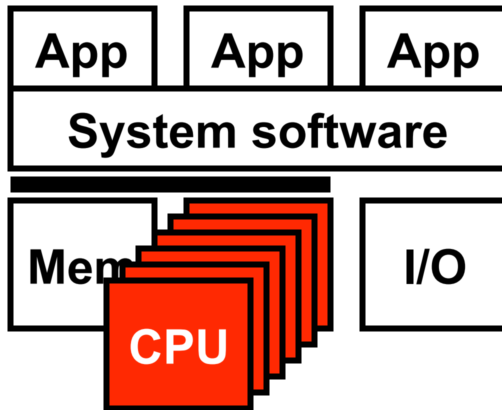
CIS 501
Computer Architecture

Unit 9: Multicore
(Shared Memory Multiprocessors)

Slides developed by Milo Martin & Amir Roth at the University of Pennsylvania
with sources that included University of Wisconsin slides
by Mark Hill, Guri Sohi, Jim Smith, and David Wood.

This Unit: Shared Memory Multiprocessors



- Thread-level parallelism (TLP)
- Shared memory model
  - Multiplexed uniprocessor
  - Hardware multihreading
  - Multiprocessing
- Synchronization
  - Lock implementation
  - Locking gotchas
- Cache coherence
  - Bus-based protocols
  - Directory protocols
- Memory consistency

Readings

- Textbook (MA:FSPTCM)
  - Sections 7.0, 7.1.3, 7.2-7.4
  - Section 8.2

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];
  }
```

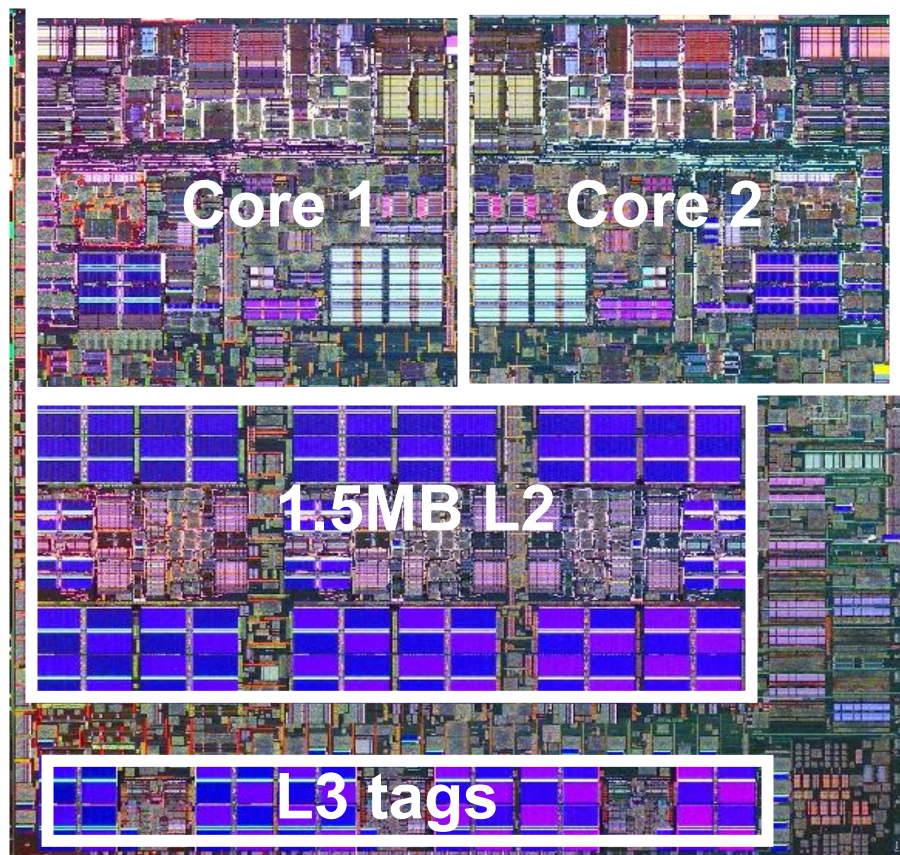
- Look familiar?
  - Hint: homework #1
- But only works if loop is actually parallel
  - If not parallel, incorrect behavior may result in unpredictable ways

Multicore & Multiprocessor Hardware

Multiplying Performance

- A single processor can only be so fast
  - Limited clock frequency
  - Limited instruction-level parallelism
  - Limited cache hierarchy
- What if we need even more computing power?
  - Use multiple processors!
  - But how?
- High-end example: Sun Ultra Enterprise 25k
  - 72 UltraSPARC IV+ processors, 1.5Ghz
  - 1024 GBs of memory
  - Niche: large database servers
  - $$$

Multicore: Mainstream Multiprocessors



Why multicore? What else would you do with 1 billion transistors?

- **Multicore chips**
- **IBM Power5**
  - Two 2+GHz PowerPC cores
  - Shared 1.5 MB L2, L3 tags
- AMD Quad Phenom
  - Four 2+ GHz cores
  - Per-core 512KB L2 cache
  - Shared 2MB L3 cache
- Intel Core i7 Quad
  - Four cores, private L2s
  - Shared 6 MB L3
- Sun Niagara
  - 8 cores, each 4-way threaded
  - Shared 2MB L2, shared FP
  - For servers, not desktop

Recall Another Reason: Energy

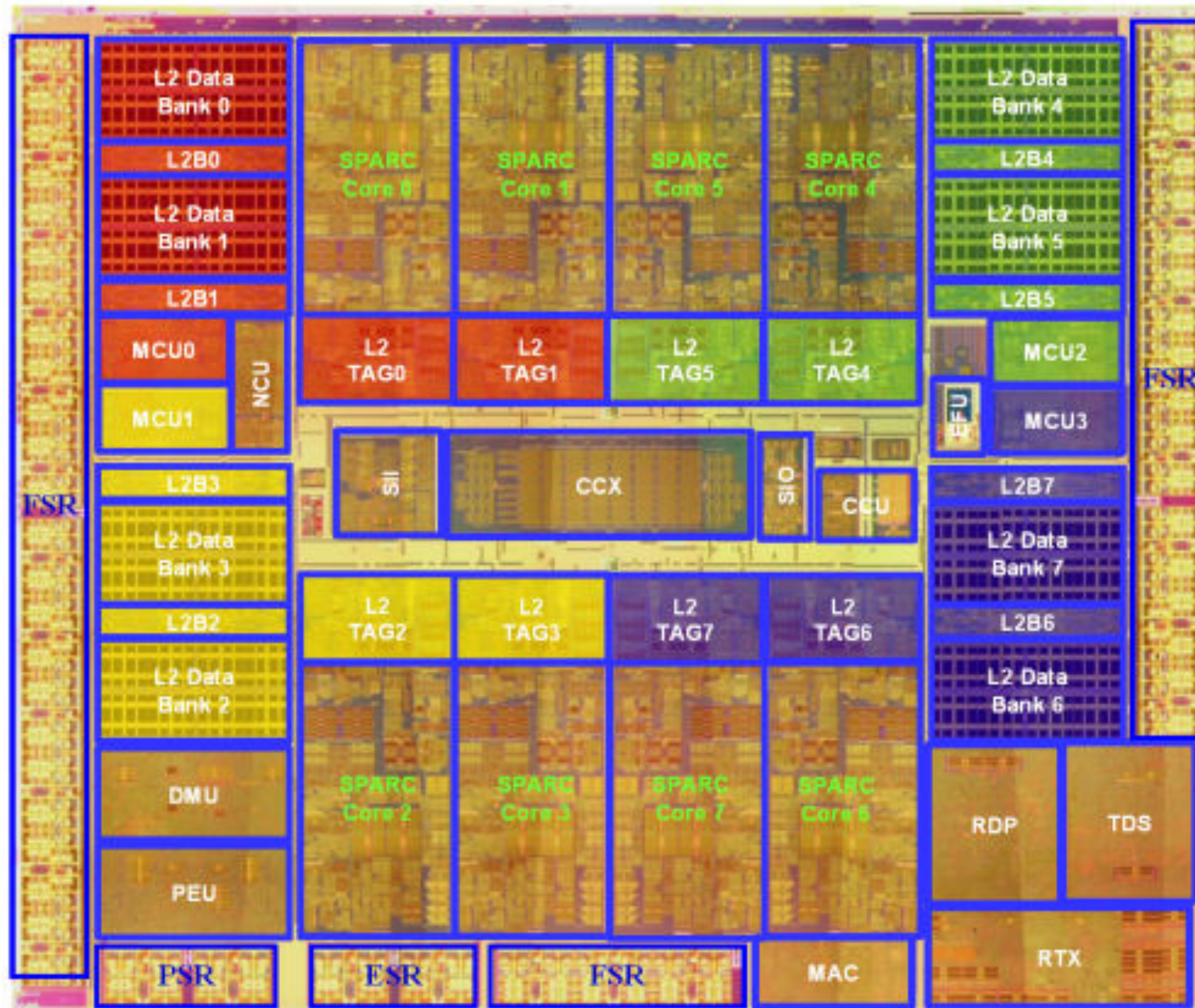
- Explicit parallelism (multicore) is highly energy efficient

- 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

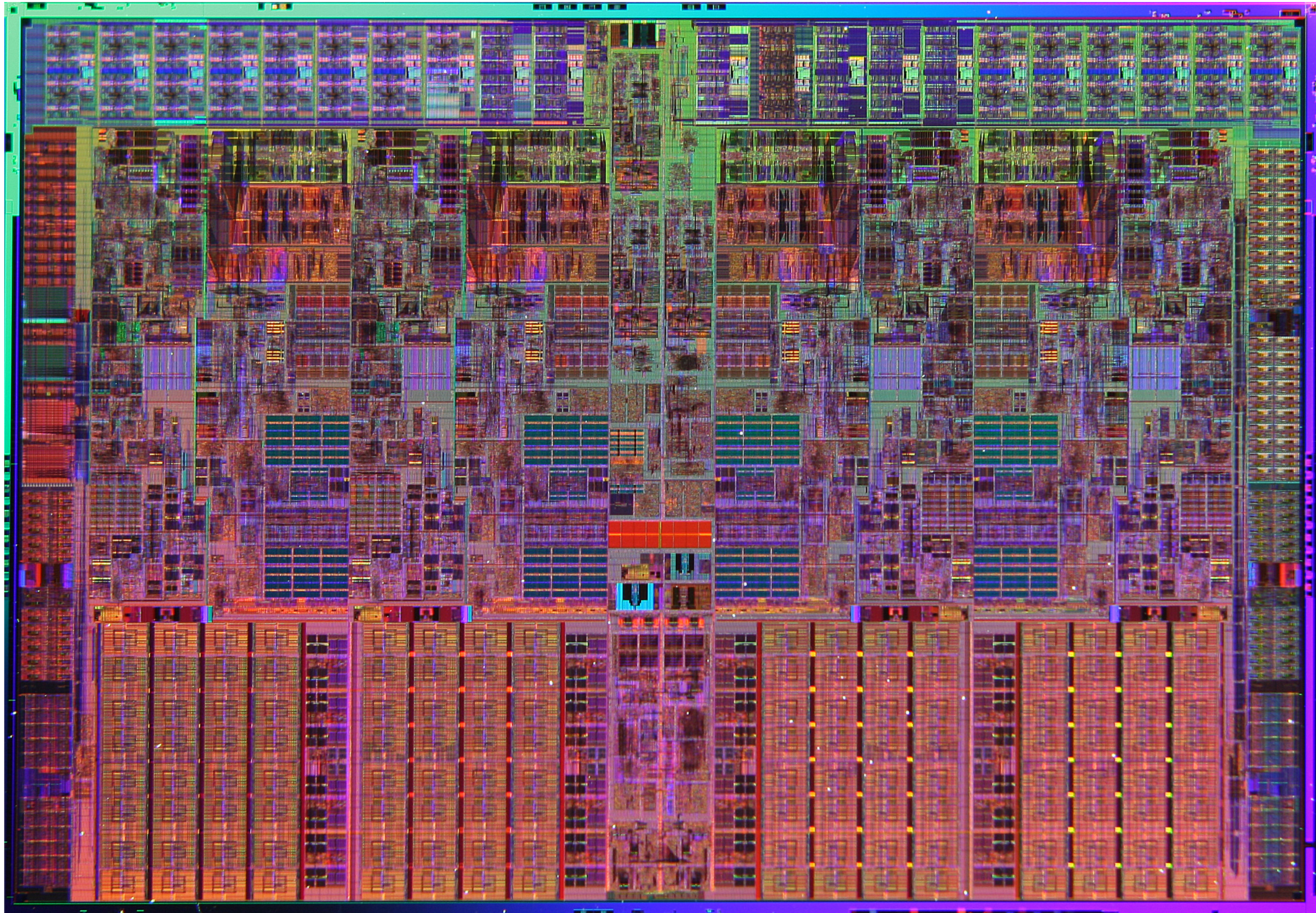
- Consider the impact of parallel execution
  - What if we used 5 Xscales at 200Mhz?
  - Similar performance as a 1Ghz Xscale, but **1/6th the energy**
    - 5 cores * 1/30th = 1/6th

- Assumes parallel speedup (a difficult task)
  - Remember Ahmdal's law

Sun Niagara II



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

“Threading” & The Shared Memory Execution Model

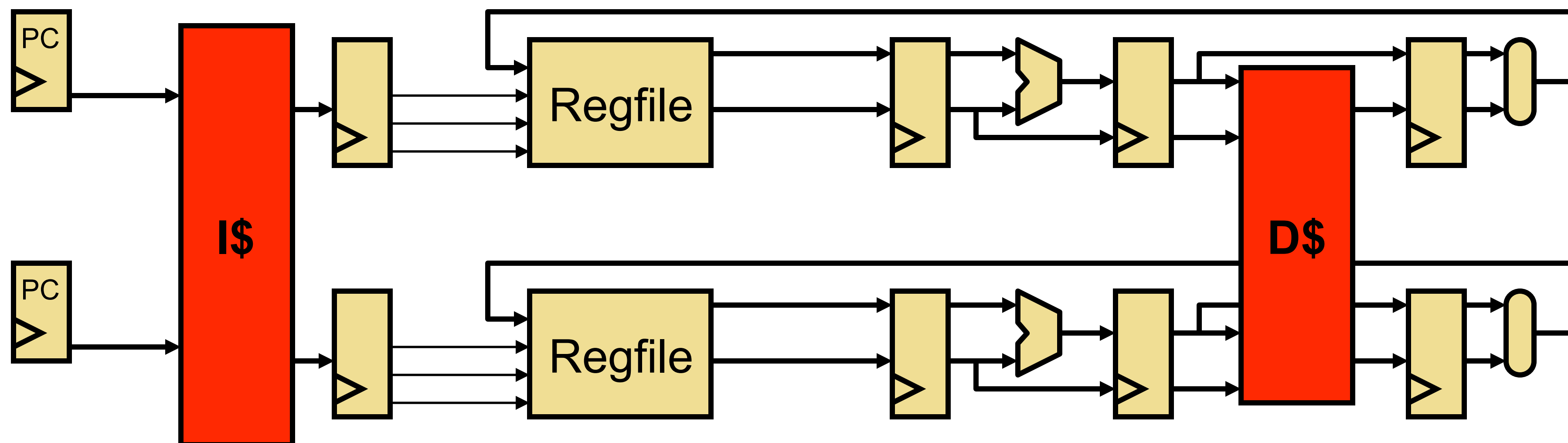
First, Uniprocessor Concurrency

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

Multithreaded Programming Model

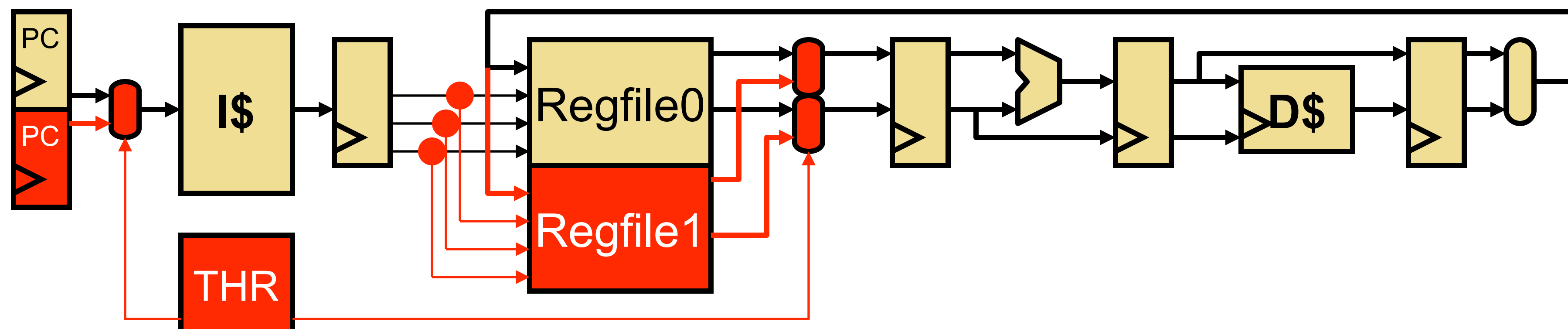
- Programmer explicitly creates multiple threads
- All loads & stores to a single **shared memory** space
  - Each thread has a private stack frame for local variables
- A "thread switch" can occur at any time
  - Pre-emptive multithreading by OS
- Common uses:
  - Handling user interaction (GUI programming)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

Simplest Multiprocessor



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

Alternative: Hardware Multithreading



- **Hardware Multithreading (MT)**
  - Multiple threads dynamically share a single pipeline
  - Replicate only per-thread structures: program counter & registers
  - Hardware interleaves instructions
  - \+ **Multithreading improves utilization and throughput**
    - Single programs utilize <50% of pipeline (branch, cache miss)
  - **Multithreading does not improve single-thread performance**
    - Individual threads run as fast or even slower
  - **Coarse-grain MT**: switch on L2 misses   Why?
  - **Simultaneous MT**: no explicit switching, fine-grain interleaving

Shared Memory Implementations

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

Four Shared Memory Issues

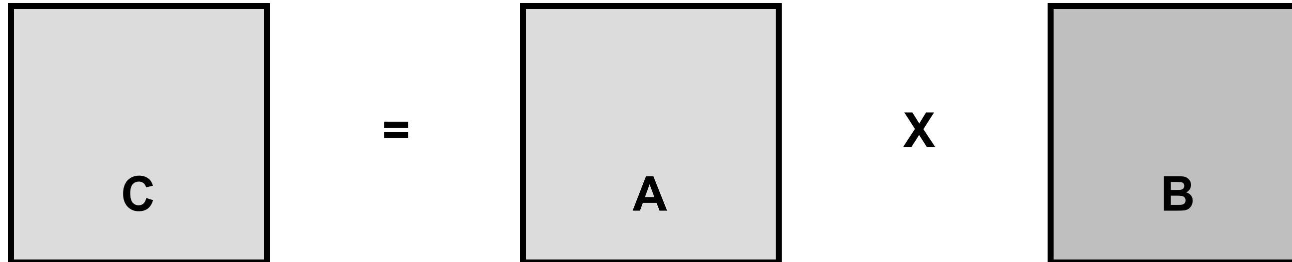
1. **Parallel programming**
   - How does the programmer express the parallelism?
2. **Synchronization**
   - How to regulate access to shared data?
   - How to implement "locks"?
3. **Cache coherence**
   - If cores have private (non-shared) caches
   - How to make writes to one cache "show up" in others?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

Parallel Programming

Parallel Programming

- One use of multiprocessors: **multiprogramming**
  - Running multiple programs with no interaction between them
  - Works great for a few cores, but what next?
- Or, programmers must **explicitly** express parallelism
  - "Coarse" parallelism beyond what the hardware can extract **implicitly**
  - Even the compiler can't extract it in most cases
- How? Several options:
  - Call libraries that perform well-known computations in parallel
    - Example: a matrix multiply routine, etc.
  - Parallel "for" loops, task-based parallelism, …
  - Add code annotations ("this loop is parallel"), OpenMP
  - Explicitly spawn "tasks", OS thread schedules them on the cores
- Parallel programming: key challenge in multicore revolution

Example: Parallelizing Matrix Multiply




```
for (I = 0; I < SIZE; I++)
  for (J = 0; J < SIZE; J++)
    for (K = 0; K < SIZE; K++)
      C[I][J] += A[I][K] * B[K][J];
```

- How to parallelize matrix multiply?
  - Replace outer "for" loop with "**parallel_for**" or OpenMP annotation
  - Supported by many parallel programming environments
- Implementation: give each of N processors loop iterations

