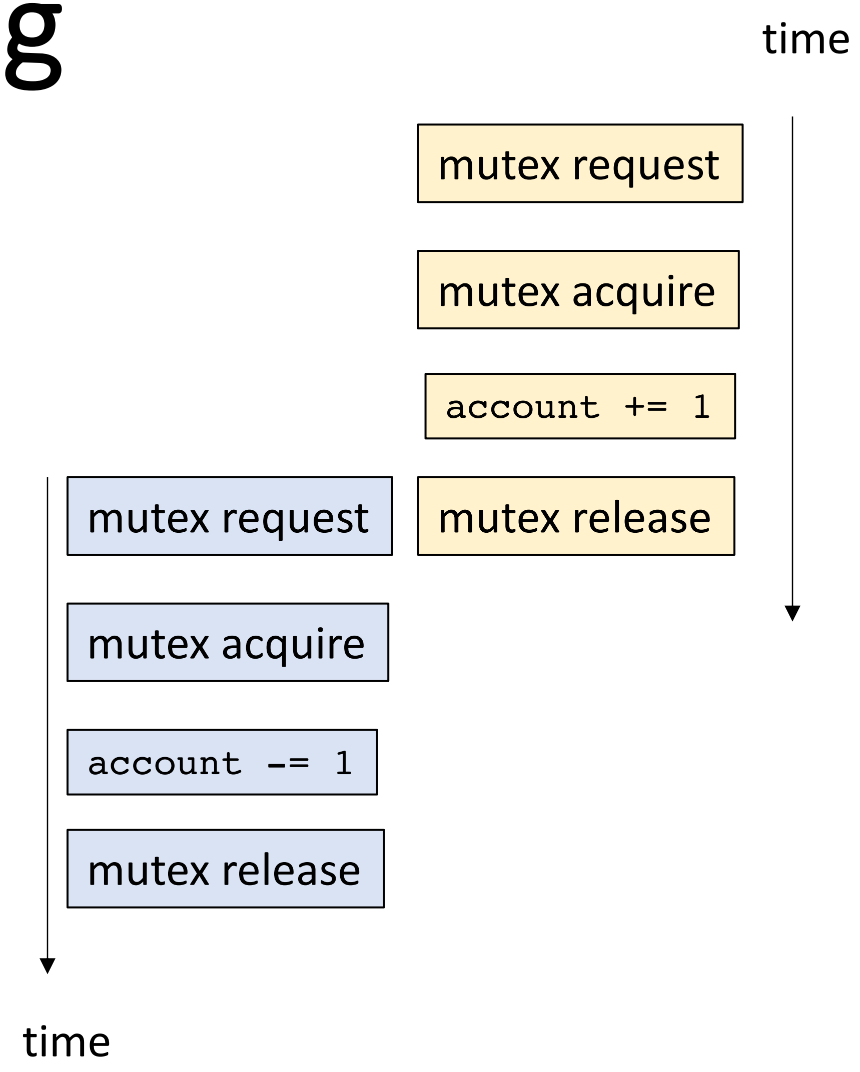
CSE113: Parallel Programming

Jan. 30, 2023

- **Topics**:
  - Mutual exclusion examples
  - Multiple mutexes
  - Mutex properties
  - Atomic operation properties



Announcements

- Hope everyone enjoyed the guest lectures by Devon and Jessica!

Announcements

- Final late day of HW 1 is today
    - No late submissions accepted after today at midnight
    - Some office hours if you need some last minute help
    - We will try to answer questions asked before 5 PM, but no guarantees afterwards.
- HW 2 is planned to be released by midnight tonight
    - Same due date structure:
    - due in 10 days
    - 4 late days if needed
- You can start doing part 1 of HW 2 after today's lecture
    - At least do the reading in the book

Quiz review

The advantage GPU accelerators provide over CPUs is?...

- ○ fewer cores, but with higher logical complexity
- ○ access to many cores, with less control over individual cores
- ○ cores with faster clock speeds
- ○ makes your system more expensive

Picking up on mutexes:

Programming with mutexes can be HARD!

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

Towards Implementations

Properties of mutexes

Three properties

- **Mutual exclusion** - Only 1 thread can hold the mutex at a time. Critical sections cannot interleave

Other threads are allowed to request, but not acquire until the thread that has acquired the mutex releases it.

concurrent execution

disallowed!

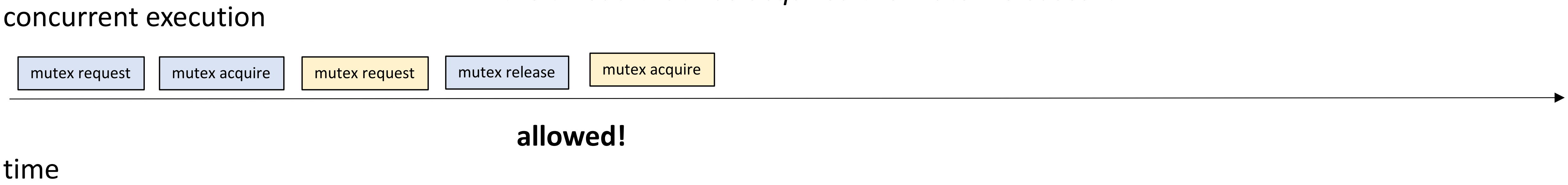
time

Properties of mutexes

Three properties

- **Mutual exclusion** - Only 1 thread can hold the mutex at a time. Critical sections cannot interleave

Other threads are allowed to request, but not acquire until the thread that has acquired the mutex releases it.



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

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

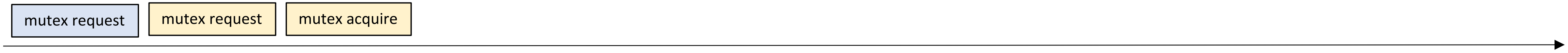
Properties of mutexes

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

concurrent execution



allowed

time

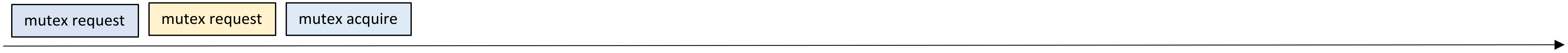
Properties of mutexes

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

concurrent execution



also allowed

time

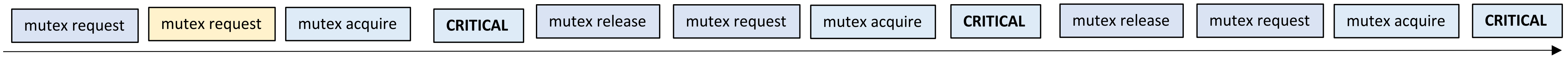
Properties of mutexes

Three properties

- **Starvation Freedom** (*Optional*) - A thread that requests the mutex must eventually obtain the mutex.

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

concurrent execution



time

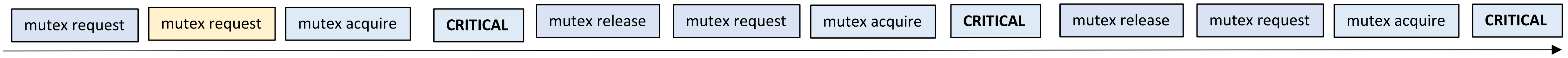
Properties of mutexes

Three properties

- **Starvation Freedom** (*Optional*) - A thread that requests the mutex must eventually obtain the mutex.

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

concurrent execution



time

Difficult to provide in practice and timing variations usually provide this property naturally

Properties of mutexes

Recap: three properties

- **Mutual Exclusion**: Two threads cannot be in the critical section at the same time
- **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
- **Starvation Freedom** (*optional*): A thread that requests the mutex must eventually obtain the mutex.

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 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_int

- Interface (C++ provides 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

```
int foo(int x) {
  x = 0;
  for (int i = 0; i < 2048; i++) {
    x++;
  }
  return x;
}
```

```
int foo(atomic x) {
  x.store(0);
  for (int i = 0; i < 2048; i++) {
    int tmp = x.load();
    tmp++;
    x.store(tmp);
  }
  return x.load();
}
```

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

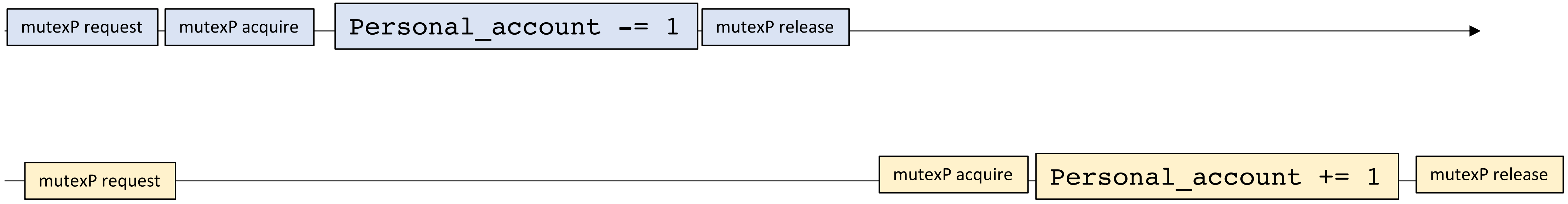
- Compiler can be aggressive with memory operations:
  - For non-atomic memory locations, the following optimizations are valid

- 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

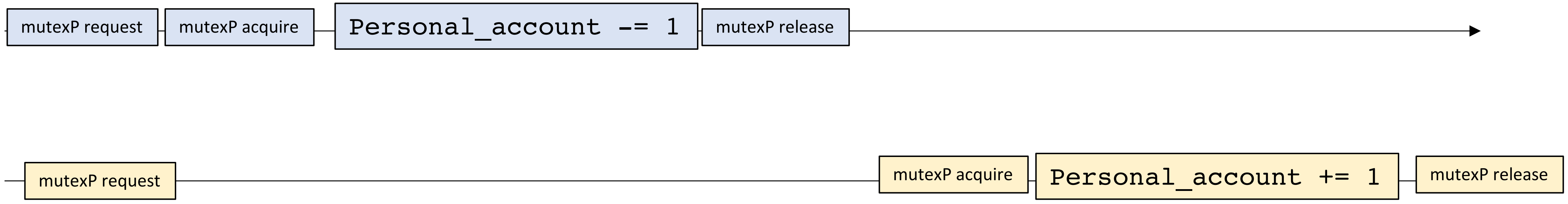


because this thread needs to see the updated view

Atomic properties

- Compiler Fence

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

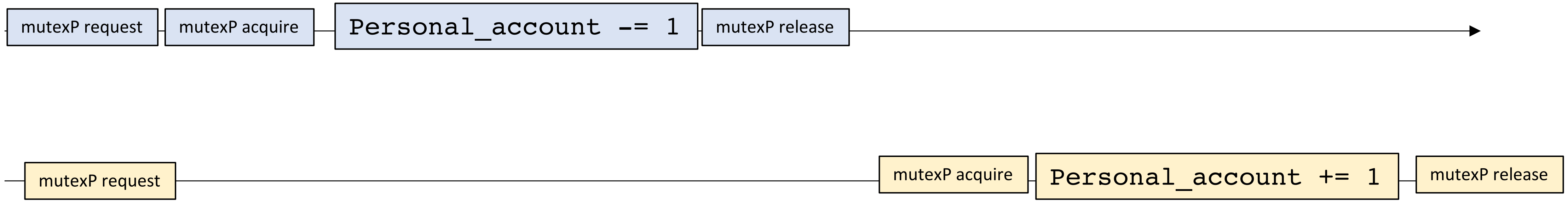


Atomic properties

- Compiler Fence

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

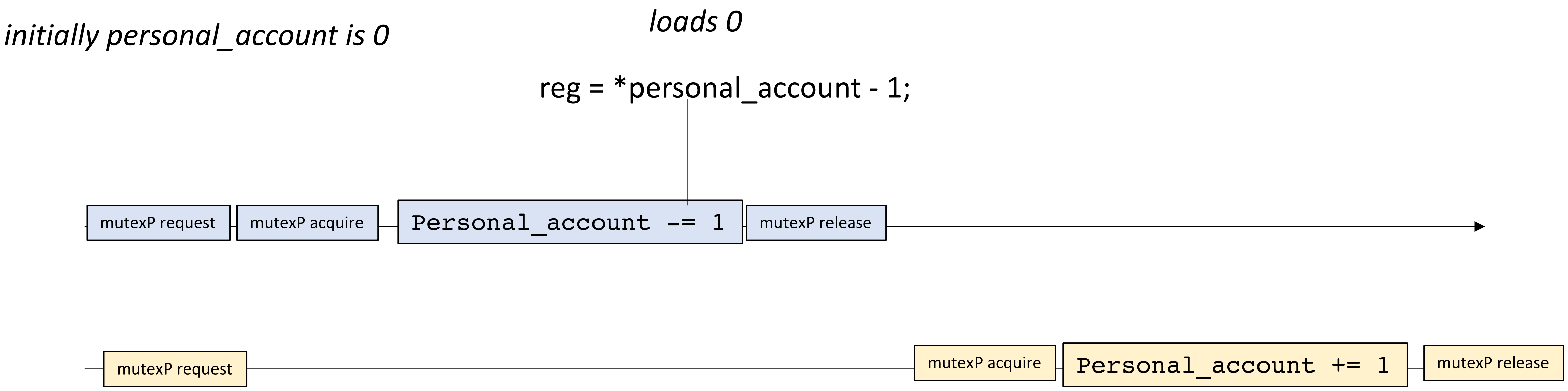
initially personal_account is 0



Atomic properties

- Compiler Fence

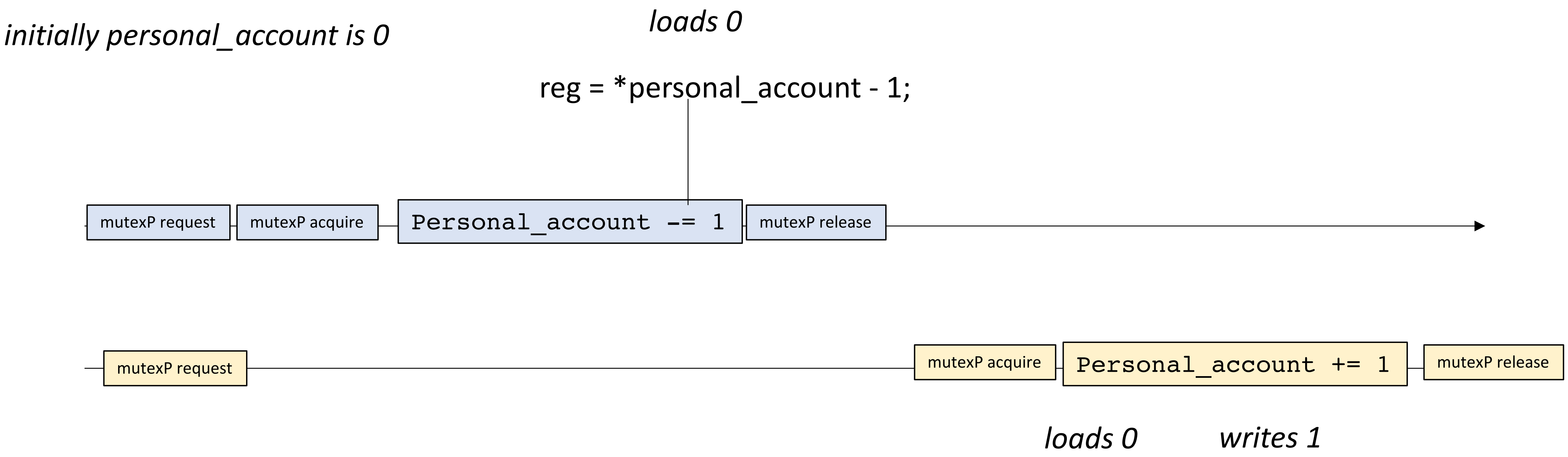
what can go wrong if the compiler doesn't write values to memory?



Atomic properties

- Compiler Fence

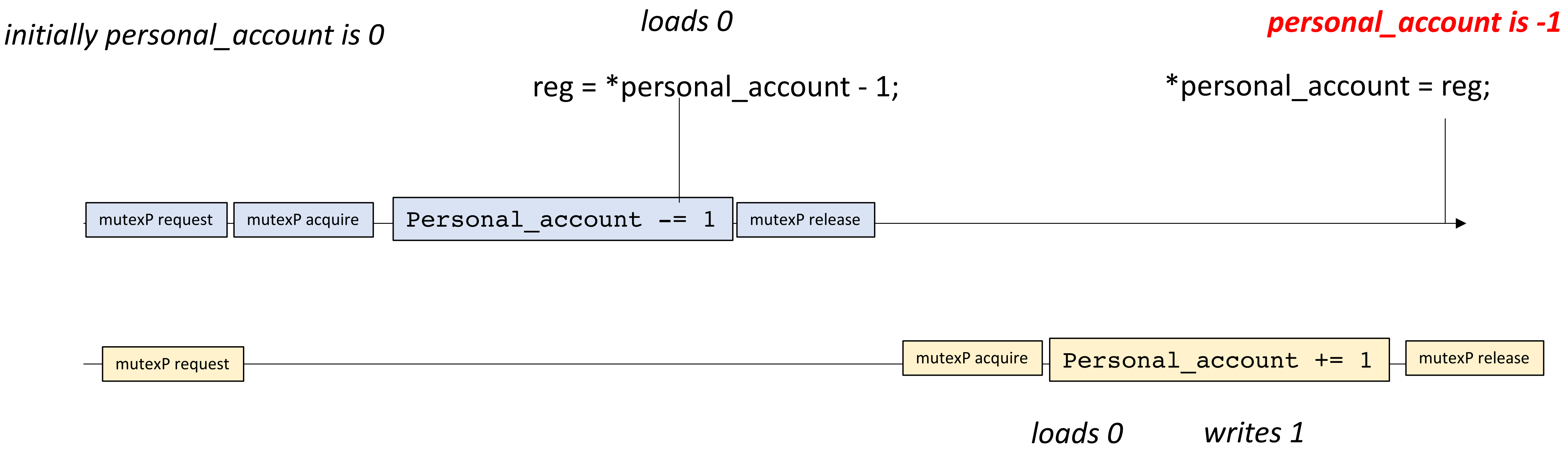
what can go wrong if the compiler doesn't write values to memory?



