Multiprocessors and Caching

CS 1541
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
    - Ajax (Asynchronous JavaScript and XML) – API for web apps
  - Data exchange protocols: TCP/IP, UDP/IP, JSON, XML…

- **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 should have learned it in CS 449.
  - But if you didn’t, don’t worry I’ll go over it.

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;
}
```

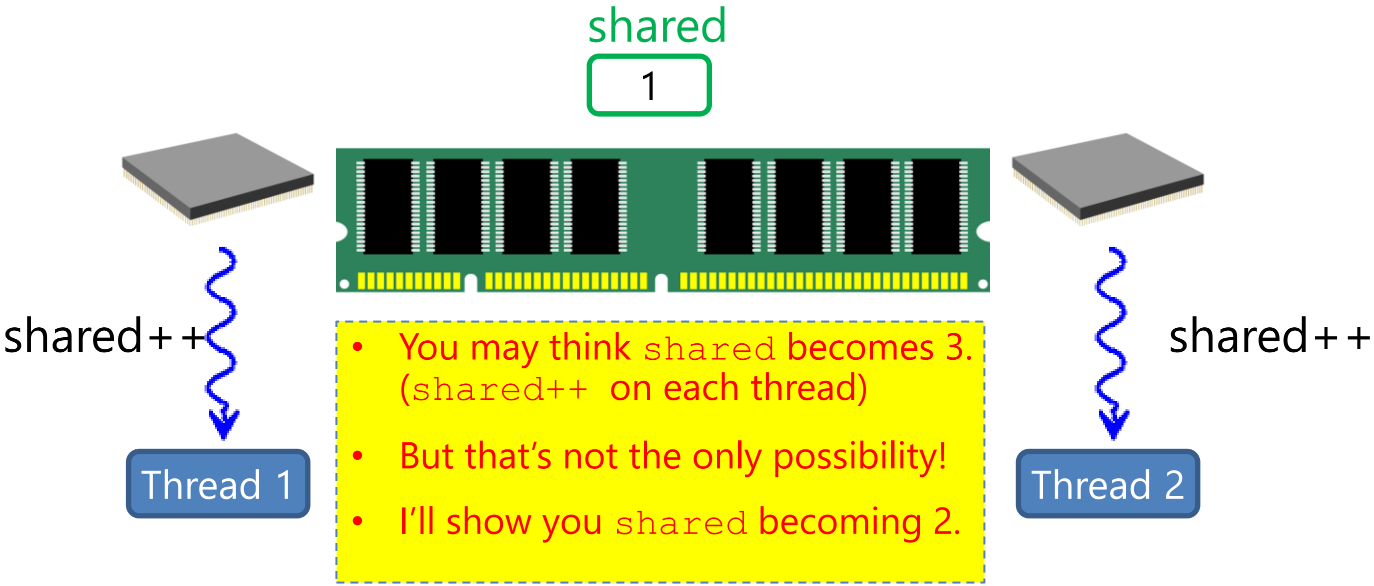
```
bash-4.2$ ./datarace
shared=1085894
bash-4.2$ ./datarace
shared=1101173
bash-4.2$ ./datarace
shared=1065494
```

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

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

Review: Datarace Example

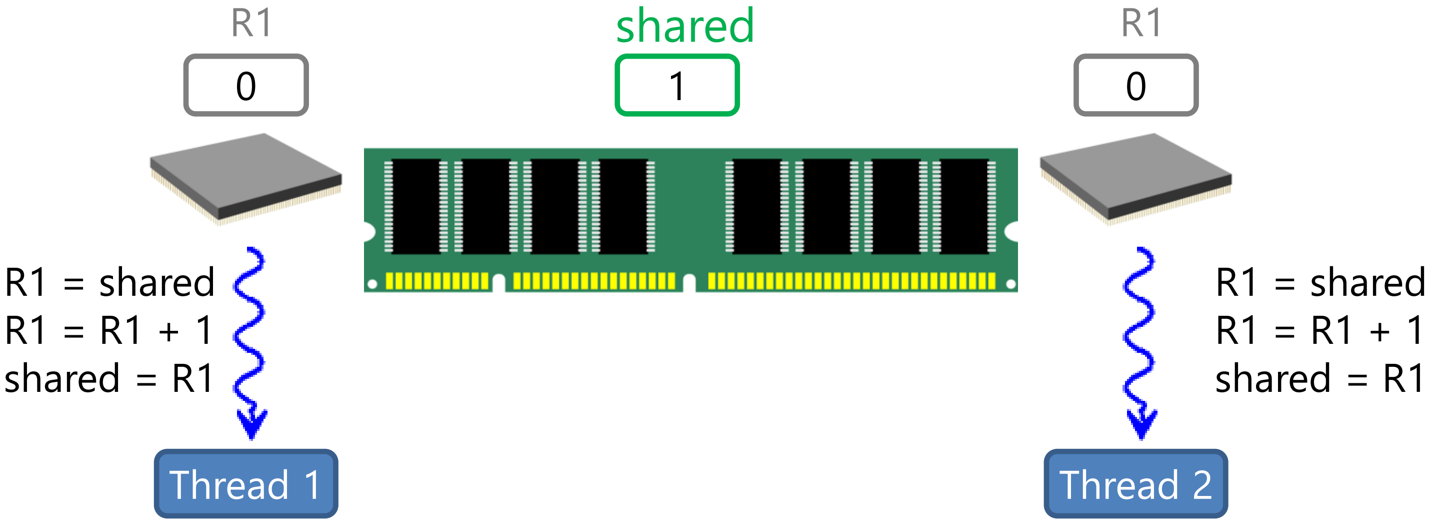
- When two threads do `shared++;` initially `shared = 1`



- 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

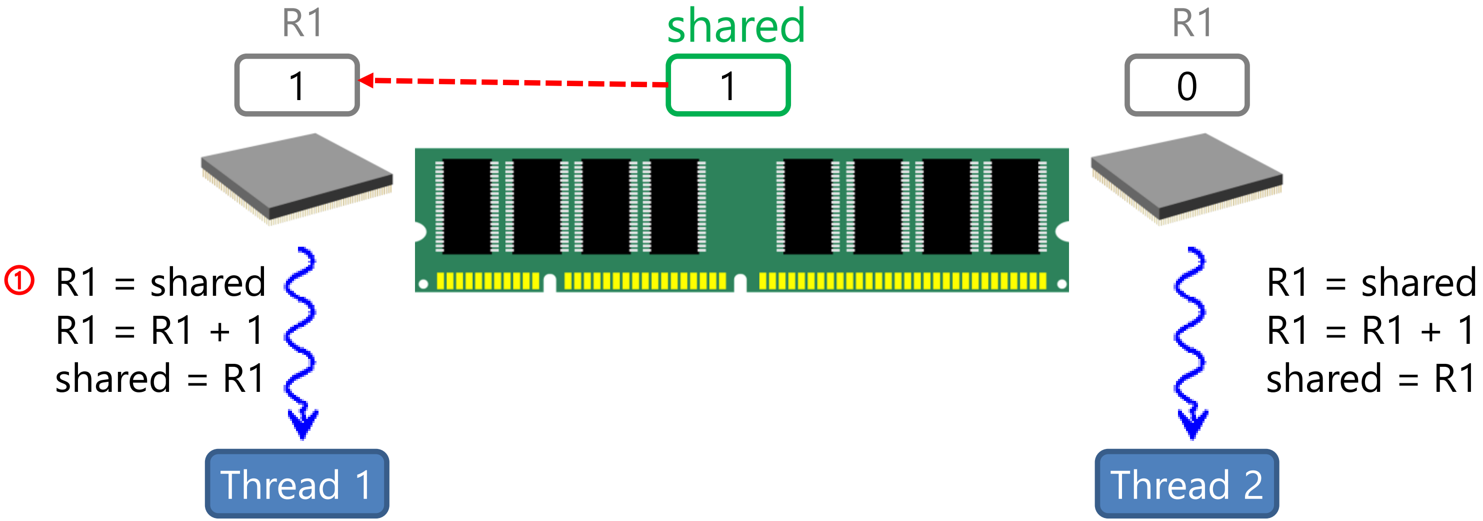


```
R1 = shared
R1 = R1 + 1
shared = R1
```

```
R1 = shared
R1 = R1 + 1
shared = R1
```

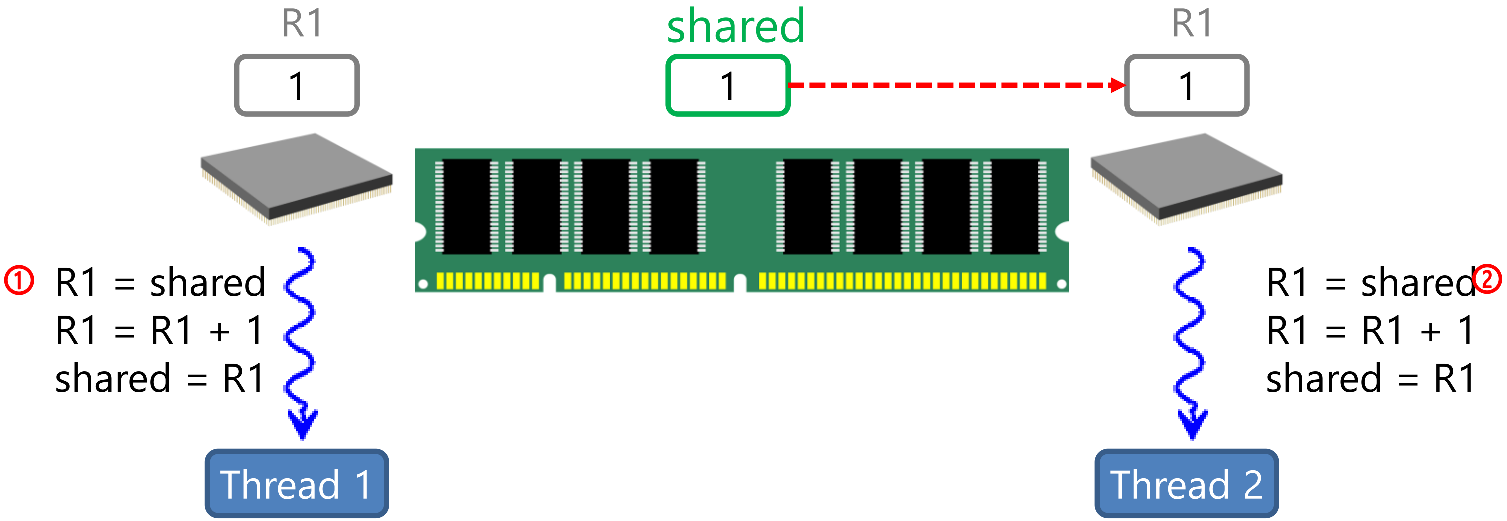
Review: Datarace Example

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



Review: Datarace Example

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



① R1 = shared
R1 = R1 + 1
shared = R1

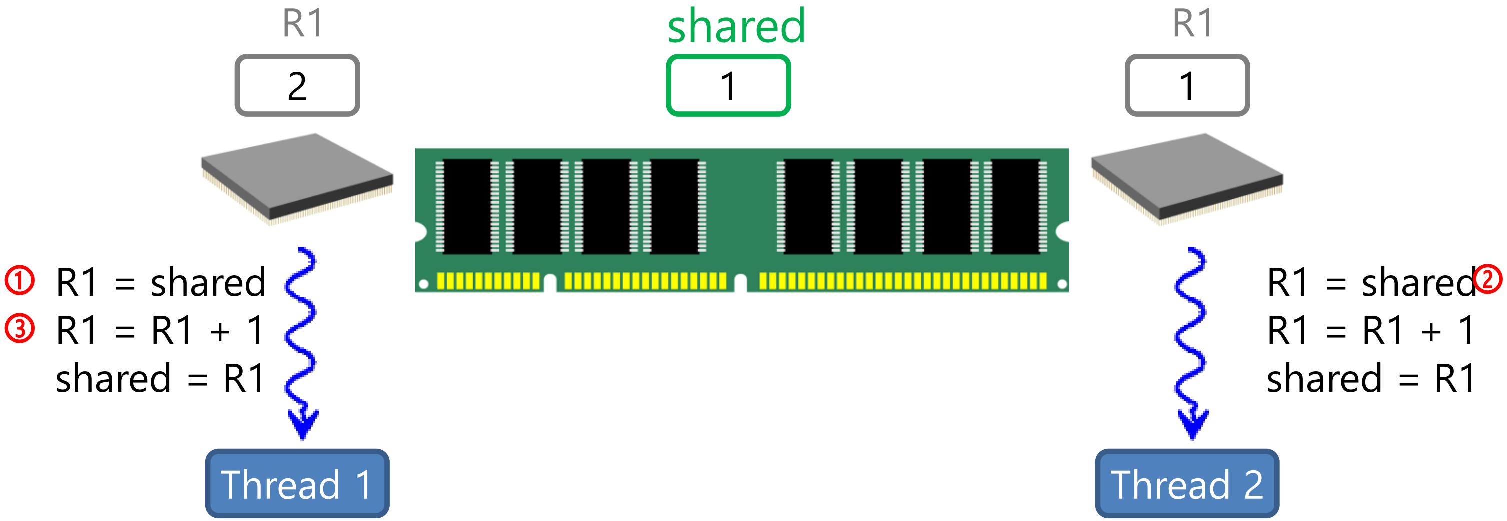
Thread 1

R1 = shared②
R1 = R1 + 1
shared = R1

Thread 2

Review: Datarace Example

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



Thread 1:

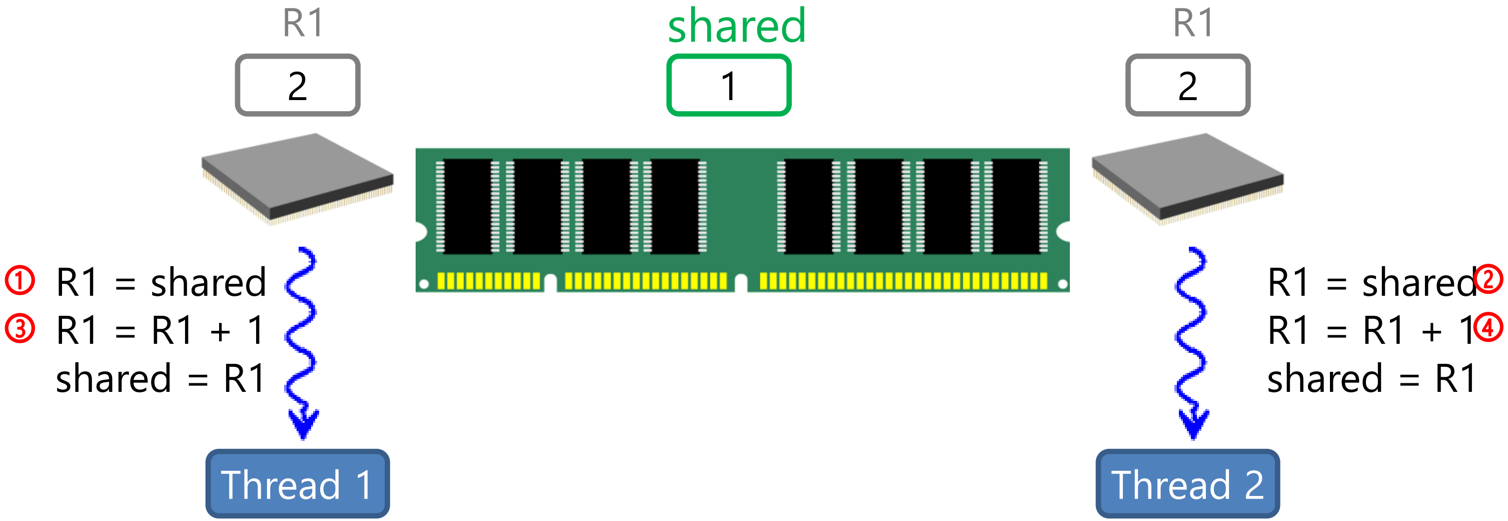
① R1 = shared
③ R1 = R1 + 1
shared = R1

Thread 2:

R1 = shared②
R1 = R1 + 1
shared = R1

Review: Datarace Example

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

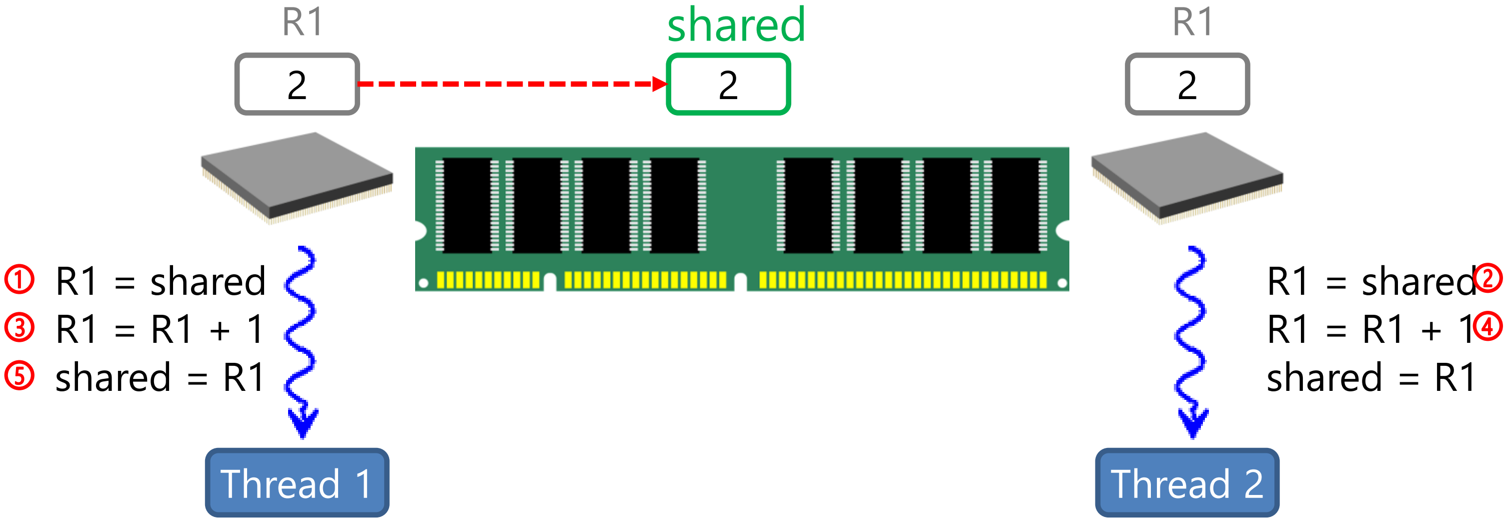


