CS5375 Computer Systems Organization and Architecture

Lecture 20

Instructor: Yong Chen, Ph.D.
Department of Computer Science
Texas Tech University
Yong.Chen@ttu.edu, 806-834-0284

Announcements

- HW#4 posted and due on 11/17, Thursday
  - Note there are 8 questions in total (not 10 questions as in the prior homework)

- Conference trip from this afternoon till early next week
  - Class on 11/15, Tue., is skipped, leave the time for you to work on HW#4 or PP#2

- Will reduce one homework to have five homework in total, instead of six
  - Written assignments – 30% (~~six~~ five written assignments, weighing ~~5%~~ 6% each)
  - Last homework, HW#5, will be for covered Chapter 5 and Chapter 6 materials

Outline

- Thread-Level Parallelism

Introduction

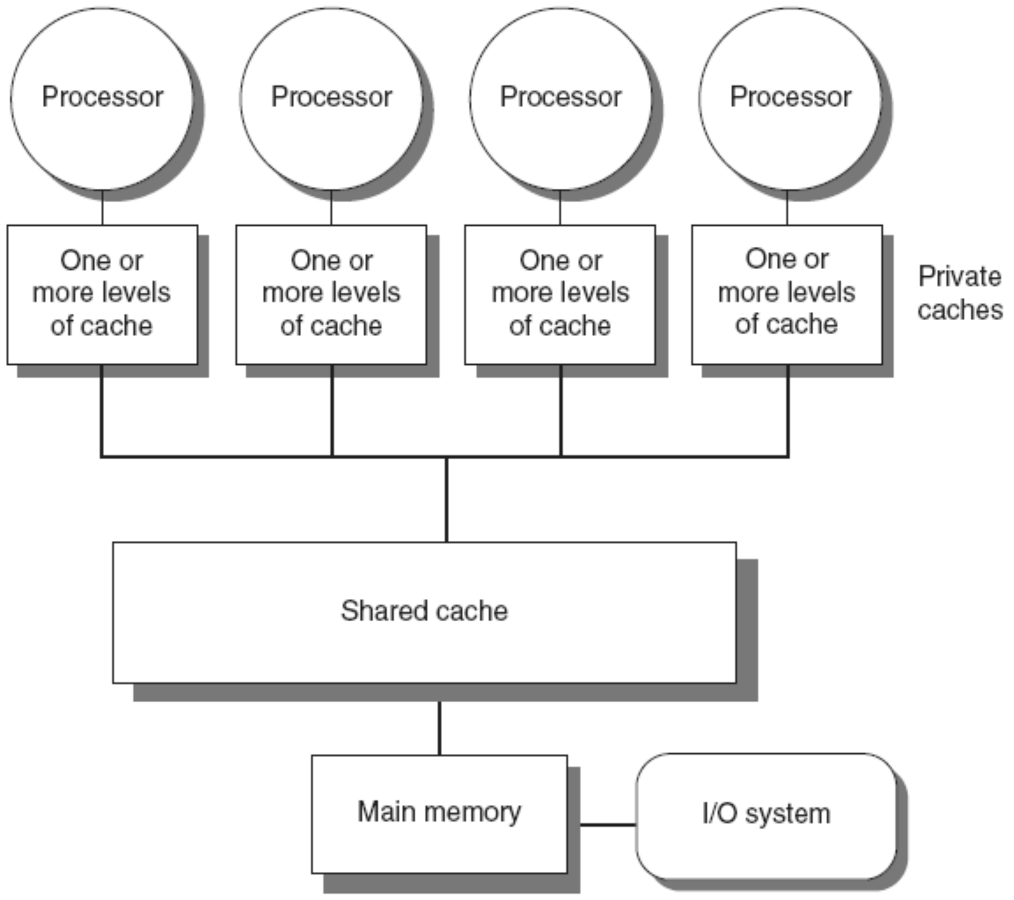
- Thread-Level Parallelism (TLP)
  - Have multiple program counters
  - Uses MIMD model
    - Multiple Instruction: every processor may execute different instruction stream, may have separate clocks
    - Multiple Data: every processor may be working with a different data stream
  - Targeted for tightly-coupled shared-memory multiprocessors
    - V.S. distributed-memory multicomputers (“clusters”, “warehouse-scale computers”)

