1. Memory technology & Hierarchy

Back to caching...

Advances in Computer Architecture

Andy D. Pimentel

Caches in a multi-processor context

Dealing with concurrent updates

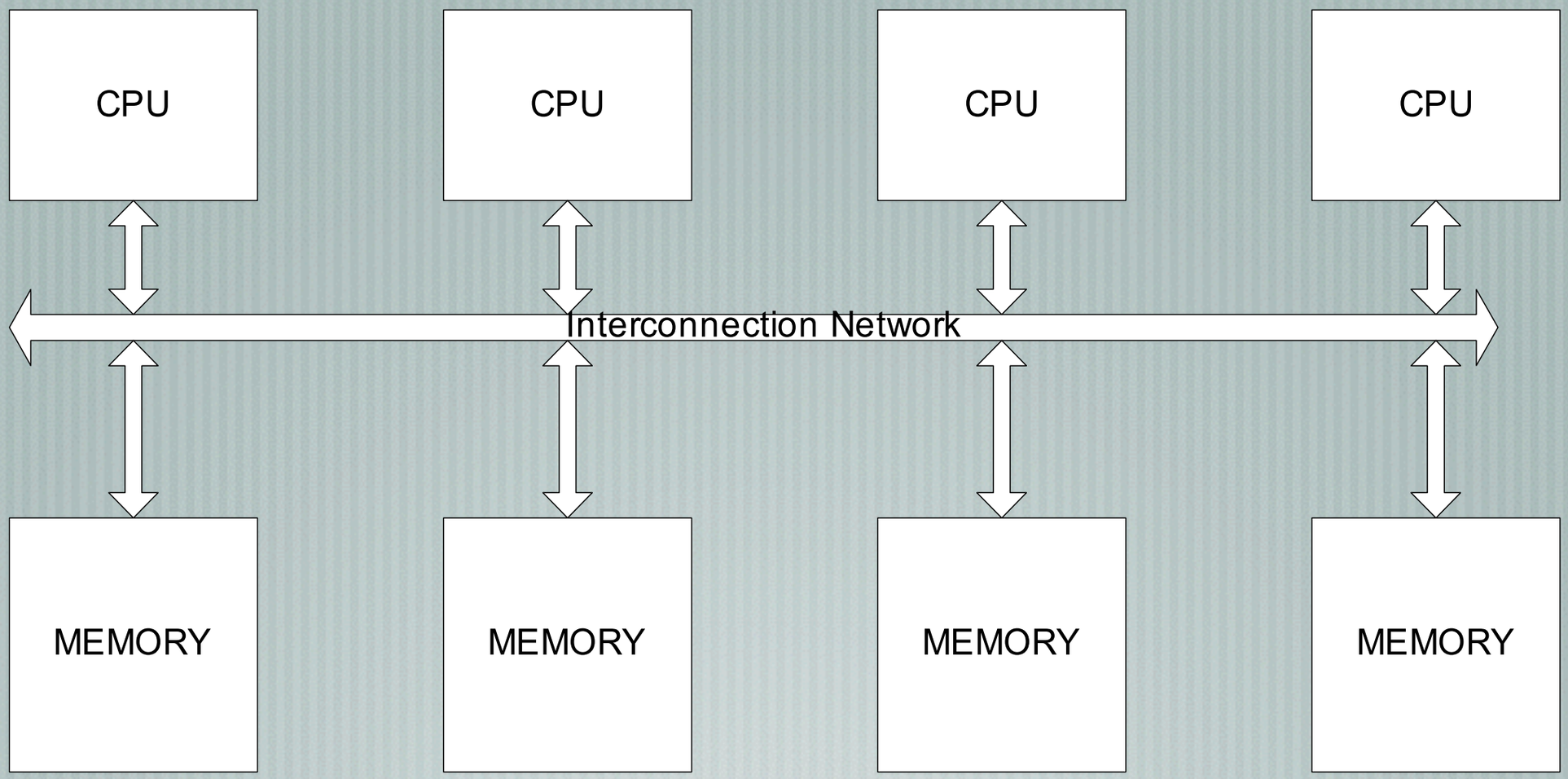
Multiprocessor architecture

- In a multiprocessor (MP) there will typically be multiple memory banks as well as processors
- There will also be an interconnection network
  - the simplest form is a **bus**, however this does not scale
  - scalability: **point-to-point networks with some topology**
- We deal with communications later in the course,
  for now we assume some mechanism for connecting processors and memories

MP Memory architectures

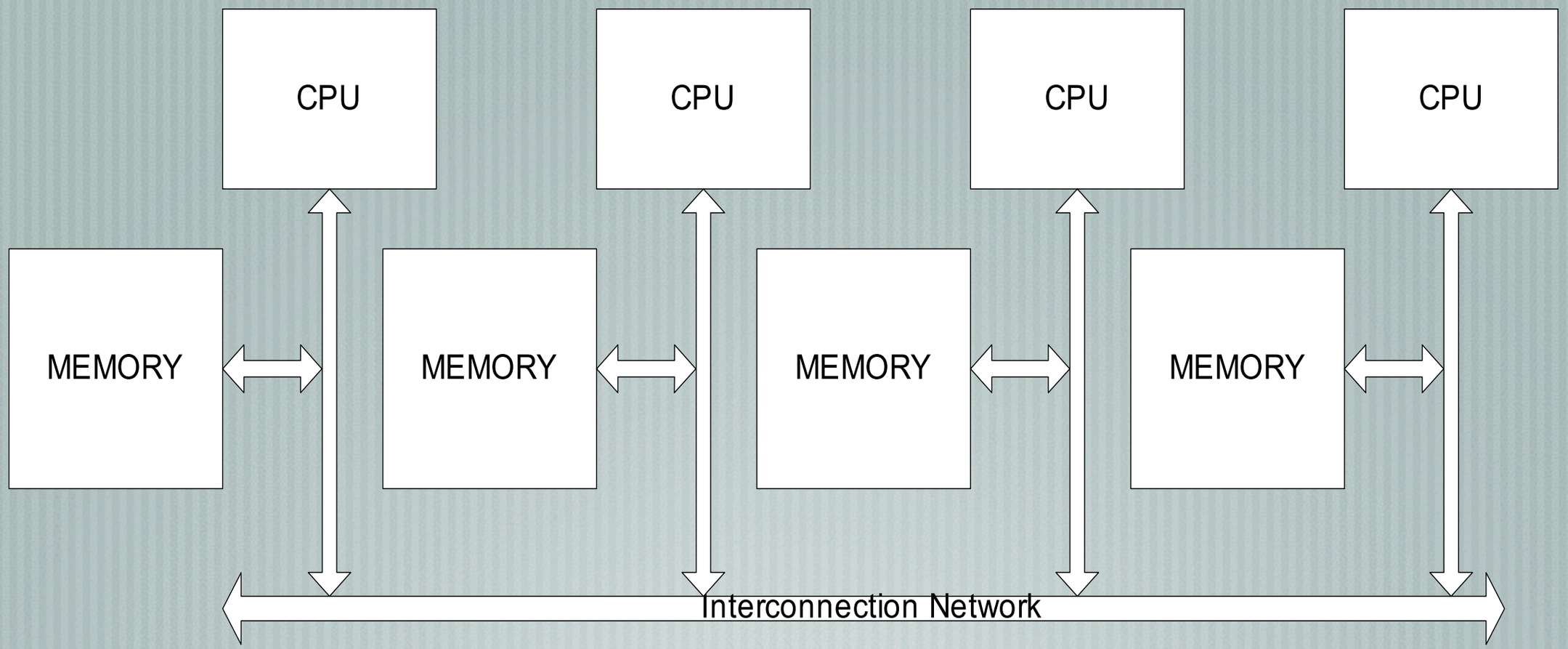
- Distributed Memory Architecture
  - Private memories associated with a single processor only
  - All communication uses message passing between processors
- “Shared Memory” Architecture
  - actually, shared **address space**
    = all processors “see” a single, big memory via the same addresses
- Shared Memory Architectures often implemented in a **distributed** fashion
  - Uniform Memory Access (UMA)
  - Non-Uniform Memory Access (NUMA)
  - Cache-Only Memory Architecture (COMA)