```
① R1 = shared
③ R1 = R1 + 1
  shared = R1
```

```
R1 = shared②
R1 = R1 + 1④
shared = R1
```

Review: Datarace Example

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



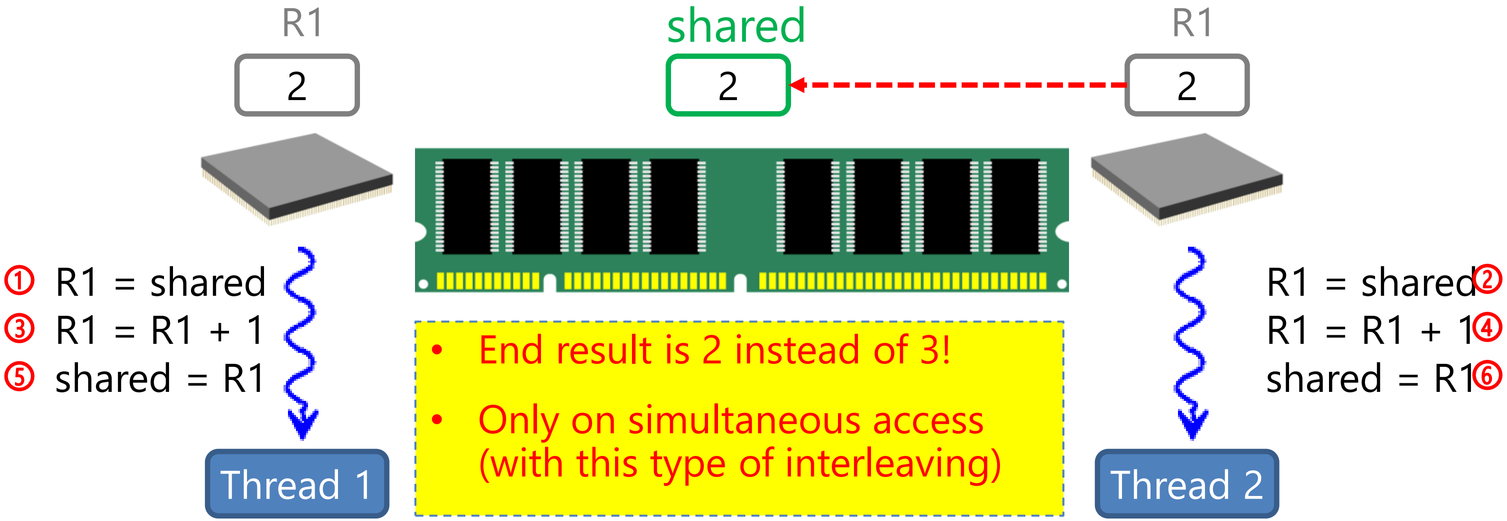
① R1 = shared
③ R1 = R1 + 1
⑤ shared = R1

R1 = shared②
R1 = R1 + 1④
shared = R1

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



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

Thread 1

R1 = shared②
R1 = R1 + 1④
shared = R1⑥

Thread 2

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

Review: Datarace Example

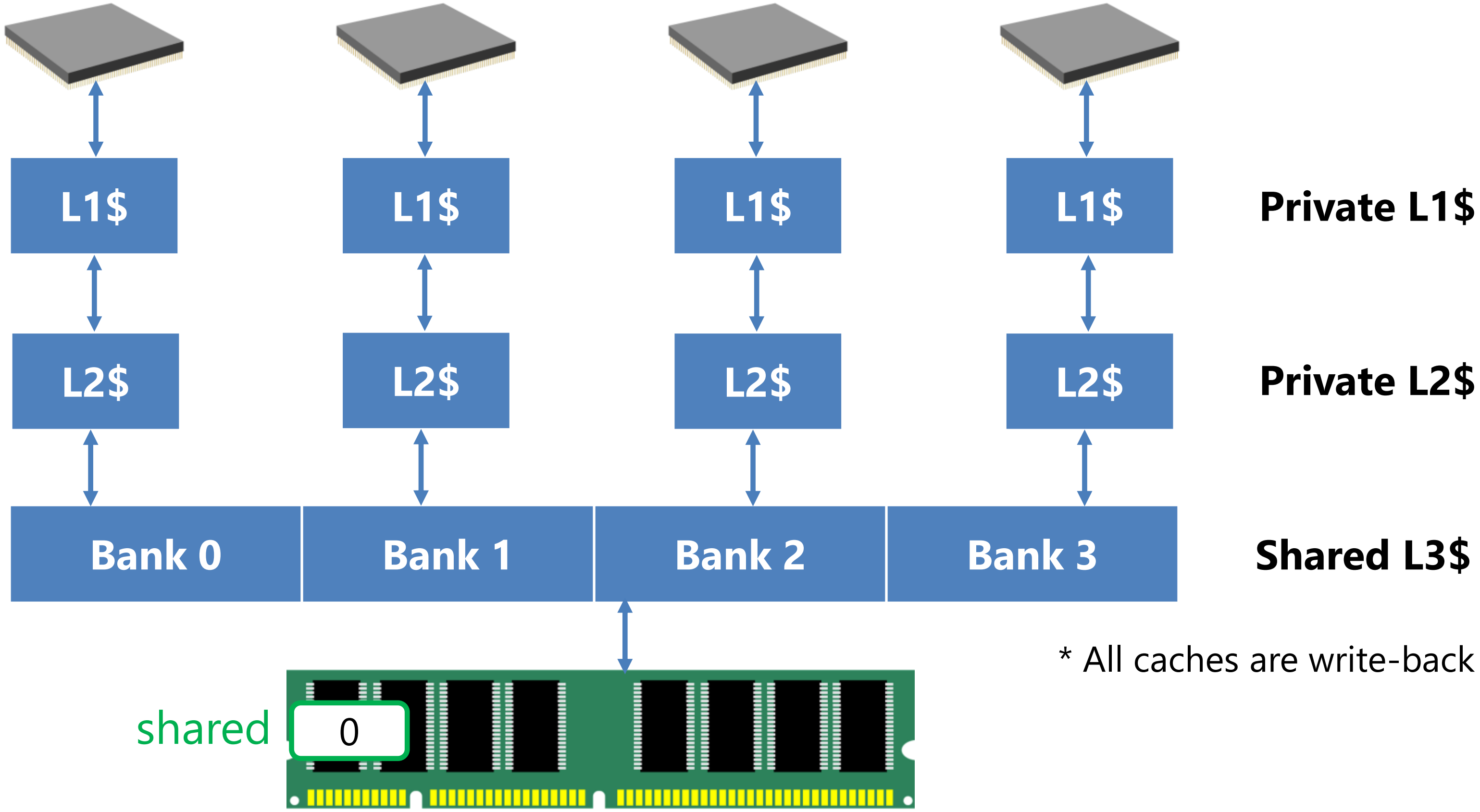
```
pthread_mutex_t lock;
int shared = 0;
void *add(void *unused) {
  for(int i=0; i < 1000000; i++) {
      pthread_mutex_lock(&lock);
      shared++;
      pthread_mutex_unlock(&lock);
  }
  return NULL;
}
int main() {
  …
}
```

```
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!

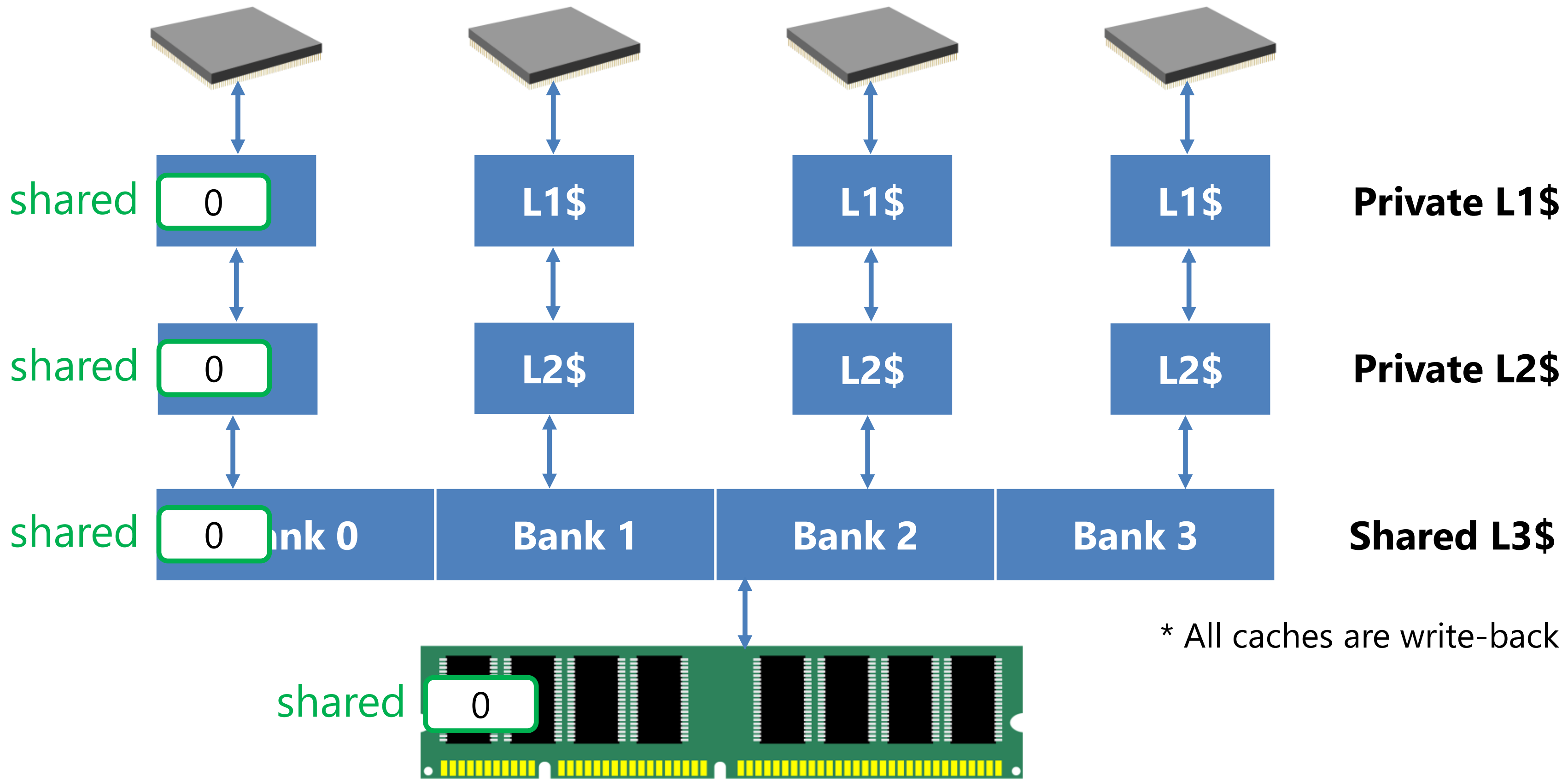
- What happens if caches sit in between processors and memory?



