EE 156/CS140: Advanced Topics in Computer Architecture

Spring 2023
Tufts University

Instructor: Prof. Mark Hempstead
mark@ece.tufts.edu

Lecture 4: Intro to Cache Coherence, Memory Consitancy

Unit 1: The Memory Hierarchy

Unit 1: The Memory Hierarchy (3-4 weeks)

- **Introduction and Performance Metrics [Chapter 1]**
- **Review of Basic Caches and Set Associativity [Appendix B]**
- **Advanced Cache Optimization Techniques and Replacement policies [Ch 2]**
  - **Cache Miss Types (3 Cs)**
  - **Cache Hierarchies**
  - **10 Optimizations (From Textbook)**
- **Prefetching [Extra Slides]**
- **Memory consistency and Cache coherence [Chapter 5]**
- **Software interfaces and memory consistency**
- **Transactional memory (paper)**
- Review of Virtual Memory and TLBs [Appendix B]
- Advanced Virtual Memory [SLCA: Bhattacharjee and Lustig]
- New Non-Volatile Memory (NVM) technologies

- **Textbook Reading: Read Sections 5.1, 5.2, 5.5, 5.6**
  Bonus Slides Section 5.3, 5.4

BRIEF INTRODUCTION TO CACHE COHERENCE [5.1, 5.2]

First, Uniprocessor Concurrency

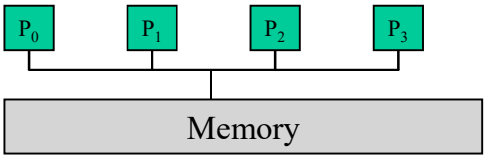
- **Software "thread":** Independent flows of execution
  - "Shared" state: global variables, heap, etc.
  - Threads generally share the same memory space
    - "Process" like a thread, but different memory space
  - "private" per-thread state
    - Context state: PC, registers
    - Stack (per-thread local variables)
  - Java has thread support built in, C/C++ using a thread library
- Generally, system software (the O.S.) manages threads
  - "Thread scheduling", "context switching"
  - In single-core system, all threads share the one processor
    - Hardware timer interrupt occasionally triggers O.S.
    - Quickly swapping threads gives illusion of concurrent execution
  - Much more in an operating systems course

Multithreaded Programming Model

- Programmer explicitly creates multiple threads
- All loads & stores to a single **shared memory** space
  - Each thread has a private stack frame for local variables
- A "thread switch" can occur at any time
  - Pre-emptive multithreading by OS
- Common uses:
  - Handling user interaction (GUI programming)
  - Handling I/O latency (send network message, wait for response)
  - **Expressing parallel work via Thread-Level Parallelism (TLP)**
    - This is our focus!

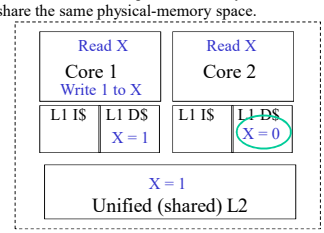
Shared-Memory Multiprocessors

- **Conceptual and older model**
  - The shared-memory abstraction
  - Familiar and feels natural to programmers
  - Older systems did have multiple chips (one core per chip) talking to memory over a common memory bus.
  - Memory access is slow, but local caches in each core allow fast cached access.
  - There is a problem, though.



Cache Coherence in Multicores

- Even if they don't share an external memory, two cores on one die often each share an L2 (but have their own L1), which causes the same problem.
- The issue is a *cache coherence* problem. It only occurs when two processes share the same physical-memory space.