MP Memory architecture - UMA



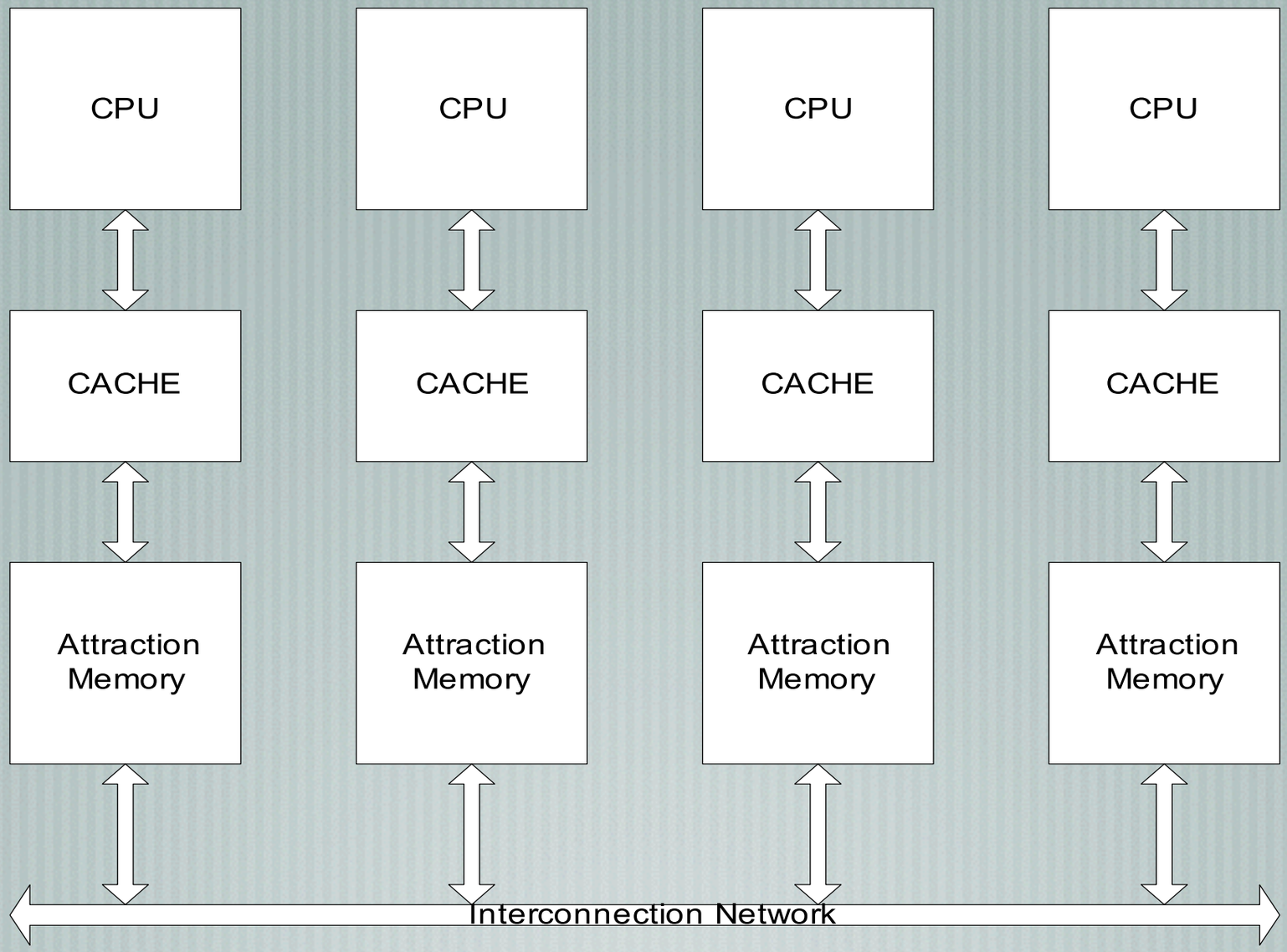
Uniform Memory Architecture

MP Memory architecture - NUMA



Non-Uniform Memory Architecture

MP Memory architecture - COMA



Cache-Only Memory Architecture

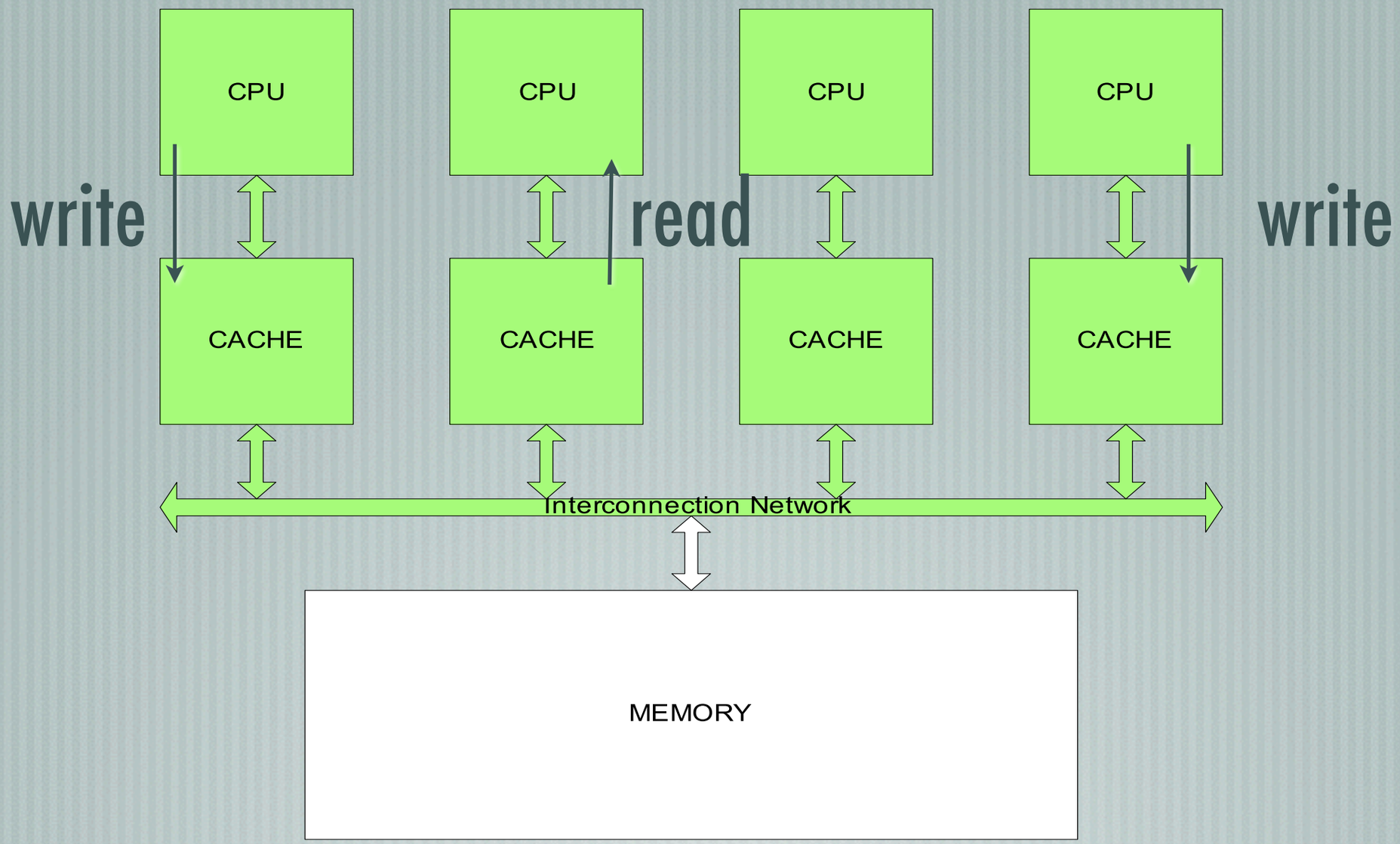
Why use caches in a MP system

- Caches can reduce traffic on the interconnection network (important because communication doesn't scale)
- Fetching cache lines is more efficient than fetching single words from DRAM memory

  as long as locality is exploited

- Problems arise as several copies of the same memory block may be cached in different processors, i.e. multiple processors sharing a storage

Shared storage



Multiple caches, one storage = what to do?

General issues of shared memory

- Sharing storage (RAM components) between processors adds hw complexity, esp. with cache implementations
- Issues that need solving:
  - **Memory consistency**: what the programmer **expects** from updates to memory, i.e. **when** writes to memory from other processors become visible
  - **Coherency protocol**: when MP consistency is desired, **how** do writes by one processor become visible to other processors?
  - **False sharing**: multiple processors write to different memory addresses with the same cache line address - causes extra coherency traffic

Memory consistency

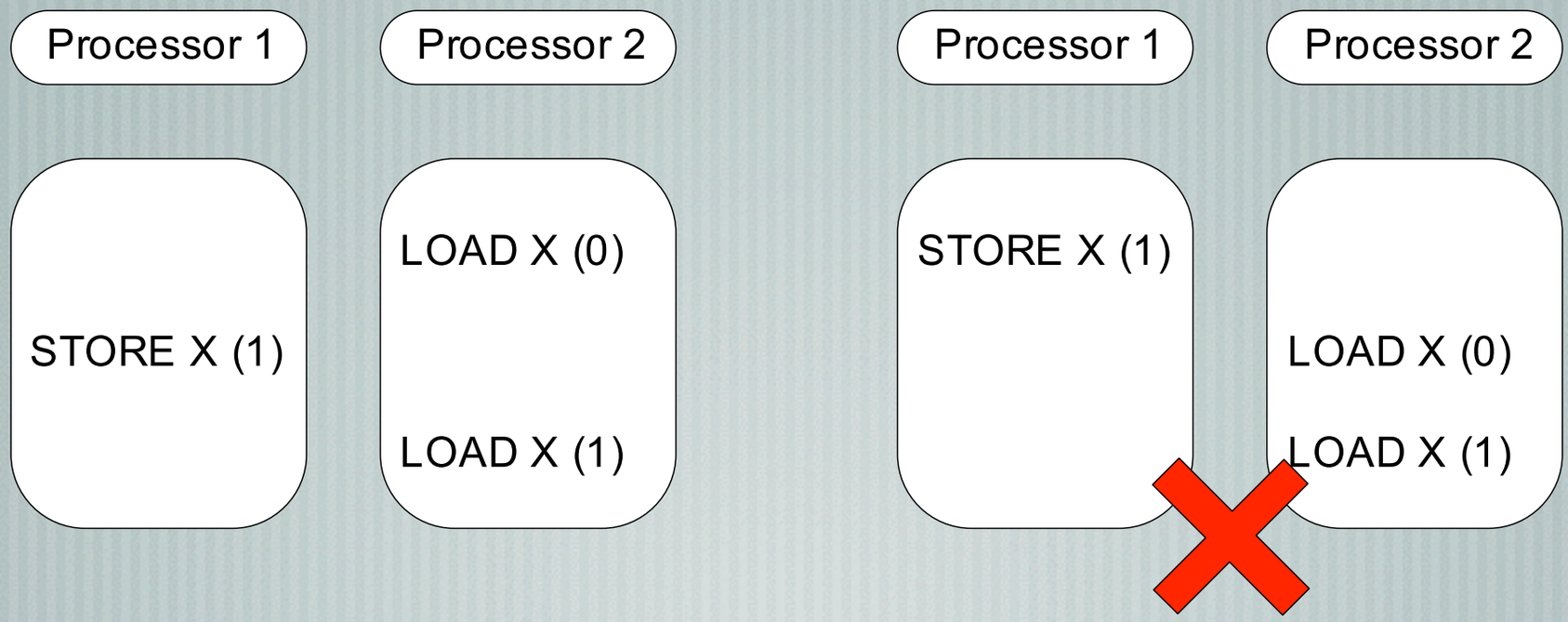
- The **memory consistency model** establishes a contract between the memory system and the programmer when there can be concurrent reads and writes to a shared storage
  - E.g. in a race condition, memory will need to consistently reflect a state and this is the contract with the programmer
- With no caching there is no problem - all races are resolved at the storage and programs will see a consistent state, i.e. every processor will see the same sequence of reads and writes
- Where multiple copies of the cache line are updated by different processors, we have to decide what is allowed
  - Chosen model has an impact on the efficiency of the cache coherency protocol; Conversely, the coherency protocol must be defined to manage a given consistency model

