Lecture 13:
Directory-Based Cache Coherence II + Memory Consistency Models

CMU 15-418: Parallel Computer Architecture and Programming (Spring 2012)

Announcements

- **Exam 1: Tuesday March 6th**
  - Covers material up through next class

- **Supplemental readings**
  - On (1) GPU and SIMD architectures and (2) cache coherence
  - From Henessey and Patterson, 5th edition

- **Exam review session Sunday afternoon**

- **Assignment 3: Due Tuesday March 6th**

Today: what you should know

- Understand two challenges of implementing directory-based cache coherence
  - Reducing the overhead of directory storage
  - Reducing the number of messages required to implement coherence protocol

- How hoes memory consistency differ from memory coherence

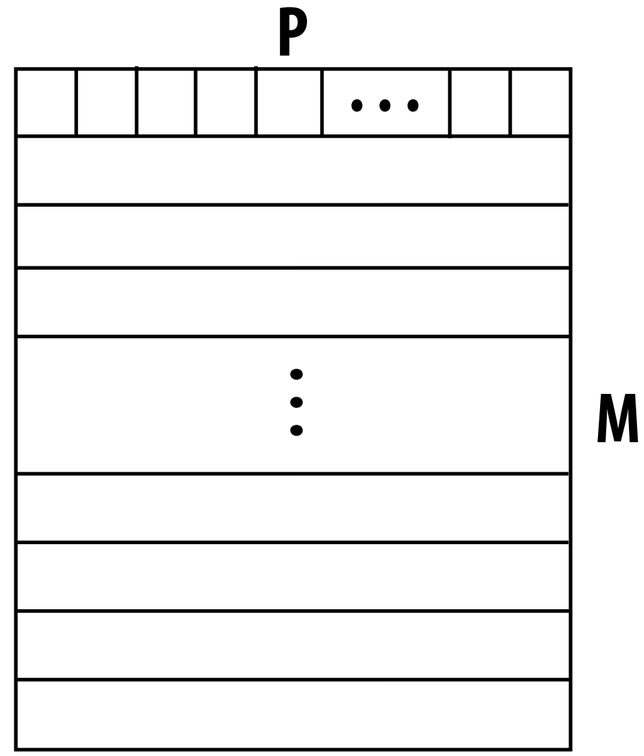
- Sequential consistency vs. relaxed consistency models
  - What is the motivation for relaxed consistency?

When we left off last time...

- We discussed ways to reduce the storage overhead of the directory structure needed for scalable cache coherence

- Last time: reduce P using limited pointer scheme
  - Most of the time, the number of sharers is low (rows of the directory are sparse)
  - Store a fixed number of pointers to sharers per block
  - Have a reasonable fallback when actual sharing exceeds this limit

- What about reducing M?



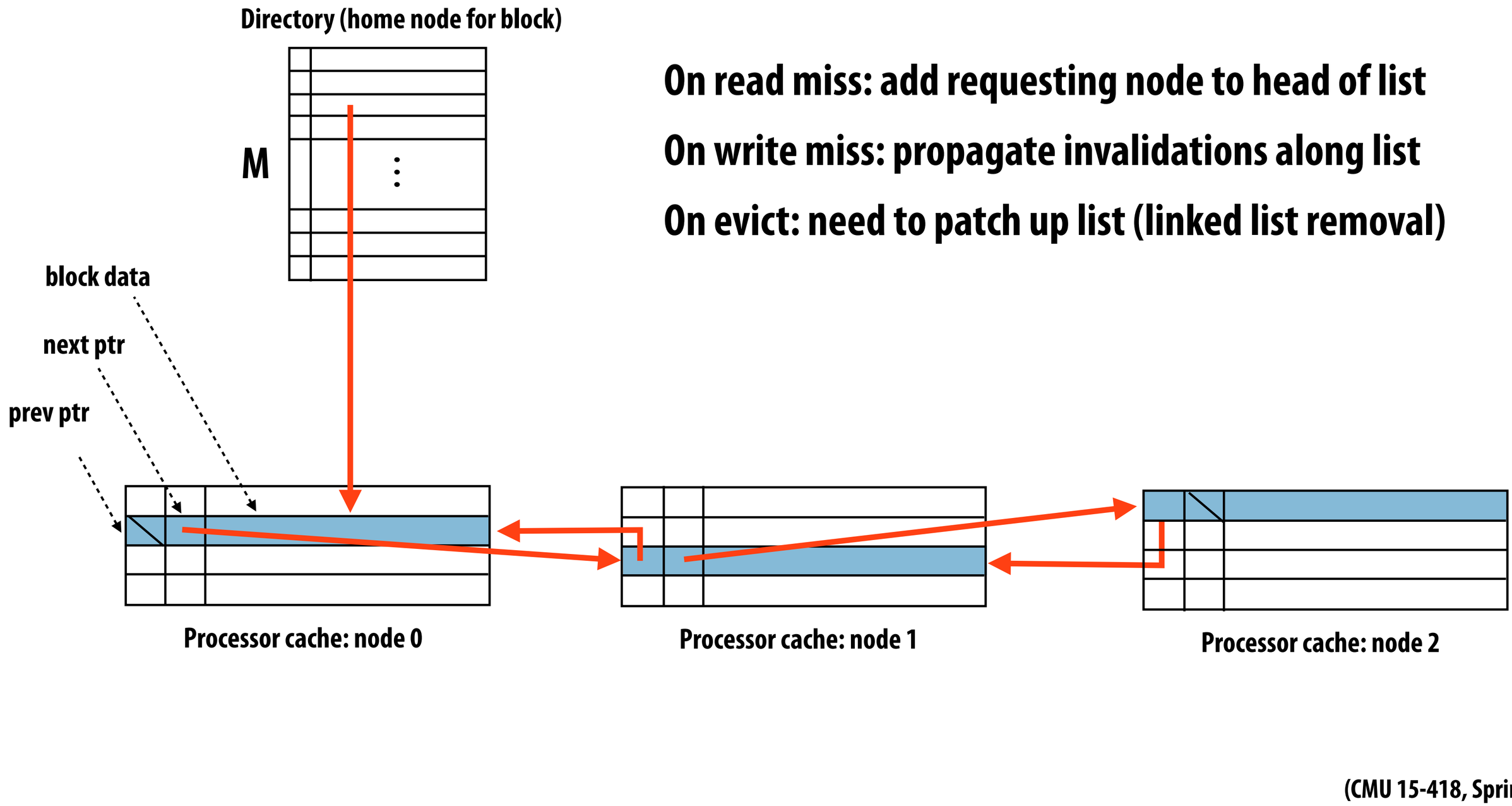
Limiting size of directory: sparse directories

- Majority of memory is NOT resident in cache. Coherence protocol only needs sharing information for cached blocks
  - So most directory entries are "idle" most of the time
  - 1 MB cache, 1 GB memory per node → 99.9% of directory entries are idle

Sparse directories

Directory at home node maintains pointer to only one node caching block (not a list of sharers)

Pointer to next node stored in the cache line



On read miss: add requesting node to head of list

On write miss: propagate invalidations along list

On evict: need to patch up list (linked list removal)

