Memory and cache Consistency

Unit-V

Syed Rameem Zahra
(Assistant Professor)
Department of CSE, NSUT

Consistency Model

- A consistency model is contract between a distributed data store and processes, in which the processes agree to obey certain rules in contrast the store promises to work correctly.
- A consistency model basically refers to the degree of consistency that should be maintained for the shared memory data.
- If a system supports the stronger consistency model, then the weaker consistency model is automatically supported, but the converse is not true.
- The types of consistency models are **Data-Centric** and client centric consistency models.

Why Consistency Models Matter

- Each thread accesses two types of memory locations
  - Private: only read/written by that thread – should conform to sequential semantics
    - “Read A” should return the result of the last “Write A” in program order
  - Shared: accessed by more than one thread – what about these?
- Answer is determined by the **Memory Consistency Model** of the system
- Determines the order in which shared-memory accesses from different threads can “appear” to execute
  - In other words, determines what value(s) a read can return
  - More precisely, the set of all writes (from all threads) whose value can be returned by a read

Difference between Cache Coherence and Memory Consistency

Cache coherence	Memory consistency
Cache Coherence describes the behavior of reads and writes to the same memory location.	Memory consistency describes the behavior of reads and writes in relation to other locations.
Cache coherence required for cache-equipped systems.	Memory consistency required in systems that have or do not have caches.
Coherence is the guarantee that caches will never affect the observable functionality of a program	Consistency is the specification of correctness for memory accesses,
It is concerned with the ordering of writes to a single memory location.	It handles the ordering of reads and writes to all memory locations
A memory system is coherent if and only if – Can serialize all operations to that location – Read returns the value written to that location by the last store.	Consistency is a feature of a memory system if – It adheres to the rules of its Memory Model. – Memory operations are performed in a specific order.

Types of Consistency

- **Strict Consistency Model:** "The strict consistency model is the strongest form of memory coherence, having the most stringent consistency requirements. A shared-memory system is said to support the strict consistency model if the value returned by a read operation on a memory address is always the same as the value written by the most recent write operation to that address, irrespective of the locations of the processes performing the read and write operations. That is, all writes instantaneously become visible to all processes."
- **Sequential Consistency Model:** "The sequential consistency model was proposed by **Lamport**. A shared-memory system is said to support the sequential consistency model if all processes see the same order of all memory access operations on the shared memory. The exact order in which the memory access operations are interleaved does not matter. If one process sees one of the orderings of three operations and another process sees a different one, the memory is not a sequentially consistent memory."
    - **Example:** Assume three operations read(R1), write(W1), read(R2) performed in an order on a memory address. Then (R1,W1,R2),(R1,R2,W1),(W1,R1,R2)(R2,W1,R1) are acceptable provided all processes see the same ordering.
- **Causal Consistency Model:** "The causal consistency model relaxes the requirement of the sequential model for better concurrency. Unlike the sequential consistency model, in the causal consistency model, all processes see only those memory reference operations in the same (correct) order that are potentially causally related. Memory reference operations that are not potentially causally related may be seen by different processes in different orders."
    - If a write(w2) operation is causally related to another write (w1) the acceptable order is (w1, w2).

Types of Consistency

- **FIFO Consistency Model:** For FIFO consistency, Writes done by a single process are seen by all other processes in the order in which they were issued, but writes from different processes may be seen in a different order by different processes.
  - **Example**: If (w11) and (w12) are write operations performed by p1 in that order and (w21),(w22) by p2. A process p3 can see them as [(w11,w12),(w21,w22)] while p4 can view them as [(w21,w22),(w11,w12)].
- **Pipelined Random-Access Memory (PRAM) Consistency Model:** The pipelined random-access memory (PRAM) consistency model provides a weaker consistency semantics than the (first three) consistency models described so far. It only ensures that all write operations performed by a single process are seen by all other processes in the order in which they were performed as if all the write operations performed by a single process are in a pipeline. Write operations performed by different processes may be seen by different processes in different orders.
- **Weak Consistency Model:** Synchronization accesses (accesses required to perform synchronization operations) are sequentially consistent. Before a synchronization access can be performed, all previous regular data accesses must be completed. Before a regular data access can be performed, all previous synchronization accesses must be completed. This essentially leaves the problem of consistency up to the programmer. The memory will only be consistent immediately after a synchronization operation.

