Synchronization in cache-coherent architectures

Performance enhancement by reducing bus traffic

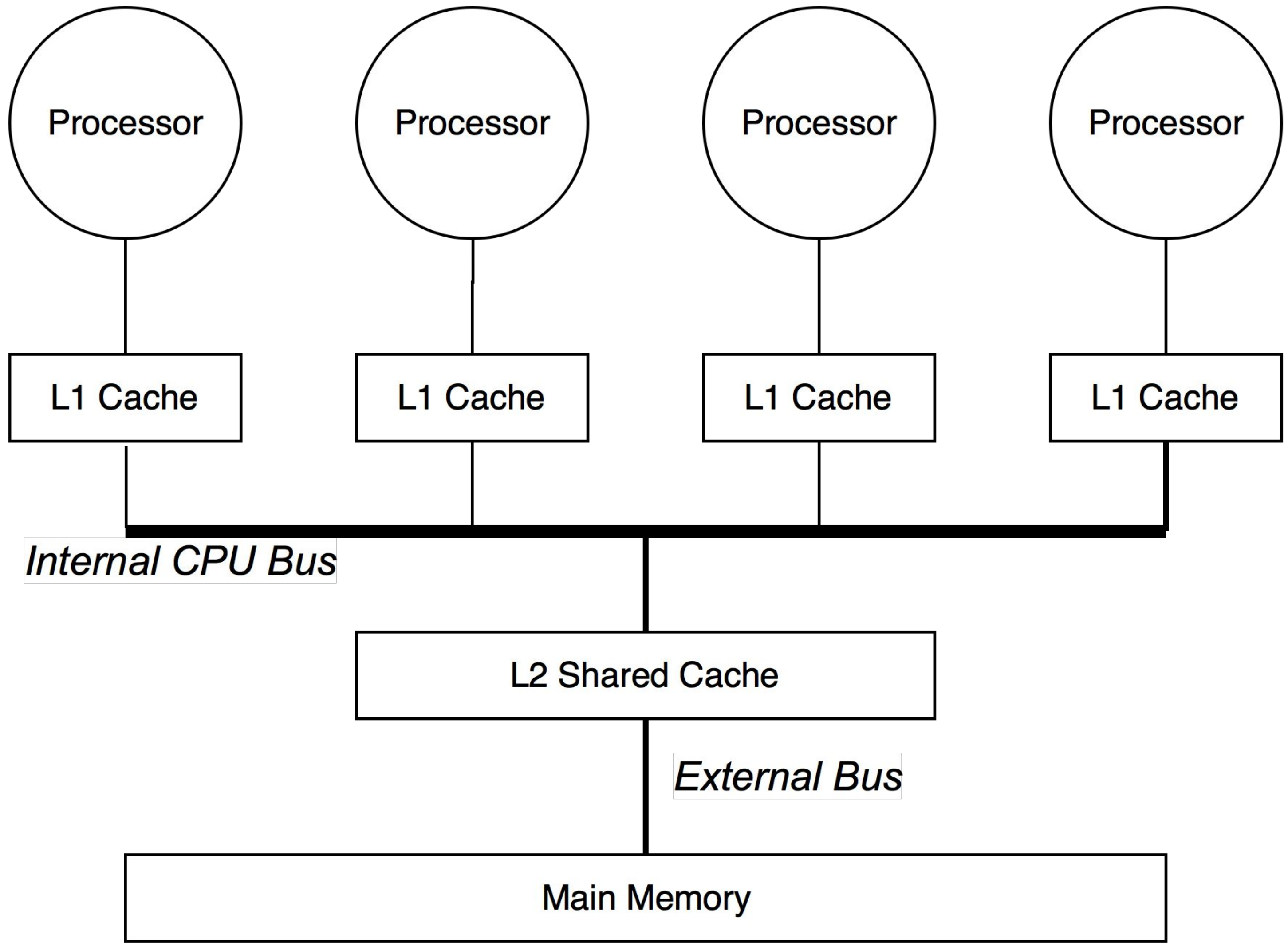
Lagani • Micera • Miliani

Outline

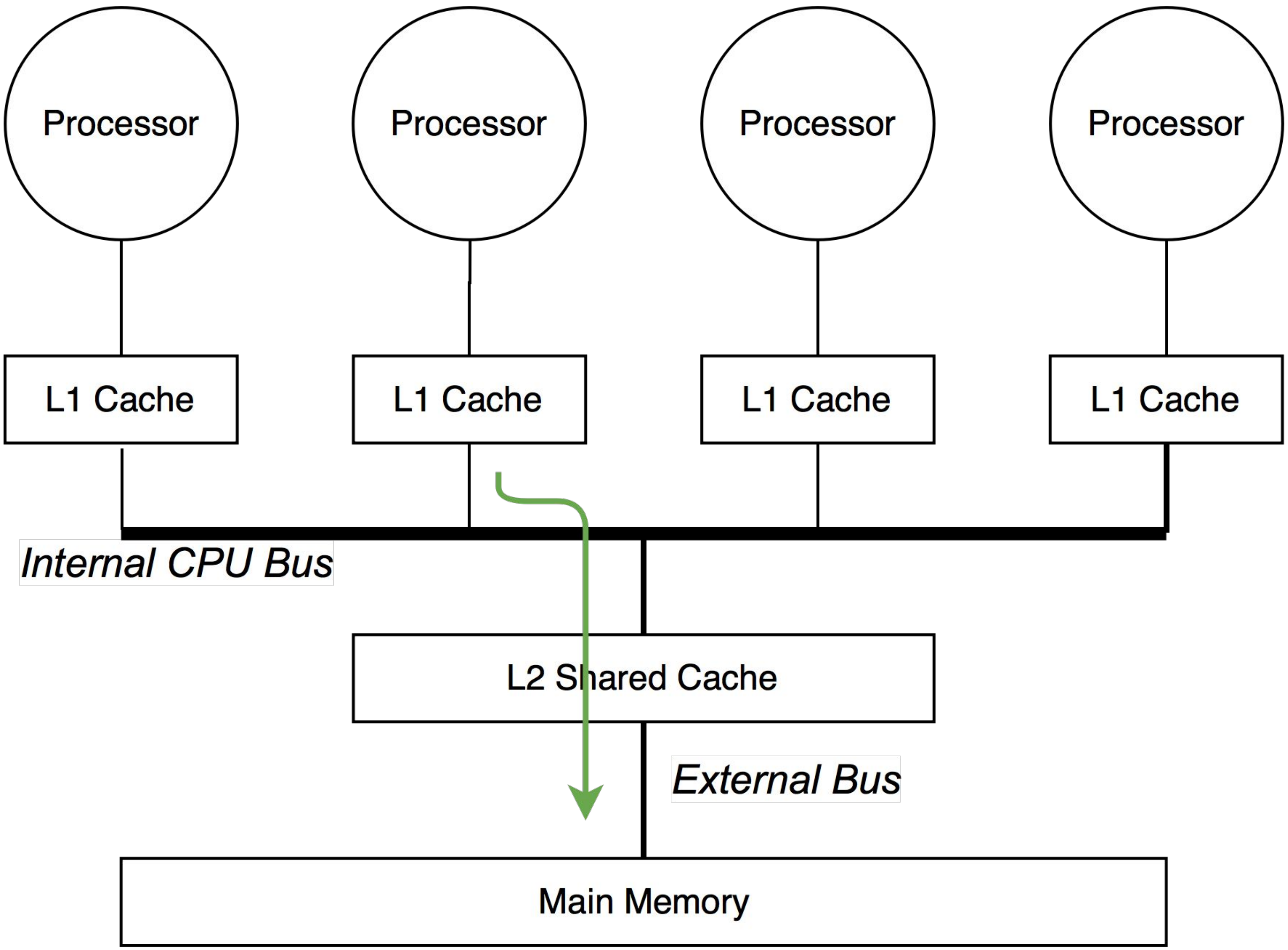
- Introduction
- Snooping-based protocols
- Synchronization
  - Lock acquisition problem
  - Atomic instructions
  - Test-and-set: lock contention problem
  - QL and QOLB

