中山大學 SUN YAT-SEN UNIVERSITY

国家超级计算广州中心 NATIONAL SUPERCOMPUTER CENTER IN GUANGZHOU

Advanced Computer Architecture

高级计算机体系结构

第16讲：TLP (2)

张献伟

xianweiz.github.io

DCS5637, 12/14/2022

Review Questions

- DDR-1000MHz, 64b interface, what's the bandwidth?

  1G x 2 x 64b/8 = 16 GB/s

- sort DDR/HBM/GDDR in bandwidth ascending order?

  DDR -> GDDR -> HBM (or, HBM -> DDR -> GDDR)

- NVM vs. DRAM?

  Larger capacity, slower access, lower cost, less power, …

- Data X is shared in processors A and B. Steps for A to write X? (note: cache is write-back)?

  Acquires bus, sends invalidate, then updates X (shared → modified)

- Next, processor B reads X. What will happen?

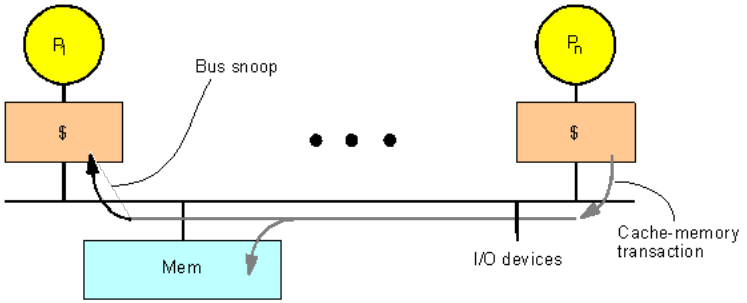
  Places a miss on bus, A responds data and also writes back to mem.

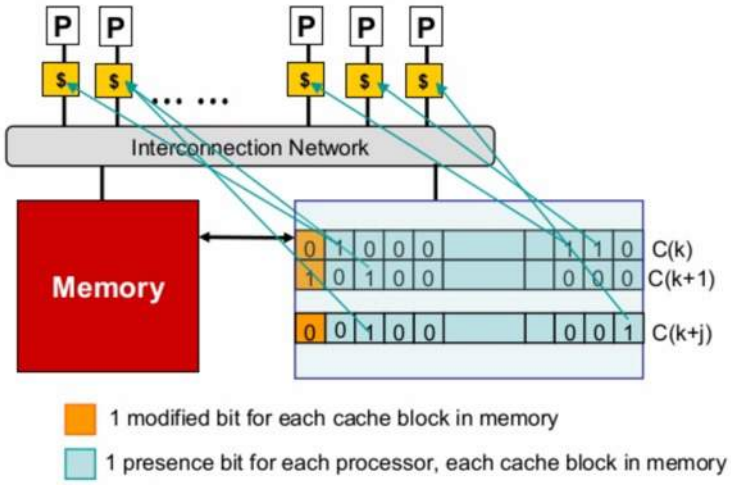
- MSI protocol?

  Modified/Shared/Invalid. Invalidation protocol for write-back $.

Coherence Protocols[缓存一致性协议]

- Cache **coherence protocols**: the rules to maintain coherence for multiple processors
  - Key is to track the state of any sharing of a data block
- Two classes of protocols
  - Snooping[窥探]
    - Each core tracks sharing status of each block
  - Directory based[基于目录]
    - Sharing status of each block kept in one location





MSI Protocol

- Invalidation protocol for write-back caches
- Each data block can be[数据块状态]
  - Uncached: not in any cache
  - Clean in one or more caches and up-to-date in memory, or
  - Dirty in exactly one cache

Dirty in more caches???

- Correspondingly, we record the coherence state of each block in a cache as[一致性状态]
  - Invalid: block contains no valid data
  - Shared: a clean block (can be shared by other caches), or
  - Modified/Exclusive: a dirty block (cannot be in any other cache)

MSI protocol = Modified/Shared/Invalid

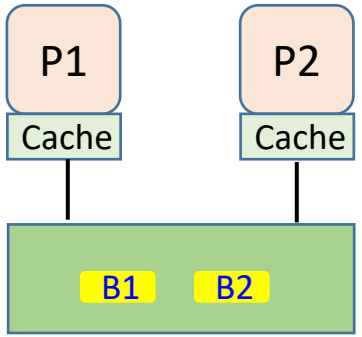
Makes sure that if a block is dirty in one cache, it is not valid in any other cache and that a read request gets the most updated data

MSI Protocol (cont.)

- A read miss to a block in a cache, *C1*, generates a bus transaction[读不命中]
  - If another cache, *C2*, has the block "modified", it has to write back the block before memory supplies it[其他cache有新数据]
    - *C1* gets data from the bus and the block becomes "shared" in both caches
- A write hit to a shared block in *C1* forces an "Invalidate"[写命中-'共享']
  - Other caches that have the block should invalidate it – the block then becomes "modified" in *C1*[其他cache作废数据]
- A write hit to a modified block does not generate "Invalidate" or change of state[写命中-'修改']
- A write miss (to an invalid block) in *C1* generates a bus transaction[写不命中]
  - If a cache, *C2*, has the block as "shared", it invalidates it
  - If a cache, *C2*, has the block in "modified", it writes back the block and changes it state in *C2* to "invalid"
  - If no cache supplies the block, the memory will supply it
  - When *C1* gets the block, it sets its state to "modified"

Example

- Assume that
  - Blocks *B1* and *B2* map to the same cache location *L*
  - Initially neither *B1* or *B2* is cached
  - Block size = one word



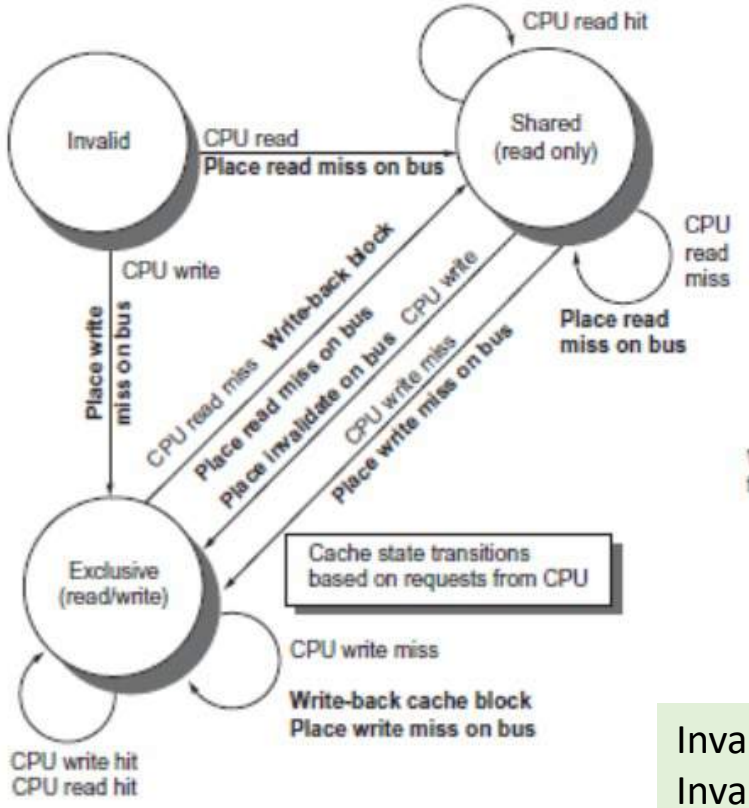
Event	In P1's cache	In P2's cache
	L = invalid	**L = invalid**
P1 writes 10 to B1 (write miss)	L <- B1 = 10 (modified)	L = invalid
P1 reads B1 (read hit)	L <- B1 = 10 (modified)	L = invalid
P2 reads B1 (read miss)	B1 is written back L <- B1 = 10 (shared)	L <- B1 = 10 (shared)
P2 writes 20 to B1 (write hit)	L = invalid	Put invalidate B1 on bus L <- B1 = 20 (modified)
P2 writes 40 to B2 (write miss)	L = invalid	B1 is written back L <- B2 = 40 (modified)
P1 reads B1 (read miss)	L <- B1 = 20 (shared)	L <- B2 = 40 (modified)

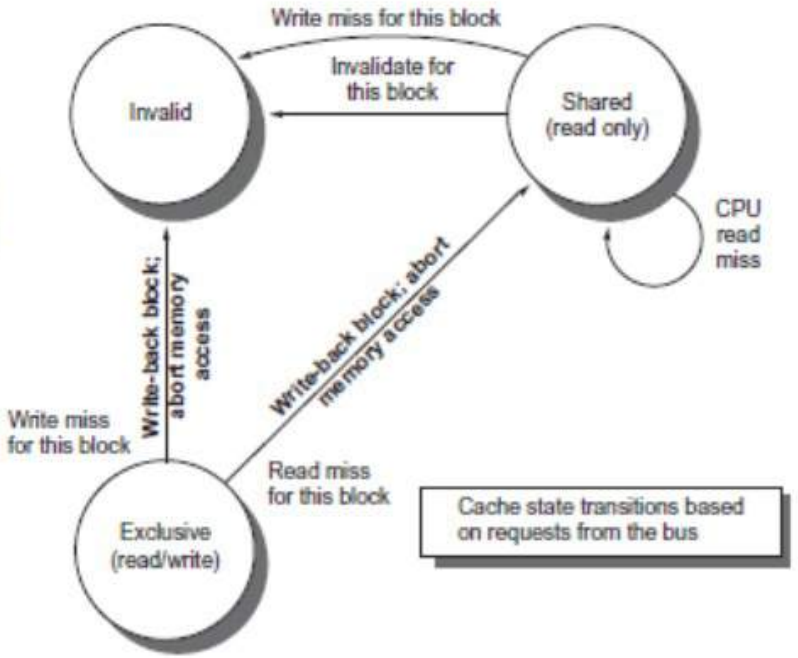
The Protocol

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, because 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 read shared data: place cache block on bus, write-back block, 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.

