

Multicore

主讲人: 邓倩妮

上海交通大学

内容来自：

Computer Organization and Design, 4th Edition, Patterson & Hennessy,

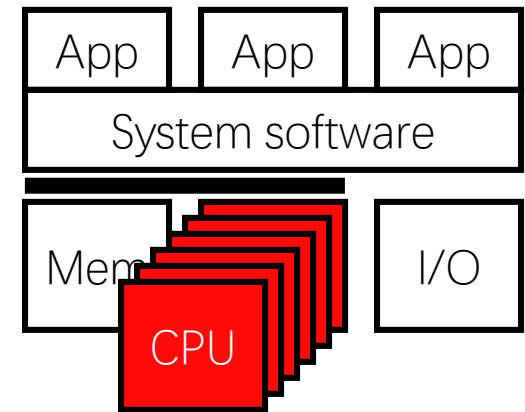


上海交通大学
SHANGHAI JIAO TONG UNIVERSITY

This Unit: Shared Memory Multiprocessors



- Thread-level parallelism (TLP)
- Shared memory model
 - Hardware multithreading
 - Multiprocessing
- Cache coherence
 - Valid/Invalid, MSI, False sharing
- Synchronization
- Memory consistency models



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

Explicit Parallelism



```
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):  
    chunk_size = SIZE / N  
    my_start = chunk_id * chunk_size  
    my_end = my_start + chunk_size  
    for (i = my_start; i < my_end; i++)  
        z[i] = a*x[i] + y[i]
```

- Assumes
 - Local variables are “private” and x, y, and z are “shared”
 - Assumes SIZE is a multiple of N (that is, $\text{SIZE} \% N == 0$)

SIZE = 400, N=4

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

Explicit Parallelism



- Consider “daxpy”:

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

- Main code then looks like:

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

Explicit (Loop-Level) Parallelism



- Another way: “OpenMP” annotations to inform the compiler

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

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

Shared Memory Programming Model



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

共享地址空间

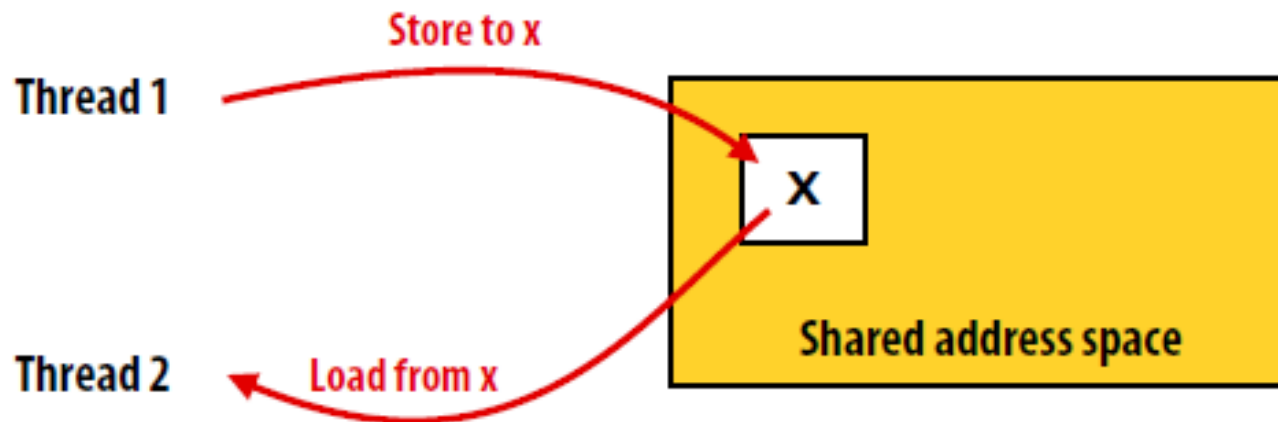


Thread 1:

```
int x = 0;  
spawn_thread(foo, &x);  
x = 1;
```

Thread 2:

```
void foo(int* x) {  
    while (x == 0) {}  
    print x;  
}
```



(Communication operations shown in red)

- Shared variables are like a big bulletin board
 - Any thread can read or write to shared variables

Shared Memory Implementations



- **Multiplexed uniprocessor**

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

- **Multiprocessors**

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

- **Hardware multithreading**

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

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



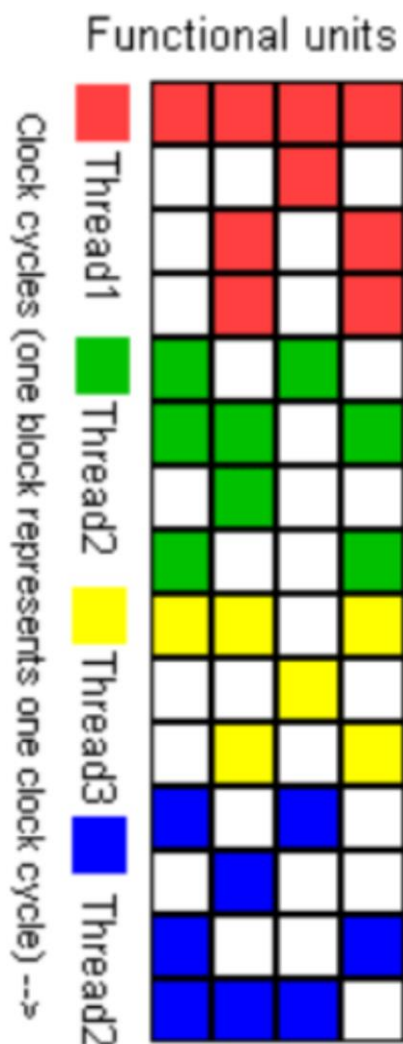
Multithread Hardware

Hardware Multithreading

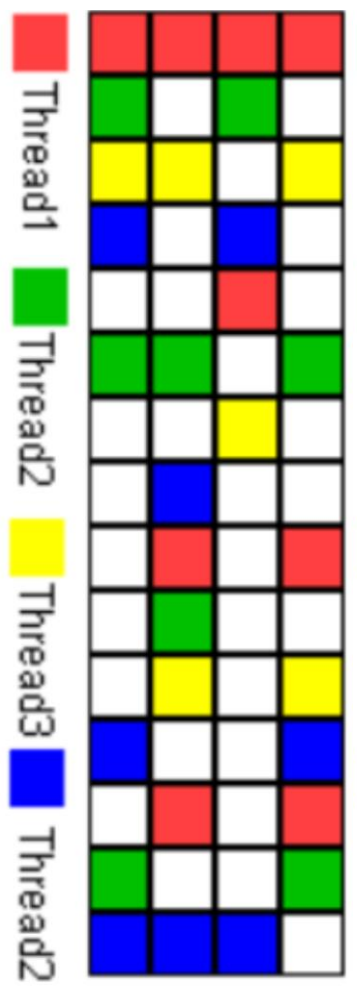


- **Not** the same as software multithreading!
- **Can multiple threads execute concurrently on the same processor?**
- Hardware Multithreading (MT)
 - Multiple hardware threads dynamically share a single pipeline
 - Hardware interleaves instructions

Approaches to Multithreading Within a Processor



Coarse-grained Multithreading $31/60 = 51.67\%$



Fine-Grained Multithreading $31/60 = 51.67\%$



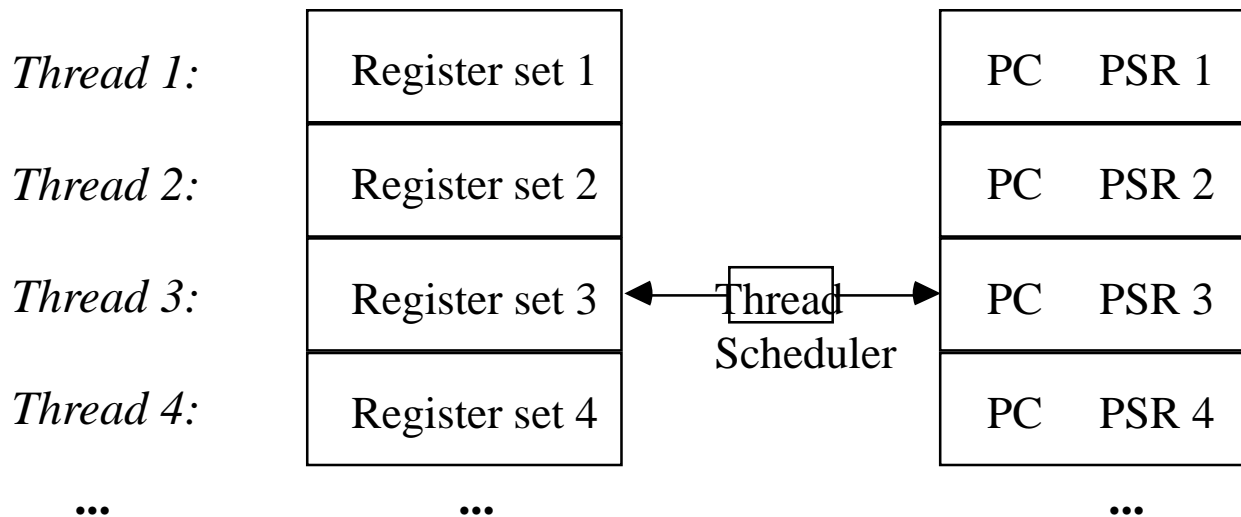
Simultaneous Multithreading $49/60 = 81.67\%$

What Resources are Shared?

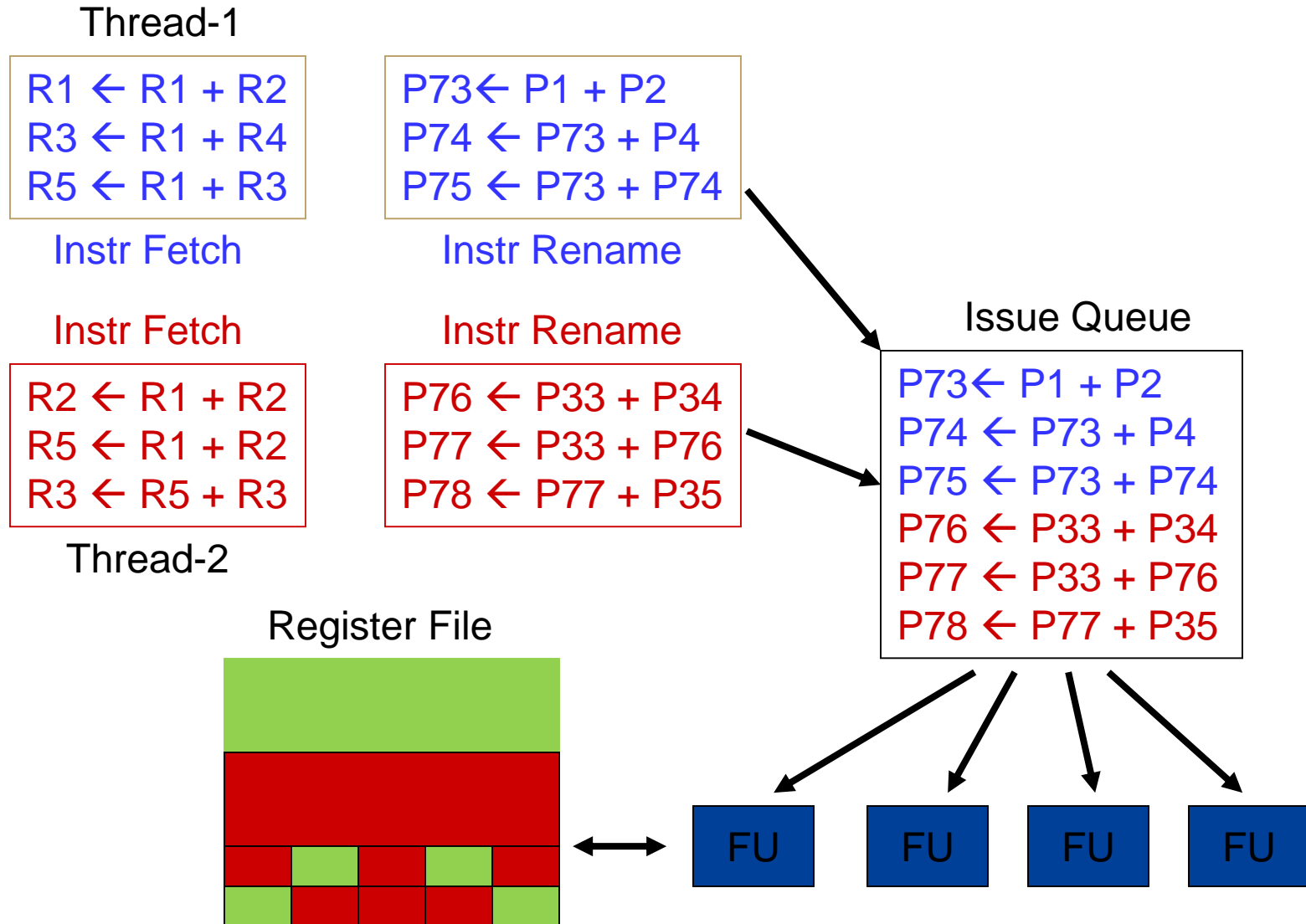


- Multiple threads are simultaneously active (in other words, a new thread can start without a context switch)

