Synchronization
+
Cache Coherence

Chris Rossbach + Calvin Lin

CS380p

Today

- Reminder: Homework & Reading
- Foundations
  - Synchronization Implementation
  - Cache coherence

Review: Schedules/Interleavings

Model of concurrent execution

- Interleave statements from each thread into a single thread
- If **any** interleaving yields incorrect results, synchronization is needed

Thread 1

```
tmp1 = X;
tmp1 = tmp1 + 1;
X = tmp1;
```

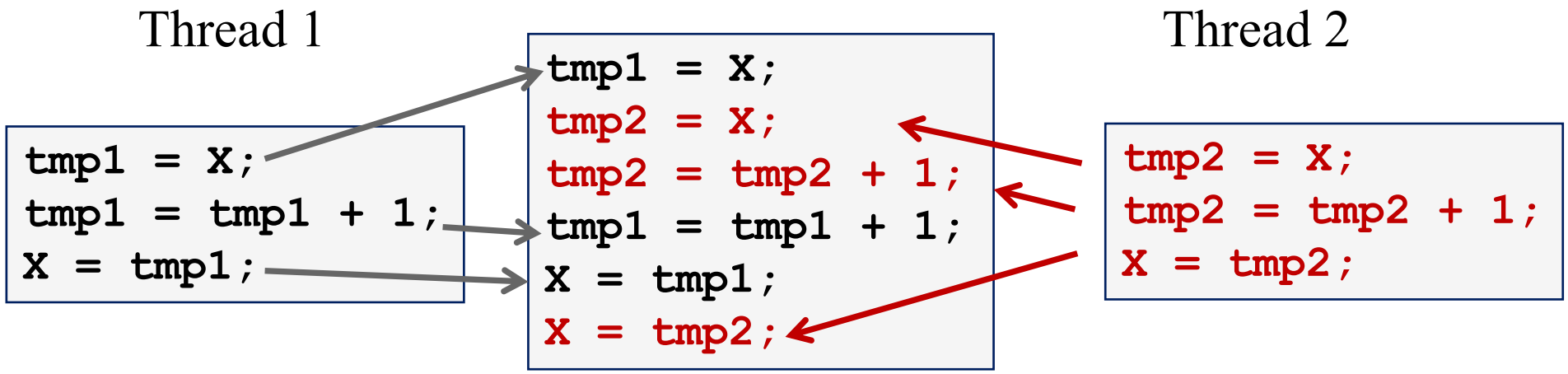
Thread 2

```
tmp2 = X;
tmp2 = tmp2 + 1;
X = tmp2;
```

Review: Schedules/Interleavings

Model of concurrent execution

- Interleave statements from each thread into a single thread
- If **any** interleaving yields incorrect results, synchronization is needed



If X==0 initially, X == 1 at the end. WRONG result!

Locks implement Mutual Exclusion

```
void increment() {
    lock.acquire();
    int temp = X;
    temp = temp + 1;
    X = temp;
    lock.release();
}
```

Mutual exclusion ensures only safe interleavings

- *But it limits concurrency, and hence scalability/performance*

Implementing Locks

```
int lock_value = 0;
int* lock = &lock_value;
```

Implementing Locks

```
int lock_value = 0;
int* lock = &lock_value;
```

```
lock::acquire() {
  while (*lock == 1)
     ; //spin
  *lock = 1;
}
```

Implementing Locks

```
int lock_value = 0;
int* lock = &lock_value;
```

```
lock::acquire() {
  while (*lock == 1)
     ; //spin
  *lock = 1;
}
```

```
lock::release() {
    *lock = 0;
}
```

Implementing Locks

```
int lock_value = 0;
int* lock = &lock_value;
```

```
lock::acquire() {
  while (*lock == 1)
     ; //spin
  *lock = 1;
}
```

```
lock::release() {
    *lock = 0;
}
```

What are the problem(s) with this?

- A. CPU usage
- B. Memory usage
- C. lock::acquire() latency
- D. Memory bus usage
- E. Does not work

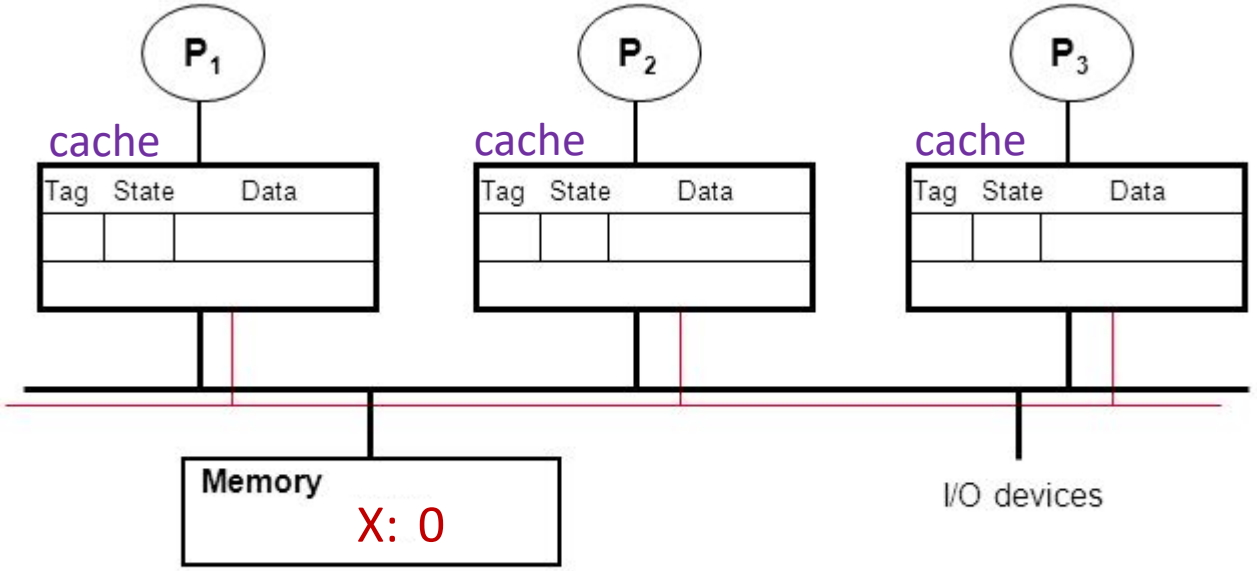
Multiprocessor Cache Coherence

$$F = ma$$

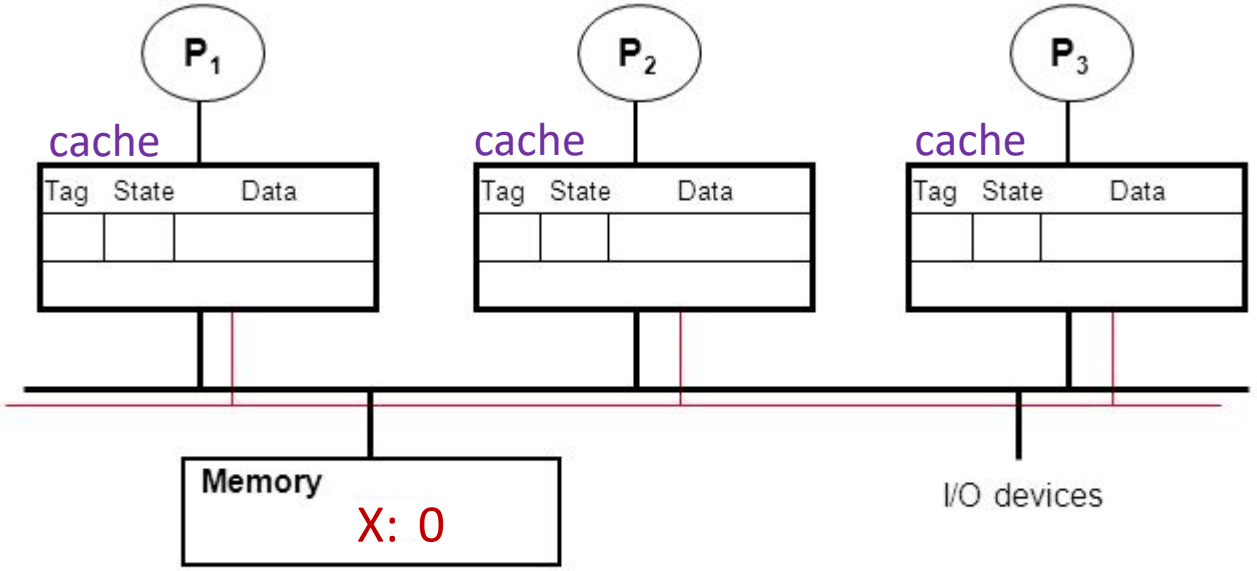
Multiprocessor Cache Coherence

Physics	Concurrency
F = ma	*~ coherence*

Multiprocessor Cache Coherence

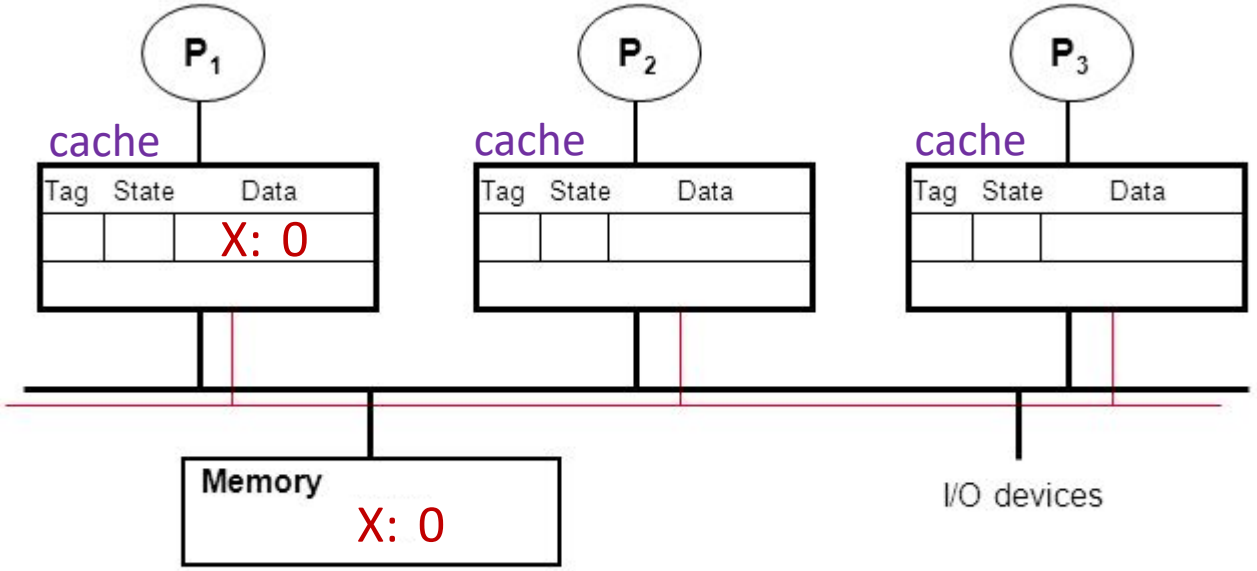


Multiprocessor Cache Coherence



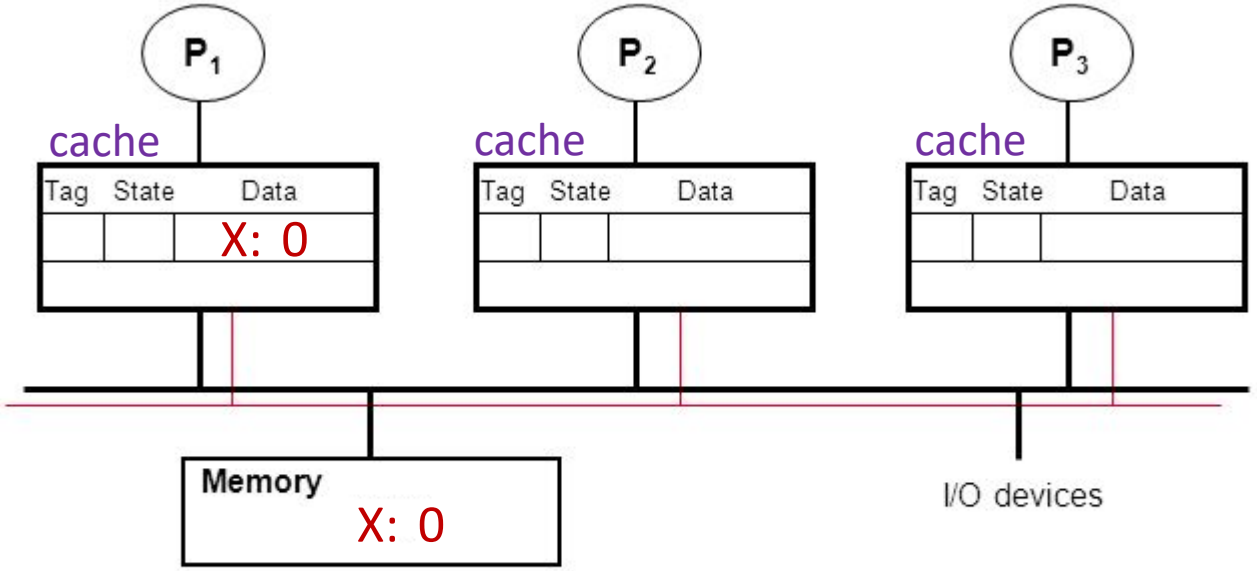
- P1: read X

Multiprocessor Cache Coherence



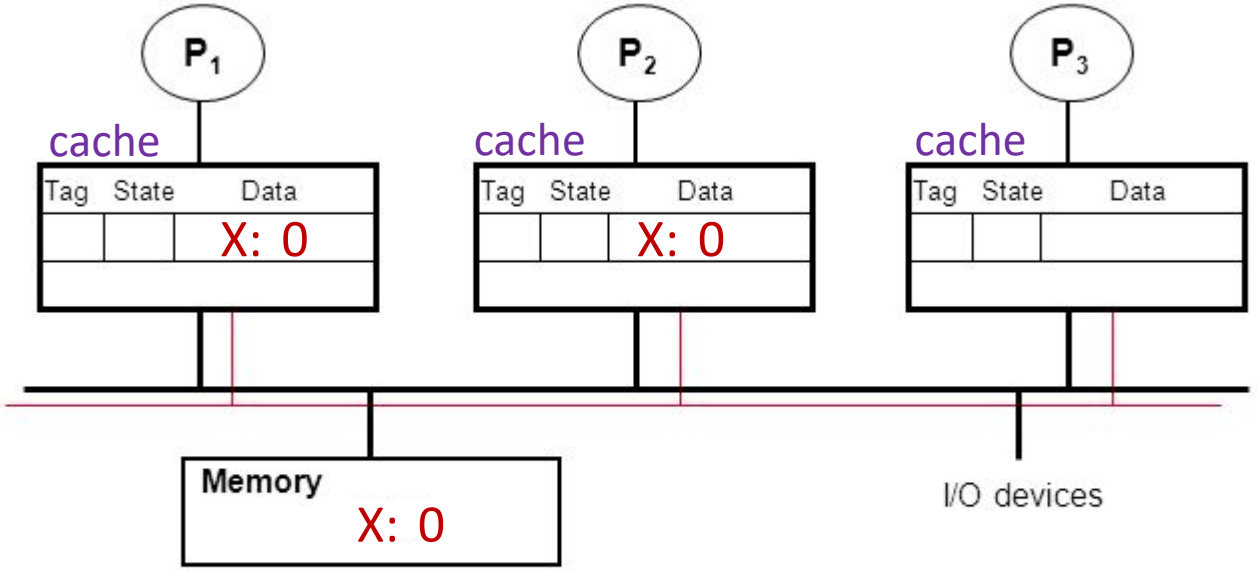
- P1: read X

Multiprocessor Cache Coherence



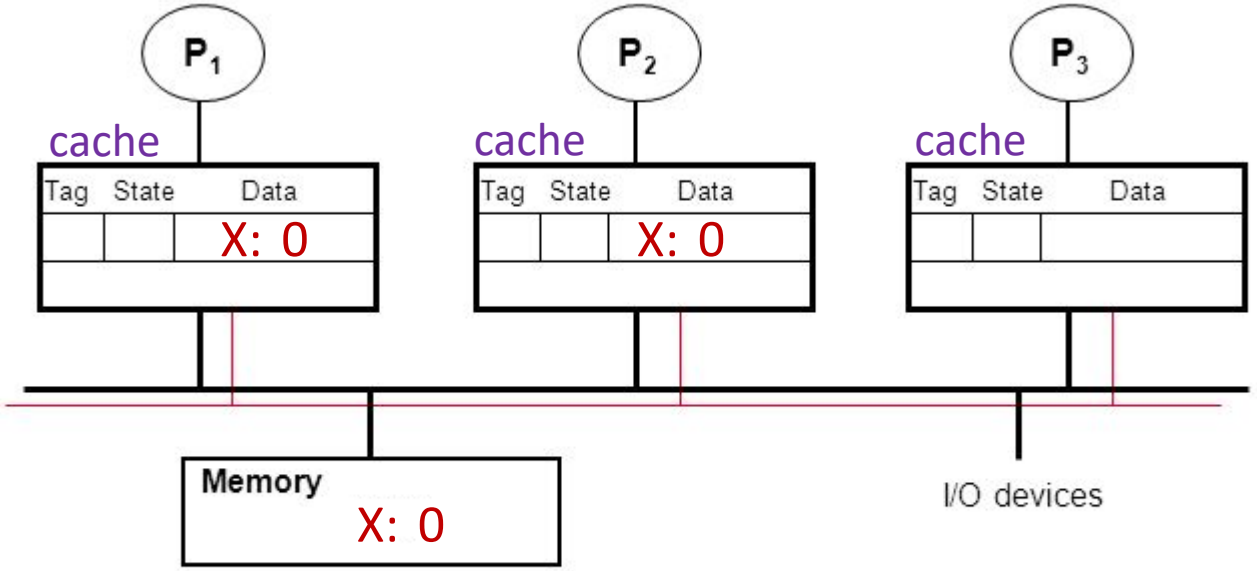
- P1: read X
- P2: read X

Multiprocessor Cache Coherence



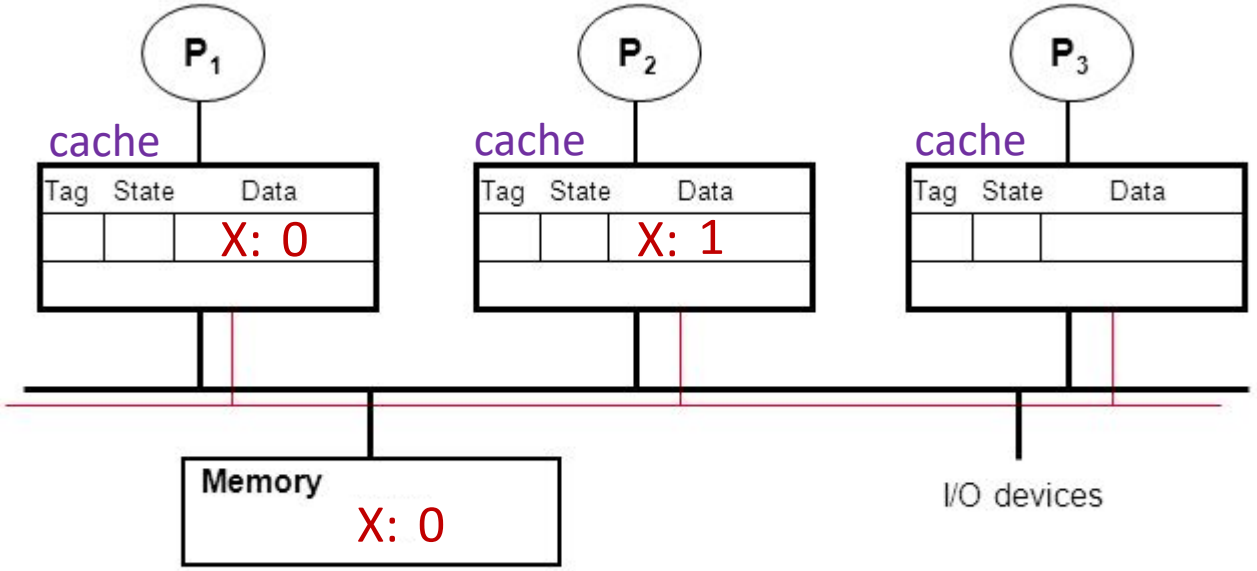
- P1: read X
- P2: read X

Multiprocessor Cache Coherence



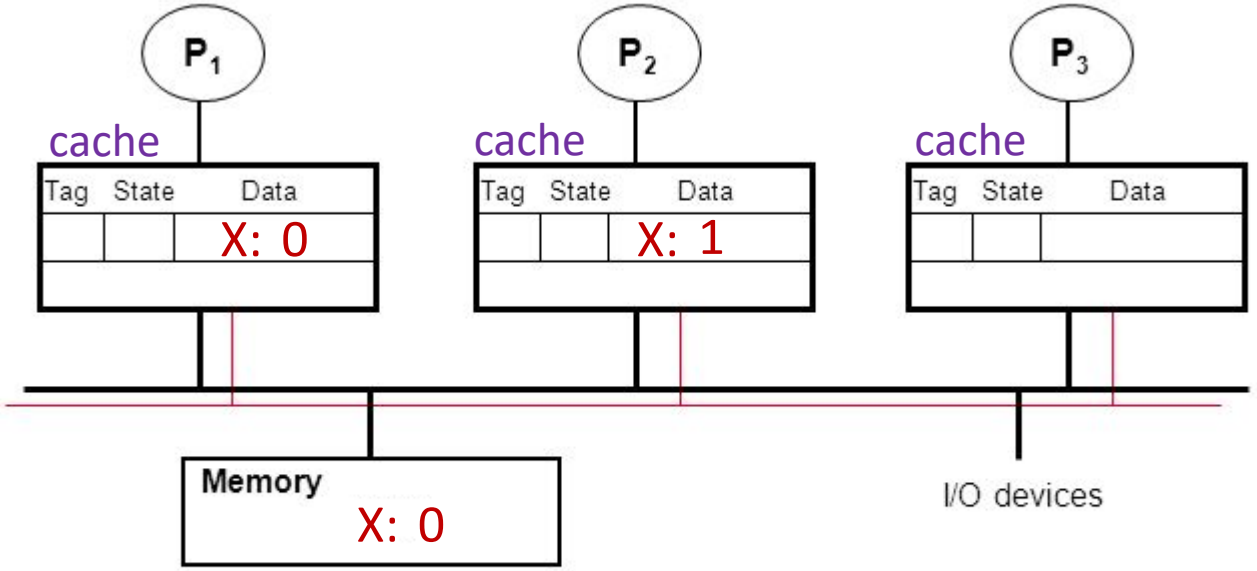
- P1: read X
- P2: read X
- P2: X++

Multiprocessor Cache Coherence



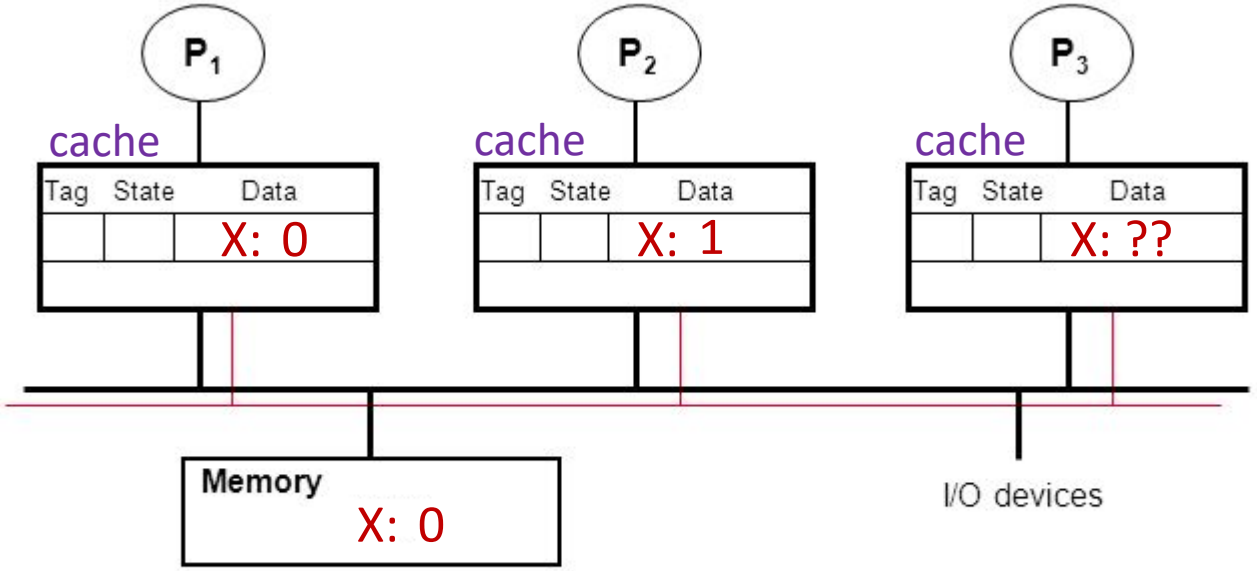
- P1: read X
- P2: read X
- P2: X++

Multiprocessor Cache Coherence



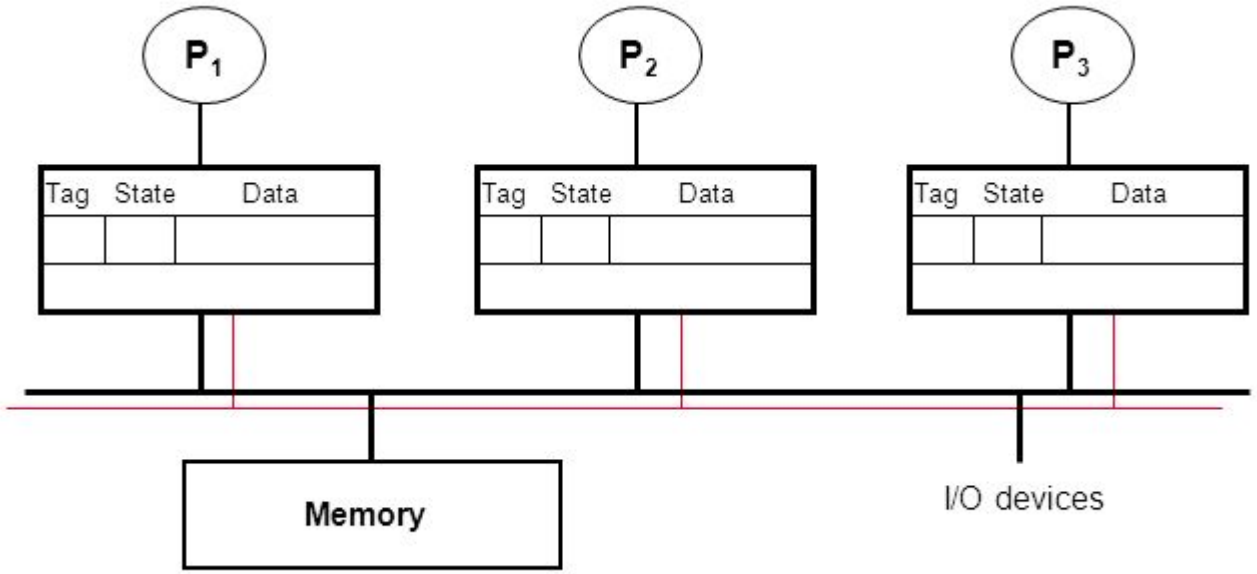
- P1: read X
- P2: read X
- P2: X++
- P3: read X

Multiprocessor Cache Coherence

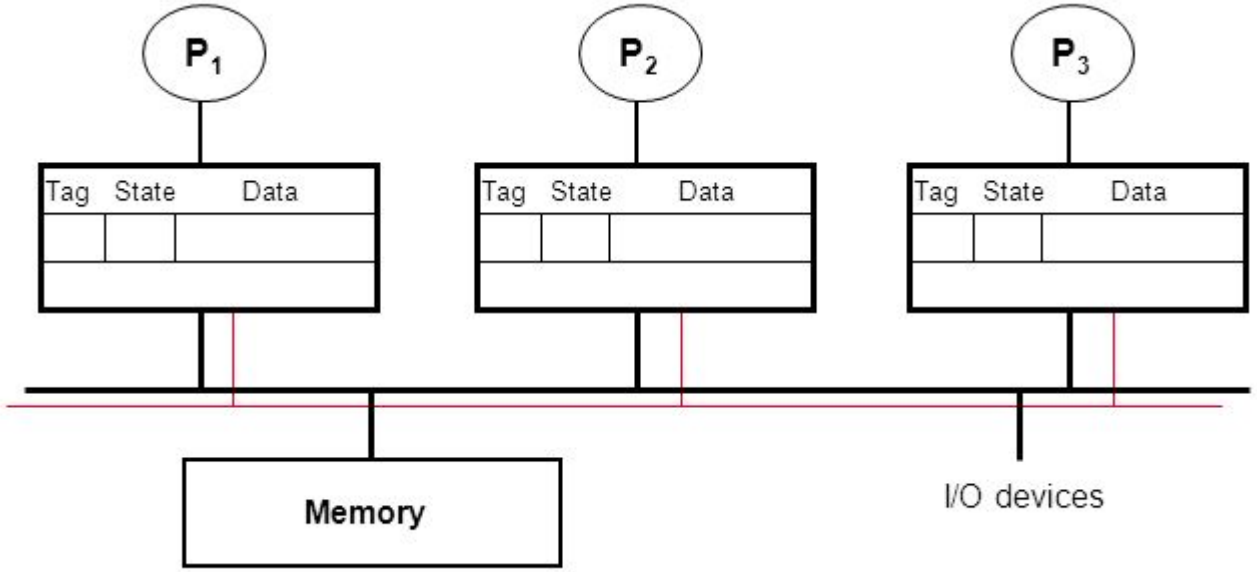


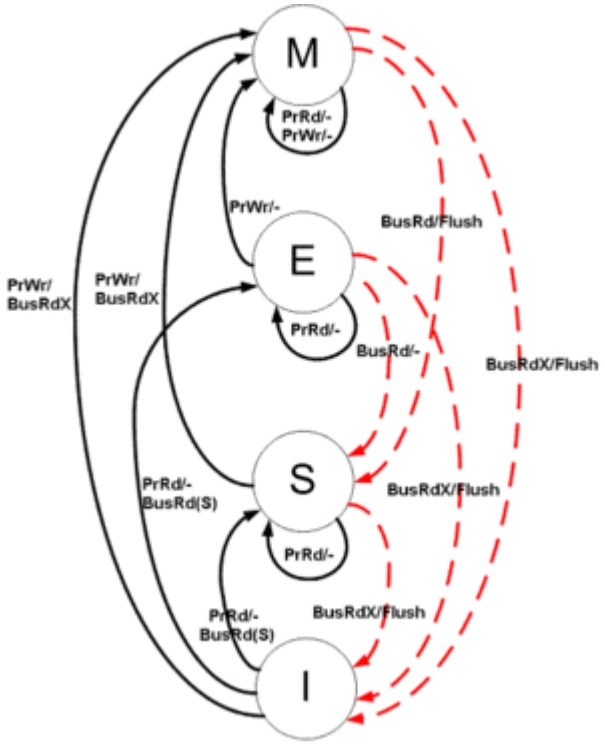
- P1: read X
- P2: read X
- P2: X++
- P3: read X

Multiprocessor Cache Coherence

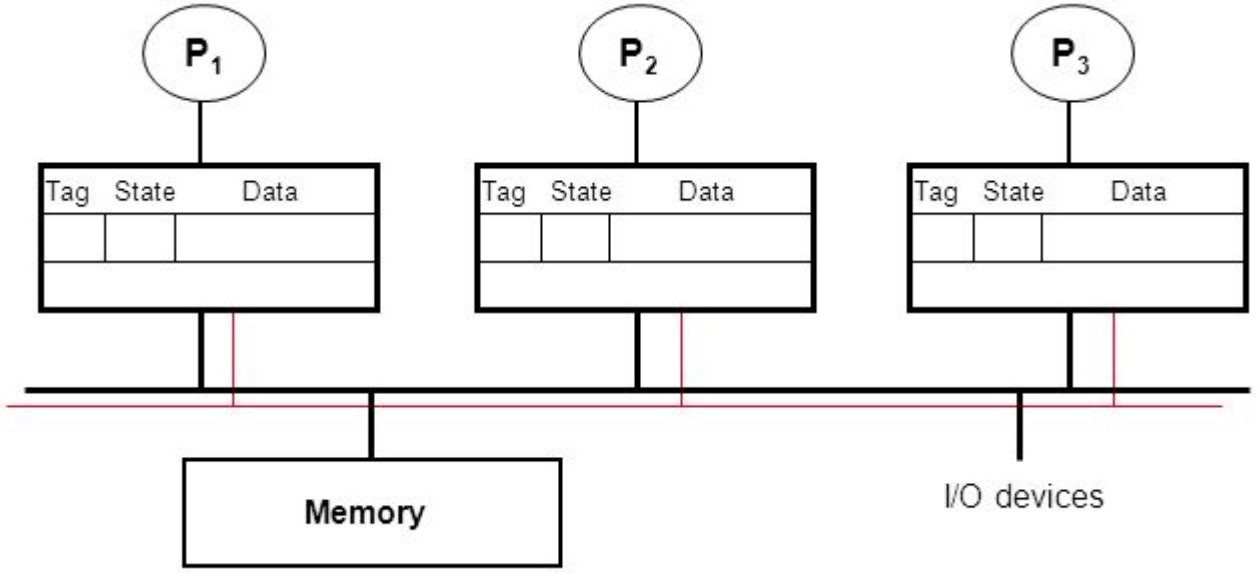


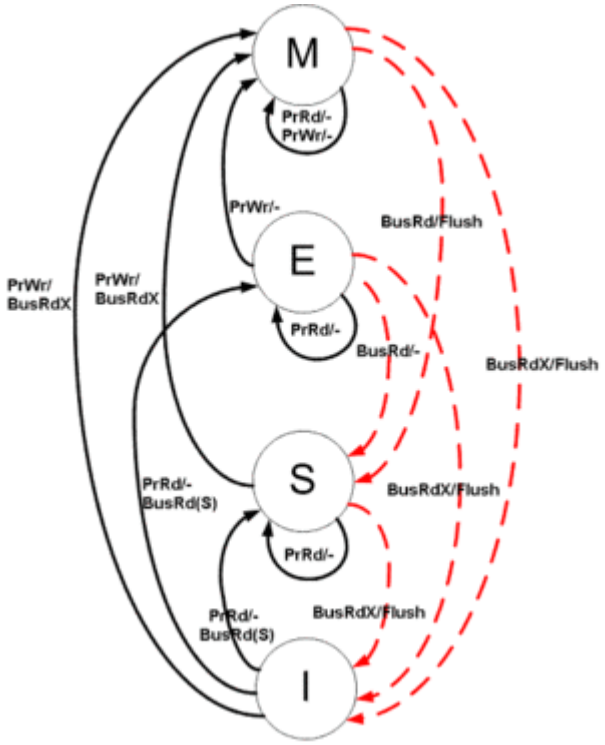
Multiprocessor Cache Coherence





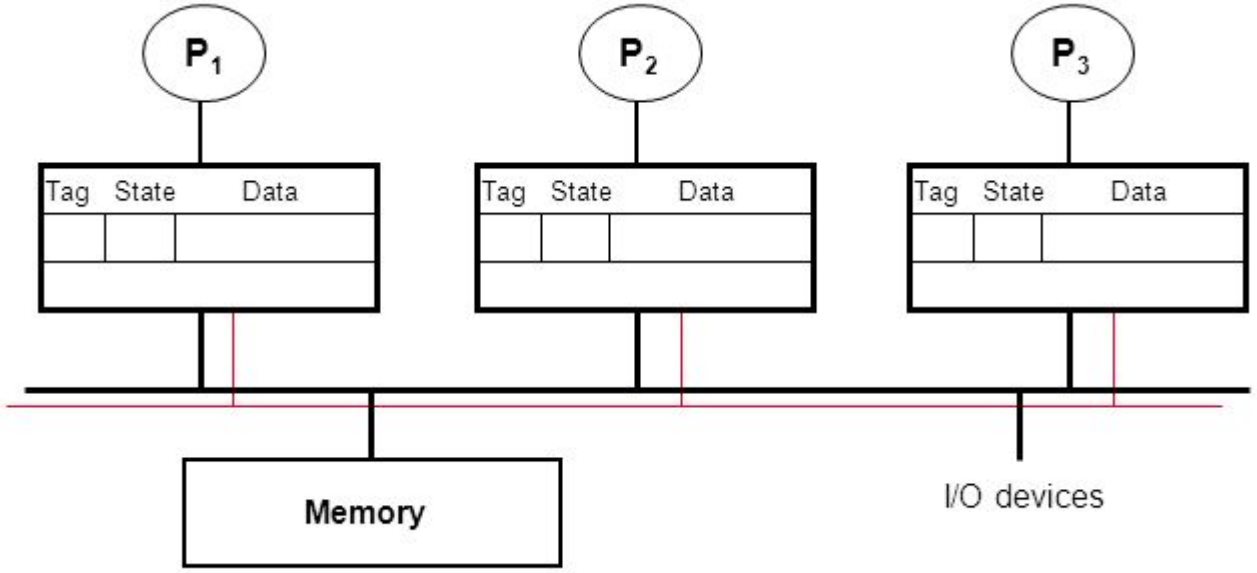
Multiprocessor Cache Coherence

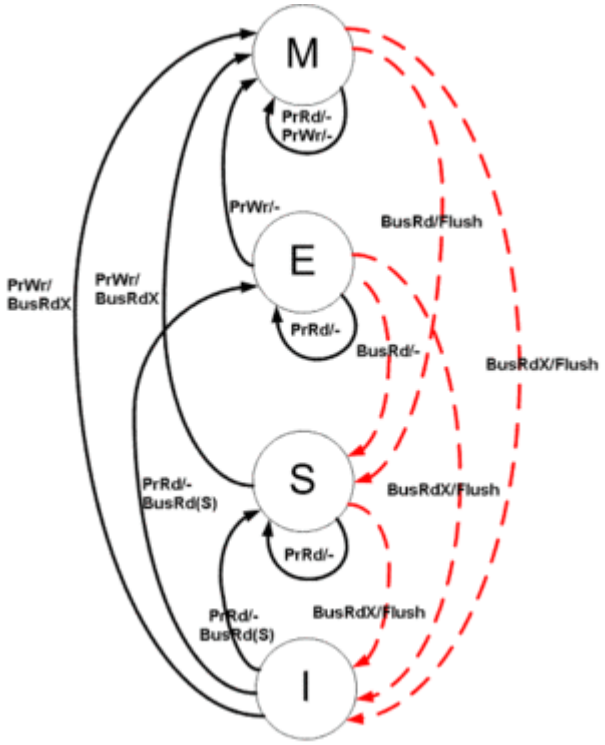




Each cache line has a state (M, E, S, I)

