

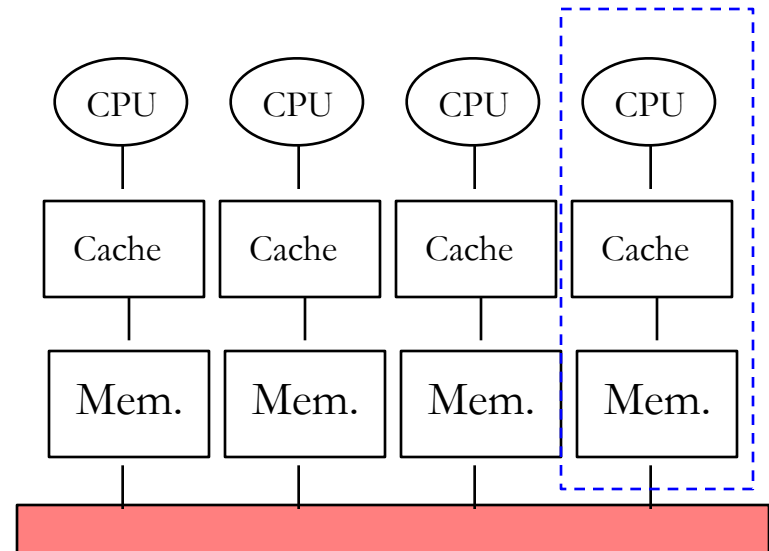
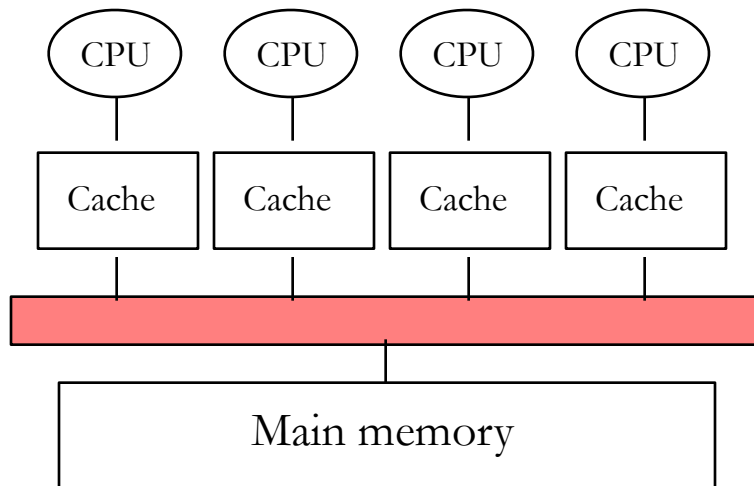
Lect. 4: Shared Memory Multiprocessors

- Obtained by connecting full processors together
 - Processors have their own connection to memory
 - Processors are capable of independent execution and control
(Thus, by this definition, GPU is not a multiprocessor as the GPU cores are not capable of independent execution, but 2nd generation Xeon Phi is!!)
- Have a single OS for the whole system, support both processes and threads, and appear as a common multiprogrammed system
(Thus, by this definition, Beowulf clusters are not multiprocessors)
- Can be used to run multiple sequential programs concurrently or parallel programs
- Suitable for parallel programs where threads can follow different code (task-level-parallelism)



Shared Memory Multiprocessors

- Recall the two common organizations:
 - Physically centralized memory, uniform memory access (UMA) (a.k.a. SMP)
 - Physically distributed memory, non-uniform memory access (NUMA)



(Note: both organizations have local caches)




Shared Memory Multiprocessors

- Recall the communication model:
 - Threads in different processors can use the same virtual address space
 - Communication is done through shared memory variables

Producer (p1)

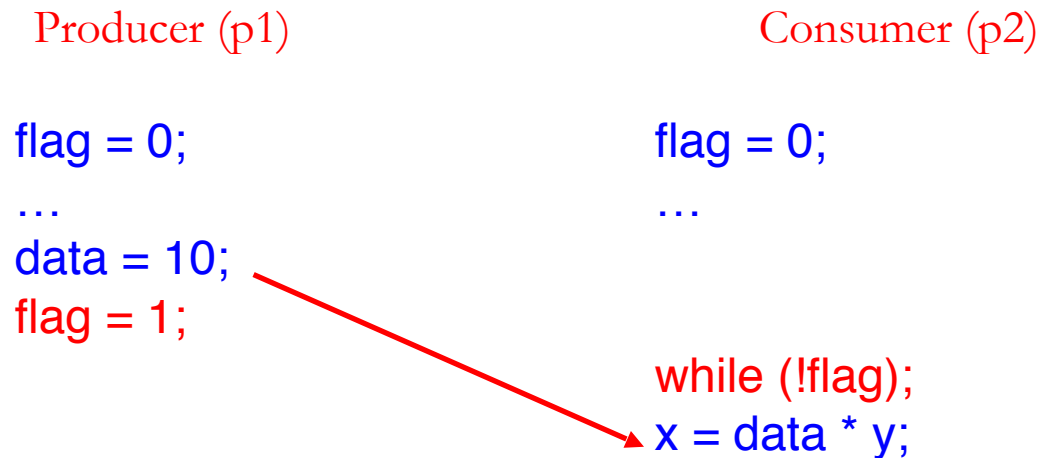
Consumer (p2)