```
int start = (SIZE/N) * my_id();
for (I = start; I < start + SIZE/N; I++)
  for (J = 0; J < SIZE; J++)
    for (K = 0; K < SIZE; K++)
      C[I][J] += A[I][K] * B[K][J];
```

- Each processor runs copy of loop above
  - Library provides `my_id()` function

Example: 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 these “debit” operations in parallel?
  - Does the order matter?

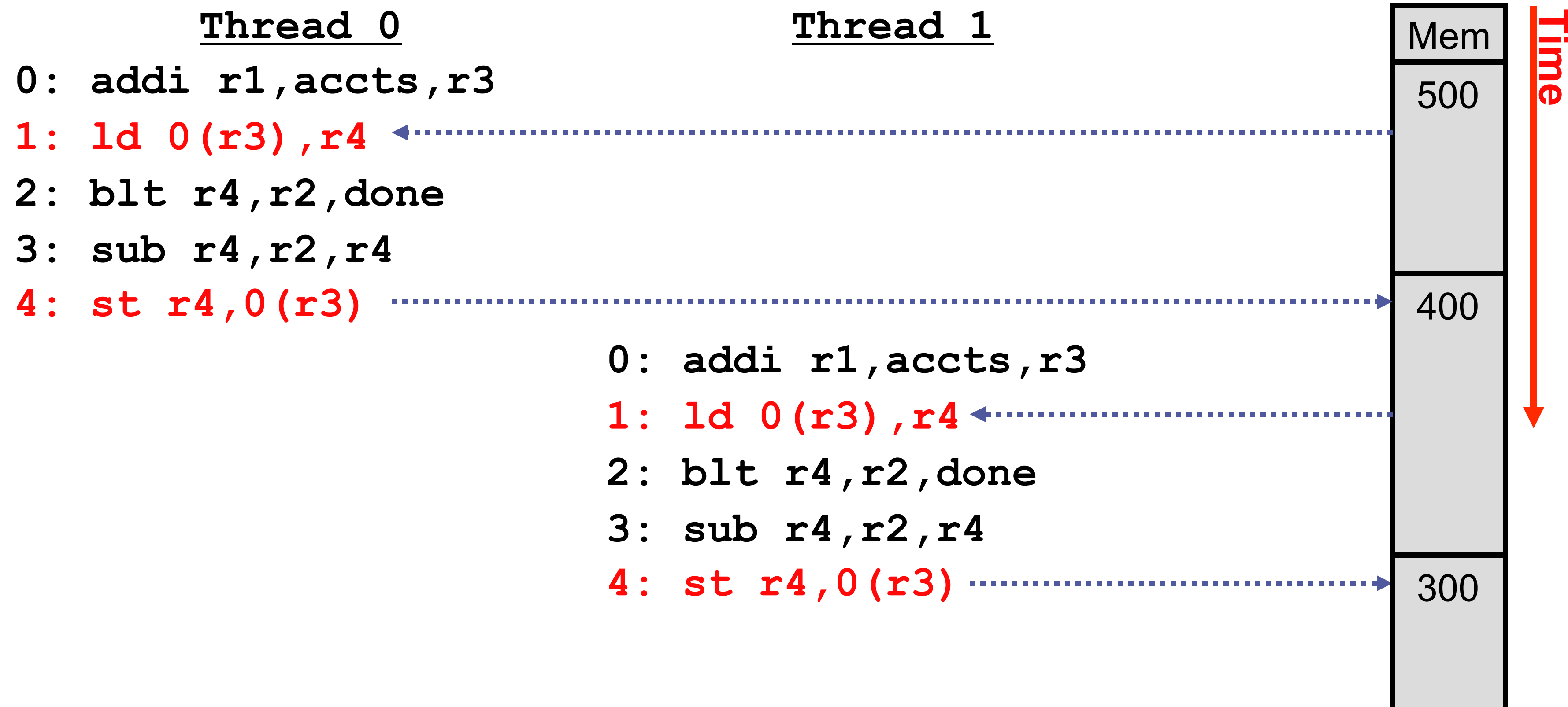
Example: Bank Accounts

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

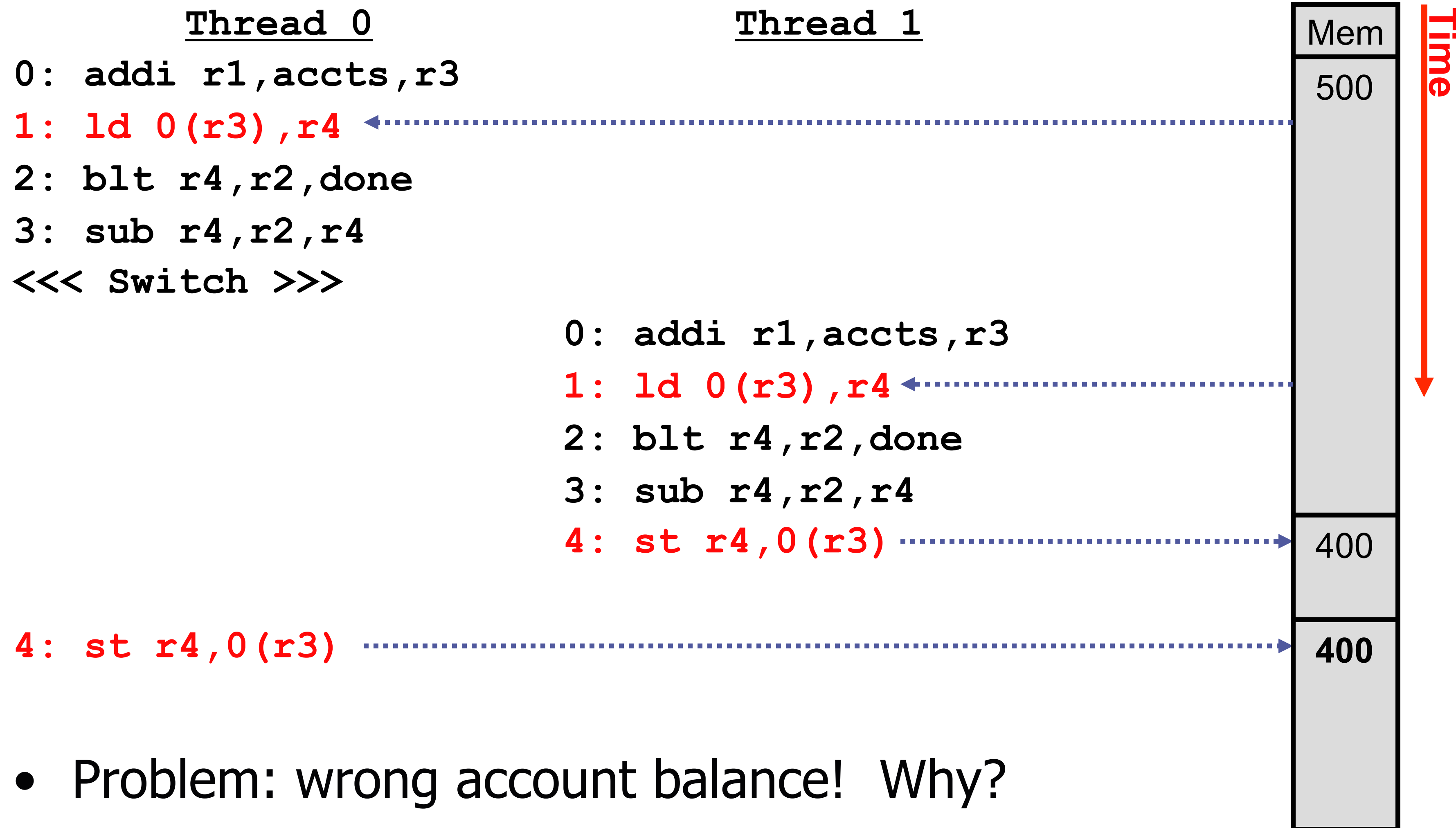
- Example of **Thread-level parallelism (TLP)**
  - Collection of asynchronous tasks: not started and stopped together
  - Data shared "loosely" (sometimes yes, mostly no), dynamically
- Example: database/web server (each query is a thread)
  - `accts` is global and thus **shared**, can't register allocate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

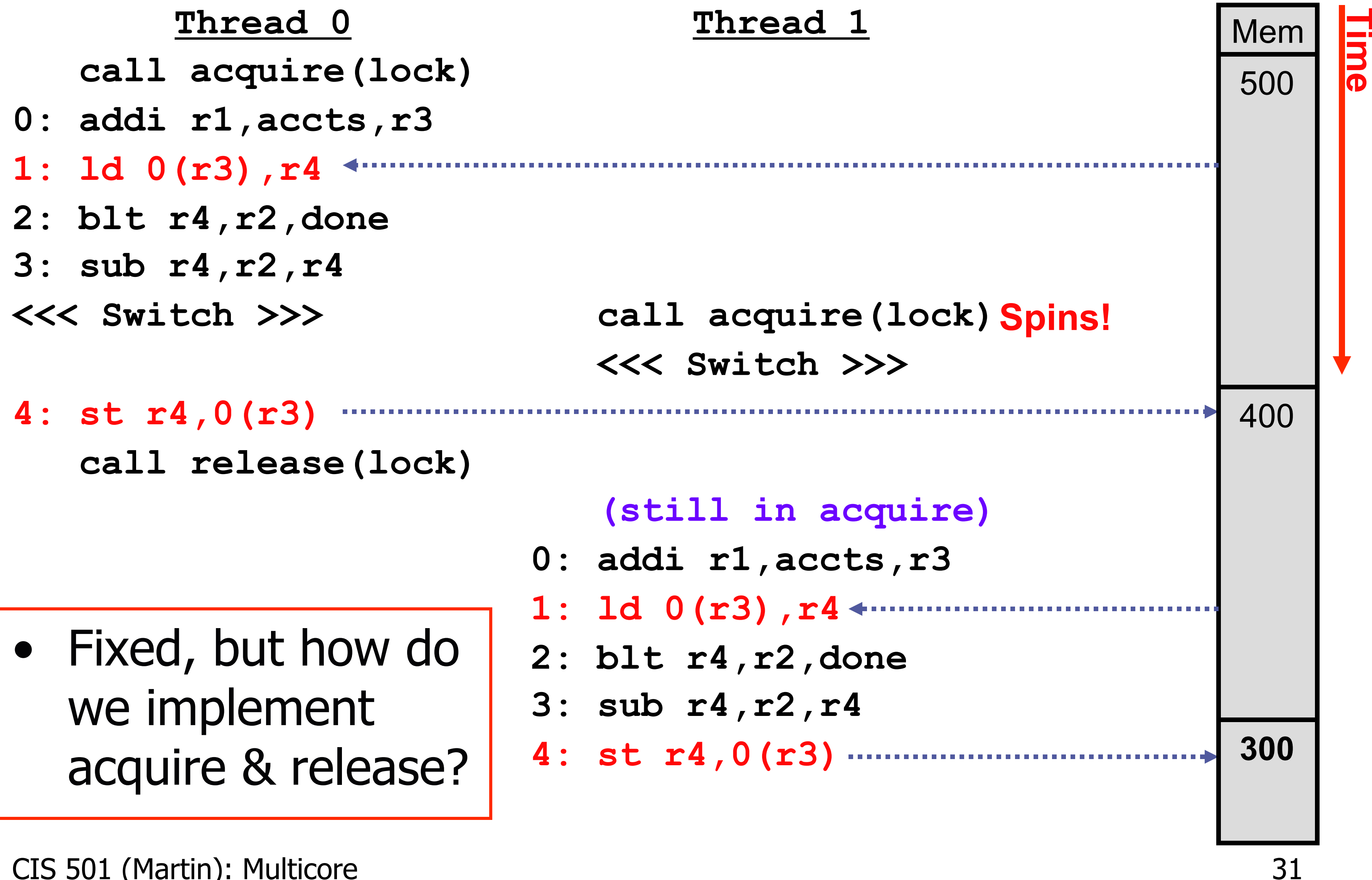
Synchronization:

- **Synchronization**: a key issue for shared memory
- Regulate access to shared data (mutual exclusion)
- Low-level primitive: **lock** (higher-level: “semaphore” or “mutex”)
  - 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)

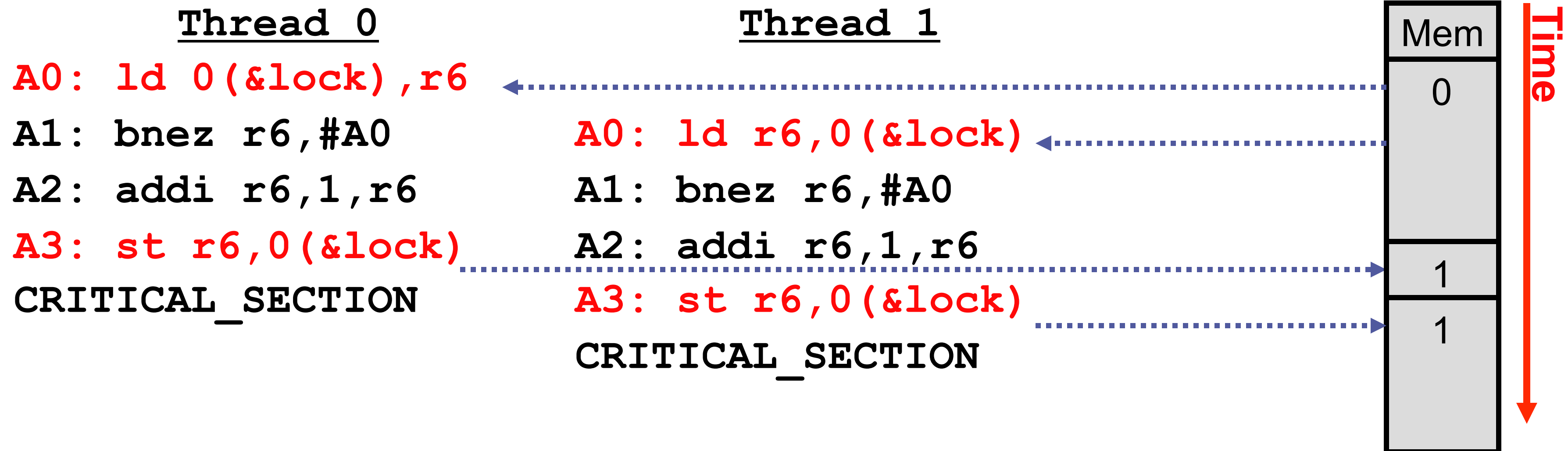
- **Spin lock**: software lock implementation
  - `acquire(lock): while (lock != 0) {} lock = 1;`
    - "Spin" while lock is 1, wait for it to turn 0

```
A0:  ld 0(&lock),r6
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
```

Strawman Lock (Incorrect)



- 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

A Correct Implementation: SYSCALL Lock

```
ACQUIRE_LOCK:
A1: disable_interrupts            atomic
A2: ld r6,0(&lock)
A3: bnez r6,#A0
A4: addi r6,1,r6
A5: st r6,0(&lock)
A6: enable_interrupts
A7: return
```

- Implement lock in a SYSCALL
  - Only kernel can control interleaving by disabling interrupts
  - \+ Works…
  - – Large system call overhead
  - – **But not in a hardware multithreading or a multiprocessor…**

Better Spin Lock: Use Atomic Swap

- ISA provides an atomic lock acquisition instruction
  - Example: **atomic swap**

    ```
    swap r1,0(&lock)
    ```

    - Atomically executes:

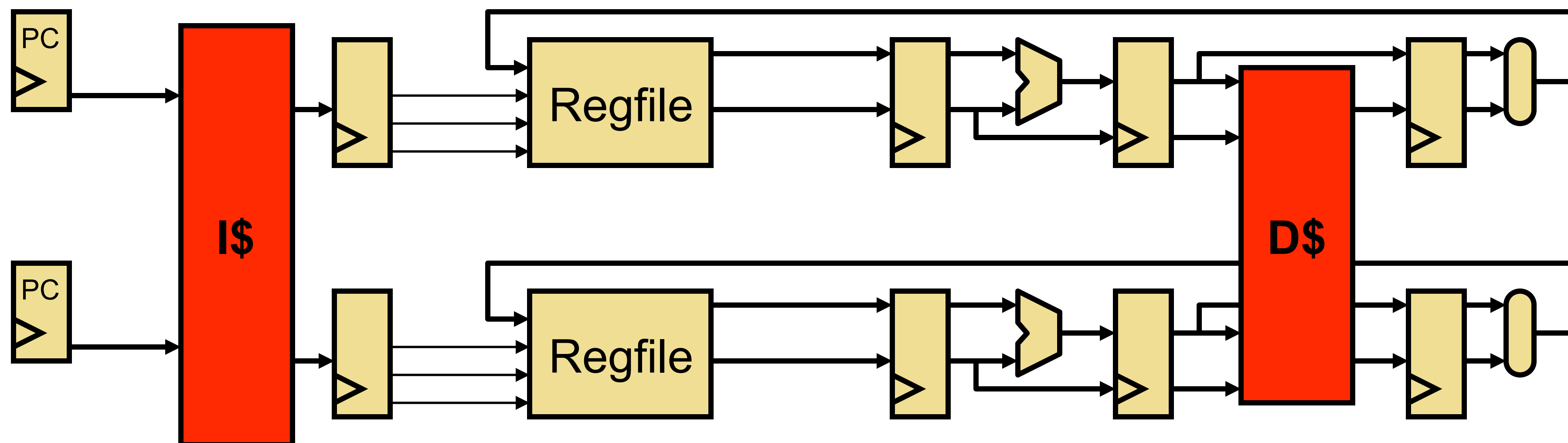
    ```
    mov r1->r2
    ld r1,0(&lock)
    st r2,0(&lock)
    ```

- New acquire sequence

  ```
  (value of r1 is 1)
  A0: swap r1,0(&lock)
  A1: bnez r1,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
- Insures lock held by **at most one thread**
  - Other variants: **exchange**, **compare-and-swap**, **test-and-set (t&s)**, or **fetch-and-add**

Atomic Update/Swap Implementation



- How is atomic swap implemented?
  - Need to ensure no intervening memory operations
  - Requires blocking access by other threads temporarily (yuck)
- How to pipeline it?
  - Both a load and a store (yuck)
  - Not very RISC-like

RISC Test-And-Set

- **`swap`**: a load and 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                      Thread 1
A0: swap r1,0(&lock)
A1: bnez r1,#A0           A0: swap r1,0(&lock)
CRITICAL_SECTION          A1: bnez r1,#A0
                          A0: swap r1,0(&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: swap r1,0(&lock)
A1: bnez r1,#A0          A0: swap r1,0(&lock)
A0: swap r1,0(&lock)     A1: bnez r1,#A0
A1: bnez r1,#A0          A0: swap r1,0(&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 `swap`, each of which includes a `st`
    - – Repeated stores by multiple processors costly (more in a bit)
    - – 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: swap r1,0(&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 **`swap`** 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 "easy" (not really)
  - Writing programs that are highly parallel is "easy" (not really)
  - **Writing programs that are both correct and parallel is difficult**
    - 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
  – Difficult to make correct: 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 waits for lock 37
  - Thread 1 has lock 37 waits for lock 241
  - Obviously this is a problem
  - The solution is …

Correct Multiple Lock Program

- **Always acquire multiple locks in same order**
  - Just 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                            Thread 1

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

acquire(accts[37].lock);            // wait to acquire lock 37
acquire(accts[241].lock);           // waiting…
// do stuff                         // …
release(accts[241].lock);           // …
release(accts[37].lock);            // …
                                    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 a slow atomic instructions
    - Specifically, acquiring write permissions to the lock
  - Ordering constraints (see soon) make it even slower

- **…and 99% of the time un-necessary**
  - Most concurrent actions don't actually share data
  - You paying 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

Research: Transactional Memory (TM)

- **Transactional Memory**
  + Programming simplicity of coarse-grain locks
  + Higher concurrency (parallelism) of fine-grain locks
    - Critical sections only serialized if data is actually shared
  + No lock acquisition overhead
  - Hottest thing since sliced bread (or was a few years ago)
  - No fewer than nine research projects:
    - Brown, Stanford, MIT, Wisconsin, Texas, Rochester, Sun/Oracle, Intel
    - Penn, too

Transactional Memory: The Big Idea

- Big idea I: **no locks, just shared data**
- Big idea II: **optimistic (speculative) concurrency**
  - Execute critical section speculatively, abort on conflicts
  - "Better to beg for forgiveness than to ask for permission"

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

Transactional Memory: Read/Write Sets

- **Read set**: set of shared addresses critical section reads
  - Example: **`accts[37].bal`**, **`accts[241].bal`**
- **Write set**: set of shared addresses critical section writes
  - Example: **`accts[37].bal`**, **`accts[241].bal`**

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

Transactional Memory: Begin

- **`begin_transaction`**
  - Take a local register checkpoint
  - Begin locally tracking read set (remember addresses you read)
    - See if anyone else is trying to write it
  - Locally buffer all of your writes (invisible to other processors)
  - \+ **Local actions only: no lock acquire**

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

Transactional Memory: End

- **end_transaction**
  - Check read set: is all data you read still valid (i.e., no writes to any)
  - Yes? Commit transactions: commit writes
  - No? Abort transaction: restore checkpoint

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

Transactional Memory Implementation

- How are read-set/write-set implemented?
  - Track locations accessed using bits in the cache

- Read-set: additional "transactional read" bit per block
  - Set on reads between begin_transaction and end_transaction
  - Any other write to block with set bit ➔ triggers abort
  - Flash cleared on transaction abort or commit

- Write-set: additional "transactional write" bit per block
  - Set on writes between begin_transaction and end_transaction
  - Before first write, if dirty, initiate writeback ("clean" the block)
  - Flash cleared on transaction commit
  - On transaction abort: blocks with set bit are invalidated

Transactional Execution

Thread 0

```
id_from = 241;
id_to = 37;

begin_transaction();
if(accts[241].bal > 100) {
   …
   // write accts[241].bal
   // abort
```

Thread 1

```
id_from = 37;
id_to = 241;

begin_transaction();
if(accts[37].bal > 100) {
   accts[37].bal -= amt;
   acts[241].bal += amt;
}
end_transaction();
// no writes to accts[241].bal
// no writes to accts[37].bal
// commit
```

Transactional Execution II (More Likely)

Thread 0

```
id_from = 241;
id_to = 37;

begin_transaction();
if(accts[241].bal > 100) {
   accts[241].bal -= amt;
   acts[37].bal += amt;
}
end_transaction();
// no write to accts[240].bal
// no write to accts[37].bal
// commit
```

Thread 1

```
id_from = 450;
id_to = 118;

begin_transaction();
if(accts[450].bal > 100) {
   accts[450].bal -= amt;
   acts[118].bal += amt;
}
end_transaction();
// no write to accts[450].bal
// no write to accts[118].bal
// commit
```

- Critical sections execute in parallel

So, Let's Just Do Transactions?

