Multiprocessors and Caching

CS 1541

Wonsun Ahn

University of Pittsburgh

Two ways to use multiple processors

- **Distributed (Memory) System**
  o Processors **do not share memory** (and by extension **data**)
  o Processors exchange data through network messages
  o Programming standards:
    - Message Passing Interface (MPI) – C/C++ API for exchanging messages
    - Ajax (Asynchronous JavaScript and XML) – API for web apps
  o Data exchange protocols: TCP/IP, UDP/IP, JSON, XML…

- **Shared Memory System** (a.k.a. Multiprocessor System)
  o Processors **share memory** (and by extension **data**)
  o Programming standards:
    - Pthreads (POSIX threads), Java threads – APIs for threading
    - OpenMP – Compiler #pragma directives for parallelization
  o **Cache coherence protocol**: protocol for exchanging data among caches
  → Just like Ethernet, caches are part of a larger network of caches

- What bad thing can happen when you have shared data?

- Dataraces!
  - You should have learned it in CS 449.
  - But if you didn't, don't worry I'll go over it.

Review: Datarace Example

```c
int shared = 0;
void *add(void *unused) {
  for(int i=0; i < 1000000; i++) { shared++; }
  return NULL;
}
int main() {
  pthread_t t;
  // Child thread starts running add
  pthread_create(&t, NULL, add, NULL);
  // Main thread starts running add
  add(NULL);
  // Wait until child thread completes
  pthread_join(t, NULL);
  printf("shared=%d\n", shared);
  return 0;
}
```

bash-4.2$ ./datarace

shared=1085894

bash-4.2$ ./datarace

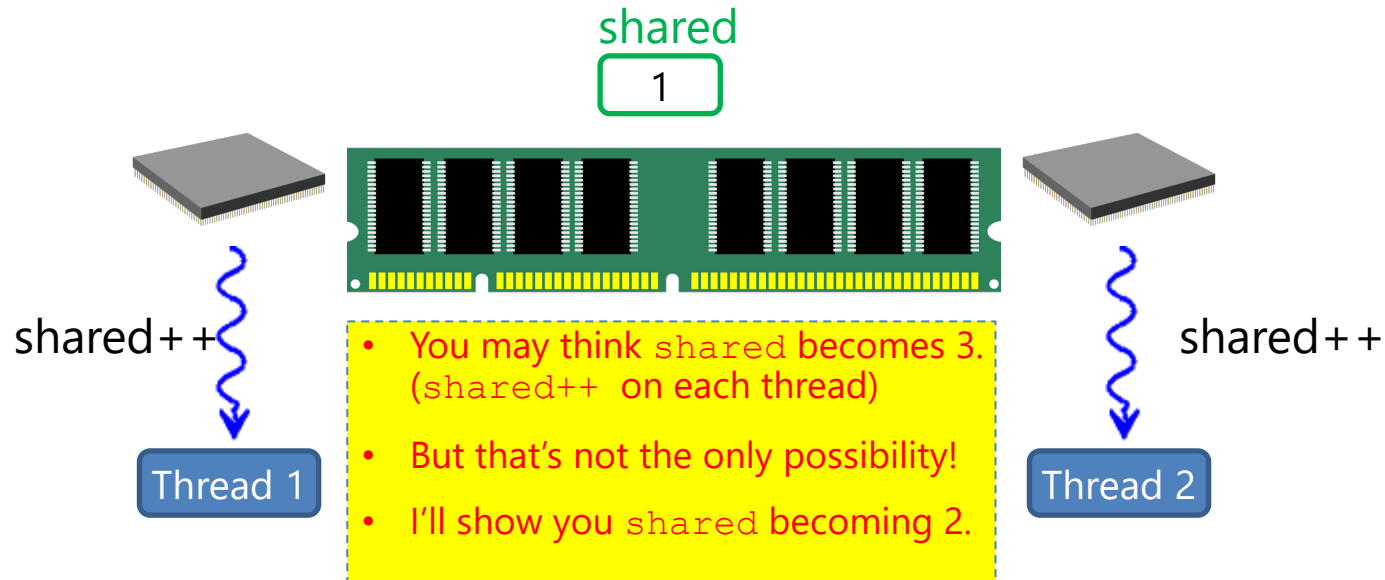shared=1101173

bash-4.2$ ./datarace

shared=1065494

Q) What do you expect from running this? Maybe shared=2000000 ?

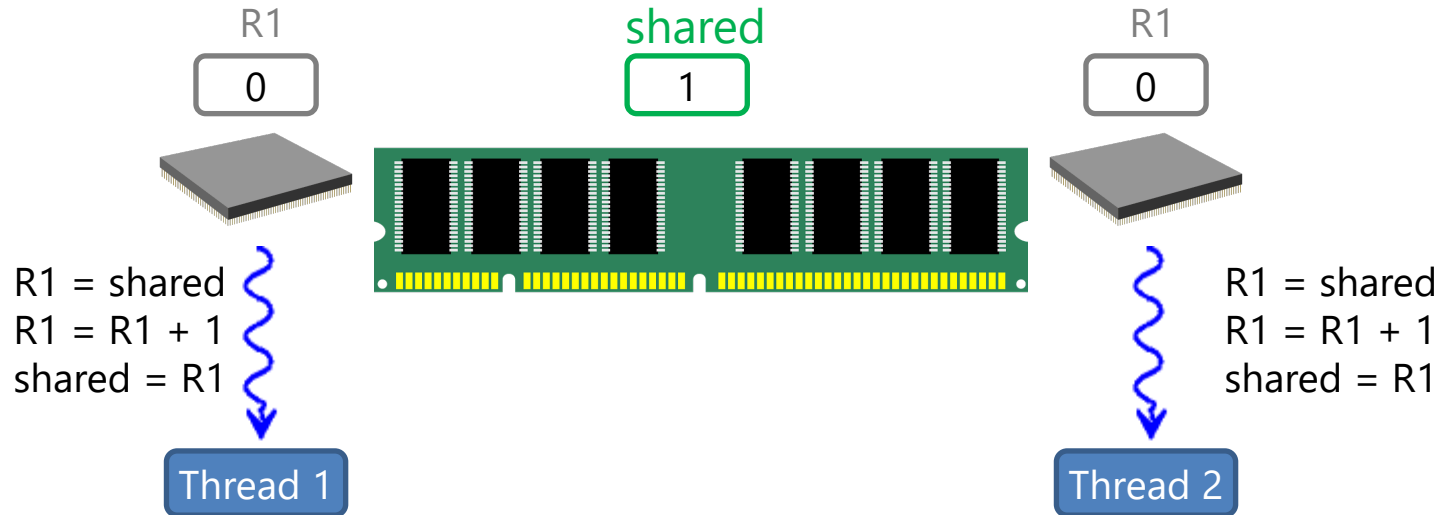A) Nondeterministic result! Due to **datarace** on `shared`.