Formal Specification[形式化定义]

- Finite state transition diagram for a single private cache block[状态转换图]
  - Transitions based on processor and bus requests, respectively

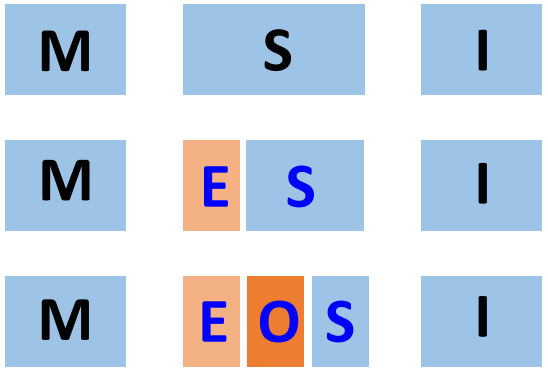




Invalid/Exclusive → Shared: a read happens
Invalid/Shared → Exclusive: a write happens
Shared/Exclusive → Invalid: write-invalidation

MSI Issues & Extensions[扩展]

- Complications for the basic MSI protocol
  - Operations are not atomic[非原子操作]
    - E.g. detect miss, acquire bus, receive a response
    - Creates possibility of deadlock and races
  - One solution: processor that sends invalidate can hold bus until other processors receive the invalidate
- MSI: always invalidate before writing
- Extensions
  - Adding additional states and transitions, which optimize certain behaviors, possibly resulting in improved performance
  - Two common extensions
    - **MESI**: new 'Exclusive'
    - **MOESI**: new 'Exclusive' and 'Owner'



MESI and MOESI

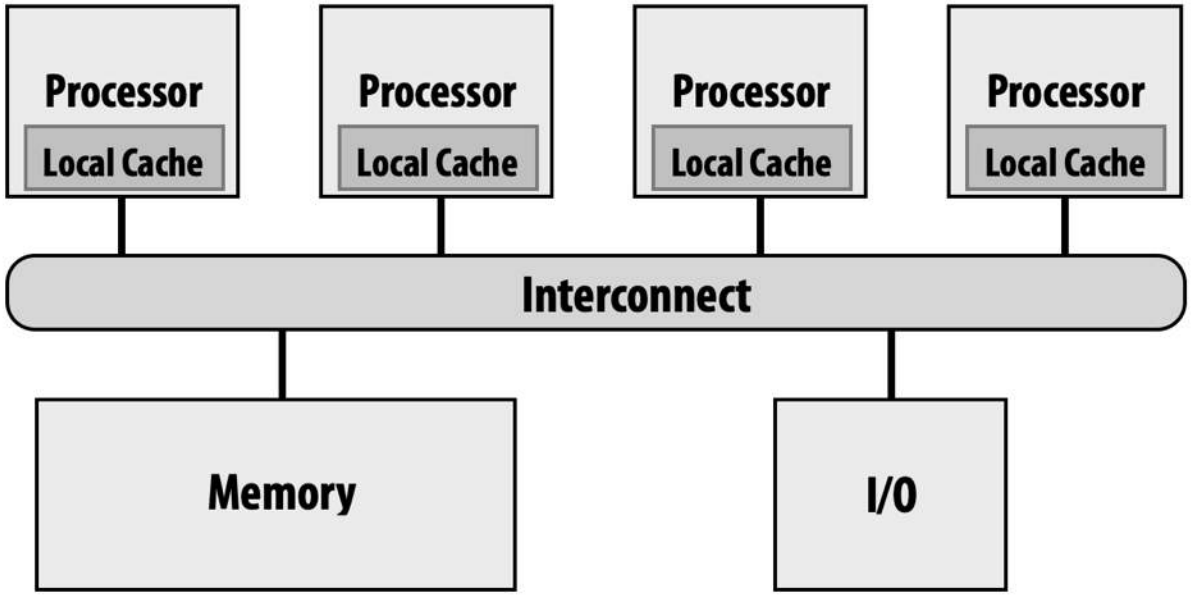
- MESI adds state Exclusive

  - Shared: Exclusive (only one cache) + Shared (2 or more caches)
  - Indicate when a cache block is resident only in a single cache but is clean[其他cache都没有]
  - A subsequent write to a block in *E* state by the same core need not acquire bus access or generate an invalidate
- MOESI further adds state Owner

  - Shared: Shared Modified (O) + Shared Clean (S)
  - Indicate that the associated block is owned by that cache and out-of-date in memory[独有，且比内存新]
  - In MSI/MESI, when sharing a block in *M* state, the state is changed to *S*, and the block must be written back to memory
  - In MOESI, the block can be changed from *M* to *O* without writing it to memory

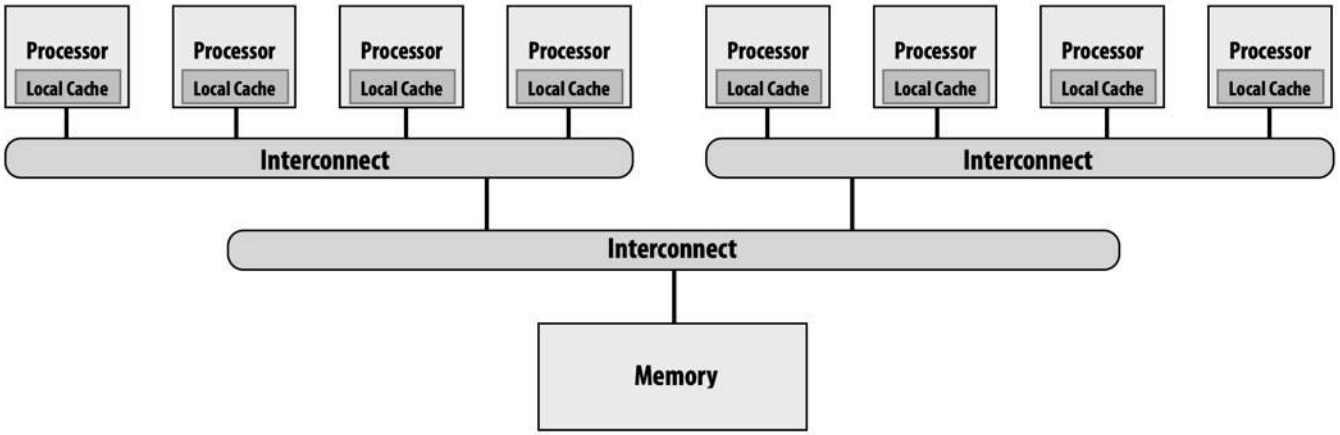
Limits of Snooping Protocol[局限]

- Snooping cache coherence protocols rely on broadcasting coherence info to all processors over the chip inter-connect[依赖于广播]
  - Cache miss occurred, triggering cache communicated with all other caches



Scaling Cache Coherence

- One possible solution: hierarchical snooping[多层级]
  - Use snooping coherence at each level



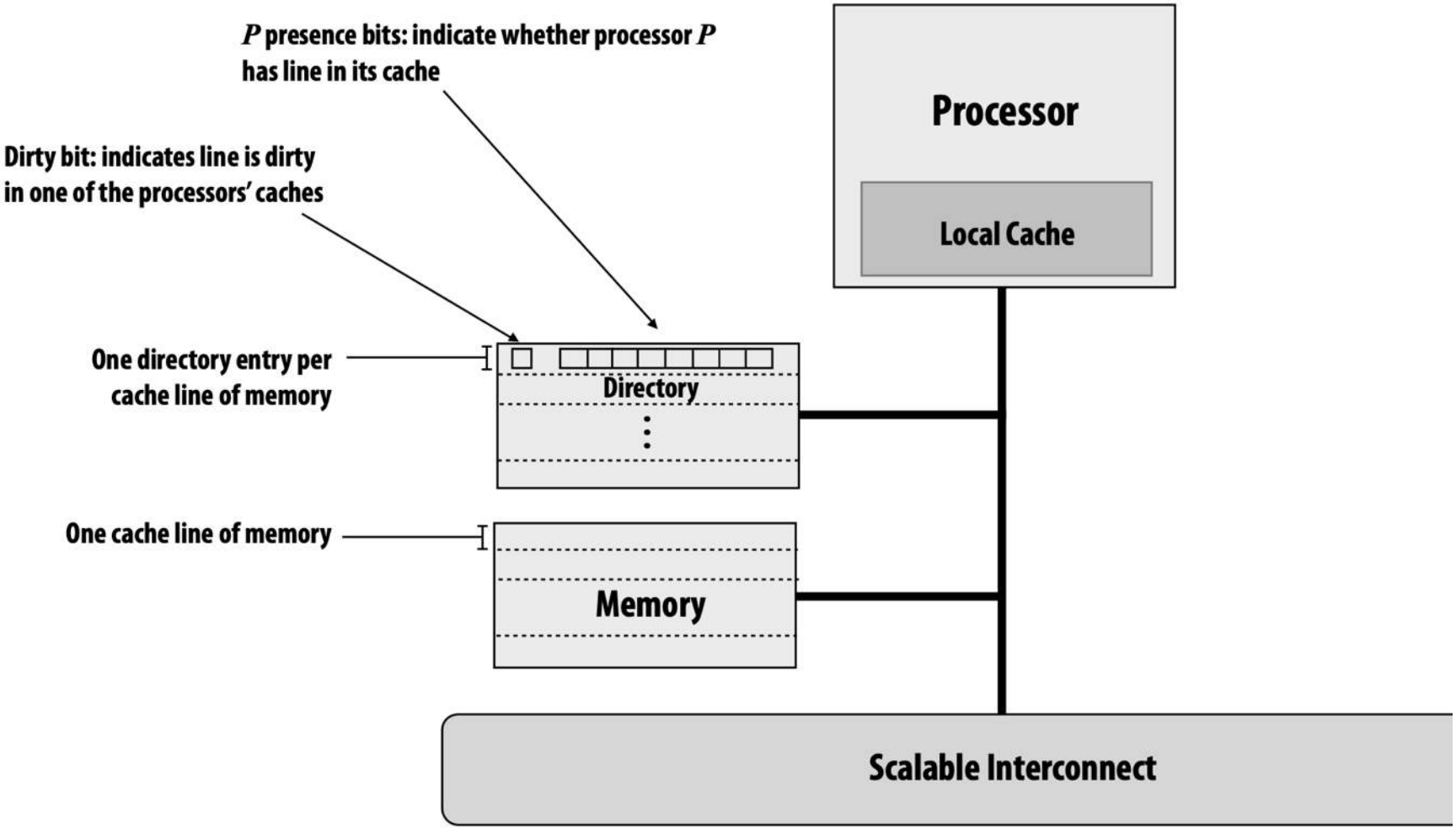
  - Advantages
    - Relatively simple to build (already have to deal with similar issues due to multi-level caches)
  - Disadvantages
    - The root of network may become a performance bottleneck
    - Larger latencies than direct communication
    - Doesn't apply to more general network topologies

