# Multiprocessors and Caching

CS/COE 1541 (Fall 2020)

Wonsun Ahn

# Two ways to use multiple processors

- **Distributed (Memory) System**
  - Processors **do not share memory** (and by extension data)
  - Processors work on own data and can't see other processors' data
  - Processors exchange data using network messages
  - Programming Standards:
    - Message Passing Interface (MPI) – APIs for exchanging messages
    - JSON / XML – Standards for encoding data into messages

- **Shared Memory System**
  - Processors **share memory** (and by extension data)
  - Usually what's meant by a "multiprocessor system"
  - Programming Standards:
    - Pthreads – C/C++ APIs for threading and synchronization
    - Java threads – Java APIs for threading and synchronization
    - OpenMP – Set of compiler #pragma directives for parallelization

→ This is what we will talk about in computer architecture

# 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

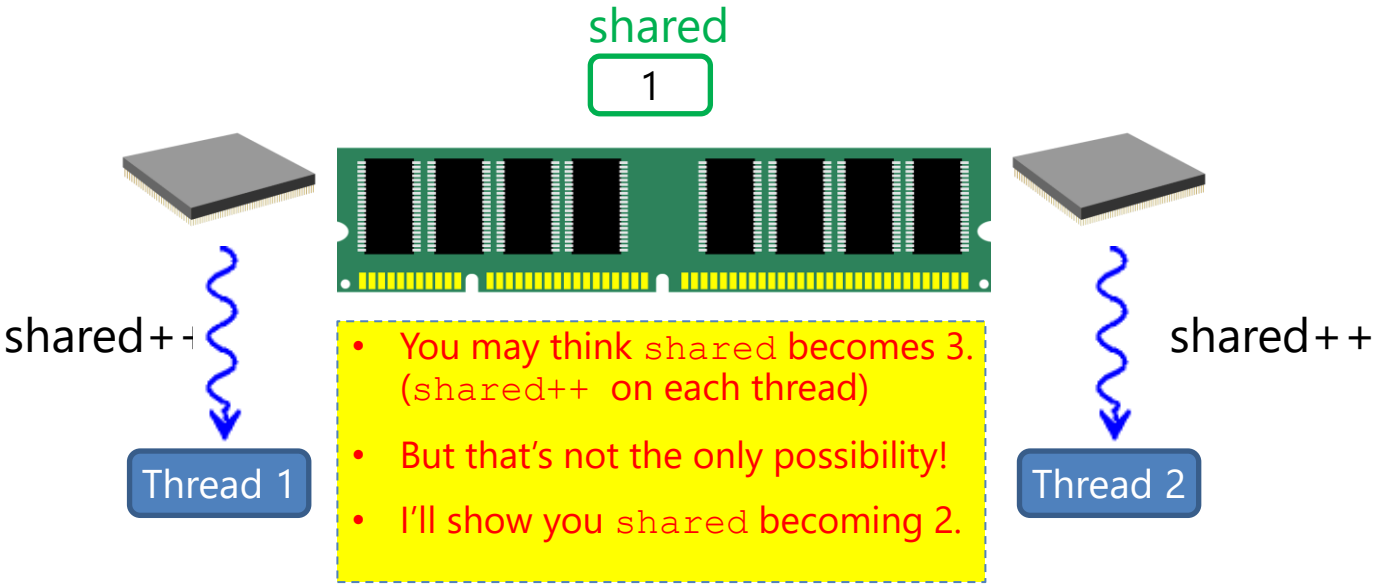
```
int shared = 0;
void *add(void *unused) {
  for(int i=0; i < 1000000; i++) { shared++; }
  return NULL;
}
int main() {
  pthread_t t;
  // Child thread starts running add
  pthread_create(&t, NULL, add, NULL);
  // Main thread starts running add
  add(NULL);
  pthread_join(t, NULL);
  printf("shared=%d\n",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

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

- What do you expect from running this?