`data = 10;`  `x = data * y;`



Shared Memory Multiprocessors

- Recall the communication model:
 - Threads in different processors can use the same virtual address space
 - Communication is done through shared memory variables
 - Explicit synchronization (e.g., variable flag below)



HW Support for Shared Memory

- Cache Coherence
 - Caches + multiprocessors → stale values
 - System must behave correctly in the presence of caches
 - Write propagation
 - Write serialization
- Memory Consistency
 - When should writes propagate?
 - How are memory operations ordered?
 - What value should a read return?
- Primitive synchronization instructions
 - Memory fences: memory ordering on demand
 - Read-Modify-writes: support for locks (critical sections)
 - Transactional memory extensions



Cache Coherence

Producer (p1)

```
flag = 0;  
...  
data = 10;  
  
flag = 1;
```

Consumer (p2)

```
flag = 0;  
...  
  
while (!flag);  
  
x = data * y;
```



The update to flag (and data) should be (eventually) visible to p2



Memory Consistency

Producer (p1)

Consumer (p2)

flag = 0;

flag = 0;

...

...

data = 10;

flag = 1;

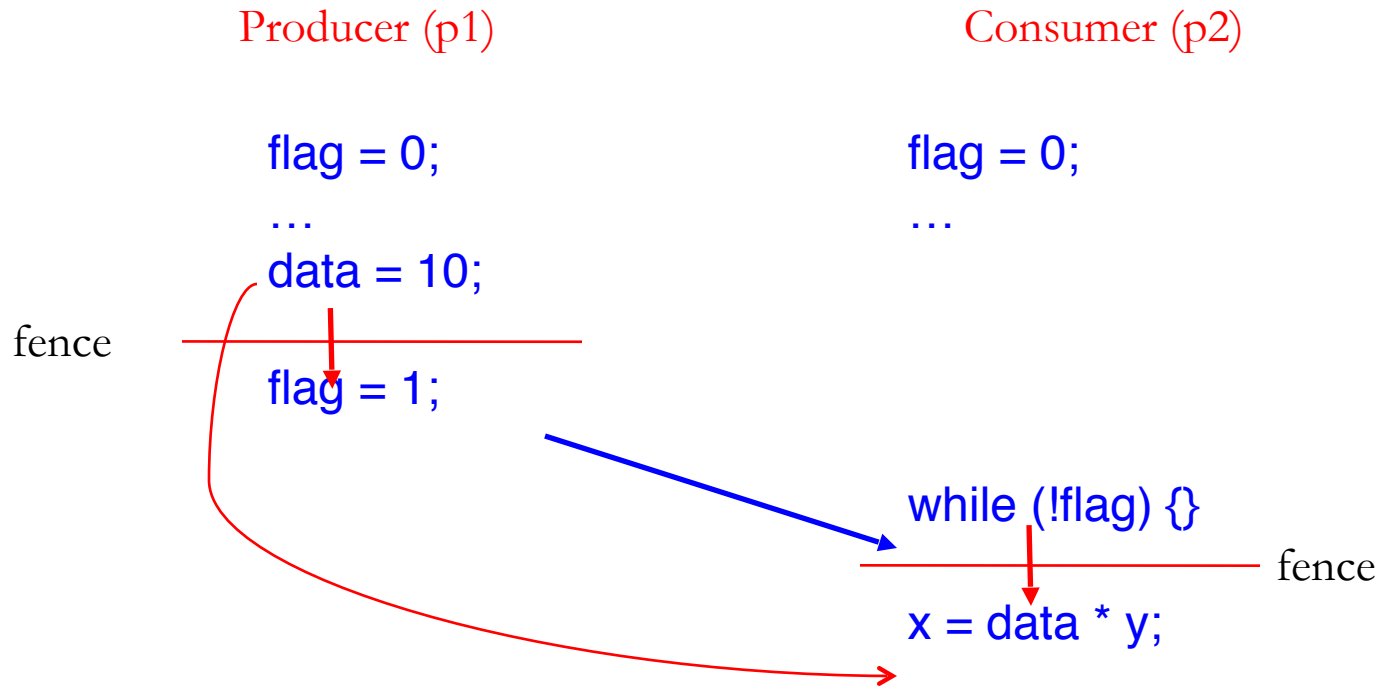
while (!flag) {}

x = data * y;

If p2 sees the update to flag, will p2 see the update to data?



