CSE 548: Computer Systems Architecture

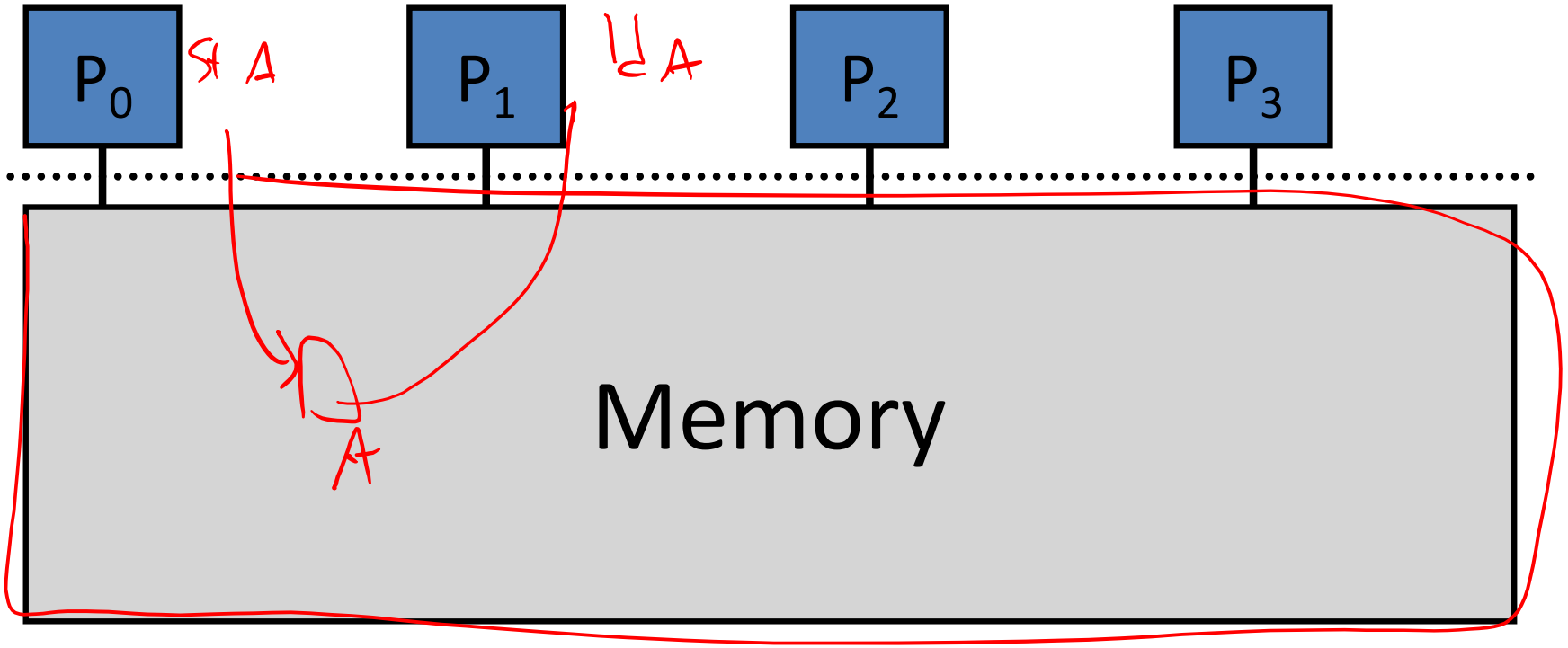
Cache Coherence

Luis Ceze, Spring 2017

(based on slides lifted from friends at UPenn, UIUC, UW, MIT, CMU)

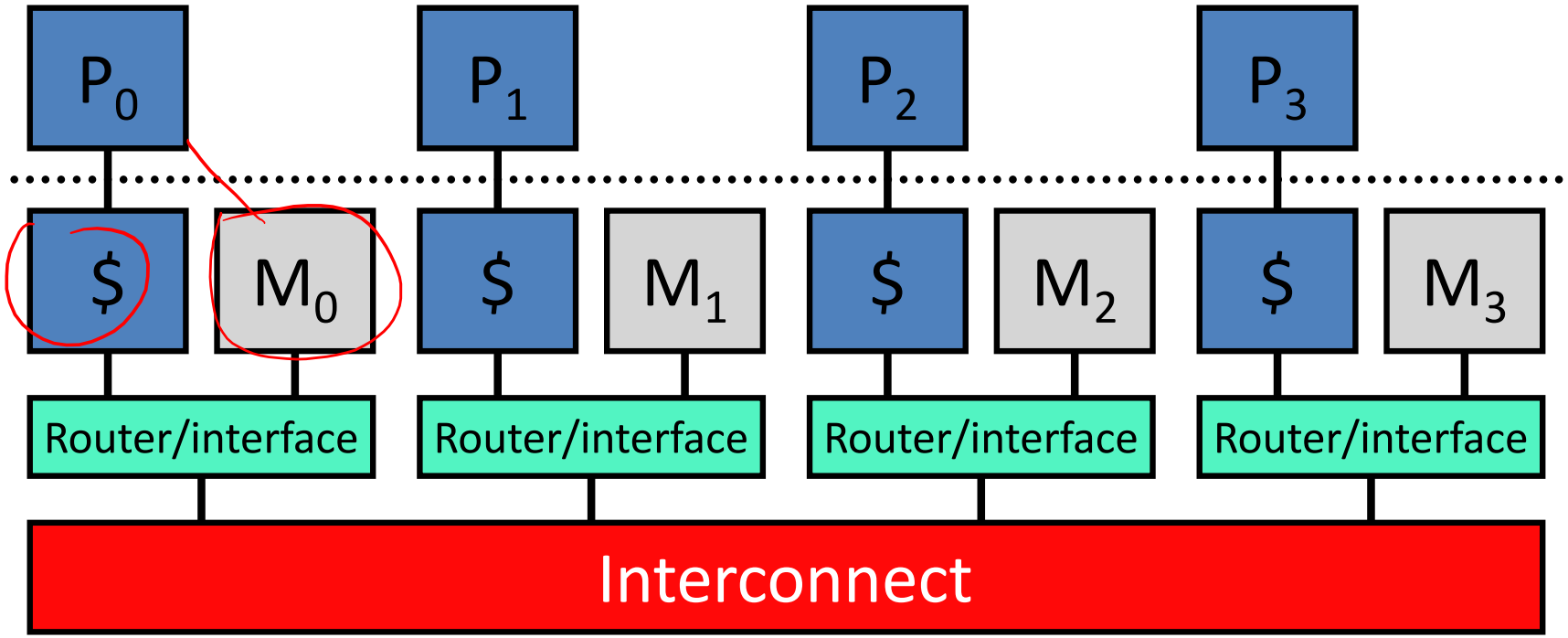
Shared-Memory Multiprocessors

- **Conceptual model**
  - The shared-memory abstraction
  - Familiar and feels natural to programmers
  - Life would be easy if systems actually looked like this…



Shared-Memory Multiprocessors

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



Problems with the Intuition

- Intuition:
  - Reading a shared location should return latest value written (by any process)
- But "last" is not well-defined
- In sequential case:
  - "last" is defined in terms of program order, not time
- In parallel case?

Problems with the Intuition

- Intuition:
  - Reading a shared location should return latest value written (by any process)
- But "last" is not well-defined
- In sequential case:
  - "last" is defined in terms of program order, not time
- In parallel case:
  - program order defined within a thread, but need to make sense of orders across thread
- Must define a meaningful semantics
  - the answer involves both "cache coherence" and an appropriate "memory consistency model"

Formal Definition of Coherence

- A memory system is *coherent* if the results of any execution of a program are such that for each location, it is possible to construct a hypothetical serial order of all operations ***to the location*** that is consistent with the results of the execution and in which:
- 1. operations issued by any particular process occur in the order issued by that thread, and
- 2. the value returned by a read is the value written by the last write to that location in the serial order
- Two necessary conditions:
  - *Write propagation*: value written must become visible to others eventually
  - *Write serialization*: writes to location seen in same order by all
    - if I see w1 after w2, you should not see w2 before w1
    - do we need to worry about read order?

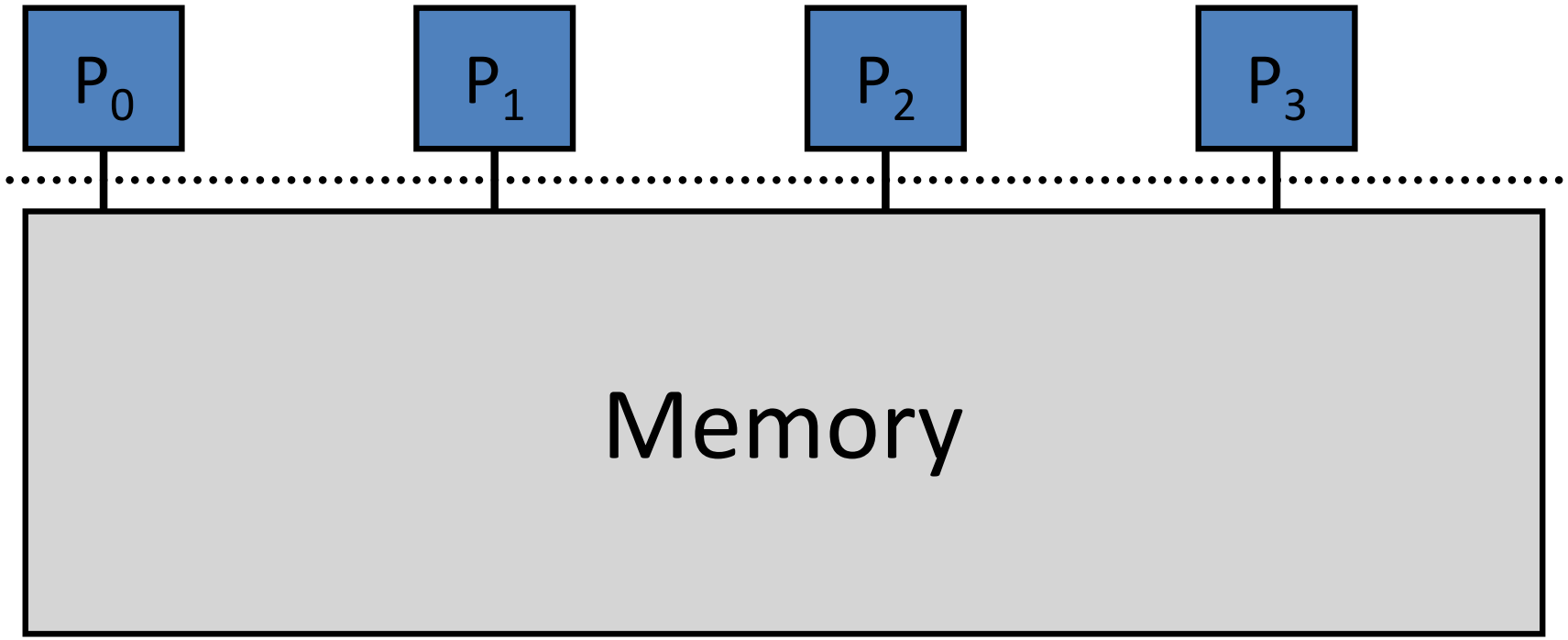
How do we provide these guarantees?

Announcements

- Next lecture in room 305
  - HW specialization/machine learning
  - Joint with deep learning class

Shared-Memory Multiprocessors

- **Conceptual model**
  - The shared-memory abstraction
  - Familiar and feels natural to programmers
  - Life would be easy if systems actually looked like this…

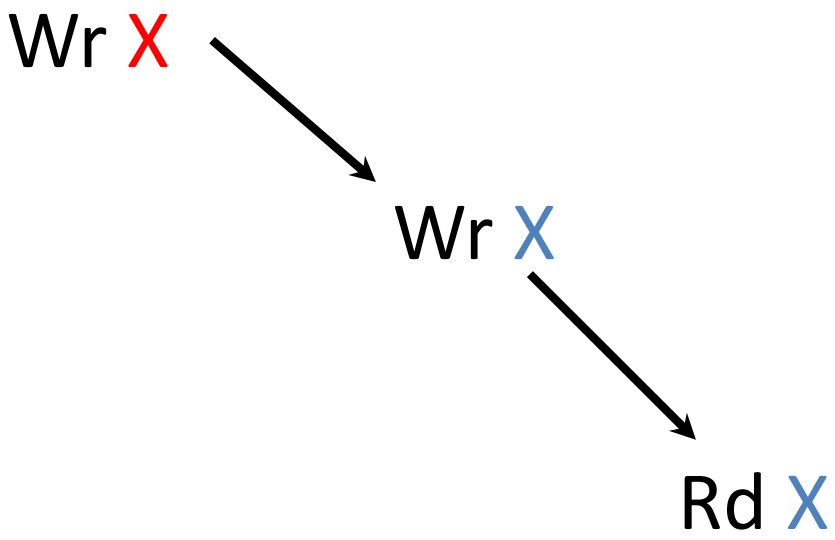


Formal Definition of Coherence

- A memory system is *coherent* if the results of any execution of a program are such that for each location, it is possible to construct a hypothetical serial order of all operations ***to the location*** that is consistent with the results of the execution and in which:
- 1. operations issued by any particular process occur in the order issued by that thread, and
- 2. the value returned by a read is the value written by the last write to that location in the serial order
- Two necessary conditions:
  - *Write propagation*: value written must become visible to others eventually
  - *Write serialization*: writes to location seen in same order by all
    - if I see w1 after w2, you should not see w2 before w1
    - do we need to worry about read order?

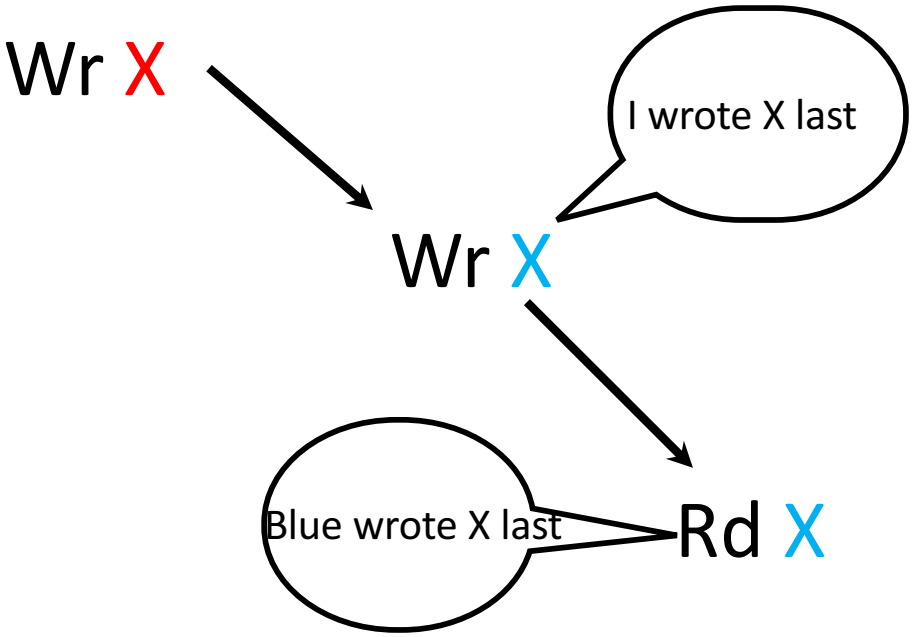
How do we provide these guarantees?

Coherence



"Reading X gets the value of the last write to X"

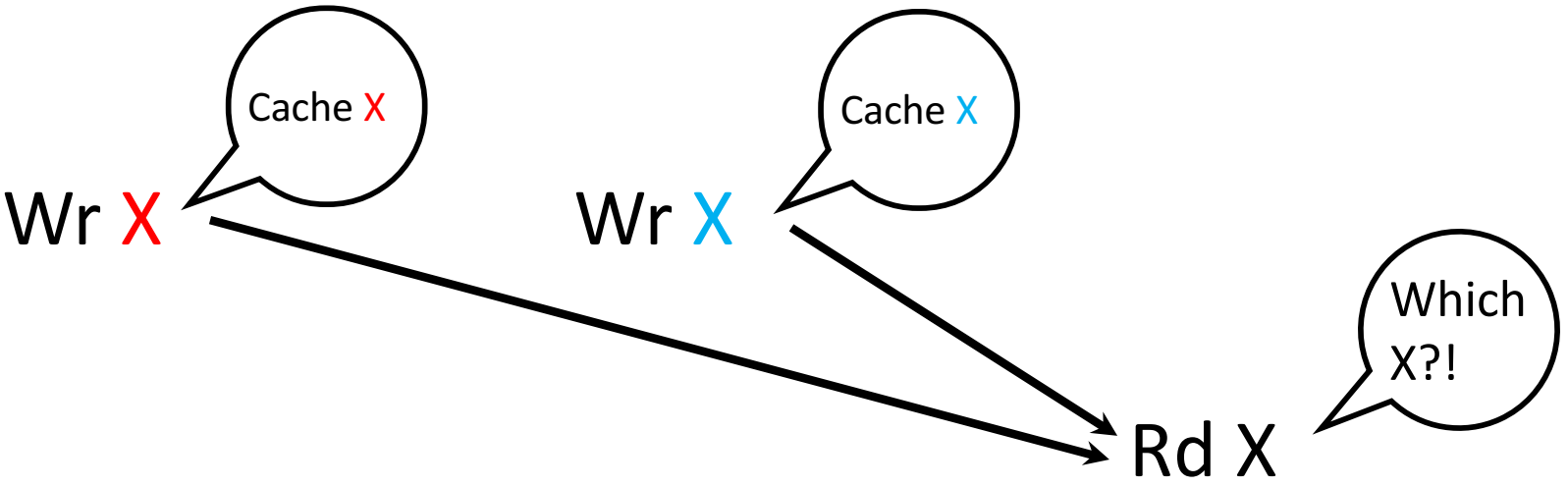
Coherence



2) “Reading X gets the value of the last write to X”

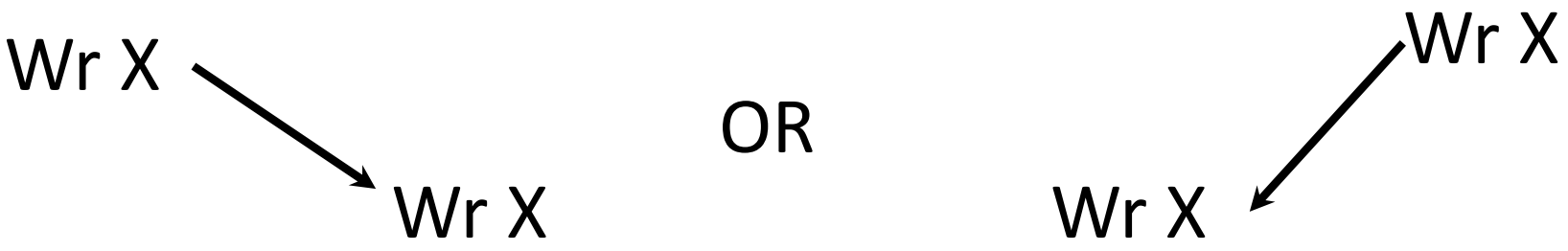
Without Coherence

(The coherence invariants prevent this from happening)



How to decide who wrote last?

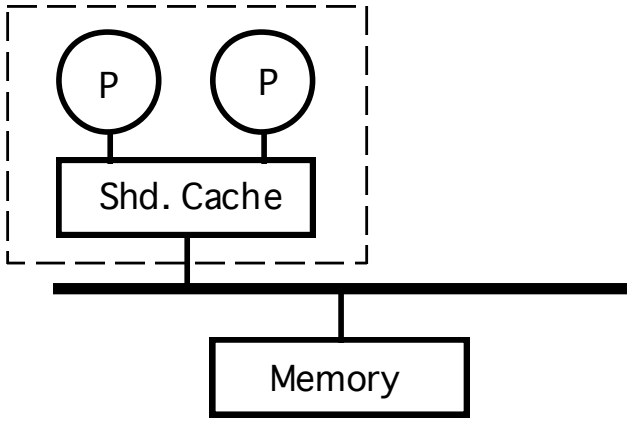
Coherence is Ordering

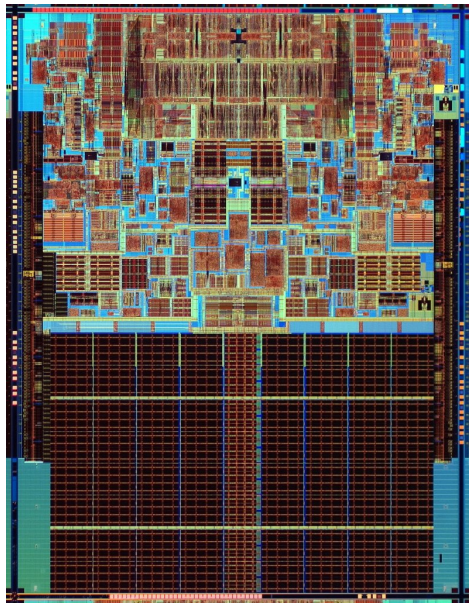


Coherence defines the set of legal orders of accesses to a single memory location

Shared Caches

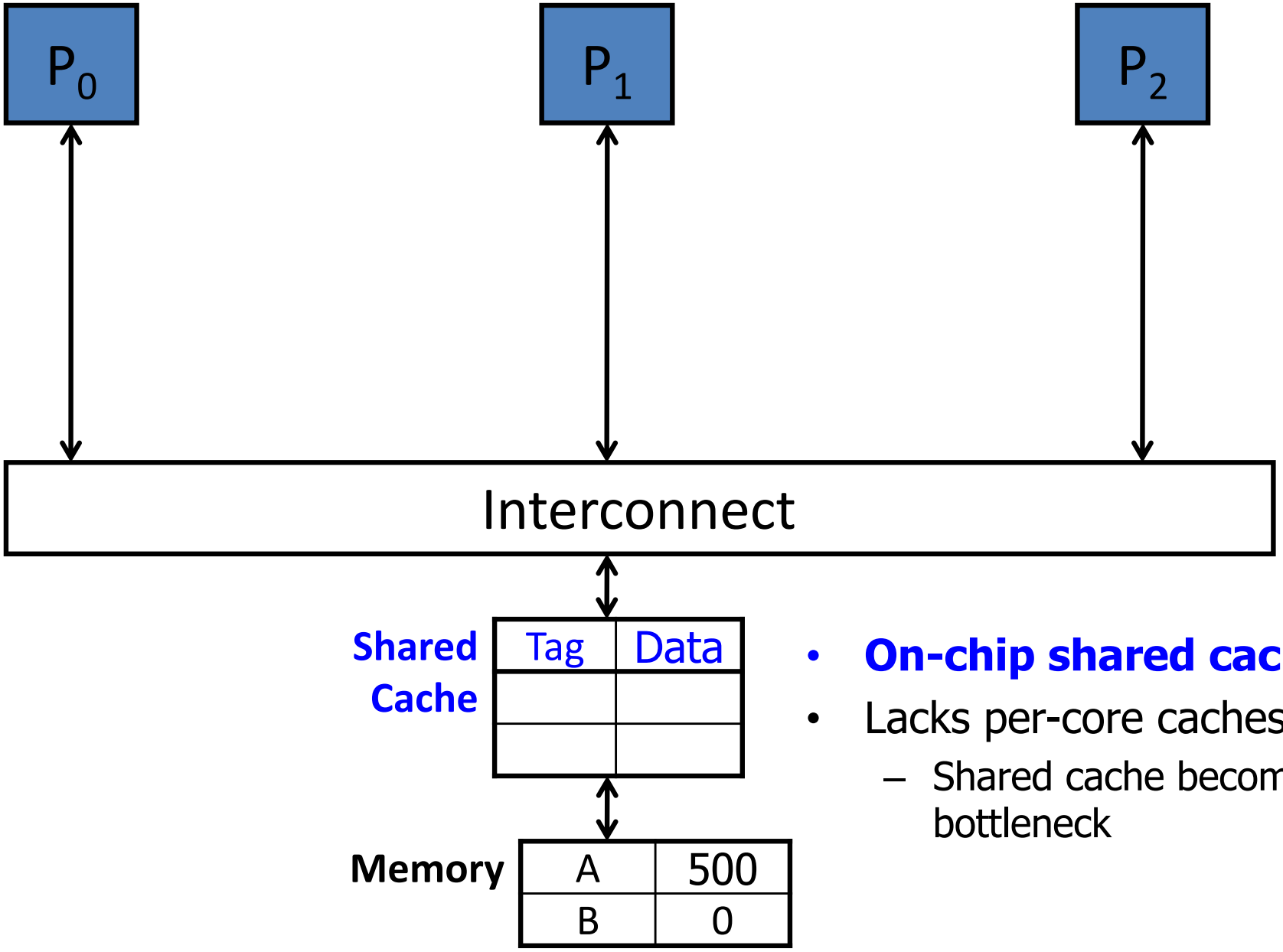
- Processors share a single cache, essentially punting the problem.
- Useful for very small machines.
  - Problems are limited cache bandwidth and cache interference
  - Benefits are fine-grain sharing and prefetch effects