Consistency models

- Consistency models
  - **define the behavior and correctness of a program**
  - **impose ordering constraints on reads and writes**
  - **balance programming complexity and performance**
  - **In effect they define a contract on what to expect between a concurrent programmer and the memory**
- They include
  - **strict consistency** - where all loads & stores are strictly ordered
  - **sequential consistency** - where all processors see the same write order
  - and various **relaxed** or **weakly consistent** models

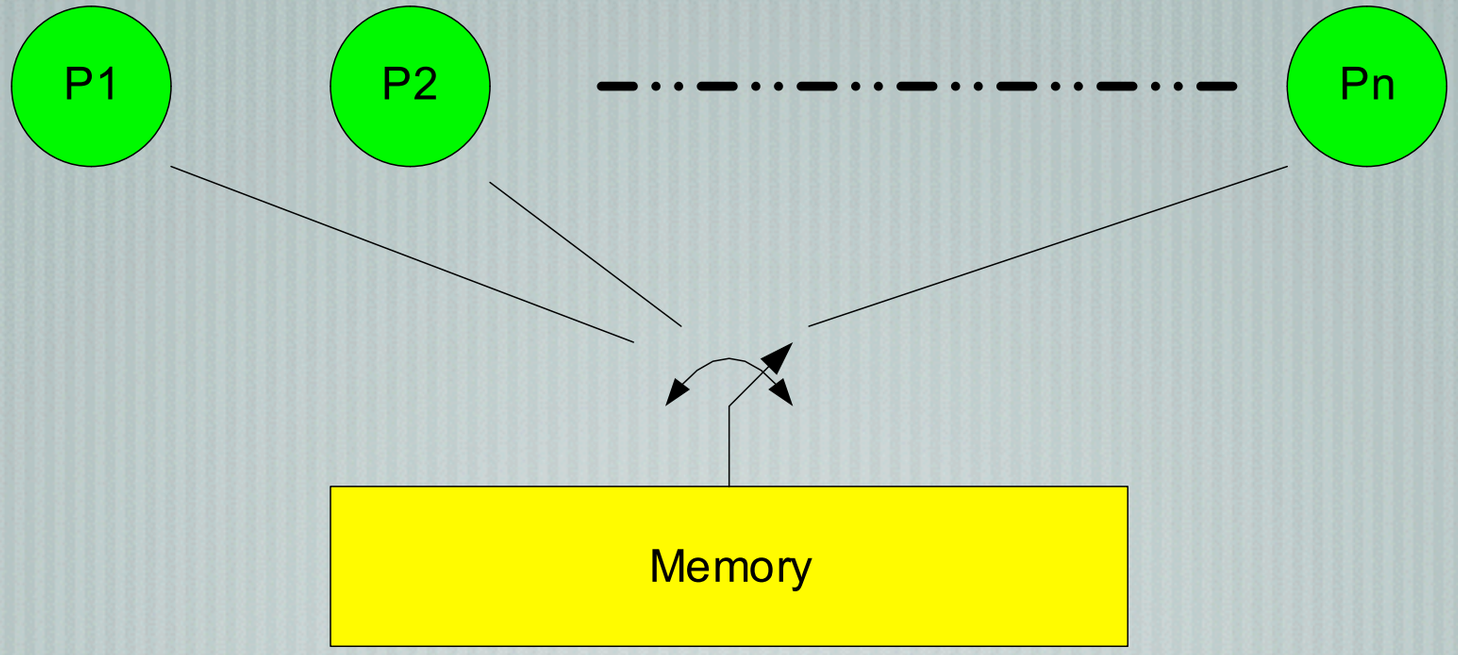
Strict consistency

- Always load the value updated by the most recent store
  - **Too strict, normally unnecessary**
  - **Heavy coherence overhead**



Sequential consistency

- "Random" switch between memory operations



Sequential consistency

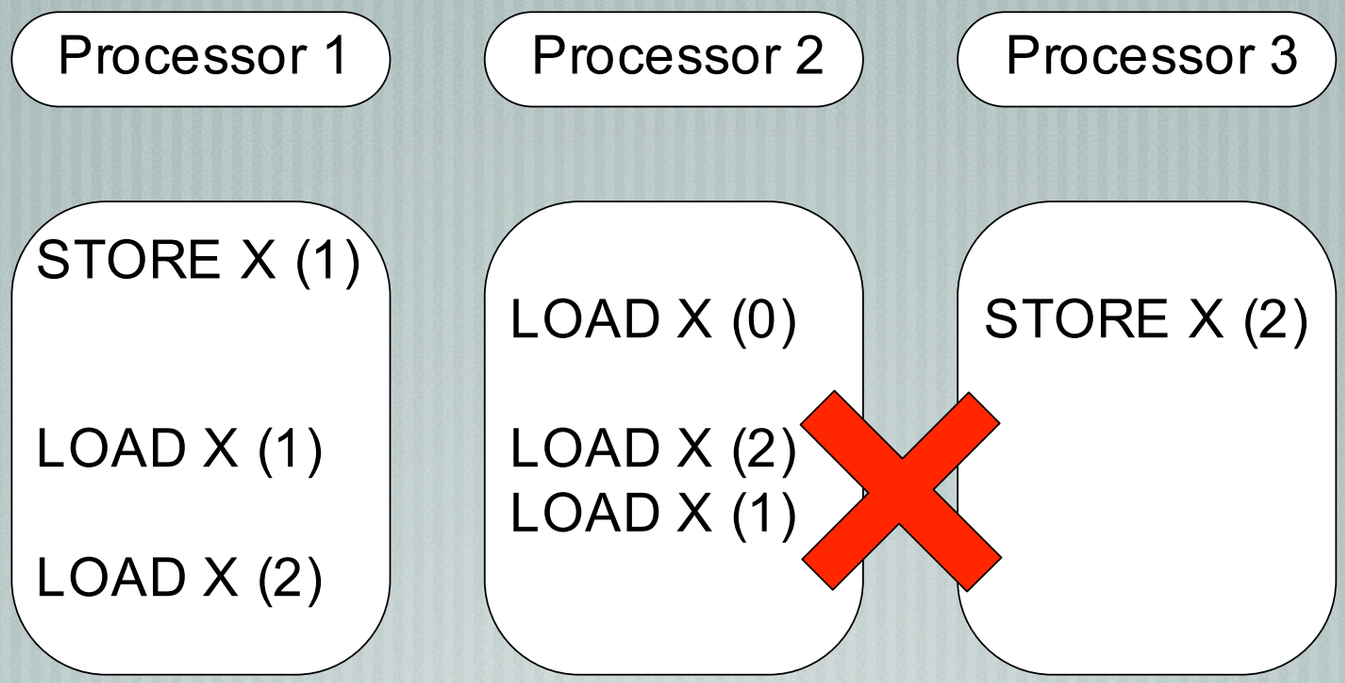
- Definition:
  - A multiprocessor is sequentially consistent
    if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program
- Property:
  - All the nodes in the system observe the write operations on the memory in the same order
  - Locally, program order is enforced.
  - **Intuitive to programmers**
  - **Still a heavy burden in coherence traffic**

