Advanced Computer Architectures

(High Performance Processors and Systems)

Parallel Processors: MIMD Architectures

Politecnico di Milano

v0

Alessandro Verosimile <alessandro.verosimile@polimi.it>

Marco D. Santambrogio <marco.santambrogio@polimi.it>

Outline

- MIMD Architectures
- Memory organization
- Communication models
- Cache Coherence

Why MIMD?

- MIMDs are flexible – they can function as single-user machines for high performances on one application, as multiprogrammed multiprocessors running many tasks simultaneously, or as some combination of such functions;

- Can be built starting from standard CPUs (such is the present case nearly for all multiprocessors!).

MIMD

- To exploit a MIMD with ***n*** processors
  - at least ***n*** threads or processes to execute
  - independent threads typically identified by the programmer or created by the compiler.
    - *thread-level parallelism*.

- Thread: from a large, independent process to parallel iterations of a loop.
- Important: *parallelism is identified by the software* (not by hardware as in superscalar CPUs!)... keep this in mind, we'll use it!

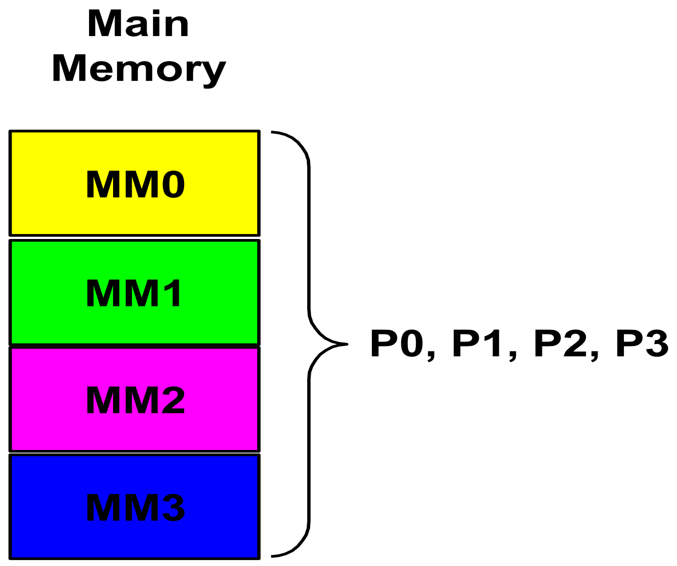
How do parallel processors share data?

Memory Address Space Model

- Single logically shared address space: A memory reference can be made by any processor to any memory location $\Rightarrow$ Shared Memory Architectures.
  - The address space is shared among processors: The same physical address on 2 processors refers to the same location in memory.

Memory Address Space Model

Single logically shared address space



Shared Address

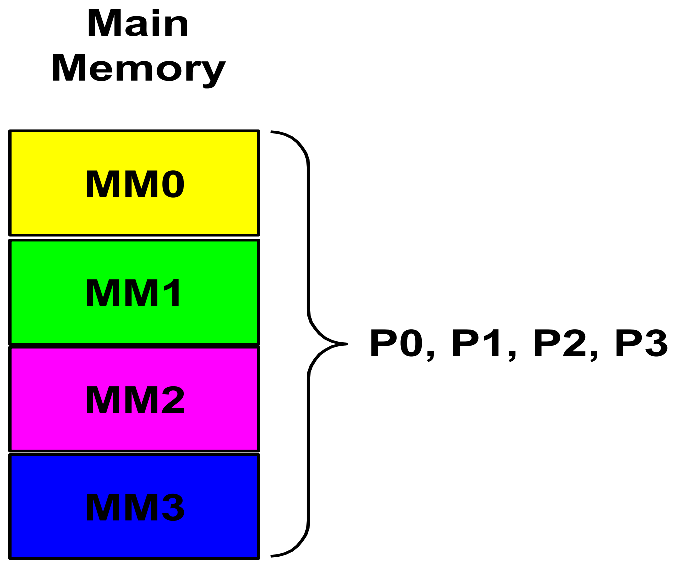
- The processors communicate among them through shared variables in memory.
- Implicit management of the communication through load/store operations to access any memory locations.
  - Oldest and most popular model
- Shared memory does not mean that there is a single centralized memory.

Memory Address Space Model

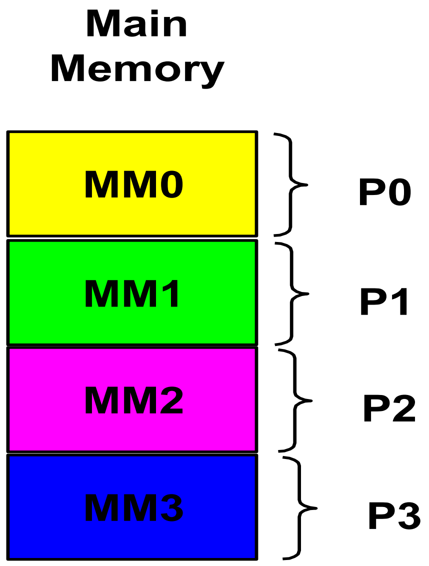
- Single logically shared address space: A memory reference can be made by any processor to any memory location ⇒ Shared Memory Architectures.
  - The address space is shared among processors: The same physical address on 2 processors refers to the same location in memory.
- Multiple and private address spaces: The processors communicate among them through send/receive primitives ⇒ Message Passing Architectures.
  - The address space is logically disjoint and cannot be addressed by different processors: the same physical address on 2 processors refers to 2 different locations in 2 different memories.

Memory Address Space Model

Single logically shared address space



Multiple and private address spaces



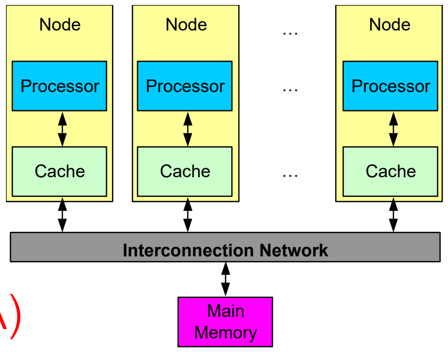
Multiple and Private Addresses

- The processors communicate among them through sending/receiving messages.
- Explicit management of the communication through send/receive primitives to access private memory locations.
- No cache coherency problem among processors.

Physical Memory Organization

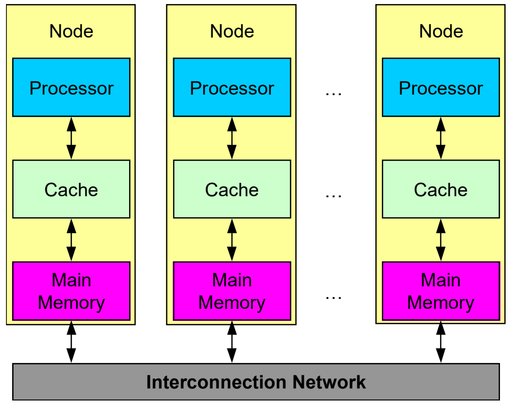
Centralized shared-memory architectures

- at most few dozen processor chips (< 100 cores)
- Large caches, single memory multiple banks
- Often called symmetric multiprocessors (SMP) and the style of architecture called Uniform Memory Access (UMA)



Distributed memory architectures

- To support large processor counts
- Requires high-bandwidth interconnect
- Cons: data communication among processors
- Non Uniform Memory Access (NUMA)



Physical Memory Organization

- The concepts of addressing space (single/multiple) and the physical memory organization (centralized/distributed) are orthogonal to each other.

- Multiprocessor systems can have single addressing space and distributed physical memory.

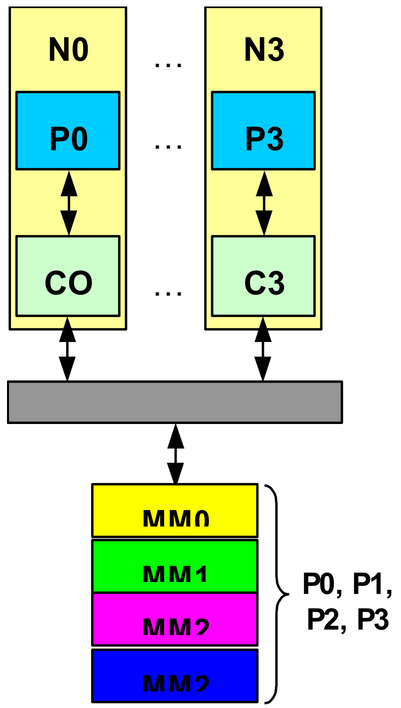
Address Space vs. Physical Memory Org.

Single Logically Shared Address Space
(Shared-Memory Architectures)

Address Space vs. Physical Memory Org.

Single Logically Shared Address Space (Shared-Memory Architectures)

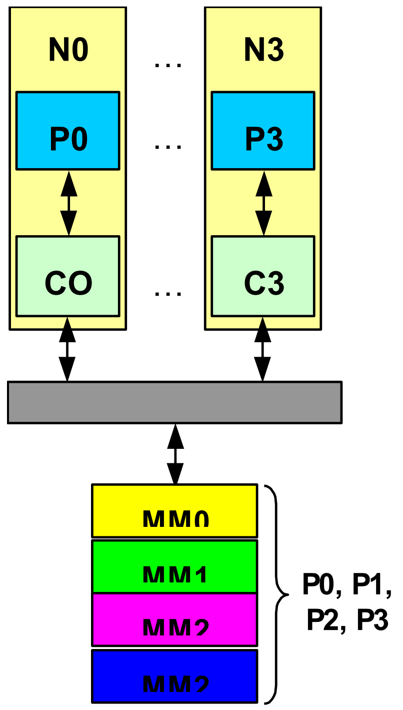
Centralized Memory



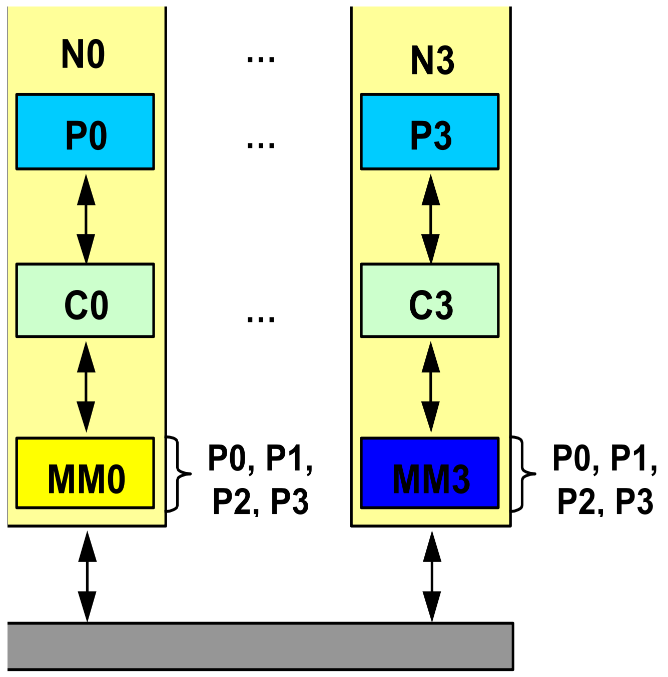
Address Space vs. Physical Memory Org.

Single Logically Shared Address Space (Shared-Memory Architectures)

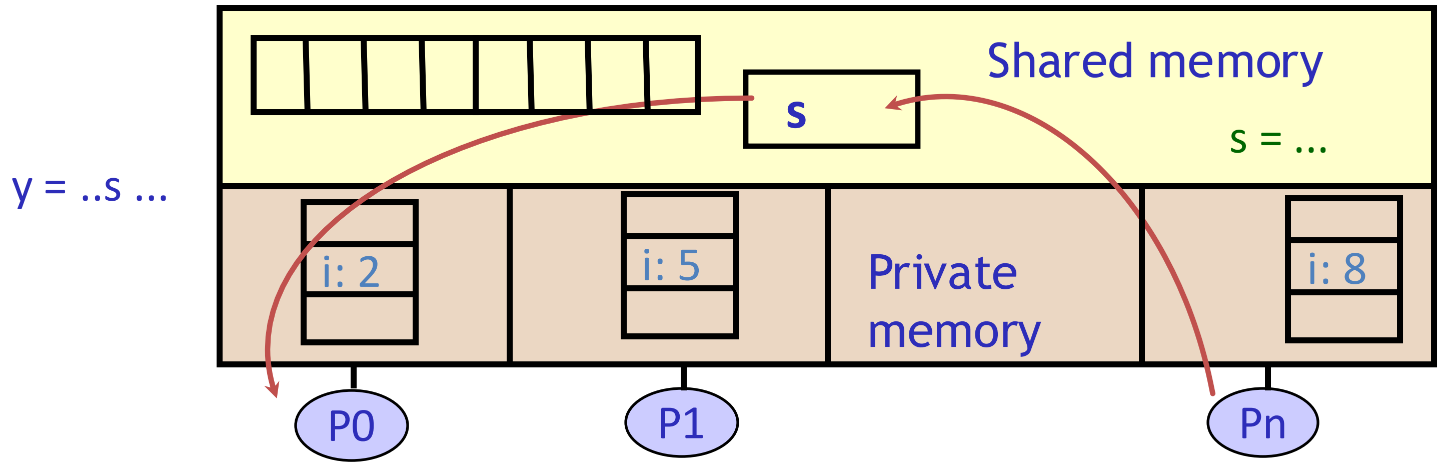
Centralized Memory



Distributed Memory



Programming Model 1: Shared Memory



- Program is a collection of threads of control.
  - Can be created dynamically, mid-execution, in some languages
- Each thread has a set of private variables, e.g., local stack variables
- Also a set of shared variables, e.g., static variables, shared common blocks, or global heap.
  - Threads communicate implicitly by writing and reading shared variables.
  - Threads coordinate by synchronizing on shared variables

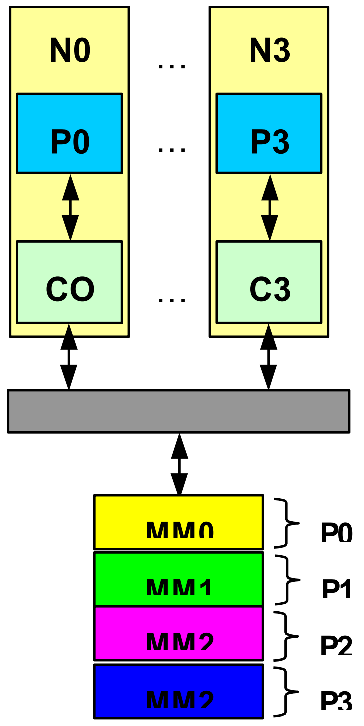
Address Space vs. Physical Memory Org.

Multiple and Private Address Space
(Message Passing Architectures)

Address Space vs. Physical Memory Org.

Multiple and Private Address Space (Message Passing Architectures)

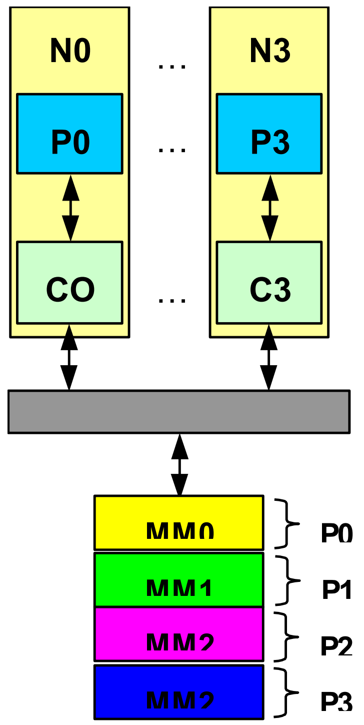
Centralized Memory



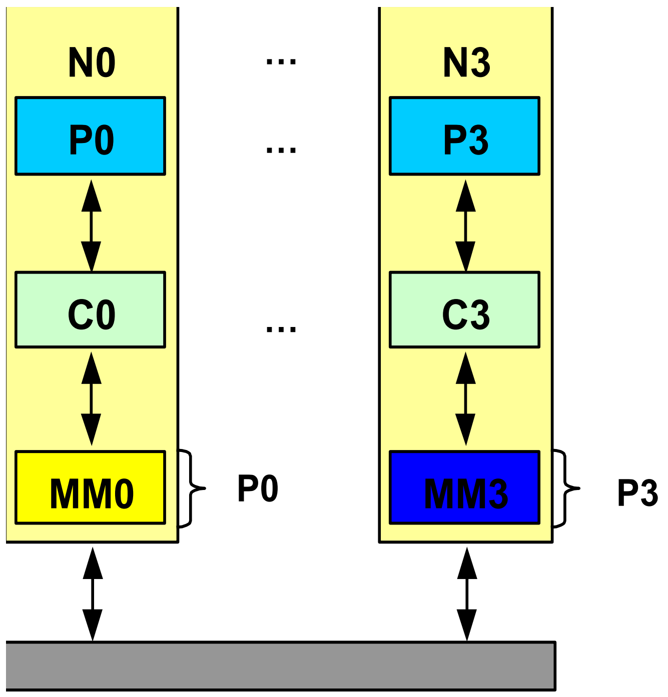
Address Space vs. Physical Memory Org.

Multiple and Private Address Space (Message Passing Architectures)

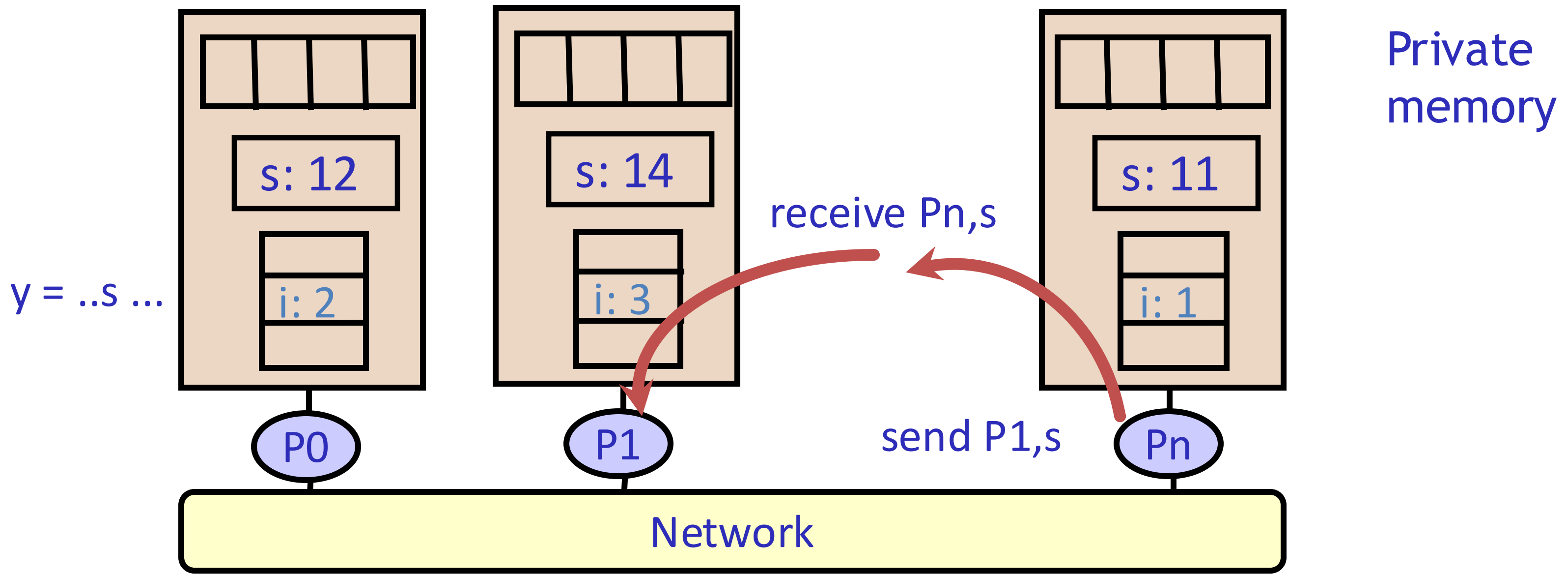
Centralized Memory



Distributed Memory



Programming Model 2: Message Passing



- Program consists of a collection of named processes.
  - Usually fixed at program startup time
  - Thread of control plus local address space -- NO shared data.
  - Logically shared data is partitioned over local processes.
- Processes communicate by explicit send/receive pairs
  - Coordination is implicit in every communication event.
  - MPI (Message Passing Interface) is the most commonly used SW

Which is better? SM or MP?