- Maybe shared=2000000 ?
- Due to datarace on `shared`.

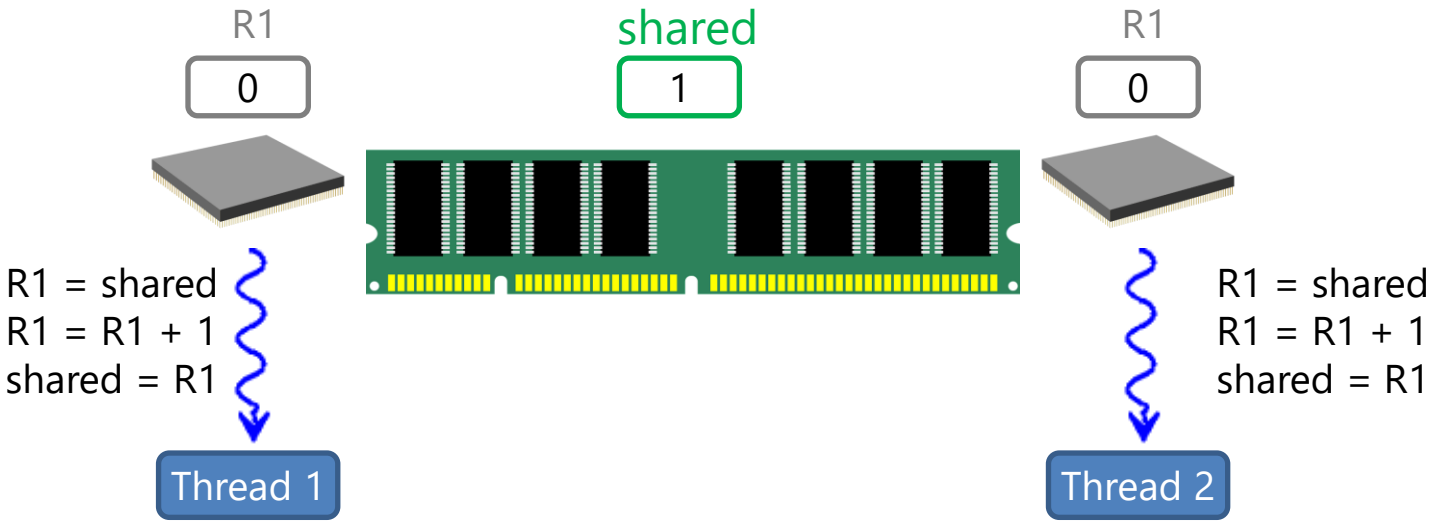
# Review: Datarace Example

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



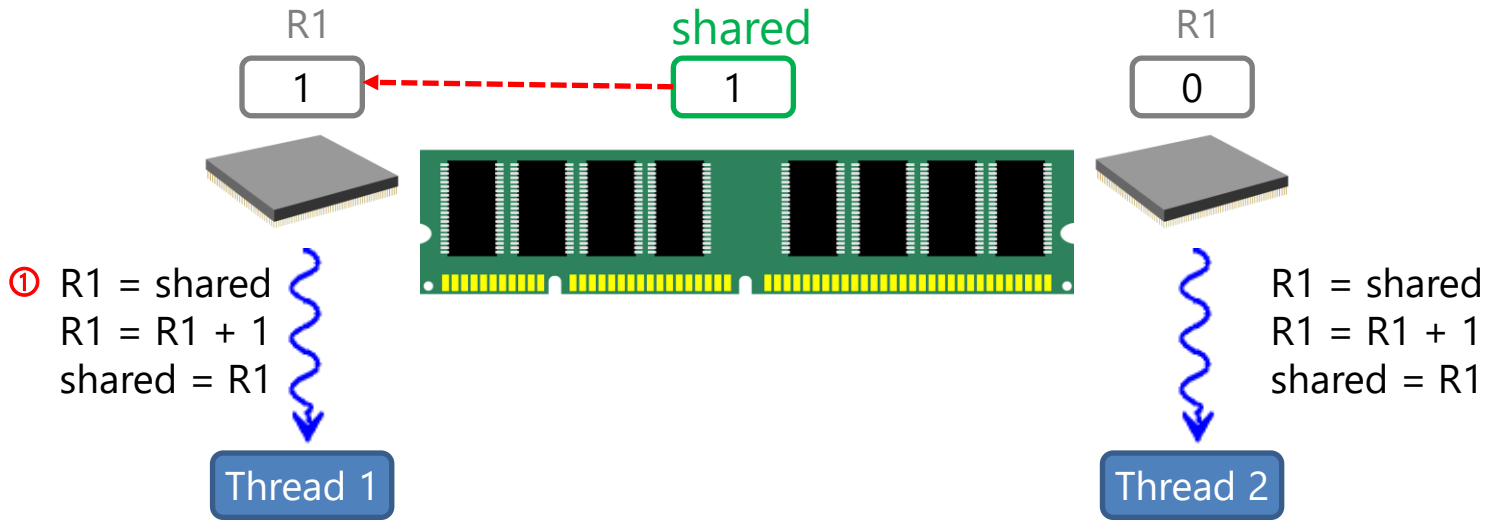
# Review: Datarace Example

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



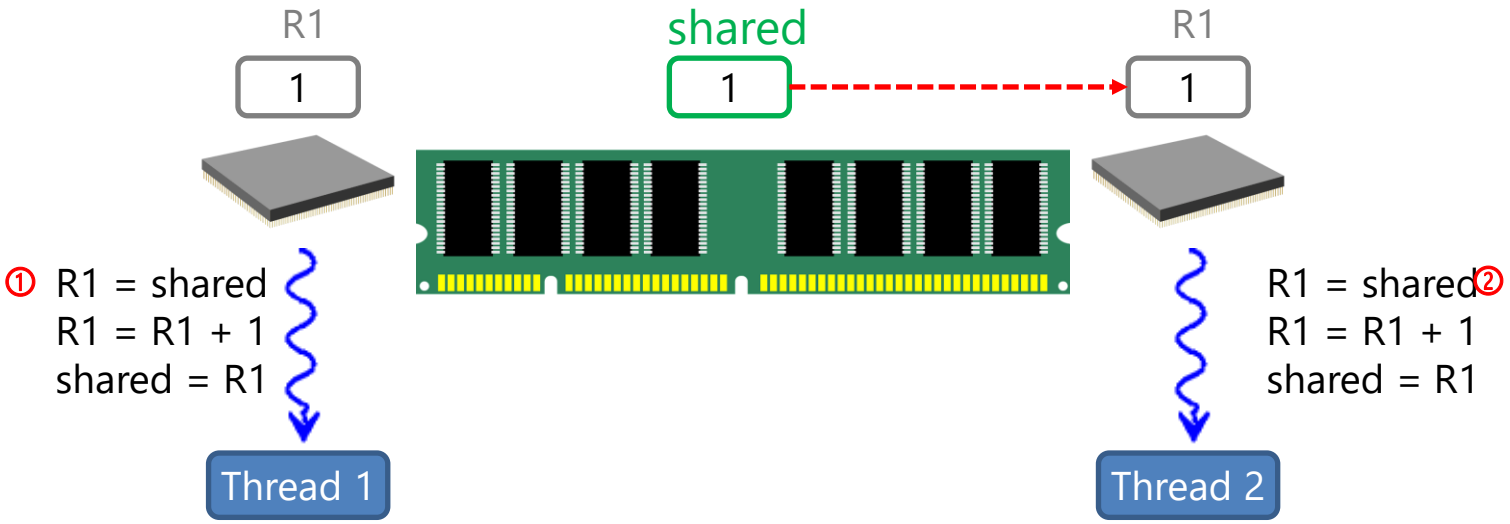
# Review: Datarace Example

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



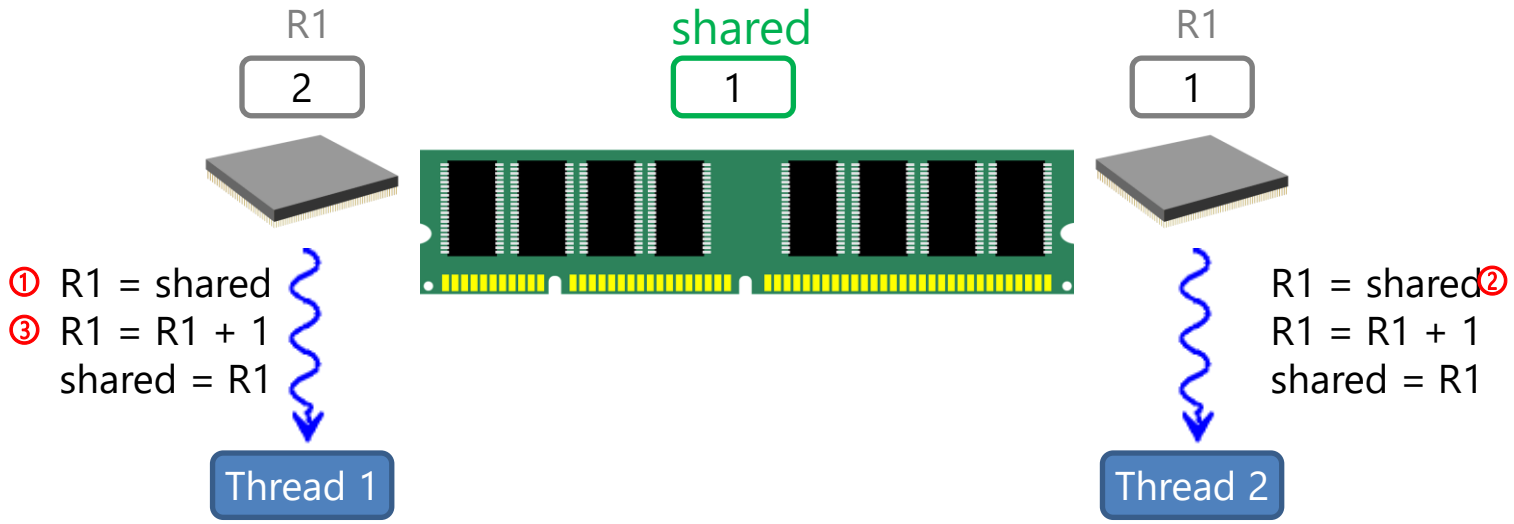
# Review: Datarace Example

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



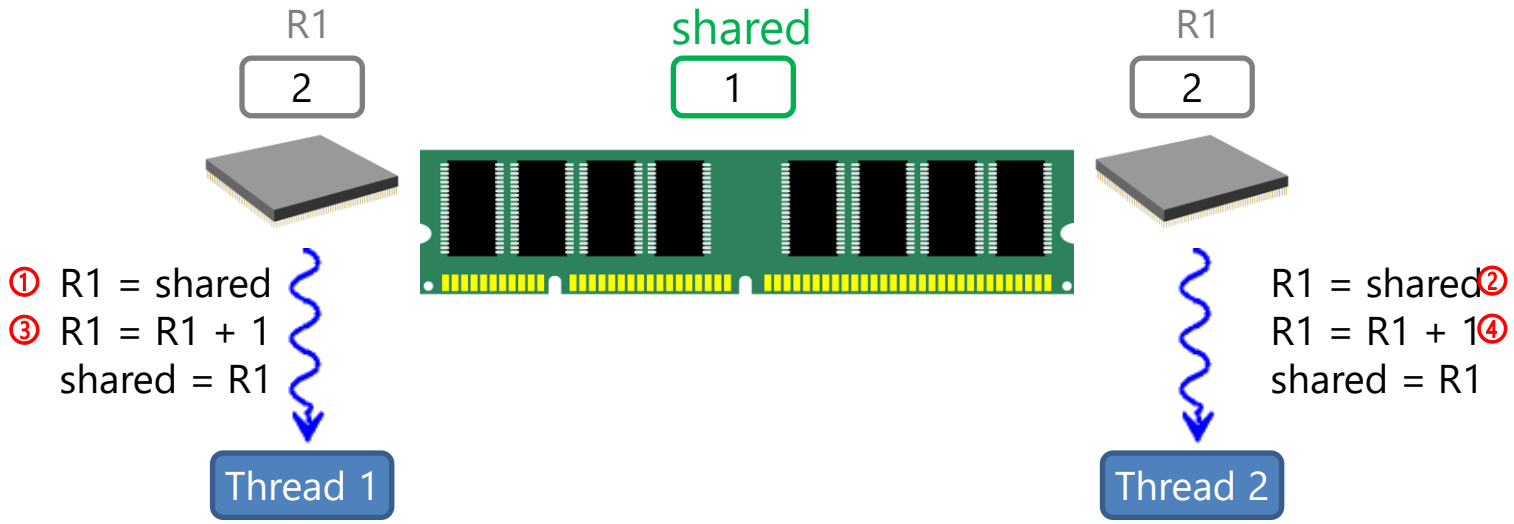
# Review: Datarace Example

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



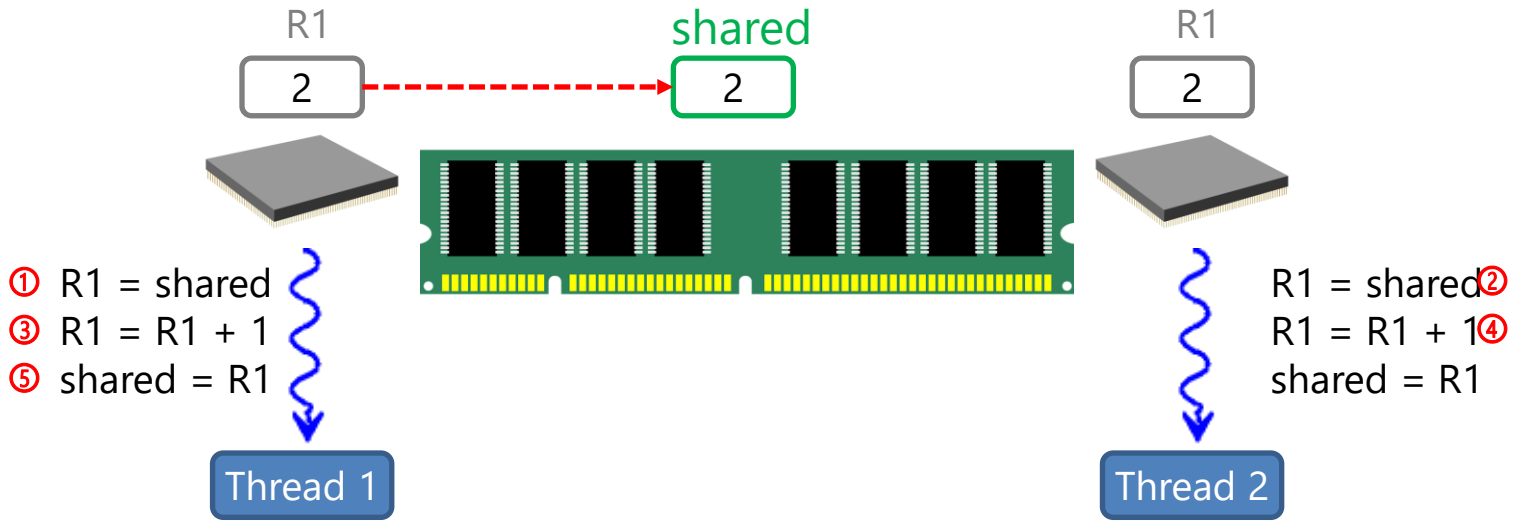