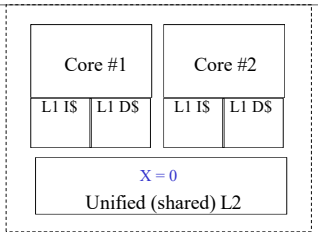


A Coherent Memory System

- The problem: when processors share slow memory, data must move to local caches in order to get reasonable latency and bandwidth. But this can produce unreasonable results.
- The solution is to have *coherency* and *consistency*.
- Coherency – defines *what values* can be returned by a read
  - Roughly speaking, a read of a data item should return the most recently written value.
  - Writes to the same location are serialized (two writes to the same location must be seen in the same order by all cores)
  - The book defines coherency more precisely.
- Consistency – determines *when* a written value will be returned by a read.
  - If core #1 writes mem[100]=1 at t=100 and core #2 reads mem[100] at t=100.00001, it need not see mem[100]=1
  - Perfect global time need not exist.

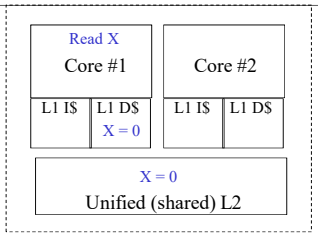
Why not?

Example of Snooping Invalidation



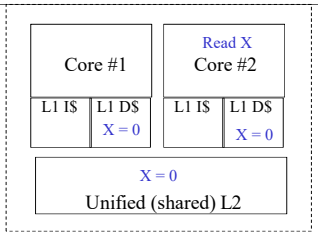
- Start with X=0 in the shared L2.

Example of Snooping Invalidation



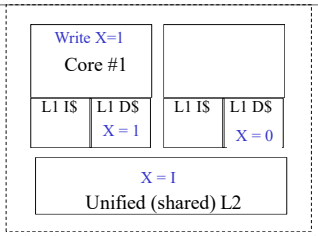
- Core #1 reads X.

Example of Snooping Invalidation



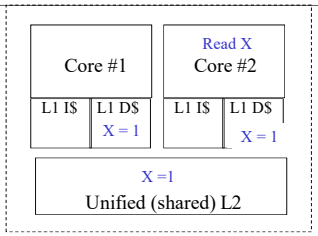
- Next, core #2 also reads X.

Example of Snooping Invalidation



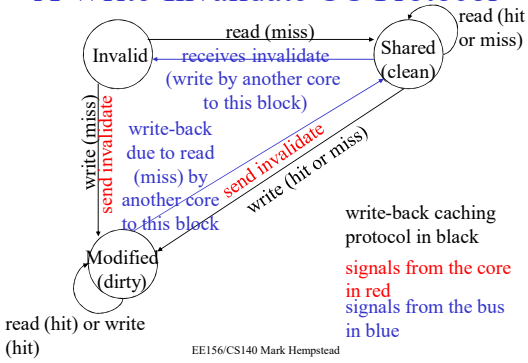
- Now the fun begins. Core #1 writes X=1. The write goes to the bus (even though core #1's L1 is writeback and hit); core #2 snoops the write and invals its L1.

Example of Snooping Invalidation



- Next, core #2 reads X again. The L1 read miss goes to the bus; core #1 sees it and responds with data (which also updates the L2)

A Write-Invalidate CC Protocol

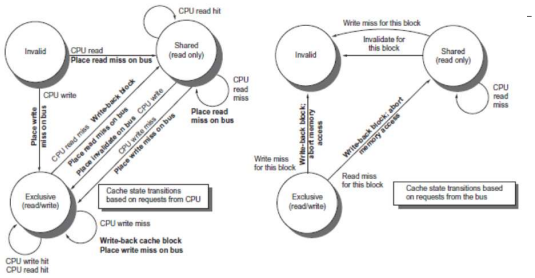


MSI Protocol State Transition Table

State	*This Processor* Load	*This Processor* Store	*Other Processor* Load Miss	*Other Processor* Store Miss
Invalid (I)	Load Miss → S	Store Miss → M	---	---
Shared (S)	Hit	Upgrade Miss → M	---	→ I
Modified (M)	Hit	Hit	Send Data → S	Send Data → I