Atomic properties

- Compiler Fence

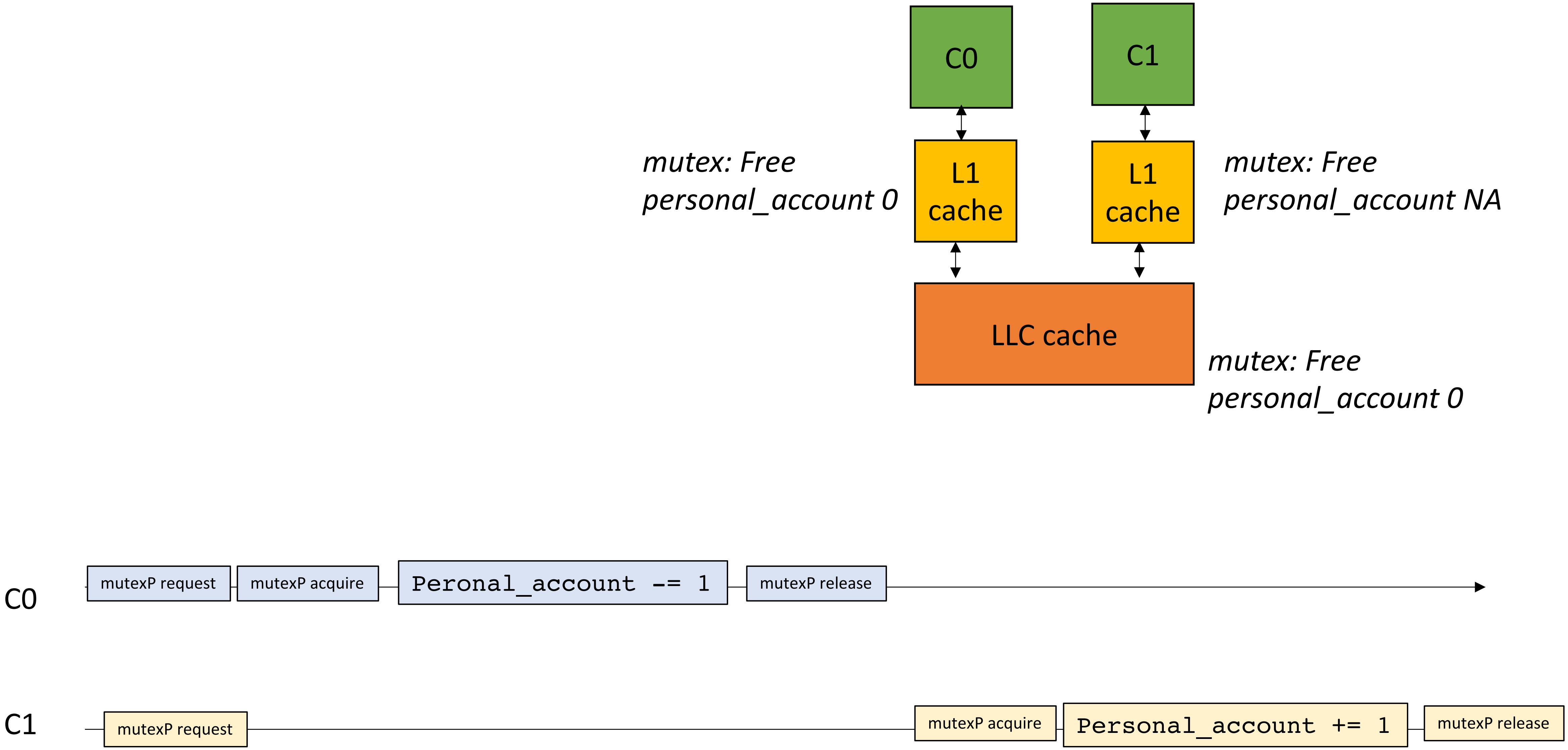
what can go wrong if the compiler doesn't write values to memory?



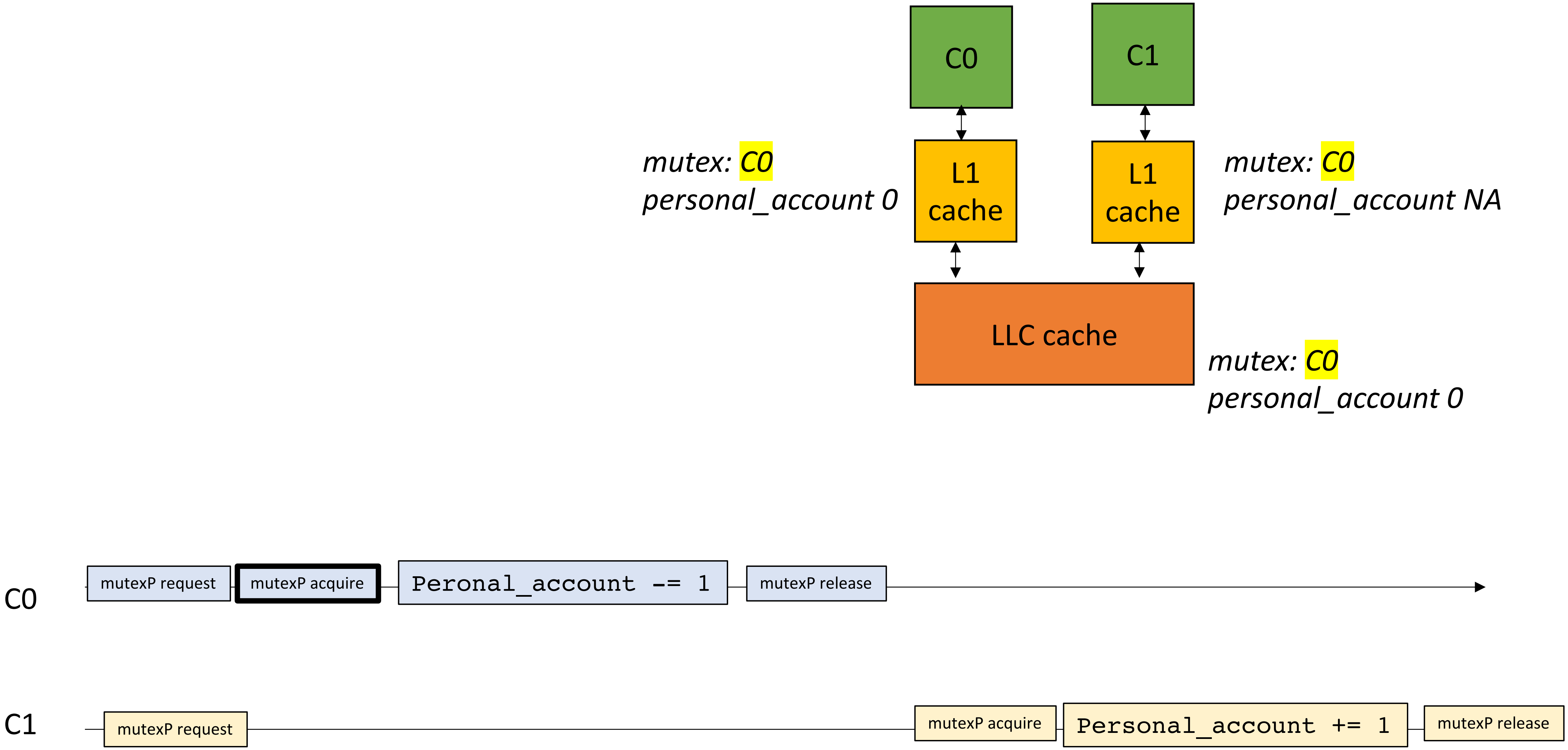
Atomic properties

- Also provides a memory barrier

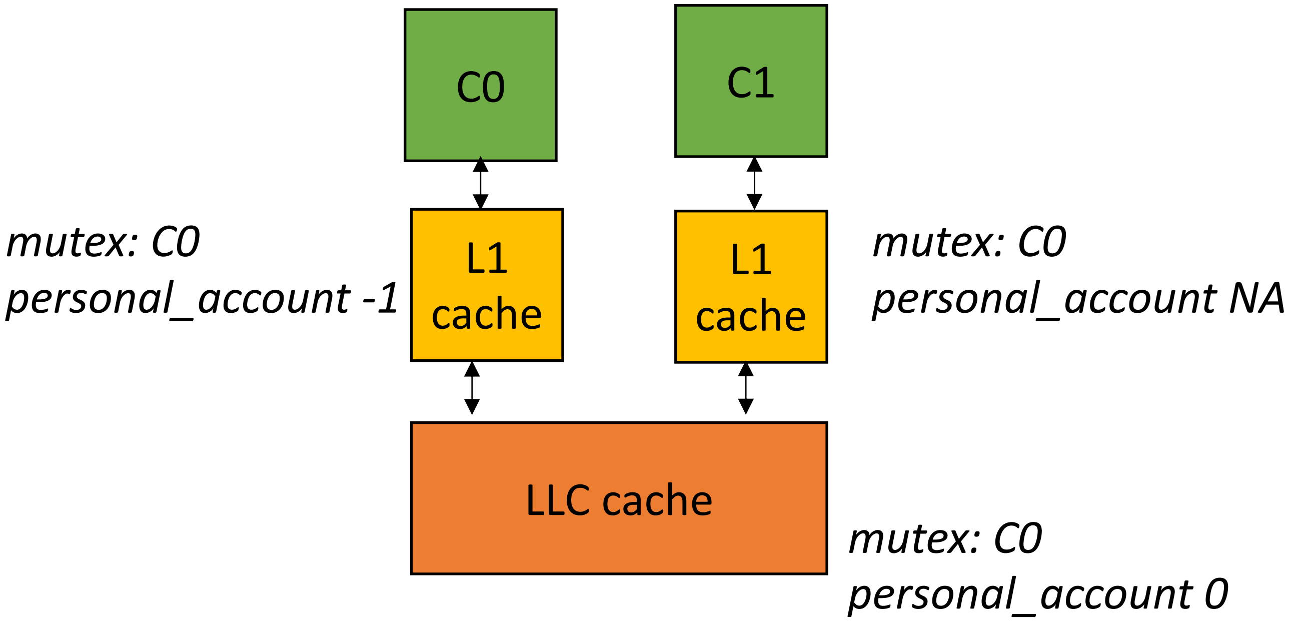
- Memory Fence (or Memory Barrier)

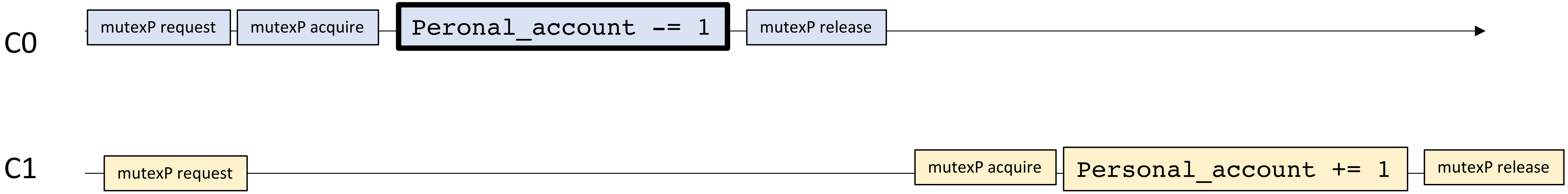


- Memory Fence (or Memory Barrier)

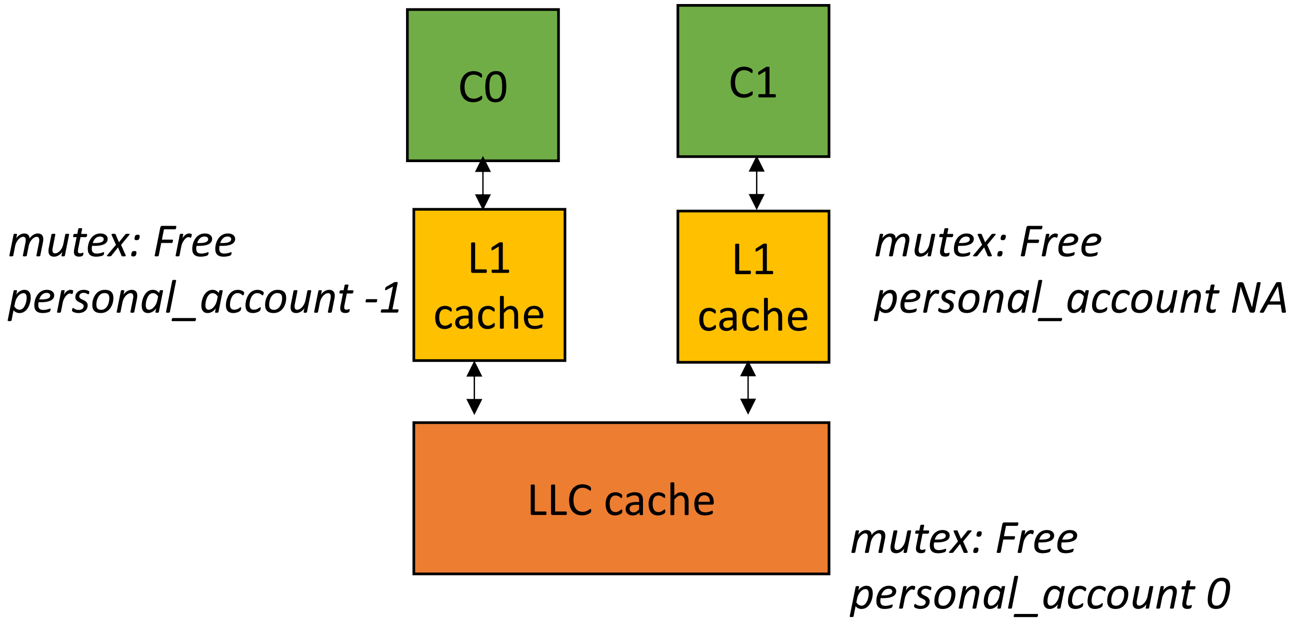


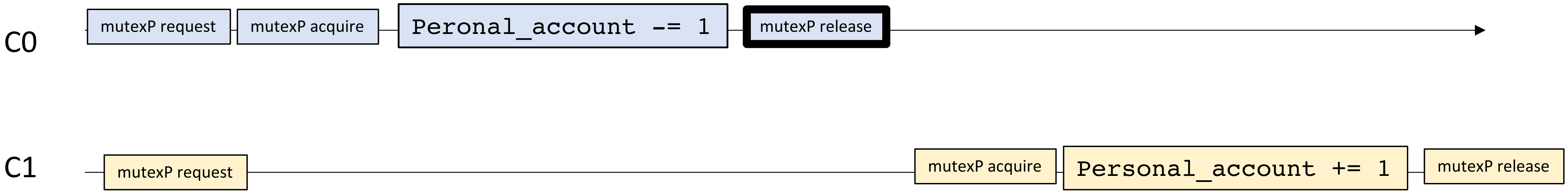
- Memory Fence (or Memory Barrier)



