

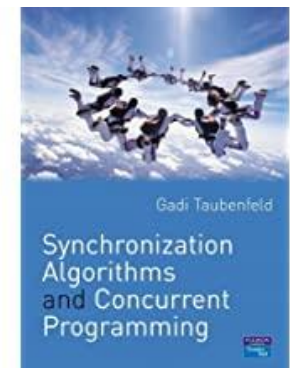
Synchronization Cache Coherence

Chris Rossbach

CS378h

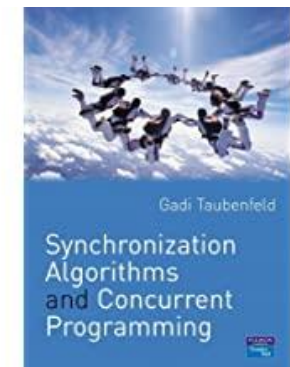
Today

- Questions?
- Administrivia
 - Lab 1 due soon
- Material for the day
 - Cache coherence
 - Lock implementation
 - Blocking synchronization
- Acknowledgements
 - Thanks to Gadi Taubenfeld: I borrowed from some of his slides on barriers
 - Gadi's materials on synchronization are a great resource



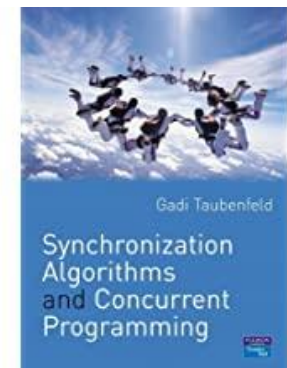
Today

- Questions?
- Administrivia
 - Lab 1 due soon
- Material for the day
 - Cache coherence
 - Lock implementation
 - Blocking synchronization
- Acknowledgements
 - Thanks to Gadi Taubenfeld: I borrowed from some of his slides on barriers
 - Gadi's materials on synchronization are a great resource



Today

- Questions?
- Administrivia
 - Lab 1 due soon
- Material for the day
 - Cache coherence
 - Lock implementation
 - Blocking synchronization
- Acknowledgements
 - Thanks to Gadi Taubenfeld: I borrowed from some of his slides on barriers
 - Gadi's materials on synchronization are a great resource



Faux Quiz (answer any 2, 5 min)

- What is the difference between spinning/busy-wait and blocking synchronization?
- Can you write shared memory parallel applications using single-threaded processes only?
- How do you choose between spinlock/mutex on a multi-processor?
- Define the states of the MESI protocol. Is the E state necessary? Why or why not?
- What is bus locking?
- What is the difference between Mesa and Hoare monitors?
- Why recheck the condition on wakeup from a monitor wait?
- How can you build barriers with spinlocks?
- How can you build barriers with monitors?
- What is the difference between a mutex and a semaphore?

Review: correctness conditions

```
while(1) {  
    Entry section  
    Critical section  
    Exit section  
    Non-critical section  
}
```

Review: correctness conditions

- Safety
 - Only one thread in the critical region

```
while(1) {  
    Entry section  
    Critical section  
    Exit section  
    Non-critical section  
}
```

Review: correctness conditions

- Safety
 - Only one thread in the critical region
- Liveness
 - Some thread that enters the entry section eventually enters the critical region
 - Even if other thread takes forever in non-critical region

```
while(1) {  
    Entry section  
    Critical section  
    Exit section  
    Non-critical section  
}
```


Review: correctness conditions

- Safety
 - Only one thread in the critical region
- Liveness
 - Some thread that enters the entry section eventually enters the critical region
 - Even if other thread takes forever in non-critical region
- Bounded waiting
 - A thread that enters the entry section enters the critical section within some bounded number of operations.

```
while(1) {  
    Entry section  
    Critical section  
    Exit section  
    Non-critical section  
}
```

Review: correctness conditions

- Safety
 - Only one thread in the critical region
- Liveness
 - Some thread that enters the entry section eventually enters the critical region
 - Even if other thread takes forever in non-critical region
- Bounded waiting
 - ~~A thread that enters the entry section enters the critical section within some bounded number of operations.~~
 - *If a thread i is in entry section, then there is a bound on the number of times that other threads are allowed to enter the critical section before thread i 's request is granted*

```
while(1) {  
    Entry section  
    Critical section  
    Exit section  
    Non-critical section  
}
```

Review: correctness conditions

- Safety
 - Only one thread in the critical region
- Liveness
 - Some thread that enters the entry section eventually enters the critical region
 - Even if other thread takes forever in non-critical region
- Bounded waiting
 - ~~A thread that enters the entry section enters the critical section within some bounded number of operations.~~
 - *If a thread i is in entry section, then there is a bound on the number of times that other threads are allowed to enter the critical section before thread i 's request is granted*

Theorem: Every property is a combination of a safety property and a liveness property.

-Bowen Alpern & Fred Schneider

<https://www.cs.cornell.edu/fbs/publications/defliveness.pdf>

```
while(1) {  
    Entry section  
    Critical section  
    Exit section  
    Non-critical section  
}
```

Review: correctness conditions

- Safety
 - Only one thread in the critical region
- Liveness
 - Some thread that enters the entry section eventually enters the critical region
 - Even if other thread takes forever in non-critical region
- Bounded waiting
 - ~~A thread that enters the entry section enters the critical section within some bounded number of operations.~~
 - *If a thread i is in entry section, then there is a bound on the number of times that other threads are allowed to enter the critical section before thread i 's request is granted*

Theorem: Every property is a combination of a safety property and a liveness property.

-Bowen Alpern & Fred Schneider

<https://www.cs.cornell.edu/fbs/publications/defliveness.pdf>

Mutex, spinlock, etc.
are ways to implement
these

```
while (1) {  
    Entry section  
    Critical section  
    Exit section  
    Non-critical section  
}
```

Review: correctness conditions

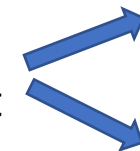
- Safety
 - Only one thread in the critical region
- Liveness
 - Some thread that enters the entry section eventually enters the critical region
 - Even if other thread takes forever in non-critical region
- Bounded waiting
 - ~~A thread that enters the entry section enters the critical section within some bounded number of operations.~~
 - *If a thread i is in entry section, then there is a bound on the number of times that other threads are allowed to enter the critical section before thread i 's request is granted*

Theorem: Every property is a combination of a safety property and a liveness property.

-Bowen Alpern & Fred Schneider

<https://www.cs.cornell.edu/fbs/publications/defliveness.pdf>

Mutex, spinlock, etc.
are ways to implement



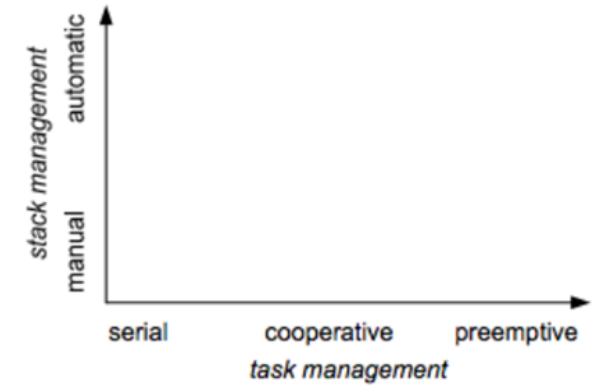
```
while (1) {  
    Entry section  
    Critical section  
    Exit section  
    Non-critical section  
}
```

Did we get all the important conditions?
Why is correctness defined in terms of locks?

Review: Execution Context Management

“Task” == “Flow of Control”, but with less typing

“Stack” == Task State

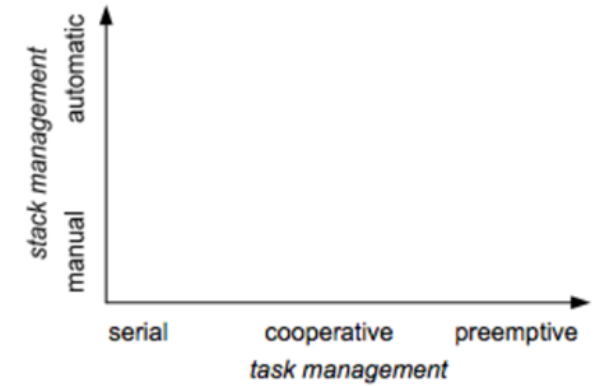


Review: Execution Context Management

“Task” == “Flow of Control”, but with less typing

“Stack” == Task State

Task Management



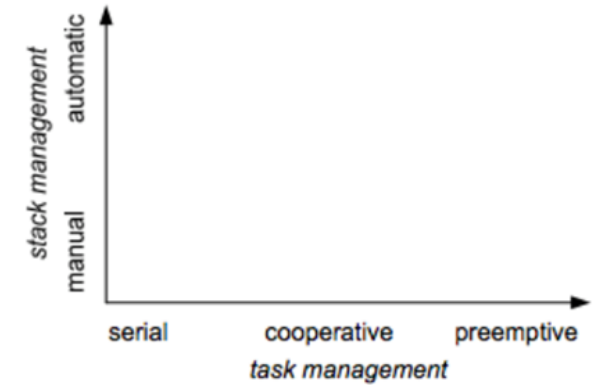
Review: Execution Context Management

“Task” == “Flow of Control”, but with less typing

“Stack” == Task State

Task Management

- Preemptive
 - Interleave on uniprocessor
 - Overlap on multiprocessor



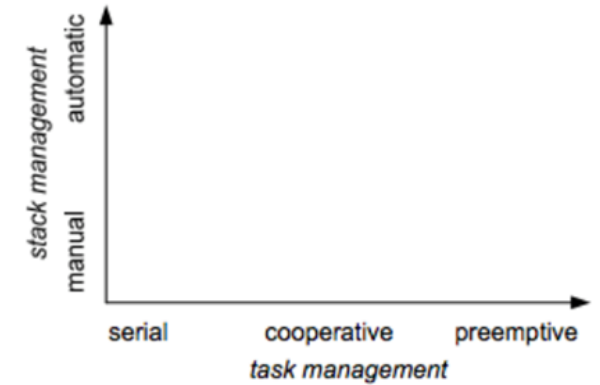
Review: Execution Context Management

“Task” == “Flow of Control”, but with less typing

“Stack” == Task State

Task Management

- Preemptive
 - Interleave on uniprocessor
 - Overlap on multiprocessor
- Serial
 - One at a time, no conflict



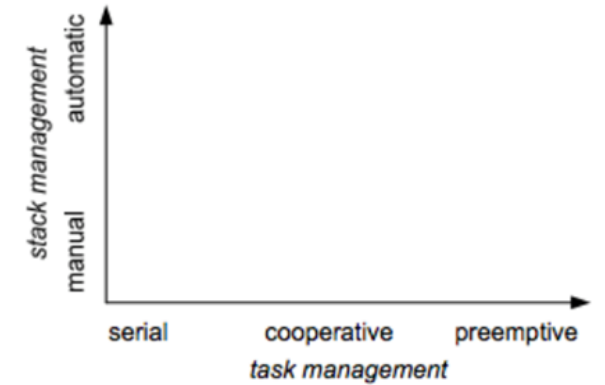
Review: Execution Context Management

“Task” == “Flow of Control”, but with less typing

“Stack” == Task State

Task Management

- Preemptive
 - Interleave on uniprocessor
 - Overlap on multiprocessor
- Serial
 - One at a time, no conflict
- Cooperative
 - Yields at well-defined points
 - E.g. wait for long-running I/O



Review: Execution Context Management

“Task” == “Flow of Control”, but with less typing

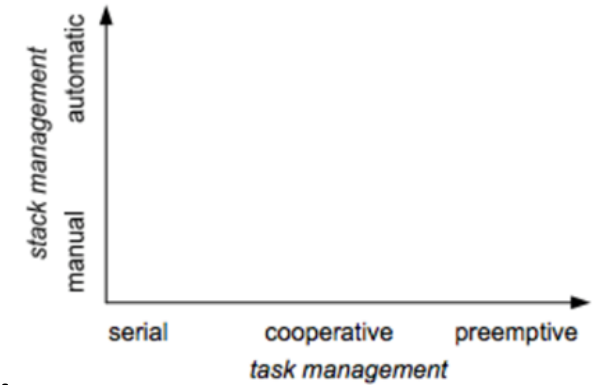
“Stack” == Task State

Task Management

- Preemptive
 - Interleave on uniprocessor
 - Overlap on multiprocessor
- Serial
 - One at a time, no conflict
- Cooperative
 - Yields at well-defined points
 - E.g. wait for long-running I/O

Stack Management

- Manual
 - Inherent in Cooperative
 - Changing at quiescent points
- Automatic
 - Inherent in pre-emptive
 - Downside: Hidden concurrency assumptions



Review: Execution Context Management

“Task” == “Flow of Control”, but with less typing

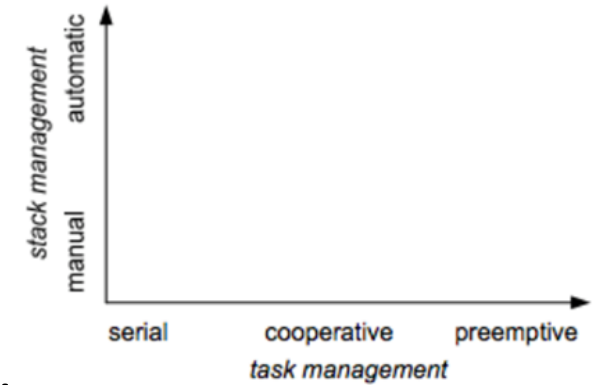
“Stack” == Task State

Task Management

- Preemptive
 - Interleave on uniprocessor
 - Overlap on multiprocessor
- Serial
 - One at a time, no conflict
- Cooperative
 - Yields at well-defined points
 - E.g. wait for long-running I/O

Stack Management

- Manual
 - Inherent in Cooperative
 - Changing at quiescent points
- Automatic
 - Inherent in pre-emptive
 - Downside: Hidden concurrency assumptions



These dimensions can be orthogonal

Review: Fibers: the Sweet Spot?

Review: Fibers: the Sweet Spot?

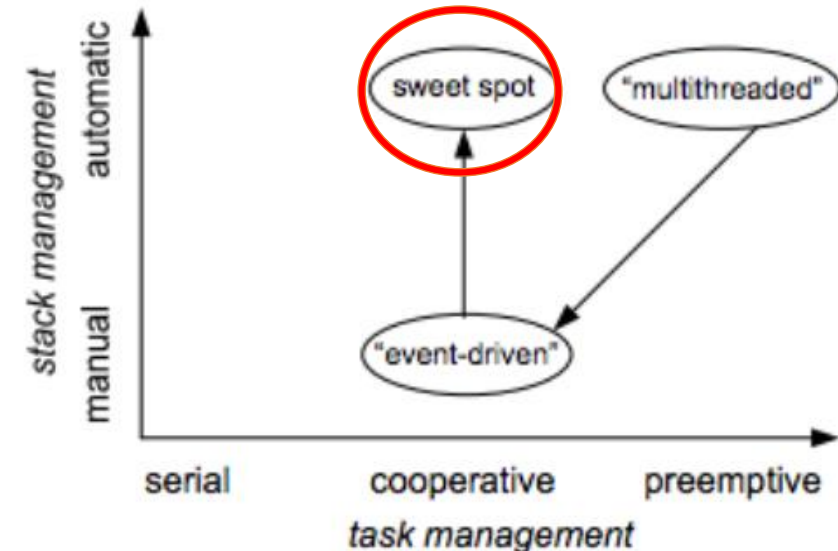
- Cooperative tasks
 - most desirable when reasoning about concurrency
 - usually associated with event-driven programming

Review: Fibers: the Sweet Spot?

- Cooperative tasks
 - most desirable when reasoning about concurrency
 - usually associated with event-driven programming
- Automatic stack management
 - most desirable when reading/maintaining code
 - Usually associated with threaded (or serial) programming

Review: Fibers: the Sweet Spot?

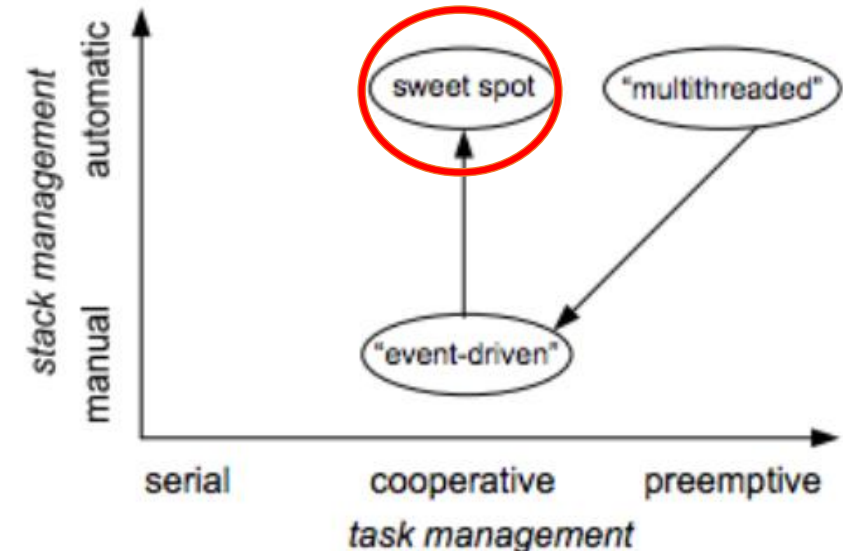
- Cooperative tasks
 - most desirable when reasoning about concurrency
 - usually associated with event-driven programming
- Automatic stack management
 - most desirable when reading/maintaining code
 - Usually associated with threaded (or serial) programming



Review: Fibers: the Sweet Spot?

- Cooperative tasks
 - most desirable when reasoning about concurrency
 - usually associated with event-driven programming
- Automatic stack management
 - most desirable when reading/maintaining code
 - Usually associated with threaded (or serial) programming

Fibers: cooperative threading
with automatic stack
management



Multiprocessor Cache Coherence

Multiprocessor Cache Coherence

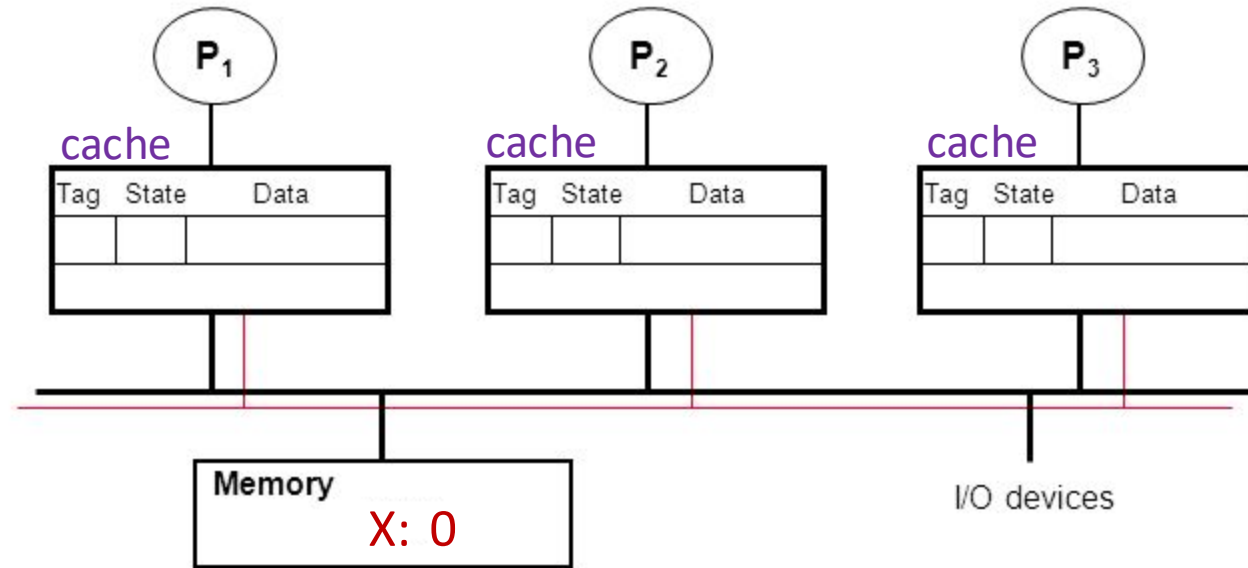
$$F = ma$$

Multiprocessor Cache Coherence

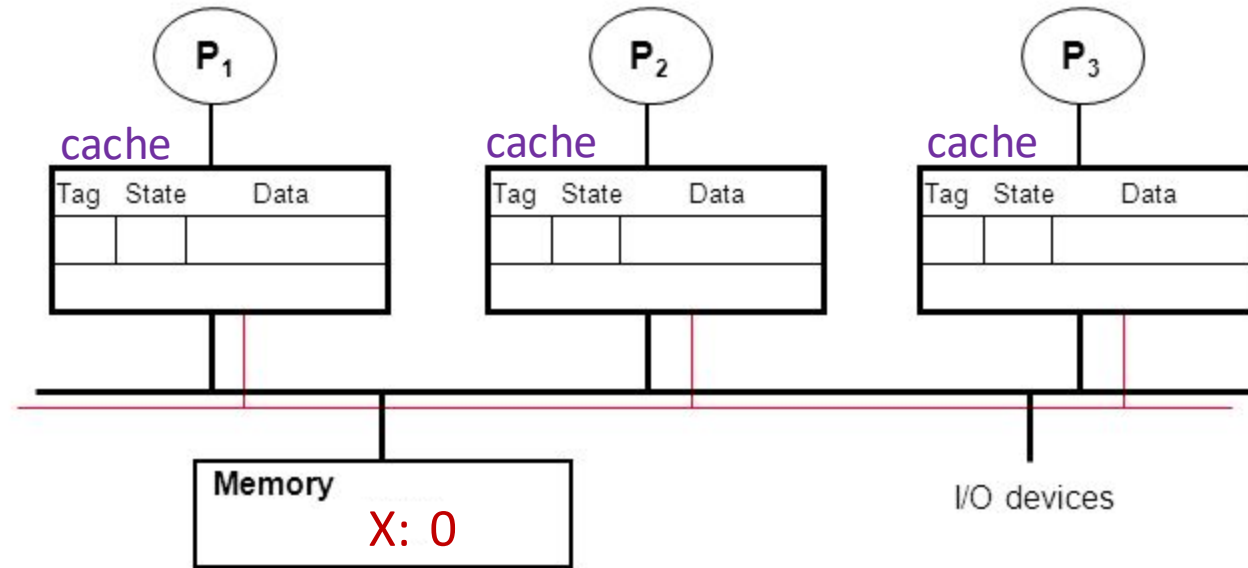
Physics | Concurrency

F = ma ~ coherence

Multiprocessor Cache Coherence

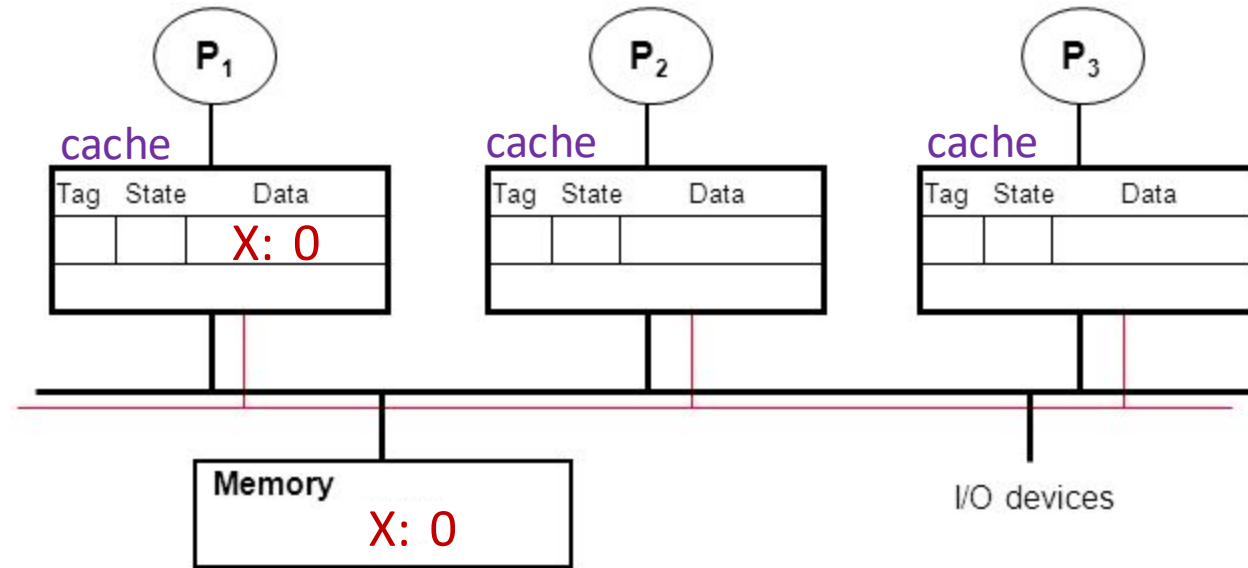


Multiprocessor Cache Coherence



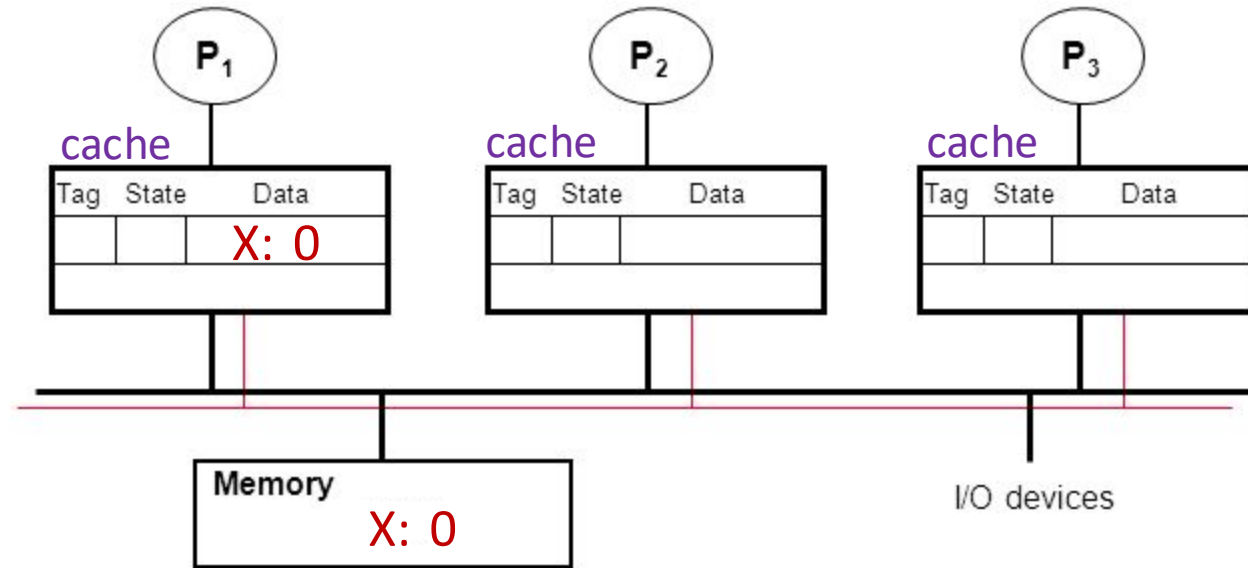
- P_1 : read X

Multiprocessor Cache Coherence



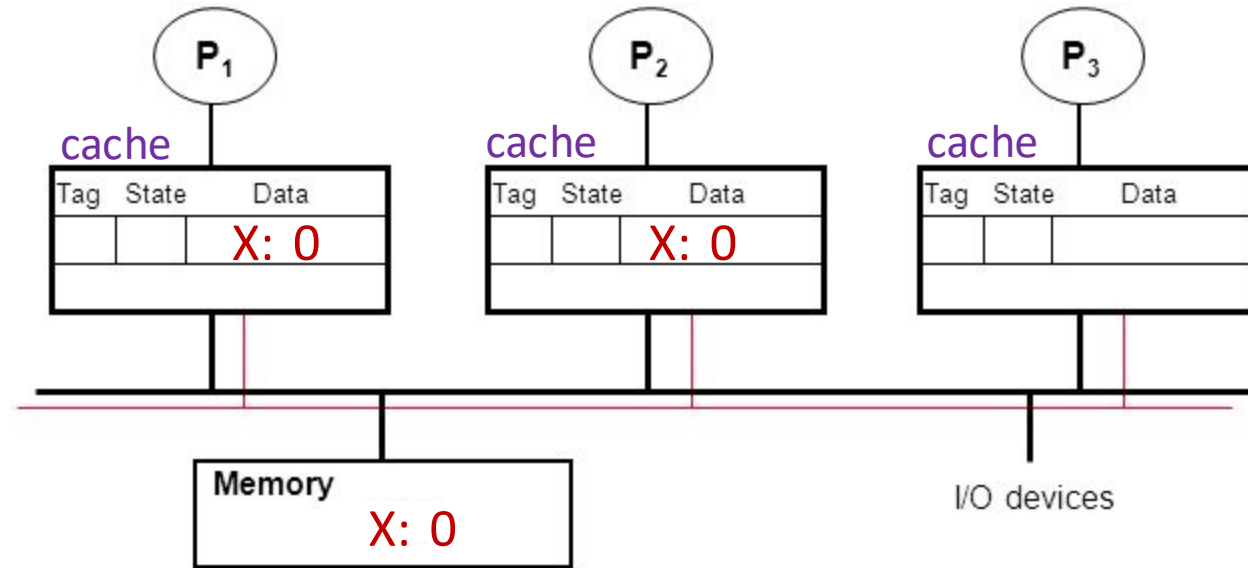
- P₁: read X

Multiprocessor Cache Coherence



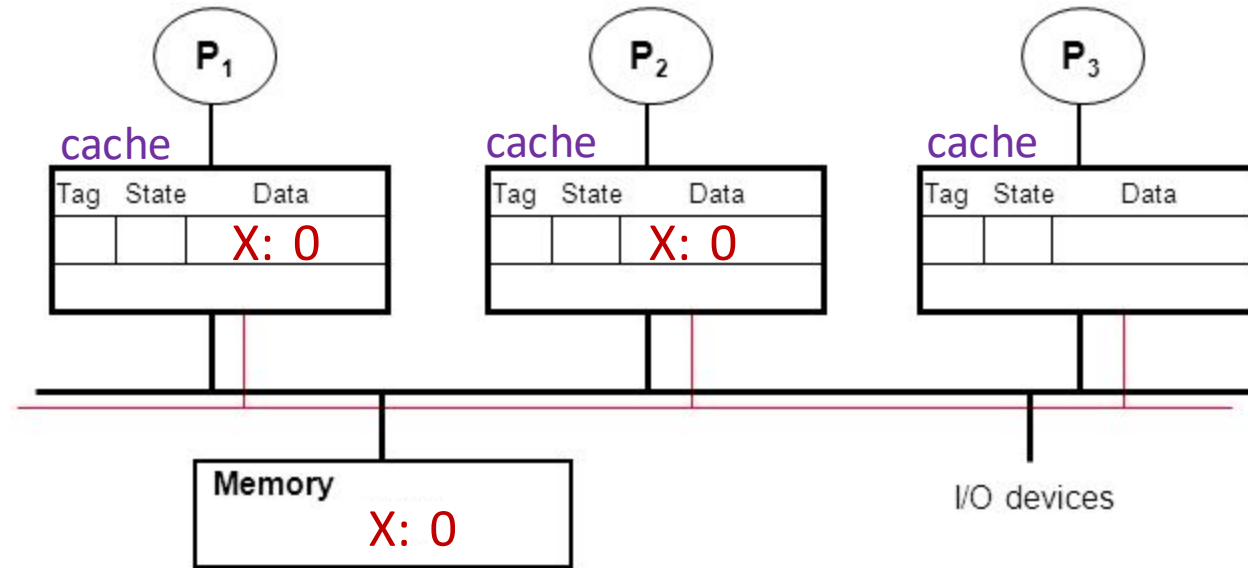
- P₁: read X
- P₂: read X

Multiprocessor Cache Coherence



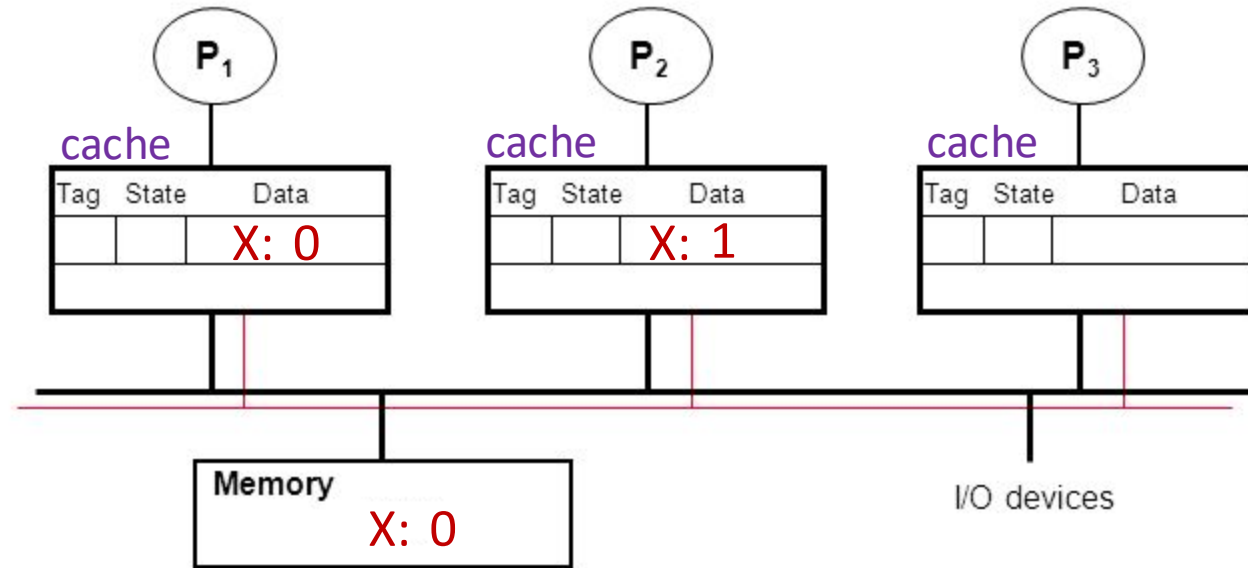
- P₁: read X
- P₂: read X

Multiprocessor Cache Coherence



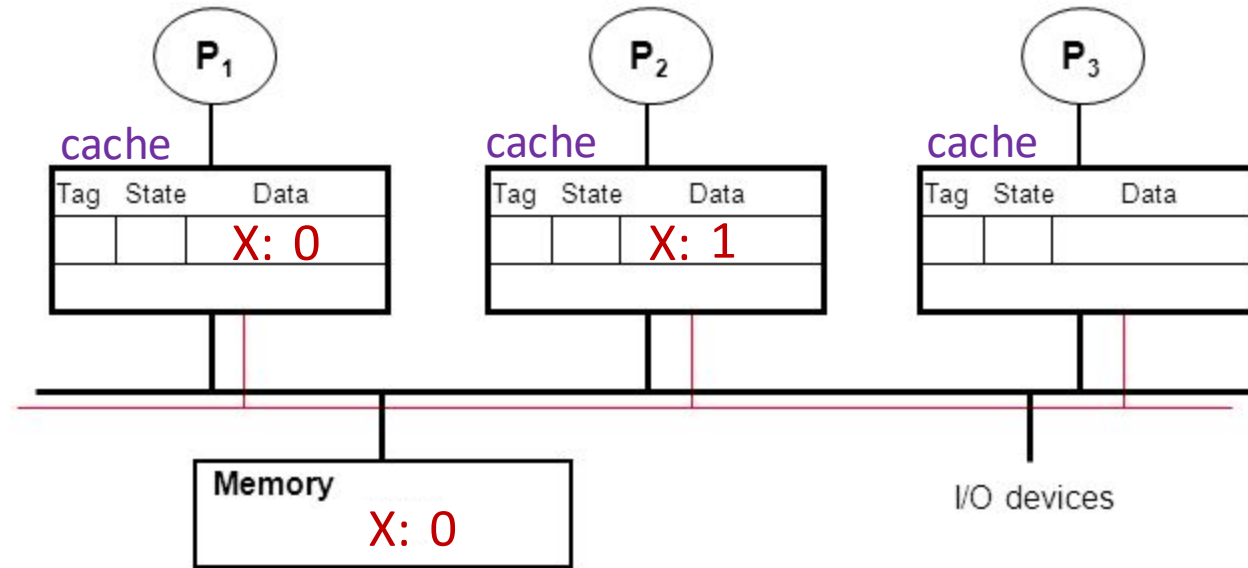
- P₁: read X
- P₂: read X
- P₂: X++

Multiprocessor Cache Coherence



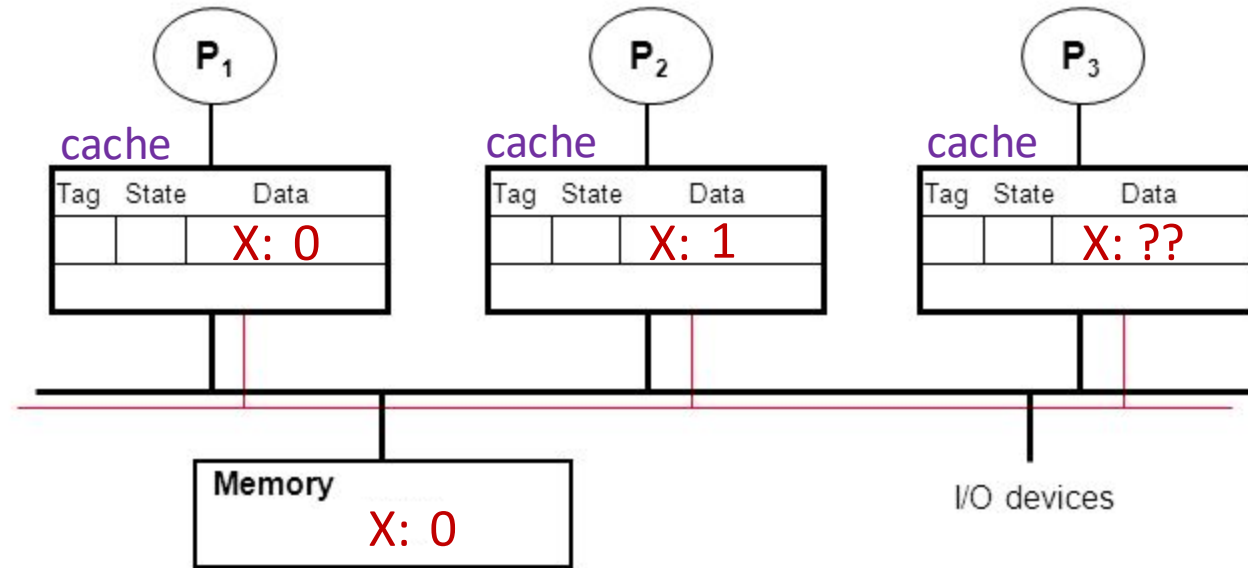
- P₁: read X
- P₂: read X
- P₂: X++

Multiprocessor Cache Coherence



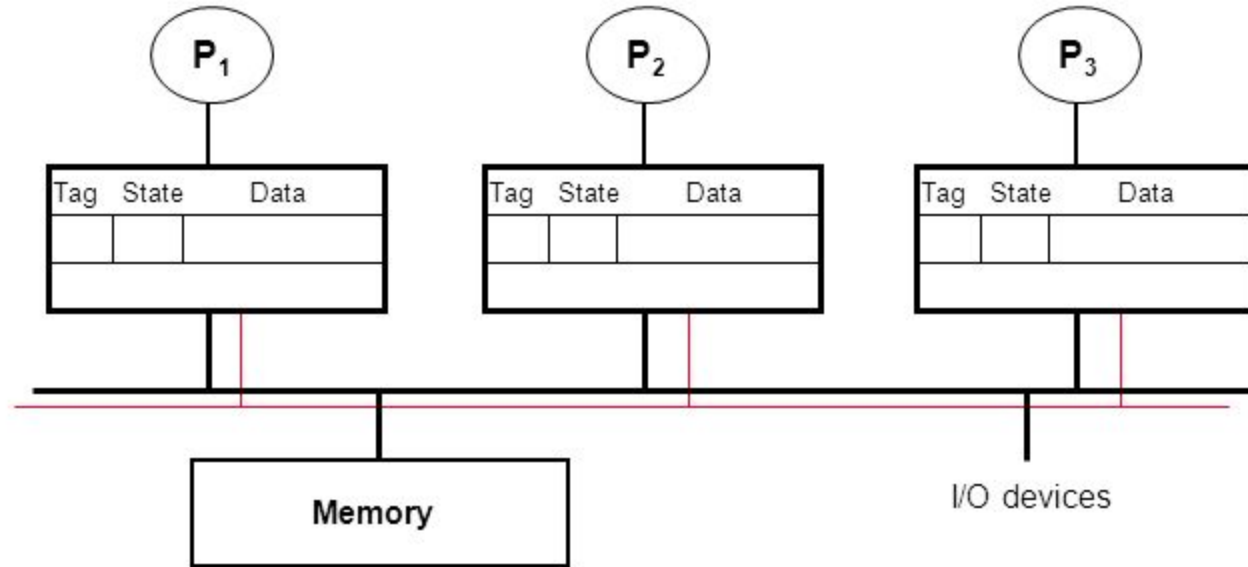
- P₁: read X
- P₂: read X
- P₂: X++
- P₃: read X

Multiprocessor Cache Coherence

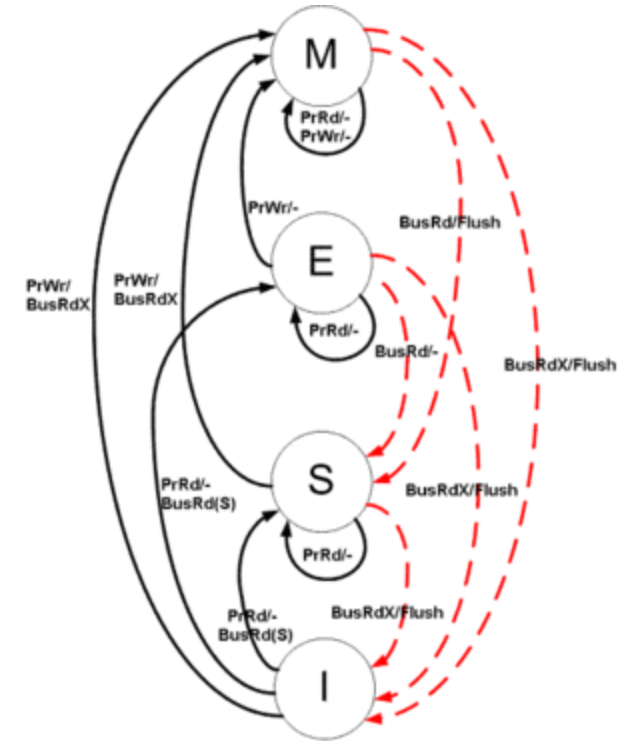
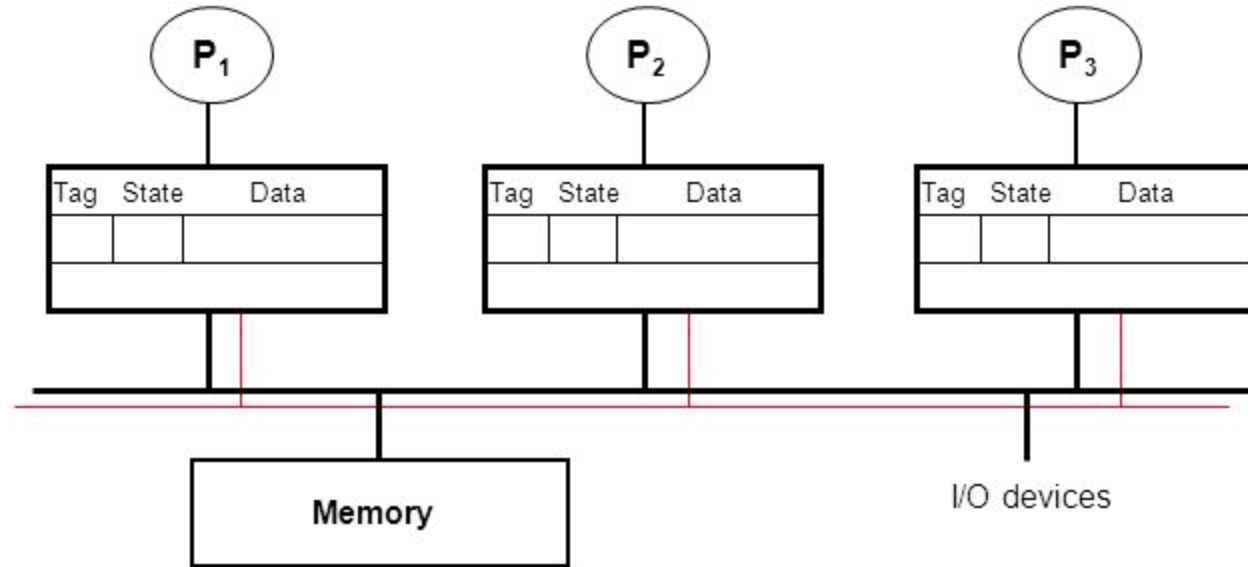


- P₁: read X
- P₂: read X
- P₂: X++
- P₃: read X

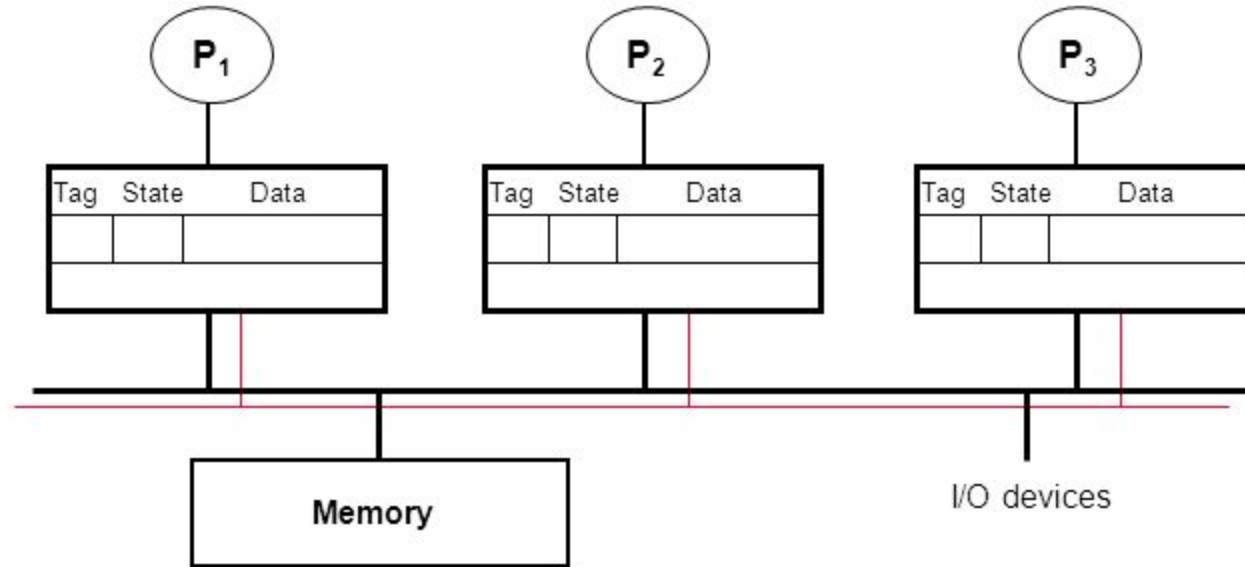
Multiprocessor Cache Coherence



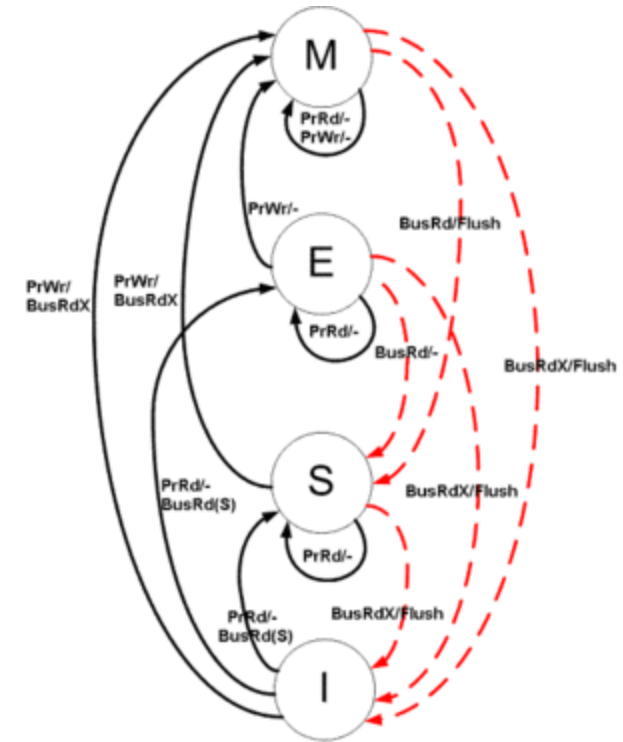
Multiprocessor Cache Coherence



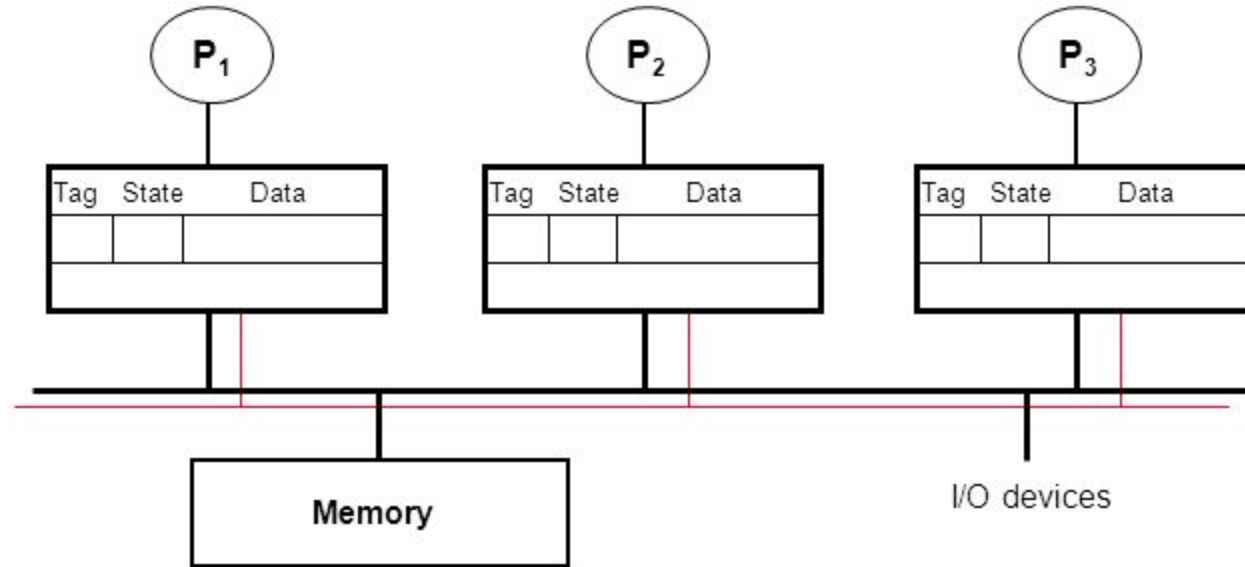
Multiprocessor Cache Coherence



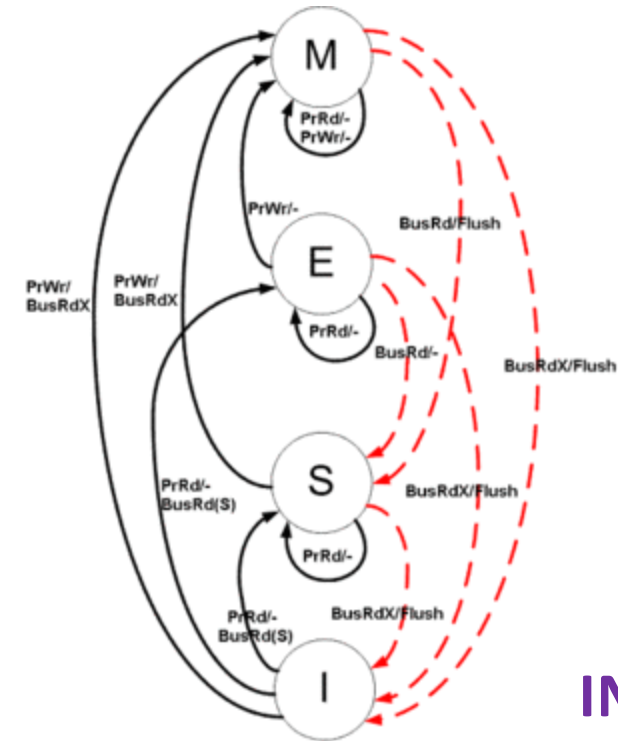
Each cache line has a state (M, E, S, I)



Multiprocessor Cache Coherence

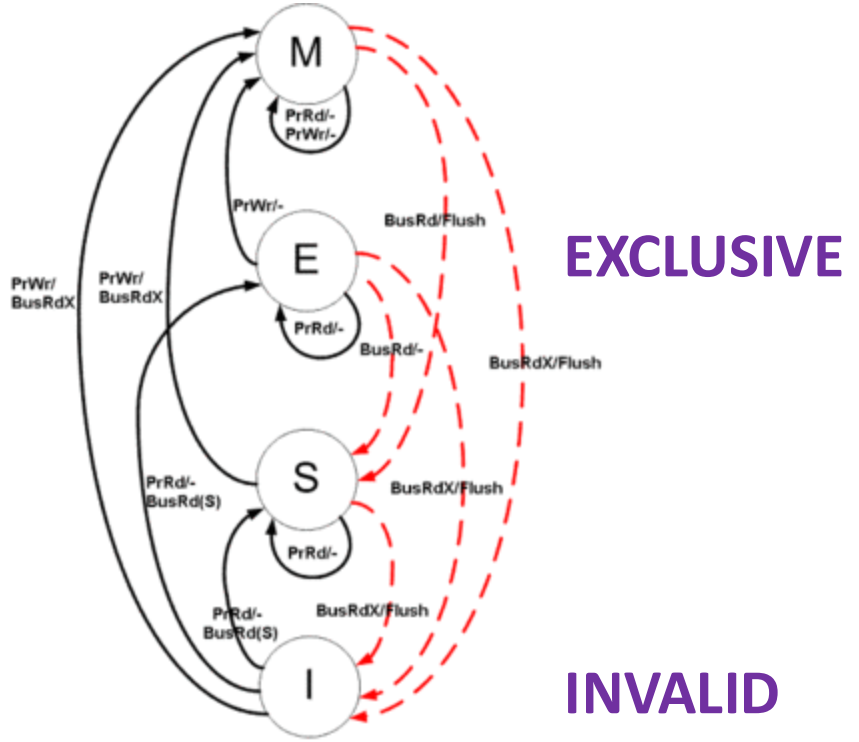
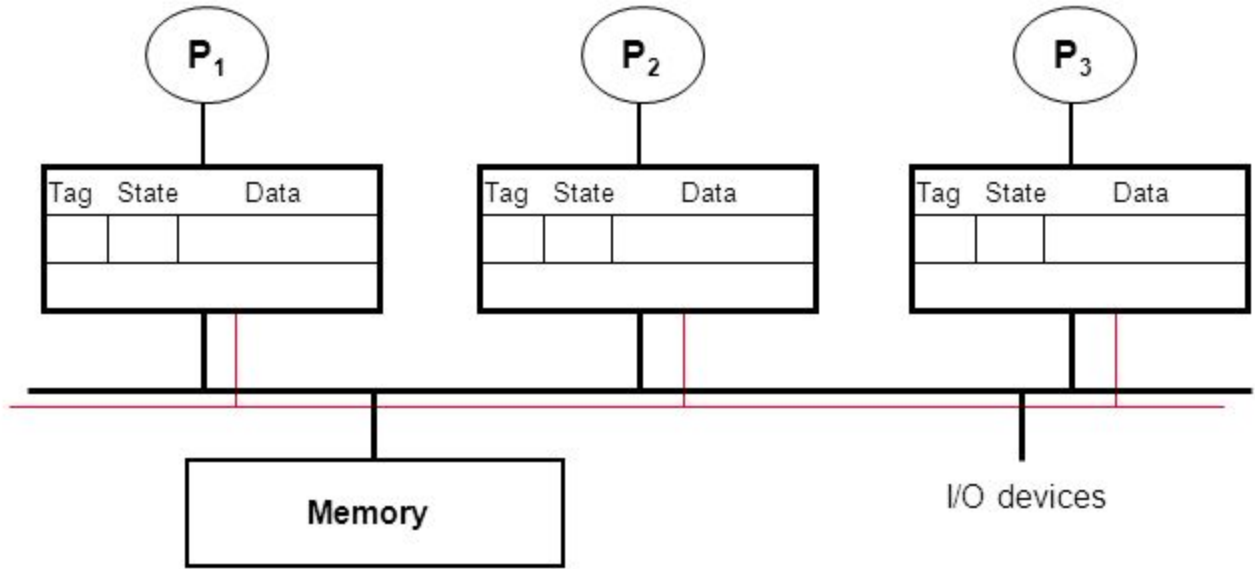