Scalable Coherence using Directories

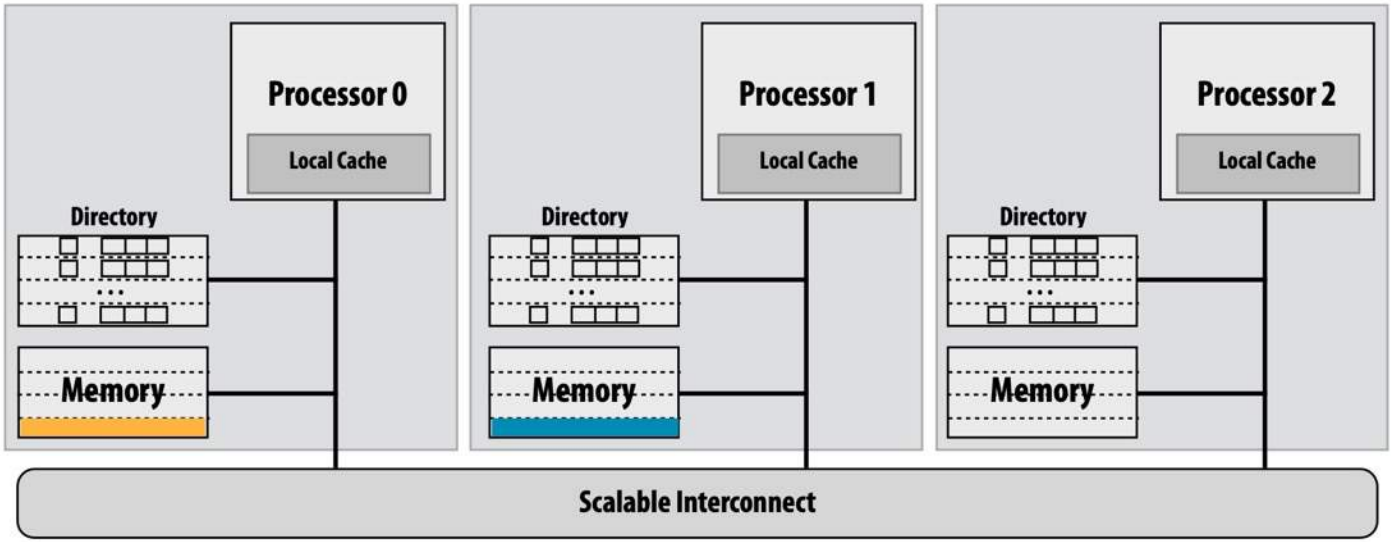
- To avoid broadcast by storing info about status of the line in one place: directory[目录]
  - The directory entry for a cache line contains information about the state of the cache line in all caches[保存状态]
  - Caches look up information from the directory as necessary[查询目录]
  - Cache coherence is maintained by point-to-point messages between the caches (not by broadcast mechanisms)[点对点通信]

- Theoretical advantages of directory-based approach
  - The root of network won't be the performance bottleneck
  - Can apply to more general network topologies(e.g. meshes, cubes)

Simple Directory Protocol Impl.



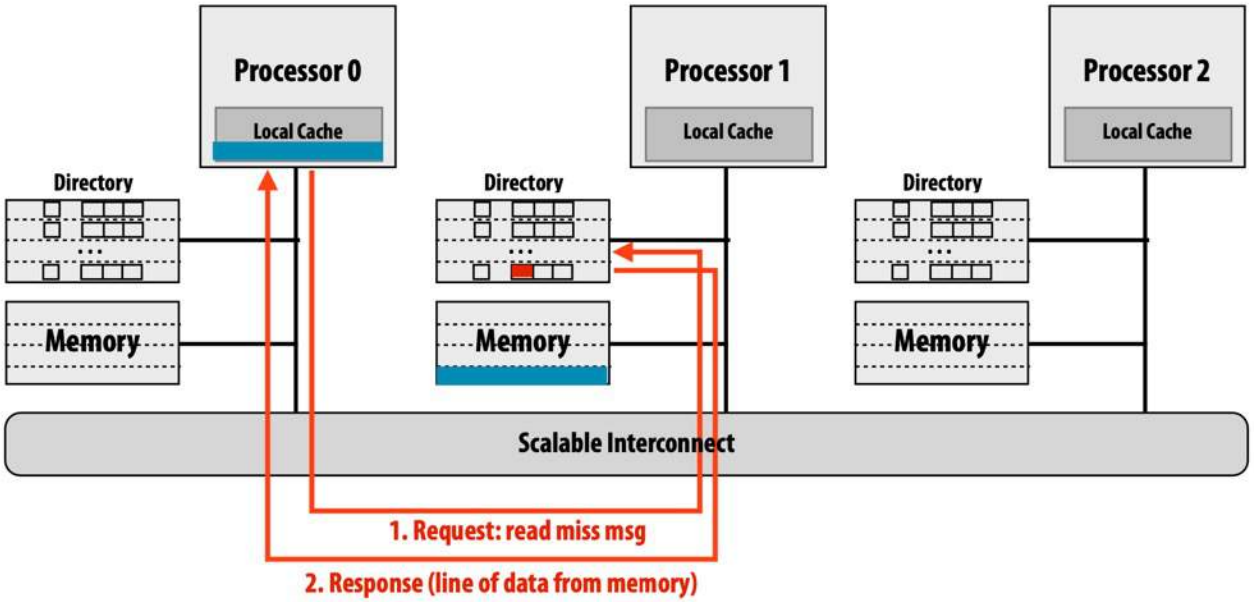
Distributed Directory: Partition[分区]



- Directory partition is co-located with memory it describes
- "**Home node**" of a line: node with memory holding the corresponding data for the line
  - For example: node 0 is the home node of orange line, node 1 is the home node of blue line
- "**Requesting node**": node containing processor requesting line

Example: read miss to clean line

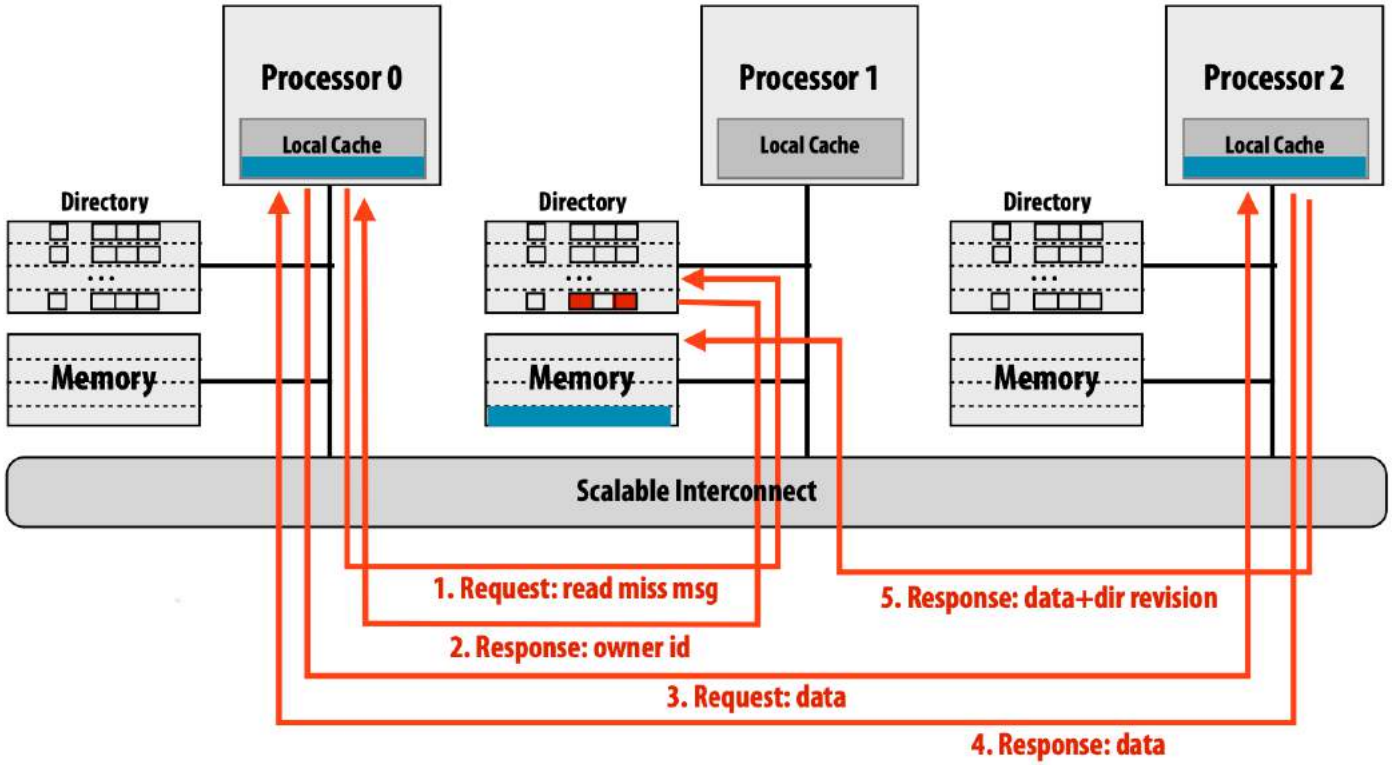
- Read from main memory by processor 0 of blue line (not dirty)