Multiprocessor Cache Coherence

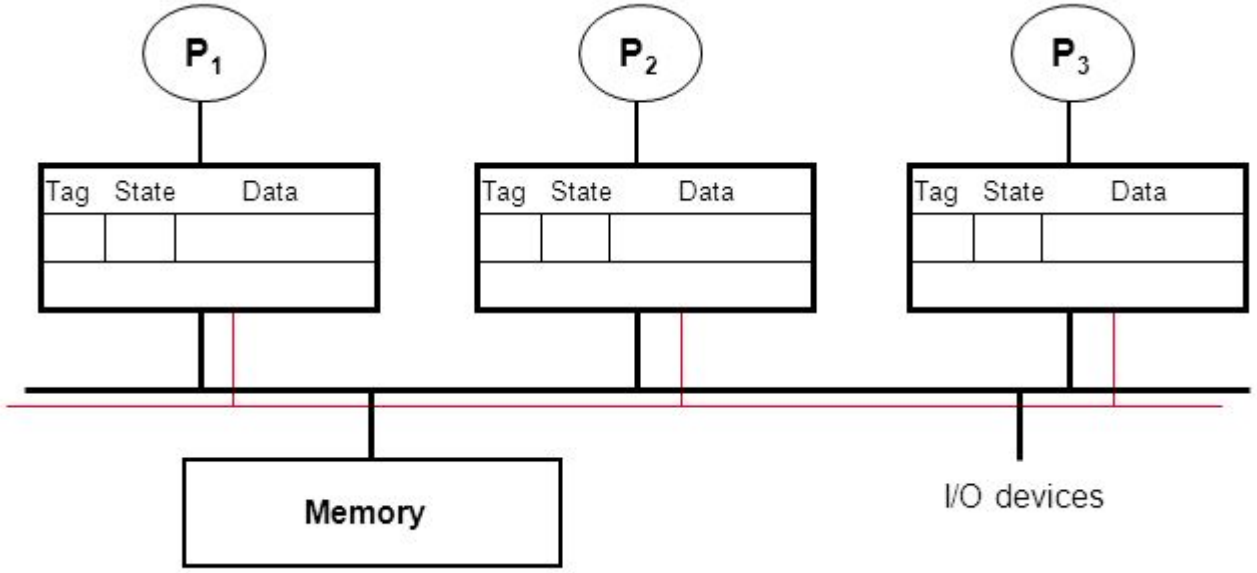


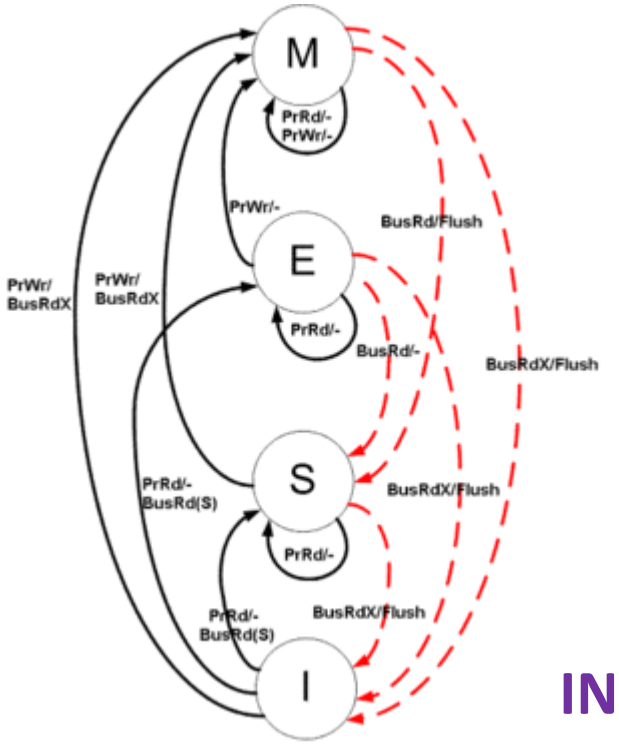


Each cache line has a state (M, E, S, I)

- Processors “snoop” bus to maintain states

Multiprocessor Cache Coherence



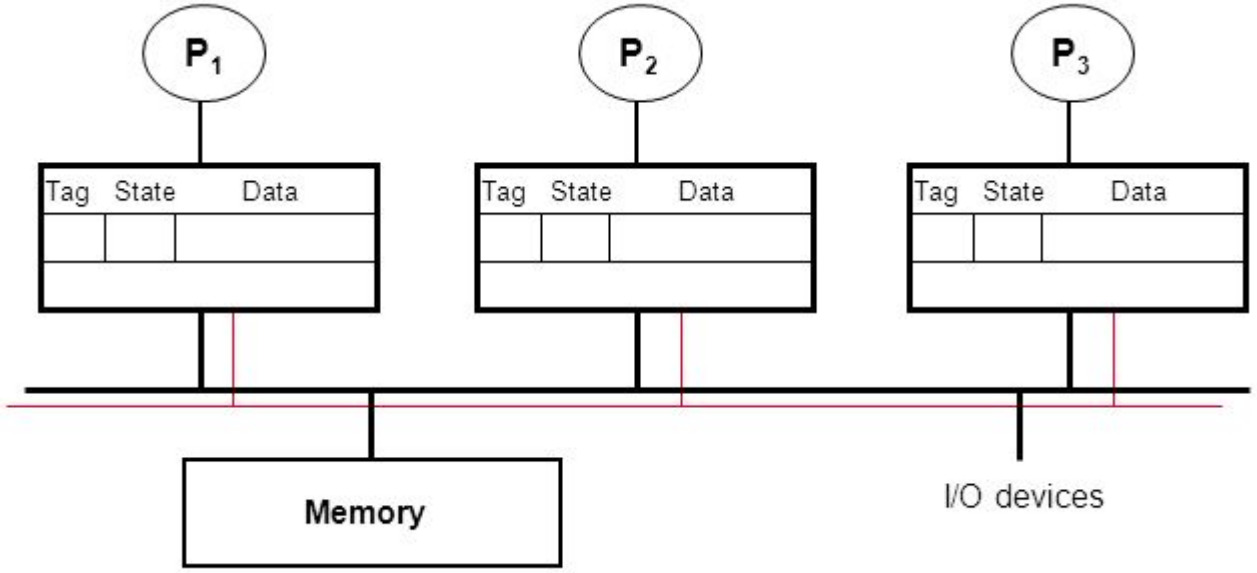


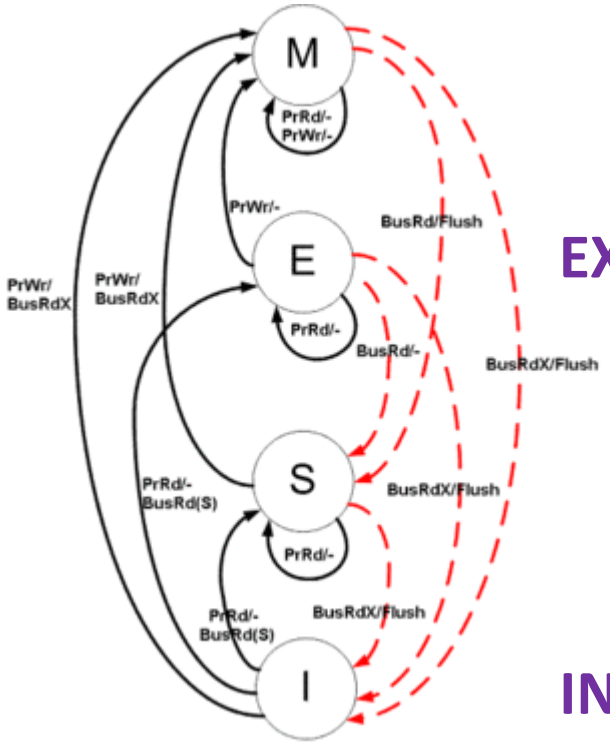
Each cache line has a state (M, E, S, I)

- Processors “snoop” bus to maintain states
- Initially → ‘I’ → Invalid

INVALID

Multiprocessor Cache Coherence





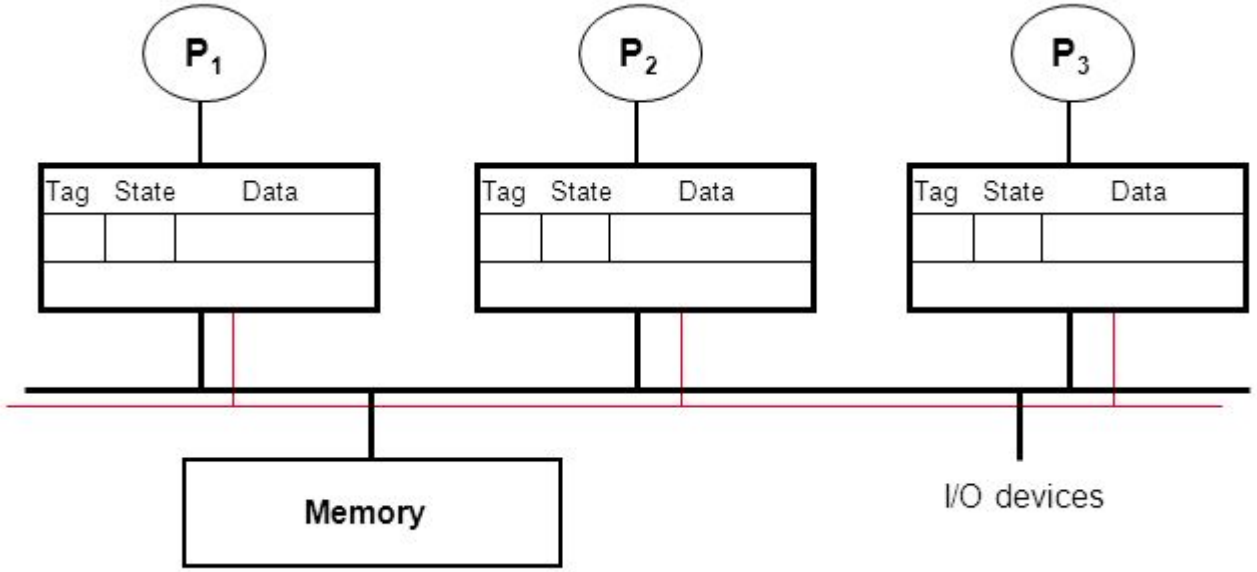
EXCLUSIVE

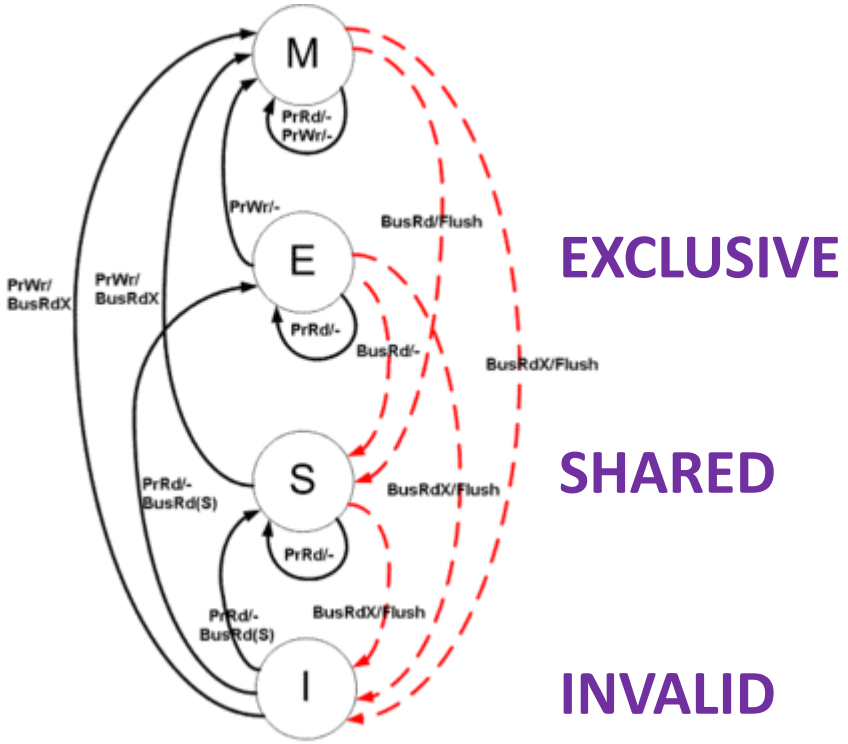
INVALID

Each cache line has a state (M, E, S, I)

- Processors “snoop” bus to maintain states
- Initially → ‘I’ → Invalid
- Read one → ‘E’ → exclusive

Multiprocessor Cache Coherence

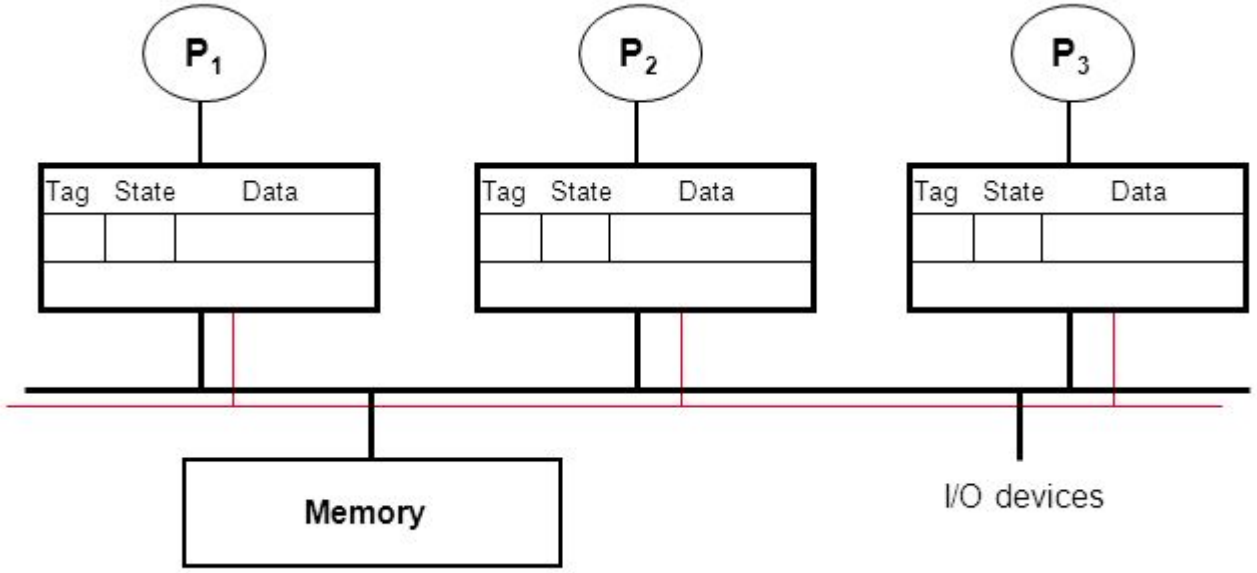


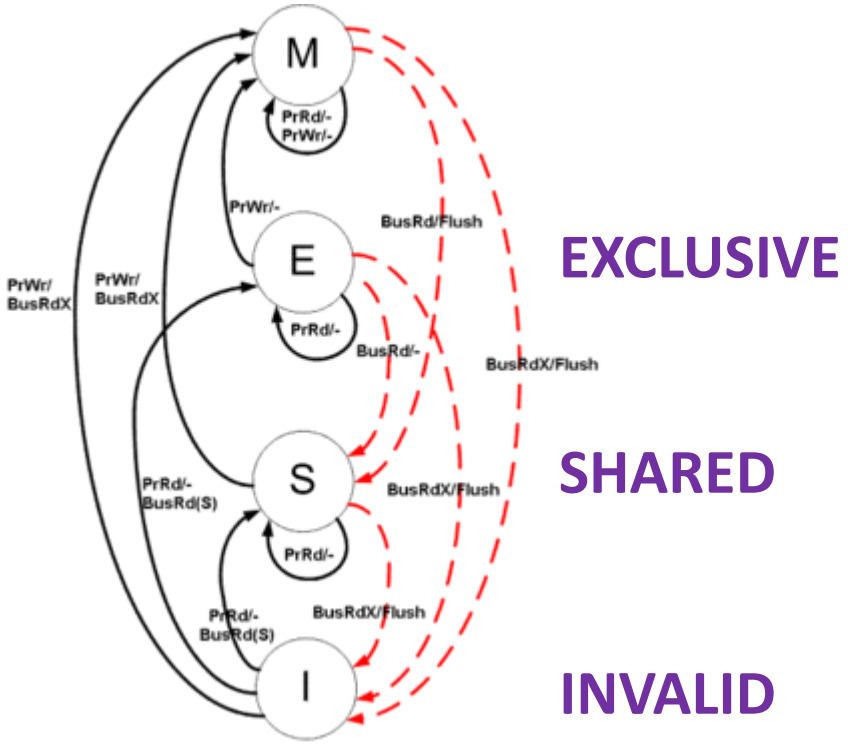


Each cache line has a state (M, E, S, I)

- Processors “snoop” bus to maintain states
- Initially → ‘I’ → Invalid
- Read one → ‘E’ → exclusive
- Reads → ‘S’ → multiple copies possible

Multiprocessor Cache Coherence

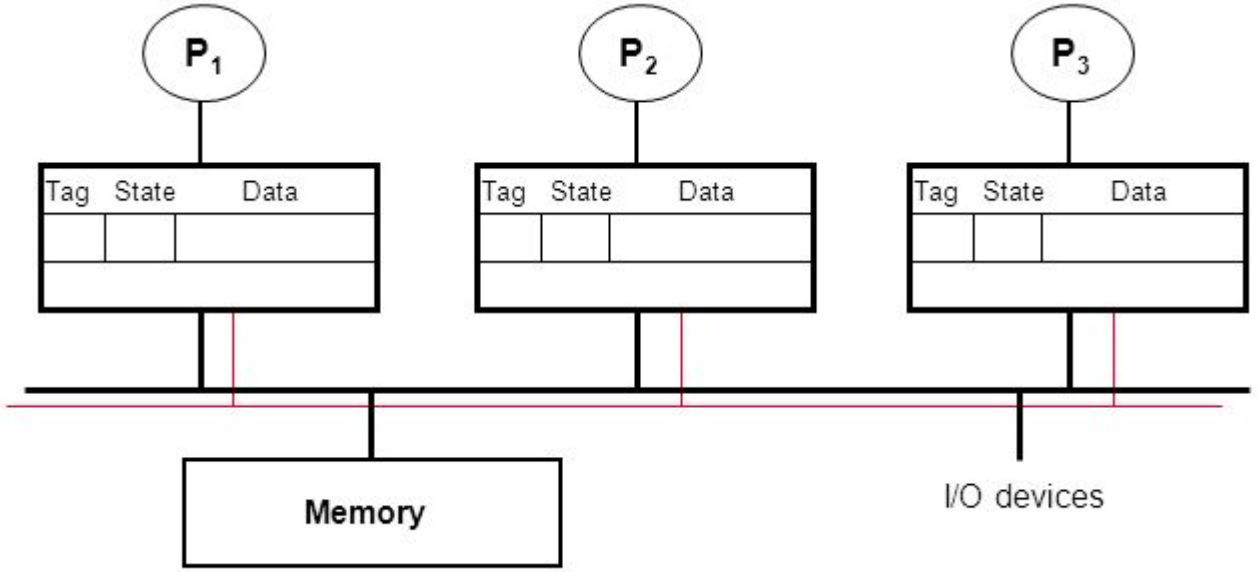


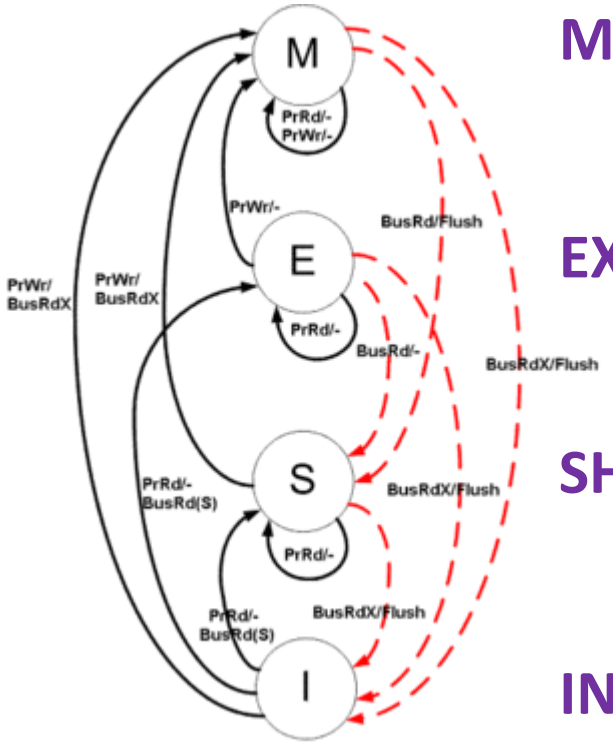


Each cache line has a state (M, E, S, I)

- Processors “snoop” bus to maintain states
- Initially → ‘I’ → Invalid
- Read one → ‘E’ → exclusive
- Reads → ‘S’ → multiple copies possible
- Write → ‘M’ → single copy → lots of cache coherence traffic

Multiprocessor Cache Coherence





MODIFIED

EXCLUSIVE

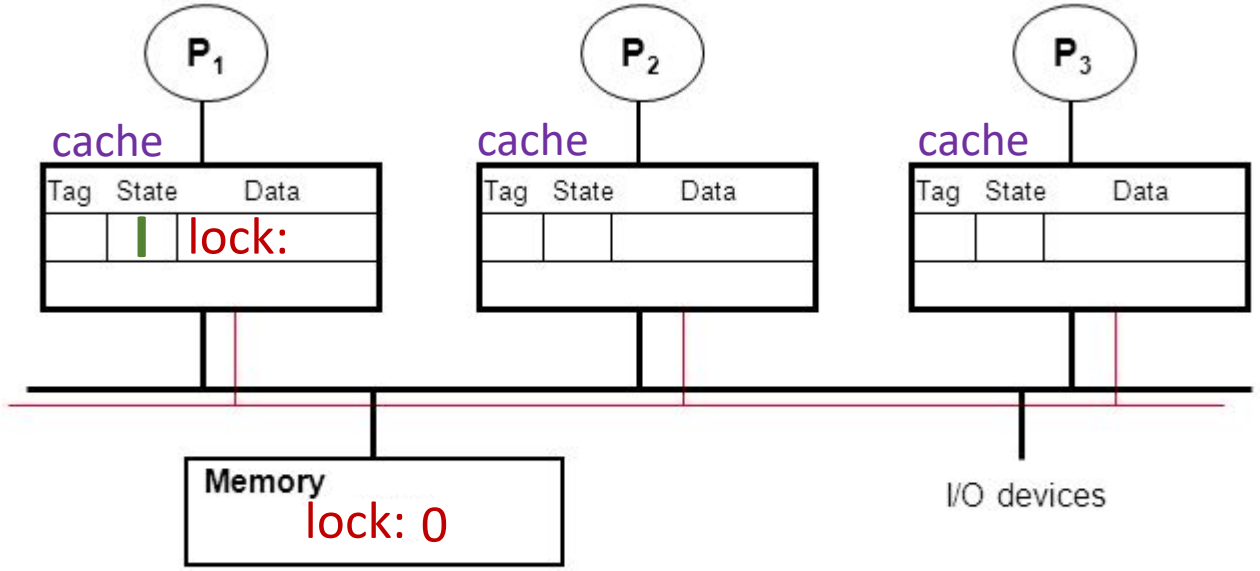
SHARED

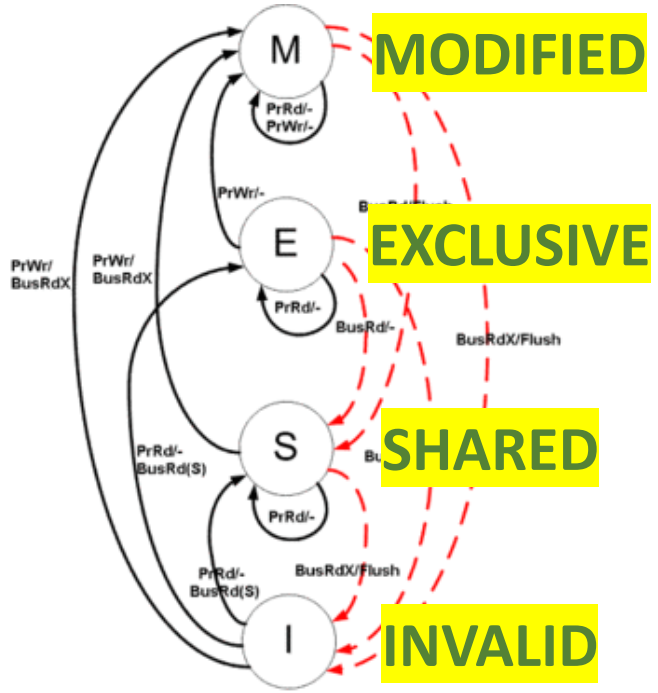
INVALID

Each cache line has a state (M, E, S, I)

- Processors “snoop” bus to maintain states
- Initially → ‘I’ → Invalid
- Read one → ‘E’ → exclusive
- Reads → ‘S’ → multiple copies possible
- Write → ‘M’ → single copy → lots of cache coherence traffic

Cache Coherence: single-thread

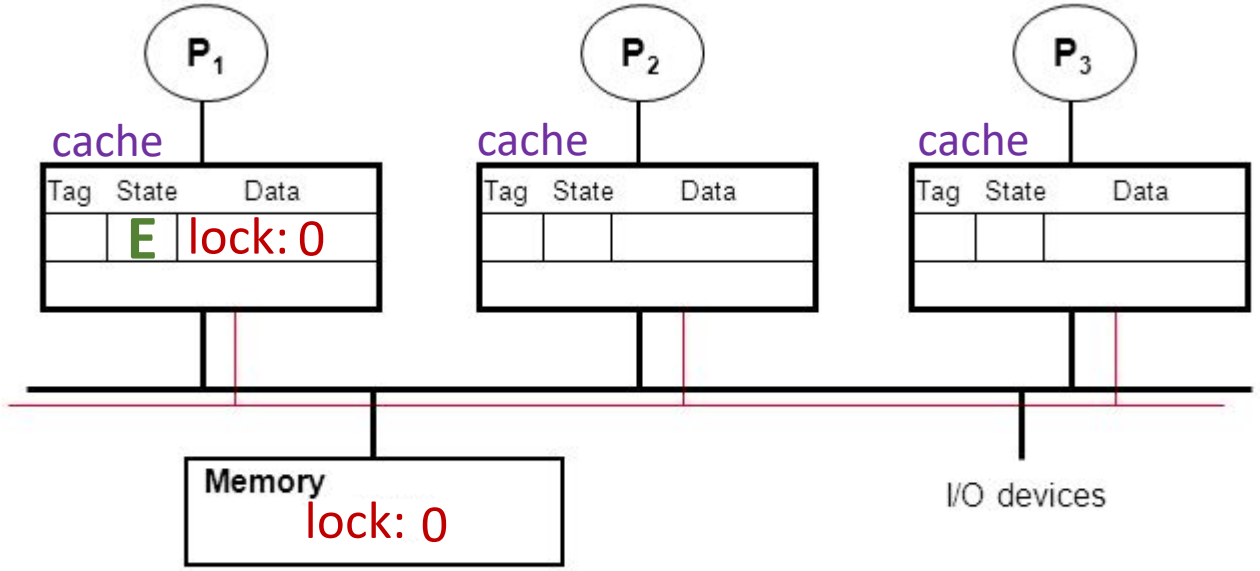


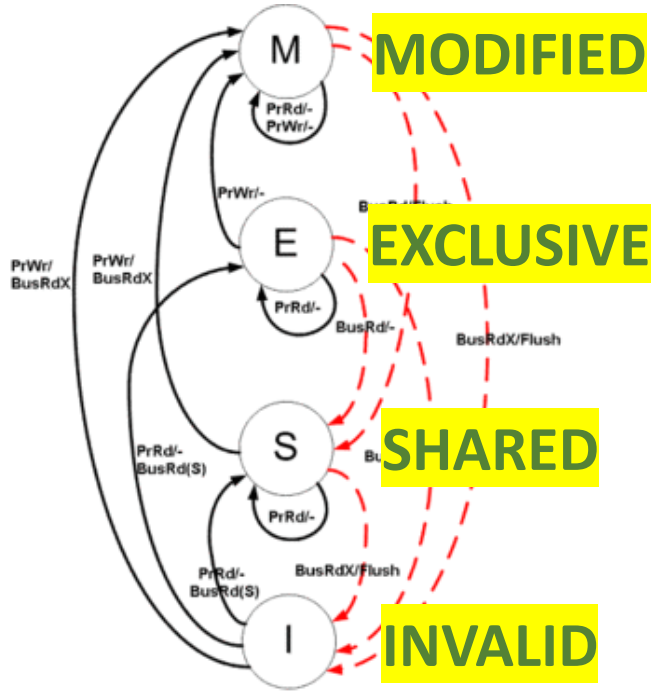


P1

```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
        test R0
        bnz try
        store lock, 1
}
```

Cache Coherence: single-thread

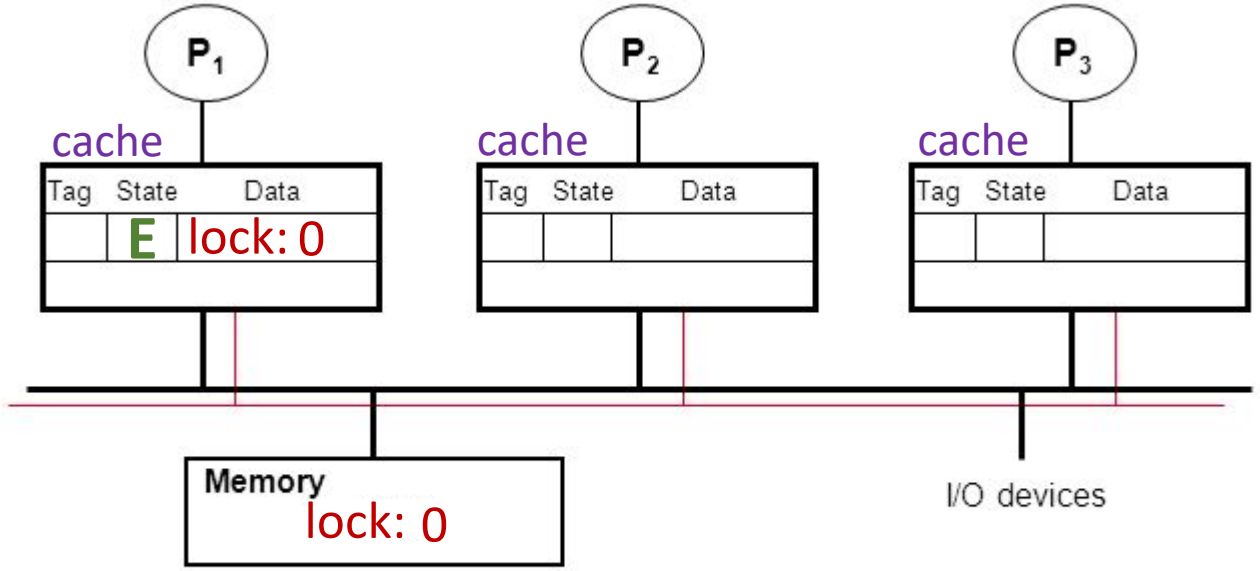


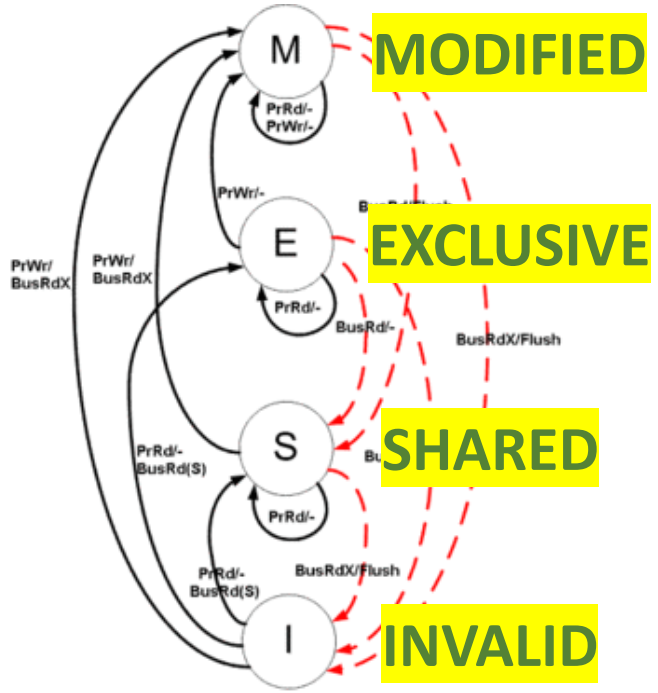


P1

```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
        test R0
        bnz try
        store lock, 1
}
```

Cache Coherence: single-thread

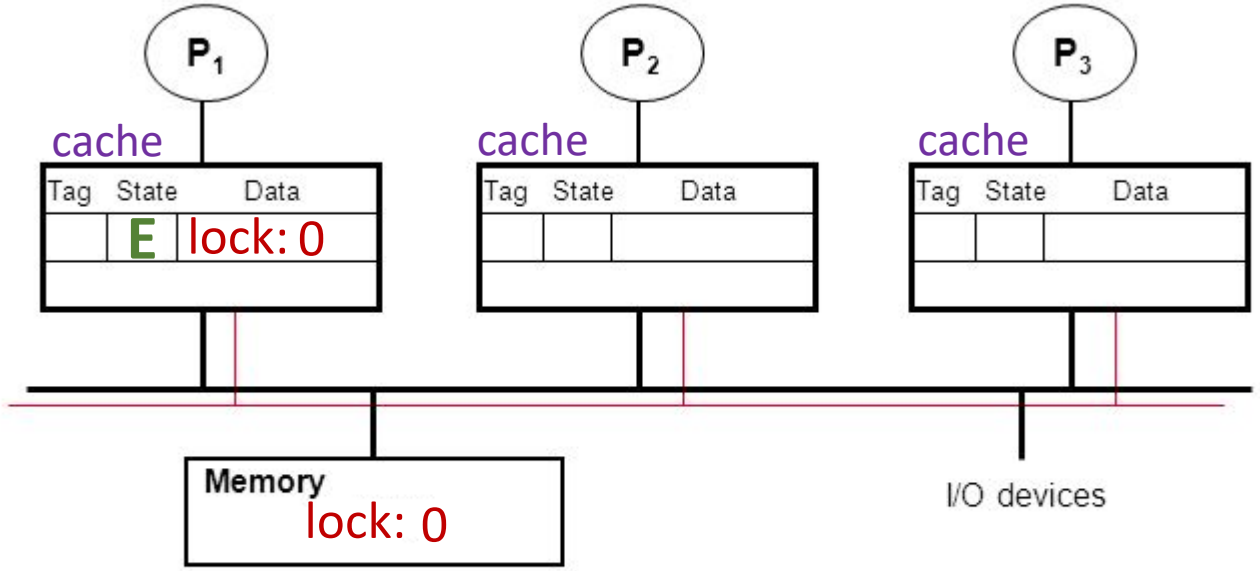


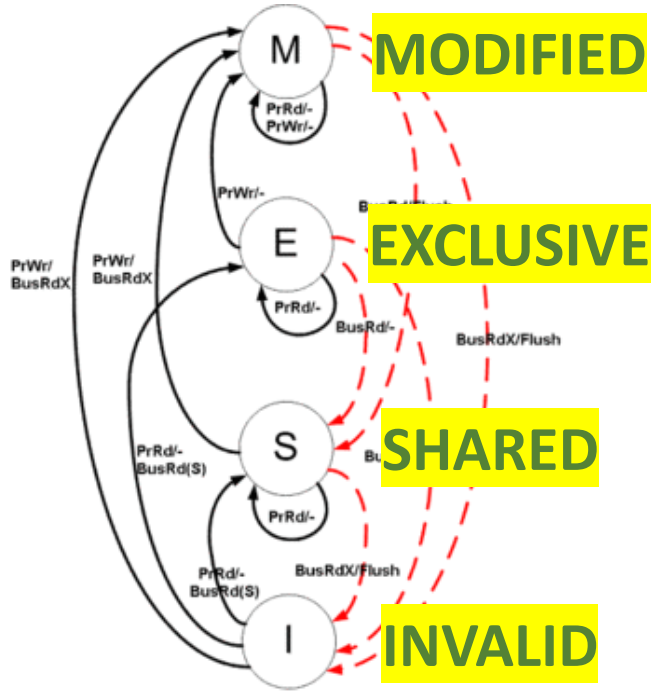


P1

```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
        test R0
        bnz try
        store lock, 1
}
```

