CMPE 110: Computer Architecture

Week 10
Multicore

Jishen Zhao (http://users.soe.ucsc.edu/~jzhao/)

[Adapted in part from Onur Mutlu, Jose Renau, Mary Jane Irwin, Joe Devietti, and others]

Reminder

- Quiz 4 will be posted today
  - Due on Nov. 23 (Wed) 11:59pm

Review: Virtual memory

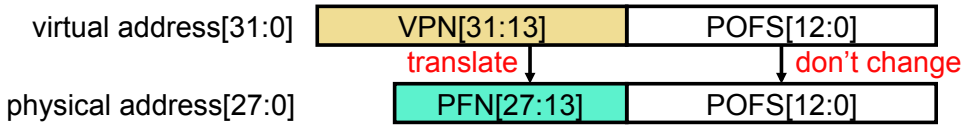
- OS virtualizes memory and I/O devices

- Virtual memory
  - "infinite" memory, isolation, protection, inter-process communication
  - Virtual-physical address translation
  - Page tables
  - TLBs
    - It is a cache, not a buffer
  - Page faults
  - DMA
  - Virtual memory and cache

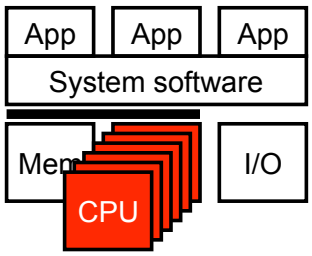
Review: Virtual memory

- Why do need virtual memory?
- What is page table? Where is it?
- What is TLB? Is it a buffer or a cache?
- What is page fault? How is page fault handled? What is DMA?
- What is MMU? What does it do?
- What is virtual cache? What is physical cache? What is virtual-physical cache?
- Virtual-physical address translation. Example question on the next slide…

Review: Address Translation

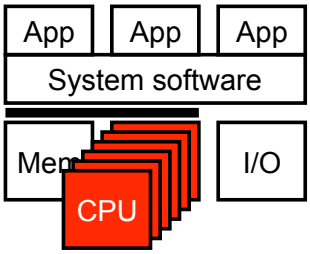


- Example above
  - What is page size in KB?
  - How many virtual pages can each process have?
    - VPN 19 bits →2^{19} virtual pages
  - how many PTEs in total in maximum for 2 processes?
    - 2^{19} PTEs in the page table for each process → 2 * 2^{19} PTEs in total for two processes



Multicore & Multiprocessor Hardware

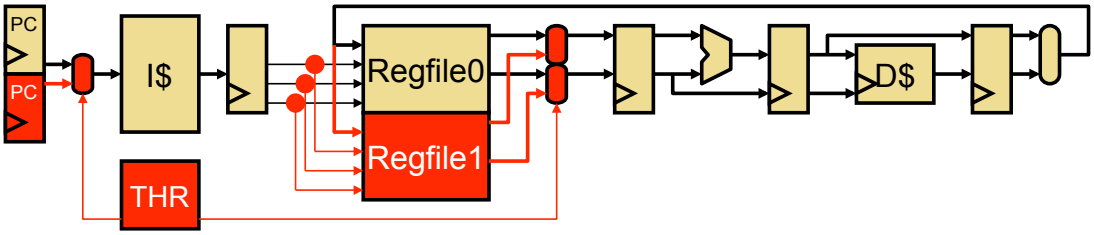
Roadmap Checkpoint



- Multicore vs. Hardware multihreading
- Cache coherence
- Memory consistency models

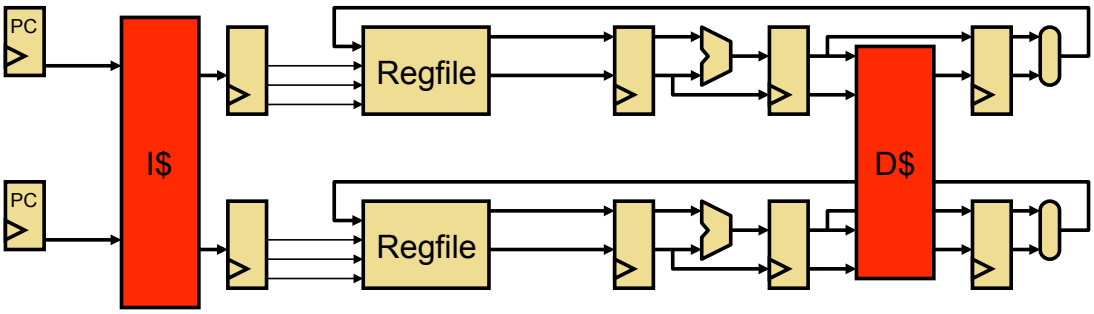
Multicore and Hardware Multithreading Implementation

Recall: Hardware Multithreading



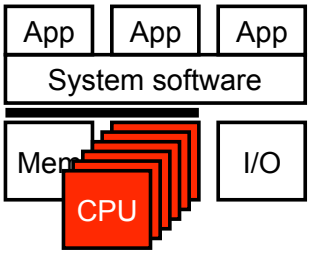
- **Hardware Multithreading (MT)**
  - Multiple threads dynamically share a single pipeline
  - Replicate only per-thread structures: program counter & registers
  - Hardware interleaves instructions
  - \+ **Multithreading improves utilization and throughput**
    - Single programs utilize <50% of pipeline (branch, cache miss)
  - **Multithreading does not improve single-thread performance**
    - Individual threads run as fast or even slower
  - **Coarse-grain MT**: switch on cache misses Why?
  - **Simultaneous MT**: no explicit switching, fine-grain interleaving

Simplest Multiprocessor



- Replicate entire processor pipeline!
  - Instead of replicating just register file & PC
  - Exception: share the caches (we'll address this bottleneck soon)
- Multiple threads execute
  - Shared memory programming model
  - Operations (loads and stores) are interleaved "at random"
  - Loads returns the value written by most recent store to location

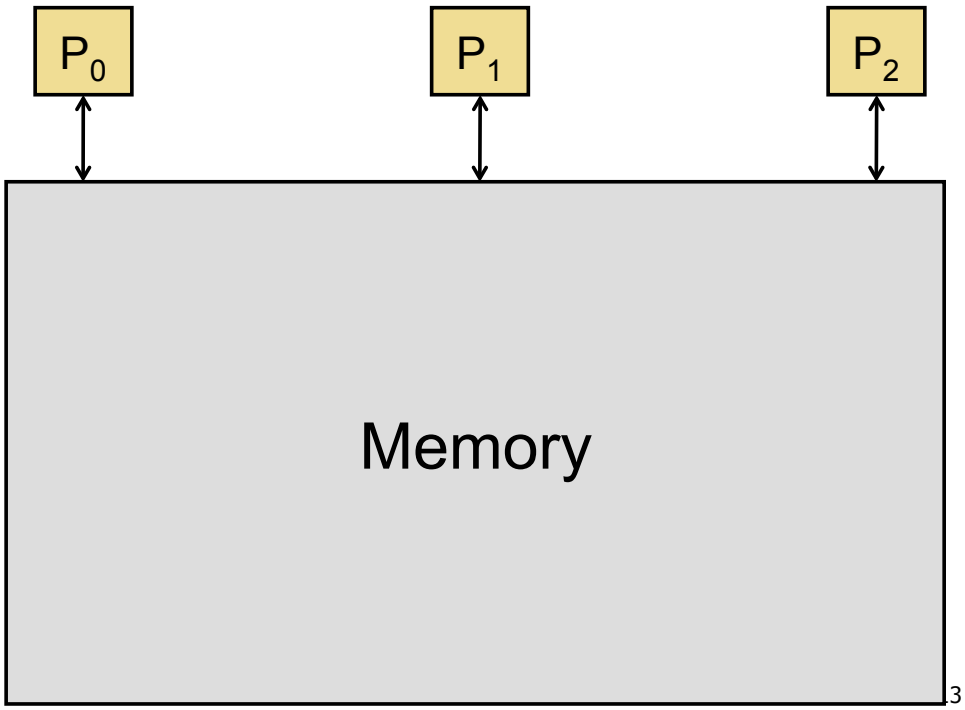
Roadmap Checkpoint



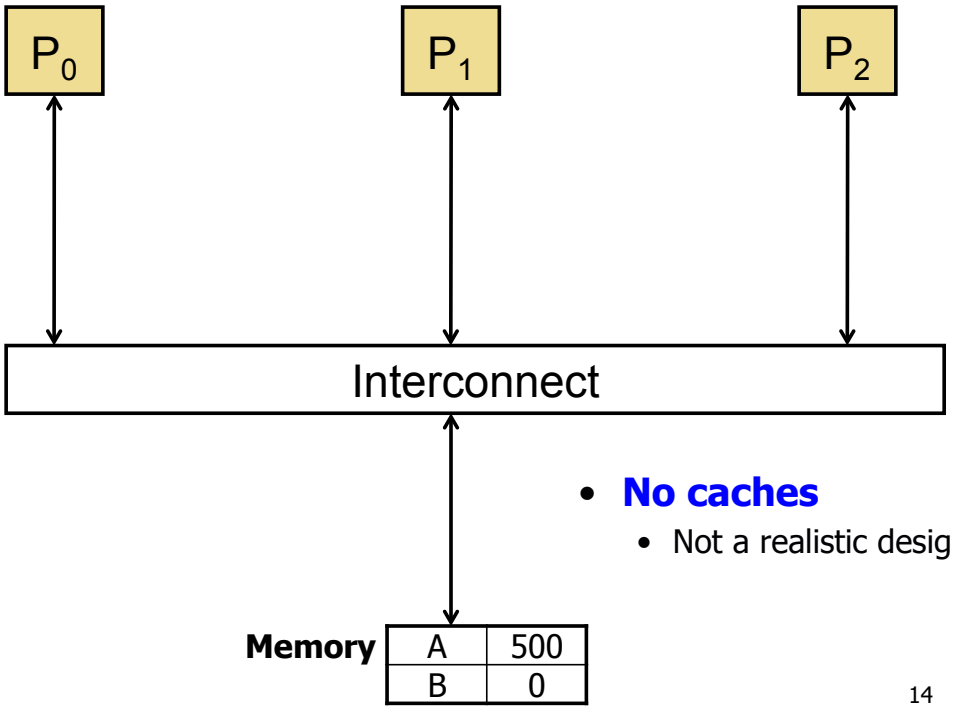
- Multicore vs. Hardware multihreading
- Cache coherence
- Memory consistency models