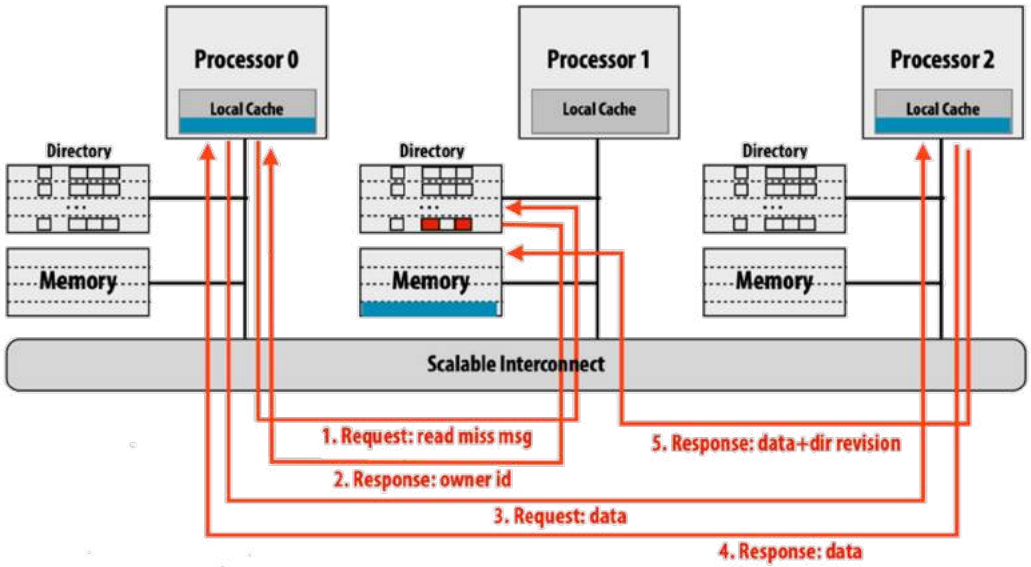
- Read miss message sent to home node of requested line
- Home directory checks entry for line
  - If dirty bit of line is OFF, respond with contents from memory, set presence[0] to true (to indicate line is cached by processor 0)

Example: read miss to dirty line

- Read from main memory by processor 0 of blue line
  - Dirty and its content is in P2's cache



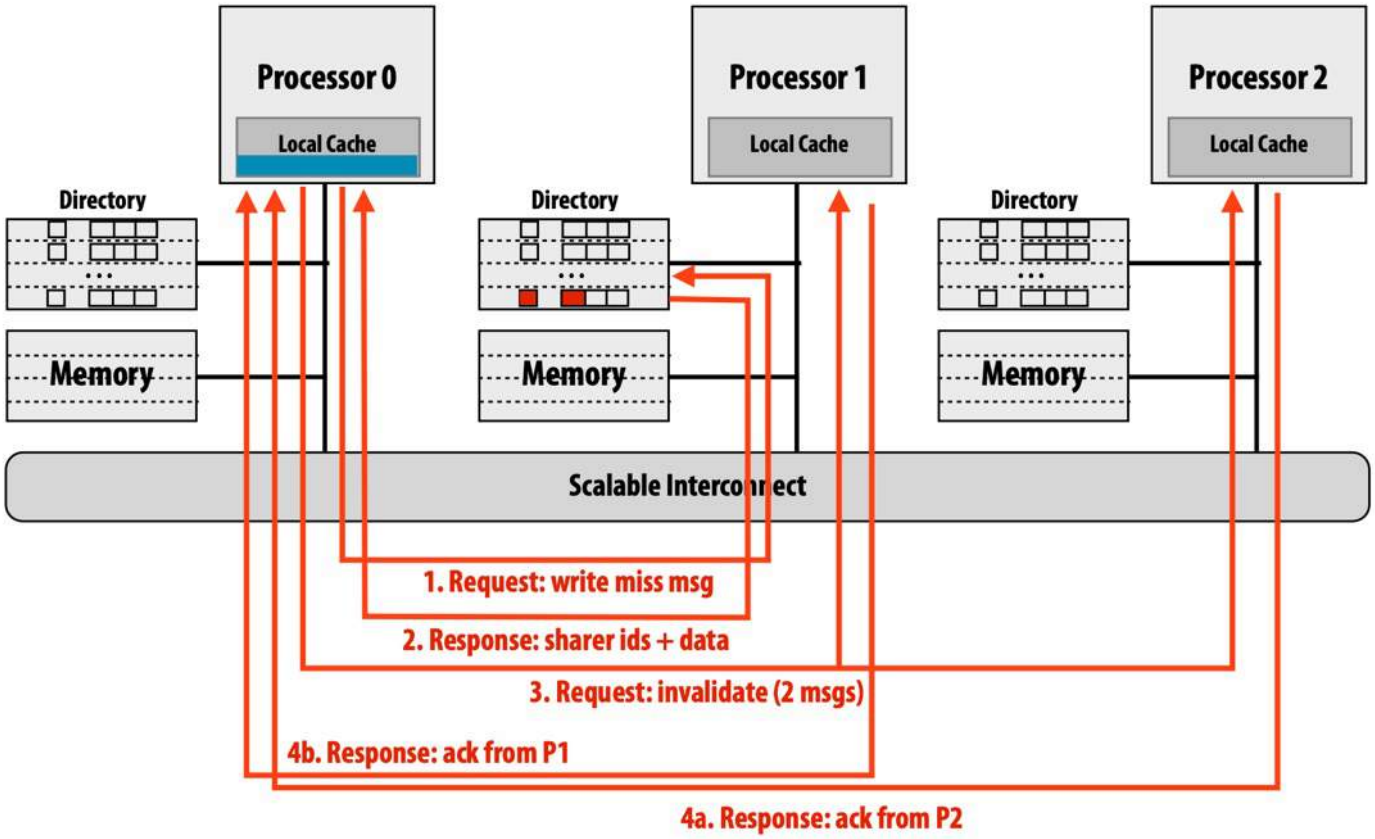
Example: read miss to dirty line (cont.)



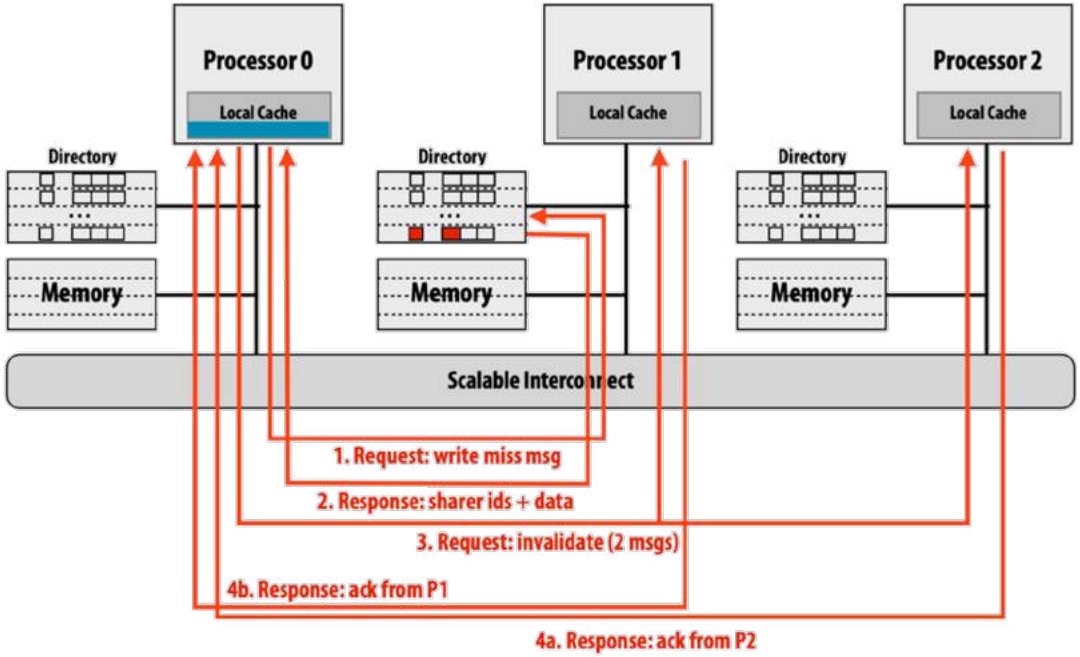
1. If dirty bit is ON, data must be sourced by another processor
2. Home node responds with id of line owner
3. Requesting node requests data from owner
4. Owner responds to requesting node
   - changes state in cache to SHARED (read only)
5. Owner also responds to home node, home clears dirty
   - updates presence bits, updates memory

Example: write miss

- Write to memory by processor 0
  - Line is clean, but resident in P1's and P2's caches



Example: write miss (cont.)