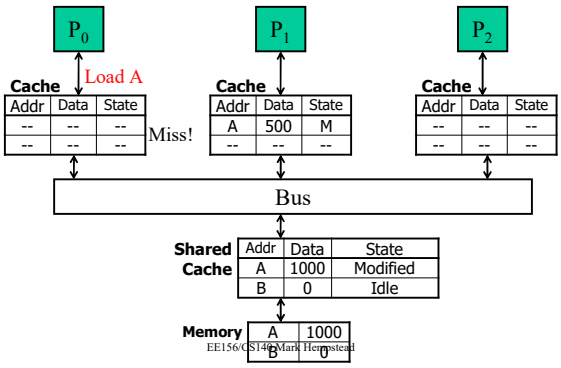
- M → S transition also updates memory
  - After which memory willl respond (as all processors will be in S)

Snoopy Coherence Protocols

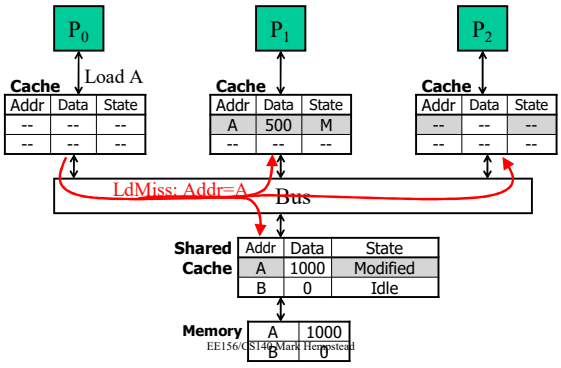


Text Book Diagram

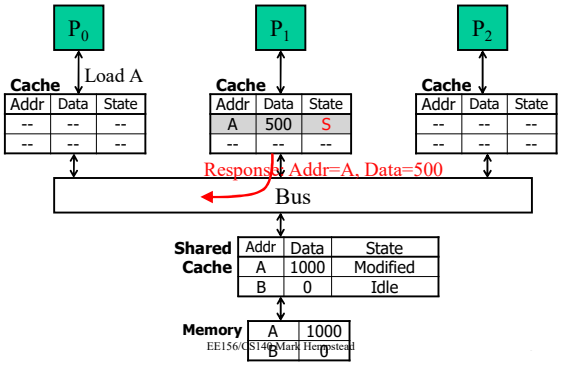
Snooping Example: Step #1



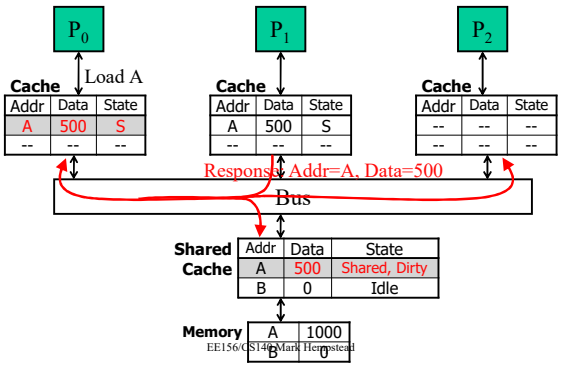
Snooping Example: Step #2



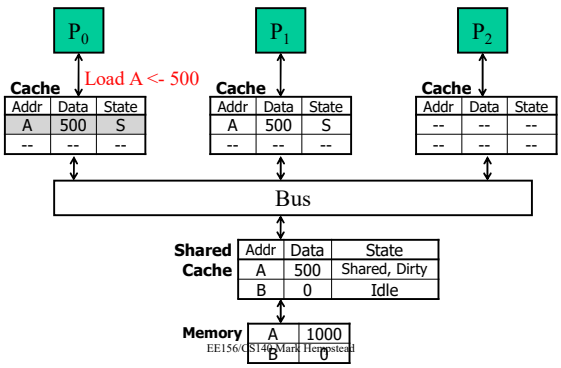
Snooping Example: Step #3



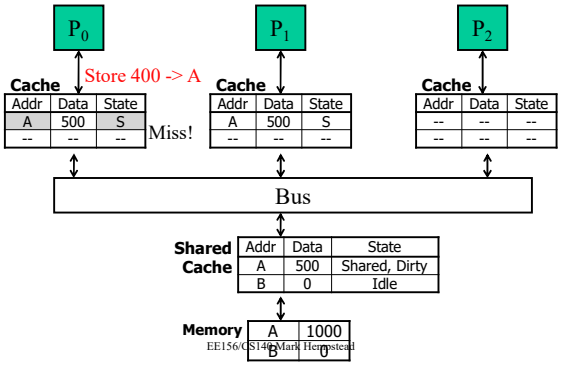
Snooping Example: Step #4



