

CS622: Advanced Computer Architecture

Thread Level Parallelism (TLP)

Classification of Computing Frameworks

- Single Instruction Single Data (SISD)

Covered

- Single Core with ILP techniques such as superscalar and speculative execution

- Multiple Instruction Multiple Data (MIMD)

Agenda

- Independent processing by different processors with correctness guarantees
 - Multi-core and multi-threaded processors

- Single Instruction Multiple Data (SIMD)

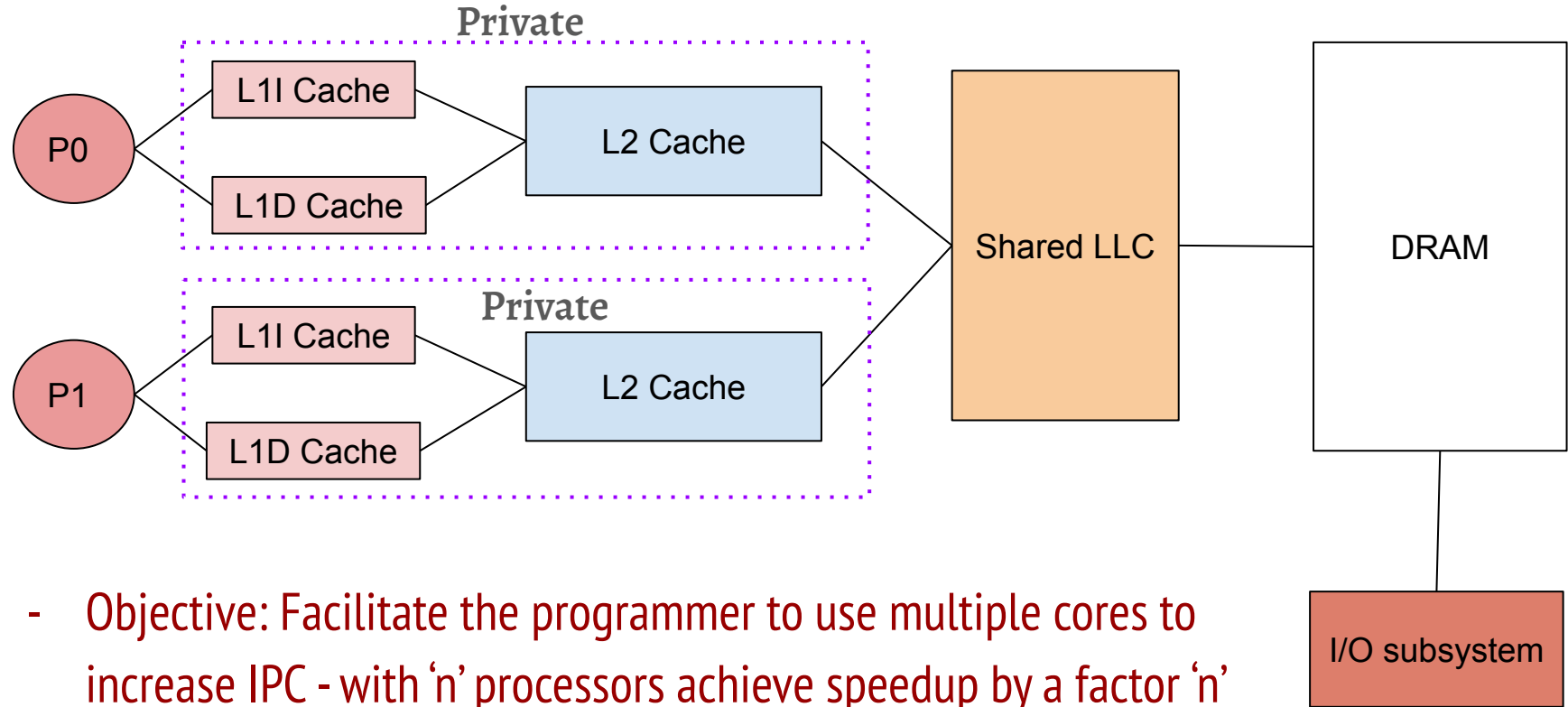
Will cover next (!GPU)

- Multiple SIMD processor execute the same instruction on multiple data to achieve data level parallelism

- Multiple Instruction Single Data (MISD)

- Not very common as concurrent operation on the same data ensuring correctness is difficult to achieve

Shared Memory Multi-processors



- Objective: Facilitate the programmer to use multiple cores to increase IPC - with 'n' processors achieve speedup by a factor 'n'
- What are the possible bottlenecks in achieving the speedup?

Bottlenecks in parallel performance

1

Hardware Bottlenecks

- Shared hardware (LLC, DRAM, Interconnects)
 - Memory wall problem
 - Resource scheduling and management

2

Software Bottlenecks

- Ordering across CPUs, atomic operations
- I/O operations and related bottlenecks

3

Cross-cutting Issues

- Cache/TLB coherence, memory consistency
- Supporting atomic ops and memory ordering

Bottlenecks in parallel performance

1

Hardware Bottlenecks

- **Shared hardware (LLC, DRAM, Interconnects)**
 - **Memory wall problem**
 - **Resource scheduling and management**

2

Software Bottlenecks

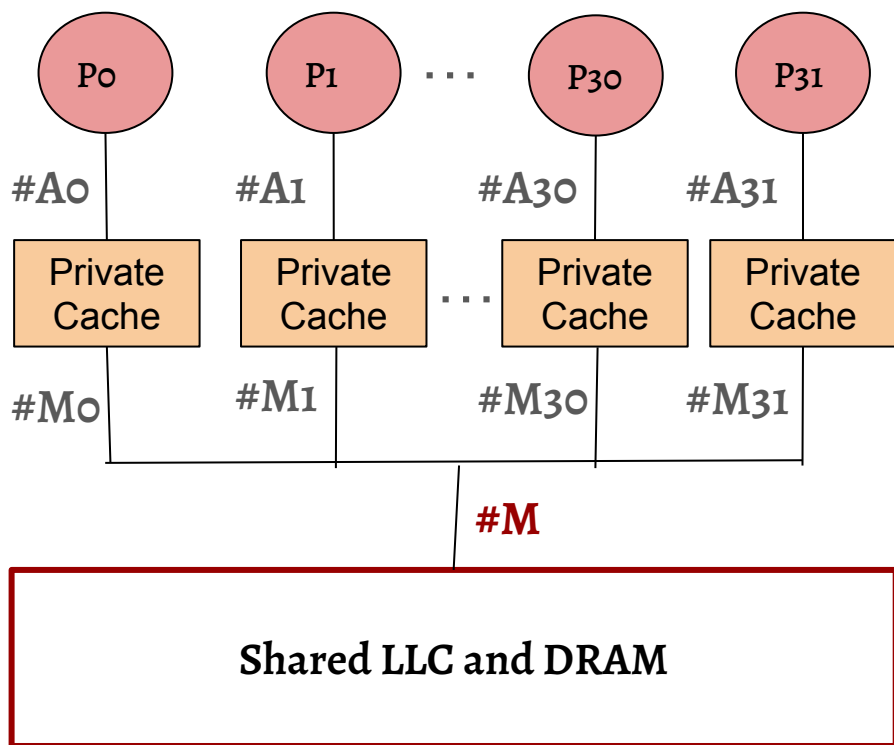
- Ordering across CPUs, atomic operations
- I/O operations and related bottlenecks

3

Cross-cutting Issues

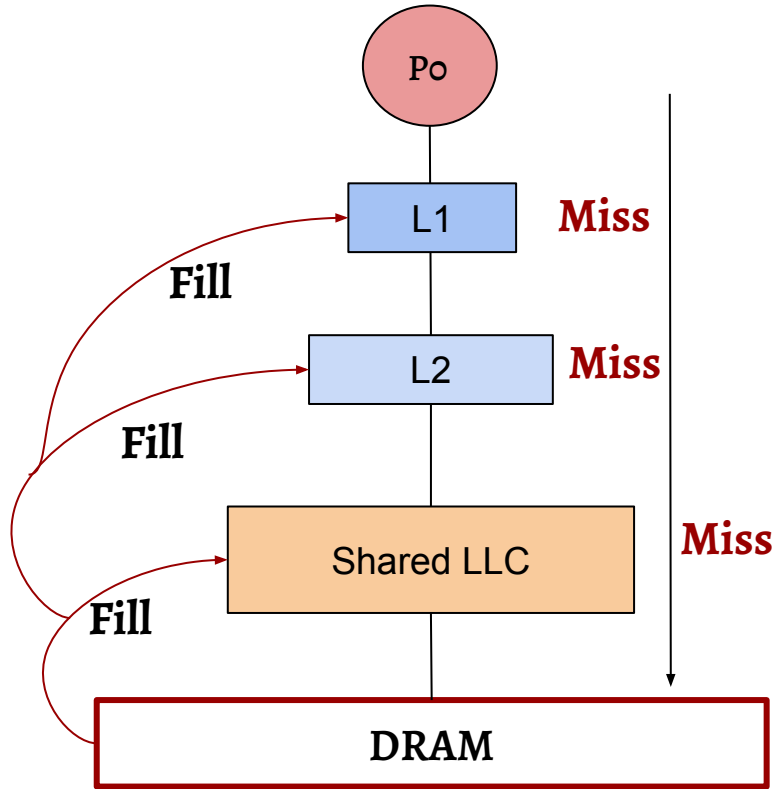
- Cache/TLB coherence, memory consistency
- Supporting atomic ops and memory ordering