Introduction

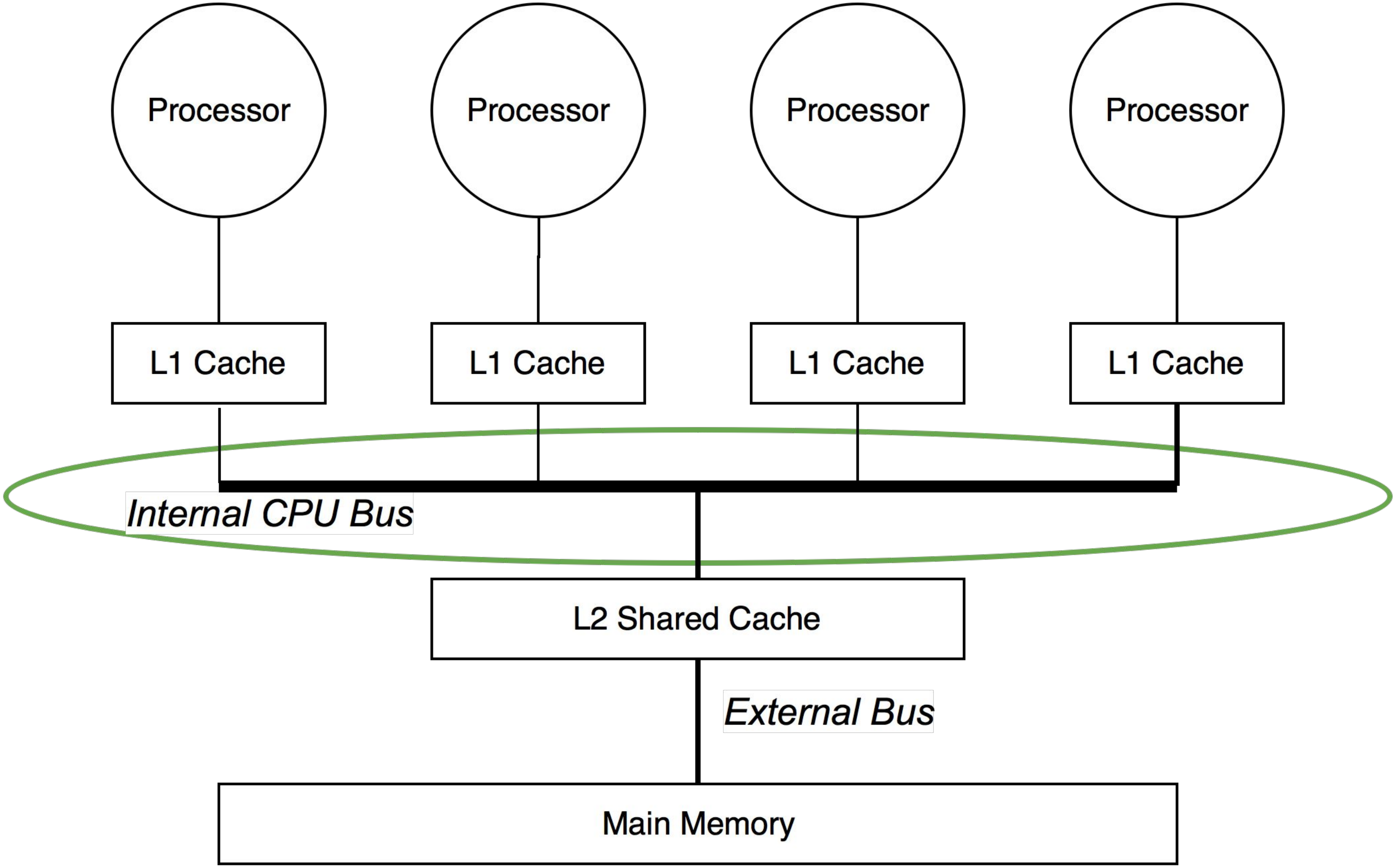
Introduction



Write-through cache

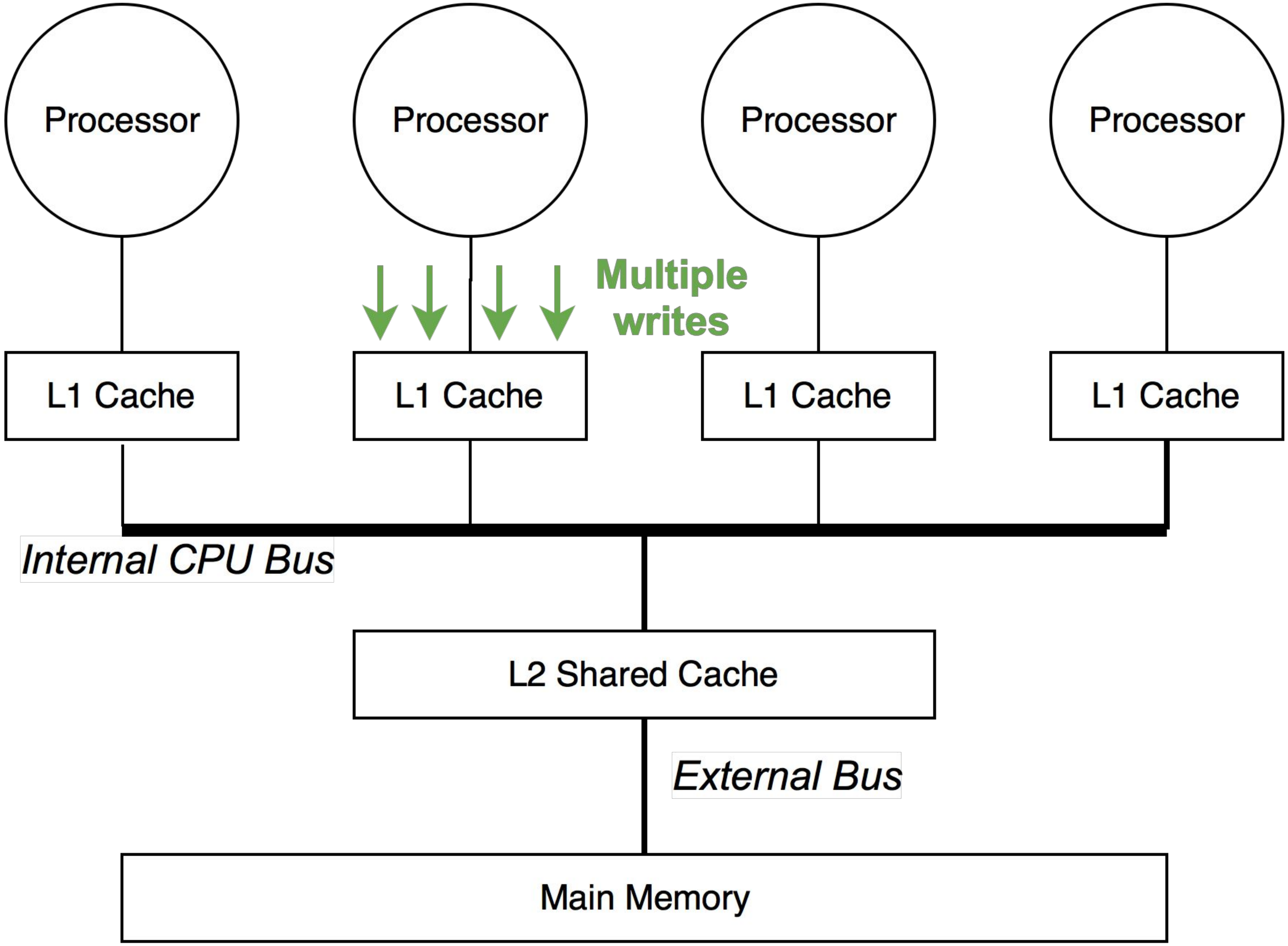


Bottleneck

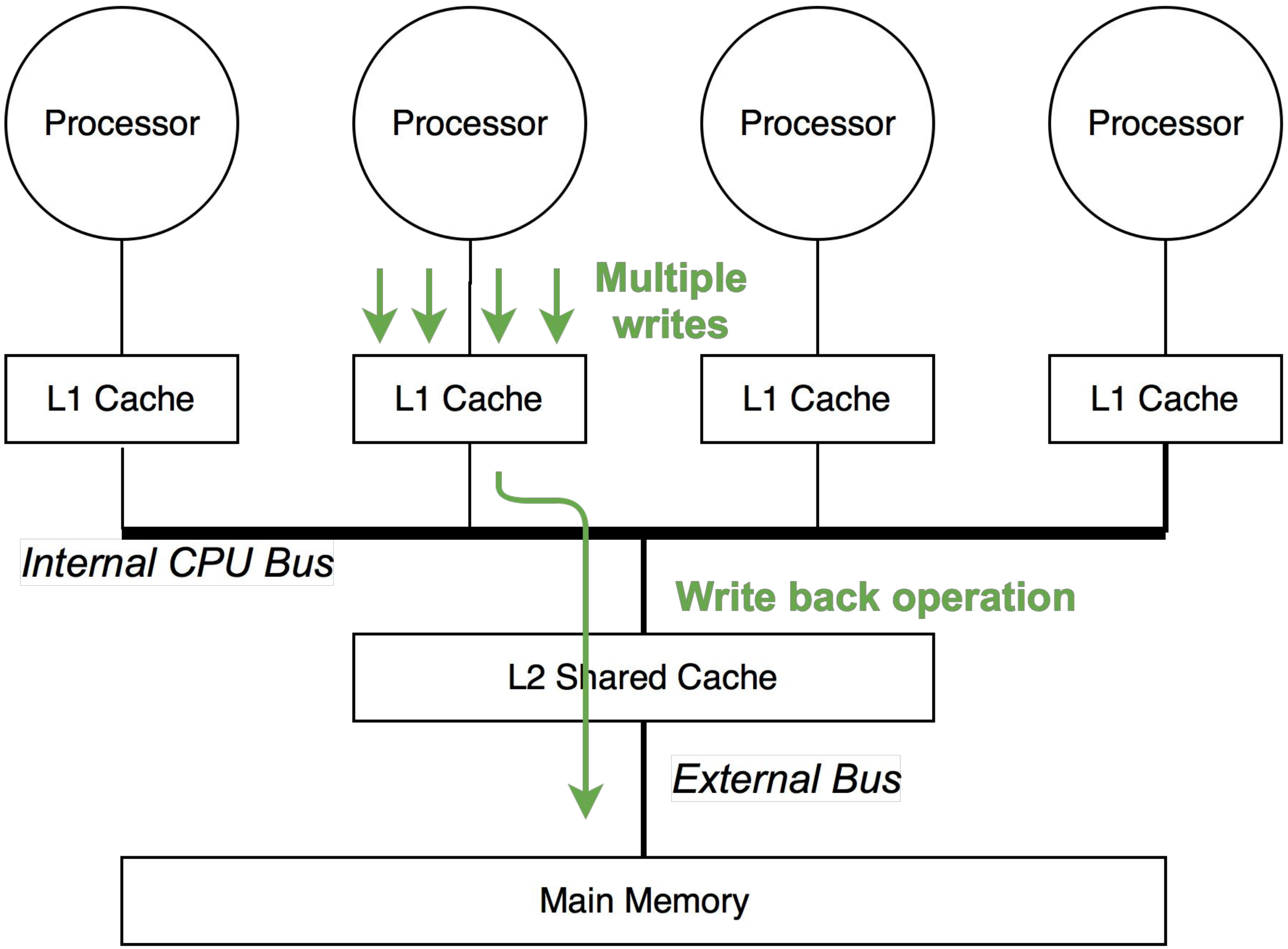


Processor
Processor
Processor
Processor
L1 Cache
L1 Cache
L1 Cache
L1 Cache
Internal CPU Bus
L2 Shared Cache
External Bus
Main Memory

