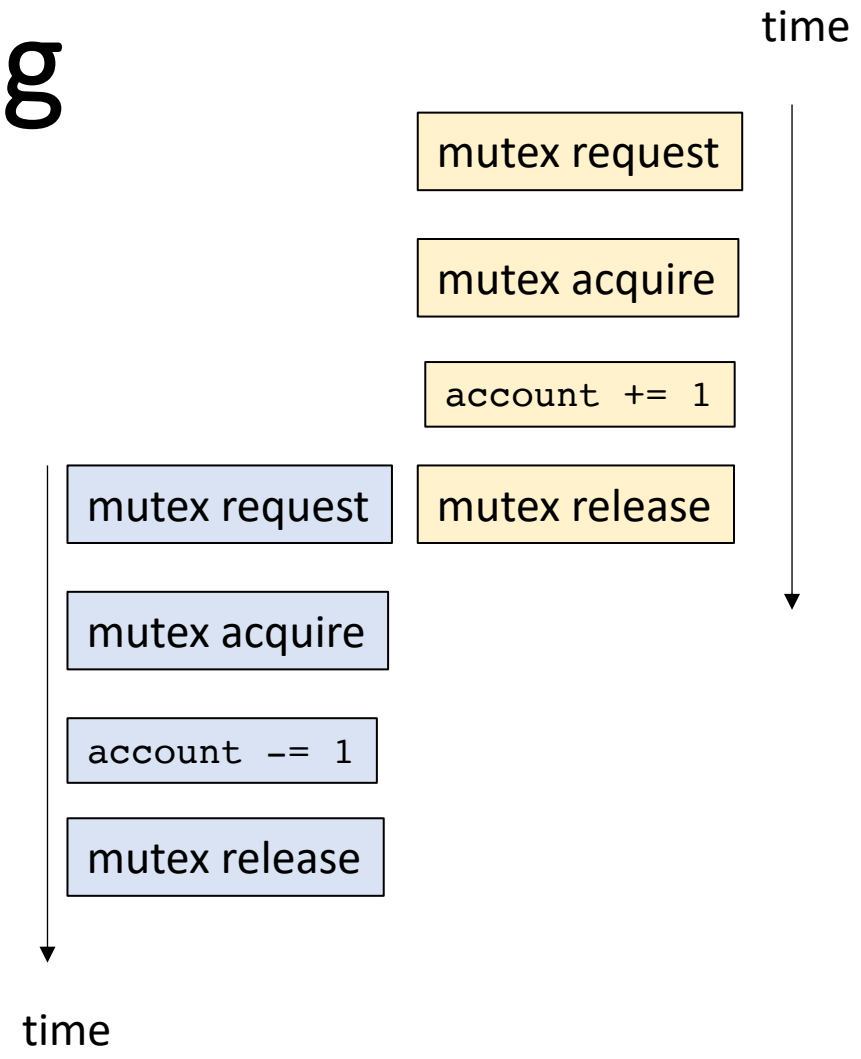
CSE113: Parallel Programming

April 15, 2021

- **Topic**: Mutual Exclusion Continued
  - Multiple Mutexes
  - Implementing Mutexes
    - Atomic instructions
    - 2-threaded mutex
    - N-threaded mutex
    - Fair mutex



Announcements

- Reese's first class ☺
  - He can tell you more about mutex implementations on GPUs
  - He has a very special announcement (Piazza)
- Homework 1 is posted:
  - Due April 22
- My next office hours are on Wednesday, 3 - 5 PM
  - TAs have office hours daily
  - They are more helpful with tool flows (docker, VSCode)
  - My last office hours before assignment 1 is due!

Homework

- Your first concern is correctness
  - speedups mean nothing if the result is incorrect!

- what sort of speedups have people seen?
  - It will change based on your CPU, compiler and system!
  - Different pipelines, super scaler, OS has different schedulers

- my speeds: ~6.5x for part 1. ~3.2x for part2

- report does not require too much detail!

Quiz

- Open Quiz for 3 minutes

Quiz

- Open Quiz for 3 minutes
- Go over quiz answers

Quick Performance Consideration

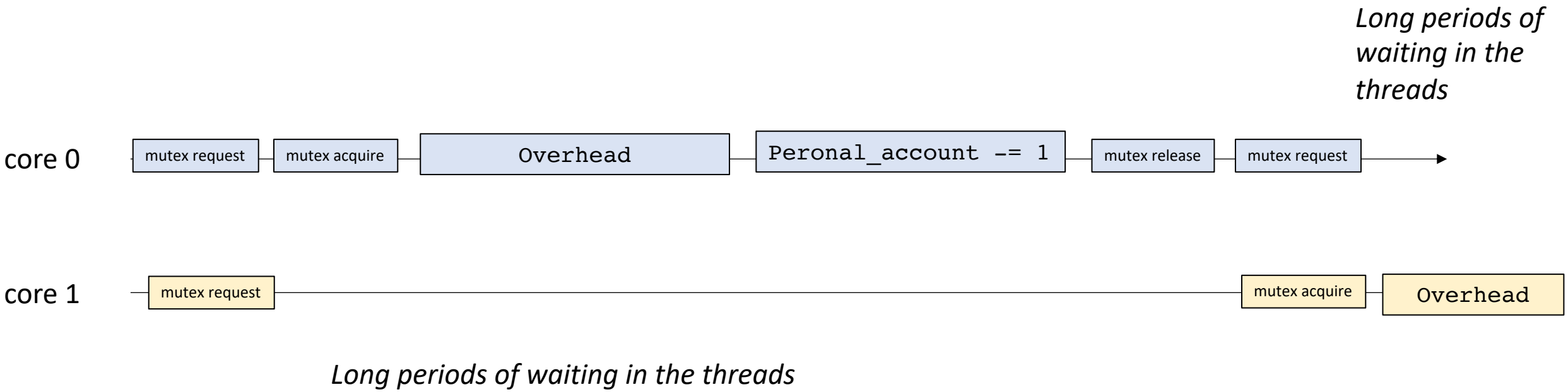
Today isn’t about performance, but try to keep mutual exclusion sections small! Protect only data conflicts!

Code example with overhead

Performance consideration

Today isn’t about performance, but try to keep mutual exclusion sections small! Protect only data conflicts!

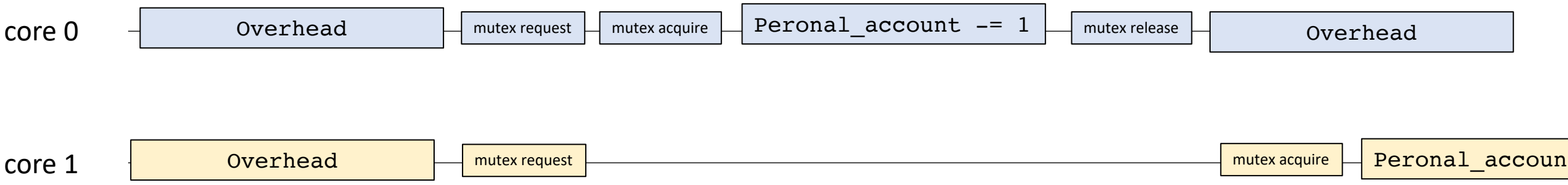
Code example with overhead



Performance consideration

Today isn’t about performance, but try to keep mutual exclusion sections small! Protect only data conflicts!

Code example with overhead



overlap the overhead (i.e. computation without any data conflicts)

Lecture Schedule

- Multiple Mutexes
- Lock-free accounts
- Implementing Mutexes
  - Atomic instructions
  - 2-threaded mutex
- Intro to performance

Lecture Schedule

- **Multiple Mutexes**
- Lock-free accounts
- Implementing Mutexes
  - Atomic instructions
  - 2-threaded mutex
- Intro to performance

Lecture Schedule

- **Multiple Mutexes**

- Implementing Mutexes
  - Atomic instructions
  - 2-threaded mutex

- Introduction to Mutex performance

Multiple mutexes

Lets say I have two accounts:

- Business account
- Personal account

- Need to protect both of them using a mutex
  - Easy, we can just the same mutex
  - Show implementation

Multiple mutexes

Lets say I have two accounts:

- Business account
- Personal account

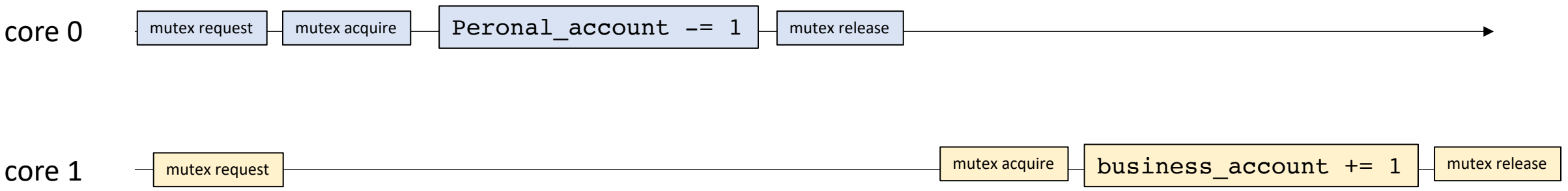
- No reason individual accounts can’t be accessed in parallel

Multiple mutexes

Lets say I have two accounts:

- Business account
- Personal account

- No reason individual accounts can’t be accessed in parallel



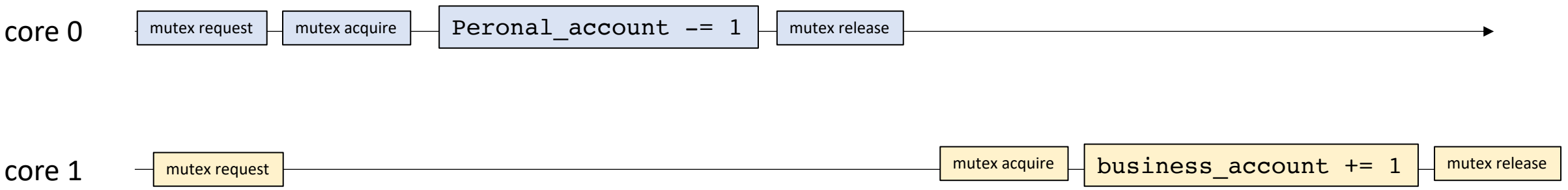
Long periods of waiting in the threads

Multiple mutexes

Mutexes are objects. We can create multiple versions of them to protect different shared data.

`MutexP` for personal account
`MutexB` for business account

Critical sections across different mutexes can overlap

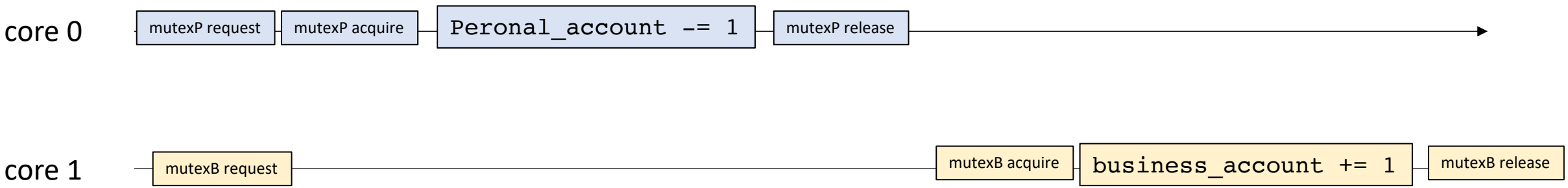


Multiple mutexes

Mutexes are objects. We can create multiple versions of them to protect different shared data.

`MutexP` for personal account
`MutexB` for business account

Critical sections across different mutexes can overlap

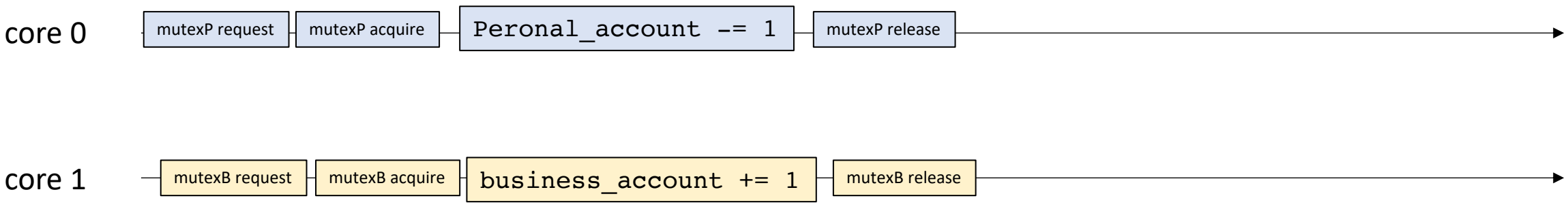


Multiple mutexes

Mutexes are objects. We can create multiple versions of them to protect different shared data.

`MutexP` for personal account
`MutexB` for business account

Critical sections across different mutexes can overlap

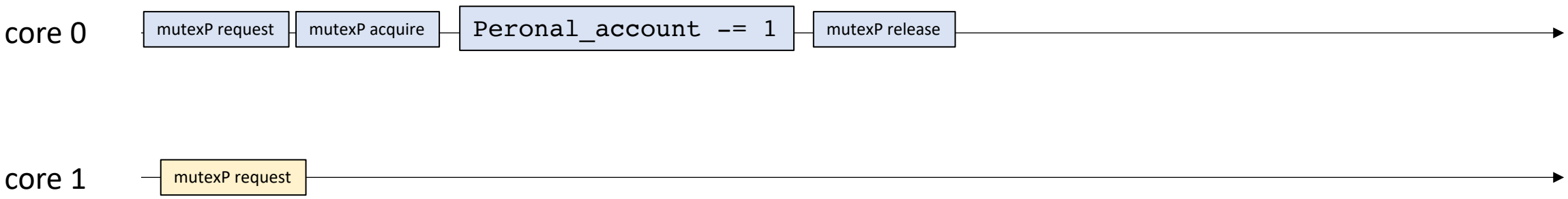


Multiple mutexes

Mutexes are objects. We can create multiple versions of them to protect different shared data.

`MutexP` for personal account
`MutexB` for business account

Critical sections across different mutexes can overlap

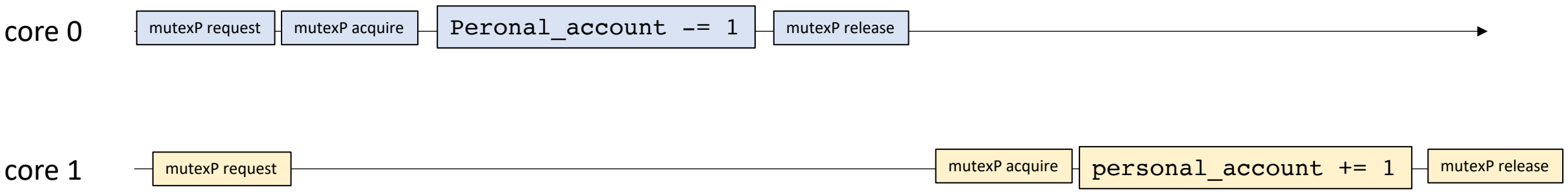


Multiple mutexes

Mutexes are objects. We can create multiple versions of them to protect different shared data.

`MutexP` for personal account
`MutexB` for business account

Critical sections across different mutexes can overlap



Multiple mutexes

Mutexes are objects. We can create multiple versions of them to protect different shared data.

`MutexP` for personal account
`MutexB` for business account

Critical sections across different mutexes can overlap

Code example

Managing multiple mutexes

Consider 2 memory locations: x and y.

Consider 3 functions that are executing concurrently

```
// reads/writes to x
void foo(int *x);
```

```
// reads/writes to y
void bar(int *y);
```

```
// reads/writes to x and y
void gaz(int *x, int *y);
```

Managing multiple mutexes

Consider 2 memory locations: x and y.

Consider 3 functions that are executing concurrently

solution: use 1 global mutex: g_mutex

```
// reads/writes to x
void foo(int *x) {
 g_mutex.lock;
 // operate on x
 g_mutex.unlock;
}
```

```
// reads/writes to y
void bar(int *y) {
 g_mutex.lock;
 // operate on y
 g_mutex.unlock;
}
```

```
// reads/writes to x and y
void gaz(int *x, int *y) {
 g_mutex.lock;
 // operate on x and y
 g_mutex.unlock;
}
```

Managing multiple mutexes

Consider 2 memory locations: x and y.

Consider 3 functions that are executing concurrently

solution: use 1 global mutex: g_mutex

issue: none of these functions can execute in parallel!

```
// reads/writes to x
void foo(int *x) {
 g_mutex.lock;
 // operate on x
 g_mutex.unlock;
}
```

```
// reads/writes to y
void bar(int *y) {
 g_mutex.lock;
 // operate on y
 g_mutex.unlock;
}
```

```
// reads/writes to x and y
void gaz(int *x, int *y) {
 g_mutex.lock;
 // operate on x and y
 g_mutex.unlock;
}
```

Managing multiple mutexes

Consider 2 memory locations: x and y.

Consider 3 functions that are executing concurrently

now foo and bar can execute in parallel!

added complexity though

A higher performant solution: multiple mutexes for the data you access: `x_mutex, y_mutex`

```
// reads/writes to x
void foo(int *x) {
 x_mutex.lock;
 // operate on x
 x_mutex.unlock;
}
```

```
// reads/writes to y
void bar(int *y) {
 y_mutex.lock;
 // operate on y
 y_mutex.unlock;
}
```

```
// reads/writes to x and y
void gaz(int *x, int *y) {
 x_mutex.lock;
 y_mutex.lock;
 // operate on x and y
 x_mutex.unlock;
 y_mutex.unlock;
}
```

Managing multiple mutexes

Consider this increasingly elaborate scheme

My accounts start being audited by two agents:

- UCSC
- IRS

- They need to examine the accounts at the same time. They need to acquire both locks.

Managing multiple mutexes

Consider this increasingly elaborate scheme

My accounts start being audited by two agents:

- UCSC
- IRS

- **Code example**

Multiple mutexes

- Our program deadlocked! What happened?

Multiple mutexes

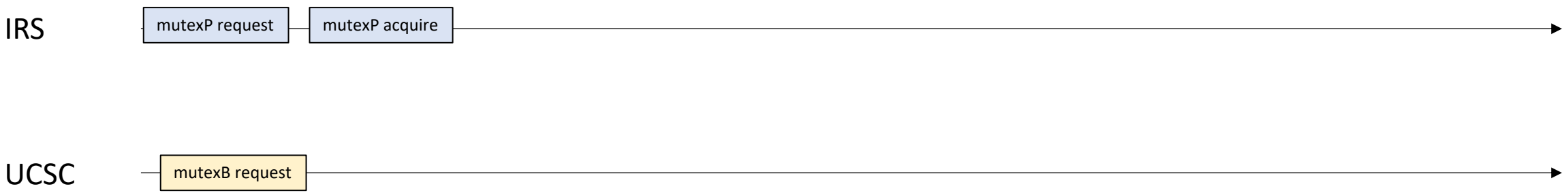
- Our program deadlocked! What happened?

Multiple mutexes

- Our program deadlocked! What happened?

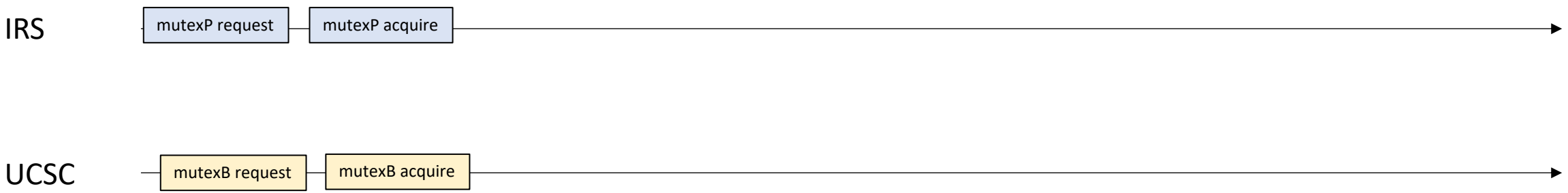
Multiple mutexes

- Our program deadlocked! What happened?



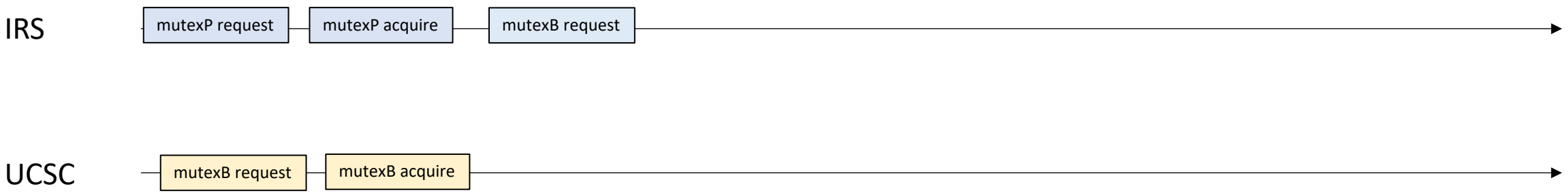
Multiple mutexes

- Our program deadlocked! What happened?



Multiple mutexes

- Our program deadlocked! What happened?



Multiple mutexes

- Our program deadlocked! What happened?

Multiple mutexes

- Our program deadlocked! What happened?

IRS has the personal mutex and won't release it until it acquires the business mutex.
UCSC has the business mutex and won't release it until it acquires the personal mutex.

This is called a deadlock!

Multiple mutexes

- Our program deadlocked! What happened?
- Fix: Acquire mutexes in the same order
- Proof sketch by contradiction
  - Thread 0 is holding mutex X waiting for mutex Y
  - Thread 1 is holding mutex Y waiting for mutex X

Assume the order that you acquire mutexes is X then Y
Thread 1 cannot hold mutex Y without holding mutex X.
Thread 1 cannot hold mutex X because thread 0 is holding mutex X
Thus the deadlock cannot occur

Multiple mutexes

- Our program deadlocked! What happened?
- Fix: Acquire mutexes in the same order
- Proof sketch by contradiction
  - Thread 0 is holding mutex X waiting for mutex Y
  - Thread 1 is holding mutex Y waiting for mutex X

Double check with testing

Assume the order that you acquire mutexes is X then Y
Thread 1 cannot hold mutex Y without holding mutex X.
Thread 1 cannot hold mutex X because thread 0 is holding mutex X
Thus the deadlock cannot occur

Programming with mutexes is tricky!

make sure all data conflicts are protected with a mutex

keep critical sections small

balance between having many mutexes (provides performance) but gives the potential for deadlocks

But its better than the alternative - reasoning about data conflicts.

Lecture Schedule

- Multiple Mutexes
- **Lock-free accounts**
- Implementing Mutexes
  - Atomic instructions
  - 2-threaded mutex
- Intro to performance

Atomic RMWs

Other ways to implement accounts?

Atomic Read-modify-write (RMWs): primitive instructions that implement a read event, modify event, and write event indivisibly, i.e. it cannot be interleaved.

```
atomic_fetch_add(atomic_int * addr, int value) {
   int tmp = *addr; // read
   tmp += value;    // modify
   *addr = tmp;     // write
}
```

other operations: max, min, etc.

