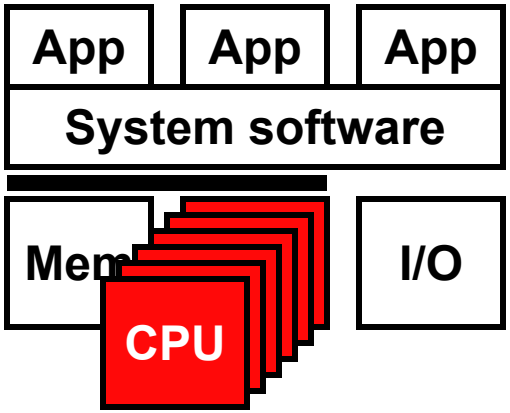
Computer Architecture

Lec 12: Multicore

This Unit: Shared Memory Multiprocessors



- Thread-level parallelism (TLP)
- Shared memory model
  - Multiplexed uniprocessor
  - Hardware multithreading
  - Multiprocessing
- Cache coherence
  - Valid/Invalid, MSI, MESI etc
- Memory Consistency

Beyond Implicit Parallelism

- Consider "daxpy":

```
double a, x[SIZE], y[SIZE], z[SIZE];
void daxpy():
  for (i = 0; i < SIZE; i++)
    z[i] = a*x[i] + y[i];
```

- Lots of instruction-level parallelism (ILP)
  - Great!
  - But how much can we really exploit? 4 wide? 8 wide?
    - Limits to (efficient) super-scalar execution

- But, if SIZE is 10,000 the loop has 10,000-way parallelism!
  - How do we exploit it?

Explicit Parallelism

- Consider "daxpy":

```
double a, x[SIZE], y[SIZE], z[SIZE];
void daxpy():
  for (i = 0; i < SIZE; i++)
    z[i] = a*x[i] + y[i];
```

- Break it up into N "chunks" on N cores!
  - Done by the programmer (or maybe a *really* smart compiler)

```
void daxpy(int chunk_id):
  chuck_size = SIZE / N
  my_start = chuck_id * chuck_size
  my_end = my_start + chuck_size
  for (i = my_start; i < my_end; i++)
    z[i] = a*x[i] + y[i]
```

SIZE = 400, N=4

Chunk ID	Start	End
0	0	99
1	100	199
2	200	299
3	300	399

- Assumes
  - Local variables are "private" and x, y, and z are "shared"
  - Assumes SIZE is a multiple of N (that is, SIZE % N == 0)

Explicit Parallelism

- Consider "daxpy":

```
double a, x[SIZE], y[SIZE], z[SIZE];
void daxpy(int chunk_id):
  chuck_size = SIZE / N
  my_start = chuck_id * chuck_size
  my_end = my_start + chuck_size
  for (i = my_start; i < my_end; i++)
    z[i] = a*x[i] + y[i]
```

- Main code then looks like:

```
parallel_daxpy():
  for (tid = 0; tid < CORES; tid++) {
    spawn_task(daxpy, tid);
  }
  wait_for_tasks(CORES);
```

Explicit (Loop-Level) Parallelism

- Another way: "OpenMP" annotations to inform the compiler

```
double a, x[SIZE], y[SIZE], z[SIZE];
void daxpy() {
  #pragma omp parallel for
  for (i = 0; i < SIZE; i++) {
    z[i] = a*x[i] + y[i];
  }
```

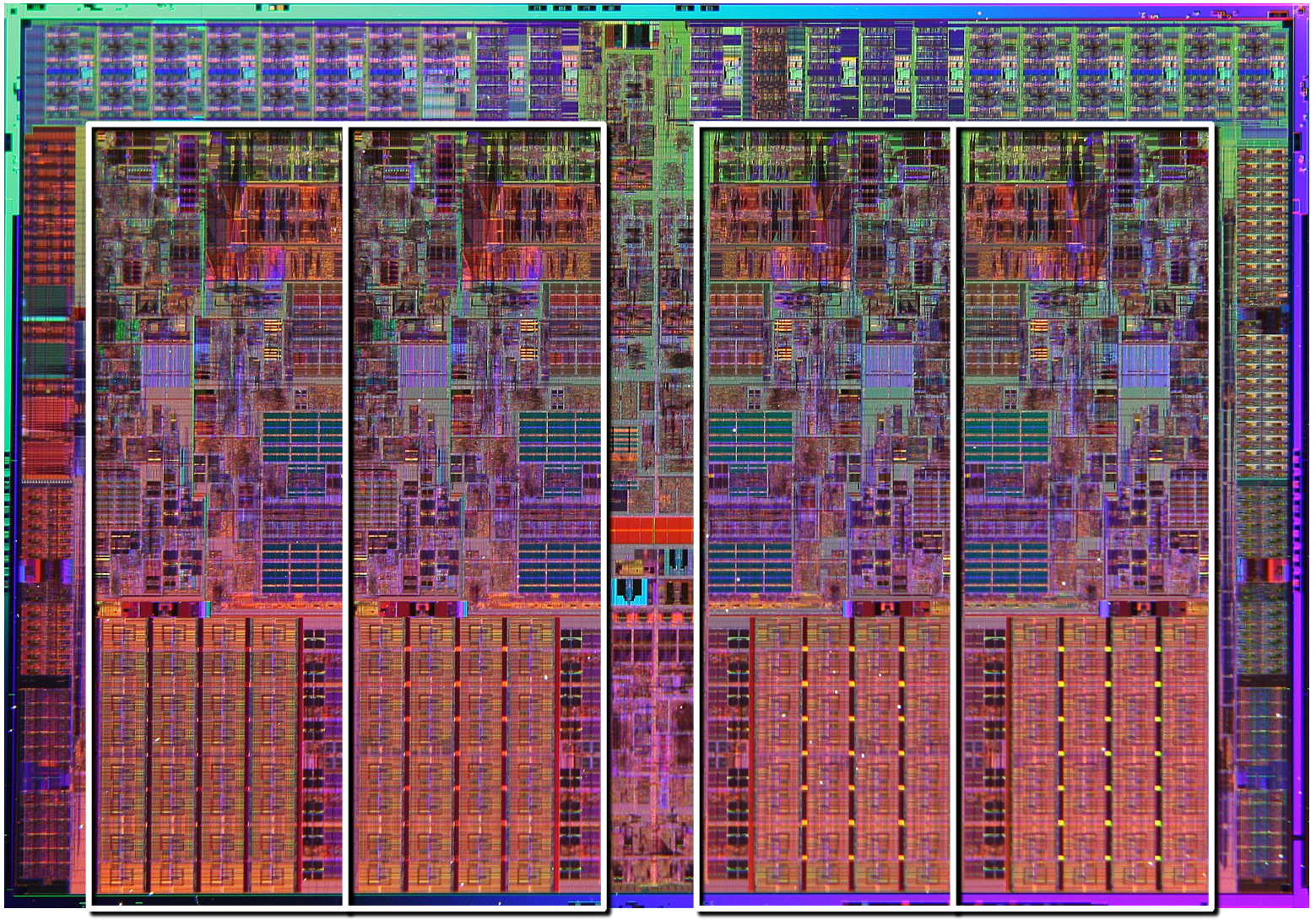
- But only works if loop is actually parallel
  - If not parallel, unpredictable incorrect behavior may result

Multicore & Multiprocessor Hardware

Multiplying Performance

- A single core can only be so fast
  - Limited clock frequency
  - Limited instruction-level parallelism
- What if we need even more computing power?
  - Use multiple cores! But how?
- Old-school (2000s): Ultra Enterprise 25k
  - 72 dual-core UltraSPARC IV+ processors
  - Up to 1TB of memory
  - Niche: large database servers
  - $$$, weighs more than 1 ton
- Today: multicore is everywhere
  - Can't buy a single-core smartphone…
    - …or even smart watch!

Intel Quad-Core “Core i7”



Application Domains for Multiprocessors

- **Scientific computing/supercomputing**
  - Examples: weather simulation, aerodynamics, protein folding
  - Large grids, integrating changes over time
  - Each processor computes for a part of the grid
- **Server workloads**
  - Example: airline reservation database
  - Many concurrent updates, searches, lookups, queries
  - Processors handle different requests
- **Media workloads**
  - Processors compress/decompress different parts of image/frames
- **Desktop workloads…**
- **Gaming workloads…**

But software must be written to expose parallelism

Multicore is Energy Efficient

- Explicit parallelism (multicore) is highly energy efficient
- Recall: dynamic voltage and frequency scaling
  - Performance vs power is NOT linear
  - Example: Intel's Xscale
    - 1 GHz → 200 MHz reduces energy used by 30x
- Consider the impact of parallel execution
  - What if we used 5 Xscales at 200Mhz?
  - Similar performance as a 1Ghz Xscale, but **1/6th the energy**
    - 5 cores * 1/30th = 1/6th
- And, amortizes background "uncore" energy among cores
- Assumes parallel speedup (a difficult task)
  - Subject to Ahmdal's law

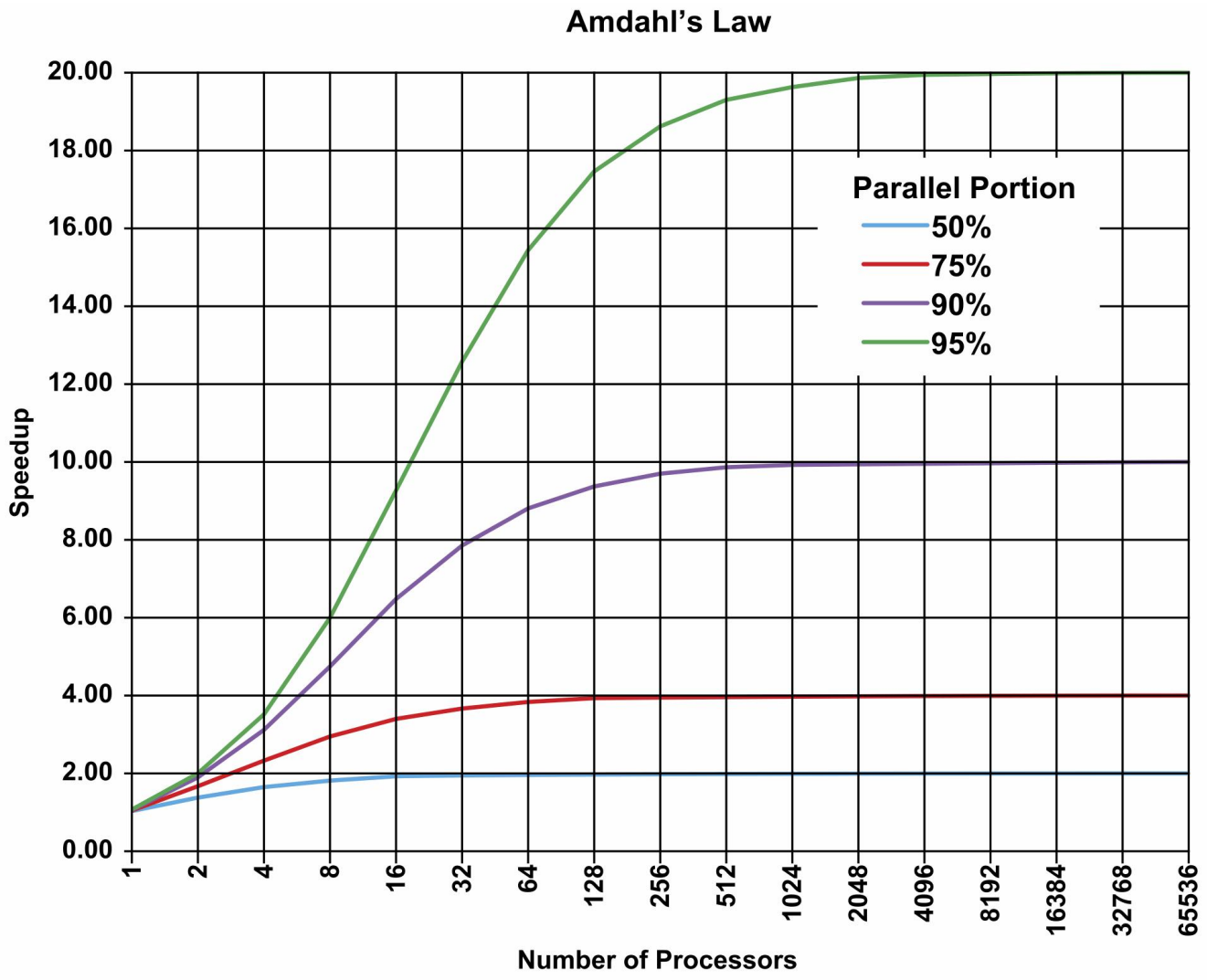
Amdahl's Law

- Restatement of the law of diminishing returns
  - Total speedup limited by non-accelerated piece
  - Analogy: drive to work & park car, walk to building

- Consider a task with a "parallel" and "serial" portion
  - What is the speedup with N cores?
  - Speedup(n, p, s) = (s+p) / (s + (p/n))
    - p is "parallel percentage", s is "serial percentage"
  - What about infinite cores?
    - Speedup(p, s) = (s+p) / s = 1 / s

- Example: can optimize 50% of program A
  - Even a "magic" optimization that makes this 50% disappear…
  - …only yields a 2X speedup

Amdahl's Law Graph



Source: Wikipedia

Threading &
The Shared Memory Programming Model

First, Uniprocessor Concurrency

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

Shared Memory Programming Model

- Programmer explicitly creates multiple threads
- All loads & stores to a single **shared memory** space
  - Each thread has its own stack frame for local variables
  - All memory shared, accessible by all threads
- 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 Model: Interleaving

- **Initially: all variables zero** (that is, x=0, y=0)

thread 1	thread 2
store 1 → y load x	store 1 → x load y

- What value pairs can be read by the two loads?

store 1 → y load x store 1 → x load y (x=0, y=1)	store 1 → y store 1 → x load x load y (x=1, y=1)	store 1 → y store 1 → x load y load x (x=1, y=1)
store 1 → x load y store 1 → y load x (x=1, y=0)	store 1 → x store 1 → y load y load x (x=1, y=1)	store 1 → x store 1 → y load x load y (x=1, y=1)

- What about (x=0, y=0)?

Shared Memory Implementations

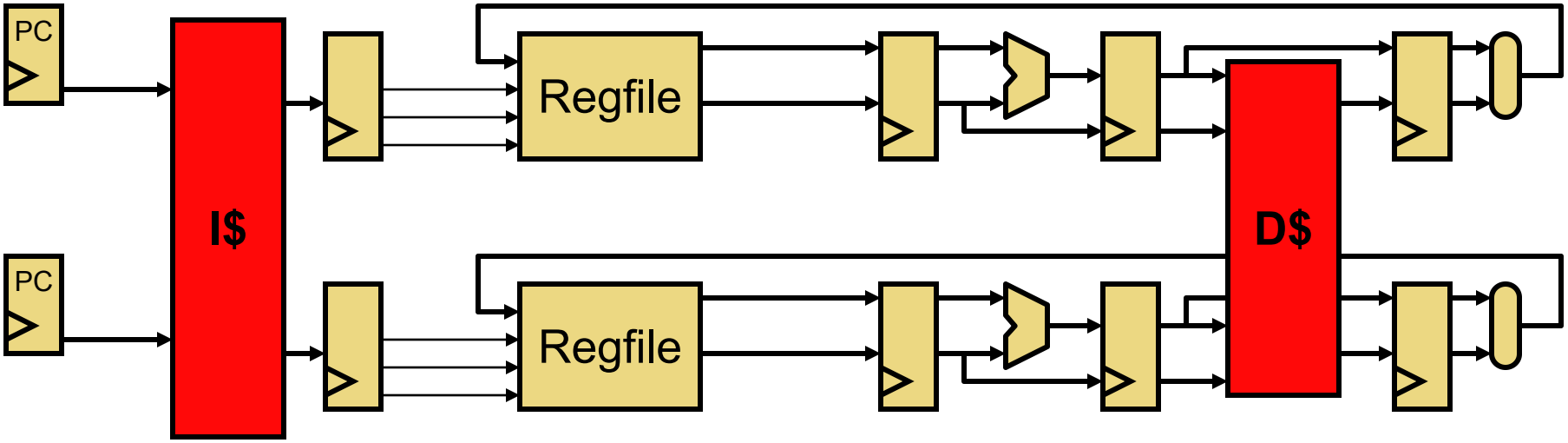
- **Multiplexed uniprocessor**
  - Runtime system and/or OS occasionally pre-empt & swap threads
  - Interleaved, but no parallelism

- **Multiprocessors**
  - Multiply execution resources, higher peak performance
  - Same interleaved shared-memory model
  - Foreshadowing: allow private caches, further disentangle cores

- **Hardware multithreading**
  - Tolerate pipeline latencies, higher efficiency
  - Same interleaved shared-memory model

- **All support the shared memory programming model**

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

Four Shared Memory Issues

1. **Cache coherence**
   - If cores have private (non-shared) caches
   - How to make writes to one cache "show up" in others?

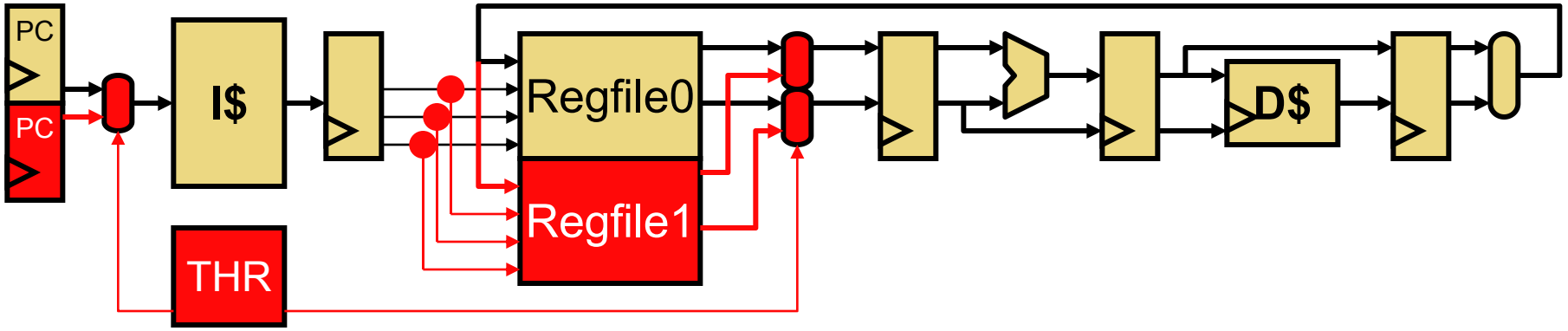
1. **Parallel programming**
   - How does the programmer express the parallelism?

2. **Synchronization**
   - How to regulate access to shared data?
   - How to implement "locks"?

3. **Memory consistency models**
   - How to keep programmer sane while letting hardware optimize?
   - How to reconcile shared memory with compiler optimizations, store buffers, and out-of-order execution?

Hardware Multithreading

- **Not** the same as software multithreading!
- A **hardware thread** is a sequential stream of insns
  - from some **software thread** (e.g., single-threaded process)



- **Hardware Multithreading (MT)**
  - Multiple hardware threads dynamically share a single pipeline
  - Replicate only per-thread structures: program counter & registers
  - Hardware interleaves instructions

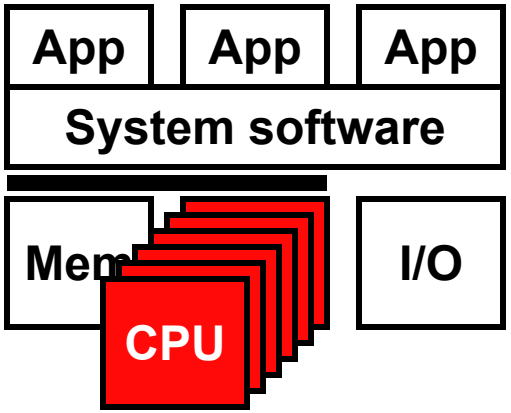
Hardware Multithreading

- Why use hw multithreading?
  - + **Multithreading improves utilization and throughput**
    - Single programs utilize <50% of pipeline (branch, cache miss)
    - allow insns from different hw threads in pipeline at once
  - **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
    - Intel's "hyperthreading"

process A
thread 1
thread 2

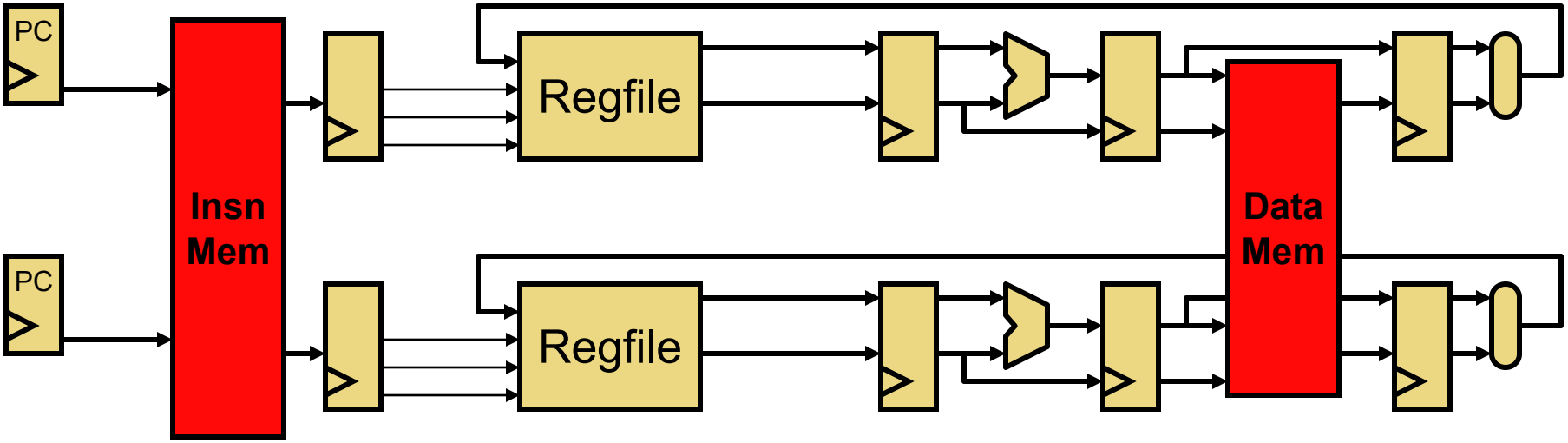
process B
(just one thread)

Roadmap Checkpoint



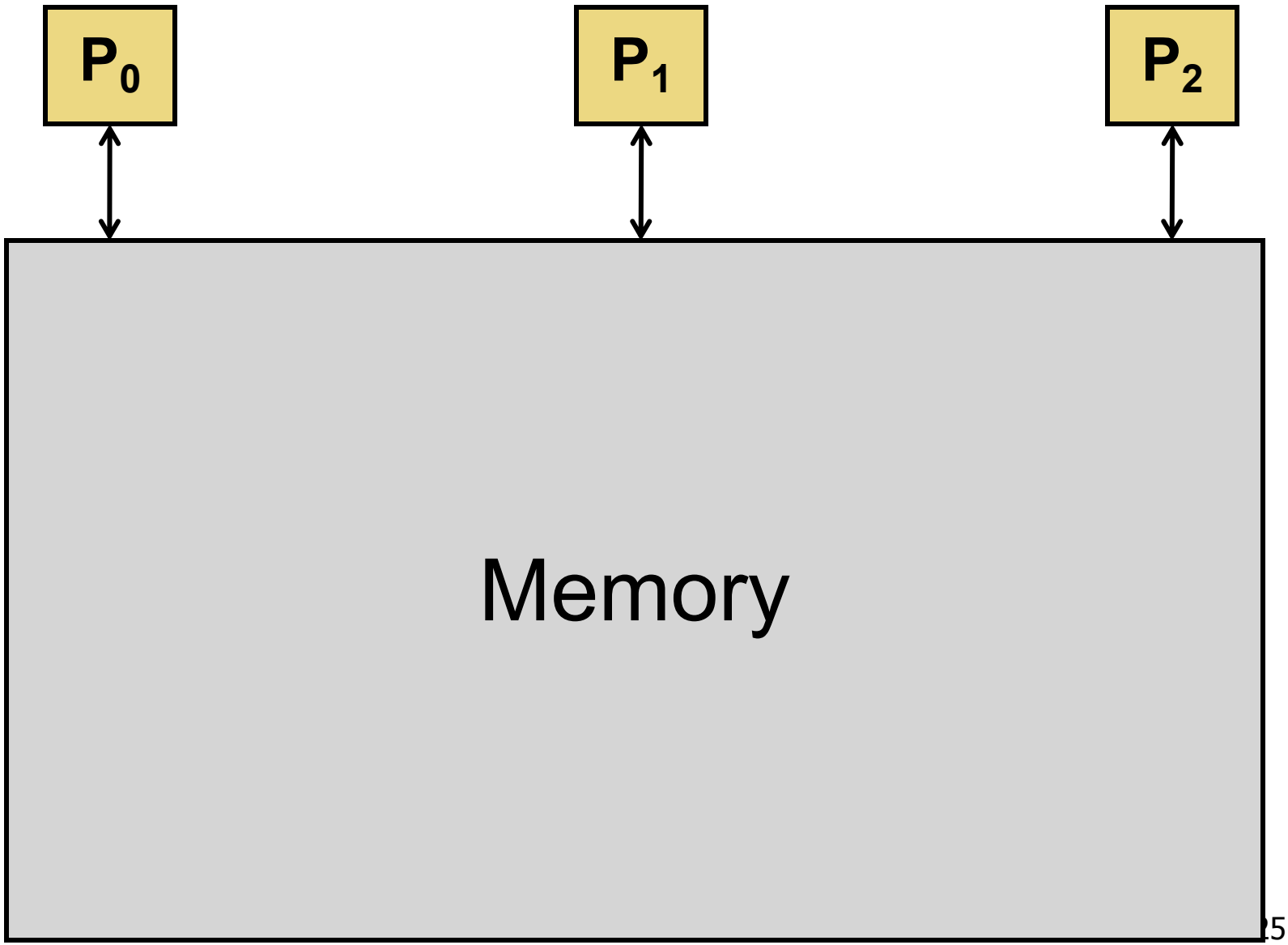
- ~~Thread-level parallelism (TLP)~~
- ~~Shared memory model~~
  - ~~Multiplexed uniprocessor~~
  - ~~Hardware multithreading~~
  - ~~Multiprocessing~~
- **Cache coherence**
  - **Valid/Invalid, MSI, MESI**
- Parallel programming
- Synchronization
  - Lock implementation
  - Locking gotchas
  - Transactional memory
- Memory consistency models

Recall: Simplest Multiprocessor

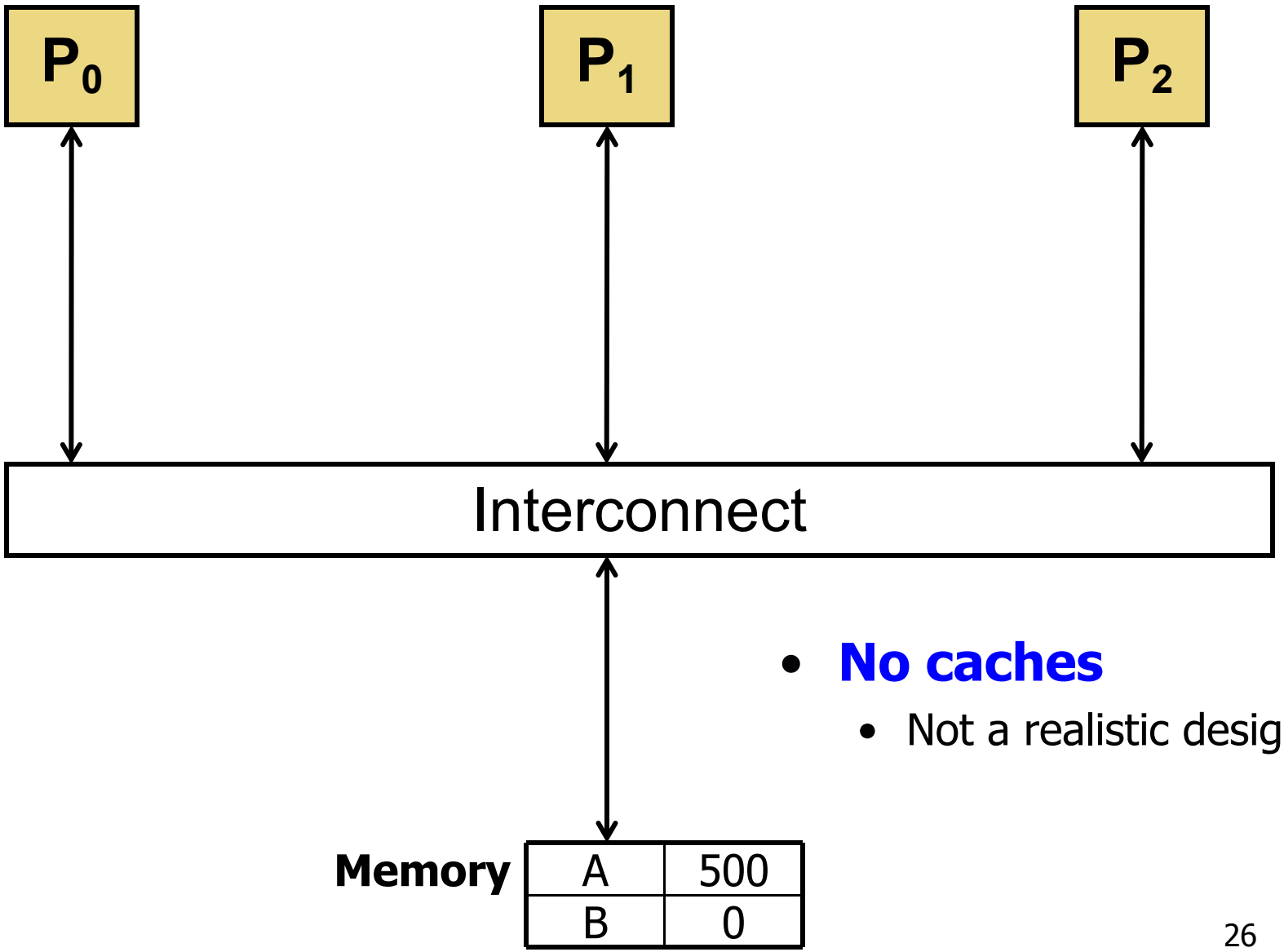


- What if we don't want to share the L1 caches?
  - Bandwidth and latency issue
- Solution: use per-processor ("private") caches
  - Coordinate them with a ***Cache Coherence Protocol***
- Must still provide shared-memory invariant:
  - **"Loads read the value written by the most recent store"**

