Multiprocessors and Caching

CS/COE 1541 (Fall 2020)

Wonsun Ahn

University of Pittsburgh

Two ways to use multiple processors

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

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

  → Just like Ethernet, caches are part of a larger network of caches

Shared Data Review

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

- Dataraces!
  - You learned it in CS/COE 449. Yes, you did.

Review: Datarace Example

```
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;
}
```

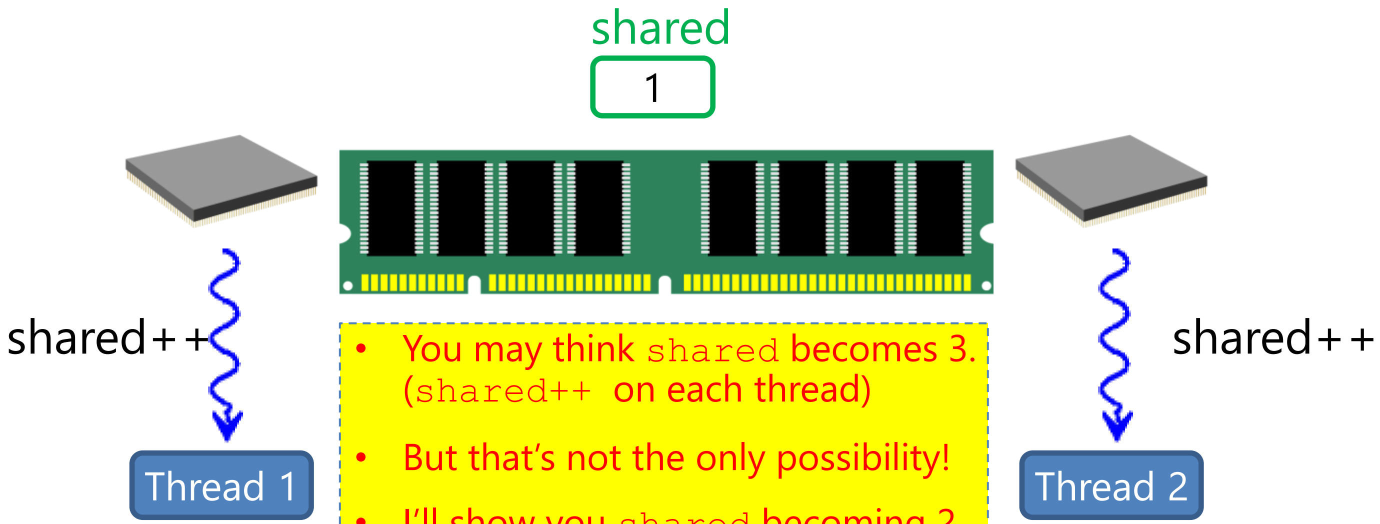
```
bash-4.2$ ./datarace
shared=1085894
bash-4.2$ ./datarace
shared=1101173
bash-4.2$ ./datarace
shared=1065494
```

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

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

Review: Datarace Example

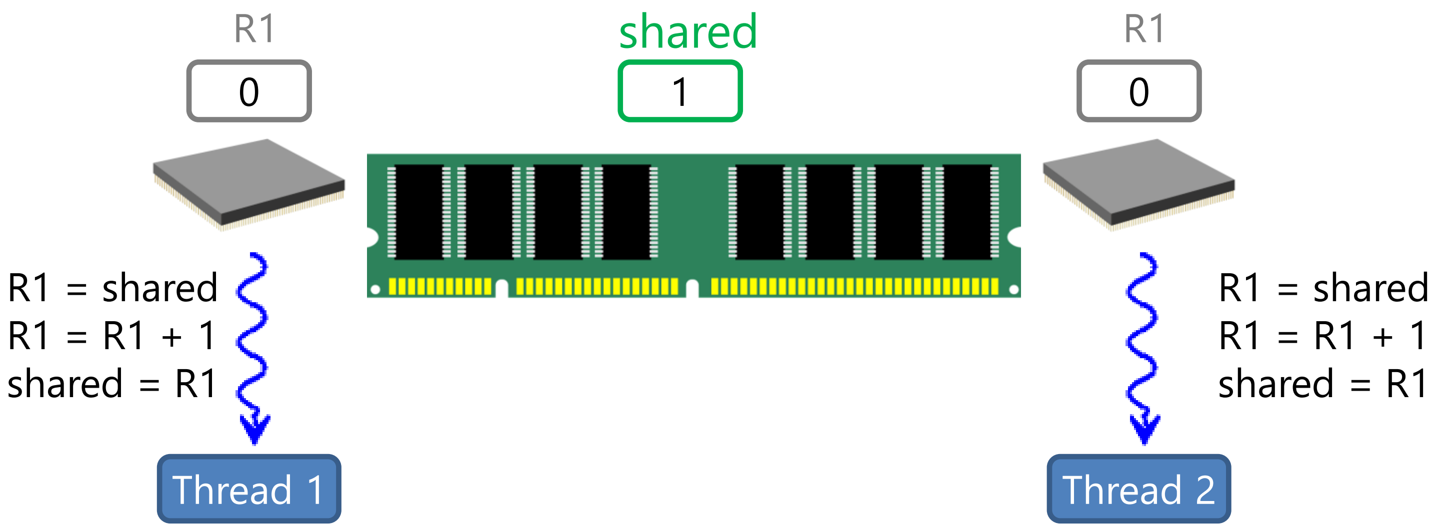
- When two threads do `shared++;` initially `shared` = 1



