ECE411: Computer Organization and Design

Lecture 17: Cache Coherence

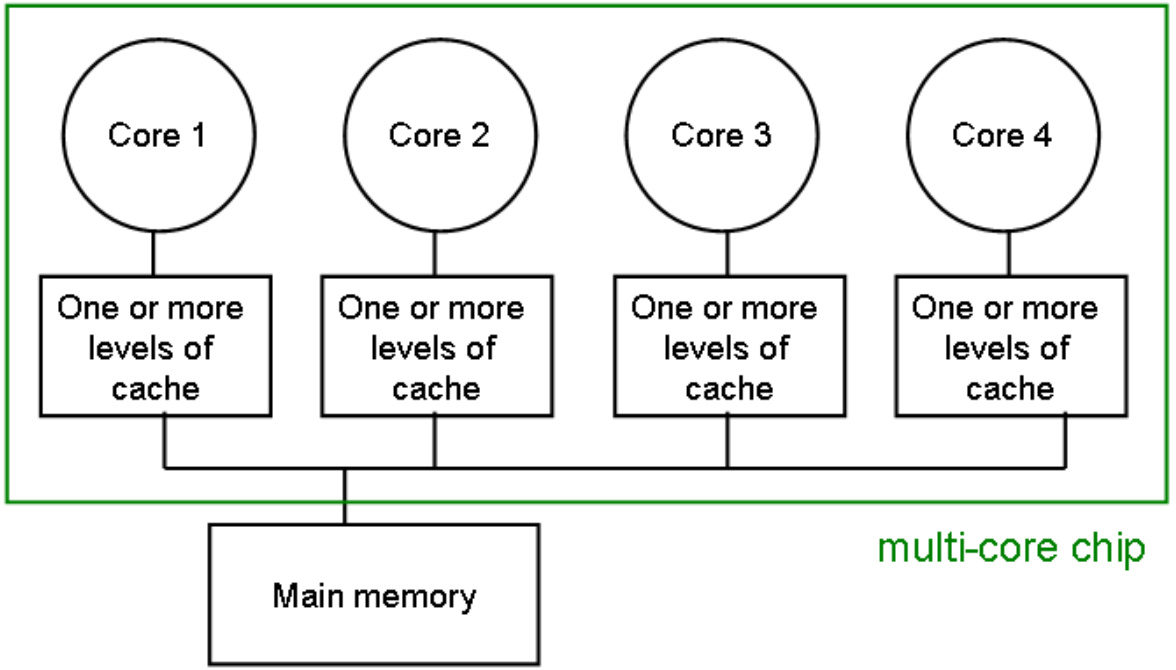
Rakesh Kumar

ILLINOIS

Cache Coherence Problem

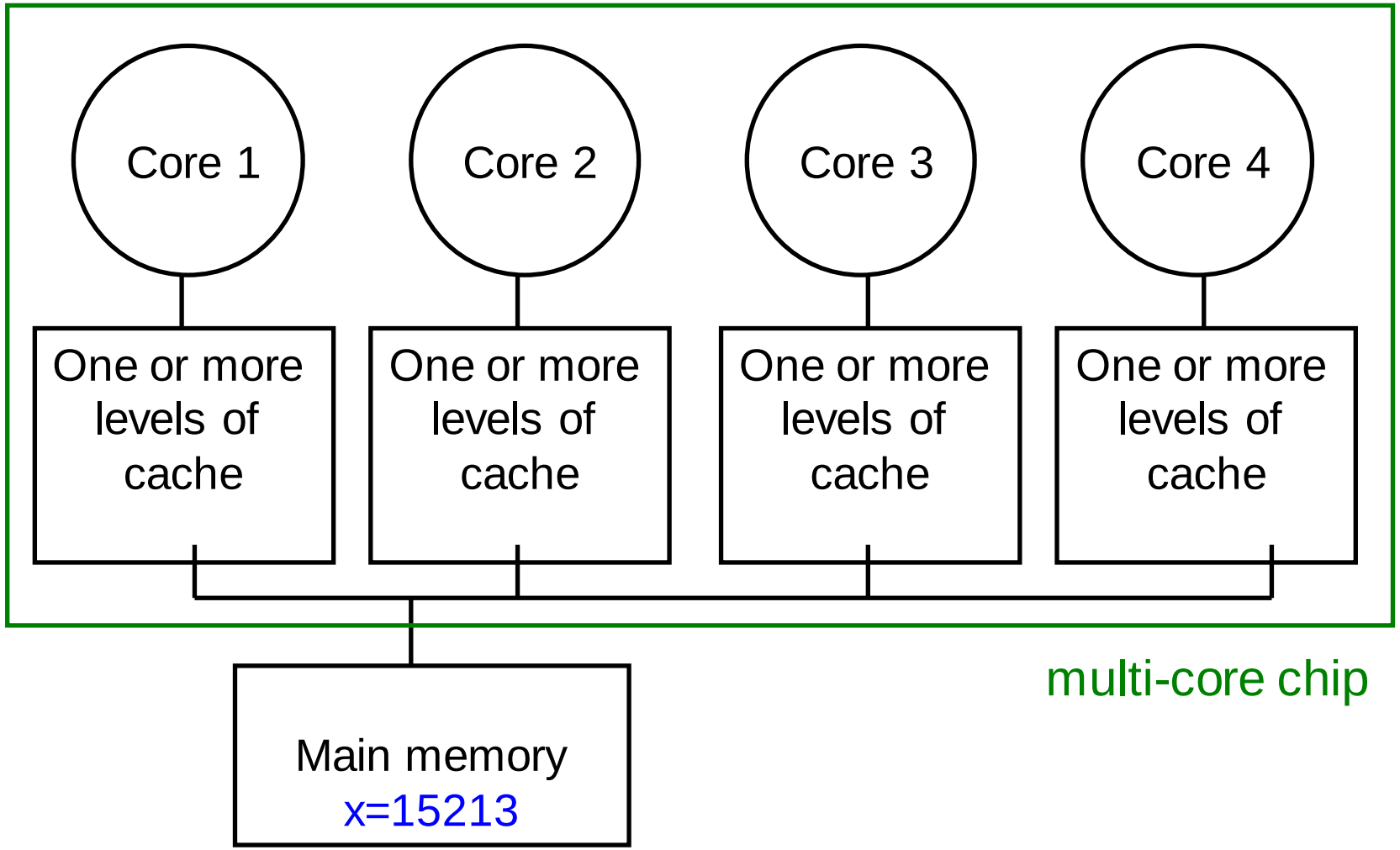
Q: Since we have private caches, how to keep the data consistent across caches?

- each core should perceive the memory as a monolithic array, shared by all the cores