The Memory Wall Problem



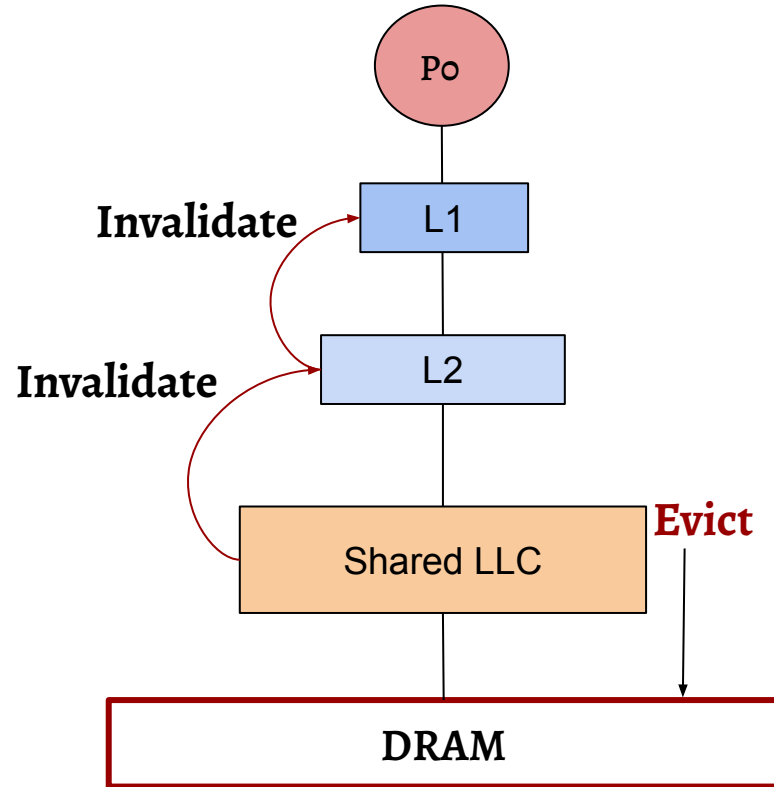
- Working set sizes increasing—more pressure on the SM system
- Different levels of shared resource requirement becomes a challenge
- Unavoidable bursts of requests
- Techniques: Improved private cache policies, banking (NUCA and NUMA), resource-conserving cache and bandwidth management

Clusivity of Memory Hierarchy: Inclusive



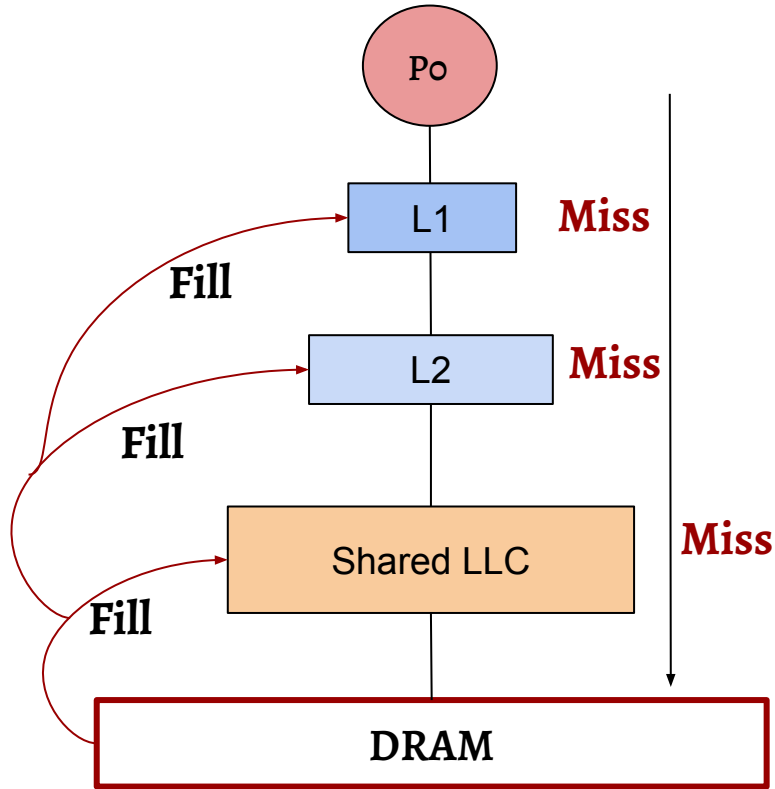
- Inclusive property: If block **B** is in upper level, **B** must be present in lower level
- Cache miss is served by filling the block at all levels.

Clusivity of Memory Hierarchy: Inclusive



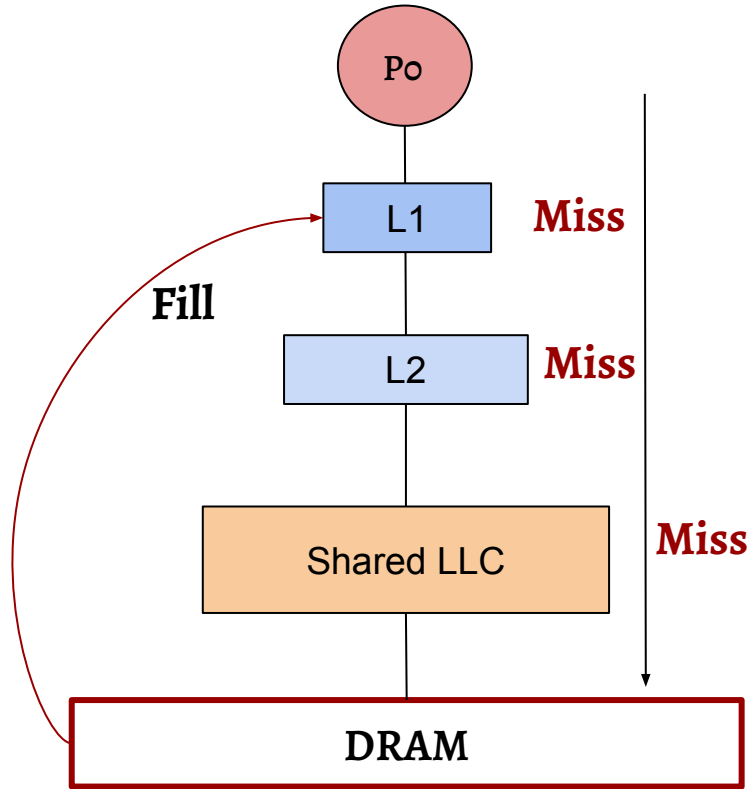
- Inclusive property: If block **B** is in upper level, **B** must be present in lower level
- Cache miss is served by filling the block at all levels.
- Eviction of a block from lower level triggers an invalidation of the block from the upper level (a.k.a. back invalidation)
- DRAM and LLC are always inclusive

Clusivity of Memory Hierarchy: non-inclusive



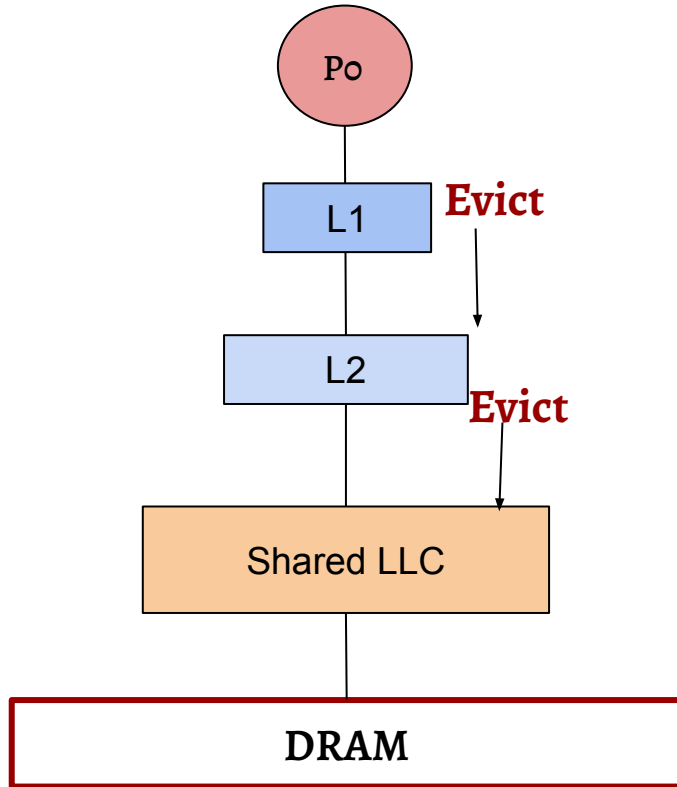
- If block **B** is in upper level, **B** may or may not be present in the lower levels
- Cache miss is served by filling the block at all levels.
- Eviction of a block from lower levels **does not** invalidate block in the upper levels

Clusivity of Memory Hierarchy: exclusive



- Exclusive property: If block **B** is in upper level, **B** should not be present in the lower levels
- Cache miss is served by filling the block into the L1 cache
- Eviction of a block from lower levels **does not** invalidate block in the upper levels

