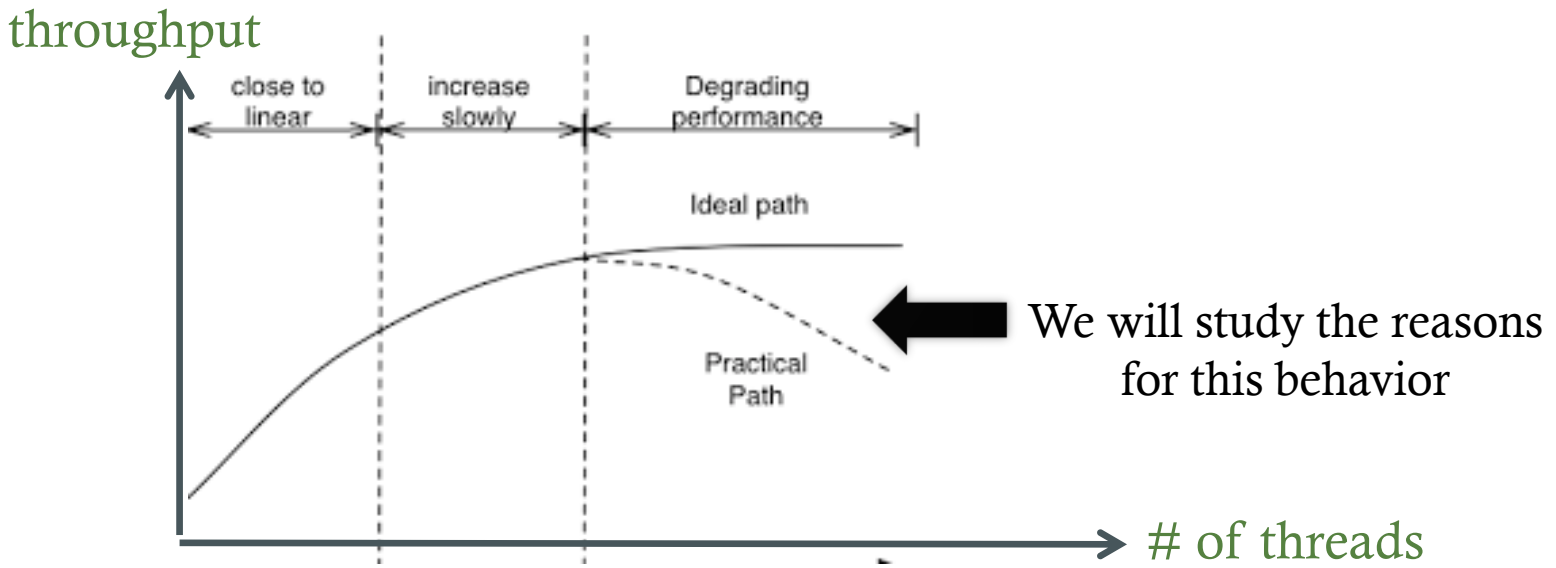
ECE 454 Computer Systems Programming

Performance Implications of Parallel Architectures

Ashvin Goel, Ding Yuan
ECE Dept, University of Toronto

Big Picture

- We know that we need parallelization
- But will more parallelization always yield better performance?



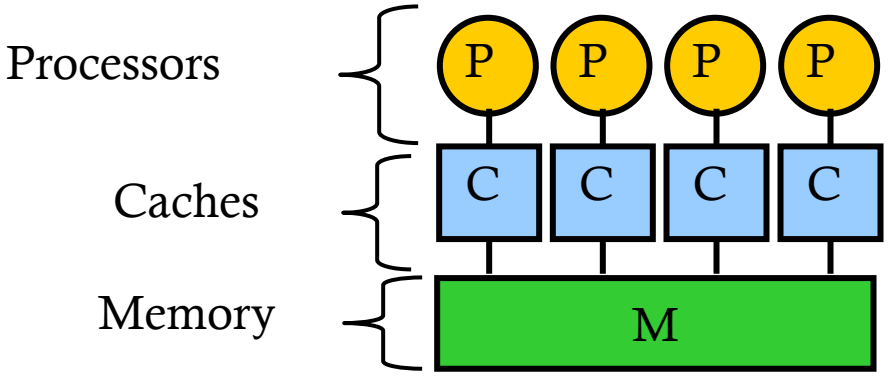
Topics

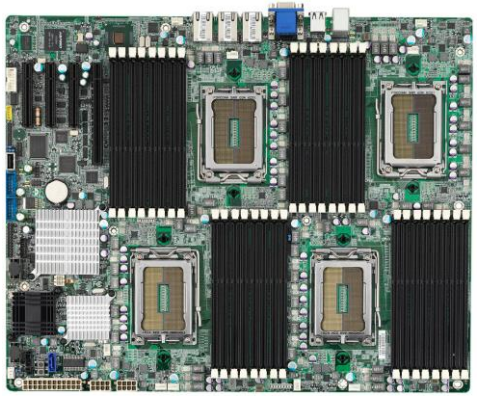
- Cache coherence
- Performance of memory operations
- Implications for software design

Cache Coherence

Modern Shared Memory Parallel Architectures

- Provide several processing elements (cores or processors)
- Provide shared memory
  - Any processor can directly reference any memory location
  - Communication occurs implicitly through loads and stores
- Cores have private caches to improve performance



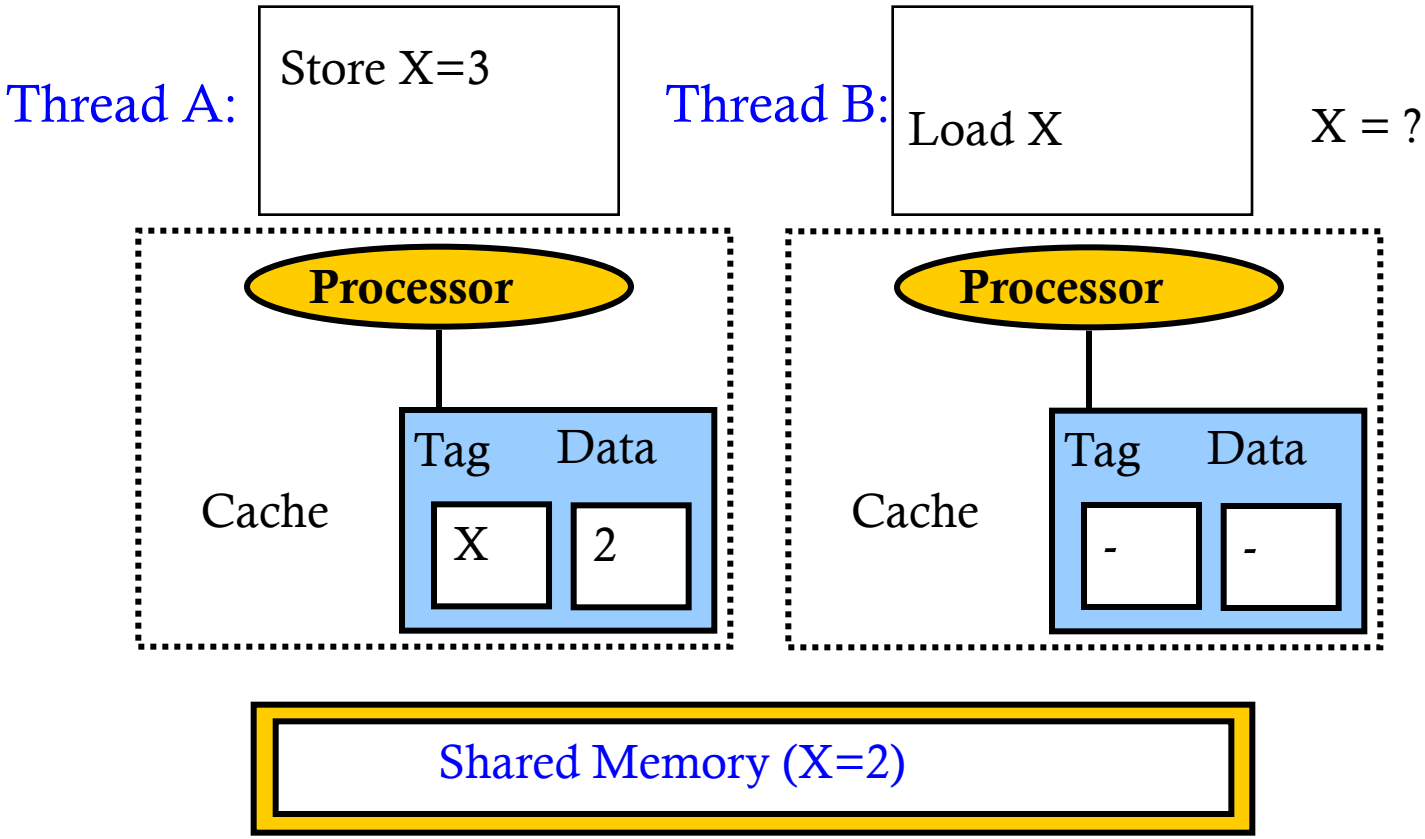


48 core AMD Opteron

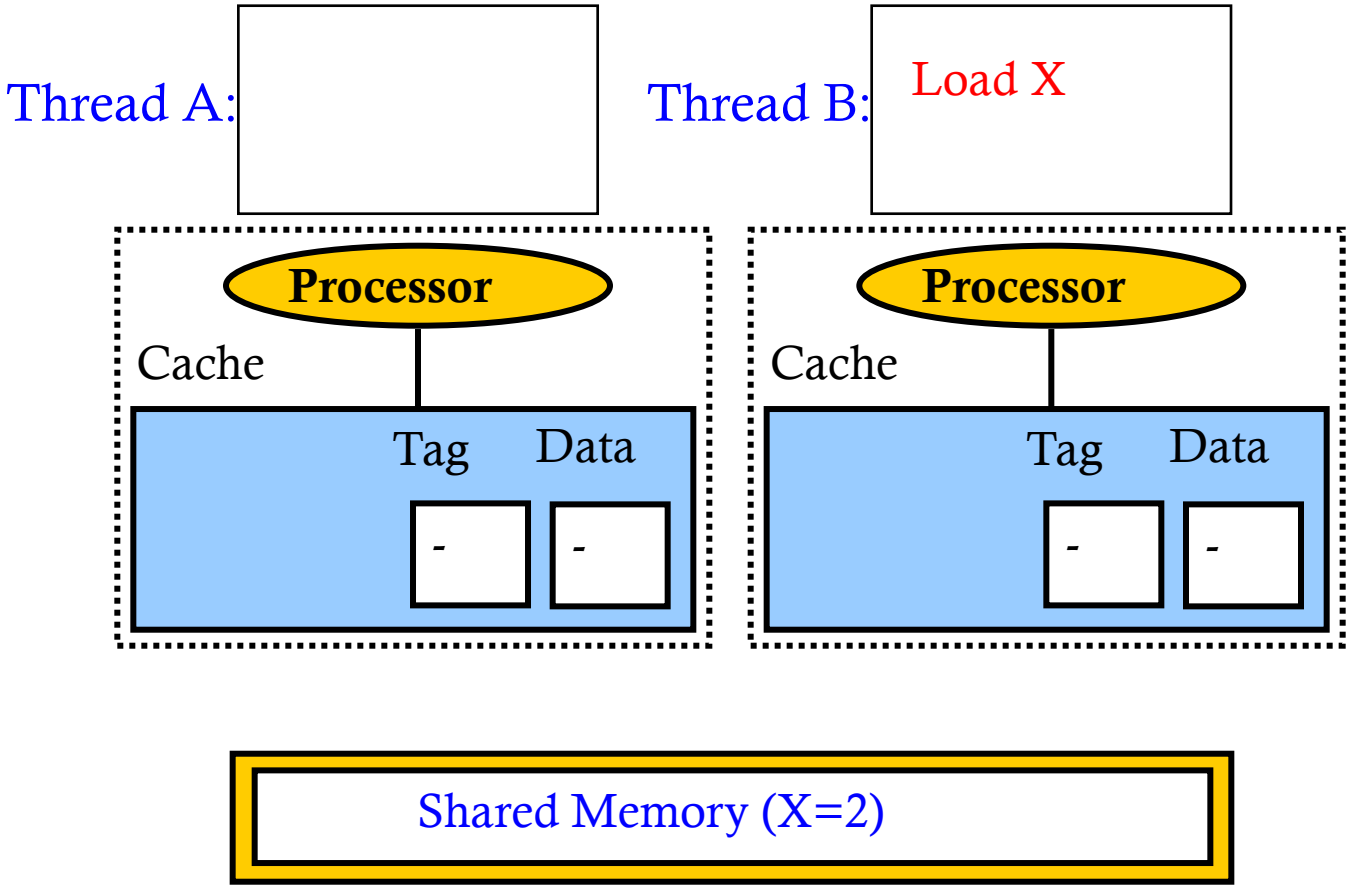
Cache Coherence Problem

- With multiple cores, data is cached in multiple locations, so how do you ensure consistency?