1. Requesting node sends the write miss to home node
2. Home node responds with ids of nodes containing this data (sharer) and data
3. Requesting sharer to invalidate corresponding data
4. Get response from P1 and P2
   - After receiving both invalidation acks, P0 can write

http://15418.courses.cs.cmu.edu/spring2017/lecture/directorycoherence

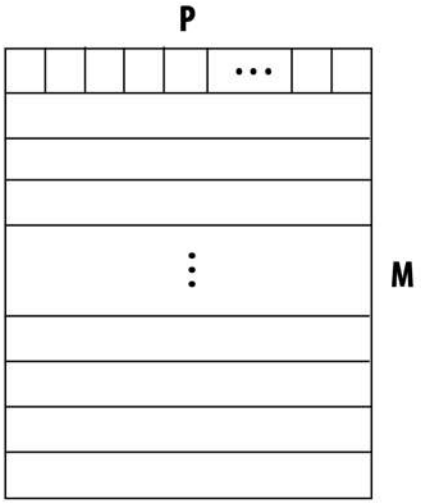
Pros of Directory Protocol

- On reads, directory tells requesting node exactly where to get the line from
  - Either from home node (if the line is clean)
  - Or from the owning node (if the line is dirty)
  - Either way, retrieving data involves only point-to-point communication

- On writes, the advantages 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 in a snooping protocol)

Cons of Directory Protocol

- Full bit vector directory representation
  - One presence bit per node

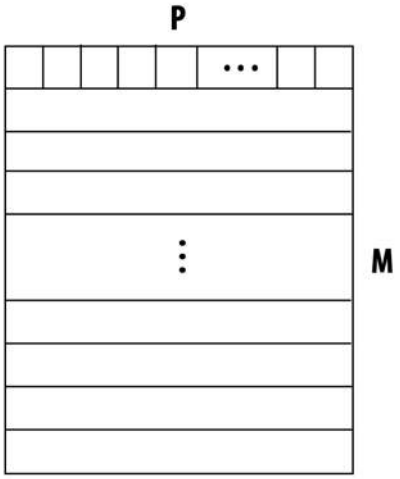


- Storage proportional to P * M
  - P = number of nodes (e.g., processors)
  - M = number of lines in memory
- Storage overhead rises with P
  - Assume 64 byte cache line size (512 bits)
  - 64 nodes (P=64) -> 12.5% overhead
  - 256 nodes (P=256) -> 50% overhead
  - 1024 nodes (P=1024) -> 200% overhead

Reducing Storage Overheads

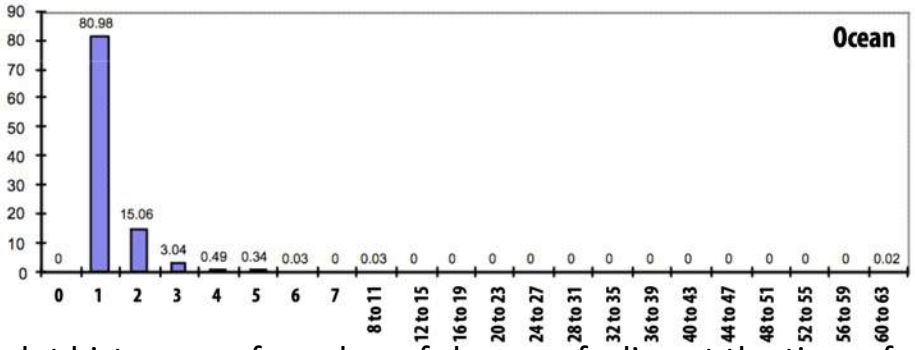
- Optimizations on full-bit vector scheme
  - Increase cache line size (reduce M term)
  - Group multiple processors into a single directory "node" (reduce P term)
    - Need only one directory bit per node, not one bit per processor
    - Hierarchical: could use snooping protocol to maintain coherence among processors in a node, directory across nodes

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



Limited Pointer Schemes (LPS)[有限指针]

- Since data is expected to only be in a few caches at once, storage for a limited number of pointers per directory entry should be sufficient (only need a list of the nodes holding a valid copy of the line)[数据通常小范围内共享]
  - Example:
    - In a 1024 processor system
    - Full bit vector scheme needs 1024 bits per line
    - Using limited pointer scheme, 1024 bits can store approximately 100 pointers to nodes holding the line (log(1024) = 10 bits per pointer)
    - In practice, we can get by with far less than this (20-80 principle)



Graphs plot histogram of number of sharers of a line at the time of a write

http://15418.courses.cs.cmu.edu/spring2017/lecture/directorycoherence

Managing Overflow in LPS[管理溢出]

If too many pointers (sharers) are required…

- Fallback to broadcast (if broadcast mechanism exists)[广播]
  - When more than max number of sharers, revert to broadcast
- If no broadcast mechanism present on machine[阈值]
  - Don't allow more than a max number of sharers
  - On overflow, newest sharer replaces an existing one (must invalidate line in the old sharer's cache)
- Coarse vector fallback[粗粒度]
  - Revert to 'bit' vector representation
  - Each bit corresponds to K nodes
  - On write, invalidate all nodes a bit corresponds to

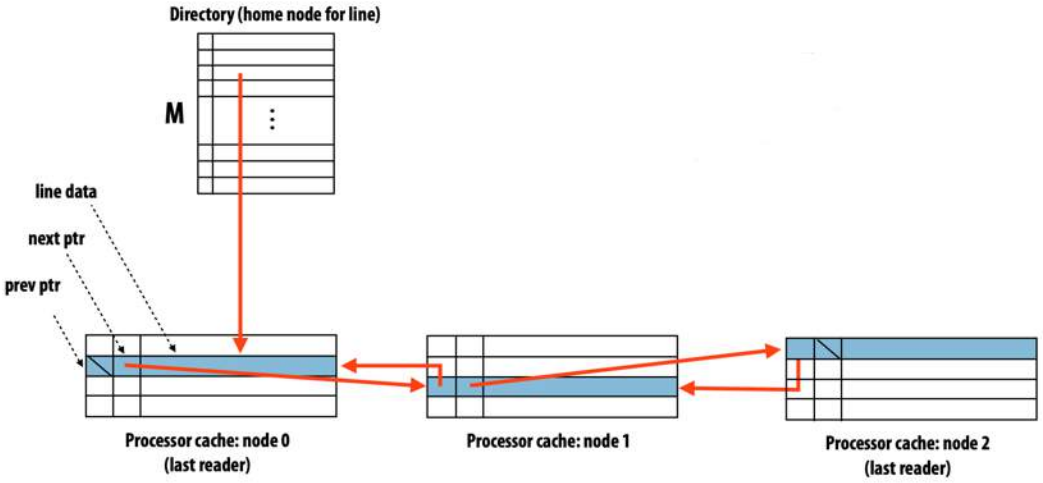
Summary of Limited Pointer Schemes

- LPS reduces directory storage overhead caused by large P
  - By adopting a compact representation of a list of shares

- But do we really need to maintain storage for a list for each cache-line chunk of data in memory?
- Key observation: the majority of memory is NOT resident in cache. And to carry out coherence protocol the system only needs sharing information for lines that are currently in cache[仅小部分数据被缓存]
  - Most directory entries are empty most of the time
  - 1 MB cache, 1 GB memory per node -> 99.9% of directory entries are idle

Sparse Directories[稀疏目录]

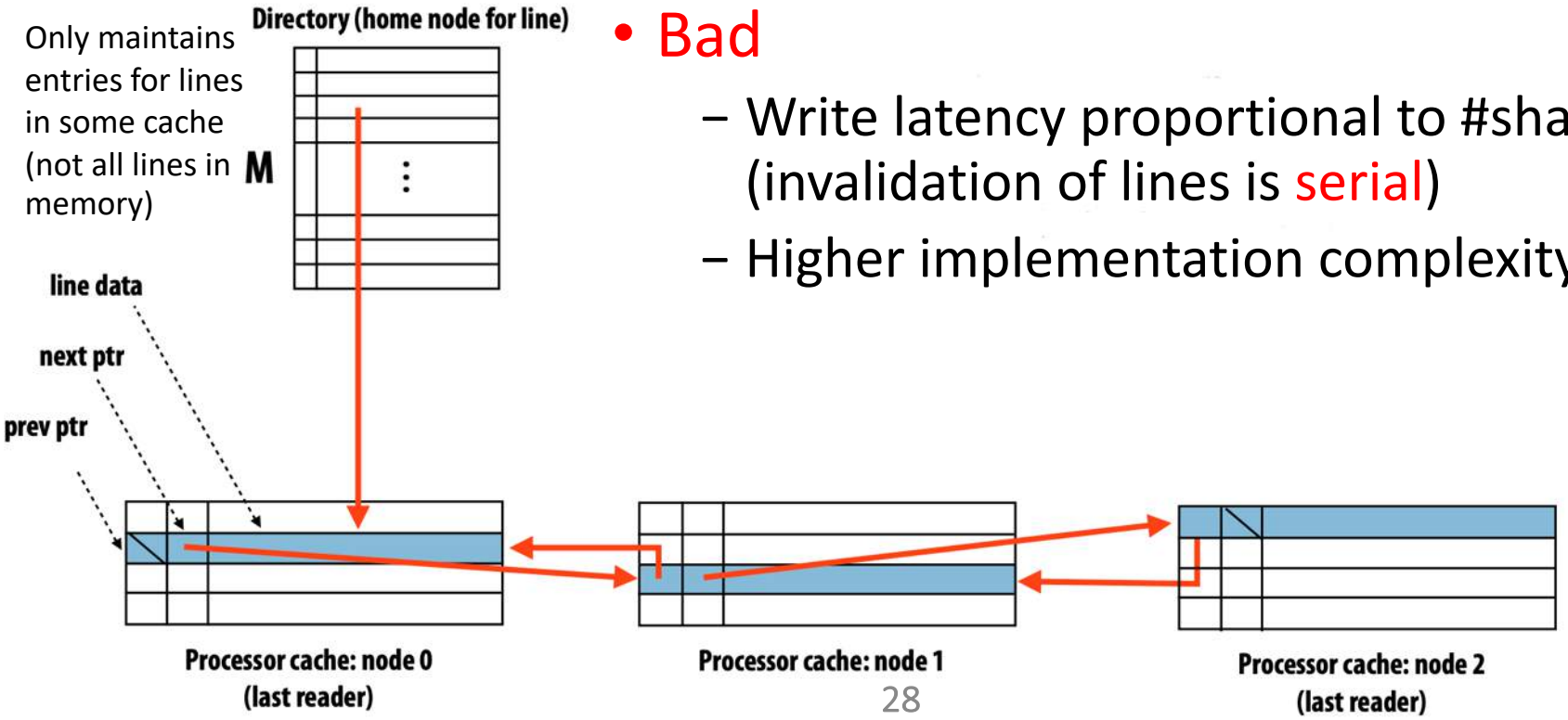
- Directory at home node maintains pointer to only one node caching line (not a list of sharers)[仅指向一个]
- Pointer to next node in list is stored as extra information in the cache line (like the line's tag, dirty bits, etc.)[链表]
- On read miss: add requesting node to head of list
- On write miss: propagate invalidations along list
- On evict: need to patch up list (linked list removal)



http://15418.courses.cs.cmu.edu/spring2017/lecture/directorycoherence

Scaling Properties of Sparse Directories

- Good
  - Low memory storage overhead (one pointer to list head per line)
  - Additional directory storage is proportional to cache size (the list stored in SRAM)
  - Traffic on write is still proportional to number of sharers