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

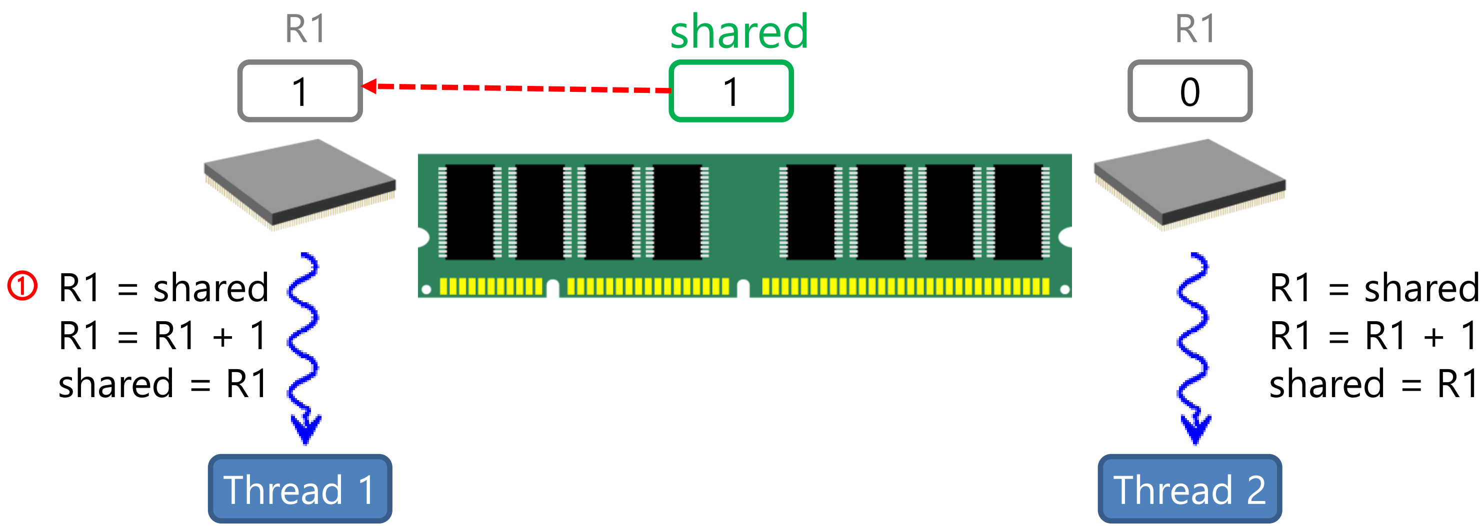
Review: Datarace Example

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



Review: Datarace Example

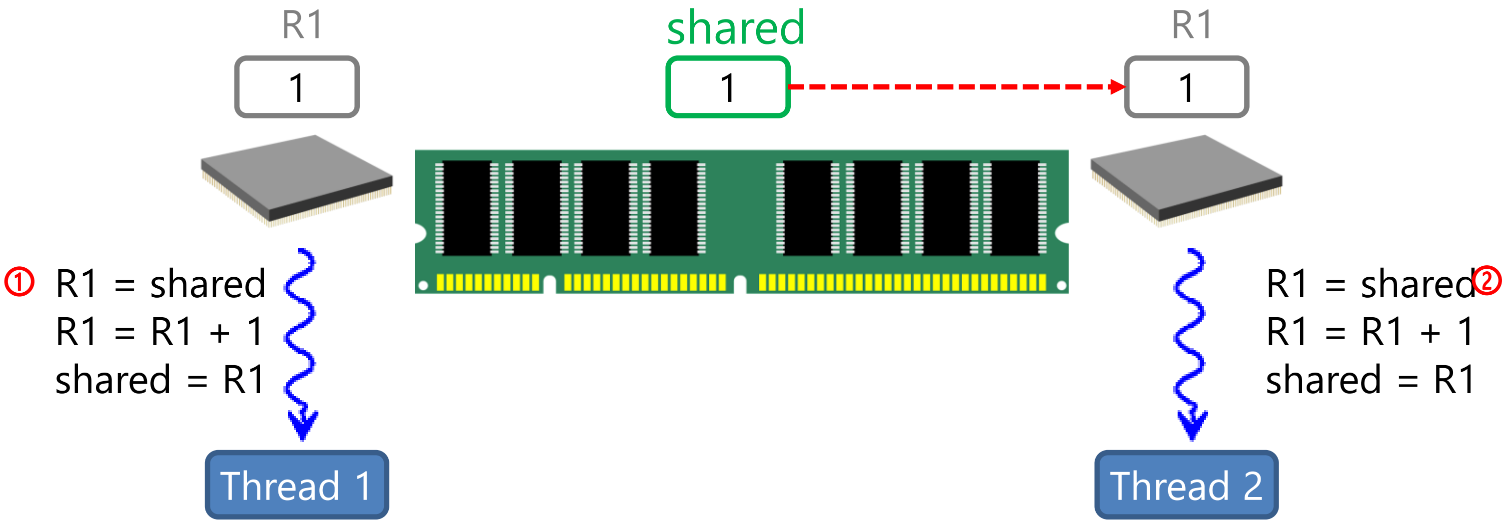
- When two threads do `shared++;` initially `shared` = 1



University of Pittsburgh

Review: Datarace Example

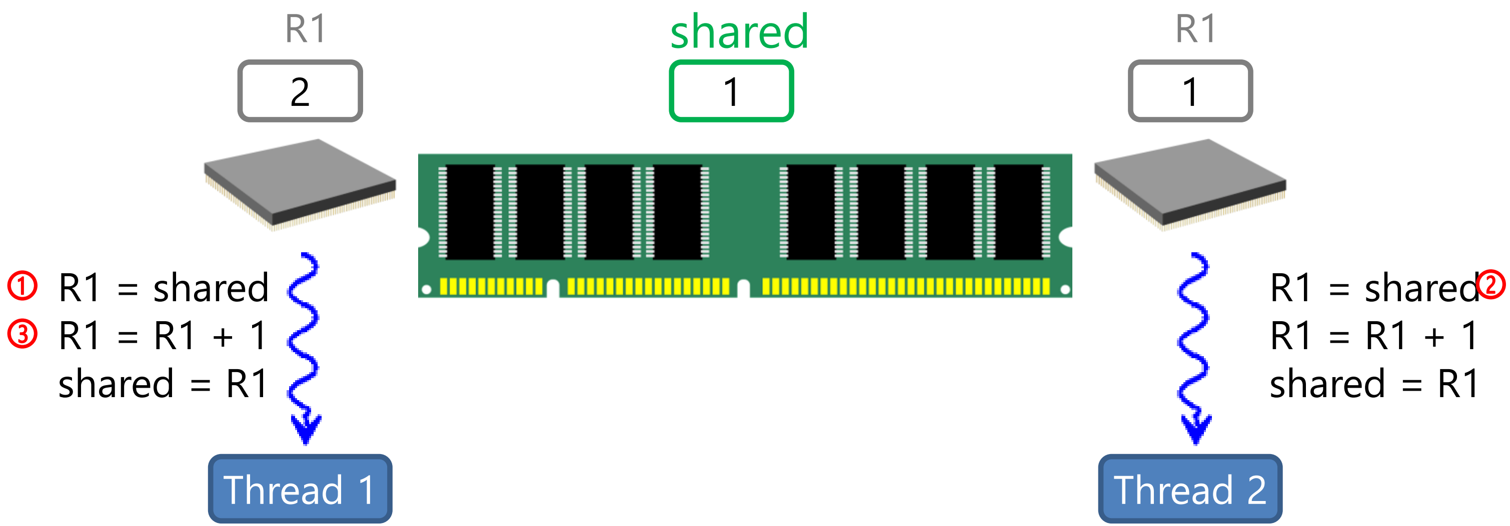
- When two threads do `shared++;` initially `shared` = 1