- What if…
  - Read-set or write-set bigger than cache?
  - Transaction gets swapped out in the middle?
  - Transaction wants to do I/O or SYSCALL (not-abortable)?
- How do we transactify existing lock based programs?
  - Replace **`acquire`** with **`begin_trans`** does not always work
- Several different kinds of transaction semantics
  - Are transactions atomic relative to code outside of transactions?
- Do we want transactions in hardware or in software?
  - What we just saw is **hardware transactional memory (HTM)**
- That's what these research groups are looking at
  - Best-effort hardware TM: Azul systems, Sun's Rock processor

Speculative Lock Elision

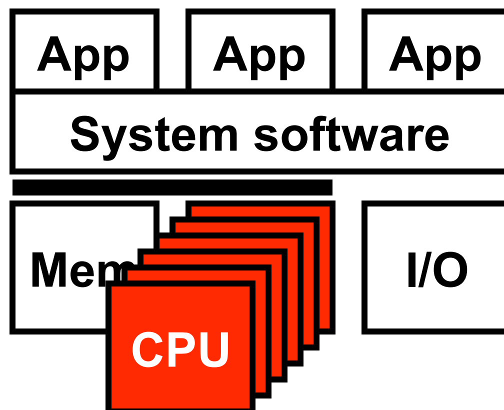
```
Processor 0

acquire(accts[37].lock); // don't actually set lock to 1
// begin tracking read/write sets
// CRITICAL_SECTION
// check read set
//  no conflicts? Commit, don't actually set lock to 0
//  conflicts? Abort, retry by acquiring lock
release(accts[37].lock);
```

- Alternatively, keep the locks, but…
- … speculatively transactify lock-based programs in hardware
  - **Speculative Lock Elision (SLE)** [Rajwar+, MICRO'01]
    - Captures most of the advantages of transactional memory…

  + No need to rewrite programs

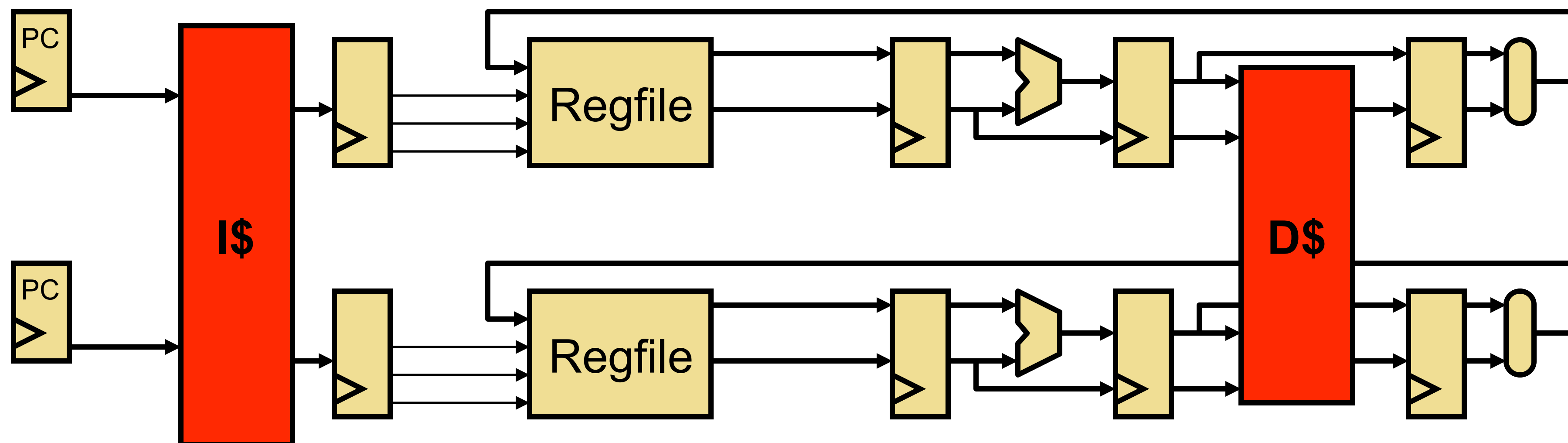
  + Can always fall back on lock-based execution (overflow, I/O, etc.)

Roadmap Checkpoint



- Thread-level parallelism (TLP)
- Shared memory model
  - Multiplexed uniprocessor
  - Hardware multihreading
  - Multiprocessing
- Synchronization
  - Lock implementation
  - Locking gotchas
- Cache coherence
  - Bus-based protocols
  - Directory protocols
- Memory consistency models

Recall: Simplest Multiprocessor

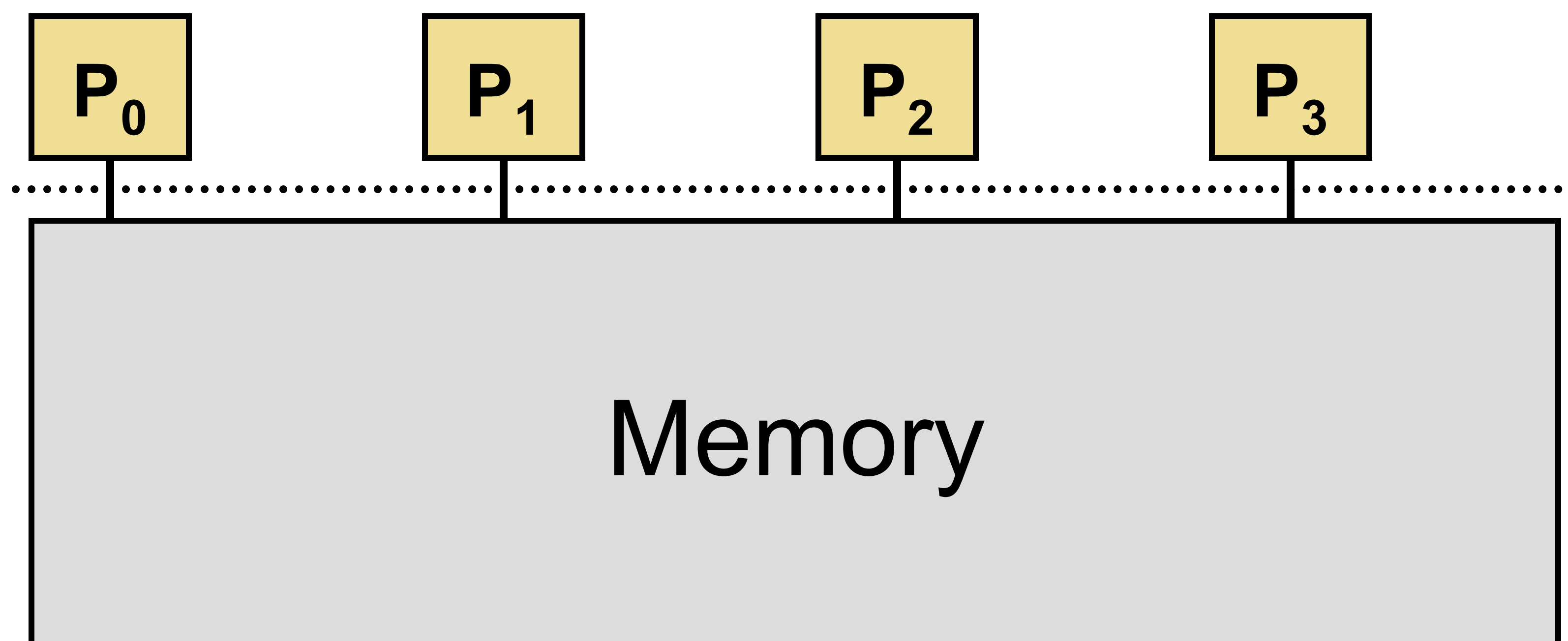


- What if we don't want to share the L1 caches?
  - Bandwidth and latency issue

- Solution: use per-processor ("private") caches
  - Coordinate them with a ***Cache Coherence Protocol***

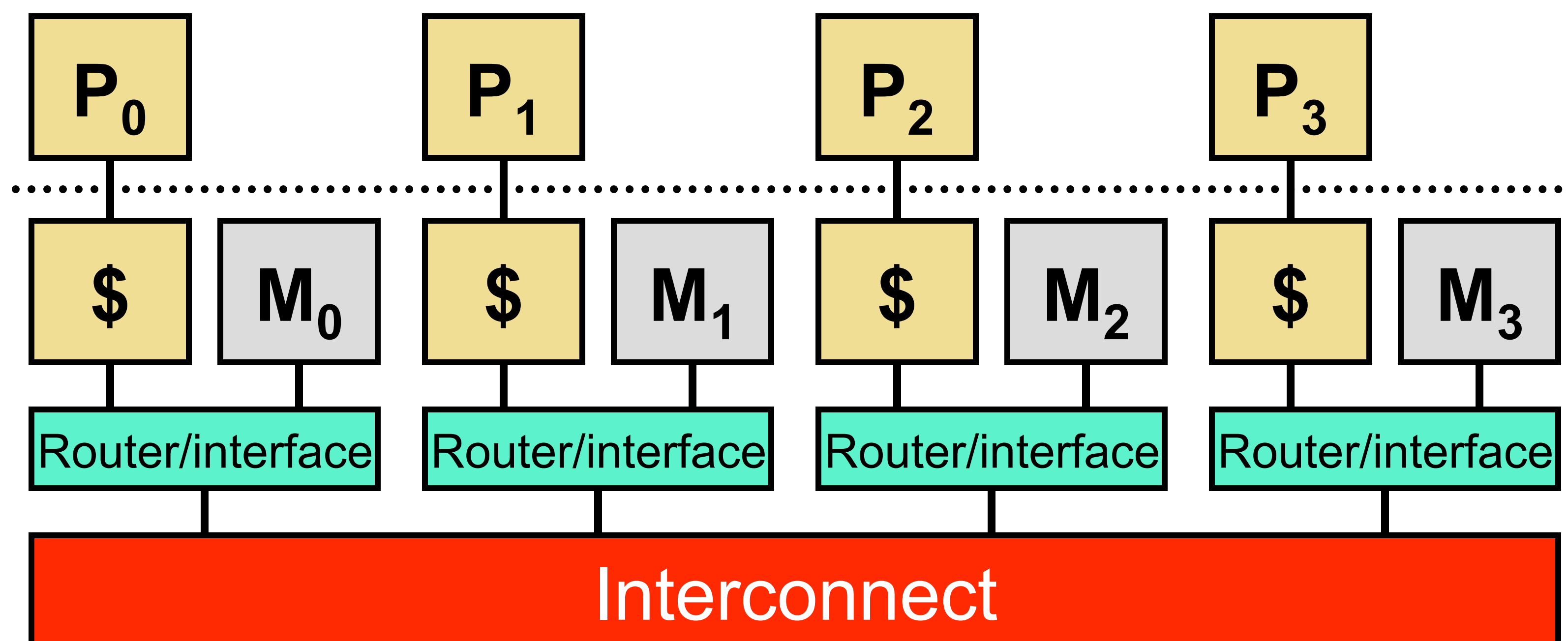
Shared-Memory Multiprocessors

- **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



Revisiting Our Motivating Example

CPU0 | CPU1 | Mem

Processor 0

```
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)
```

critical section (locks not shown)

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)
```

critical section (locks not shown)

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

No-Cache, No-Problem

```
Processor 0
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)
```

```
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)
```

CPU0	CPU1	Mem
		$500
		$500
		$400
		$400
		$300

- Scenario I: processors have no caches
  - No problem

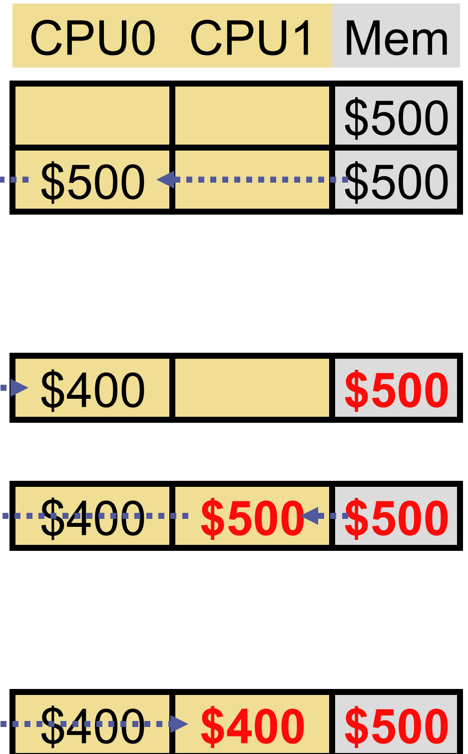
Cache Incoherence

Processor 0

```
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)
```

Processor 1

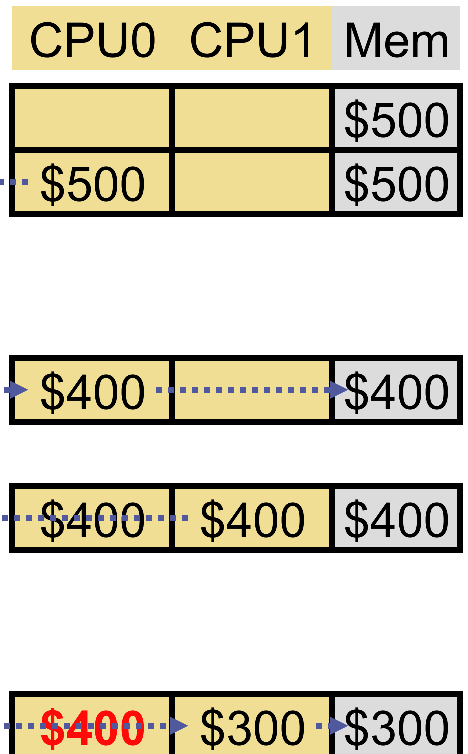
```
0: addi $r3,$r1,&accts
1: lw $r4,0($r3)
2: blt $r4,$r2,6
3: sub $r4,$r4,$r2
4: sw $r4,0($r3)
```



- Scenario II(a): processors have write-back caches
  - Potentially 3 copies of `accts[241].bal`: memory, two caches
  - Can get incoherent (inconsistent)

Write-Through Doesn't Fix It

