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

University of Pittsburgh

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

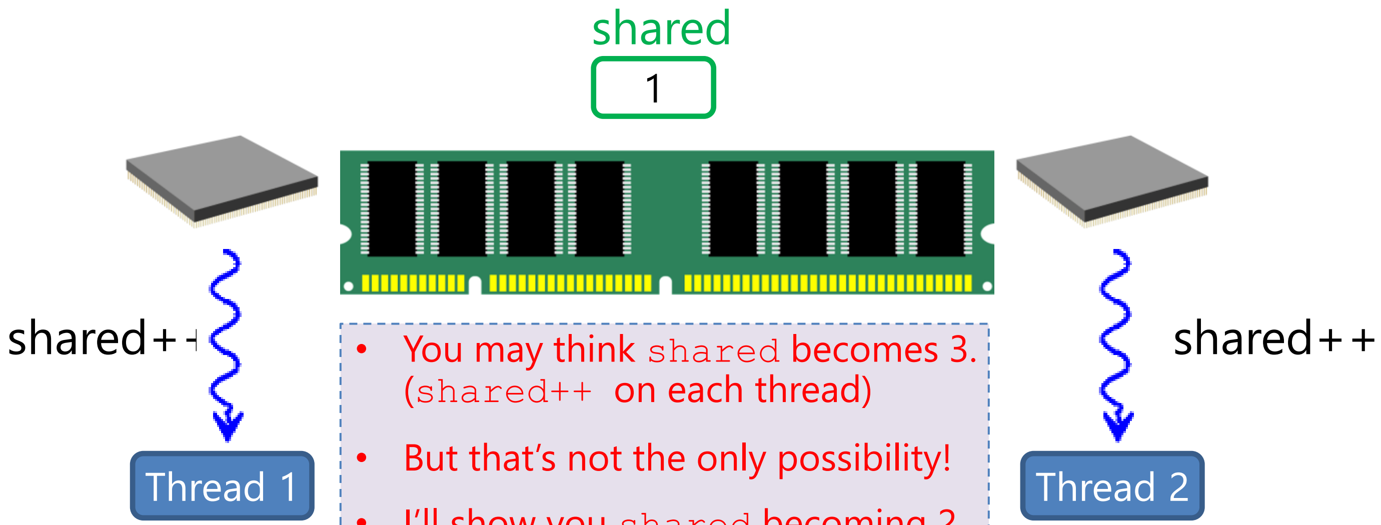
```c
int shared = 0;
void *add(void *unused) {
  for(int i=0; i < 1000000; i++) { shared++; }
  return NULL;
}
int main() {
  pthread_t t;
  // Child thread starts running add
  pthread_create(&t, NULL, add, NULL);
  // Main thread starts running add
  add(NULL);
  pthread_join(t, NULL);