* All caches are write-back

Caching also does replication!

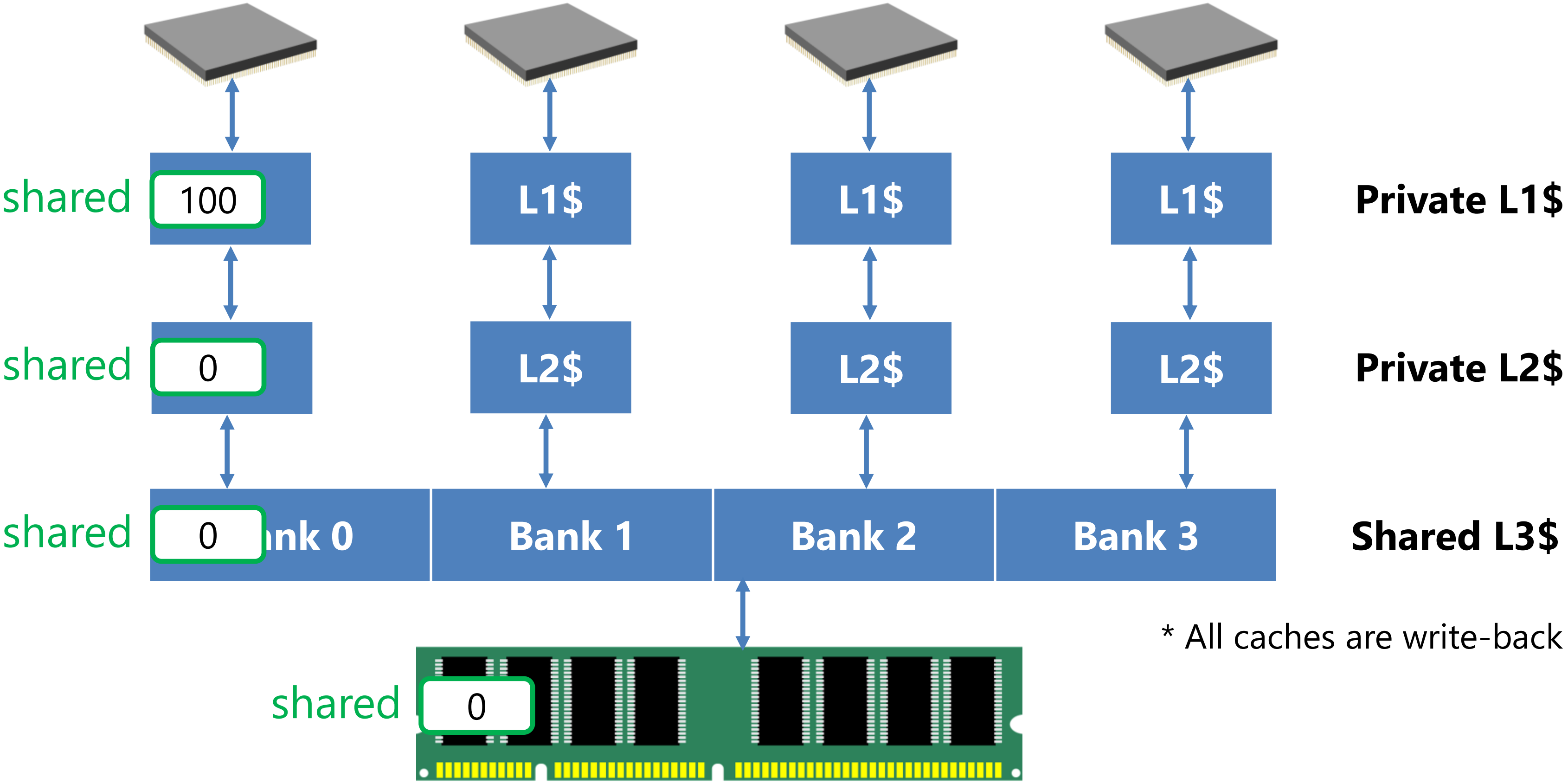
- Let's say CPU 0 first fetches `shared` for incrementing



* All caches are write-back

Caching also does replication!

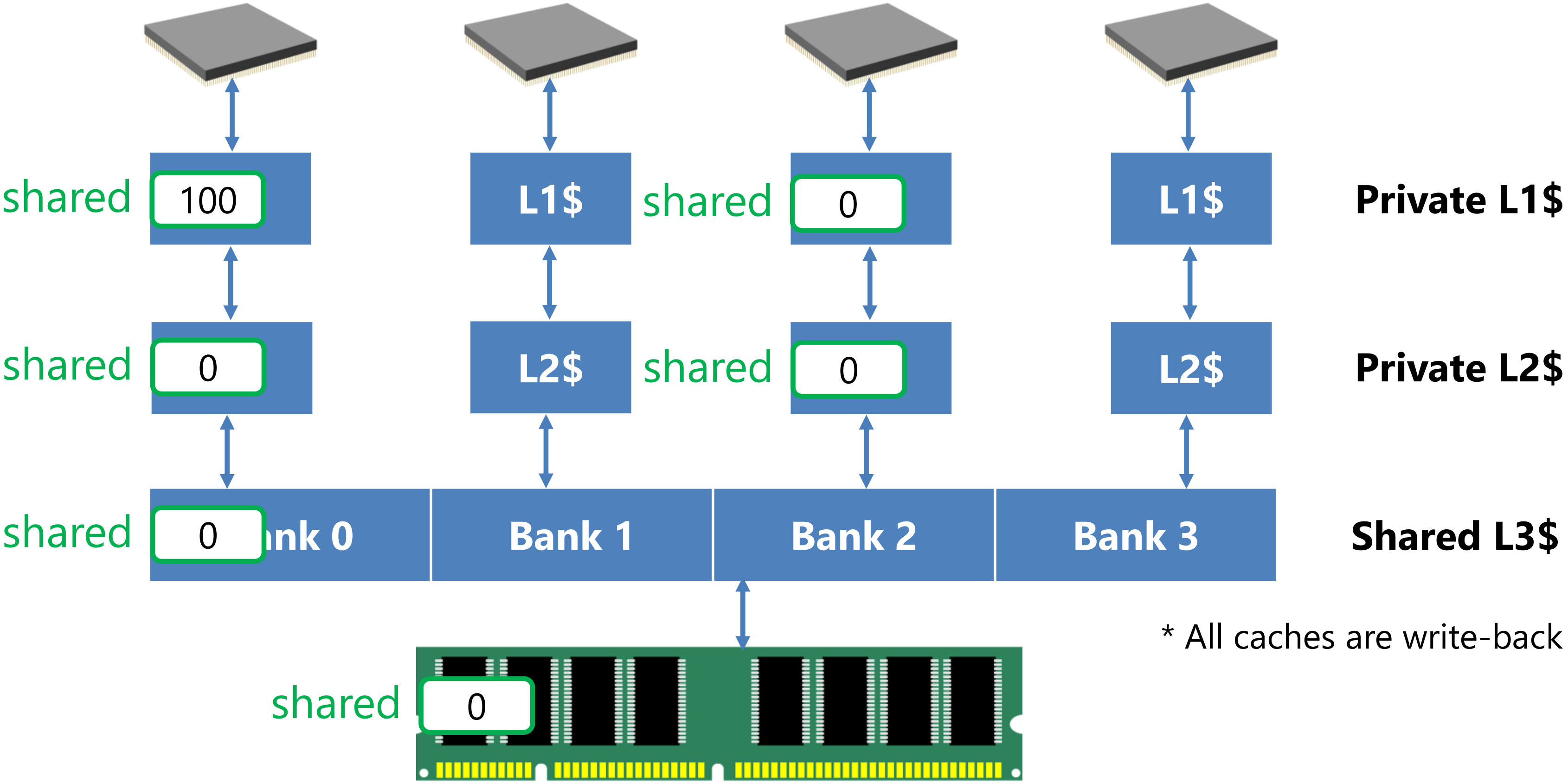
- Then CPU 0 increments `shared` 100 times to 100



* All caches are write-back

Caching also does replication!

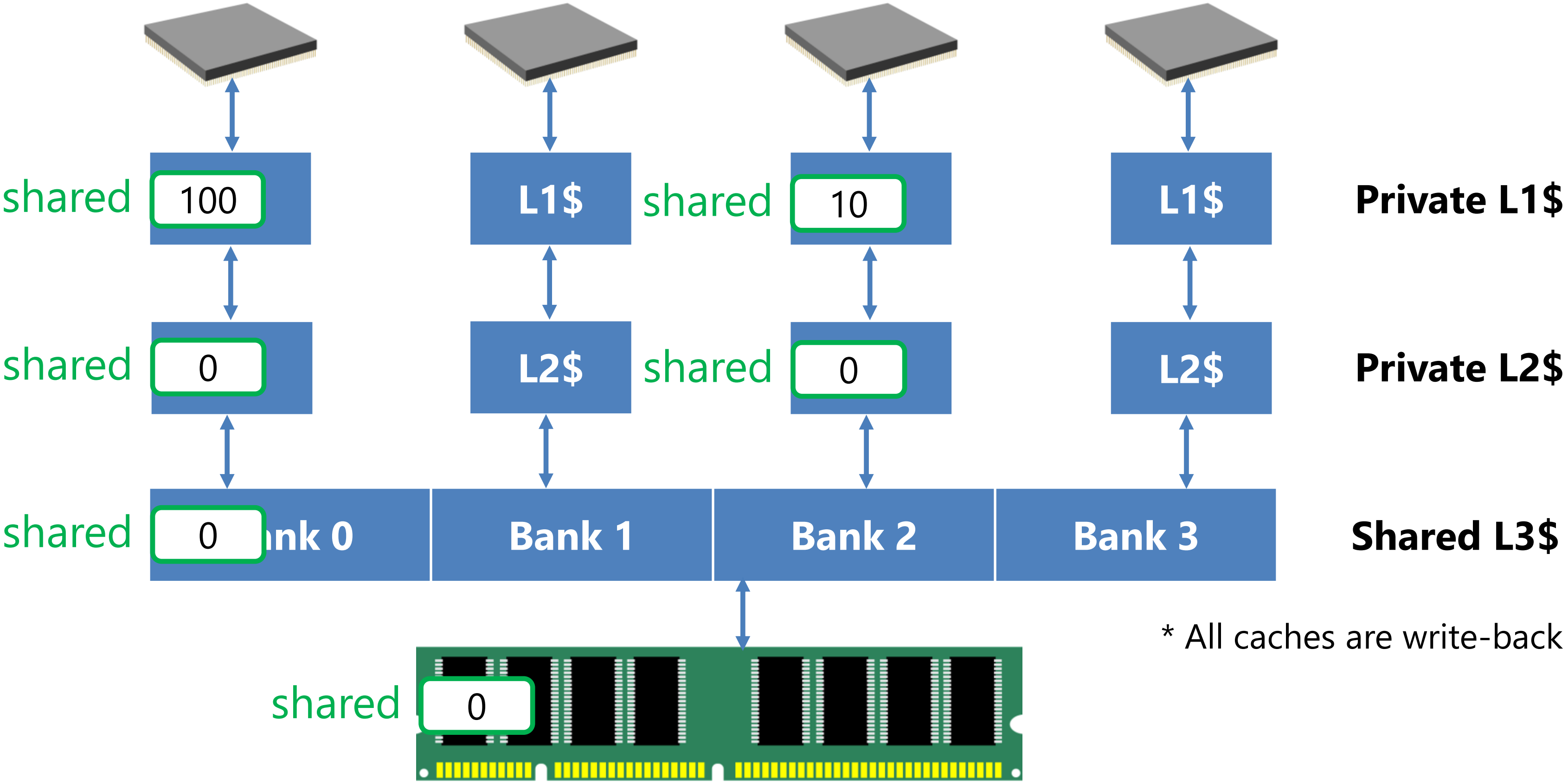
- Then CPU 2 gets hold of the mutex and fetches `shared` from L3



* All caches are write-back

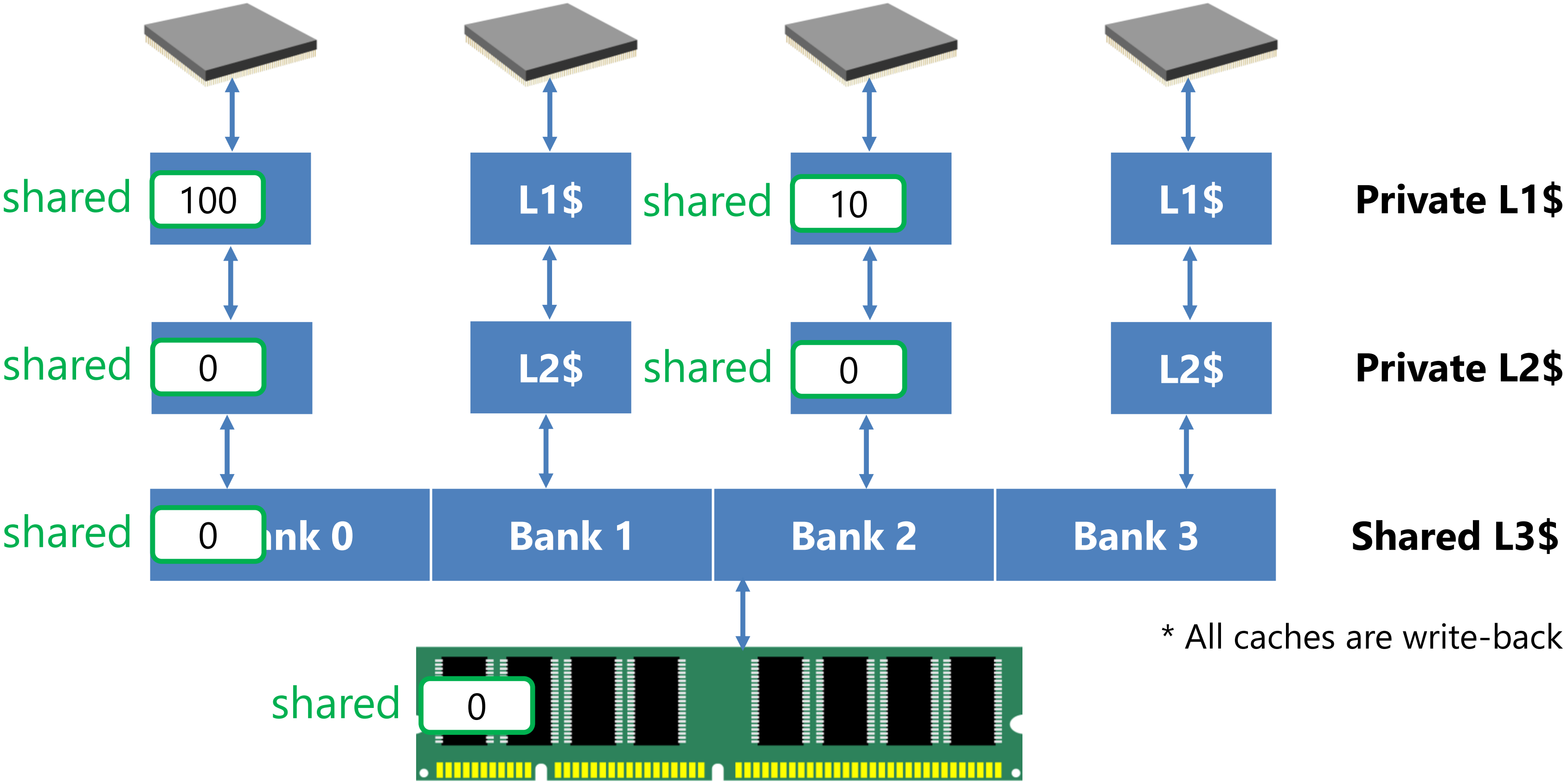
Caching also does replication!

