Lecture 12: Directory-Based Cache Coherence

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

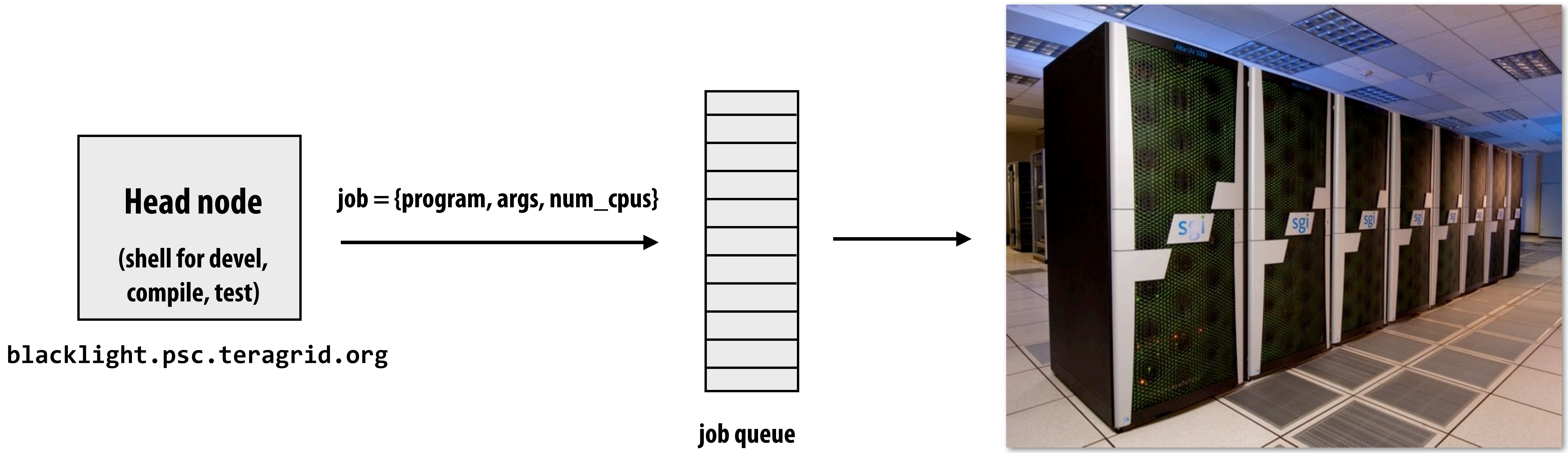
Assignment 3

- **Extending the due date to 11:59 pm on March 6th**
  (was previously March 2nd)

- **This is the day of Exam I**
  - **Previous deadline forced you to submit, then study for three days**
  - **Intent is to allow you to manage your time accordingly**

Assignment 3 primer

- **Run programs on Blacklight using a job queue system**
  - May have to wait a few minutes for completion


Head node
(shell for devel, compile, test)
blacklight.psc.teragrid.org
job = {program, args, num_cpus}
job queue

Blacklight Supercomputer

OpenMP

- API/runtime for writing parallel programs
  - C compiler directives
  - Runtime library routines
  - Builtin environment variables

- All programs start off executing serially
- Fork-join model of parallelism

sequential execution
(just regular C code)

SPMD parallel
execution

```c
#include <omp.h>

void main()  {

    int nthreads, tid;

    /* Check how many processors are available */

    printf("There are %d processors\n", omp_get_num_procs());

    /* Set the number of threads to 4 */
    omp_set_num_threads(4);


    /* Fork a team of threads giving them their own
       copies of variables */

#pragma omp parallel private(nthreads, tid)
    {

        /* Obtain and print thread id */
        tid = omp_get_thread_num();
        printf("Hello World from thread = %d\n", tid);


        /* Only master thread does this */
        if (tid == 0)
            {
                nthreads = omp_get_num_threads();
                printf("Number of threads = %d\n", nthreads);
            }

    }  /* All threads join master thread and terminate */

}
```

OpenMP

sequential execution
(just regular C code)

SPMD parallel
execution

```
#include <omp.h>

int numProcs = 10;

// Set the number of threads for the parallel region
omp_set_num_threads(numProcs);

// Fork a team of threads to execute the for loop in
// parallel
#pragma omp parallel for default(shared) private(i) schedule(dynamic)
    for (i=0; i < size; i++) {
        c[i] = a[i] + b[i];

    } // Implied barrier: all threads join master thread and
      // terminate
```

Dynamically assign iterations to pool of 10 threads
By default, treat variables as shared (like ISPC uniform)
Loop counter variable `i` is private per thread

Many ways to tell OMP how to assign iterations to threads (static assignment, blocked, interleaved, etc). See docs.

OpenMP

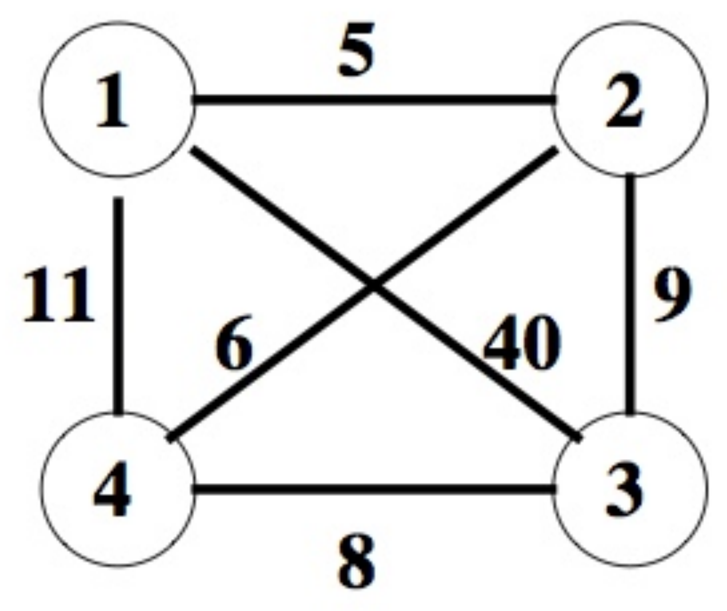
Basic synchronization example

```
#include <omp.h>

omp_lock_t indexLock;

void SlaveStart()
{
    // acquire lock
    omp_set_lock(&indexLock);

    // DO STUFF THAT REQUIRES MUTUAL EXCLUSION!

    // release lock
    omp_unset_lock(&gm->indexLock);

    // Stop at barrier to synchronize, not necessary
    // in this example (just an example of barrier syntax)
#pragma omp barrier
}

void main() {
   // Initialize the lock
   omp_init_lock(&gm->indexLock);

#pragma omp parallel
    {
        // Every thread, executes SlaveStart
        SlaveStart();
    }

   // Uninitialize the lock
   omp_destroy_lock(&indexLock);
}
```

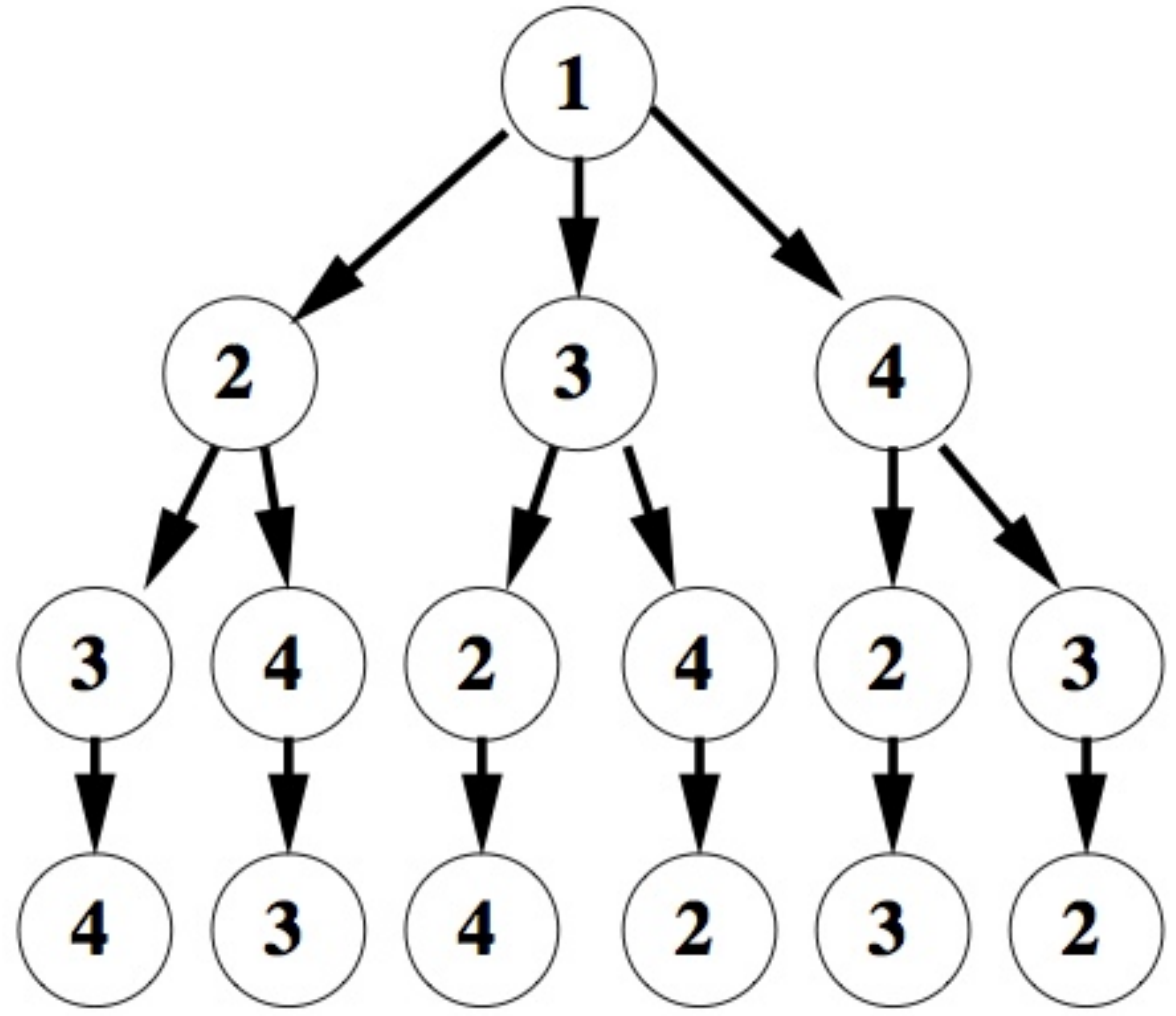
Assignment 3

- Interviewing undergrad problem
  - (a.k.a wandering salesman problem)
- Given: N cities + distances between cities
- Compute: shortest path starting at first city, that visits all cities
  - You get a job. So you don't return home.
  - Traveling salesman returns home.