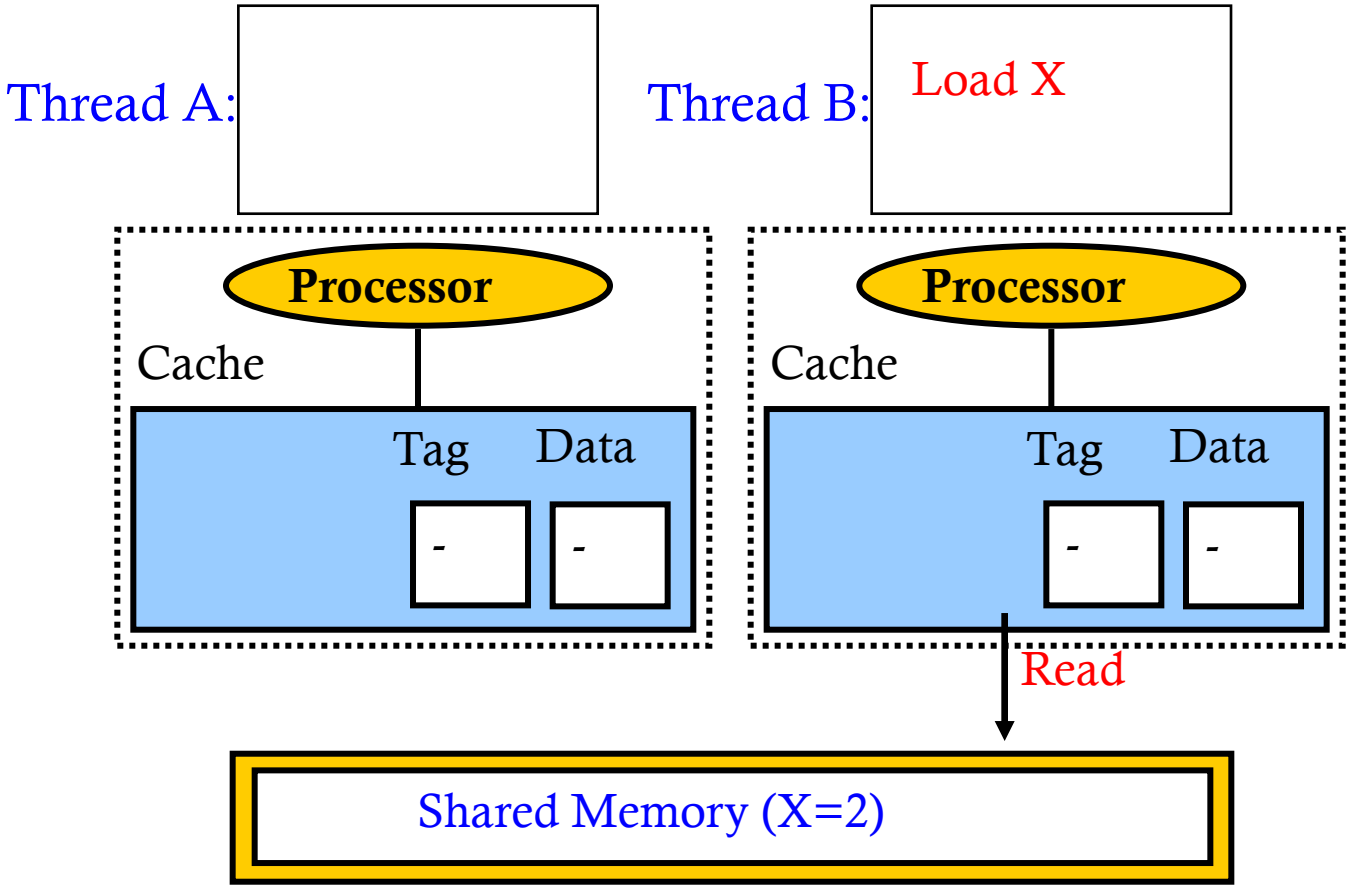
Example 1: Coherence Problem



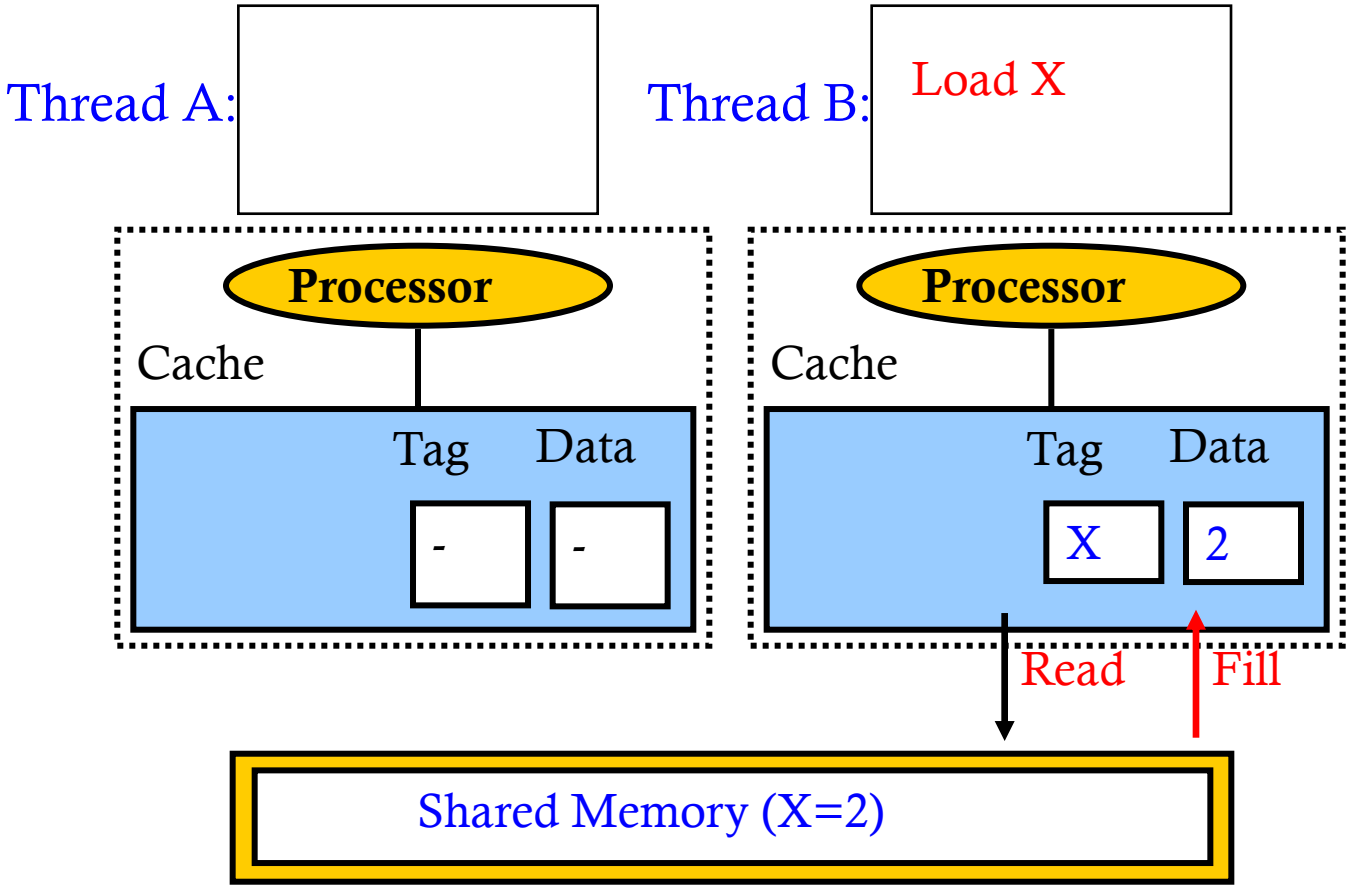
Example 2: Coherence Problem



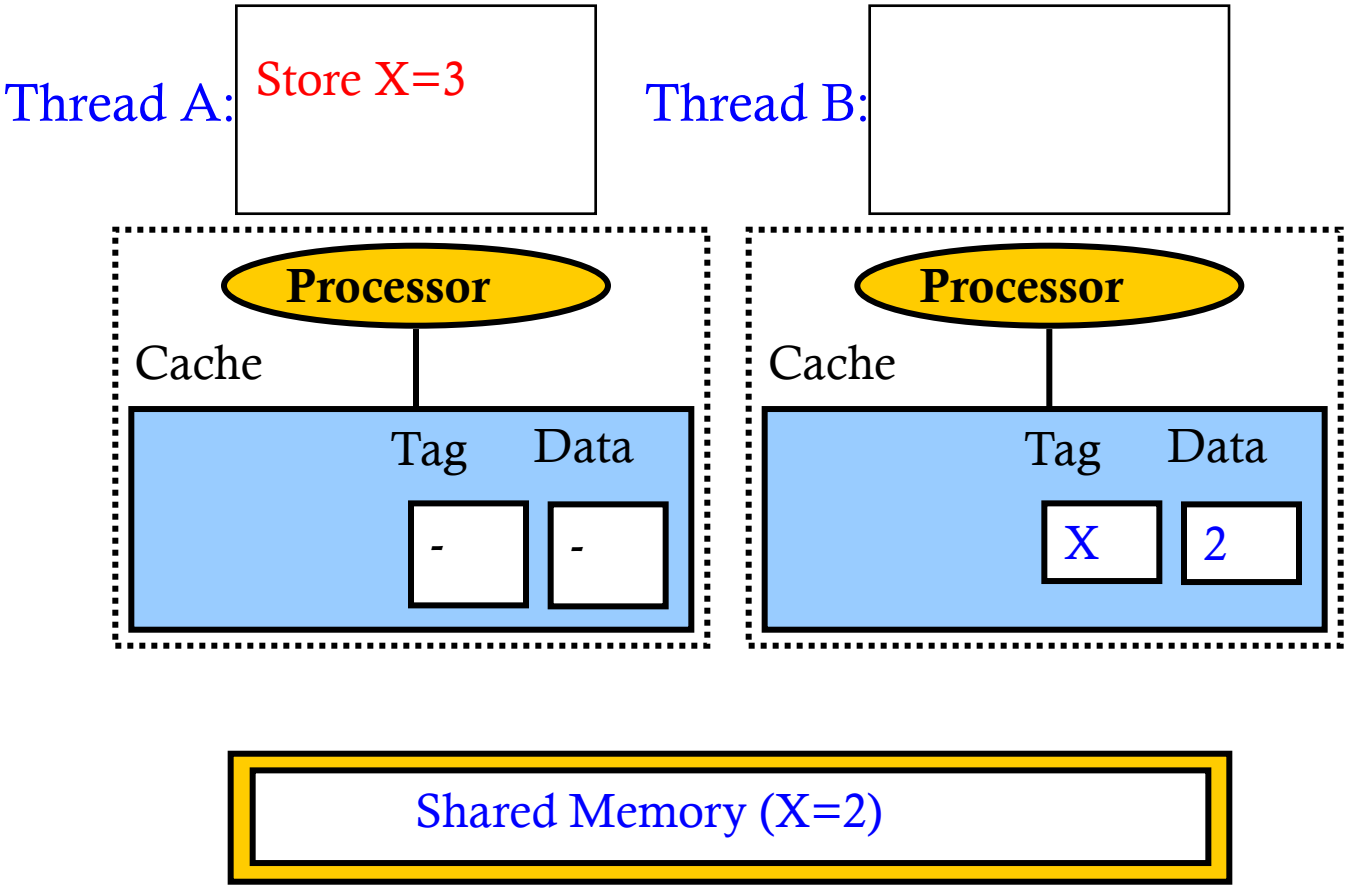
Example 2: Coherence Problem



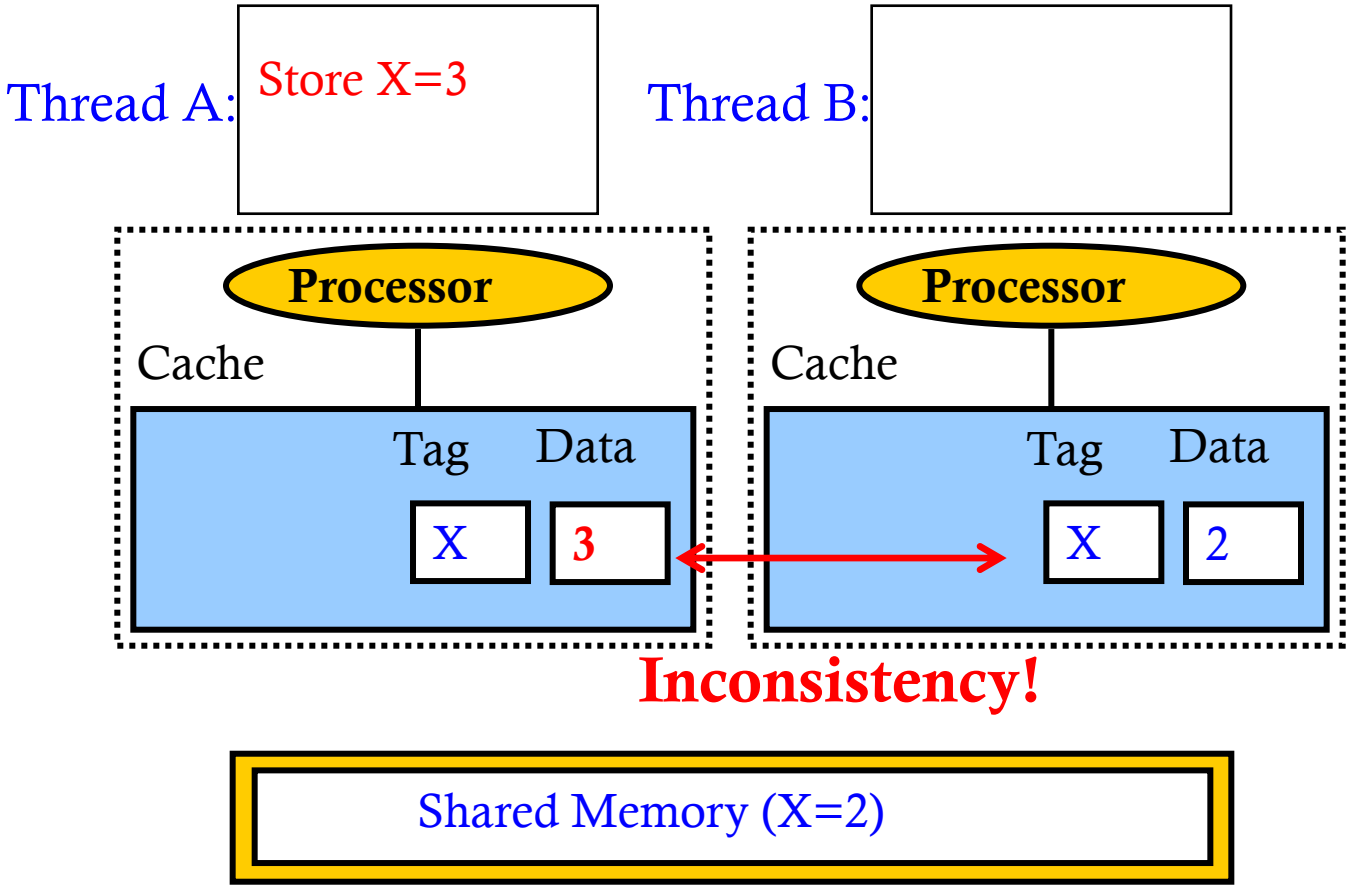
Example 2: Coherence Problem



Example 2: Coherence Problem

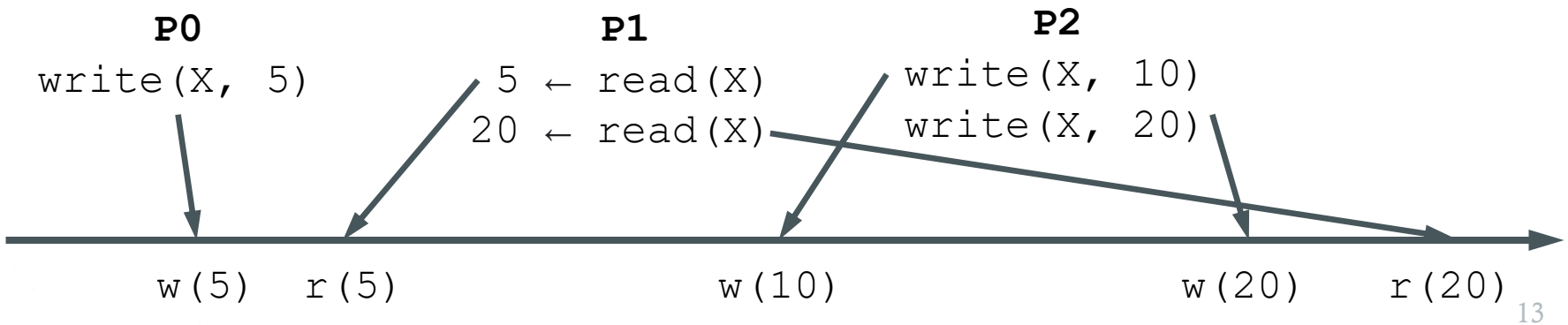


Example 2: Coherence Problem



Cache (or Memory) Coherence

- The behavior of the system is equivalent to there being only a single copy of the data except for the performance benefit of the cache. [Gray and Cheriton 83]
- Cache coherence ensures that all processors have a consistent view of a single memory location (e.g., X)
  - All loads and stores to X can be put on a timeline (total order) that respects the program order of loads and stores of each processor



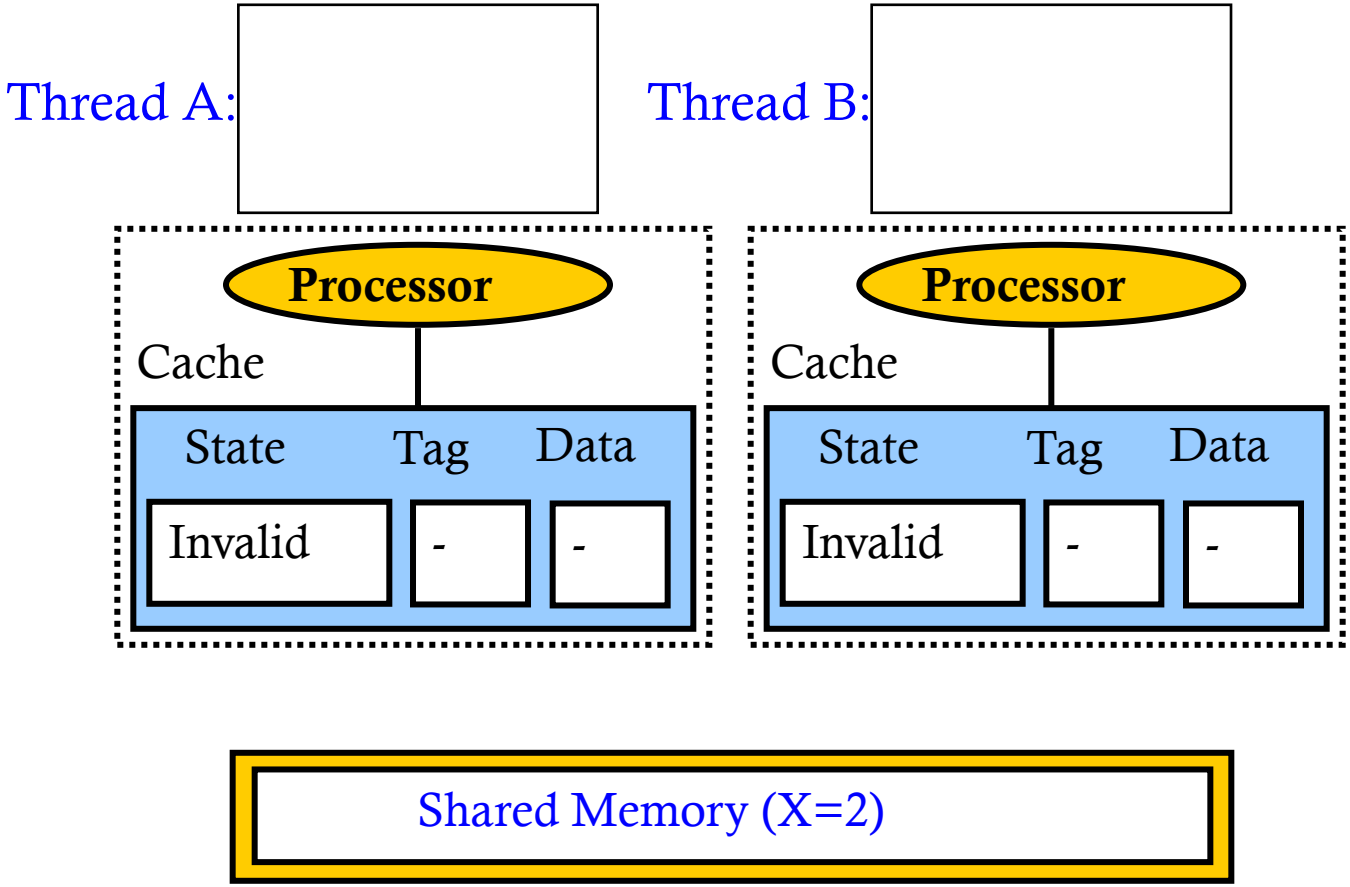
Why Cache Coherence?

- With non-cache coherent machines, e.g., Intel Rack Scale, The Machine from HP, loads and stores to the same location are not synchronized
  - Loads may read stale data, i.e., store is not visible to later load
  - Stores are not sequenced, i.e., stores visible in different orders
  - Really complicates the programming model

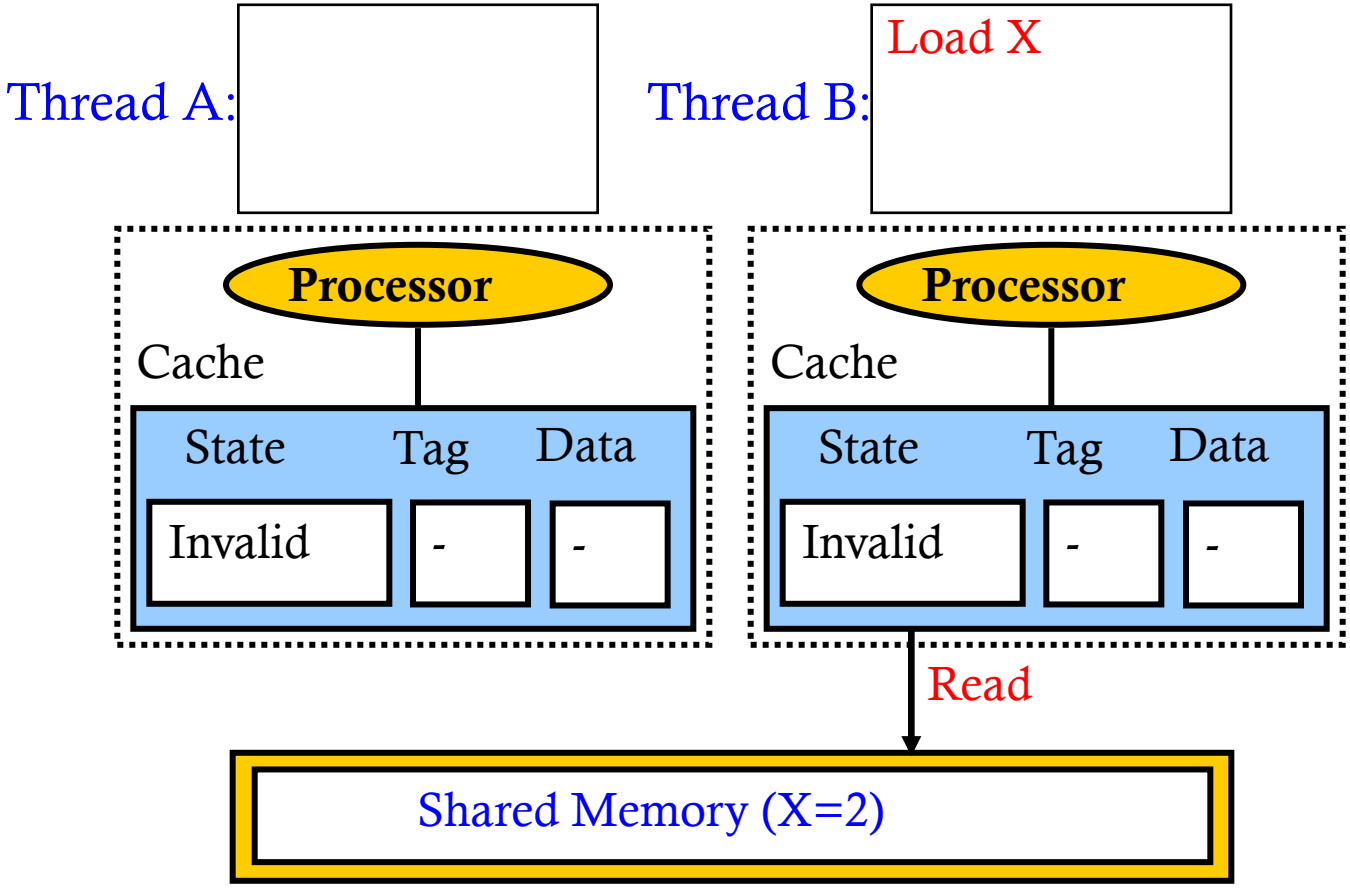
MSI Coherence Protocol

- Ensuring coherence requires hardware support
  - Called coherence protocol
- Add three states to each cache line (on each core):
  - Invalid – data is not cached
  - Modified – core has written to the cache line
    - Cache line is inconsistent with primary storage
    - Cache line is not shared with other cores
  - Shared – core has read from the cache line
    - Cache line is consistent with primary storage
    - Cache line may be shared with other cores

