CS302

Synchronization

Spring 2025

Arkaprava Basu & Babak Falsafi

parsa.epfl.ch/course-info/cs302

Adapted from slides originally developed by Profs. Falsafi, Fatahalian, Mowry, Wenisch of CMU, Michigan

Where are We?

M	T	W	T	F
17-Feb	18-Feb	19-Feb	20-Feb	21-Feb
24-Feb	25-Feb	26-Feb	27-Feb	28-Feb
3-Mar	4-Mar	5-Mar	6-Mar	7-Mar
10-Mar	11-Mar	12-Mar	13-Mar	14-Mar
17-Mar	18-Mar	19-Mar	20-Mar	21-Mar
24-Mar	25-Mar		27-Mar	28-Mar
31-Mar	1-Apr	2-Apr	3-Apr	4-Apr
7-Apr	8-Apr	9-Apr	10-Apr	11-Apr
14-Apr	15-Apr	16-Apr	17-Apr	18-Apr
21-Apr	22-Apr	23-Apr	24-Apr	25-Apr
28-Apr	29-Apr	30-Apr	1-May	2-May
5-May	6-May	7-May	8-May	9-May
12-May	13-May	14-May	15-May	16-May
19-May	20-May	21-May	22-May	23-May
26-May	27-May	28-May	29-May	30-May

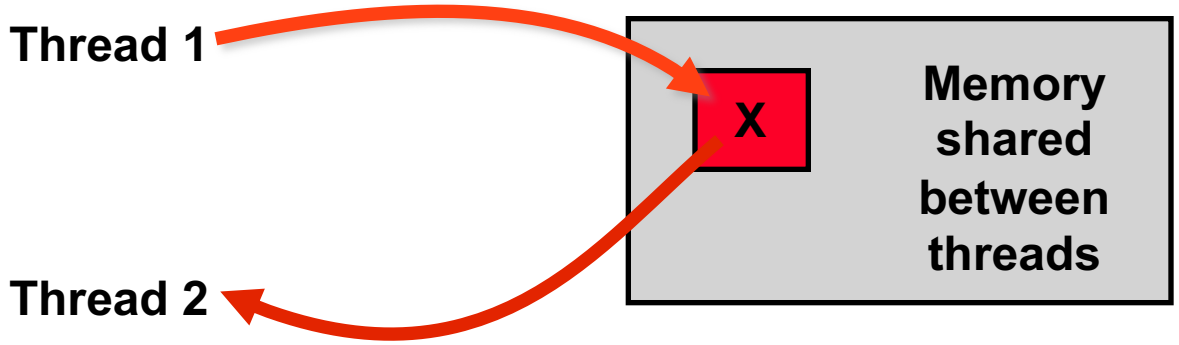
- Synchronization
  - Locks
- Exercise session
  - Instruction streams under various consistency models
- Next Tuesday:
  - Synchronization continued

Reminder: Principles of Parallel Computing

- From Lec. 2, need these for a parallel program:
  - Express the parallelism
  - Divide the work appropriately (load balancing)
  - Communication & Synchronization
- This lecture: how to synchronize in hardware
  - Closely tied to coherence & consistency model

Shared Memory Synchronization

- Recall Lectures 2 & 6:
  - Threads communicate by reading/writing shared vars.
  - Memory like a bulletin board, mostly global access



Thread 1:

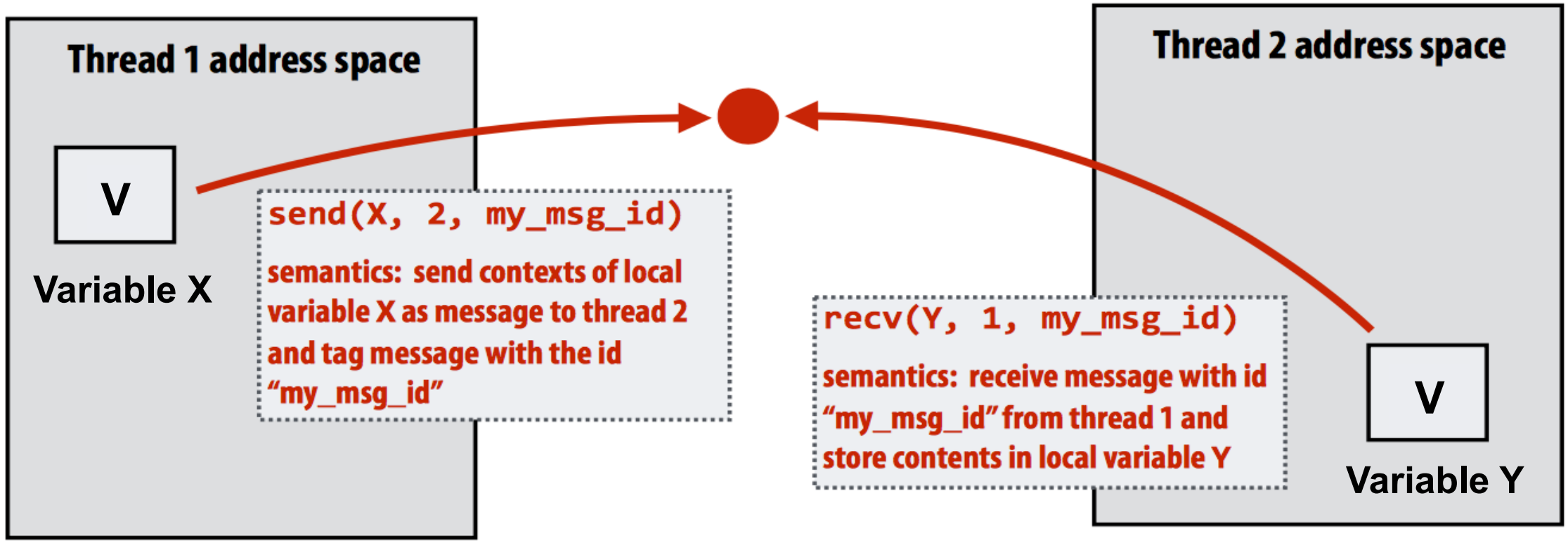
```
int X = 0;
X = 1;
```

Thread 2:

```
int X;
while (X == 0) {}
print X;
```

Other Method: Message Passing

- Threads can communicate via explicit messages
  - Synchronization is inherent in sending and receiving (Not in this lecture)



(Communication operations shown in red)

Synchronization Objectives

- Low overhead
  - Synchronization can limit scalability
    (E.g., single-lock OS kernels)
- Correctness (and ease of programming)
  - Synchronization failures are difficult to debug
- Coordination of HW and SW
  - SW semantics must be specified to prove correctness
  - HW can often improve efficiency

Synchronization Methods

- Mutual exclusion
  - Locks
- Point-to-point synchronization
  - Flags, barriers
- Software methods (not in this lecture)
  - Queues, counters, software pipelines
- Coarse-grain concurrency control
  - Transactional Memory

Phases of Synchronization

- Acquire method
  - How thread attempts to gain access to the resource
- Release method
  - How to enable other threads to access the resource
- Waiting algorithm
  - How thread waits for access to the resource

Locks

Focus on Implementation

- Previously we talked about locks mainly from a software perspective
  - What to lock, and when
- In this lecture, we focus on **how** locks work and their interactions with hardware/OS
  - Ties back to cache coherence protocol
- Because locks require communication…
  - Use memory locations to implement them!