```
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)
                               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)
```



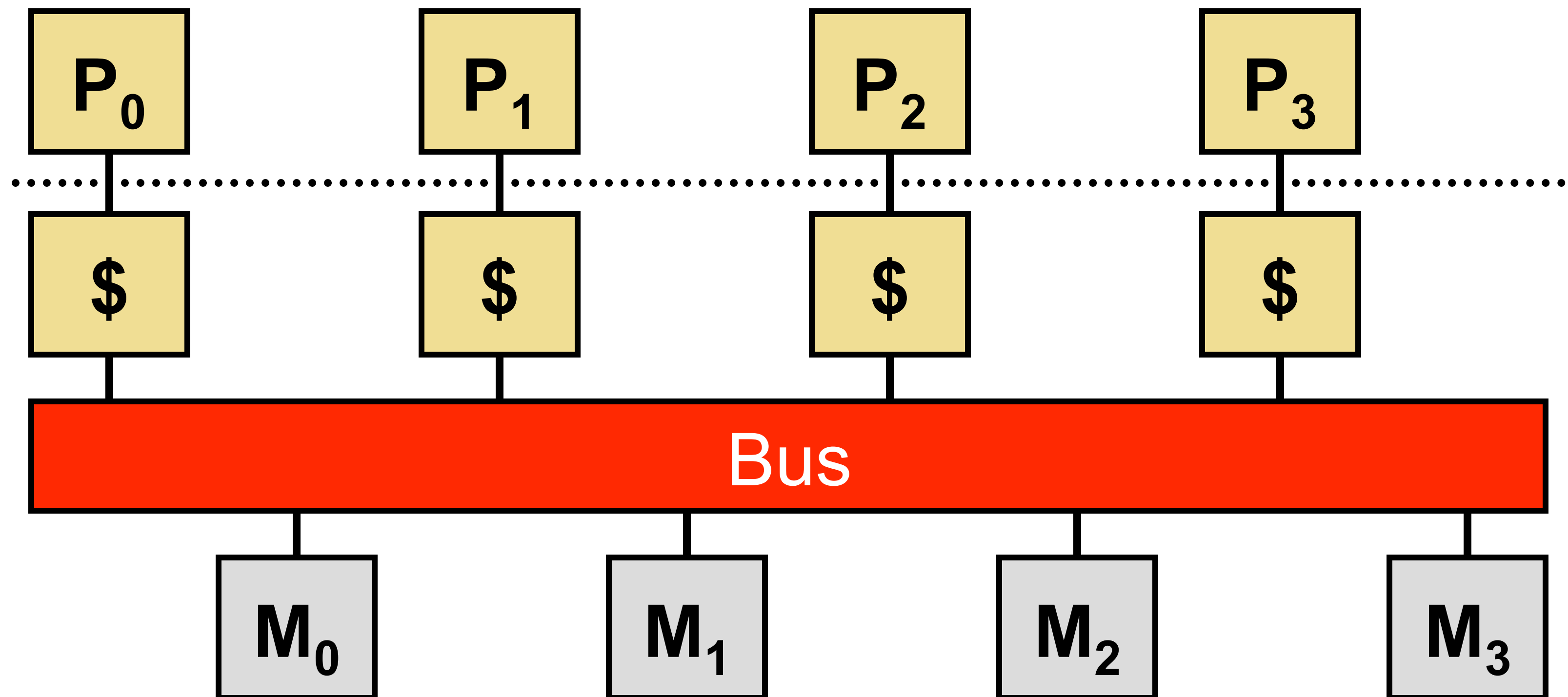
- Scenario II(b): processors have write-through caches
  - This time only two (different) copies of `accts[241].bal`
  - No problem? What if another withdrawal happens on processor 0?

What To Do?

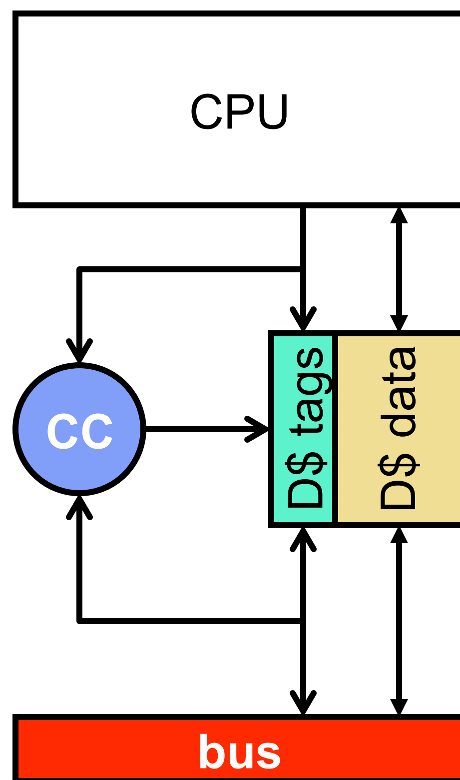
- No caches?
  - – Too slow
- Make shared data uncachable?
  - – Faster, but still too slow
  - Entire **`accts`** database is technically "shared"
- Flush all other caches on writes to shared data?
  - Can work well in some cases, but can make caches ineffective
- **Hardware cache coherence**
  - Rough goal: all caches have same data at all times
  - \+ Minimal flushing, maximum caching → best performance

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

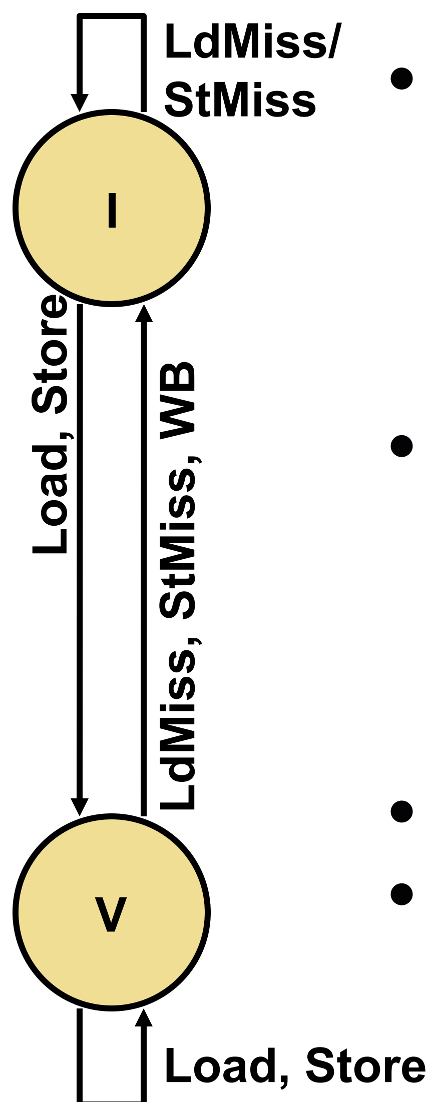


Hardware Cache Coherence



- **Coherence**
  - all copies have same data at all times
- **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
- **Each processors runs a state machine**
- Three processor-initiated events
  - **Ld**: load **St**: store **WB**: write-back
- Two remote-initiated events
  - **LdMiss**: read miss from ***another*** processor
  - **StMiss**: write miss from ***another*** processor

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 & 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

VI Protocol State Transition Table

	This Processor		*Other Processor*	
State	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**

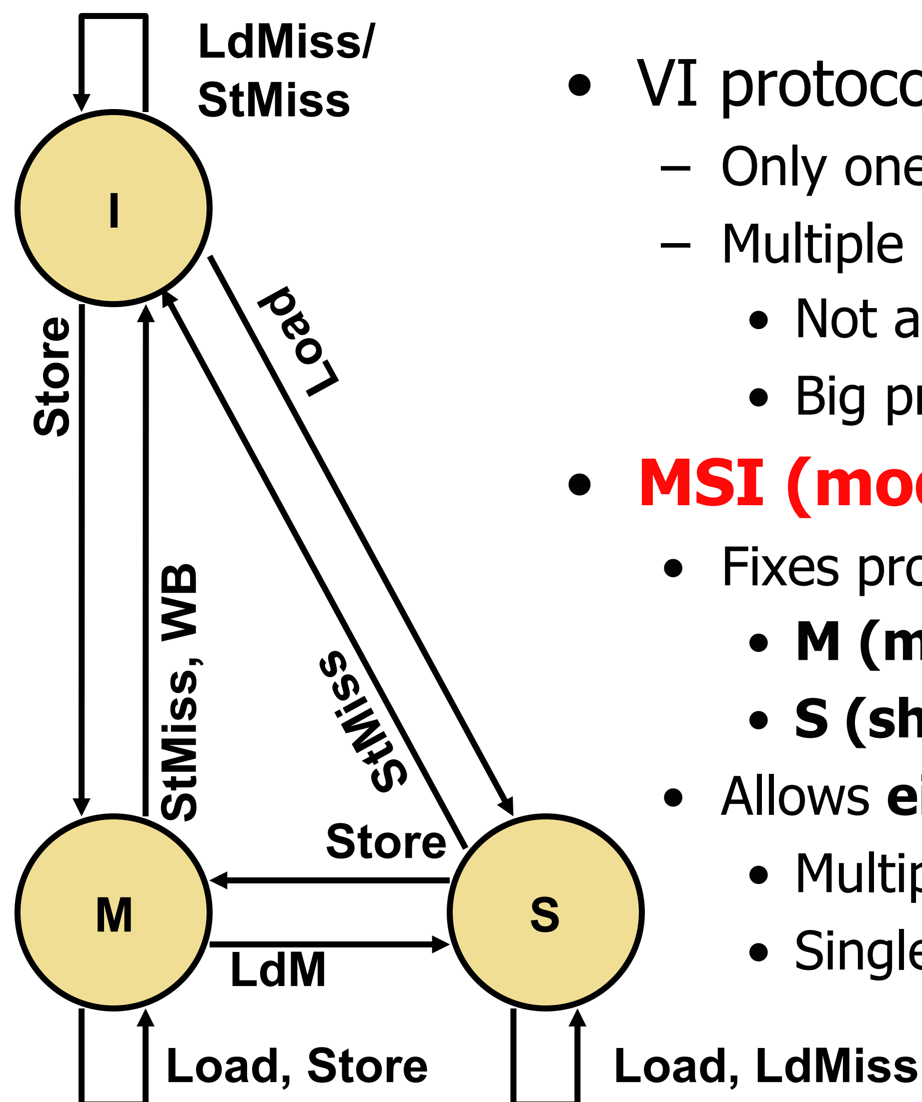
VI Protocol (Write-Back Cache)

```
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)
                                 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)
```

CPU0	CPU1	Mem
		500
V:500		500
V:400		500
I:	V:400	400
	V:300	400

- `lw` by processor 1 generates an "other load miss" event (LdMiss)
  - Processor 0 responds by sending its dirty copy, transitioning to **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)

MSI Protocol State Transition Table

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

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

MSI Protocol (Write-Back Cache)