University of Pittsburgh

Review: Datarace Example

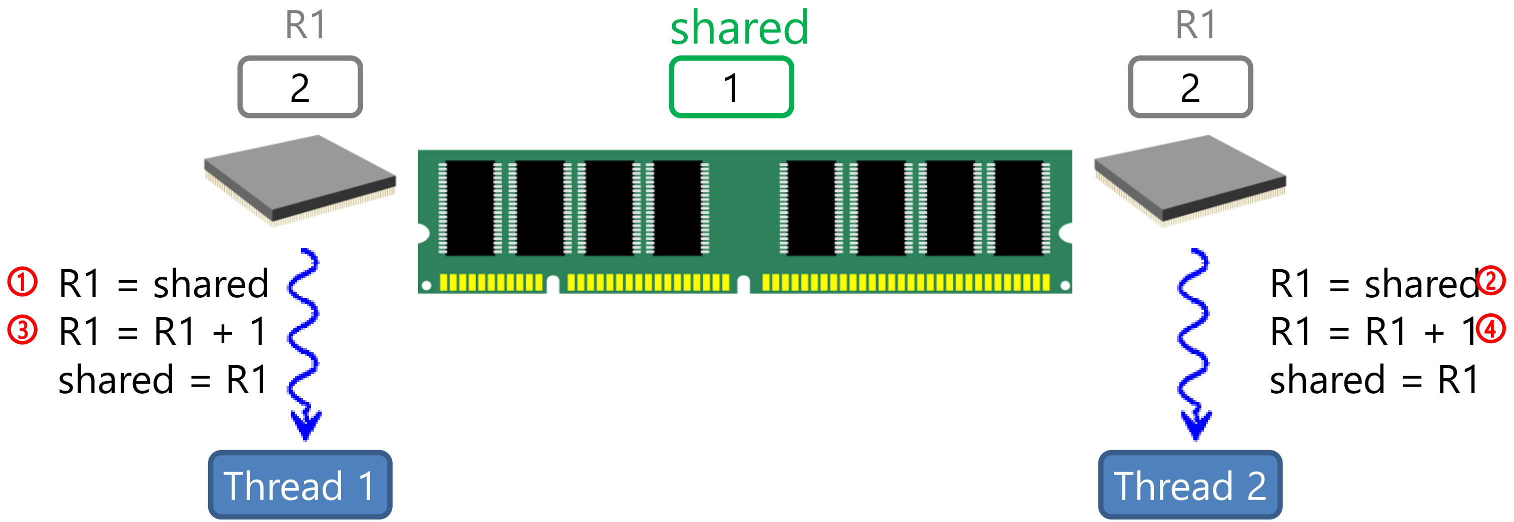
- When two threads do `shared++;` initially `shared` = 1



University of Pittsburgh

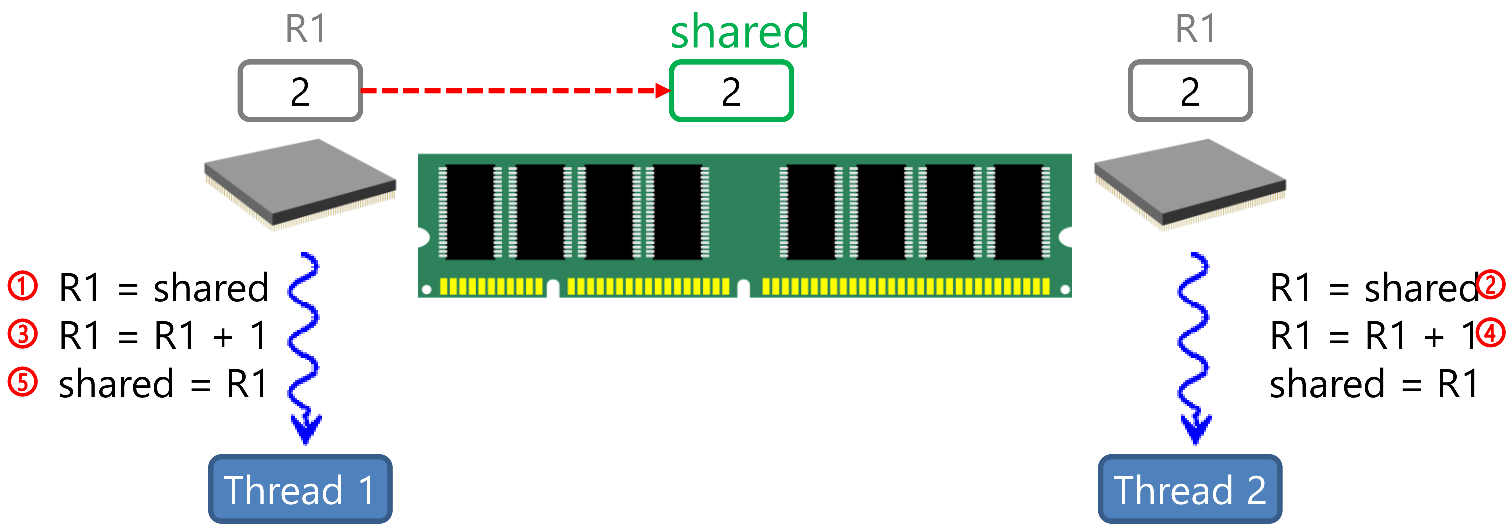
Review: Datarace Example

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



Review: Datarace Example

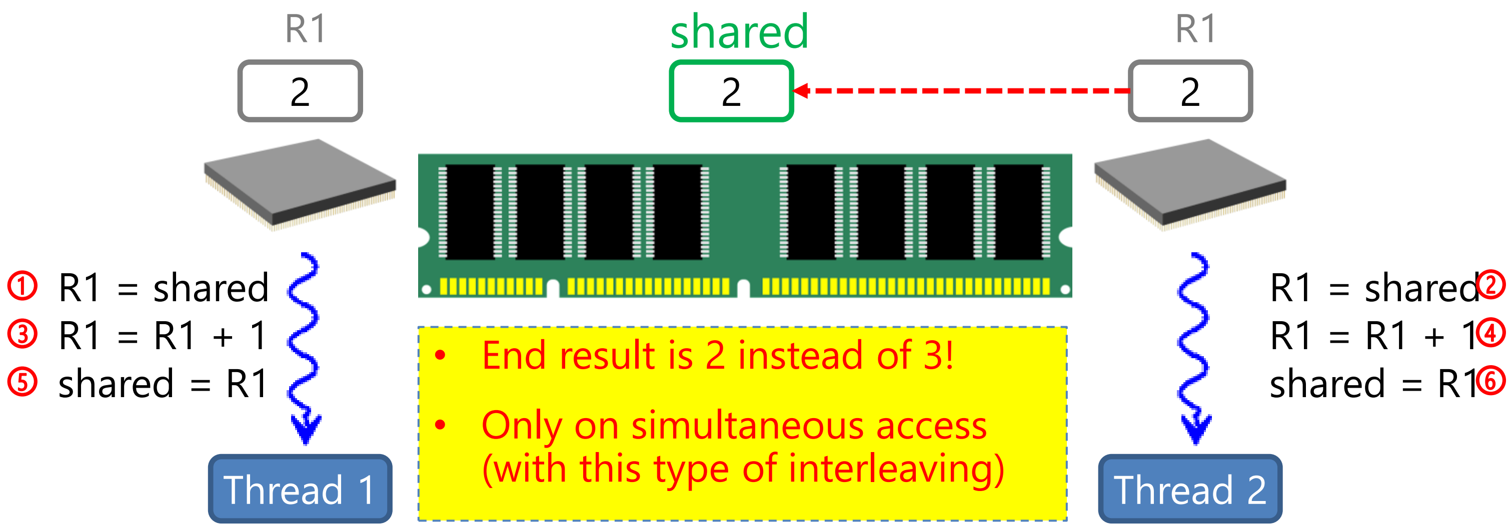
- When two threads do `shared++;` initially `shared` = 1