University of Pittsburgh

- When two threads do `shared++;` initially `shared = 1`
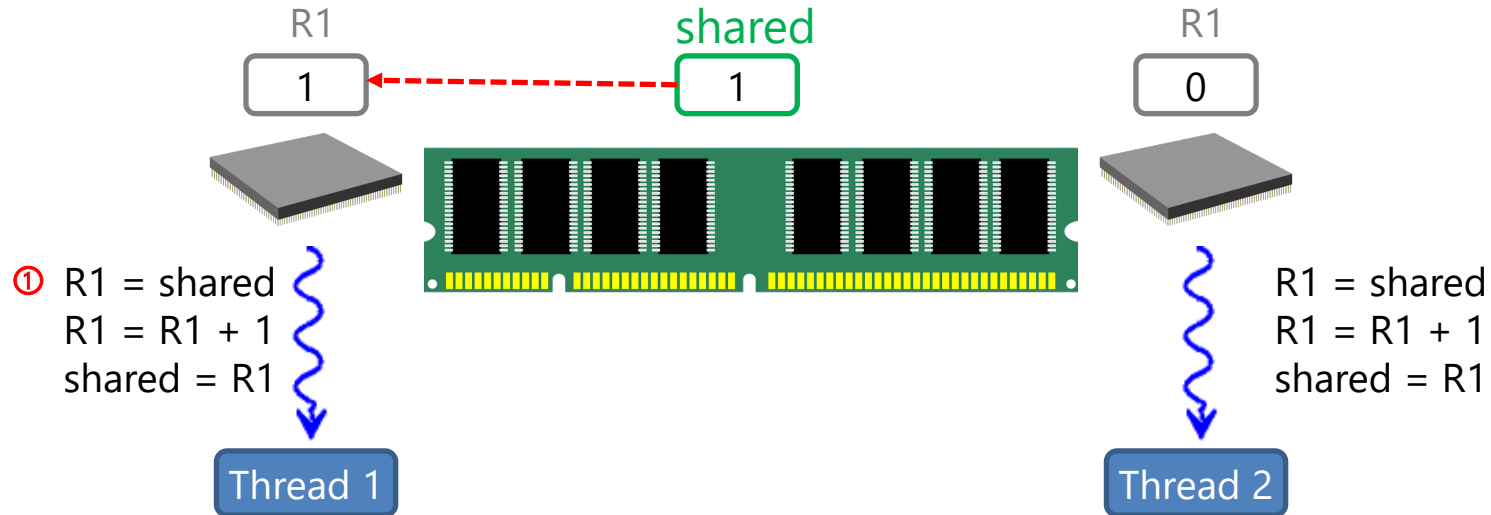
shared

1

shared++

Thread 1

shared++

Thread 2

- You may think `shared` becomes 3. (`shared++` on each thread)
- But that's not the only possibility!
- I'll show you `shared` becoming 2.

University of Pittsburgh

- When two threads do `shared++;` initially `shared` = 1

R1

0

shared

1

R1

0

R1 = shared
R1 = R1 + 1
shared = R1

R1 = shared
R1 = R1 + 1
shared = R1

Thread 1

Thread 2

- When two threads do `shared++;` initially `shared = 1`

- When two threads do `shared++;` initially `shared = 1`

R1

1

shared

1

R1

1

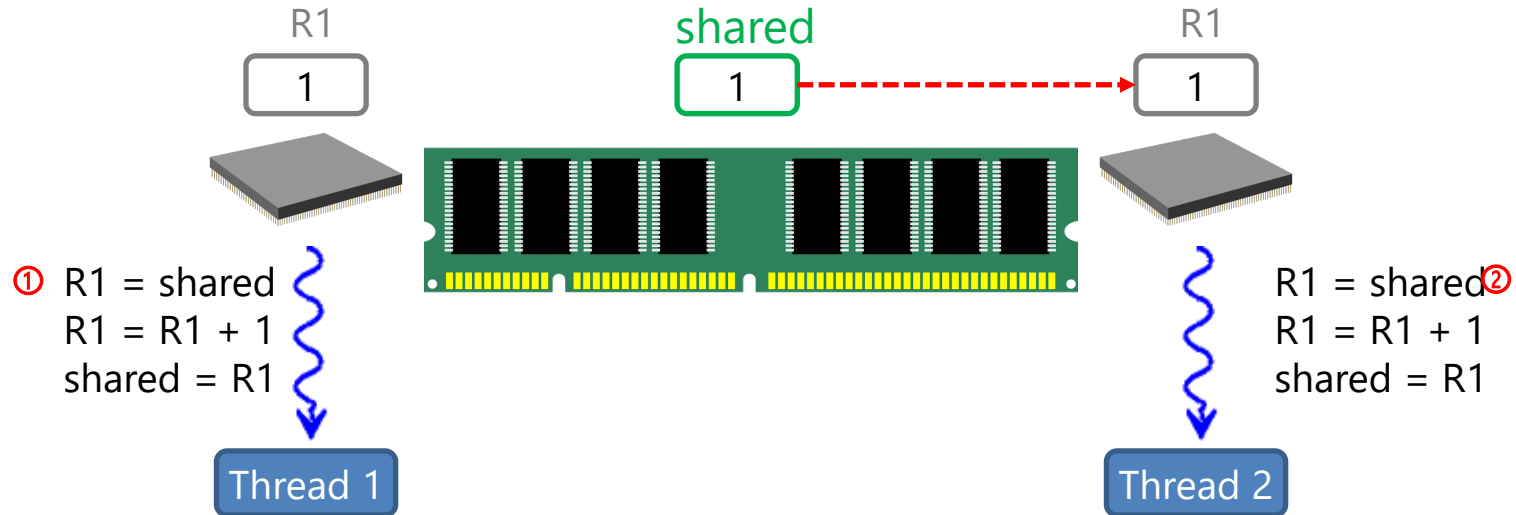① R1 = shared
R1 = R1 + 1
shared = R1

Thread 1

R1 = shared ②
R1 = R1 + 1
shared = R1

Thread 2

University of Pittsburgh

8

- When two threads do `shared++;` initially `shared = 1`

R1 | shared | R1
2 | 1 | 1

① R1 = shared
③ R1 = R1 + 1
shared = R1

Thread 1

R1 = shared ②
R1 = R1 + 1
shared = R1

Thread 2

- When two threads do `shared++;` initially `shared = 1`

- When two threads do `shared++;` initially `shared` = 1

R1
2

shared
2

R1
2

① R1 = shared
③ R1 = R1 + 1
⑤ shared = R1

R1 = shared②
R1 = R1 + 1④
shared = R1

Thread 1

Thread 2

- Why did this occur in the first place?

- Because data was **replicated** to CPU registers and each worked on its own copy!