No-Cache (Conceptual) Implementation

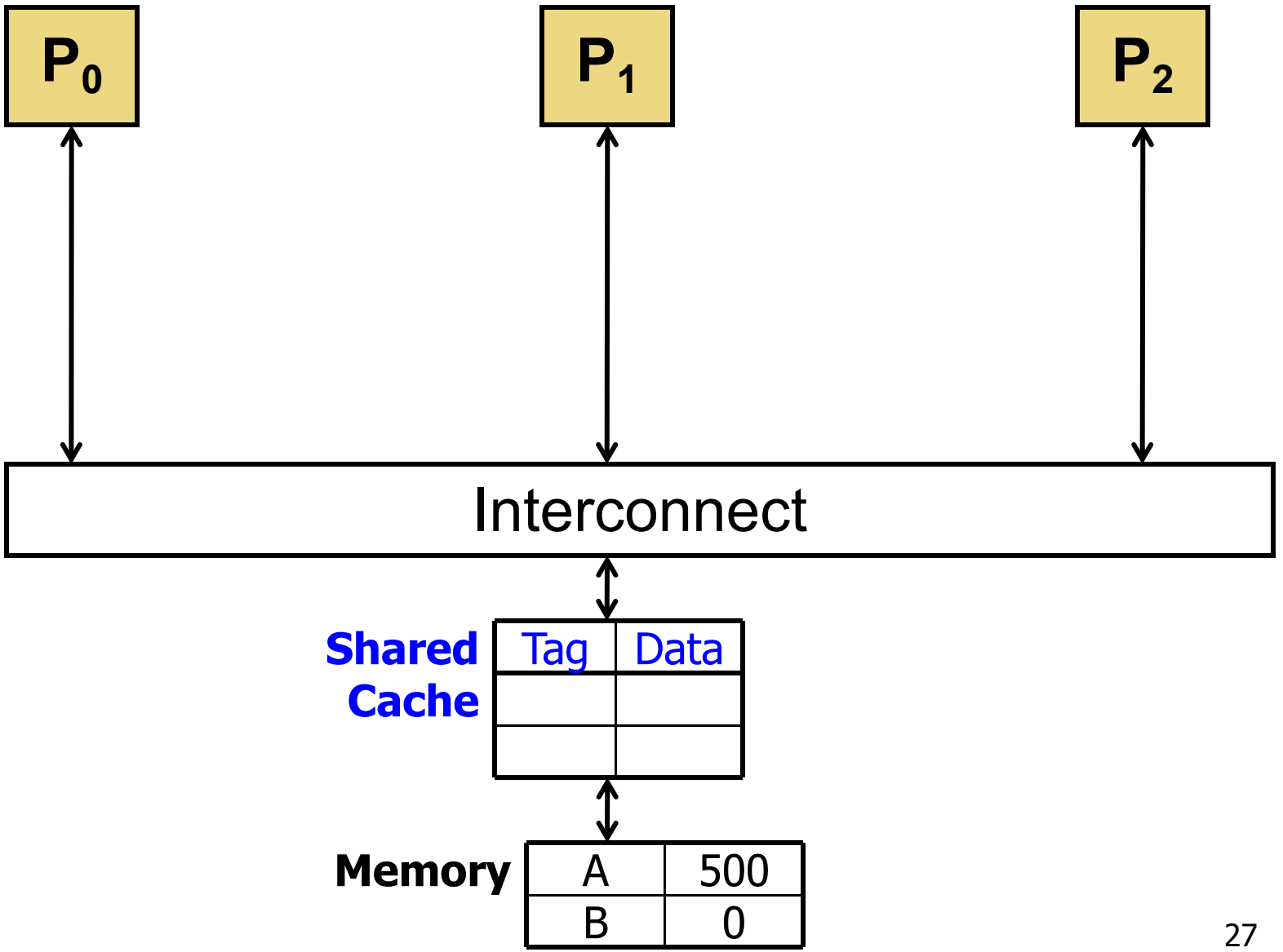


No-Cache (Conceptual) Implementation

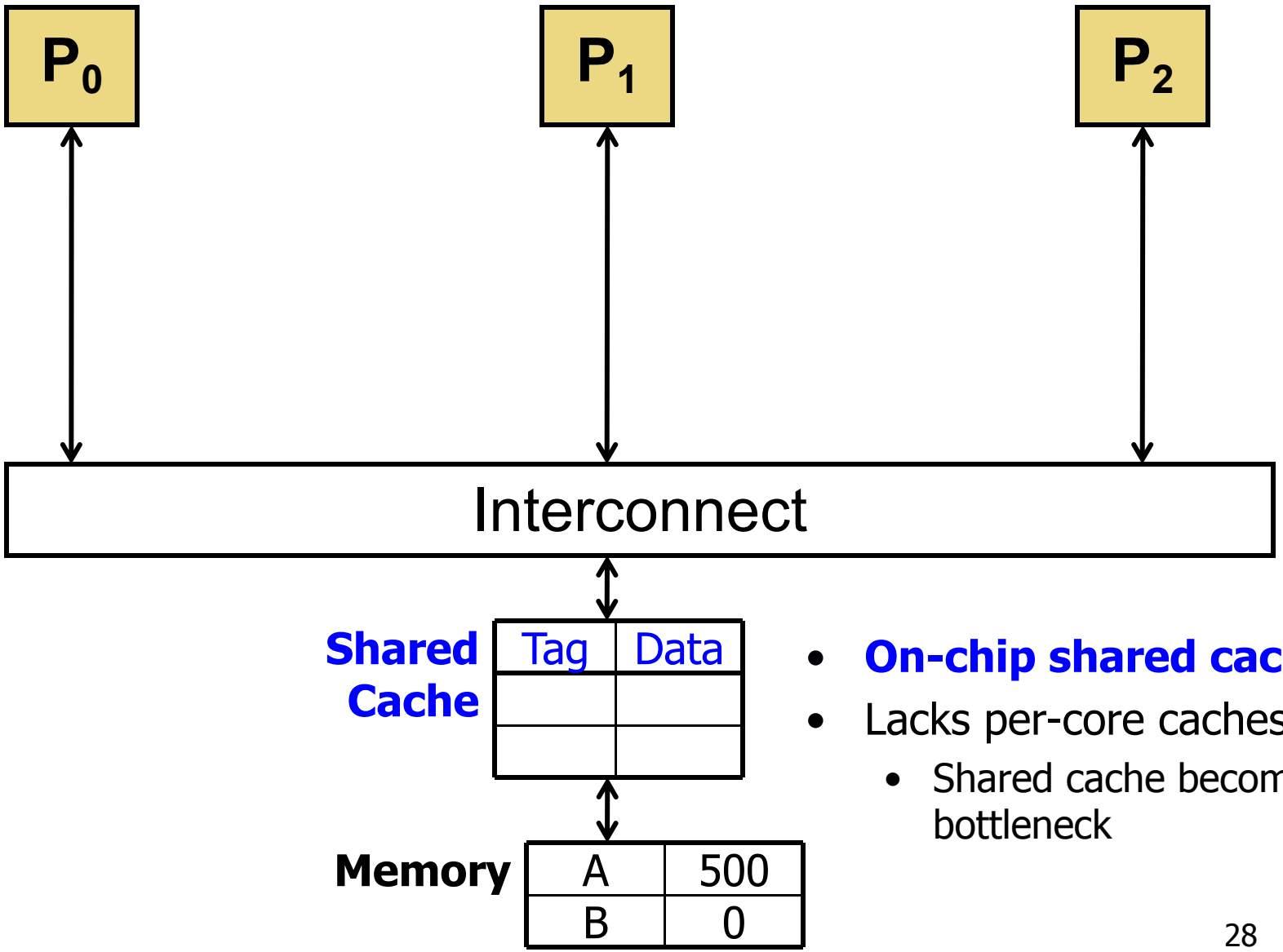


- **No caches**
  - Not a realistic design

Shared Cache Implementation

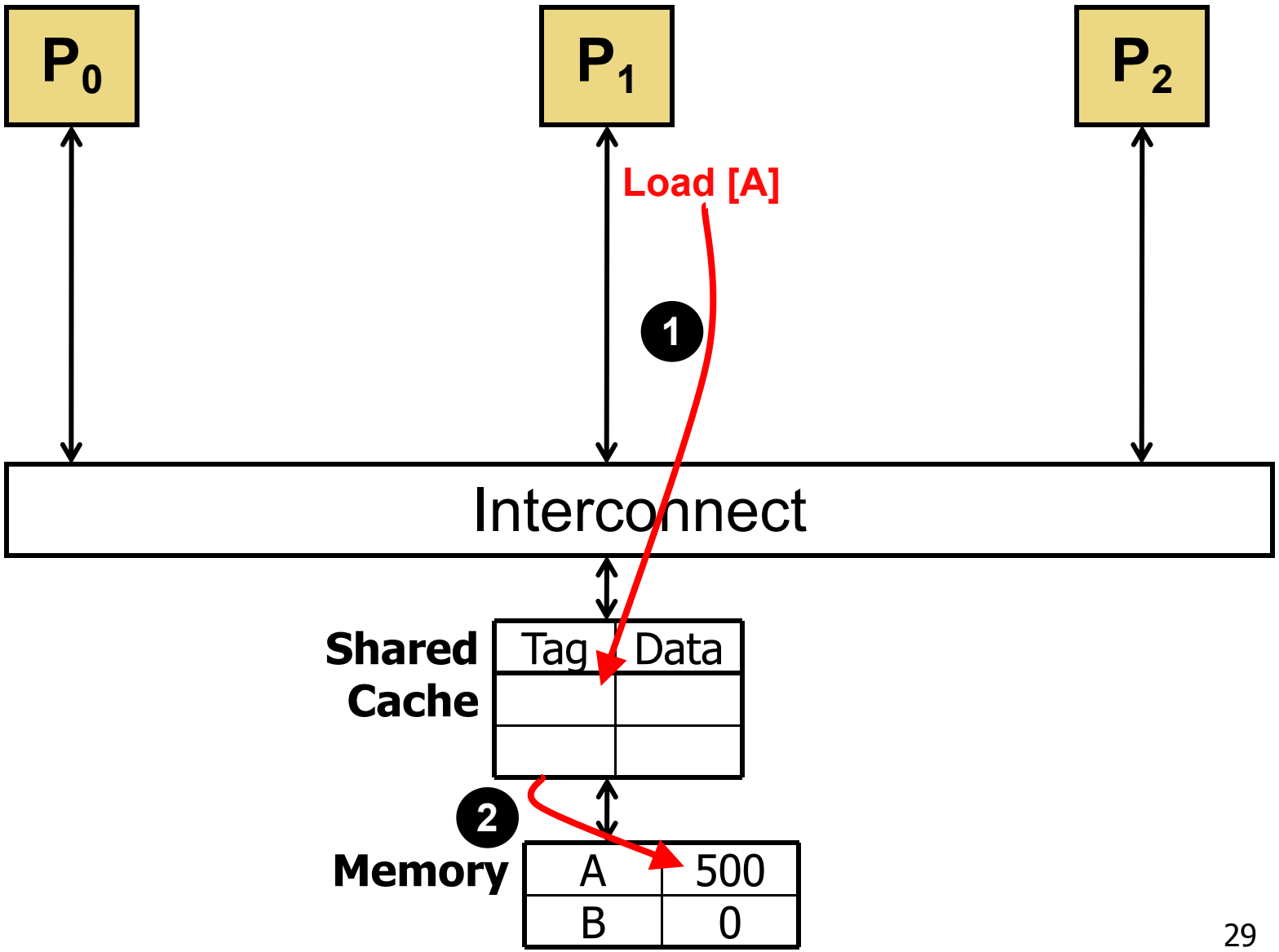


Shared Cache Implementation

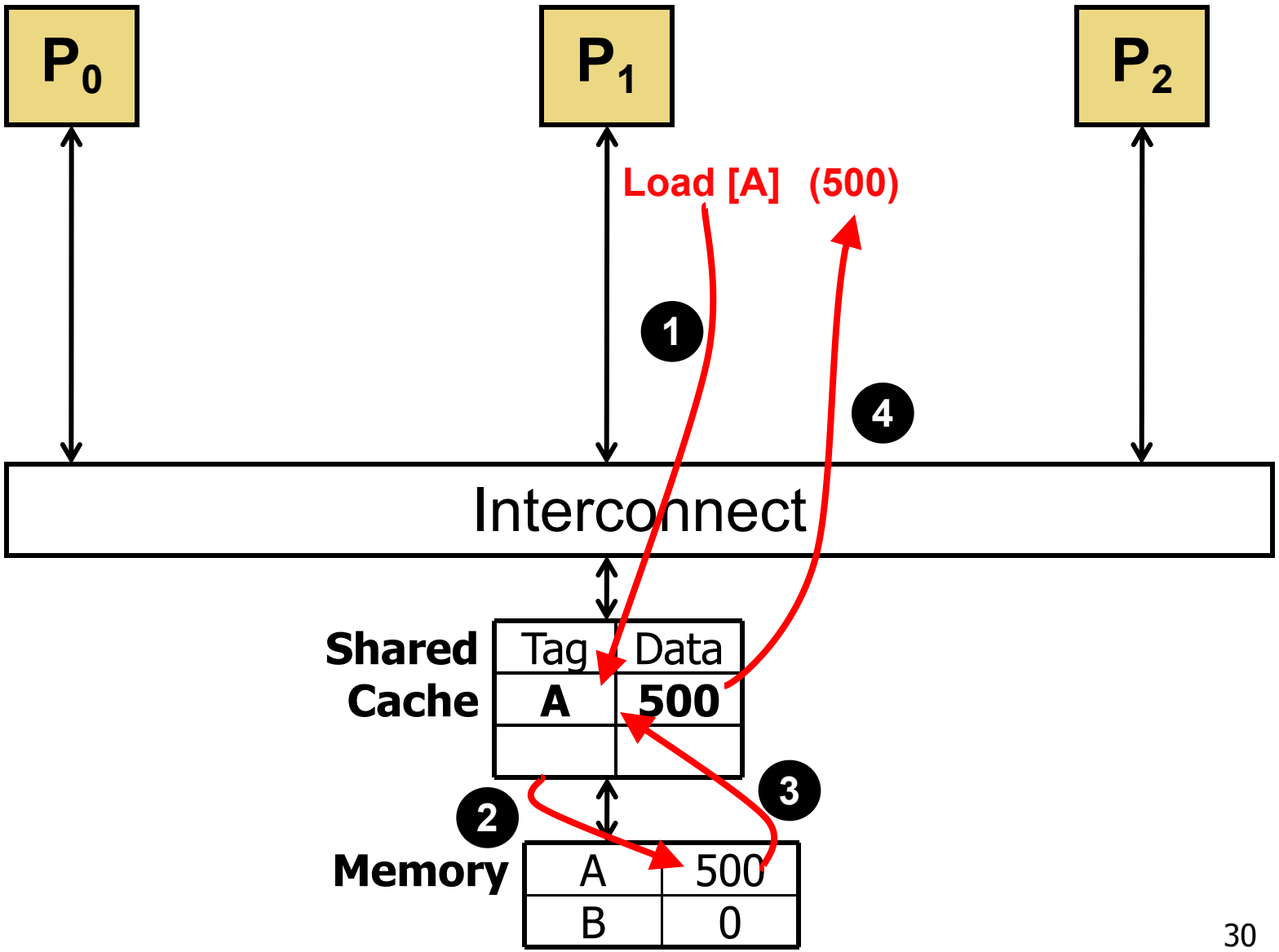


- **On-chip shared cache**
- Lacks per-core caches
  - Shared cache becomes bottleneck

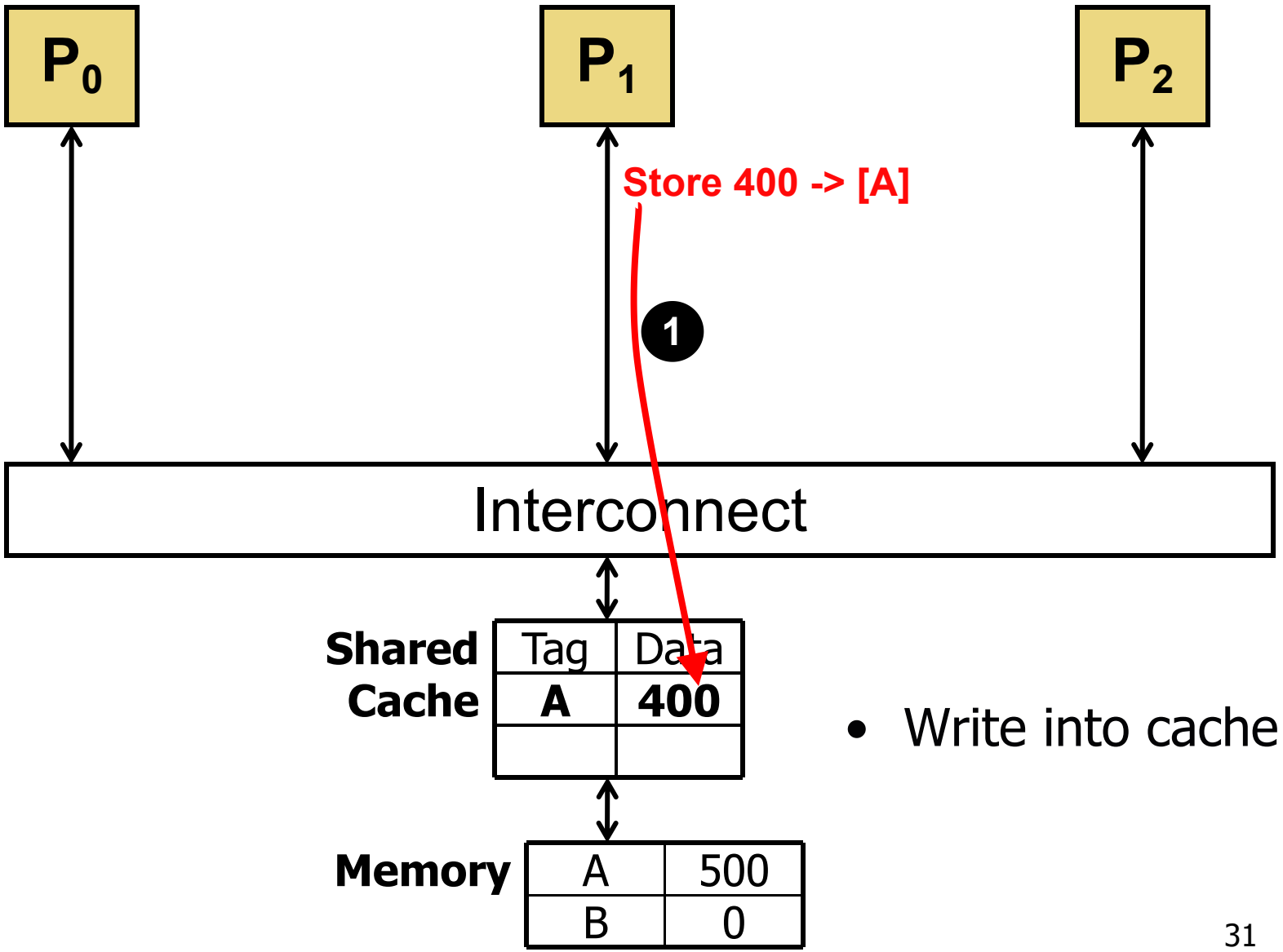
Shared Cache Implementation



Shared Cache Implementation

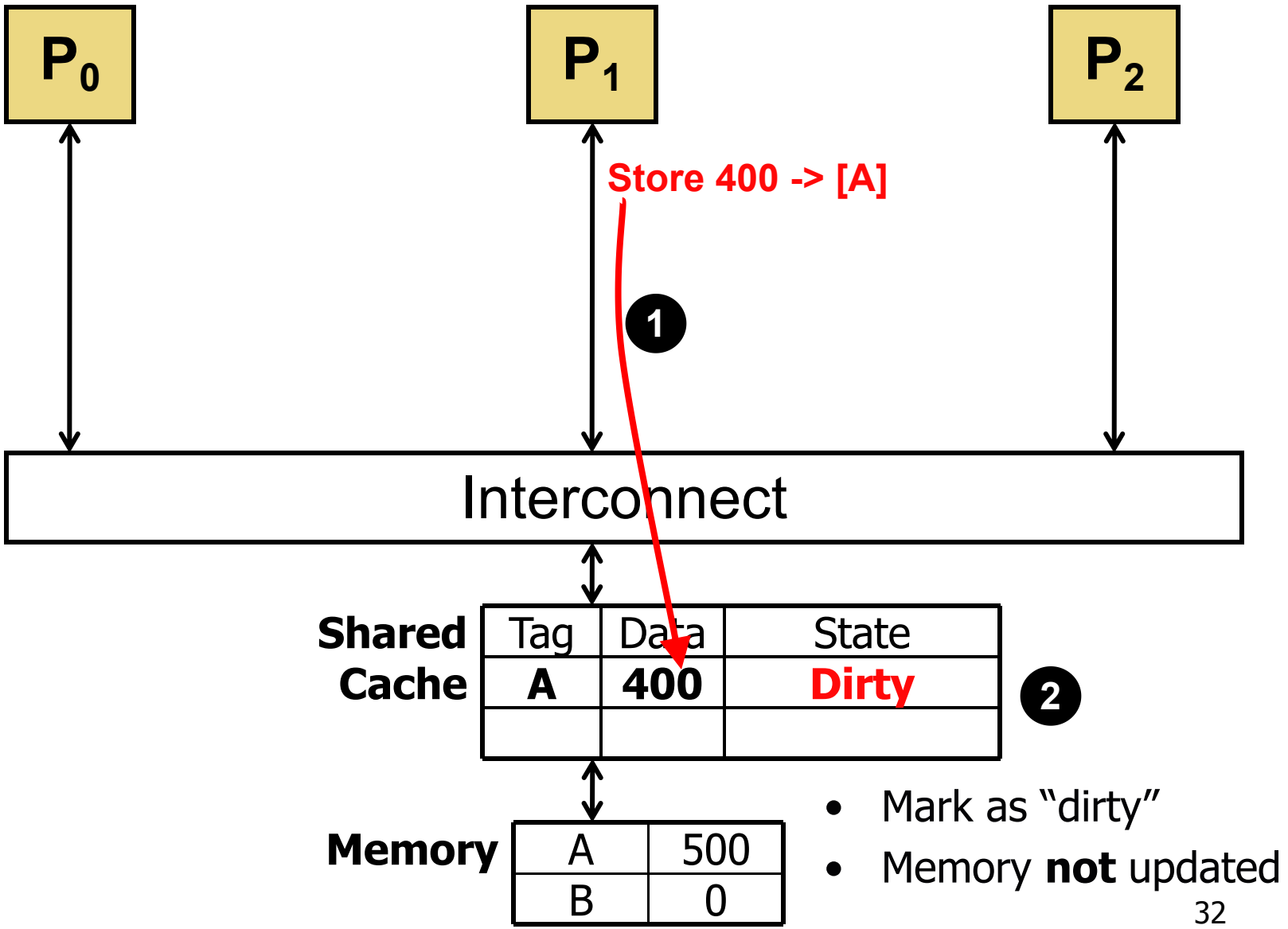


Shared Cache Implementation

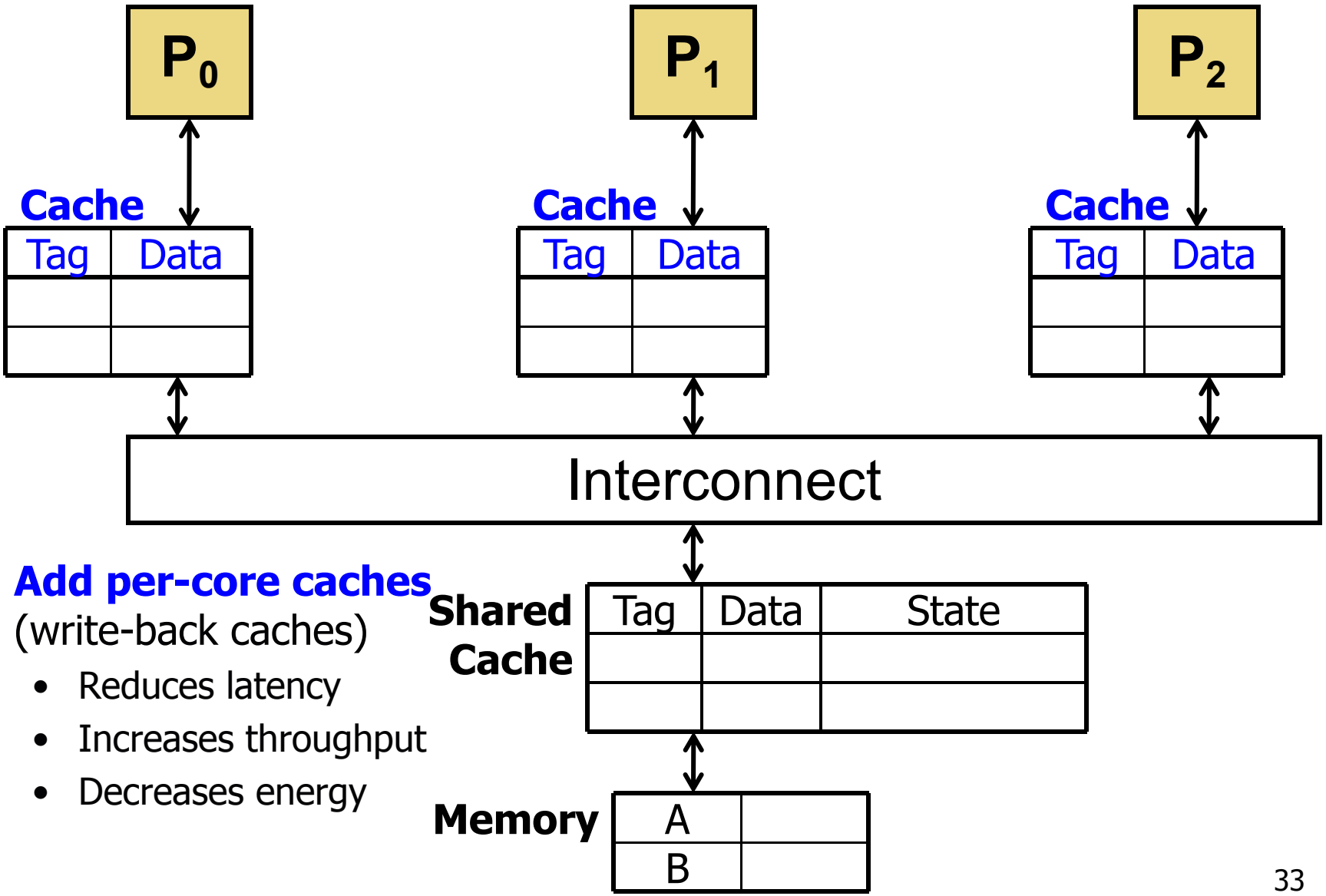


- Write into cache

Shared Cache Implementation

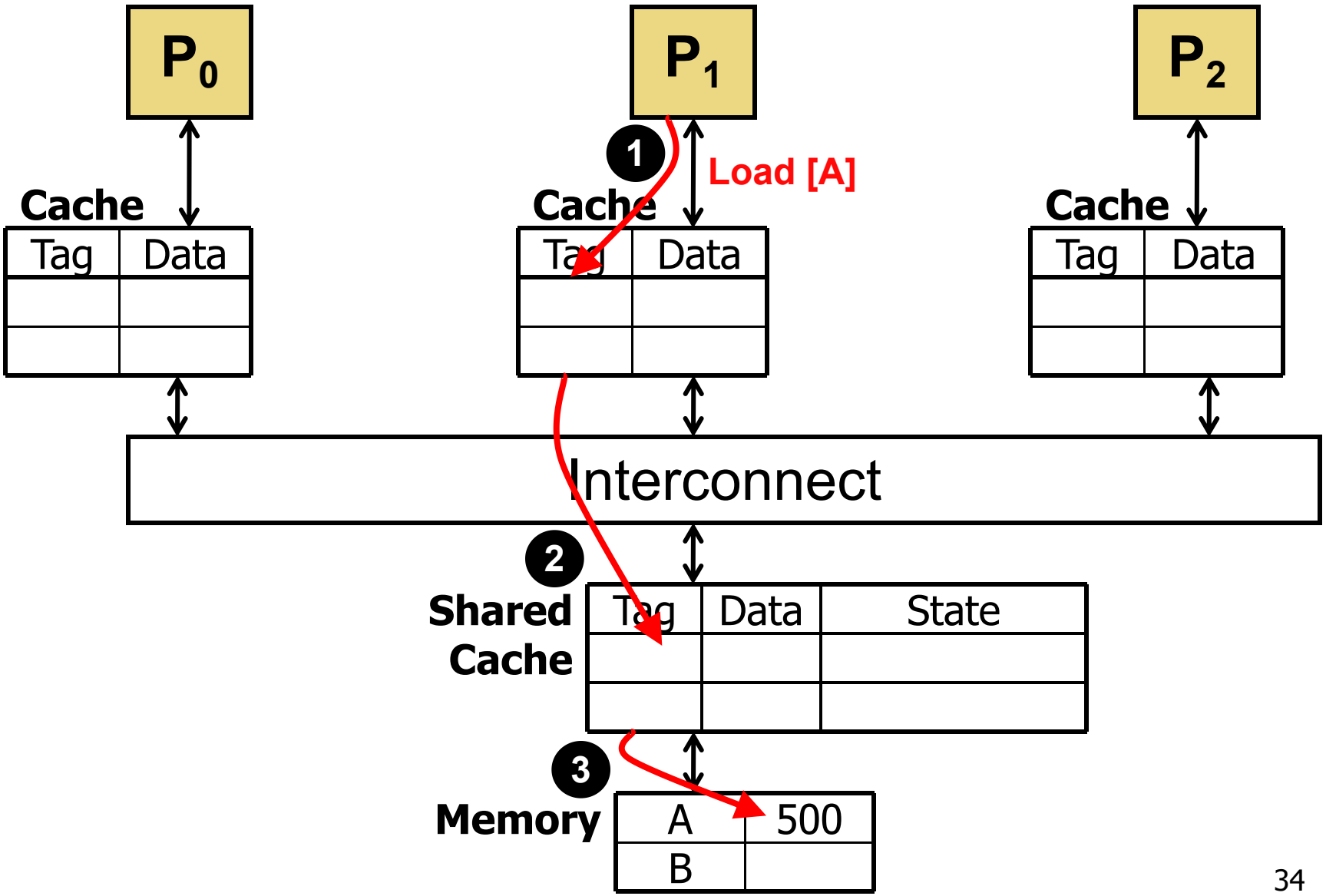


Adding Private Caches

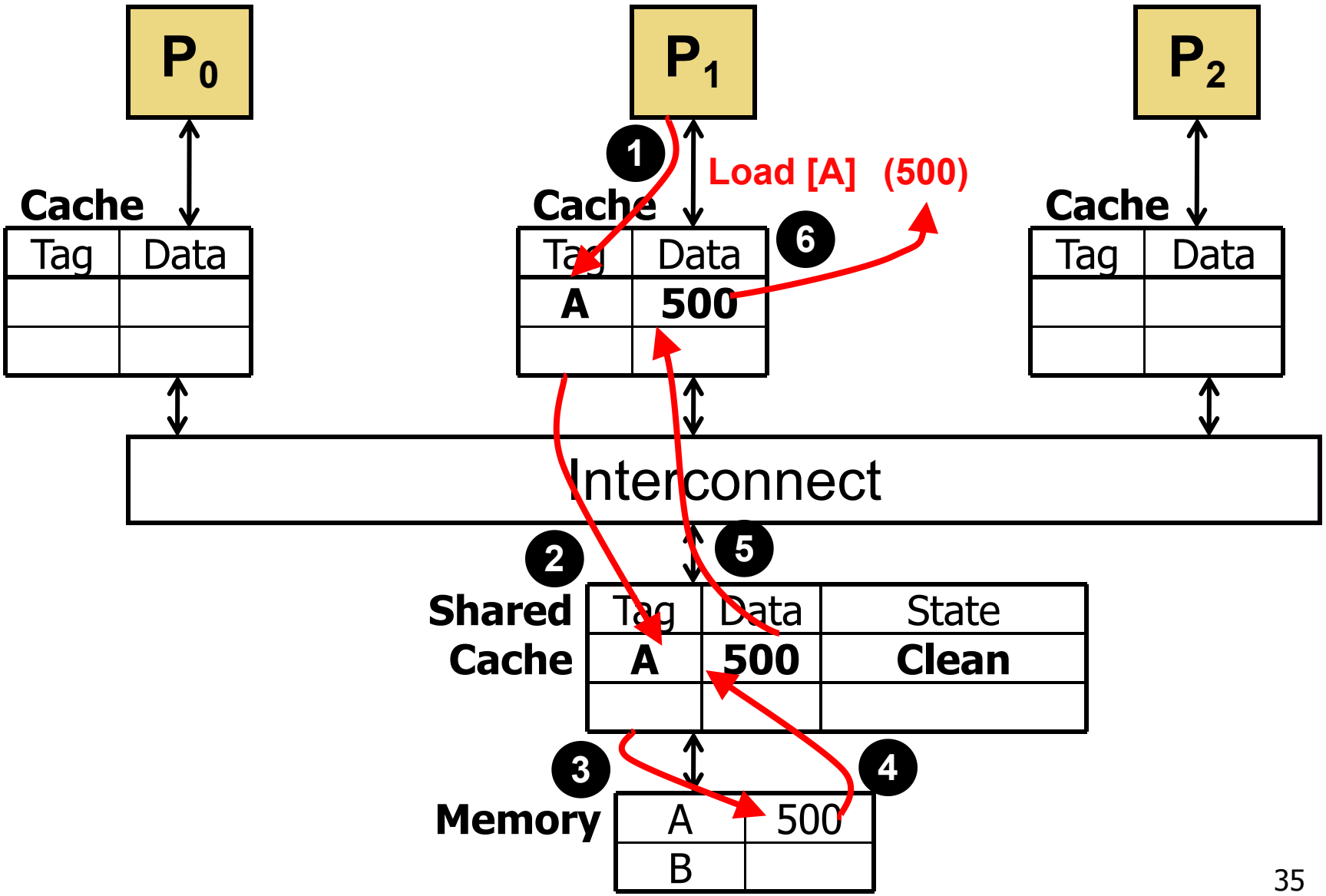


- **Add per-core caches** (write-back caches)
  - Reduces latency
  - Increases throughput
  - Decreases energy