- Each cache line has a state (M, E, S, I)
- Processors “snoop” bus to maintain states
 - Initially → ‘I’ → Invalid



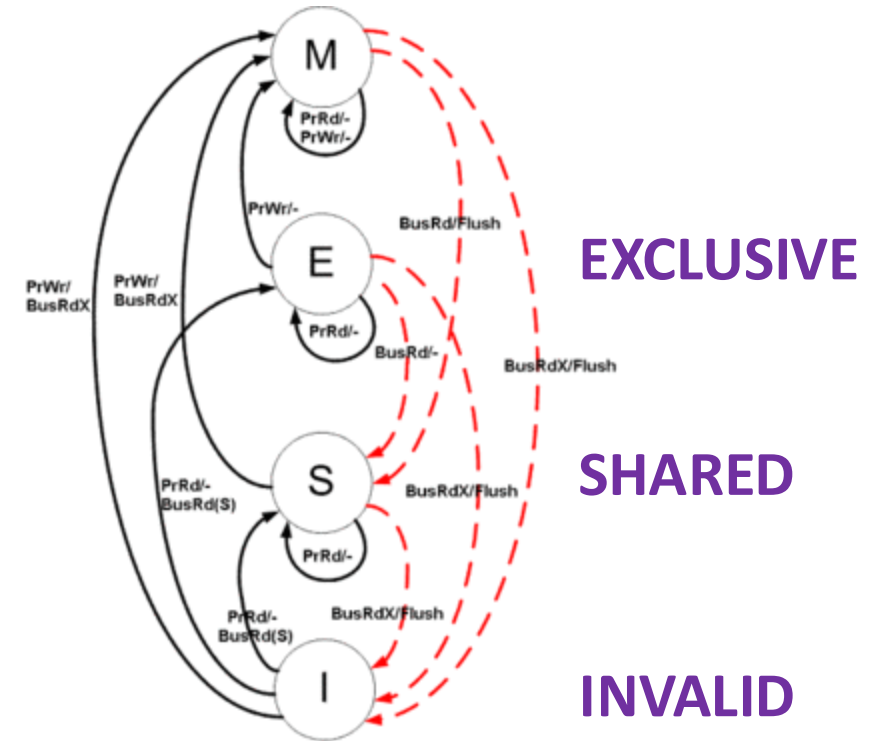
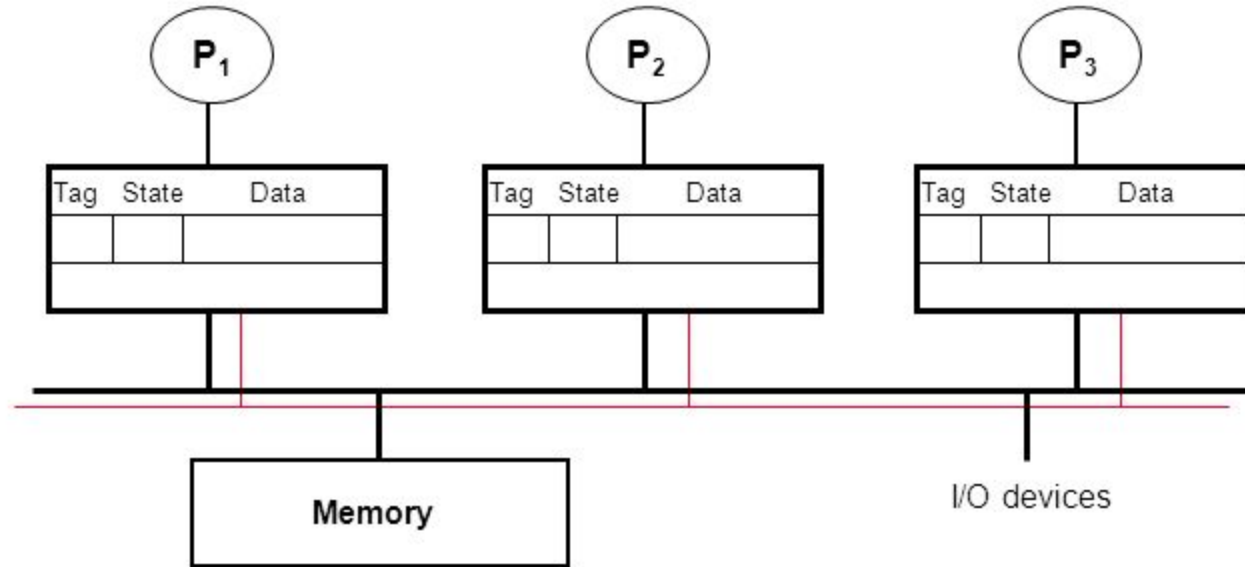
INVALID

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

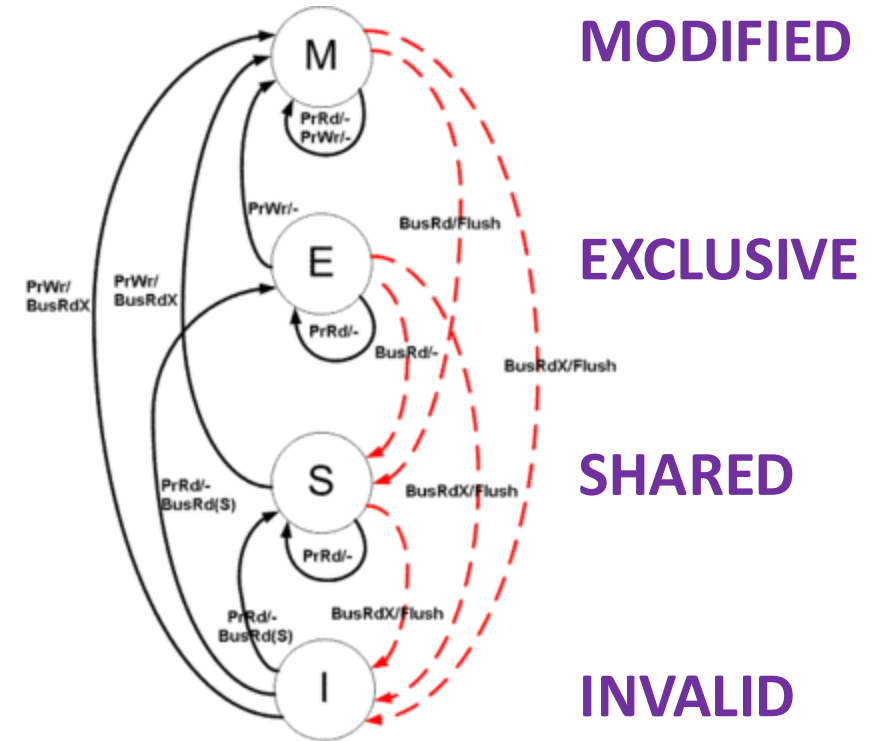
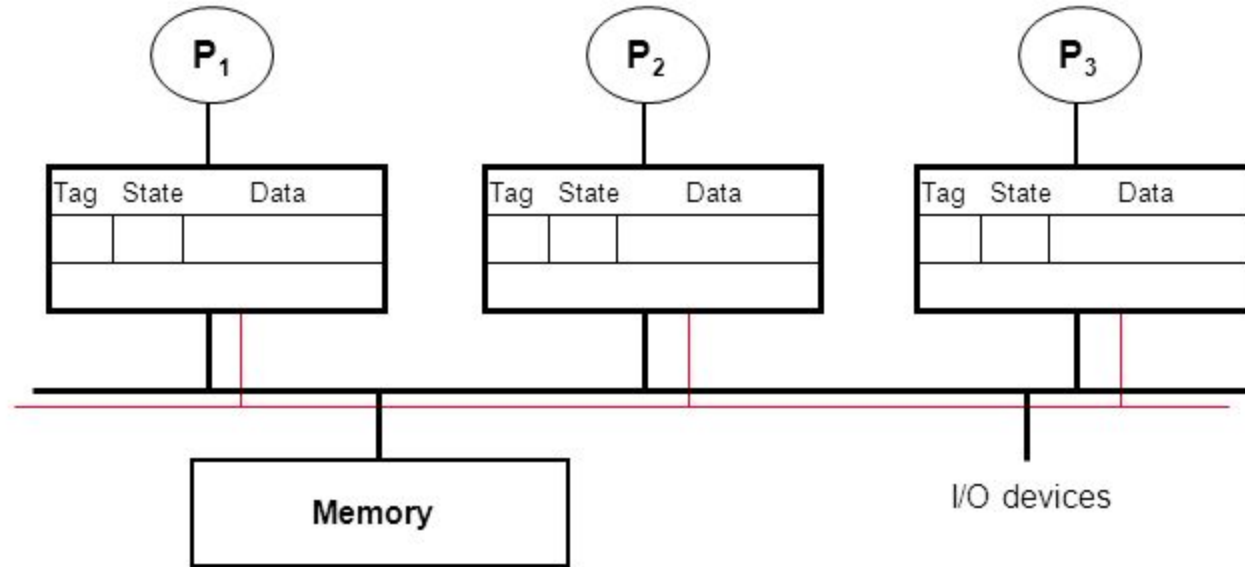
Multiprocessor Cache Coherence



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

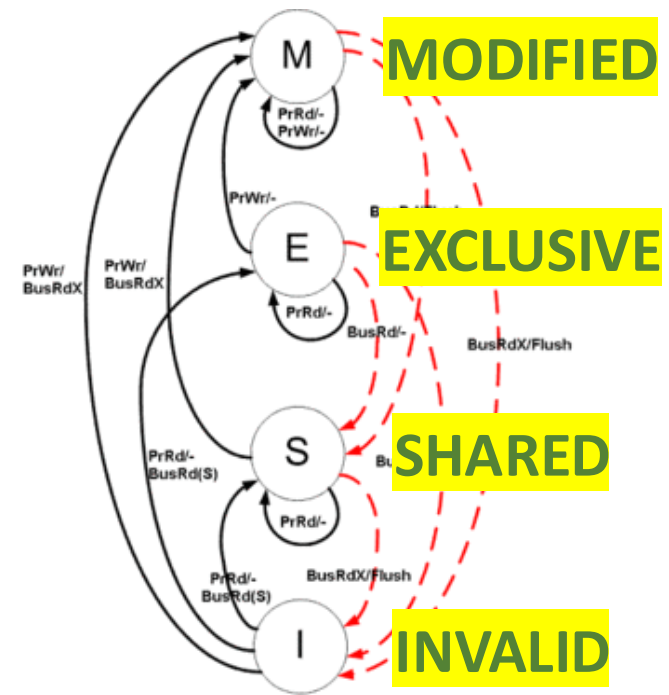
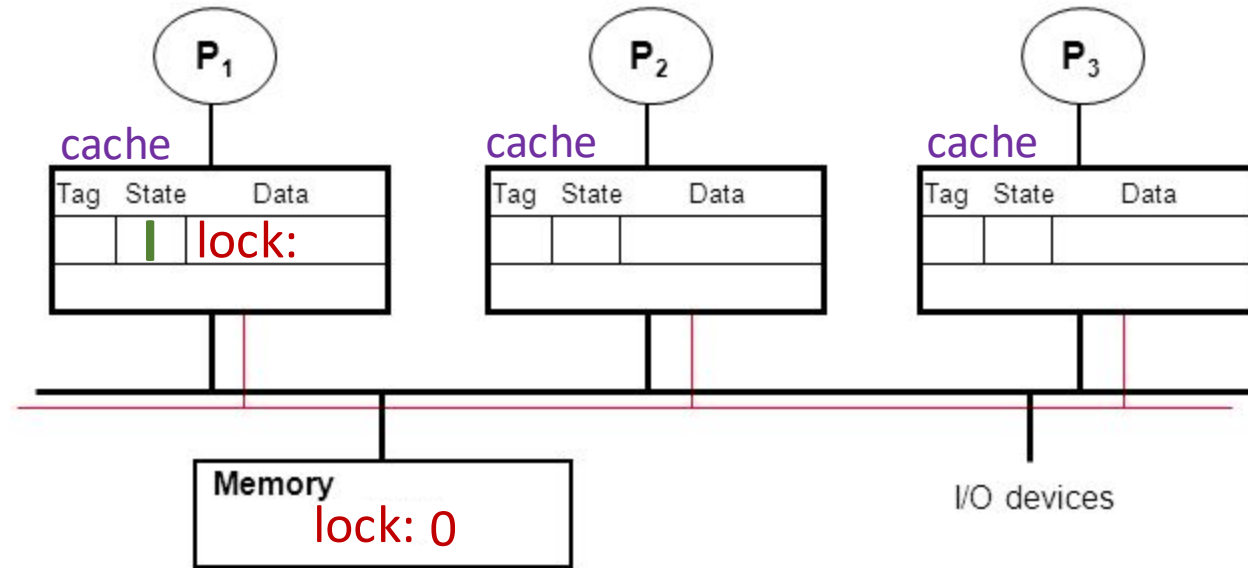
Multiprocessor Cache Coherence



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

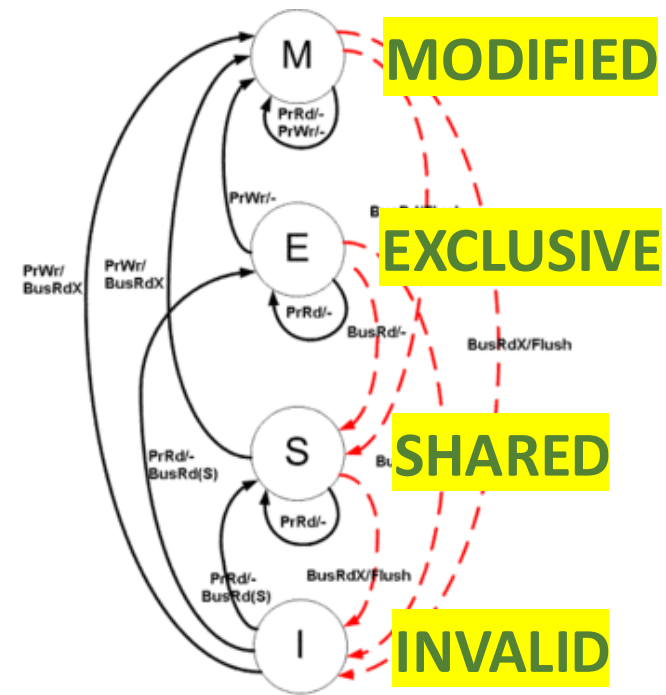
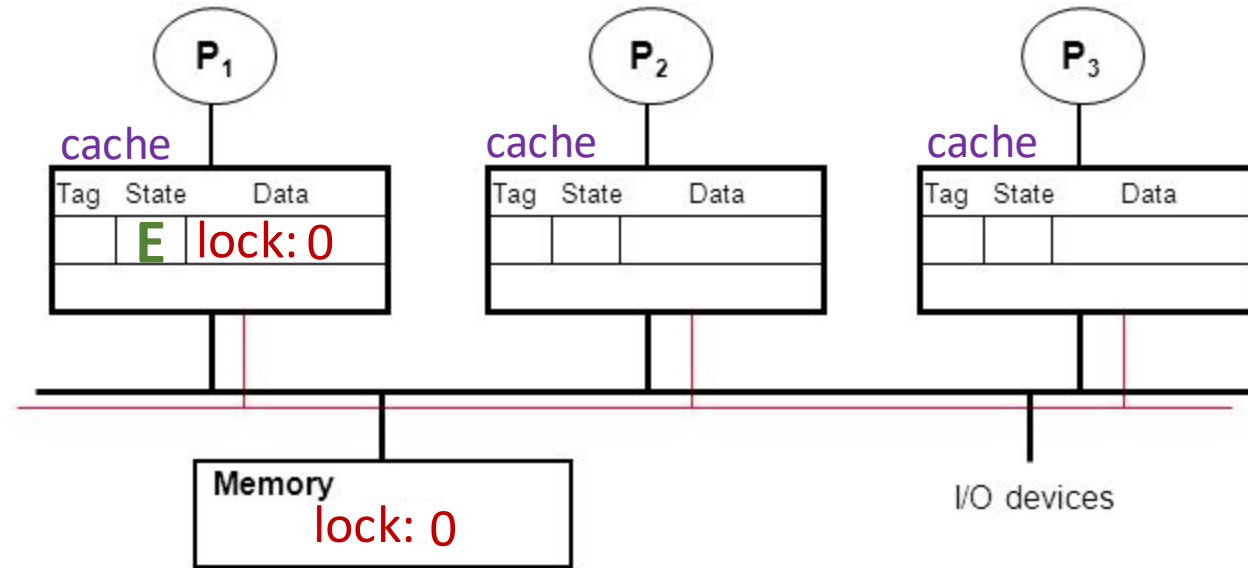


P1

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



Cache Coherence: single-thread

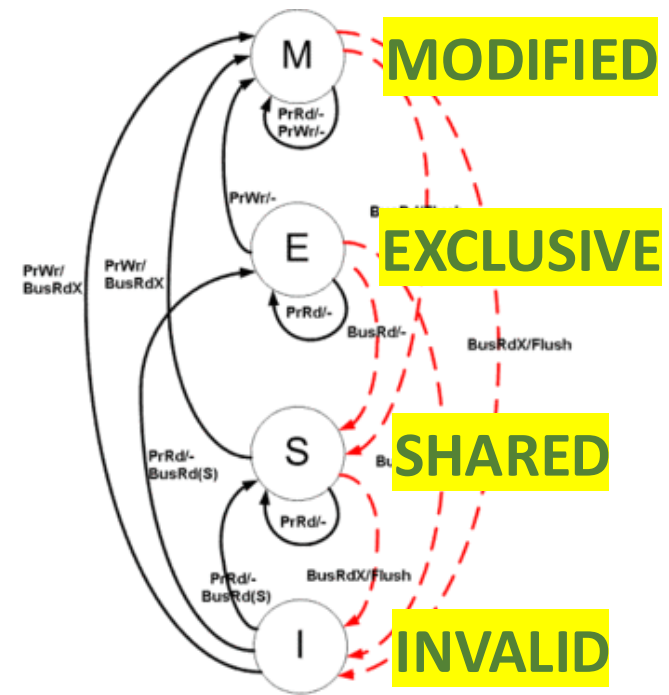
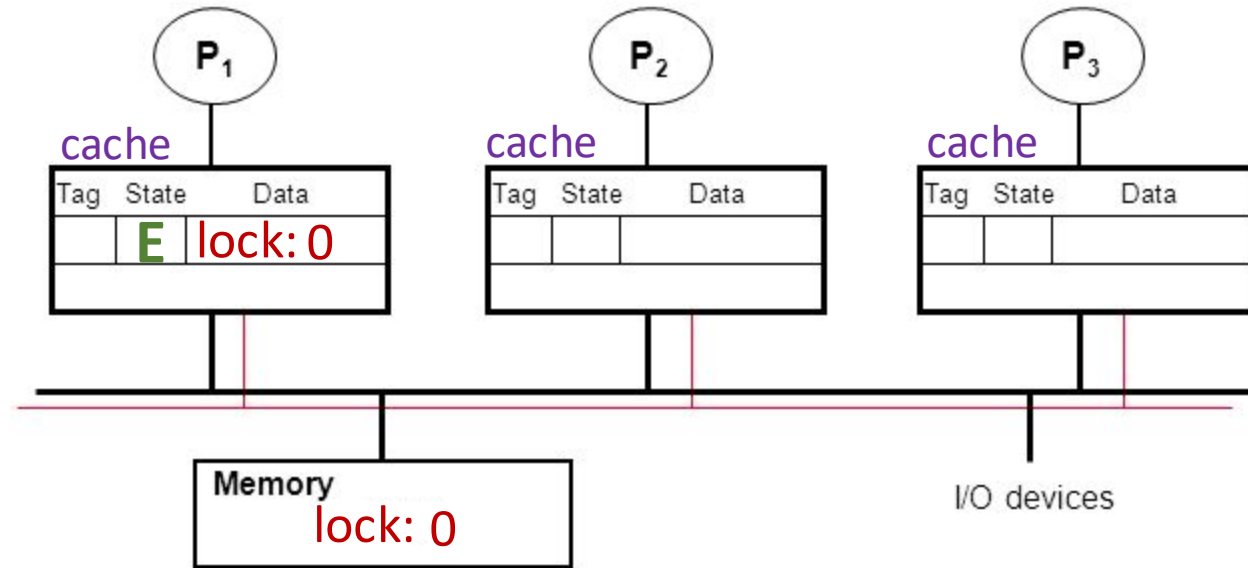


P1

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



Cache Coherence: single-thread

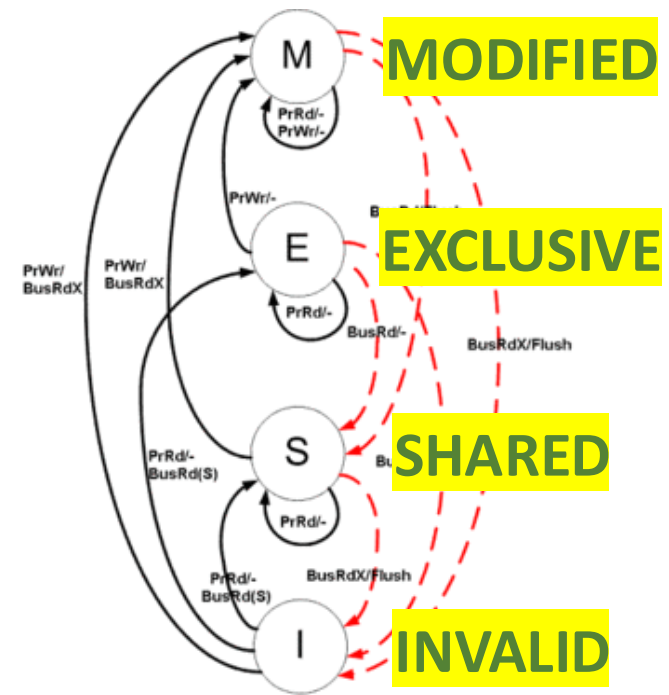
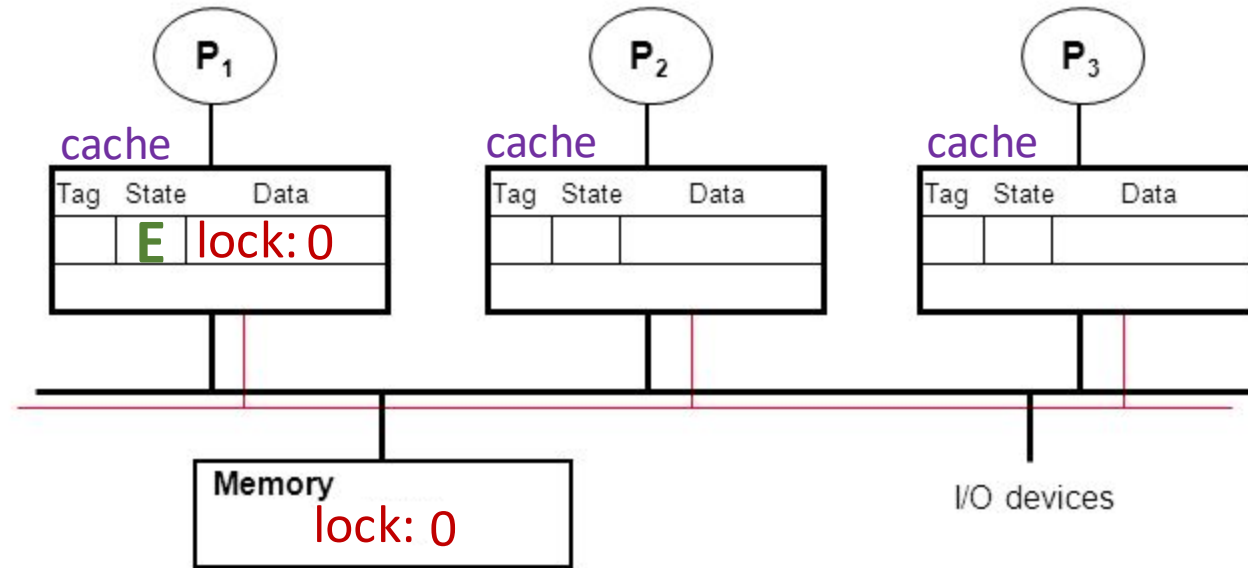


P1

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



Cache Coherence: single-thread

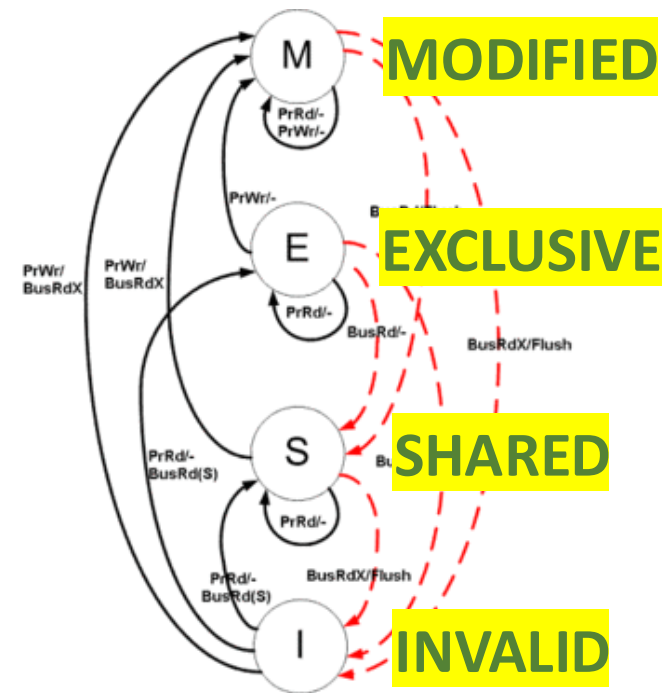
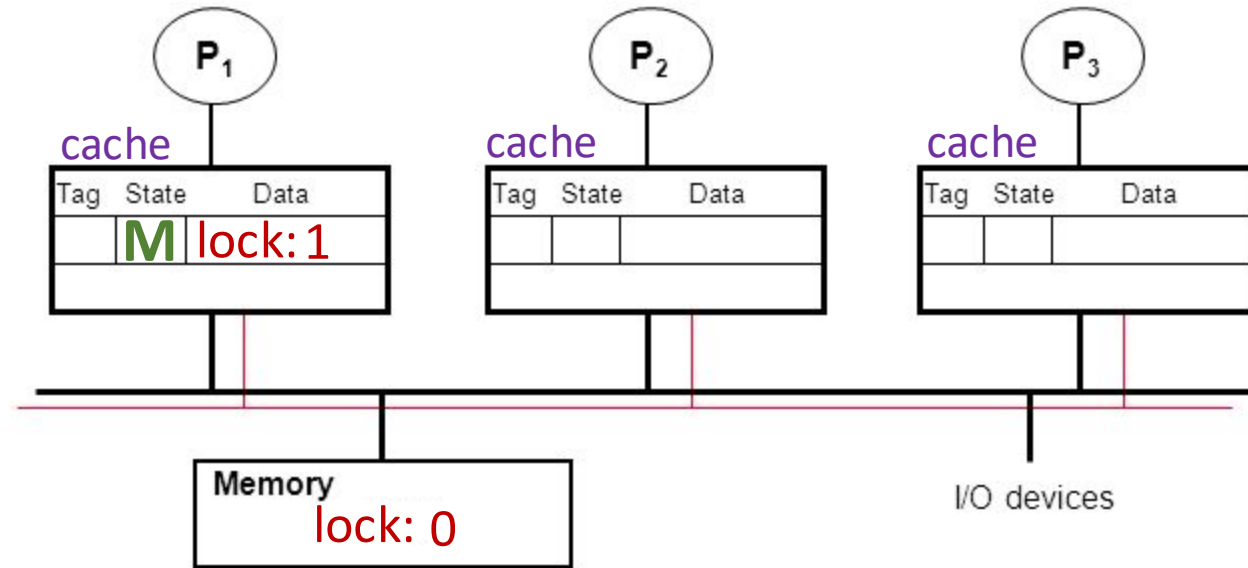


P1

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



Cache Coherence: single-thread

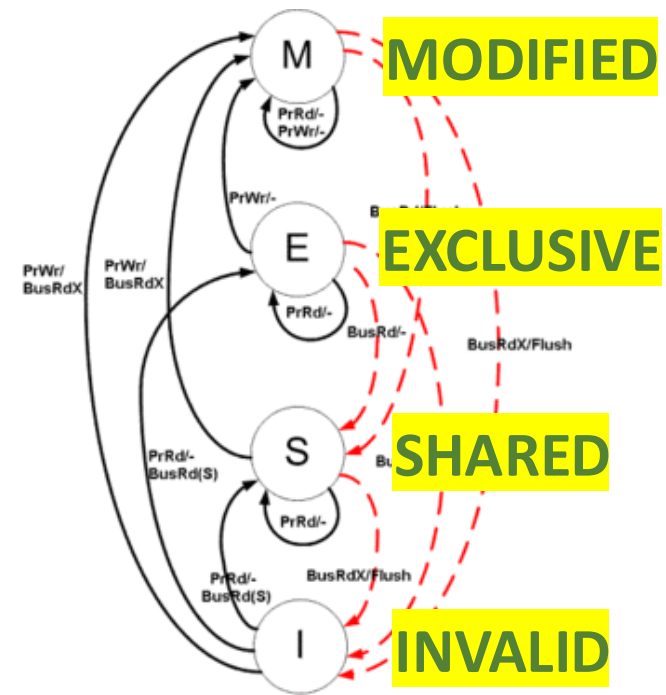
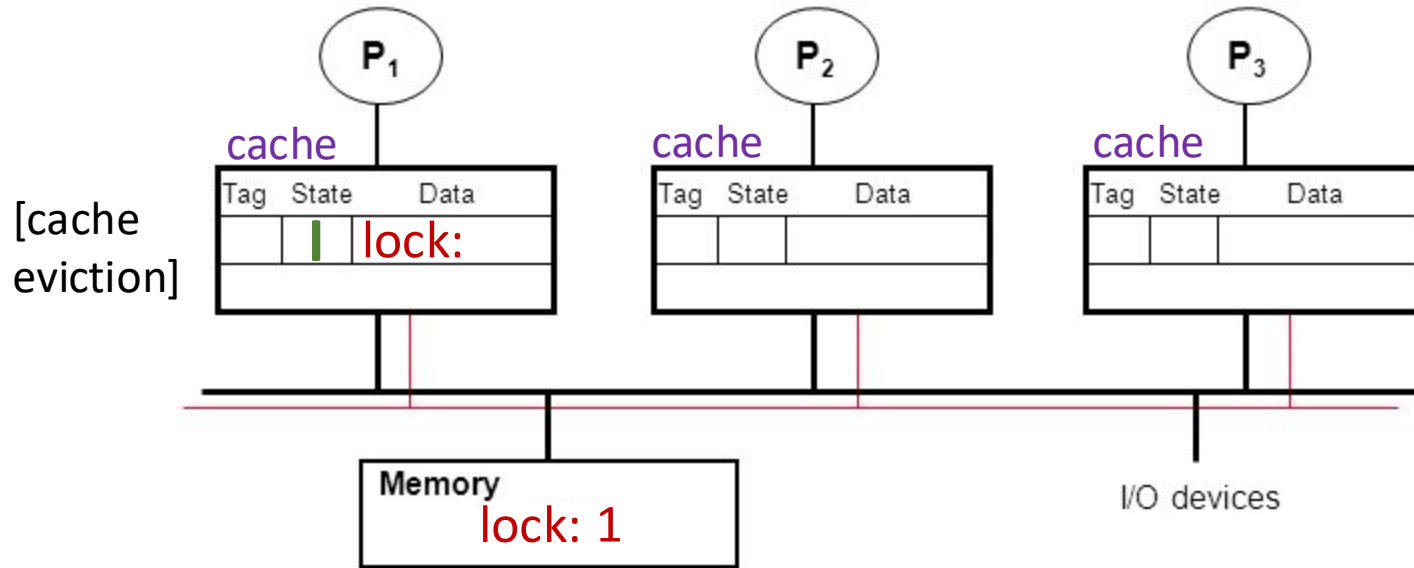


P1

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



Cache Coherence: single-thread

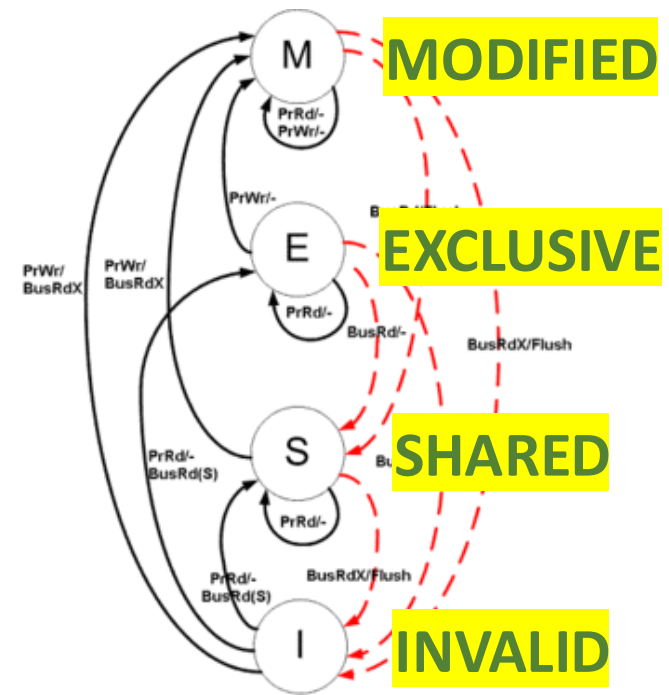
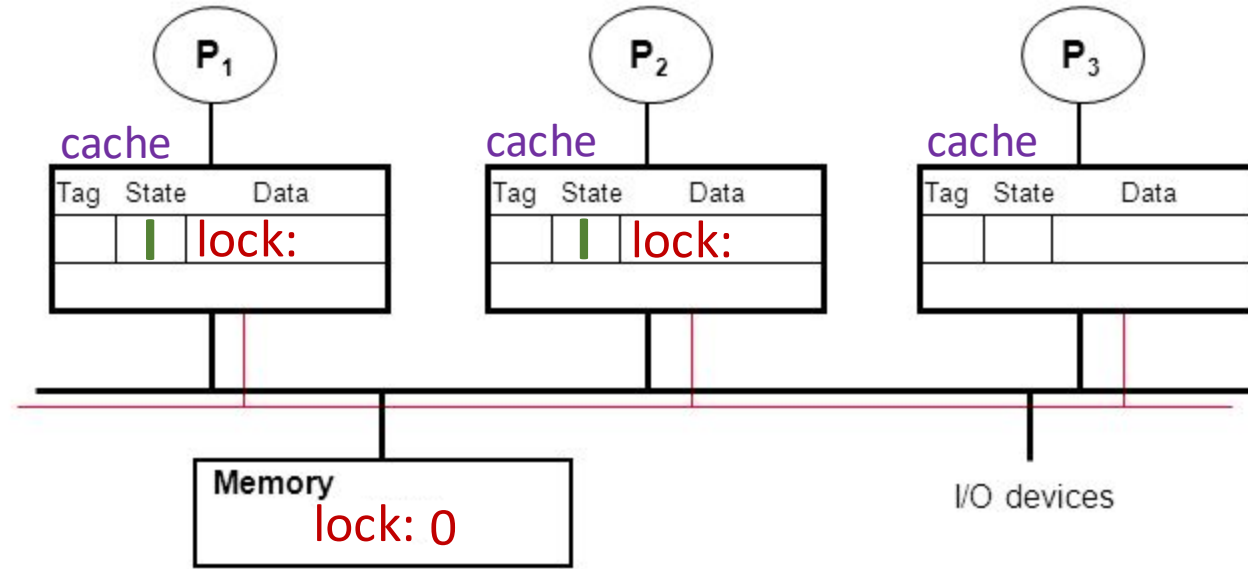


P1

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



Cache Coherence Action Zone



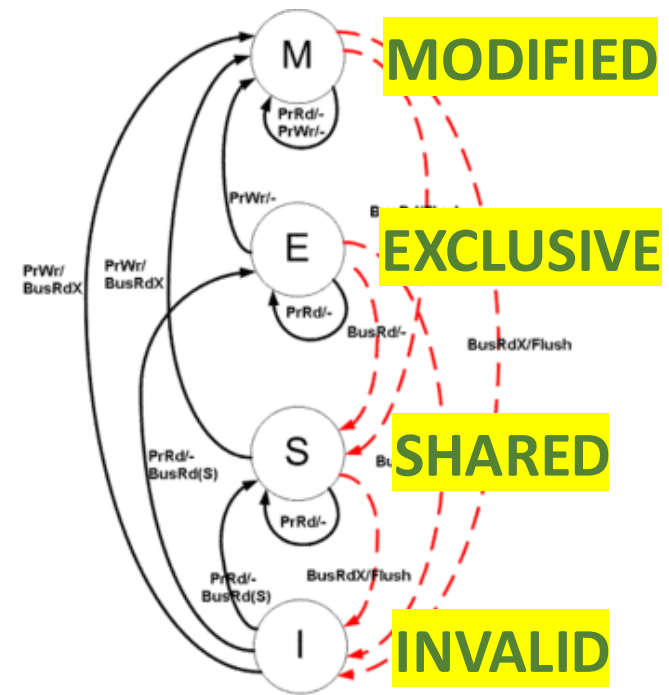
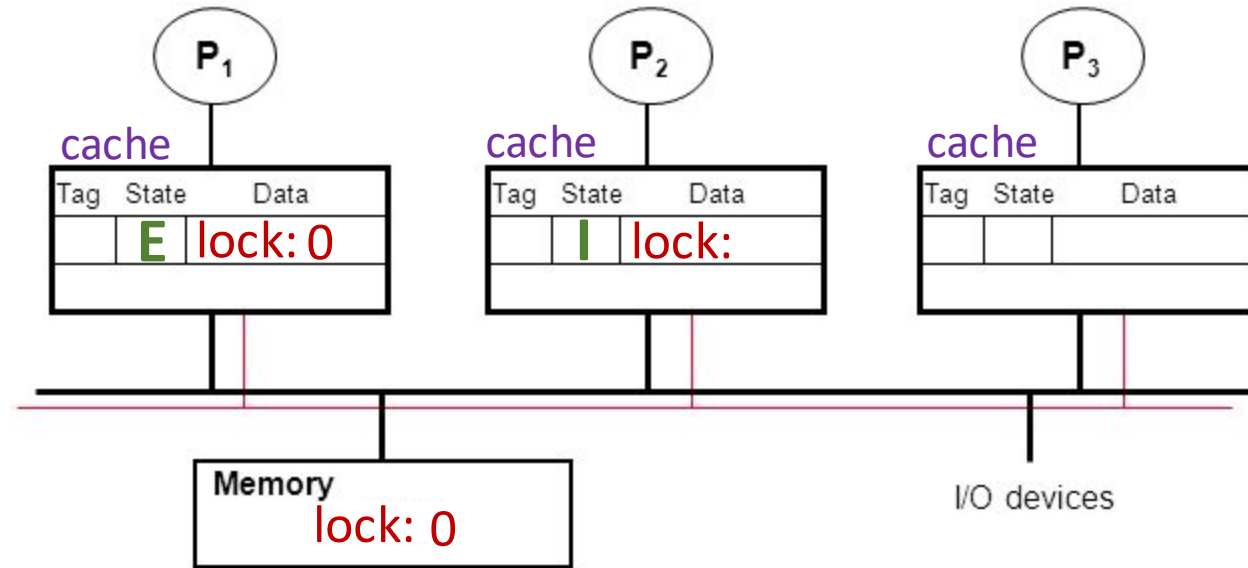
P1

```
// (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

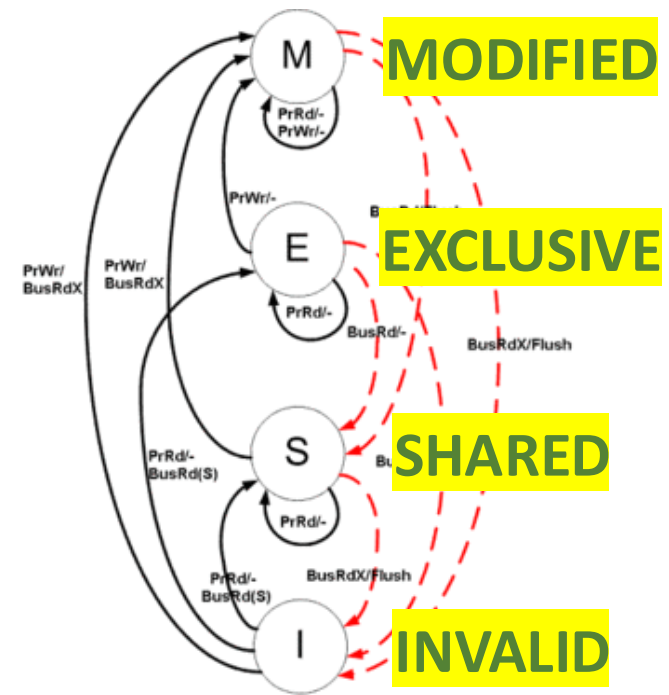
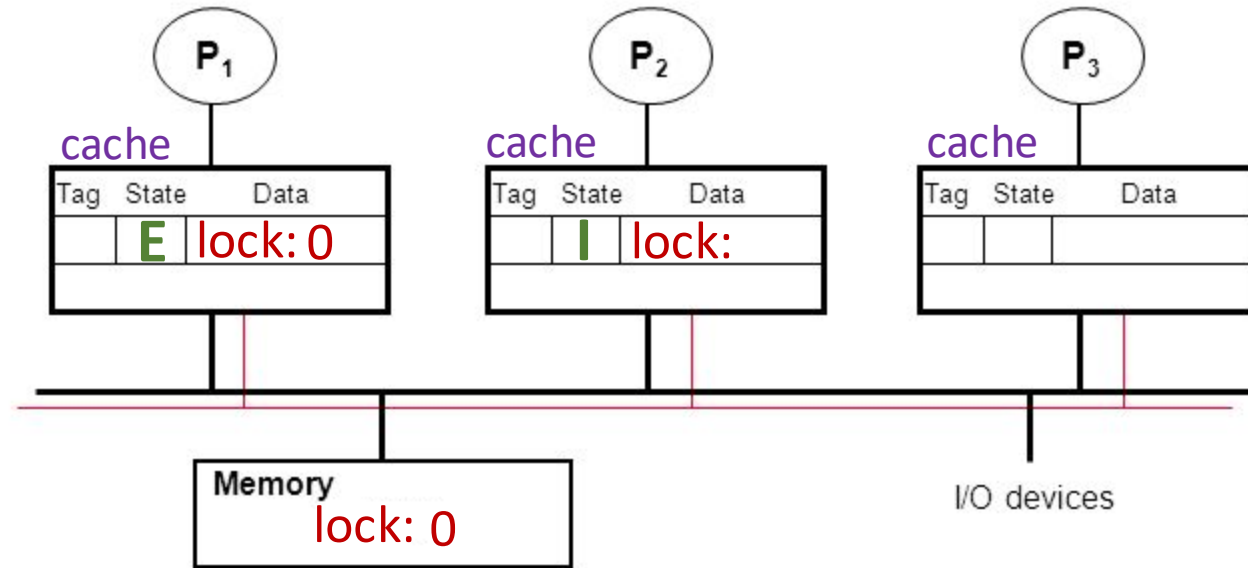
```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try: load lock, R0
    test R0
    bnz try
    store lock, 1
}
```

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

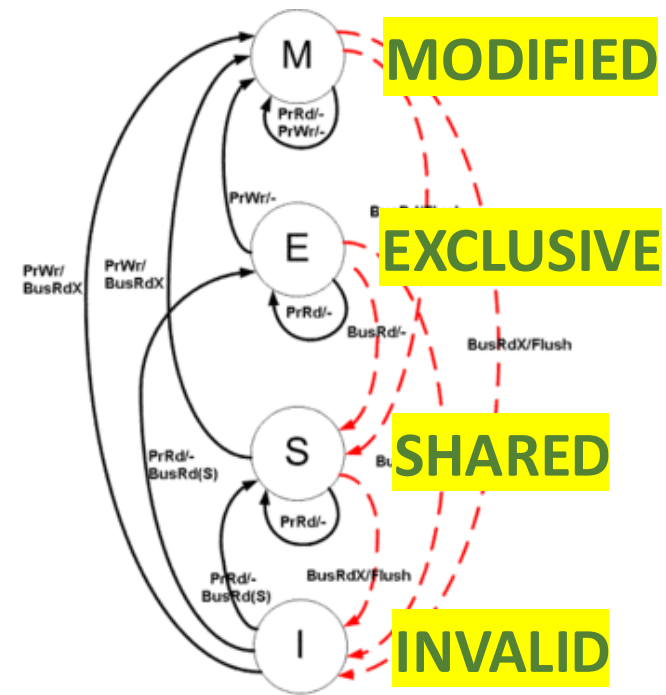
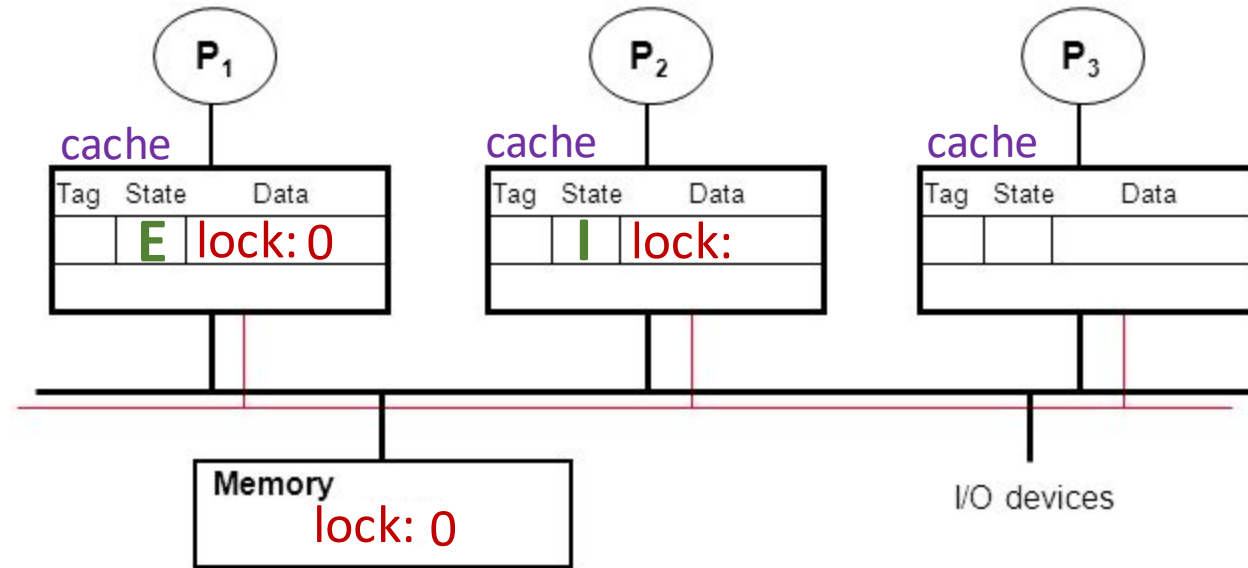
```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try: load lock, R0
    test R0
    bnz try
    store lock, 1
}
```

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

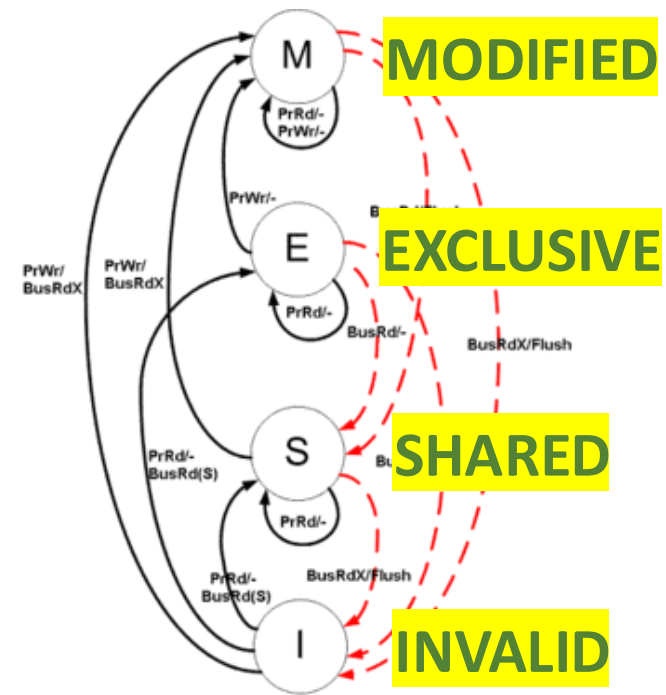
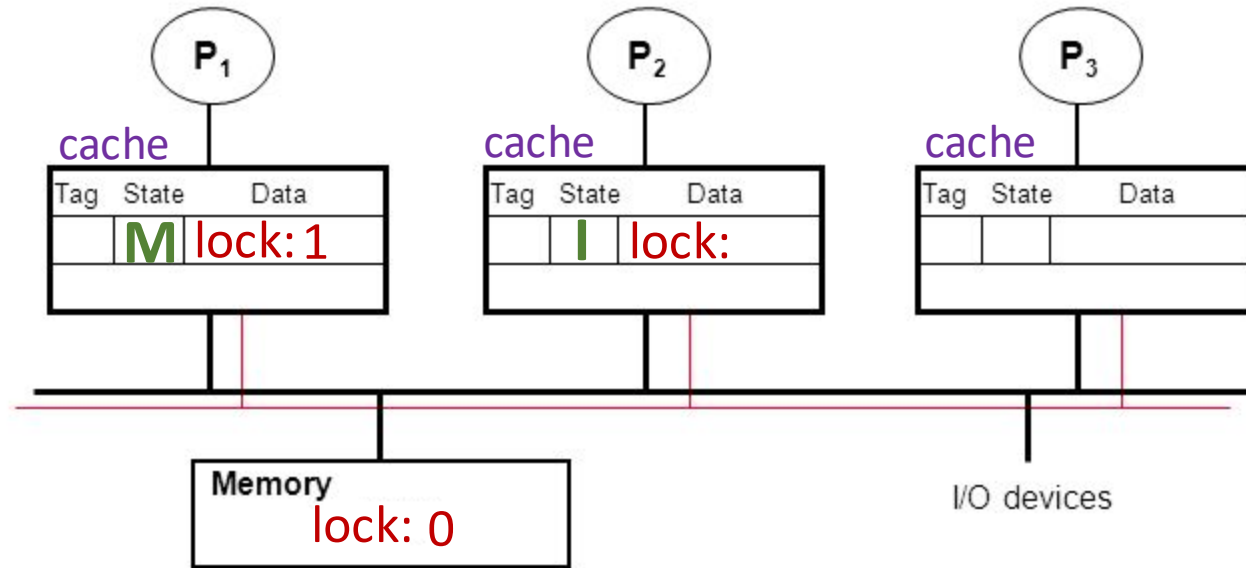
```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try: load lock, R0
    test R0
    bnz try
    store lock, 1
}
```

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



P2

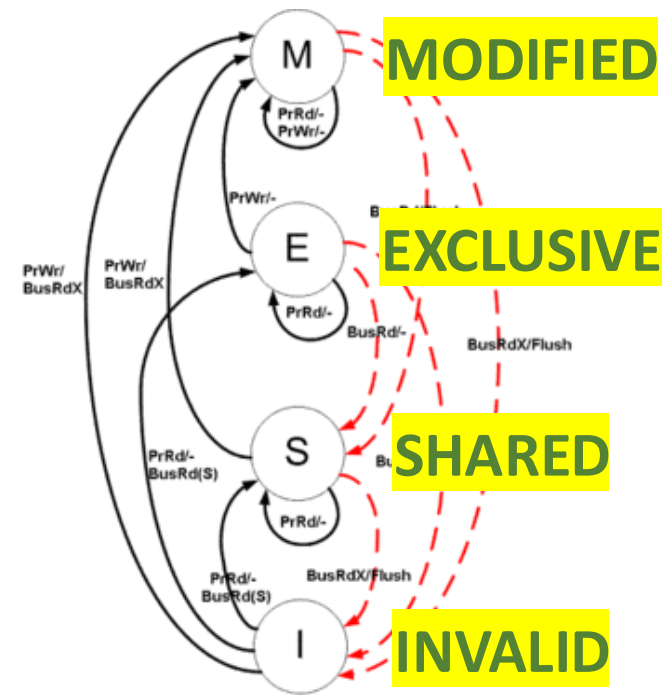
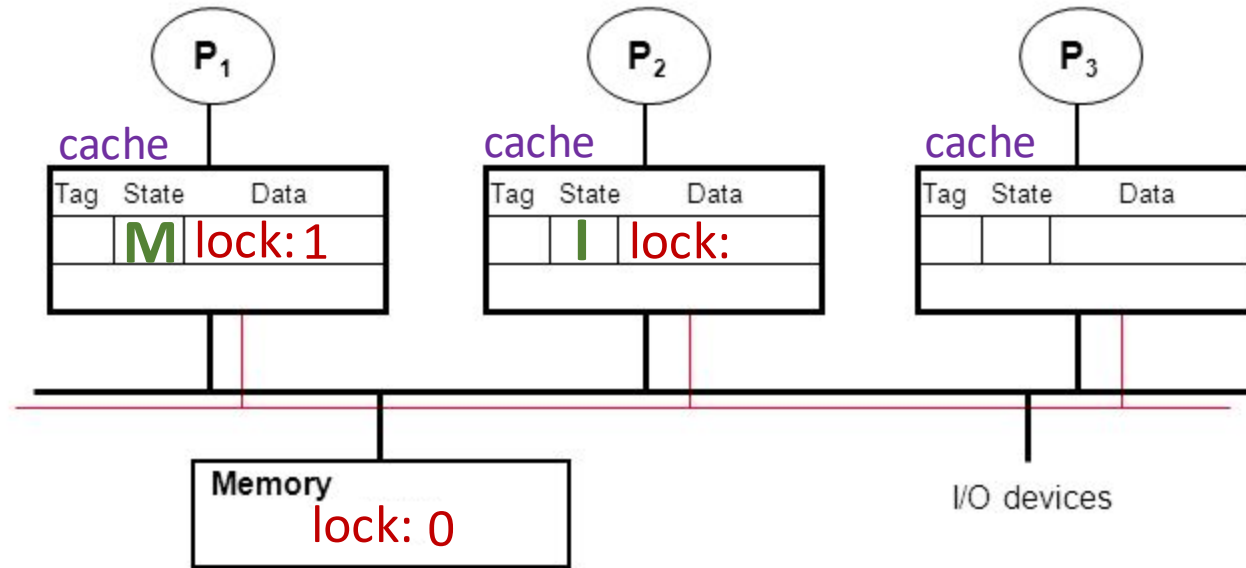
```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try: load lock, R0
    test R0
    bnz try
    store lock, 1
}
```



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



Cache Coherence Action Zone



P1

P2

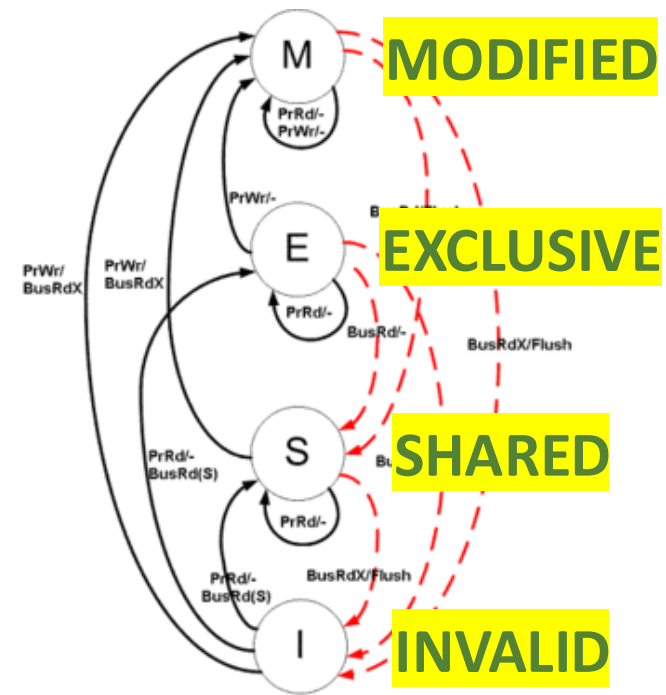
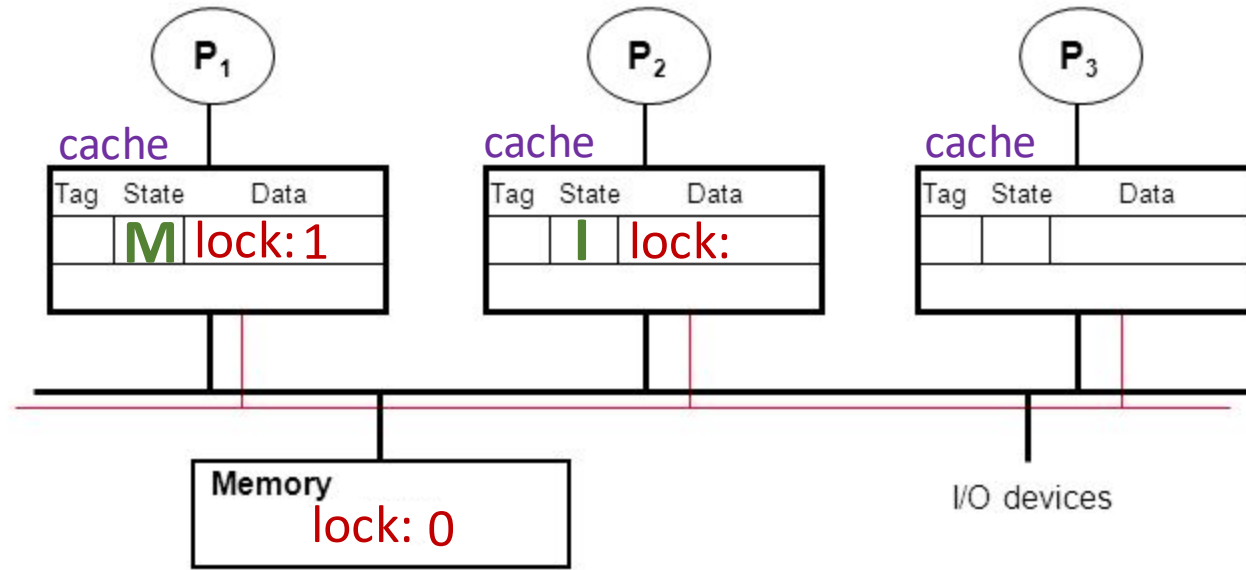
```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
      test R0
      bnz try
      store lock, 1
}
```



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



Cache Coherence Action Zone



P1

P2

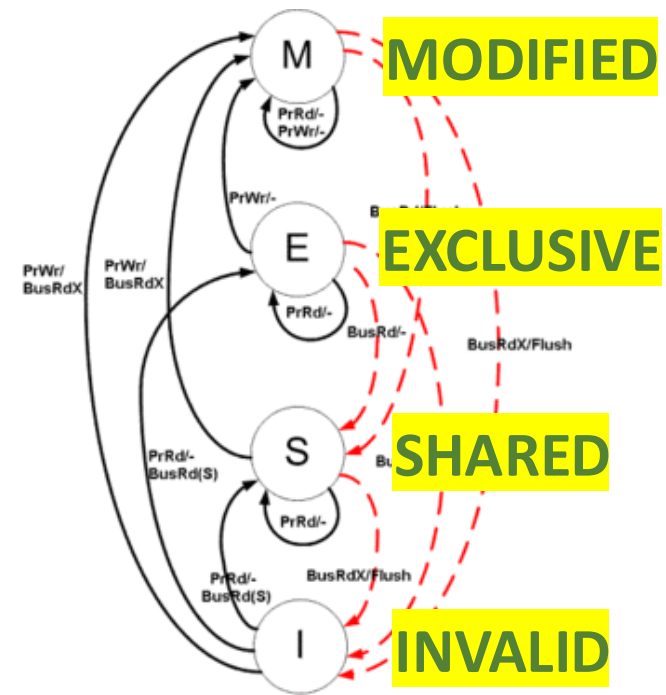
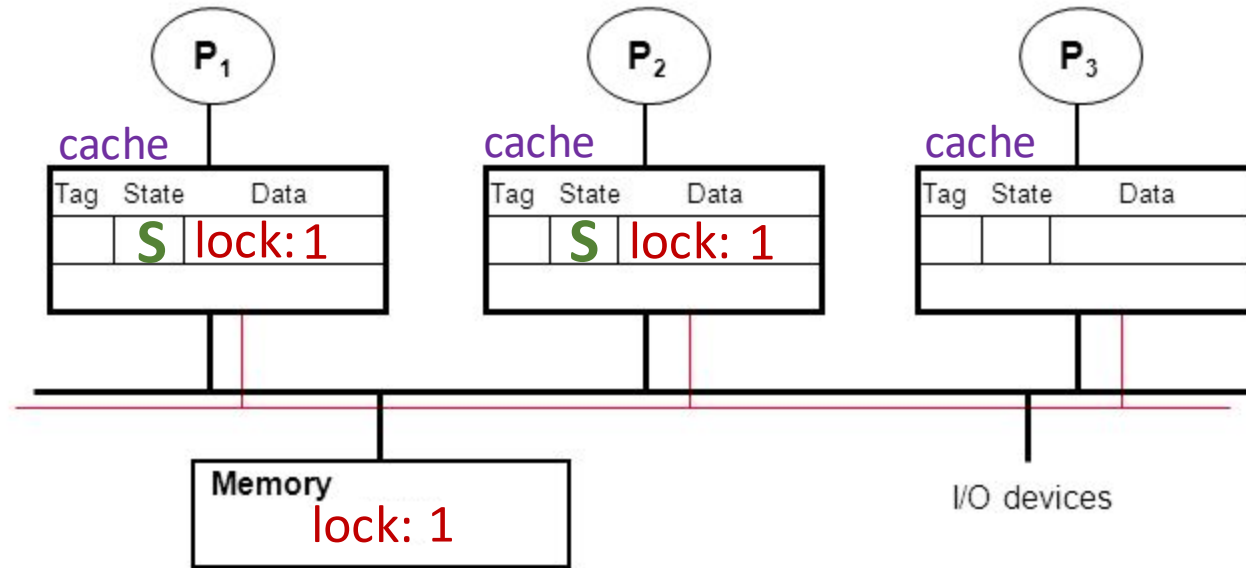
```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
      test R0
      bnz try
      store lock, 1
}
```



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



Cache Coherence Action Zone



P1

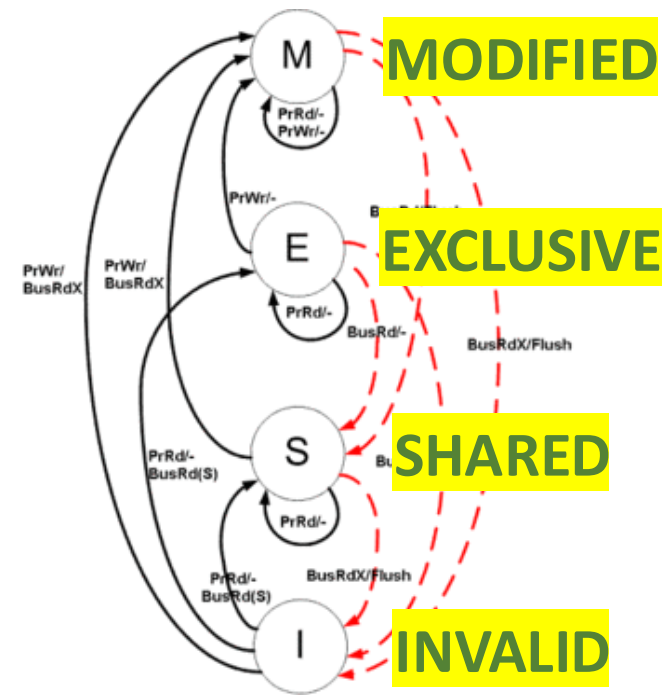
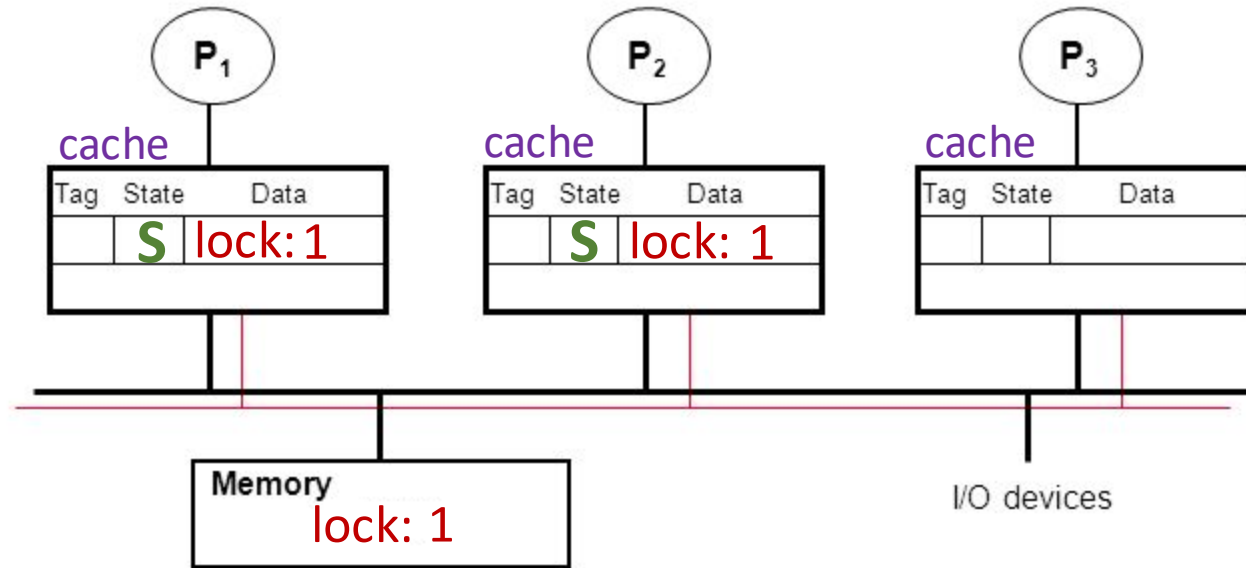
P2