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



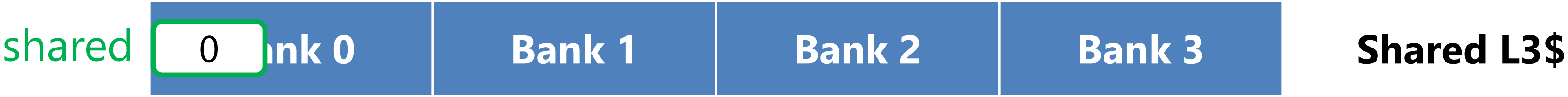
Caching also does replication!

- 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!

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

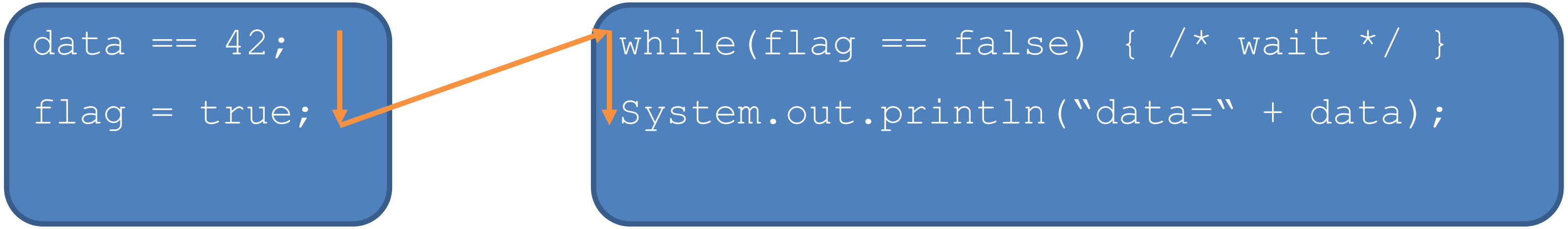
- Each ISA has a different definition of what that “view” means
  - **Memory consistency model**: definition of what that “view” is

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

How Memory Consistent Model affects correctness

- Initially, `data == 0`, `flag == false`.

Producer Thread **Consumer Thread**



```
data == 42;
flag = true;
```

```
while(flag == false) { /* wait */ }
System.out.println(“data=” + data);
```

Q) What do you expect the value of `data` will be when it gets printed?

A) Most people will say 42 because that is the logical ordering.

But is it? Not always. There are situations where `data` is still 0!

Scenario 1: Stores arrive out-of-order

- Initially, `data == 0`, `flag == false`.

Producer Thread

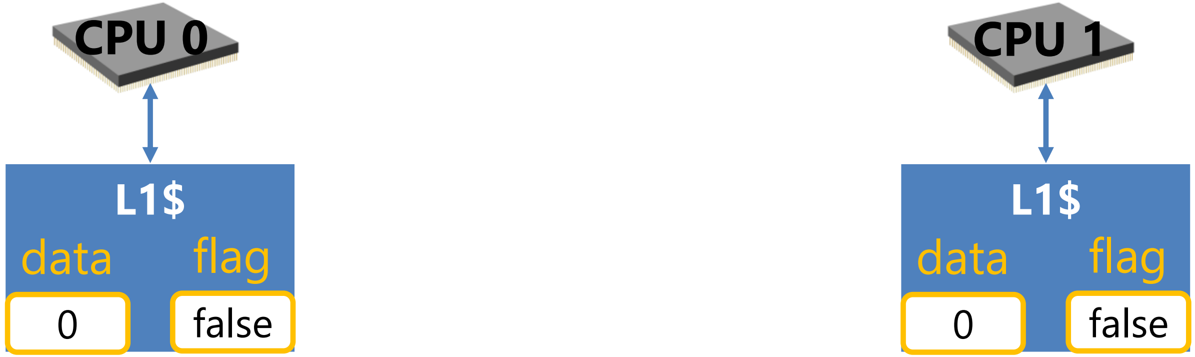
```
data == 42;

flag = true;
```

Consumer Thread

```
while(flag == false) { /* wait */ }

System.out.println(“data=” + data);
```



Let’s assume initially both data and flag are cached in each CPU’s L1 caches.

Scenario 1: Stores arrive out-of-order

- Initially, `data == 0`, `flag == false`.

Producer Thread

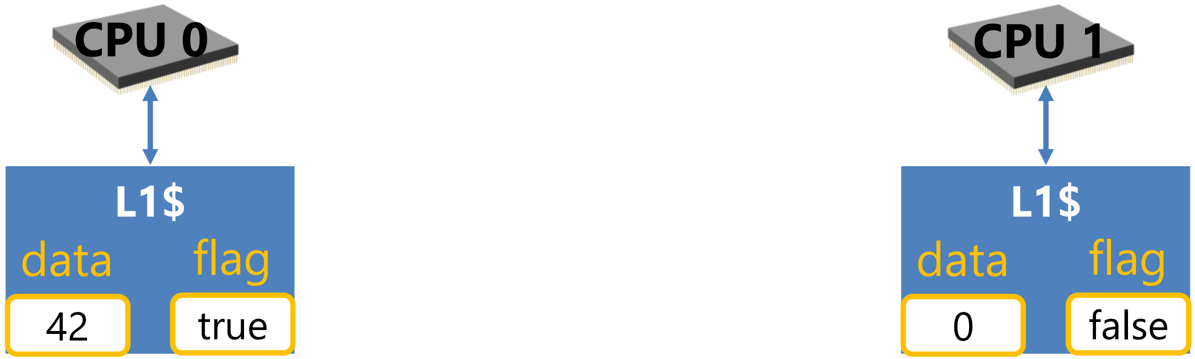
```
data == 42;

flag = true;
```

Consumer Thread

```
while(flag == false) { /* wait */ }

System.out.println("data=" + data);
```



CPU 0 updates both `data` and `flag` to `42` and `true`.

Scenario 1: Stores arrive out-of-order

- Initially, `data == 0`, `flag == false`.

Producer Thread

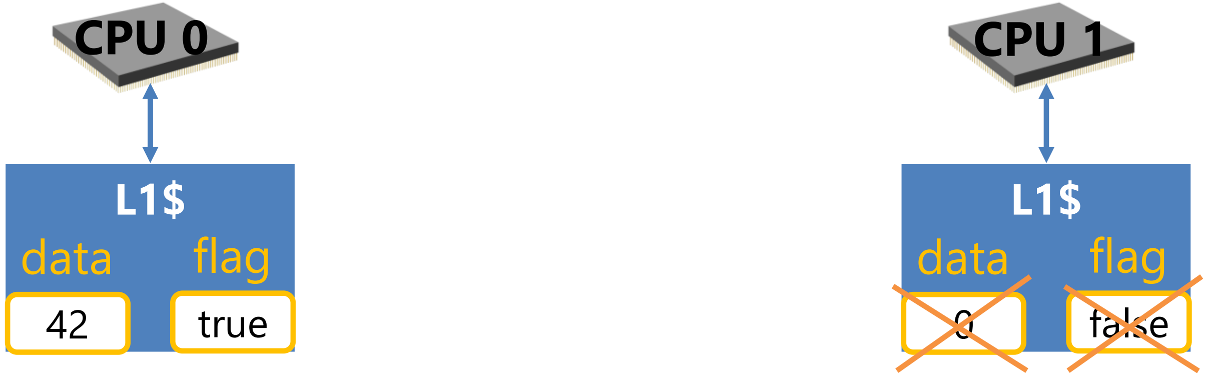
```
data == 42;

flag = true;
```

Consumer Thread

```
while(flag == false) { /* wait */ }

System.out.println("data=" + data);
```



Now the cached values in CPU 1 are stale and need to be **invalidated**.

Invalidation: act of marking a cache block with stale data invalid.

Scenario 1: Stores arrive out-of-order

- Initially, `data == 0`, `flag == false`.

Producer Thread

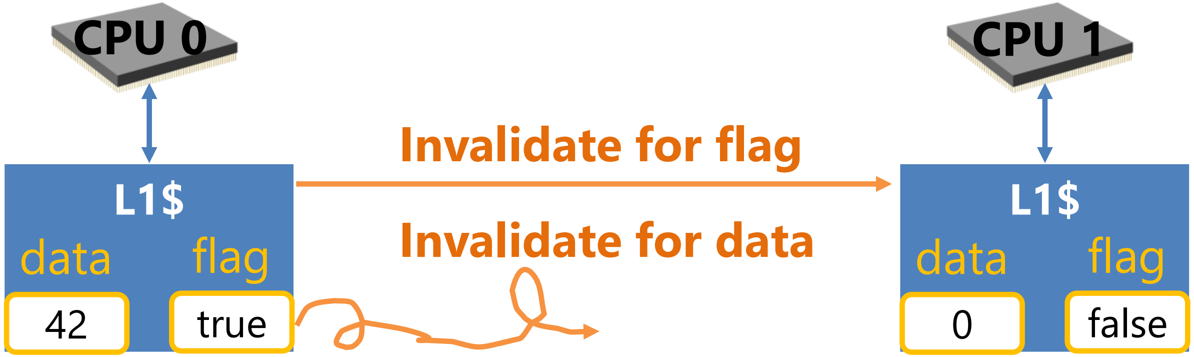
```
data == 42;

flag = true;
```

Consumer Thread

```
while(flag == false) { /* wait */ }

System.out.println("data=" + data);
```



The invalidate messages travel through a network and may arrive out-of-order.
Let's say invalidate for flag arrives first to CPU 1 and marks flag invalid.

Scenario 1: Stores arrive out-of-order

- Initially, `data == 0`, `flag == false`.

Producer Thread

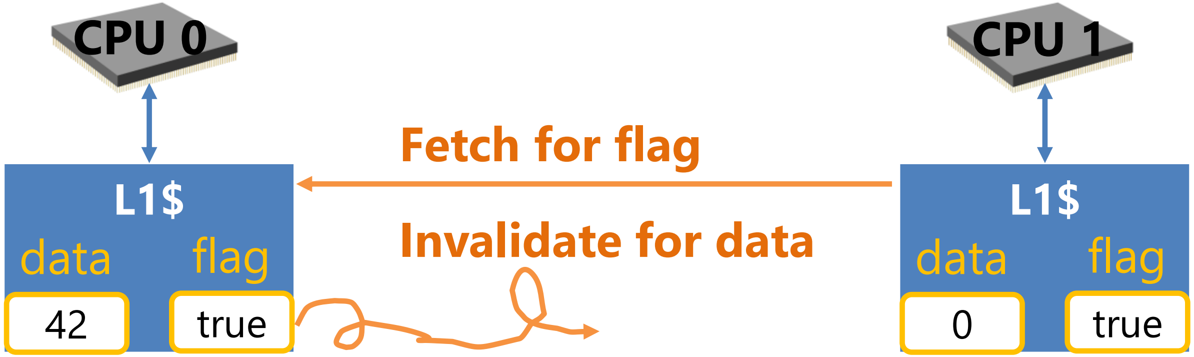
```
data == 42;

flag = true;
```

Consumer Thread

```
while(flag == false) { /* wait */ }

System.out.println("data=" + data);
```



CPU 1 fetches updated flag from CPU 0 when comparing `flag == false`.

Invalidate for data is still traveling through the network.

Scenario 1: Stores arrive out-of-order

- Initially, `data == 0`, `flag == false`.

Producer Thread

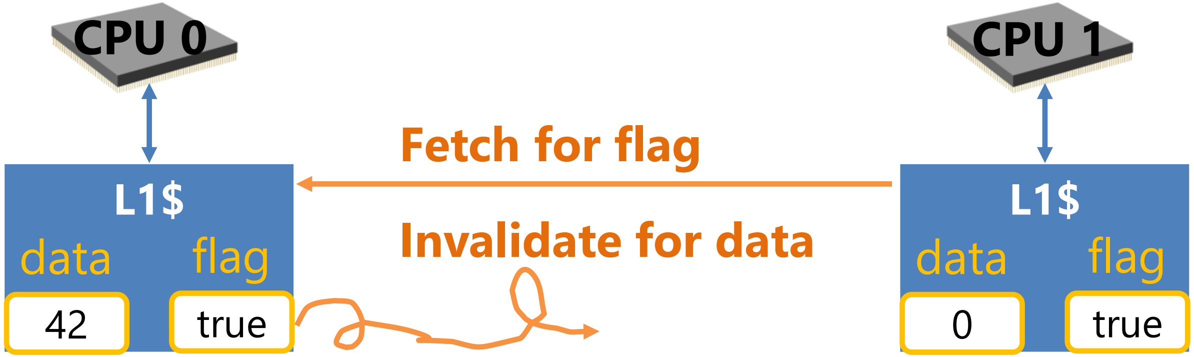
```
data == 42;

flag = true;
```

Consumer Thread

```
while(flag == false) { /* wait */ }

System.out.println("data=" + data);
```



Since `flag` is `true`, CPU 1 breaks out of while loop and prints `data`.