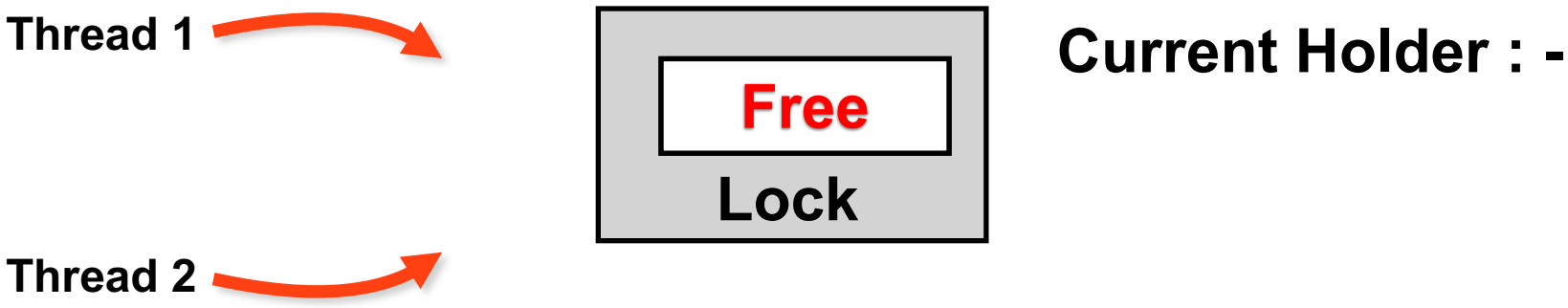
Desirable Lock Characteristics

- Low latency
  - Processors should be able to acquire free locks quickly
- Low traffic
  - Waiting for lock should generate little/no traffic
  - A busy lock should be handed off between processors with as little traffic as possible
- Scalability
  - Latency/traffic should scale reasonably with number of processors
- Low storage cost
- Fairness
  - Avoid starvation or substantial unfairness
  - Ideal: processors should acquire lock in order they request access

Conceptual Model of Locks

- Goal of locking is to force threads to access a location one at a time
  - This is called **mutual exclusion** because threads both exclude each other from executing

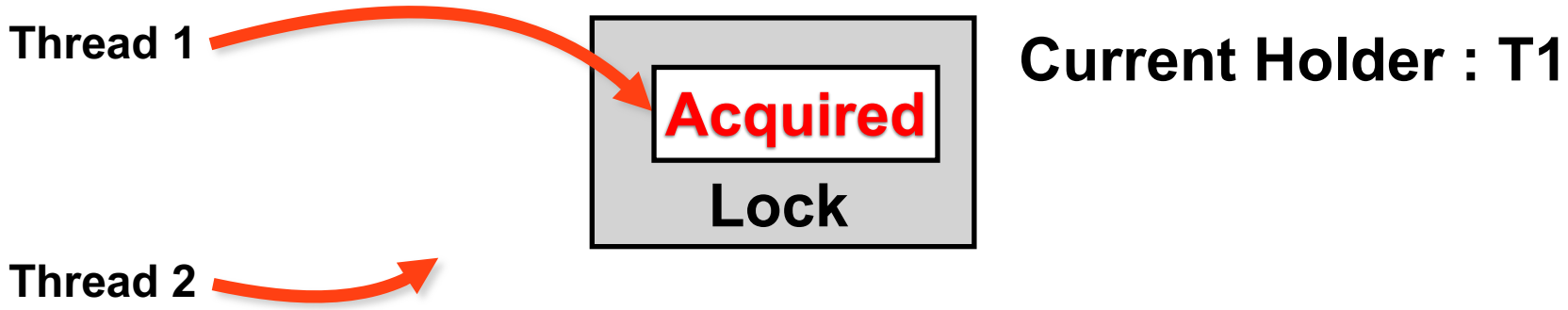
- To implement: use a construct called a **lock**
  - Just a memory location where threads communicate wrt. who can execute and who must wait



Conceptual Model of Locks

- Goal of locking is to force threads to access a location one at a time
  - This is called **mutual exclusion** because threads both exclude each other from executing

- To implement: use a construct called a **lock**
  - Just a memory location where threads communicate wrt. who can execute and who must wait



First Try Creating a Lock

- Simple idea:
  - If memory location holds 0, lock is free
  - Store 1 into it to “acquire” the lock, other threads have to wait until the holder stores 0 again

- Lock:

```
while (lock != 0);
lock = 1;
```

- Unlock:

```
lock = 0;
```

First Try Creating a Lock

- Simple idea:
  - If memory location holds 0, lock is free
  - Store 1 into it to “acquire” the lock, other threads have to wait until the holder stores 0 again
- Does this work?

```
Lock:
     ld   r1, mem[addr]        // load word into r1
     cmp  r1, #0               // if 0, store 1
     bnz  Lock                 // else, try again
     st   mem[addr], #1

Unlock:
     st   mem[addr], #0        // store 0 to address
```

First Try Creating a Lock

- Simple idea: Load and check if lock is 0
  - Does this work? (Hint: Think cache coherence)
  - Answer: No!
    - Instructions from load to store not executed atomically
    - Two cores can load 0, believe the lock is free, and we lose mutual exclusion

```
Lock:
     ld   r1, mem[addr]        // load word into r1
     cmp  r1, #0               // if 0, store 1
     bnz  Lock                 // else, try again
     st   mem[addr], #1

Unlock:
     st   mem[addr], #0        // store 0 to address
```

Need Atomics in Hardware

- Create a new instruction, “test-and-set” (TS)
  - **`ts reg, mem[addr]`**
- Atomically load memory location into reg **and** set contents of location to 1
  - Exercise: Sketch out how to create a lock using TS

Need Atomics in Hardware

- Create a new instruction, “test-and-set” (TS)
  - `ts reg, mem[addr]`
- Atomically load memory location into reg **and** set contents of location to 1
  - Exercise: Sketch out how to create a lock using TS
  - Answer:

```
Lock:
     ts   r1, mem[addr]       // load word into r1
     bnz  Lock                // if 0, lock obtained
     // fall-through, critical section
Unlock:
     st   mem[addr], #0       // store 0 to address
```

Simple TS Lock

- High-level pseudo code
  - Actual code in assembly

- Lock:

```
void Lock(int* lock){
  while (Test_and_Set(lock) != 0);
}
```

- Unlock:

```
void Unlock(volatile int* lock){
  *lock = 0;
}
```

Example: TS in x86

- x86
  - XCHG (swaps memory values)
  - LOCK prefix
  - Add to “ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, CMPXCH8B, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD” when destination operand is memory
  - Reachable using intrinsics, e.g., __atomic_exchange(..) on GCC/Clang

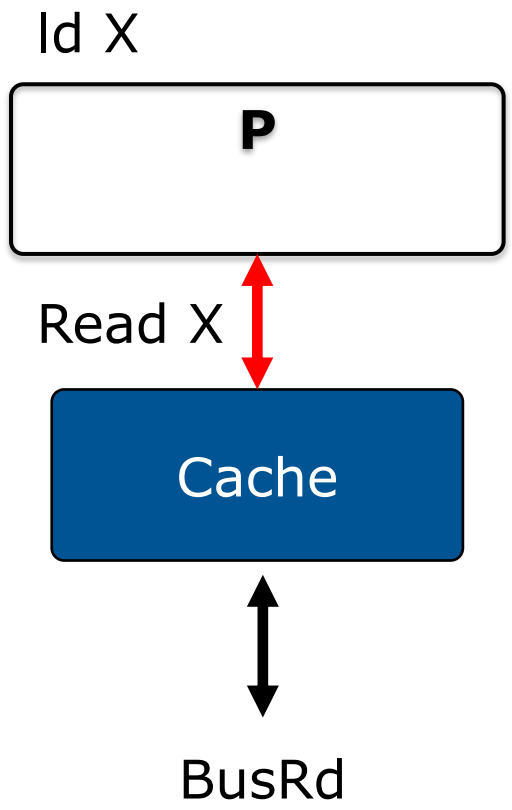
```
Test_and_set:
    mv %eax, #1
    xchg %eax, %(MEM)
    cmp %eax, #0
    bnz Test_and_set
```

Reminder: P-to-Cache Transactions

ld X – Read X

If block in cache

- in readable state, hit
- in invalid state or not in cache, request a readable copy

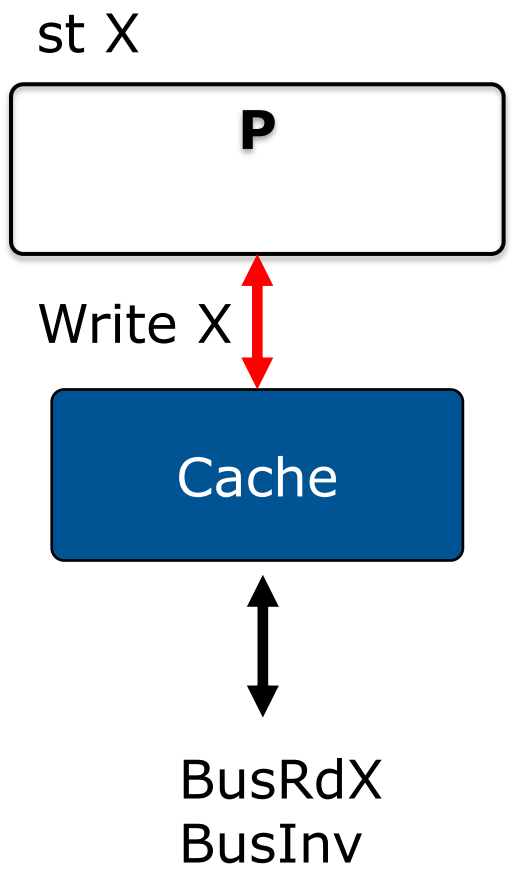


Reminder: P-to-Cache Transactions

st X – Write X

If block in cache

- in writable state, hit
- in readable, invalid state or not in cache, request a writable copy