Distances between cities 1-4



Enumeration of paths



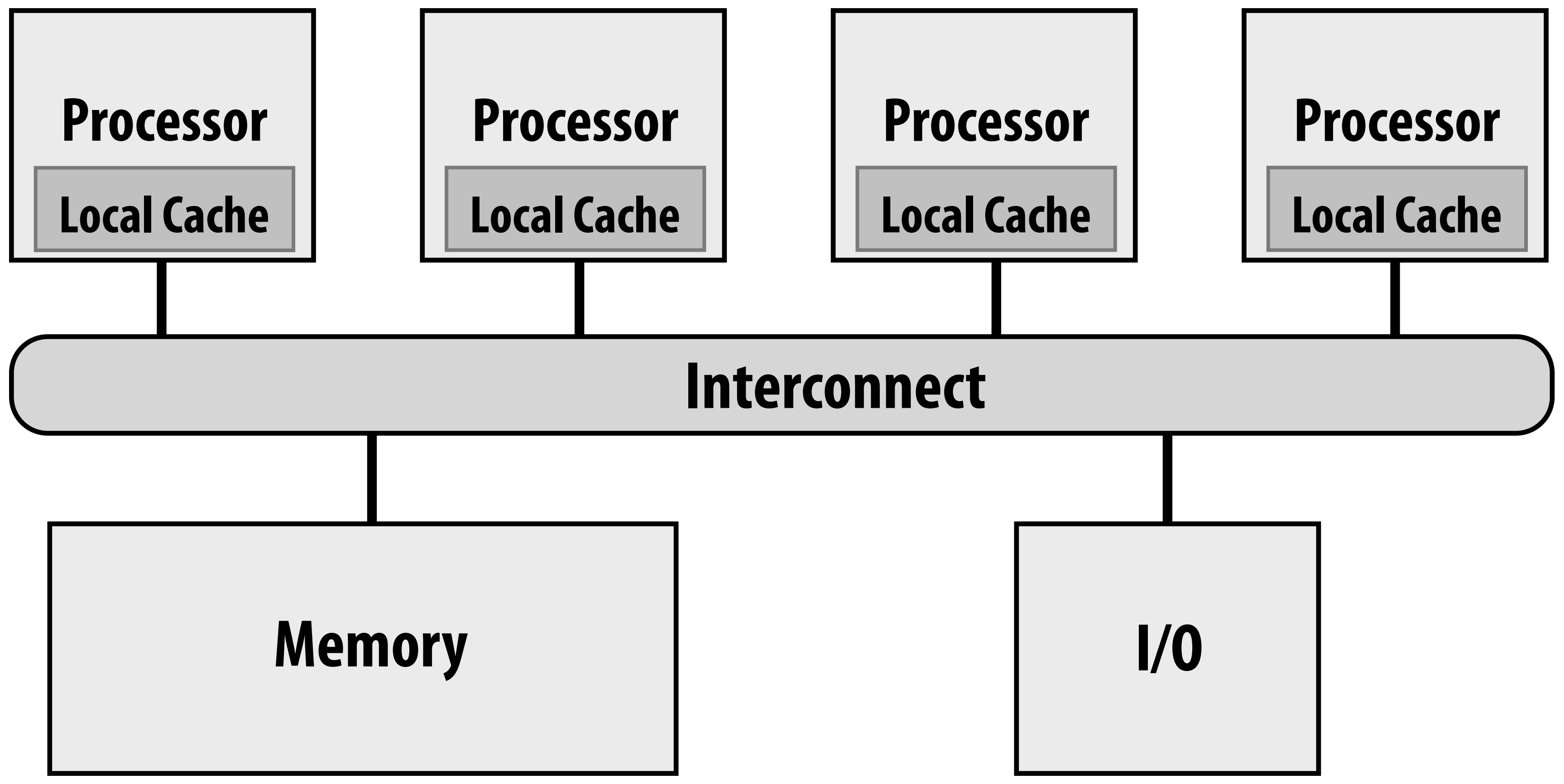
Today: what you should know

- What limits the scalability of snooping-based approaches to cache coherence
- How does a directory-based scheme avoid these problems?
- How can the storage overhead of the directory be reduced? (and at what cost?)

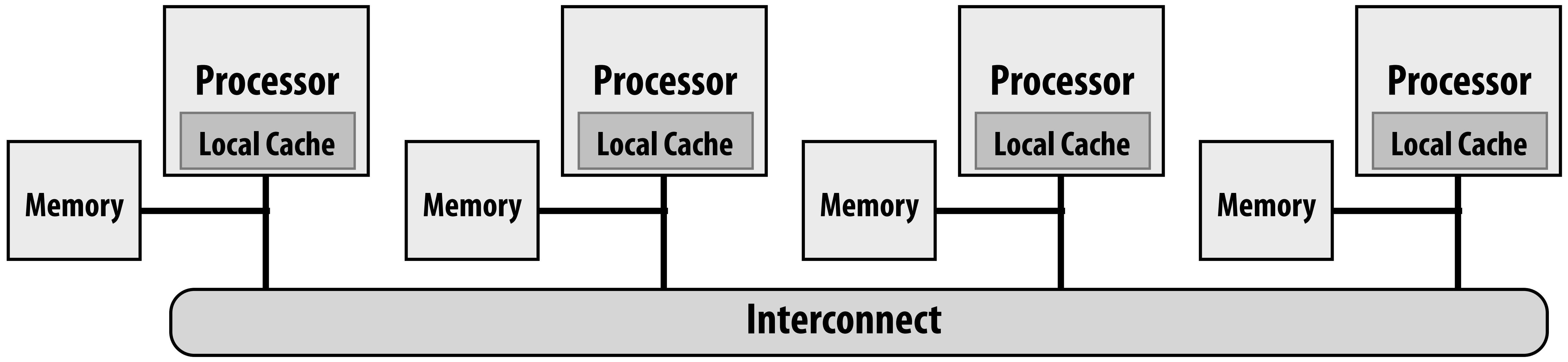
Implementing cache coherence

Last two lectures: snooping cache coherence implementations: relied on broadcast

Every time there is a cache miss, must communicate with all other caches!



Problem: scaling cache coherence to large machines



Recall non-uniform memory access (NUMA) shared memory systems

By distributing memories near the processors, can increase scalability: higher aggregate BW and reduced latency (especially when there is locality in the application)

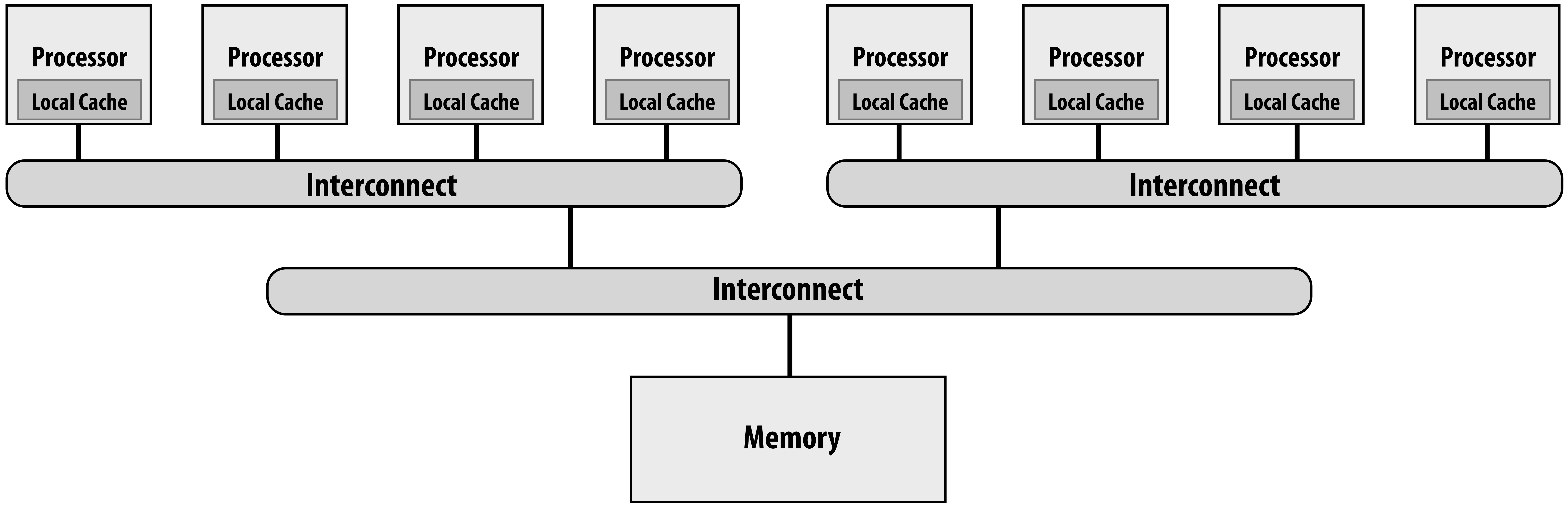
But... efficiency of NUMA system does little good if the coherence protocol can't also be scaled!

Some terminology:

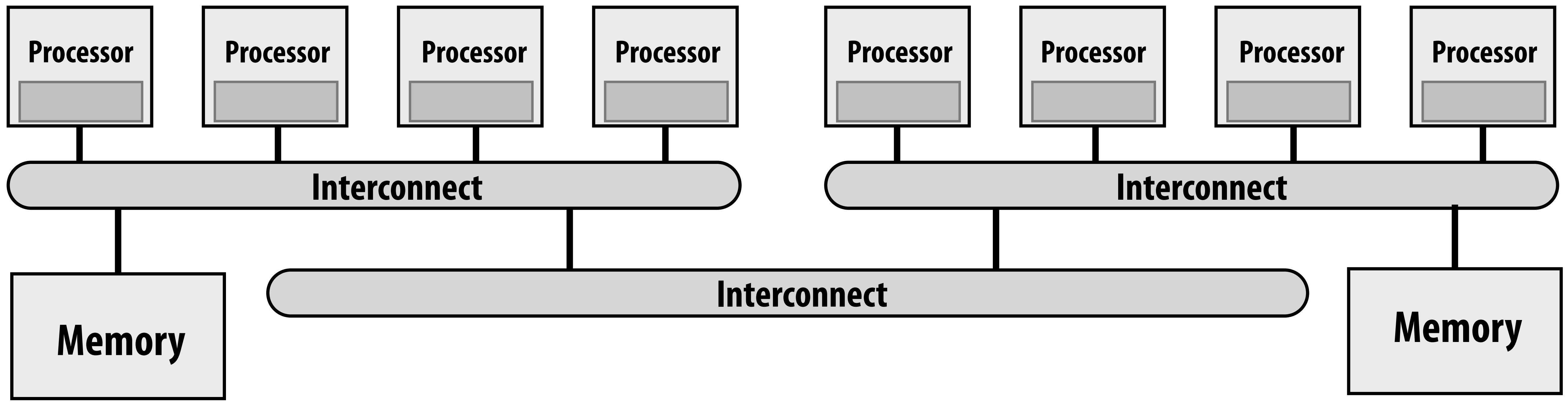
- **cc-NUMA = "cache-coherent, non-uniform memory access"**
- **Distributed shared memory system (DSM): cache coherent, shared address space architecture implemented by physically distributed memories**