- Amount of computation assigned to each thread = grain size
  - Threads can be used for data-level parallelism, but the overheads may outweigh the benefit

Types

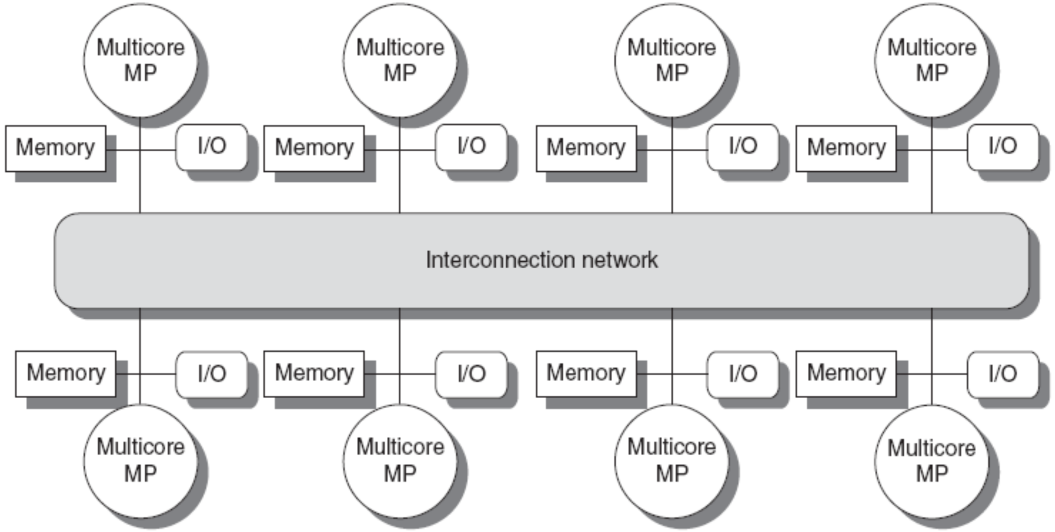
- Symmetric multiprocessors (SMP)
  - Small number of cores
  - Share single memory with uniform memory latency

- Also called as centralized shared-memory multiprocessors



Types (cont.)

- Distributed shared memory (DSM)
  - Memory distributed among processors
  - Non-uniform memory access/latency (NUMA)
  - Processors connected via direct (switched) and non-direct (multi-hop) interconnection networks



  - Note this is different from "distributed-memory multicomputers", where memory is not shared, and some form of "message passing" to solve a problem

Cache Coherence

- Processors may see different values through their caches (private caches):

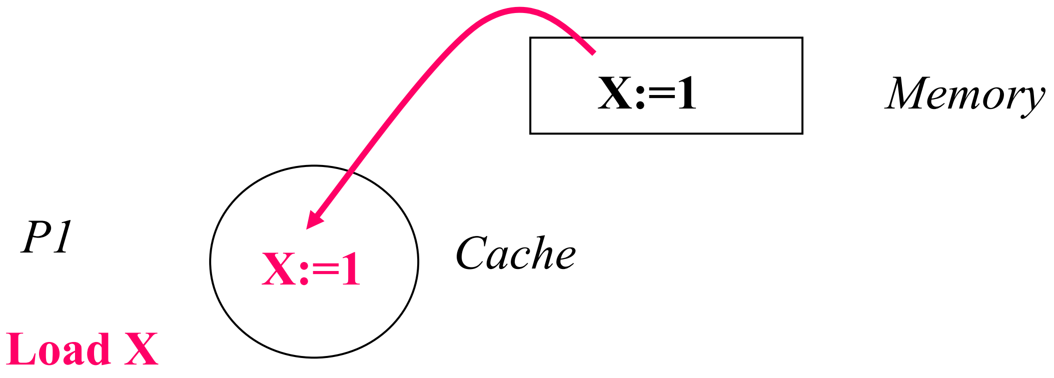
Time	Event	Cache contents for processor A	Cache contents for processor B	Memory contents for location X
0				1
1	Processor A reads X	1		1
2	Processor B reads X	1	1	1
3	Processor A stores 0 into X	0	1	0