New P-to-Cache (Atomic) Transaction

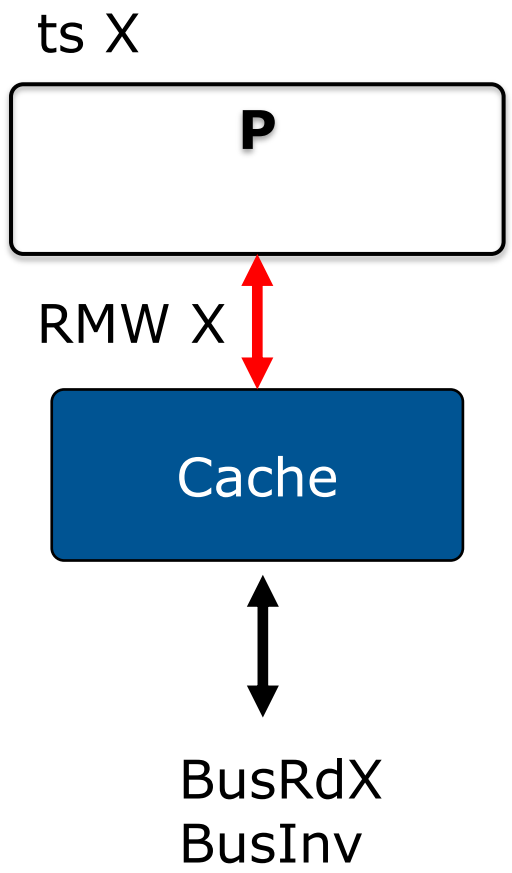
ts X – Read-Modify-Write X

- RMW for short
- Atomically reads and updates

Bus transactions similar to Write

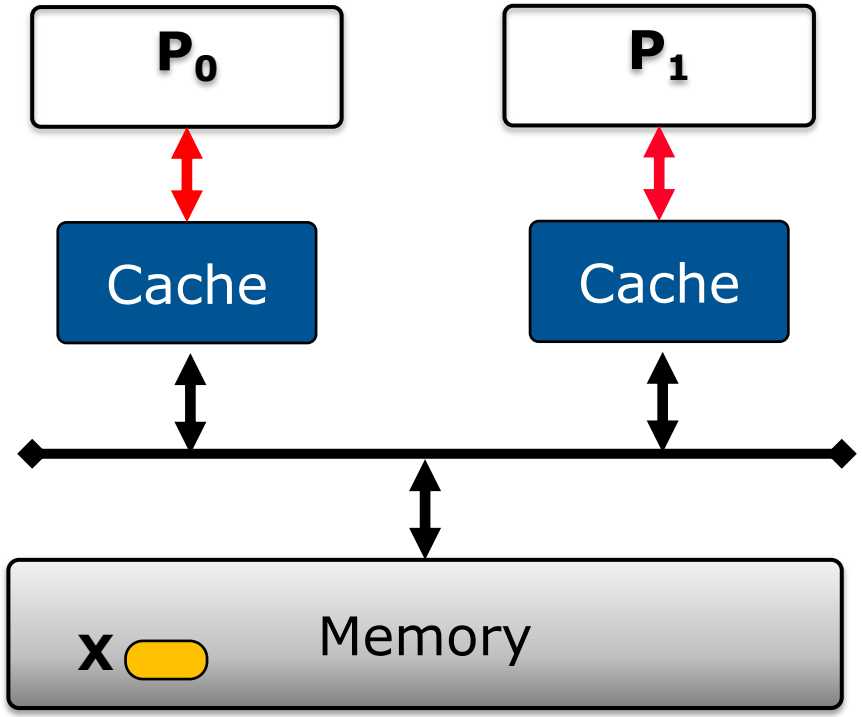
If block in cache

- in writable state, hit
- in readable, invalid state or not in cache, request a writable copy



Example Critical Section & Coherence Traffic

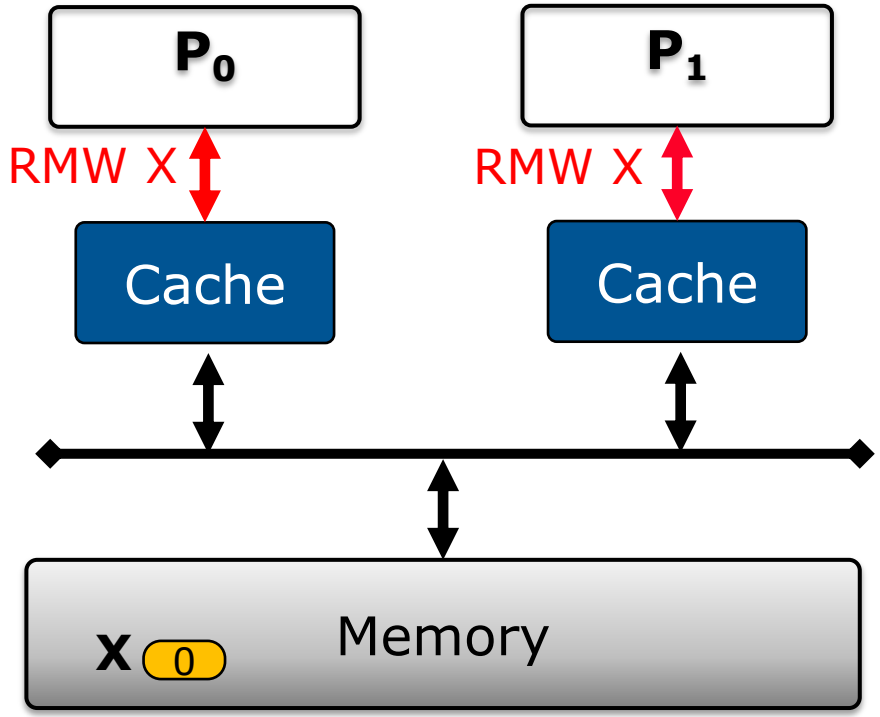
```
while ( Test-and-set(&X) != 0);
      ….    // update data structure
X= 0;     // unlock
```