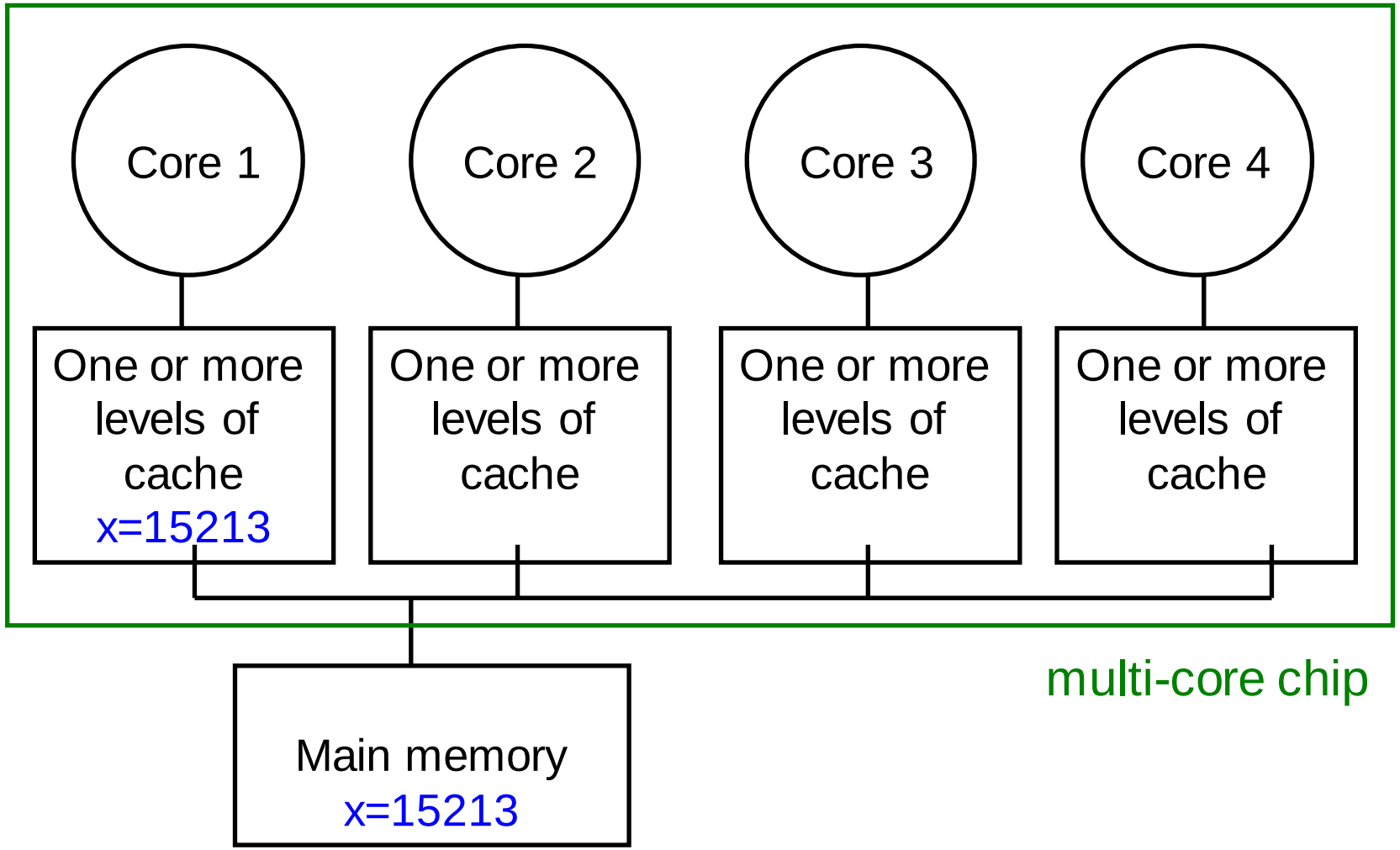
Cache Coherence Problem

- suppose variable x initially contains 15213



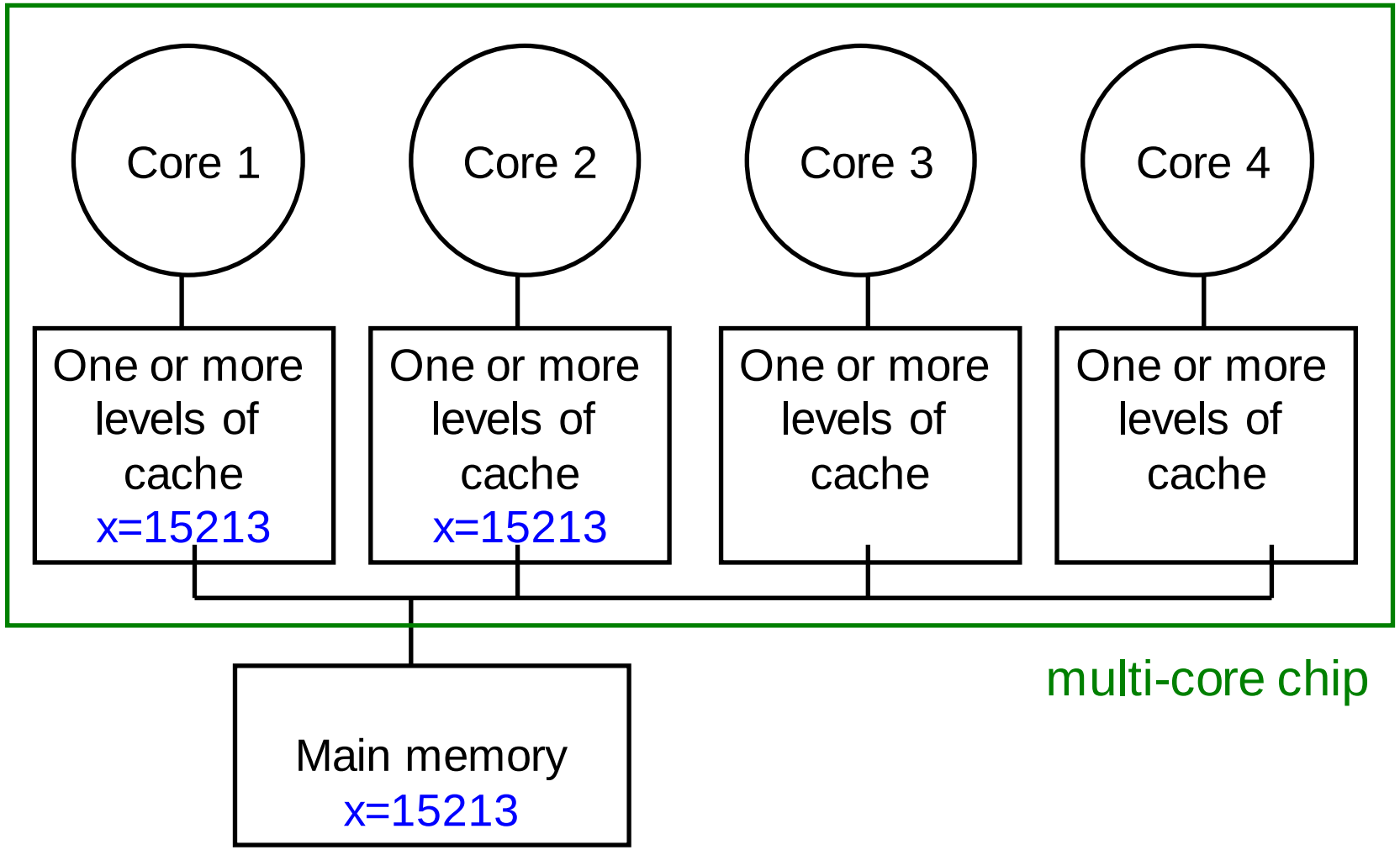
Cache Coherence Problem

- core 1 reads x



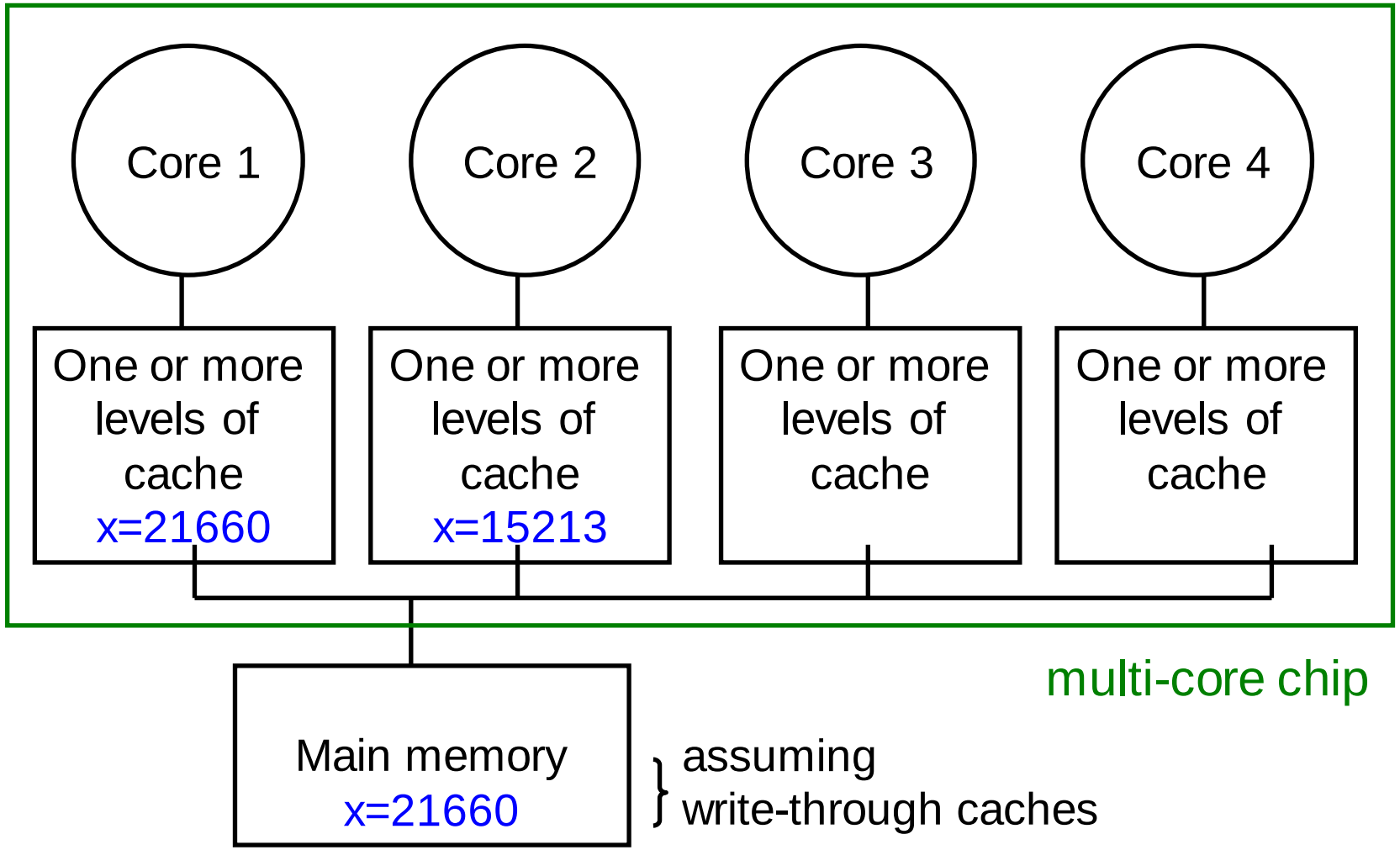
Cache Coherence Problem

- core 2 reads x



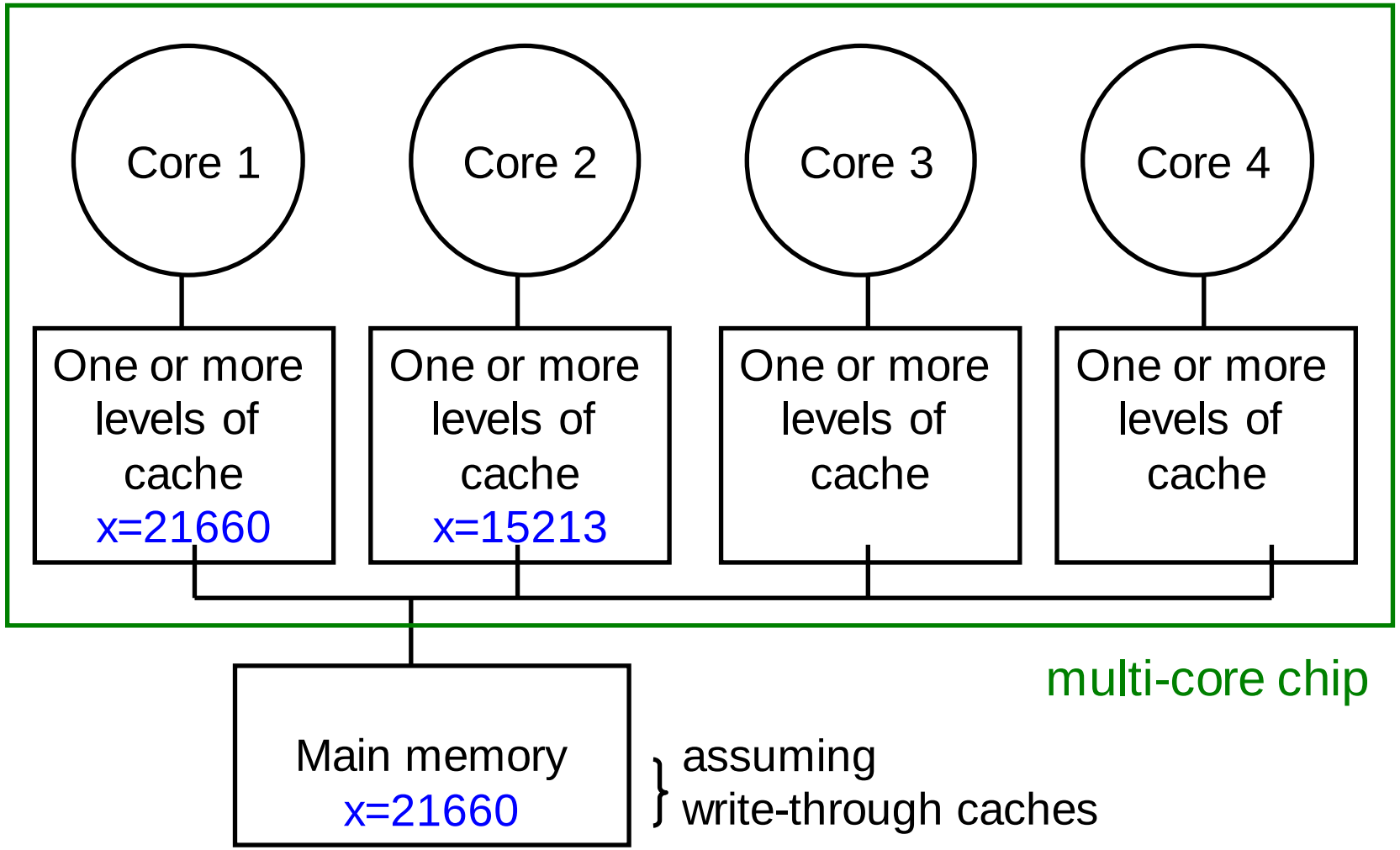
Cache Coherence Problem

- core 1 writes to x, setting it to 21660



Cache Coherence Problem

- core 2 attempts to read x… gets a stale copy



Solutions for Cache Coherence Problem

- General problem w/ multiprocessors, not limited just to multi-core