```
// (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
}
```



Cache Coherence Action Zone



P1

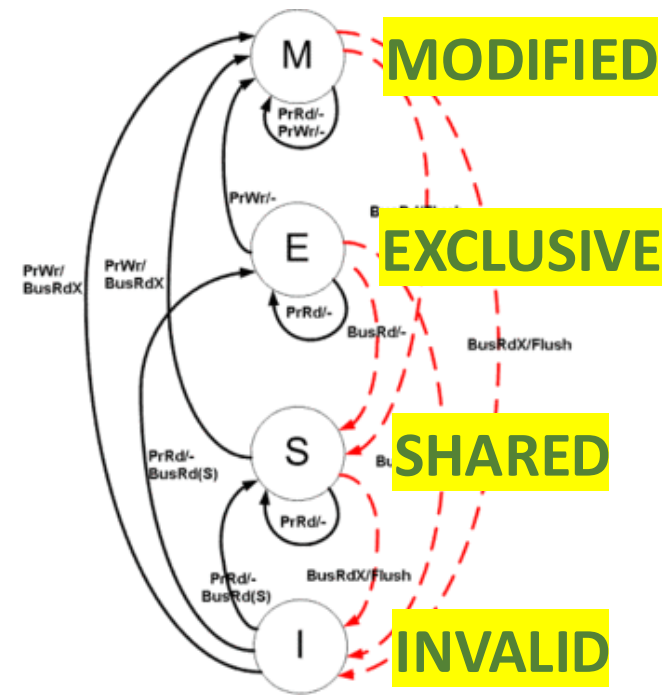
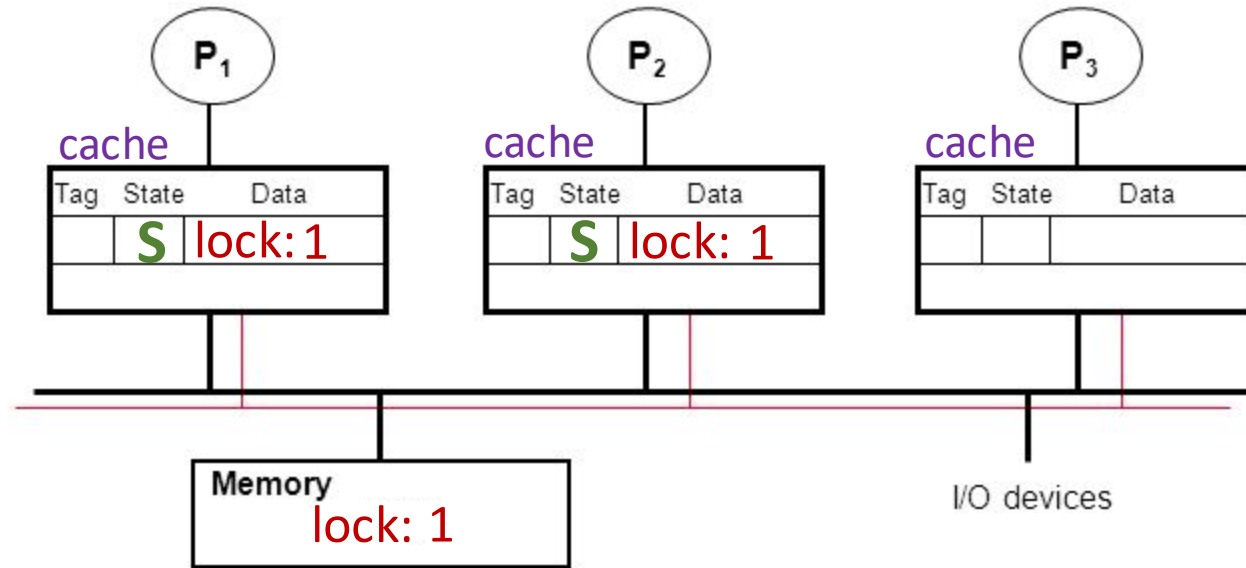
P2

```
// (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
}
```



Cache Coherence Action Zone



P1

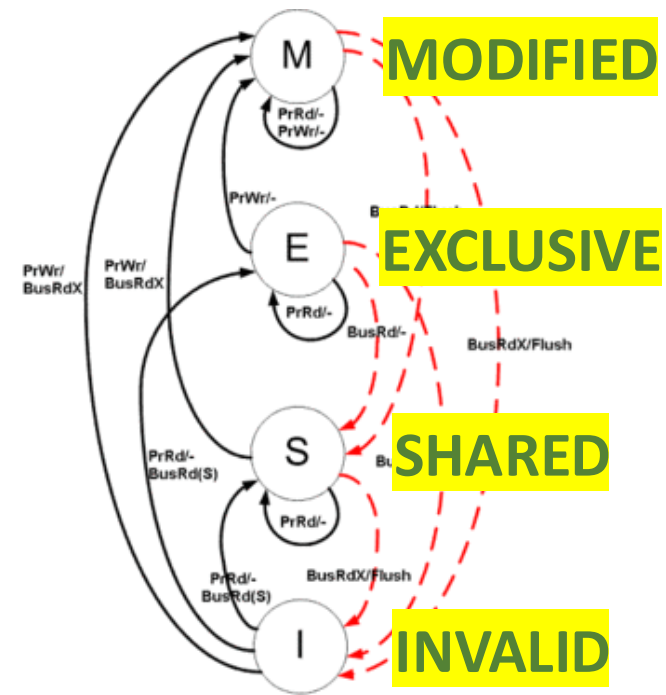
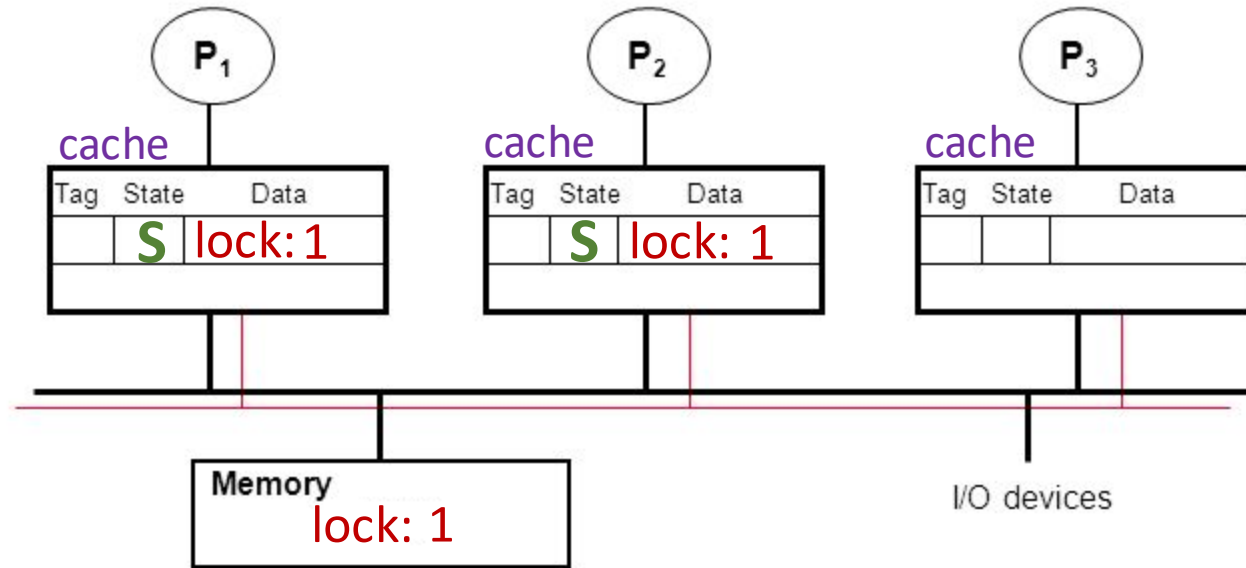
P2

```
// (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
}
```



Cache Coherence Action Zone



P1

P2

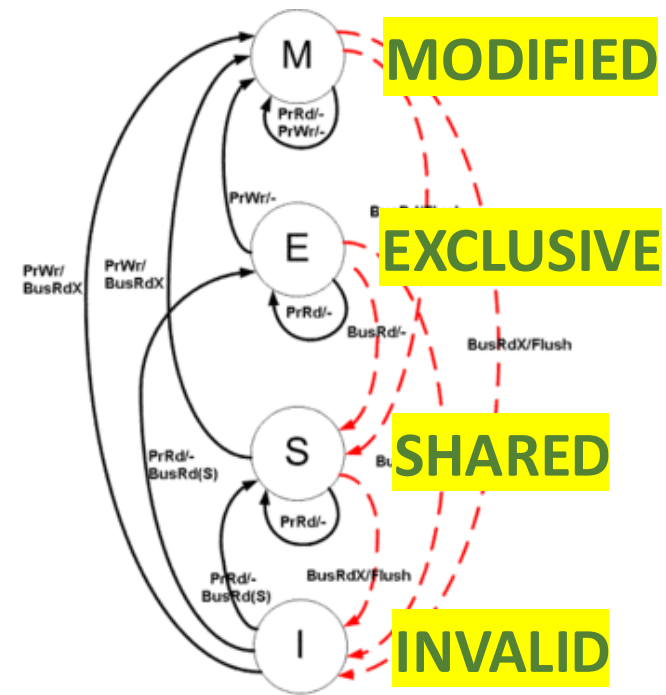
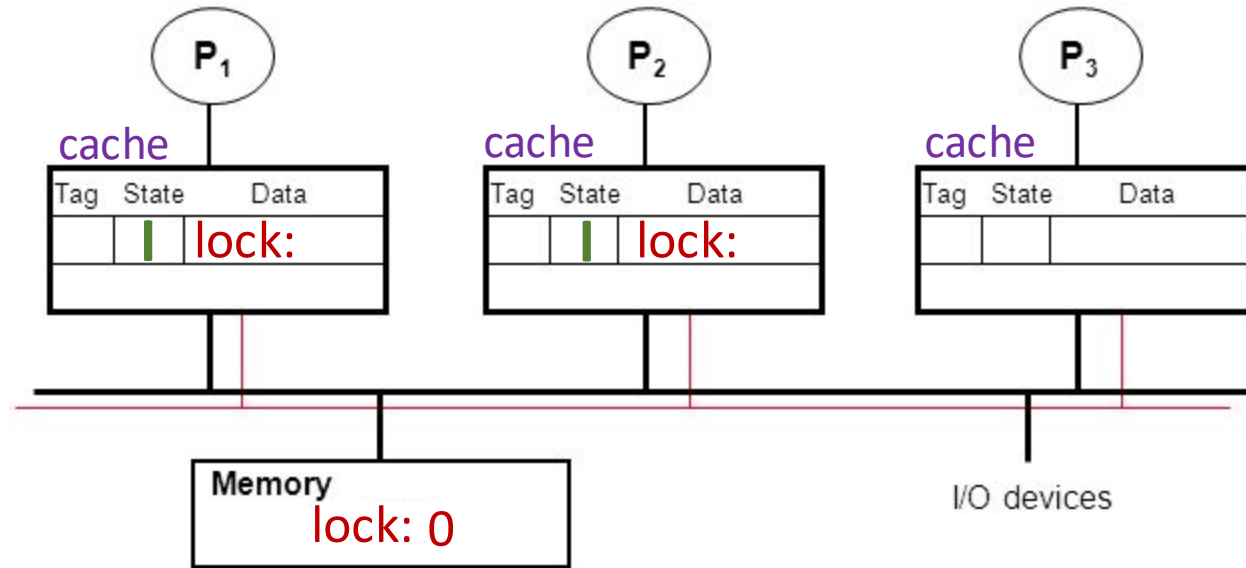
```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try: load lock, R0
    test R0
    bnz try
    store lock, 1
}
```

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



SAFE!

Cache Coherence Action Zone II



P1

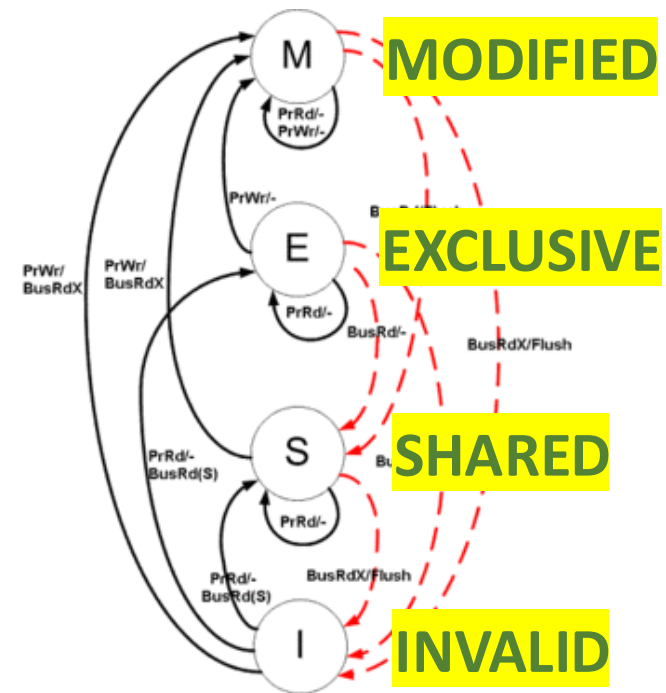
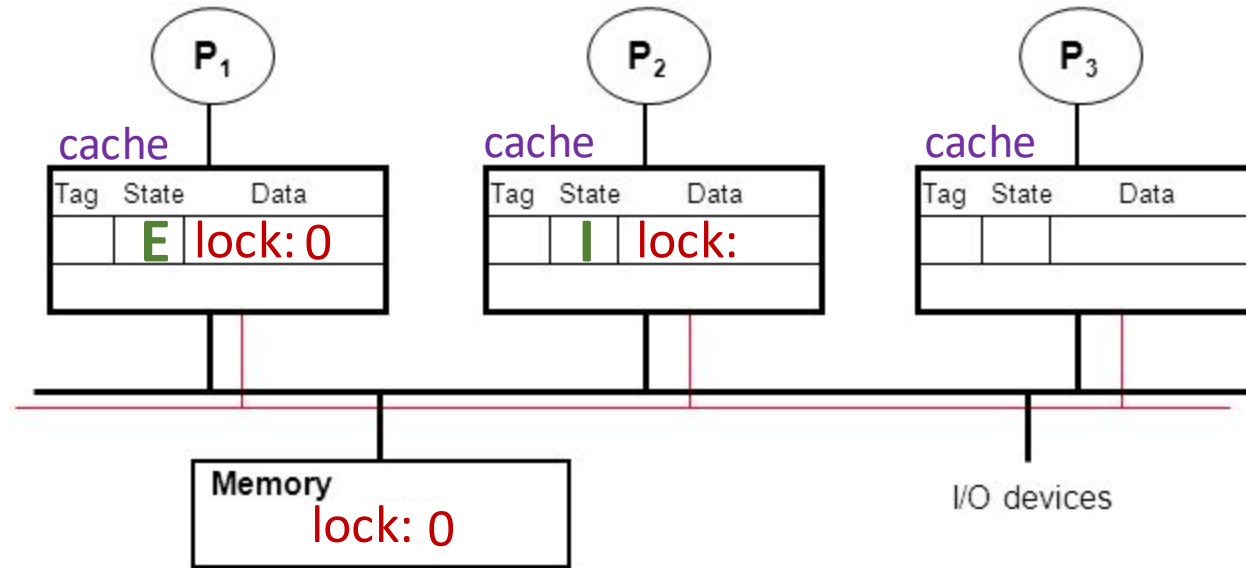
```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try: load lock, R0
    test R0
    bnz try
    store lock, 1
}
```

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

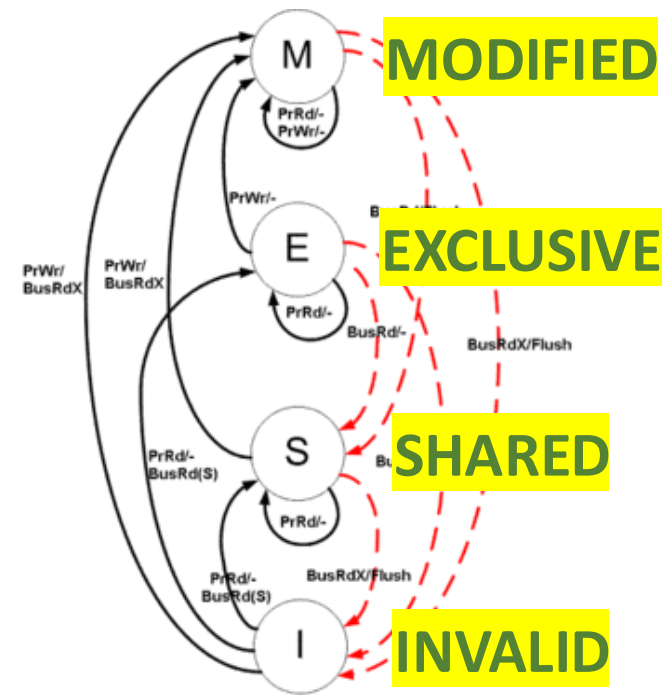
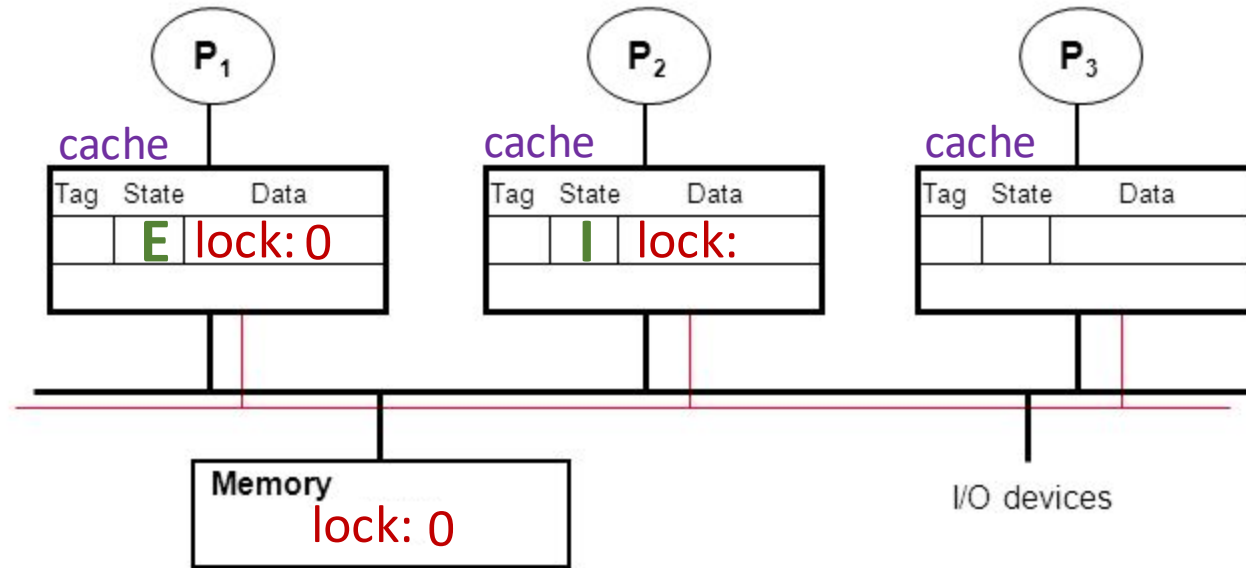
```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try: load lock, R0
    test R0
    bnz try
    store lock, 1
}
```

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

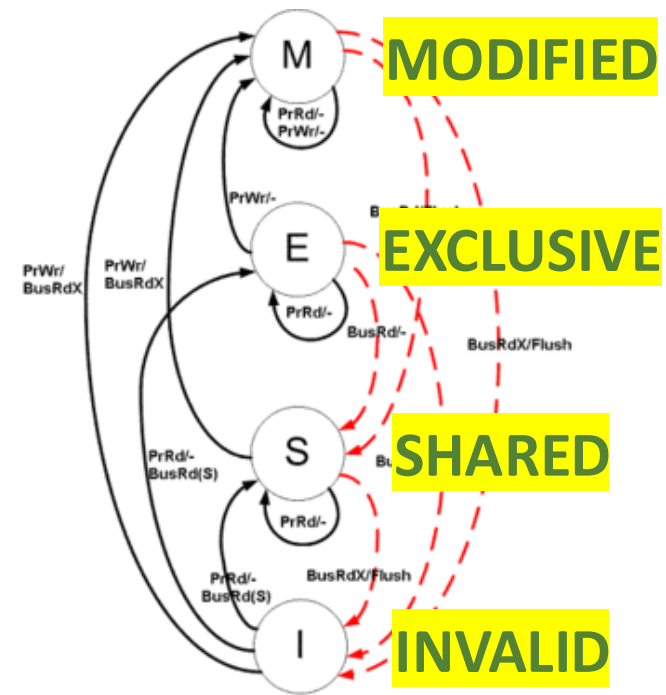
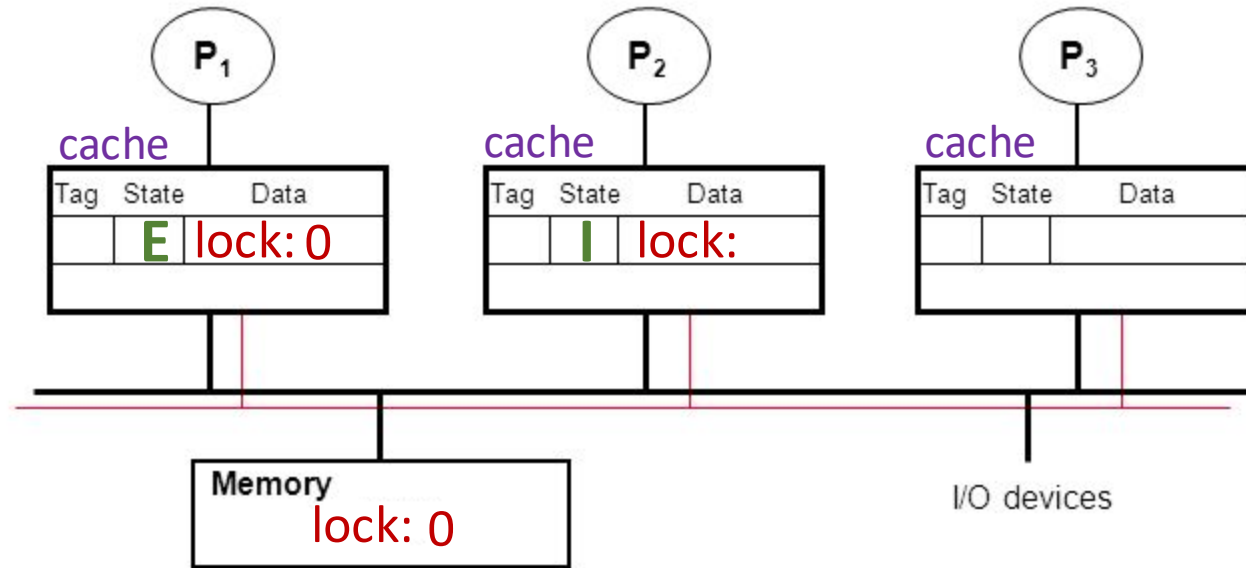
```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try: load lock, R0
    test R0
    bnz try
    store lock, 1
}
```

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
}
    
```



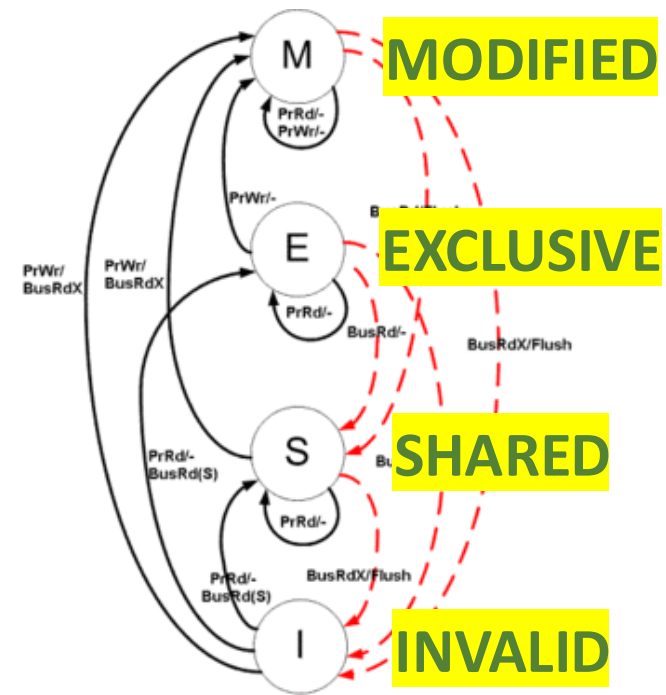
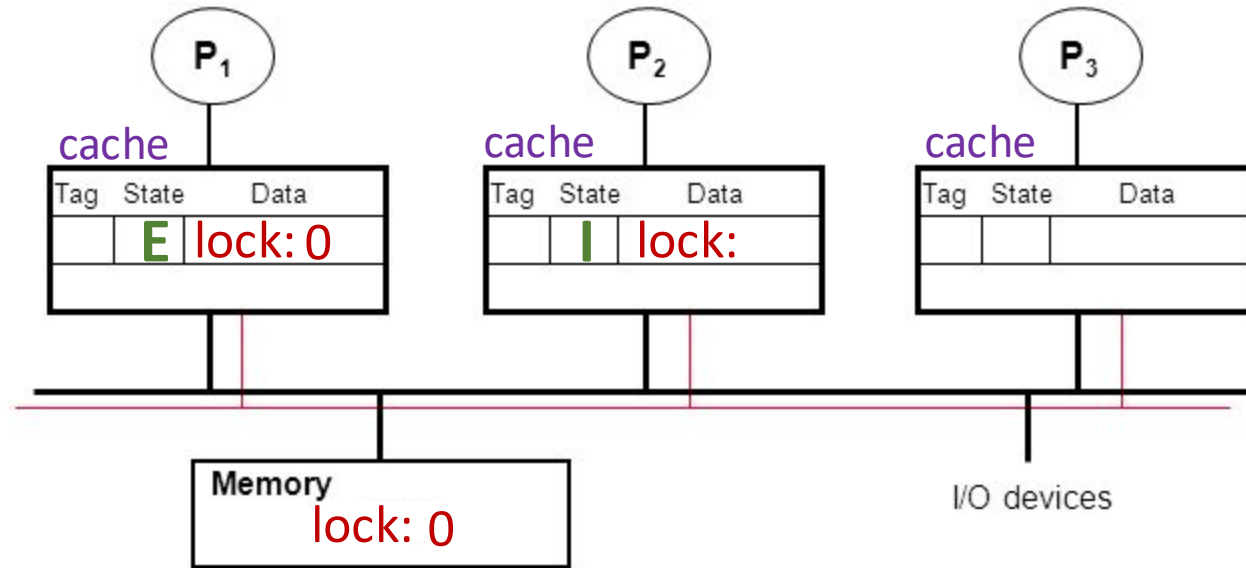
P2

```

// (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

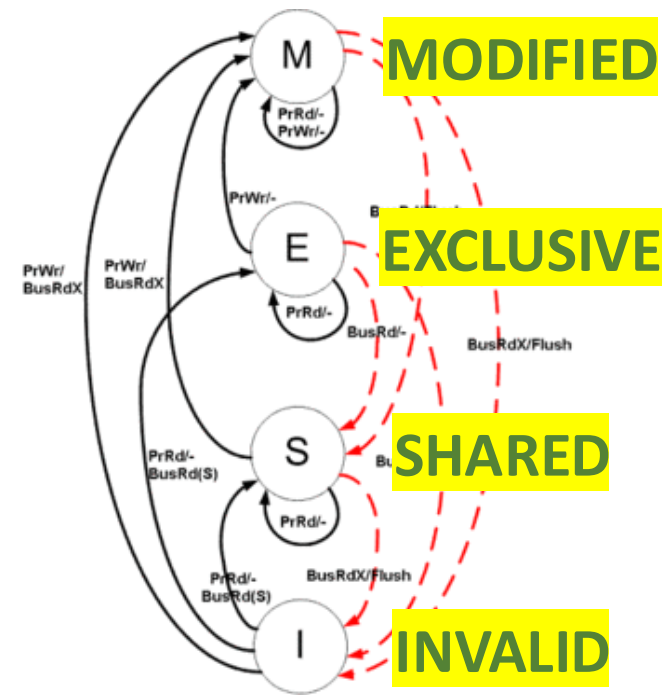
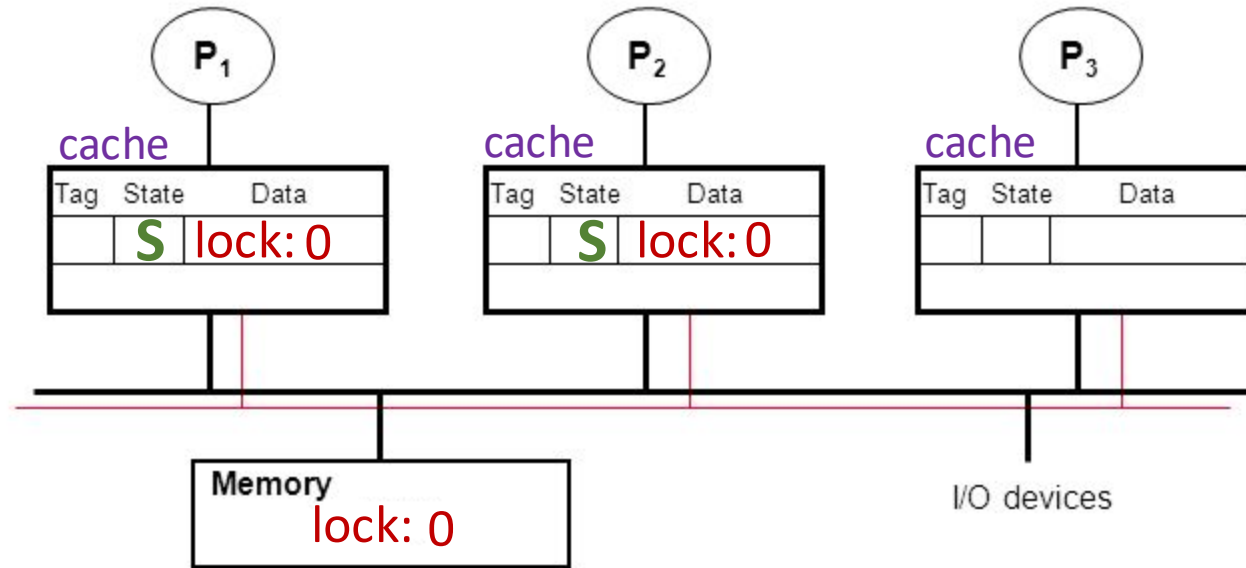
```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try: load lock, R0
    test R0
    bnz try
    store lock, 1
}
```

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

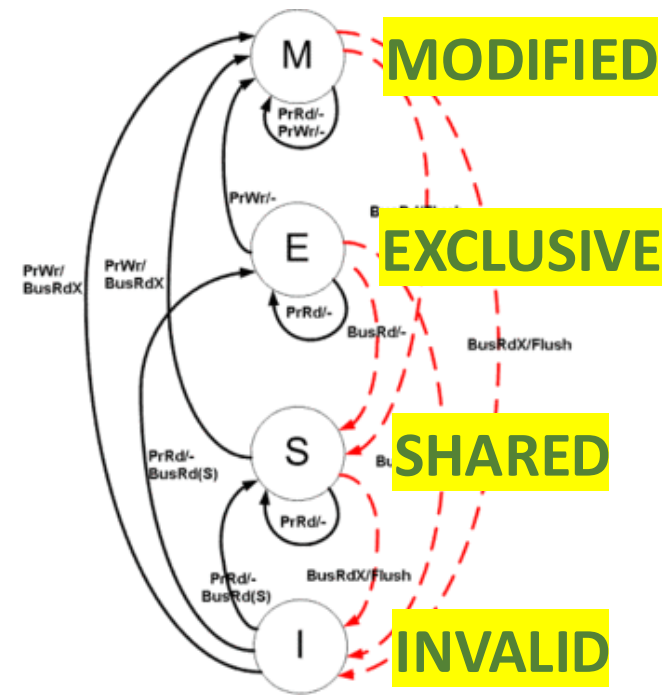
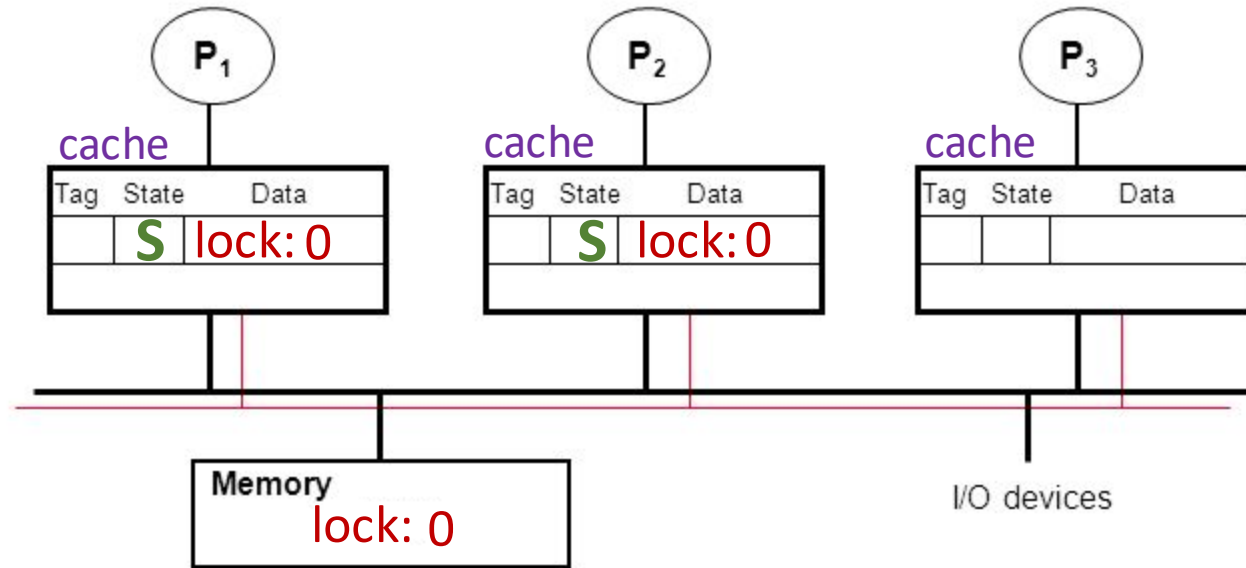
```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try: load lock, R0
    test R0
    bnz try
    store lock, 1
}
```

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

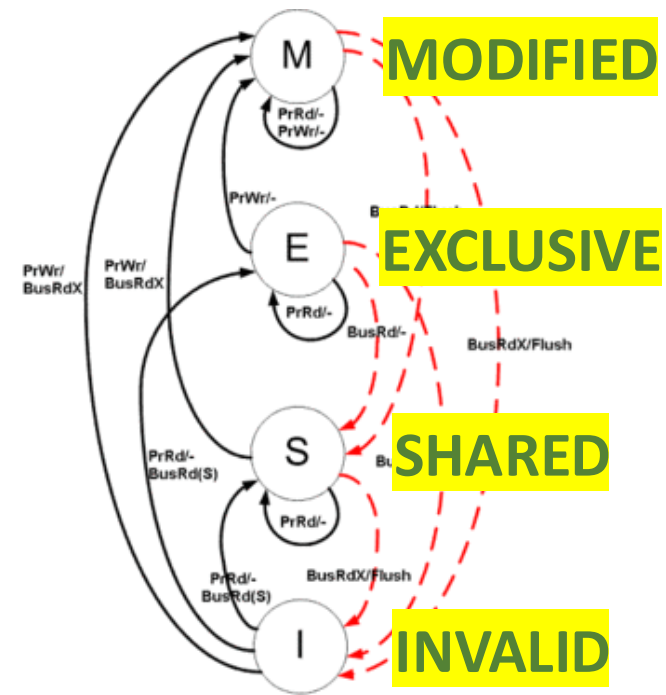
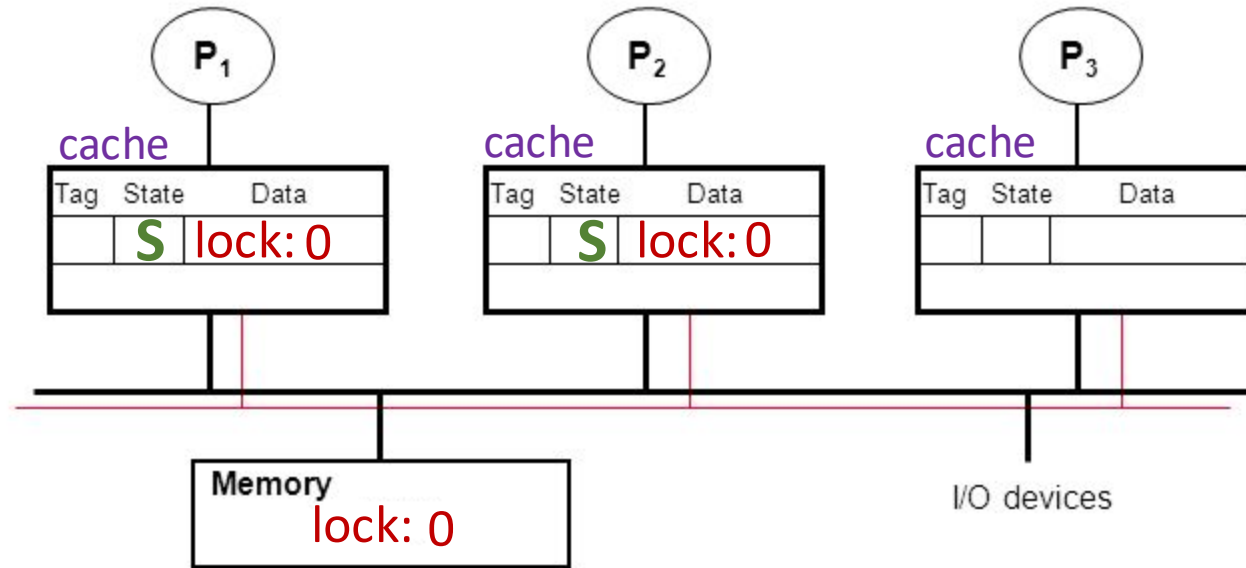
```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try: load lock, R0
    test R0
    bnz try
    store lock, 1
}
```

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

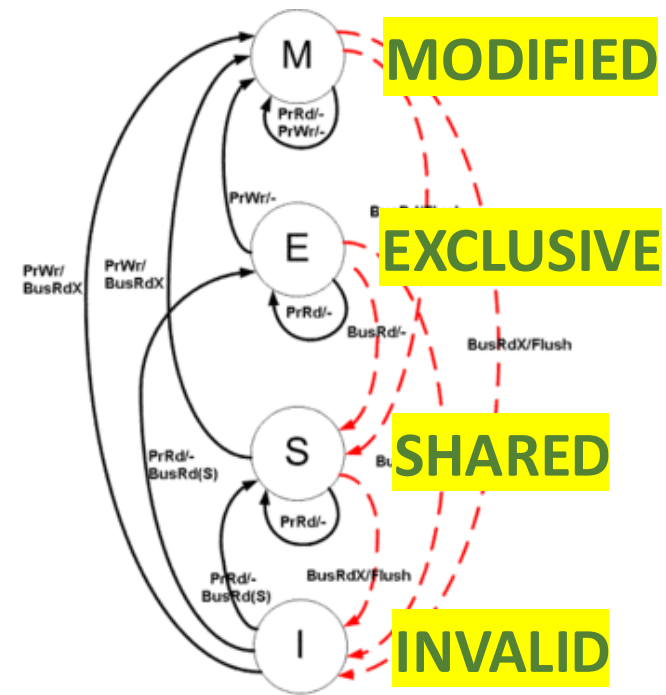
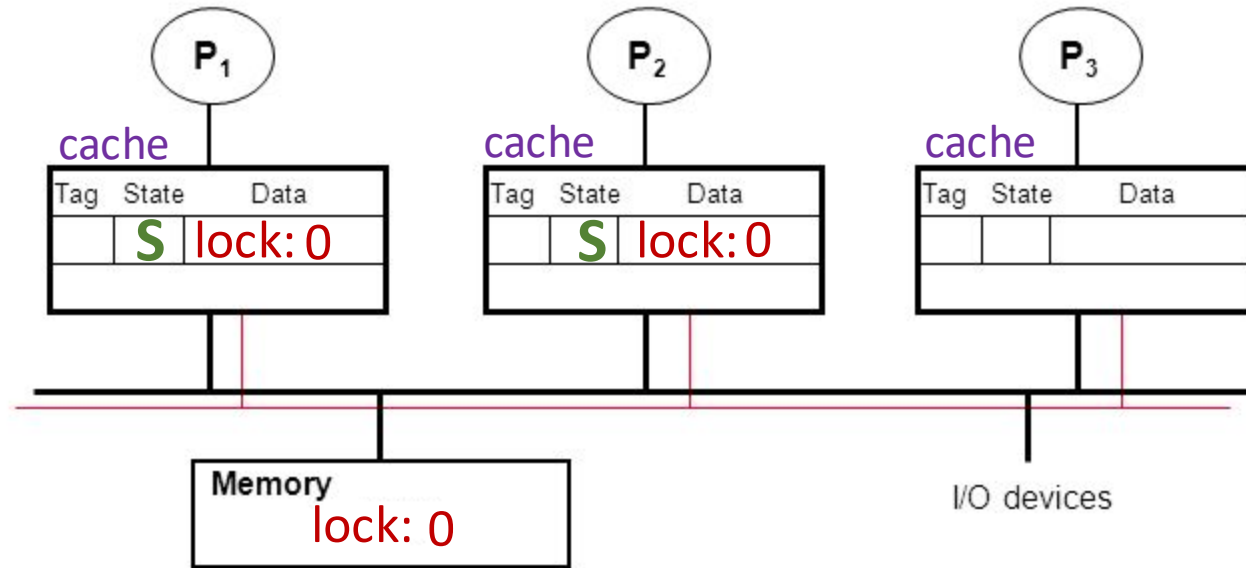
```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
      test R0
      bnz try
      store lock, 1
}
```

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

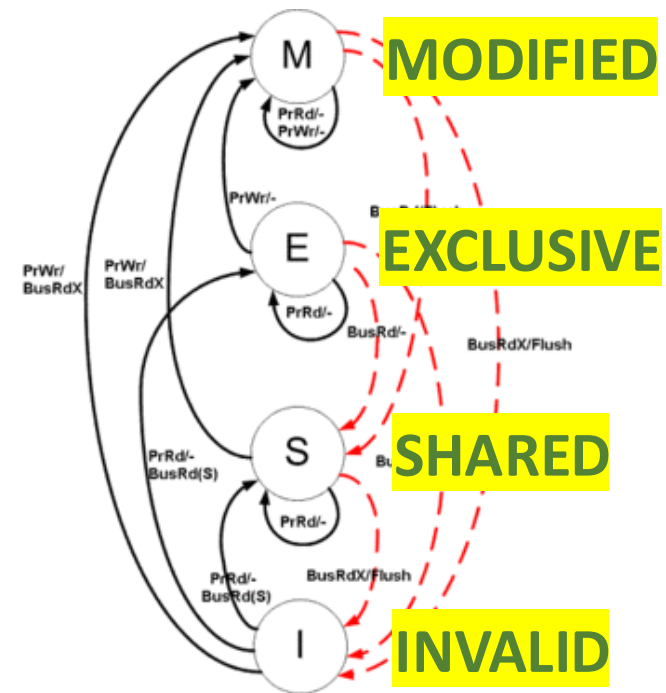
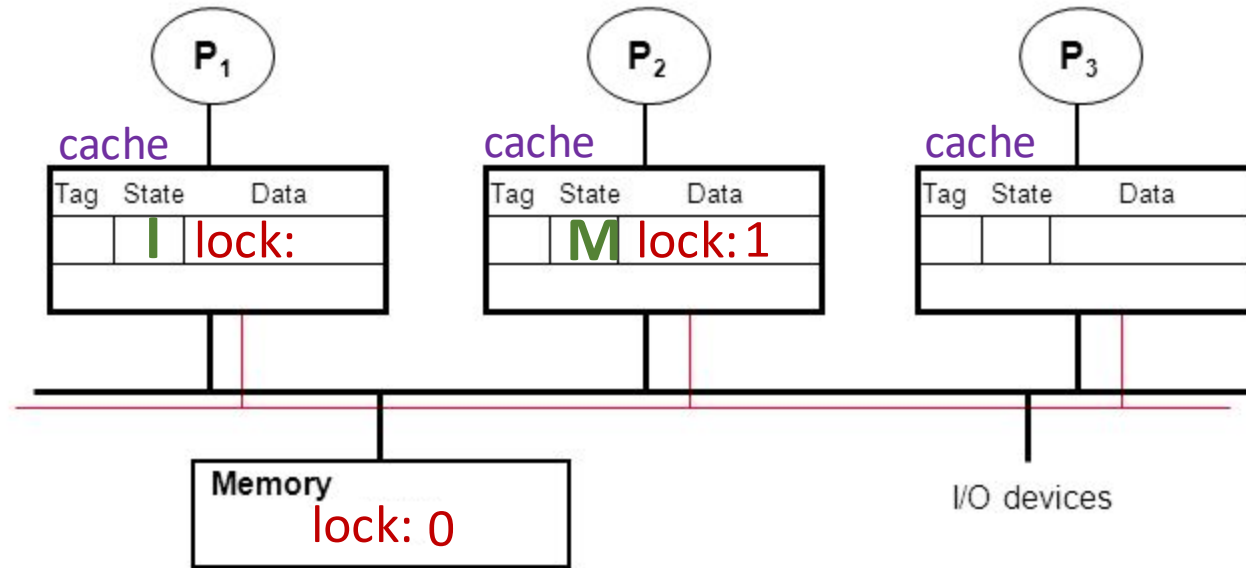
```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try: load lock, R0
    test R0
    bnz try
    store lock, 1
}
```

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



P2

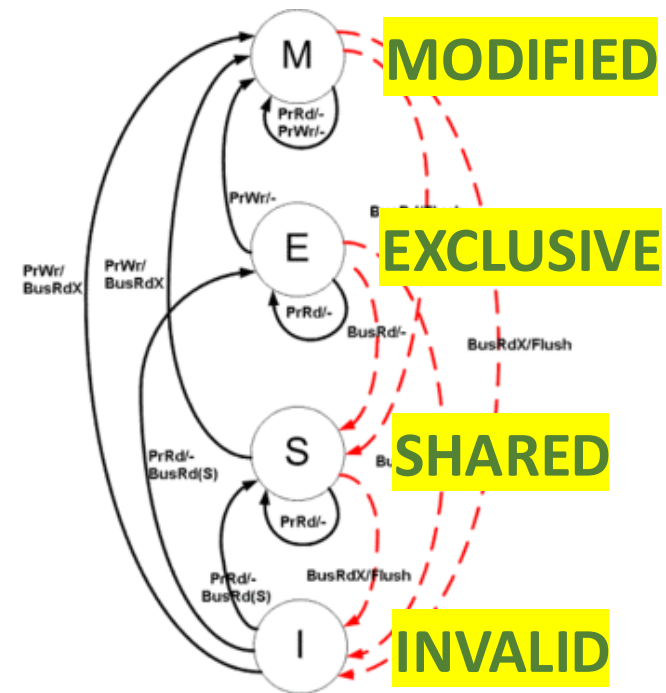
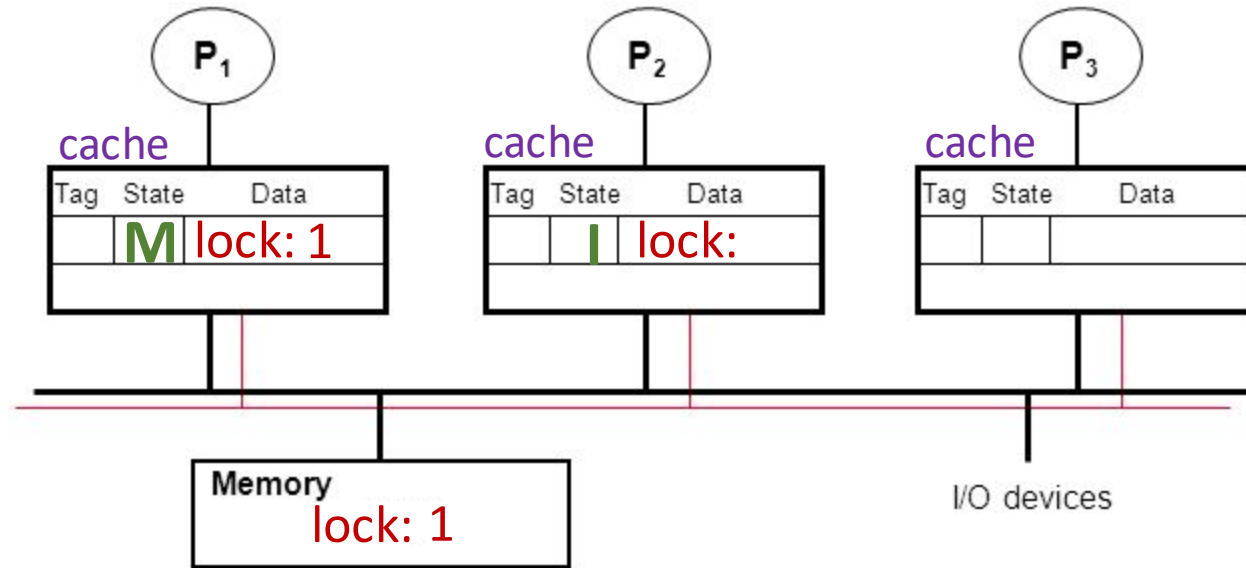
P1