Modify these programs to use atomic RMWs

Tyler's coffee addiction:

```
m.lock();
tylers_account -= 1;
m.unlock();
```

time

Tyler's employer

```
m.lock();
tylers_account += 1;
m.unlock();
```

time

Modify these programs to use atomic RMWs

Tyler's coffee addiction:

```
~~m.lock();~~
tylers_account -= 1;
~~m.unlock();~~
```

Tyler's employer

```
~~m.lock();~~
tylers_account += 1;
~~m.unlock();~~
```

Modify these programs to use atomic RMWs

Tyler's coffee addiction:

```
tylers_account -= 1;
```

time

Tyler's employer

```
tylers_account += 1;
```

time

Modify these programs to use atomic RMWs

Tyler's coffee addiction:

```
atomic_fetch_add(&tylers_account, -1);
```

time

Tyler's employer

```
atomic_fetch_add(&tylers_account, 1);
```

time

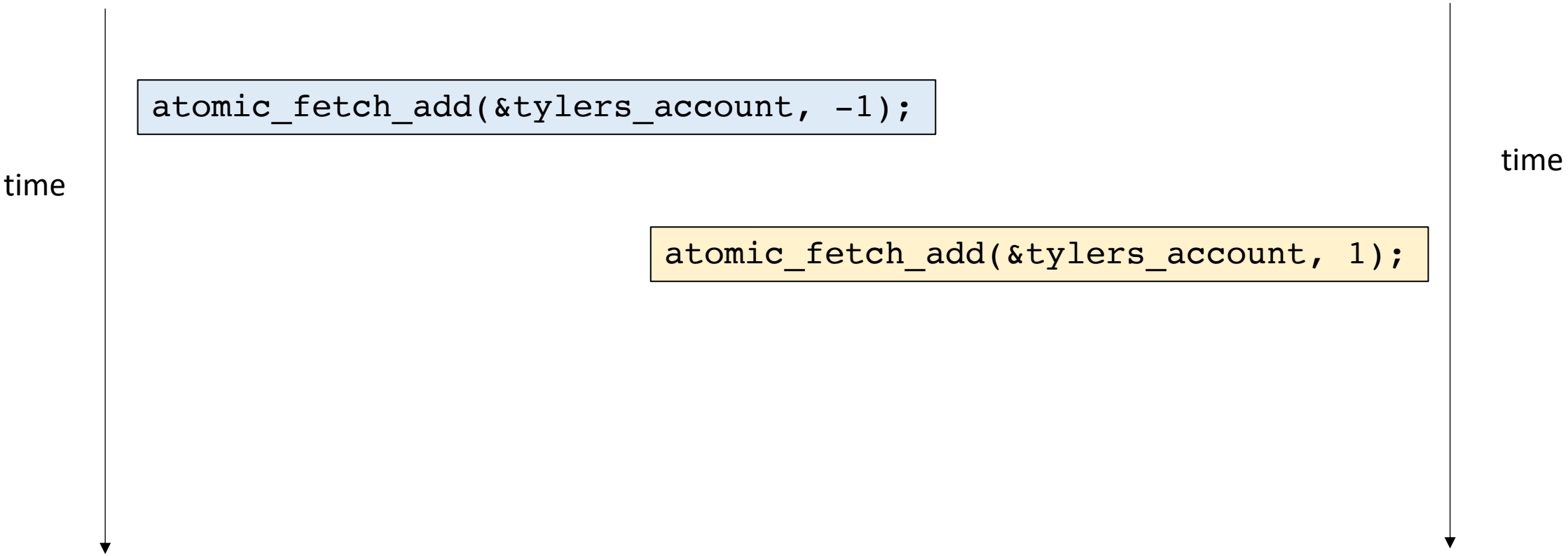
Modify these programs to use atomic RMWs

Tyler's coffee addiction:

```
atomic_fetch_add(&tylers_account, -1);
```

Tyler's employer

```
atomic_fetch_add(&tylers_account, 1);
```



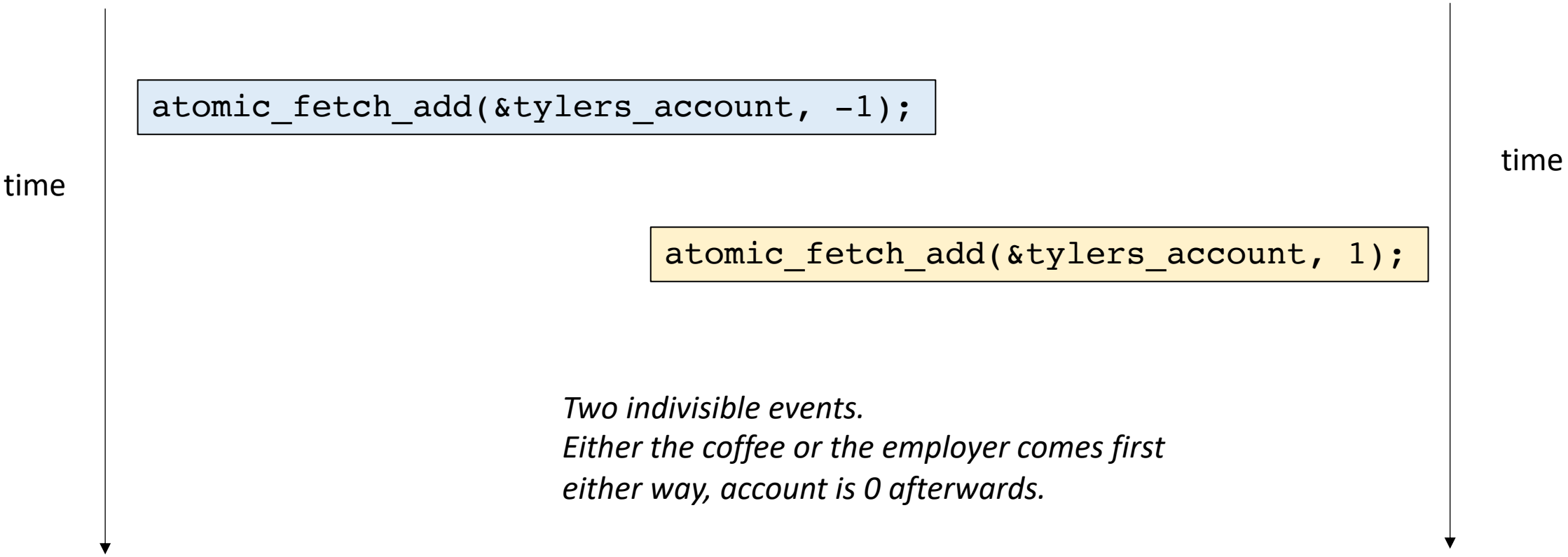
Modify these programs to use atomic RMWs

Tyler's coffee addiction:

```
atomic_fetch_add(&tylers_account, -1);
```

Tyler's employer

```
atomic_fetch_add(&tylers_account, 1);
```



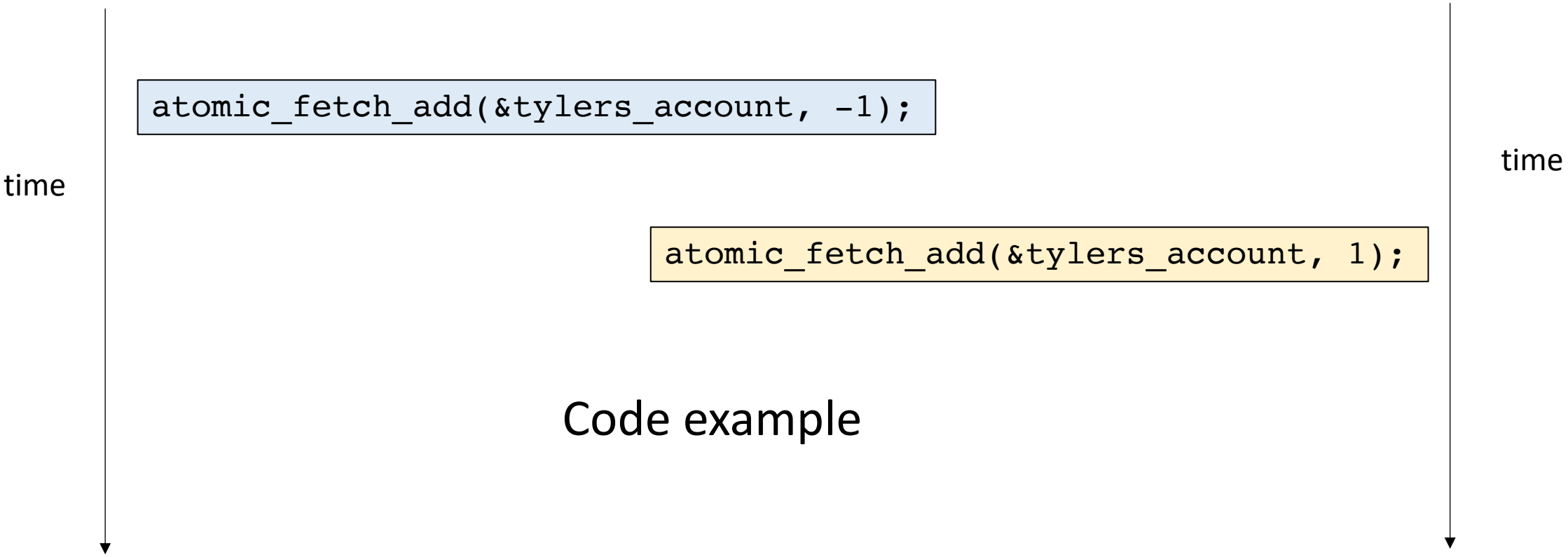
Modify these programs to use atomic RMWs

Tyler's coffee addiction:

```
atomic_fetch_add(&tylers_account, -1);
```

Tyler's employer

```
atomic_fetch_add(&tylers_account, 1);
```



Code example

Atomic RMWs

Pros? Cons?

Atomic RMWs

Pros? Cons?

Not all architectures support RMWs (although more common with C++11)

Limits critical section (what if account needs additional updating?)

atomic types need to propagate through the entire application

Lecture Schedule

- Multiple Mutexes
- Lock-free accounts
- **Implementing Mutexes**
  - Atomic instructions
  - 2-threaded mutex
- Intro to performance

Lecture Schedule

- Multiple Mutexes
- Lock-free accounts
- **Implementing Mutexes**
  - **Atomic instructions**
  - 2-threaded mutex
- Intro to performance

Mutex Implementations

- A mutex is not a primitive data structure! (built out of primitives)
  - think back to your data structure class
  - Stacks and queues are not primitives, they have an API and we implement their API using primitives: arrays, int, etc.

- While C++ has a fine mutex, we want to learn how to implement our own.
  - Why?

Building blocks

- Memory reads and memory writes
  - later: read-modify-writes
- We need to guarantee that our reads and writes actually go to memory.
  - And other properties we will see soon
- To do this, we will use C++ atomic operations

A historical perspective

- Adding concurrency support to a programming language is hard!
- The memory model defines how threads can safely share memory

- Java tried to do this,

wikipedia

> The original Java memory model, developed in 1995, was widely perceived as broken, preventing many runtime optimizations and not providing strong enough guarantees for code safety. It was updated through the Java Community Process, as Java Specification Request 133 (JSR-133), which took effect in 2004, for Tiger (Java 5.0).[1][2]

Brian Goetz (2019)

> It is worth noting that broken techniques like double-checked locking are still broken under the new memory model, a

A historical perspective

- How is C++?

- Has issues (imprecise, not modular)
  - but at least considered safe
  - Specification makes it difficult to reason about all programs
  - Open problem!

- Luckily mutexes (and their implementations) avoid the problematic areas of the language!

Our primitive instructions

- Types: atomic_
- Interface (C++ might provide overloaded operators):
  - load
  - store
- Properties:
  - loads and stores will always go to memory.
  - compiler memory fence
  - hardware memory fence

Atomic properties

- loads and stores will always go to memory
- Compiler example, performance difference

Atomic properties

- loads and stores will always go to memory
- Compiler example, performance difference
- Compiler makes reasoning about parallel code hard, but big performance improvements:
  - O(2048) vs. O(1)

Atomic properties

- Compiler Fence

- Compiler can be aggressive with memory operations:
  - For non-atomic memory locations, the following optimizations are valid

Atomic properties

- Compiler Fence

- Compiler can be aggressive with memory operations:
  - For non-atomic memory locations, the following optimizations are valid

```
a[i] = 0;
a[i] = 1;
```

can be optimized to:

```
a[i] = 1;
```

Atomic properties

- Compiler Fence

- Compiler can be aggressive with memory operations:
  - For non-atomic memory locations, the following optimizations are valid

```
a[i] = 0;
a[i] = 1;
```

can be optimized to:

```
a[i] = 1;
```

```
x = a[i];
x2 = a[i];
```

can be optimized to:

```
x = a[i];
x2 = x;
```

Atomic properties

- Compiler Fence

- Compiler can be aggressive with memory operations:
  - For non-atomic memory locations, the following optimizations are valid

```
a[i] = 0;
a[i] = 1;
```

can be optimized to:

```
a[i] = 1;
```

```
x = a[i];
x2 = a[i];
```

can be optimized to:

```
x = a[i];
x2 = x;
```

```
a[i] = 6;
x = a[i];
```

can be optimized to:

```
x = 6;
```

Atomic properties

- Compiler Fence

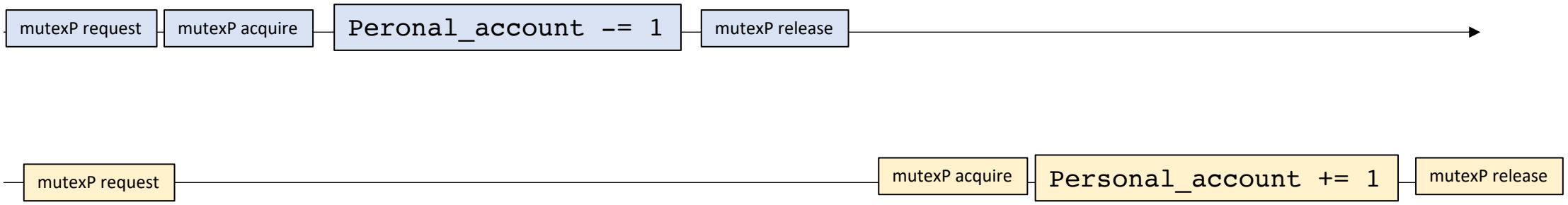
- Compiler can be aggressive with memory operations:
  - For non-atomic memory locations, the following optimizations are valid

- And many others... especially when you consider mixing with other optimizations
  - Very difficult to understand when/where memory accesses will actually occur in your code

Atomic properties

- Compiler Fence

Compiler cannot keep personal_account
in a register past the mutex

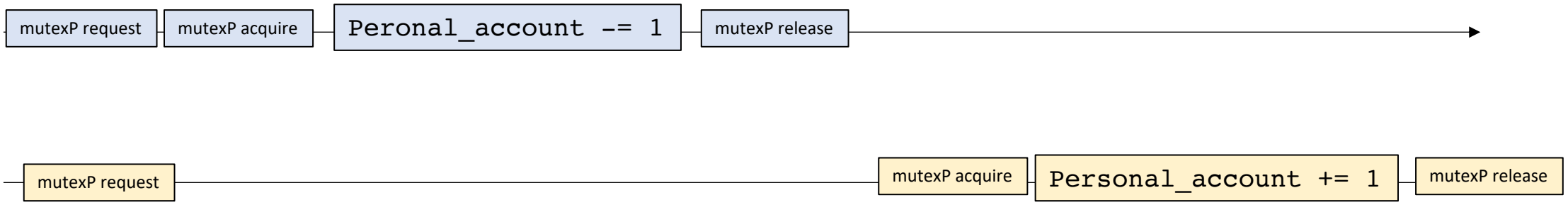


because this thread needs to see the
updated view

Atomic properties

- Compiler Fence

what can go wrong if the compiler doesn't write values to memory?

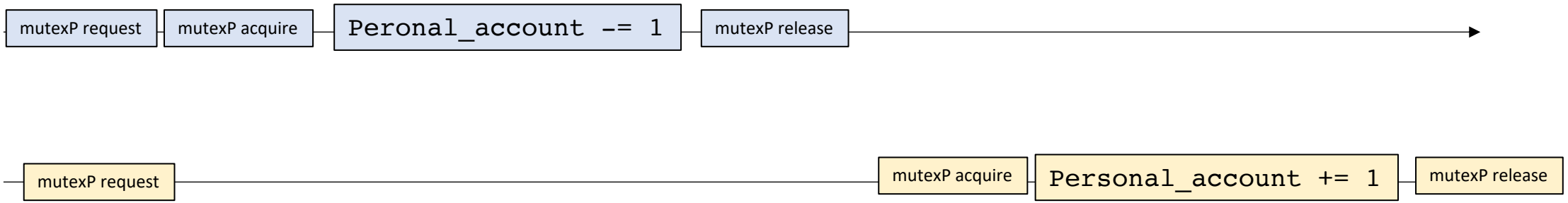


Atomic properties

- Compiler Fence

what can go wrong if the compiler doesn't write values to memory?

initially personal_account is 0



Atomic properties

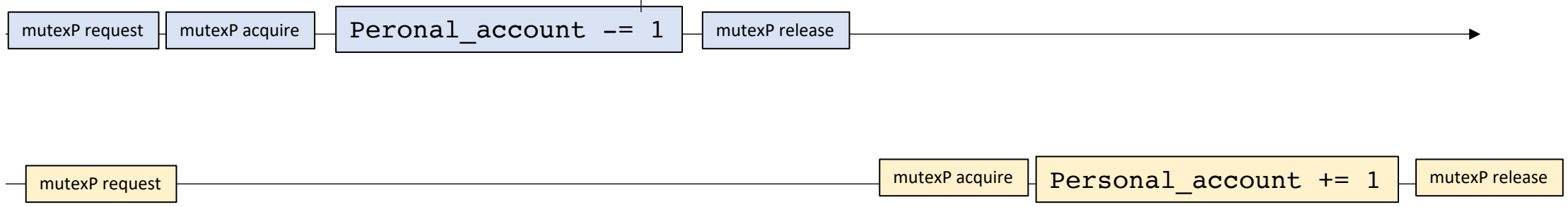
- Compiler Fence

what can go wrong if the compiler doesn't write values to memory?

initially personal_account is 0

loads 0

reg = *personal_account - 1;



Atomic properties

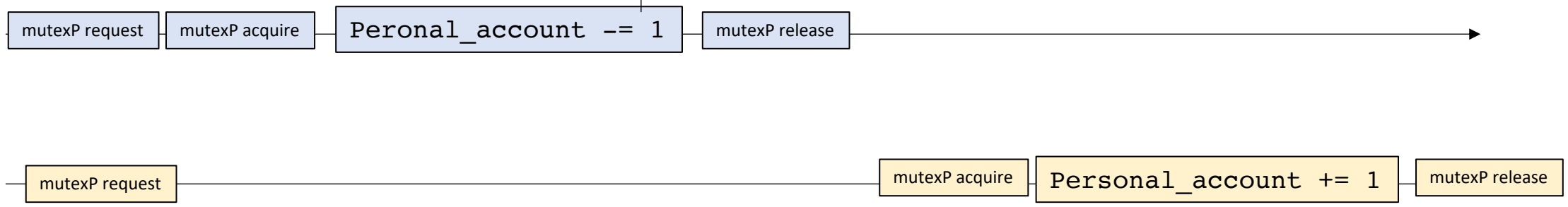
- Compiler Fence

what can go wrong if the compiler doesn't write values to memory?

initially personal_account is 0

loads 0

reg = *personal_account - 1;

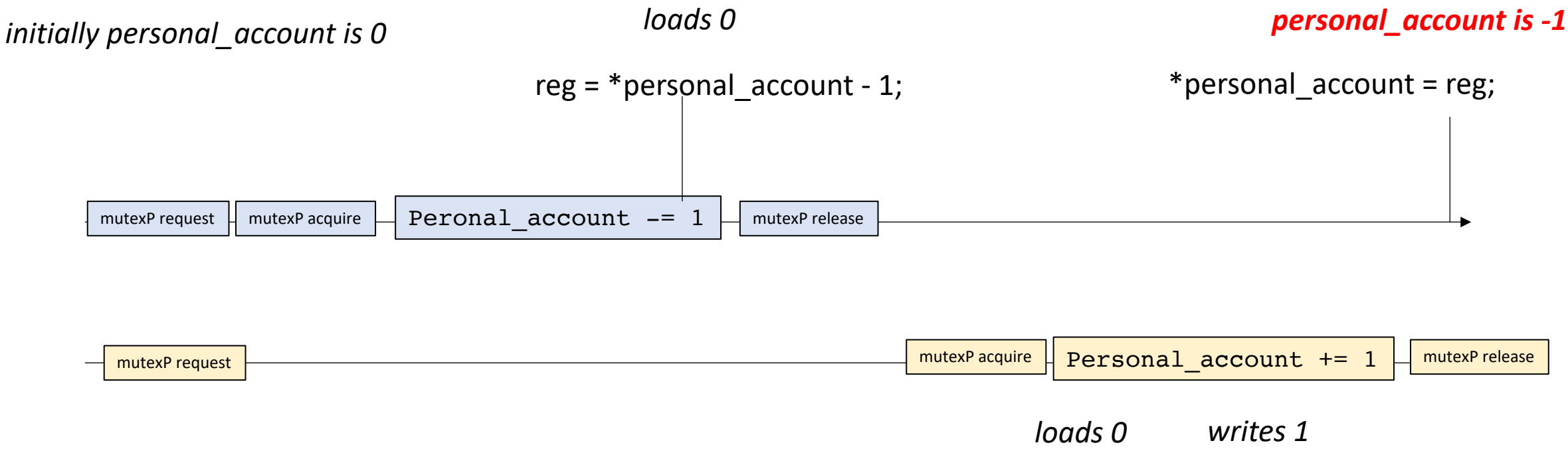


loads 0 *writes 1*

Atomic properties

- Compiler Fence

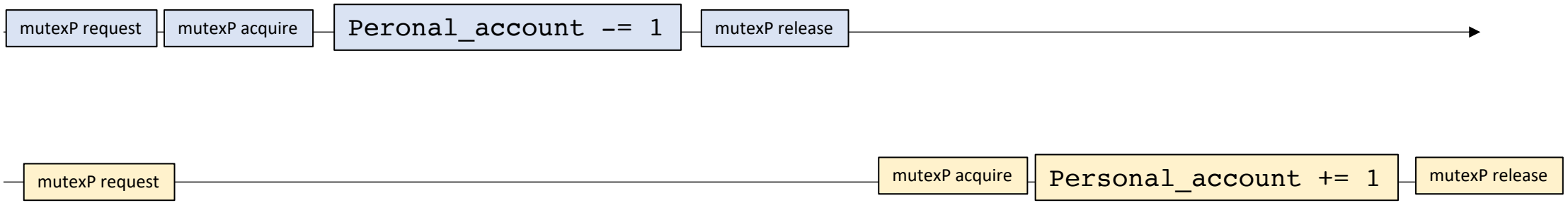
what can go wrong if the compiler doesn't write values to memory?