Types of Consistency

- **Release Consistency Model:** Release consistency is essentially the same as weak consistency, but synchronization accesses must only be processor consistent with respect to each other. Synchronization operations are broken down into acquire and release operations. All pending acquires (e.g., a lock operation) must be done before a release (e.g., an unlock operation) is done. Local dependencies within the same processor must still be respected. "Release consistency is a further relaxation of weak consistency without a significant loss of coherence.
- **Entry Consistency Model:** variants of release consistency, it requires the programmer (or compiler) to use acquire and release at the start and end of each critical section, respectively. However, unlike release consistency, entry consistency requires each ordinary shared data item to be associated with some synchronization variable, such as a lock or barrier. If it is desired that elements of an array be accessed independently in parallel, then different array elements must be associated with different locks. When an acquire is done on a synchronization variable, only those data guarded by that synchronization variable are made consistent.
- **Processor Consistency Model:** Writes issued by a processor are observed in the same order in which they were issued. However, the order in which writes from two processors occur, as observed by themselves or a third processor, need not be identical. That is, two simultaneous reads of the same location from different processors may yield different results.
- **General Consistency Model:** A system supports general consistency if all the copies of a memory location eventually contain the same data when all the writes issued by every processor have completed.

What Does Coherence Mean?

- Informally: – Any read must return the most recent write
  - Too strict and very difficult to implement
- Better: – Any write must eventually be seen by a read
  - All writes are seen in proper order (“serialization”)
- Two rules to ensure this:
  - If P writes x and P1 reads it, P’s write will be seen by P1 if the read and write are sufficiently far apart
  - Writes to a single location are serialized: seen in one order
    - Latest write will be seen
    - Otherwise could see writes in illogical order (could see older value after a newer value)

Potential Solutions Cache Coherence

- **Snooping Solution (Snoopy Bus):**
  - Send all requests for data to all processors
  - Processors snoop to see if they have a copy and respond accordingly – Requires broadcast, since caching information is at processors
  - Works well with bus (natural broadcast medium) – Dominates for small scale machines (most of the market)
- **Directory-Based Schemes:**
  - Keep track of what is being shared in one centralized place
  - Distributed memory => distributed directory (avoids bottlenecks) – Send point-to-point requests to processors
  - Scales better than Snoop
  - Actually existed BEFORE Snoop-based schemes

Basic Snoopy Protocols

- Write Invalidate Protocol:
  - Multiple readers, single writer
  - Write to shared data: an invalidate is sent to all caches which snoop and invalidate any copies
  - Read Miss:
    - Write-through: memory is always up-to-date
    - Write-back: snoop in caches to find most recent copy
- Write Broadcast Protocol:
  - Write to shared data: broadcast on bus, processors snoop, and update copies
  - Read miss: memory is always up-to-date

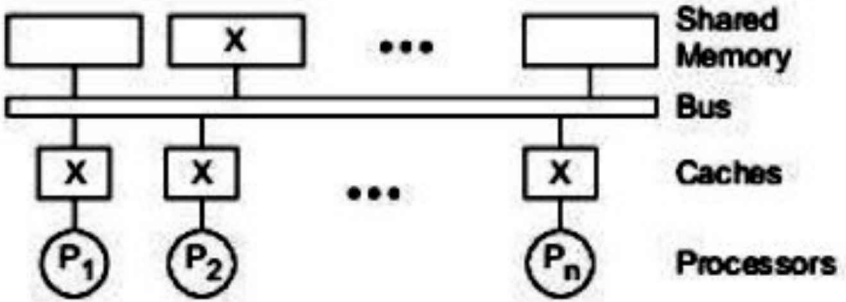
Basic Snoopy Protocols

- Invalidation protocol, write-back cache
- Each block of memory is in one state:
  - Clean in all caches and up-to-date in memory (Shared)
  - OR Dirty in exactly one cache (Exclusive)
  - OR Not in any caches
- Each cache block is in one state:
  - Shared : block can be read
  - OR Exclusive : cache has only copy, its writeable, and dirty
  - OR Invalid : block contains no data
- Read misses: cause all caches to snoop bus
- Writes to clean line are treated as misses

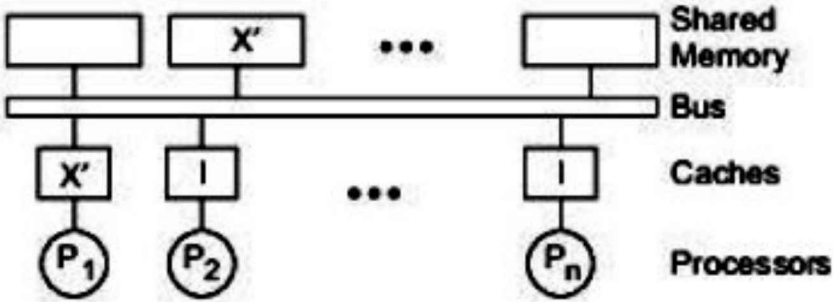
Write Invalidate versus Broadcast

- Invalidate requires one transaction per write-run
- Invalidate uses spatial locality: one transaction per block
- Broadcast has lower latency between write and read
- Broadcast: BW (increased) vs. latency (decreased)