```
// (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
}
```



Cache Coherence Action Zone II



P1

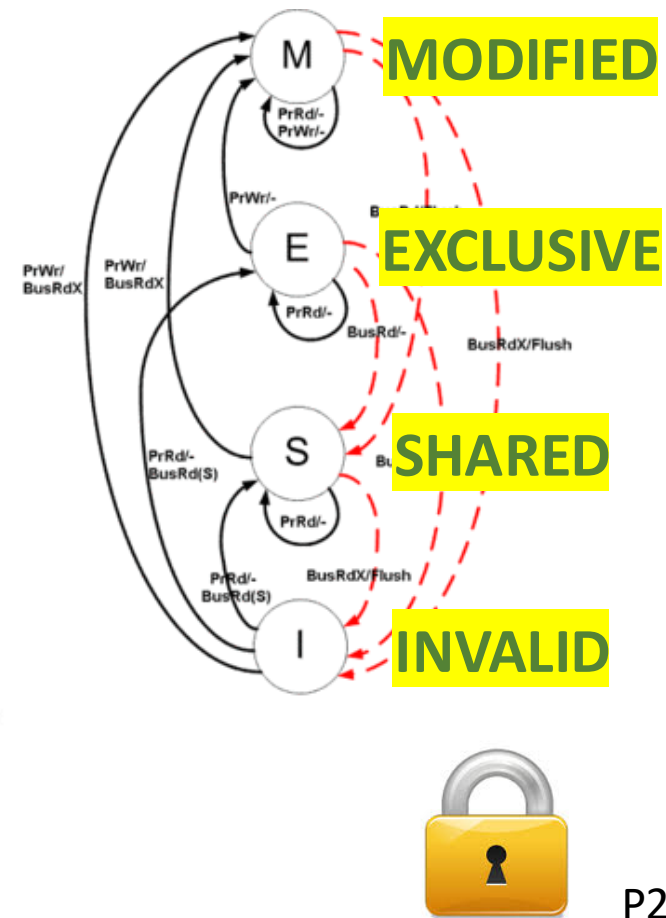
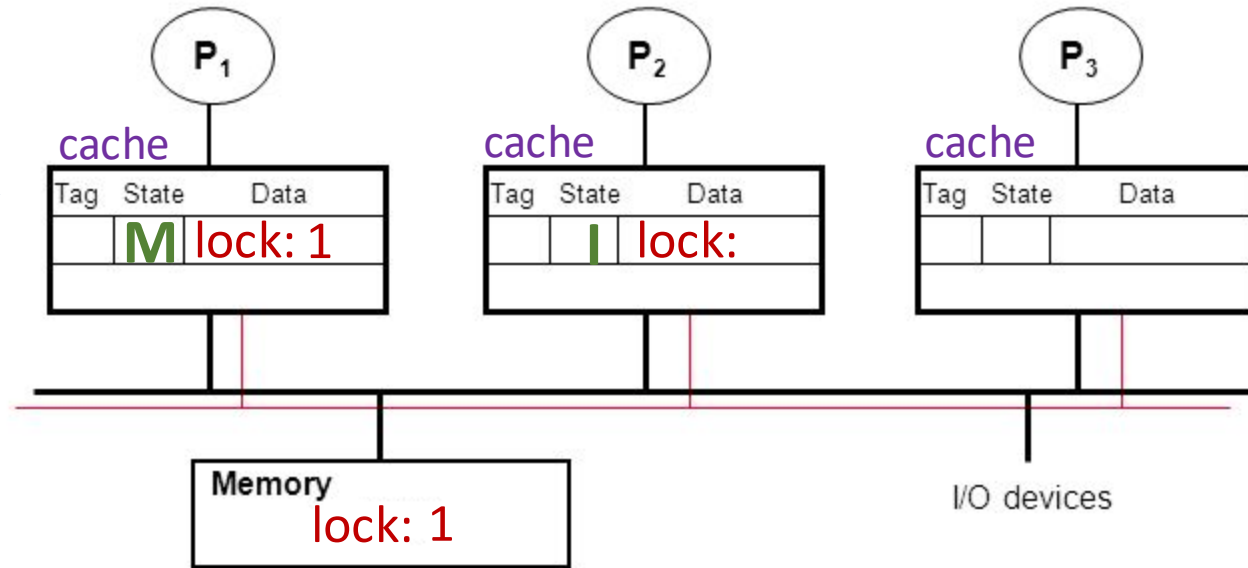
P2

```
// (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
}
```



Cache Coherence Action Zone II



P1

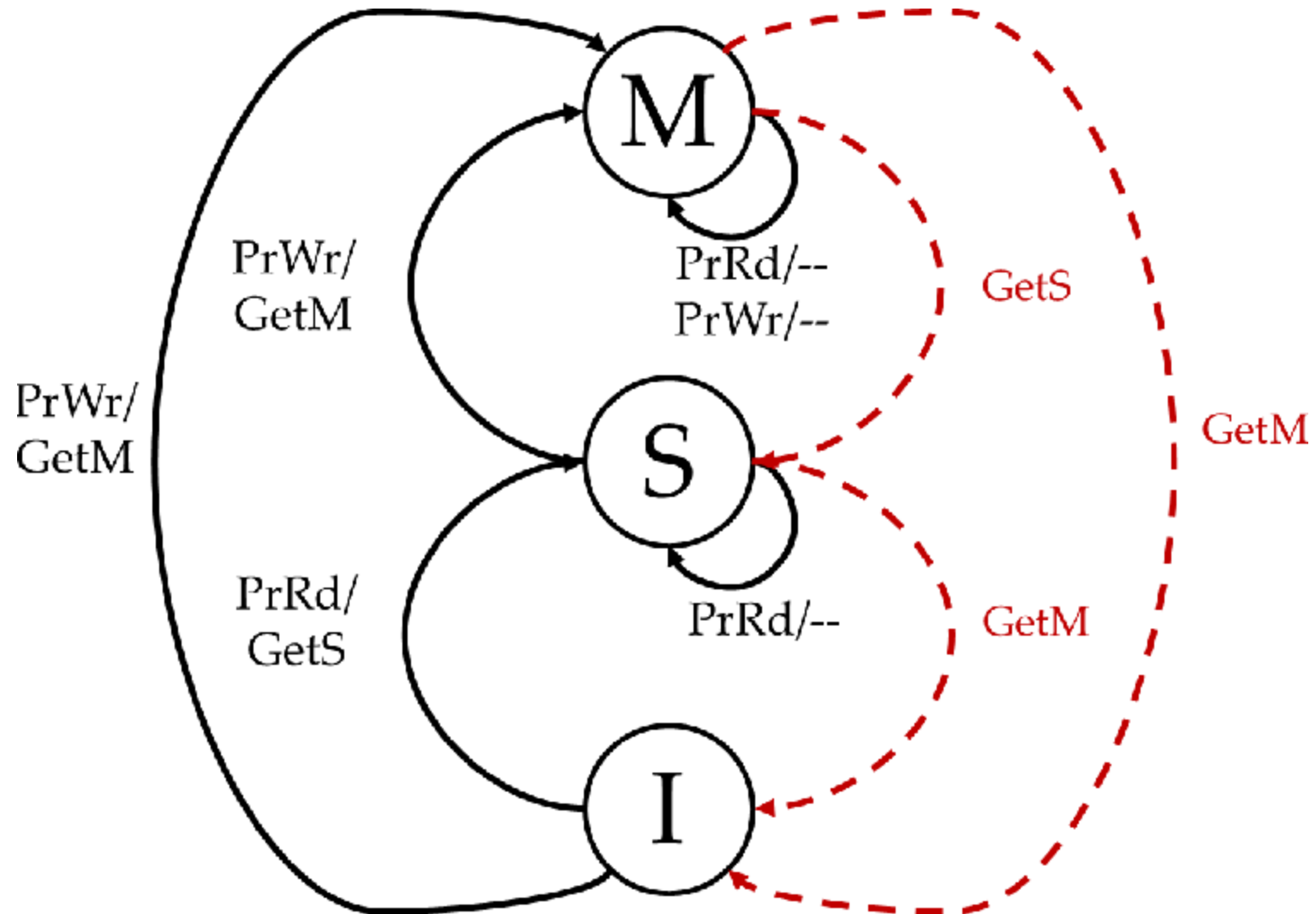
P2

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

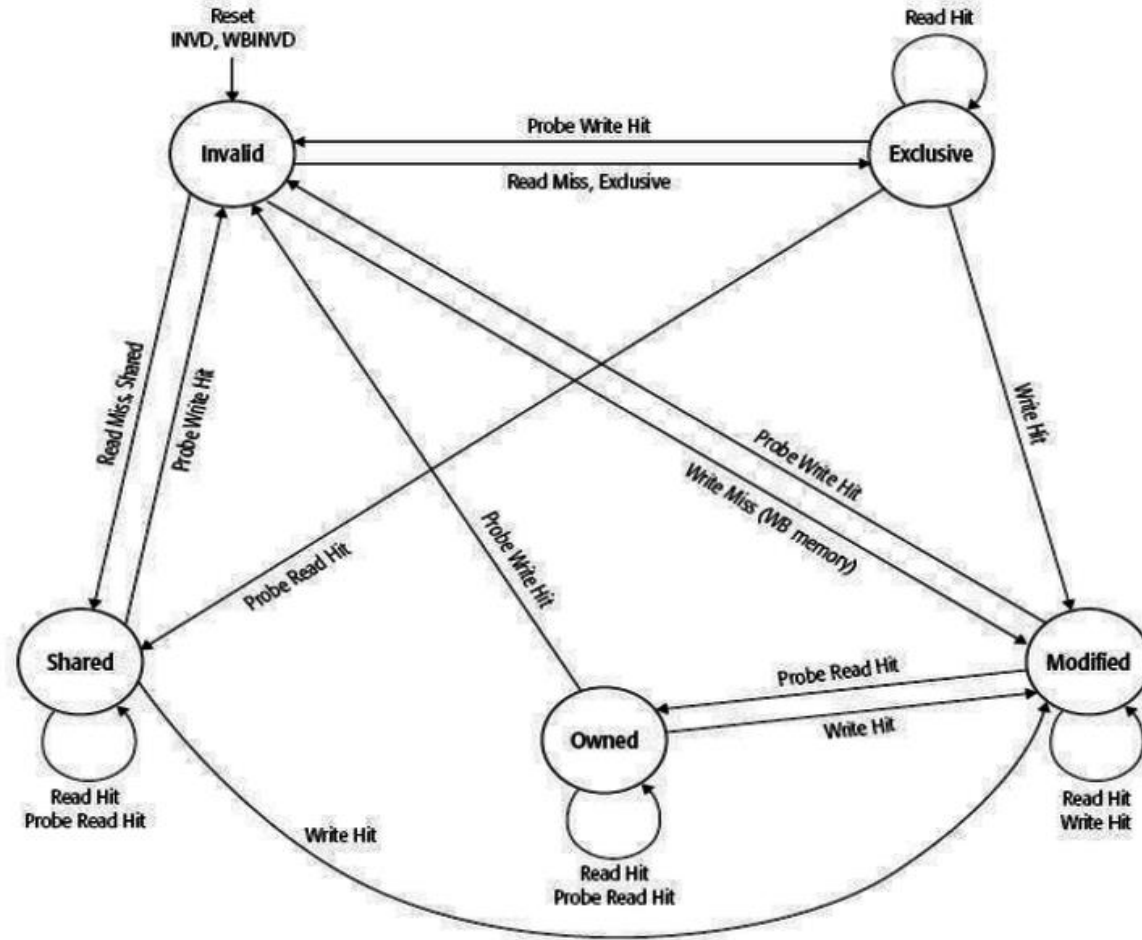
```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try: load lock, R0
    test R0
    bnz try
    store lock, 1
}
```



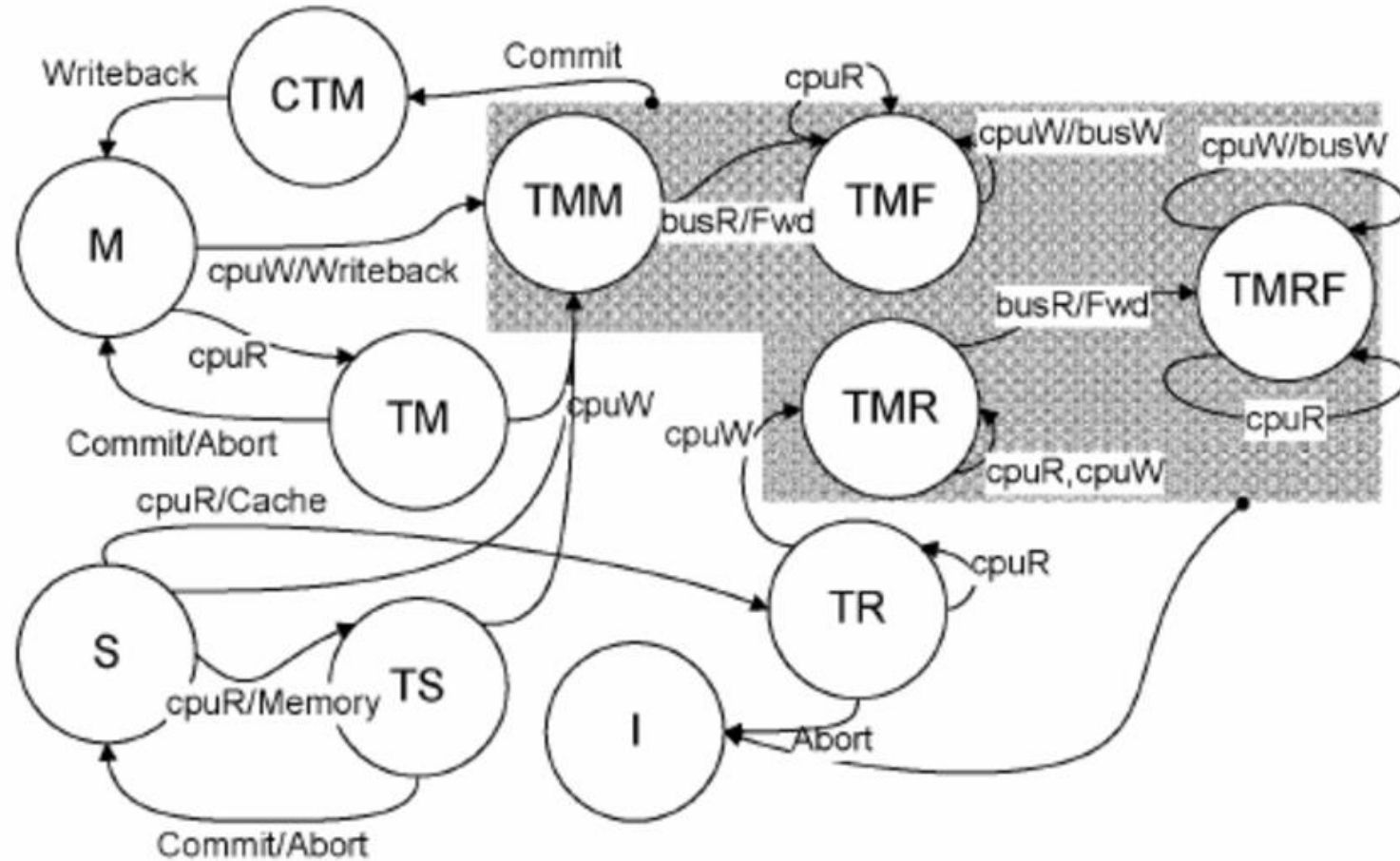
Other Coherence Protocols: MSI



Other Coherence Protocols: MOESI

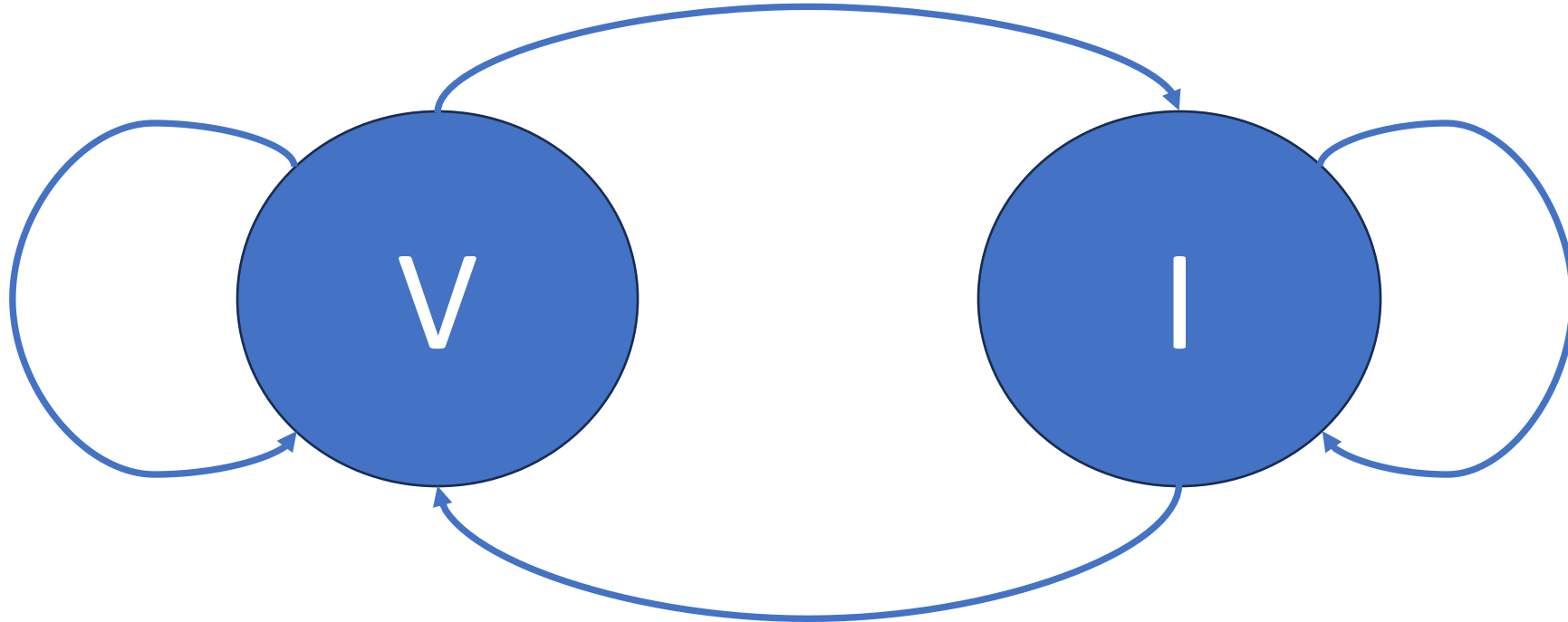


Other Coherence Protocols: FRMSI



Exercise: VI Coherence

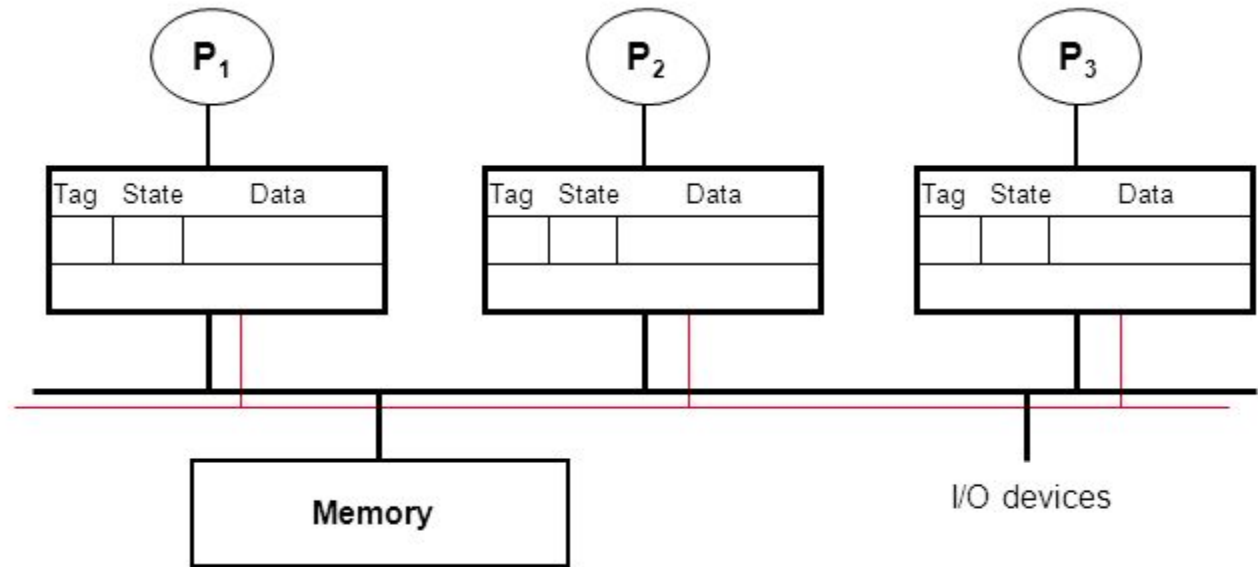
- 2 states: Valid, Invalid
- What are the processor events that drive transitions?
- What are bus events that drive transitions?



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
<pre>int TST(addr) { atomic { ret = *addr; if(!*addr) *addr = 1; return ret; } }</pre>	<pre>bool cas(addr, old, new) { atomic { if(*addr == old) { *addr = new; return true; } return false; } }</pre>	<pre>int XCHG(addr, val) { atomic { ret = *addr; *addr = val; return ret; } }</pre>	<pre>bool LLSC(addr, val) { ret = *addr; atomic { if(*addr == ret) { *addr = val; return true; } return false; } }</pre>

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
<pre>int TST(addr) { atomic { ret = *addr; if(!*addr) *addr = 1; return ret; } }</pre>	<pre>bool cas(addr, old, new) { atomic { if(*addr == old) { *addr = new; return true; } return false; } }</pre>	<pre>int XCHG(addr, val) { atomic { ret = *addr; *addr = val; return ret; } }</pre>	<pre>bool LLSC(addr, val) { ret = *addr; atomic { if(*addr == ret) { *addr = val; return true; } return false; } }</pre>

```
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
<pre>int TST(addr) { atomic { ret = *addr; if(!*addr) *addr = 1; return ret; } }</pre>	<pre>bool cas(addr, old, new) { atomic { if(*addr == old) { *addr = new; return true; } return false; } }</pre>	<pre>int XCHG(addr, val) { atomic { ret = *addr; *addr = val; return ret; } }</pre>	<pre>bool LLSC(addr, val) { ret = *addr; atomic { if(*addr == ret) { *addr = val; return true; } return false; } }</pre>

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

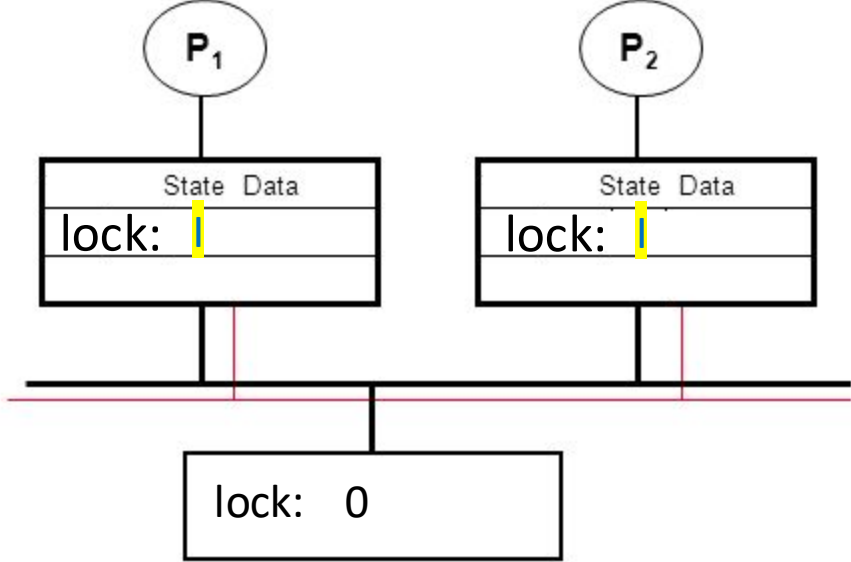
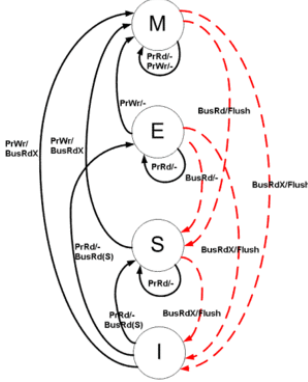
PPC, Alpha, MIPS

```
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



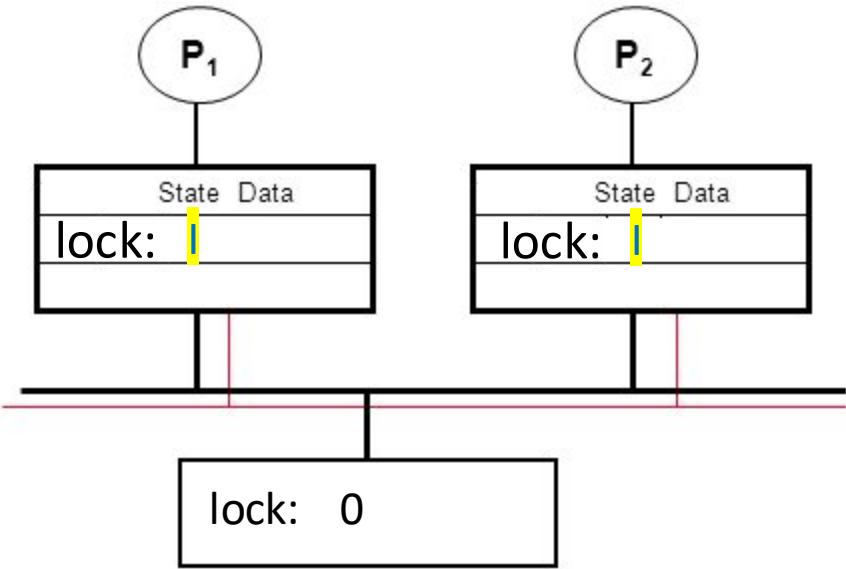
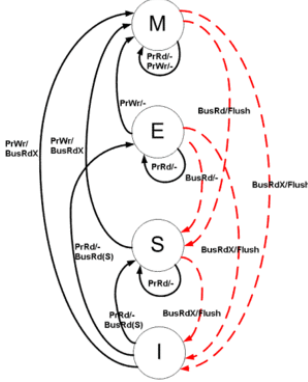
```

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



```

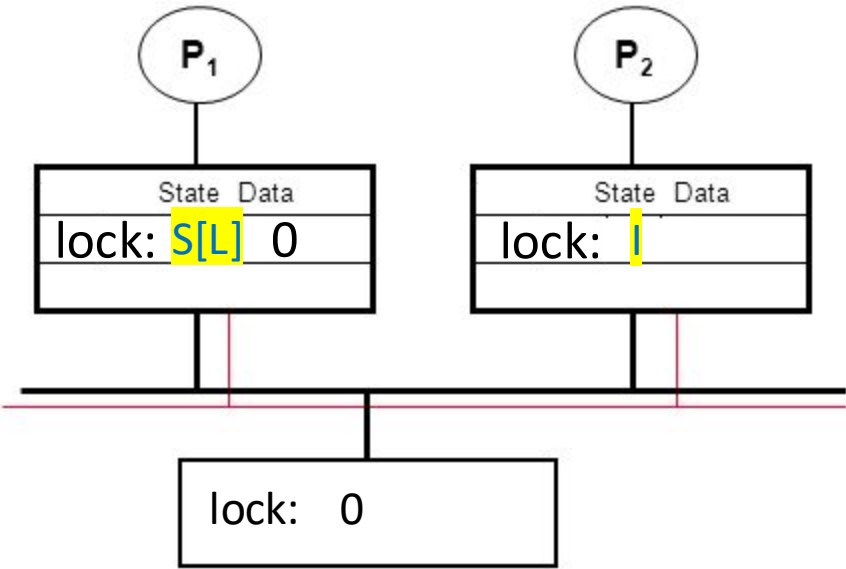
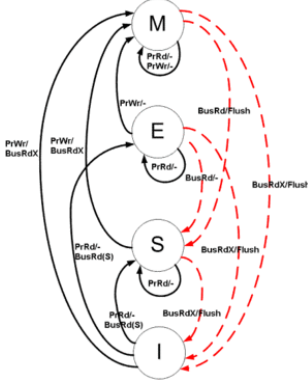
P1
lock(lock) {
  while(1) {
    old = ll(lock);
    if(old == 0)
      if(sc(lock, 1))
        return;
  }
}
    
```



```

P2
lock(lock) {
  while(1) {
    old = ll(lock);
    if(old == 0)
      if(sc(lock, 1))
        return;
  }
}
    
```


LLSC Lock Action Zone



```

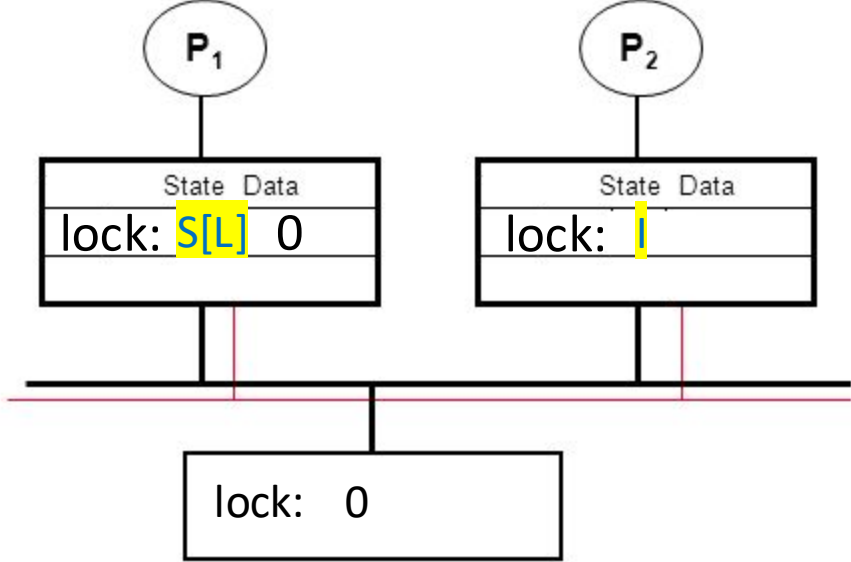
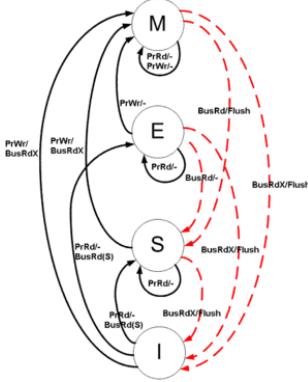
P1
lock(lock) {
  while(1) {
    old = ll(lock);
    if(old == 0)
      if(sc(lock, 1))
        return;
  }
}
    
```



```

P2
lock(lock) {
  while(1) {
    old = ll(lock);
    if(old == 0)
      if(sc(lock, 1))
        return;
  }
}
    
```

LLSC Lock Action Zone



```

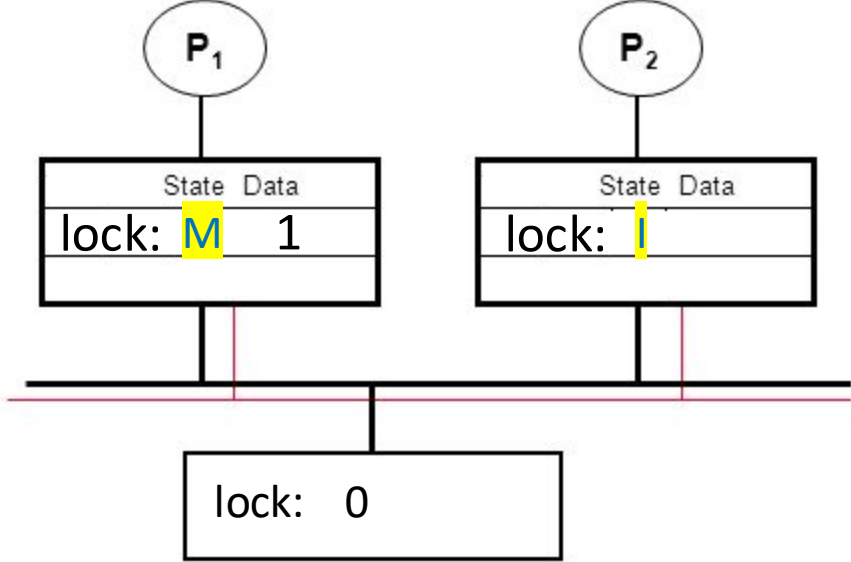
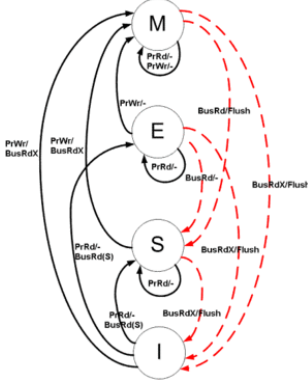
P1
lock(lock) {
  while(1) {
    old = ll(lock);
    if(old == 0)
      if(sc(lock, 1))
        return;
  }
}
    
```

```

P2
lock(lock) {
  while(1) {
    old = ll(lock);
    if(old == 0)
      if(sc(lock, 1))
        return;
  }
}
    
```



LLSC Lock Action Zone



```

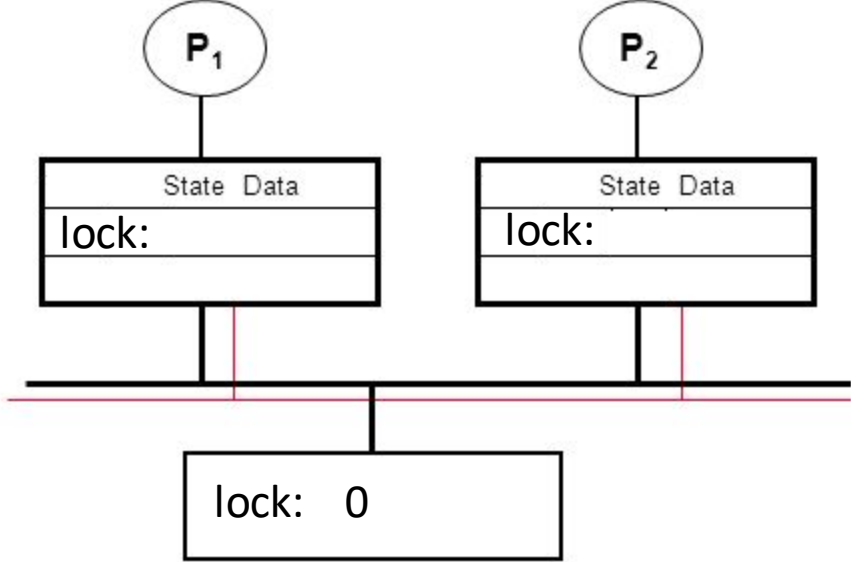
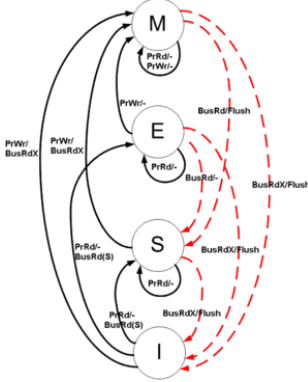
P1
lock(lock) {
  while(1) {
    old = ll(lock);
    if(old == 0)
      if(sc(lock, 1))
        return;
  }
}
    
```

```

P2
lock(lock) {
  while(1) {
    old = ll(lock);
    if(old == 0)
      if(sc(lock, 1))
        return;
  }
}
    
```



LLSC Lock Action Zone II



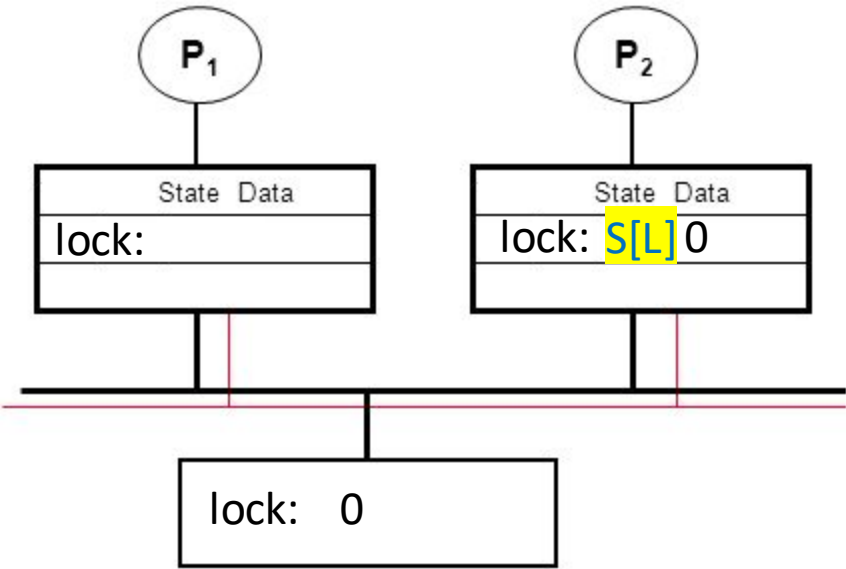
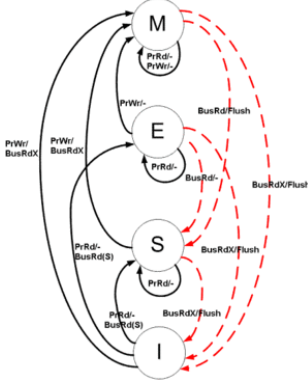
```

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



```

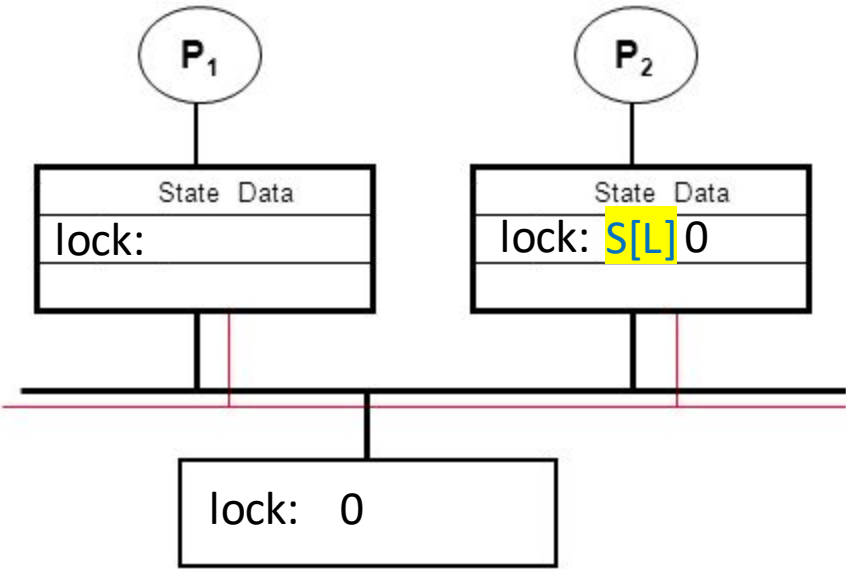
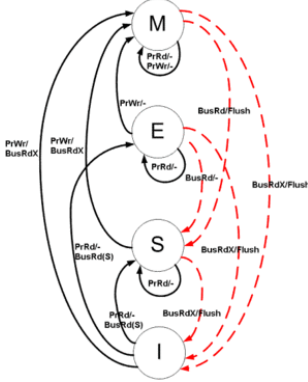
P1
lock(lock) {
  while(1) {
    old = ll(lock);
    if(old == 0)
      if(sc(lock, 1))
        return;
  }
}
    
```



```

P2
lock(lock) {
  while(1) {
    old = ll(lock);
    if(old == 0)
      if(sc(lock, 1))
        return;
  }
}
    
```

LLSC Lock Action Zone II



```

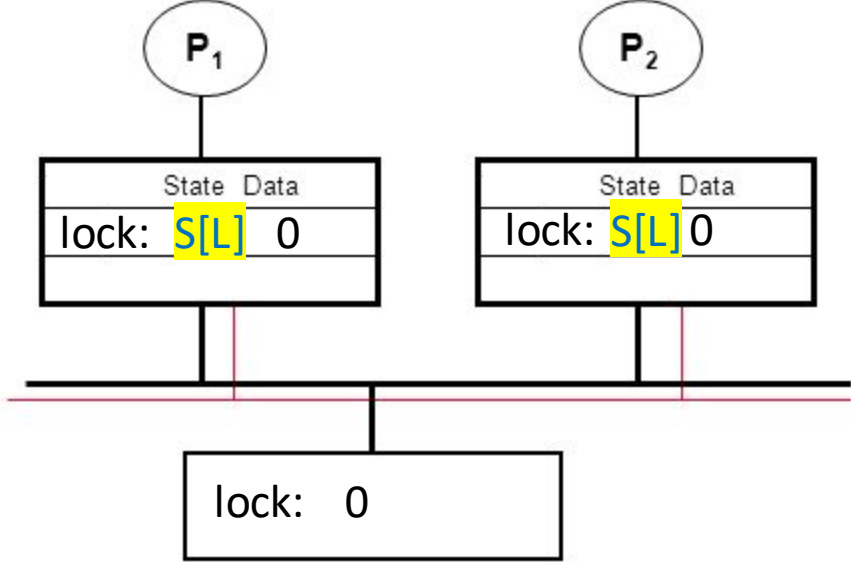
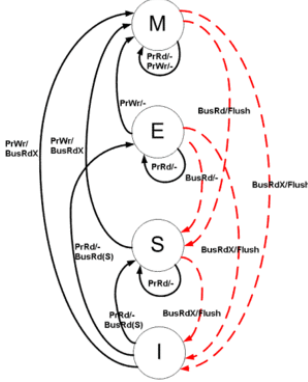
P1
lock(lock) {
  while(1) {
    old = ll(lock);
    if(old == 0)
      if(sc(lock, 1))
        return;
  }
}
    
```



```

P2
lock(lock) {
  while(1) {
    old = ll(lock);
    if(old == 0)
      if(sc(lock, 1))
        return;
  }
}
    
```

LLSC Lock Action Zone II



```

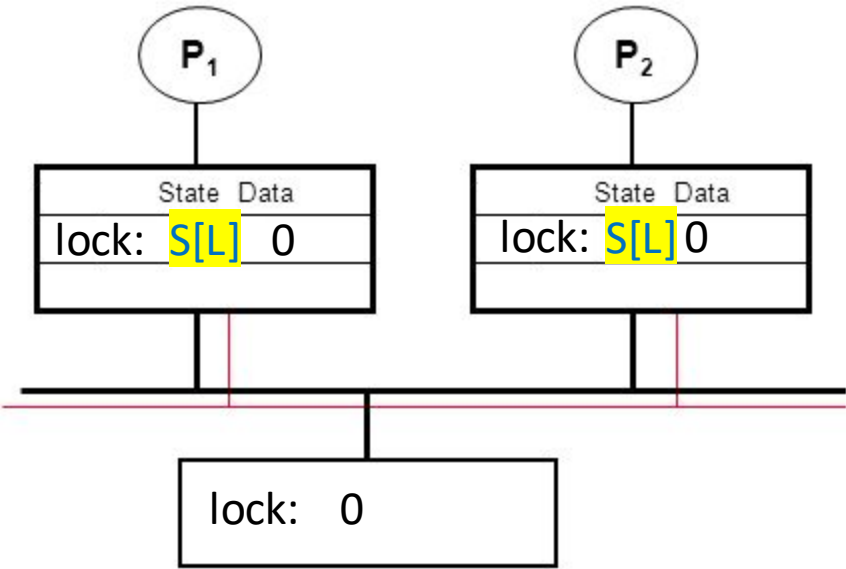
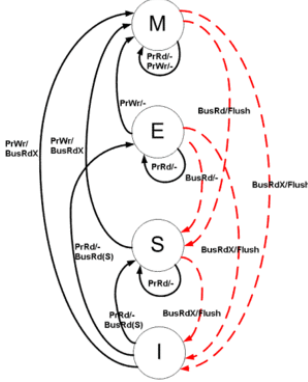
P1
lock(lock) {
  while(1) {
    old = ll(lock);
    if(old == 0)
      if(sc(lock, 1))
        return;
  }
}
    
```



```

P2
lock(lock) {
  while(1) {
    old = ll(lock);
    if(old == 0)
      if(sc(lock, 1))
        return;
  }
}
    
```

LLSC Lock Action Zone II



```

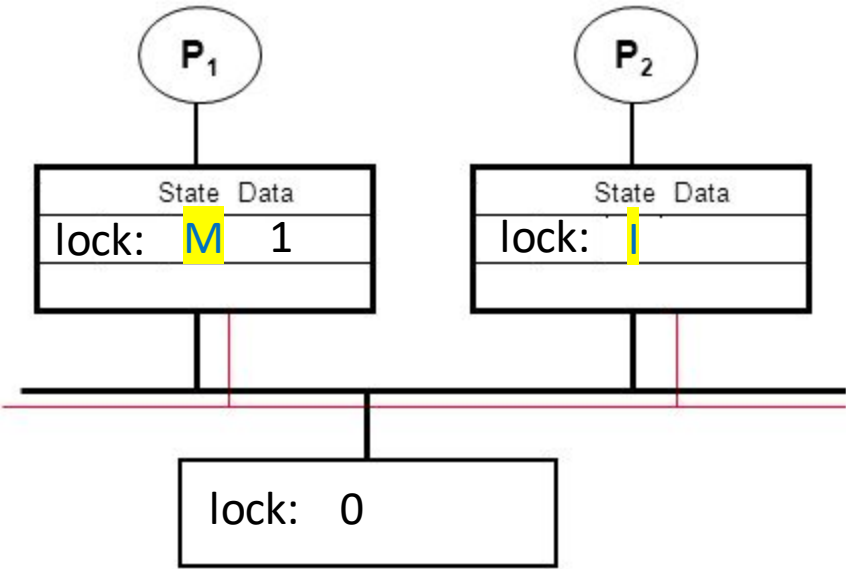
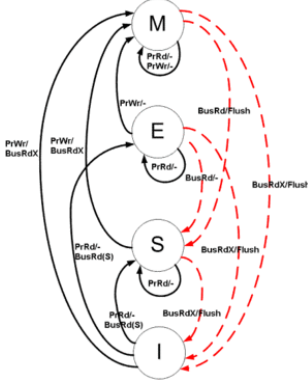
P1
lock(lock) {
  while(1) {
    old = ll(lock);
    if(old == 0)
      if(sc(lock, 1))
        return;
  }
}
    
```



```

P2
lock(lock) {
  while(1) {
    old = ll(lock);
    if(old == 0)
      if(sc(lock, 1))
        return;
  }
}
    
```


LLSC Lock Action Zone II



```

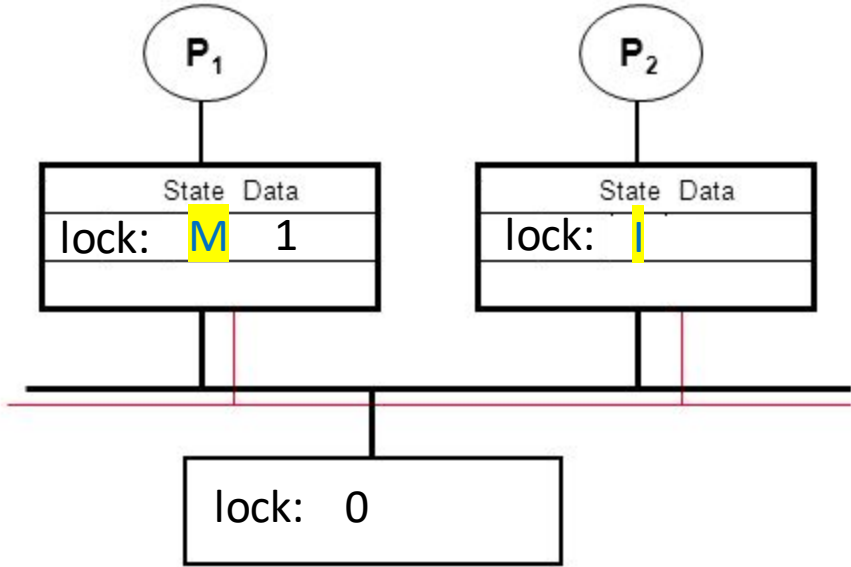
P1
lock(lock) {
  while(1) {
    old = ll(lock);
    if(old == 0)
      if(sc(lock, 1))
        return;
  }
}
  
```



```

P2
lock(lock) {
  while(1) {
    old = ll(lock);
    if(old == 0)
      if(sc(lock, 1))
        return;
  }
}
  
```

LLSC Lock Action Zone II

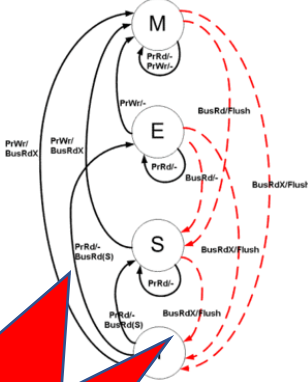


```

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;
  }
}
    
```



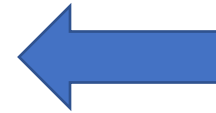
Implementing Locks with Test&set

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

Implementing Locks with Test&set

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

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



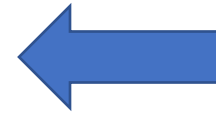
(test & set ~ CAS ~ LLSC)

Implementing Locks with Test&set

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

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

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



(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  
}
```