- Depends on the program!
- Both are “communication Turing complete”
  - i.e. can build Shared Memory with Message Passing and vice-versa

Which is better? SM or MP?

- Advantages of Shared Memory:
  - Implicit communication (loads/stores)
  - Low overhead when cached
- Disadvantages of Shared Memory:
  - Complex to build in way that scales well
  - Requires synchronization operations
  - Hard to control data placement within caching system

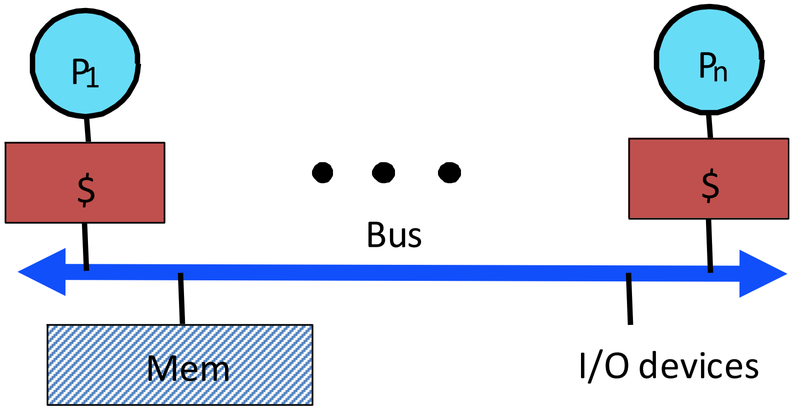
Which is better? SM or MP?

- Advantages of Message Passing
  - Explicit Communication (sending/receiving of messages)
  - Easier to control data placement (no automatic caching)
- Disadvantages of Message Passing
  - Message passing overhead can be quite high
  - More complex to program
  - Introduces question of reception technique (interrupts/polling)

Message Passing Massively Parallel Processors Problems

- All data layout must be handled by software
  - cannot retrieve remote data except with message request/reply
- Message passing has high software overhead
  - early machines had to invoke OS on each message (100μs-1ms/message)
  - even user level access to network interface has dozens of cycles overhead (NI might be on I/O bus)
  - sending messages can be cheap (just like stores)
  - receiving messages is expensive, need to poll or interrupt

Bus-Based Symmetric Shared Memory



- Dominate the server market even now
  - Building blocks for larger systems; arriving to desktop
- Attractive as throughput servers and for parallel programs
  - Fine-grain resource sharing
  - Uniform access via loads/stores
  - Automatic data movement and coherent replication in caches
  - Cheap and powerful extension
- Normal uniprocessor mechanisms to access data
  - Key is extension of memory hierarchy to support multiple processors

Shared Memory Machines

- Two main categories
  - Non cache coherent
  - Hardware cache coherent
- Will work with any data placement (but might be slow)
  - Can choose to optimize only critical portions of code
- Load and store instructions used to communicate data
  - No OS involvement
  - Low software overhead
- Usually some special synchronization primitives
- In large scale systems, the logically distributed shared memory is implemented as physically distributed memory modules

I MAY HAVE BAD MEMORY
BUT AT LEAST I DON'T HAVE BAD MEMORY
quickmeme.com

The Problem of Cache Coherence

- Shared-Memory Architectures cache both private data (used by a single processor) and shared data (used by multiple processors to provide communication).
- When shared data are cached, the shared value may be replicated in multiple caches.
- In addition to the reduction in access latency and required memory bandwidth, this replication provides a reduction of shared data contention read by multiple processors simultaneously.
- Private processor caches create a problem
  - Copies of a variable can be present in multiple caches
  - A write by one processor may not become visible to others
- The use of multiple copies of same data introduces a new problem: cache coherence.

Example: 2 CPUs with Write-Through Caches

Time	Event	Cache content CPU A	Cache content CPU B	Memory Content for location X
0	-	-	-	1

Example: 2 CPUs with Write-Through Caches

Time	Event	Cache content CPU A	Cache content CPU B	Memory Content for location X
0	-	-	-	1
1	CPU A reads X	1	-	1

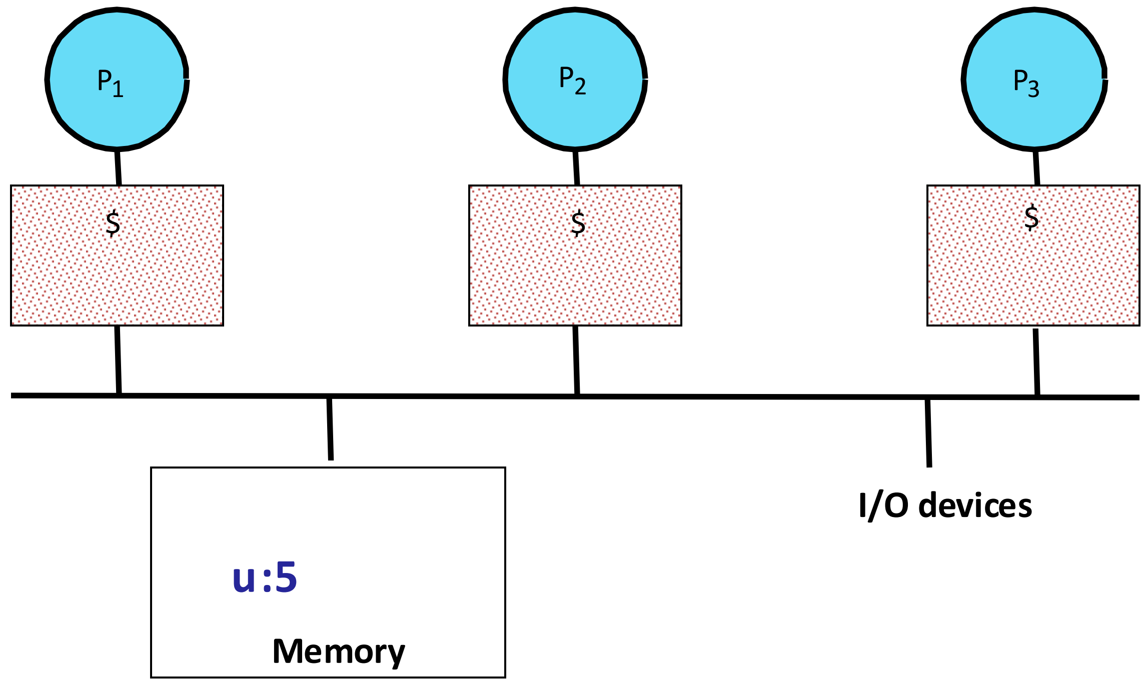
Example: 2 CPUs with Write-Through Caches

Time	Event	Cache content CPU A	Cache content CPU B	Memory Content for location X
0	-	-	-	1
1	CPU A reads X	1	-	1
2	CPU B reads X	1	1	1

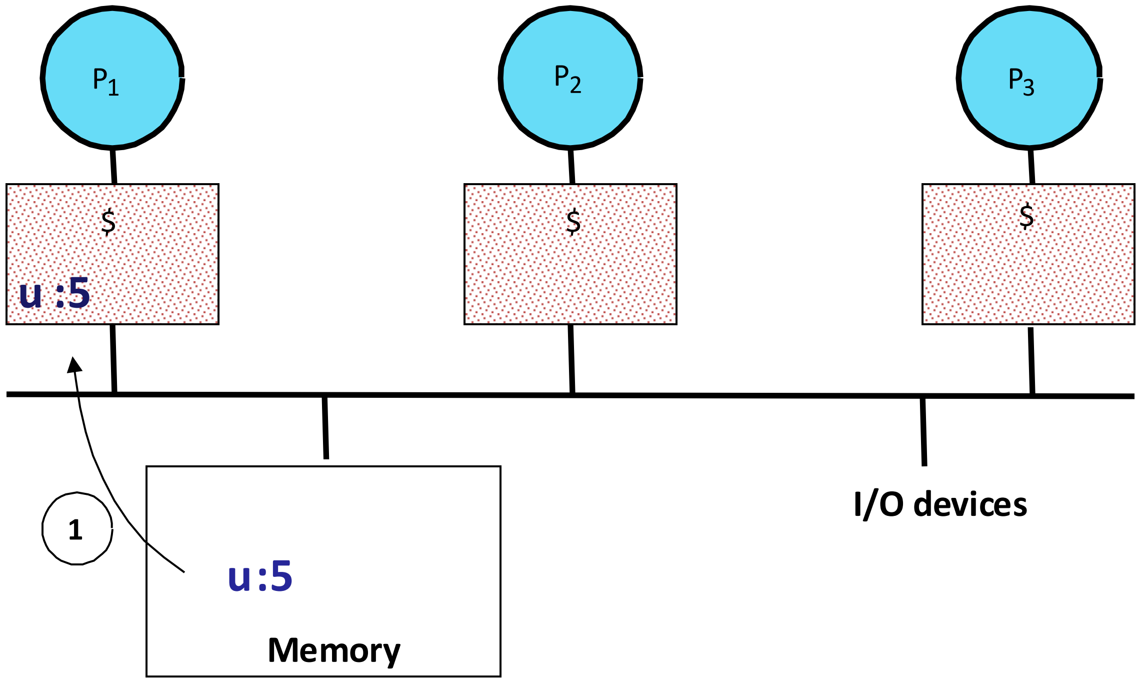
Example: 2 CPUs with Write-Through Caches

Time	Event	Cache content CPU A	Cache content CPU B	Memory Content for location X
0	-	-	-	1
1	CPU A reads X	1	-	1
2	CPU B reads X	1	1	1
3	CPU A stores 0 into X	0	1	0