One possible solution: hierarchy of snooping

Use snooping coherence at each level

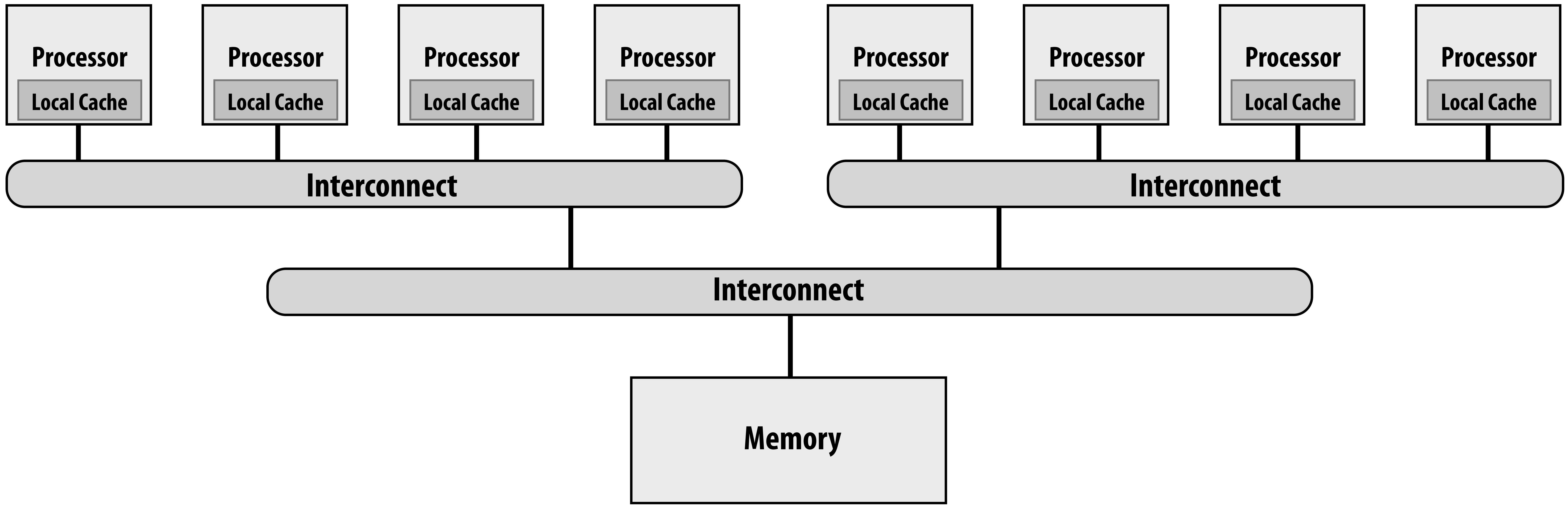


Another example: with memory localized with the groups of processors, rather than centralized



One possible solution: hierarchical snooping

Use snooping coherence at each level

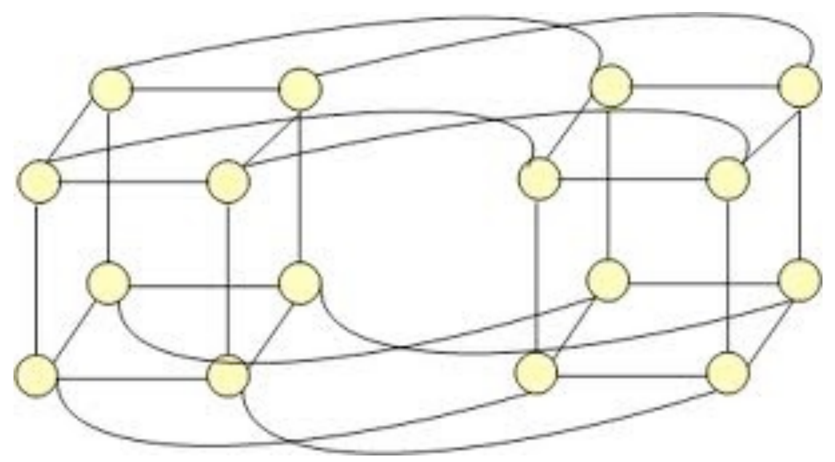


Advantages

- Relatively simple to build (already have to deal with similar issues due to multi-level caches)

Disadvantages

- The root becomes a bottleneck (low "bisection bandwidth")
- Larger latencies than direct networks
- Does not apply to more general network topologies (meshes, cubes)

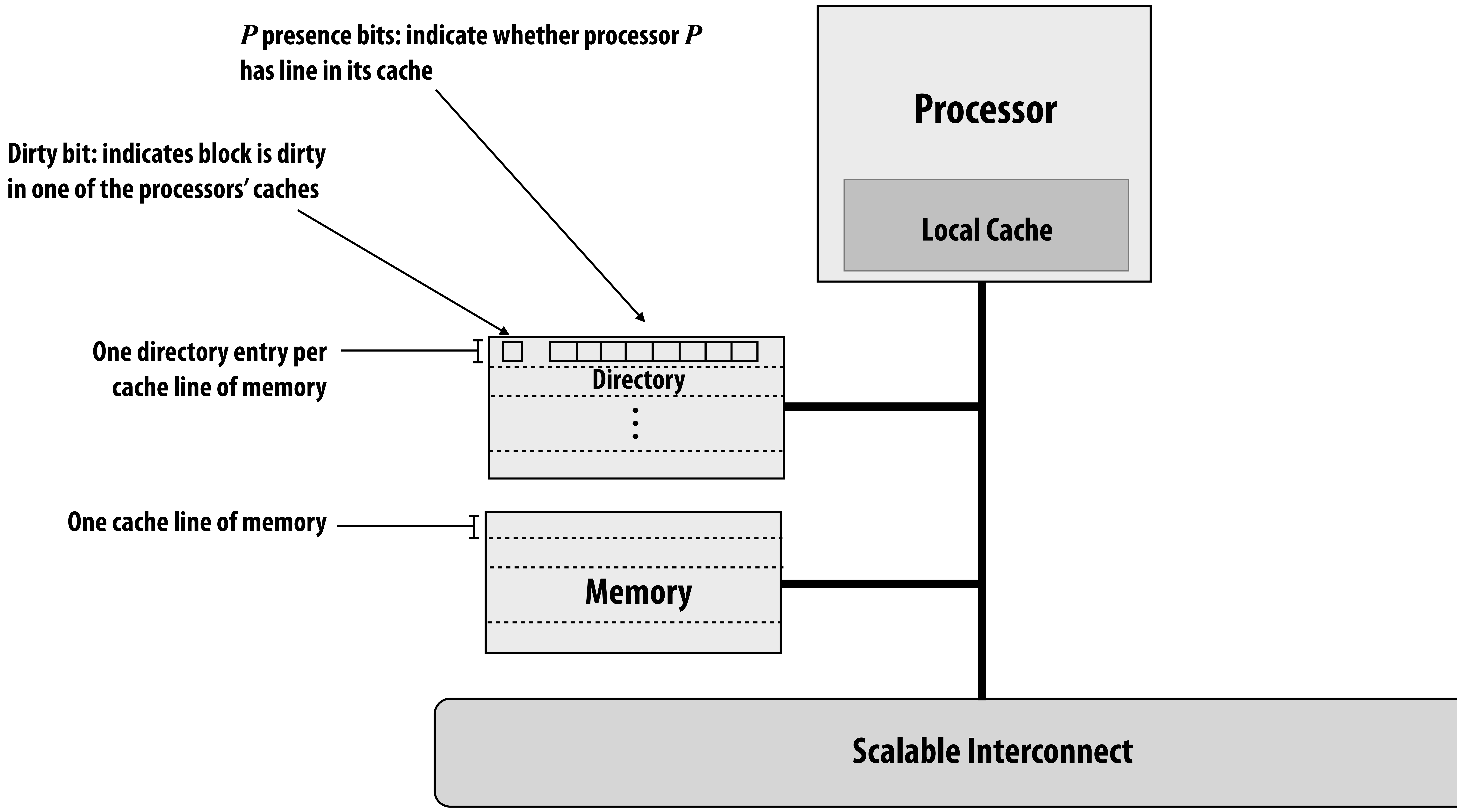


Scalable cache coherence using directories

- In a snooping scheme, the broadcast mechanism is used by caches to determine the state of a block in the other caches

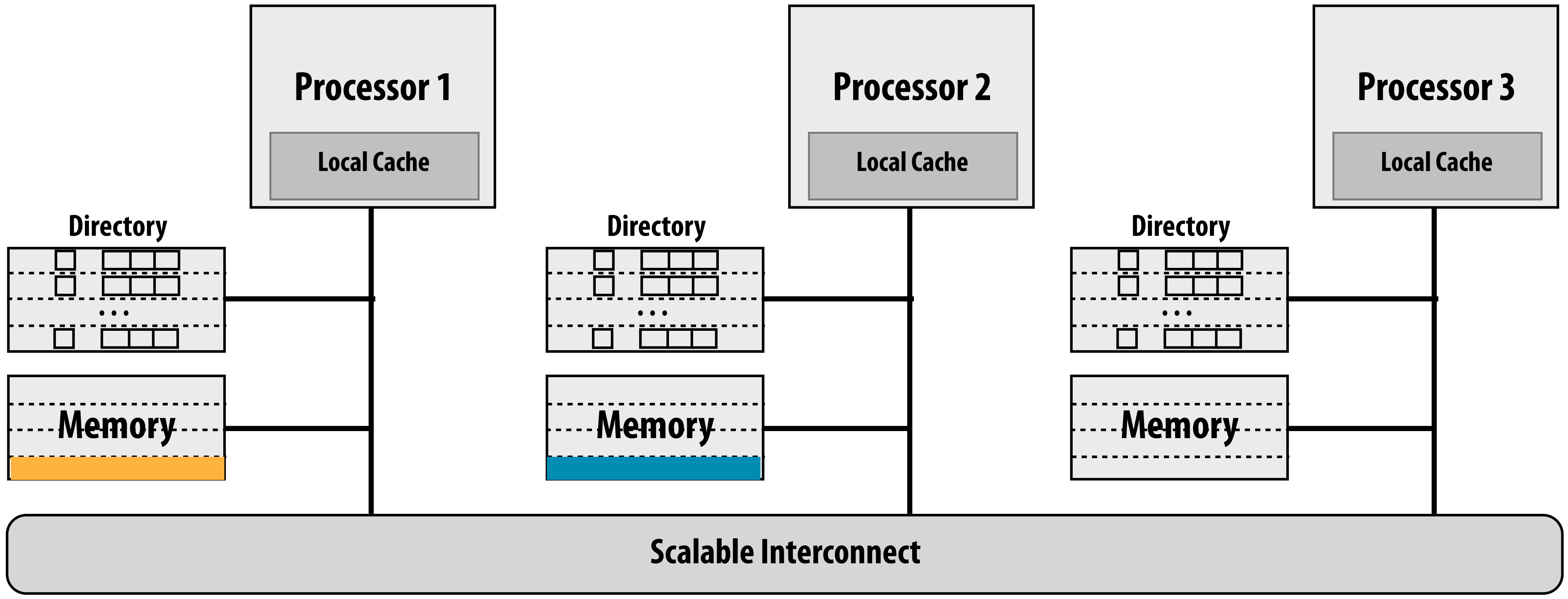
- Alternative idea: avoid broadcast by storing this information about the block in a "directory"
  - Caches look up information from the directory as necessary
  - Cache coherence is maintained by point-to-point messages between the caches (no reliable on broadcast mechanisms)

A very simple directory


P presence bits: indicate whether processor *P* has line in its cache
Dirty bit: indicates block is dirty in one of the processors' caches
One directory entry per cache line of memory
One cache line of memory
Directory
Memory
Processor
Local Cache
Scalable Interconnect