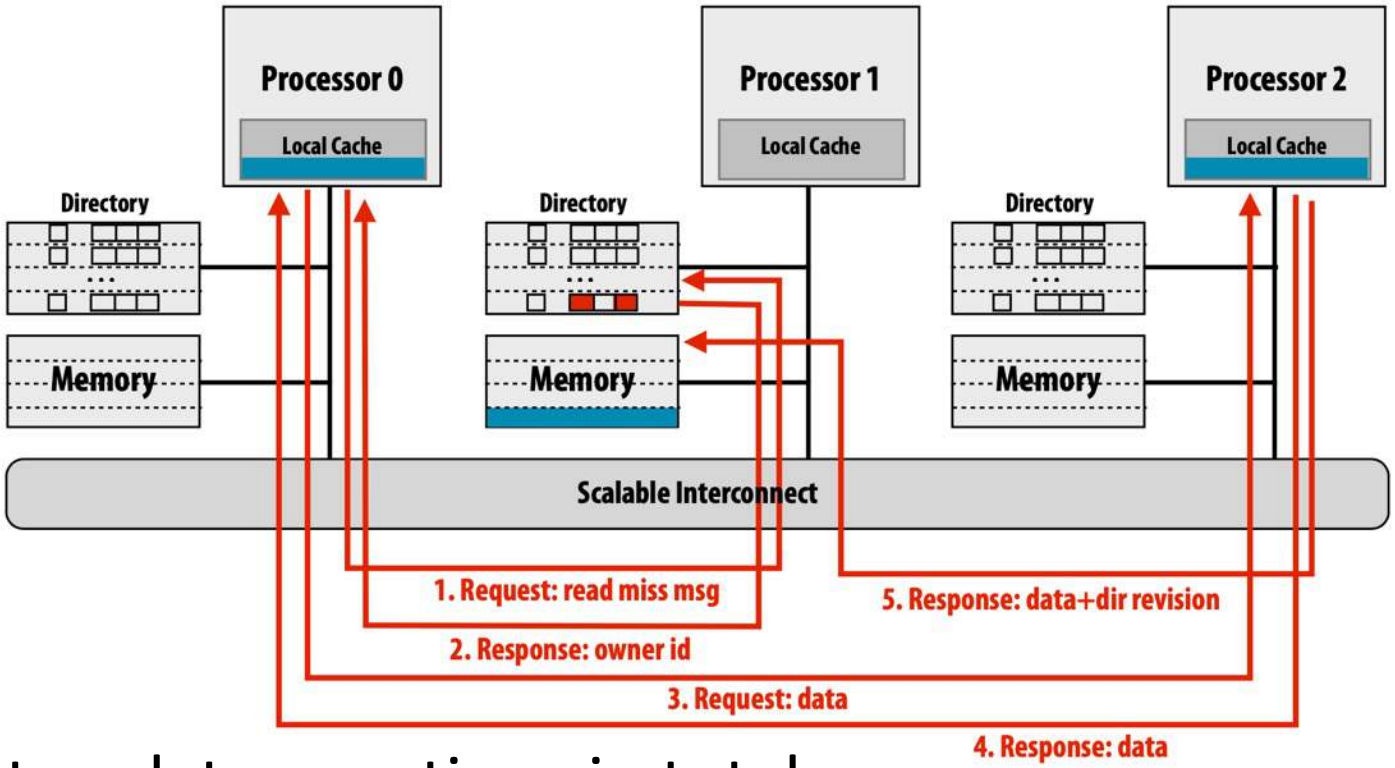


- Bad
  - Write latency proportional to #sharers (invalidation of lines is serial)
  - Higher implementation complexity

http://15418.courses.cs.cmu.edu/spring2017/lecture/directorycoherence

Reduce #msg. Sent

Read from main memory by P0 of the blue line: line is dirty (contained in P2's cache)

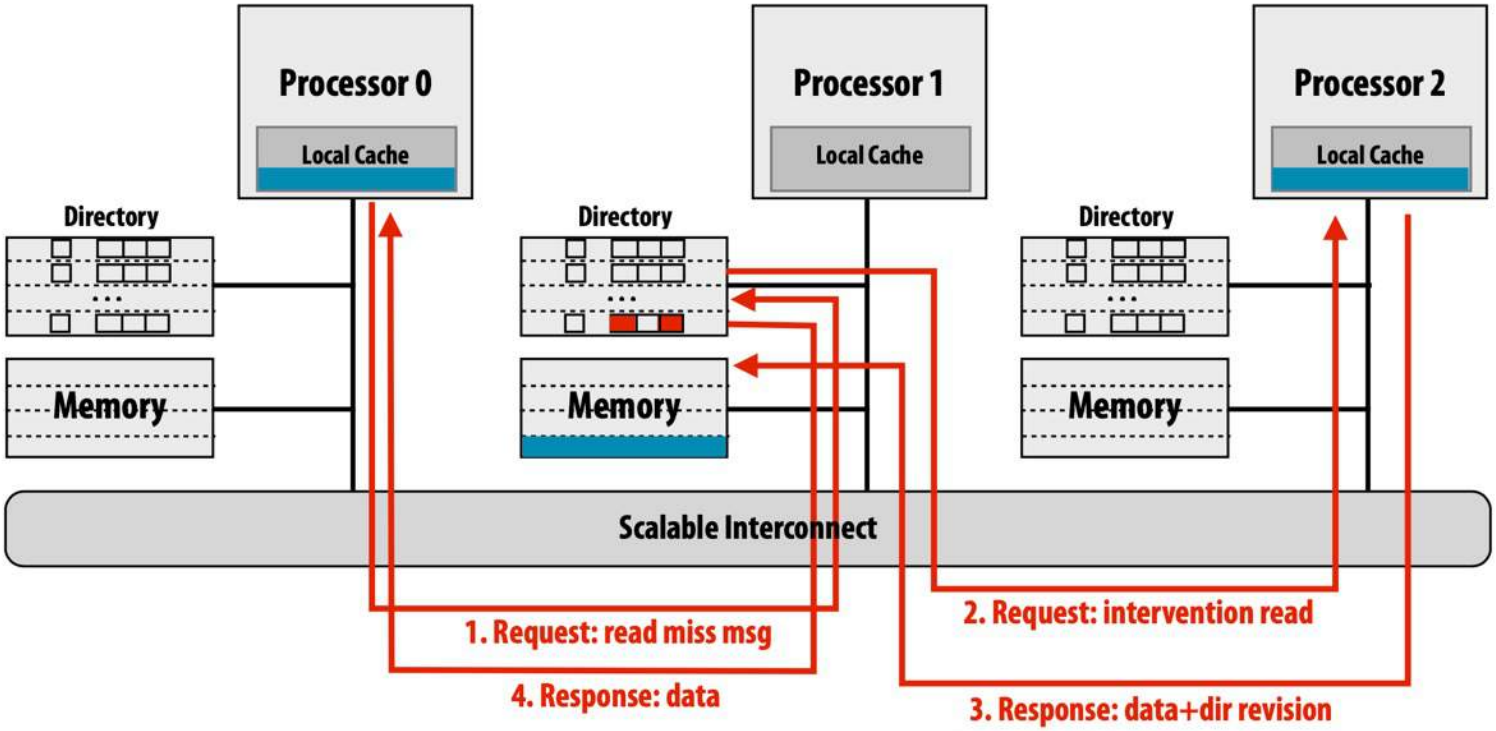


Five network transactions in total

Four of them are sequential (transaction 4 & 5 can parallel)