Example Critical Section & Coherence Traffic

X is initially 0

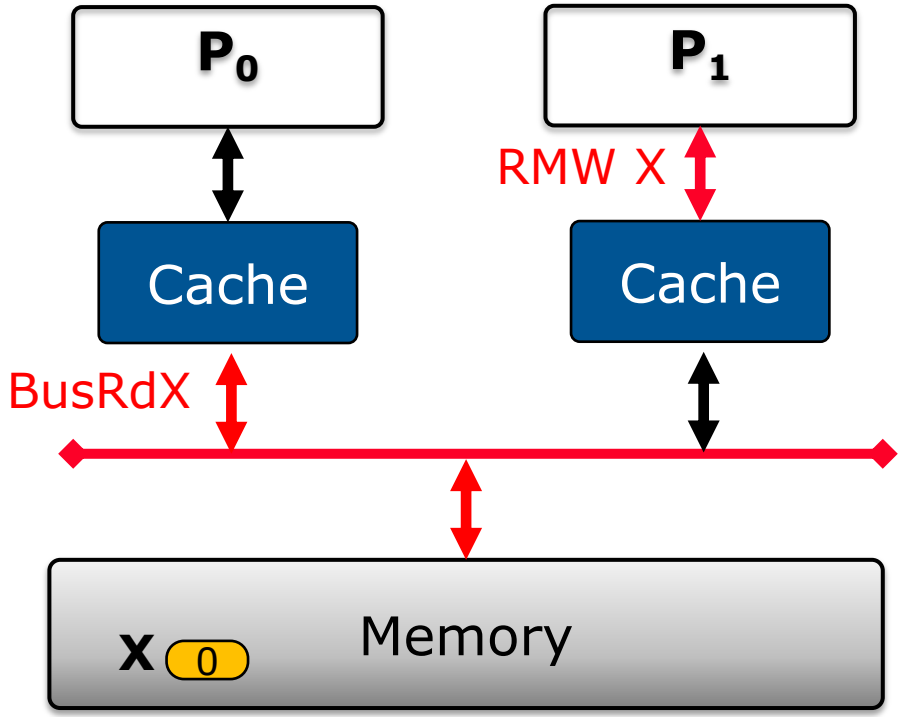
1. P0 & P1 try lock acquire



Example Critical Section & Coherence Traffic

X is initially 0

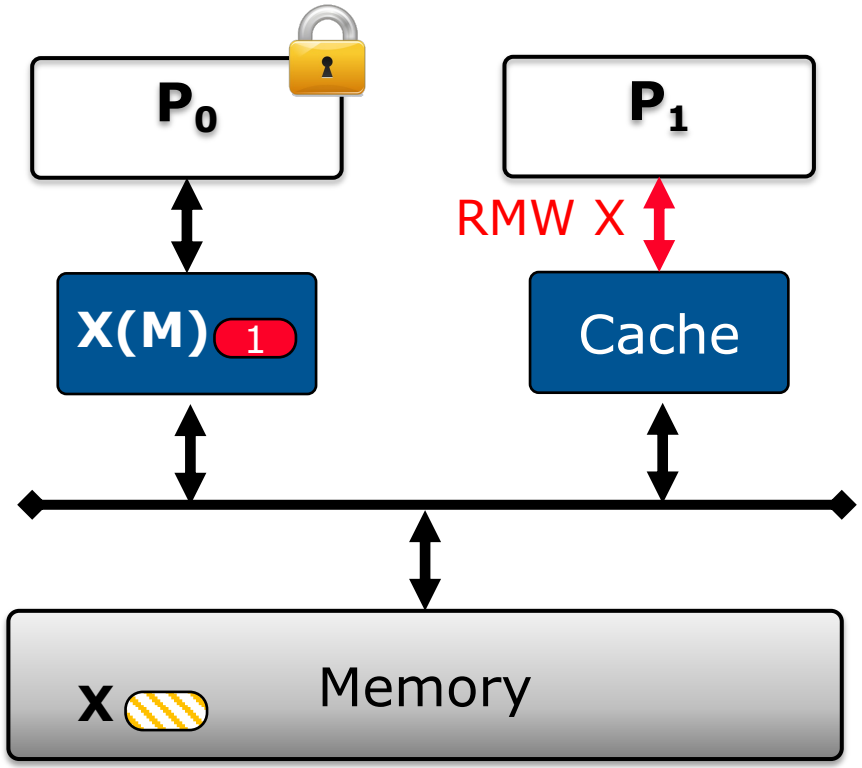
1. P0 & P1 try lock acquire
2. P0 wins bus arbitration



Example Critical Section & Coherence Traffic

X is initially 0

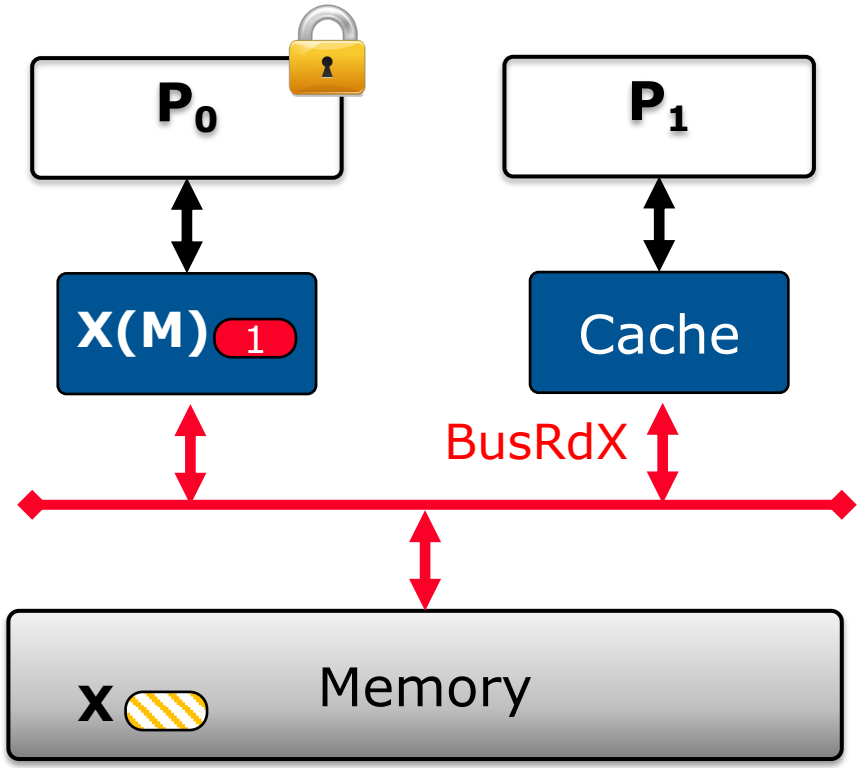
1. P0 & P1 try lock acquire
2. P0 wins bus arbitration
   - Reads 0
   - Writes 1



Example Critical Section & Coherence Traffic

X is initially 0

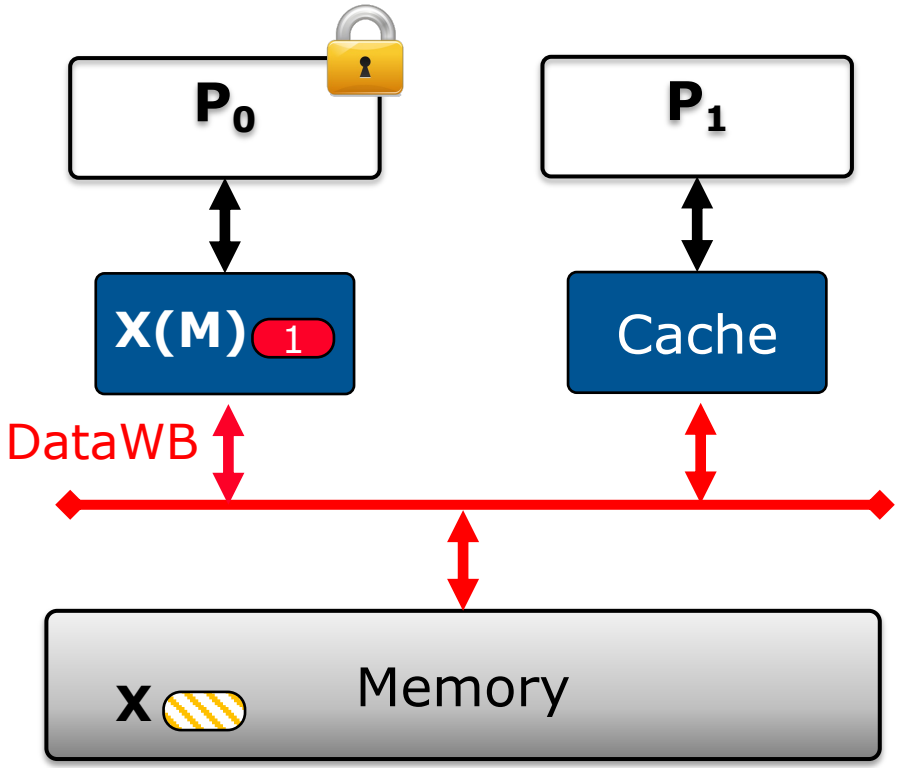
1. P0 & P1 try lock acquire
2. P0 wins bus arbitration
3. P1 gets a bus grant