Replicate only per-thread structures: program counter & registers

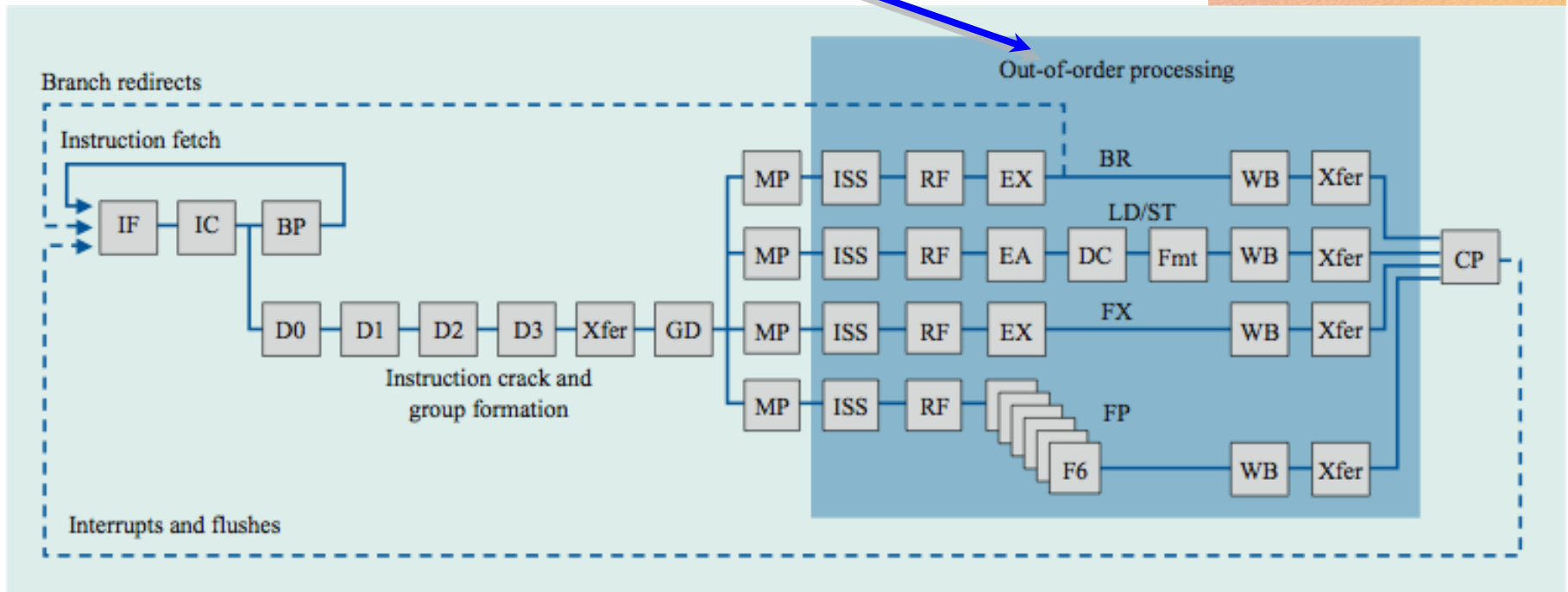


Resource Sharing in Simultaneous Multithreading



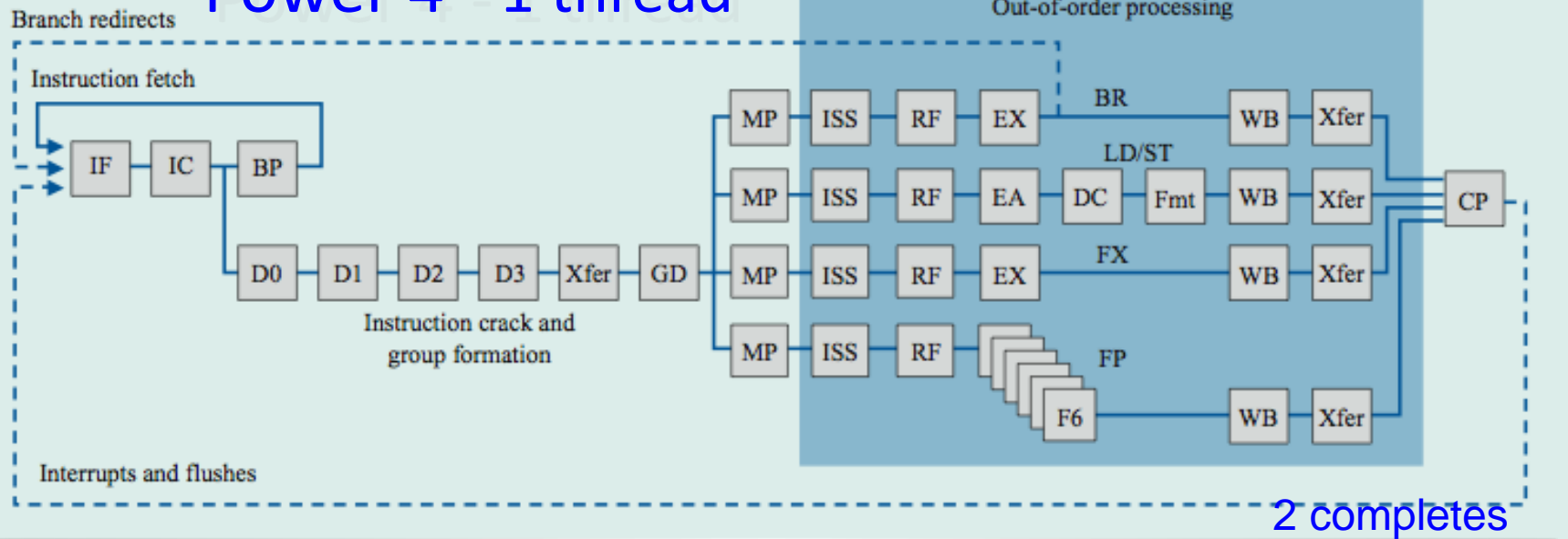
Power 4

Single-threaded predecessor to Power 5. Eight execution units in an out-of-order engine, each unit may issue one instruction each cycle.



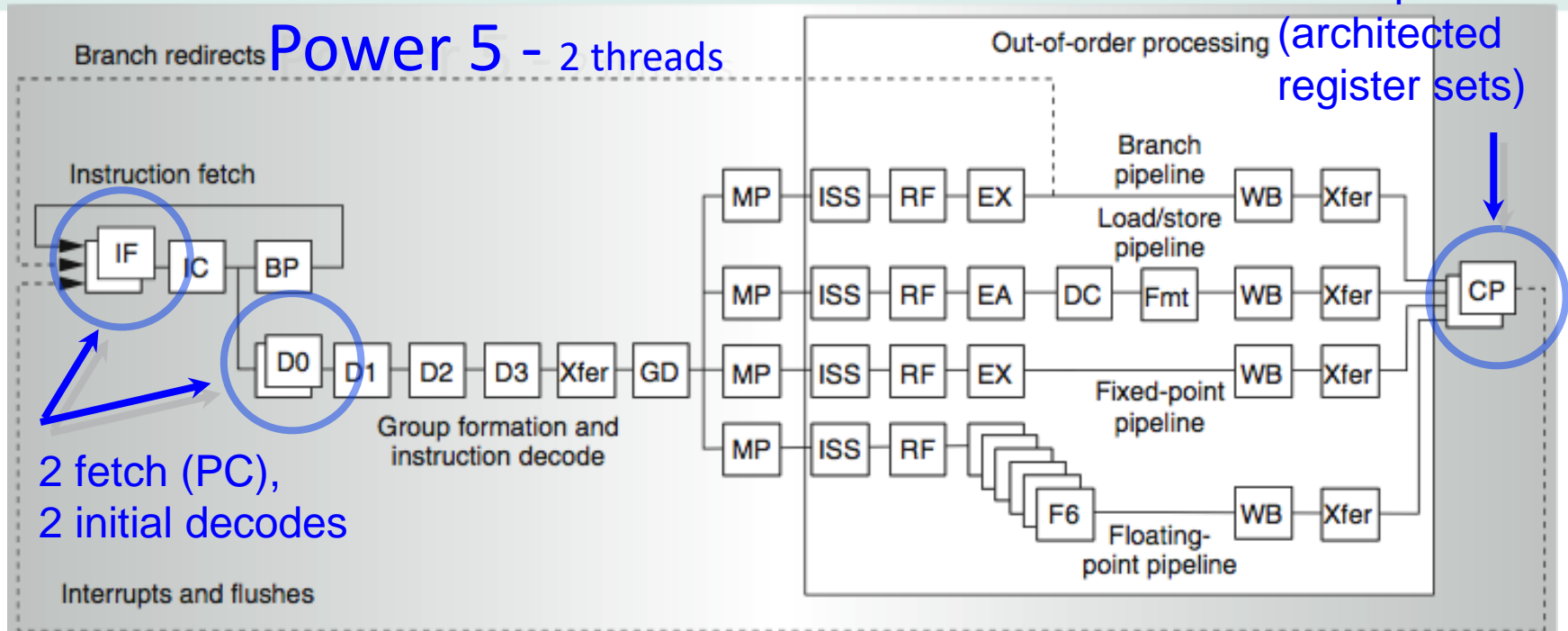
Instruction pipeline (IF: instruction fetch, IC: instruction cache, BP: branch predict, D0: decode stage 0, Xfer: transfer, GD: group dispatch, MP: mapping, ISS: instruction issue, RF: register file read, EX: execute, EA: compute address, DC: data caches, F6: six-cycle floating-point execution pipe, Fmt: data format, WB: write back, and CP: group commit)

Power 4 - 1 thread



2 completes
(architected
register sets)

Power 5 - 2 threads



HW mechanisms to support multithreading



- Large set of virtual registers that can be used to hold the register sets of independent threads
- Register renaming provides unique register identifiers, so instructions from multiple threads can be mixed in datapath without confusing sources and destinations across threads
- Out-of-order completion allows the threads to execute out of order, and get better utilization of the HW
- Independent commitment can be supported by logically keeping a separate reorder buffer for each thread

Changes in Power 5 to support SMT



- (**Increased/decreased?**) associativity of L1 instruction cache and the instruction address translation buffers
- Added per thread load and store queues
- (**Increased/decreased?**) size of the L2 (1.92 vs. 1.44 MB) and L3 caches
- Added separate instruction prefetch and buffering per thread
- (**Increased/decreased?**) the number of virtual registers from 152 to 240
- Increased the size of several issue queues
- The Power5 core is about 24% (**larger/smaller?**) than the Power4 core because of the addition of SMT support