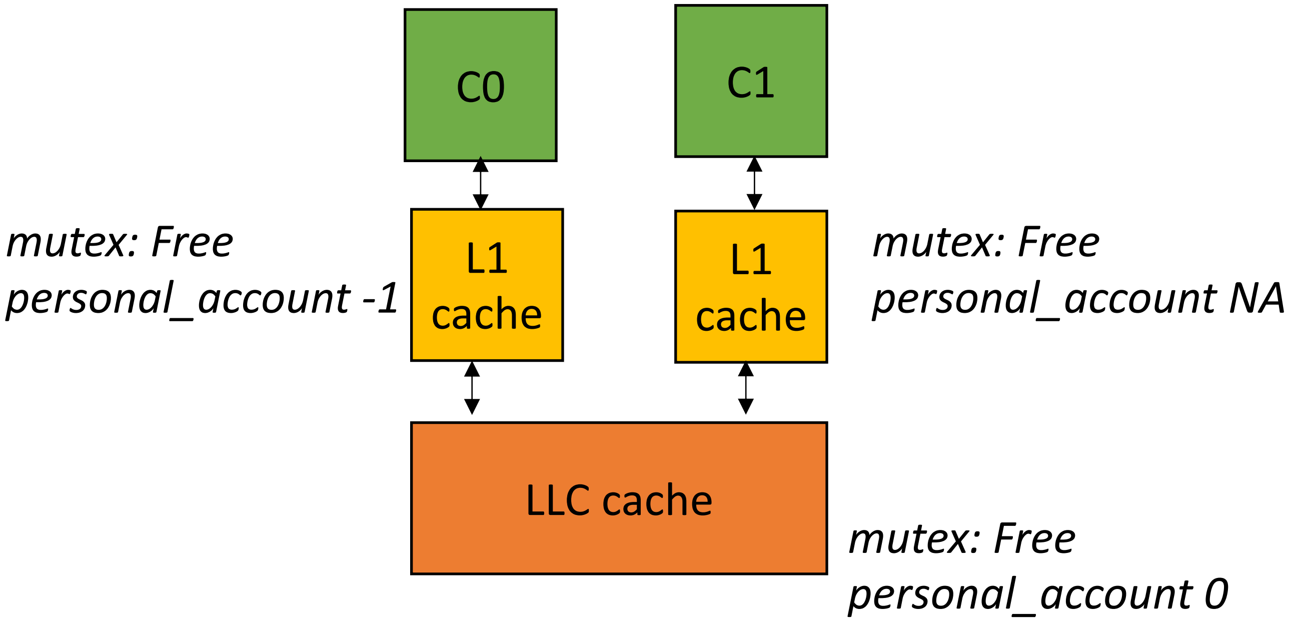


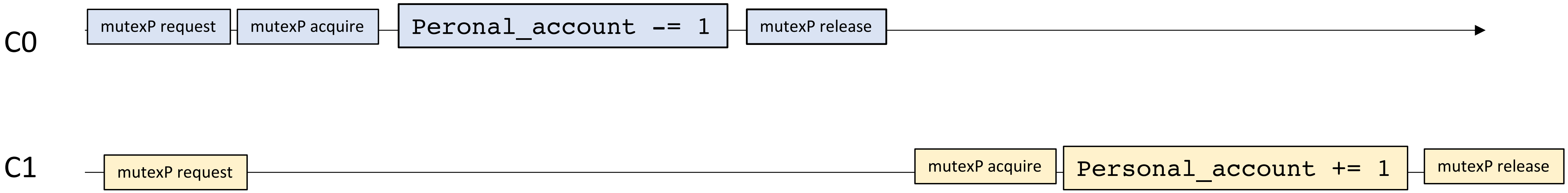
- Memory Fence (or Memory Barrier)



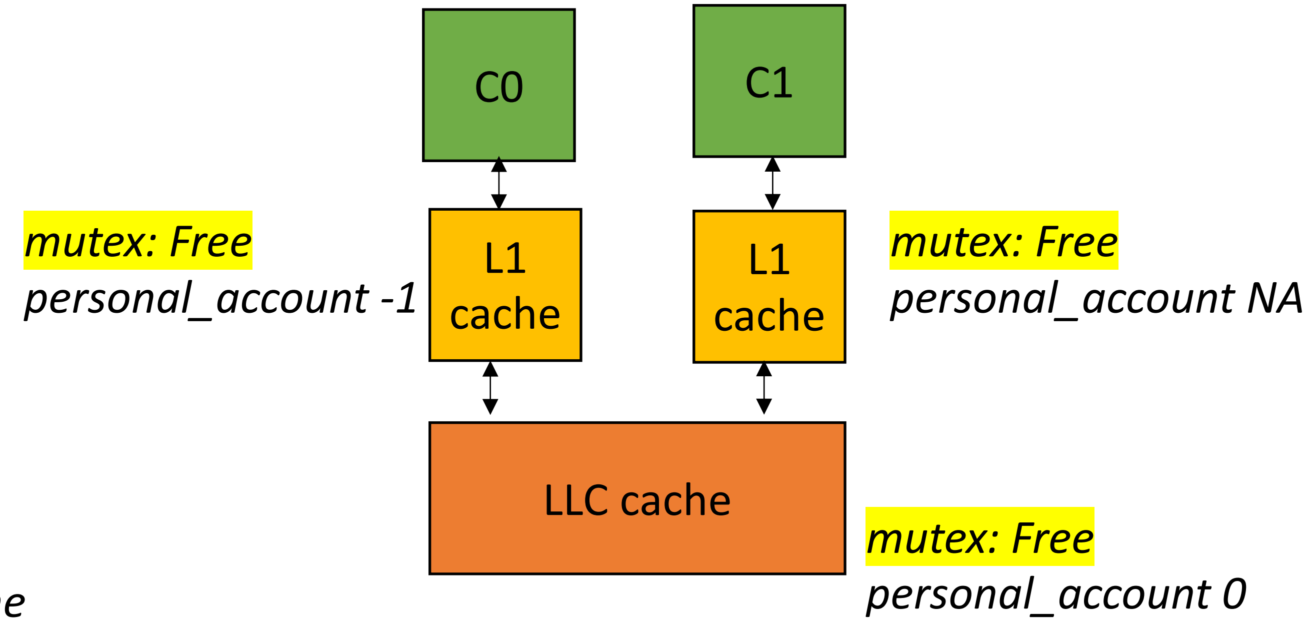


- Memory Fence (or Memory Barrier)

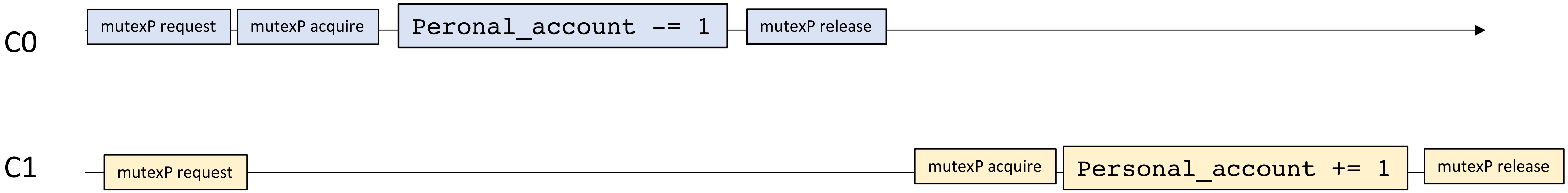




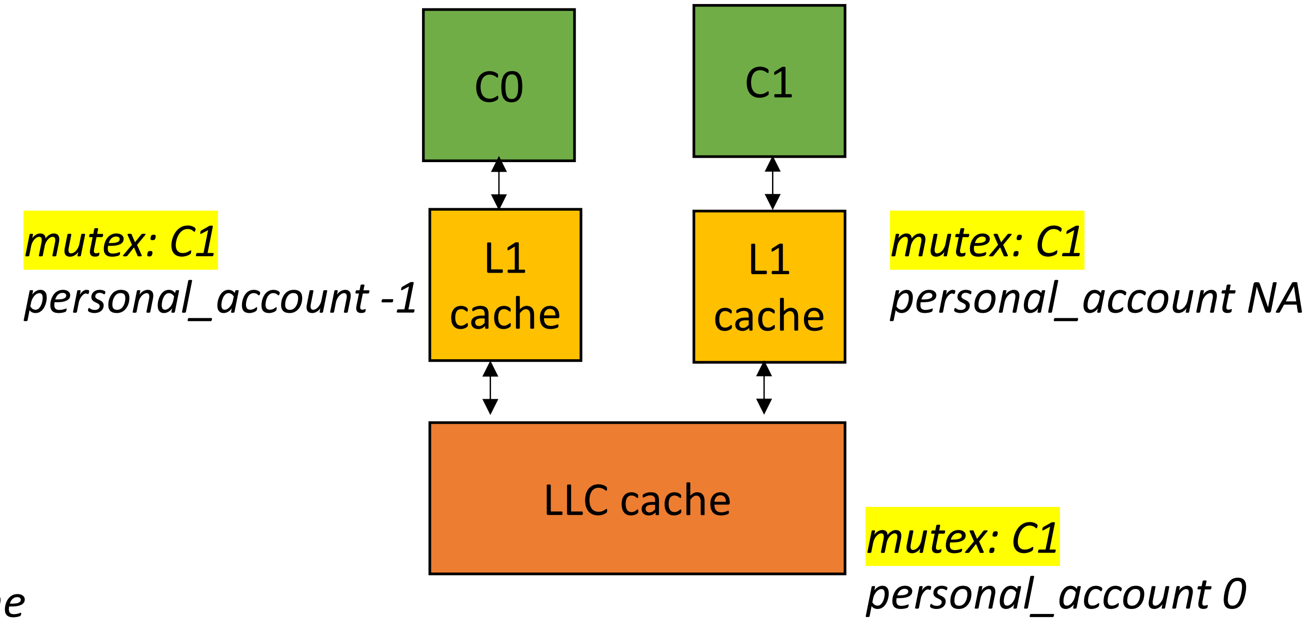
- Memory Fence (or Memory Barrier)



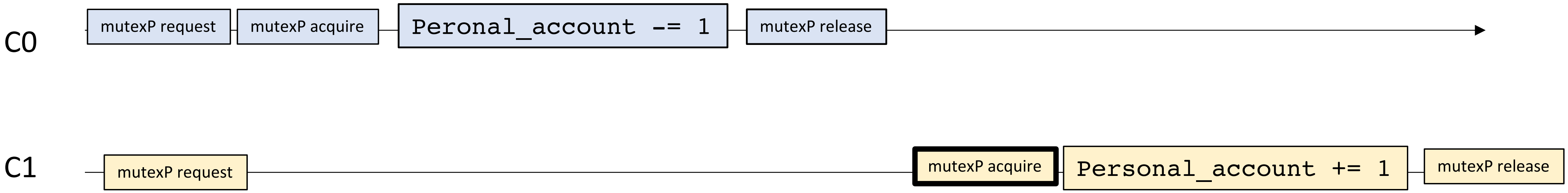
Possible for the mutex message to travel through the cache before the personal_account information!



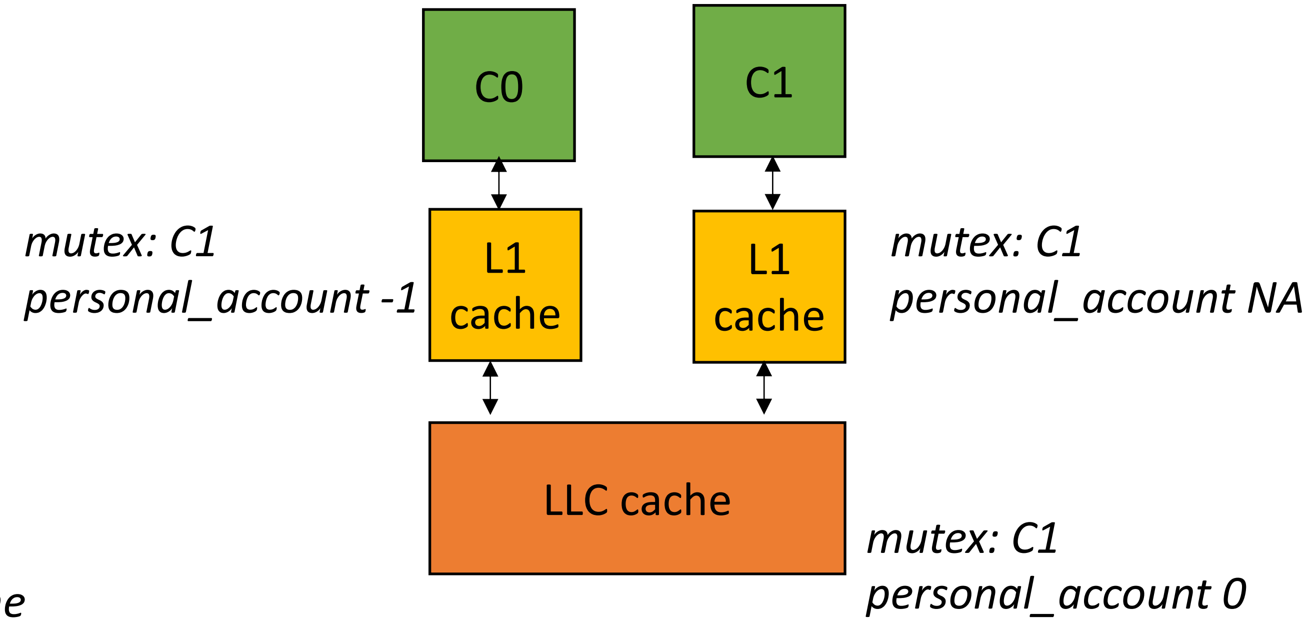
- Memory Fence (or Memory Barrier)



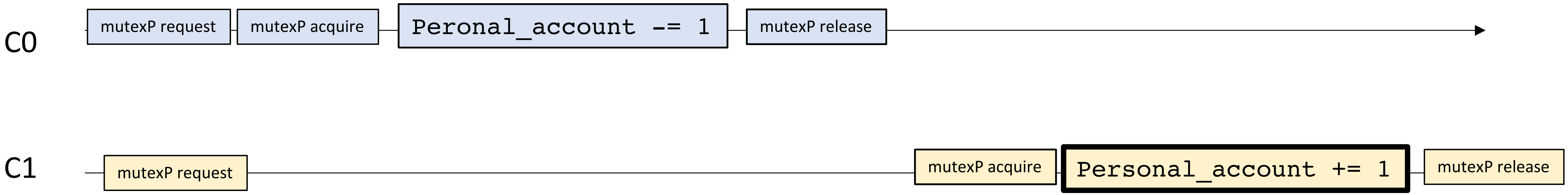
Possible for the mutex message to travel through the cache before the personal_account information!



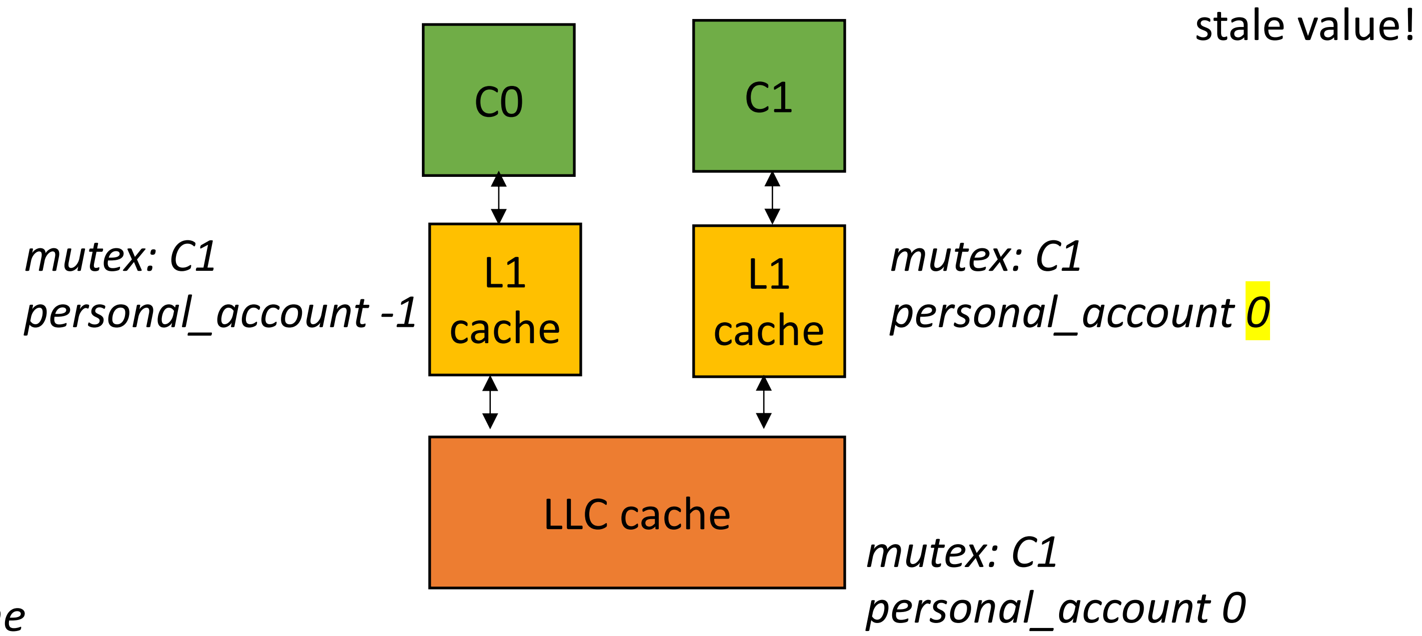
- Memory Fence (or Memory Barrier)



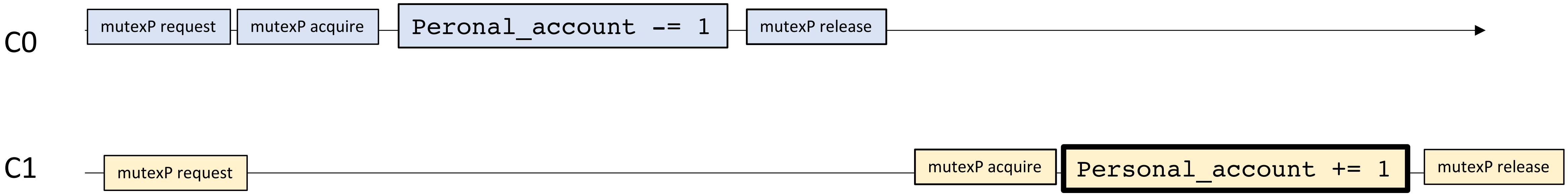
Possible for the mutex message to travel through the cache before the personal_account information!