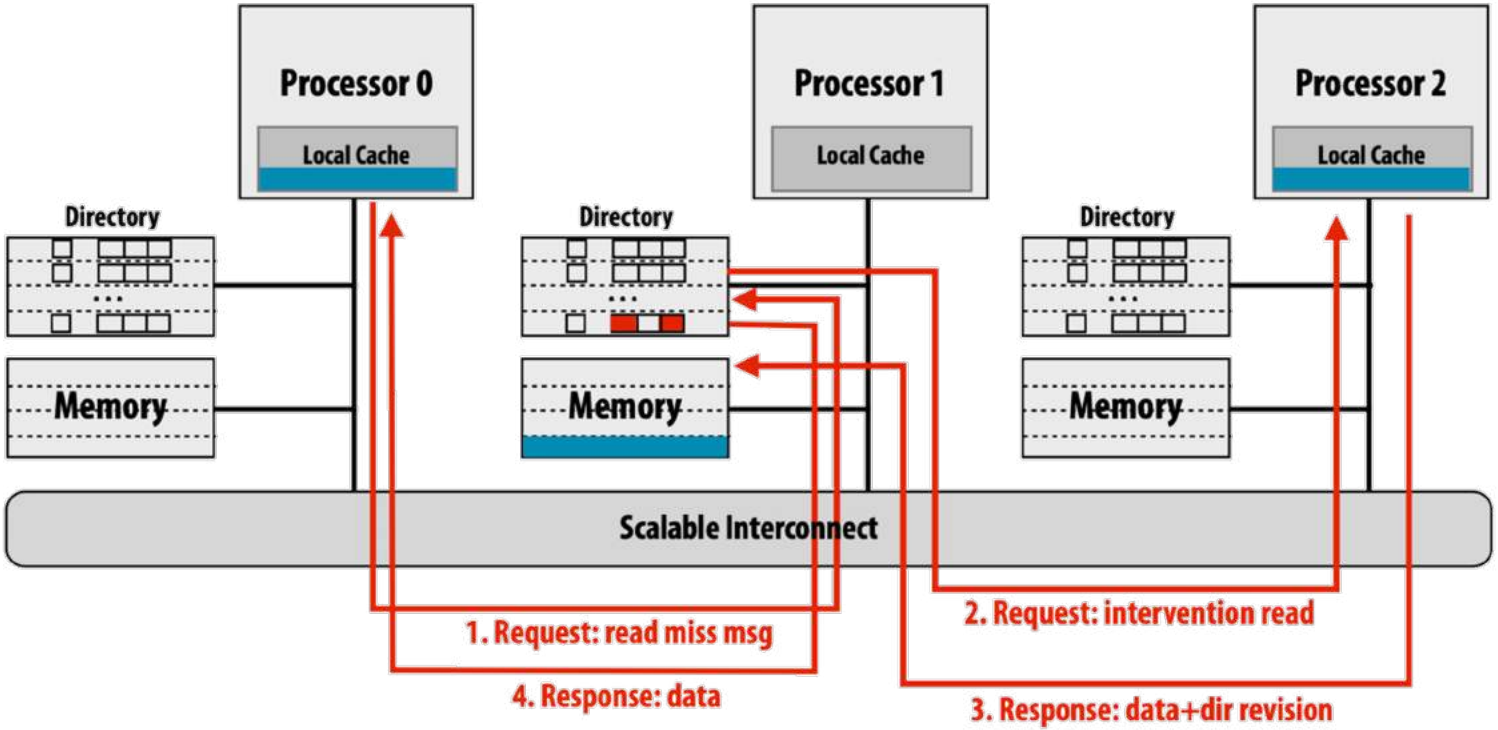
Intervention Forwarding[干预转发]

Read from main memory by P0 of the blue line: line is dirty (contained in P2's cache)



Total 4 transactions are needed

Intervention Forwarding (cont.)

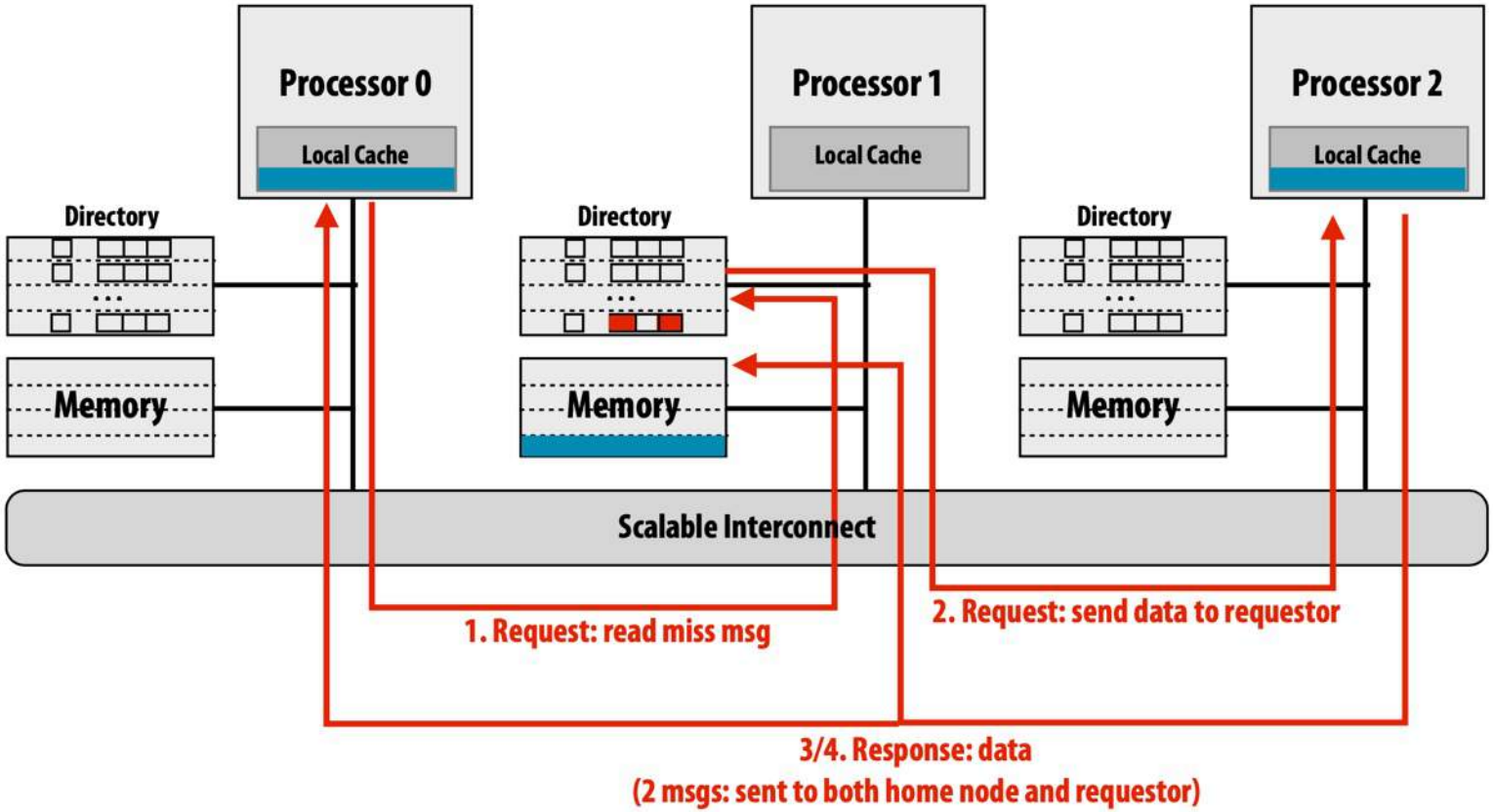


1. Requests to read miss message on home node (P1)
2. Home node requests data from owner node (P2)
3. Owning node response
4. Home node updates directory, responds to requesting node with requested data

All transactions are sequential, can they be parallel?

Request Forwarding[请求转发]

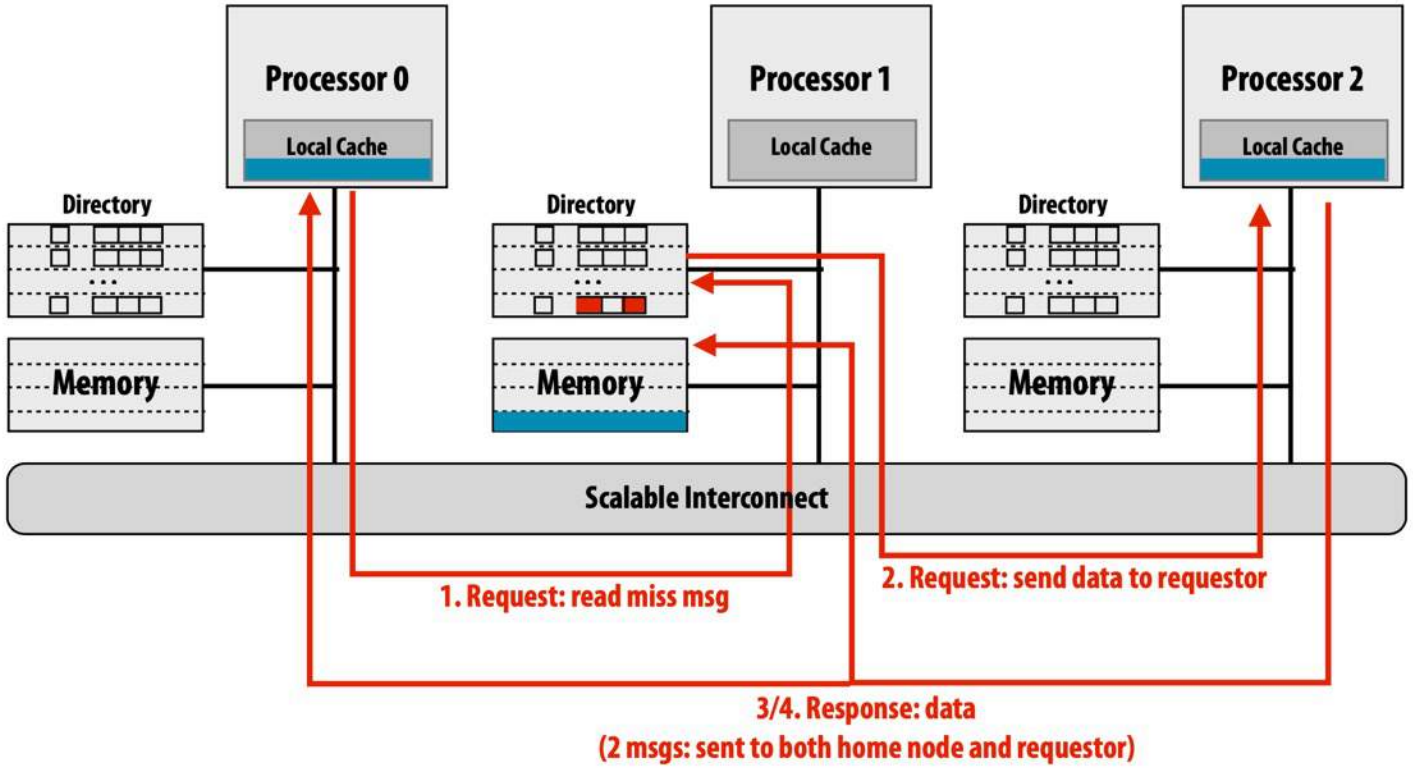
Read from main memory by P0 of the blue line: line is dirty (contained in P2's cache)



Only 3 transactions are in serial

Transaction 3 & 4 can be parallel

Request Forwarding (cont.)