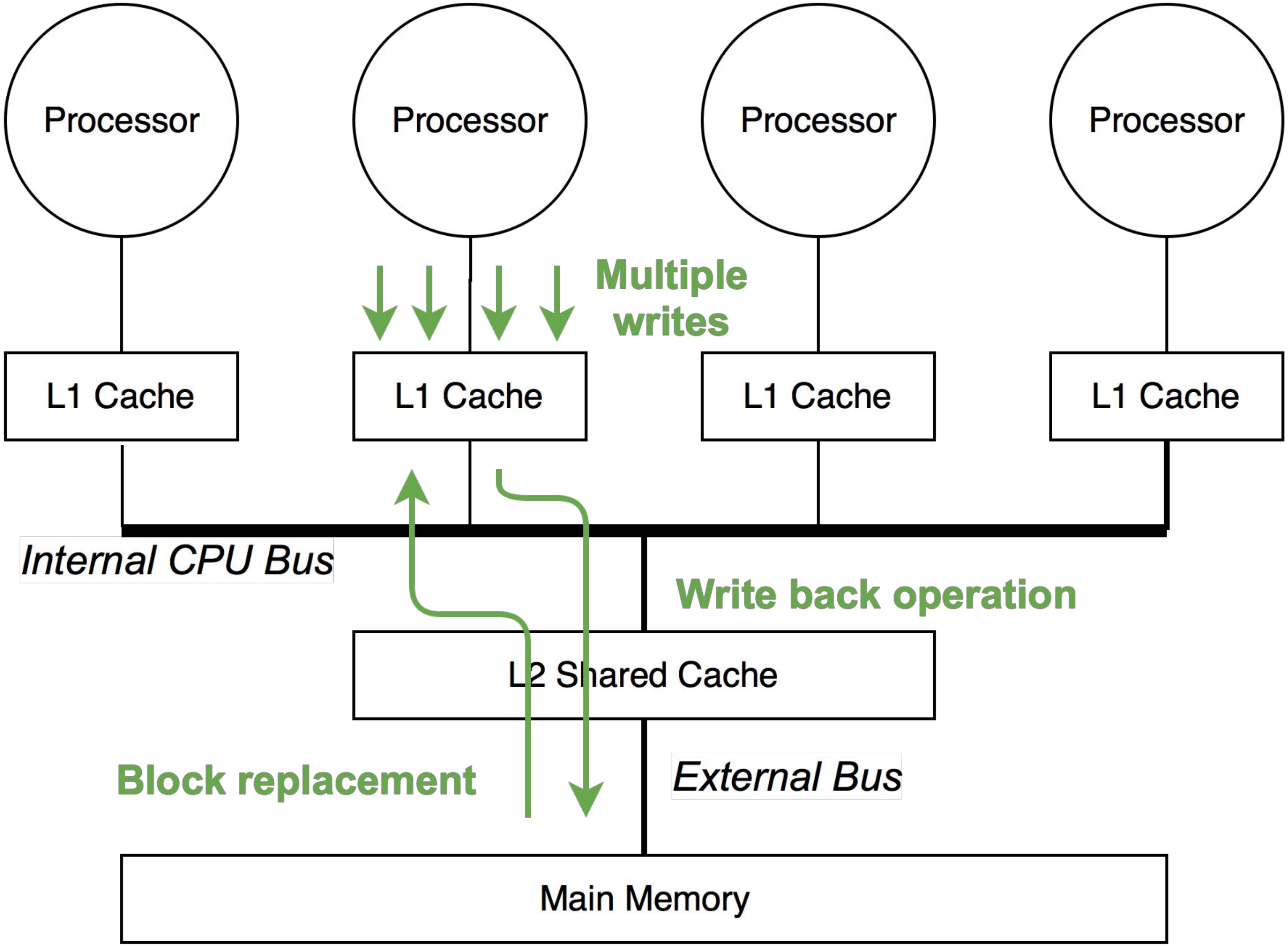
Write-back cache



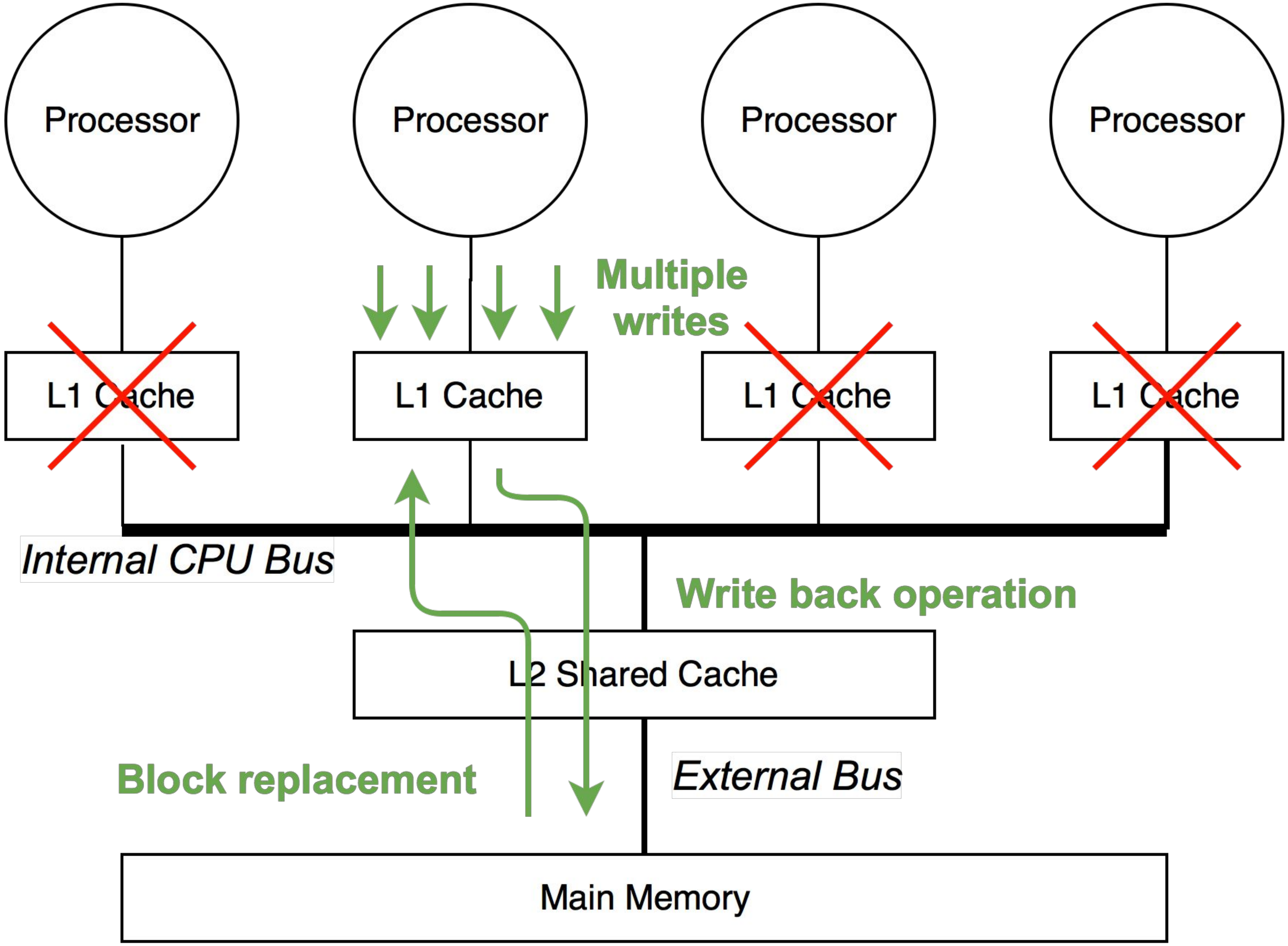
Write-back cache



Write-back cache



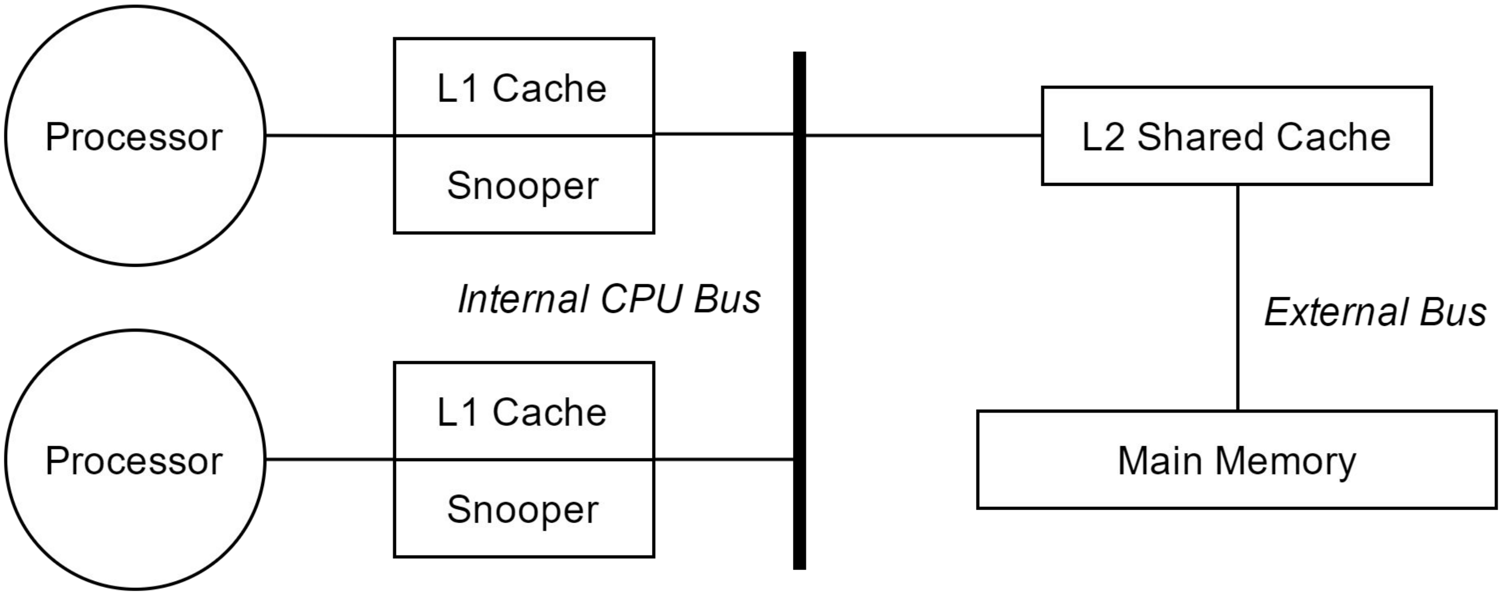
Incoherence



Snooping based protocols

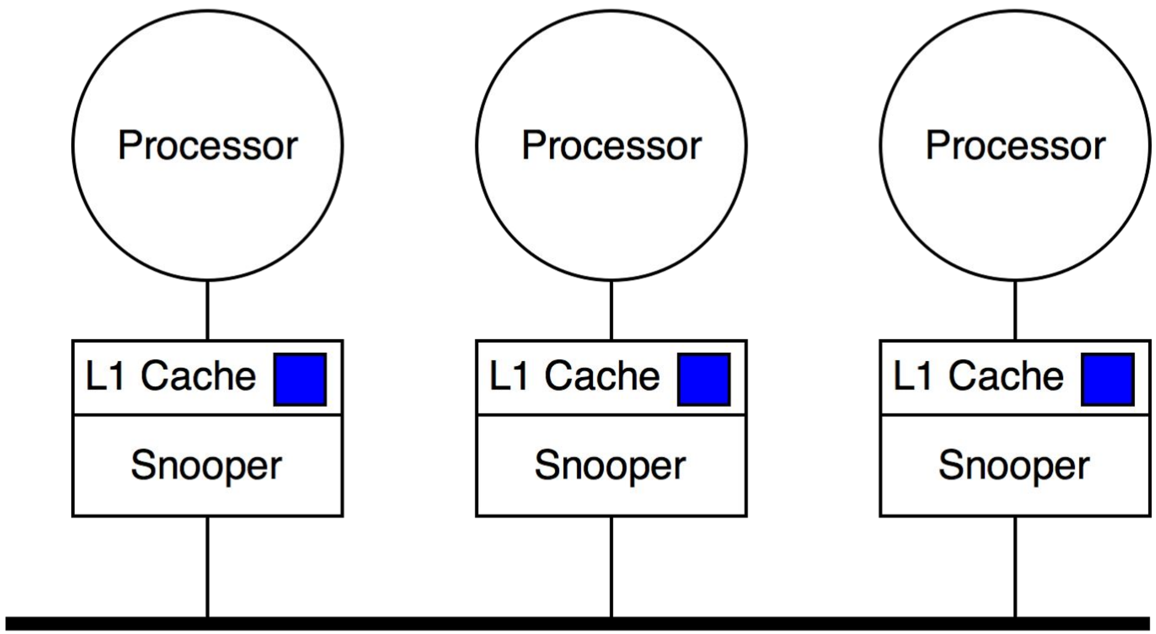
Snooping based protocols

- Cache controllers (snoopers) snoop bus transactions to maintain coherency.
- Two possible behaviours when a cache block is modified:
  - Write-**update**
  - Write-**invalidate**



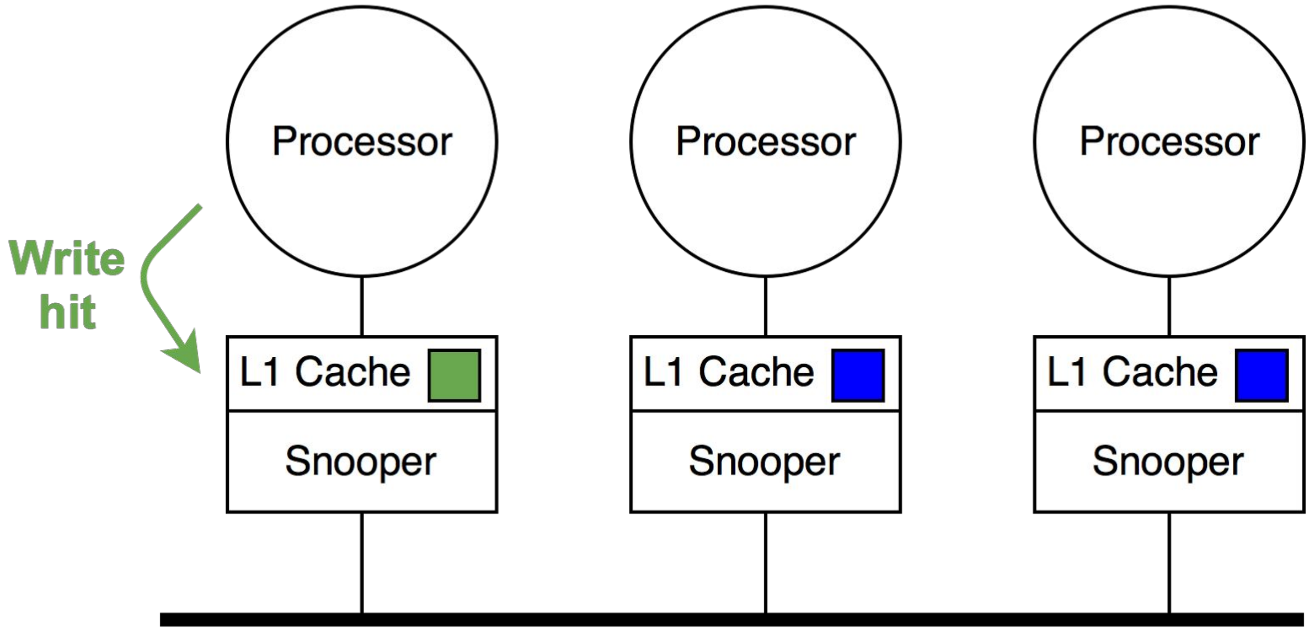
Write-update protocol

- Writing processor's snooper propagates the updated cache block
- Other snoopers snoop the new cache block and update their own cache block copy



