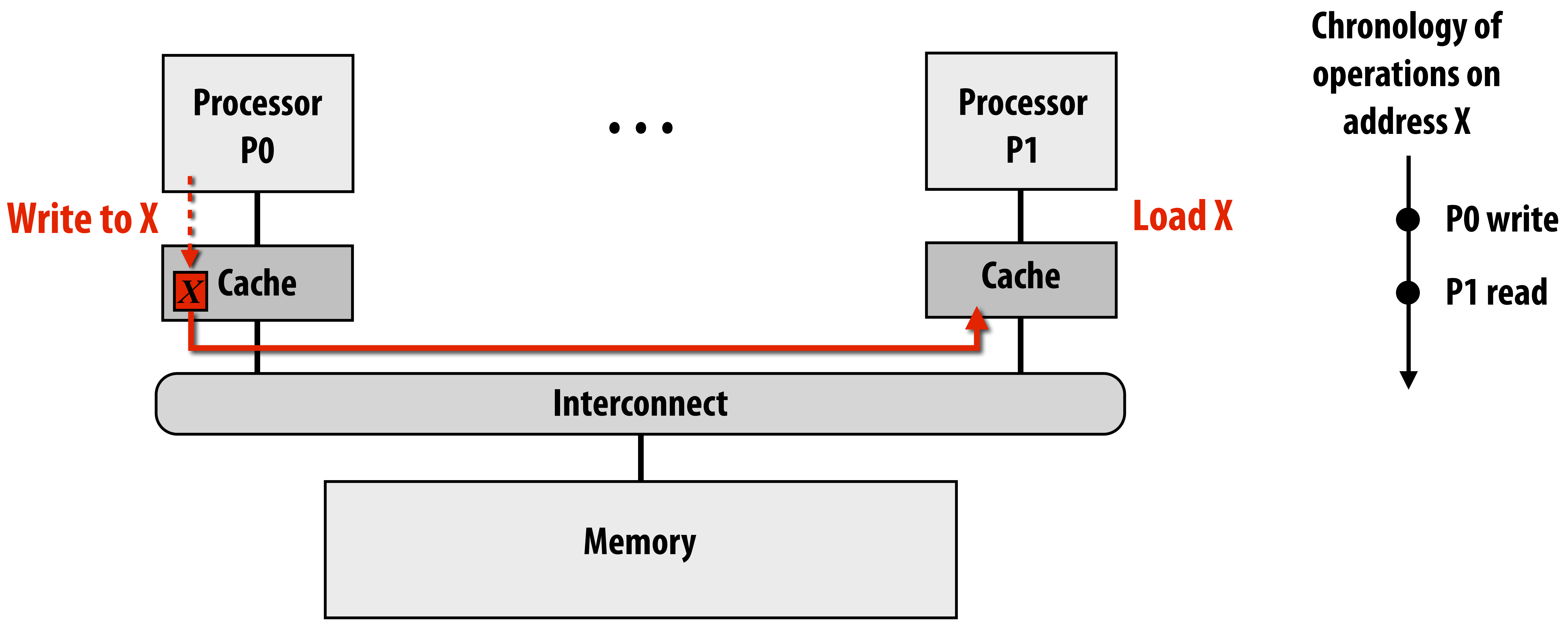
- **Every write operation goes out to memory**
  - Very high bandwidth requirements

- **Write-back caches absorb most write traffic as cache hits**
  - Significantly reduces bandwidth requirements
  - But now how do we ensure write propagation/serialization?
  - Require more sophisticated coherence protocols

Recall cache line state bits



Cache coherence with write-back caches



- **Dirty state of cache line now indicates exclusive ownership**
  - Exclusive: cache is only cache with a valid copy of line (so it can safely be written to)
  - Owner: cache is responsible for supplying data upon request (otherwise a load from another processor will get stale data)

Invalidation-based write-back protocol

- **A line in the "exclusive" state can be modified without notifying the other caches**
  - No other caches have the line resident, so other processors cannot read these values [without generating a memory read transaction]

