


Introduction to Multiprocessors (Part II)



Cristina Silvano
Politecnico di Milano

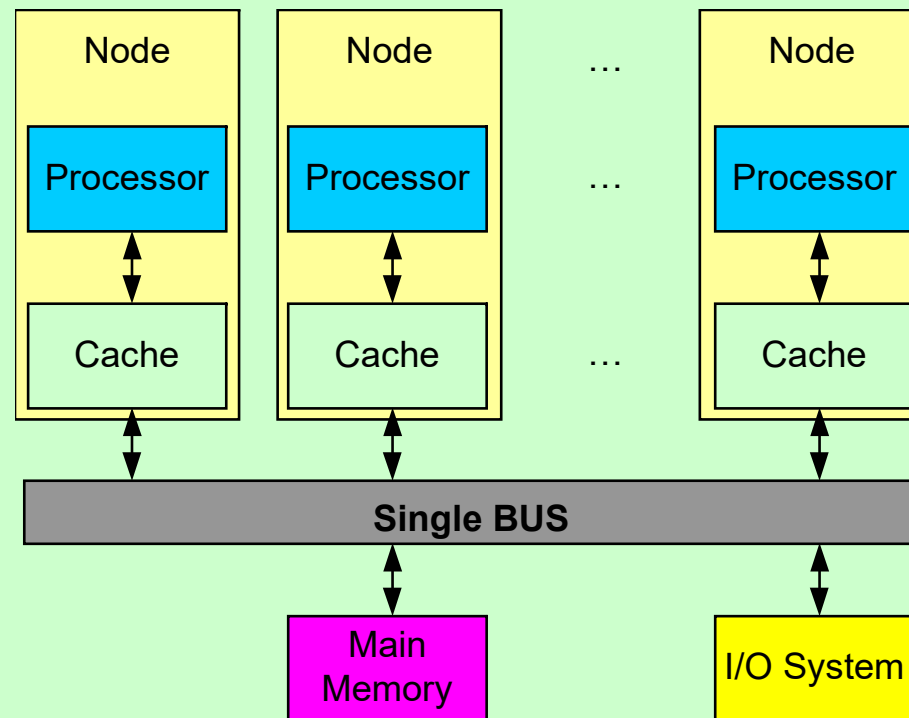
Outline



- ❑ The problem of cache coherence
 - Snooping protocols
 - Directory-based protocols

The Problem of Cache Coherence

- ❑ Caches serve to:
 - Increase bandwidth versus bus/memory
 - Reduce latency of accesses
 - Valuable for both private data and shared data
- ❑ What about cache coherence?



The Problem of Cache Coherence



- ❑ Shared-Memory Architectures cache both **private data** (used by a single processor) and **shared data** (used by multiple processors to provide communication).
- ❑ When shared data are cached, ***the shared values may be replicated in multiple caches.***
- ❑ In addition to the reduction in access latency and required memory bandwidth, this replication provides a reduction of shared data contention read by multiple processors simultaneously.
- ❑ The use of multiple copies of same data introduces a new problem: cache coherence.

Cache Coherence: 2 Processors with Write-Thru Caches

- ❑ Processors may see different values through their caches:

Time	Event	Cache contents for processor A	Cache contents for processor B	Memory contents for location X
0				1
1	Processor A reads X	1		1
2	Processor B reads X	1	1	1
3	Processor A stores 0 into X	0	1	0

The Problem of Cache Coherence



- ❑ Alternatively, accesses to shared data could be forced always to go around the cache to main memory (i.e. shared data are not cached) \Rightarrow slow solution that requires a very high bus bandwidth.
- ❑ Maintain coherence has two components:
read and ***write***.
- ❑ Multiple copies are not a problem when reading, but a processor must have exclusive access to write a word.
- ❑ Processors must have the most recent copy when reading an object, so all processors must get new values after a write.

The Problem of Cache coherence



- ❑ Coherence protocols must locate all the caches that share an object to be written.
- ❑ A write to a shared data can cause
 - either ***to invalidate*** all other copies
 - or ***to update*** the shared copies.

Solutions: Cache-Coherence Protocols



- ❑ HW-based solutions to maintain coherence:
Cache-Coherence Protocols
- ❑ Key issues to implement a cache coherent protocol in multiprocessors is tracking the state of any sharing of a data block.
- ❑ Two classes of protocols:
 - ***Snooping Protocols***
 - ***Directory-Based Protocols***

Cache Coherence Protocols



Two classes of protocols:

- ❑ **Snooping protocols:** each core tracks the sharing state of each block
 - A cache controller monitors (*snoops*) on the bus, to see what is being requested by another cache
- ❑ **Directory-based protocols:** the sharing state of each block is kept in one location, called the directory
 - In **SMPs**: a single directory
 - In **DSMs**: multiple, distributed directories (one for each main memory)

Snooping Protocols (Snoopy Bus)



Snooping Protocols (Snoopy Bus)



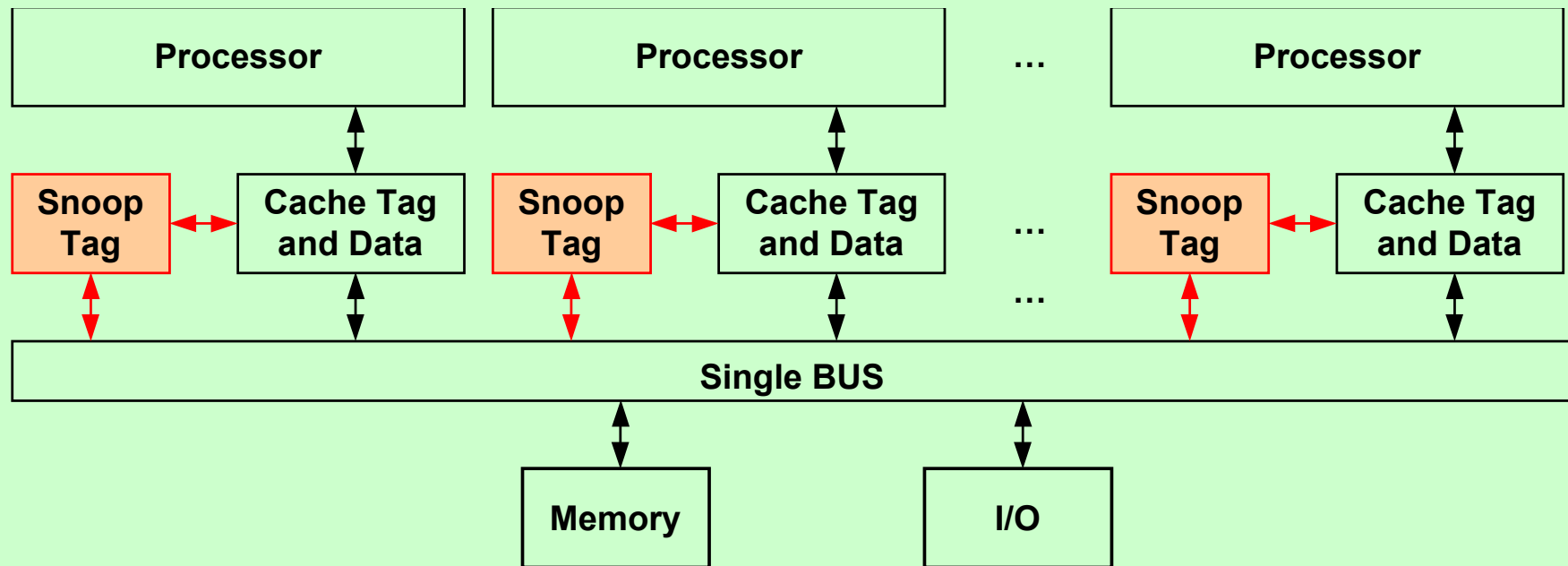
- ❑ All cache controllers monitor **(snoop)** on the bus to determine whether or not they have a copy of the block requested on the bus and respond accordingly.
- ❑ Every cache that has a copy of the shared block, also has a copy of the sharing state of the block, and no centralized state is kept.
- ❑ Send all requests for shared data to all processors.
- ❑ Require broadcast, since caching info is at processors.
- ❑ Suitable for **Centralized Shared-Memory Architectures**, and in particular for small scale multiprocessors with ***single snoopy bus***.

Snooping Protocols (Snoopy Bus)

Problems:

- ❑ Since every bus transaction checks the cache address tags, this checking can interfere with the processor operations.
- ❑ When there is interference, the processor will likely stall because the cache is unavailable.
- ❑ To reduce the interference with the processor's accesses to the cache, we duplicate the address tag portion of the cache (not the whole cache) for snooping activities.
- ❑ In practice, ***an extra read port is added to the address tag portion of the cache.***

Snoop Tag



Basic Snooping Protocols



- ❑ Two types of snooping protocols depending on what happens on a write operation:
 - ***Write-Invalidate Protocol***
 - ***Write-Update (or Write-Broadcast) Protocol***

Write-Invalidate Protocol



- ❑ The writing processor issues an ***invalidation*** signal over the bus to cause all copies in other caches to be invalidated before changing its local copy.
- ❑ The writing processor is then free to update the local data until another processor asks for it.
- ❑ All caches on the bus check to see if they have a copy of the data and, if so, they must invalidate the block containing the data.
- ❑ This scheme allows ***multiple*** readers but only a ***single*** writer.

Write-Invalidate Protocol



- ❑ This scheme uses the bus only on the ***first*** write to invalidate the other copies.
- ❑ Subsequent writes do not result in bus activity.
- ❑ This protocol provides ***similar benefits to write-back*** protocols in terms of reducing demands on bus bandwidth.
- ❑ ***Read Miss:***
 - ***Write-Through:*** Memory always up-to-date
 - ***Write-Back:*** Snoop in caches to find the most recent copy.

Write-Update Protocol



- ❑ The writing processor broadcasts the new data over the bus; all caches check if they have a copy of the data and, if so, all copies are **updated** with the new value.
- ❑ This scheme requires the continuous broadcast of writes to shared data (while write-invalidate deletes all other copies so that there is only one local copy for subsequent writes)
- ❑ This protocol is **like write-through** because all writes go over the bus to update copies of the shared data.
- ❑ This protocol has the advantage of making the new values appear in caches sooner \Rightarrow reduced latency
- ❑ **Read Miss:**
 - **Write-Through:** Memory always up-to-date

Snooping Protocols

❑ Write invalidate (write-back to memory)

- On write: invalidate all other cache copies and memory not up-to-date
- On next read miss: first write-back to memory, then copy in cache

Processor activity	Bus activity	Contents of processor A's cache	Contents of processor B's cache	Contents of memory location X
				0
Processor A reads X	Cache miss for X	0		0
Processor B reads X	Cache miss for X	0	0	0
Processor A writes a 1 to X	Invalidation for X	1		0
Processor B reads X	Cache miss for X	1	1	1

❑ Write update (write-through in memory)

- On write, update all copies in caches and in memory

Snooping Protocols



- ❑ Most part of commercial cache-based multiprocessors uses:
 - ***Write-Back Caches*** to reduce bus traffic
⇒ they allows more processors on a single bus.
 - ***Write-Invalidate Protocol*** to preserve bus bandwidth.
- ❑ Write serialization due to bus serializing request:
bus is single point of arbitration.
- ❑ Problem of **Bus Arbitration**

Write Invalidate vs Update



- ❑ Invalidate requires one transaction per write-run
- ❑ Invalidate uses spatial locality: one transaction per block
- ❑ Update has lower latency between write and read
- ❑ Update: increased bandwidth versus decreased latency.