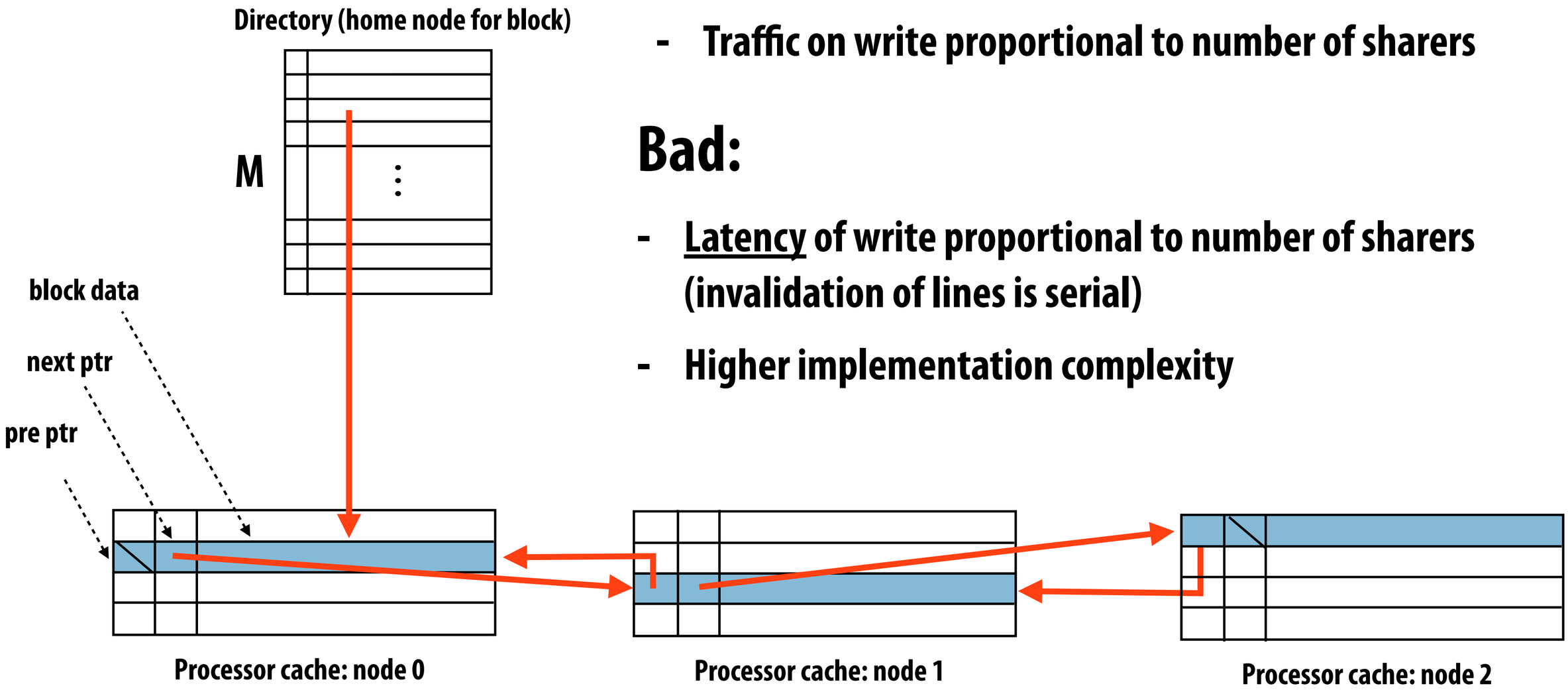
Sparse directories: scaling properties

Good:

- Low memory storage overhead (one pointer per block)
- Storage proportional to cache size (and list stored in SRAM)
- Traffic on write proportional to number of sharers

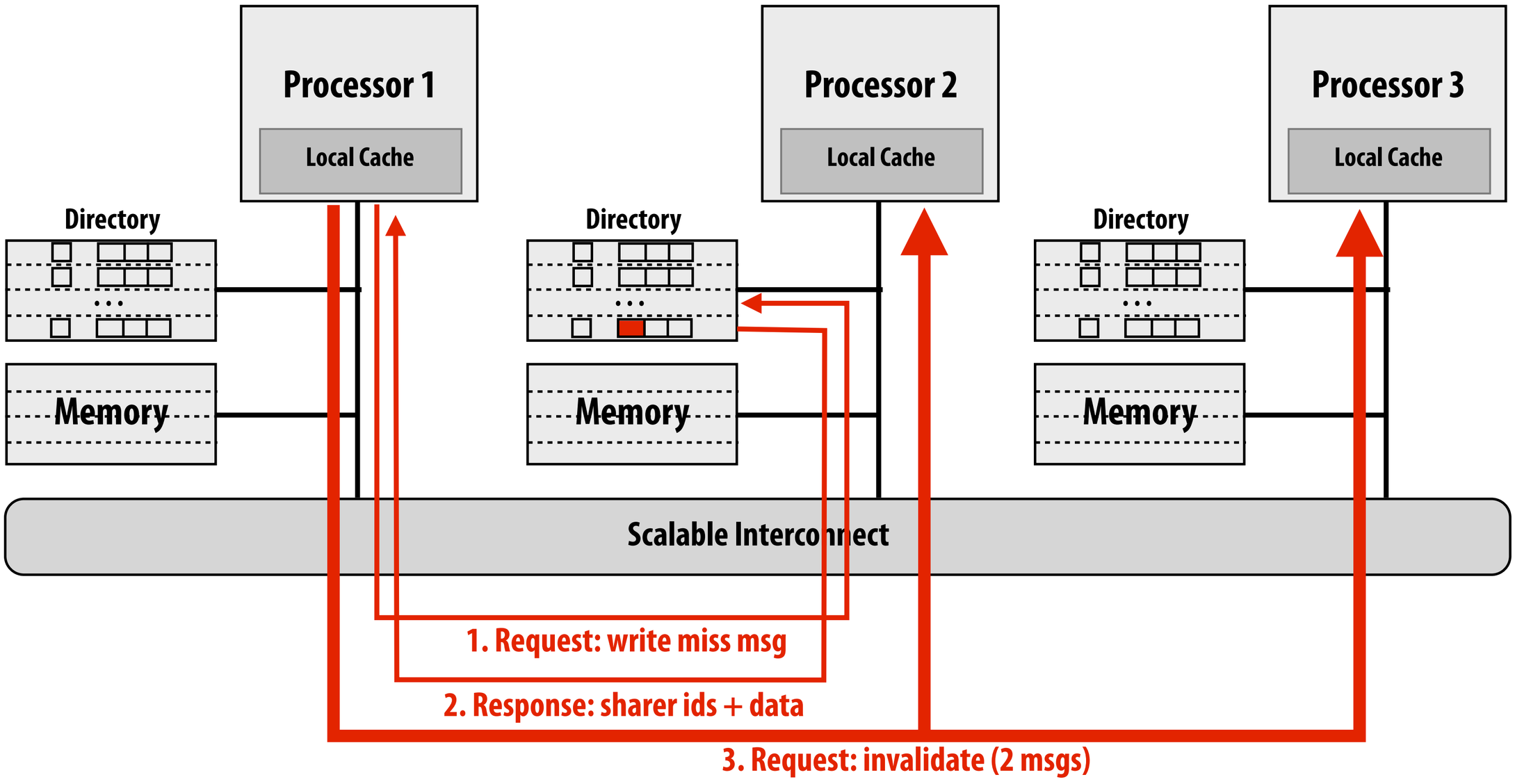
Bad:

- Latency of write proportional to number of sharers (invalidation of lines is serial)
- Higher implementation complexity



Recall: write miss in full bit vector scheme

Write to memory by processor 1: block is clean, but resident in P2's and P3's caches



Original bit-vector scheme sends same number of invalidation messages as sparse directory approach, but invalidation messages can be sent to all processors in parallel