Example Critical Section & Coherence Traffic

X is initially 0

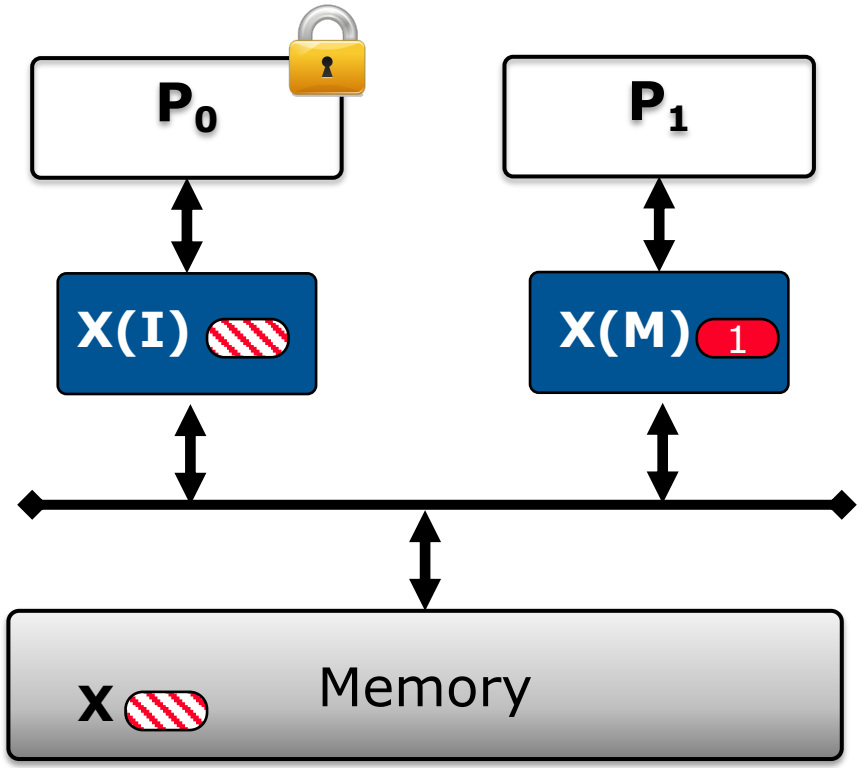
1. P0 & P1 try lock acquire
2. P0 wins bus arbitration
3. P1 gets a bus grant
   - P0 writes back X



Example Critical Section & Coherence Traffic

X is initially 0

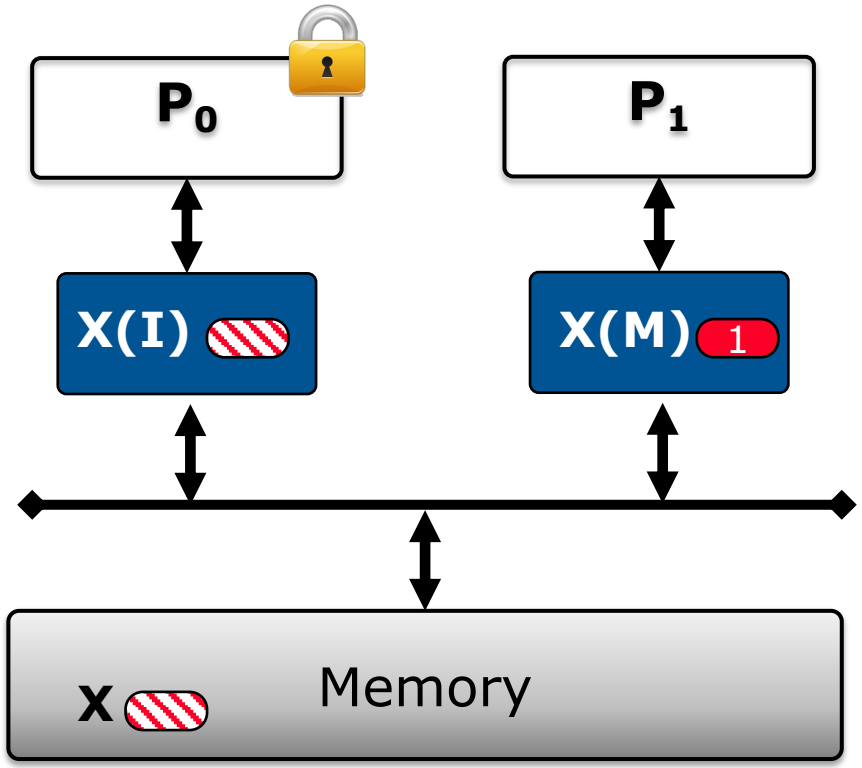
1. P0 & P1 try lock acquire
2. P0 wins bus arbitration
3. P1 gets a bus grant
   - P0 writes back X
   - P1 reads 1
   - P1 writes 1



Example Critical Section & Coherence Traffic

X is initially 0

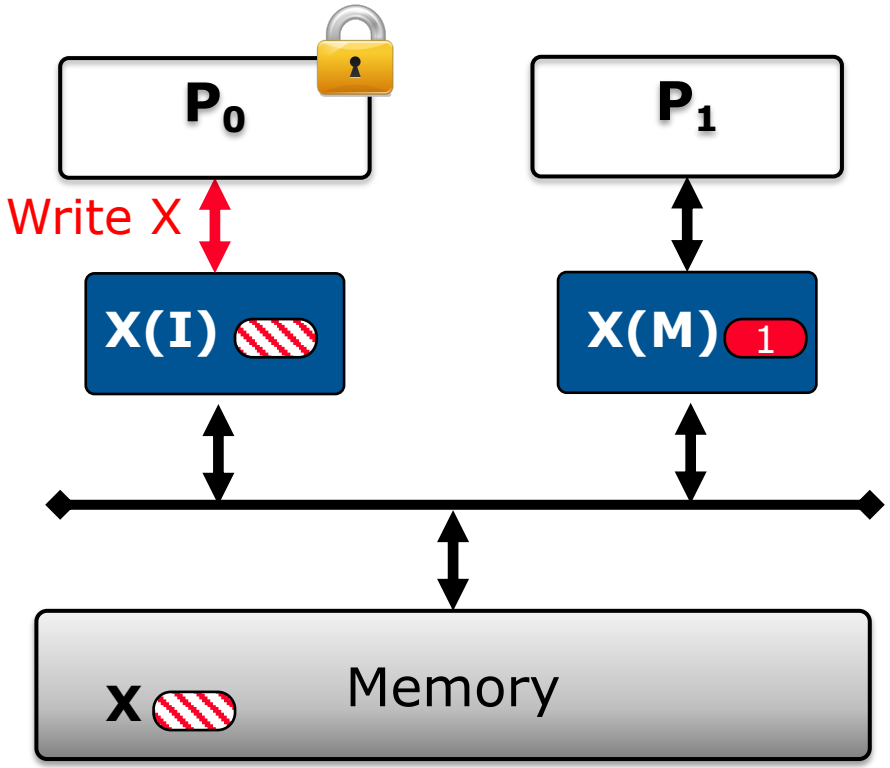
1. P0 & P1 try lock acquire
2. P0 wins bus arbitration
3. P1 gets a bus grant
4. P0 in critical section
   - P1 keeps looping on RMW reading 1
   - Not shown in diagram



Example Critical Section & Coherence Traffic

X is initially 0

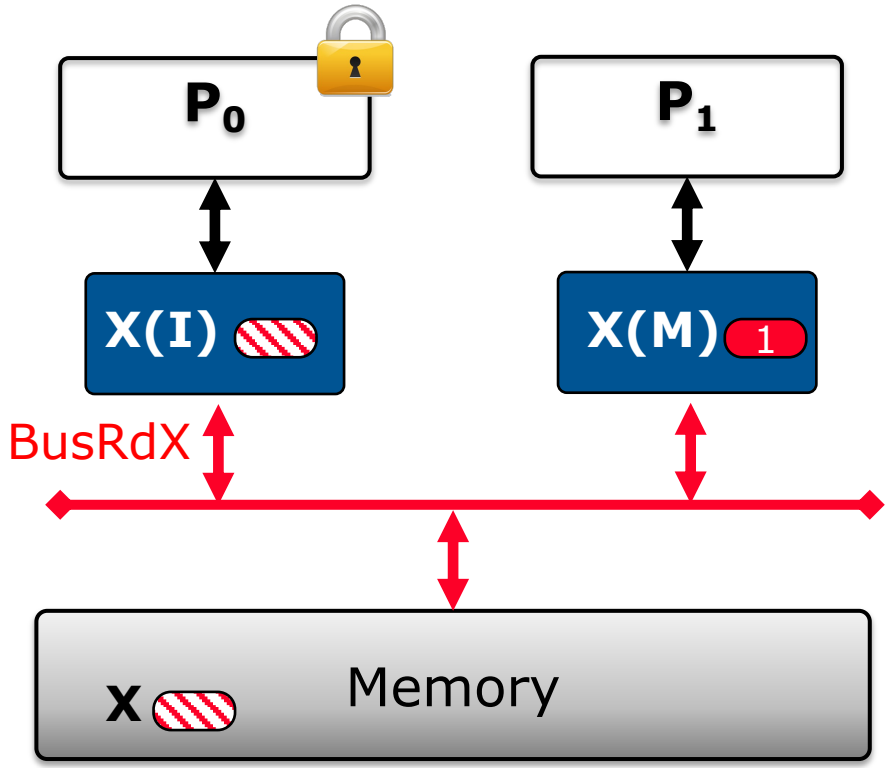
1. P0 & P1 try lock acquire
2. P0 wins bus arbitration
3. P1 gets a bus grant
4. P0 in critical section
5. P0 releases the lock
   - Writes 0