● When two threads do `shared++;` initially `shared = 1`

R1

2

shared

2

R1

2

① R1 = shared
③ R1 = R1 + 1
⑤ shared = R1

R1 = shared ②
R1 = R1 + 1 ④
shared = R1 ⑥

- End result is 2 instead of 3!

- Only on simultaneous access (with this type of interleaving)

Thread 1

Thread 2

University of Pittsburgh
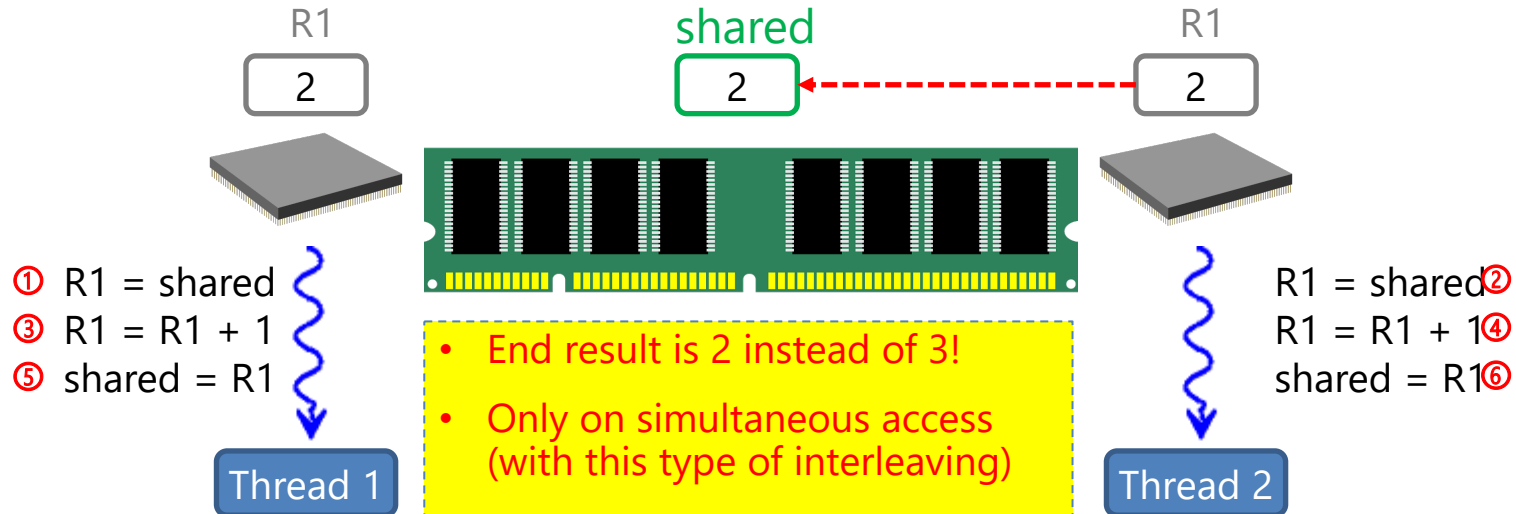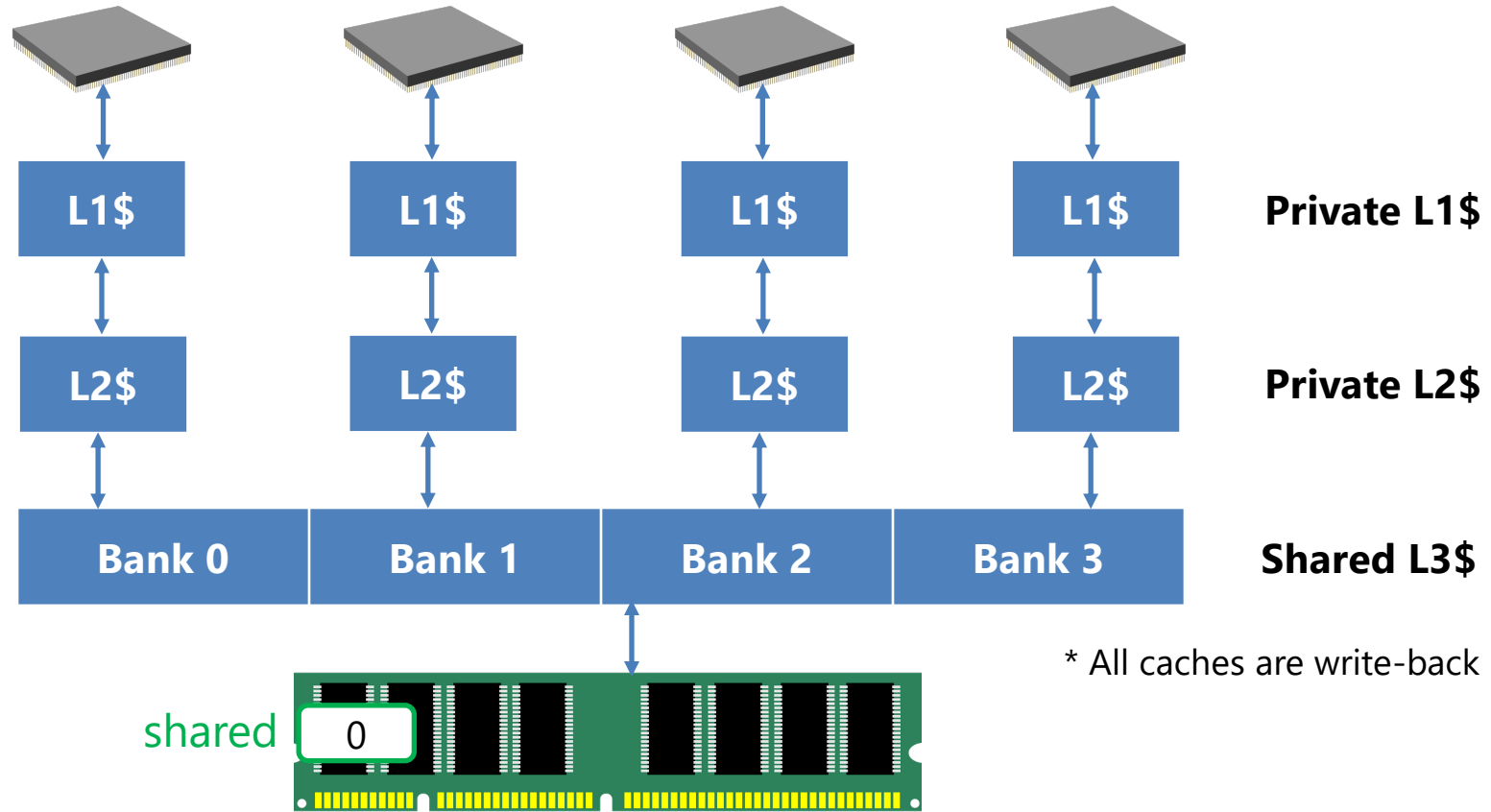
```
pthread_mutex_t lock;
int shared = 0;
void *add(void *unused) {
    for(int i=0; i < 1000000; i++) {
        pthread_mutex_lock(&lock);
        shared++;
        pthread_mutex_unlock(&lock);
    }
    return NULL;
}
int main() {
    …
}
```

bash-4.2$ ./datarace

shared=2000000

bash-4.2$ ./datarace

shared=2000000

bash-4.2$ ./datarace

shared=2000000

- Data race is fixed! Now shared is always 2000000.
- Problem solved? No! CPU registers is not the only place **replication** happens!