# Review: Datarace Example

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



# Review: Datarace Example

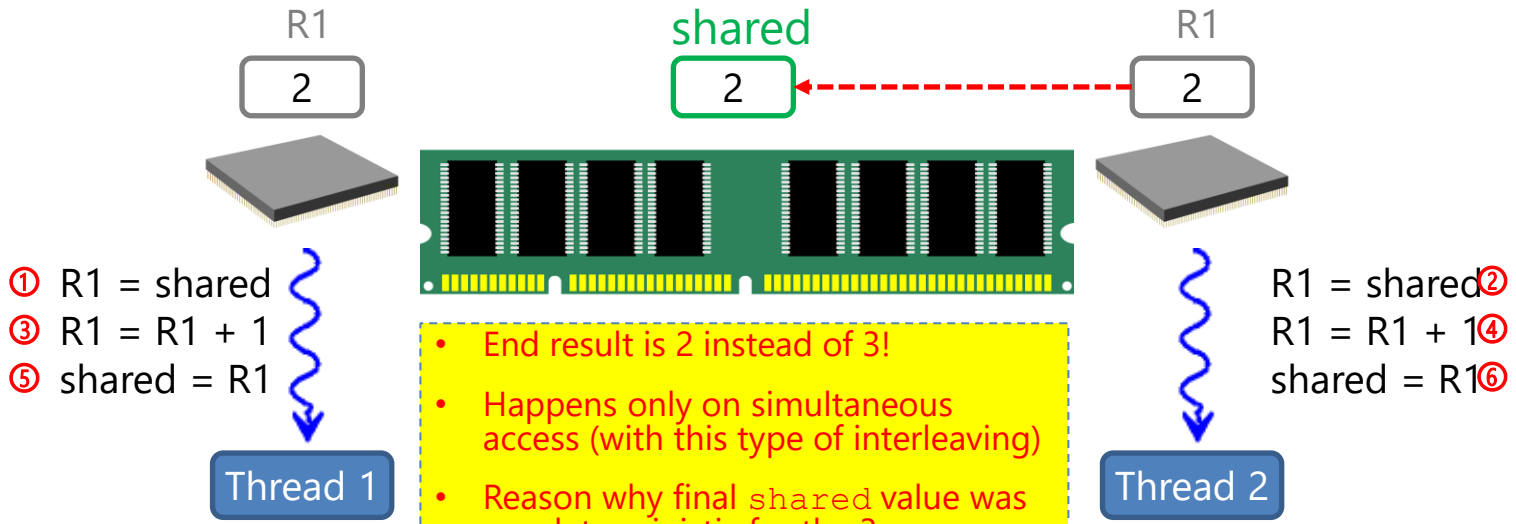
- When two threads do `shared++;` initially `shared` = 1



# Review: Datarace Example

- Why did this occur in the first place?
- Because you **replicated** to CPU registers and worked on your own copy!

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



# Review: Datarace Example

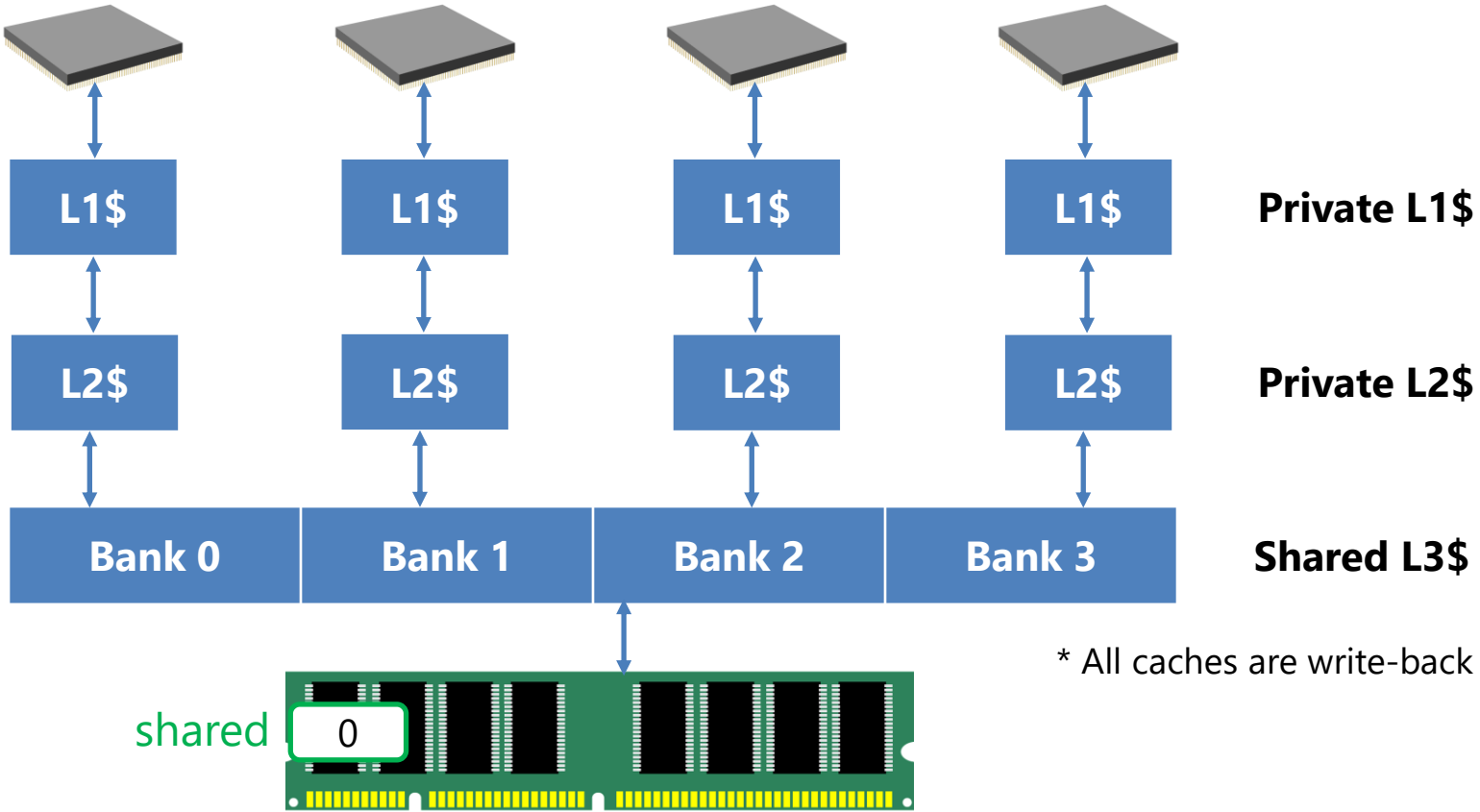
```
pthread_mutex_t lock;
int shared = 0;
void *add(void *unused) {
   for(int i=0; i < 1000000; i++) {
        pthread_mutex_lock(&lock);
        shared++;
        pthread_mutex_unlock(&lock);
   }
   return NULL;
}
int main() {
   …
}
```