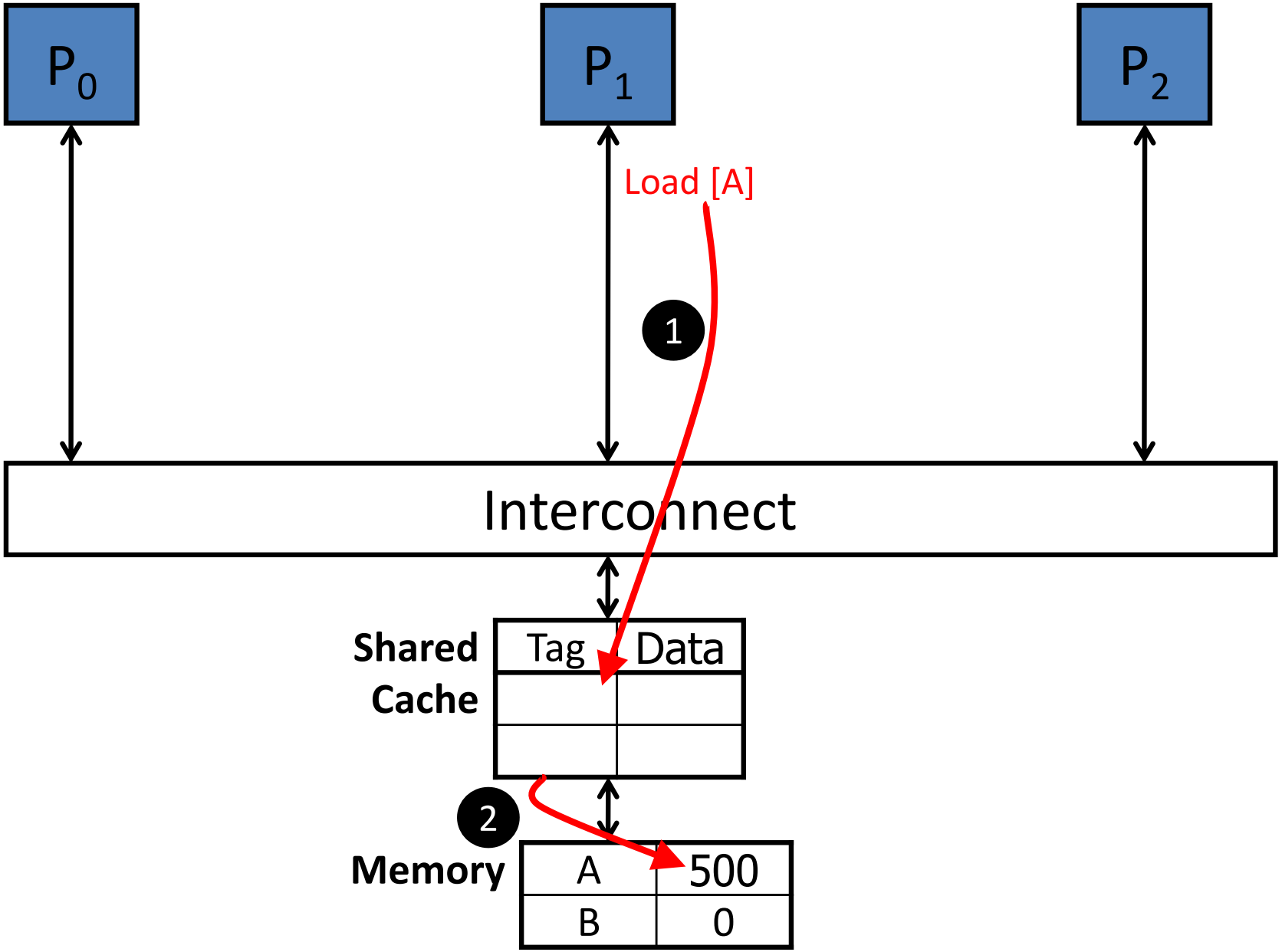
Intel Core 2 Duo (Conroe)

Shared Cache Implementation

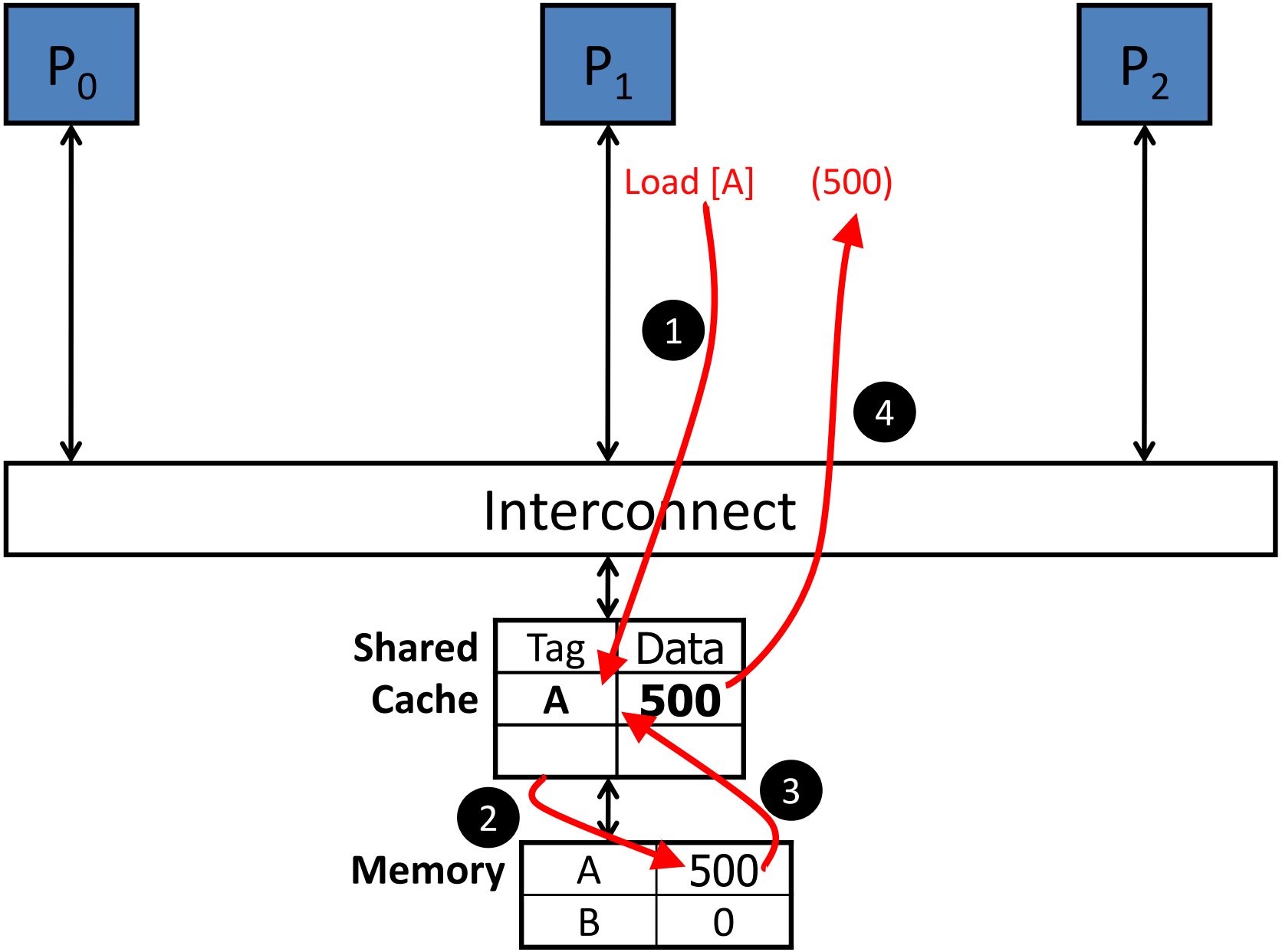


- **On-chip shared cache**
- Lacks per-core caches
  - Shared cache becomes bottleneck

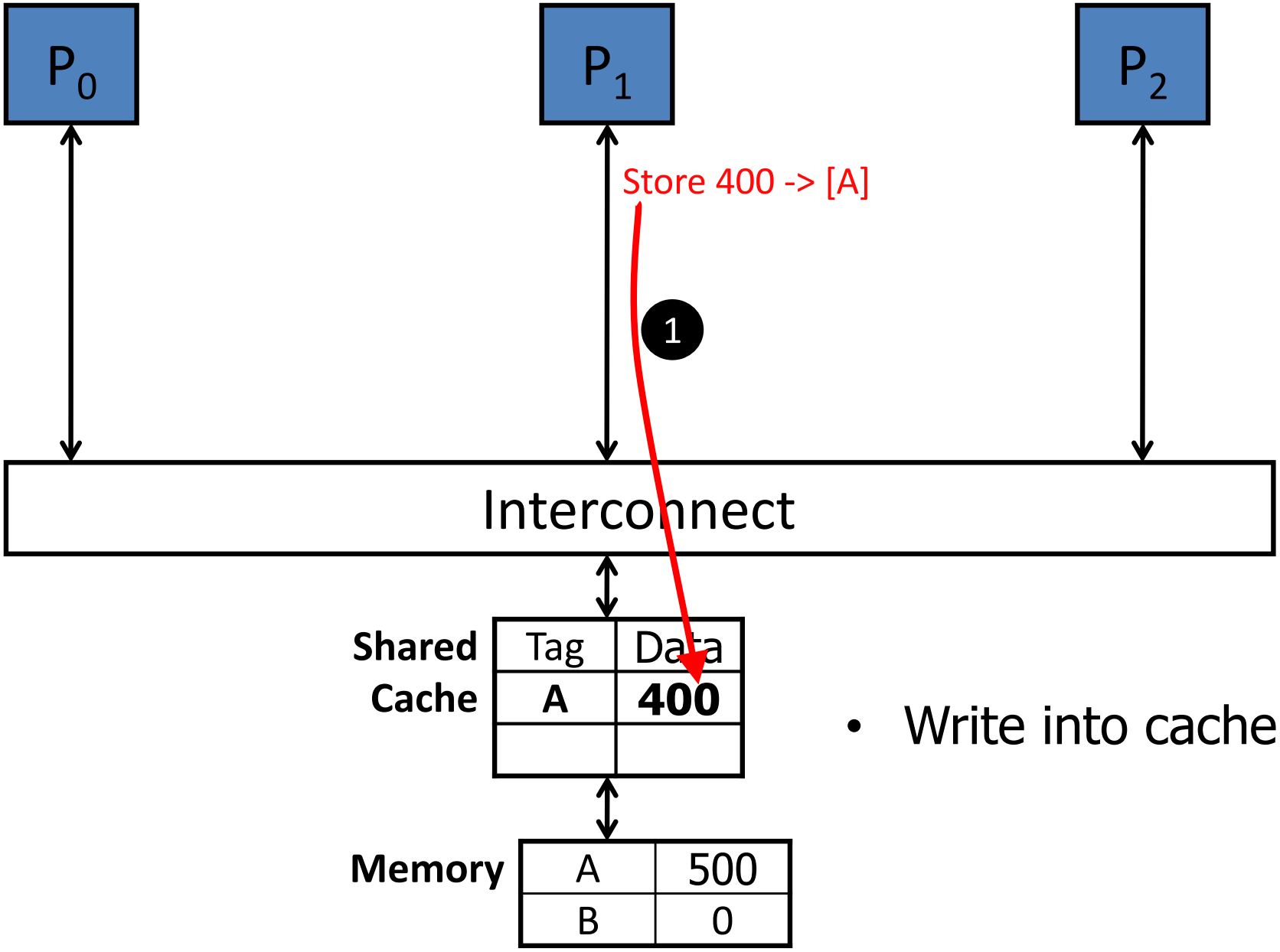
Shared Cache Implementation



Shared Cache Implementation

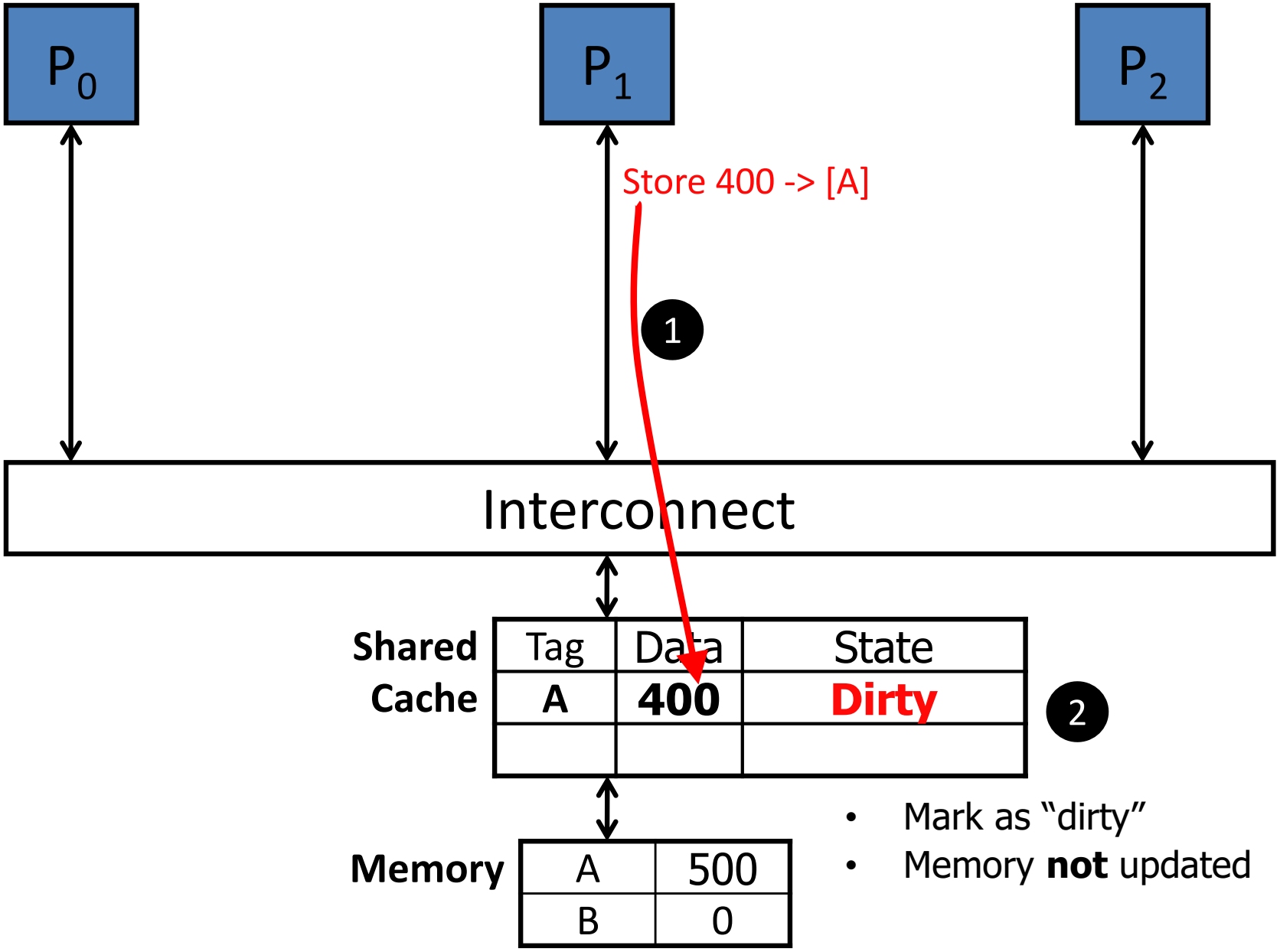


Shared Cache Implementation

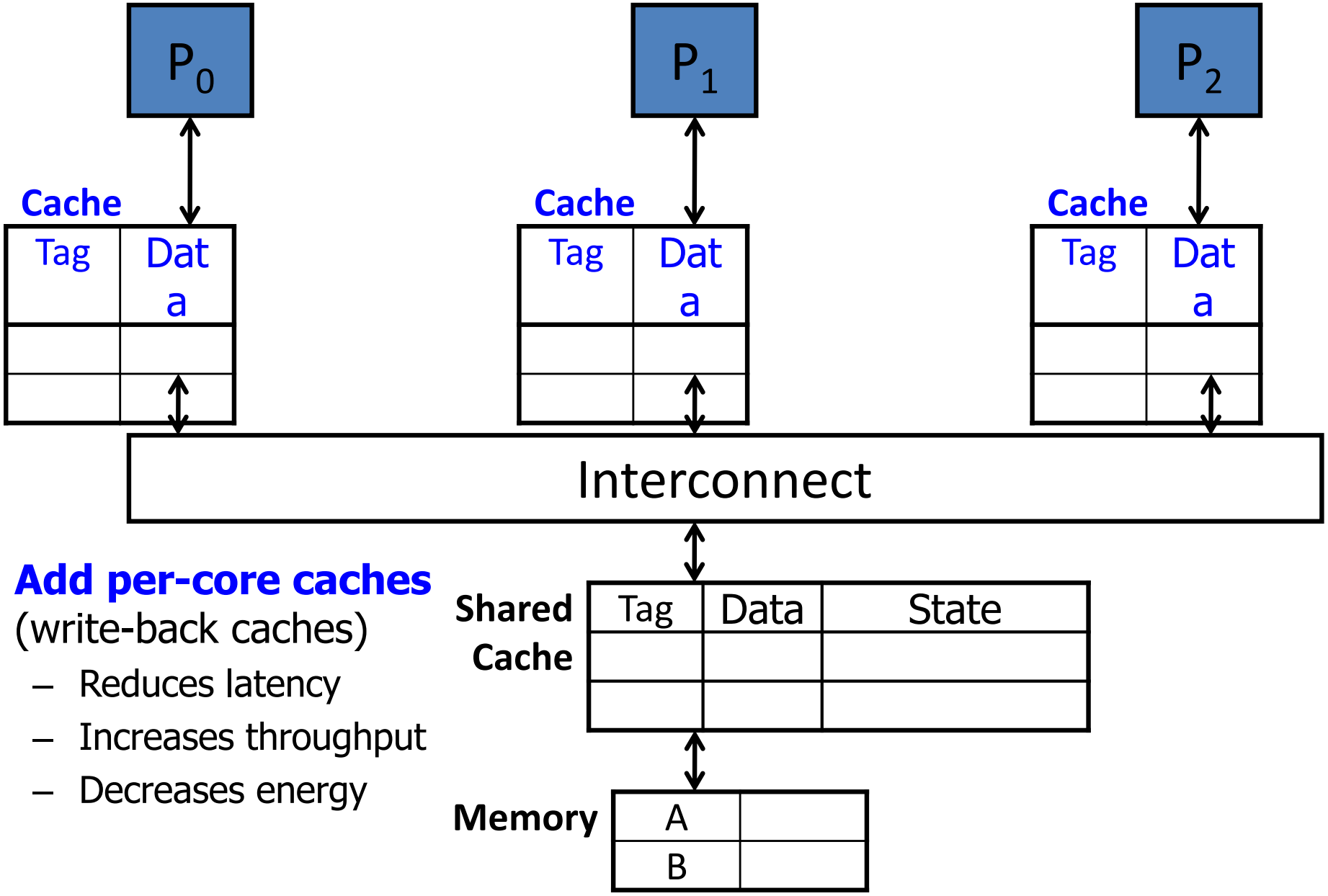


- Write into cache

Shared Cache Implementation

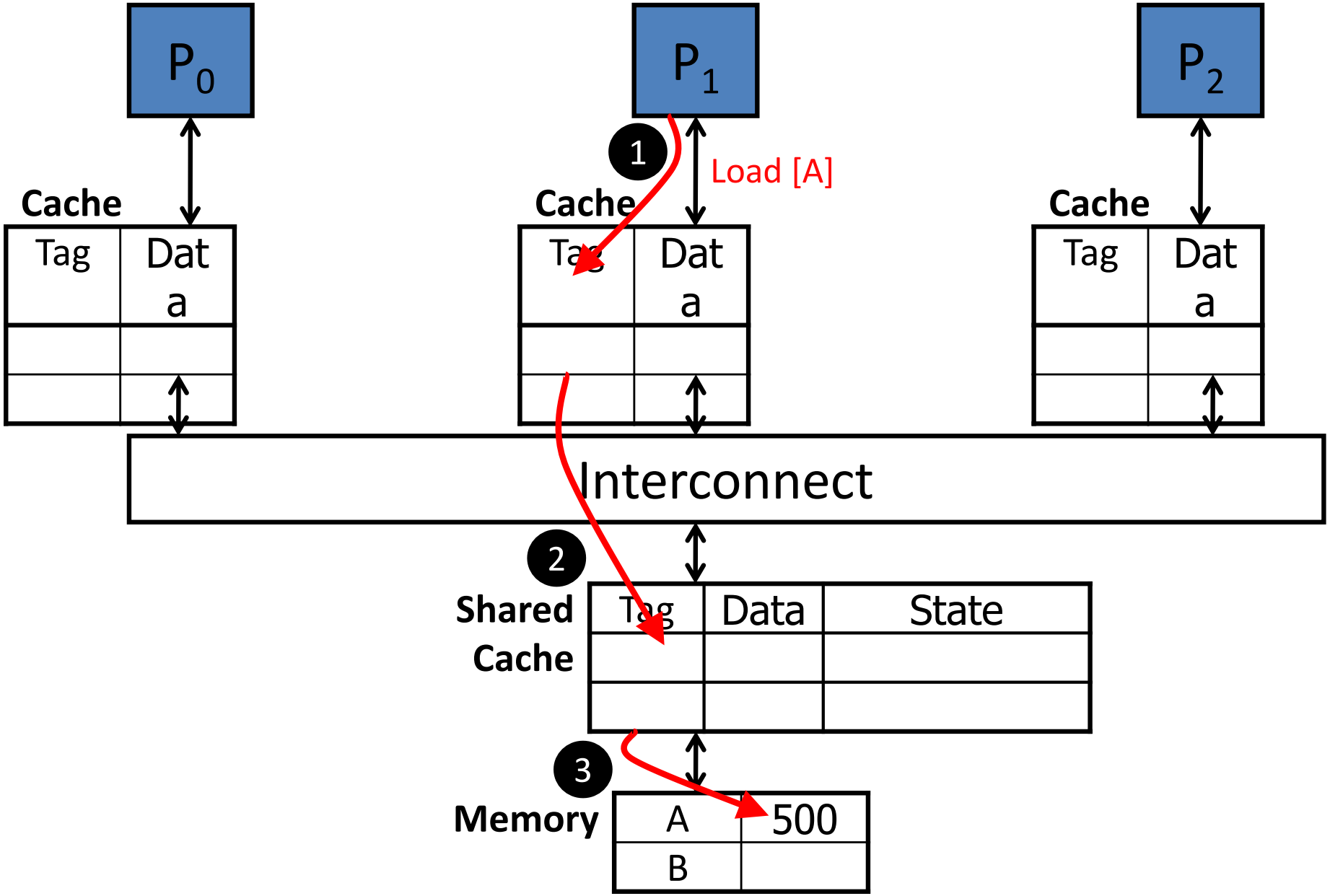


Adding Private Caches

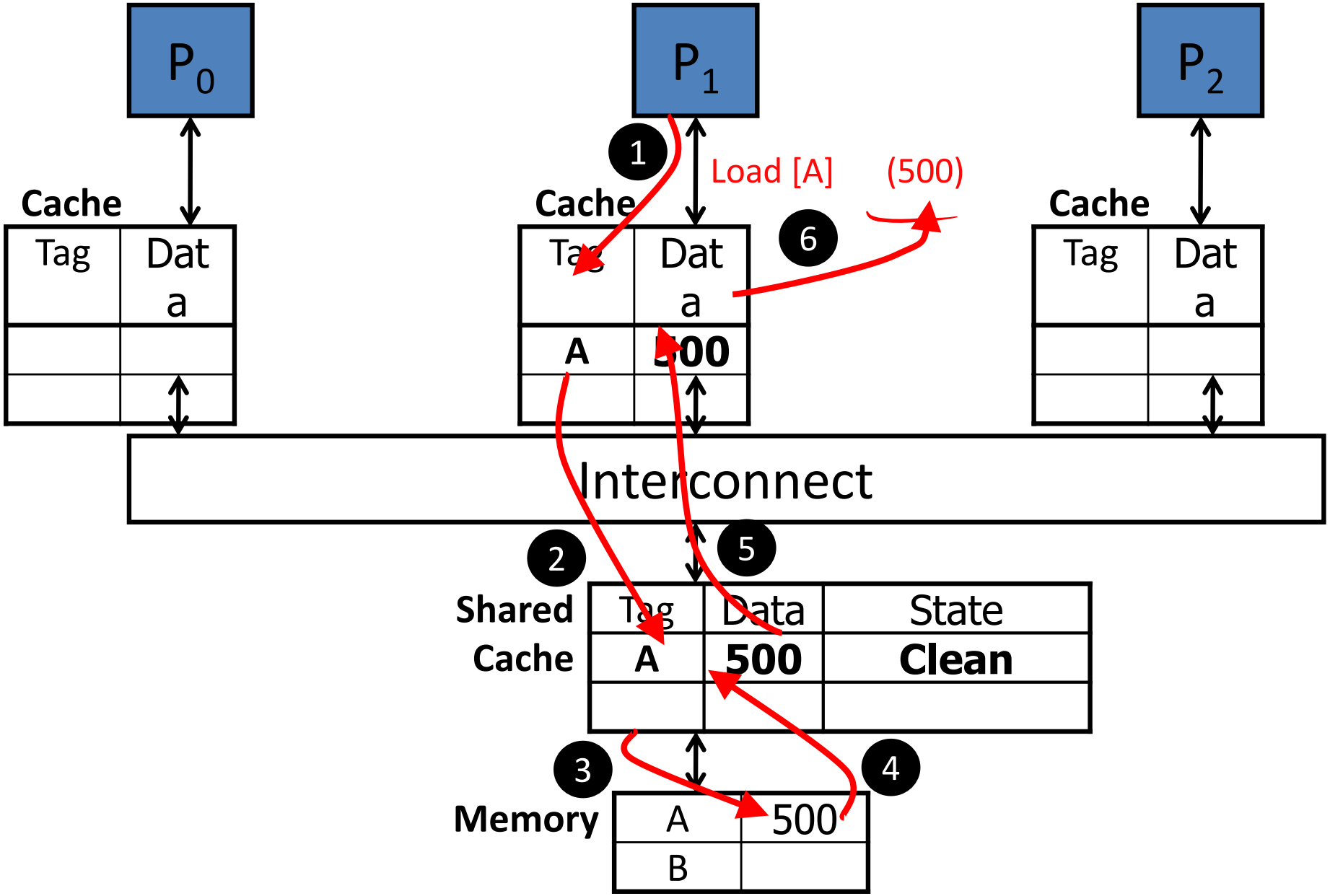


- **Add per-core caches** (write-back caches)
  - Reduces latency
  - Increases throughput
  - Decreases energy