`data=0` gets printed!

Scenario 2: Loads perform out-of-order

- Initially, `data == 0`, `flag == false`.

Producer Thread

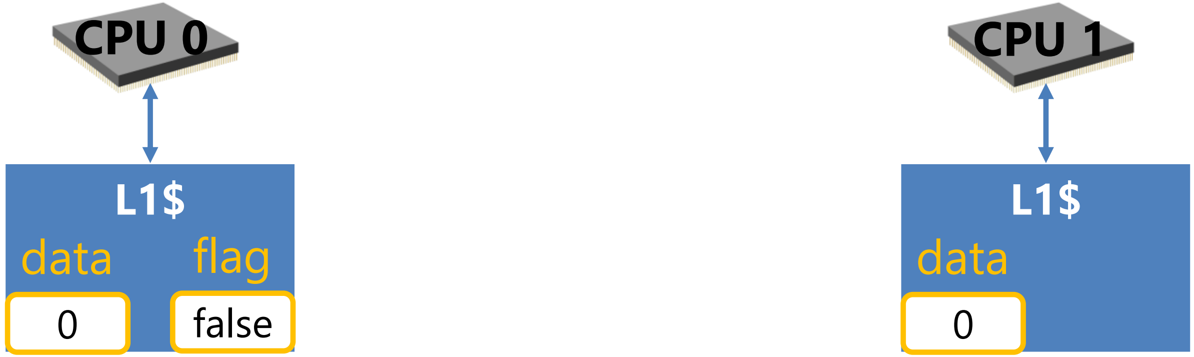
```
data == 42;

flag = true;
```

Consumer Thread

```
while(flag == false) { /* wait */ }

System.out.println(“data=” + data);
```



Let’s assume now `flag` is not cached in CPU 1.

CPU 1 suffers a cache miss on `flag` when it compares `flag == false`.

Scenario 2: Loads perform out-of-order

- Initially, `data == 0`, `flag == false`.

Producer Thread

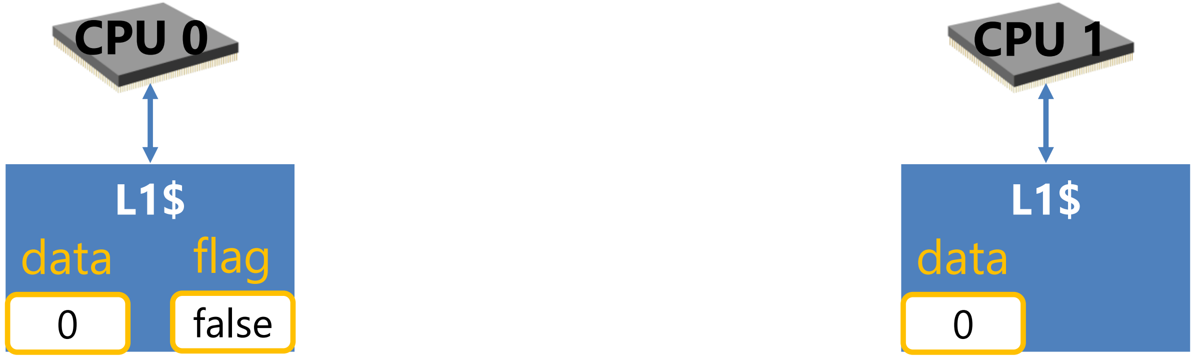
```
data == 42;

flag = true;
```

Consumer Thread

```
while(flag == false) { /* wait */ }

System.out.println("data=" + data);
```



Instruction Queue
lw r1, flag **(miss)**
beq r1, $zero, _loop
lw r2, data **(hit)**
call println on r2

Instead of stalling, CPU 1 predicts the branch not taken and issues lw r2, data. Now, `r2 == 0.` (Unless pipeline flushes due to branch misprediction.)

Scenario 2: Loads perform out-of-order

- Initially, `data == 0`, `flag == false`.

Producer Thread

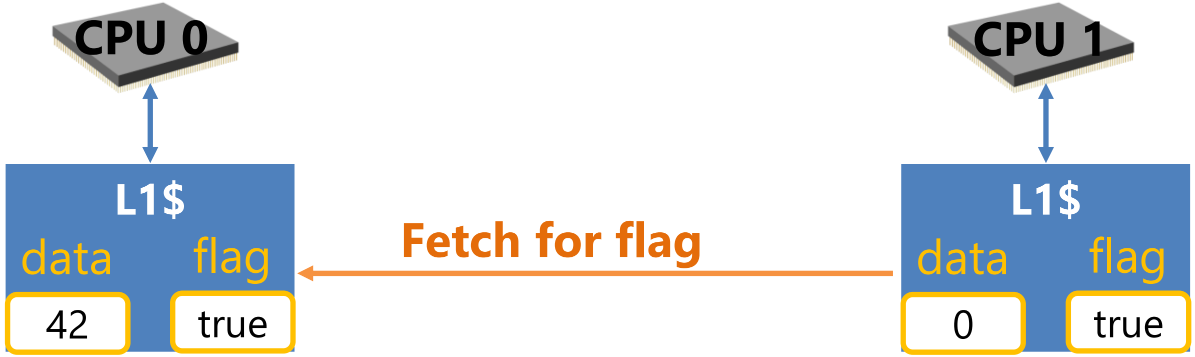
```
data == 42;

flag = true;
```

Consumer Thread

```
while(flag == false) { /* wait */ }

System.out.println(“data=” + data);
```



Instruction Queue
lw r1, flag **(miss)**
beq r1, $zero, _loop
lw r2, data **(hit)**
call println on r2

Now let’s say CPU 0 updates data and flag before the fetch for flag arrives.

Now, lw r1, flag completes, allowing beq r1, $zero, _loop to issue (with `r1 == true`)

Scenario 2: Loads perform out-of-order

- Initially, `data == 0`, `flag == false`.

Producer Thread

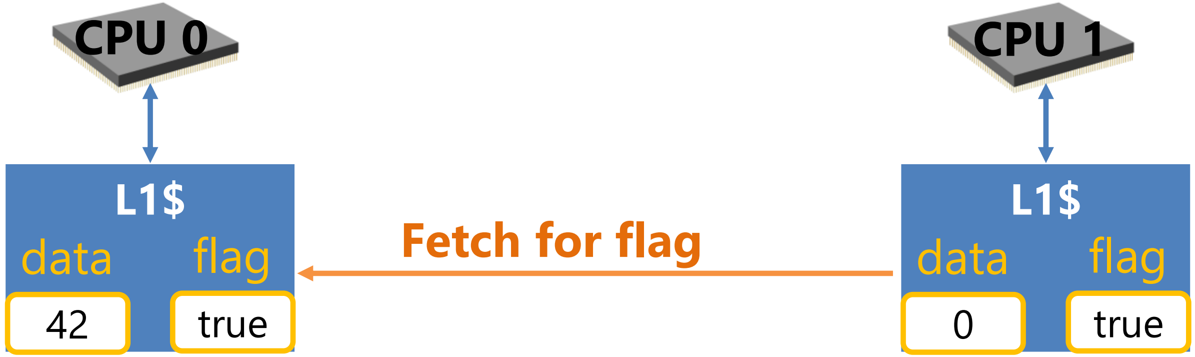
```
data == 42;

flag = true;
```

Consumer Thread

```
while(flag == false) { /* wait */ }

System.out.println("data=" + data);
```



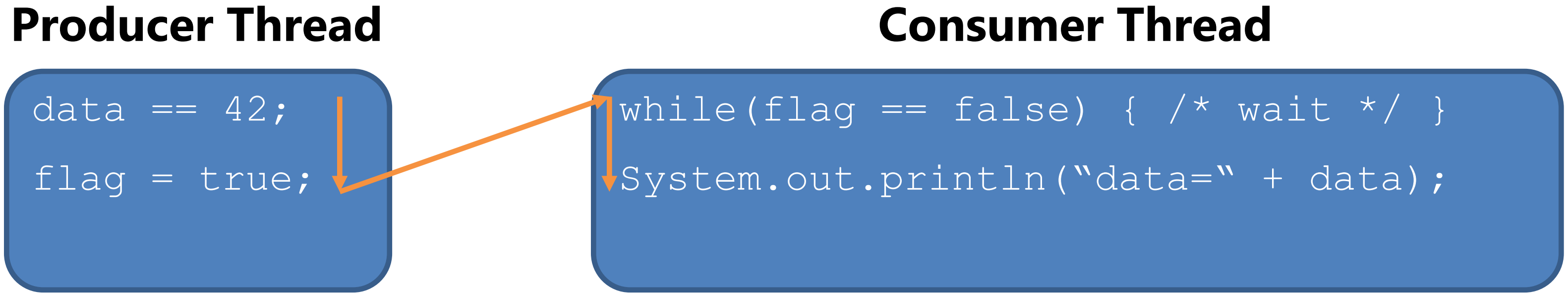
Instruction Queue
lw r1, flag **(miss)**
beq r1, $zero, _loop
lw r2, data **(hit)**
call println on r2

Since `r1 == true`, that validates the not-taken prediction for the branch.

Since `r1 == 0`, the println outputs `data=0`!

Memory Consistency Models are often very lax

- Initially, `data == 0`, `flag == false`.



Producer Thread

```
data == 42;
flag = true;
```

Consumer Thread

```
while(flag == false) { /* wait */ }
System.out.println(“data=” + data);
```

- A memory consistency model where above ordering is guaranteed is called, Sequential Consistency (SC): Instructions appear to execute sequentially.
- Real models allow many other orderings to allow optimizations:
  - Write buffers that allow multiple stores to be pending and perform out-of-order
  - Instruction queues that allow loads and other instructions to perform out-of-order
  - Compiler optimizations to reschedule stores and loads out-of-order
- Intel, ARM, Java Virtual Machine all have relaxed memory consistency models
- Moral: **never do custom synchronization** unless you know what you are doing!

Memory Consistency Models are often very lax

- Initially, `data == 0`, `flag == false`.

Producer Thread

```
data == 42;

flag = true;
```

Consumer Thread

```
while(flag == false) { /* wait */ }

System.out.println(“data=“ + data);
```

- Regardless of memory consistency model, they all agree on one thing: that values of `data` and `flag` must be made coherent *eventually*.
  - They only disagree on when that eventually is.
- This property is called **cache coherence**.

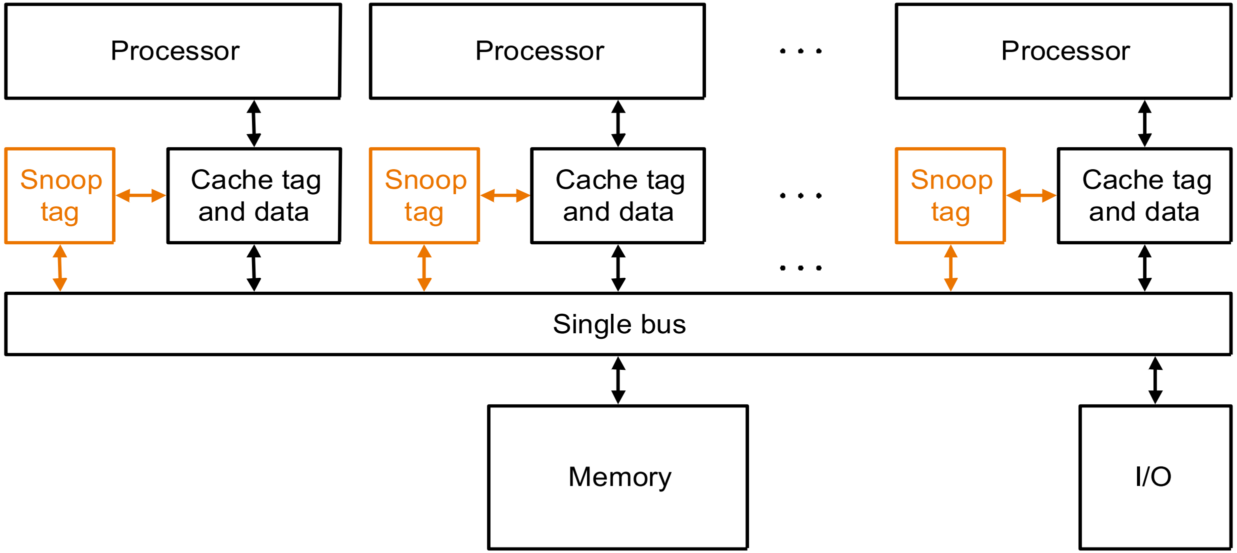
Implementing Cache Coherence

- 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.