  printf("shared=%d\n",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

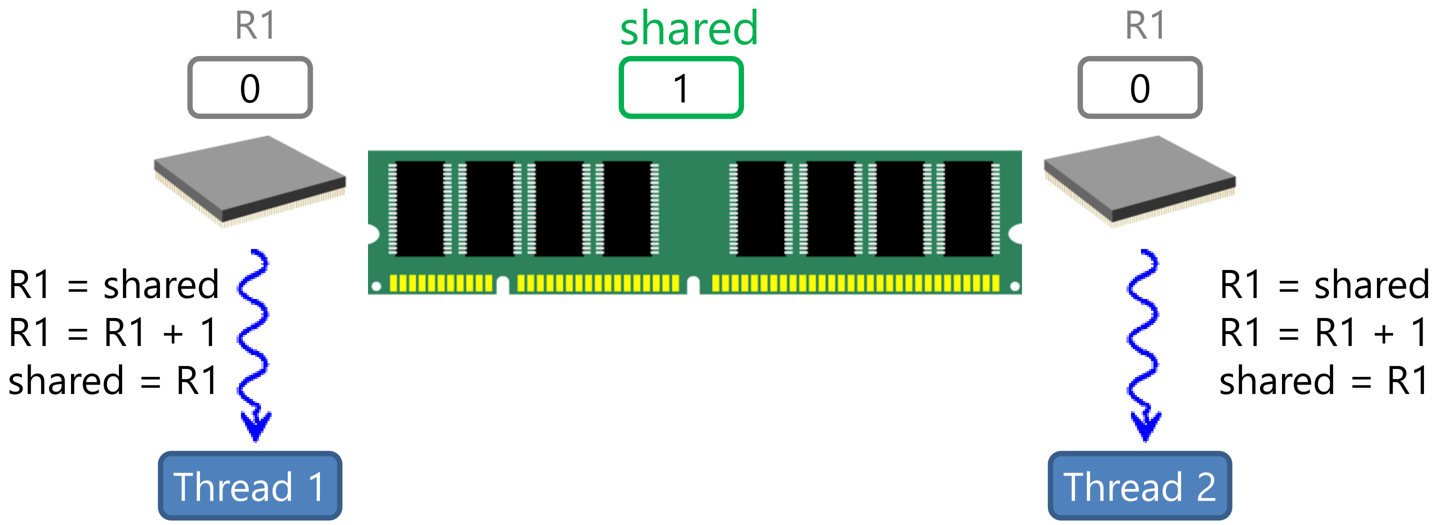
- When two threads do `shared++;` initially `shared` = 1



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

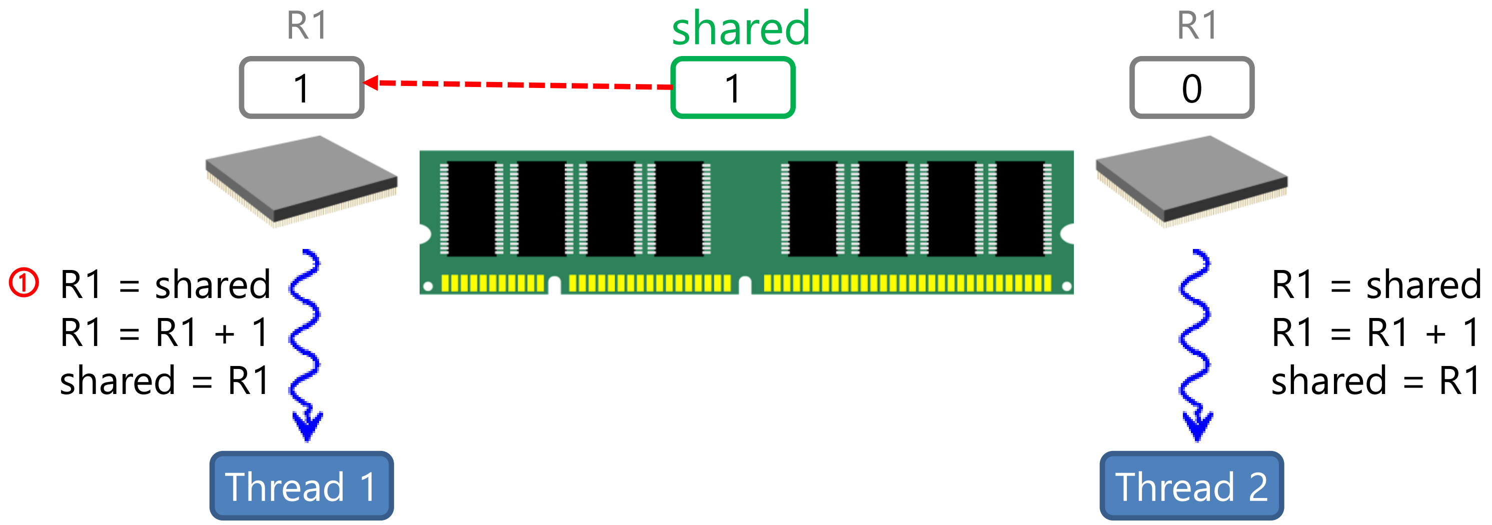
Review: Datarace Example

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



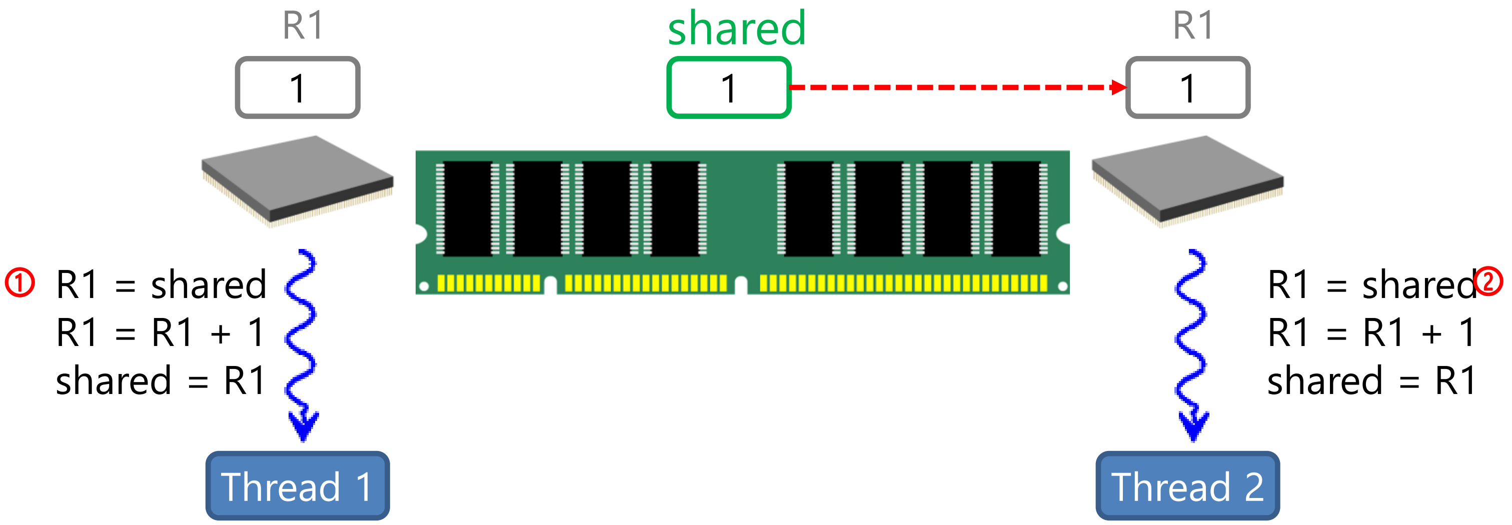
Review: Datarace Example

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



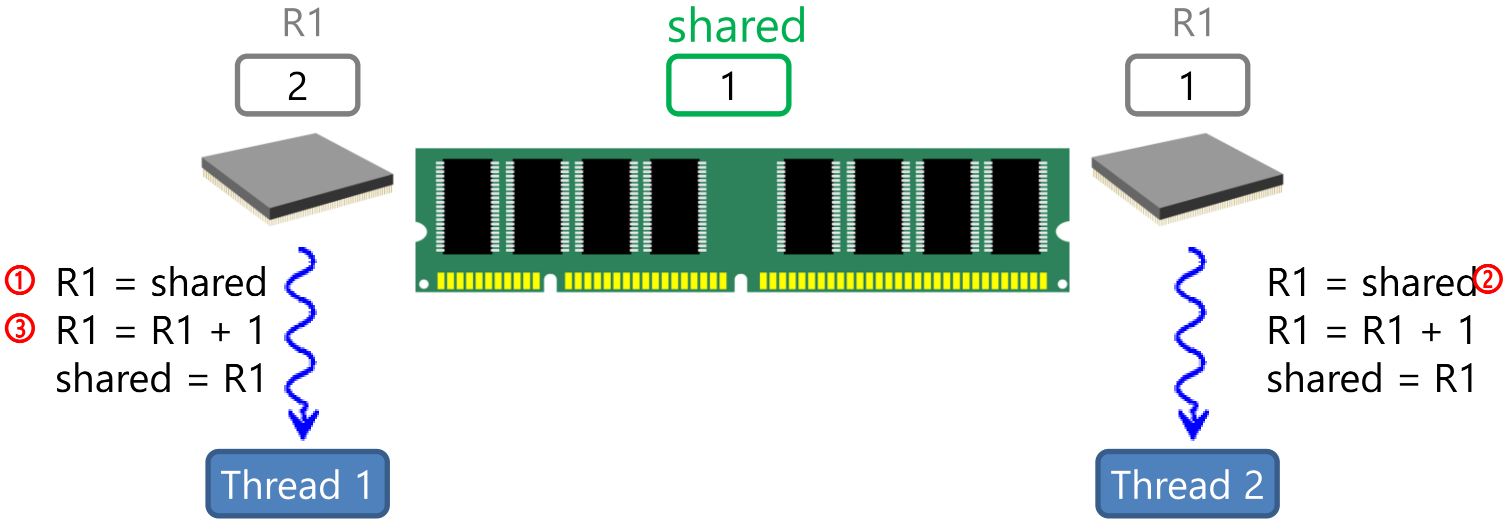
Review: Datarace Example