Clusivity of Memory Hierarchy: exclusive



- Inclusive property: If block **B** is in upper level, **B** must be present in lower level
- Cache miss is served by filling the block at all levels.
- Eviction of a block from upper level pushes the block into the immediate next level
- No back invalidation on eviction

Bottlenecks in parallel performance

1

Hardware Bottlenecks

- Shared hardware (LLC, DRAM, Interconnects)
 - Memory wall problem
 - Resource scheduling and management

2

Software Bottlenecks

- **Ordering across CPUs, atomic operations**
- **I/O operations and related bottlenecks**

3

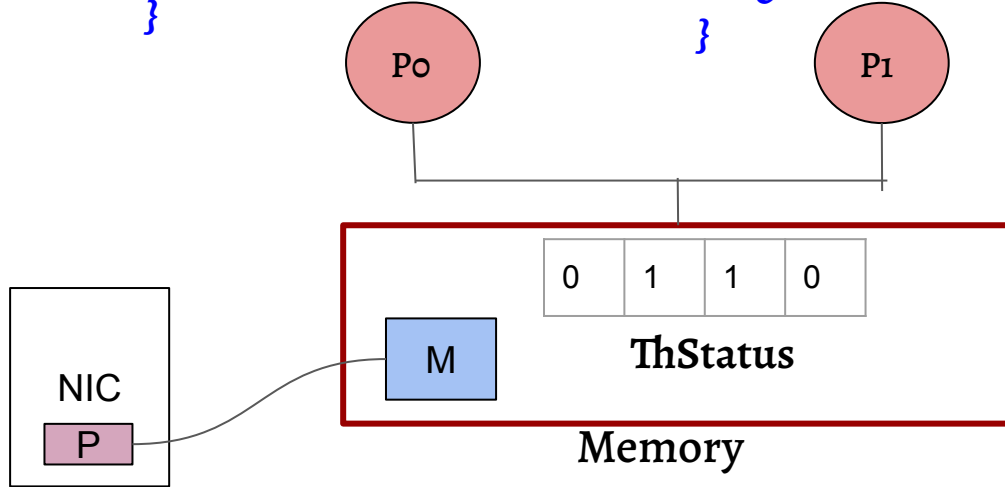
Hybrid Issues

- Cache/TLB coherence, memory consistency
- Supporting atomic ops and memory barriers

Serialization and I/O Bottlenecks

```
HandleClient(){  
  While(!packet) wait();  
  Recv(p, M);  
  Signal(FindFree(ThStatus));  
}
```

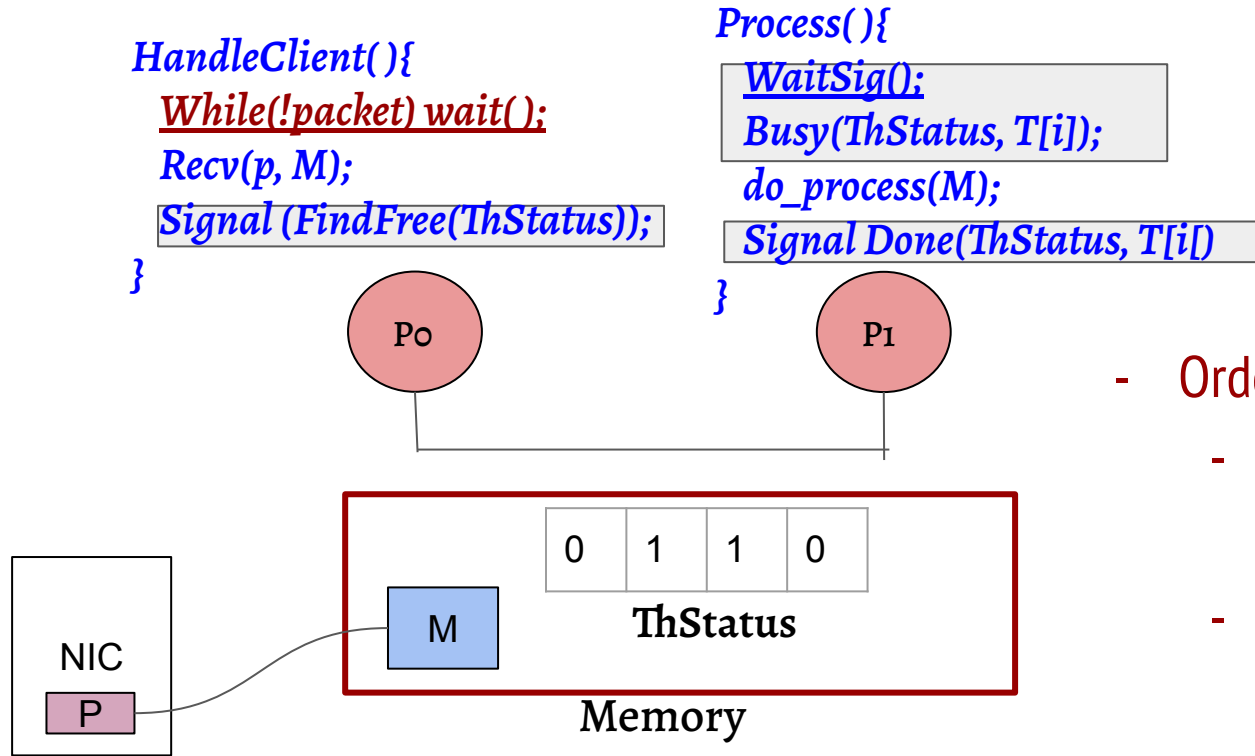
```
Process(){  
  WaitSig(ST[i]);  
  Busy(ThStatus, T[i]);  
  do_process(M);  
  Signal Done(ThStatus, T[i])  
}
```



- Ordering requirement
 - Waiting for I/O, Waiting for software signal
 - OS scheduling can help to some extent

- Parallel tasks mostly communicate through memory

Serialization and I/O Bottlenecks



- Ordering requirement
 - Waiting for I/O, Waiting for software signal
 - OS scheduling can help to some extent

- Parallel tasks mostly communicate through memory

Bottlenecks in parallel performance

1

Hardware Bottlenecks

- Shared hardware (LLC, DRAM, Interconnects)
 - Memory wall problem
 - Resource scheduling and management

2

Software Bottlenecks

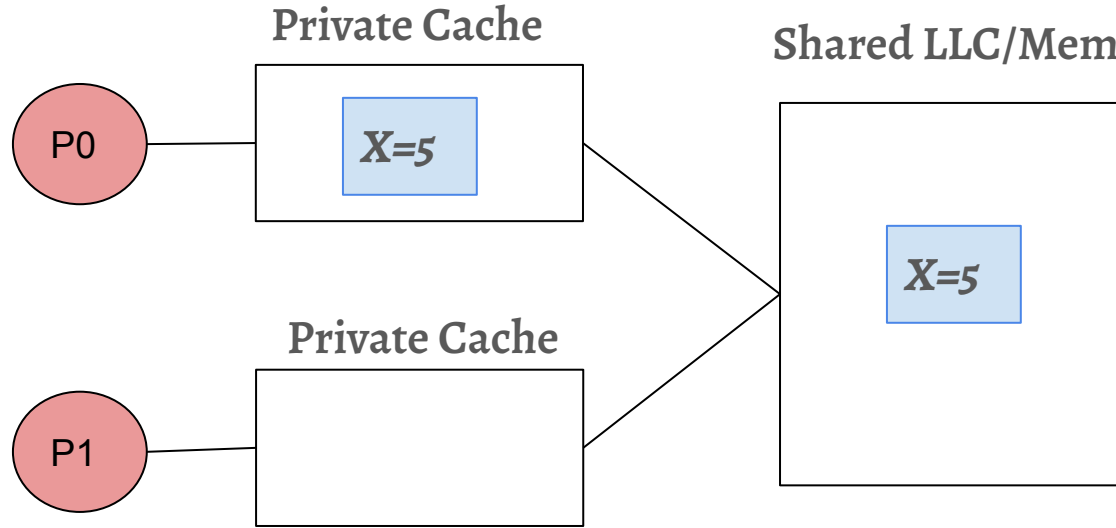
- Ordering across CPUs, atomic operations
- I/O operations and related bottlenecks

3

Hybrid Issues

- **Cache/TLB coherence, memory consistency**
- Supporting atomic ops and memory ordering