- Many solution algorithms, coherence protocols, etc.
  - A simple solution: invalidation-based protocol with snooping
  - MSI, MESI (Modified, Exclusive, Shared, Invalid)

Cache Coherence Definition

- A memory system is coherent if
  1. A read R from address X on processor P1 returns the value written by the most recent write W to X on P1 if no other processor has written to X between W and R.

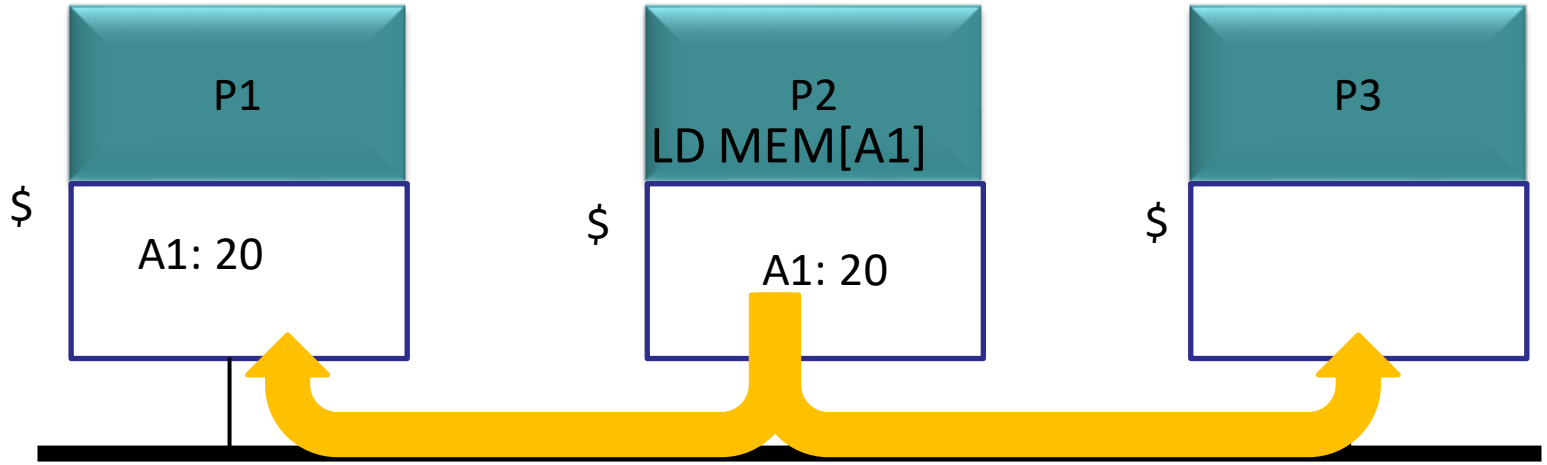
  2. If P1 writes to X and P2 reads X after a sufficient time, and there are no other writes to X in between, P2's read returns the value written by P1's write.