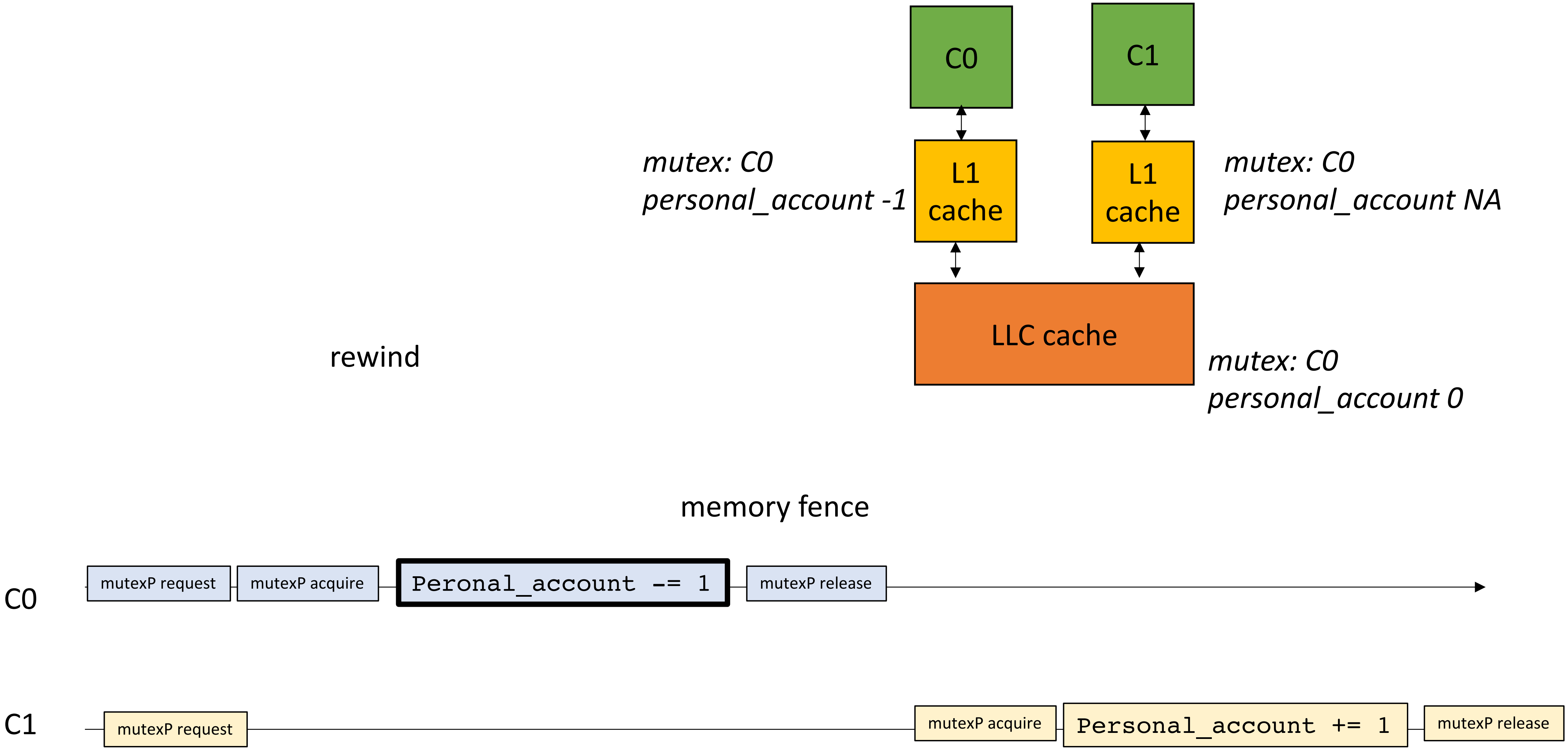
- Memory Fence (or Memory Barrier)



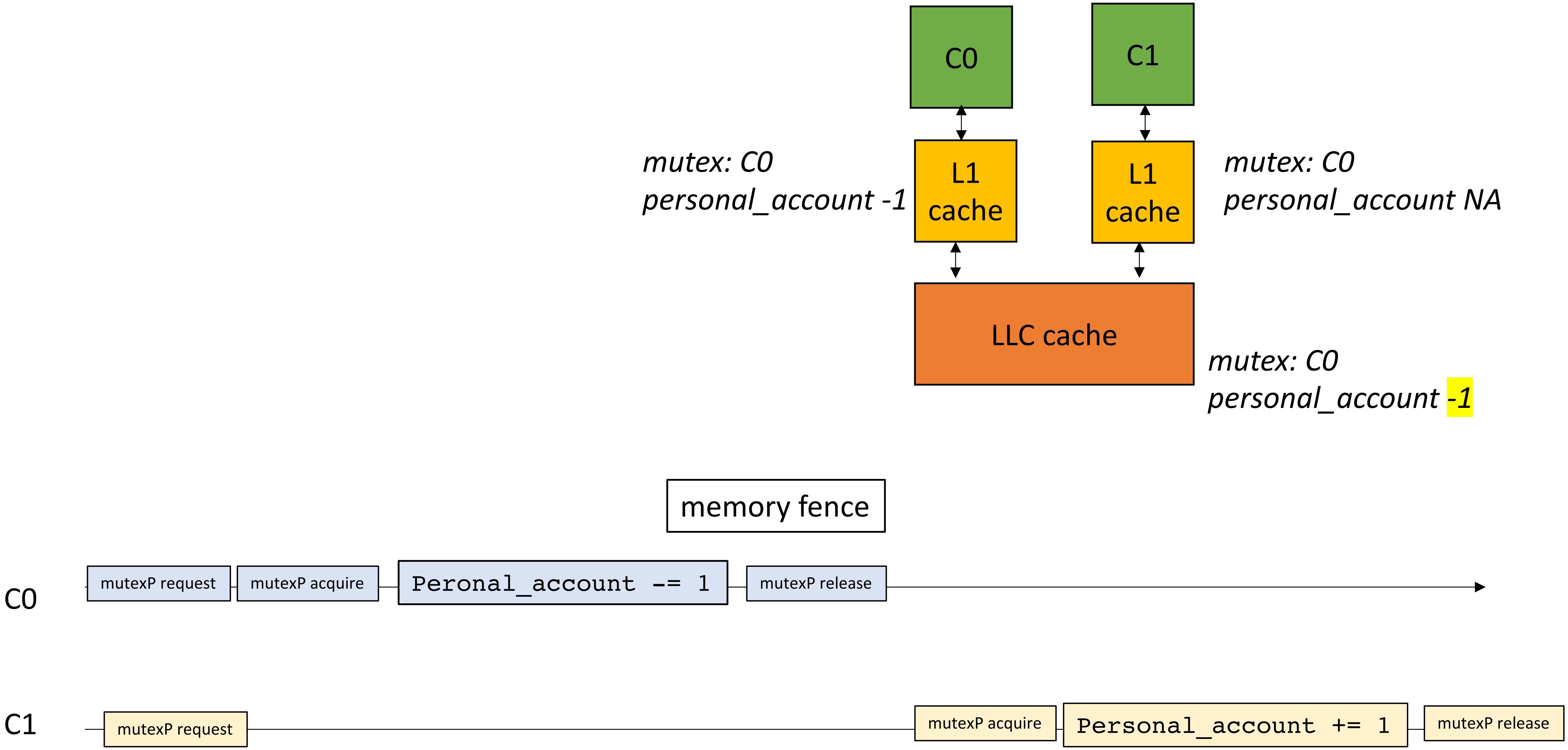
Possible for the mutex message to travel through the cache before the personal_account information!



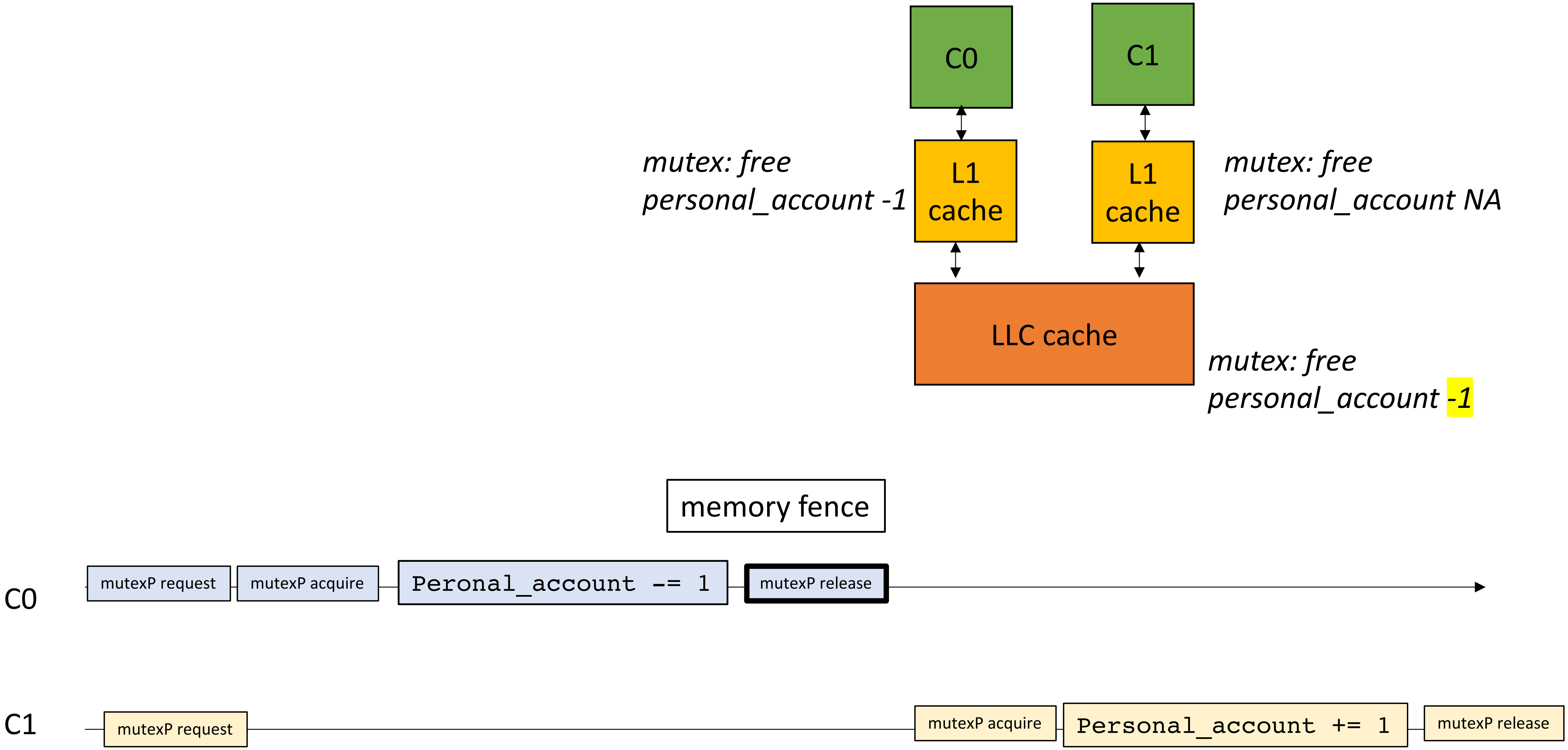
- Memory Fence (or Memory Barrier)



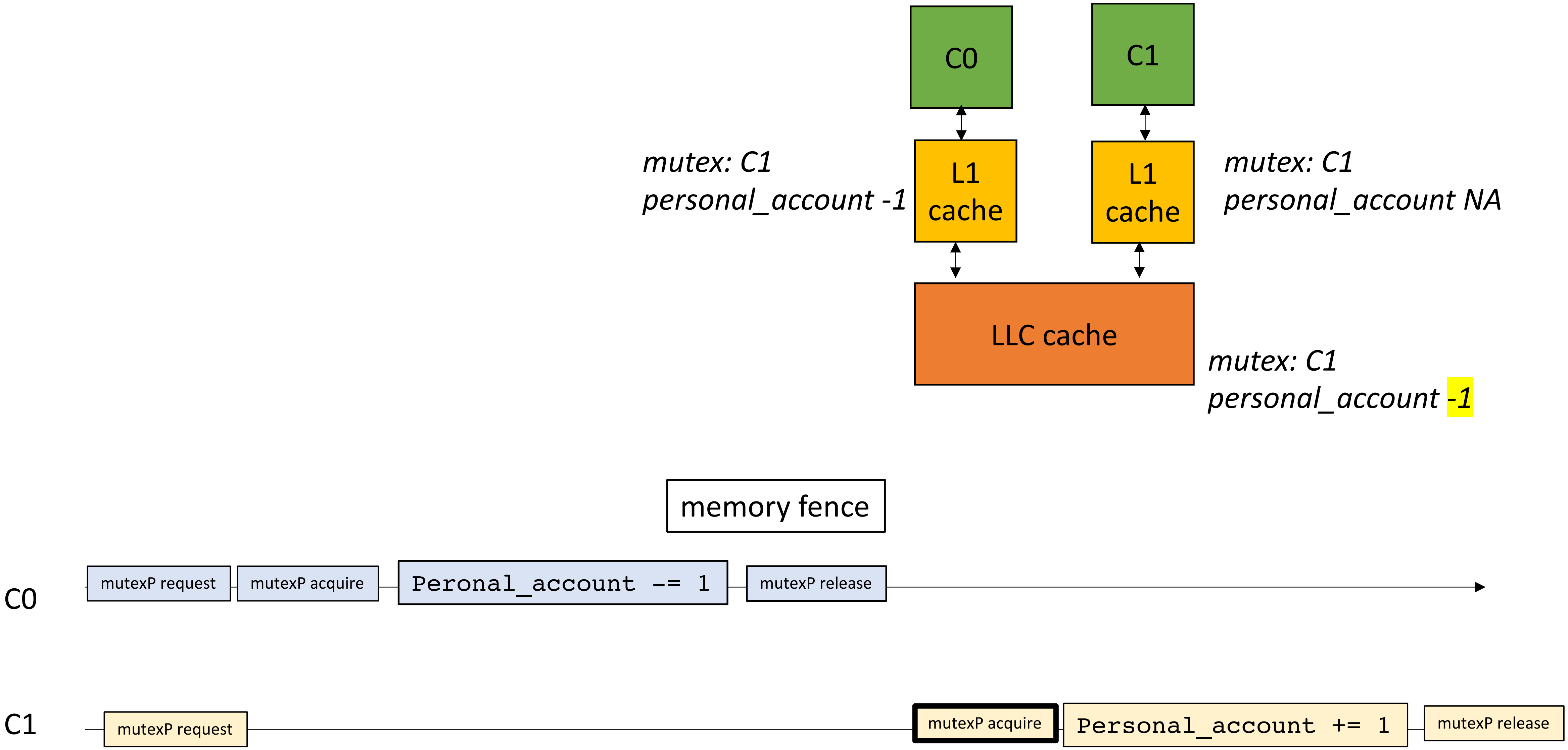
- Memory Fence (or Memory Barrier)



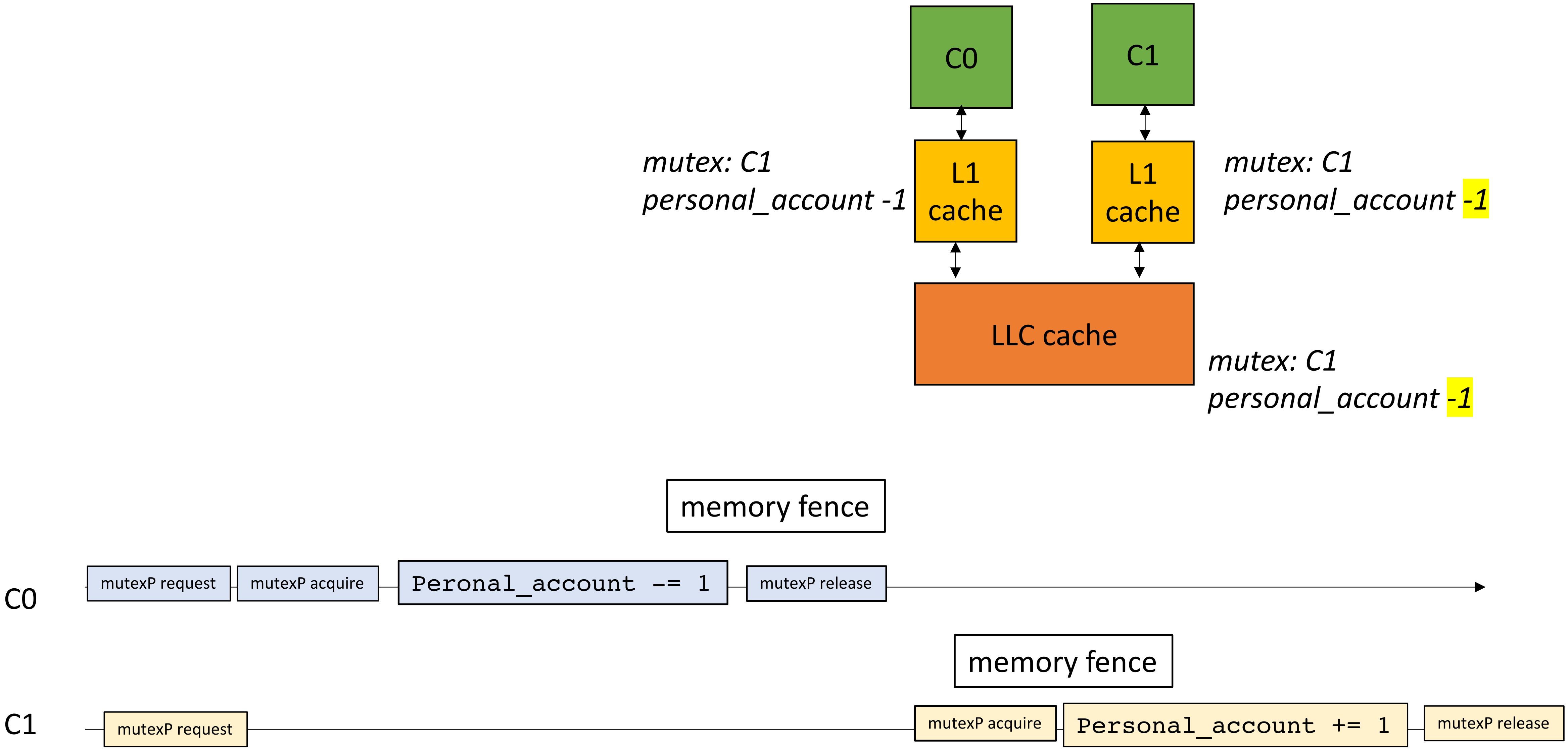
- Memory Fence (or Memory Barrier)