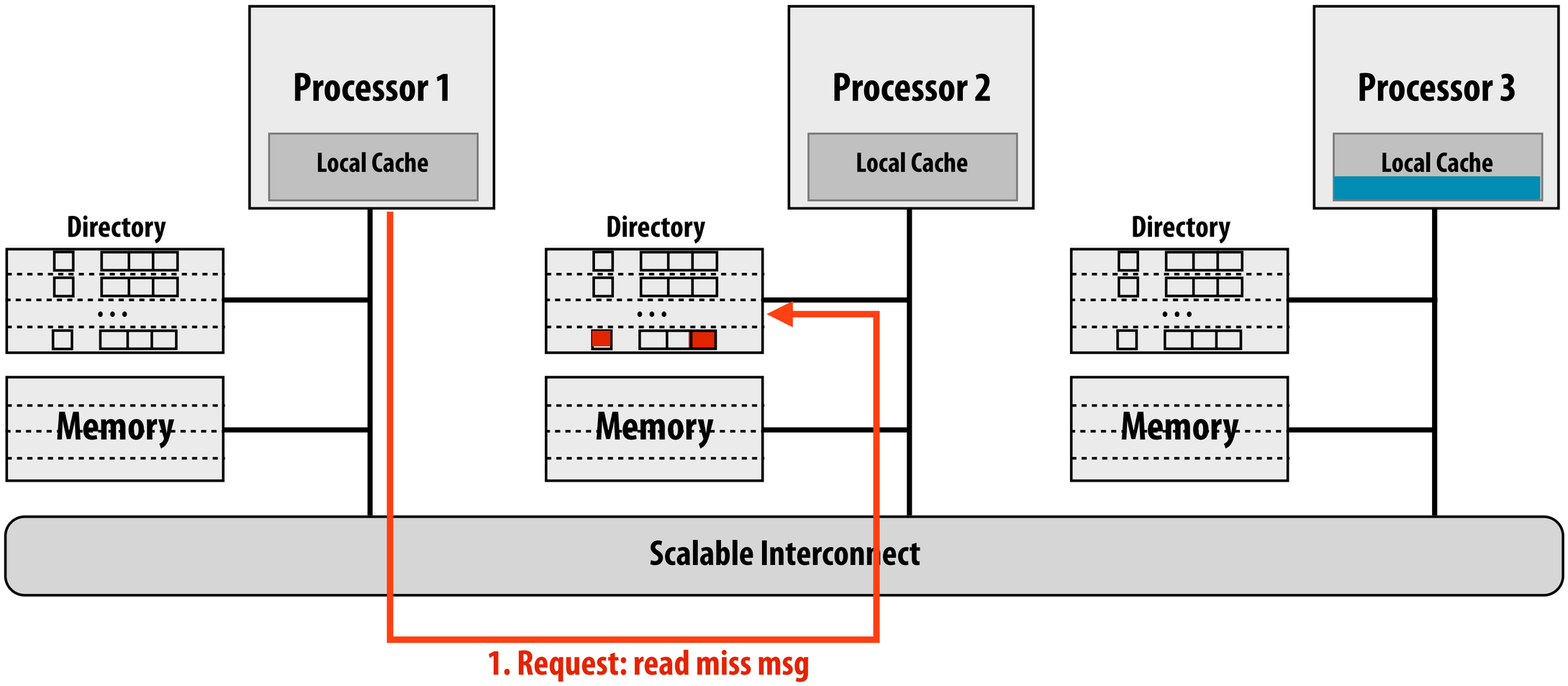
Optimizing directory-based coherence

- Reducing storage overhead of directory data structure

- Reducing number of messages sent to implement coherence protocol

Recall: read miss to dirty block

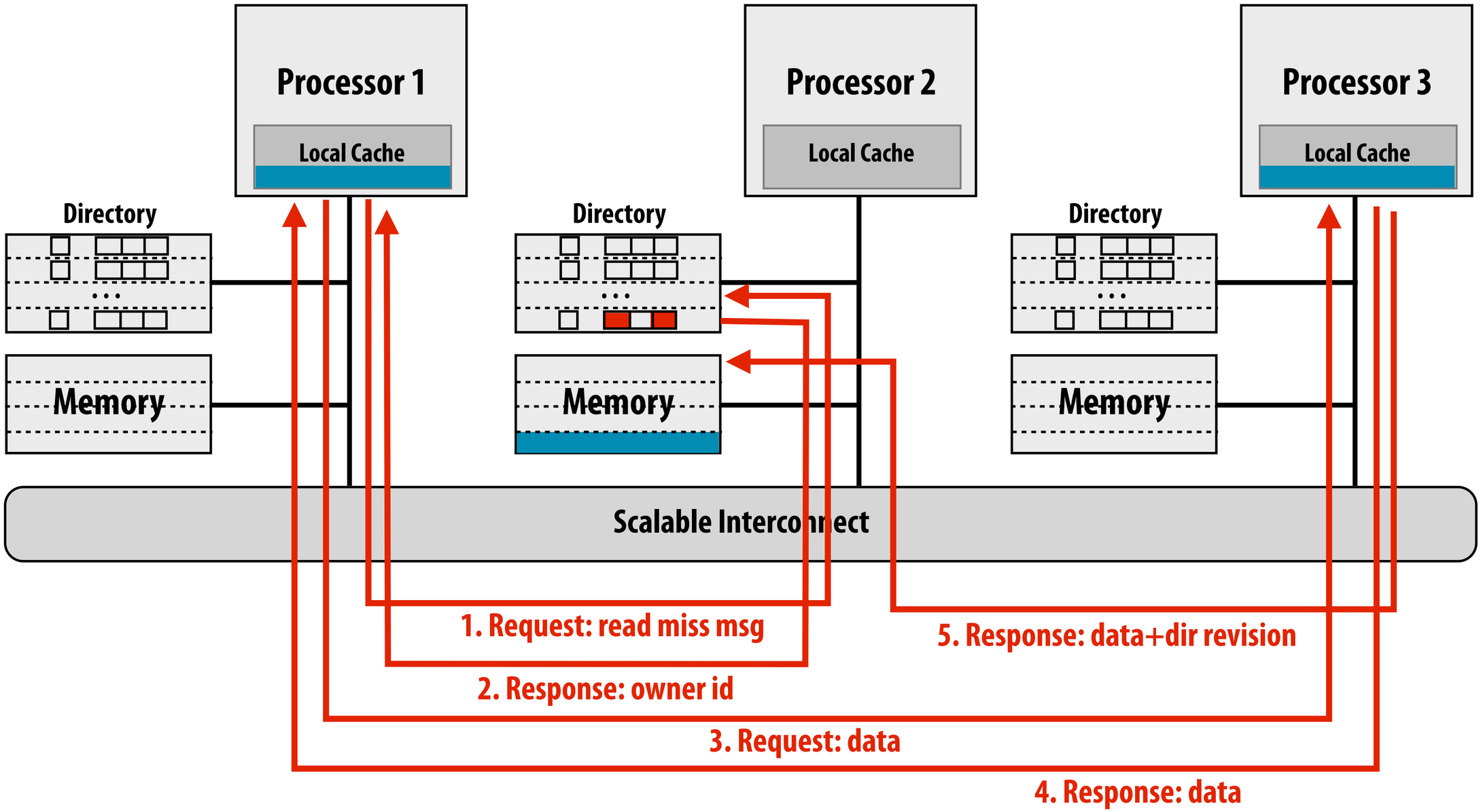
Read from main memory by processor 1 of the blue block: block is dirty (contained in P3's cache)



Recall: read miss to dirty block

Read from main memory by processor 1 of the blue block: block is dirty (contained in P3's cache)

(Note: figure below shows final state of system after operation is complete)



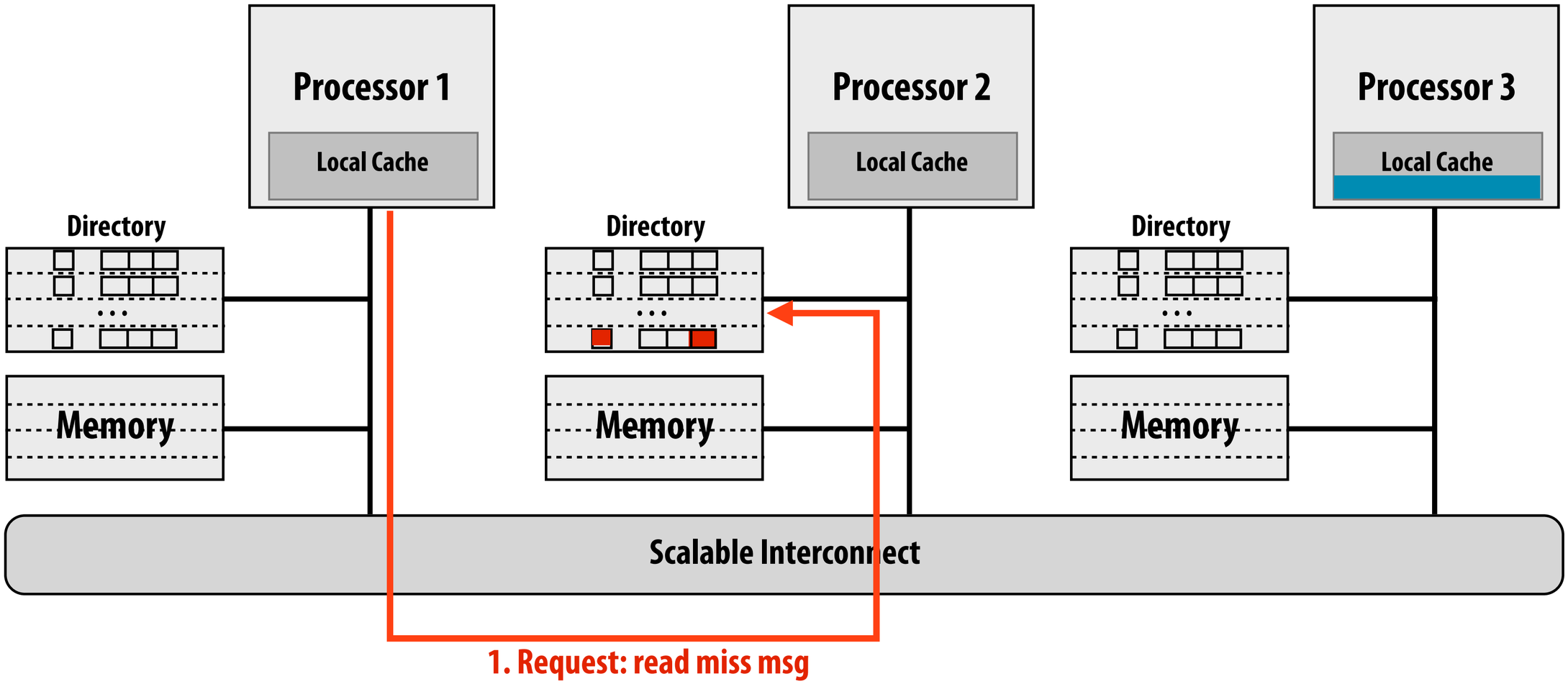
Five network transactions in total

Four of the transactions are on the "critical path" (transactions 4 and 5 can be done in parallel)

- Critical path: sequence of dependent operations that must occur to complete operation