Snooping Protocols: MSI



❑ *Write-Invalidate Snooping Protocol, Write-Back Cache*

❑ Each cache block can be in one of *three* states:

- *Modified (or Dirty)* : cache has only copy, its writeable, and dirty (block cannot be shared anymore)
- *Shared (or Clean)* (read only): the block is clean (not modified) and can be read
- *Invalid* : block contains no valid data

❑ Each block of memory is in one of *three* states:

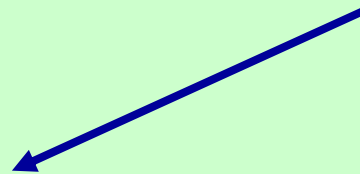
- *Shared* in all caches and up-to-date in memory (Clean)
- *Modified* in exactly one cache (Dirty)
- *Uncached* when not in any caches

INVALID Cache Block

- ❑ An example of INVALID block in cache C1: always miss

state	TAG	DATA
INVALID	04	SALLY

CPU Read Miss
Place Read Miss
TAG08 on Bus

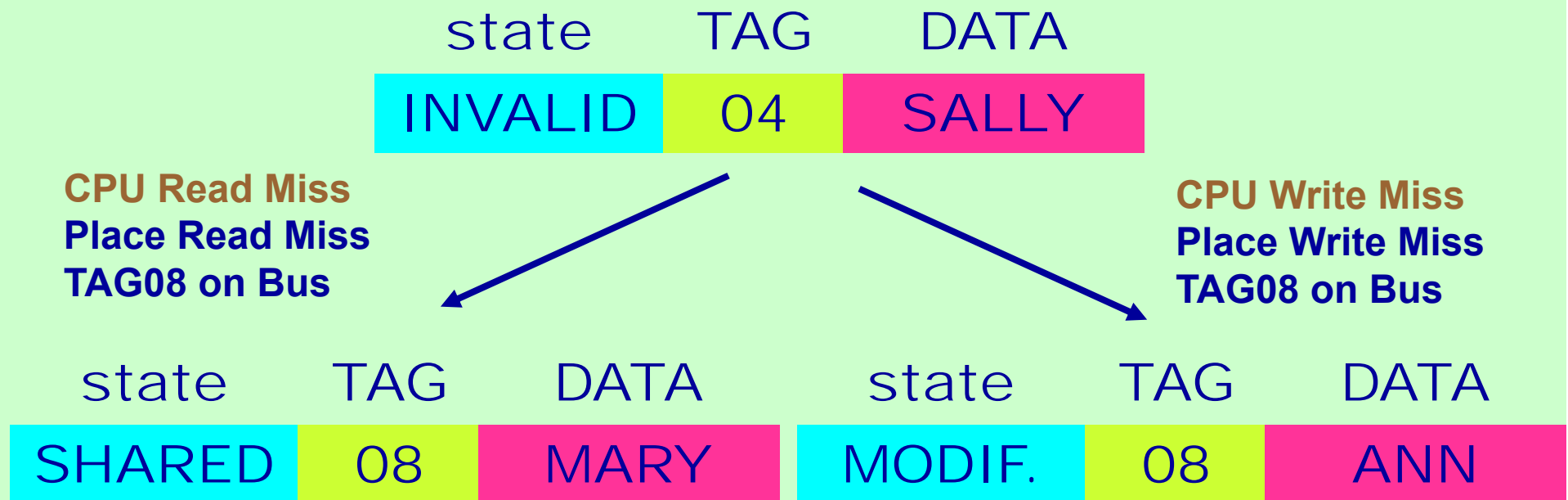


state	TAG	DATA
SHARED	08	MARY

IF (other caches have TAG08 SHARED)
 ⇒ blocks stay SHARED;
IF (another cache C2 has TAG08 MODIF.)
 ⇒WRITE-BACK block in memory;
 block SHARED in C2;
LOAD from mem in C1 as SHARED

INVALID Cache Block

- An example of INVALID block in cache C1: always **miss**



IF (other caches have TAG08 SHARED)

⇒ blocks stay SHARED;

IF (another cache C2 has TAG08 MODIF.)

⇒WRITE-BACK block TAG08 in mem; block SHARED in C2;

LOAD block TAG08 from mem in C1 as SHARED

IF (other caches have TAG08 SHARED)

⇒blocks INVALDATE in others;

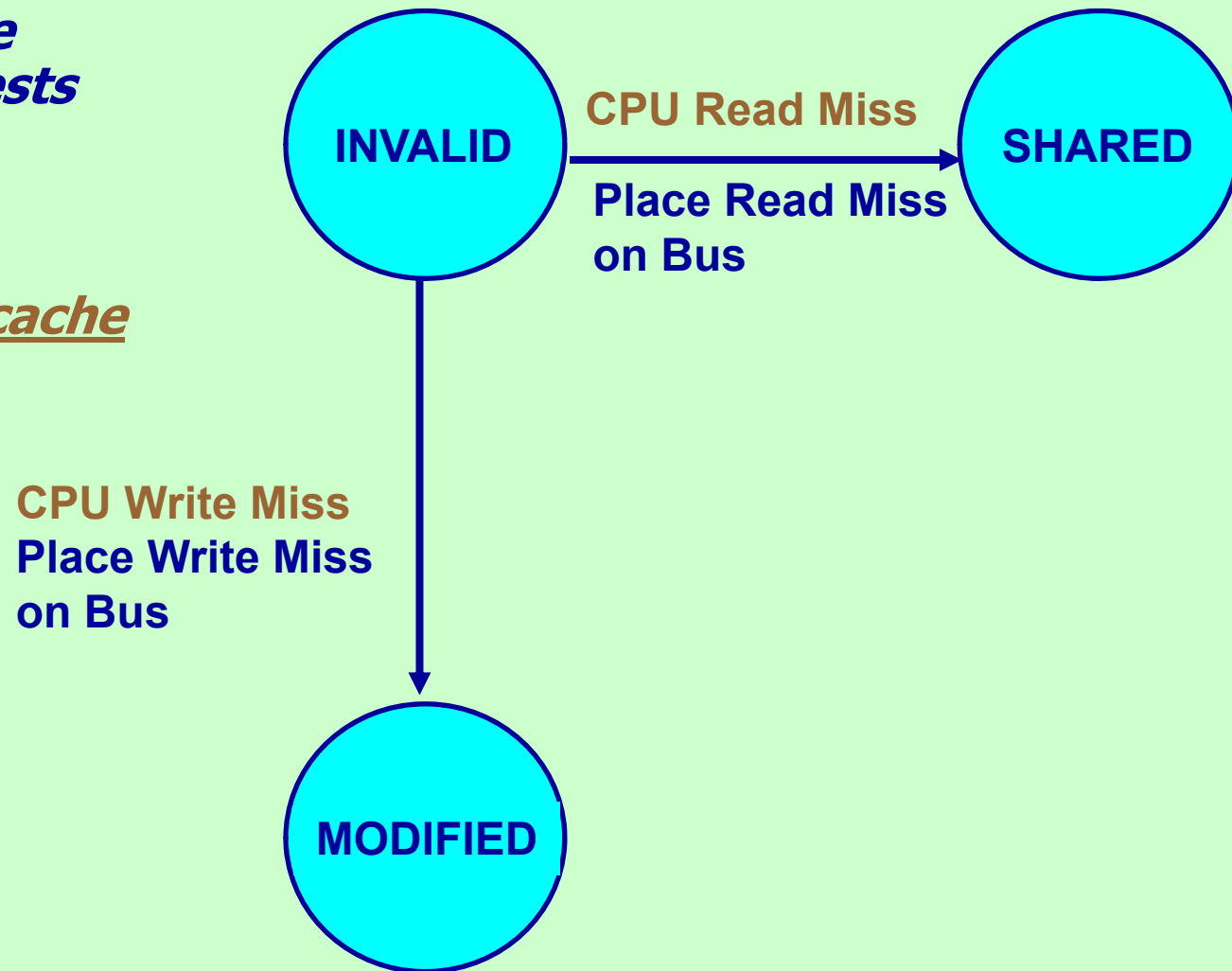
IF (another cache C2 has TAG08 MODIF.)

⇒WRITE-BACK block TAG08 in mem; block INVALDATE in C2;

LOAD block TAG08 from mem in C1 as MODIF; WRITE in C1

INVALID State in Snoopy-Cache FSM

- ❑ *State machine for CPU requests for each cache block*
- ❑ *Each node represents a cache block state*



SHARED Cache Block

state TAG DATA

SHARED 04 SALLY

CPU Read Hit; Cache Block Unchanged

CPU Read Miss; Place Read Miss TAG08 on Bus

state TAG DATA

SHARED 08 MARY

CPU Write Hit; Send Write Invalidate TAG04 on Bus

state TAG DATA

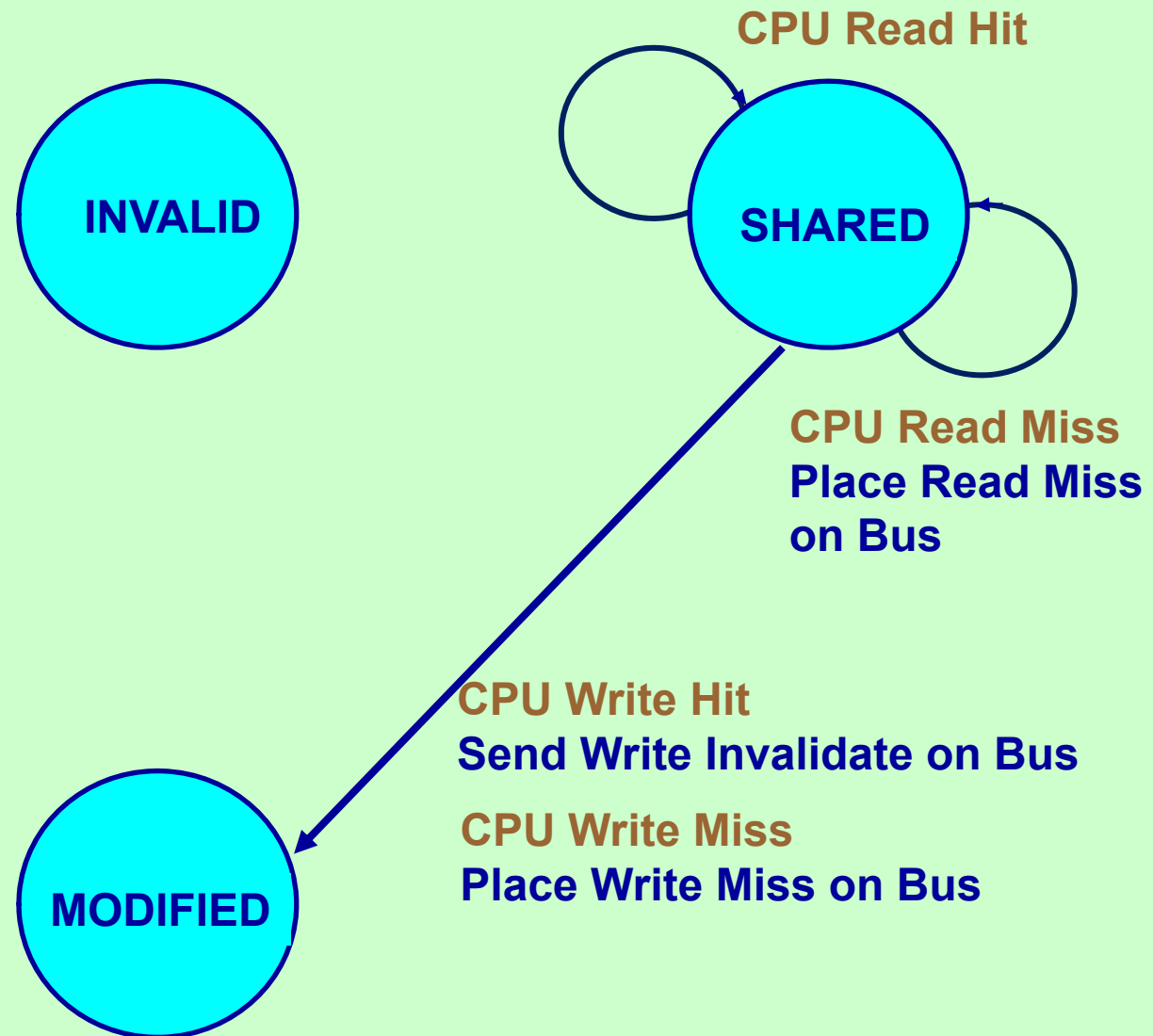
MODIF. 04 JANE

CPU Write Miss; Place Write Miss TAG08 on Bus

state TAG DATA

MODIF. 08 ANN

SHARED State in Snoopy-Cache FSM



MODIFIED Cache Block

state TAG DATA

MODIF.