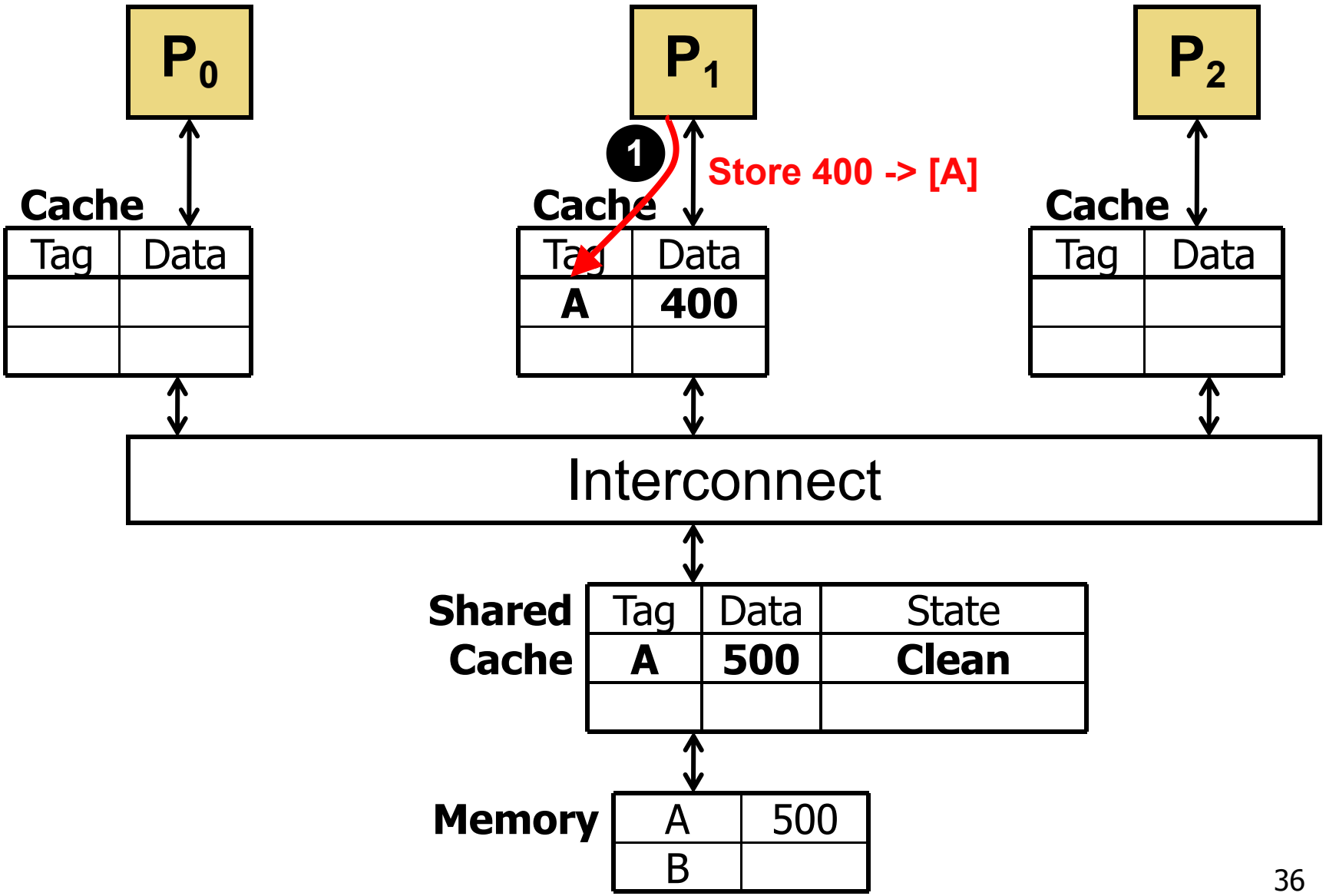
Adding Private Caches



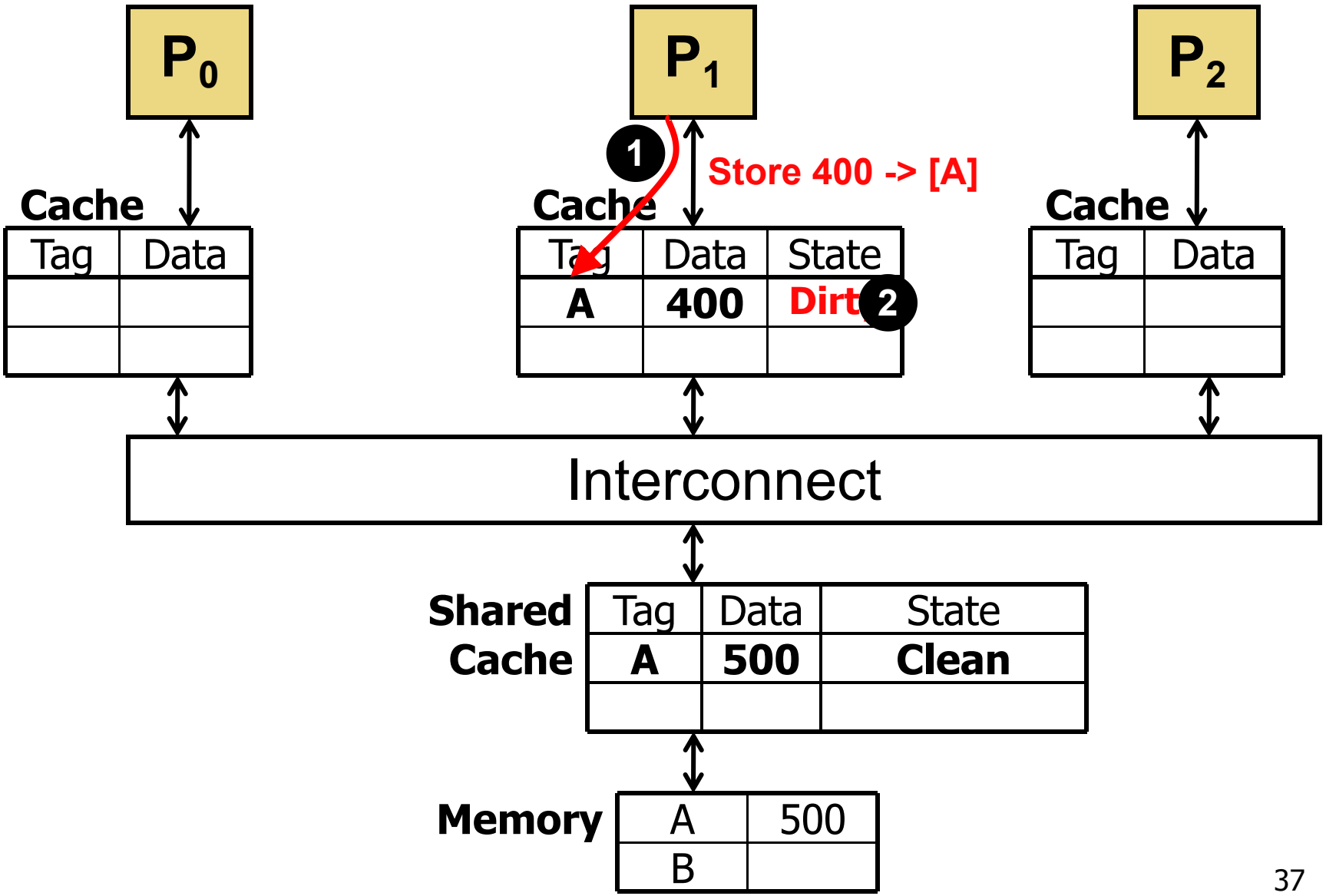
Adding Private Caches



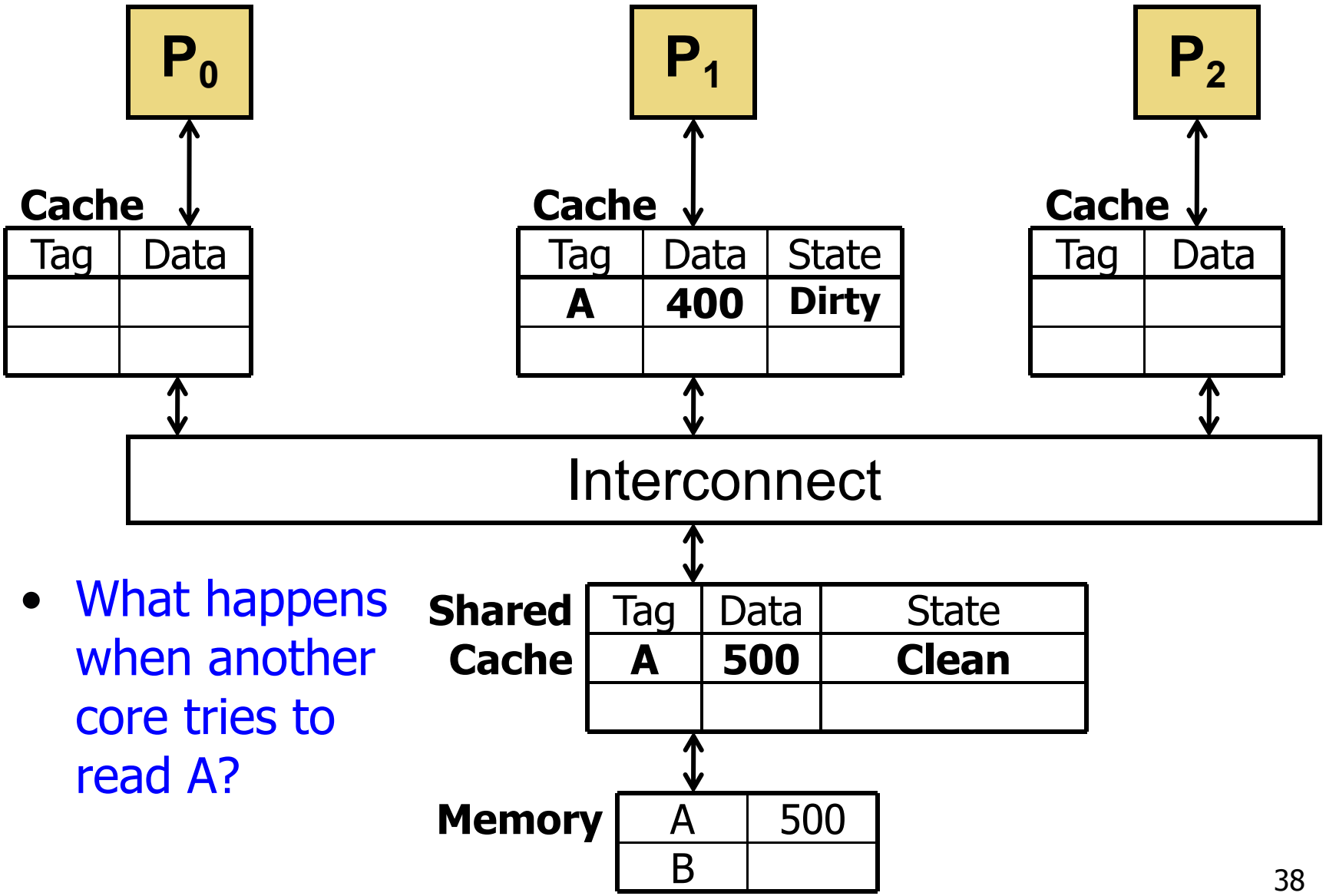
Adding Private Caches



Adding Private Caches

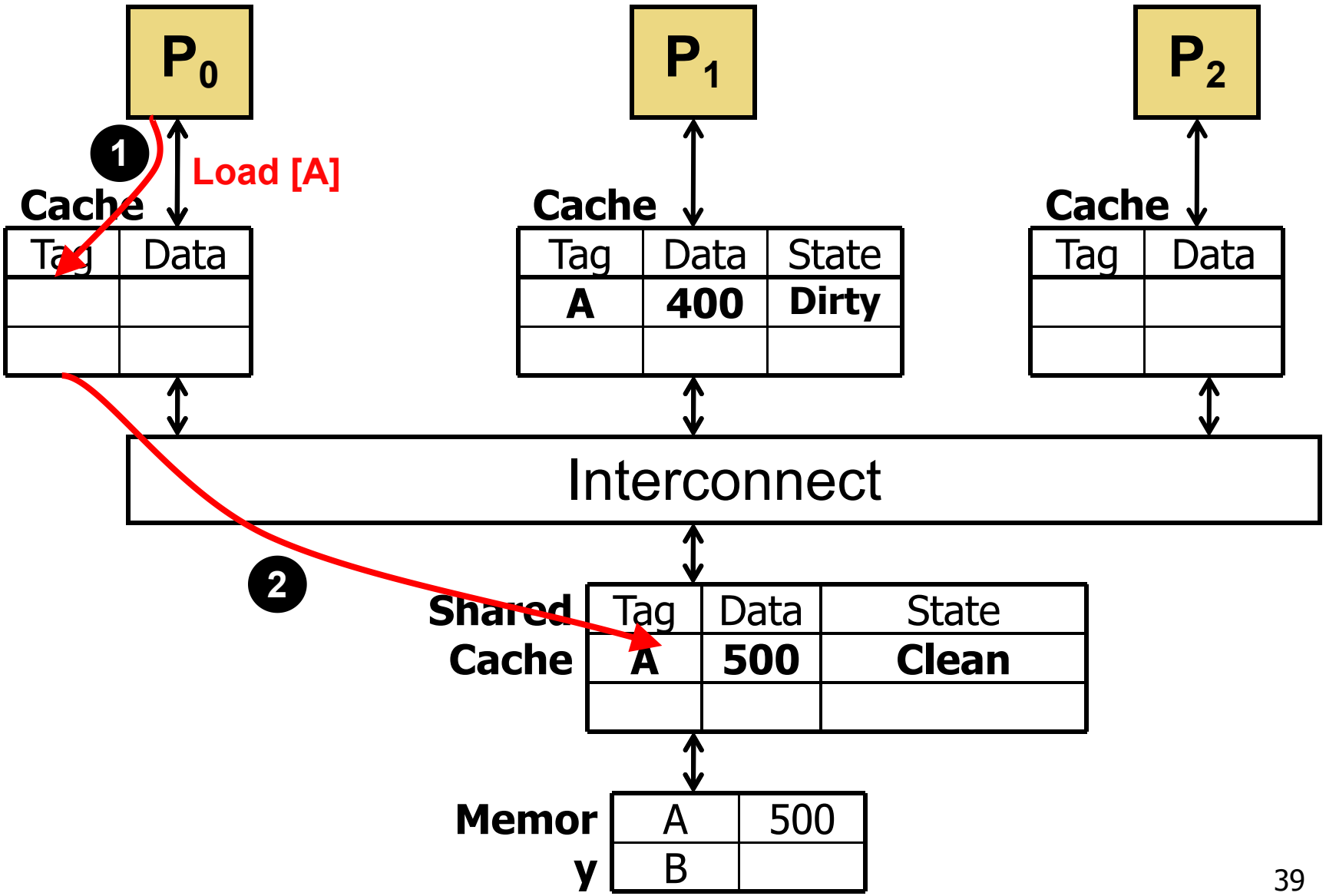


Private Cache Problem: Incoherence

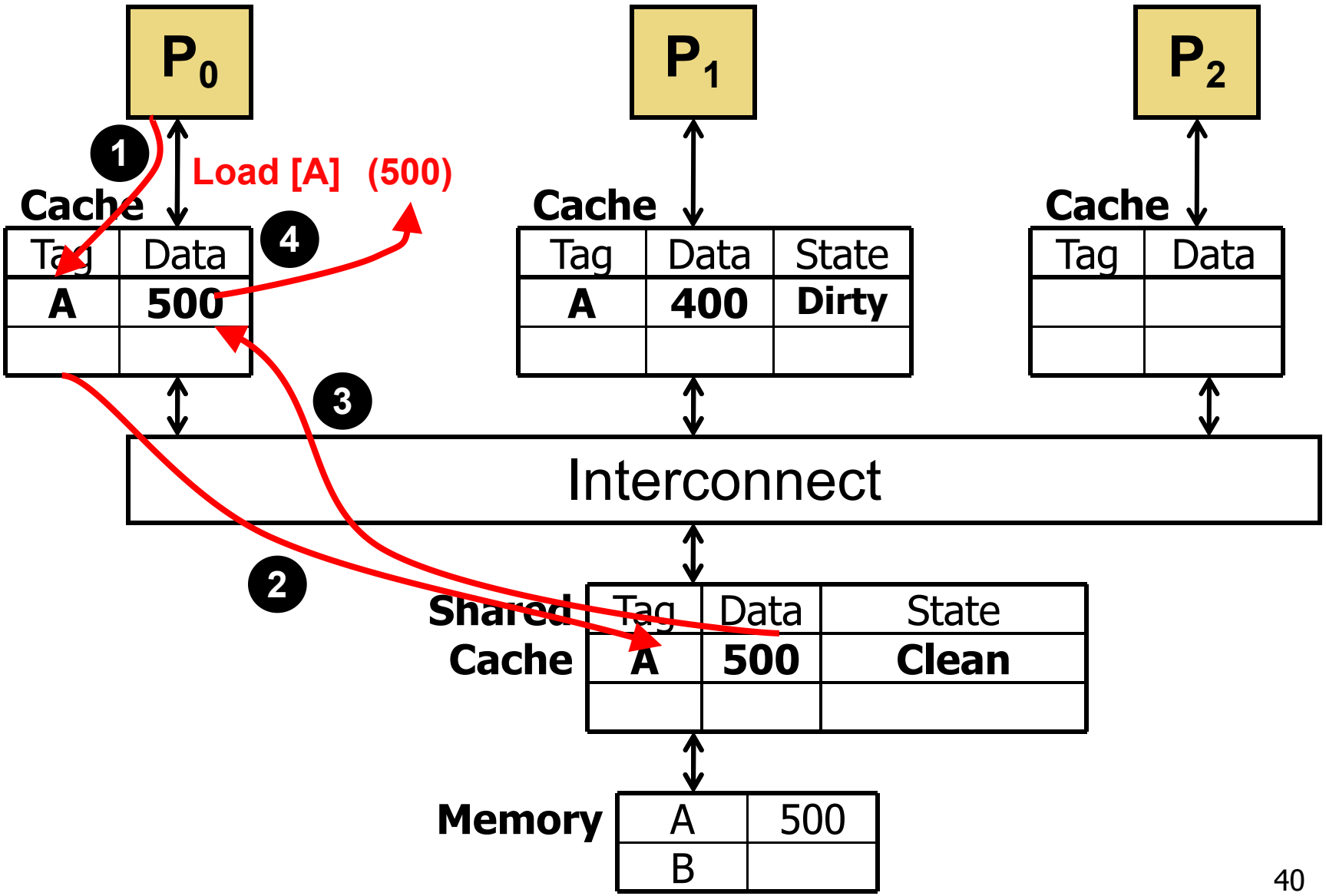


- What happens when another core tries to read A?

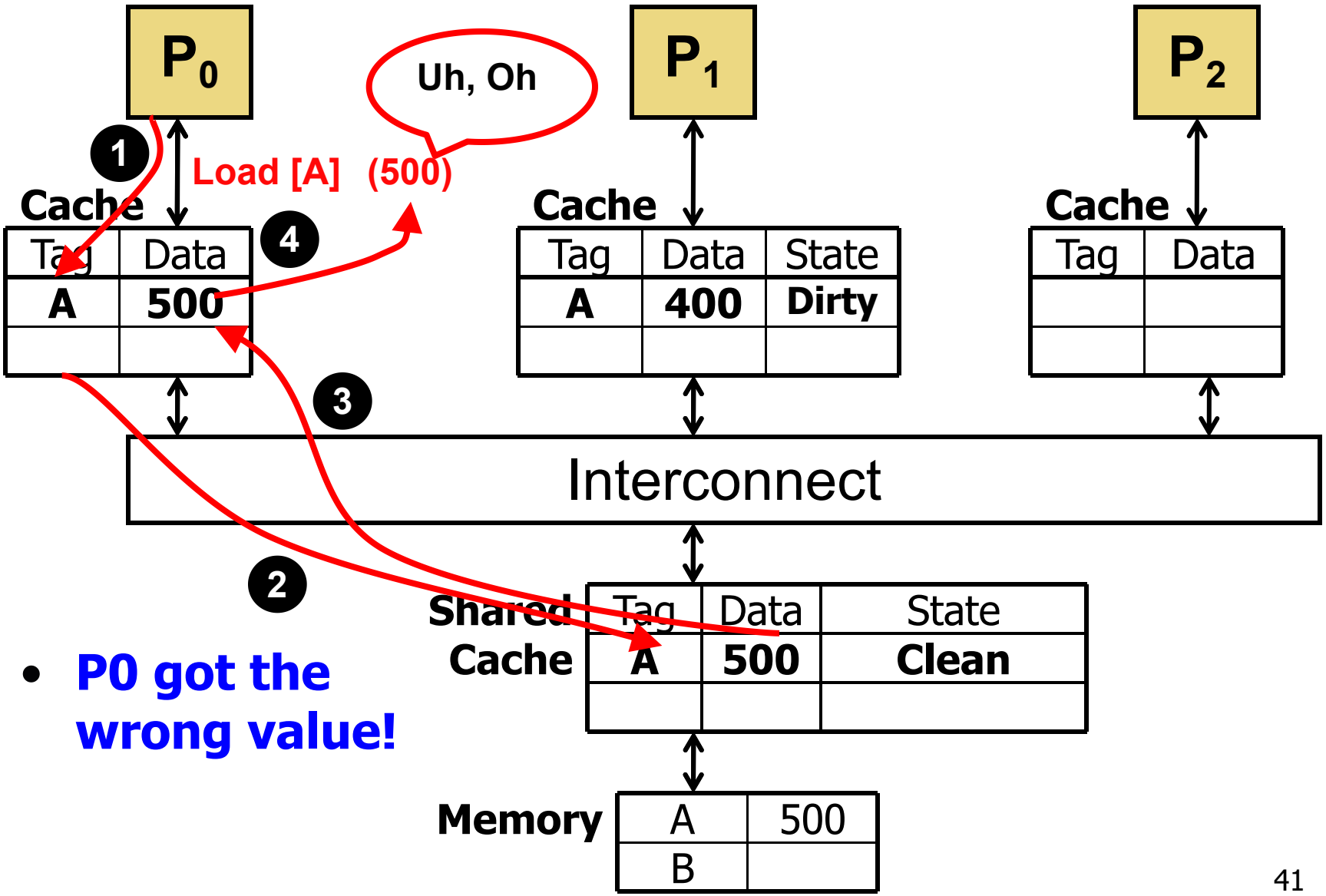
Private Cache Problem: Incoherence



Private Cache Problem: Incoherence

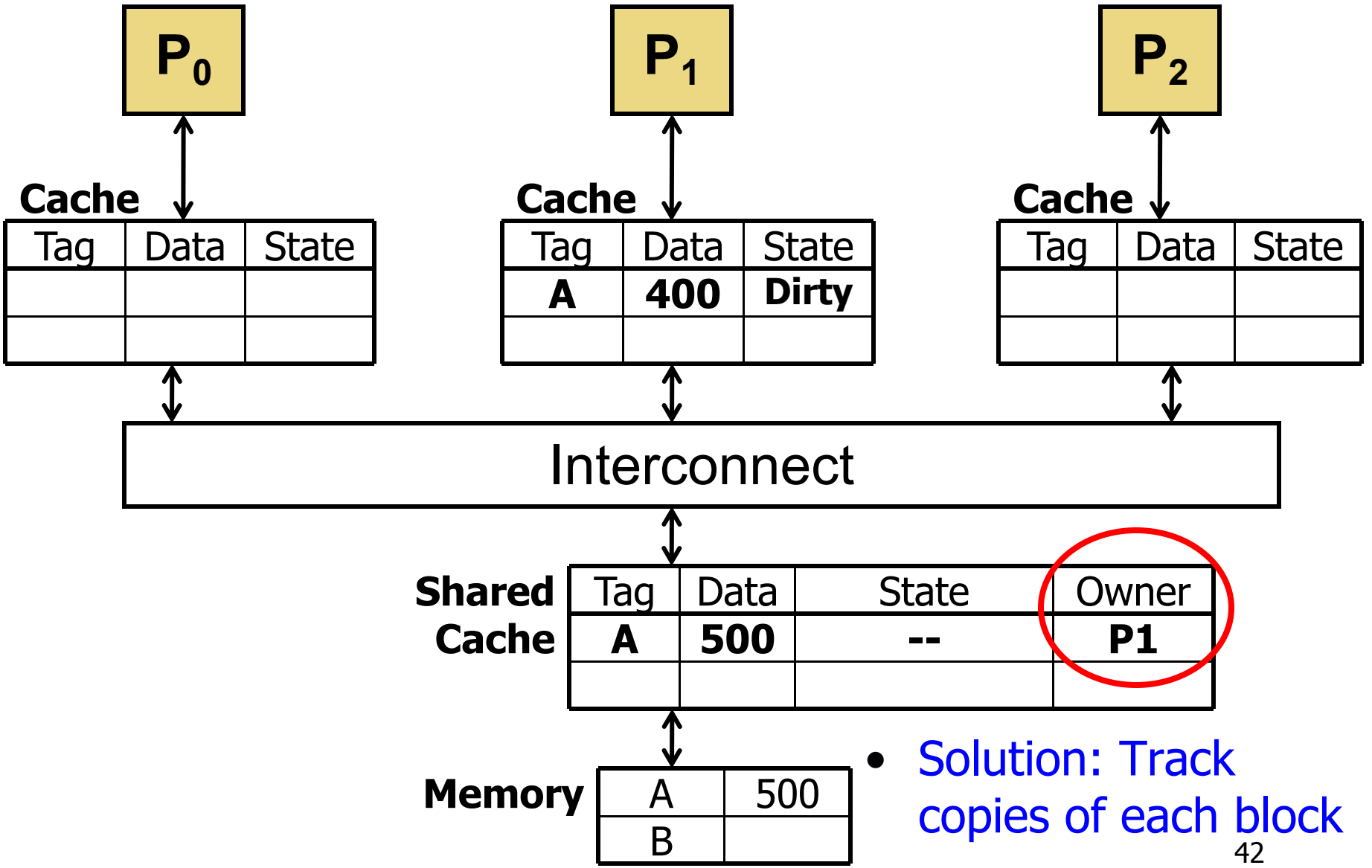


Private Cache Problem: Incoherence



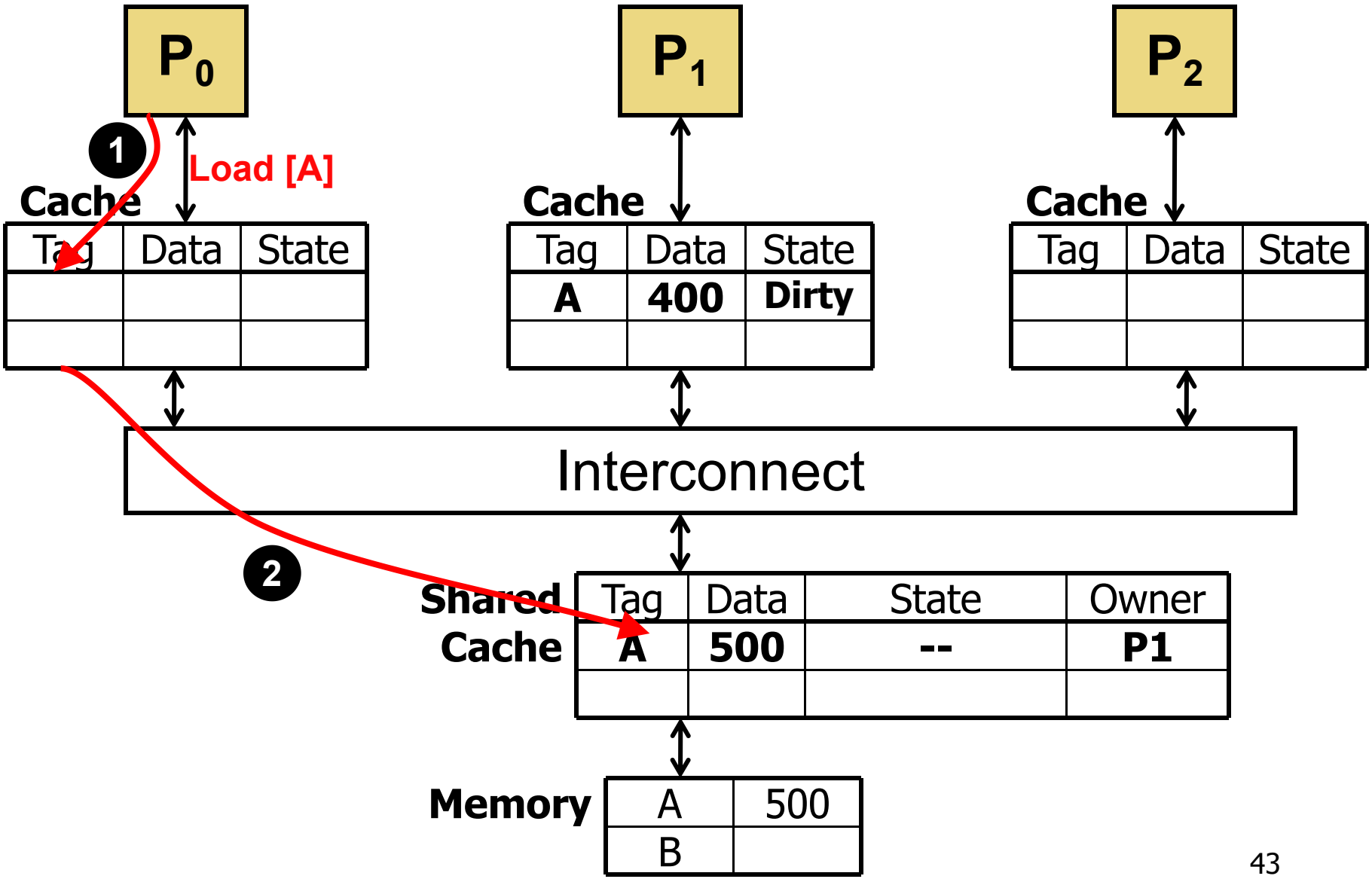
- **P0 got the wrong value!**

Rewind: Fix Problem by Tracking Sharers

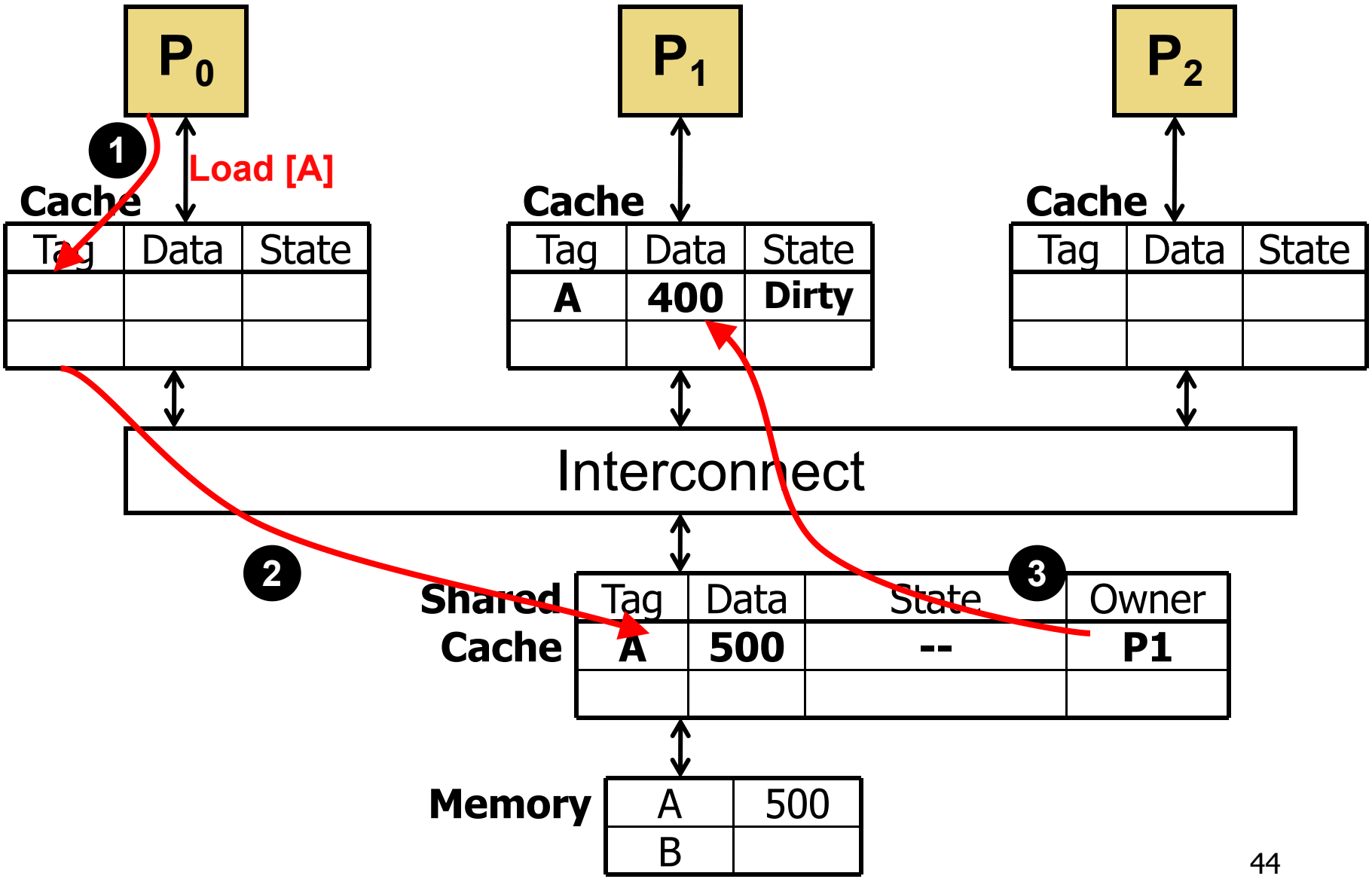


- Solution: Track copies of each block

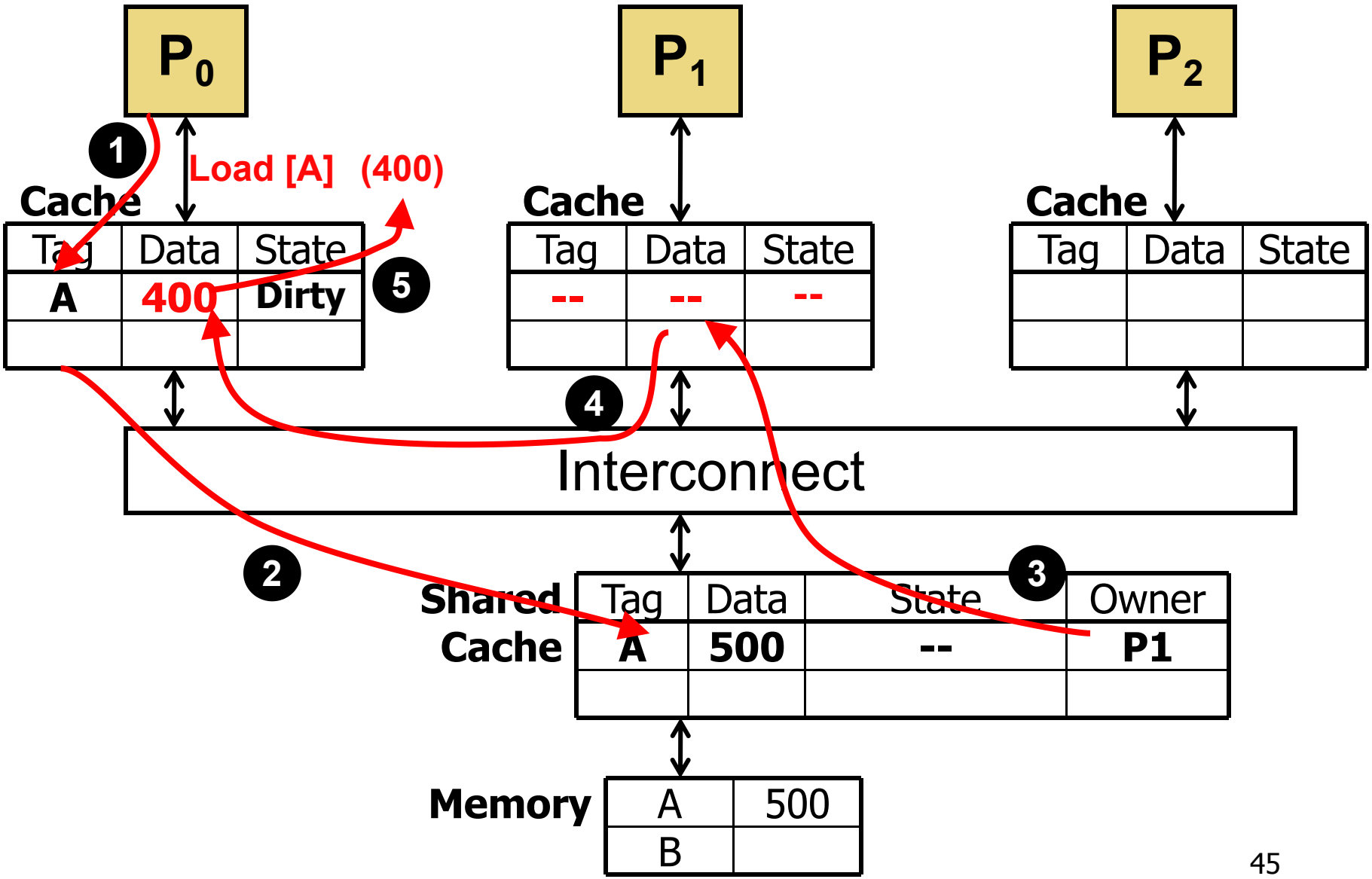
Use Tracking Information to "Invalidate"



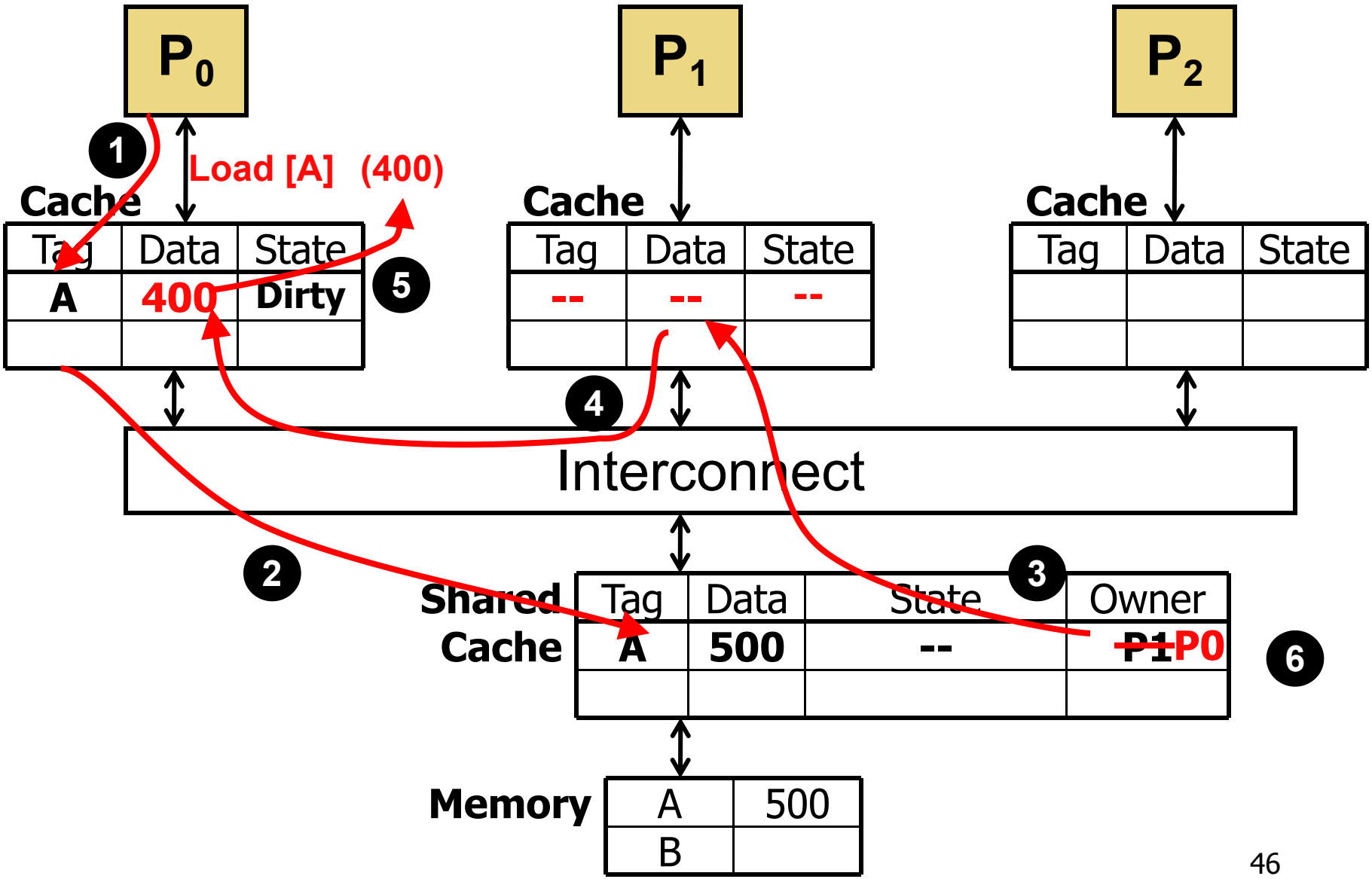
Use Tracking Information to "Invalidate"



Use Tracking Information to "Invalidate"



Use Tracking Information to "Invalidate"



"Valid/Invalid" Cache Coherence

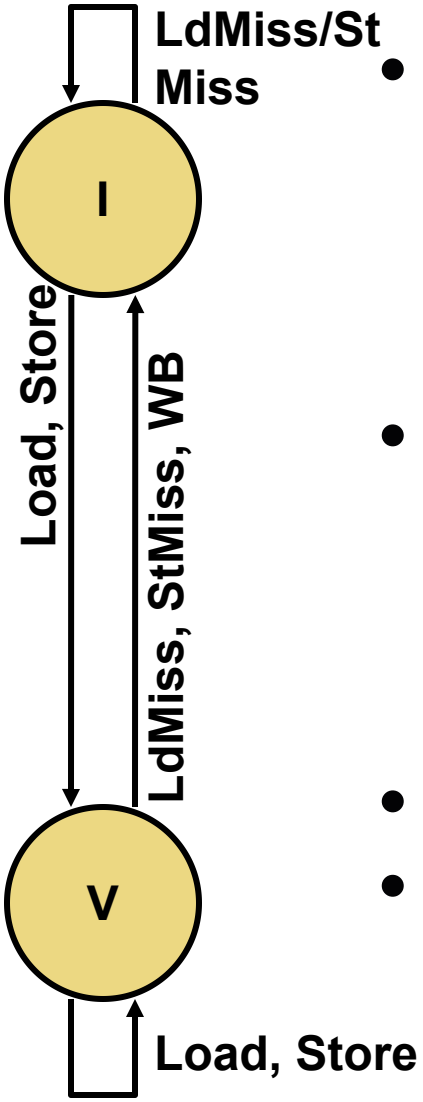
- To enforce the shared memory invariant…
  - "Loads read the value written by the most recent store"
- Enforce the invariant…
  - **"At most one valid copy of the block"**
  - Simplest form is a **two-state "valid/invalid" protocol**
  - If a core wants a copy, must find and "invalidate" it
- On a cache 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
- **Problem**: multiple copies can't exist, even if read-only
  - Consider mostly-read data structures, instructions, etc.