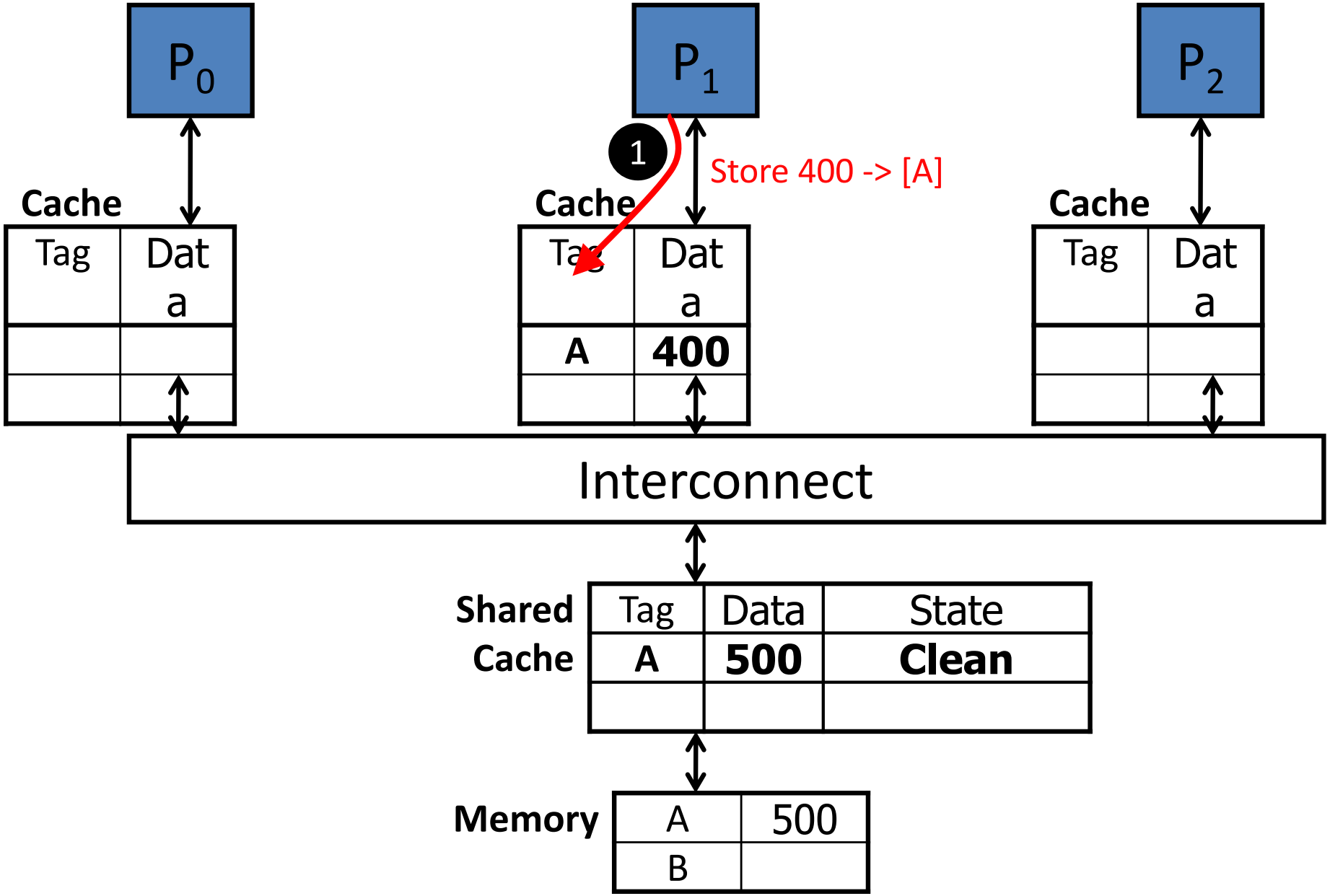
Adding Private Caches



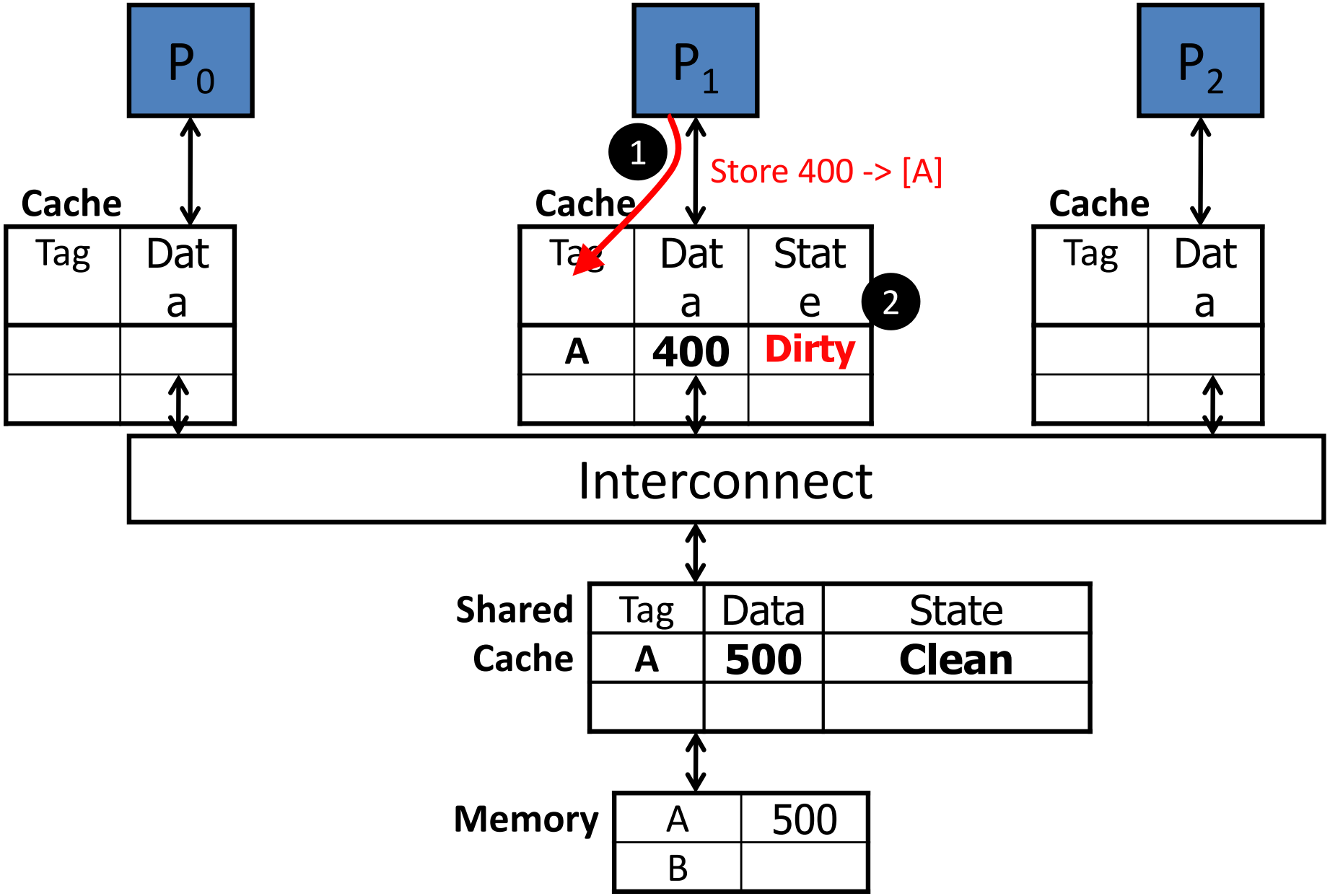
Adding Private Caches



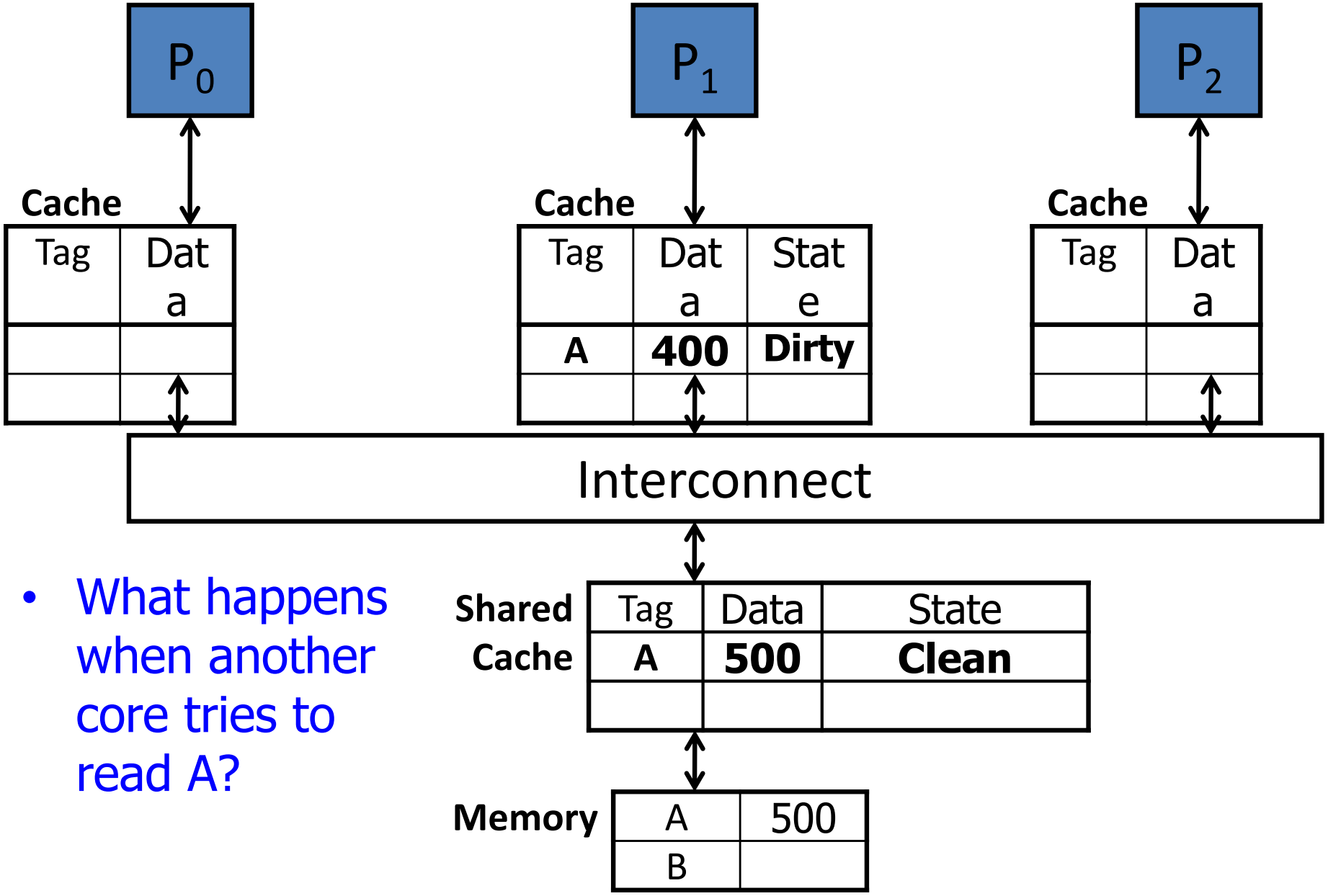
Adding Private Caches



Adding Private Caches

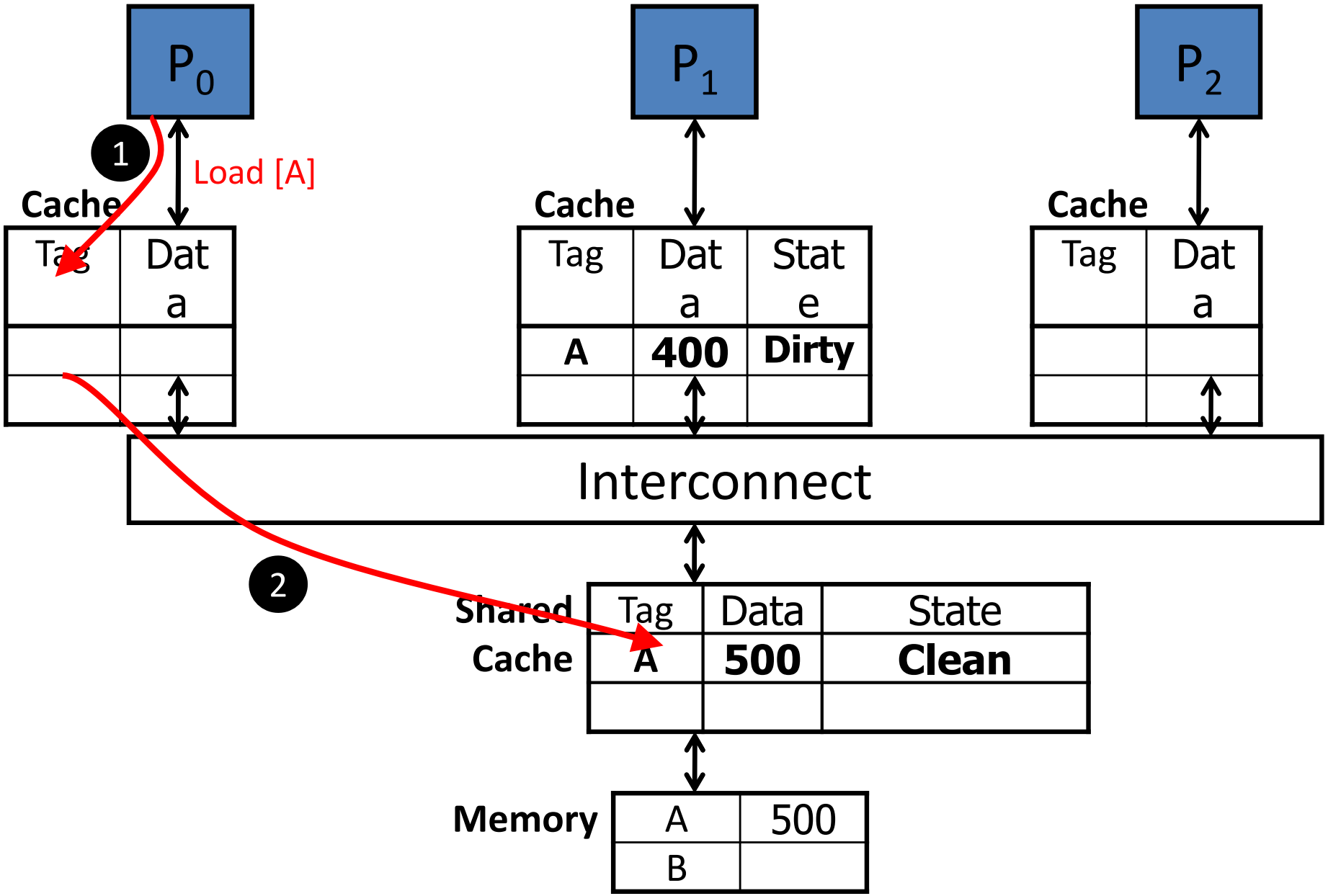


Will It Always Work?

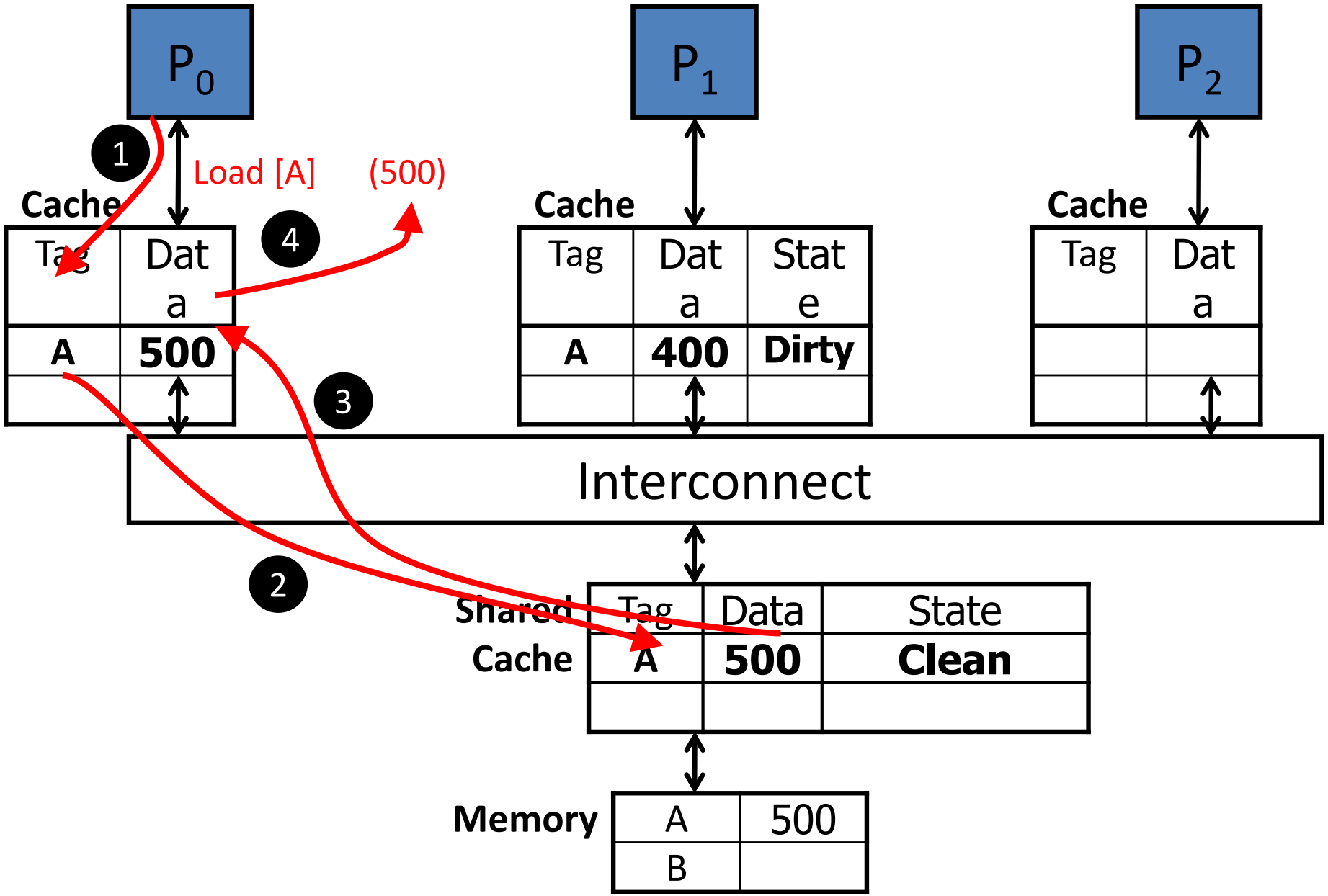


- What happens when another core tries to read A?