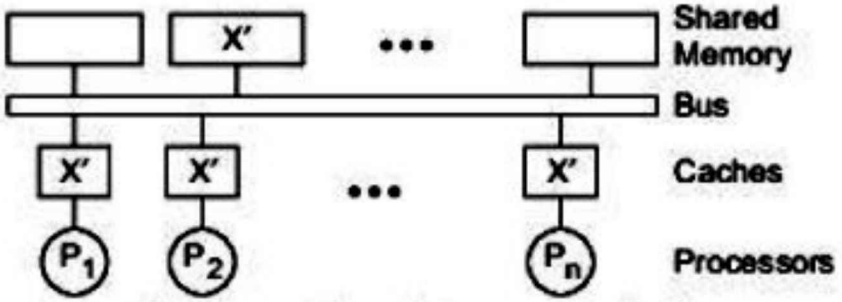
Write-Invalidate and write-update coherence protocols For write through caches (I: invalidate)



(a) Consistent copies of block X are in shared memory and three processor caches



(b) After a write-invalidate operation by P_1



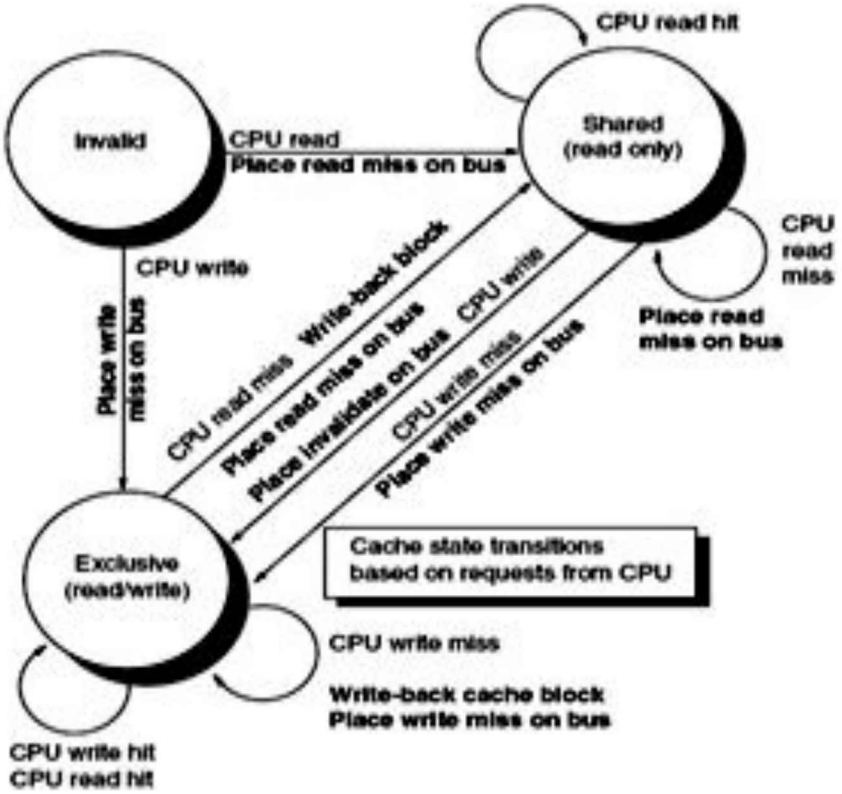
(c) After a write-update operation by P_1

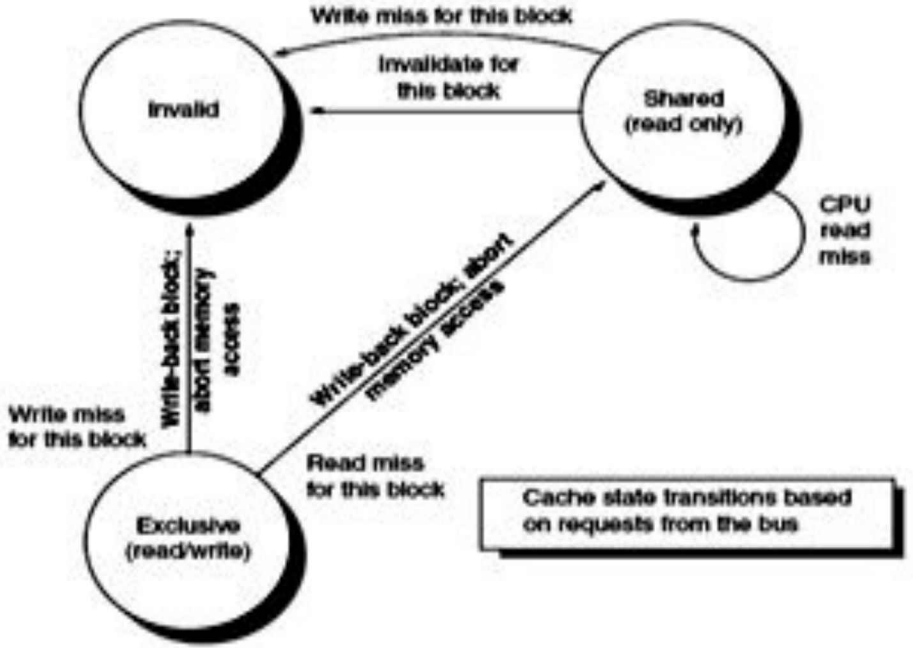
Cache Event and Action:

- *Read-miss:* When a processor wants to read a block that is not in the cache, a *read-miss* occurs. A *bus-read* operation will be initiated. If no *dirty* copy exists, then main memory has a consistent copy and supplies a copy to the requesting cache. If a *dirty* copy does exist in a remote cache, that cache will inhibit the main memory and send a copy to the requesting cache. In all cases, the cache copy will enter the *valid* state after a read-miss.
- *Write-hit:* If the copy is in the *dirty* or *reserved* state, the *write* can be carried out locally and the new state is *dirty*. If the new state is *valid*, a *write-invalidate* command is broadcast to all caches, invalidating their copies. The shared memory is *written through*, and the resulting state is *reserved* after this first *write*.
- *Write-miss*: When a processor fails to write in a local cache, the copy must come either from the main memory or from a remote cache with a dirty block. This is accomplished by sending a *read-invalidate* command which will invalidate all cache copies. The local copy is thus updated and ends up in a *dirty* state.
- *Read-hit*: Read-hits can always be performed in a local cache without causing a state transition or using the snoopy bus for invalidation.
- *Block Replacement:* If a copy is *dirty*, it has to be written back to main memory by block replacement. If the copy is *clean* (i.e., in either the *valid*, *reserved*, or *invalid* state), no replacement will take place.

Request	Source	State of addressed cache block	Type of cache action	Function and explanation
Read hit	Processor	Shared or modified	Normal hit	Read data in local cache.
Read miss	Processor	Invalid	Normal miss	Place read miss on bus.
Read miss	Processor	Shared	Replacement	Address conflict miss: place read miss on bus.
Read miss	Processor	Modified	Replacement	Address conflict miss: write-back block, then place read miss on bus.
Write hit	Processor	Modified	Normal hit	Write data in local cache.
Write hit	Processor	Shared	Coherence	Place invalidate on bus. These operations are often called upgrade or *ownership* misses, since they do not fetch the data but only change the state.
Write miss	Processor	Invalid	Normal miss	Place write miss on bus.
Write miss	Processor	Shared	Replacement	Address conflict miss: place write miss on bus.
Write miss	Processor	Modified	Replacement	Address conflict miss: write-back block, then place write miss on bus.
Read miss	Bus	Shared	No action	Allow shared cache or memory to service read miss.
Read miss	Bus	Modified	Coherence	Attempt to share data: place cache block on bus and change state to shared.
Invalidate	Bus	Shared	Coherence	Attempt to write shared block; invalidate the block.
Write miss	Bus	Shared	Coherence	Attempt to write shared block; invalidate the cache block.
Write miss	Bus	Modified	Coherence	Attempt to write block that is exclusive elsewhere; write-back the cache block and make its state invalid in the local cache.

A finite-state transition diagram for a single cache block using a write invalidation protocol and a write-back cache