Cache coherence

No-Cache (Conceptual) Implementation

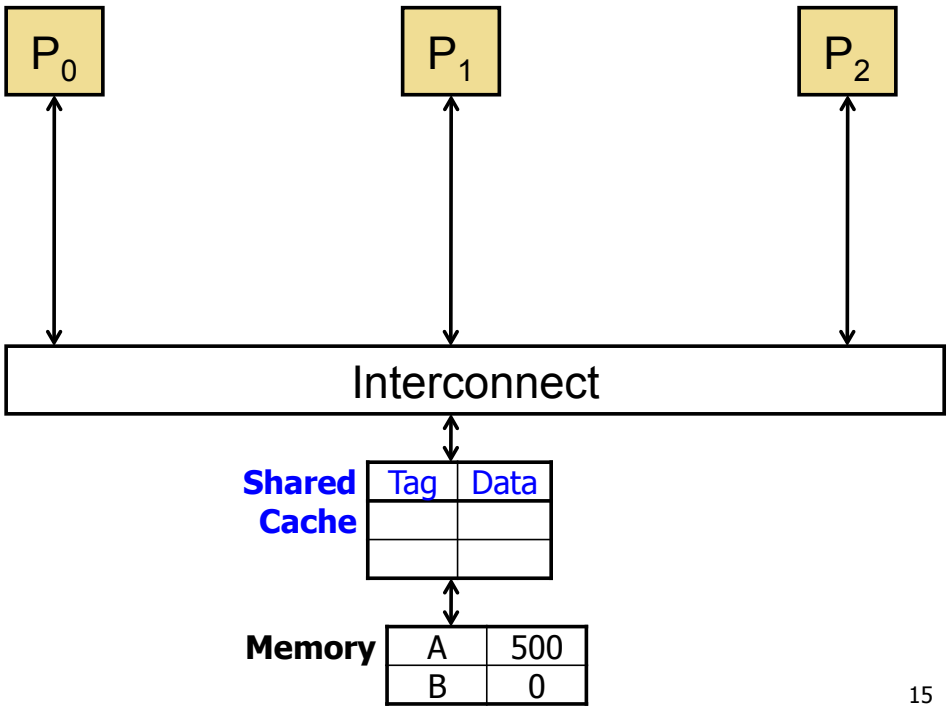


No-Cache (Conceptual) Implementation

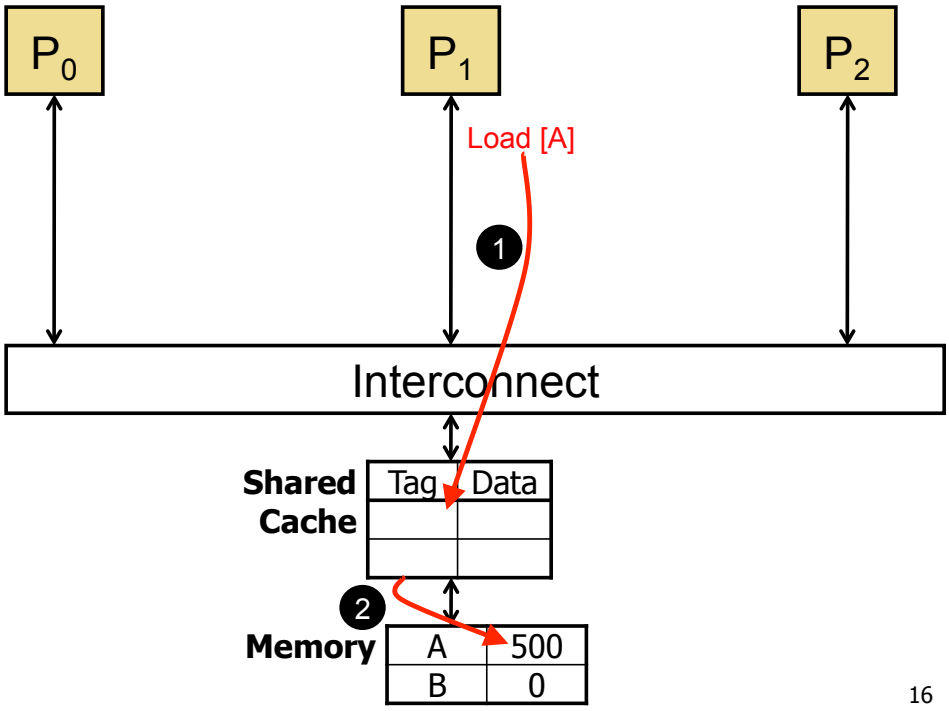


- **No caches**
  - Not a realistic design

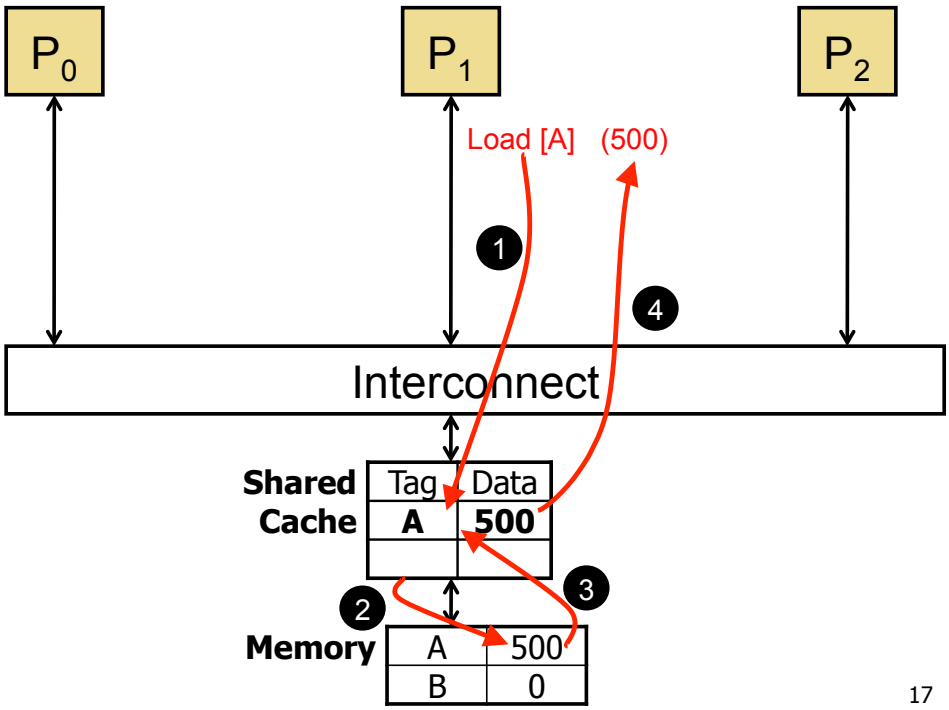
Shared Cache Implementation



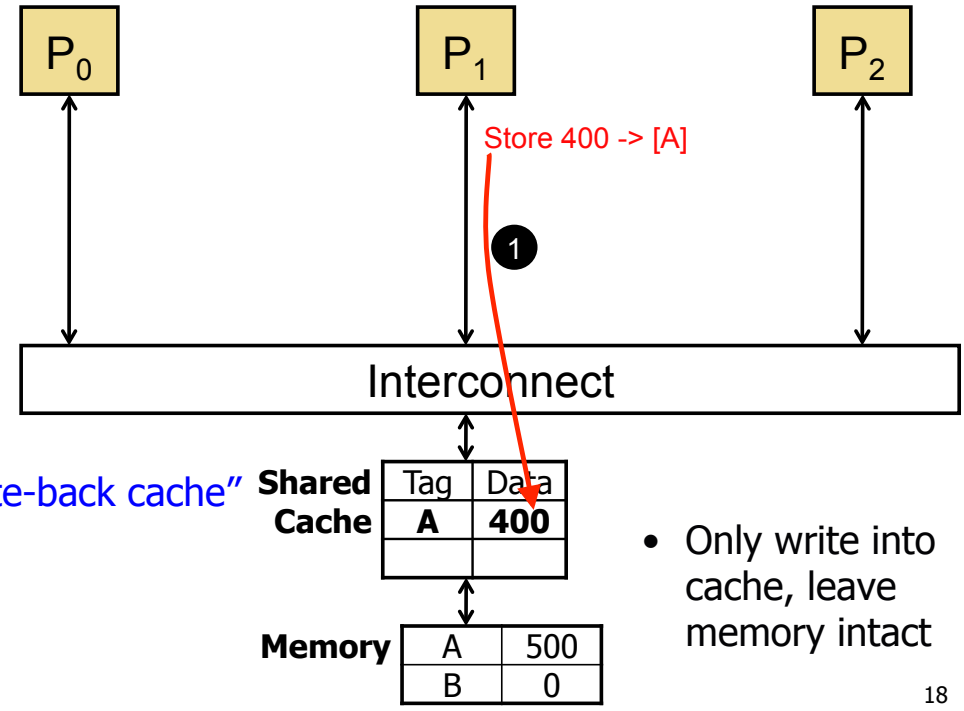
Shared Cache Implementation



Shared Cache Implementation

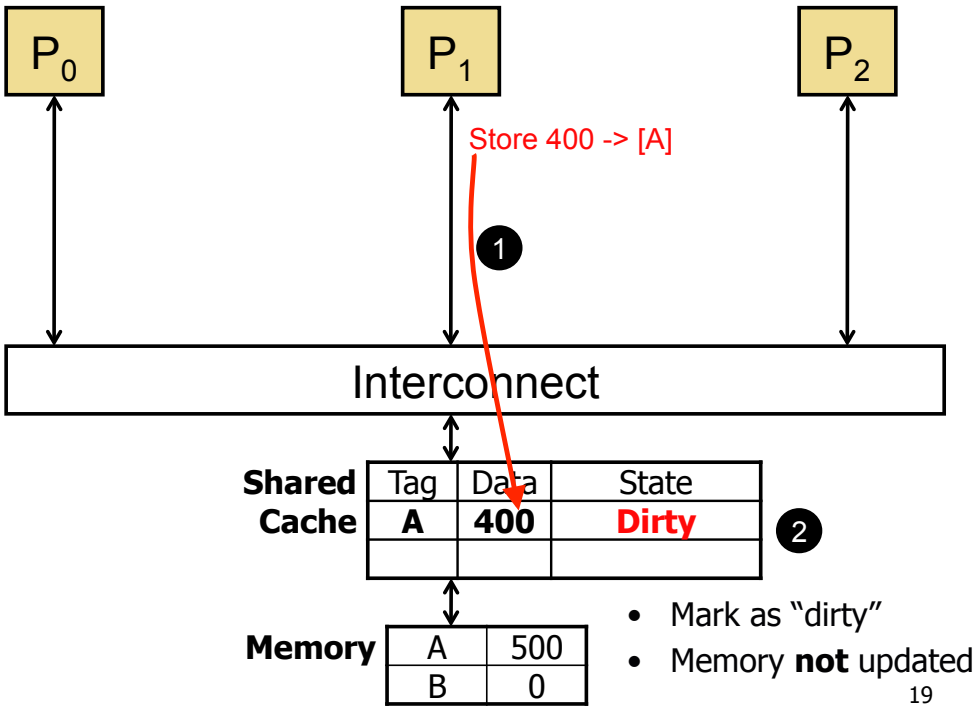


Shared Cache Implementation



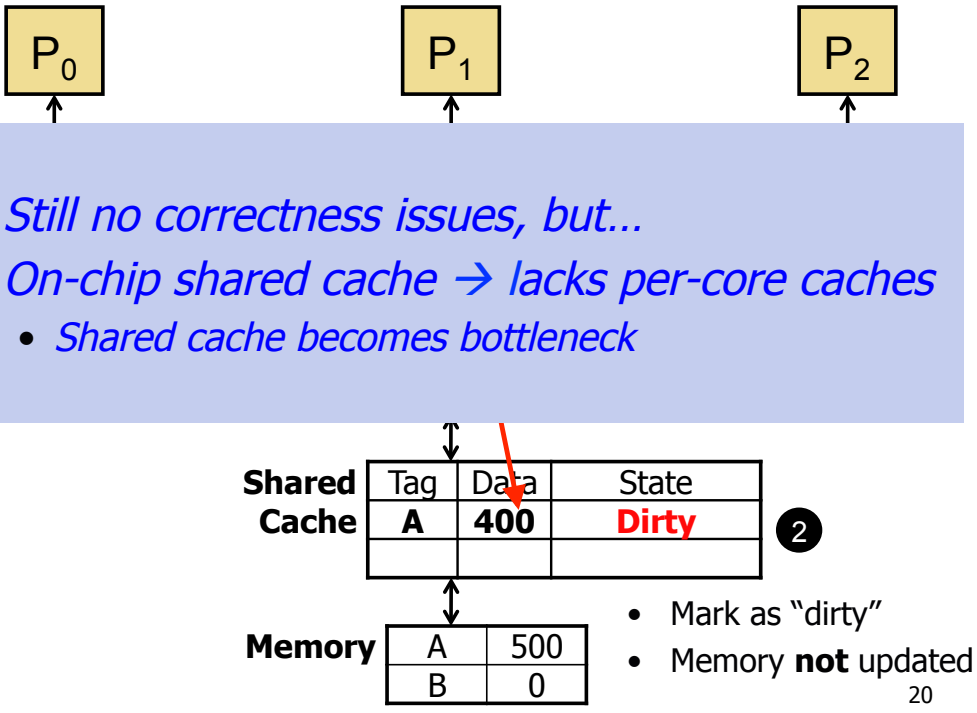
- Only write into cache, leave memory intact