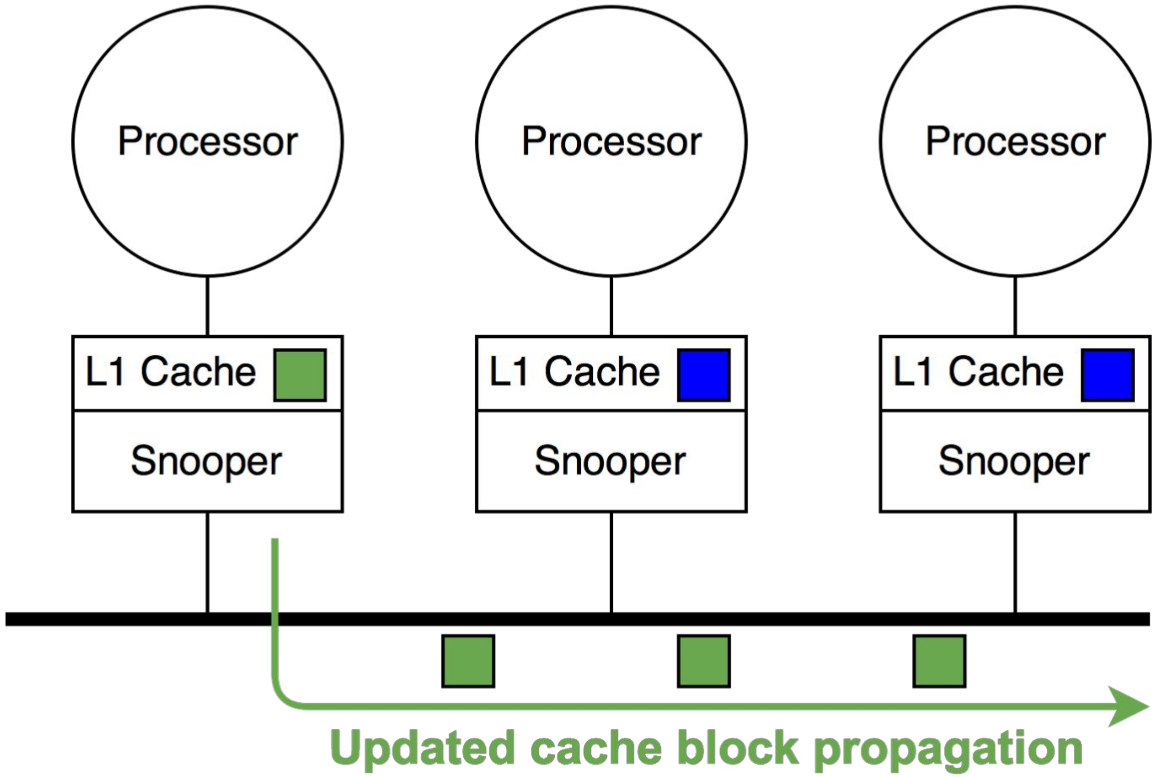
Write-update protocol

- Writing processor's snooper propagates the updated cache block
- Other snoopers snoop the new cache block and update their own cache block copy



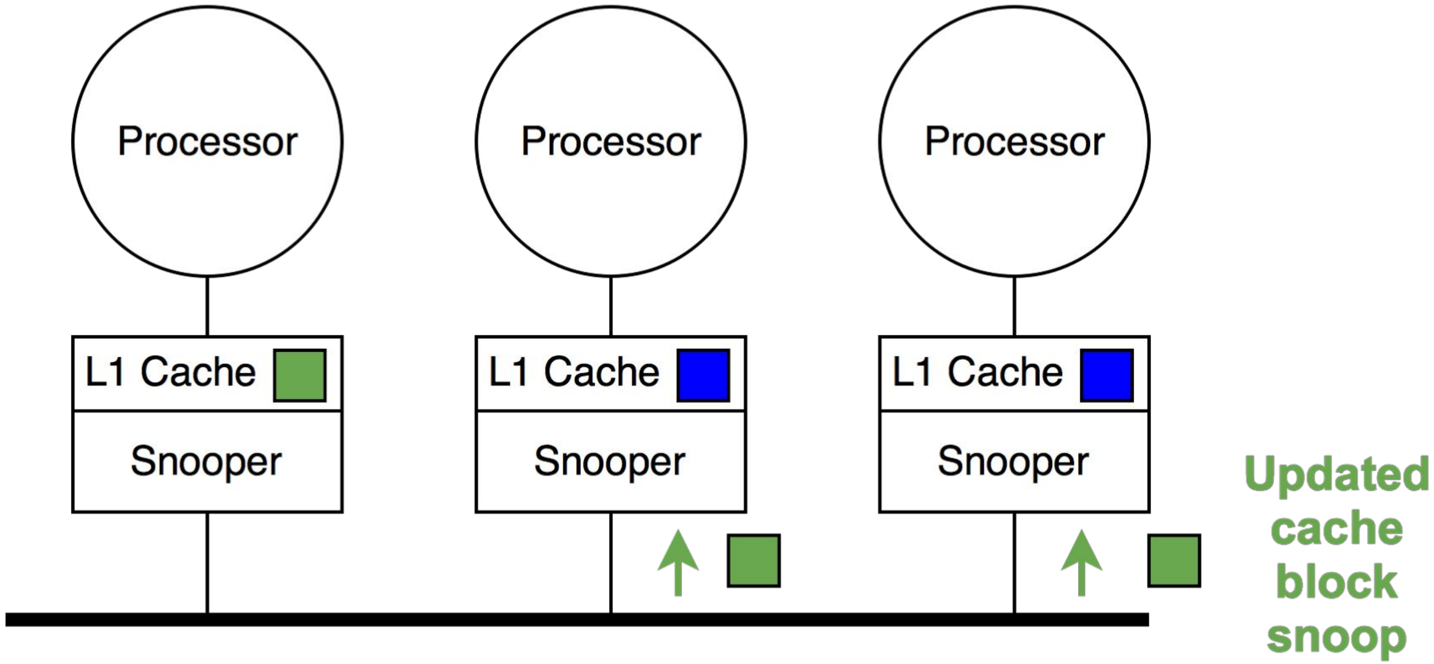
Write-update protocol

- Writing processor's snooper propagates the updated cache block
- Other snoopers snoop the new cache block and update their own cache block copy



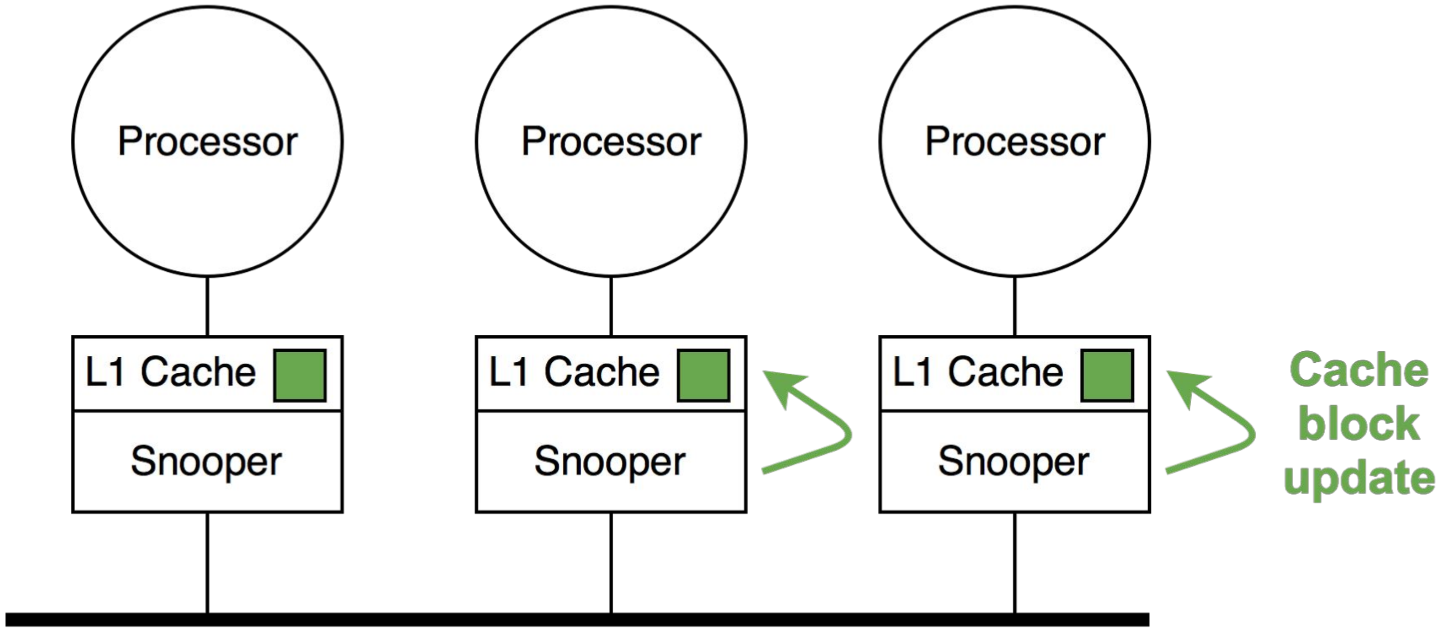
Write-update protocol

- Writing processor's snooper propagates the updated cache block
- Other snoopers snoop the new cache block and update their own cache block copy



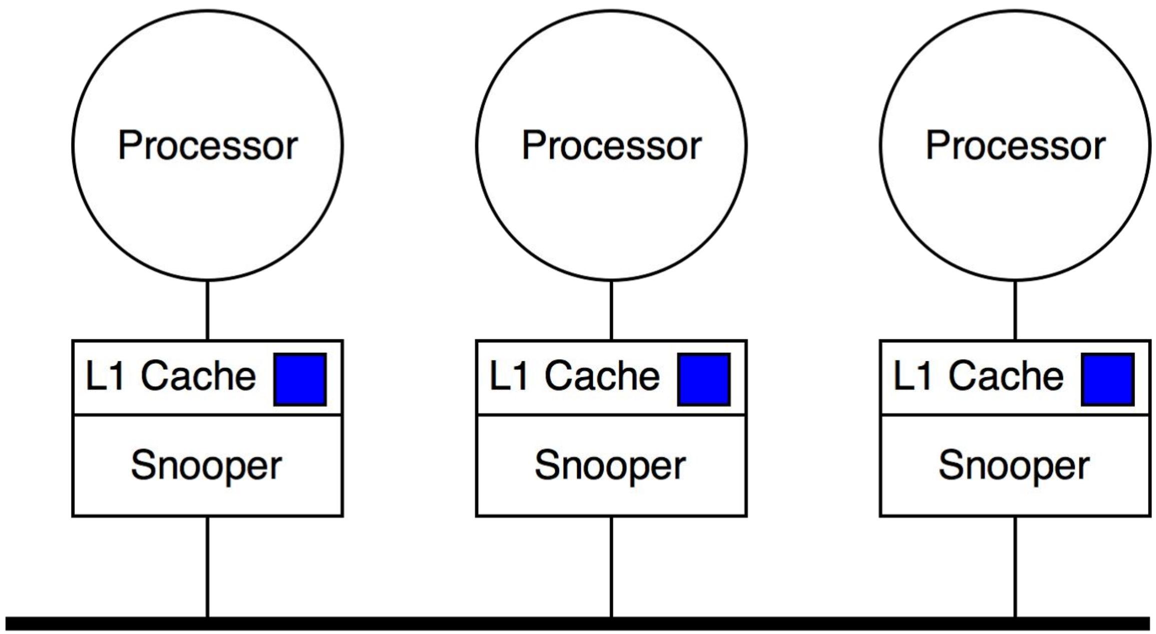
Write-update protocol

- Writing processor's snooper propagates the updated cache block
- Other snoopers snoop the new cache block and update their own cache block copy