University of Pittsburgh
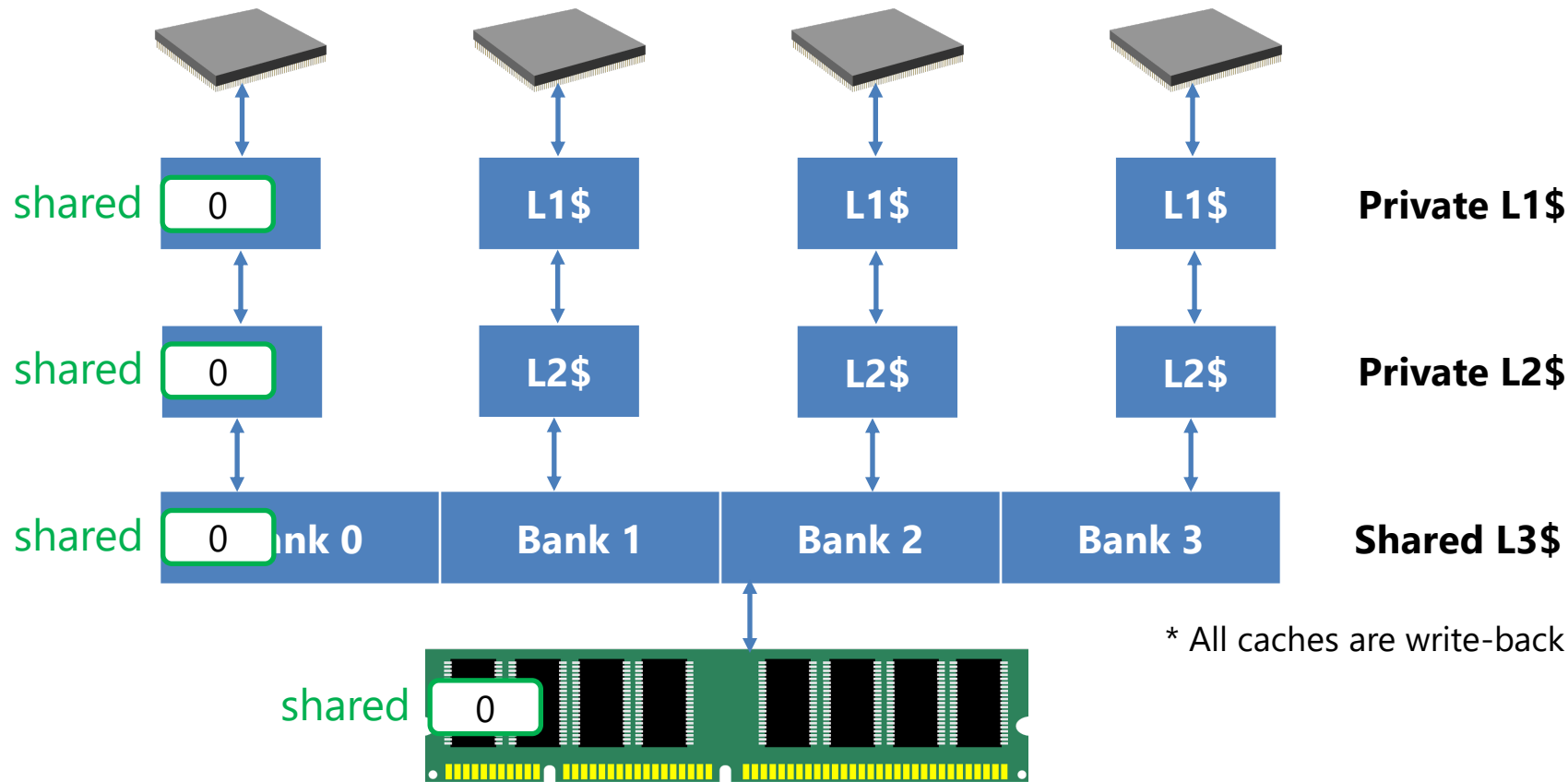
Caching also does replication!

- What happens if caches sit in between processors and memory?

Private L1$

Private L2$

Bank 0 | **Bank 1** | **Bank 2** | **Bank 3**   **Shared L3$**

* All caches are write-back

shared `0`

Caching also does replication!

- Let's say CPU 0 first fetches `shared` for incrementing

shared [0]   L1$ L1$ L1$ **Private L1$**

shared [0]   L2$ L2$ L2$ **Private L2$**

shared [0]nk 0   **Bank 1**   **Bank 2**   **Bank 3**   **Shared L3$**

* All caches are write-back

shared [0]

Caching also does replication!