Shared Cache Implementation



- Mark as "dirty"
- Memory **not** updated

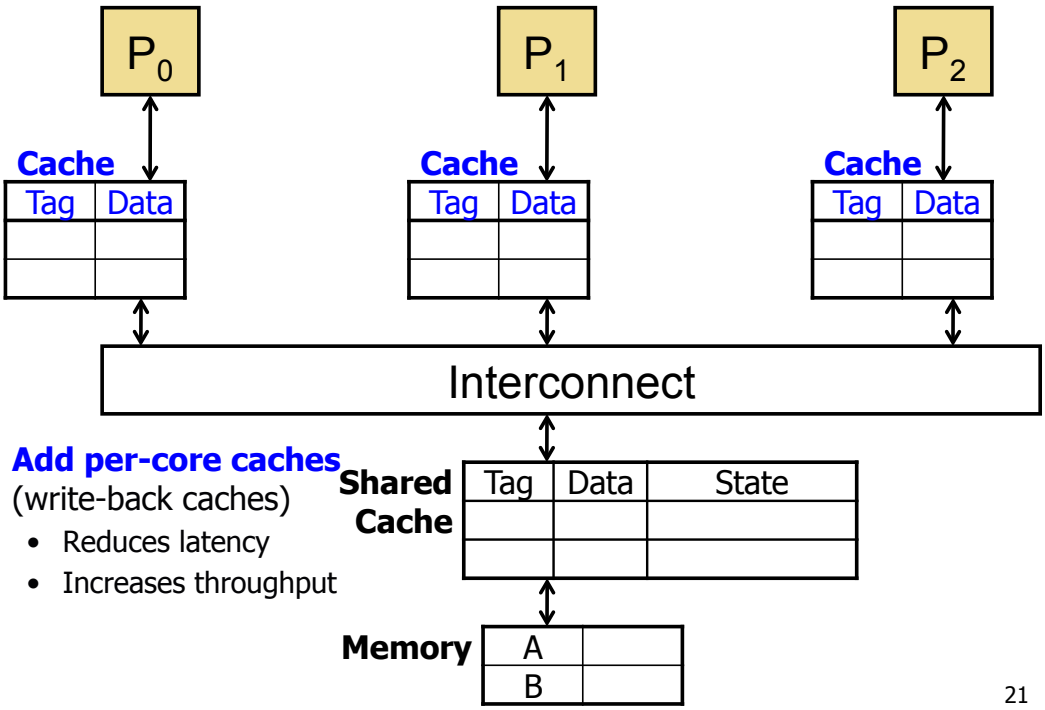
Shared Cache Implementation



- *Still no correctness issues, but…*
- *On-chip shared cache → lacks per-core caches*
  - *Shared cache becomes bottleneck*

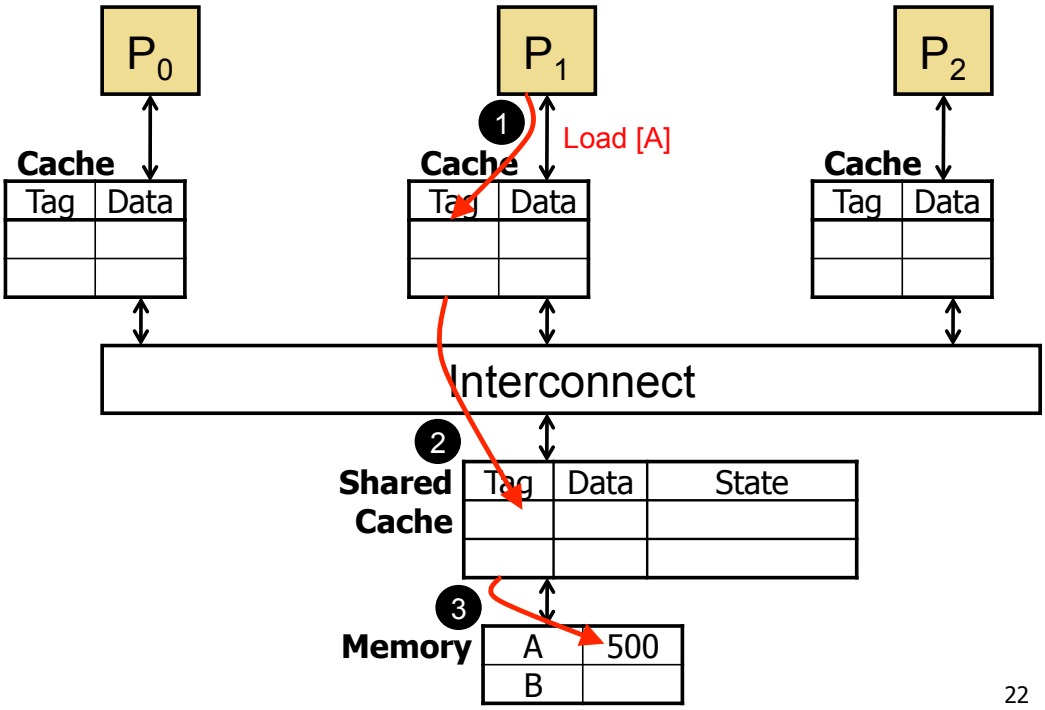
- Mark as "dirty"
- Memory **not** updated

Adding Private Caches

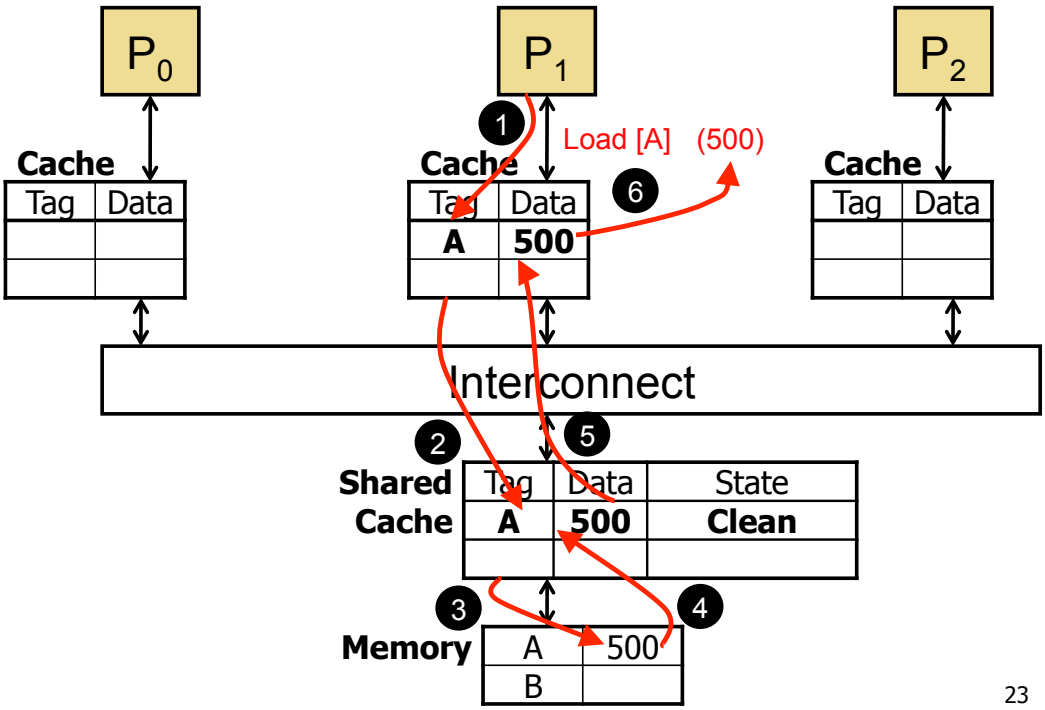


- **Add per-core caches** (write-back caches)
  - Reduces latency
  - Increases throughput