Cache Coherence: single-thread

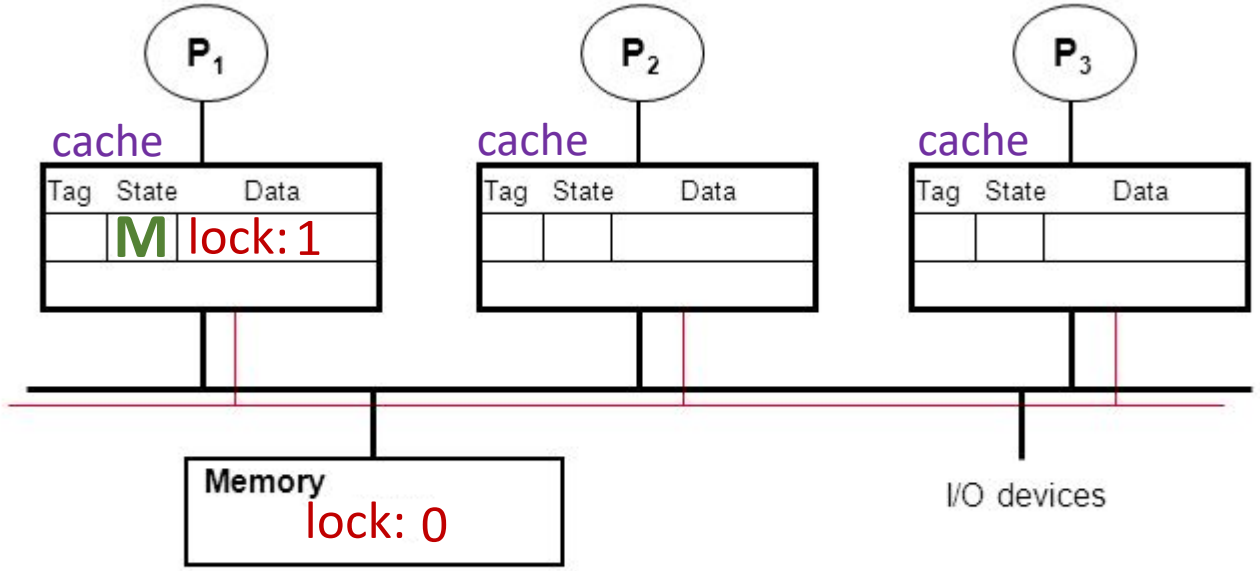


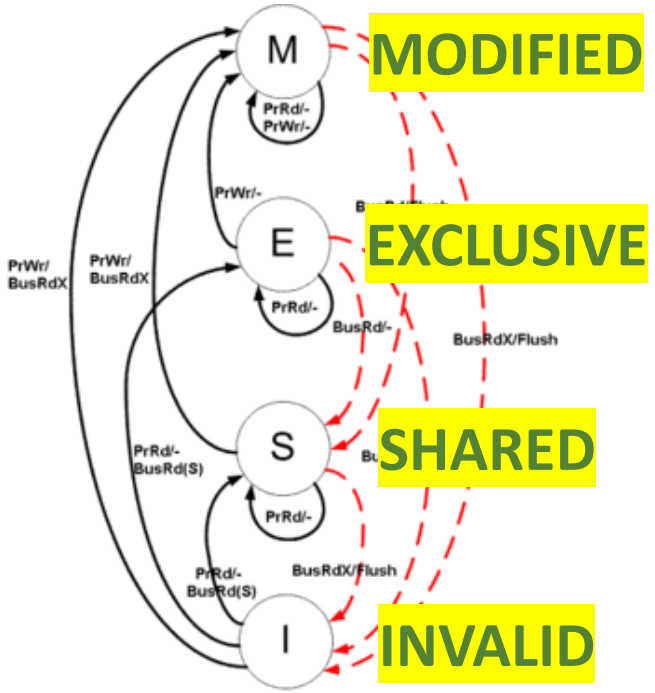


P1

```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
        test R0
        bnz try
        store lock, 1
}
```

Cache Coherence: single-thread

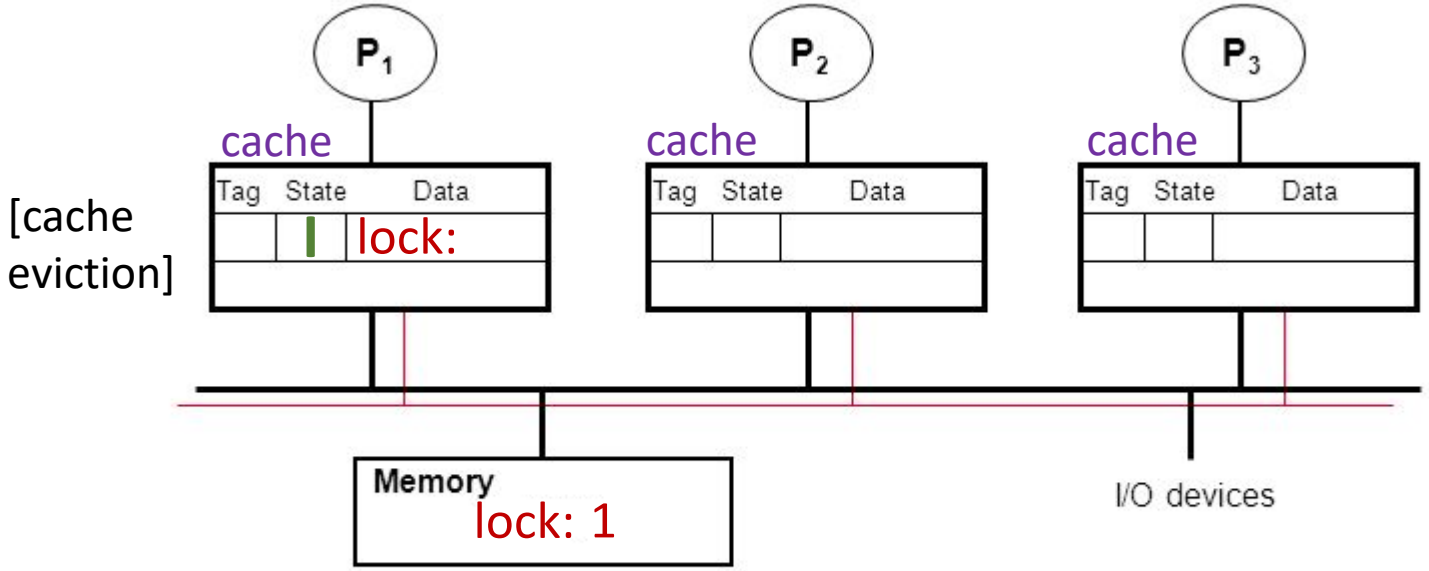


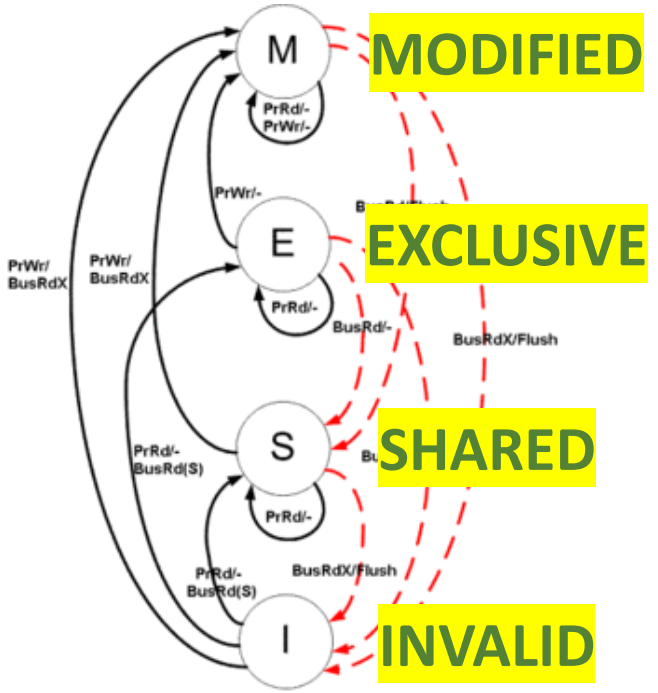


P1

```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
        test R0
        bnz try
        store lock, 1
}
```

Cache Coherence: single-thread

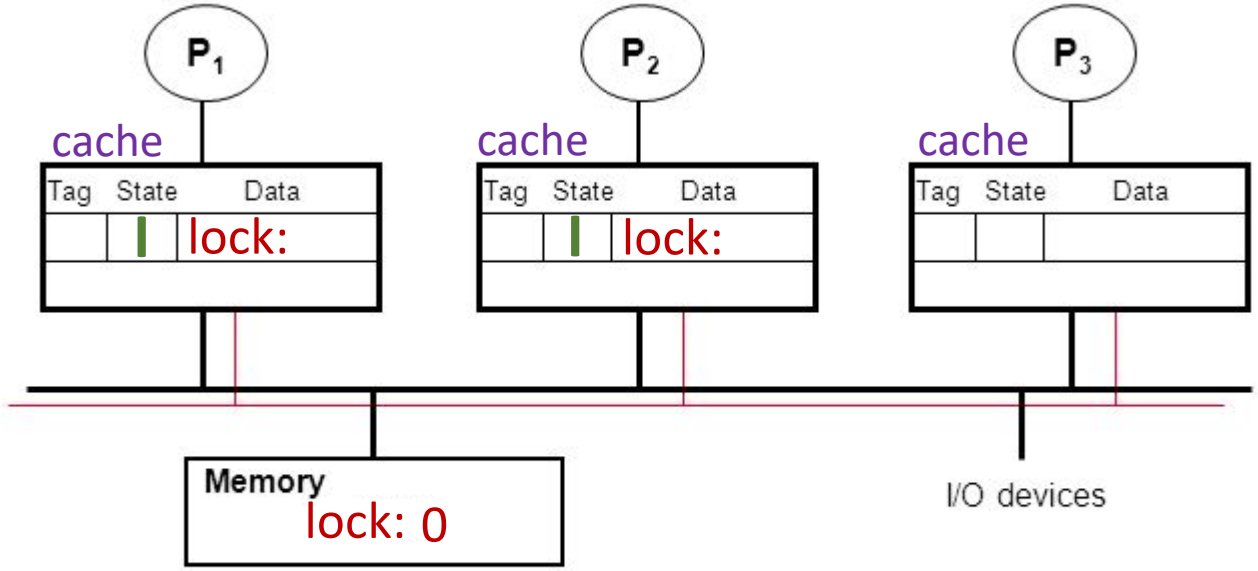


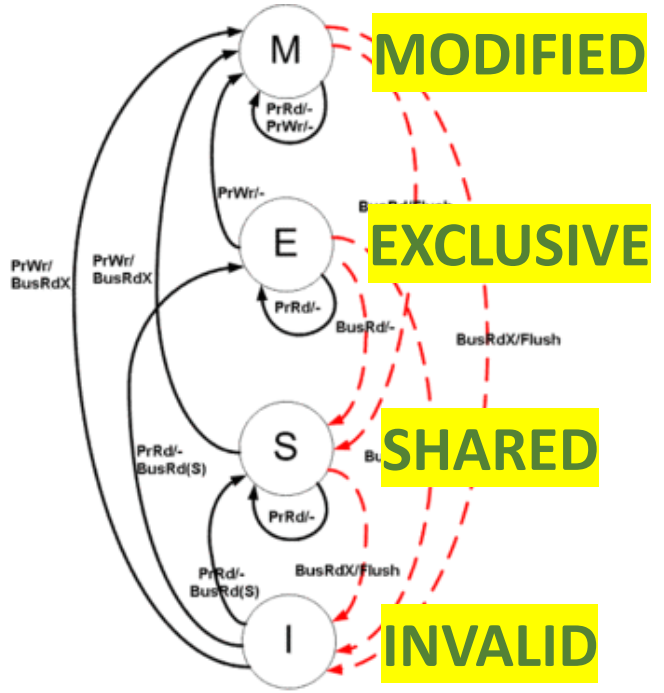


P1

```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
        test R0
        bnz try
        store lock, 1
}
```

Cache Coherence Action Zone





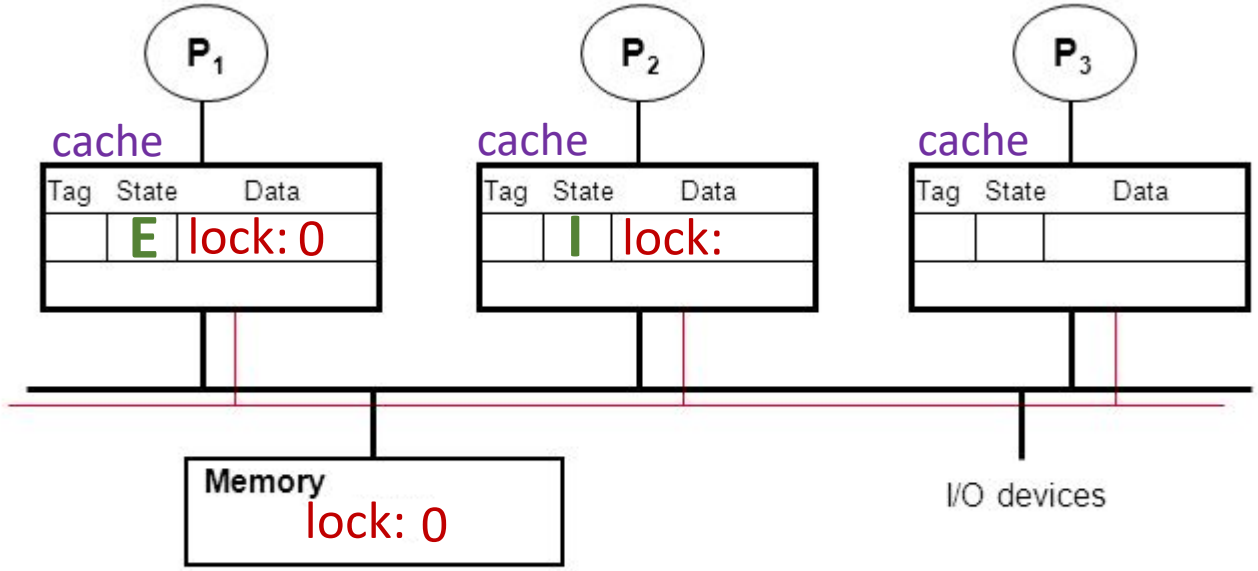
P1

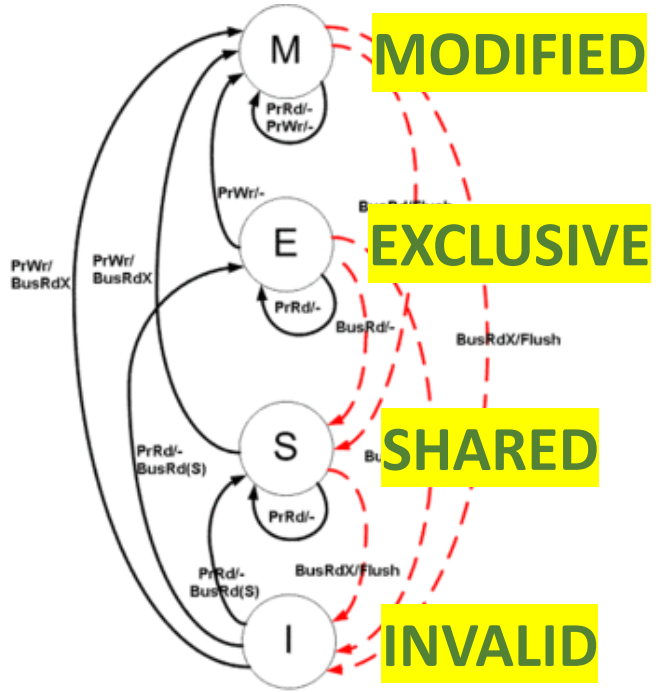
```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
        test R0
        bnz try
        store lock, 1
}
```

P2

```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
        test R0
        bnz try
        store lock, 1
}
```

Cache Coherence Action Zone





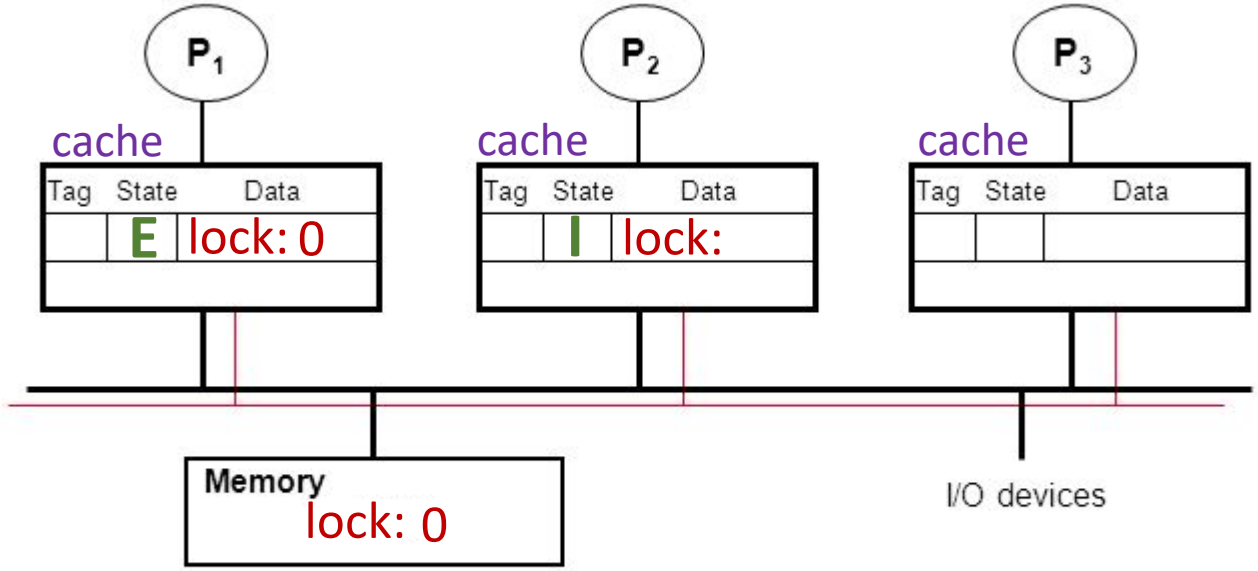
P1

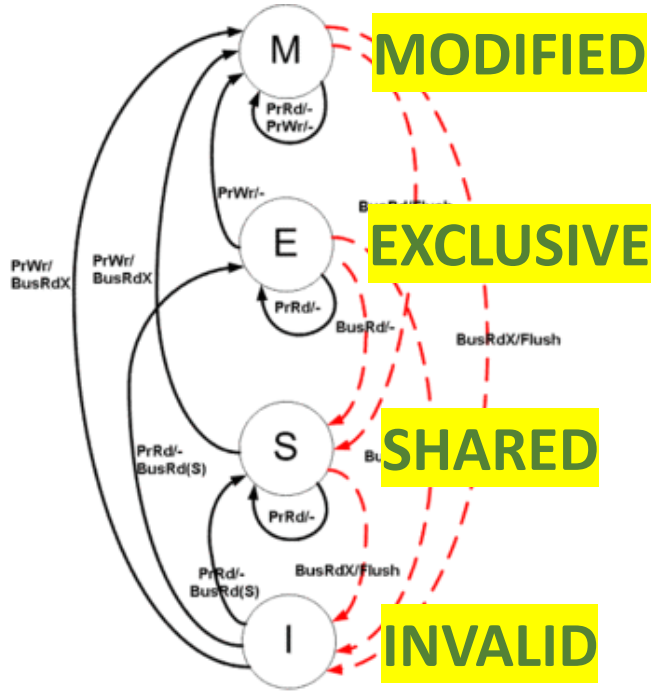
```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
        test R0
        bnz try
        store lock, 1
}
```

P2

```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
        test R0
        bnz try
        store lock, 1
}
```

Cache Coherence Action Zone





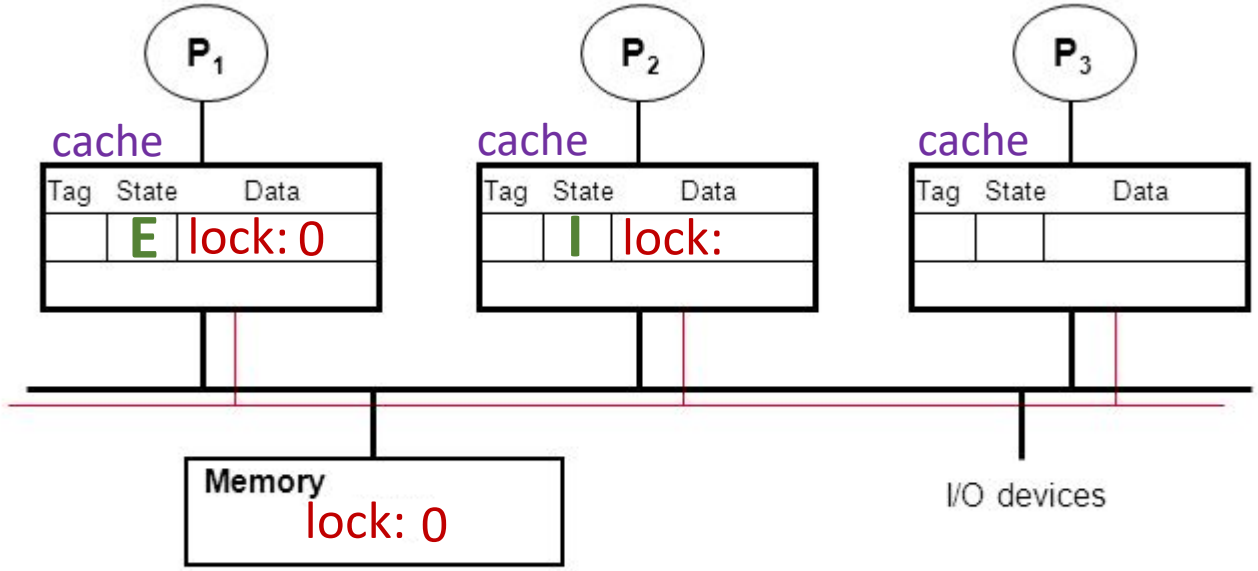
P1

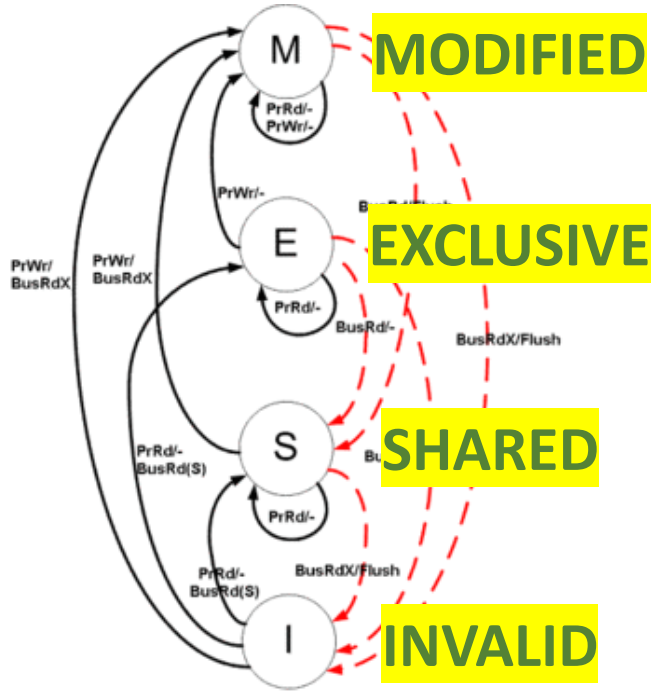
```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
        test R0
        bnz try
        store lock, 1
}
```

P2

```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
        test R0
        bnz try
        store lock, 1
}
```

Cache Coherence Action Zone





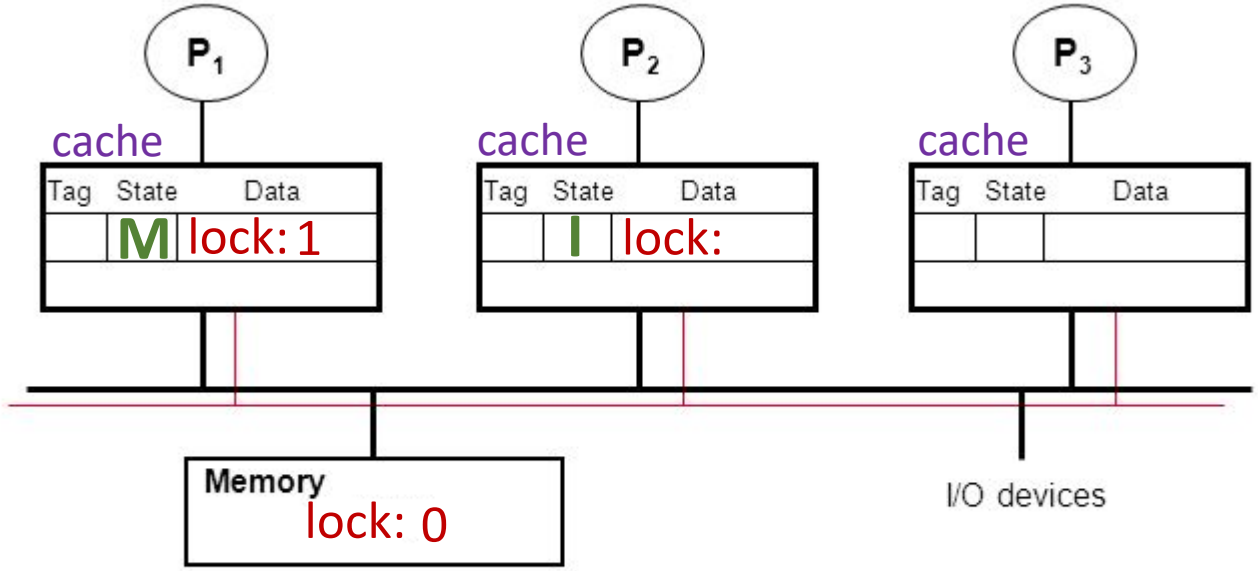
P1

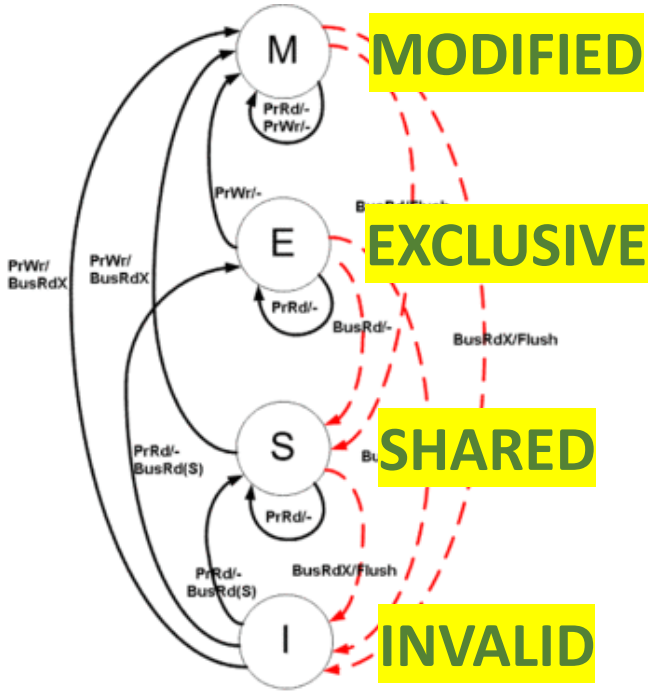
```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
        test R0
        bnz try
        store lock, 1
}
```

P2

```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
        test R0
        bnz try
        store lock, 1
}
```

Cache Coherence Action Zone





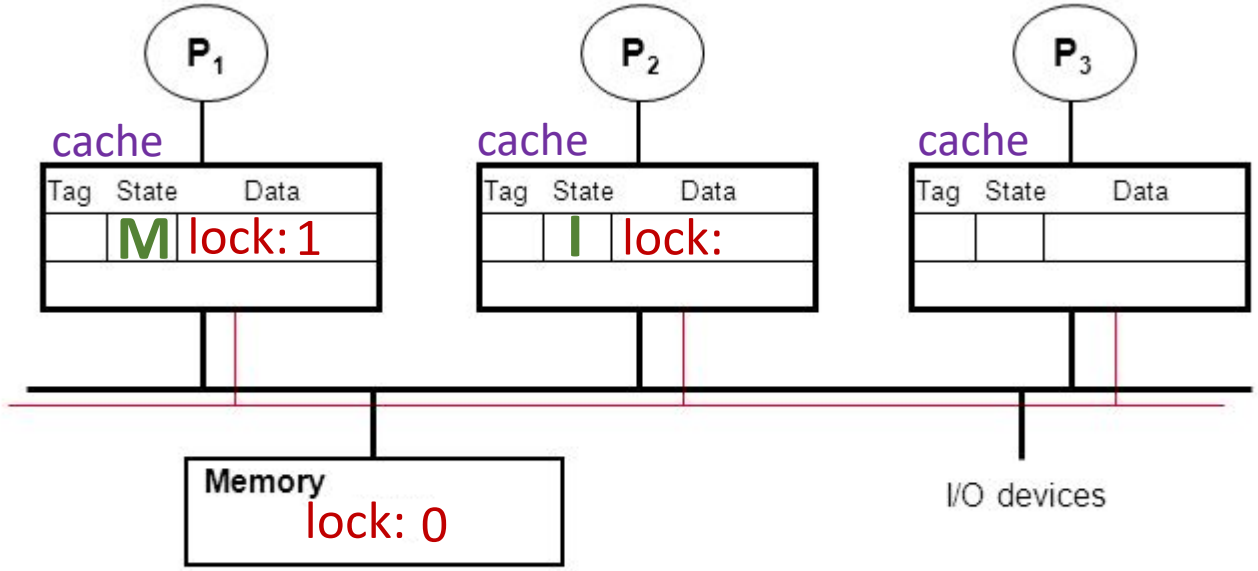
P1

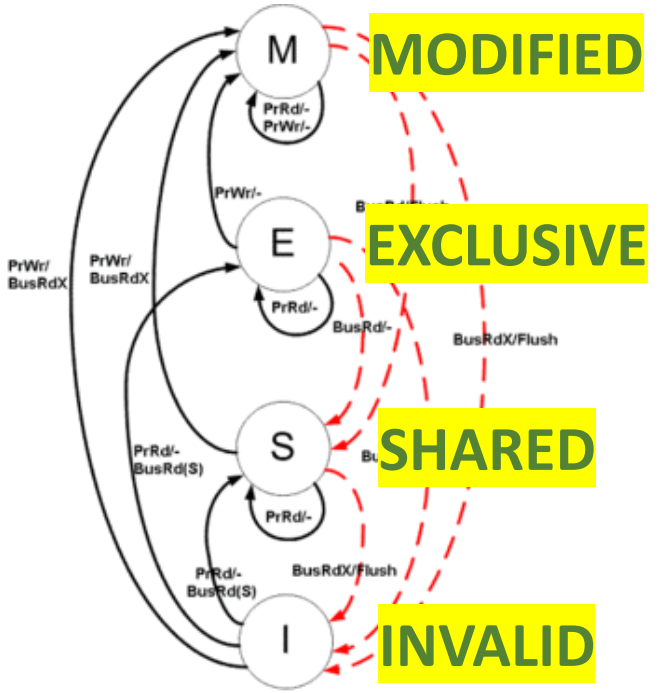
```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
        test R0
        bnz try
        store lock, 1
}
```

P2

```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
        test R0
        bnz try
        store lock, 1
}
```

Cache Coherence Action Zone





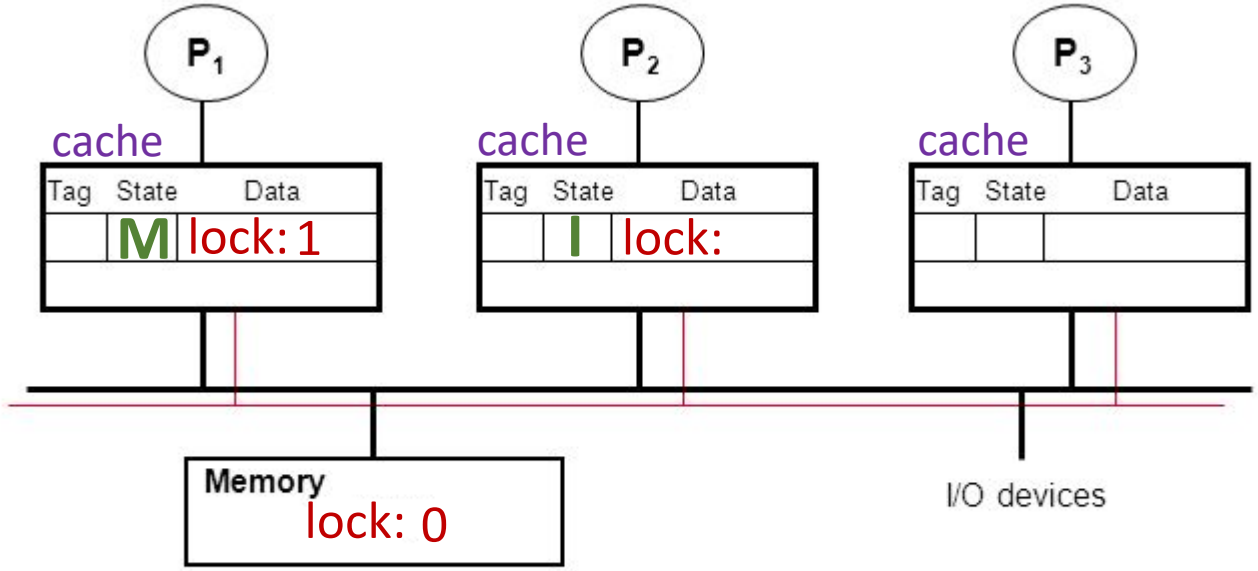
P1

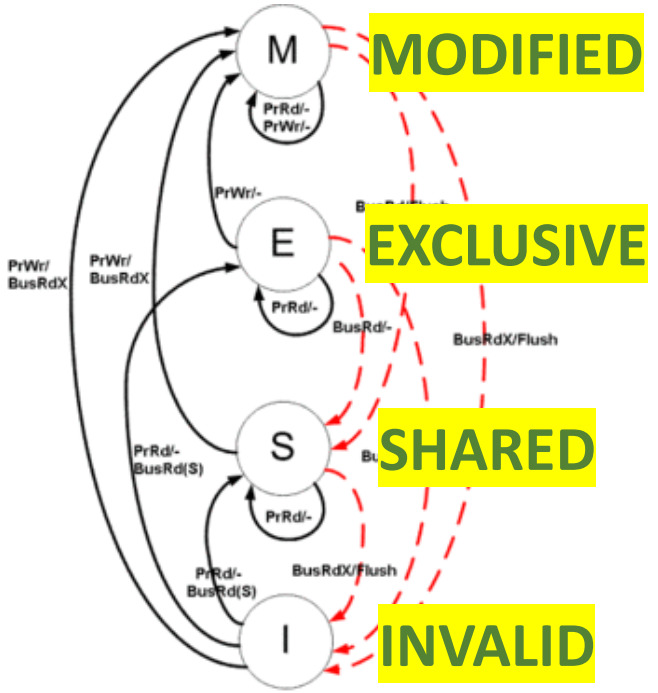
```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
        test R0
        bnz try
        store lock, 1
}
```

P2

```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
        test R0
        bnz try
        store lock, 1
}
```

Cache Coherence Action Zone





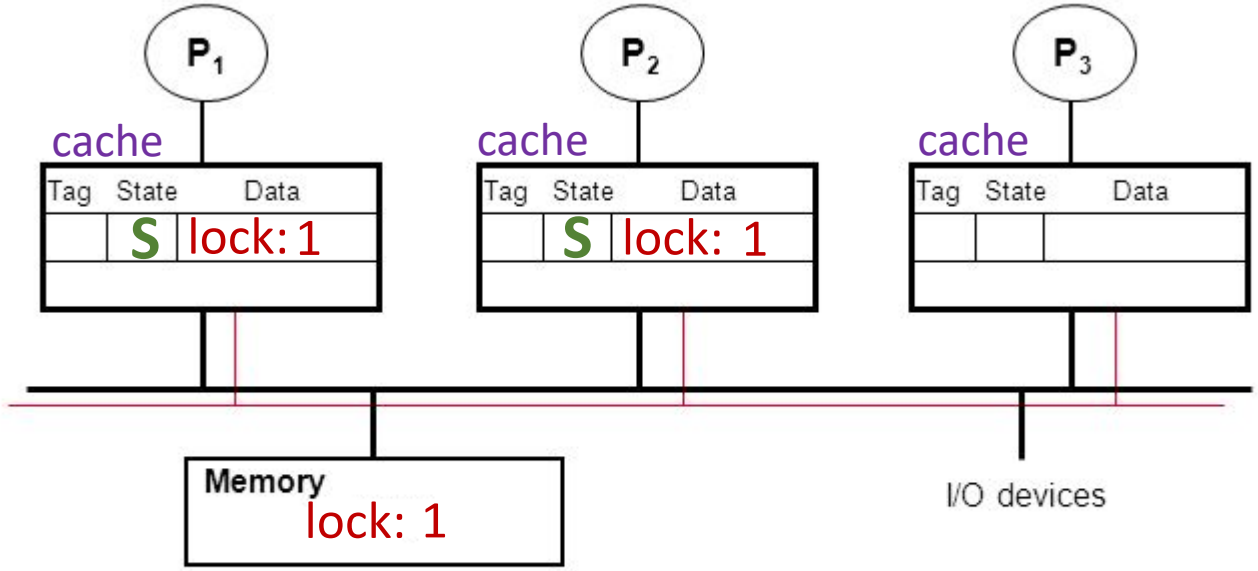
P1

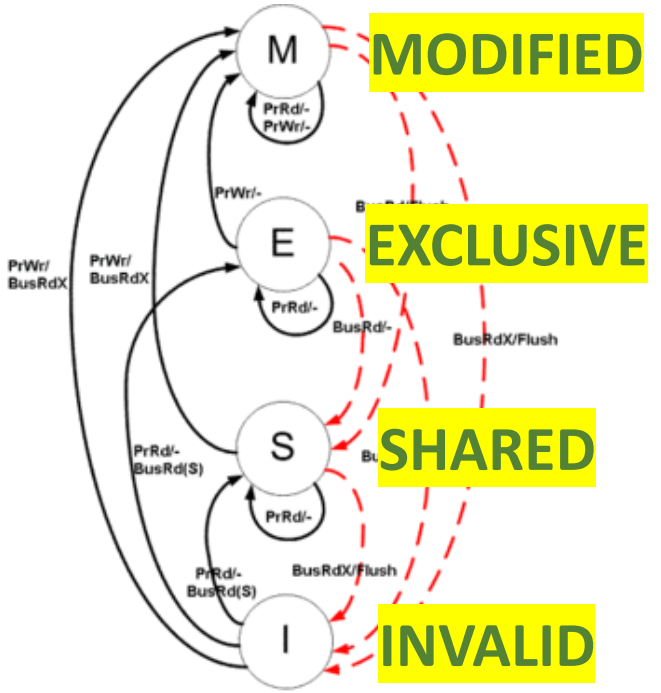
```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
        test R0
        bnz try
        store lock, 1
}
```