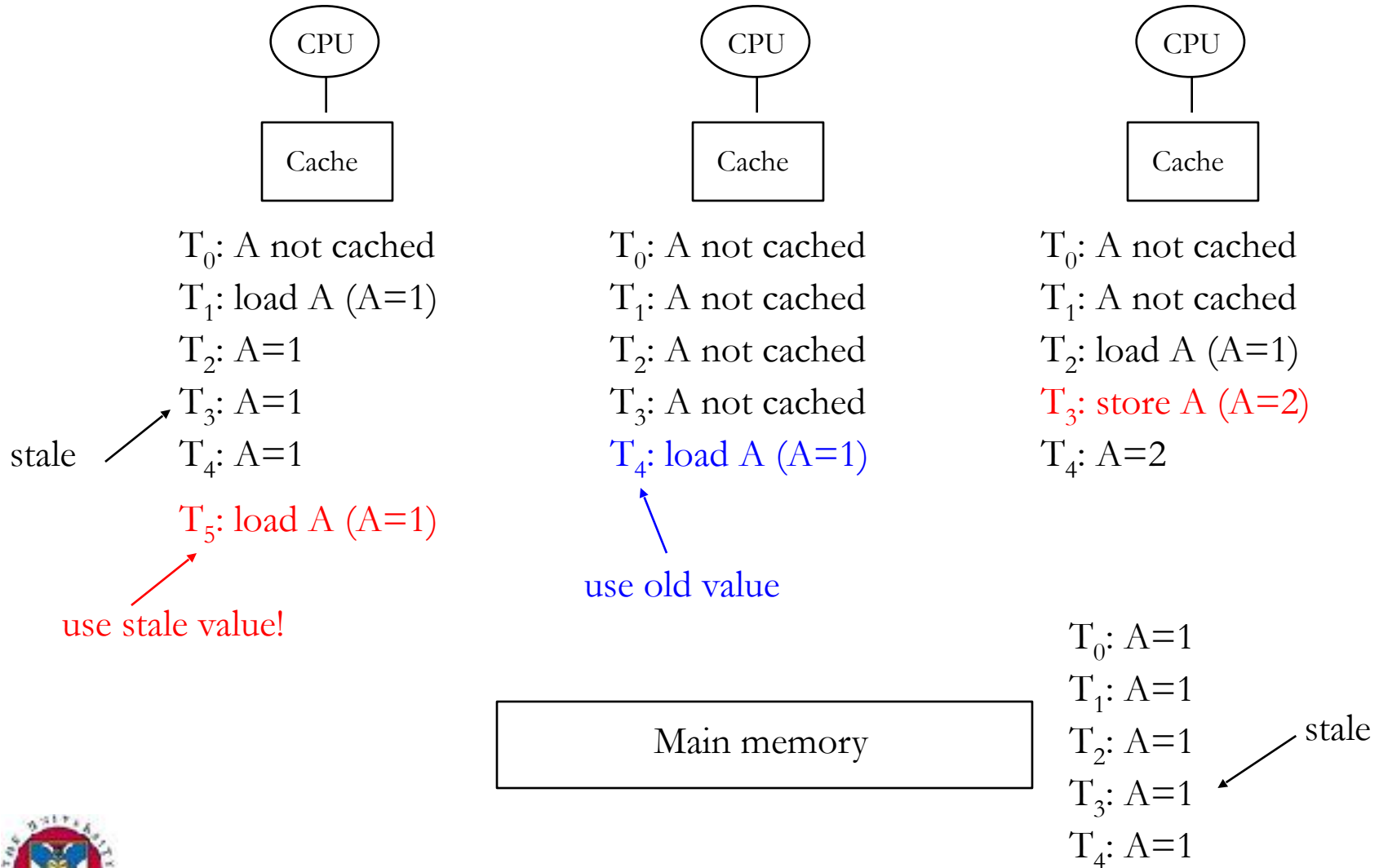
Primitive Synchronization



If p2 sees the update to flag, will it see the update to data?



The Cache Coherence Problem



Cache Coherence

- Write Propagation
 - writes are (eventually) visible in all processors
- Write Serialization
 - Writes are observable in the same order from all processors

// Initially all values are 0.

P1

P2

P3

P4

X= 1

X=2

=X(1)

=X(2)

=X(2)

=X(1)