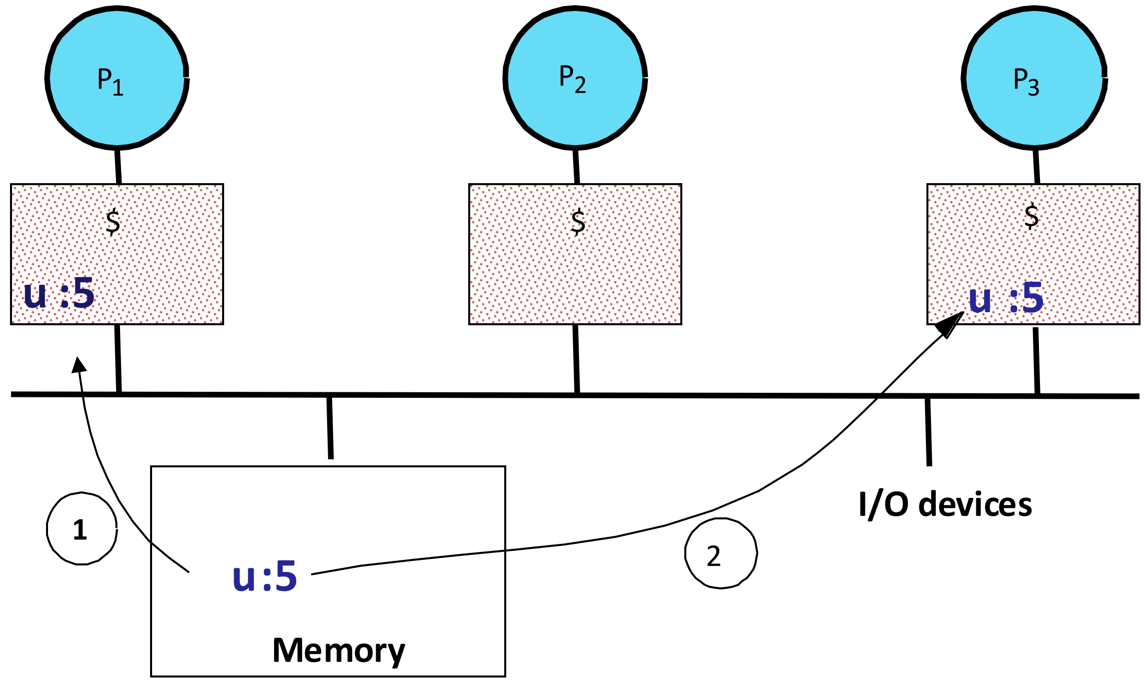
Example Cache Coherence Problem



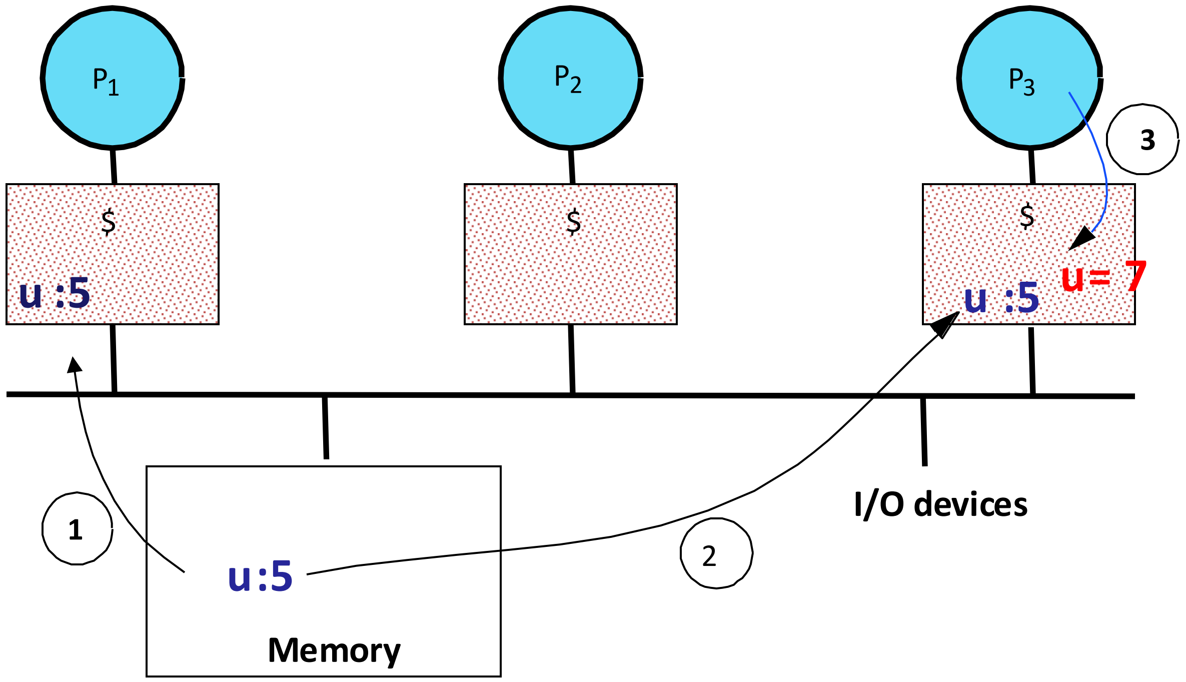
Example Cache Coherence Problem



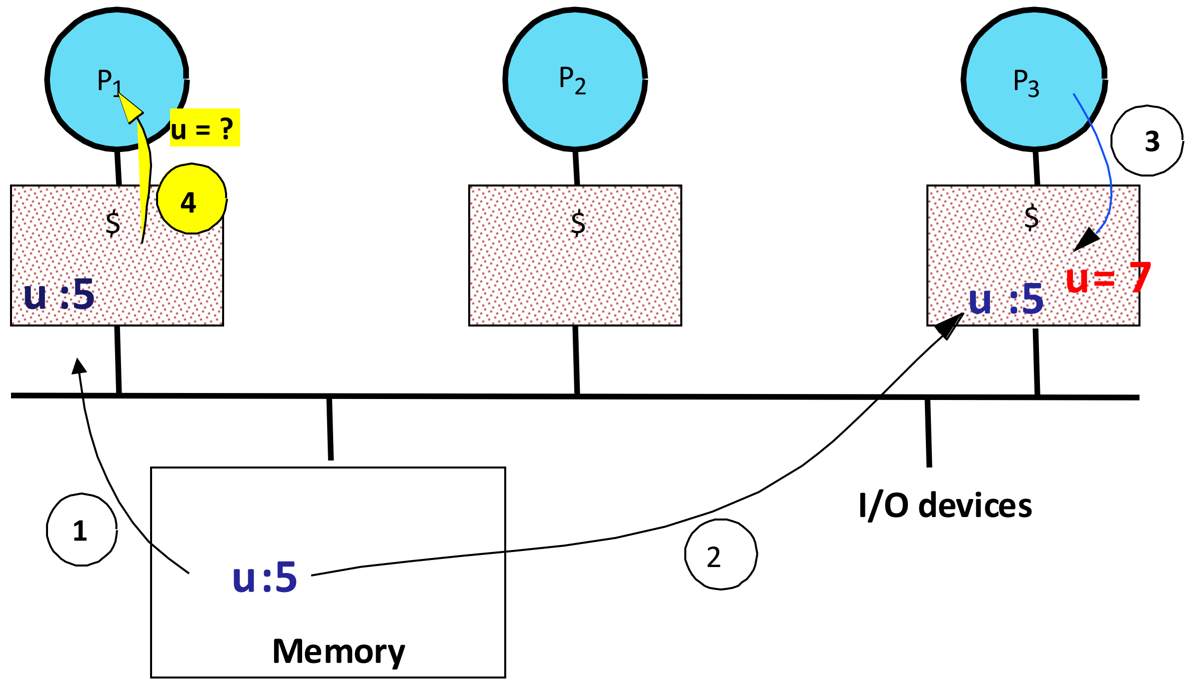
Example Cache Coherence Problem



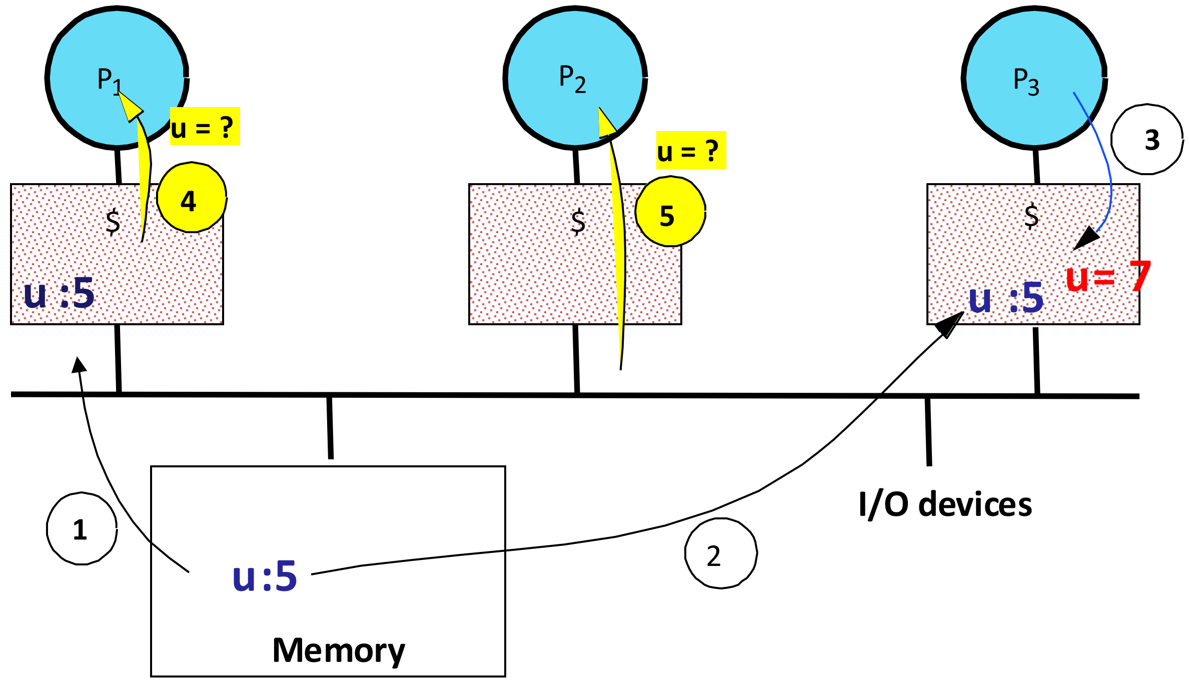
Example Cache Coherence Problem



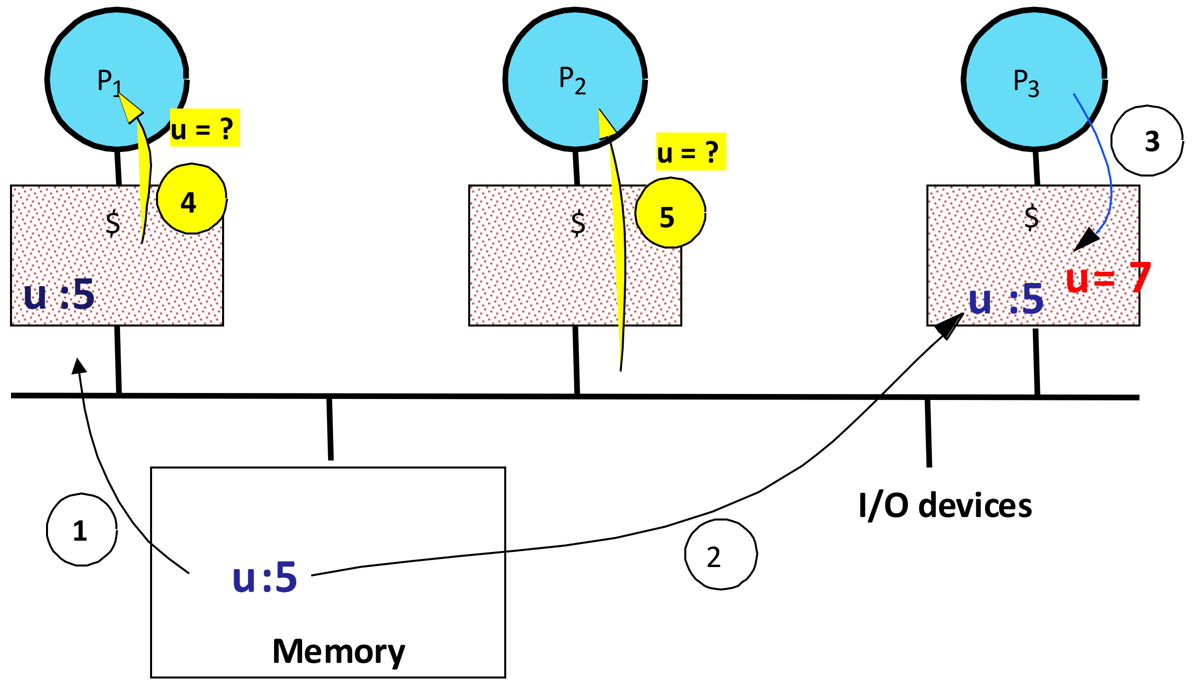
Example Cache Coherence Problem



Example Cache Coherence Problem



Example Cache Coherence Problem



- Things to note:
  - Processors see different values for u after event 3
  - With write back caches, value written back to memory depends on happenstance of which cache flushes or writes back value
    - Processes accessing main memory may see very stale value
  - Unacceptable to programs, and frequent!

What Does Coherency Mean?

- Informally:
  - "Any read must return the most recent write"
  - Too strict and too difficult to implement
- Better:
  - "Any write must eventually be seen by a read"
  - All writes are seen in proper order ("serialization")
- Two rules to ensure this:
  1. If P writes x and P1 reads it, P's write will be seen by P1 if the read and write are sufficiently far apart and no other writes to x occur between the two accesses
  2. Writes to a single location are serialized: two writes to the same location by any two processors are seen in the same order by all processors.
     - Latest write will be seen
     - Otherwise could see writes in illogical order (could see older value after a newer value)

Potential Solutions

- HW-based solutions to maintain coherency: Cache-Coherence Protocols
- Key issues to implement a cache coherent protocol in multiprocessors is tracking the status of any sharing of a data block.
- Two classes of protocols:
  - Snooping Protocols
  - Directory-Based Protocols

Potential Solutions

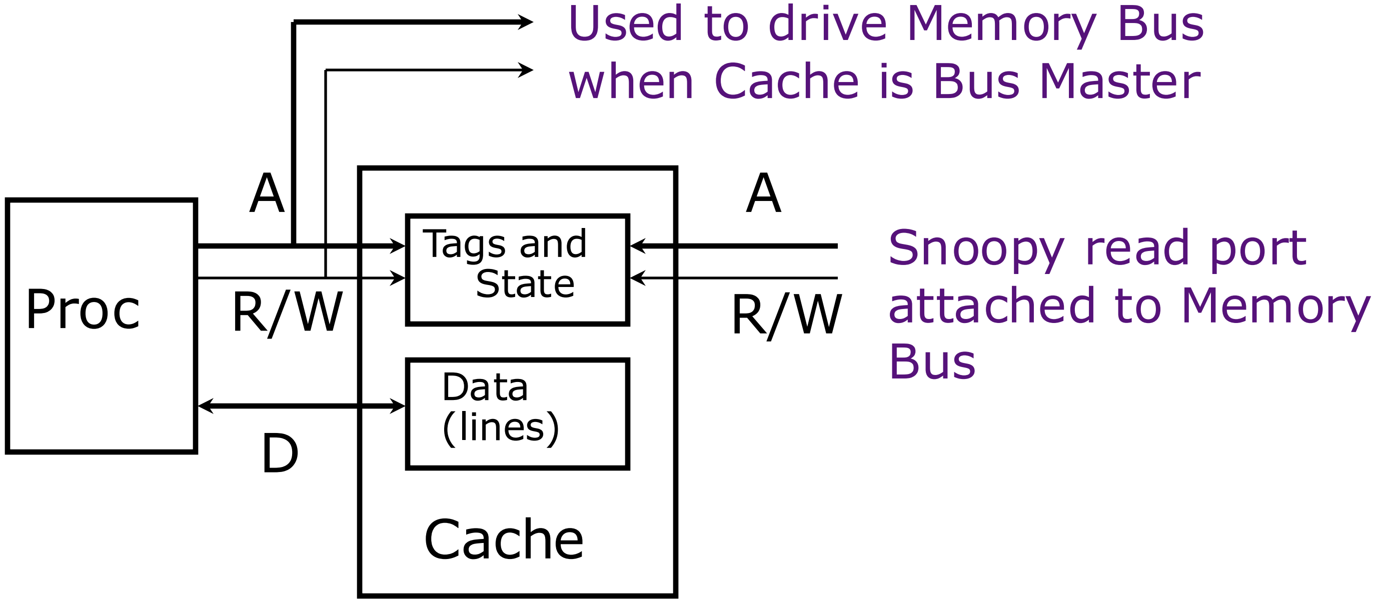
- HW-based solutions to maintain coherency: Cache-Coherence Protocols
- Key issues to implement a cache coherent protocol in multiprocessors is tracking the status of any sharing of a data block.
- Two classes of protocols:
  - Snooping Protocols
  - Directory-Based Protocols

Snooping Protocols (Snoopy Bus)

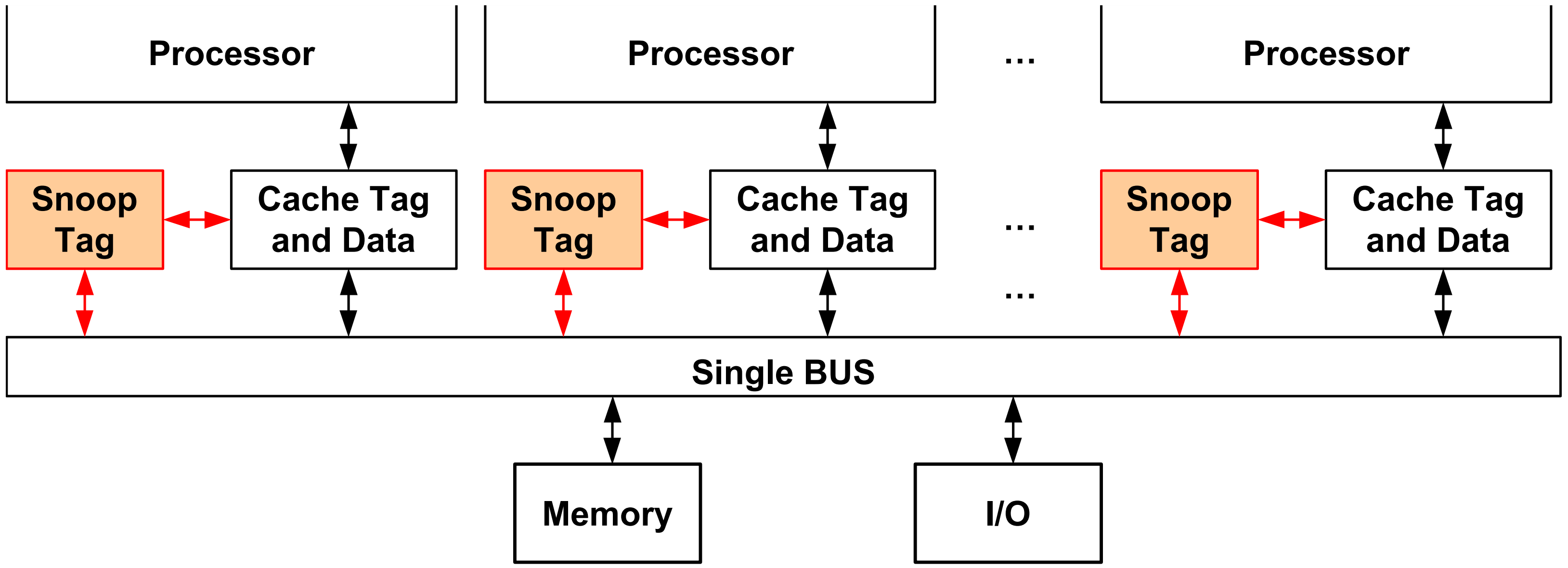
- All cache controllers monitor (snoop) on the bus to determine whether or not they have a copy of the block requested on the bus and respond accordingly.
- Every cache that has a copy of the shared block, also has a copy of the sharing status of the block, and no centralized state is kept.
- Send all requests for shared data to all processors.
- Require broadcast, since caching info is at processors.
- Suitable for Centralized Shared-Memory Architectures, and in particular for small scale multiprocessors with single shared bus.

Snoopy Cache Goodman 1983

- Idea: Have cache watch (or snoop upon) DMA transfers, and then "do the right thing"
- Snoopy cache tags are dual-ported


Used to drive Memory Bus when Cache is Bus Master
A
R/W
Proc
D
Tags and State
Data (lines)
Cache
A
R/W
Snoopy read port attached to Memory Bus

Snoop Tag


Processor
Processor
…
Processor
Snoop Tag
Cache Tag and Data
Snoop Tag
Cache Tag and Data
…
Snoop Tag
Cache Tag and Data
…
Single BUS
Memory
I/O

Basic Snooping Protocols

- Snooping Protocols are of two types depending on what happens on a write operation:
  - Write-Invalidate Protocol
  - Write-Update or Write-Broadcast Protocol

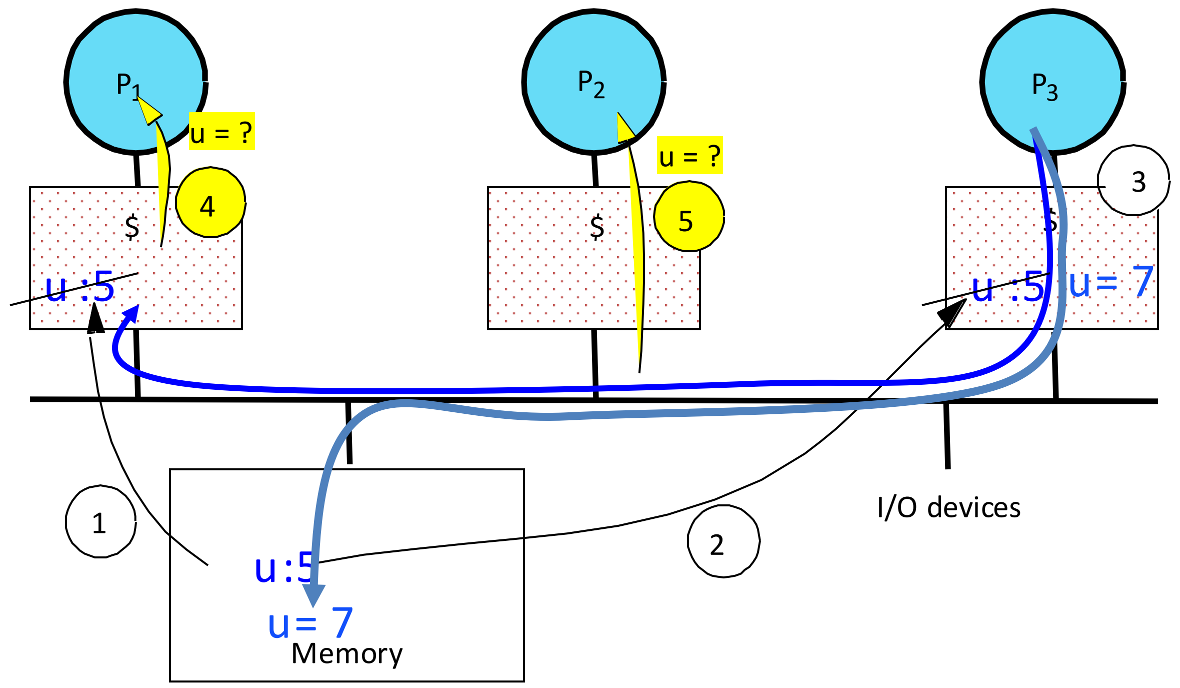
Write-Invalidate Protocol

- The writing processor issues an invalidation signal over the bus to cause all copies in other caches to be invalidated before changing its local copy.
- The writing processor is then free to update the local data until another processor asks for it.
- All caches on the bus check to see if they have a copy of the data and, if so, they must invalidate the block containing the data.
- This scheme allows multiple readers but only a single writer.

Write-Invalidate Protocol

- This scheme uses the bus only on the first write to invalidate the other copies.
- Subsequent writes do not result in bus activity.
- This protocol provides similar benefits to write-back protocols in terms of reducing demands on bus bandwidth.
- Read Miss:
  - Write-Through: Memory always up-to-date
  - Write-Back: Snoop in caches to find the most recent copy.

Example: Write-through Invalidate



Write-Update Protocol

- The writing processor broadcasts the new data over the bus; all caches check if they have a copy of the data and, if so, all copies are updated with the new value.
- This scheme requires the continuous broadcast of writes to shared data (while write-invalidate deletes all other copies so that there is only one local copy for subsequent writes)
- This protocol is like write-through because all writes go over the bus to update copies of the shared data.
- This protocol has the advantage of making the new values appear in caches sooner $\Rightarrow$ reduced latency
- Read Miss: Memory always up-to-date.