Assume a write-through cache

- Such a problem is referred to as the cache coherence problem

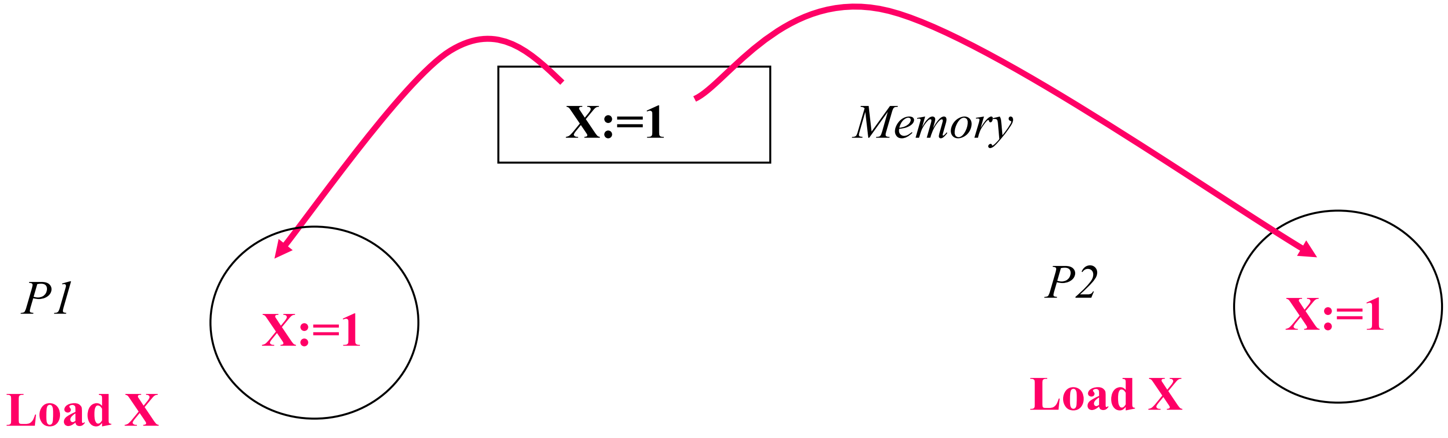
Cache Coherence

- P1 loads X from main memory into its cache



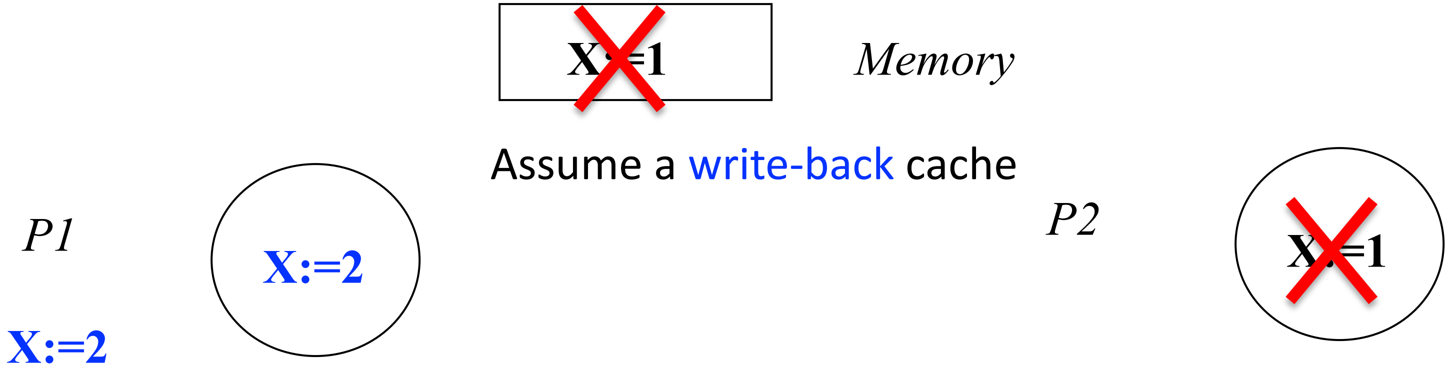
Cache Coherence

- P1 loads X from main memory into its cache
- P2 loads X from main memory into its cache