VI Protocol State Transition Table

State	*This Processor*		*Other Processor*	
	Load	Store	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 (Modified I) Coherence Protocol



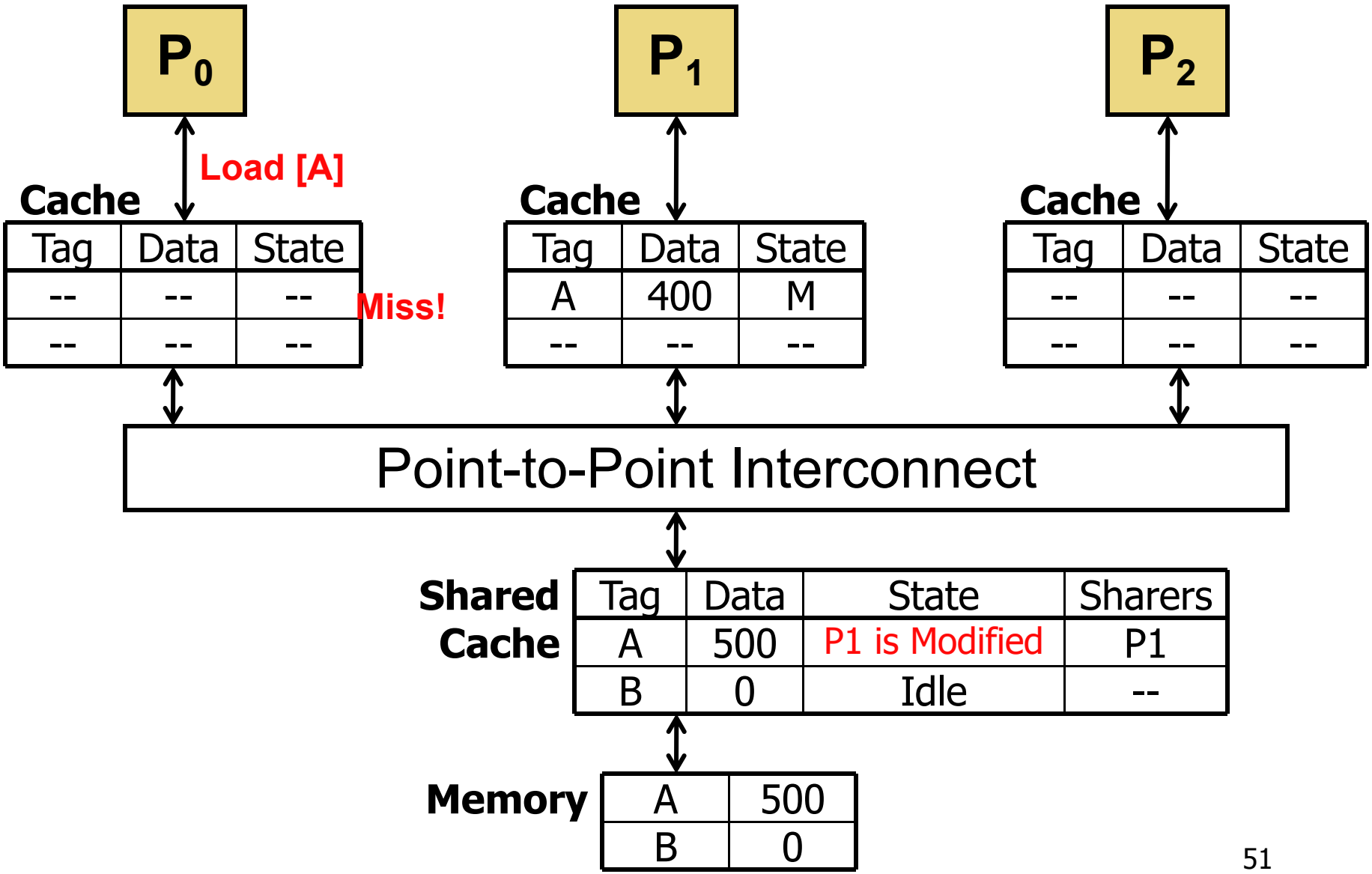
- **VI (valid-invalid) protocol**: aka "MI"
  - Two states (per block in cache)
    - **V (valid)**: have block
    - **I (invalid)**: don't have block

    + Can implement with valid bit
- Protocol diagram (left & next slide)
  - 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**
- **Update protocol**: copy data, don't invalidate
  - Sounds good, but uses too much bandwidth

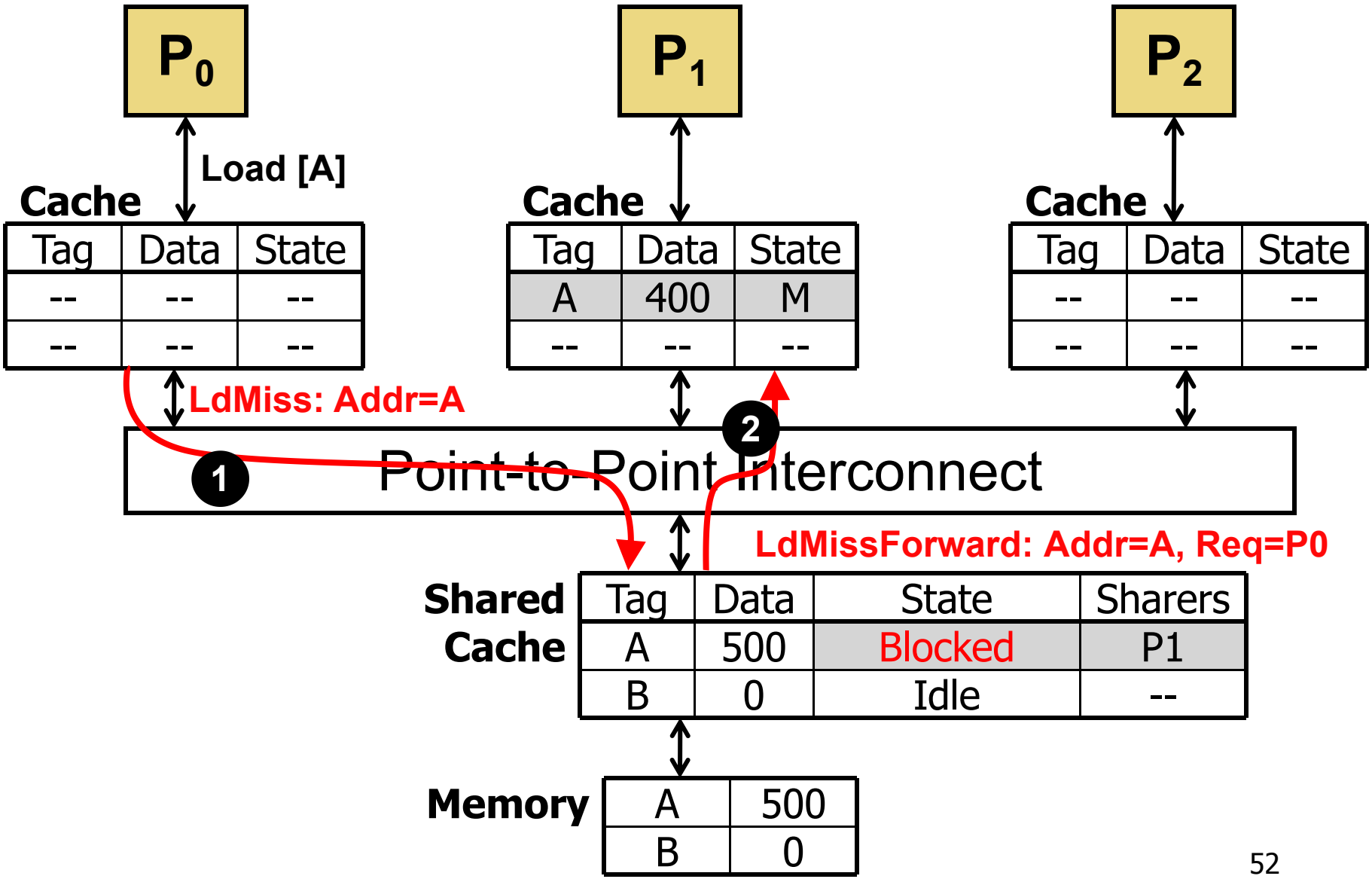
MSI Cache Coherence Protocol

- 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")

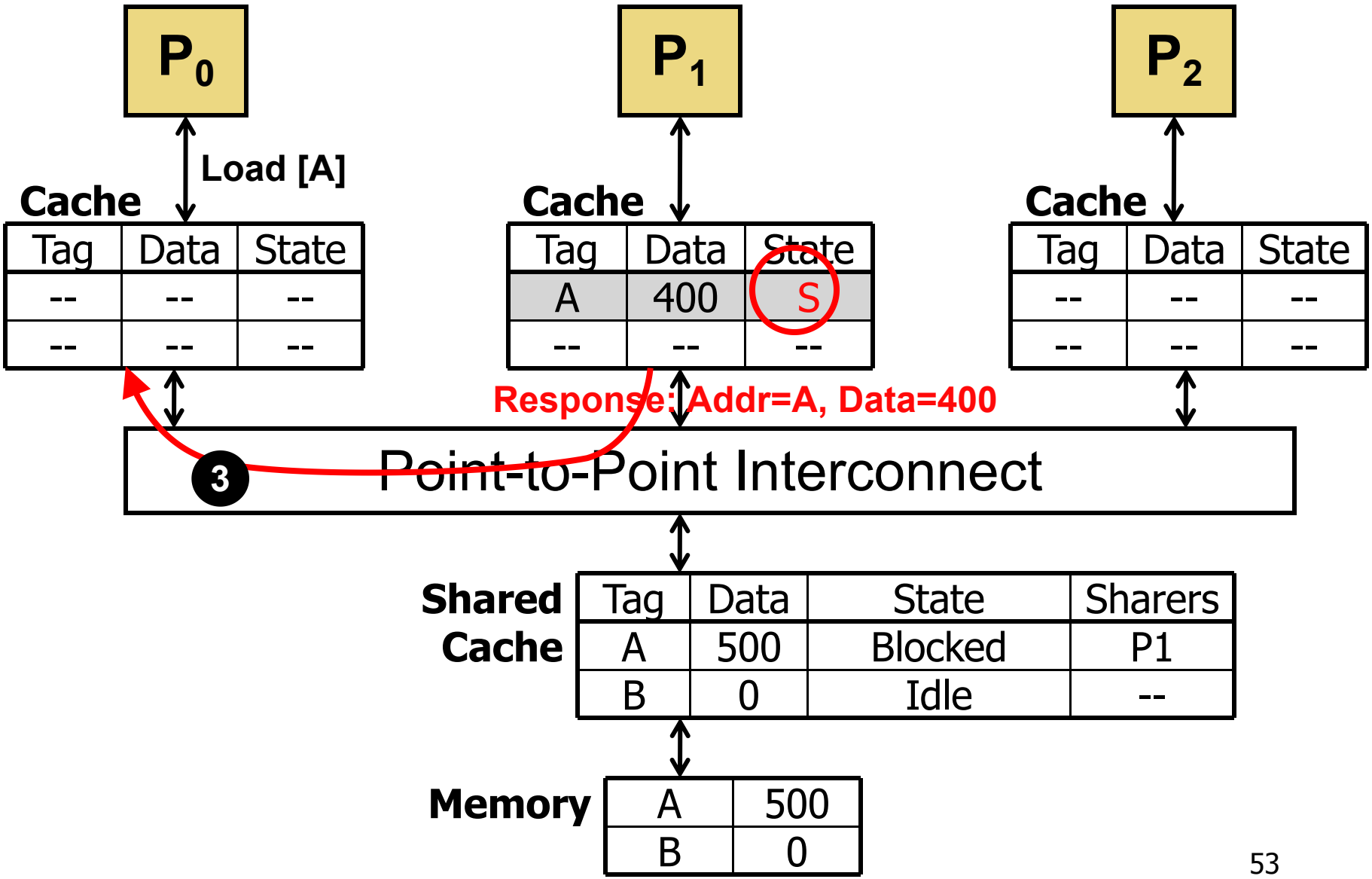
MSI Coherence Example: Step #1



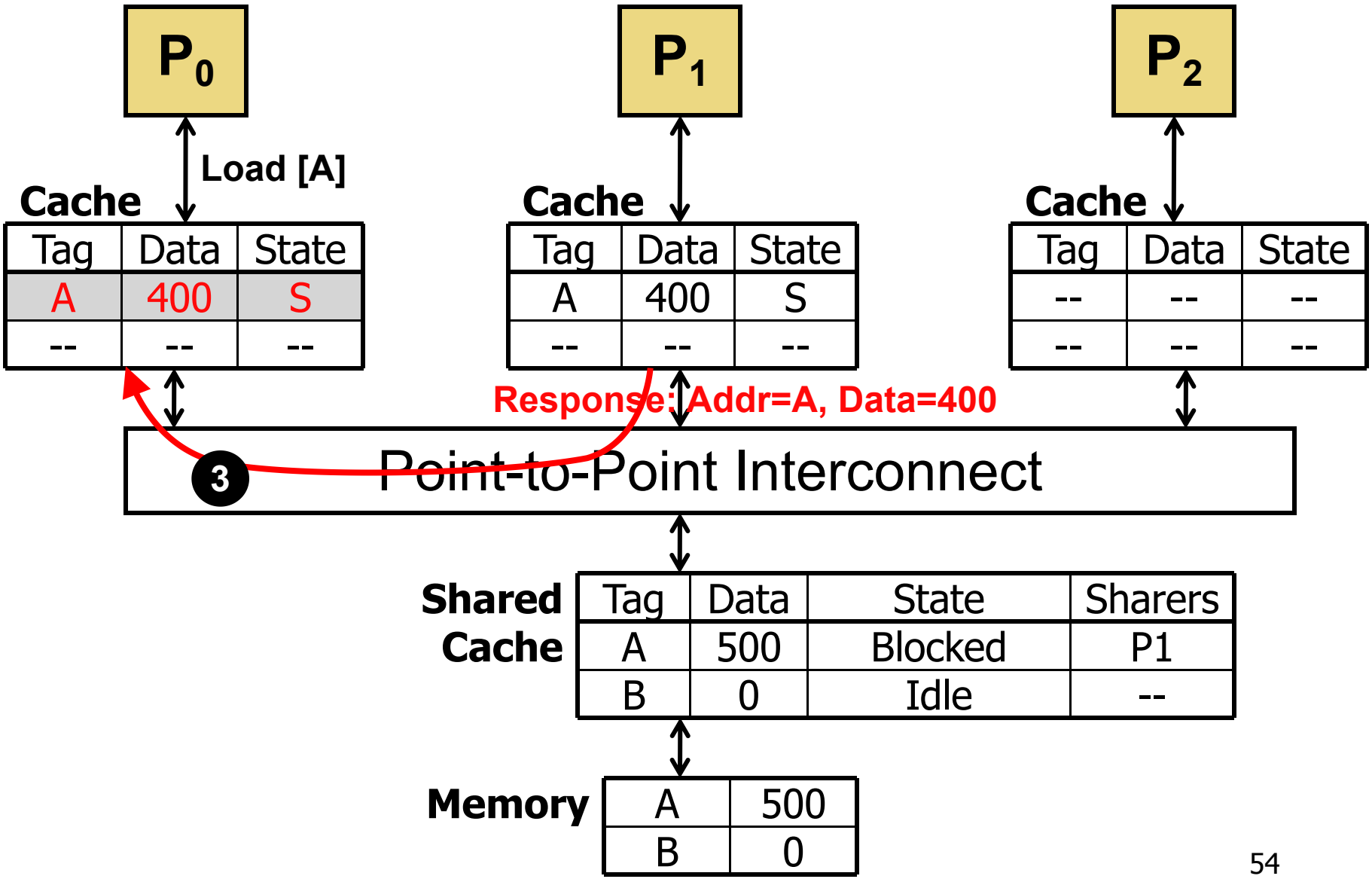
MSI Coherence Example: Step #2



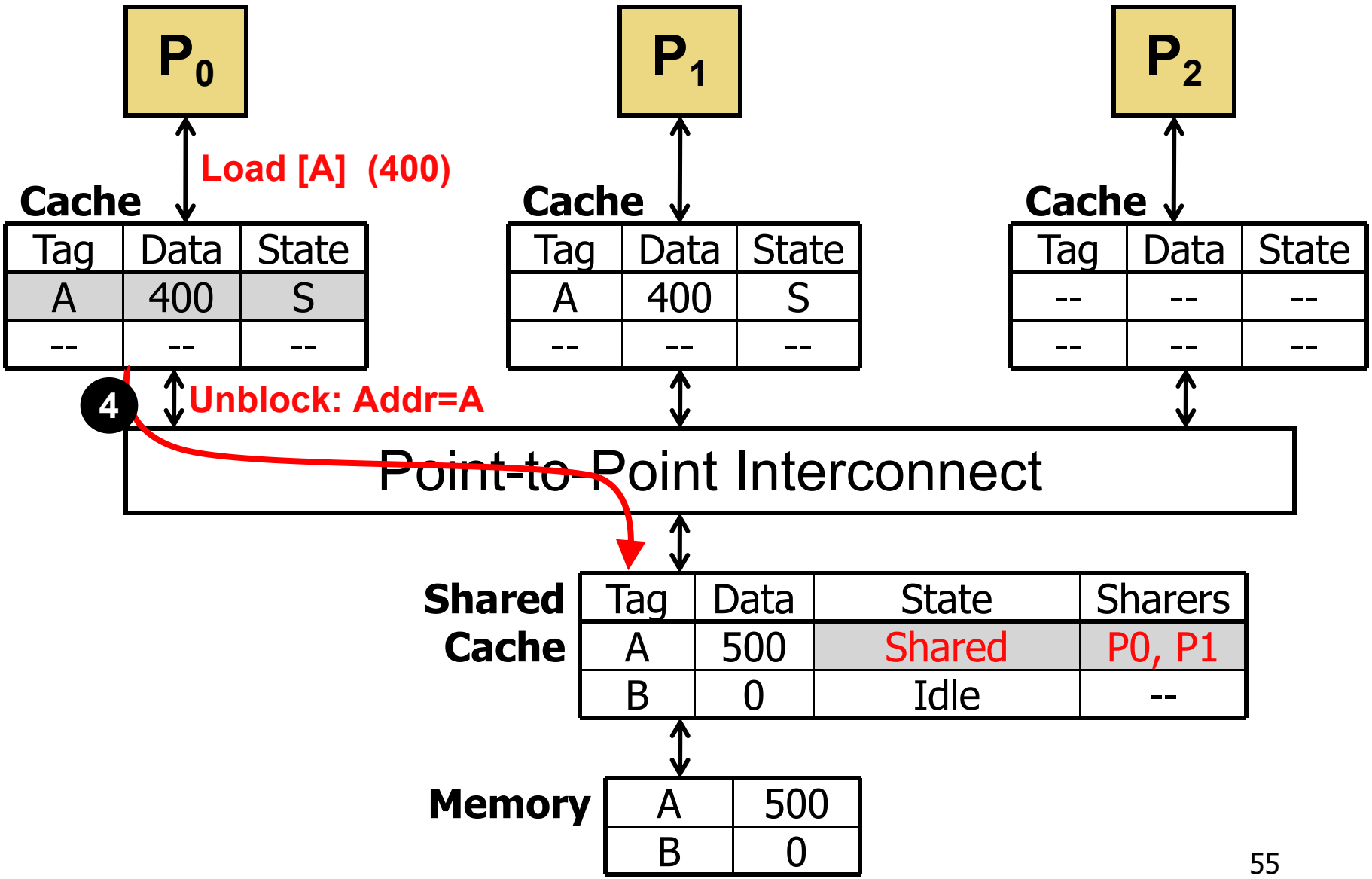
MSI Coherence Example: Step #3



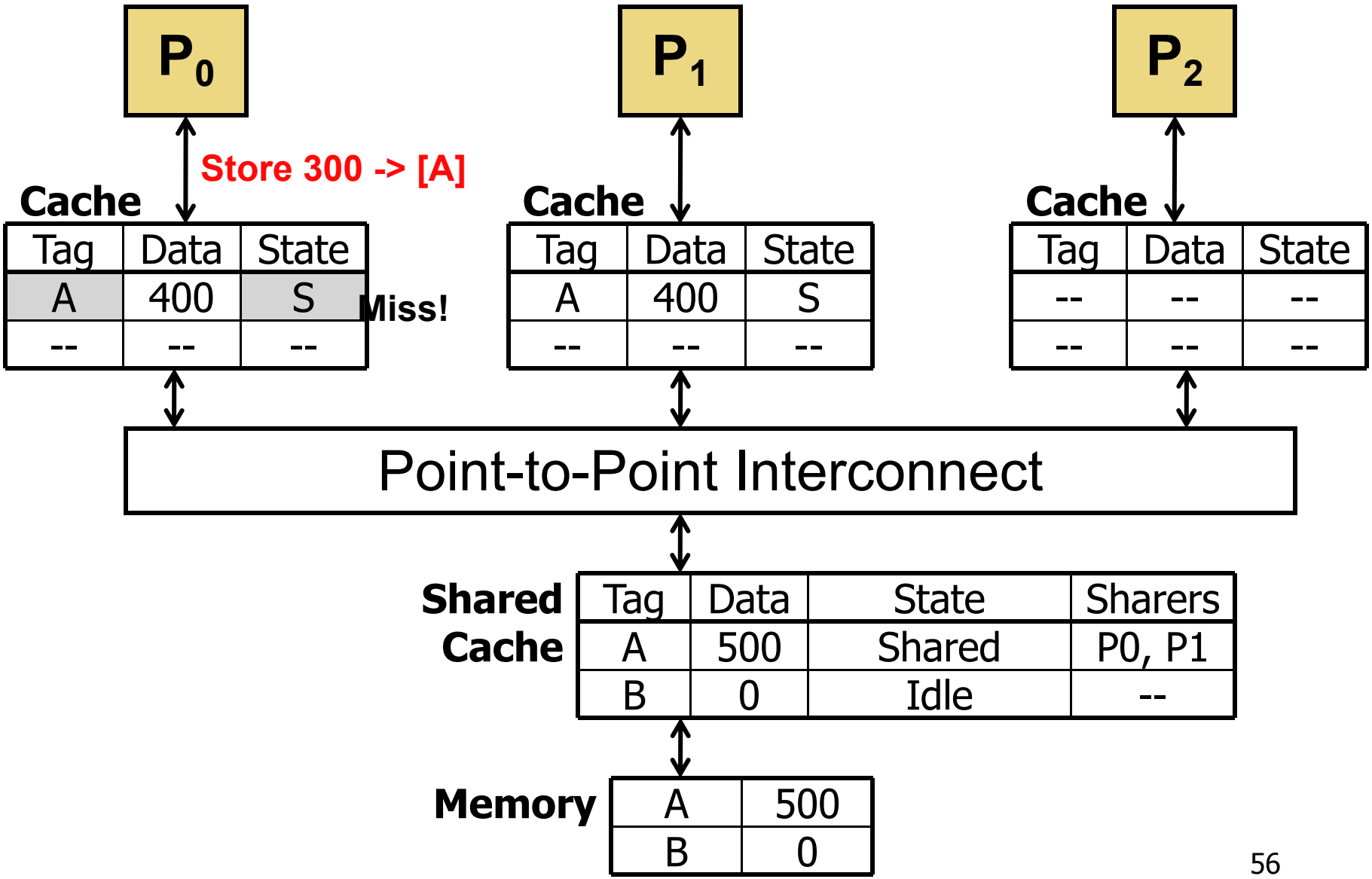
MSI Coherence Example: Step #4



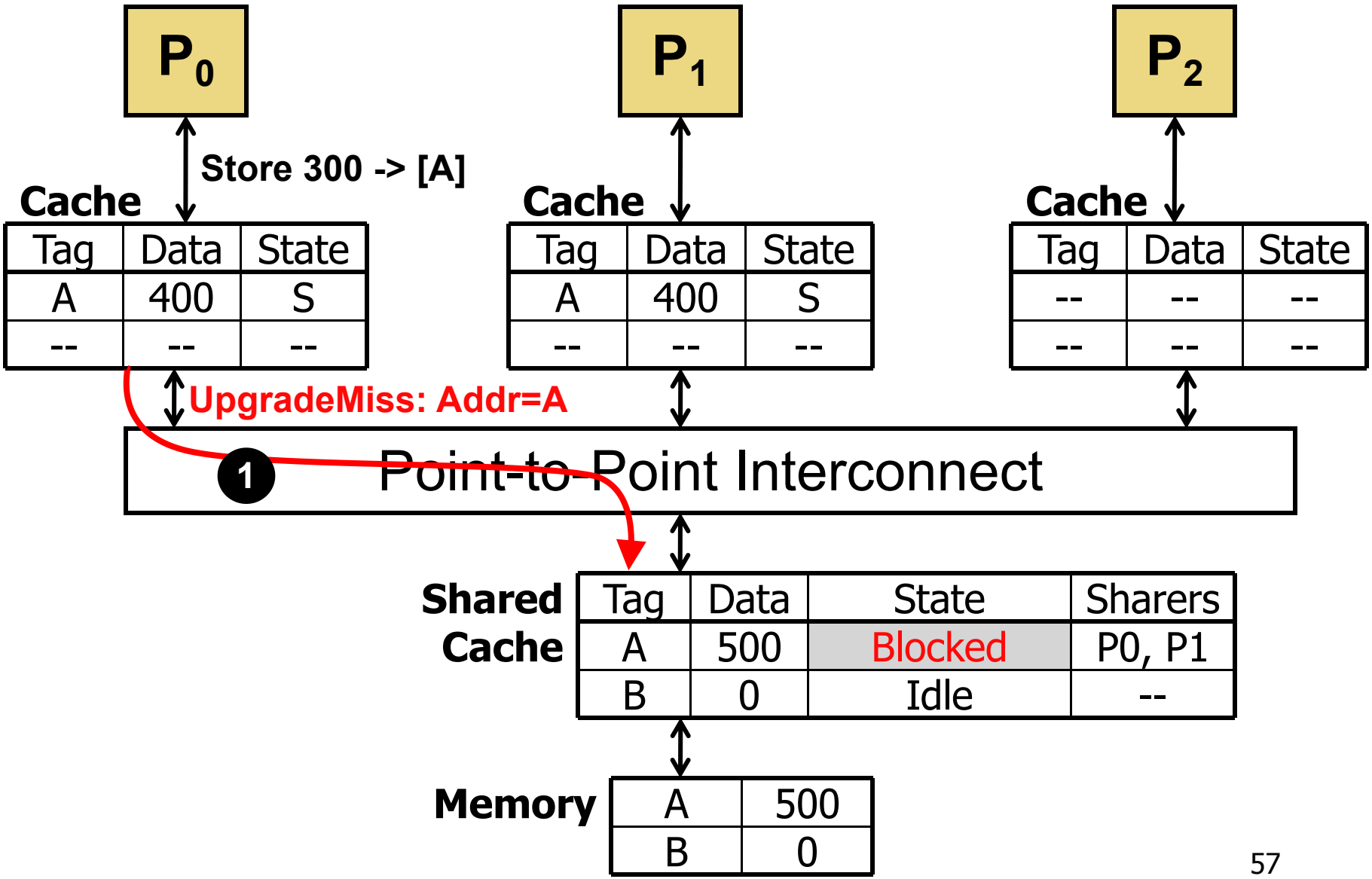
MSI Coherence Example: Step #5



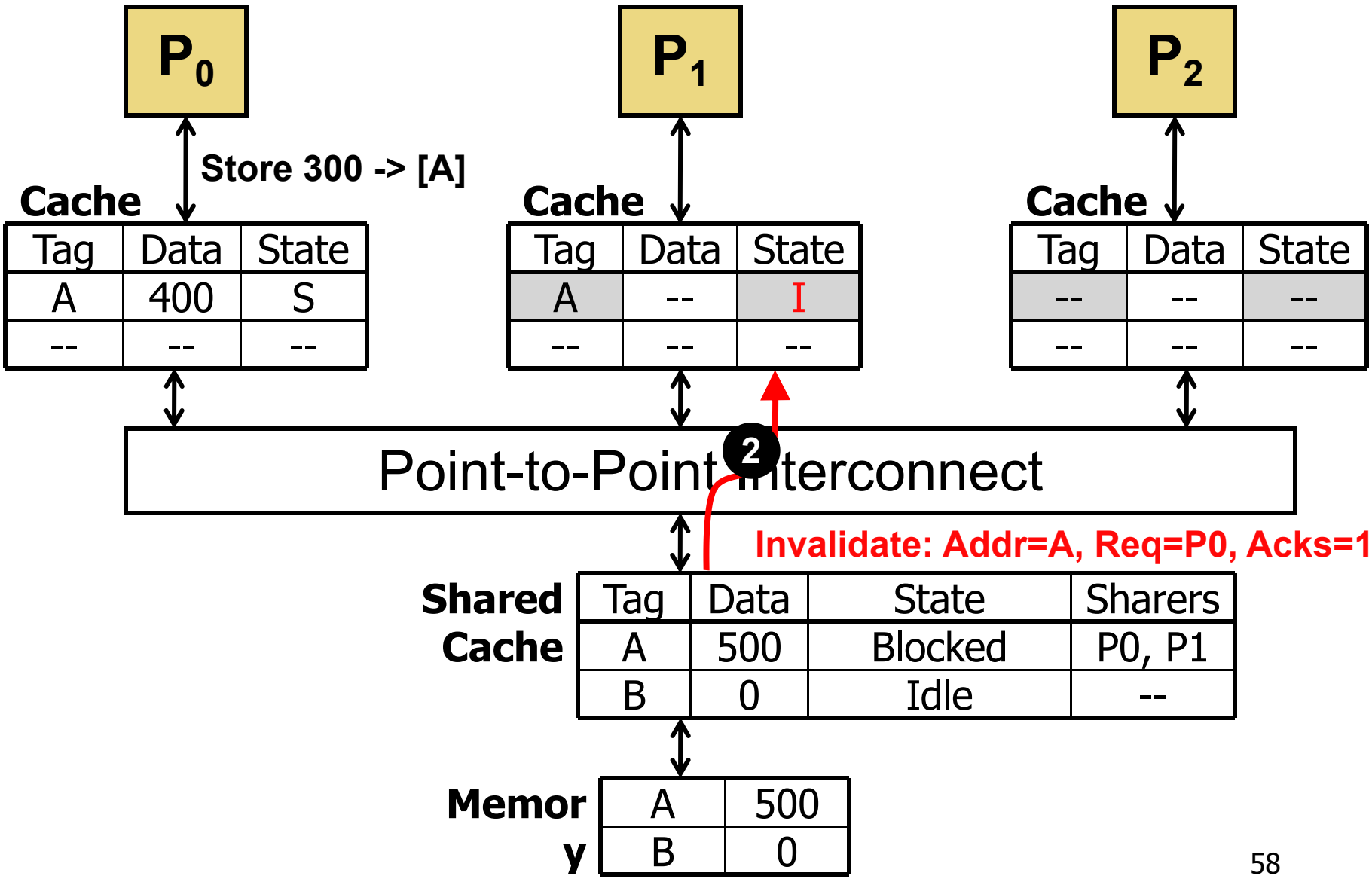
MSI Coherence Example: Step #6



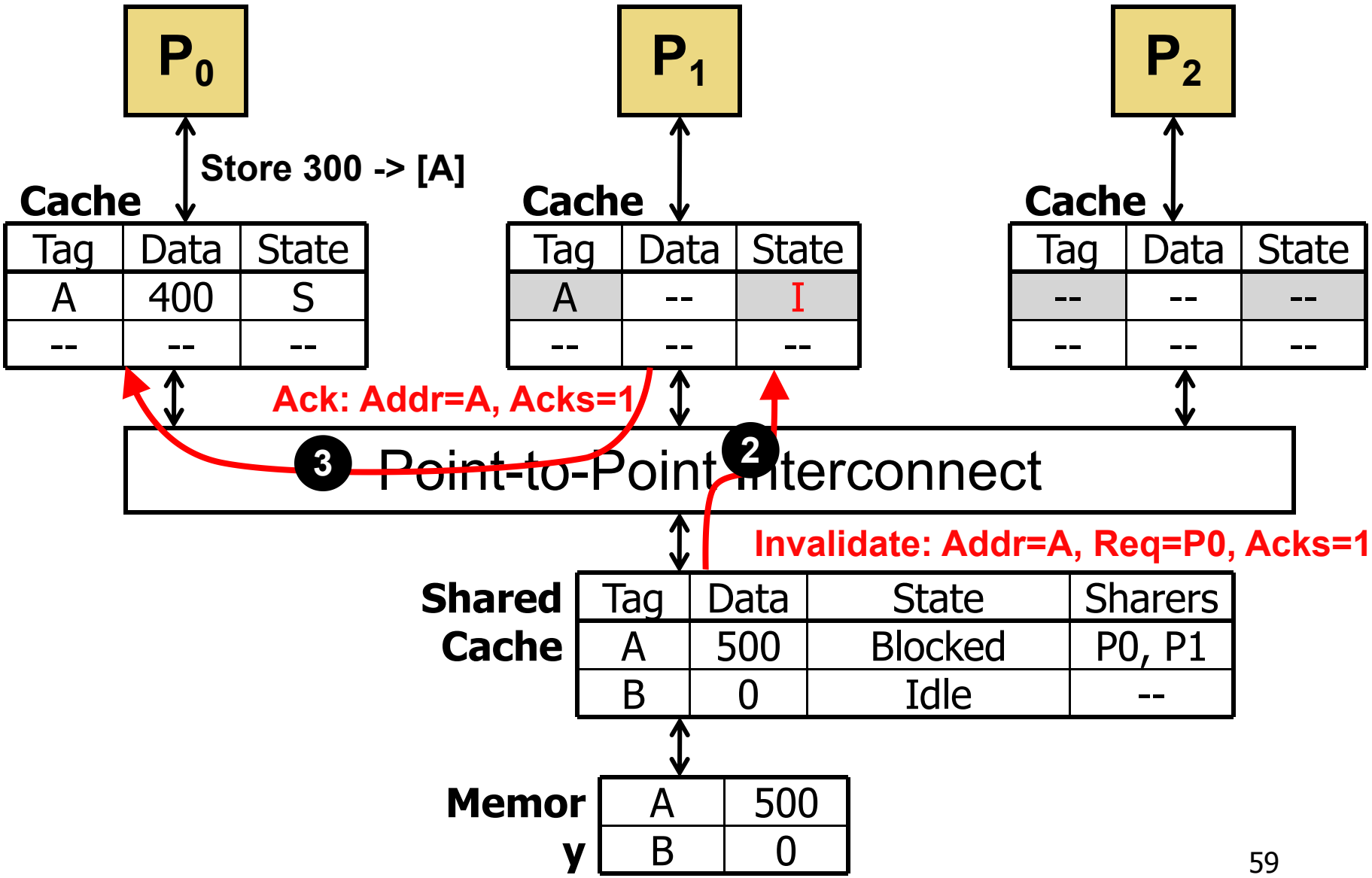
MSI Coherence Example: Step #7



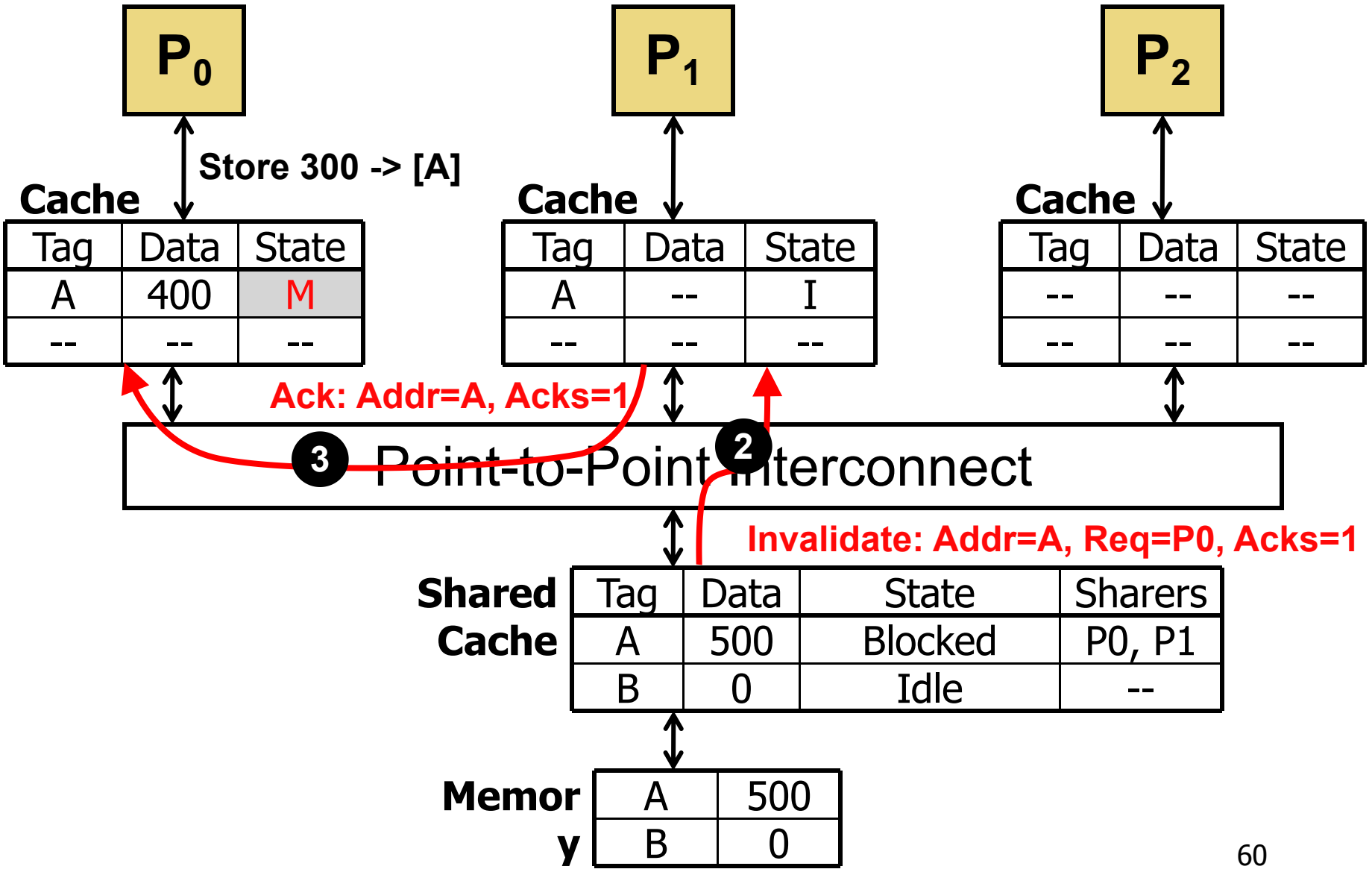
MSI Coherence Example: Step #8



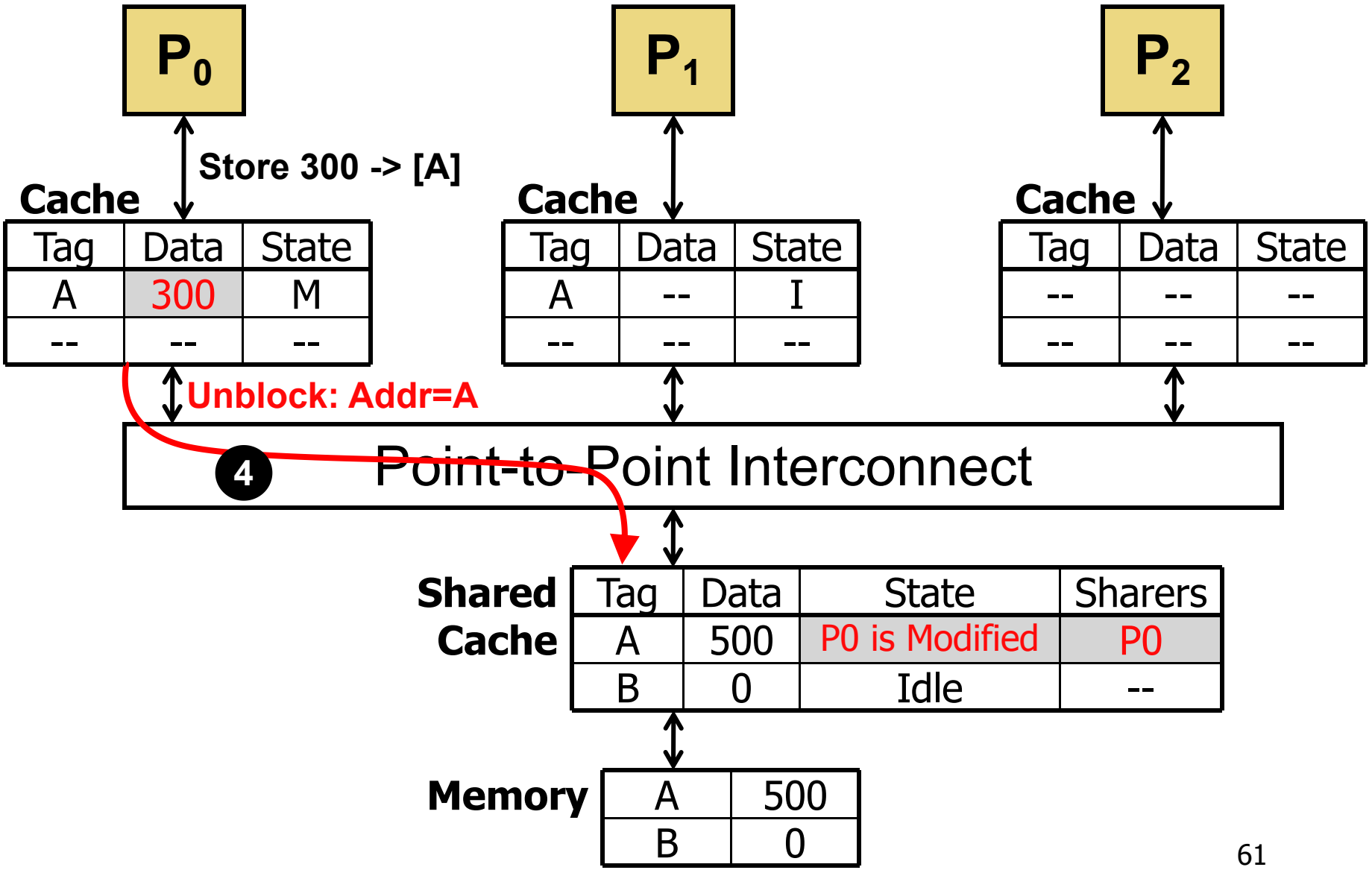
MSI Coherence Example: Step #9



MSI Coherence Example: Step #10



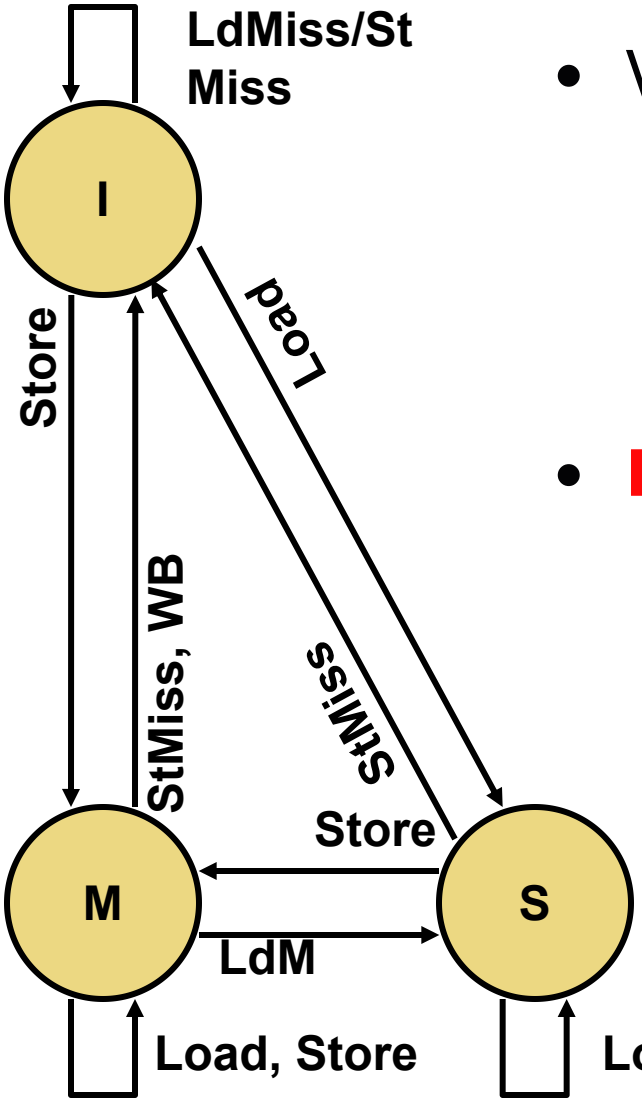
MSI Coherence Example: Step #11



MSI Protocol State Transition Table

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

VI → MSI



- 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)