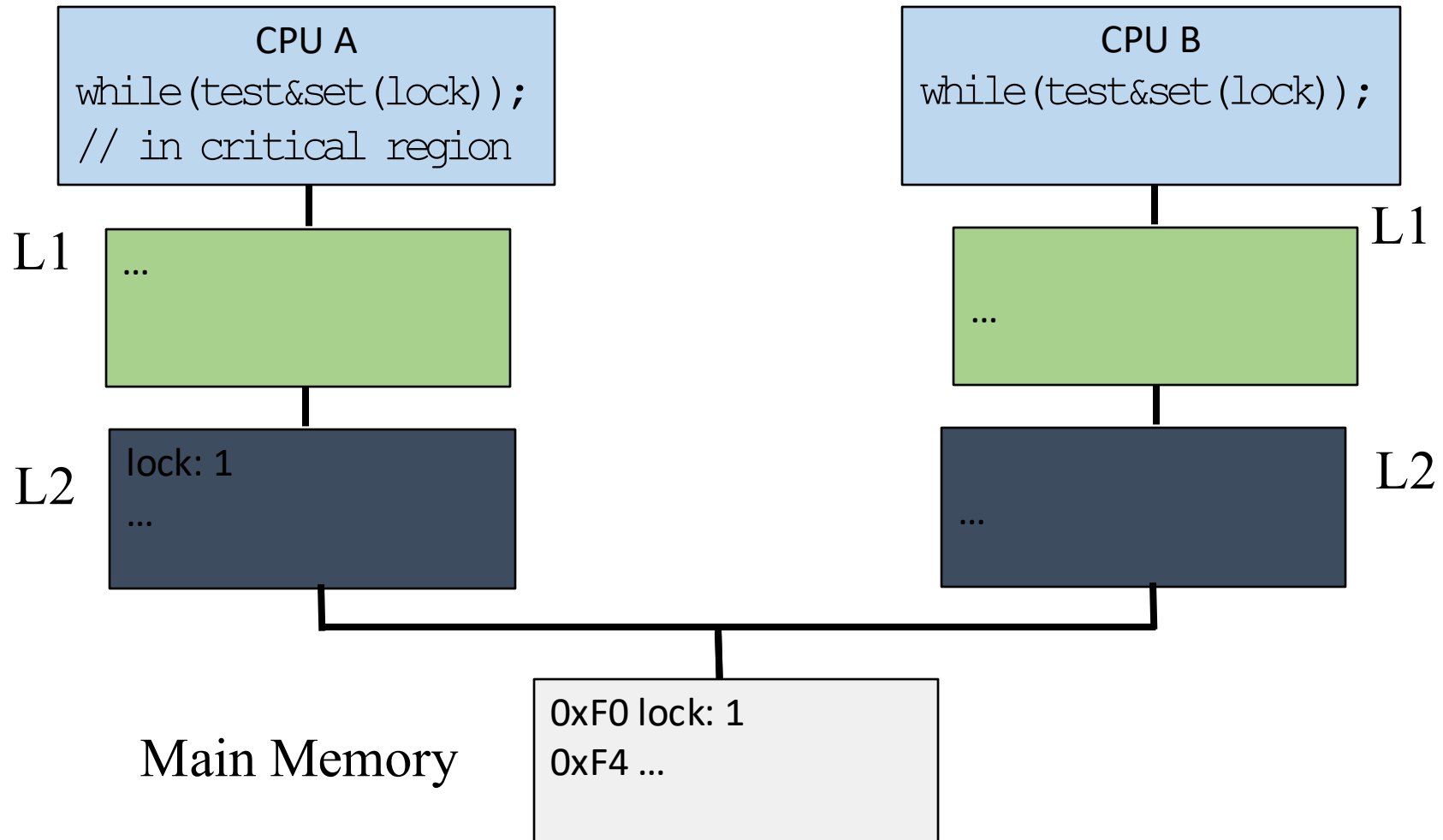
(test & set ~ CAS ~ LLSC)



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

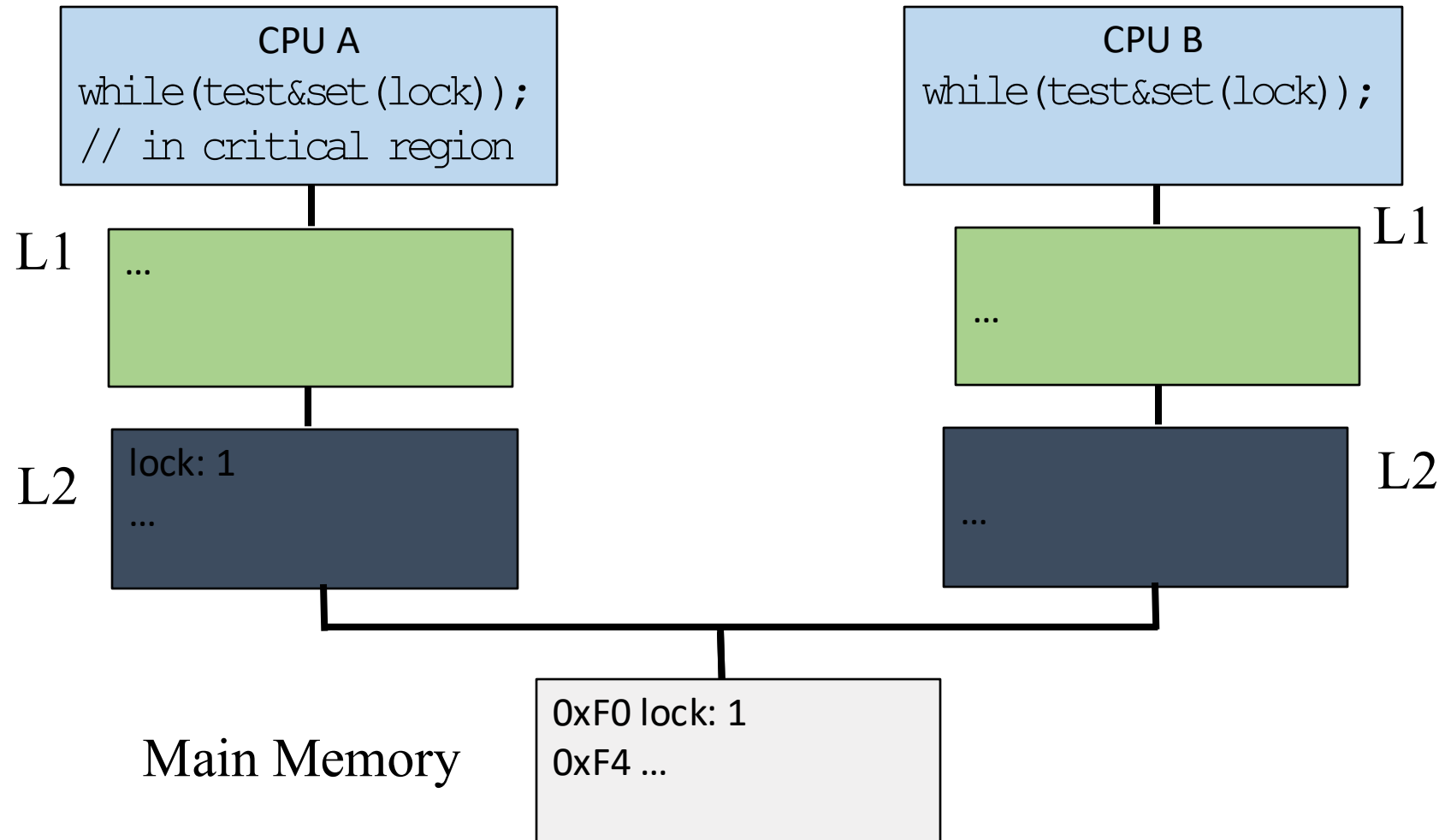
- ◆ What is the problem with this?
 - A. CPU usage B. Memory usage C. Lock::Acquire() latency
 - D. Memory bus usage E. Does not work

Test & Set with Memory Hierarchies



Test & Set with Memory Hierarchies

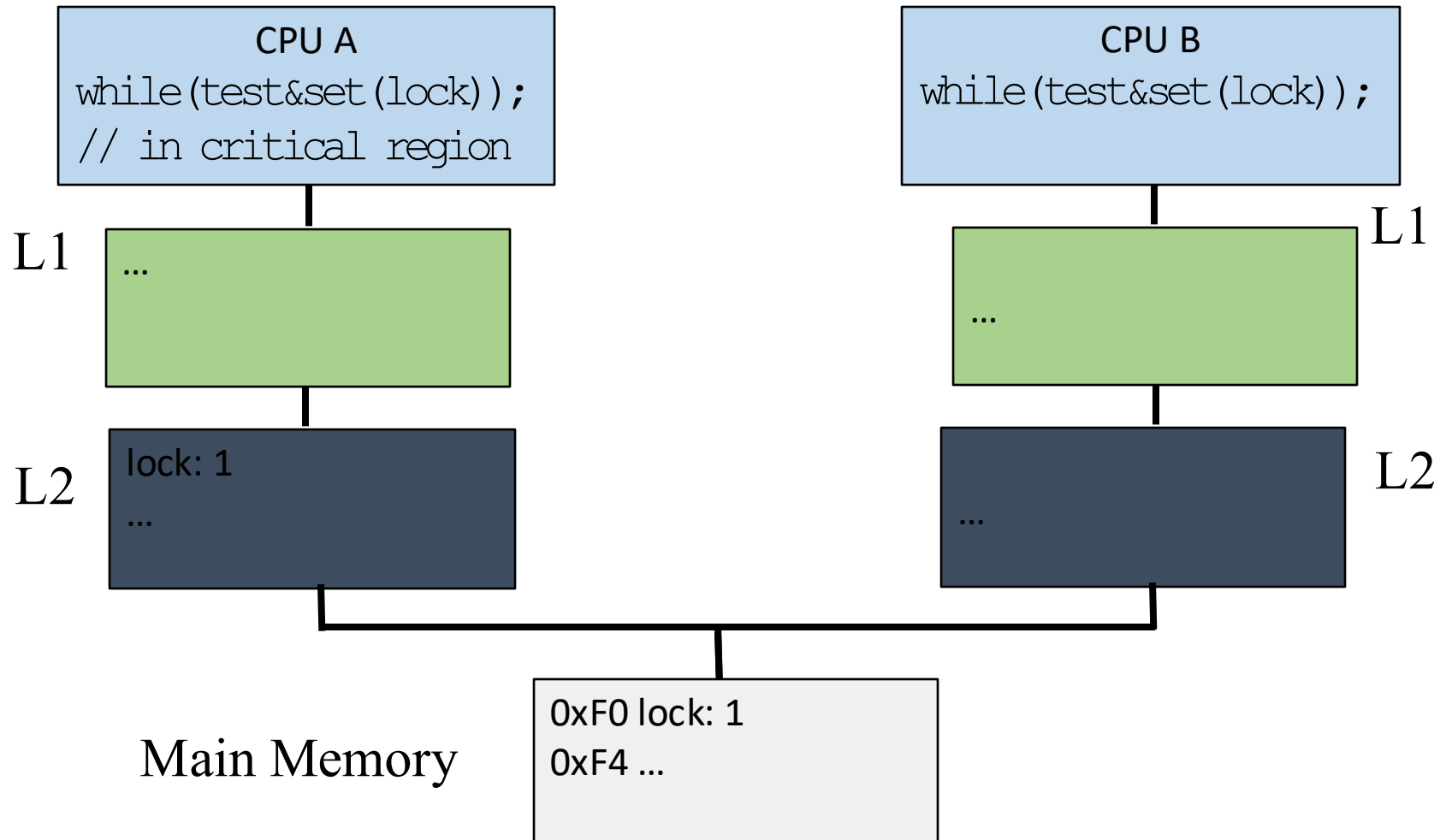
- Initially, lock held by CPU C



Test & Set with Memory Hierarchies

```
CPU C  
// in critical region
```

- Initially, lock held by CPU C

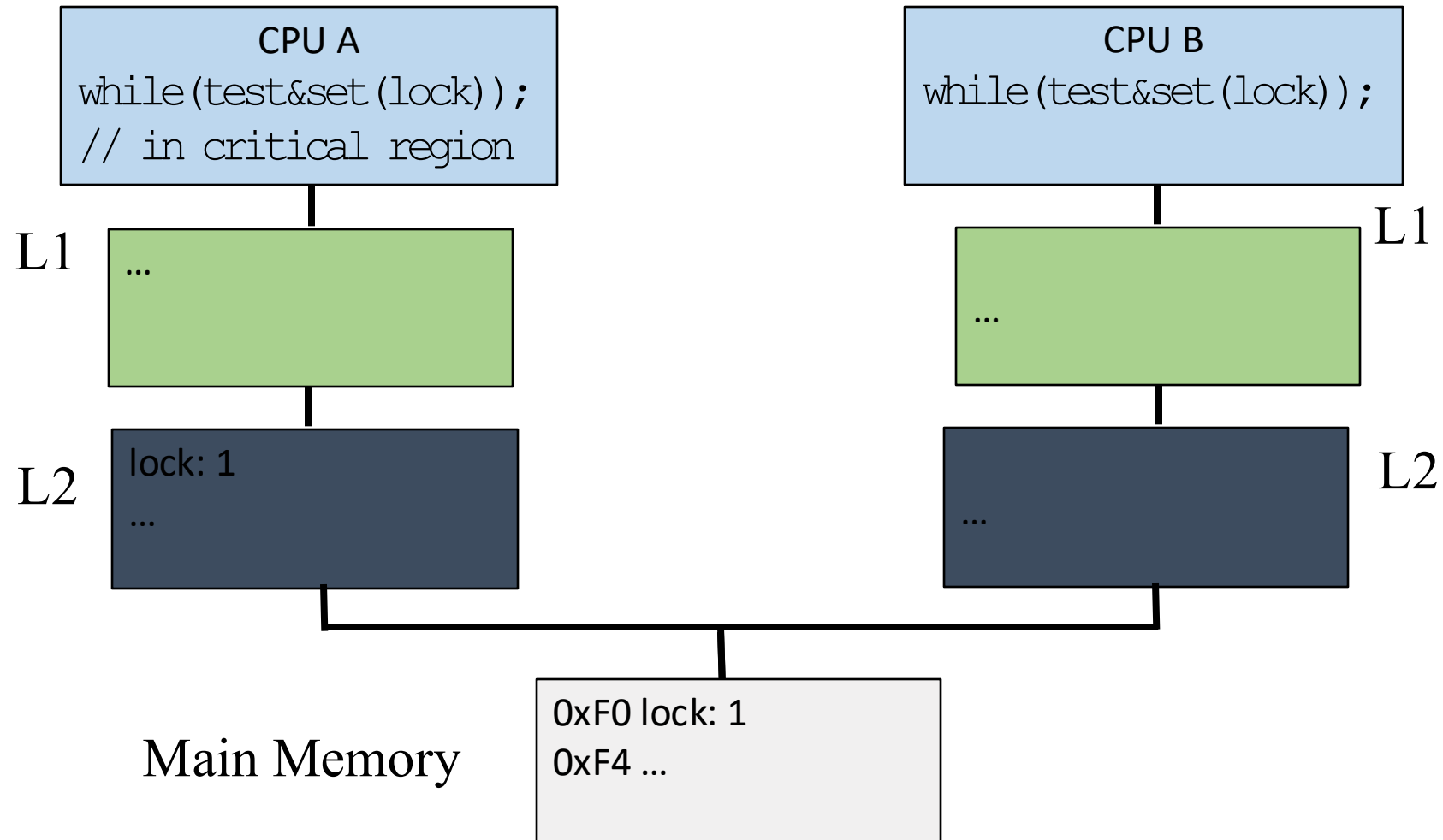


Test & Set with Memory Hierarchies

CPU C
// in critical region



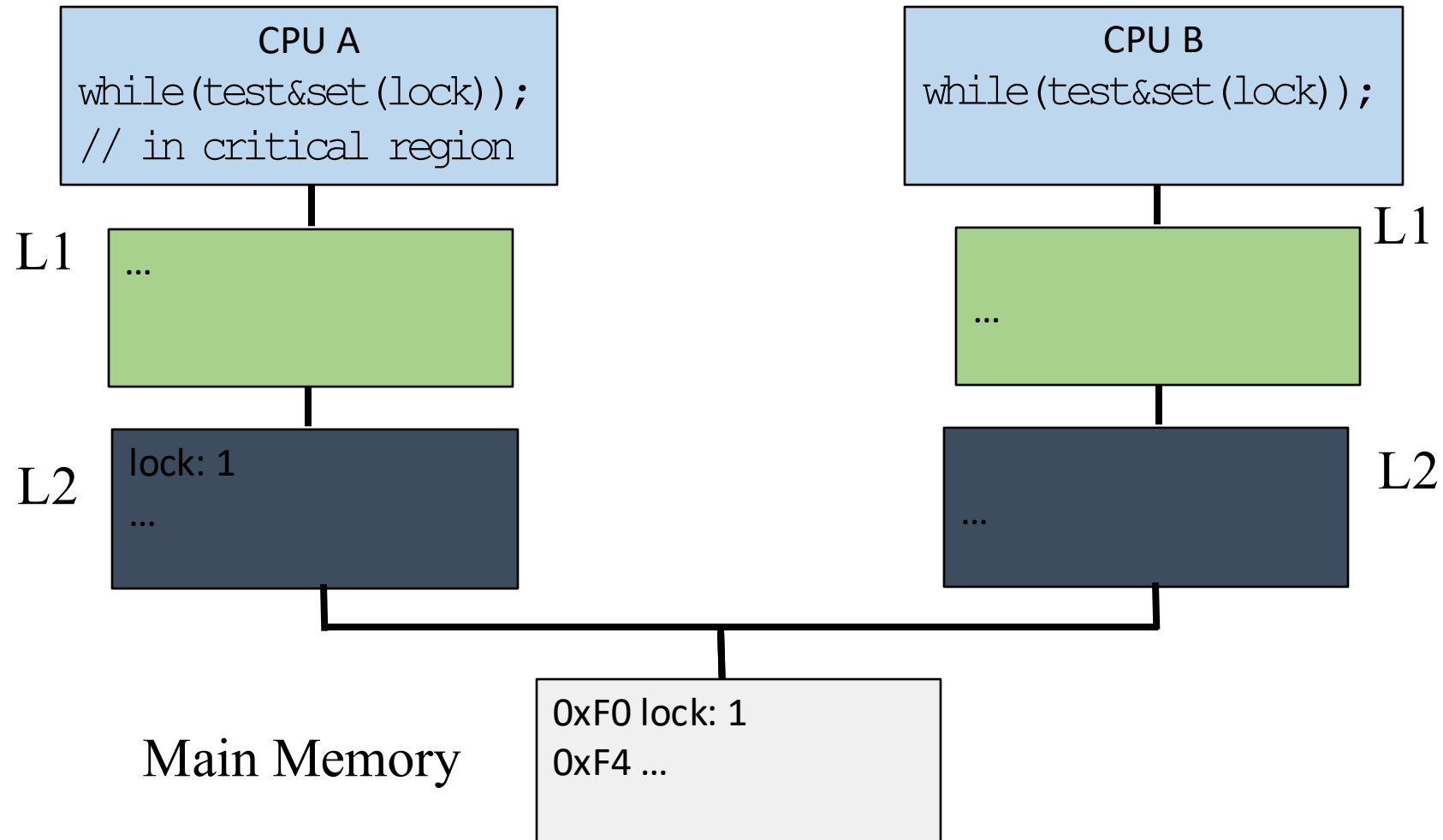
- Initially, lock held by CPU C



Test & Set with Memory Hierarchies

- Initially, lock held by CPU C
- CPU A, B busy-waiting

CPU C
// in critical region

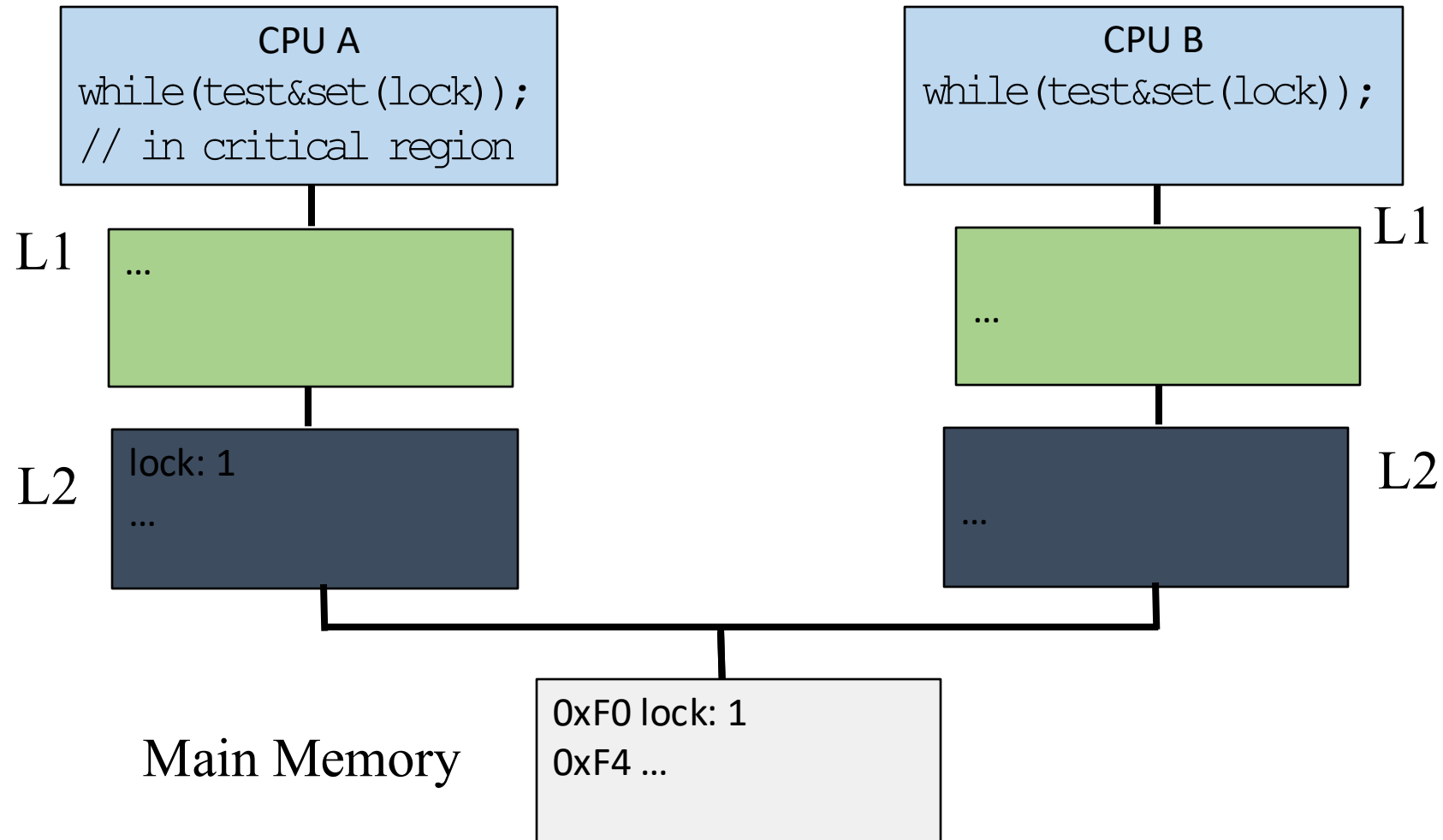


Test & Set with Memory Hierarchies

CPU C
// in critical region




- Initially, lock held by CPU C
- CPU A, B busy-waiting
- What happens to lock variable's cache line when different CPUs contend?

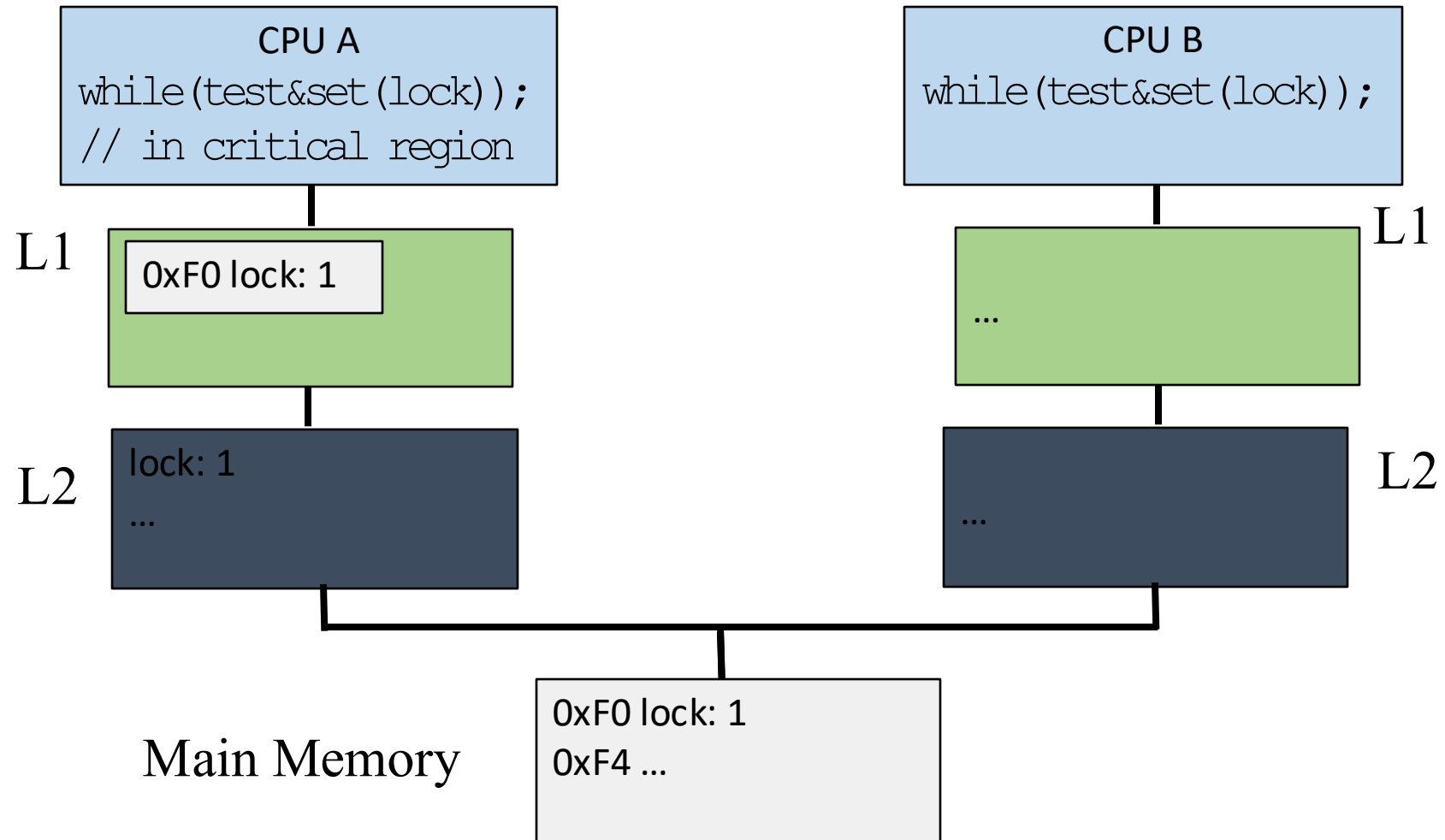


Test & Set with Memory Hierarchies

CPU C
// in critical region



- Initially, lock held by CPU C
- CPU A, B busy-waiting
- What happens to lock variable's cache line when different CPUs contend?

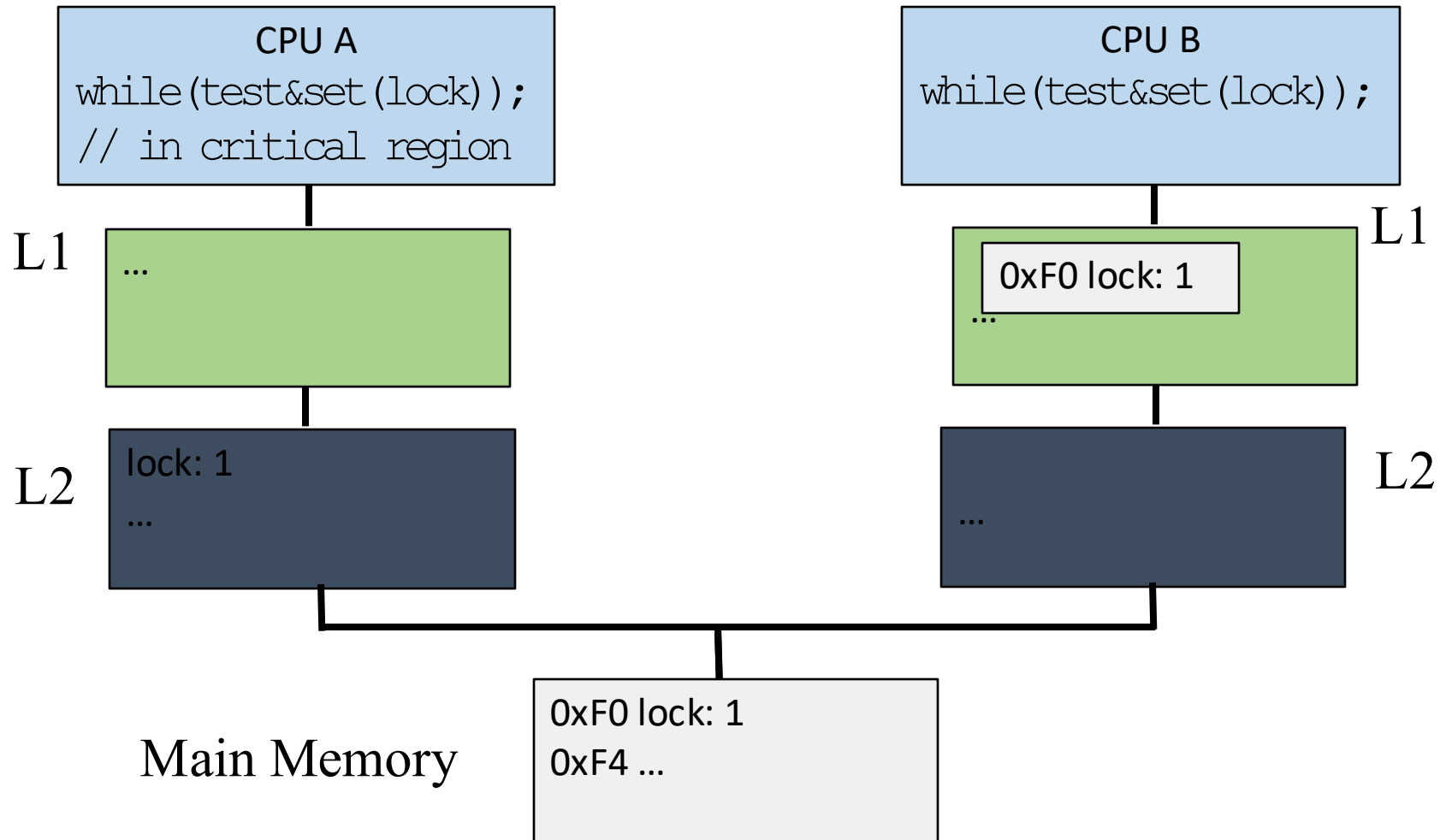


Test & Set with Memory Hierarchies

CPU C
// in critical region




- Initially, lock held by CPU C
- CPU A, B busy-waiting
- What happens to lock variable's cache line when different CPUs contend?

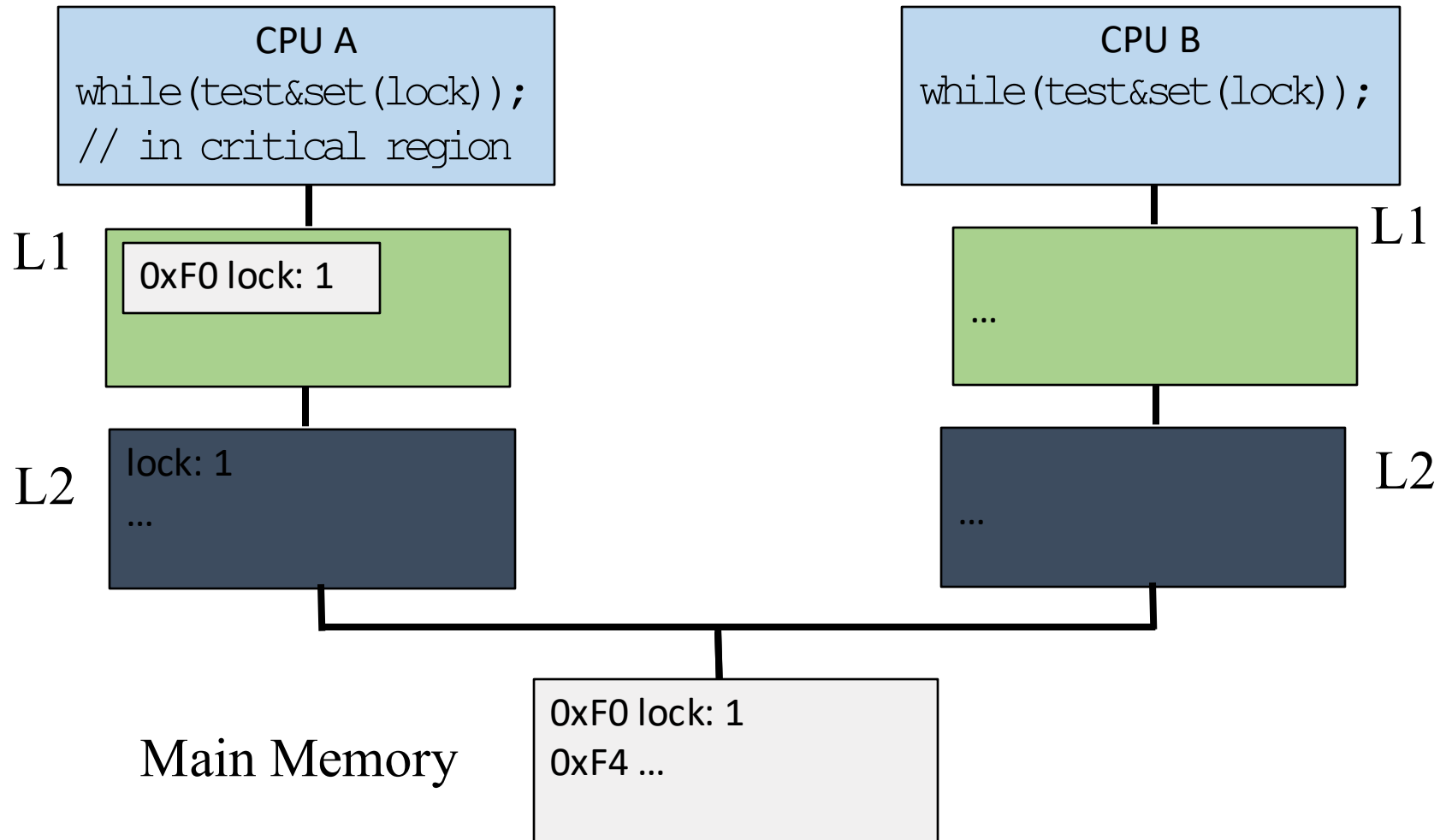


Test & Set with Memory Hierarchies

CPU C
// in critical region




- Initially, lock held by CPU C
- CPU A, B busy-waiting
- What happens to lock variable's cache line when different CPUs contend?

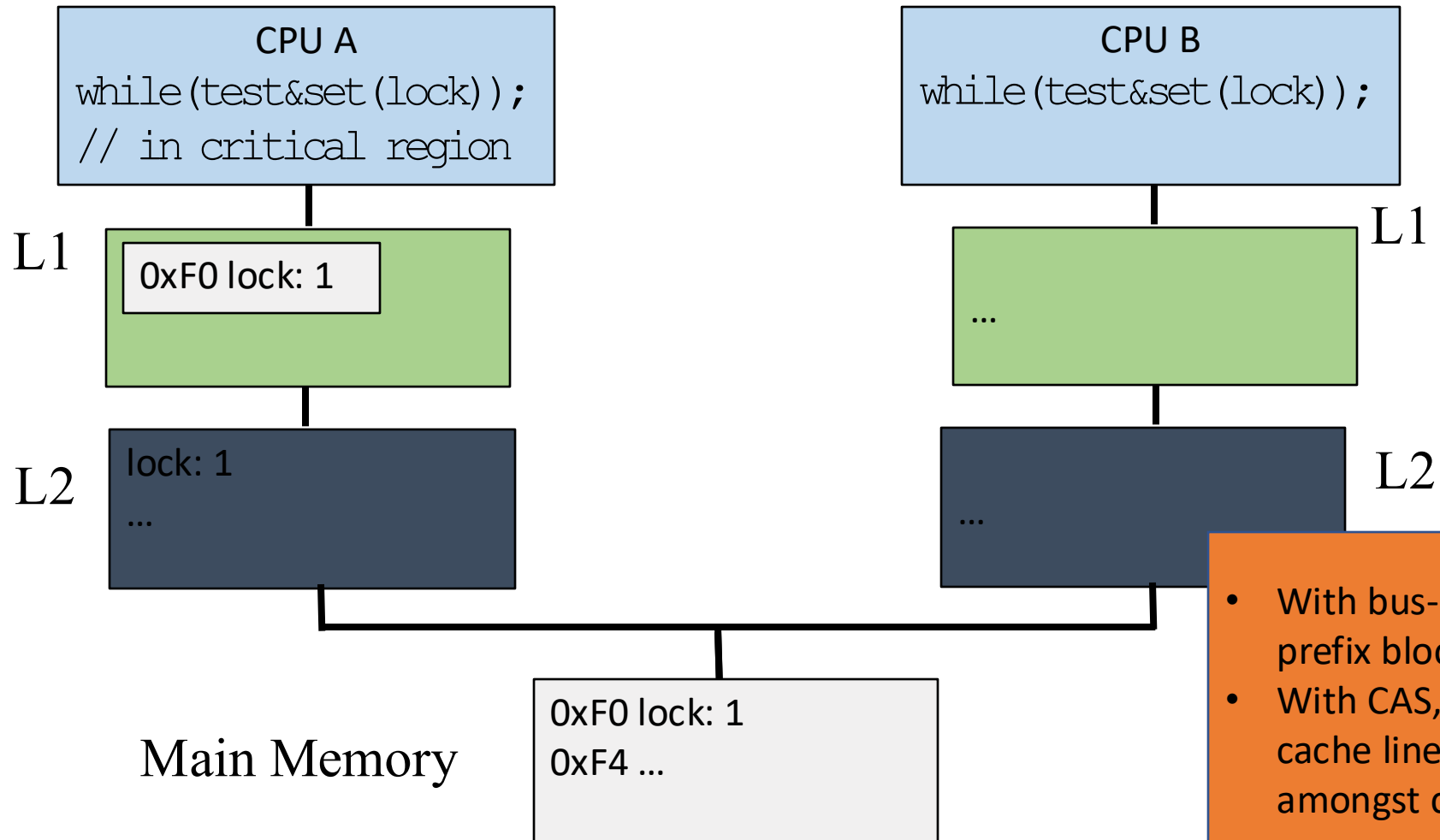


Test & Set with Memory Hierarchies

CPU C
// in critical region



- Initially, lock held by CPU C
- CPU A, B busy-waiting
- What happens to lock variable's cache line when different CPUs 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);  
}
```

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;  
}
```

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

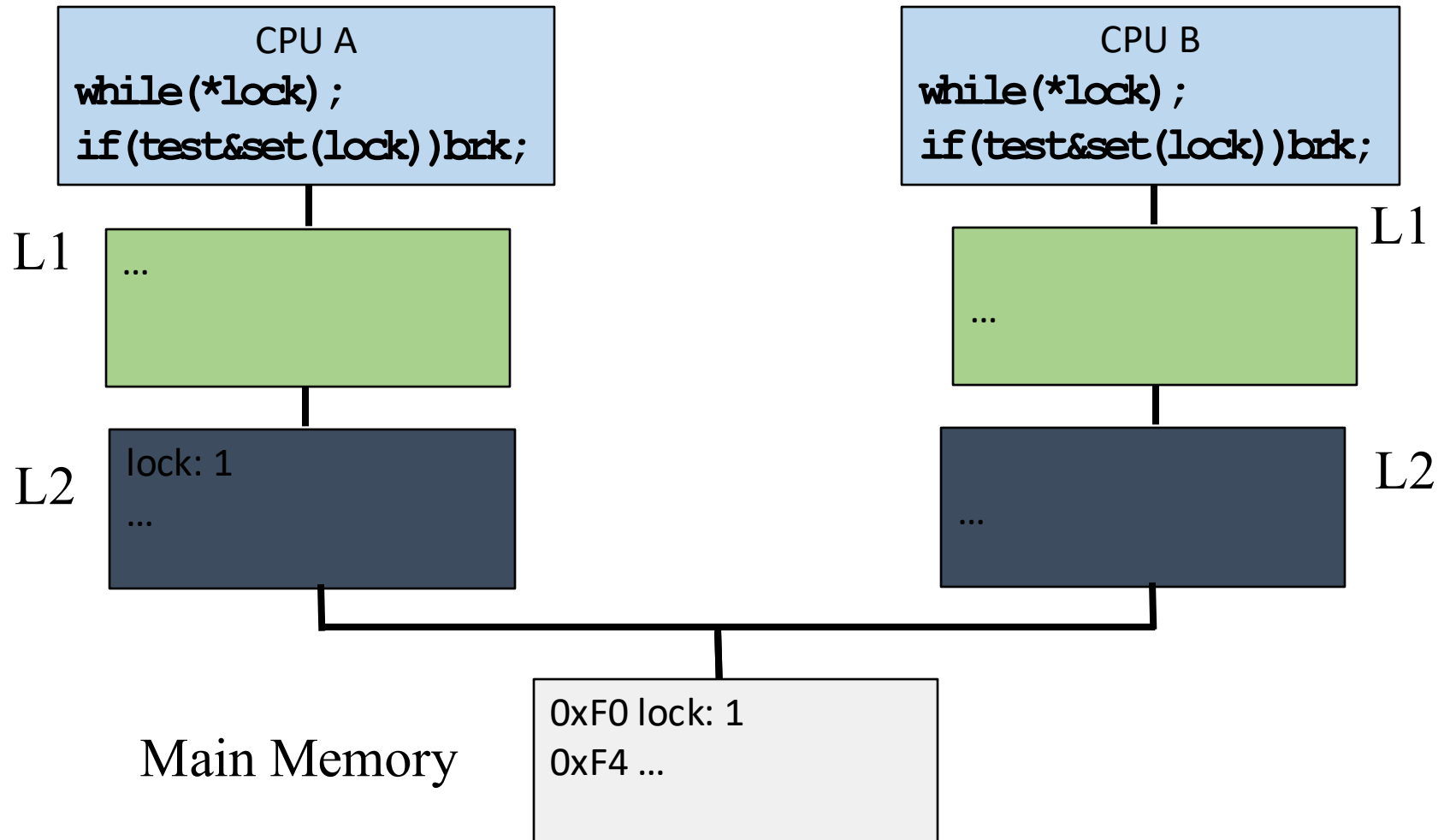
```
Lock::Acquire() {  
while(1) {  
while (*lock == 1) ; // spin just reading  
if (test&set(lock) == 0) break;  
}
```

Busy-wait on cached copy

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

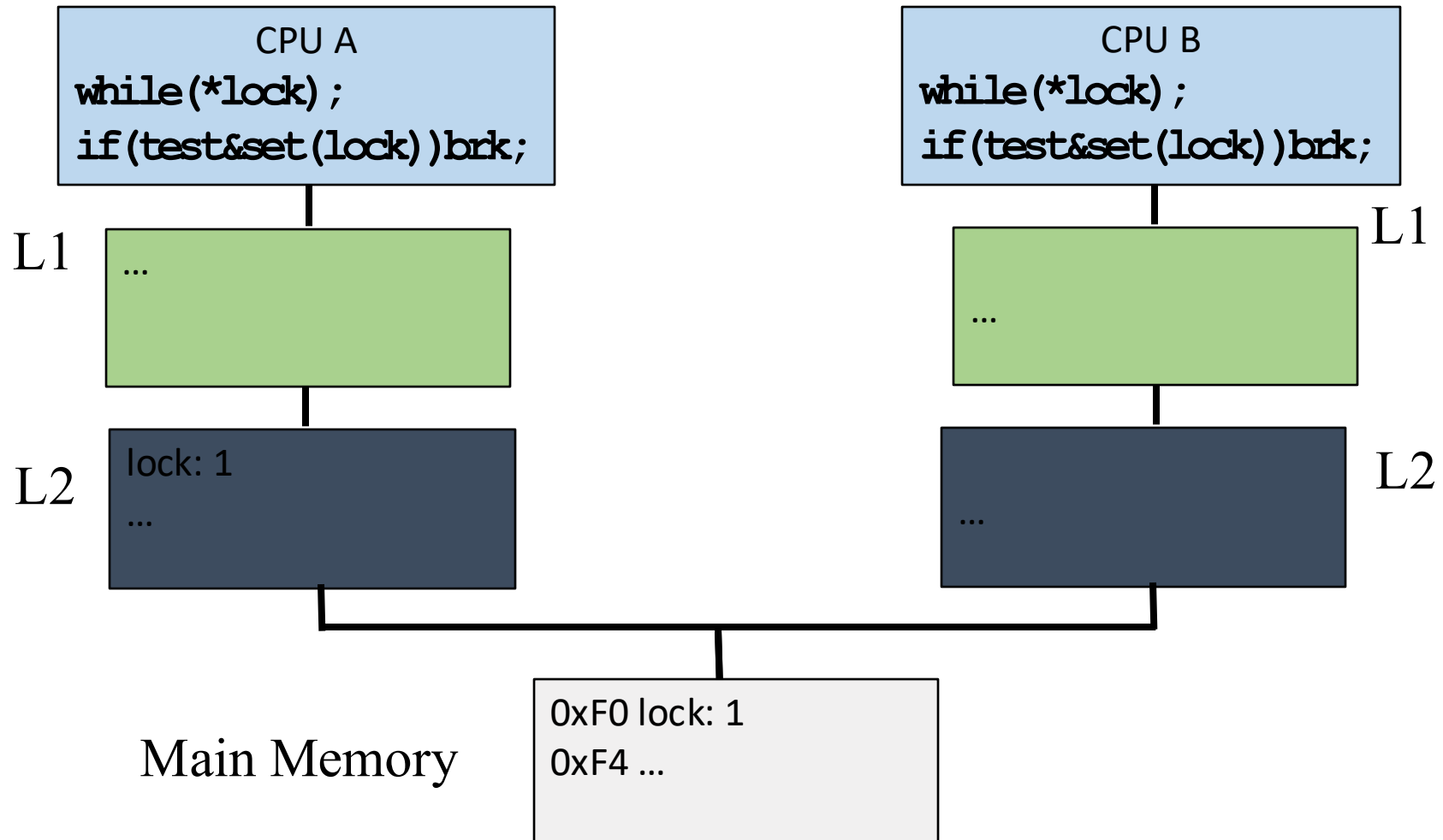
- What is the problem with this?
 - A. CPU usage B. Memory usage C. Lock::Acquire() latency
 - D. Memory bus usage E. Does not work

Test & Test & Set w Memory Hierarchies



Test & Test & Set w Memory Hierarchies

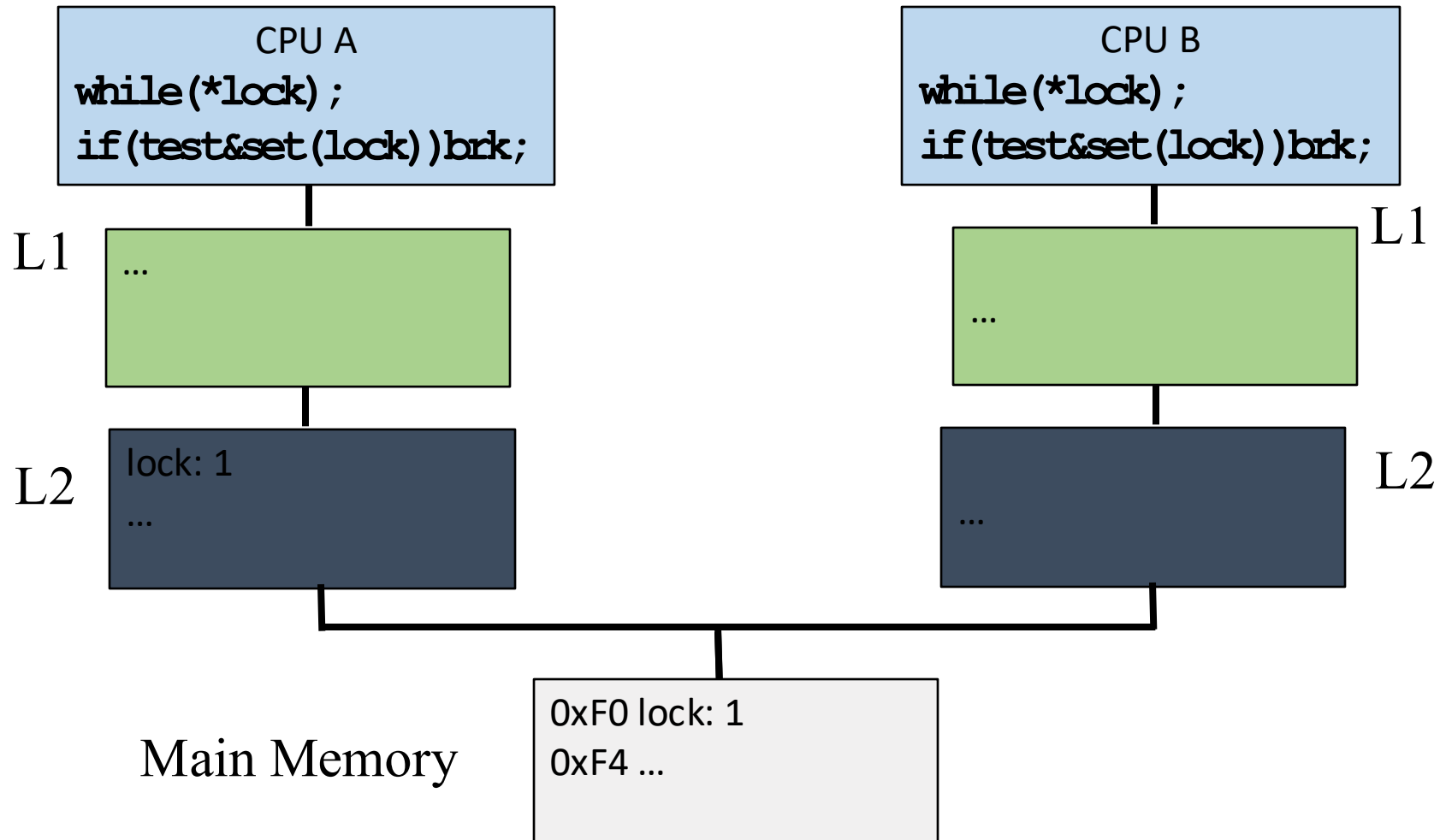
- Initially, lock held by CPU C



Test & Test & Set w Memory Hierarchies

```
CPU C  
// in critical region
```

- Initially, lock held by CPU C

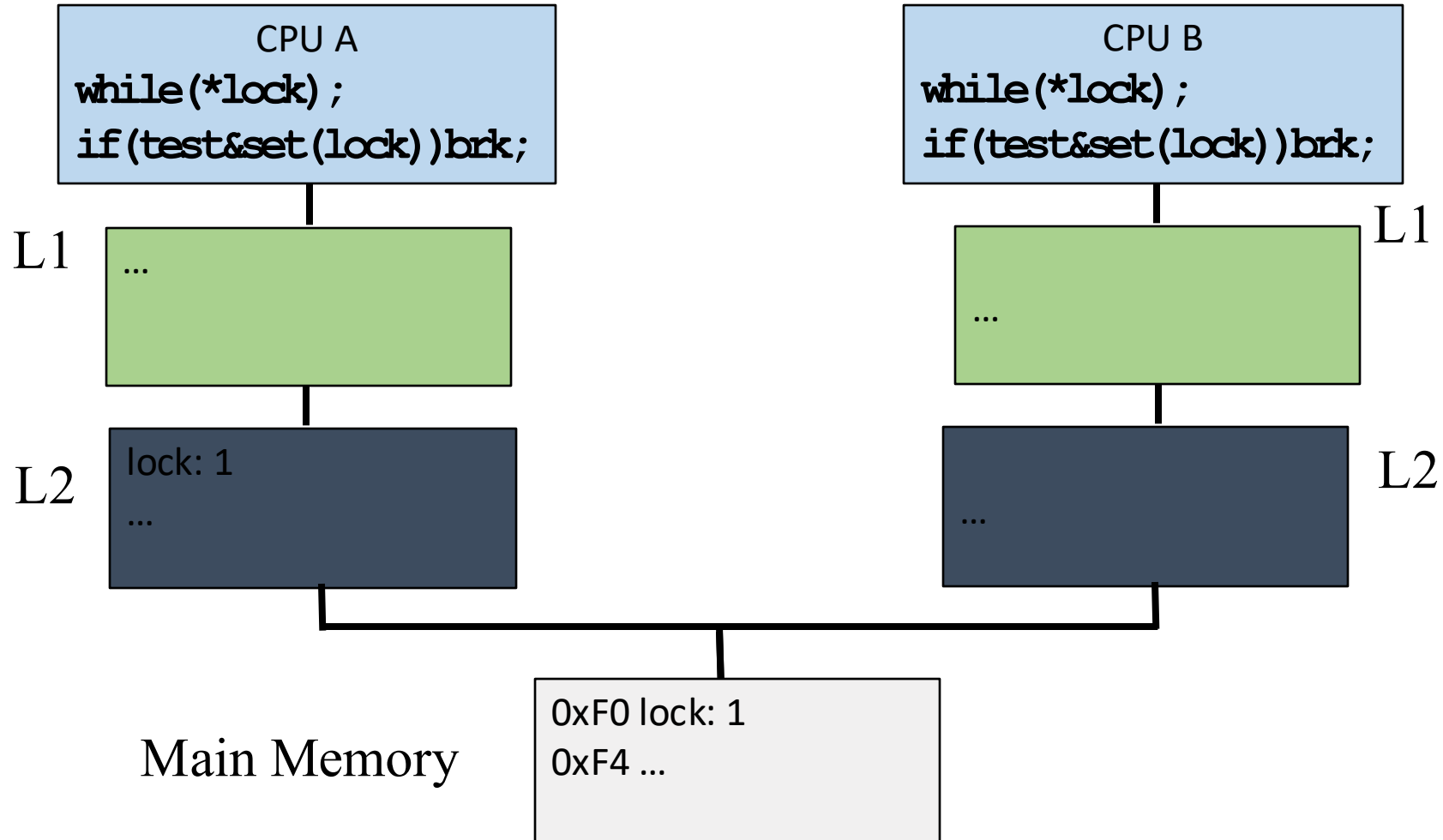


Test & Test & Set w Memory Hierarchies

CPU C
// in critical region



- Initially, lock held by CPU C

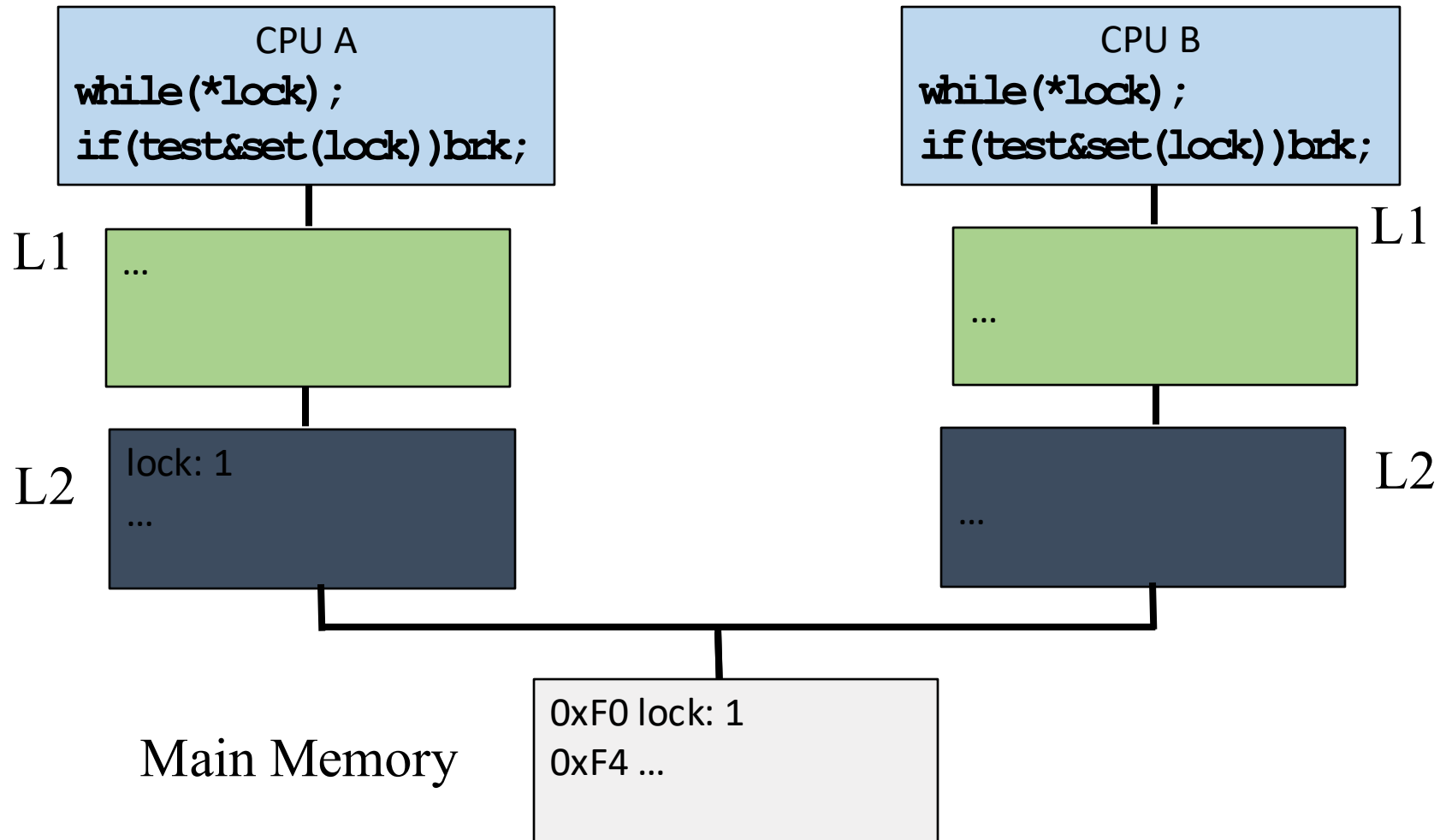


Test & Test & Set w Memory Hierarchies

CPU C
// in critical region



- Initially, lock held by CPU C
- CPU A, B busy-waiting

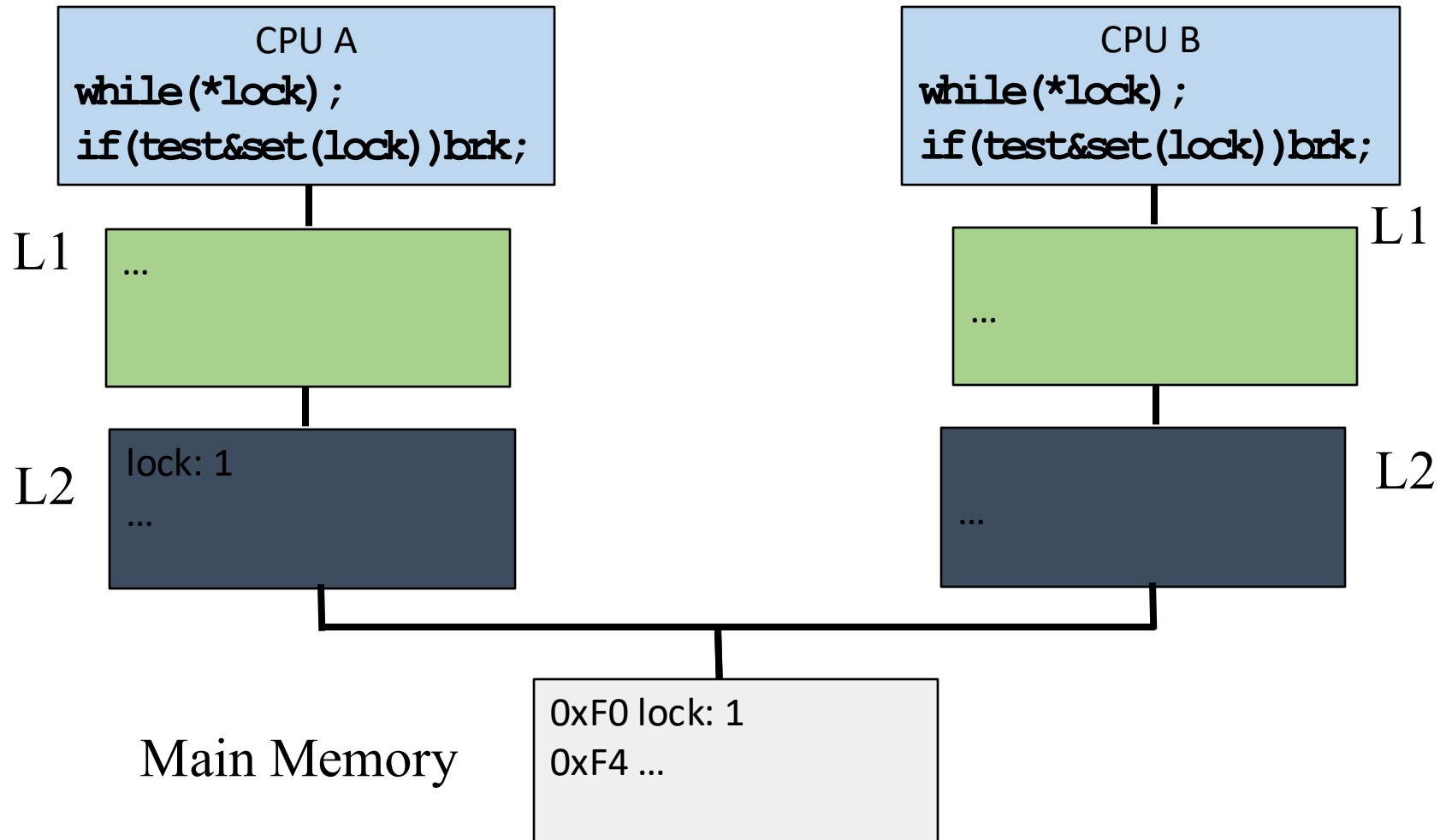


Test & Test & Set w Memory Hierarchies

CPU C
// in critical region



- Initially, lock held by CPU C
- CPU A, B busy-waiting
- Now** what happens to lock variable's cache line when different CPUs contend?