```
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!

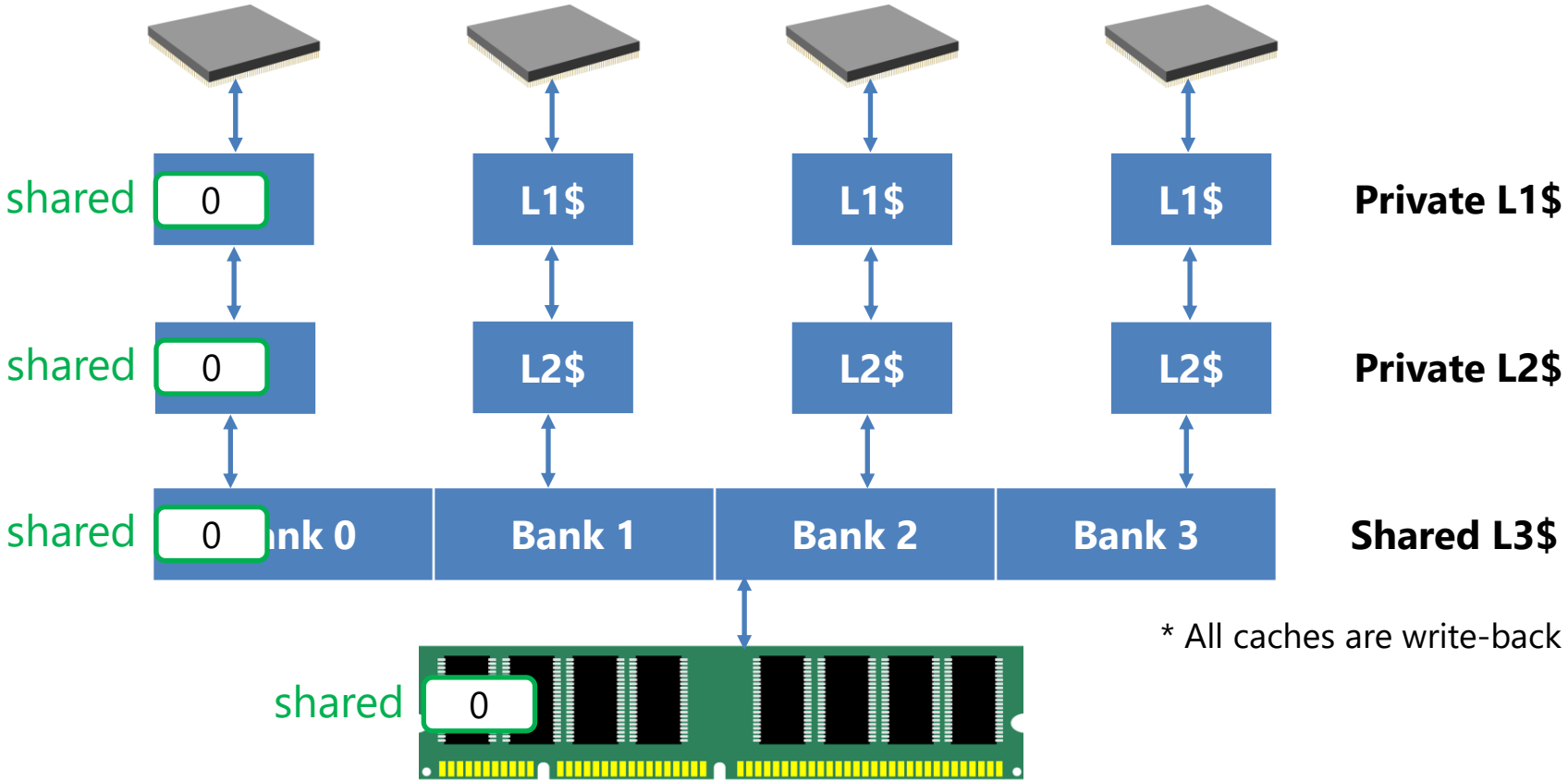
# Caching also does replication!

- What happens if caches sit between processors and memory?



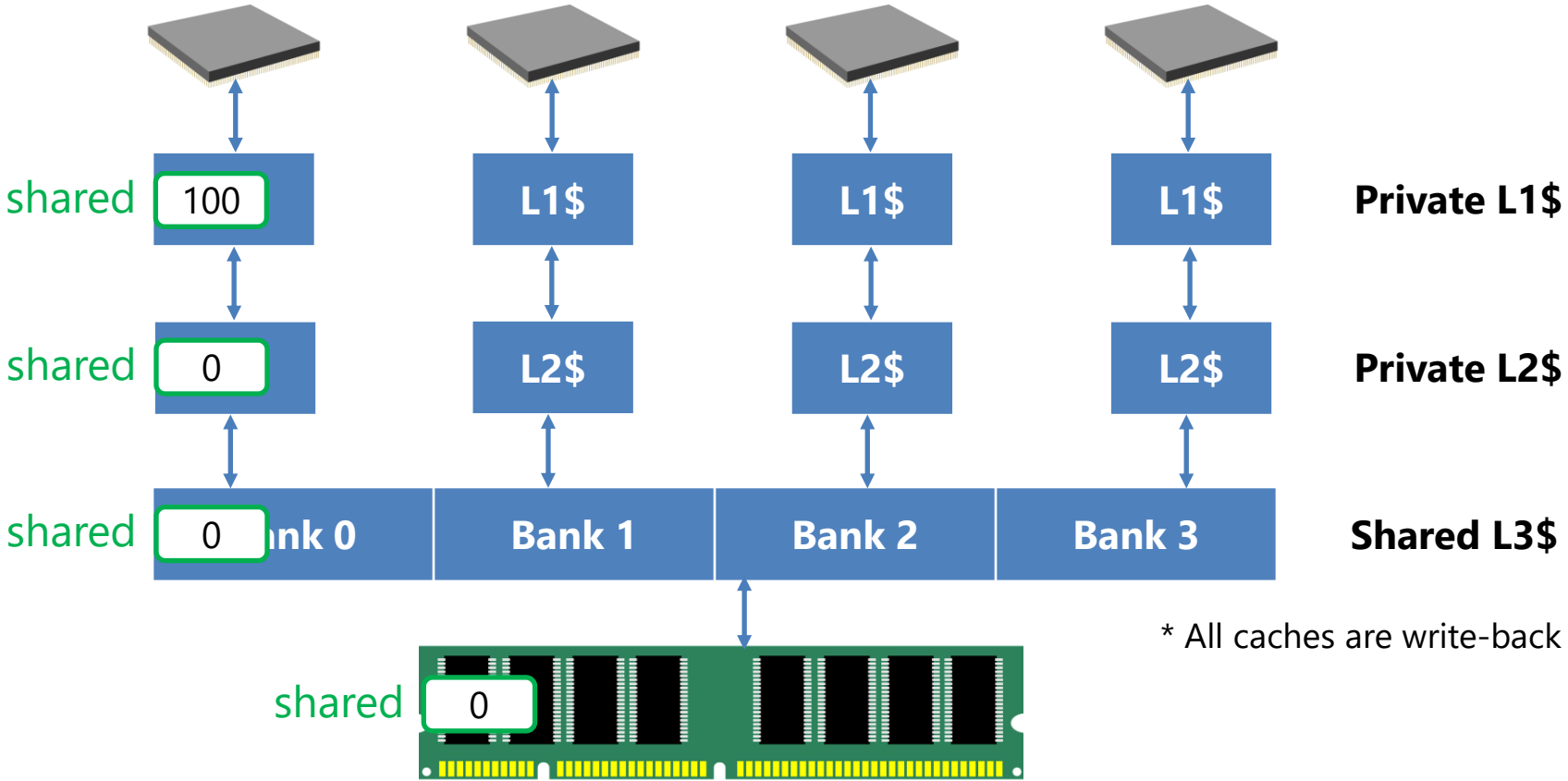
# Caching also does replication!