Review: Datarace Example

- 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`



- End result is 2 instead of 3!
- Only on simultaneous access (with this type of interleaving)

Review: Datarace Example

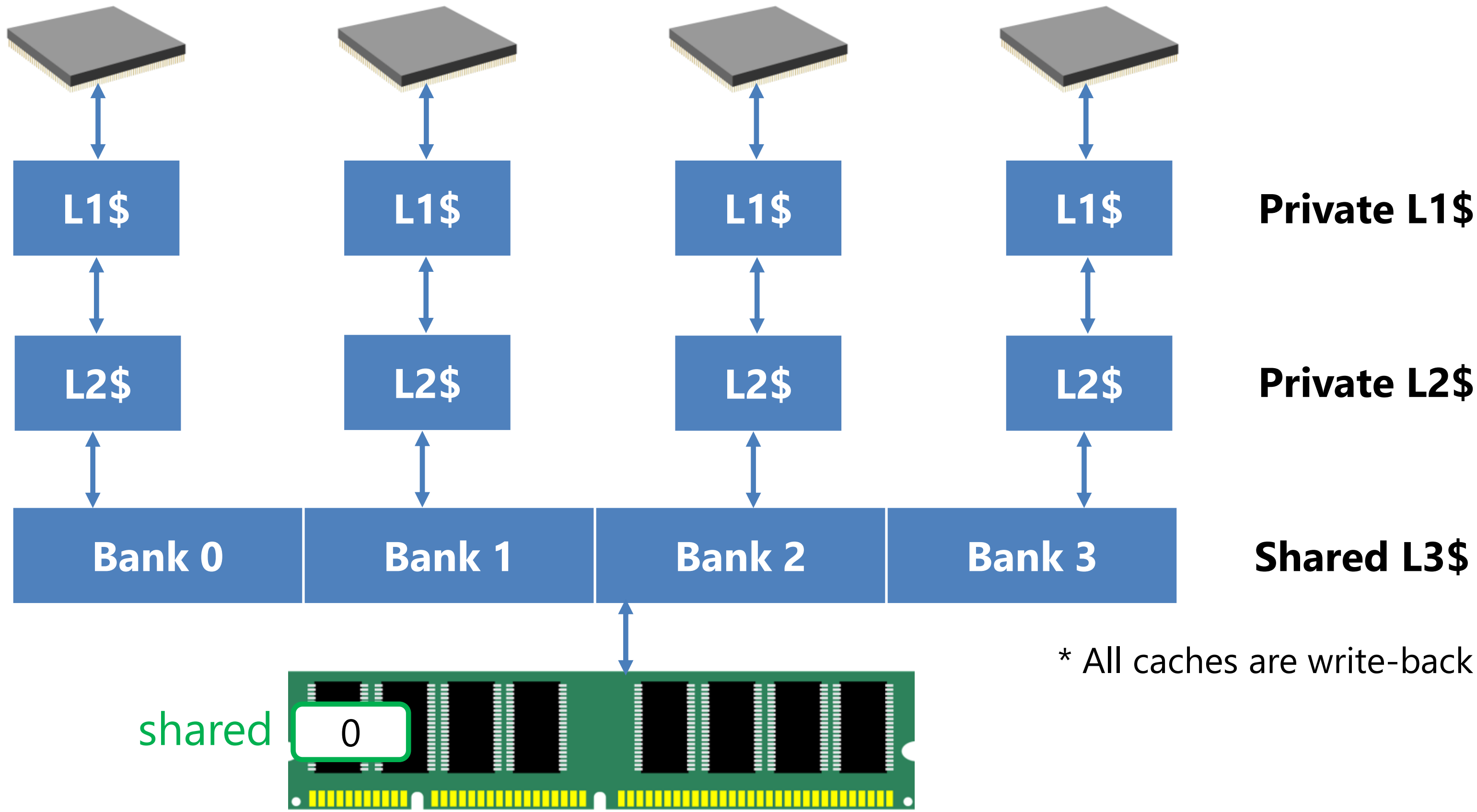
```
pthread_mutex_t lock;
int shared = 0;
void *add(void *unused) {
  for(int i=0; i < 1000000; i++) {
      pthread_mutex_lock(&lock);
      shared++;
      pthread_mutex_unlock(&lock);
  }
  return NULL;
}
int main() {
  …
}
```

```
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!