04

SALLY

CPU Read Hit; Cache Block Unchanged

CPU Read Miss; Write Back Block TAG04
Place Read Miss TAG08 on Bus

state

TAG

DATA

SHARED

08

MARY

CPU Write Hit (No Send Write Invalidate TAG04 on Bus)

state

TAG

DATA

MODIF.

04

JANE

CPU Write Miss; Write Back Block TAG04
Place Write Miss TAG08 on Bus

state

TAG

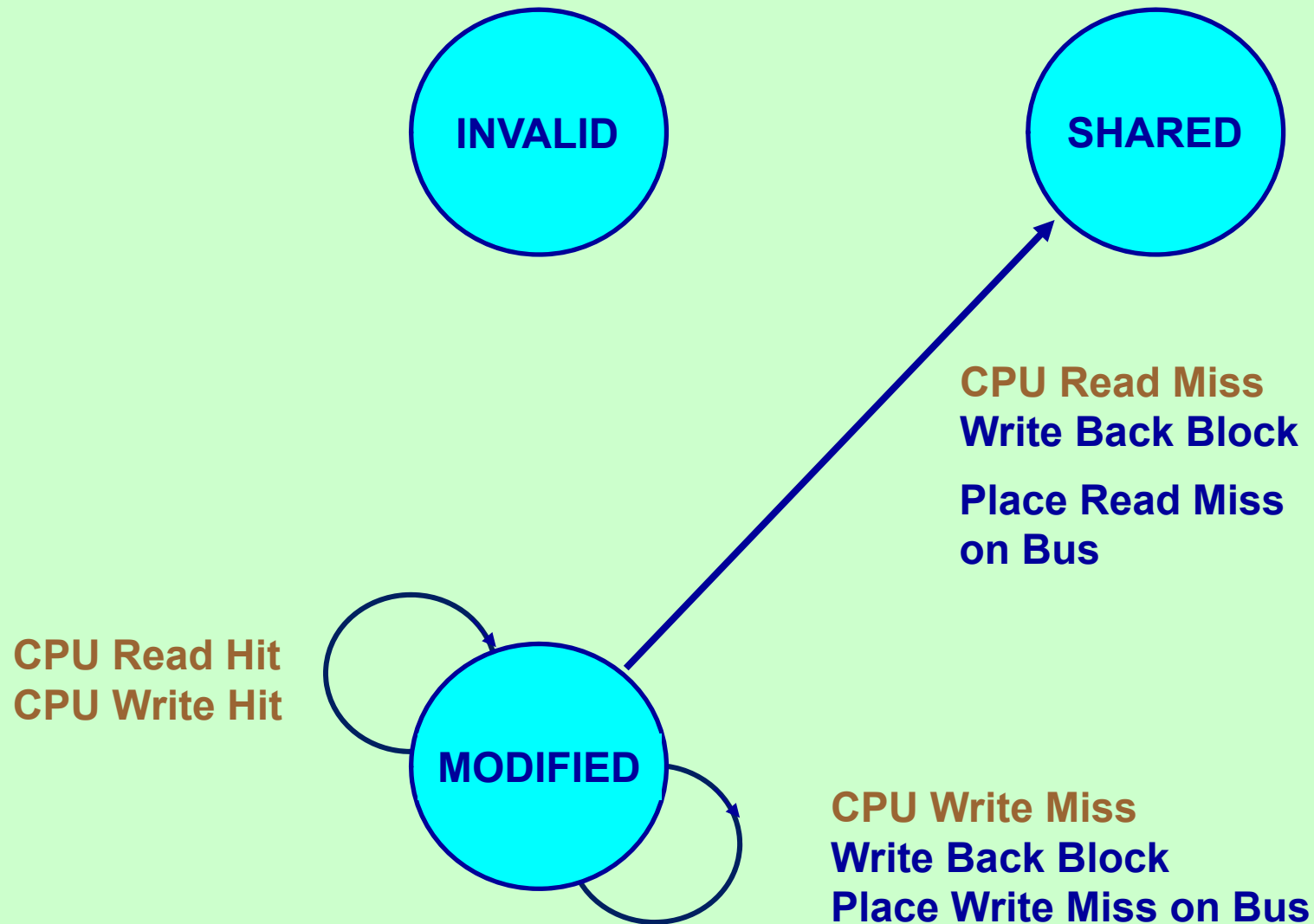
DATA

MODIF.