  3. Writes to the same location are serialized: two writes to location X are seen in the same order by all processors.

Cache Coherence Definition

- Property 1. preserves program order
  - It says that in the absence of sharing, each processor behaves as a uniprocessor
- Property 2. says that any write to an address must eventually be seen by all processors
  - If P1 writes to X and P2 keeps reading X,
    P2 must eventually see the new value
- Property 3. preserves causality
  - Suppose X starts at 0. Processor P1 increments X and processor P2 waits until X is 1 and then increments it to 2. Processor P3 must eventually see that X becomes 2.
  - If different processors could see writes in different order, P2 can see P1's write and do its own write, while P3 first sees the write by P2 and then the write by P1. Now we have two processors that will forever disagree about the value of A.

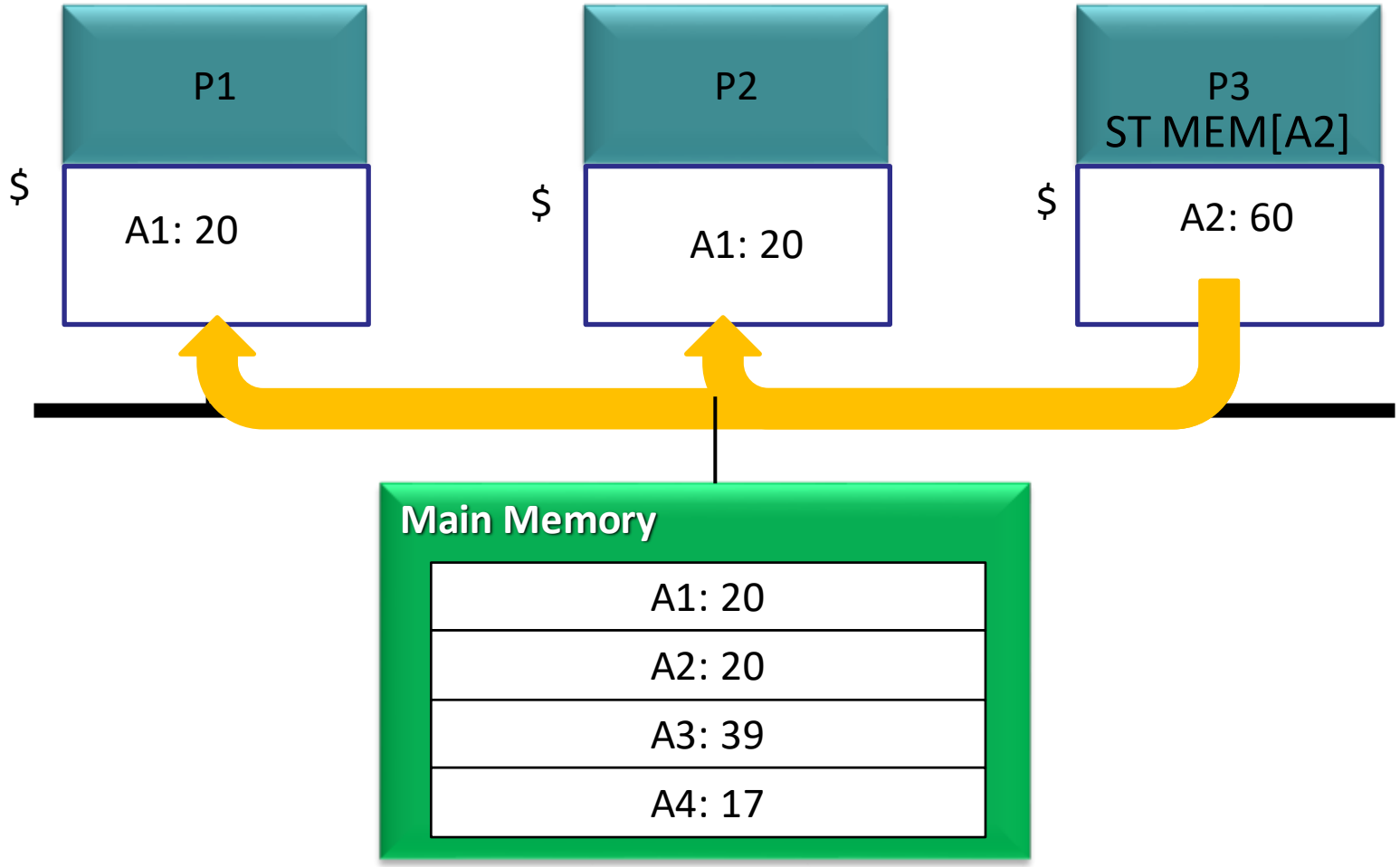
Snooping



Main Memory

A1: 10
A2: 20
A3: 39
A4: 17

Snooping


P1
$
A1: 20
P2
$
A1: 20
P3
ST MEM[A2]
$
A2: 60
Main Memory
A1: 20
A2: 20
A3: 39
A4: 17

Snooping

- Typically used for bus-based (SMP) multiprocessors
  - Serialization on the bus used to maintain coherence property 3

- Two flavors
  - Write-update (write broadcast)
    - A write to shared data is broadcast to update all copies
    - All subsequent reads will return the new written value (property 2)
    - All see the writes in the order of broadcasts on bus == one order seen by all (property 3)
  - Write-invalidate
    - Write to shared data forces invalidation of all other cached copies
    - Subsequent reads miss and fetch new value (property 2)
    - Writes ordered by invalidations on the bus (property 3)

Update vs. Invalidate

- A burst of writes by a processor to one address
  - Update: each sends an update
  - Invalidate: possibly only the first invalidation is sent

- Writes to different words of a block
  - Update: update sent for each word
  - Invalidate: possibly only the first invalidation is sent