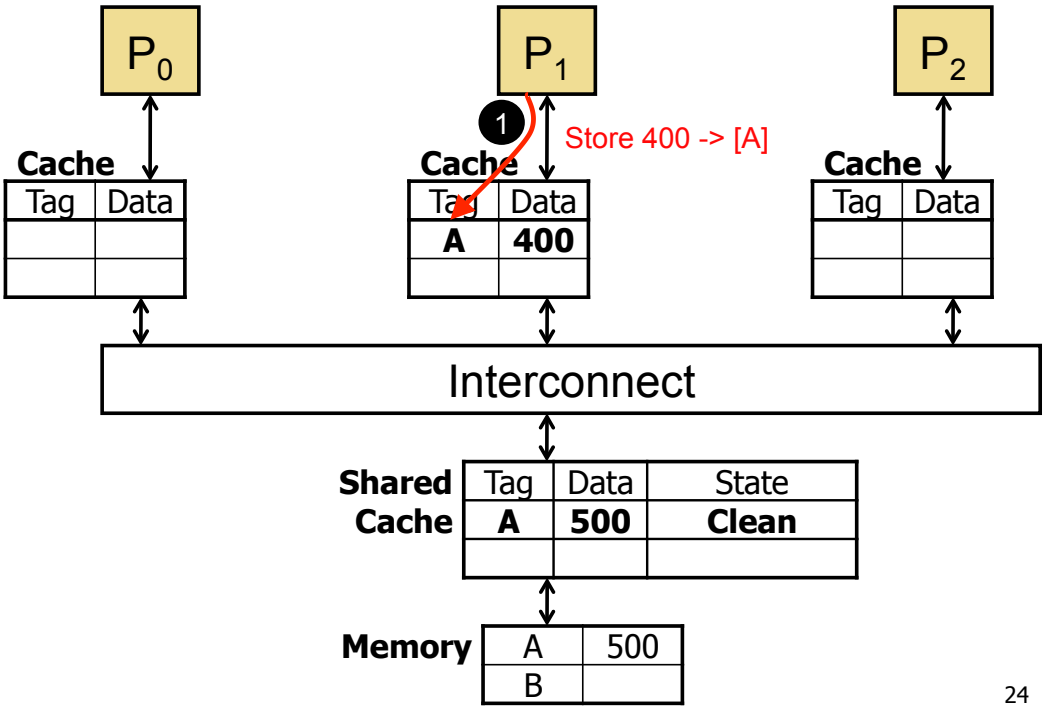
Adding Private Caches



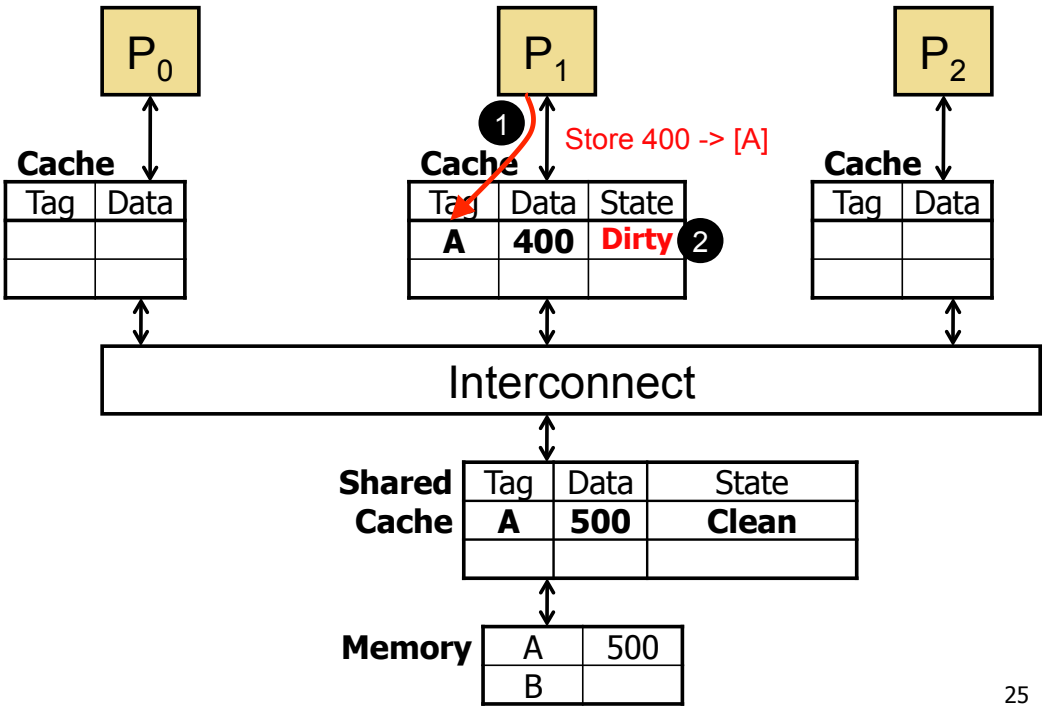
Adding Private Caches



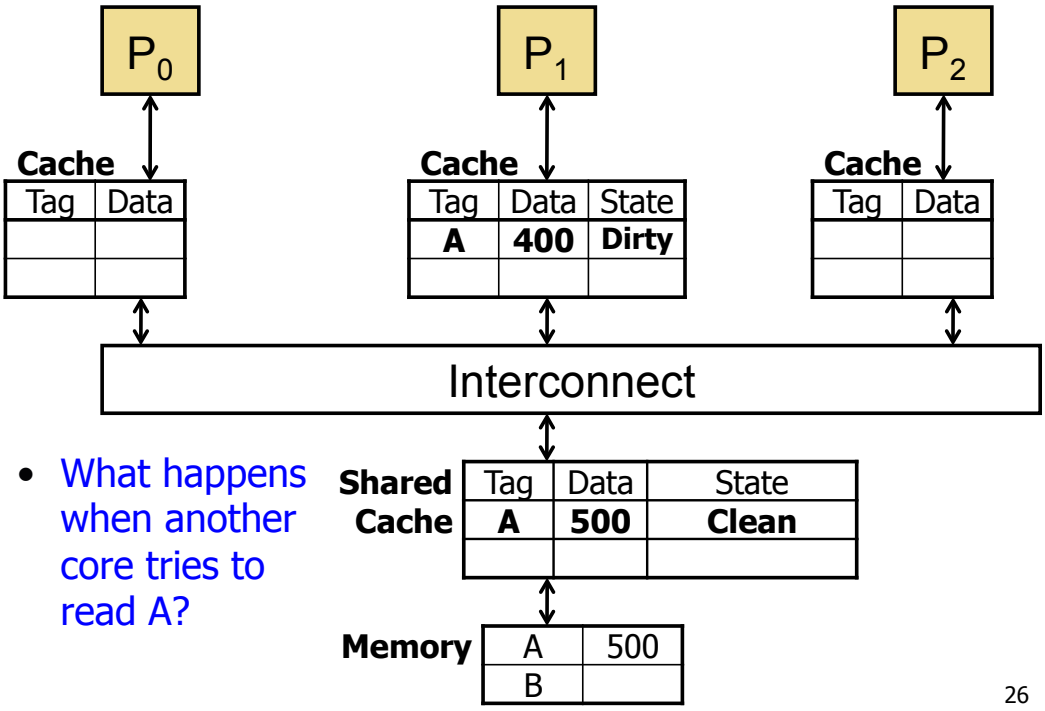
Adding Private Caches



Adding Private Caches

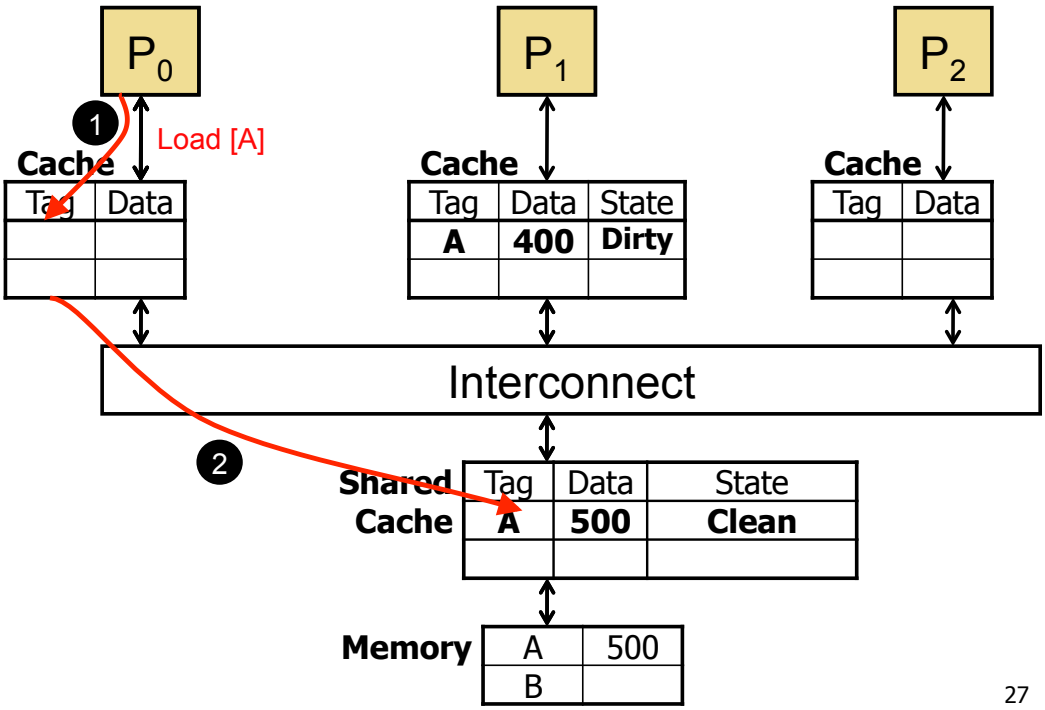


Private Cache Problem: Incoherence

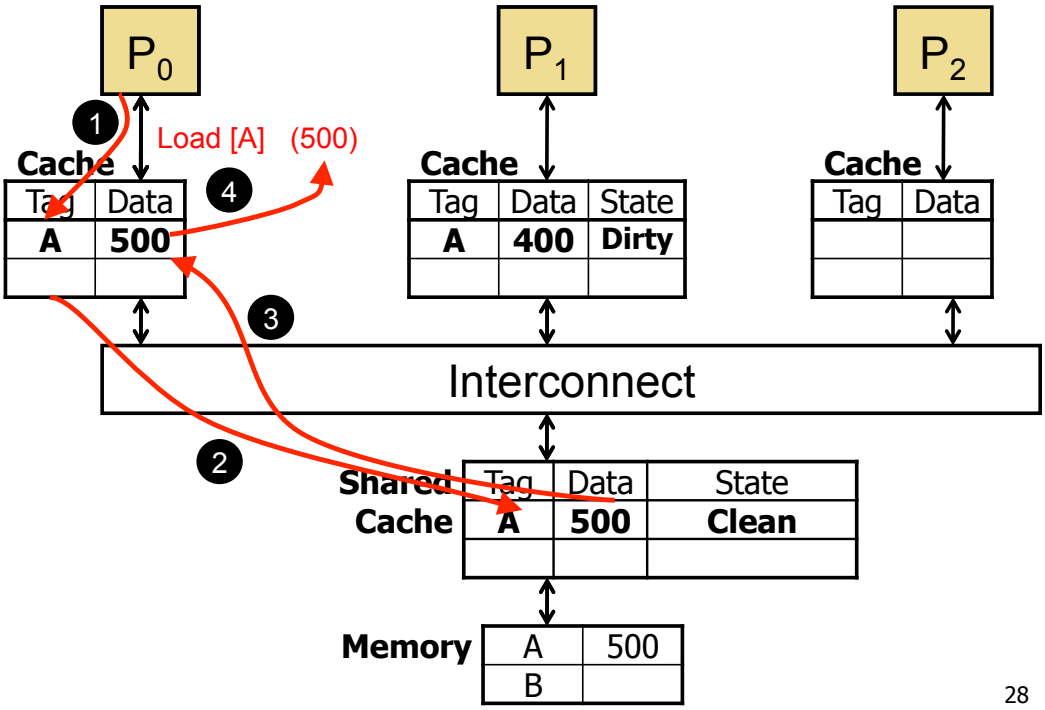


- What happens when another core tries to read A?

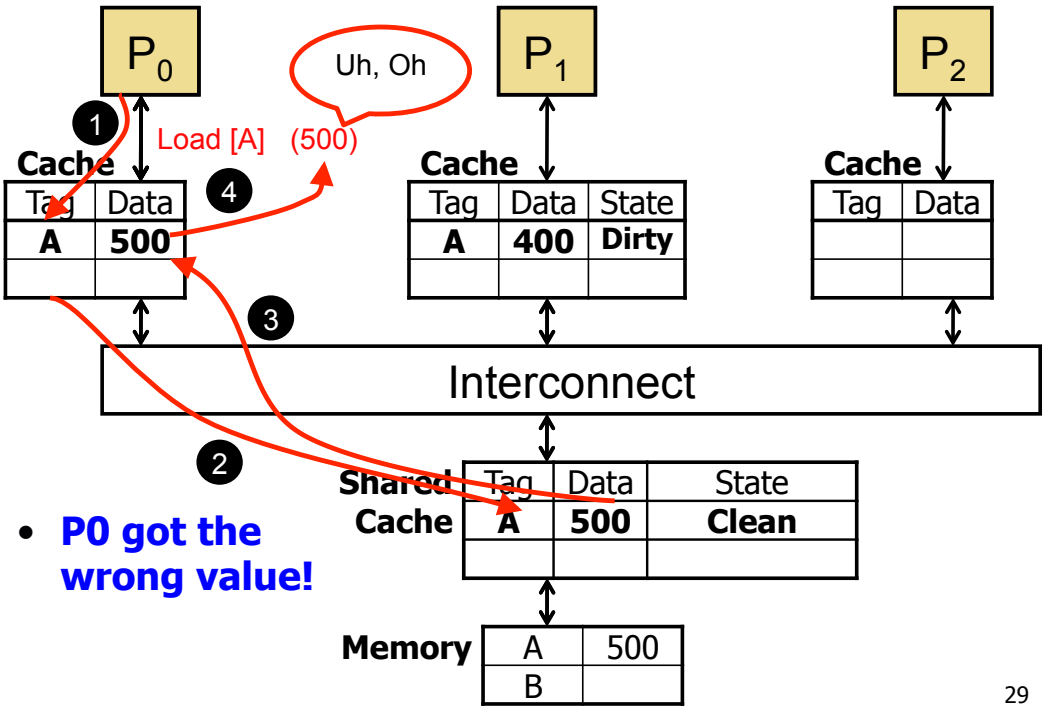
Private Cache Problem: Incoherence



Private Cache Problem: Incoherence



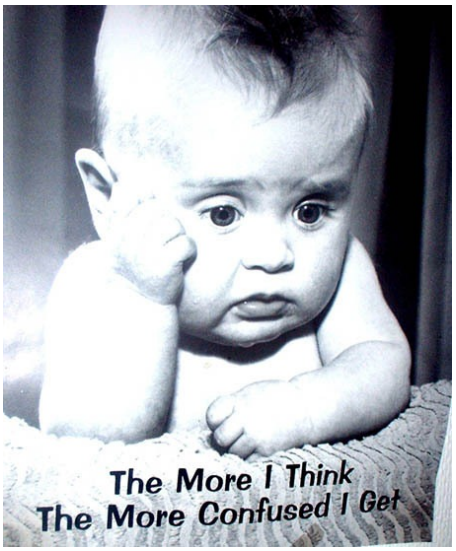
Private Cache Problem: Incoherence

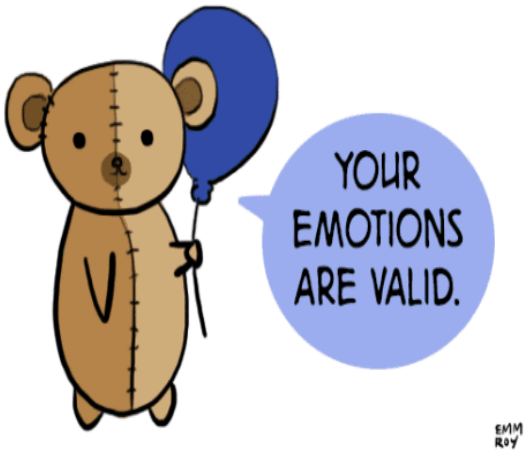


- **P0 got the wrong value!**

Cache Coherence: Who bears the brunt?