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



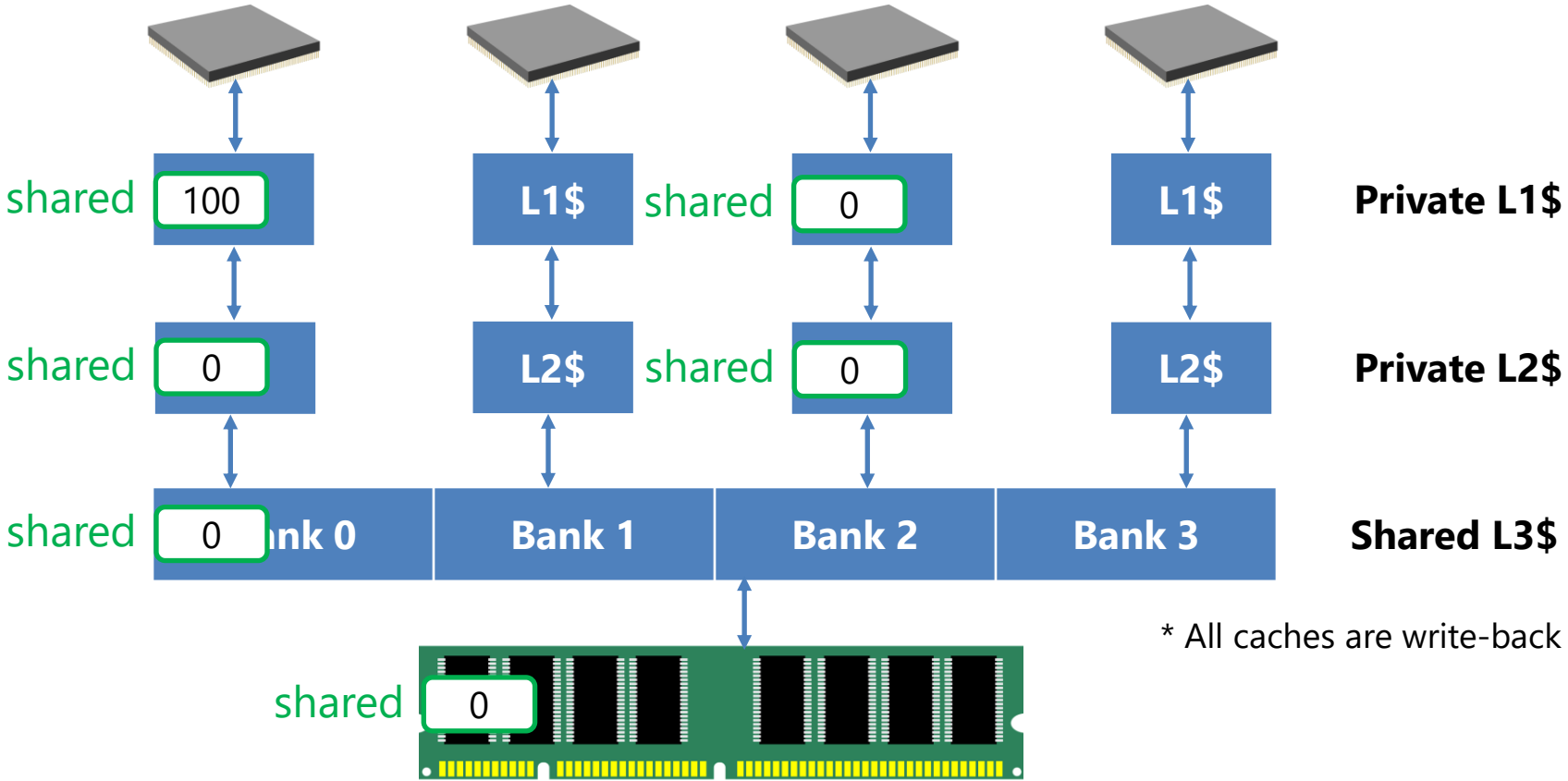
# Caching also does replication!