P2

```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
        test R0
        bnz try
        store lock, 1
}
```

Cache Coherence Action Zone

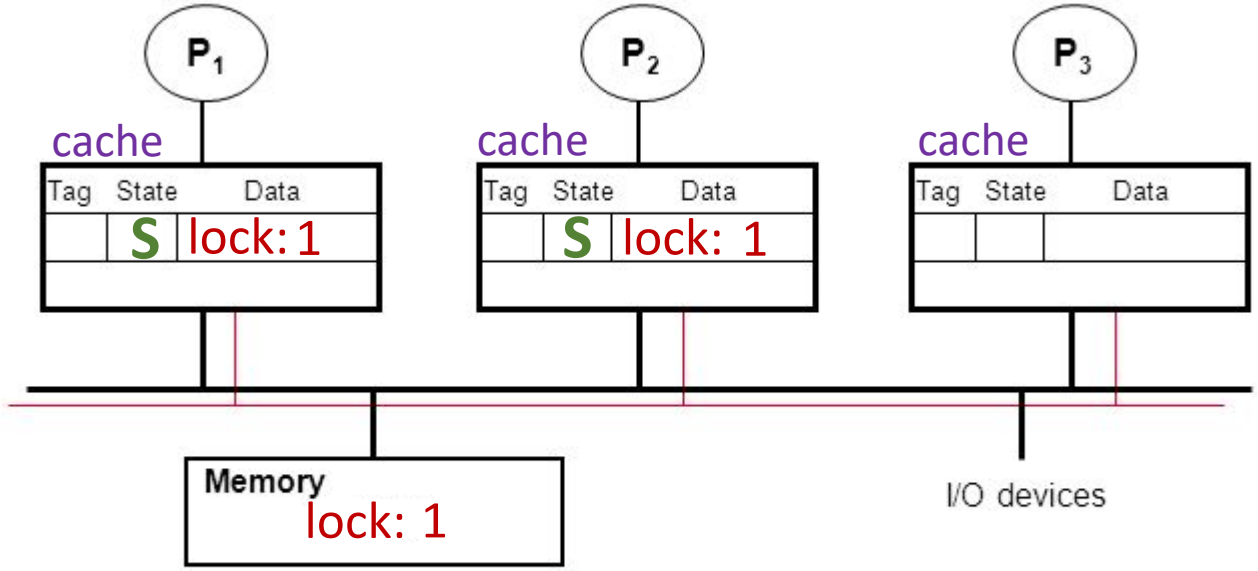


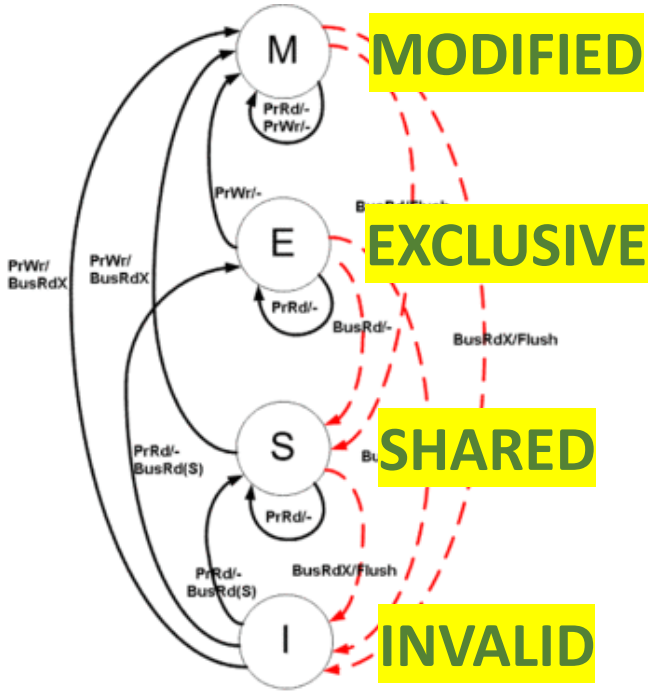


```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
        test R0
        bnz try
        store lock, 1
}
```

```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
        test R0
        bnz try
        store lock, 1
}
```

Cache Coherence Action Zone

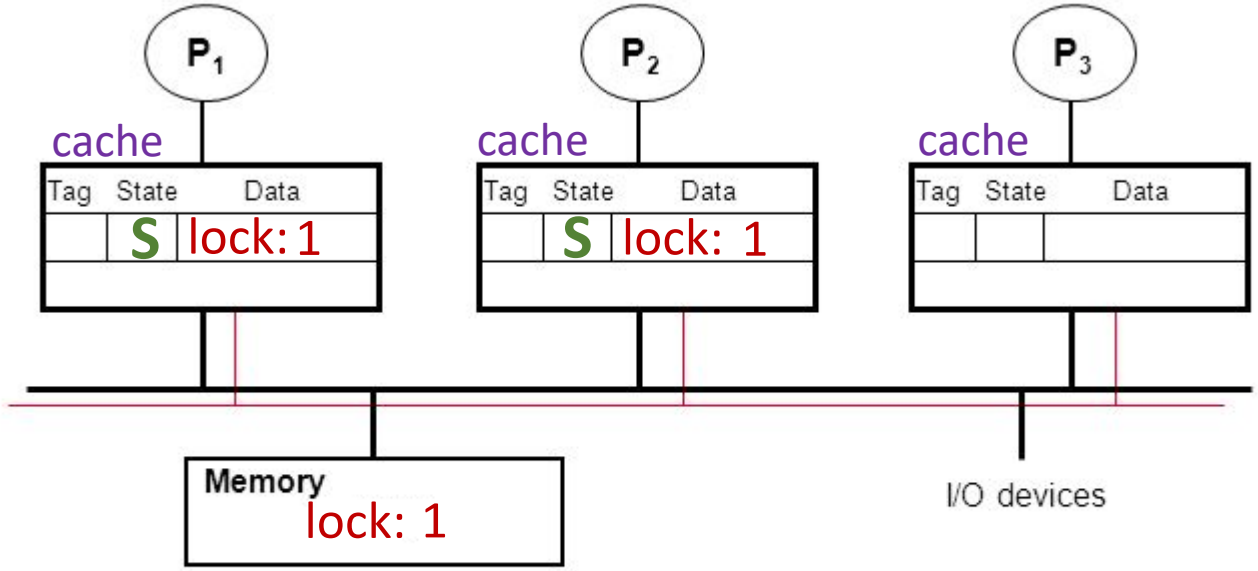


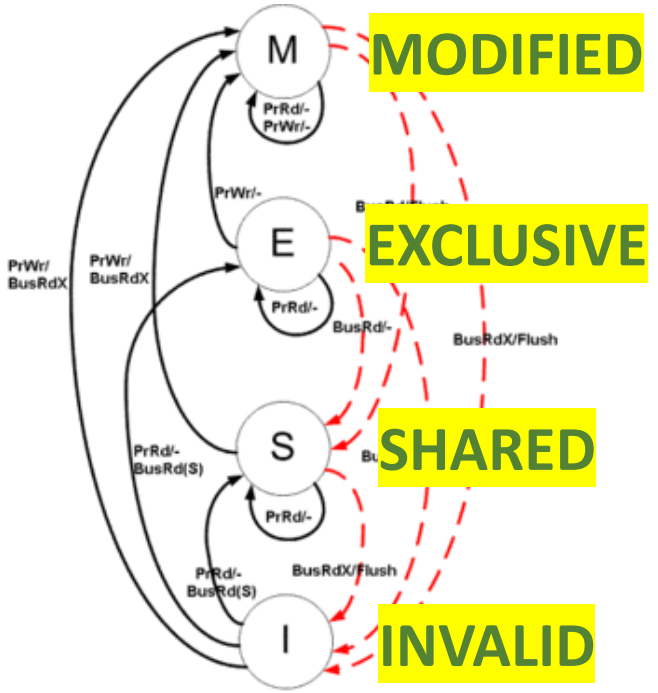


```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
        test R0
        bnz try
        store lock, 1
}
```

```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
        test R0
        bnz try
        store lock, 1
}
```

Cache Coherence Action Zone





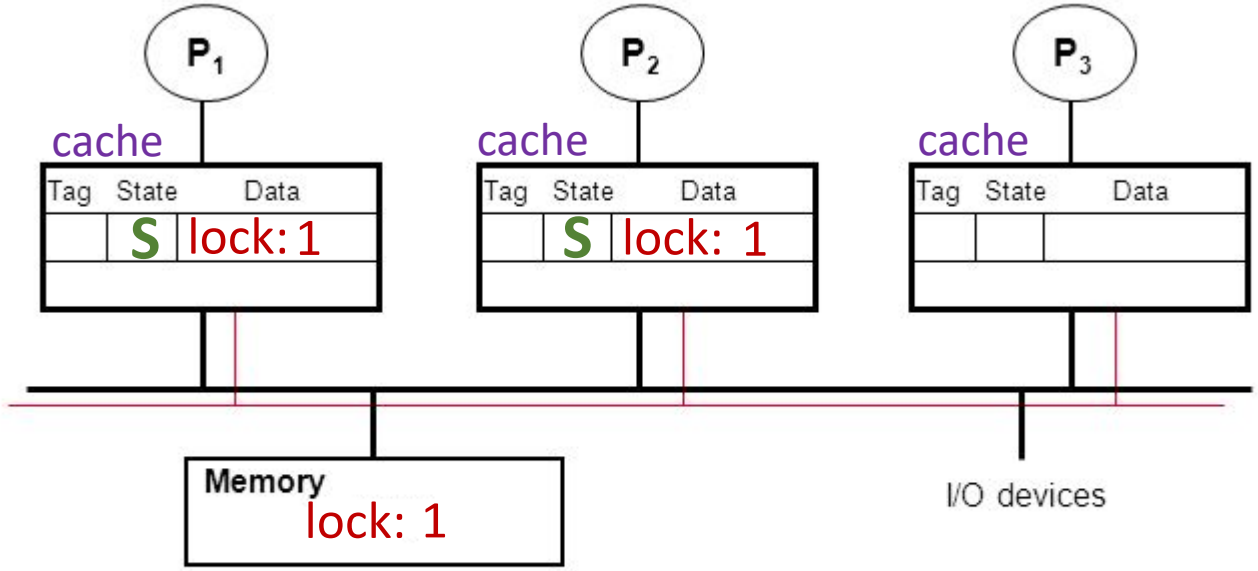
P1

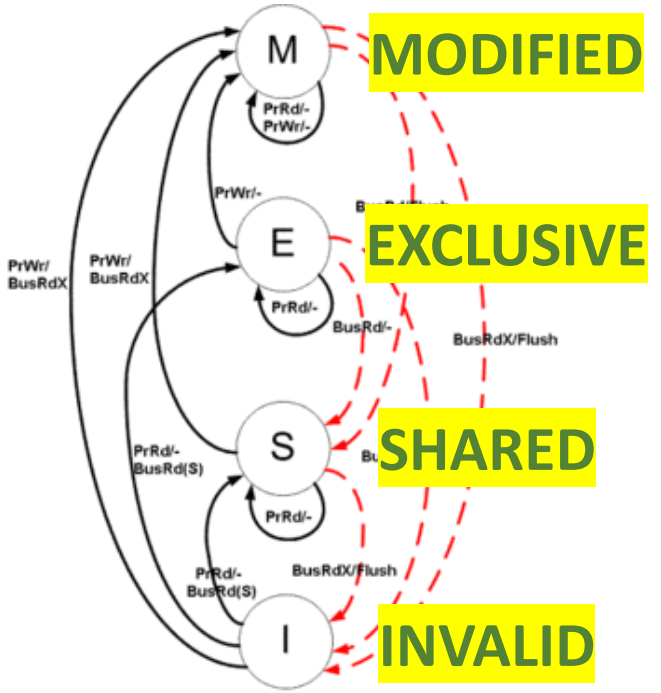
```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
        test R0
        bnz try
        store lock, 1
}
```

```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
        test R0
        bnz try
        store lock, 1
}
```

P2

Cache Coherence Action Zone





P1

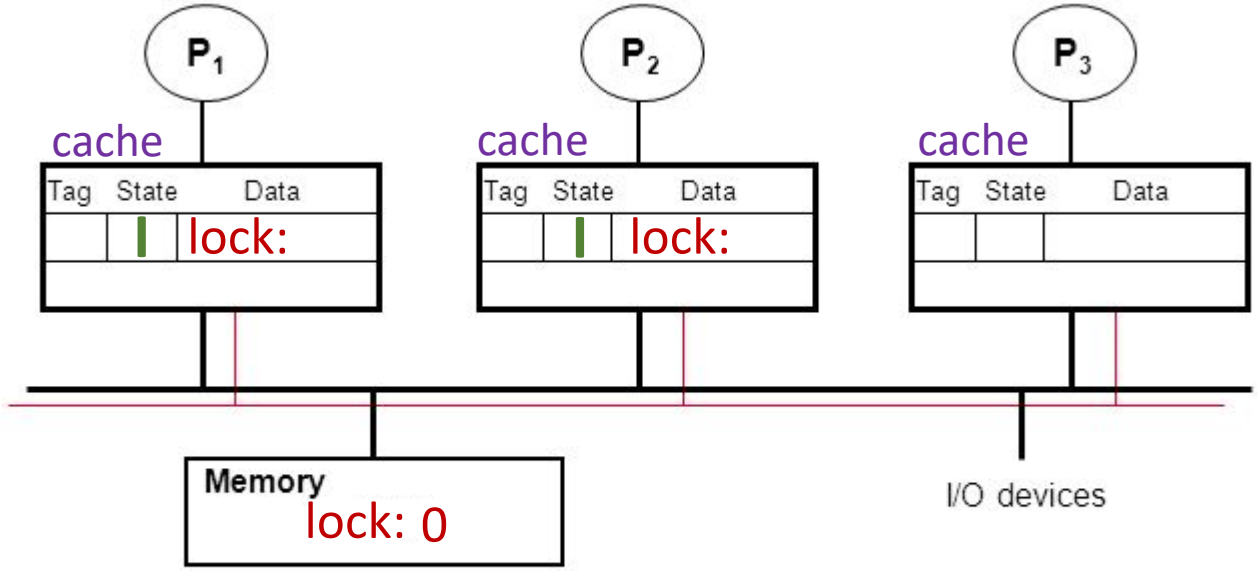
```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
        test R0
        bnz try
        store lock, 1
}
```

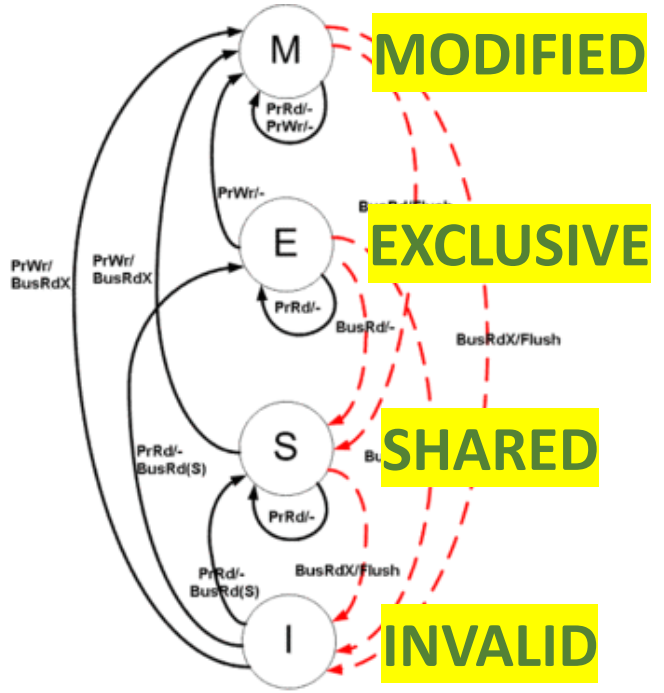
SAFE!

P2

```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
        test R0
        bnz try
        store lock, 1
}
```

Cache Coherence Action Zone II





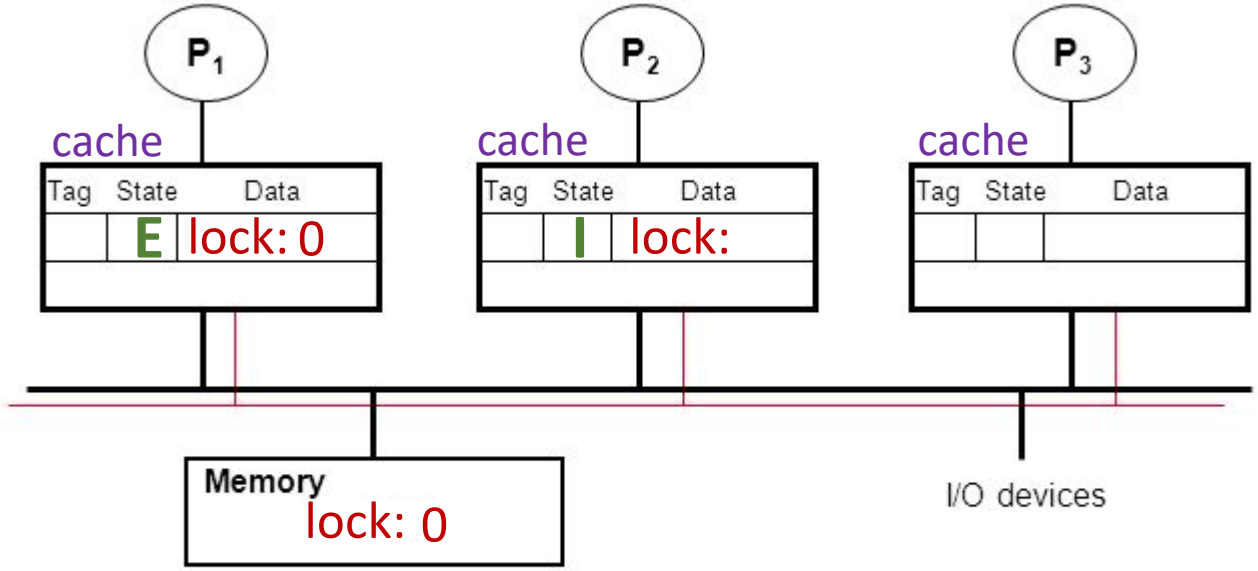
P1

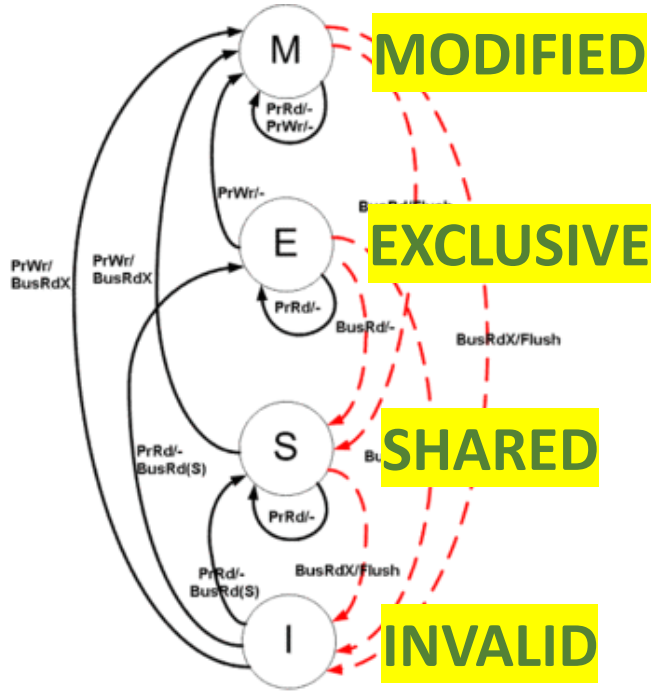
```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
        test R0
        bnz try
        store lock, 1
}
```

P2

```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
        test R0
        bnz try
        store lock, 1
}
```

Cache Coherence Action Zone II





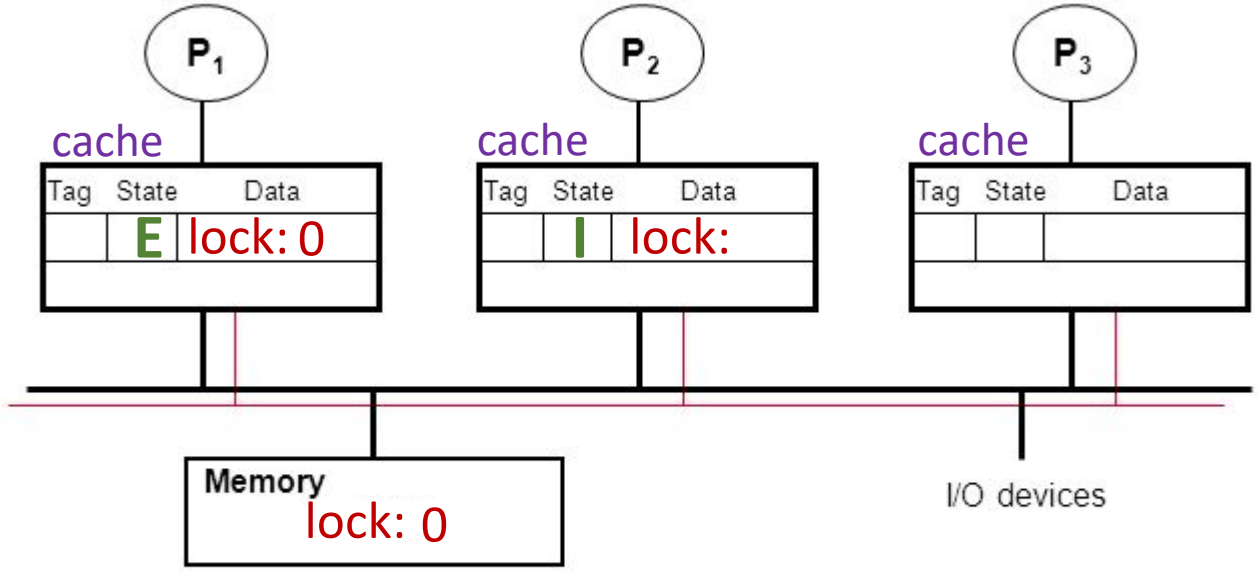
P1

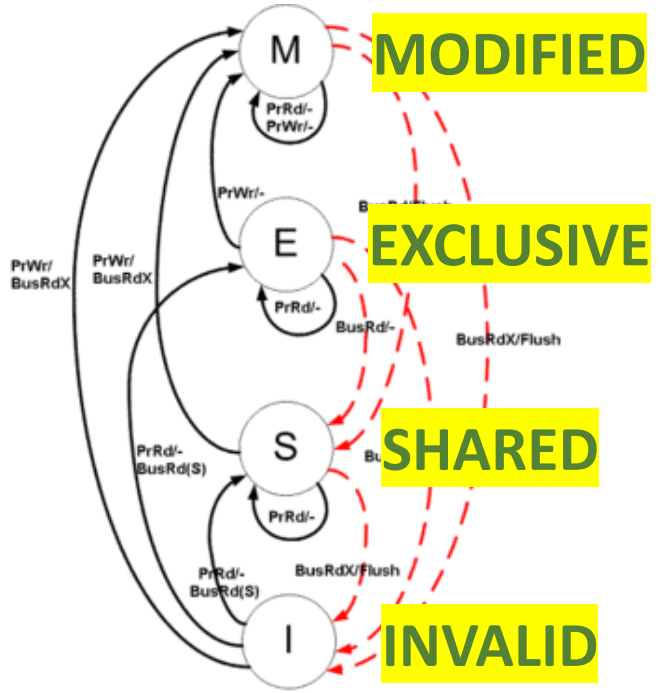
```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
        test R0
        bnz try
        store lock, 1
}
```

P2

```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
        test R0
        bnz try
        store lock, 1
}
```

Cache Coherence Action Zone II





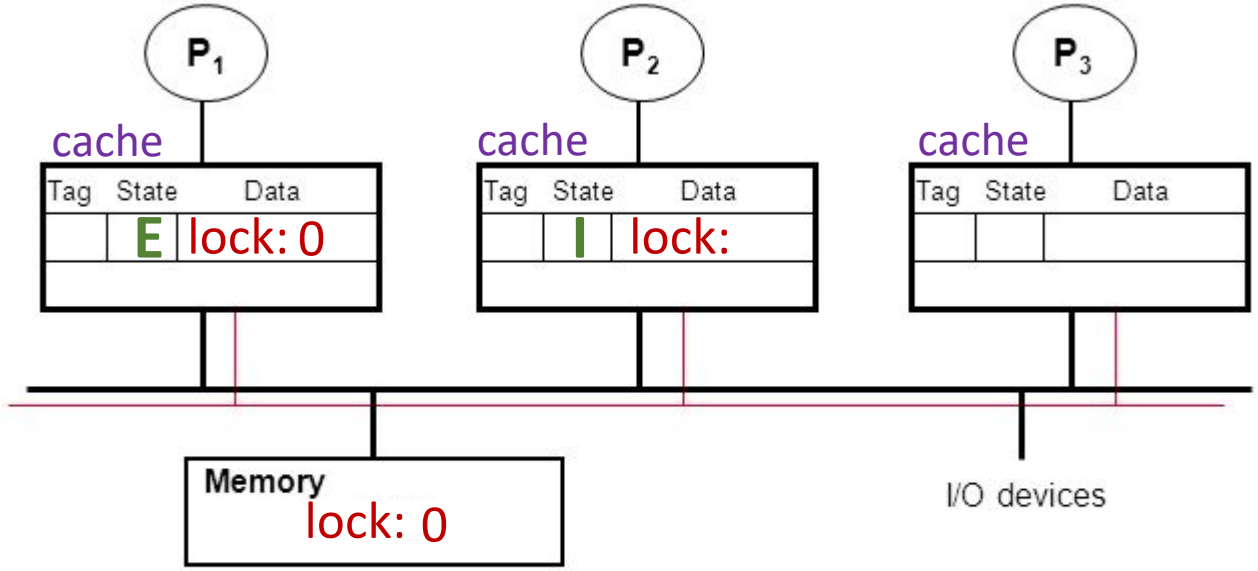
P1

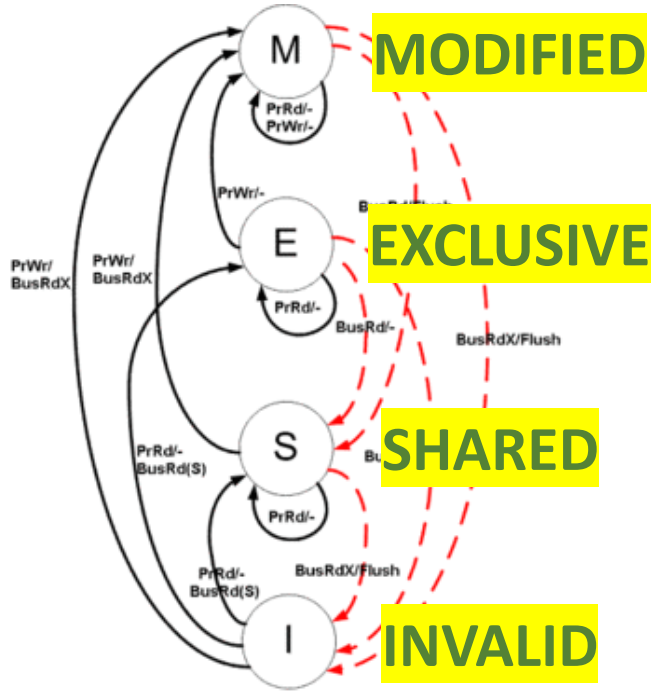
```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
        test R0
        bnz try
        store lock, 1
}
```

P2

```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
        test R0
        bnz try
        store lock, 1
}
```

Cache Coherence Action Zone II





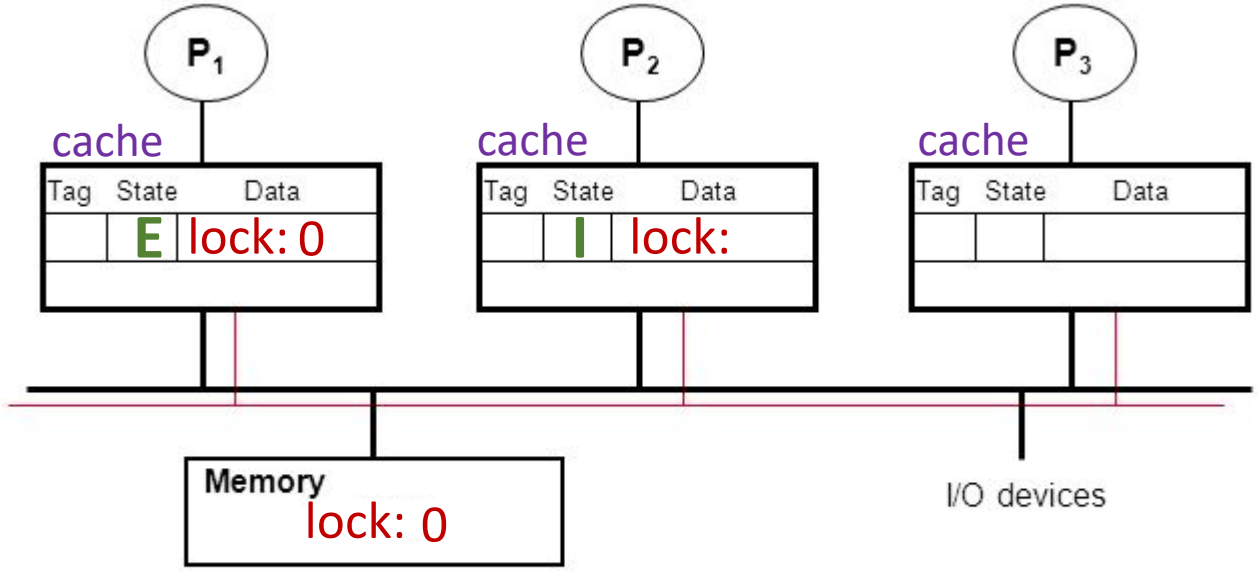
P1

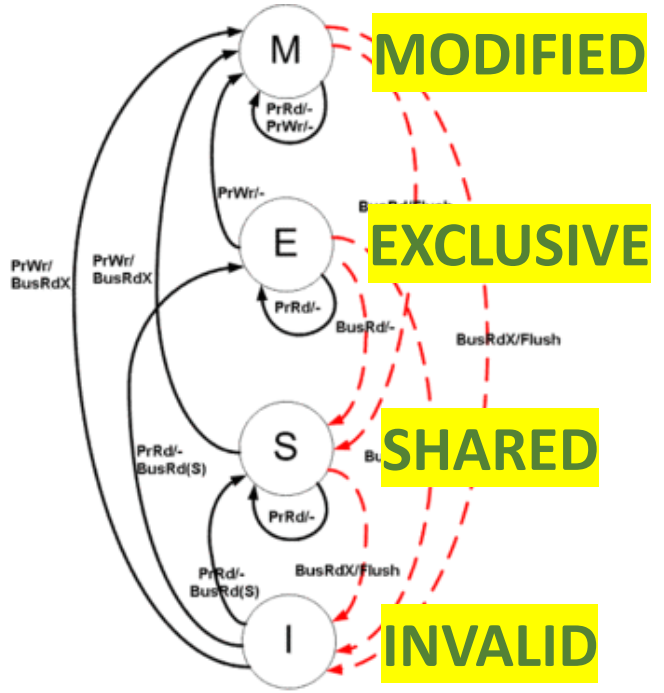
```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
        test R0
        bnz try
        store lock, 1
}
```

P2

```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
        test R0
        bnz try
        store lock, 1
}
```

Cache Coherence Action Zone II





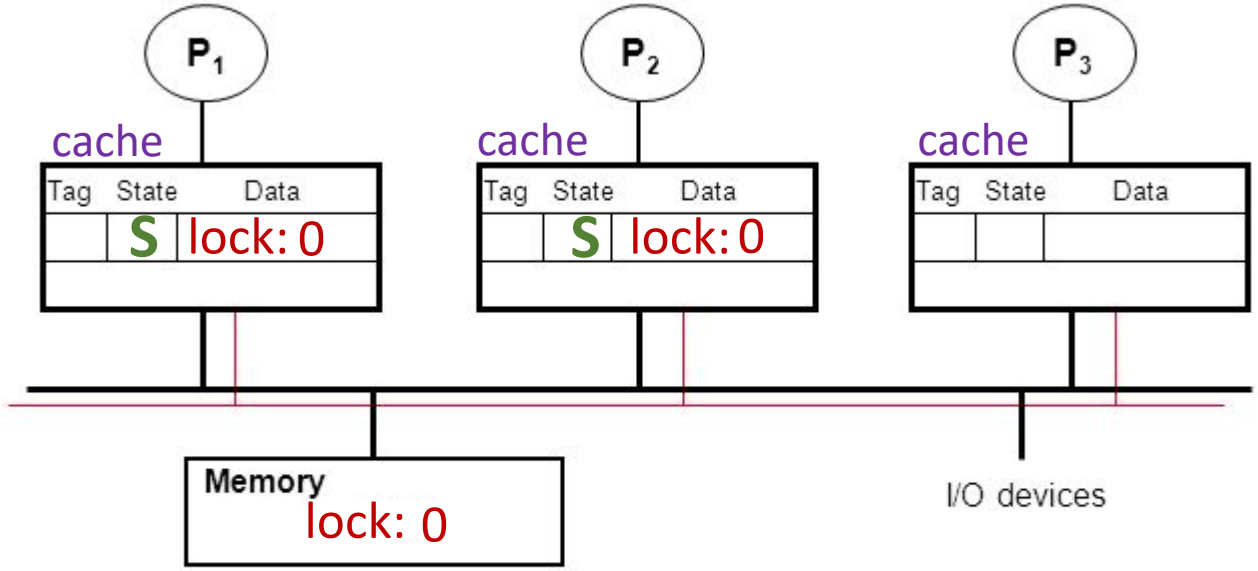
P1

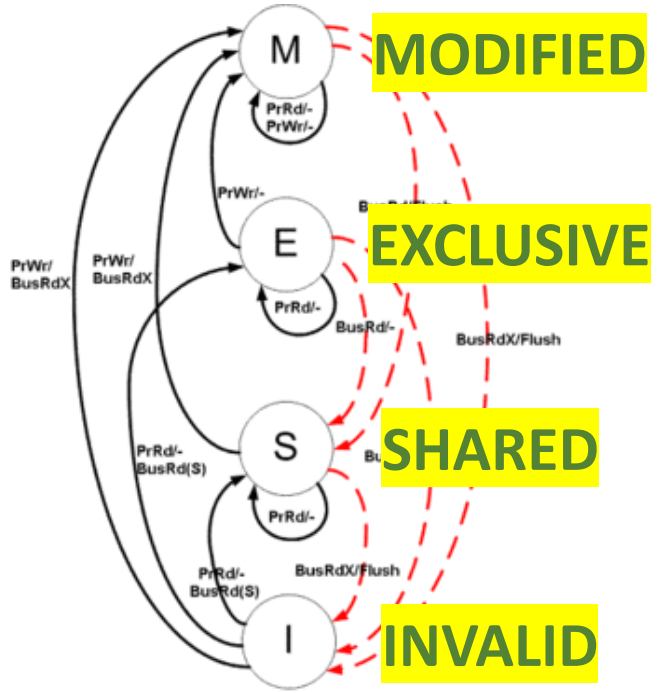
```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
        test R0
        bnz try
        store lock, 1
}
```

P2

```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
        test R0
        bnz try
        store lock, 1
}
```

Cache Coherence Action Zone II





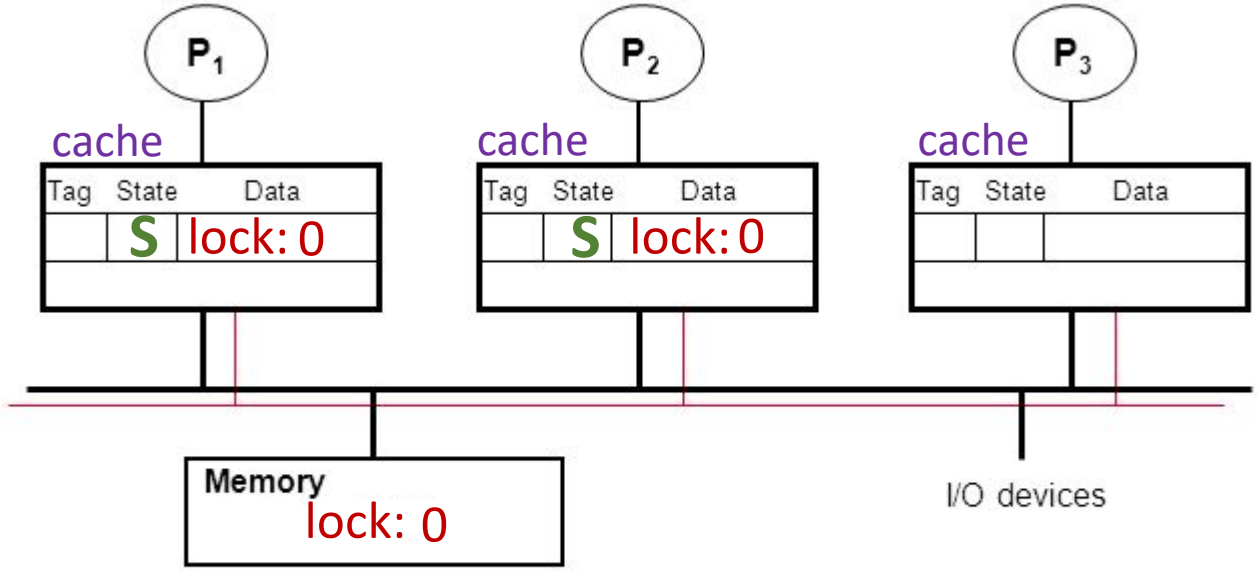
P1

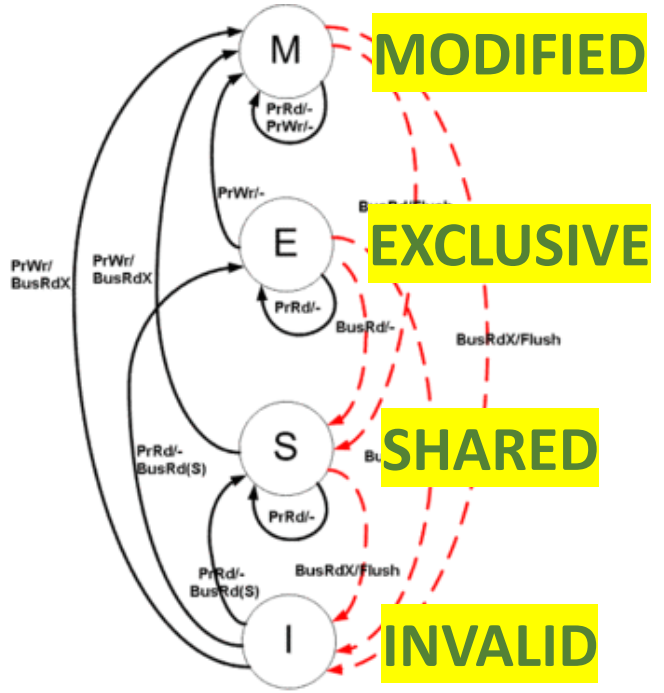
```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
        test R0
        bnz try
        store lock, 1
}
```