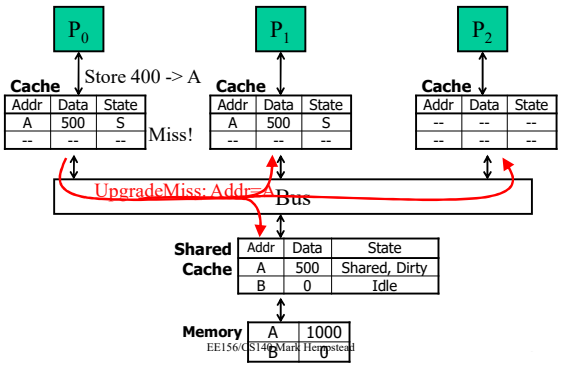
Snooping Example: Step #5



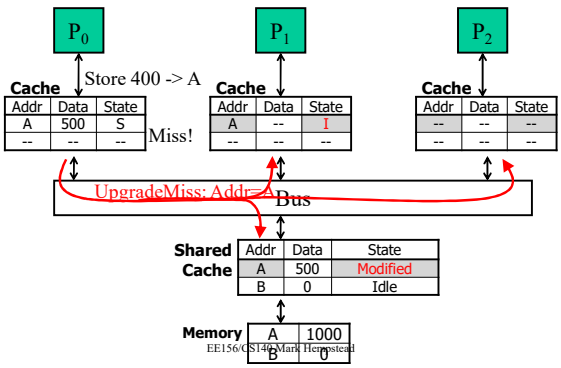
Snooping Example: Step #6



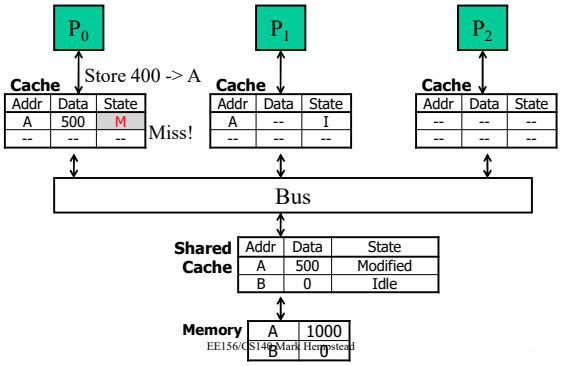
Snooping Example: Step #7



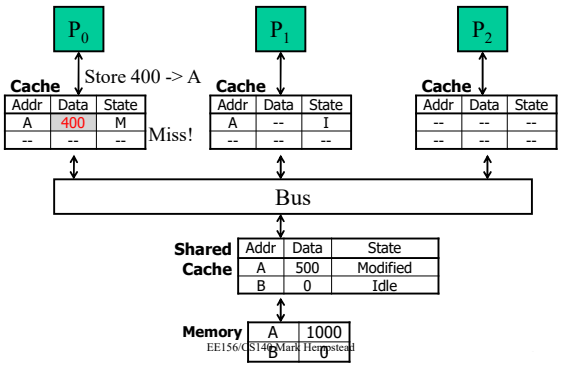
Snooping Example: Step #8



Snooping Example: Step #9

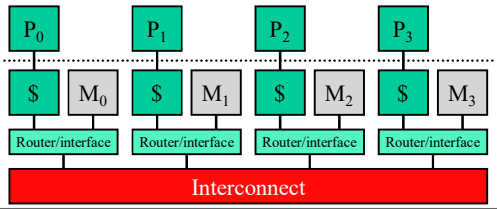


Snooping Example: Step #10



Shared-Memory Multiprocessors

- …but systems actually look more like this
  - Processors have caches
  - Memory may be physically distributed
  - Arbitrary interconnect