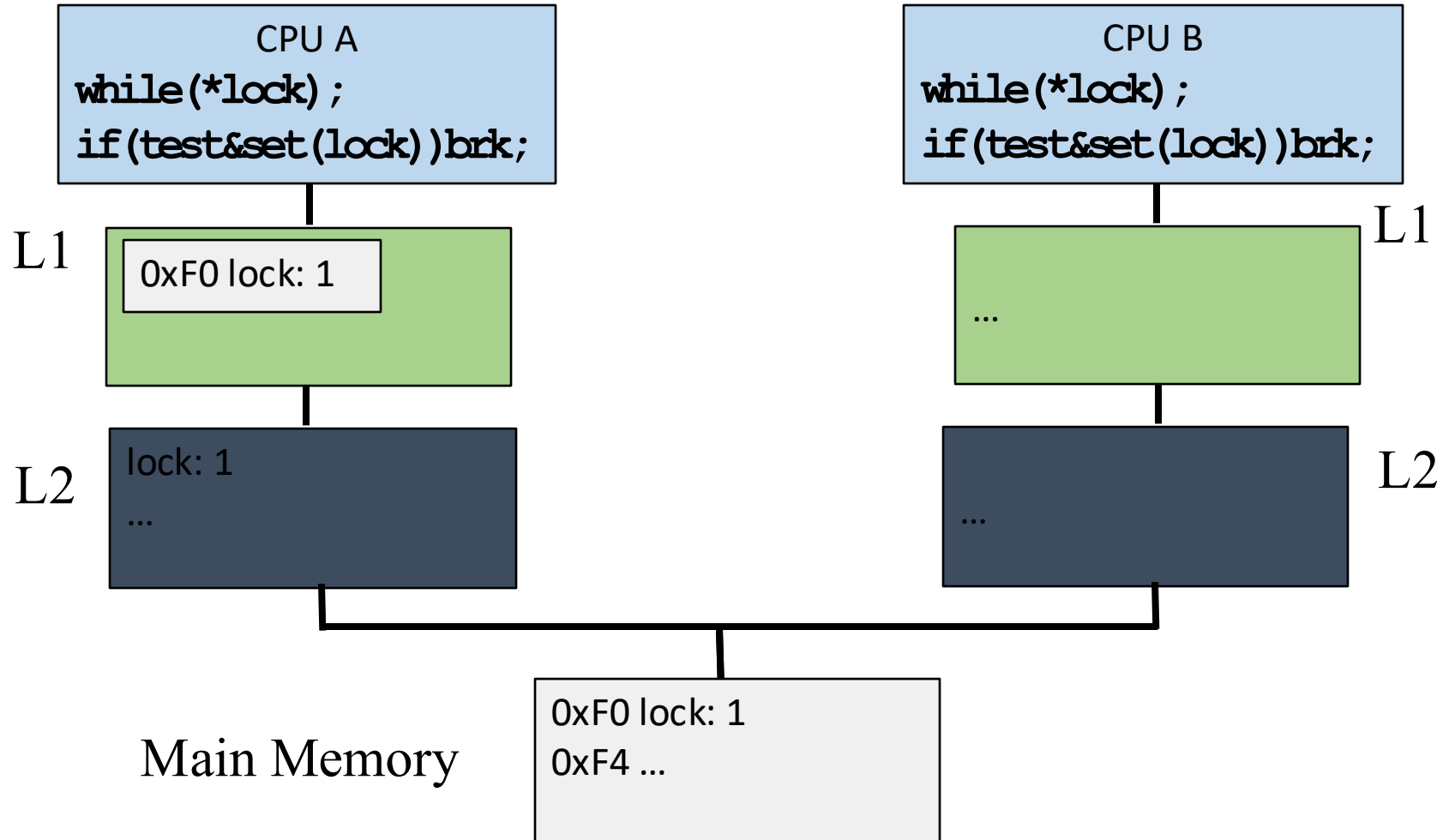


Test & Test & Set w Memory Hierarchies

CPU C
// in critical region



- Initially, lock held by CPU C
- CPU A, B busy-waiting
- Now** what happens to lock variable's cache line when different CPUs contend?

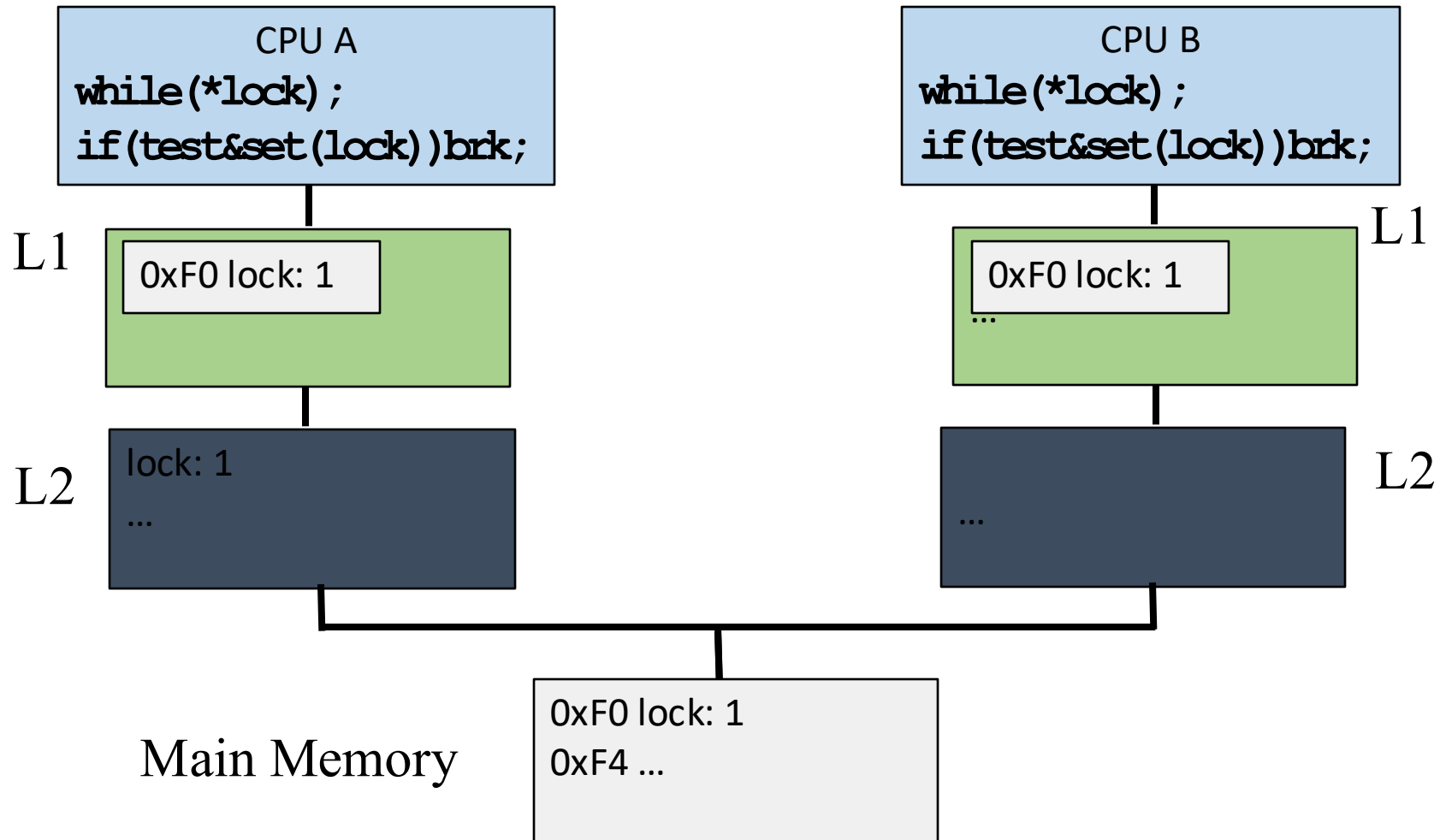


Test & Test & Set w Memory Hierarchies

CPU C
// in critical region

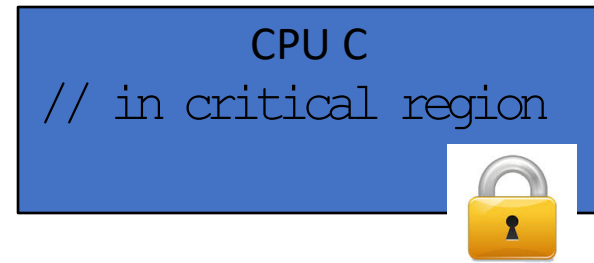


- Initially, lock held by CPU C
- CPU A, B busy-waiting
- Now** what happens to lock variable's cache line when different CPUs contend?

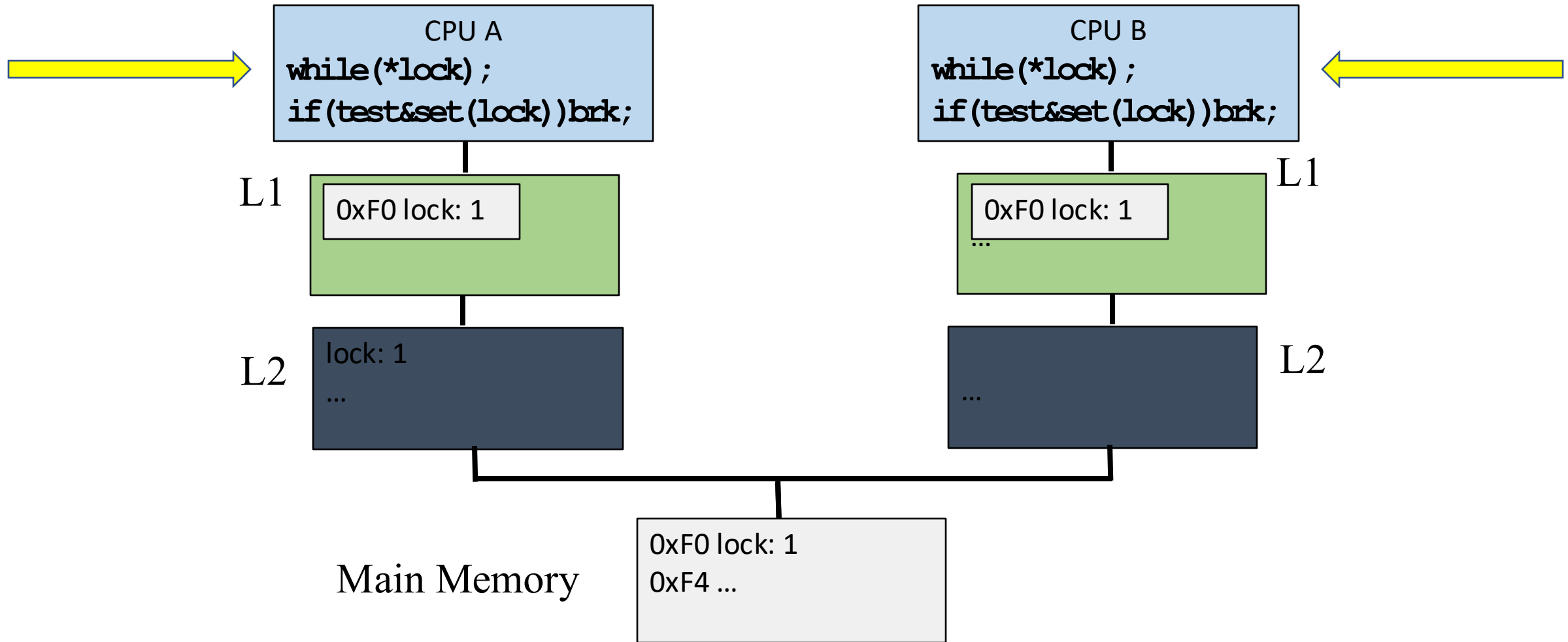


Test & Test & Set w Memory Hierarchies

CPU C
// in critical region




- Initially, lock held by CPU C
- CPU A, B busy-waiting
- **Now** what happens to lock variable's cache line when different CPUs contend?

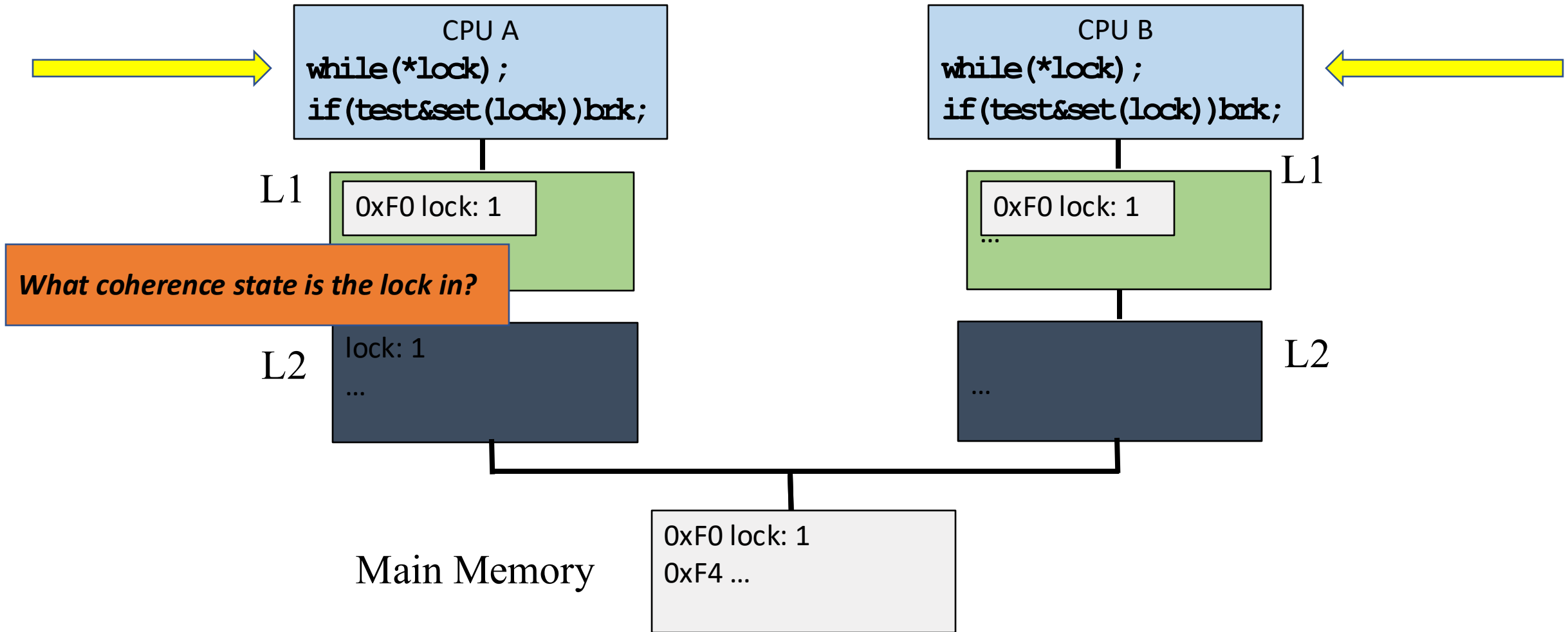


Test & Test & Set w Memory Hierarchies

```
CPU C  
// in critical region
```




- Initially, lock held by CPU C
- CPU A, B busy-waiting
- Now** what happens to lock variable's cache line when different CPUs contend?

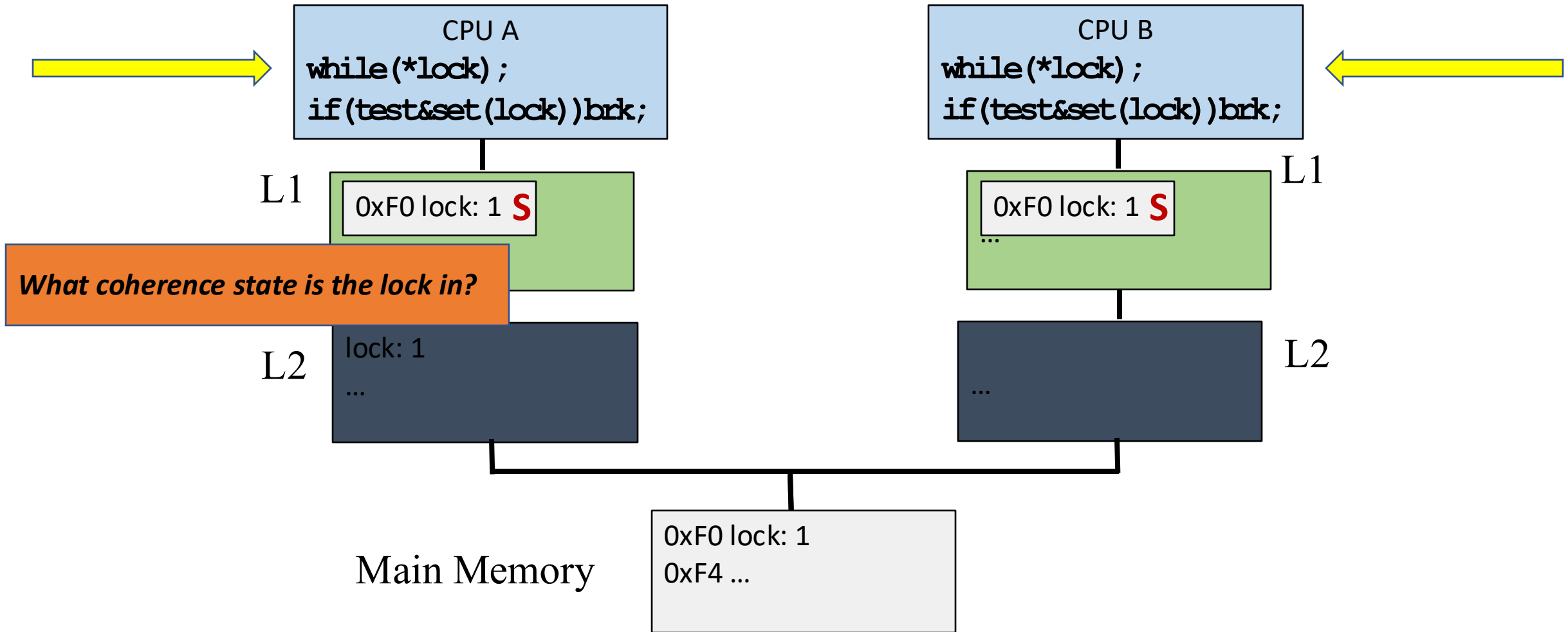


Test & Test & Set w Memory Hierarchies

```
CPU C  
// in critical region
```

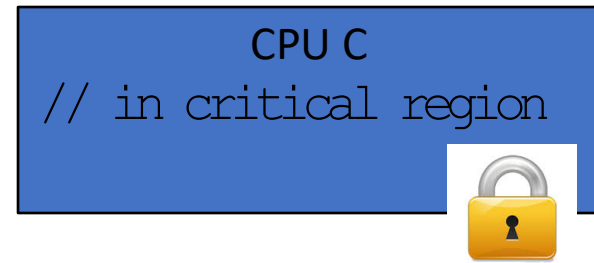


- Initially, lock held by CPU C
- CPU A, B busy-waiting
- Now** what happens to lock variable's cache line when different CPUs contend?

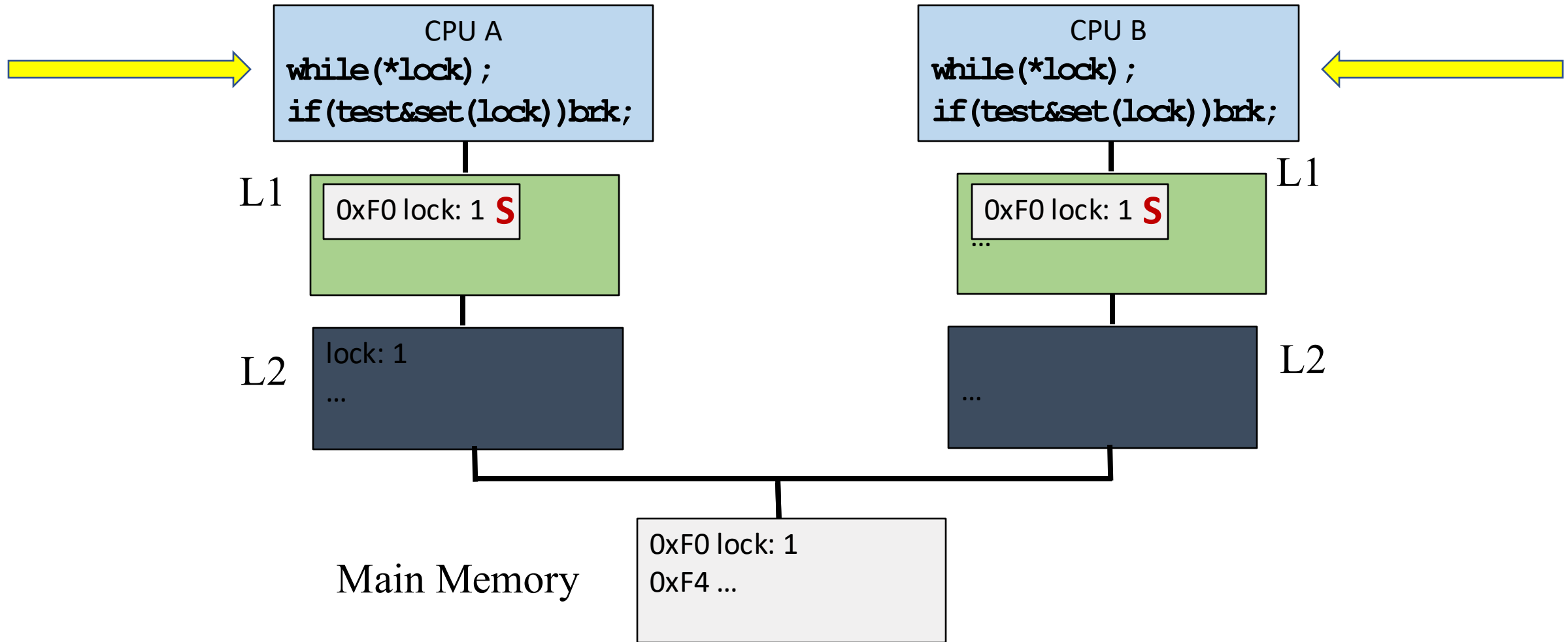


Test & Test & Set w Memory Hierarchies

CPU C
// in critical region




- Initially, lock held by CPU C
- CPU A, B busy-waiting
- **Now** what happens to lock variable's cache line when different CPUs contend?

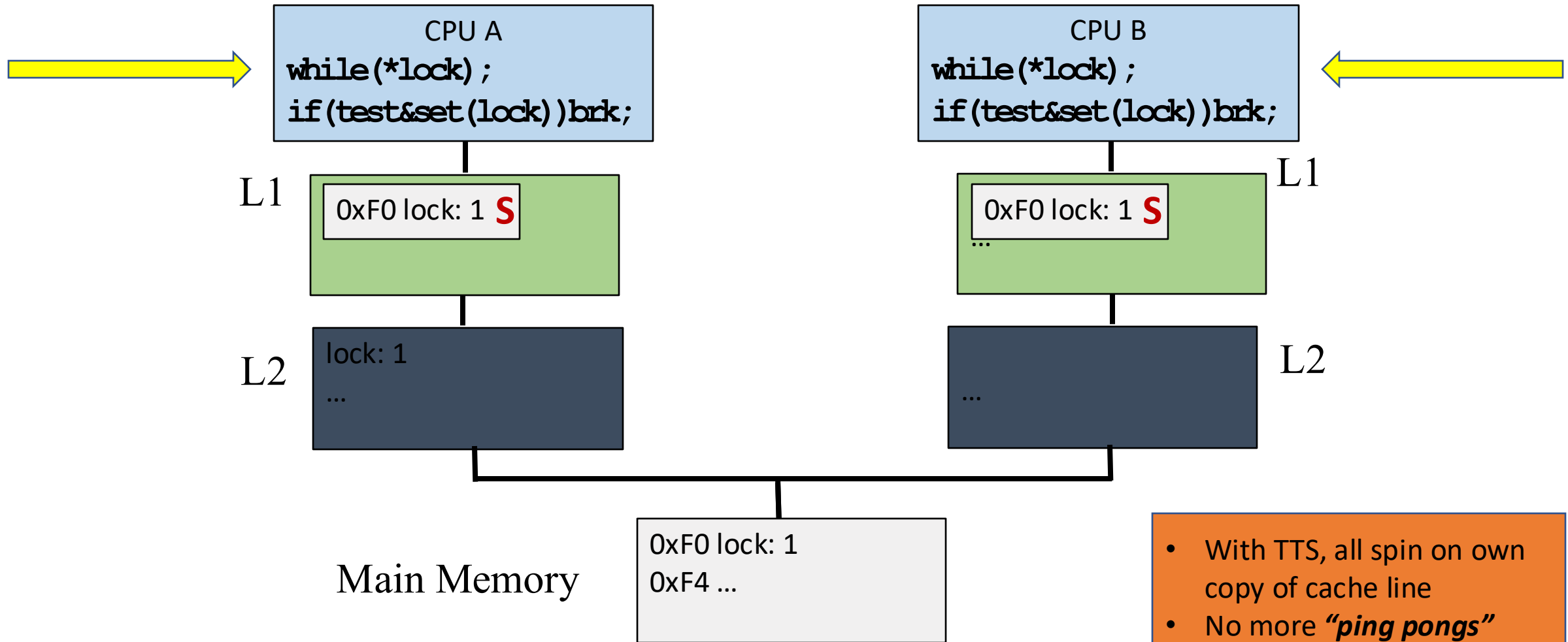


Test & Test & Set w Memory Hierarchies

CPU C
// in critical region

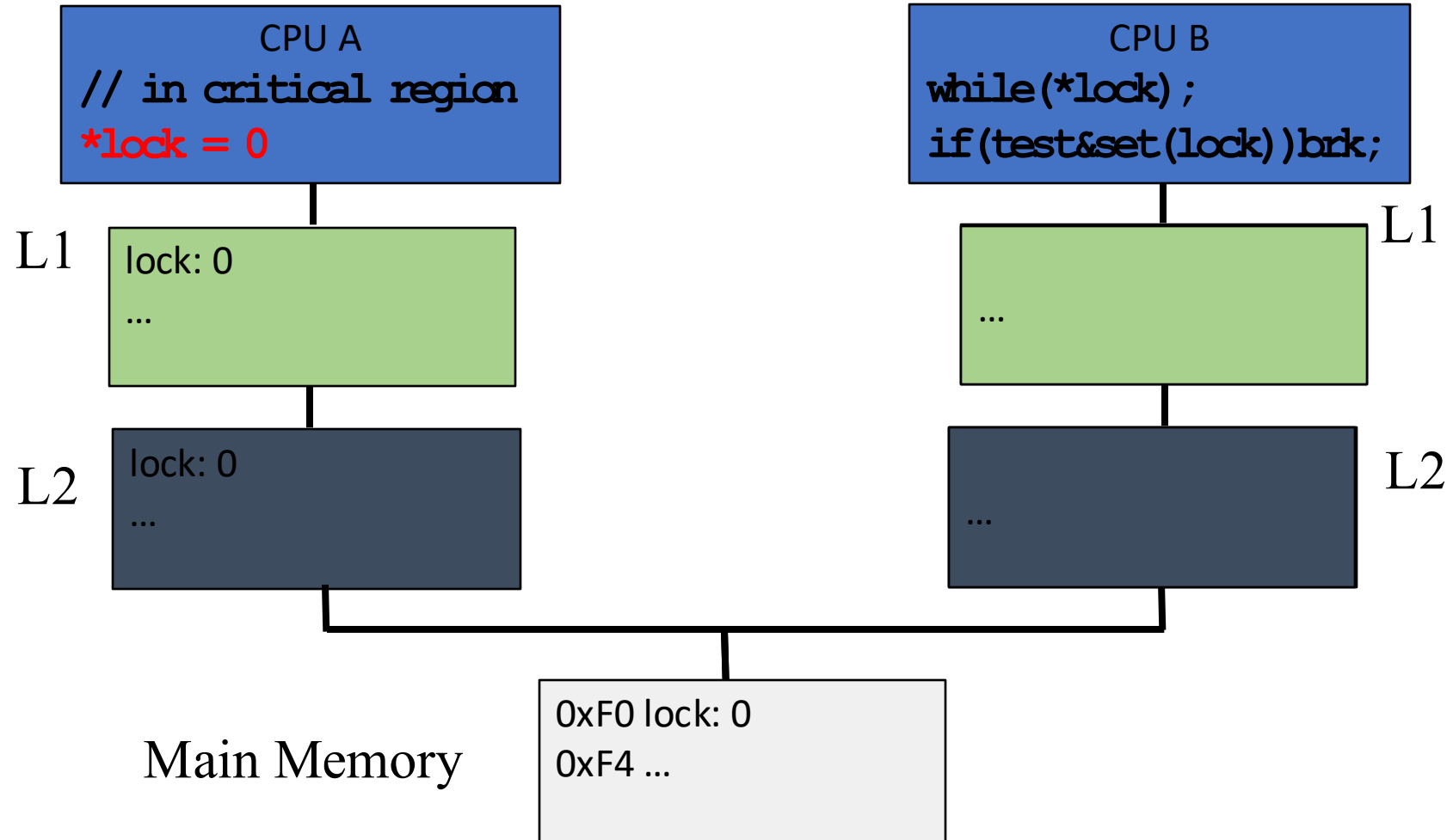


- Initially, lock held by CPU C
- CPU A, B busy-waiting
- Now** what happens to lock variable's cache line when different CPUs contend?



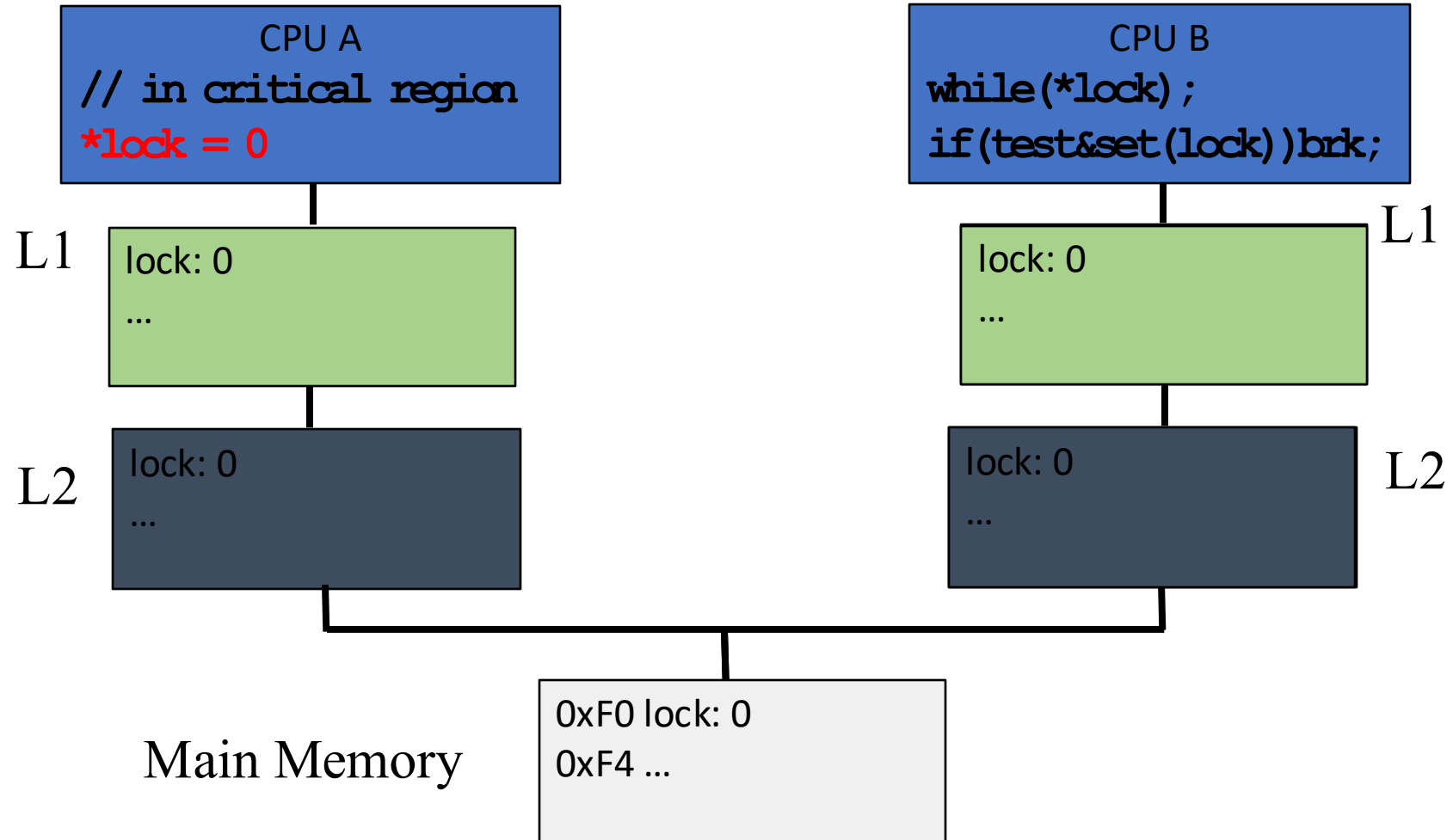
Test & Test & Set with Memory Hierarchies

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



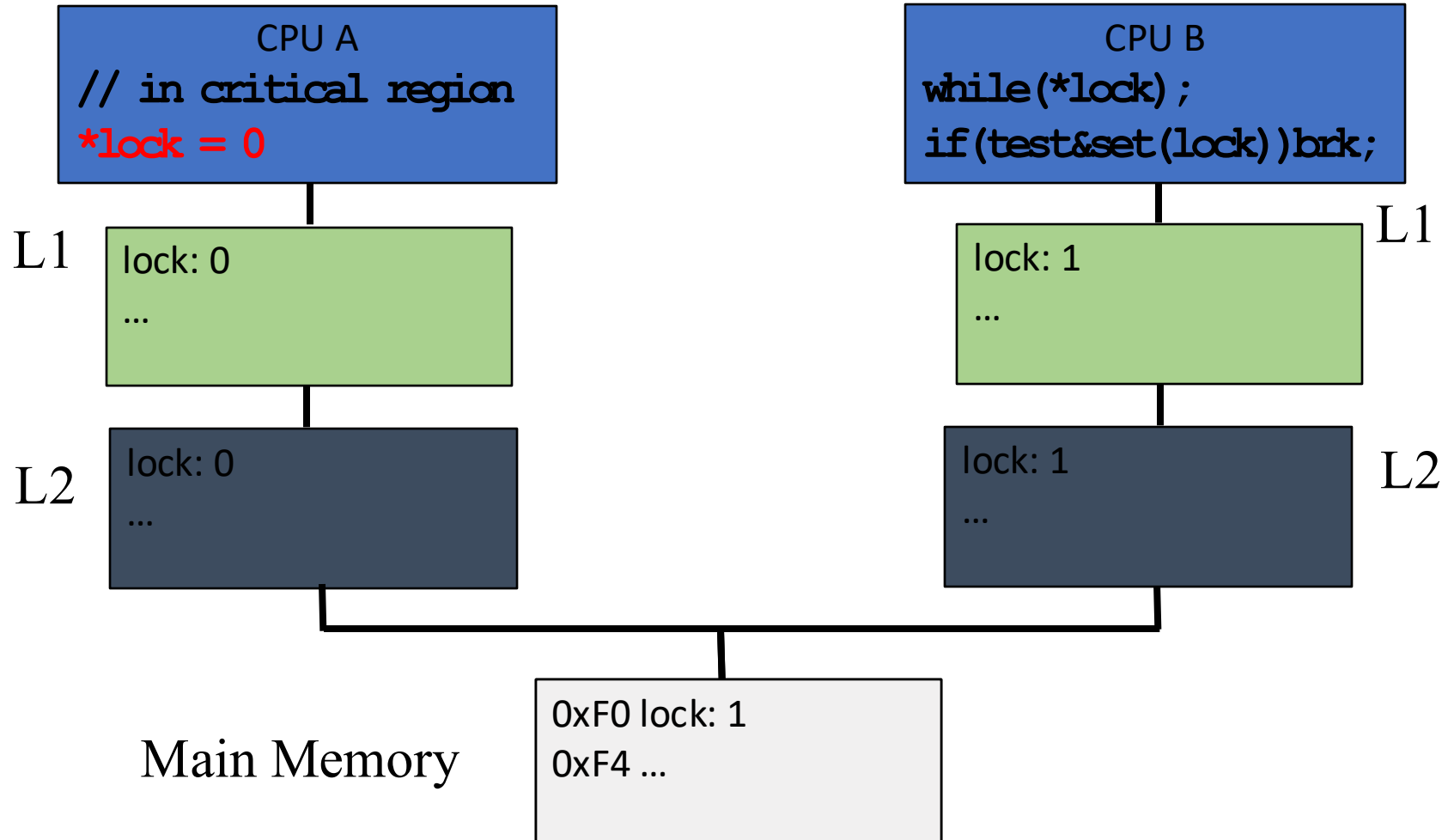
Test & Test & Set with Memory Hierarchies

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



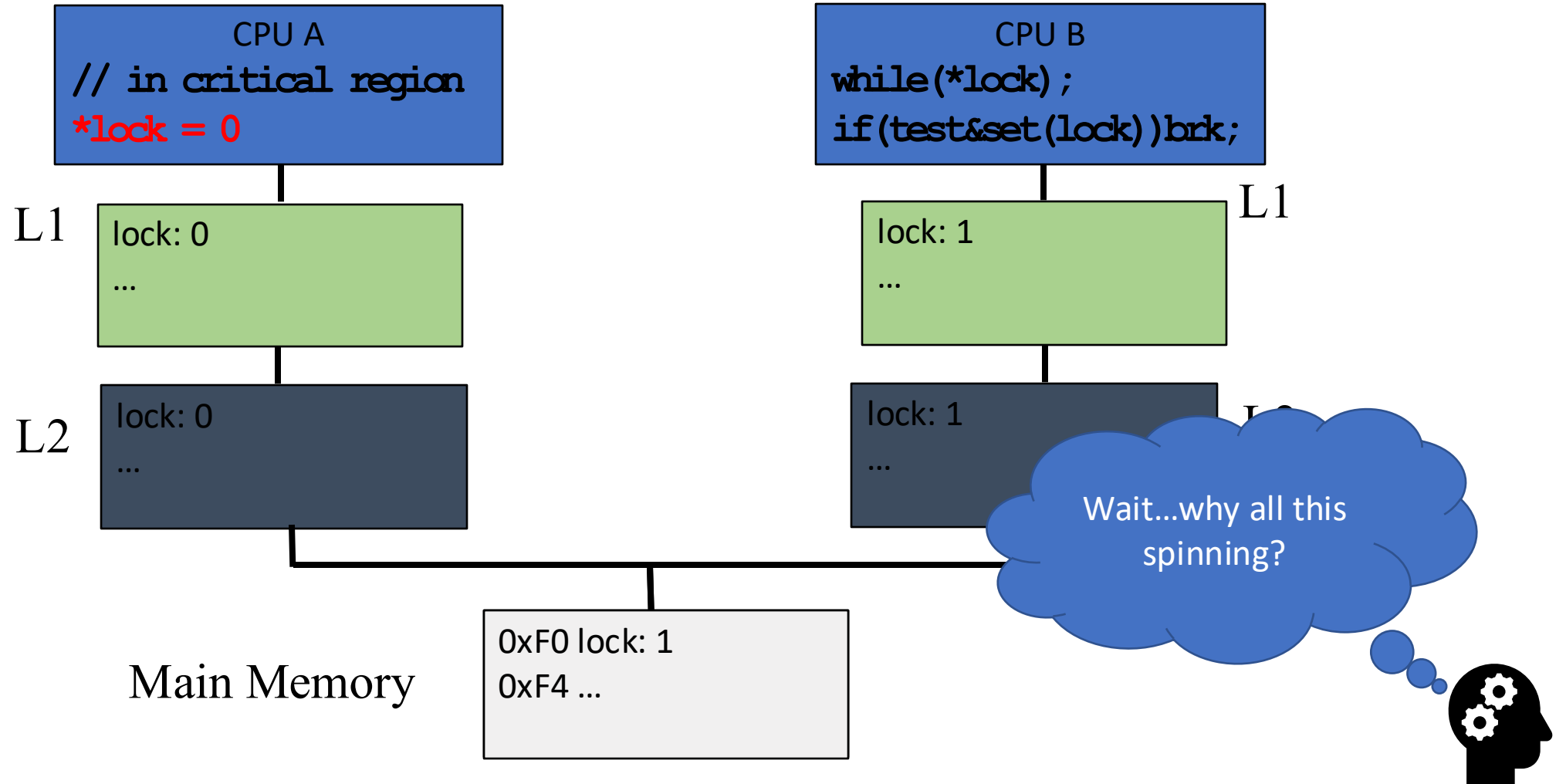
Test & Test & Set with Memory Hierarchies

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
                __CLREXB(); // 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
                __CLREXB(); // 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
                __CLREXB(); // 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?

Lock Pitfalls...

A(prio-0) → `lock(my_lock);`

B(prio-100) → `lock(my_lock);`

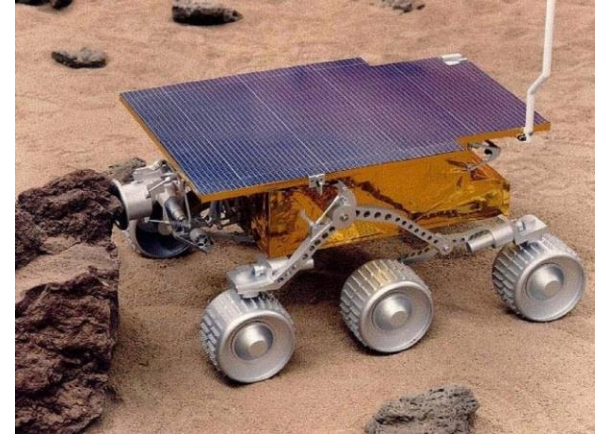


Lock Pitfalls...

A(prio-0) → `lock(my_lock);`

B(prio-100) → `lock(my_lock);`

ACK! Priority Inversion!



Lock Pitfalls...

A(prio-0) → `lock(my_lock);`

B(prio-100) → `lock(my_lock);`

Solution?

ACK! Priority Inversion!



Lock Pitfalls...

A(prio-0) → `lock(my_lock);`

B(prio-100) → `lock(my_lock);`

ACK! Priority Inversion!

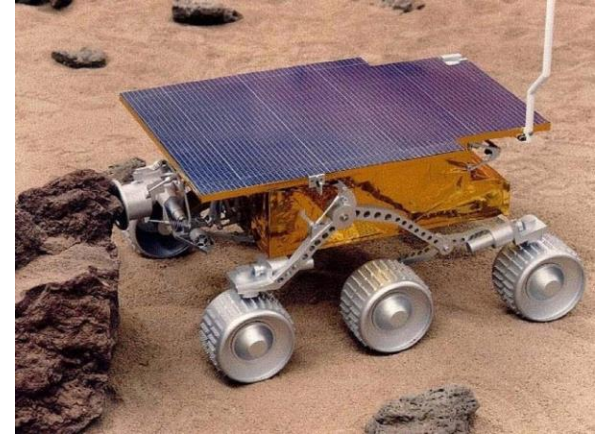
Solution?

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?



Can you build a lock without HW RMW?

Can you build a lock without HW RMW?