- Software
  - Caches are invisible to the programmer

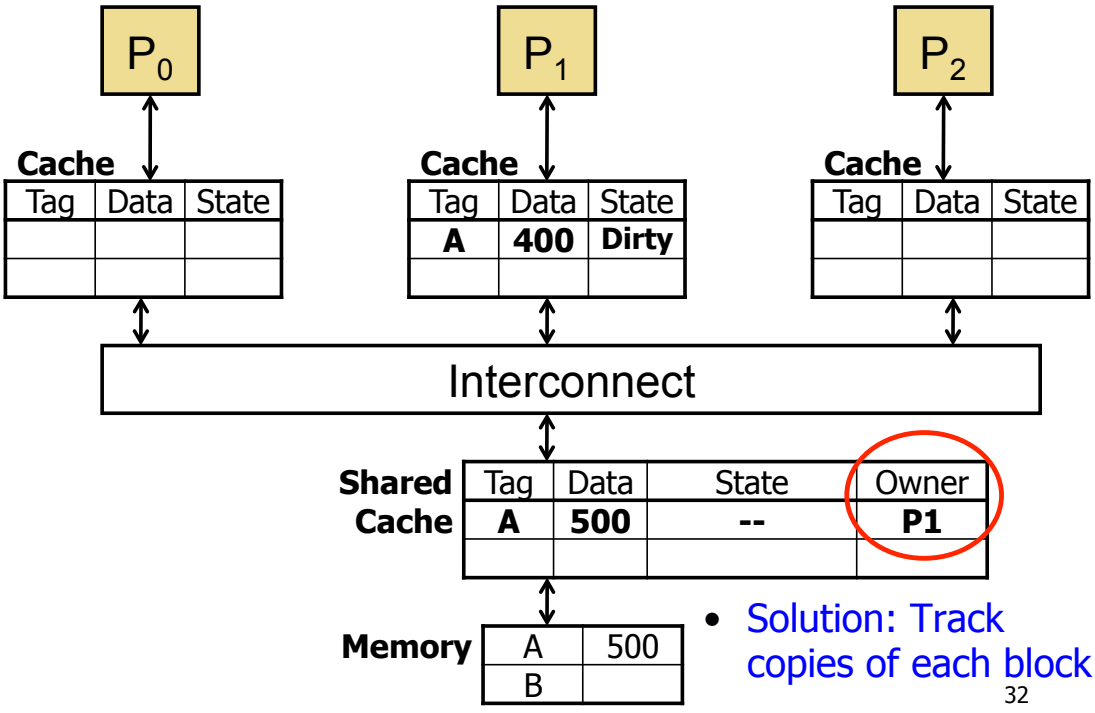




Cache Coherence: Who bears the brunt?

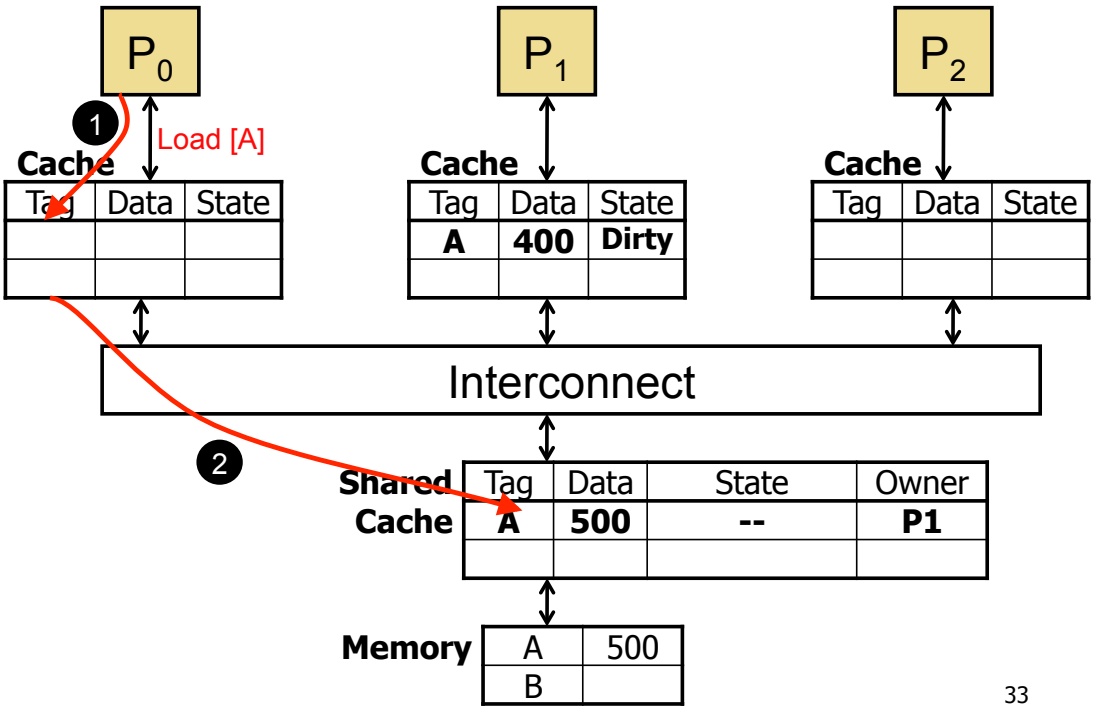
Let's do it in hardware

Rewind: Fix Problem by Tracking Sharers

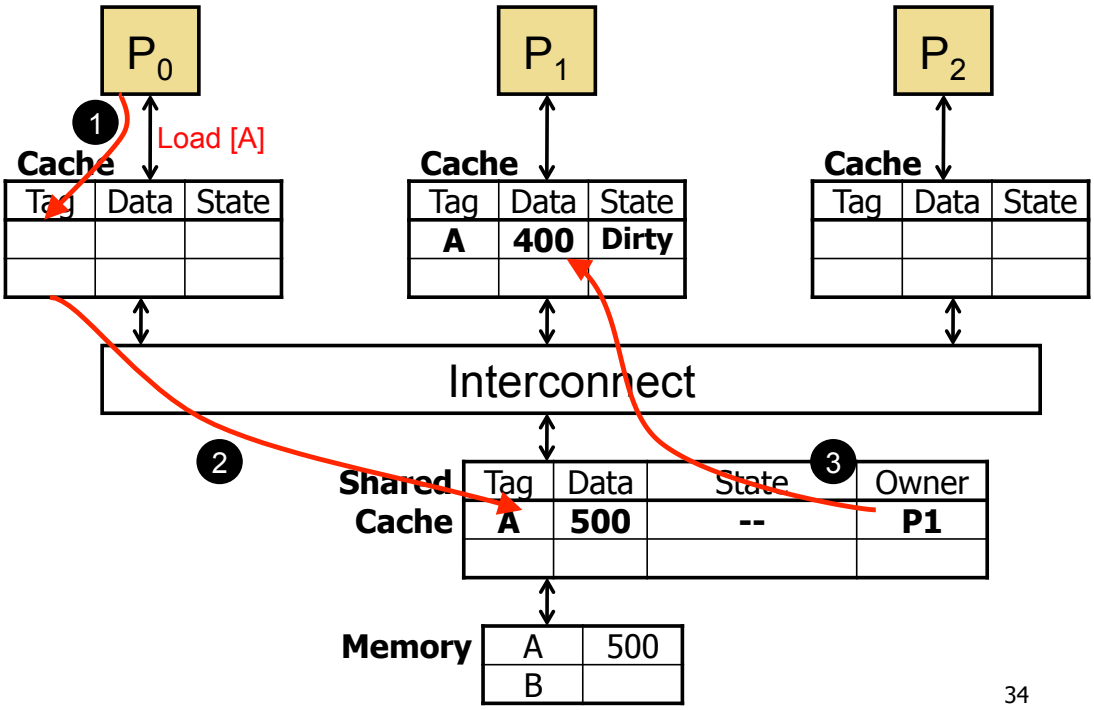


- Solution: Track copies of each block

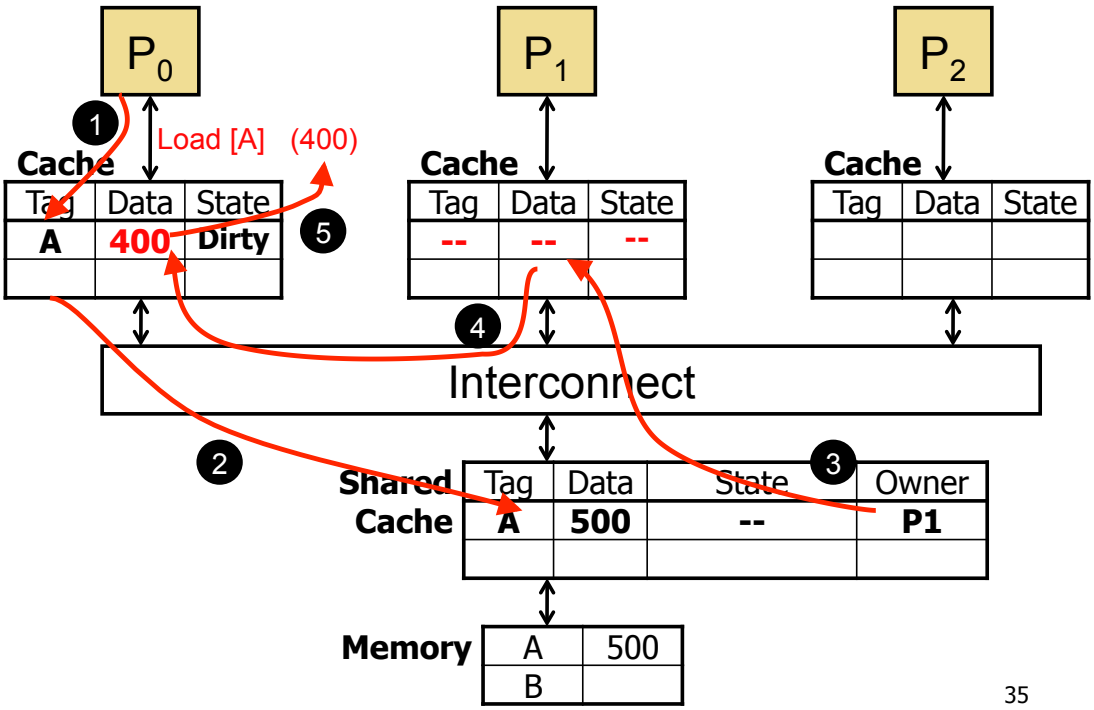
Use Tracking Information to "Invalidate"