08

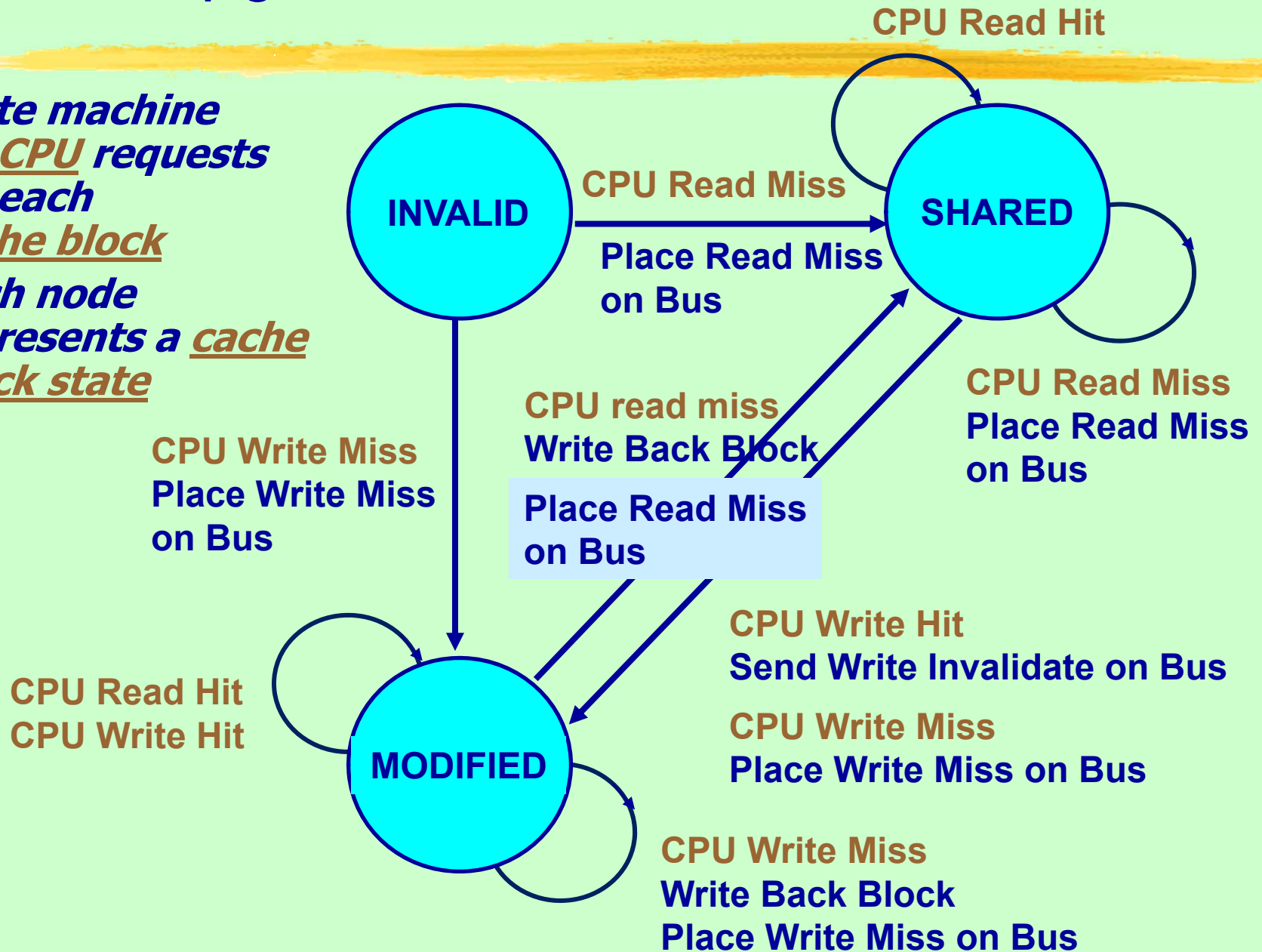
ANN

MODIFIED State in Snoopy-Cache FSM



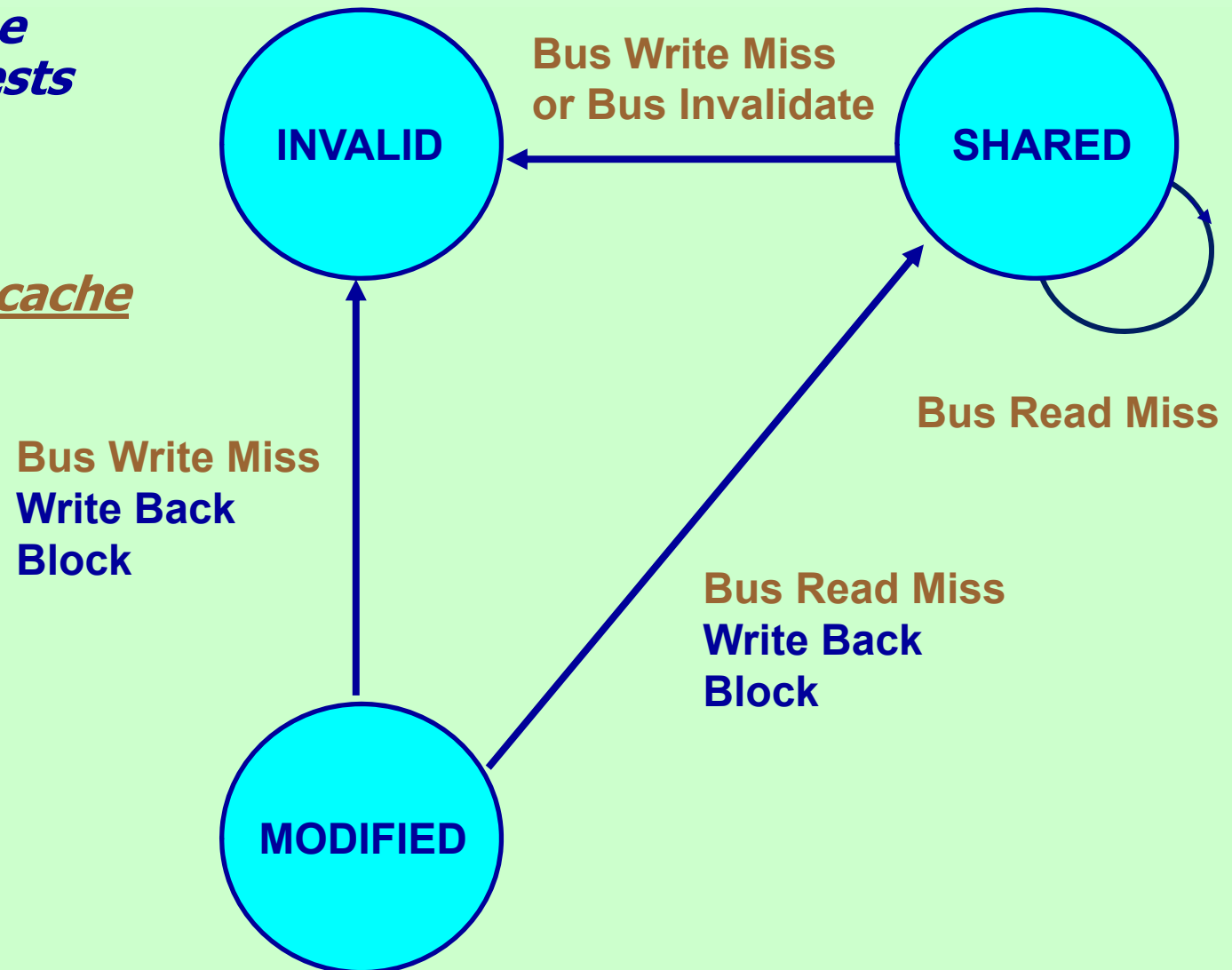
Snoopy-Cache FSM

- ❑ *State machine for CPU requests for each cache block*
- ❑ *Each node represents a cache block state*



Snoopy-Cache FSM

- ❑ *State machine for Bus requests for each cache block*
- ❑ *Each node represents a cache block state*



Snooping Protocol Variation: MESI



- ❑ **MESI Protocol:** Write-Invalidate
- ❑ Each cache block can be in one of *four* states:
 - **Modified** : the block is dirty and cannot be shared; cache has only copy, its writeable.
 - **Exclusive: the block is clean and cache has only copy;**
 - **Shared**: the block is clean and other copies of the block are in cache;
 - **Invalid**: block contains no valid data

MESI Protocol



- ❑ In both **Shared** and **Exclusive**, the memory has an up-to-date version of the data
- ❑ **Benefit: A write to an Exclusive block does not require to send the invalidation signal on the bus, since no other copies of the block are in cache.**
- ❑ A write to a **Shared** block implies the invalidation of the other copies of the block in cache.

Directory-Based Protocols

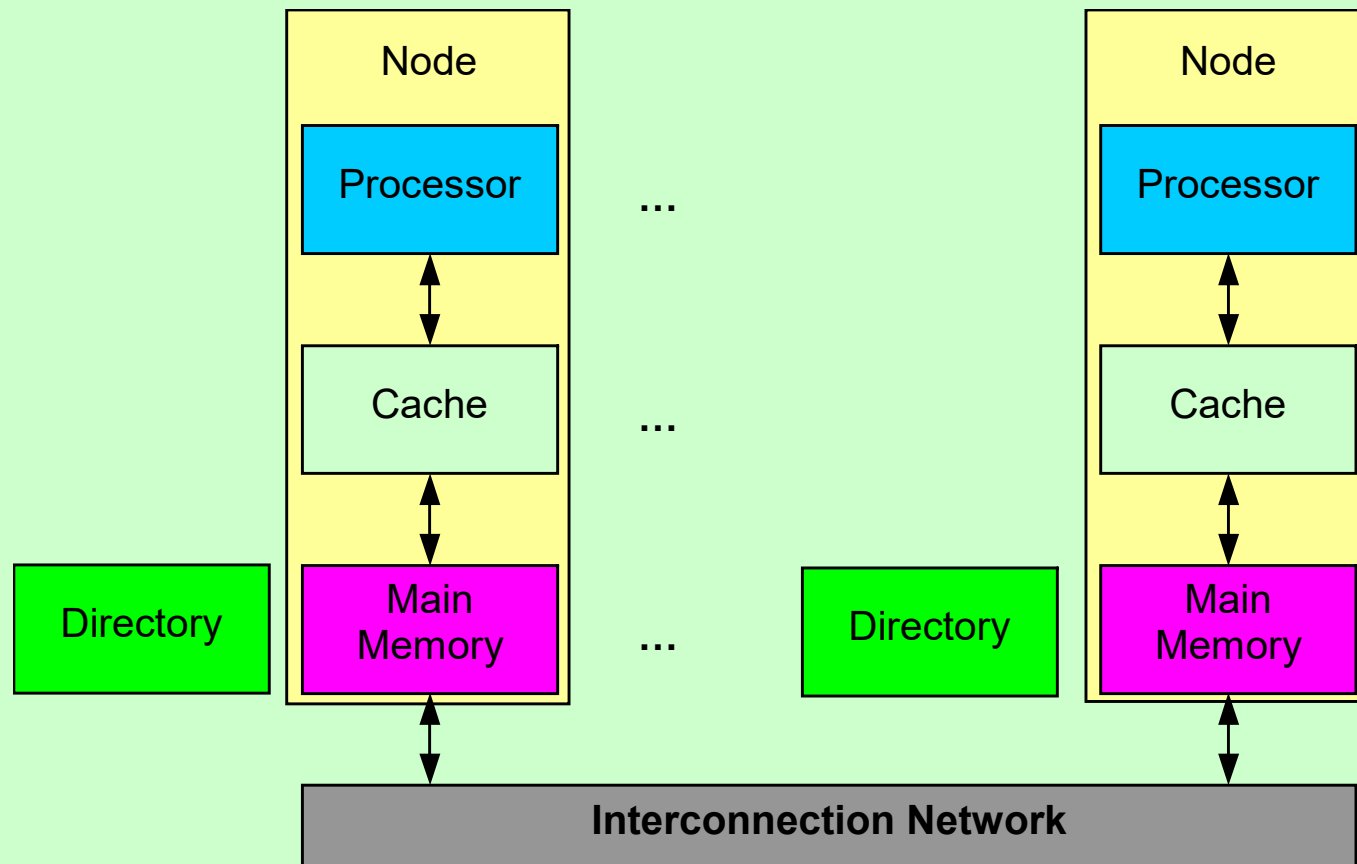


Directory-Based Protocols

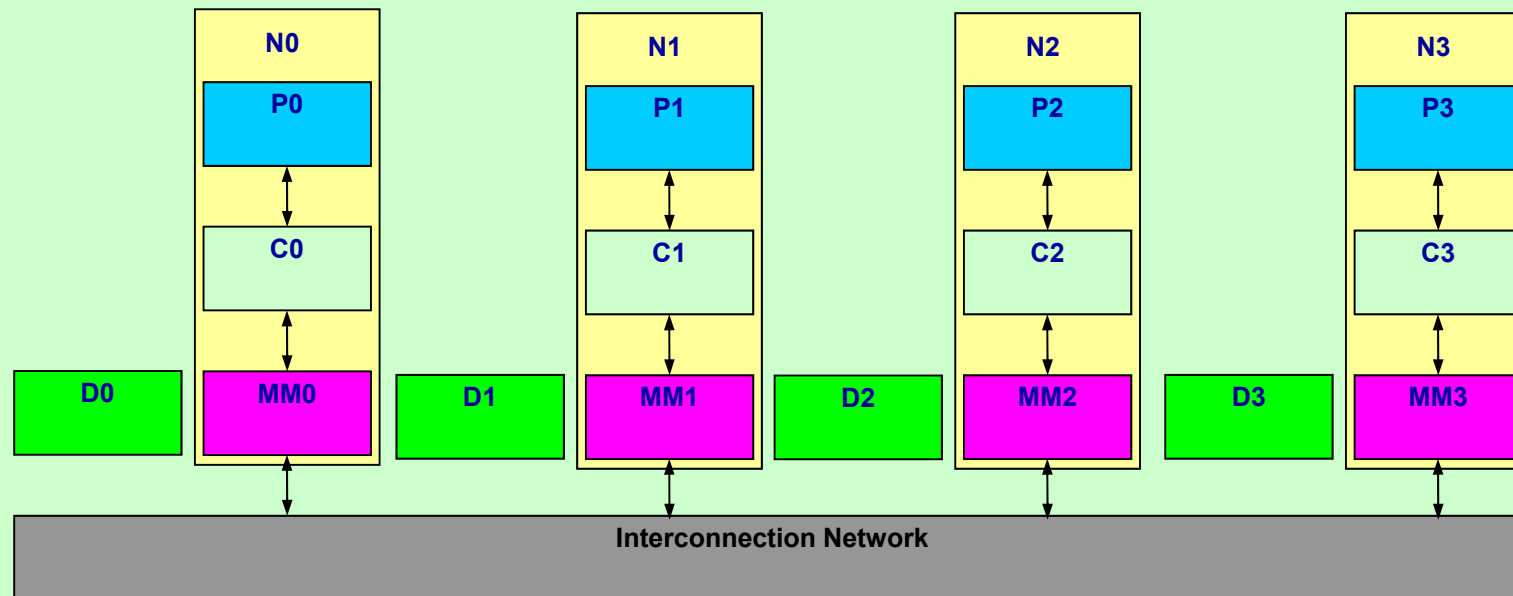


- ❑ The sharing state of a block of physical memory is kept in just one location, called **directory**.
- ❑ Each entry in the directory is associated to each block in the memory (directory size is proportional to the number of memory blocks time the number of processors).
- ❑ For Distributed Shared-Memory Architectures, the directory is **distributed** (as well as the memory) to avoid bottlenecks.
- ❑ To avoid broadcast: send point-to-point requests to processors.
- ❑ Better scalable than snooping protocols.
- ❑ The entries of the directory are distributed, but the sharing state of a block is stored in a single location in the directory.