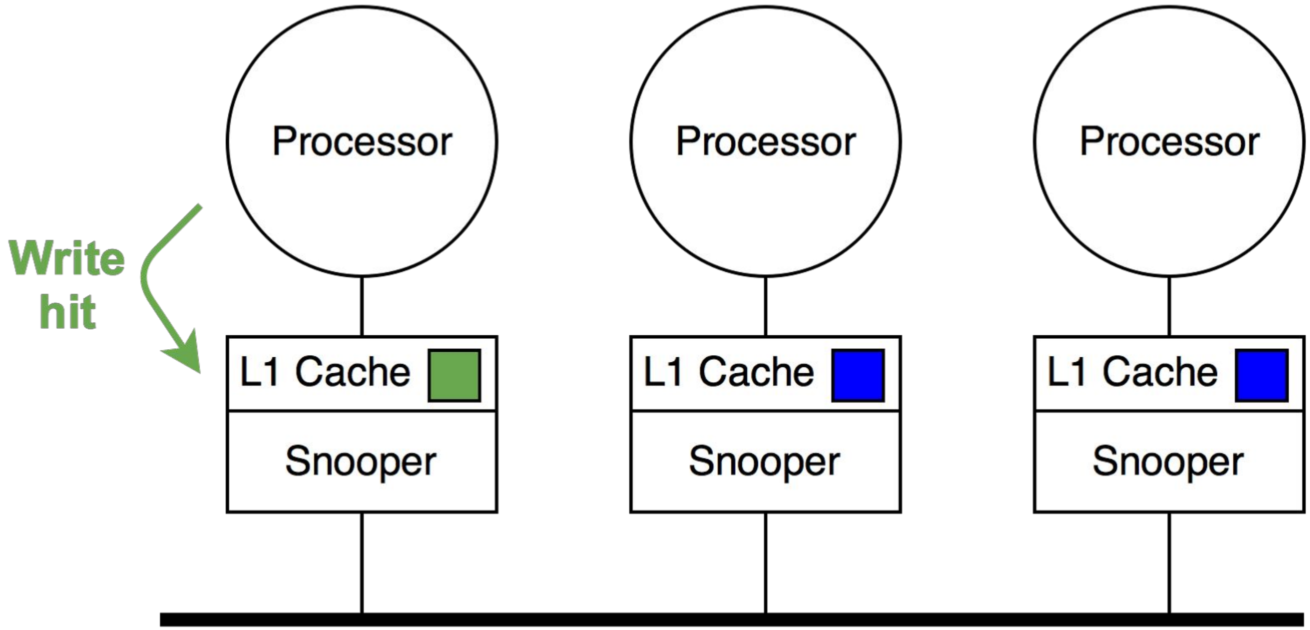
Write-invalidate protocol

- Writing processor issues an invalidation signal just for the first write
- All other snoopers invalidate their own cache block copy



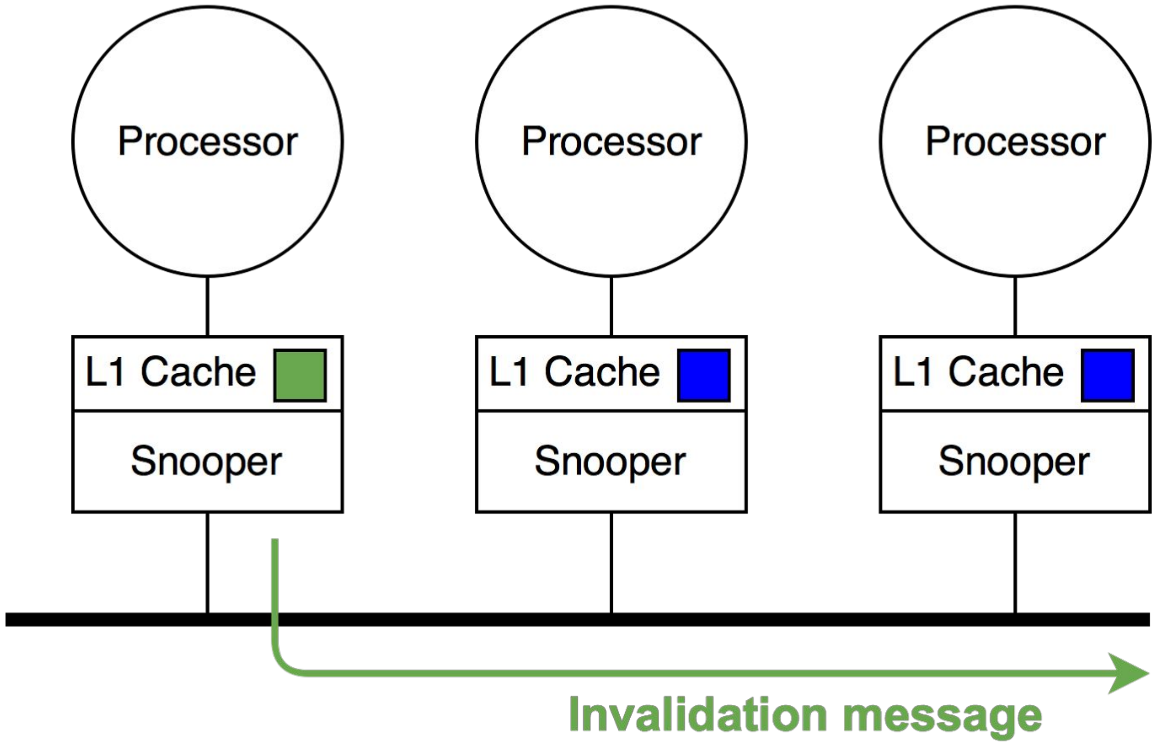
Write-invalidate protocol

- Writing processor issues an invalidation signal just for the first write
- All other snoopers invalidate their own cache block copy



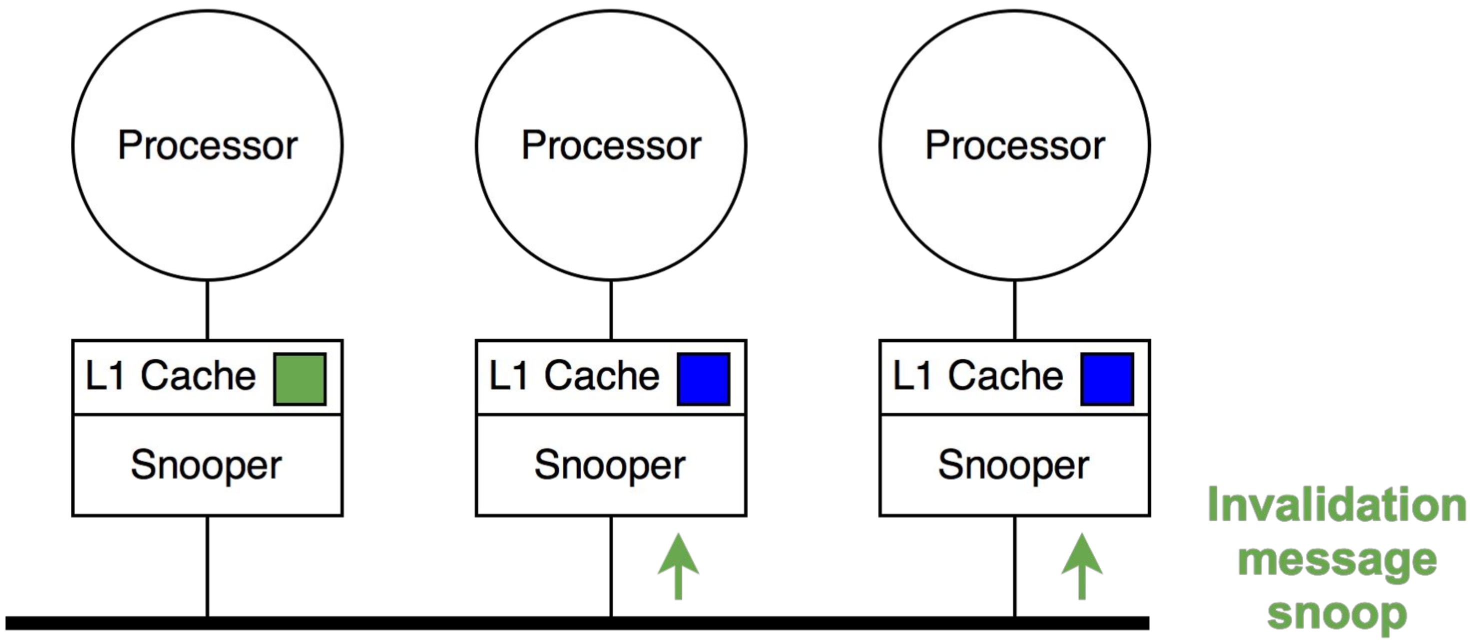
Write-invalidate protocol

- Writing processor issues an invalidation signal just for the first write
- All other snoopers invalidate their own cache block copy



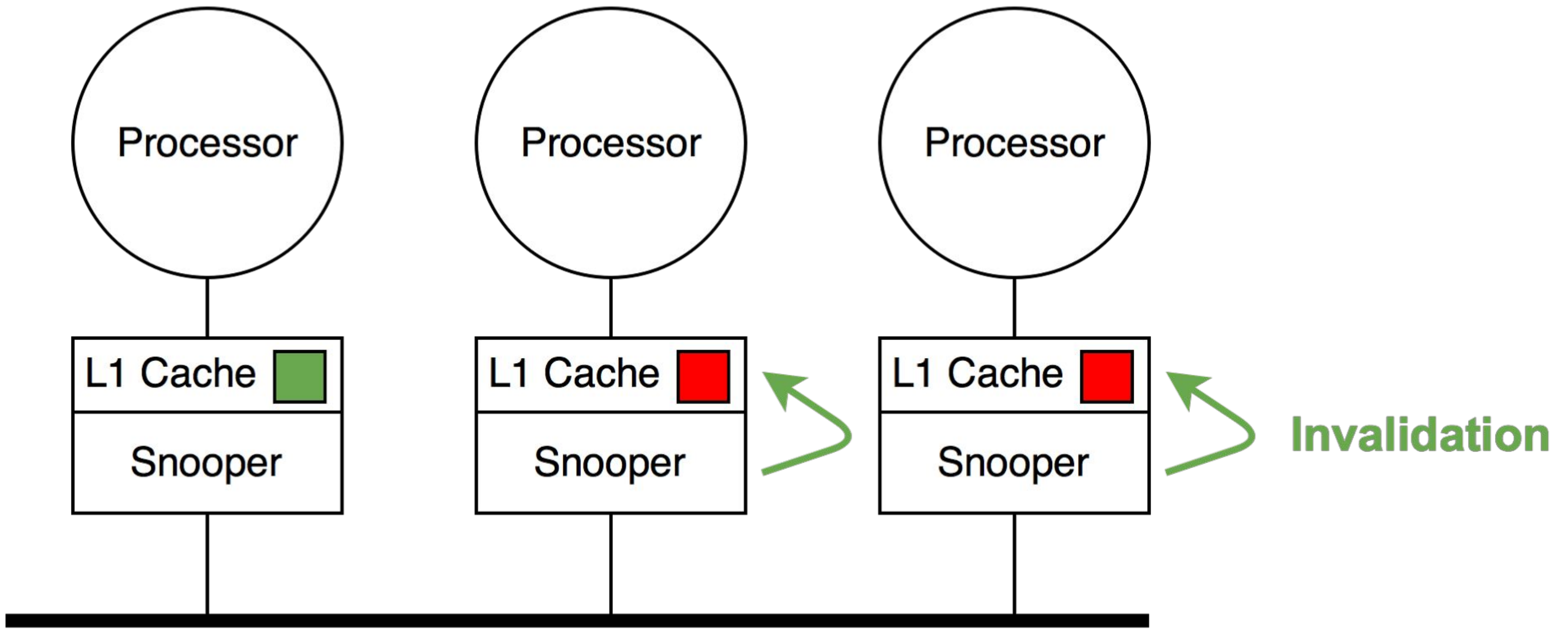
Write-invalidate protocol

- Writing processor issues an invalidation signal just for the first write
- All other snoopers invalidate their own cache block copy