P2

```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
        test R0
        bnz try
        store lock, 1
}
```

Cache Coherence Action Zone II





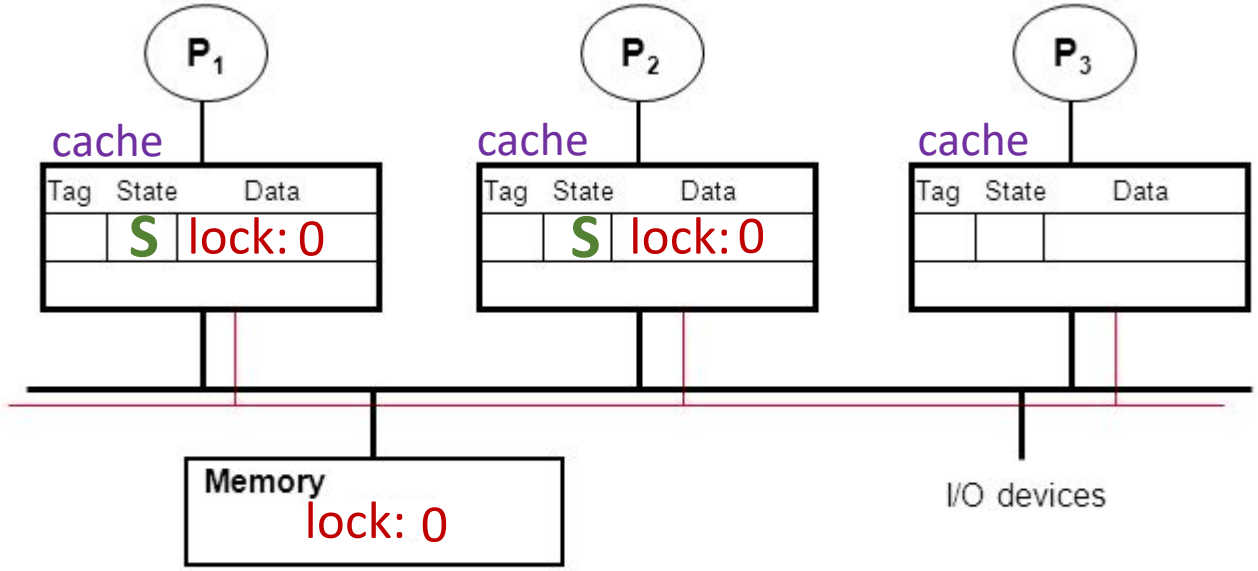
P1

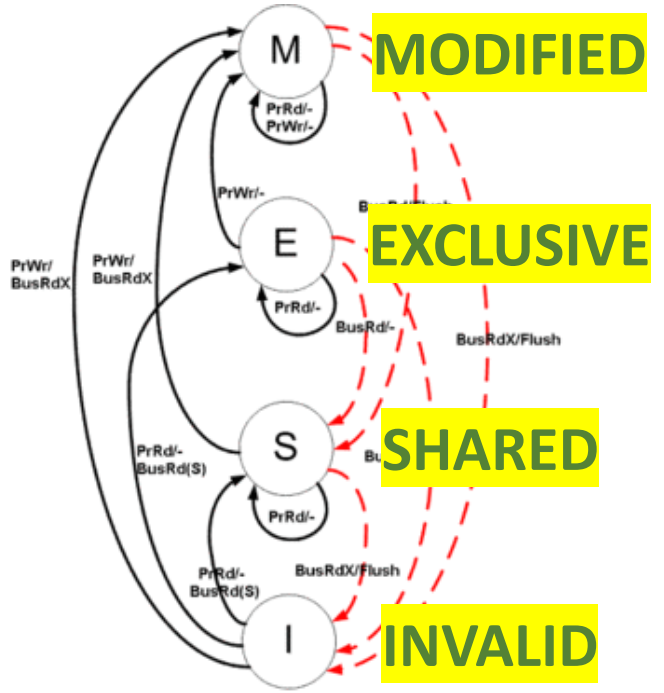
```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
        test R0
        bnz try
        store lock, 1
}
```

P2

```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
        test R0
        bnz try
        store lock, 1
}
```

Cache Coherence Action Zone II





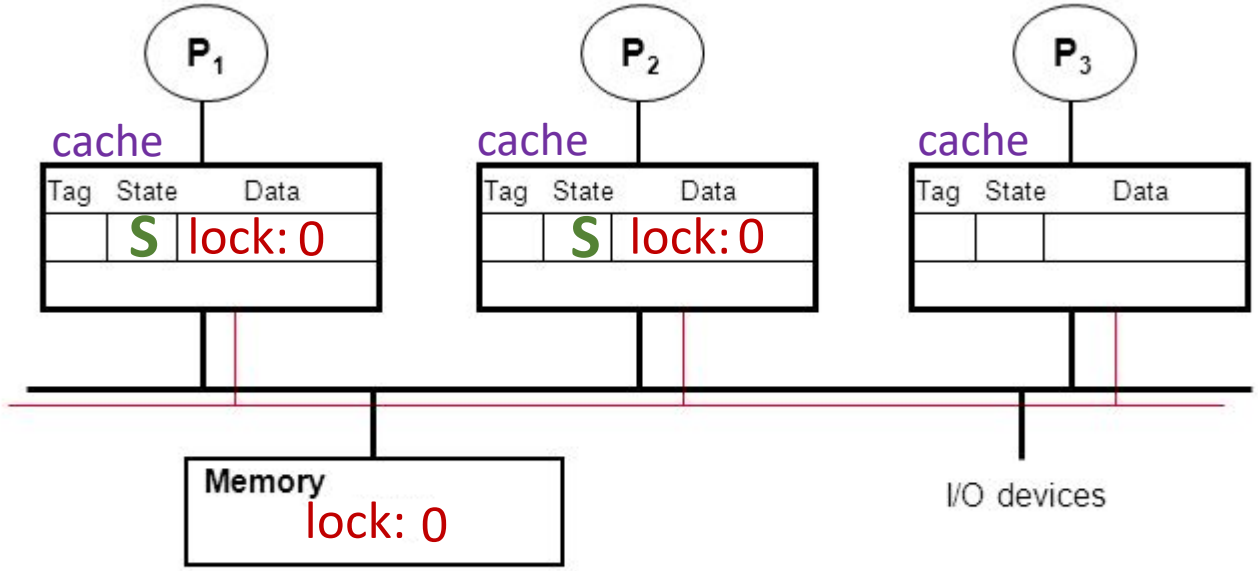
P1

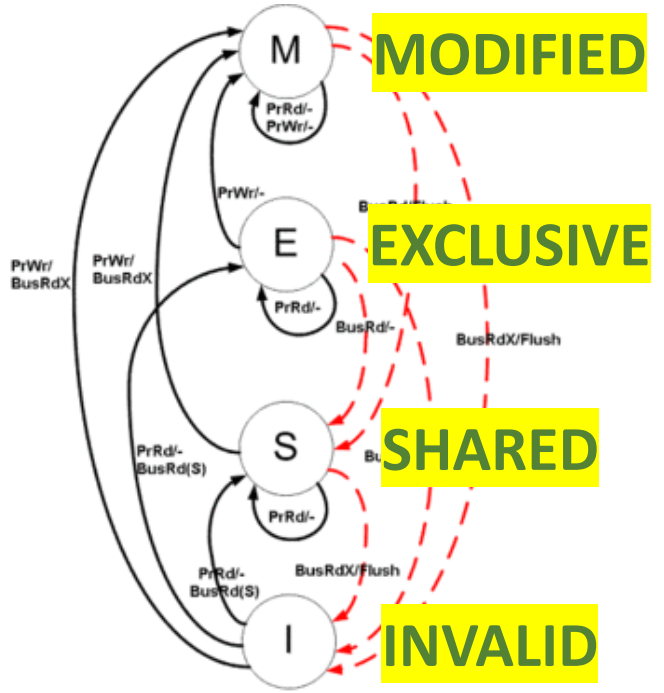
```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
        test R0
        bnz try
        store lock, 1
}
```

P2

```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
        test R0
        bnz try
        store lock, 1
}
```

Cache Coherence Action Zone II





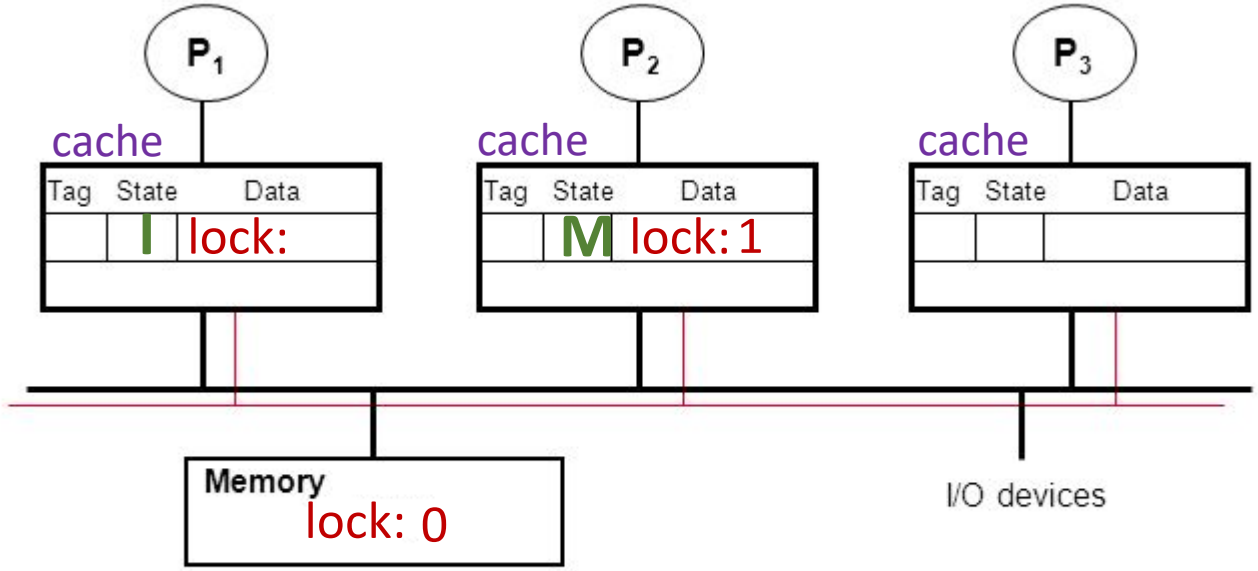
P1

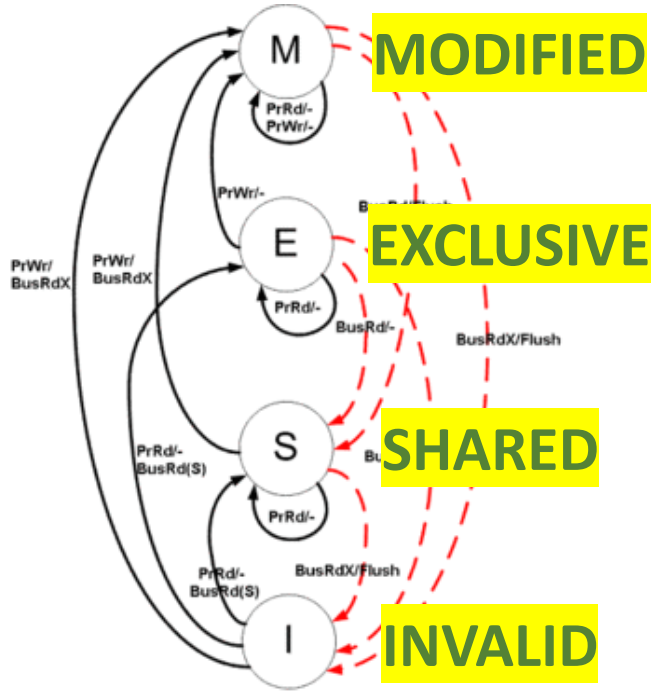
```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
        test R0
        bnz try
        store lock, 1
}
```

P2

```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
        test R0
        bnz try
        store lock, 1
}
```

Cache Coherence Action Zone II





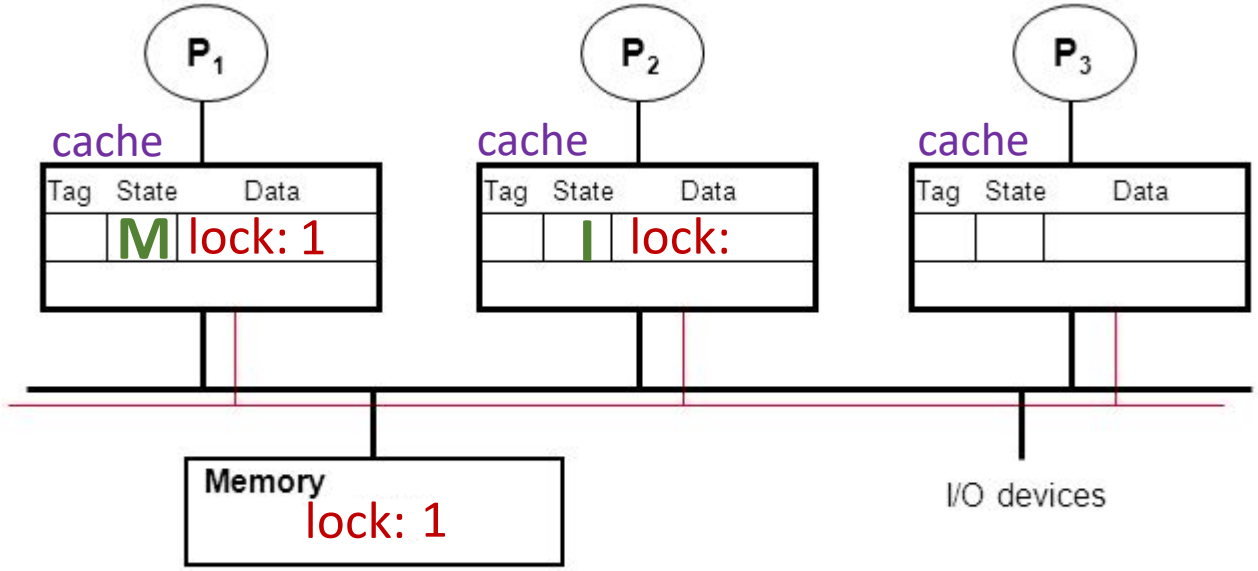
P1

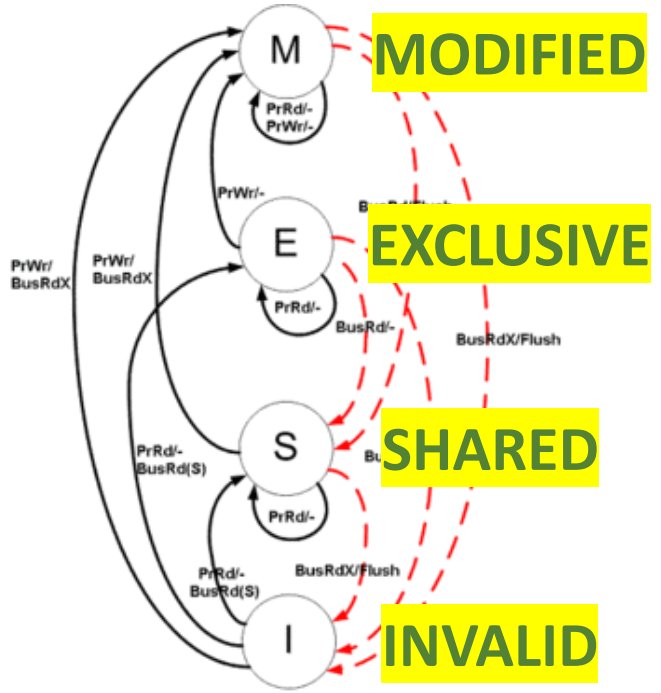
```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
        test R0
        bnz try
        store lock, 1
}
```

P2

```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
        test R0
        bnz try
        store lock, 1
}
```

Cache Coherence Action Zone II

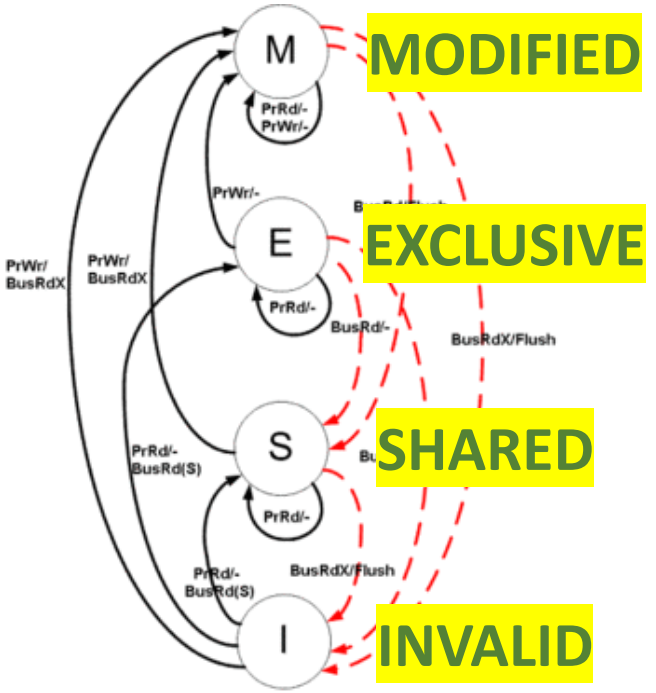




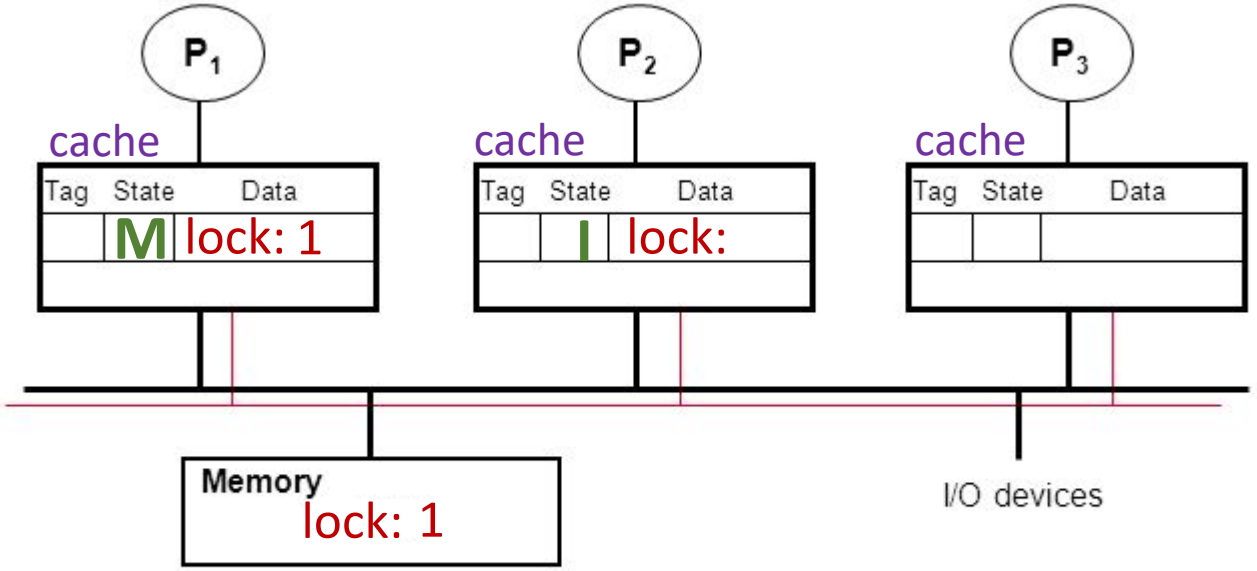
```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
        test R0
        bnz try
        store lock, 1
}
```

```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
        test R0
        bnz try
        store lock, 1
}
```

Cache Coherence Action Zone II



NOT SAFE!



P1

P2

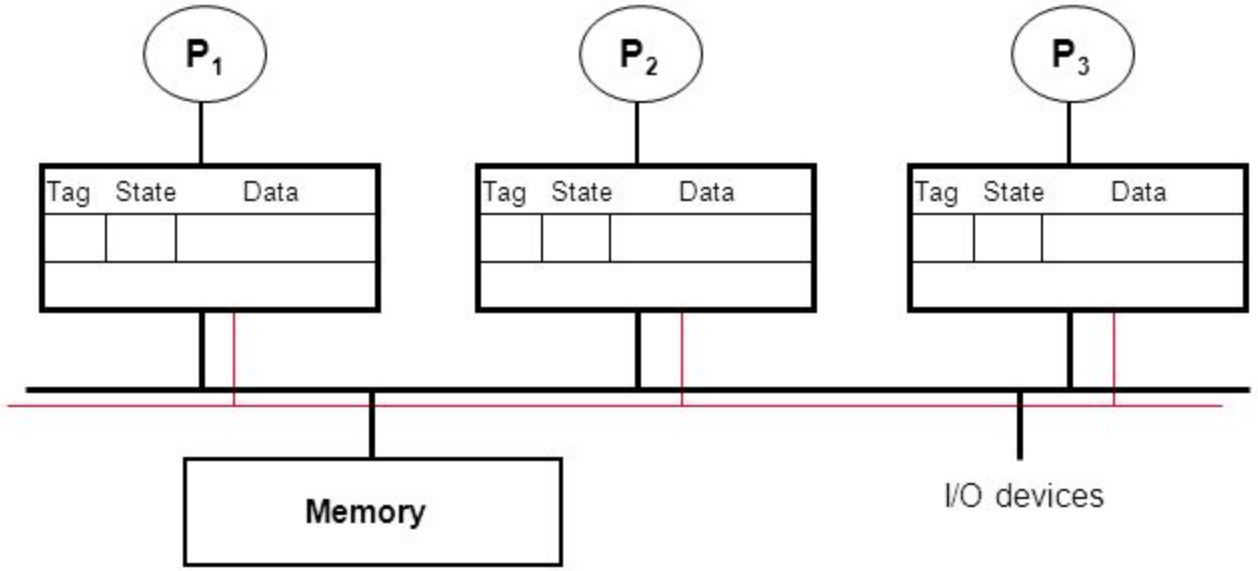
```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
        test R0
        bnz try
        store lock, 1
}
```

```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
        test R0
        bnz try
        store lock, 1
}
```

Read-Modify-Write (RMW)

- Implementing locks requires read-modify-write operations
- Required effect is:
  - An atomic and isolated action
    1. read memory location **AND**
    2. write a new value to the location
  - RMW is *very tricky* in multi-processors
  - Cache coherence alone doesn't solve it

```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
      test R0
      bnz try
      store lock, 1
}
```



Essence of HW-supported RMW

```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
      test R0
      bnz try
      store lock, 1
}
```

Make this into a single
(atomic hardware instruction)

HW Support for Read-Modify-Write (RMW)

Test & Set	CAS	Exchange, locked increment/decrement,	LLSC: load-linked store-conditional
Most architectures	Many architectures	x86	PPC, Alpha, MIPS

```
int TST(addr) {
  atomic {
    ret = *addr;
    if(!*addr)
      *addr = 1;
    return ret;
  }
}
```

```
bool cas(addr, old, new) {
  atomic {
    if(*addr == old) {
      *addr = new;
      return true;
    }
    return false;
  }
}
```

```
int XCHG(addr, val) {
  atomic {
    ret = *addr;
    *addr = val;
    return ret;
  }
}
```

```
bool LLSC(addr, val) {
 ret = *addr;
 atomic {
   if(*addr == ret) {
     *addr = val;
     return true;
  }
  return false;
}
```

HW Support for Read-Modify-Write (RMW)

Test & Set	CAS	Exchange, locked increment/decrement,	LLSC: load-linked store-conditional
Most architectures	Many architectures	x86	PPC, Alpha, MIPS

```
int TST(addr) {
  atomic {
    ret = *addr;
    if(!*addr)
      *addr = 1;
    return ret;
  }
}
```

```
bool cas(addr, old, new) {
  atomic {
    if(*addr == old) {
      *addr = new;
      return true;
    }
    return false;
  }
}
```

```
int XCHG(addr, val) {
  atomic {
    ret = *addr;
    *addr = val;
    return ret;
  }
}
```

```
bool LLSC(addr, val) {
 ret = *addr;
 atomic {
   if(*addr == ret) {
     *addr = val;
     return true;
  }
  return false;
}
```

```
void CAS_lock(lock) {
  while(CAS(&lock, 0, 1) != true);
}
```

HW Support for Read-Modify-Write (RMW)

Test & Set	CAS	Exchange, locked increment/decrement,	LLSC: load-linked store-conditional
Most architectures	Many architectures	x86	PPC, Alpha, MIPS

```
int TST(addr) {
  atomic {
    ret = *addr;
    if(!*addr)
      *addr = 1;
    return ret;
  }
}
```

```
bool cas(addr, old, new) {
  atomic {
    if(*addr == old) {
      *addr = new;
      return true;
    }
    return false;
  }
}
```

```
int XCHG(addr, val) {
  atomic {
    ret = *addr;
    *addr = val;
    return ret;
  }
}
```

```
bool LLSC(addr, val) {
 ret = *addr;
 atomic {
   if(*addr == ret) {
     *addr = val;
     return true;
  }
  return false;
}
```

HW Support for RMW: LL-SC

LLSC: load-linked store-conditional

PPC, Alpha, MIPS

```
bool LLSC(addr, val) {
 ret = *addr;
 atomic {
   if(*addr == ret) {
     *addr = val;
     return true;
  }
  return false;
}
```

- load-linked is a load that is “linked” to a subsequent store-conditional
- Store-conditional only succeeds if value from linked-load is unchanged

HW Support for RMW: LL-SC

LLSC: load-linked store-conditional

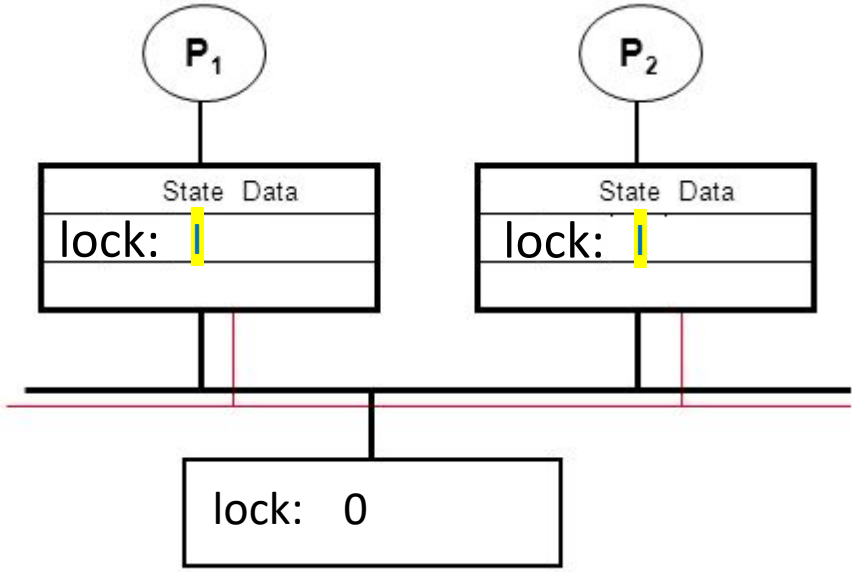
PPC, Alpha, MIPS

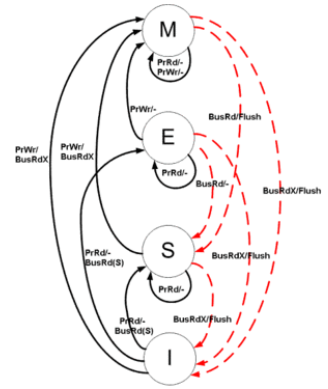
```
bool LLSC(addr, val) {
 ret = *addr;
 atomic {
   if(*addr == ret) {
     *addr = val;
     return true;
  }
  return false;
}
```

```
void LLSC_lock(lock) {
  while(1) {
    old = load-linked(lock);
    if(old == 0 && store-cond(lock, 1))
      return;
  }
}
```

- load-linked is a load that is “linked” to a subsequent store-conditional
- Store-conditional only succeeds if value from linked-load is unchanged

LLSC Lock Action Zone

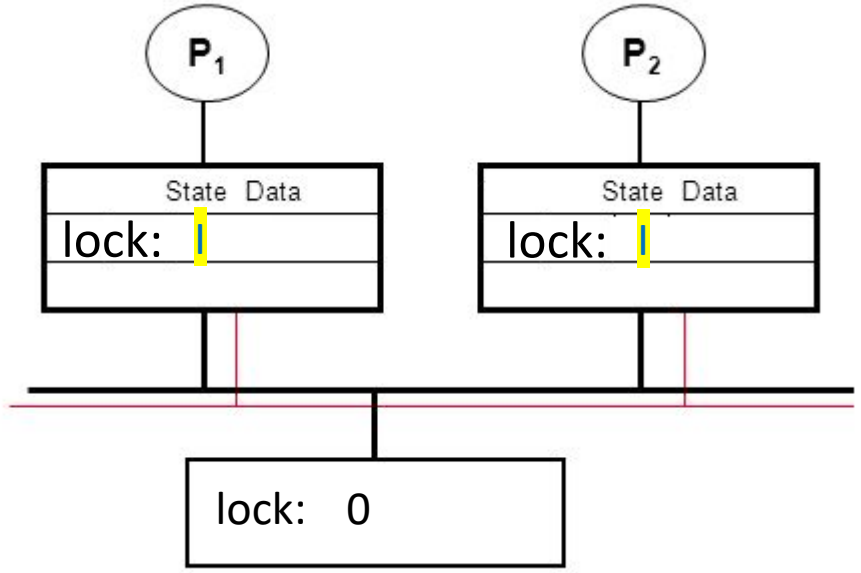


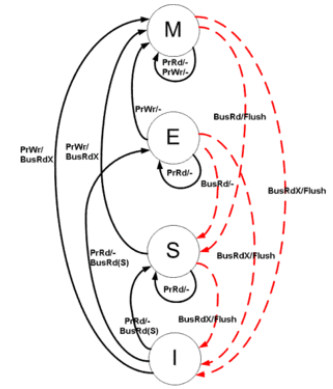



```
_______P1_____________
lock(lock) {
  while(1) {
    old = ll(lock);
    if(old == 0)
      if(sc(lock, 1))
        return;
  }
}
```

```
_______P2_____________
lock(lock) {
  while(1) {
    old = ll(lock);
    if(old == 0)
      if(sc(lock, 1))
        return;
  }
}
```

LLSC Lock Action Zone

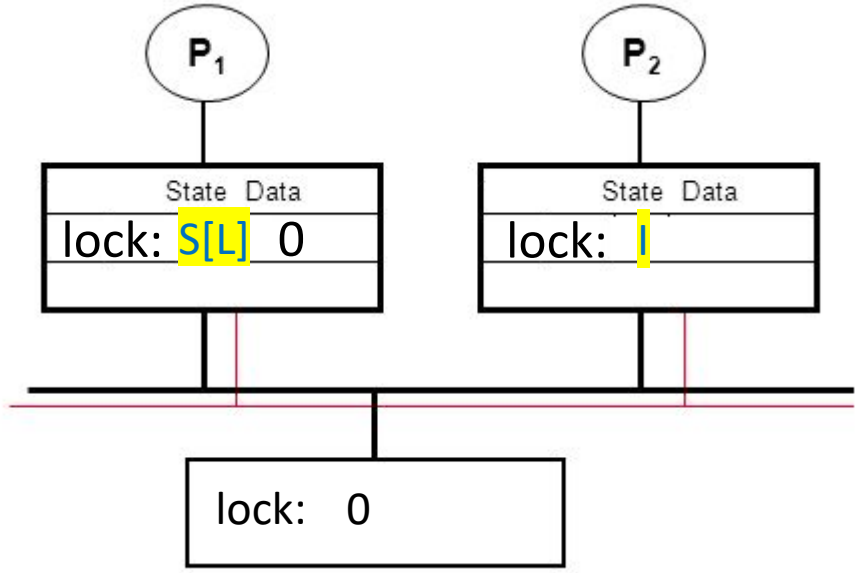


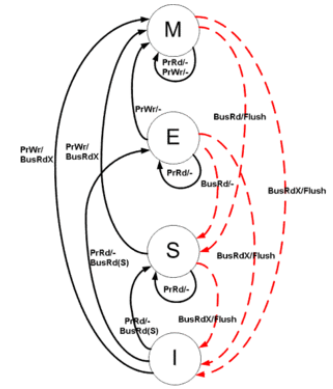



```
_____P1_____
lock(lock) {
  while(1) {
    old = ll(lock);
    if(old == 0)
      if(sc(lock, 1))
        return;
  }
}
```

```
_____P2_____
lock(lock) {
  while(1) {
    old = ll(lock);
    if(old == 0)
      if(sc(lock, 1))
        return;
  }
}
```

LLSC Lock Action Zone

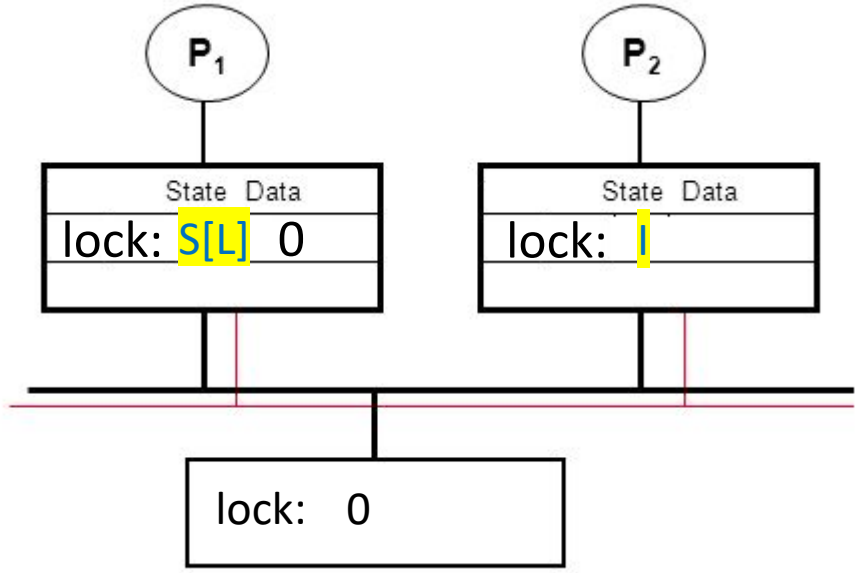


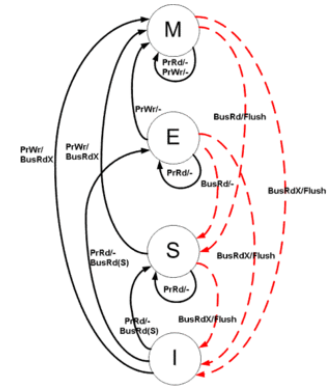



```
_____P1_____
lock(lock) {
  while(1) {
    old = ll(lock);
    if(old == 0)
      if(sc(lock, 1))
        return;
  }
}
```

```
_____P2_____
lock(lock) {
  while(1) {
    old = ll(lock);
    if(old == 0)
      if(sc(lock, 1))
        return;
  }
}
```

LLSC Lock Action Zone

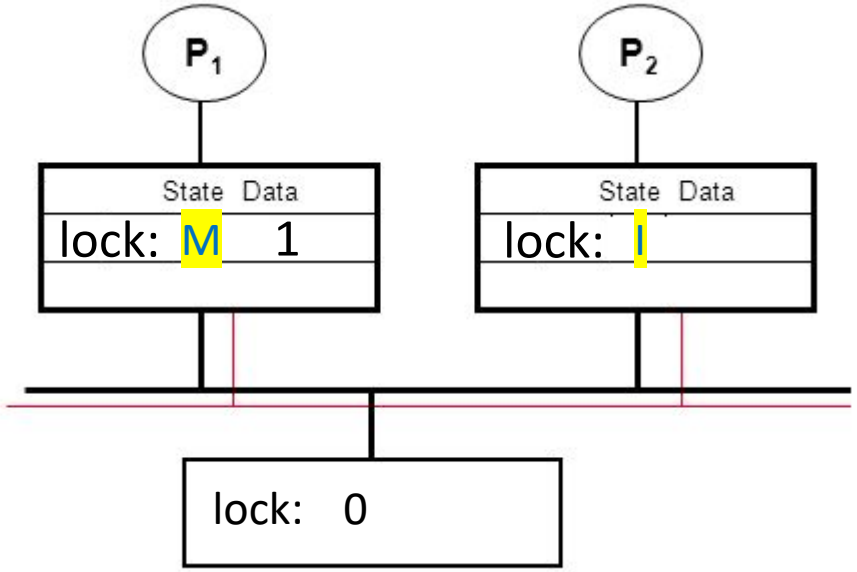


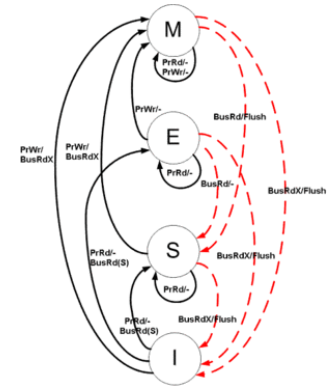



```
_____P1_____
lock(lock) {
  while(1) {
    old = ll(lock);
    if(old == 0)
      if(sc(lock, 1))
        return;
  }
}
```

```
_____P2_____
lock(lock) {
  while(1) {
    old = ll(lock);
    if(old == 0)
      if(sc(lock, 1))
        return;
  }
}
```