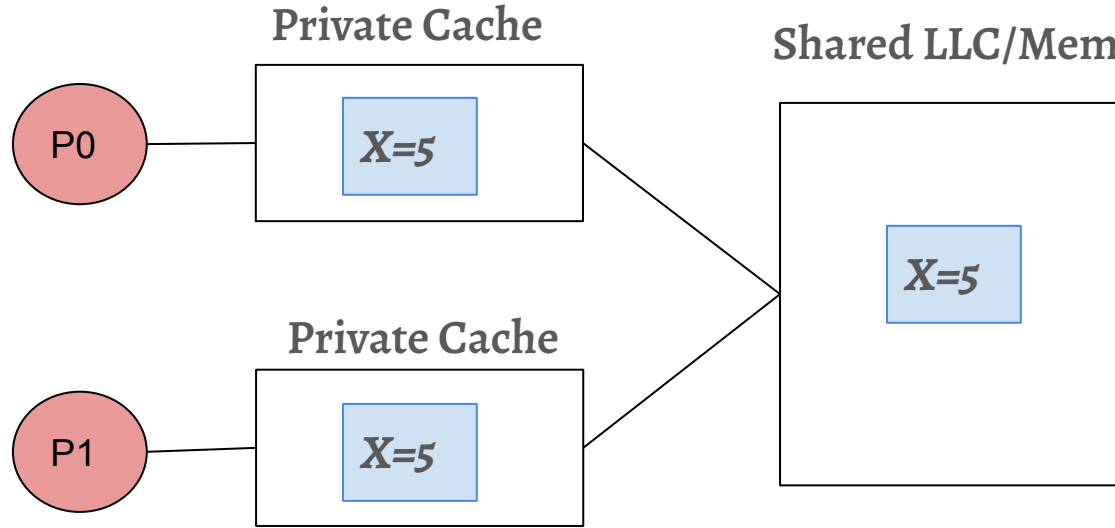
Cache Coherence



(T=1) Po: *Read (X)*
(T=2) P1: *Read (X)*
(T=3) Po: *Write (X, 10)*
(T=4) P1: *Read (X)*

- Private caches (can be of multiple levels) do not contain X
- We assume that memory access to the private caches merge at shared LLC (if present) or at the shared memory.
- P0: Perform Read(X)

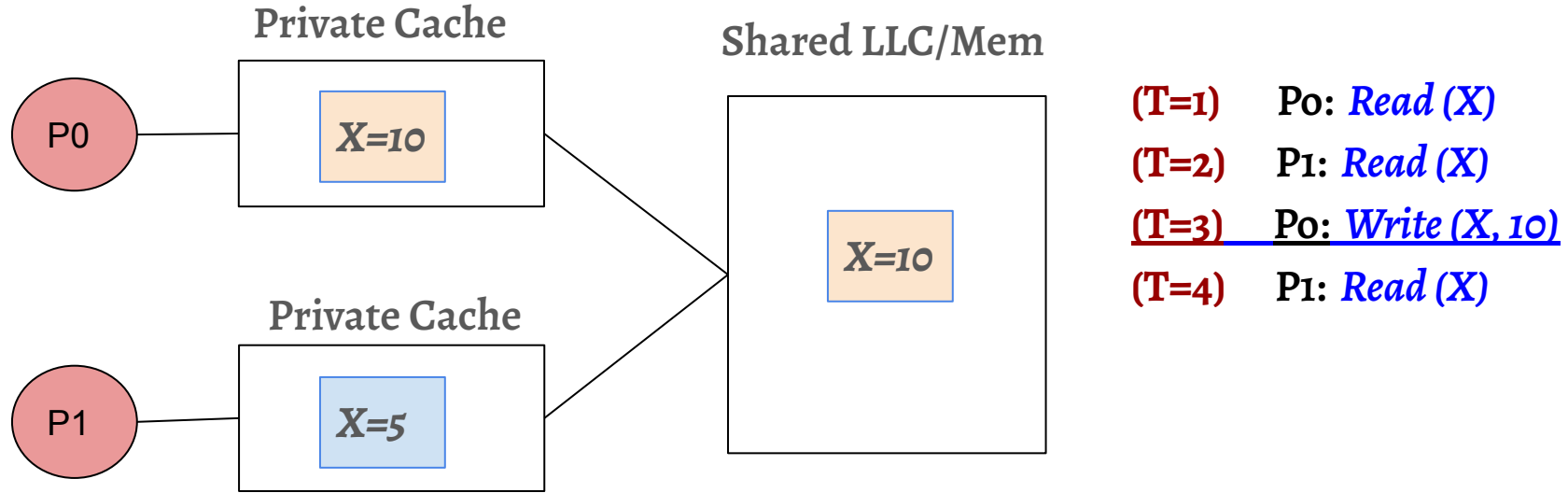
Cache Coherence



(T=1) Po: *Read (X)*
(T=2) P1: *Read (X)*
(T=3) Po: *Write (X, 10)*
(T=4) P1: *Read (X)*

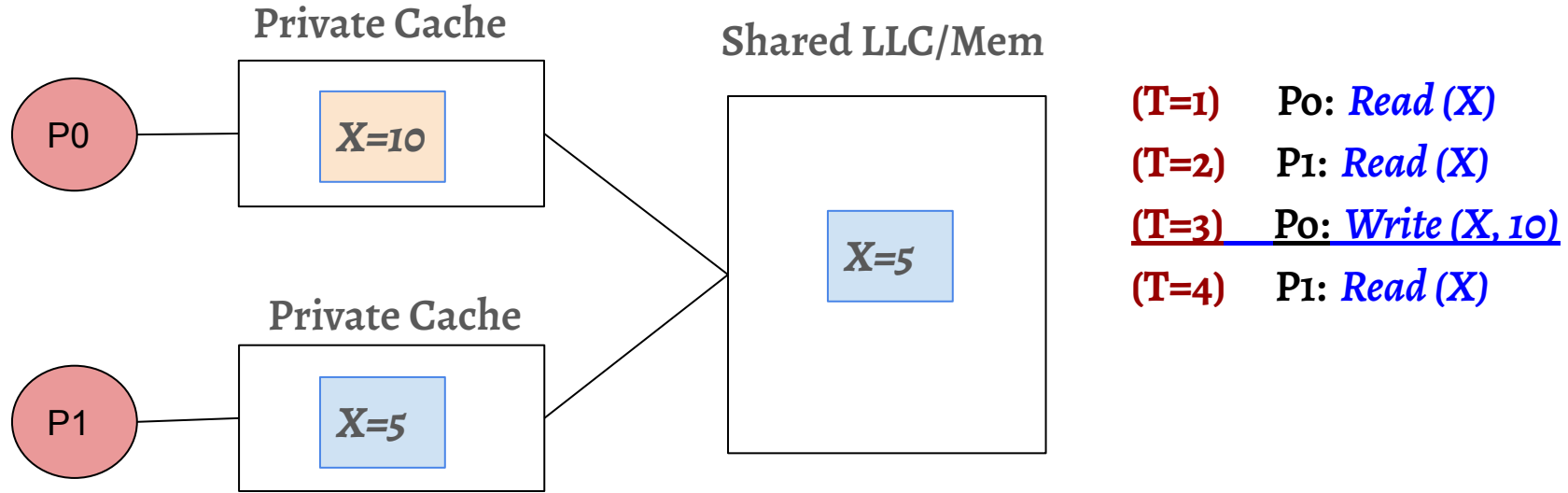
- Private caches (can be of multiple levels) do not contain X
- We assume that memory access to the private caches merge at shared LLC (if present) or at the shared memory.
- P1: Perform Read(X)

Cache Coherence - Write-through cache



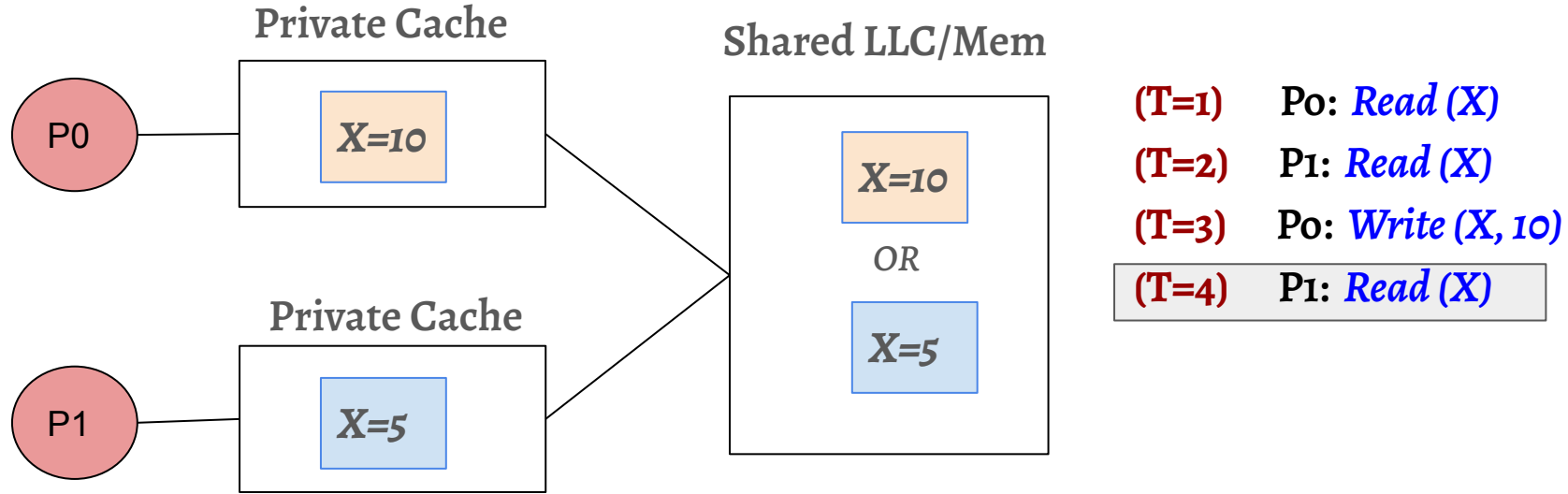
- Write-through caches would perform write to the private caches and propagate the write to the shared LLC and memory
- P0: Perform *Write(X, 10)*