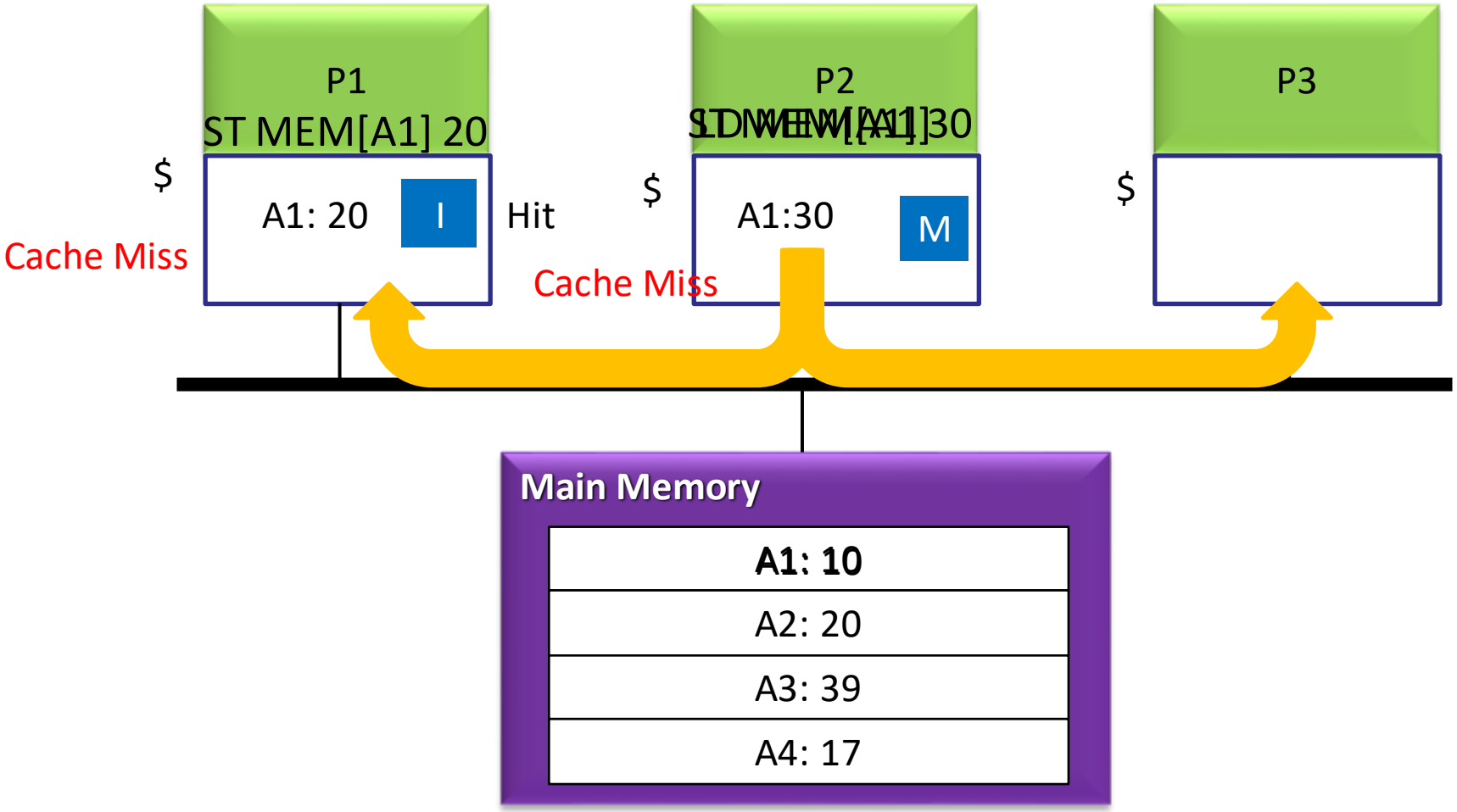
- Producer-consumer communication latency
  - Update: producer sends an update,
    consumer reads new value from its cache
  - Invalidate: producer invalidates consumer's copy,
    consumer's read misses and has to request the block

- Which is better depends on application
  - But write-invalidate is simpler and implemented in most MP-capable processors today

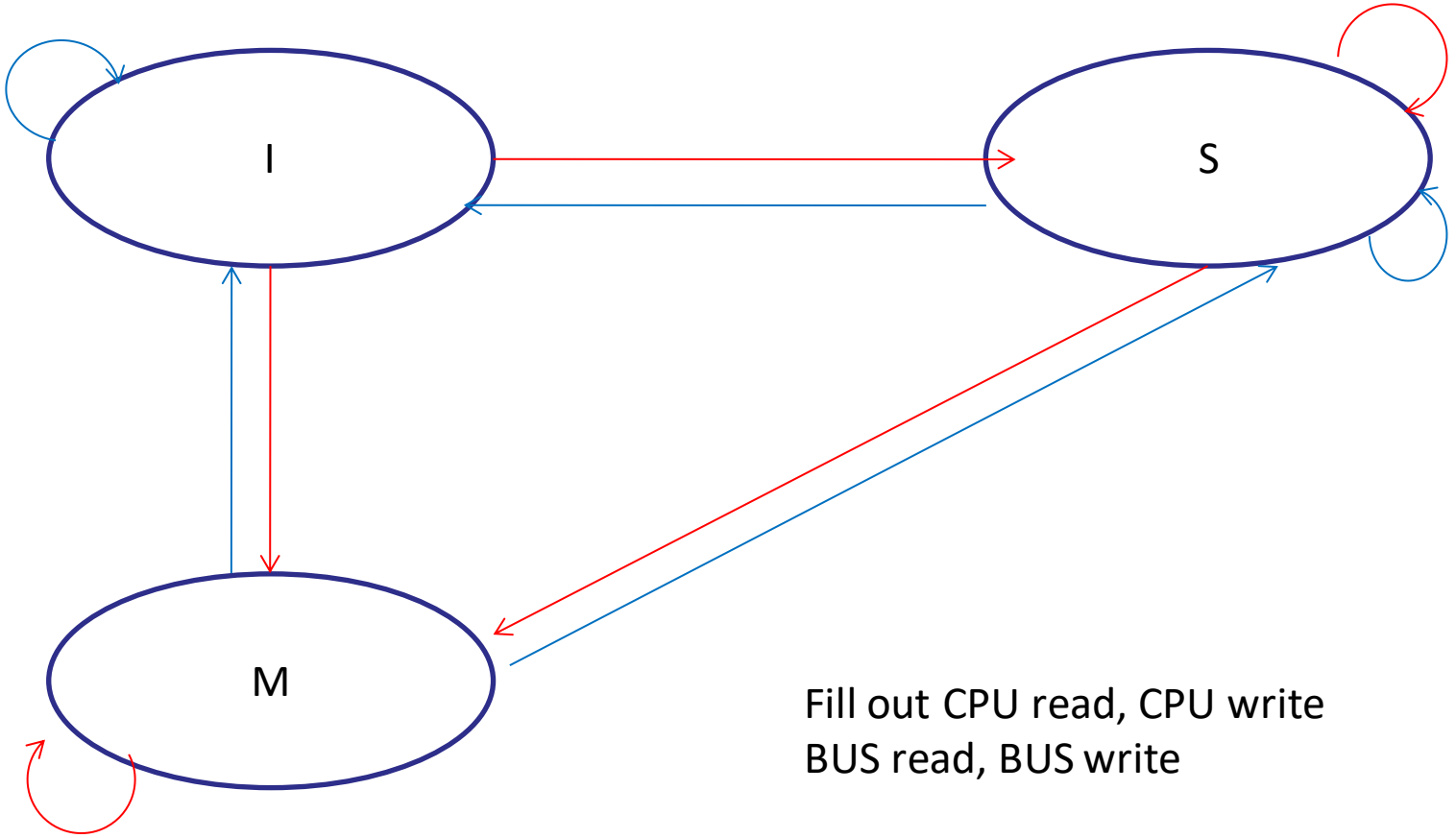
MSI Snoopy Protocol

- State of block B in cache C can be
  - Invalid: B is not cached in C
    - To read or write, must make a request on the bus
  - Modified: B is dirty in C
    - has the block, no other cache has the block, and C must update memory when it displaces B
    - Can read or write B without going to the bus
  - Shared: B is clean in C
    - C has the block, other caches have the block, and C needs not update memory when it displaces B
    - Can read B without going to bus
    - ***To write, must send an upgrade request to the bus***