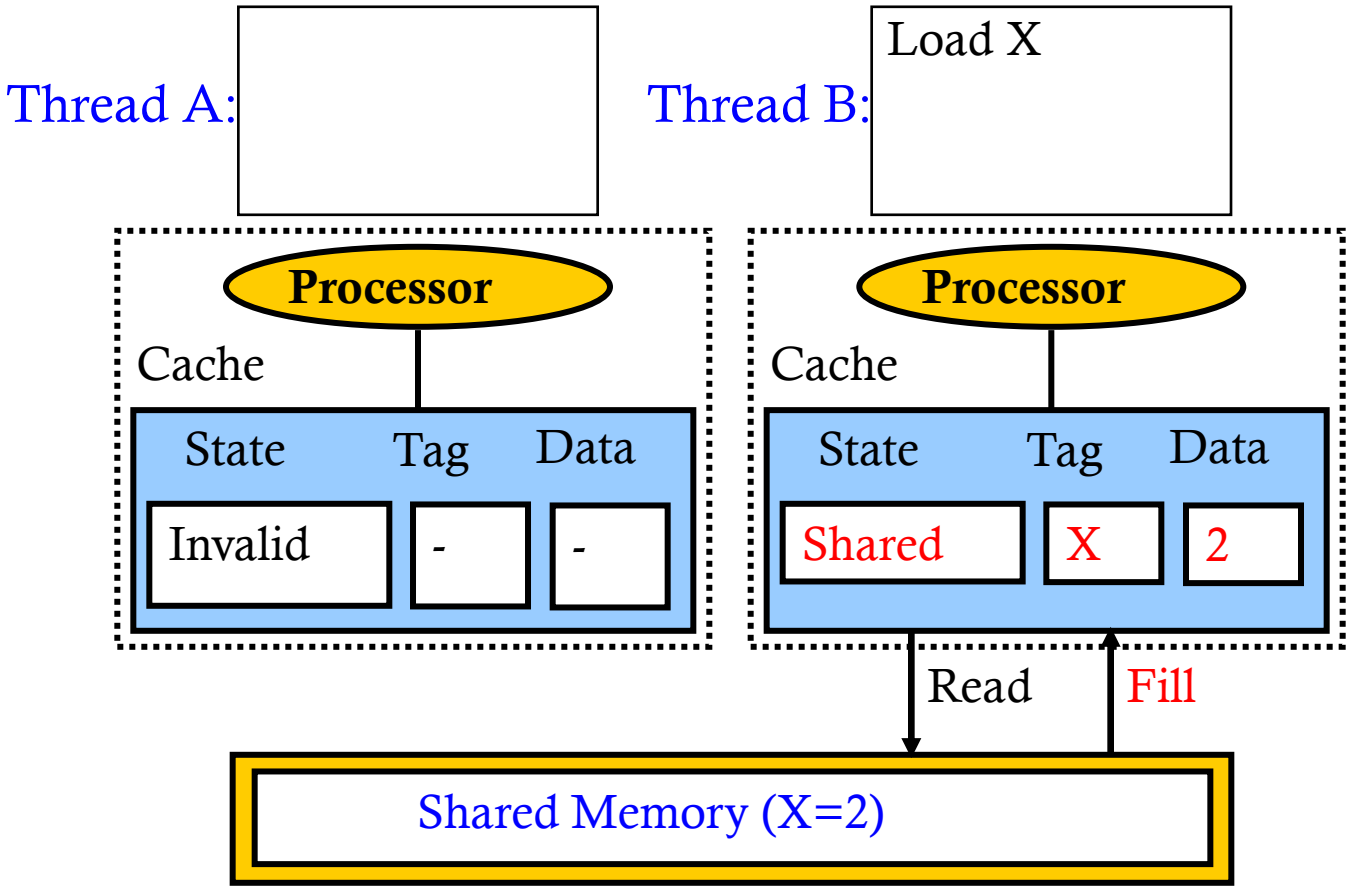
MSI Coherence Protocol



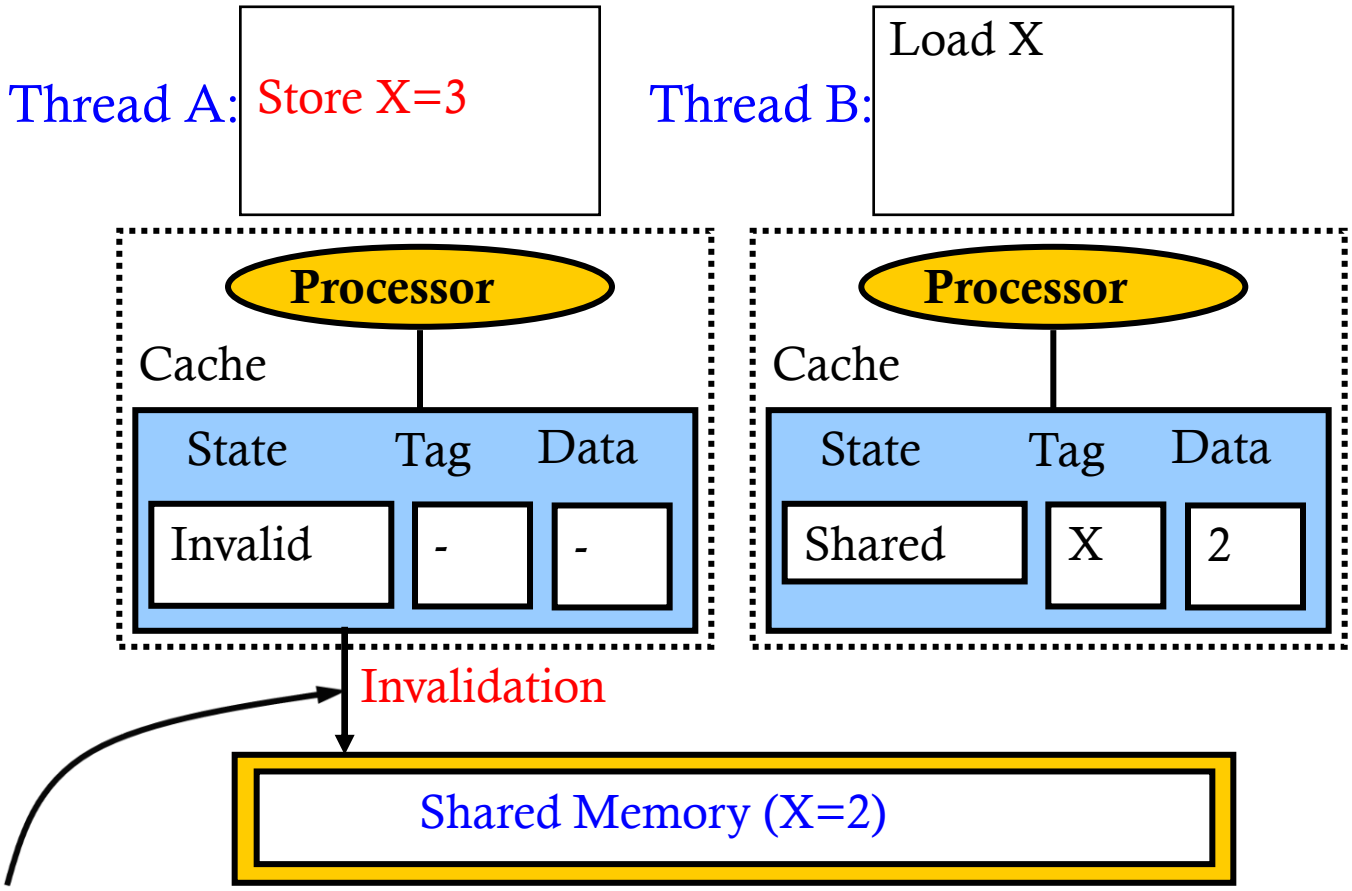
MSI Coherence Protocol



MSI Coherence Protocol

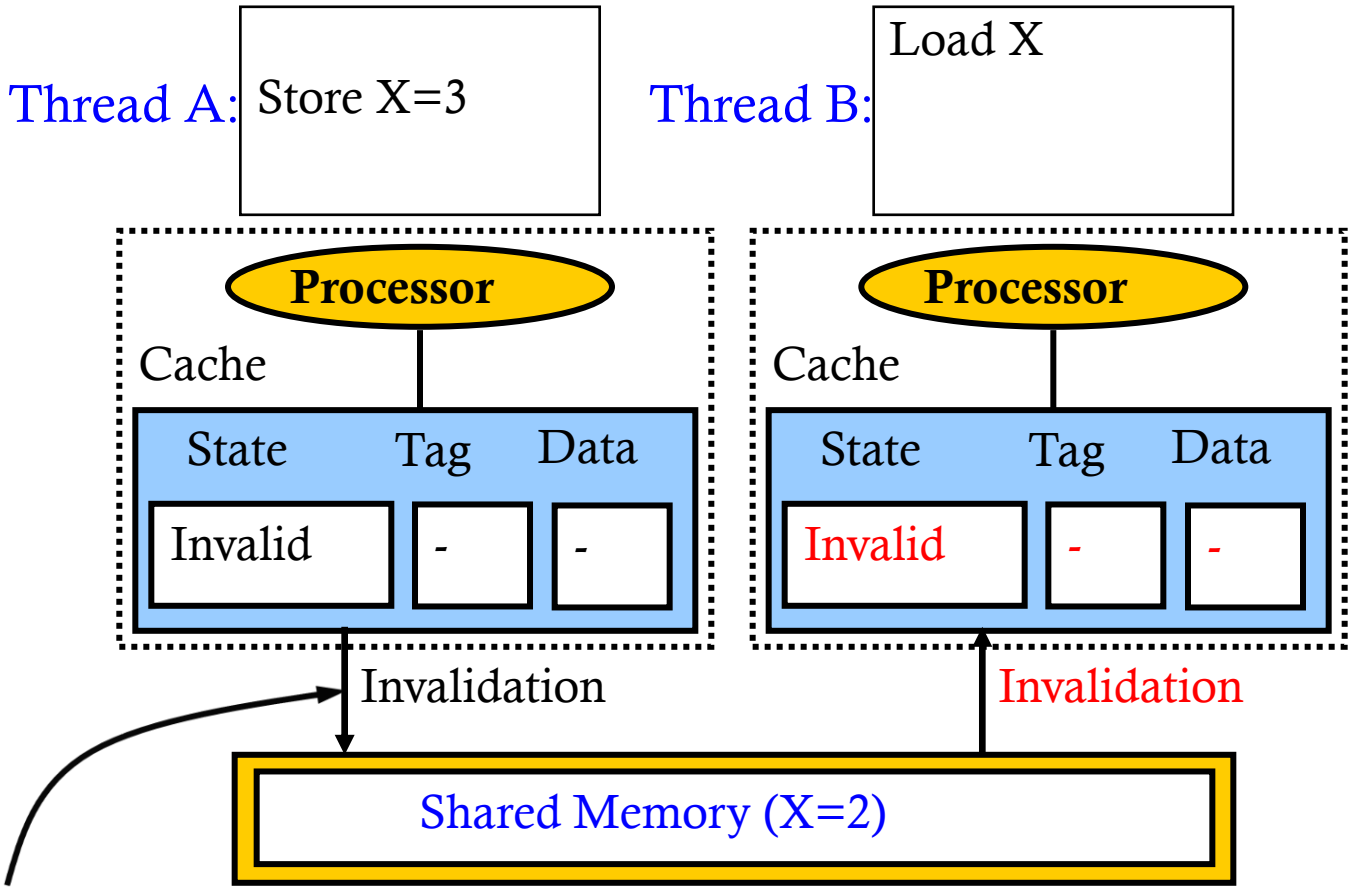


MSI Coherence Protocol



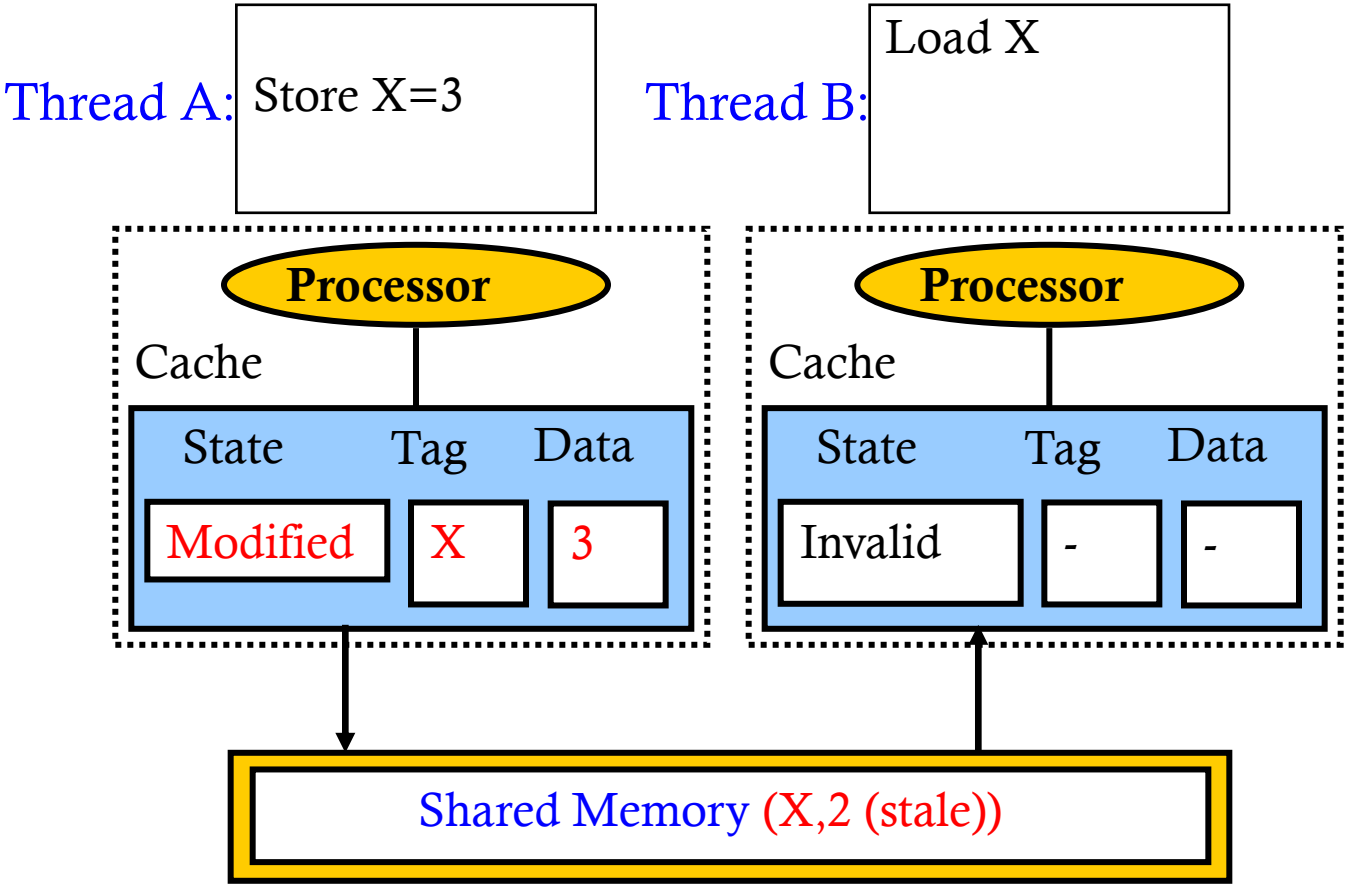
invalidates all other copies

MSI Coherence Protocol



invalidates all other copies

MSI Coherence Protocol



Problem with MSI

- If a core reads a value that is not cached on any core and then writes to the value, then two cache coherence requests are generated
  - A request to read the value (required)
  - A request to write the value (unnecessary invalidation request sent because the MSI protocol doesn't know that no one else has a copy)

MESI (aka Illinois) Protocol

- Four (exclusive) states of each cache line:
  - Invalid – data is not cached
  - Modified – core has written to the cache line
    - Cache line is inconsistent with primary storage
    - Cache line is not shared with other cores
  - Shared – core has read from the cache line
    - Cache line is consistent with primary storage
    - Cache line may be shared with other cores
  - Exclusive: core has read from the cache line
    - Cache line is consistent with primary storage
    - Cache line is not shared by other cores
    - Write to Exclusive state does not generate invalidation request

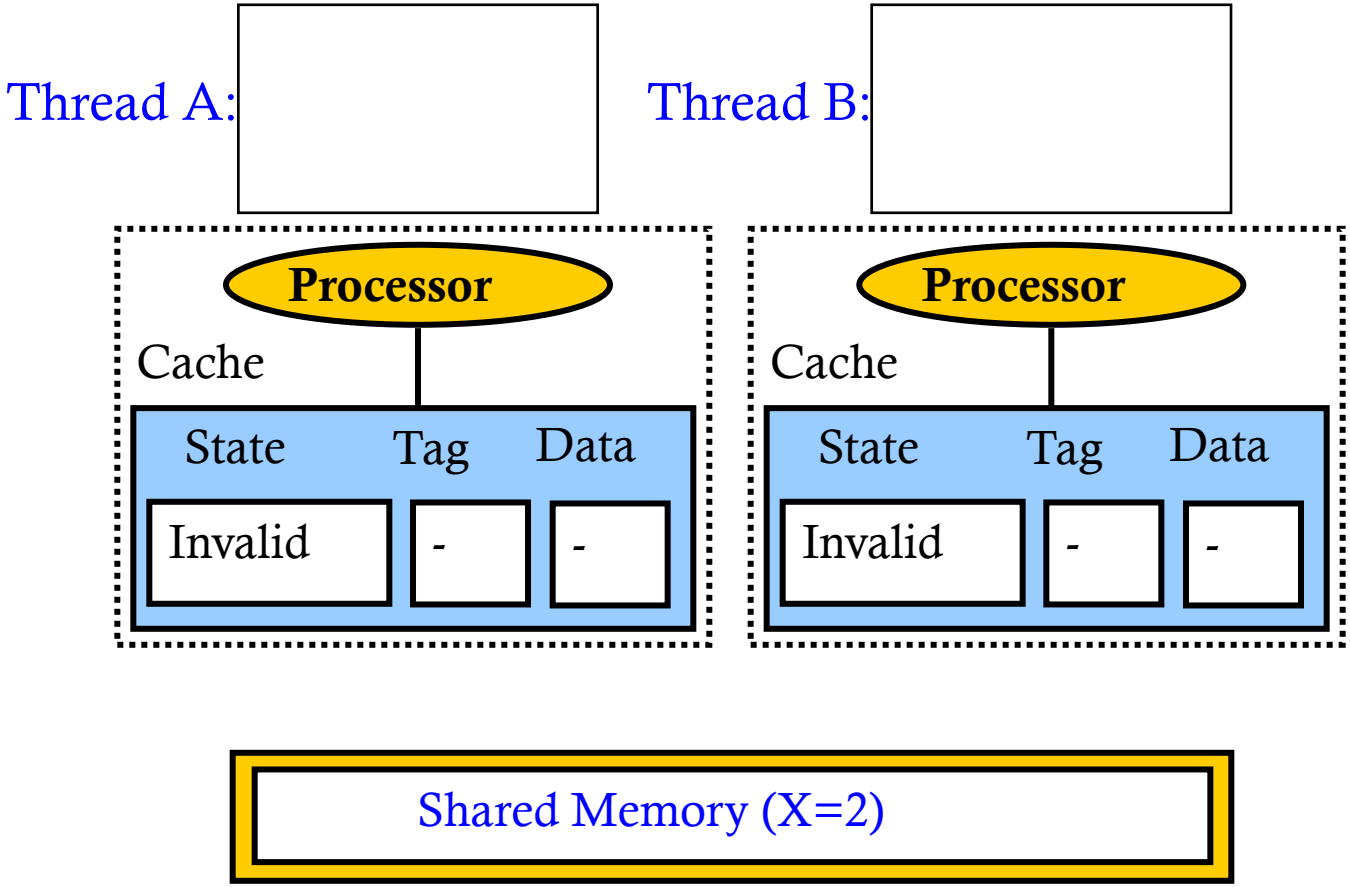
MESI Details: Writing

- An attempt to write to a block that is in `Invalid` state is called a write miss
  - Must cache the block in `Exclusive` state before writing to it
  - Generates a read-exclusive (or read for ownership) request
    - Read + invalidation broadcast (intent to write to memory address)
    - If other caches have copy of data, they send it, invalidate their copy
    - Completes when there are no more valid copies
    - Marks local cache line in exclusive state
  - Can then perform the write and enter the `modified` state
    - This step doesn't require invalidation request

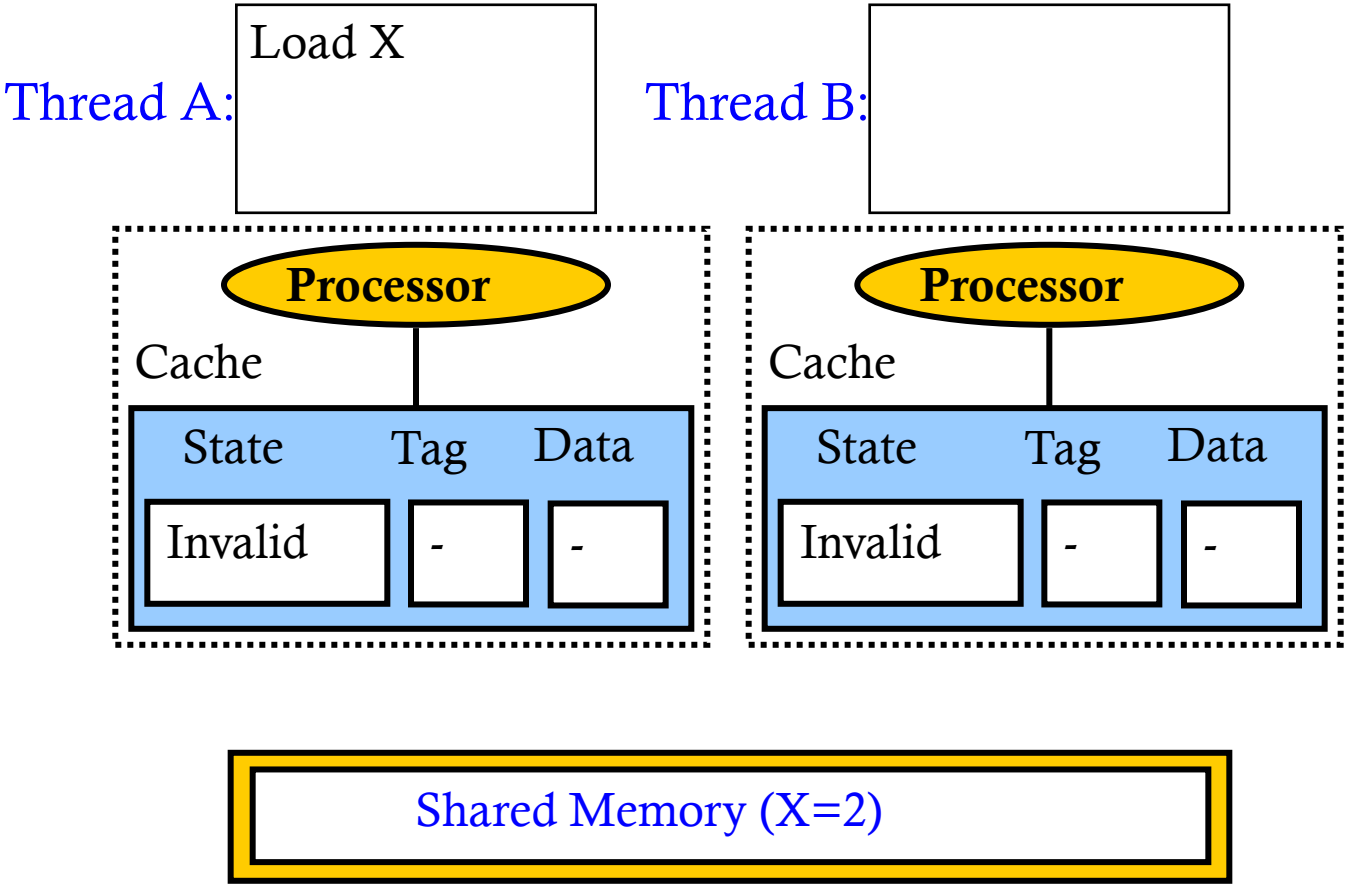
MESI Examples

- Example 1: load on one core followed by load on another core
- Example 2: load on one core followed by store on another core
- Example 3: store on one core followed by load on another core