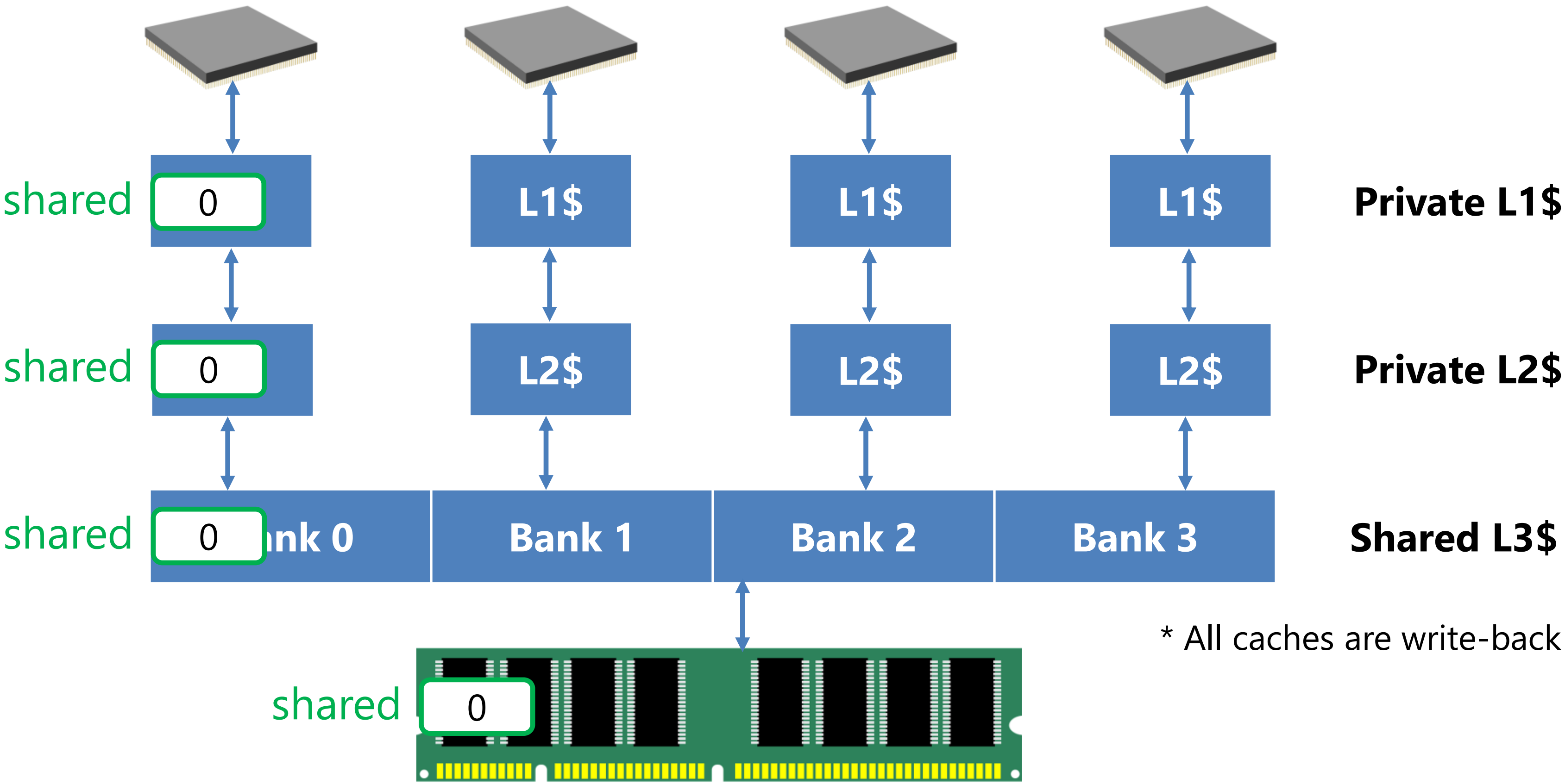
Caching also does replication!

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



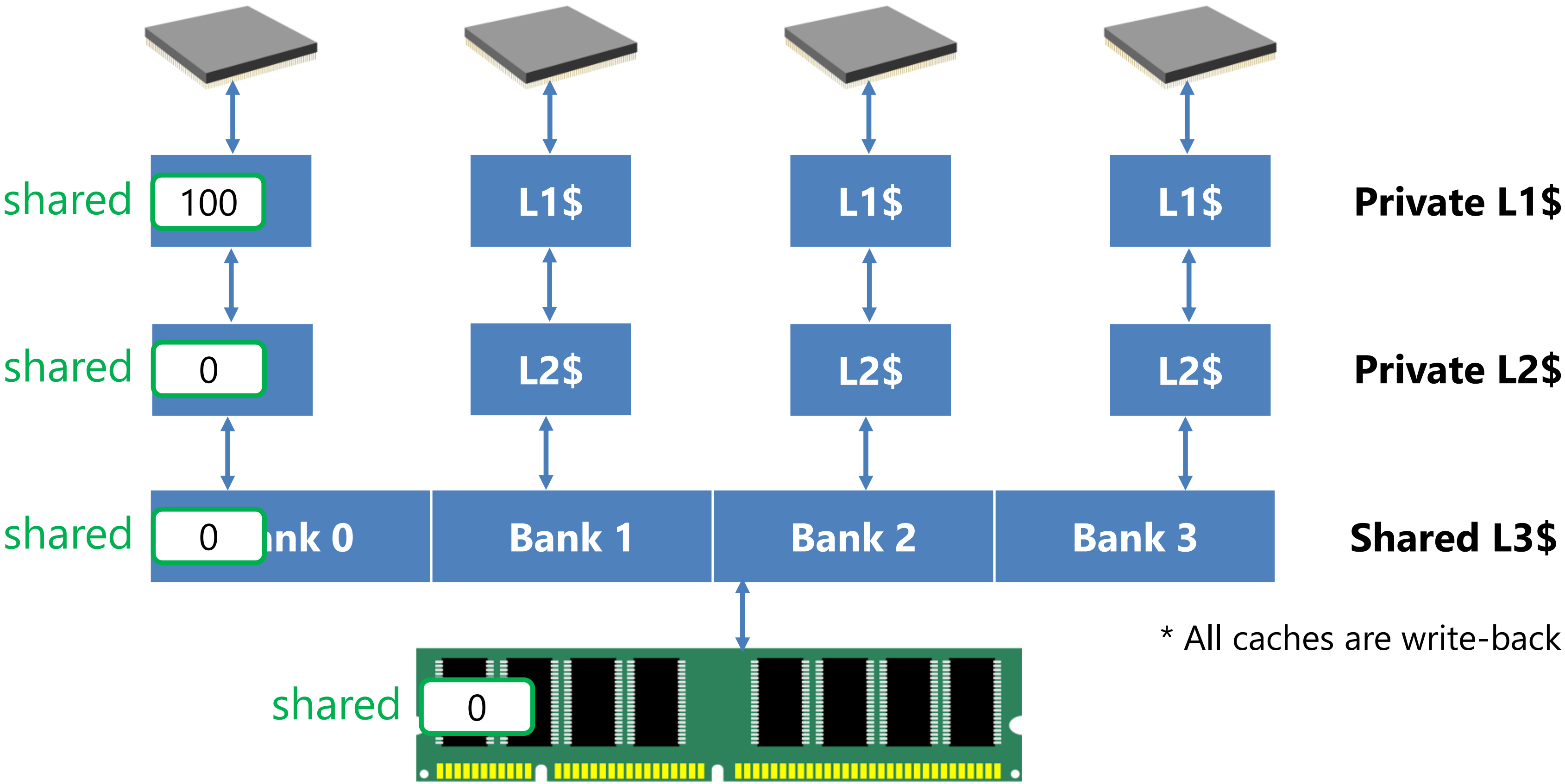
Caching also does replication!

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



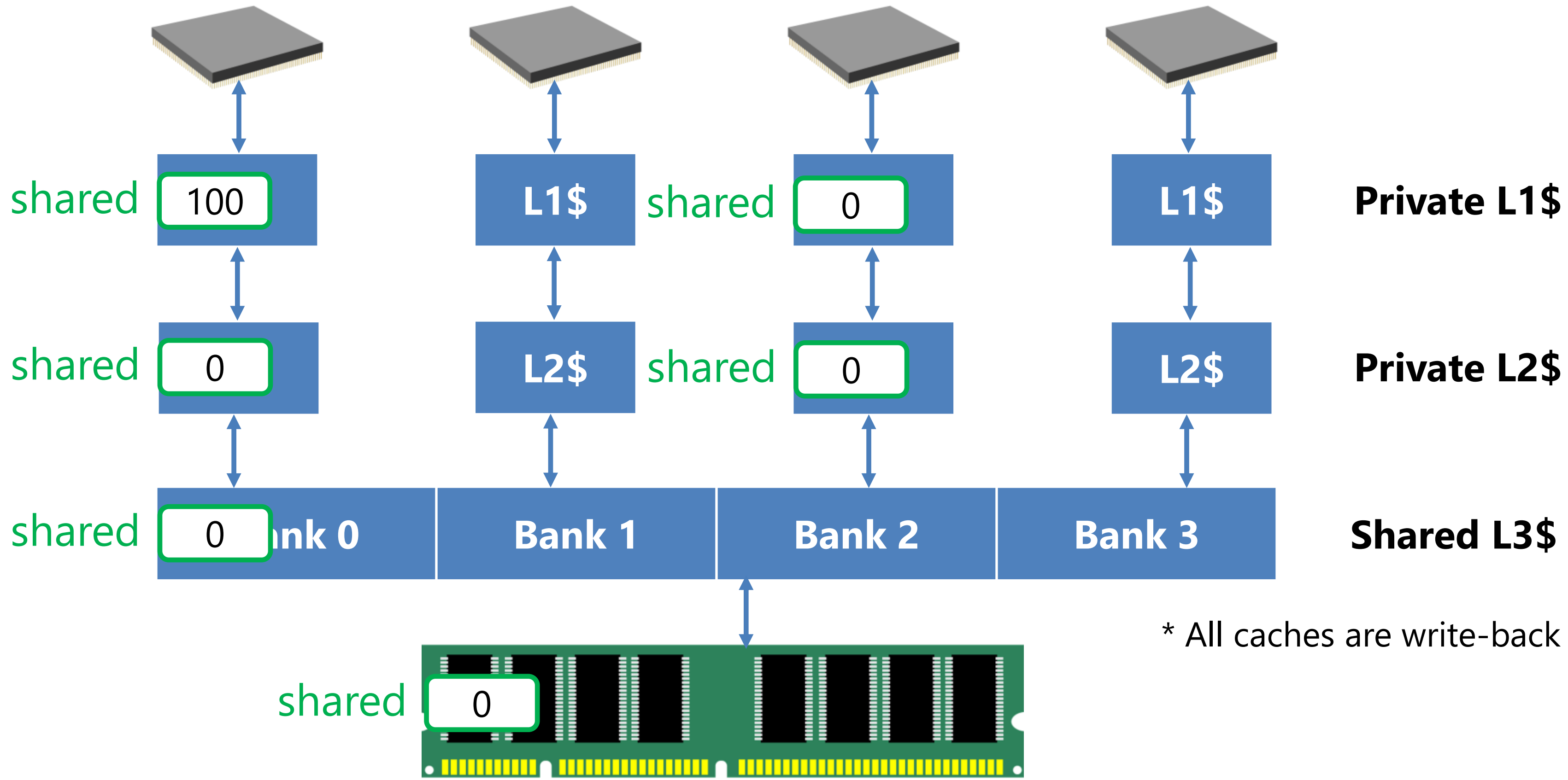
Caching also does replication!

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



Caching also does replication!

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

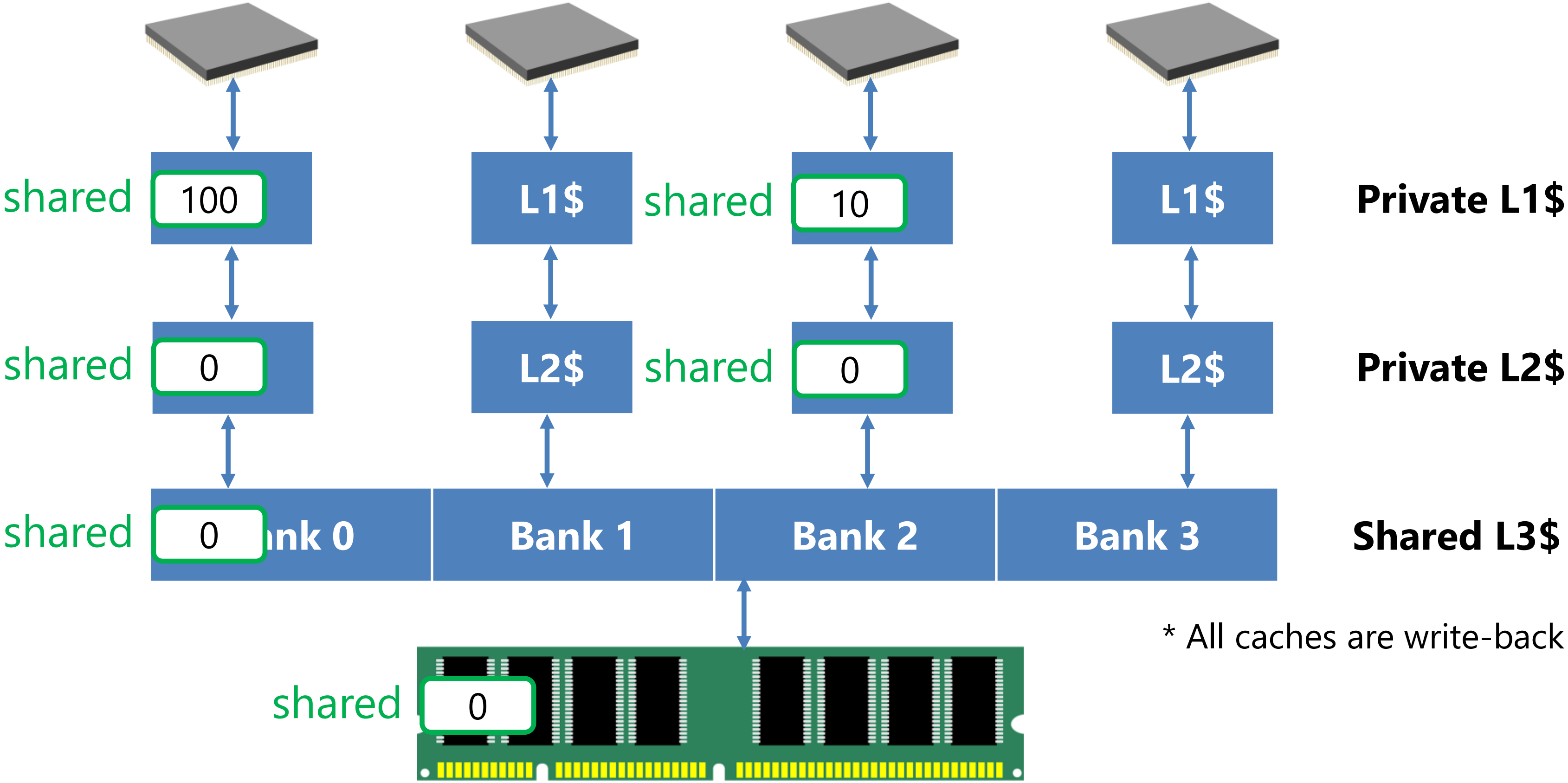


* All caches are write-back