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



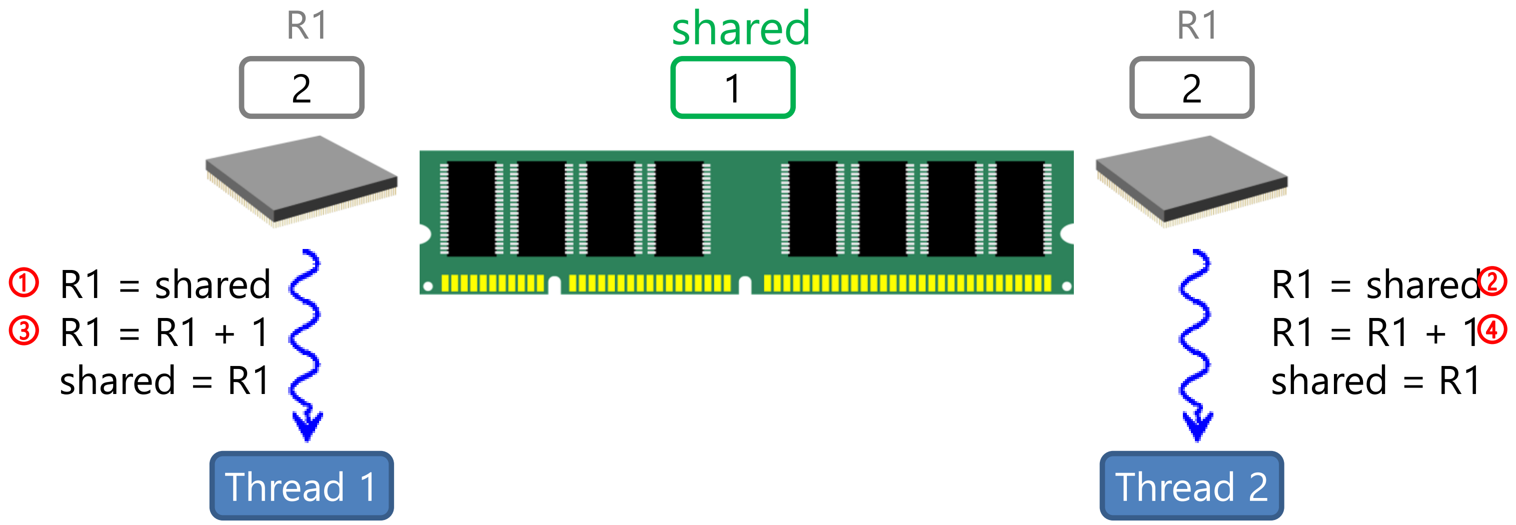
Review: Datarace Example

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



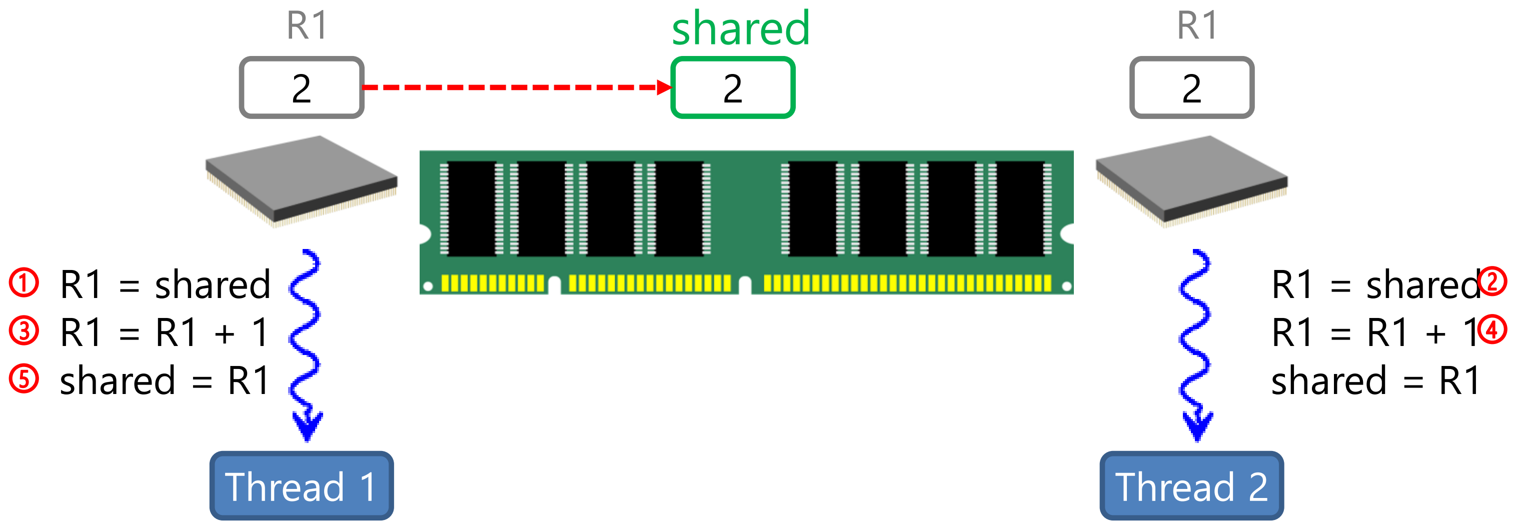
Review: Datarace Example

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



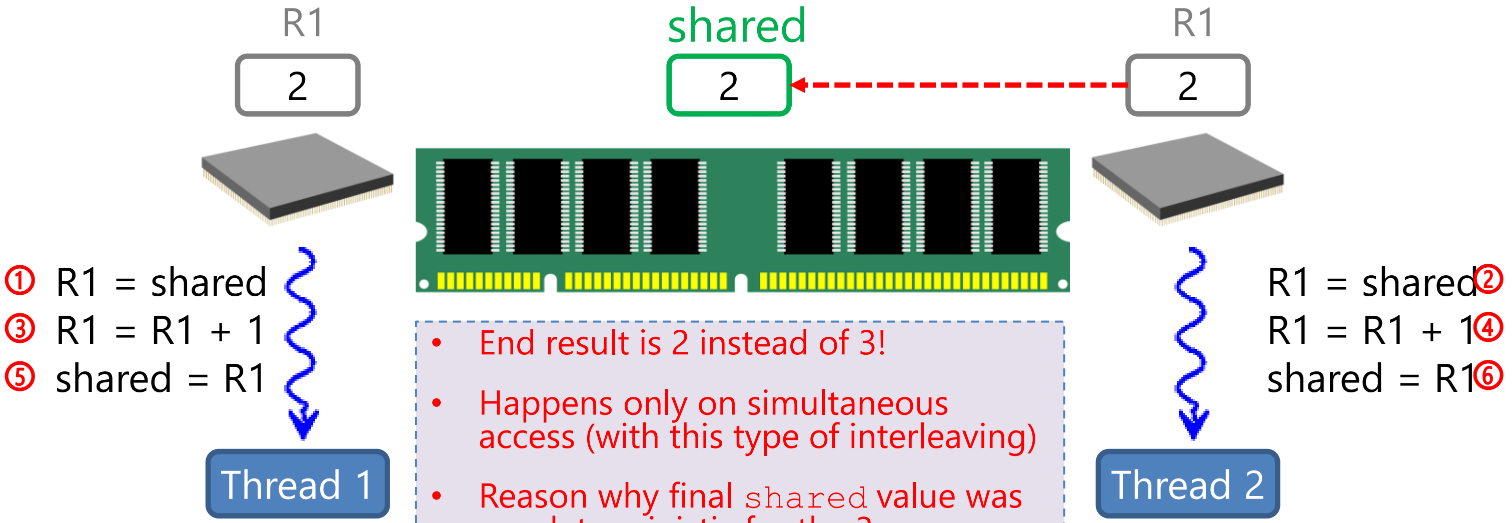
Review: Datarace Example

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



Review: Datarace Example

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