Invalidate vs. Update

- Basic question of program behavior:
  - Is a block written by one processor later read by others before it is overwritten?
- Invalidate
  - yes: readers will take a miss
  - no: multiple writes without addition traffic
    - also clears out copies that will never be used again
- Update
  - yes: avoids misses on later references
  - no: multiple useless updates

➢ Need to look at program reference patterns and hardware complexity

➢ Can we tune this automatically????

but first - correctness

Snooping Protocols

- Most part of commercial cache-based multiprocessors uses:
  - Write-Back Caches to reduce bus traffic ⇒ they allow more processors on a single bus.
  - Write-Invalidate Protocol to preserve bus bandwidth
- Write serialization due to bus serializing request: bus is single point of arbitration
  - A write to a shared data item cannot actually complete until it obtains bus access

Write back cache

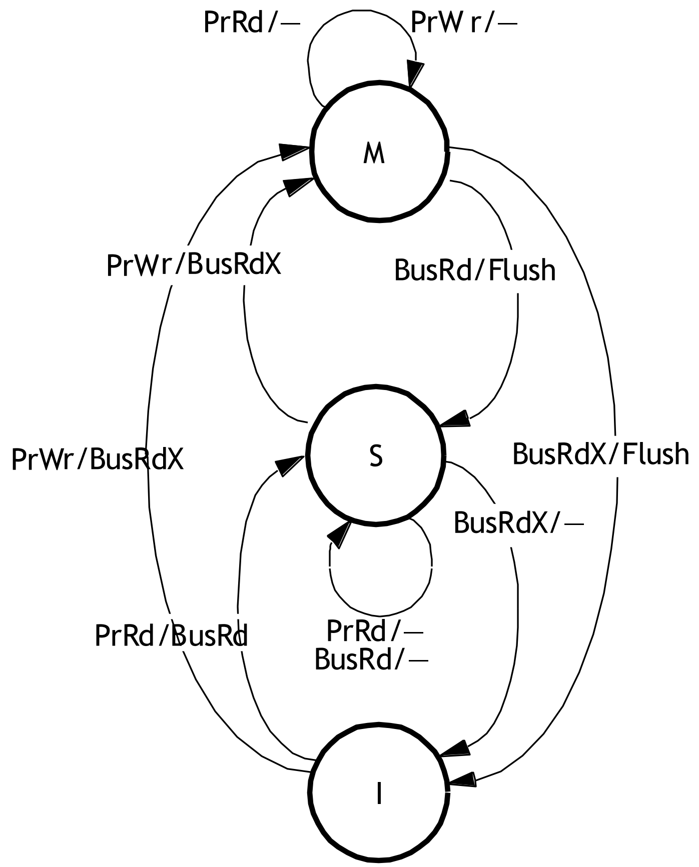
- How to identify the most recent data value of a cache block in case of cache miss?
  - It can be in a cache rather in a memory
- Can use the same snooping scheme both for cache misses and writes
  - Each processor snoops every address placed on the bus
  - If a processor finds that it has a dirty copy of the requested cache block, it provides the cache block in response to the read request
  - memory access is aborted

Snooping Protocols: An Example

- Write-Invalidate Protocol, Write-Back Cache
- Each block of memory is in one of three states:
  - Clean in all caches and up-to-date in memory (Shared)
  - OR Dirty in exactly one cache (Exclusive)
  - OR Not in any caches
- Each cache block can be in one of three states:
  - Clean (or Shared) (read only): the block is clean (not modified) and can be read
  - Dirty (or Modified or Exclusive) : cache has only copy, its writeable, and dirty (block cannot be shared)
  - Invalid : block contains no valid data

MSI Invalidate Protocol

- Three States:
  - "M": "Modified"
  - "S": "Shared"
  - "I": "Invalid"
- Read obtains block in "shared"
  - even if only cache copy
- Obtain exclusive ownership before writing
  - BusRdx causes others to invalidate (demote)
  - If M in another cache, will flush
  - BusRdx even if hit in S
    - promote to M (upgrade)
- What about replacement?
  - S->I, M->I



Snoopy Coherence Protocols

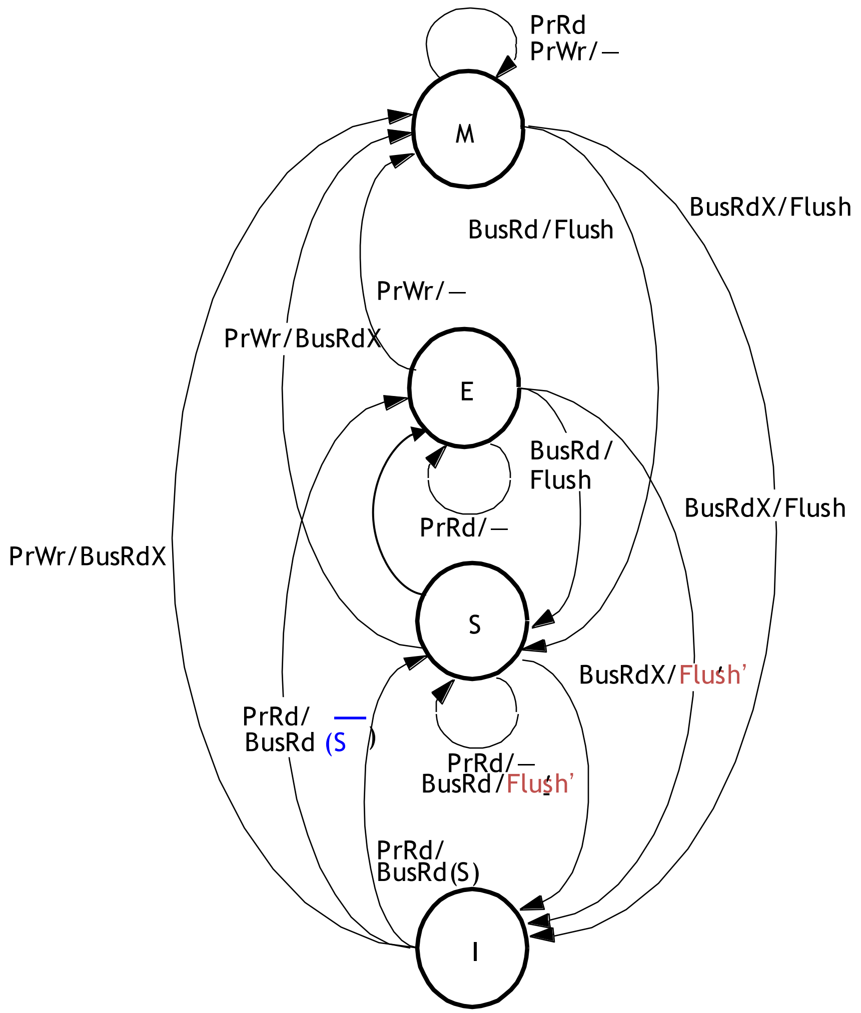
- Complications for the basic MSI protocol:
  - Operations are not atomic
    - E.g. detect miss, acquire bus, receive a response
    - Creates possibility of deadlock and races
    - One solution: processor that sends invalidate can hold bus until other processors receive the invalidate
- Extensions:
  - Add exclusive state to indicate clean block in only one cache (MESI protocol)
    - Prevents needing to write invalidate on a write
  - Owned state

Snooping Cache Variations

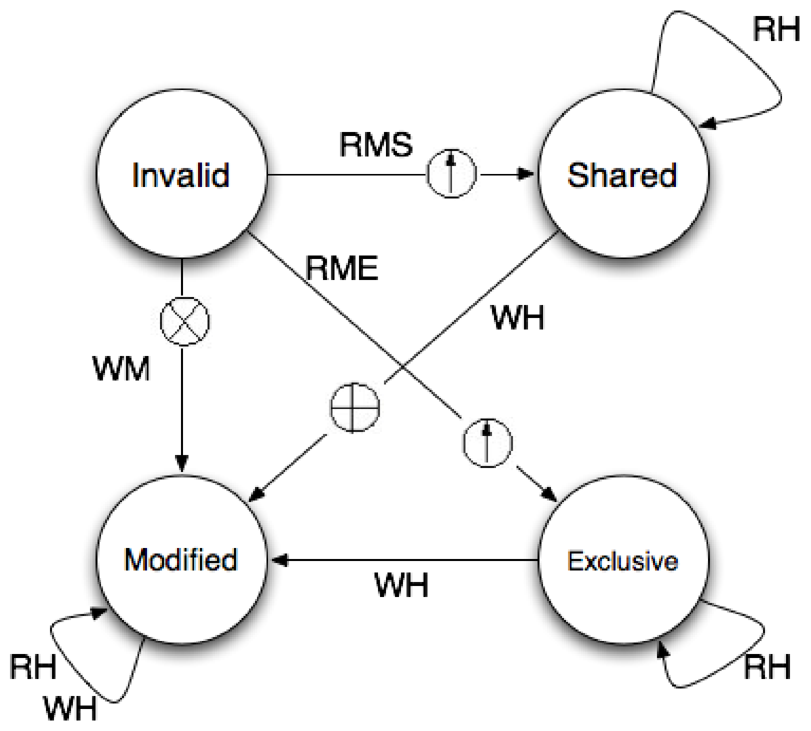
- MESI Protocol: Write-Invalidate
- Each cache block can be in one of four states:
  - Modified : the block is dirty and cannot be shared; cache has only copy, its writeable.
  - Exclusive : the block is clean and cache has only copy;
  - Shared: the block is clean and other copies of the block are in cache;
  - Invalid : block contains no valid data
- Add exclusive state to distinguish exclusive (writable) and owned (written)

MESI State Transition Diagram

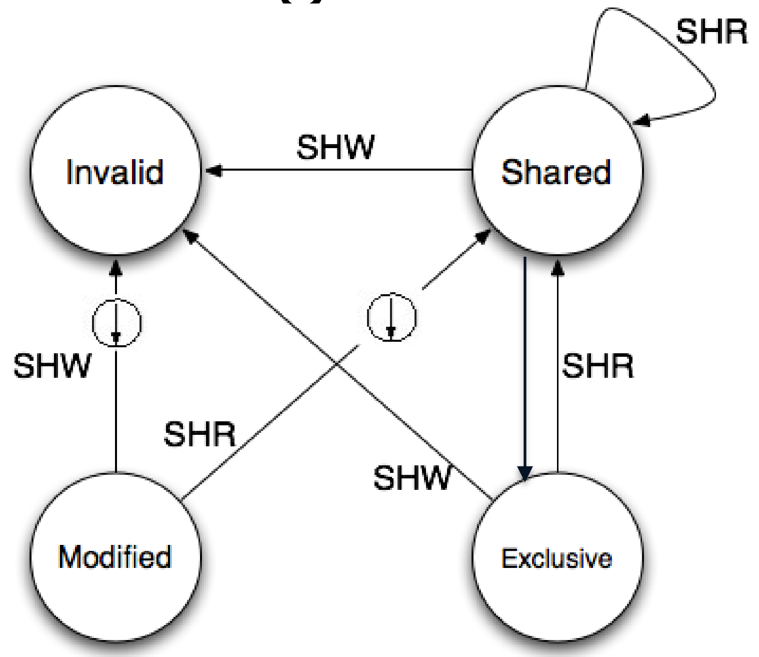
- BusRd(S) means shared line asserted on BusRd transaction
- Flush’ : if cache-to-cache xfers
  - only one cache flushes data
- Replacement:
  - S→I can happen without telling other caches
  - E→I, M→I



MESI State Transition Diagram V2



Cache line in the "acting" processor



Transaction due to events snooped on the common BUS

BUS Transactions

RH =	Read Hit	↓ =	Snoop Push
RMS =	Read Miss, Shared	⊗ =	Invalidate Transaction
RME =	Read Miss, Exclusive	⊕ =	Read-with-Intent-to-Modify
WH =	Write Hit	↑ =	Cache Block Fill
WM =	Write Miss		
SHR =	Snoop Hit on a Read		
SHW =	Snoop Hit on a Write or Read-with-Intent-to-Modify		

MESI Protocol

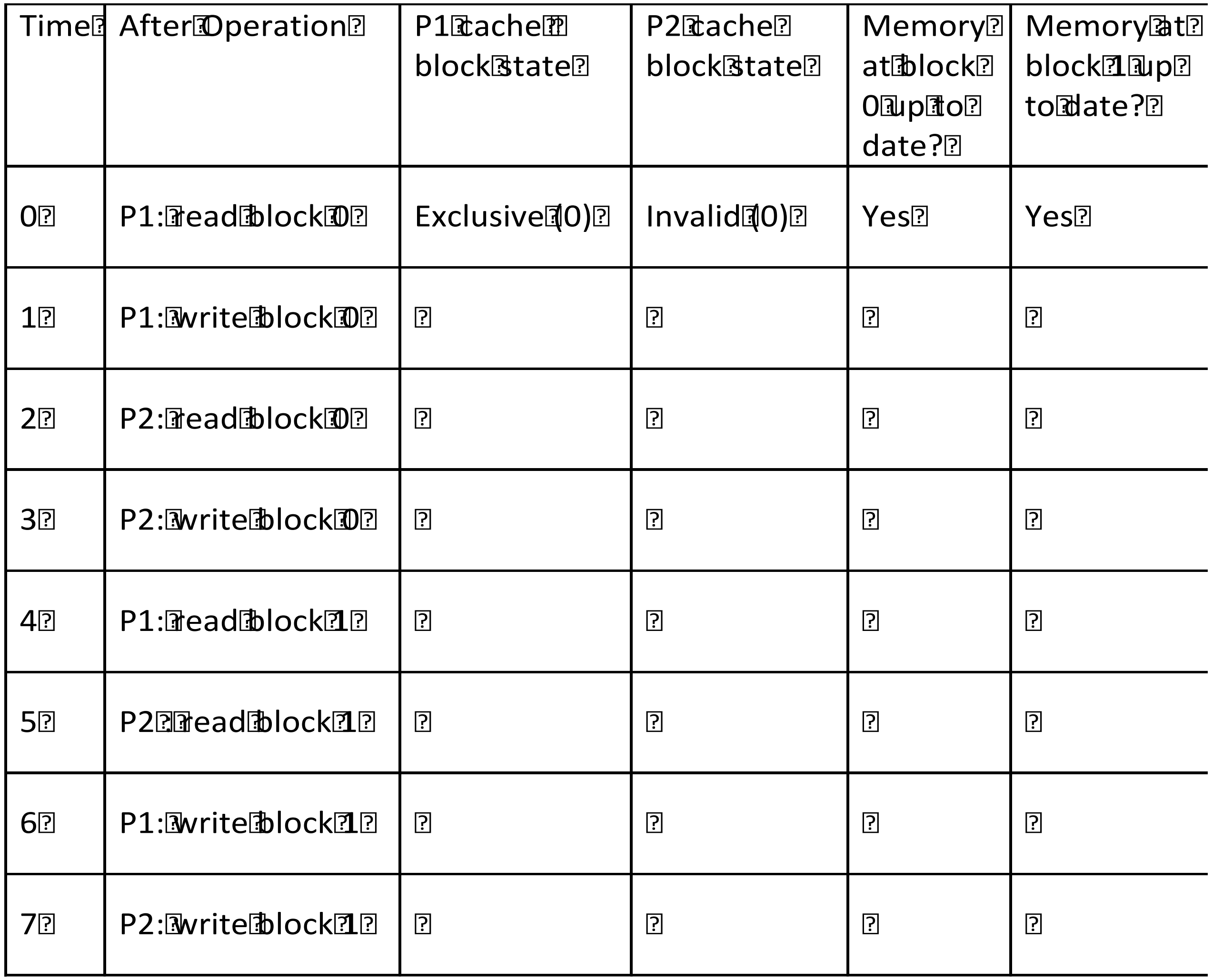
- In both S and E, the memory has an up-to-date version of the data
- A write to a E block does not require to send the invalidation signal on the bus, since no other copies of the block are in cache.
- A write to a S block implies the invalidation of the other copies of the block in cache.

MESI Working Example

- Consider the following access pattern on a two-processor system with a direct-mapped, write-back cache with one cache block and a two cache block memory.

- Assume the MESI protocol is used, with write-back caches, write-allocate, and invalidation of other caches on write (instead of updating the value in the other caches).

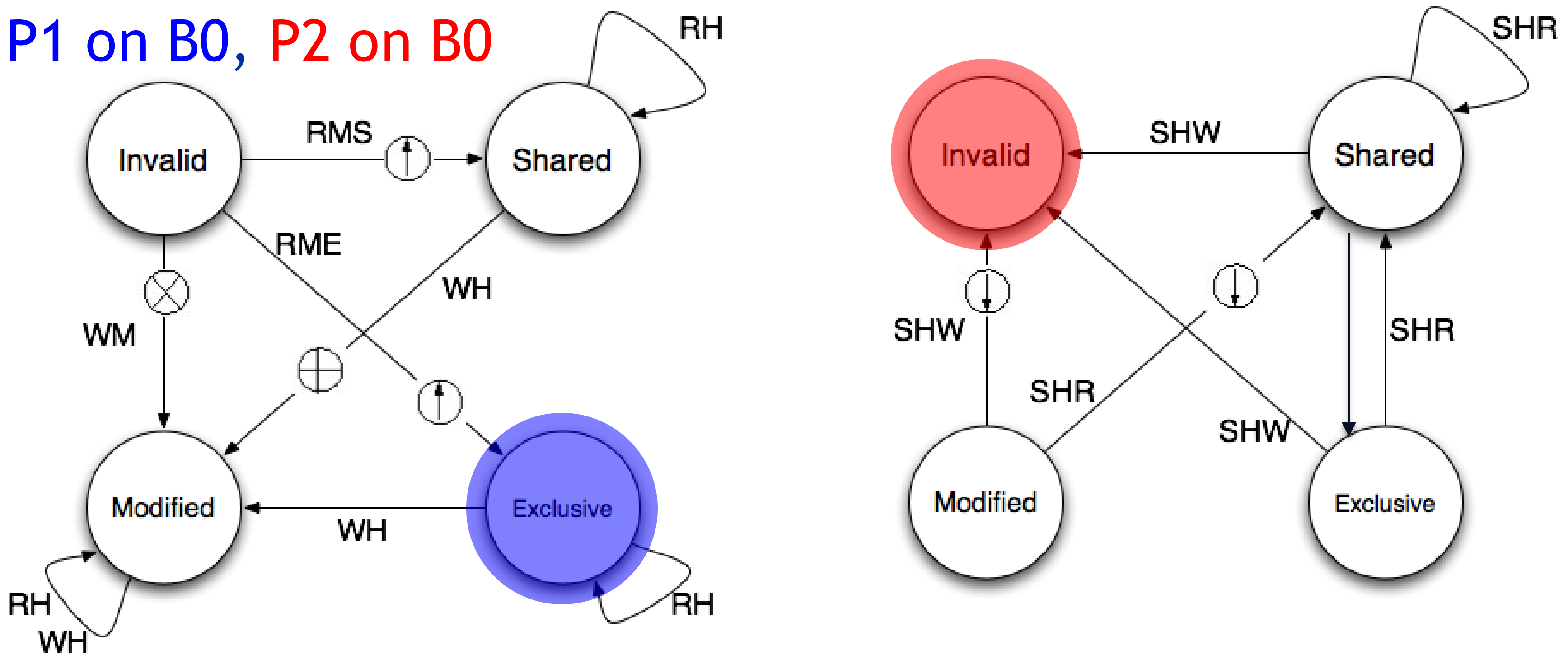
Example: Access Pattern



Time	After Operation	P1 cache block state	P2 cache block state	Memory at block 0 up to date?	Memory at block 1 up to date?
0	P1: read block 0	Exclusive (0)	Invalid (0)	Yes	Yes
1	P1: write block 0				
2	P2: read block 0				
3	P2: write block 0				
4	P1: read block 1				
5	P2: read block 1				
6	P1: write block 1				
7	P2: write block 1				