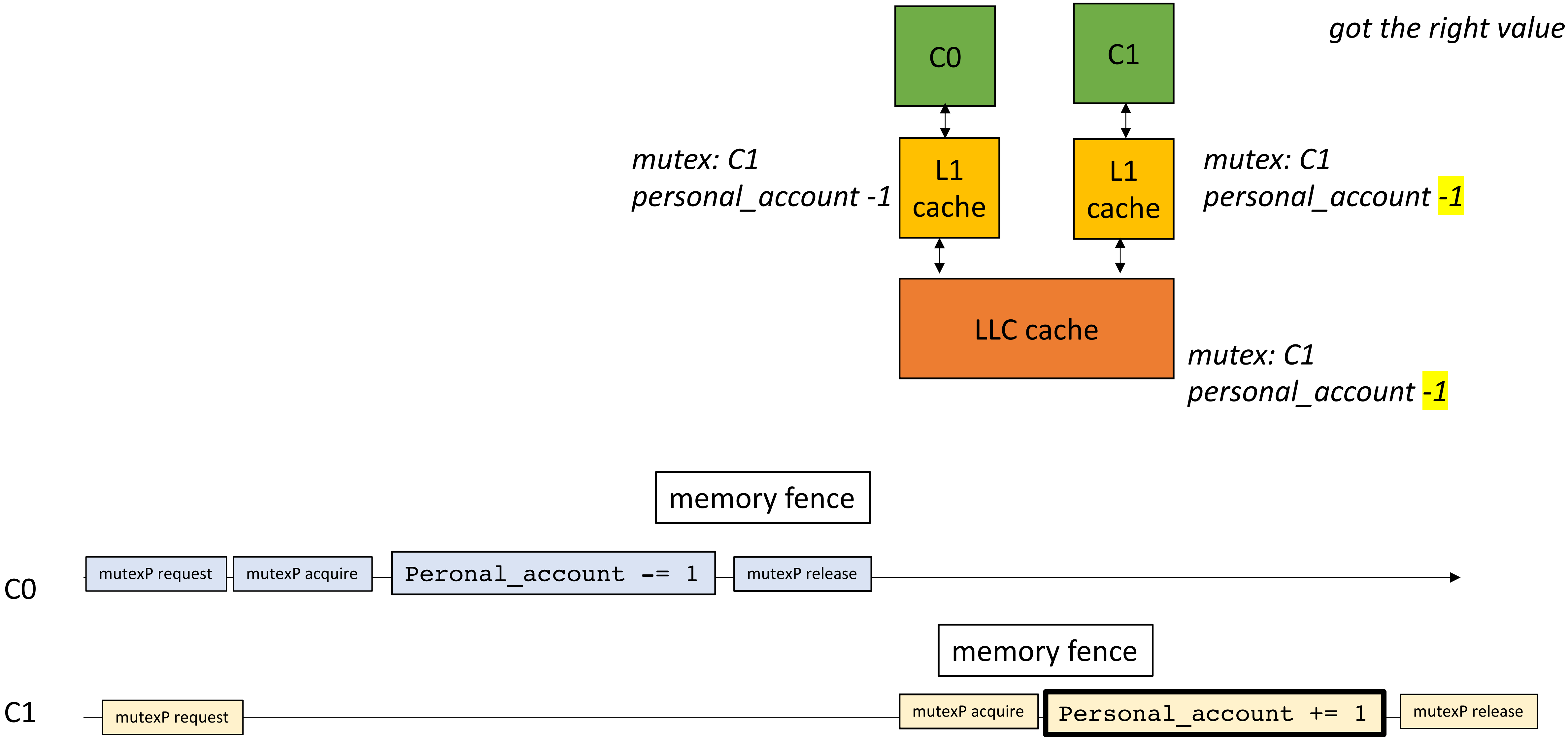
- Memory Fence (or Memory Barrier)



- 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

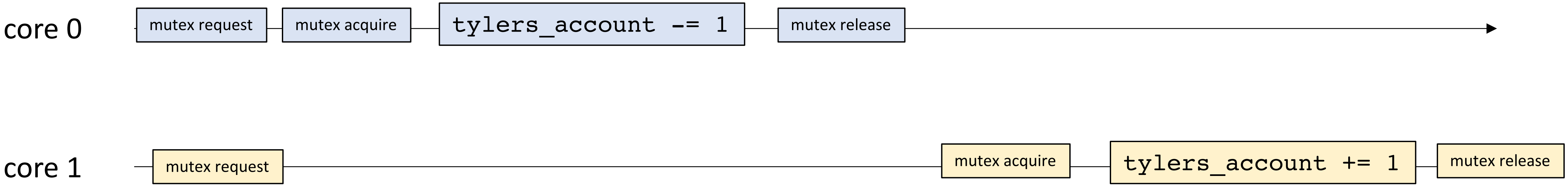
ARM and PowerPC let cache values flow out-of-order
GPUs let caches flow out-of-order

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

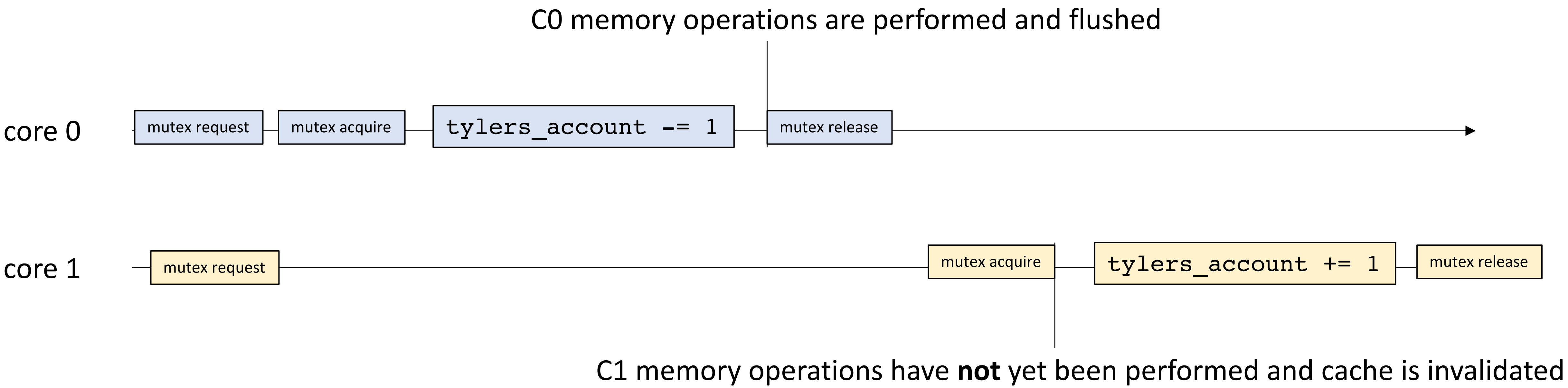
- What do those fences (compiler and memory) give us?
- Atomics were designed so that we can implement things like mutexes!



Atomics

- What do those fences (compiler and memory) give us?

- Atomics were designed so that we can implement things like mutexes!



Mutex Implementations

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();
```

core 0

core 1

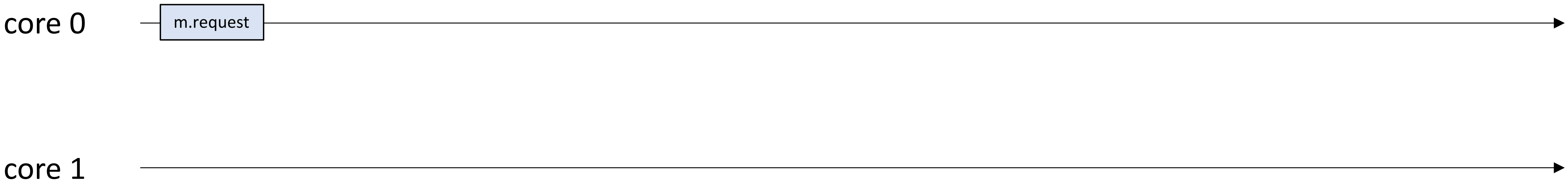
Analysis

```
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);
}
```

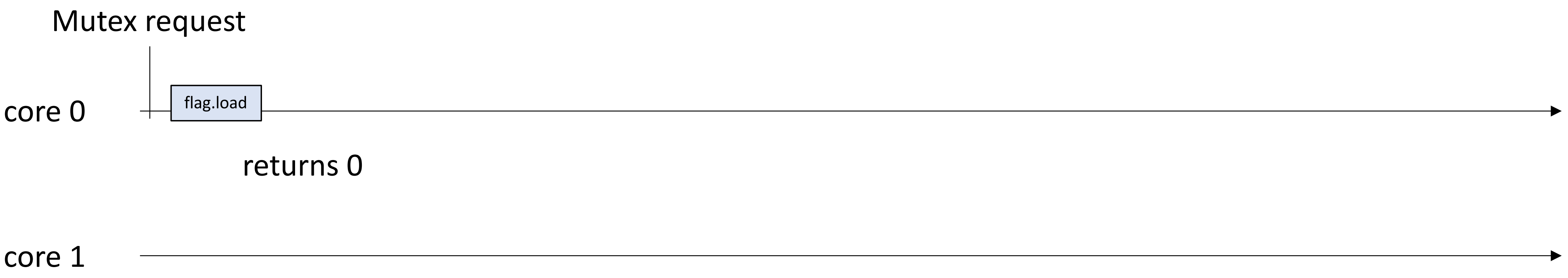
```
void unlock() {
  flag.store(0);
}
```

Thread 0:

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

Thread 1:

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



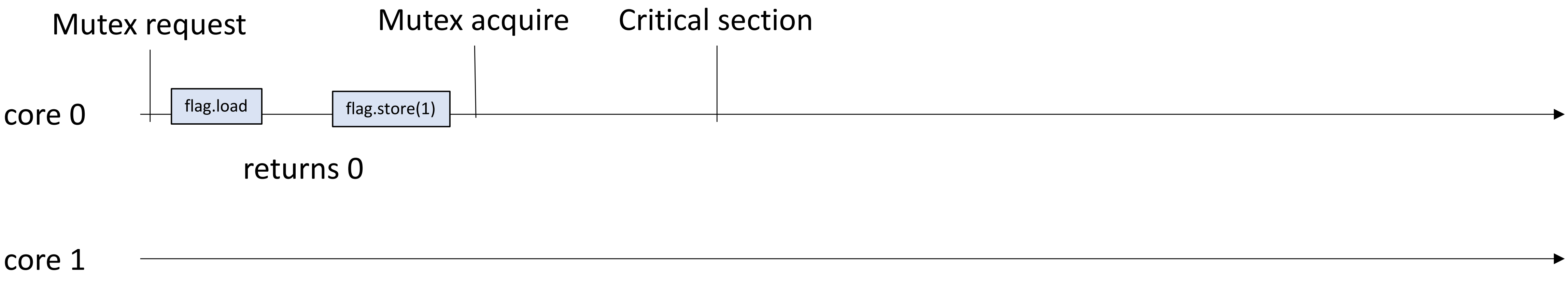
Analysis

```
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();
```



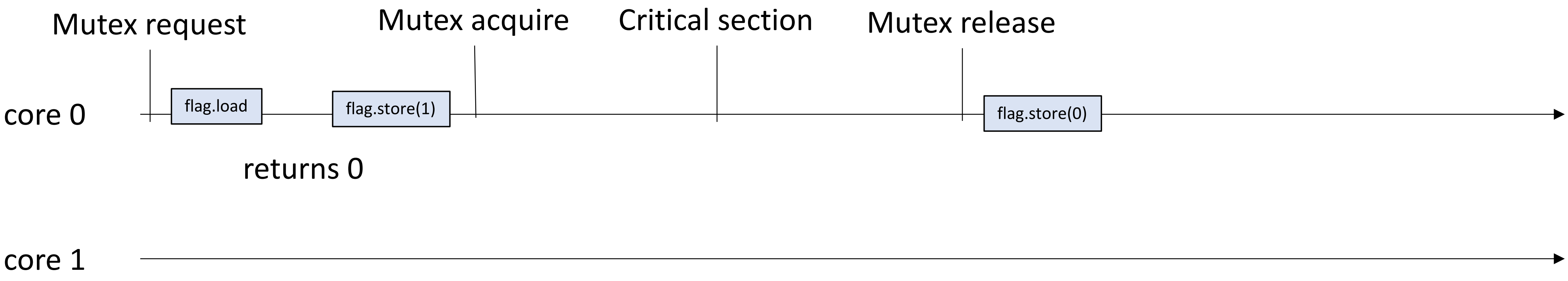
Analysis

```
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();
```



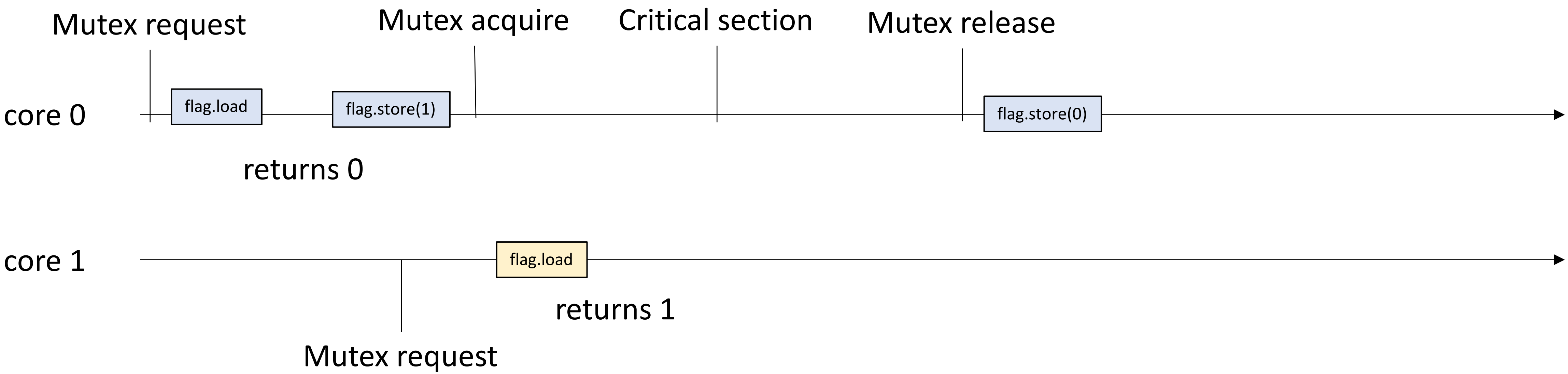
Analysis

```
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);
}
```

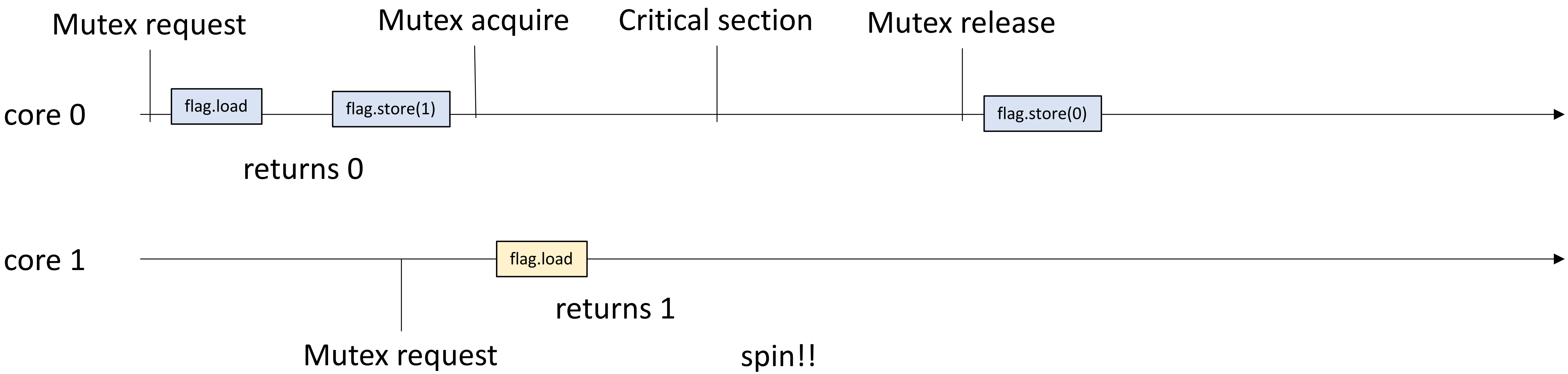
```
void unlock() {
  flag.store(0);
}
```

Thread 0:

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

Thread 1:

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



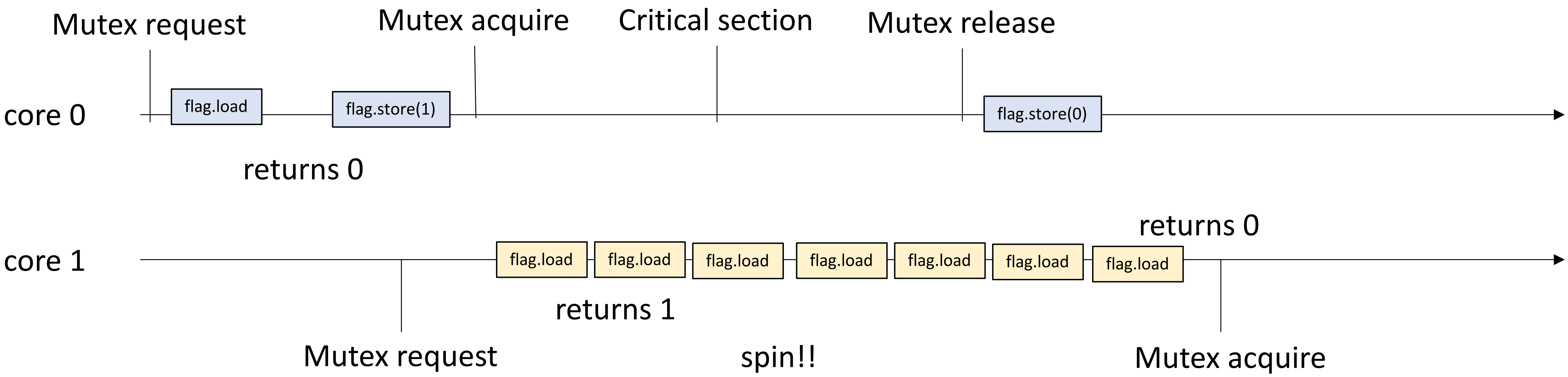
Analysis