- End result is 2 instead of 3!
- Happens only on simultaneous access (with this type of interleaving)
- Reason why final `shared` value was nondeterministic for the 3 runs

Review: Datarace Example

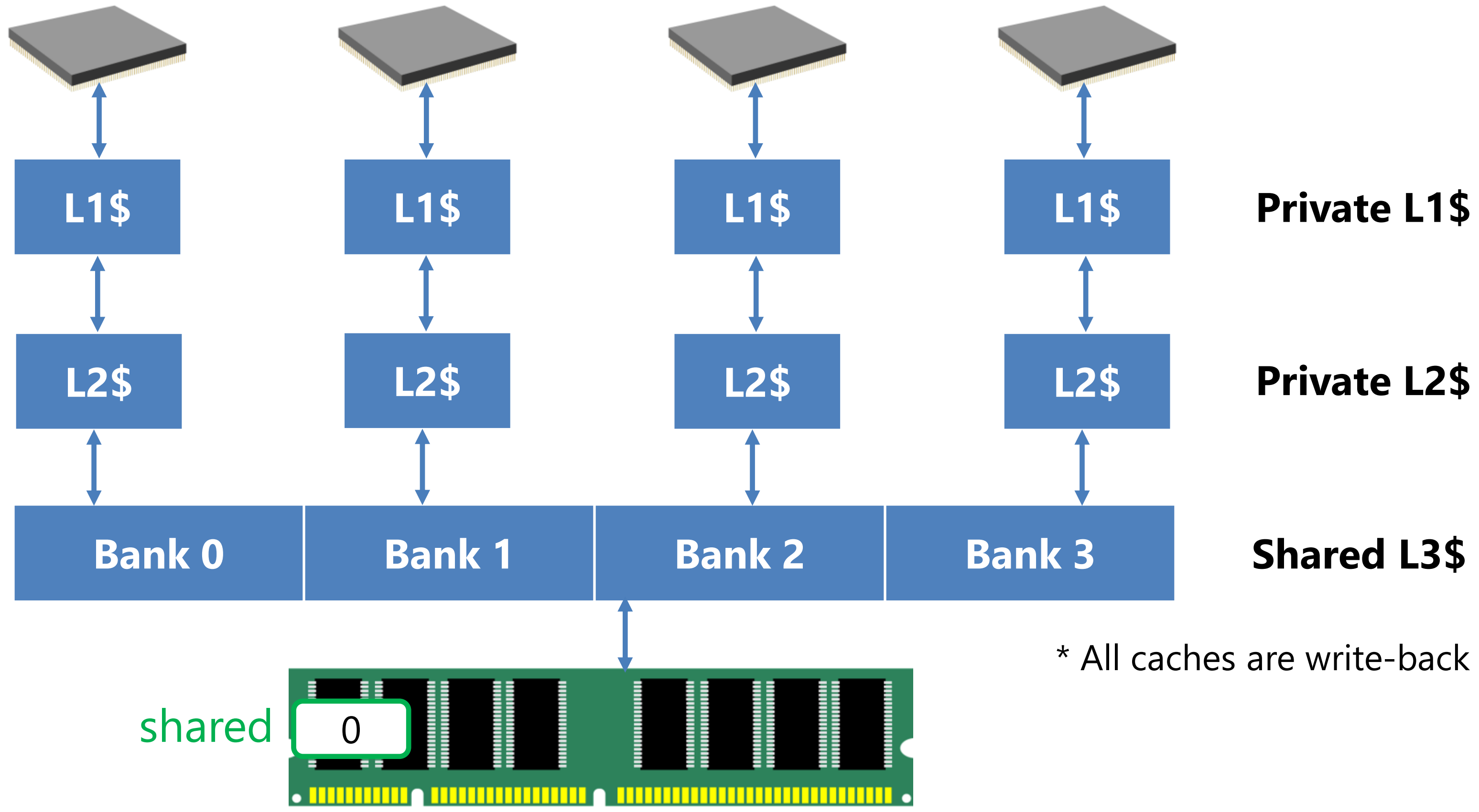
```
pthread_mutex_t lock;
int shared = 0;
void *add(void *unused) {
   for(int i=0; i < 1000000; i++) {
         pthread_mutex_lock(&lock);
         shared++;
         pthread_mutex_unlock(&lock);
   }
   return NULL;
}
int main() {
   …
}
```