One cache line of memory

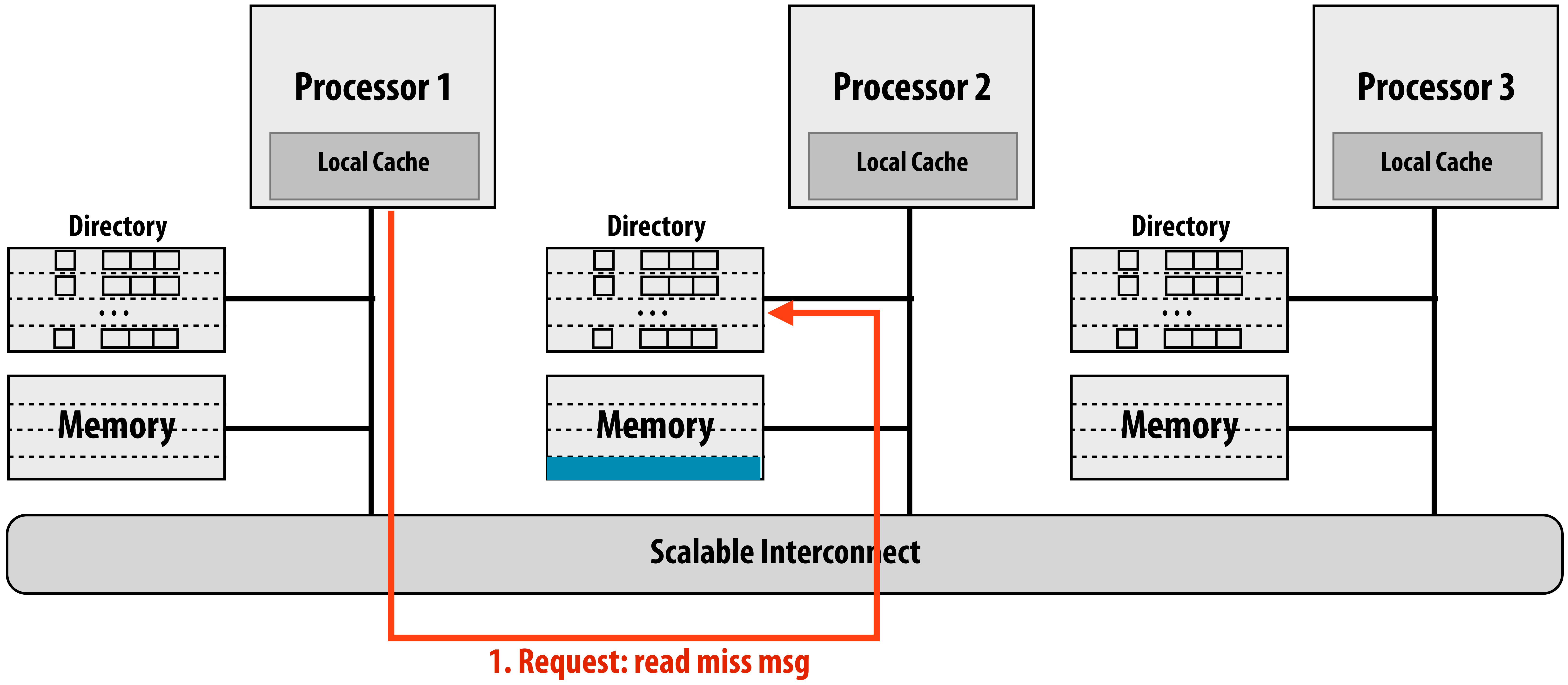
A directory for each node



- "Home node" of a block: node whose memory node is allocated in
  - Example: node 1 is the home node of the orange block, node 2 is the home node of the blue block
- "Requesting node": node containing processor requesting block

Example 1: read miss to clean block

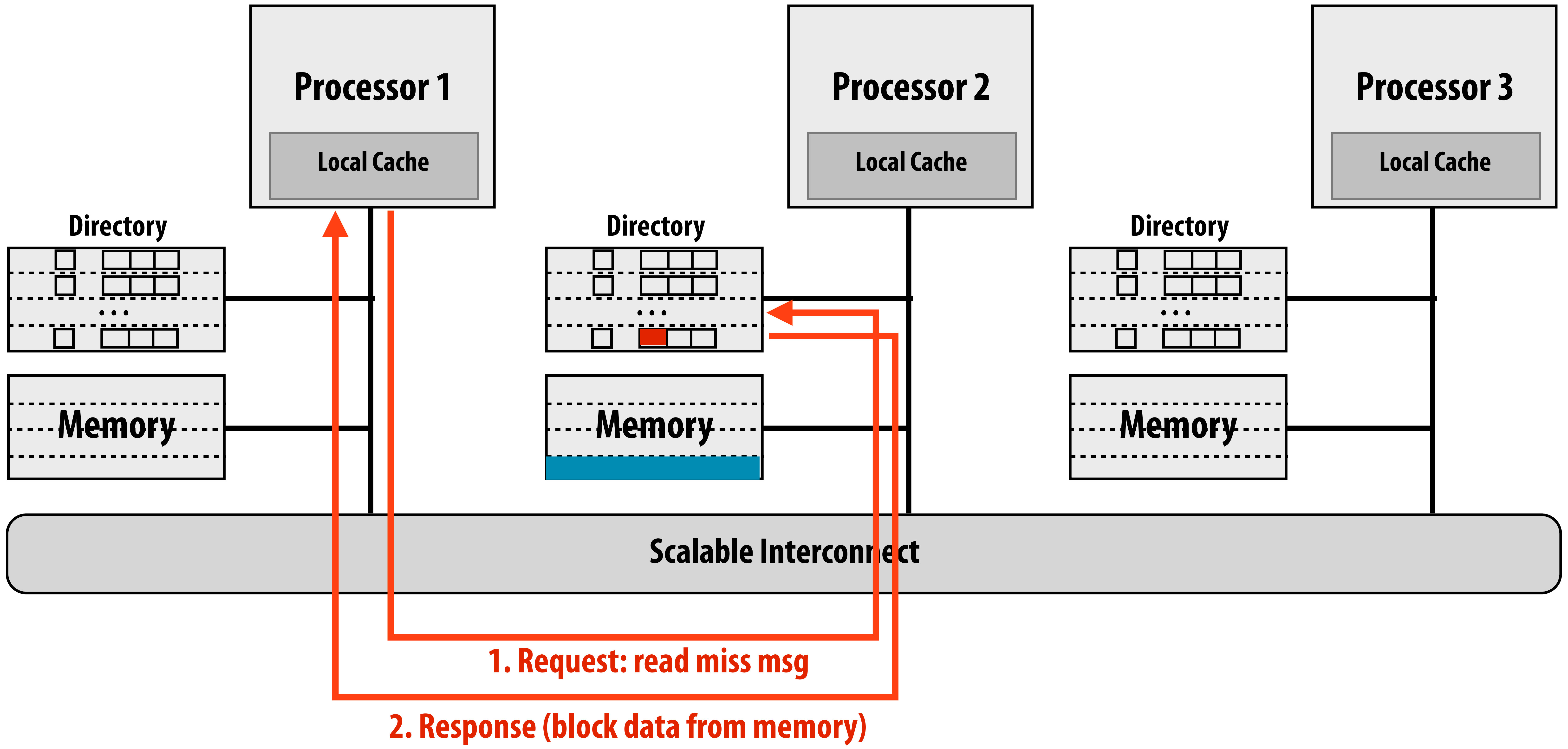
Read from main memory by processor 1 of the blue block: block is not dirty



- Read miss message sent to home node of the requested block
- Home directory checks entry for block

Example 1: read miss to clean block

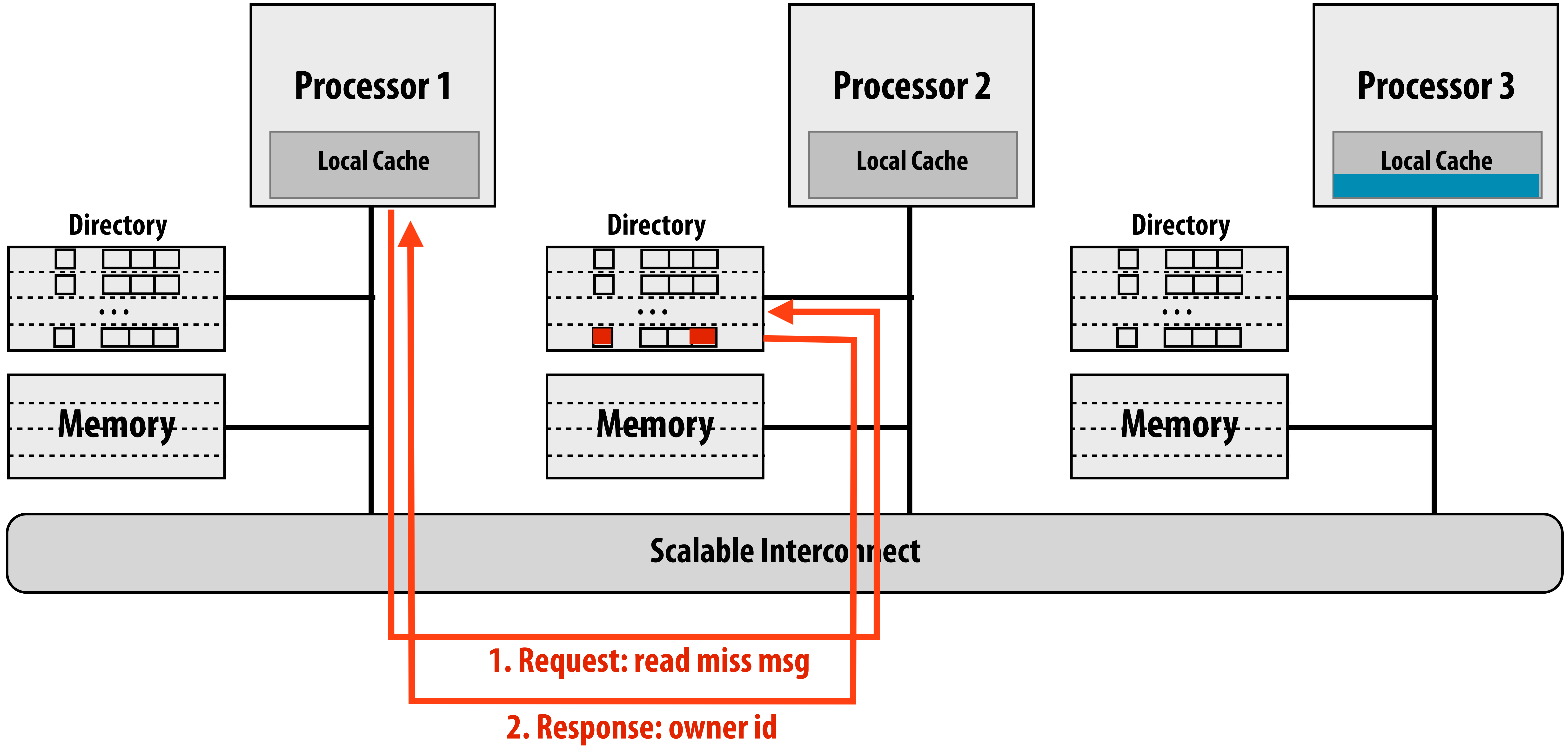
Read from main memory by processor 1 of the blue block: block is not dirty