Example Critical Section & Coherence Traffic

X is initially 0

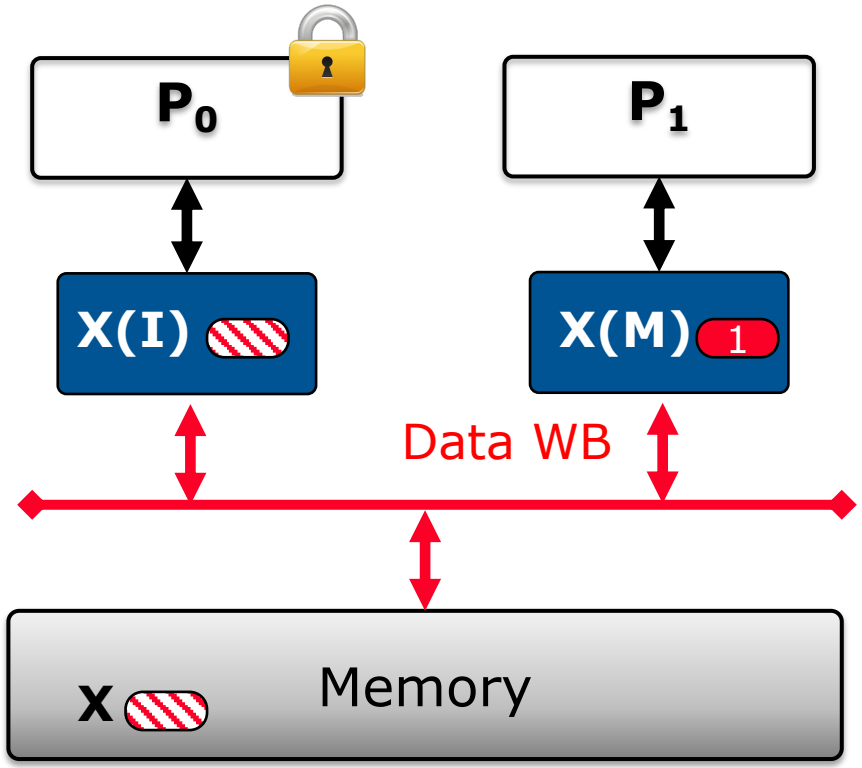
1. P0 & P1 try lock acquire
2. P0 wins bus arbitration
3. P1 gets a bus grant
4. P0 in critical section
5. P0 releases the lock
   - Writes 0



Example Critical Section & Coherence Traffic

X is initially 0

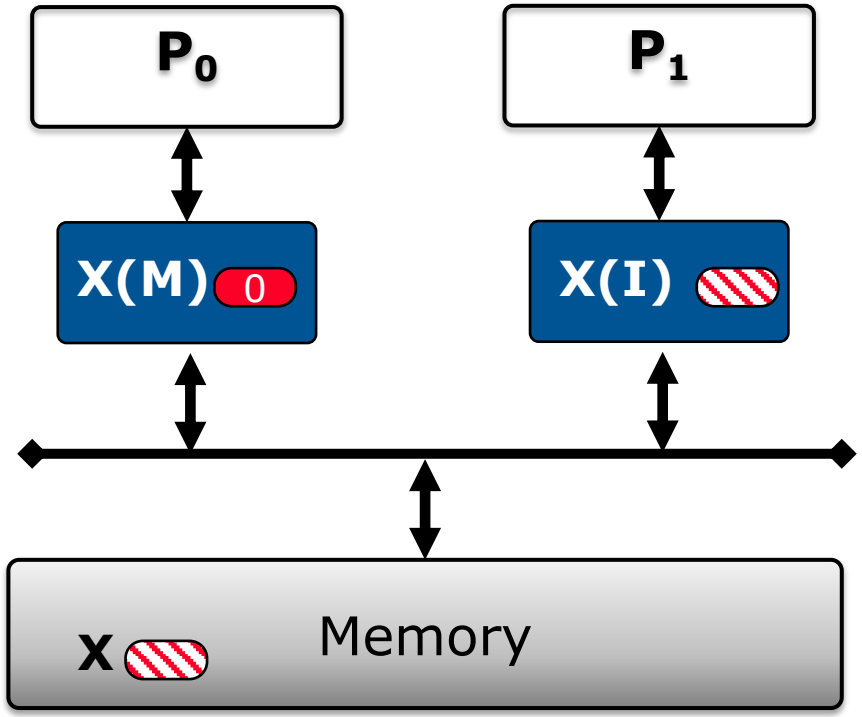
1. P0 & P1 try lock acquire
2. P0 wins bus arbitration
3. P1 gets a bus grant
4. P0 in critical section
5. P0 releases the lock
   - Writes 0



Example Critical Section & Coherence Traffic

X is initially 0

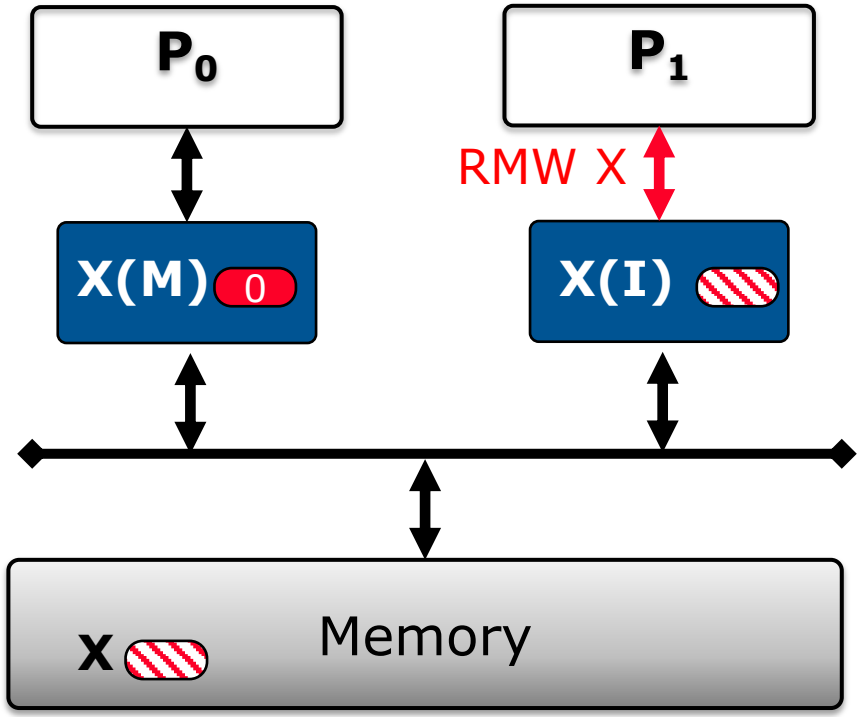
1. P0 & P1 try lock acquire
2. P0 wins bus arbitration
3. P1 gets a bus grant
4. P0 in critical section
5. P0 releases the lock
   - Writes 0



Example Critical Section & Coherence Traffic

X is initially 0

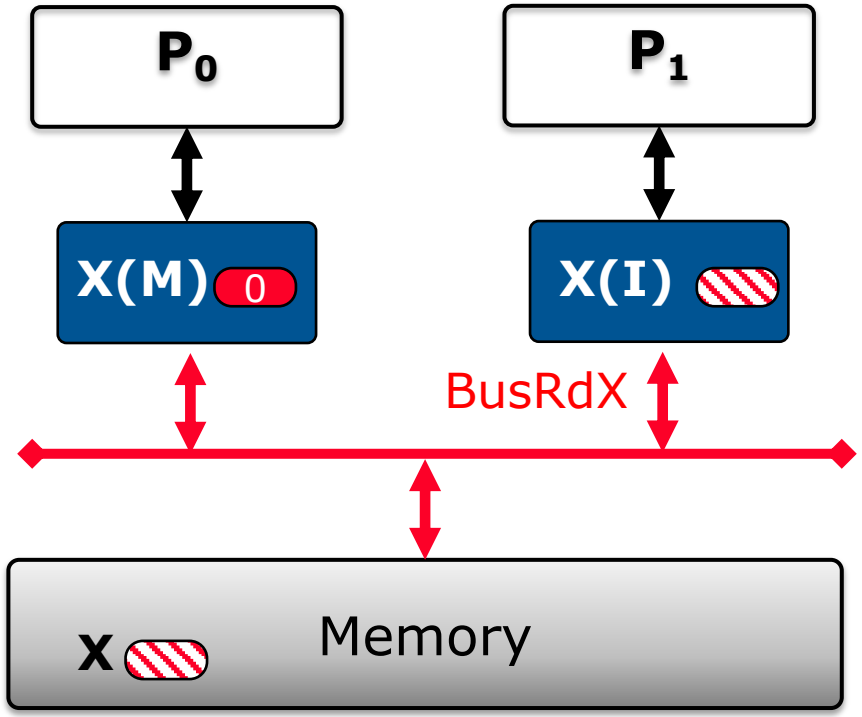
1. P0 & P1 try lock acquire
2. P0 wins bus arbitration
3. P1 gets a bus grant
4. P0 in critical section
5. P0 releases the lock
6. P1 retries lock acquire