Atomic properties

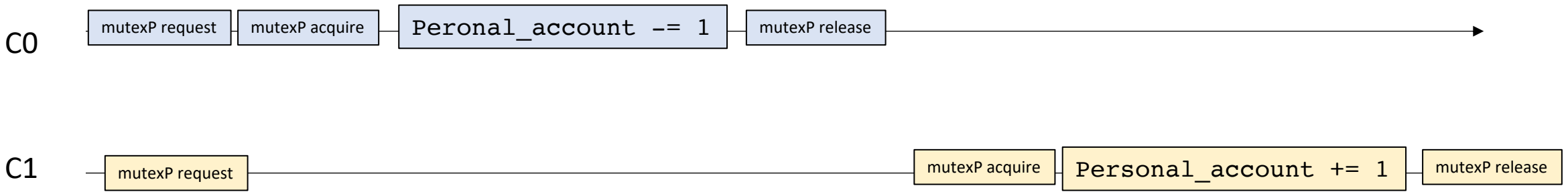
- Compiler Fence

compiler example

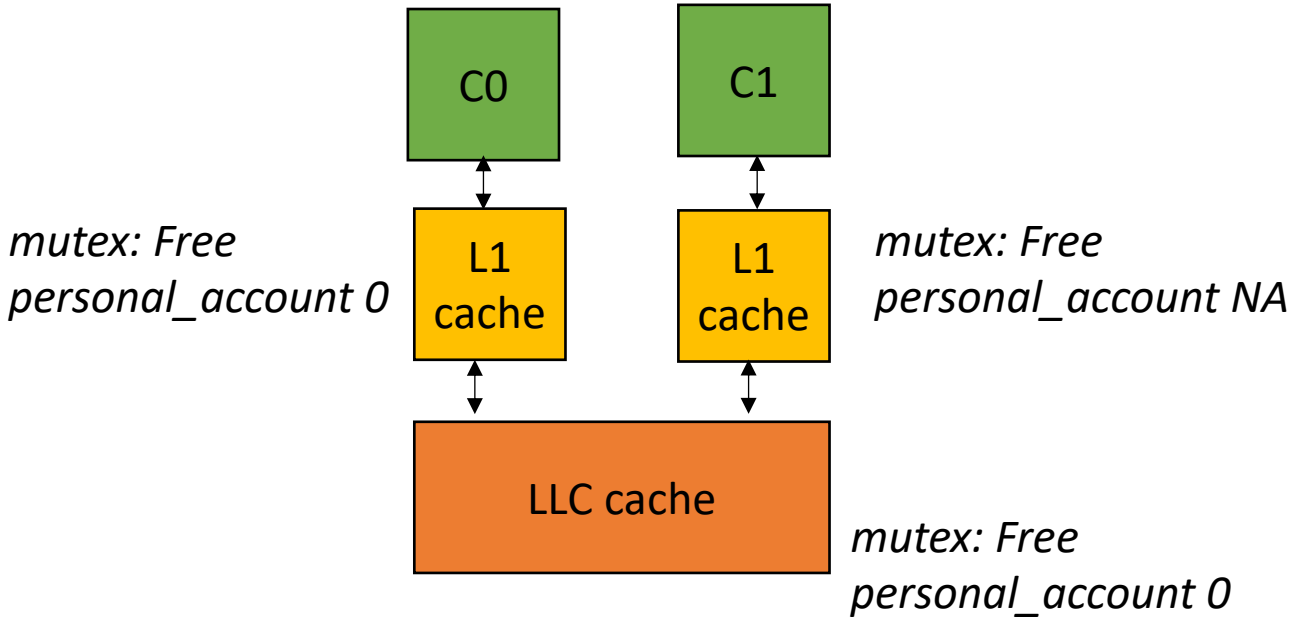


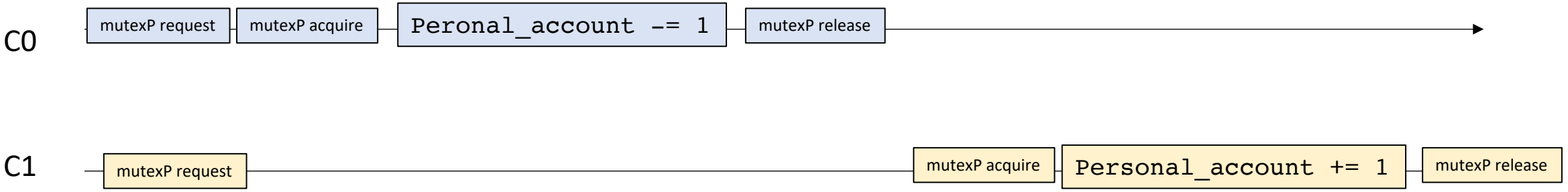
- Memory Fence (or Memory Barrier)

Compiler example: `dmb` for ARM

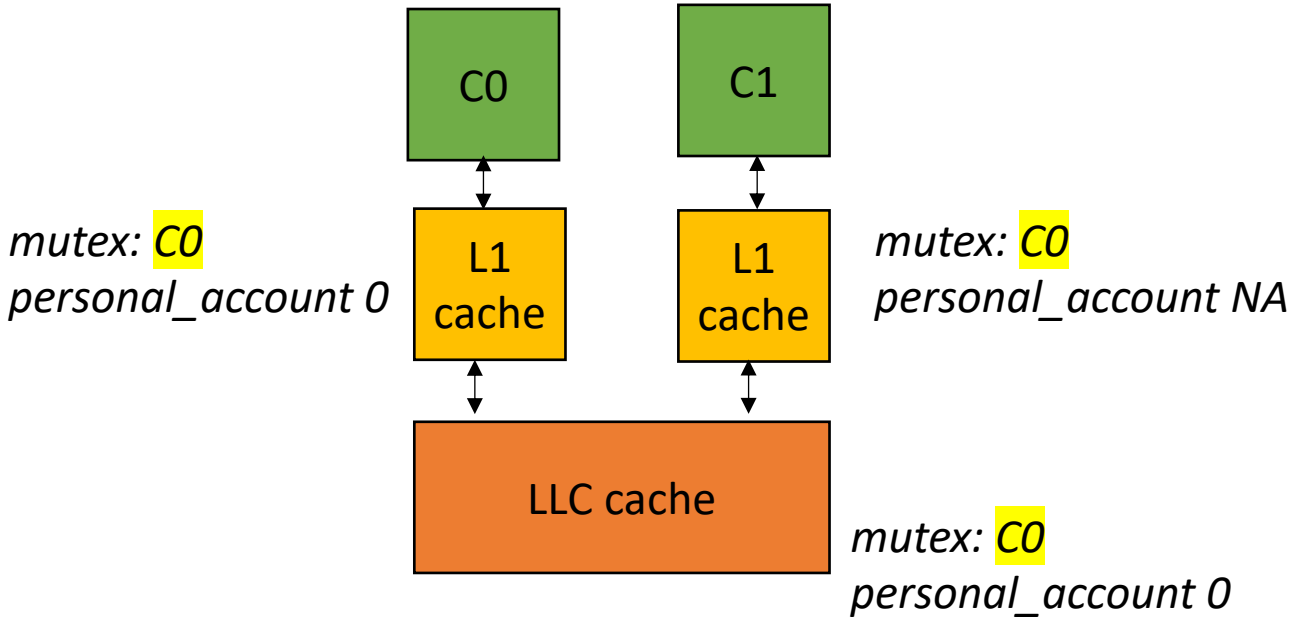


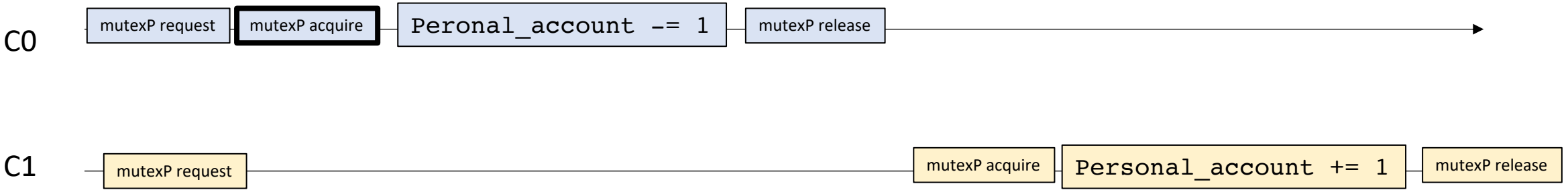
- Memory Fence (or Memory Barrier)



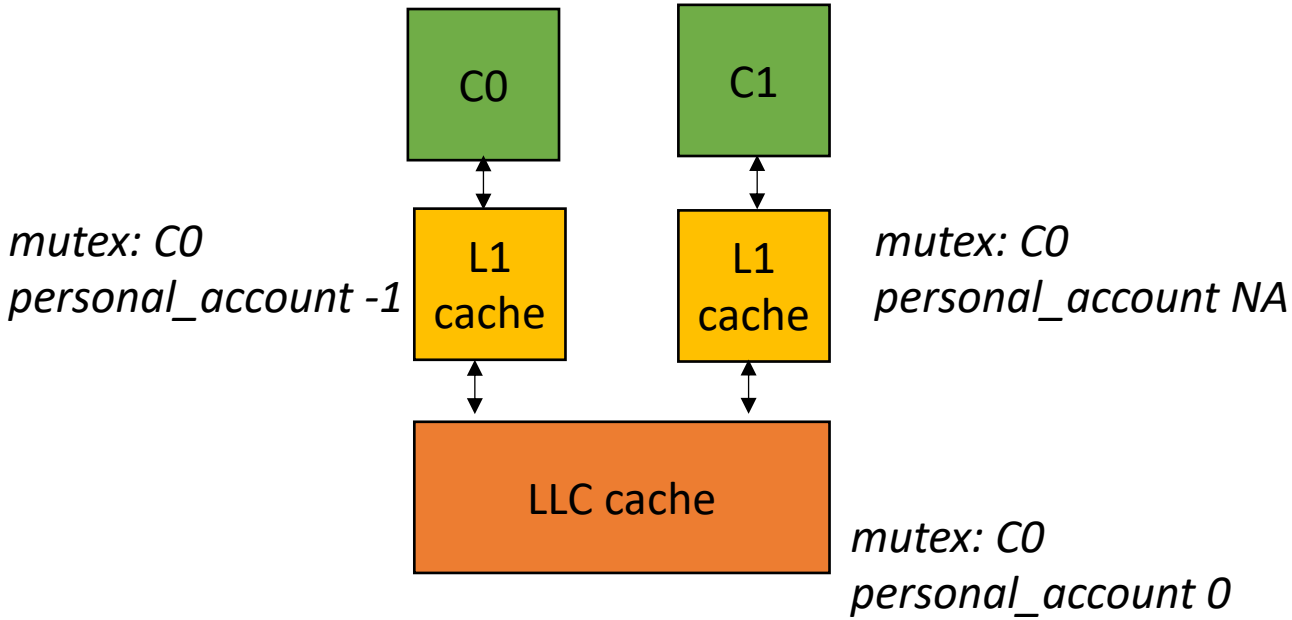


- Memory Fence (or Memory Barrier)

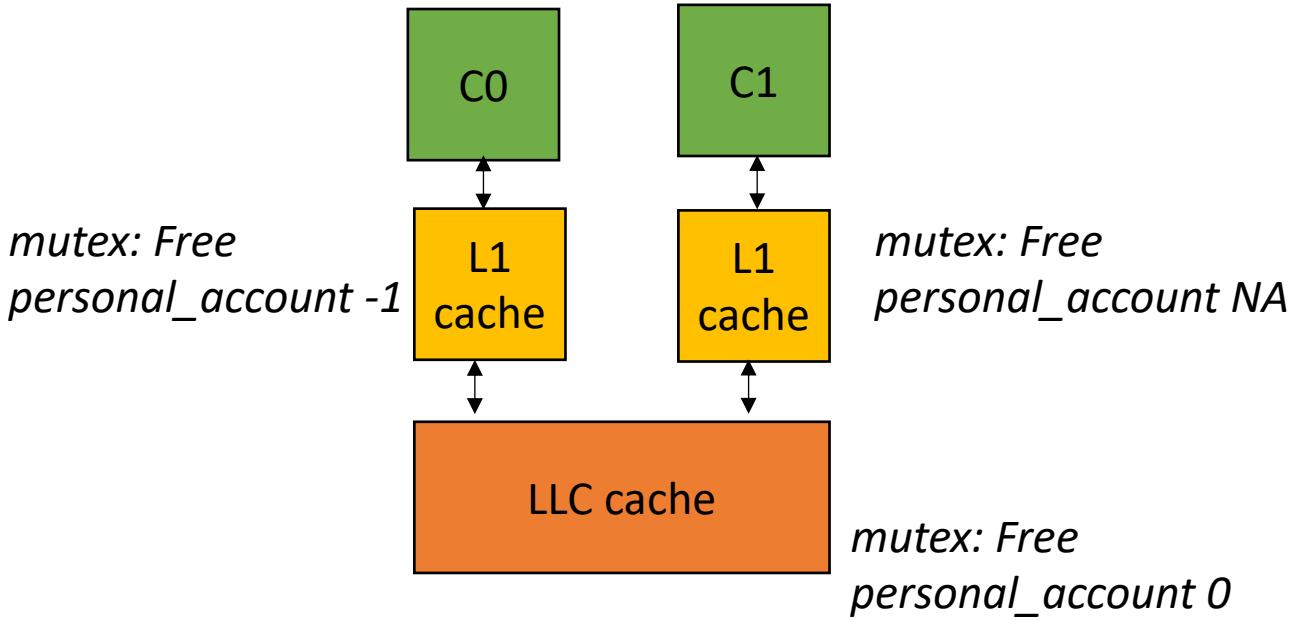


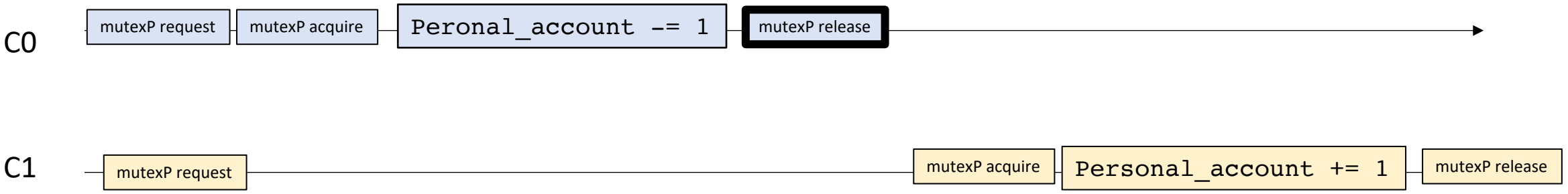


- Memory Fence (or Memory Barrier)

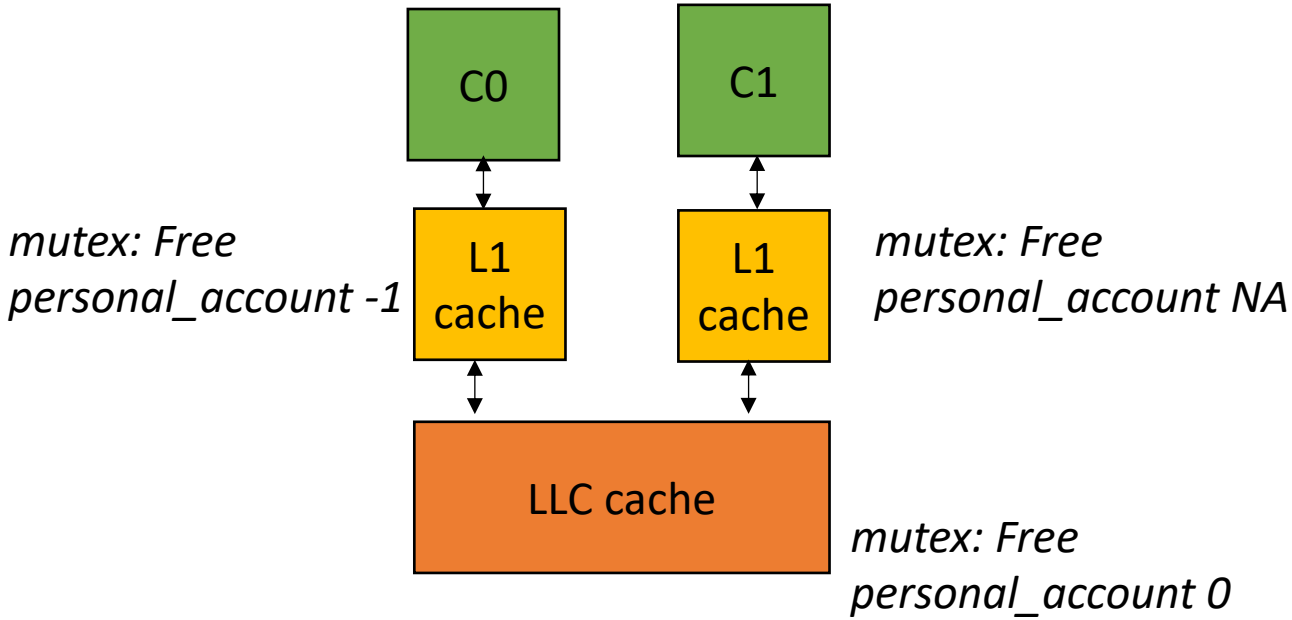


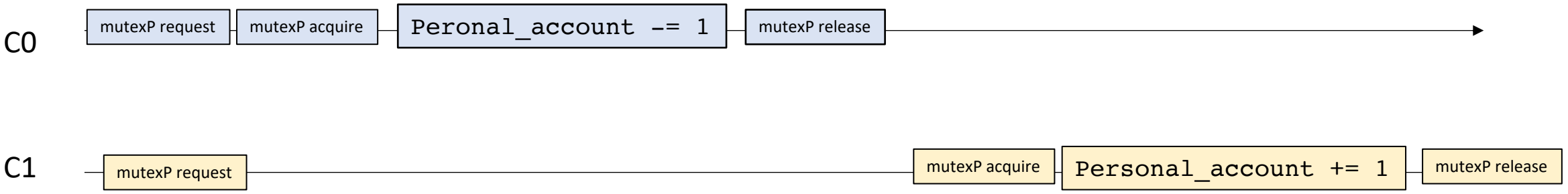
- Memory Fence (or Memory Barrier)



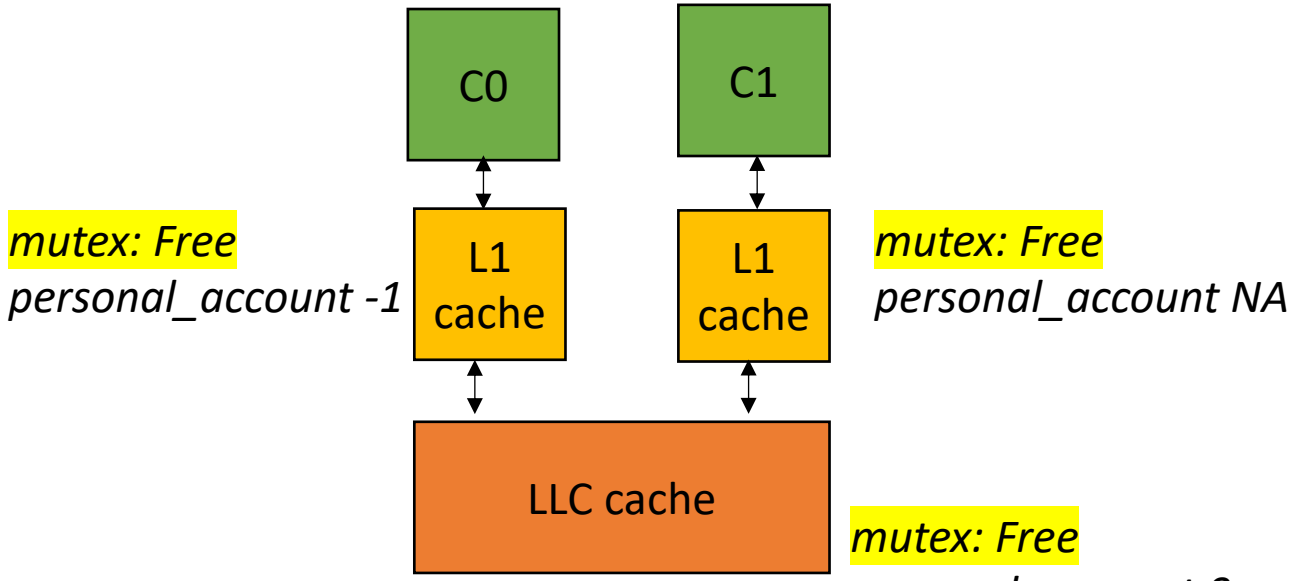


- Memory Fence (or Memory Barrier)

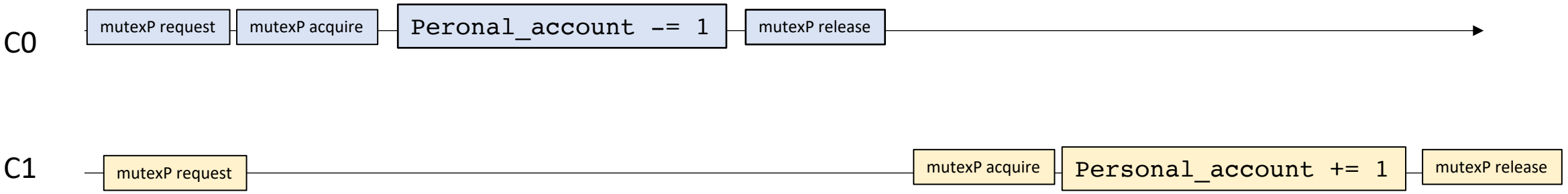




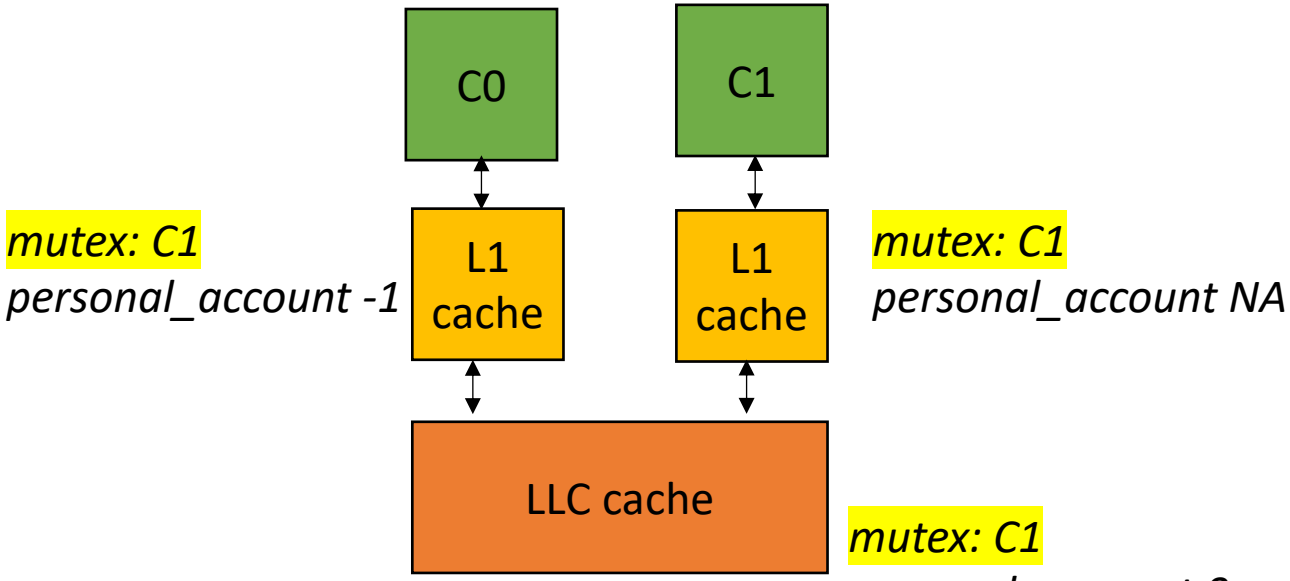
- Memory Fence (or Memory Barrier)



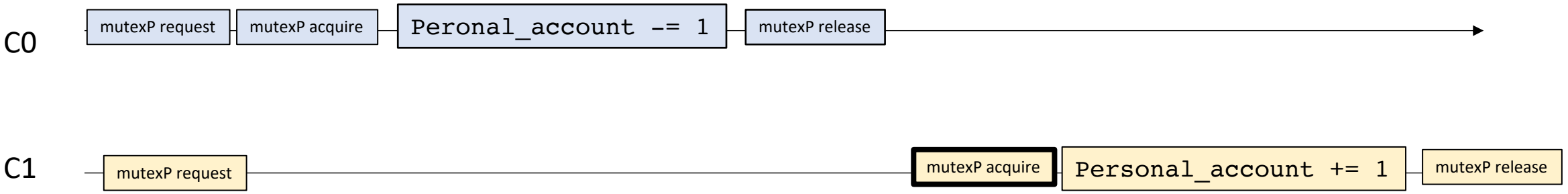
Possible for the mutex message to travel through the cache before the personal_account information!