- Read miss message sent to home node of the requested block
- Home directory checks entry for block
    - If dirty bit for block is OFF, respond with contents from memory, set presence[1] to true

Example 2: read miss to dirty block

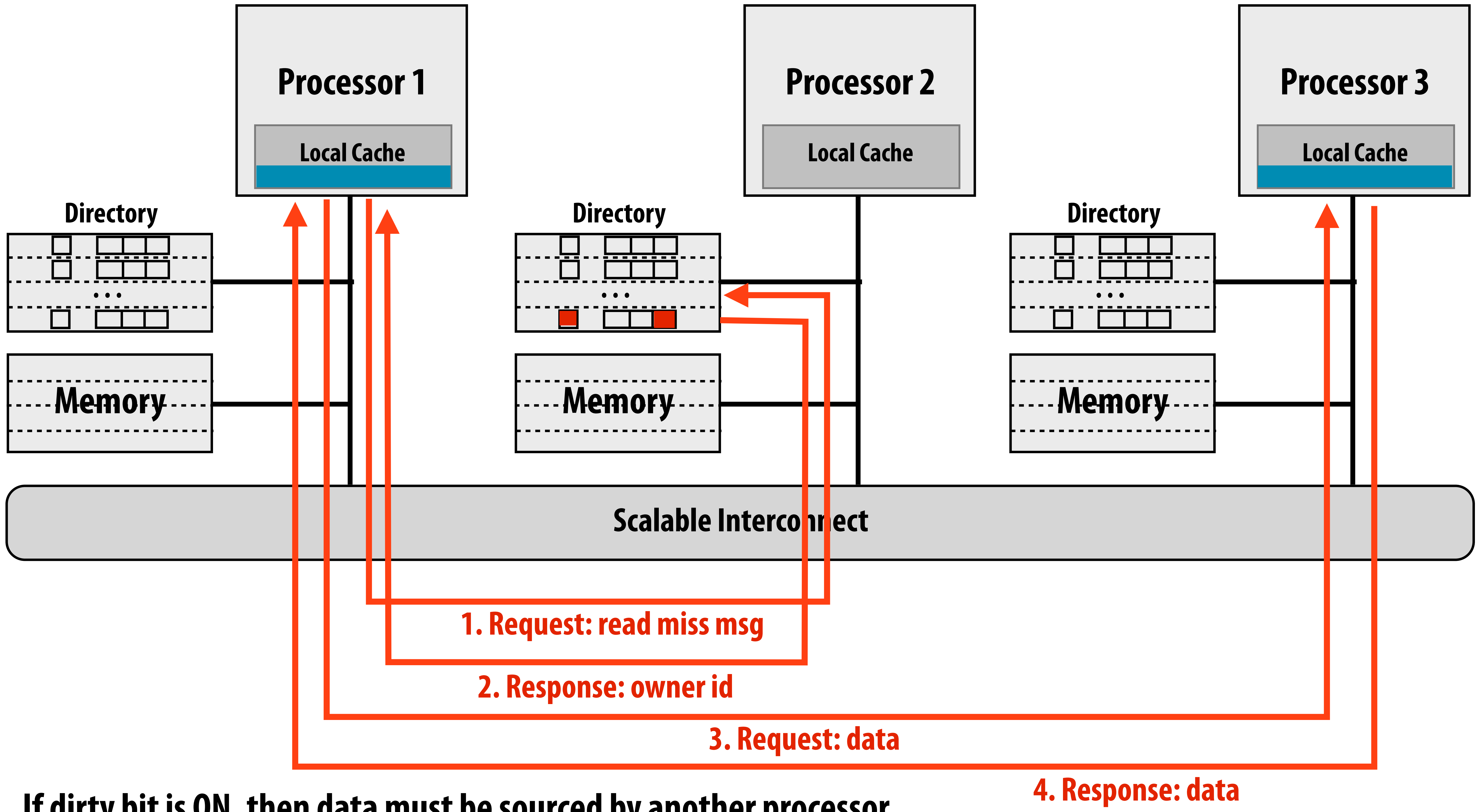
Read from main memory by processor 1 of the blue block: block is dirty (contents in P3's cache)



- If dirty bit is ON, then data must be sourced by another processor
- Home node must tell requesting node where to find data
  - Responds with message providing identity of block owner ("get it from P3")

Example 2: read miss to dirty block

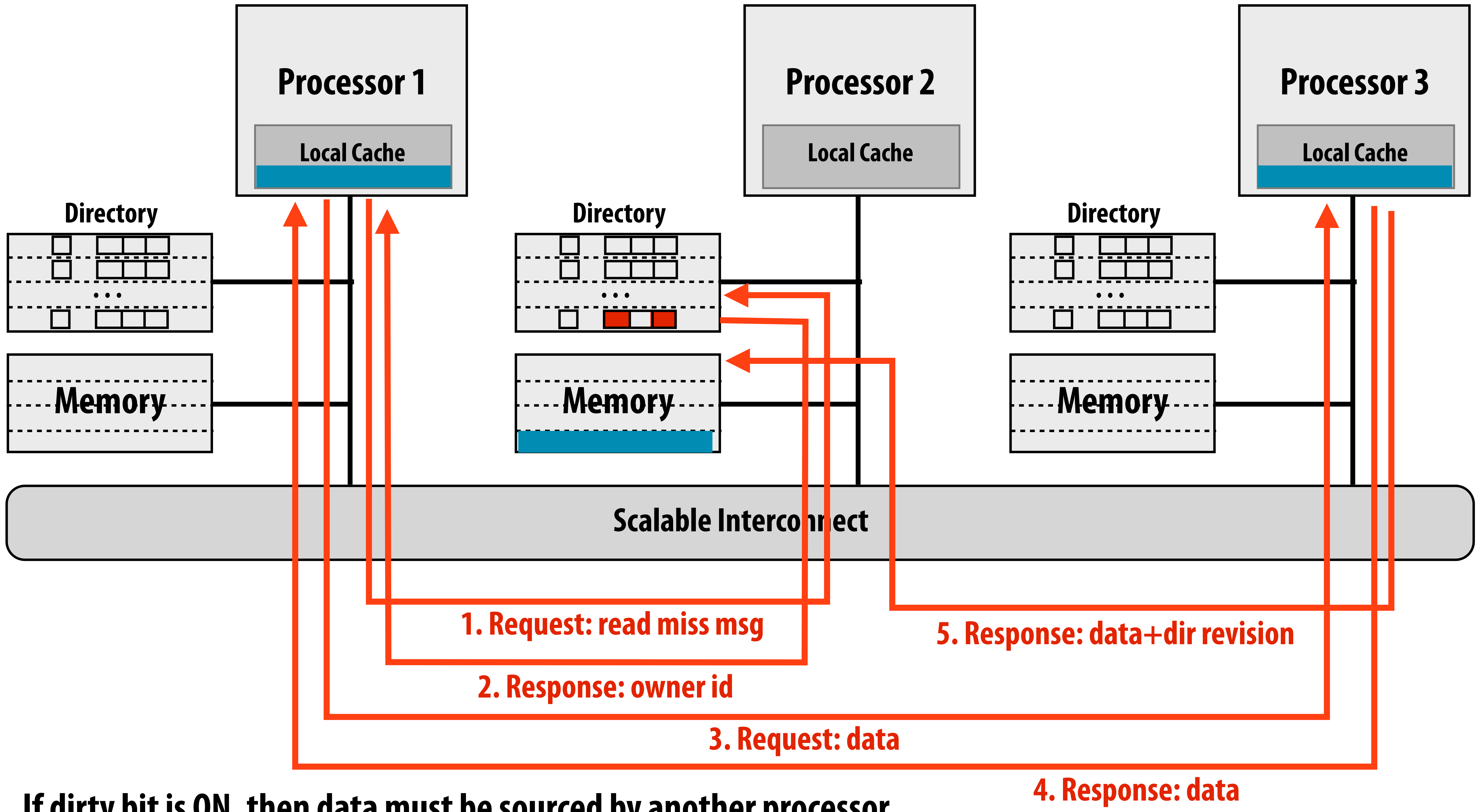
Read from main memory by processor 1 of the blue block: block is dirty (contents in P3's cache)



1. If dirty bit is ON, then data must be sourced by another processor
2. Home node responds with message providing identity of block owner
3. Requesting node requests data from owner
4. Owner responds to requesting node

Example 2: read miss to dirty block

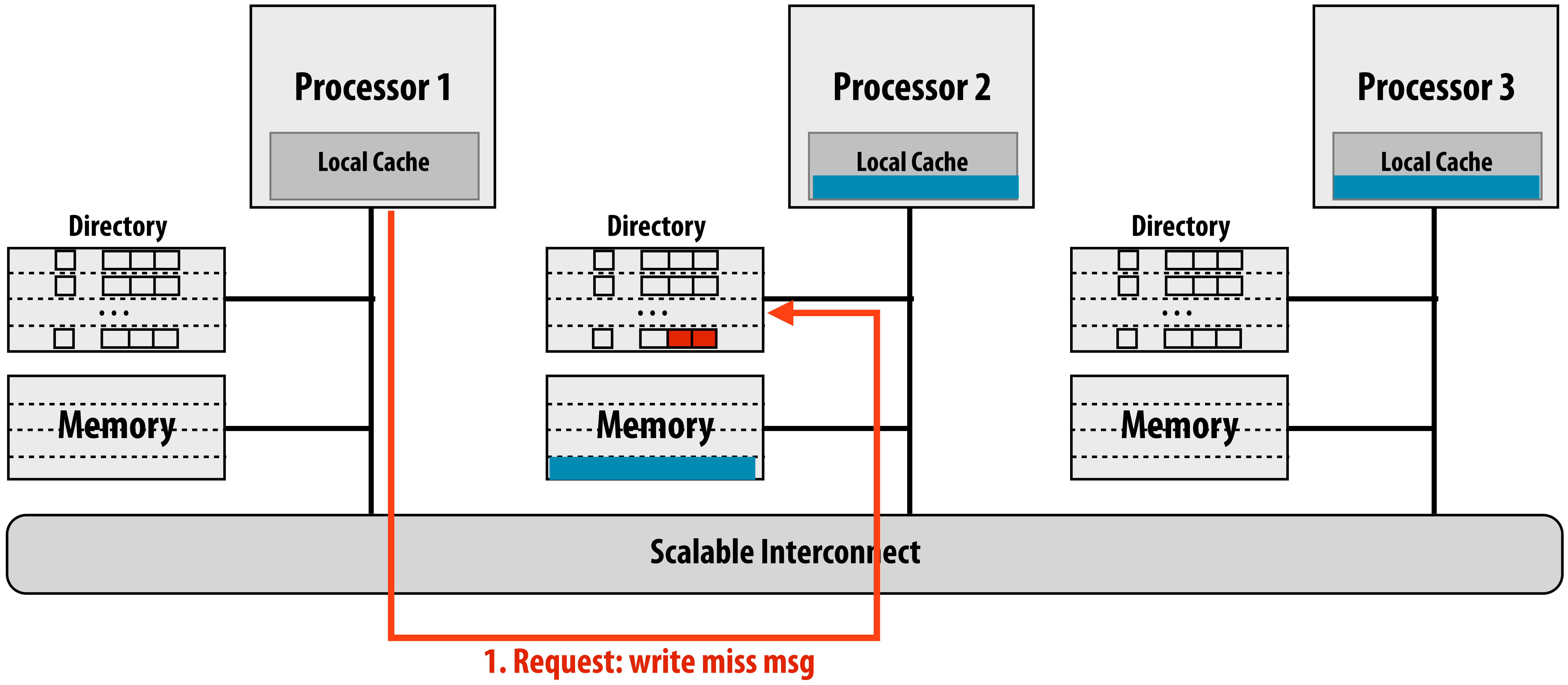
Read from main memory by processor 1 of the blue block: block is dirty (contents in P3's cache)