LLSC Lock Action Zone

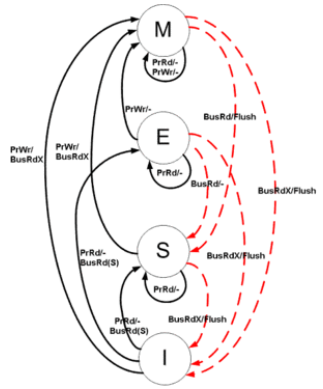


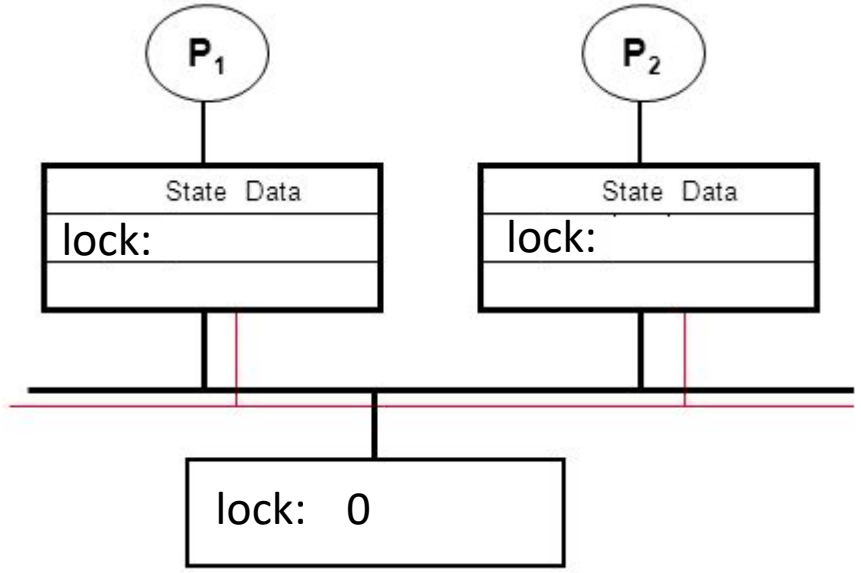



```
_____P1_____
lock(lock) {
  while(1) {
    old = ll(lock);
    if(old == 0)
      if(sc(lock, 1))
        return;
  }
}
```

```
_____P2_____
lock(lock) {
  while(1) {
    old = ll(lock);
    if(old == 0)
      if(sc(lock, 1))
        return;
  }
}
```

LLSC Lock Action Zone II

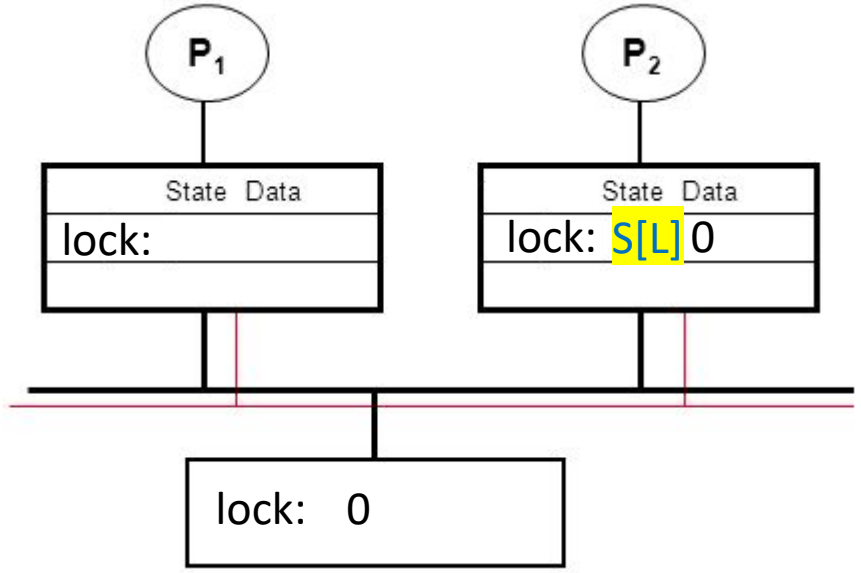


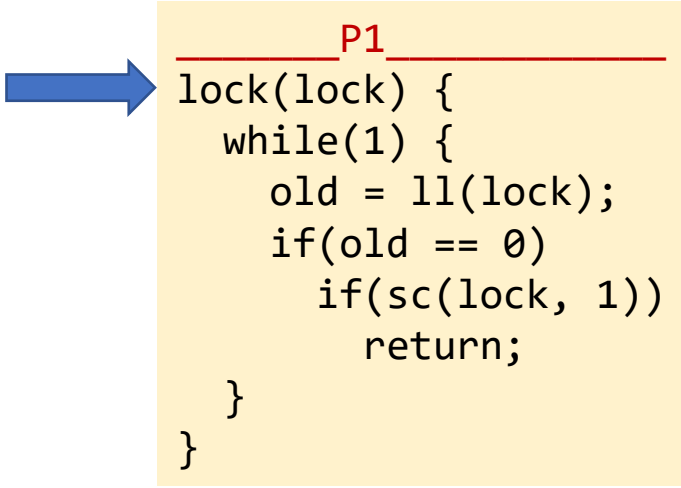



```
_______P1_____________
lock(lock) {
  while(1) {
    old = ll(lock);
    if(old == 0)
      if(sc(lock, 1))
        return;
  }
}
```

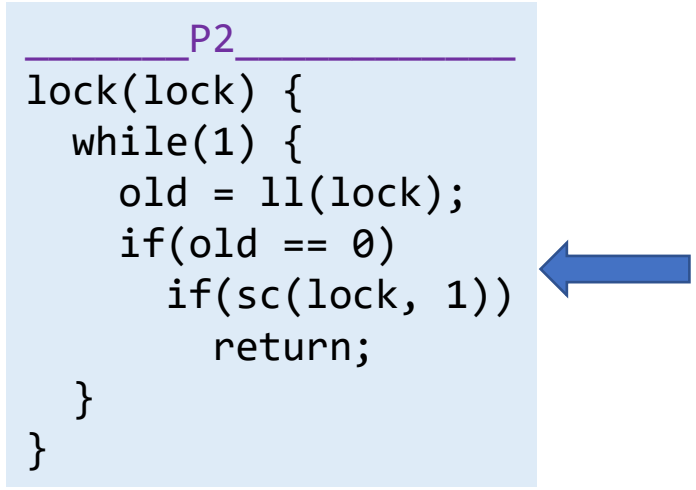
```
_______P2_____________
lock(lock) {
  while(1) {
    old = ll(lock);
    if(old == 0)
      if(sc(lock, 1))
        return;
  }
}
```

LLSC Lock Action Zone II

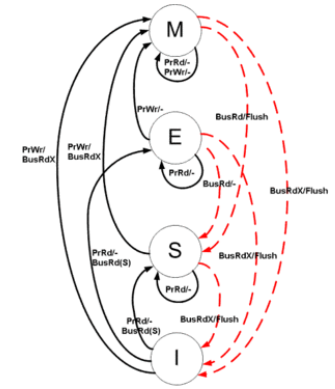




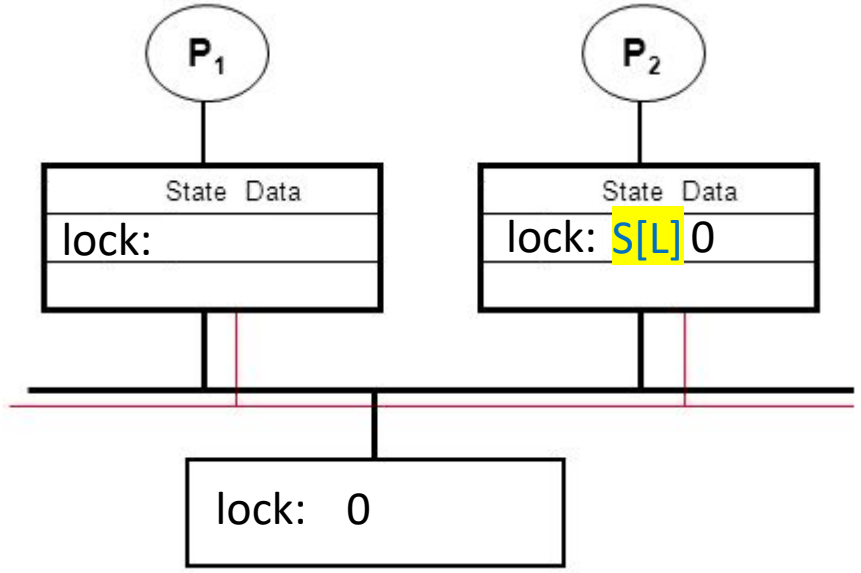
```
_____P1_____
lock(lock) {
  while(1) {
    old = ll(lock);
    if(old == 0)
      if(sc(lock, 1))
        return;
  }
}
```

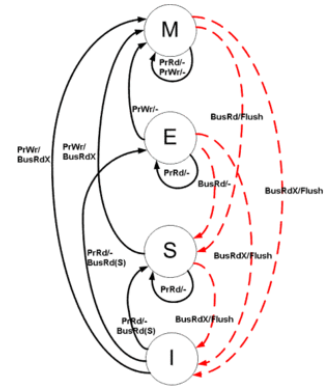


```
_____P2_____
lock(lock) {
  while(1) {
    old = ll(lock);
    if(old == 0)
      if(sc(lock, 1))
        return;
  }
}
```



LLSC Lock Action Zone II

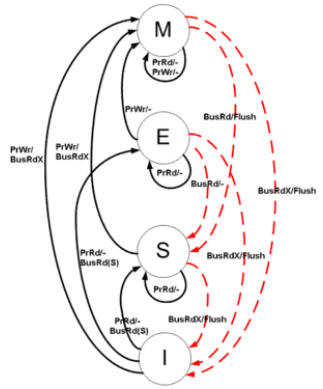


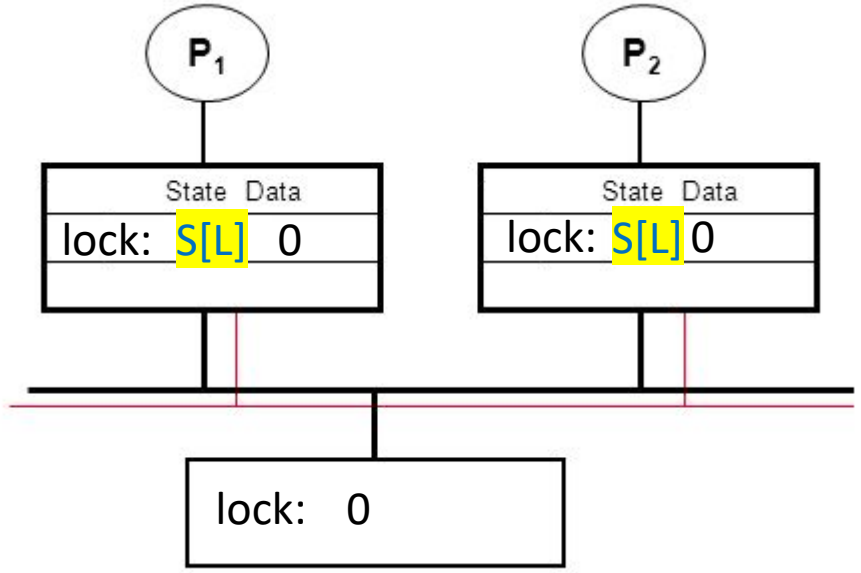



```
_____P1_____
lock(lock) {
  while(1) {
    old = ll(lock);
    if(old == 0)
      if(sc(lock, 1))
        return;
  }
}
```

```
_____P2_____
lock(lock) {
  while(1) {
    old = ll(lock);
    if(old == 0)
      if(sc(lock, 1))
        return;
  }
}
```

LLSC Lock Action Zone II

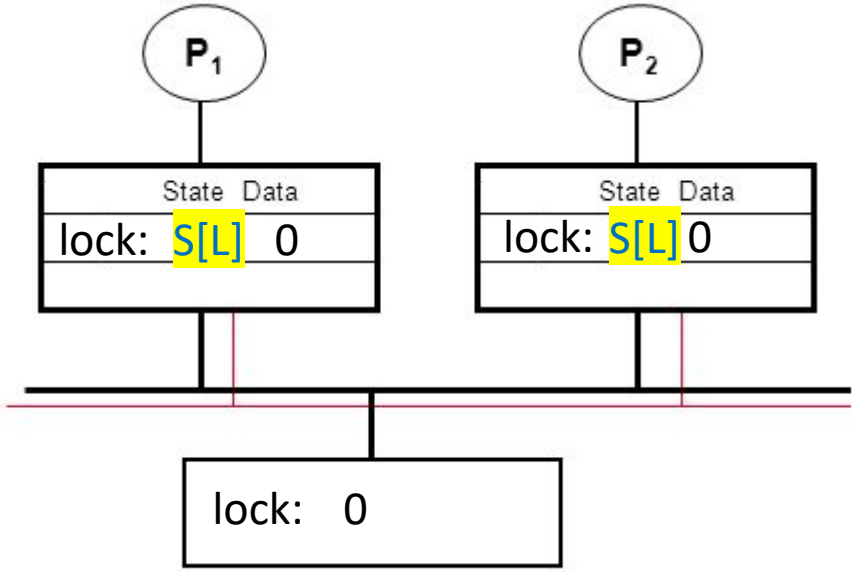


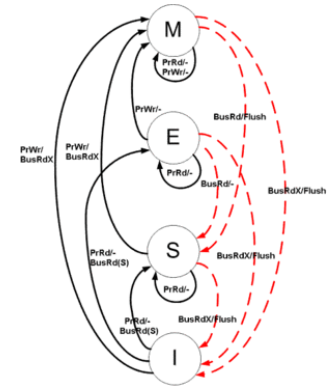



```
_____P1_____
lock(lock) {
  while(1) {
    old = ll(lock);
    if(old == 0)
      if(sc(lock, 1))
        return;
  }
}
```

```
_____P2_____
lock(lock) {
  while(1) {
    old = ll(lock);
    if(old == 0)
      if(sc(lock, 1))
        return;
  }
}
```

LLSC Lock Action Zone II

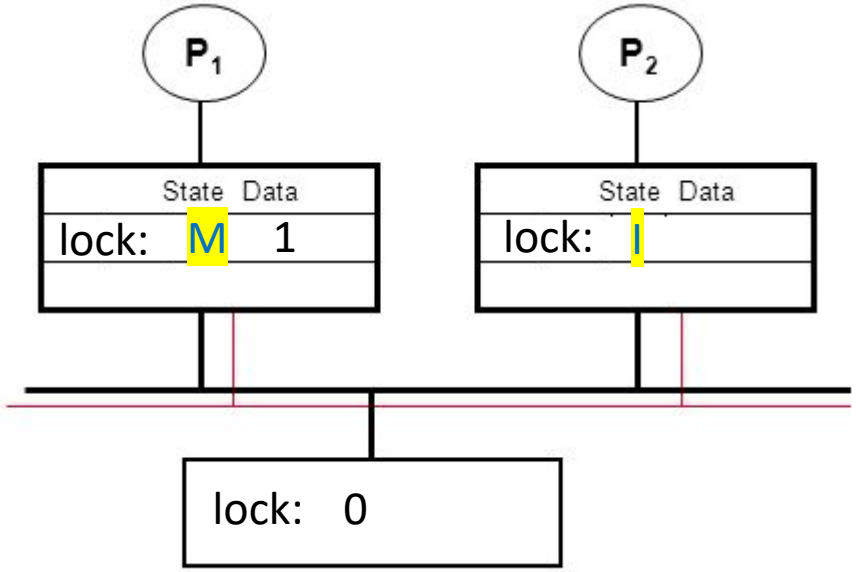


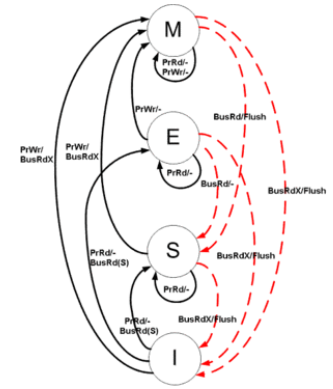



```
_____P1_____
lock(lock) {
  while(1) {
    old = ll(lock);
    if(old == 0)
      if(sc(lock, 1))
        return;
  }
}
```

```
_____P2_____
lock(lock) {
  while(1) {
    old = ll(lock);
    if(old == 0)
      if(sc(lock, 1))
        return;
  }
}
```

LLSC Lock Action Zone II

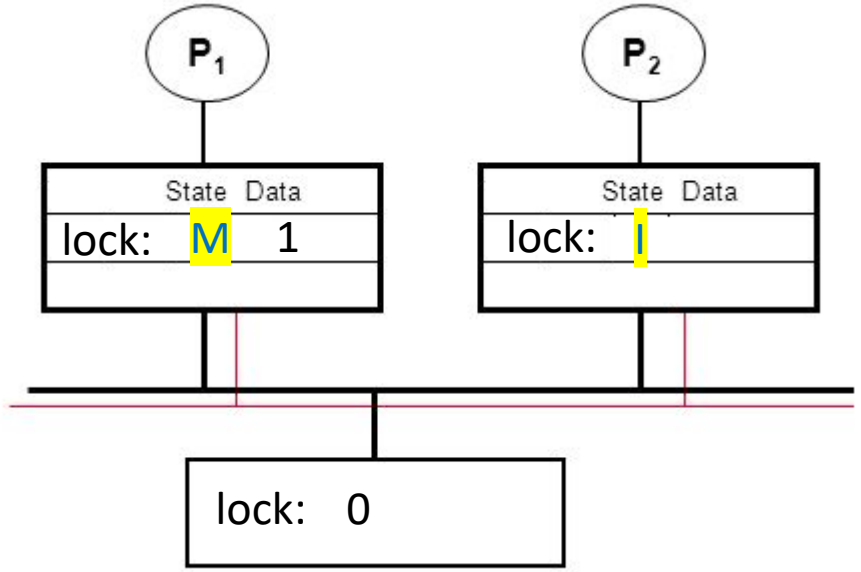


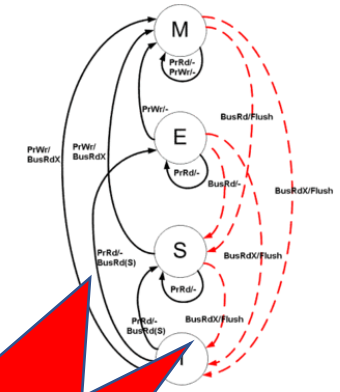



```
_____P1_____
lock(lock) {
  while(1) {
    old = ll(lock);
    if(old == 0)
      if(sc(lock, 1))
        return;
  }
}
```

```
_____P2_____
lock(lock) {
  while(1) {
    old = ll(lock);
    if(old == 0)
      if(sc(lock, 1))
        return;
  }
}
```

LLSC Lock Action Zone II





Store conditional fails

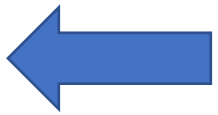
```
_____P1_____
lock(lock) {
  while(1) {
    old = ll(lock);
    if(old == 0)
      if(sc(lock, 1))
        return;
  }
}
```

```
_____P2_____
lock(lock) {
  while(1) {
    old = ll(lock);
    if(old == 0)
      if(sc(lock, 1))
        return;
  }
}
```

Implementing Locks with Test&set

```
int lock_value = 0;
int* lock = &lock_value;
```

```
Lock::Acquire() {
while (test&set(lock) == 1)
  ; //spin
}
```

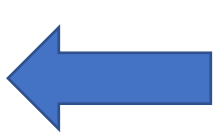


(test & set ~ CAS ~ LLSC)

Implementing Locks with Test&set

```
int lock_value = 0;
int* lock = &lock_value;
```

```
Lock::Acquire() {
while (test&set(lock) == 1)
  ; //spin
}
```



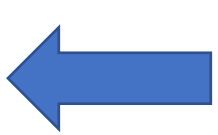
(test & set ~ CAS ~ LLSC)

```
Lock::Release() {
   *lock = 0;
}
```

Implementing Locks with Test&set

```
int lock_value = 0;
int* lock = &lock_value;
```

```
Lock::Acquire() {
while (test&set(lock) == 1)
  ; //spin
}
```



(test & set ~ CAS ~ LLSC)

```
Lock::Release() {
   *lock = 0;
}
```

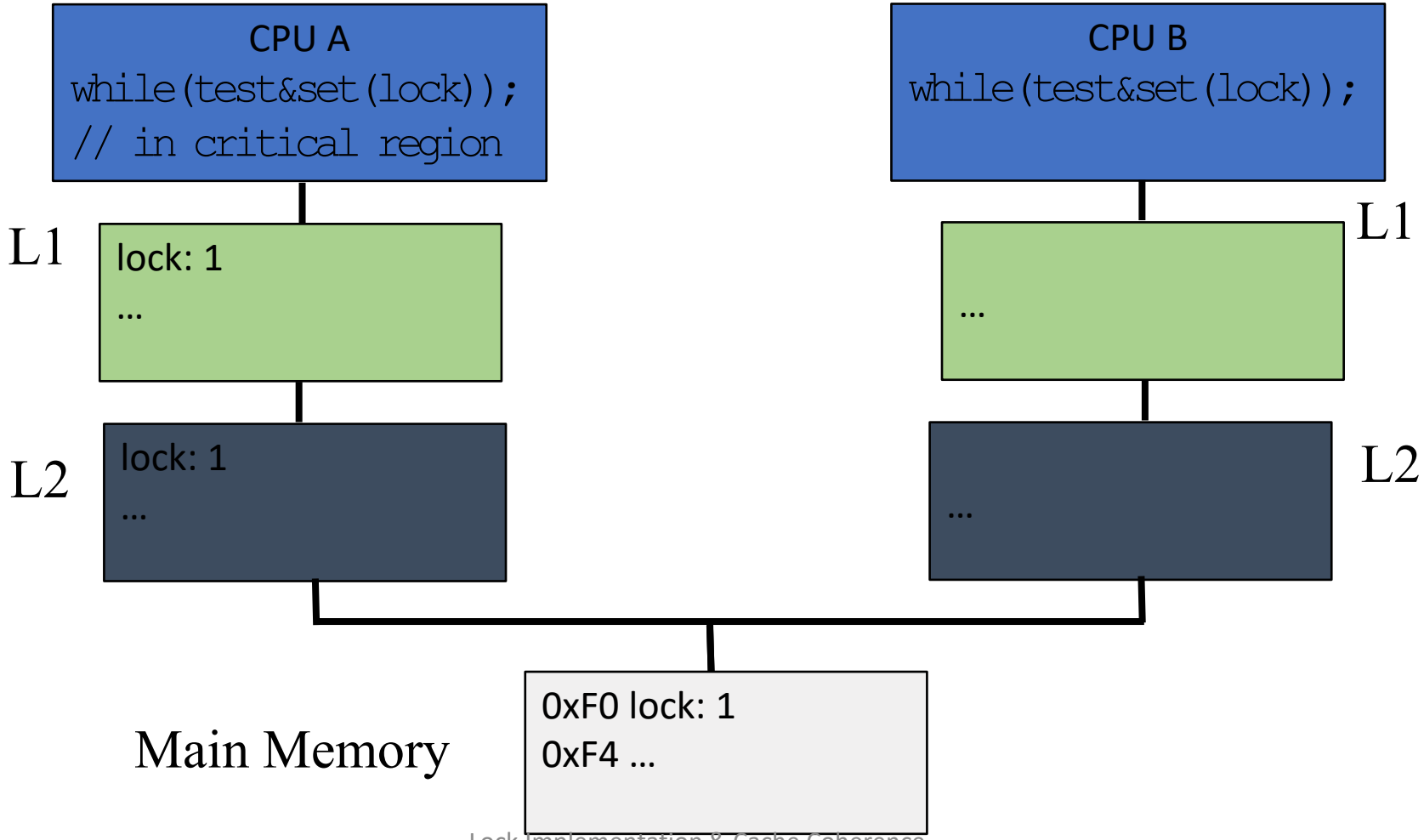
What is the problem with this?

- A. CPU usage B. Memory usage C. Lock::Acquire() latency
- D. Memory bus usage E. Does not work

Test & Set with Memory Hierarchies

Initially, lock already held by some other CPU—A, B busy-waiting

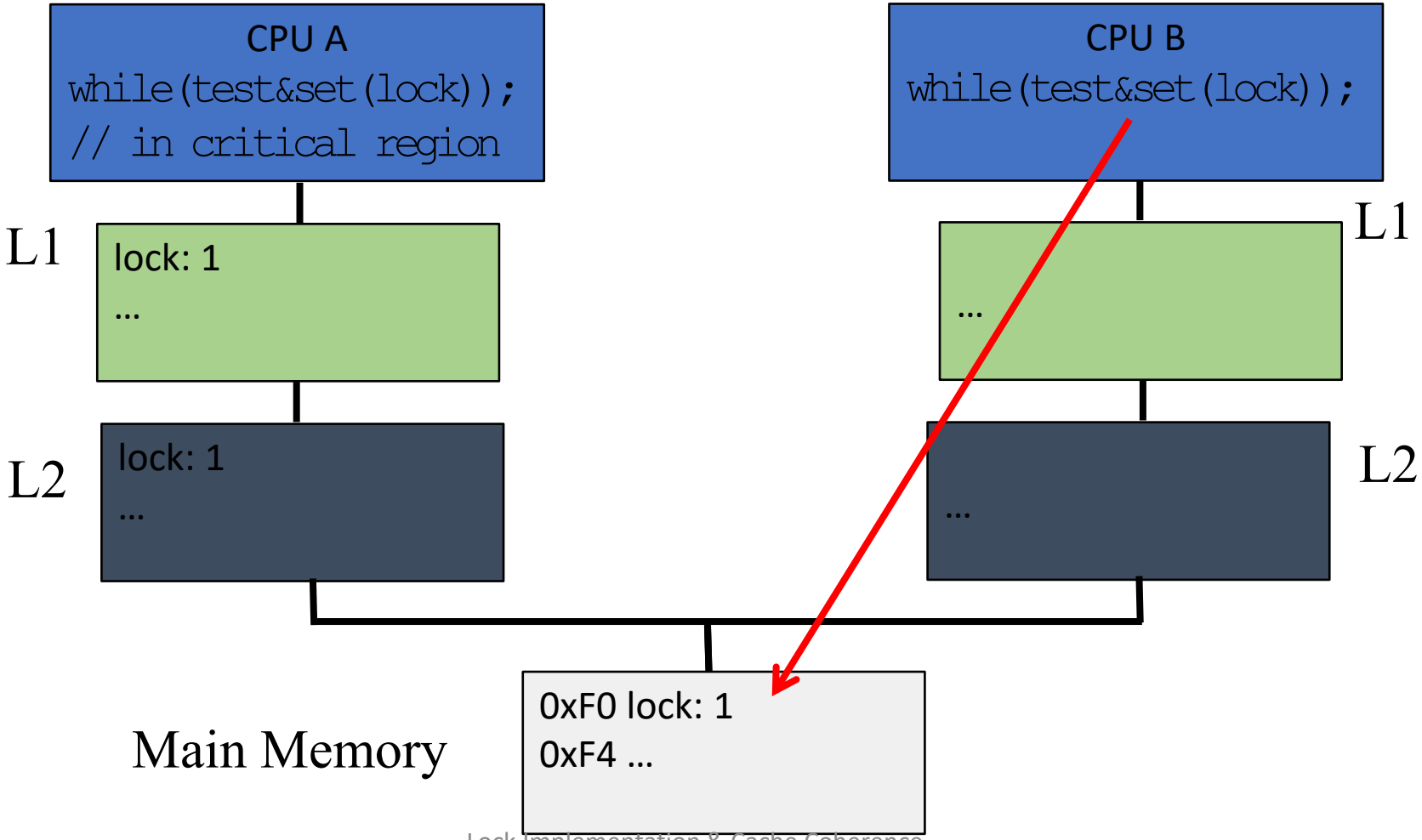
What happens to lock variable's cache line when different cpu's contend?



Test & Set with Memory Hierarchies

Initially, lock already held by some other CPU—A, B busy-waiting

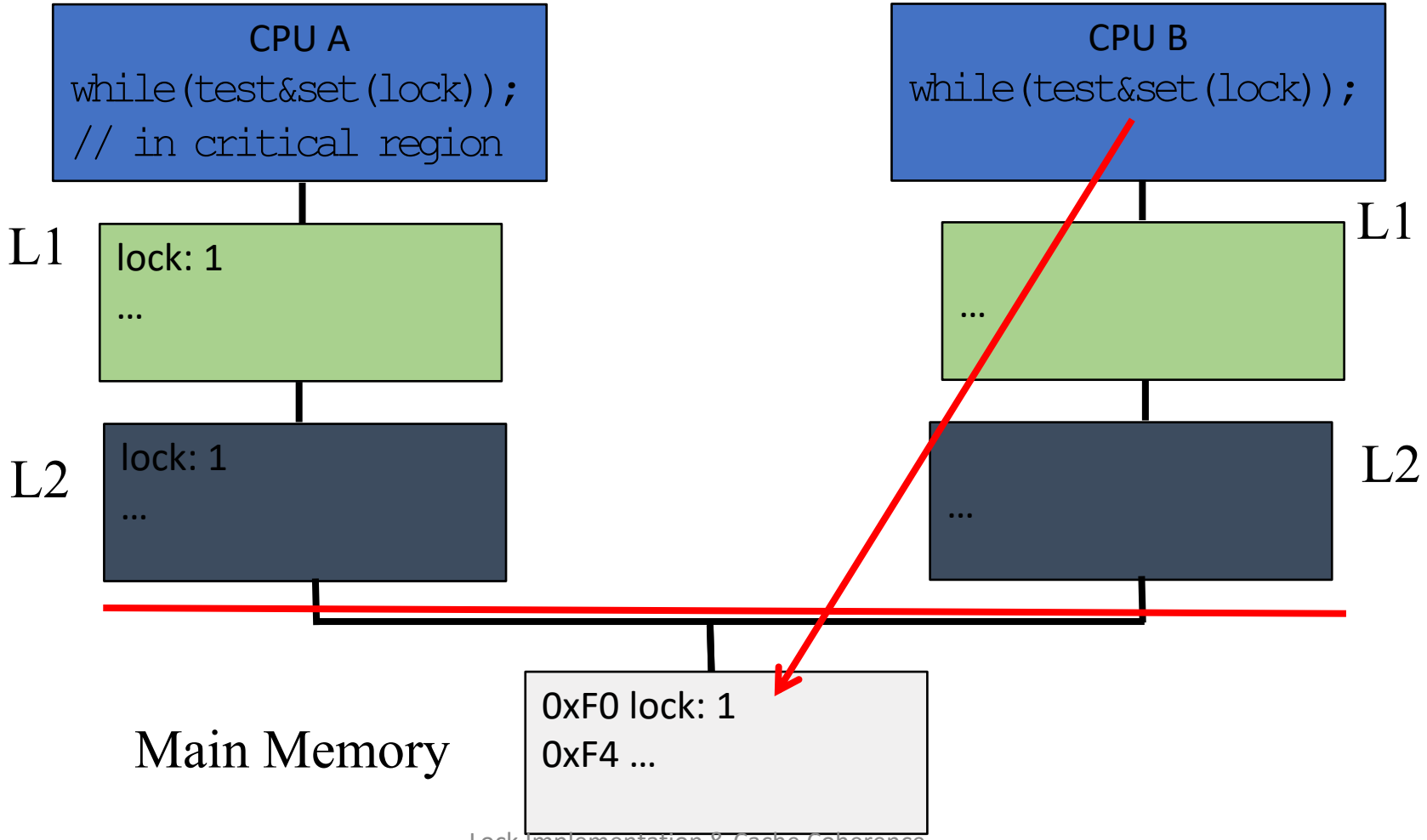
What happens to lock variable's cache line when different cpu's contend?



Test & Set with Memory Hierarchies

Initially, lock already held by some other CPU—A, B busy-waiting

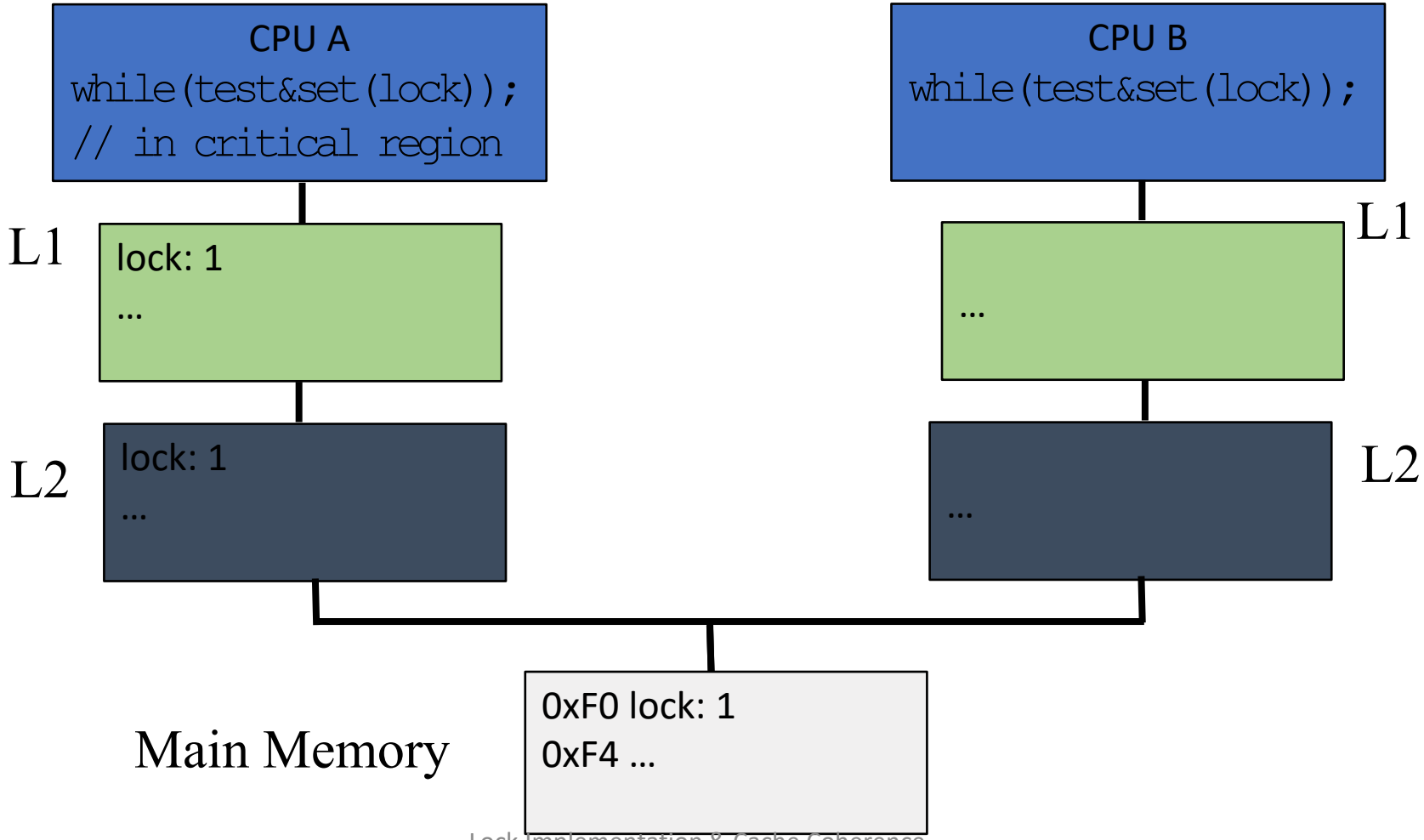
What happens to lock variable's cache line when different cpu's contend?