```
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?

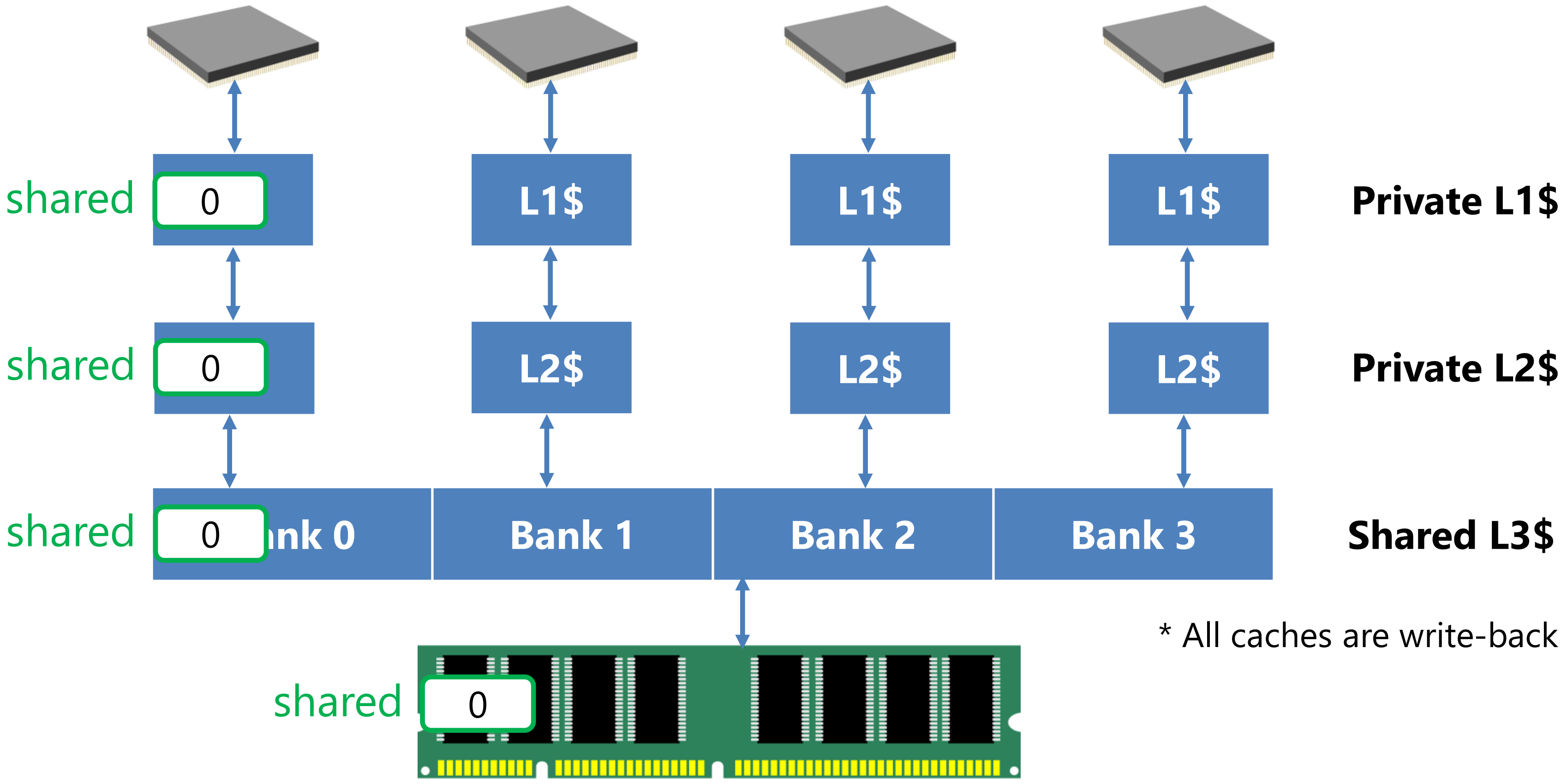
Caching can complicate things

- What happens if caches sit between processors and memory?