```
Processor 0
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)
```

```
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)
```

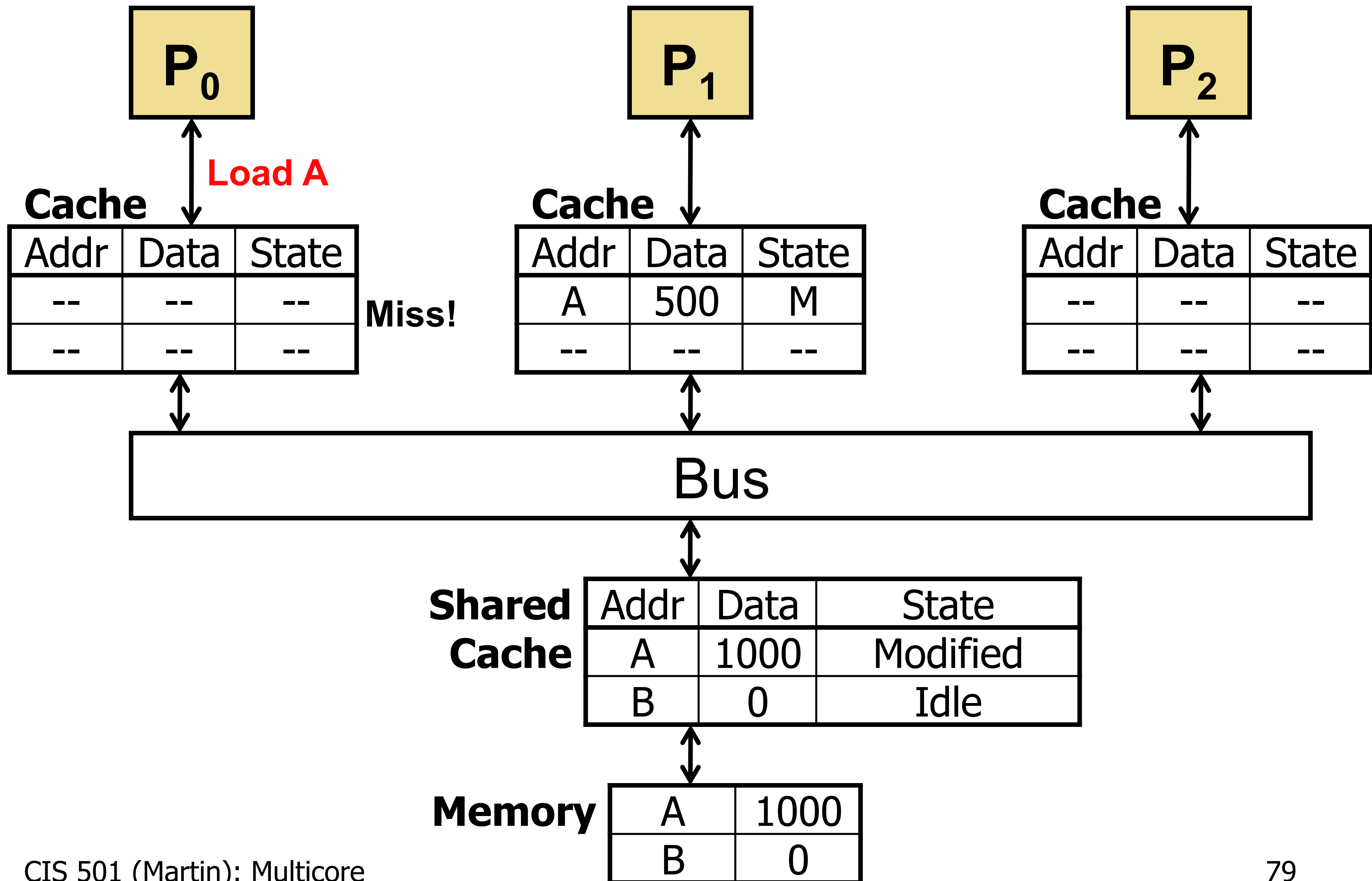
CPU0	CPU1	Mem
		500
S:500		500
M:400		**500**
S:400	S:400	400
I:	M:300	**400**

- `lw` by processor 1 generates a "other load miss" event (LdMiss)
  - Processor 0 responds by sending its dirty copy, transitioning to **S**
- `sw` by processor 1 generates a "other store miss" event (StMiss)
  - Processor 0 responds by transitioning to **I**

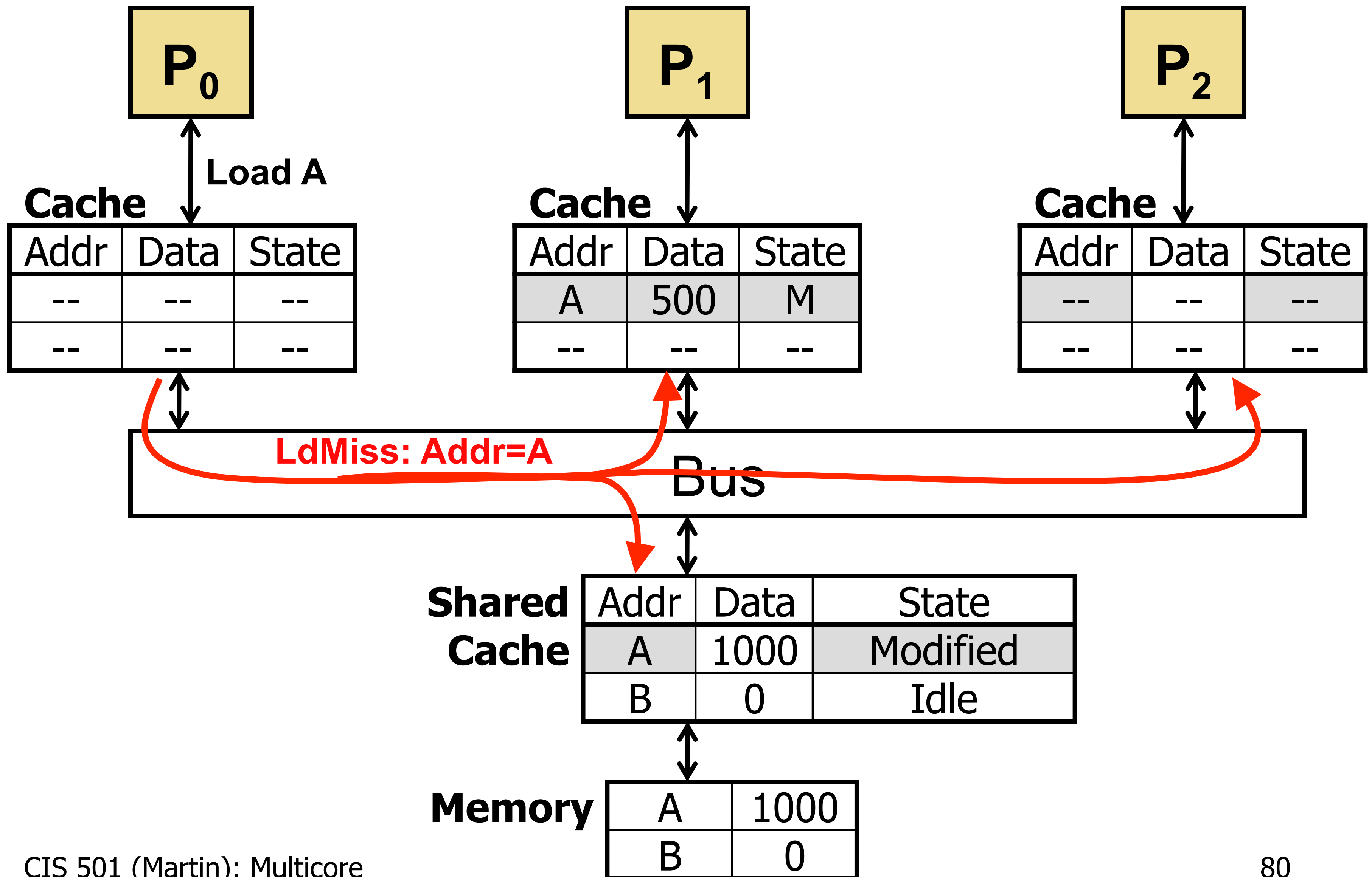
Cache Coherence and Cache Misses

- 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
- Making the cache larger…
  - Doesn't reduce these type 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

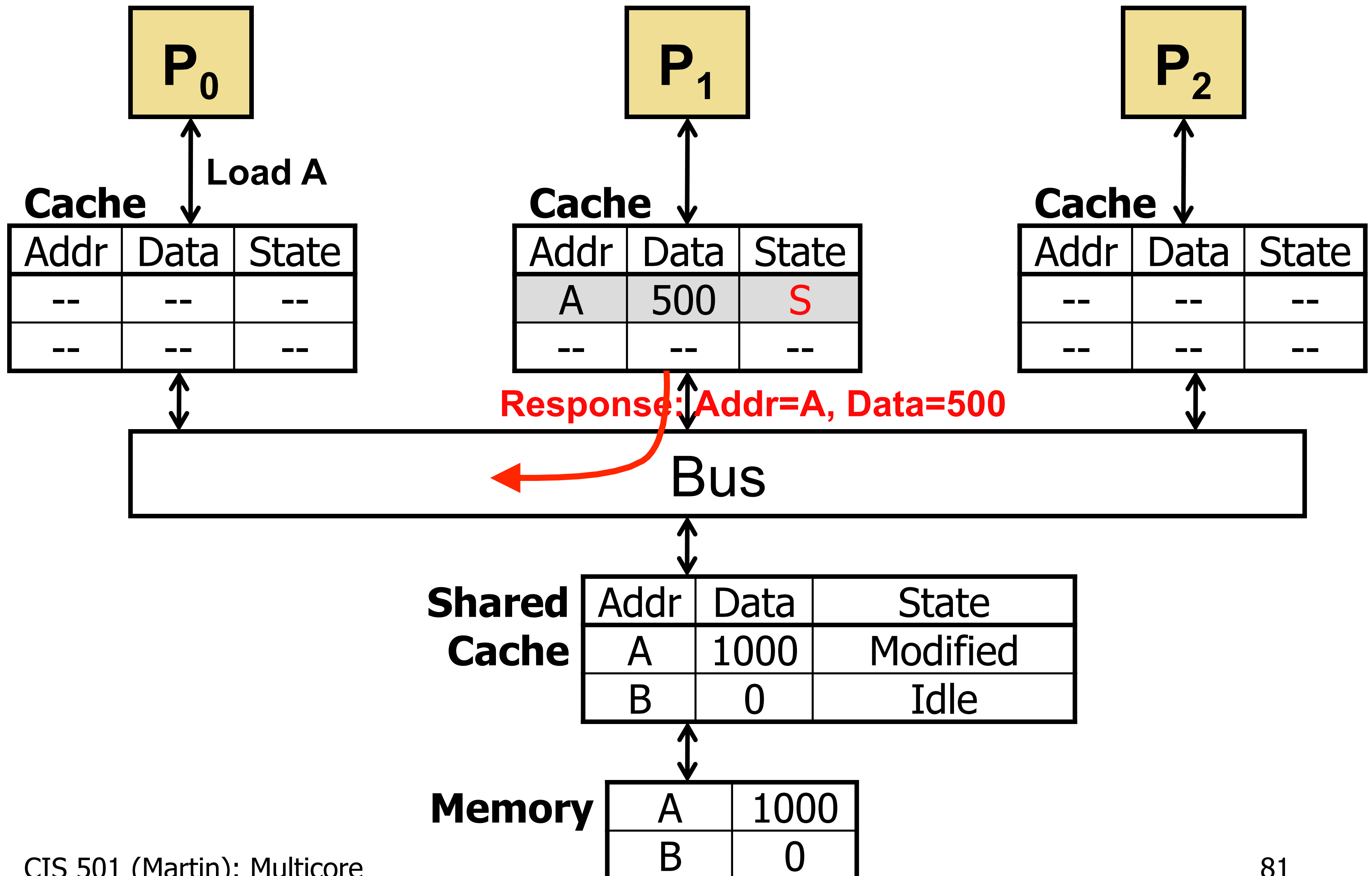
Snooping Example: Step #1



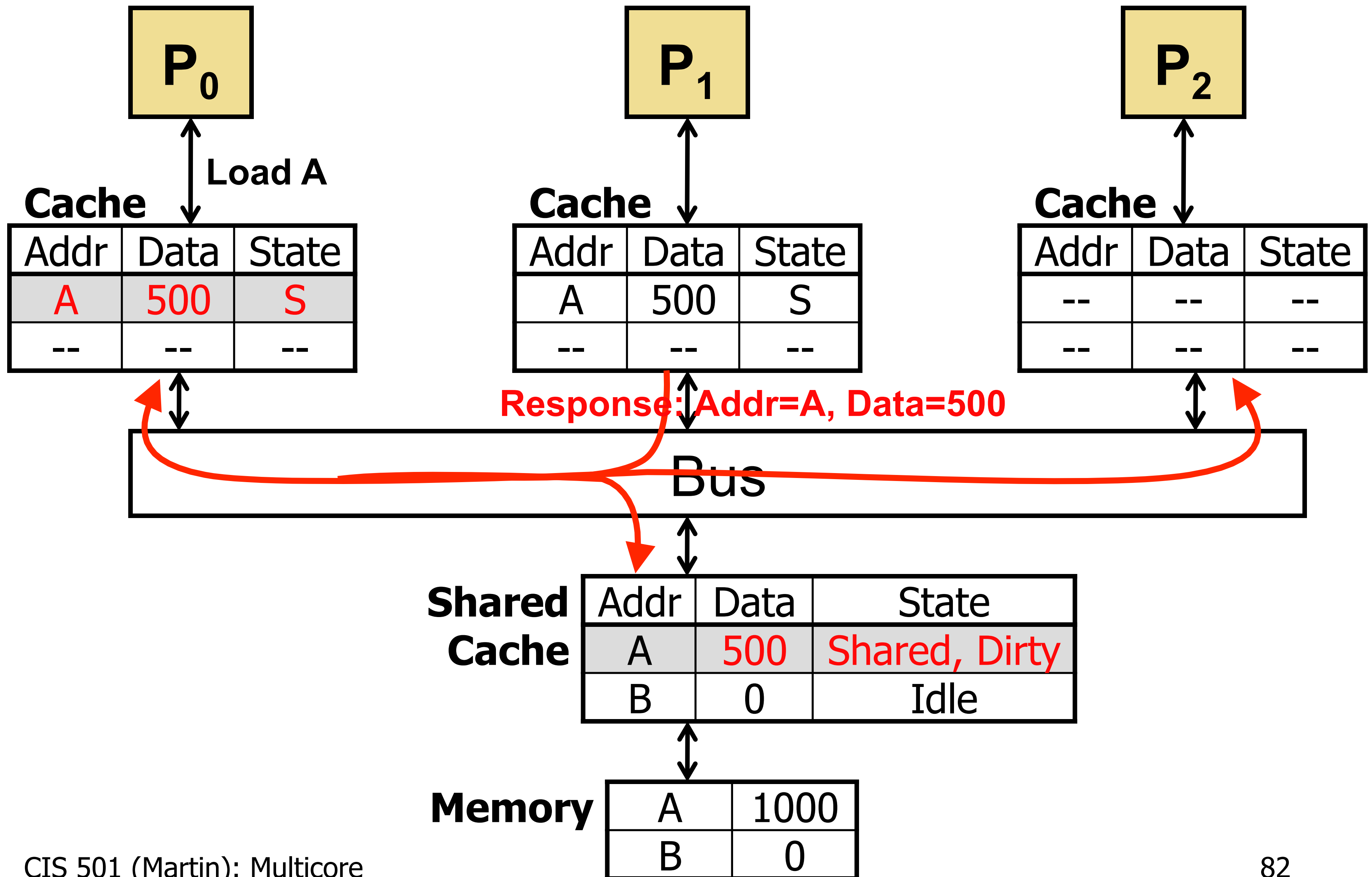
Snooping Example: Step #2



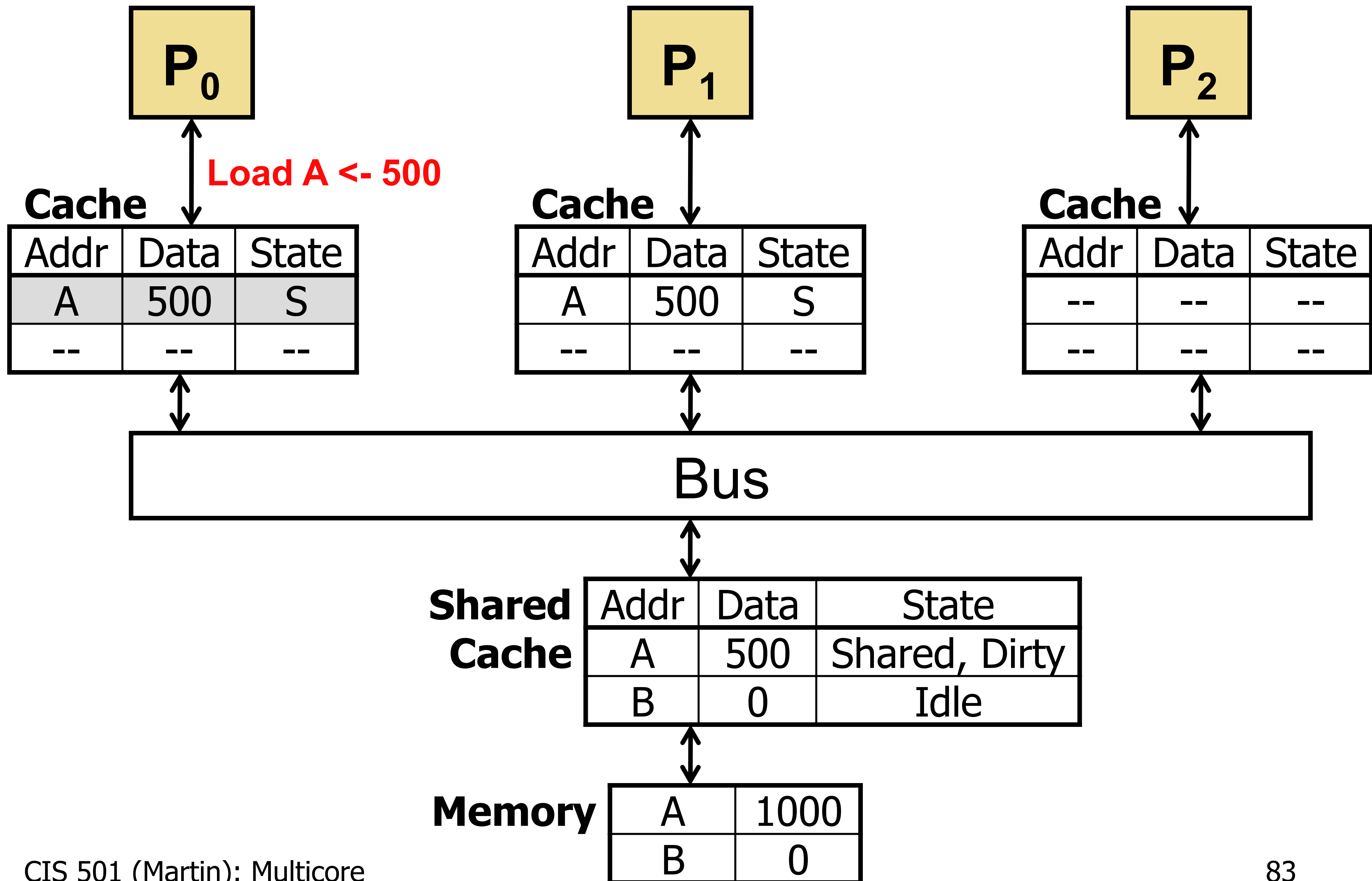
Snooping Example: Step #3



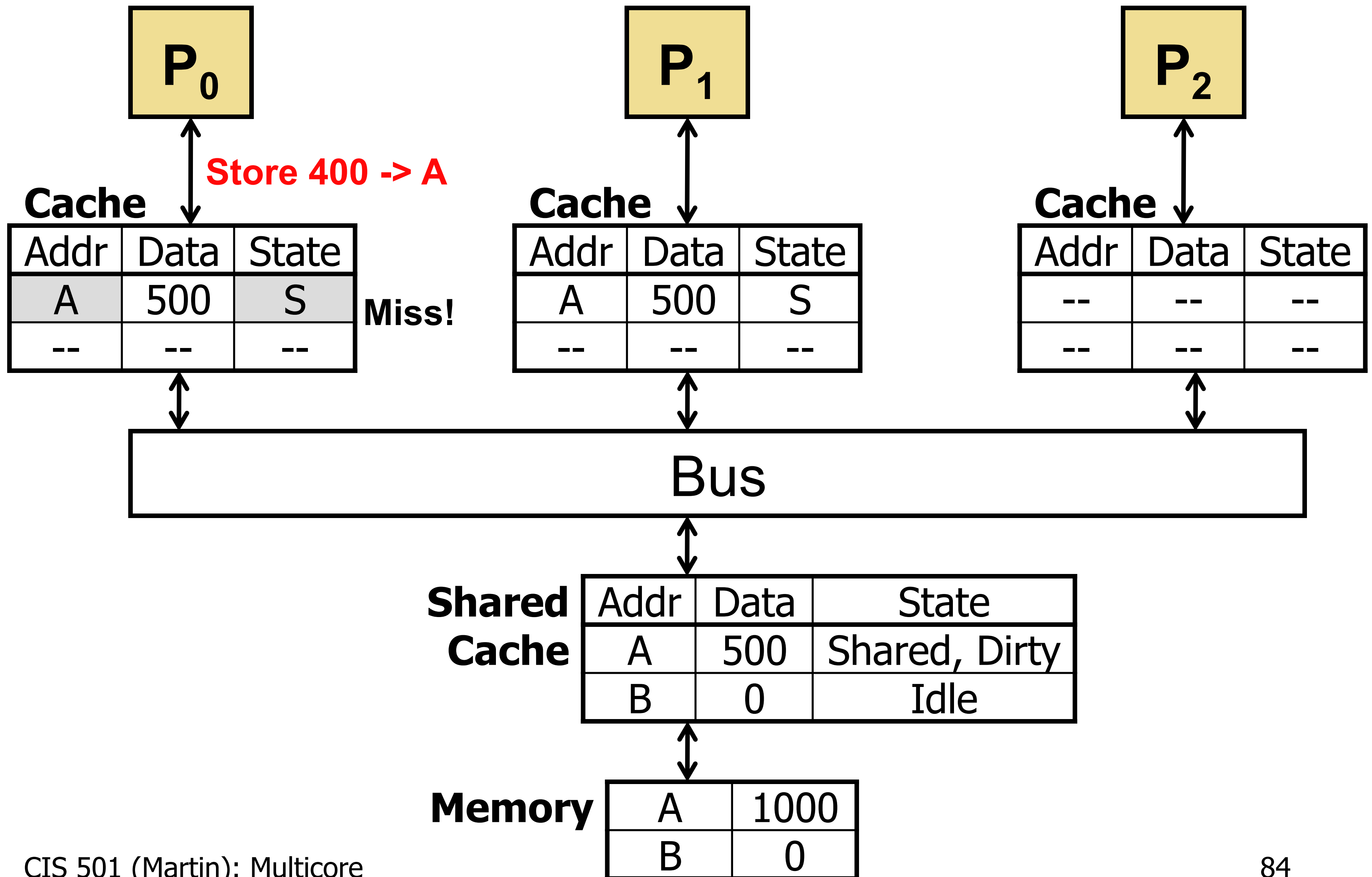
Snooping Example: Step #4



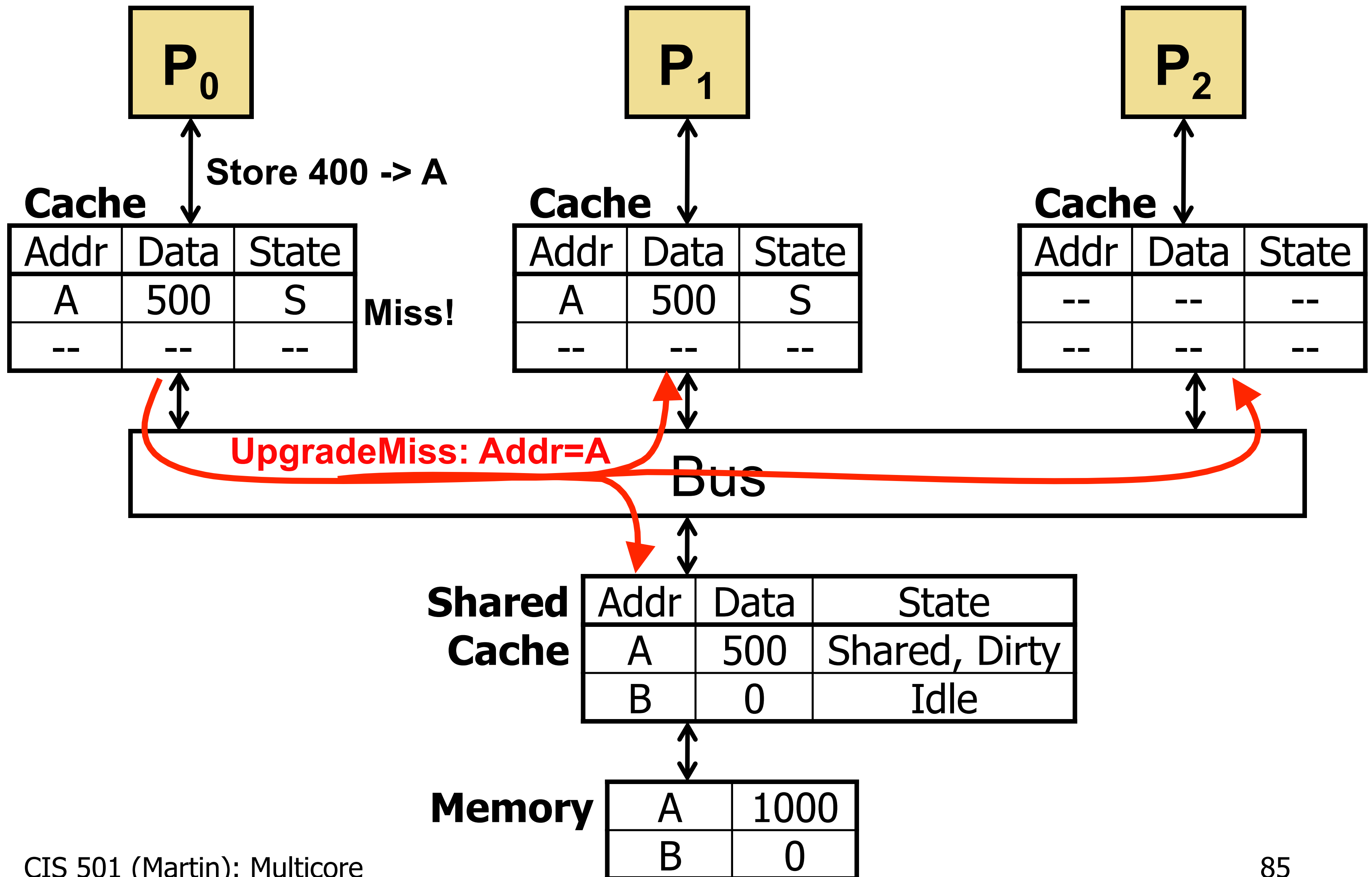
Snooping Example: Step #5



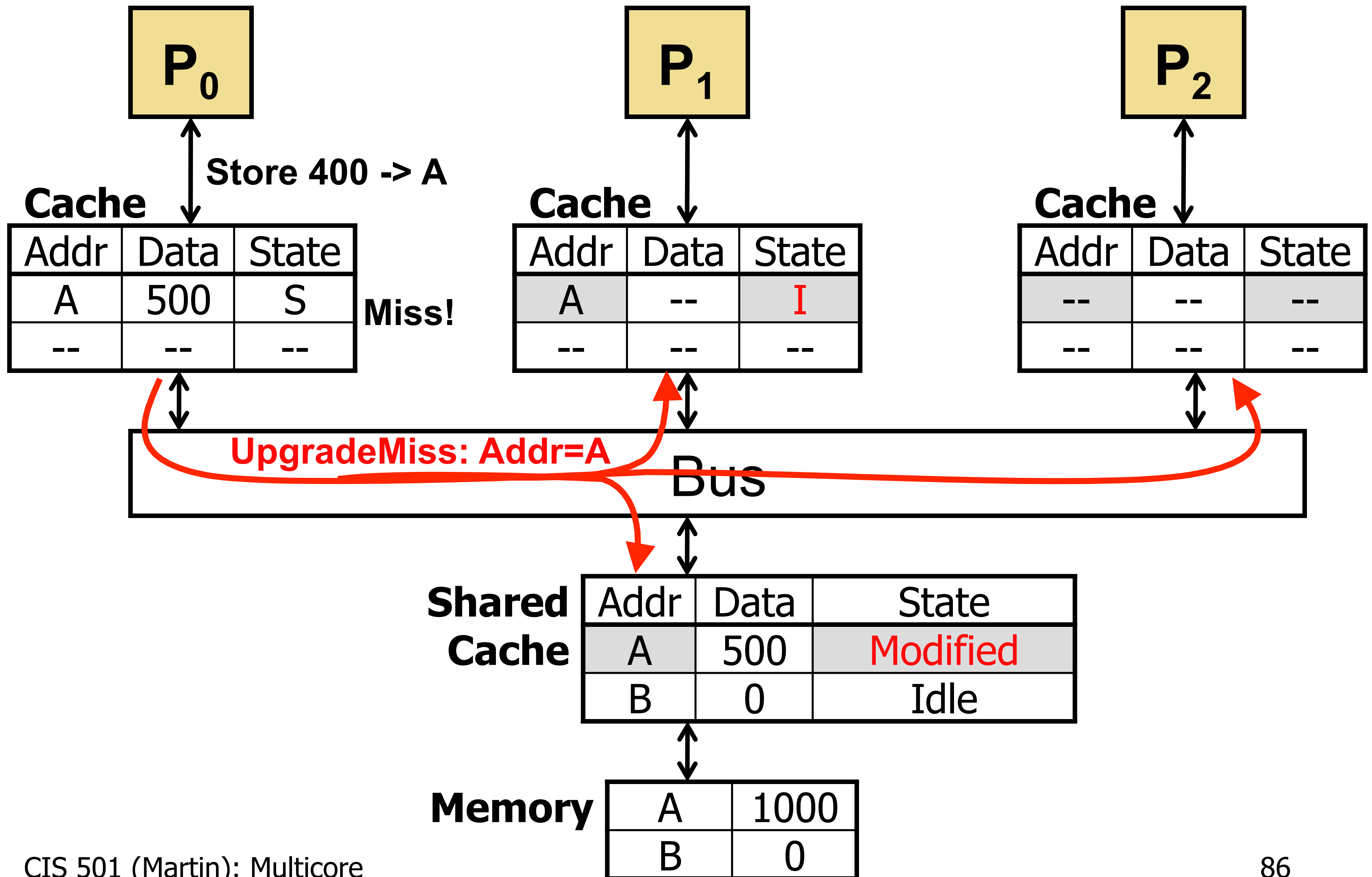
Snooping Example: Step #6



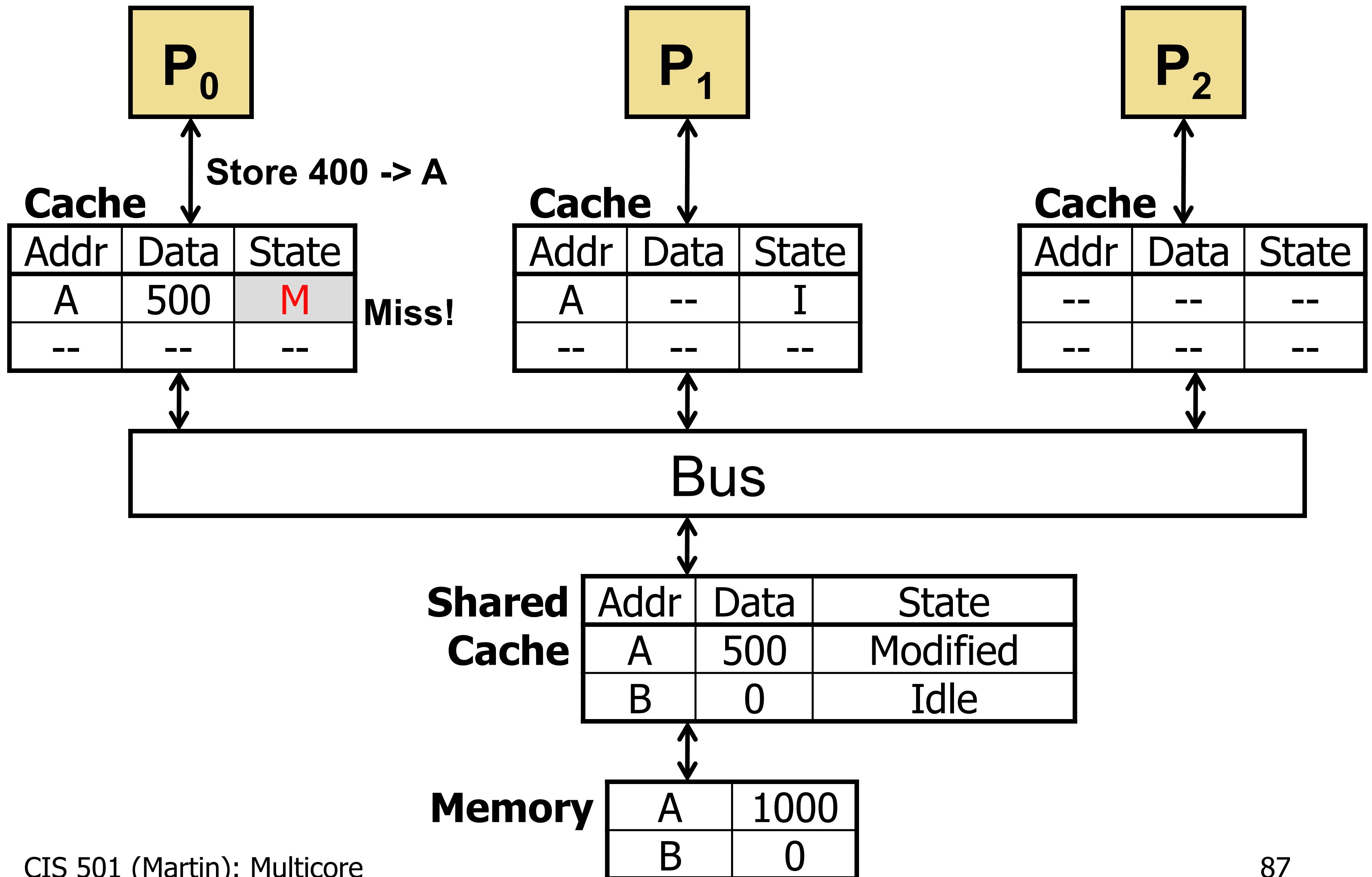
Snooping Example: Step #7



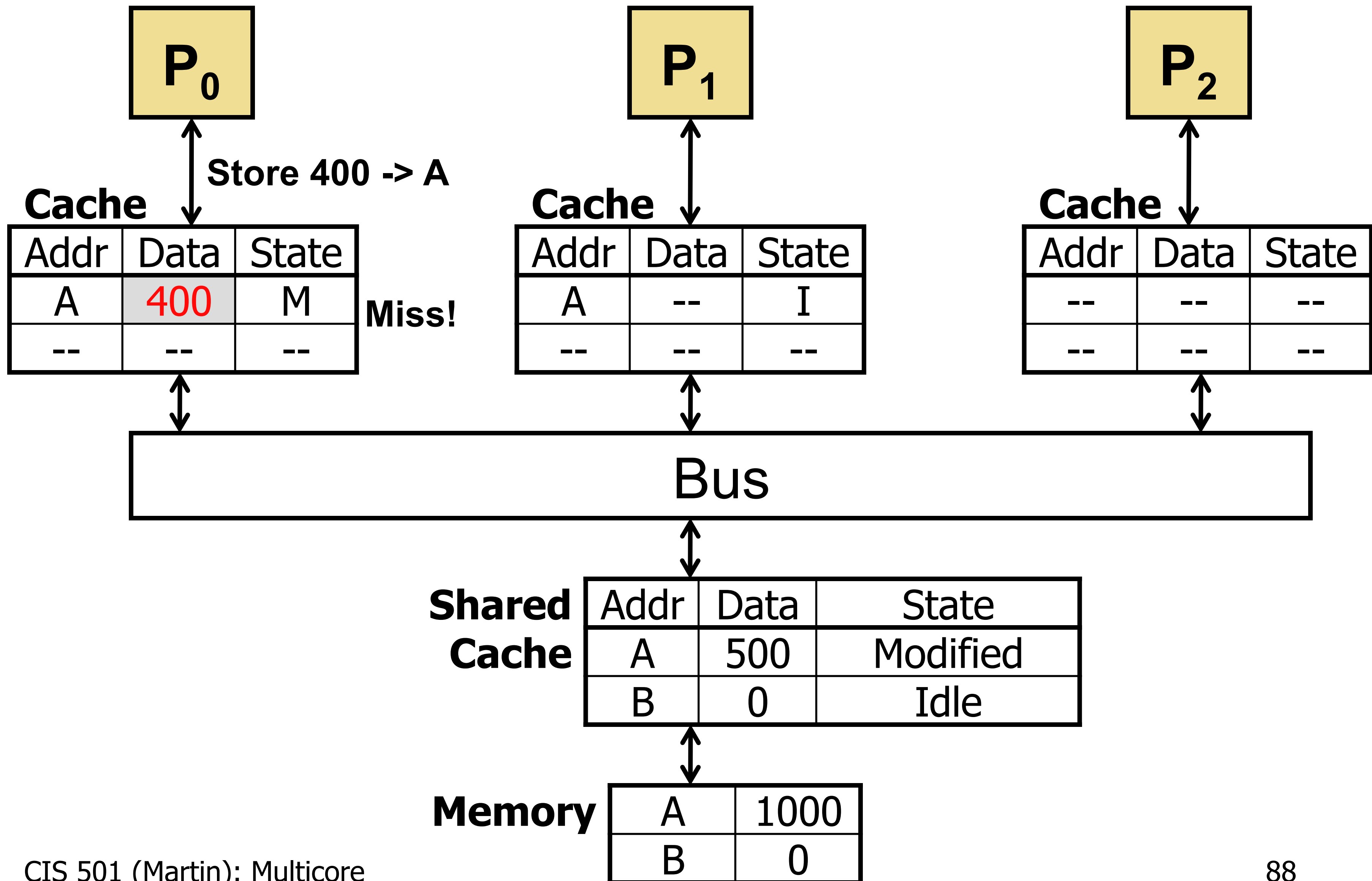
Snooping Example: Step #8



Snooping Example: Step #9



Snooping Example: Step #10



Exclusive Clean Protocol Optimization