- Then CPU 0 increments `shared` 100 times to 100

shared | 100 | L1$ L1$ L1$ **Private L1$**

shared | 0 | L2$ L2$ L2$ **Private L2$**

shared | 0 | Bank 0   Bank 1   Bank 2   Bank 3   **Shared L3$**

* All caches are write-back
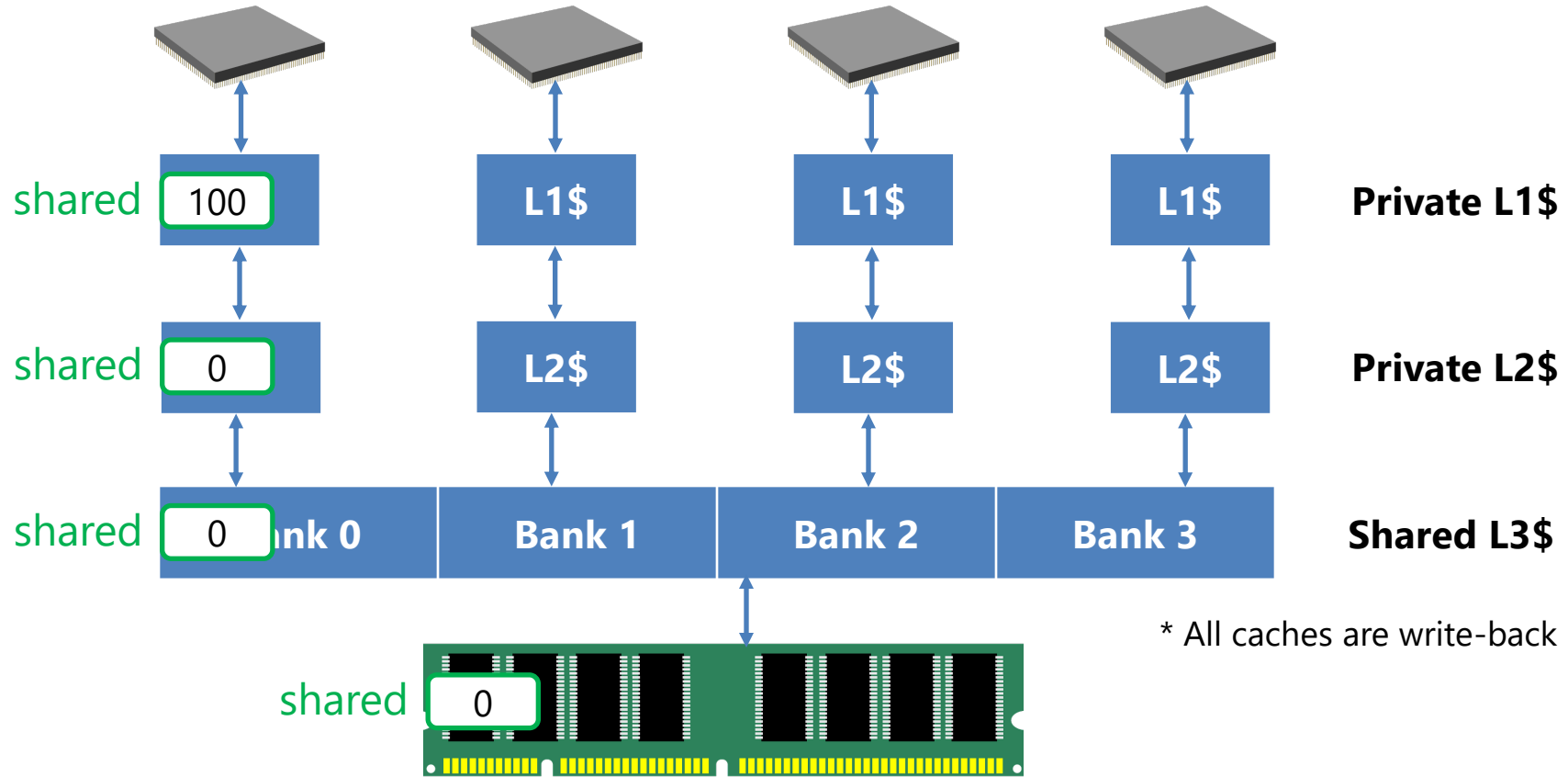
shared | 0

Caching also does replication!

- Then CPU 2 gets hold of the mutex and fetches `shared` from L3

shared `100` | **L1$** shared `0` | **L1$** **Private L1$**

shared `0` | **L2$** shared `0` | **L2$** **Private L2$**

shared `0` nk 0 | **Bank 1** | **Bank 2** | **Bank 3** **Shared L3$**

* All caches are write-back
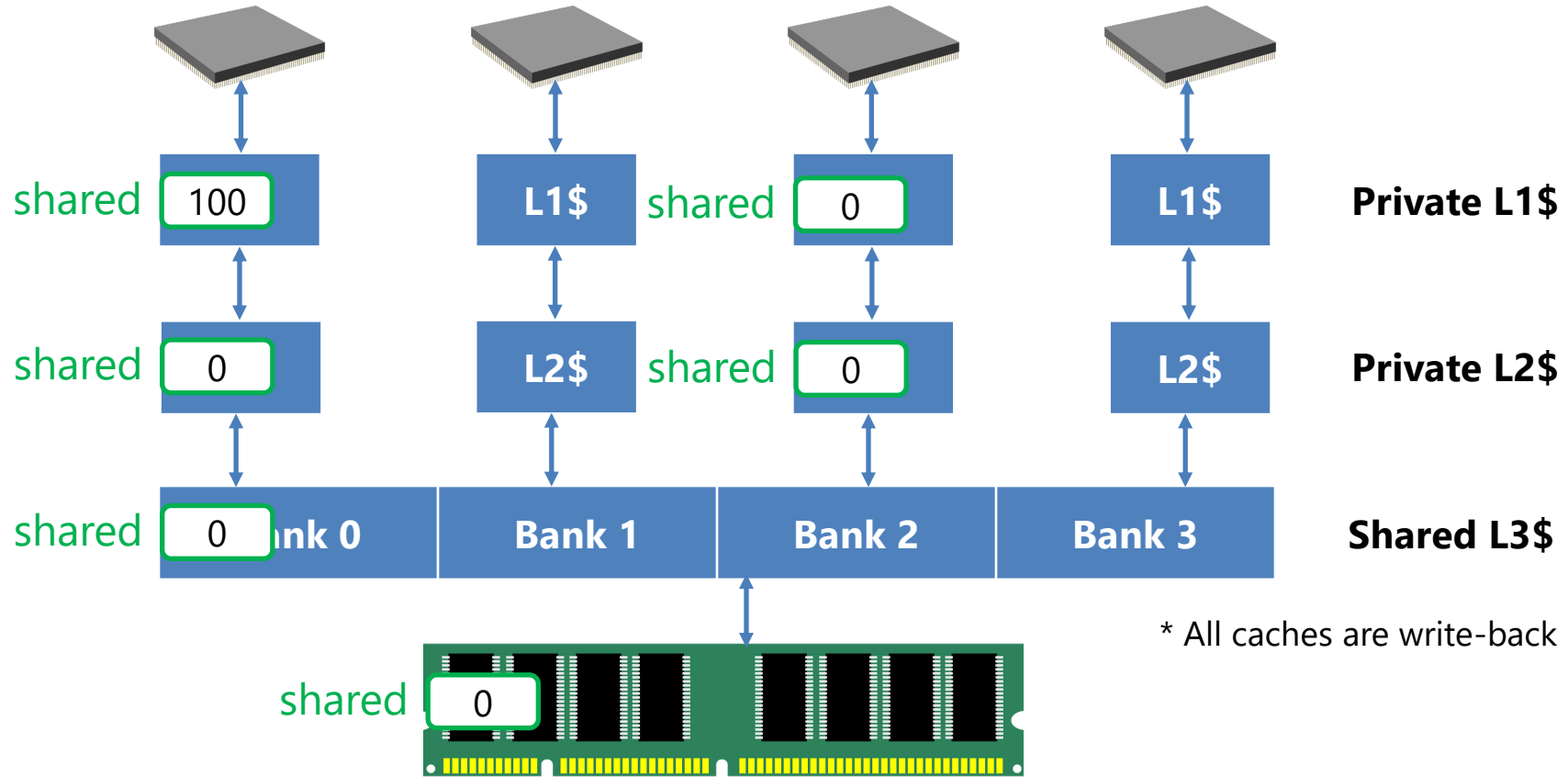
shared `0`

Caching also does replication!