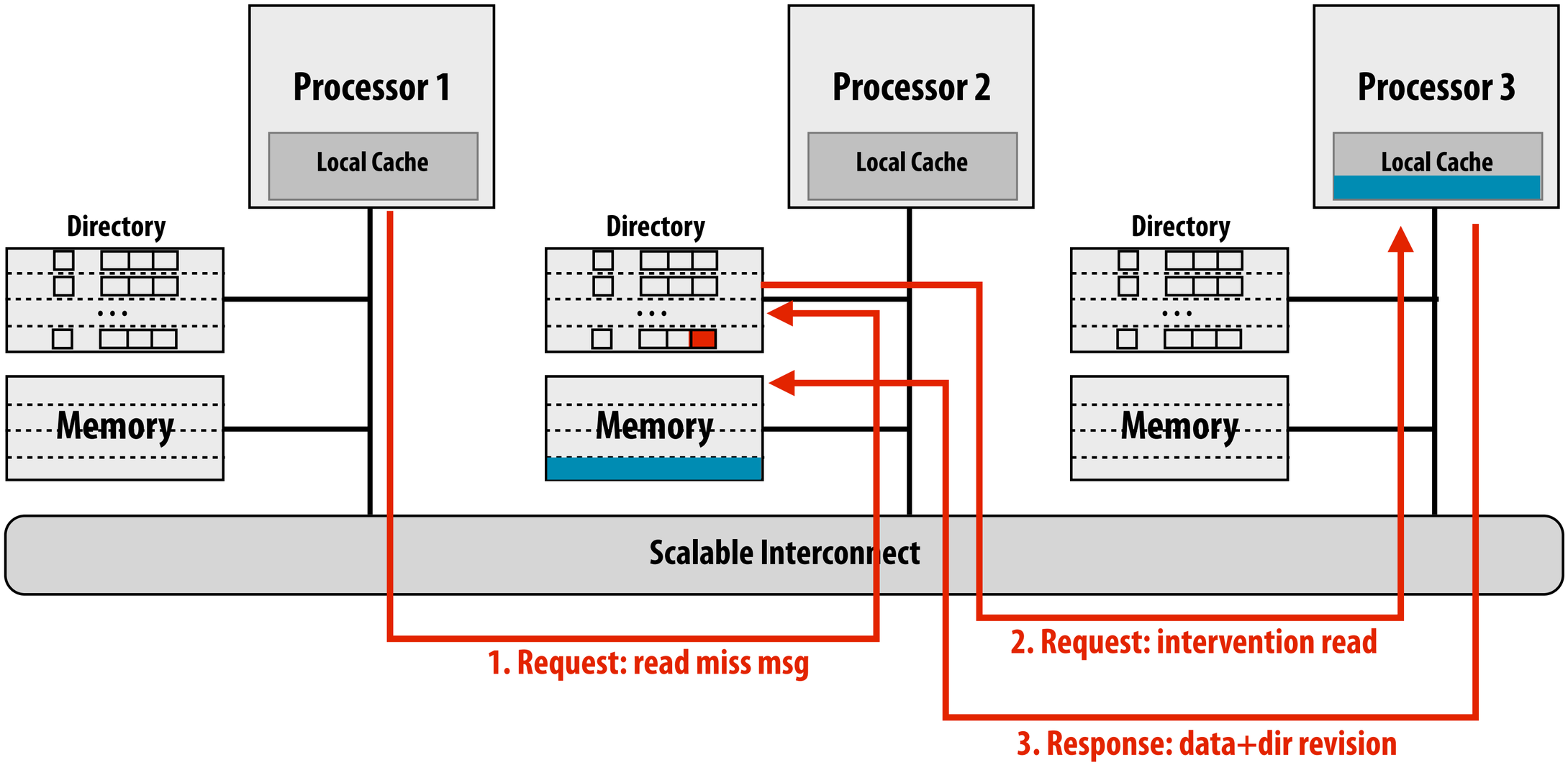
Intervention forwarding

Read from main memory by processor 1 of the blue block: block is dirty (contained in P3's cache)



Intervention forwarding

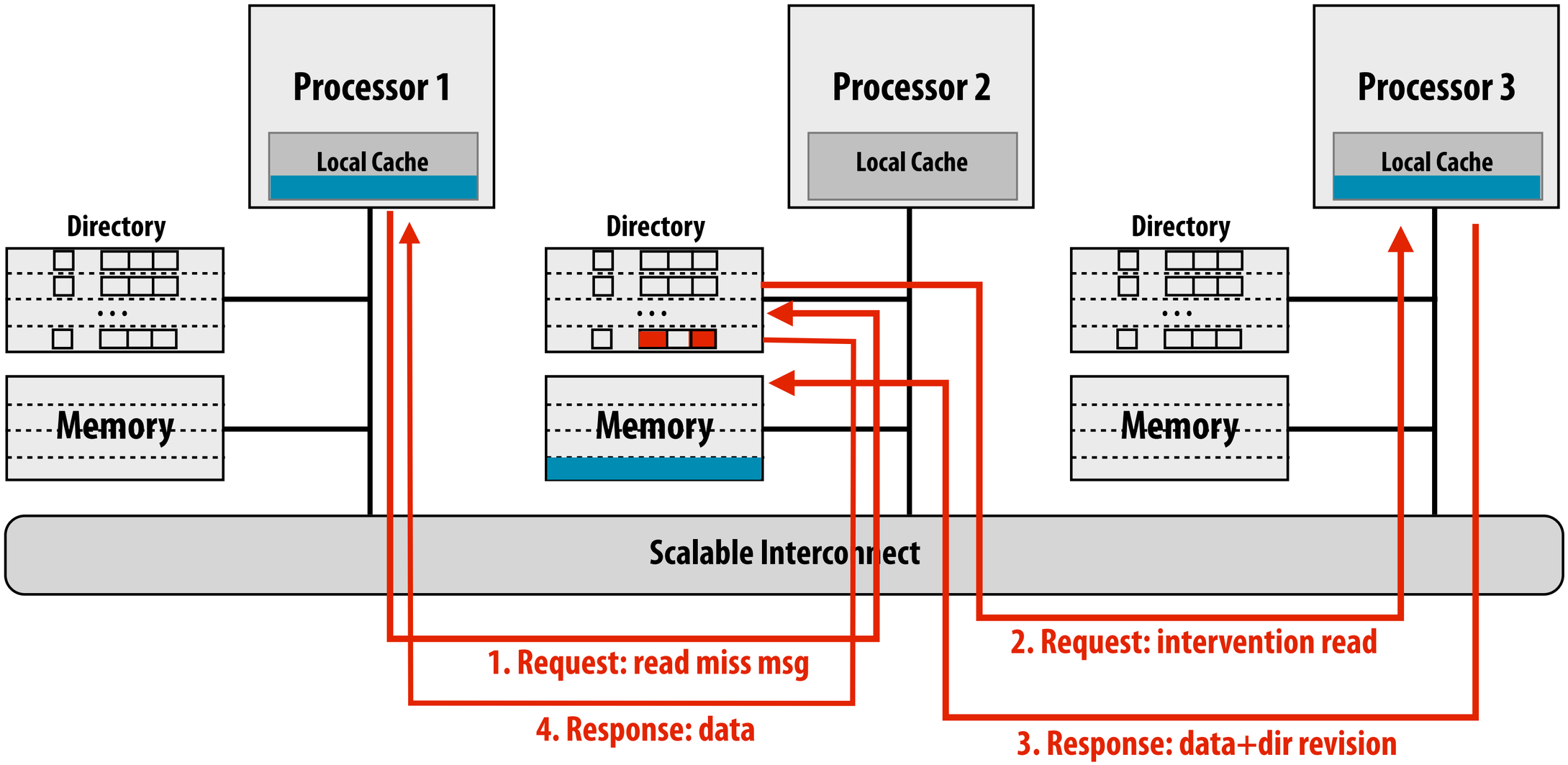
Read from main memory by processor 1 of the blue block: block is dirty (contained in P3's cache)



2. Home node requests data from owner node (processor 3)
3. Owning node responds

Intervention forwarding

Read from main memory by processor 1 of the blue block: block is dirty (contained in P3's cache)



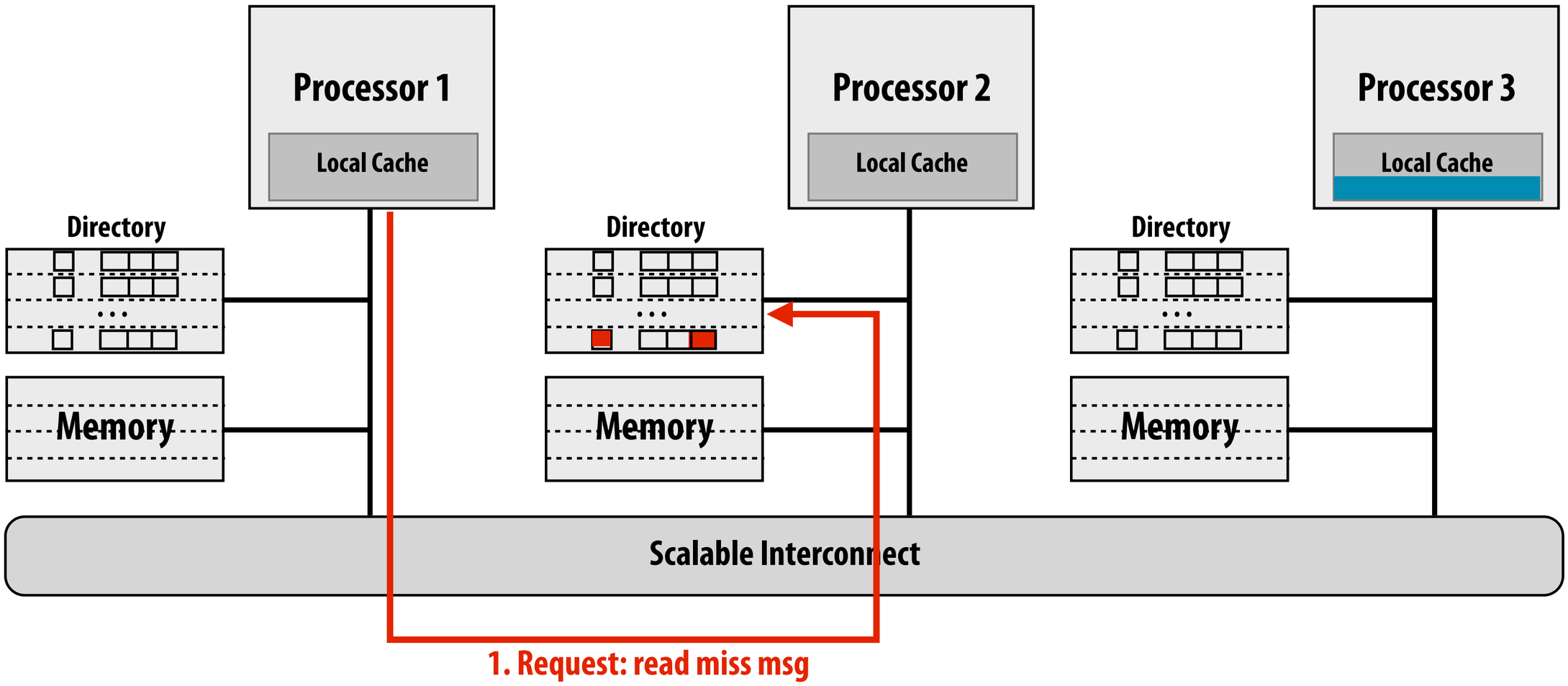
4. Home node updates directory, and responds to requesting node with data