Cache Coherence

- P1 stores 2 into X
- We don't have consistent values for X across the memory hierarchy



Cache Coherence

- Cache coherence problem has two aspects

Single variable updation

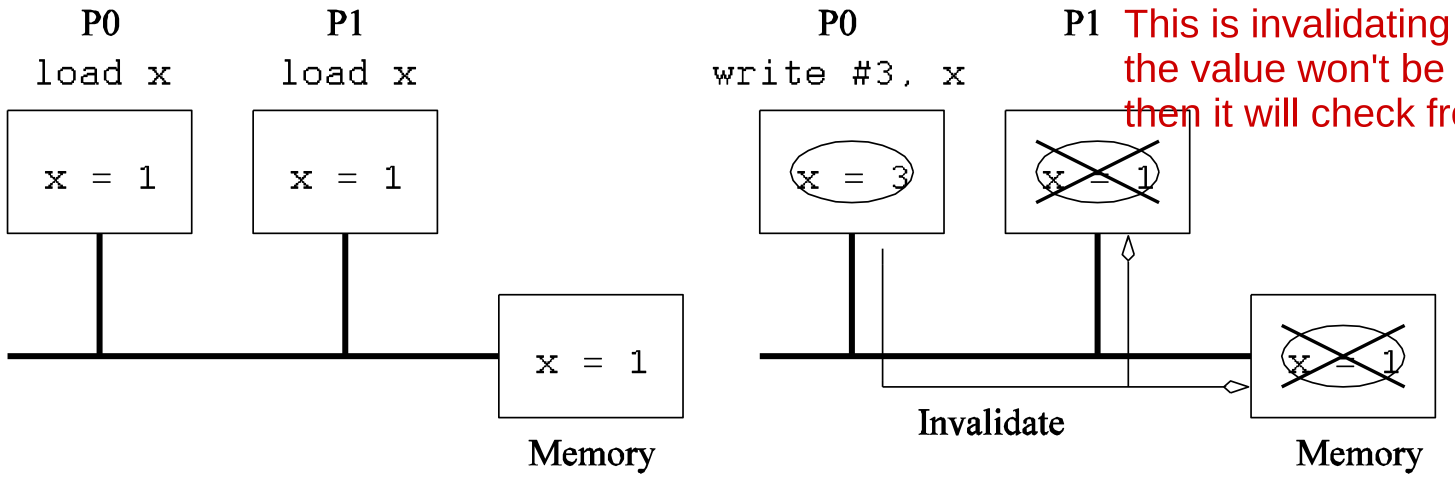
- Coherence aspect: defines *what* a written value returned by a read
  - All reads by any processor must return the most recently written value
  - Writes to the same location by any two processors are seen in the same order by all processors

This is for Multiple variables updation

- Consistency aspect: defines *when* a written value returned by a read
  - If a processor writes location A followed by location B, any processor that sees the new value of B must also see the new value of A

Enforcing Coherence

When the value of a variable is changed, all its copies must either be invalidated or updated.