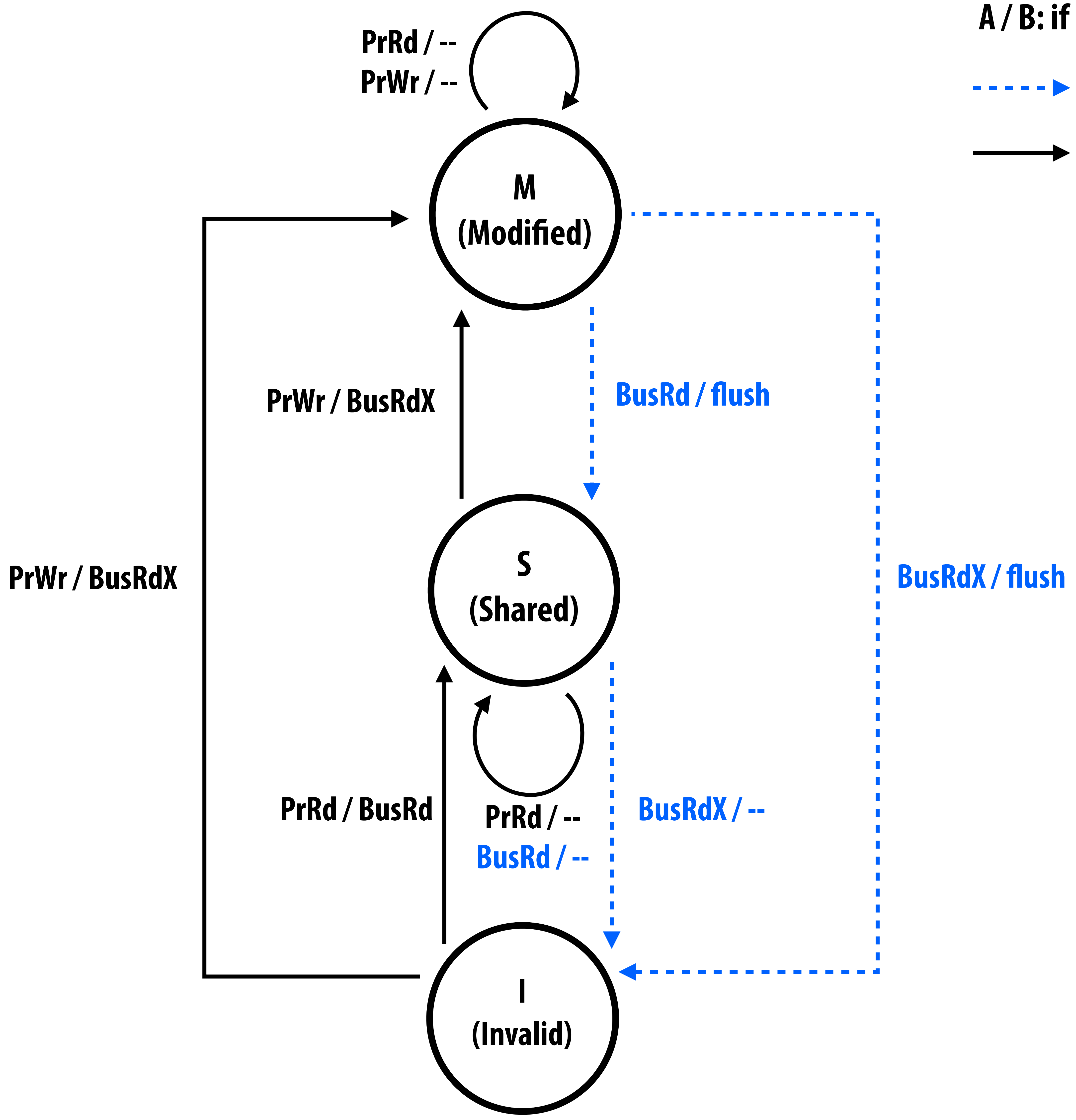
- **Processor can only write to lines in the exclusive state**
  - If processor performs a write to line that is not exclusive in cache, cache controller must first broadcast a <u>read-exclusive</u> transaction
  - Read-exclusive tells other caches about impending write ("you can't read any more, because I'm going to write")
  - Read-exclusive transaction is required even if line is valid (but not exclusive) in processor's local cache
  - Dirty state implies exclusive

- **When cache controller snoops a read exclusive for a line it contains**
  - Must invalidate the line in its cache