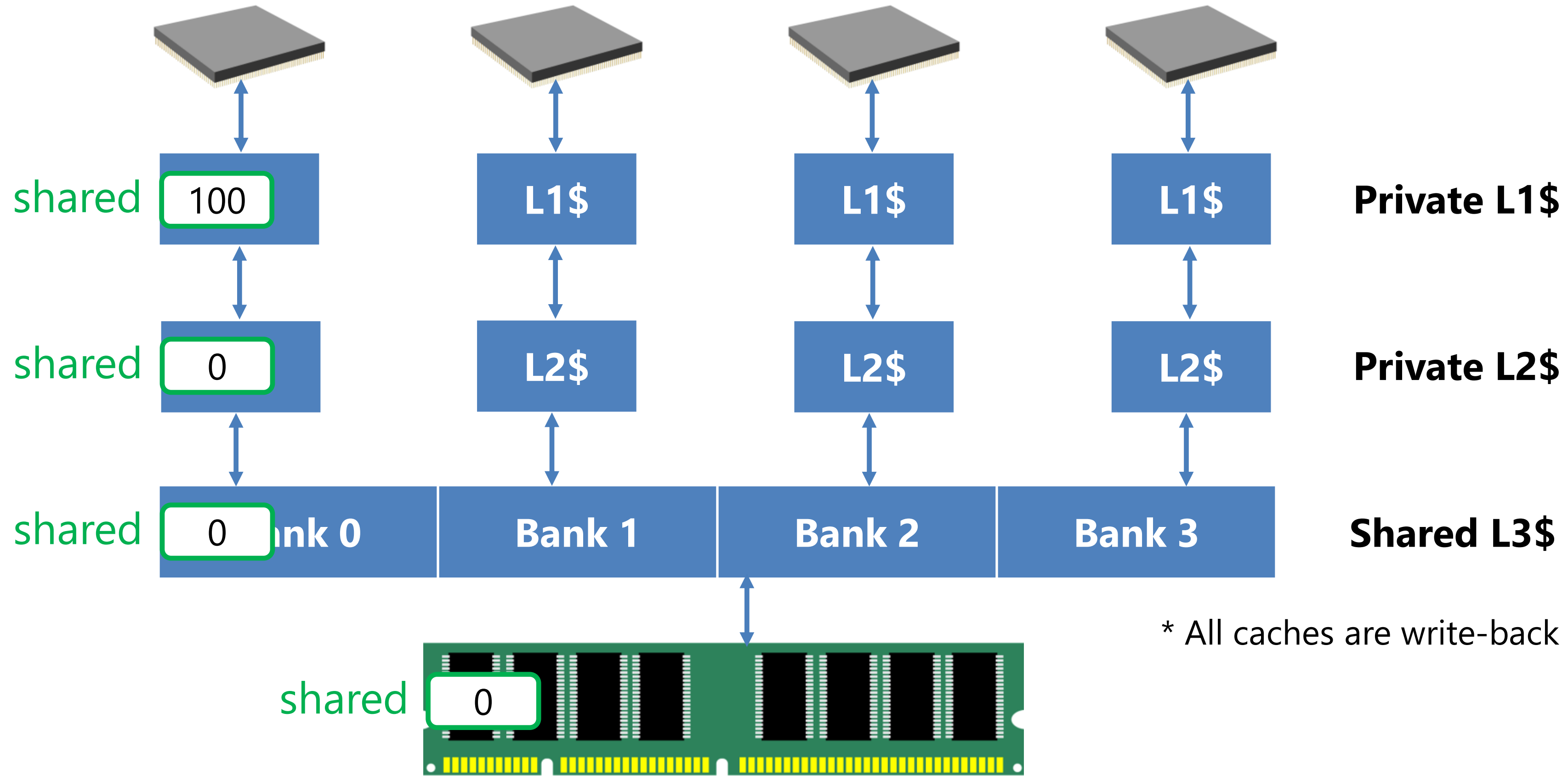
Caching can complicate things

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



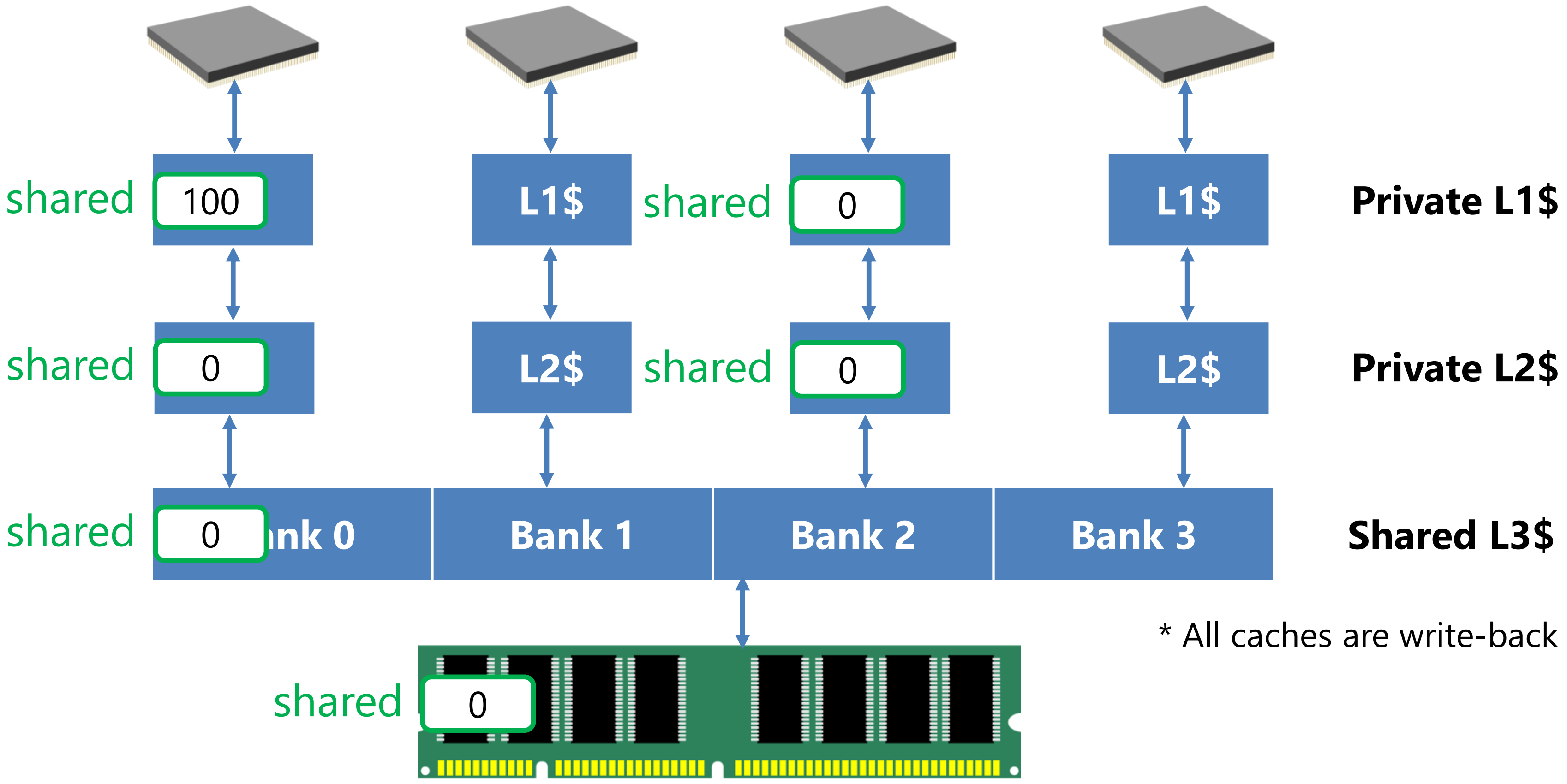
Caching can complicate things

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



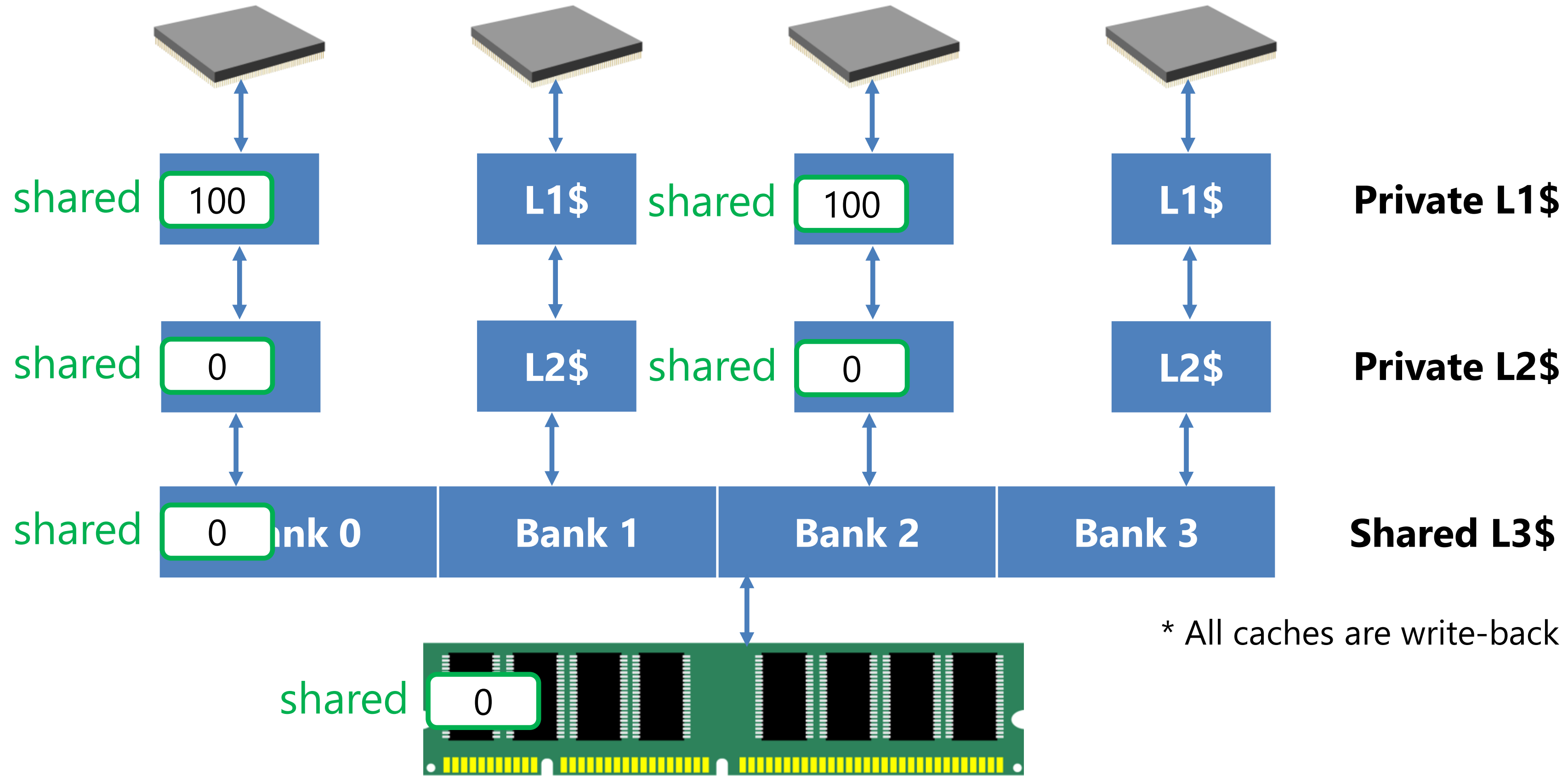
Caching can complicate things

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