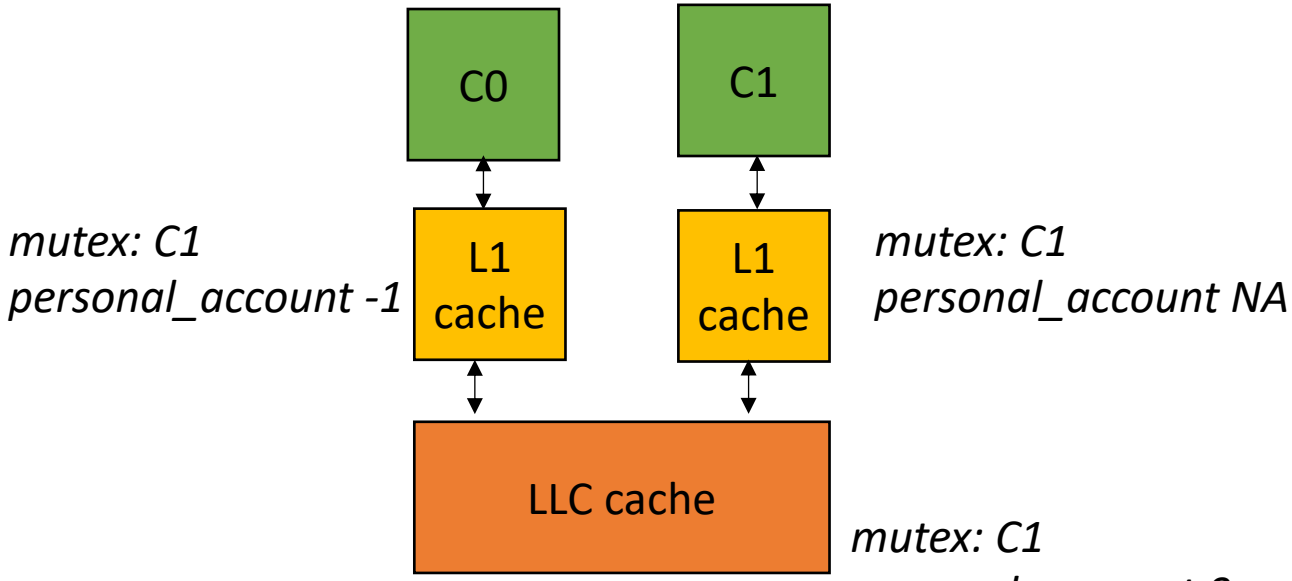
- Memory Fence (or Memory Barrier)



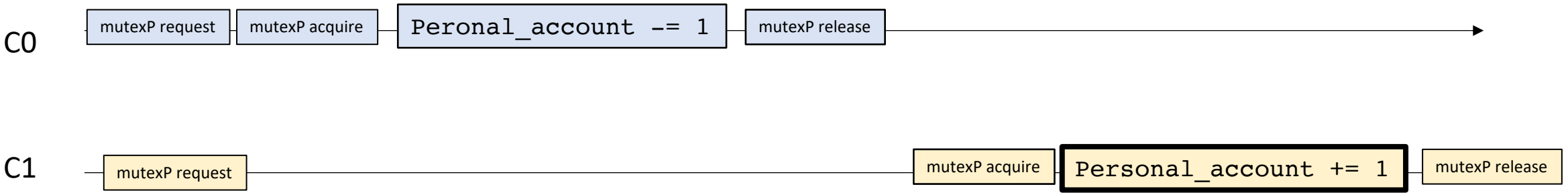
Possible for the mutex message to travel through the cache before the personal_account information!



- Memory Fence (or Memory Barrier)

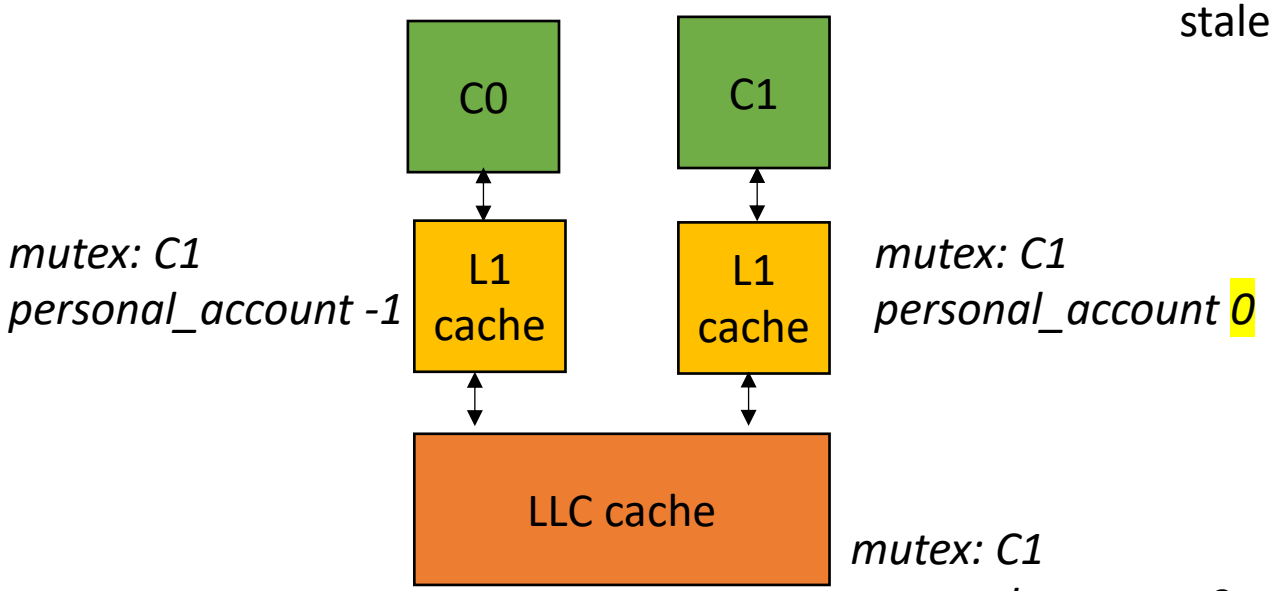


Possible for the mutex message to travel through the cache before the personal_account information!

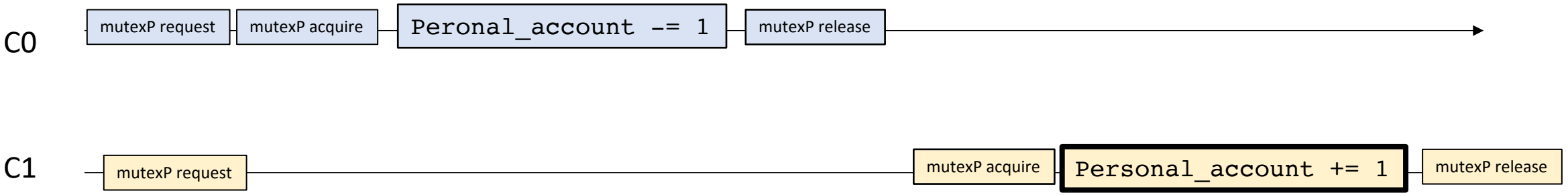


- Memory Fence (or Memory Barrier)

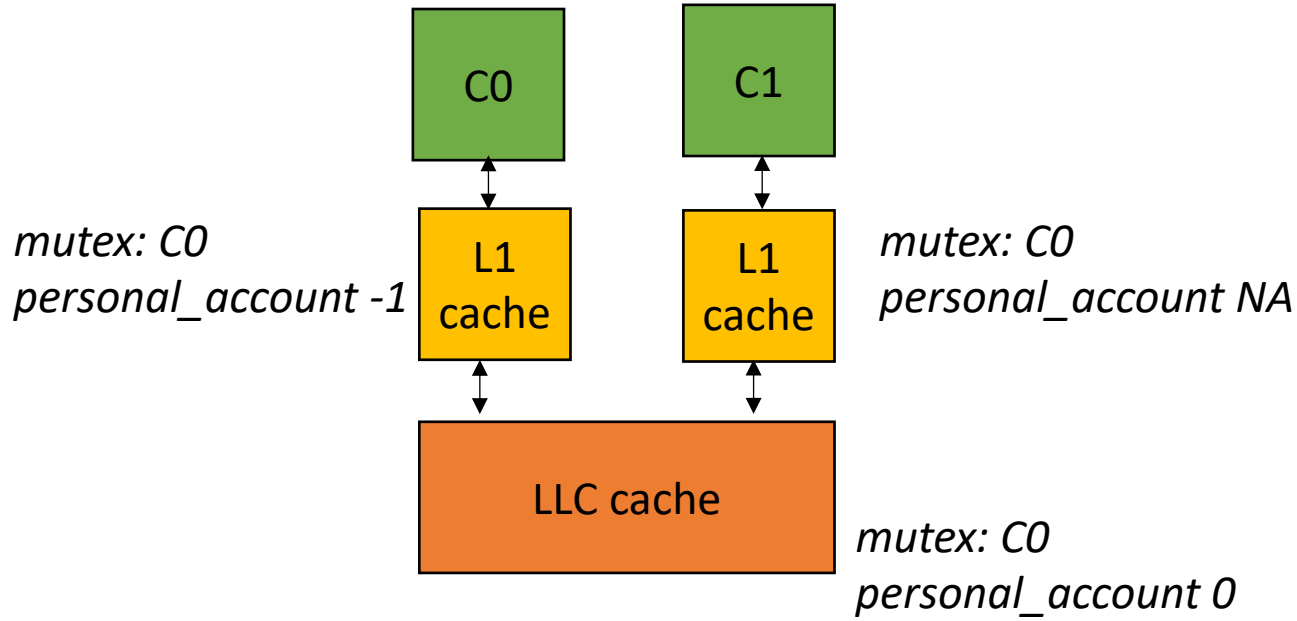
stale value!

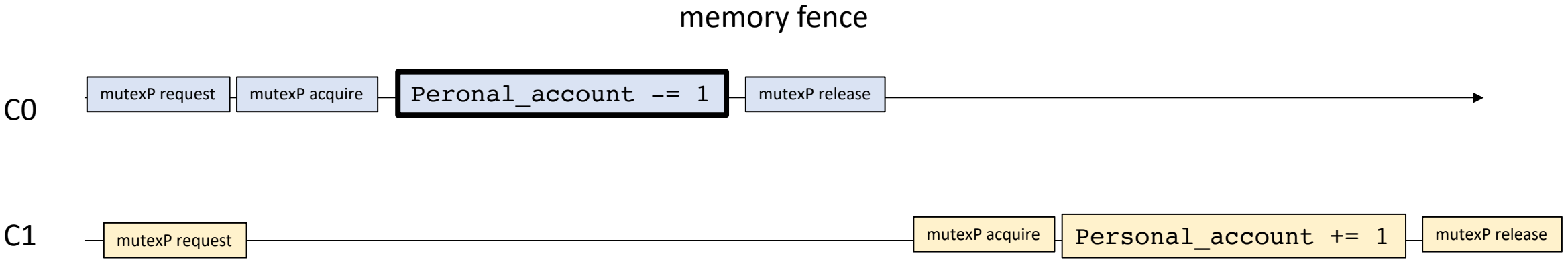


Possible for the mutex message to travel through the cache before the personal_account information!

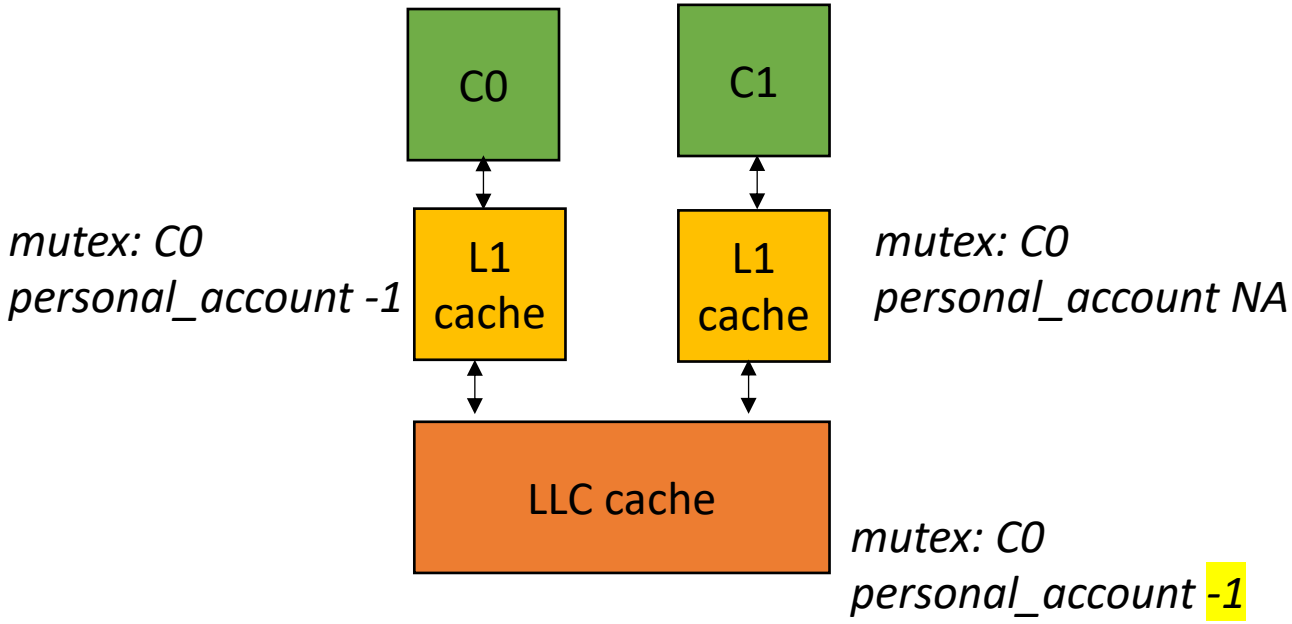


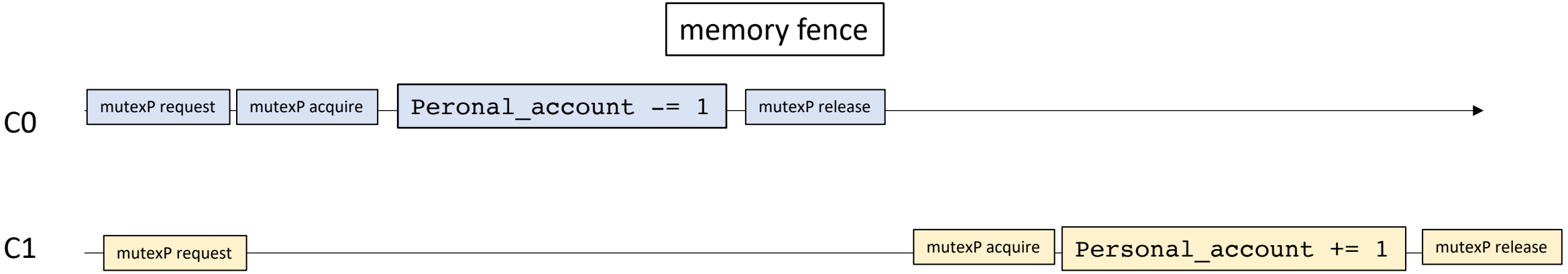
- Memory Fence (or Memory Barrier)



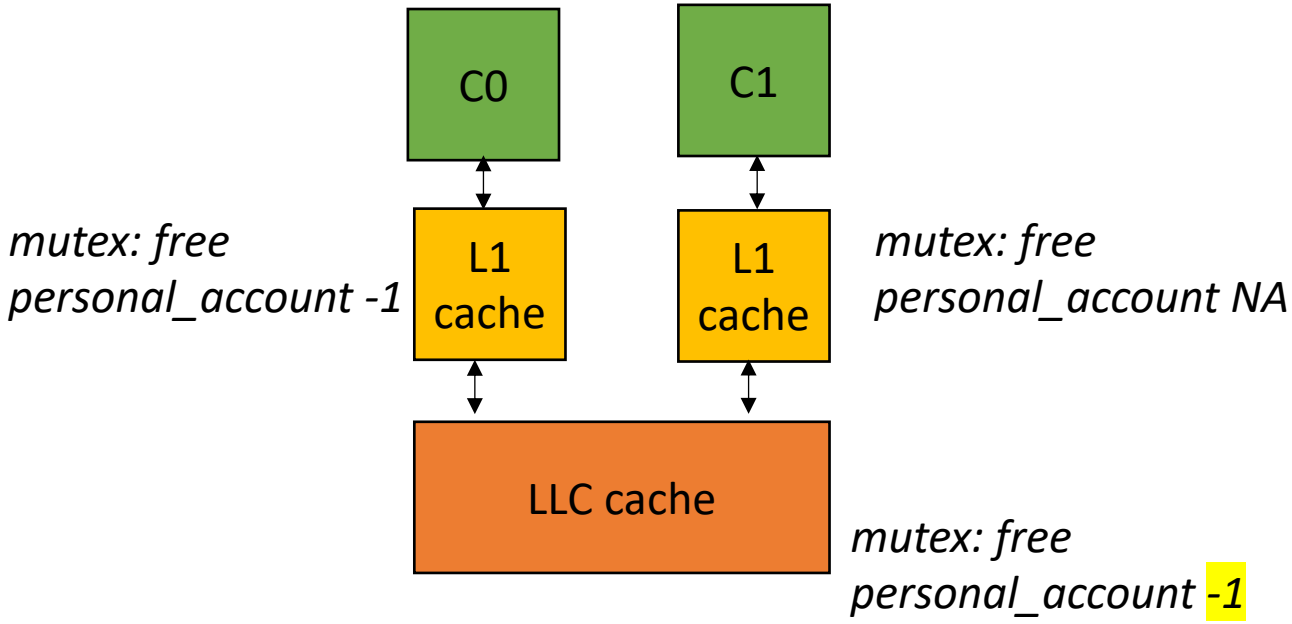


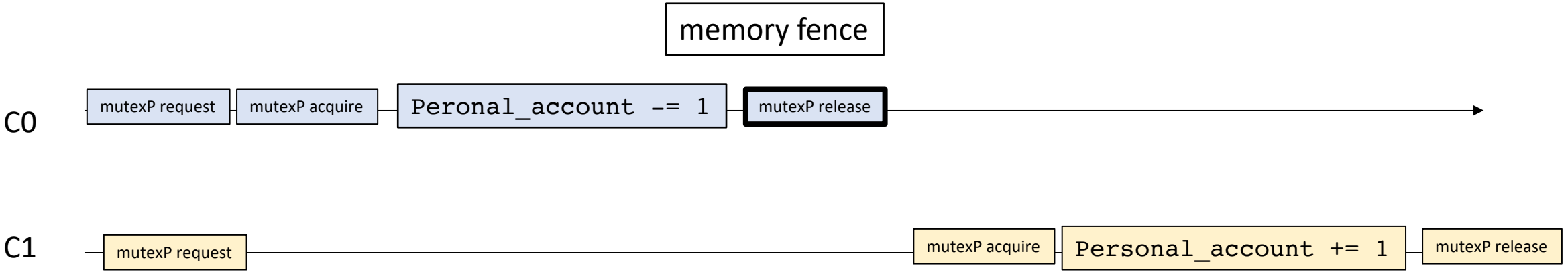
- Memory Fence (or Memory Barrier)



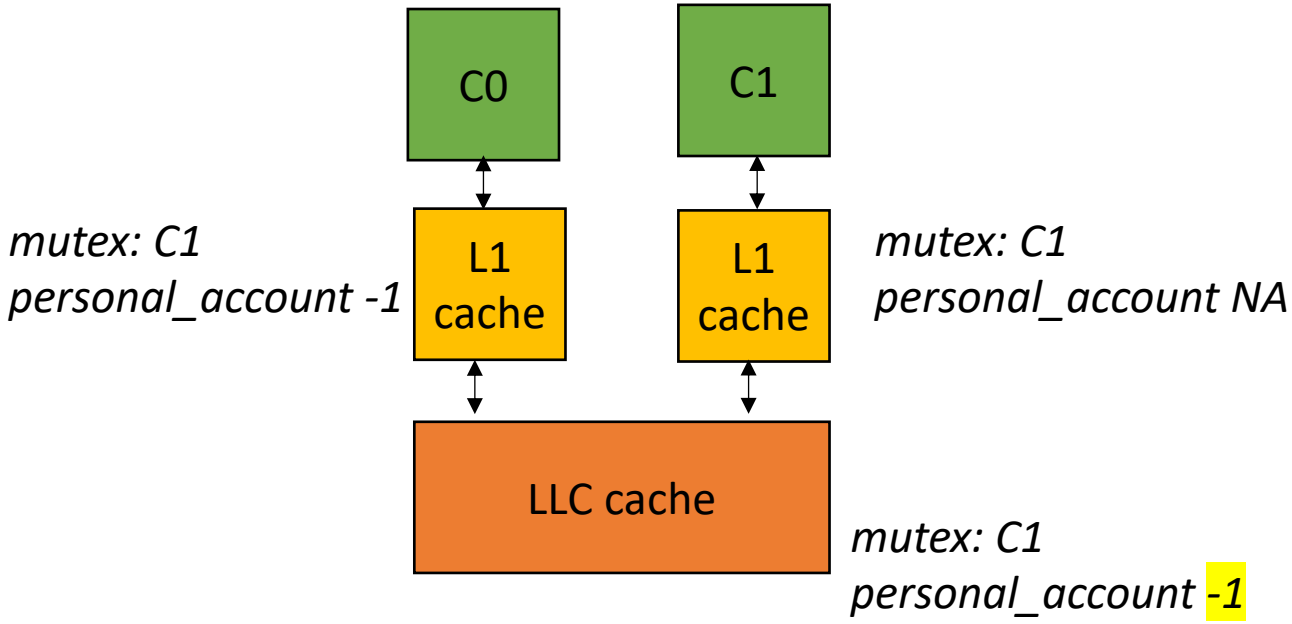


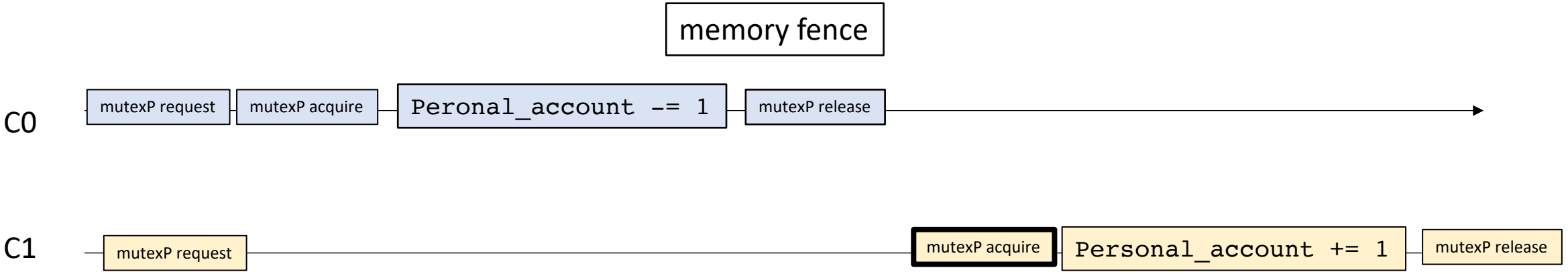
- Memory Fence (or Memory Barrier)



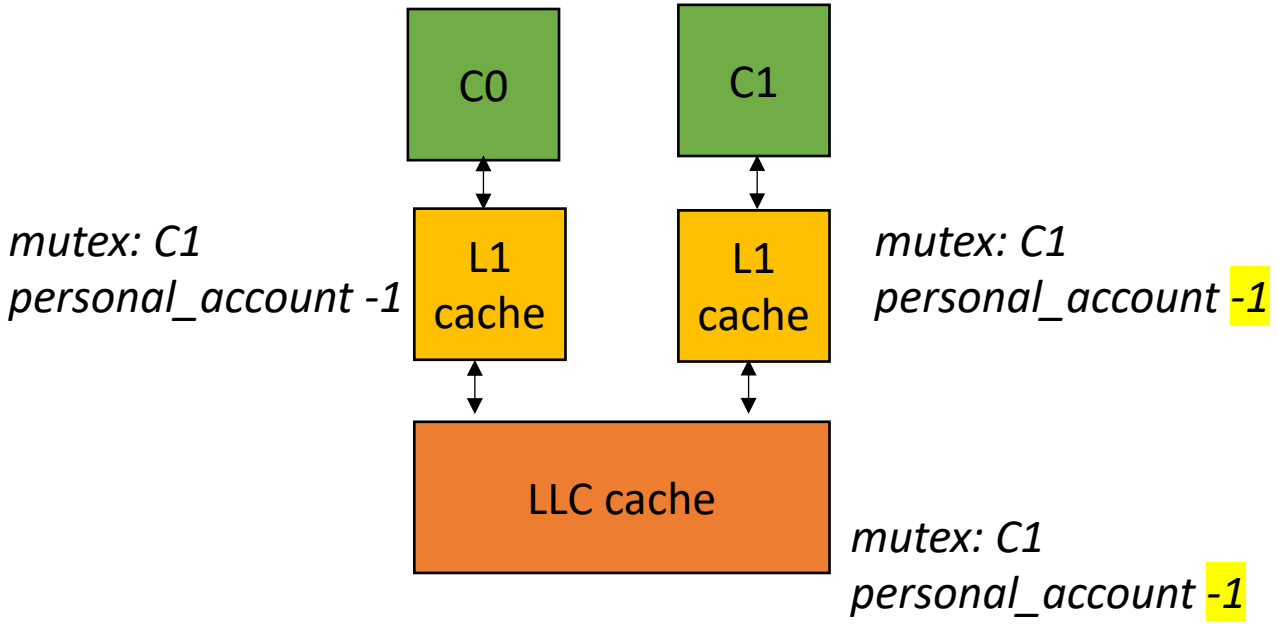


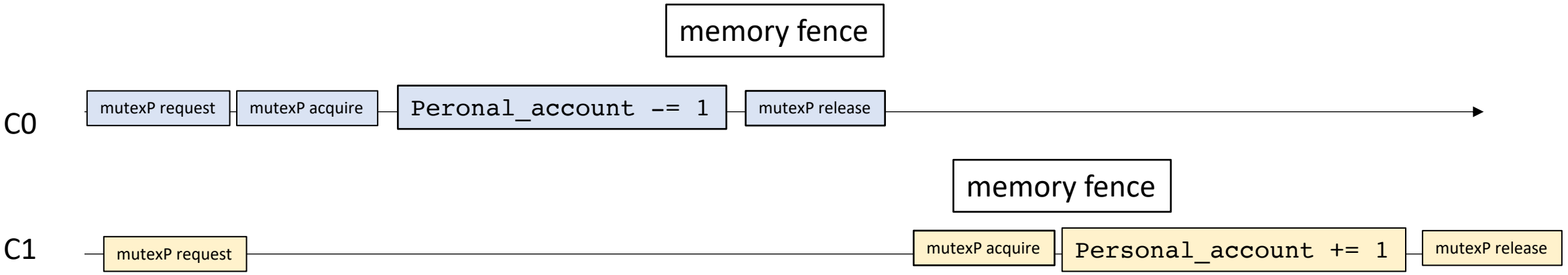
- Memory Fence (or Memory Barrier)