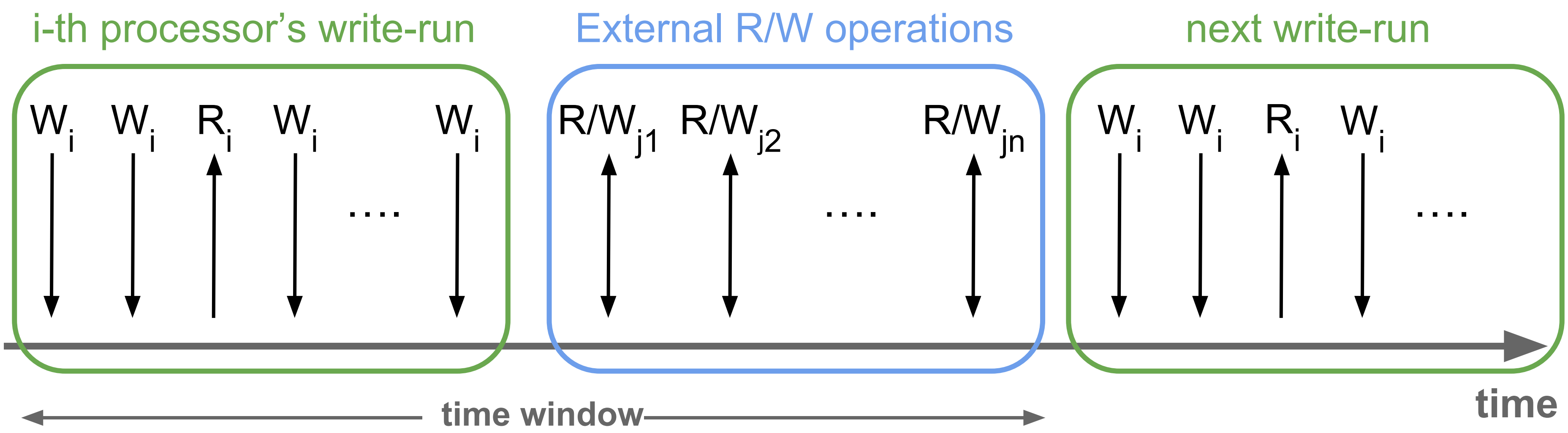
Write-invalidate protocol

- Writing processor issues an invalidation signal just for the first write
- All other snoopers invalidate their own cache block copy



Update vs invalidate: bus transactions

- How many transactions do these protocols require?



- **Write-run**:
  - Set of consecutive writes from the **same** processor which ends with a read or write from **another** processor
  - Let W_r be the average number of writes in it
- Let also n be the average number of operations made by other processors on the same cache block after each write-run

Update vs invalidate: cost evaluation

Write-update	Write-invalidate
• W_r write transactions	• *1* invalidation message • *n* misses ○ Each operation after the write-run will cause a miss

- C_u: updating message cost
- C_i: invalidation message cost

- Average write-invalidate cost per time window
  - $C_{Invalidate} = C_i + n * C_u$
- Average write-update cost per time window
  - $C_{Update} = W_r * C_u$

Update vs invalidate: cost evaluation

- Write-invalidate outperforms write-update when:

$$C_{Invalidate} < C_{Update}$$

$$C_i + n * C_u < W_r * C_u$$

$$W_r > n + C_i / C_u$$

- The best protocol to use depends on $\mathbf{W_r}$ and **n**

Update vs invalidate: which is better

Write-invalidate

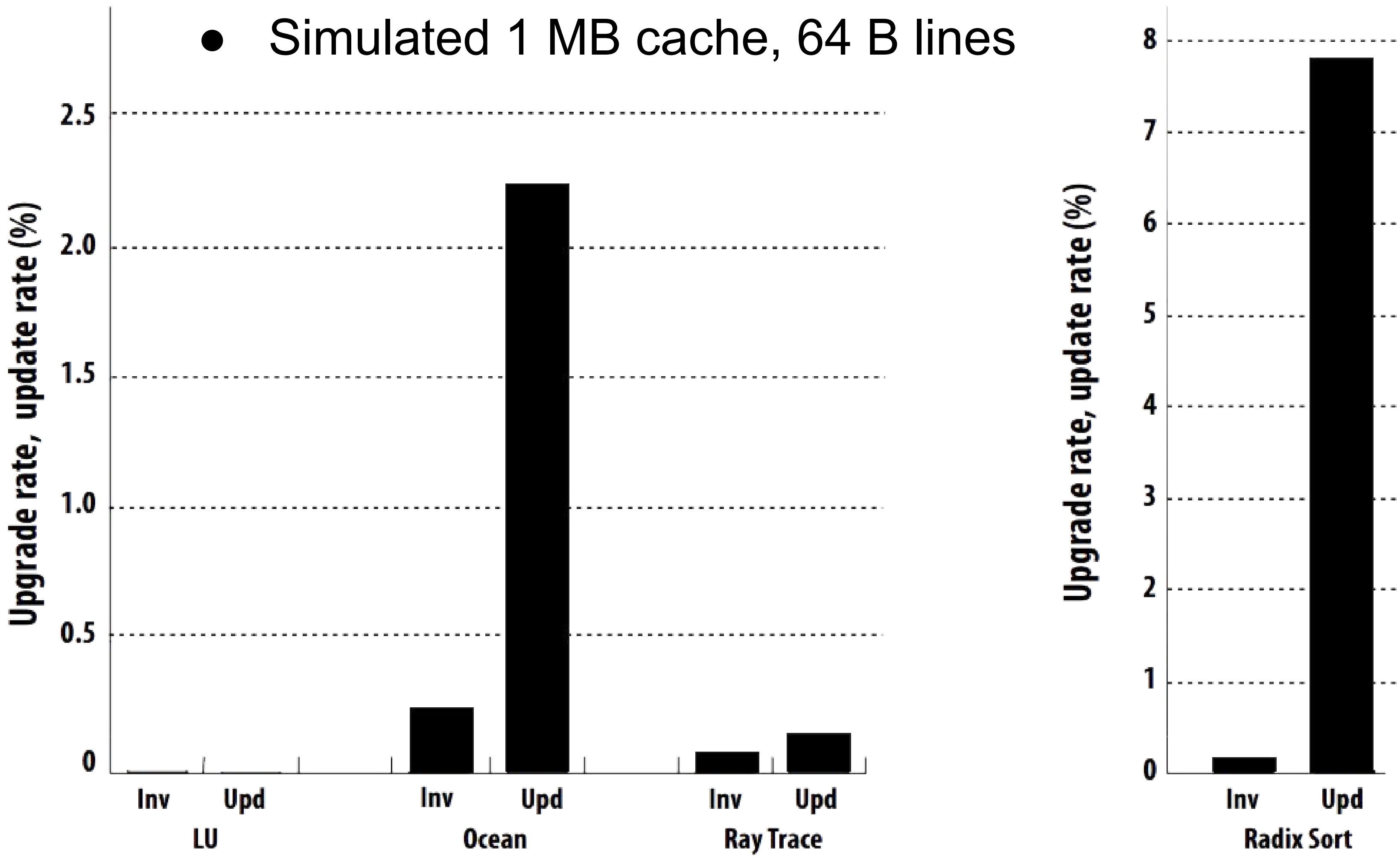
- Better to use when the write-run is long
- Misses will have to be served synchronously, hence they cannot be delayed

Write-update

- Better to use when there’s high contention between processors
- Block updates are asynchronous and can be delayed

Invalidate vs. update evaluation: traffic

- Simulated 1 MB cache, 64 B lines



Fatahalian, K. (2017). Snooping Cache Coherence: Part II - CMU 15-418: Parallel Computer Architecture and Programming. Available at http://15418.courses.cs.cmu.edu/spring2017/lecture/cachecoherence1/slide_041

Typical commercial solutions

- Most of the commercial multiprocessors use:
  - **Write-Back** caches
    - to reduce bus traffic
    - they allow more processors on a single bus
  - **Write-Invalidate** protocol
    - to preserve bus bandwidth
- Typical write-back/write-invalidate protocols:
  - MOESI
  - MESIF

MOESI in AMD and ARM

- Current AMD and ARM cache coherence implementations use MOESI protocol

7.3 Memory Coherency and Protocol

Implementations that support caching support a cache-coherency protocol for maintaining coherency between main memory and the caches. The cache-coherency protocol is also used to maintain coherency between all processors in a multiprocessor system. The cache-coherency protocol supported by the AMD64 architecture is the *MOESI* (modified, owned, exclusive, shared, invalid) protocol. The states of the MOESI protocol are:

http://support.amd.com/TechDocs/24593.pdf

6.5.1 Data cache coherency

The Cortex-A73 processor uses the MOESI protocol to maintain data coherency between multiple cores.

MOESI describes the state that a shareable line in a L1 data cache can be in:

http://infocenter.arm.com/help/topic/com.arm.doc.100048_0002_05_en/cortex_a73_trm_100048_0002_05_en.pdf

MESIF in Intel

- Another cache coherency protocol developed by Intel
- Uses state F instead of state O

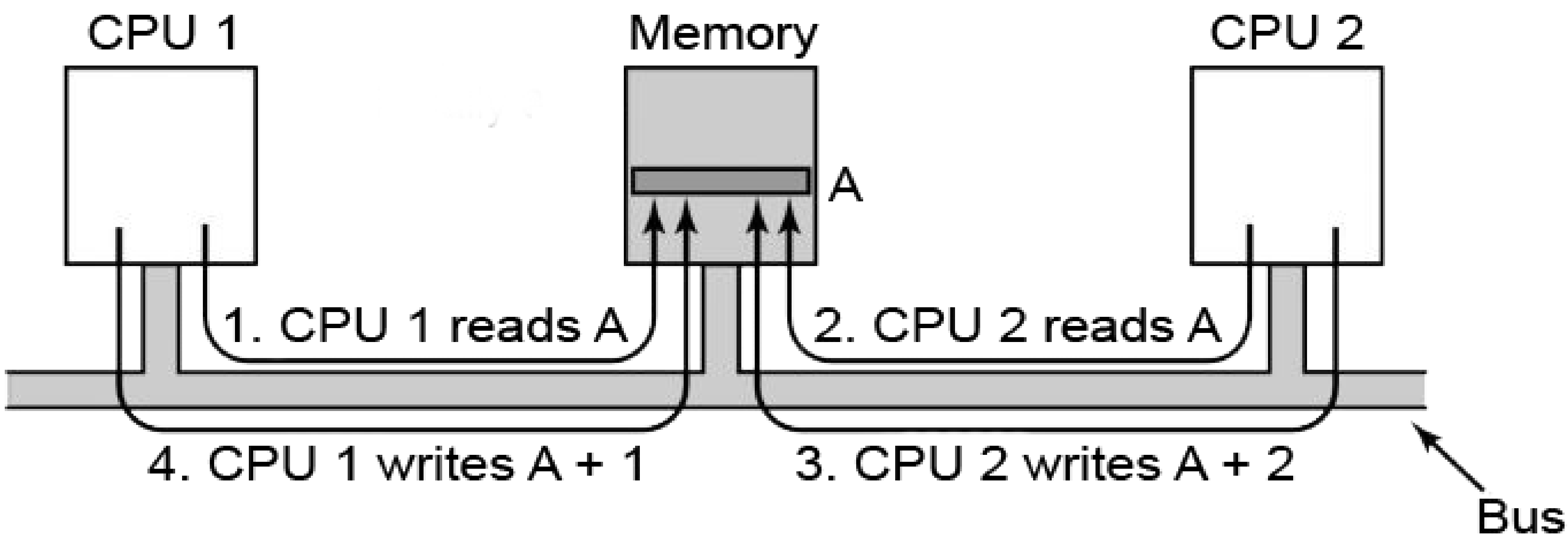
With the introduction of the Intel® QuickPath Interconnect protocol the 4 MESI states are supplemented with a fifth, Forward (F) state, for lines forwarded from on socket to another.

https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf

Synchronization

Multiprocessor synchronization

- Concurrent processes may want to
  - access shared data (or acquire a physical resource) concurrently
  - coordinate their progress relative to each other
- This implies that concurrent processes must be synchronized
  - Cooperation among processors (e.g. Producer–Consumer relationship)



Final result is A + 1 instead of A + 3.