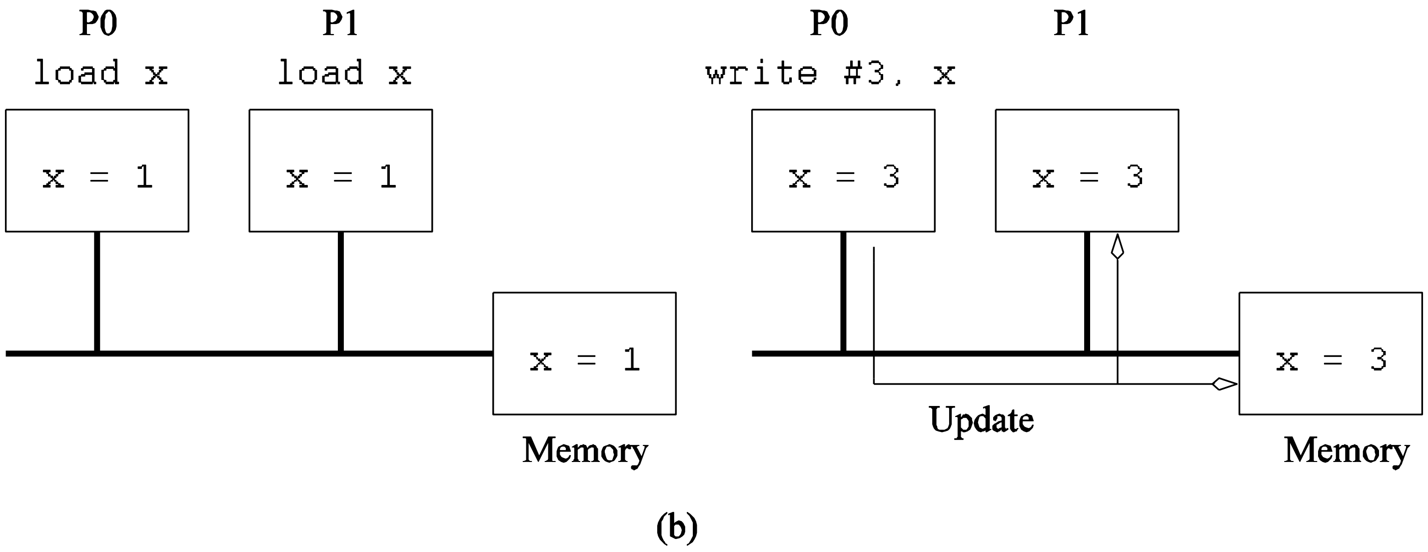
This is invalidating the value which means the value won't be present here then it will check from the original one

(a)

Cache coherence in multiprocessor systems: (a) Invalidate protocol

Enforcing Coherence (cont.)

When the value of a variable is changed, all its copies must either be invalidated or updated.



Cache coherence in multiprocessor systems: (b) Update protocol for shared variables.

Update v.s. Invalidate Protocols

- If a processor just reads a value once and does not need it again, an update protocol may generate significant overhead

- If two processors make interleaved test and updates to a variable, an update protocol is better
  - Ping-pong