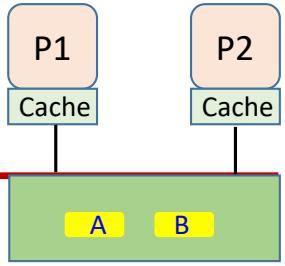


1. Requests to read miss message on home node (P1)
2. Home node sends target data to owner
3. Owning node responses data to the home node
4. Owning node responses data to the requesting node

Summary of Directory-base Coherence

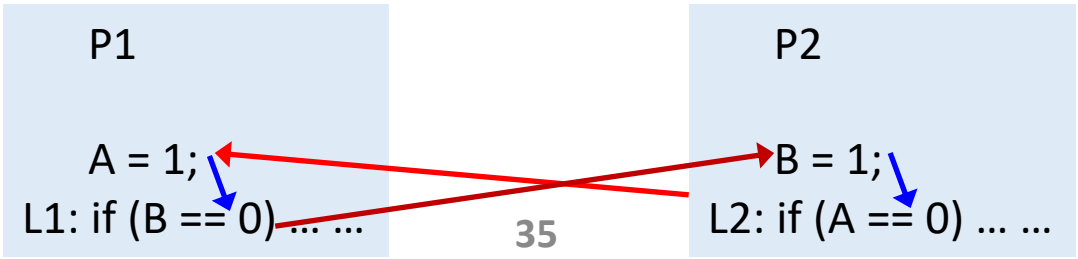
- Primary observation: broadcast doesn't scale, but we don't need to broadcast to ensure coherence because often the number of caches containing a copy of a line is small
- Instead of snooping, just store the list of sharers in a directory and check the list when necessary
- One challenge on storage[存储]
  - Use hierarchies of processors or larger cache size
  - Limited pointer schemes: exploit fact that most processors not sharing line
  - Sparse directory schemes: exploit fact that most lines not in cache
- Another challenge on communication[通信]
  - Reduce messages sent (traffic) and parallelize trans (latency)

Example



- Assume that
  - Processes *P1* and *P2* are running on two different processors
  - Locations *A* and *B* are originally cached by both processors with the initial value of *0*
- If writes always take <u>immediate</u> effect and are immediately seen by other processors
  - Then impossible for both *IF* to be true

    Reaching the IF means that either A or B must have been assigned the value 1 (i.e., IF is false)
- If write invalidate can be <u>delayed</u>, and the processor is allowed to continue during this delay
  - Then possible to that *P1* and *P2* haven't seen the invalidations before they attempt to read the values



Coherence vs. Consistency[对比]

- **Cache coherence** defines requirements for the observed behavior of reads and writes to the same memory location
  - Goal: to ensure that the memory system in a parallel computer behaves as if the caches were not there
    - A system without caches would have no need for cache coherence
  - Write value will be seen if sufficiently separated in time
- **Memory consistency** defines the behavior of reads and writes to different locations
  - The allowed behavior of memory should be specified whether or not caches are present
  - Coherence only guarantees that writes to address X will eventually propagate to other processors
  - Consistency deals with when writes to X propagate to other processors, relative to reads and writes to other addresses

Memory Consistency[内存一致性]

- Memory consistency specifies the ordering behaviors
  - What ordering behavior should be allowed?
  - Under what conditions?

- Example: a program running two threads, where A and B are initially both 0. What this program can output?
  - 01: (1)(2)(3)(4) or (3)(4)(1)(2)
  - 11: (1)(3)(2)(4) or (1)(3)(4)(2)
  - 00: intuitively, it shouldn't be possible

Thread 1

```
(1) A = 1
(2) print(B)
```

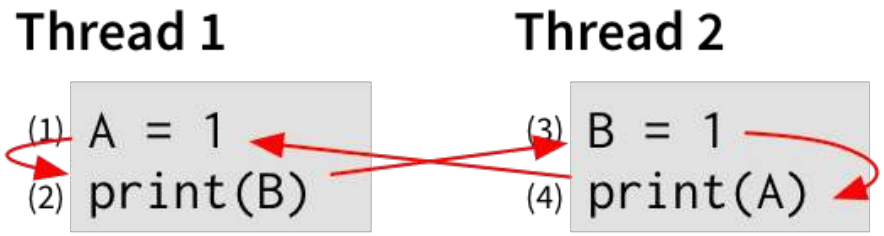
Thread 2

```
(3) B = 1
(4) print(A)
```


https://www.cs.utexas.edu/~bornholt/post/memory-models.html

The Example

- x → y: x must happen before y
  - (2) to print 0: (2) → (3)
  - (4) to print 0: (4) → (1)
  - If each thread's events happen in order
    - (1) → (2)
    - (3) → (4)
- Start from (1), follow the edges
  - (1) → (2) → (3) → (4) → (1)
  - (1) must happen before itself ???



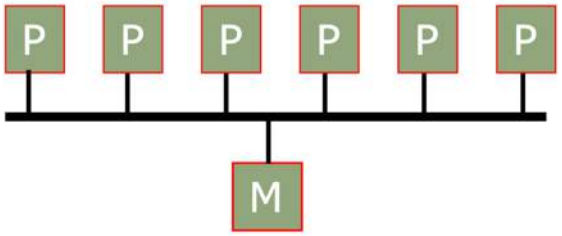
https://www.cs.utexas.edu/~bornholt/post/memory-models.html

Memory Operation Ordering[访存先后]

- A program defines a sequence of loads and stores (this is the "program order" of the loads and stores)[程序顺序]
- Four types of memory operation orderings[4类顺序]
  - **W→R**: write to X must commit before subsequent read from Y
    - When a write comes before a read in program order, the write must commit (its results are visible) by the time the read occurs
  - **R→R**: read from X must commit before subsequent read from Y
  - **R→W**: read to X must commit before subsequent write to Y
  - **W→W**: write to X must commit before subsequent write to Y
- A sequentially consistent memory system maintains all four memory operation orderings[顺序一致]
- Certain orderings can be violated ???[违背一些顺序？]

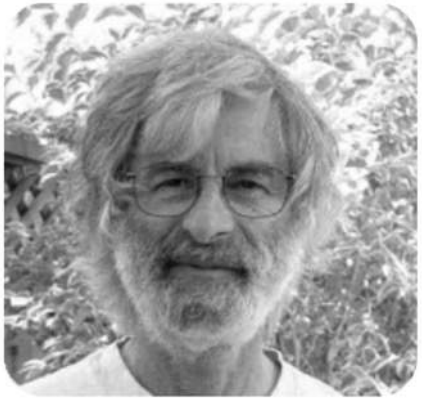
Sequential Consistency[顺序一致性]

- The most straightforward model for memory consistency
  - Intuitive idea: multiple threads running in parallel are manipulating a single main memory, and so everything must happen in order
    - But what order?
  - Intuitive order: the events in a single thread happen in the order in which they were written[程序顺序]
    - Intuitive to programmers
- **Sequential consistency** requires that the result of any execution be the same as though
  - Memory accesses executed by each proc. were kept in order
  - The accesses among different processors were arbitrarily interleaved



Sequential Consistency (cont.)

- Sequential consistency (SC)
  - Formalized by Leslie Lamport in 1979
  - "A system is **sequentially consistent** if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in the order specified by the program" [看起来像。。。]
  - Defining SC is one of the many achievements that earned Lamport the Turing award in 2013



LESLIE LAMPORT DL

United States – 2013

CITATION

For fundamental contributions to the theory and practice of distributed and concurrent systems, notably the invention of concepts such as causality and logical clocks, safety and liveness, replicated state machines, and sequential consistency.

Time, Clocks, and the Ordering of Events in a Distributed System

Leslie Lamport
Massachusetts Computer Associates, Inc.

https://lamport.azurewebsites.net/pubs/time-clocks.pdf

The Examples

- With SC,
- Example-1:
  - Must delay the read of *A* or *B* (*A == 0* or B == 0) until the previous write has completed (*B = 1* or *A = 1*)
  - Cannot simply place the write in a buffer and continue with the read

P1	P2
A = 1; L1: if (B == 0) … …	B = 1; L2: if (A == 0) … …

- Example-2:
  - *print(B)*/*print(A)* cannot happen before *A = 1*/*B = 1*
    - 00 cannot be printed

Thread 1	Thread 2
(1) A = 1 (2) print(B)	(3) B = 1 (4) print(A)

Memory Consistency Model[一致性模型]

- **Memory consistency model** (or just "**memory model**") defines the allowed orderings of multiple threads on a multiprocessor
  - SC is one such model
    - E.g., orderings that print 01/11 are allowed, but not 00
- A memory consistency model is a contract between the hw and sw
  - The hw promises to only reorder operations in ways allowed by the model[硬件承诺]
  - In return, the sw acknowledges that all such reorderings are possible and that is needs to account for them[软件认可]

Thread 1	Thread 2
(1) A = 1	(3) B = 1
(2) print(B)	(4) print(A)

https://www.cs.utexas.edu/~bornholt/post/memory-models.html