Four network transactions in total (less traffic)

But all four of the transactions are on the "critical path"

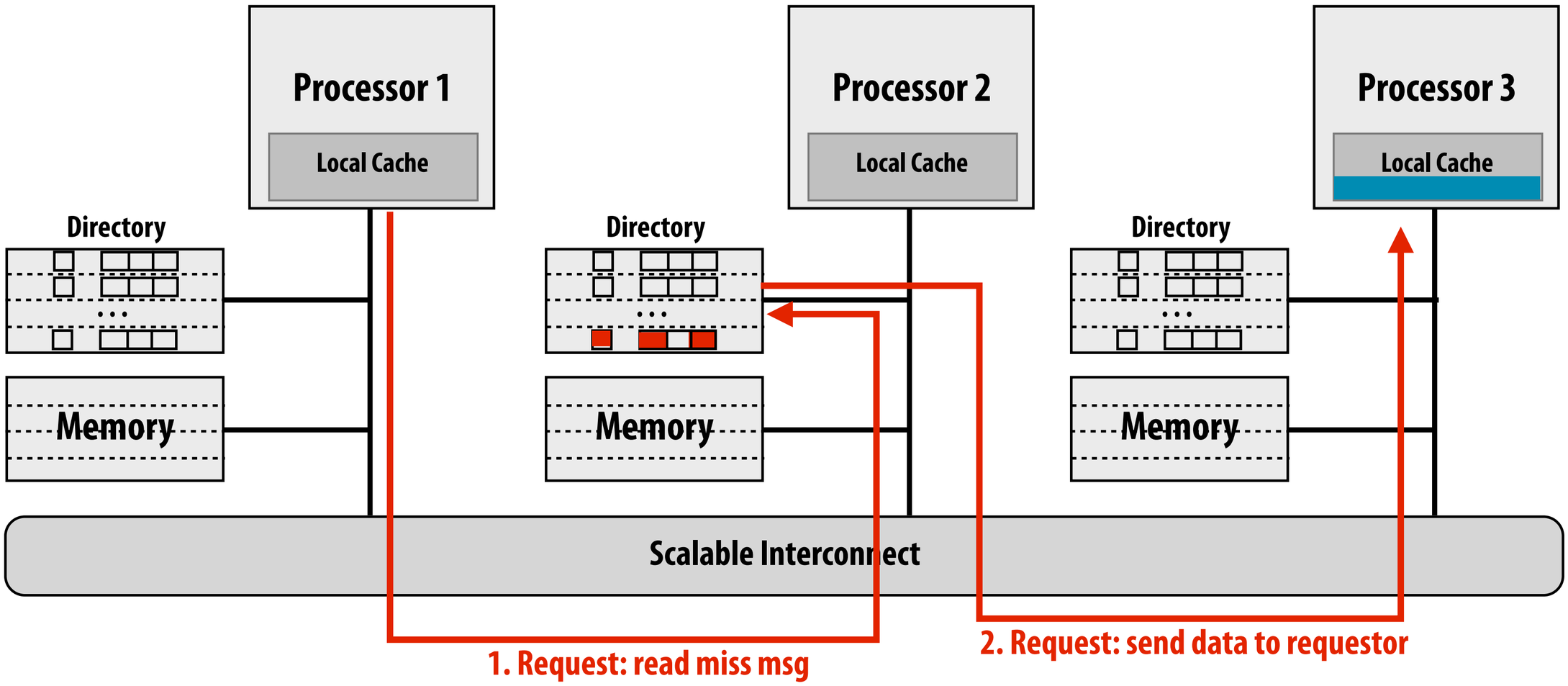
Request forwarding

Read from main memory by processor 1 of the blue block: block is dirty (contained in P3's cache)



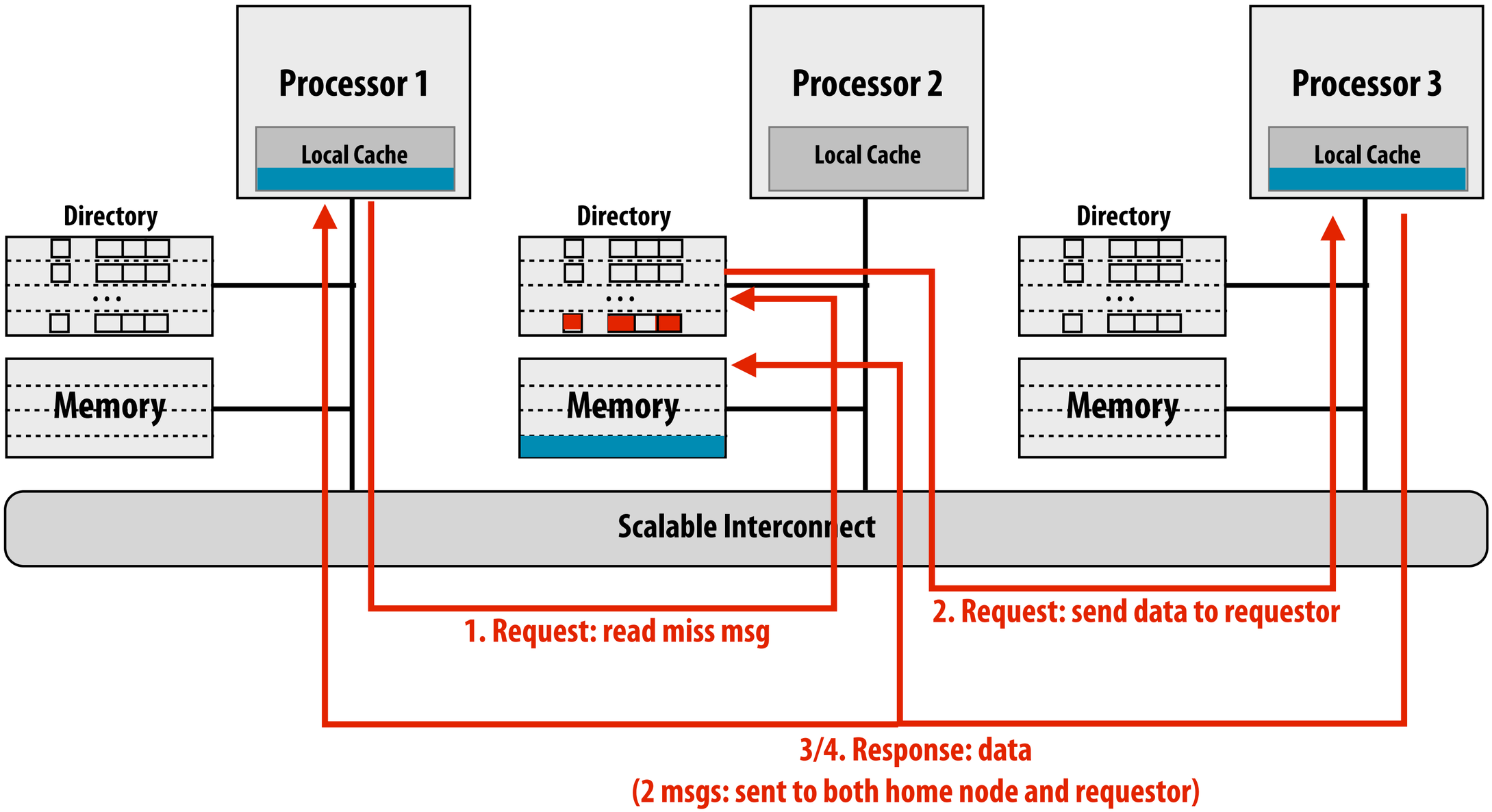
Request forwarding

Read from main memory by processor 1 of the blue block: block is dirty (contained in P3's cache)



Request forwarding

Read from main memory by processor 1 of the blue block: block is dirty (contained in P3's cache)



Four network transactions in total

Only three of the transactions are on the critical path (transactions 3 and 4 can be done in parallel)