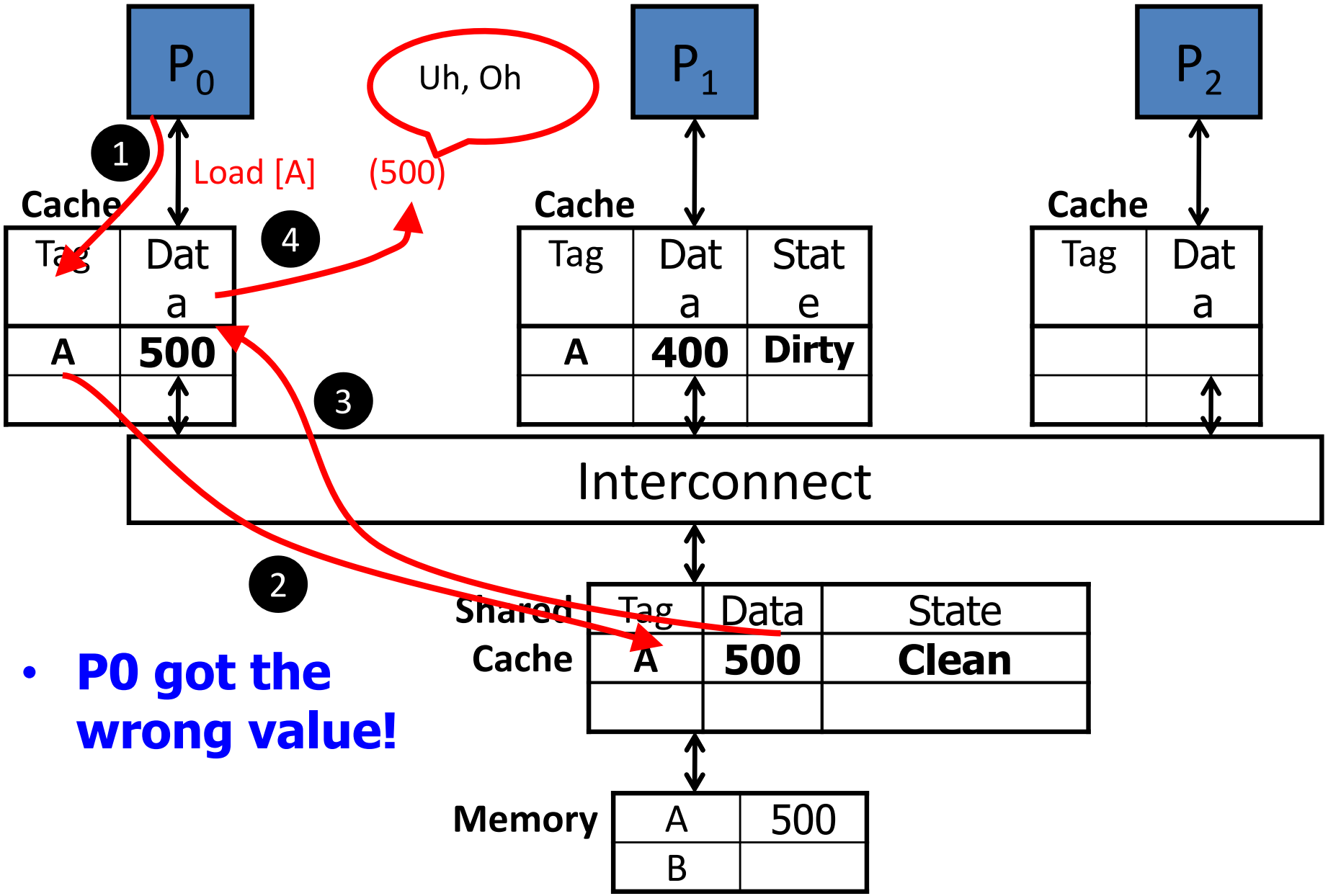
Private Cache Problem: Incoherence



Private Cache Problem: Incoherence



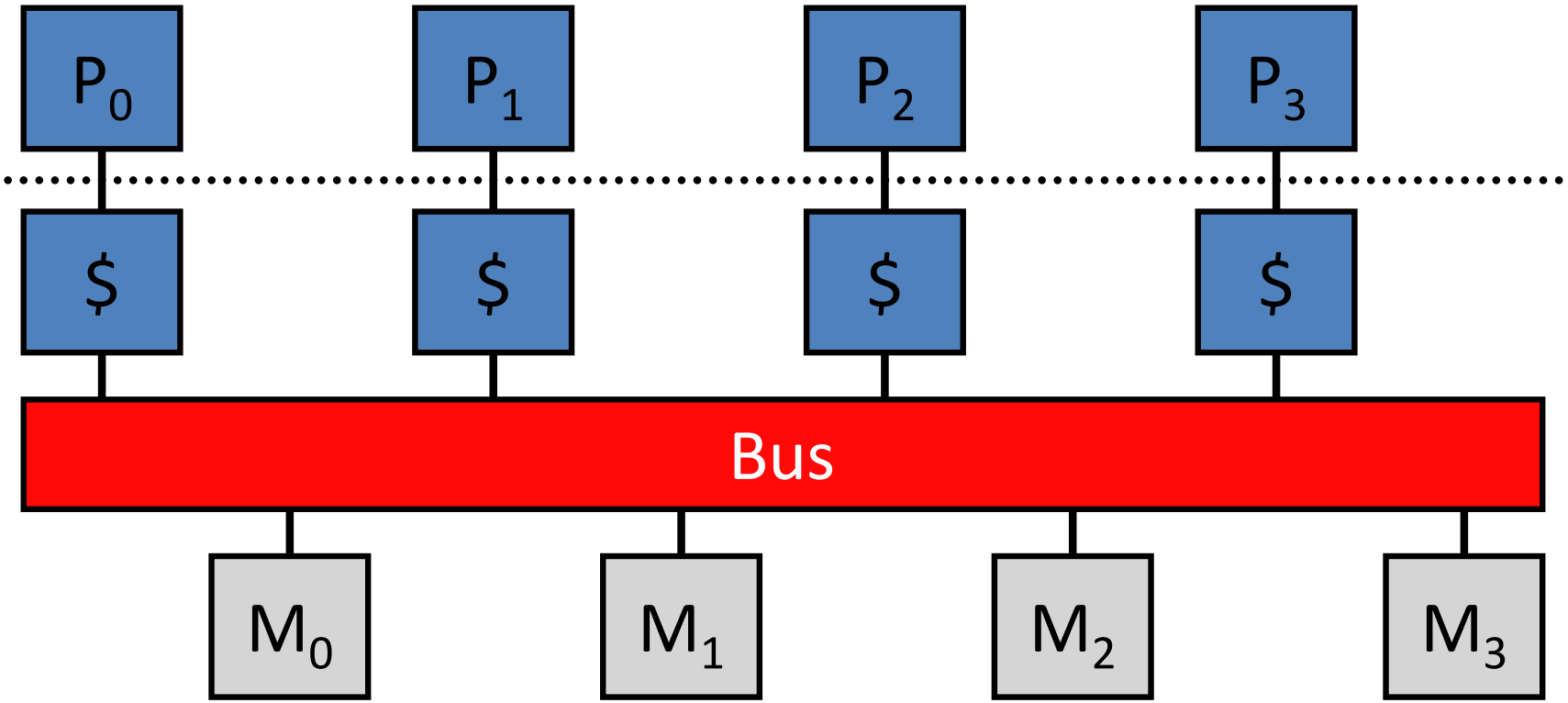
Private Cache Problem: Incoherence



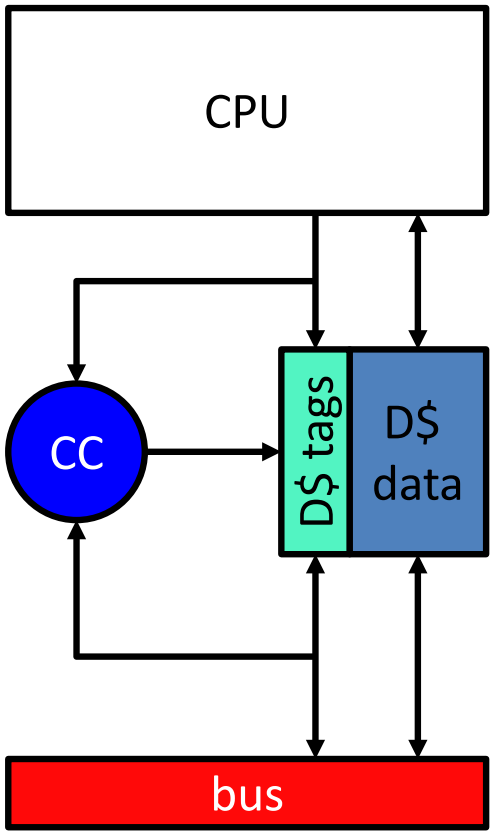
- **P0 got the wrong value!**

Bus-based Multiprocessor

- Simple multiprocessors use a bus
  - **All** processors see all requests at the **same time**, same order
- Memory
  - Single memory module, **-or-**
  - Banked memory module

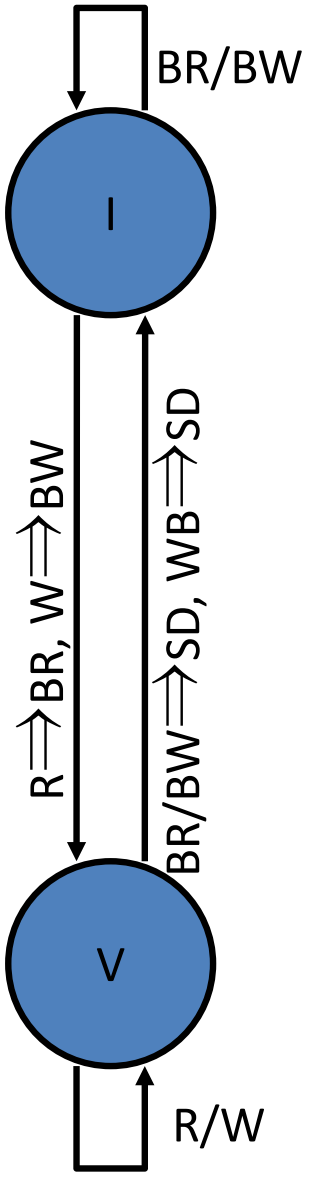


Snoopy Hardware Cache Coherence



- **Coherence controller**:
  - Examines bus traffic (addresses and data)
  - Executes **coherence protocol**
    - What to do with local copy when you see different things happening on bus
- Protocol is a distributed algorithm: cooperating state machines
  - Set of states, state transition diagram, actions
- Granularity of coherence is a cache block
- Bus messages totally ordered and atomic (unless split-transaction)
- *What should the protocol (s) look like?*

VI (MI) 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)
  - Convention: event⇒generated-event
  - 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**

Three processor-initiated events
R: read **W**: write **WB**: write-back
One response event: **SD:** send data
Two remote-initiated events
BR: bus-read, read miss from ***another*** processor
BW: bus-write, write miss from ***another*** processor

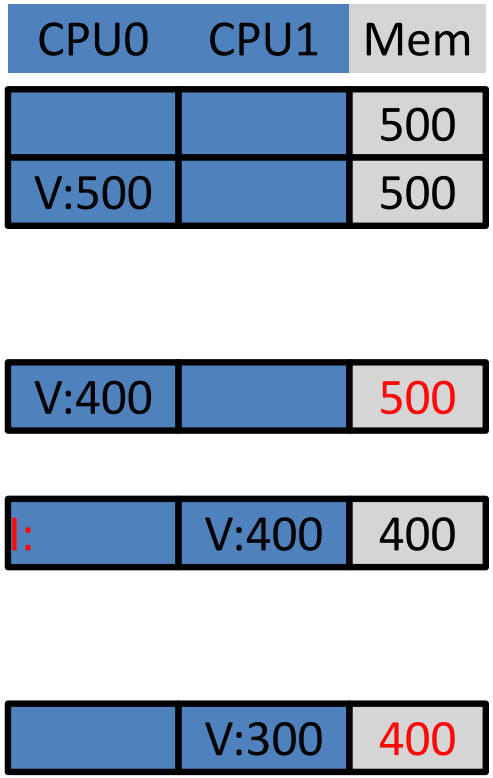
VI Protocol State Transition Table

State	This Processor		Other Processor	
	Load	Store	Load Miss	Store Miss
Invalid (I)	Miss => V	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
  - Responds with no other processor would respond

VI Protocol (Write-Back Cache)

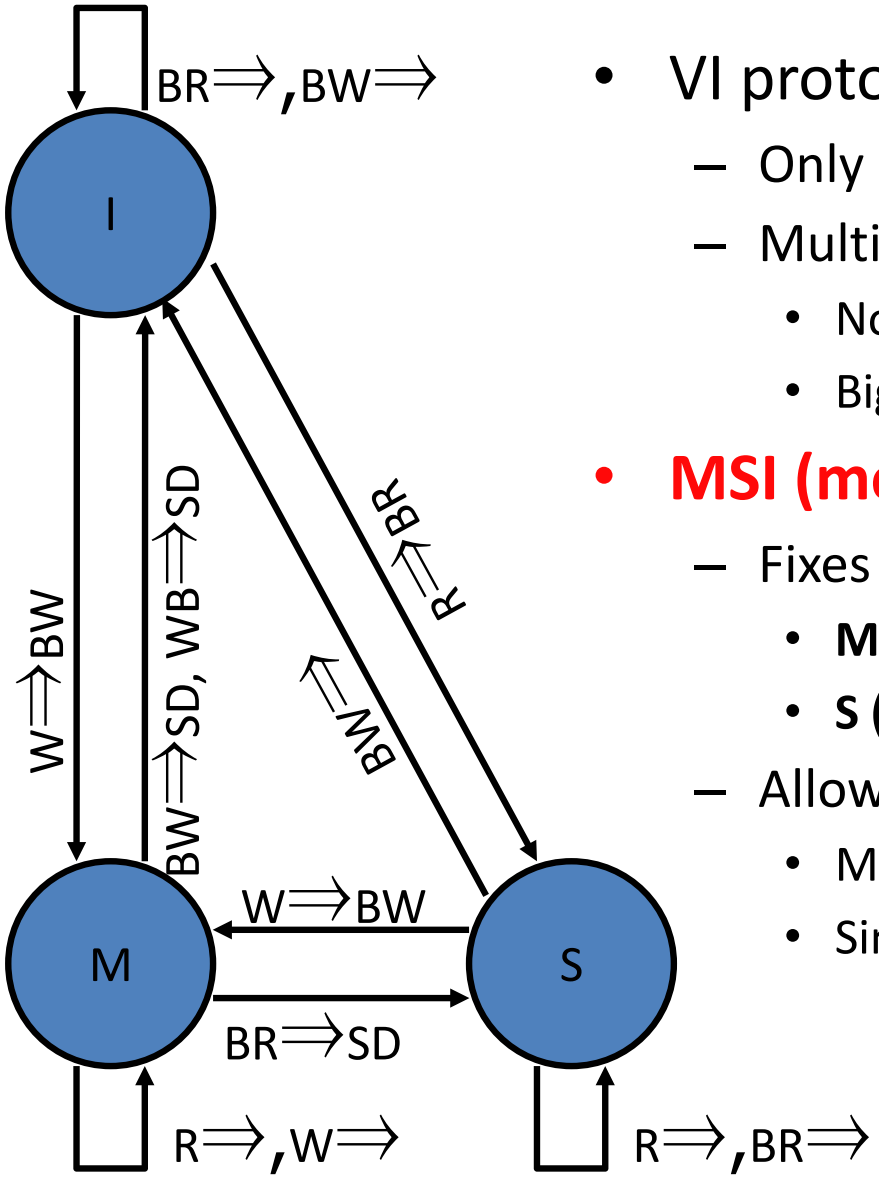
```
Processor 0                    Processor 1
0: addi $r3,$r1,&accts
1: lw $r4,0($r3)
2: blt $r4,$r2,6
3: sub $r4,$r4,$r2
4: sw $r4,0($r3)
5: jal dispense_cash           0: addi $r3,$r1,&accts
                               1: lw $r4,0($r3)
                               2: blt $r4,$r2,6
                               3: sub $r4,$r4,$r2
                               4: sw $r4,0($r3)
                               5: jal dispense_cash
```



- **`lw`** by processor 1 generates a BR (bus read)
  - processor 0 responds by sending its dirty copy, transitioning to **I**

Is this a good protocol? What is it actually doing? Can we do better? ☺

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)

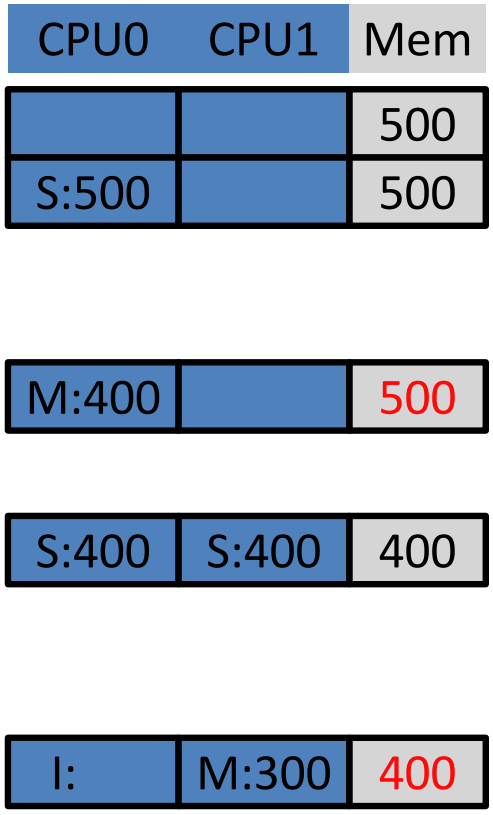
MSI Protocol State Transition Table

State	This Processor		Other Processor	
	Load	Store	Load Miss	Store Miss
Invalid (I)	Miss => S	Miss => M	---	---
Shared (S)	Hit	Upg Miss => M	---	=> I
Modified (M)	Hit	Hit	Send Data => S	Send Data => I

- M => S transition ***also updates memory,*** *why*?
  - After which memory willl respond (as all processors will be in S)

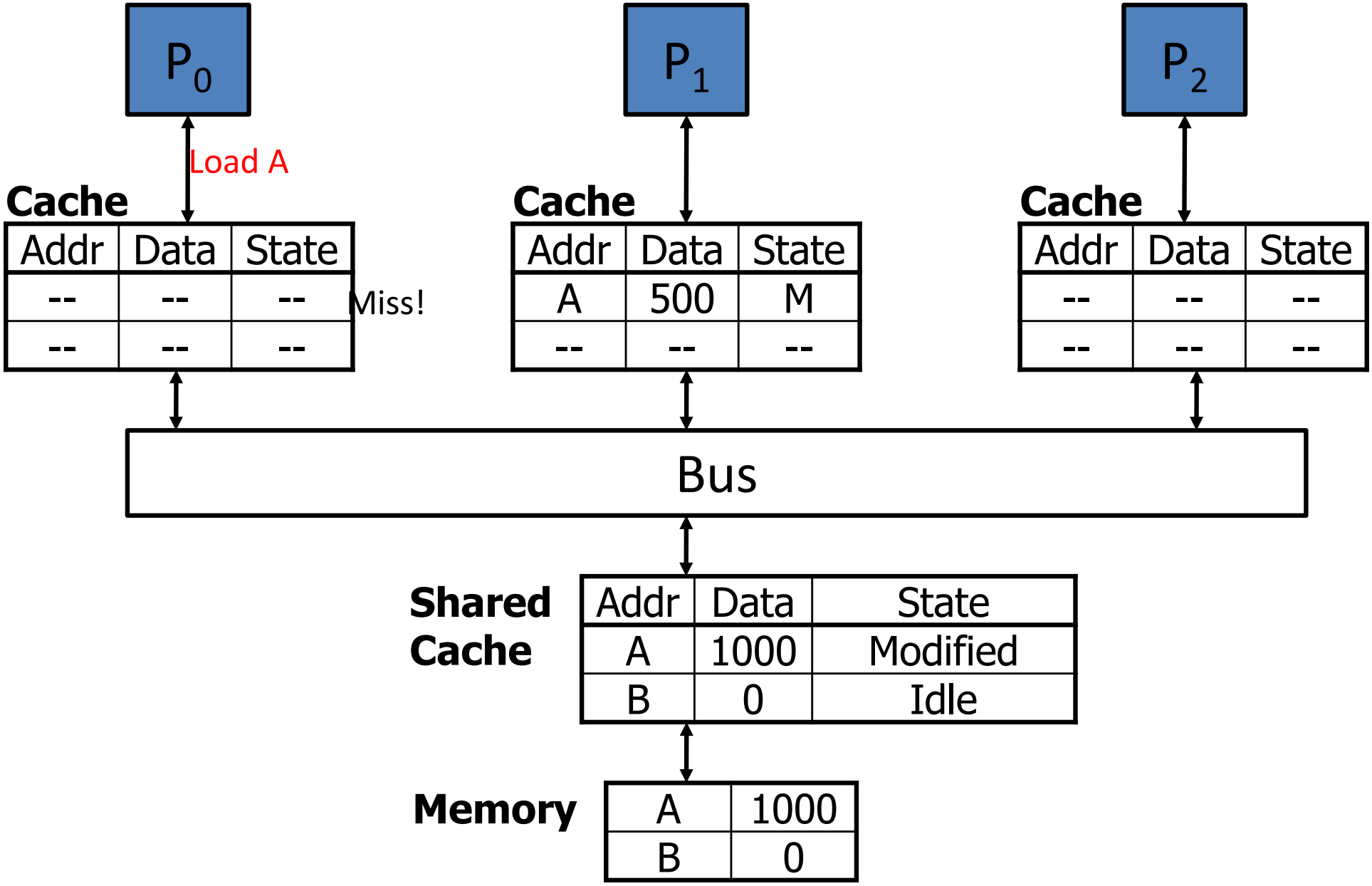
MSI Protocol (Write-Back Cache)

```
Processor 0                  Processor 1
0: addi $r3,$r1,&accts
1: lw $r4,0($r3)
2: blt $r4,$r2,6
3: sub $r4,$r4,$r2
4: sw $r4,0($r3)
5: jal dispense_cash         0: addi $r3,$r1,&accts
                             1: lw $r4,0($r3)
                             2: blt $r4,$r2,6
                             3: sub $r4,$r4,$r2
                             4: sw $r4,0($r3)
                             5: jal dispense_cash
```

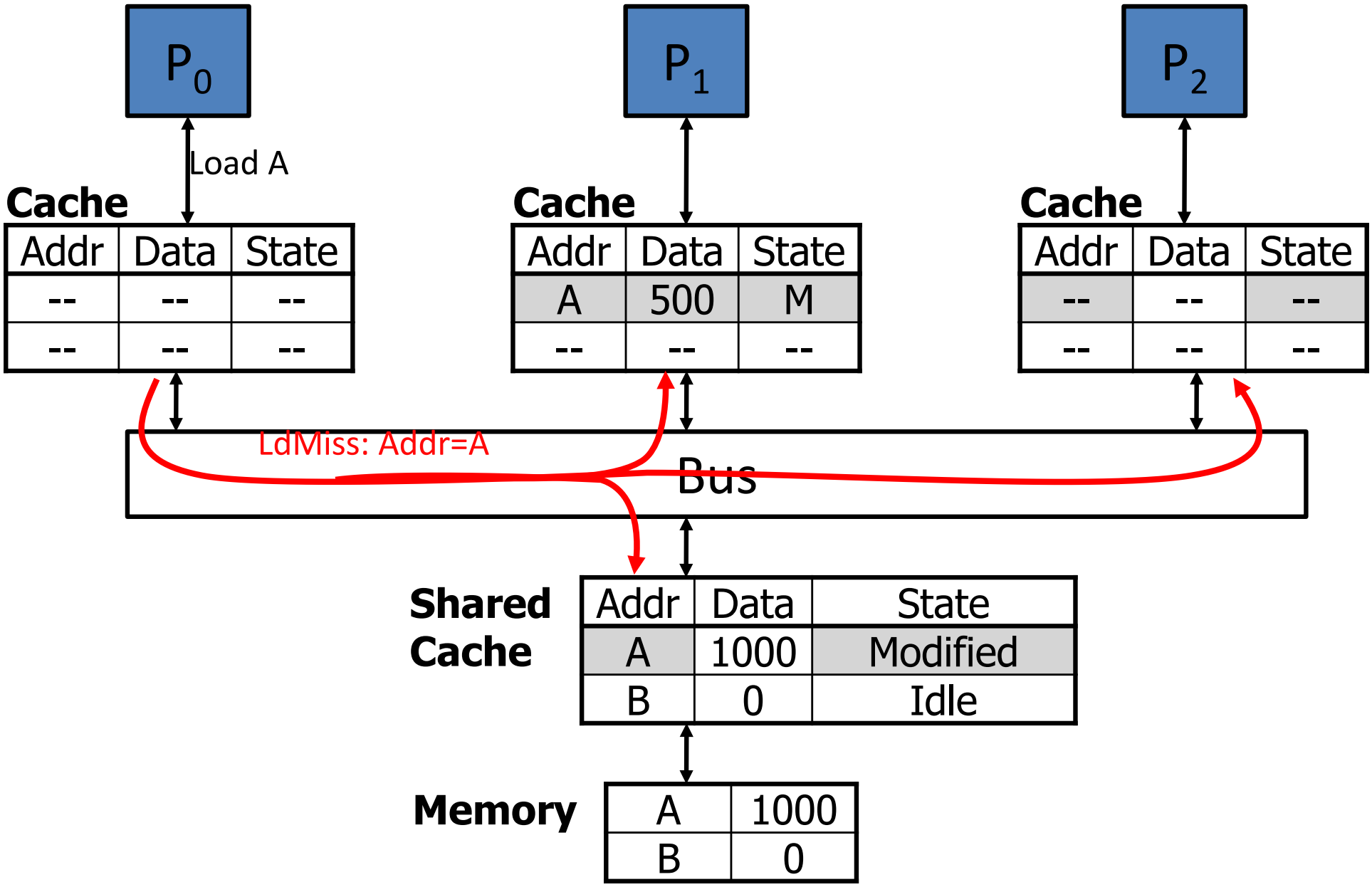


- **lw** by processor 1 generates a BR
  - Processor 0 responds by sending its dirty copy, transitioning to **S**
- **sw** by processor 1 generates a BW
  - Processor 0 responds by transitioning to **I**

