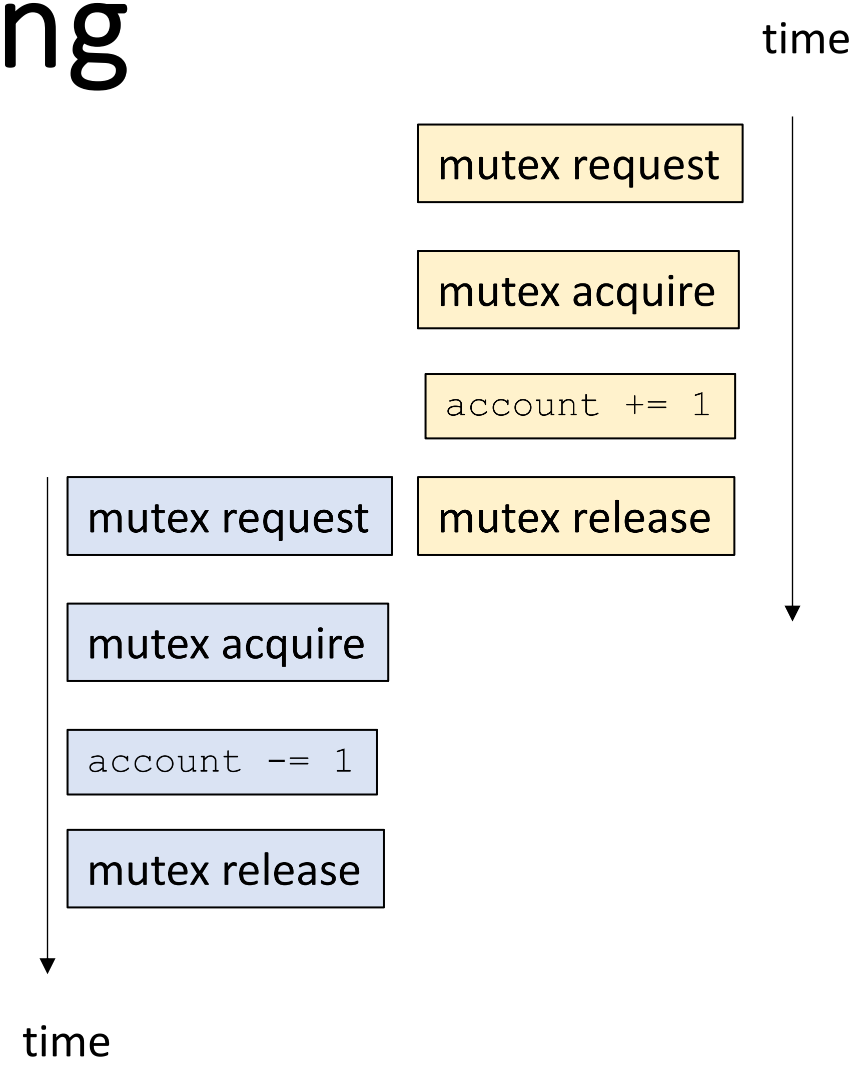
CSE113: Parallel Programming

- **Topics**:
  - Intro to mutual exclusion
    - Different types of parallelism
    - Data conflicts
    - Protecting shared data



Announcements

- Second lecture in Module 2: mutexes!

- HW 2 will be assigned today at midnight. You'll have what you need to complete part 1 by end of today.

- No guarantee of homework help after 5 PM or weekends.

Announcements

- Midterm is in 2 weeks
  - In-person test
  - 3 pages of notes front and back (but no memorization questions)
  - 10% of your grade

Previous quiz

Previous quiz

It is possible to interleave the load and store operations of RMW atomic operations; however, it is so rare that it does not matter in practice.

Mutex alternatives?

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

<u>Tyler's coffee addiction:</u>

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

<u>Tyler's employer</u>

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

time

time

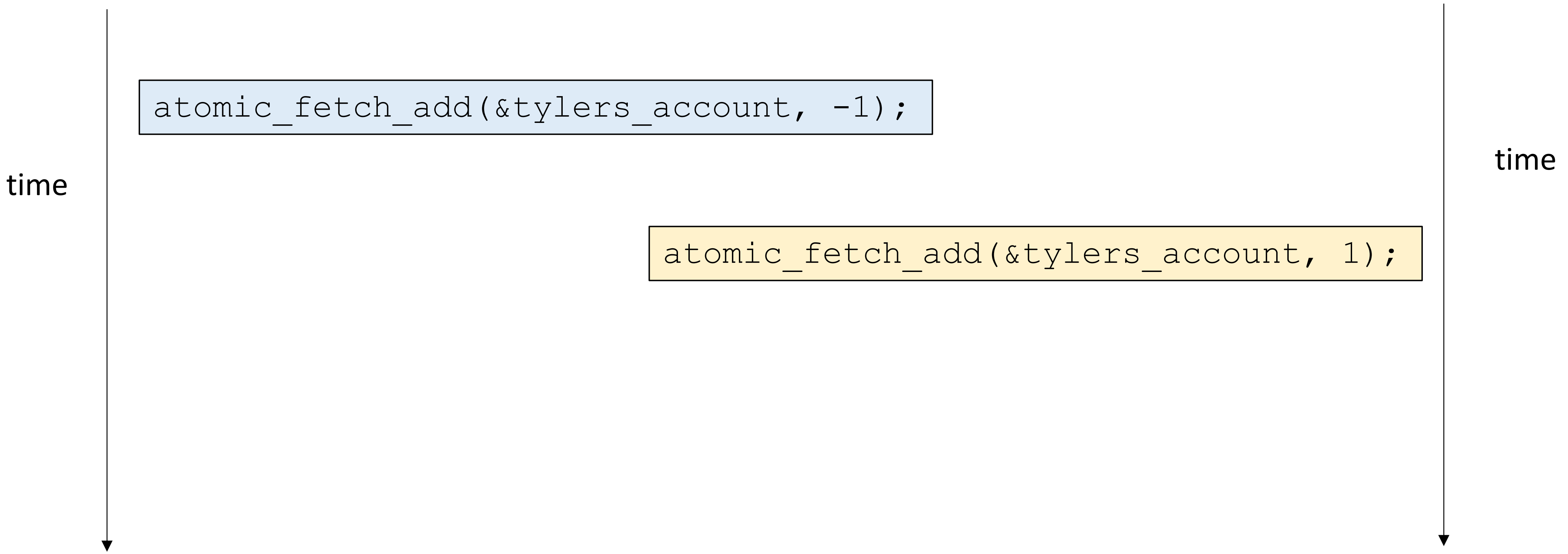
Modify these programs to use atomic RMWs

Tyler's coffee addiction:

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

Tyler's employer

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



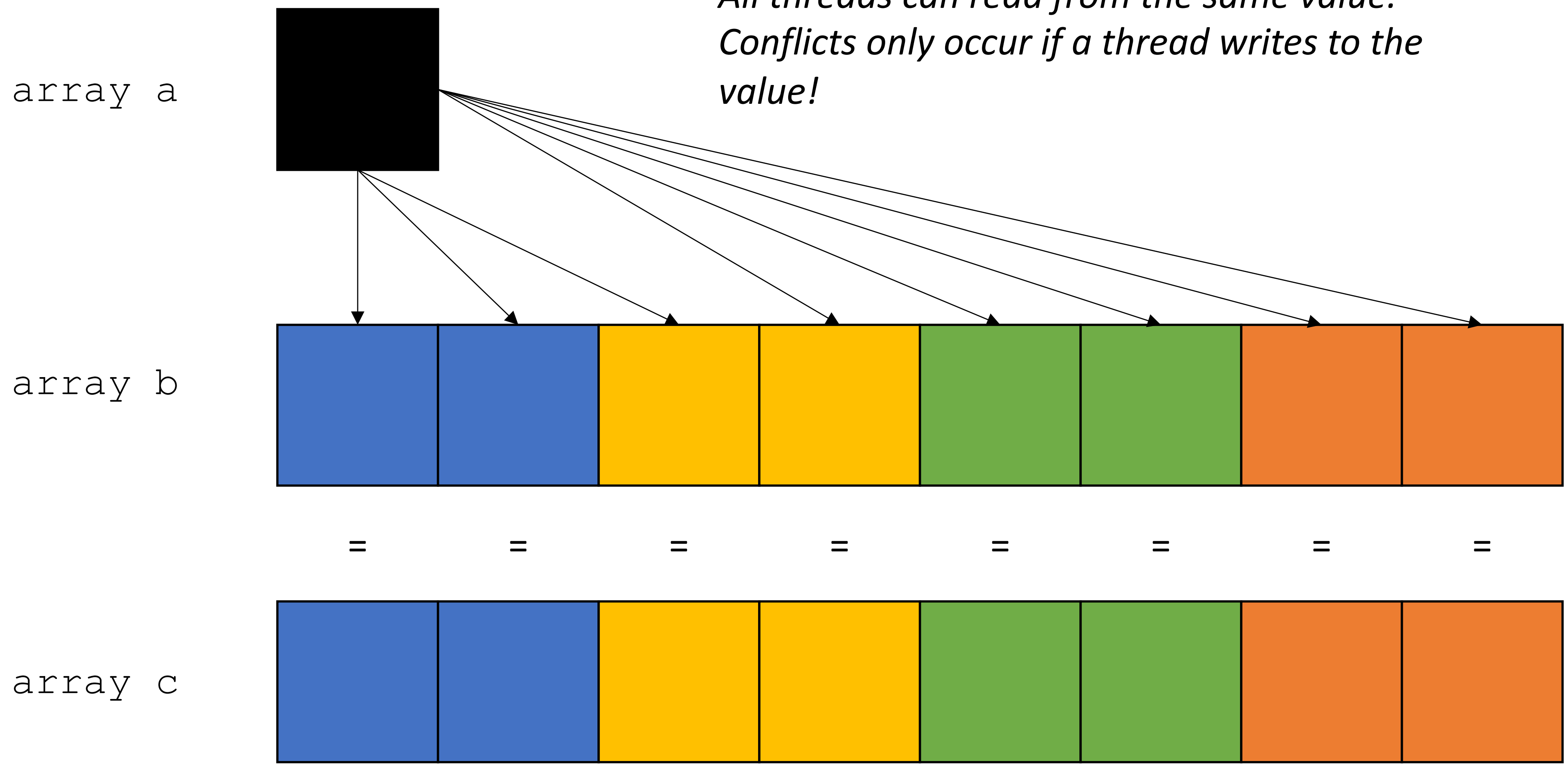
Previous quiz

A data conflict is when two threads access the same memory location.

Embarrassingly parallel

All threads can read from the same value. Conflicts only occur if a thread writes to the value!

Thread 0 - Blue
Thread 1 - Yellow
Thread 2 - Green
Thread 3 - Orange

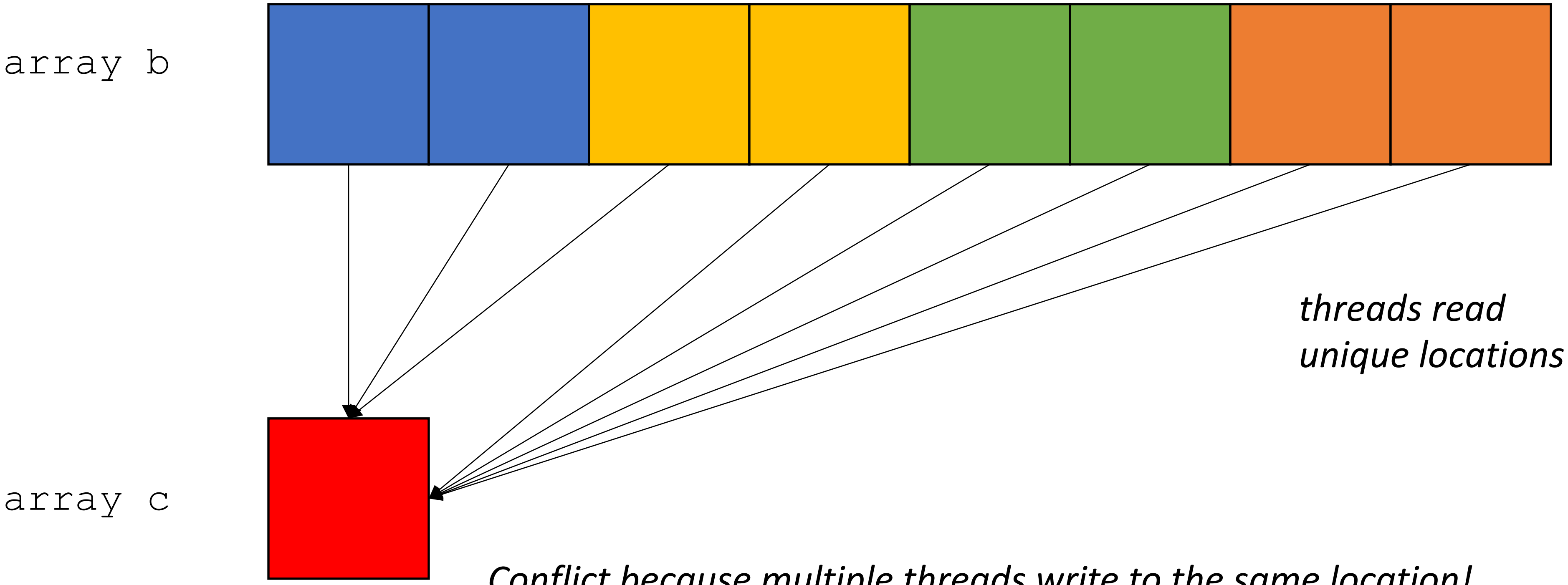


is this problem embarrassingly parallel?

Embarrassingly parallel

Note: Reductions have some parallelism in them, as seen in your homework.

Thread 0 - Blue
Thread 1 - Yellow
Thread 2 - Green
Thread 3 - Orange



threads read unique locations

is this problem embarrassingly parallel?

Conflict because multiple threads write to the same location!

Previous quiz

How many interleavings are possible with 3 threads, each them executing 1 event?

- ○ 1
- ○ 3
- ○ 6
- ○ 12

Previous quiz

How many extra arguments are required to turn a function into an SPMD function?

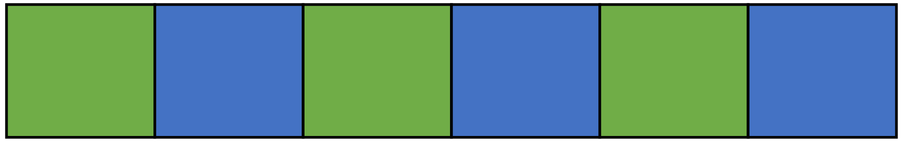
- ○ 0
- ○ 1
- ○ 2
- ○ 3

SPMD programming model

```
void increment_array(int *a, int a_size, int tid, int num_threads) {
    for (int i = tid; i < a_size; i+=num_threads) {
        a[i]++;
    }
}
```

switch to thread 1

iterations computed by thread 1



array a

Assume 2 threads
lets step through thread 1
i.e.
tid = 1
num_threads = 2

Previous quiz

Write a few sentences about how you can remove data-conflicts from your program. We have mentioned a few ways in class, but feel free to mention other ways you can think of!