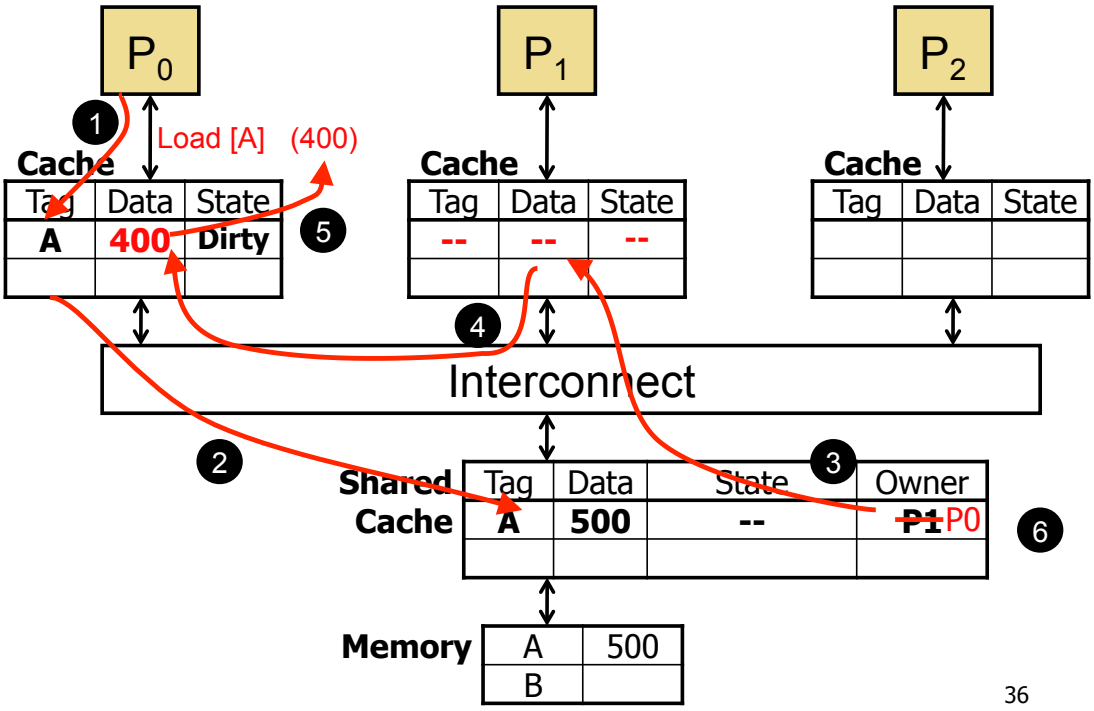
Use Tracking Information to "Invalidate"



Use Tracking Information to "Invalidate"



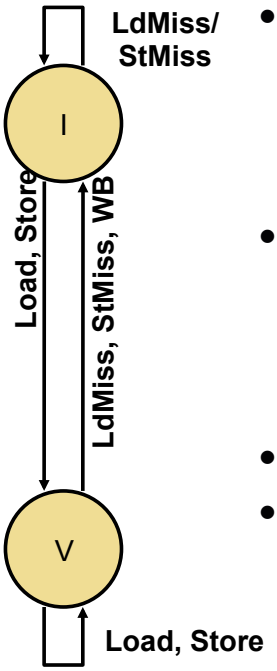
Use Tracking Information to "Invalidate"



…This is "Valid/Invalid" Cache Coherence

- To enforce the shared memory invariant…
  - "Loads read the value written by the most recent store"
- Enforce the invariant…
  - **"At most one valid copy of the block"**
  - Simplest form is a **two-state "valid/invalid" protocol**
  - If a core wants a copy, must find and "invalidate" it
- On a cache miss, how is the valid copy found?
  - Option #1 "**Snooping**": broadcast to all, whoever has it responds
  - Option #2: "**Directory**": track sharers with separate structure

VI (MI) Coherence Protocol



- **VI (valid-invalid) protocol**:
  - Two states (per block in cache)
    - **V (valid)**: own the block
    - **I (invalid)**: don't own block

    + Can implement with "valid bit"
  - Protocol state transition (the left figure)
    - Summary
      - If anyone wants to read/write block
      - Give it up: transition to **I** state
      - Write-back if your own copy is dirty
  - This is an **invalidate protocol**
  - VI protocol is inefficient
    - Only one cached copy allowed in entire system
    - Multiple copies can't exist even if read-only
      - Not a problem in example
      - Big problem in reality

VI Protocol State Transition Table

State	*This Processor* Load	Store	*Other Processor* Load Miss	Store Miss
Invalid (I)	Load Miss ➔ V	Store Miss ➔ V	---	---
Valid (V)	Hit	Hit	Send Data ➔ I	Send Data ➔ I

- Rows are "states"
  - I vs V
- Columns are "events"
- Writeback events not shown
- Memory controller not shown
- **Memory sends data when no processor responds**

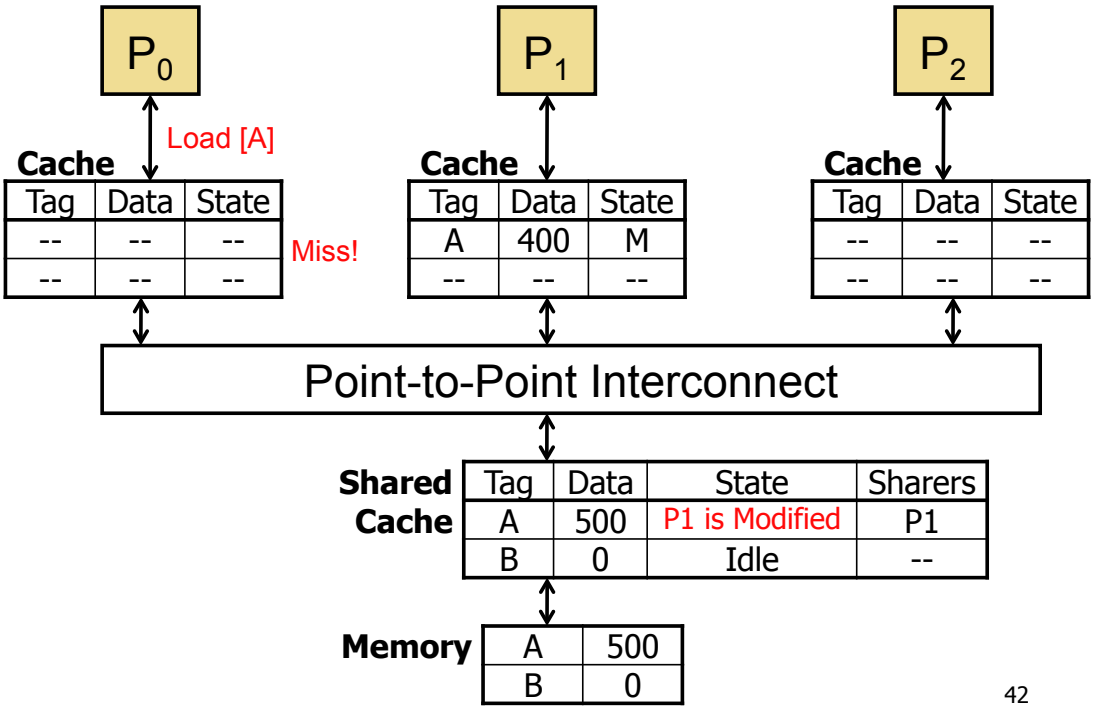
VI (MI) Coherence Inefficiency

- **VI (valid-invalid) protocol** is inefficient
  - Only one cached copy allowed in entire system
  - Multiple copies can't exist even if read-only
    - Not a problem in example
    - Big problem in reality

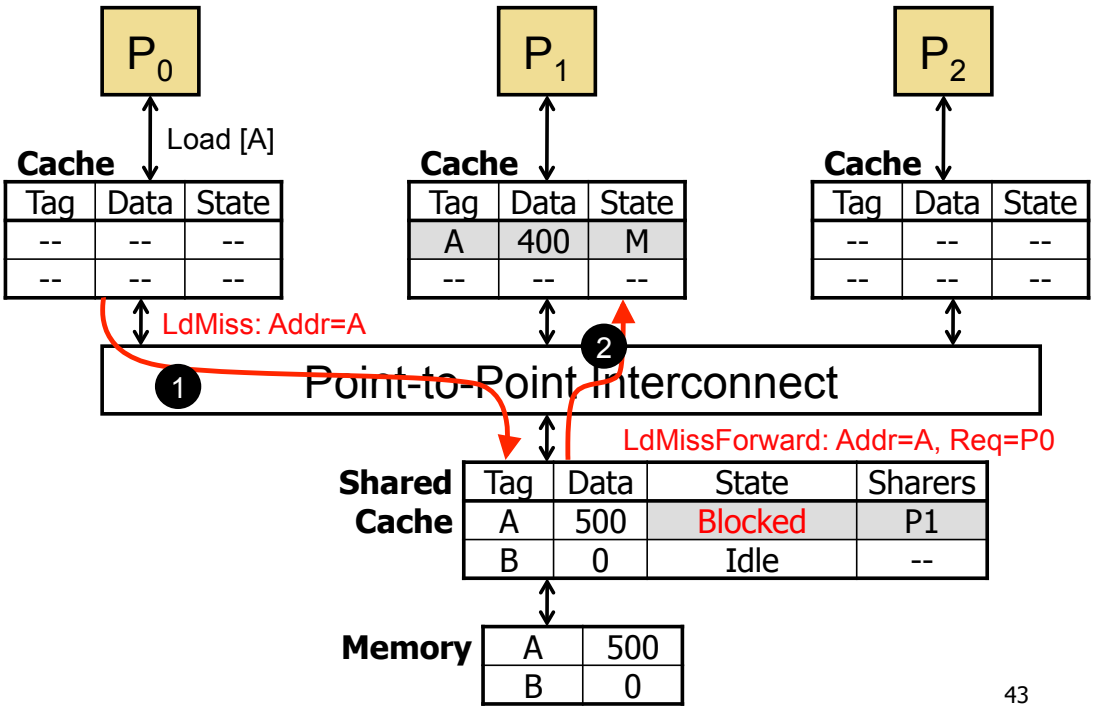
MSI Cache Coherence Protocol

- Solution: enforce the invariant…
  - **Multiple read-only copies —OR—**
  - **Single read/write copy**
- Track these MSI permissions (states) in per-core caches
  - **Modified (M): read/write permission**
  - **Shared (S): read-only permission**
  - **Invalid (I): no permission**
- Also track a **"Sharer" bit vector** in shared cache
  - One bit per core; tracks all shared copies of a block
  - Then, *invalidate all readers* when a write occurs
- Allows for many readers…
  - …while still enforcing shared memory invariant ("Loads read the value written by the most recent store")