Lock acquisition problem

- Synchronization quite often implies the acquisition and release of **locks**
- These primitives are used by sync libraries which allow developers to write something like:

```
while(!acquire(lock)) { waiting algorithm }
Computation on shared data
release(lock)
```

- Waiting algorithms:
  1. Busy waiting
  2. Blocking
- Acquisition process must be **atomic**

Test-and-set

- Test-and-set is an atomic operation that atomically reads the value of a memory location and writes 1 in it
- In early implementations, the operation was performed by blocking the bus for all the duration of the instruction, but there is a more efficient solution based on cache coherence:
  - Reads the lock value
  - Sets it to 1 anyway
    - If, in the meantime, the copy was invalidated → another processor got the lock → returns 1
    - Returns the lock initial value otherwise

```
loop: test-and-set R2, lock   // test and set the value in location lock
      bnz          R2, loop   // if the result is not zero, spin
      ld           R1, A      // the lock has been acquired
      addi         R1, R1, 1  // increment A
      st           R1, A      // store A
      st           R0, lock   // release the lock; R0 contains 0
```

Other implementations

- Other test-and-set generalizations
  - *Exchange-and-swap*
  - *Compare-and-swap*
- ***Fetch-and-Θ*** operation is a generic name for:
  - Fetch-and-increment
  - Fetch-and-add
  - …
- Its use it’s way more simpler than the test-and-set
  - `fetch-and-increment  A`
  - `fetch-and-add        A, R1`
  - …

Test-and-set: lock contention problem

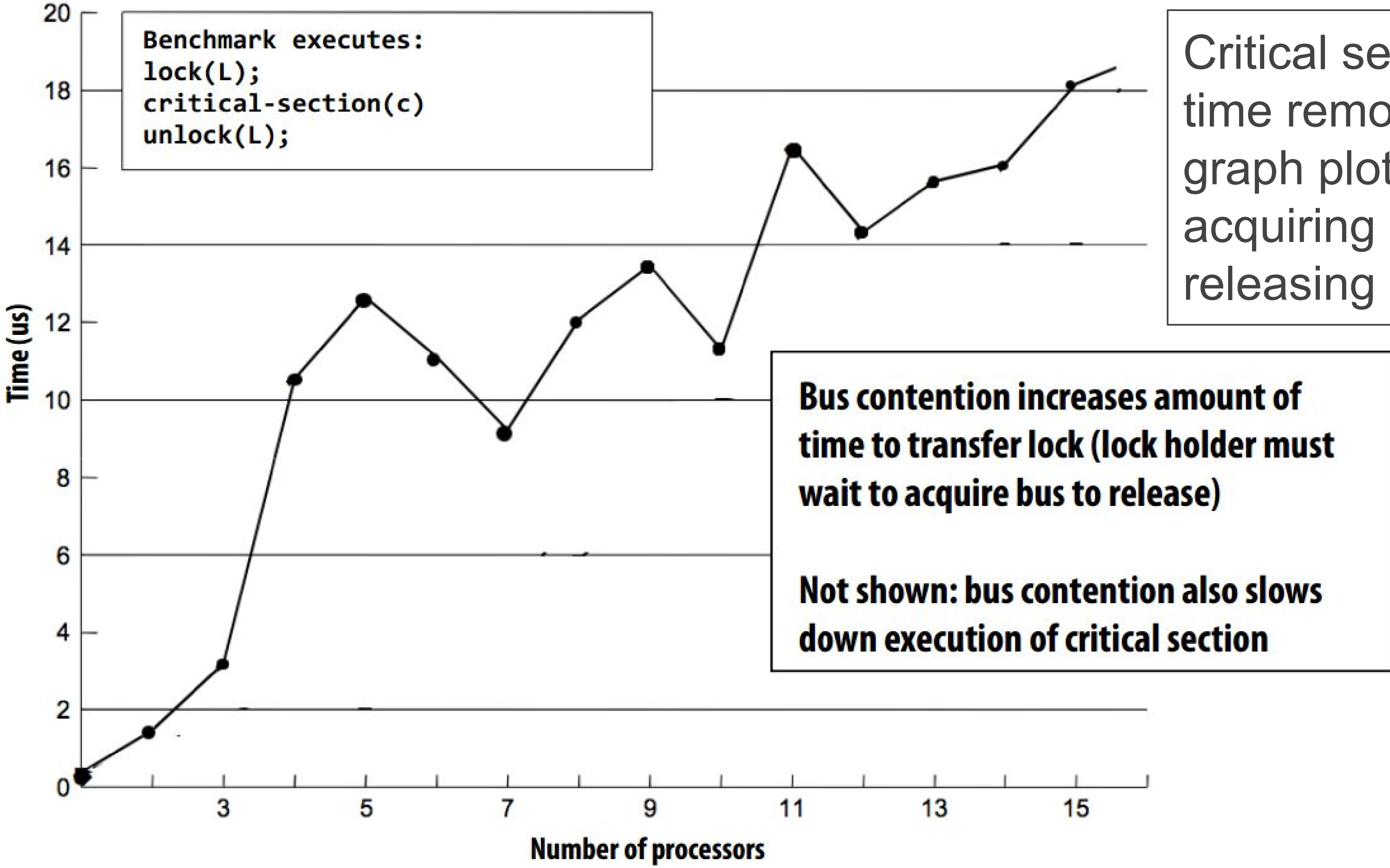
- **Lock contention in spinning locks implementation**
- The first processor that wants to acquire a lock succeeds and caches the lock in a line in the modify (M) state
- The first processor that requests the lock subsequently will get a copy of the lock and test it (unsuccessfully)
  - A write operation is always performed due to the test-and-set implementation: it will then invalidate the holder cache block copy
  - The processor keeps its cached copy in the M state
- The last processor that requests the block will have the unique copy of it in the M state.
- All requesters are repeatedly trying to read and modify the lock, which is in the M state in another cache.

Test-and-set: lock contention problem

- **Lock contention in spinning locks implementation**
- The first processor that wants to acquire a lock succeeds and caches the lock in a line in the modify (M) state
- The first processor that requests the lock subsequently will get a copy of the lock and test it (unsuccessfully)
  - A write operation is always performed due to the test-and-set implementation: it will then invalidate the holder cache block copy
  - The processor keeps its cached copy in the M state
- The last processor that requests the block will have the unique copy of it in the M state.
- All requesters are repeatedly trying to read and modify the lock, which is in the M state in another cache.

Problem: heavy bus utilization

Test-and-set lock performance

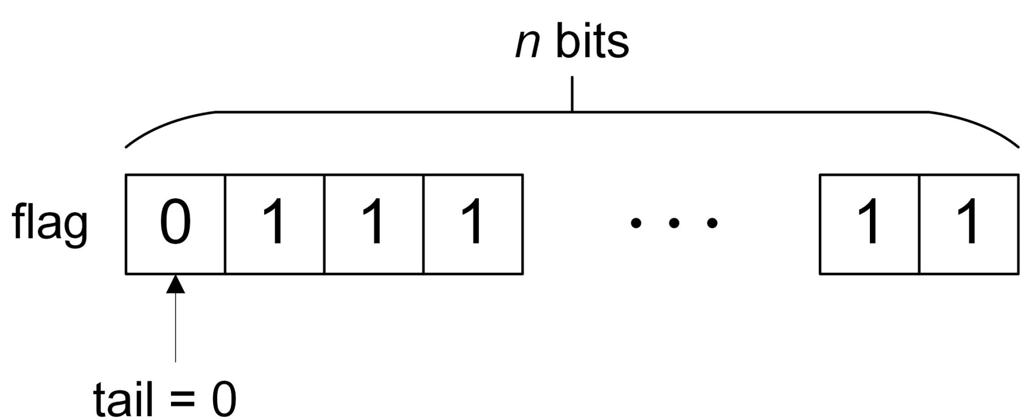


Critical section time removed so graph plots only acquiring and releasing lock time

Fatahalian, K. (2017). Snooping Cache Coherence: Part II - CMU 15-418: Parallel Computer Architecture and Programming. Available at http://15418.courses.cs.cmu.edu/spring2017/lecture/synchronization/slide_023

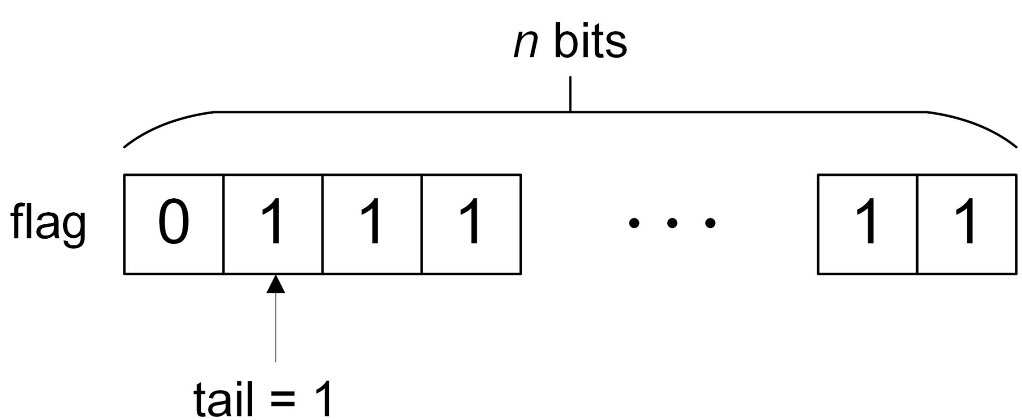
First solution: queueing locks

- Let *n* be the number of processors
- Contention can be reduced by having requesting processors enter a *n* bits long FIFO queue
  - Each bit represents the lock state for each processor
    each lock bit must be in a different cache line, otherwise the lock contention problem appears again
  - Initially the first element contains a “free lock” flag, so the first processor can acquire the lock



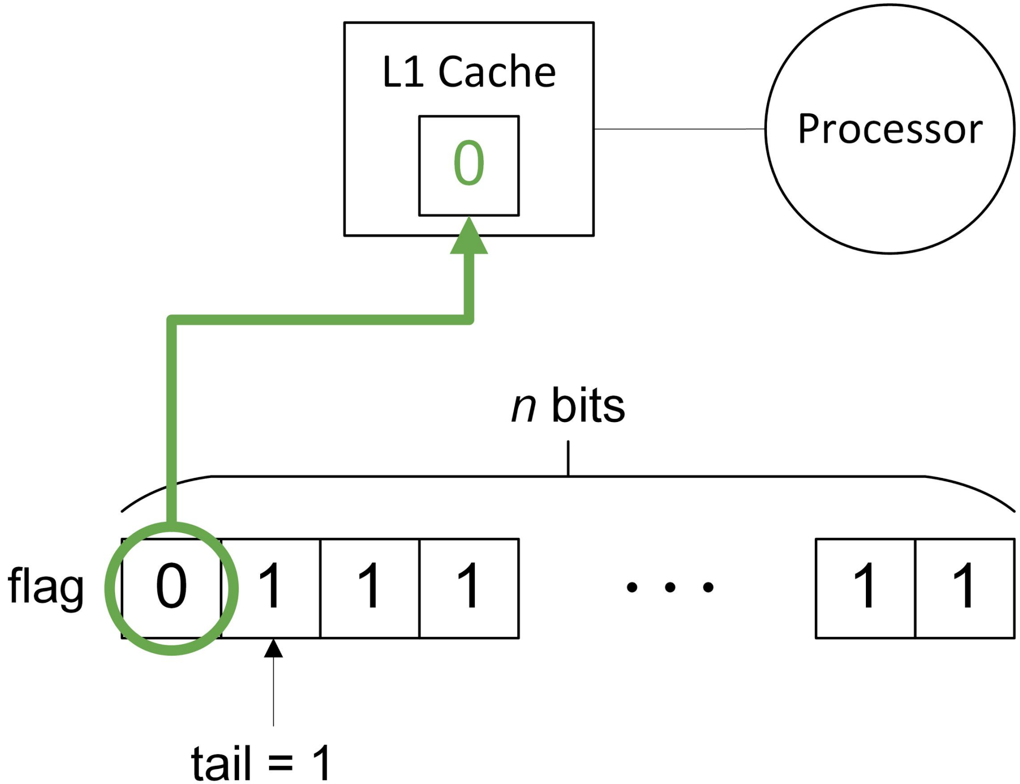
First solution: queueing locks

- A processor requesting the lock will perform:
  `my_index = fetch-and-increment(tail)`
- After this function call:
  - `my_index` is 0
  - `tail` is 1