MSI Example

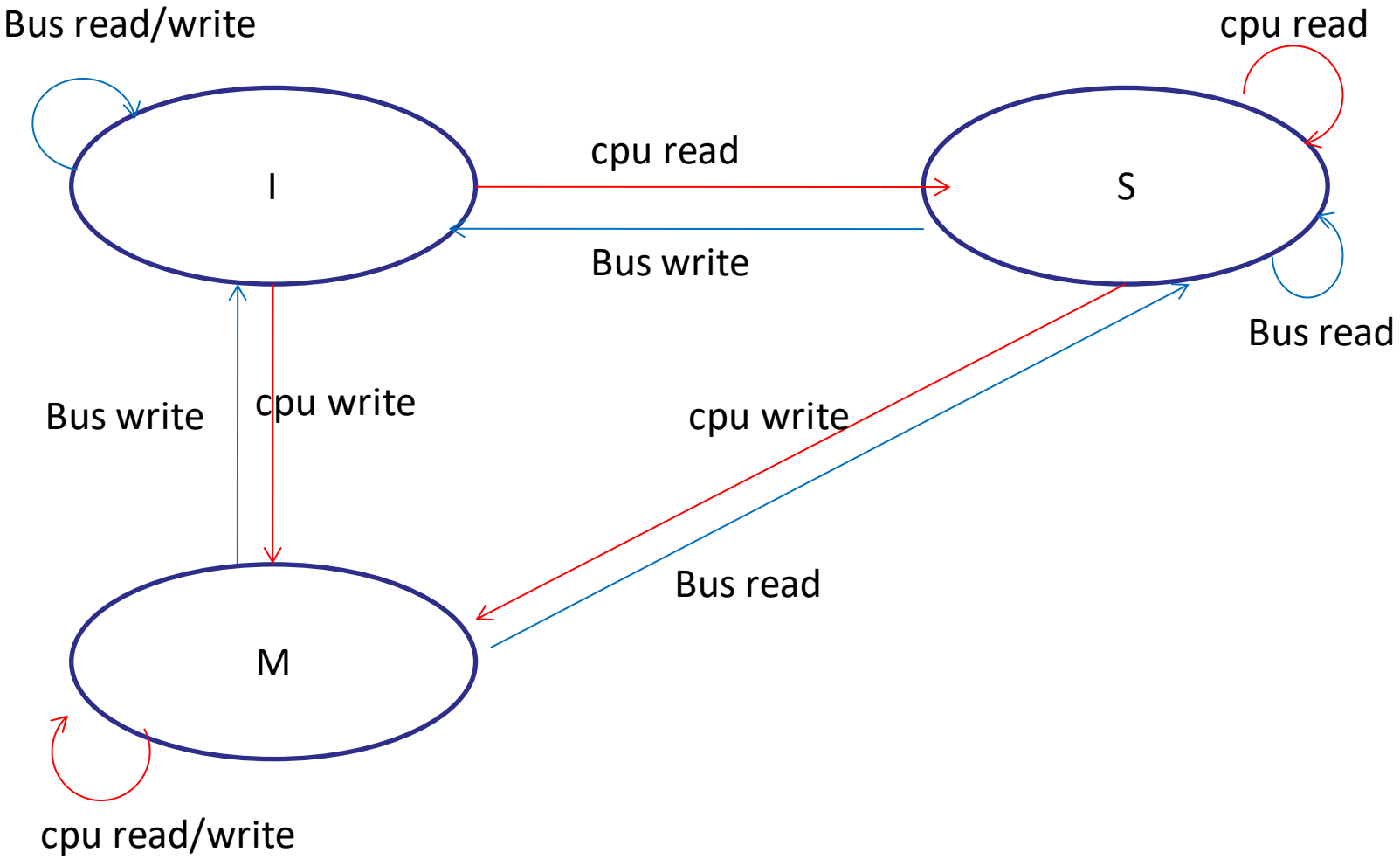

P1
ST MEM[A1] 20
$
A1: 20
I
Hit
Cache Miss
P2
$
A1:30
M
Cache Miss
P3
$
Main Memory
A1: 10
A2: 20
A3: 39
A4: 17

MSI



Fill out CPU read, CPU write
BUS read, BUS write

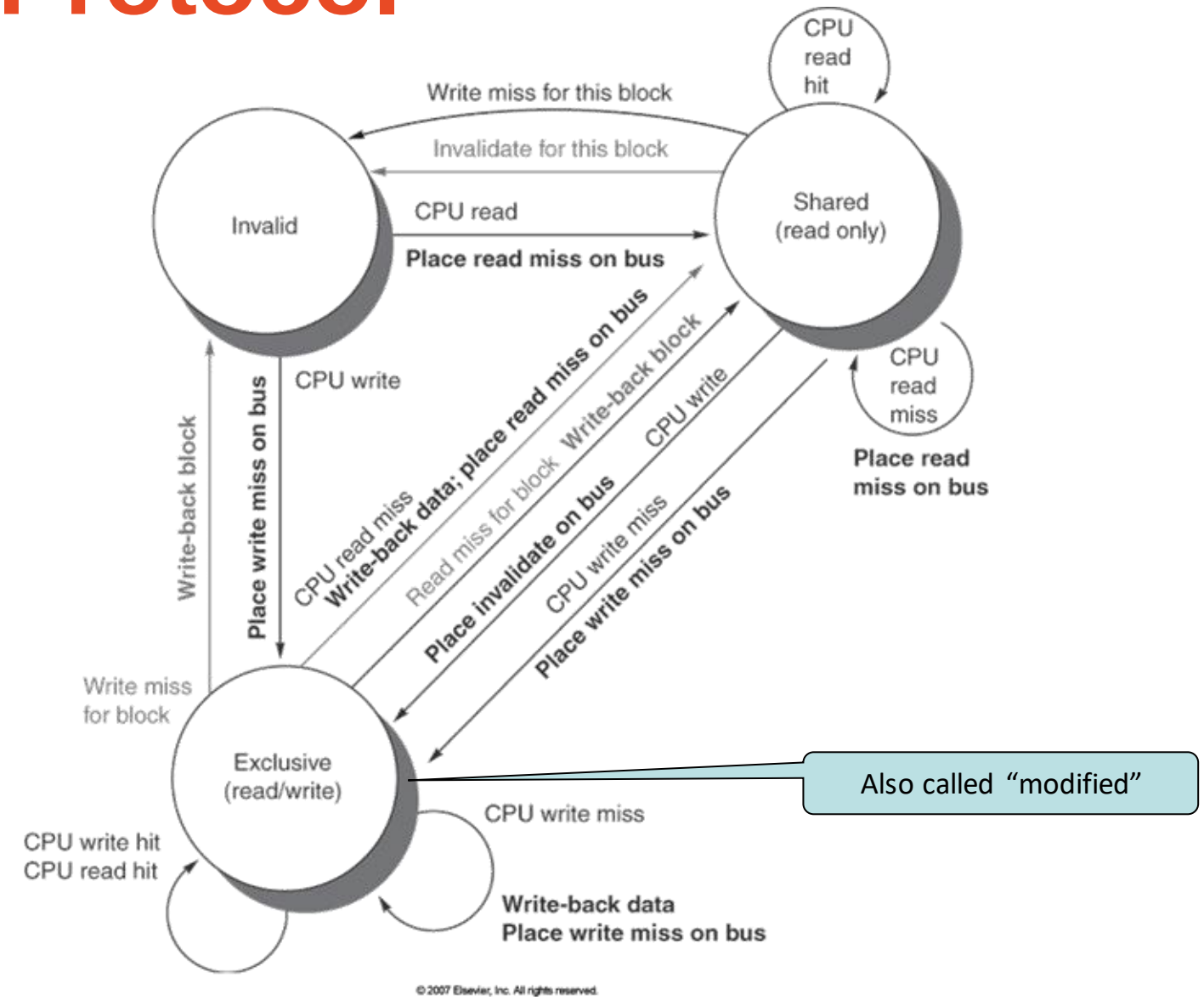
MSI


Bus read/write
cpu read
I
cpu read
S
Bus write
Bus read
Bus write
cpu write
cpu write
Bus read
M
cpu read/write

MSI Snoopy Protocol

My RD/WR:
My action

Someone else



Review Question

- MSI protocol : cache block size is 4B
  - P1 LDB mem[A]
  - P2 STB mem[A]
  - P3 LDB mem[A]
  - P1 STB mem[A]
  - P2 STB mem[A]
  - P3 LDB mem[A]

P1	P2	P3
S	X	X
I	M	X
I	S	S
M	I	I
I	M	I
I	S	S

Cache to Cache Transfer

- Problem
  - P1 has block B in M state
  - P2 wants to read B, puts a RdReq on bus
  - Who should serve the data?
    - If P1 does nothing, memory will supply the data to P2