it is a problem caused by cahce

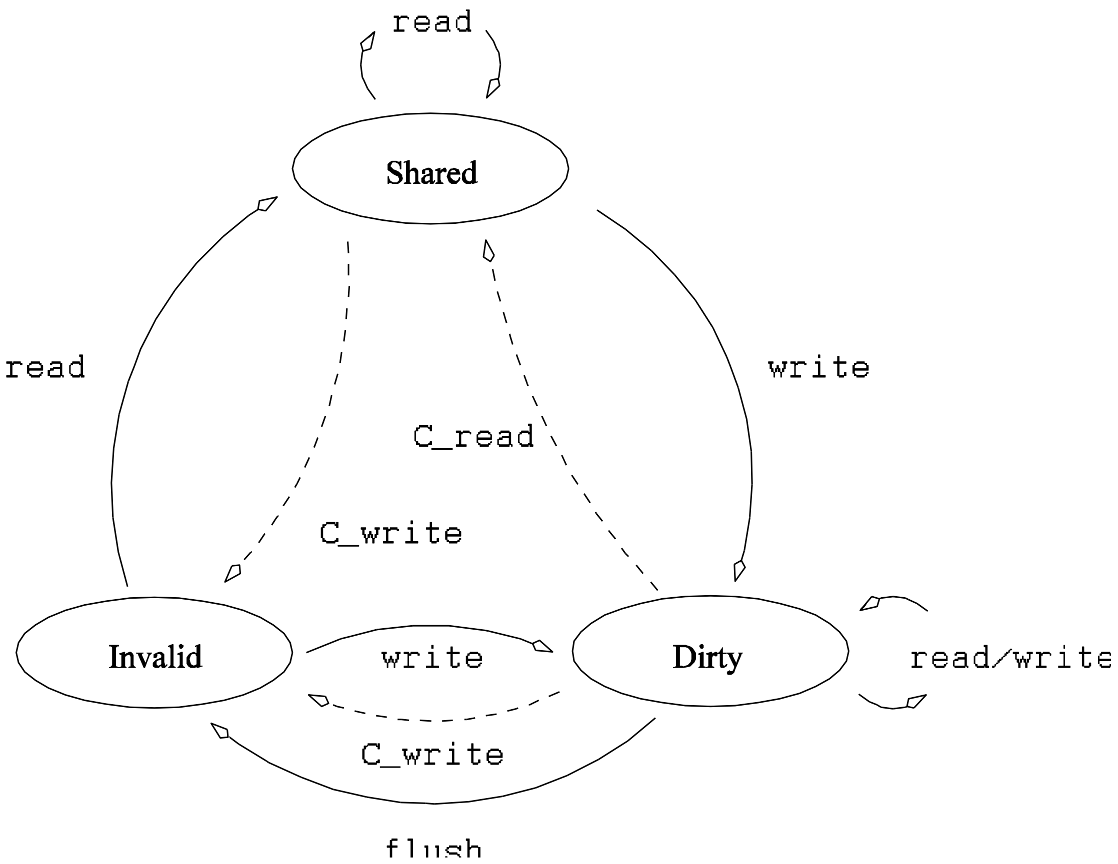
- Both protocols suffer from false sharing overheads (two words that are not shared, however, they lie on the same cache line)

- Most current machines use invalidate protocols

Using Invalidate Protocols

- Each copy of a data item is associated with a state
- One example of such a set of states is, shared, invalid, or dirty/exclusive
- In shared state, there are multiple valid copies of the data item (and therefore, an invalidate would have to be generated on a write (or store))
- In dirty state, only one copy exists and therefore, no invalidates need to be generated
- In invalid state, the data copy is invalid, therefore, a read (or load) generates a data request (and associated state changes)

Using Invalidate Protocols (cont.)



State diagram of a simple three-state coherence protocol.

Using Invalidate Protocols (cont.)

Time	Instruction at Processor 0	Instruction at Processor 1	Variables and their states at Processor 0	Variables and their states at Processor 1	Variables and their states in Global mem.
					x = 5, D y = 12, D
	read x		x = 5, S		x = 5, S
		read y		y = 12, S	y = 12, S
	x = x + 1		x = 6, D		x = 5, I
		y = y + 1		y = 13, D	y = 12, I
	read y		y = 13, S	y = 13, S	y = 13, S
		read x	x = 6, S	x = 6, S	x = 6, S
	x = x + y		x = 19, D	x = 6, I	x = 6, I
		y = x + y	y = 13, I	y = 19, D	y = 13, I
	x = x + 1		x = 20, D		x = 6, I
		y = y + 1		y = 20, D	y = 13, I