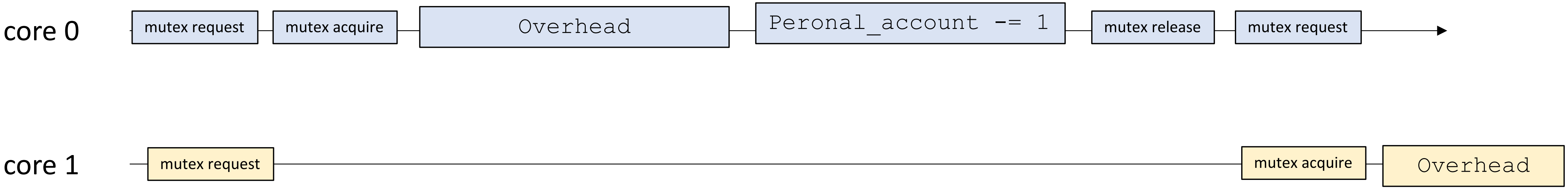
Review

Mutex Performance

Try to keep mutual exclusion sections small! Protect only data conflicts!

Code example with overhead

Long periods of waiting in the threads

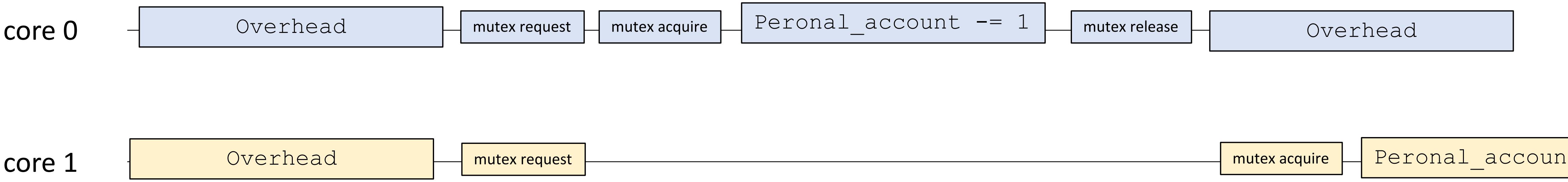


Long periods of waiting in the threads

Mutex Performance

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)

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

Multiple mutexes

Lets say I have two accounts:

- Business account
- Personal account

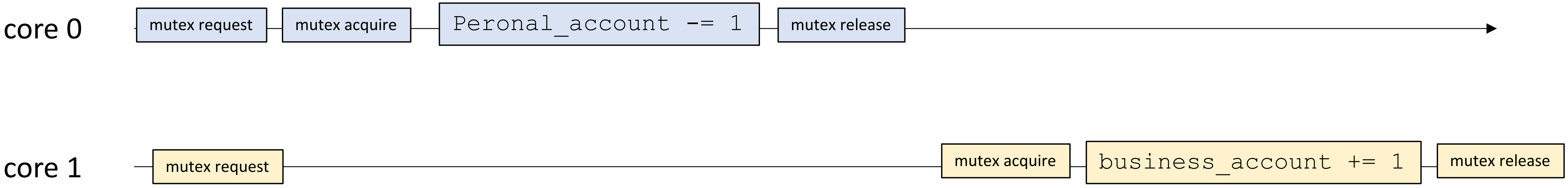
- No reason individual accounts can’t be accessed in parallel

Multiple mutexes

Lets say I have two accounts:

- Business account
- Personal account

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



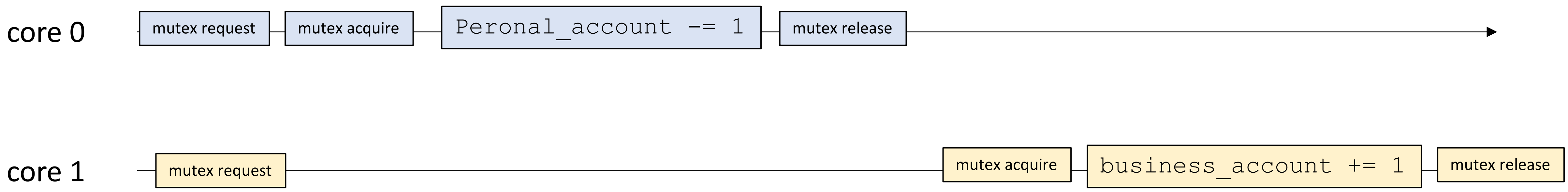
Long periods of waiting in the threads

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

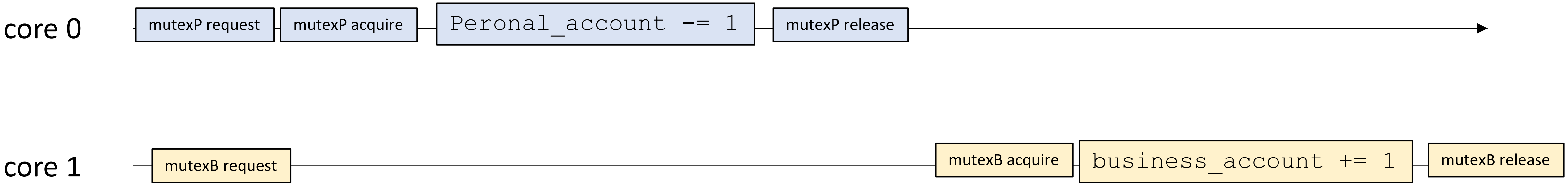


Multiple mutexes

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

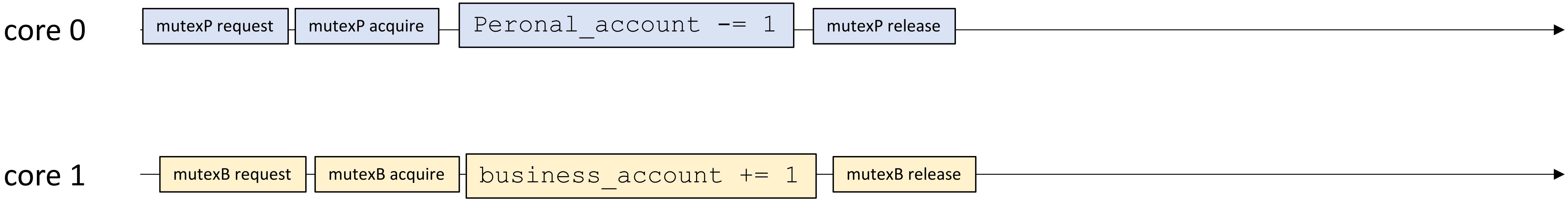


Multiple mutexes

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

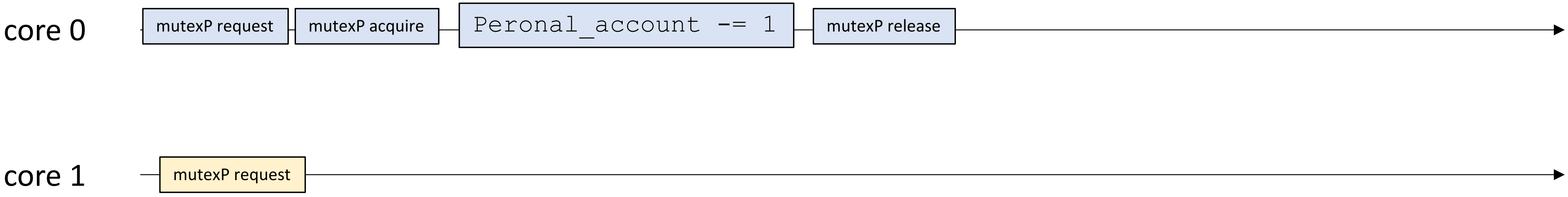


Multiple mutexes

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

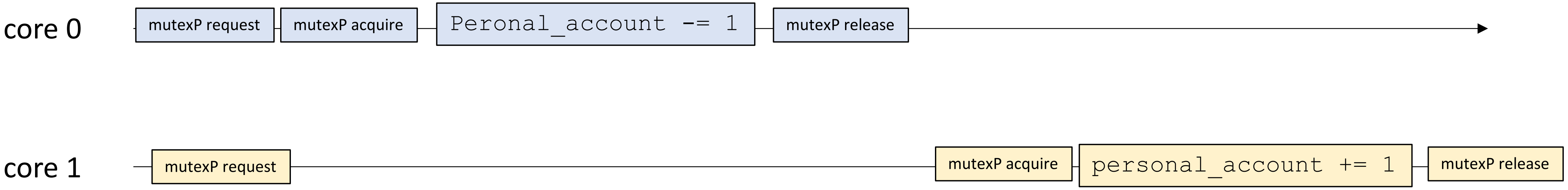


Multiple mutexes

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



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

```
void irs_audit() {
  for (int i = 0; i < NUM_AUDITS; i++) {
    tylers_personal_account_mutex.lock();
    tylers_business_account_mutex.lock();

    AUDIT(tylers_personal_account, tylers_business_account);

    tylers_personal_account_mutex.unlock();
    tylers_business_account_mutex.unlock();
  }
}
```