Basic MSI write-back invalidation protocol

- **Key tasks of protocol**
  - Obtaining exclusive access for a write
  - Locating most recent copy of data on cache miss
- **Cache line states**
  - Invalid (I)
  - Shared (S): line valid in one or more caches
  - Modified (M): line valid in exactly one cache (a.k.a. "dirty" or "exclusive" state)
- **Processor events**
  - PrRd (read)
  - PrWr (write)
- **Bus transactions**
  - BusRd: obtain copy of line with no intent to modify
  - BusRdX: obtain copy of line with intent to modify
  - BusWB: write line out to memory

MSI state transition diagram



A / B: if action A is observed by cache controller, action B is taken

- - - ▶ Remote processor (coherence) initiated transaction

——▶ Local processor initiated transaction

flush = flush dirty line to memory

Alternative state names:
- E (exclusive, read/write access)
- S (potentially shared, read-only access)
- I (invalid, no access)

Does MSI satisfy coherence?

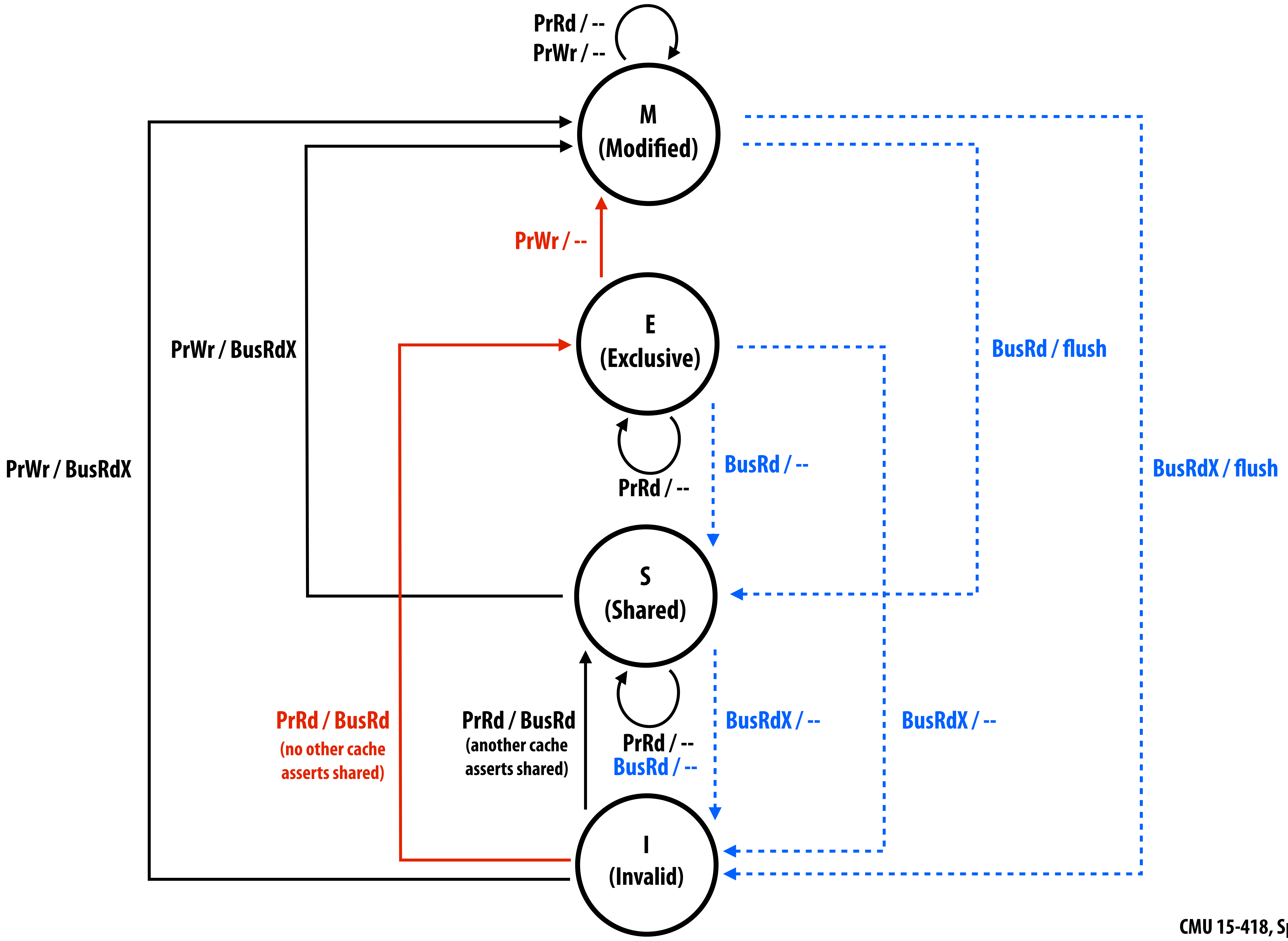
- **Write propagation**
  - Via combination of invalidation on BusRdX, and flush from M-state on subsequent busRd/busRdX from another processors
- **Write serialization**
  - Writes that appear on interconnect are ordered by the order they appear on interconnect (BusRdX)
  - Reads that appear on interconnect are ordered by order they appear on interconnect (BusRd)
  - Writes that don't appear on the interconnect (PrWr to line already in M state):
    - Sequence of writes to line comes between two bus transactions for the line
    - All writes in sequence performed by same processor, P (that processor certainly observes them in correct sequential order)
    - All other processors observe notification of these writes only after a bus transaction for the line. So all the writes come before the transaction.
    - So all processors see writes in the same order.