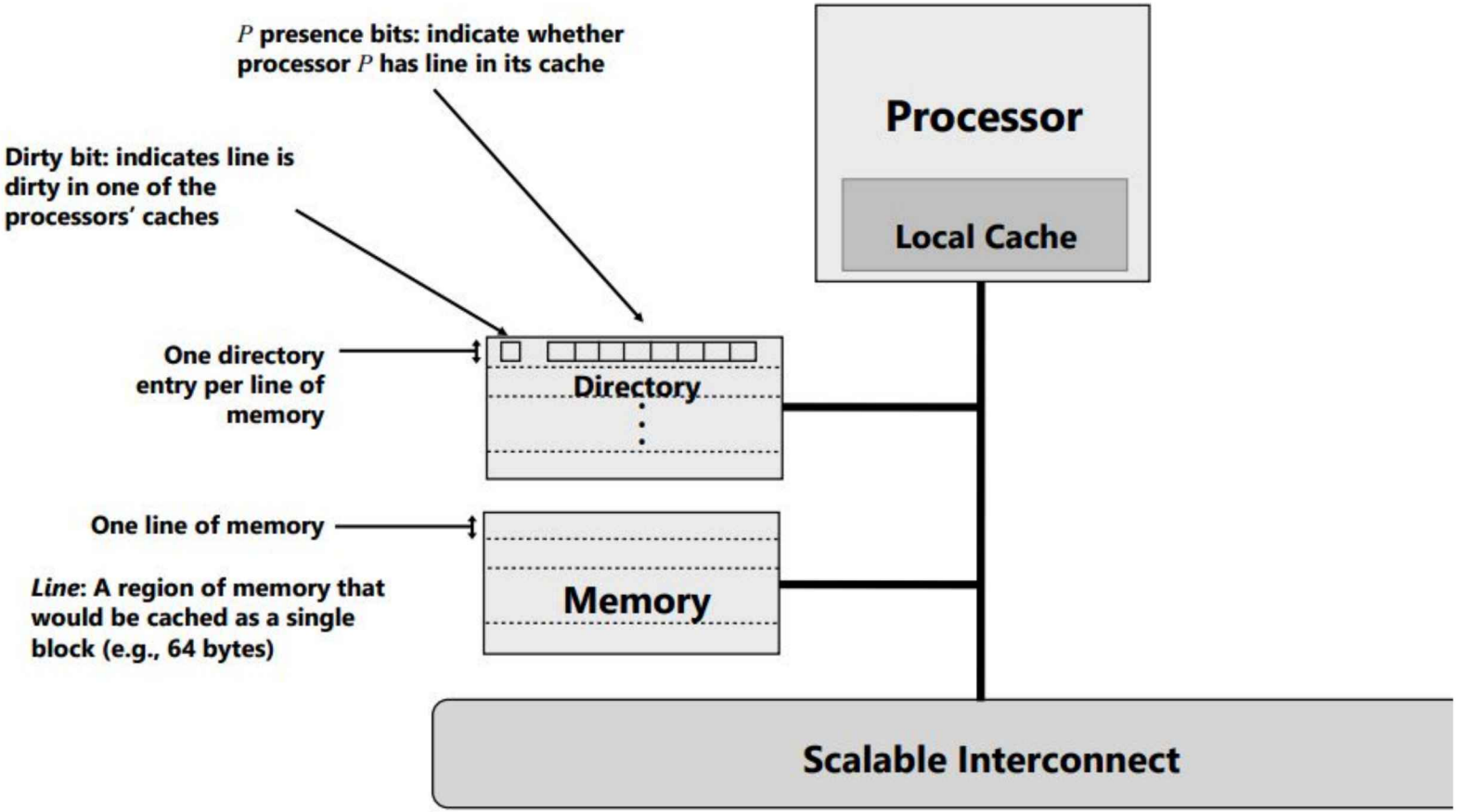
Limitations in Symmetric Shared-Memory Multiprocessors and Snooping Protocols

- As the number of processors in a multiprocessor grows, or as the memory demands of each processor grow, any centralized resource in the system can become a bottleneck.
- In the simple case of a bus -based multiprocessor, the bus and the memory become a bottleneck. So, scalability becomes an issue.
- In order to increase the communication bandwidth between processors and memory, designers have used multiple buses as well as interconnection networks, such as crossbars or small point-to-point networks. In such designs, the memory system can be configured into multiple physical banks, so as to boost the effective memory bandwidth while retaining uniform access time to memory.

Directory Protocol

- In a directory-based protocols system, data to be shared are placed in a common directory that maintains the coherence among the caches.
- Here, the directory acts as a filter where the processors ask permission to load an entry from the primary memory to its cache memory.
- If an entry is changed the directory either updates it or invalidates the other caches with that entry.