Hardware Multithreading



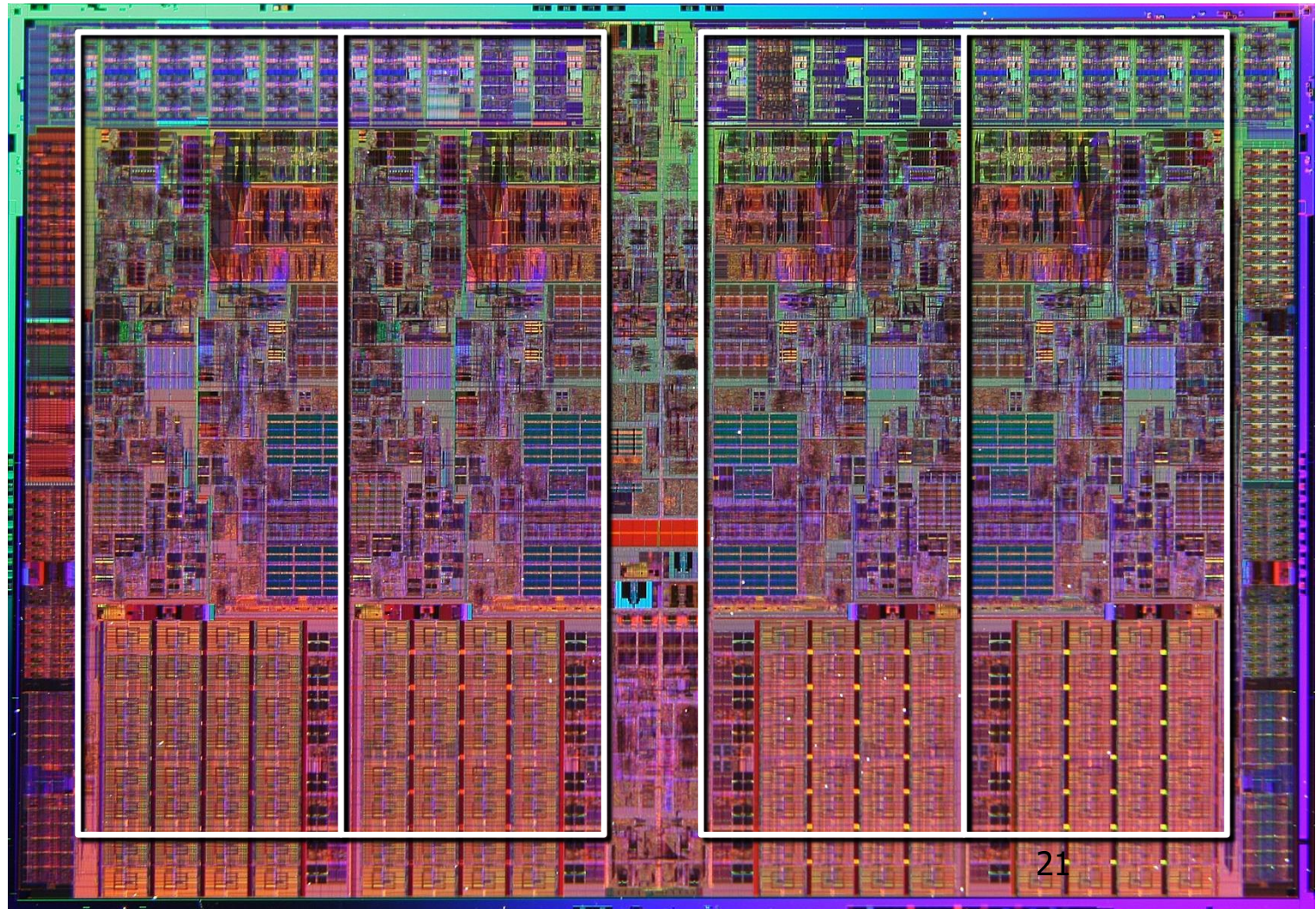
- Why use hw multithreading?
 - **Multithreading improves utilization and throughput**
 - Single programs utilize <50% of pipeline (branch, cache miss)
 - allow insns from different hw threads in pipeline at once
 - **Multithreading does not improve single-thread performance**
 - Individual threads run as fast or even slower
 - **Coarse-grain MT**: switch on cache misses Why?
 - **Simultaneous MT**: no explicit switching, fine-grain interleaving
 - Intel's "hyperthreading"



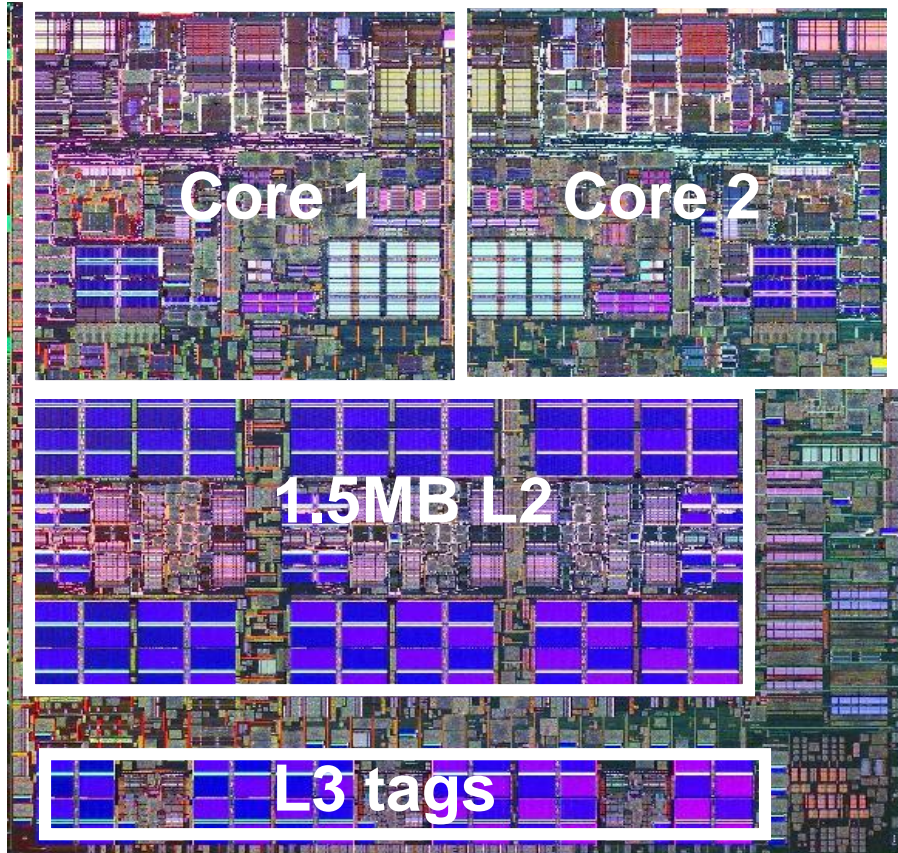
Multicore Hardware



Intel Quad-Core "Core i7"



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 tag
- **AMD Quad Phenom**
 - Four 2+ GHz cores
 - Per-core 512KB L2 cache
 - Shared 2MB L3 cache
- **Intel Core i7 Quad**
 - Four cores, private L2 cache
 - Shared 8 MB L3
- **Sun Niagara**
 - 8 cores, each 4-way threaded
 - Shared 2MB L2
 - For servers, not desktop

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

Three Shared Memory Issues



1. Cache coherence

- If cores have private (non-shared) caches
- How to make writes to one cache “show up” in others?

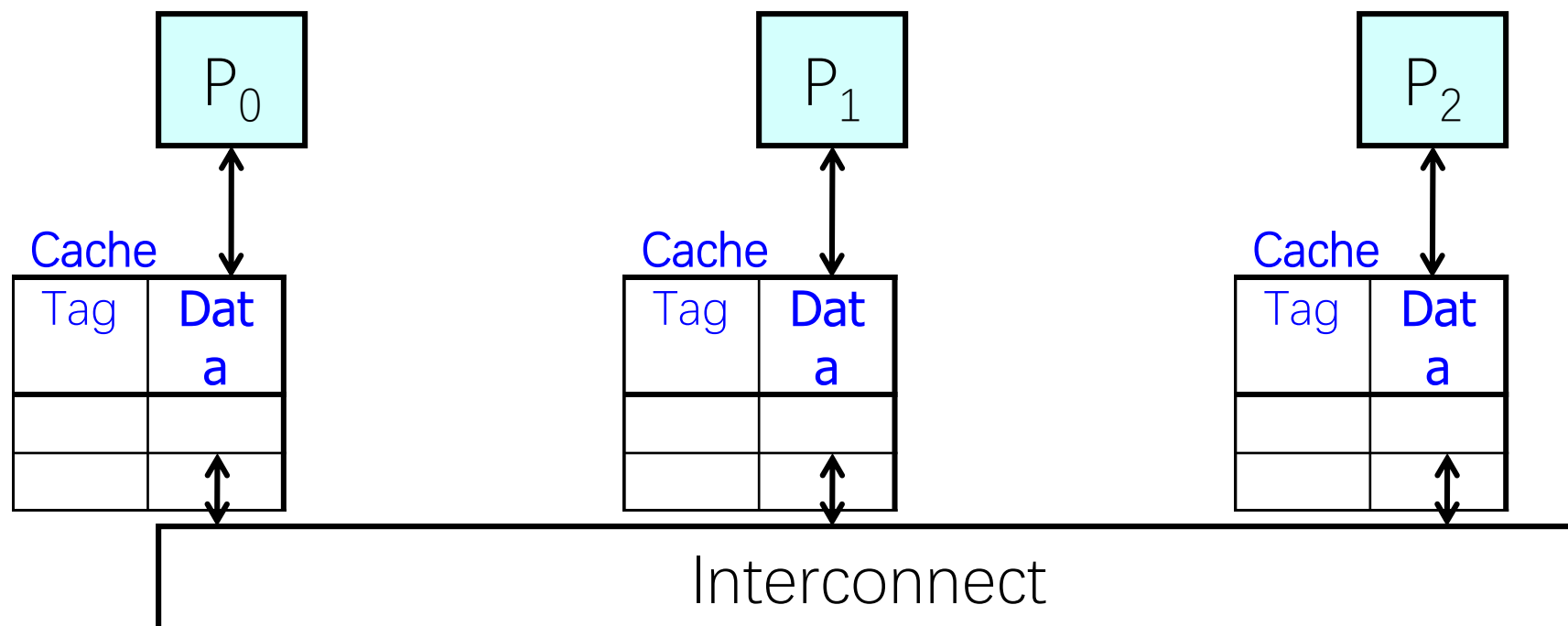
2. Synchronization

- How to regulate access to shared data?
- How to implement “locks”?