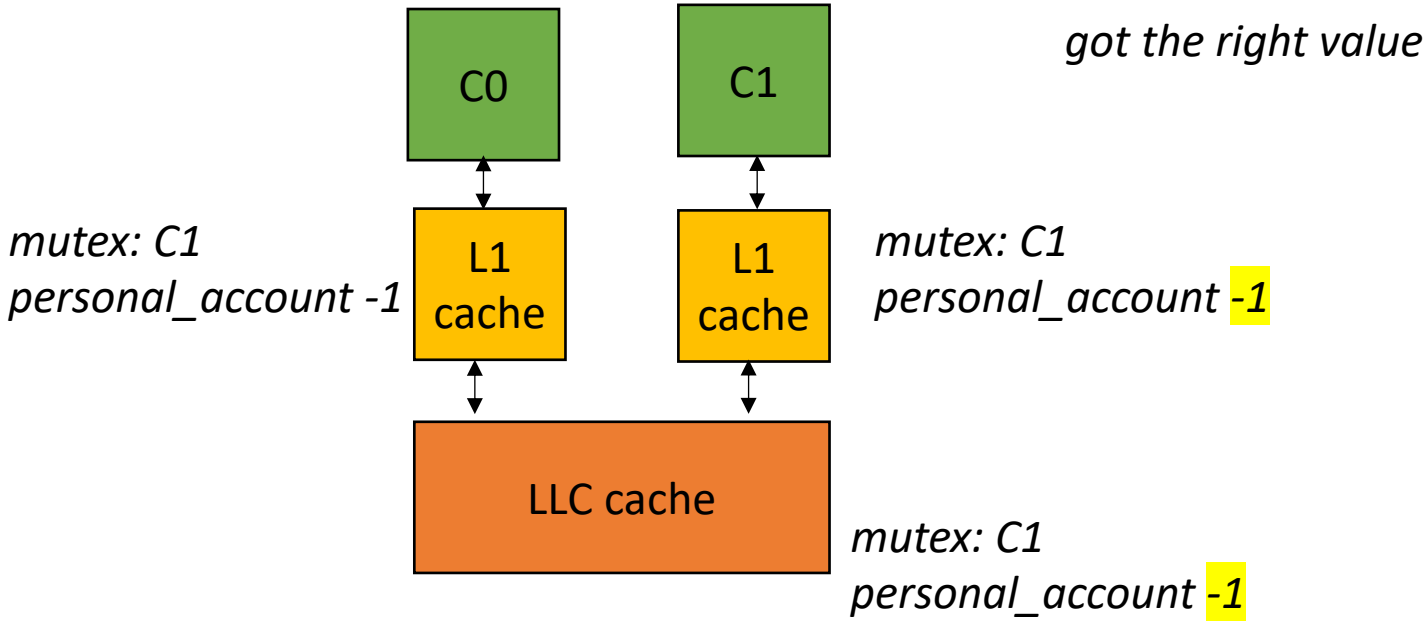


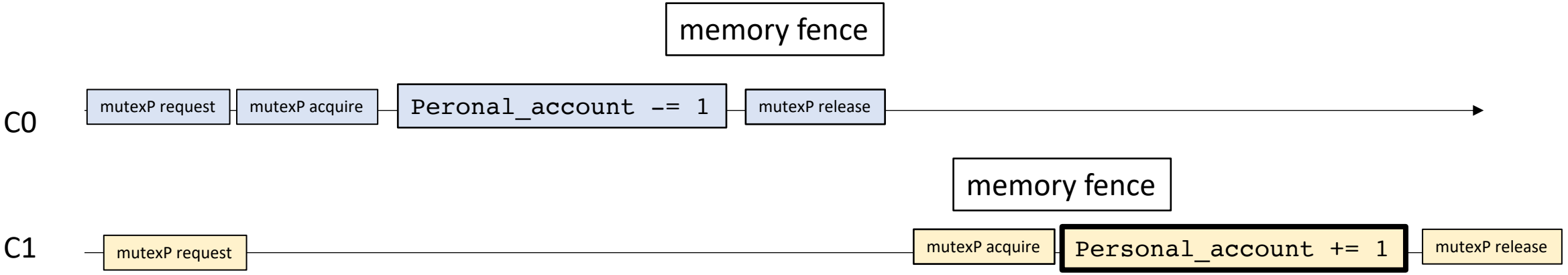
- Memory Fence (or Memory Barrier)





- Memory Fence (or Memory Barrier)





- **Memory Fence (or Memory Barrier)**

different architectures have different memory barriers

Intel X86 naturally manages caches in order

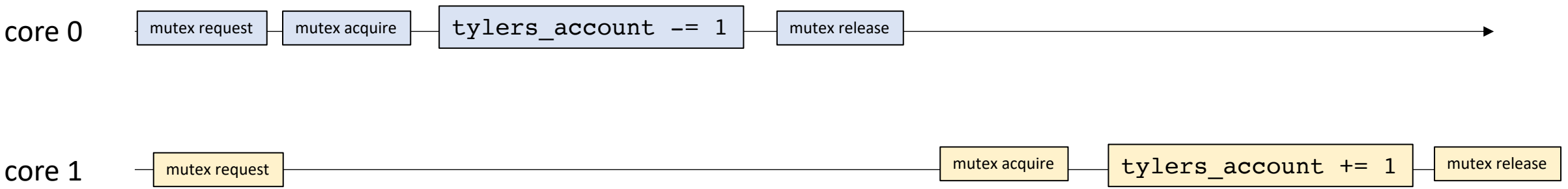
ARM and PowerPC let cache values flow out-of-order
GPUs let caches flow out-of-order

RISC-V has two models:
  more like x86: easier to program
  more like ARM: faster and more energy efficient

For mutexes, atomics will naturally handle the memory fences for us!

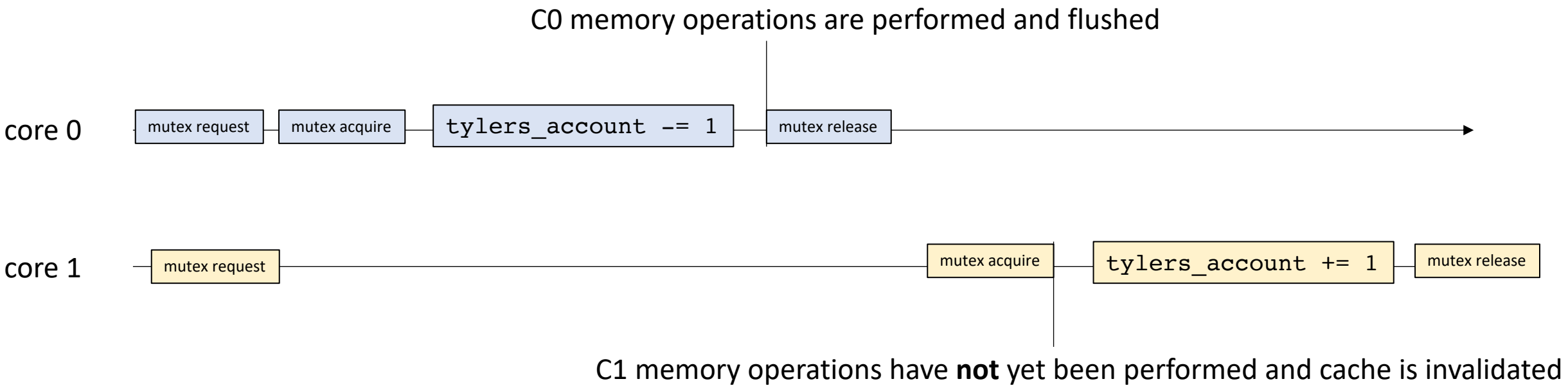
Atomics

- What do those fences (compiler and memory) give us?
- Atomics were designed so that we can implement things like mutexes!



Atomics

- What do those fences (compiler and memory) give us?
- Atomics were designed so that we can implement things like mutexes!



Mutex Implementations

- We will just consider two threads for now, with thread ids 0, 1

- A first attempt:
  - A mutex contains a boolean.
  - The mutex value set to 0 means that it is free. 1 means that some thread is holding it.
  - To lock the mutex, you wait until it is set to 0, then you store 1 in the flag.
  - To unlock the mutex, you set the mutex back to 0.

Mutex Implementations

```
#include <atomic>
using namespace std;

class Mutex {
public:
  Mutex() {
    flag = 0;
  }
  void lock();
  void unlock();
private:
  atomic_bool flag;
};
```

mutex is initialized to “free”

atomic_bool for our memory location

Mutex Implementations

```
void lock() {
  while (flag.load() == 1);
  flag.store(1);
}
```

While the mutex is not available (i.e. another thread has it)

Once the mutex is available, we will claim it

Mutex Implementations

```
void lock() {
  while (flag.load() == 1);
  flag.store(1);
}
```

While the mutex is not available (i.e. another thread has it)

Once the mutex is available, we will claim it

Whats up with this while loop?

Mutex Implementations

```
void unlock() {
    flag.store(0);
}
```

To release the mutex, we just set it back to 0 (available)

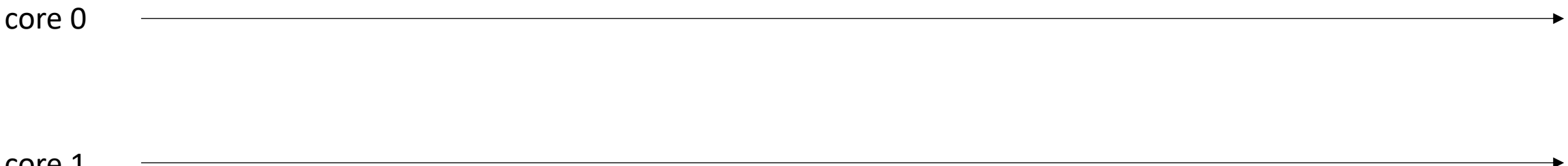
Analysis

```
void lock() {
  while (flag.load() == 1);
  flag.store(1);
}
```

```
void unlock() {
  flag.store(0);
}
```

Thread 0:
```
m.lock();
m.unlock();
```

Thread 1:
```
m.lock();
m.unlock();
```



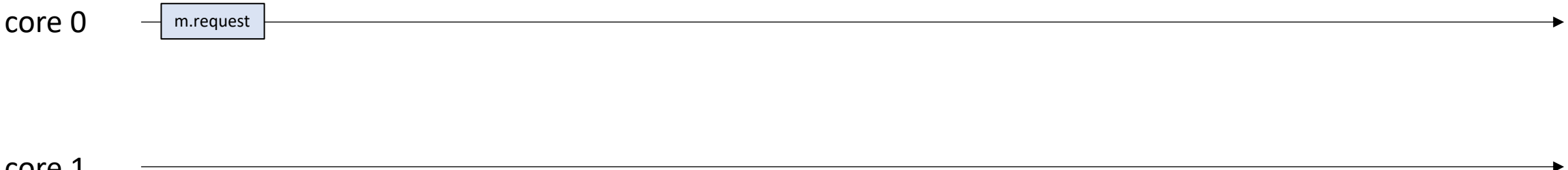
Analysis

```
void lock() {
  while (flag.load() == 1);
  flag.store(1);
}
```

```
void unlock() {
  flag.store(0);
}
```

Thread 0:
```
m.lock();
m.unlock();
```

Thread 1:
```
m.lock();
m.unlock();
```



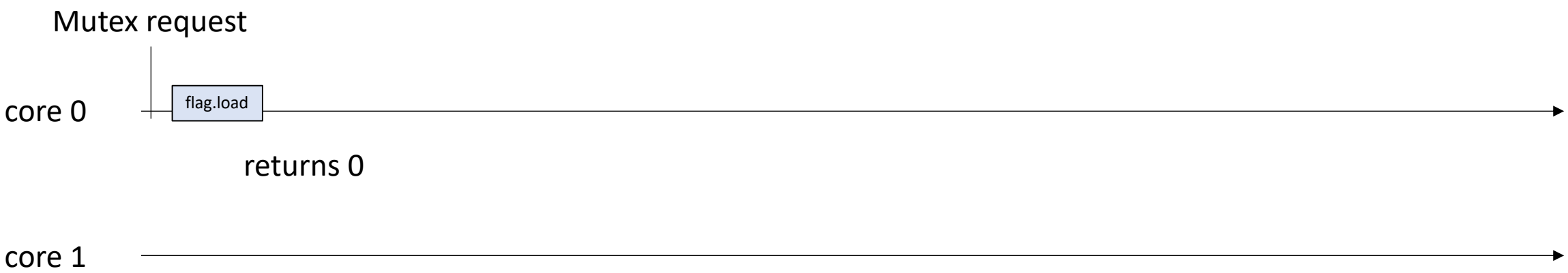
Analysis

```
void lock() {
  while (flag.load() == 1);
  flag.store(1);
}
```

```
void unlock() {
  flag.store(0);
}
```

Thread 0:
`m.lock();`
`m.unlock();`

Thread 1:
`m.lock();`
`m.unlock();`



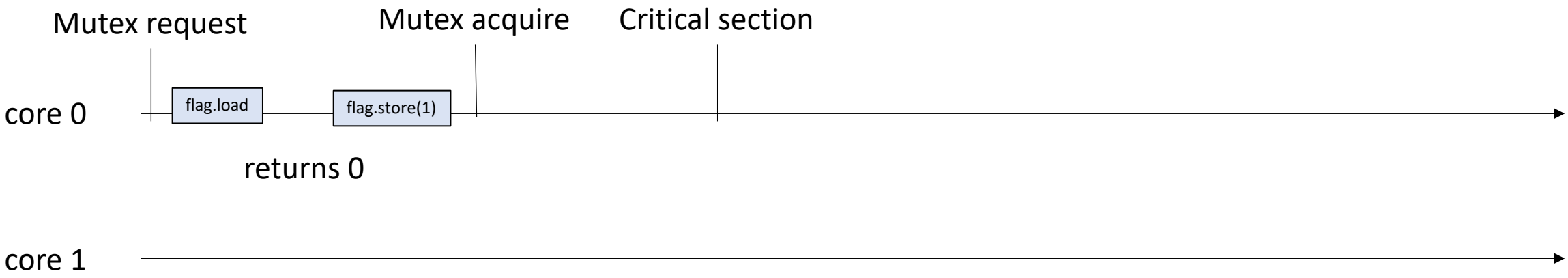
Analysis

```
void lock() {
  while (flag.load() == 1);
  flag.store(1);
}
```

```
void unlock() {
  flag.store(0);
}
```

Thread 0:
```
m.lock();
m.unlock();
```

Thread 1:
```
m.lock();
m.unlock();
```



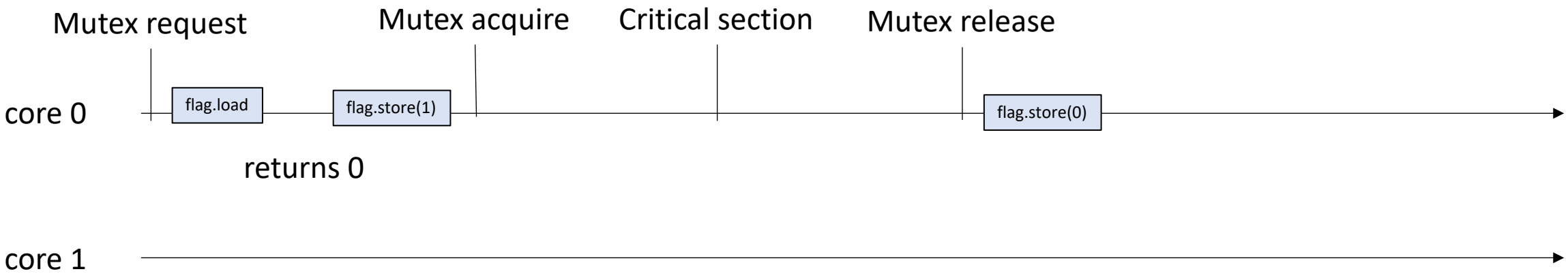
Analysis

```
void lock() {
  while (flag.load() == 1);
  flag.store(1);
}
```

```
void unlock() {
  flag.store(0);
}
```

Thread 0:
```
m.lock();
m.unlock();
```

Thread 1:
```
m.lock();
m.unlock();
```



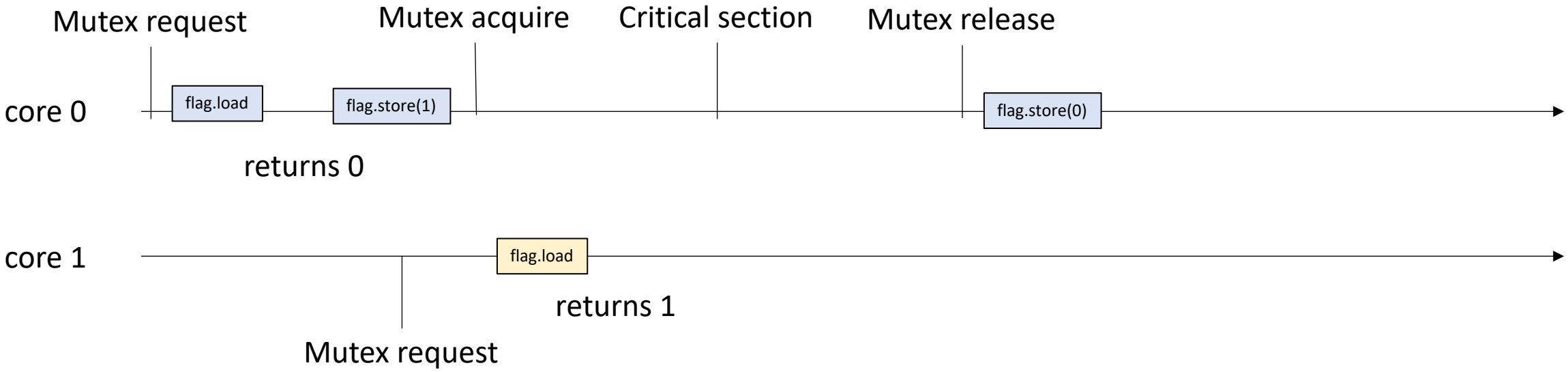
Analysis

```
void lock() {
  while (flag.load() == 1);
  flag.store(1);
}
```

```
void unlock() {
  flag.store(0);
}
```

Thread 0:
`m.lock();`
`m.unlock();`

Thread 1:
`m.lock();`
`m.unlock();`



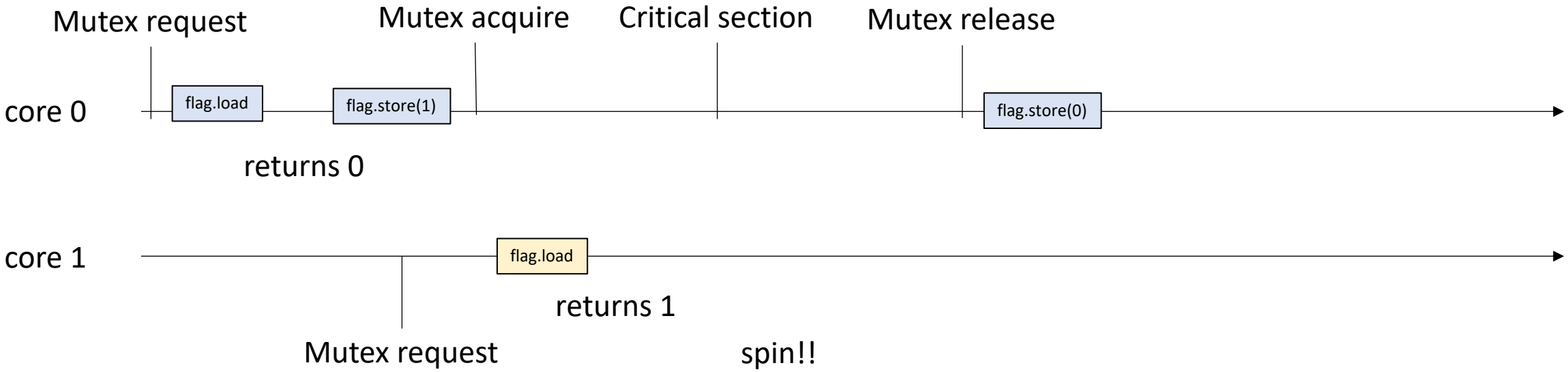
Analysis

```
void lock() {
  while (flag.load() == 1);
  flag.store(1);
}
```

```
void unlock() {
  flag.store(0);
}
```

Thread 0:
```
m.lock();
m.unlock();
```

Thread 1:
```
m.lock();
m.unlock();
```



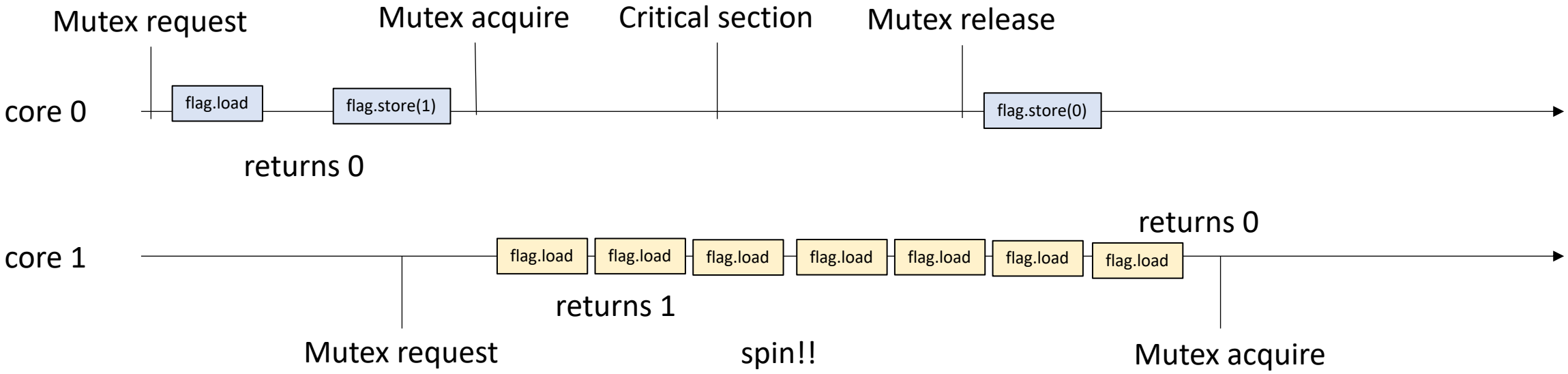
Analysis

```
void lock() {
  while (flag.load() == 1);
  flag.store(1);
}
```

```
void unlock() {
  flag.store(0);
}
```