A Simple Directory



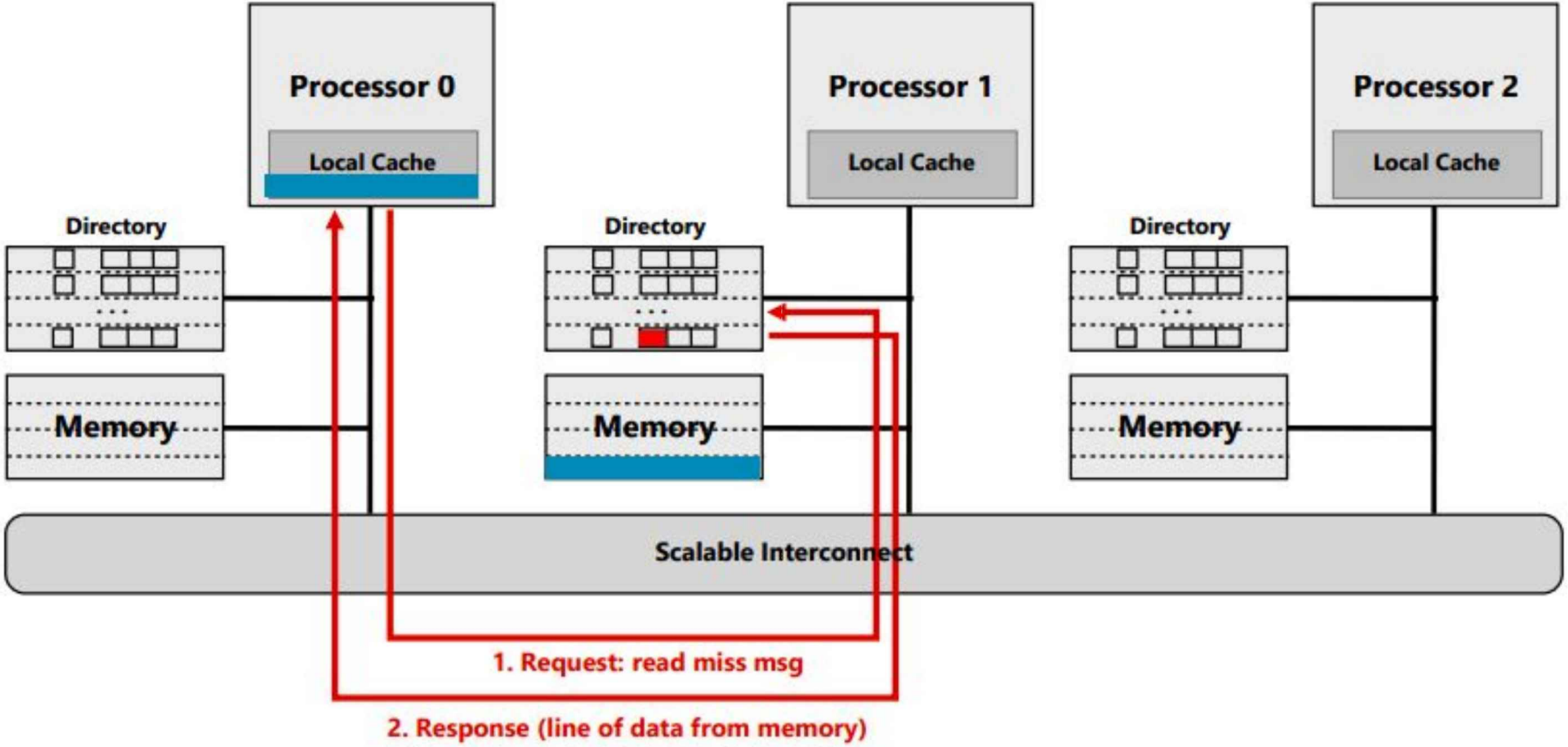
Directory Protocol

- Similar to Snoopy Protocol: Three states
  - **Shared:** 1 processors have data, memory up-to-date
  - **Uncached** (no processor has it; not valid in any cache)
  - **Exclusive:** 1 processor (owner) has data; memory out of-date
- In addition to cache state, must track which processors have data when in the shared state (usually bit vector, 1 if processor has copy)
- Keep it simple(r):
  - Writes to non-exclusive data => write miss
  - Processor blocks until access completes
  - Assume messages received and acted upon in order sent