Cache Coherence - Write-back cache



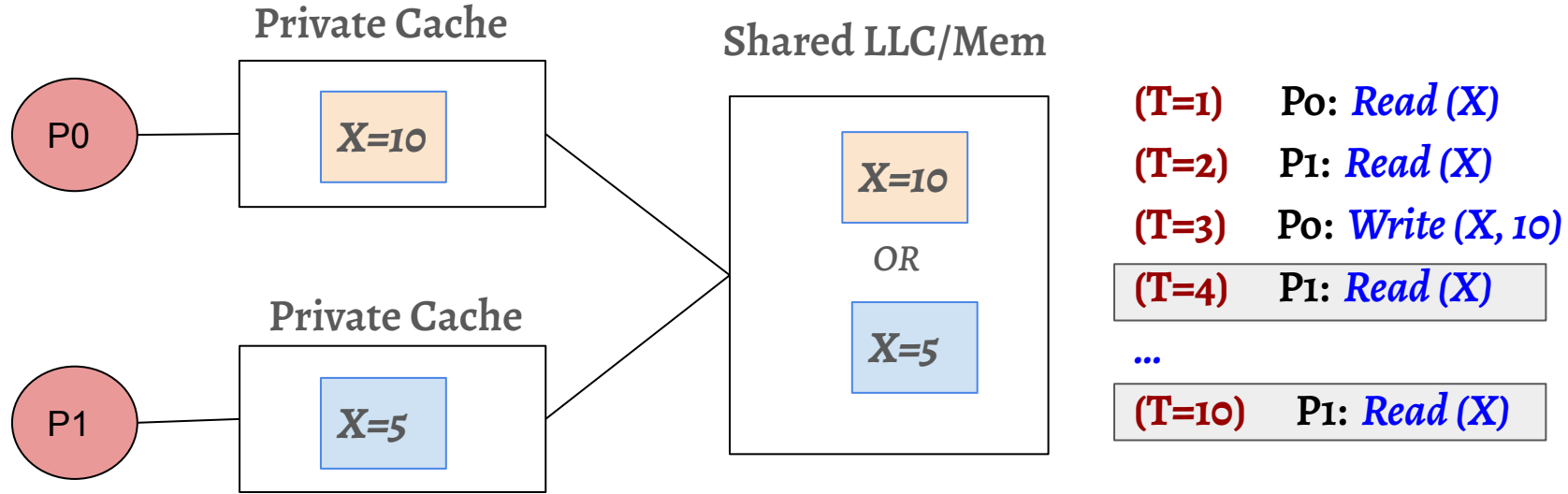
- Write-through caches would perform write to the private caches and propagate the write to the shared LLC and memory
- P0: Perform *Write(X, 10)*

Cache Coherence - Read after write



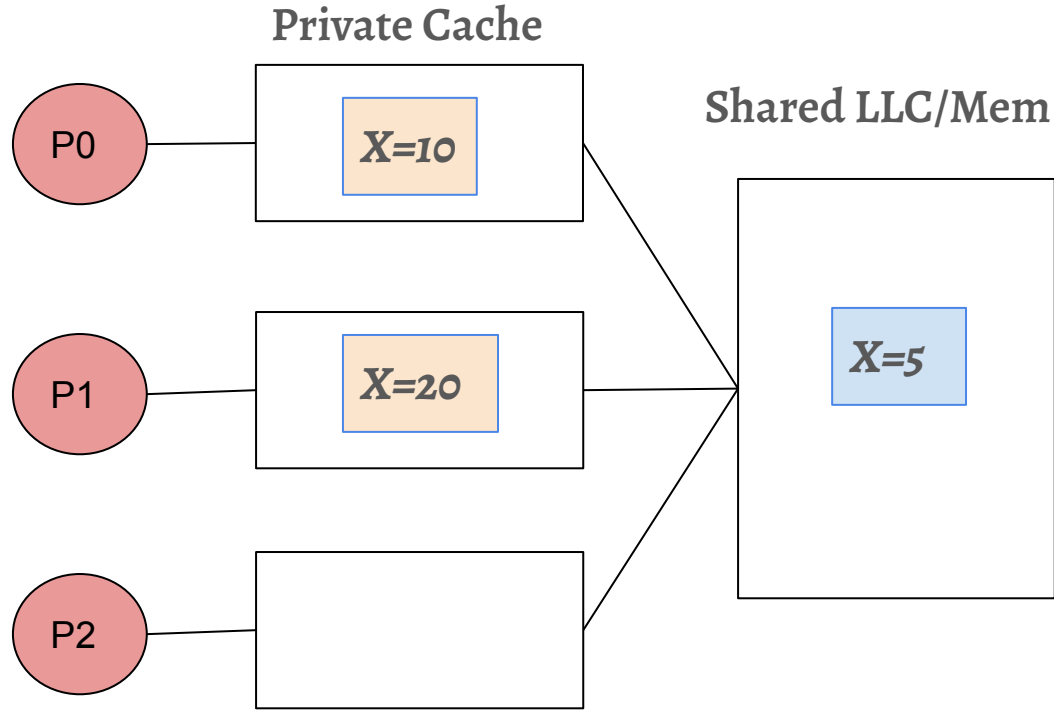
- P1: Perform Read(X, 10)
- What should be the value read by P1?
 - Can the hardware ensure P1 to read X=10?
 - Is this read semantics always required by the programmer?

Cache Coherence - Read after write



- P1: Perform Read(X, 10)
- What should be the value read by P1 (at T=10)?
 - P1 should “eventually” read X=10 (if X is not allocated on a register) i.e., P1 can not infinitely read X=5

Cache Coherence - Order of Writes



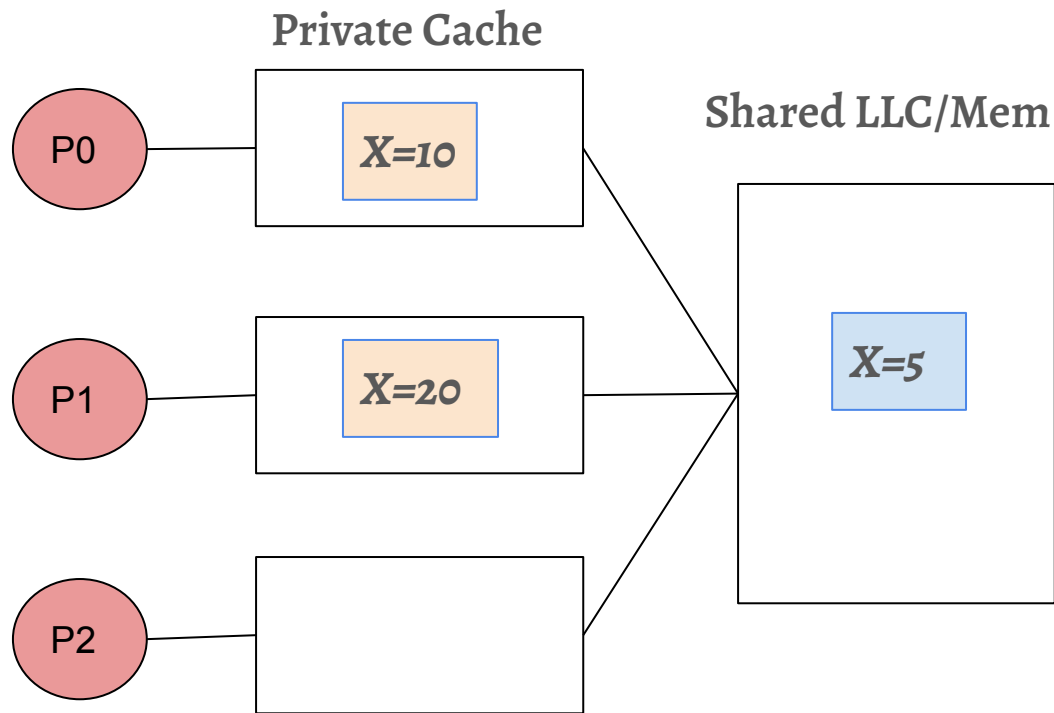
(T=1) P0: *Write (X, 10)*

(T=2) P1: *Write (X, 20)*

(T=3) P2: *Read (X)*

- In what order the writes to 'X' should be visible to P2? What if there is no ordering guarantee?