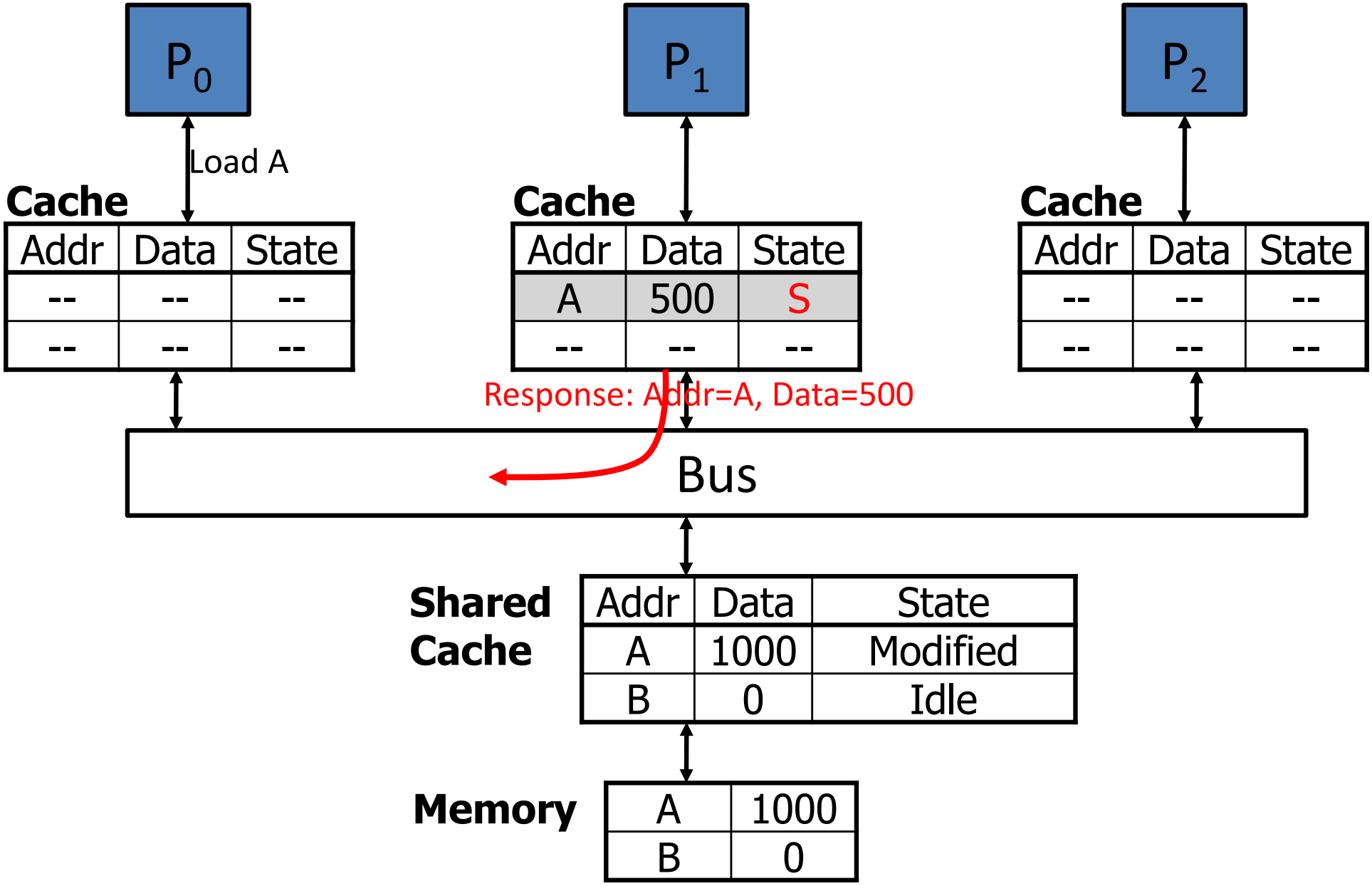
MSI Example: Step #1



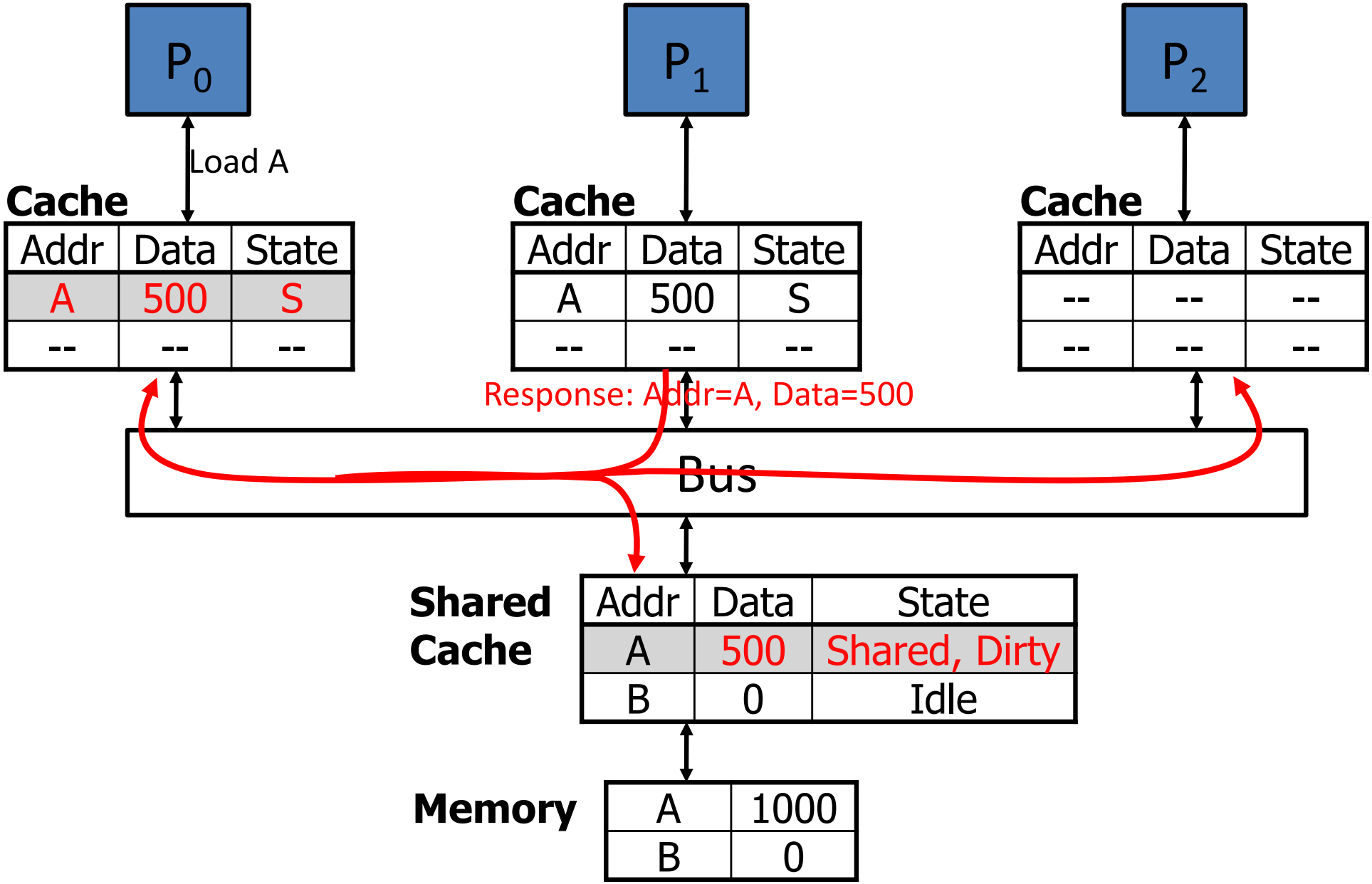
MSI Example: Step #2



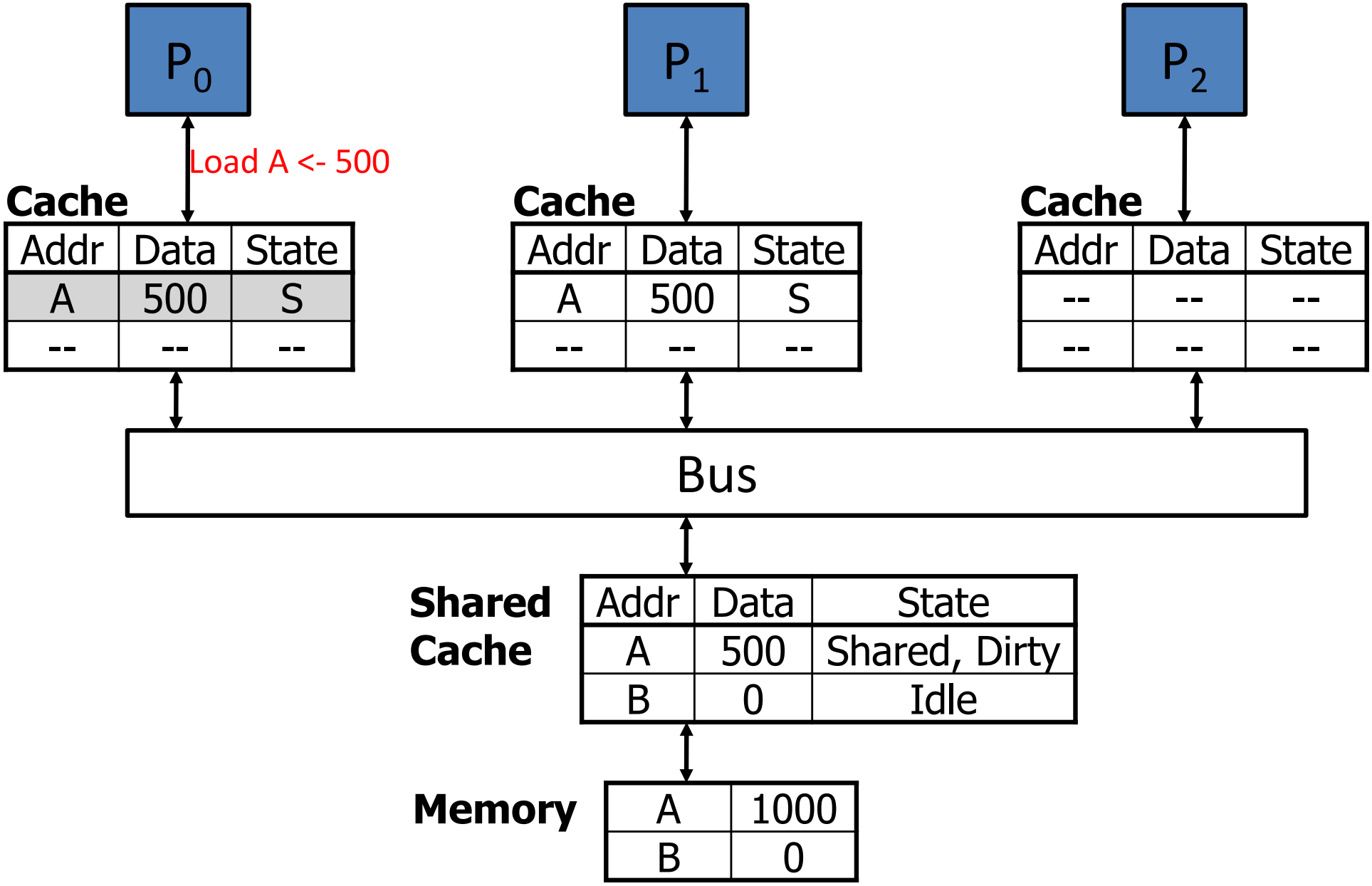
MSI Example: Step #3



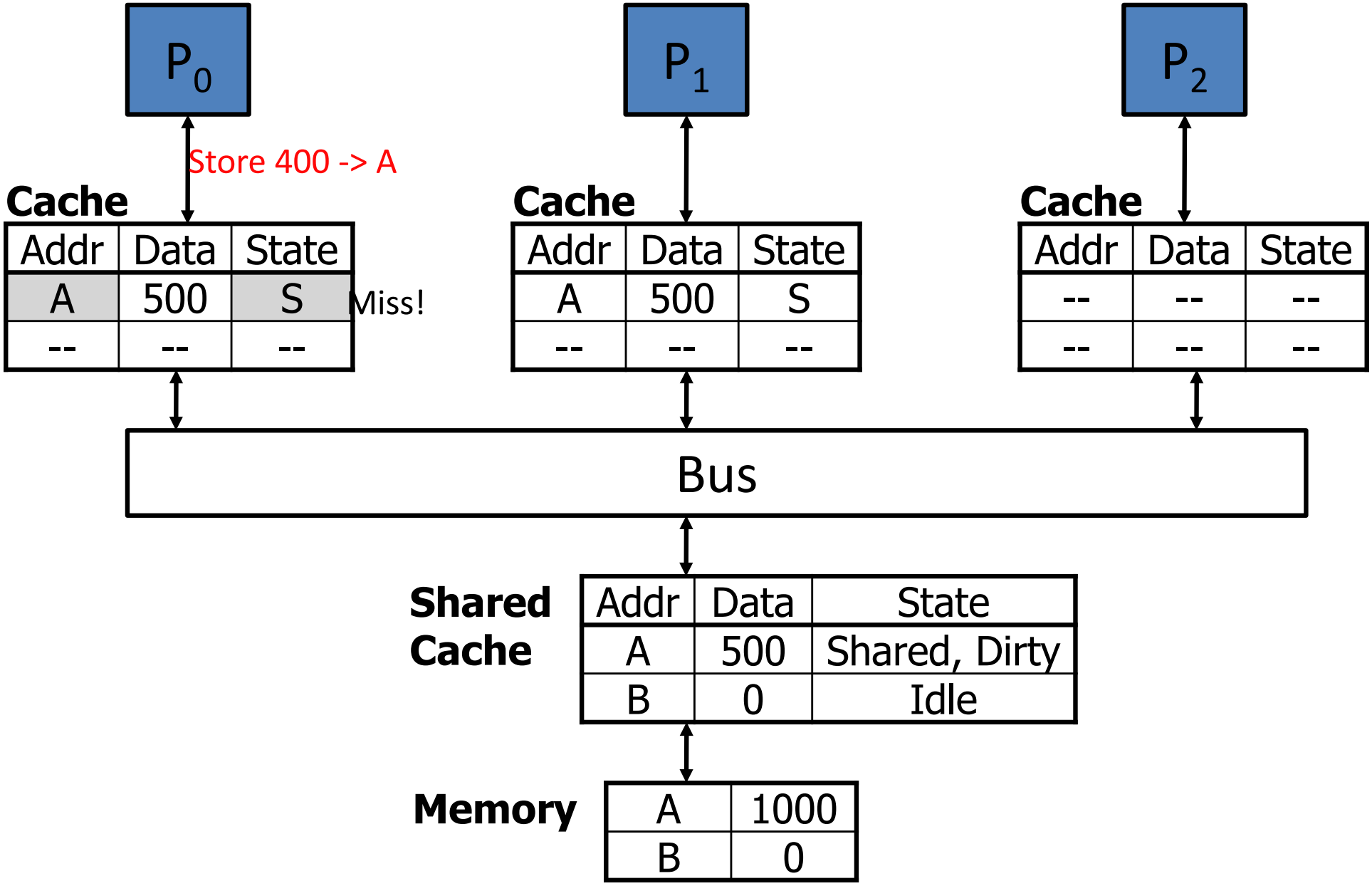
MSI Example: Step #4



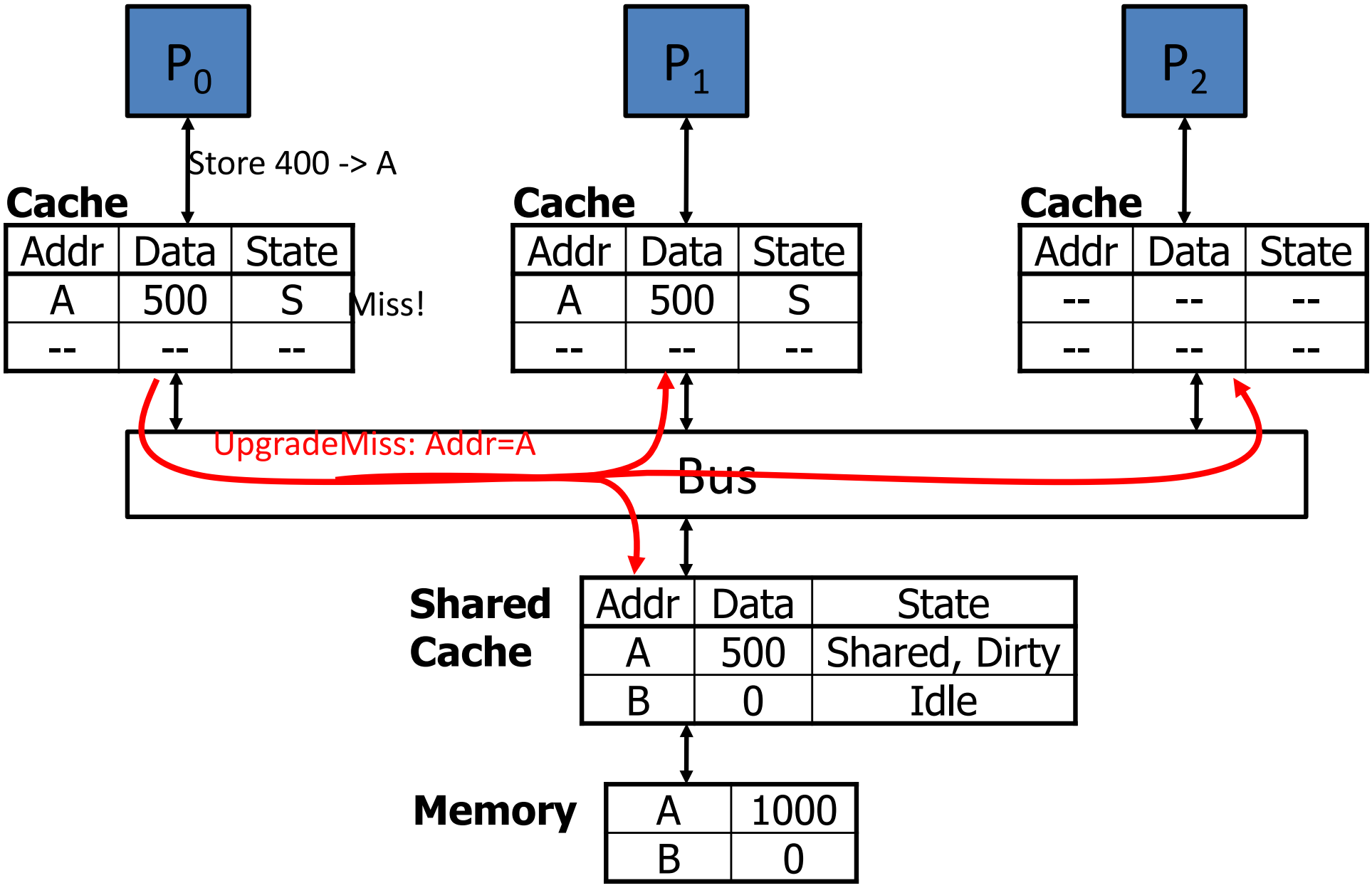
MSI Example: Step #5



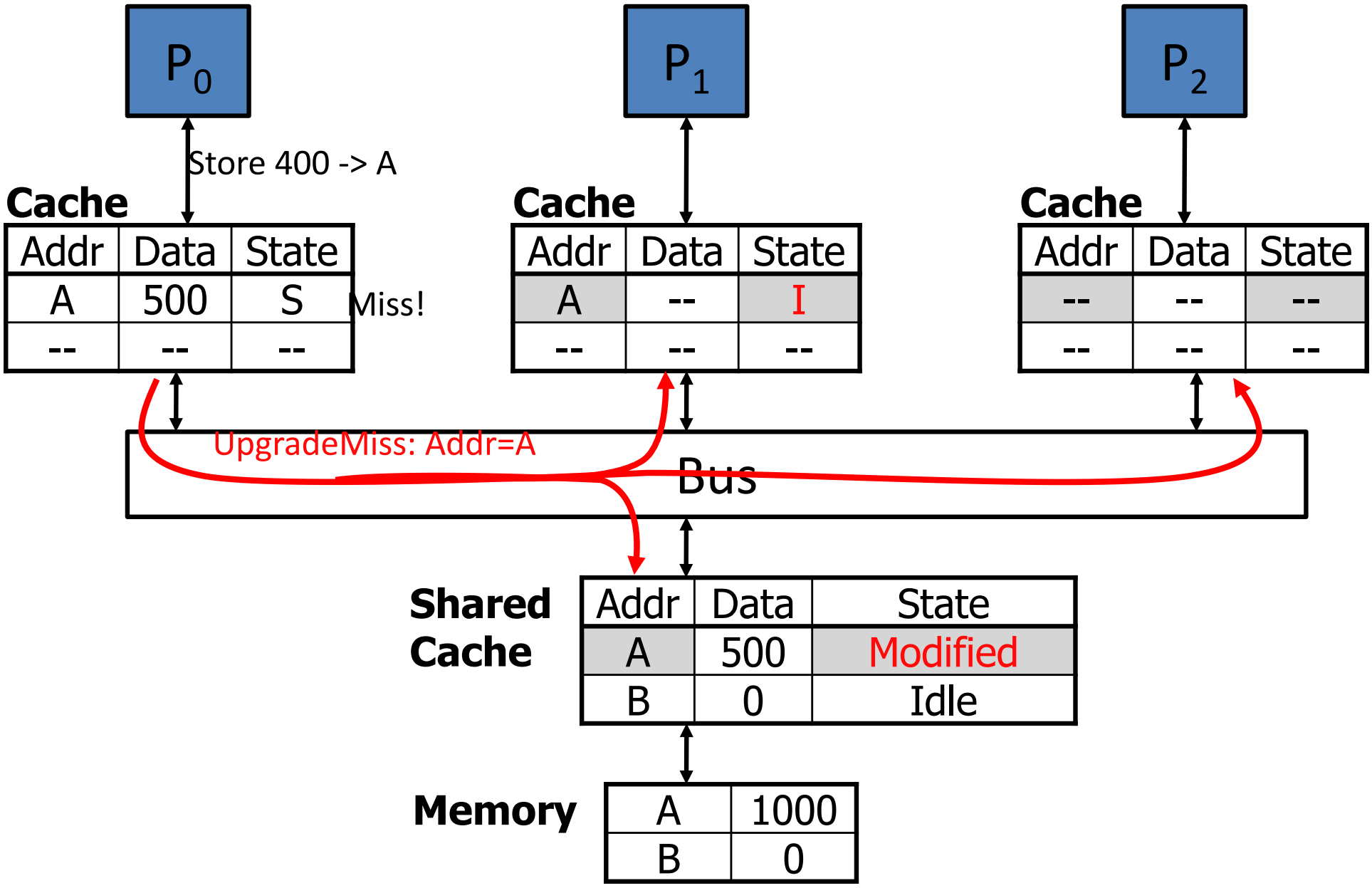
MSI Example: Step #6



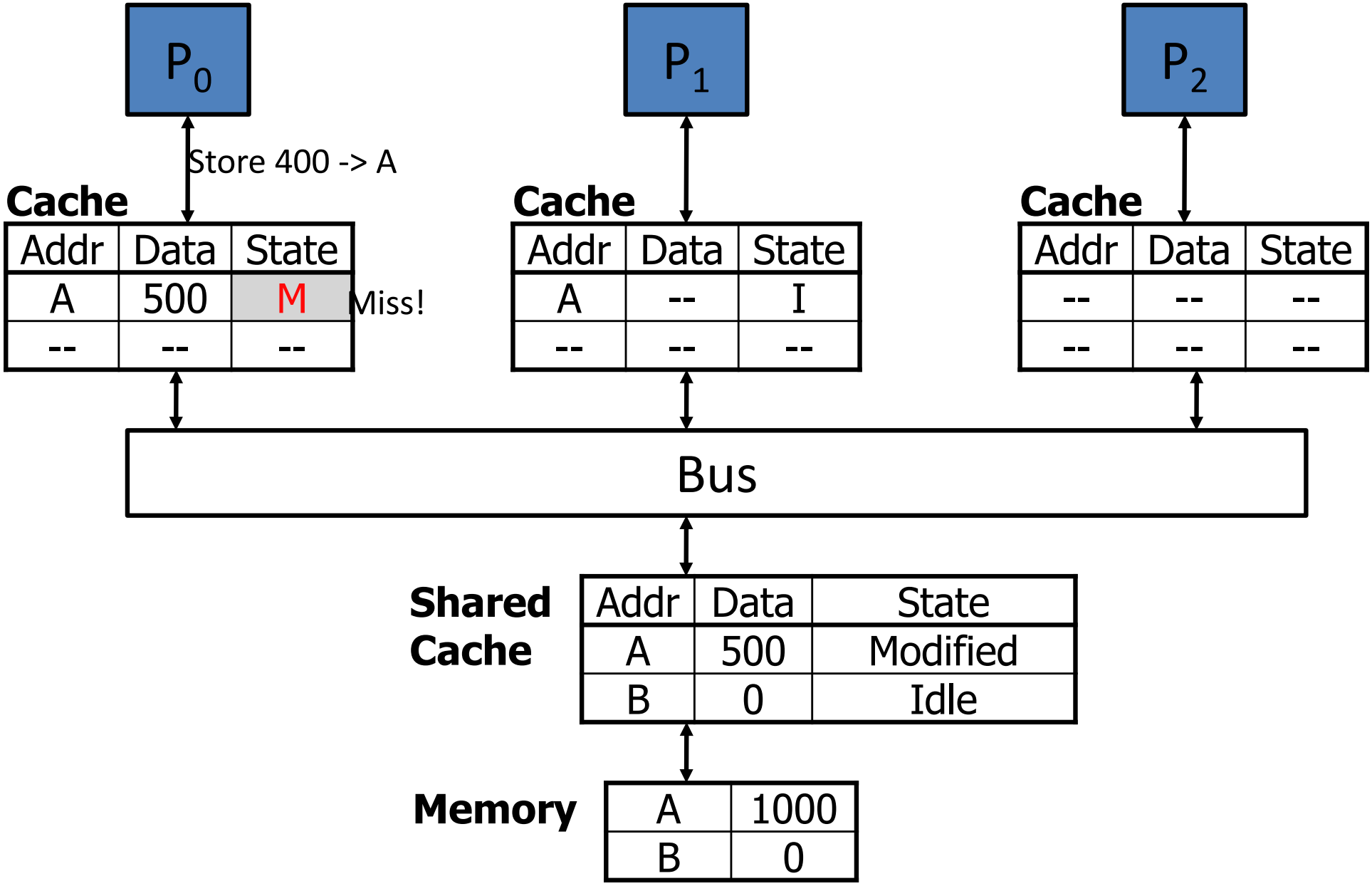
MSI Example: Step #7



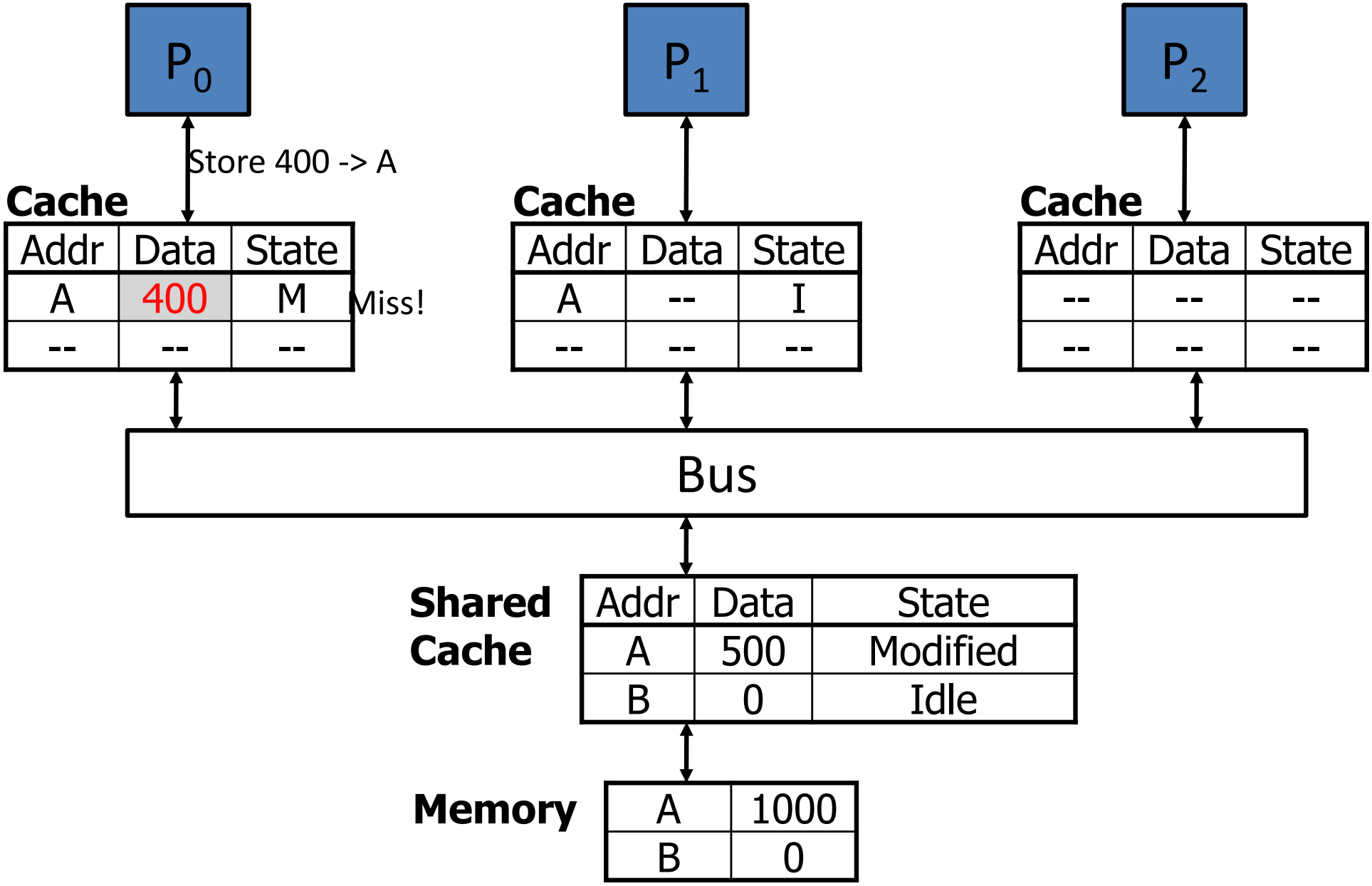
MSI Example: Step #8



MSI Example: Step #9



MSI Example: Step #10



Cache Coherence and Cache Misses

- Coherence introduces two new kinds of cache misses
  - **Upgrade miss**
    - On 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/invalidate
- Making the cache larger…
  - Doesn't reduce these type of misses
  - May increase these type of misses

Hmm.. What would happen in this case?