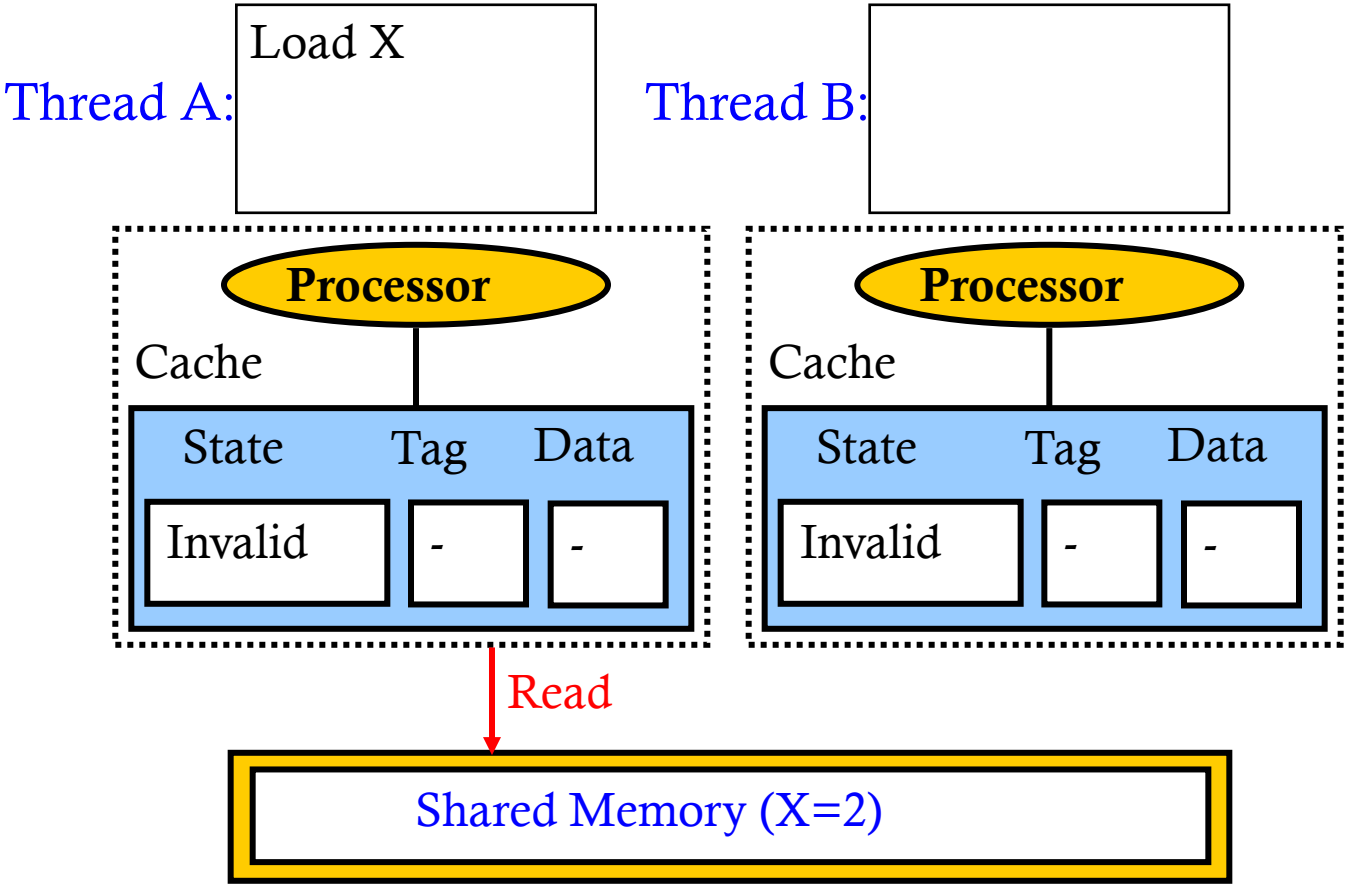
Example 1: MESI Coherence



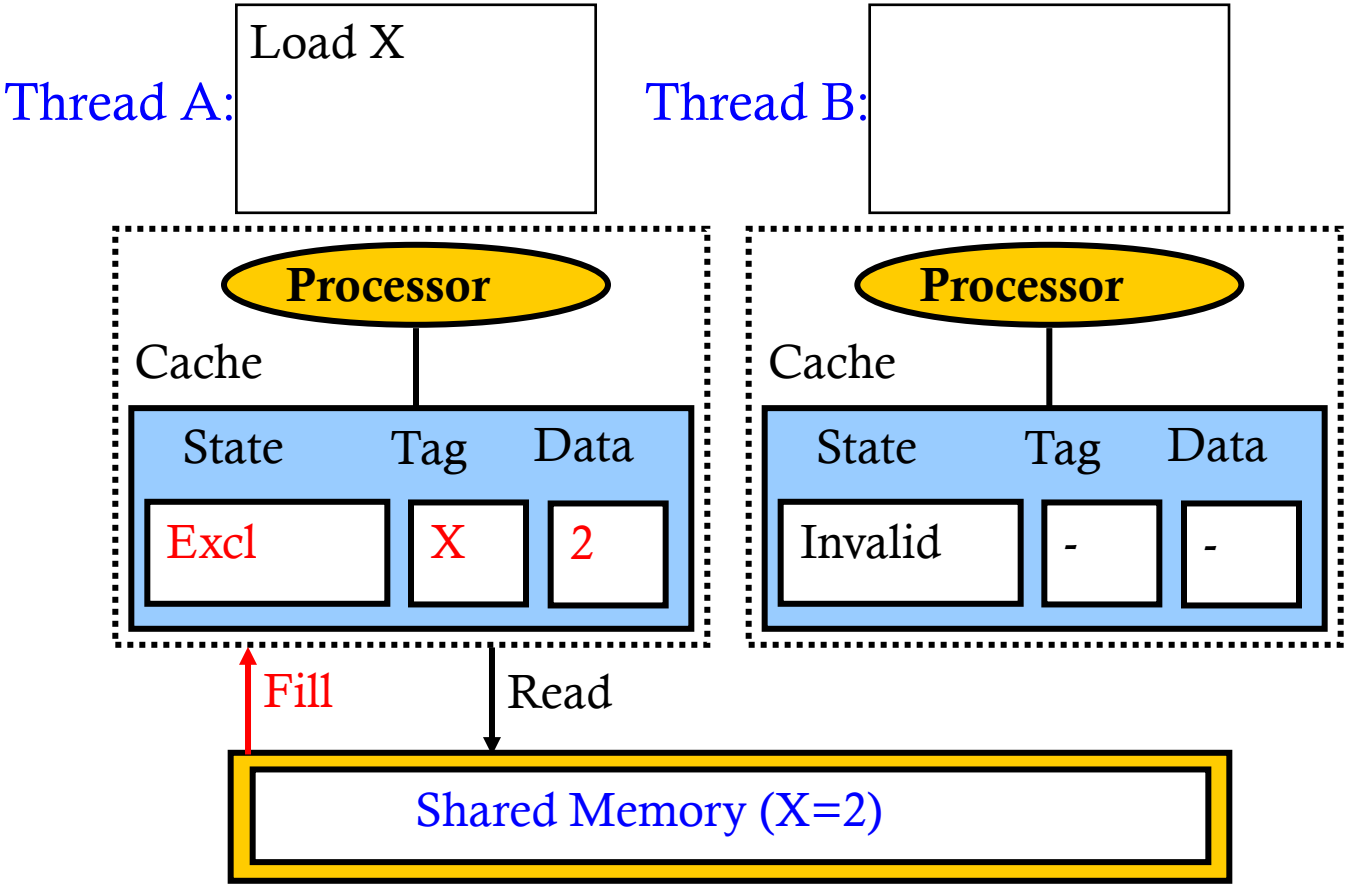
Example 1: MESI Coherence



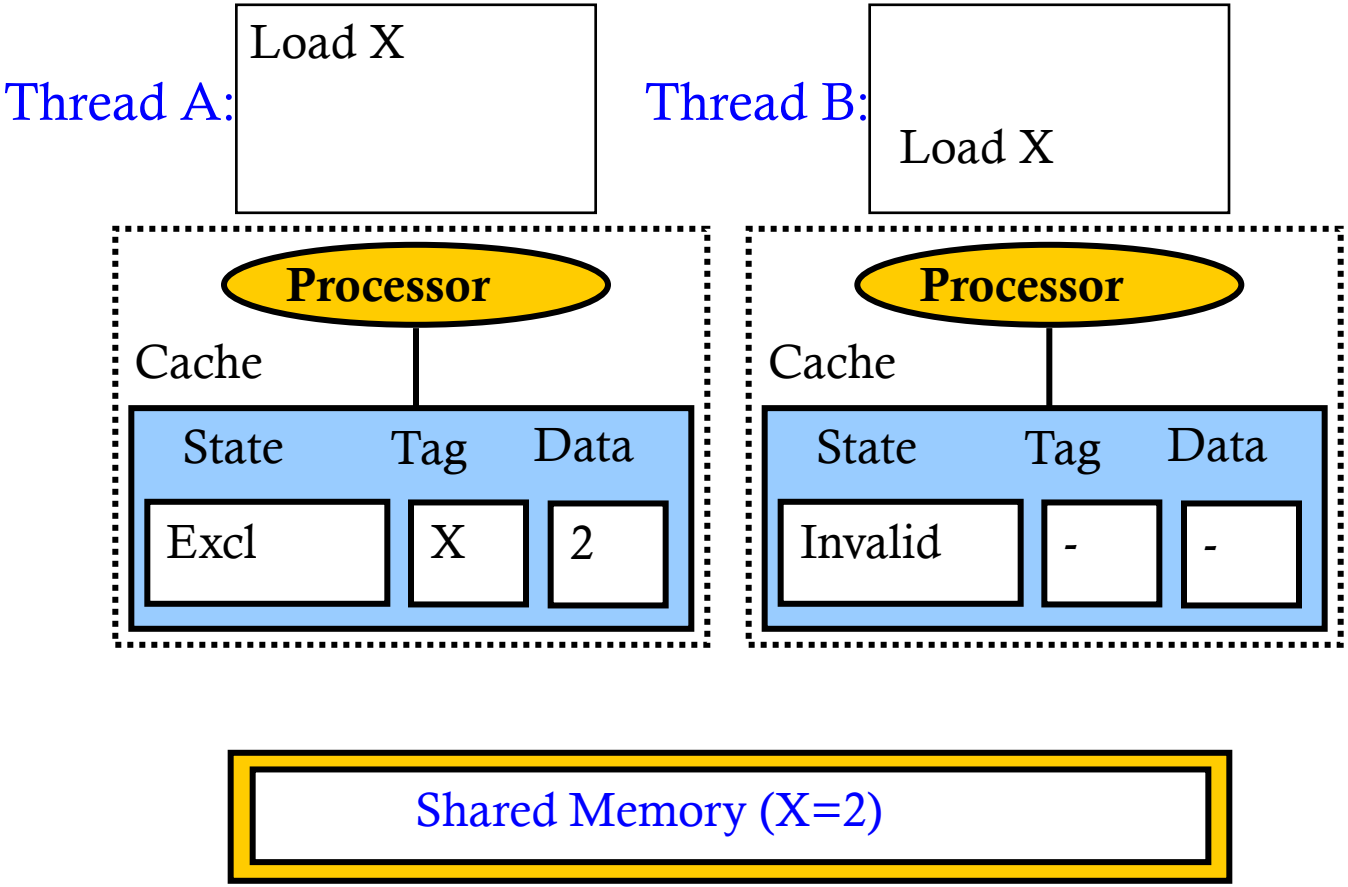
Example 1: MESI Coherence



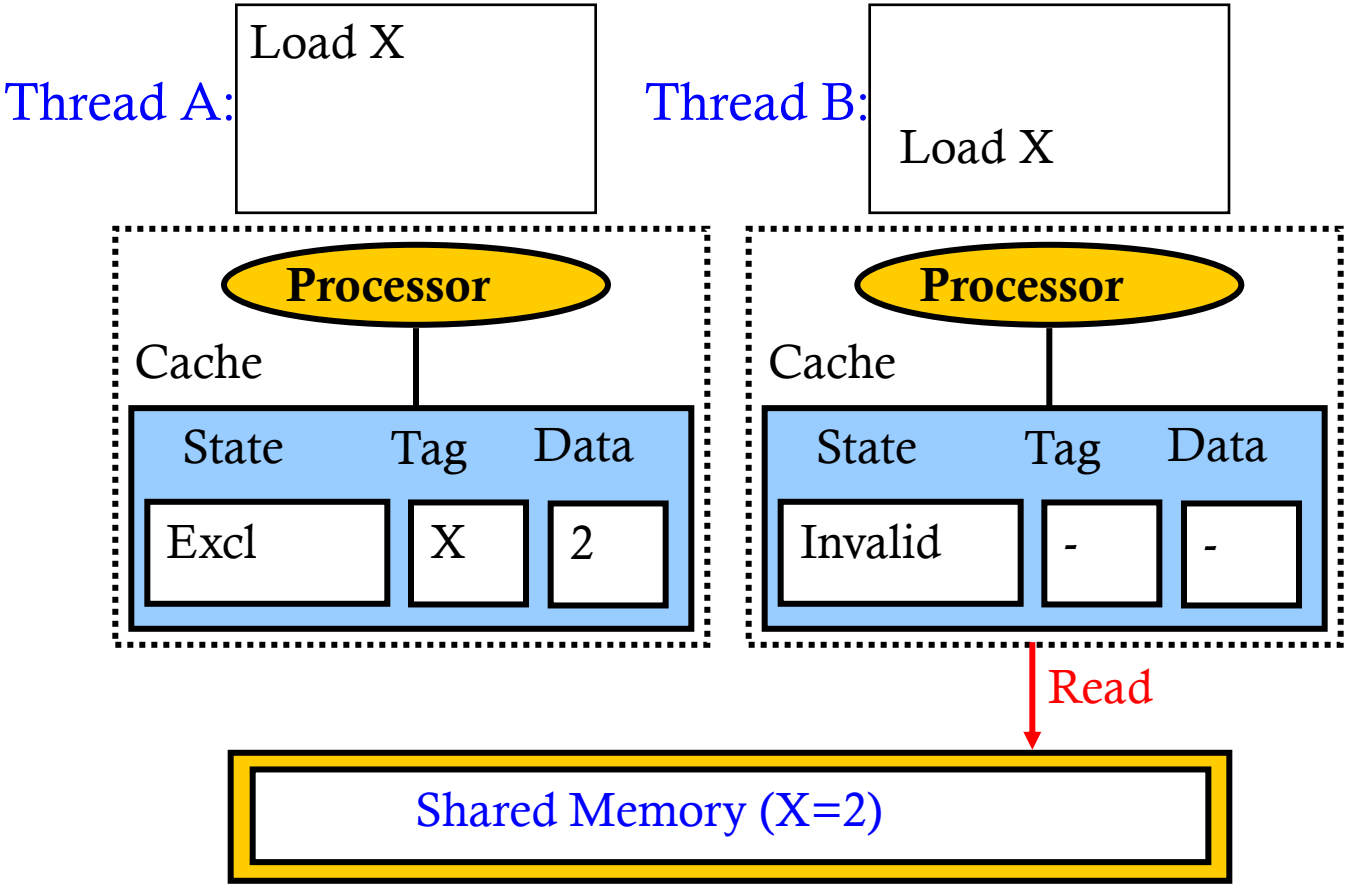
Example 1: MESI Coherence



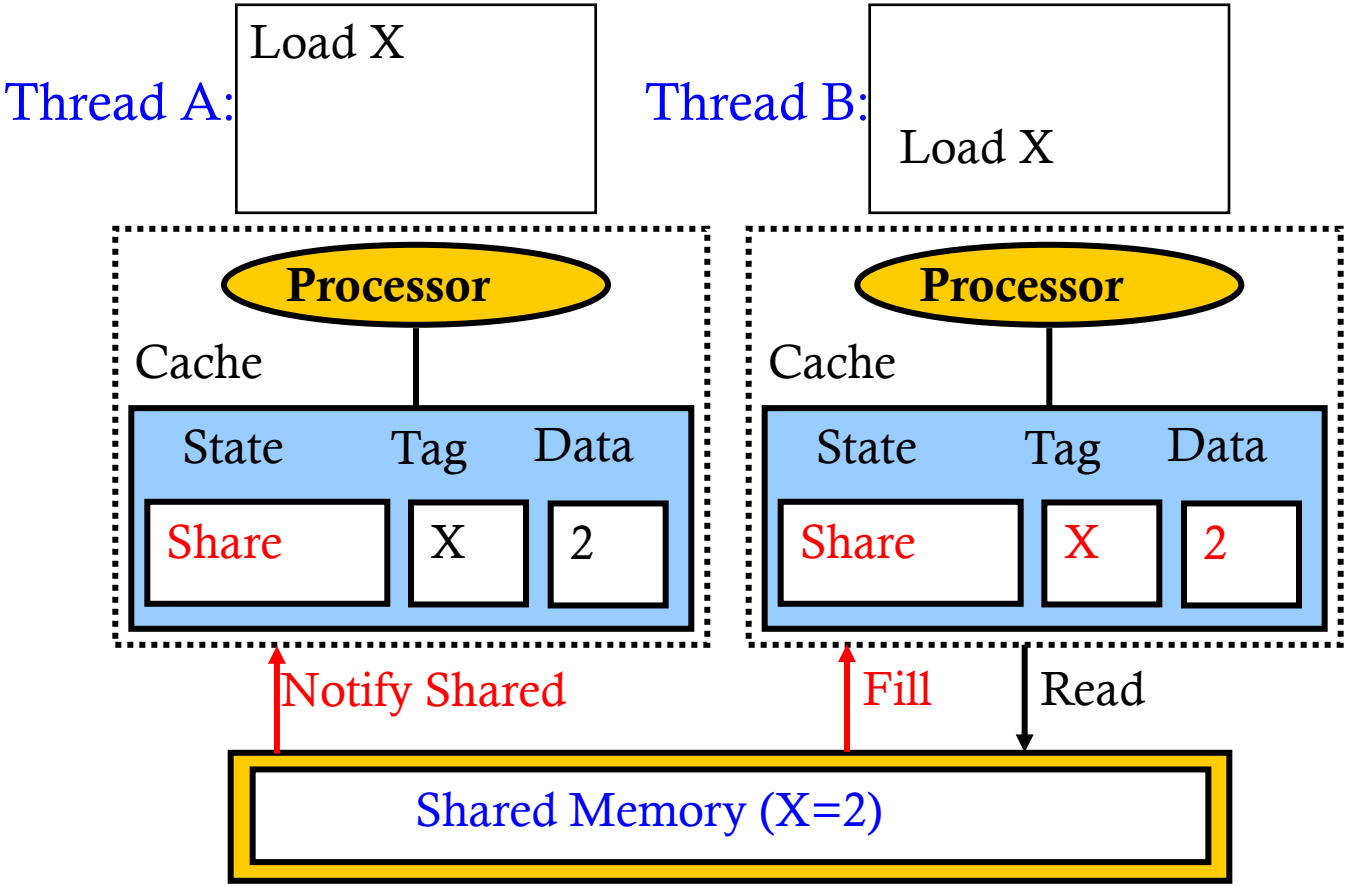
Example 1: MESI Coherence



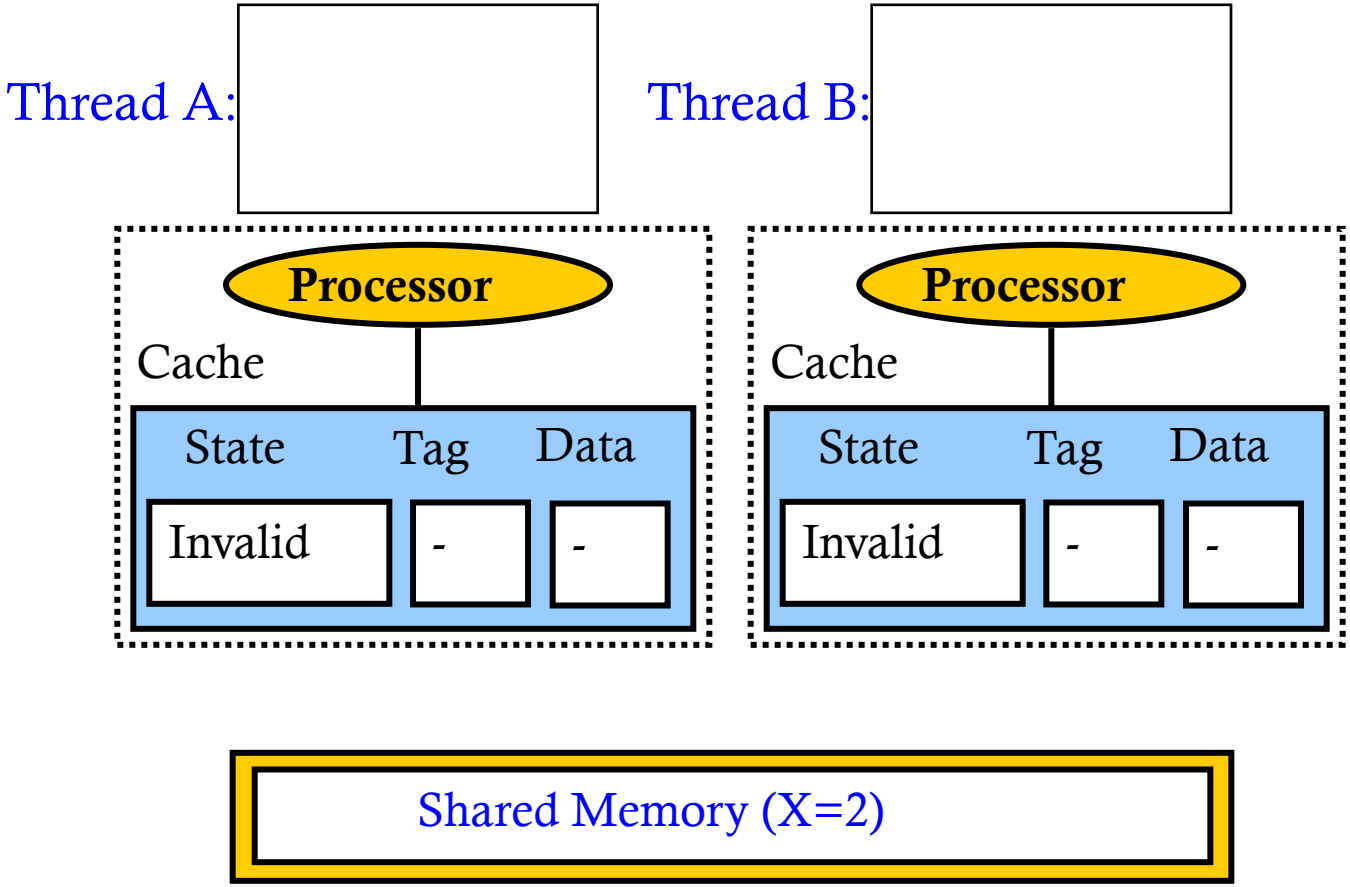
Example 1: MESI Coherence



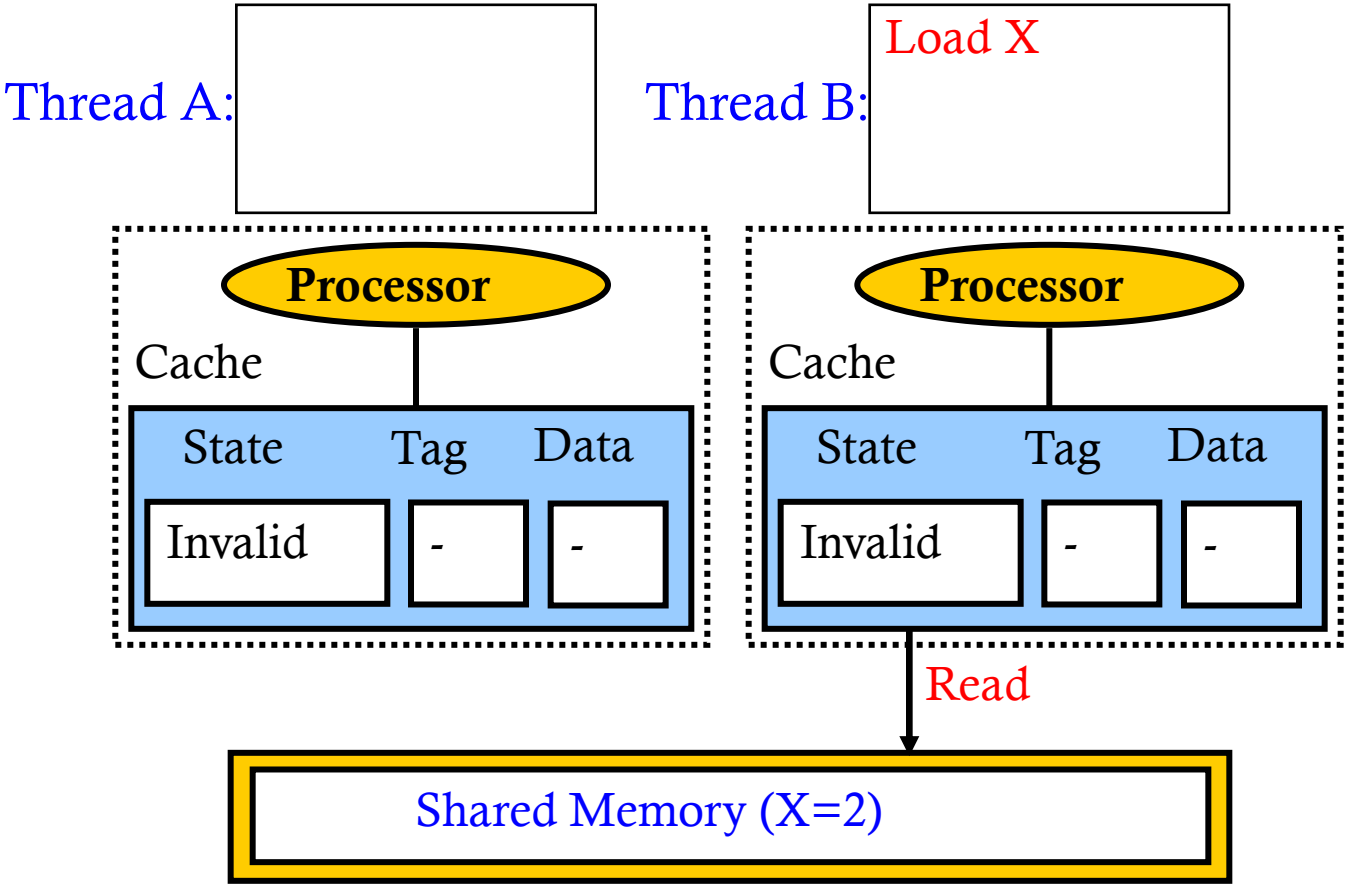