Dekker's Algorithm

```

variables
  wants_to_enter : array of 2 booleans
  turn : integer

wants_to_enter[0] ← false
wants_to_enter[1] ← false
turn ← 0 // or 1
  
```

```

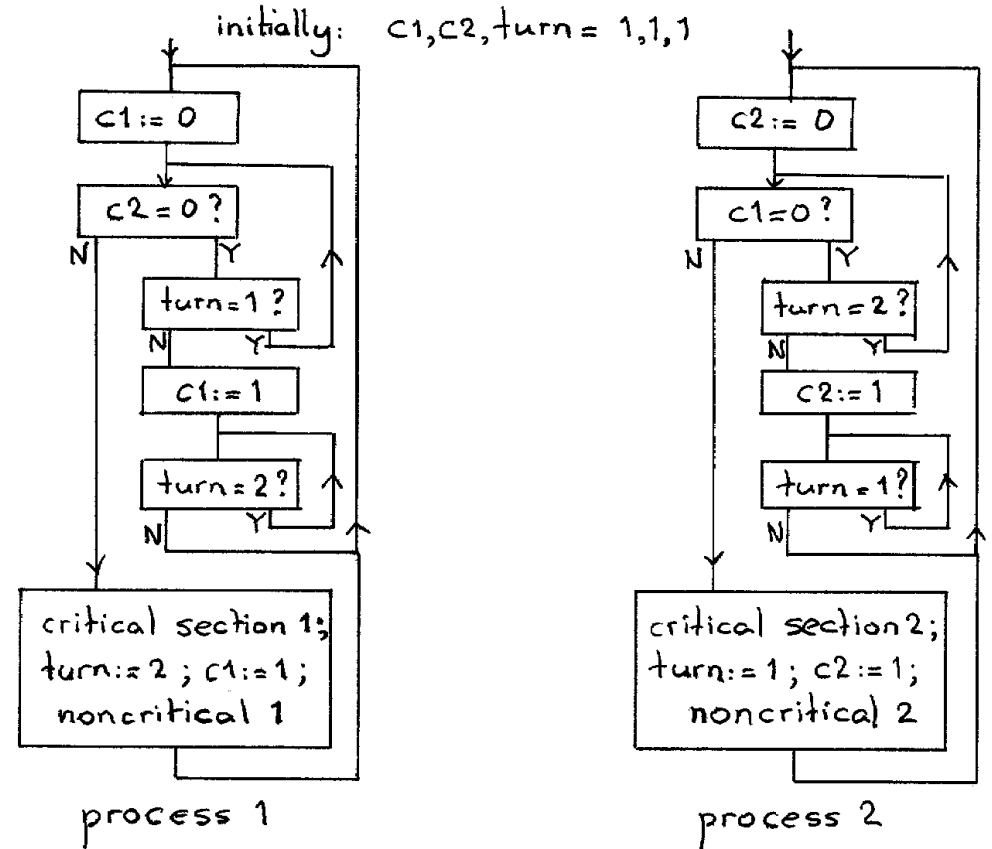
p0:
  wants_to_enter[0] ← true
  while wants_to_enter[1] {
    if turn ≠ 0 {
      wants_to_enter[0] ← false
      while turn ≠ 0 {
        // busy wait
      }
      wants_to_enter[0] ← true
    }
  }

  // critical section
  ...
  turn ← 1
  wants_to_enter[0] ← false
  // remainder section
  
```

```

p1:
  wants_to_enter[1] ← true
  while wants_to_enter[0] {
    if turn ≠ 1 {
      wants_to_enter[1] ← false
      while turn ≠ 1 {
        // busy wait
      }
      wants_to_enter[1] ← true
    }
  }

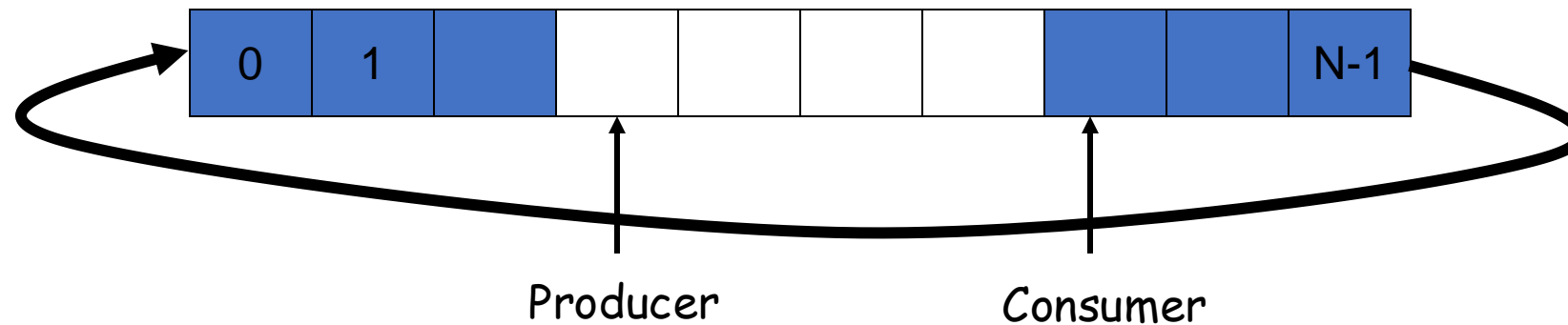
  // critical section
  ...
  turn ← 0
  wants_to_enter[1] ← false
  // remainder section
  
```



Th. J. Dekker's Solution

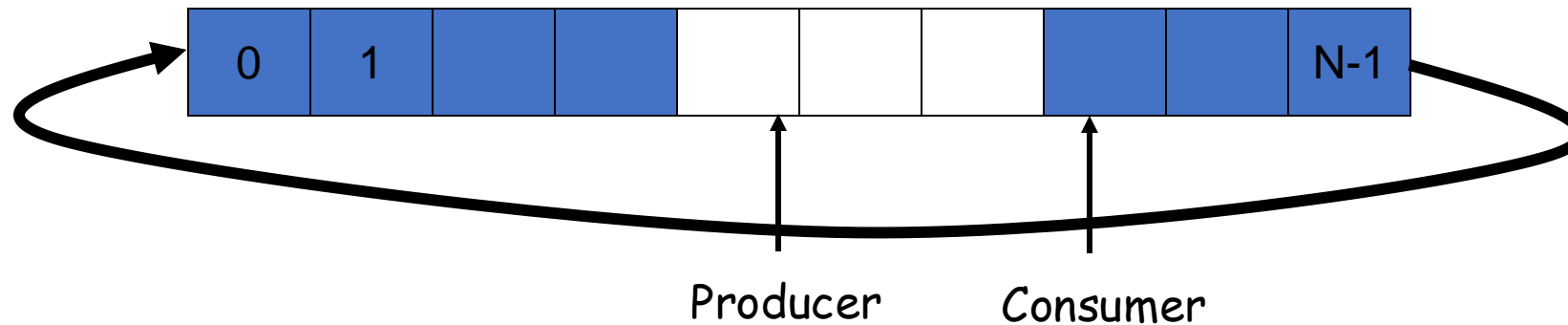
Producer-Consumer (Bounded-Buffer) Problem

- Bounded buffer: size 'N'
 - Access entry 0... N-1, then "wrap around" to 0 again
- Producer process writes data to buffer
 - Must not write more than 'N' items more than consumer "consumes"
- Consumer process reads data from buffer
 - Should not try to consume if there is no data



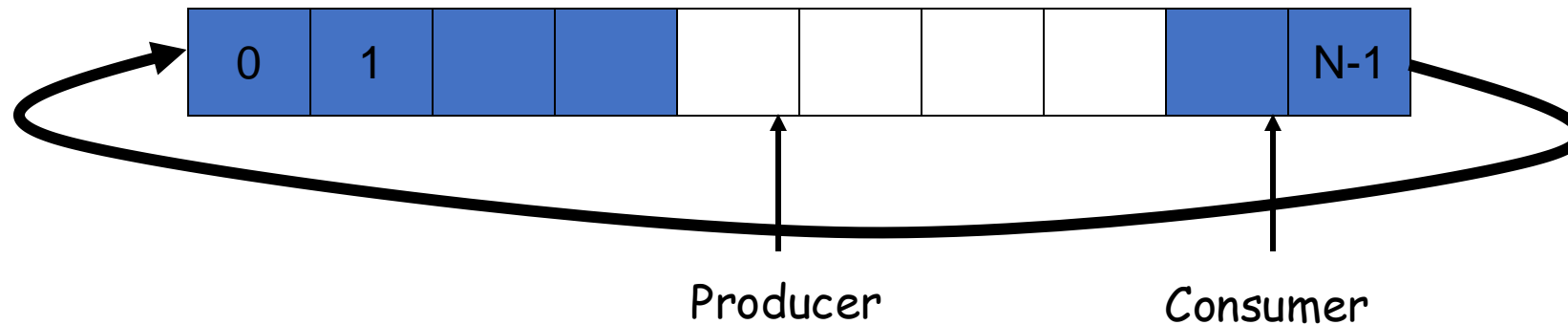
Producer-Consumer (Bounded-Buffer) Problem

- Bounded buffer: size 'N'
 - Access entry 0... N-1, then “wrap around” to 0 again
- Producer process writes data to buffer
 - Must not write more than 'N' items more than consumer “consumes”
- Consumer process reads data from buffer
 - Should not try to consume if there is no data



Producer-Consumer (Bounded-Buffer) Problem

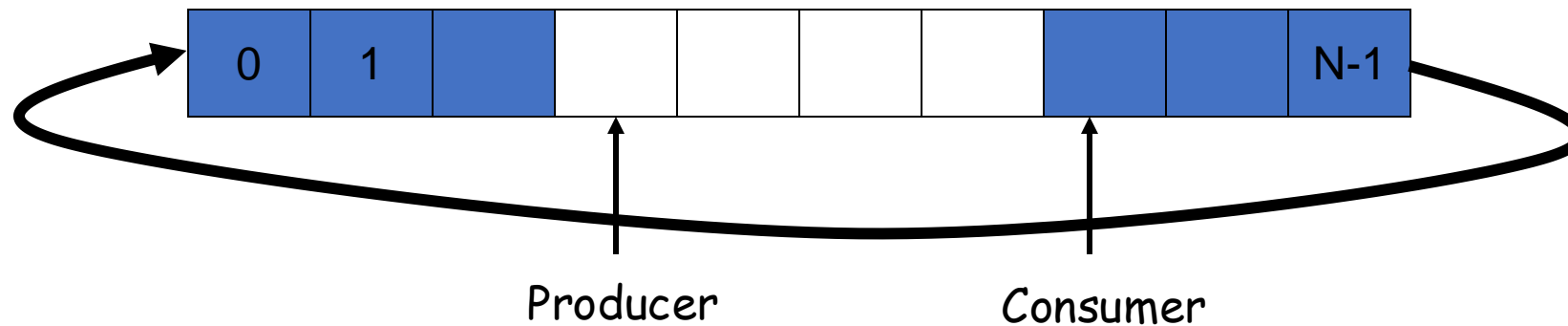
- Bounded buffer: size 'N'
 - Access entry 0... N-1, then "wrap around" to 0 again
- Producer process writes data to buffer
 - Must not write more than 'N' items more than consumer "consumes"
- Consumer process reads data from buffer
 - Should not try to consume if there is no data



OK, let's write some code for this
(using locks only)

```
object array[N]  
void enqueue(object x);  
object dequeue();
```

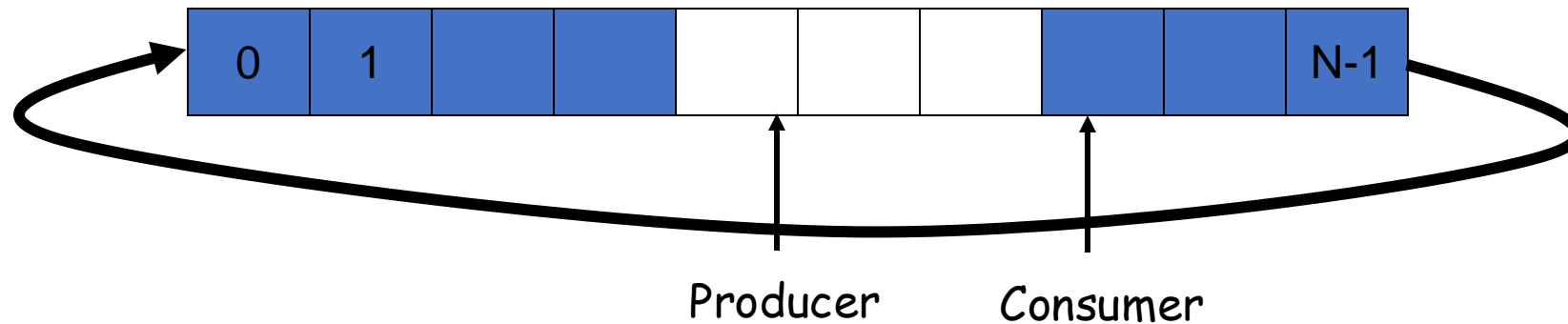
- Bounded buffer: size 'N'
 - Access entry 0... N-1, then "wrap around" to 0 again
- Producer writes data
- Consumer reads data



OK, let's write some code for this
(using locks only)

```
object array[N]  
void enqueue(object x);  
object dequeue();
```

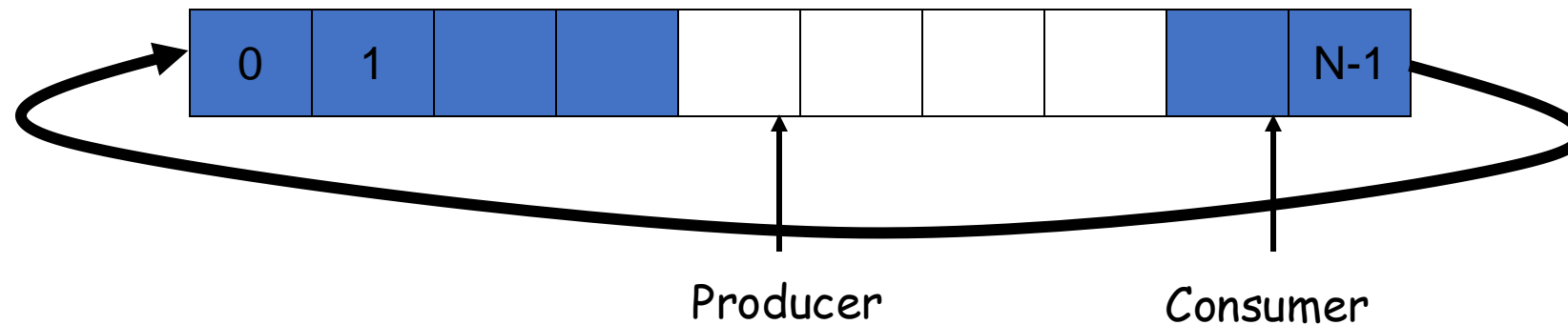
- Bounded buffer: size 'N'
 - Access entry 0... N-1, then "wrap around" to 0 again
- Producer writes data
- Consumer reads data



OK, let's write some code for this
(using locks only)

```
object array[N]  
void enqueue(object x);  
object dequeue();
```

- Bounded buffer: size 'N'
 - Access entry 0... N-1, then "wrap around" to 0 again
- Producer writes data
- Consumer reads data



Semaphore Motivation

Semaphore Motivation

- Problem with locks: mutual exclusion, but *no ordering*

Semaphore Motivation

- Problem with locks: mutual exclusion, but *no ordering*
- Inefficient for producer-consumer (and lots of other things)
 - **Producer**: creates a resource
 - **Consumer**: uses a resource
 - **bounded buffer** between them
 - You need synchronization for correctness, *and...*
 - Scheduling order:
 - **producer waits if buffer full, consumer waits if buffer empty**

Semaphores

- Synchronization variable
 - Integer value
 - Can't access value directly
 - **Must** initialize to some value
 - `sem_init(sem_t *s, int pshared, unsigned int value)`
 - Two operations
 - `sem_wait`, or `down()`, `P()`
 - `sem_post`, or `up()`, `V()`

Semaphores

- Synchronization variable
 - Integer value
 - Can't access value directly
 - **Must** initialize to some value
 - `sem_init(sem_t *s, int pshared, unsigned int value)`
 - Two operations
 - `sem_wait`, or `down()`, `P()`
 - `sem_post`, or `up()`, `V()`

```
int sem_wait(sem_t *s) {  
    wait until value of semaphore s  
    is greater than 0  
    decrement the value of  
    semaphore s by 1  
}
```

```
int sem_post(sem_t *s) {  
    increment the value of  
    semaphore s by 1  
    if there are 1 or more  
    threads waiting, wake 1  
}
```

Semaphores

- Synchronization variable

- Integer value

- Can't access value directly
 - **Must** initialize to some value

- `sem_init(sem_t *s, int pshared, unsigned int value)`

- Two operations

- `sem_wait`, or `down()`, `P()`
 - `sem_post`, or `up()`, `V()`

```
int sem_wait(sem_t *s) {  
    wait until value of semaphore s  
    is greater than 0  
    decrement the value of  
    semaphore s by 1  
}
```

```
int sem_post(sem_t *s) {  
    increment the value of  
    semaphore s by 1  
    if there are 1 or more  
    threads waiting, wake 1  
}
```

```
function V(semaphore S, integer I):  
    [S ← S + I]  
function P(semaphore S, integer I):  
    repeat:  
        if S ≥ I:  
            S ← S - I  
        break ]
```

Semaphore Uses

- Mutual exclusion
 - Semaphore as mutex
 - What should initial value be?

```
// initialize to X  
sem_init(&s, 0, X)
```

```
sem_wait(&s);  
// critical section  
sem_post(&s);
```

Semaphore Uses

- Mutual exclusion
 - Semaphore as mutex
 - What should initial value be?
 - Binary semaphore: $X=1$

```
// initialize to X  
sem_init(s, 0, X)
```

```
sem_wait(s);  
// critical section  
sem_post(s);
```

Semaphore Uses

- Mutual exclusion
 - Semaphore as mutex
 - What should initial value be?
 - Binary semaphore: $X=1$
 - (Counting semaphore: $X>1$)

```
// initialize to X  
sem_init(s, 0, X)
```

```
sem_wait(s);  
// critical section  
sem_post(s);
```

Semaphore Uses

- Mutual exclusion
 - Semaphore as mutex
 - What should initial value be?
 - Binary semaphore: $X=1$
 - (Counting semaphore: $X>1$)
- Scheduling order
 - One thread waits for another

```
// initialize to X  
sem_init(&s, 0, X)
```

```
sem_wait(&s);  
// critical section  
sem_post(&s);
```

Semaphore Uses


- Mutual exclusion
 - Semaphore as mutex
 - What should initial value be?
 - Binary semaphore: $X=1$
 - (Counting semaphore: $X>1$)
- Scheduling order
 - One thread waits for another

```
// initialize to X  
sem_init(s, 0, X)
```

```
sem_wait(s);  
// critical section  
sem_post(s);
```

```
//thread 0  
... // 1st half of computation  
sem_post(s);
```

```
// thread 1  
  
sem_wait(s);  
... //2nd half of computation
```




Semaphore Uses

- Mutual exclusion
 - Semaphore as mutex
 - What should initial value be?
 - Binary semaphore: $X=1$
 - (Counting semaphore: $X>1$)

- Scheduling order
 - One thread waits for another
 - What should initial value be?

```
//thread 0  
... // 1st half of computation  
sem_post(s);
```

```
// thread 1  
  
sem_wait(s);  
... //2nd half of computation
```



```
// initialize to X  
sem_init(s, 0, X)
```

```
sem_wait(s);  
// critical section  
sem_post(s);
```


Producer-Consumer with semaphores

- Two semaphores
 - `sem_t full; // # of filled slots`
 - `sem_t empty; // # of empty slots`

Producer-Consumer with semaphores

- Two semaphores
 - `sem_t full; // # of filled slots`
 - `sem_t empty; // # of empty slots`

```
sem_init(&full, 0, 0);  
sem_init(&empty, 0, N);
```

```
producer() {  
    sem_wait(empty);  
    ... // fill a slot  
    sem_post(full);  
}
```

```
consumer() {  
    sem_wait(full);  
    ... // empty a slot  
    sem_post(empty);  
}
```

Producer-Consumer with semaphores

- Two semaphores
 - `sem_t full; // # of filled slots`
 - `sem_t empty; // # of empty slots`

Is this correct?

```
sem_init(&full, 0, 0);  
sem_init(&empty, 0, N);
```

```
producer() {  
    sem_wait(empty);  
    ... // fill a slot  
    sem_post(full);  
}
```

```
consumer() {  
    sem_wait(full);  
    ... // empty a slot  
    sem_post(empty);  
}
```

Producer-Consumer with semaphores

- Two semaphores
 - `sem_t full; // # of filled slots`
 - `sem_t empty; // # of empty slots`

```
sem_init(&full, 0, 0);  
sem_init(&empty, 0, N);
```

```
producer() {  
    sem_wait(empty);  
    ... // fill a slot  
    sem_post(full);  
}
```

```
consumer() {  
    sem_wait(full);  
    ... // empty a slot  
    sem_post(empty);  
}
```

Producer-Consumer with semaphores

- Two semaphores
 - `sem_t full; // # of filled slots`
 - `sem_t empty; // # of empty slots`
- **Problem: mutual exclusion?**

```
sem_init(&full, 0, 0);  
sem_init(&empty, 0, N);
```

```
producer() {  
    sem_wait(empty);  
    ... // fill a slot  
    sem_post(full);  
}
```

```
consumer() {  
    sem_wait(full);  
    ... // empty a slot  
    sem_post(empty);  
}
```

Producer-Consumer with semaphores

- Three semaphores
 - `sem_t full;` // # of filled slots
 - `sem_t empty;` // # of empty slots
 - `sem_t mutex;` // mutual exclusion

```
sem_init(&full, 0, 0);  
sem_init(&empty, 0, N);  
sem_init(&mutex, 0, 1);
```

```
producer() {  
    sem_wait(empty);  
    sem_wait(&mutex);  
    ... // fill a slot  
    sem_post(&mutex);  
    sem_post(full);  
}
```

```
consumer() {  
    sem_wait(full);  
    sem_wait(&mutex);  
    ... // empty a slot  
    sem_post(&mutex);  
    sem_post(empty);  
}
```

Pthreads and Semaphores

- Type: `pthread_semaphore_t`

```
int pthread_semaphore_init(pthread_spinlock_t *lock);
```

```
int pthread_semaphore_destroy(pthread_spinlock_t *lock);
```

...

- ??????

Pthreads and Semaphores

Pthreads and Semaphores

- No `pthread_semaphore_t`!

Pthreads and Semaphores

- No `pthread_semaphore_t`!
- POSIX does define standard

Pthreads and Semaphores

- No `pthread_semaphore_t`!
- POSIX does define standard
- `#include <semaphore.h>`

■ `int sem_wait(sem_t *sem)`

- P action
- blocks until the semaphore count pointed to by `sem` is greater than zero and then atomically decrements the count

■ `int sem_post(sem_t *sem)`

- V action
- Atomically **increments** the count of the semaphore pointed to by `sem`. If there are any threads blocked on the semaphore, one will be unblocked

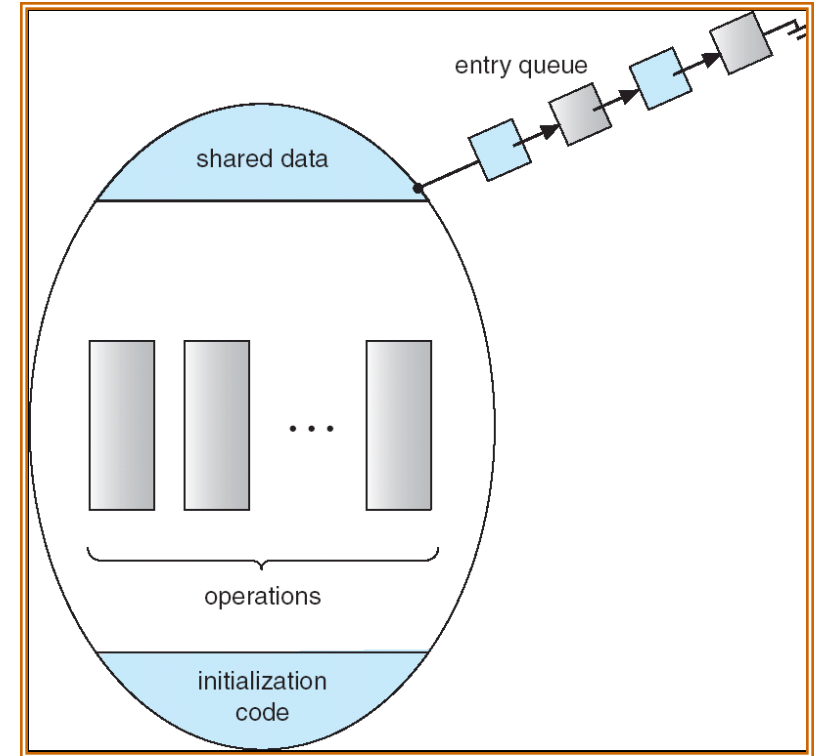
■ `int sem_init(sem_t *sem, int pshared, unsigned int value)`

- Initialize the semaphore to a value
- If `pshared` is 0 then, semaphore is shared between threads of the process
- else shared between processes

What is a monitor?

What is a monitor?

- ❑ Monitor: one big lock for set of operations/ methods
- ❑ Language-level implementation of mutex
- Entry procedure: called from outside
- Internal procedure: called within monitor
- Wait within monitor releases lock



Many variants...

Pthreads and conditions/monitors

- Type `pthread_cond_t`

```
int pthread_cond_init(pthread_cond_t *cond,  
                      const pthread_condattr_t *attr);  
int pthread_cond_destroy(pthread_cond_t *cond);  
int pthread_cond_wait(pthread_cond_t *cond,  
                      pthread_mutex_t *mutex);  
int pthread_cond_signal(pthread_cond_t *cond);  
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Pthreads and conditions/monitors

Why the `pthread_mutex_t` parameter for `pthread_cond_wait`?

- Type `pthread_cond_t`

```
int pthread_cond_init(pthread_cond_t *cond,  
                      const pthread_condattr_t *attr);  
int pthread_cond_destroy(pthread_cond_t *cond);  
int pthread_cond_wait(pthread_cond_t *cond,  
                      pthread_mutex_t *mutex);  
int pthread_cond_signal(pthread_cond_t *cond);  
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Pthreads and conditions/monitors

- Type `pthread_cond_t`

```
int pthread_cond_init(pthread_cond_t *cond,  
                      const pthread_condattr_t *attr);
```

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

```
int pthread_cond_wait(pthread_cond_t *cond,  
                      pthread_mutex_t *mutex);
```

```
int pthread_cond_signal(pthread_cond_t *cond);
```

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Java:

synchronized keyword

wait() / notify() / notifyAll()

C#: Monitor class

Enter() / Exit() /

Pulse() / PulseAll()

Does this code work?

Does this code work?

```
1 public class SynchronizedQueue<T> {
2
3     public void enqueue(T item) {
4         lock.lock();
5         try {
6             if(head == tail - 1)
7                 notFull.wait();
8             Q[head] = item;
9             if(++head == MAX_Q)
10                 head = 0;
11             notEmpty.signal();
12         } finally {
13             lock.unlock();
14         }
15     }
16
17     public T dequeue() {
18         T retval = null;
19         lock.lock();
20         try {
21             if(head == tail)
22                 notEmpty.wait();
23             retval = Q[tail];
24             if(++tail == MAX_Q)
25                 tail = 0;
26             notFull.signal();
27         } finally {
28             lock.unlock();
29         }
30     }
31 }
```

Does this code work?

```
1 public class SynchronizedQueue<T> {  
2  
3     public void enqueue(T item) {  
4         lock.lock();  
5         try {  
6             if(head == tail - 1)  
7                 notFull.wait();  
8             Q[head] = item;  
9             if(++head == MAX_Q)  
10                 head = 0;  
11             notEmpty.signal();  
12         } finally {  
13             lock.unlock();  
14         }  
15     }  
16  
17     public T dequeue() {  
18         T retval = null;  
19         lock.lock();  
20         try {  
21             if(head == tail)  
22                 notEmpty.wait();  
23             retval = Q[tail];  
24             if(++tail == MAX_Q)  
25                 tail = 0;  
26             notFull.signal();  
27         } finally {  
28             lock.unlock();  
29         }  
30     }  
31 }
```

```
private Lock lock = new ReentrantLock();  
private Condition notEmpty = lock.newCondition();  
private Condition notFull = lock.newCondition();  
private int head = 0;  
private int tail = 0;  
private int size = MAX_Q;  
private T[] Q = new T[size];
```

Does this code work?

```
1 public class SynchronizedQueue<T> {  
2  
3     public void enqueue(T item) {  
4         lock.lock();  
5         try {  
6             if(head == tail - 1)  
7                 notFull.wait();  
8             Q[head] = item;  
9             if(++head == MAX_Q)  
10                 head = 0;  
11             notEmpty.signal();  
12         } finally {  
13             lock.unlock();  
14         }  
15     }  
16  
17     public T dequeue() {  
18         T retval = null;  
19         lock.lock();  
20         try {  
21             if(head == tail)  
22                 notEmpty.wait();  
23             retval = Q[tail];  
24             if(++tail == MAX_Q)  
25                 tail = 0;  
26             notFull.signal();  
27         } finally {  
28             lock.unlock();  
29         }  
30     }  
31 }
```

```
private Lock lock = new ReentrantLock();  
private Condition notEmpty = lock.newCondition();  
private Condition notFull = lock.newCondition();  
private int head = 0;  
private int tail = 0;  
private int size = MAX_Q;  
private T[] Q = new T[size];
```

- Uses “if” to check invariants.

Does this code work?

```
1 public class SynchronizedQueue<T> {  
2  
3     public void enqueue(T item) {  
4         lock.lock();  
5         try {  
6             if(head == tail - 1)  
7                 notFull.wait();  
8             Q[head] = item;  
9             if(++head == MAX_Q)  
10                 head = 0;  
11             notEmpty.signal();  
12         } finally {  
13             lock.unlock();  
14         }  
15     }  
16  
17     public T dequeue() {  
18         T retval = null;  
19         lock.lock();  
20         try {  
21             if(head == tail)  
22                 notEmpty.wait();  
23             retval = Q[tail];  
24             if(++tail == MAX_Q)  
25                 tail = 0;  
26             notFull.signal();  
27         } finally {  
28             lock.unlock();  
29         }  
30     }  
31 }
```

```
private Lock lock = new ReentrantLock();  
private Condition notEmpty = lock.newCondition();  
private Condition notFull = lock.newCondition();  
private int head = 0;  
private int tail = 0;  
private int size = MAX_Q;  
private T[] Q = new T[size];
```

- Uses “if” to check invariants.
- Why doesn't if work?

Does this code work?

```
1 public class SynchronizedQueue<T> {
2
3     public void enqueue(T item) {
4         lock.lock();
5         try {
6             if(head == tail - 1)
7                 notFull.wait();
8             Q[head] = item;
9             if(++head == MAX_Q)
10                head = 0;
11             notEmpty.signal();
12         } finally {
13             lock.unlock();
14         }
15     }
16
17     public T dequeue() {
18         T retval = null;
19         lock.lock();
20         try {
21             if(head == tail)
22                 notEmpty.wait();
23             retval = Q[tail];
24             if(++tail == MAX_Q)
25                 tail = 0;
26             notFull.signal();
27         } finally {
28             lock.unlock();
29         }
30     }
31 }
```

```
private Lock lock = new ReentrantLock();
private Condition notEmpty = lock.newCondition();
private Condition notFull = lock.newCondition();
private int head = 0;
private int tail = 0;
private int size = MAX_Q;
private T[] Q = new T[size];
```

- Uses “if” to check invariants.
- Why doesn't if work?
- How could we MAKE it work?

Hoare-style Monitors

(aka blocking condition variables)

Given entrance queue 'e', signal queue 's', condition var 'c'

```
enter:  
  if (locked):  
    e.push_back(thread)  
  else  
    lock
```

```
schedule:  
  if s.any()  
    t ← s.pop_first()  
    t.run  
  else if e.any()  
    t ← e.pop_first()  
    t.run  
  else  
    unlock // monitor unoccupied
```

```
wait C:  
  
  C.q.push_back(thread)  
  schedule // block this thread
```

```
signal C :  
  
  if (C.q.any())  
    t = C.q.pop_front() // t → "the signaled thread"  
    s.push_back(t)  
    t.run  
    // block this thread
```

- Leave calls schedule
- Signaler must wait, but gets priority over threads on entrance queue
- How is this different from Mesa monitors?
- Is s queue necessary?

Mesa-style monitors

(aka non-blocking condition variables)

```
enter:
    if locked:
        e.push_back(thread)
        block
    else
        lock
```

```
schedule:
    if e.any()
        t ← e.pop_front
        t.run
    else
        unlock
```

notify C:

```
if C.q.any()
t ← C.q.pop_front() // t is "notified"
e.push_back(t)
```

wait C:

```
C.q.push_back(thread)
schedule
block
```

- (Leave calls schedule)
- Can be extended with extra queues for priority
- What are the differences?

Example: anyone see a bug?

StorageAllocator: MONITOR = BEGIN
 availableStorage: INTEGER;
 moreAvailable: CONDITION;

Allocate: ENTRY PROCEDURE [*size*: INTEGER
RETURNS [*p*: POINTER] = BEGIN
 UNTIL *availableStorage* \geq *size*
 DO WAIT *moreAvailable* ENDLOOP;
 p \leftarrow <remove chunk of size words & update *availableStorage*>
 END;

Free: ENTRY PROCEDURE [*p*: POINTER, *Size*: INTEGER] = BEGIN
 <put back chunk of size words & update *availableStorage*>;
 NOTIFY *moreAvailable* END;

Expand: PUBLIC PROCEDURE [*pOld*: POINTER, *size*: INTEGER] RETURNS [*pNew*: POINTER] = BEGIN
 pNew \leftarrow *Allocate*[*size*];
 <copy contents from old block to new block>;
 Free[*pOld*] END;

END.

Example: anyone see a bug?

StorageAllocator: MONITOR = BEGIN
 availableStorage: INTEGER;
 moreAvailable: CONDITION;

Allocate: ENTRY PROCEDURE [*size*: INTEGER
RETURNS [*p*: POINTER] = BEGIN
 UNTIL *availableStorage* \geq *size*
 DO WAIT *moreAvailable* ENDLOOP;
 p \leftarrow <remove chunk of size words & update *availableStorage*>
 END;

Free: ENTRY PROCEDURE [*p*: POINTER, *Size*: INTEGER] = BEGIN
 <put back chunk of size words & update *availableStorage*>;
 NOTIFY *moreAvailable* END;

Expand: PUBLIC PROCEDURE [*pOld*: POINTER, *size*: INTEGER] RETURNS [*pNew*: POINTER] = BEGIN
 pNew \leftarrow *Allocate*[*size*];
 <copy contents from old block to new block>;
 Free[*pOld*] END;

END.