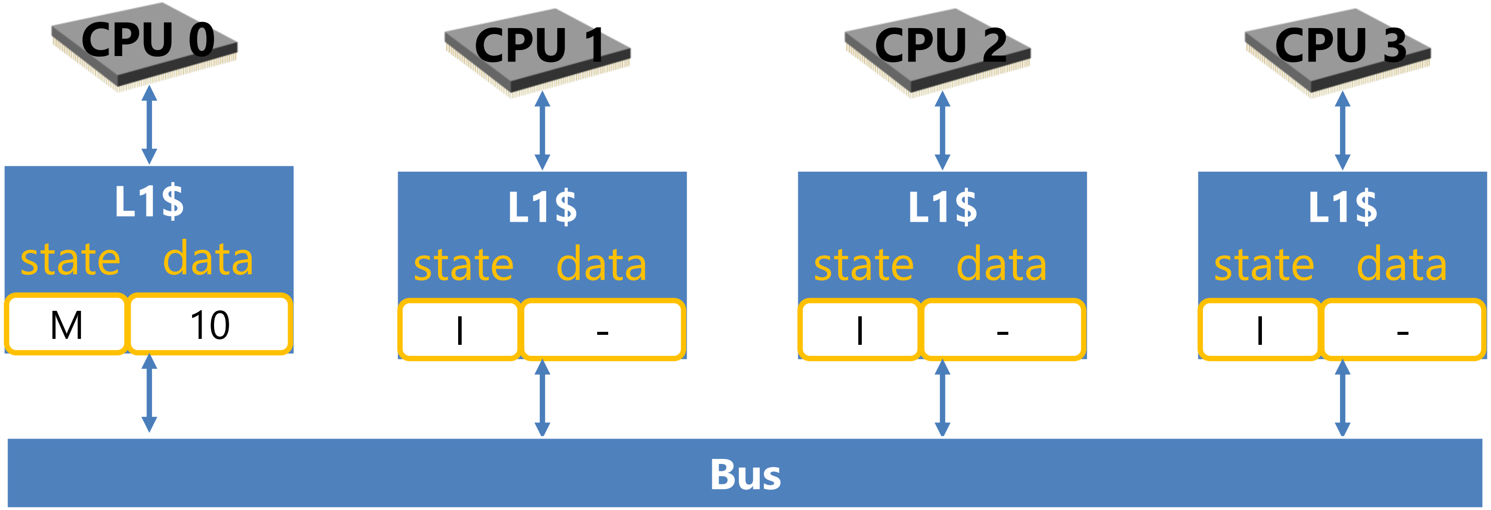
MSI Snoopy Cache Coherence Protocol



- 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

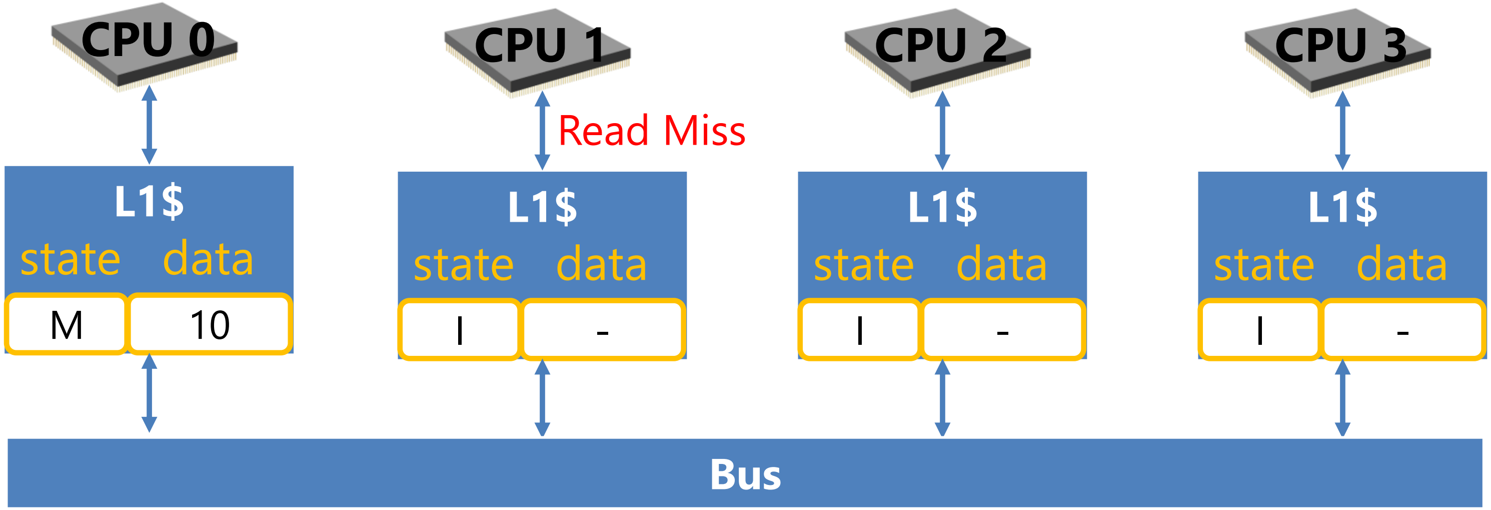
University of Pittsburgh

Scenario 1: Read Snooped on Bus



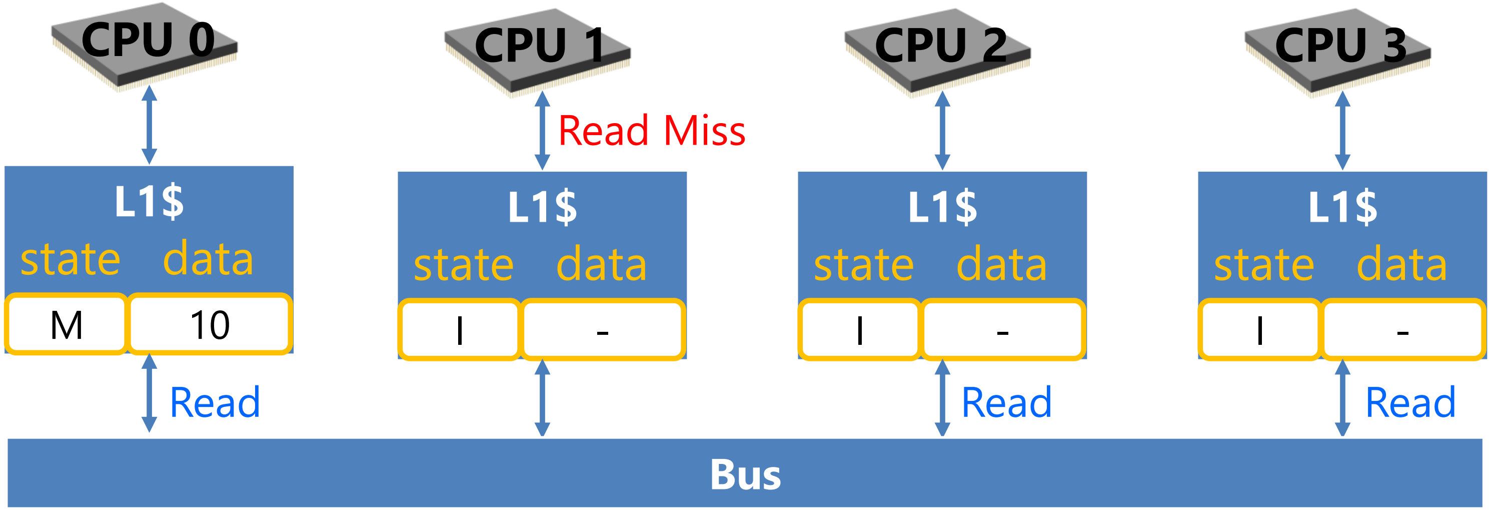
1. CPU 0: line in Modified state, CPU 1~3: line in Invalid state

Scenario 1: Read Snooped on Bus



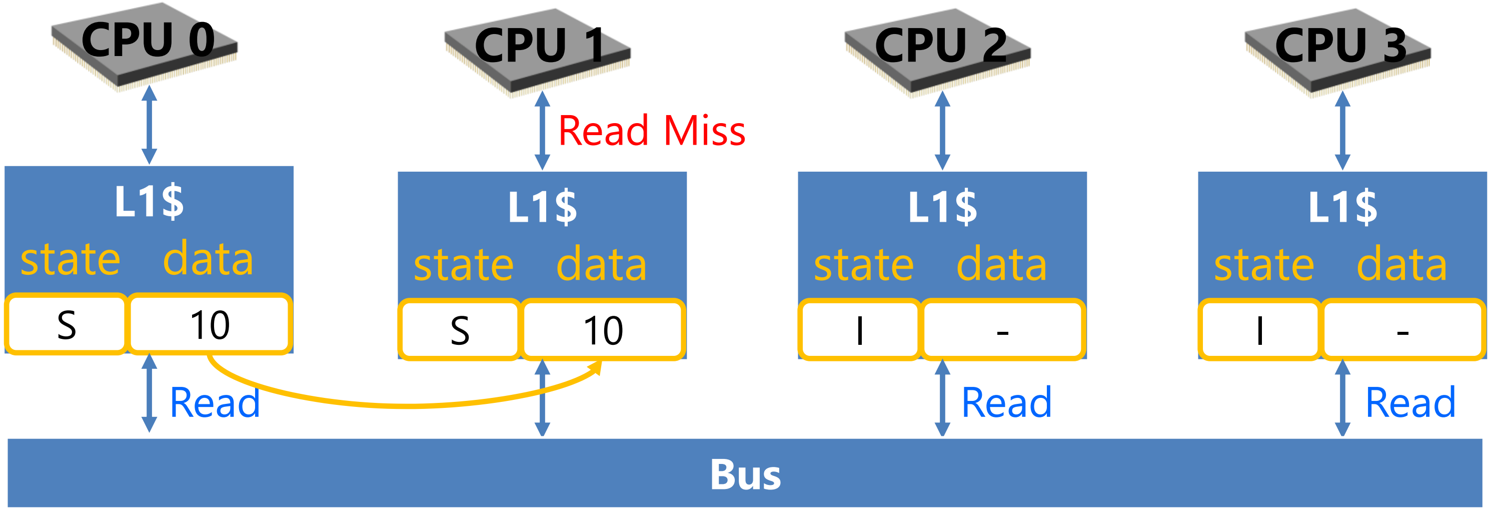
1. CPU 0: line in Modified state, CPU 1~3: line in Invalid state
2. CPU 1 reads line, causing a Read Miss

Scenario 1: Read Snooped on Bus



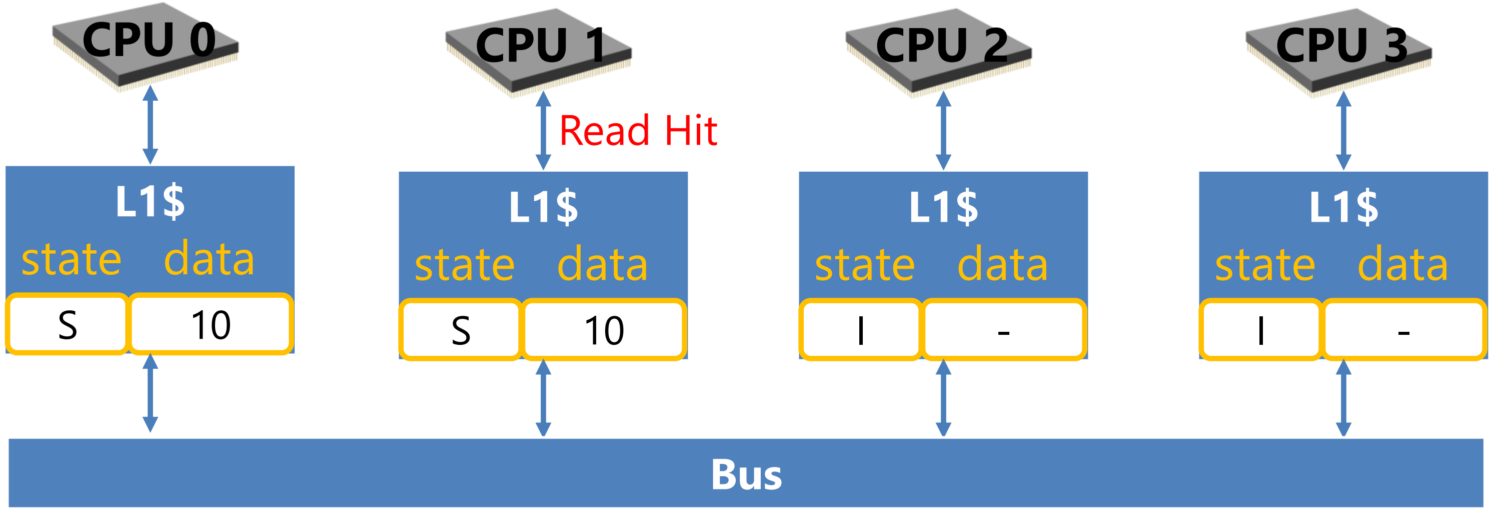
1. CPU 0: line in Modified state, CPU 1~3: line in Invalid state
2. CPU 1 reads line, causing a Read Miss
3. A Read message is broadcast on the Bus

Scenario 1: Read Snooped on Bus



1. CPU 0: line in Modified state, CPU 1~3: line in Invalid state
2. CPU 1 reads line, causing a Read Miss
3. A Read message is broadcast on the Bus
4. CPU 0 snoops message and provides line to CPU1
   - Line in CPU 0 and CPU 1 both transition to Shared state

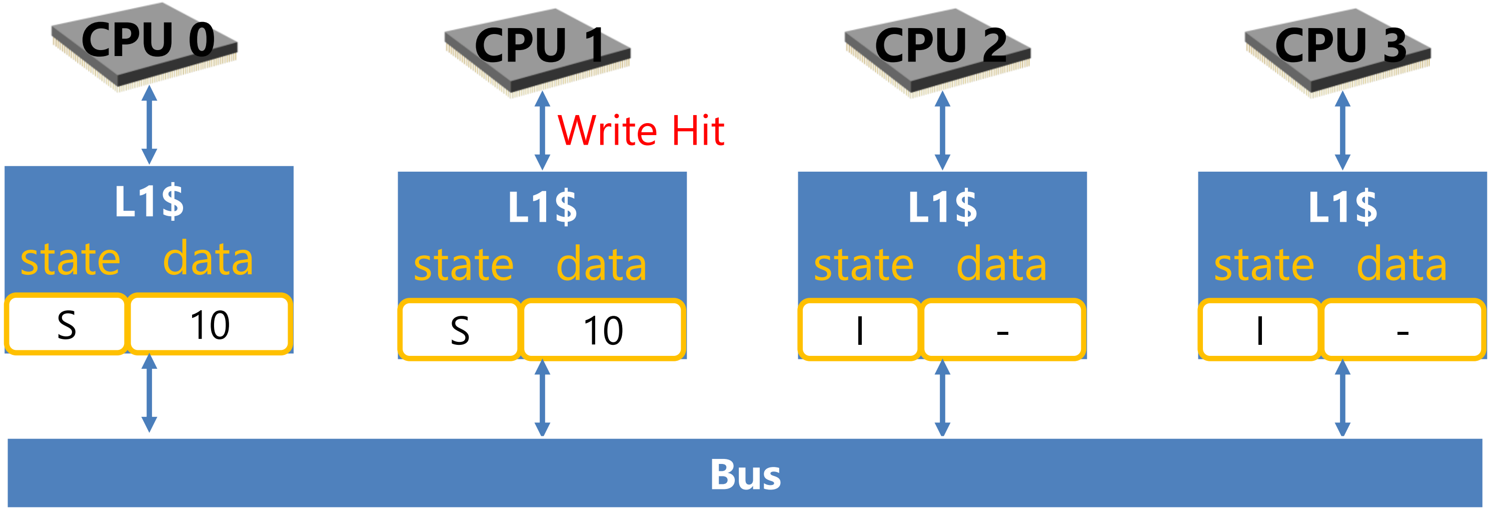
Scenario 1: Read Snooped on Bus



- Subsequent read hits on CPU 0 or CPU 1 don't generate messages

- Only read misses generate messages, not read hits
  → Reduces bus bandwidth pressure since misses are rare!