Summary: directory-based coherence

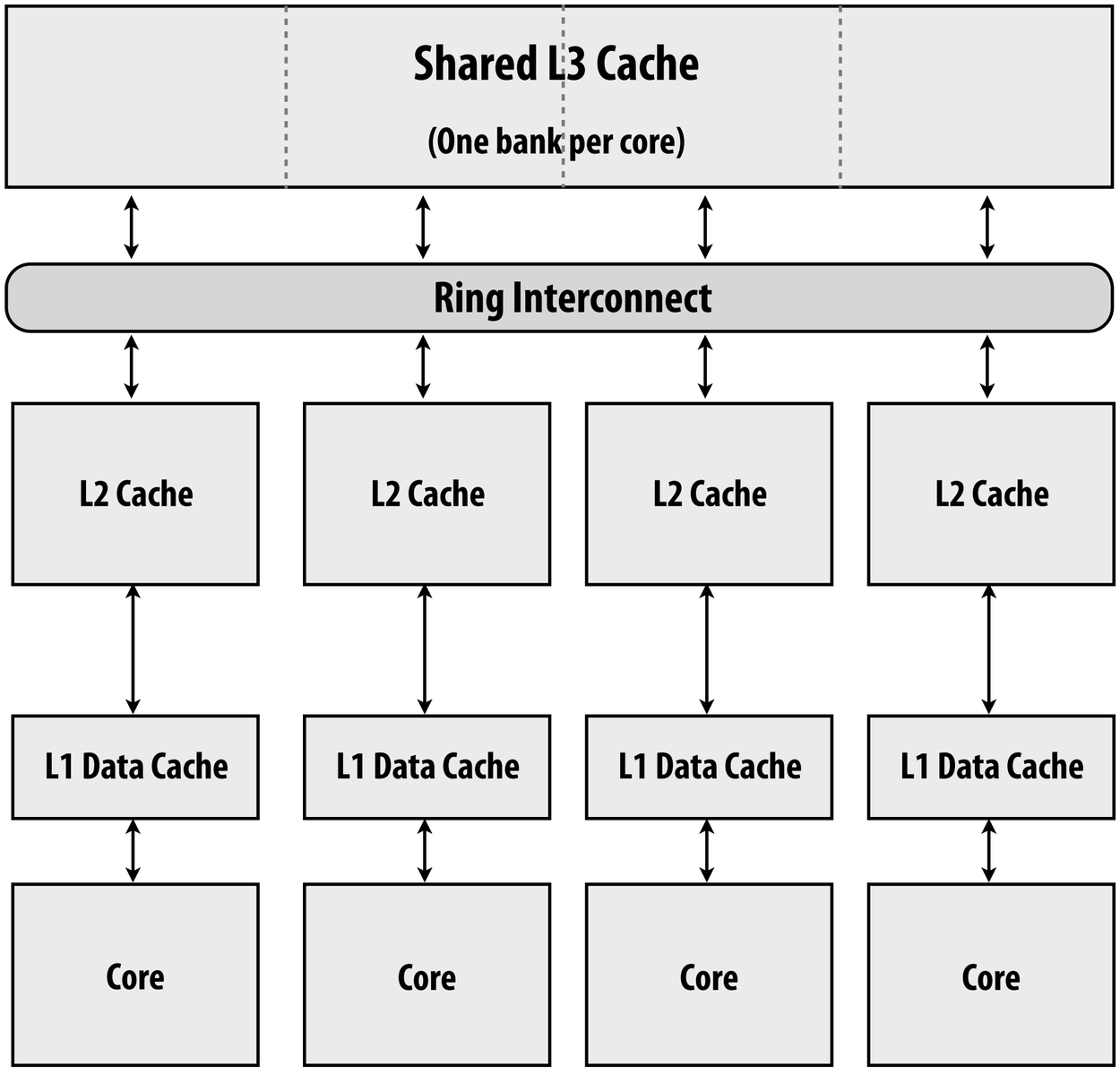
- Primary observation: broadcast doesn't scale, but luckily we don't need to broadcast to ensure coherence because often the number of caches containing a block is small

- Instead of snooping, just store the list of sharers in a "directory" and check the list as necessary

- One challenge: reducing overhead of directory storage
  - use hierarchies of processors or larger block sizes
  - limited pointer schemes: exploit fact the most processors not sharing block
  - sparse directory schemes: exploit fact that most blocks are not in cache

- Another challenge: reducing the number of messages sent (traffic) and critical path (latency) of message chains needed to implement coherence operations

Intel Core i7 CPU



- Centralized directory for all blocks in the L3 cache
  (note importance of inclusion property)
- Directory maintains list of L2 caches containing block
- Instead of broadcasting coherence traffic to all L2's, only send coherence messages to L2's that contain the block
  (remember interconnect is a ring, not a bus)
- Directory dimensions:
  - P=4
  - M = number of L3 cache blocks

Memory Consistency

Terminology

How does processor 1 "observe" the result of a write by processor 2 to address A?

It reads address A

Processor 2 writes value X to address A

Processor 1 "observes" or "sees" the write if its read of address A returns the value X

Coherent memory ensures that if processor 1's read comes sufficiently long after processor 2's write, processor 1 will observe the write.

Thread 1 (on processor 1)

```
A = 0;
...
A = 1;
if (B == 0)
{
    ...
}
```

Thread 2 (on processor 2)

```
B = 0;          // A and B be initialized to 0 here
...
B = 1;
if (A == 0)
{
    ...
}
```

Assume coherent shared memory system:
Can code enter the "if" clause in both threads?

Thread 1 (on processor 1)

```
A = 0;
...
A = 1;
if (B == 0)
{
    ...
}
```

Thread 2 (on processor 2)

```
B = 0;
...
B = 1;
if (A == 0)
{
    ...
}
```

Assume: processors can proceed past writes
Let: A and B be resident in shared state in both processors caches (after the "...")

time	Processor 1 operations	Processor 2 operations
↓	Attempt write (A) // can't proceed (block not exclusive)	Attempt write (B) // can't proceed (block not exclusive)
	Send invalidate (A) to P2	Send invalidate (B) to P1
	Read (B) // observe value 0 from cached value of B	Read (A) // observe value 0 from cached value of A
	Recv invalidate (B), invalidate block (B), send ack	Recv invalidate (A), invalidate block (A), send ack
	Receive ack on (A) from P2 // block now exclusive	Receive ack on (B) from P2 // block now exclusive
	Complete write (A) // update value in cache	Complete write (B) // update value in cache

Thread 1 (on processor 1)

```
A = 0;
...
A = 1;
if (B == 0)
{
    ...
}
```

Thread 2 (on processor 2)

```
B = 0;
...
B = 1;
if (A == 0)
{
    ...
}
```

time	Processor 1 operations	Processor 2 operations
	Attempt write (A) // can't proceed (block not exclusive)	Attempt write (B) // can't proceed (block not exclusive)
	Send invalidate (A) to P2	Send invalidate (B) to P1
	Read (B) // observe value 0 from cached value of B	Read (A) // observe value 0 from cached value of A
	Recv invalidate (B), invalidate block (B), send ack	Recv invalidate (A), invalidate block (A), send ack
	Receive ack on (A) from P2 // block now exclusive	Receive ack on (B) from P2 // block now exclusive
	Complete write (A) // update value in cache	Complete write (B) // update value in cache

Result: both threads end up entering IF clause.

Should this behavior be allowed?

Another example: potentially frustrating if you were a developer

Thread 1 (on processor 1) **Thread 2 (on processor 2)**

(Let A be initialized to 0)

```
A = 1;                    while (flag == 0);
flag = 1;                 print A;
```

What possible outcomes do you expect?

Thread 1 (on processor 1) **Thread 2 (on processor 2)**

(Let A and B be initialized to 0)

```
A = 1;                  print B;
B = 2;                  print A;
```

```
0 0    // P2 completes before P1 starts

0 1    // P1 sets A, but not B, before first P2 print

2 1    // P1 completes before P2 starts printing

2 0    // by behavior on previous slide, this could happen
```

Memory consistency

- **Memory coherence**
  - Ensures all processors see a consistent view of memory
  - From defn. in previous lecture: all writes to SAME address should be seen by all processors in the same order (write serialization)
  - Writes to an address by one processor will eventually be observed by other processors. But when?