University of Pittsburgh

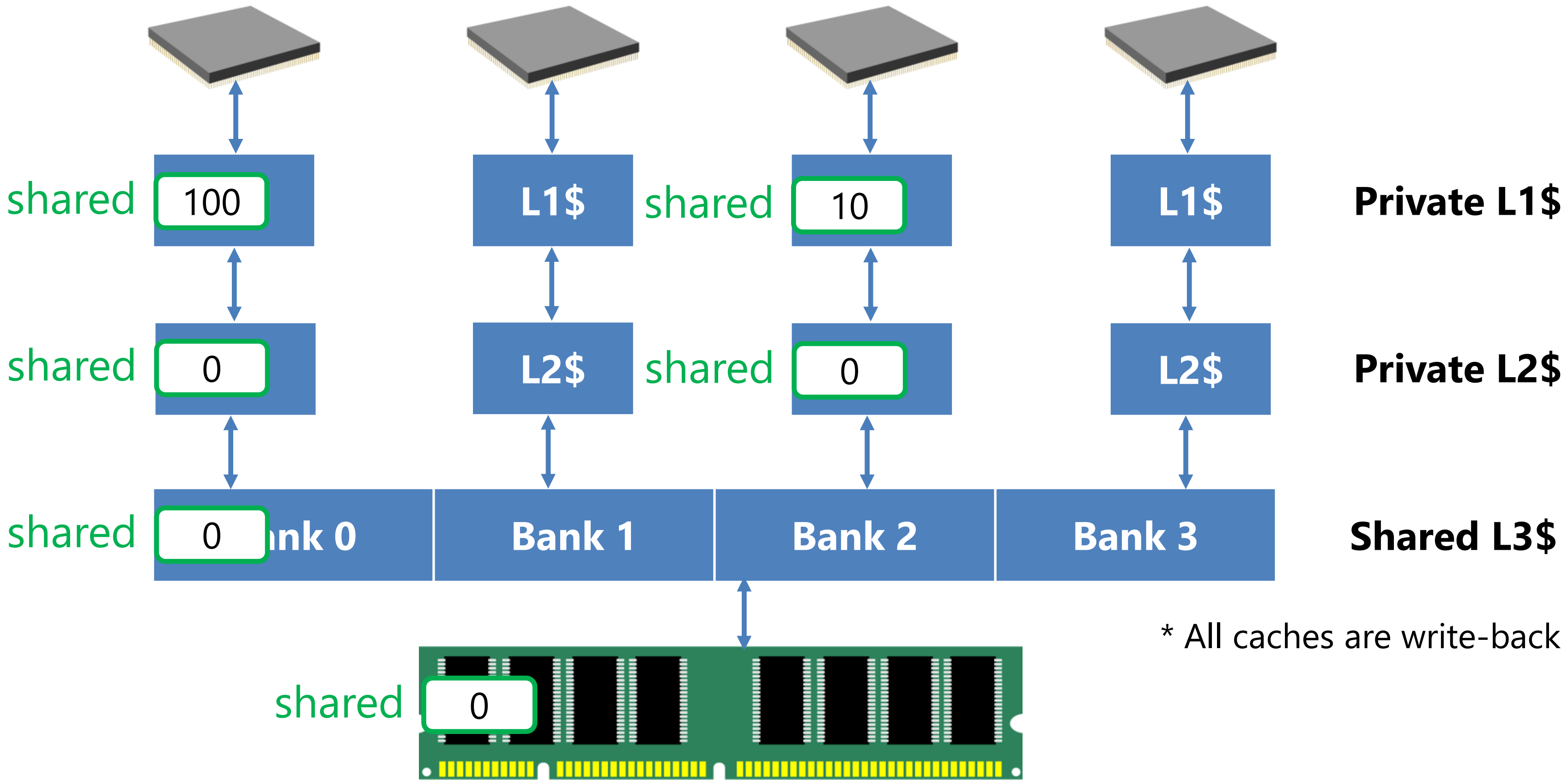
Caching also does replication!

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



Caching also does replication!

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



Cache Incoherence: Problem with Private Caches

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

  - The problem exists only with private caches.

- 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!

University of Pittsburgh

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