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



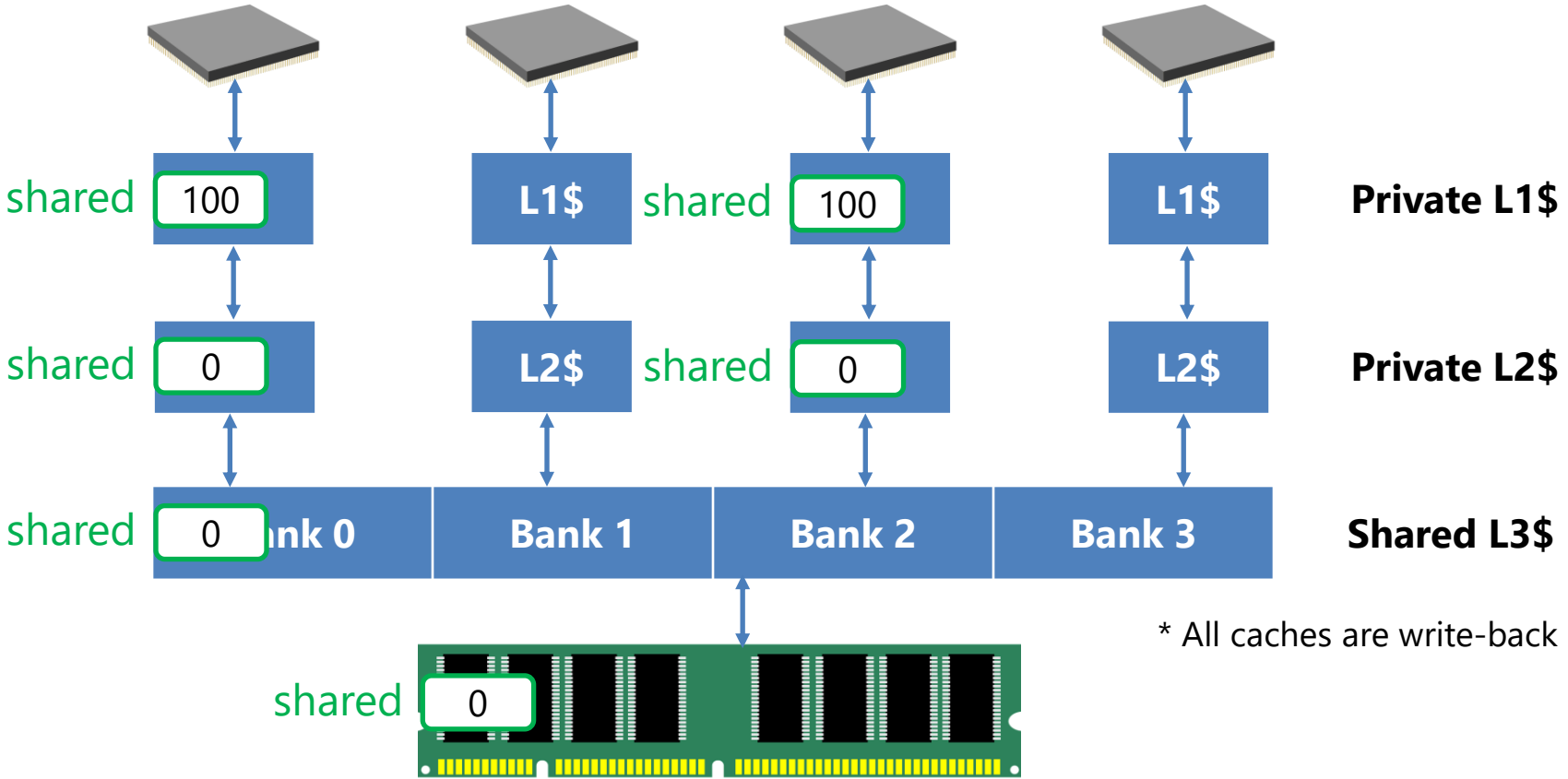
# Caching also does replication!

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



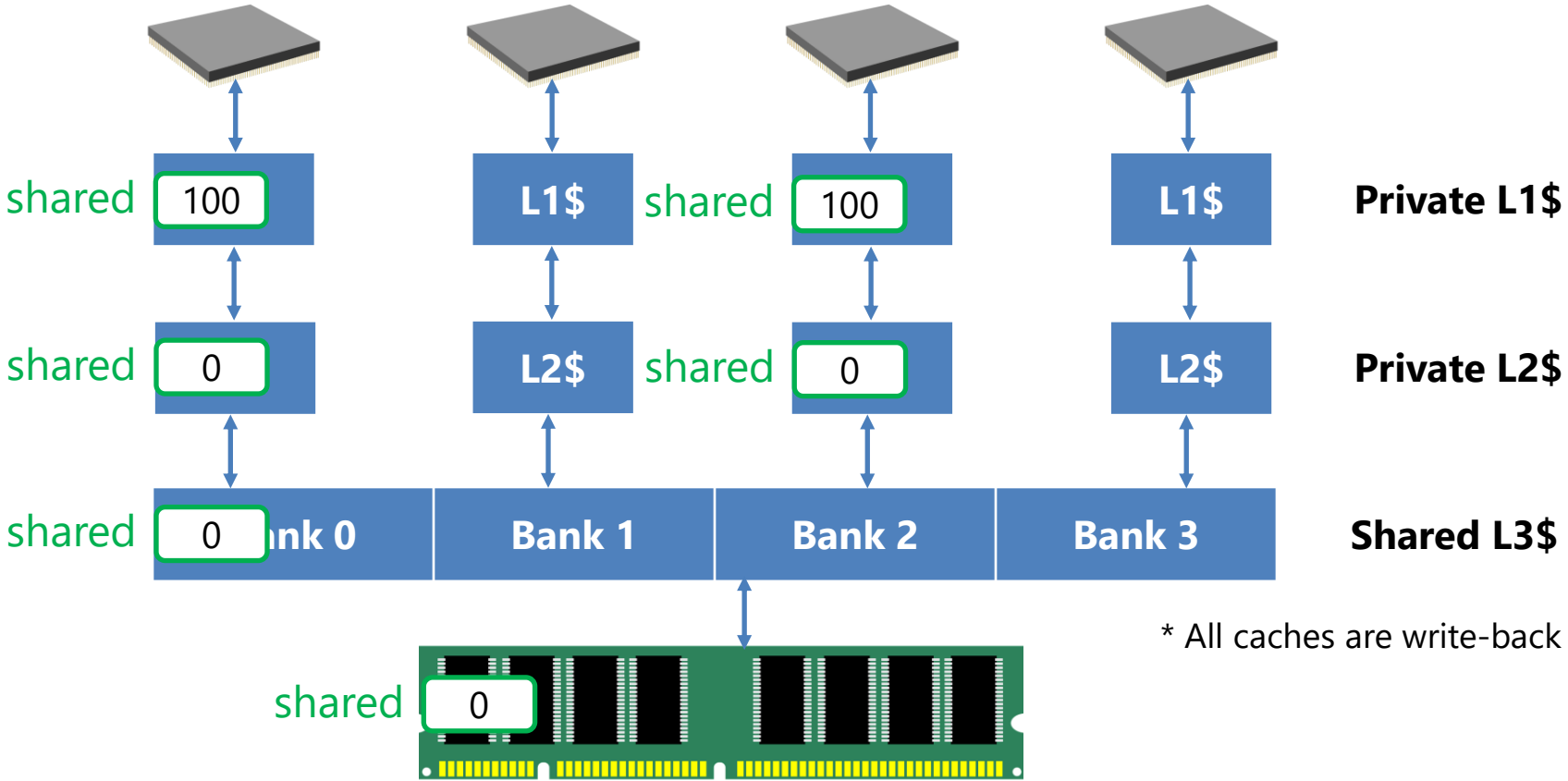
# Caching also does replication!

- Then CPU 2 increments `shared` 100 times to 100 again



# Caching also does replication!

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



# Caching also does replication!

- Q: Does this problem occur with a **shared cache**?
  - A: No! All processors share and work on a **single copy** of data.


  - The problem exists only with private caches.

- Q: How about **write-through** private caches?
  - A: No! Each private copy is **kept consistent at all times**.

- The problem exists for mostly **write-back private** caches.
  - So those are the caches we will focus on.