Cache Coherence - Order of Writes



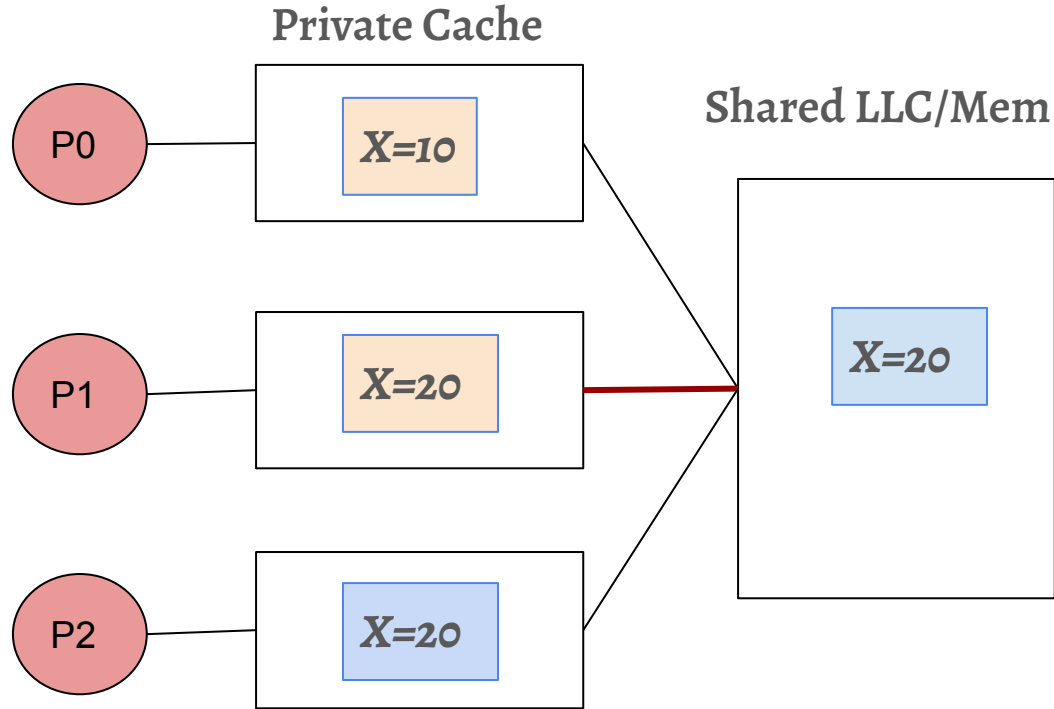
(T=1) P0: *Write (X, 10)*

(T=2) P1: *Write (X, 20)*

(T=3) P2: *Read (X)*

- In what order the writes to 'X' should be visible to P2? What if there is no ordering guarantee?
- Let us assume that write-back from P1's private cache takes place before P0's private cache

Cache Coherence - Order of Writes



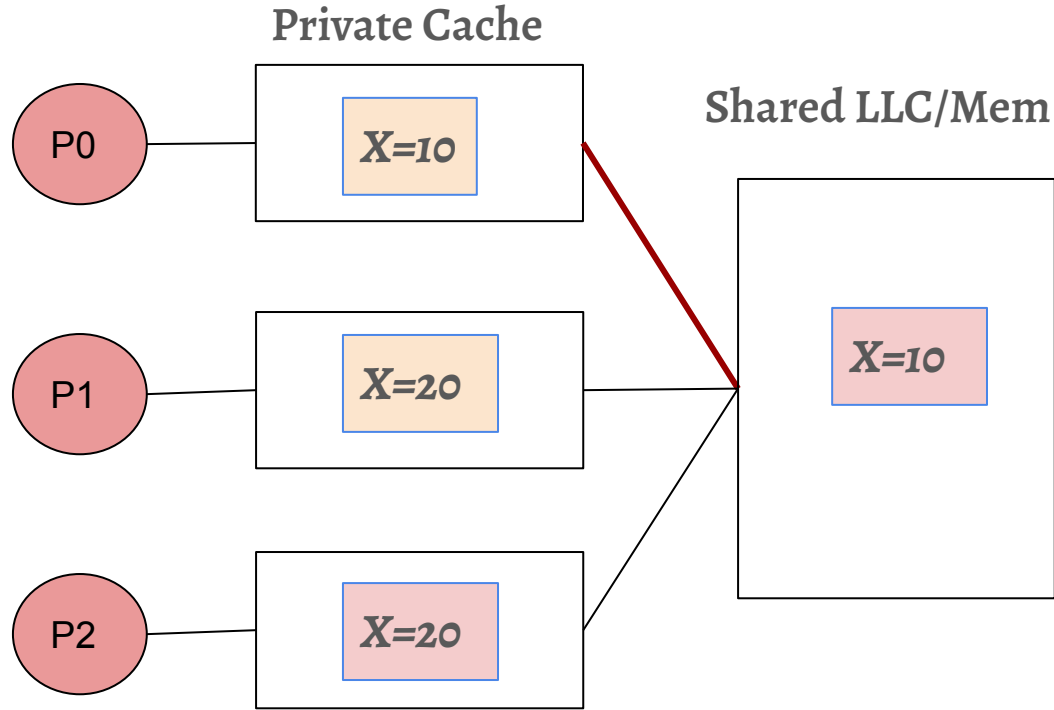
(T=1) P0: *Write (X, 10)*

(T=2) P1: *Write (X, 20)*

(T=3) P2: *Read (X)*

- In what order the writes to 'X' should be visible to P2? What if there is no ordering guarantee?
- Let us assume that write-back from P1's private cache takes place before P0's private cache

Cache Coherence - Order of Writes



(T=1) P0: *Write (X, 10)*

(T=2) P1: *Write (X, 20)*

(T=3) P2: *Read (X)*

- In what order the writes to 'X' should be visible to P2? What if there is no ordering guarantee?
- Let us assume that write-back from P1's private cache takes place before P0's private cache

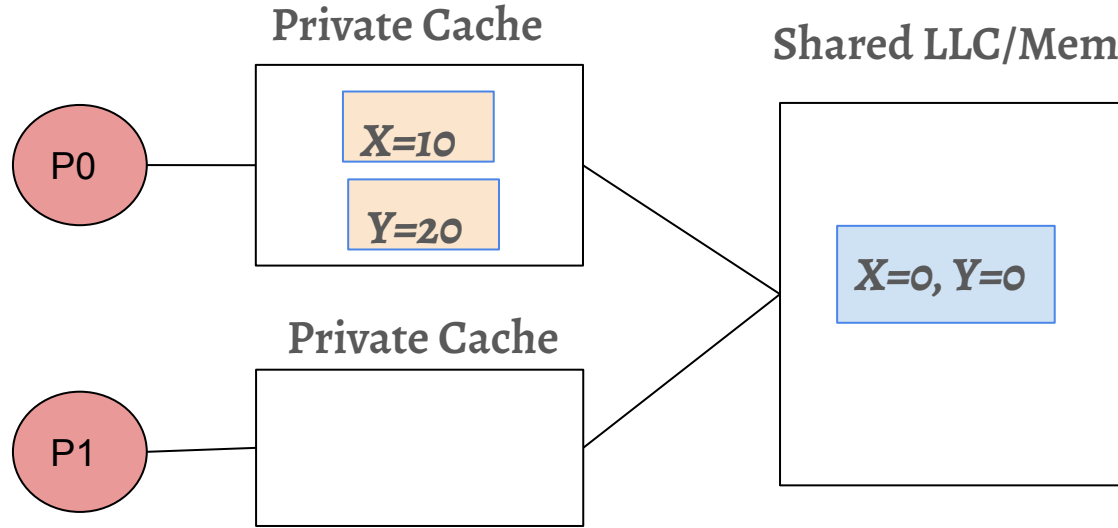
For a cache block, ensuring correct write visibility with ordering is required.

Cache Coherence Techniques

- Employ a protocol to maintain coherence. Two classes of protocol implementation – (1) Snoop (2) Directory-based
- Snoop
 - A processor arbitrate for the bus before performing actions on a cache block
 - Broadcast a corresponding protocol message in the shared bus
 - Each processor snoop for message, find the cache block and perform actions (if present)
- Directory-based
 - A central directory (at LLC) maintains state of each cache block present in private caches of all processors (in SMP)
 - Directory controller maintains the correctness by following the protocol

Snoop and Directory-based introduce additional messages and serialization.

Memory Consistency



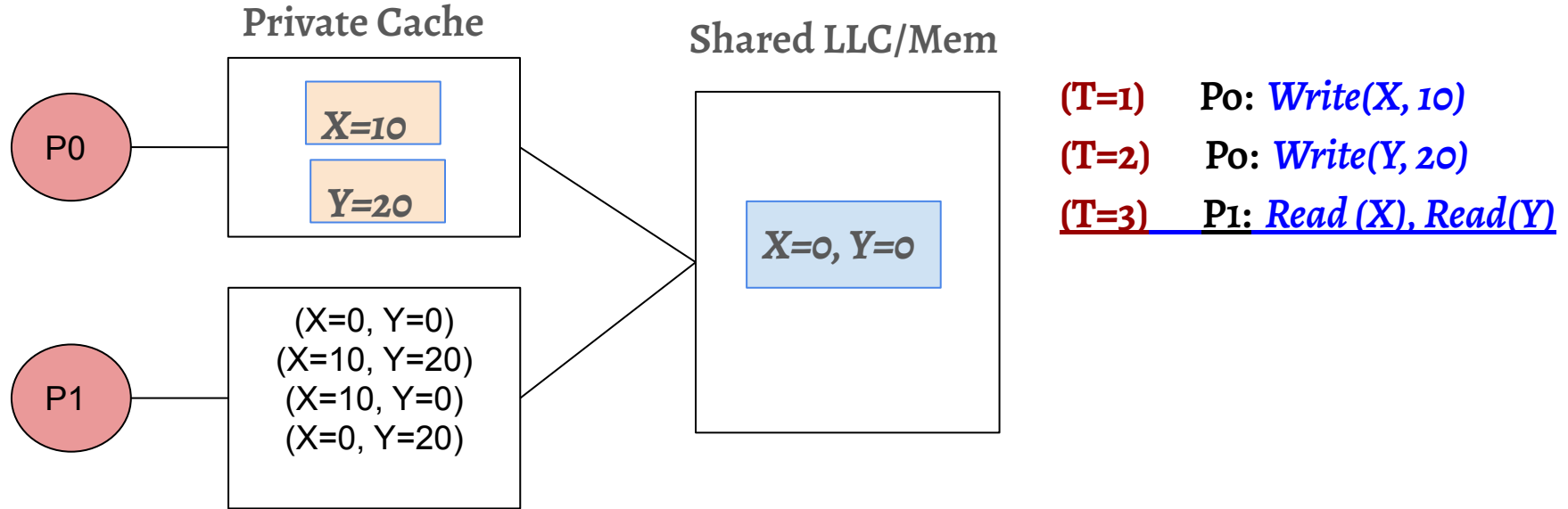
(T=1) **Po:** Write(X, 10)