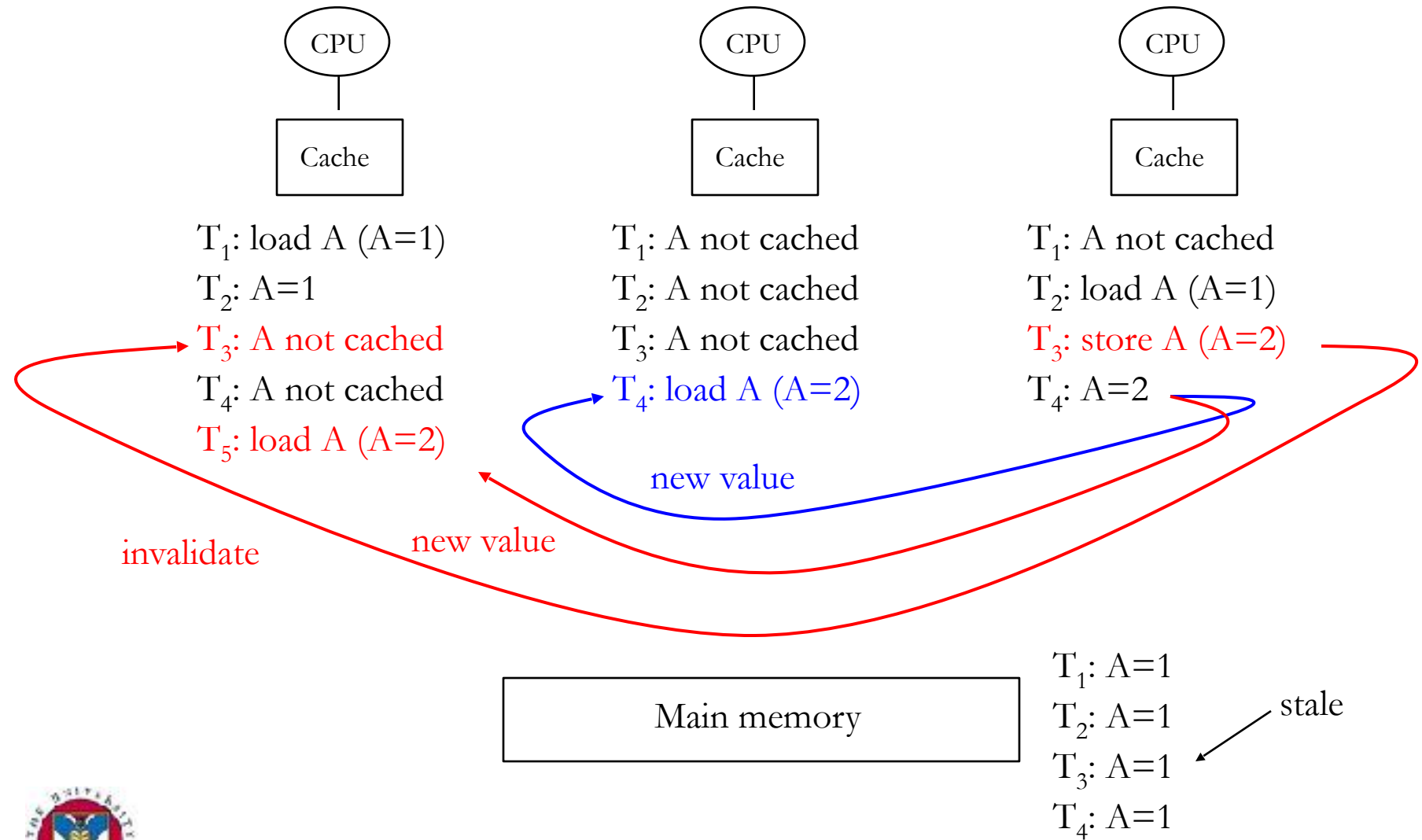


Cache Coherence Protocols

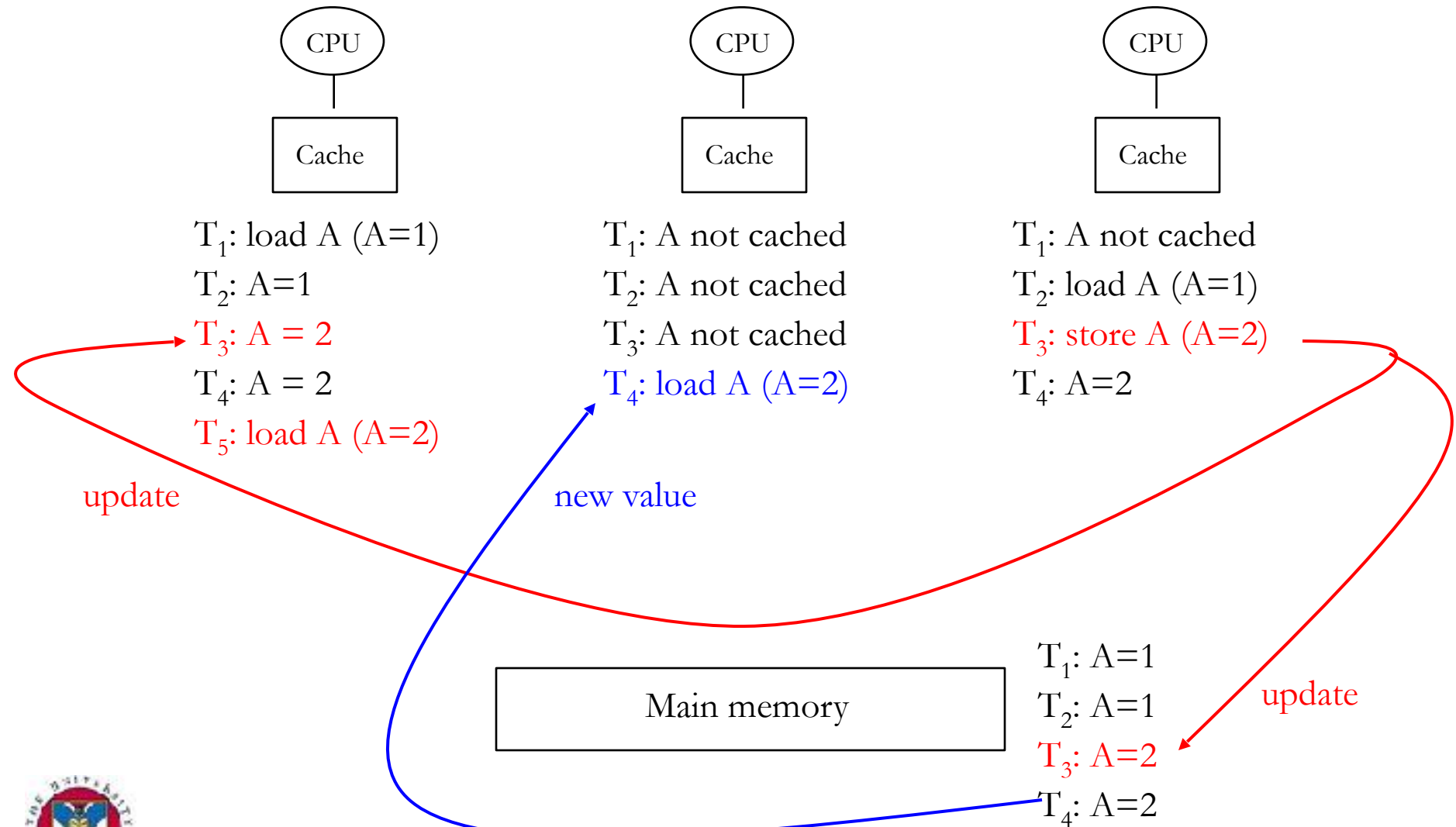
- Idea:
 - Keep track of what processors have copies of what data
 - Enforce that at any given time a single value of every data exists:
 - By getting rid of copies of the data with old values → invalidate protocols
 - By updating everyone's copy of the data → update protocols
- In practice:
 - Guarantee that old values are eventually invalidated/updated (write propagation)
(recall that without synchronization there is no guarantee that a load will return the new value anyway)
 - Guarantee that only a single processor is allowed to modify a certain datum at any given time (write serialization)
 - Must appear as if no caches were present
 - Note: must fit with cache's operation at the granularity of lines