Example 1: MESI Coherence



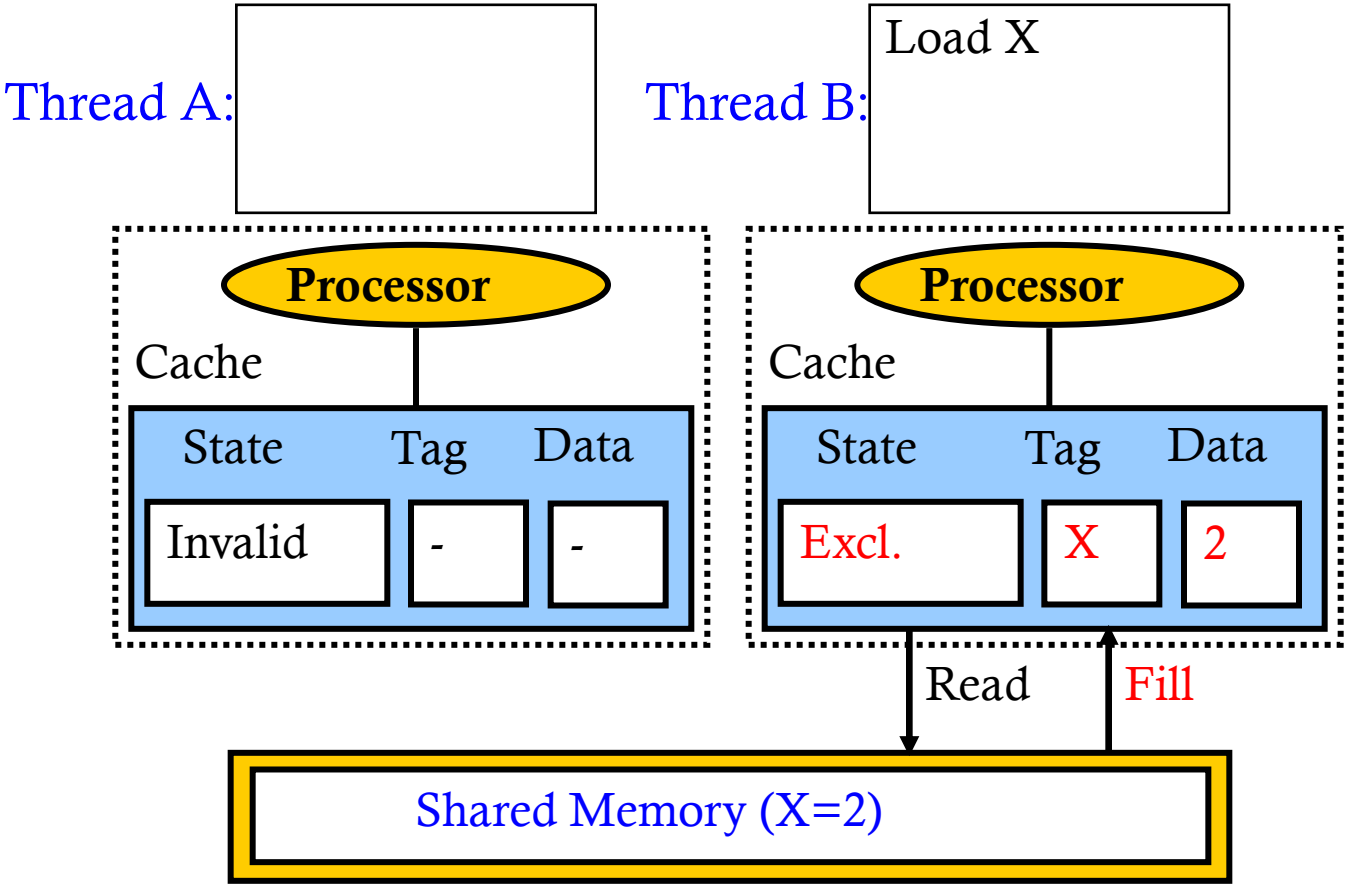
Example 2: MESI Coherence



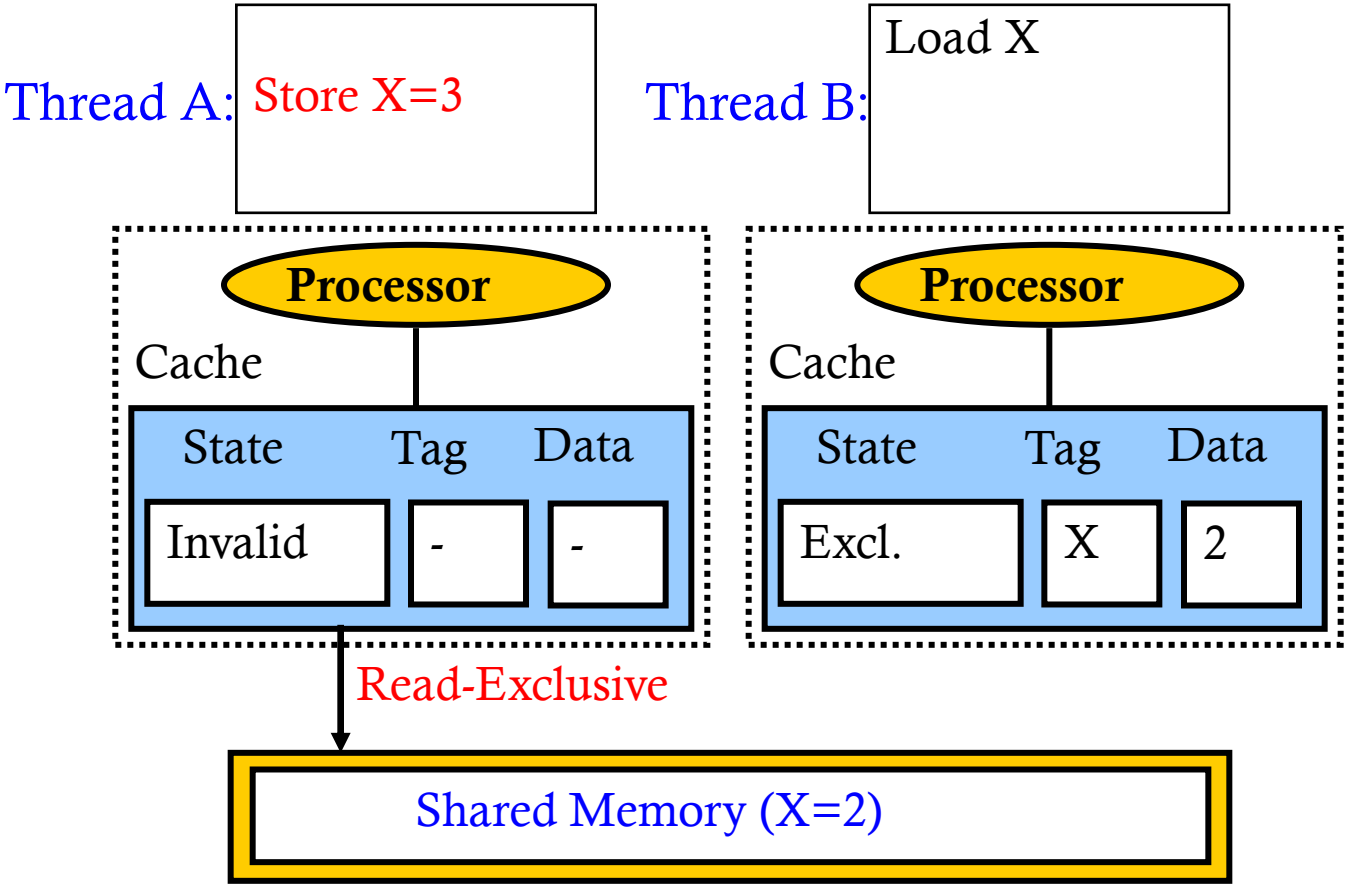
Example 2: MESI Coherence



Example 2: MESI Coherence

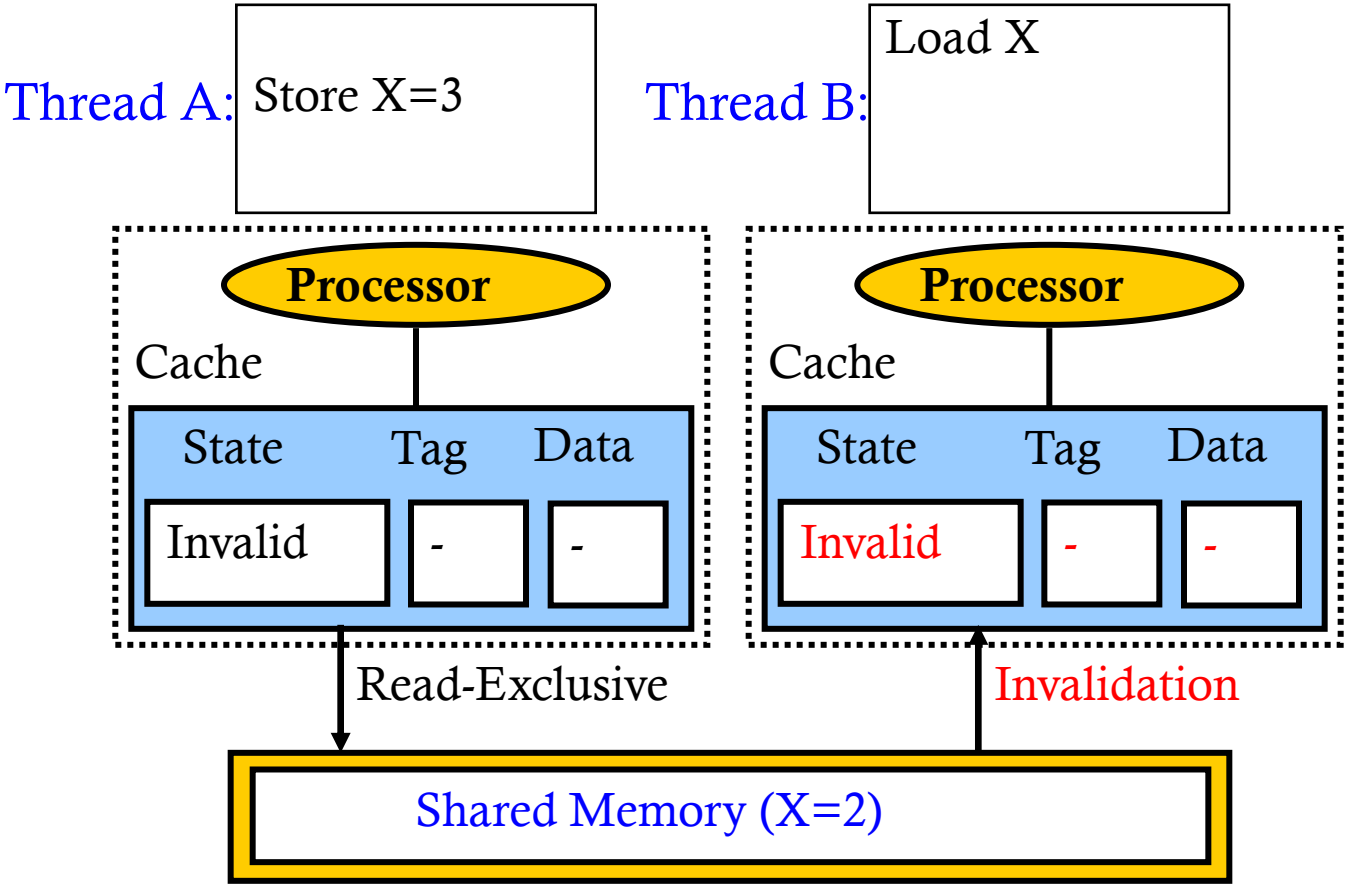


Example 2: MESI Coherence



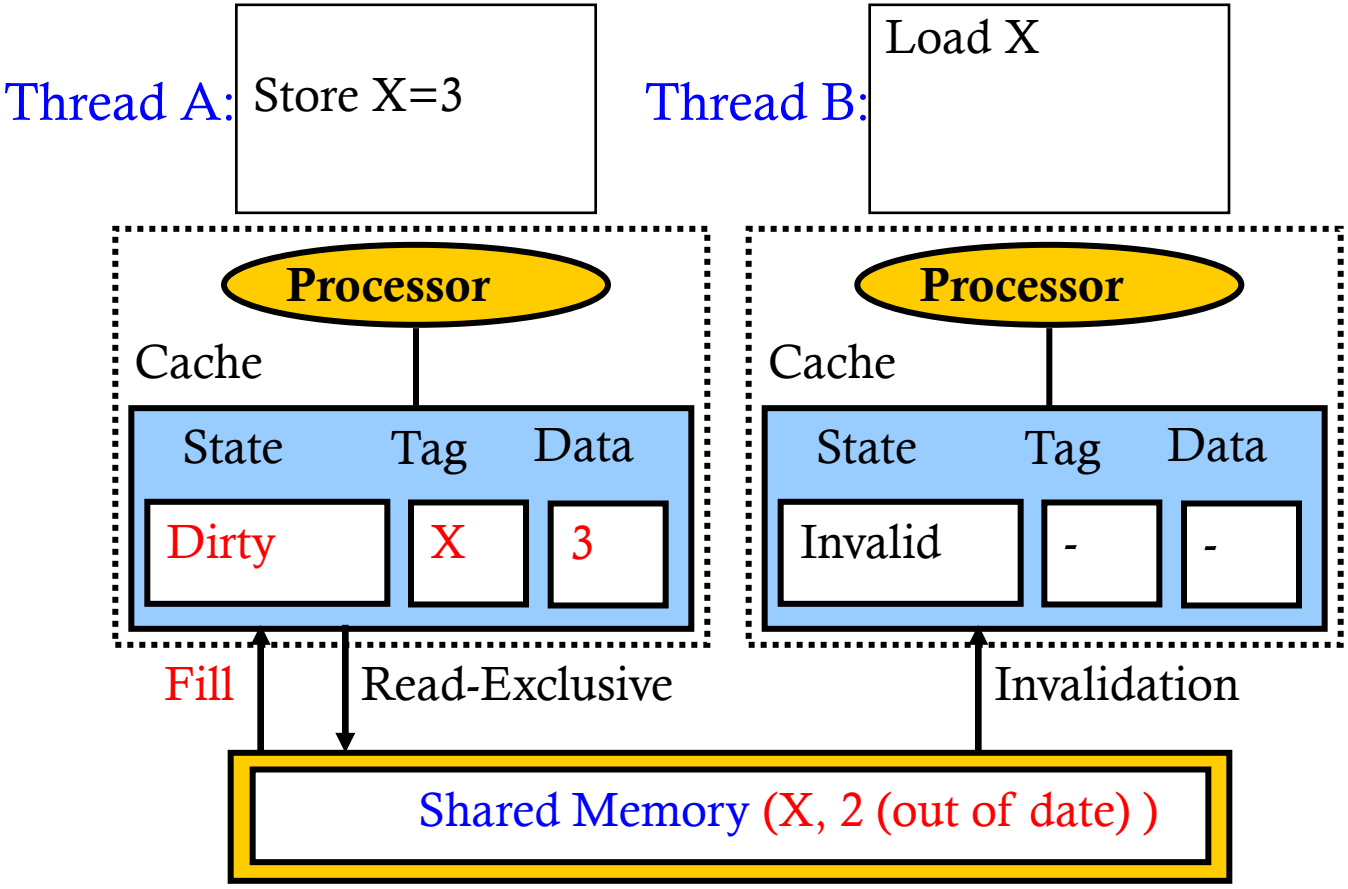
read-exclusive invalidates all other copies

Example 2: MESI Coherence



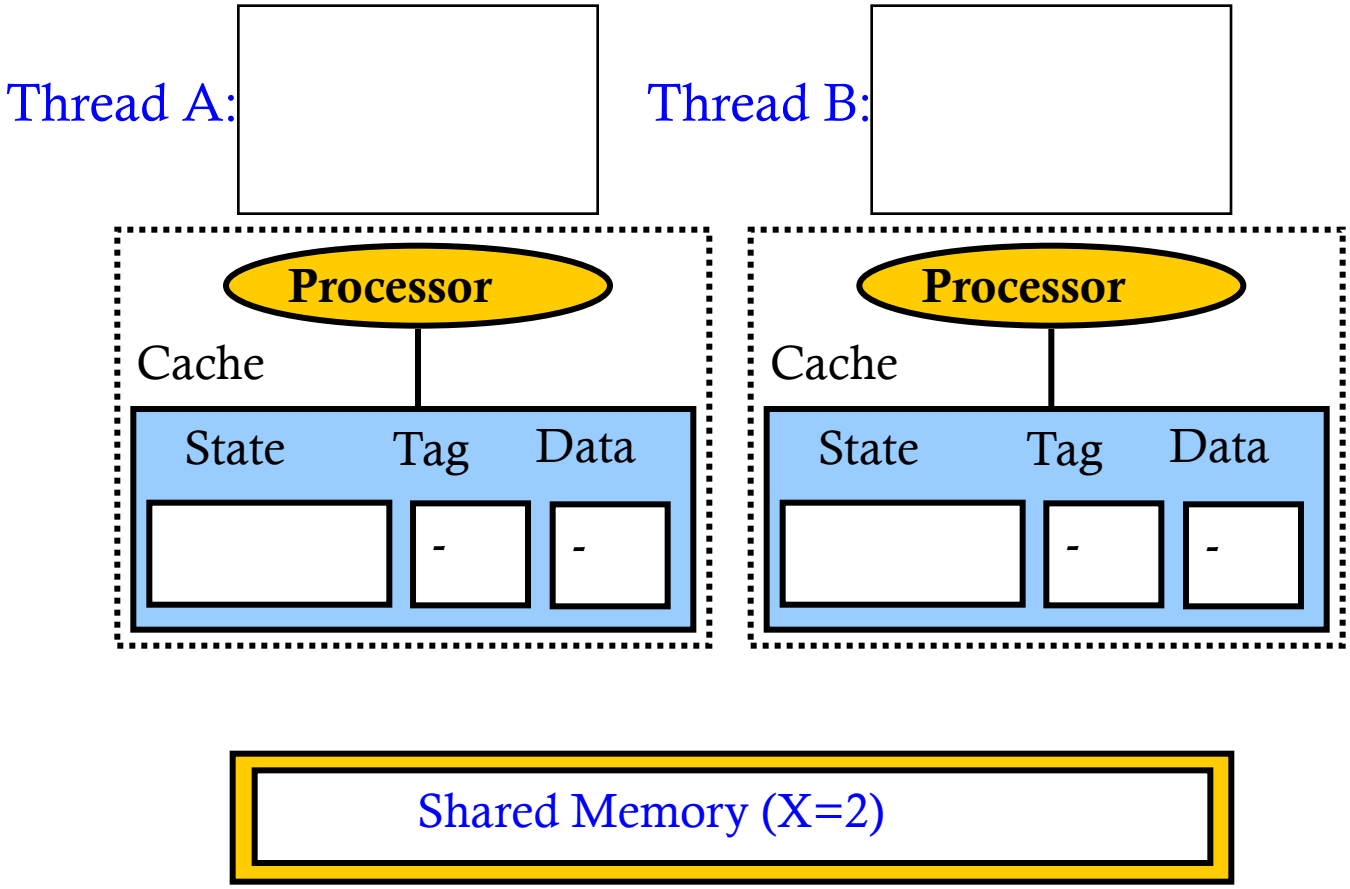
read-exclusive invalidates all other copies