Cache Coherence and Cache Misses

- Coherence introduces two new kinds of cache misses
  - **Upgrade miss:** stores to read-only blocks
    - Delay to acquire write permission to read-only block
  - **Coherence miss**
    - Miss to a block evicted by another processor's requests
- Making the cache larger…
  - Doesn't reduce these types of misses
  - So, as cache grows large, these sorts of misses dominate
- **False sharing**
  - Two or more processors sharing parts of the same block
  - But *not* the same bytes within that block (no actual sharing)
  - Creates pathological "ping-pong" behavior
  - Careful data placement may help, but is difficult

MESI Cache Coherence

- Ok, we have read-only and read/write with MSI

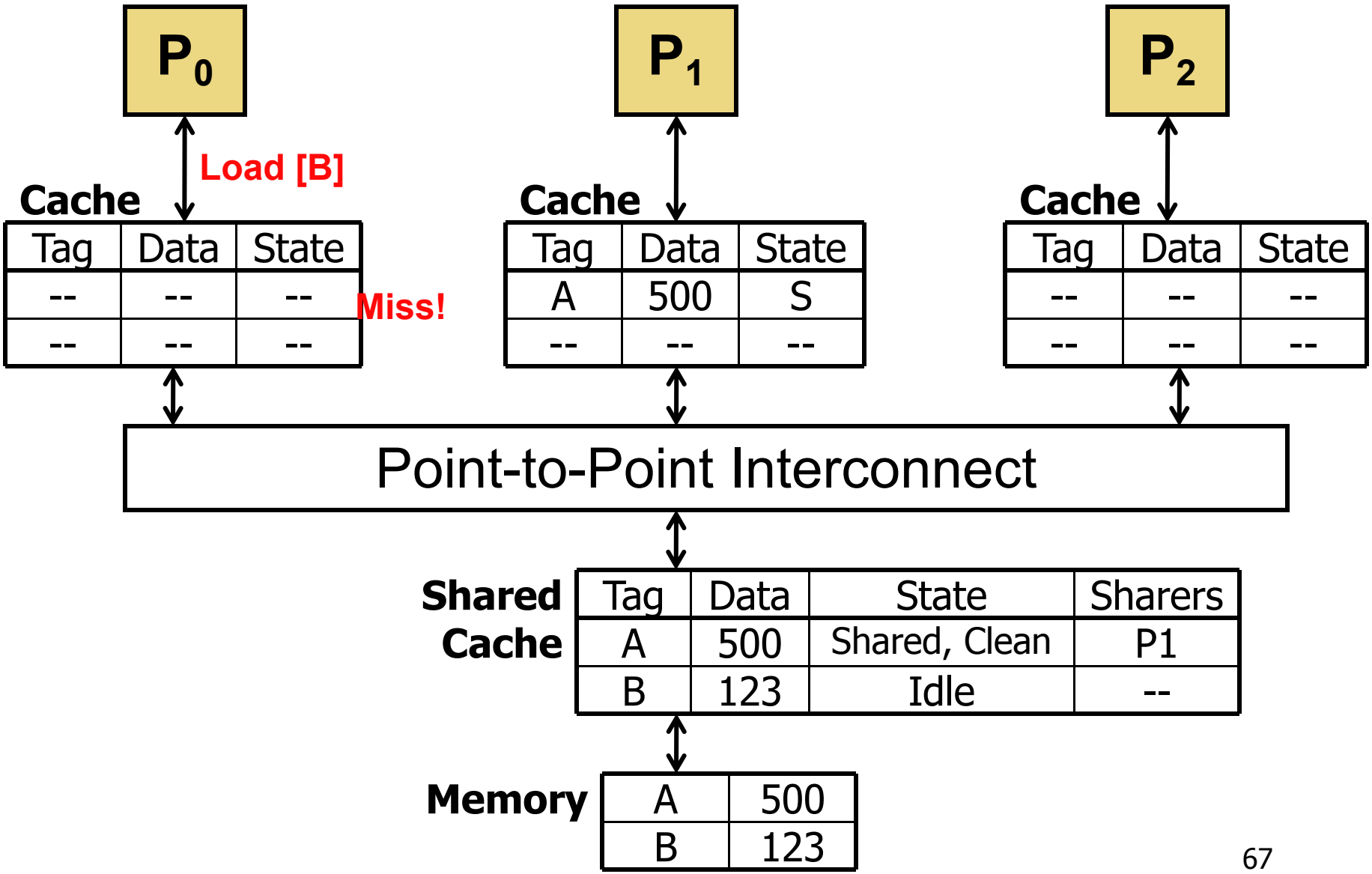
- But consider load & then store of a block by same core
  - Under coherence as described, **this would be two misses: "Load miss" plus an "upgrade miss"…**
  - … even if the block isn't shared!
  - Consider programs with 99% (or 100%) private data
    - Potentially doubling number of misses (bad)

- Solution:
  - Most modern protocols also include **E (exclusive)** state
  - Interpretation: "I have the only cached copy, and it's a **clean** copy"
    - Has read/write permissions
    - Just like "Modified" but "clean" instead of "dirty".

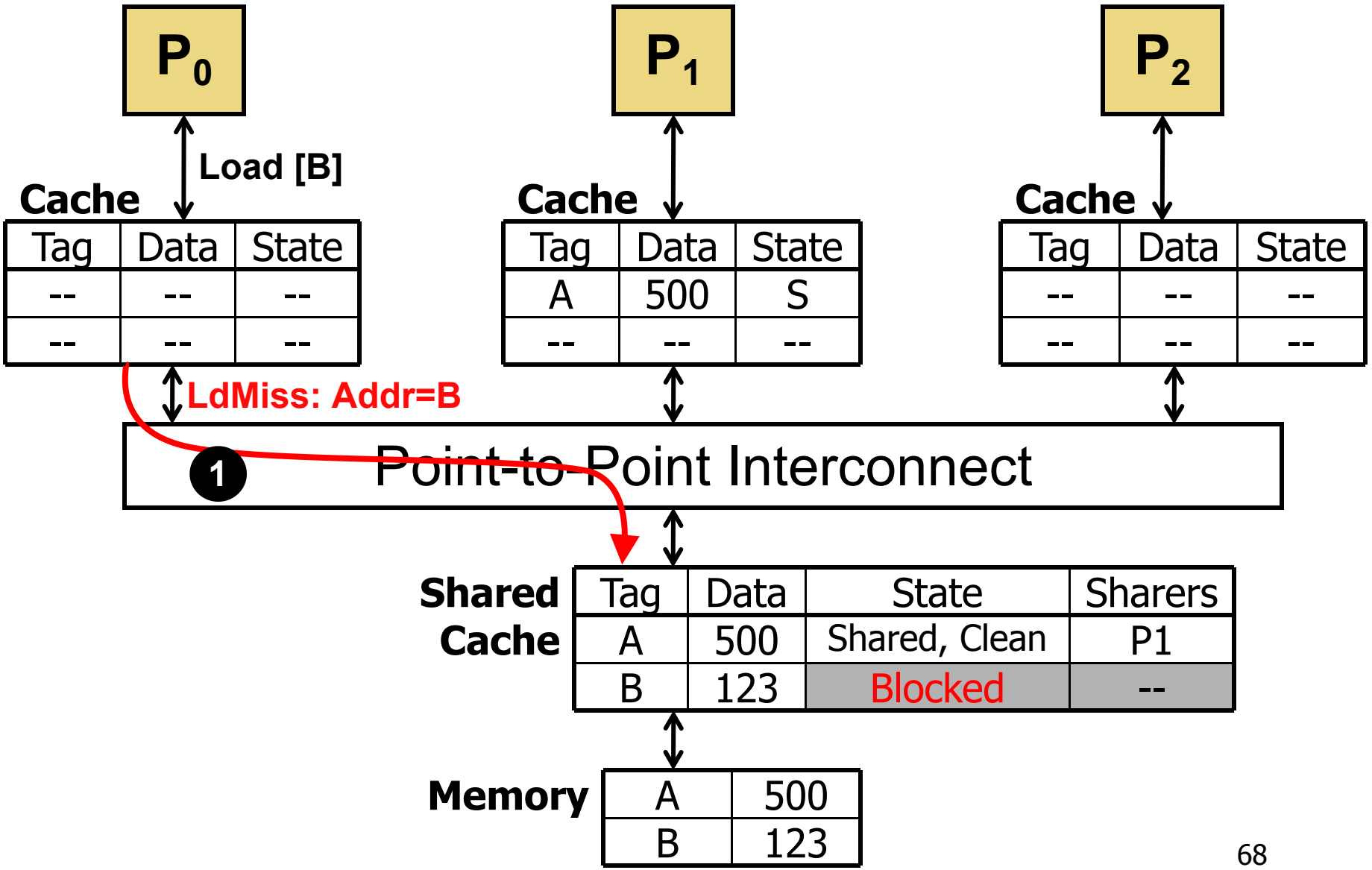
MESI Operation

- Goals:
  - Avoid "upgrade" misses for non-shared blocks
  - While not increasing eviction (aka writeback or replacement) traffic
- Two cases on a load miss to a block…
  - **Case #1**: … with no current sharers
    (that is, no sharers in the set of sharers)
    - Grant requester "Exclusive" copy with read/write permission
  - **Case #2**: … with other sharers
    - As before, grant just a "Shared" copy with read-only permission
- A store to a block in "Exclusive" changes it to "Modified"
  - **Instantaneously & silently** (no latency or traffic)
- On block eviction (aka writeback or replacement)…
  - If "Modified", block is dirty, must be written back to next level
  - If "Exclusive", writing back the data is not necessary
    (but notification may or may not be, depending on the system)

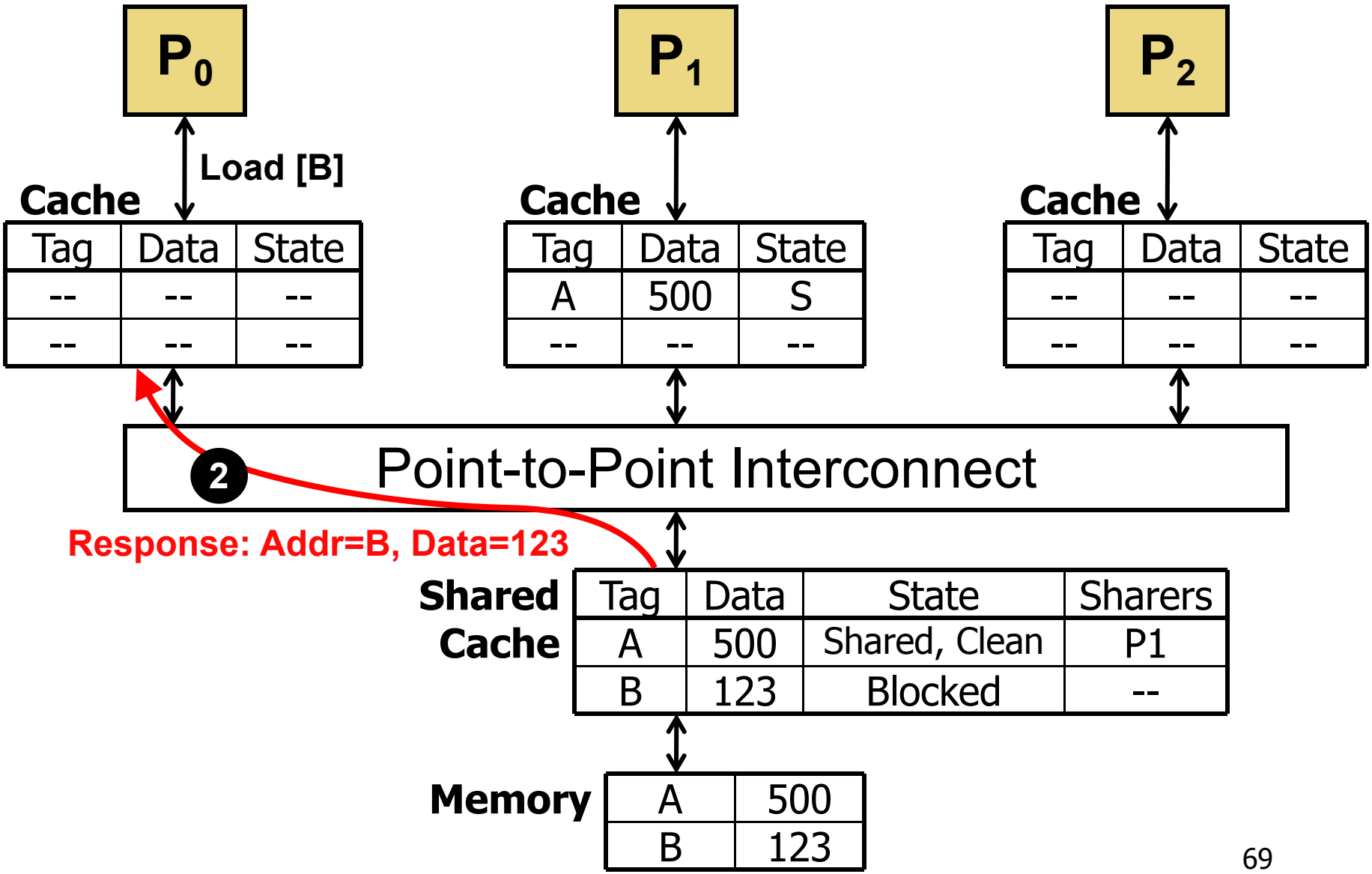
MESI Coherence Example: Step #1



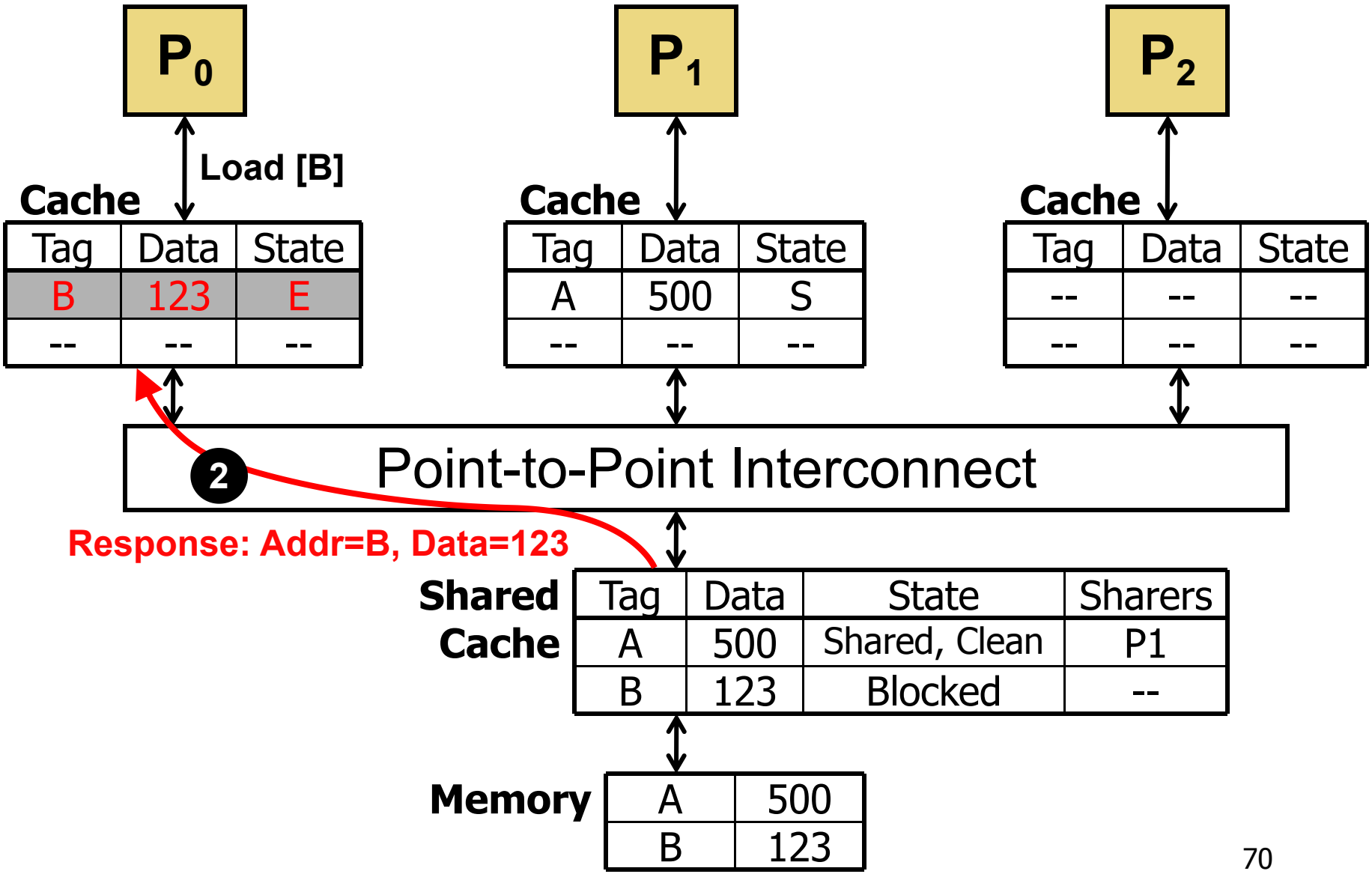
MESI Coherence Example: Step #2



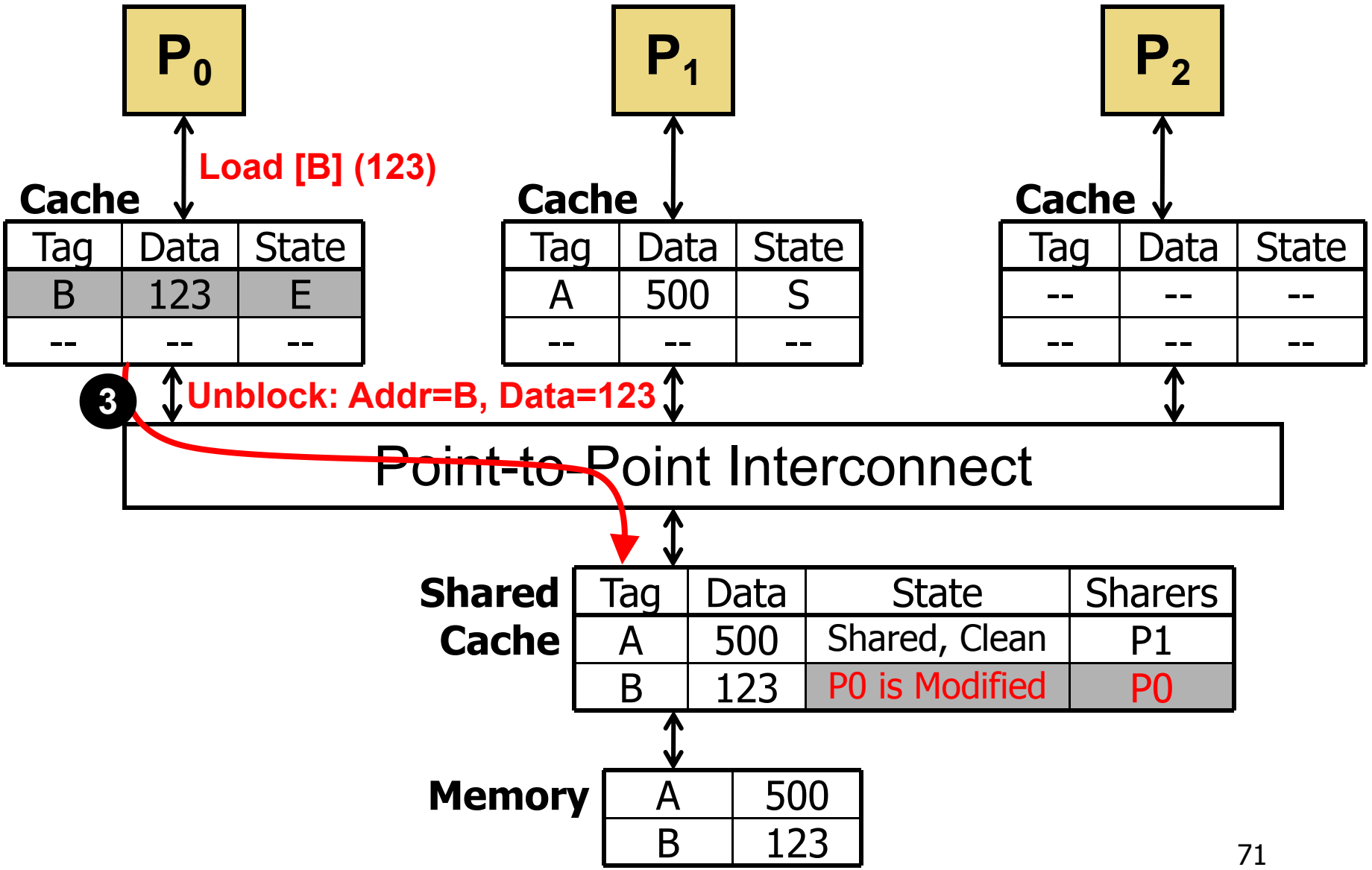
MESI Coherence Example: Step #3



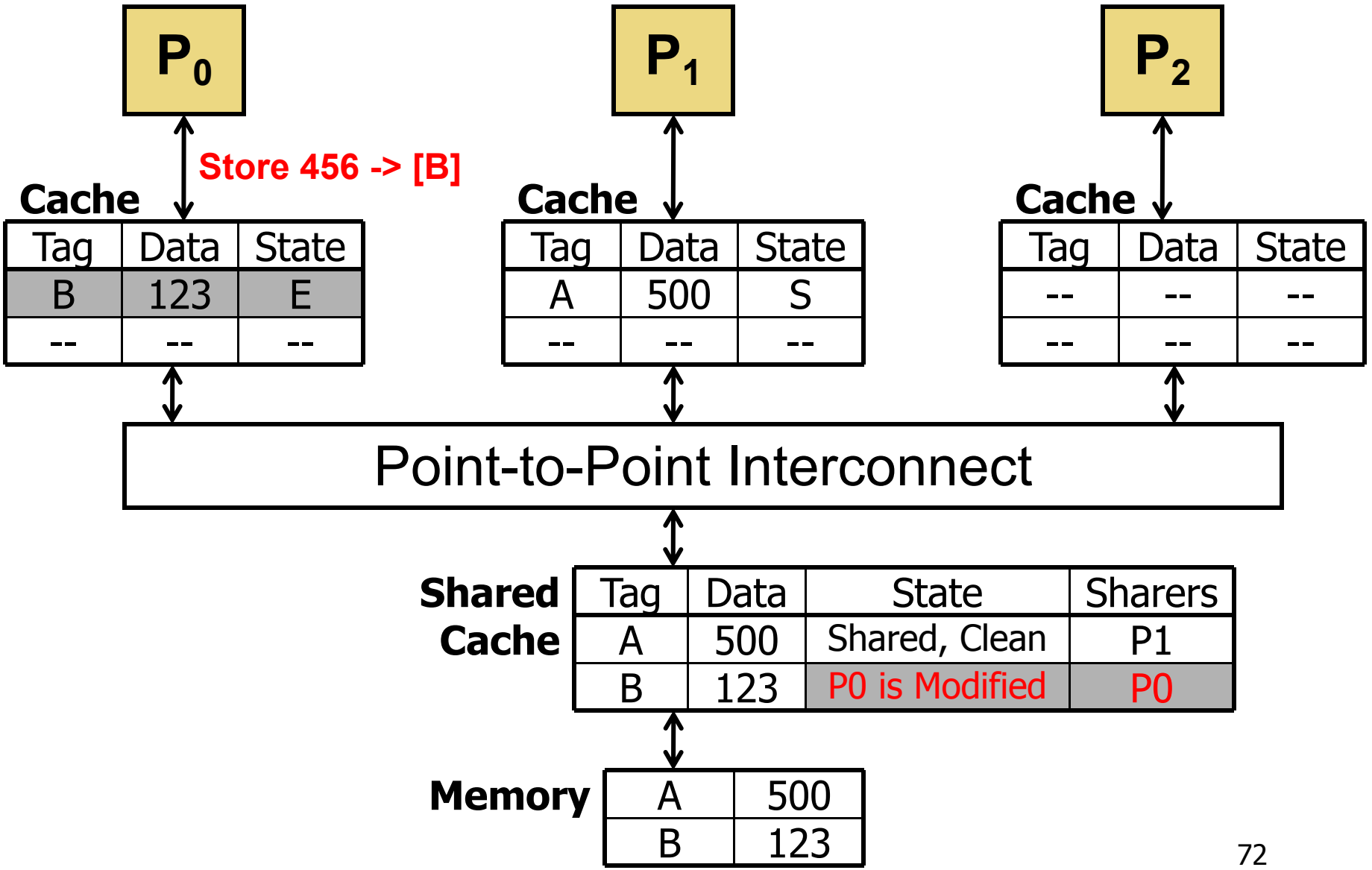
MESI Coherence Example: Step #4



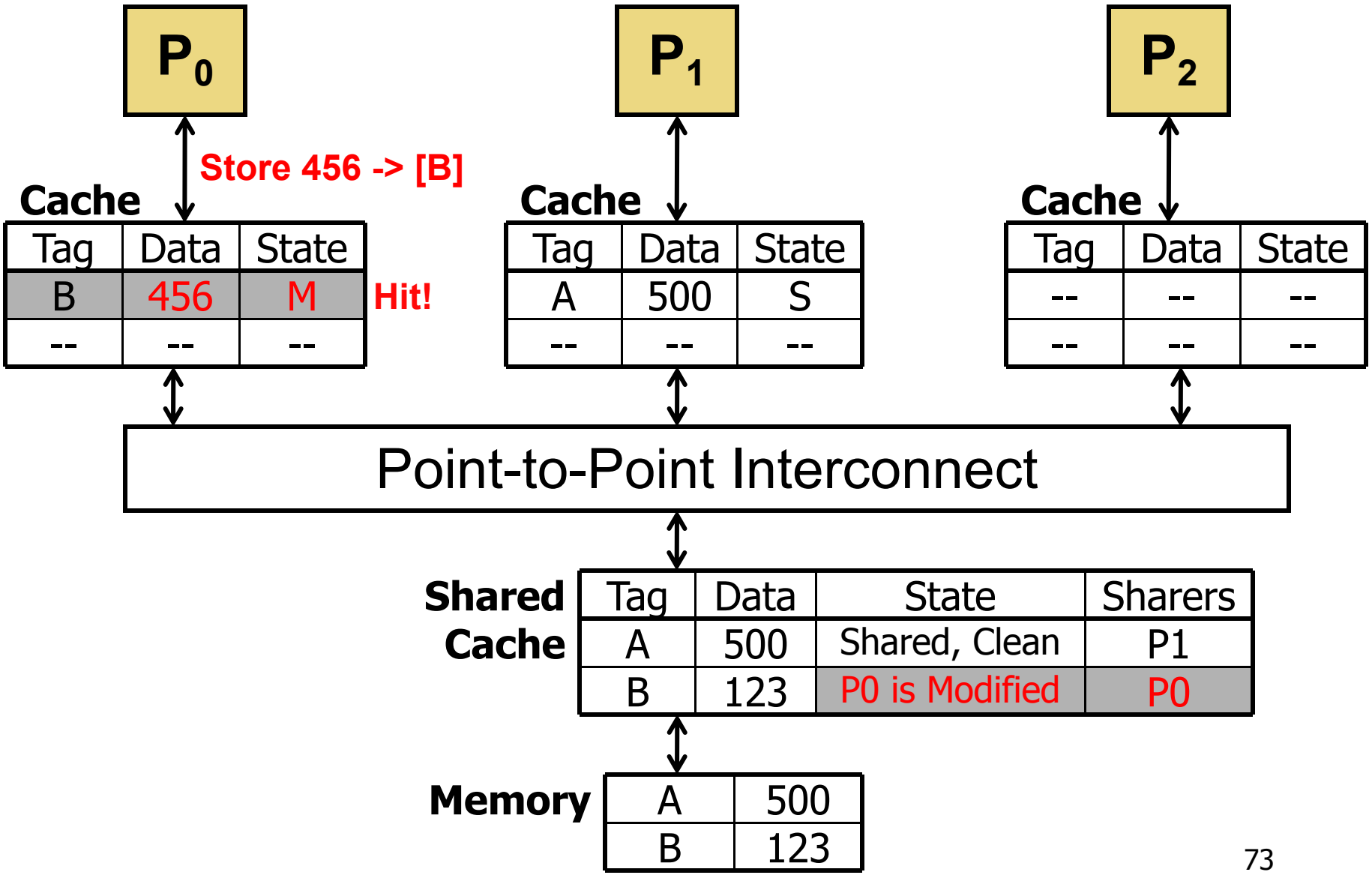
MESI Coherence Example: Step #5



MESI Coherence Example: Step #6



MESI Coherence Example: Step #7

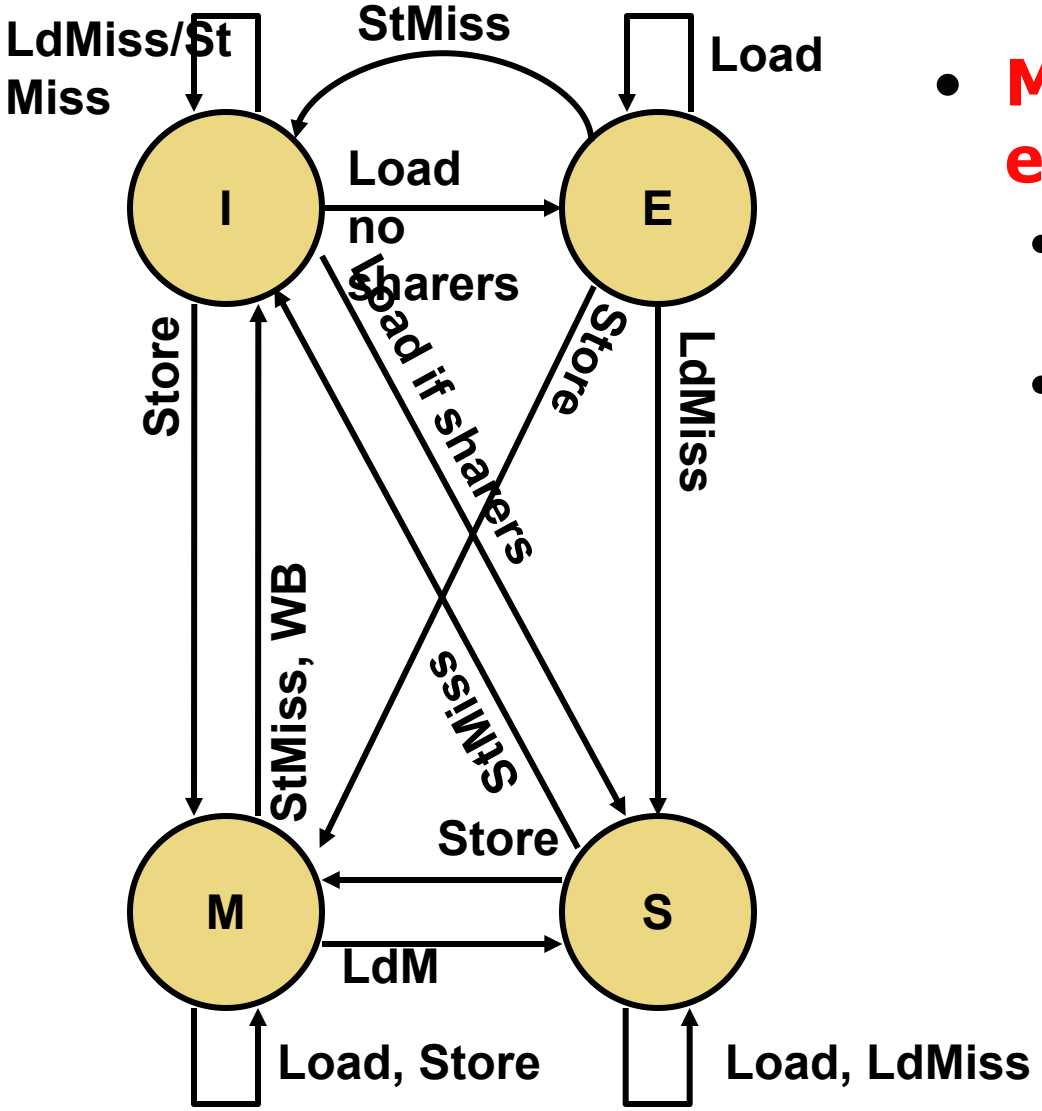


MESI Protocol State Transition Table

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

MSI → MESI



- **MESI (modified-exclusive-shared-invalid)**
  - Eliminates the cost of coherence when there's no sharing
  - Keeps single-threaded programs fast on multicores

Coherence and writeback caches

- there's redundancy between dirty bit and coherence state
  - an Exclusive block is always clean
  - a Modified block is always dirty
  - a Shared block could be clean *or* dirty
    - block was Modified, then later downgraded to Shared
- When should we write back Shared blocks?
  - Track a separate dirty bit
  - Always write back Shared blocks?
    - if k writebacks occur, k-1 writebacks are redundant
  - Never write back Shared blocks?
    - previous stores to the Shared block are lost
- Solution: integrate dirty bit into coherence state

MOESI coherence protocol

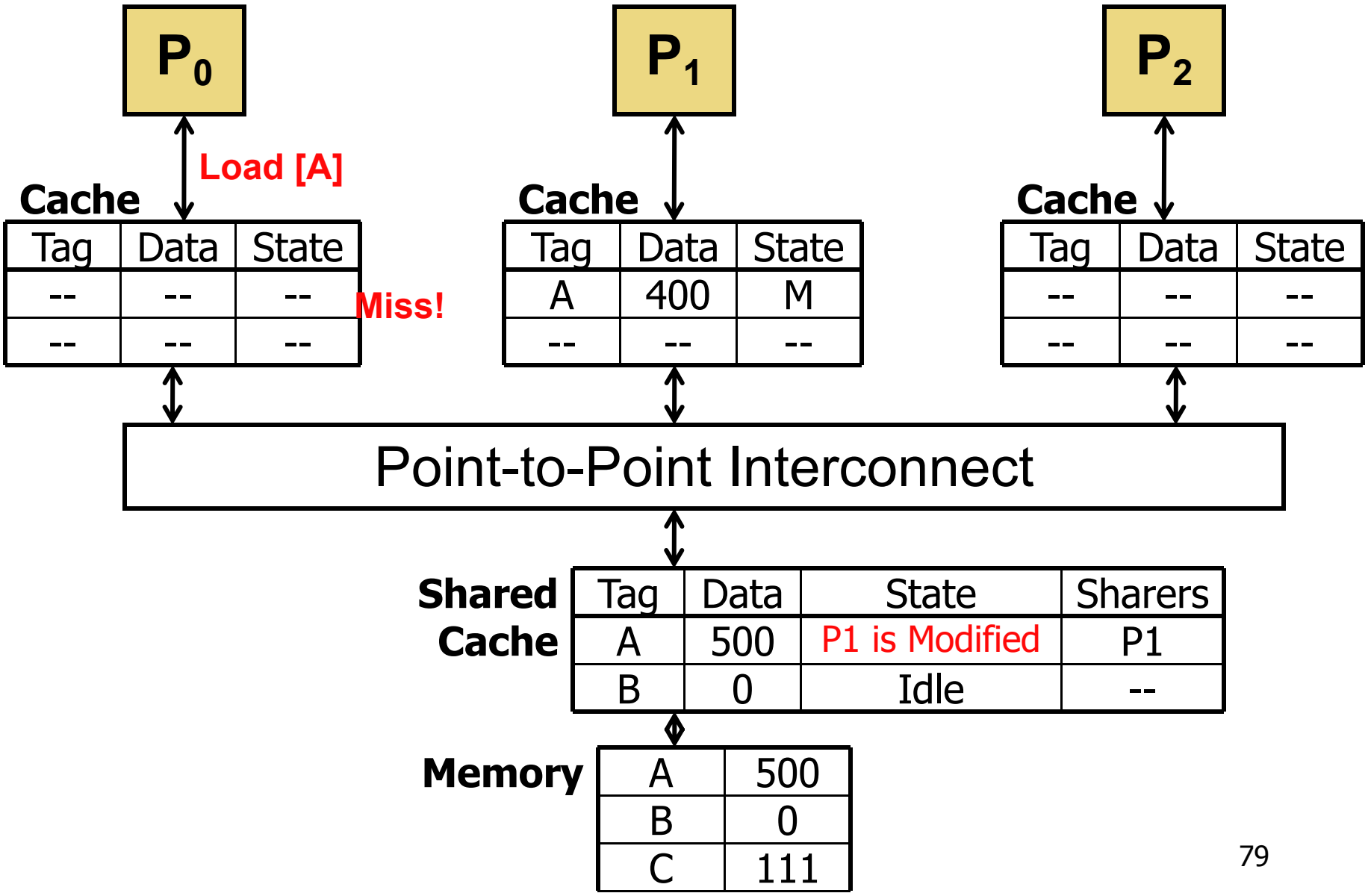
- split Shared state into **Shared** and **Owned** states
  - a Shared block is always clean
  - an Owned block is always dirty
    - an evicted Owned block is written back to the next-level $
    - bonus: don't need multiple Owned copies
      - avoid redundant writebacks
  - dirty bit is fully-redundant now, so scrap it
- this is the **MOESI** ("mow-zee") coherence protocol
  - **common in industrial designs**