Write-invalidate Example



Write-update Example



Invalidate vs. Update Protocols

■ Invalidate:

- + Multiple writes by the same processor to the cache block only require one invalidation
 - + No need to send the new value of the data (less bandwidth)
 - Caches must be able to provide up-to-date data upon request
 - Must write-back data to memory when evicting a modified block
- Usually used with write-back caches (more popular)

■ Update:

- + New value can be re-used without the need to ask for it again
 - + Data can always be read from memory
 - + Modified blocks can be evicted from caches silently
 - Possible multiple useless updates (more bandwidth)
- Usually used with write-through caches (less popular)



Cache Coherence Protocols

- Implementation can either be in software or hardware.
- Software coherence
 - Expose *writeback* and *self-invalidate* to software
 - Insert these at appropriate points by leveraging static analysis.
 - Problem: conservatism of static analysis
- Hardware coherence
 - Add state bits to cache lines to track state of the line
 - Most common: Modified, Owned, Exclusive, Shared, Invalid
 - Protocols usually named after the states supported
 - Cache lines transition between states upon load/store operations from the local processor and by remote processors
 - These state transitions must guarantee:
 - write propagation and
 - write serialization: no two cache copies can be simultaneously modified (SWMR: Single writer multiple readers)



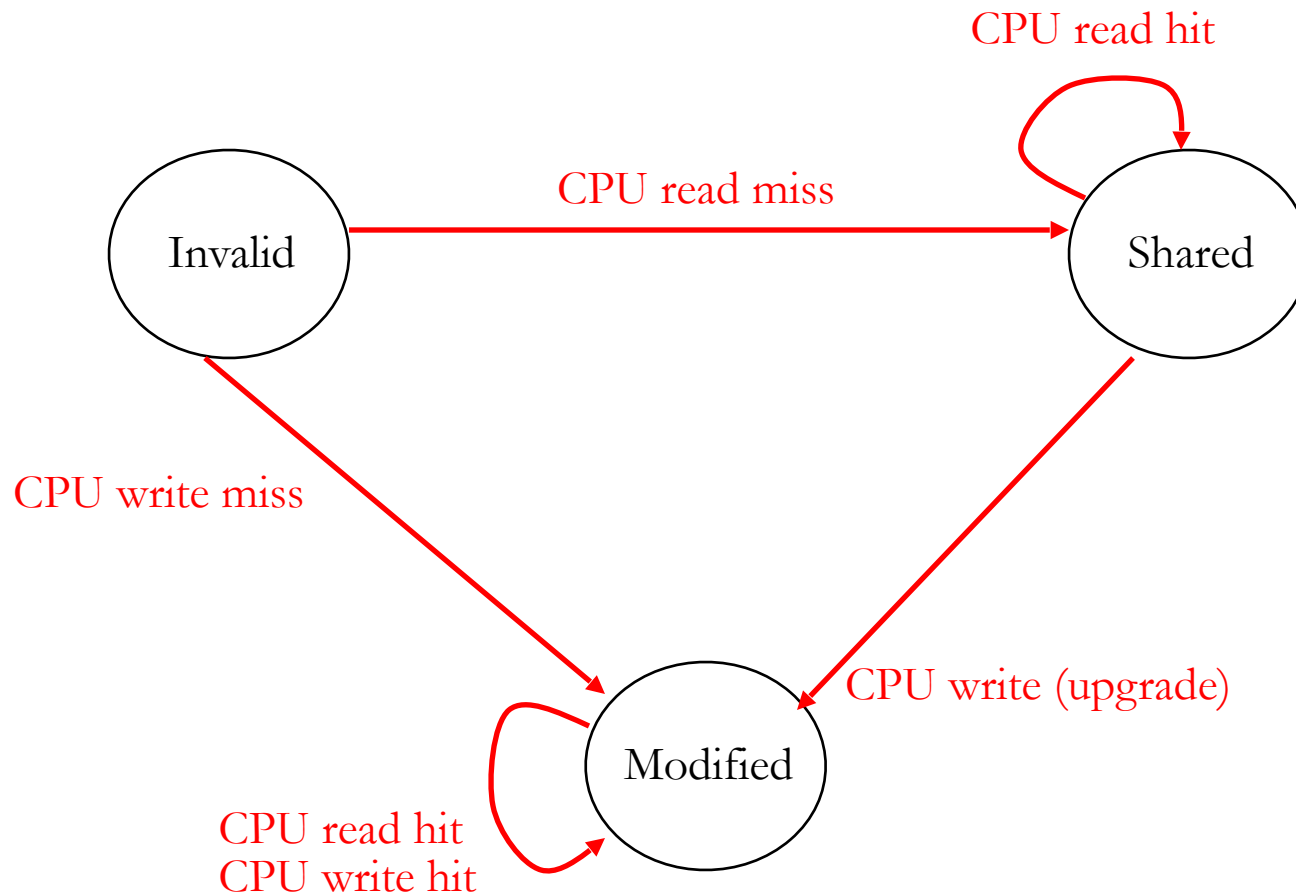
Example: MSI Protocol

- States:
 - **Modified (M)**: block is cached only in this cache and has been modified
 - **Shared (S)**: block is cached in this cache and possibly in other caches (no cache can modify the block)
 - **Invalid (I)**: block is not cached



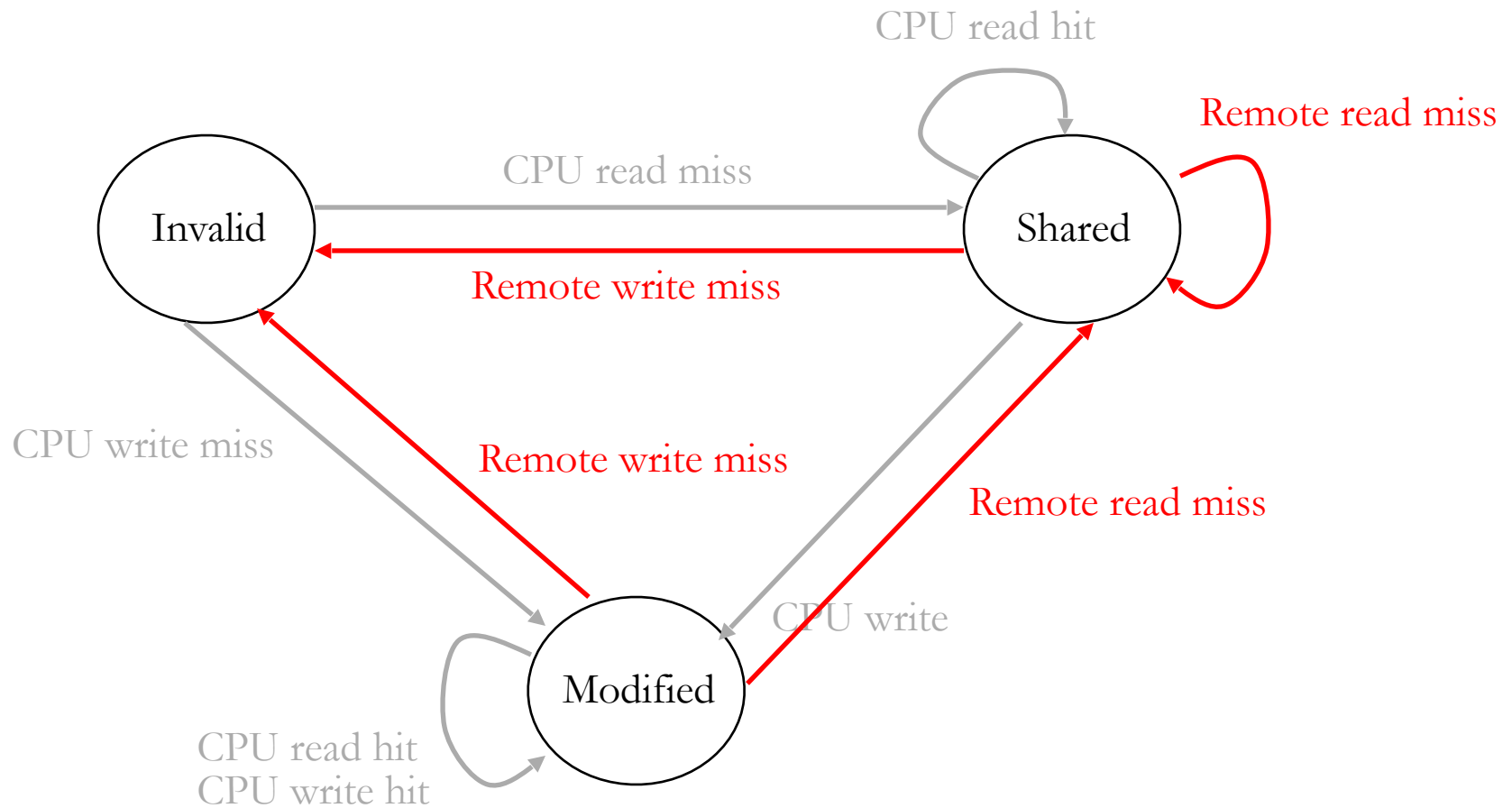
Example: MSI Protocol

- Transactions originated at this CPU:



Example: MSI Protocol

- Transactions originated at other CPU:



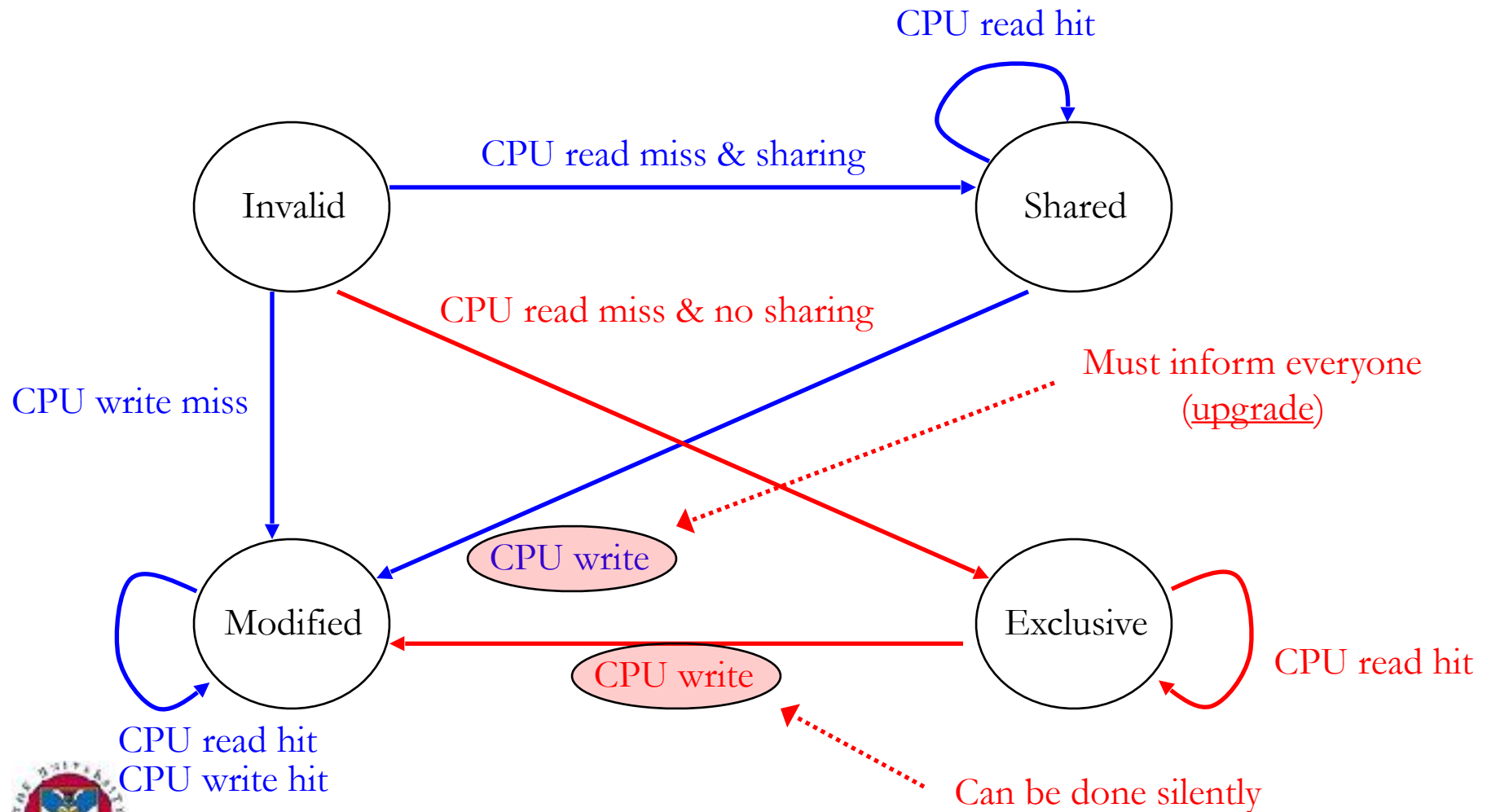
Example: MESI Protocol

- States:
 - Modified (M): block is cached only in this cache and has been modified
 - **Exclusive (E)**: block is cached only in this cache, has not been modified, but can be modified at will
 - Shared (S): block is cached in this cache and possibly in other caches
 - Invalid (I): block is not cached
- State E is obtained on reads when no other processor has a shared copy
 - All processors must answer if they have copies or not
 - Or some device must know if processors have copies
- Advantage over MSI
 - Often variables are loaded, modified in register, and then stored
 - The store on state E then does not require asking for permission to write