Example 2: MESI Coherence

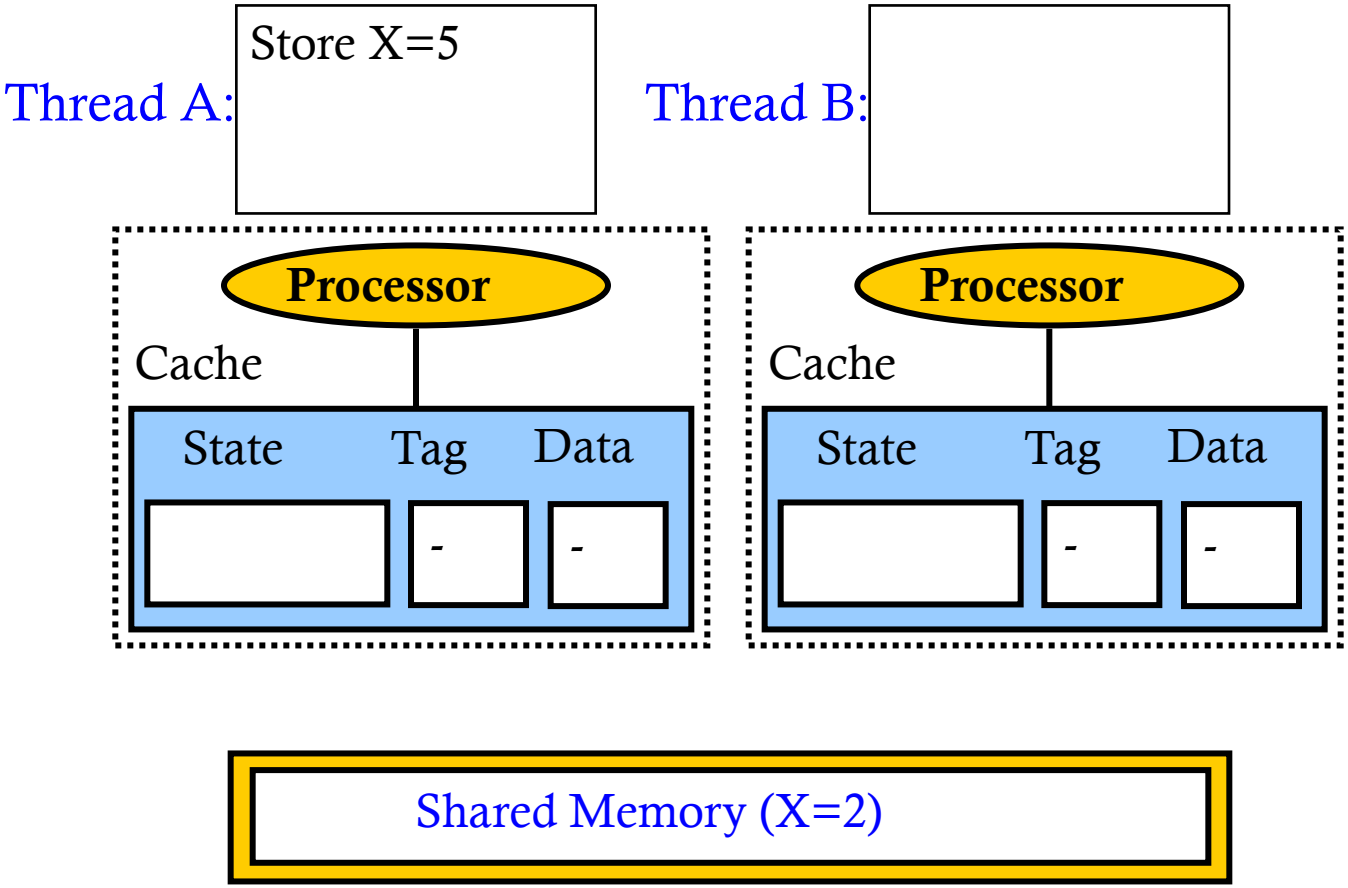


the state 'dirty' implies exclusiveness

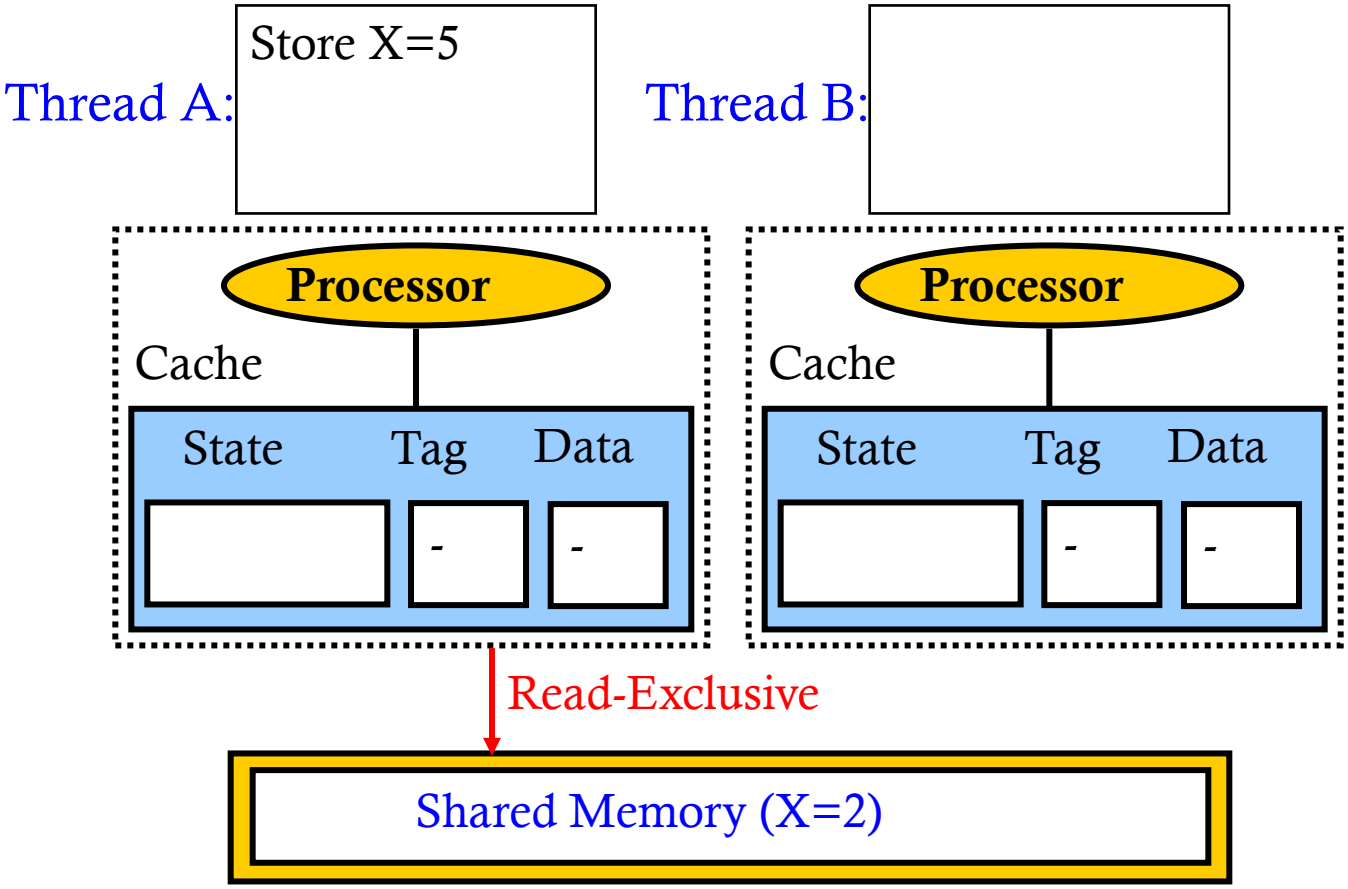
Example 3: MESI Coherence



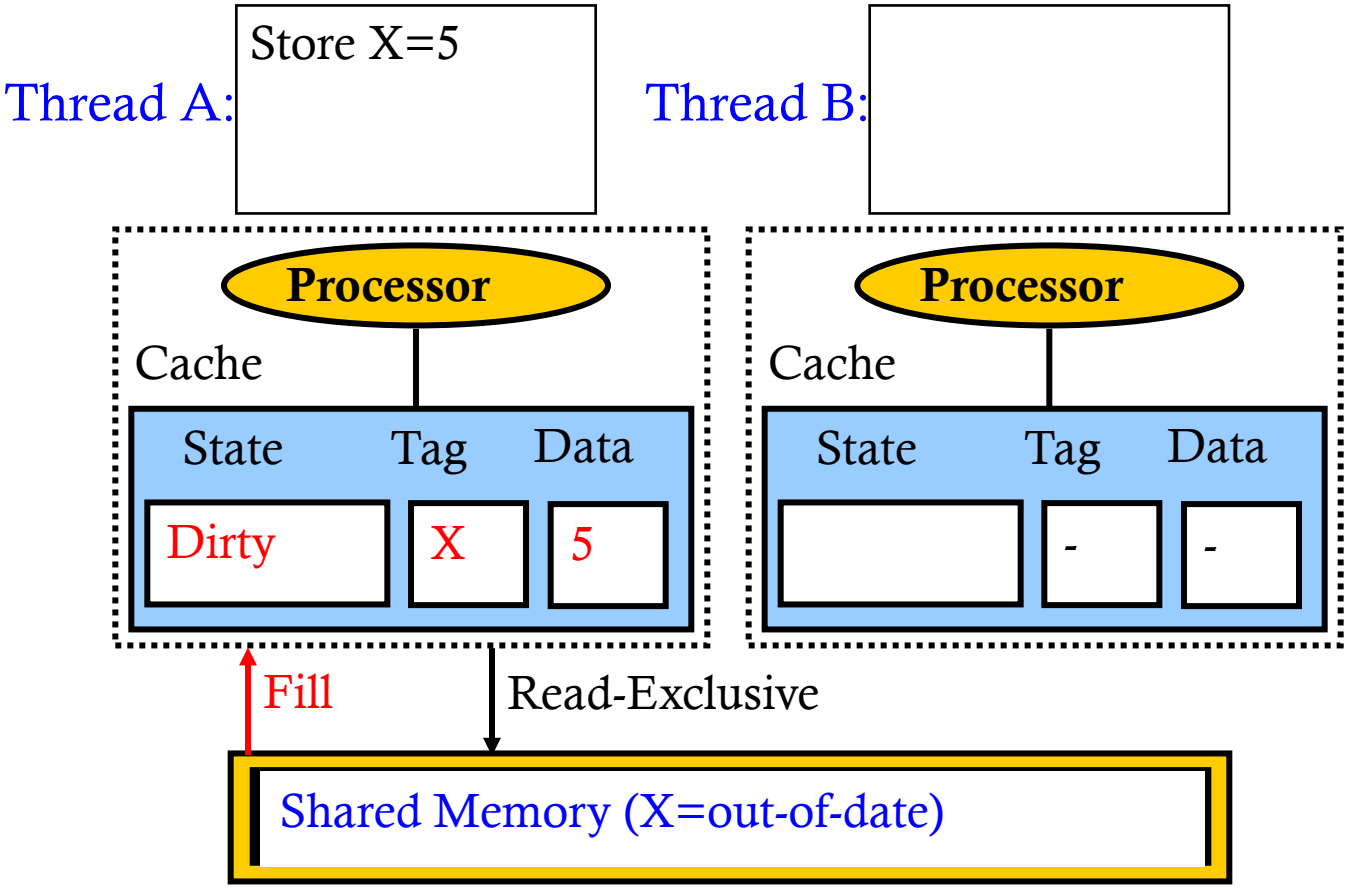
Example 3: MESI Coherence



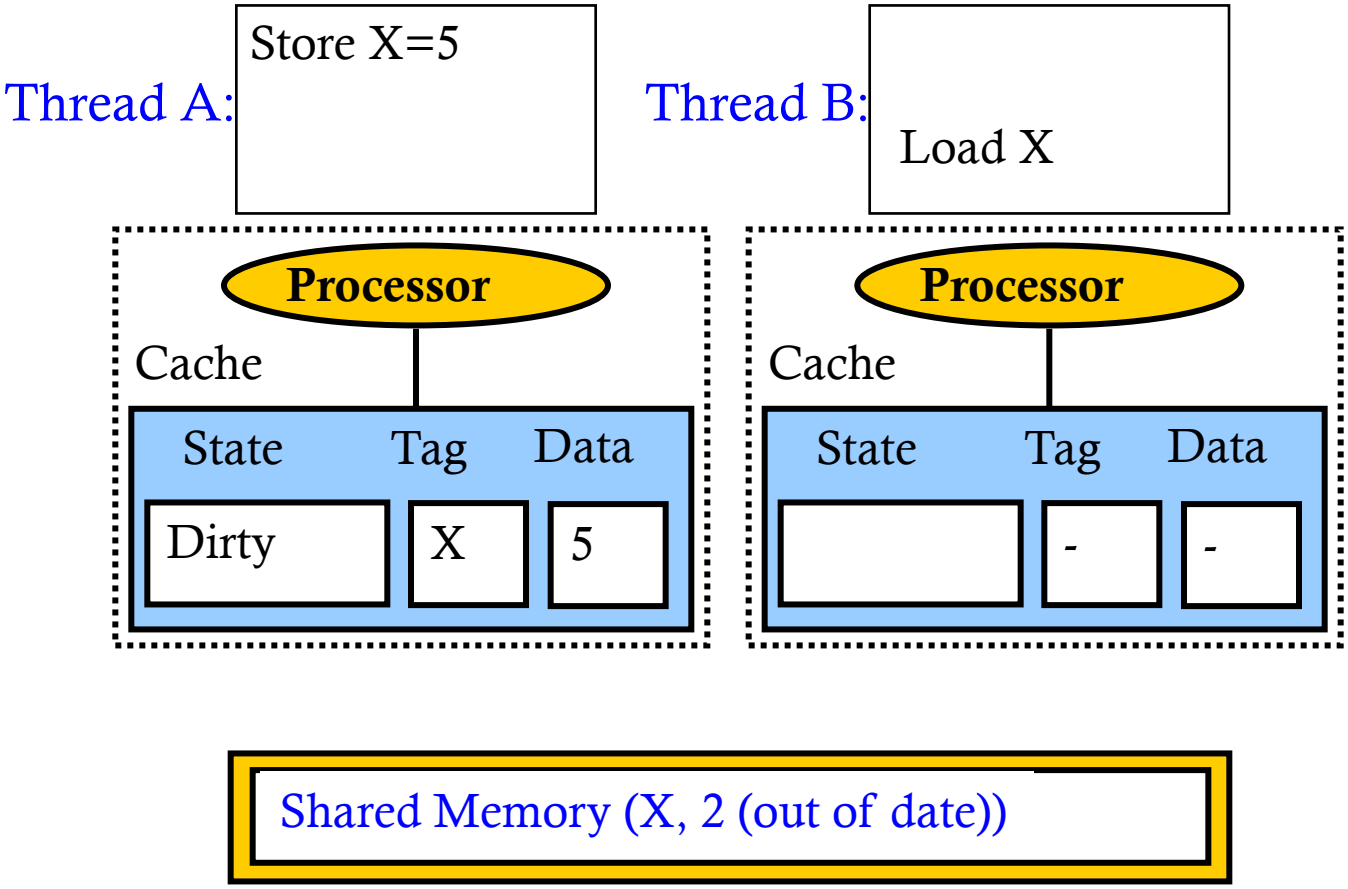
Example 3: MESI Coherence



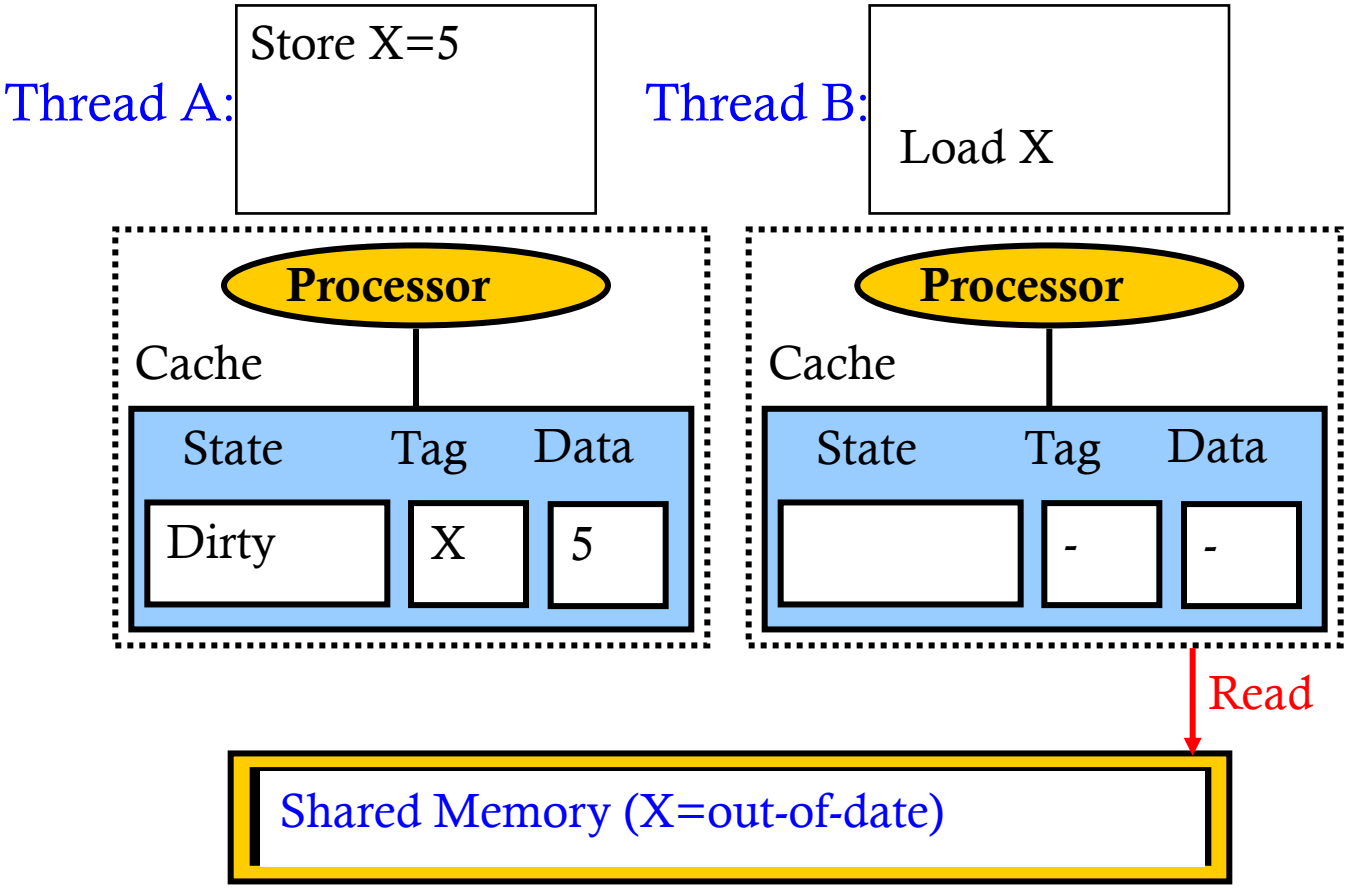
Example 3: MESI Coherence



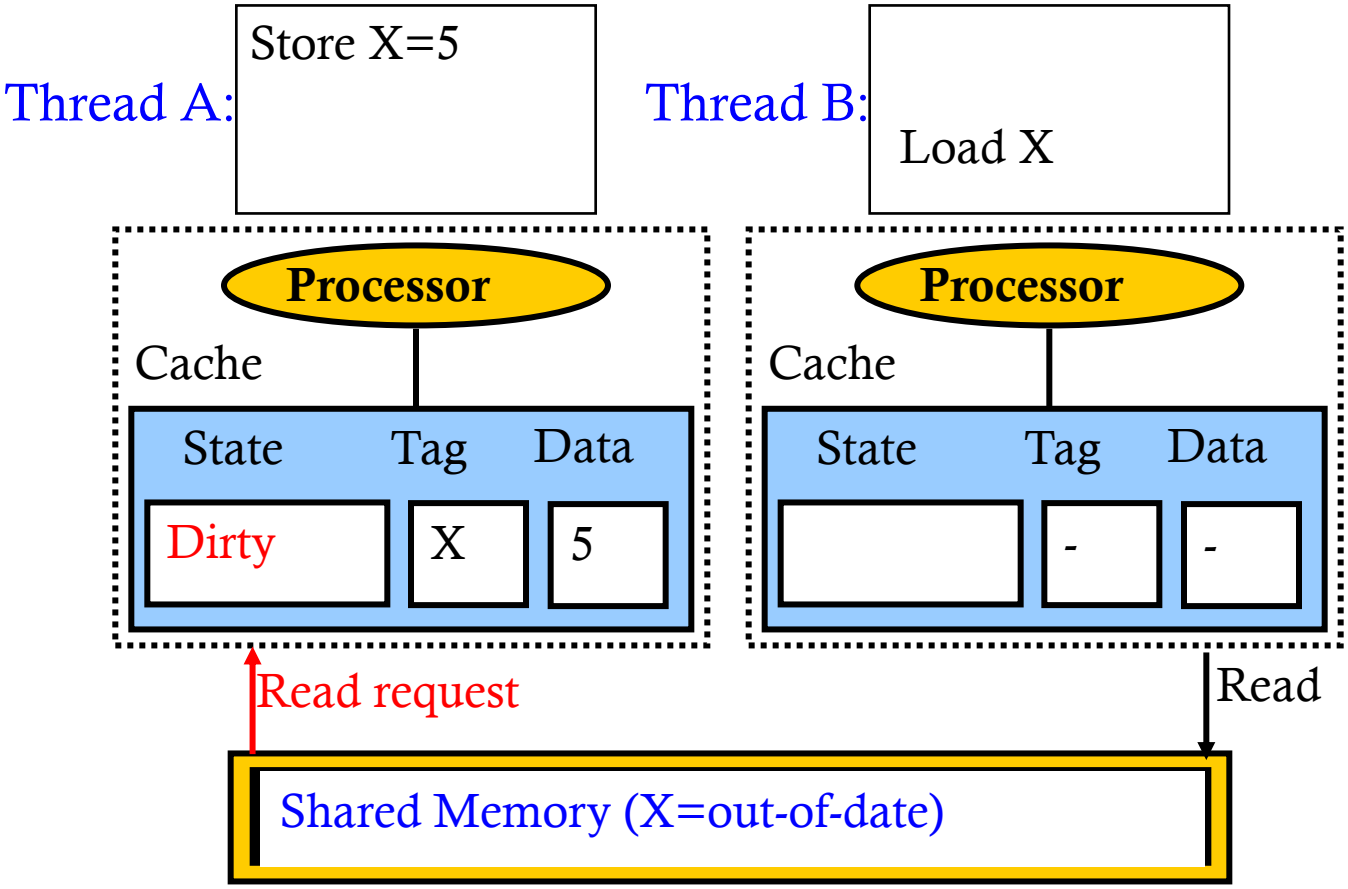
Example 3: MESI Coherence



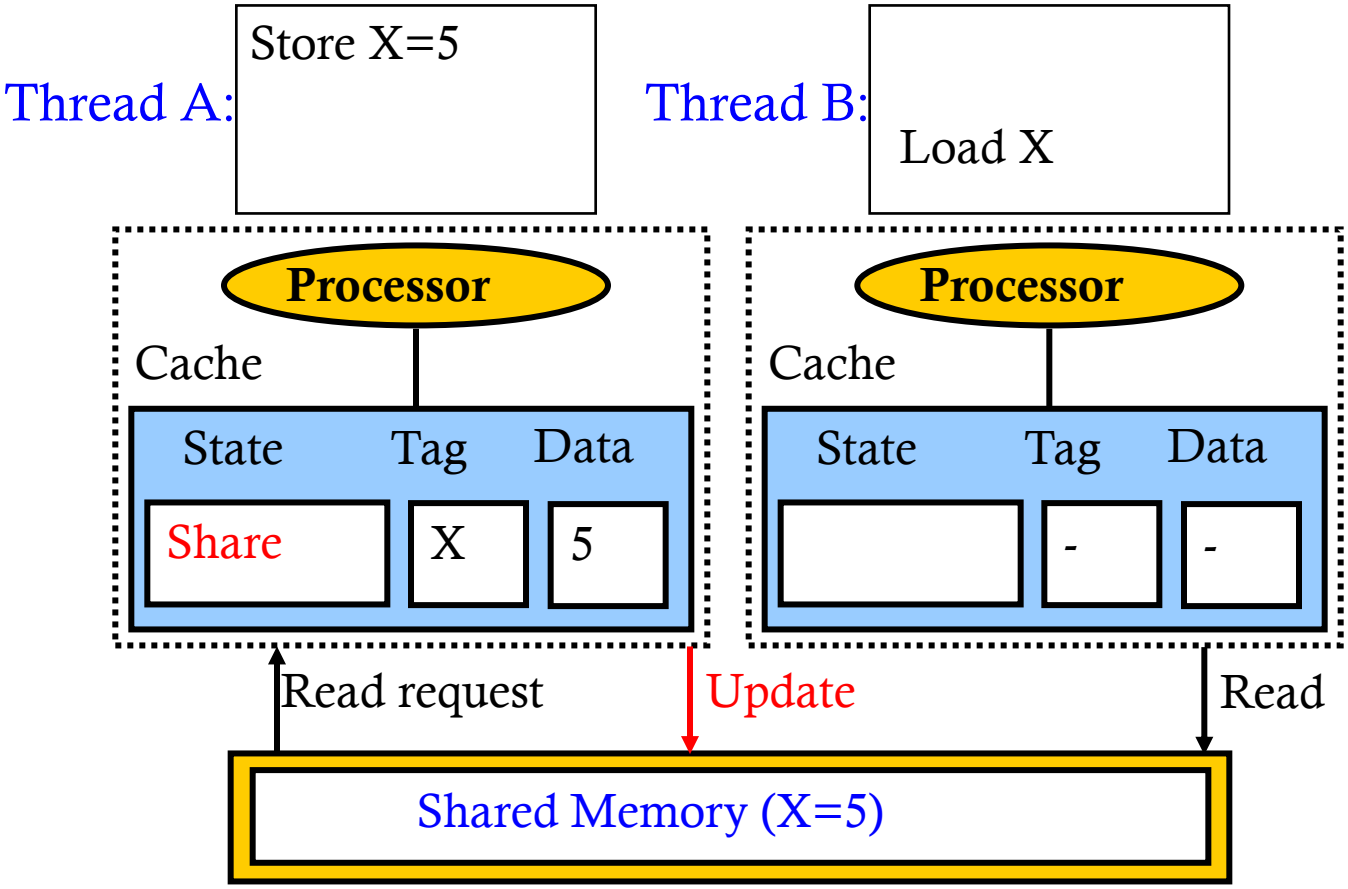
Example 3: MESI Coherence



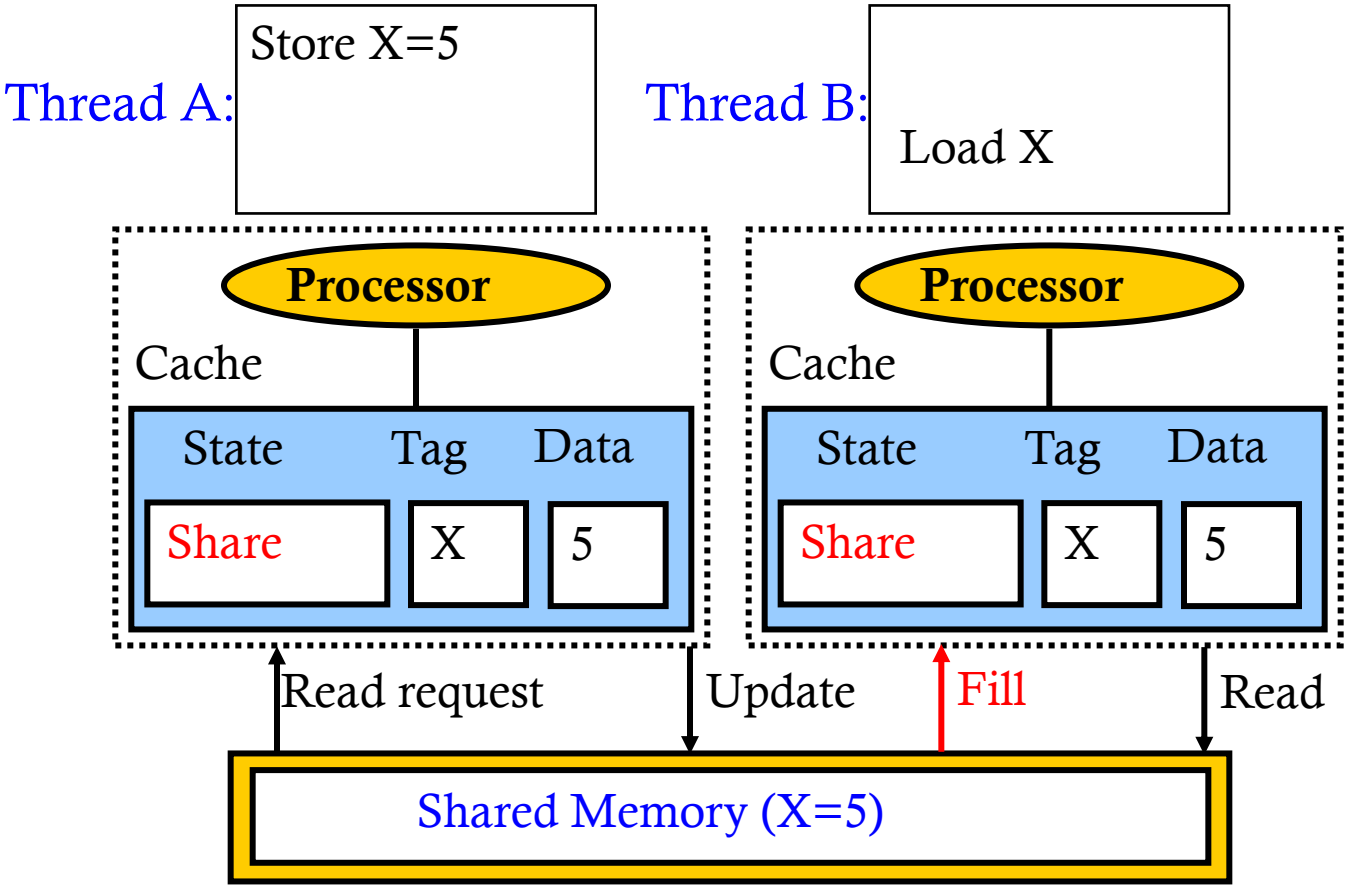
Example 3: MESI Coherence



Example 3: MESI Coherence



Example 3: MESI Coherence



MESI Permitted States, Transitions

permitted states on any pair of processor caches

	M	E	S	I
M	✗	✗	✗	✓
E	✗	✗	✗	✓
S	✗	✗	✓	✓
I	✓	✓	✓	✓

Local Event	Initial State	Local	Message	Remote
Read miss	I	I → (S, E)	READ	(S, E) → S M → S + WB
Read hit	S, E, M			
Write miss	I	I → M	READEX	(S,E) → I M → I + WB
Write hit	S E, M	S → M E → M	INVALIDATE	S → I

Performance of Memory Operations

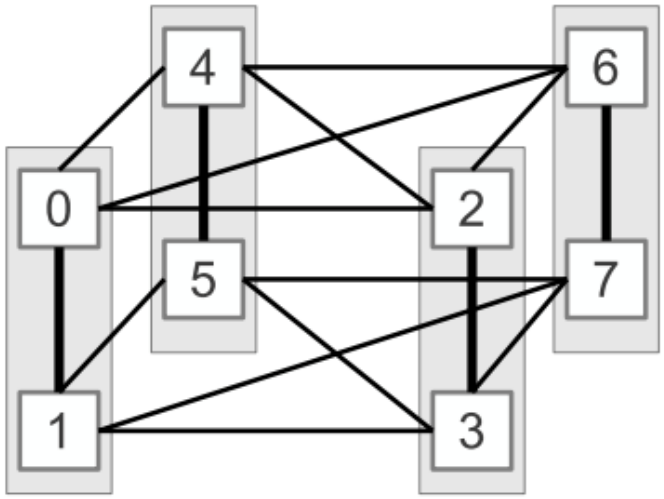
Material from: Everything you always wanted to know about synchronization but were afraid to ask, SOSP 2013.

Local Caches and Memory Latencies

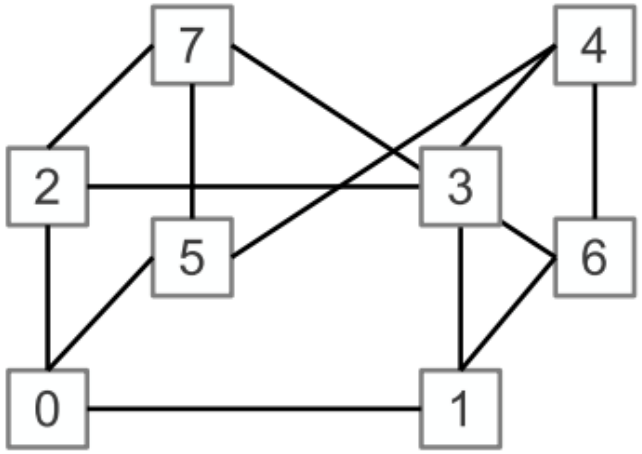