1. If dirty bit is ON, then data must be sourced by another processor
2. Home node responds with message providing identity of block owner
3. Requesting node requests data from owner
4. Owner responds to requesting node, changes state in cache to SHARED
5. Owner also responds to home node, home clears dirty and updates presence bits

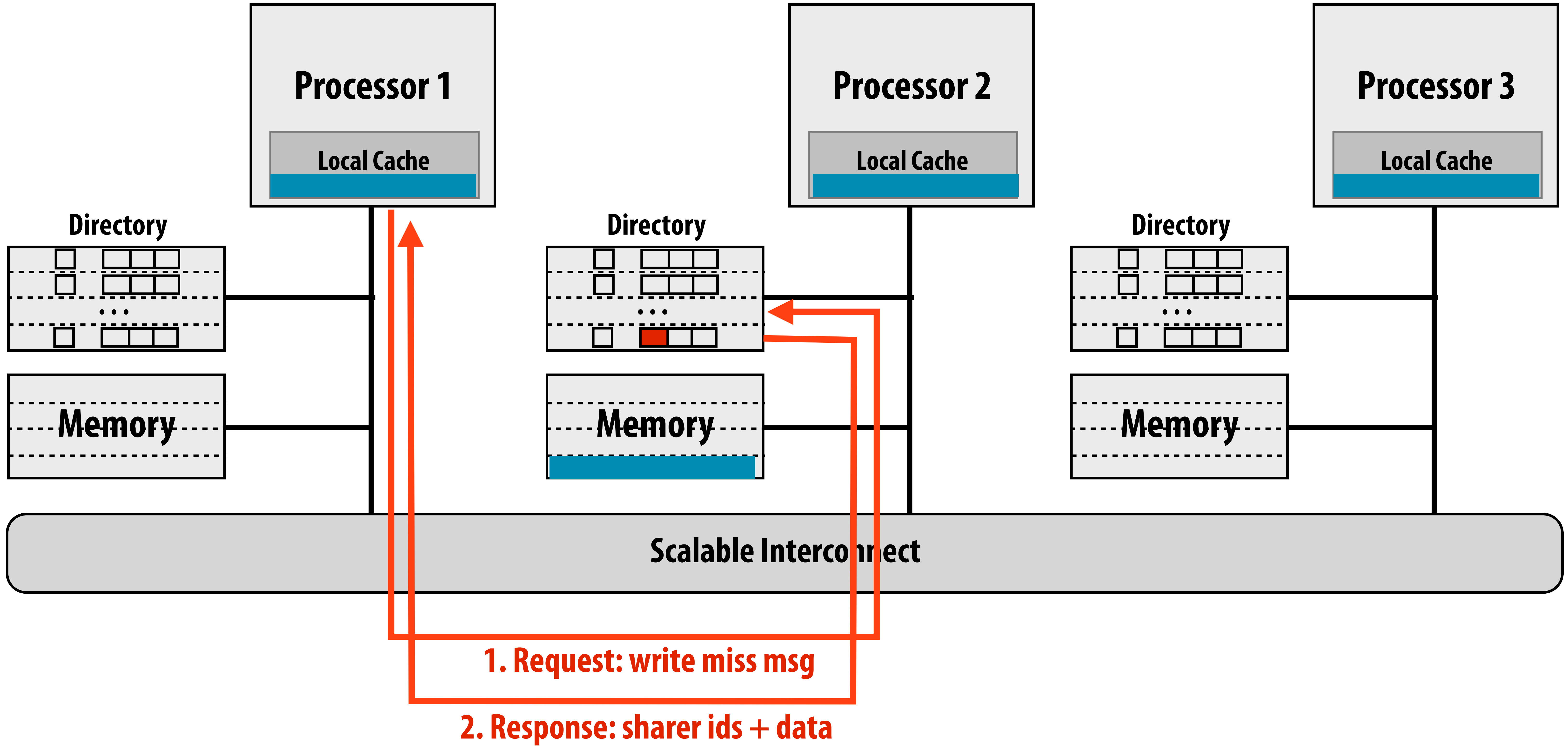
Example 3: write miss

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



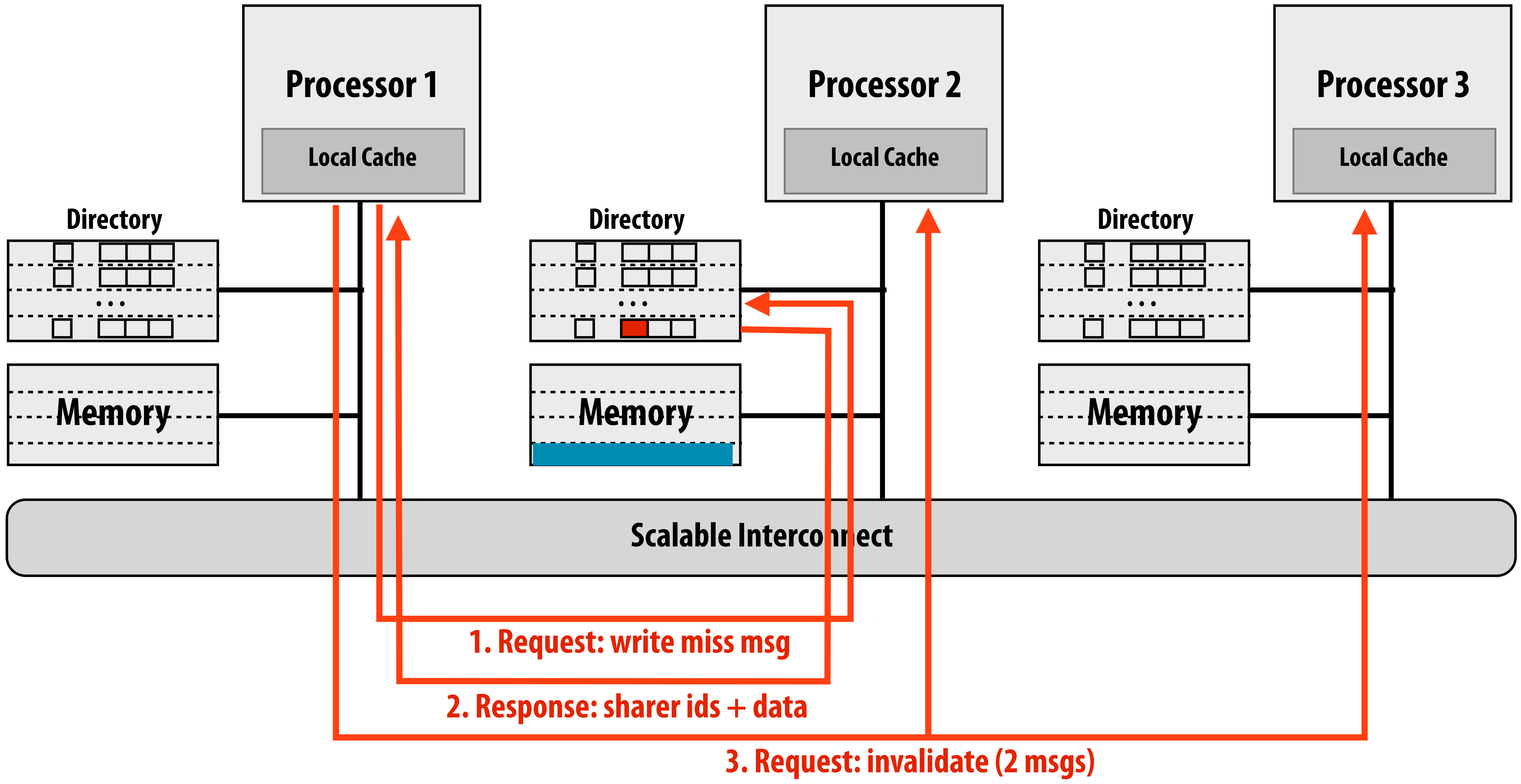
Example 3: write miss

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



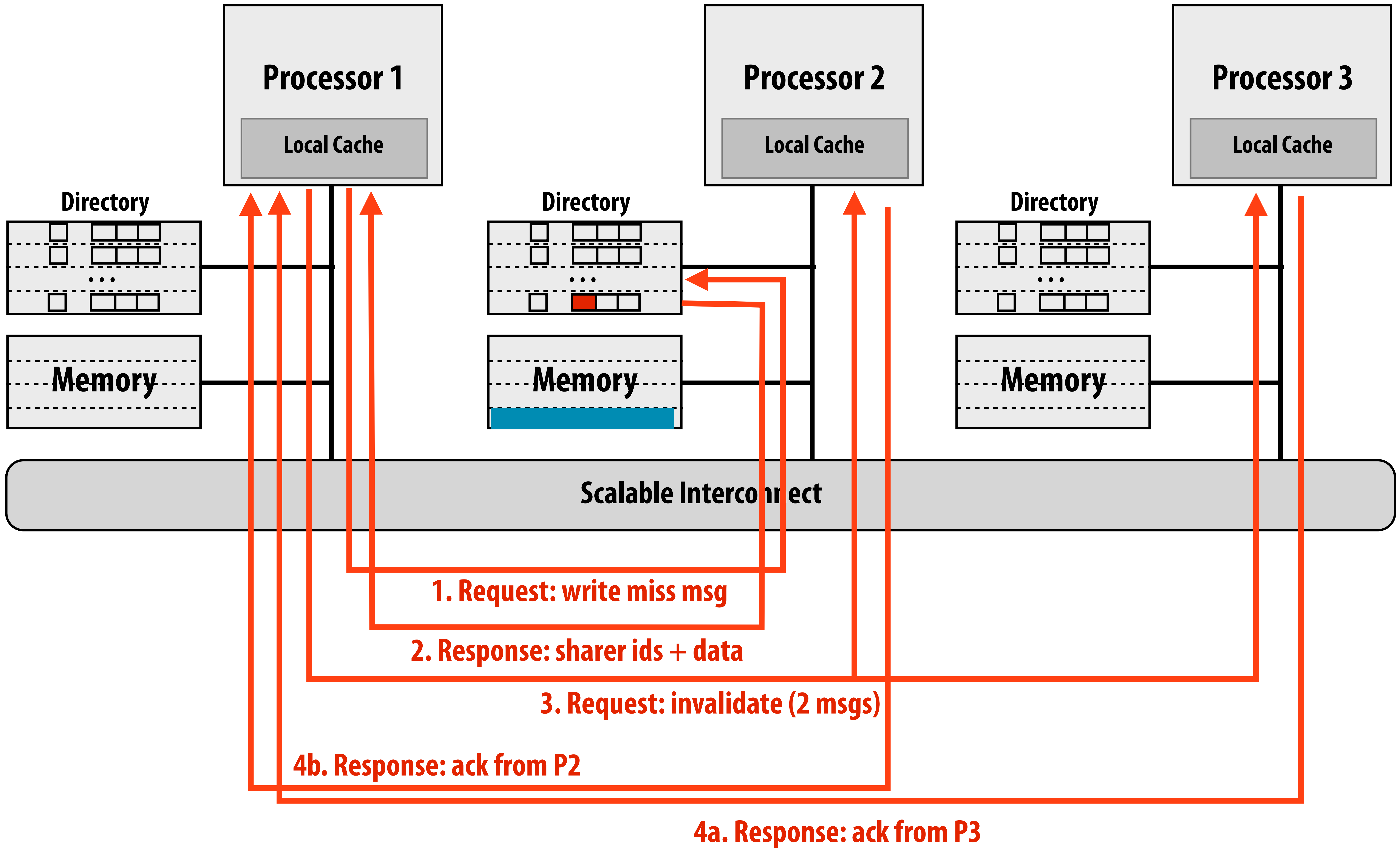
Example 3: write miss

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



Example 3: write miss

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


Processor 1
Local Cache
Processor 2
Local Cache
Processor 3
Local Cache
Directory
Directory
Directory
Memory
Memory
Memory
Scalable Interconnect
1. Request: write miss msg
2. Response: sharer ids + data
3. Request: invalidate (2 msgs)
4b. Response: ack from P2
4a. Response: ack from P3