3. Memory consistency models

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

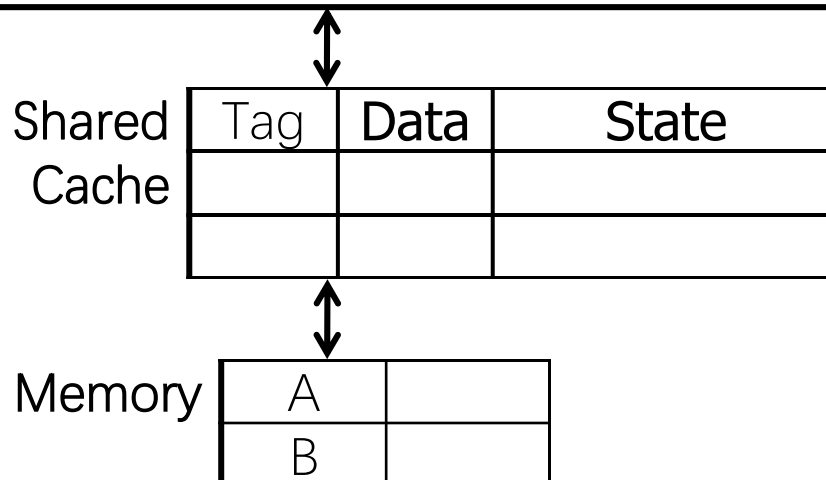
Adding Private Caches



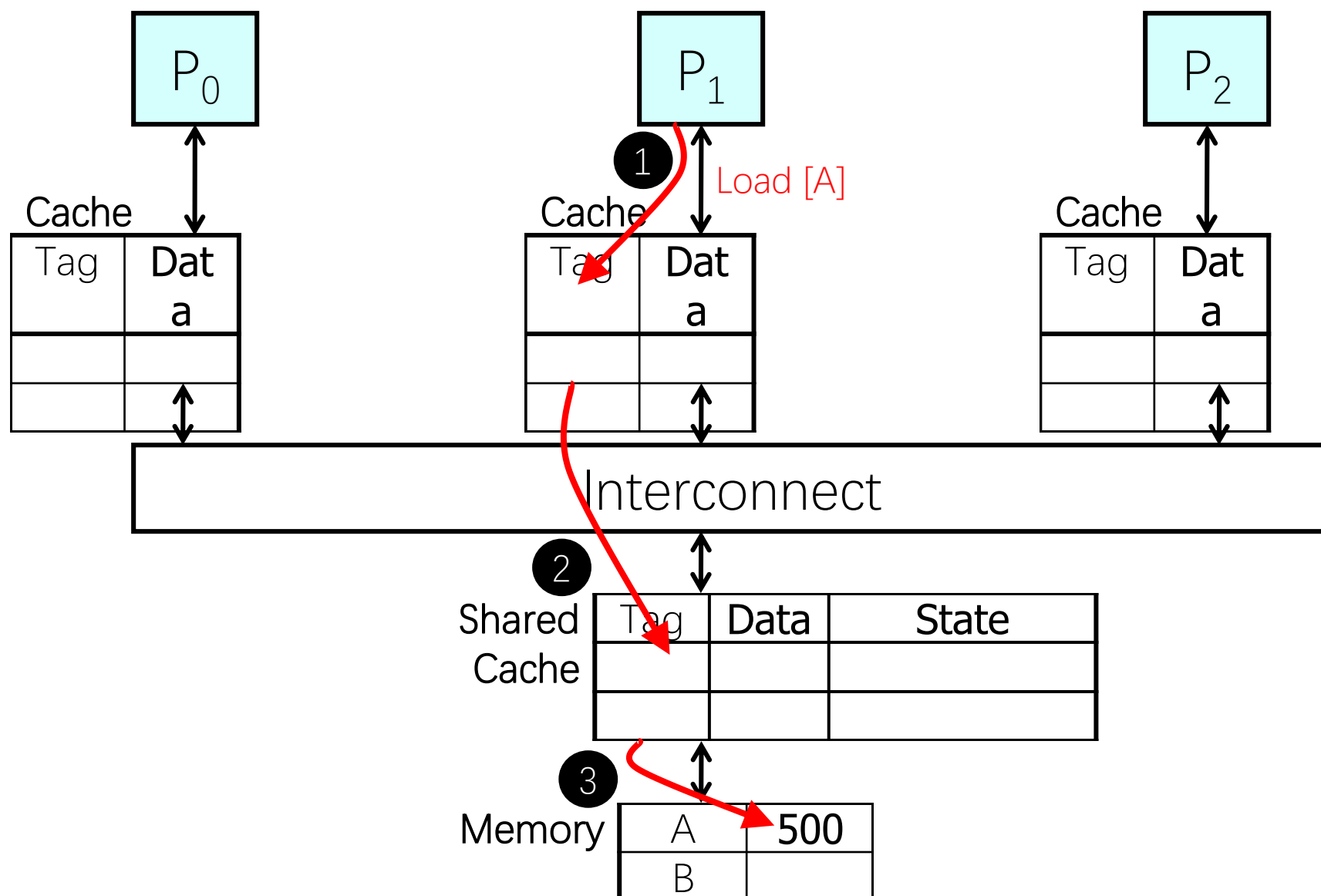
▪ Add per-core caches

(write-back caches)

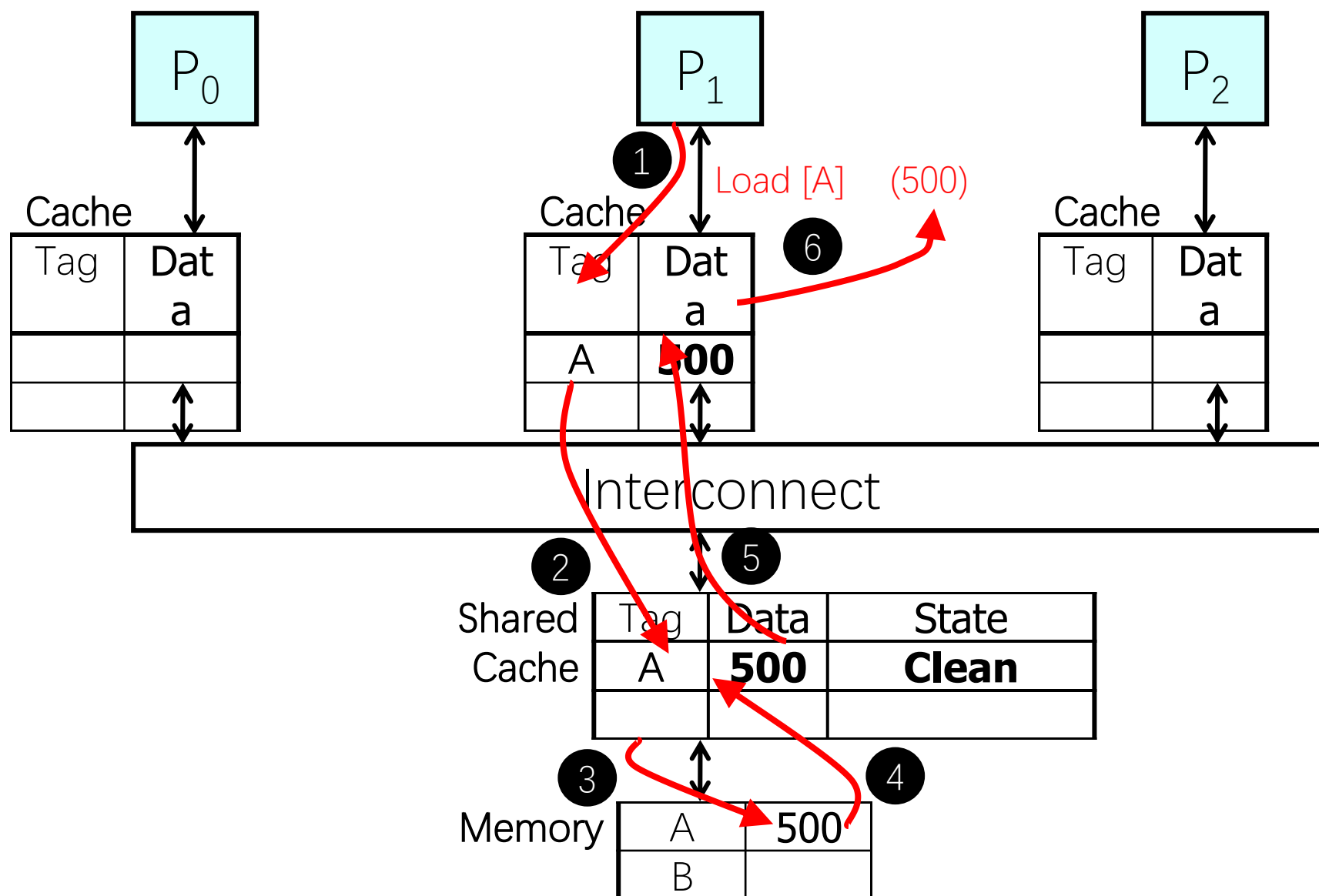
- Reduces latency
- Increases throughput
- Decreases energy