(T=2) **Po:** Write(Y, 20)

(T=3) **P1:** Read (X), Read(Y)

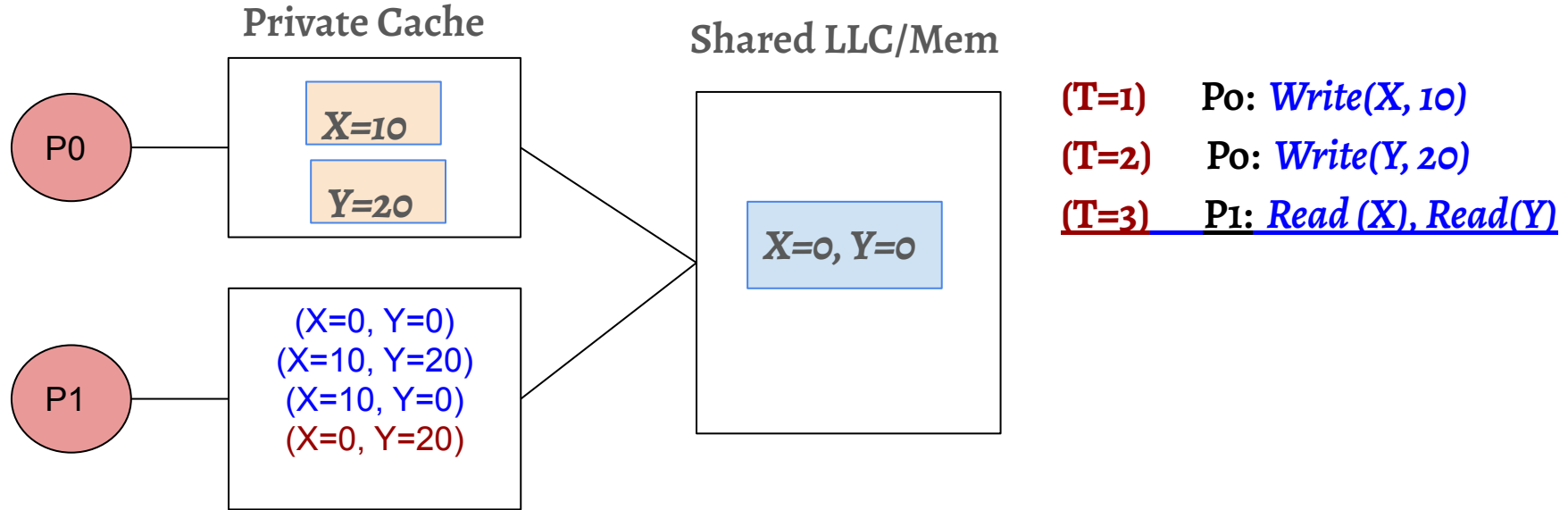
- Assume $X=0$ and $Y=0$ before it is written from P0
- The update is performed in the private cache of P0

Memory Consistency



- What should be the values of X and Y when they are read from P1?
 - What is the programmers expectation?

Memory Consistency

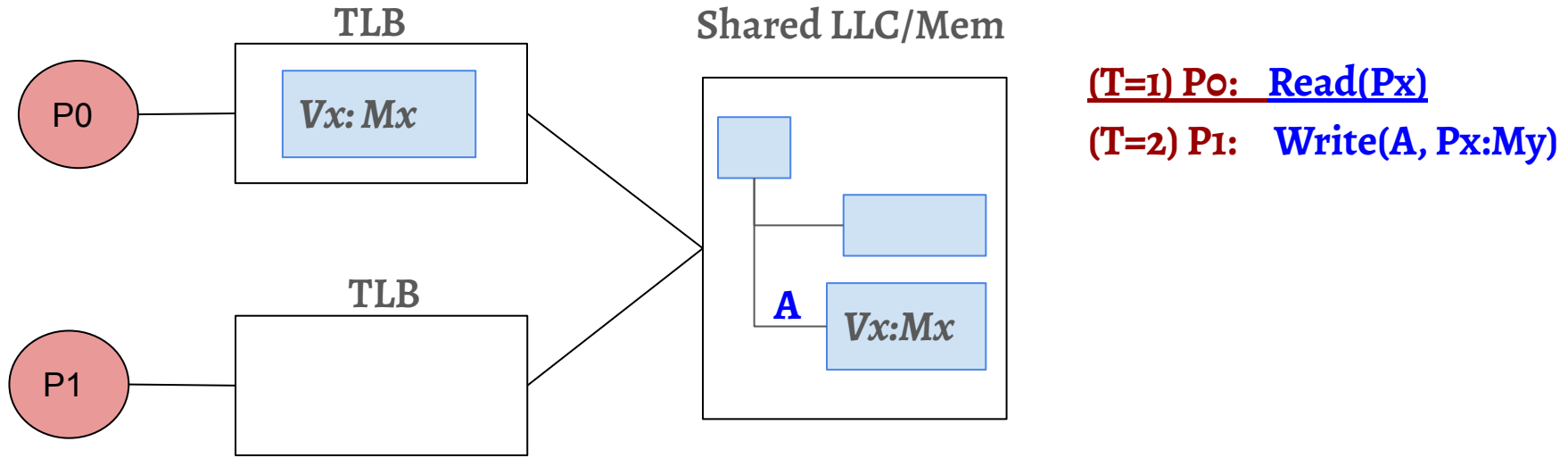


- What should be the values of X and Y when they are read from P1?
 - What is the programmers expectation? ($X=0$ and $Y=20$) is not “sequential”

Memory consistency deals with “when” the write is visible.

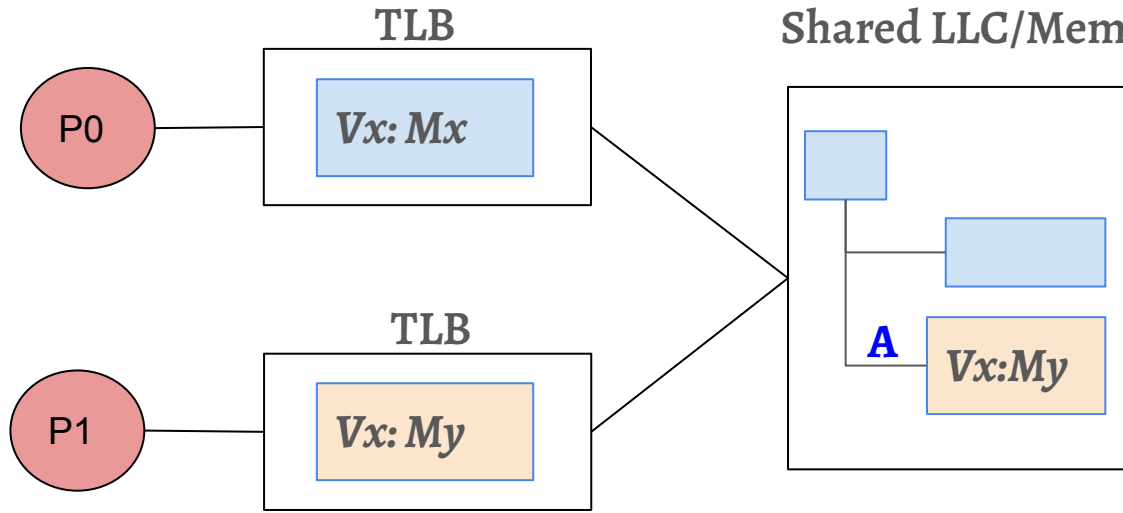
Sequential consistency: performance implications, Relaxed consistency: difficult to code

TLB Coherence



- P0 performs read with virtual address Vx which is translated to Mx using the page table entry stored at address A
- The corresponding TLB entry is made in the P0 TLB (TLBs are private, not shared across cores)

TLB Coherence

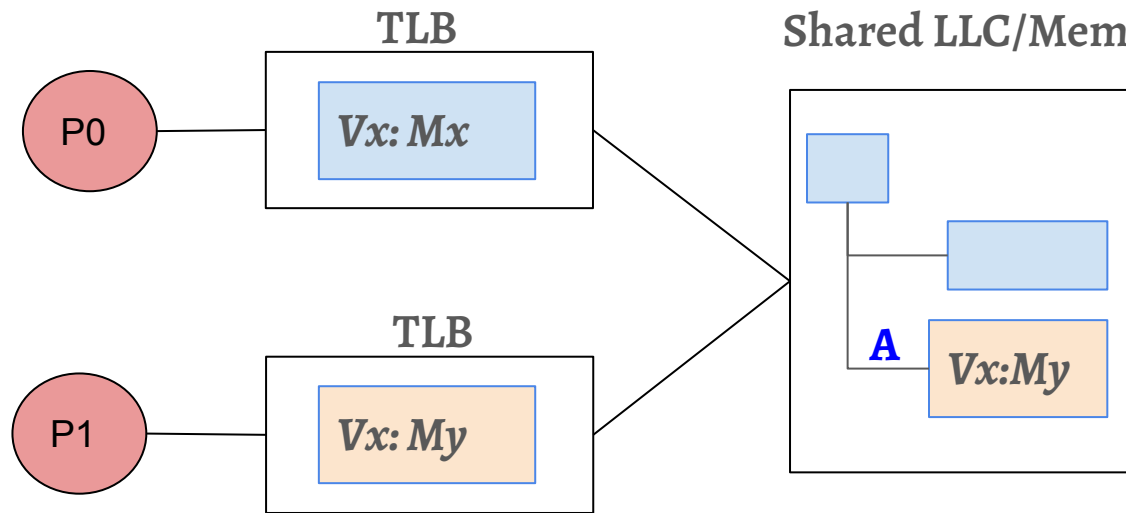


(T=1) P0: **Read(Px)**

(T=2) P1: **Write(A, Px:My)**

- P1 updates the page table entry stored at address **A**
- What are the issues if P0 continues to use the old translation?