```
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();
```



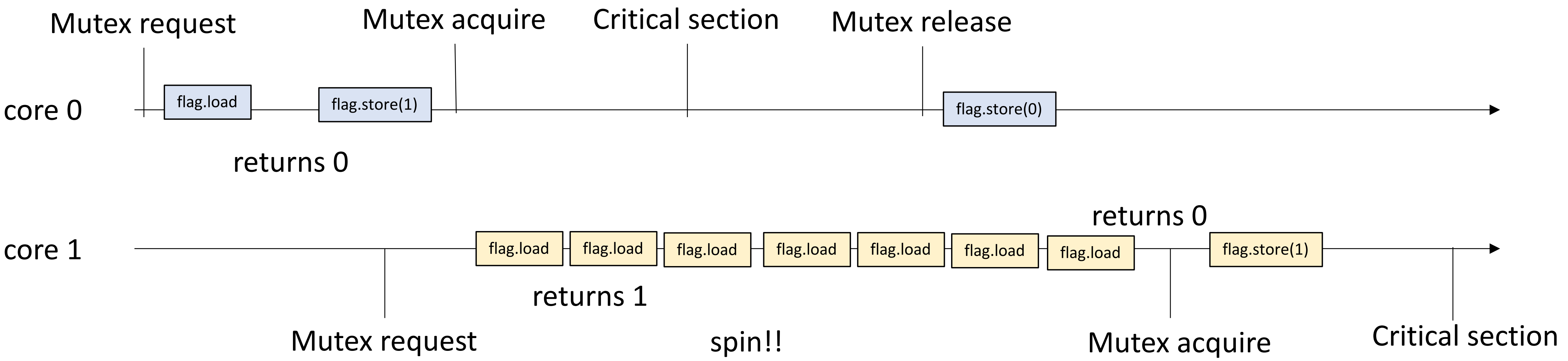
Analysis

```
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

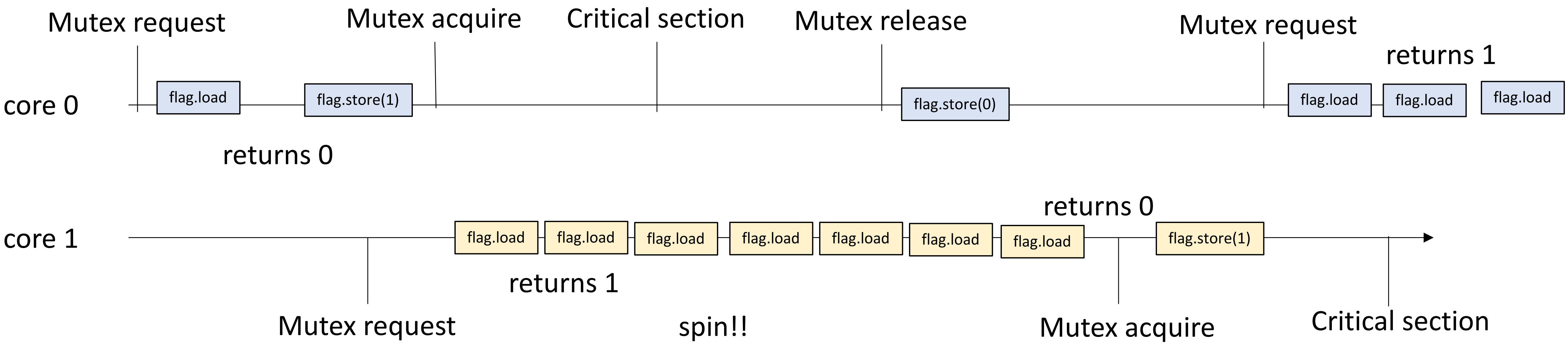
```
void lock() {
  while (flag.load() == 1);
  flag.store(1);
}
```

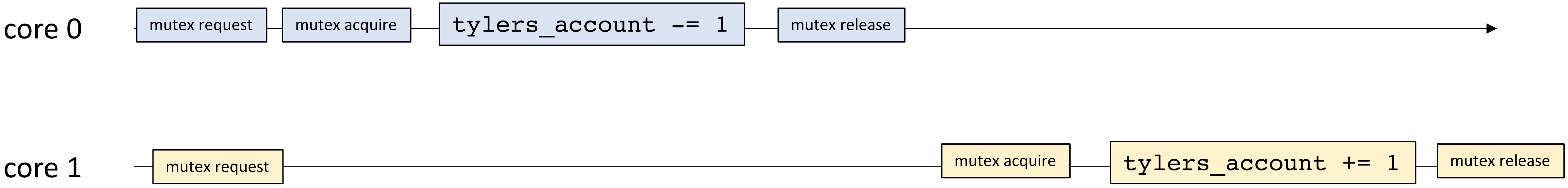
```
void unlock() {
  flag.store(0);
}
```

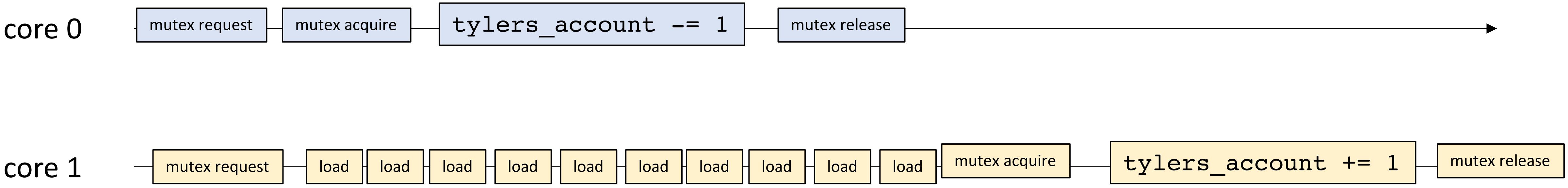
Thread 0:
```
m.lock();
m.unlock();
m.lock();
m.unlock();
```

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

Mutual Exclusion property!
critical sections do not overlap!




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);
}
```

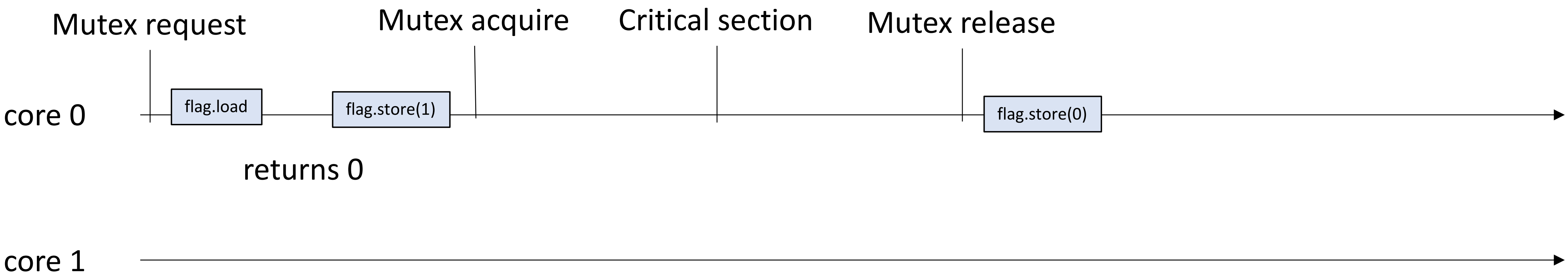
Thread 0:

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

Thread 1:

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

Lets try another interleaving



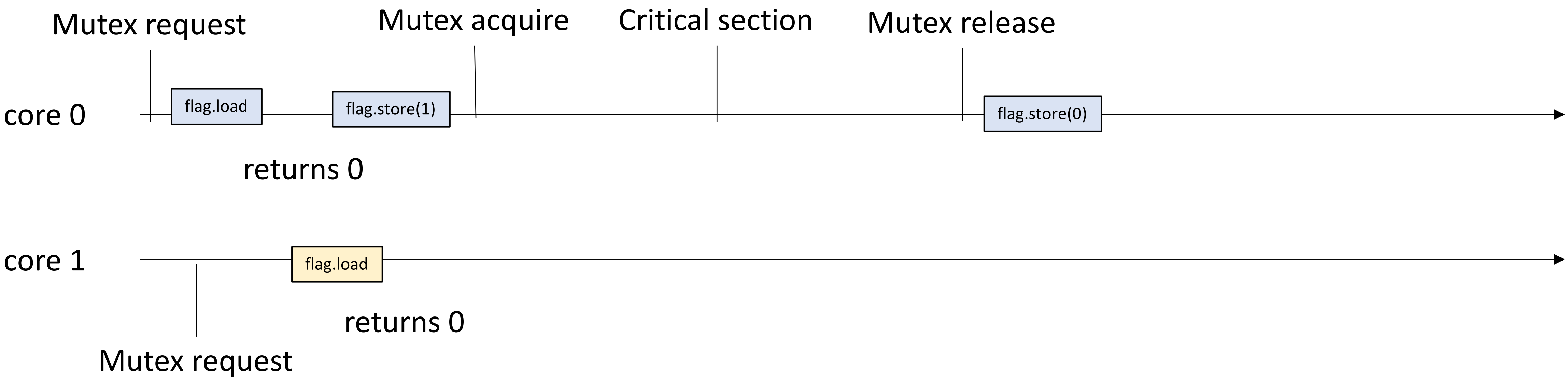
Analysis

```
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);
}
```

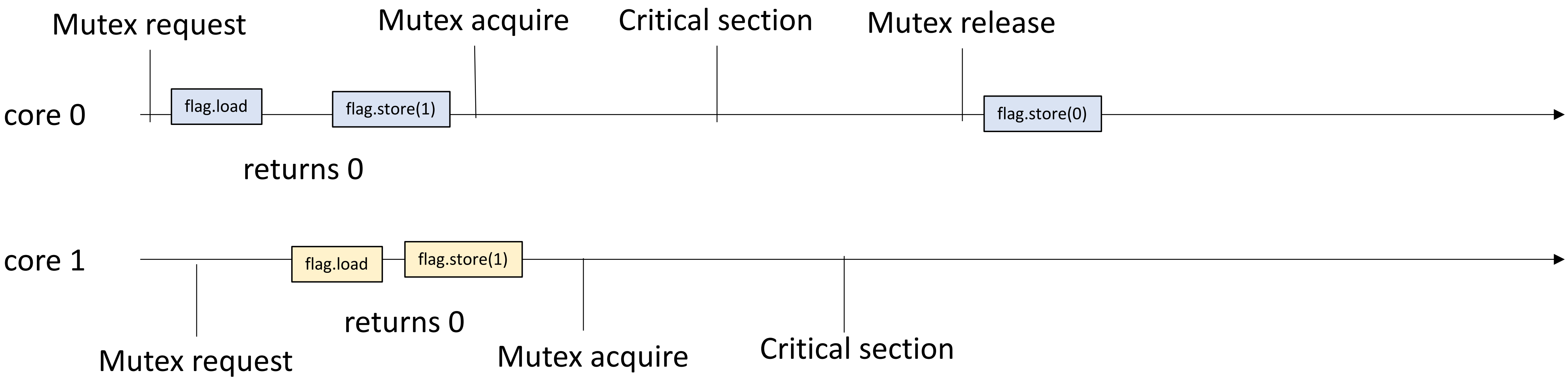
Thread 0:

```
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();
```

core 0

core 1

Analysis

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

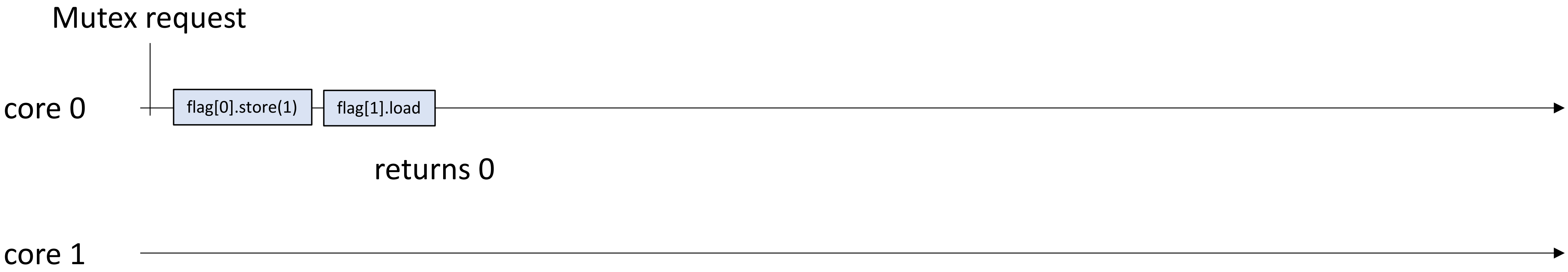
```
void unlock() {
  int i = thread_id;
  flag[i].store(0);
}
```

Thread 0:

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

Thread 1:

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



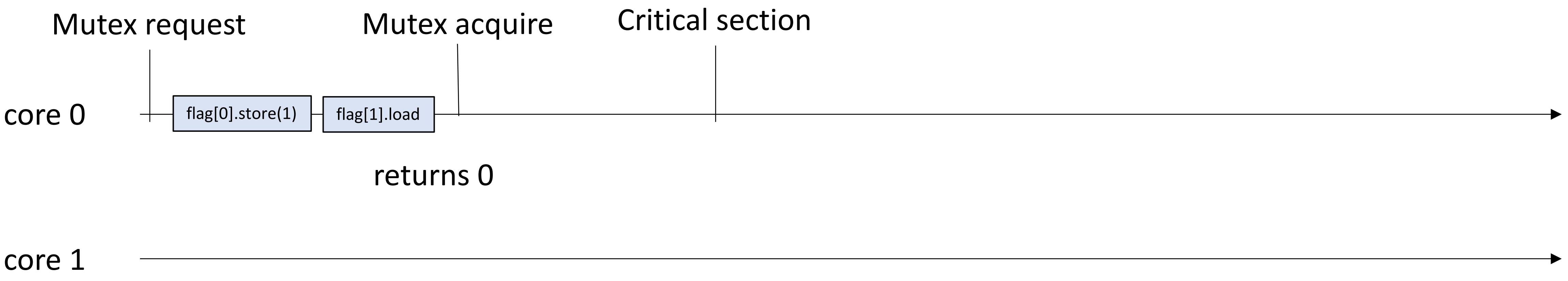
Analysis

```
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();
```



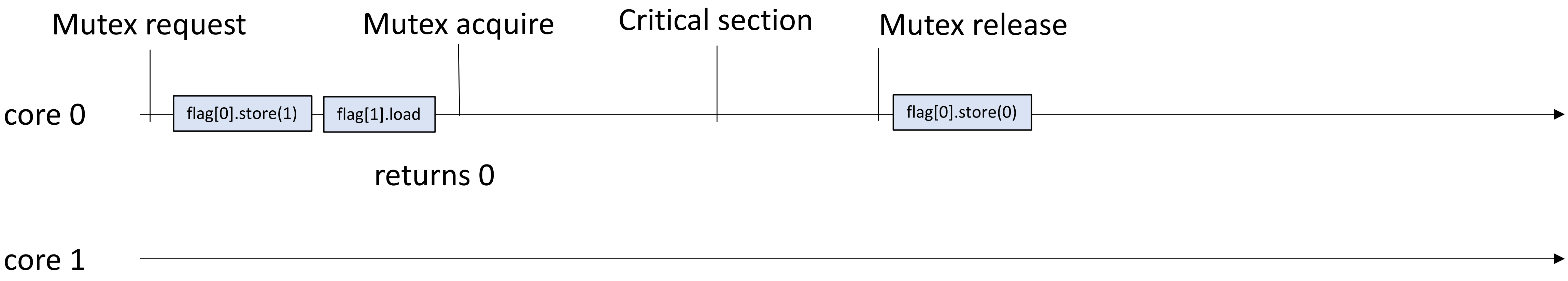
Analysis