Barriers

Barriers

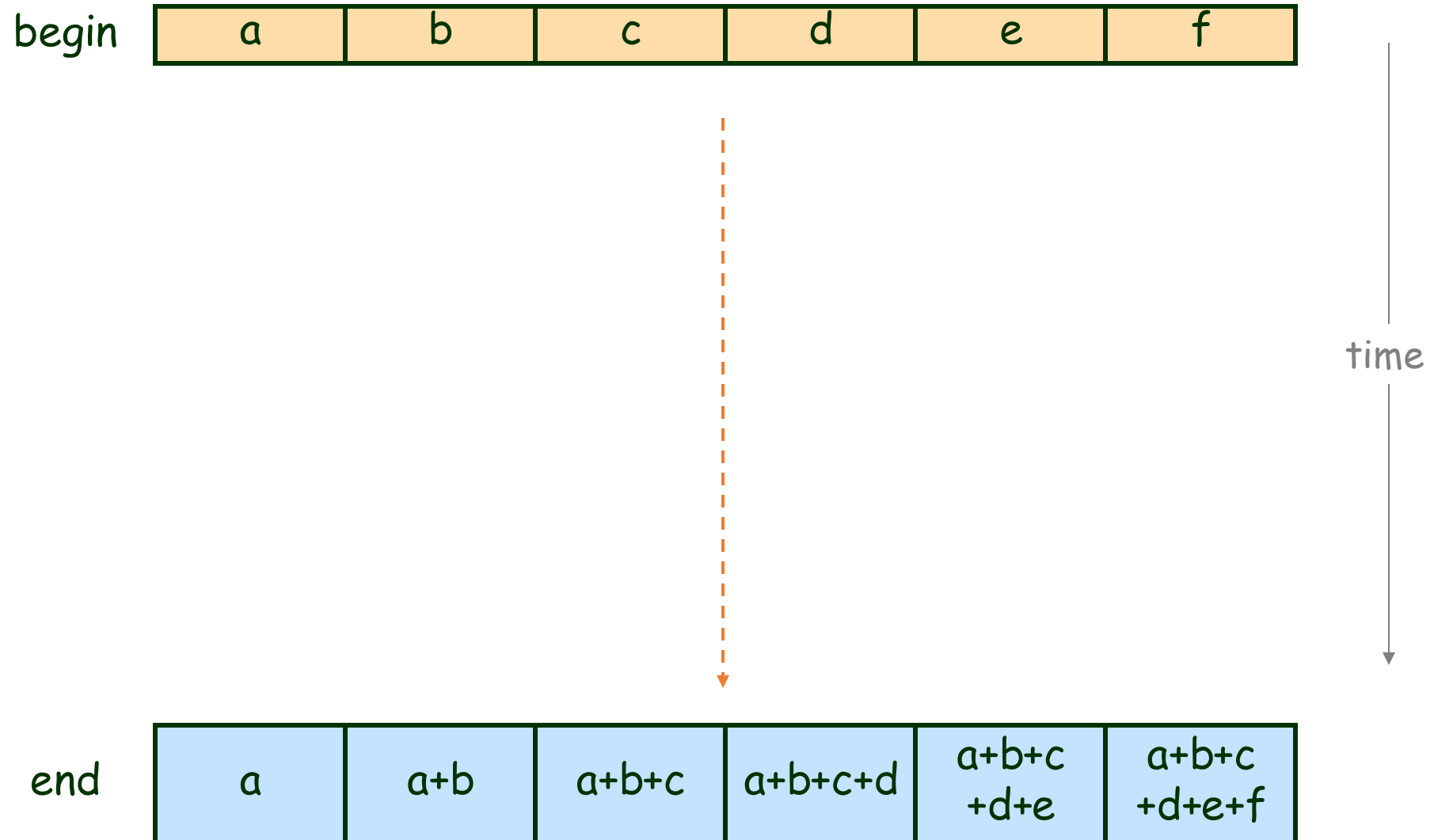


Prefix Sum

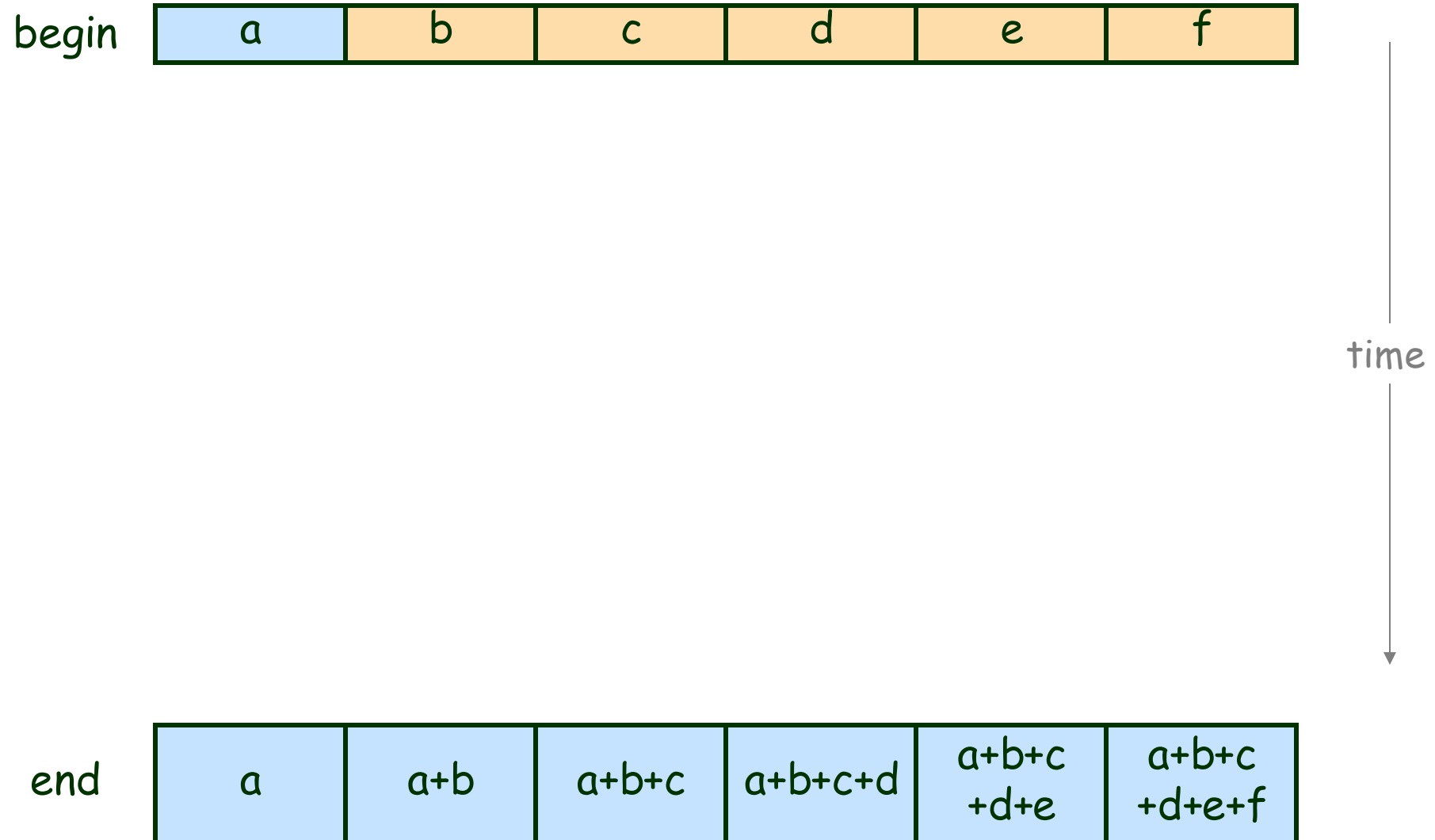
Prefix Sum



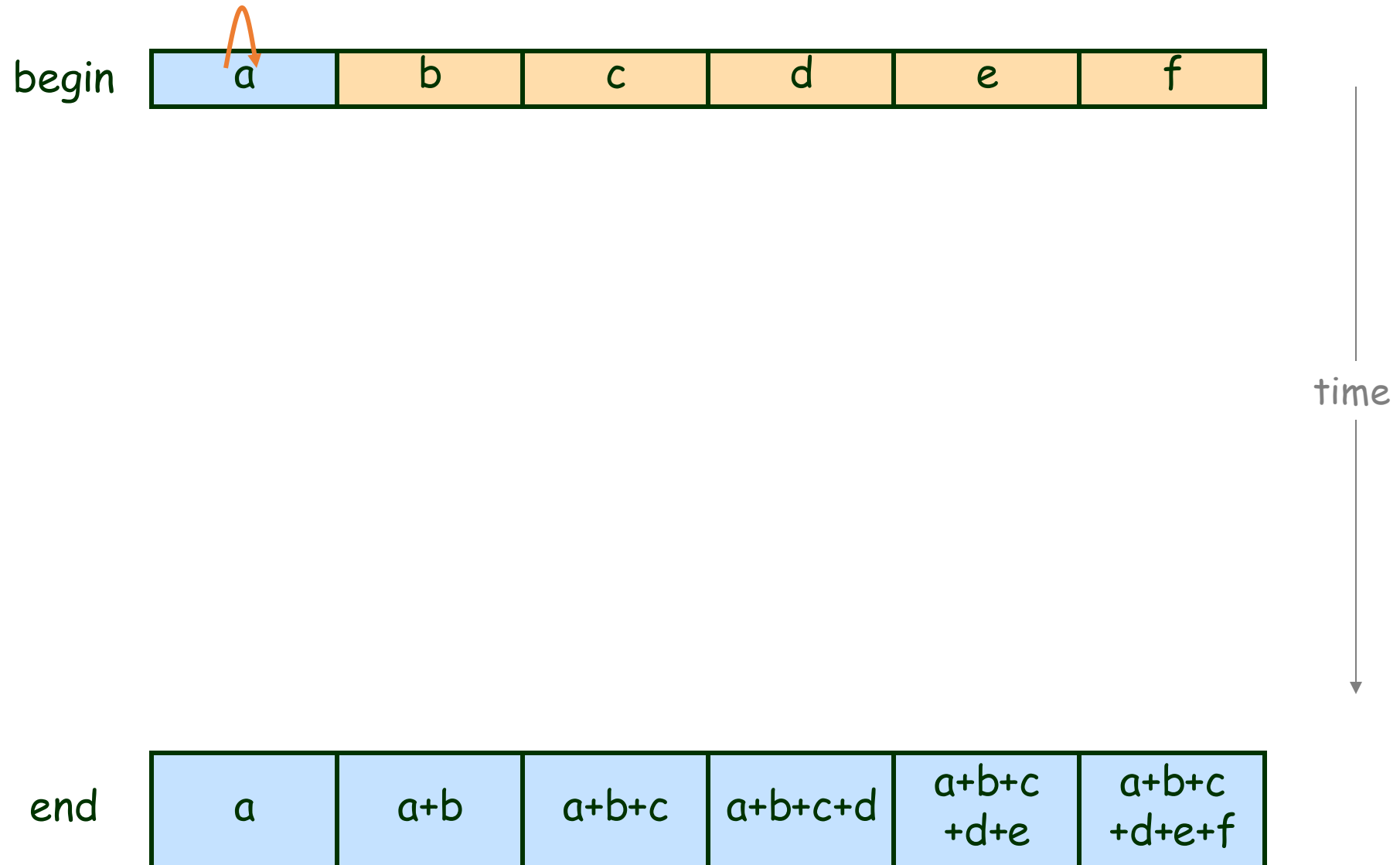
Prefix Sum



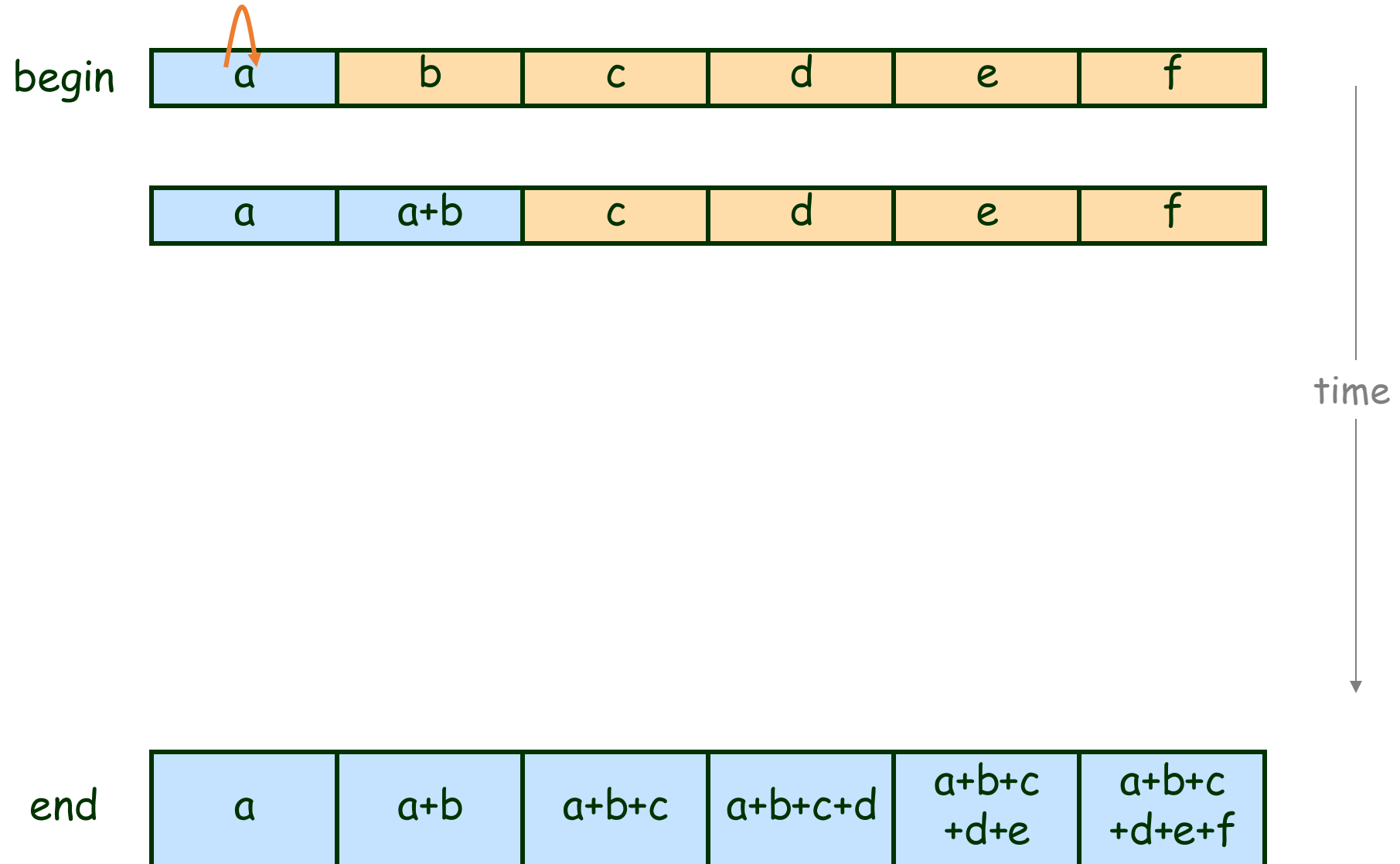
Prefix Sum



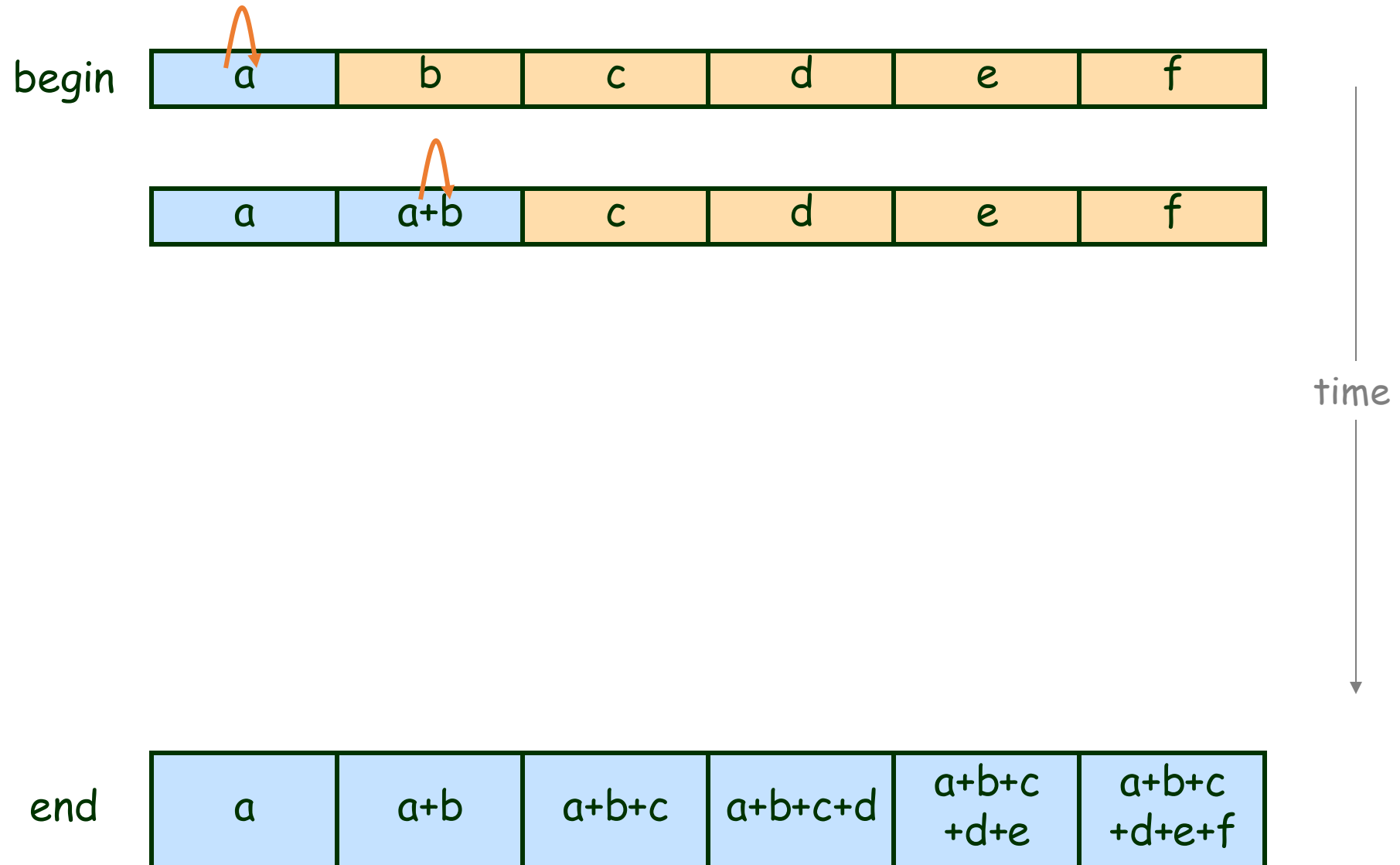
Prefix Sum



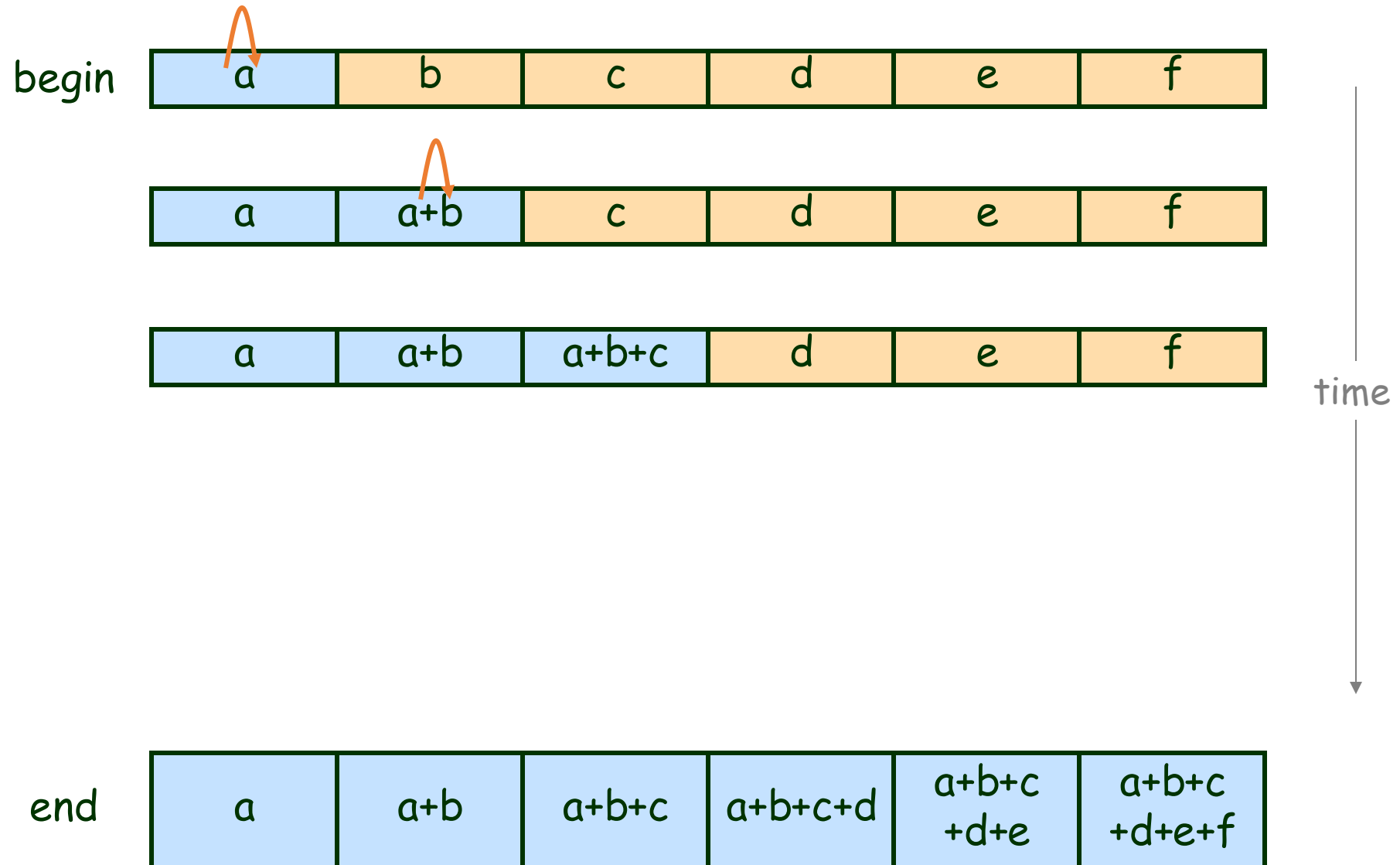
Prefix Sum



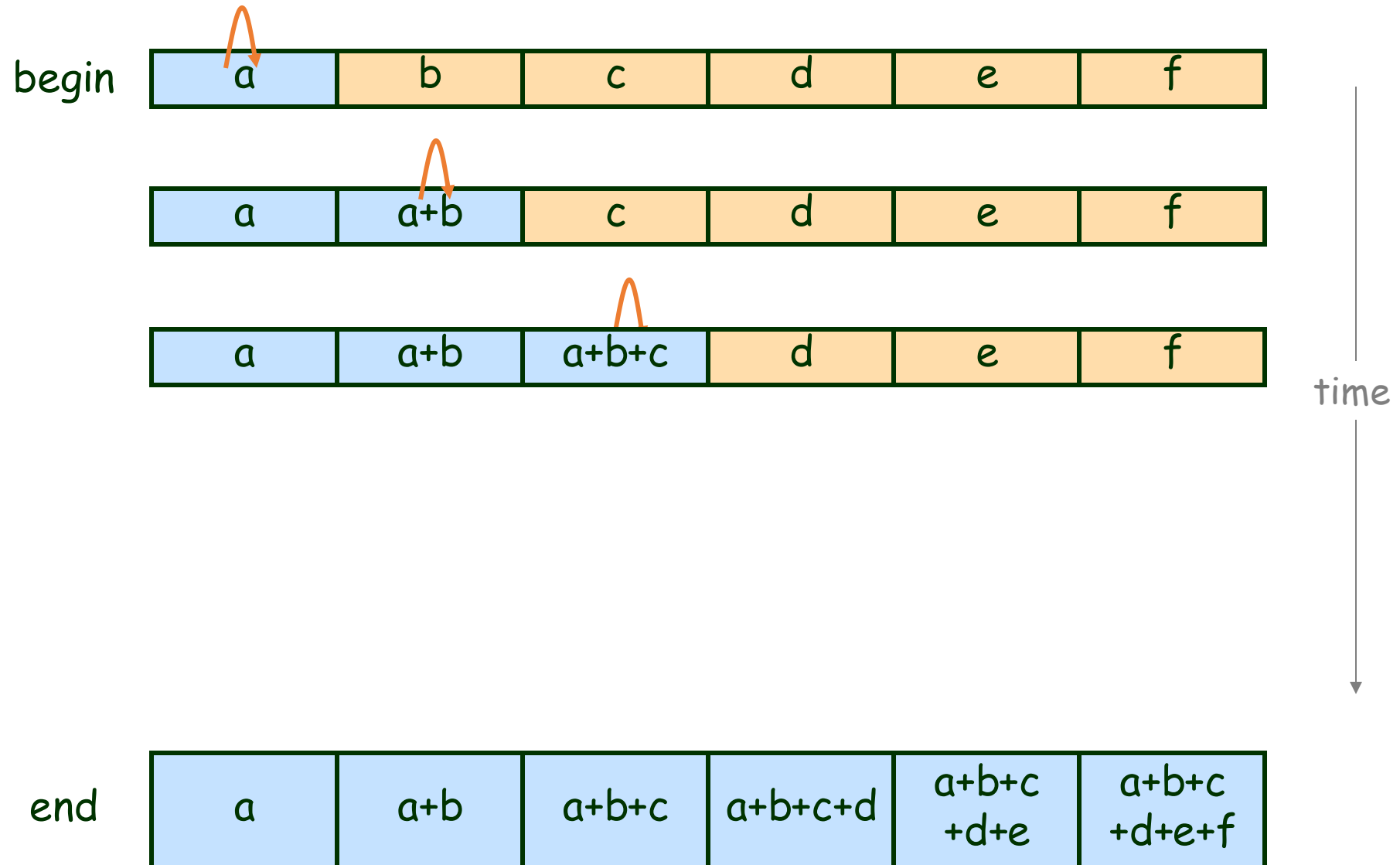
Prefix Sum



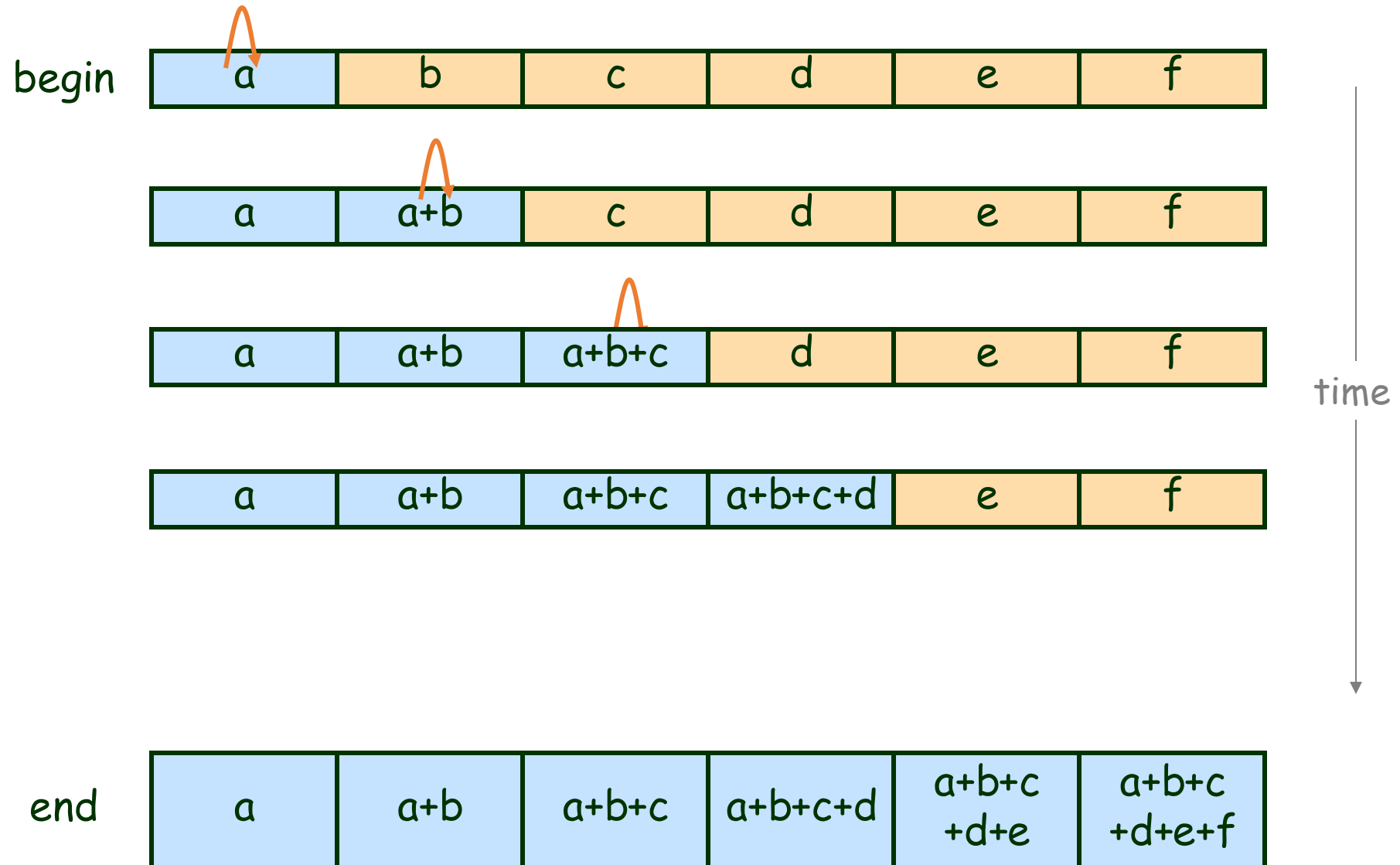
Prefix Sum



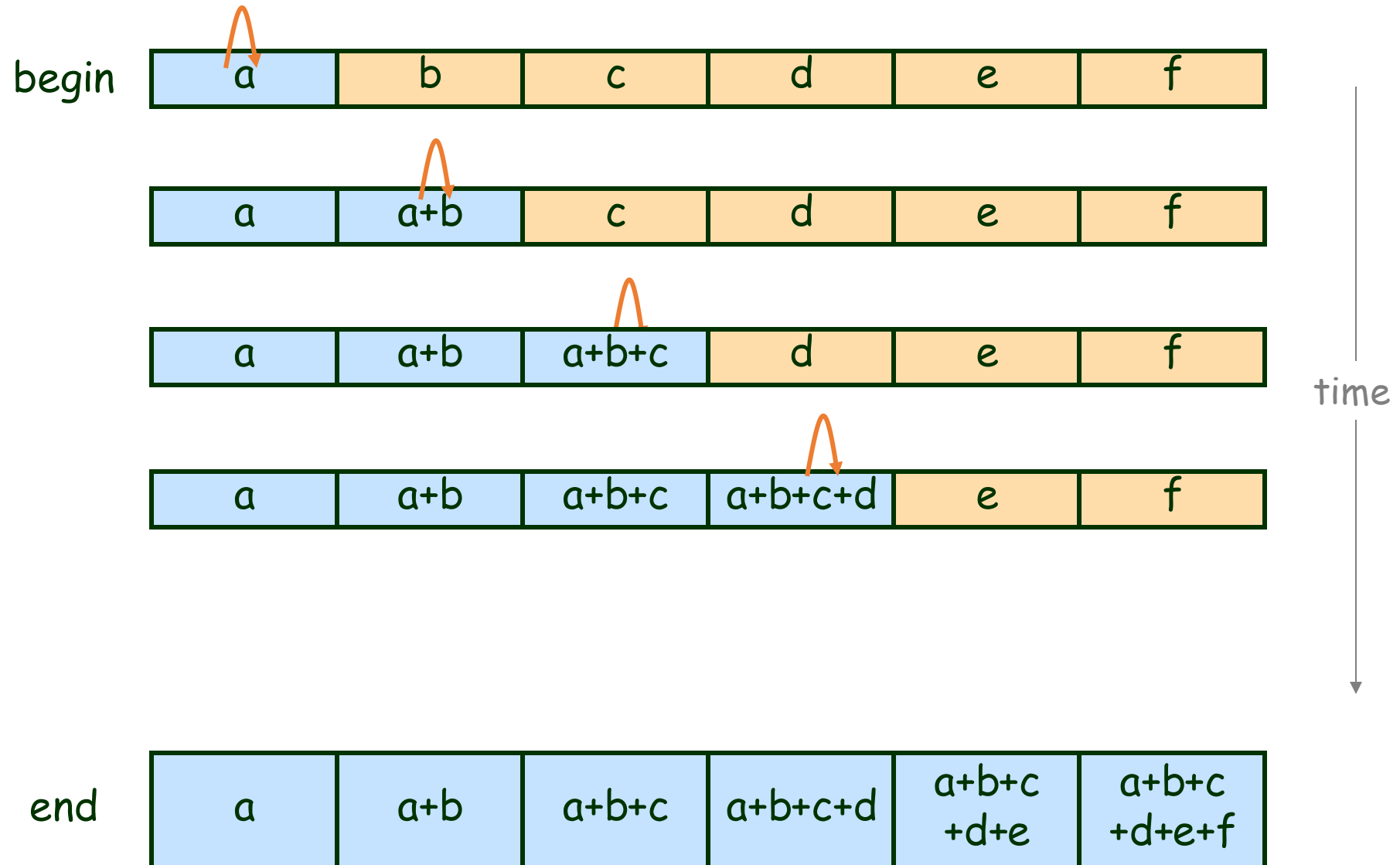
Prefix Sum



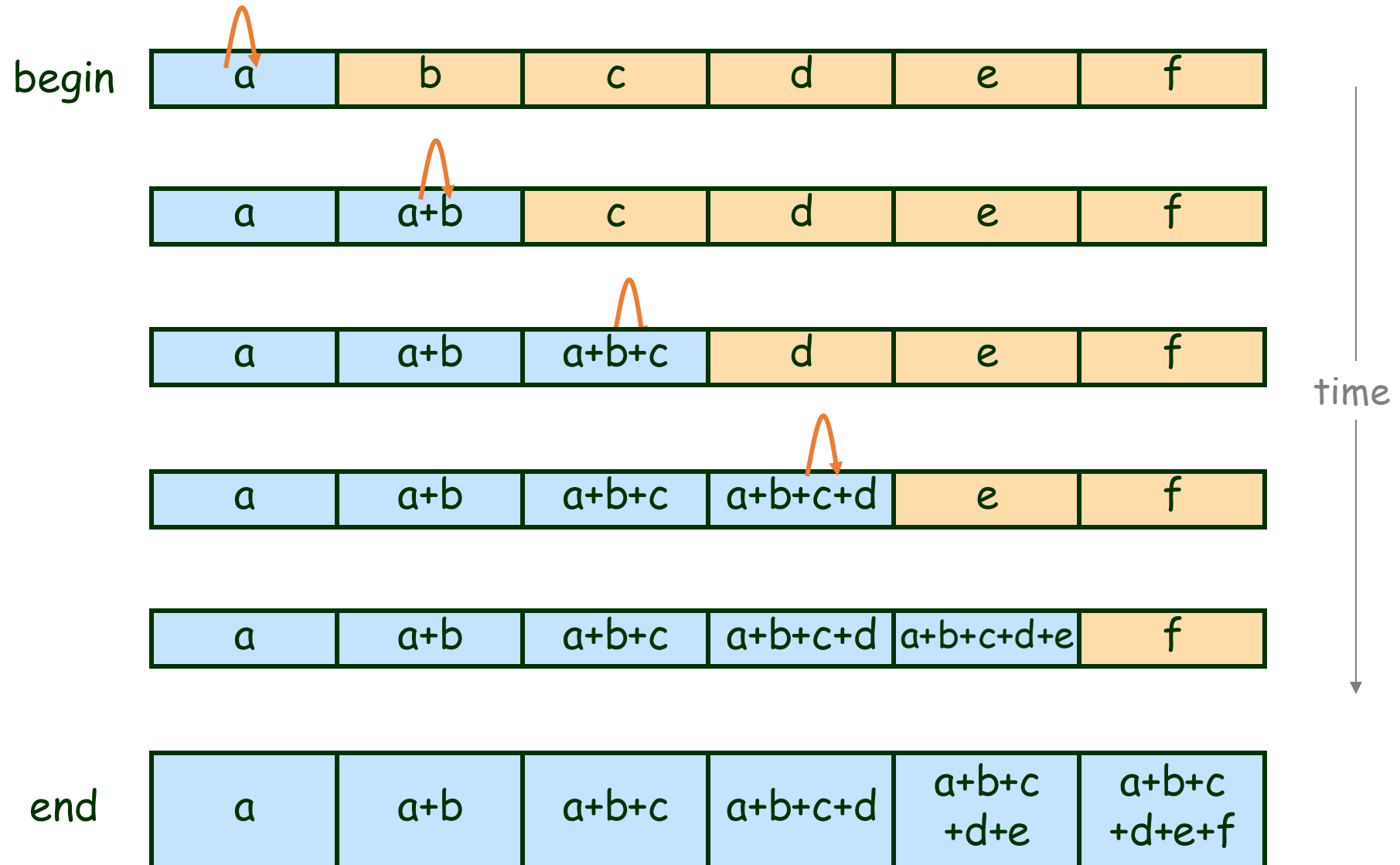
Prefix Sum



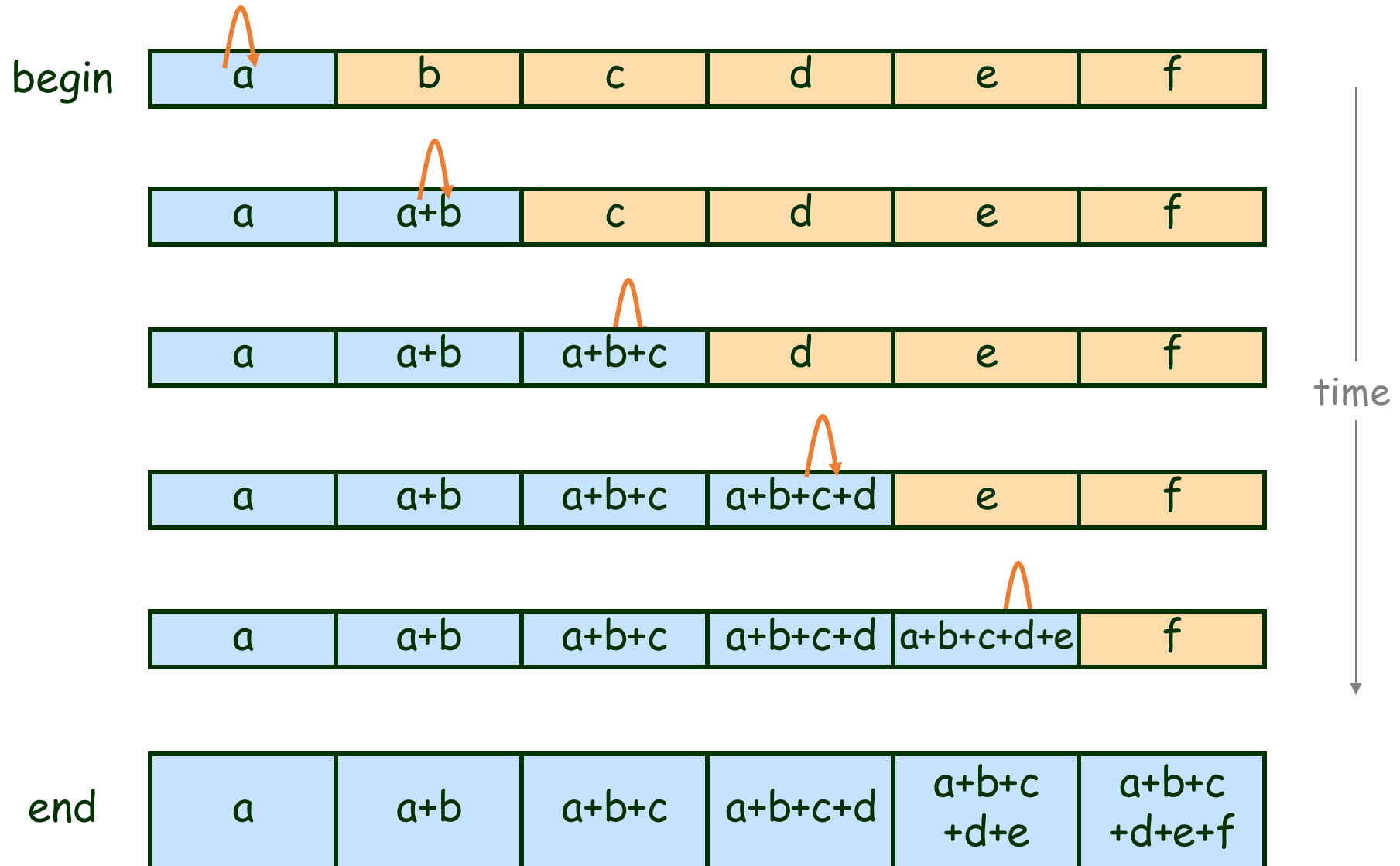
Prefix Sum



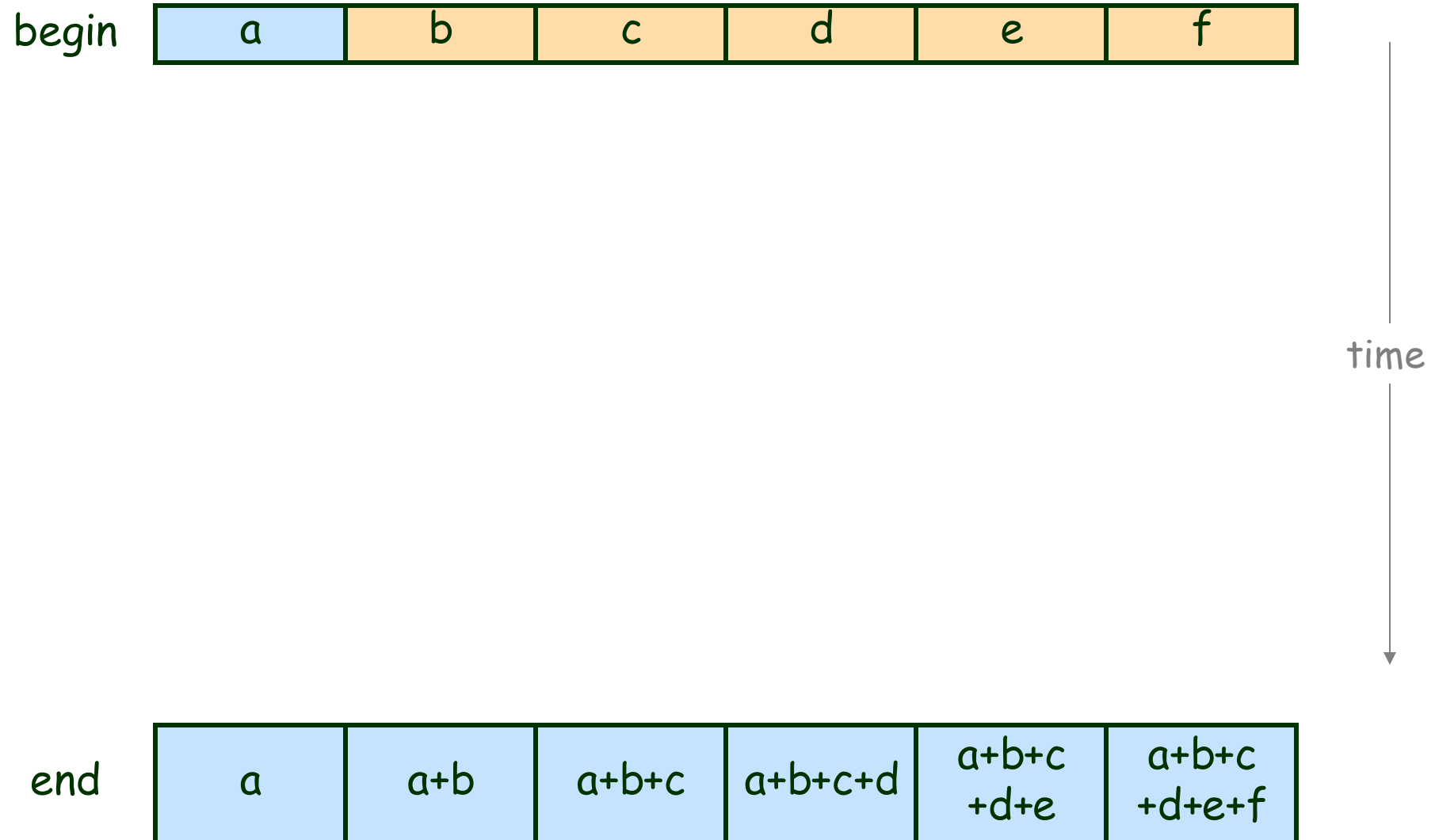
Prefix Sum



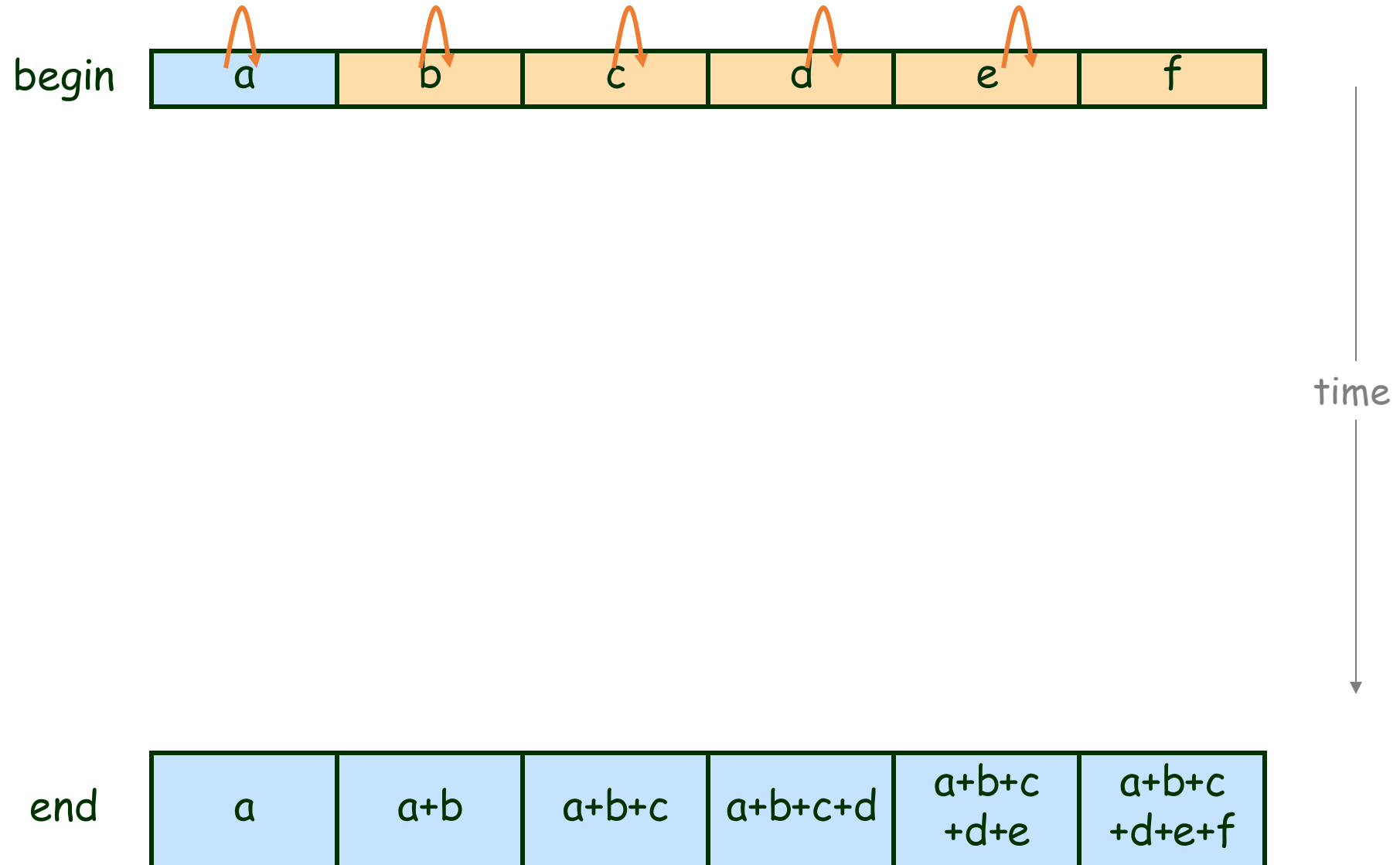
Prefix Sum



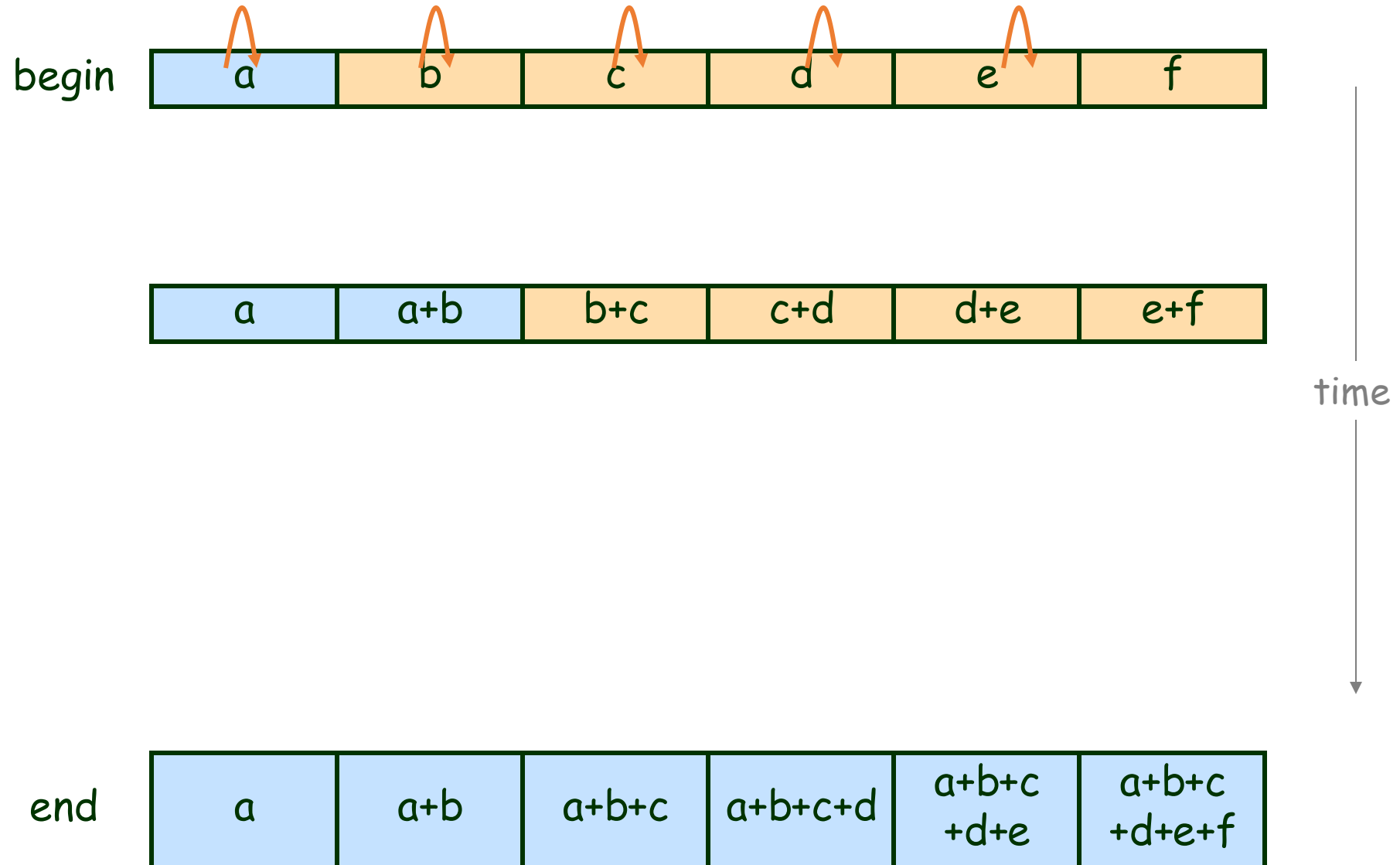
Parallel Prefix Sum



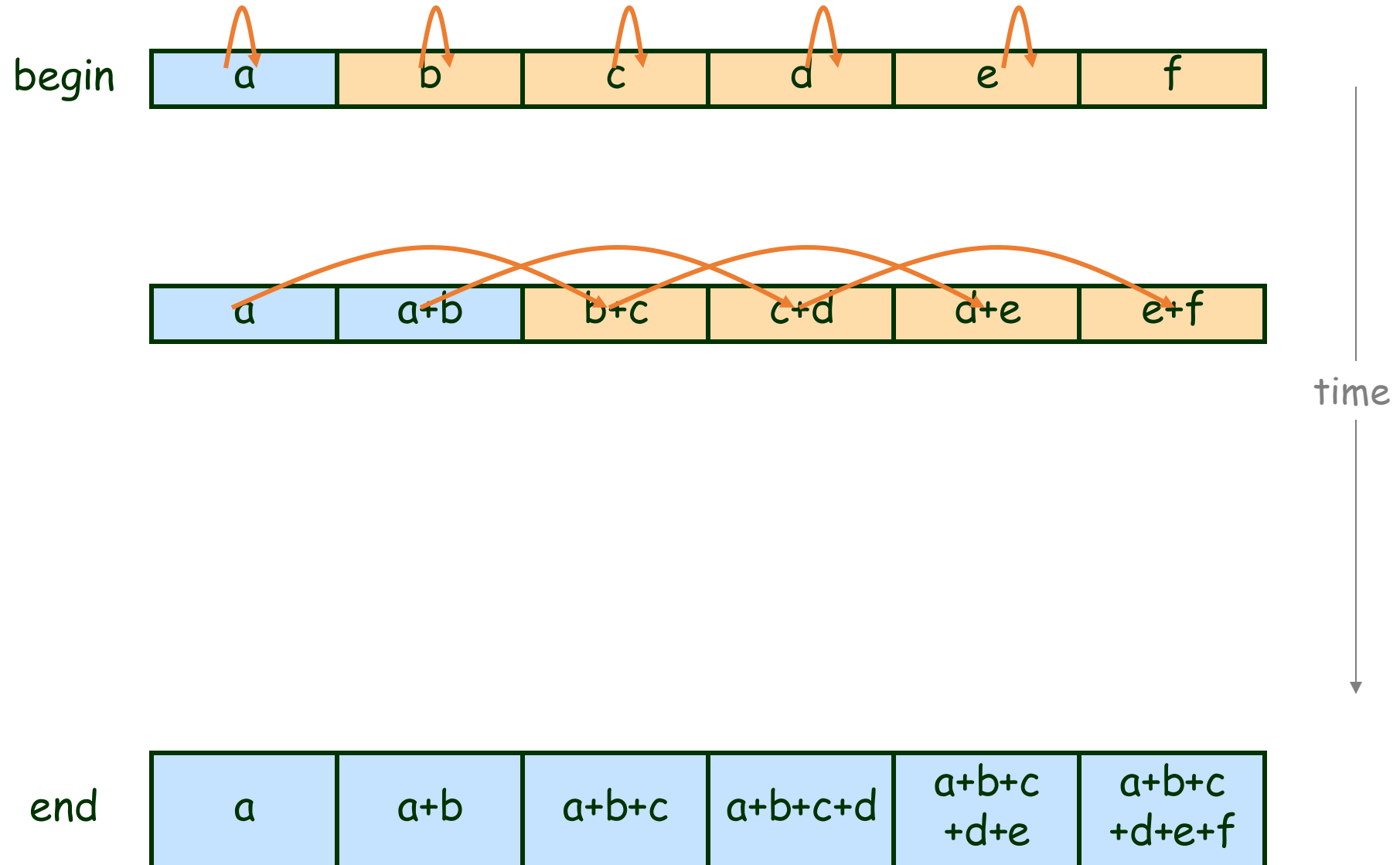
Parallel Prefix Sum



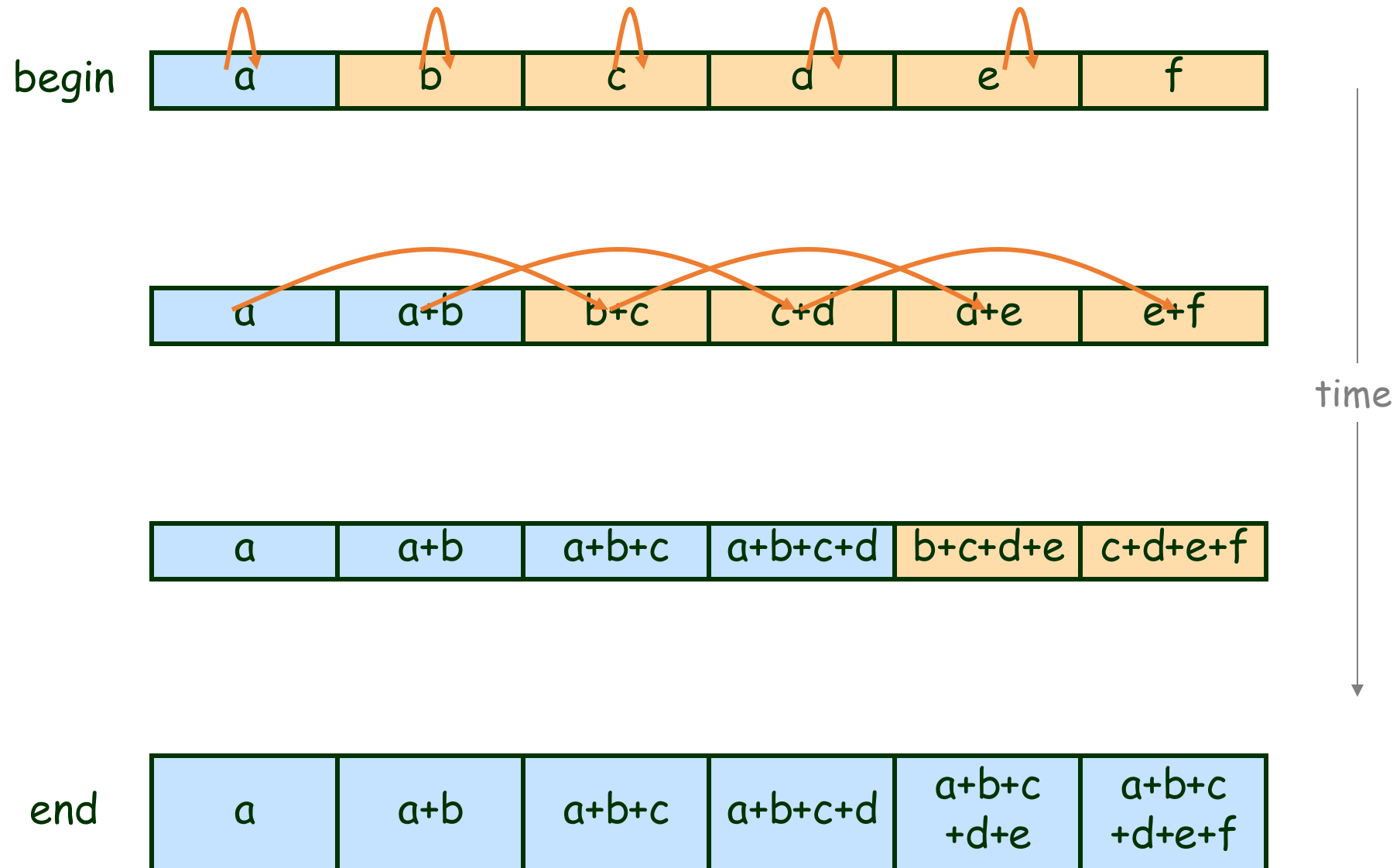
Parallel Prefix Sum



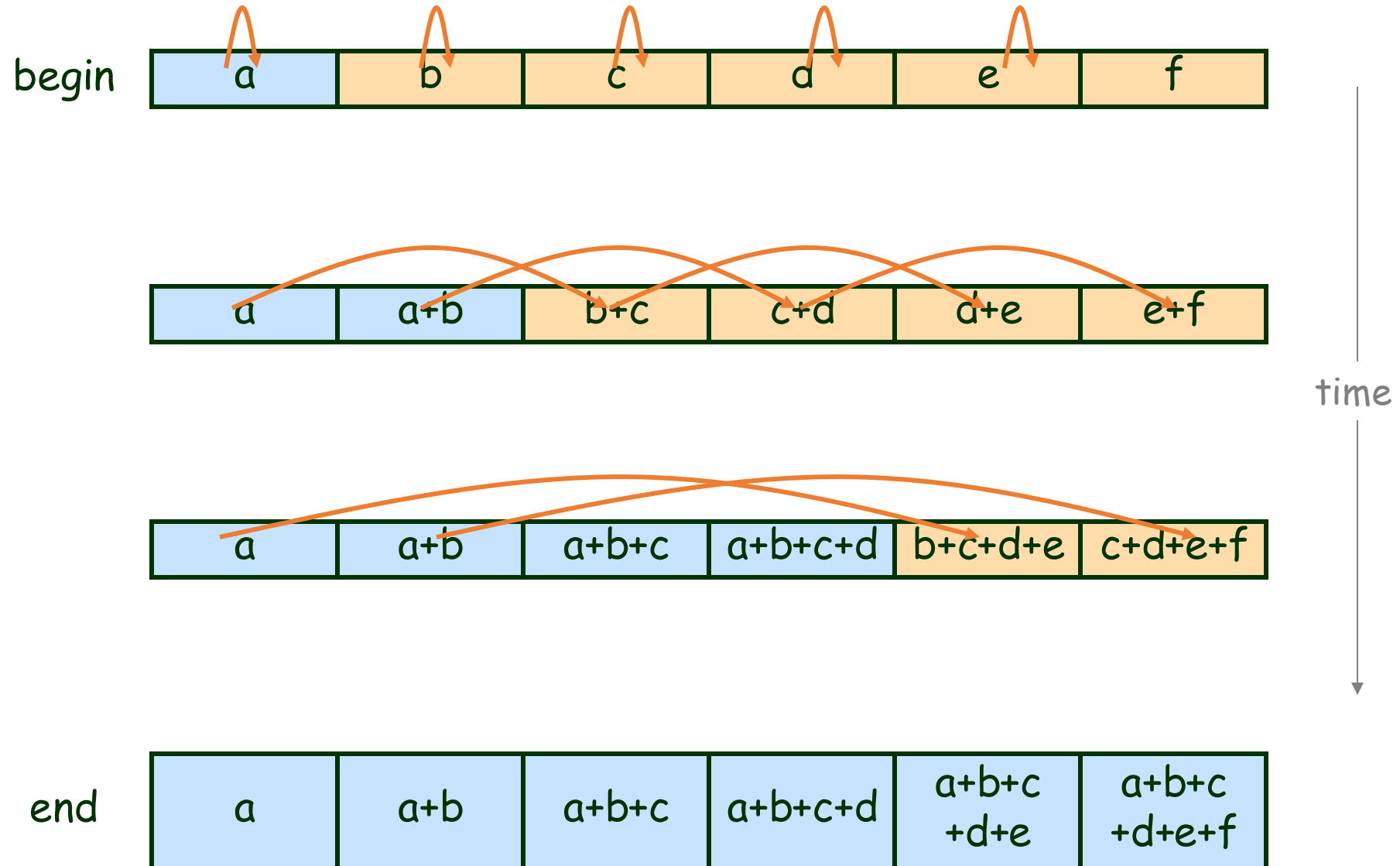
Parallel Prefix Sum



Parallel Prefix Sum

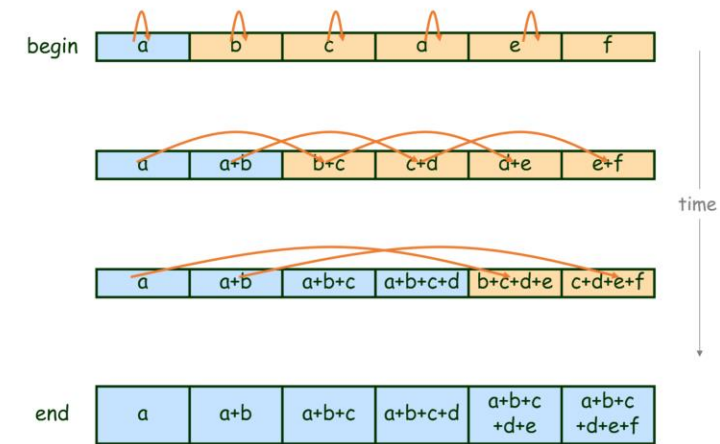


Parallel Prefix Sum



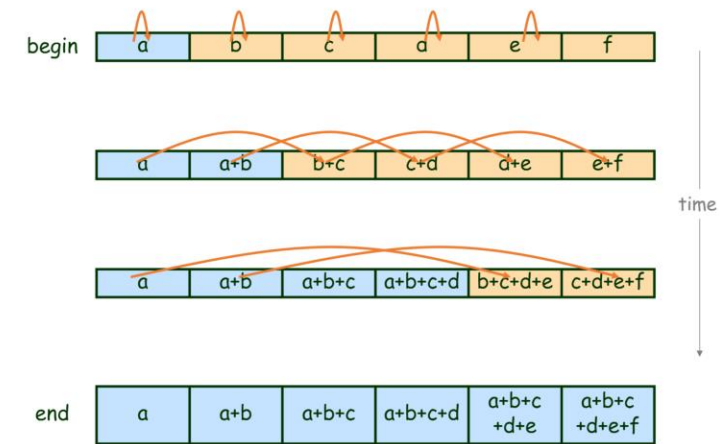
Pthreads Parallel Prefix Sum

```
int g_values[N] = { a, b, c, d, e, f };  
  
void prefix_sum_thread(void * param) {  
  
    int i;  
    int id = *((int*)param);  
    int stride = 0;  
  
    for(stride=1; stride<=N/2; stride<<1) {  
        g_values[id+stride] += g_values[id];  
    }  
  
}
```



Pthreads Parallel Prefix Sum

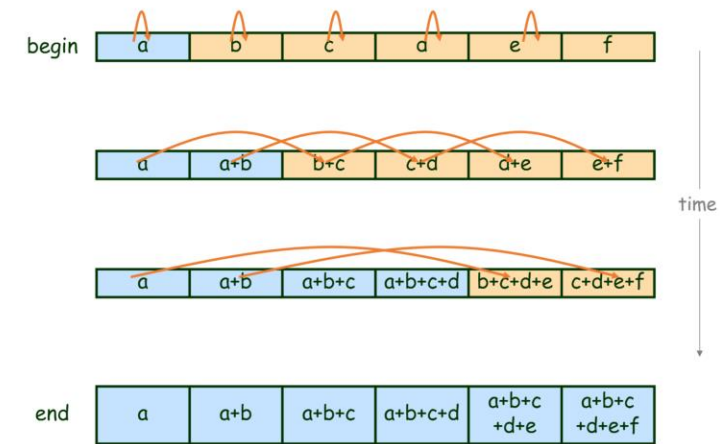
```
int g_values[N] = { a, b, c, d, e, f };  
  
void prefix_sum_thread(void * param) {  
  
    int i;  
    int id = *((int*)param);  
    int stride = 0;  
  
    for(stride=1; stride<=N/2; stride<<1) {  
        g_values[id+stride] += g_values[id];  
    }  
  
}
```



Will this
work?

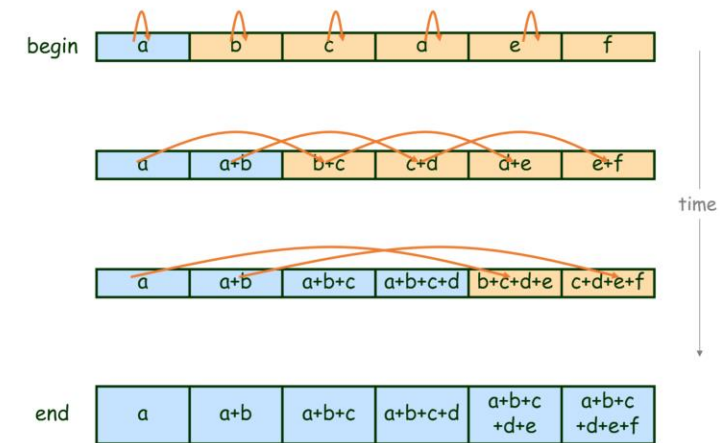
Pthreads Parallel Prefix Sum

```
pthread_mutex_t g_locks[N] = { MUTEX_INITIALIZER, ...};  
int g_values[N] = { a, b, c, d, e, f };  
  
void prefix_sum_thread(void * param) {  
  
    int i;  
    int id = *((int*)param);  
    int stride = 0;  
  
    for(stride=1; stride<=N/2; stride<<1) {  
        pthread_mutex_lock(&g_locks[id]);  
        pthread_mutex_lock(&g_locks[id+stride]);  
        g_values[id+stride] += g_values[id];  
        pthread_mutex_unlock(&g_locks[id]);  
        pthread_mutex_unlock(&g_locks[id+stride]);  
    }  
  
}
```



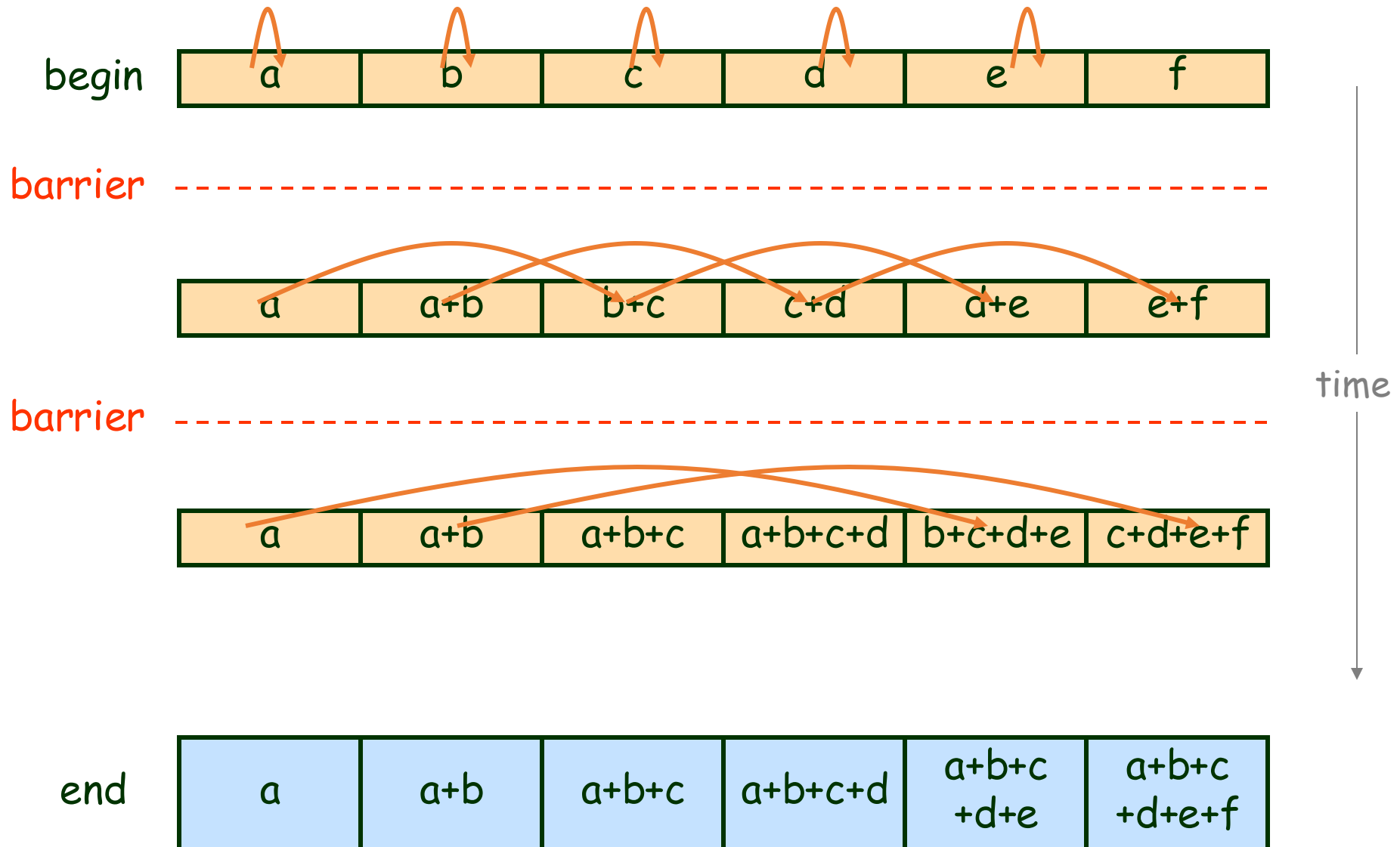
Pthreads Parallel Prefix Sum

```
pthread_mutex_t g_locks[N] = { MUTEX_INITIALIZER, ...};  
int g_values[N] = { a, b, c, d, e, f };  
  
void prefix_sum_thread(void * param) {  
  
    int i;  
    int id = *((int*)param);  
    int stride = 0;  
  
    for(stride=1; stride<=N/2; stride<<1) {  
        pthread_mutex_lock(&g_locks[id]);  
        pthread_mutex_lock(&g_locks[id+stride]);  
        g_values[id+stride] += g_values[id];  
        pthread_mutex_unlock(&g_locks[id]);  
        pthread_mutex_unlock(&g_locks[id+stride]);  
    }  
  
}
```



fixed?

Parallel Prefix Sum

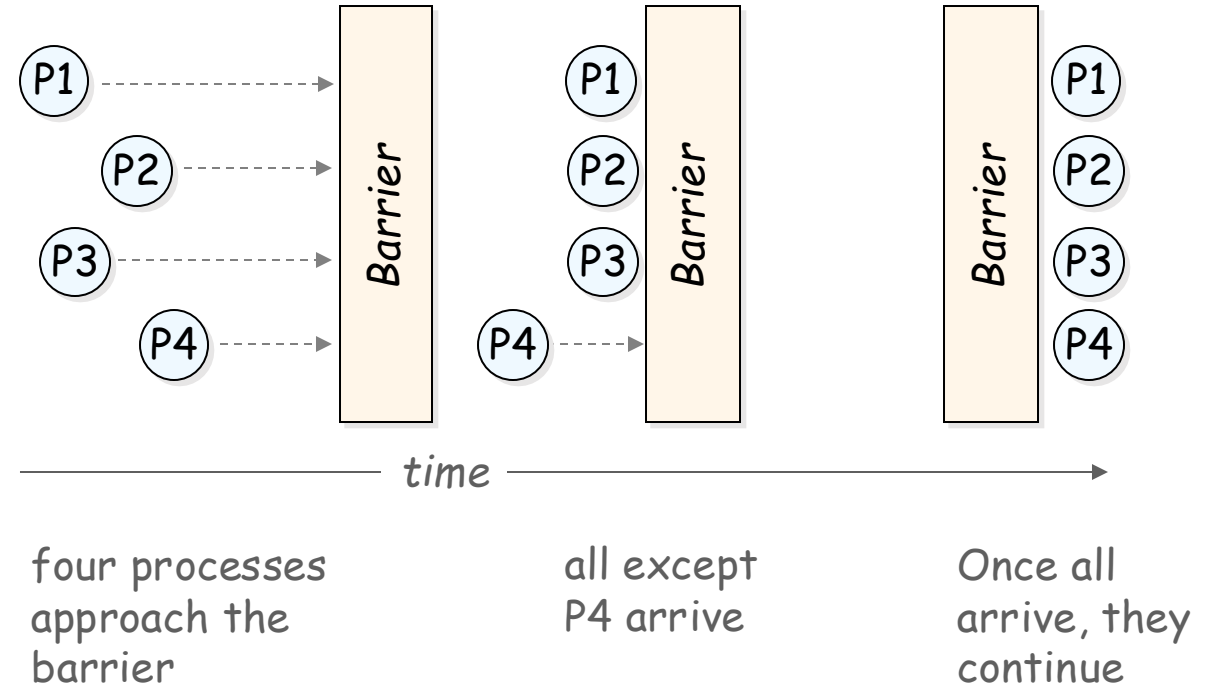


What is a Barrier ?

- *Coordination mechanism (algorithm)*
- *forces processes/threads to wait until each one of them has reached a certain point.*
- *Once all the processes/threads reach barrier, they all can pass the barrier.*

What is a Barrier ?

- Coordination mechanism (algorithm)
- forces processes/threads to wait until each one of them has reached a certain point.
- Once all the processes/threads reach barrier, they all can pass the barrier.



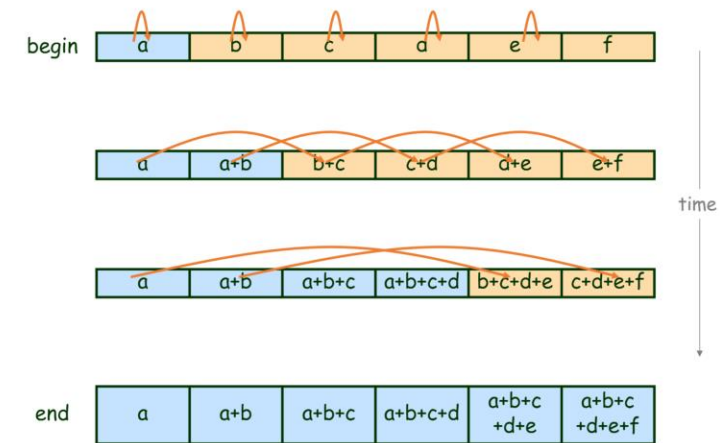
Pthreads and barriers

- Type `pthread_barrier_t`

```
int pthread_barrier_init(pthread_barrier_t *barrier,  
                        const pthread_barrierattr_t *attr,  
                        unsigned count);  
int pthread_barrier_destroy(pthread_barrier_t *barrier);  
int pthread_barrier_wait(pthread_barrier_t *barrier);
```

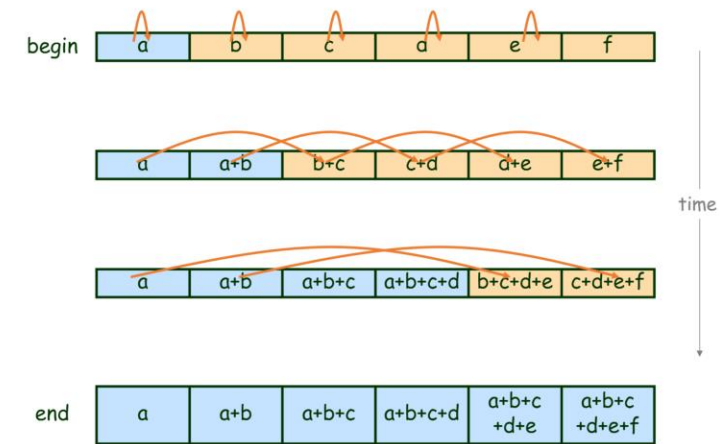

Pthreads Parallel Prefix Sum

```
pthread_barrier_t g_barrier;  
pthread_mutex_t g_locks[N];  
int g_values[N] = { a, b, c, d, e, f };  
  
void init_stuff() {  
    ...  
    pthread_barrier_init(&g_barrier, NULL, N-1);  
}  
  
void prefix_sum_thread(void * param) {  
  
    int i;  
    int id = *((int*)param);  
    int stride = 0;  
  
    for(stride=1; stride<=N/2; stride<<1) {  
  
        pthread_mutex_lock(&g_locks[id]);  
        pthread_mutex_lock(&g_locks[id+stride]);  
        g_values[id+stride] += g_values[id];  
        pthread_mutex_unlock(&g_locks[id]);  
        pthread_mutex_unlock(&g_locks[id+stride]);  
  
        pthread_barrier_wait(&g_barrier);  
  
    }  
}
```



Pthreads Parallel Prefix Sum

```
pthread_barrier_t g_barrier;  
pthread_mutex_t g_locks[N];  
int g_values[N] = { a, b, c, d, e, f };  
  
void init_stuff() {  
    ...  
    pthread_barrier_init(&g_barrier, NULL, N-1);  
}  
  
void prefix_sum_thread(void * param) {  
  
    int i;  
    int id = *((int*)param);  
    int stride = 0;  
  
    for(stride=1; stride<=N/2; stride<<1) {  
  
        pthread_mutex_lock(&g_locks[id]);  
        pthread_mutex_lock(&g_locks[id+stride]);  
        g_values[id+stride] += g_values[id];  
        pthread_mutex_unlock(&g_locks[id]);  
        pthread_mutex_unlock(&g_locks[id+stride]);  
  
        pthread_barrier_wait(&g_barrier);  
  
    }  
}
```



fixed?

Barrier Goals

Ideal barrier properties:

Barrier Goals

Ideal barrier properties:

- Low shared memory space complexity
- Low contention on shared objects
- Low shared memory references per process
- No need for shared memory initialization
- Symmetric-ness (same amount of work for all processes)
- Algorithm simplicity
- Simple basic primitive
- Minimal propagation time
- Reusability of the barrier (must!)

Barrier Building Blocks

- Semaphores
- Atomic Bit
- Atomic Register
- Fetch-and-increment register
- Test and set bits
- Read-Modify-Write register

Barrier with Semaphores



Barrier using Semaphores

Algorithm for n processes

shared arrival: binary semaphore, initially 1
departure: binary semaphore, initially 0
counter: atomic register ranges over $\{0, \dots, n\}$, initially 0

Barrier using Semaphores

Algorithm for n processes

shared arrival: binary semaphore, initially 1
departure: binary semaphore, initially 0
counter: atomic register ranges over {0, ..., n}, initially 0

```
1  sem_wait(arrival)
2  counter := counter + 1      // atomic register
3  if counter < n then sem_post(arrival) else sem_post(departure)
4  sem_wait(departure)
5  counter := counter - 1
6  if counter > 0 then sem_post(departure) else sem_post(arrival)
```


Barrier using Semaphores

Algorithm for n processes

shared arrival: binary semaphore, initially 1
departure: binary semaphore, initially 0
counter: atomic register ranges over {0, ..., n}, initially 0

```
1  sem_wait(arrival)
2  counter := counter + 1      // atomic register
3  if counter < n then sem_post(arrival) else sem_post(departure)
4  sem_wait(departure)
5  counter := counter - 1
6  if counter > 0 then sem_post(departure) else sem_post(arrival)
```

Question:

Would this barrier be correct if the shared counter won't be an **atomic** register?

Barrier using Semaphores

Properties

- Pros:

- Cons:

Barrier using Semaphores

Properties

- **Pros:**

- Very Simple
- Space complexity $O(1)$
- Symmetric

- **Cons:**

Barrier using Semaphores

Properties

- **Pros:**

- Very Simple
- Space complexity $O(1)$
- Symmetric

- **Cons:**

- Required a strong object
 - Requires some central manager
 - High contention on the semaphores
- Propagation delay $O(n)$

Questions?