Test & Set with Memory Hierarchies

Initially, lock already held by some other CPU—A, B busy-waiting

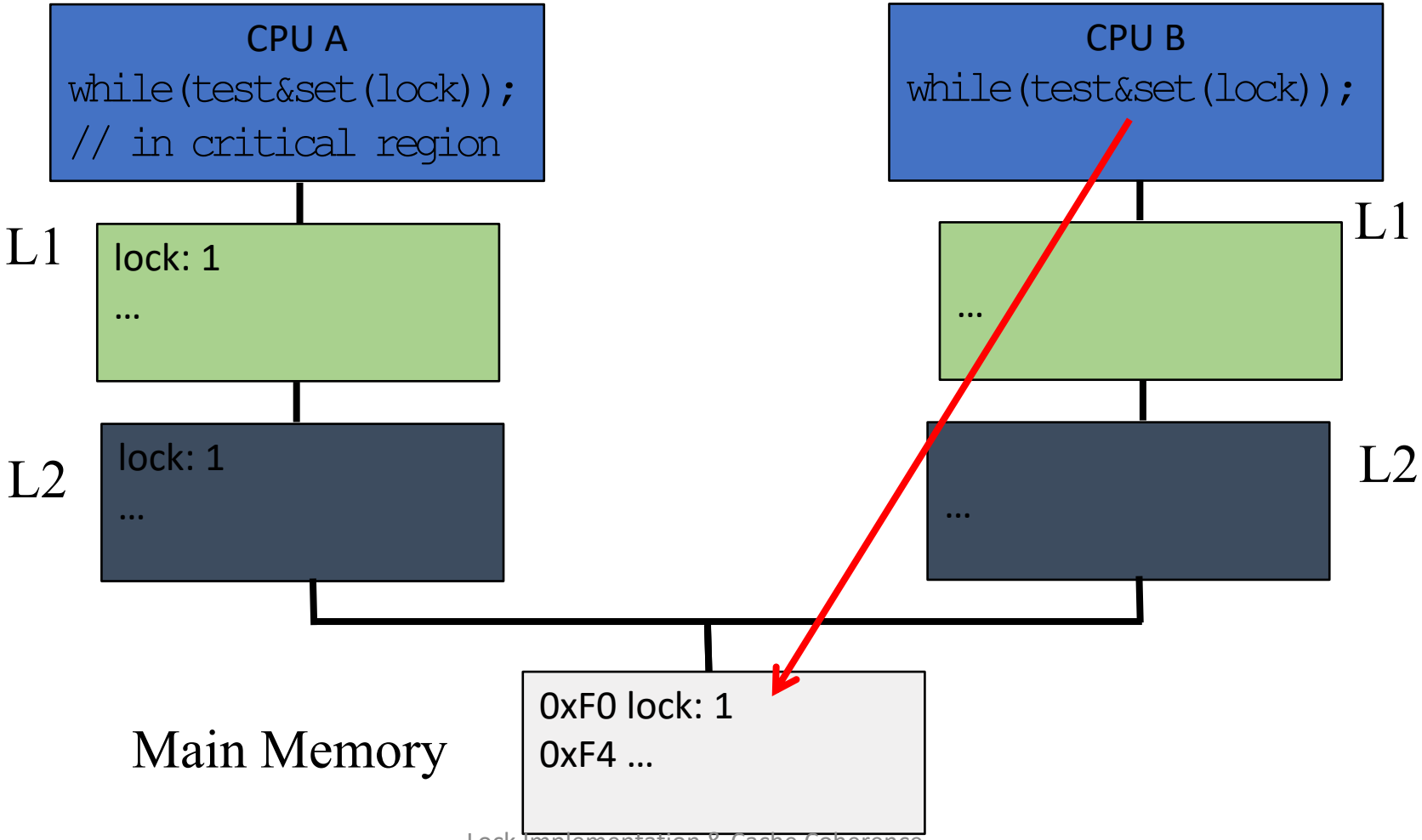
What happens to lock variable's cache line when different cpu's contend?



Test & Set with Memory Hierarchies

Initially, lock already held by some other CPU—A, B busy-waiting

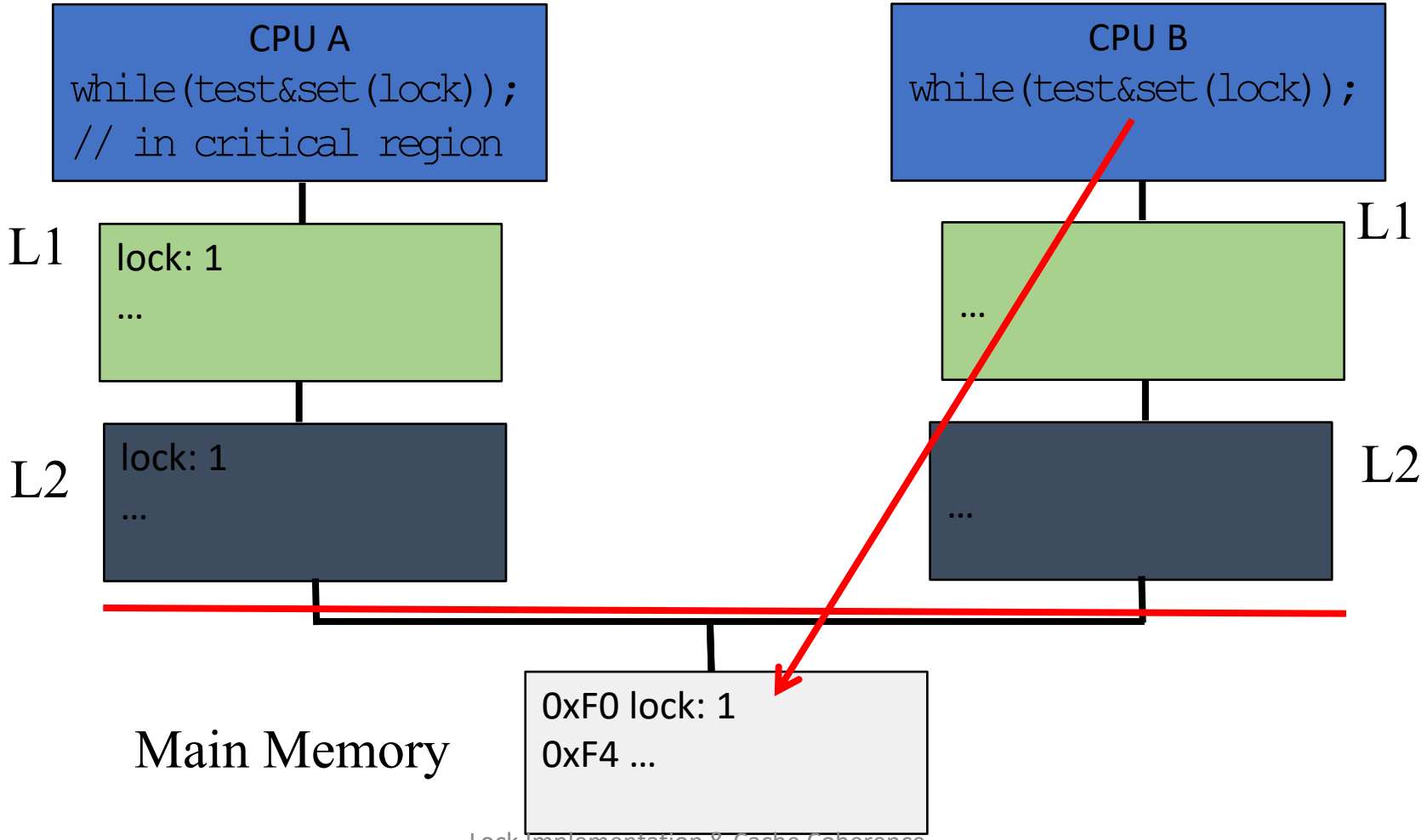
What happens to lock variable's cache line when different cpu's contend?



Test & Set with Memory Hierarchies

Initially, lock already held by some other CPU—A, B busy-waiting

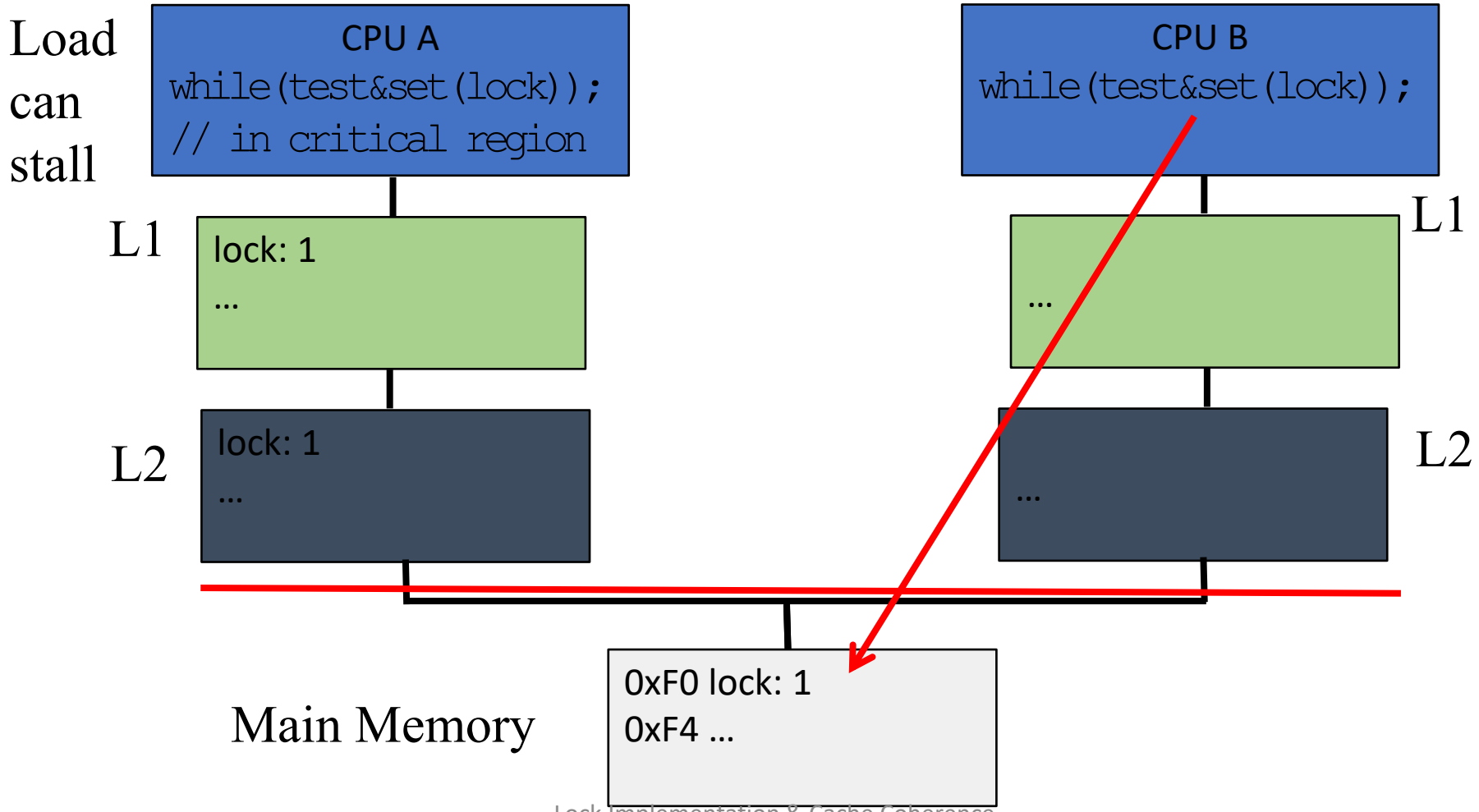
What happens to lock variable's cache line when different cpu's contend?



Test & Set with Memory Hierarchies

Initially, lock already held by some other CPU—A, B busy-waiting

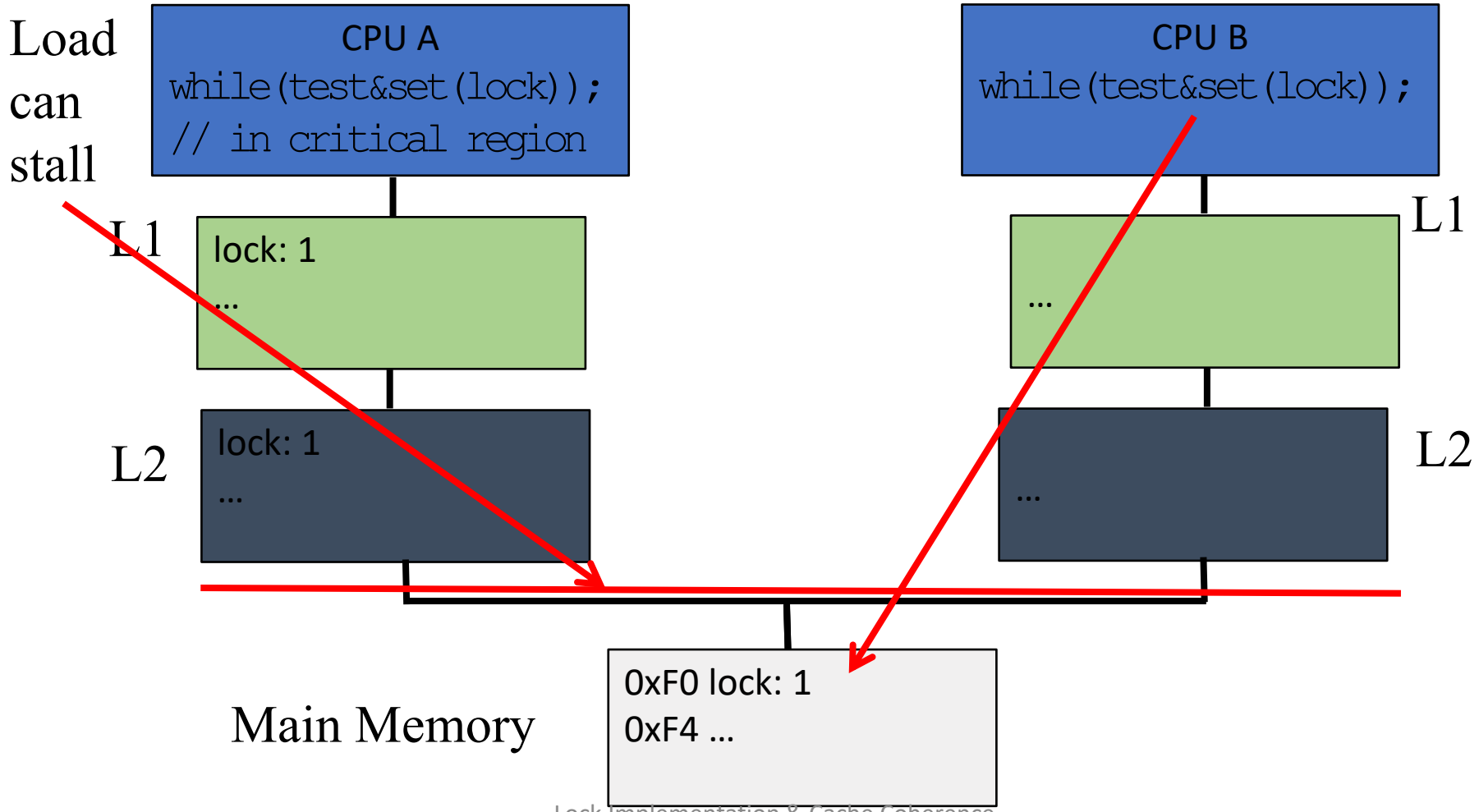
What happens to lock variable's cache line when different cpu's contend?



Test & Set with Memory Hierarchies

Initially, lock already held by some other CPU—A, B busy-waiting

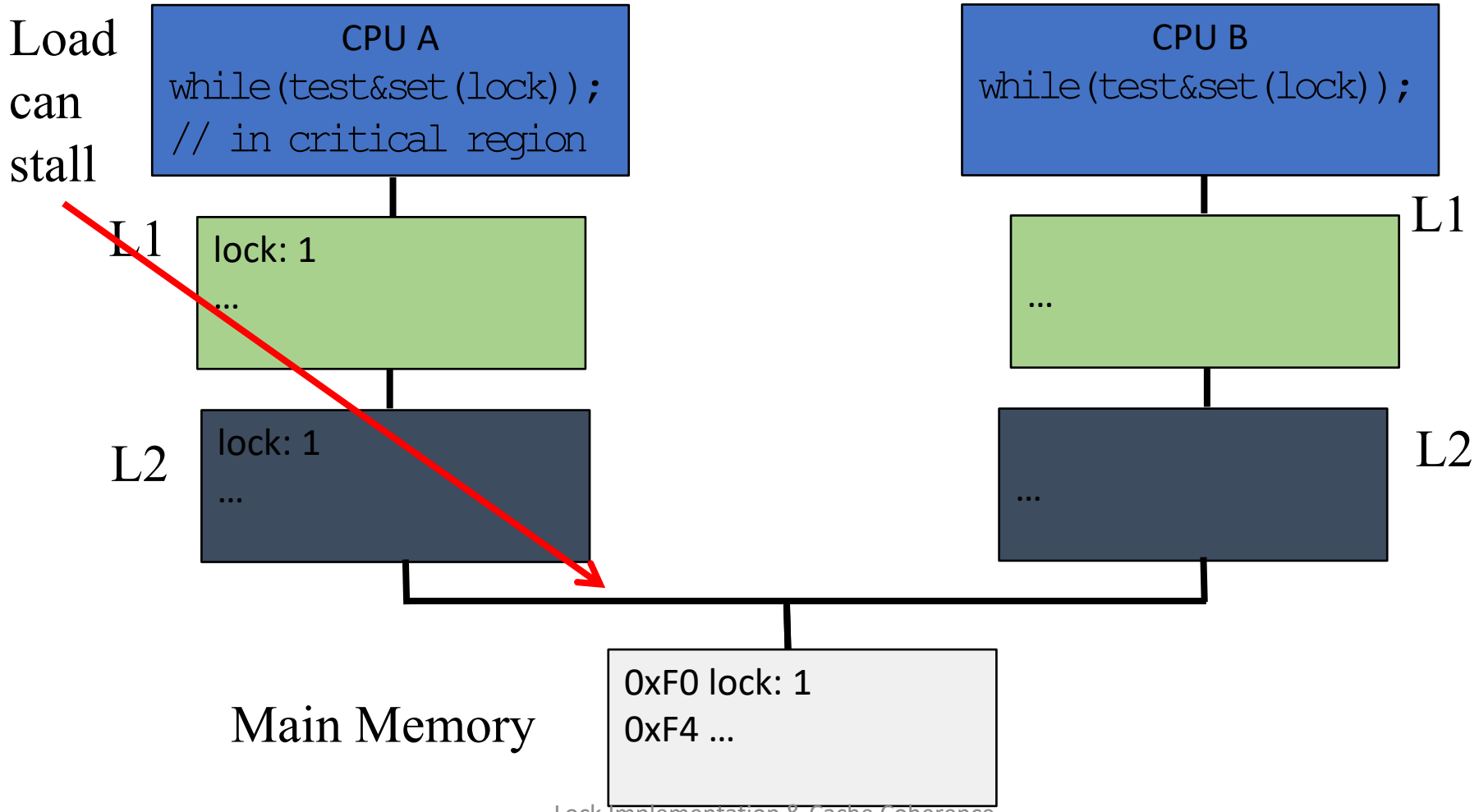
What happens to lock variable's cache line when different cpu's contend?



Test & Set with Memory Hierarchies

Initially, lock already held by some other CPU—A, B busy-waiting

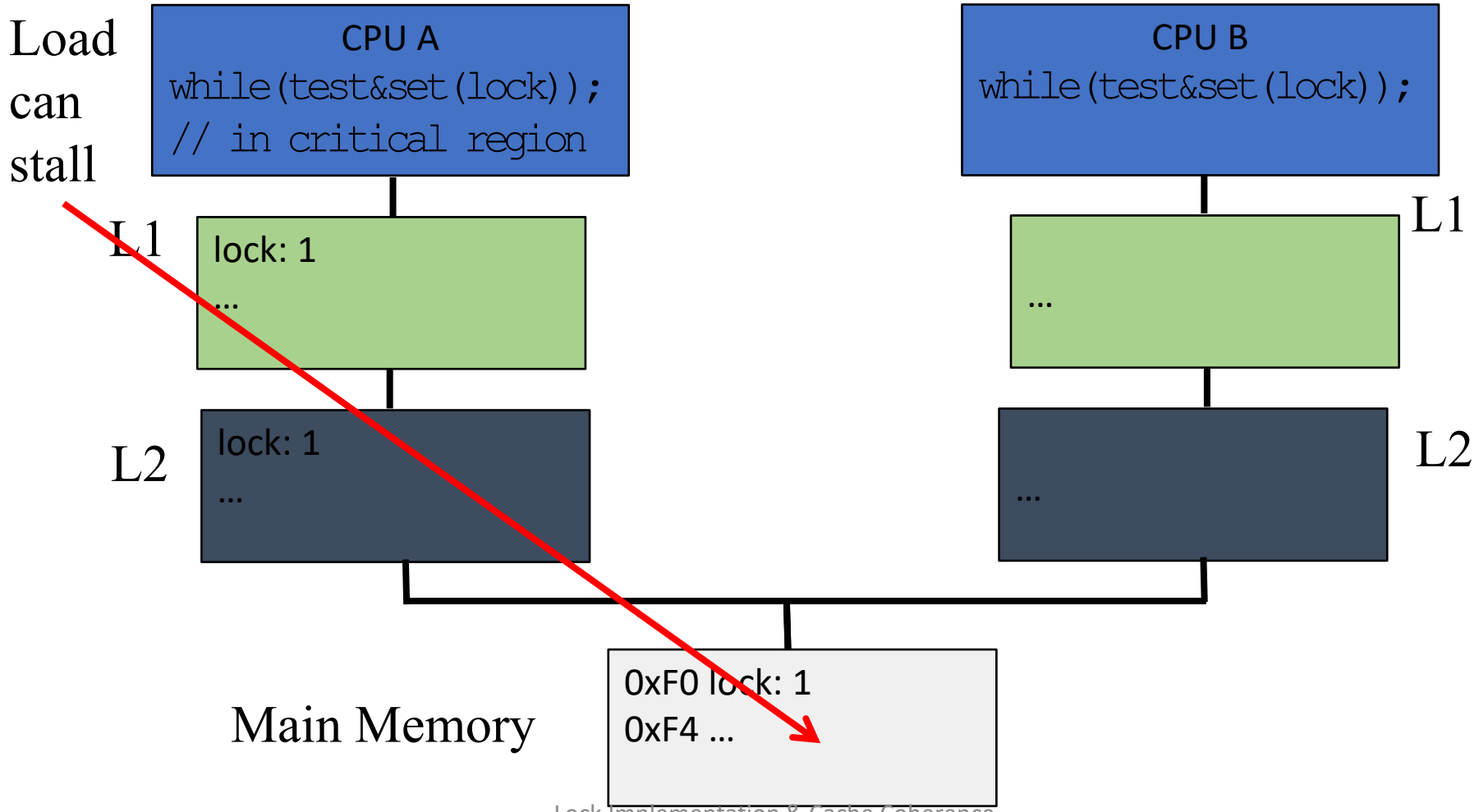
What happens to lock variable's cache line when different cpu's contend?



Test & Set with Memory Hierarchies

Initially, lock already held by some other CPU—A, B busy-waiting

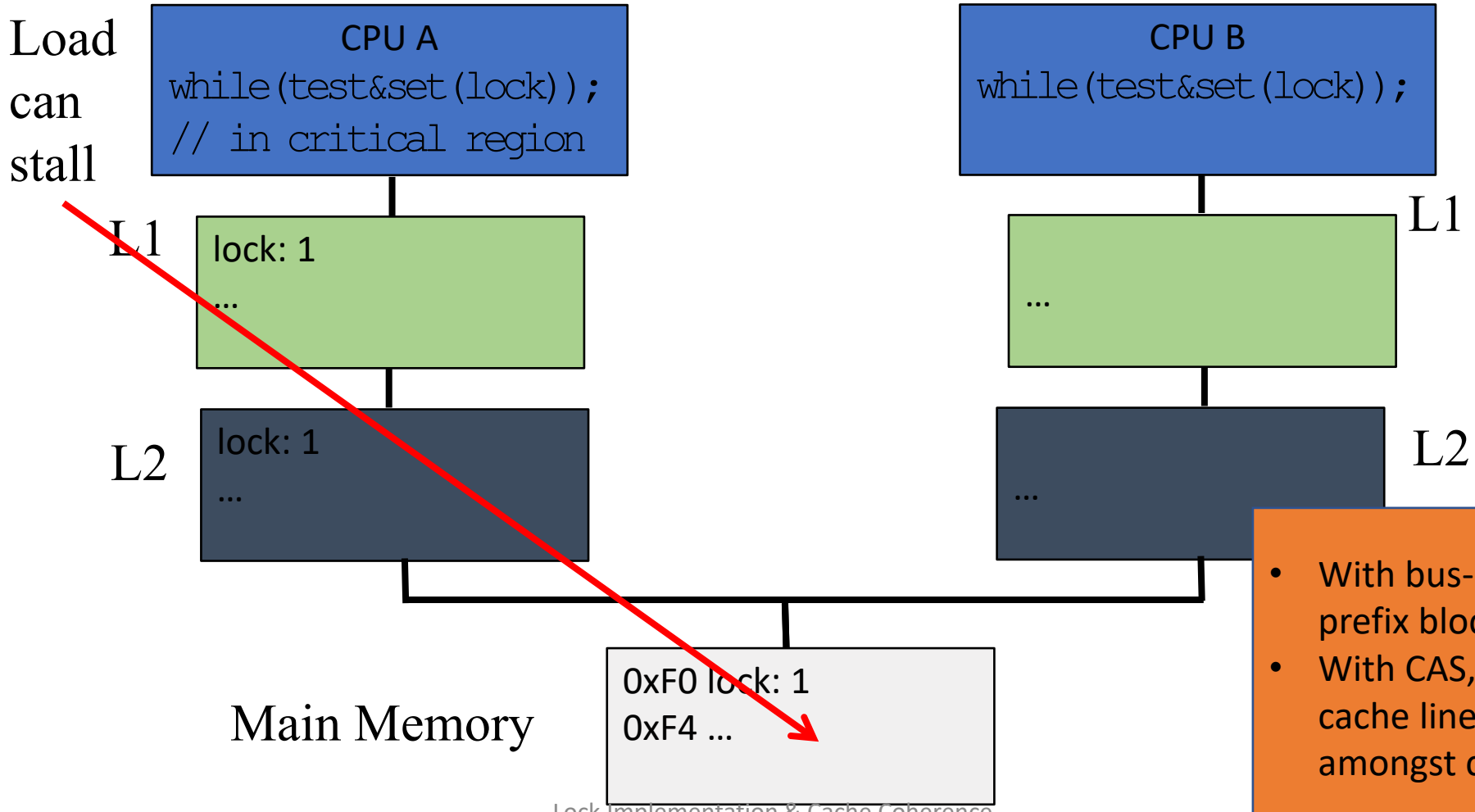
What happens to lock variable's cache line when different cpu's contend?



Test & Set with Memory Hierarchies

Initially, lock already held by some other CPU—A, B busy-waiting

What happens to lock variable's cache line when different cpu's contend?



- With bus-locking, lock prefix blocks *everyone*
- With CAS, LL-SC, cache line cache line "ping pongs" amongst contenders

TTS: Reducing busy wait contention

Test&Set

```
Lock::Acquire() {
while (test&set(lock) == 1);
}
```

Test&Test&Set

```
Lock::Acquire() {
while(1) {
  while (*lock == 1) ; // spin just reading
  if (test&set(lock) == 0)  break;
}
```

Busy-wait on in-memory copy

Busy-wait on cached copy

```
Lock::Release() {
    *lock = 0;
}
```

```
Lock::Release() {
*lock = 0;
}
```

TTS: Reducing busy wait contention

Test&Set

```
Lock::Acquire() {
while (test&set(lock) == 1);
}
```

Busy-wait on in-memory copy

```
Lock::Release() {
    *lock = 0;
}
```

Test&Test&Set

```
Lock::Acquire() {
while(1) {
  while (*lock == 1) ; // spin just reading
  if (test&set(lock) == 0)  break;
}
```

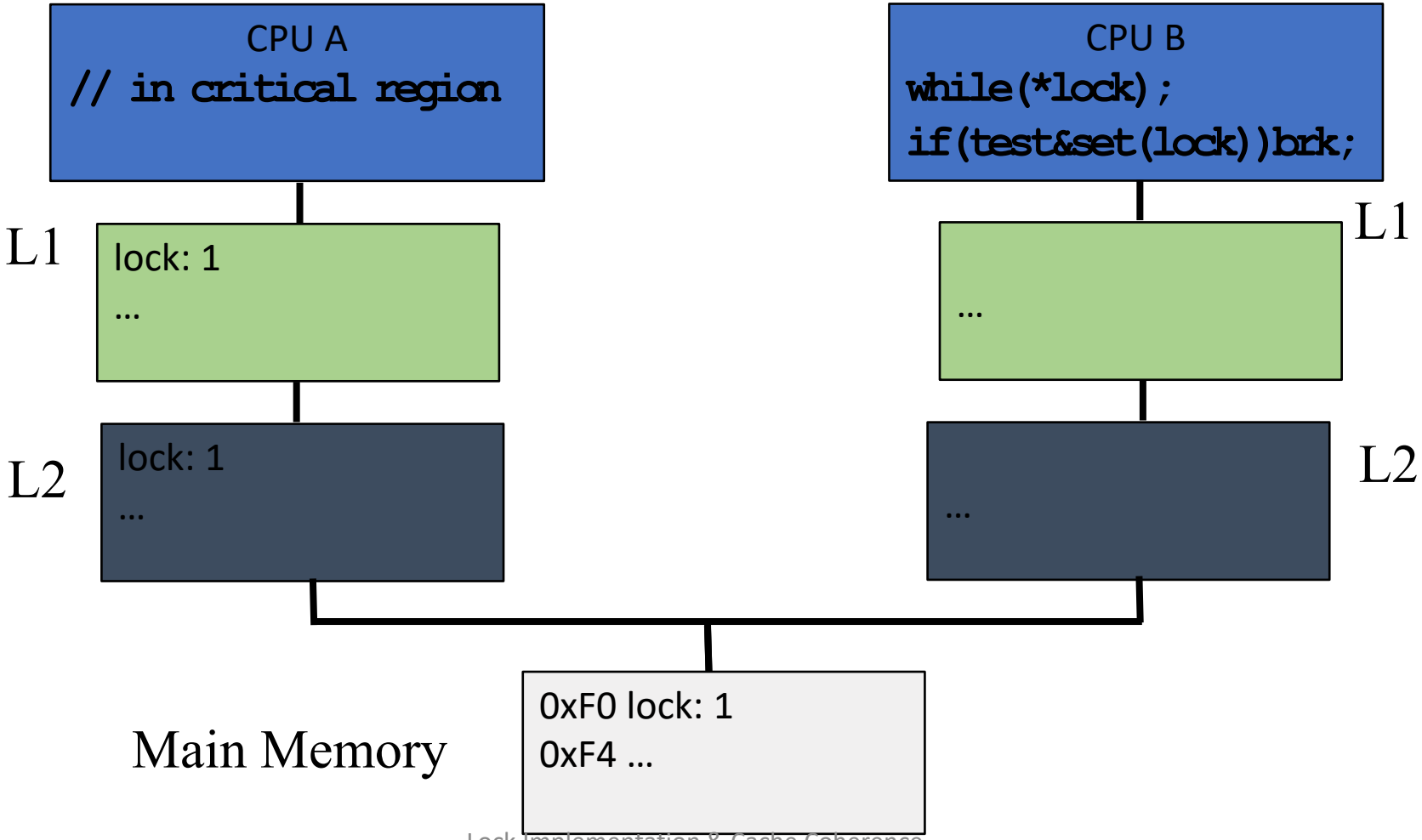
Busy-wait on cached copy

```
Lock::Release() {
*lock = 0;
}
```

- What is the problem with this?
  - A. CPU usage B. Memory usage C. Lock::Acquire() latency
  - D. Memory bus usage E. Does not work

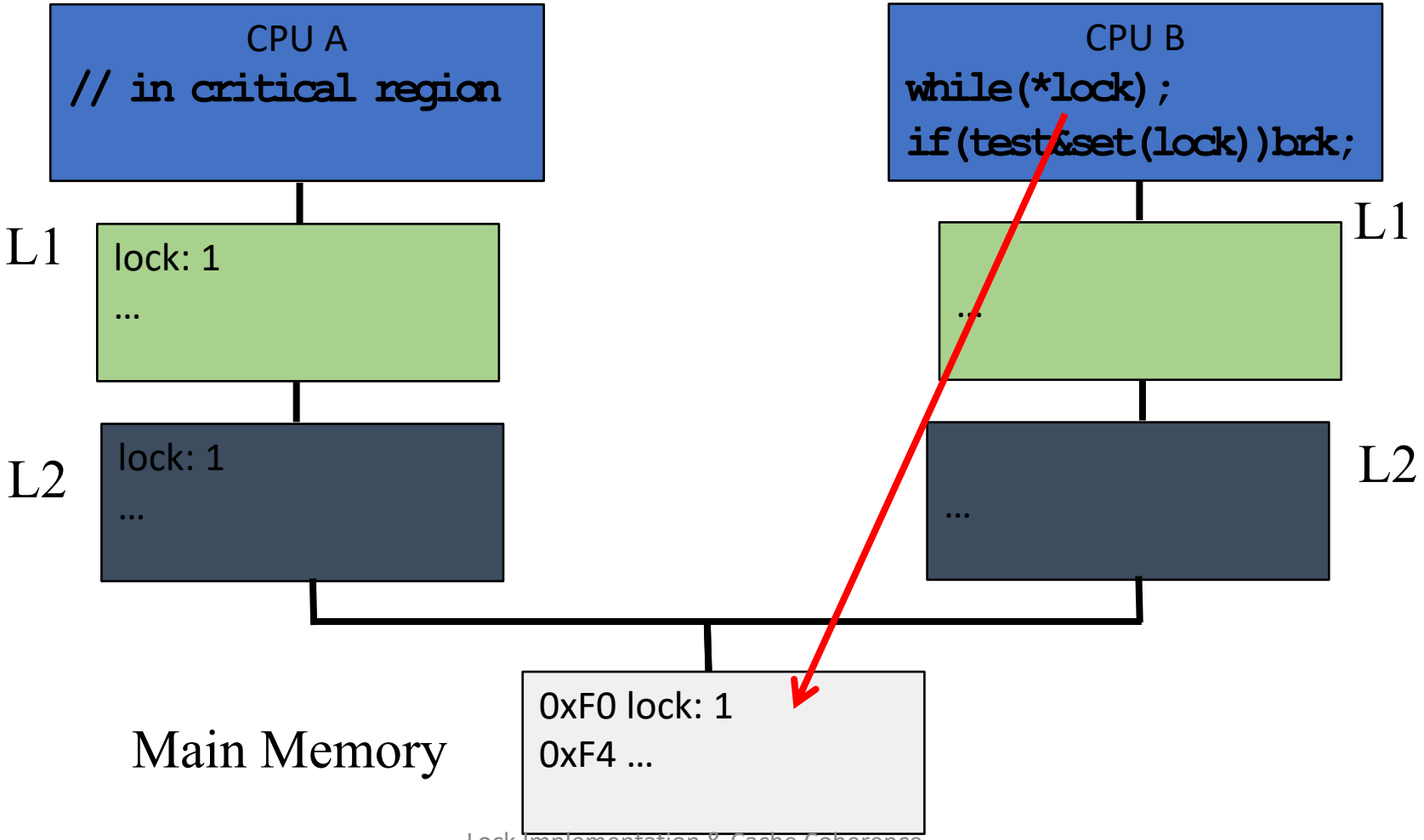
Test & Test & Set with Memory Hierarchies

What happens to lock variable's cache line when different cpu's contend for the same lock?



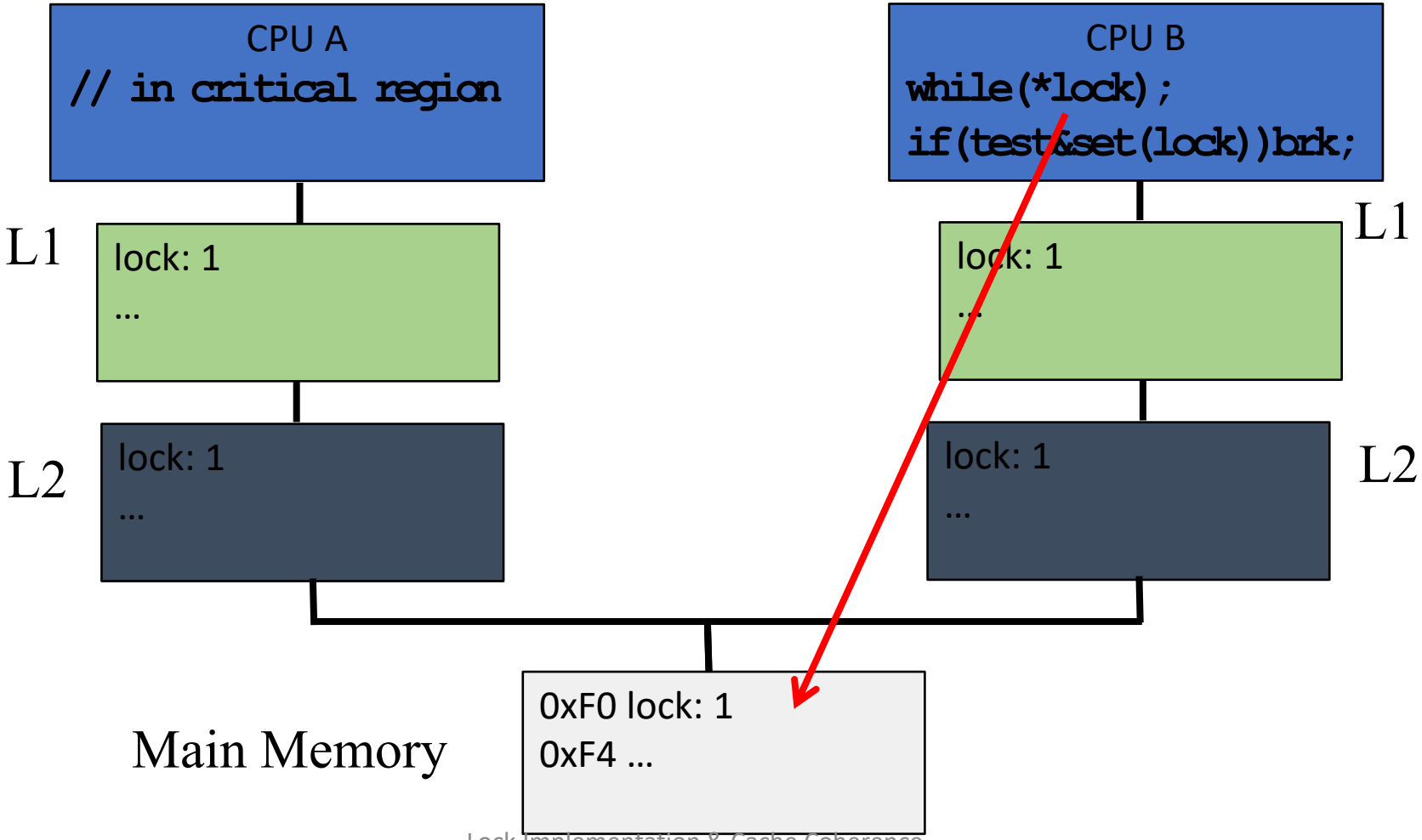
Test & Test & Set with Memory Hierarchies

What happens to lock variable's cache line when different cpu's contend for the same lock?