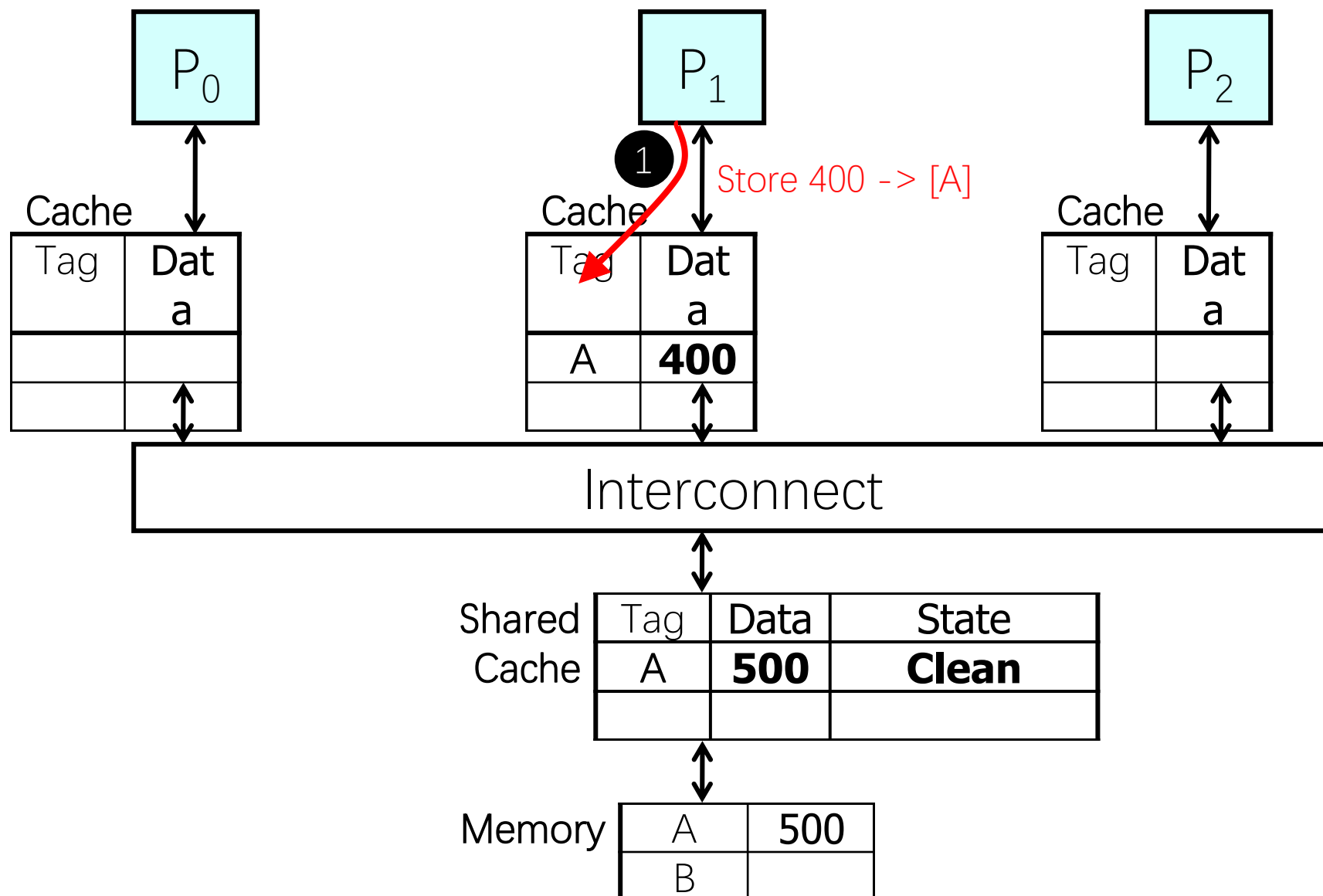
Adding Private Caches



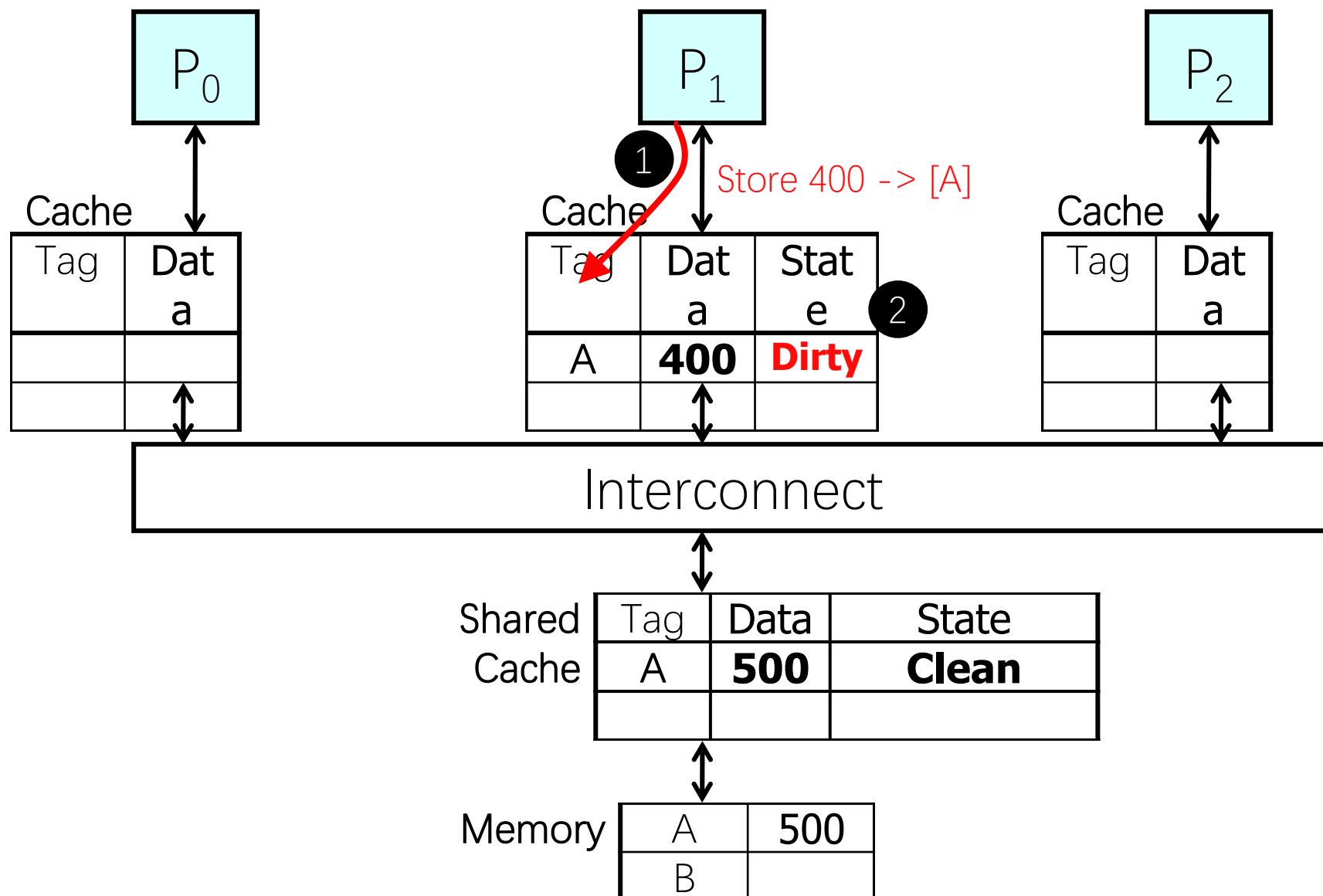
Adding Private Caches



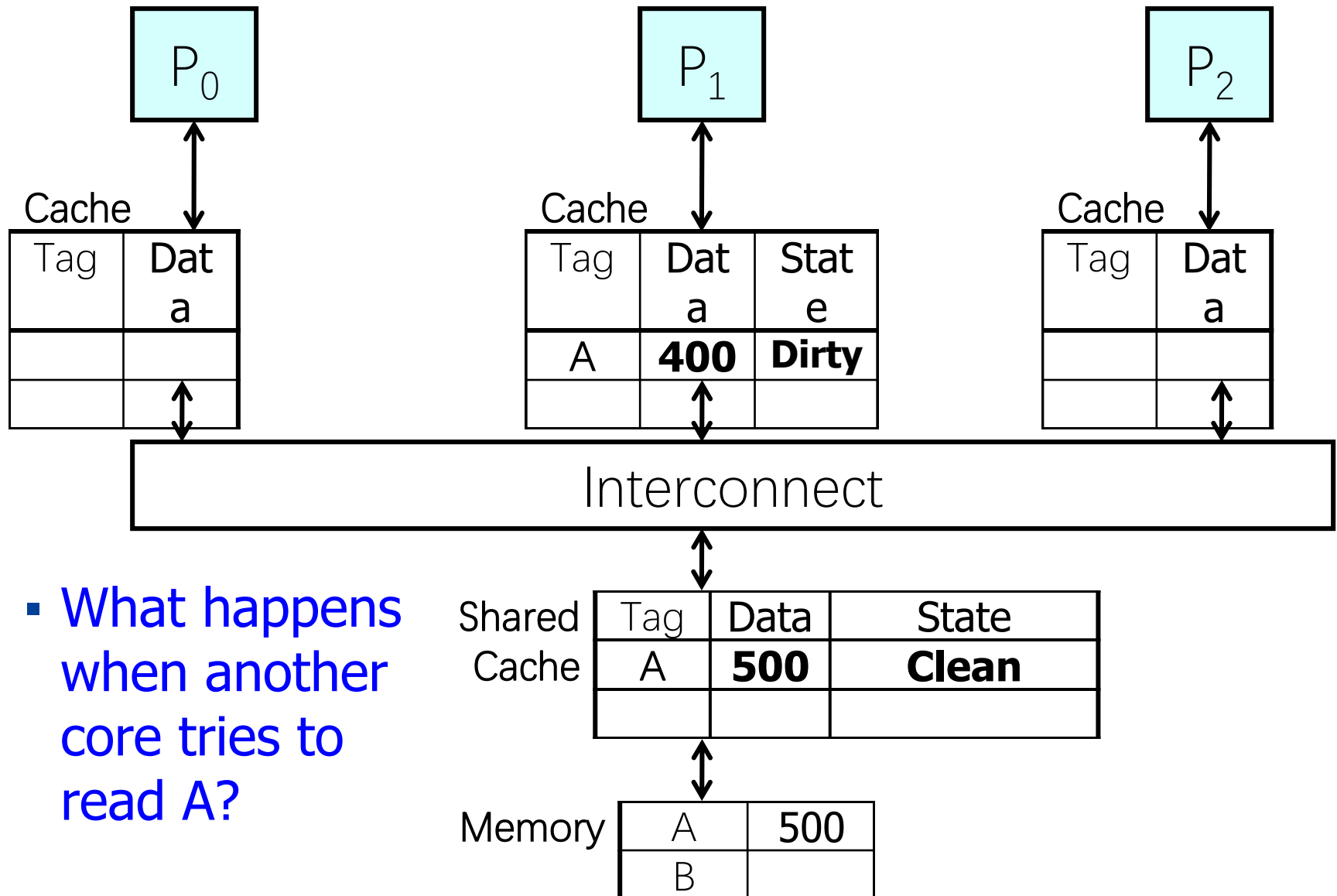
Adding Private Caches



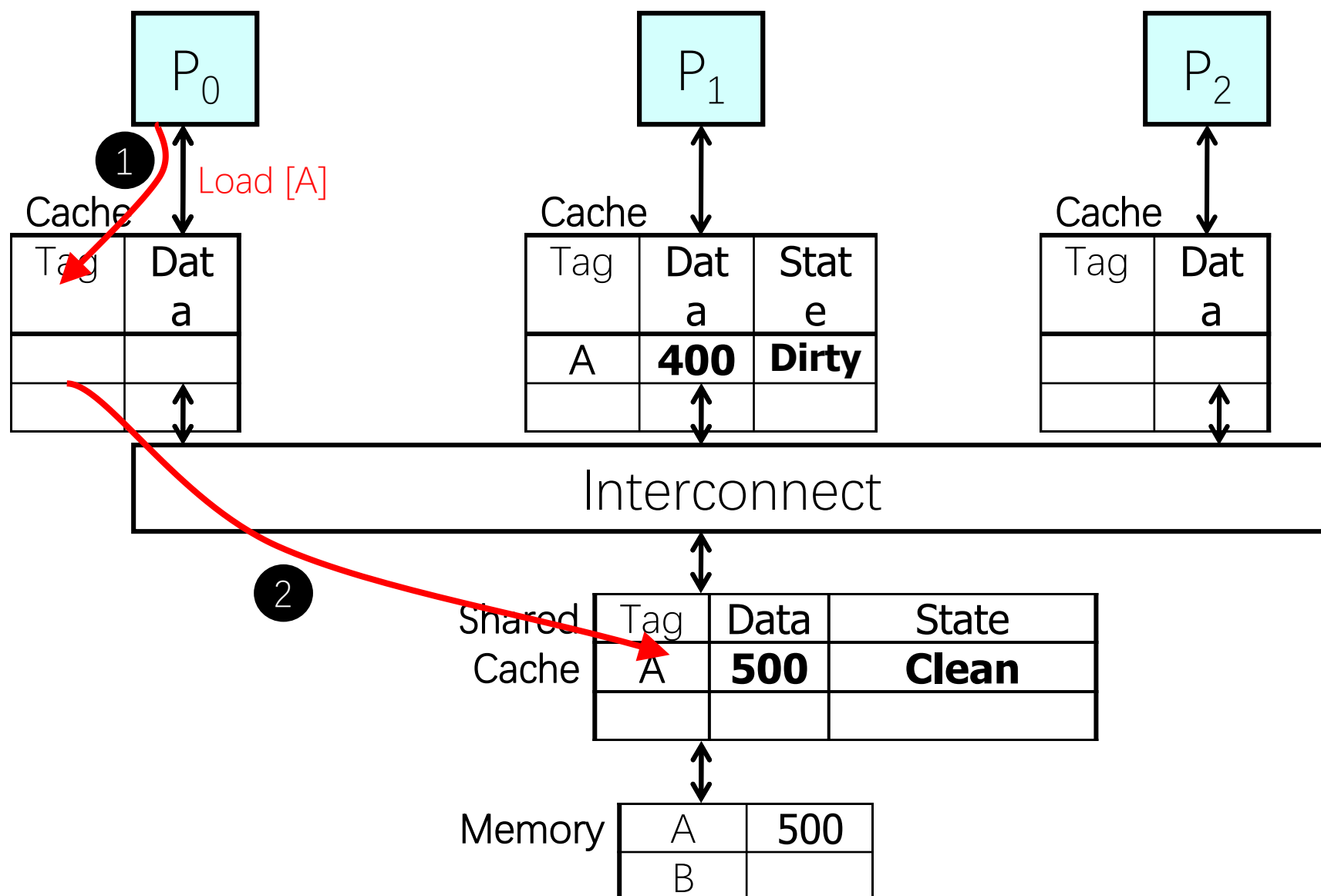
Adding Private Caches



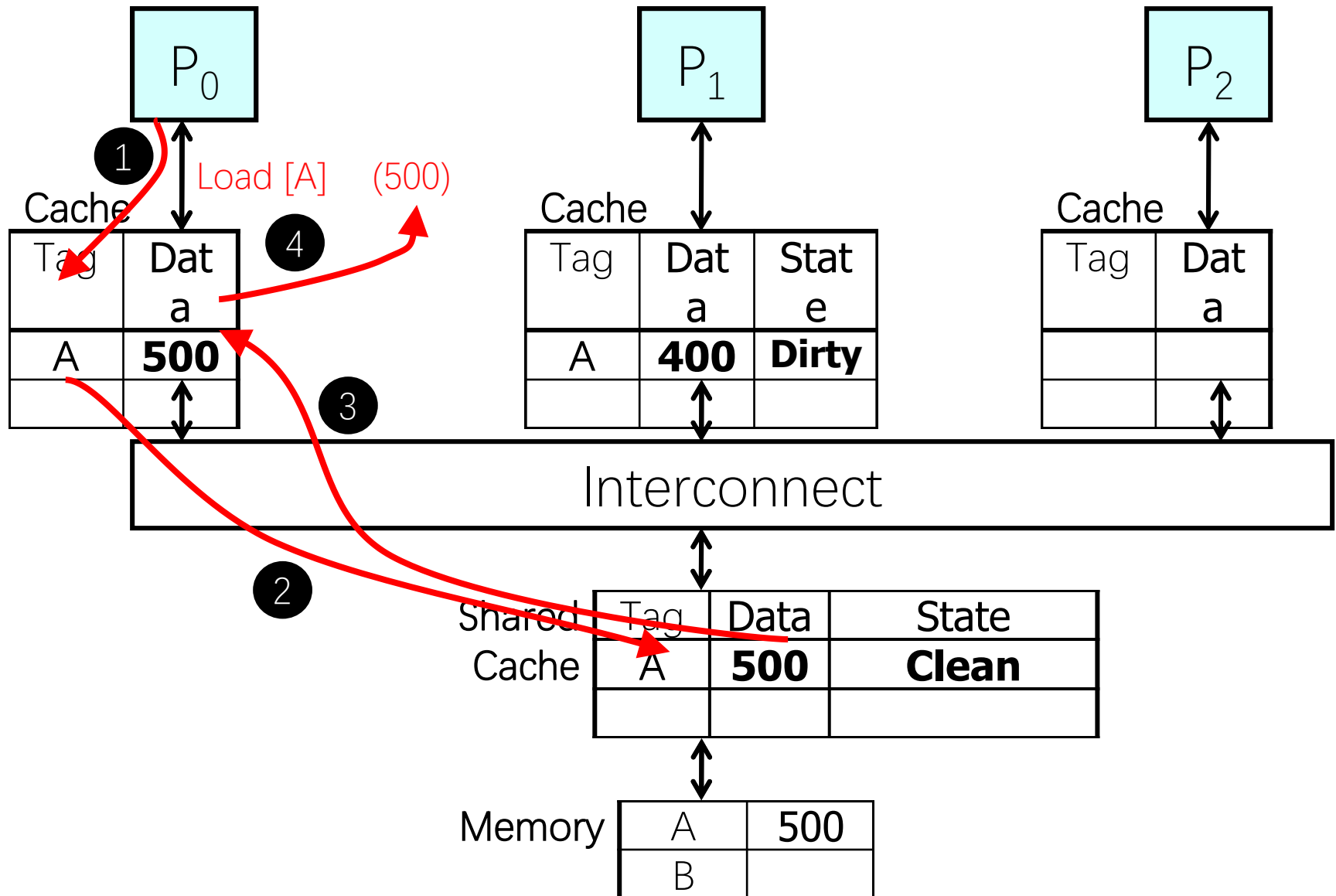
Private Cache Problem: Incoherence



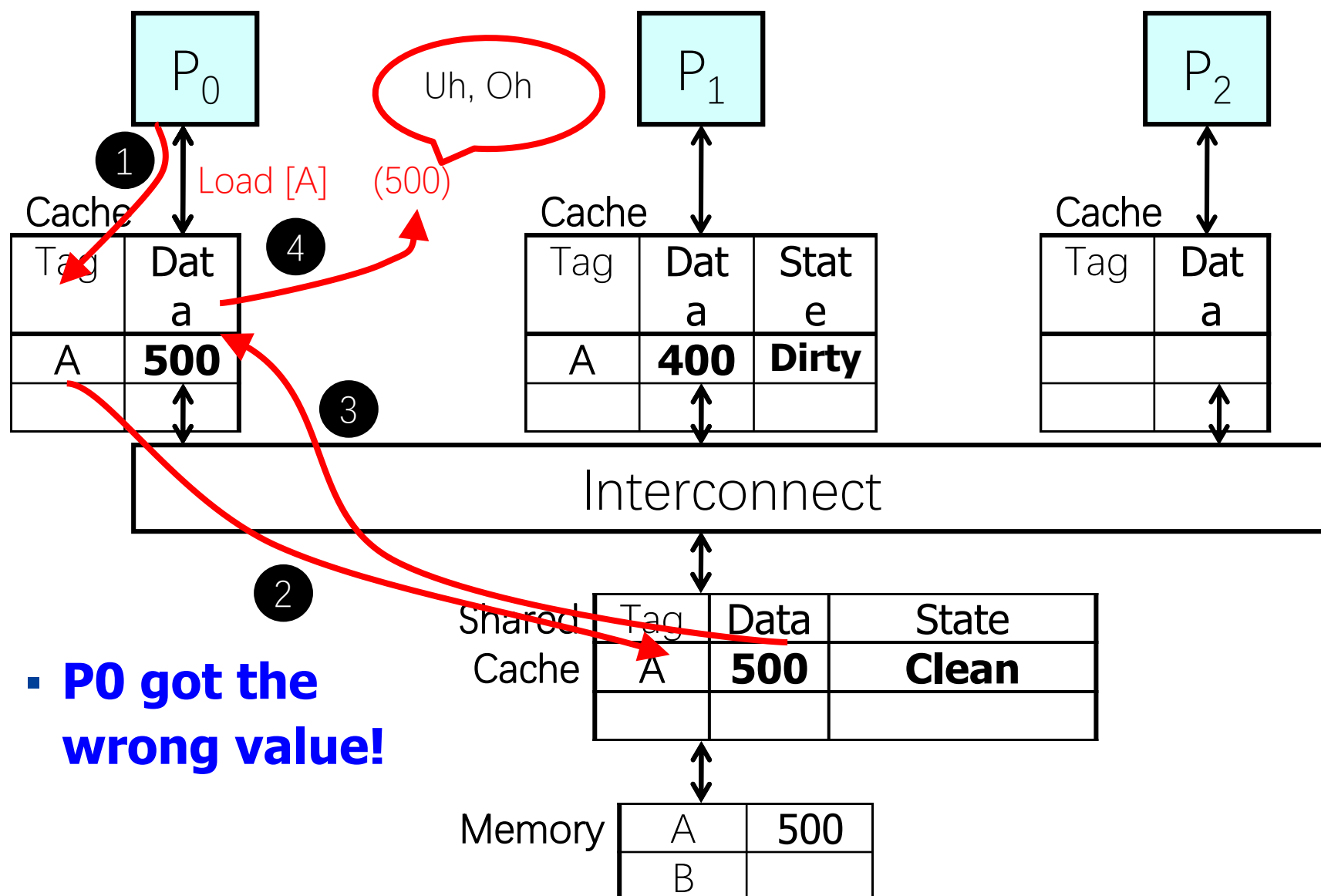
Private Cache Problem: Incoherence



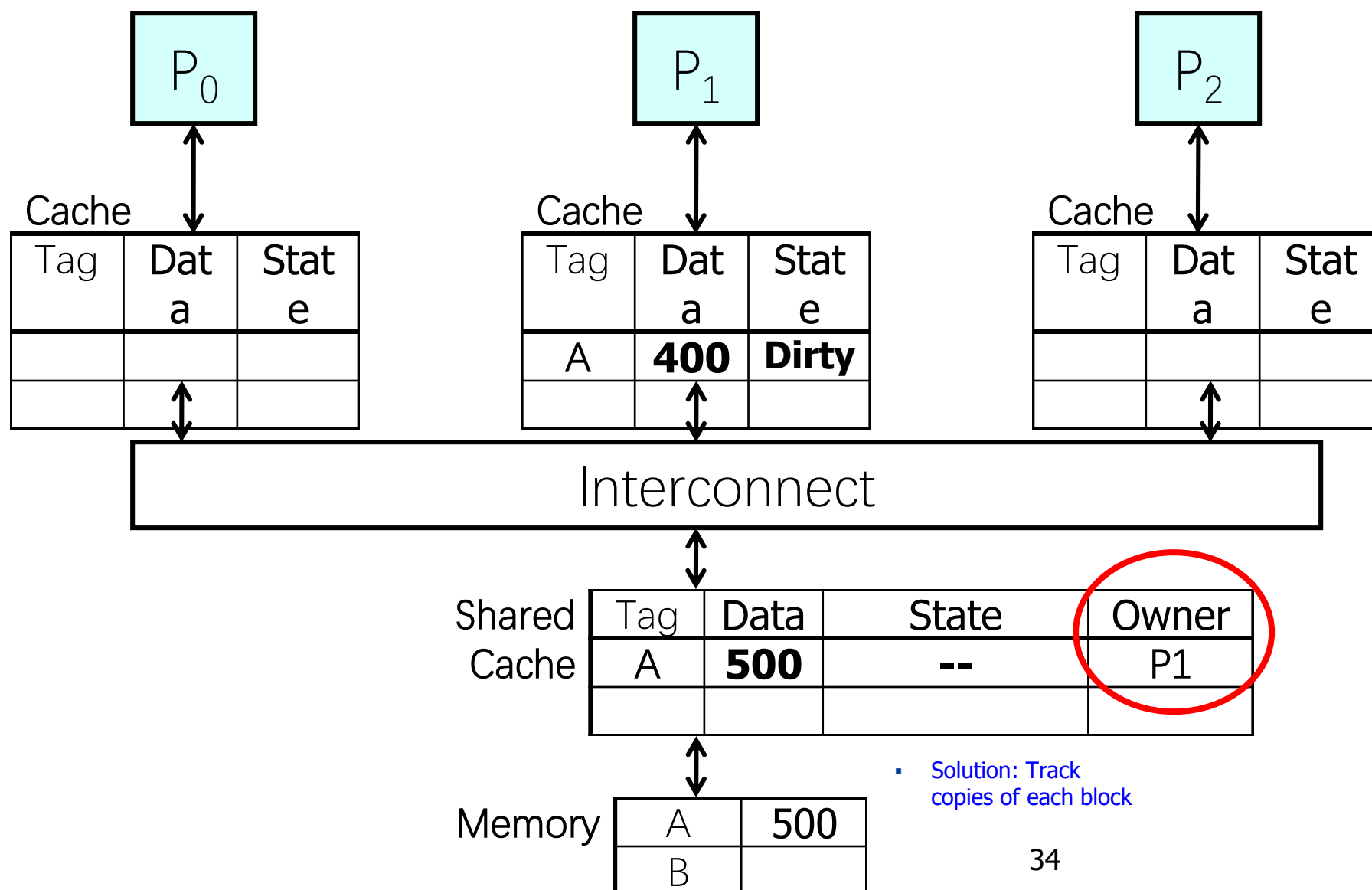
Private Cache Problem: Incoherence



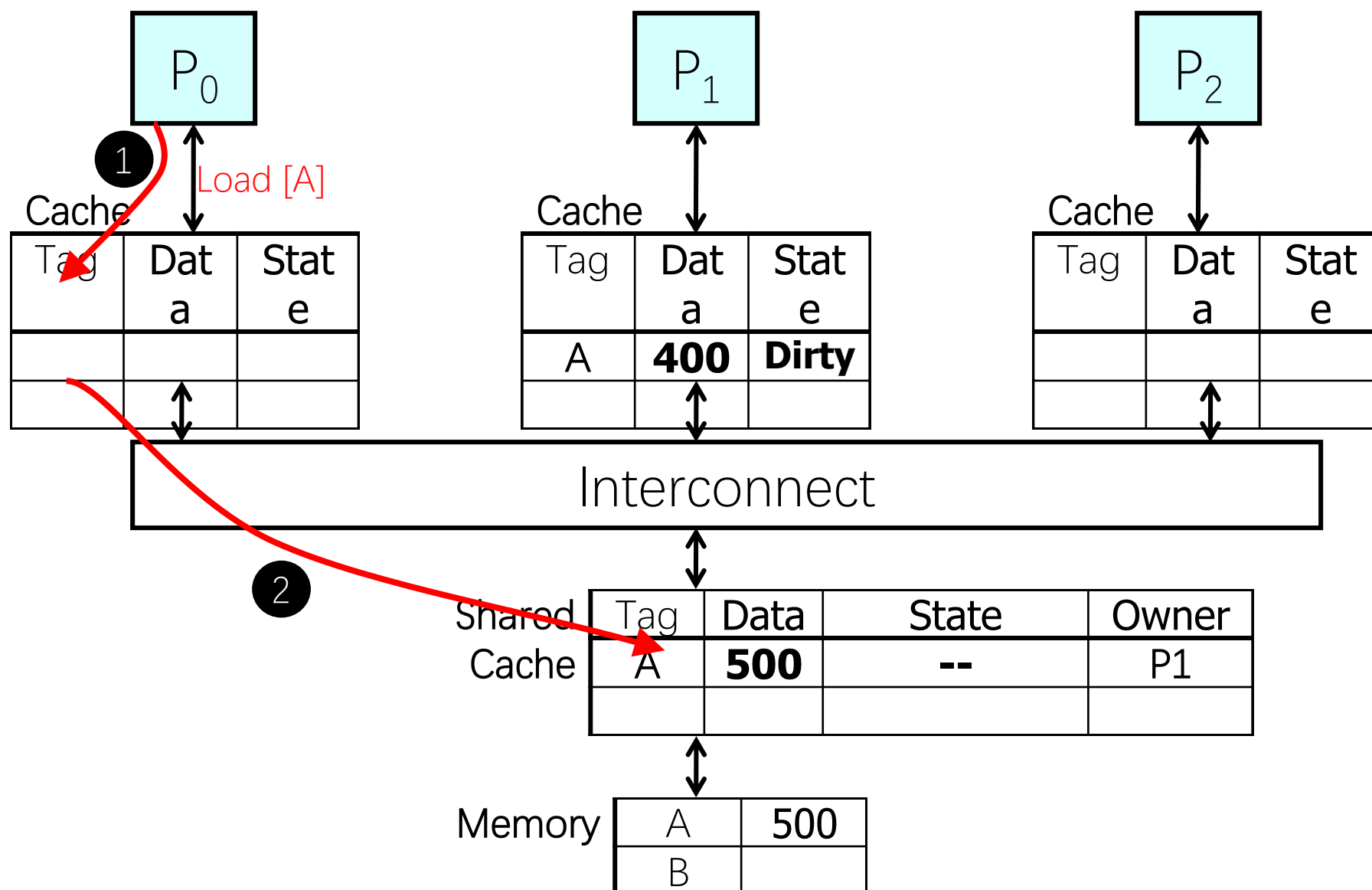