```
Processor 0
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)
```

```
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)
```

	CPU0	CPU1	Mem
			500
	E:500		500
(No miss!)	M:400		**500**
	S:400	S:400	400
	I:	M:300	**400**

- 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

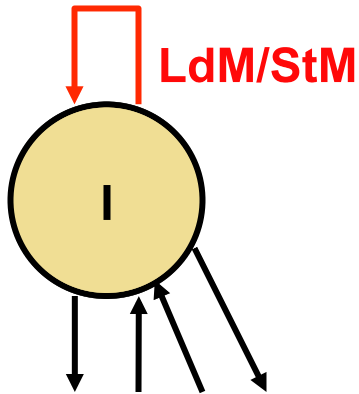
State	*This Processor* Load	*This Processor* Store	*Other Processor* Load Miss	*Other Processor* Store Miss
Invalid (I)	Miss → **S or E**	Miss → M	---	---
Shared (S)	Hit	Upg Miss → M	---	→ I
Exclusive (E)	Hit	Hit → M	Send Data → S	Send Data → I
Modified (M)	Hit	Hit	Send Data → S	Send Data → I

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

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?

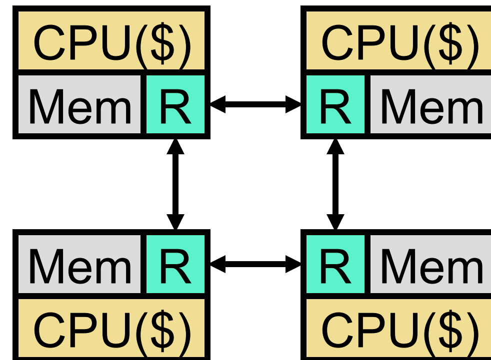
“Scalable” Cache Coherence

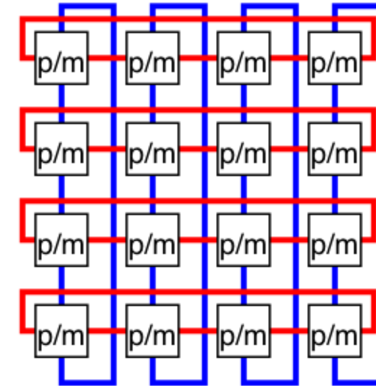


- Part I: **bus bandwidth**
  - Replace non-scalable bandwidth substrate (bus)…
  - …with scalable one (point-to-point network, e.g., mesh)

- Part II: **processor snooping bandwidth**
  - Most snoops result in no action
  - Replace non-scalable broadcast protocol…
  - …with scalable **directory protocol** (only notify processors that care)

Point-to-Point Interconnects





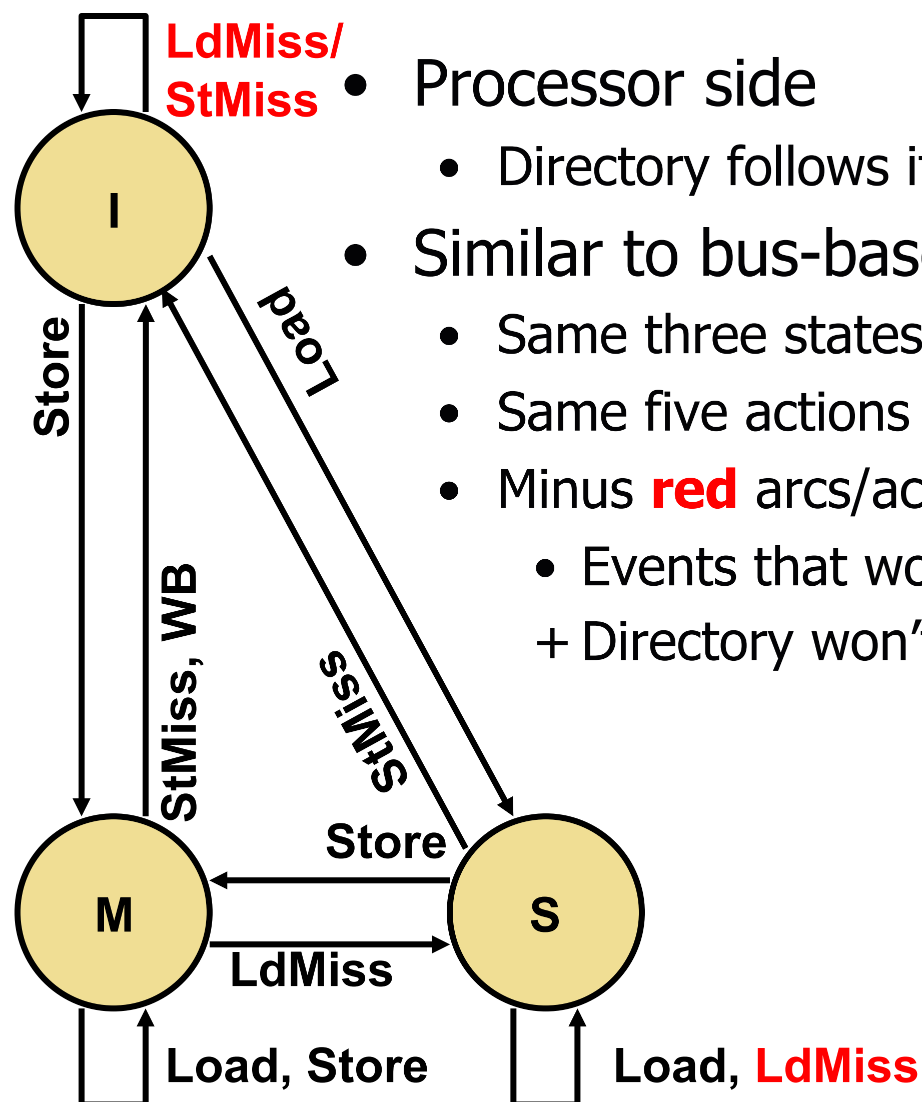
- Single "bus" does not scale to larger core counts
  - Also poor electrical properties (long wires, high capacitance, etc.)
- Alternative: on-chip interconnection network
  - Routers move packets over short point-to-point links
  - Examples: on-chip mesh or ring interconnection networks
- Used within a multicore chip
  - Each "node": a core, L1/L2 caches, and a "bank" (1/nth) of the L3 cache
  - Multiple memory controllers (which talk to off-chip DRAM)
- Can also connect arbitrarily large number of chips
  - **Massively parallel processors (MPPs)**
  - Distributed memory: non-uniform memory architecture (NUMA)

Directory Coherence Protocols

- **Directories**: non-broadcast coherence protocol
  - 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
  - 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

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)
```

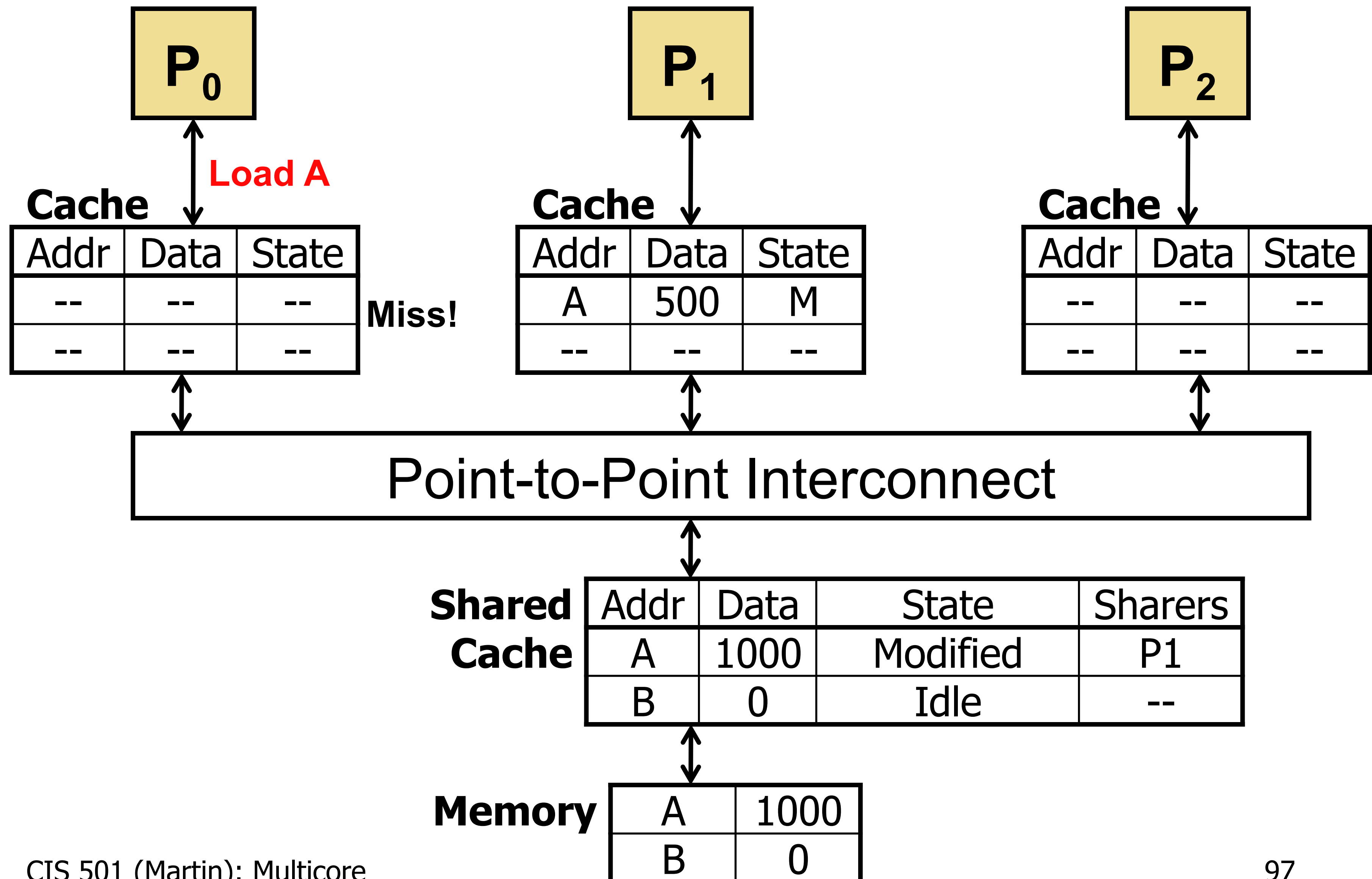
Processor 1

```
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)
```

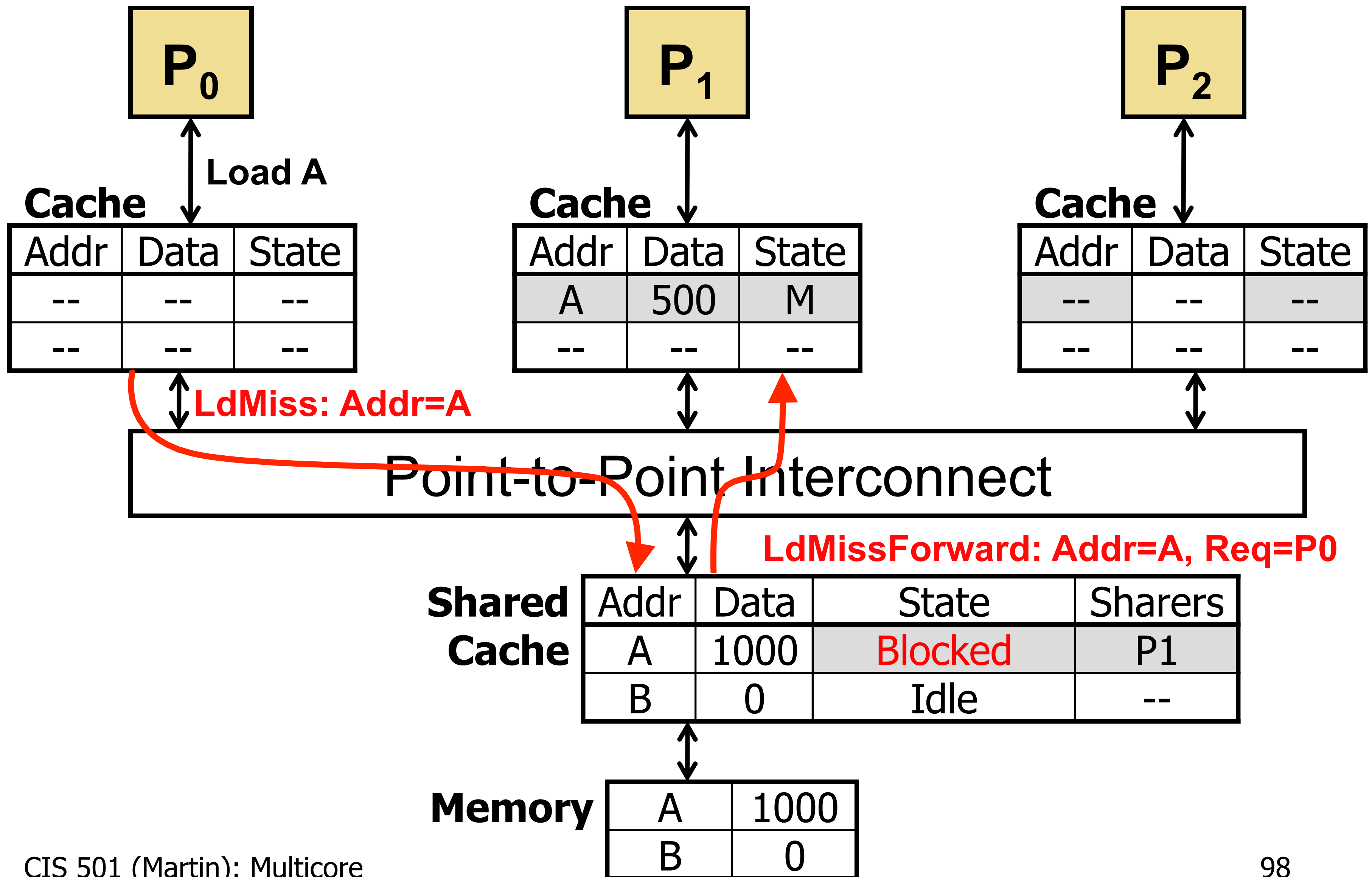
P0	P1	Directory
		–:–:500
S:500		S:0:500
M:400		M:0:500 (stale)
S:400	S:400	S:0,1:400
	M:300	M:1:400