Time 0: P1 Read Block 0

P1 on B0, P2 on B0



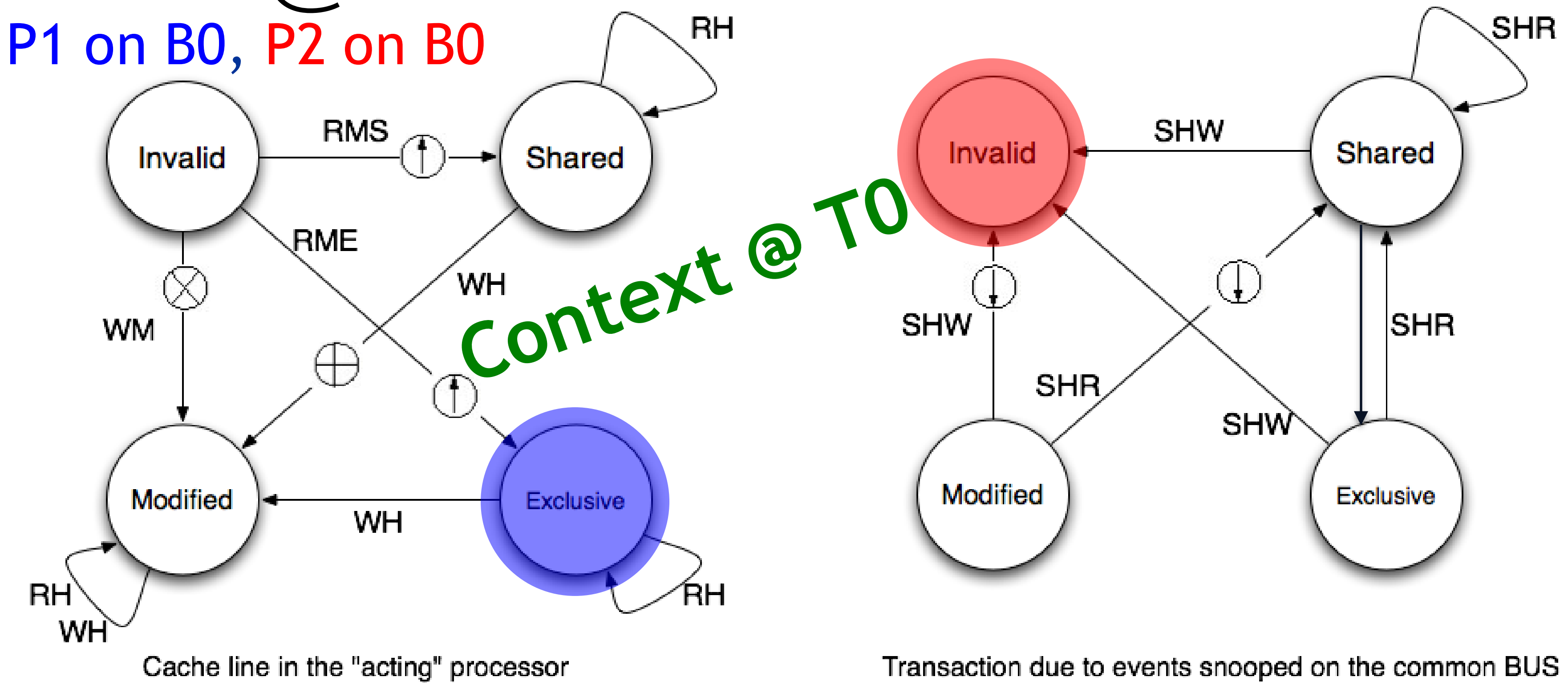
Cache line in the "acting" processor

Transaction due to events snooped on the common BUS

BUS Transactions

RH =	Read Hit	↓ =	Snoop Push
RMS =	Read Miss, Shared	⊗ =	Invalidate Transaction
RME =	Read Miss, Exclusive	⊕ =	Read-with-Intent-to-Modify
WH =	Write Hit	↑ =	Cache Block Fill
WM =	Write Miss		
SHR =	Snoop Hit on a Read		
SHW =	Snoop Hit on a Write or Read-with-Intent-to-Modify		

@T1: P1 Write Block 0


P1 on B0, P2 on B0
Context @ T0
Invalid
Shared
Modified
Exclusive
RMS
RME
WM
WH
RH
SHW
SHR
Cache line in the "acting" processor
Transaction due to events snooped on the common BUS

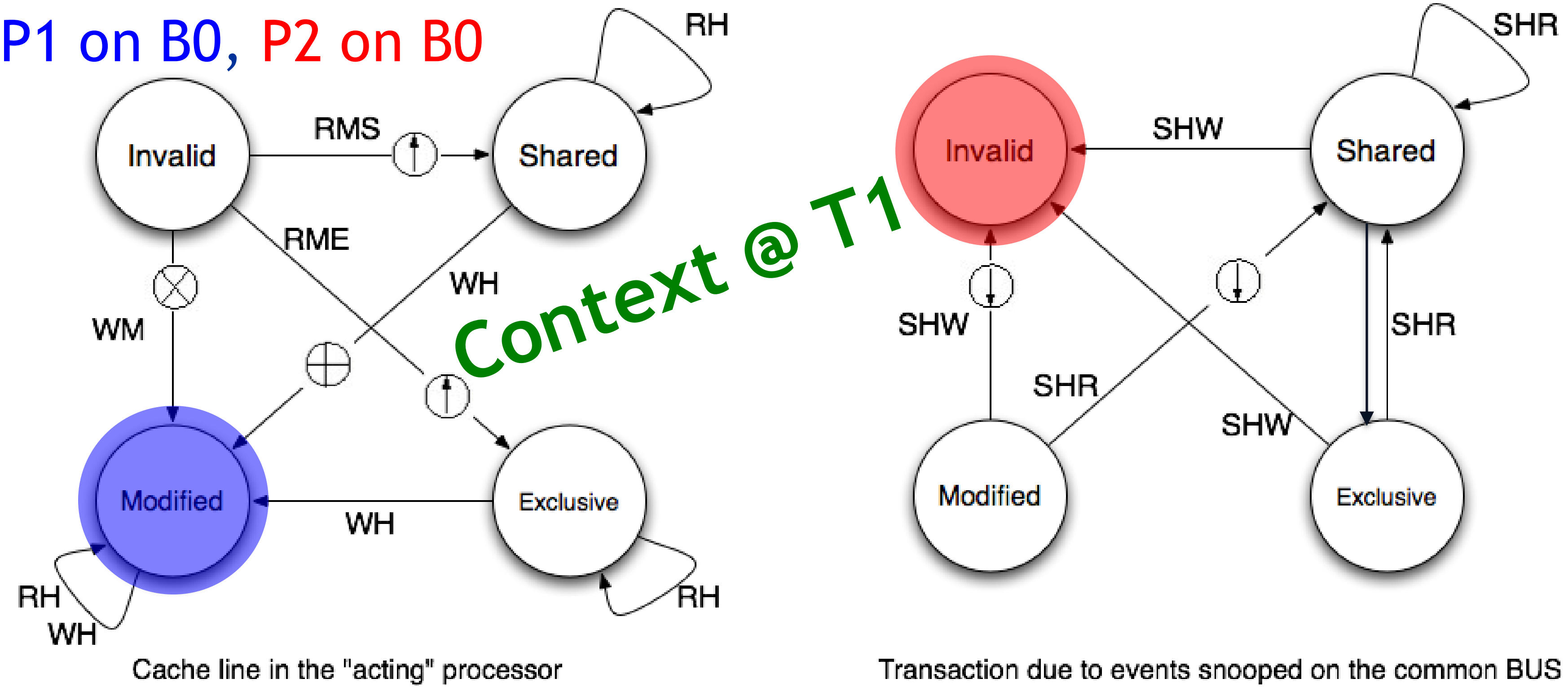
BUS Transactions

RH =	Read Hit
RMS =	Read Miss, Shared
RME =	Read Miss, Exclusive
WH =	Write Hit
WM =	Write Miss
SHR =	Snoop Hit on a Read
SHW =	Snoop Hit on a Write or Read-with-Intent-to-Modify

⏀↓ =	Snoop Push
⊗ =	Invalidate Transaction
⊕ =	Read-with-Intent-to-Modify
⏀↑ =	Cache Block Fill

@T1: P1 Write Block 0

P1 on B0, P2 on B0


Context @ T1
Invalid
Shared
Modified
Exclusive
RMS
RME
WH
WM
RH
Cache line in the "acting" processor
SHW
SHR
Transaction due to events snooped on the common BUS

BUS Transactions

RH =	Read Hit
RMS =	Read Miss, Shared
RME =	Read Miss, Exclusive
WH =	Write Hit
WM =	Write Miss
SHR =	Snoop Hit on a Read
SHW =	Snoop Hit on a Write or Read-with-Intent-to-Modify

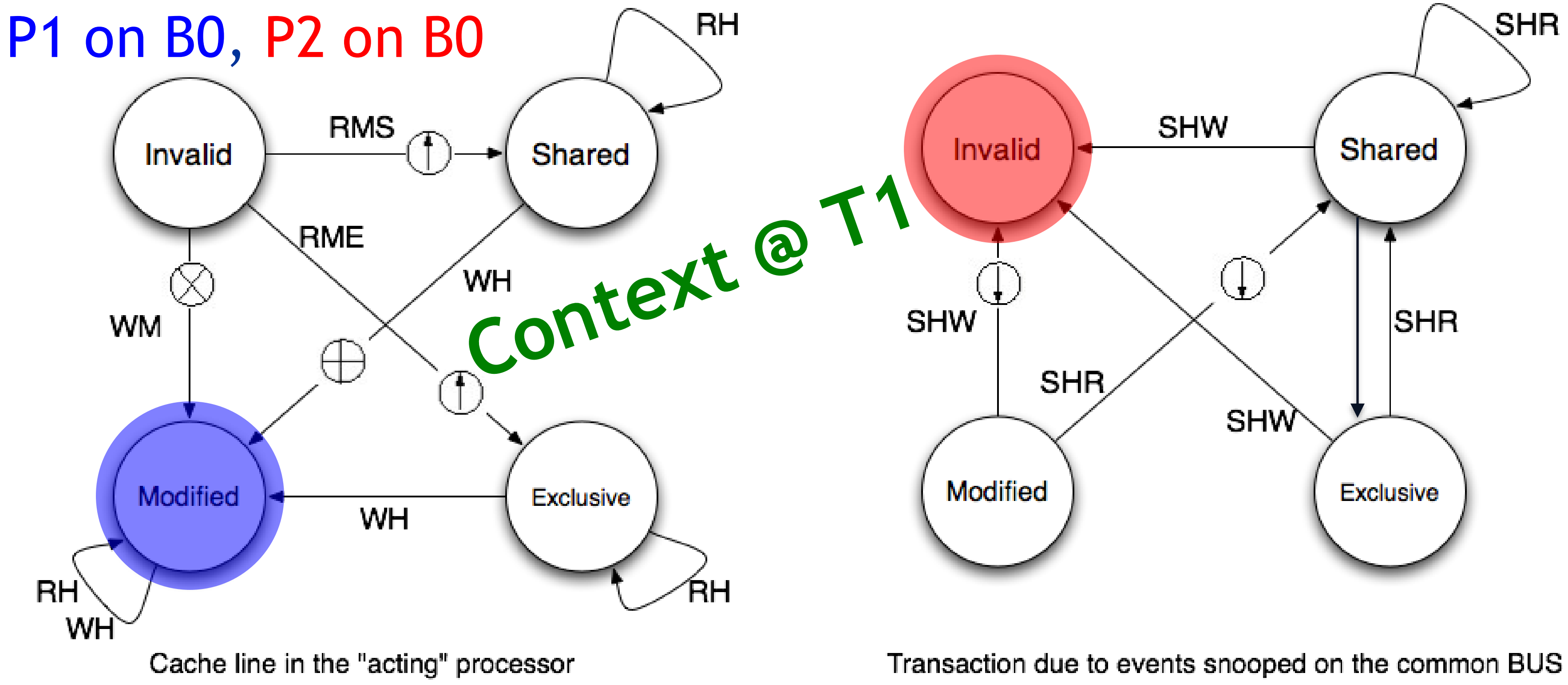
⊥ =	Snoop Push
⊗ =	Invalidate Transaction
⊕ =	Read-with-Intent-to-Modify
⊤ =	Cache Block Fill

Time: 1

Time	After Operation	P1 cache block state	P2 cache block state	Memory at block 0 up to date?	Memory at block 1 up to date?
0	P1: read block 0	Exclusive (0)	Invalid (0)	Yes	Yes
1	P1: write block 0	Modified (0)	Invalid (0)	No	Yes
2	P2: read block 0				
3	P2: write block 0				
4	P1: read block 1				
5	P2: read block 1				
6	P1: write block 1				
7	P2: write block 1				

@T2: P2 Read Block 0

P1 on B0, P2 on B0



Cache line in the "acting" processor

Transaction due to events snooped on the common BUS

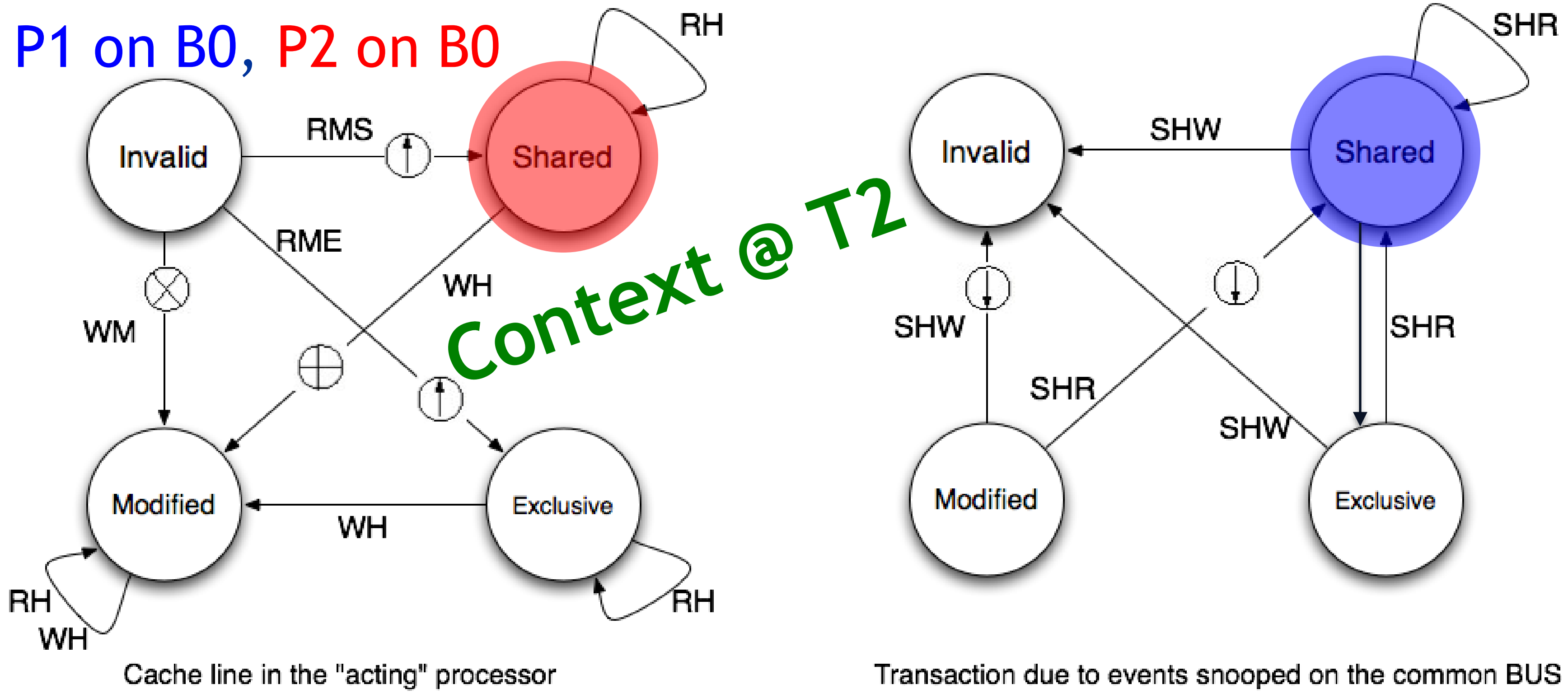
BUS Transactions

RH =	Read Hit
RMS =	Read Miss, Shared
RME =	Read Miss, Exclusive
WH =	Write Hit
WM =	Write Miss
SHR =	Snoop Hit on a Read
SHW =	Snoop Hit on a Write or Read-with-Intent-to-Modify

⦶ =	Snoop Push
⊗ =	Invalidate Transaction
⊕ =	Read-with-Intent-to-Modify
⦶ =	Cache Block Fill

@T2: P2 Read Block 0

P1 on B0, P2 on B0



BUS Transactions

RH =	Read Hit
RMS =	Read Miss, Shared
RME =	Read Miss, Exclusive
WH =	Write Hit
WM =	Write Miss
SHR =	Snoop Hit on a Read
SHW =	Snoop Hit on a Write or Read-with-Intent-to-Modify

- ⏀ = Snoop Push
- ⊗ = Invalidate Transaction
- ⊕ = Read-with-Intent-to-Modify
- ⏀ = Cache Block Fill

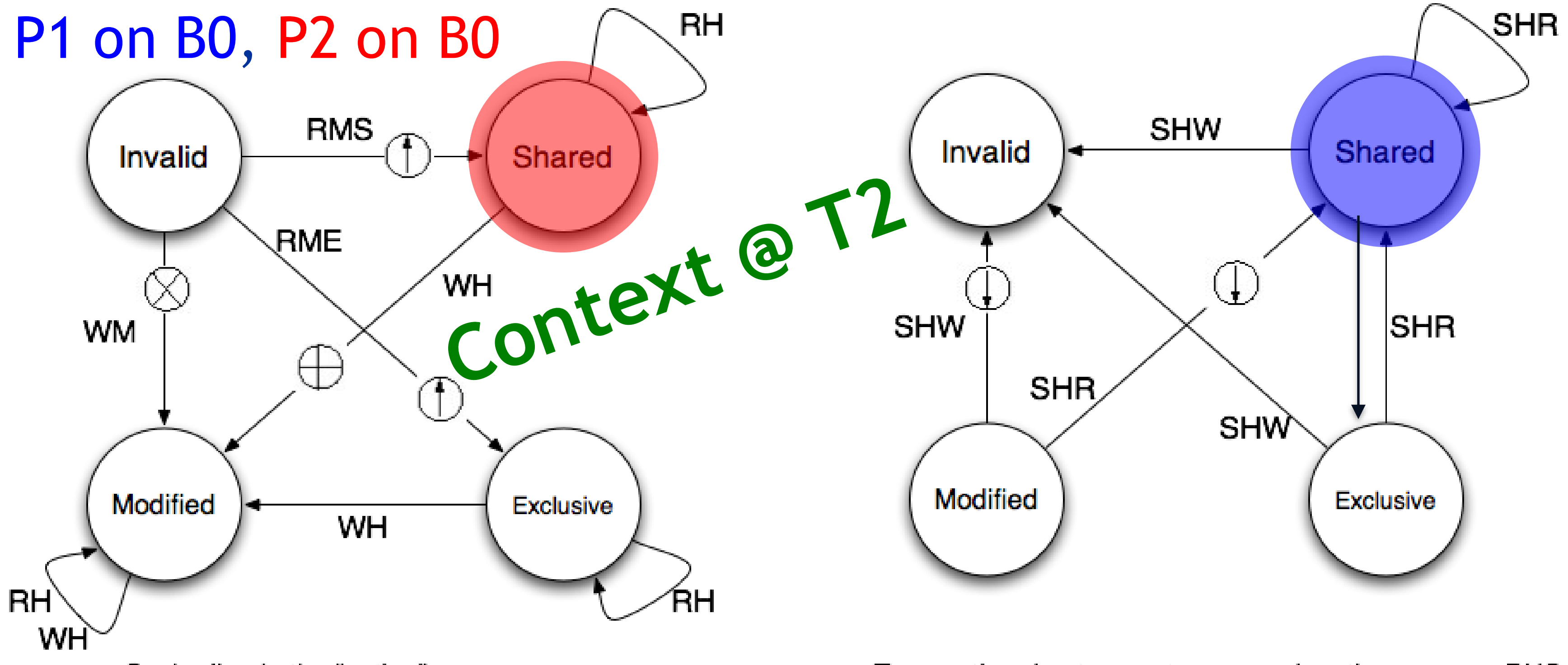
Time 2

Time	After Operation	P1 cache block state	P2 cache block state	Memory at block 0 up to date?	Memory at block 1 up to date?
0	P1: read block 0	Exclusive (0)	Invalid (0)	Yes	Yes
1	P1: write block 0	Modified (0)	Invalid (0)	No	Yes
2	P2: read block 0	Shared (0)	Shared (0)	Yes	Yes
3	P2: write block 0				
4	P1: read block 1				
5	P2: read block 1				
6	P1: write block 1				
7	P2: write block 1				