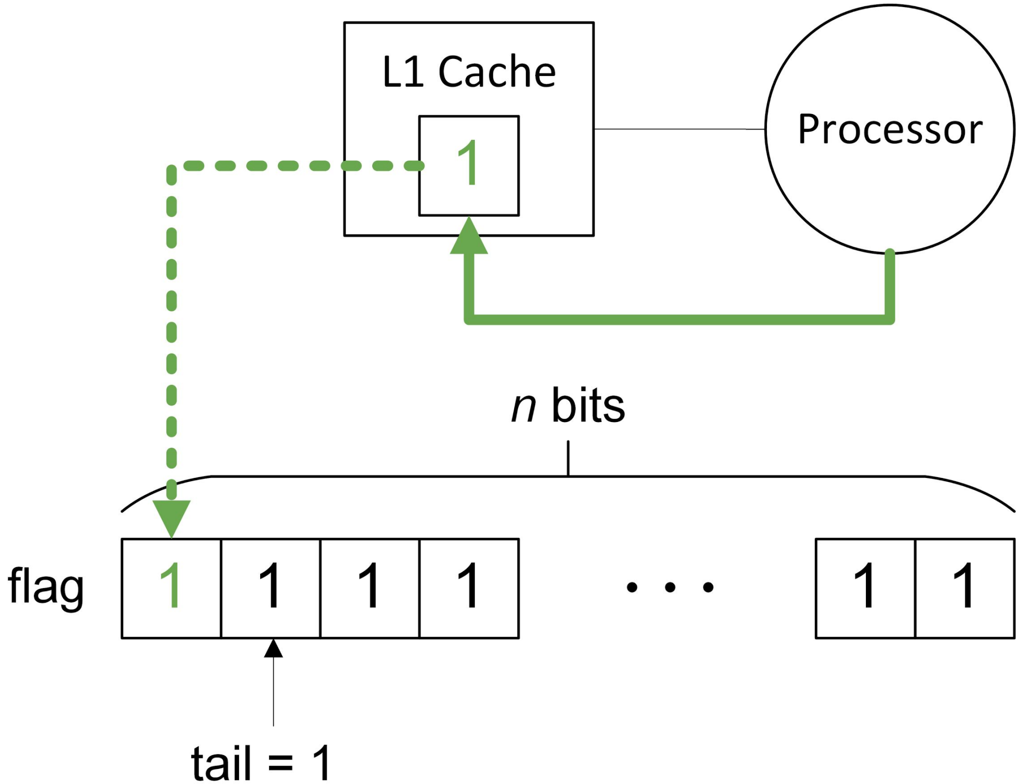
First solution: queueing locks

- The processor reads `flag[my_index]` and caches it
  - its value is 0, so the processor can enter the critical section
  - otherwise it would have continued spinning



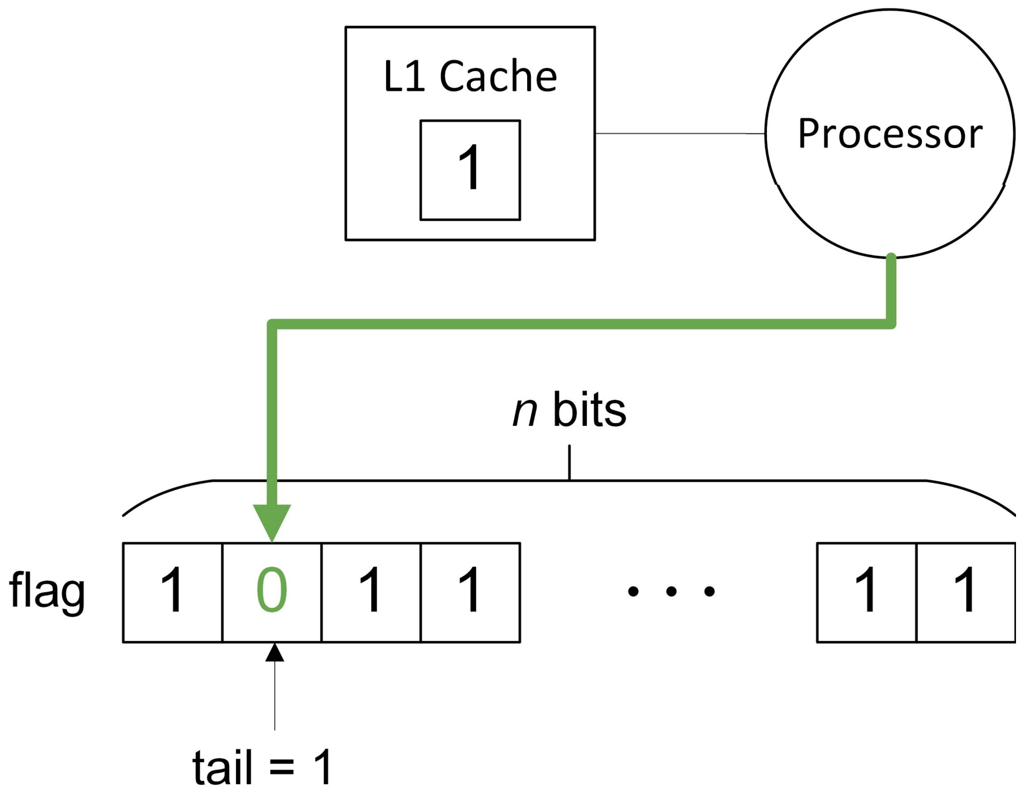
First solution: queueing locks

- The processor, once it entered the critical section, sets his lock state to 1 to make it busy for the next round



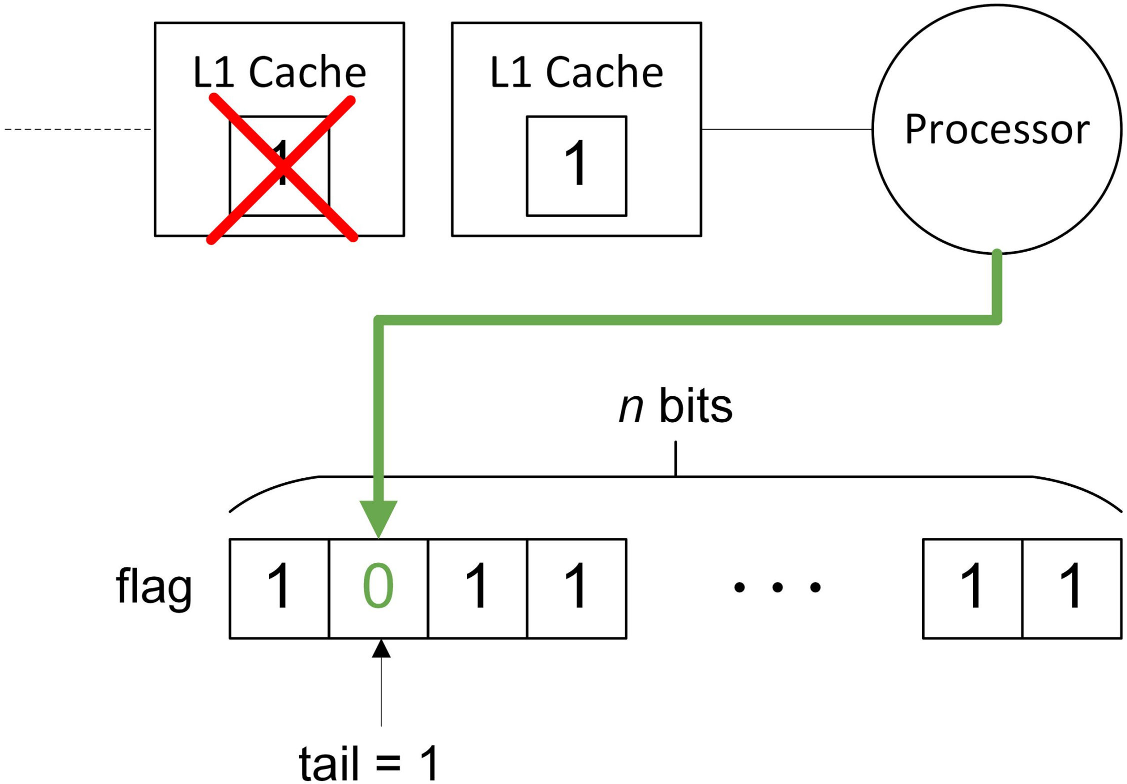
First solution: queueing locks

- At the end of its critical section, the processor releases the lock to the next processor, by setting `flag[(my_index + 1) % n]` to 0



First solution: queueing locks

- The write operation will invalidate the line containing the lock in the cache of the processor corresponding to myindex + 1.
    - This will generate a read miss for the latter, and upon reading of its flag, its test will be successful.



Software implementation of QL

```
init:    flag[0] := 0; // Initially, 1st processor can have the lock
         for(i:= 0; i < n; i++) // All other processors will see a busy lock
             flag[i] := 1;
         tail := 0;

acq:     myindex := fetch-and-increment(tail);  // Increment is modulo n
         while(flag[myindex] == 1); // Spins while the lock is held elsewhere

         // The processors gots the lock and makes it busy for the next round
         flag[myindex] := 1;

rel:      // Releases the lock and passes it on the next processor
         flag[(myindex + 1) % n] := 0;
```

Queuing Locks: pros & cons

- Advantages:
  - Reduced bus traffic
- Drawbacks:
  - Relying on fetch-and-increment
  - Each lock must be in a different cache line (distributed lock), or contention will occur while performing fetch-and-increment.
    - No shared data coallocation

Second solution: QOLB

- Completely in hardware (Queue On Locked Bit)
- Hardware queue of waiting processors' IDs
- Only **one lock variable**
  - Enqueue operation allocates a shadow copy of the line containing the lock in the processor's cache
  - Spinning is performed in cache if the lock bit is set to busy
  - When the processor holding the lock releases it, it performs a *dequeue operation* that directly **sends the freed lock and the data in the same line to the next waiting processor**
- Pro: QOLB outperforms other schemes
- Cons: Significant complexity cost
  - Further complications in coherence protocols
  - **Direct transfer from one cache to another is required**

References

- Jean-Loup Baer, Microprocessor Architecture: from Simple Pipelines to Chip Multiprocessor
- Hennessy & Patterson, Computer Architecture, 5th edition
- http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.cortexa.cortexa73/index.html
- http://www.anandtech.com/show/10347/arm-cortex-a73-artemis-unveiled/2
- http://ark.intel.com/products/series/53672/Intel-Xeon-Processor-E7-8800-Product-Family
- https://en.wikipedia.org/wiki/MOESI_protocol
- https://en.wikipedia.org/wiki/MESIF_protocol
- http://www.realworldtech.com/common-system-interface/5/
- http://15418.courses.cs.cmu.edu/spring2017content/lectures/16_synchronization/16_synchronization_slides.pdf
- https://it.wikipedia.org/wiki/Memoria_cache#Protocolli_di_Coerenza
- http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1676921

Any questions?

Thank you for your attention