- **Memory consistency model**
  - Defines constraints on the order in which memory operations must appear to be performed (the "when")
  - Includes operations to same location, and to different locations

Sequential consistency (SC)

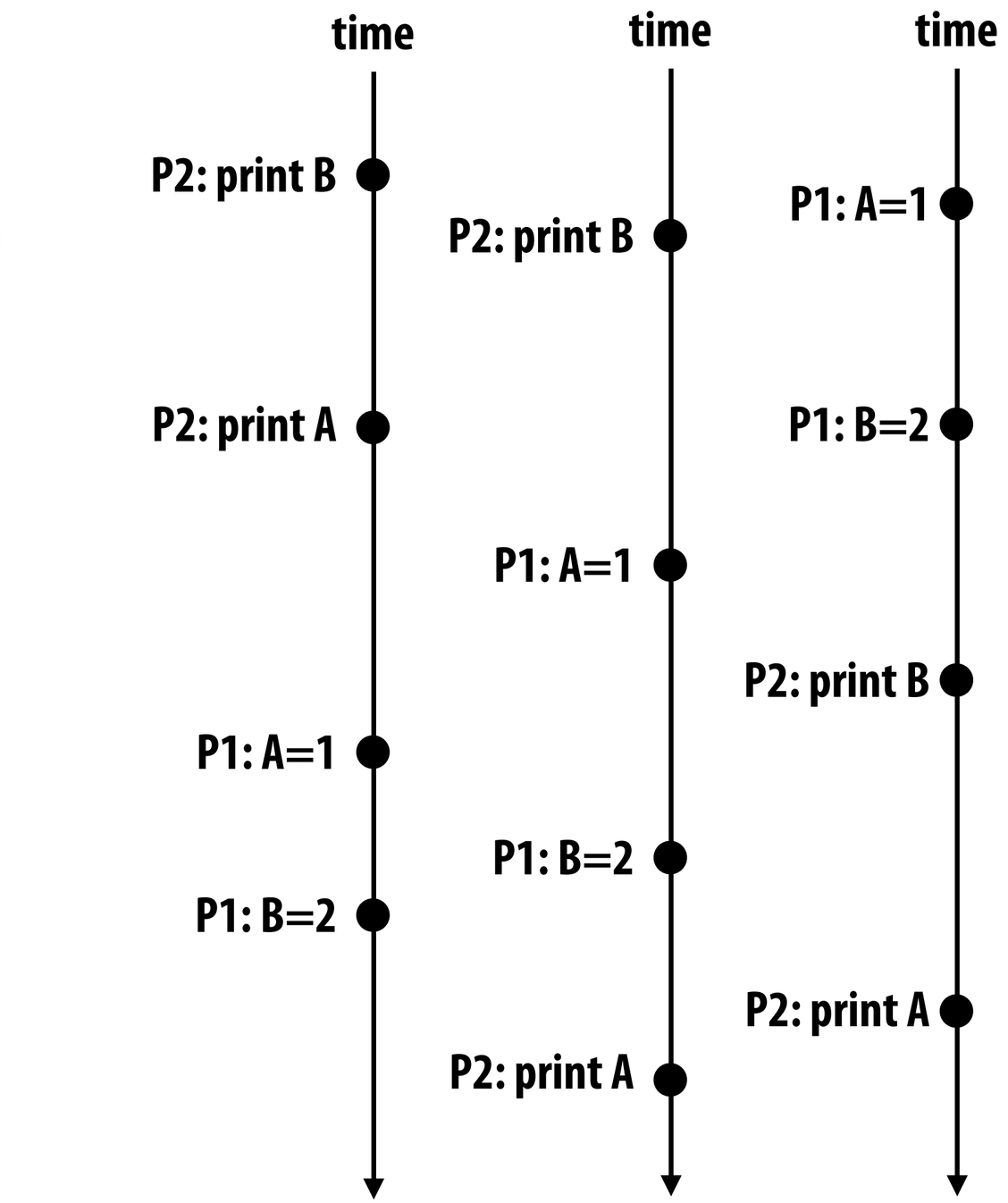
(Let A and B be initialized to 0)

Thread 1 (on processor 1)	Thread 2 (on processor 2)
`A = 1;`	`print B;`
`B = 2;`	`print A;`

Sequential consistency is most intuitive notion of consistency.

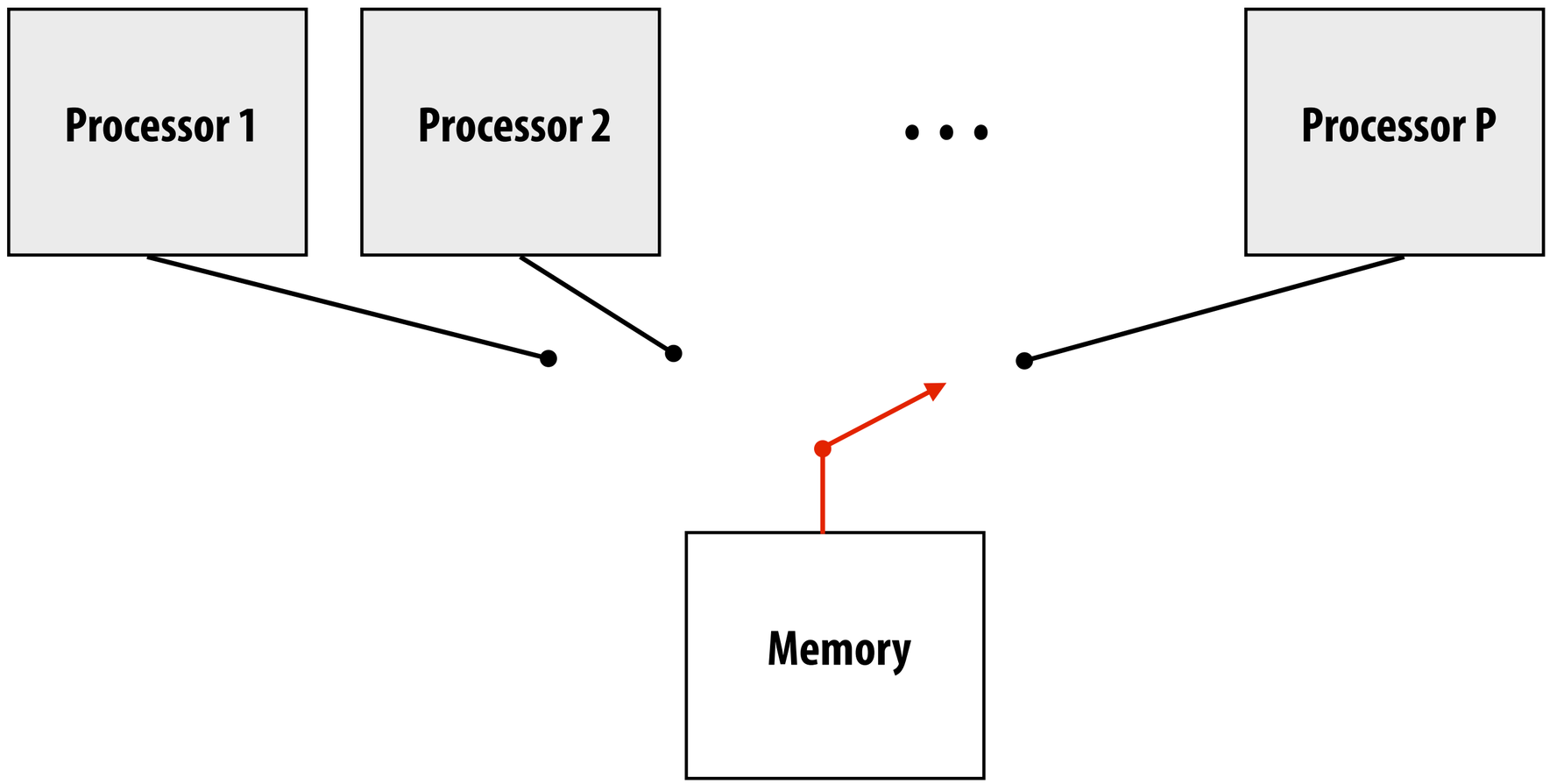
It requires that the result of program execution is consistent with an order where memory accesses from the same processor are kept in order and memory accesses from different processors are arbitrarily interleaved.

(Another way of saying this is: memory accesses by different processors are executed in some sequential order)



(Note that `2 0` is not an output that is allowed by sequential consistency)

Another way to think about SC



Processors appear to share a single logical memory.

Memory services requests from processors one at a time.

Sufficient conditions* for sequential consistency

1. Every process issues memory operations in program order

2. Each processor waits for its own writes to complete before continuing to next operation

3. When a processor reads, it waits for the write producing the value returned by the read to complete before continuing to next operation
   - In other words: if a processor observes a write, it waits for all processors to observe the write before continuing.