MESI invalidation protocol

MESI, not Messi !

- MSI requires two bus transactions for the common case of reading data, then writing to it
  - Transaction 1: BusRd to move from I to S state
  - Transaction 2: BusRdX to move from S to M state
- This inefficiency exists even if application has no sharing at all
- Solution: add additional state E ("exclusive clean")
  - Line not modified, but only this cache has copy
  - Decouples exclusivity from line ownership (line not dirty, so copy in memory is valid copy of data)
  - Upgrade from E to M does not require a bus transaction

MESI state transition diagram



Lower-level choices

- Who should supply data on a cache miss when line is in the E or S state of another cache?
    - Can get data from memory or can get data from another cache
    - If source is another cache, which one should provide it?

- Cache-to-cache transfers add complexity, but commonly used today to reduce both latency of access and memory bandwidth requires

Increasing efficiency (and complexity)

- **MESIF (5-stage invalidation-based protocol)**
  - Like MESI, but one cache holds shared line in F state rather than S (F=”forward”)
  - Cache with line in F state services miss
  - Simplifies decision of which cache should service miss (basic MESI: all caches respond)
  - Used by Intel processors

- **MOESI (5-stage invalidation-based protocol)**
  - In MESI protocol, transition from M to S requires flush to memory
  - Instead transition from M to O (O=”owned, but not exclusive”) and <u>do not</u> flush to memory
  - Other processors maintain shared line in S state, one processor maintains line in O state
  - Data in memory is stale, so cache with line in O state must service cache misses
  - Used in AMD Opteron

Implications of implementing coherence

- Each cache must listen for and react to all coherence traffic broadcast on interconnect
  - Typical to duplicate cache tags so that tag lookup in response to coherence actions does not interfere with processor loads and stores (caches can check tags in response to processor and remote coherence message in parallel)

- Additional traffic on chip interconnect
  - Can be significant when scaling to higher core counts

- Most modern multi-core CPUs implement cache coherence

- To date, GPUs do not implement cache coherence
  - Thus far, overhead of coherence deemed not worth benefits for graphics and scientific computing applications (NVIDIA GPUs provide single shared L2 + atomic memory operations)

Implications to software developer

What could go wrong with this code?

```
// allocate per-thread variable for local accumulation
int myCounter[NUM_THREADS];
```

Better:

```
// allocate per thread variable for local accumulation
struct PerThreadState {
  int myCounter;
  char padding[64 - sizeof(int)];
};
PerThreadState myCounter[NUM_THREADS];
```