Sequential consistency

- Every processor observes all stores in the same order

Sequential consistency

Every processor observes all stores in the same order



Sequential consistency

Atomic and strongly ordered memory accesses

```
Processor 1:                Processor 2:

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

Weak consistency

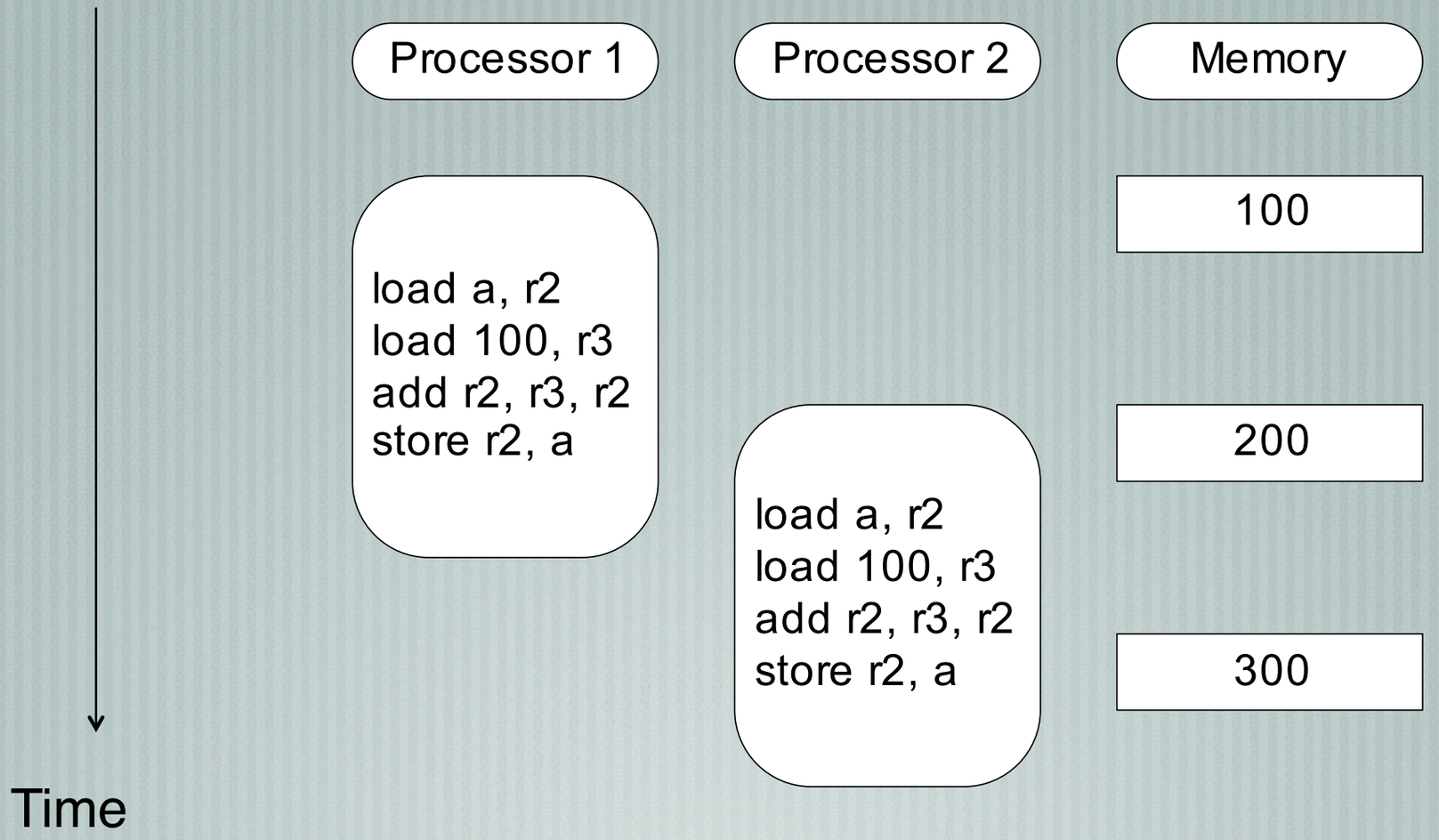
- Typically utilizes explicit synchronization operations
- For example, semi-explicit consistency with **fences**:
  - All previous synchronizations must be performed before a memory read
  - All memory writes must be performed before a synchronization operation

Cache coherency

- Used to maintain consistency according to a given model
- If multiple processors cache the same block and one processor writes to it then the two copies of the same block of memory will contain different data
- Protocols are used to maintain coherency of the cached data and to manage writes to shared cache lines
  - **typically when one processor writes to a cache line all other copies of that lines must be made invalid**

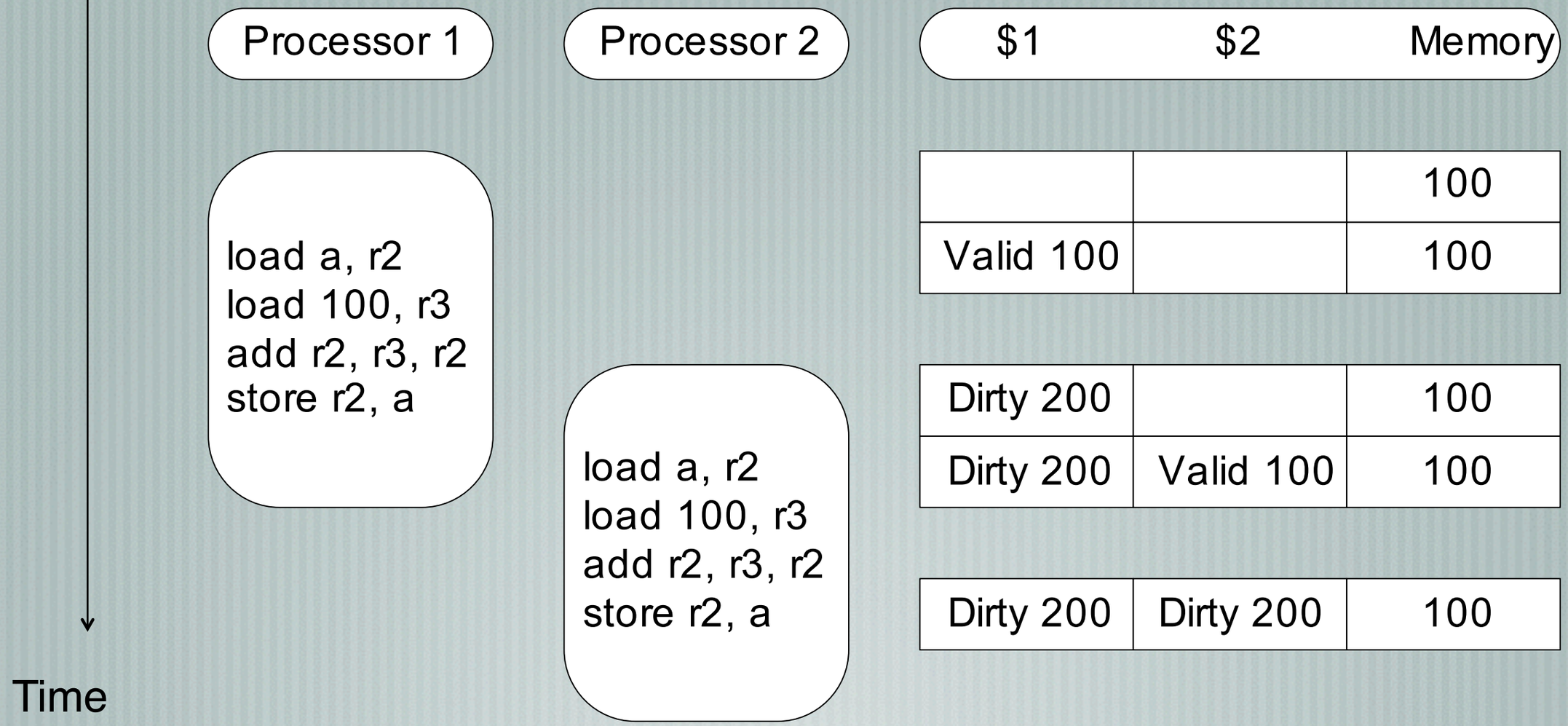
Cache coherency

Without cache:



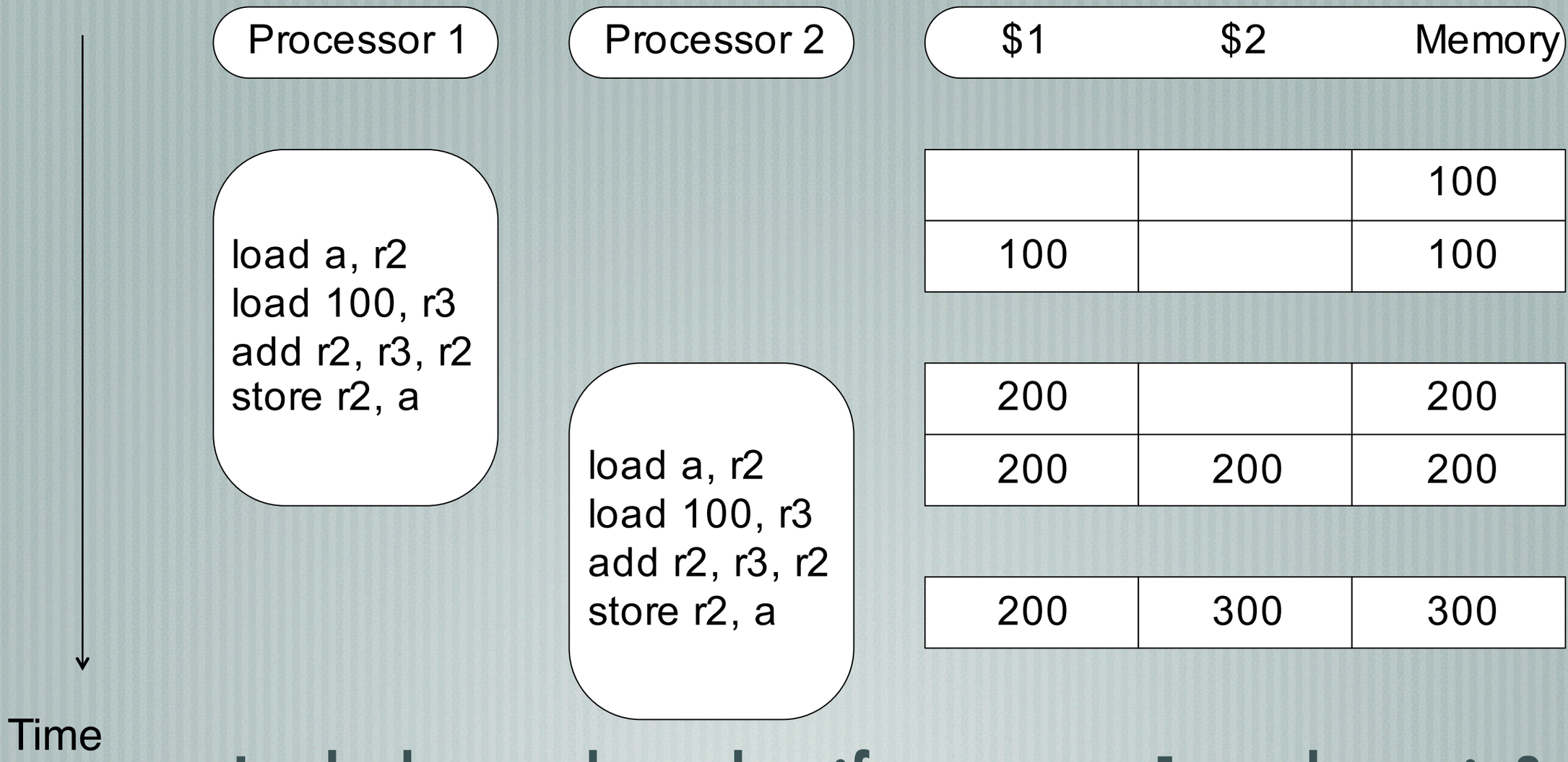
Cache coherency - the problem

Using a copy-back protocol:



Cache coherency - the problem

Write through, maybe?

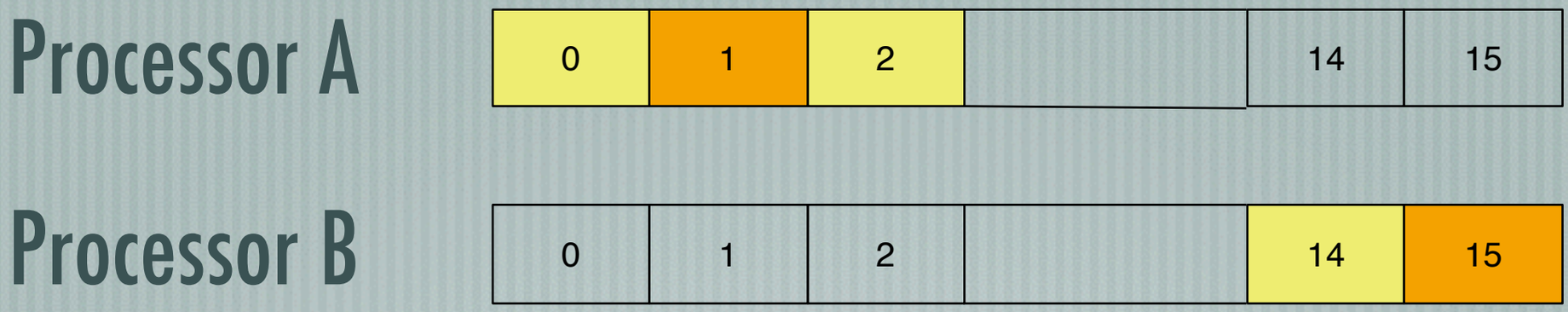


Looks better, but what if processor 1 reads again?

False sharing

- Generally the larger the cache line the greater the effect of locality
  i.e fewer accesses to main memory
- However, with a very large cache line, if the same memory block is cached in two processors and the copies must be consistent:
  - when a cache line is written by one processor
  - the other processor must mark that cache line as invalid regardless of whether the particular word written is required by the second processor