@T3: P2 Write Block 0

P1 on B0, P2 on B0

Context @ T2



Cache line in the "acting" processor

Transaction due to events snooped on the common BUS

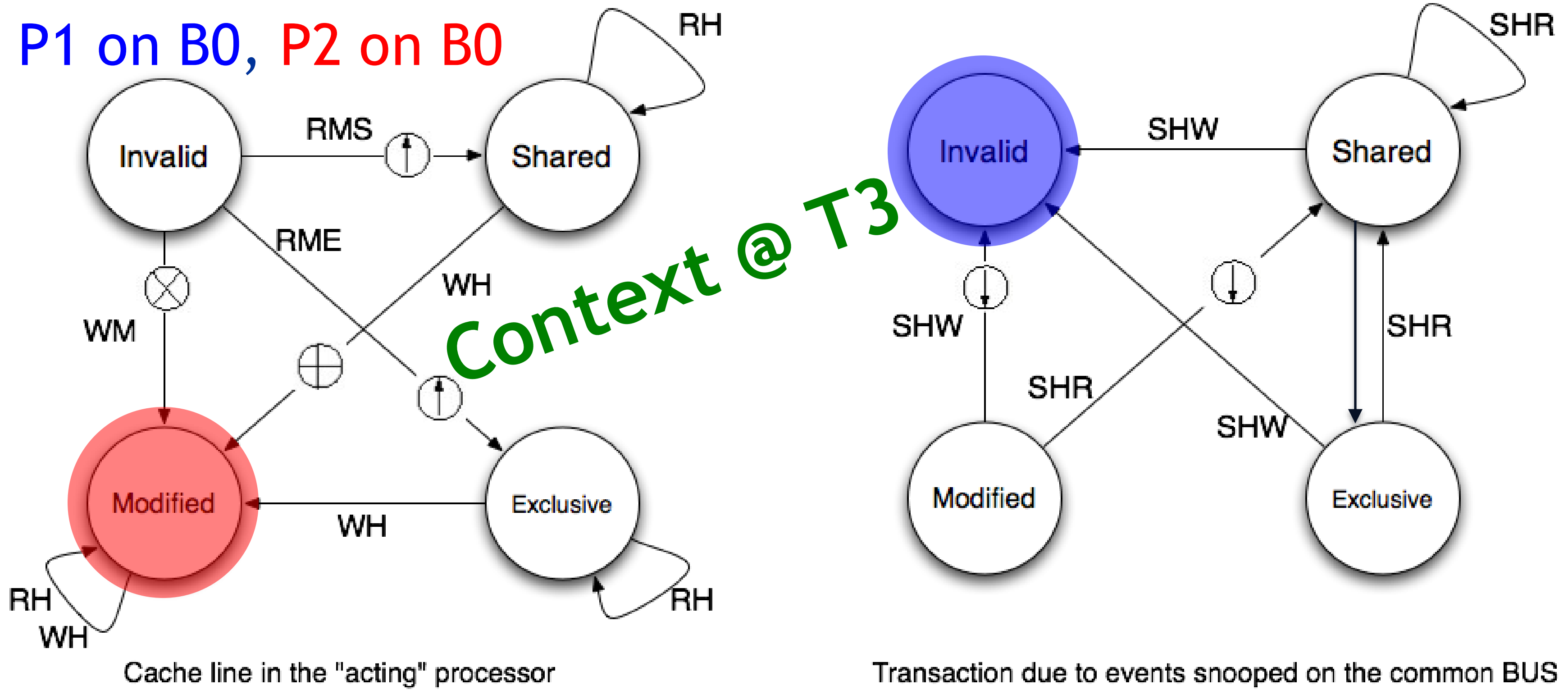
BUS Transactions

RH =	Read Hit
RMS =	Read Miss, Shared
RME =	Read Miss, Exclusive
WH =	Write Hit
WM =	Write Miss
SHR =	Snoop Hit on a Read
SHW =	Snoop Hit on a Write or Read-with-Intent-to-Modify

ⓘ	= Snoop Push
⊗	= Invalidate Transaction
⊕	= Read-with-Intent-to-Modify
ⓘ	= Cache Block Fill

@T3: P2 Write Block 0

P1 on B0, P2 on B0



Cache line in the "acting" processor

Transaction due to events snooped on the common BUS

BUS Transactions

RH =	Read Hit
RMS =	Read Miss, Shared
RME =	Read Miss, Exclusive
WH =	Write Hit
WM =	Write Miss
SHR =	Snoop Hit on a Read
SHW =	Snoop Hit on a Write or Read-with-Intent-to-Modify

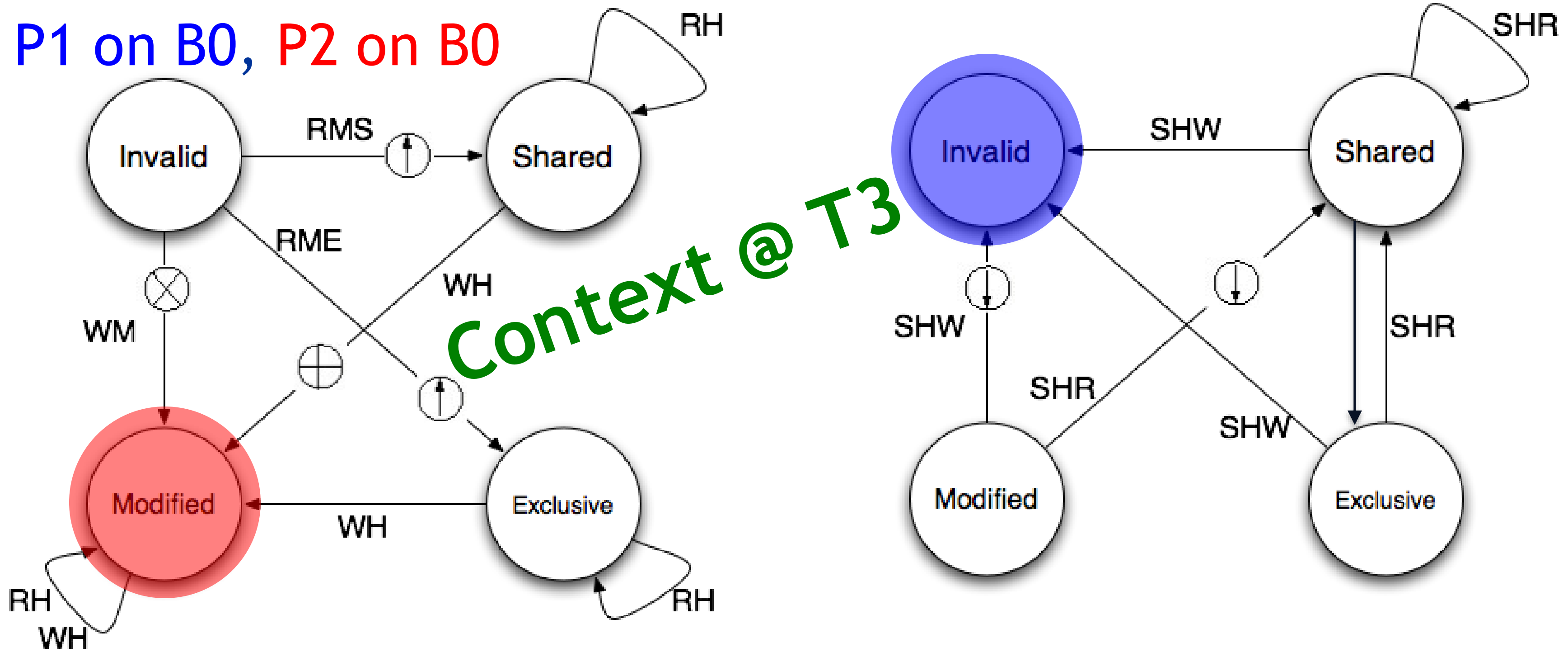
⊕↓	= Snoop Push
⊗	= Invalidate Transaction
⊕	= Read-with-Intent-to-Modify
⊕↑	= Cache Block Fill

Time 3

Time	After Operation	P1 cache block state	P2 cache block state	Memory at block 0 up to date?	Memory at block 1 up to date?
0	P1: read block 0	Exclusive (0)	Invalid (0)	Yes	Yes
1	P1: write block 0	Modified (0)	Invalid (0)	No	Yes
2	P2: read block 0	Shared (0)	Shared (0)	Yes	Yes
3	P2: write block 0	Invalid (0)	Modified (0)	No	Yes
4	P1: read block 1				
5	P2: read block 1				
6	P1: write block 1				
7	P2: write block 1				

@T4: P1 Read Block 1

P1 on B0, P2 on B0



Cache line in the "acting" processor

Transaction due to events snooped on the common BUS

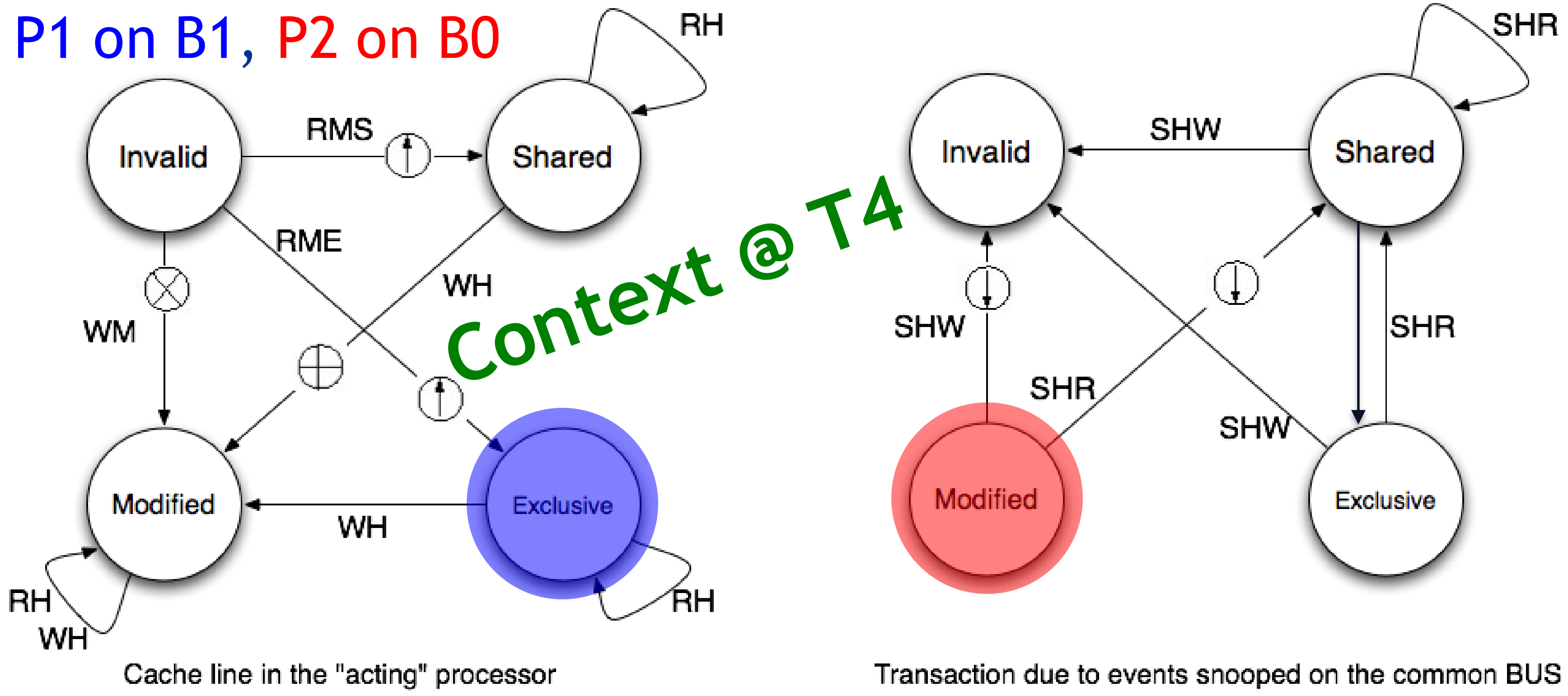
BUS Transactions

RH =	Read Hit
RMS =	Read Miss, Shared
RME =	Read Miss, Exclusive
WH =	Write Hit
WM =	Write Miss
SHR =	Snoop Hit on a Read
SHW =	Snoop Hit on a Write or Read-with-Intent-to-Modify

ⓘ↓ =	Snoop Push
⊗ =	Invalidate Transaction
⊕ =	Read-with-Intent-to-Modify
ⓘ↑ =	Cache Block Fill

@T4: P1 Read Block 1

P1 on B1, P2 on B0



BUS Transactions

RH =	Read Hit
RMS =	Read Miss, Shared
RME =	Read Miss, Exclusive
WH =	Write Hit
WM =	Write Miss
SHR =	Snoop Hit on a Read
SHW =	Snoop Hit on a Write or Read-with-Intent-to-Modify

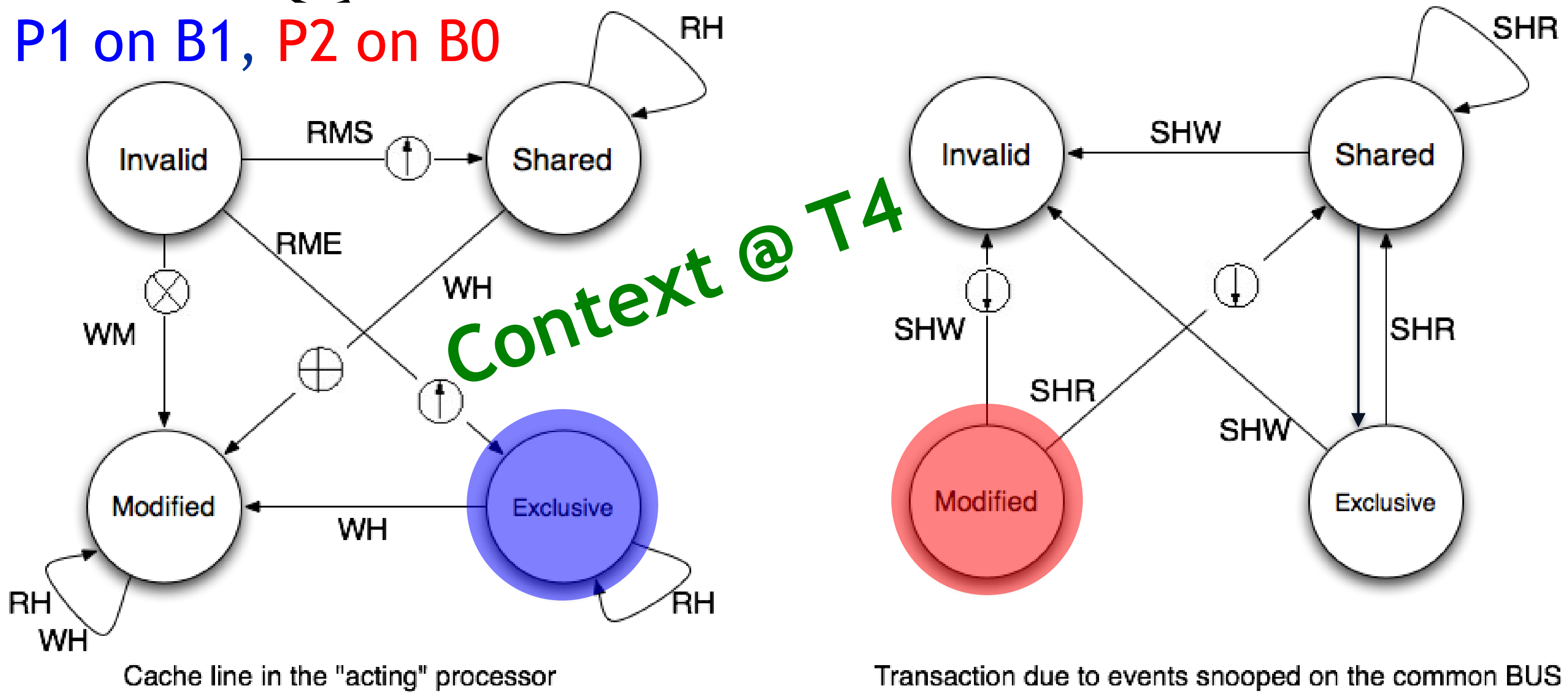
⊙↓	= Snoop Push
⊗	= Invalidate Transaction
⊕	= Read-with-Intent-to-Modify
⊙↑	= Cache Block Fill

Time 4

Time	After Operation	P1 cache block state	P2 cache block state	Memory at block 0 up to date?	Memory at block 1 up to date?
0	P1: read block 0	Exclusive (0)	Invalid (0)	Yes	Yes
1	P1: write block 0	Modified (0)	Invalid (0)	No	Yes
2	P2: read block 0	Shared (0)	Shared (0)	Yes	Yes
3	P2: write block 0	Invalid (0)	Modified (0)	No	Yes
4	P1: read block 1	Exclusive (1)	Modified (0)	No	Yes
5	P2: read block 1				
6	P1: write block 1				
7	P2: write block 1				

@T5: P2 Read Block 1

P1 on B1, P2 on B0


Context @ T4
Invalid
Shared
Modified
Exclusive
RMS
RME
RH
WM
WH
Cache line in the "acting" processor
SHW
SHR
Transaction due to events snooped on the common BUS