After receiving both invalidation acks, P1 can perform write

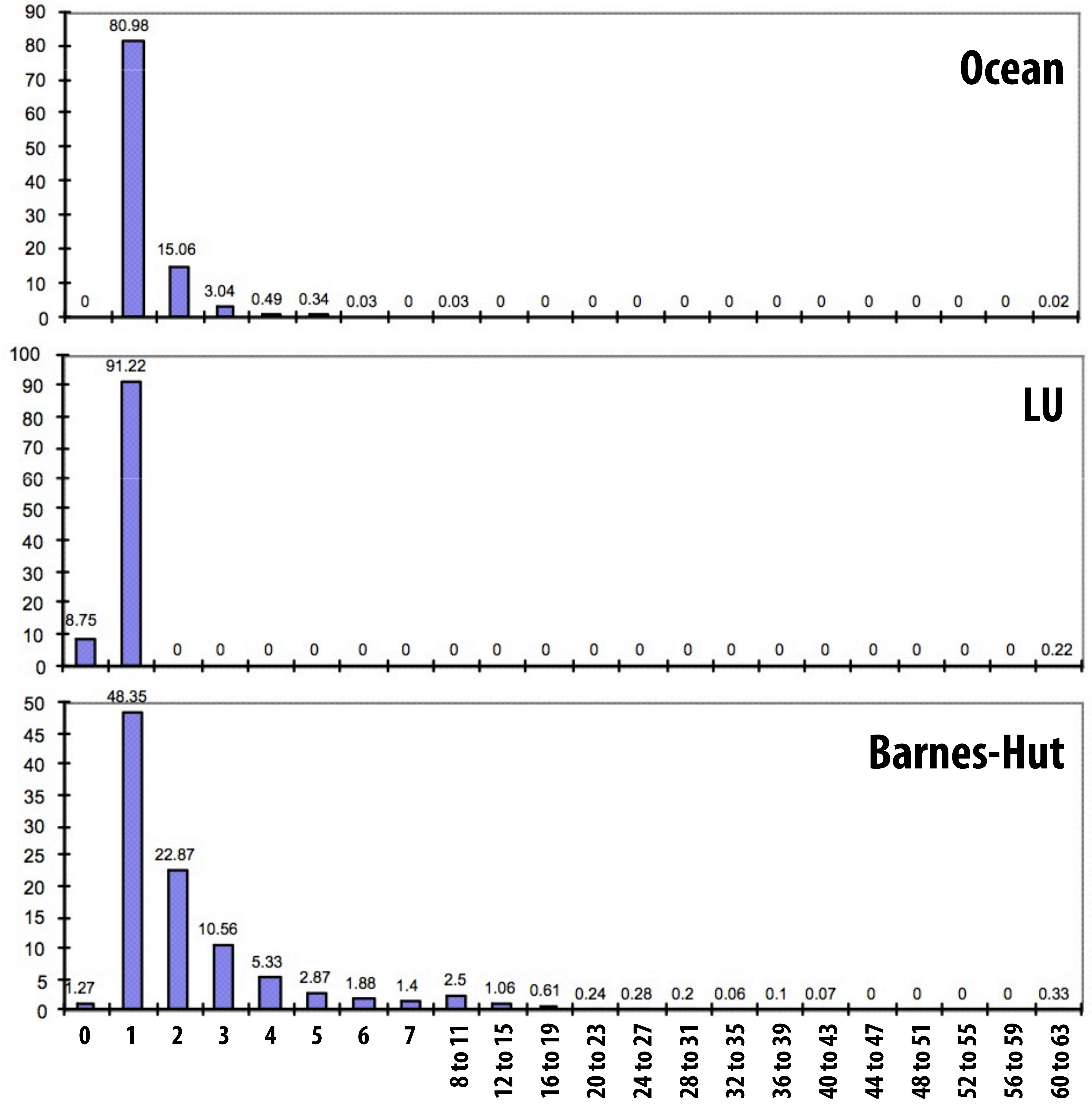
Advantage of directories

- **On reads, directory tells requesting node exactly where to get the block from**
  - Either from home node (if the block is clean)
  - Or from owning node (if the block is dirty)
  - Either way, it's point-to-point communication

- **On writes, the advantage of directories depends on the number of sharers**
  - In the limit, if all caches are sharing data, all caches must be communicated with (just like broadcast)

Cache invalidation patterns

64 processor system



In general, at the time of writes, only a few processors share the block

Also, the number of sharers increases slowly with P (good!)

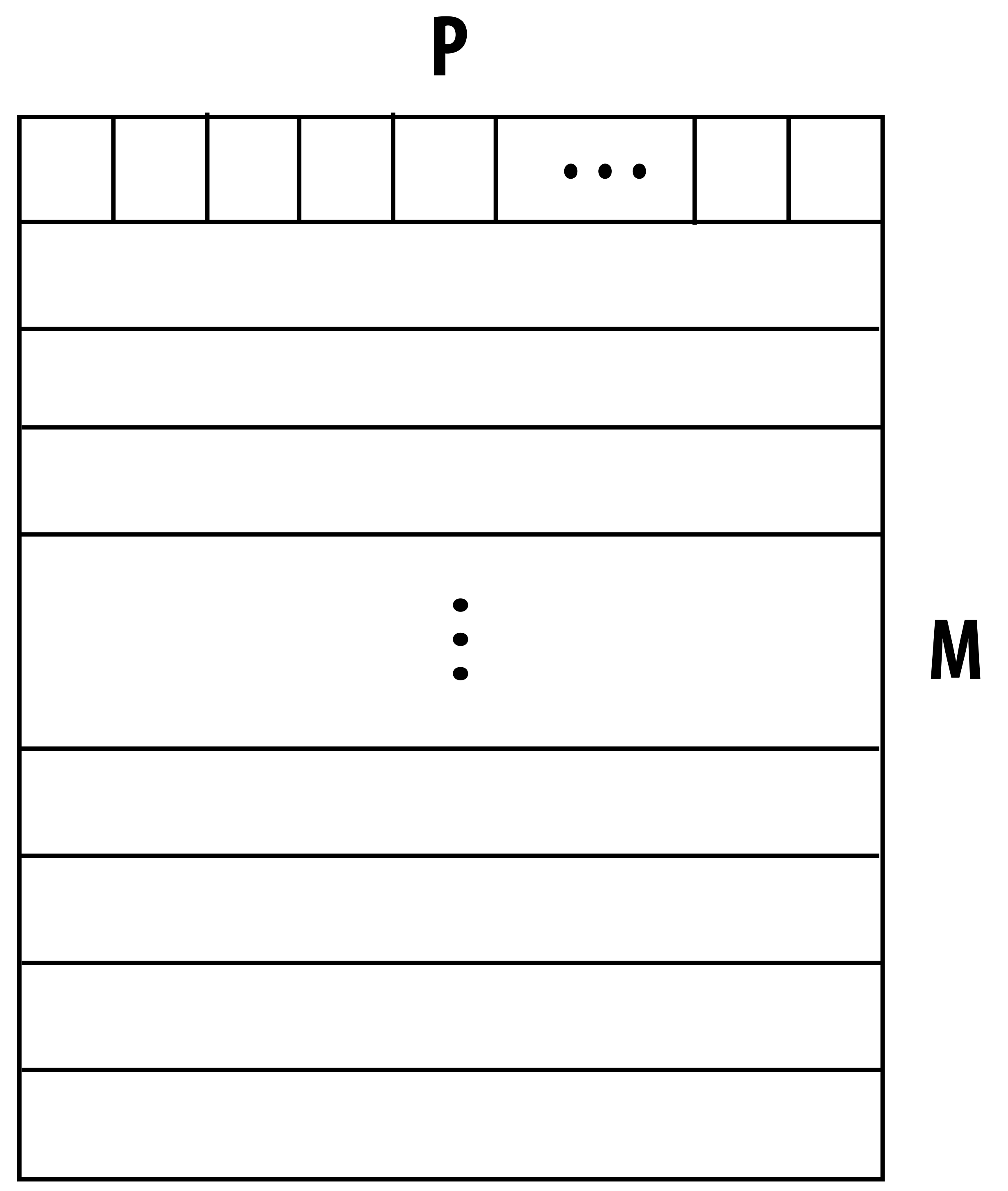
In general, only a few sharers during a write

- **Access patterns**
  - Mostly-read objects: lots of sharers but writes are infrequent, so minimal impact on performance (e.g., root node in Barnes-Hut)
  - Migratory objects: very few sharers, count does not scale with number of processors
  - Frequently read/written objects: frequent invalidations, but few of then because sharer count cannot build up between invalidations (e.g, shared task queue)
  - Low-contention locks: no problem, infrequent invalidations. (high-contention locks do present a challenge)

- **Implication 1: directories useful for limiting amount of traffic**
- **Implication 2: suggests ways to optimize directory implementation (reduce storage overhead)**

Full bit vector approach

- Recall: one presence bit per node
- Storage overhead proportion to P*M
  - P = number of nodes
  - M = number of blocks in memory
- Scales poorly with P
  - Assume 64 byte cache line size
  - 64 nodes → 12% overhead
  - 256 nodes → 50% overhead
  - 1024 nodes → 200% overhead



Reducing storage overhead of directory

- **Optimizations on full-bit vector scheme**
  - Increase cache block size (reduce M term)
    - What are possible problems with this approach? (consider graphs from last lecture)
  - Place multiple processors in a “node” (reduce P term)
    - Need one directory bit per node, not bit per processor
    - Hierarchical: use snooping protocol amongst processors in a node

- **Alternative schemes**
  - Limited pointer schemes (reduce P)
  - Sparse directories (reduce M)