D for dirty

S for shared

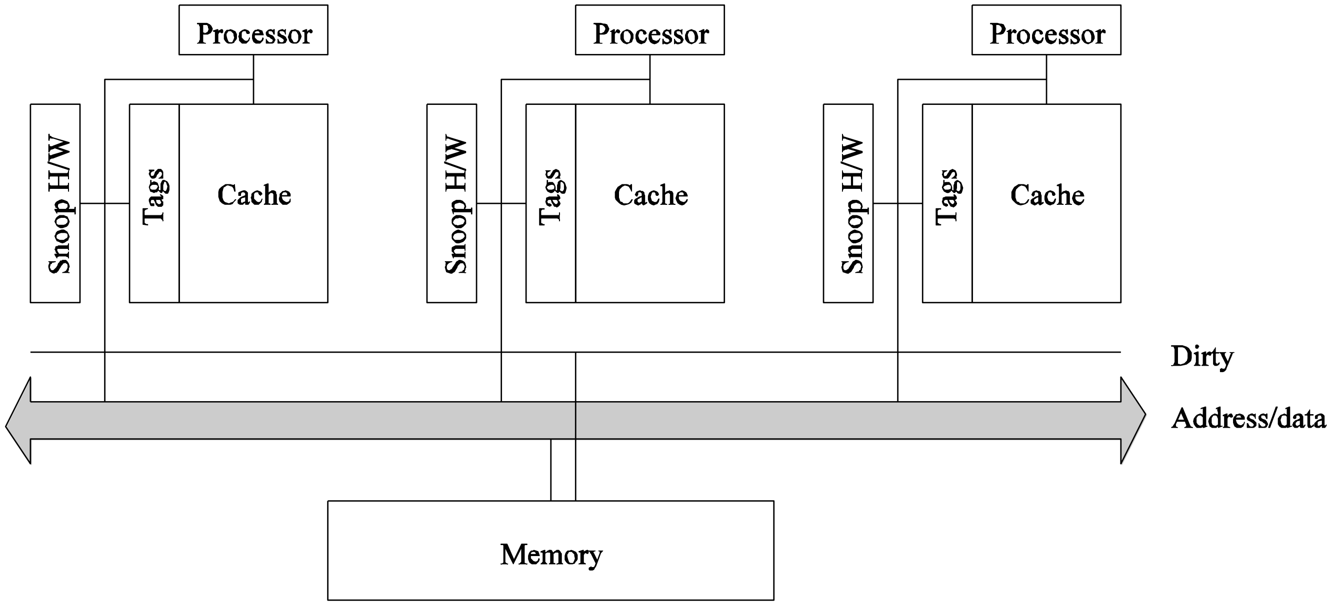
I for invalid

Example of a thread program execution with the simple three-state coherence protocol

Snoopy/Snooping Cache Systems

How are invalidates sent to the right processors?

In snoopy caches, there is a broadcast media that listens to all invalidates and read requests and performs appropriate coherence operations locally



A simple snooping bus based cache coherence system.

Performance of Snoopy Caches

- Once copies of data are tagged dirty, all subsequent operations can be performed locally on the cache without generating external traffic.

- If a data item is read by a number of processors, it transitions to the shared state in the cache and all subsequent read operations become local

- If processors read and update data at the same time, they generate coherence requests on the bus - which is ultimately bandwidth limited

Performance of Snoopy Caches

- Coherence influences cache miss rate
  - Coherence misses
  - True sharing misses
    - Write to shared block (transmission of invalidation)
    - Read an invalidated block
  - False sharing misses
    - Read an unmodified word in an invalidated block

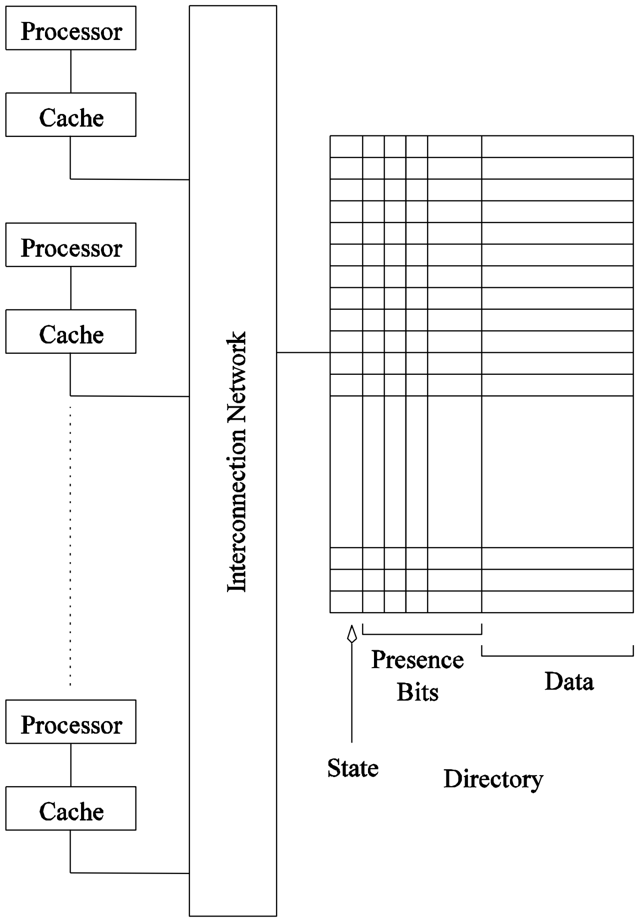
Directory Based Cache Coherence Systems

- In snoopy caches, each coherence operation is sent to all processors
  - This is an inherent limitation

- Why not send coherence requests to only those processors that need to be notified?