Thread 0:
```
m.lock();
m.unlock();
```

Thread 1:
```
m.lock();
m.unlock();
```



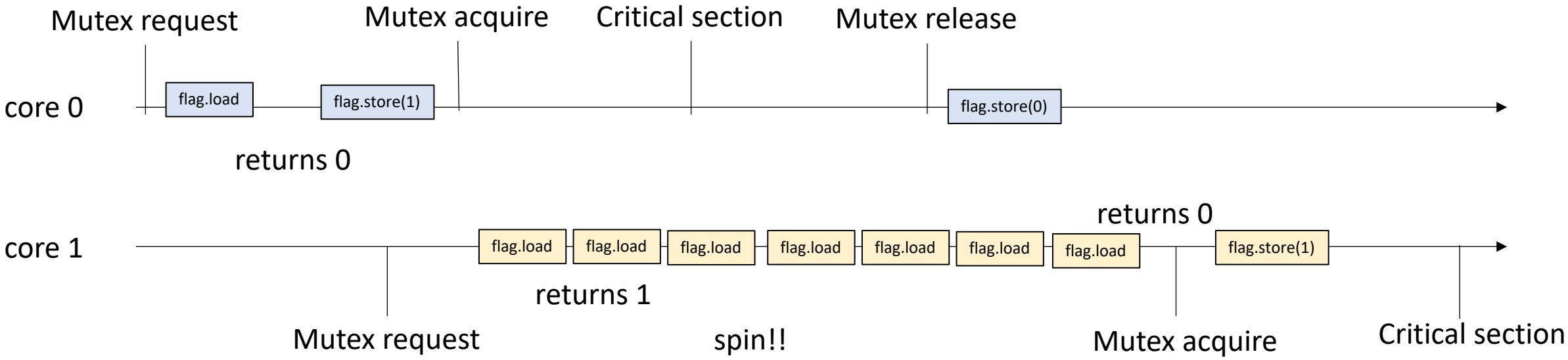
Analysis

```
void lock() {
  while (flag.load() == 1);
  flag.store(1);
}
```

```
void unlock() {
  flag.store(0);
}
```

Thread 0:
```
m.lock();
m.unlock();
```

Thread 1:
```
m.lock();
m.unlock();
```



Analysis

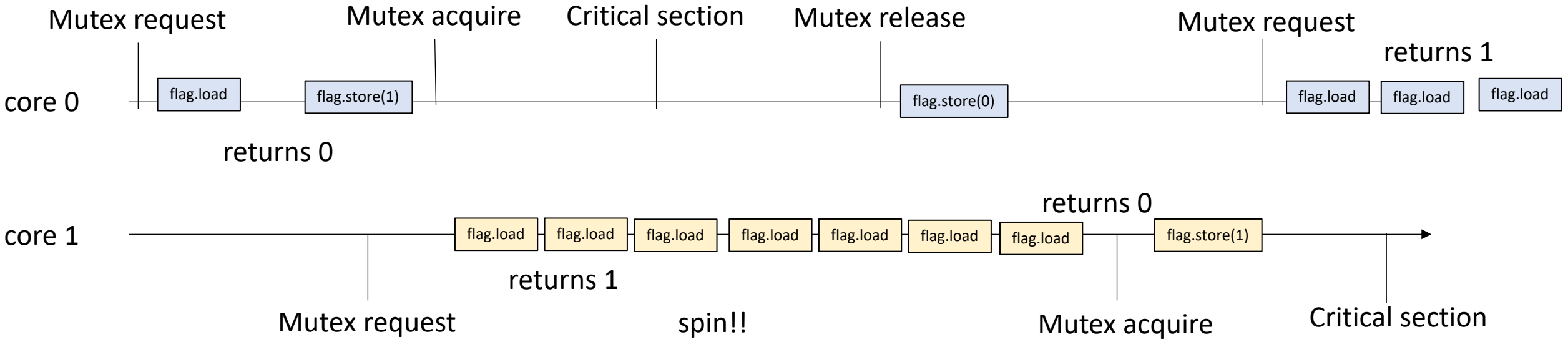
```
void lock() {
  while (flag.load() == 1);
  flag.store(1);
}
```

```
void unlock() {
  flag.store(0);
}
```

Thread 0:
```
m.lock();
m.unlock();
m.lock();
m.unlock();
```

Thread 1:
```
m.lock();
m.unlock();
```

Mutual Exclusion property!
critical sections do not overlap!



Analysis

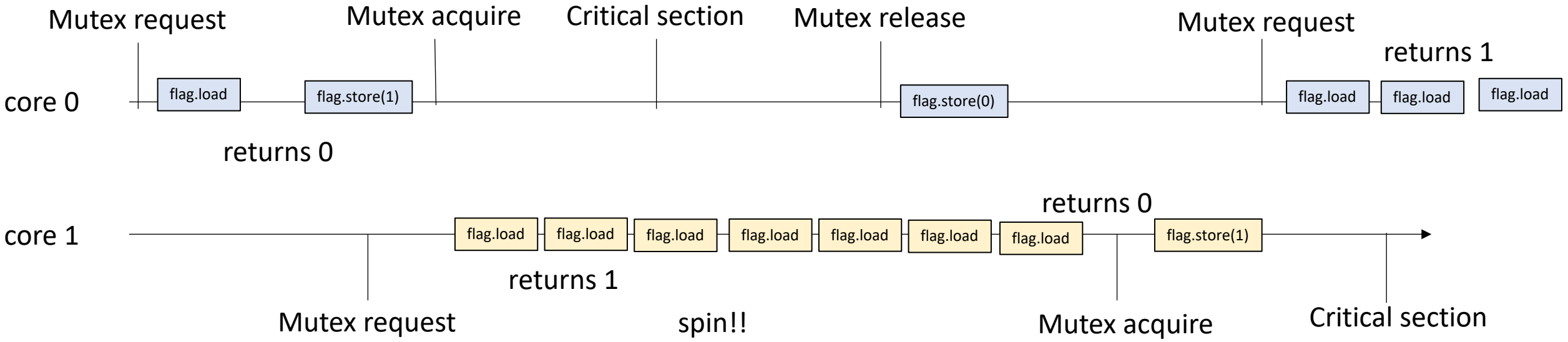
```
void lock() {
  while (flag.load() == 1);
  flag.store(1);
}
```

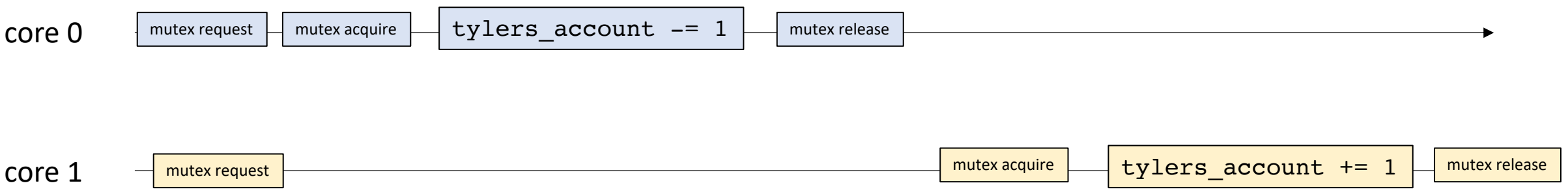
```
void unlock() {
  flag.store(0);
}
```

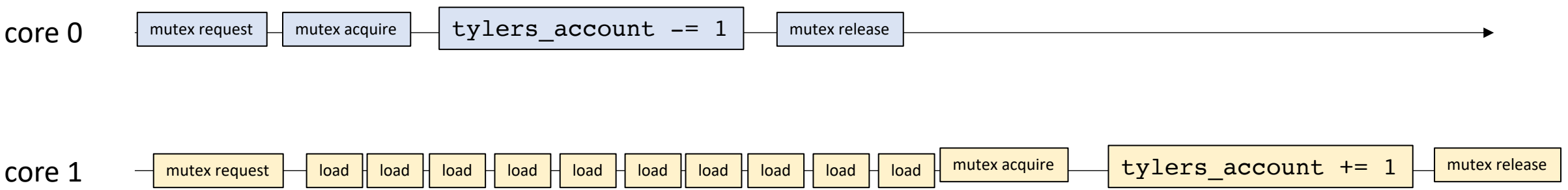
Thread 0:
```
m.lock();
m.unlock();
m.lock();
m.unlock();
```

Thread 1:
```
m.lock();
m.unlock();
```

Mutual Exclusion property!
critical sections do not overlap!




core 0
mutex request
mutex acquire
tylers_account -= 1
mutex release
core 1
mutex request
mutex acquire
tylers_account += 1
mutex release


core 0
mutex request
mutex acquire
tylers_account -= 1
mutex release
core 1
mutex request
load
load
load
load
load
load
load
load
load
load
mutex acquire
tylers_account += 1
mutex release

Analysis

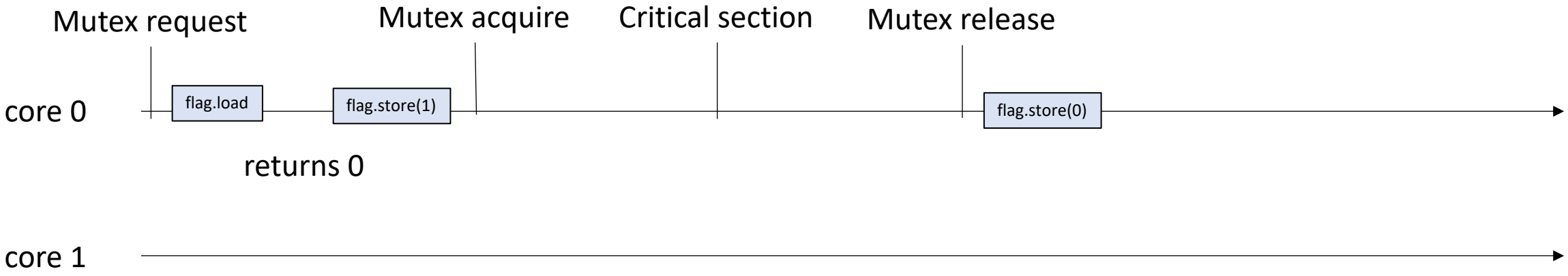
```
void lock() {
  while (flag.load() == 1);
  flag.store(1);
}
```

```
void unlock() {
  flag.store(0);
}
```

Thread 0:
`m.lock();`
`m.unlock();`

Thread 1:
`m.lock();`
`m.unlock();`

Lets try another interleaving



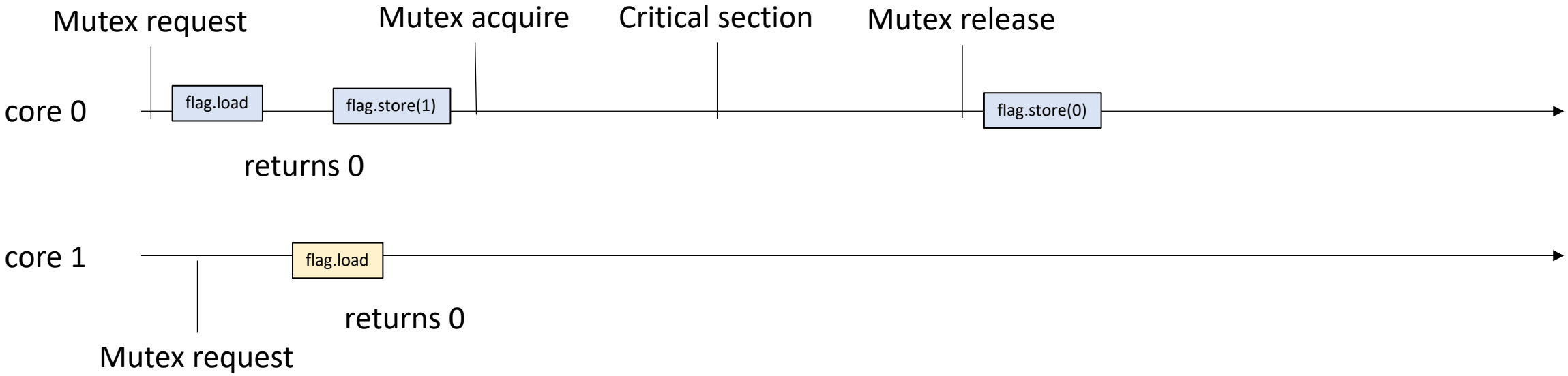
Analysis

```
void lock() {
  while (flag.load() == 1);
  flag.store(1);
}
```

```
void unlock() {
  flag.store(0);
}
```

Thread 0:
```
m.lock();
m.unlock();
```

Thread 1:
```
m.lock();
m.unlock();
```



Analysis

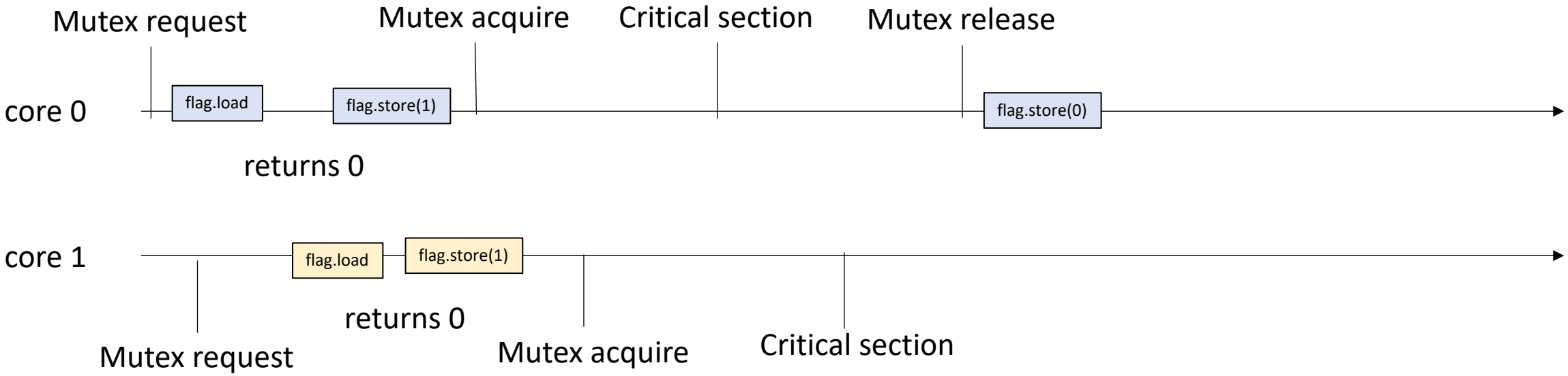
```
void lock() {
  while (flag.load() == 1);
  flag.store(1);
}
```

```
void unlock() {
  flag.store(0);
}
```

Thread 0:
```
m.lock();
m.unlock();
```

Thread 1:
```
m.lock();
m.unlock();
```

Critical sections overlap! This mutex implementation is not correct!



Mutex Implementations

- Second attempt:
  - A flag for each thread (2 flags)
  - If you want the mutex, set your flag to 1.
  - Spin while the other flag is 1 (the other thread has the mutex)
  - To release the mutex, set your flag to 0

Mutex Implementations

```
#include <atomic>
using namespace std;

class Mutex {
public:
  Mutex() {
    flag[0] = flag[1] = 0;
  }

  void lock();
  void unlock();

private:
  atomic_bool flag[2];
};
```

both initialized to 0

two flags this time

Mutex Implementations

```
void lock() {
  int i = thread_id;
  flag[i].store(1);
  int j = i == 0 ? 1 : 0;
  while (flag[j].load() == 1);
}
```

Thread id (0, or 1)

Mark your intention to take the lock

Wait for other thread to leave the critical section

Mutex Implementations

```
void unlock() {
  int i = thread_id;
  flag[i].store(0);
}
```

Thread id (0, or 1)

Mark your flag to say you have left the critical section.

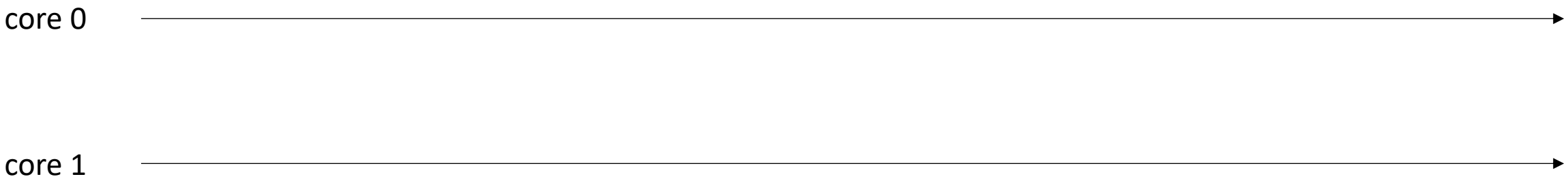
Analysis

```
void lock() {
  int i = thread_id;
  flag[i].store(1);
  int j = i == 0 ? 1 : 0;
  while (flag[j].load() == 1);
}
```

```
void unlock() {
  int i = thread_id;
  flag[i].store(0);
}
```

Thread 0:
```
m.lock();
m.unlock();
```

Thread 1:
```
m.lock();
m.unlock();
```



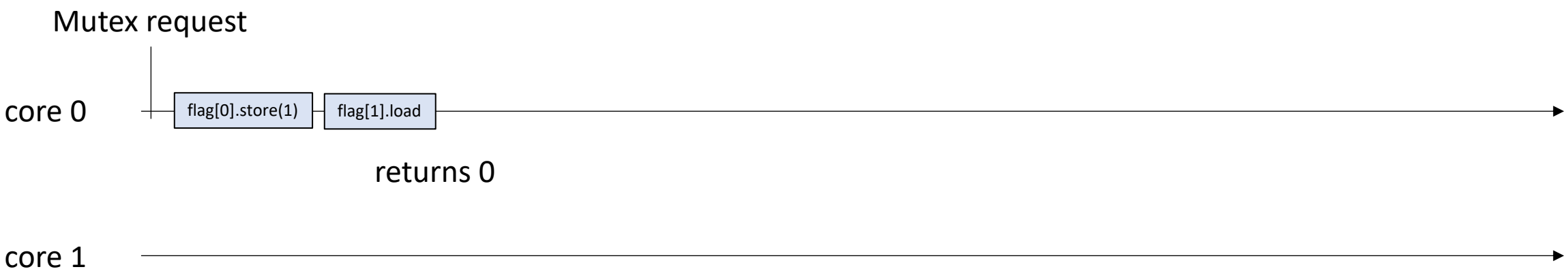
Analysis

```
void lock() {
  int i = thread_id;
  flag[i].store(1);
  int j = i == 0 ? 1 : 0;
  while (flag[j].load() == 1);
}
```

```
void unlock() {
  int i = thread_id;
  flag[i].store(0);
}
```

Thread 0:
```
m.lock();
m.unlock();
```

Thread 1:
```
m.lock();
m.unlock();
```



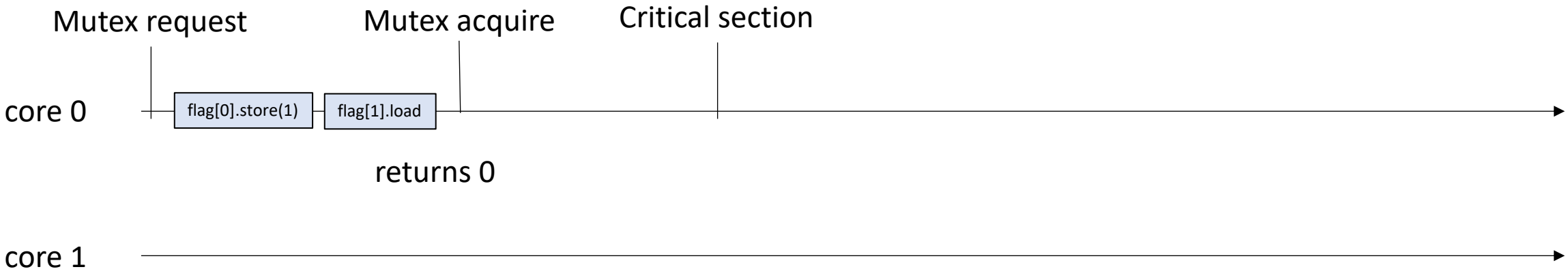
Analysis

```
void lock() {
  int i = thread_id;
  flag[i].store(1);
  int j = i == 0 ? 1 : 0;
  while (flag[j].load() == 1);
}
```

```
void unlock() {
  int i = thread_id;
  flag[i].store(0);
}
```

Thread 0:
```
m.lock();
m.unlock();
```

Thread 1:
```
m.lock();
m.unlock();
```



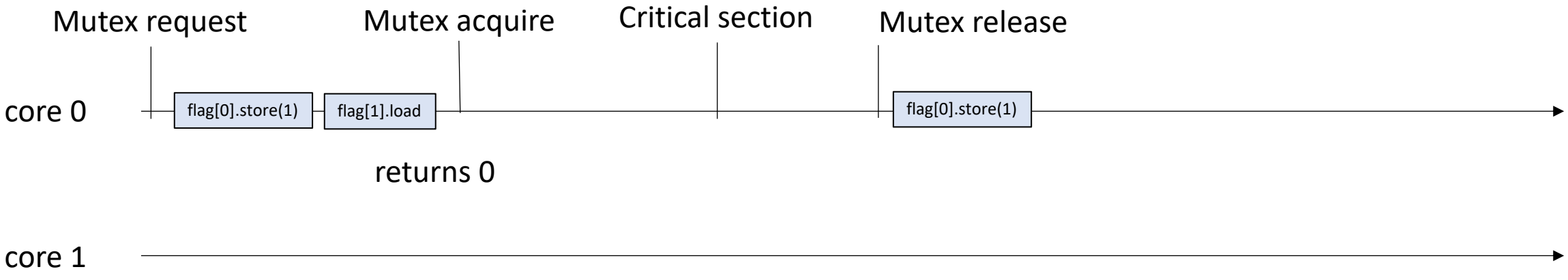
Analysis

```
void lock() {
  int i = thread_id;
  flag[i].store(1);
  int j = i == 0 ? 1 : 0;
  while (flag[j].load() == 1);
}
```

```
void unlock() {
  int i = thread_id;
  flag[i].store(0);
}
```

Thread 0:
```
m.lock();
m.unlock();
```

Thread 1:
```
m.lock();
m.unlock();
```



Analysis

```
void lock() {
  int i = thread_id;
  flag[i].store(1);
  int j = i == 0 ? 1 : 0;
  while (flag[j].load() == 1);
}
```

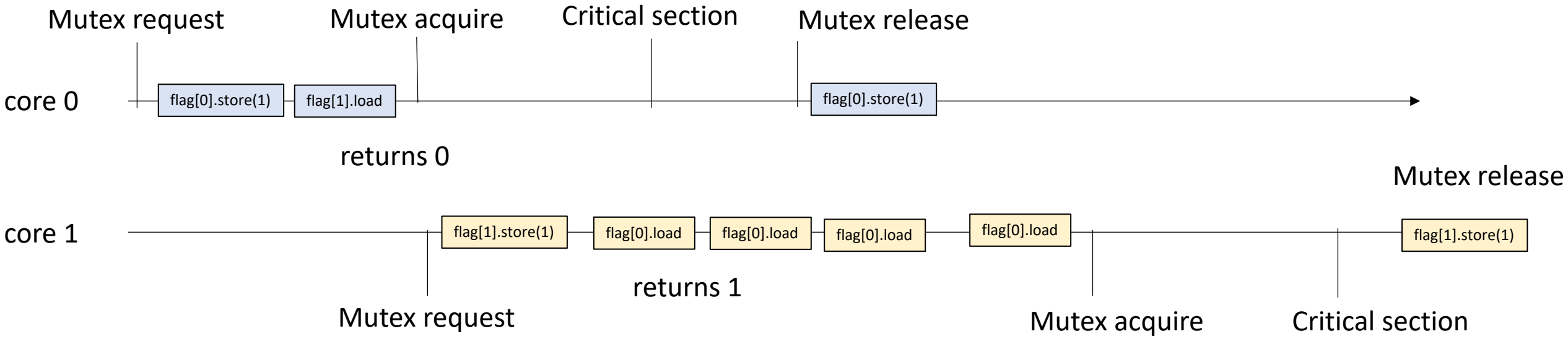
```
void unlock() {
  int i = thread_id;
  flag[i].store(0);
}
```

Thread 0:
```
m.lock();
m.unlock();
```

Thread 1:
```
m.lock();
m.unlock();
```

critical sections do not overlap!

proof?