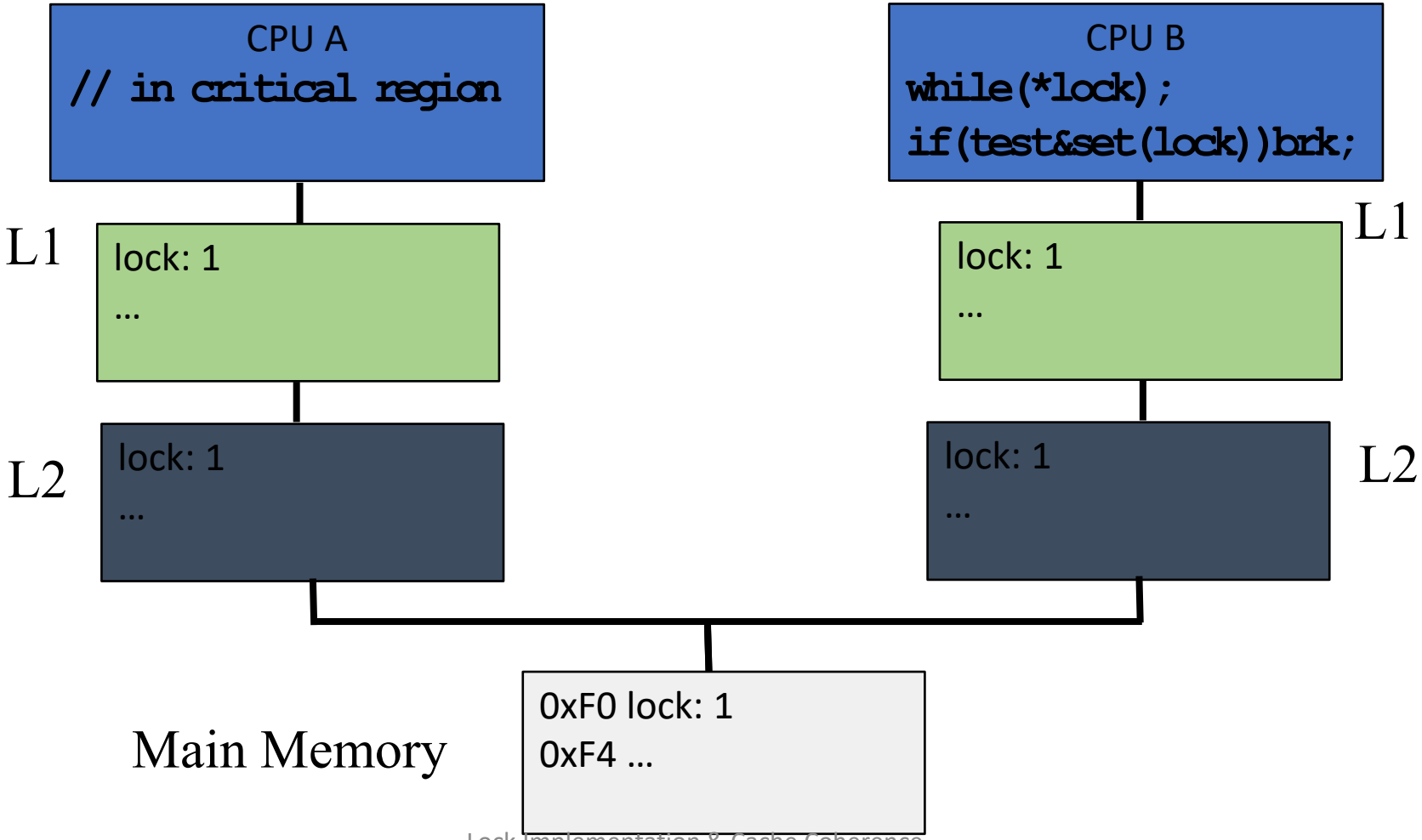
Test & Test & Set with Memory Hierarchies

What happens to lock variable's cache line when different cpu's contend for the same lock?



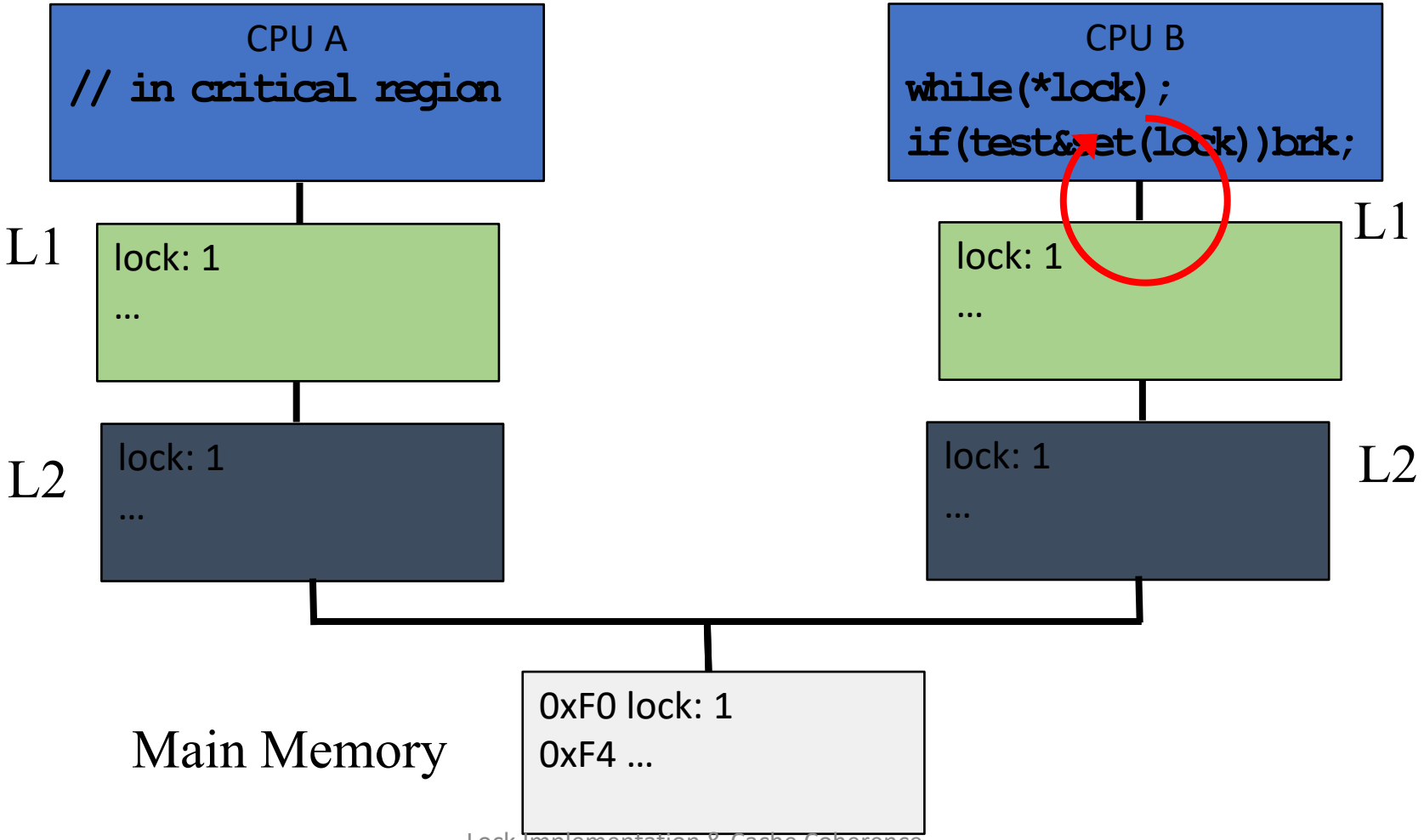
Test & Test & Set with Memory Hierarchies

What happens to lock variable's cache line when different cpu's contend for the same lock?



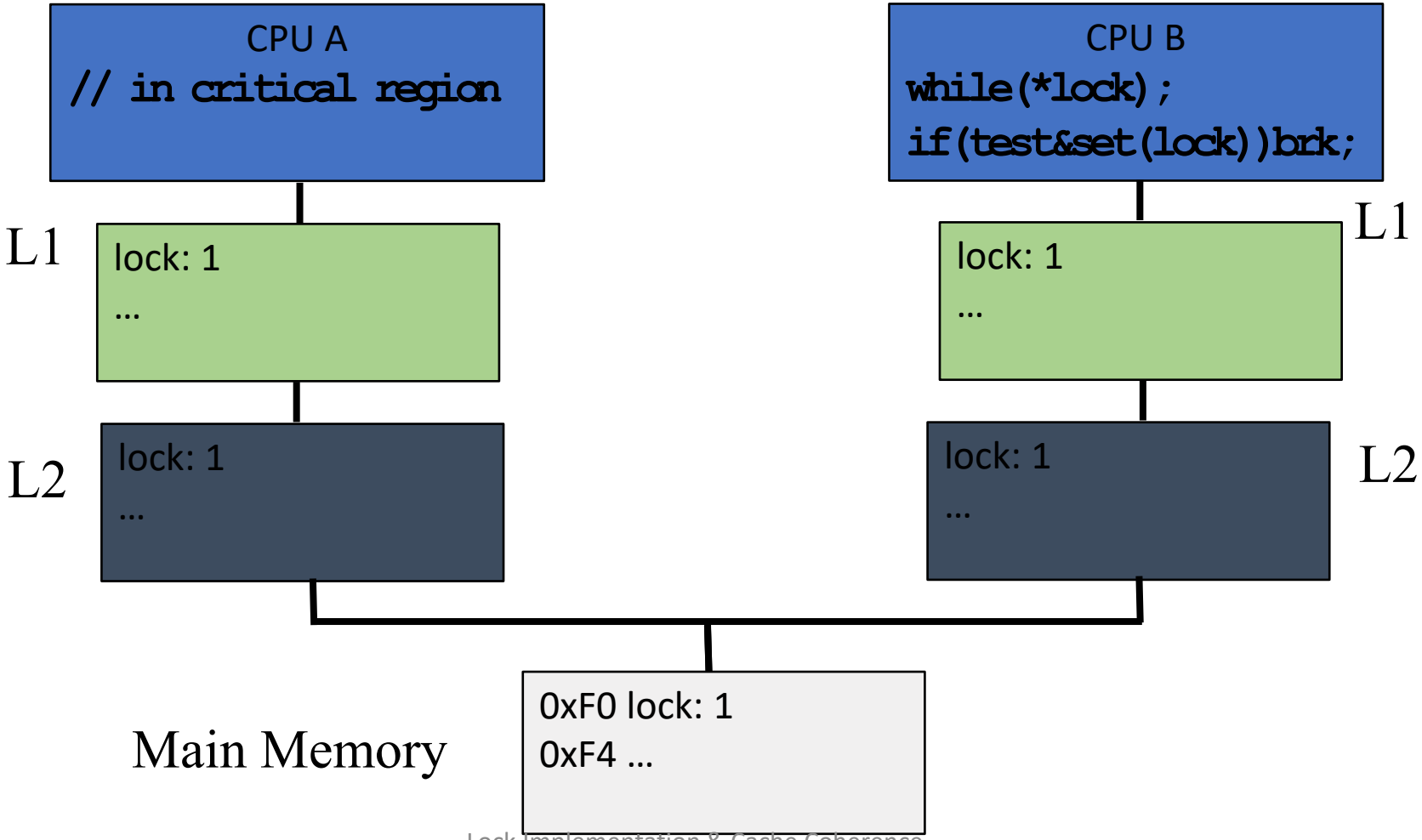
Test & Test & Set with Memory Hierarchies

What happens to lock variable's cache line when different cpu's contend for the same lock?



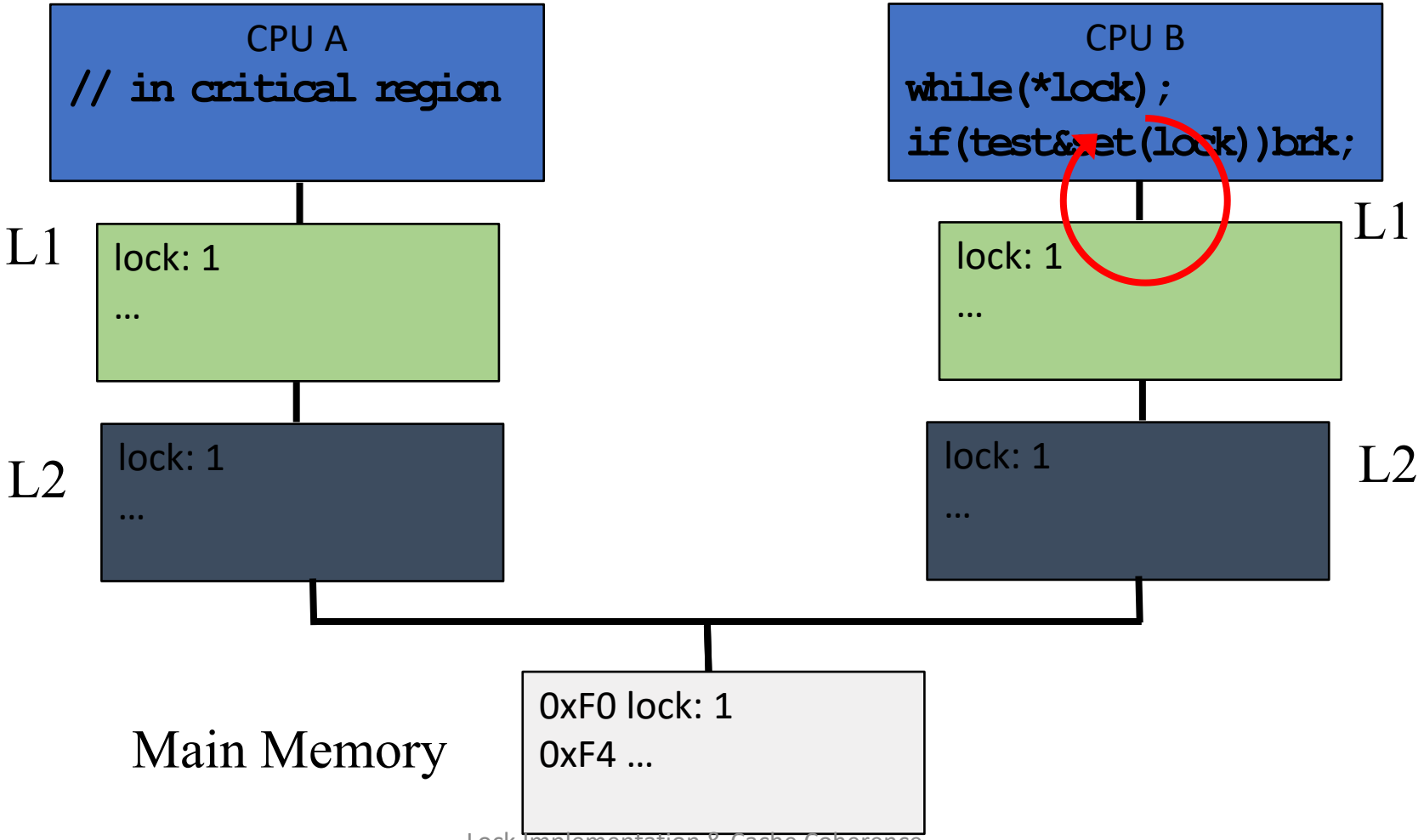
Test & Test & Set with Memory Hierarchies

What happens to lock variable's cache line when different cpu's contend for the same lock?



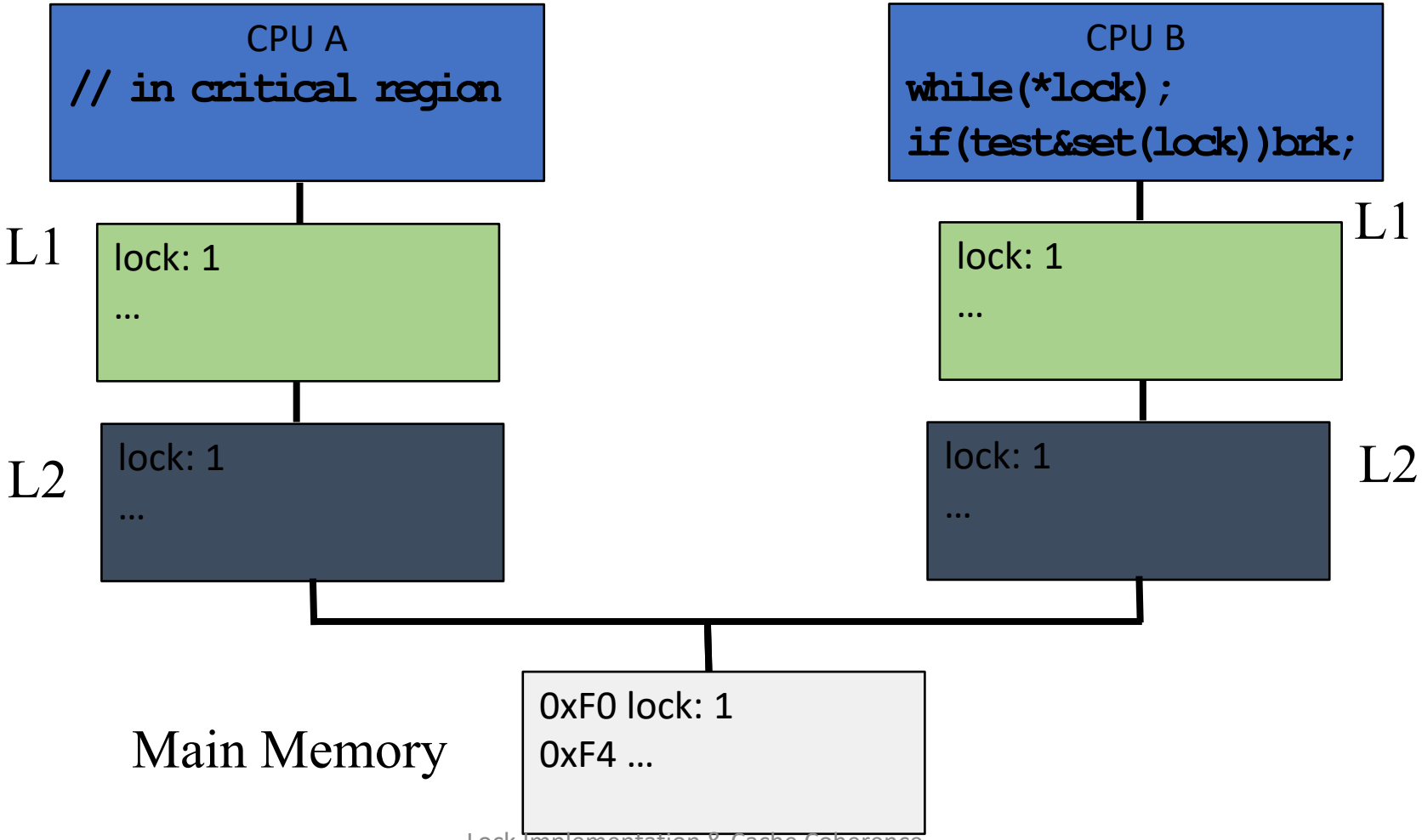
Test & Test & Set with Memory Hierarchies

What happens to lock variable's cache line when different cpu's contend for the same lock?



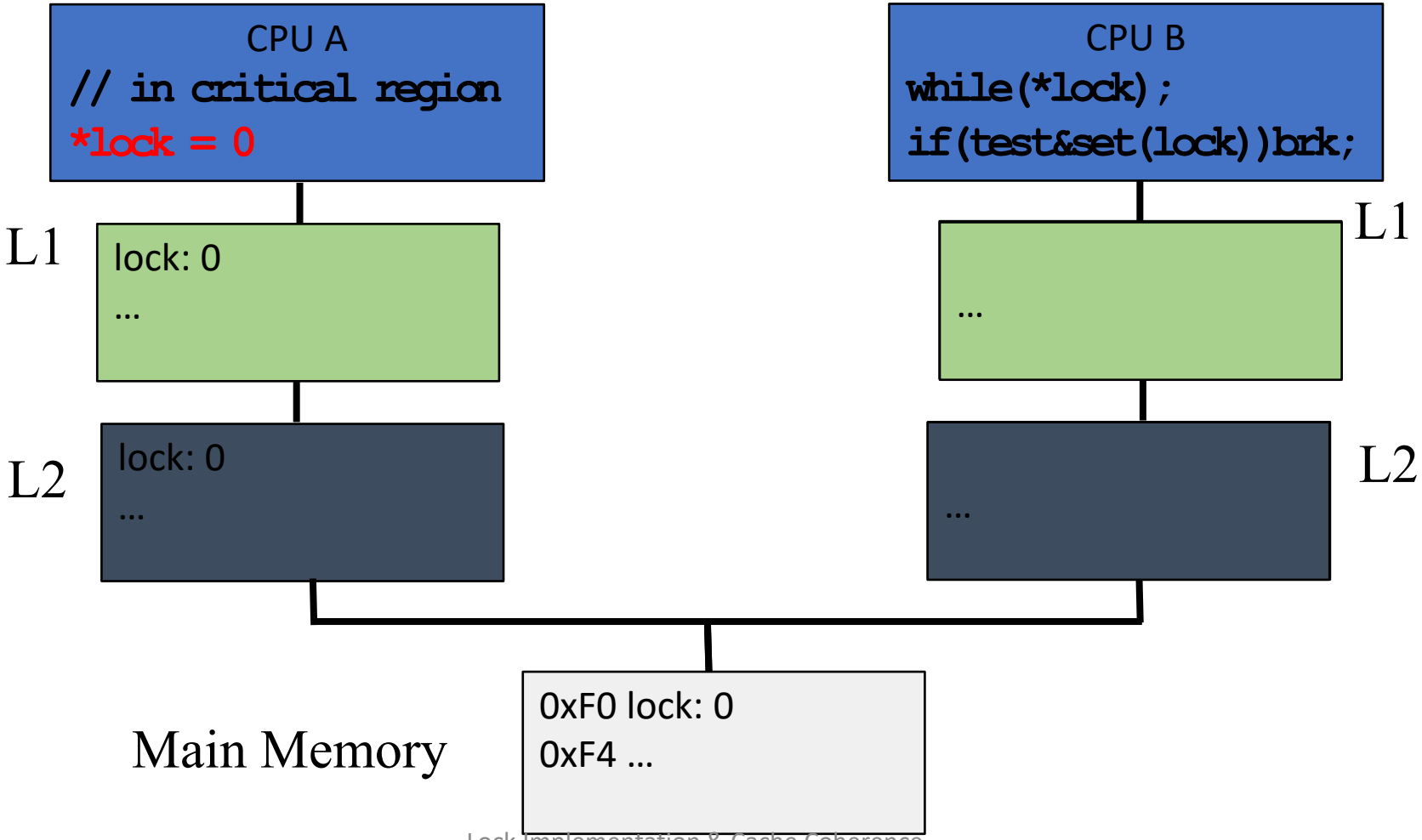
Test & Test & Set with Memory Hierarchies

What happens to lock variable's cache line when different cpu's contend for the same lock?



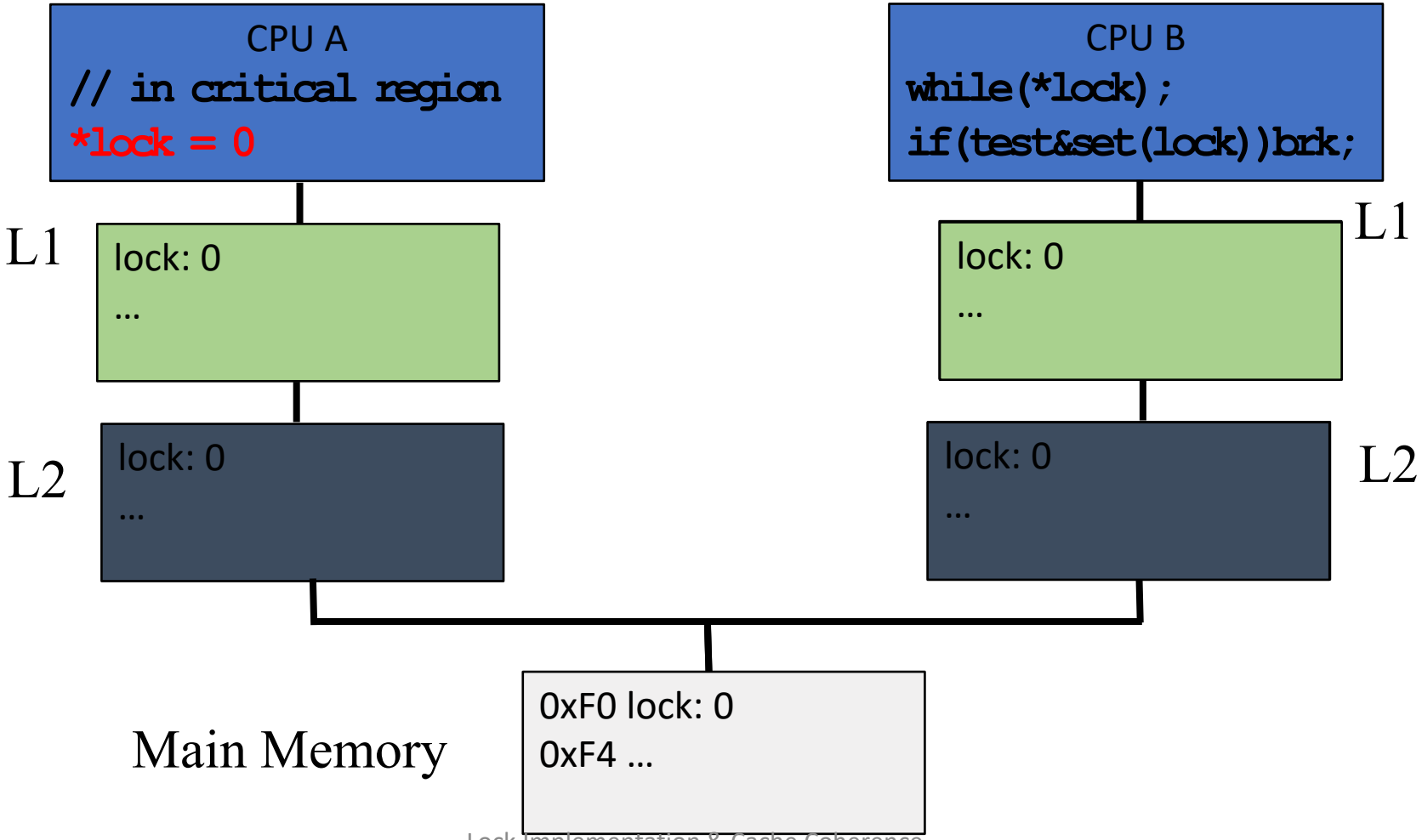
Test & Test & Set with Memory Hierarchies

What happens to lock variable's cache line when different cpu's contend for the same lock?



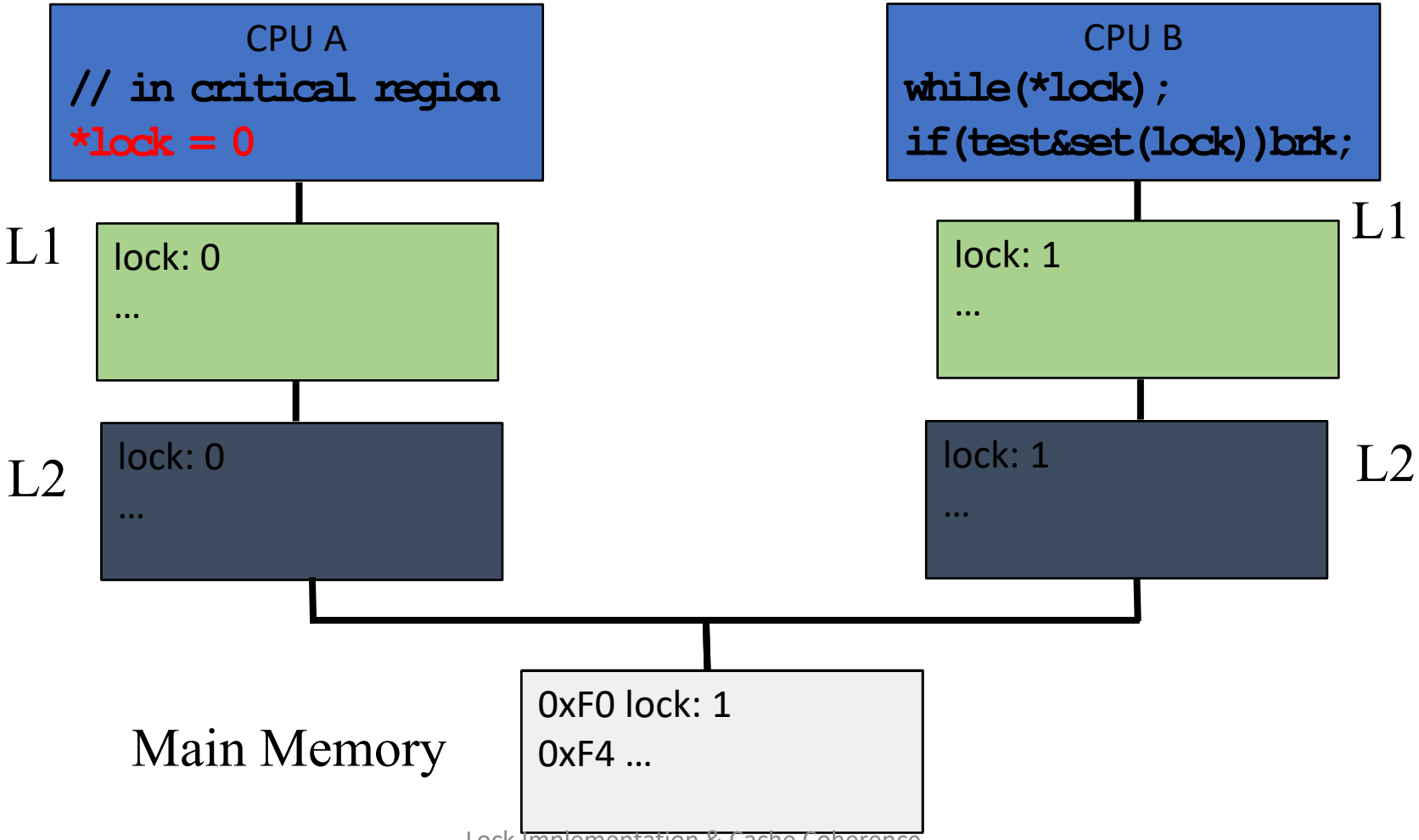
Test & Test & Set with Memory Hierarchies

What happens to lock variable's cache line when different cpu's contend for the same lock?



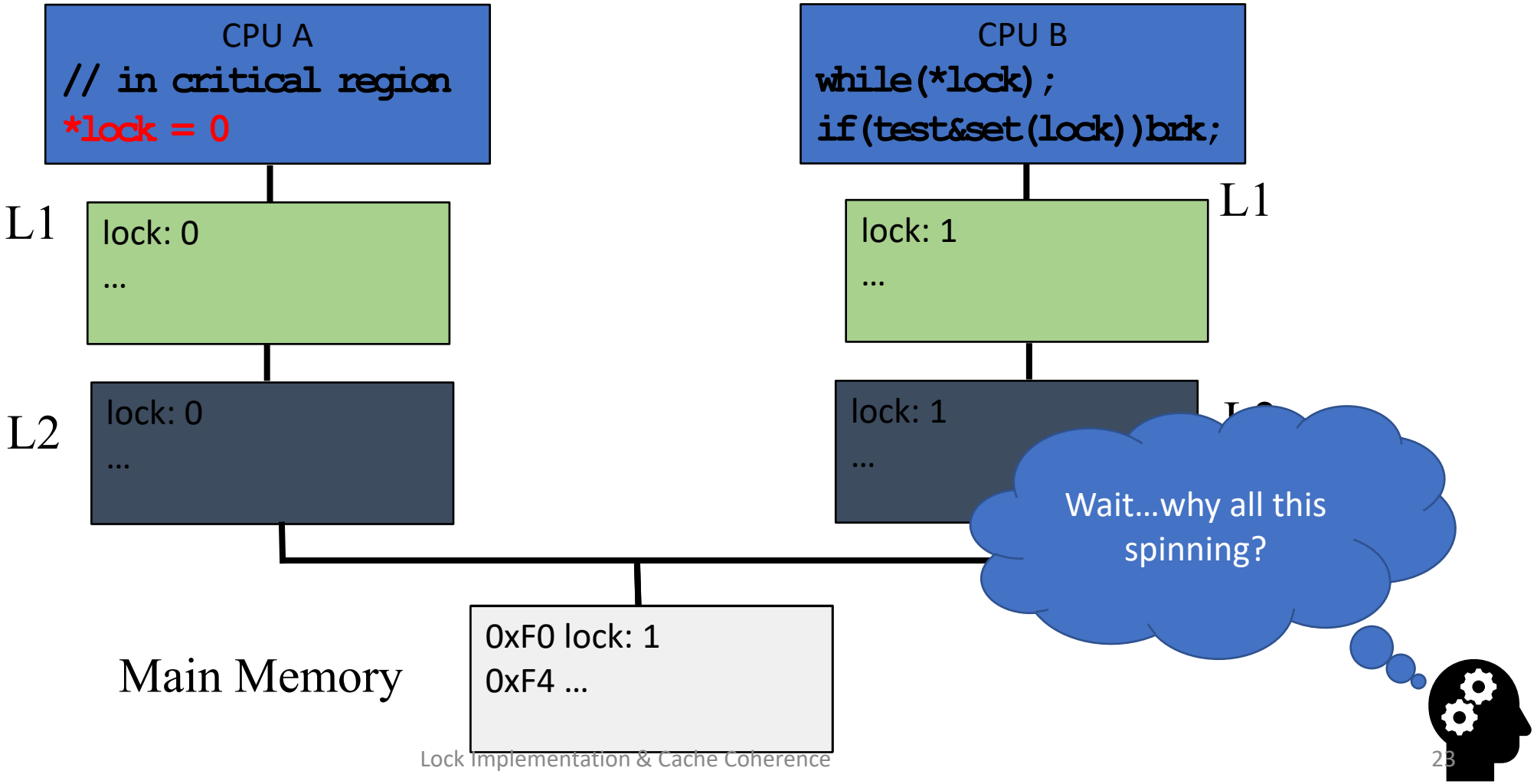
Test & Test & Set with Memory Hierarchies

What happens to lock variable's cache line when different cpu's contend for the same lock?



Test & Test & Set with Memory Hierarchies

What happens to lock variable's cache line when different cpu's contend for the same lock?



How can we improve over busy-wait?

```
Lock::Acquire() {
while(1) {
  while (*lock == 1) ; // spin just reading
  if (test&set(lock) == 0)  break;
}
```

Mutex

- Same abstraction as spinlock
- But is a “blocking” primitive
  - Lock available → same behavior
  - Lock held → yield/block
- Many ways to yield
- Simplest case of semaphore

```
void cm3_lock(u8_t* M) {
  u8_t LockedIn = 0;
  do {
   if (__LDREXB(Mutex) == 0) {
     // unlocked: try to obtain lock
     if ( __STREXB(1, Mutex)) { // got lock
       __CLREX(); // remove __LDREXB() lock
       LockedIn = 1;
     }
     else task_yield(); // give away cpu
   }
   else task_yield();   // give away cpu
} while(!LockedIn);
```

Mutex

- Same abstraction as spinlock
- But is a “blocking” primitive
  - Lock available → same behavior
  - Lock held → yield/block
- Many ways to yield
- Simplest case of semaphore

```
void cm3_lock(u8_t* M) {
  u8_t LockedIn = 0;
  do {
   if (__LDREXB(Mutex) == 0) {
     // unlocked: try to obtain lock
     if ( __STREXB(1, Mutex)) { // got lock
       __CLREX(); // remove __LDREXB() lock
       LockedIn = 1;
     }
     else task_yield(); // give away cpu
   }
   else task_yield();   // give away cpu
} while(!LockedIn);
```

- Is it better to use a spinlock or mutex on a uni-processor?

Mutex

- Same abstraction as spinlock
- But is a “blocking” primitive
  - Lock available → same behavior
  - Lock held → yield/block
- Many ways to yield
- Simplest case of semaphore

```
void cm3_lock(u8_t* M) {
  u8_t LockedIn = 0;
  do {
   if (__LDREXB(Mutex) == 0) {
     // unlocked: try to obtain lock
     if ( __STREXB(1, Mutex)) { // got lock
       __CLREX(); // remove __LDREXB() lock
       LockedIn = 1;
     }
     else task_yield(); // give away cpu
   }
   else task_yield();   // give away cpu
} while(!LockedIn);
```

- Is it better to use a spinlock or mutex on a uni-processor?
- Is it better to use a spinlock or mutex on a multi-processor?

Mutex

- Same abstraction as spinlock
- But is a “blocking” primitive
  - Lock available → same behavior
  - Lock held → yield/block
- Many ways to yield
- Simplest case of semaphore

```
void cm3_lock(u8_t* M) {
  u8_t LockedIn = 0;
  do {
   if (__LDREXB(Mutex) == 0) {
     // unlocked: try to obtain lock
     if ( __STREXB(1, Mutex)) { // got lock
       __CLREX(); // remove __LDREXB() lock
       LockedIn = 1;
     }
     else task_yield(); // give away cpu
   }
   else task_yield();   // give away cpu
} while(!LockedIn);
```

- Is it better to use a spinlock or mutex on a uni-processor?
- Is it better to use a spinlock or mutex on a multi-processor?
- How do you choose between spinlock/mutex on a multi-processor?

Priority Inversion

A(prio-0) → enter(l);

B(prio-100) → enter(l); → must wait.

Solution?

Priority Inversion

A(prio-0) → enter(l);

B(prio-100) → enter(l); → must wait.

Solution?

Priority inheritance: A runs at B's priority
MARS pathfinder failure:
http://wiki.csie.ncku.edu.tw/embedded/priority-inversion-on-Mars.pdf

Other ideas?