Directory-Based Protocols



Example: 4 processors (P0, P1, P2, P3)



Directory-Based Protocols



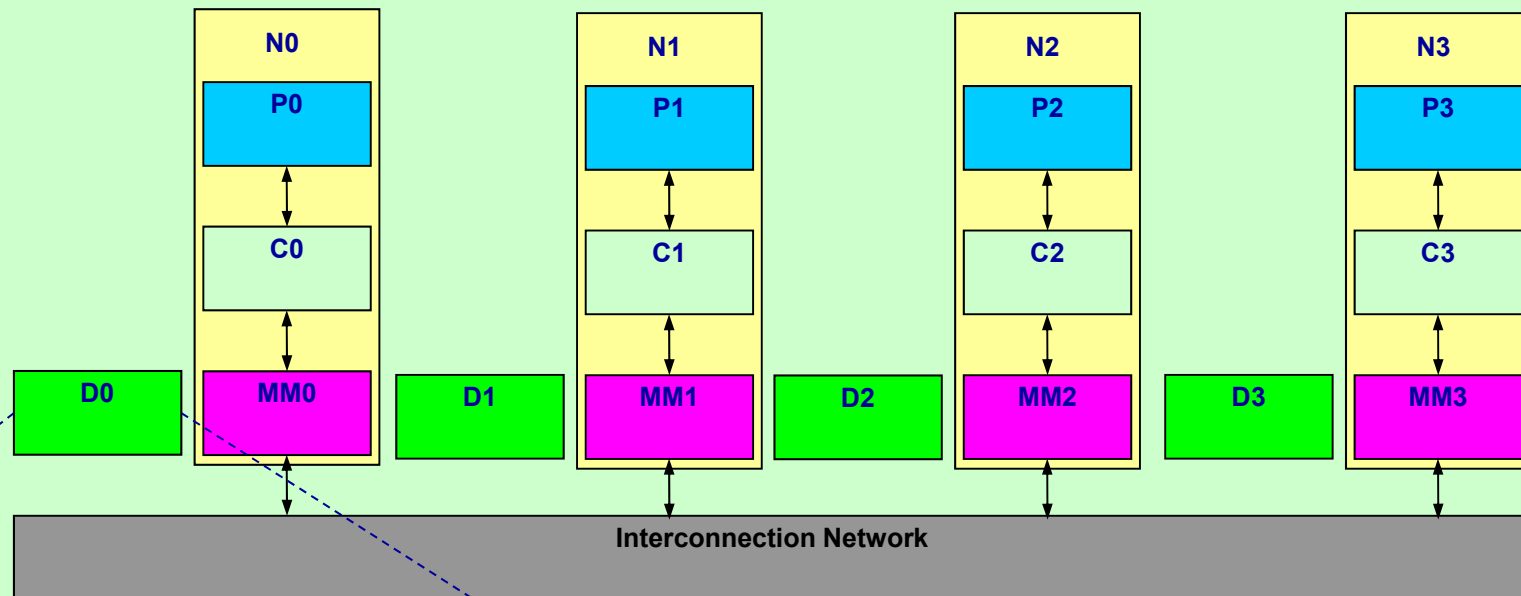
- ❑ The directory maintains info regarding:
 - The **state** of each block
 - **Which processor(s)** has(have) a copy of the block (usually bit vector, 1 if processor has copy).
 - Which processor is the **owner** of the block (when the block is in the exclusive state, 1 if processor is owner).

Directory-Based Protocols



- ❑ **Three** possible states for each cache block **in the directory**:
 - **Uncached**: no processor has a copy of the cache block; block not valid in any cache;
 - **Shared**: one or more processors have cache block, and memory is up-to-date; sharer set of proc. IDs;
 - **Modified**: only one processor (**the owner**) has data that has been modified so the memory is out-of-date; owner proc. ID;

Example: 4-block directory info for 4 nodes (N0, N1, N2, N3)



Block	Coherence State	Sharer / Owner Bits
B0	Uncached	- - - -
B1	Shared	1 0 1 0
B2	Shared	0 0 1 0
B3	Modified	0 1 0 0

Directory-Based Protocols



- ❑ ***Message-oriented protocol:*** The requests generate messages sent between nodes (point-to-point requests) to maintain coherence and all messages must receive explicit answers.
- ❑ No bus and don't want to broadcast:
 - Interconnect no longer single arbitration point
 - All messages have explicit responses
- ❑ The snooping protocols are ***transaction-based:*** all nodes must snoop on the bus transactions.

Coherence states in cache



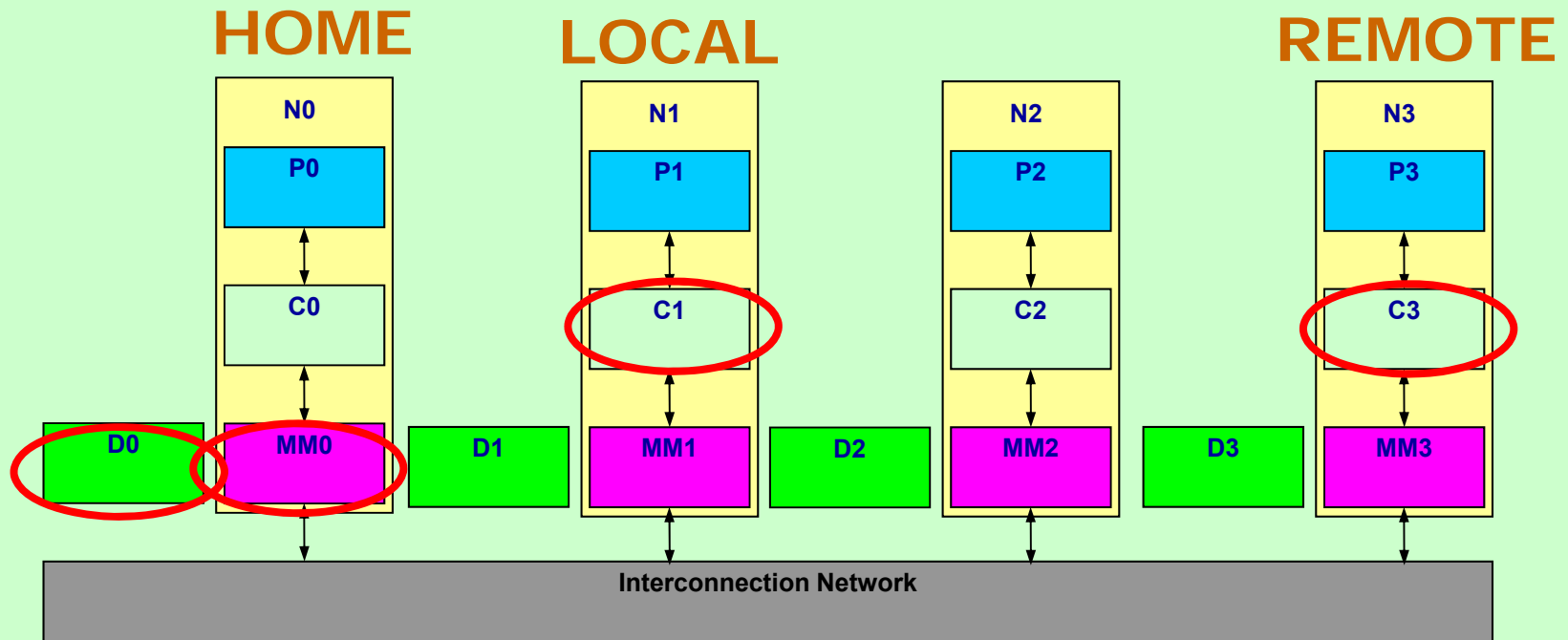
- ❑ Each block **in the cache** can be in *three* possible coherence states (such as in snooping protocols):
 - **Shared** (read only): the block is valid/up-to-date (clean) and it is shared with other processors.
 - **Modified** (read/write): the block is dirty and this processor is the only owner;
 - **Invalid**: block contains no valid data

Local, Home and Remote Nodes



- ❑ Terms: typically **three** processors involved:
 - **Local node** where a request originates;
 - **Home node** where the memory location (and directory entry) of an address resides;
 - **Remote node** has a copy of a cache block, whether dirty or shared.

Local, Home and Remote Nodes



Local, Home and Remote Nodes

- ❑ Terms: typically **three** processors involved:
 - **Local node** where a request originates;
 - **Home node** where the memory location (and directory entry) of an address resides;
 - **Remote node** has a copy of a cache block, whether dirty or shared.
- ❑ The **L** node can be the **H** node and vice-versa (if the **L** node is equal to the **H** node we can use ***intra-node*** transactions instead of ***inter-node*** messages based on the same protocol).
- ❑ The **R** node can be the **H** node and vice-versa
- ❑ Obviously the **L** node and the **R** node are different.

Directory-Based Protocols



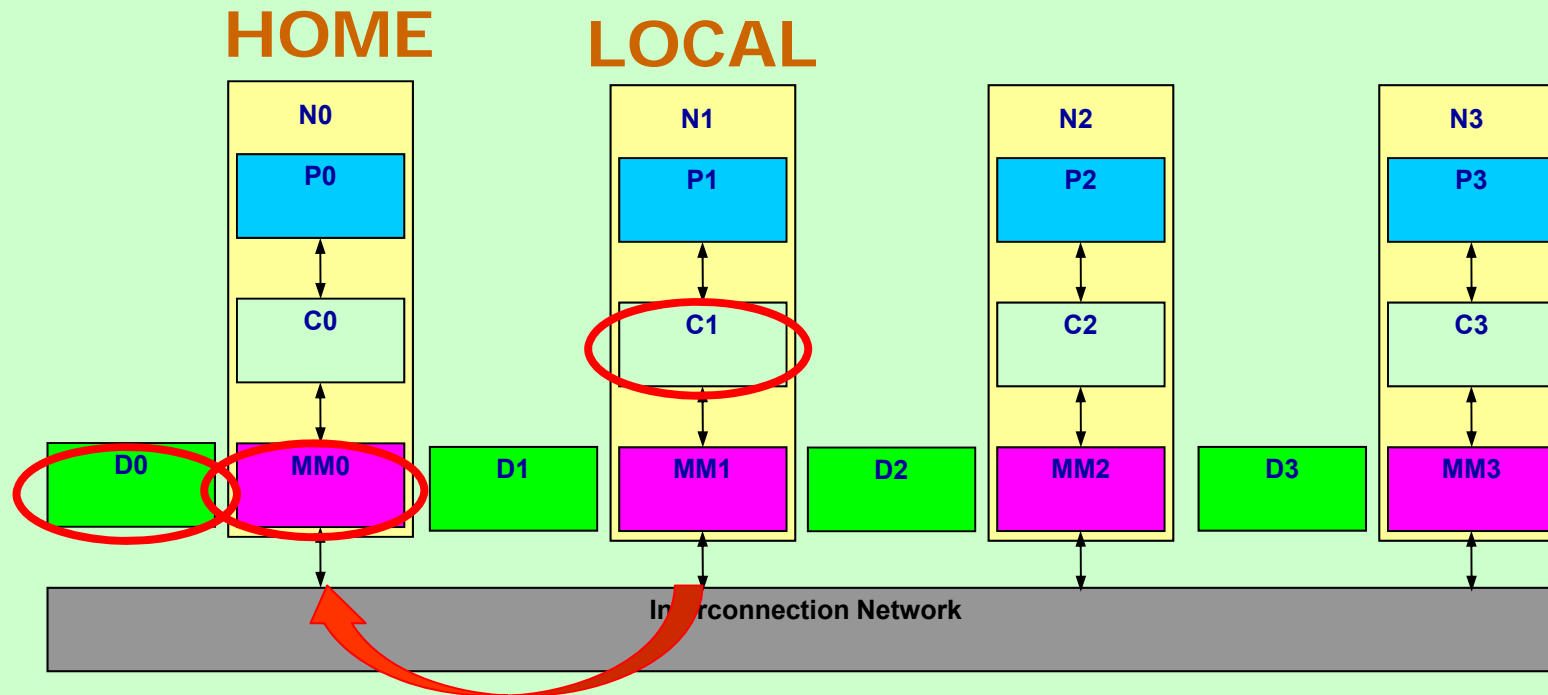
- ❑ Directory-based protocol must implement two primary operations:
 - handling a ***read miss***
 - handling a ***write*** to a shared, clean cache block.
- ❑ Handling a ***write miss*** to a shared block is a simple combination of these two basic operations.
- ❑ To implement these operations, the directory must track the ***state*** of each cache block (Uncached, Shared, Modified).