MOESI Protocol State Transition Table

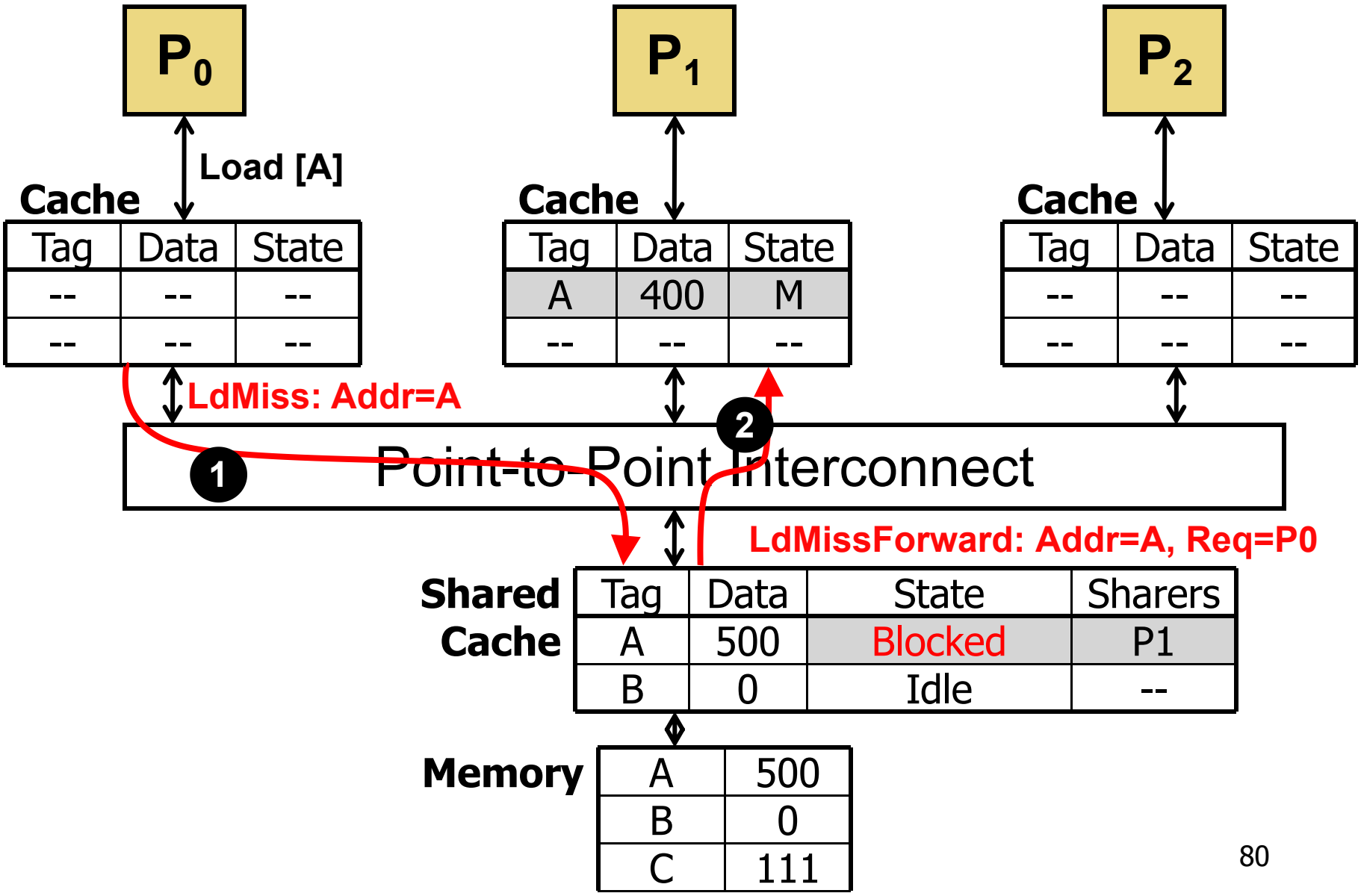
State	*This Processor*		*Other Processor*	
	Load	Store	Load Miss	Store Miss
Invalid (I)	Miss → S or E	Miss → M	---	---
Shared (S)	Hit	Upg Miss → M	Send Data → S	→ I
Owned (O)	Hit	Upg Miss → M	Send Data → O	→ I
Exclusive (E)	Hit	Hit → M	Send Data → S	Send Data → I
Modified (M)	Hit	Hit	Send Data → O	Send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

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

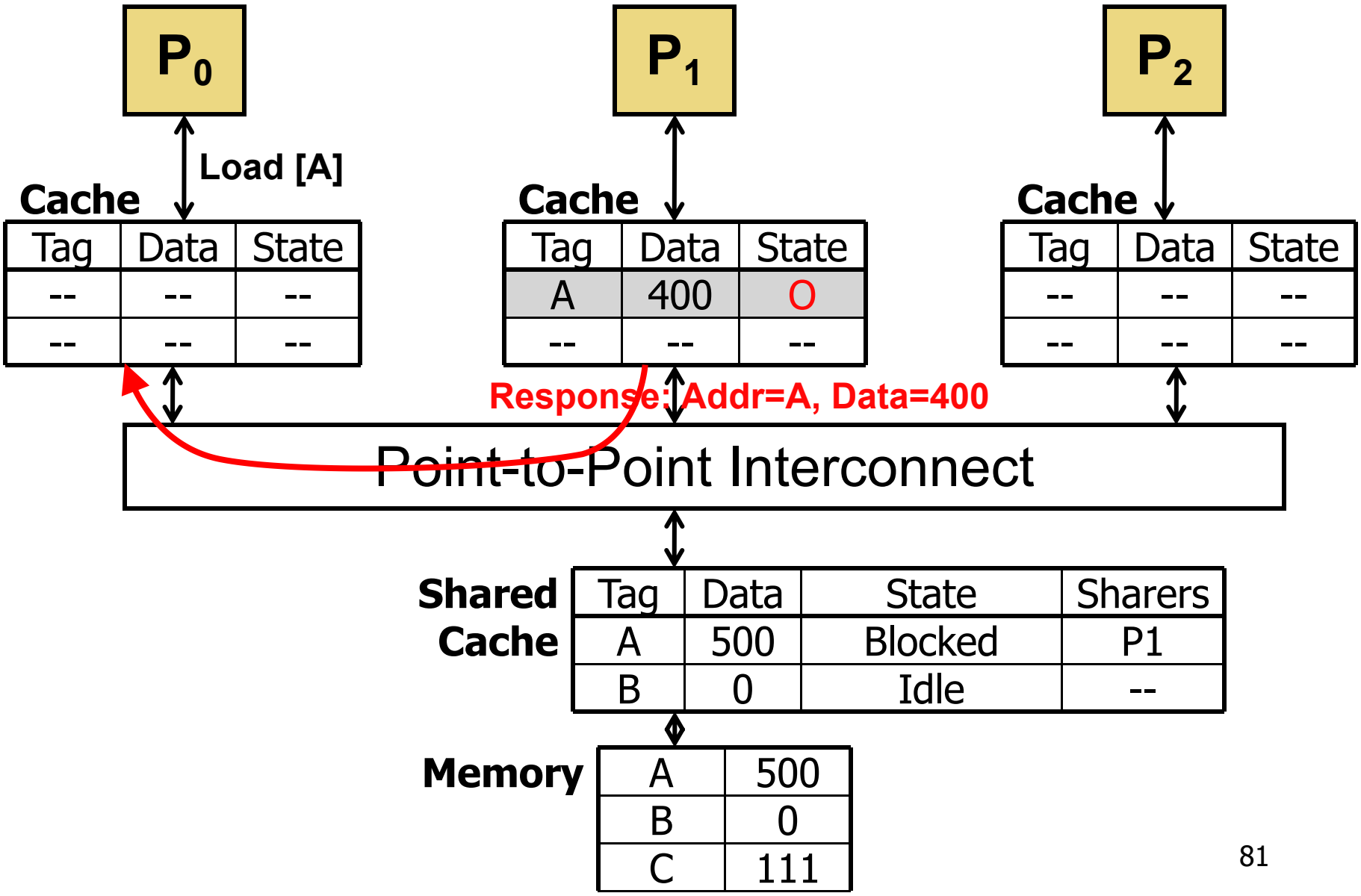
MOESI Coherence Example: Step #1



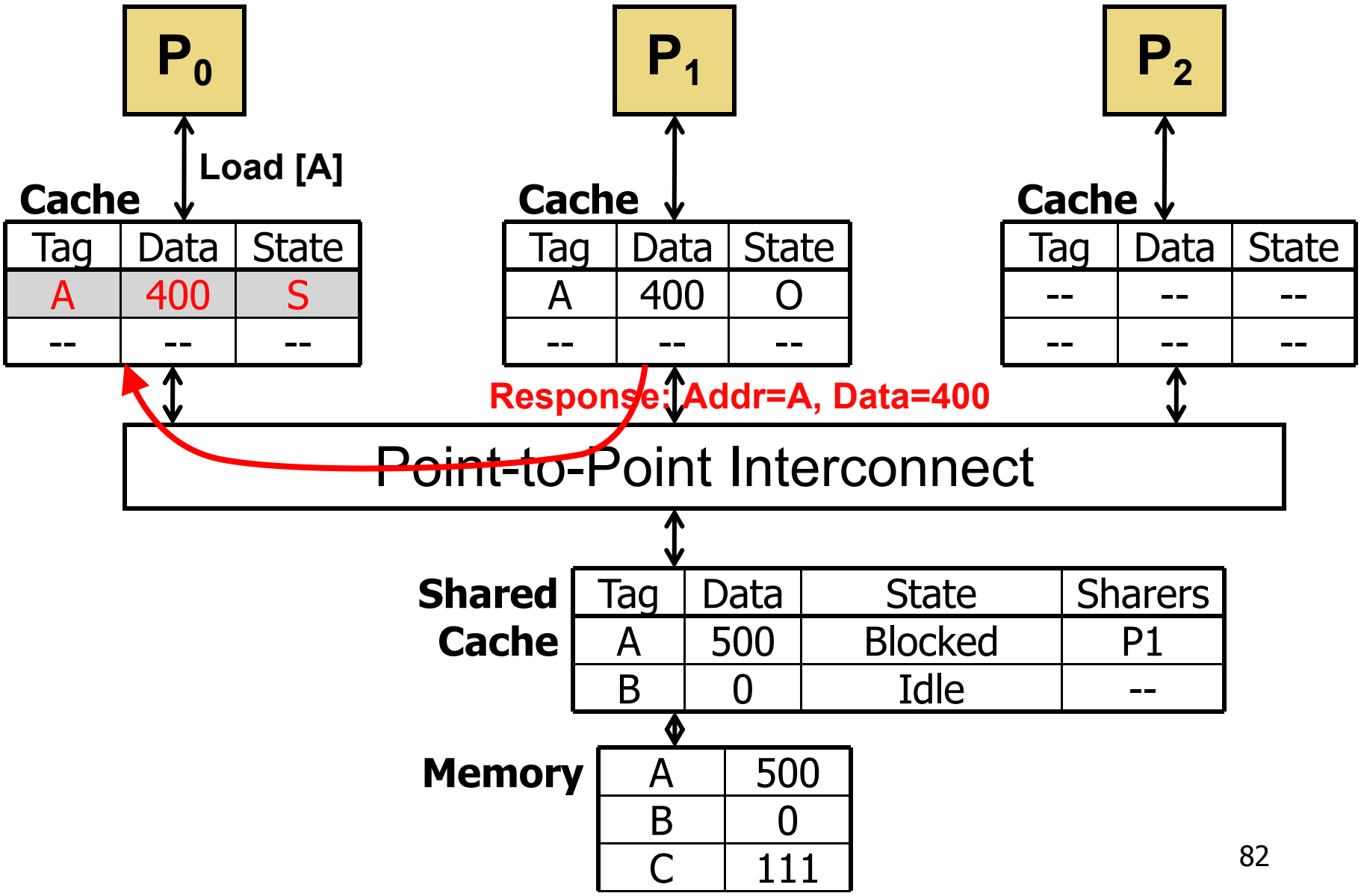
MOESI Coherence Example: Step #2



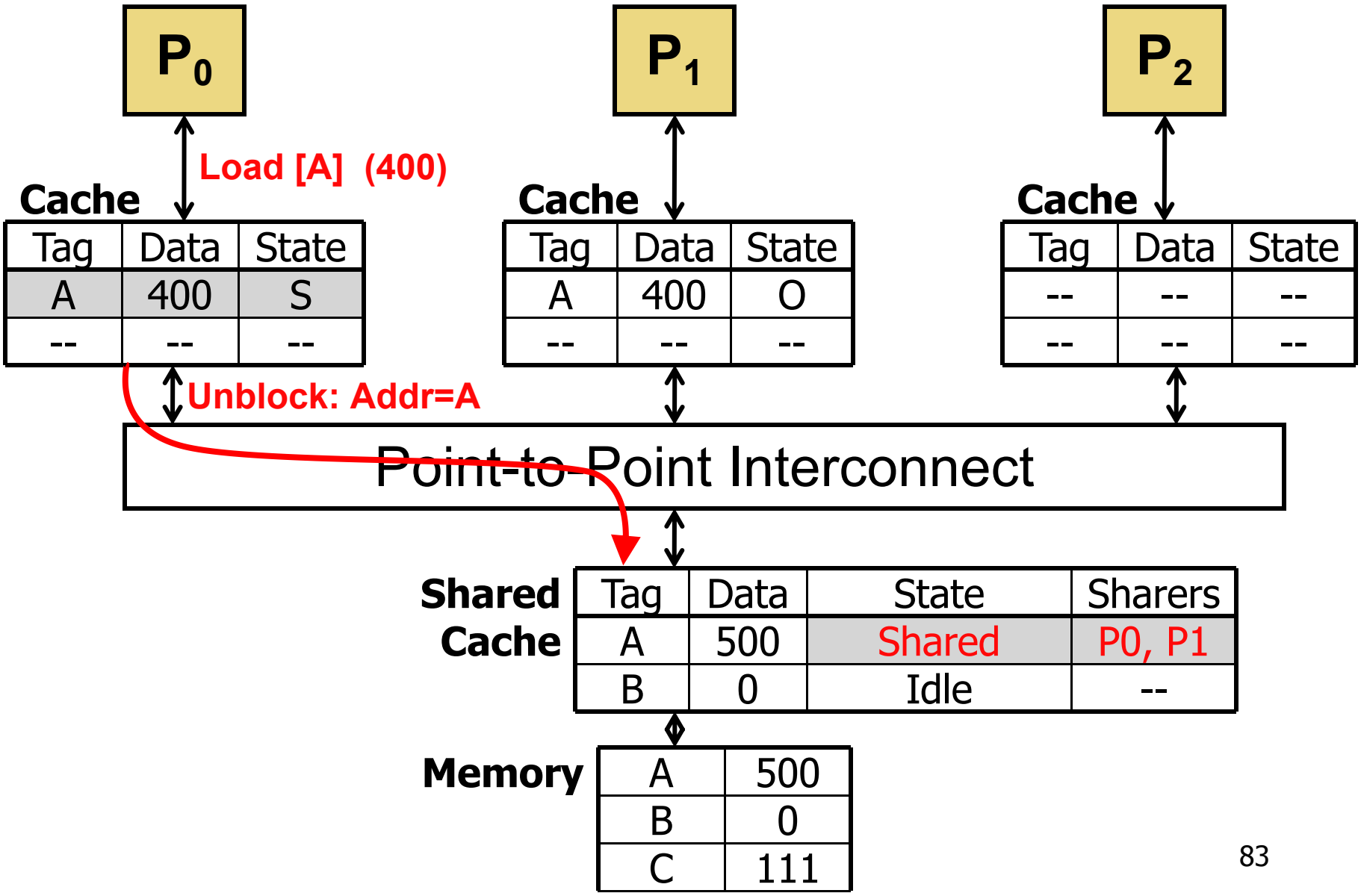
MOESI Coherence Example: Step #3



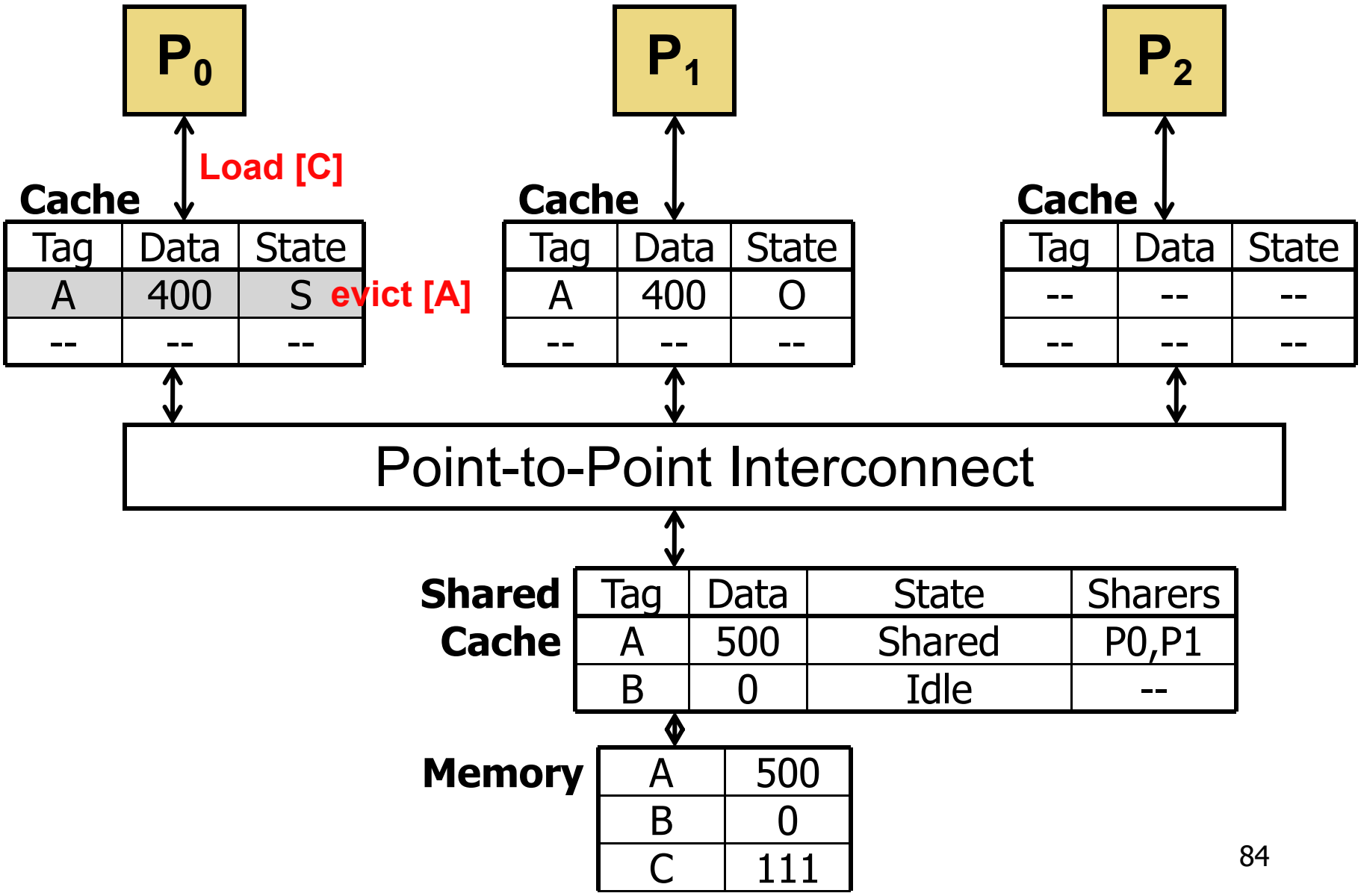
MOESI Coherence Example: Step #4



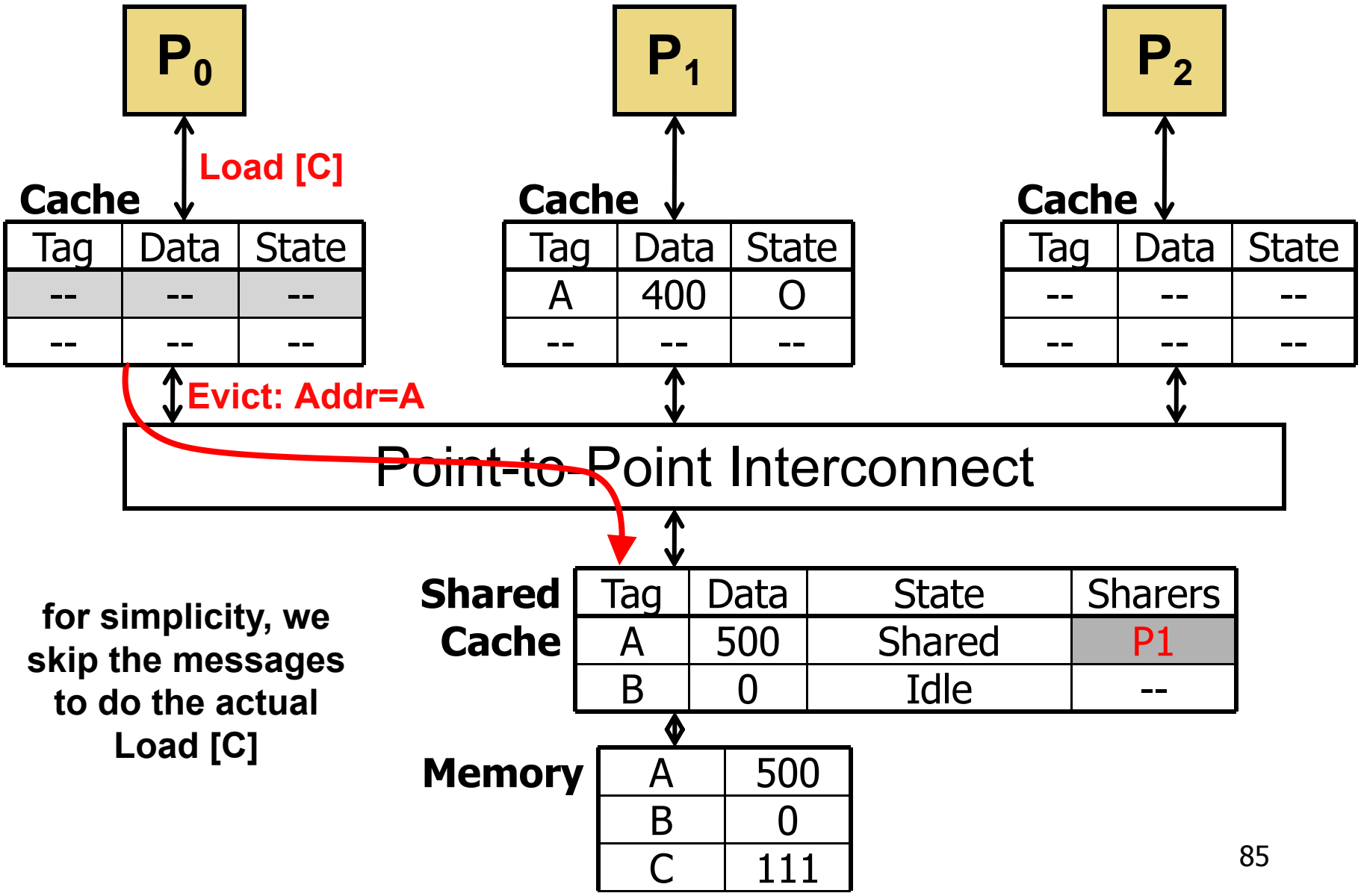
MOESI Coherence Example: Step #5



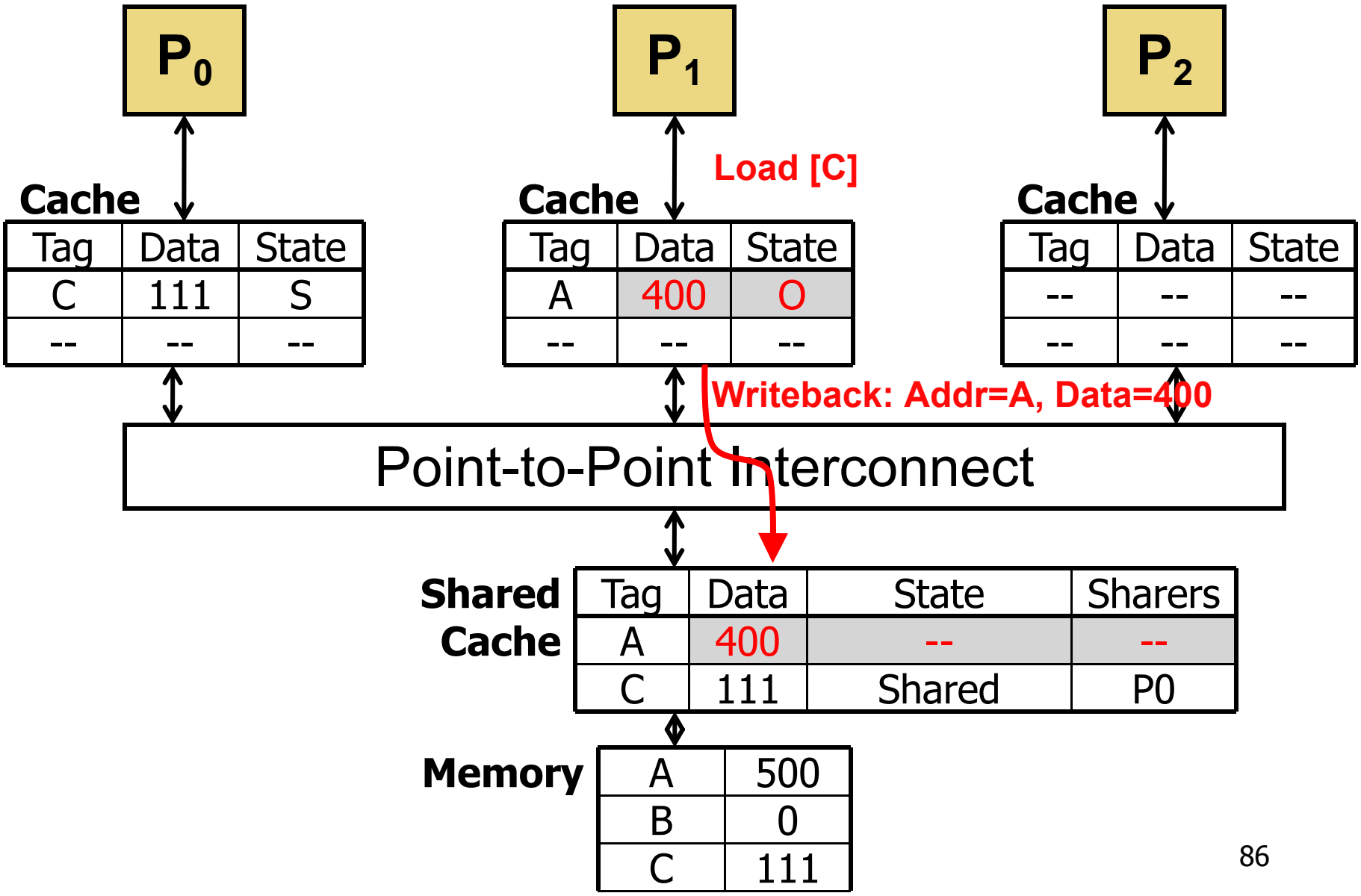
MOESI Coherence Example: Step #6



MOESI Coherence Example: Step #7



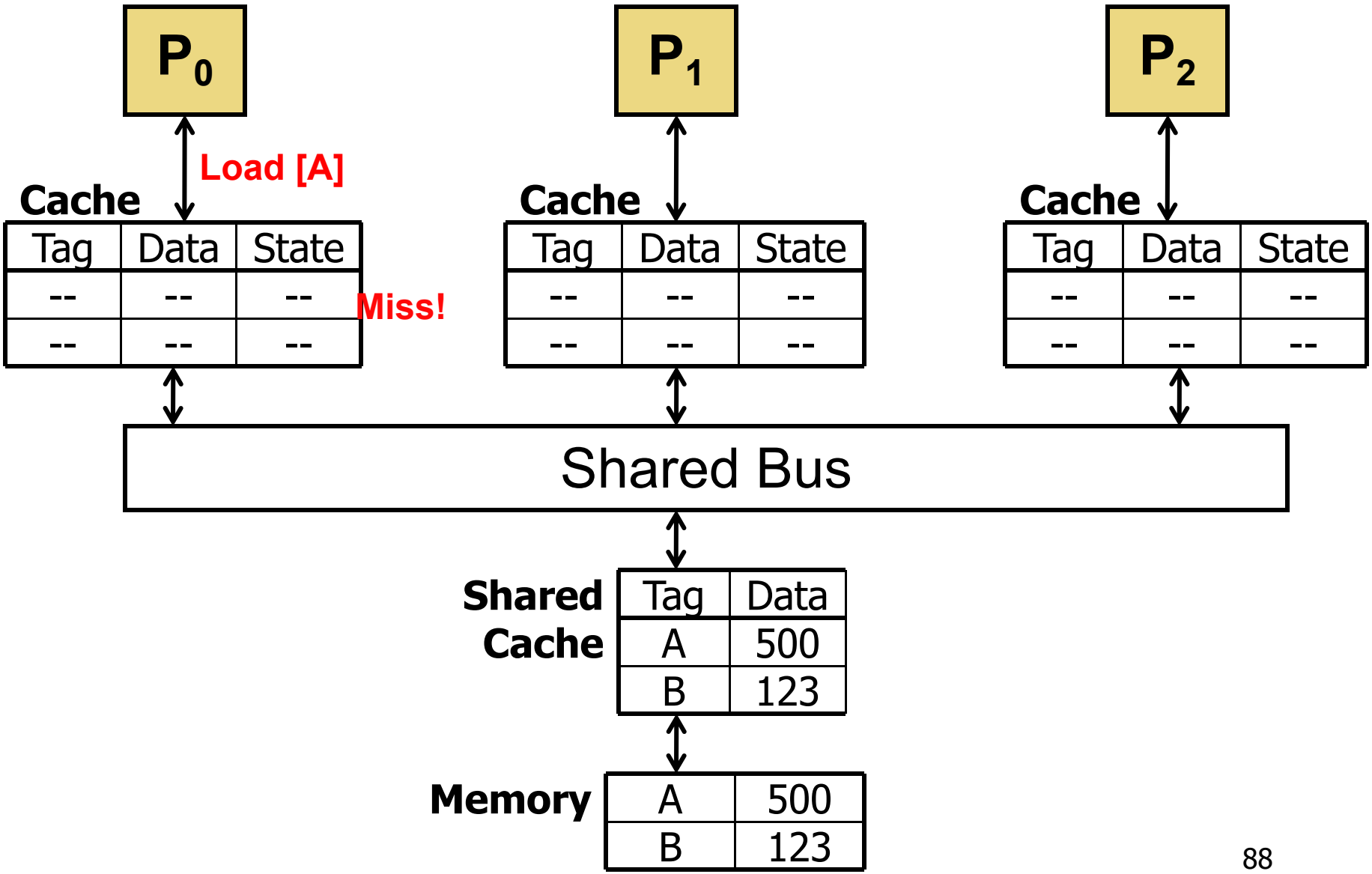
MOESI Coherence Example: Step #8



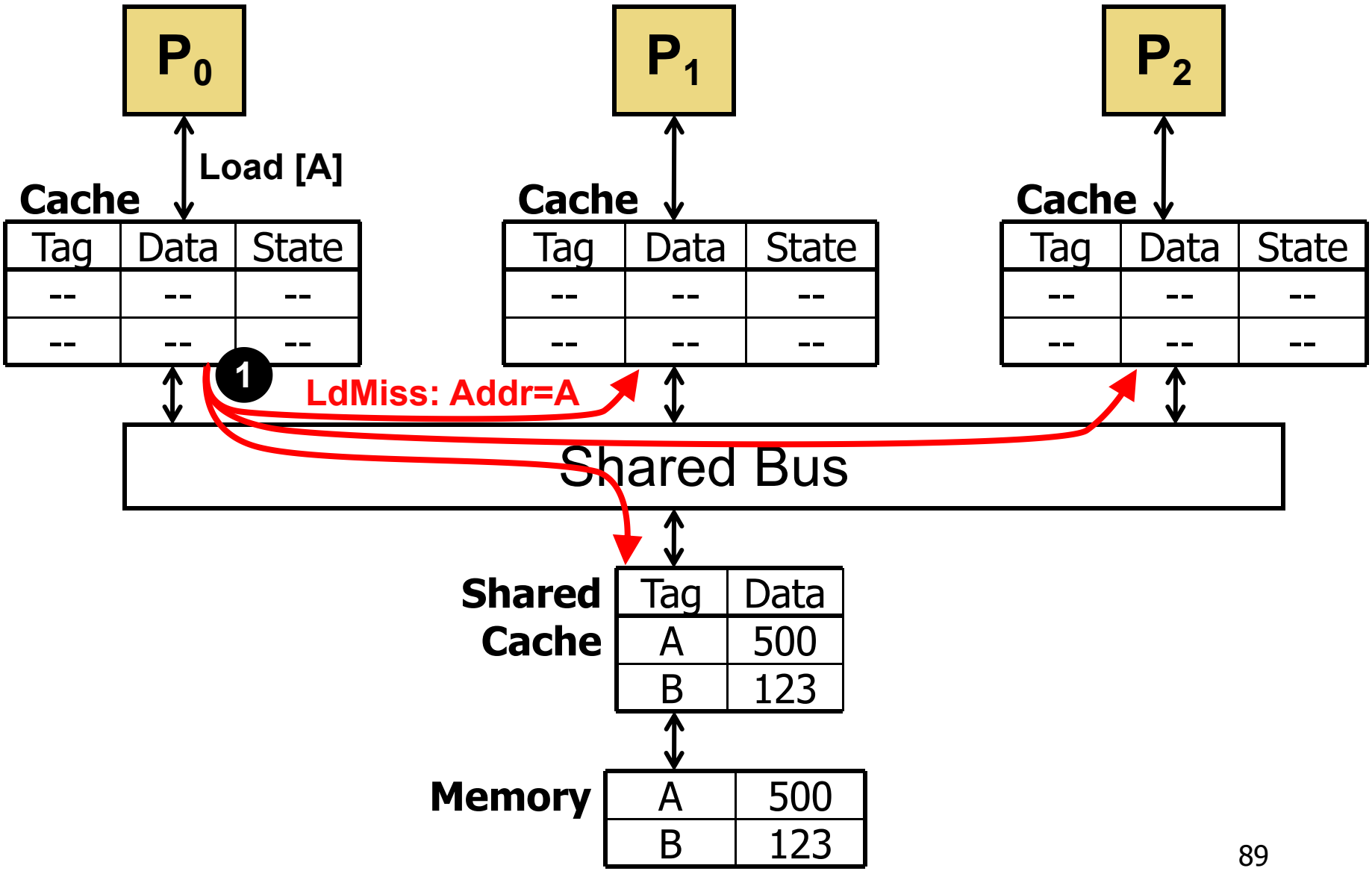
Cache Coherence Protocols

- Two general types
  - Update-based cache coherence
    - Write-through update to all caches
    - Too much traffic; used in the past, not common today
  - **Invalidation-based cache coherence** (examples shown)
- Of invalidation-based cache coherence, two types:
  - Snooping/broadcast-based cache coherence (example next)
    - No explicit state, but too much traffic for large systems
  - **Directory-based cache coherence** (examples shown)
    - Track sharers of blocks
- For directory-based cache coherence, two options:
  - Enforce "inclusion"; if in per-core cache, must be in last-level cache
    - **Encoding sharers in cache tags** (examples shown & Core i7)
  - No inclusion? "directory cache" parallel to last-level cache (AMD)