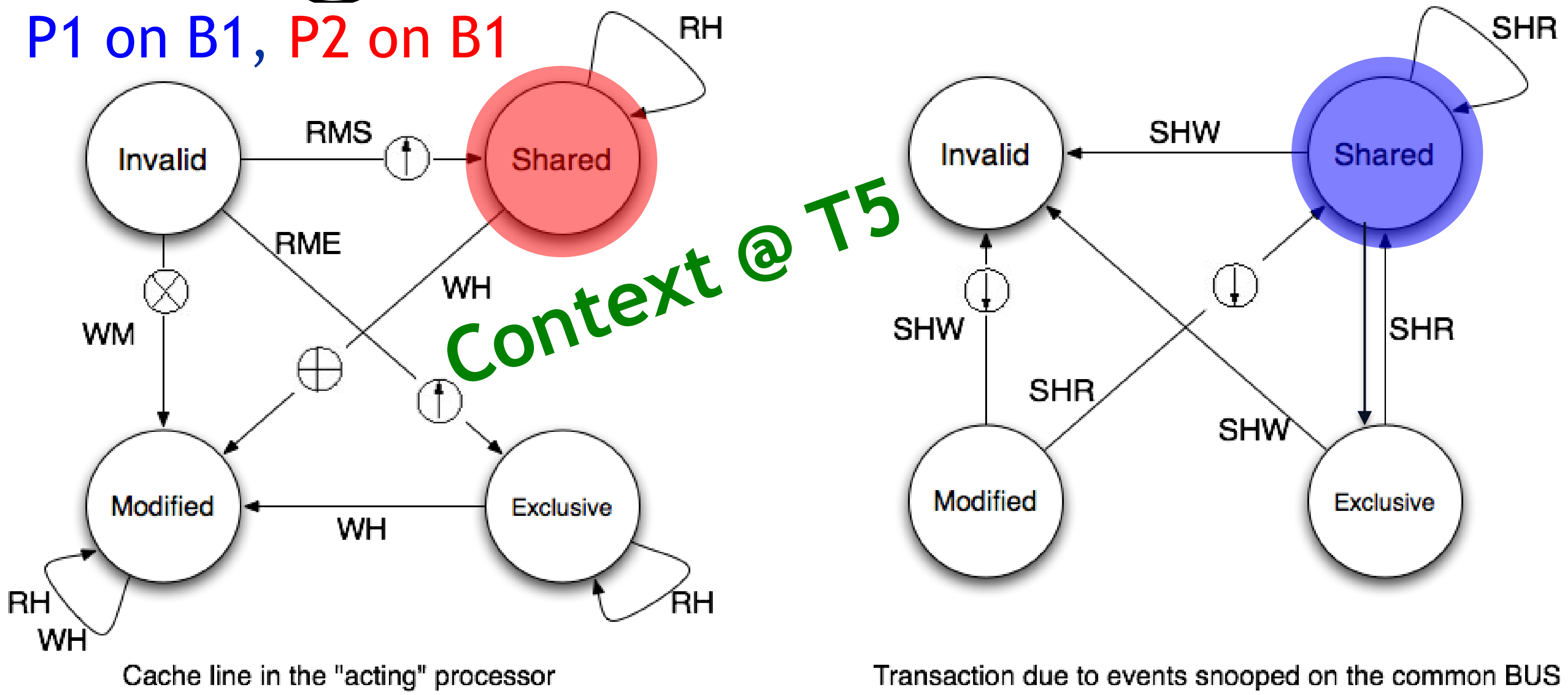
BUS Transactions

RH =	Read Hit
RMS =	Read Miss, Shared
RME =	Read Miss, Exclusive
WH =	Write Hit
WM =	Write Miss
SHR =	Snoop Hit on a Read
SHW =	Snoop Hit on a Write or Read-with-Intent-to-Modify

⊥ =	Snoop Push
⊗ =	Invalidate Transaction
⊕ =	Read-with-Intent-to-Modify
↑ =	Cache Block Fill

@T5: P2 Read Block 1

P1 on B1, P2 on B1



Cache line in the "acting" processor

Transaction due to events snooped on the common BUS

BUS Transactions

RH =	Read Hit
RMS =	Read Miss, Shared
RME =	Read Miss, Exclusive
WH =	Write Hit
WM =	Write Miss
SHR =	Snoop Hit on a Read
SHW =	Snoop Hit on a Write or Read-with-Intent-to-Modify

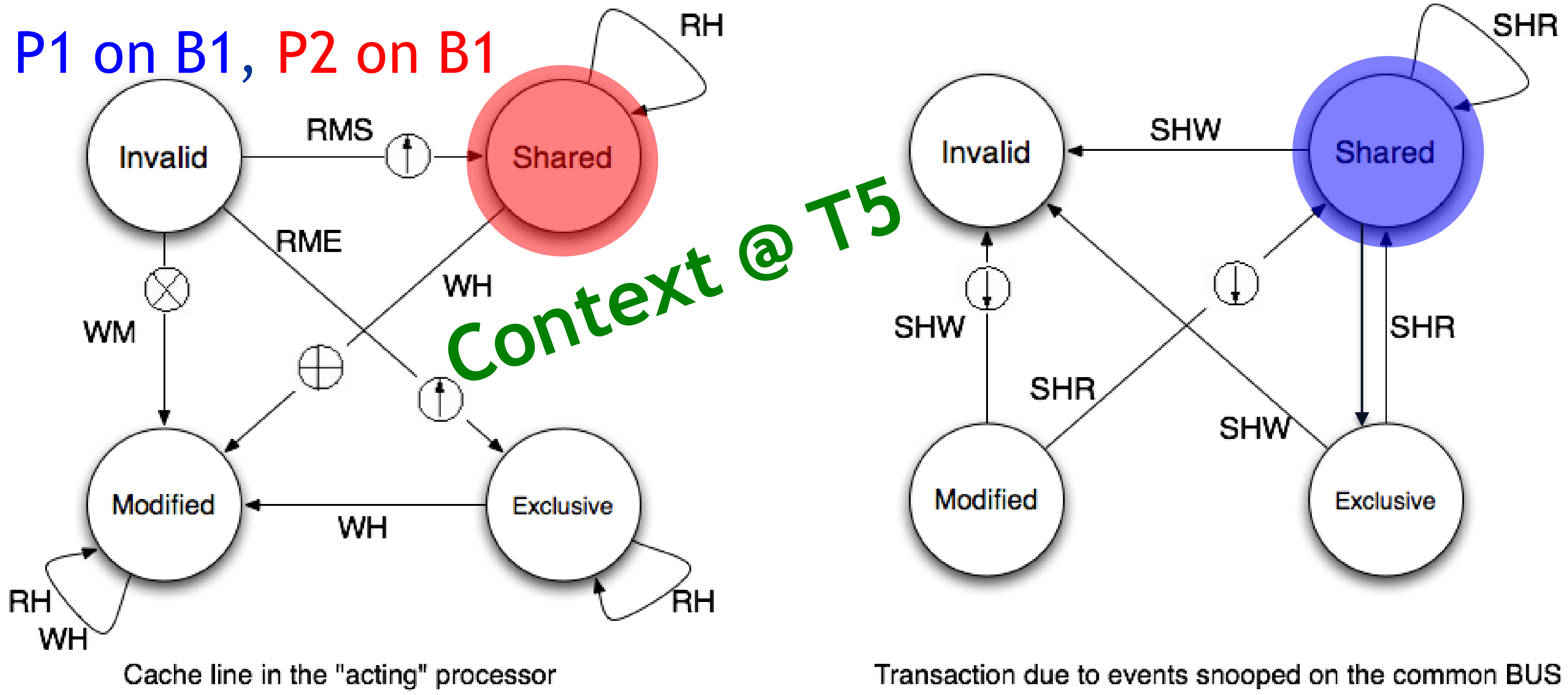
↓	= Snoop Push
⊗	= Invalidate Transaction
⊕	= Read-with-Intent-to-Modify
↑	= Cache Block Fill

Time 5

Time	After Operation	P1 cache block state	P2 cache block state	Memory at block 0 up to date?	Memory at block 1 up to date?
0	P1: read block 0	Exclusive (0)	Invalid (0)	Yes	Yes
1	P1: write block 0	Modified (0)	Invalid (0)	No	Yes
2	P2: read block 0	Shared (0)	Shared (0)	Yes	Yes
3	P2: write block 0	Invalid (0)	Modified (0)	No	Yes
4	P1: read block 1	Exclusive (1)	Modified (0)	No	Yes
5	P2: read block 1	Shared (1)	Shared (1)	Yes	Yes
6	P1: write block 1				
7	P2: write block 1				

@T6: P1 Write Block 1

P1 on B1, P2 on B1



BUS Transactions

RH =	Read Hit
RMS =	Read Miss, Shared
RME =	Read Miss, Exclusive
WH =	Write Hit
WM =	Write Miss
SHR =	Snoop Hit on a Read
SHW =	Snoop Hit on a Write or Read-with-Intent-to-Modify

= Snoop Push

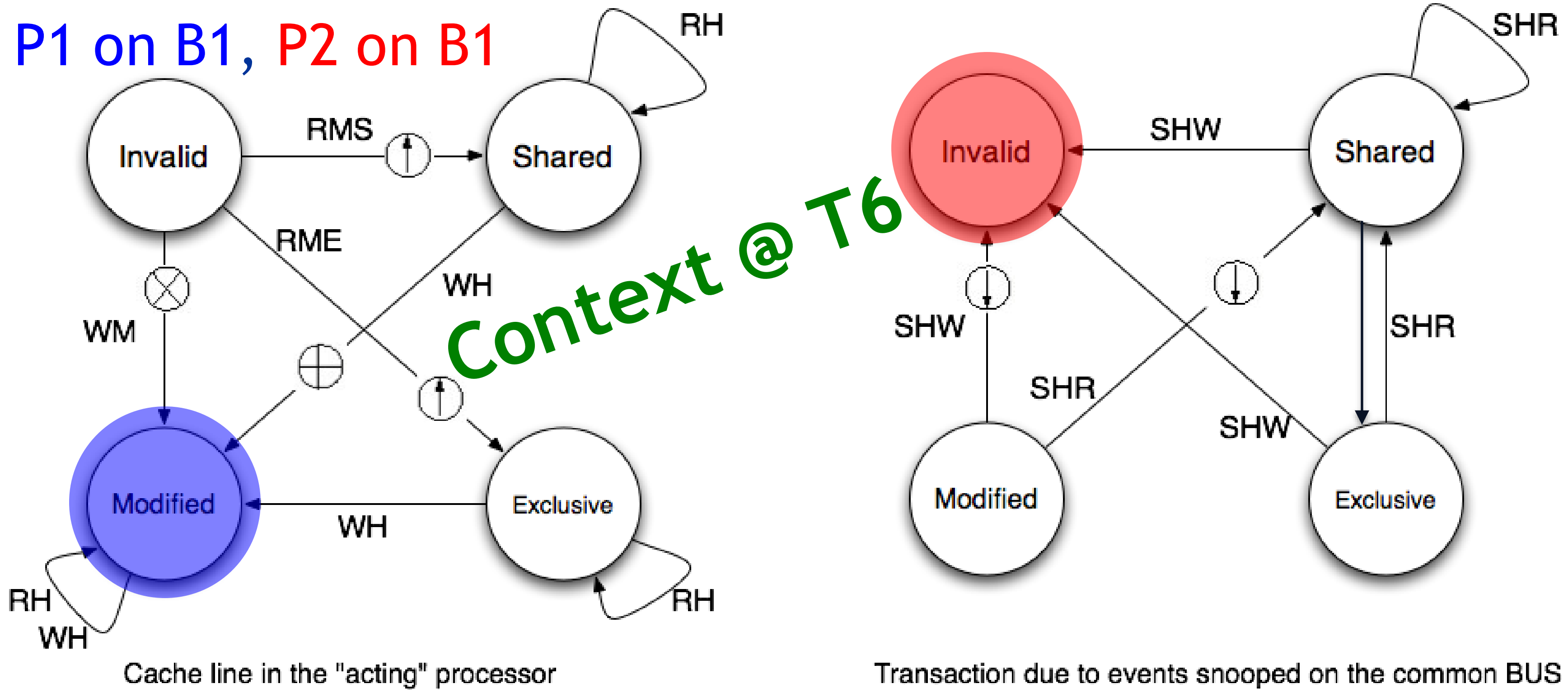
= Invalidate Transaction

= Read-with-Intent-to-Modify

= Cache Block Fill

@T6: P1 Write Block 1

P1 on B1, P2 on B1



Cache line in the "acting" processor

Transaction due to events snooped on the common BUS

BUS Transactions

RH =	Read Hit
RMS =	Read Miss, Shared
RME =	Read Miss, Exclusive
WH =	Write Hit
WM =	Write Miss
SHR =	Snoop Hit on a Read
SHW =	Snoop Hit on a Write or Read-with-Intent-to-Modify

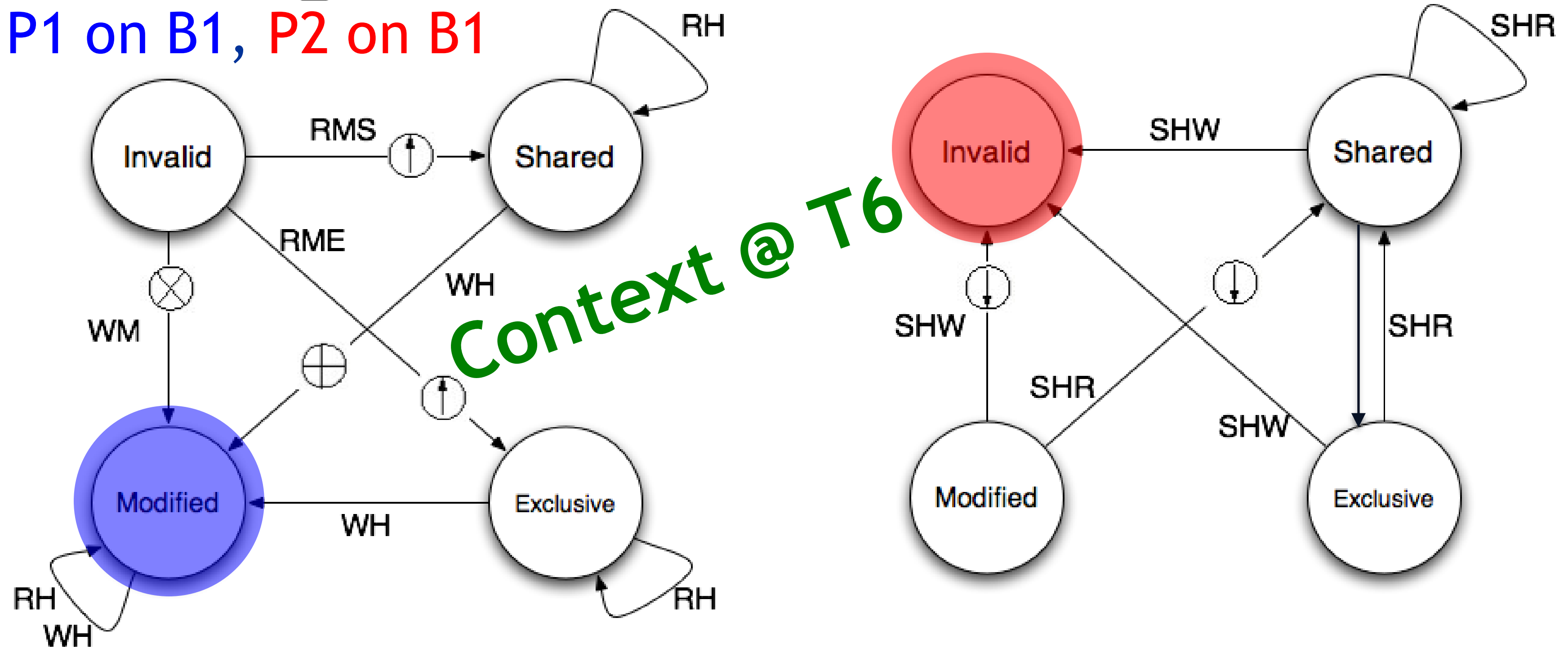
ⓘ	= Snoop Push
⊗	= Invalidate Transaction
⊕	= Read-with-Intent-to-Modify
ⓘ	= Cache Block Fill

Time 6

Time	After Operation	P1 cache block state	P2 cache block state	Memory at block 0 up to date?	Memory at block 1 up to date?
0	P1: read block 0	Exclusive (0)	Invalid (0)	Yes	Yes
1	P1: write block 0	Modified (0)	Invalid (0)	No	Yes
2	P2: read block 0	Shared (0)	Shared (0)	Yes	Yes
3	P2: write block 0	Invalid (0)	Modified (0)	No	Yes
4	P1: read block 1	Exclusive (1)	Modified (0)	No	Yes
5	P2: read block 1	Shared (1)	Shared (1)	Yes	Yes
6	P1: write block 1	Modified (1)	Invalid (1)	Yes	No
7	P2: write block 1				

@T7: P2 Write Block 1

P1 on B1, P2 on B1



Cache line in the "acting" processor

Transaction due to events snooped on the common BUS

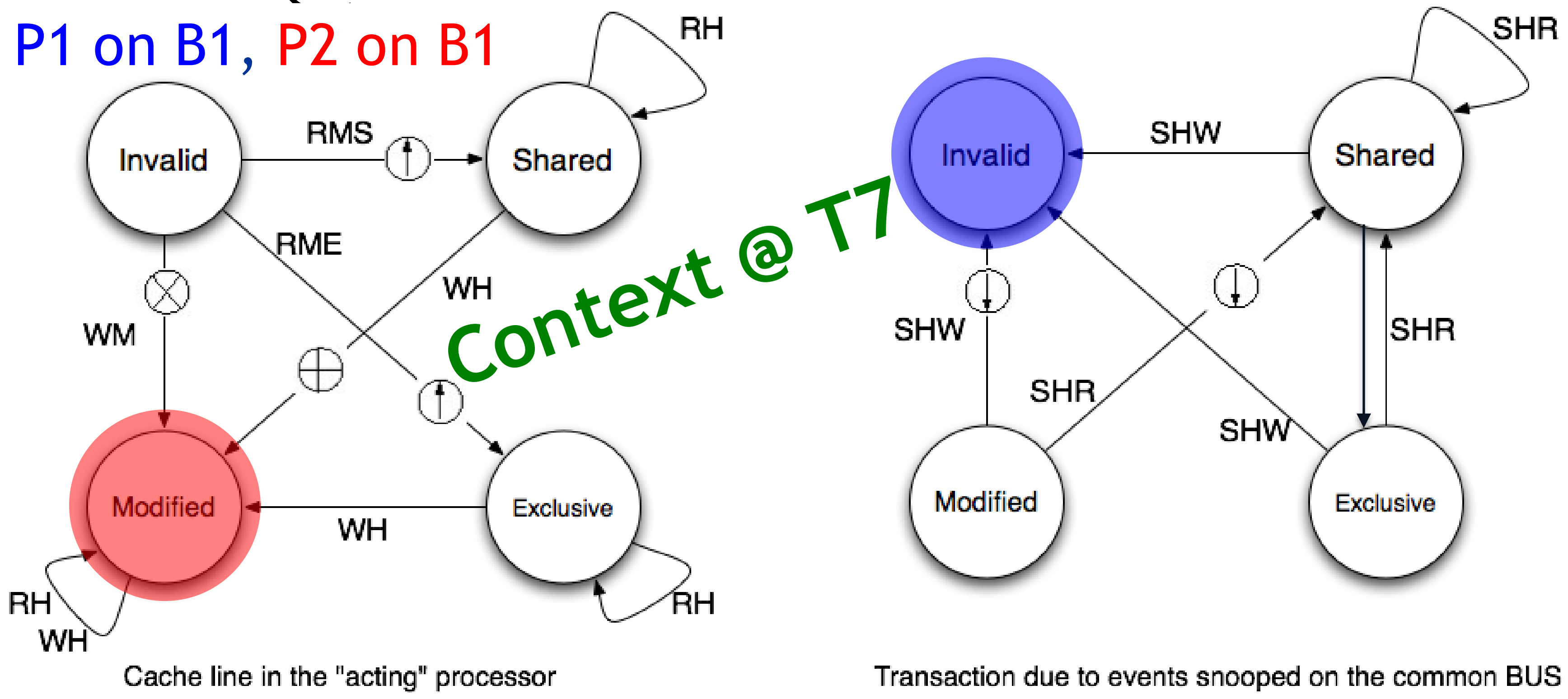
BUS Transactions

RH =	Read Hit
RMS =	Read Miss, Shared
RME =	Read Miss, Exclusive
WH =	Write Hit
WM =	Write Miss
SHR =	Snoop Hit on a Read
SHW =	Snoop Hit on a Write or Read-with-Intent-to-Modify