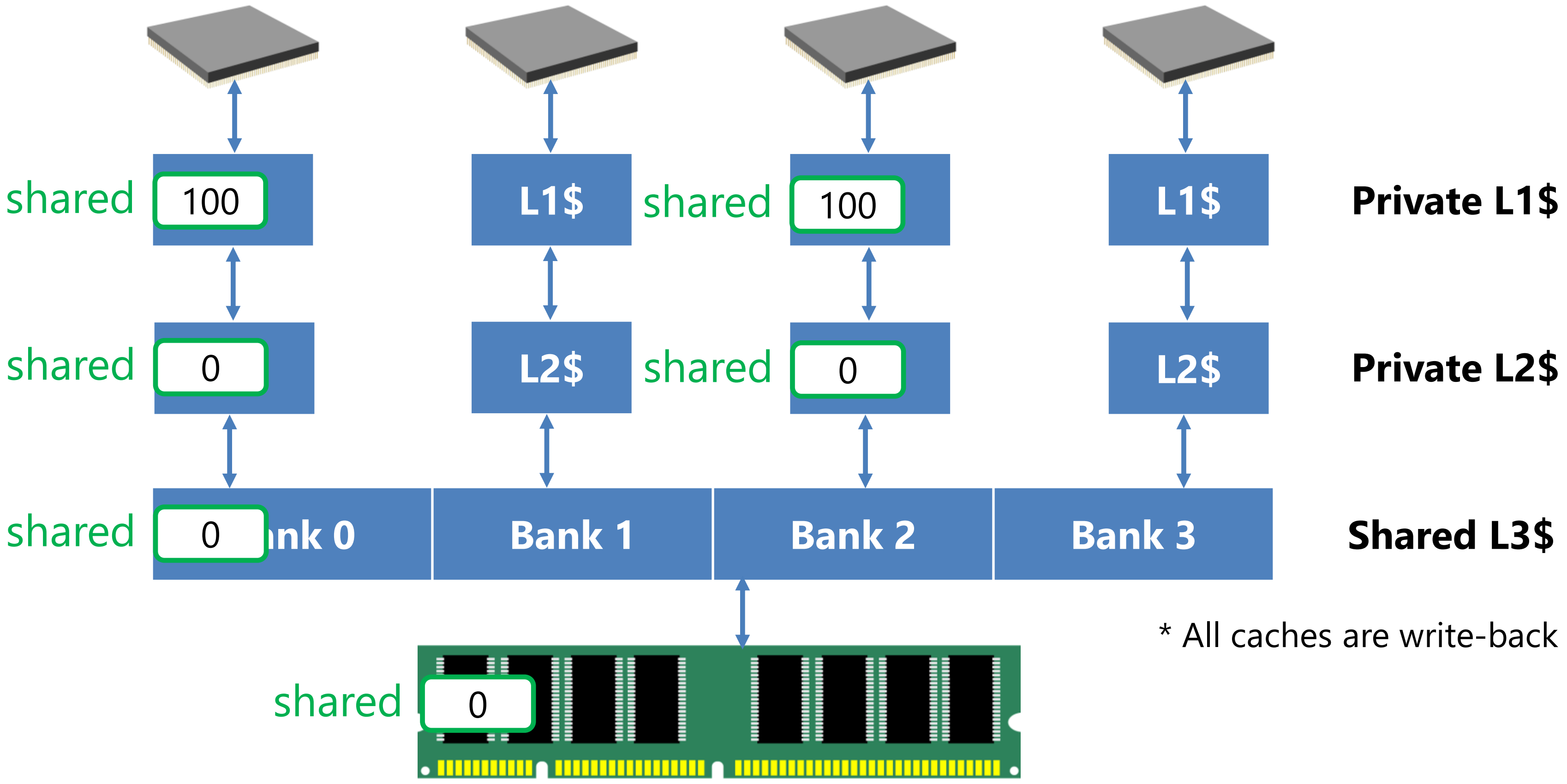
Caching can complicate things

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



Caching can complicate things

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



Cache Coherence

Cache Coherence

- **Cache coherence** (loosely defined):
  - All processors of system should see the **same view of memory**