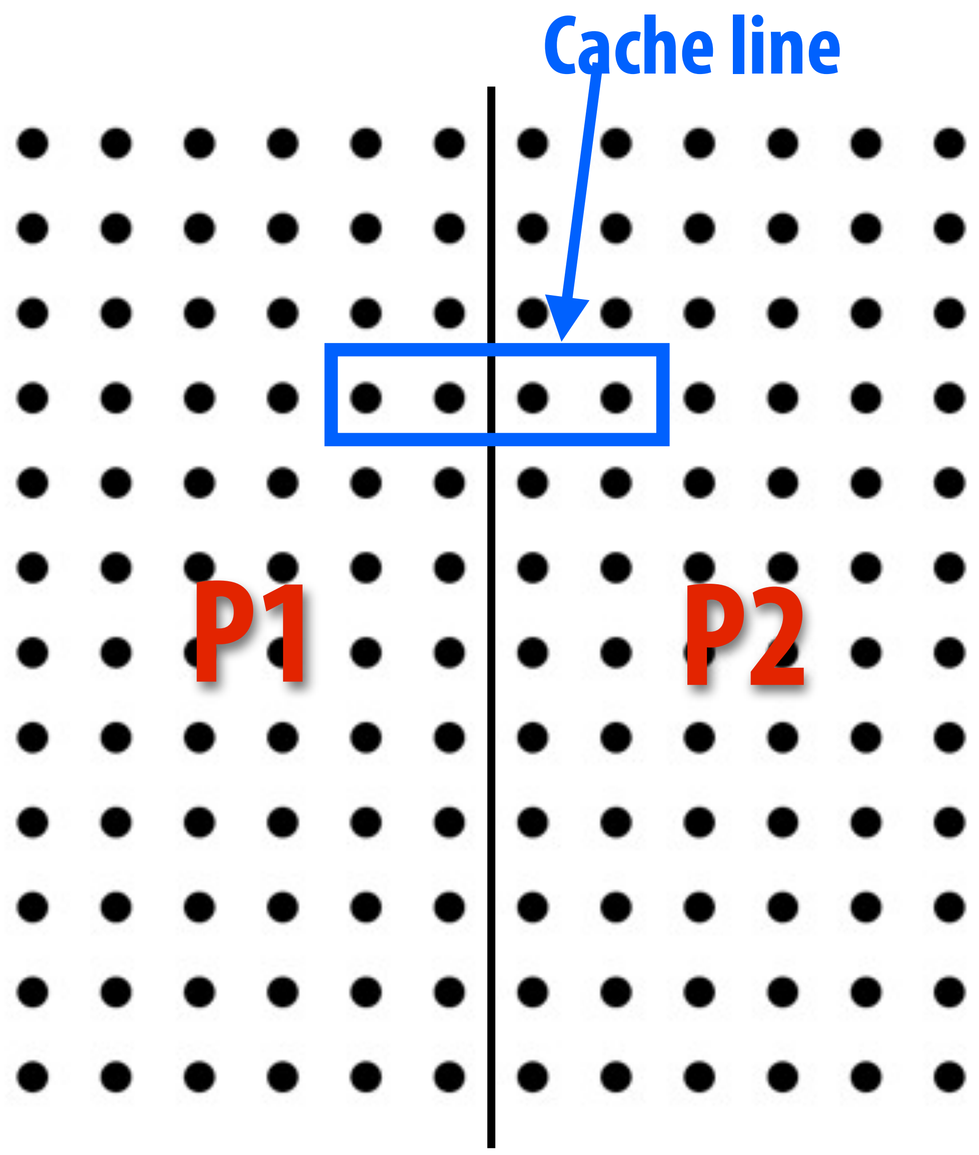
False sharing

- Condition where two processors write to different addresses, but addresses map to the same cache line

- Cache line "ping-pongs" between caches of writing processors, generating significant amounts of communication due to the coherence protocol

- No inherent communication, this is entirely <u>artifactual communication</u>

- Can be a factor in when programming for cache coherent architectures



Summary

- **The cache coherence problem exists because the abstraction of a single shared address space is not actually implemented by a single storage unit in a machine**
    - Storage is distributed among main memory and local processor caches

- **Main idea of snooping-based cache coherence: Whenever a cache operation occurs that could affect coherence, the cache controller broadcasts a notification to all other cache controllers.**
    - Challenge for HW architects: minimizing overhead of coherence implementation
    - Challenge for SW developers: be wary of artifactual communication due to coherence protocol (e.g., false sharing)