↓	= Snoop Push
⊗	= Invalidate Transaction
⊕	= Read-with-Intent-to-Modify
↑	= Cache Block Fill

@T7: P2 Write Block 1

P1 on B1, P2 on B1



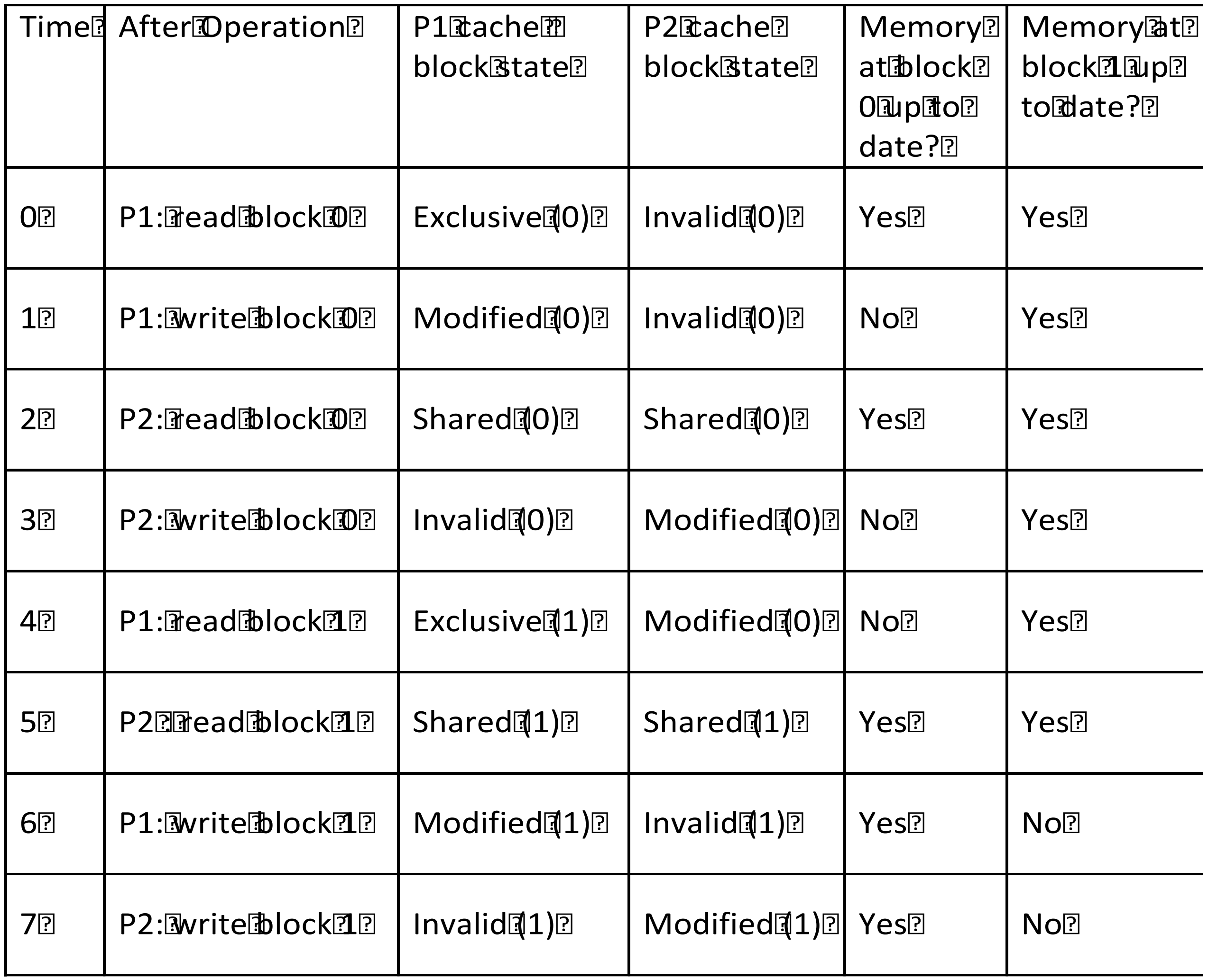
Context @ T7
Invalid
Shared
Modified
Exclusive
RMS
RME
WM
WH
RH
SHW
SHR
Cache line in the "acting" processor
Transaction due to events snooped on the common BUS

BUS Transactions

RH =	Read Hit
RMS =	Read Miss, Shared
RME =	Read Miss, Exclusive
WH =	Write Hit
WM =	Write Miss
SHR =	Snoop Hit on a Read
SHW =	Snoop Hit on a Write or Read-with-Intent-to-Modify

ⓘ =	Snoop Push
⊗ =	Invalidate Transaction
⊕ =	Read-with-Intent-to-Modify
ⓘ =	Cache Block Fill

Example



Time	After Operation	P1 cache block state	P2 cache block state	Memory at block 0 up to date?	Memory at block 1 up to date?
0	P1: read block 0	Exclusive (0)	Invalid (0)	Yes	Yes
1	P1: write block 0	Modified (0)	Invalid (0)	No	Yes
2	P2: read block 0	Shared (0)	Shared (0)	Yes	Yes
3	P2: write block 0	Invalid (0)	Modified (0)	No	Yes
4	P1: read block 1	Exclusive (1)	Modified (0)	No	Yes
5	P2: read block 1	Shared (1)	Shared (1)	Yes	Yes
6	P1: write block 1	Modified (1)	Invalid (1)	Yes	No
7	P2: write block 1	Invalid (1)	Modified (1)	Yes	No

The Problem of Memory Consistency

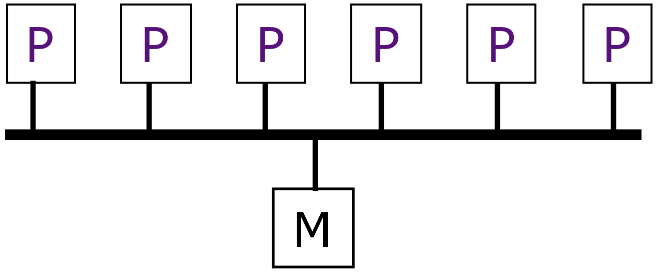
- What is consistency? When must a processor see the new value of a data updated by another processor?

```
P1:    A = 0;                    P2:   B = 0;
   .....                        .....
  A = 1;                         B = 1;
L1:    if (B == 0) ...           L2:   if (A == 0) ...
```

- Impossible for both if statements L1 & L2 to be true?
  - What if write invalidate is delayed & processor continues?
- Memory consistency models:
  what are the rules for such cases?

Sequential Consistency

A Memory Model



"A system is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in the order specified by the program"

Leslie Lamport

Sequential Consistency =

arbitrary order-preserving interleaving of memory references of sequential programs

The Problem of Memory Consistency

- Schemes faster execution to sequential consistency
- Not really an issue for most programs; they are synchronized
  - A program is synchronized if all access to shared data are ordered by synchronization operations

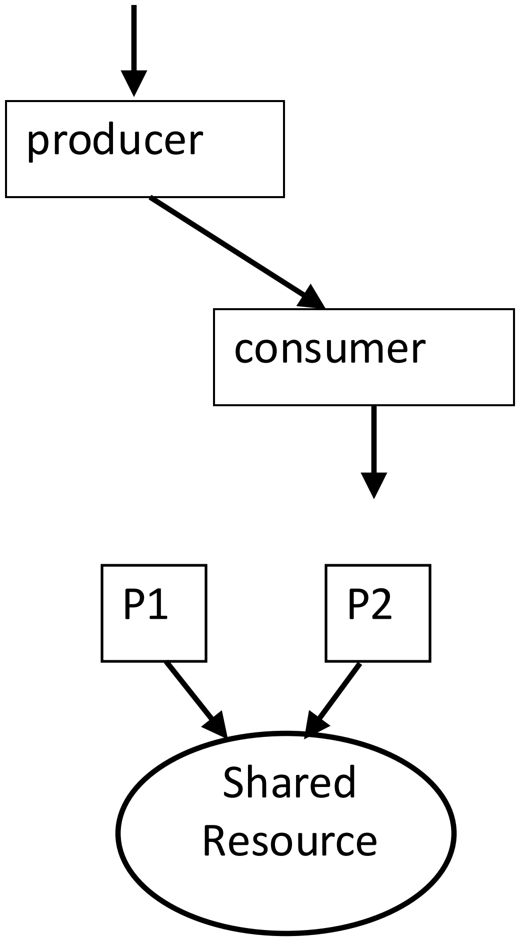
```
write (x)
...
release (s) {unlock}
...
acquire (s) {lock}
...
read(x)
```

Synchronization

The need for synchronization arises whenever there are concurrent processes in a system (*even in a uniprocessor system*).

Two classes of synchronization:

- *Producer-Consumer:* A consumer process must wait until the producer process has produced data
- *Mutual Exclusion:* Ensure that only one process uses a resource at a given time



ISA Support for Mutual-Exclusion Locks

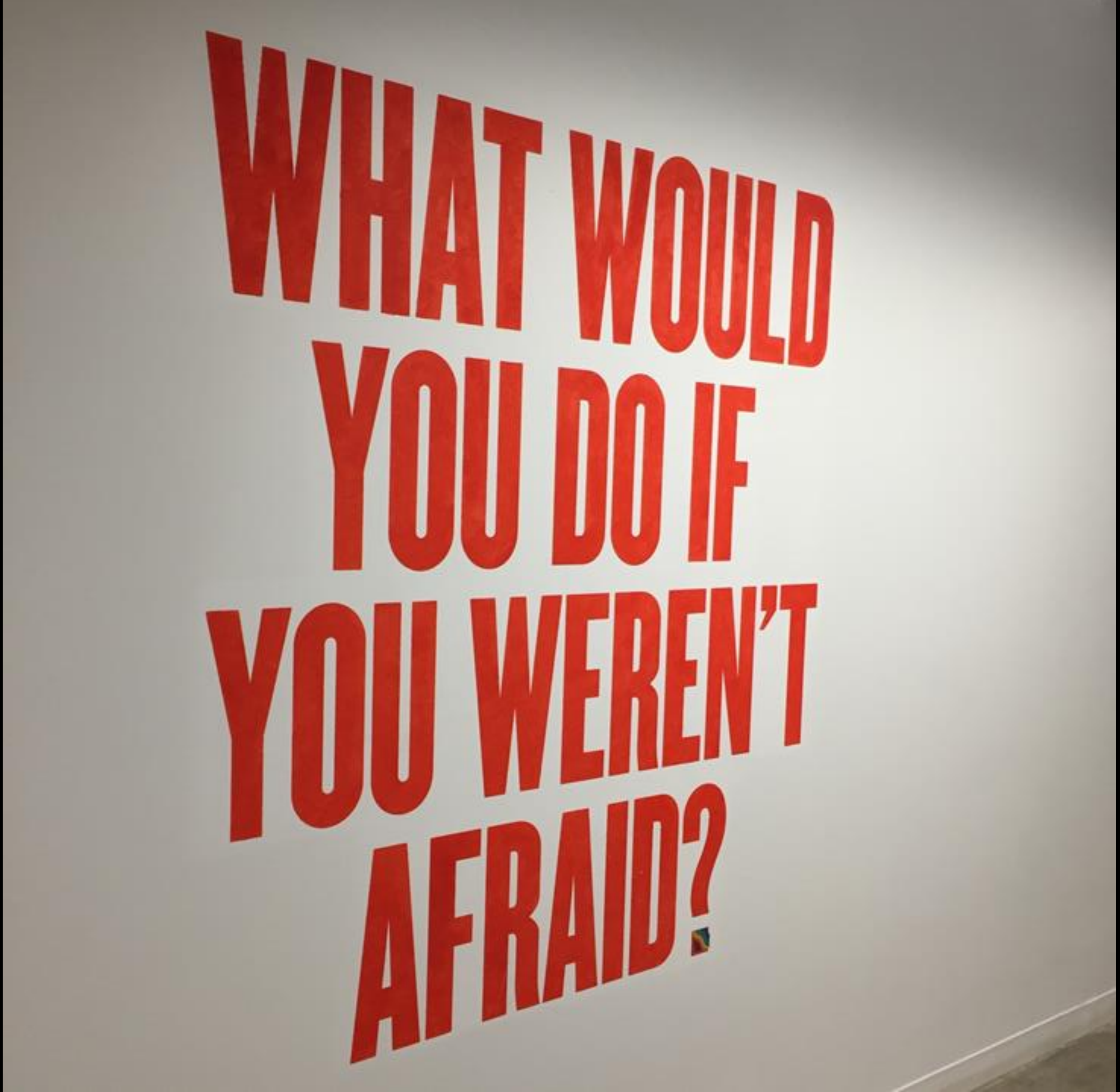
- Regular loads and stores in SC model (plus fences in weaker model) sufficient to implement mutual exclusion, but inefficient and complex code
- Therefore, atomic read-modify-write (RMW) instructions added to ISAs to support mutual exclusion

- Many forms of atomic RMW instruction possible, some simple examples:
  - Test and set (reg_x = M[a]; M[a]=1)
  - Swap (reg_x=M[a]; M[a] = reg_y)

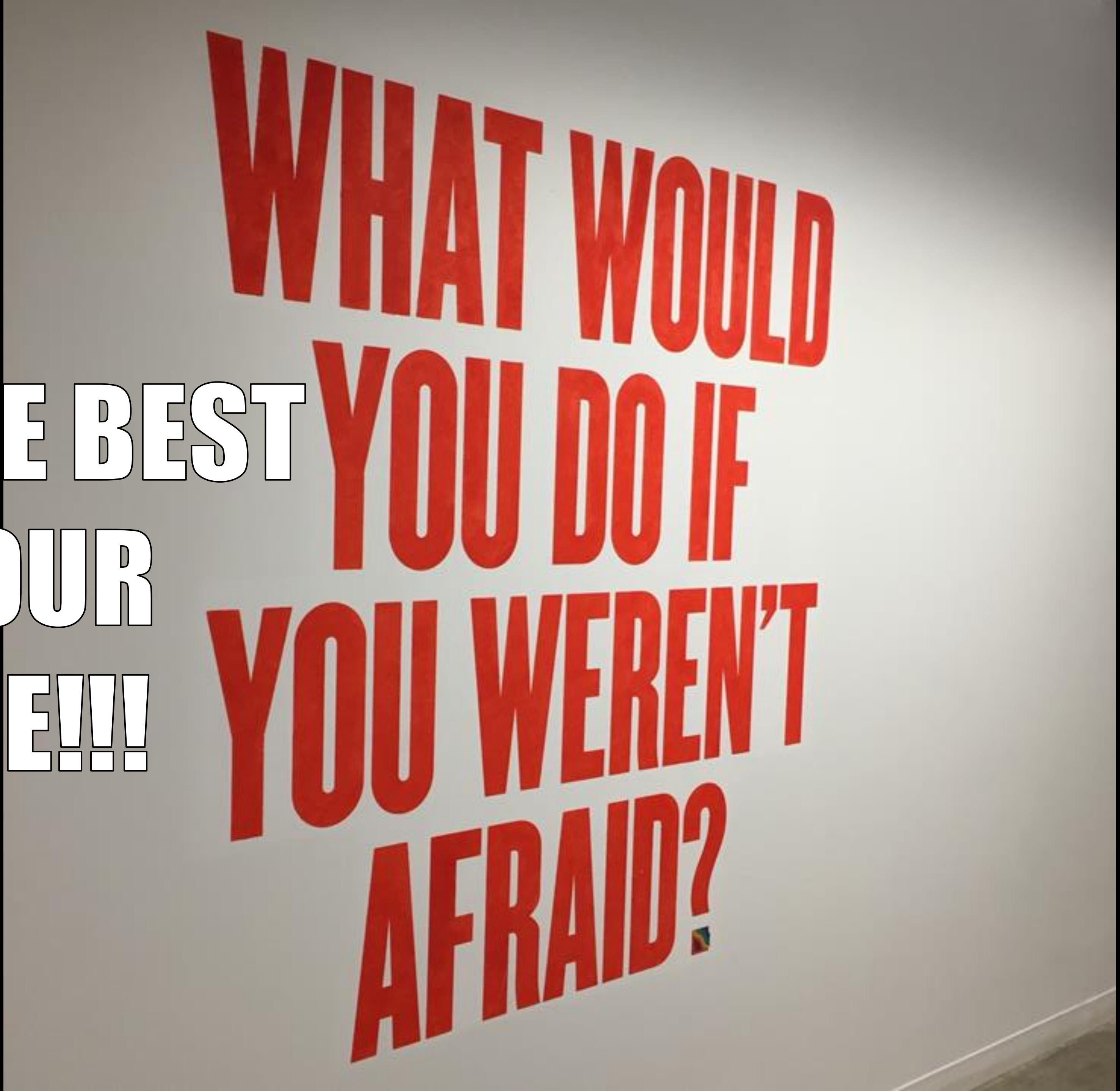
More on locks…

- Smartlocks: Lock Acquisition Scheduling for Self-Aware Synchronization
  - Jonathan Eastep, Michael D. Wingate, Marco D. Santambrogio, Anant Agarwal.

- PDF available under OEPN Access Policy: https://dspace.mit.edu/handle/1721.1/85868


WHAT WOULD
YOU DO IF
YOU WEREN'T
AFRAID?

ALL THE BEST FOR YOUR FUTURE!!!



LEONARDO

It always seems impossible until it is done

The Multimagineering Project

CSI POLITECNICO MILANO 1863

A 5 CFU Official PoliMi Course, starting Sept 2025
Launch event: 23.07.2025, 6pm - Conference Room, DEIB

Advanced Computer Architectures

(High Performance Processors and Systems)

Parallel Processors: MIMD Architectures

Politecnico di Milano

v0