- This is called **false sharing**

Example false sharing



- Processors A and B access and share the same cache line
  - A reads words 0, 1 and 2 and writes only 1
  - B reads words 14 and 15 and writes only 15
- each write invalidates the other's copy of the cache line. This is false sharing as neither needs the data written by the other, and yet this may impact the hit rates.
- Generally cache hit ratios increase with cache line size
  but in a multi-processor system, false sharing generates more misses for larger line sizes.

Coherency protocols

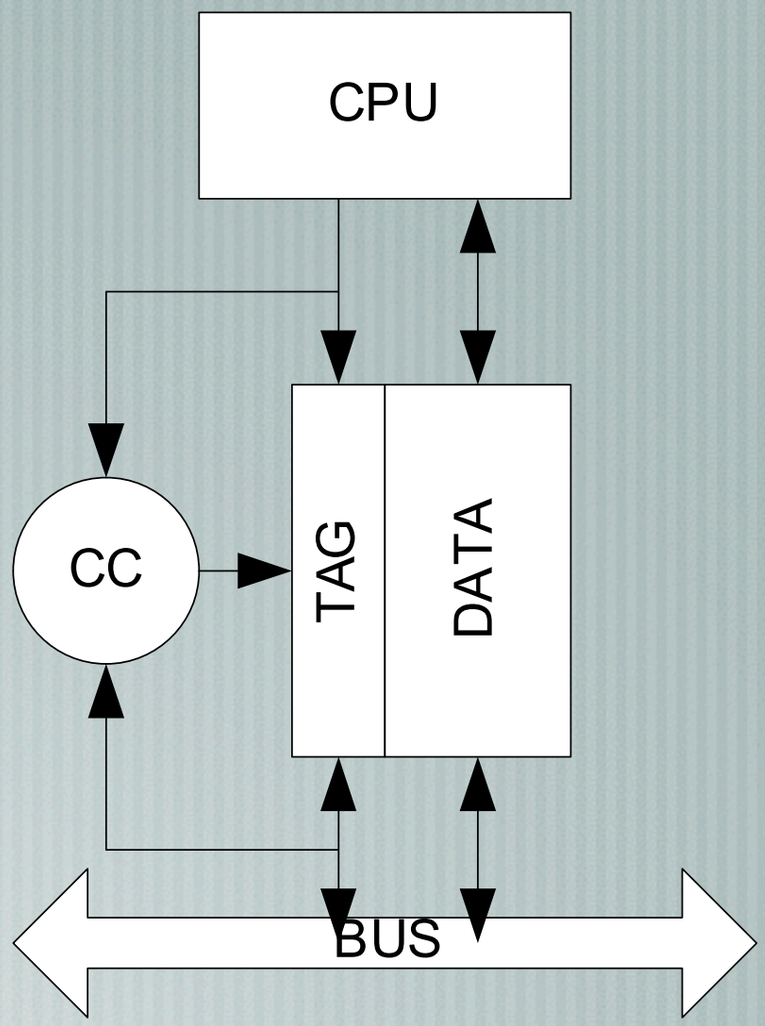
Coherency protocols

- To maintain coherence on different copies of the same block of memory, there must be an exchange of information when the state of that data changes in any one of the caches
- This uses a protocol and a state associated with every cache line
  - **the cache line or directory maintains the state information**
  - **in addition information may also be required about which other processors/cache controllers share a line**
- The simplest example of state associated with a cache line would be: {valid, invalid}

Coherency protocols

Cache Coherence Controller

- Handles CPU requests
- Handles bus/network events



Coherency protocols

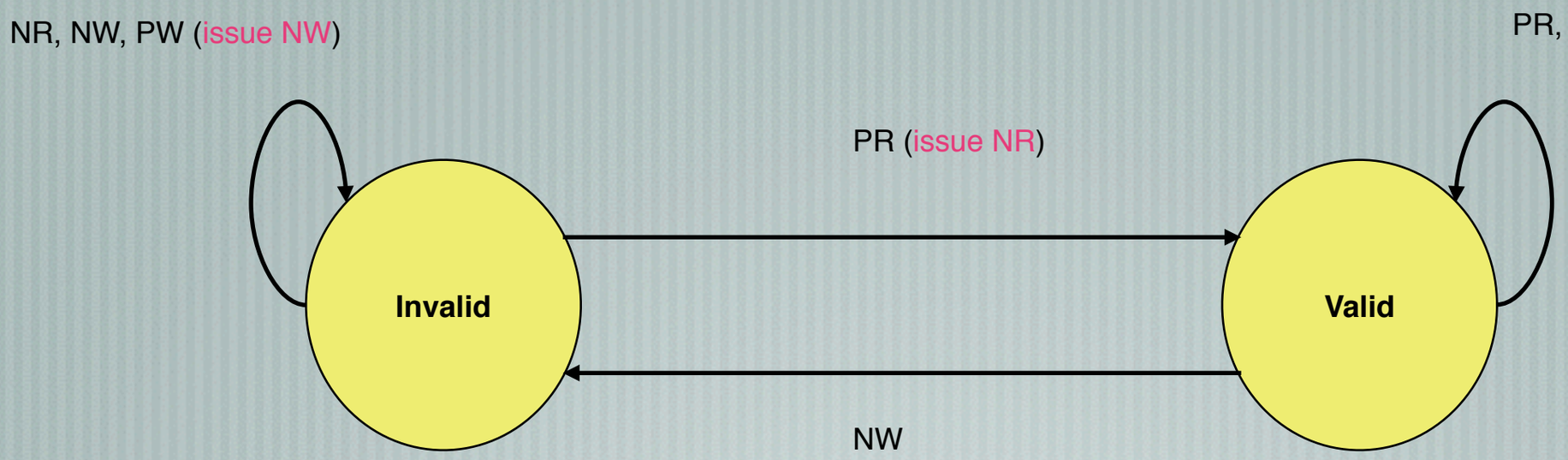
- There are two classes of protocols,
  the choice is determined largely by the communication medium
  - **Snoopy protocols** - where each processor can "see" every memory transaction and can maintain state information on shared cache lines (e.g. bus)
    - here, every cache lines maintains information on the state of the data stored
  - **Directory protocols** – where information about a block of memory is held in just one place, a directory, that must identify the nodes or caches that share the memory block