- Then CPU 2 increments `shared` 10 times to 10

shared `100`

shared `0`

shared `0`

L1$**   shared `10`   **L1$ **Private L1$**

L2$**   shared `0`   **L2$ **Private L2$**

nk 0   **Bank 1**   **Bank 2**   **Bank 3**   **Shared L3$**

shared `0`

* All caches are write-back
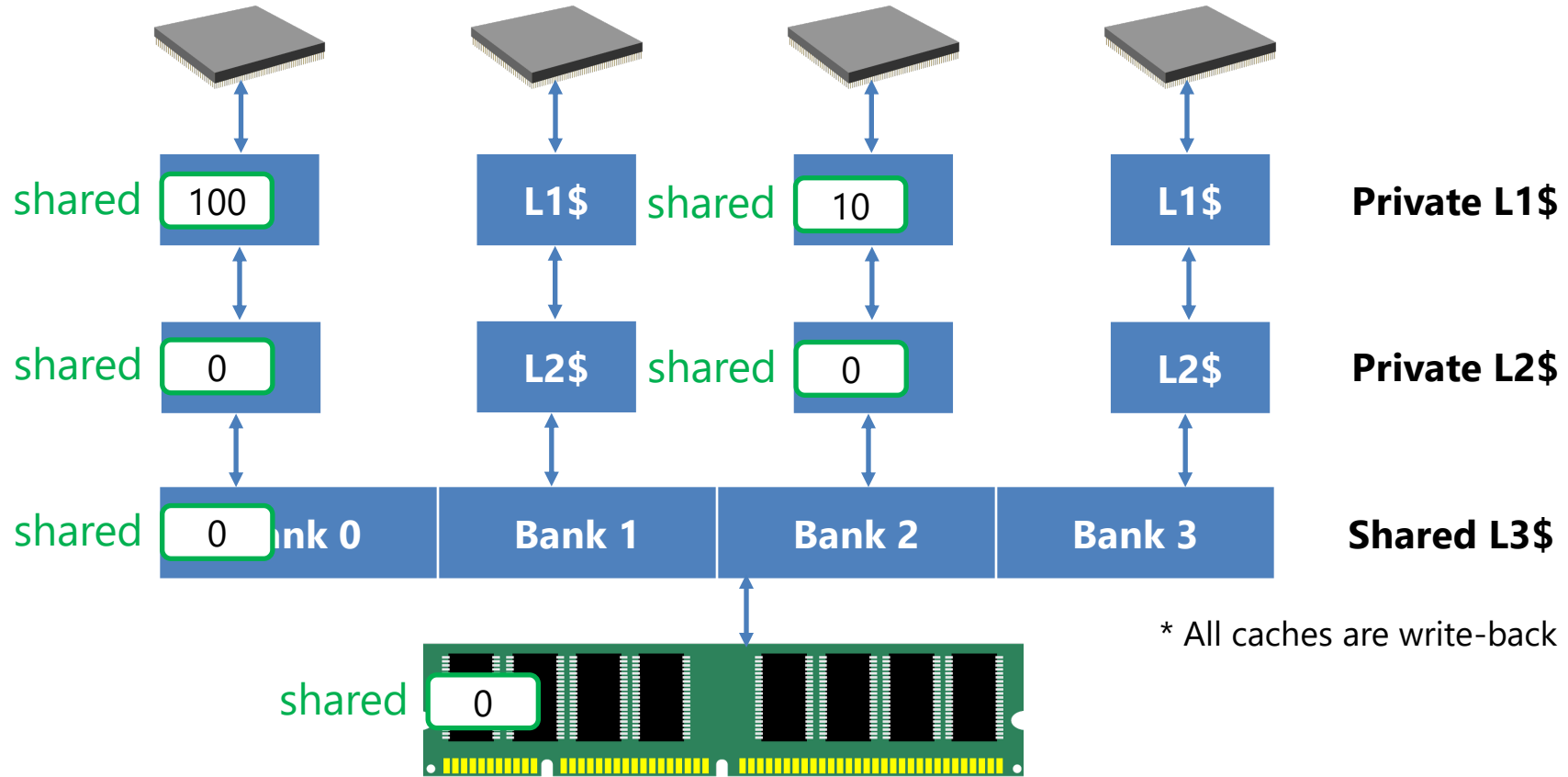
Caching also does replication!

- Clearly this is wrong. L1 caches of CPU 0 and CPU 2 are **incoherent**.

shared **100**   **L1$** shared **10**   **L1$**   **Private L1$**

shared **0**   **L2$** shared **0**   **L2$**   **Private L2$**

shared **0** nk 0   **Bank 1**   **Bank 2**   **Bank 3**   **Shared L3$**

* All caches are write-back
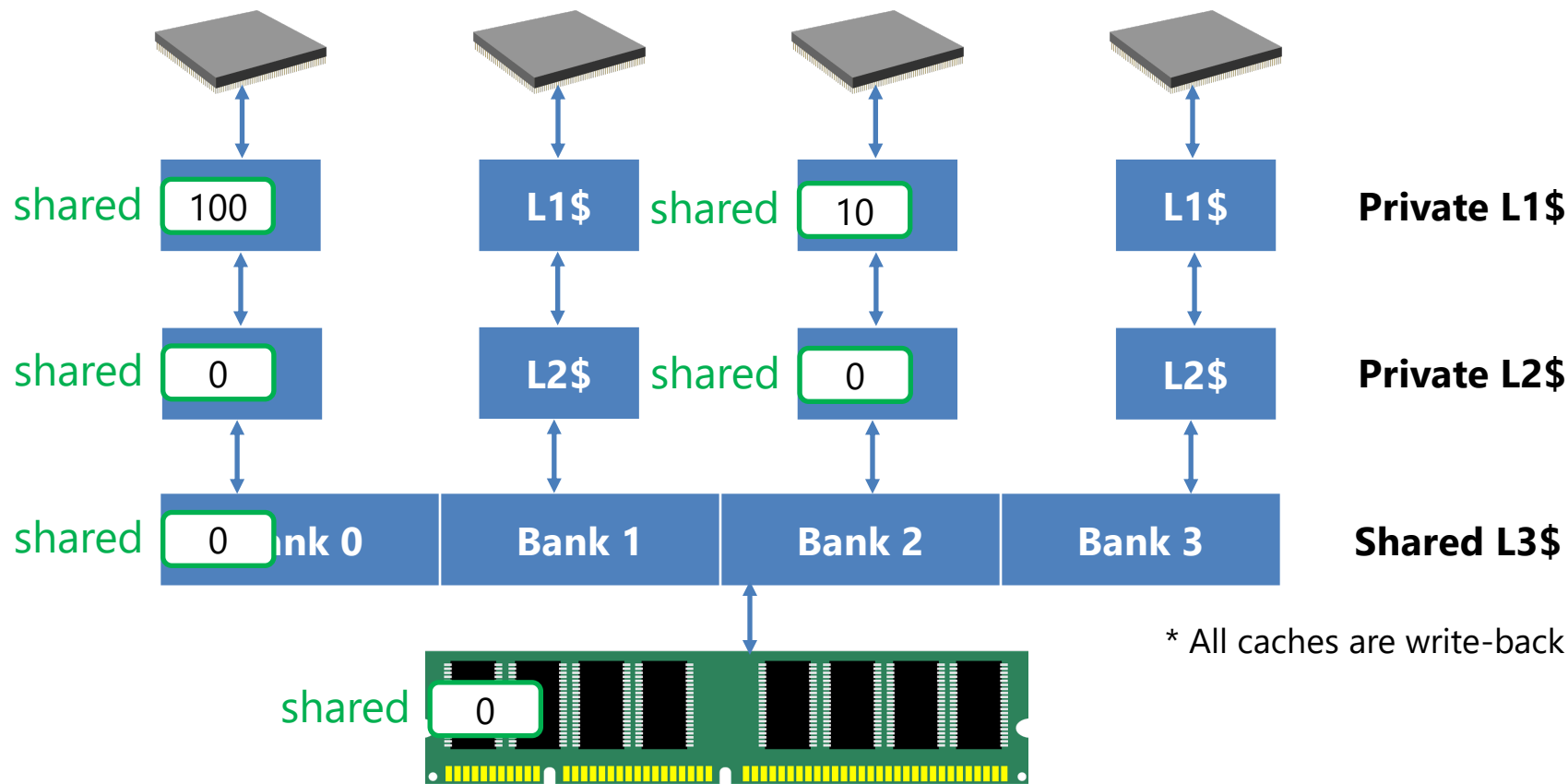
shared **0**