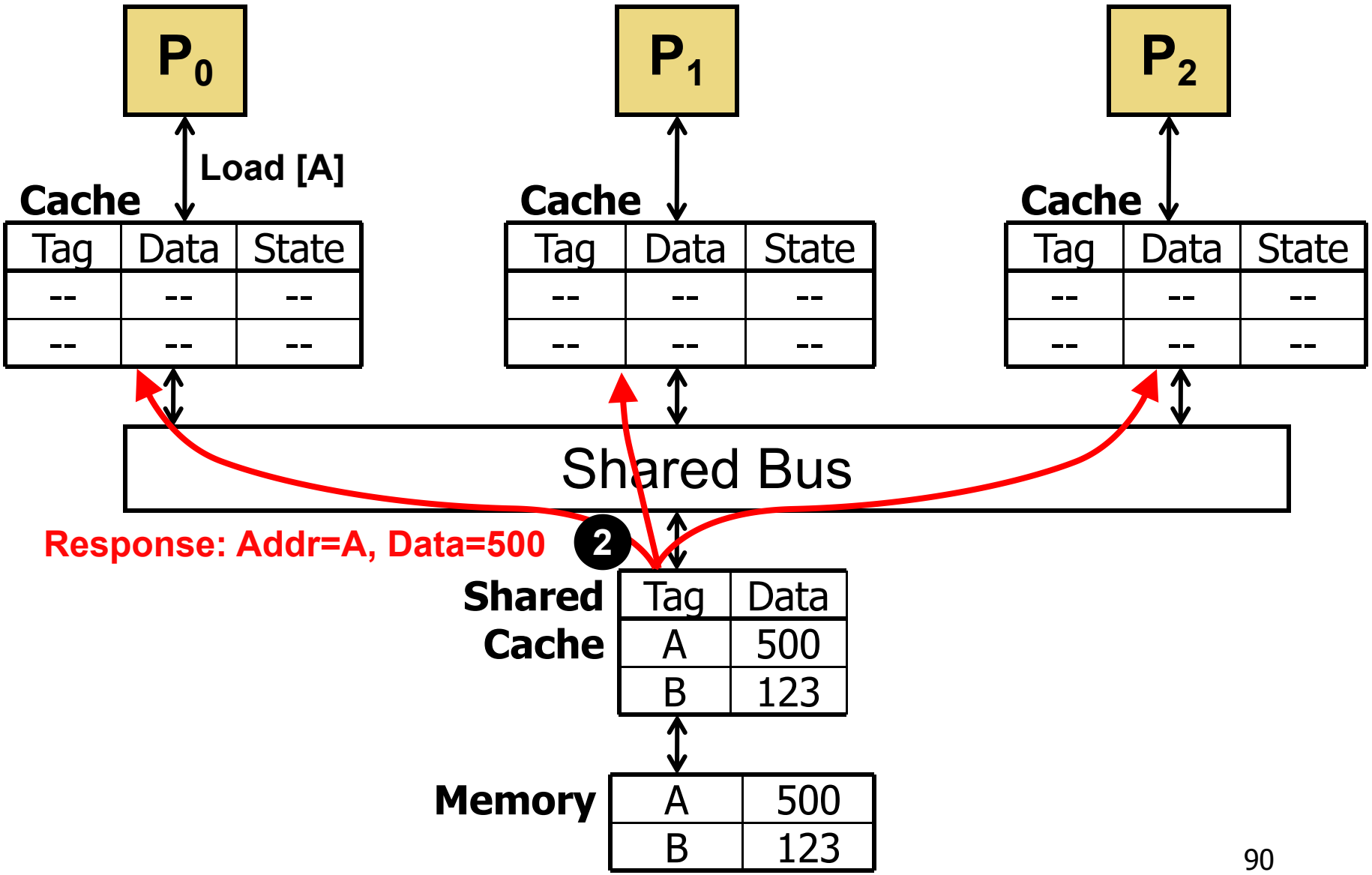
MESI Bus-Based Coherence: Step #1



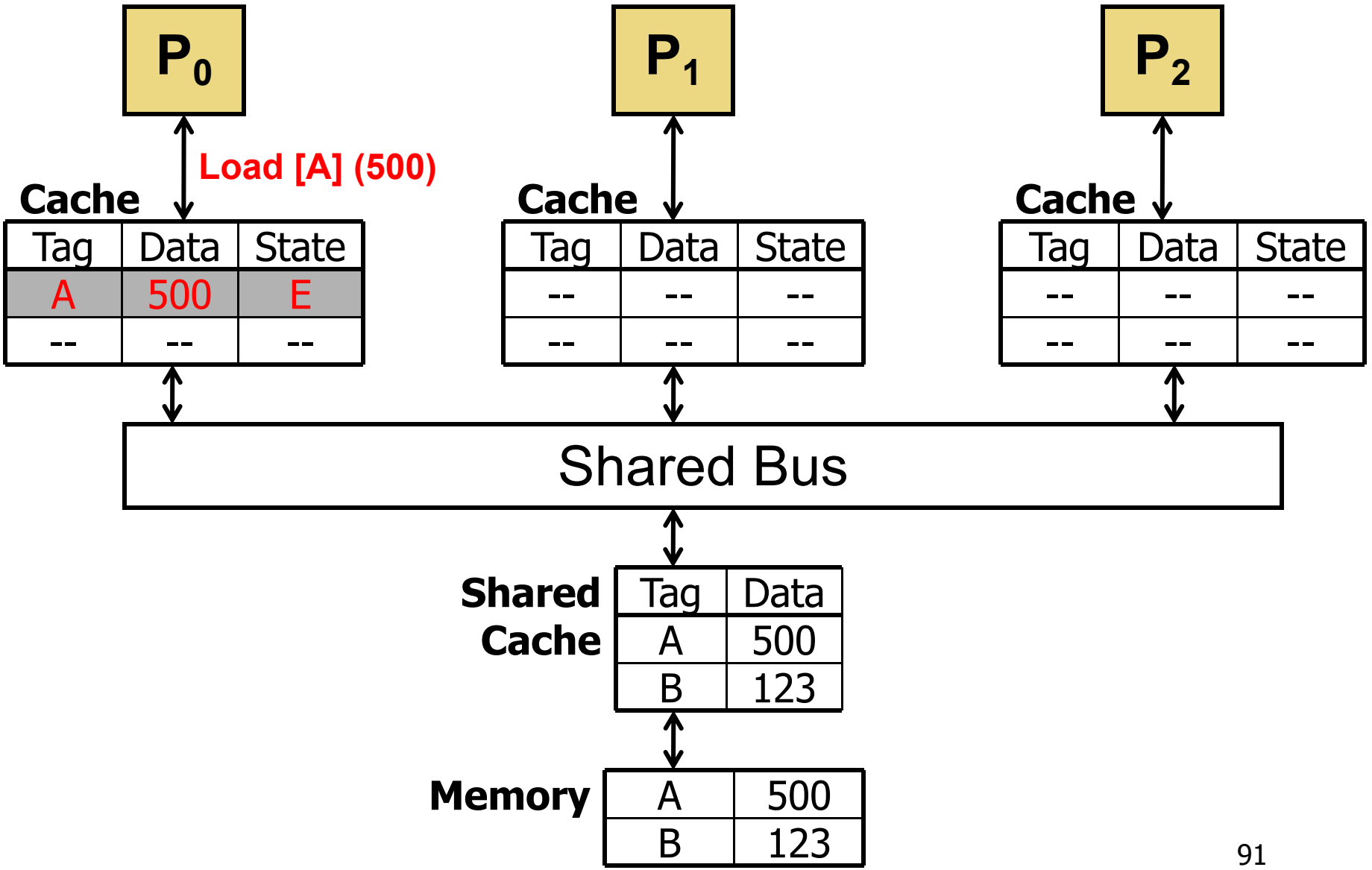
MESI Bus-Based Coherence: Step #2



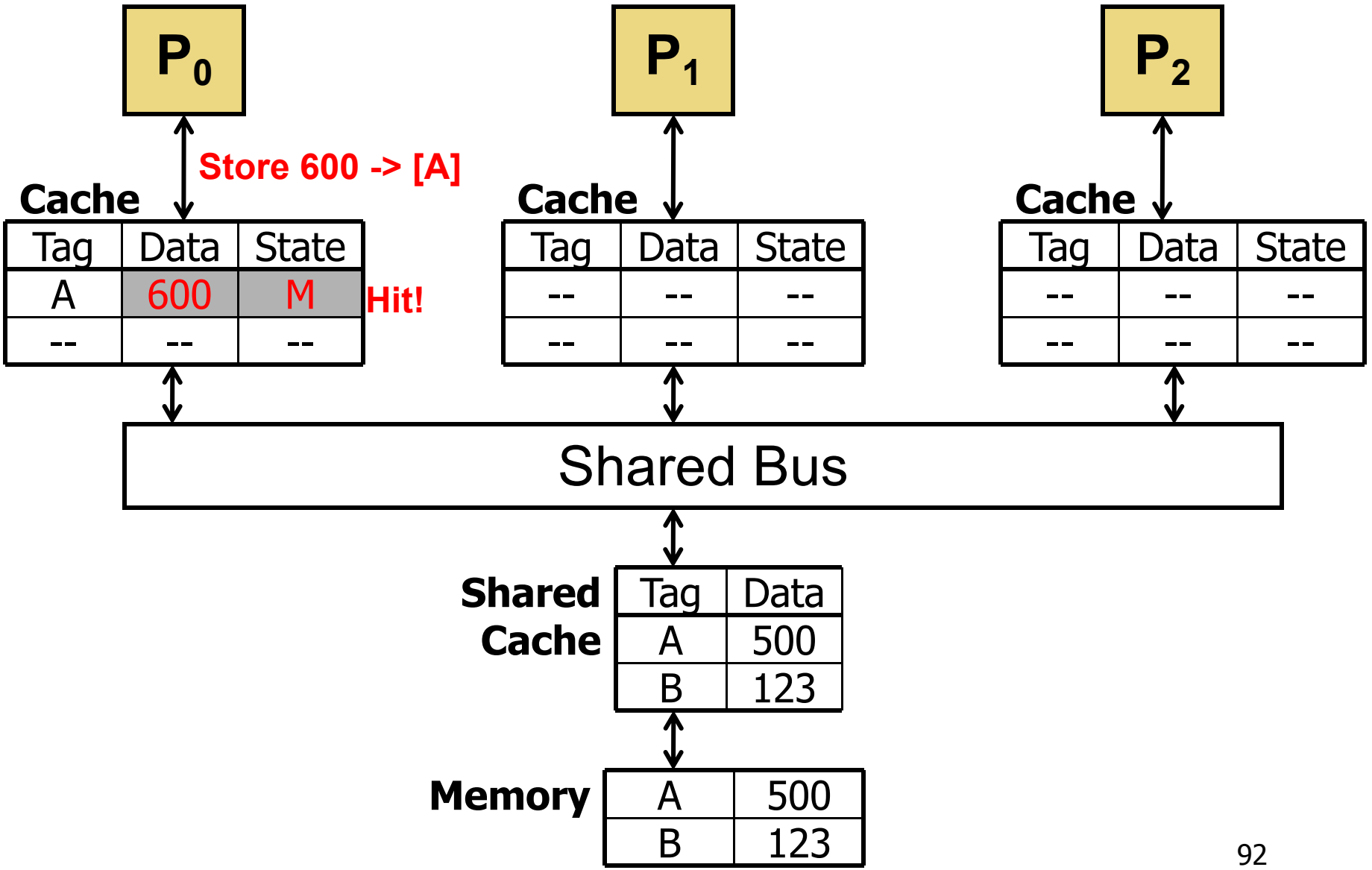
MESI Bus-Based Coherence: Step #3



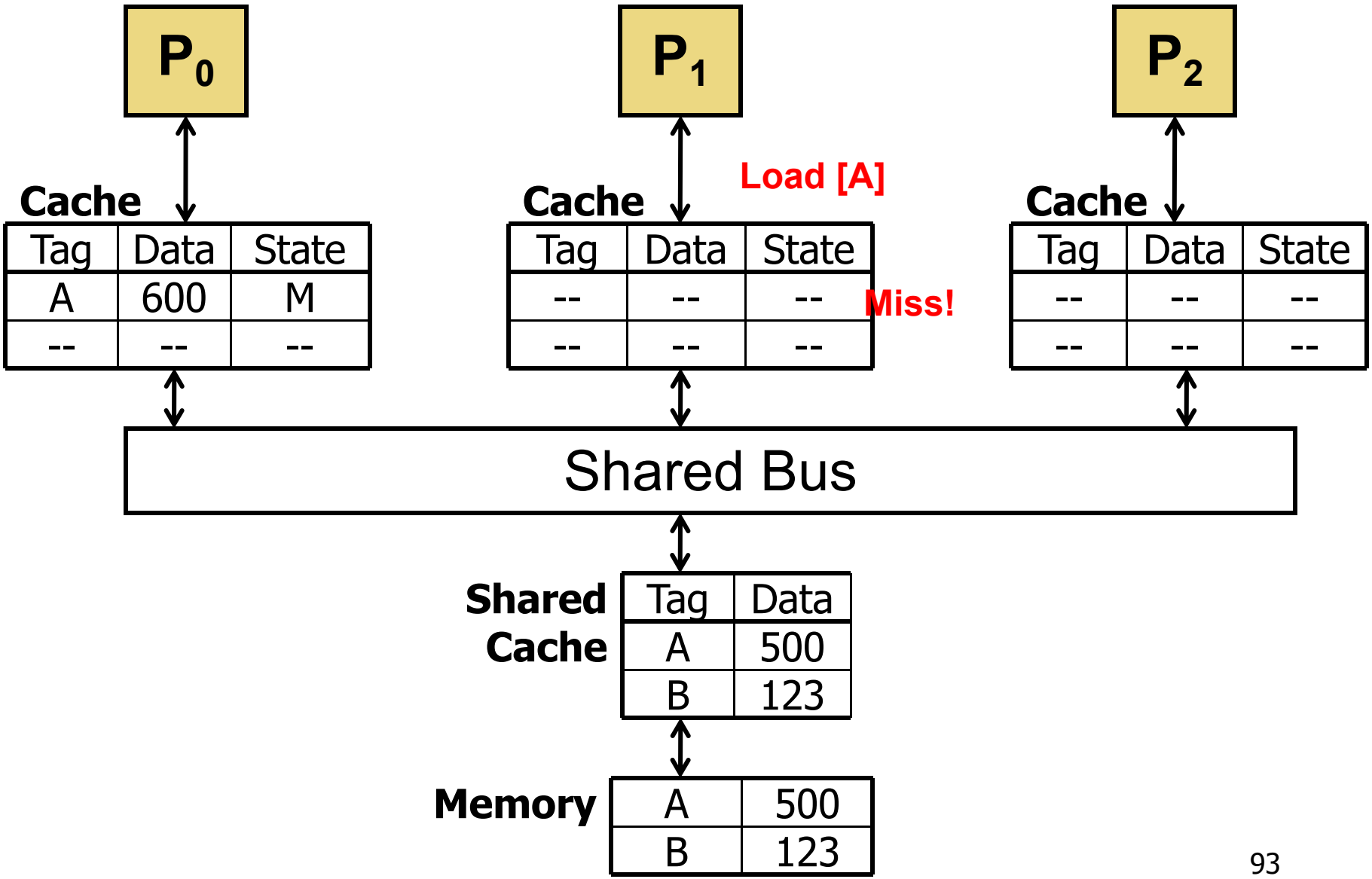
MESI Bus-Based Coherence: Step #4



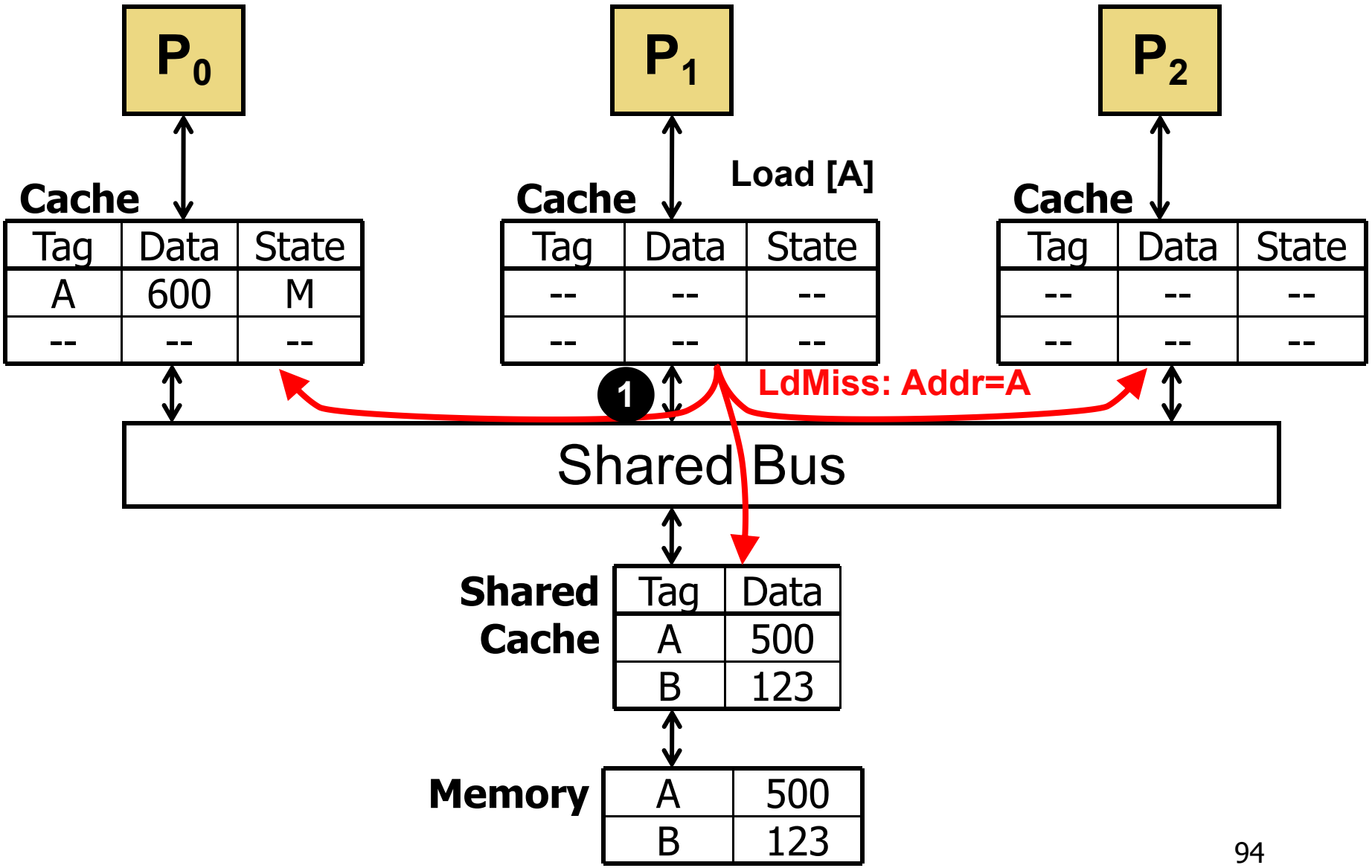
MESI Bus-Based Coherence: Step #5



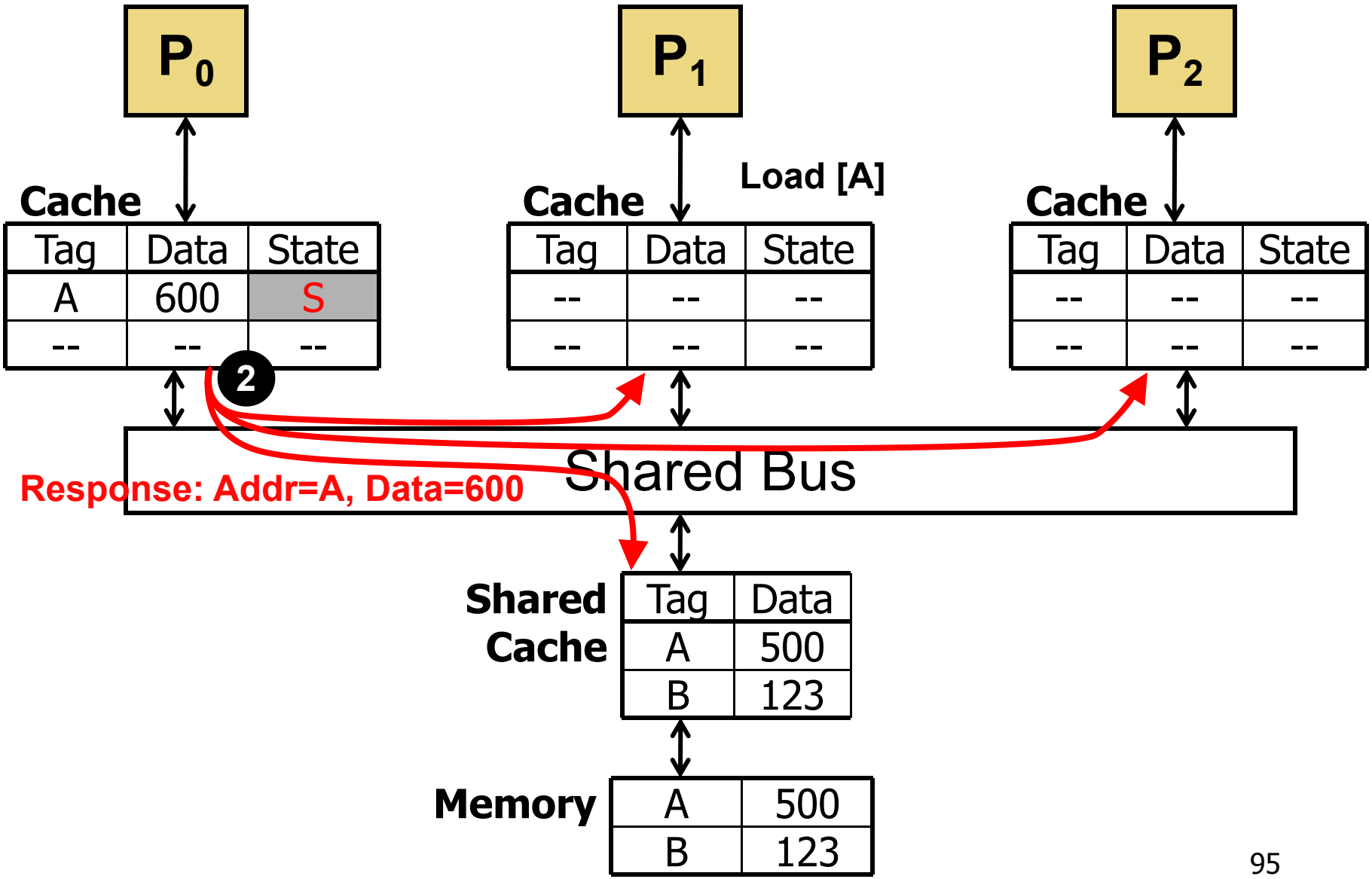
MESI Bus-Based Coherence: Step #6



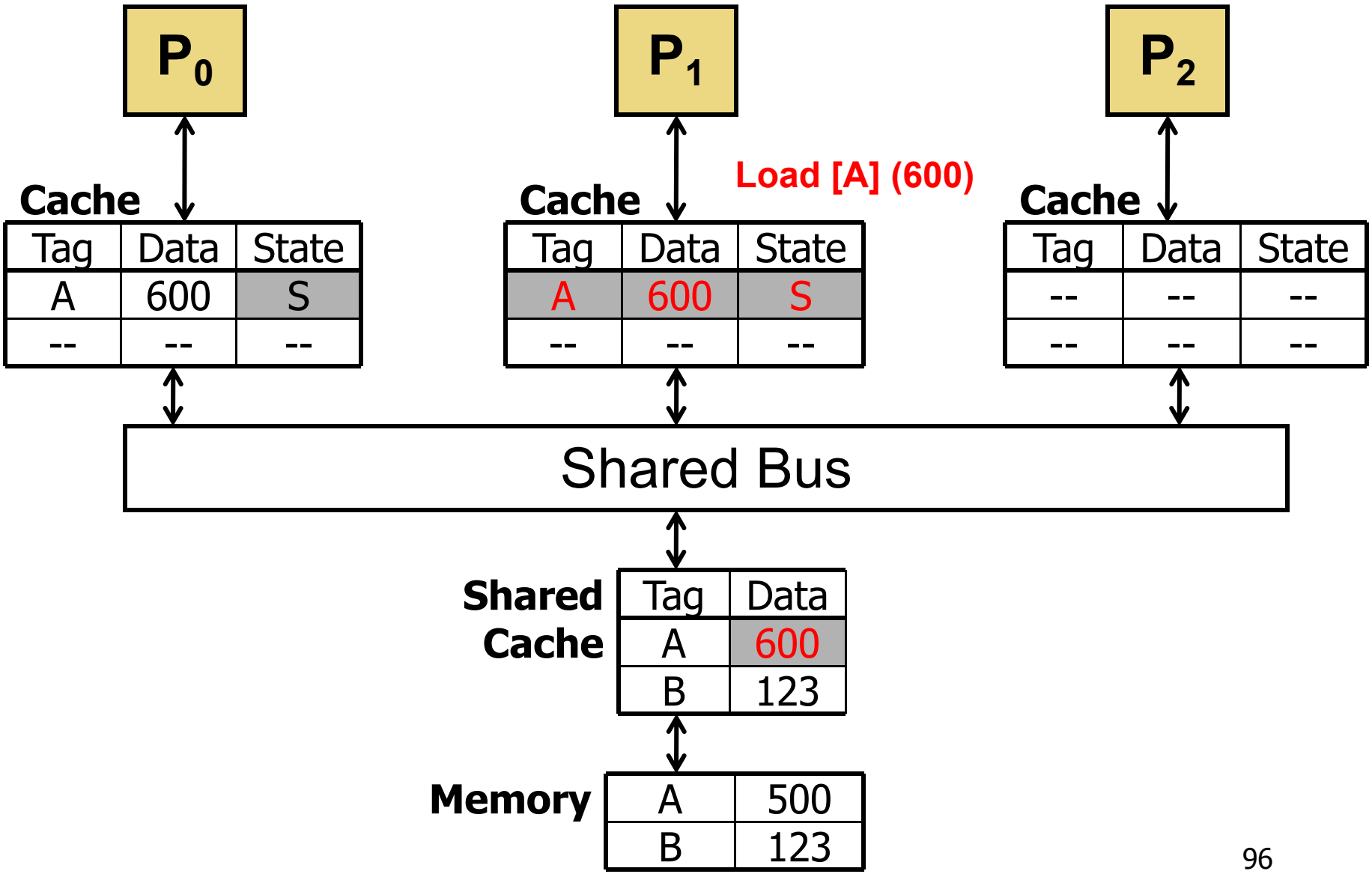
MESI Bus-Based Coherence: Step #7



MESI Bus-Based Coherence: Step #8

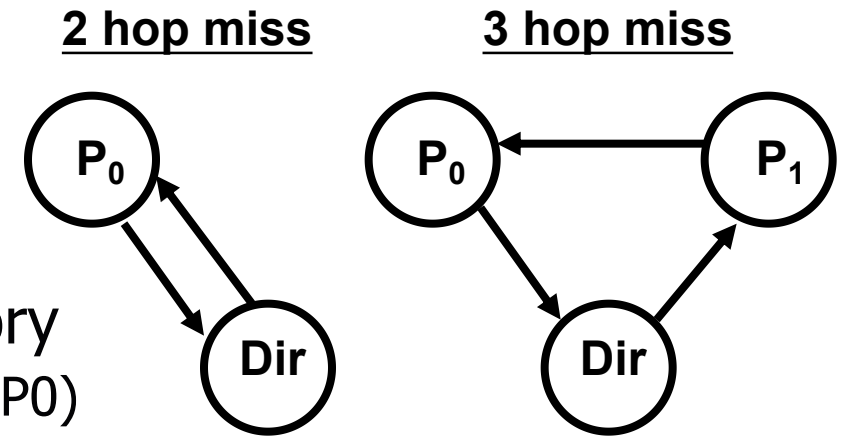


MESI Bus-Based Coherence: Step #9



Directory Downside: Latency

- Directory protocols
  - + Lower bandwidth consumption → more scalable
  - – Longer latencies



- Two read miss situations
- Unshared: get data from memory
  - Snooping: 2 hops (P0→memory→P0)
  - Directory: 2 hops (P0→memory→P0)
- Shared or exclusive: get data from other processor (P1)
  - Assume cache-to-cache transfer optimization
  - Snooping: 2 hops (P0→P1→P0)
  - – Directory: **3 hops** (P0→memory→P1→P0)
  - Common, with many processors high probability someone has it

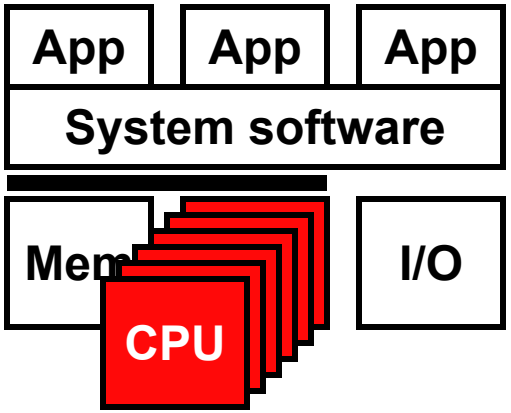
Coherence Recap & Alternatives

- Keeps caches "coherent"
  - Load returns the most recent stored value by any processor
  - And thus keeps caches transparent to software

- Alternatives to cache coherence
  - #1: no caching of shared data (slow)
  - #2: requiring software to explicitly "flush" data (hard to use)
    - Using some new instructions
  - #3: message passing (programming without shared memory)
    - Used in clusters of machines for high-performance computing

- However, directory-based coherence protocol scales well
  - Perhaps to 1000s of cores

Roadmap Checkpoint



- ~~Thread-level parallelism (TLP)~~
- ~~Shared memory model~~
  - ~~Multiplexed uniprocessor~~
  - ~~Hardware multihreading~~
  - ~~Multiprocessing~~
- ~~Cache coherence~~
  - ~~Valid/Invalid, MSI, MESI~~
- **Synchronization**
  - **Lock implementation**
  - **Locking gotchas**
  - **Transactional memory**
- Memory consistency models

Synchronization

Example #1: Bank Accounts

- Consider

```
struct acct_t { int balance; … };
struct acct_t accounts[MAX_ACCT];     // current balances

struct trans_t { int id; int amount; };
struct trans_t transactions[MAX_TRANS];  // debit amounts

for (i = 0; i < MAX_TRANS; i++) {
  debit(transactions[i].id, transactions[i].amount);
}

void debit(int id, int amount) {
  if (accounts[id].balance >= amount) {
    accounts[id].balance -= amount;
  }
}
```

- Can we do "debit" operations in parallel?
  - Does the order matter?

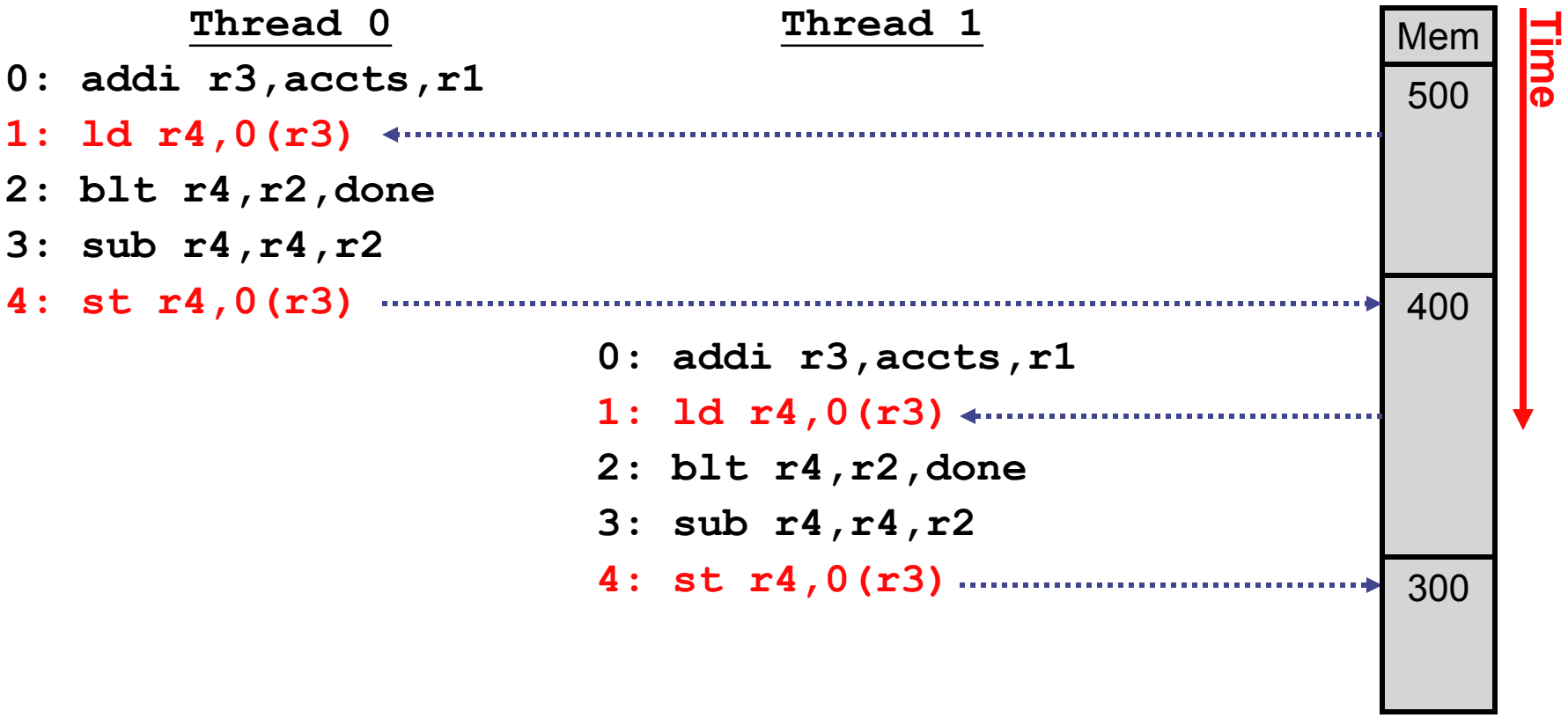
Example #1: Bank Accounts

```
struct acct_t { int bal; … };
shared struct acct_t accts[MAX_ACCT];
void debit(int id, int amt) {
  if (accts[id].bal >= amt)
  {
     accts[id].bal -= amt;
  }
}
```

```
0: addi r3,accts,r1
1: ld r4,0(r3)
2: blt r4,r2,done
3: sub r4,r4,r2
4: st r4,0(r3)
```

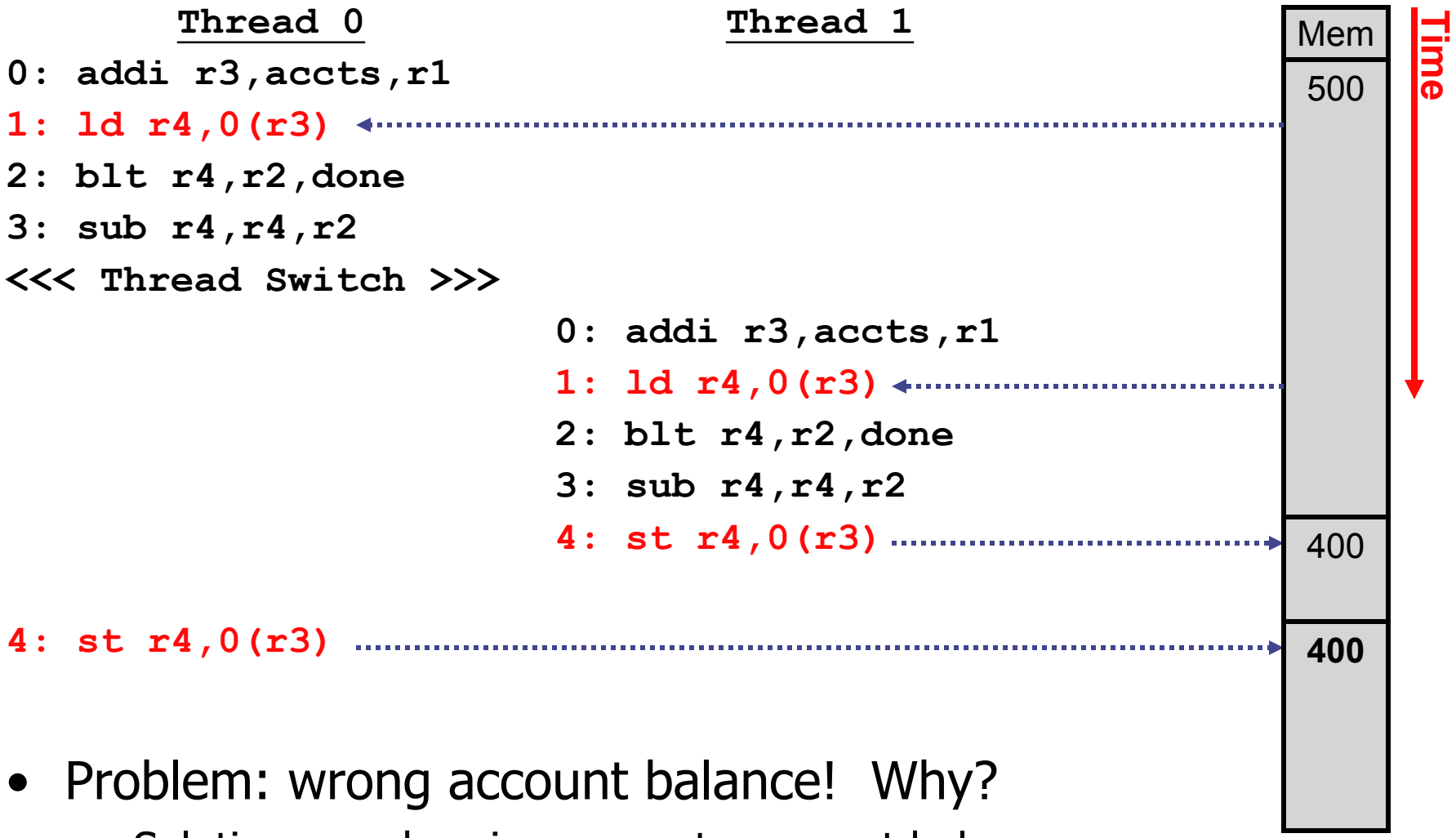
- Example of **Thread-level parallelism (TLP)**
  - Collection of asynchronous tasks: not started and stopped together
  - Data shared "loosely" (sometimes yes, mostly no), dynamically
- Example: database/web server (each query is a thread)
  - `accts` is global and thus **shared**, can't register allocate
  - `id` and `amt` are private variables, register allocated to `r1`, `r2`
- Running example

An Example Execution



- Two $100 withdrawals from account #241 at two ATMs
  - Each transaction executed on different processor
  - Track `accts[241].bal` (address is in `r3`)

A **Problem** Execution



- Problem: wrong account balance! Why?
  - Solution: synchronize access to account balance

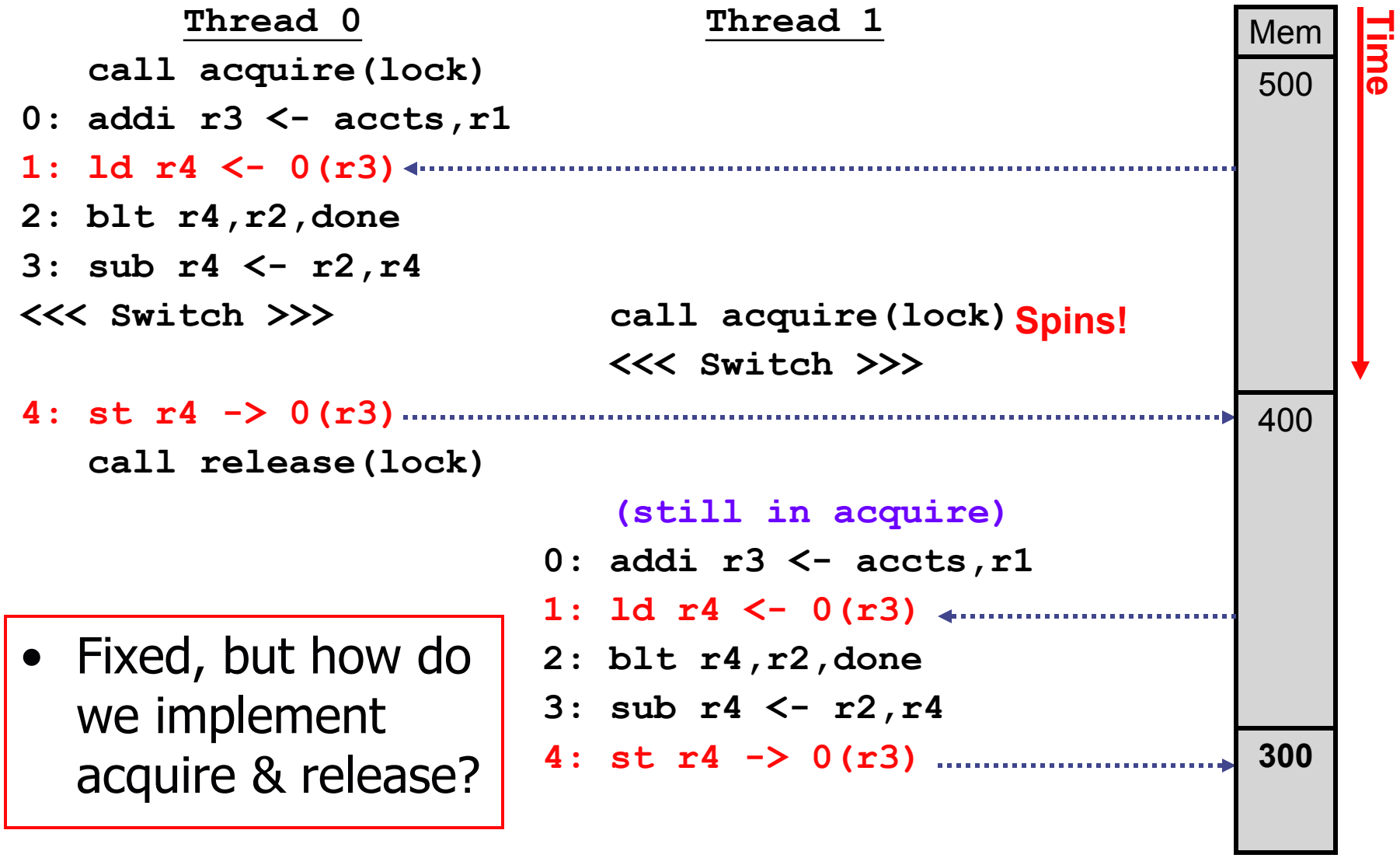
Synchronization:

- **Synchronization**: a key issue for shared memory
- Regulate access to shared data (mutual exclusion)
- Low-level primitive: **lock** (higher-level: "semaphore")
  - Operations: `acquire(lock)` and `release(lock)`
  - Region between `acquire` and `release` is a **critical section**
  - Must interleave `acquire` and `release`
  - Interfering `acquire` will block
- Another option: **Barrier synchronization**
  - Blocks until all threads reach barrier, used at end of "parallel_for"

```
struct acct_t { int bal; … };
shared struct acct_t accts[MAX_ACCT];
shared int lock;
void debit(int id, int amt):
  acquire(lock);
  if (accts[id].bal >= amt) {
     accts[id].bal -= amt;
  }
  release(lock);
```

critical section

A Synchronized Execution



- Fixed, but how do we implement acquire & release?