- Cost of accessing memory
  - Best case
    - Data is cached locally: L1 < 10 cycles (remember this)
  - Worst case
    - Data is accessed from DRAM: 136 – 355 cycles (remember this)

	Opteron	Xeon
L1	3	5
L2	15	11
LLC	40	44
RAM	136	355

Interconnect Between Sockets



(a) AMD Opteron



(b) Intel Xeon

Cross-sockets communication can be 2-hops

Latency of Remote Access: Read (cycles)

System	Opteron			Xeon		
State \ Hops	same die	one hop	two hops	same die	one hop	two hops
Modified	81	172	252	109	289	400
Exclusive	83	175	253	92	273	383
Shared	83	176	254	44	223	334

- Local cache line state is invalid
- State is the MESI state of a cache line in a remote cache
- Cross-socket communication is expensive!
  - Xeon: cross-socket latency is 4-7.5 larger than within socket
  - Opteron: cross-socket latency even larger than RAM

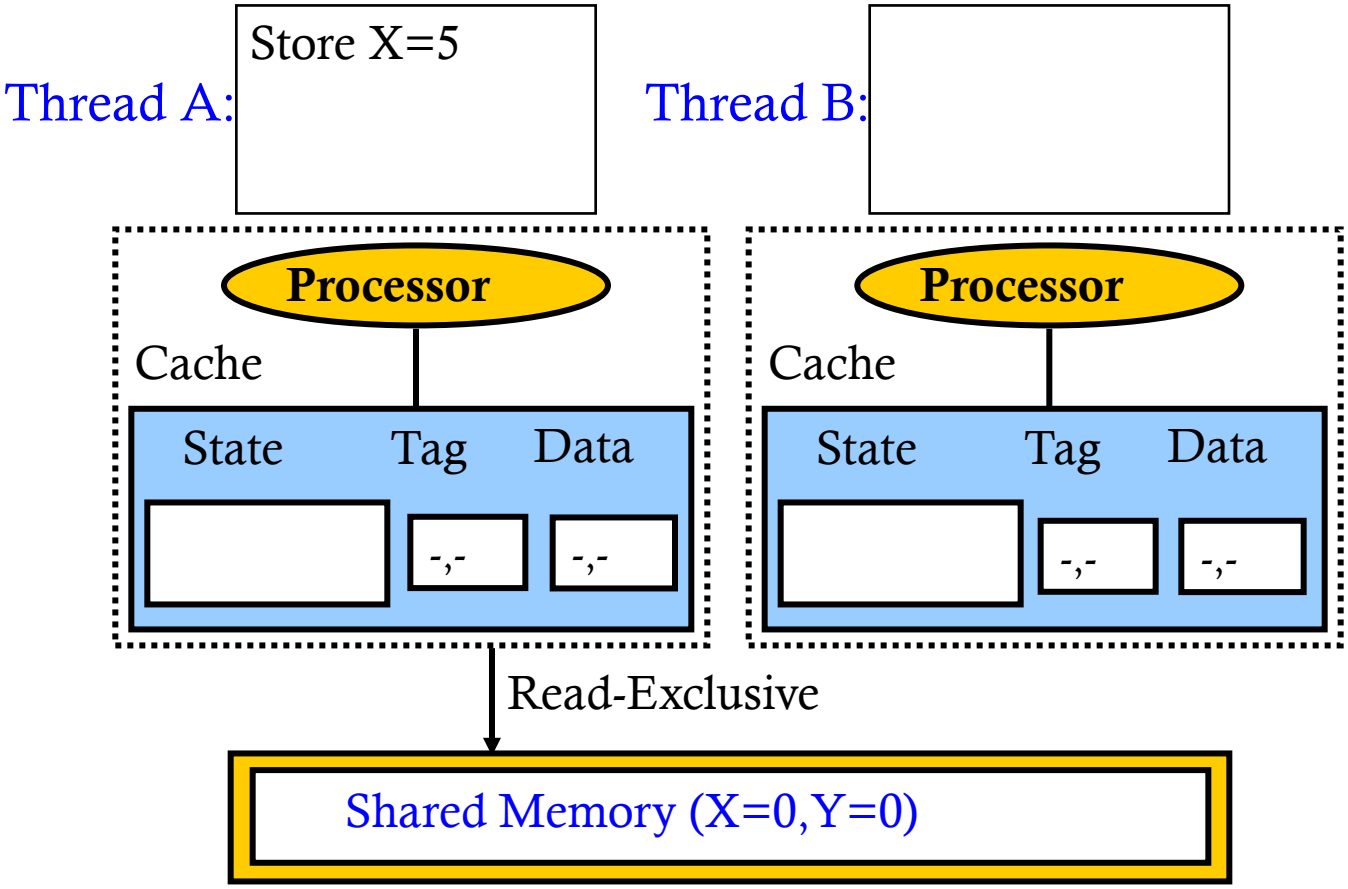
Latency of Remote Access: Write (cycles)

System	Opteron			Xeon		
State \ Hops	same die	one hop	two hops	same die	one hop	two hops
Modified	83	191	273	115	320	431
Exclusive	83	191	271	115	315	425
Shared	246	286	296	116	318	428

- Local cache line state is invalid
- State is the MESI state of a cache line in a remote cache
- Cross-socket communication is expensive!
  - Comparable or more expensive than DRAM accesses