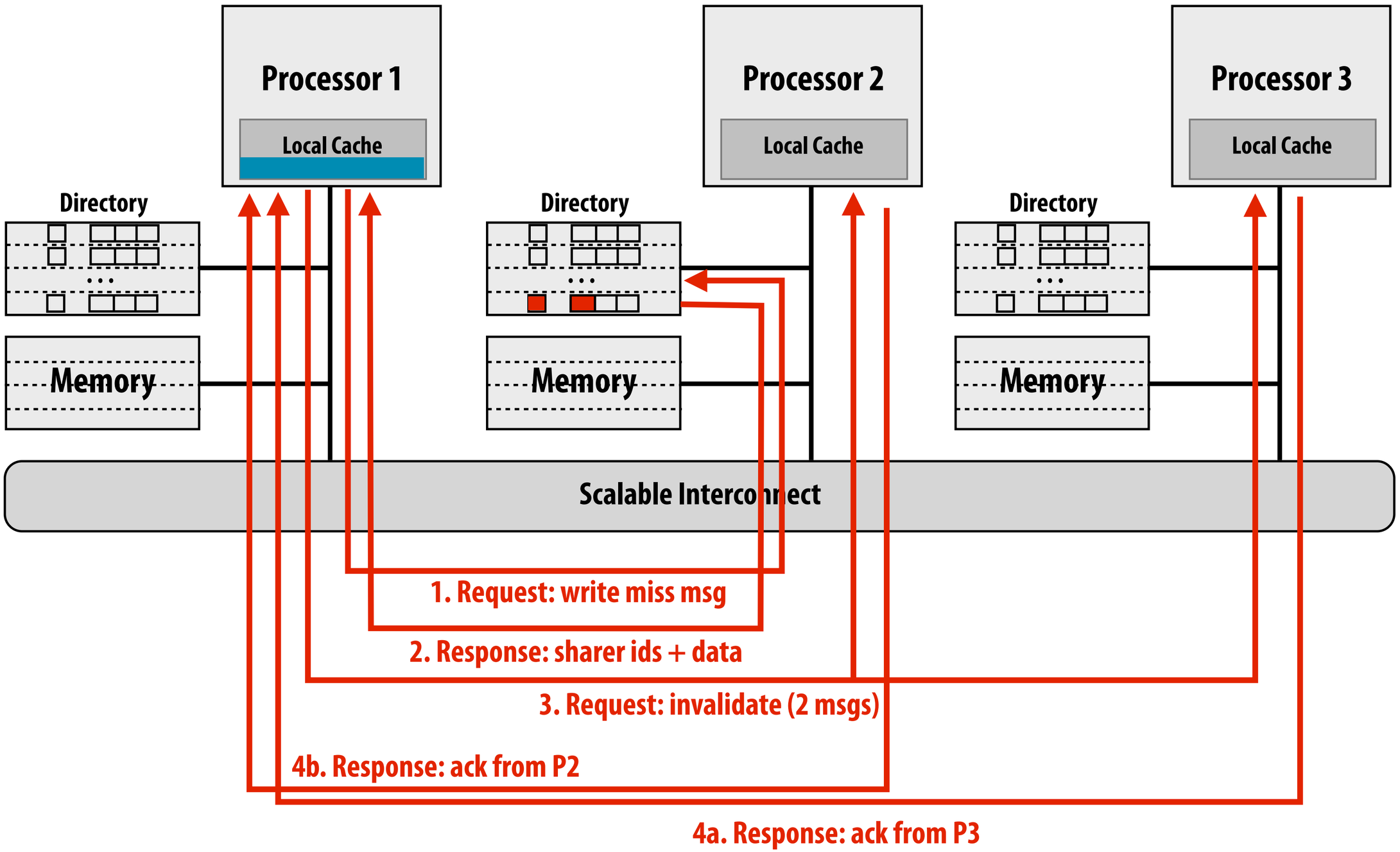
* sufficient, but not necessary (optimizations do exist)

Performance implications

- Sequential consistency provides intuitive semantics to the programmer, but incurs a performance penalty
  - Example 1: processor must wait for write to complete before issuing next operation (write completion can take a long time: send invalidations, receive acks, etc.)
  - Example 2: common compiler optimizations (instruction reordering, loop unrolling, common subexpression elimination) can cause sequential consistency to be violated even if hardware implements sequential consistency

Example: write miss in directory coherence protocol

Write to memory by processor 1: block is clean, but resident in P2's and P3's caches



Consider P1 performing two different writes.

After receiving both invalidation acks from first write, then P1 can perform write, then issue second write

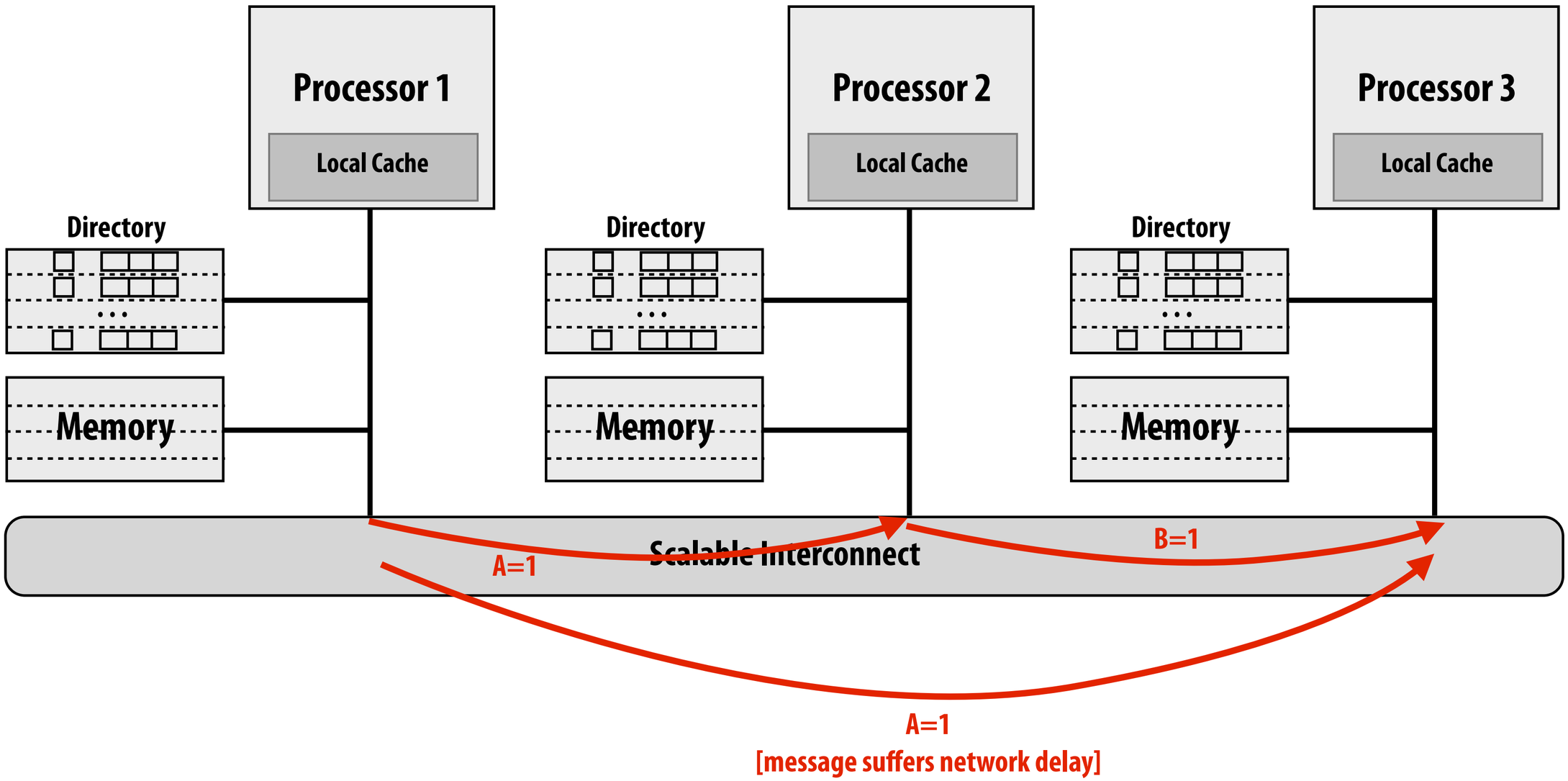
Why wait for acks? Why not just wait for directory to be updated? (2)

Example:

```
A = 1;
```

```
while (A==0);
B=1;
```

```
while (B==0);
print A;
```



Assume P1 does not wait for A=1 acks from P2 and P3 before sending B=1

Due to network delay, B=1 arrives at P3 before A=1

Original program code

Thread 1 (on P1)	Thread 2 (on P2)
`B = 0;`	`A = 0;`
`A = 1;`	`B = 1;`
`x = B;`	`y = A;`

After compiler register allocation

Thread 1 (on P1)	Thread 2 (on P2)
`r1 = 0;`	`r2 = 0;`
`A = 1;`	`B = 1;`
`x = r1;`	`y = r2;`
`B = r1;`	`A = r2;`

Code shown above-right is the result of very reasonable compiler optimizations: reordering operations on different memory locations

SC specifies original program cannot result in state x=0, y=0

SC specifies that modified program *must* result in x=0, y=0

Relaxed consistency models

- Allow reads and writes to complete out of order (for increased performance)
- Assumption is that in most programs, reads and writes to shared variables are ordered by explicit synchronization operations* rather than regular loads/stores to shared variables
  - As a result, program behaves as if the machine was sequentially consistent

```
P1:              P2:
lock()           lock()
A = 1;           print A;
unlock();        unlock();
```

```
P1:              P2:
A = 1;           barrier();
barrier();       print A;
```

* and that synchronization operations trigger flush of relevant outstanding memory operations

Relaxing memory operation orderings

- W→R, R→R, R→W, W→W
- Sequential consistency maintains all four orderings
- Relaxed memory consistency allows certain ordering to be violated
  - Relaxing W→R: Allows reads to just ahead of writes, but retains ordering among writes. "Total store ordering" (TSO), "processor consistency (PC)". Programs that operate under SC often also operate under TSO/PC without needing additional synchronization
  - Relaxing W→W: partial store ordering
  - Other variants that relax R→W and R→R

Summary

- Memory consistency models define constraints on the order in which memory operations must appear to be performed

- Sequential consistency
  - All processors share a single memory, memory services processors one at a time
  - Most intuitive (simple)
  - But has performance cost

- Relaxed memory consistency models gain performance be reducing constraints on order
  - Typically synchronization primitives used by application when specific orderings are required