Multiple mutexes

```
void ucsc_audit() {
  for (int i = 0; i < NUM_AUDITS; i++) {
    tylers_business_account_mutex.lock();
    tylers_personal_account_mutex.lock();

    AUDIT(tylers_personal_account, tylers_business_account);

    tylers_personal_account_mutex.unlock();
    tylers_business_account_mutex.unlock();
  }
}
```

- 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!
The locks must be acquired in the same order across the application.

New material

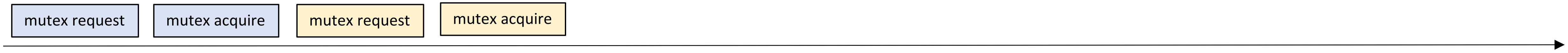
Properties of mutexes

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.

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

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

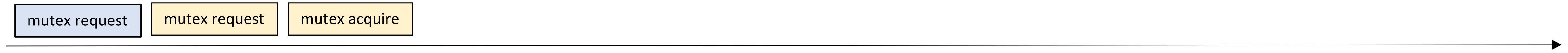
Properties of mutexes

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

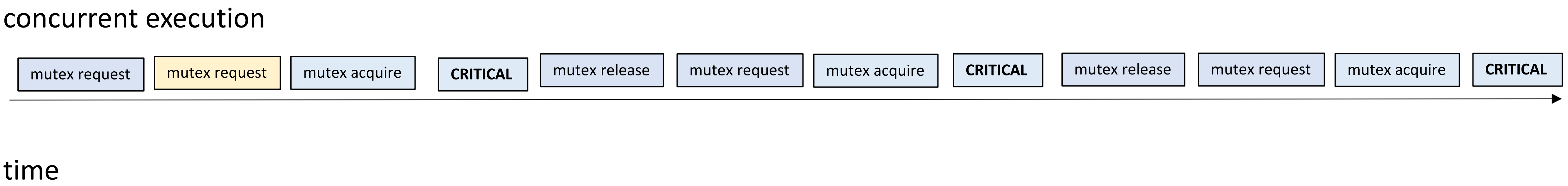
Program cannot hang here
Either thread 0 or thread 1 must acquire the mutex

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



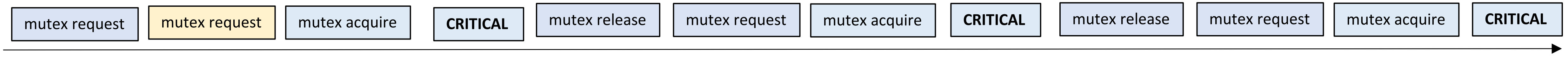
Properties of mutexes

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

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!

- Race-free program are safe! Use either locks or atomic variables.

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 for sequential code:
  - O(ITERS) 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

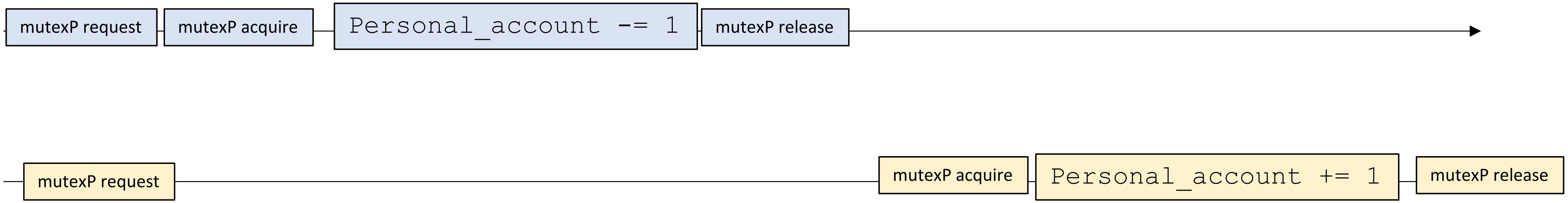
- Compiler can be aggressive with memory operations:
  - For non-atomic memory locations, the following optimizations are valid