Scenario 2: Invalidate Snooped on Bus



CPU 0

CPU 1

CPU 2

CPU 3

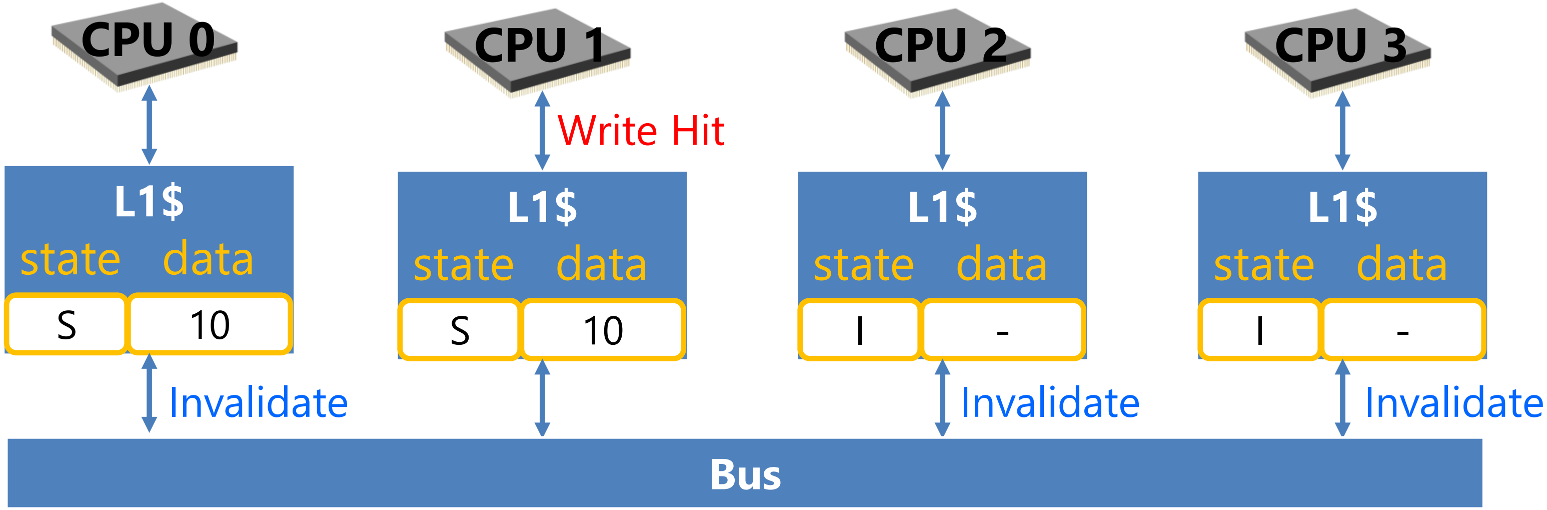
Write Hit

	CPU 0 L1$	CPU 1 L1$	CPU 2 L1$	CPU 3 L1$
state	S	S	I	I
data	10	10	-	-

Bus

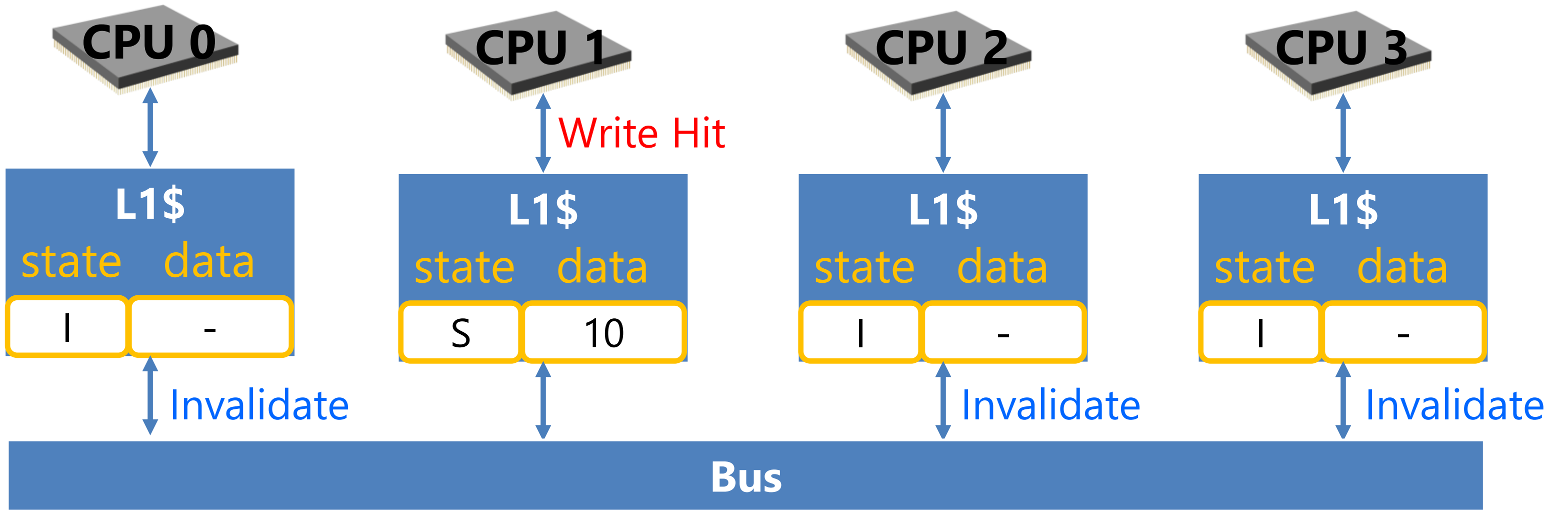
1. CPU 1 writes to line, causing a Write Hit

Scenario 2: Invalidate Snooped on Bus



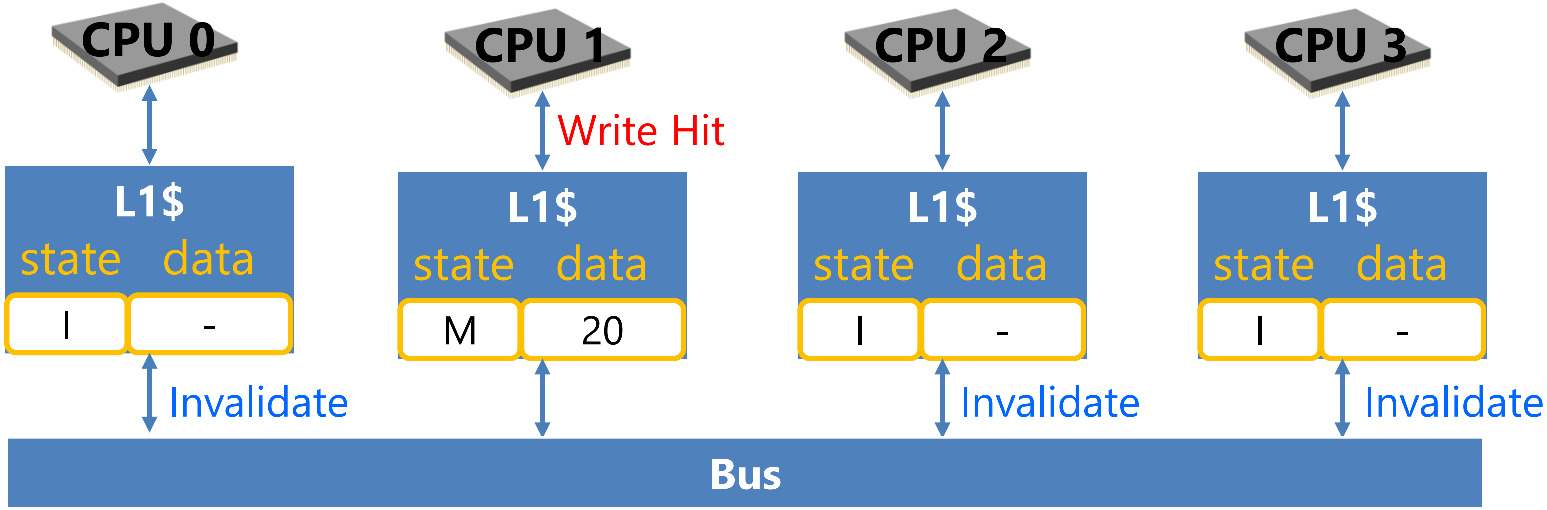
1. CPU 1 writes to line, causing a Write Hit
2. An Invalidate message is broadcast on the Bus
   - To remove all copies of the line that may become inconsistent

Scenario 2: Invalidate Snooped on Bus



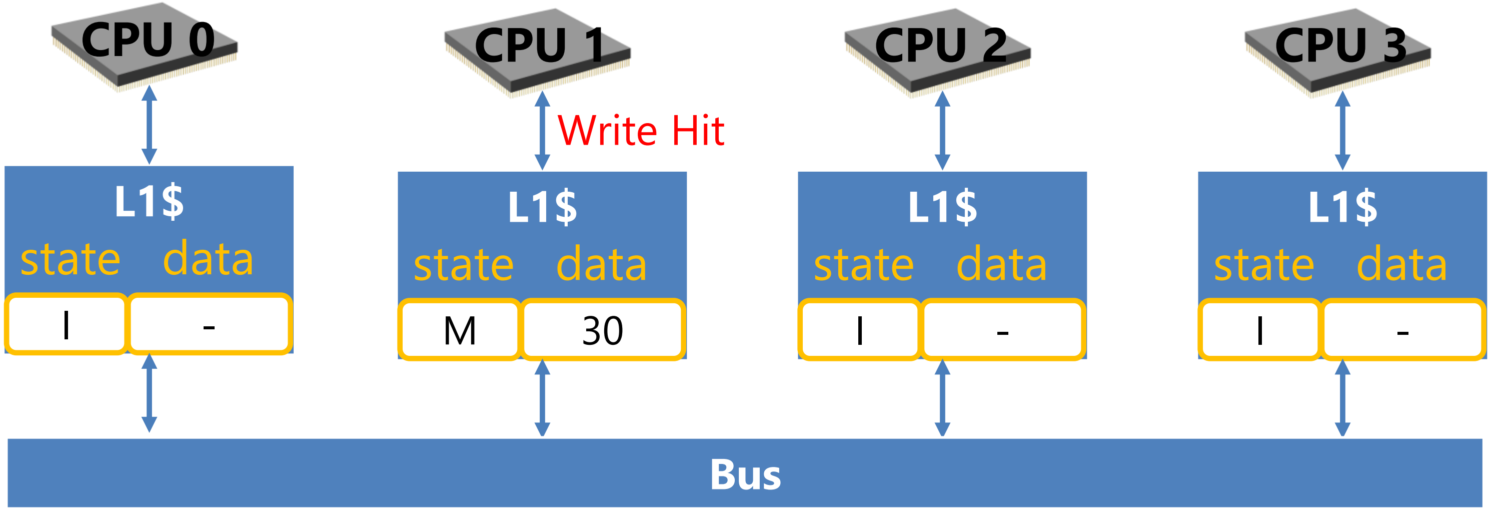
1. CPU 1 writes to line, causing a Write Hit
2. An Invalidate message is broadcast on the Bus
   - To remove all copies of the line that may become inconsistent
3. CPU 0 snoops message and invalidates its line

Scenario 2: Invalidate Snooped on Bus



1. CPU 1 writes to line, causing a Write Hit
2. An Invalidate message is broadcast on the Bus
   - To remove all copies of the line that may become inconsistent
3. CPU 0 snoops message and invalidates its line
4. Line in CPU 1 transitions to Modified state