# 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)

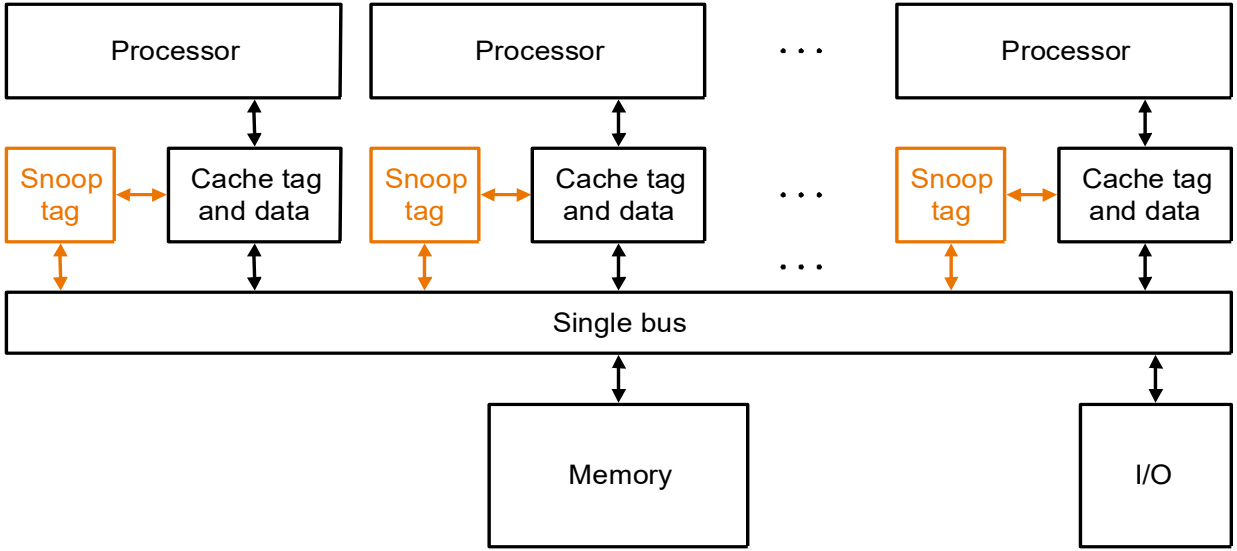
# Implementing Cache Coherence

- How do you guarantee all processors have the same view?

- **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**
- On **read miss** request: other caches check state of local block
  - If **modified**, supply block and write it back. Change state to **shared**.
- On **write miss** request: other caches check state of local block
  - If **shared** or **modified**, **invalidate** (soon to be stale) copy.
  - If **modified**, supply block and write it back.

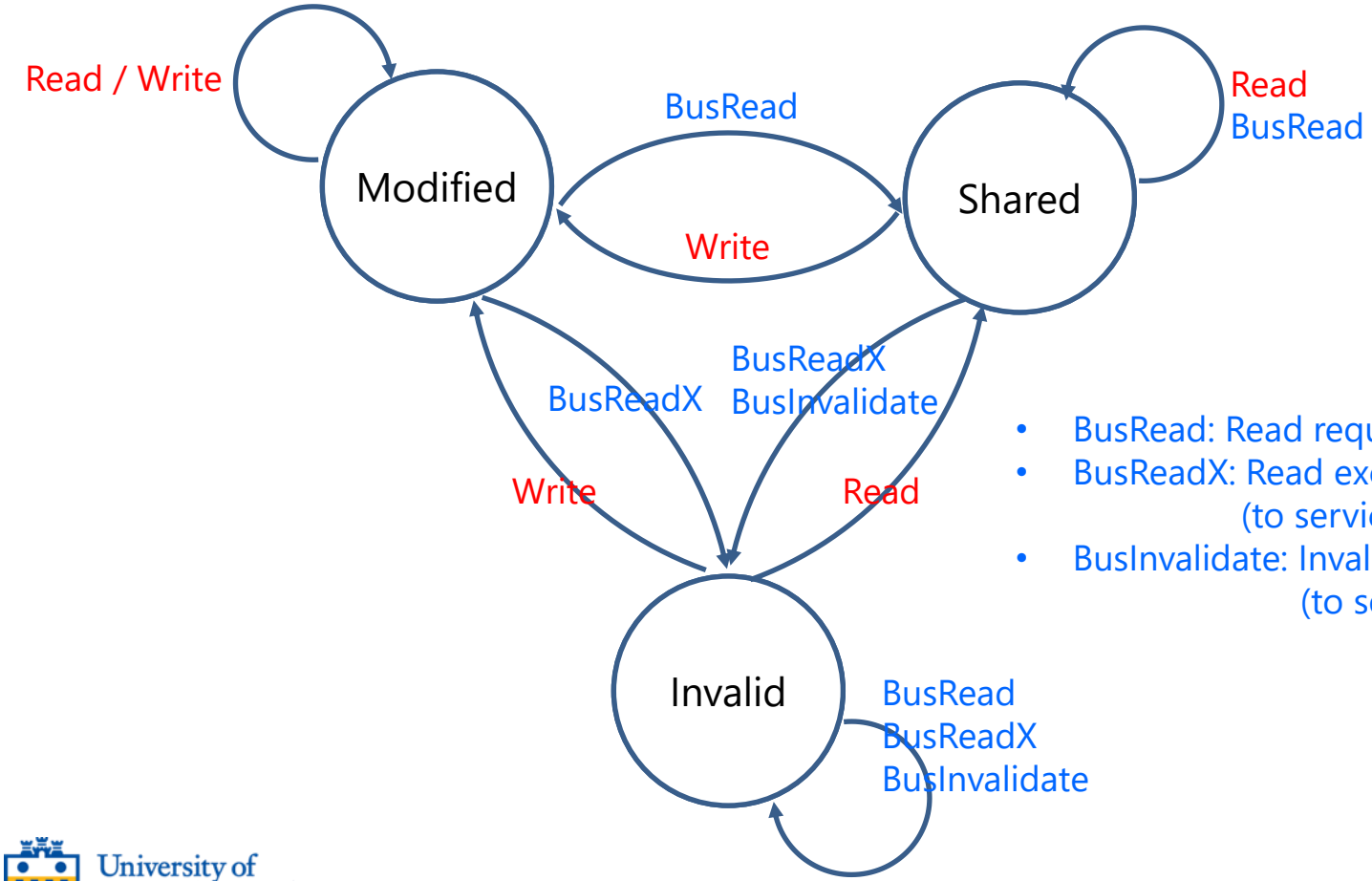
# MSI: Example

- 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 (to write in P1)<br>L = invalid |
| P1 reads A (read hit) | L ← A = 10 (modified) | L = invalid |
| P2 reads A (read miss) | Read A (to read in P2)<br>L ← A = 10 (shared) | L ← A = 10 (shared) |
| P2 writes 20 to A (write hit) | Invalidate A (to write in P2)<br>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 (to write in P1)<br>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 (to service write miss)
- BusInvalidate: Invalidate request is snooped (to service write hit on shared)