- 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

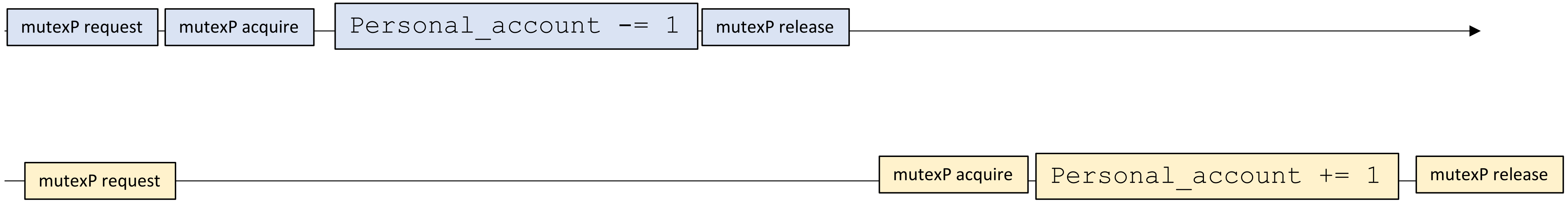


because this thread needs to see the updated view

Atomic properties

- Compiler Fence

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

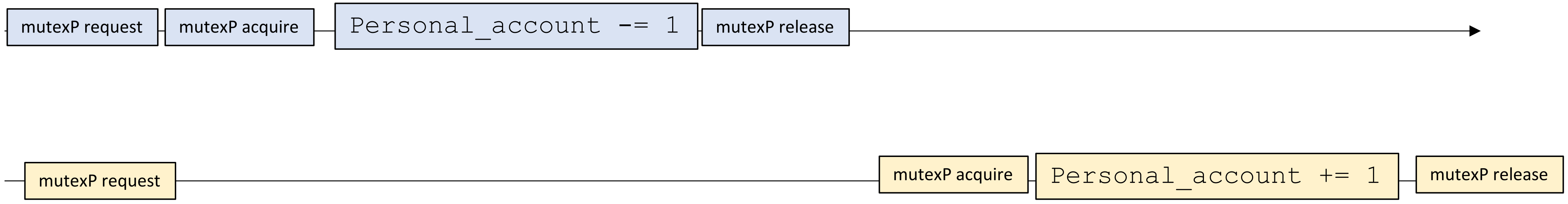


Atomic properties

- Compiler Fence

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

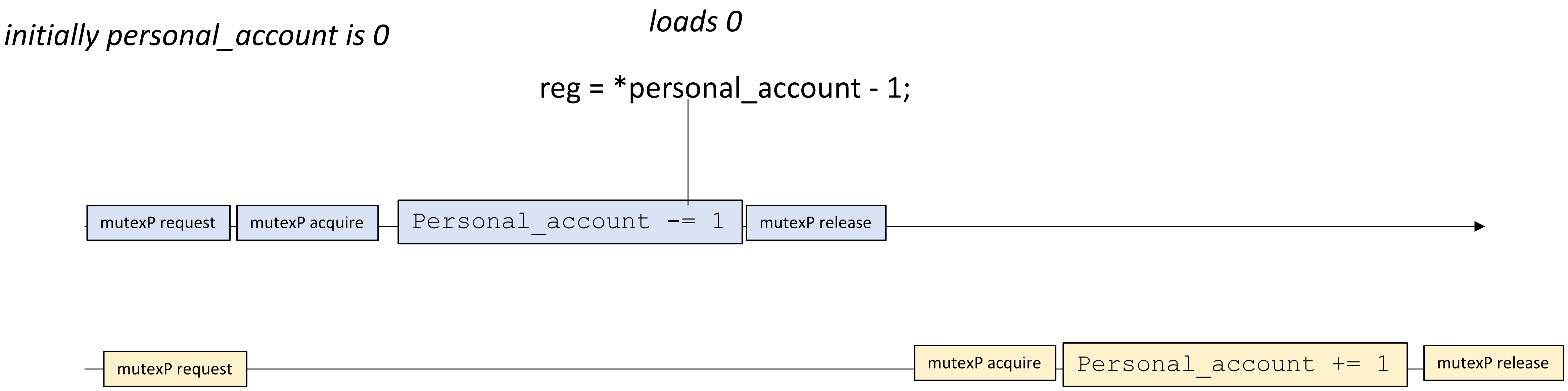
initially personal_account is 0



Atomic properties

- Compiler Fence

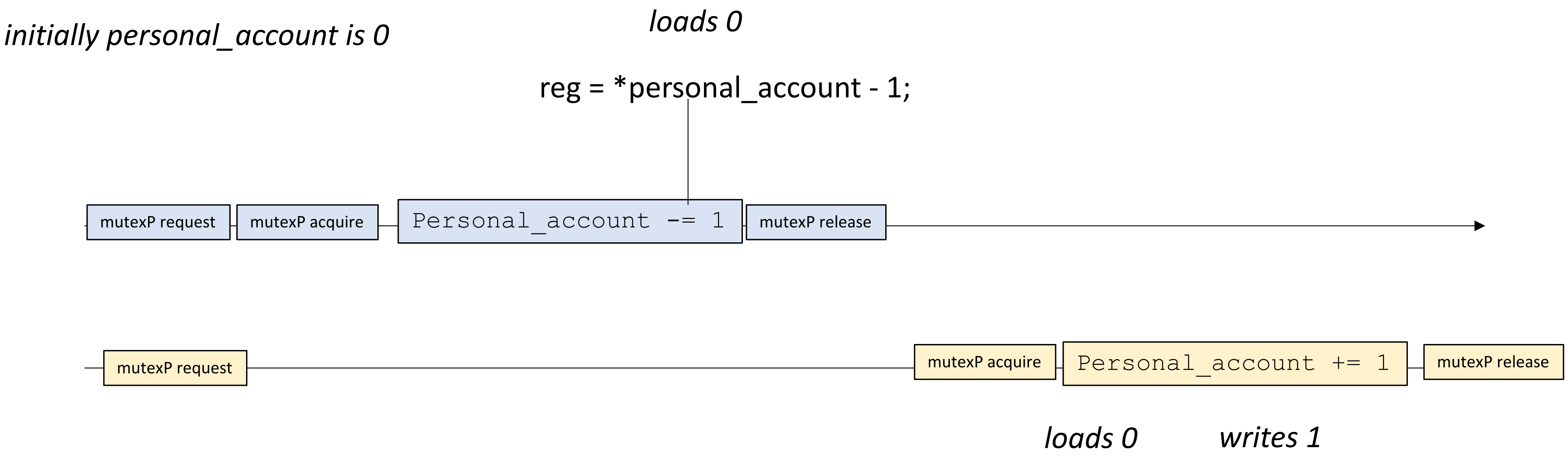
what can go wrong if the compiler doesn't write values to memory?



Atomic properties

- Compiler Fence

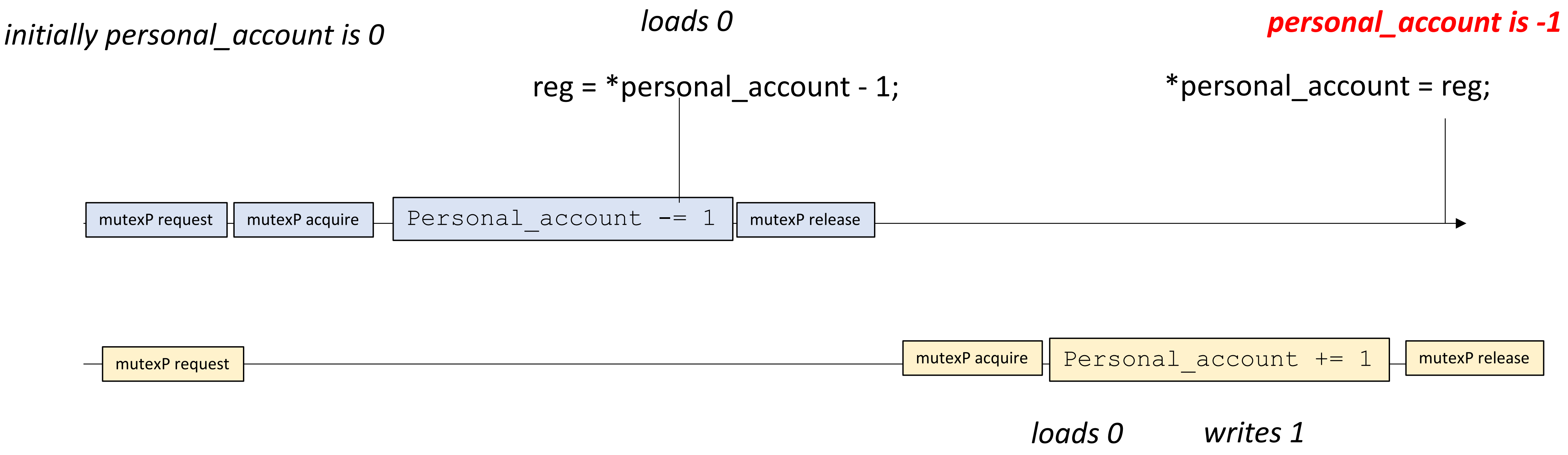
what can go wrong if the compiler doesn't write values to memory?