  - That’s a discussion for another day…

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

University of Pittsburgh

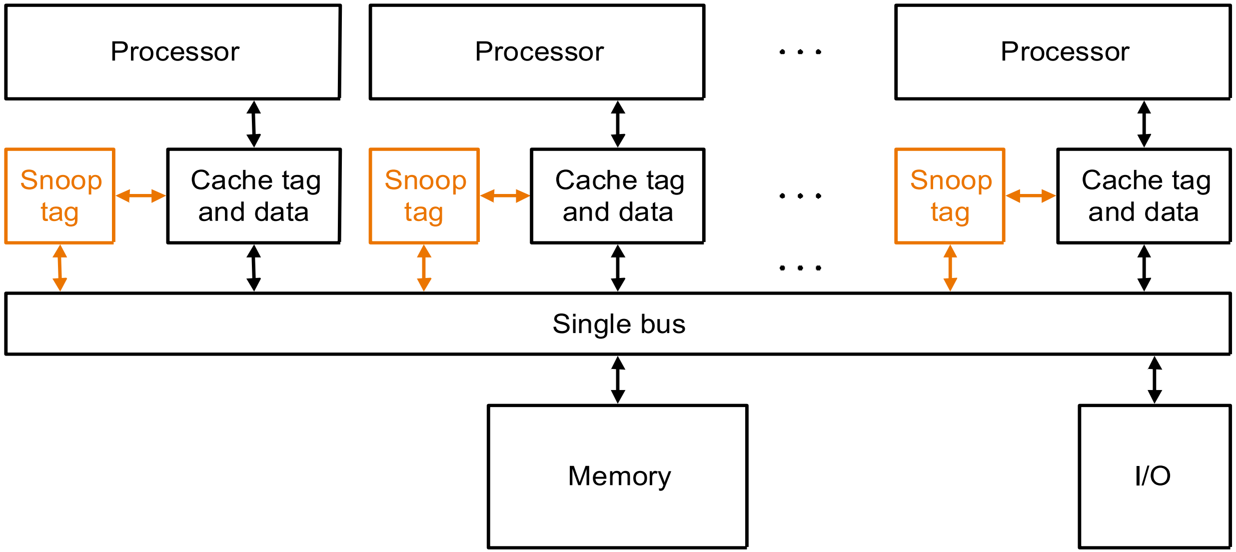
Implementing Cache Coherence

- 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):
  - Modified: Dirty. Only this cache has copy.
  - Shared: Clean. Other caches may have copy.
  - Invalid: Block contains no data.

MSI Snoopy Cache Coherence Protocol



- 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

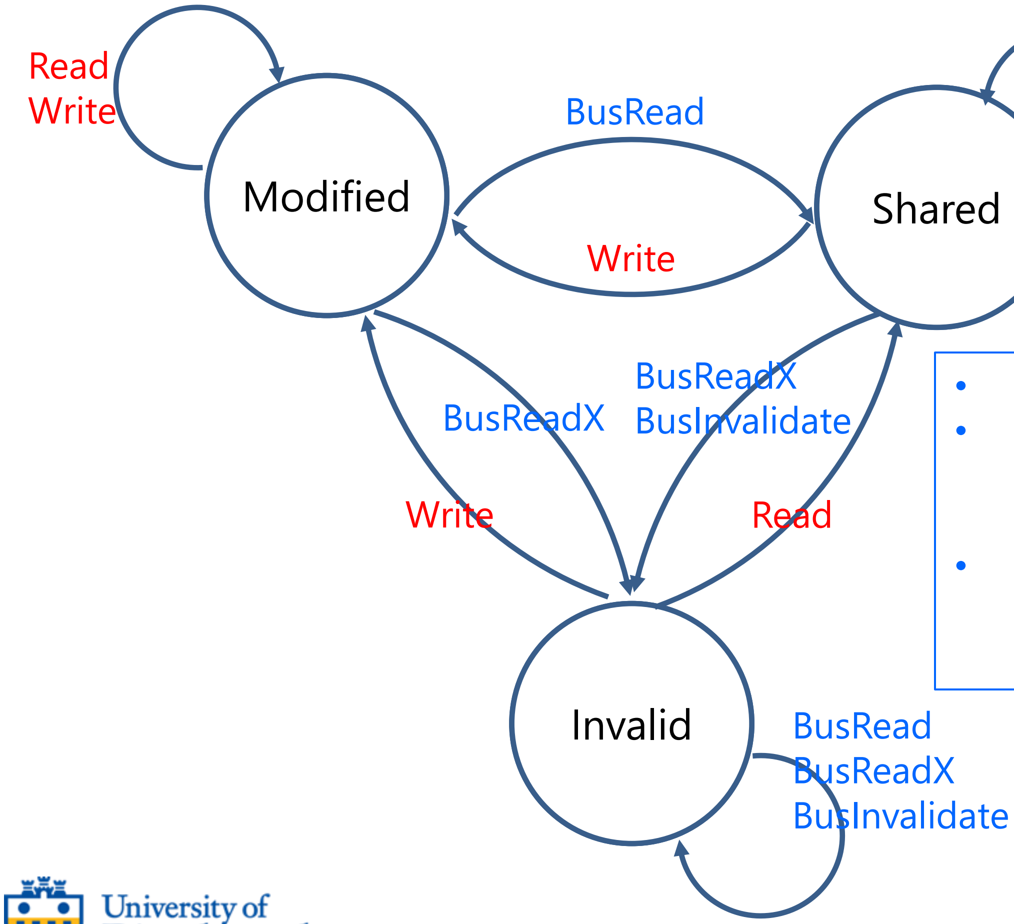
MSI: Example