```
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

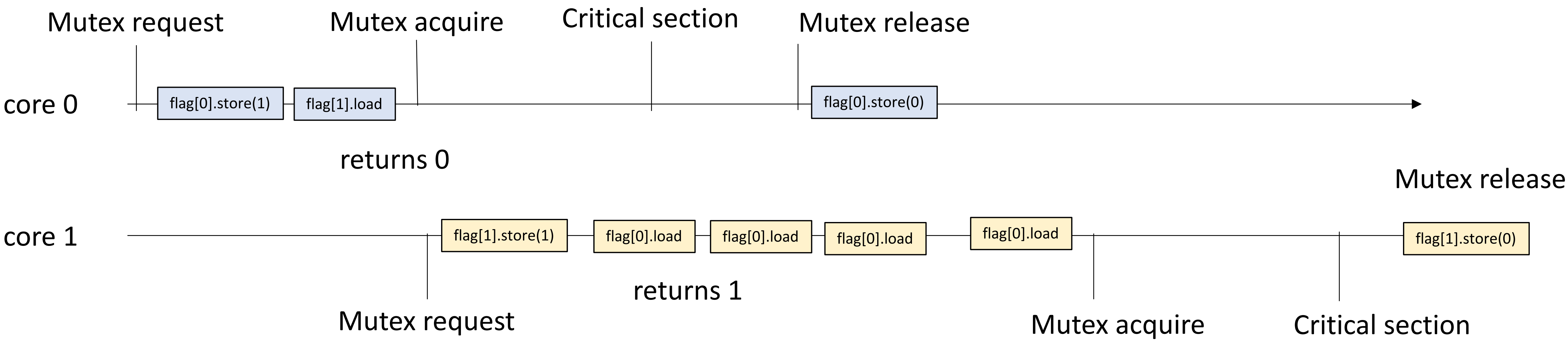
```
void lock() {
  int i = thread_id;
  flag[i].store(1);
  int j = i == 0 ? 1 : 0;
  while (flag[j].load() == 1);
}
```

```
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!



Analysis

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

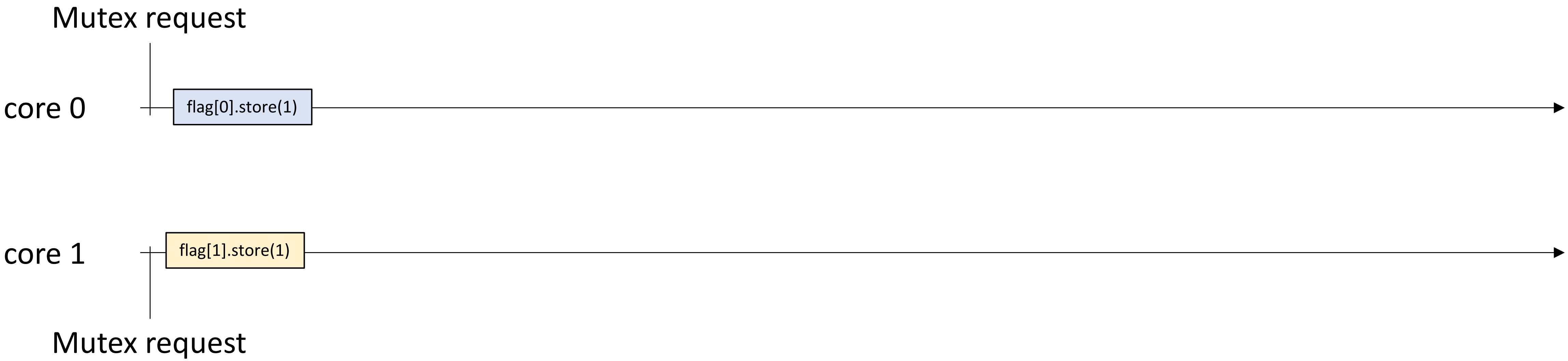
```
void unlock() {
  int i = thread_id;
  flag[i].store(0);
}
```

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);
}
```

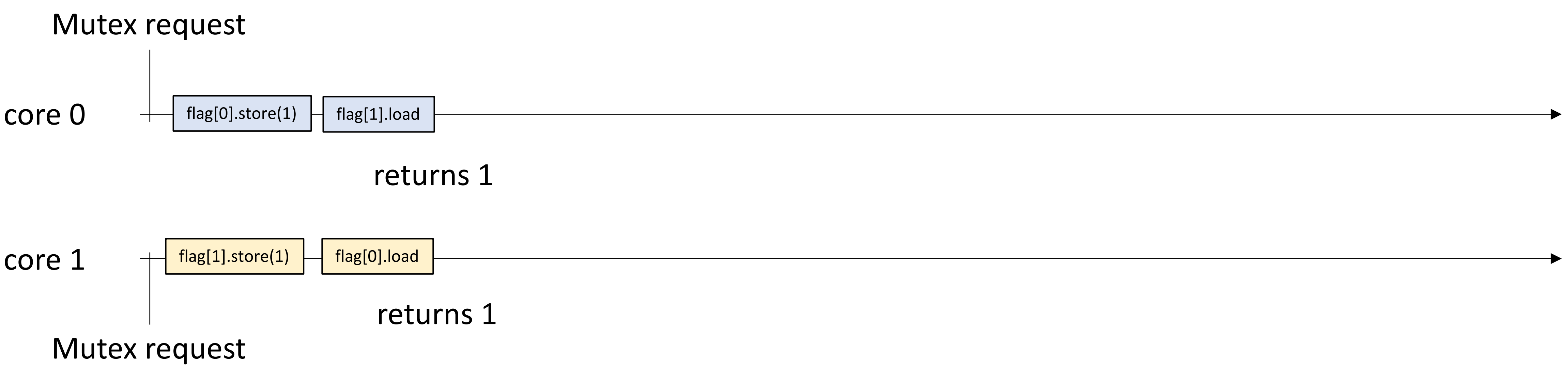
```
void unlock() {
  int i = thread_id;
  flag[i].store(0);
}
```

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);
}
```

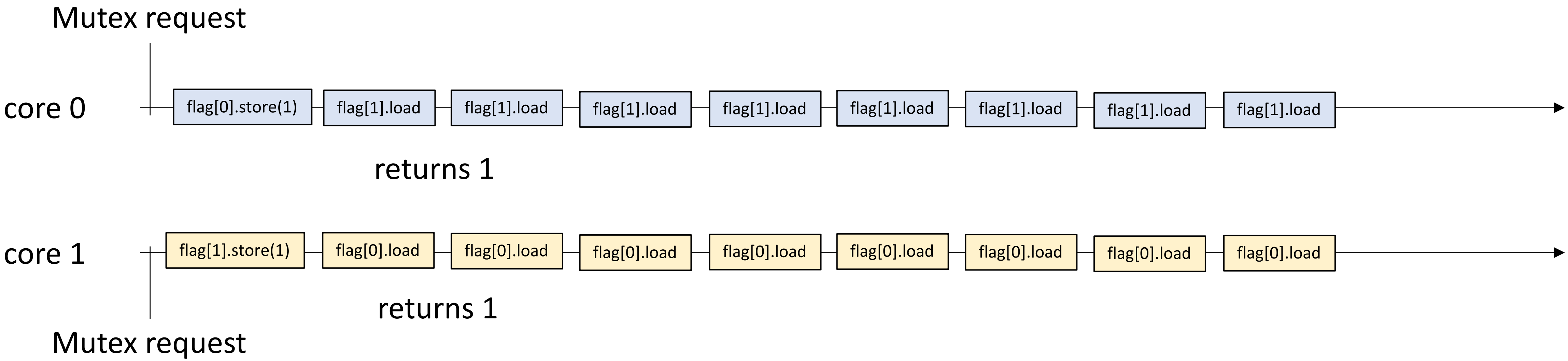
Thread 0:

```
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();
```

Mutex request

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

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

Thread 0:

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

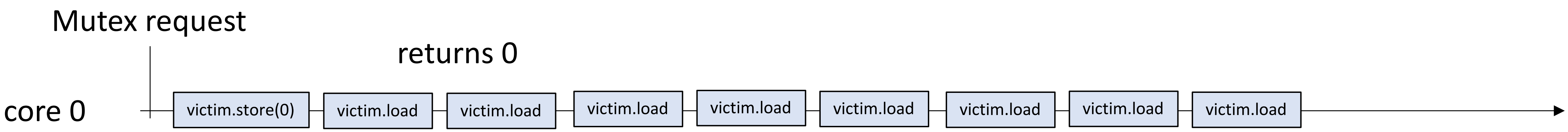
```
void lock() {
  victim.store(thread_id);
  while (victim.load() == thread_id);
}
```

```
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);
}
```

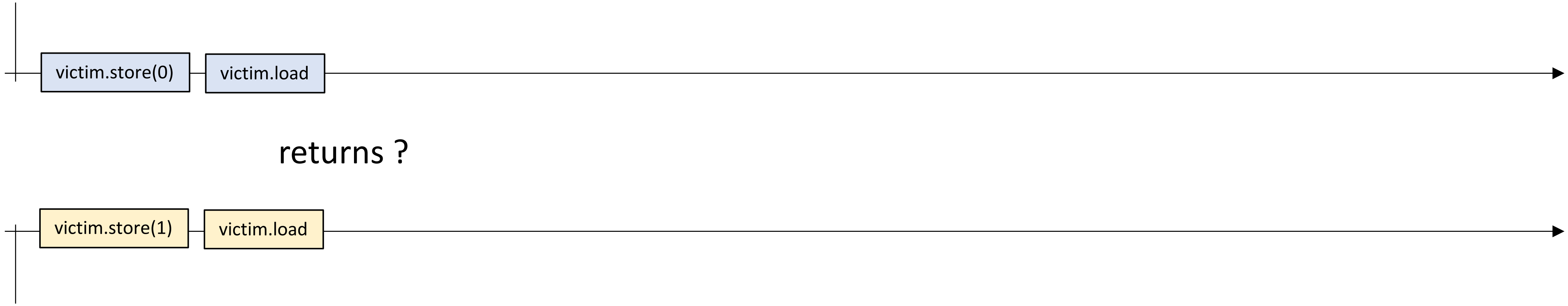
```
void unlock() {}
```

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

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

Mutex request

core 0



core 1

Mutex request

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

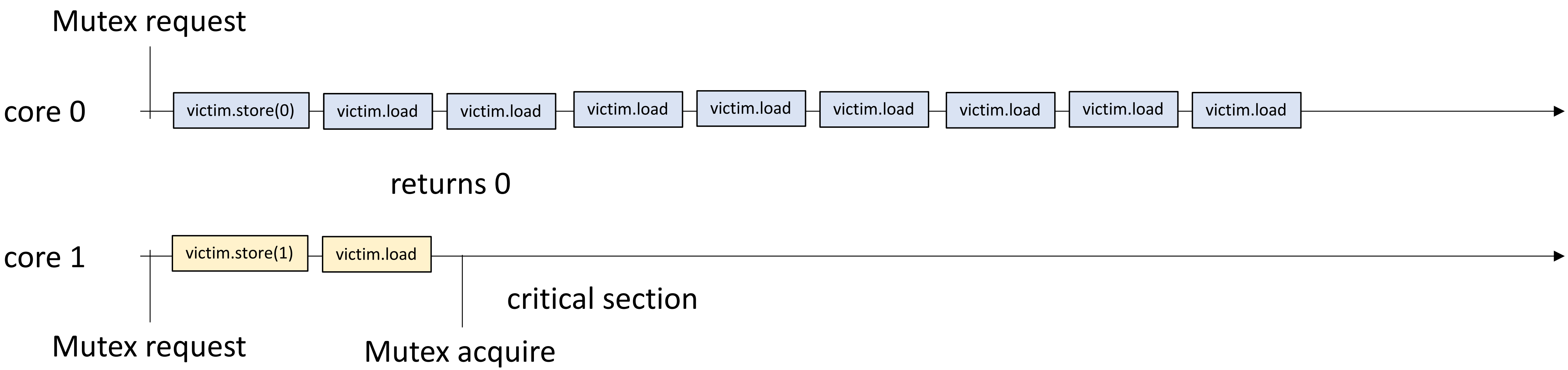
```
void unlock() {}
```

Thread 0:

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

Thread 1:

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



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

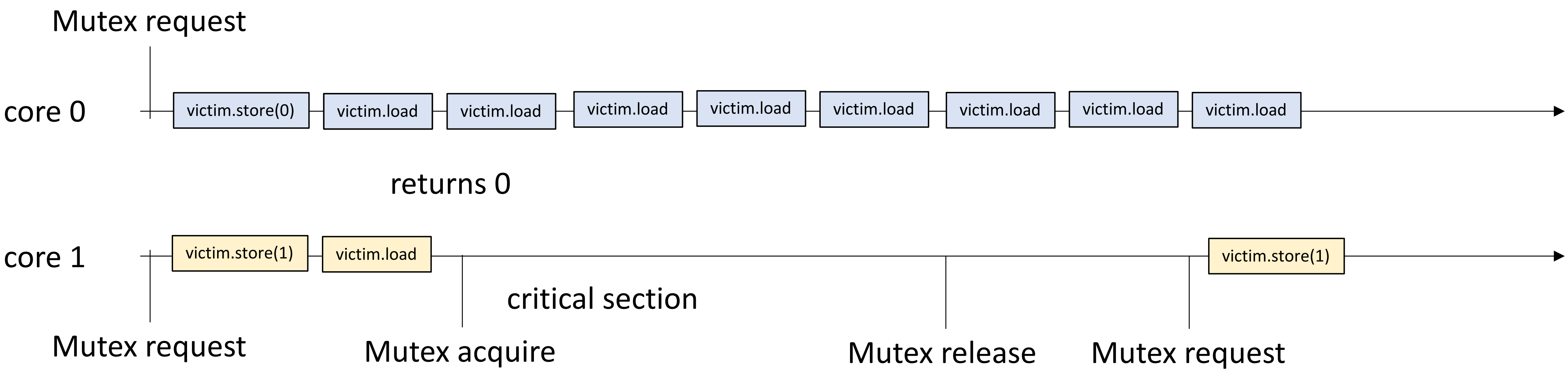
```
void unlock() {}
```

Thread 0:

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

Thread 1:

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



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

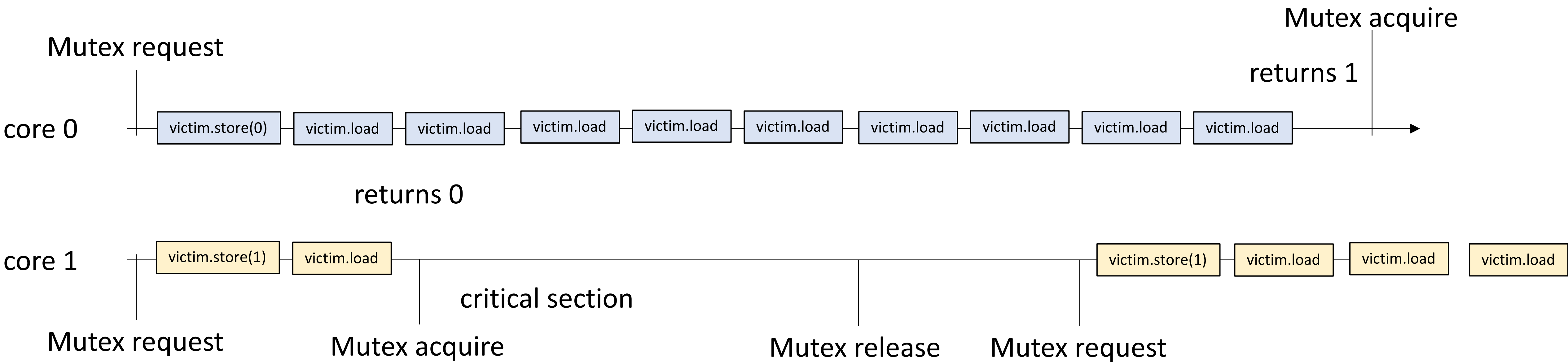
```
void unlock() {}
```

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];
};
```

Initially:
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:

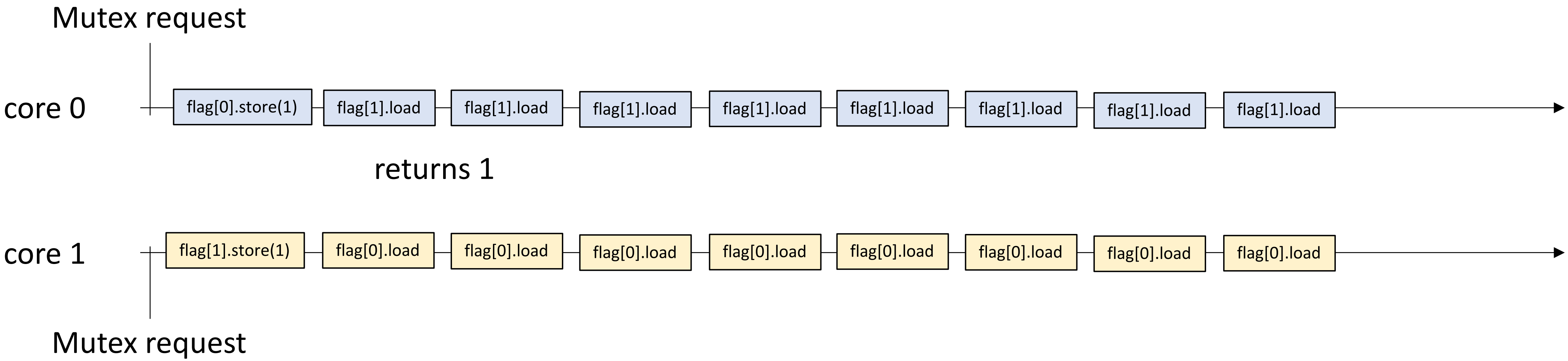
```
m.lock();
m.unlock();
```

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);
}
```

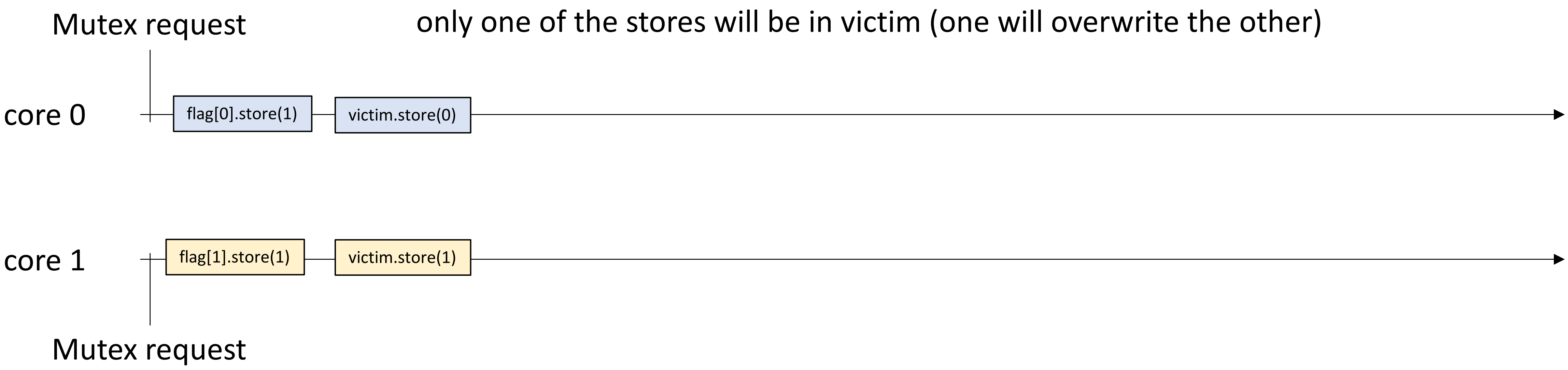
Thread 0:

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

Thread 1:

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

only one of the stores will be in victim (one will overwrite the other)



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);
}
```

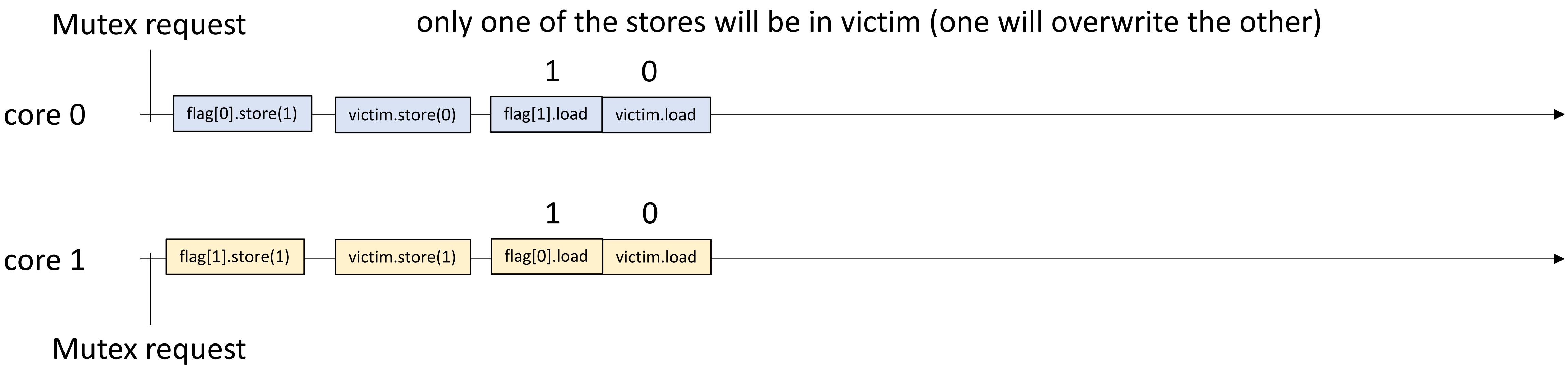
```
void unlock() {
  int i = thread_id;
  flag[i].store(0);
}
```

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);
}
```

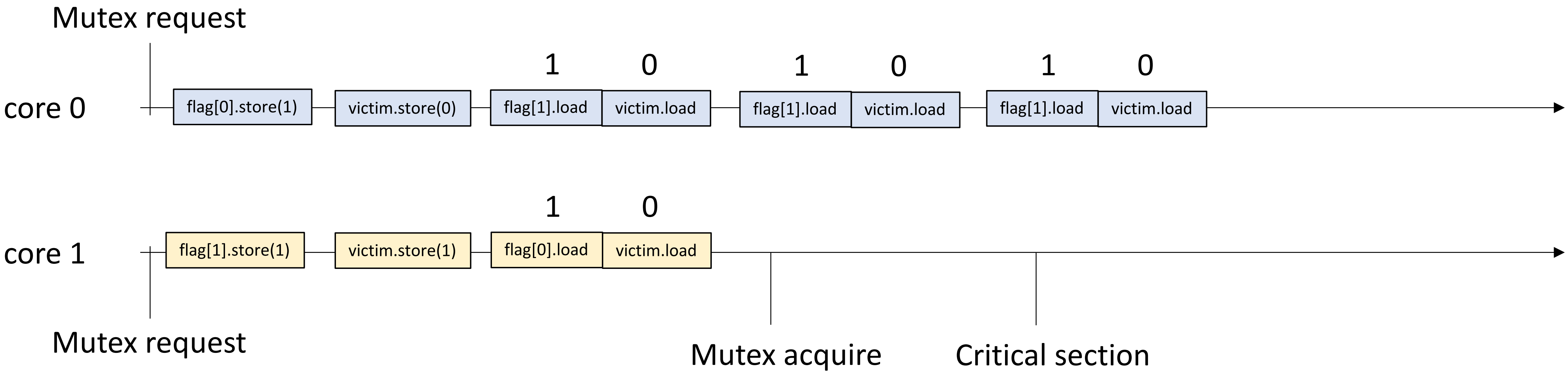
```
void unlock() {
  int i = thread_id;
  flag[i].store(0);
}
```

Thread 0:

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

Thread 1:

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



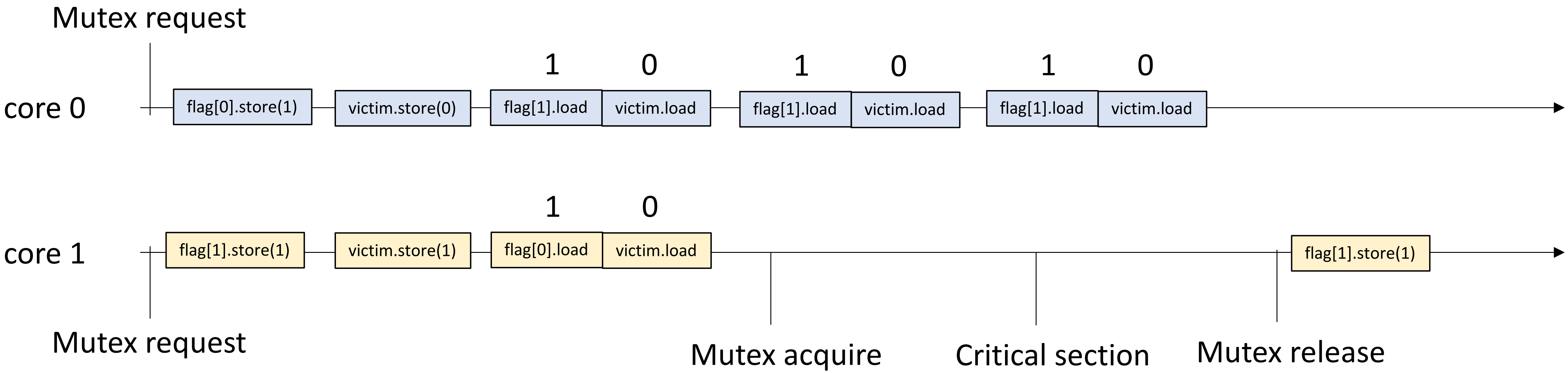
Tie breaking with victim

```
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();`



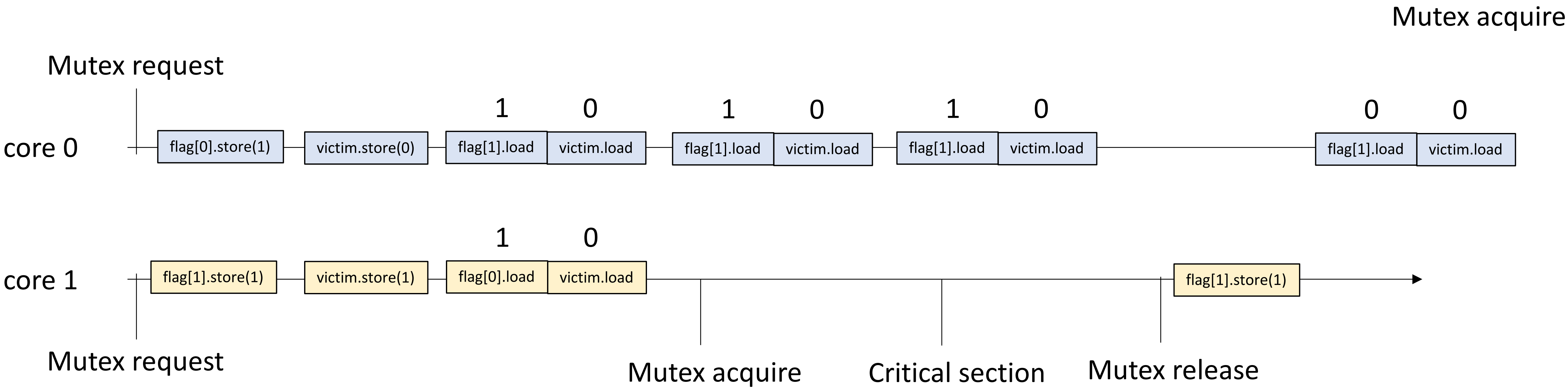
Tie breaking with victim

```
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);
}
```

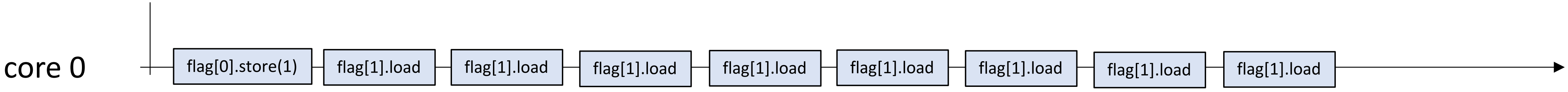
```
void unlock() {}
```

Thread 0:

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

will spin forever!

Mutex request



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();
```

Mutex request

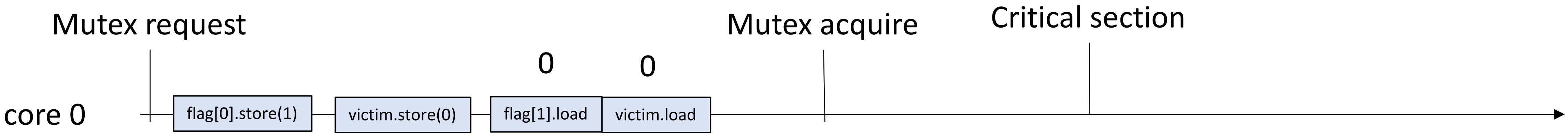
previous flag issue

```
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

This lock satisfies the two critical properties

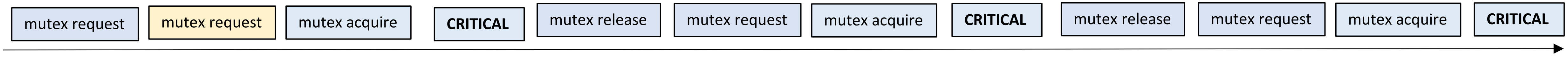
- Mutual exclusion
- Deadlock freedom
- *More formal proof given in the textbook*

What about starvation

recall the starvation property:

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

concurrent execution



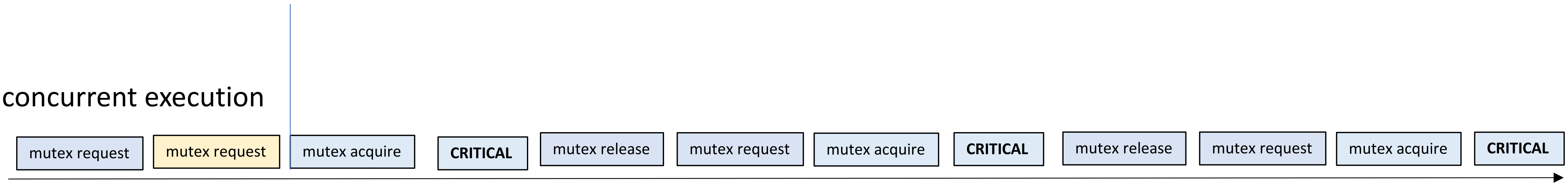
time

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

concurrent execution



time

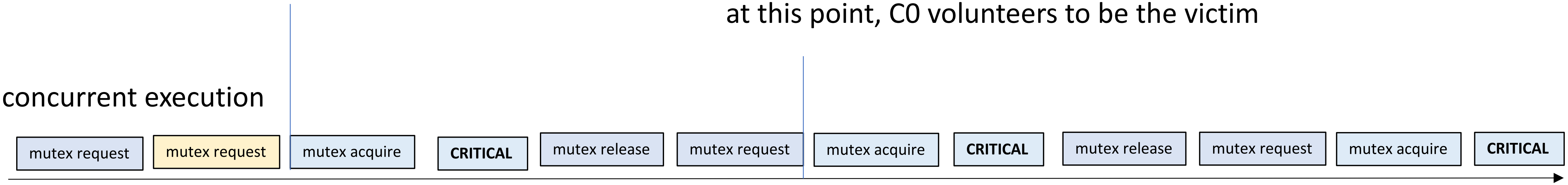
What about starvation

```
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

at this point, C0 volunteers to be the victim

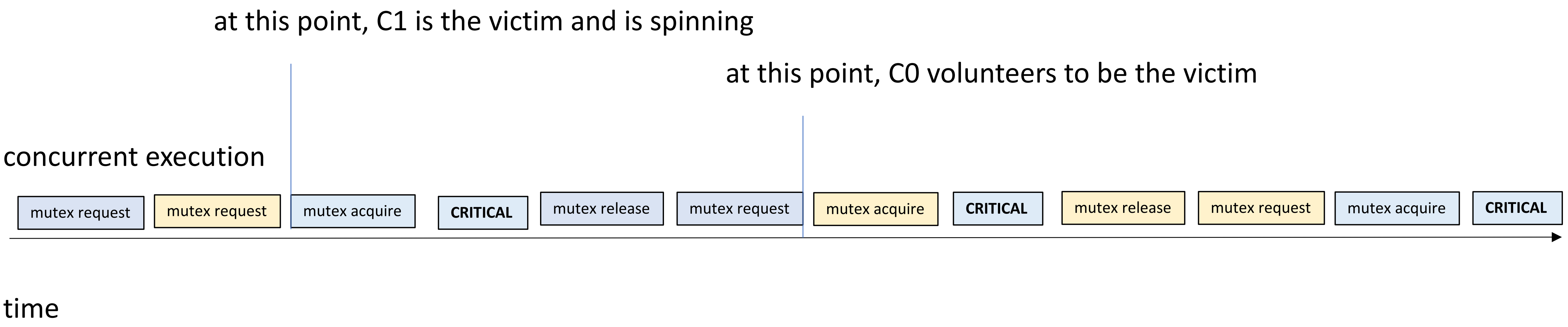
concurrent execution



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);
}
```



What about starvation

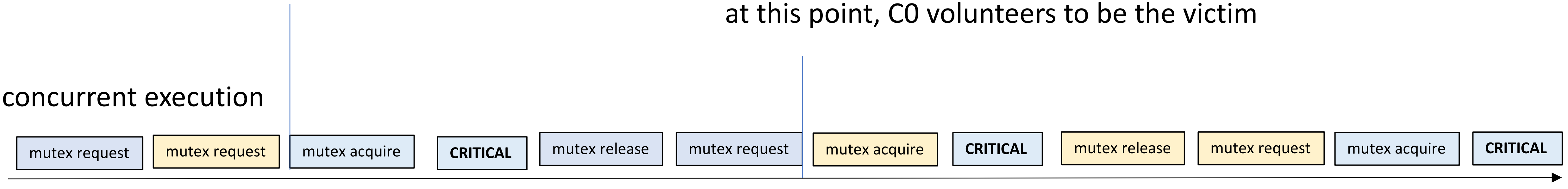
Threads take turns in petersons algorithm. It is starvation free

```
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);
}
```

at this point, C1 is the victim and is spinning

at this point, C0 volunteers to be the victim

concurrent execution



time

Mutex Implementations

Peterson only works with 2 threads.

Generalizes to the Filter Lock (Read chapter 2 in the book, part 1 of your homework!)

Thanks!

- Next time:
  - practical mutual exclusion

- Finish homework 1 and look out for homework 2!
  - use office hours, piazza and tutors

- Do the quiz!