Private Cache Problem: Incoherence



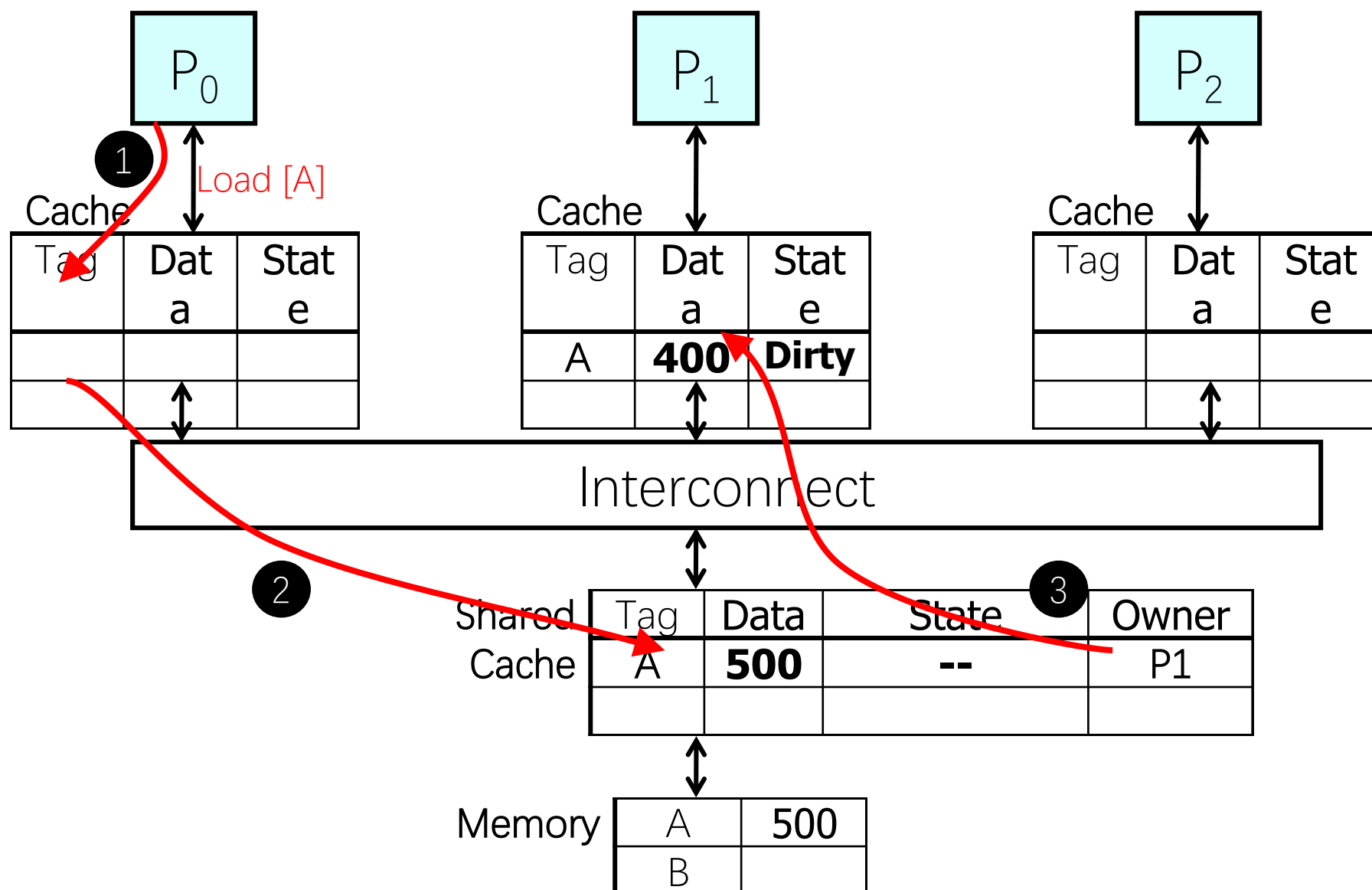
Rewind: Fix Problem by Tracking Sharers



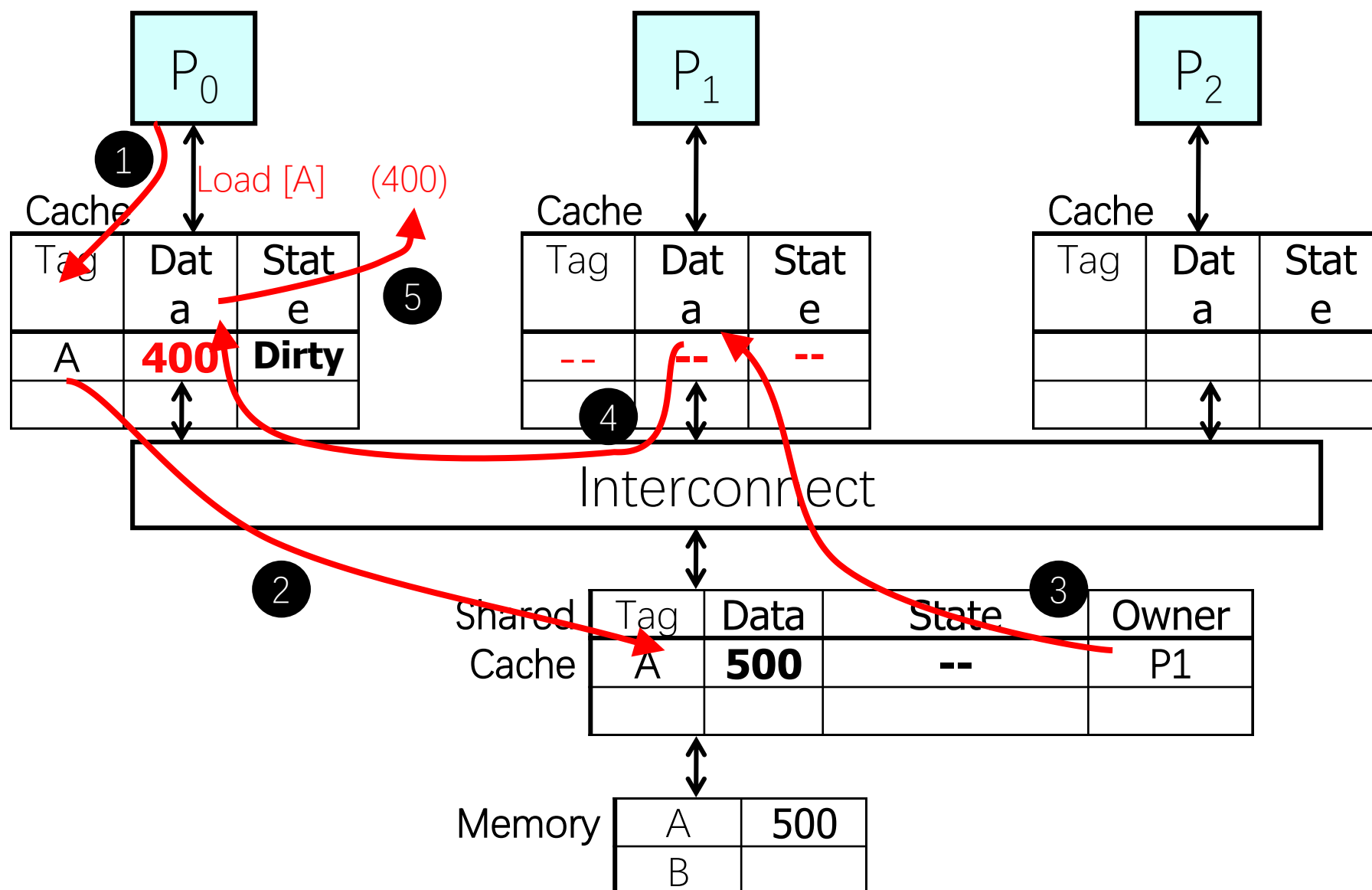
Use Tracking Information to “Invalidate”



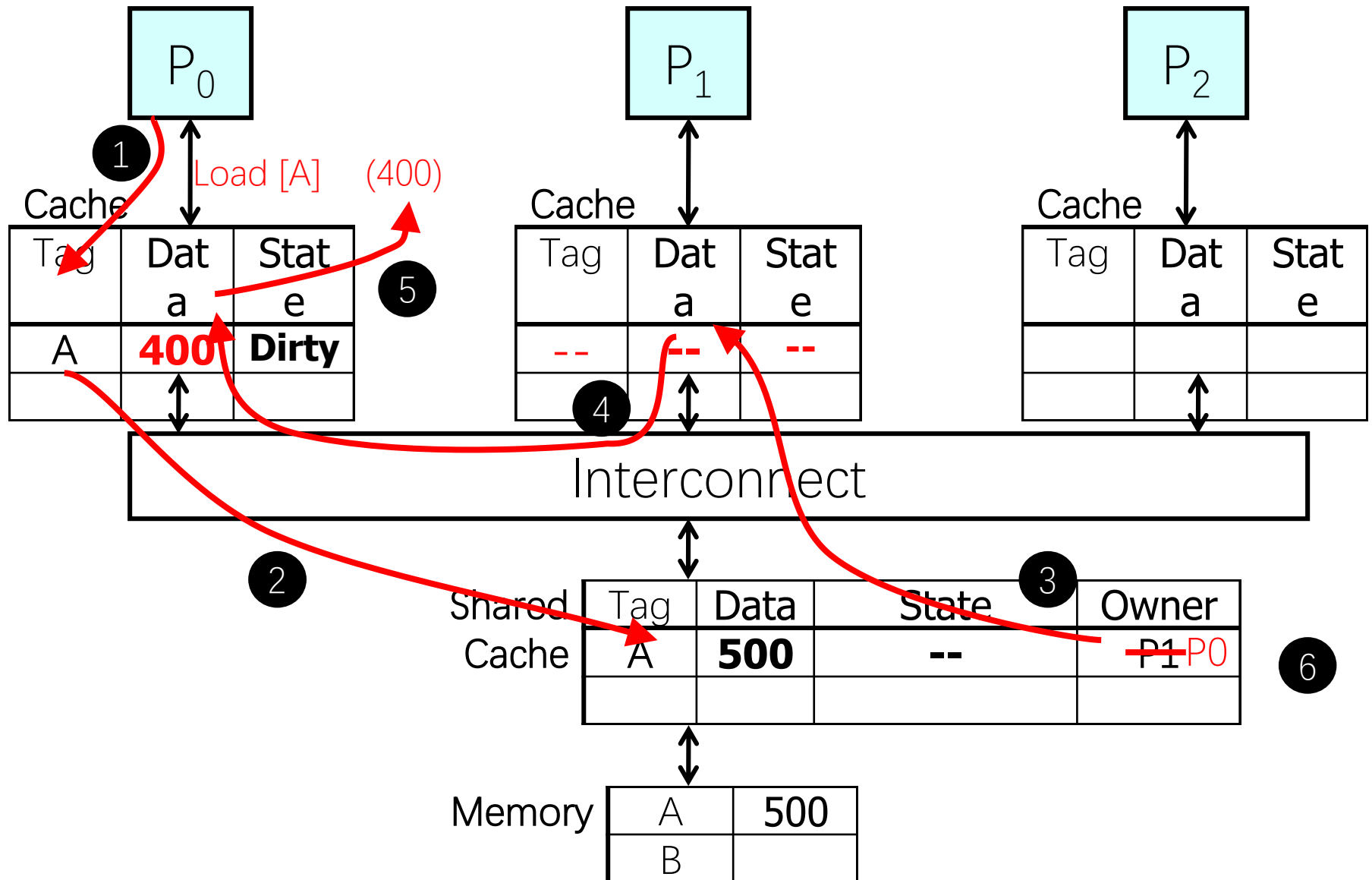
Use Tracking Information to "Invalidate"



Use Tracking Information to "Invalidate"



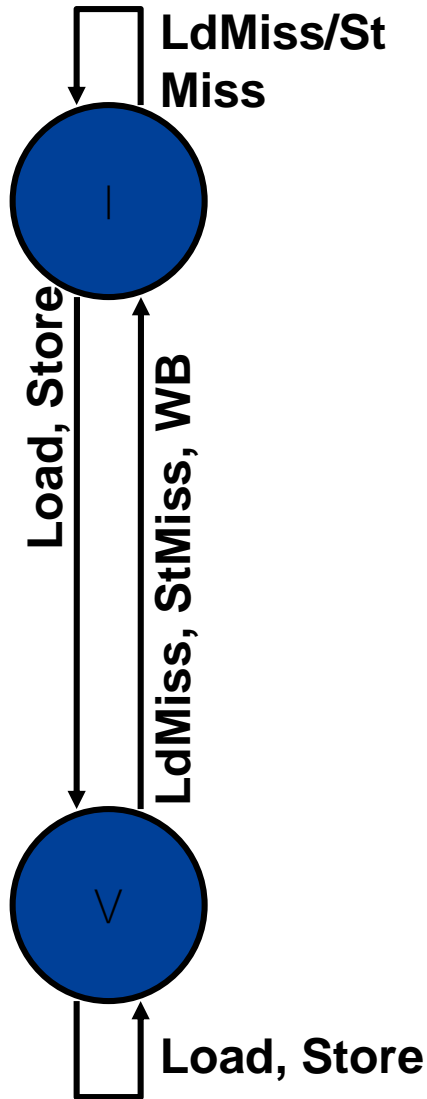
Use Tracking Information to “Invalidate”



“Valid/Invalid” Cache Coherence

- To enforce the shared memory invariant...
 - “Loads read the value written by the most recent store”
- Enforce the invariant...
 - **“At most one valid copy of the block”**
 - Simplest form is a **two-state “valid/invalid” protocol**
 - If a core wants a copy, must find and “invalidate” it
- On a cache miss, how is the valid copy found?
 - Option #1 **“Snooping”**: broadcast to all, whoever has it responds
 - Option #2: **“Directory”**: track sharers with separate structure
- **Problem**: multiple copies can’t exist, even if read-only
 - Consider mostly-read data structures, instructions, etc.