Strawman Lock (Incorrect)

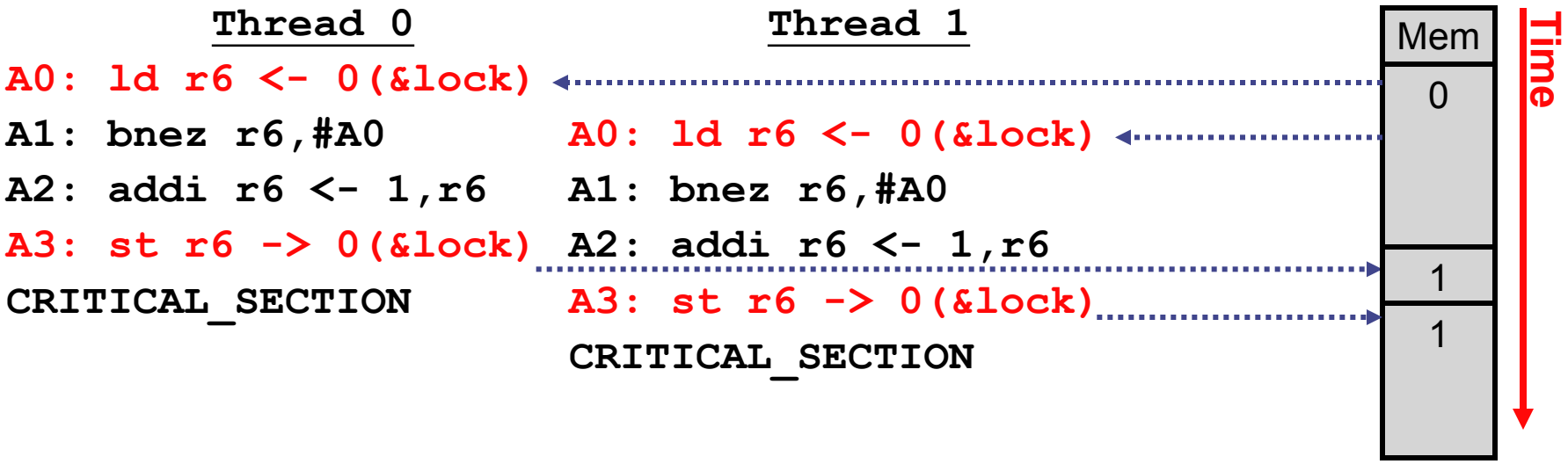
- **Spin lock**: software lock implementation
  - `acquire(lock): while (lock != 0) {} lock = 1;`
    - "Spin" while lock is 1, wait for it to turn 0

```
A0:  ld r6 <- 0(&lock)
A1:  bnez r6,A0
A2:  addi r6 <- 1,r6
A3:  st r6 -> 0(&lock)
```

  - `release(lock): lock = 0;`

```
R0:  st r0 -> 0(&lock)      // r0 holds 0
```

Incorrect Lock Implementation



- Spin lock makes intuitive sense, but doesn't actually work
  - Loads/stores of two `acquire` sequences can be interleaved
  - Lock `acquire` sequence also not atomic
  - **Same problem as before!**

- Note, `release` is trivially atomic

Correct Spin Lock: Compare and Swap

- ISA provides an atomic lock acquisition instruction
  - Example: **atomic compare-and-swap (CAS)**

    `cas r3 <- r1,r2,0(&lock)`

    - Atomically executes:

```
ld r3 <- 0(&lock)
if r3 == r2:
  st r1 -> 0(&lock)
```

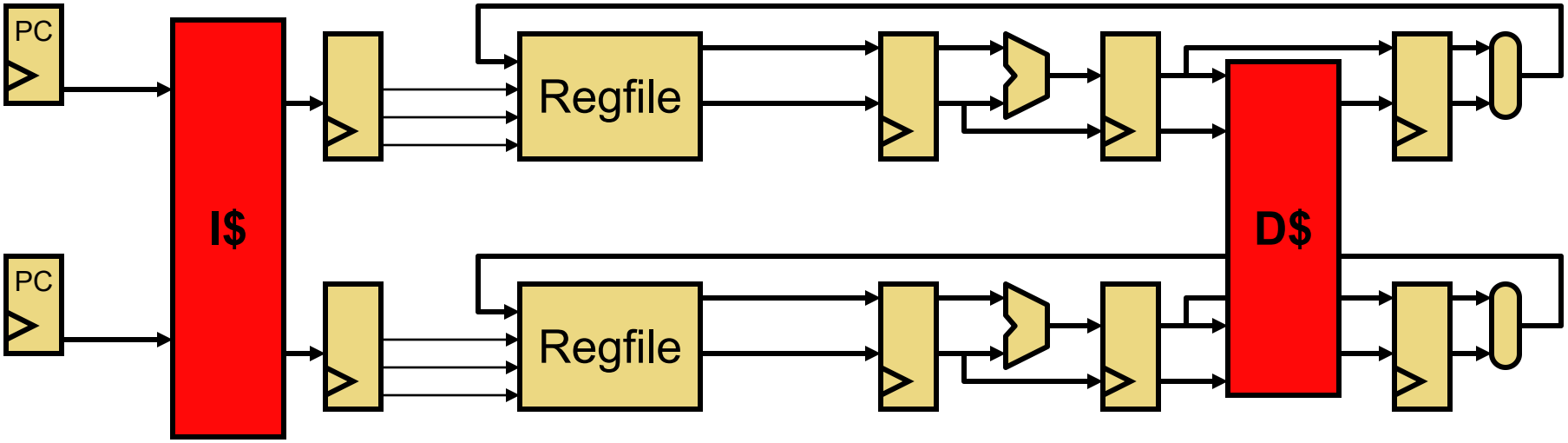
- New acquire sequence

```
A0: cas r3 <- 1,0,0(&lock)
A1: bnez r3,A0
```

  - If lock was initially busy (1), doesn't change it, **keep looping**
  - If lock was initially free (0), acquires it (sets it to 1), break loop

- Ensures lock held by **at most one thread**
  - Other variants: **exchange**, **compare-and-set**, **test-and-set (t&s)**, or **fetch-and-add**

CAS Implementation



- How is CAS implemented?
  - Need to ensure no intervening memory operations
  - Requires blocking access by other threads temporarily

- How to pipeline it?
  - Both a load and a store
  - Not very RISC-like

RISC CAS

- **`CAS`**: a load+branch+store in one insn is not very "RISC"
  - Broken up into micro-ops, but then how is it made atomic?
- "Load-link" / "store-conditional" pairs
  - Atomic load/store pair

```
label:
    load-link r1 <- 0(&lock)
    // potentially other insns
    store-conditional r2 -> 0(&lock)
    branch-not-zero label    // check for failure
```

  - On **`load-link`**, processor remembers address…
    - …And looks for writes by other processors
    - If write is detected, next **`store-conditional`** will fail
      - Sets failure condition
- Used by ARM, PowerPC, MIPS, Itanium

Lock Correctness

Thread 0

```
A0: cas r1 <- 1,0,0(&lock)
A1: bnez r1,#A0
CRITICAL_SECTION
```

Thread 1

```
A0: cas r1 <- 1,0,0(&lock)
A1: bnez r1,#A0
A0: cas r1 <- 1,0,0(&lock)
A1: bnez r1,#A0
```

+ Lock actually works…
  - Thread 1 keeps spinning

- Sometimes called a "test-and-set lock"
  - Named after the common "test-and-set" atomic instruction

"Test-and-Set" Lock Performance

```
Thread 0                          Thread 1
A0: cas r1 <- 1,0,0(&lock)
A1: bnez r1,#A0                   A0: cas r1 <- 1,0,0(&lock)
A0: cas r1 <- 1,0,0(&lock)        A1: bnez r1,#A0
A1: bnez r1,#A0                   A0: cas r1 <- 1,0,0(&lock)
                                  A1: bnez r1,#A0
```

- …but performs poorly
  - Consider 3 processors rather than 2
  - Processor 2 (not shown) has the lock and is in the critical section
  - But what are processors 0 and 1 doing in the meantime?
    - Loops of **cas**, each of which includes a **st**
    - Repeated stores by multiple processors costly
    - Generating a ton of useless interconnect traffic

Test-and-Test-and-Set Locks

- Solution: **test-and-test-and-set locks**
  - New acquire sequence

```
A0: ld r1 <- 0(&lock)
A1: bnez r1,A0
A2: addi r1 <- 1,r1
A3: cas r1 <- r1,0,0(&lock)
A4: bnez r1,A0
```

  - Within each loop iteration, before doing a **swap**
    - Spin doing a simple test (**ld**) to see if lock value has changed
    - Only do a **swap** (**st**) if lock is actually free
  - Processors can spin on a busy lock locally (in their own cache)

    + Less unnecessary interconnect traffic
  - Note: test-and-test-and-set is not a new instruction!
    - Just different software

Queue Locks

- Test-and-test-and-set locks can still perform poorly
  - If lock is contended for by many processors
  - Lock release by one processor, creates "free-for-all" by others
  - Interconnect gets swamped with `cas` requests
- **Software queue lock**
  - Each waiting processor spins on a different location (a queue)
  - When lock is released by one processor...
    - Only the next processors sees its location go "unlocked"
    - Others continue spinning locally, unaware lock was released
  - Effectively, passes lock from one processor to the next, in order
  + Greatly reduced network traffic (no mad rush for the lock)
  + Fairness (lock acquired in FIFO order)
  – Higher overhead in case of no contention (more instructions)
  – Poor performance if one thread is descheduled by O.S.

Programming With Locks Is Tricky

- Multicore processors are the way of the foreseeable future
  - thread-level parallelism anointed as parallelism model of choice
  - Just one problem…
- Writing lock-based multi-threaded programs is tricky!
- More precisely:
  - Writing programs that are correct is not easy
  - Writing programs that are highly parallel is not easy
  - **Writing programs that are correct and parallel is even harder**
    - And that's the whole point, unfortunately
  - Selecting the "right" kind of lock for performance
    - Spin lock, queue lock, ticket lock, read/writer lock, etc.
  - **Locking granularity issues**

Coarse-Grain Locks: Correct but Slow

- **Coarse-grain locks**: e.g., one lock for entire database
  - \+ Easy to make correct: no chance for unintended interference
  - – Limits parallelism: no two critical sections can proceed in parallel

```
struct acct_t { int bal; … };
shared struct acct_t  accts[MAX_ACCT];
shared Lock_t lock;
void debit(int id, int amt) {
  acquire(lock);
  if (accts[id].bal >= amt) {
     accts[id].bal -= amt;
  }
  release(lock);
}
```

Fine-Grain Locks: Parallel But Difficult

- **Fine-grain locks**: e.g., multiple locks, one per record
  - + Fast: critical sections (to different records) can proceed in parallel
  - – Easy to make mistakes
    - This particular example is easy
      - Requires only one lock per critical section

```
struct acct_t { int bal, Lock_t lock; …  };
shared struct acct_t  accts[MAX_ACCT];

void debit(int id, int amt) {
  acquire(accts[id].lock);
  if (accts[id].bal >= amt) {
     accts[id].bal -= amt;
  }
  release(accts[id].lock);
}
```

- What about critical sections that require two locks?

Multiple Locks

- **Multiple locks**: e.g., acct-to-acct transfer
  - Must acquire both **id_from**, **id_to** locks
  - Running example with accts 241 and 37
  - Simultaneous transfers 241 → 37 and 37 → 241
  - Contrived… but even contrived examples must work correctly too

```
struct acct_t { int bal, Lock_t lock; …};
shared struct acct_t  accts[MAX_ACCT];
void transfer(int id_from, int id_to, int amt) {
  acquire(accts[id_from].lock);
  acquire(accts[id_to].lock);
  if (accts[id_from].bal >= amt) {
     accts[id_from].bal -= amt;
     accts[id_to].bal += amt;
  }
  release(accts[id_to].lock);
  release(accts[id_from].lock);
}
```

Multiple Locks And Deadlock

Thread 0

```
id_from = 241;
id_to = 37;

acquire(accts[241].lock);
// wait to acquire lock 37
// waiting…
// still waiting…
```

Thread 1

```
id_from = 37;
id_to = 241;

acquire(accts[37].lock);
// wait to acquire lock 241
// waiting…
// …
```

- **Deadlock**: circular wait for shared resources
  - Thread 0 has lock 241 and waits for lock 37
  - Thread 1 has lock 37 and waits for lock 241
  - Obviously this is a problem
  - The solution is …

Coffman Conditions for Deadlock

- 4 necessary conditions
  - mutual exclusion
  - hold+wait
  - no preemption
  - circular waiting
- break any **one** of these conditions to get deadlock freedom

Correct Multiple Lock Program

- **Always acquire multiple locks in same order**
  - Yet another thing to keep in mind when programming

```
struct acct_t { int bal, Lock_t lock; … };
shared struct acct_t  accts[MAX_ACCT];
void transfer(int id_from, int id_to, int amt) {
  int id_first = min(id_from, id_to);
  int id_second = max(id_from, id_to);

  acquire(accts[id_first].lock);
  acquire(accts[id_second].lock);
  if (accts[id_from].bal >= amt) {
     accts[id_from].bal -= amt;
     accts[id_to].bal += amt;
  }
  release(accts[id_second].lock);
  release(accts[id_first].lock);
}
```

Correct Multiple Lock Execution

Thread 0

```
id_from = 241;
id_to = 37;
id_first = min(241,37)=37;
id_second = max(37,241)=241;

acquire(accts[37].lock);
acquire(accts[241].lock);
// do stuff
release(accts[241].lock);
release(accts[37].lock);
```

Thread 1

```
id_from = 37;
id_to = 241;
id_first = min(37,241)=37;
id_second = max(37,241)=241;

// wait to acquire lock 37
// waiting…
// …
// …
// …
acquire(accts[37].lock);
```

- Great, are we done? No

More Lock Madness

- What if…
  - Some actions (e.g., deposits, transfers) require 1 or 2 locks…
  - …and others (e.g., prepare statements) require all of them?
  - Can these proceed in parallel?
- What if…
  - There are locks for global variables (e.g., operation id counter)?
  - When should operations grab this lock?
- What if… what if… what if…

- **So lock-based programming is difficult…**
- **…wait, it gets worse**

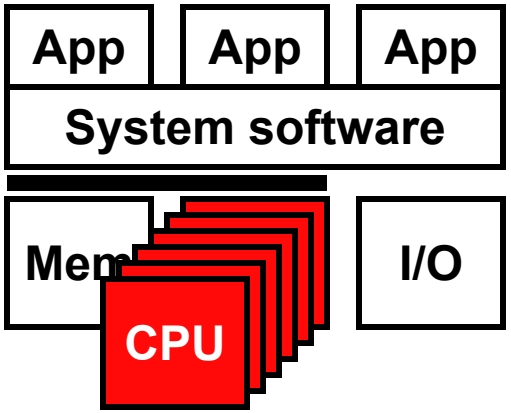
And To Make It Worse…

- **Acquiring locks is expensive…**
  - By definition requires slow atomic instructions
    - Specifically, acquiring write permissions to the lock
  - Ordering constraints (up next) make it even slower

- **…and 99% of the time un-necessary**
  - Most concurrent actions don't actually share data
  - – You pay to acquire the lock(s) for no reason

- Fixing these problem is an area of active research
  - One proposed solution "Transactional Memory"
  - Programmer uses construct: "atomic { … code … }"
    - Hardware, compiler & runtime executes the code "atomically"
    - Uses **speculation**, rolls back on conflicting accesses

Roadmap Checkpoint



- ~~Thread-level parallelism (TLP)~~
- ~~Shared memory model~~
  - ~~Multiplexed uniprocessor~~
  - ~~Hardware multihreading~~
  - ~~Multiprocessing~~
- ~~Cache coherence~~
  - ~~Valid/Invalid, MSI, MESI~~
- ~~Parallel programming~~
- ~~Synchronization~~
  - ~~Lock implementation~~
  - ~~Locking gotchas~~
  - ~~Transactional memory~~
- **Memory consistency models**

Shared Memory Example #1

- **Initially: all variables zero** (that is, x is 0, y is 0)

thread 1	thread 2
`store 1 → y` `load x`	`store 1 → x` `load y`

- What value pairs can be read by the two loads?

Shared Memory Example #1: "Answer"

- **Initially: all variables zero** (that is, x is 0, y is 0)

thread 1	thread 2
store 1 → y load x	store 1 → x load y

- What value pairs can be read by the two loads?

store 1 → y load x store 1 → x load y (x=0, y=1)	store 1 → y store 1 → x load x load y (x=1, y=1)	store 1 → y store 1 → x load y load x (x=1, y=1)
store 1 → x load y store 1 → y load x (x=1, y=0)	store 1 → x store 1 → y load y load x (x=1, y=1)	store 1 → x store 1 → y load x load y (x=1, y=1)

- What about (x=0, y=0)? Nope…*or can it?*

Shared Memory Example #2

- **Initially: all variables zero** ("flag" is 0, "a" is 0)

thread 1	thread 2
`store 1 → a` `store 1 → flag`	`loop: if (flag == 0) goto loop` `load a`

- What value can be read by "load a"?

Shared Memory Example #2: "Answer"

- **Initially: all variables zero** ("flag" is 0, "a" is 0)

thread 1	thread 2
`store 1 → a` `store 1 → flag`	`loop: if (flag == 0) goto loop` `load a`

- What value can be read by "load a"?

- Can "load a" read the value zero?
  - Unfortunately, yes.

What is Going On?

- Reordering of memory operations to different addresses!

- **In the hardware**
  1. To tolerate write latency
     - Cores don't wait for writes to complete (via store buffers)
     - And why should they? No reason to wait with single-thread code
  2. To simplify out-of-order execution

- **In the compiler**
  3. Compilers are generally allowed to re-order memory operations to **different addresses**
  - Many compiler optimizations reorder memory operations.

Memory Consistency

- **Cache coherence**
  - Creates globally uniform (consistent) view of **a single cache block**
  - Not enough on its own:
    - What about accesses to different cache blocks?
    - Some optimizations skip coherence(!)

- **Memory consistency model**
  - Specifies the semantics of shared memory operations
  - i.e., what value(s) a load may return

- Who cares? Programmers
  - Globally inconsistent memory creates mystifying behavior

3 Classes of Memory Consistency Models

- **Sequential consistency (SC)** (MIPS, PA-RISC)
  - **Typically what programmers expect**
  - 1. Processors see their own loads and stores in program order
  - 2. Processors see others' loads and stores in program order
  - 3. All processors see same global load/store ordering
  - Corresponds to some sequential interleaving of uniprocessor orders
- **Total Store Order (TSO)** (**x86**, SPARC)
  - Allows an in-order (FIFO) store buffer
    - Stores can be deferred, but must be put into the cache in order
- **Release consistency (RC)** (**ARM**, Itanium, **PowerPC**)
  - Allows an un-ordered coalescing store buffer
    - Stores can be put into cache in any order
  - Loads re-ordered, too.

Sequential Consistency

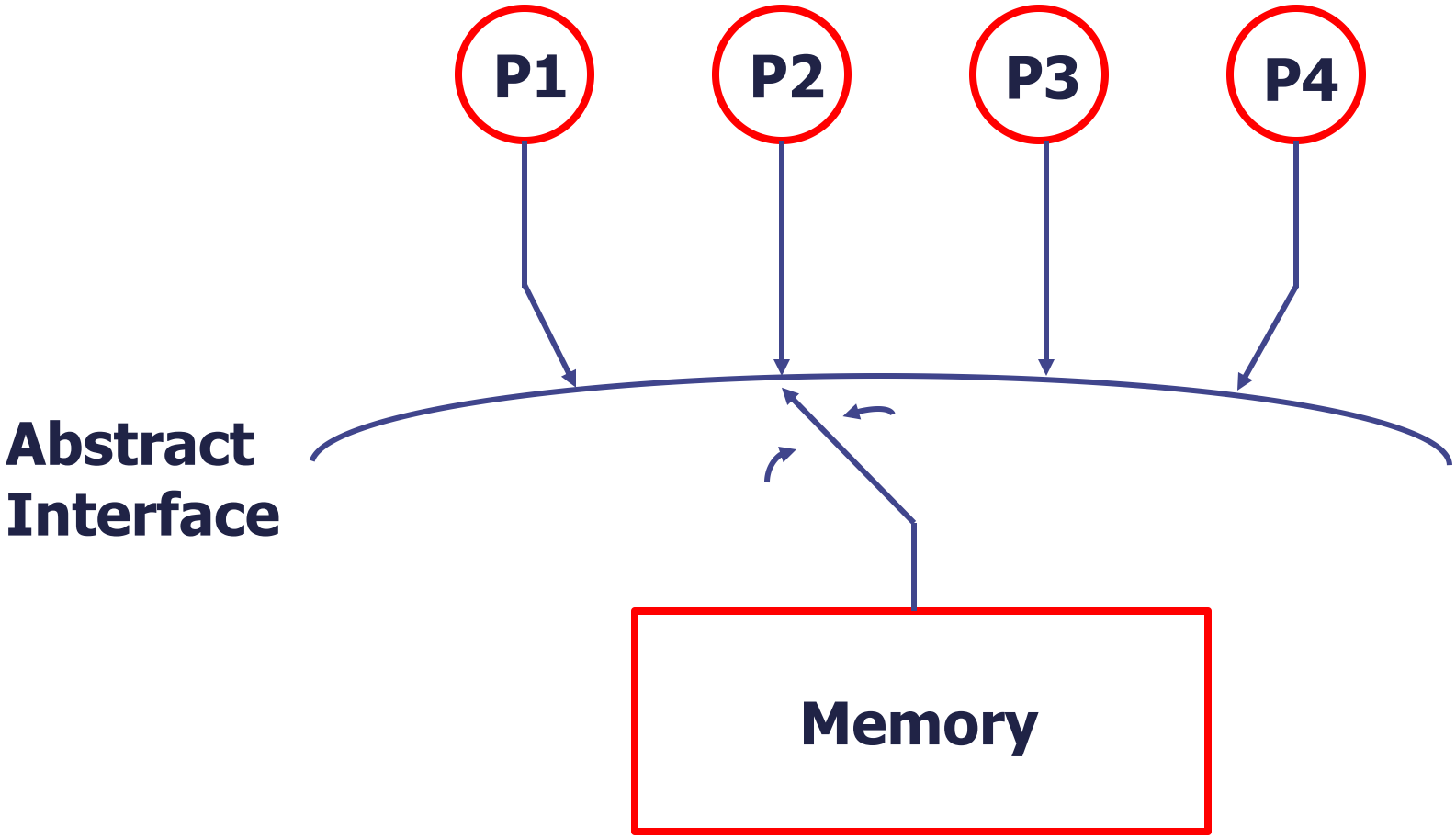
thread 1	thread 2
store 1 → y load x	store 1 → x load y

store 1 → y load x store 1 → x load y (x=0, y=1)	store 1 → y store 1 → x load x load y (x=1, y=1)	store 1 → y store 1 → x load y load x (x=1, y=1)
store 1 → x load y store 1 → y load x (x=1, y=0)	store 1 → x store 1 → y load y load x (x=1, y=1)	store 1 → x store 1 → y load x load y (x=1, y=1)

All the above cases can guarantee SC

Sequential Consistency

A Programmer's View



TSO (x86) Axiomatic Semantics

store and *load* can be reorderd in TSO

thread 1	thread 2
store 1 → y load x	store 1 → x load y

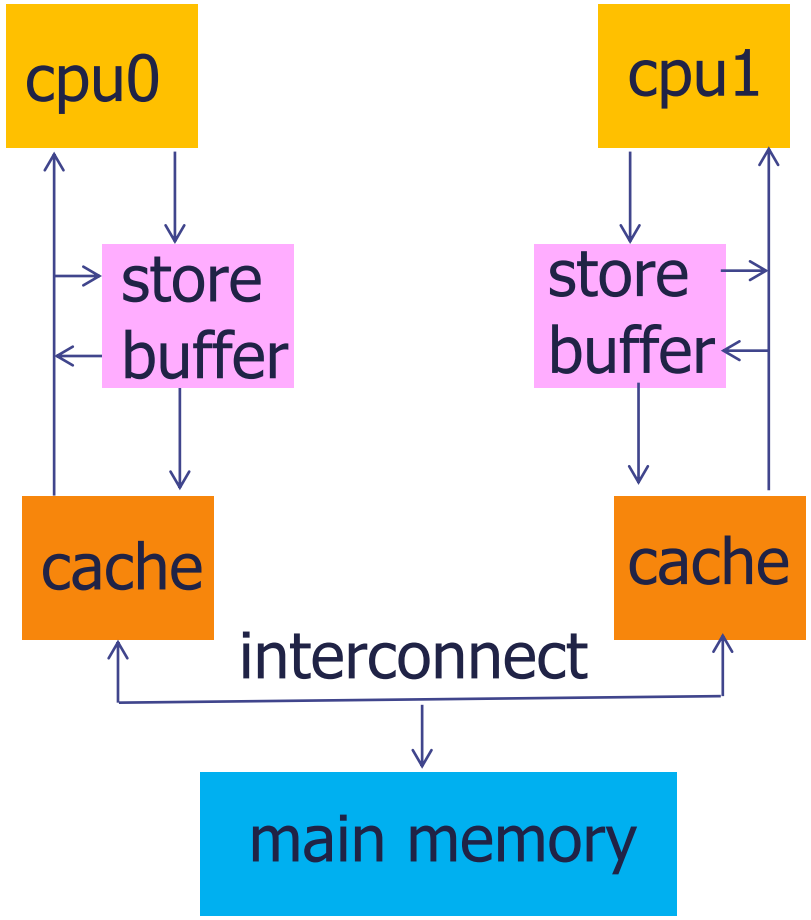
```
load x
load y
store 1 → x
store 1 → y
(x=0, y=0)
```

How to use the relaxation to achieve better execution speed? -- store buffer

TSO (x86) Axiomatic Semantics

For a *store* operation, CPU directly puts the data in the *store buffer*, and the *store buffer* writes the data to L1 Cache in the *FIFO* manner.

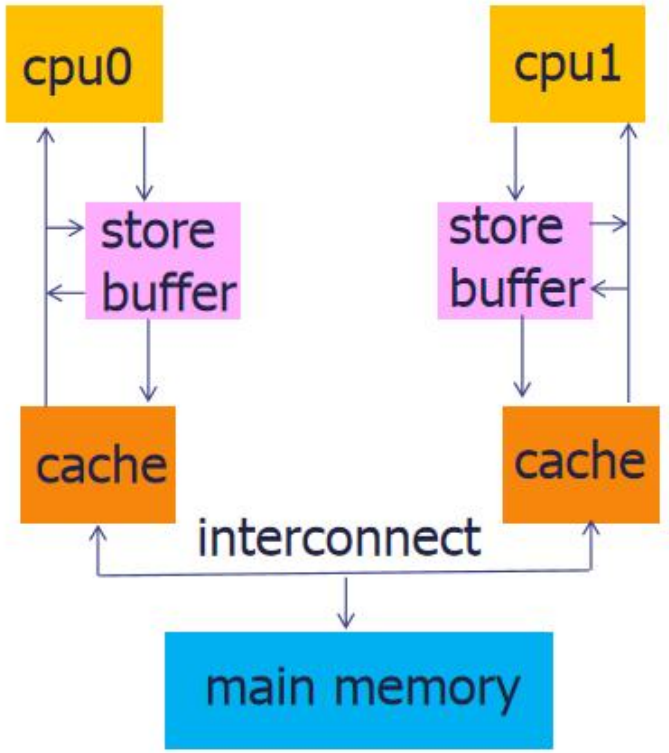
How this would affect application performance?



TSO (x86) Axiomatic Semantics

thread 1	thread 2
store 1 → y	store 1 → x
load x	load y

```
load x
load y
store 1 → x
store 1 → y
(x=0, y=0)
```



After execution, both x and y are 0, reorder between *load* and *store* happens for both thread1 and thread2

TSO (x86) Axiomatic Semantics

With TSO model, how the programmers guarantee SC at the software level?

thread 1	thread 2
store 1 → y FENCE load x	store 1 → x FENCE load y

FENCE is a memory barrier, which can ensure the inst after FENCE are not executed earlier than the inst before FENCE

```
store 1 → x
load y
store 1 → y
load x
(x=1, y=0)
```

```
store 1 → x
store 1 → y
load y
load x
(x=1, y=1)
```

```
store 1 → x
store 1 → y
load x
load y
(x=1, y=1)
```

TSO (x86) Axiomatic Semantics

With TSO model, how the programmers guarantee SC at the software level?

thread 1	thread 2
store 1 → y FENCE load x	store 1 → x FENCE load y

FENCE is a memory barrier, which can ensure the inst after FENCE are not executed earlier than the inst before FENCE

```
store 1 → y
load x
store 1 → x
load y
(x=0, y=1)
```

```
store 1 → y
store 1 → x
load x
load y
(x=1, y=1)
```

```
store 1 → y
store 1 → x
load y
load x
(x=1, y=1)
```

Shared Memory Example #1: Answer

- **Initially: all variables zero** (that is, x is 0, y is 0)

thread 1	thread 2
store 1 → y load x	store 1 → x load y

- What value pairs can be read by the two loads?

store 1 → y load x store 1 → x load y (x=0, y=1)	store 1 → y store 1 → x load x load y (x=1, y=1)	store 1 → y store 1 → x load y load x (x=1, y=1)
store 1 → x load y store 1 → y load x (x=1, y=0)	store 1 → x store 1 → y load y load x (x=1, y=1)	store 1 → x store 1 → y load x load y (x=1, y=1)

- What about (x=0,y=0)? Yes! (for x86, SPARC, ARM, PowerPC)

Release Consistency Axiomatic Semantics

Two synchronization operations: acquire and release

1) ACQUIRE: for all the other participants, the load/store after ACQUIRE must happen later than ACQUIRE

2) RELEASE: for all the other parcipants, the load/store before RELEASE must happen earlier than RELEASE

RELEASE and ACQUIRE form a memory barrier

Release Consistency Axiomatic Semantics

CPU-A:

```
a.store(3);
b.store(4);
m.store(5, release);
```

CPU-B

```
m.load(acquire);
g.load();
h.load();
```

When `m.store(5, release)` completes, `a.store` and `b.store` must have completed，while the order of `a.store` and `b.store` can be arbitrary

When `g.load` and `h.load` completes，`m.load` must have completed, while the order of `g.load` and `h.load` can be arbitrary

Shared Memory Example #2: Answer

- **Initially: all variables zero** (flag == a == 0)

thread 1	thread 2
`store 1 → a` `store 1 → flag`	`loop: if (flag == 0) goto loop` `load a`

- What value can be read by "load a"?
  - "load a" can see the value "1"
- Can "load a" read the value zero?
  - **Yes! (for ARM, PowerPC, Itanium, and Alpha)**
  - **No! (for Intel/AMD x86, Sun SPARC, IBM 370)**
    - Assuming the compiler didn't reorder anything…

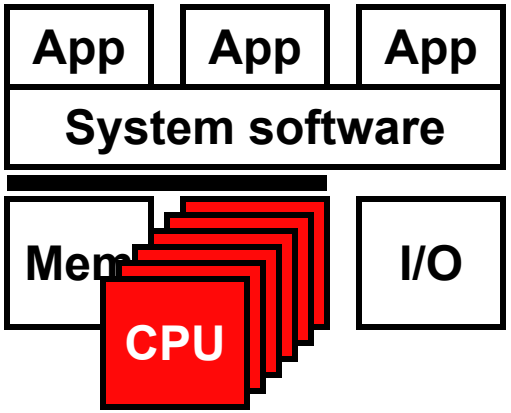
SC for DRF programs

- If a program is **data-race-free**, all consistency models guarantee sequentially-consistent behavior
  - hw/compiler still reorder operations
  - but they promise that you won't notice!
- a **data race** consists of:
  - two memory accesses
  - from different threads
  - to the same byte(s)
  - where at least one access is a write
  - without synchronization
- What if we do have a data race?
  - C/C++: anything can happen (just as with buffer overflows)
  - Java: weird reorderings, but no out-of-thin-air reads

Recap: Four Shared Memory Issues

1. **Cache coherence**
   - If cores have private (non-shared) caches
   - How to make writes to one cache "show up" in others?
2. **Parallel programming**
   - How does the programmer express the parallelism?
3. **Synchronization**
   - How to regulate access to shared data?
   - How to implement "locks"?
4. **Memory consistency models**
   - How to keep programmer sane while letting hw/compiler optimize?

Summary



- Thread-level parallelism (TLP)
- Shared memory model
  - Multiplexed uniprocessor
  - Hardware multihreading
  - Multiprocessing
- Cache coherence
  - Valid/Invalid, MSI, MESI
- Synchronization
  - Lock implementation
  - Locking gotchas
  - Transactional memory
- Memory consistency models