Cache to Cache Transfer

- Intervention works if some cache has data in M state
  - Nobody else has the correct data, clear who supplies the data

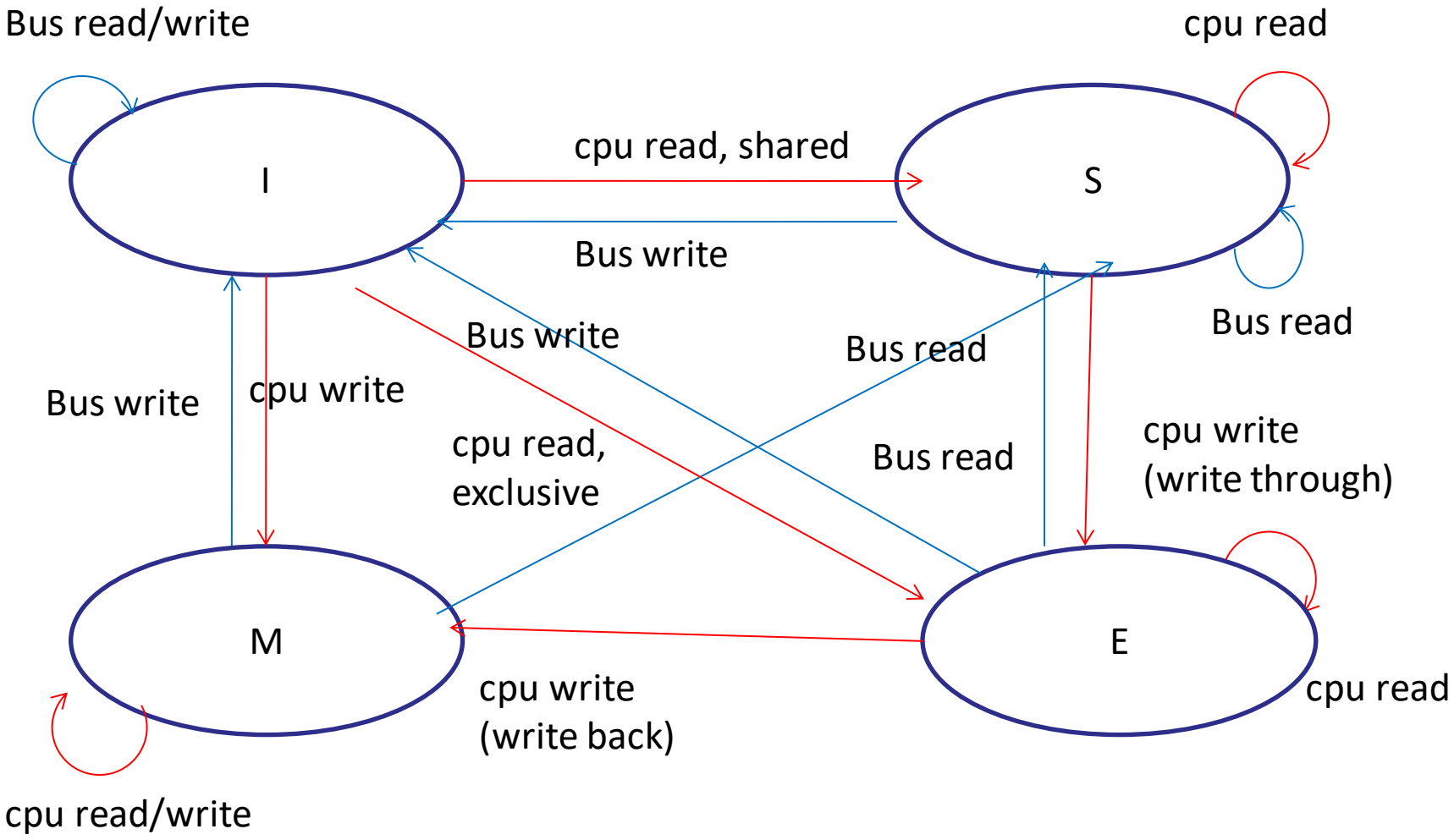
- What if a cache has requested data in S state
  - There might be others who have it, who should supply the data?
  - Solution 1: let memory supply the data
  - Solution 2: whoever wins arbitration supplies the data
  - Solution 3: A separate state similar to S that indicates there are maybe others who have the block in S state, but if anybody asks for the data, we should supply it

MESI Protocol

- New state: exclusive
  - data is clean
  - but I have the only copy (except memory)

- Benefit: bandwidth reduction
  - No broadcasting from E→ M when updating the data because I have the only copy

MESI (Illinois Protocol)



MESI Example

- MESI protocol : cache block size is 4B
  - P1 LDB mem[A]
  - P1 STB mem[A]
  - P3 LDB mem[A]
  - P1 STB mem[A]
  - P2 STB mem[A]
  - P2 STB mem[A]
  - P3 LDB mem[A]

P1	P2	P3
E	X	X
M	X	X
S	X	S
E	X	I
I	M	I
I	M	I
I	S	S

MOESI

- M: Modified (dirty, unique)
  - I have the only copy, and it's dirty (memory is stale)
- O: Owned (dirty, shared)
  - I have the most up-to-date copy, others may have copies, too, but I am responsible for sourcing data
- E: Exclusive (clean, unique)
  - I have the only copy (clean)
- S: Shared (clean, shared)
  - Everyone has a clean copy (incl. memory)
- I: Invalid