Atomic properties

- Compiler Fence

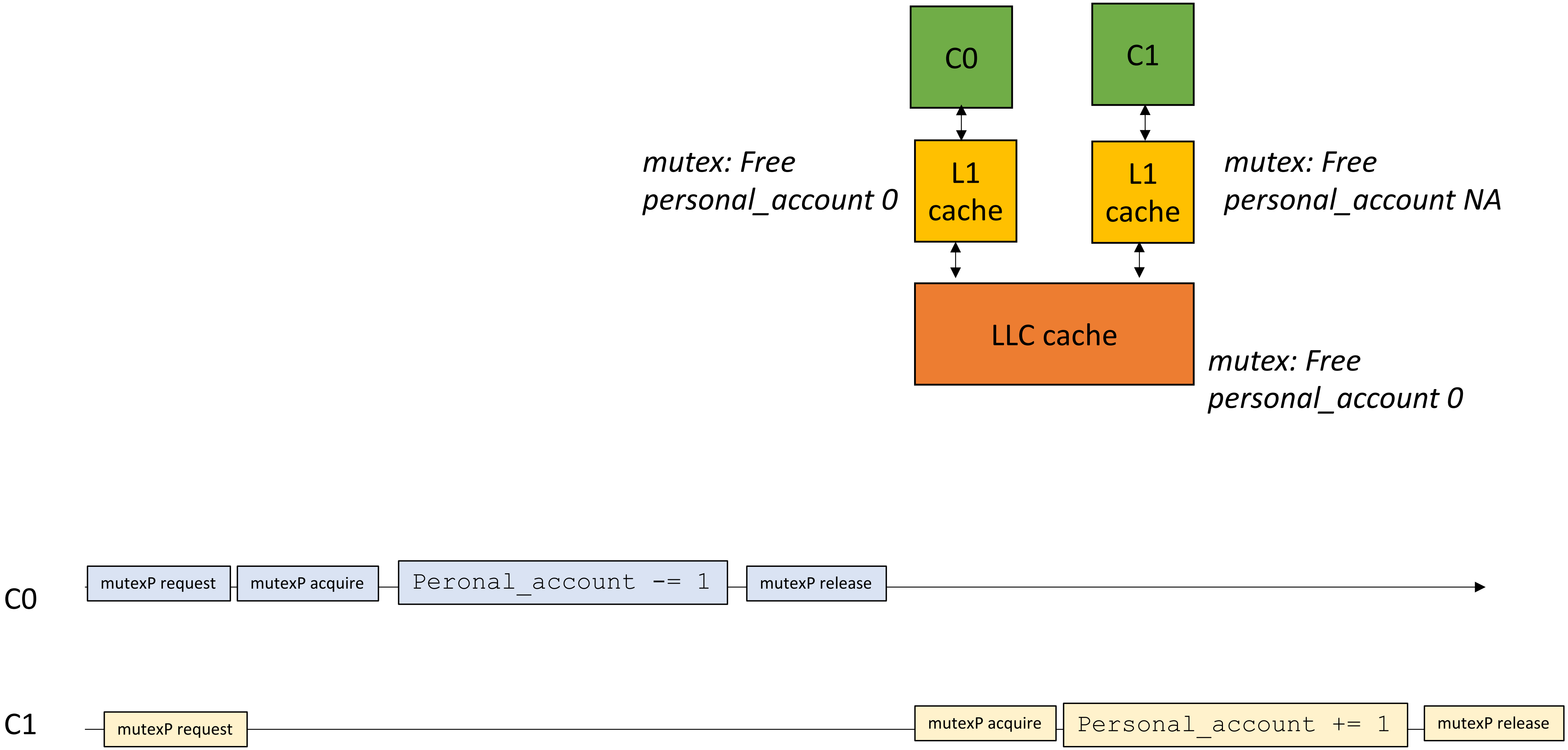
what can go wrong if the compiler doesn't write values to memory?



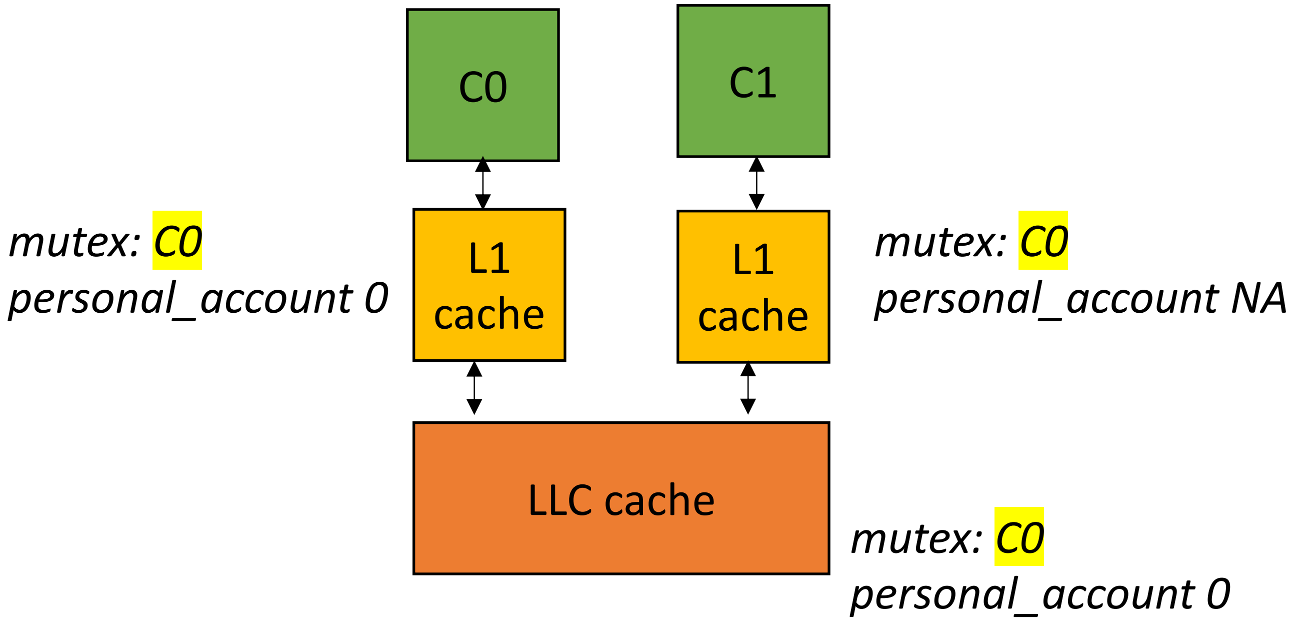
Atomic properties

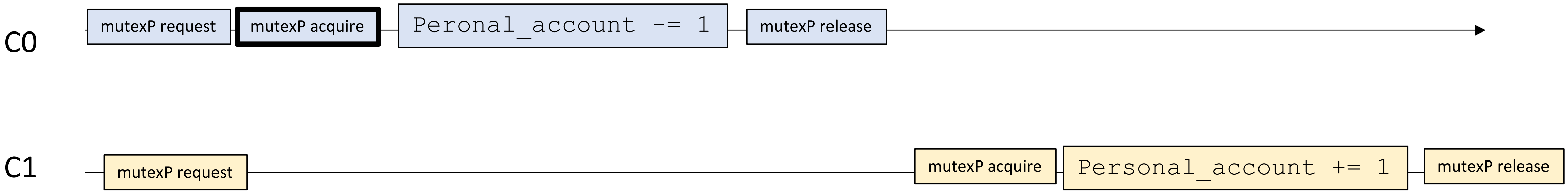
- Also provides a memory barrier

- Memory Fence (or Memory Barrier)