Directory Protocol Messages

<i>Message type</i>	<i>Source</i>	<i>Destination</i>	<i>Msg Content</i>
Read miss	Local cache	Home directory	P, ADD
<i>➤ Processor P has a read miss at address ADD; request data to home and make P as read sharer.</i>			

- ❑ **Miss** requests sent by the Local cache to the Home directory.
- ❑ It will follow a **Data Value Reply** from the Home directory to the Local cache

Read miss from local node N1 to home N0



Directory Protocol Messages

<i>Message type</i>	<i>Source</i>	<i>Destination</i>	<i>Msg Content</i>
---------------------	---------------	--------------------	--------------------

Read miss	Local cache	Home directory	P, ADD
------------------	--------------------	-----------------------	---------------

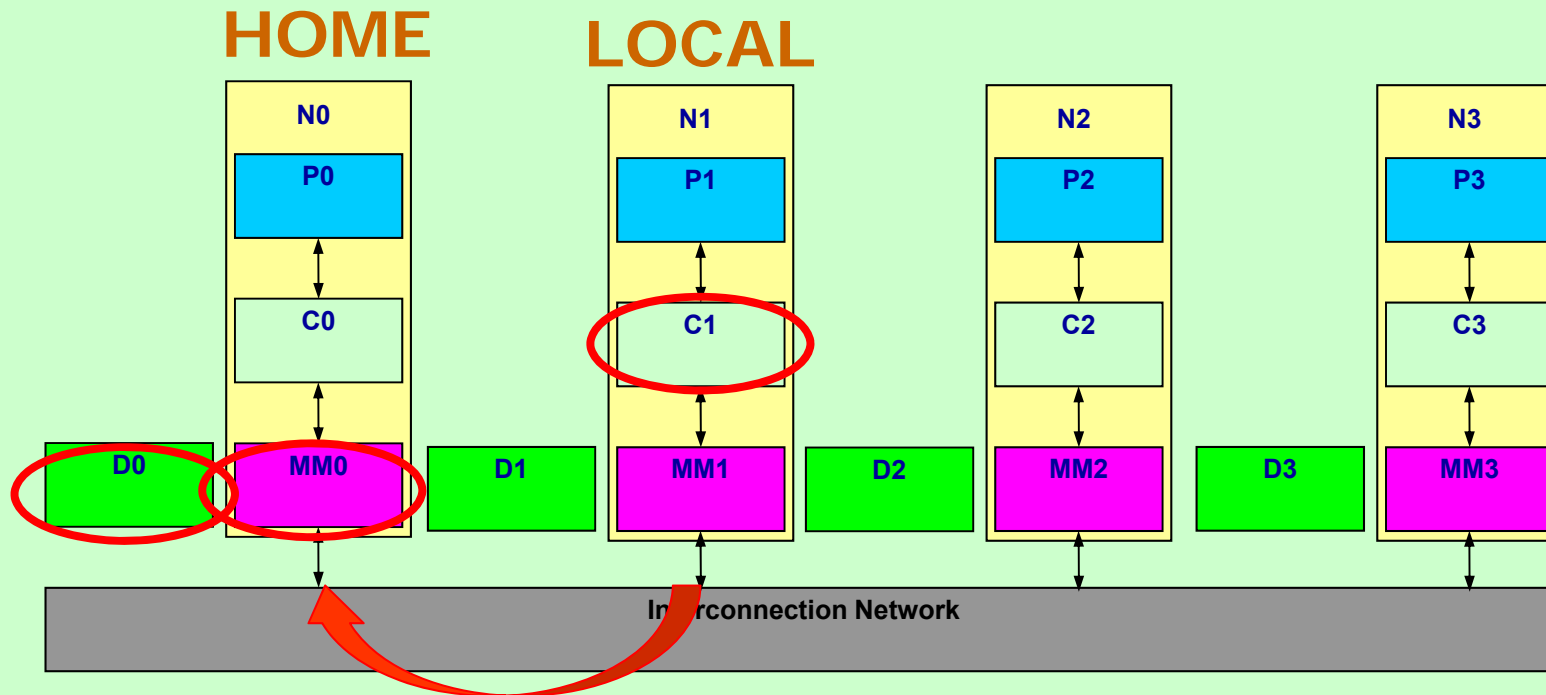
➤ *Processor P has a read miss at address ADD;
request data to home and make P a read sharer.*

Write miss	Local cache	Home directory	P, ADD
-------------------	--------------------	-----------------------	---------------

➤ *Processor P has a write miss at address ADD;
request data to home and make P the exclusive owner.
(The home will send an **Invalidate** message to other sharing
caches).*

- ❑ Both messages are **miss** requests sent by the Local cache to the Home directory.
- ❑ To both messages will follow a **Data Value Reply** from the Home directory to the Local cache

Write miss from local node N1 to home N0



Directory Protocol Messages

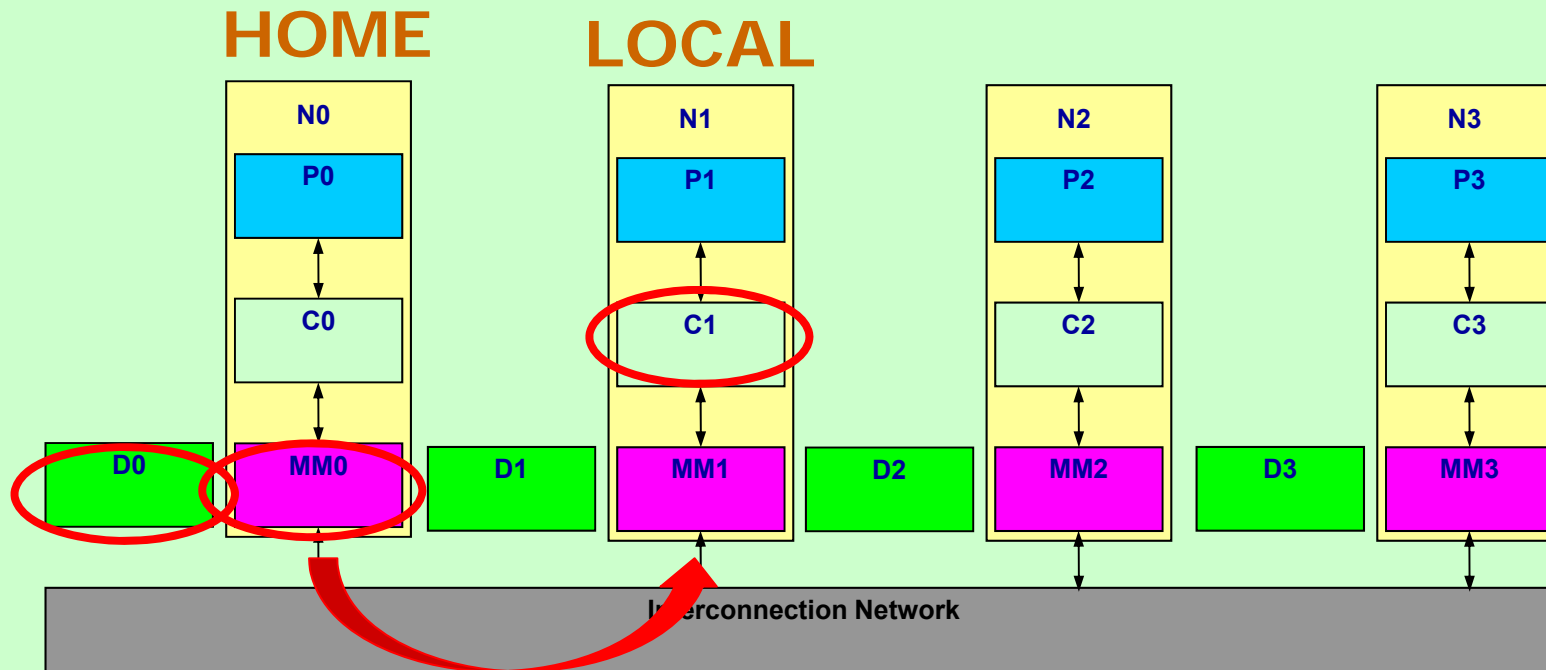


<i>Message type</i>	<i>Source</i>	<i>Destination</i>	<i>Msg Content</i>
----------------------------	----------------------	---------------------------	---------------------------

Data value reply	Home directory	Local cache	Data
-------------------------	-----------------------	--------------------	-------------

- *Return a data value from the Home memory back to the requesting Local node. (**Read/Write Miss** response)*

Data value reply from home node N1 to local node N0



Directory Protocol Messages

<i>Message type</i>	<i>Source</i>	<i>Destination</i>	<i>Msg Content</i>
---------------------	---------------	--------------------	--------------------

Invalidate	Local cache	Home directory	ADD
-------------------	--------------------	-----------------------	------------

➤ *Request to Home to send **Invalidate** to all remote caches that are caching the block at address ADD.*

Invalidate	Home directory	Remote cache	ADD
-------------------	-----------------------	---------------------	------------

➤ ***Invalidate** a shared copy of data at address ADD in all remote caches*

Directory Protocol Messages

<i>Message type</i>	<i>Source</i>	<i>Destination</i>	<i>Msg Content</i>
Fetch	Home directory	Remote cache (owner)	ADD

- **Fetch** the block in Home at address ADD and owner will send data to its Home directory (through **Data Write Back**); change the block state in Remote cache from Modified to Shared. (Block state in Home changes from Modified to Shared).

Fetch/Inv.	Home directory	Remote cache (owner)	ADD
-------------------	-----------------------	---------------------------------------	------------

- **Fetch** the block in home at address ADD and owner will send data to its home directory (through **Data Write Back**); **Invalidate** the block in the remote cache. (Block state in Home stays Modified – **but owner changed!**).

Directory Protocol Messages

<i>Message type</i>	<i>Source</i>	<i>Destination</i>	<i>Msg Content</i>
Data write-back	Remote Cache (owner)	Home directory	ADD, Data

- *Write-back a data value for address ADD from Remote cache owner to the Home memory (fetch and fetch/invalidate response)*

□ **Data write-back** occurs for two reasons:

- *when a block is replaced in a cache and must be written-back to its home memory;*
- *in reply to fetch or fetch/invalidate messages from the home.*

Directory Protocol Messages

Message type	Source	Destination	Message contents	Function of this message
Read miss	Local cache	Home directory	P, A	Node P has a read miss at address A; request data and make P a read sharer.
Write miss	Local cache	Home directory	P, A	Node P has a write miss at address A; request data and make P the exclusive owner.
Invalidate	Local cache	Home directory	A	Request to send invalidates to all remote caches that are caching the block at address A.
Invalidate	Home directory	Remote cache	A	Invalidate a shared copy of data at address A.
Fetch	Home directory	Remote cache	A	Fetch the block at address A and send it to its home directory; change the state of A in the remote cache to shared.
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	D	Return a data value from the home memory.
Data write-back	Remote cache	Home directory	A, D	Write-back a data value for address A.