- This is done using a directory, which maintains a presence vector for each data item (cache line) along with its state.

Directory Based Cache Coherence Systems



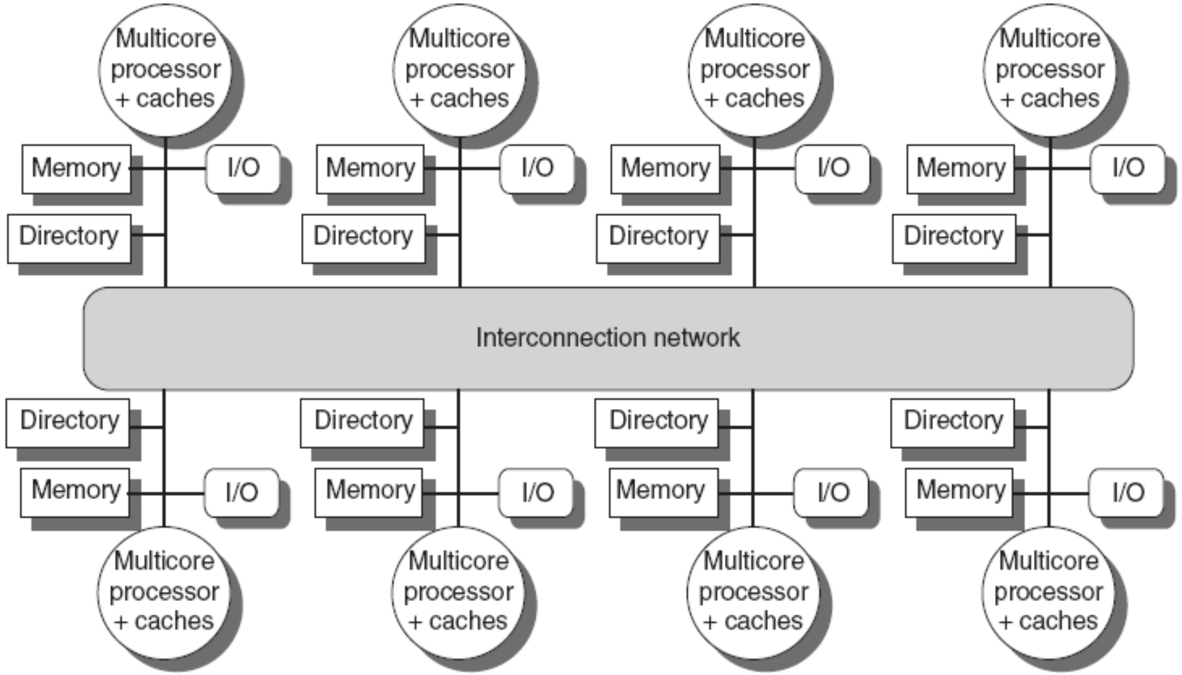
Architecture of typical directory based systems: a centralized directory

Performance of Directory Based Cache Coherence

- The need for a snoopy hardware is replaced by the directory
- The additional bits to store the directory may add significant overhead
  - How much overhead?
- The interconnection network must be able to carry all the coherence requests
- The directory is a point of contention, therefore, distributed directory schemes can be used

Directory Based Cache Coherence Systems

- Permit $O(p)$ simultaneous coherence operations
- More scalable than snoopy or centralized directory systems
- Remains significant overhead of directory storage



Architecture of typical directory based systems: a distributed directory

Readings

- Chapter 5, 5.1-5.4