TLB Coherence

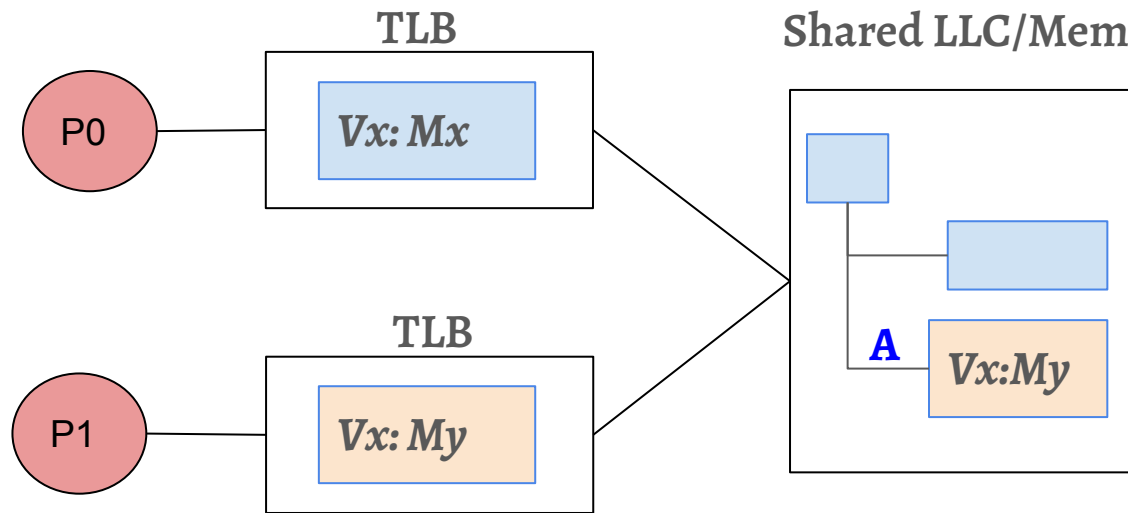


(T=1) P0: Read(Px)

(T=2) P1: Write(A, Px:My)

- P1 updates the page table entry stored at address **A**
- What are the issues of P0 continues to use the old translation?
 - Isolation can be broken (because of permission etc.)
 - Correctness issues (system crash etc.)

TLB Coherence

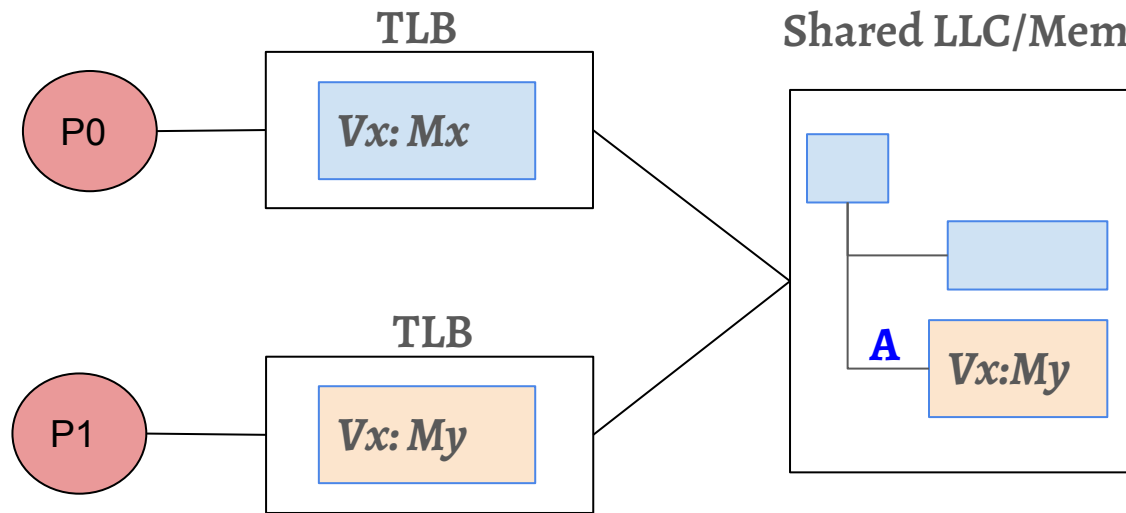


(T=1) P0: Read(Px)

(T=2) P1: Write(A, Px:My)

- P1 updates the page table entry stored at address **A**
- All other cores using the translation from their TLB must use the updated translation
- Typically, this operation is relatively infrequent (compared to caches) and performed by system software (OS or hypervisors)

TLB Coherence



(T=1) P0: Read(Px)

(T=2) P1: Write(A, Px:My)

- P1 updates the page table entry stored at address **A**
- All other cores using the translation from their TLB must use the updated translation

Solutions at hardware and/or software layers, non-trivial performance overheads.

Bottlenecks in parallel performance

1

Hardware Bottlenecks

- Shared hardware (LLC, DRAM, Interconnects)
 - Memory wall problem
 - Resource scheduling and management

2

Software Bottlenecks

- Ordering across CPUs, atomic operations
- I/O operations and related bottlenecks

3

Hybrid Issues

- Cache/TLB coherence, memory consistency
- **Supporting atomic ops and memory ordering**

Software synchronization requirements

- Examples of synchronization requirements
- Two memory operations should follow a sequential order
 - If not followed, can create issues for some external state e.g., I/O, RDTSC
 - Hardware provide serializing instructions (e.g., memory fence)

Software synchronization requirements

- Examples of synchronization requirements
- Two memory operations should follow a sequential order
 - If not followed, can create issues for some external state e.g., I/O, RDTSC
 - Hardware provide serializing instructions (e.g., memory fence)
- T1 should read a value after T2 has updated it
- An object should be atomically updated in memory
- A block of code is executed in a serial manner
 - Some basic atomic operations (e.g., atomic CAS, atomic XCHG) can be built upon to complex synchronization primitives

Software synchronization requirements

- Examples of synchronization requirements
- Two memory operations should follow a sequential order
 - If not followed, can create issues for some external state e.g., I/O, RDTSC
 - Hardware provide serializing instructions (e.g., memory fence)
- T1 should read a value after T2 has updated it
- An object should be atomically updated in memory
- A block of code is executed in a serial manner
 - Some basic atomic operations (e.g., atomic CAS, atomic XCHG) can be built upon to complex synchronization primitives
- Can we implement a 'lock' without any hardware guarantees?

Software synchronization requirements

- Examples of synchronization requirements
- Two memory operations should follow a sequential order
 - If not followed, can create issues for some external state e.g., I/O, rdtsc
 - Hardware provide serializing instructions (e.g., memory fence)
- T1 should read a value after T2 has updated it
- An object should be atomically updated in memory
- A block of code is executed in a serial manner
 - Some basic atomic operations (e.g., atomic CAS, atomic XCHG) can be built upon to complex synchronization primitives

Implementing a lock in software

```
int flag[2] = {0,0}; int turn = 0;
void lock (int id) /*id = 0 or 1 */
{
    flag[id] = 1;
    turn = id ^ 1;
    while(flag[id ^ 1] && turn == (id ^ 1));
}
void unlock (int id)
{
    flag[id] = 0;
}
```

- What is/are the assumptions for this locking scheme to work?
- Hint: If we reorder the first two lines of the “lock” function will the locking scheme work?

Implementing a lock in software

```
int flag[2] = {0,0}; int turn = 0;
void lock (int id) /*id = 0 or 1 */
{
    flag[id] = 1;
    turn = id ^ 1;
    while(flag[id ^ 1] && turn == (id ^ 1));
}
void unlock (int id)
{
    flag[id] = 0;
}
```

- What is/are the assumptions for this locking scheme to work?
- Hint: If we reorder the first two lines of the “lock” function will the locking scheme work?
- It will lead to breaking mutual exclusion T0: turn=1, T1: flag[1] = 1 turn=0, While(false) → Acquire, T0: flag[0] = 1, While(false) → Acquire

Atomic primitives and ordering mechanisms can facilitate S/W synchronization requirements