Example Critical Section & Coherence Traffic

X is initially 0

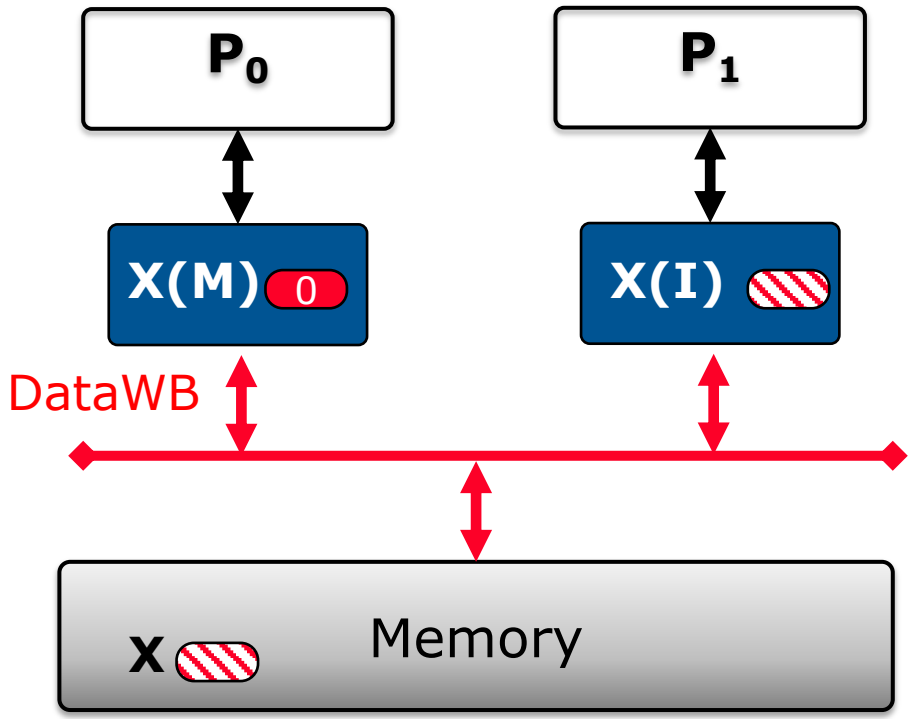
1. P0 & P1 try lock acquire
2. P0 wins bus arbitration
3. P1 gets a bus grant
4. P0 in critical section
5. P0 releases the lock
6. P1 retries lock acquire



Example Critical Section & Coherence Traffic

X is initially 0

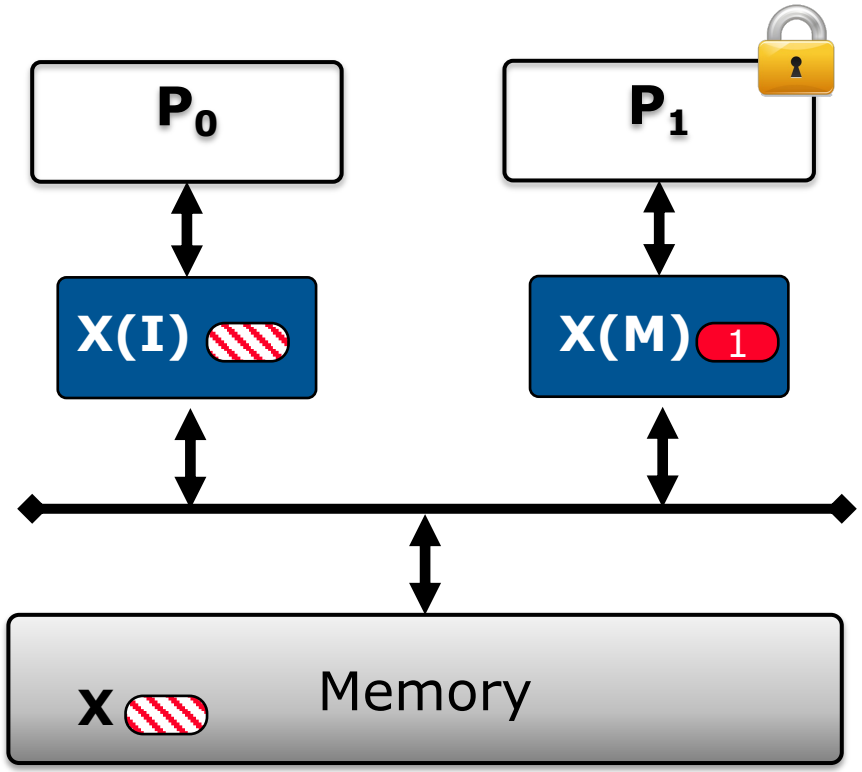
1. P0 & P1 try lock acquire
2. P0 wins bus arbitration
3. P1 gets a bus grant
4. P0 in critical section
5. P0 releases the lock
6. P1 retries lock acquire



Example Critical Section & Coherence Traffic

X is initially 0

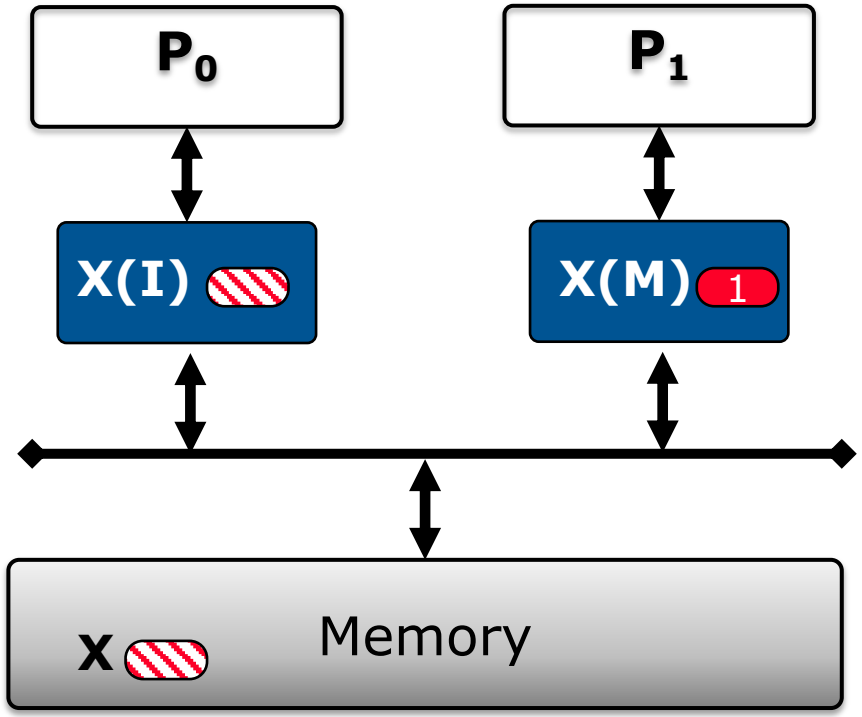
1. P0 & P1 try lock acquire
2. P0 wins bus arbitration
3. P1 gets a bus grant
4. P0 in critical section
5. P0 releases the lock
6. P1 retries lock acquire
   - Reads 0
   - Writes 1