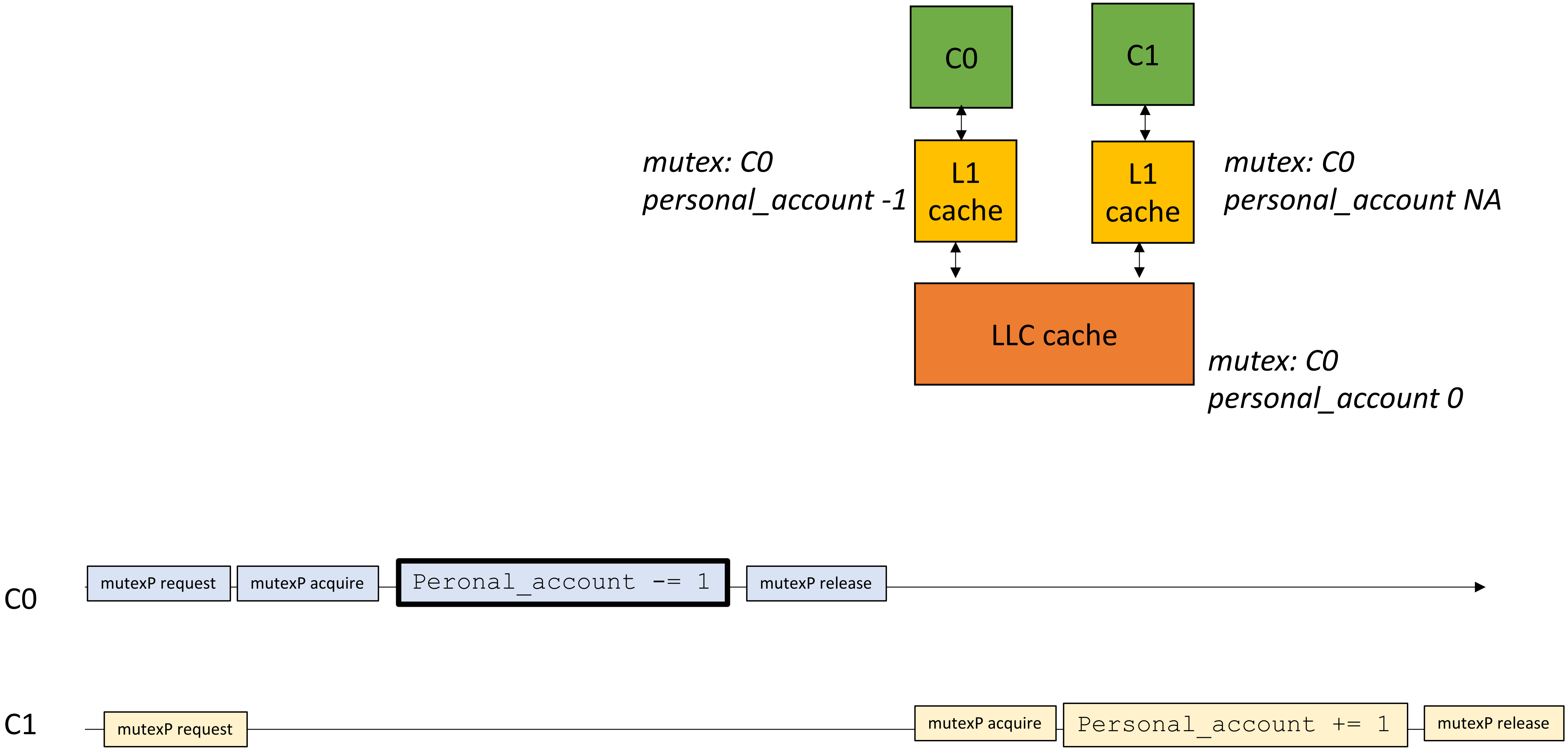


- Memory Fence (or Memory Barrier)

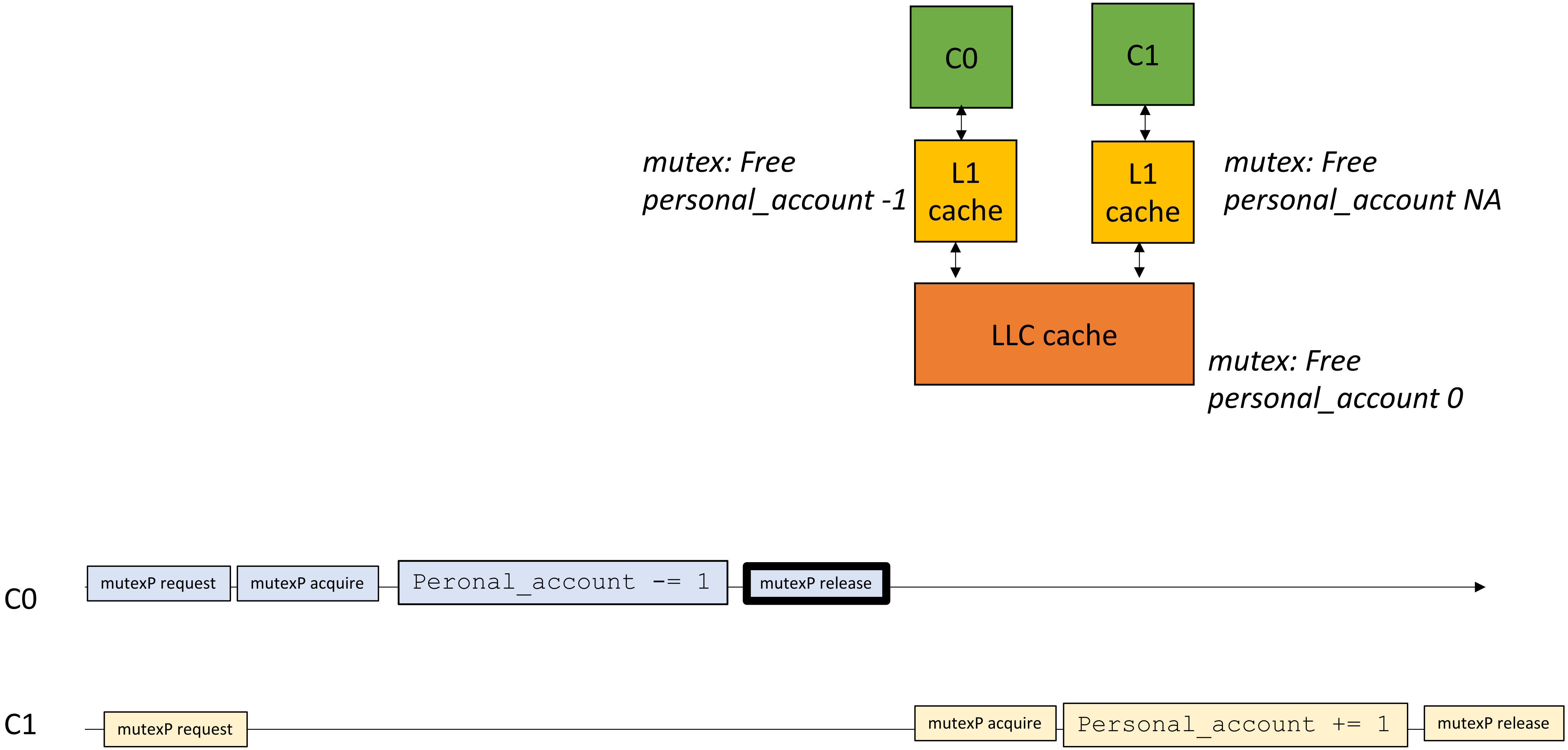




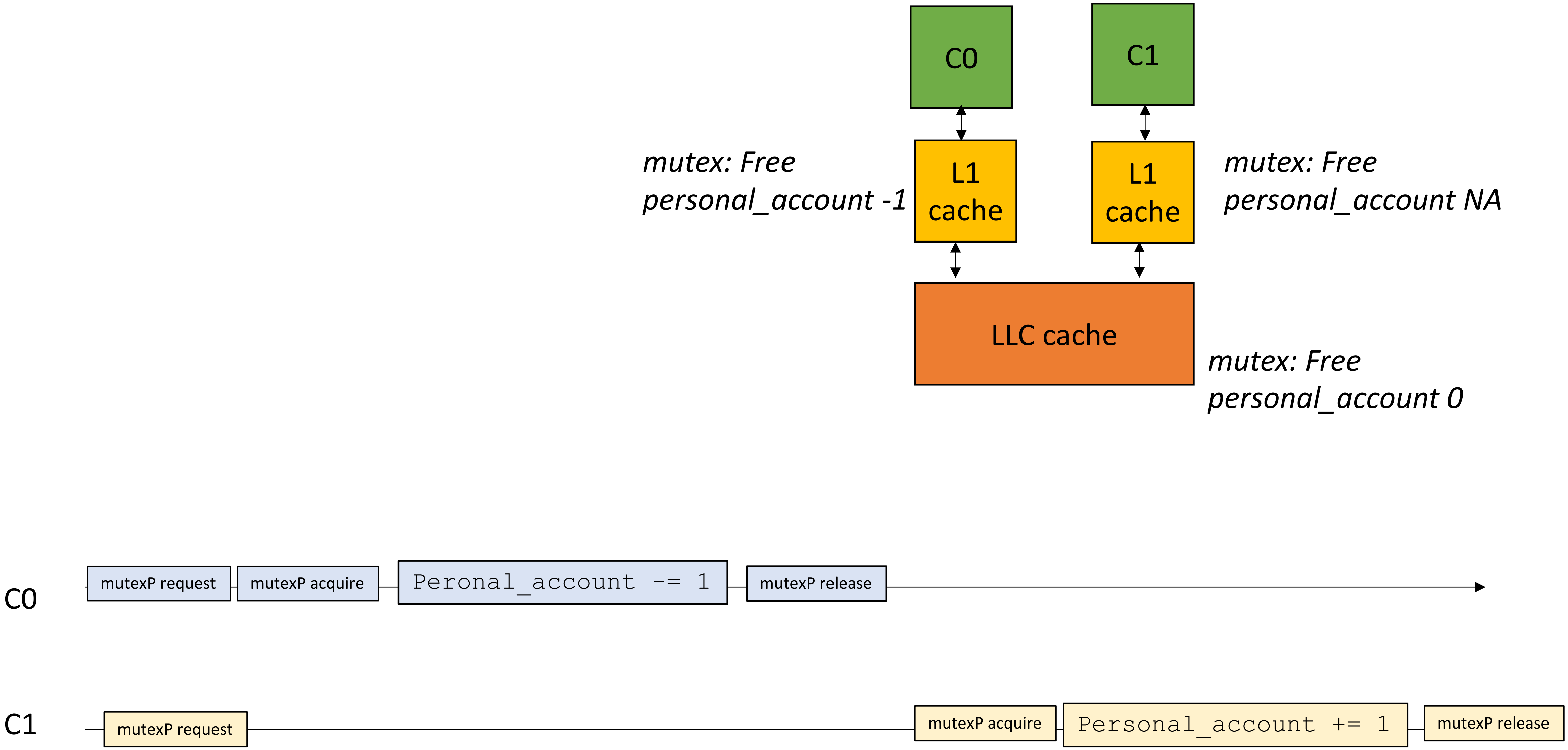
- Memory Fence (or Memory Barrier)