- 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

Cache Controller FSM for MSI Protocol

- Processor activity in red, Bus activity in blue



- BusRead: Read request is snooped
- BusReadX: Read exclusive request is snooped
  - Sent by a 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

University of Pittsburgh

How about TLBs?

- 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?

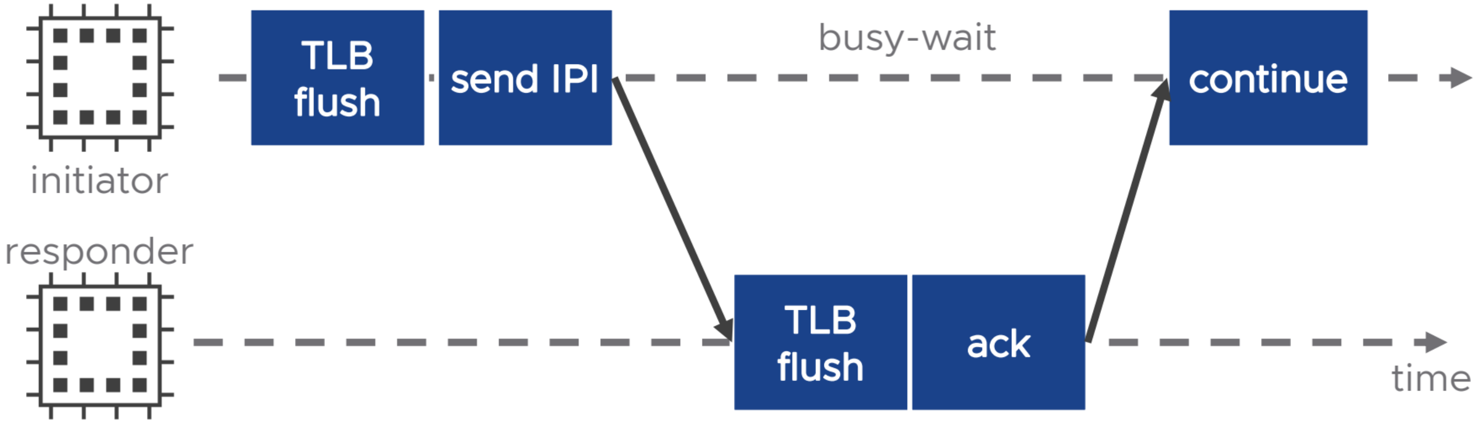
- Unfortunately, there is no hardware coherence for TLBs. ☹

- 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