Example Critical Section & Coherence Traffic

X is initially 0

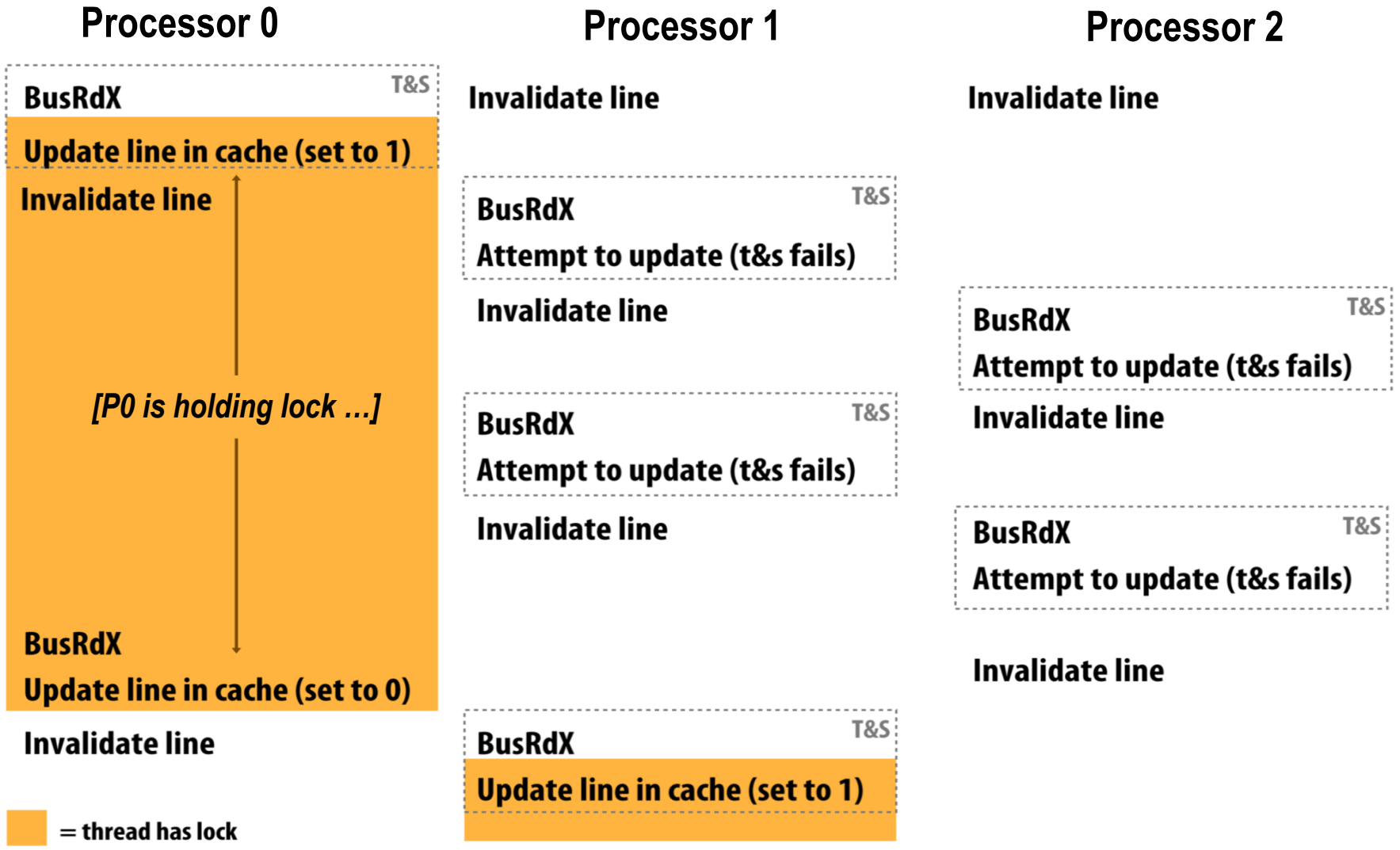
1. P0 & P1 try lock acquire
2. P0 wins bus arbitration
3. P1 gets a bus grant
4. P0 in critical section
5. P0 releases the lock
6. P1 retries lock acquire
   - Reads 0
   - Writes 1
7. P1 in critical section



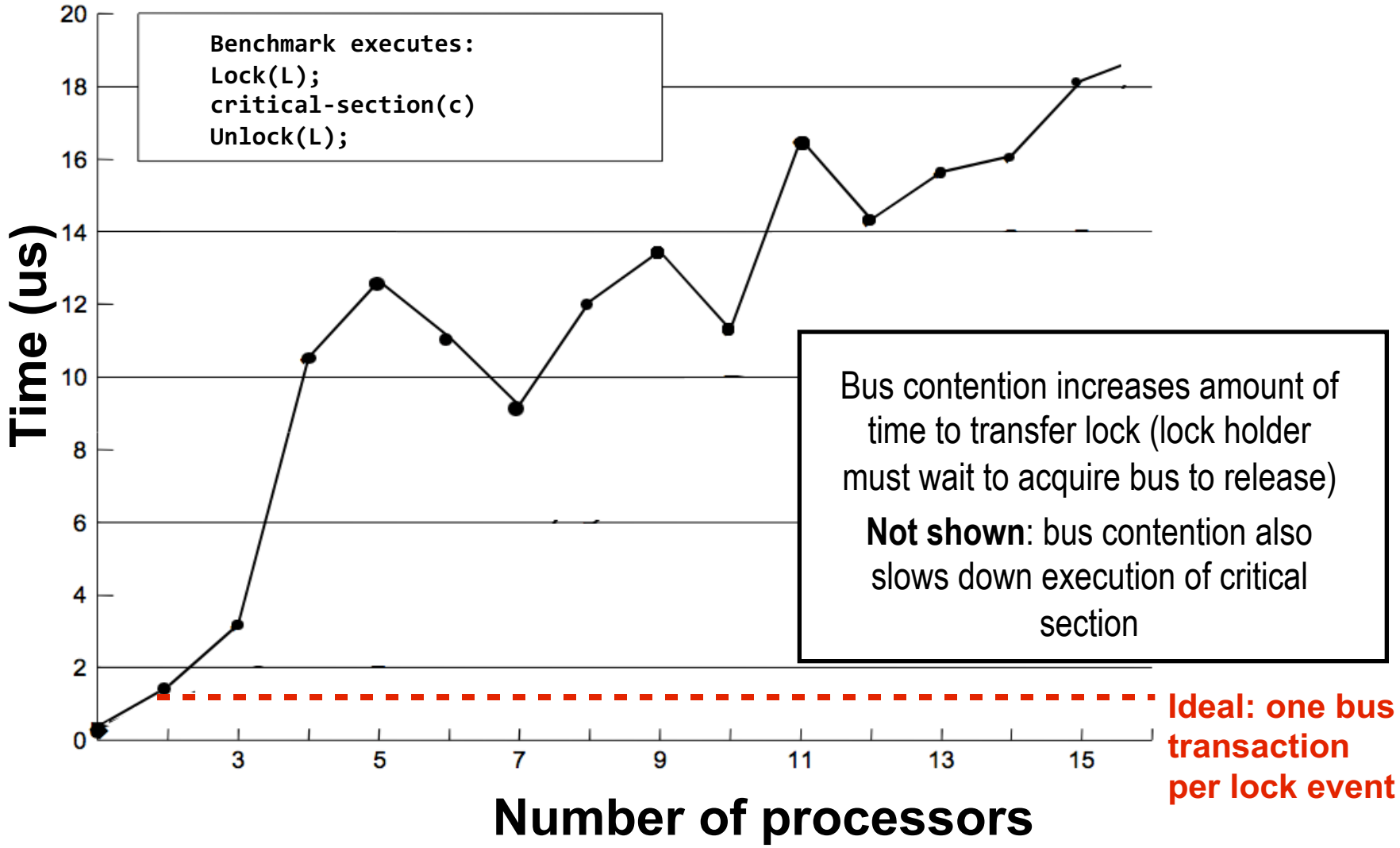
Reminder: Desirable Lock Characteristics

- Low latency
  - Processors should be able to acquire free locks quickly
- Low traffic
  - Waiting for lock should generate little/no traffic
  - A busy lock should be handed off between processors with as little traffic as possible
- Scalability
  - Latency/traffic should scale reasonably with number of processors
- Low storage cost
- Fairness
  - Avoid starvation or substantial unfairness
  - Ideal: processors should acquire lock in order they request access

TS Lock: **Excessive** Coherence Traffic



Test-and-Set Lock's Performance



Summary

- Desirable Lock Characteristics
- Basic TS lock hardware primitives
- TS lock's lack of scalability due to contention
  - Contention while spinning
  - Contention when lock is released