Coherence states in the directory



- ❑ Let us consider the three possible coherence states for each block **in the home directory**:
 - **U** Uncached
 - **S** Shared
 - **M** Modified

Uncached State

- When a **directory block B0 in home N0** is in the ***U state***, the only copy in memory is up to date, so the only two possible requests for the block are:

Block	Coherence State	Sharer / Owner Bits
B0	Uncached	- - - -

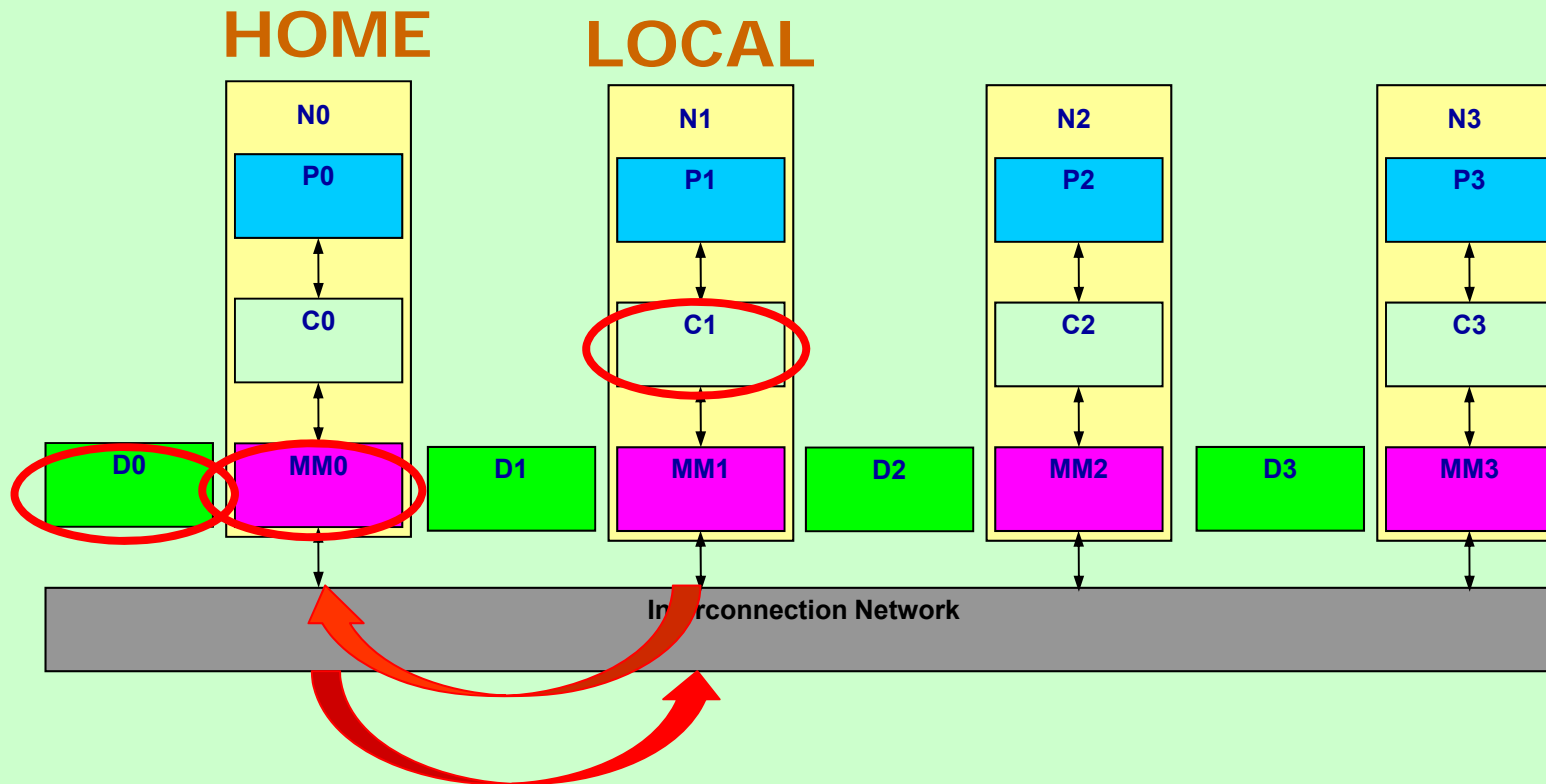
- 1. *Read Miss from local cache (ex. N1):*** Requested data are sent by ***Data Value Reply*** from home memory N0 to local cache C1 and requestor N1 is made the only sharing node. The state of the block is made ***S***.

Block	Coherence State	Sharer / Owner Bits
B0	Shared	0 1 0 0

- 2. *Write Miss from local cache (ex. N1):*** Requested data are sent by ***Data Value Reply*** from home memory N0 to local cache C1 and N1 becomes the owner node. The block is made ***M*** to indicate that the only valid copy is cached. Sharer bits indicate the identity of the **owner** of the block.

Block	Coherence State	Sharer / Owner Bits
B0	Modified	0 1 0 0

Read/Write Miss from local node N1 to home N0 + Data Value Reply from home N0 to local cache C1



Shared State

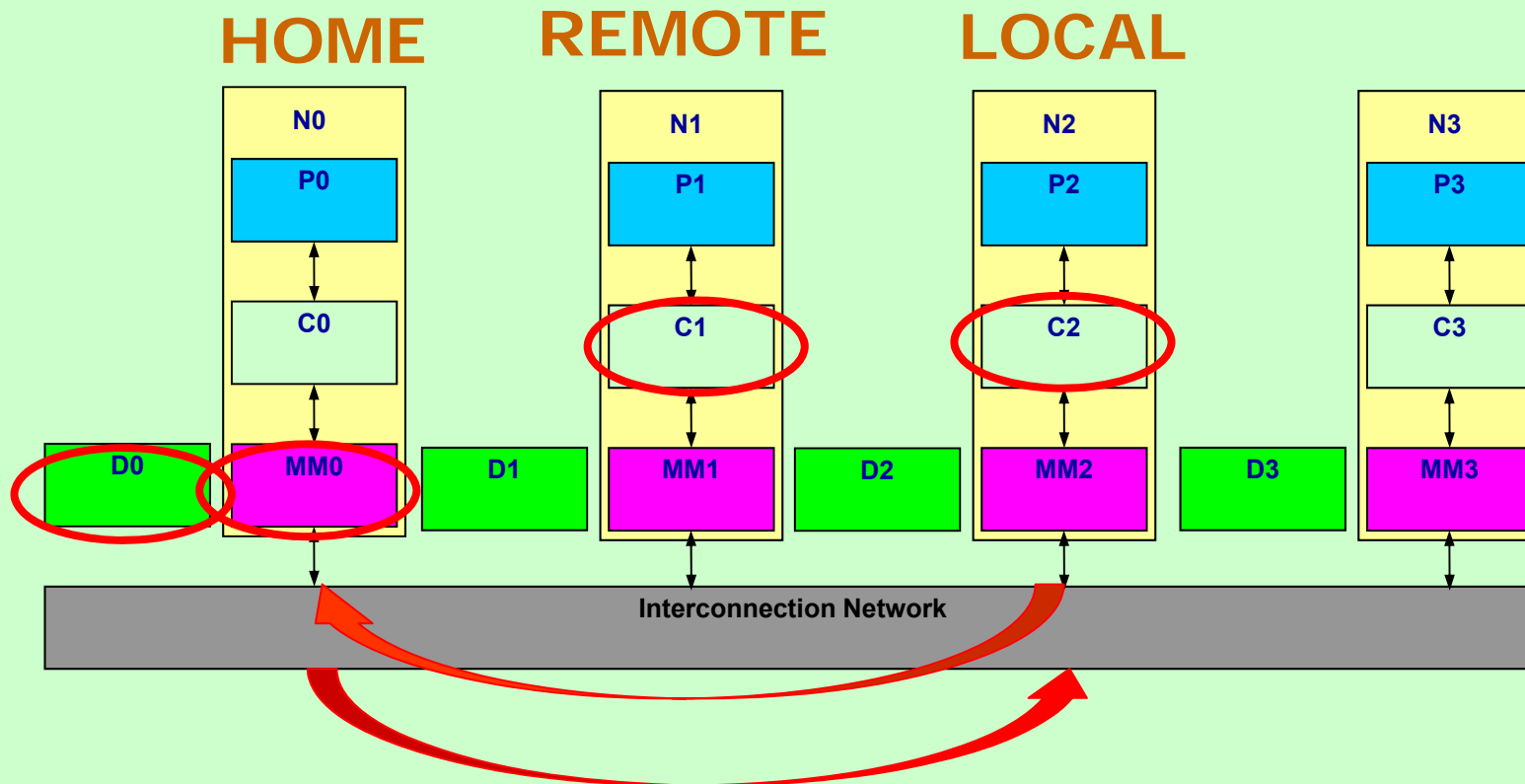
- When a directory block B0 in home N0 is in the ***shared state***, the memory value is up to date, we can have:

Block	Coherence State	Sharer / Owner Bits
B0	Shared	0 1 0 0

- 1. Read Miss from local cache (ex. N2):*** Requested data are sent by ***Data Value Reply*** from home memory N0 to local cache C2 and the requestor (ex. N2) is added to the Sharer bits. The state of the block stays **S**.

Block	Coherence State	Sharer / Owner Bits
B0	Shared	0 1 1 0

Read Miss from local node N2 to home N0 +
Data Value Reply from home N0 to local cache C2



Shared State

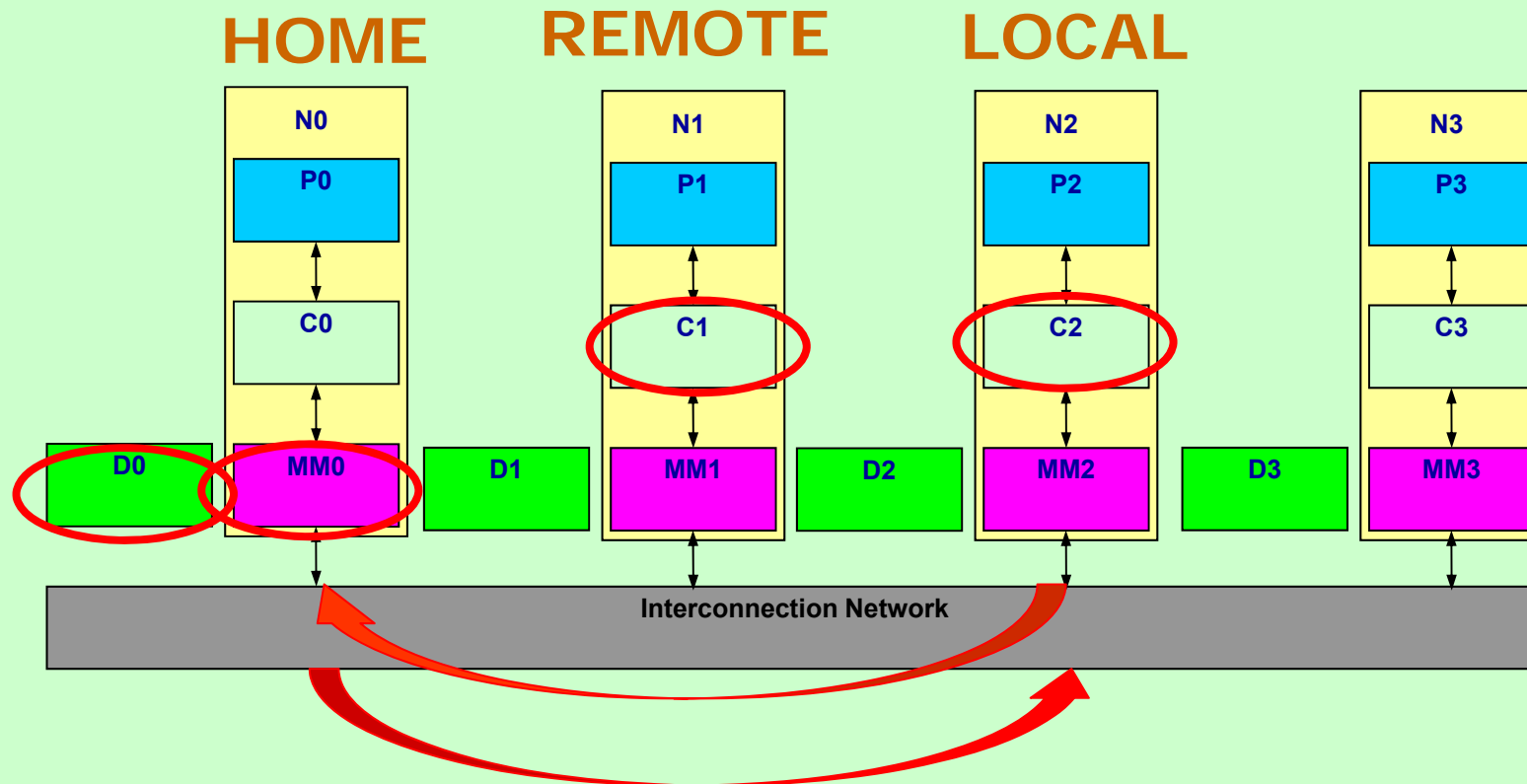
- When a directory block B0 in home N0 is in the ***shared state***, the memory value is up to date, we can have:

Block	Coherence State	Sharer / Owner Bits
B0	Shared	0 1 0 0

- 2. Write Miss from local cache (ex. N2):** Requested data are sent by **Data Value Reply** from home memory N0 to local cache C2. **Invalidate** messages are sent from home N0 to remote sharer(s) (P1) and bits are set to the identity of requestor (P2 **owner**). The state of the block becomes **M**.