Processor 0

load A miss

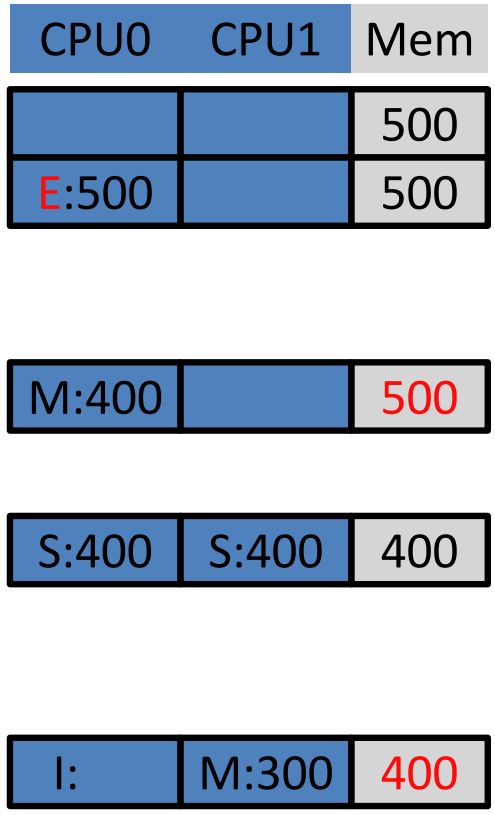
store A

MESI (4-state) Invalidation Protocol

- Problem with MSI protocol
  - Reading and modifying data is 2 bus transactions, even if no sharing
    - e.g. even in sequential program
    - BusRd (I->S) followed by BusRdX or BusUpgr (S->M)
- Add *exclusive* state: write locally without transaction, but not modified
  - Main memory is up to date, so cache not necessarily owner
  - States
    - invalid
    - exclusive or *exclusive-clean* (only this cache has copy, but not modified)
    - shared (two or more caches may have copies)
    - modified (dirty)
  - I -> E on PrRd if no other processor has a copy

MESI Example

```
Processor 0                 Processor 1
0: addi $r3,$r1,&accts
1: lw $r4,0($r3)
2: blt $r4,$r2,6
3: sub $r4,$r4,$r2
4: sw $r4,0($r3)
```



- Most modern protocols also include **E (exclusive)** state
  - Interpretation: "I have the only cached copy, and it's a **clean** copy"
  - Why would this state be useful?

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	---	=> I
Exclusive (E)	Hit	Hit => M	Send Data => S	Send Data => I
Modified (M)	Hit	Hit	Send Data => S	Send Data => I

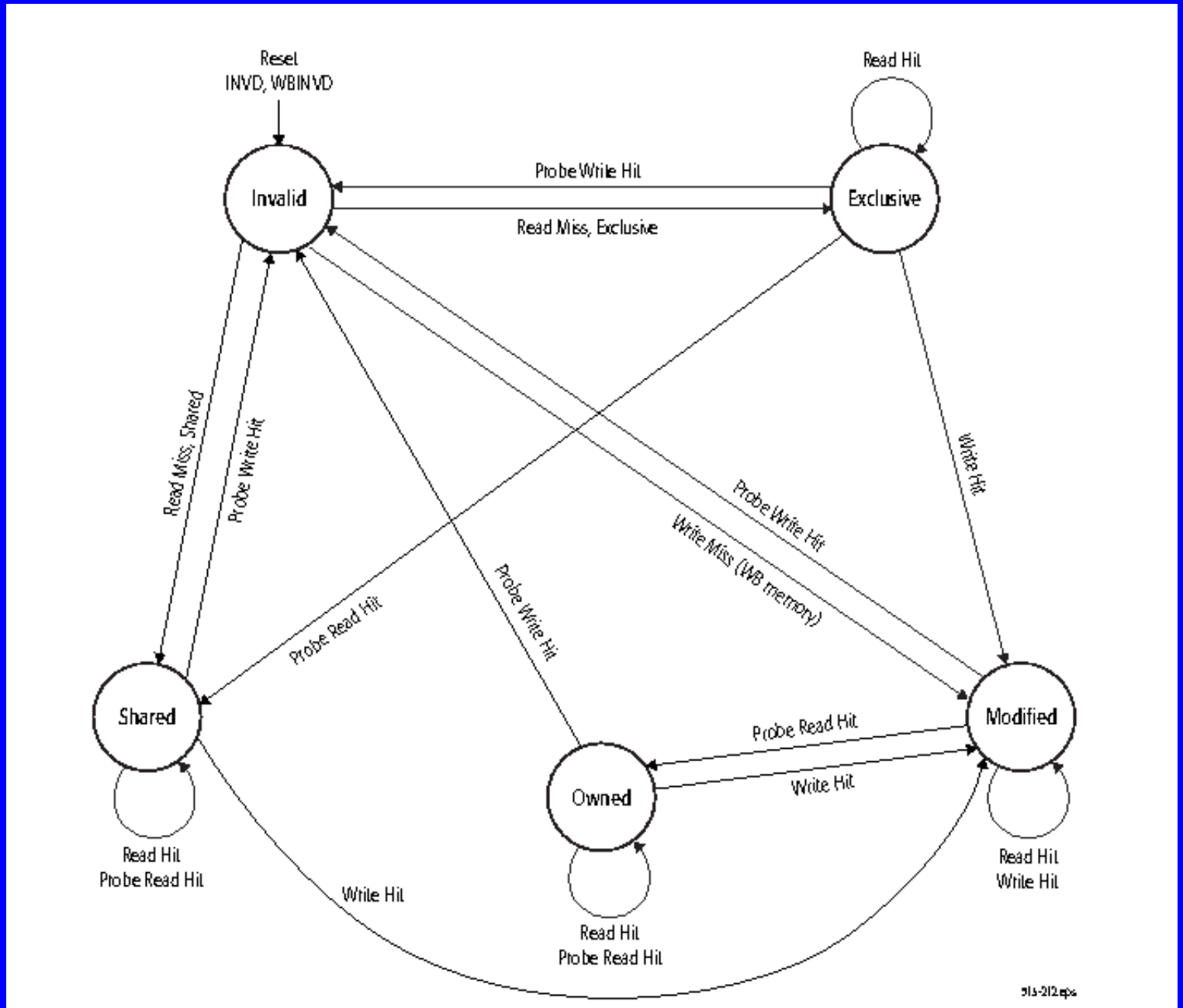
- Load misses lead to "E" if no other processors is caching the block

Can we do better than MESI?

- What if we have a lot of producer-consumer?

Can we do better than MESI?

- What if we have a lot of producer-consumer?
  - That would suck… you can't shared cache-cache dirty data without writing back…
- MOESI: Add yet another state "Owned"
  - Owned means: I am responsible for write back
  - Can alternate between O and M without writing data back to memory, hence useful for producer-consumer


Reset
INVD, WBINVD
Read Hit
Invalid
Probe Write Hit
Exclusive
Read Miss, Exclusive
Read Miss, Shared
Probe Write Hit
Probe Write Hit
Write Hit
Write Miss (WB memory)
Probe Read Hit
Probe Write Hit
Shared
Modified
Probe Read Hit
Owned
Write Hit
Read Hit
Probe Read Hit
Read Hit
Write Hit
Write Hit
Read Hit
Probe Read Hit
513-212.eps

Snooping Bandwidth Scaling Problems

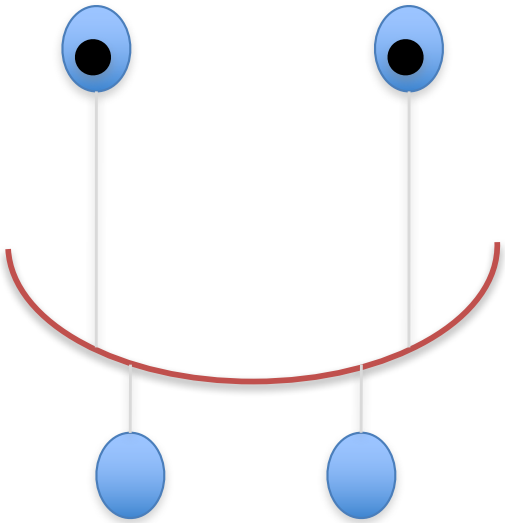
- Coherence events generated on…
  - L2 misses (and writebacks)
- Problem#1: **N^2 bus traffic**
  - All N processors send their misses to all N-1 other processors
  - Assume: 2 IPC, 2 Ghz clock, 0.01 misses/insn **per processor**
  - 0.01 misses/insn * 2 insn/cycle * 2 cycle/ns * 64 B blocks
    = 2.56 GB/s… per processor
    - With 16 processors, that's 40 GB/s! With 128 that's 320 GB/s!!
  - You can use multiple buses… but that complicates the protocol
- Problem#2: **N^2 processor snooping bandwidth**
  - 0.01 events/insn * 2 insn/cycle = 0.02 events/cycle per processor
  - 16 processors: 0.32 bus-side tag lookups per cycle
    - Add 1 extra port to cache tags? Okay
  - 128 processors: 2.56 tag lookups per cycle! 3 extra tag ports?
- Now add a GPU to the coherence domain, what happens?

Shared broadcast bus

Key: bus arbiter services one request at a time

Advantage: conceptually simple

Disadvantage: shared, difficult to scale



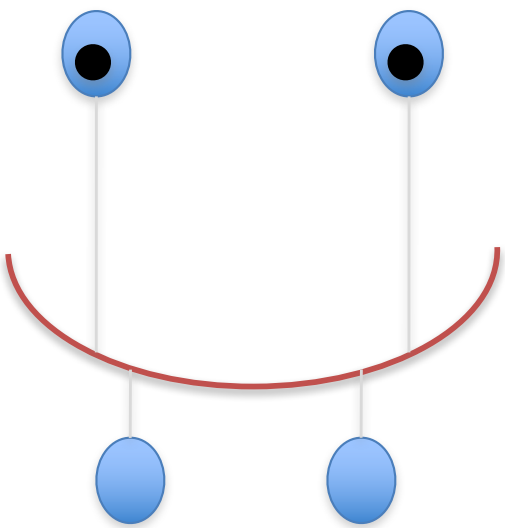
- Coherence events generated on…
  - L2 misses (and writebacks)
- Problem#1: **N^2 bus traffic**
  - All N processors send their misses to all N-1 other processo
  - Assume: 2 IPC, 2 Ghz clock, 0.01 misses/insn **per process**
    = 2.56 GB/s… per processor
- Problem#2: **N^2 processor snooping bandwid**
  - 0.01 events/insn * 2 insn/cycle = 0.02 events/cycle per pro
  - 16 processors: 0.32 bus-side tag lookups per cycle
  - 128 processors: 2.56 tag lookups per cycle! 3 extra tag po
- Now add a GPU to the coherence domain, what happens?

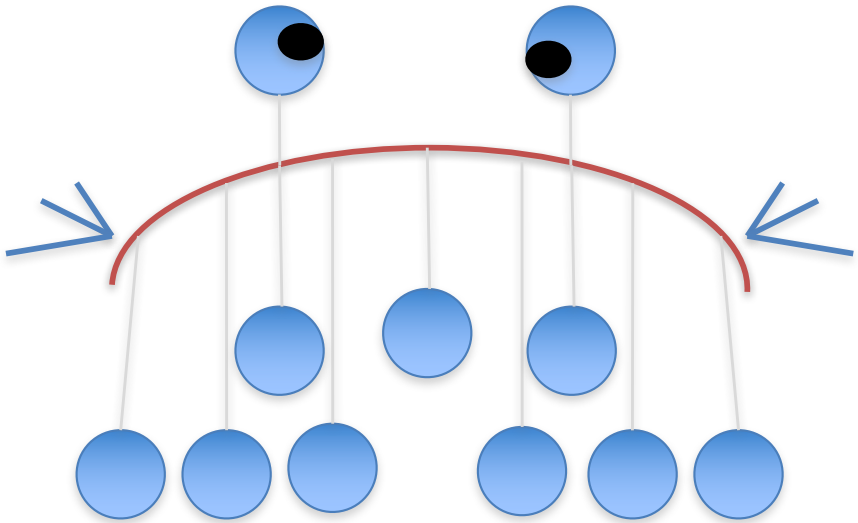
Shared broadcast bus

Key: bus arbiter services one request at a time

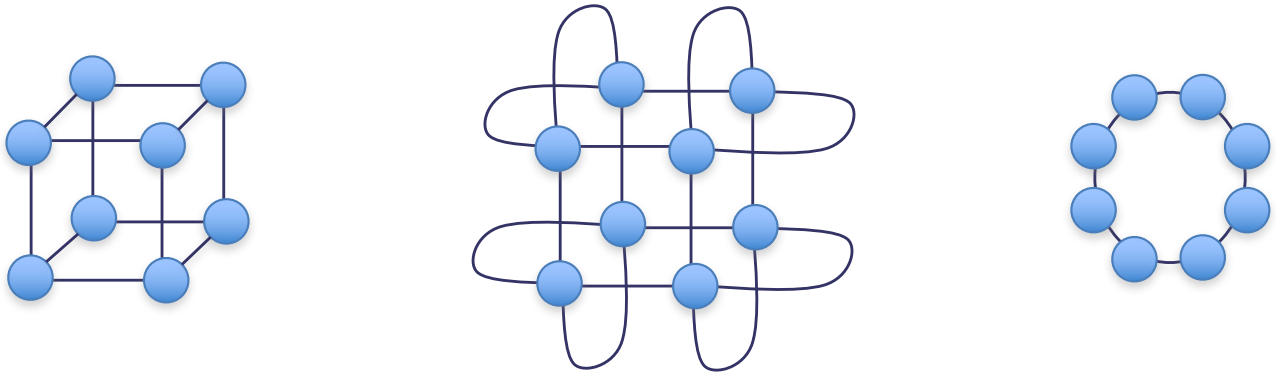
Advantage: conceptually simple

Disadvantage: shared, difficult to scale





point-to-point networks



Advantages:

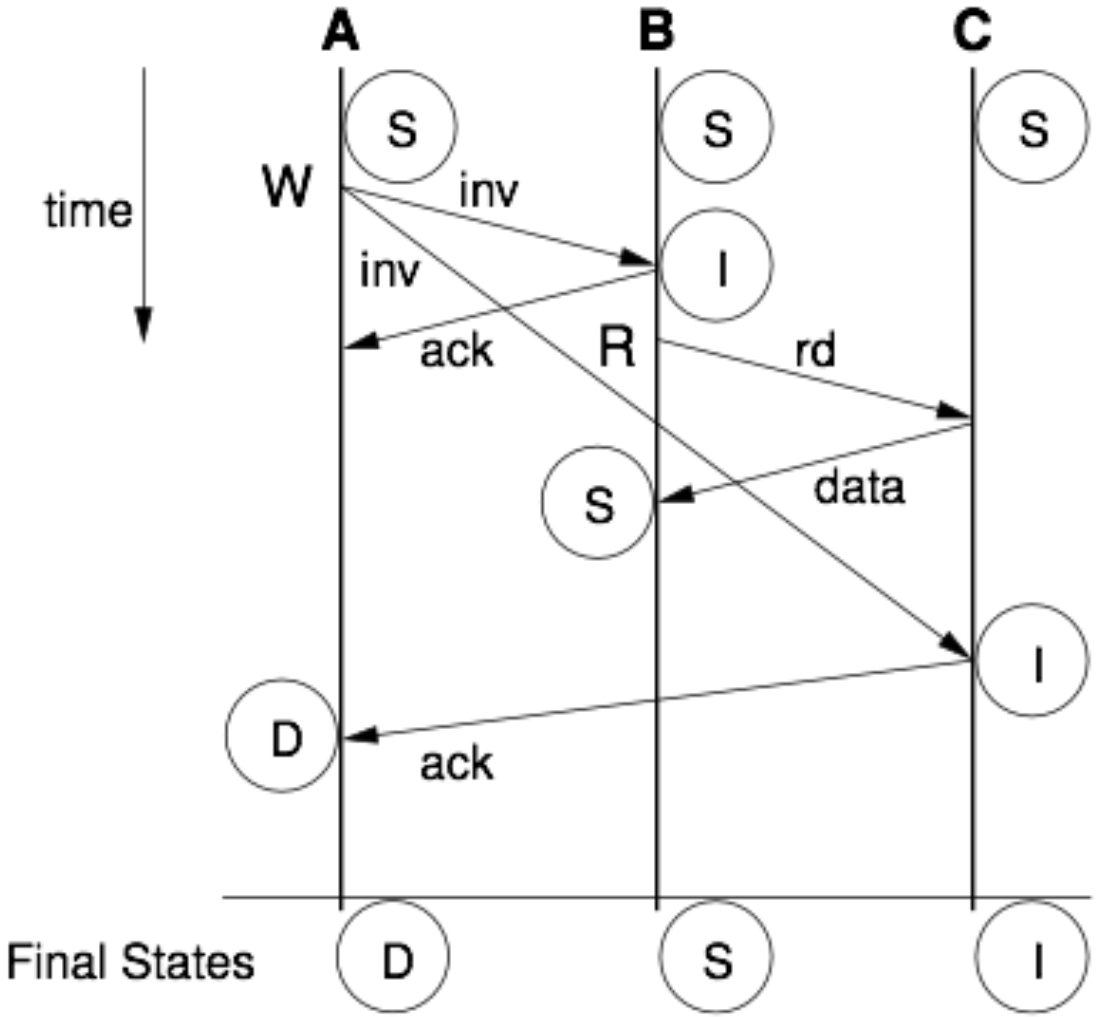
- better electrical behavior (shorter wires)
- coherence transaction parallelism

Problem: unordered network

Nodes may observe messages in different orders

Is this a problem?

Ordering – What is wrong here?



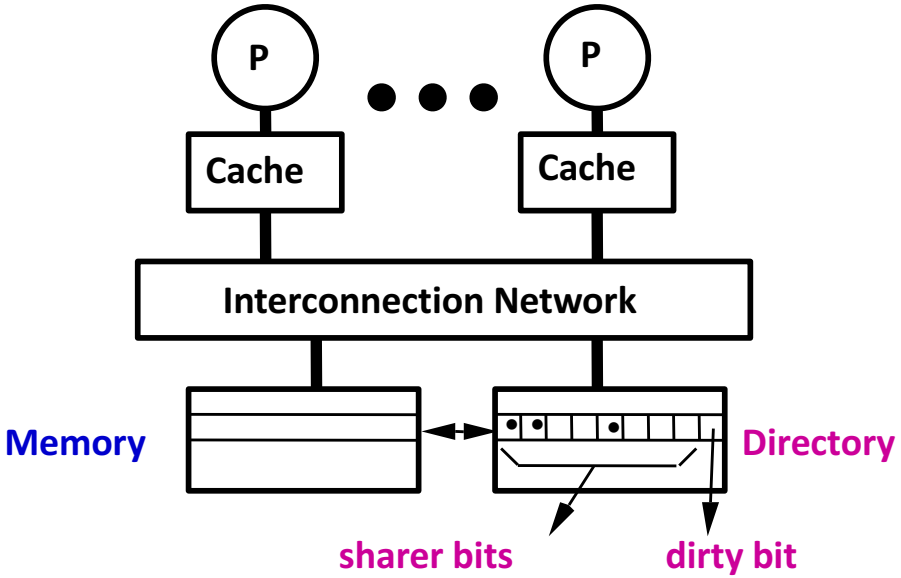
How do we fix it?

Directory Coherence Protocols

- **Directories**:
  - Extend memory (or shared cache) to track caching information
  - For each physical cache block, track:
    - **Owner**: which processor has a dirty copy (I.e., M state)
    - **Sharers**: which processors have clean copies (I.e., S state)
  - Processor sends coherence event to directory
    - Directory sends events only to processors **as needed**
  - Avoids non-scalable broadcast used by snooping protocols, does not require strong ordering properties from the network
  - For multicore with shared L3 cache, put directory info in cache tags

- For high-throughput, directory can be banked/partitioned
  + Use address to determine which bank/module holds a given block
    - That bank/module is called the “home” for the block

Basic Scheme

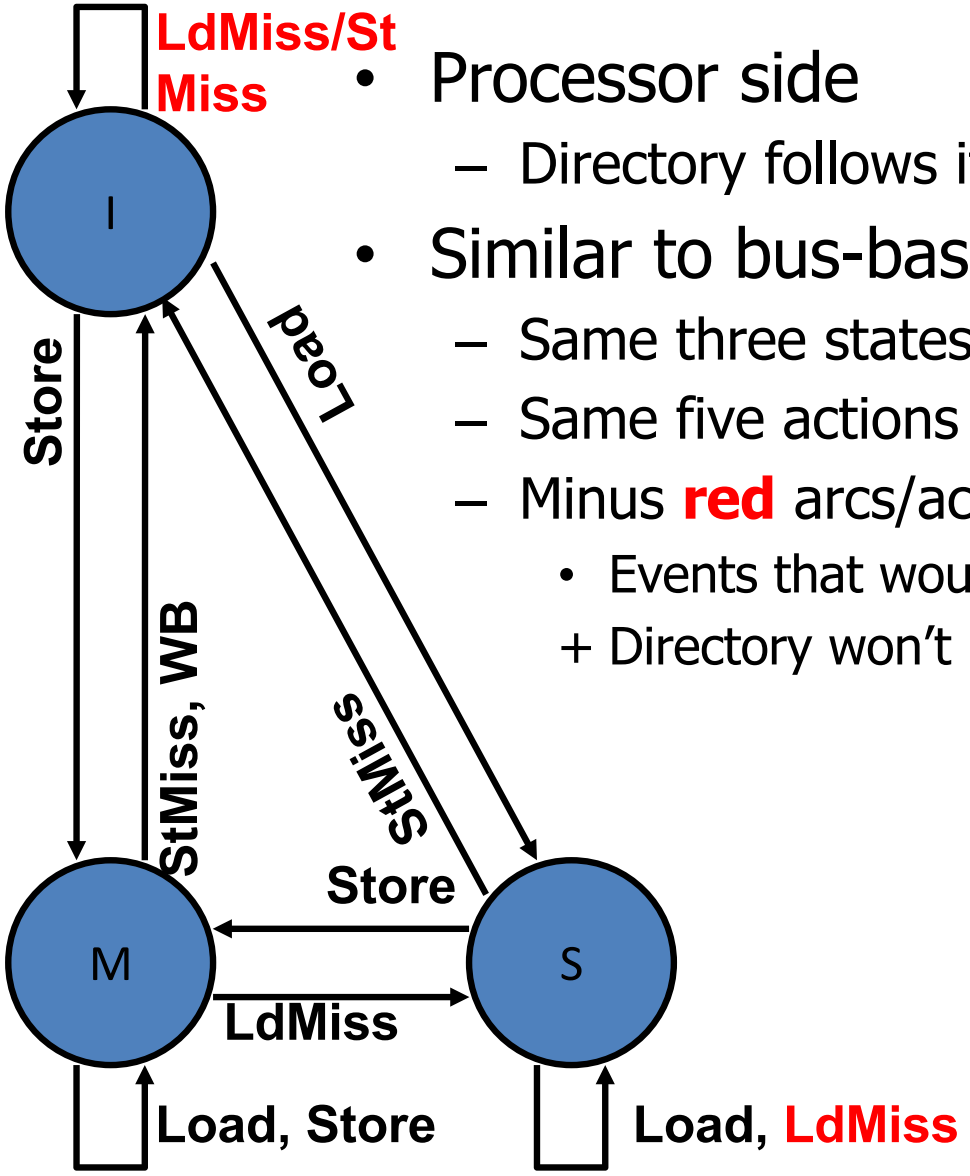


- **Assume P processors**
- **With each cache-block in memory:**
  - **P sharer bits**
  - **1 dirty bit**
- **With each cache-block in cache:**
  - **1 valid bit**
  - **1 dirty (owner) bit**

– Read from main memory by PE-i:
  - if dirty-bit is OFF then { read from main memory; turn p[i] ON; }
  - if dirty-bit is ON then { recall line from dirty PE (cache state to shared); update memory; turn dirty-bit OFF; turn p[i] ON; supply recalled data to PE-i; }

– Write to main memory by PE-i:
  - if dirty-bit OFF then { supply data to PE-i; send invalidations to all PEs caching that block; turn dirty-bit ON; turn P[i] ON; ... }
  - ...