- `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**

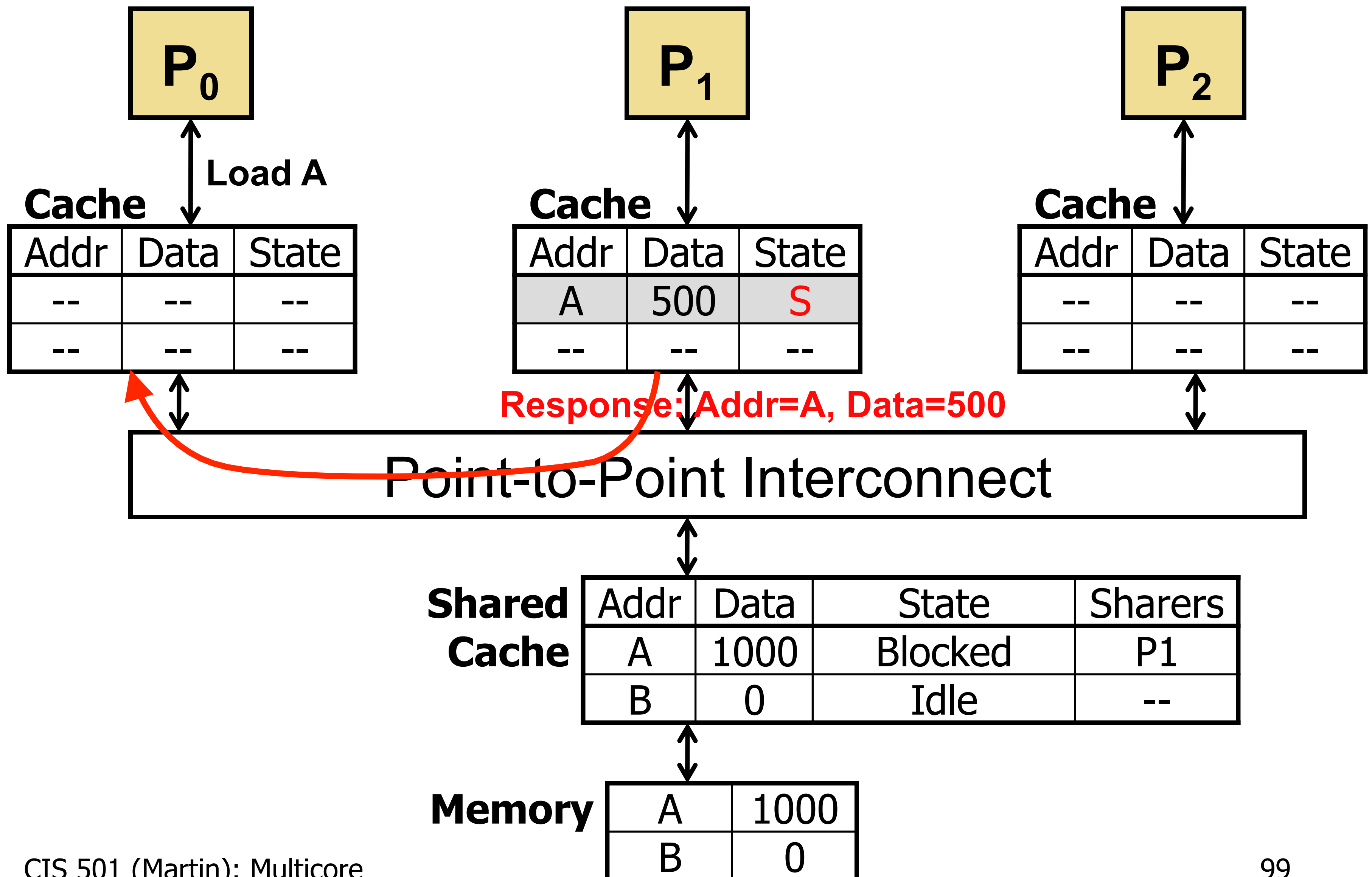
Directory Example: Step #1



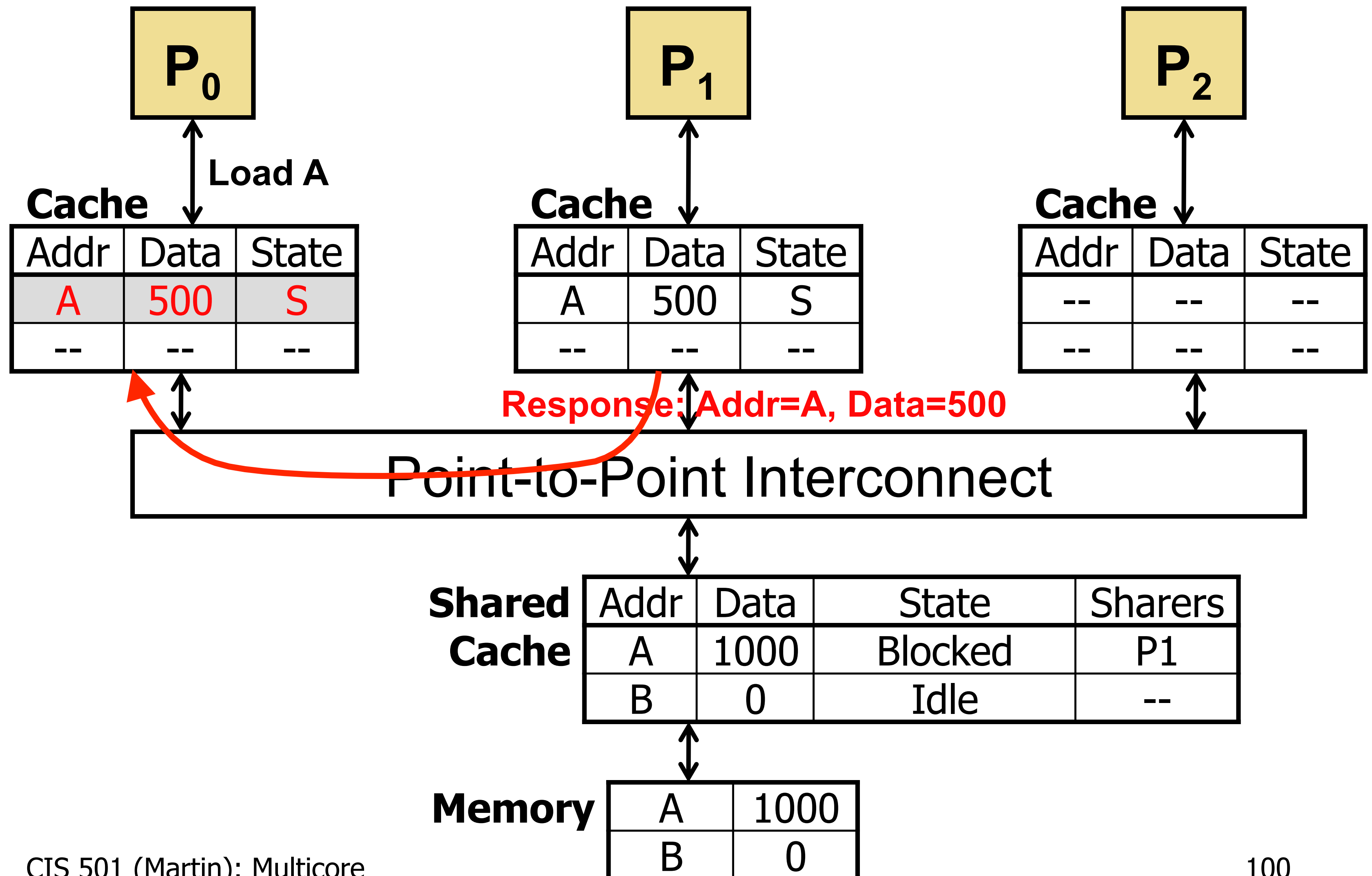
Directory Example: Step #2



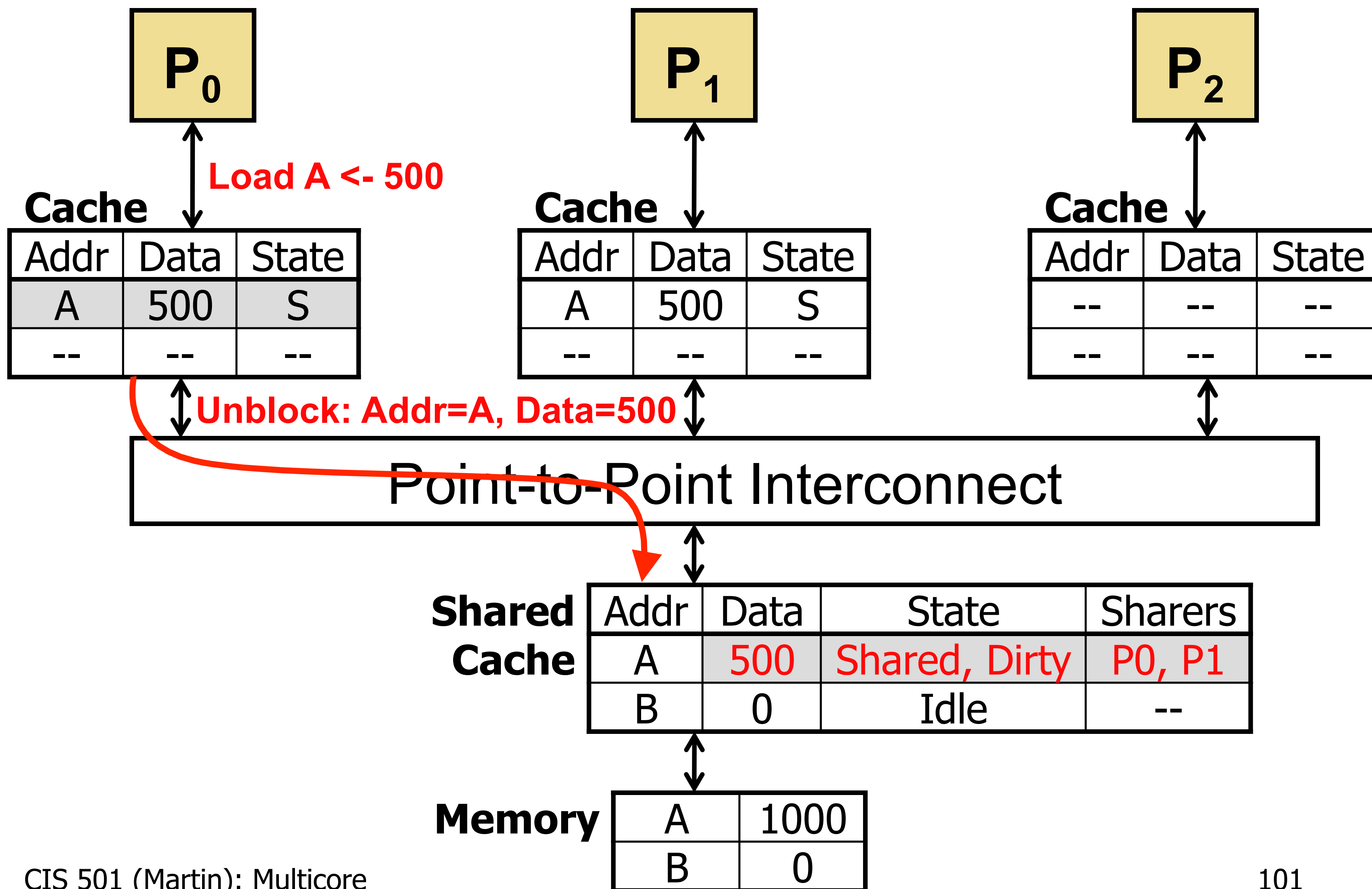
Directory Example: Step #3



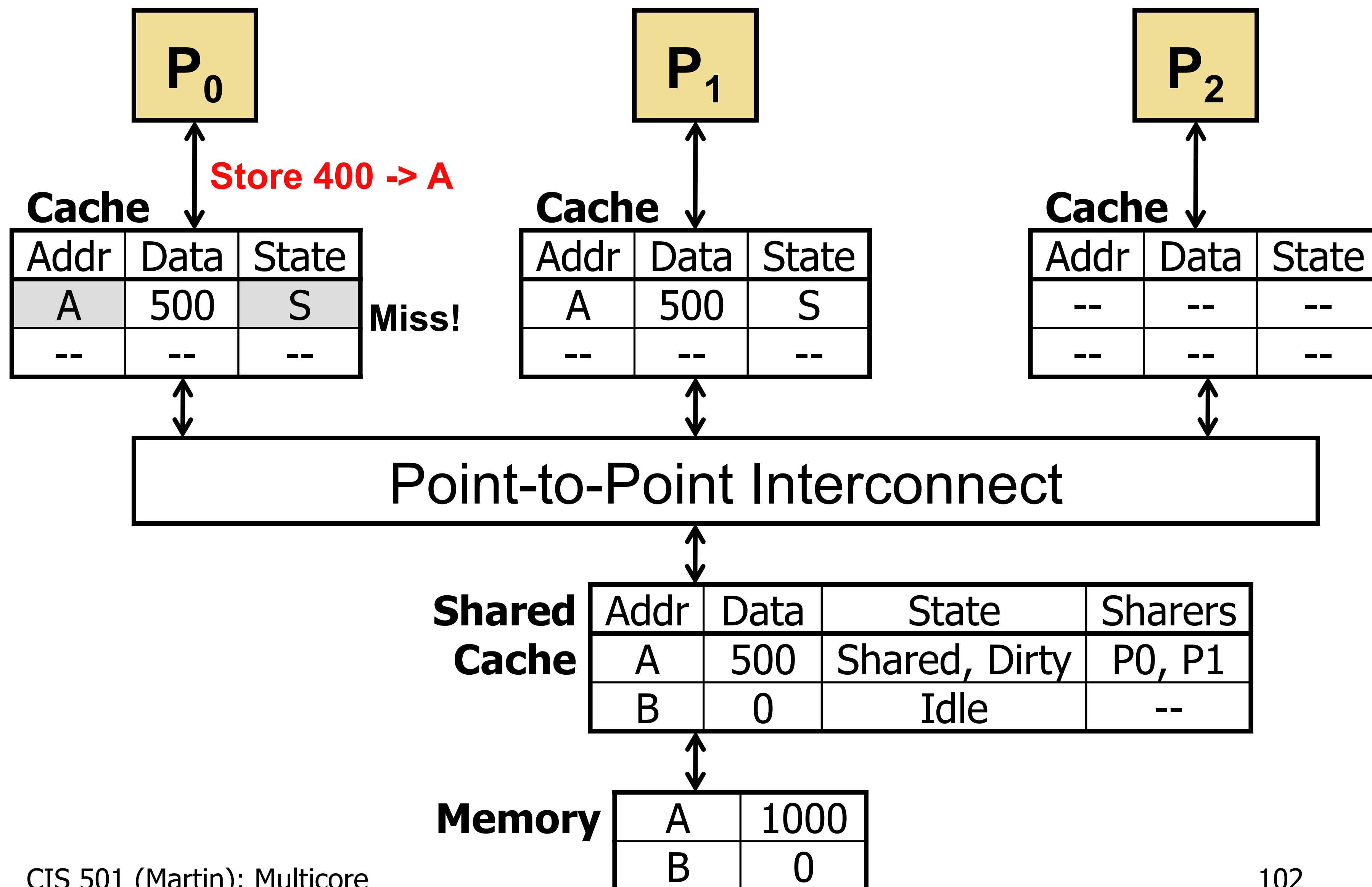
Directory Example: Step #4



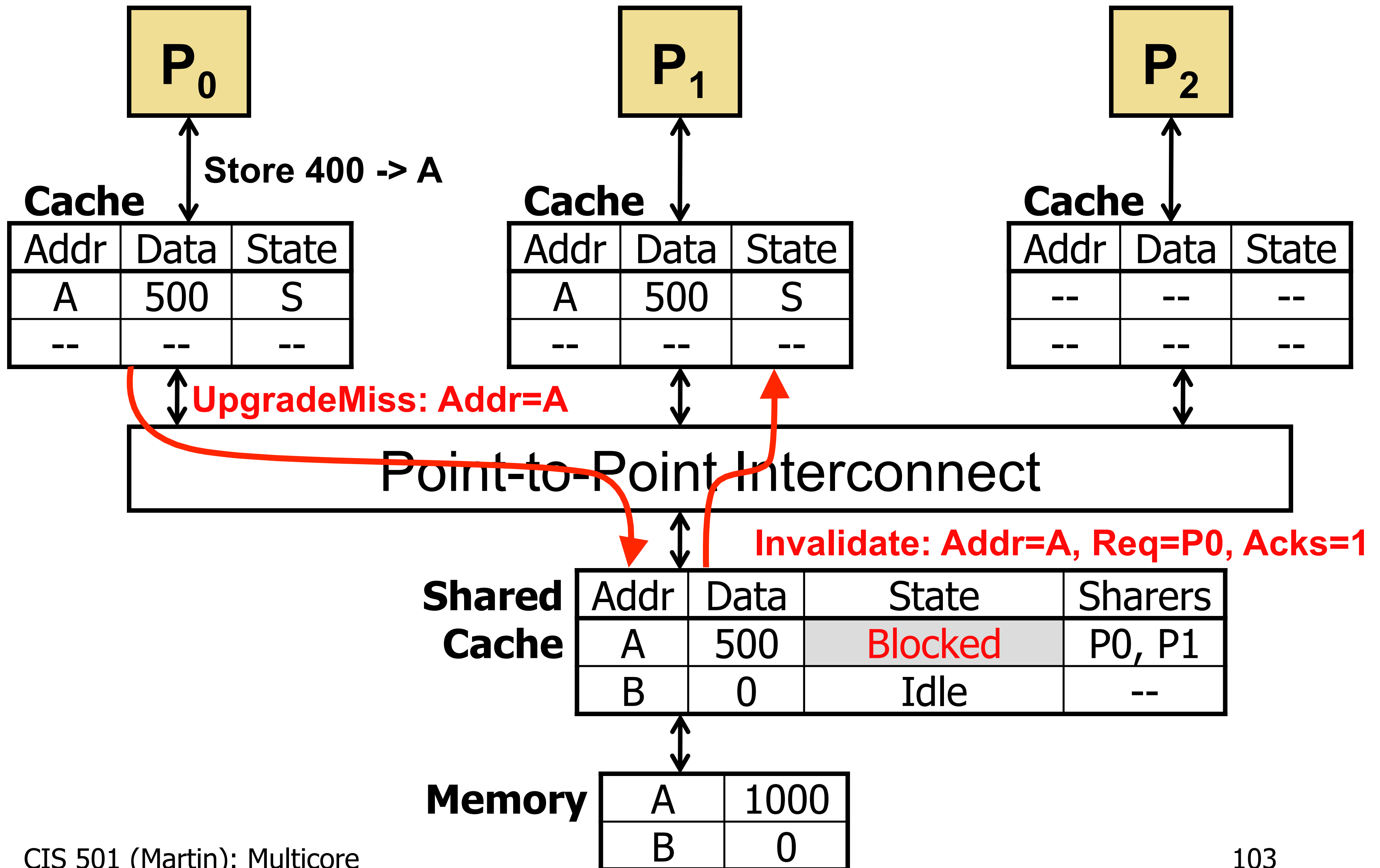
Directory Example: Step #5



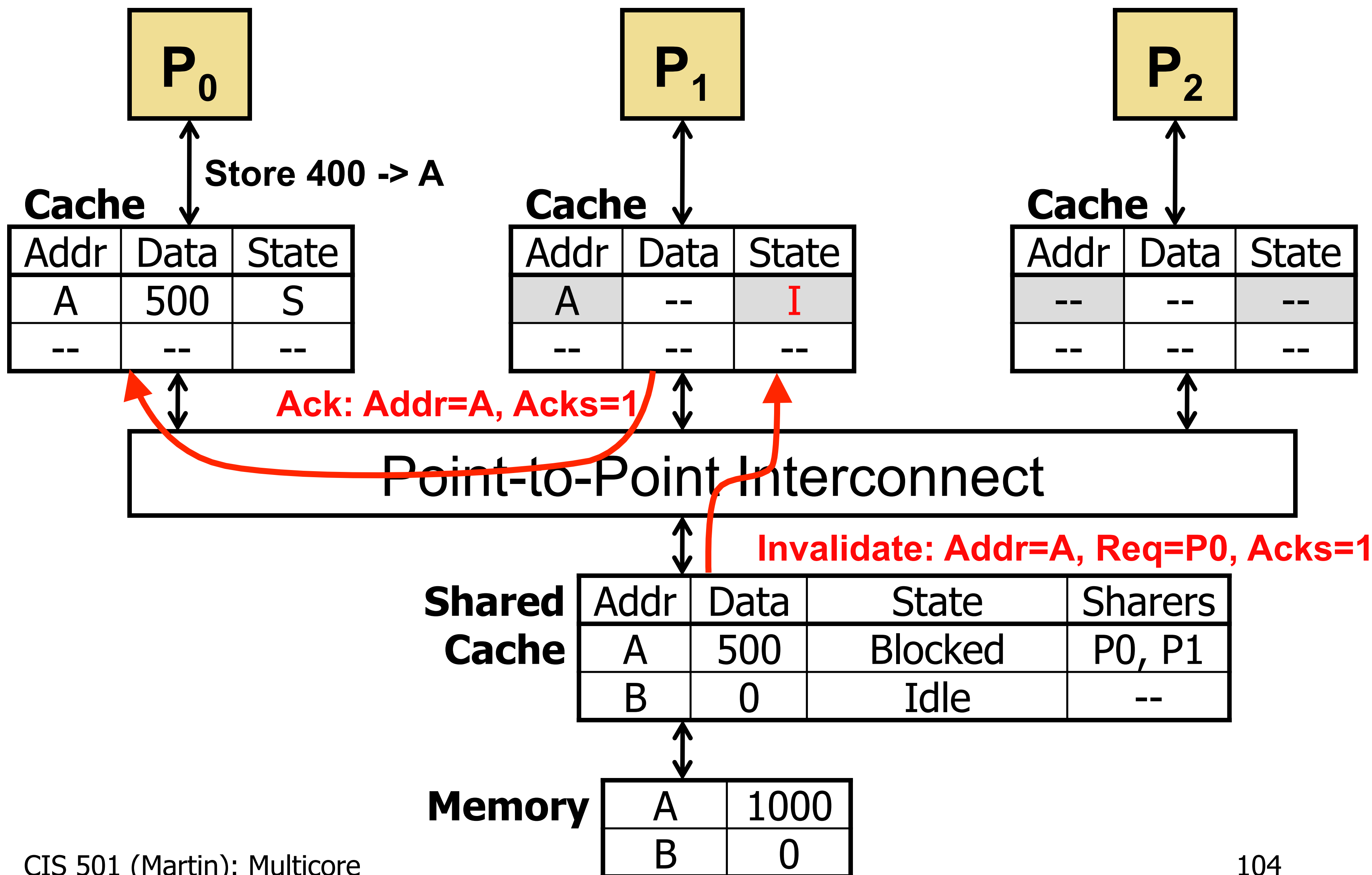
Directory Example: Step #6



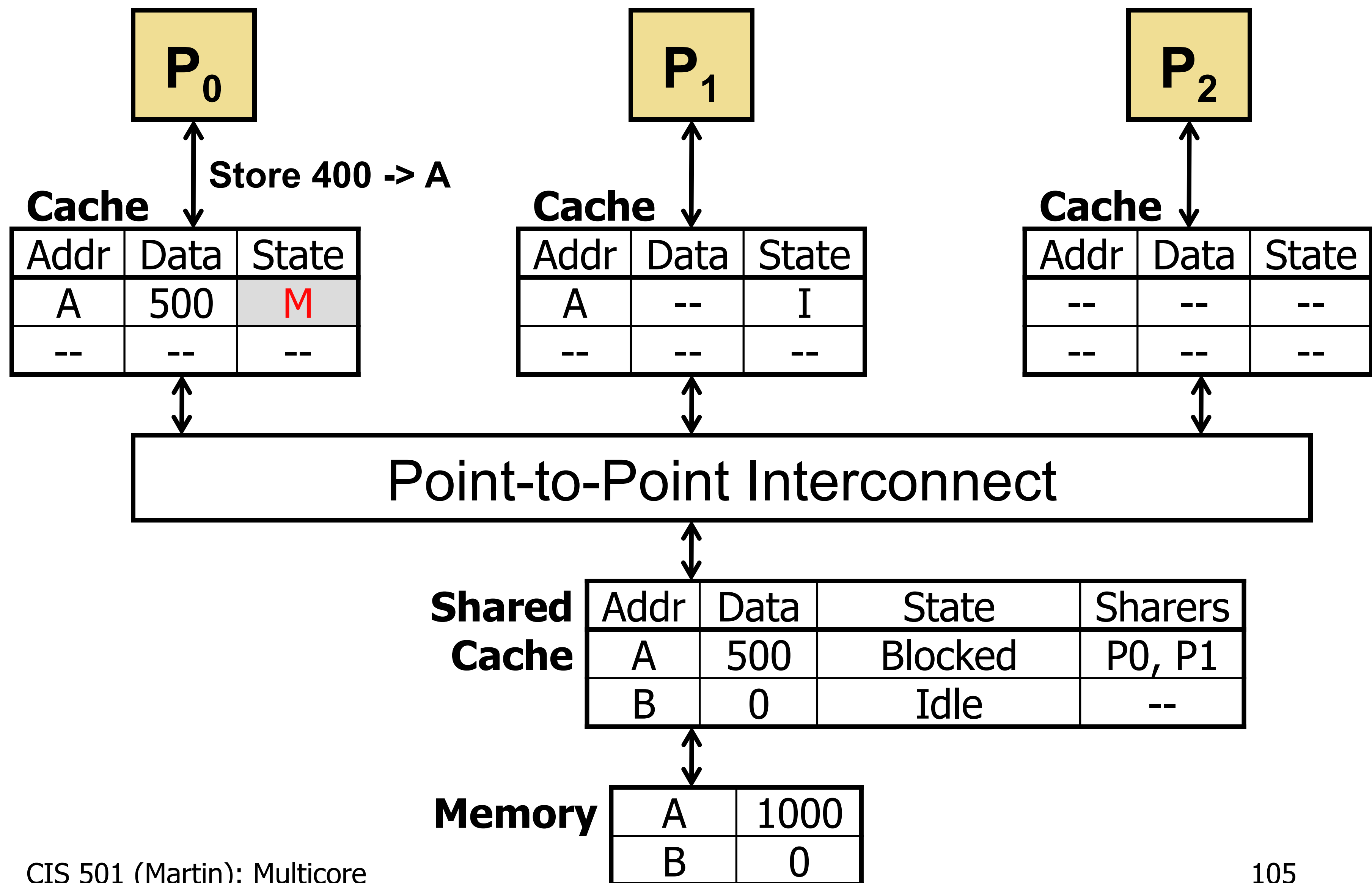
Directory Example: Step #7



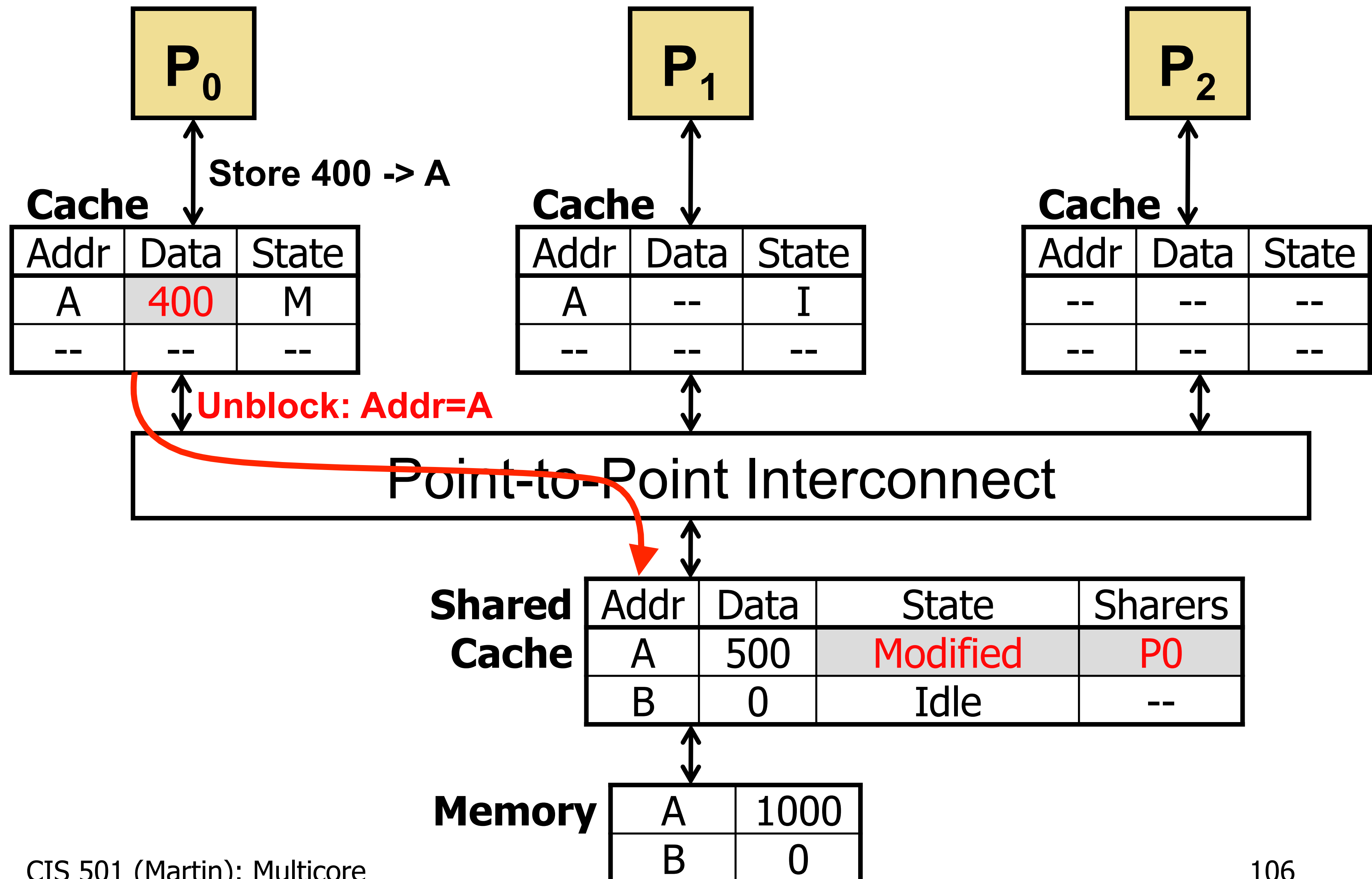
Directory Example: Step #8



Directory Example: Step #9

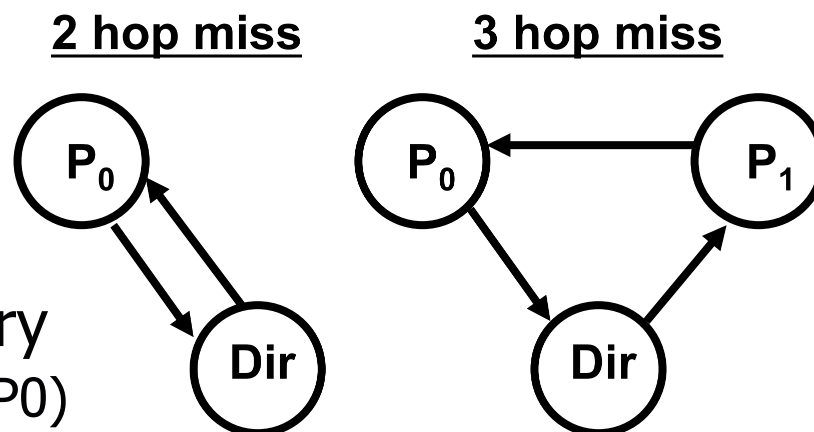


Directory Example: Step #10



Directory Flip Side: Latency

- Directory protocols
  + Lower bandwidth consumption → more scalable
  – Longer latencies
- Two read miss situations

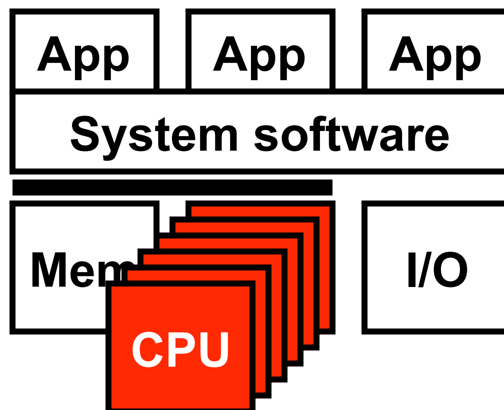


- Unshared: get data from memory
  - Snooping: 2 hops (P0→memory→P0)
  - Directory: 2 hops (P0→memory→P0)
- Shared or exclusive: get data from other processor (P1)
  - Assume cache-to-cache transfer optimization
  - Snooping: 2 hops (P0→P1→P0)
  – Directory: **3 hops** (P0→memory→P1→P0)
  - Common, with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

Coherence Recap & Alternatives

- Keeps caches "coherent"
  - Load returns the most recent stored value by any processor
  - And thus keeps caches transparent to software
- Directory-based protocol scale coherence
  - Perhaps to 1000s of cores
  - See "Why On-Chip Cache Coherence is Here to Stay"
- 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

Roadmap Checkpoint



- Thread-level parallelism (TLP)
- Shared memory model
  - Multiplexed uniprocessor
  - Hardware multihreading
  - Multiprocessing
- Synchronization
  - Lock implementation
  - Locking gotchas
- Cache coherence
  - Bus-based protocols
  - Directory protocols
- **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? (x, y)

Shared Memory Example #2

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

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

- What value pairs can be read by the two loads? (x, y)

Shared Memory Example #3

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

thread 1	thread 2
`store 1 → a` `store 1 → flag`	`while(flag == 0) { }` `load a`

- What value can be read by “load a”?

"Answer" to 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?

```
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)?

"Answer" to Example #2

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

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

- What value pairs can be read by the two loads?
  - (x=1, y=1)
  - (x=0, y=0)
  - (x=0, y=1)

- Is (x=1, y=0) allowed?

"Answer" to Example #3

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

thread 1	thread 2
`store 1 → a` `store 1 → flag`	`while(flag == 0) { }` `load a`

- What value can be read by "load a"?
  - "load a" can see the value "1"
- Can "load a" read the value zero?

What is Going On?

- Reordering of memory operations to different addresses!

- **In the compiler**
  - Compiler is generally allowed to re-order memory operations to different addresses
  - Many other compiler optimizations also cause problems

- **In the hardware**
  - To tolerate write latency
    - Processes don't wait for writes to complete
    - And why should they? No reason on a uniprocessors
  - To simplify out-of-order execution

Memory Consistency

- **Memory coherence**
  - Creates globally uniform (consistent) view…
  - Of **a single memory location** (in other words: cache blocks)
  - Not enough
    - Cache blocks A and B can be individually consistent…
    - But inconsistent with respect to each other

- **Memory consistency**
  - Creates globally uniform (consistent) view…
  - Of **all memory locations relative to each other**

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

Coherence vs. Consistency

```
                A=0  flag=0
Processor 0             Processor 1
A=1;                    while (!flag); // spin
flag=1;                 print A;
```

- **Intuition says**: P1 prints A=1
- **Coherence says**: absolutely nothing
  - P1 can see P0's write of `flag` before write of `A`!!! How?
    - **P0 has a coalescing store buffer that reorders writes**
    - **Or out-of-order load execution**
    - **Or compiler reorders instructions**
- Imagine trying to figure out why this code sometimes "works" and sometimes doesn't
- **Real systems** are allowed to act in this strange manner
  - What is allowed? defined as part of the ISA and/or language

Why? To Hide Store Miss Latency

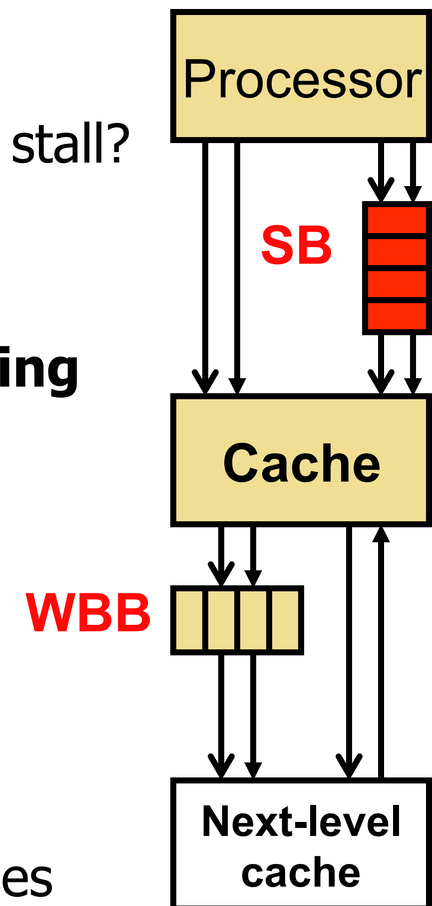
- Why? Why Allow Such Odd Behavior?
  - Reason #1: hiding store miss latency
- Recall (back from caching unit)
  - Hiding store miss latency
  - How? Store buffer
- Said it would complicate multiprocessors
  - Yes. It does.

Recall: Write Misses and Store Buffers

- Read miss?
  - Load can't go on without the data, it must stall
- Write miss?
  - Technically, no instruction is waiting for data, why stall?
- **Store buffer**: a small buffer
  - Stores put address/value to store buffer, **keep going**
  - Store buffer writes stores to D$ in the background
  - Loads must search store buffer (in addition to D$)