Block	Coherence State	Sharer / Owner Bits
B0	Modified	0 0 1 0

Write Miss from local node N2 to home N0 +
Data Value Reply from home N0 to local cache C2
Invalidate from home N0 to remote sharer P1



Modified State

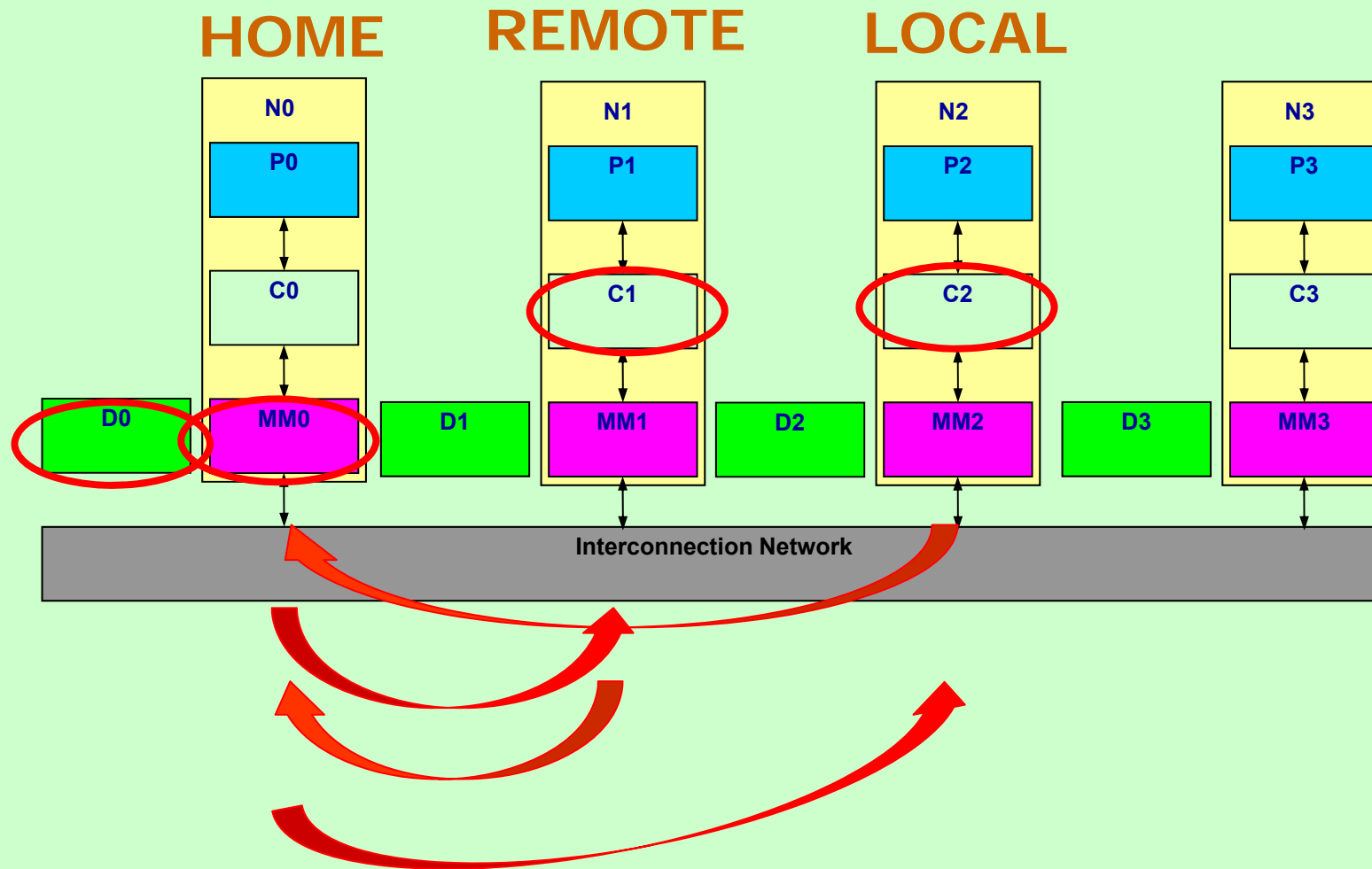
- ❑ When a directory block B0 in home N0 is in the **M state**, the current value of the block is held *in the cache* of the **owner** processor (ex. P1) identified by the sharer bits, so 3 possible requests can occur:

Block	Coherence State	Sharer / Owner Bits
B0	Modified	0 1 0 0

- 1. Read Miss from local cache (ex. N2):** To the owner node P1 is sent a **Fetch** message, causing state of the block in the owner's cache to transition to **S** and the owner send data to the home directory (through **Data Write Back**); data written to home memory are sent to requesting cache N2 by **Data Value Reply**. The identity of the requesting processor (P2) is added to the Sharers set, which still contains the identity of the processor that was the owner (since it still has a readable copy). Block state in directory is set to **S**.

Block	Coherence State	Sharer / Owner Bits
B0	Shared	0 1 1 0

Read Miss from local node N2 to home N0 +
Fetch from home N0 to the remote owner N1 +
Data Write Back from owner N1 to home N0 +
Data Value Reply from home N0 to local cache C2



Modified State

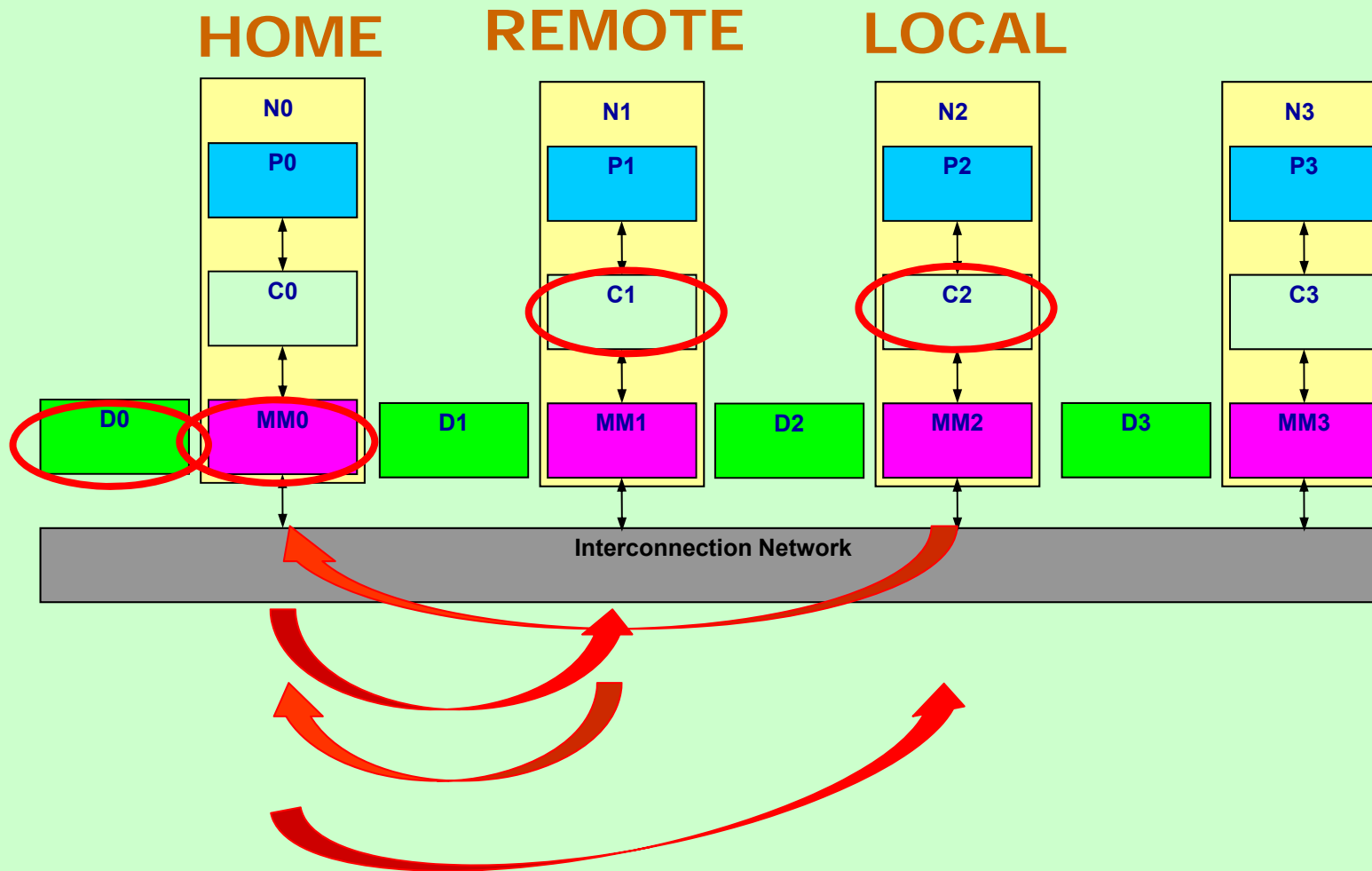
- 2. Data Write-Back from remote owner:** The owner cache is replacing the block and therefore must write it back to home. This make the memory copy up to date (the home dir. becomes the **owner**), the block becomes **U**, and the Sharer set is empty.

Block	Coherence State	Sharer / Owner Bits
B0	Uncached	- - - -

- 3. Write Miss local cache (ex. N2):** A **Fetch/Inv.** msg is sent to the **old owner** (ex. N1) causing to **invalidate** the cache block and the cache C1 to send data to the home directory (**Data Write Back**), from which the data are sent to the requesting node N2 (**Data Value Reply**), which becomes the **new owner**. Sharer is set to the identity of the new owner, and the state of the block remain **M (but owner changed)**

Block	Coherence State	Sharer / Owner Bits
B0	Modified	0 0 1 0

Write Miss from local node N2 to home N0 +
Fetch/Inv. from home N0 to the remote owner N1 +
Data Write Back from owner N1 to home N0 +
Data Value Reply from home N0 to local cache C2



Directory FSM

□ *State machine for dir. requests for each memory block*