    - a write is notified to the directory, which in turn can initiate invalidations to only those caches sharing the line

Simple coherency protocol

- Simplest protocol uses just a valid bit
  - allows cached reads and supports write-through and write-around cache policies
  - every write updates memory and also invalidates all other copies of the cached line
- To allow coherent cached writes a dirty bit is used
  - indicates that the cache line has been written to and memory is not updated, i.e. a copy back policy
  - the processor may now write multiple times to the same cache line without incurring new memory transactions
  - loads from a dirty line forces a fetch from other caches / memory

Example protocols (snoopy)

Simplest protocol supports write through and write around

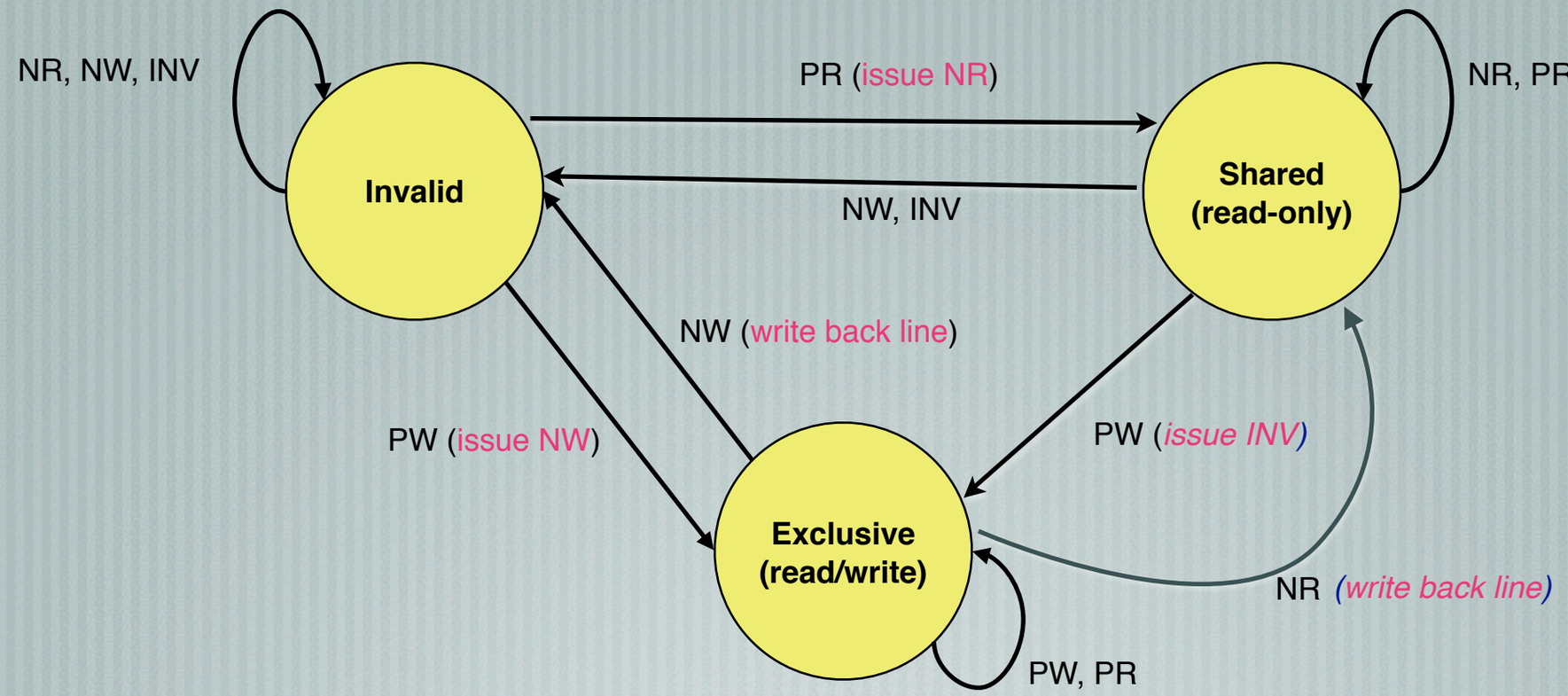


PR - processor read
PW - processor write

NR - network read (observed bus read)
NW - network write (observed bus write)

Example protocols (snoopy)

Write-invalidate, supports copy back



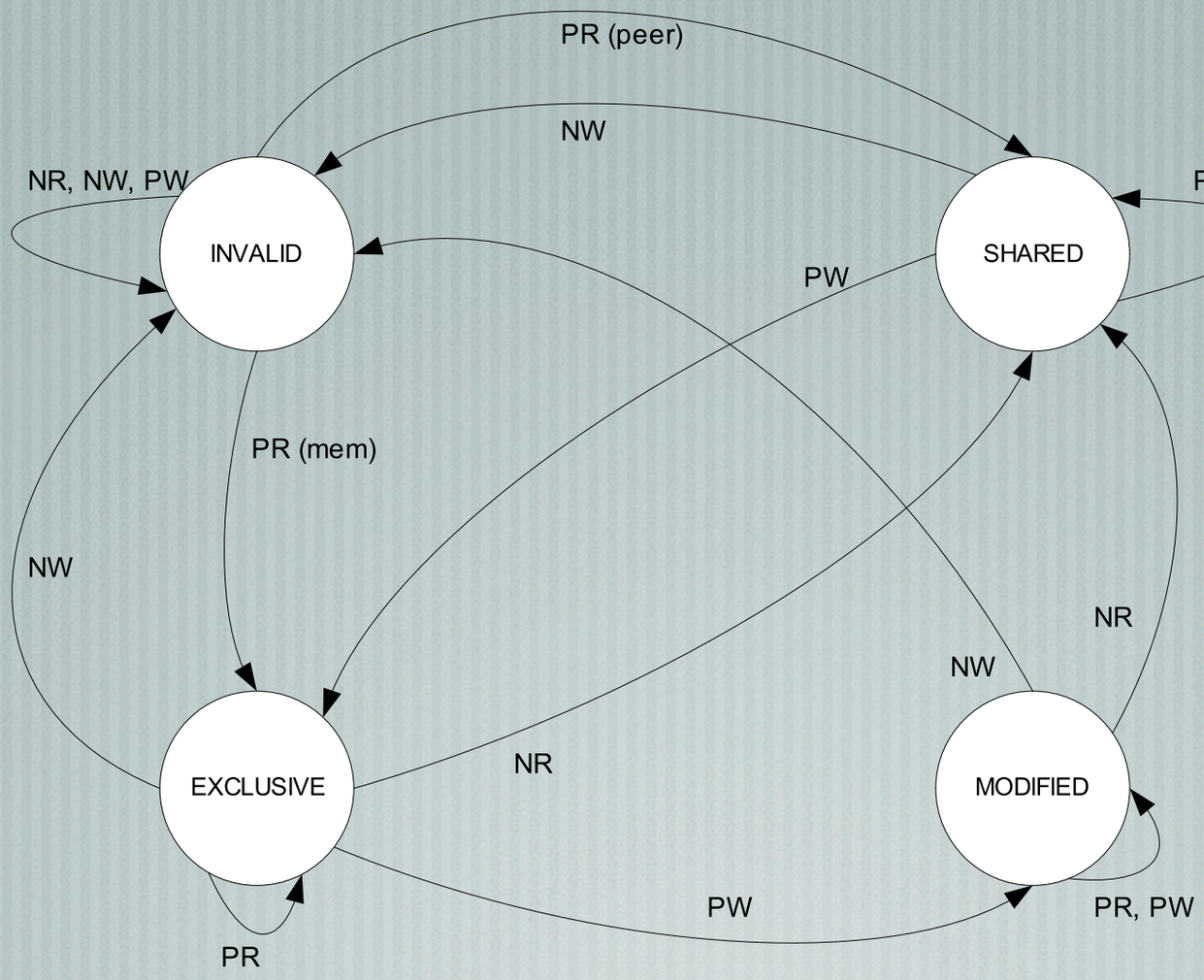
PR - processor read
PW - processor write
INV - invalidate

NR - network read (observed bus read)
NW - network write (observed bus write)

This is a simple, common protocol in bus-based SMPs

Example protocols (snoopy)

MESI: write back and write around



PR - Processor Read

NR - Network Read (Observed Bus Read)

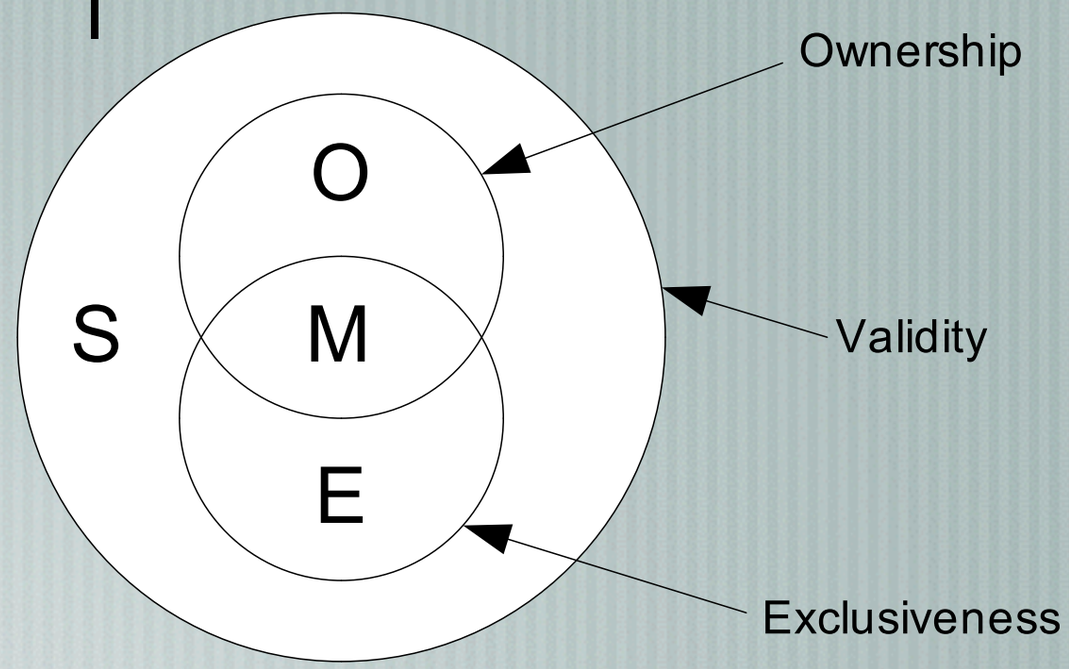
PW - Processor Write

NW - Network Write (Observed Bus Write)

Example protocols

MOESI

- Modified: cache holds dirty data
- Owned: cache owns the dirty data
- Exclusive: cache holds clean data, which is the only one in the caches
- Shared: clean data
- Invalid: empty



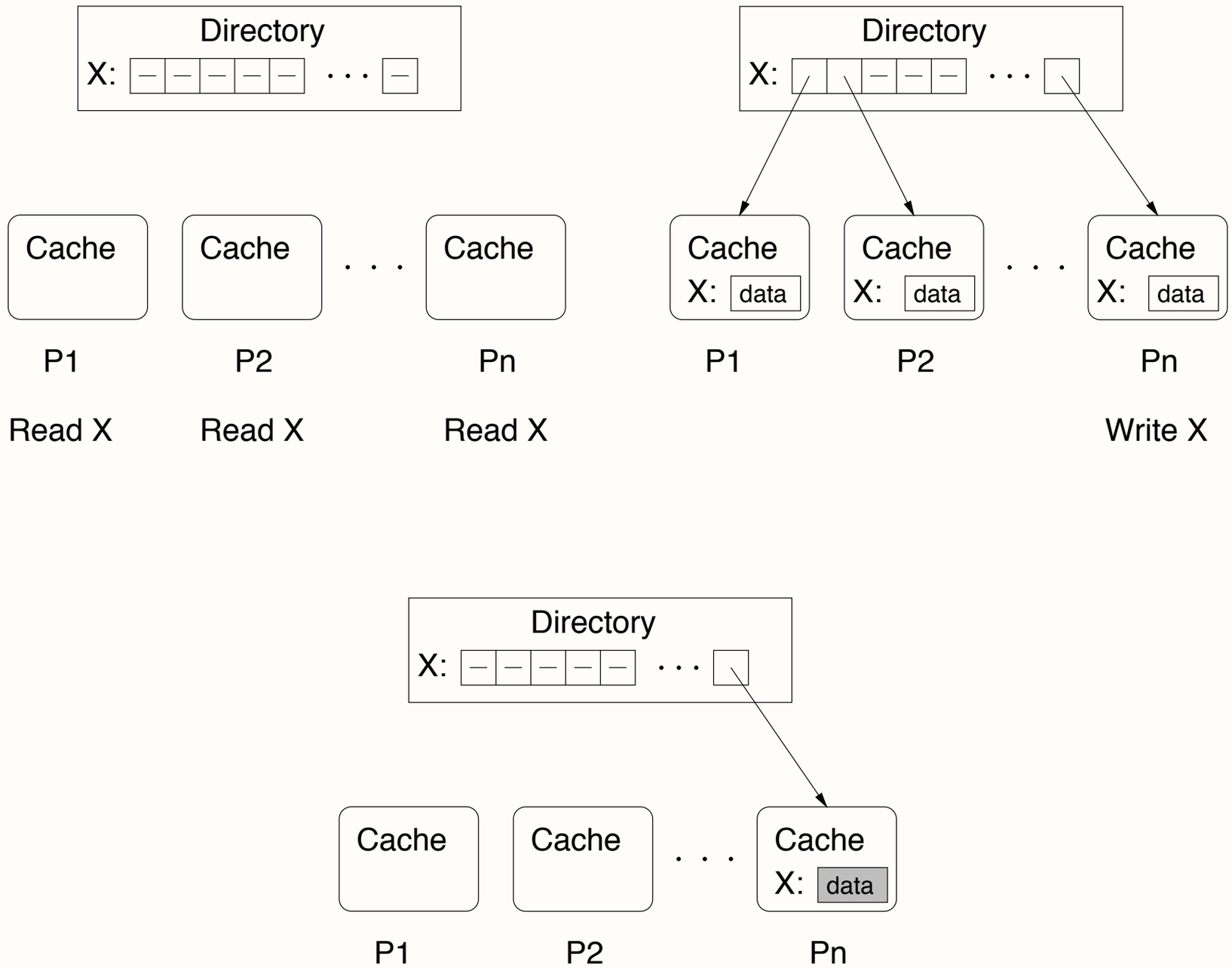
Snoopy protocols