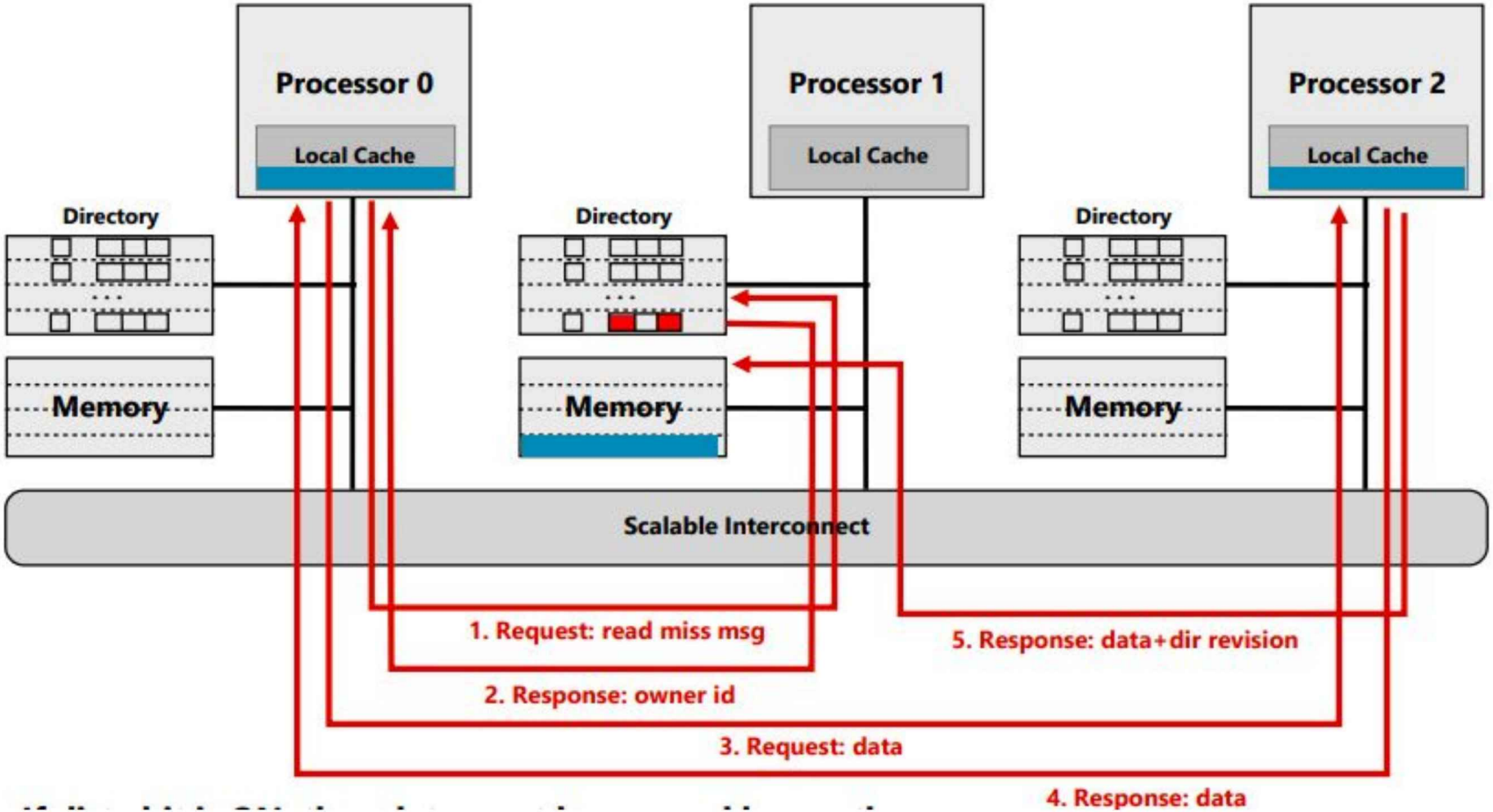
Directory Protocol

- No bus and don’t want to broadcast:
  - interconnect no longer single arbitration point
  - all messages have explicit responses
- Terms:
  - **Local node** is the node where a request originates
  - **Home node** is the node where the memory location of an address resides
  - **Remote node** is the node that has a copy of a cache block, whether exclusive or shared