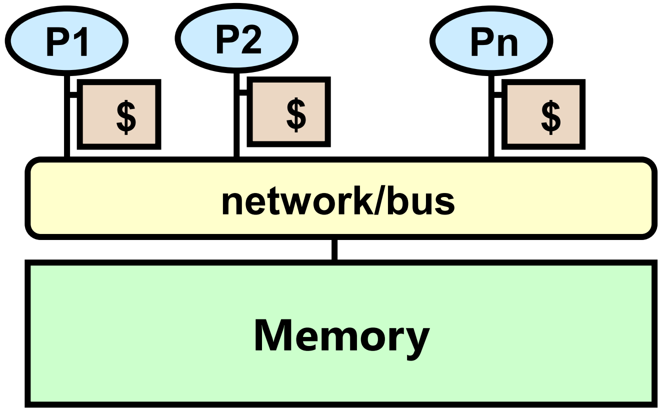
- Sounds simple but hard to pin down what that "view" is:
  - If read by processor P1 follows write by processor P2 to addr X
    → P1 must read value written by P2
  - If write of processor P1 follows write by processor P2 to addr X
    → Other processors must "view" the two writes in that order
  - … and a bunch of other rules

- Each ISA has a different definition of what that "view" means
  - **Memory consistency model**: definition of what "view" means
  - But that's a discussion for another day…

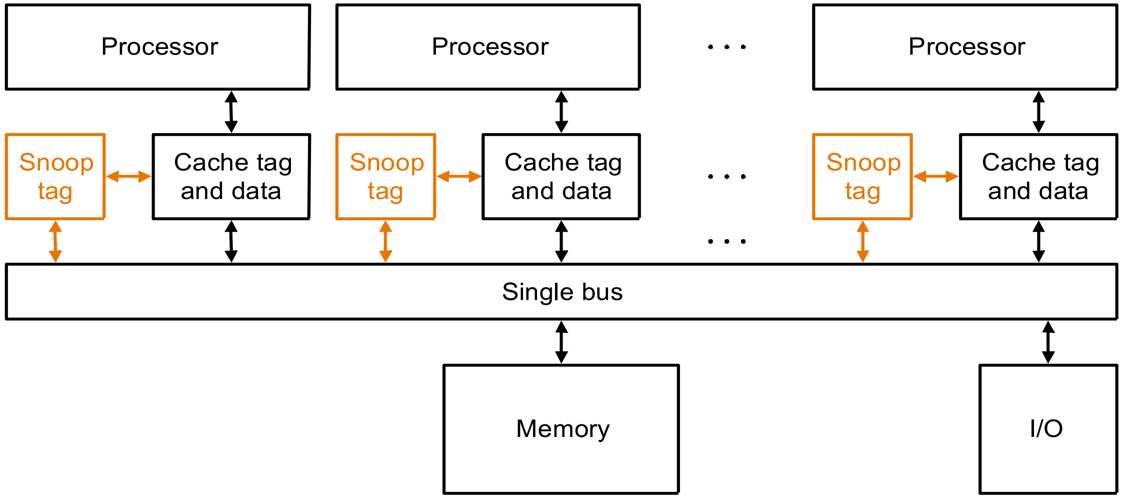
Implementing Cache Coherence

- Working off the lose definition of “same view of memory”, 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)
  - … usually named after the states in cache controller FSM



Snooping Cache Coherence Protocol



- Each processor monitors (snoops) the activity on the bus
- On a read, all caches check to see if they have a copy of the requested block. If yes, they may have to supply the data (will see how).
- On a write, all caches check to see if they have a copy of the data. If yes, they either invalidate the local copy, or update it with the new value.
- Can have either *write back* or *write through* policy (the former is usually more efficient, but harder to maintain coherence).

University of Pittsburgh

MSI: A Snoopy Cache Coherence Protocol

- A coherence protocol for write-back caches (use invalidation)
- Each block of memory can be:
  - Clean in all caches and up-to-date in memory (Read-Only), or
  - Dirty in exactly one cache (Read/Write), or
  - Uncached: not in any caches
- Correspondingly, we record the sate of each block in a cache as one of:
  - Shared: block can be read (clean, read-only)
  - Modified: cache has only copy, it is writeable, and dirty
  - Invalid: block contains no data
- A read miss to a block generates a bus transaction -- if a cache has the block “modified”, it writes back the block and the block becomes “shared”.
- A write hit to a shared block generates an “invalidate” on the bus-- all other caches that have the block should invalidate that block – the block becomes “modified”.
- A write hit to a “modified” block does not generate a write back or change of state.
- A write miss (to an invalid block) generates a bus transaction (request)
  - If a cache has the block as “shared”, it invalidates it (memory supplies the block)
  - If a cache has the block in “modified”, it supplies the block and changes it state to “invalid”. In this case, memory does not supply the block.

University of Pittsburgh

MSI: Example

- Assumes that blocks A and B map to same cache location L.
- Initially neither A nor B is cached
- Block size = one word

Event	In P1's cache	In P2's cache
	L = invalid	L = invalid
P1 writes 10 to A (write miss)	P1 requests A(to write) L ← A = 10 (modified)	L = invalid
P1 reads A (read hit)	L ← A = 10 (modified)	L = invalid
P2 reads A (read miss)	A is written back L ← A = 10 (shared)	P2 requests A(to read) L ← A = 10 (shared)
P2 writes 20 to A (write hit)	L = invalid	Put invalidate A on bus L ← A = 20 (exclusive)
P2 writes 40 to A (write hit)	L = invalid	L ← A = 40 (exclusive)
P1 write 50 to A (write miss)	P requests A(to write) L ← A1 = 50 (modified)	A is written back L = invalid

University of Pittsburgh