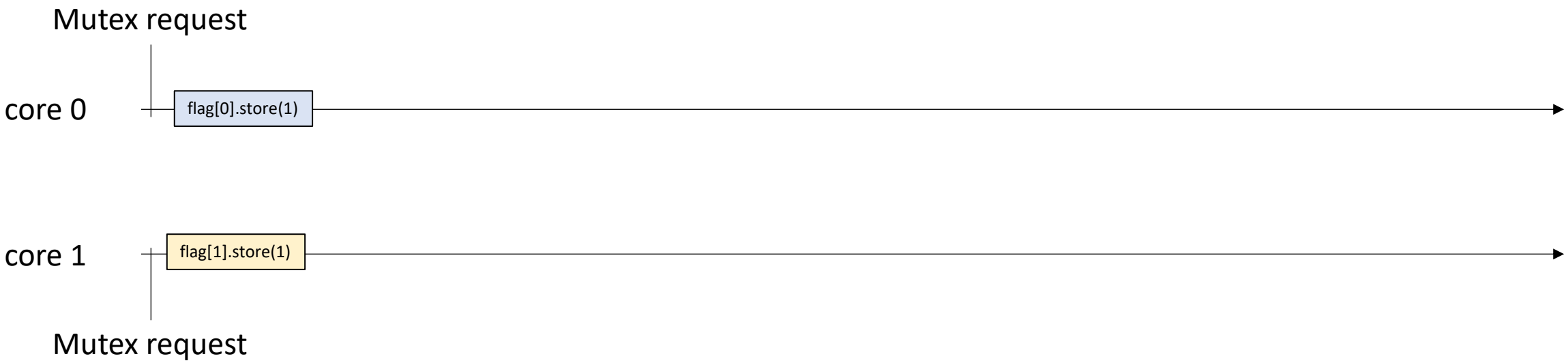
Analysis

```
void lock() {
  int i = thread_id;
  flag[i].store(1);
  int j = i == 0 ? 1 : 0;
  while (flag[j].load() == 1);
}
```

```
void unlock() {
  int i = thread_id;
  flag[i].store(0);
}
```

Thread 0:
```
m.lock();
m.unlock();
```

Thread 1:
```
m.lock();
m.unlock();
```



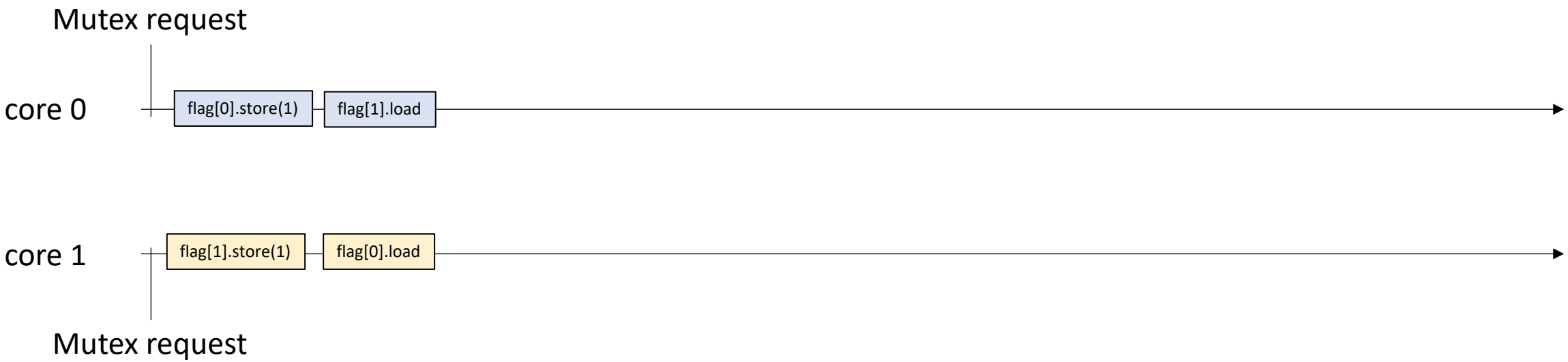
Analysis

```
void lock() {
  int i = thread_id;
  flag[i].store(1);
  int j = i == 0 ? 1 : 0;
  while (flag[j].load() == 1);
}
```

```
void unlock() {
  int i = thread_id;
  flag[i].store(0);
}
```

Thread 0:
```
m.lock();
m.unlock();
```

Thread 1:
```
m.lock();
m.unlock();
```



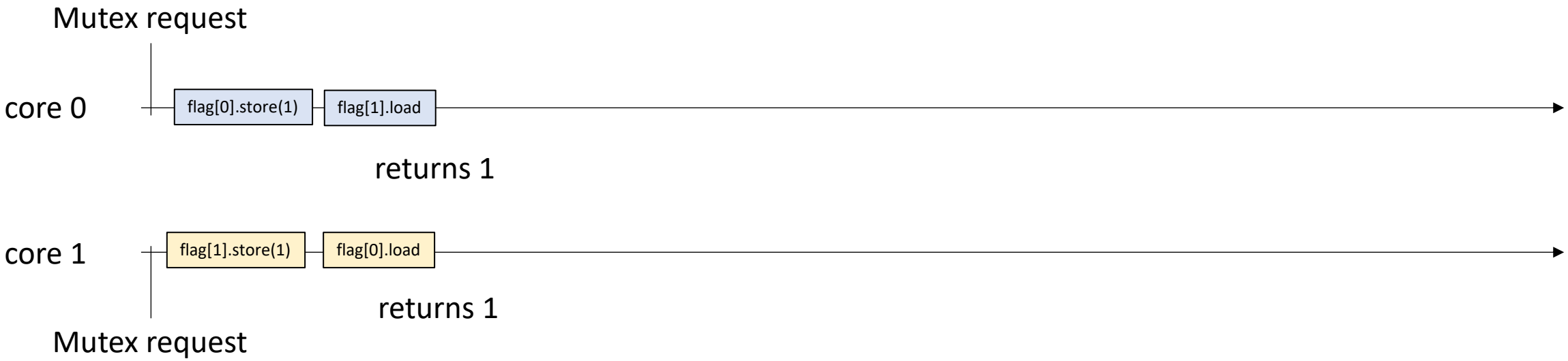
Analysis

```
void lock() {
  int i = thread_id;
  flag[i].store(1);
  int j = i == 0 ? 1 : 0;
  while (flag[j].load() == 1);
}
```

```
void unlock() {
  int i = thread_id;
  flag[i].store(0);
}
```

Thread 0:
`m.lock();`
`m.unlock();`

Thread 1:
`m.lock();`
`m.unlock();`



Analysis

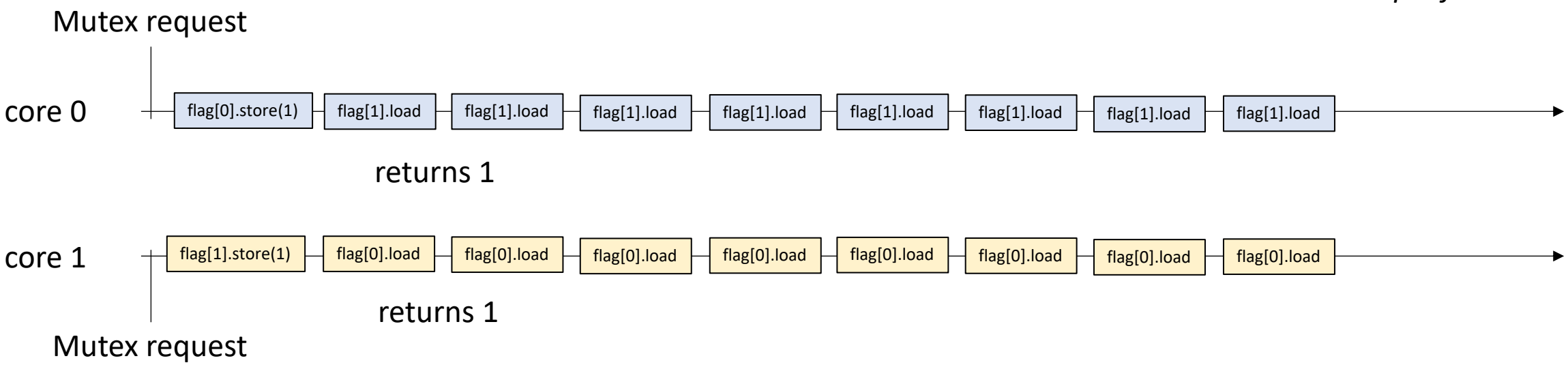
```
void lock() {
  int i = thread_id;
  flag[i].store(1);
  int j = i == 0 ? 1 : 0;
  while (flag[j].load() == 1);
}
```

```
void unlock() {
  int i = thread_id;
  flag[i].store(0);
}
```

Thread 0:
`m.lock();`
`m.unlock();`

Thread 1:
`m.lock();`
`m.unlock();`

Both will spin forever!

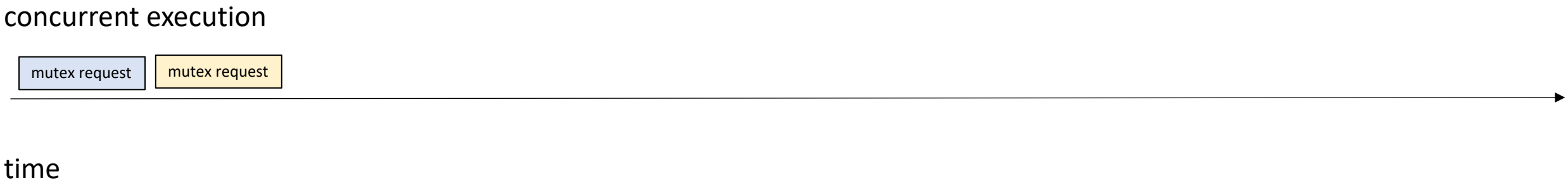


Properties of mutexes

Three properties

- **Deadlock Freedom -** If a thread has requested the mutex, and no thread currently holds the mutex, the mutex must be acquired by one of the requesting threads

Program cannot hang here
Either thread 0 or thread 1 must acquire the mutex



Mutex Implementations

Third attempt

Mutex Implementations

```
class Mutex {
public:
  Mutex() {
    victim = -1;
  }

  void lock();
  void unlock();


private:
  atomic_int victim;
};
```

initialized to -1

back to a single variable

Mutex Implementations

```
void lock() {
  victim.store(thread_id);
  while (victim.load() == thread_id);
}
```

Volunteer to be the victim

Victims only job is to spin

Mutex Implementations

```
void unlock() {}
```

No unlock!

```
void lock() {
  victim.store(thread_id);
  while (victim.load() == thread_id);
}
```

```
void unlock() {}
```

Thread 0:

```
m.lock();
m.unlock();
```

```
void lock() {
  victim.store(thread_id);
  while (victim.load() == thread_id);
}
```

```
void unlock() {}
```

Thread 0:

```
m.lock();
m.unlock();
```

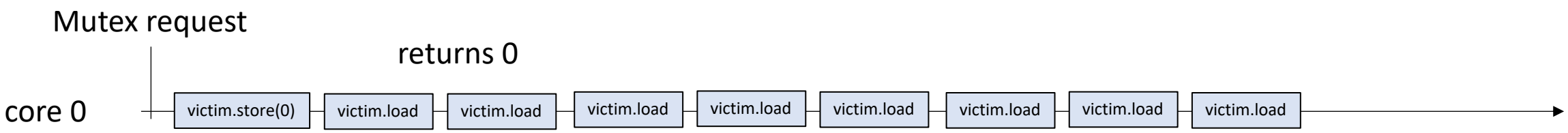
```
void lock() {
  victim.store(thread_id);
  while (victim.load() == thread_id);
}
```

```
void unlock() {}
```

Thread 0:

```
m.lock();
m.unlock();
```

spins forever if
the second thread
never tries to take the mutex!

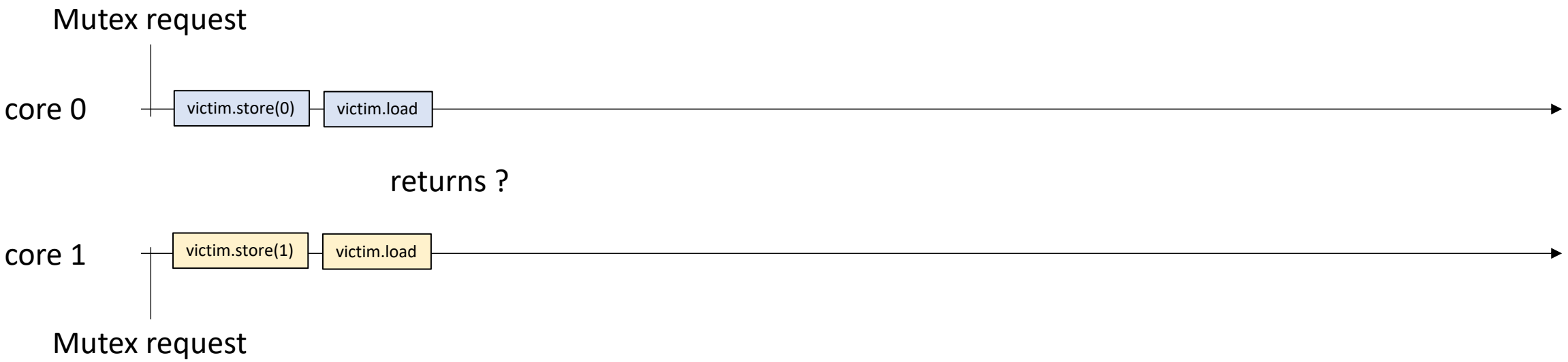


```
void lock() {
  victim.store(thread_id);
  while (victim.load() == thread_id);
}
```

```
void unlock() {}
```

Thread 0:
`m.lock();`
`m.unlock();`

Thread 1:
`m.lock();`
`m.unlock();`

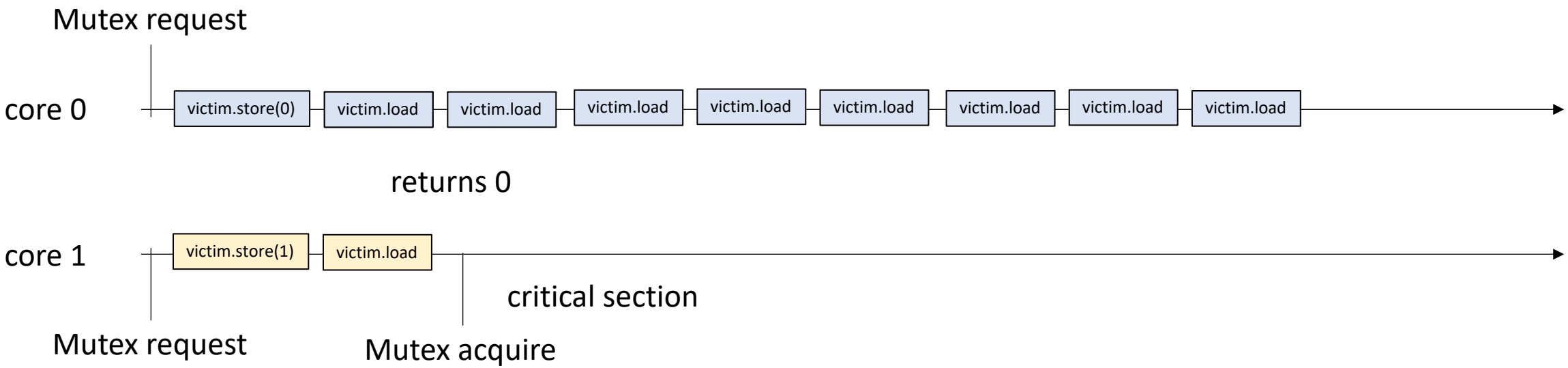


```
void lock() {
  victim.store(thread_id);
  while (victim.load() == thread_id);
}
```

```
void unlock() {}
```

Thread 0:
`m.lock();`
`m.unlock();`

Thread 1:
`m.lock();`
`m.unlock();`

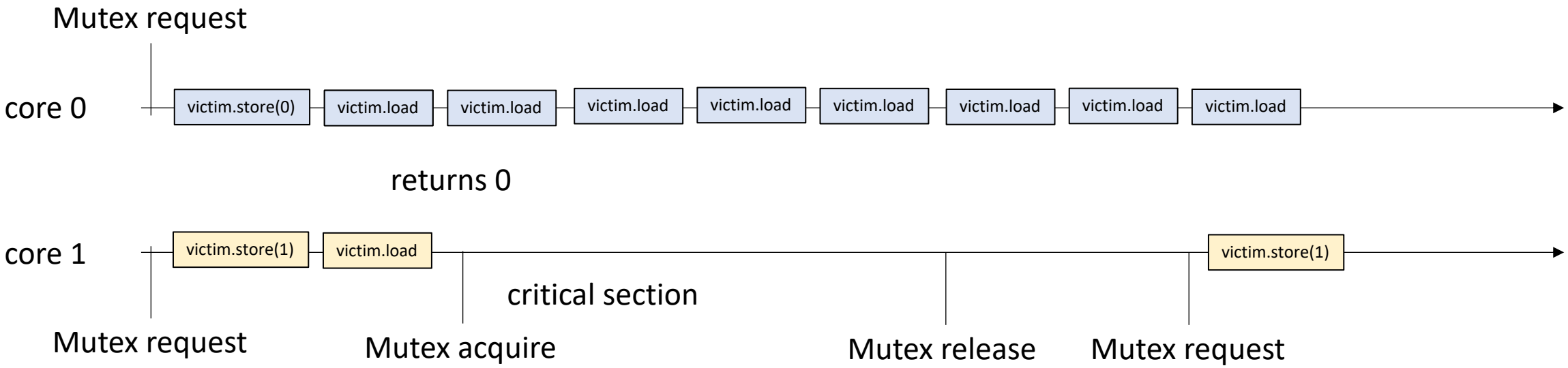


```
void lock() {
  victim.store(thread_id);
  while (victim.load() == thread_id);
}
```

```
void unlock() {}
```

Thread 0:
```
m.lock();
m.unlock();
```

Thread 1:
```
m.lock();
m.unlock();
```

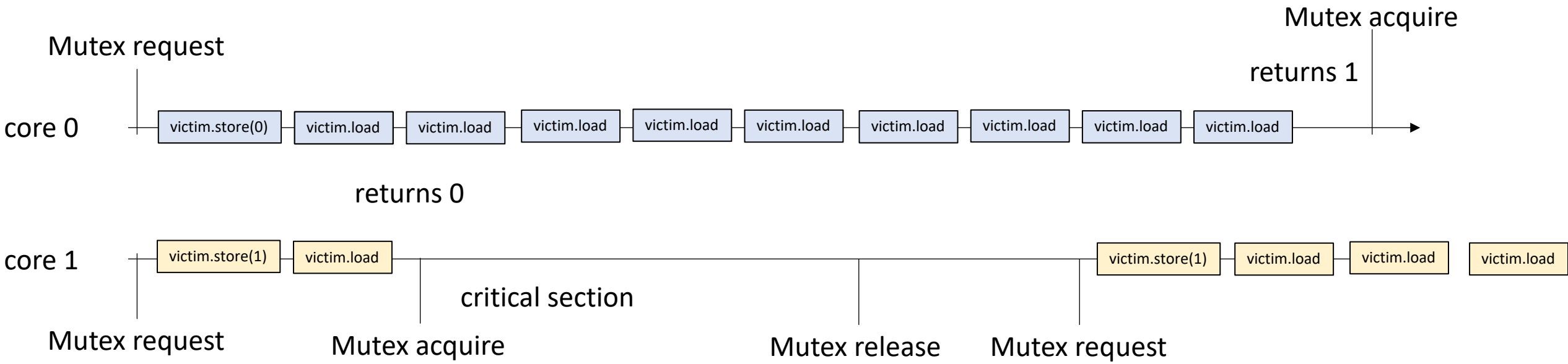


```
void lock() {
  victim.store(thread_id);
  while (victim.load() == thread_id);
}
```

```
void unlock() {}
```

Thread 0:
`m.lock();`
`m.unlock();`

Thread 1:
`m.lock();`
`m.unlock();`



Mutex Implementations

Finally, we can can make a mutex that works:

Use flags to mark interest

Use victim to break ties

Called the **Peterson Lock**

Mutex Implementations

```
class Mutex {
public:
  Mutex() {
    victim = -1;
    flag[0] = flag[1] = 0;
  }

  void lock();
  void unlock();


private:
  atomic_int victim;
  atomic_bool flag[2];
};
```

No victim and no threads are interested in the critical section

flags and victim

Mutex Implementations

```
void lock() {
  int j = thread_id == 0 ? 1 : 0;
  flag[thread_id].store(1);
  victim.store(thread_id);
  while (victim.load() == thread_id
         && flag[j] == 1);
}
```

j is the other thread

Mark ourself as interested

volunteer to be the victim in case of a tie

Spin only if:
there was a tie in wanting the lock,
and I won the volunteer raffle to spin

Mutex Implementations

```
void unlock() {
  int i = thread_id;
  flag[i].store(0);
}
```

mark ourselves as uninterested

previous flag issue

```
void lock() {
  int i = thread_id;
  flag[i].store(1);
  int j = i == 0 ? 1 : 0;
  while (flag[j].load() == 1);
}
```

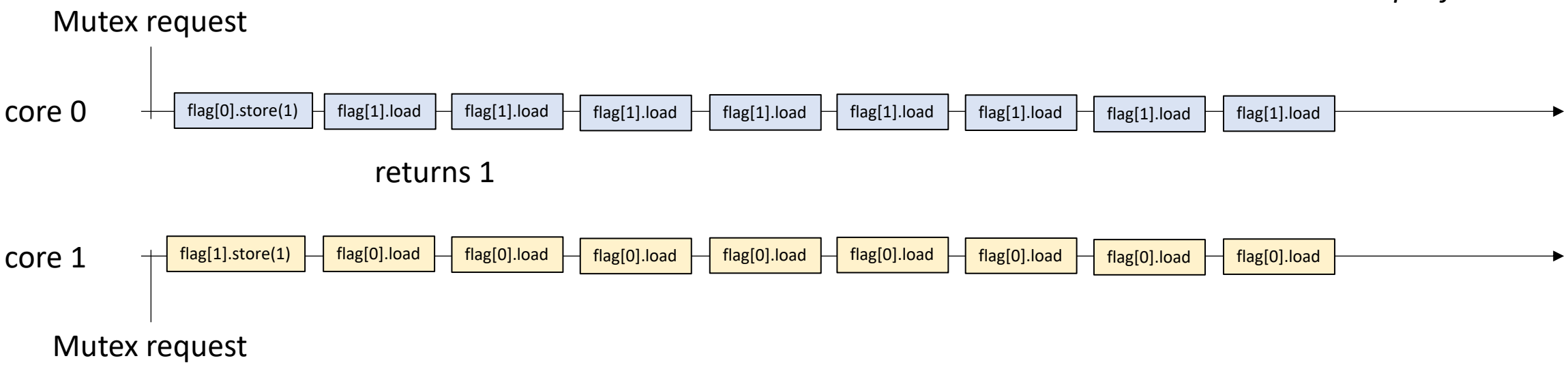
```
void unlock() {
  int i = thread_id;
  flag[i].store(0);
}
```

Thread 0:
`m.lock();`
`m.unlock();`

Thread 1:
`m.lock();`
`m.unlock();`

how does petersons solve this?

Both will spin forever!