MSI Directory Protocol



- Processor side
  - Directory follows its own protocol
- Similar to bus-based MSI
  - Same three states
  - Same five actions (keep BR/BW names)
  - Minus **red** arcs/actions
    - Events that would not trigger action anyway
    + Directory won't bother you unless you need to act

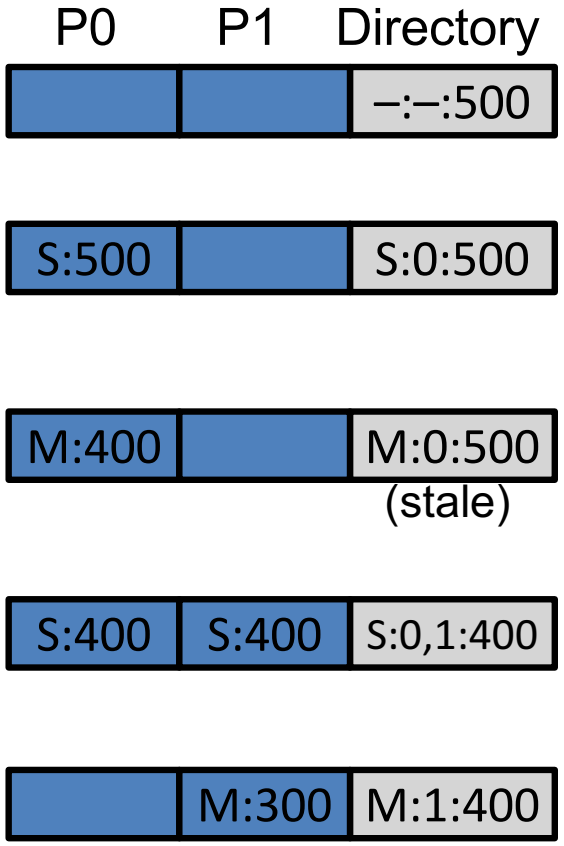
MSI Directory Protocol

Processor 0

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

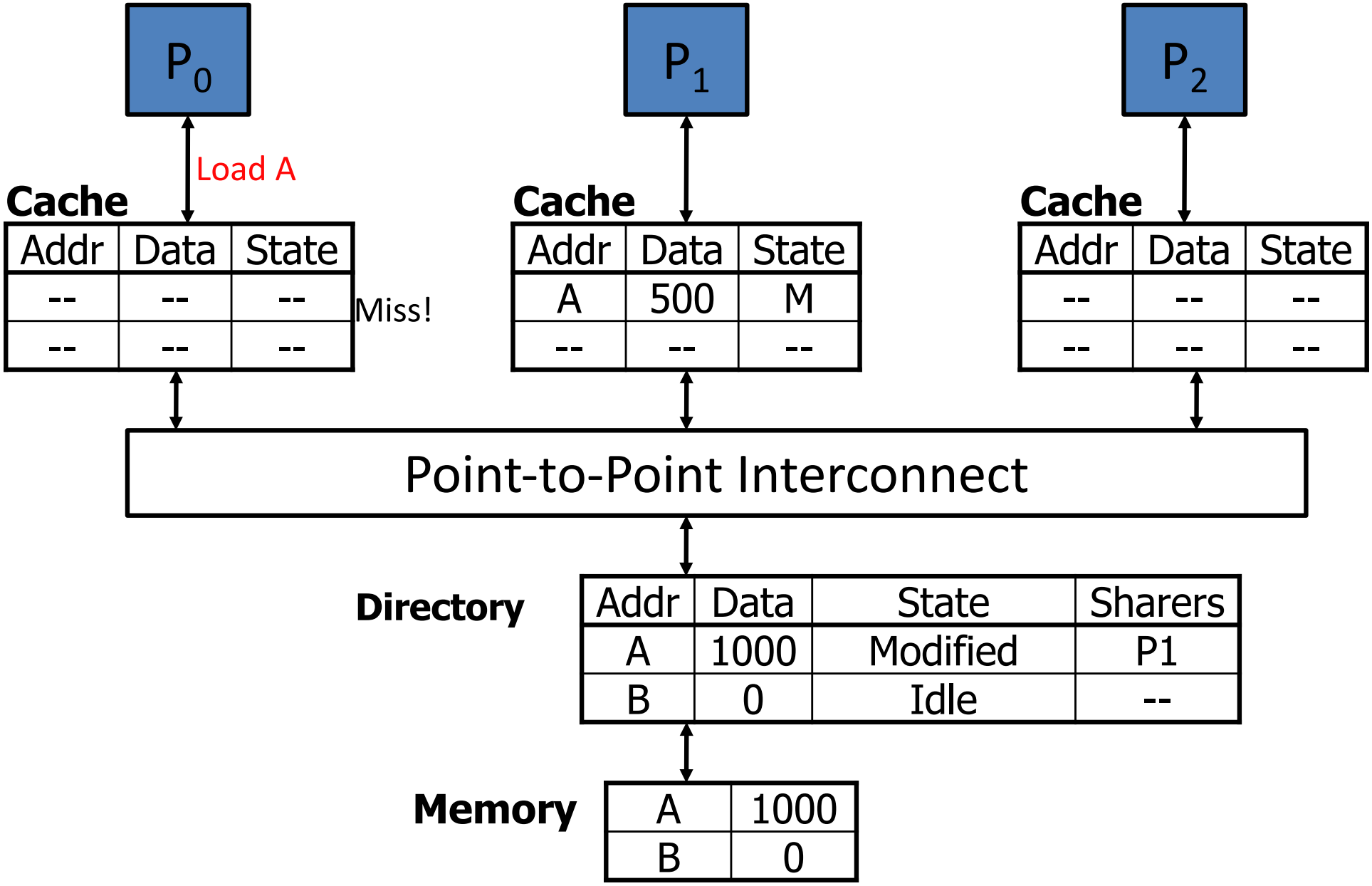
Processor 1

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

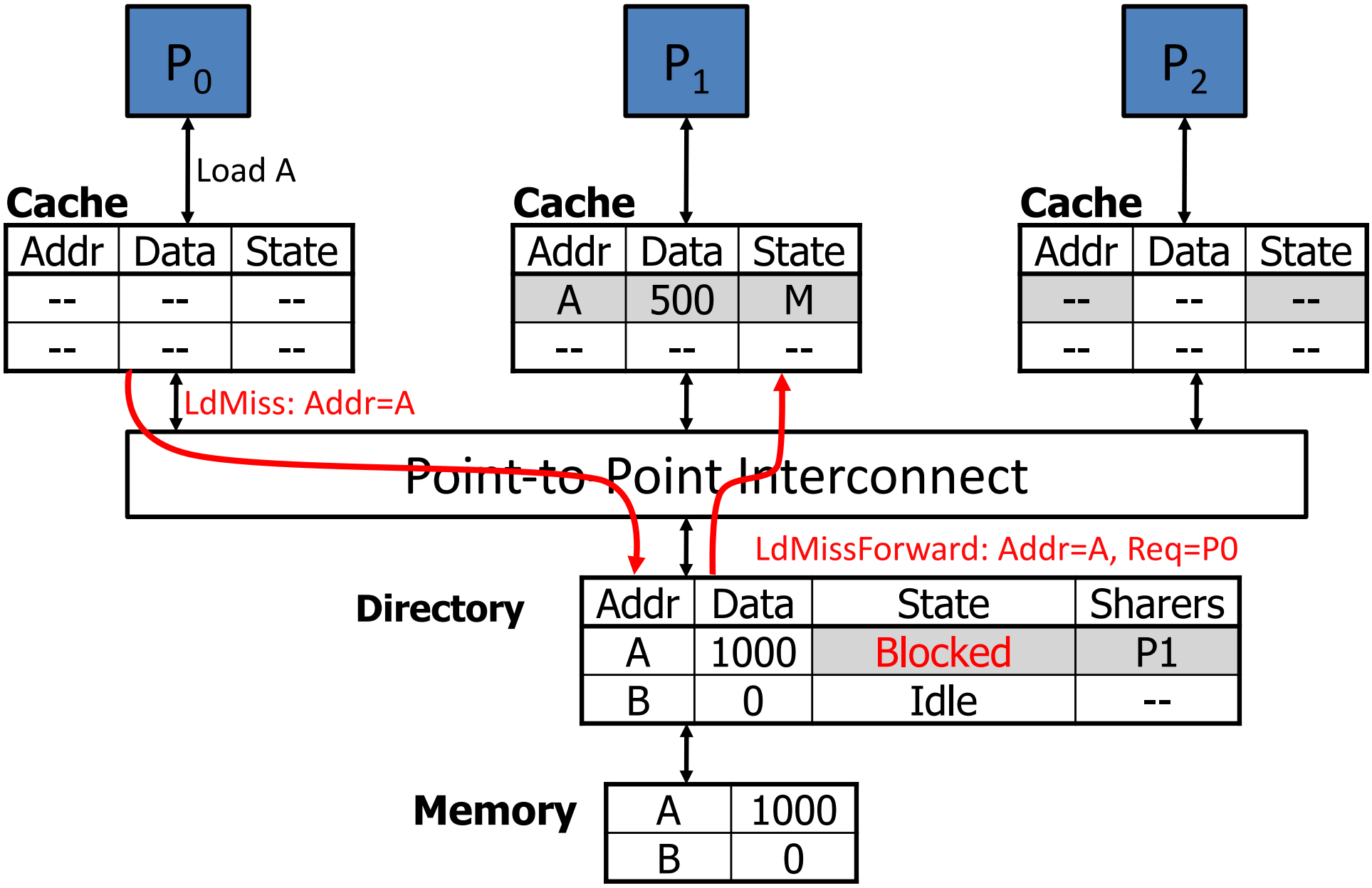


- **ld** by P1 sends BR to directory
  - Directory sends BR to P0, P0 sends P1 data, does WB, goes to **S**
- **st** by P1 sends BW to directory
  - Directory sends BW to P0, P0 goes to **I**