Snooping Bandwidth Scaling Problems

- Coherence events generated on…
  - L2 misses (and writebacks)
- Problem#1: **N^2 bus traffic**
  - All N processors send their misses to all N-1 other processors
  - Assume: 2 IPC, 2 Ghz clock, 0.01 misses/insn **per processor**
  - 0.01 misses/insn * 2 insn/cycle * 2 cycle/ns * 64 B blocks
    - = 2.56 GB/s… per processor
    - With 16 processors, that's 40 GB/s! With 128 that's 320 GB/s!!
  - You can use multiple buses… but that complicates the protocol
- Problem#2: **N^2 processor snooping bandwidth**
  - 0.01 events/insn * 2 insn/cycle = 0.02 events/cycle per processor
  - 16 processors: 0.32 bus-side tag lookups per cycle
    - Add 1 extra port to cache tags? Okay
  - 128 processors: 2.56 tag lookups per cycle! 3 extra tag ports?
- Directory-based cache coherency protocols are often used to overcome this limit (covered in chapter 5 but not this course)

MESI Protocol State Transition Table

State	*This Processor* Load	*This Processor* Store	*Other Processor* Load Miss	*Other Processor* Store Miss
Invalid (I)	Miss → **S or E**	Miss → M	---	---
Shared (S)	Hit	Upg Miss → M	---	→ I
Exclusive (E)	Hit	Hit → M	Send Data → S	Send Data → I
Modified (M)	Hit	Hit	Send Data → S	Send Data → I

- Load misses lead to "E" if no other processors is caching the block

MESI

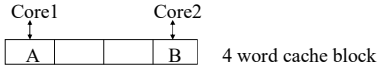
- Each L1 already needs Modified & Valid bits
- MESI: each L1 can hold a line in one of 4 states:
  - Modified: I have this line, and it's dirty.
  - Exclusive: I have a read-only copy, nobody else does
  - Shared: More than one L1 has a read-only copy
  - Invalid: OK, we actually don't hold it at all ☺.
- Why do we need an "invalid" state?
  - So that an empty entry doesn't randomly match a tag.
- The rules:
  - The HN grants an L1 a line in M, E or S.
  - An L1 can upgrade from E to M without telling the HN. The Home Node must take this into account.
  - If the HN grants P1 a line in E, and then P2 requests a load, the HN must downgrade P1 from E to S.
    - What is the consequence?

MESI changes the ISA

- We now need two types of load
  - Regular load, requesting data in S
  - Read for ownership (RFO), requesting data in E.
- RFO is a new instruction.
  - Or a slight modification of the usual load instruction.
- Compiler knows if it's doing i=i+1
  - Issues LD or RFO accordingly

False sharing

- Writes to one word in a multi-word block mean that the full block is invalidated
- Multi-word blocks can also result in false sharing: when two cores are writing to two different variables that happen to fall in the same cache block
  - With write-invalidate false sharing increases cache miss rates



4 word cache block

- Compilers can help reduce false sharing by allocating highly correlated data to the same cache block

SOFTWARE INTERFACES FOR MEMORY CONSISTENCY [5.5, 5.6]

Protecting Shared Data

- Threads are not allowed to update shared data concurrently
  - For correctness purposes
- Accesses to shared data are encapsulated inside *critical sections* or protected via *synchronization constructs (locks, semaphores, condition variables)*
- Only one thread can execute a critical section at a given time
  - Mutual exclusion principle
- A multiprocessor should provide the *correct* execution of synchronization primitives to enable the programmer to protect shared data