Limited pointer schemes

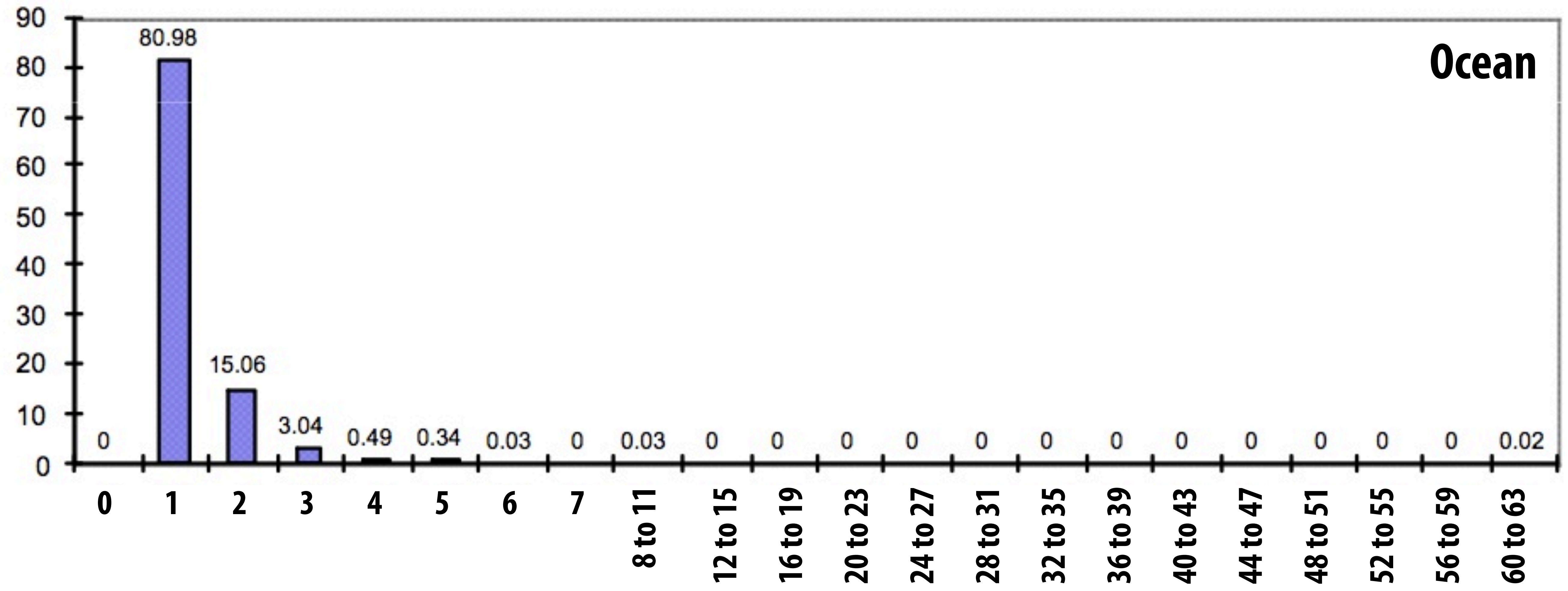
Since data is expected to only be in a few caches at once, a limited number of pointers per directory entry should be sufficient (don't need information about all nodes)

Example: 1024 processor system

Full bit vector scheme needs 1024 bits per block

Instead, can store 10 pointers to nodes holding the block ($\log_2(1024)$=10 bits per pointer)

In practice, our workload evaluation says we can get by with less than 10



Managing overflow in limited pointer schemes

Many possible approaches

- **Broadcast**
  - When more than max number of sharers, revert to broadcast
- **No broadcast**
  - Do not allow more than a max number of sharers
  - On overflow, newest sharer replaces an existing one (invalidate the old sharer)
- **Coarse vector**
  - Change representation so that each bit corresponds to K nodes
  - On write, invalidate all nodes a bit belongs to
- **Dynamic pointers**
  - Hardware maintains a pool of pointers (free list)
  - Manages allocation of pointers to directory blocks

Aside:

Optimizing for the common case

Limited pointer schemes are a great example of understanding and optimizing for the common case:

1. **Workload driven observation: in general the number of sharers is low**
2. **Make the common case simple and fast: array of pointers for first N sharers**
3. **Uncommon case is still handled correctly, just with a slower, more complicated mechanism (the program still works!)**
4. **Extra expense is tolerable, since it happens infrequently**

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

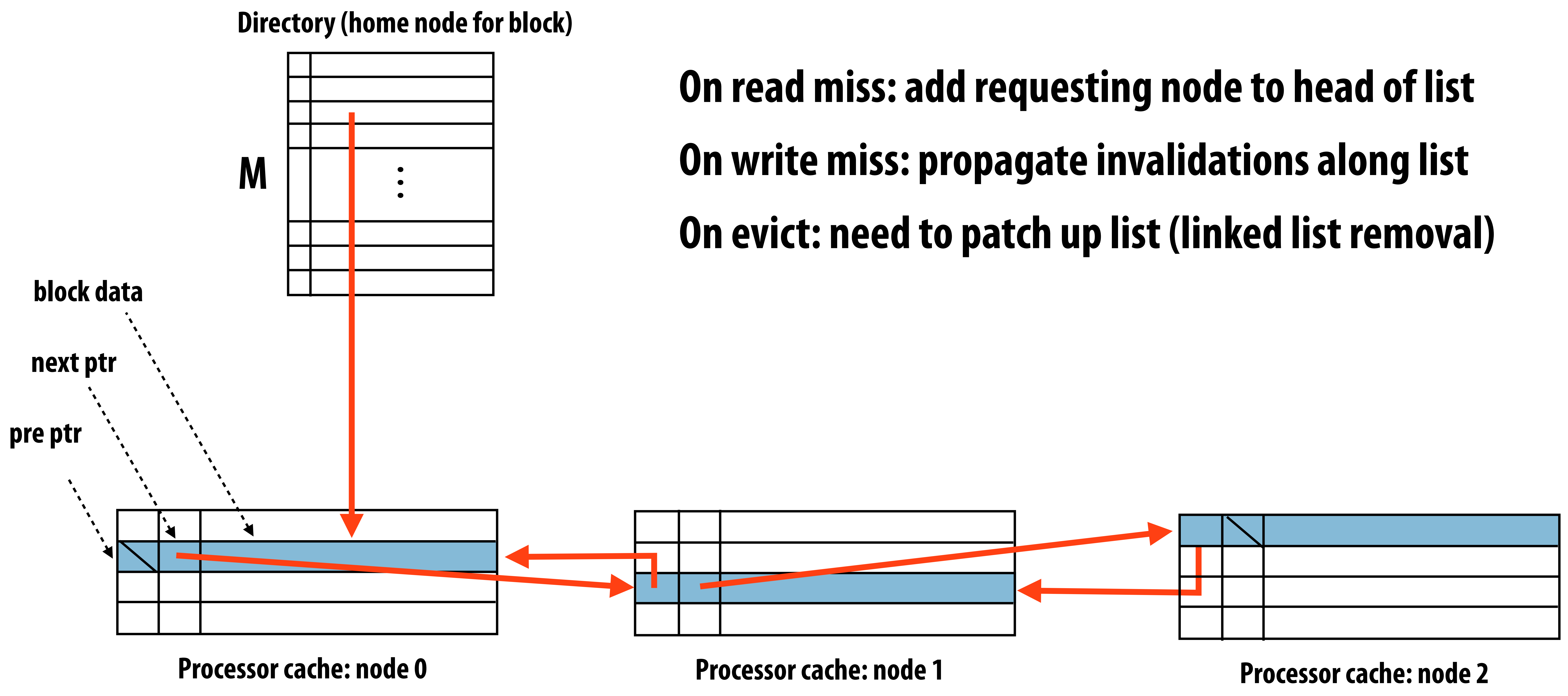
Directory at home node maintains only pointer to one node caching block.

Pointer to next node stored in the cache line

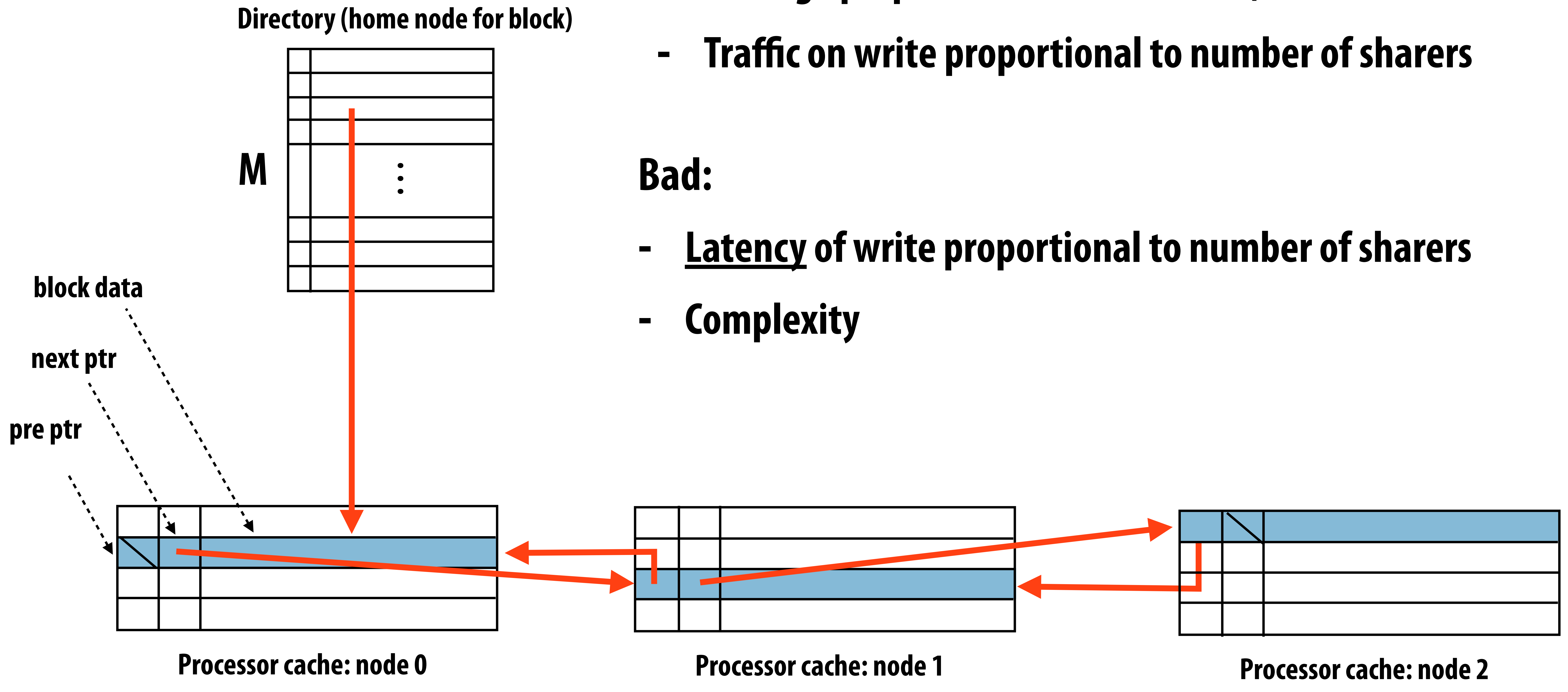
On read miss: add requesting node to head of list

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
- Complexity

Summary: directories

- **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 look it up**

- **One challenge: reducing overhead of directory storage**
  - **limited pointer schemes: exploit fact the most processors not sharing**
  - **sparse directory schemes: exploit fact that most blocks are not in cache**