- A bus allows snoopy protocols to be implemented efficiently
  - **A bus is a broadcast medium and hence all processors can see all transactions on the bus**
    - In networks this is not the case as networks are divergent, i.e. point-to-point with branching
- A snoopy cache controller monitors all bus transactions and maintains cache line state for every line it has cached
  - i.e. a write on the bus to a cached line would generate an invalidate signal locally for any processor caching that line

Directory-based protocols

- **Busses provide a simple solution to the cache coherence problem but they do not scale**
  - the concept of all processors monitoring all transactions of all other processors is in itself is not scalable
- **Scalable multiprocessors require a network solution to be adopted**
  - networks are point-to-point communication mediums and generally do not support broadcast
  - even if they do support broadcast, if used by all processors, this would very quickly saturate the network
- The solution is to use a protocol based on a directory that maintains state on cache lines and a mechanism for determining sharing
  - in this way the communication can be a multi-cast involving only the nodes/caches sharing a given block of data

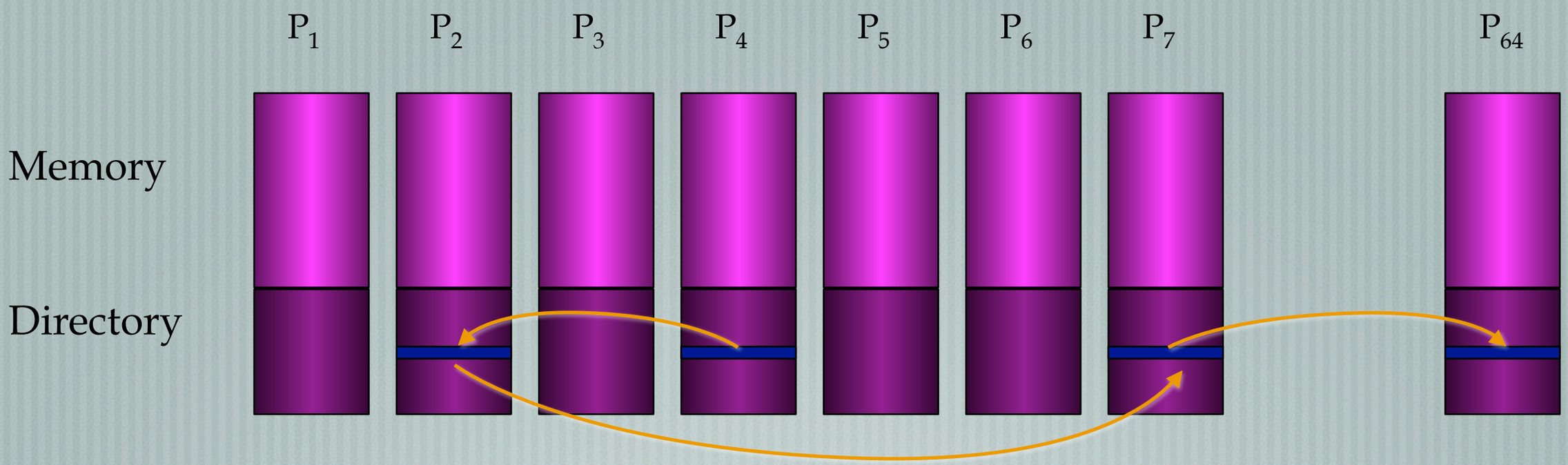
Full-map directories



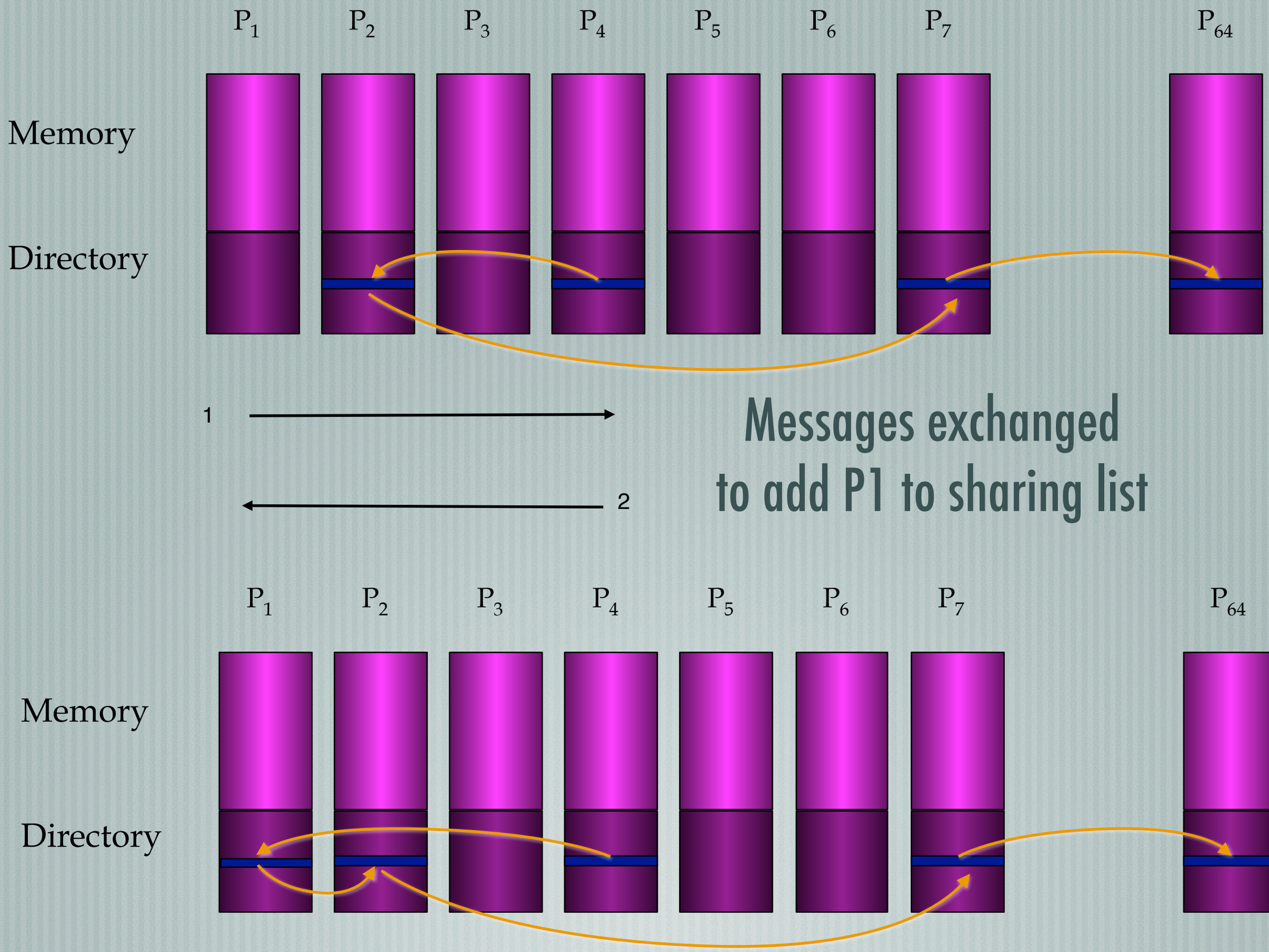
Sharing-list directories

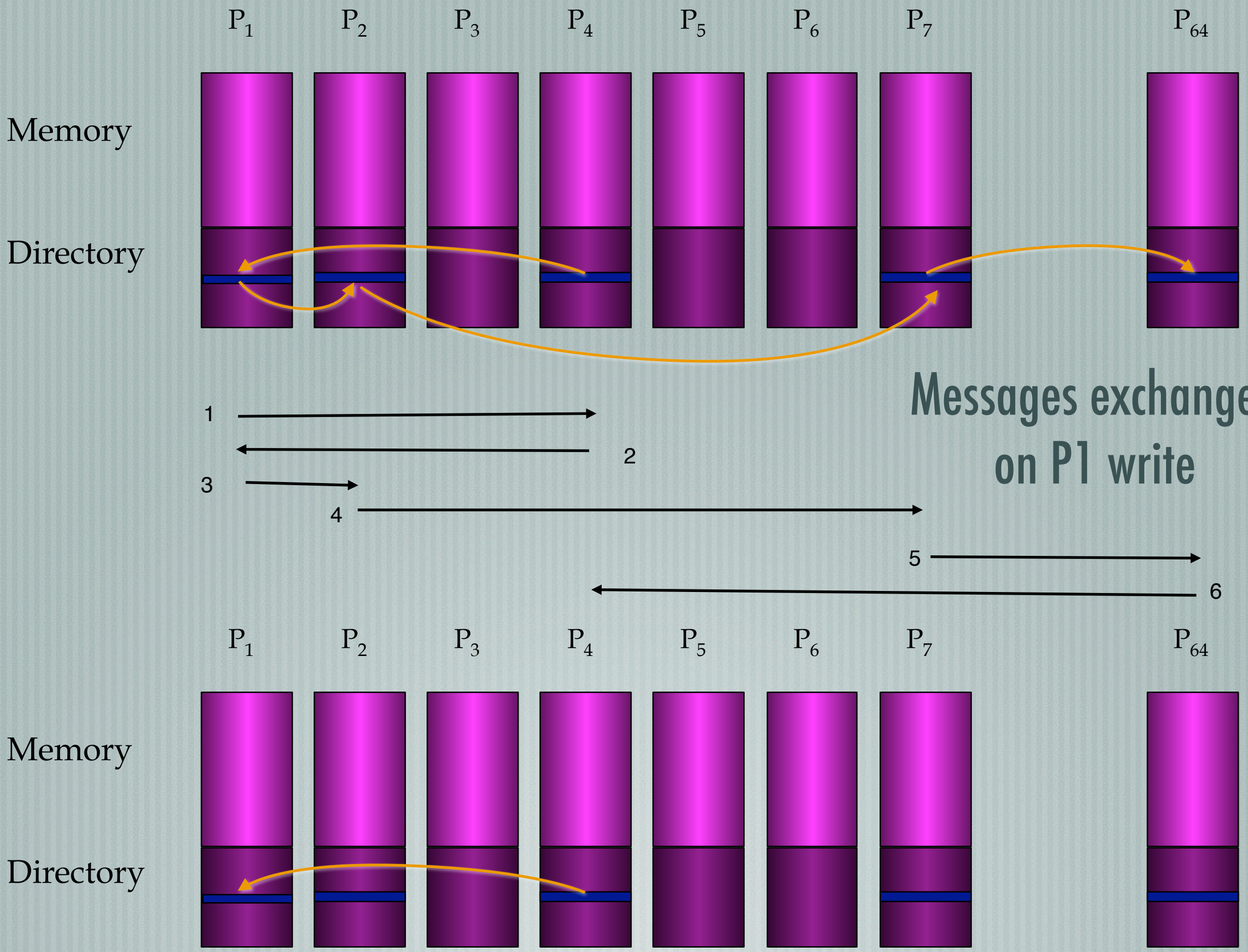
- A **sharing list** is a distributed data structure stored as a linked list of the processors sharing a given memory block
  - the advantage of this scheme is that message generation is distributed
  - protocol messages follow the linked list between nodes/processors in the list
- Each cache directory needs memory to store a link for every cache line