  \+ Eliminates stalls on write misses (mostly)

  – Creates some problems (later)
- Store buffer vs. writeback-buffer
  - Store buffer: "in front" of D$, for hiding store misses
  - Writeback buffer: "behind" D$, for hiding writebacks



Two Kinds of Store Buffers

- FIFO (First-in, First-out) store buffers
  - All stores enter the store buffer, drain into the cache in-order
  - In an in-order processor...
    - Allows later loads to execute under store miss
  - In an out-of-order processor…
    - Instructions “commit” with older stores still in the store queue
- “Coalescing” store buffers
  - Organized like a mini-cache (tags, blocks, etc.)
    - But with per-byte valid bits
  - At commit, stores that miss the cache placed in store buffer
    - Stores that hit in the cache, written into cache
  - When the store miss returns, all stores to that address drain into the cache
    - That is, not necessarily in FIFO (first-in, first-out) order

Store Buffers & Consistency

```
                A=0   flag=0
Processor 0                Processor 1
A=1;                       while (!flag); // spin
flag=1;                    print A;
```

- Consider the following execution:
  - Processor 0's write to A, misses the cache. Put in store buffer.
  - Processor 0 keeps going
  - Processor 0 write "1" to flag hits, writes to the cache
  - Processor 1 reads flag, misses cache, gets the value "1" from P0
  - Processor 1 exits loop
  - Processor 1 prints "0" for A (sees "old" value)
- Ramification: store buffers can cause "strange" behavior
  - How strange depends on lots of things
- Out-of-order execution also can cause problems...

Simplifying Out-of-Order Execution

- Why? Why Allow Such Odd Behavior?
  - Reason #2: simplifying out-of-order execution
- One key benefit of out-of-order execution:
  - Out-of-order execution of loads to (same or different) addresses

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

  - Uh, oh.

Simplifying Out-of-Order Execution

- Two options:
  - Option #1: **allow** this sort of "odd" reordering
  - Option #2: hardware **detects & prevents** such reorderings
- How to prevent?
  - Scan the Load Queue (LQ) on stores from **other** threads
  - Flush and rollback on conflict
- How to detect these stores from other threads?
  - Leverage cache coherence!
  - As long as a block remains in a private per-core cache…
    - Another core can't write to it!
  - Thus, anytime a block leaves the cache (invalidation or eviction)…
    - Scan the load queue. If any loads to the address have executed but not committed, squash the pipeline and restart

3 Classes of Memory Consistency Models

- **Sequential consistency (SC)** (MIPS, PA-RISC)
  - **Formal definition of memory view 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
  - **Indistinguishable from multi-programmed uni-processor**
- **Processor consistency (PC)** (**x86**, SPARC)
  - Allows a 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.

Answer to 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?

`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)

Answer to Example #2

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

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

- What value pairs can be read by the two loads?
  - (x=1, y=1)
  - (x=0, y=0)
  - (x=0, y=1)
- Is (x=1, y=0) allowed?
  - Yes! (for ARM, PowerPC, Itanium, and Alpha)
  - No! (for Intel/AMD x86, Sun SPARC, IBM 370)
    - Assuming the compiler didn't reorder anything…

Answer to Example #3

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

thread 1	thread 2
`store 1 → a` `store 1 → flag`	`while(flag == 0) { }` `load a`

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

Restoring Order (Hardware)

- Sometimes we need ordering (mostly we don't)
  - Prime example: ordering between "lock" and data
- How? insert **Fences (memory barriers)**
  - Special instructions, part of ISA
- Example
  - Ensure that loads/stores don't cross lock acquire/release operation

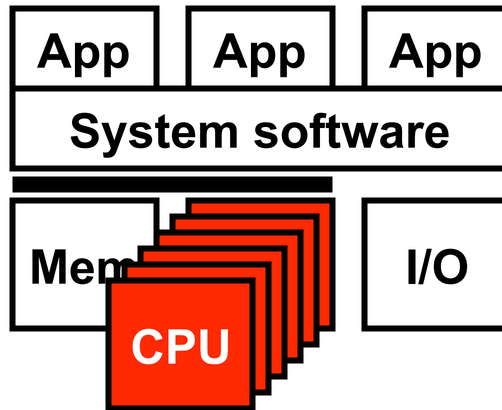
```
acquire
fence
critical section
fence
release
```

- How do fences work?
  - They stall exeuction until write buffers are empty
  - Makes lock acquisition and release slow(er)
- **Use synchronization library, don't write your own**

Restoring Order (Software)

- These slides have focused mostly on hardware reordering
  - But the compiler also reorders instructions (reason #3)
- How do we tell the compiler to not reorder things?
  - Depends on the language…
- In Java:
  - The built-in "synchronized" constructs informs the compiler to limit its optimization scope (prevent reorderings across synchronization)
  - Or, programmer uses "volatile" keyword to explicitly mark variables
  - Java compiler also inserts the hardware-level ordering instructions
- In C/C++:
  - Much more murky, as language doesn't define synchronization
  - Lots of hacks: "inline assembly", volatile, atomic (newly proposed)
  - Programmer may need to explicitly insert hardware-level fences
- **Use synchronization library, don't write your own**

Summary



- Explicit parallelism
- Shared memory model
  - Multiplexed uniprocessor
  - Hardware multihreading
  - Multiprocessing
- Synchronization
  - Lock implementation
  - Locking gotchas
- Cache coherence
  - VI, MSI, MESI
  - Bus-based protocols
  - Directory protocols
- Memory consistency