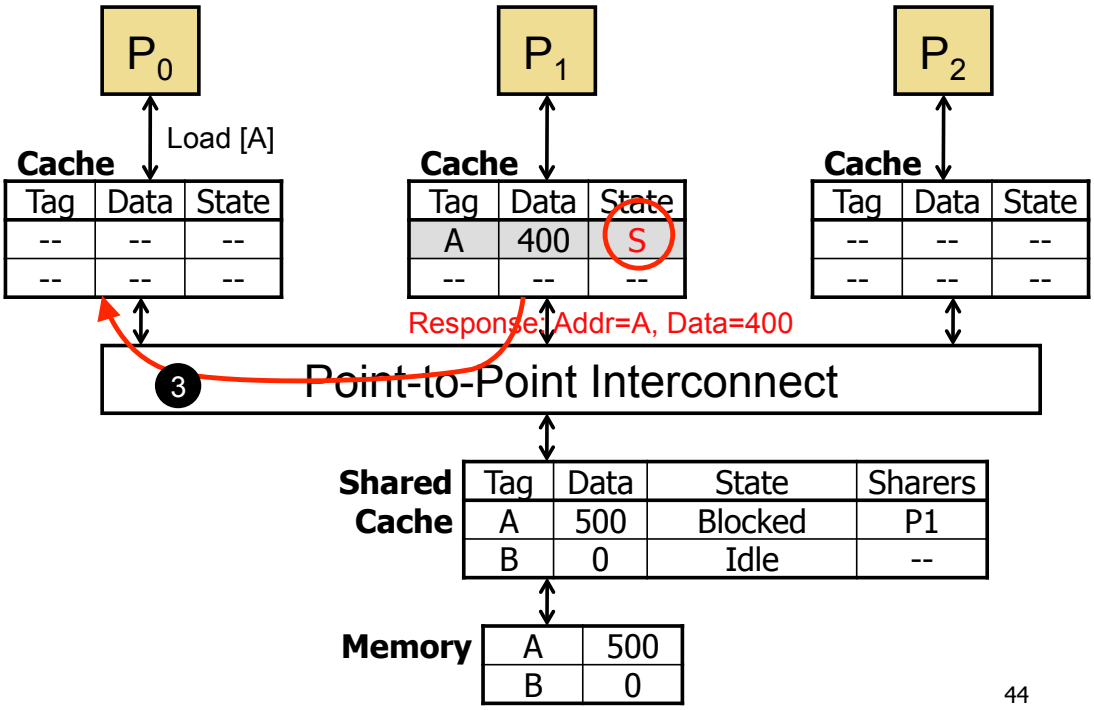
MSI Coherence Example: Step #1



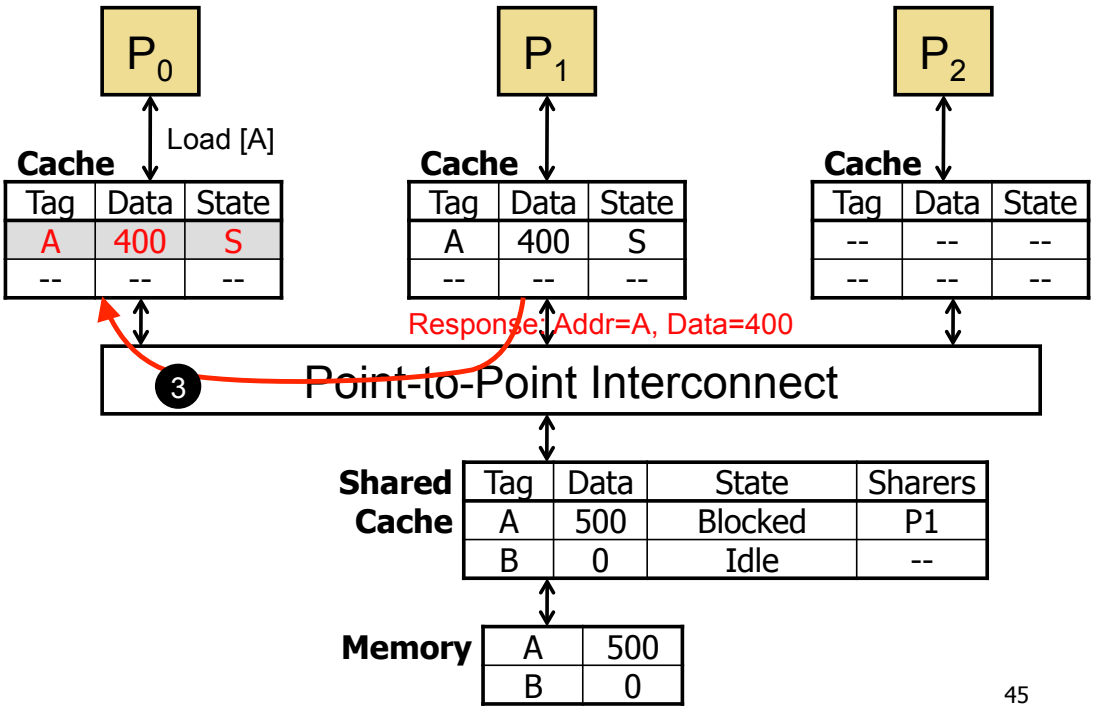
MSI Coherence Example: Step #2



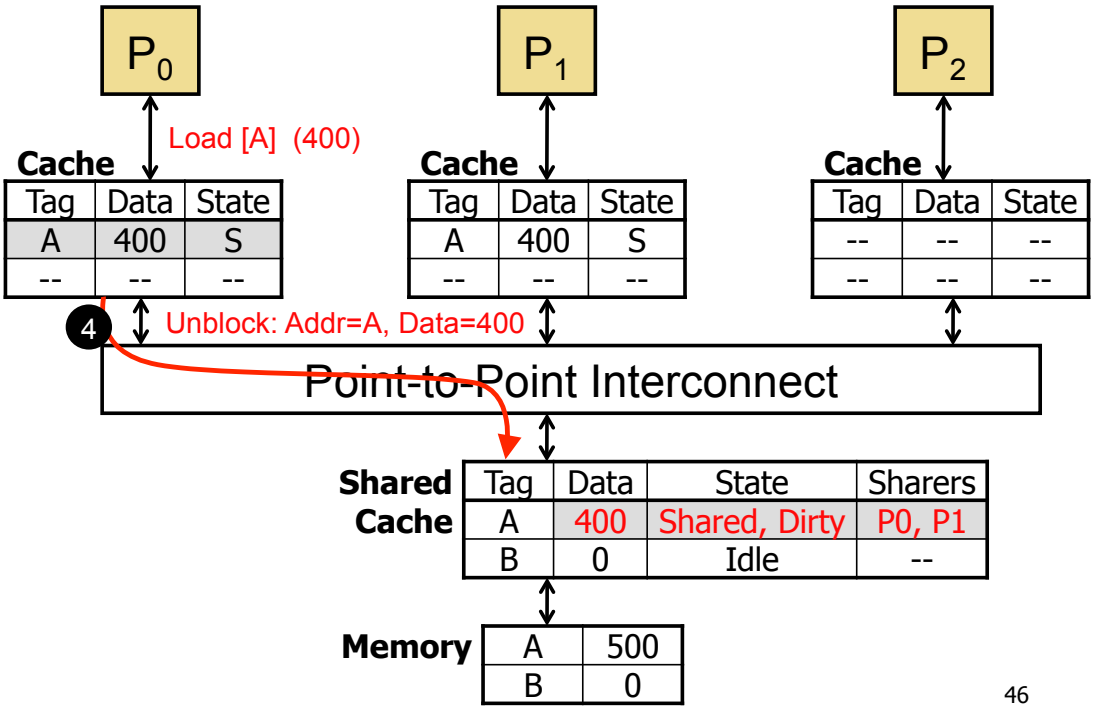
MSI Coherence Example: Step #3



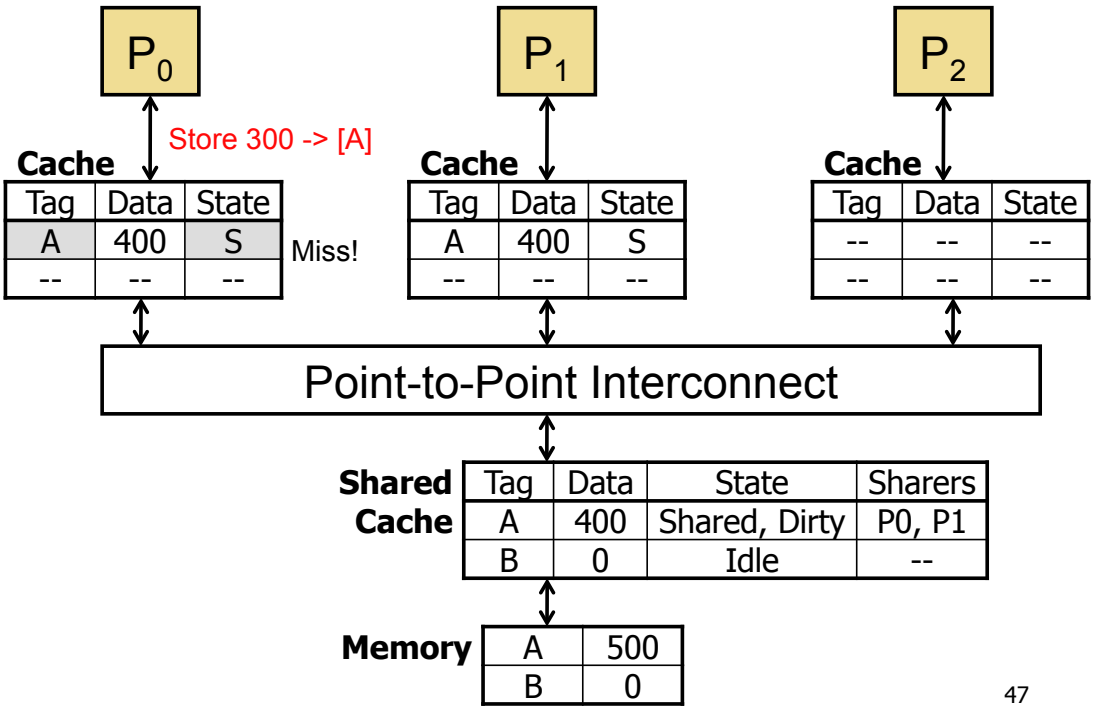
MSI Coherence Example: Step #4



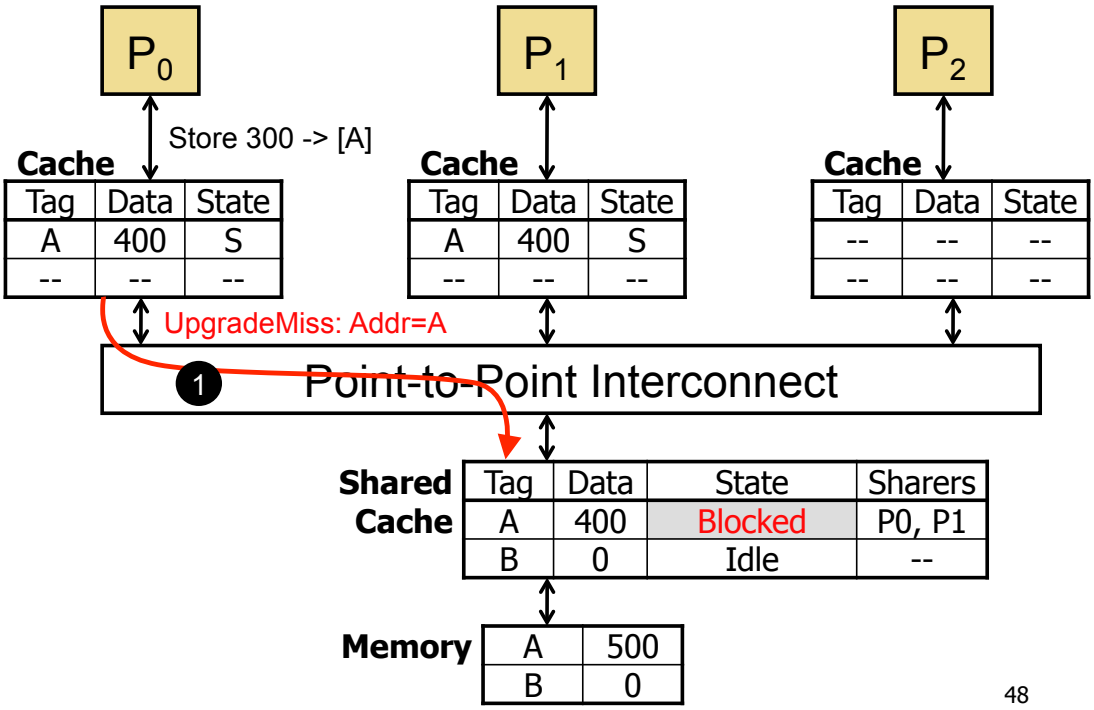
MSI Coherence Example: Step #5



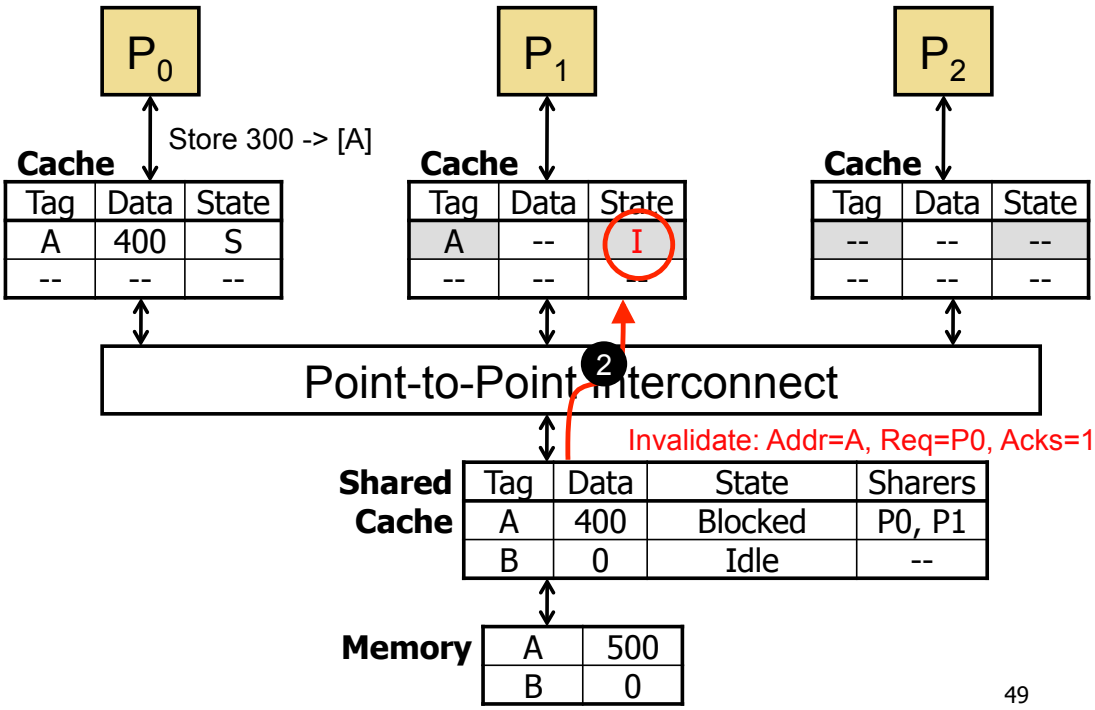
MSI Coherence Example: (store) Step #6



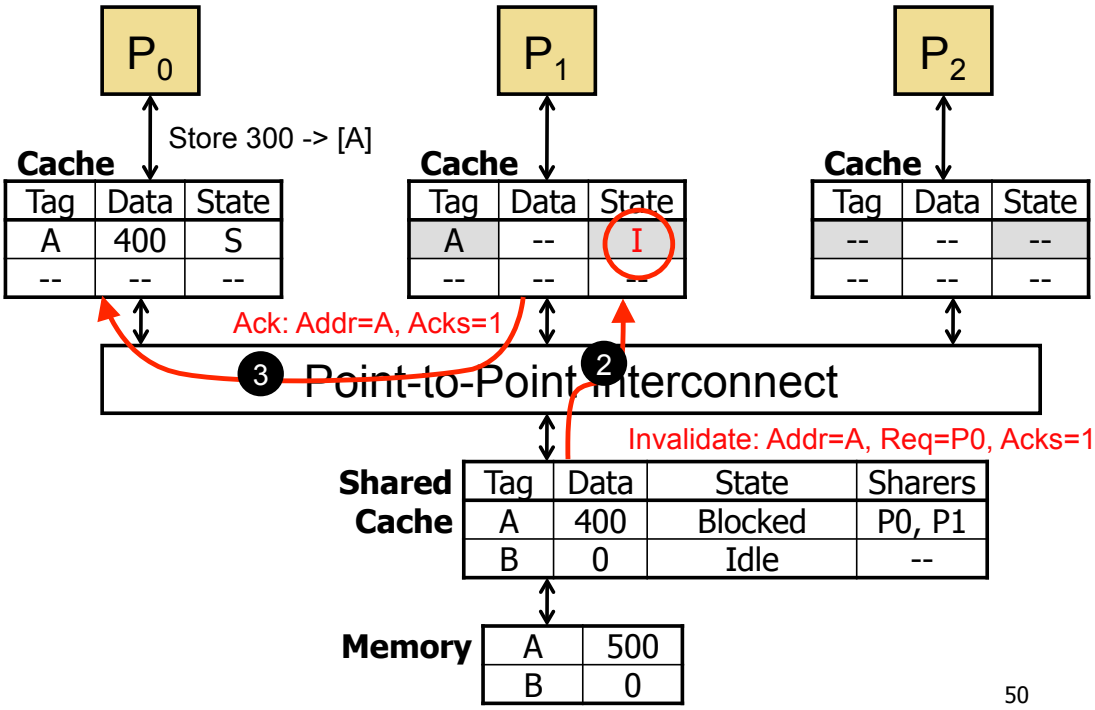
MSI Coherence Example: Step #7



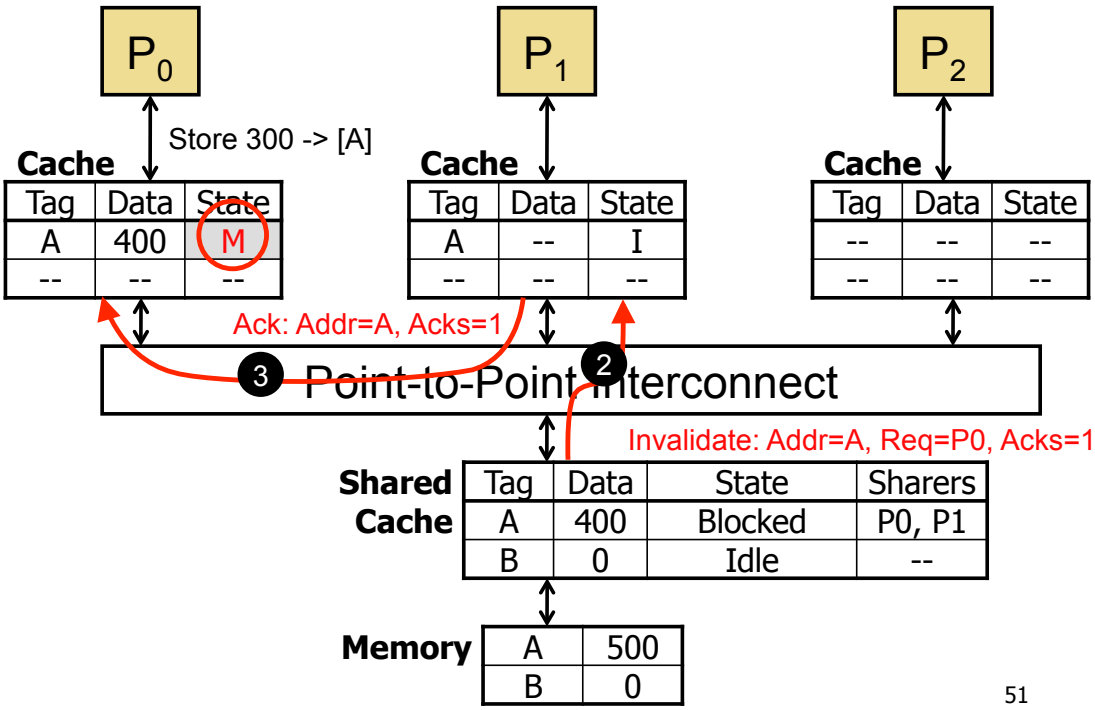
MSI Coherence Example: Step #8



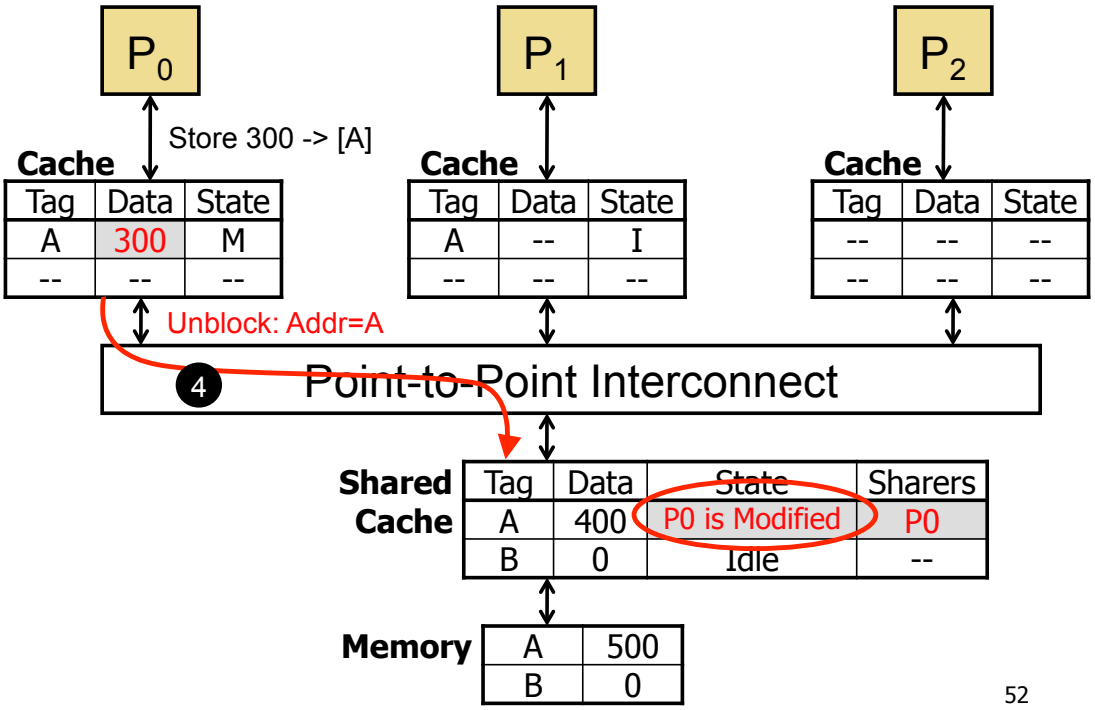
MSI Coherence Example: Step #9



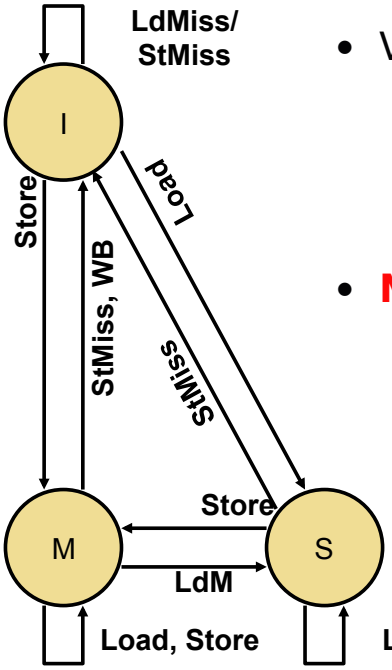
MSI Coherence Example: Step #10



MSI Coherence Example: Step #11



MSI summary



- VI protocol is inefficient
  - Only one cached copy allowed in entire system
  - Multiple copies can't exist even if read-only
    - Not a problem in example
    - Big problem in reality
- **MSI (modified-shared-invalid)**
  - Fixes problem: splits "V" state into two states
    - **M (modified)**: local dirty copy
    - **S (shared)**: local clean copy
  - Allows **either**
    - Multiple read-only copies (S-state) **--OR--**
    - Single read/write copy (M-state)

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)

Classifying Misses: 3C Model

- Divide cache misses into three categories
  - **Compulsory (cold)**: never seen this address before
    - **Would miss even in infinite cache**
  - **Capacity**: miss caused because cache is too small
    - **Would miss even in fully associative cache**
    - Identify? Consecutive accesses to block separated by access to at least N other distinct blocks (N is number of frames in cache)
  - **Conflict**: miss caused because cache associativity is too low
    - Identify? **All other misses**
- One more case of cache misses
  - **(COHERENCE): MISS DUE TO EXTERNAL INVALIDATIONS**
    - **ONLY IN SHARED MEMORY MULTIPROCESSORS**