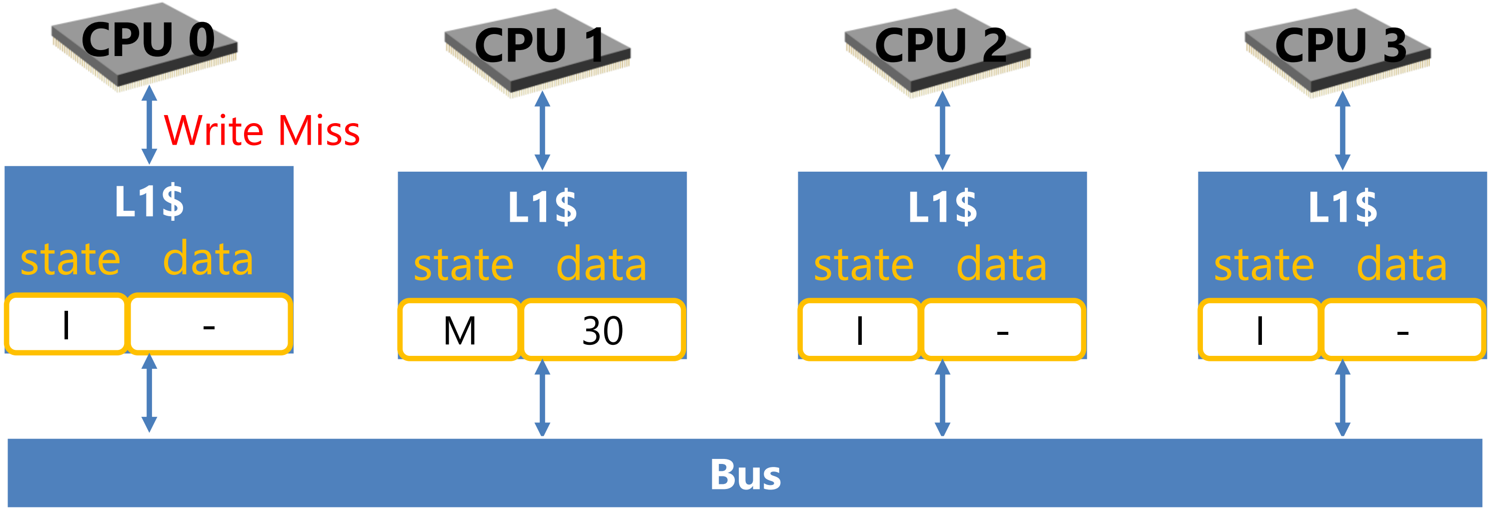
Scenario 2: Invalidate Snooped on Bus



- Subsequent write hits on CPU 1 don't generate messages

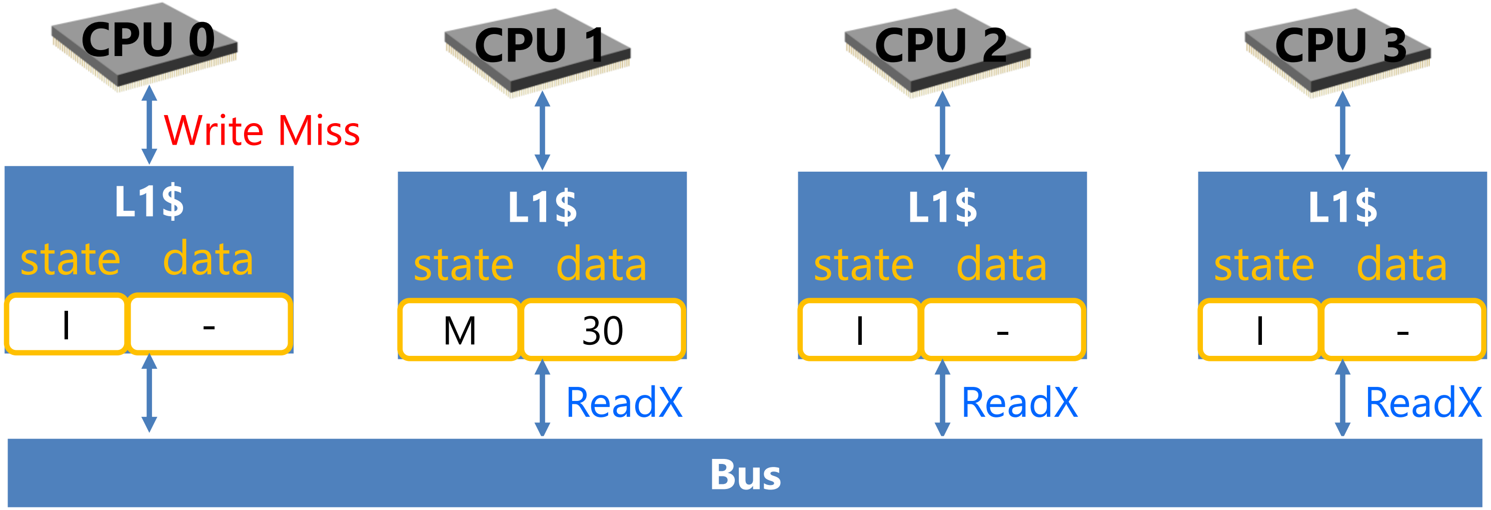
- Only write hit in Shared state generate messages
  → Reduces bandwidth since most write hits are to Modified state

Scenario 3: Read Exclusive Snooped on Bus



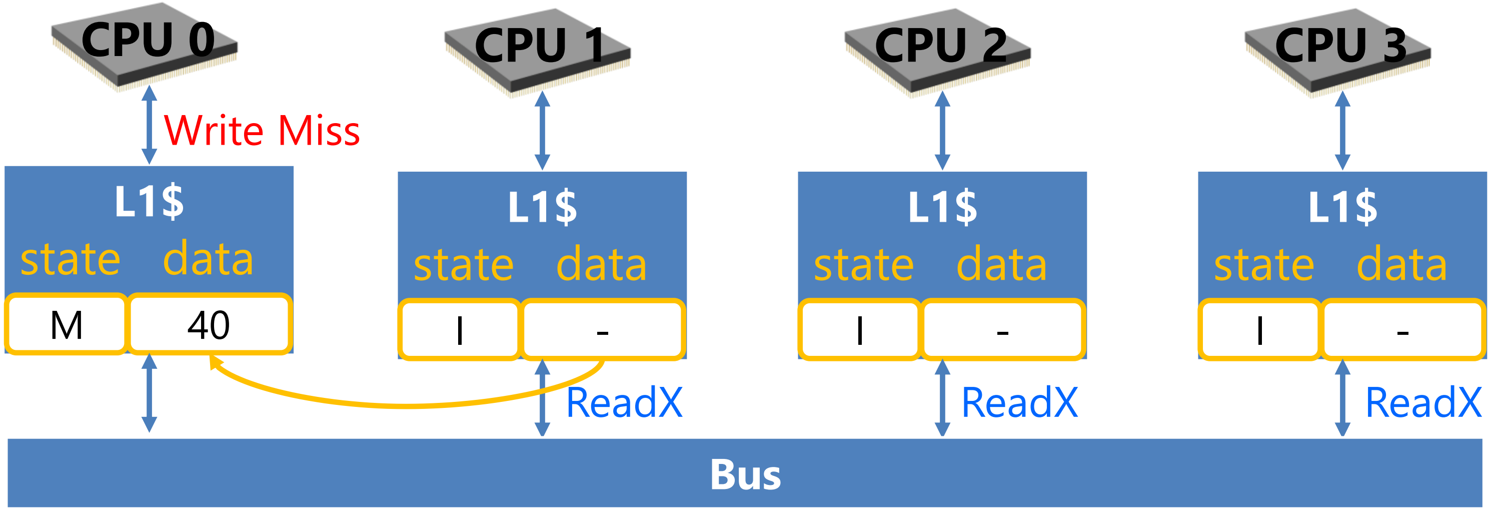
1. CPU 0 writes to line, causing a Write Miss

Scenario 3: Read Exclusive Snooped on Bus



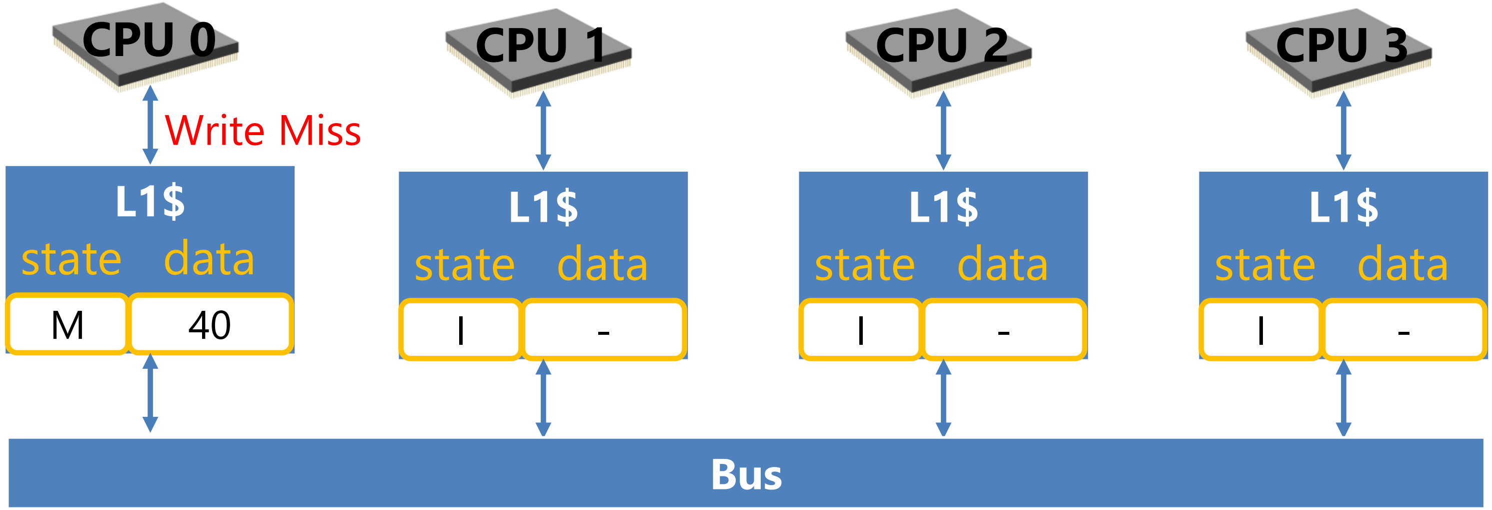
1. CPU 0 writes to line, causing a Write Miss
2. A ReadX (read exclusive) message is broadcast on the Bus
   - To read line into CPU 0 before writing to it (remember?)
   - To remove all copies of the line that may become inconsistent

Scenario 3: Read Exclusive Snooped on Bus



1. CPU 0 writes to line, causing a Write Miss
2. A ReadX (read exclusive) message is broadcast on the Bus
   - To read line into CPU 0 before writing to it (remember?)
   - To remove all copies of the line that may become inconsistent
3. CPU 1 snoops message and provides line to CPU 0
   - Line in CPU 1 is invalidated before line in CPU 0 is modified

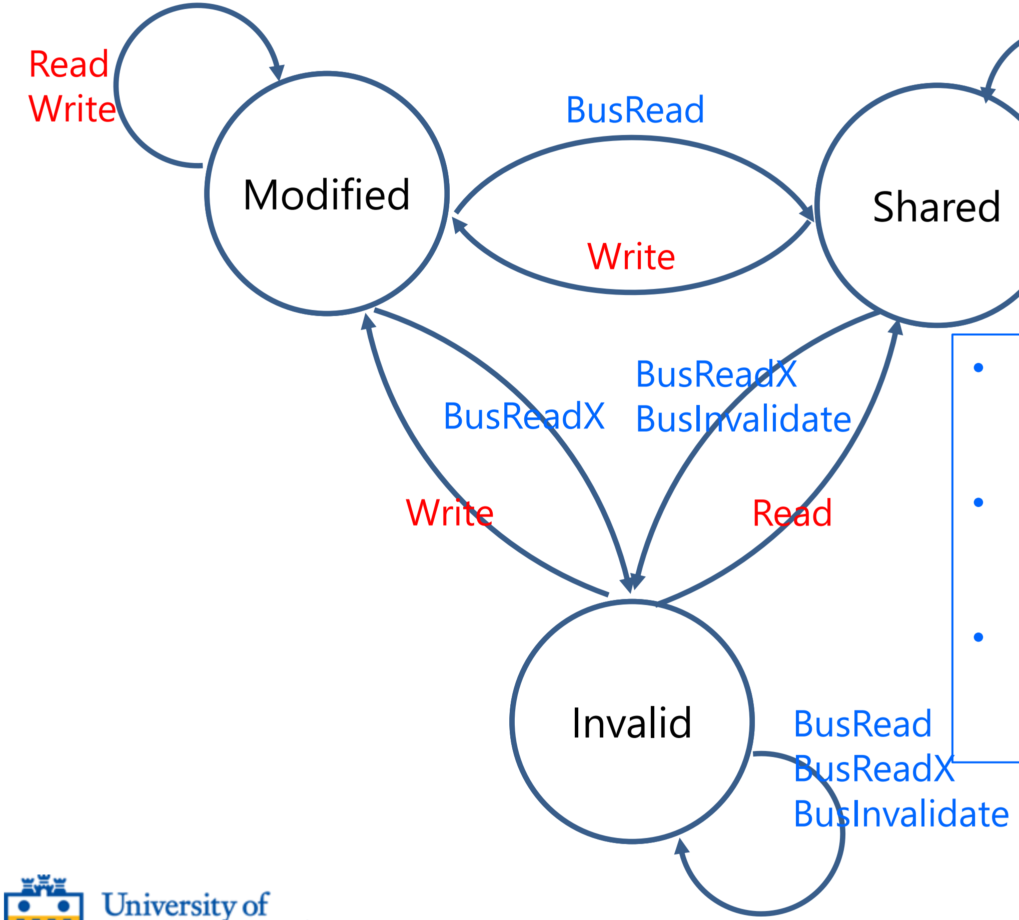
Scenario 3: Read Exclusive Snooped on Bus



- Subsequent write hits on CPU 0 don't generate messages
- Write miss generates message, but not subsequent write hits
  → Reduces bus bandwidth pressure since misses are rare!

Cache Controller FSM for MSI Protocol

- Processor activity in red, Bus activity in blue



- BusRead: Read request is snooped
  - Sent by another processor on a read miss
  - If Modified, transition to Shared state
- BusInvalidate: Invalidate request is snooped
  - Sent on a write hit on shared cache line
  - Action: Invalidate shared line (if it exists)
- BusReadX: Read exclusive request is snooped
  - Sent by another processor on a write miss
  - Action: Invalidate shared line (if it exists)

TLB Coherence

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?

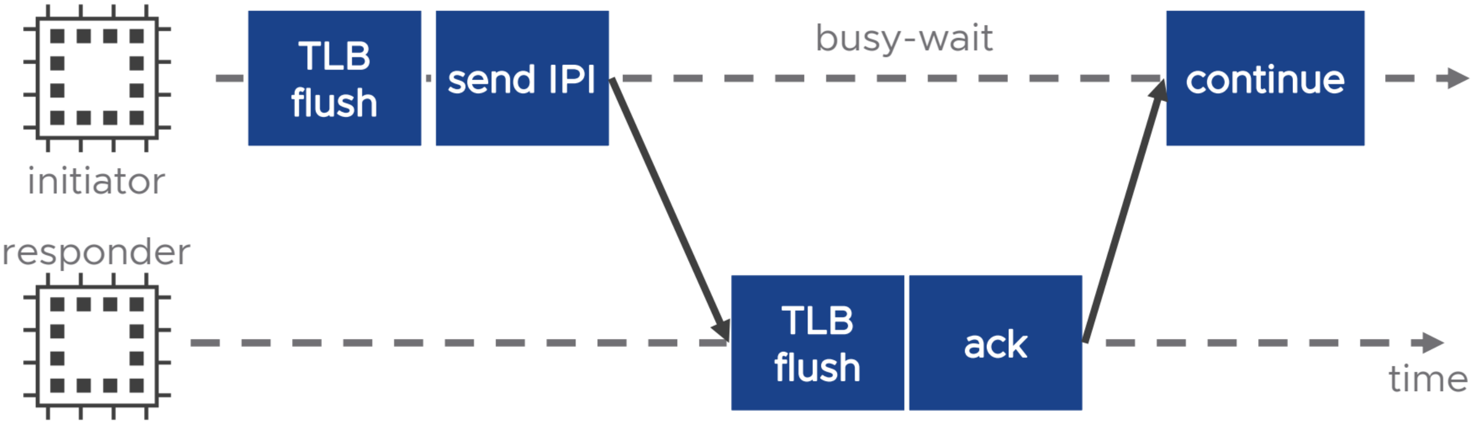
- Unfortunately, there is no hardware coherence for TLBs. ☹

- 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