34

Supporting Mutual Exclusion

- Programmer needs to make sure mutual exclusion (synchronization) is correctly implemented
  - We will assume this
  - But, correct parallel programming is an important topic
  - Reading: Dijkstra, "Cooperating Sequential Processes," 1965.
    - http://www.cs.utexas.edu/users/EWD/transcriptions/EWD01xx/EWD123.html
    - See Dekker's algorithm for mutual exclusion
- Programmer relies on hardware primitives to support correct synchronization
- If hardware primitives are not correct (or unpredictable), programmer's life is tough
- If hardware primitives are correct but not easy to reason about or use, programmer's life is still tough

35

Sequential Consistency

- Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," IEEE Transactions on Computers, 1979
- A multiprocessor 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

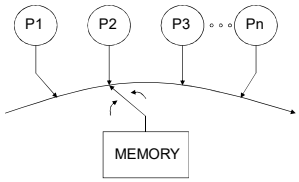
AND

  - the operations of each individual processor appear in this sequence in the order specified by its program
- This is a memory ordering model, or memory model
  - Specified by the ISA

36

Programmer's Abstraction

- Memory is a switch that services one load or store at a time from any processor
- All processors see the currently serviced load or store at the same time
- Each processor's operations are serviced in program order



37

Sequentially Consistent Operation Orders

- Potential correct global orders (all are correct):

- A B X Y
- A X B Y
- A X Y B
- X A B Y
- X A Y B
- X Y A B

- Which order (interleaving) is observed depends on implementation and dynamic latencies

38

Consequences of Sequential Consistency

- Corollaries

1. Within the same execution, all processors see the same global order of operations to memory
   → No correctness issue
   → Satisfies the "happened before" intuition

2. Across different executions, different global orders can be observed (each of which is sequentially consistent)
   → Debugging is still difficult (as order changes across runs)

39

Issues with Sequential Consistency?

- Nice abstraction for programming, but two issues:
  - Too conservative ordering requirements
  - Limits the aggressiveness of performance enhancement techniques
- Is the total global order requirement too strong?
  - Do we need a global order across all operations and all processors?
  - How about a global order only across all stores?
    - Total store order memory model; unique store order model
  - How about a enforcing a global order only at the boundaries of synchronization?
    - Relaxed memory models
    - Acquire-release consistency model

40

Issues with Sequential Consistency?

- Performance enhancement techniques that could make SC implementation difficult
- Out-of-order execution
  - Loads happen out-of-order with respect to each other and with respect to independent stores → makes it difficult for all processors to see the same global order of all memory operations
- Caching
  - A memory location is now present in multiple places
  - Prevents the effect of a store to be seen by other processors → makes it difficult for all processors to see the same global order of all memory operations

41

Weaker Memory Consistency

- The ordering of operations is important when the order affects operations on shared data → i.e., when processors need to synchronize to execute a "program region"
- Weak consistency
  - Idea: Programmer specifies regions in which memory operations do not need to be ordered
  - "Memory fence" instructions delineate those regions
    - All memory operations before a fence must complete before fence is executed
    - All memory operations after the fence must wait for the fence to complete
    - Fences complete in program order
  - All synchronization operations act like a fence

42

Tradeoffs: Weaker Consistency

- Advantage
  - No need to guarantee a very strict order of memory operations
    - → Enables the hardware implementation of performance enhancement techniques to be simpler
    - → Can be higher performance than stricter ordering
- Disadvantage
  - More burden on the programmer or software (need to get the "fences" correct)
- Another example of the programmer-microarchitect tradeoff

Summary

- With multicore systems maintaining memory consistency is necessary for correctness
- Snoopy based MSI/MESI
- Coherence is the 4C, can actually cause misses
- Cache coherence does not guarantee correctness, you will still need lock/mutexes to protect critical sections in code.