- This problem does not occur with a **shared cache**.
  - All processors share and work on a **single copy** of data.

| shared | 0 nk 0 | Bank 1 | Bank 2 | Bank 3 | **Shared L3$** |

- The problem exists for **private** caches.
  - Private copy is at times inconsistent with lower memory.
  - **Incoherence** occurs when private copies differ from each other.
    → Means processors return different values for same location!

Cache Coherence

Cache Coherence

- **Cache coherence** (loosely defined):
  - All processors of system should see the **same view of memory**
  - Copies of values cached by processors should adhere to this rule

- Each ISA has a different definition of what that "view" means
  - **Memory consistency model**: definition of what that "view" is

- All models agree on one thing:
  - That a change in value should reflect on all copies (eventually)

- How to guarantee changes in value are propagated to all caches?

- **Cache coherence protocol**: A protocol, or set of rules, that all caches must follow to ensure coherence between caches
  - MSI (Modified-Shared-Invalid)
  - MESI (Modified-Exclusive-Shared-Invalid)
  - … often named after the **states** in cache controller **FSM**

- Three states of **MSI** protocol (maintained for each block):
  - <u>Modified</u>: Dirty. Only this cache has copy.
  - <u>Shared</u>: Clean. Other caches may have copy.
  - <u>Invalid</u>: Block contains no data.

University of
Pittsburgh
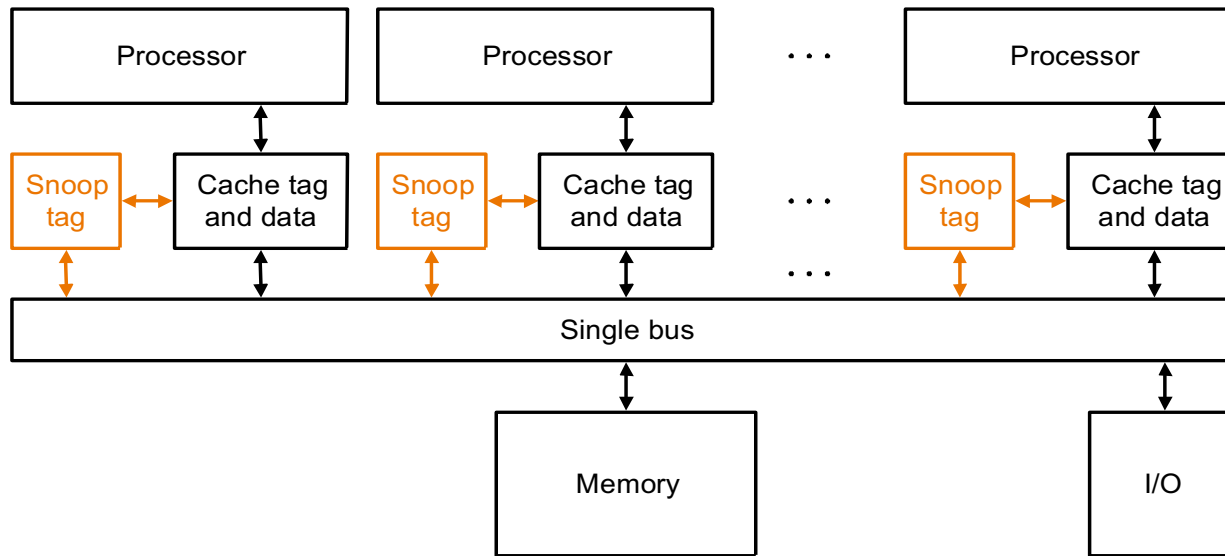
- Each processor **monitors (snoops)** the activity on the **bus**
  - In much the same way as how nodes snoop the Ethernet
- Cache state changes in response to both:
  - Read / writes from the **local processor**
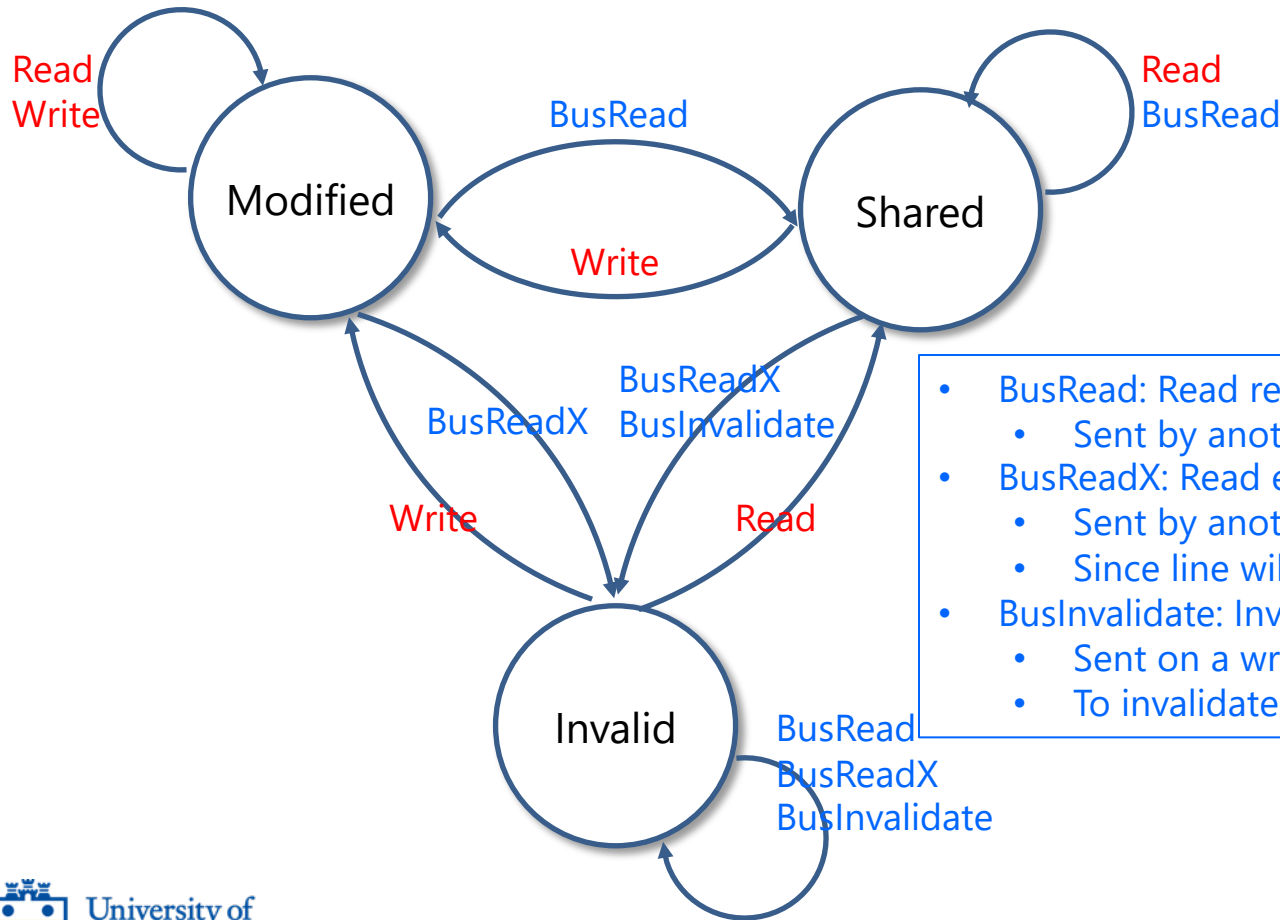 - Read misses / write misses from **remote processors** it snoops