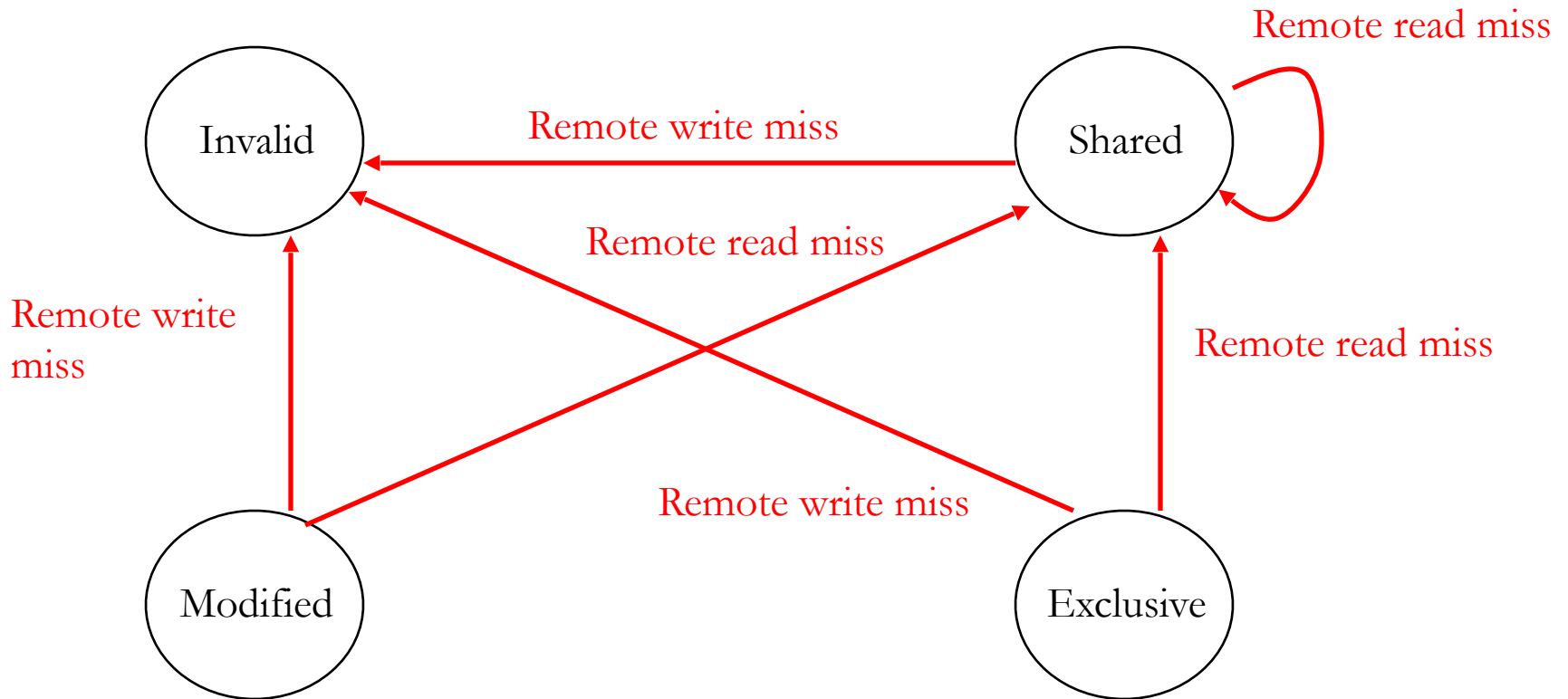
Example: MESI Protocol

- Transactions originated at this CPU:



Example: MESI Protocol

- Transactions originated at other CPU:



Possible Implementations

- Two ways of implementing coherence protocols in hardware
 - Snooping: all cache controllers monitor all other caches' activities and maintain the state of their lines
 - Commonly used with buses and in many CMP's today
 - Directory: a central control device directly handles all cache activities and tells the caches what transitions to make
 - Can be of two types: physically centralized and physically distributed
 - Commonly used with scalable interconnects and in many CMP's today



Behavior of Cache Coherence Protocols

- Uniprocessor cache misses (the 3 C's):
 - Cold (or compulsory) misses: when a block is accessed for the first time
 - Capacity misses: when a block is not in the cache because it was evicted because the cache was full
 - Conflict misses: when a block is not in the cache because it was evicted because the cache set was full
- Coherence misses: when a block is not in the cache because it was invalidated by a write from another processor
 - Hard to reduce → relates to intrinsic communication and sharing of data in the parallel application
 - False sharing coherence misses: processors modify different words of the cache block (no real communication or sharing) but end up invalidating the complete block



Behavior of Cache Coherence Protocols

- False sharing misses increases with larger cache line size
 - Only true sharing remains with single word/byte cache lines
- False sharing misses can be reduced with better placement of data in memory
- True sharing misses tends to decrease with larger cache line sizes (due to locality)
- Classifying misses in a multiprocessor is not straightforward
 - E.g., if P0 has line A in the cache and evicts it due to capacity limitation, and later P1 writes to the same line: is this a capacity or a coherence miss?
It is both, as fixing one problem (e.g., increasing cache size) won't fix the other (see Figure 5.20 of Culler&Singh for a complete decision chart)



Behavior of Cache Coherence Protocols

- Common types of data access patterns
 - Private: data that is only accessed by a single processor
 - Read-only shared: data that is accessed by multiple processors but only for reading (this includes instructions)
 - Migratory: data that is used and modified by multiple processors, but in turns
 - Producer-consumer: data that is updated by one processor and consumed by another
 - Read-write: data that is used and modified by multiple processors simultaneously
 - Data used for synchronization

- **Bottom-line: threads don't usually read and write the same data indiscriminately**



References and Further Reading

- “Parallel Computer Architecture”, David E. Culler, Jaswinder Pal Singh (Chapter 5).
- “A Primer on Memory Consistency and Cache Coherence”, Daniel J. Sorin, Mark D. Hill, David A. Wood