  - the directory can be implemented as a dedicated memory
  - in main memory
  - or as a combination with caching

Example



In this example processor P4 is the owner of a cache line which is shared by processors P2, P7, and P64


P1
P2
P3
P4
P5
P6
P7
P64
Memory
Directory
1
2
Messages exchanged
to add P1 to sharing list



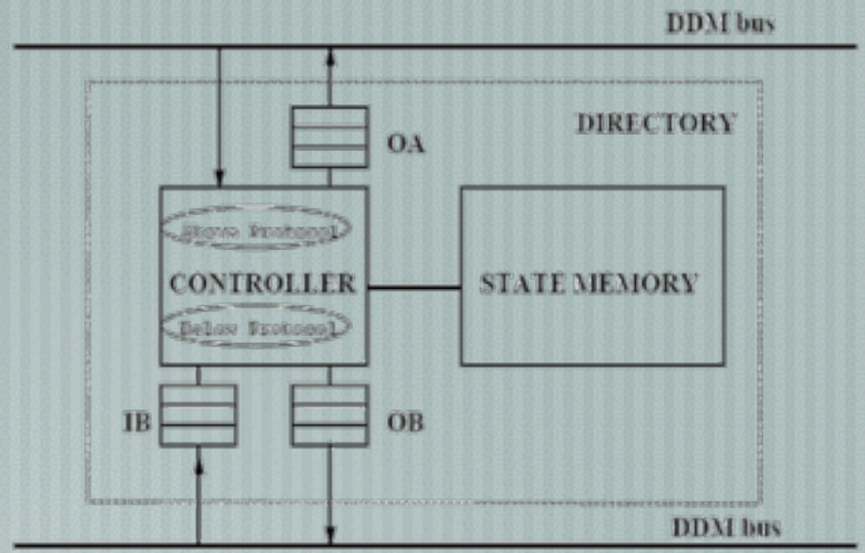
Messages exchanged on P1 write

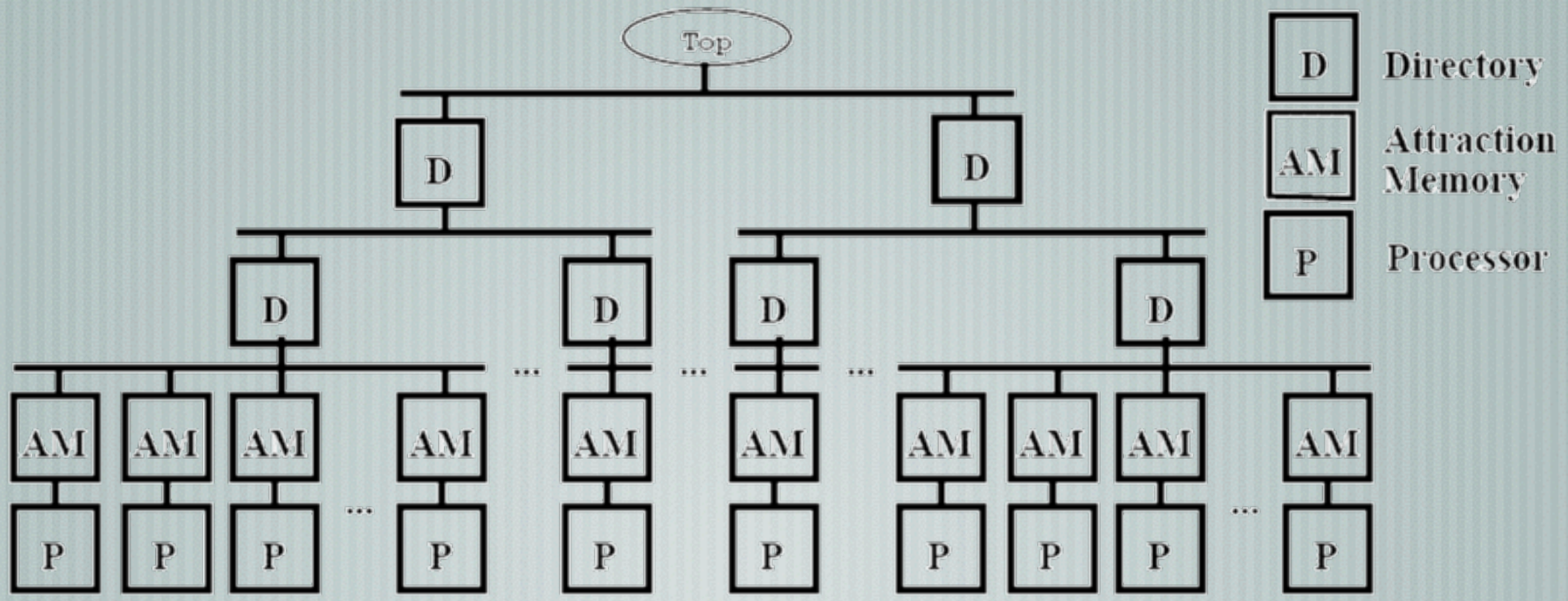
Cache-only Memory Architectures

- COMA
  - Data with no home location: attraction memory similar to L2 cache
  - Memory includes data, tags and states
  - Large capacity (normally DRAM)
- Properties:
  - Allow data to flow dynamically in the system
  - Always have to preserve the last copy
  - Hard to locate a data item

Data diffusion memory

- Hierarchical bus-based COMA
- Attraction memory ≈ big cache





General COMA properties

- In a Cache Only Memory Architecture (COMA) there is no home location for a block of data in memory: data **migrates** to where it is being used
- All memory is in a cache somewhere - usually DRAM
- Data in COMA has a global address but this is maintained as a tag using normal cache mapping techniques
  - all valid data must be assigned to a cache somewhere in the system, it can also be shared
  - directories maintain coherence information
- The KSR 1 is an example of a COMA (Kendall Square Research, Cambridge, MA, USA, 1986)

COMA - Summary

- **Advantages**
  - Replication of data not constrained by small local caches
  - Win at poor data placement by software (i.e., capacity cache misses in NUMA)
  - Better scalability to larger systems
- **Disadvantages (compared with NUMA)**
  - Complicated coherence protocol
  - Long coherence latency
  - Hard to locate a data item
  - Needs non-standard memory management hardware to implement AMs