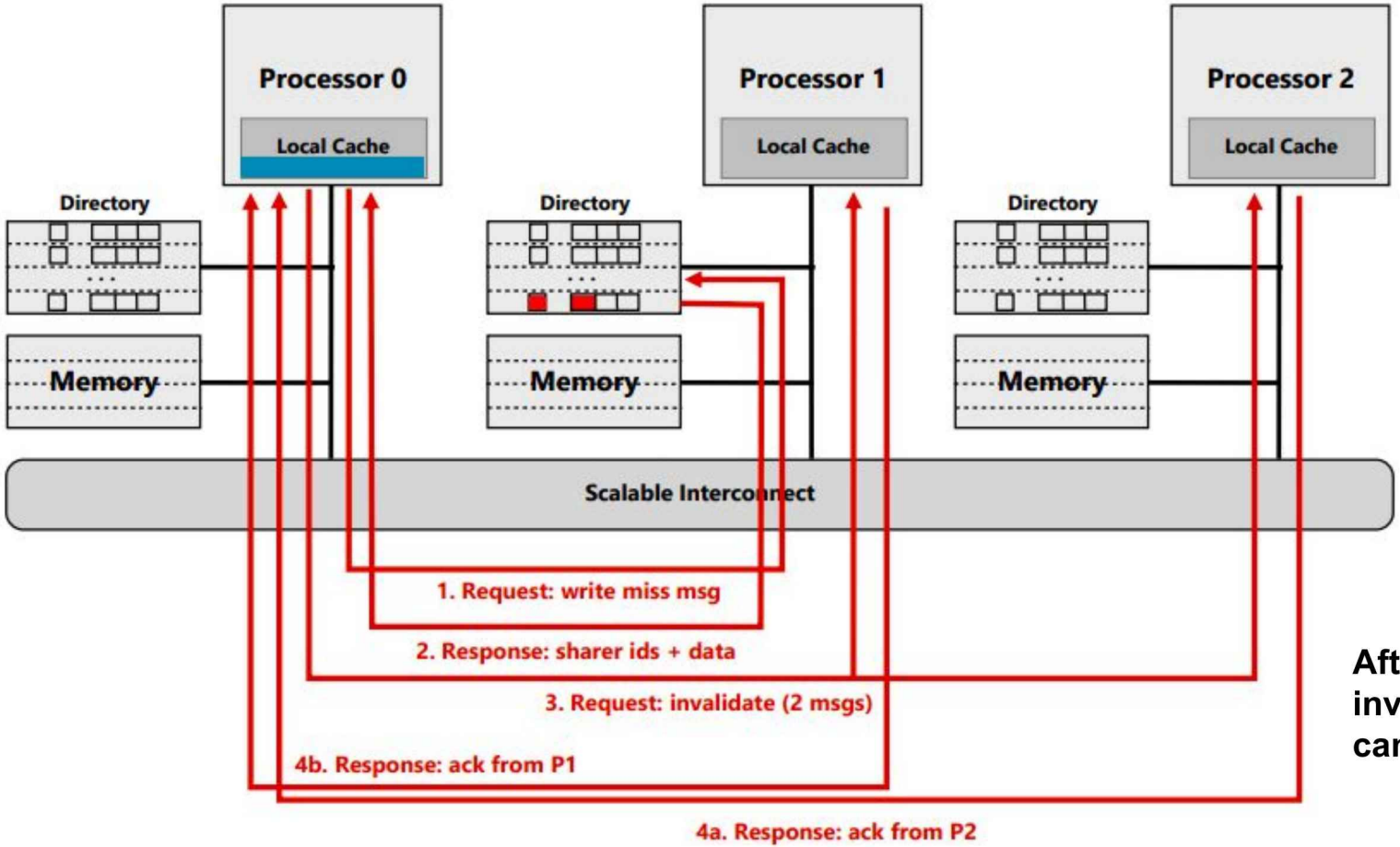
Example 1: read miss to clean line



Example 2: read miss to dirty line



Example 3: write miss



After receiving both invalidation acks, P0 can perform write

Directory Protocol Messages

Message type	Source	Destination	Msg
Read miss	Local cache	Home directory	P, A
		– *Processor P reads data at address A; send data and make P a read sharer*	
Write miss	Local cache	Home directory	P, A
		– *Processor P writes data at address A; send data and make P the exclusive owner*	
Invalidate	Home directory	Remote caches	A
		– *Invalidate a shared copy at address A.*	
Fetch	Home directory	Remote cache	A
		– *Fetch the block at address A and send it to its home directory*	
Fetch/Invalidate	Home directory	Remote cache	A
		– *Fetch the block at address A and send it to its home directory; invalidate the block in the cache*	
Data value reply	Home directory	Local cache	Data
		– *Return a data value from the home memory*	
Data write-back	Remote cache	Home directory	A, Data
		– *Write-back a data value for address A*	

P = processor number,
A = address

Directory Protocol

- Message sent to directory causes two actions:
  - Update the directory
  - More messages to satisfy request
- Block is in **Uncached** state: the copy in memory is the current value; only possible requests for that block are:
  - **Read miss:** requesting processor sent data from memory & requestor made only sharing node; state of block made Shared.
  - **Write miss:** requesting processor is sent the value & becomes the Sharing node. The block is made Exclusive to indicate that the only valid copy is cached. Sharers indicates the identity of the owner.
- Block is **Shared** => the memory value is up-to-date:
  - **Read miss:** requesting processor is sent back the data from memory & requesting processor is added to the sharing set.
  - **Write miss:** requesting processor is sent the value. All processors in the set Sharers are sent invalidate messages, & Sharers is set to identity of requesting processor. The state of the block is made Exclusive.

Directory Protocol

- Block is **Exclusive:** current value of the block is held in the cache of the processor identified by the set Sharers (the owner) => three possible directory requests:
  - **Read miss:** owner processor sent data fetch message, which causes state of block in owner's cache to transition to Shared and causes owner to send data to directory, where it is written to memory & sent back to requesting processor. Identity of requesting processor is added to set Sharers, which still contains the identity of the processor that was the owner (since it still has a readable copy).
  - **Data write-back:** owner processor is replacing the block and hence must write it back. This makes the memory copy up-to date (the home directory essentially becomes the owner), the block is now uncached, and the Sharer set is empty.
  - **Write miss:** block has a new owner. A message is sent to old owner causing the cache to send the value of the block to the directory from which it is sent to the requesting processor, which becomes the new owner. Sharers is set to identity of new owner, and state of block is made Exclusive.