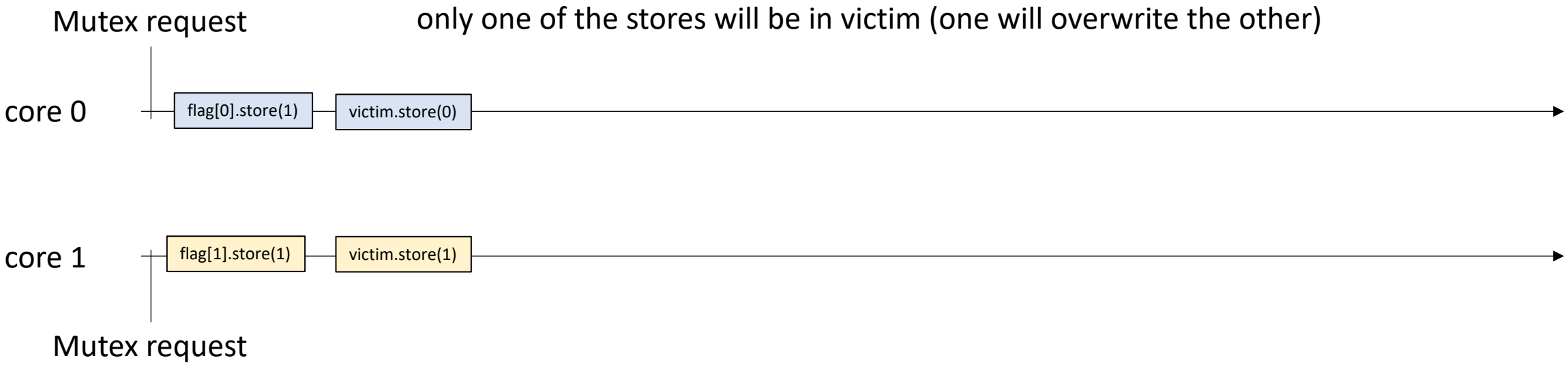
Tie breaking with victim

```
void lock() {
  int j = thread_id == 0 ? 1 : 0;
  flag[thread_id].store(1);
  victim.store(thread_id);
  while (victim.load() == thread_id
         && flag[j] == 1);
}
```

```
void unlock() {
  int i = thread_id;
  flag[i].store(0);
}
```

Thread 0:
```
m.lock();
m.unlock();
```

Thread 1:
```
m.lock();
m.unlock();
```



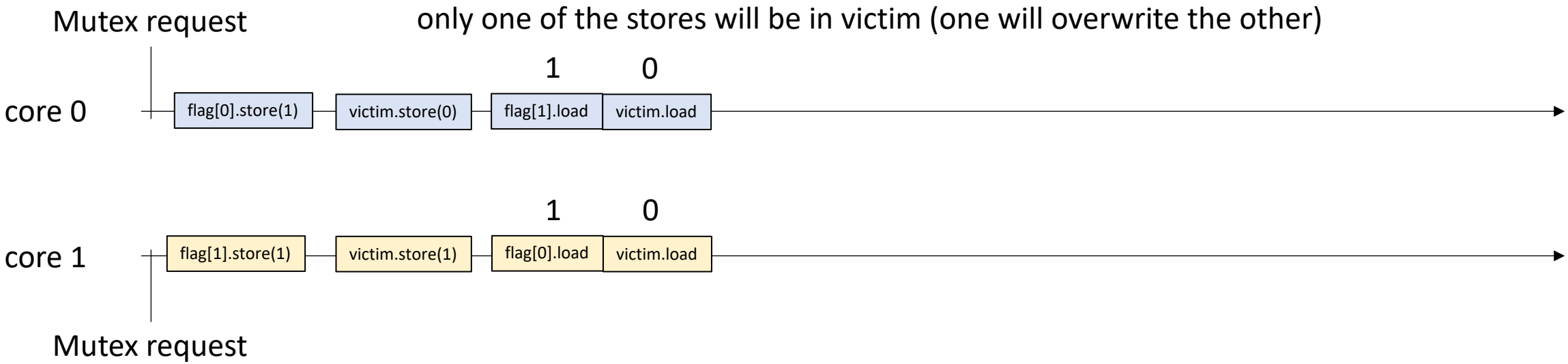
Tie breaking with victim

```
void lock() {
  int j = thread_id == 0 ? 1 : 0;
  flag[thread_id].store(1);
  victim.store(thread_id);
  while (victim.load() == thread_id
         && flag[j] == 1);
}
```

```
void unlock() {
  int i = thread_id;
  flag[i].store(0);
}
```

Thread 0:
```
m.lock();
m.unlock();
```

Thread 1:
```
m.lock();
m.unlock();
```



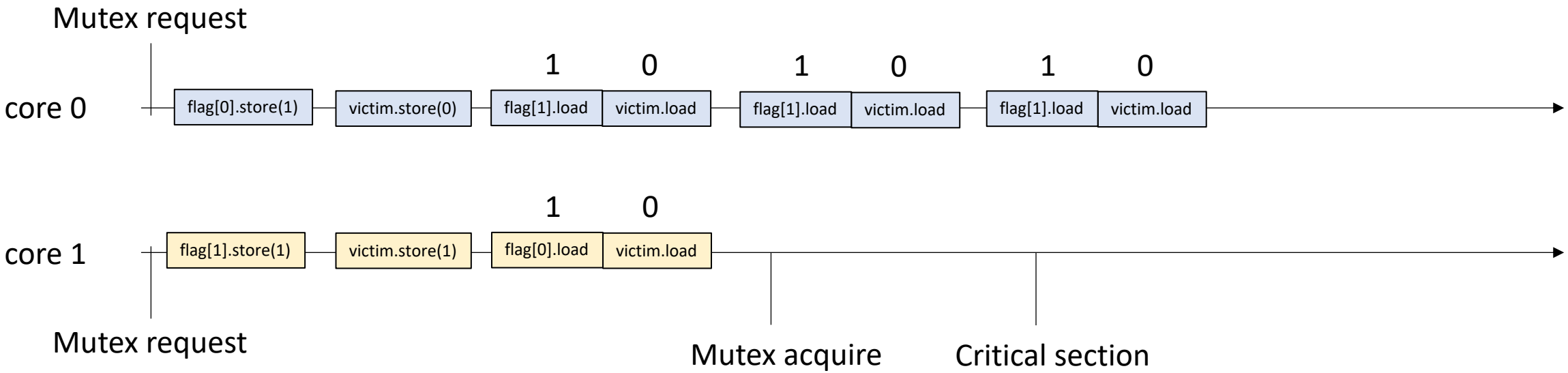
Tie breaking with victim

```
void lock() {
  int j = thread_id == 0 ? 1 : 0;
  flag[thread_id].store(1);
  victim.store(thread_id);
  while (victim.load() == thread_id
         && flag[j] == 1);
}
```

```
void unlock() {
  int i = thread_id;
  flag[i].store(0);
}
```

Thread 0:
```
m.lock();
m.unlock();
```

Thread 1:
```
m.lock();
m.unlock();
```



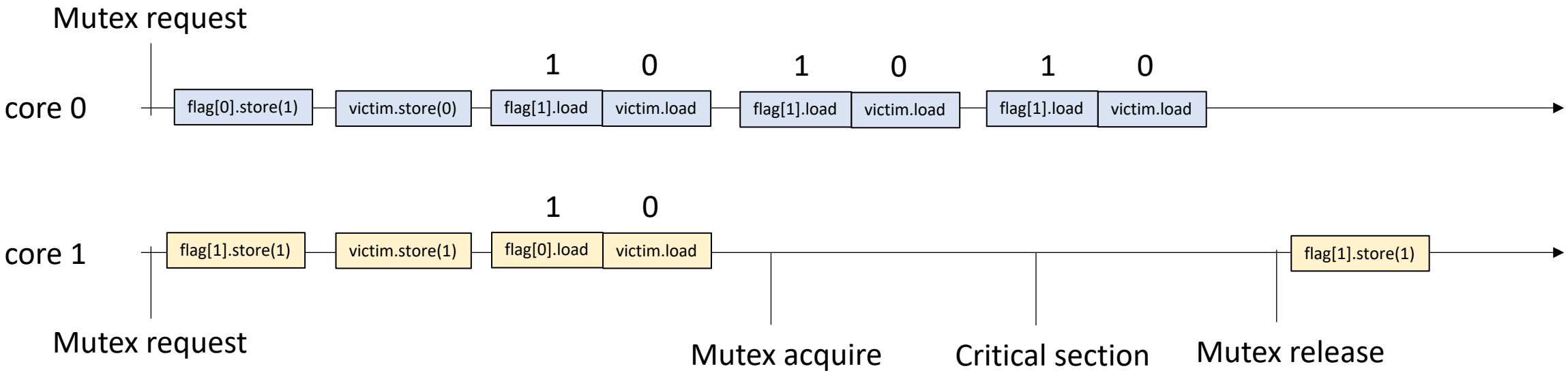
Tie breaking with victim

```
void lock() {
  int j = thread_id == 0 ? 1 : 0;
  flag[thread_id].store(1);
  victim.store(thread_id);
  while (victim.load() == thread_id
         && flag[j] == 1);
}
```

```
void unlock() {
  int i = thread_id;
  flag[i].store(0);
}
```

Thread 0:
```
m.lock();
m.unlock();
```

Thread 1:
```
m.lock();
m.unlock();
```



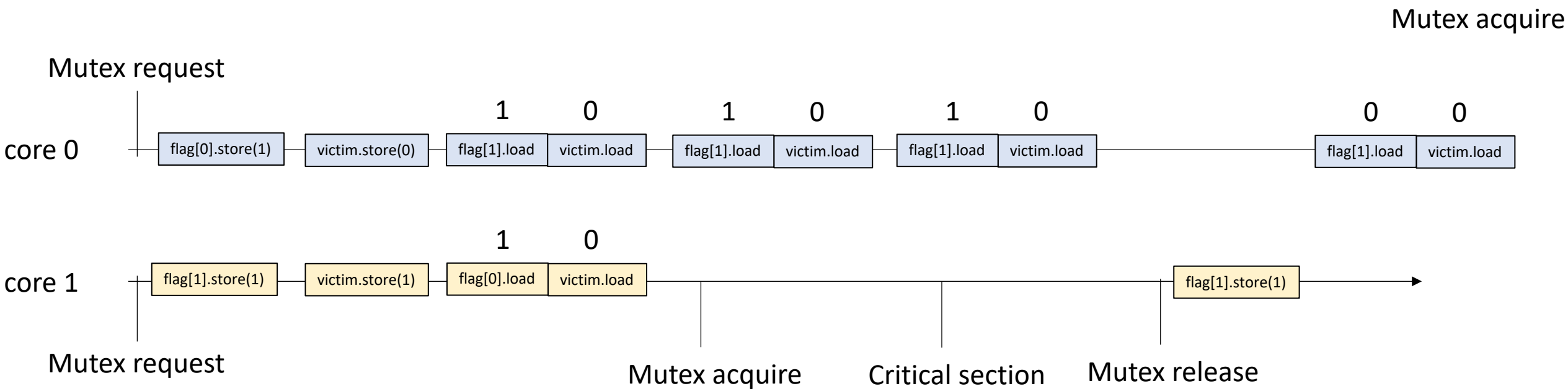
Tie breaking with victim

```
void lock() {
  int j = thread_id == 0 ? 1 : 0;
  flag[thread_id].store(1);
  victim.store(thread_id);
  while (victim.load() == thread_id
         && flag[j] == 1);
}
```

```
void unlock() {
  int i = thread_id;
  flag[i].store(0);
}
```

Thread 0:
```
m.lock();
m.unlock();
```

Thread 1:
```
m.lock();
m.unlock();
```



previous victim issue

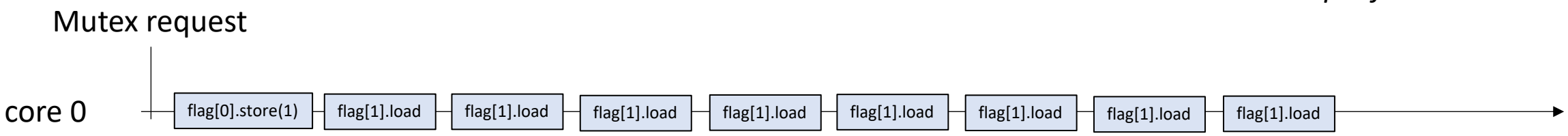
```
void lock() {
  victim.store(thread_id);
  while (victim.load() == thread_id);
}
```

```
void unlock() {}
```

Thread 0:

```
m.lock();
m.unlock();
```

will spin forever!



previous flag issue

```
void lock() {
  int j = thread_id == 0 ? 1 : 0;
  flag[thread_id].store(1);
  victim.store(thread_id);
  while (victim.load() == thread_id
         && flag[j] == 1);
}
```

```
void unlock() {
  int i = thread_id;
  flag[i].store(0);
}
```

Thread 0:

```
m.lock();
m.unlock();
```

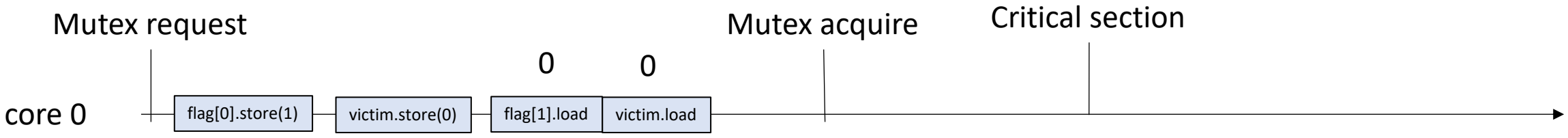
previous flag issue

```
void lock() {
  int j = thread_id == 0 ? 1 : 0;
  flag[thread_id].store(1);
  victim.store(thread_id);
  while (victim.load() == thread_id
         && flag[j] == 1);
}
```

```
void unlock() {
  int i = thread_id;
  flag[i].store(0);
}
```

Thread 0:

```
m.lock();
m.unlock();
```



we can enter critical section because the other thread isn't interested

```
void lock() {
  int j = thread_id == 0 ? 1 : 0;
  flag[thread_id].store(1);
  victim.store(thread_id);
  while (victim.load() == thread_id
         && flag[j] == 1);
}
```

```
void unlock() {
  int i = thread_id;
  flag[i].store(0);
}
```

Does it satisfy mutual exclusion?

Proof by contradiction sketch

```
void lock() {
  int j = thread_id == 0 ? 1 : 0;
  flag[thread_id].store(1);
  victim.store(thread_id);
  while (victim.load() == thread_id
         && flag[j] == 1);
}
```

```
void unlock() {
  int i = thread_id;
  flag[i].store(0);
}
```

Does it satisfy mutual exclusion?

Proof by contradiction sketch

Assume C0 and C1 are both in the critical section. That means both of them broke out of the while loop

what we know:

flag[0] is 1

flag[1] is 1

```
void lock() {
  int j = thread_id == 0 ? 1 : 0;
  flag[thread_id].store(1);
  victim.store(thread_id);
  while (victim.load() == thread_id
         && flag[j] == 1);
}
```

```
void unlock() {
  int i = thread_id;
  flag[i].store(0);
}
```

Does it satisfy mutual exclusion?

Proof by contradiction sketch

Assume C0 and C1 are both in the critical section. That means both of them broke out of the while loop

We know from the flag line that both flags are set to 1.

what we know:

flag[0] is 1

flag[1] is 1

```
void lock() {
  int j = thread_id == 0 ? 1 : 0;
  flag[thread_id].store(1);
  victim.store(thread_id);
  while (victim.load() == thread_id
         && flag[j] == 1);
}
```

```
void unlock() {
  int i = thread_id;
  flag[i].store(0);
}
```

Does it satisfy mutual exclusion?

Proof by contradiction sketch

Assume C0 and C1 are both in the critical section. That means both of them broke out of the while loop

We know from the flag line that both flags are set to 1.

We know from the victim line that the victim must be equal to one of the thread ids

what we know:

flag[0] is 1

flag[1] is 1

```
void lock() {
  int j = thread_id == 0 ? 1 : 0;
  flag[thread_id].store(1);
  victim.store(thread_id);
  while (victim.load() == thread_id
         && flag[j] == 1);
}
```

```
void unlock() {
  int i = thread_id;
  flag[i].store(0);
}
```

Does it satisfy mutual exclusion?

Proof by contradiction sketch

Assume C0 and C1 are both in the critical section. That means both of them broke out of the while loop

We know from the flag line that both flags are set to 1.

We know from the victim line that the victim must be equal to one of the thread ids

For thread 0 to be in critical section, Thread 1 must have written victim while Thread 0 was spinning
But then Thread 1 would be spinning (contradiction)

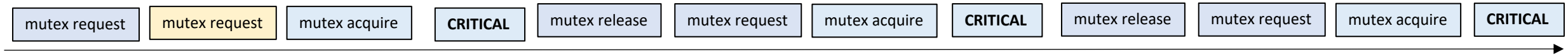
Vice Versa

What about starvation

recall the starvation property:

Thread 1 (yellow) requests the mutex but never gets it

concurrent execution

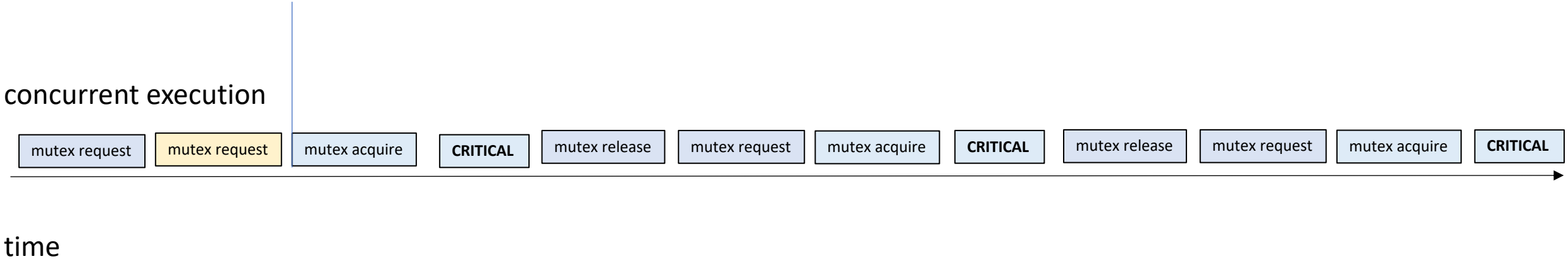


time

What about starvation

```
void lock() {
  int j = thread_id == 0 ? 1 : 0;
  flag[thread_id].store(1);
  victim.store(thread_id);
  while (victim.load() == thread_id
         && flag[j] == 1);
}
```

at this point, C1 is the victim and is spinning

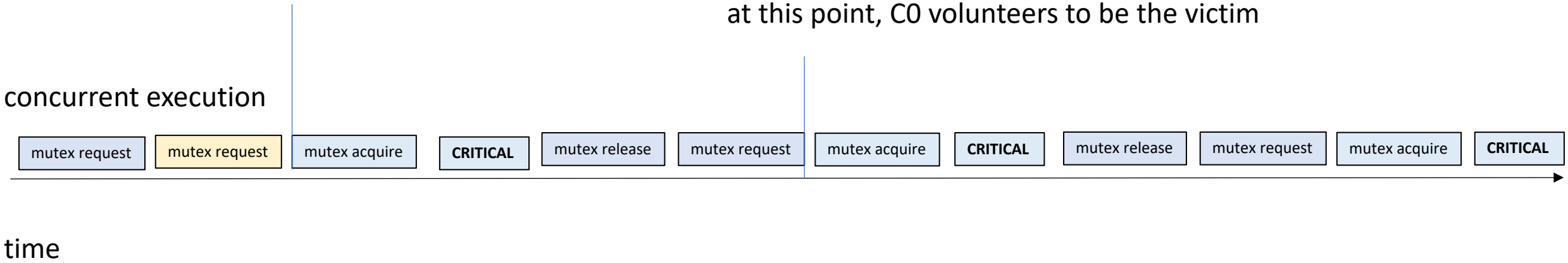


What about starvation

```
void lock() {
  int j = thread_id == 0 ? 1 : 0;
  flag[thread_id].store(1);
  victim.store(thread_id);
  while (victim.load() == thread_id
         && flag[j] == 1);
}
```

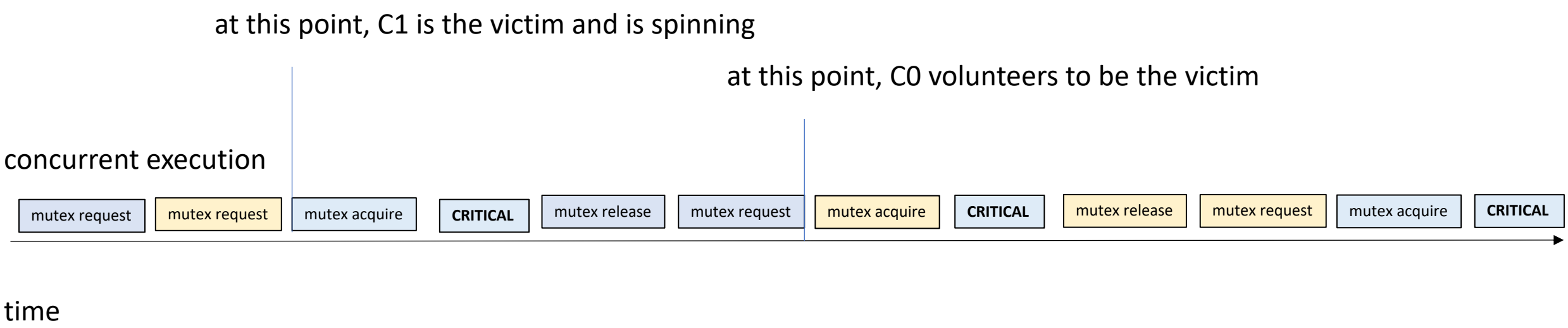
at this point, C1 is the victim and is spinning

at this point, C0 volunteers to be the victim



What about starvation

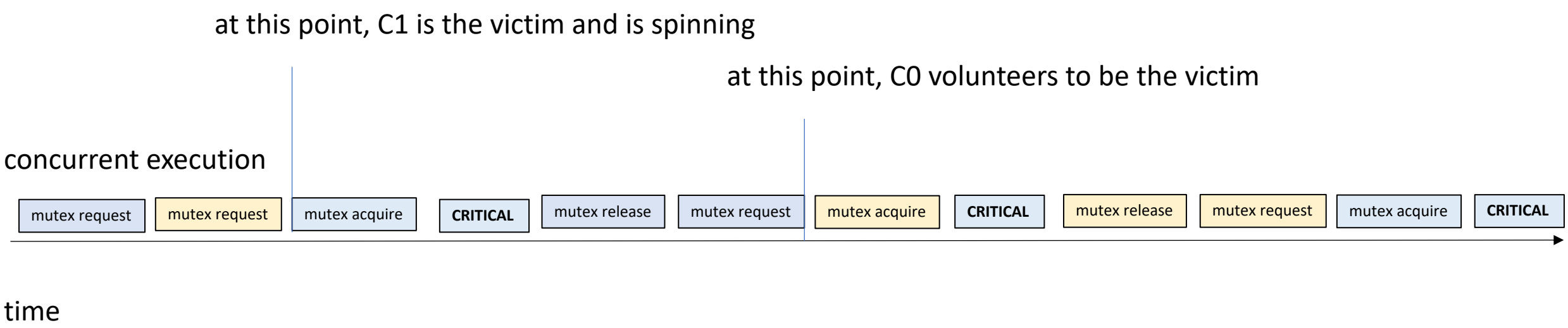
```
void lock() {
  int j = thread_id == 0 ? 1 : 0;
  flag[thread_id].store(1);
  victim.store(thread_id);
  while (victim.load() == thread_id
         && flag[j] == 1);
}
```



What about starvation

Threads take turns in petersons algorithm. It is starvation free

```
void lock() {
  int j = thread_id == 0 ? 1 : 0;
  flag[thread_id].store(1);
  victim.store(thread_id);
  while (victim.load() == thread_id
         && flag[j] == 1);
}
```



Mutex Implementations

Peterson only works with 2 threads.

Generalizes to the Filter Lock (Read chapter 2 in the book)

So it works!

Now what about performance

Phew....

- Lots of thinking about implementations for today!
- RMWs make lock implementations much simpler
  - And more performant.
- We will do those next week

Next week

- How do we make our mutexes easier to reason about and faster?
  - Atomic RMWs
  - Backoff
  - Thread Sanitizer