Optional Slide

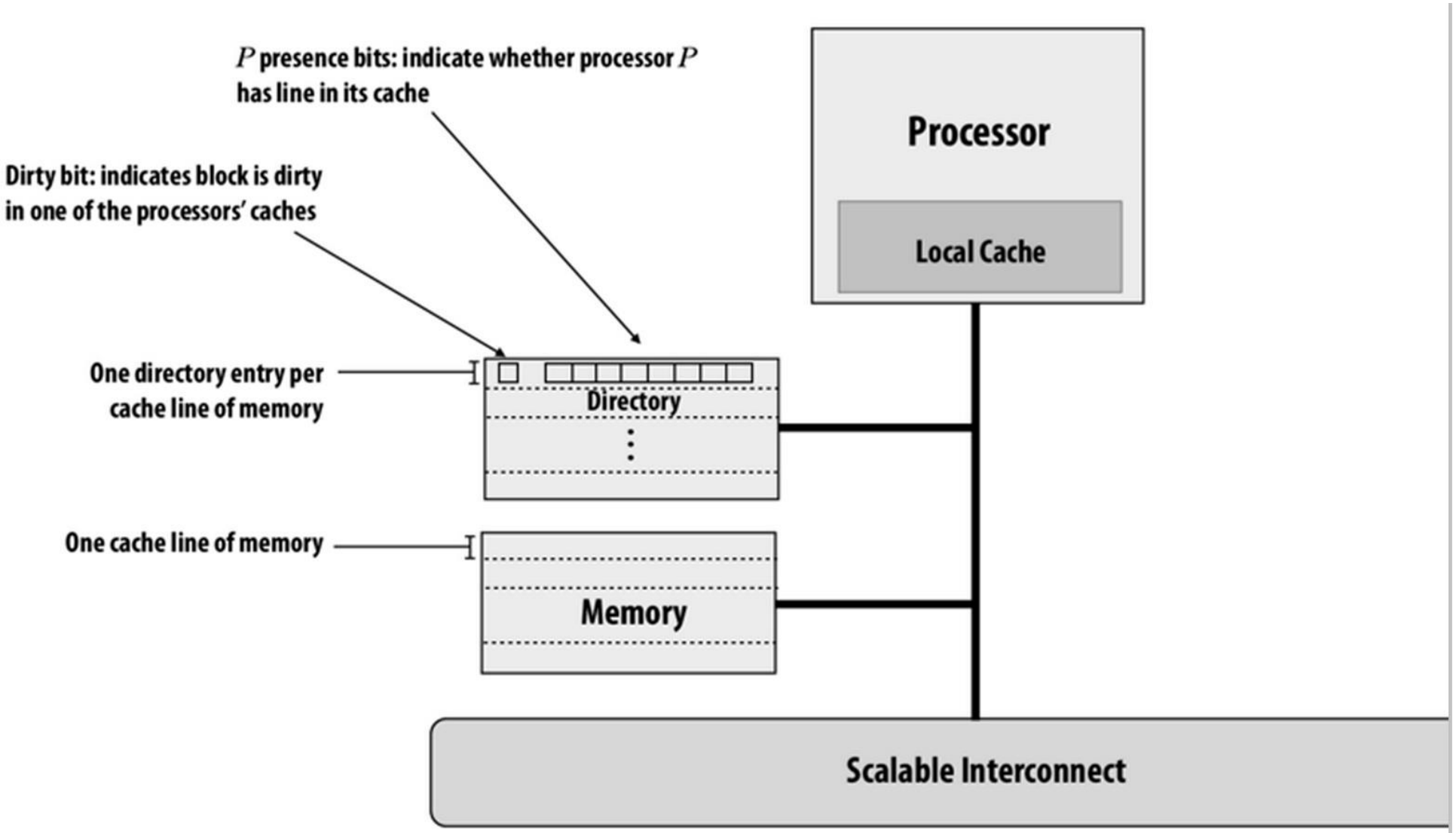
Directory-Based Coherence

- Typically, in distributed shared memory
- For every local memory block, local directory has an entry
- Directory entry indicates
  - Who has cached copies of the block
  - In what state do they have the block

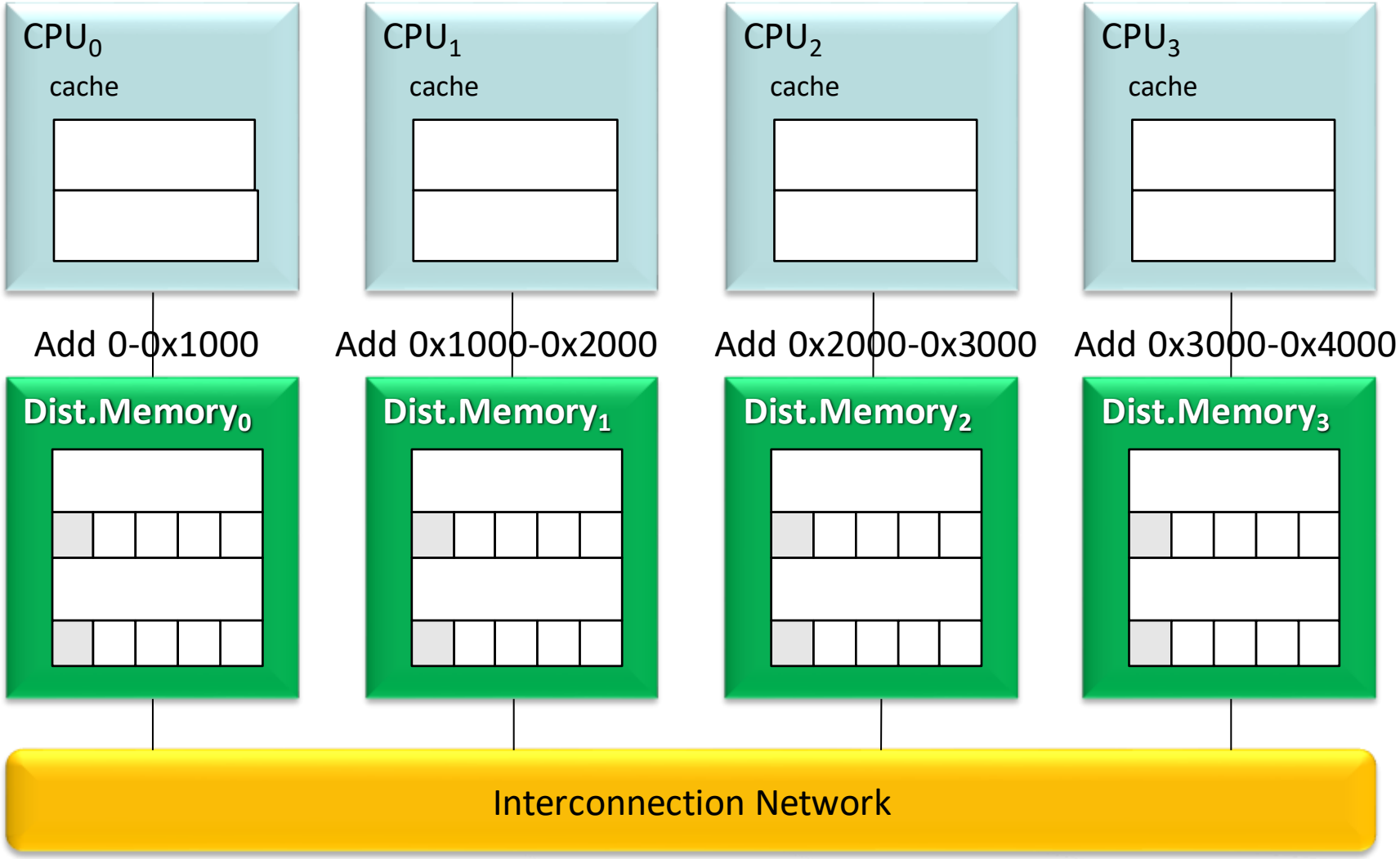
Basic Directory Scheme

- Each entry has
  - One dirty bit (1 if there is a dirty cached copy)
  - A presence vector (1 bit for each node)
    Tells which nodes may have cached copies
- All misses sent to block's home
- Directory performs needed coherence actions
- Eventually, directory responds with data

Basic Directory Scheme



Distributed MSI Example



Read Miss

- Processor P_k has a read miss on block B, sends request to home node of the block
- Directory controller
  - Finds entry for B, checks D bit
  - If D=0
    - Read memory and send data back, set P[k]
  - If D=1
    - Request block from processor whose P bit is 1
    - When block arrives, update memory, clear D bit, send block to P_k and set P[k]

Directory Operation

- Network controller connected to each bus
  - A proxy for remote caches and memories
    - Requests for remote addresses forwarded to home, responses from home placed on the bus
    - Requests from home placed on the bus, cache responses sent back to home node
- Each cache still has its own coherence state
  - Directory is there just to avoid broadcasts and order accesses to each location

Shared Memory Performance

- Another “C” for cache misses
  - Still have Compulsory, Capacity, Conflict
  - Now have Coherence, too
    - We had it in our cache and it was invalidated
- Two sources for coherence misses
  - True sharing
    - Different processors access the same data
  - False sharing
    - Different processors access different data, ***but they happen to be in the same block***

False Sharing

- X and Y are in the same cache block, P1 and P2 read/write X & Y

time	P1	P2	Cache hit,miss ? /Coherence state (P1, P2) [M,S,I]
1	Read X		Miss (S, X)
2		Read X	Miss (S,S)
3	Write X		Hit (M, I)
4		Read X	Miss (S, S)

True Sharing

time	P1	P2	Cache hit,miss ? (P1,P2)
1	Read X		Miss (S,X)
2		Read Y	Miss (S,S)
3	Write X		Hit (M, I)
4		Read Y	Miss (S,S)

False Sharing

Code Example

```
struct foo {
    int x;
    int y;
};

static struct foo f;

/* The two following functions are running
concurrently: */

int sum_a(void)
{
    int s = 0;
    for (int i = 0; i < 1000000; ++i)
        s += f.x;
    return s;
}
```

```
void inc_b(void)
{
    for (int i = 0; i < 1000000; ++i)
        ++f.y;
}
```

Takeaways

- Multi-core Architecture
- SMT (Hyper-Threading)
- Memory Hierarchy for Multi-core Processor
- Cache Coherence
- Snooping Technique
- Update vs. Invalidate
- MSI vs. MESI
- Directory-based Cache Coherency
- False Sharing

Announcement

- Next Lecture: SIMD and GPU
  - Ch. 4.3 – 4.4 (HP1)
- MP
  - MP4 CP1 Lab Session: Wednesday 10/18/2023
- 2nd Midterm Exam
  - 7-9pm next Thursday, 10/26/2023