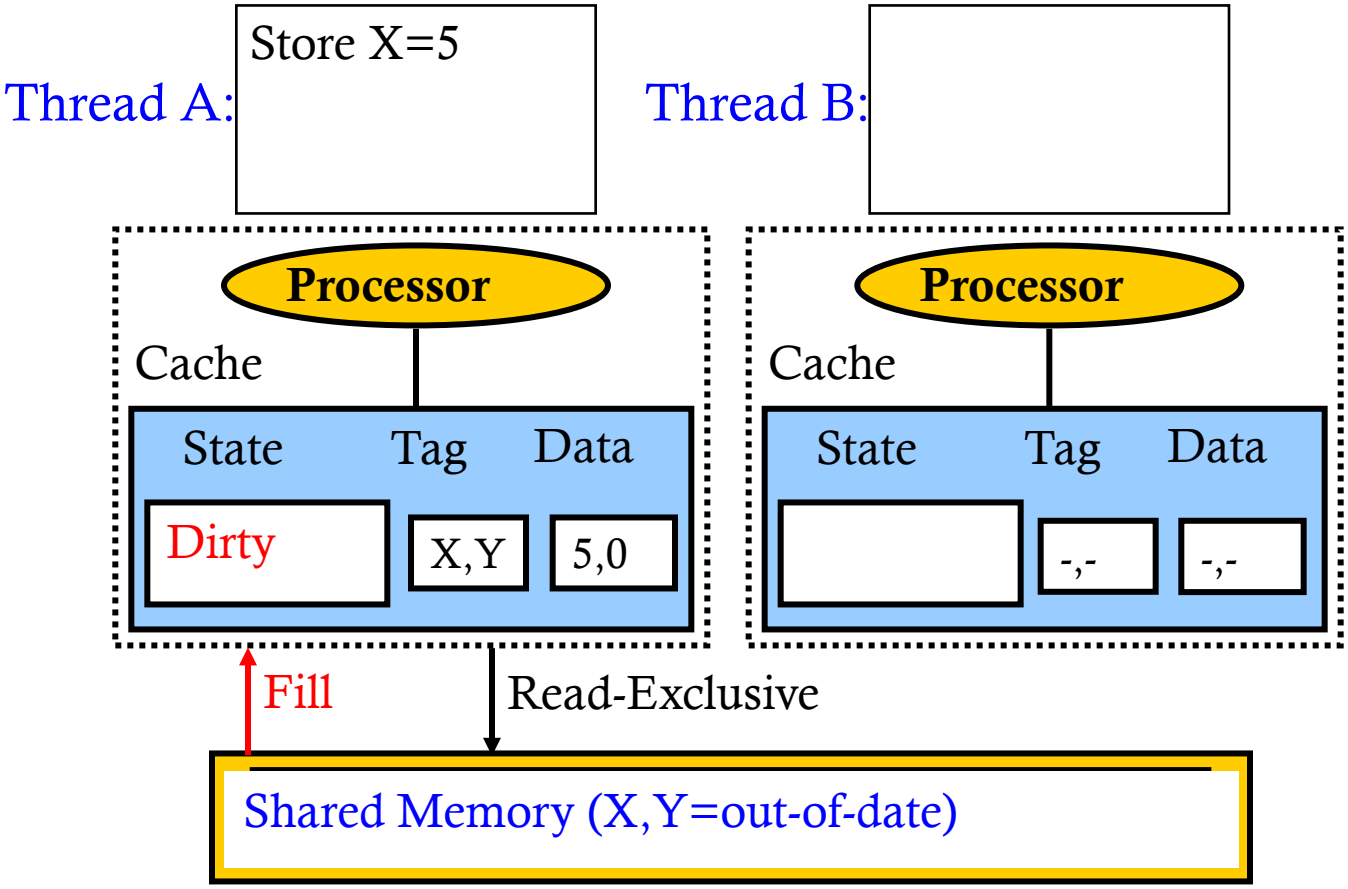
Implications for Software Design

False Sharing

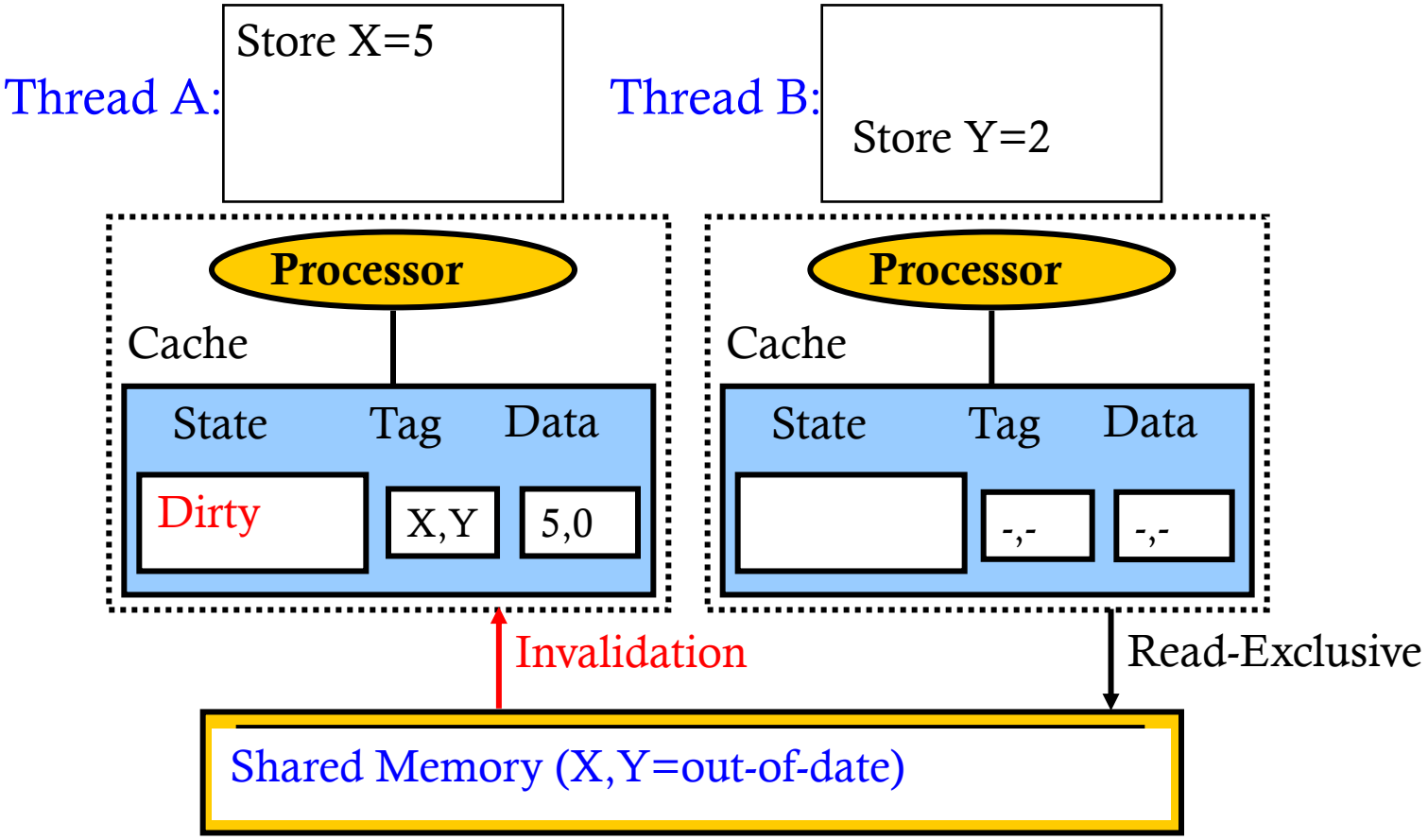


X and Y are on the same cache line

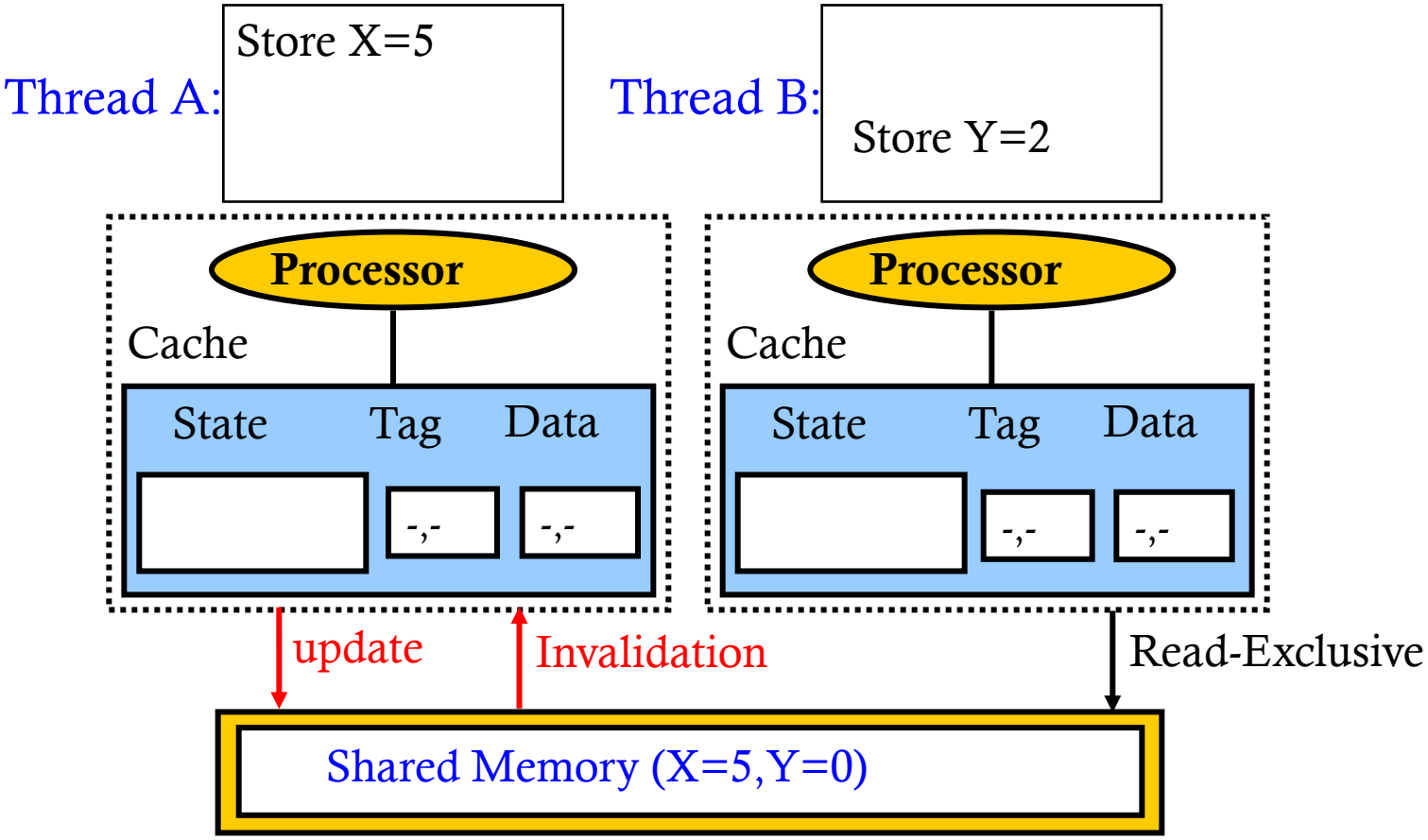
False Sharing



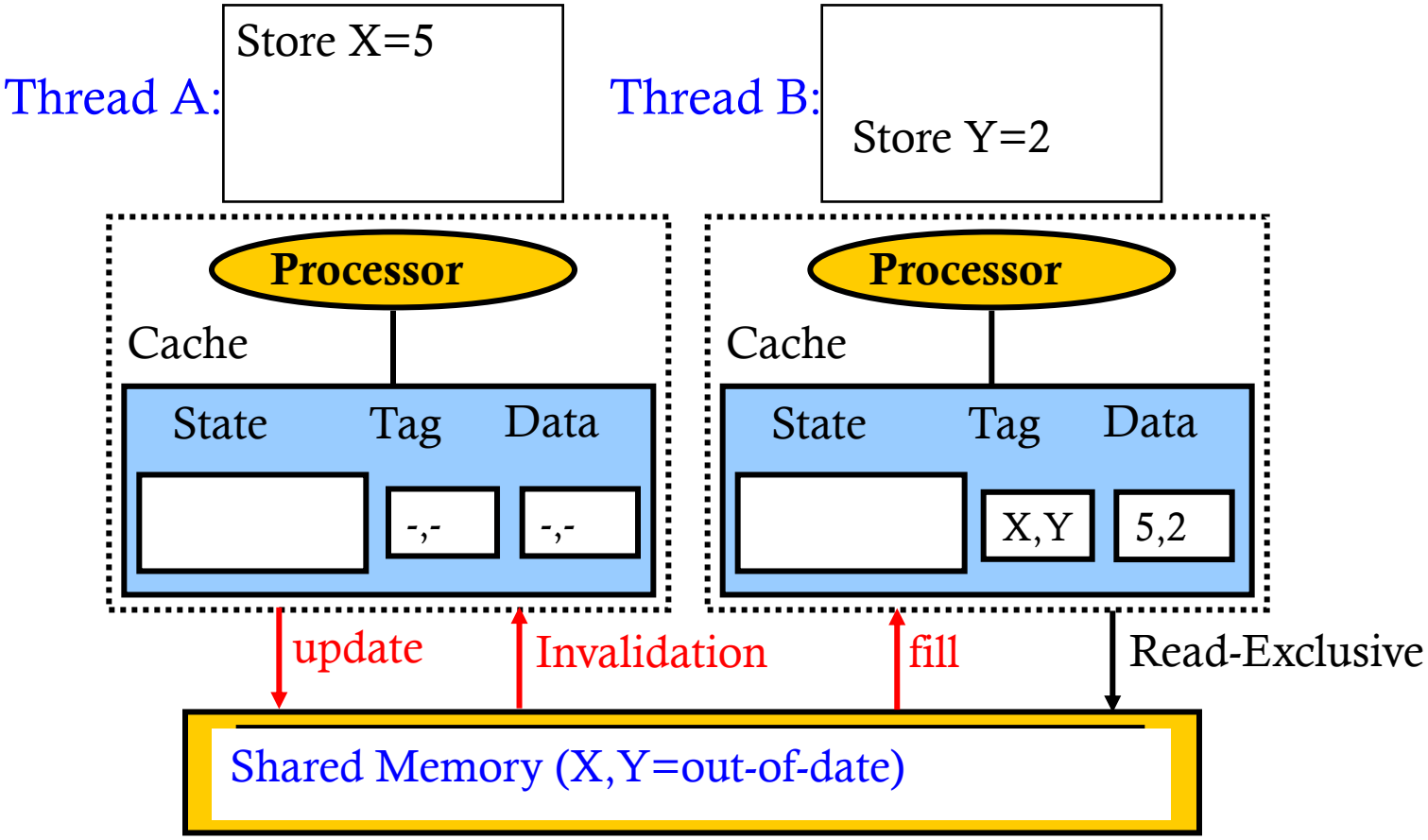
False Sharing



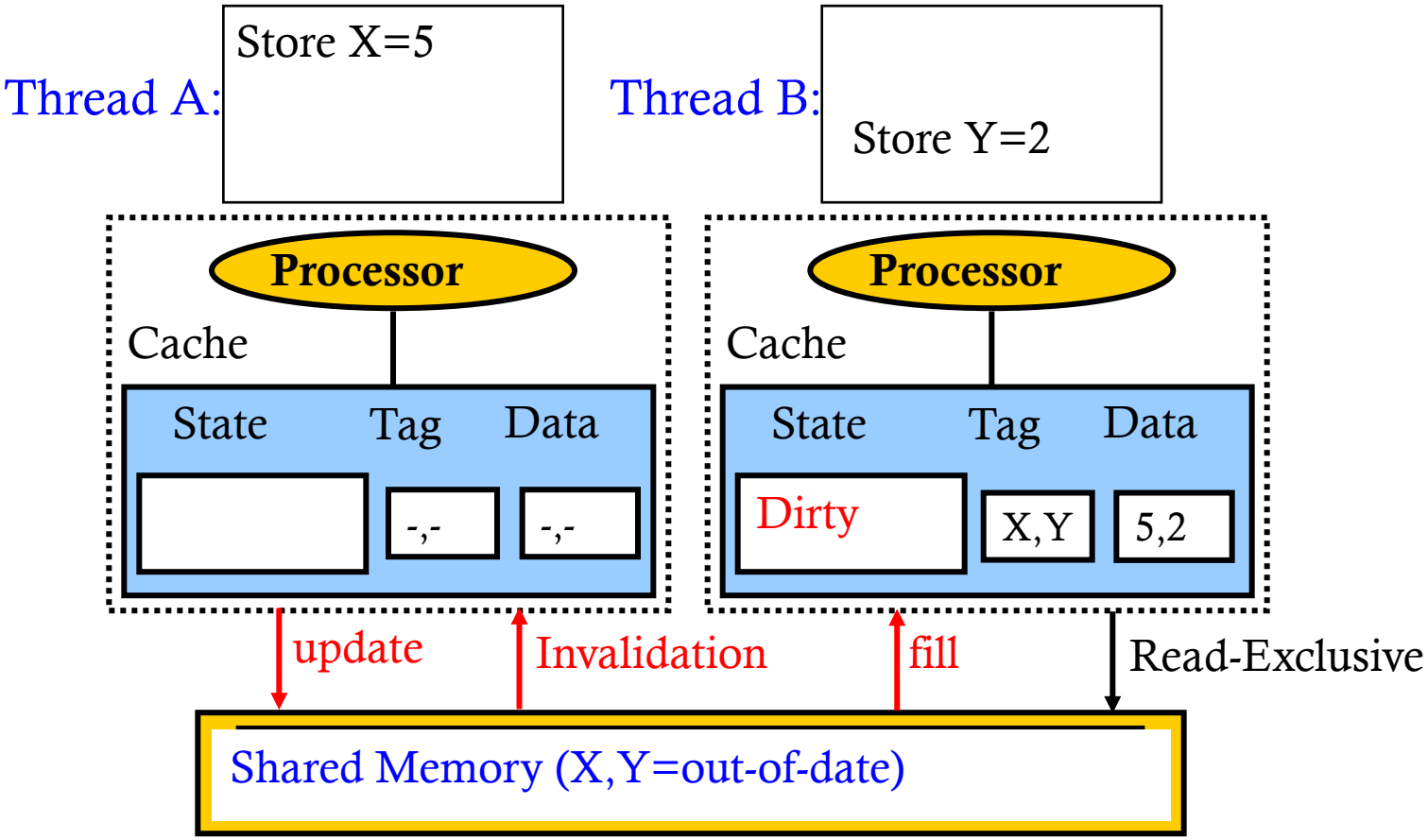
False Sharing



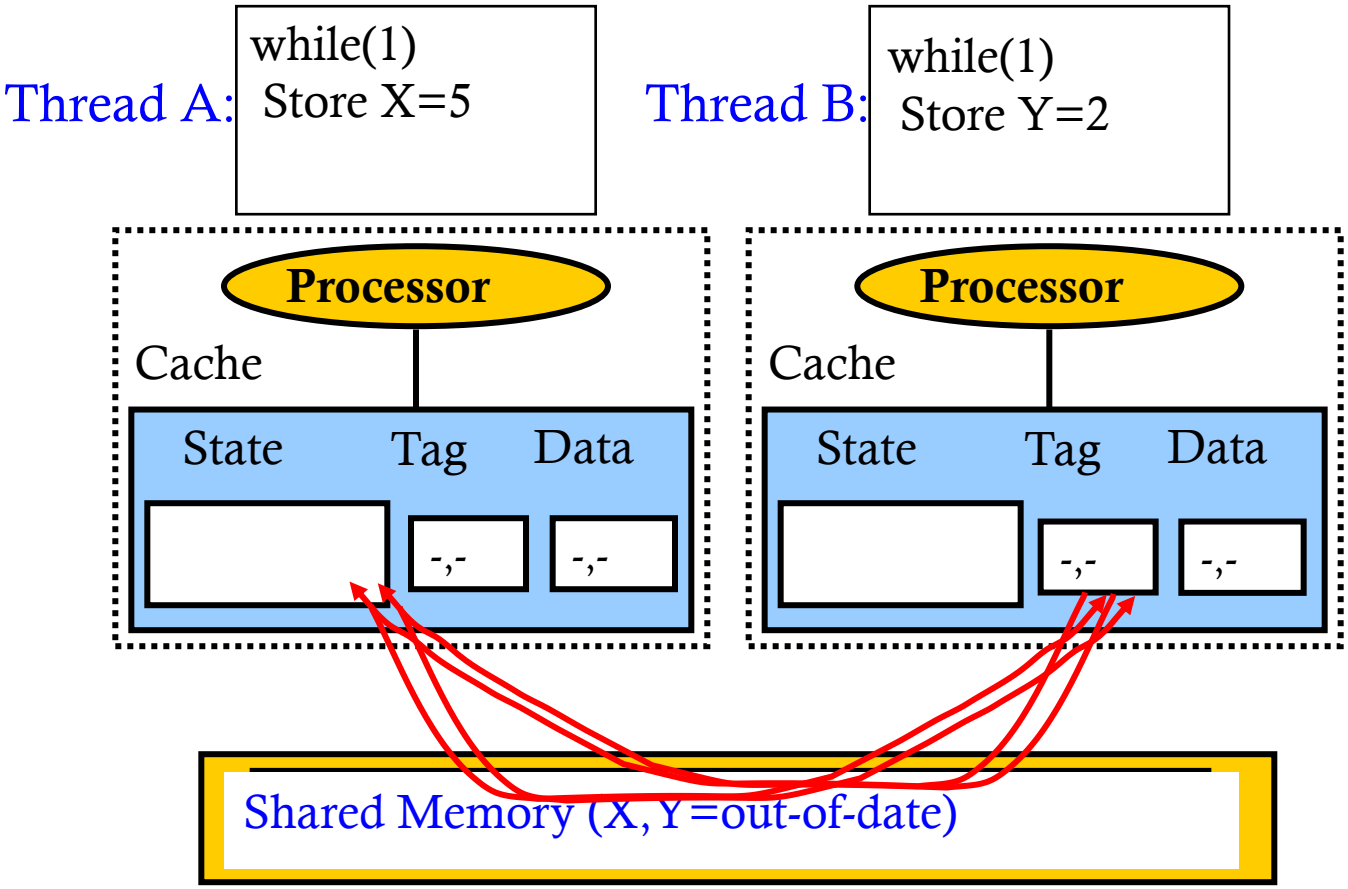
False Sharing



False Sharing



False Sharing



X,Y cache line will ping-pong back & forth

Implications for Programmers

- Cache coherence is expensive (more than you thought)
  - Avoid unnecessary sharing (e.g., false sharing)
- Crossing processors/sockets is a killer
  - Can be slower than running the same program on single core!
  - Pthreads provides CPU affinity mask
    - Pin cooperative threads on cores within the same die
- Later, we will see other implications of modern architectures on software design

False Sharing Summary

- False sharing
  - Threads on different cores access unrelated objects
  - Objects are located in same cache block
  - Block will ping-pong between caches on different cores
- Avoid false sharing by careful data arrangement
  - Ensure that unrelated elements are mapped to separate blocks
    - E.g., insert padding (unused data) between shared items
  - Use allocator that partitions allocations for different threads
    - Next, we discuss a memory allocator designed for parallel applications

Case Study: jemalloc

- Allocator designed to scale on multiprocessors and to minimize fragmentation
- Original design: Jason Evans, "A Scalable Concurrent malloc(3) Implementation for FreeBSD", BSDCan 2006
  - Please read for details, although the description is not great ☹
- Various versions used in several BSD releases, in Firefox
- Facebook has made several optimizations ☺
  - Please read details about the design philosophy at: "Scalable memory allocation using jemalloc", https://engineering.fb.com/2011/01/03/core-data/scalable-memory-allocation-using-jemalloc/

Motivation

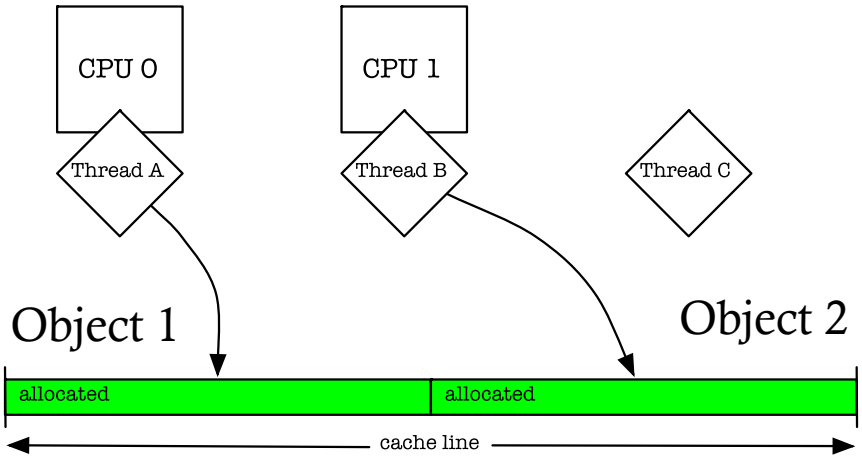
- Traditional allocators focus on optimizing for
  - Space utilization
  - Performance (on single core)
- However, these allocators have two issues that lead to poor multi-core scaling
  - Data packing (causes false sharing)
  - Locking scheme (leads to scalability bottlenecks)

Data Packing

- Traditional malloc packs data as efficiently as possible
  - Improves caching, which is critical for program performance
  - Helps with growing disparity between CPU and DRAM speeds (memory wall)
- However, packing data structures can lead to poor multi-threaded performance

Packing Data: False Cache-Line Sharing

- Heap objects 1 and 2 accessed by Thread A and Thread B
  - The two objects are not shared by the threads
- Say, malloc allocates objects within same cache line



- Reads and writes on different cores for unrelated objects cause cache line invalidation, reduce performance dramatically

Avoiding Cache-Line Sharing

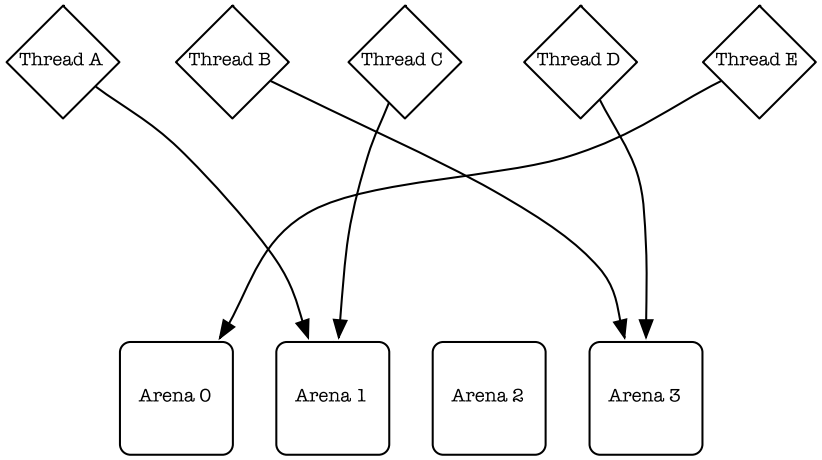
- malloc could add padding to the objects
  - However, padding causes internal fragmentation, making it cache-unfriendly
- Require user to pad objects that cause multi-core scaling bottleneck
  - Pros: Causes internal fragmentation for some objects only
  - Cons: Requires programmers to carefully align data structures that cause contention (i.e., limit scaling), requires detailed profiling to determine contention
- Neither scheme is attractive

Locking Scheme

- Traditional malloc used simple locking scheme
  - E.g., lock all data structures, serialize malloc and free calls
  - Sufficient with limited # of cores
  - Doesn't scale well today since allocator locks become a performance bottleneck

Avoiding Allocator Bottlenecks

- Use memory arenas
  - Split heap memory into arenas, or continuous blocks of memory
  - Each thread allocates memory from a single arena
  - Memory allocated from an arena is freed into same arena
- Memory arenas reduce cache line sharing and lock contention



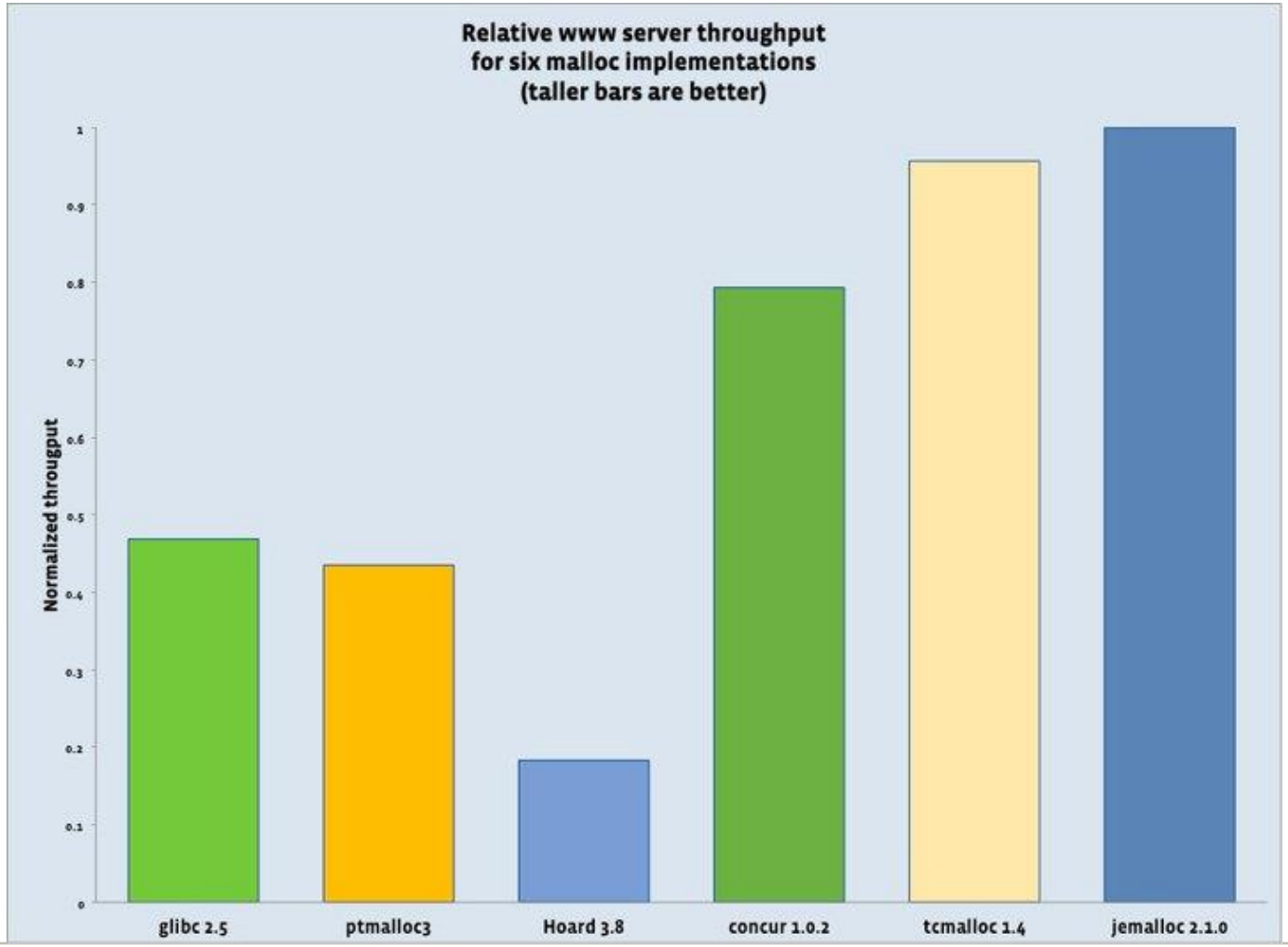
Memory Arenas for Threaded Applications

- Total arenas typically limited to 4-8 times the number of cores
- Options for mapping threads to arenas
  - Use simple round-robin
    - Need to account for thread exit, thread locking behavior, etc.
  - Use a hashing mechanism
    - Can lead to load imbalance due to poor hashing
  - Another optimization: During mapping, switch to another arena if `malloc` encounters an arena that is locked by another thread
    - Using `pthread_mutex_trylock()`)

Jemalloc: Scaling for Multi-Cores

- Each thread allocates from an arena
  - Reduces locking overheads, since few threads access each arena
  - However, arenas still require synchronization
    - Allocation requires locking a bin in the arena and/or the whole arena
- Add a per-thread allocator cache
  - Each thread maintains a cache (segregated by object size)
  - Most allocation requests hit the cache and require no locking
  - # of cached objects per size class is capped so that arena synchronization is reduced by ~10-100X
    - Higher caching causes increased fragmentation
    - Allocator reduces fragmentation by periodically returning unused cache objects to underlying arena

Jemalloc Performance



Jemalloc Summary

- Designed to scale on multiprocessors and to minimize fragmentation
- Multi-processor scaling
  - Minimize cache line sharing by using arenas
  - Use round-robin assignment of threads to arenas to reduce skew
  - Use per-thread allocator cache
- Minimize fragmentation
  - Carefully choose size classes (what are the tradeoffs?)
  - Prefer low addresses for allocation, similar to phkmalloc
  - Tight limits on metadata overhead
    - < 2%, ignoring fragmentation