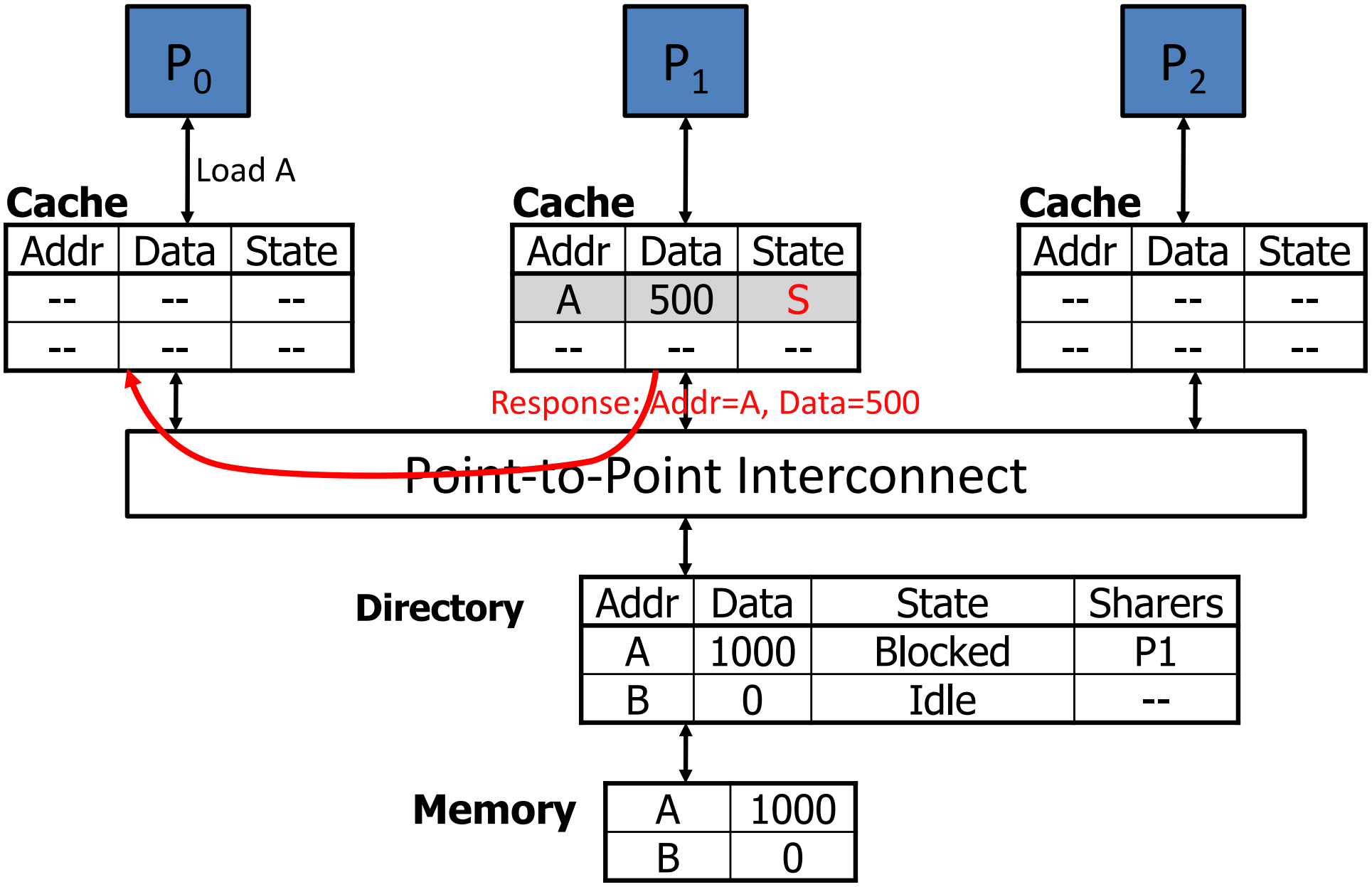
Directory Example: Step #1



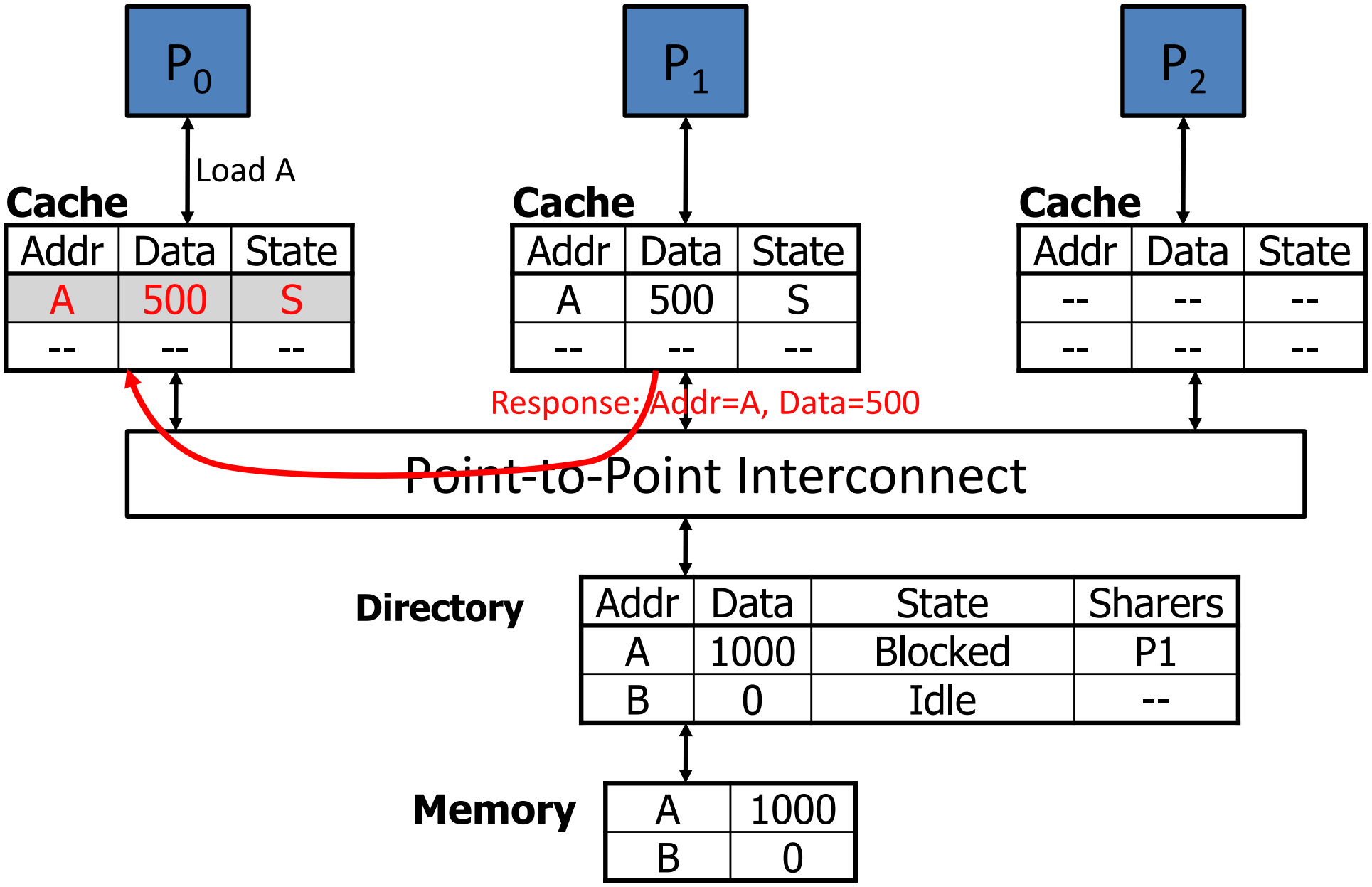
Directory Example: Step #2



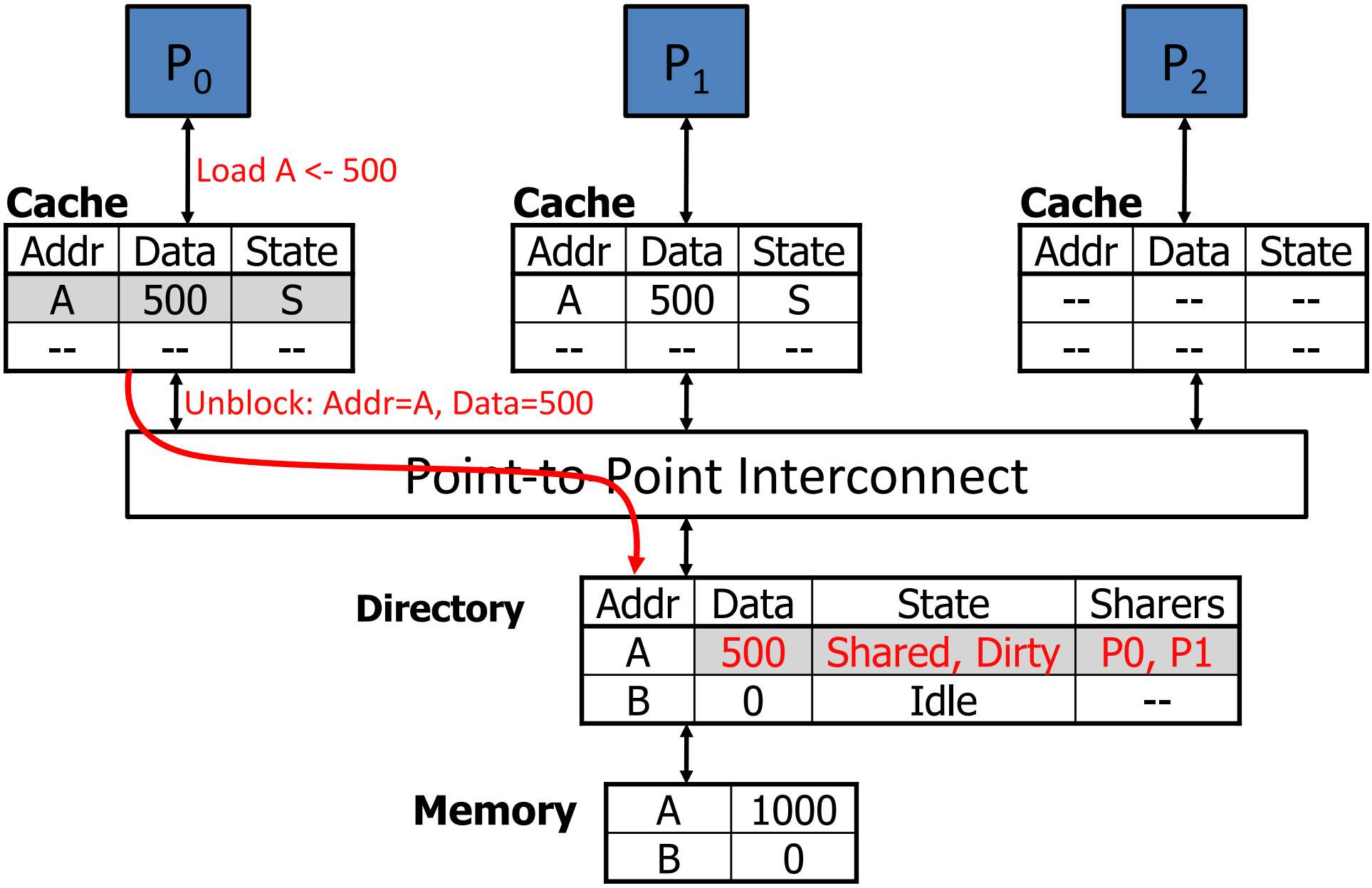
Directory Example: Step #3



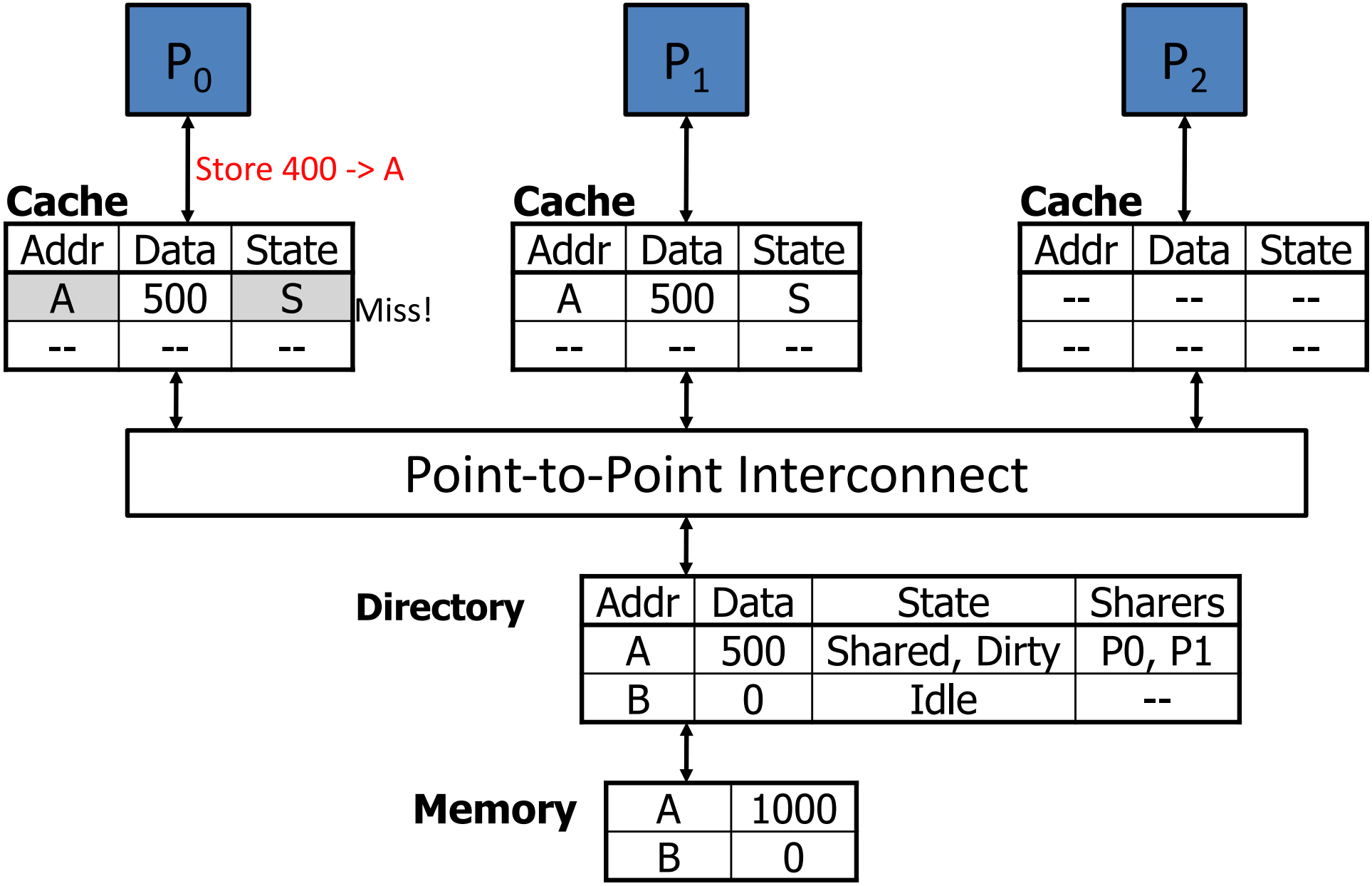
Directory Example: Step #4



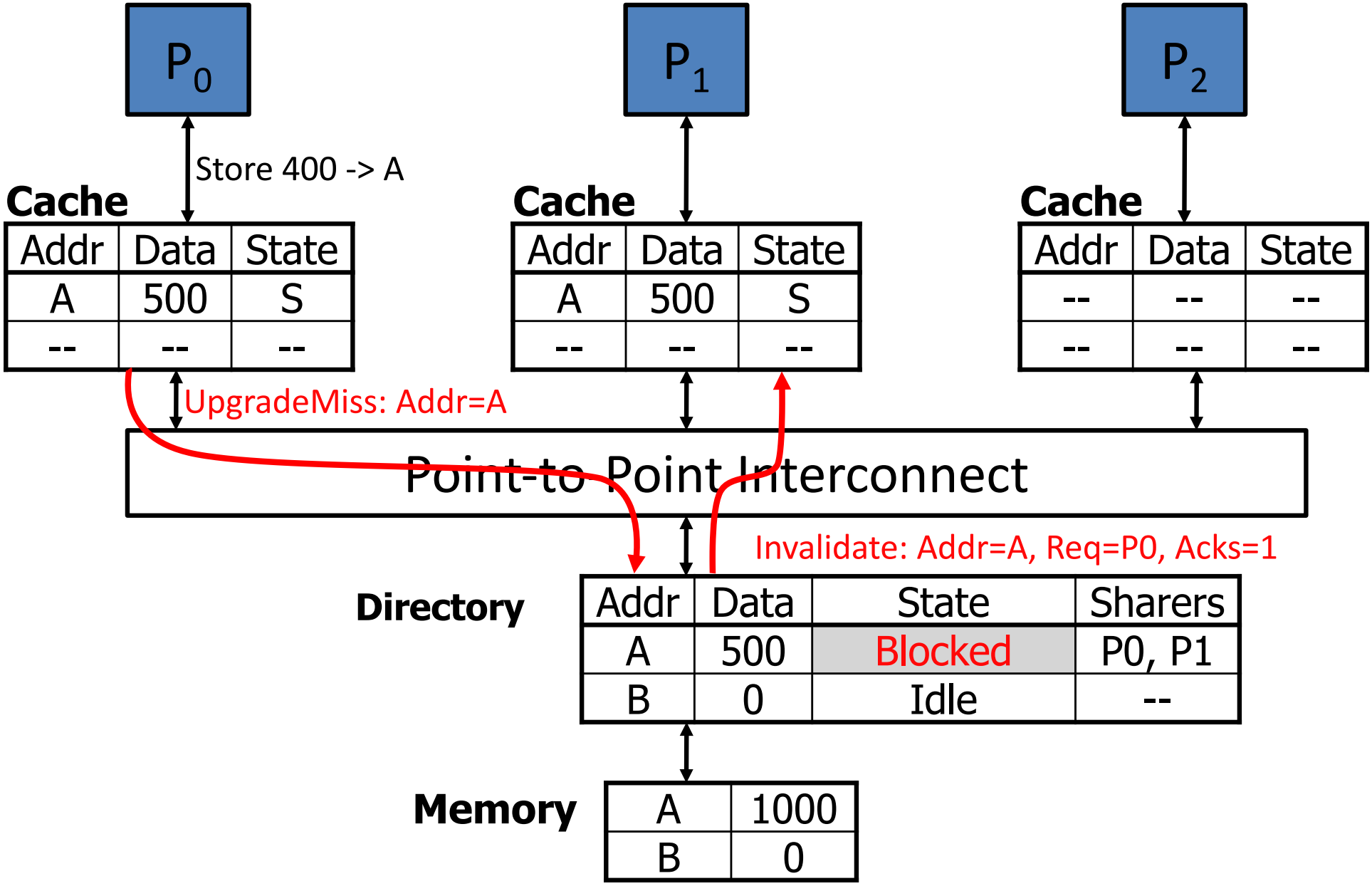
Directory Example: Step #5



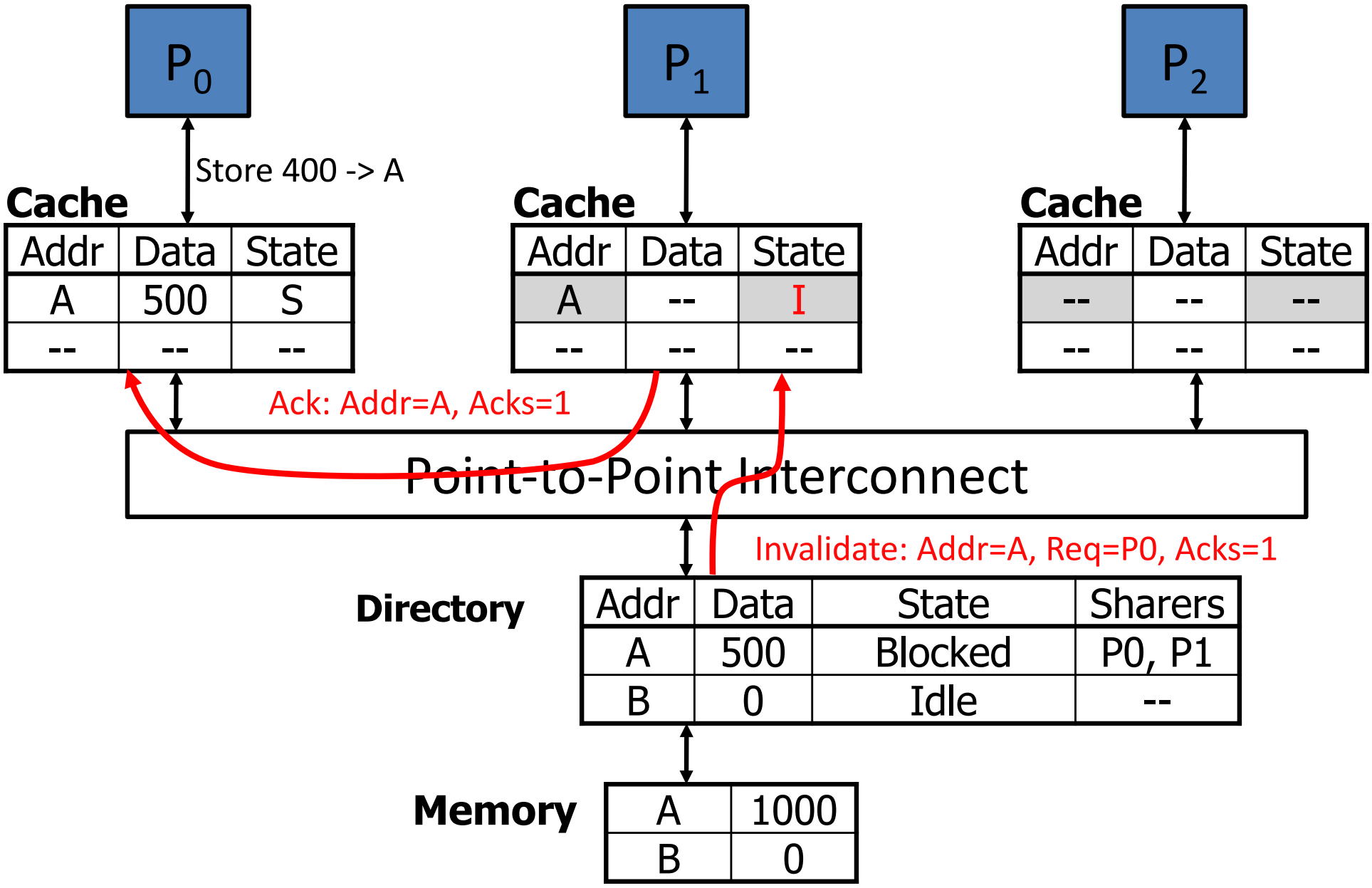
Directory Example: Step #6



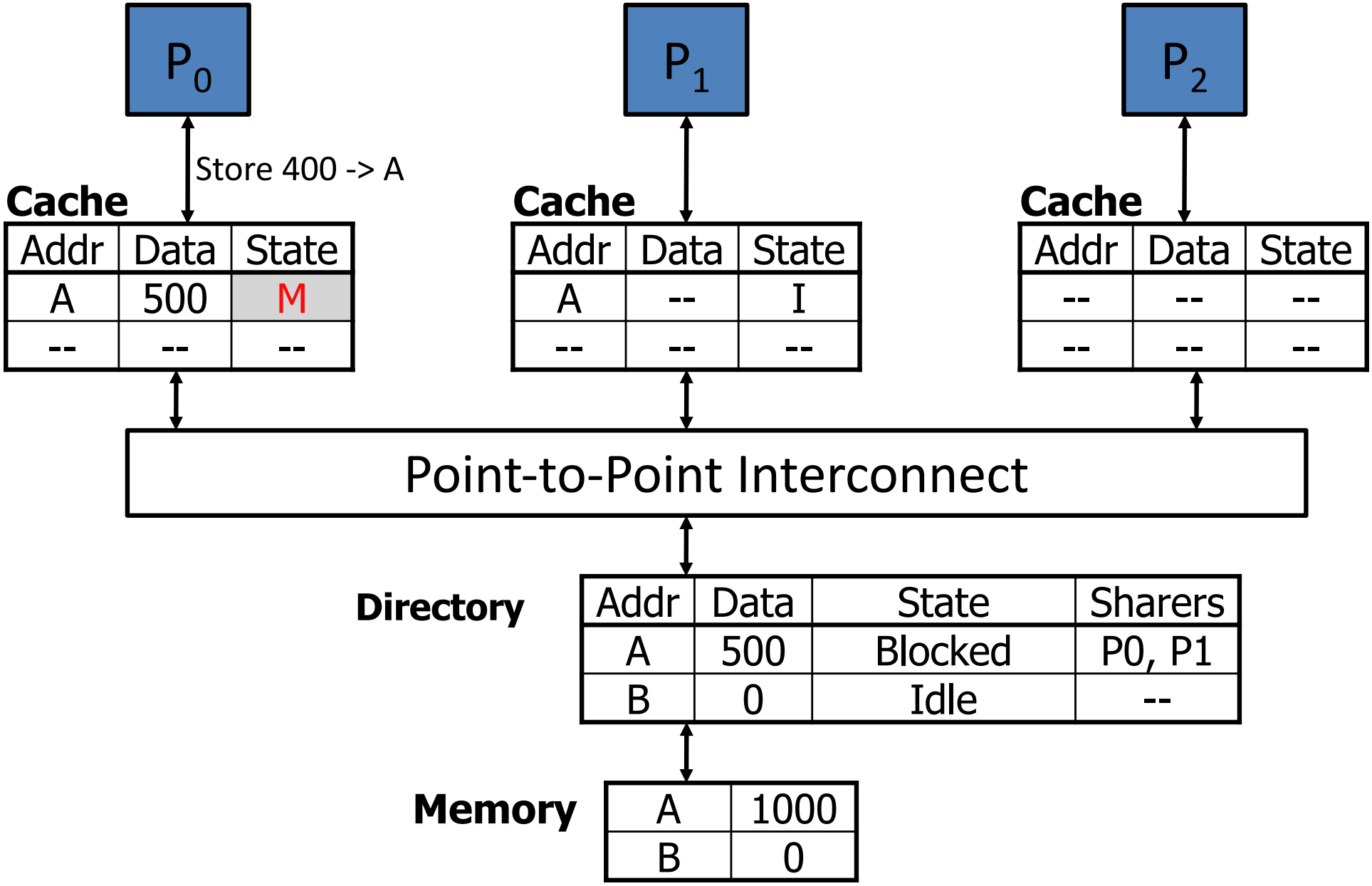
Directory Example: Step #7



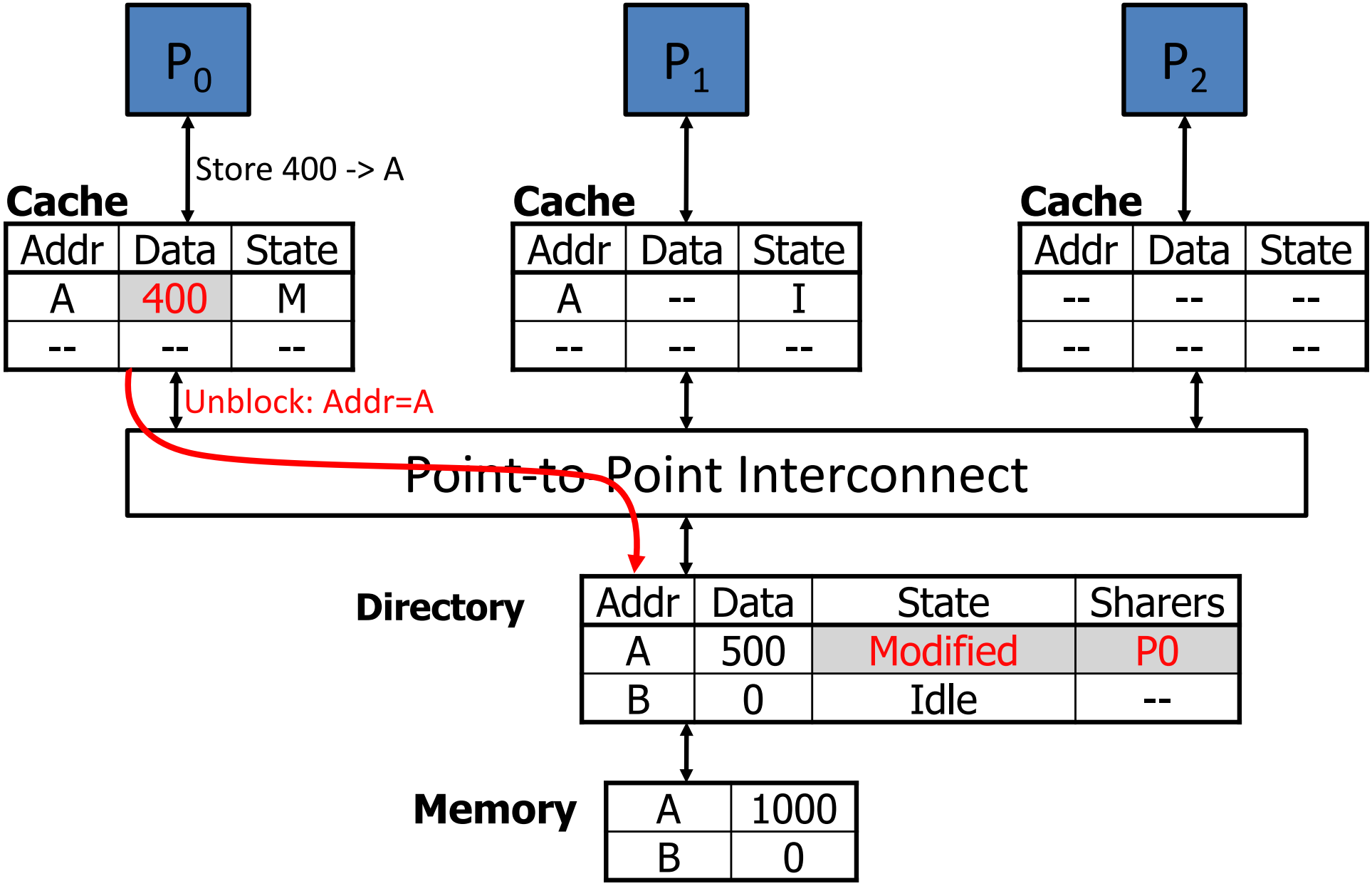
Directory Example: Step #8



Directory Example: Step #9

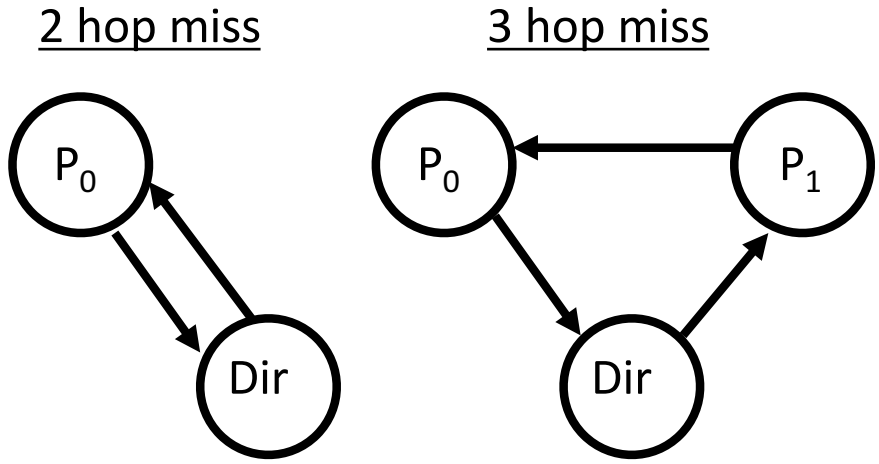


Directory Example: Step #10



Directory Flip Side: 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

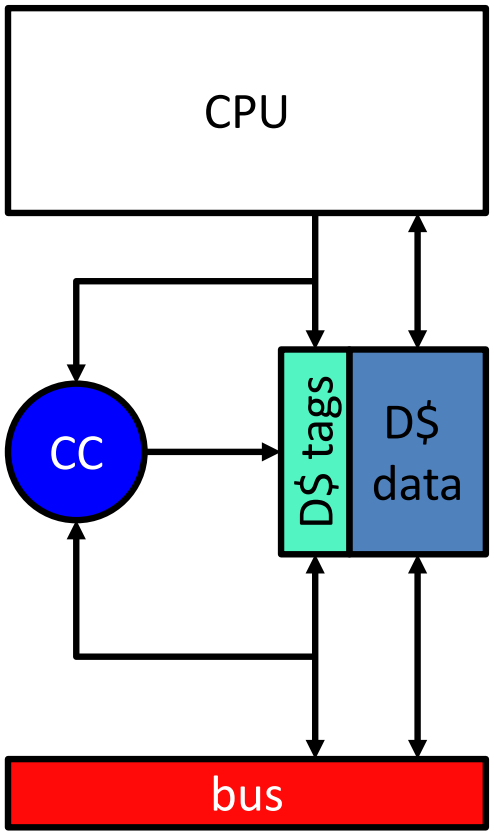
Directory Flip Side: Complexity

- Latency not only issue for directories
  - Subtle correctness issues as well
  - Stem from unordered nature of underlying inter-connect
- Individual requests to single cache must be ordered
  - Point-to-point network: requests may arrive in different orders
    - Directory has to enforce ordering explicitly
    - Cannot initiate actions on request B…
    - Until all relevant processors have completed actions on request A
    - Requires directory to collect acks, queue requests, etc.
- Directory protocols
  - Obvious in principle
  - Complicated in practice

Best of Both Worlds?

- Can we combine best features of snooping and directories?
  - From snooping: fast two-hop cache-to-cache transfers
  - From directories: scalable point-to-point networks
  - In other words…
- Can we use broadcast on an unordered network?
  - Yes, and most of the time everything is fine
  - But sometimes it isn't … **protocol race**
  - Example: IBM Power servers (ring network, no directory)

D$/I$ coherence issues?

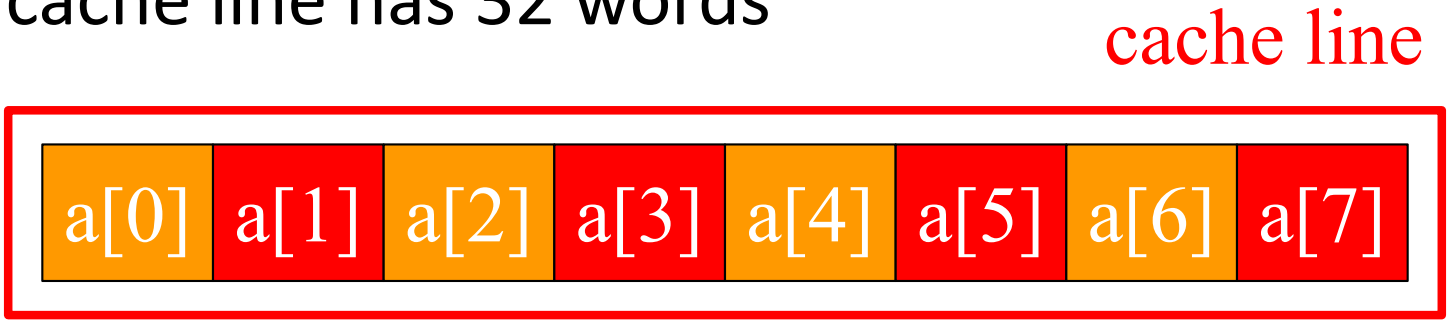


- Can D$ and I$ ever be incoherent?
- When would that be a problem?

Performance danger

```
for( i=0; i<n; i++ )
    a[i] = b[i];
```

- Let's assume we parallelize code:
  - p = 2
  - element of a takes 4 words
  - cache line has 32 words



Written by processor 0

Written by processor 1

False Sharing

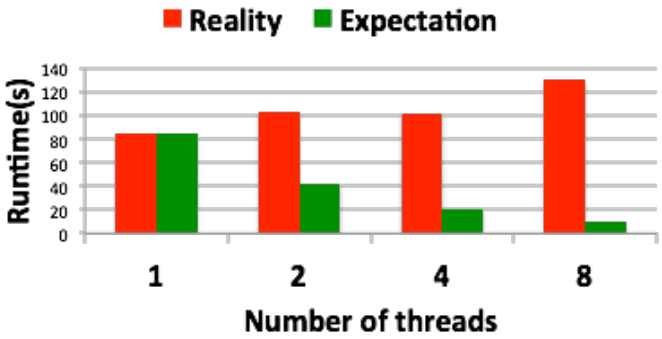
- Two or more processors sharing parts of the same block
- But not the same bytes within that block (no actual sharing)
- Creates pathological “ping-pong” behavior
- Careful data placement may help, but is difficult
  - It can happen every where and in different type: intra-object and inter-object, heap, globals.

False sharing discussion

- False sharing rate
  - Larger block?
  - Larger cache?
- Impact of false sharing
  - As miss penalty increases?
  - Traffic?
  - Overall effects on performance?
- How can we reduce it?
  - Data layout?
  - Compiler optimizations?
  - HW support?
  - Tools (e.g., Sherif from UMass)

```
int count[8]; //Global array

thread_func(int id) {
   for(i = 0; i < M; i++)
      count[id]++;
}
```



Coherence on Real Machines

- Many uniprocessors designed with on-chip snooping logic
  - Can be easily combined to form multi-processors
    - E.g., Intel Pentium4 Xeon
  - Multi-core
- Xeon Phi (MESI, directory-based)
  - 60 cores, bi-directional ring bus
  - Minor extension to support modified-shared to avoid broadcast storms
- ARM CoreLink
  - MOESI
- Larger scale (directory) systems built from smaller MPs
  - E.g., Sun Wildfire, NUMA-Q, IBM Summit
- Some shared memory machines are **not cache coherent**
  - E.g., CRAY-T3D/E
  - Shared data is un-cacheable
  - If you want to cache shared data, copy it to private data section
  - Basically, cache coherence implemented in software
    - Have to really know what you are doing as a programmer

Cache Coherence in Heterogeneous Systems

- E.g., CPU+GPU sharing address space in a cache-coherent way
- GPUs touch a lot of data, have very different access pattern
  - How to isolate traffic and still provide coherence
- Several approaches:
  - Self-invalidation
  - Traffic filtering
    - Learn from traffic what each node cares about, filter based on that
  - Region Coherence
    - Do it at a coarse grain, then fine-grain
    - A-priori configuration of what each node cares about
  - Heterogeneous System Coherence
    - Directory-based region coherence
    - Use direct access as opposed to coherent interface

DOI:10.1145/2209249.2209269

On-chip hardware coherence can scale gracefully as the number of cores increases.

BY MILO M.K. MARTIN, MARK D. HILL, AND DANIEL J. SORIN

Why On-Chip Cache Coherence Is Here to Stay

SHARED MEMORY IS the dominant low-level communication paradigm in today's mainstream multicore processors. In a shared-memory system,

Cache col
nate the m
for legacy
ware cach
formance
achievable
coherence
advantage
compatibi
ware, inc
written f
memory s

Althou
in today's
ventional
coherence
number o
on future
ence's all
es from c
and inter
and conce
Such clain
cores in fu
employ co
nicate wit
ence, exp
memories
(without s

Here,
ventional
way to sca
in which
fic storag