VI (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

State	<i>This Processor</i>		<i>Other Processor</i>	
	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**

False sharing problem



- A cache block contains more than one word
- Cache-coherence is done at the block-level and not word-level
- Suppose P_1 writes word _{i} and P_2 writes word _{k} and both words have the same block address.
- *What can happen?*

多处理器cache一致性问题：例题



- 如下代码在SMP环境下执行，sum和sum_local是全局变量，被NUM_THREADS个线程所共享：
- `double sum=0.0, sum_local[NUM_THREADS];`
- 解释代码所引起的cache假共享（false sharing）问题，为什么cache假共享问题会影响多线程的性能。

```
#pragma omp parallel num_threads(NUM_THREADS)
//由NUM_THREADS个线程执行以下相同的代码段
{
    int me = omp_get_thread_num();
    sum_local[me] = 0.0;
```

#pragma omp for //并行for语句，不同的线程处理0~N中的一部分数据，将结果存入对应该线程的sum_local元素中

```
    for (i = 0; i < N; i++)
        sum_local[me] += x[i] * y[i];

    #pragma omp atomic //并行原子操作,
    sum += sum_local[me];
}
```



Parallel Programming

Example #1: Bank Accounts



- Consider

```
struct acct_t { int balance; ... };  
struct acct_t accounts[MAX_ACCT];           // current balances  
  
struct trans_t { int id; int amount; };  
struct trans_t transactions[MAX_TRANS];     // debit amounts  
  
for (i = 0; i < MAX_TRANS; i++) {  
    debit(transactions[i].id, transactions[i].amount);  
}  
  
void debit(int id, int amount) {  
    if (accounts[id].balance >= amount) {  
        accounts[id].balance -= amount;  
    }  
}
```

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

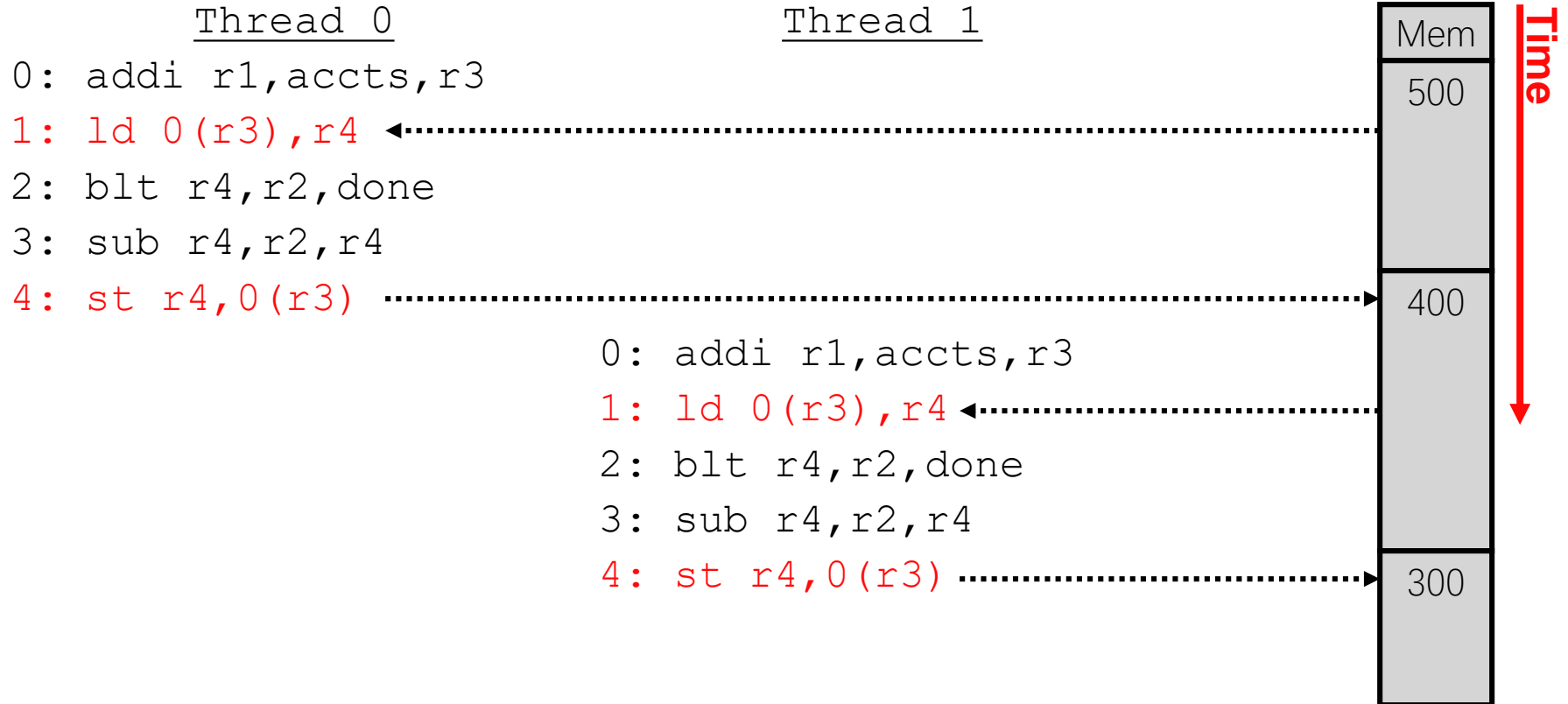
Example #1: Bank Accounts

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

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

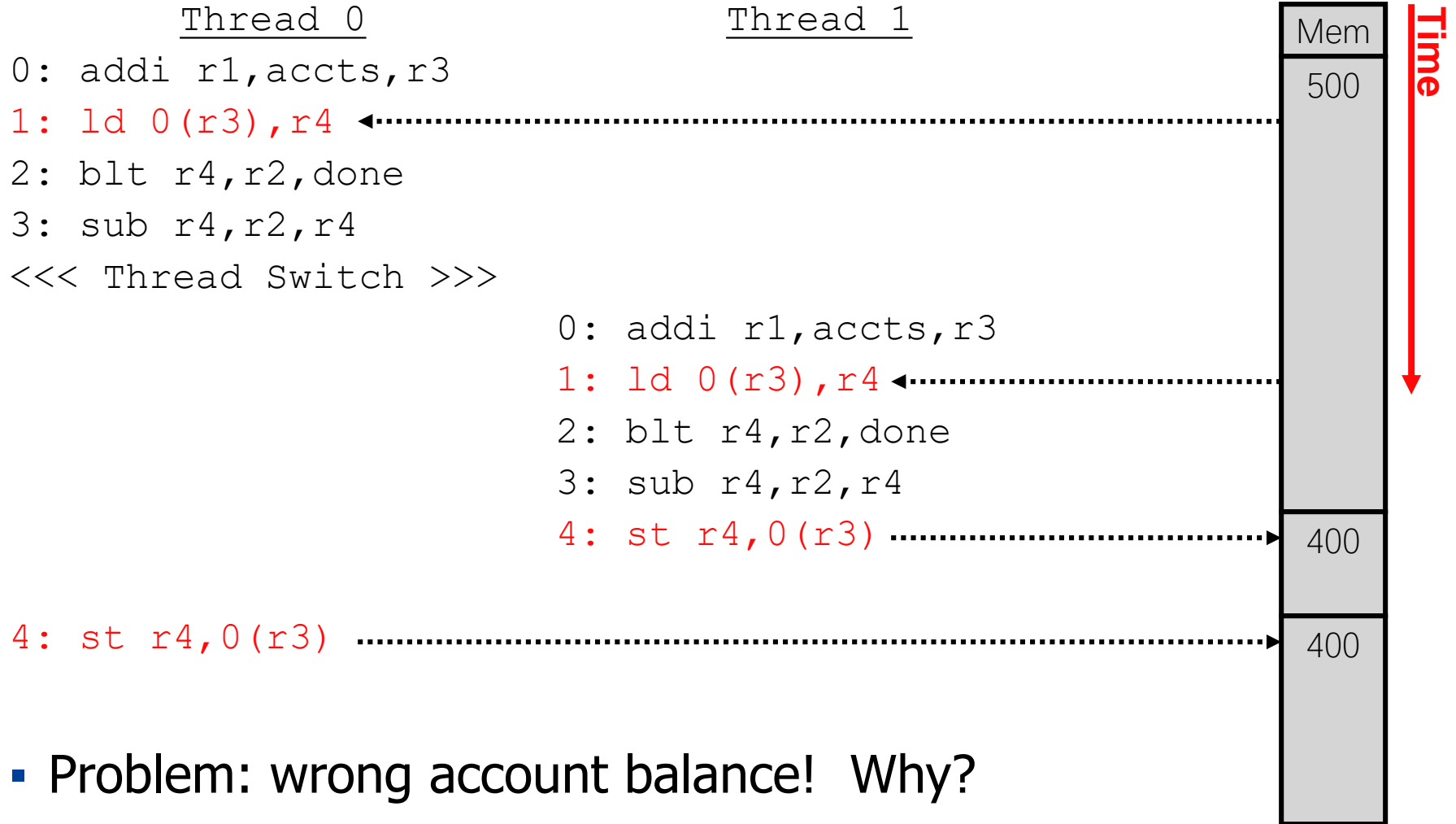
- Example of **Thread-level parallelism (TLP)**
 - Collection of asynchronous tasks: not started and stopped together
 - Data shared “loosely” (sometimes yes, mostly no), dynamically
- Example: database/web server (each query is a thread)
 - **accts** is global and thus **shared**, can’t register allocate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

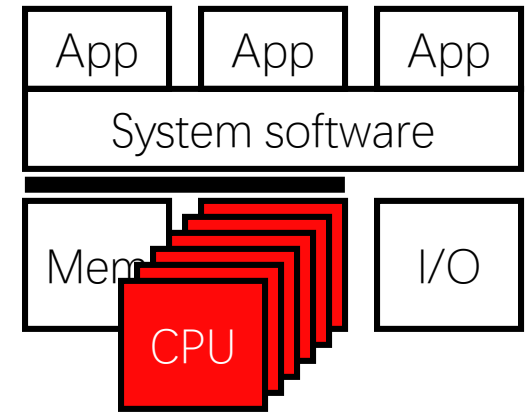


- Problem: wrong account balance! Why?
 - Solution: synchronize access to account balance

总结 Shared Memory Multiprocessors



- Thread-level parallelism (TLP)
- Shared memory model
 - Hardware multithreading
 - Multiprocessing
- Cache coherence
 - Valid/Invalid, MSI, False sharing
- Synchronization
- Memory consistency models



Suggested reading



- “A Primer on Memory Consistency and Cache Coherence”
(Synthesis Lectures on Computer Architecture) by Daniel Sorin, Mark Hill, and David Wood, November 2011
- “Why On-Chip Cache Coherence is Here to Stay”
by Milo Martin, Mark Hill, and Daniel Sorin,
Communications of the ACM (CACM), July 2012.

谢谢！