MSI: Example

- All bus activity is show in blue. Cache changes block state in response.
- Bus activity is generated only for cache misses, or for invalidates
- Other caches must maintain coherence by monitoring that bus activity

Event	In P1's cache	In P2's cache
	L = invalid	L = invalid
P1 writes 10 to A (write miss)	L ← A = 10 (modified)	Read Exclusive A (from write in P1) L = invalid
P1 reads A (read hit)	L ← A = 10 (modified)	L = invalid
P2 reads A (read miss)	Read A (from read in P2) L ← A = 10 (shared)	L ← A = 10 (shared)
P2 writes 20 to A (write hit)	Invalidate A (from write in P2) L = invalid	L ← A = 20 (modified)
P2 writes 40 to A (write hit)	L = invalid	L ← A = 40 (modified)
P1 write 50 to A (write miss)	L ← A = 50 (modified)	Read Exclusive A (from write in P1) L = invalid

University of Pittsburgh

Cache Controller FSM for MSI Protocol

- Processor activity in red, Bus activity in blue

Modified

Read
Write

BusRead

Write

Shared

Read
BusRead

BusReadX
BusInvalidate

BusReadX

Write

Read

Invalid

BusRead
BusReadX
BusInvalidate

- BusRead: Read request is snooped
  - Sent by another processor on read miss
- BusReadX: Read exclusive request is snooped
  - Sent by another processor on a write miss
  - Since line will be modified, need to invalidate
- BusInvalidate: Invalidate request is snooped
  - Sent on a write hit on shared cache line
  - To invalidate all other shared lines in system

University of Pittsburgh

TLB Coherence

- We said TLBs are also a type of cache that caches PTEs.
  - So what happens if a processor changes a PTE?
  - How does that change get propagated to other processor TLBs?

- Unfortunately, there is no hardware coherence for TLBs. ☹

- That means software (the OS) must handle the coherence
  - Which is of course much much slower

TLB shootdown

- In order to update a PTE (page table entry)
  - Initiator OS must first flush its own TLB
  - Send IPIs (Inter-processor interrupts) to other processors
    - To flush the TLBs for all other processors too
  - Source of significant performance overhead

TLB Flushes in Linux and FreeBSD
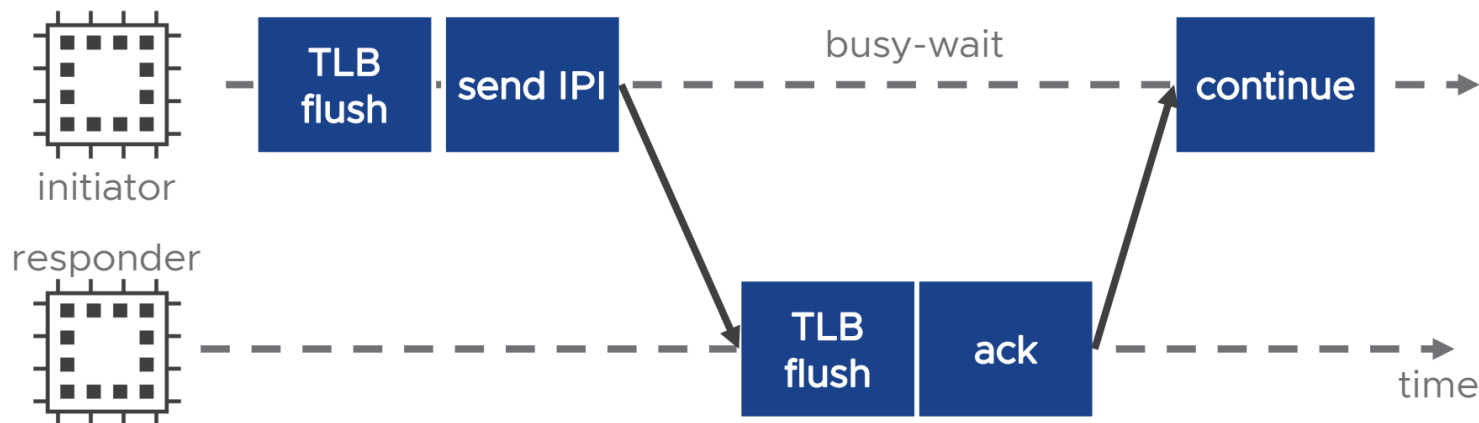
* Courtesy of Nadav Amit et al. at VMWare