Covered in the Textbook but not in the course in detail. See a Parallel Computing course for more details.

BACKUP SLIDES: SCALING AND DIRECTORIES [5.4]

Snoopy, power and scaling

- Snoopy requires sending lots of message to every P in the system. But the Ps are far apart, which implies sending lots of data on lots of long wires.
  - Delay on long wires scales poorly. So snoopy is slow.
  - Long wire have large capacitance, which implies lots of dynamic power. So snoopy is power hungry.
- Big busses are thus not a realistic form of interconnect nowadays. Let's look at one alternative.

Snoopy runs out of steam

- The genius of a ring cache (or of most modern distributed interconnect) is that many messages can be in flight at the same time, each using a small part of the interconnect.
- But snoopy requires that all caches know about each others' misses, which requires global broadcasts.
- Snoopy thus prevents modern interconnect from doing what it does best.
- Snoopy doesn't scale beyond small systems.

Directories

- We still have to solve the problem of coherence.
- The problem with snoopy was that the data we want could be held in any L1 cache in the system. So we had to talk to everybody.
- What if we could send messages only to the cache(s) that actually do hold our data?
  - Tremendously fewer messages to be sent.
  - But that implies somebody is keeping track, for every block of data, which caches are storing it and if they've modified it.
  - Will that become another bottleneck?

Directories – the big picture

- On a memory access to a block, we need to:
  - Find out about the state of the block in other caches.
  - Locate additional copies if needed.
  - Communicate with other copies when appropriate.
- Approach must scale well so that it can be used in very large systems.

Basic Directory Approach

- Every address in the system has a **home node,** which manages coherency for that address.
- For time being, assume just one big home node somewhere.
- Each memory block has a directory entry, which tracks copies of block and state information.
- Read and write operations change the state information via network transactions.

Example: Directory Approach

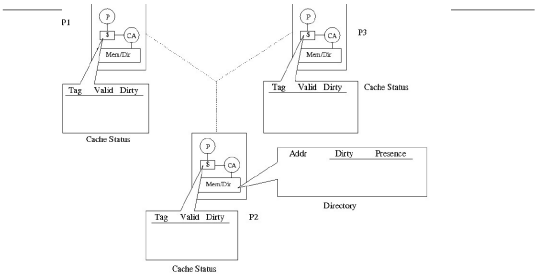
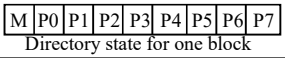


Figure shows directory table for P2 only, but all nodes also have corresponding (but not overlapping tables)

State Bits

- **For Caches**
  - Standard state bits may apply (e.g. Valid/Invalid, Dirty)
- **For Directories**
  - M is a dirty (i.e., modified) bit.
    - Why would we want it? Isn't this info also in the higher-level caches?
      Yes, but having two copies lets the home node make decisions more quickly.
  - *Presence bits* track which P(s) currently owns this block.
  - Only one presence bit can be set for a modified block; many can be set on reads.
  - One such entry for every line cached by any P.



Directory state for one block

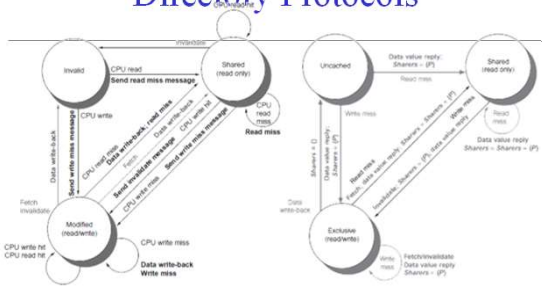
Terminology

- Some definitions that you may see:
  - **home node** where main memory block is controlled
  - **dirty node** where the block has been modified (has current copy)
  - **owner node** where data will be supplied from on a request (either home or dirty node)
  - **exclusive node** the only place where data resides in a cache
  - **local node** where the request for a block originates

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.

Directory Protocols



55