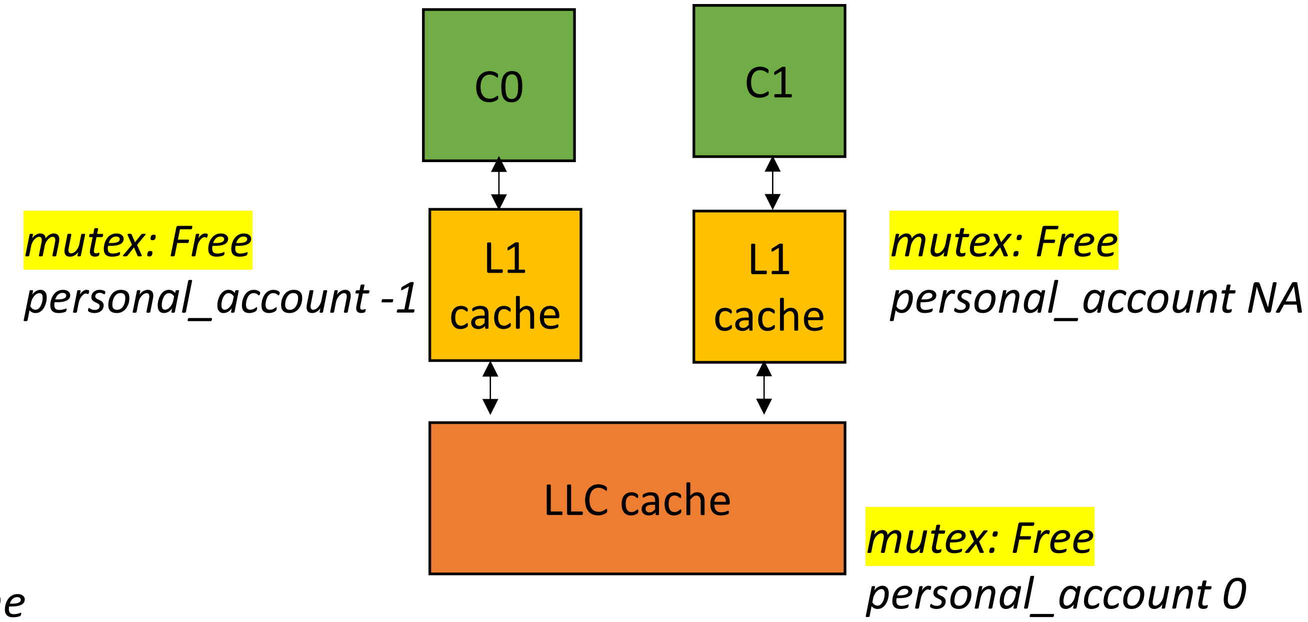
- Memory Fence (or Memory Barrier)



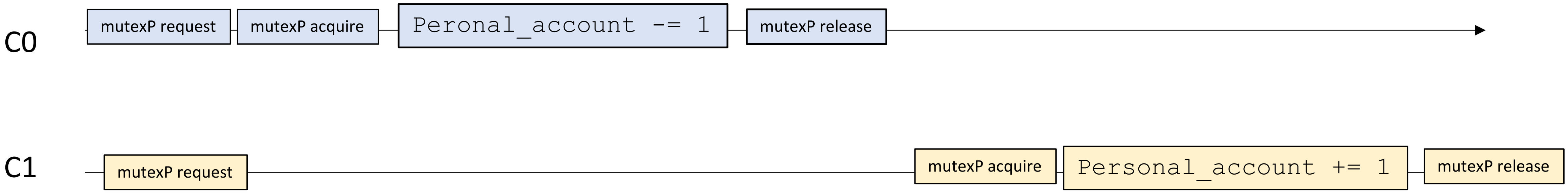
- Memory Fence (or Memory Barrier)



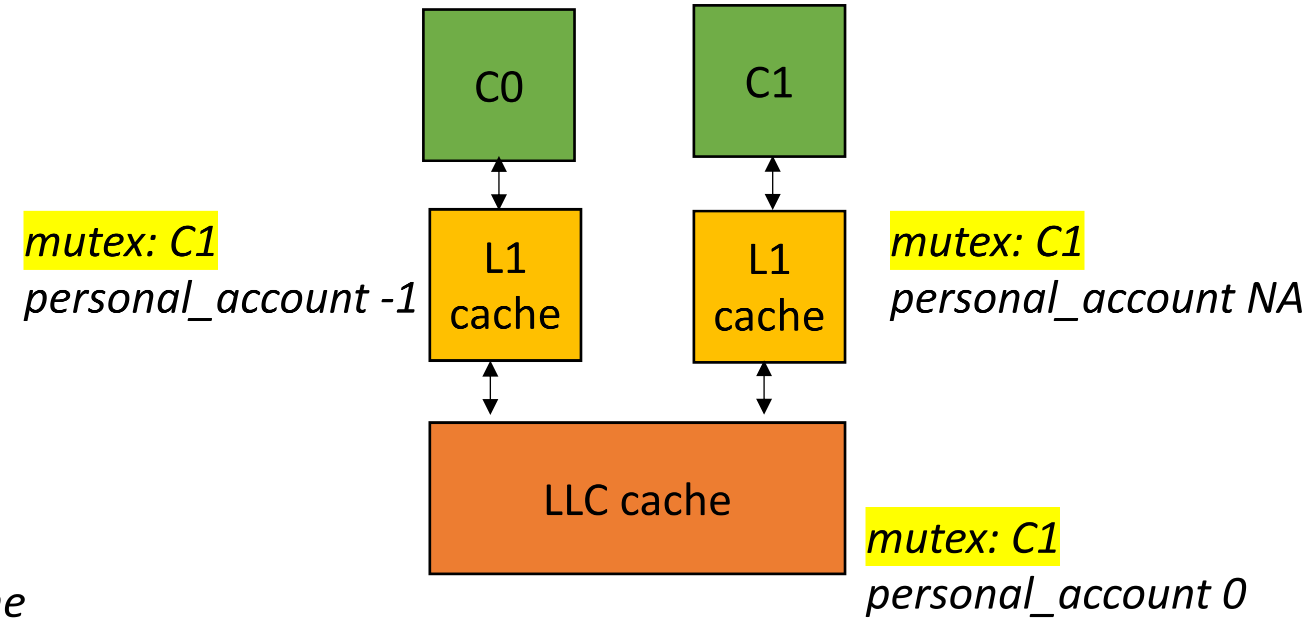
- Memory Fence (or Memory Barrier)



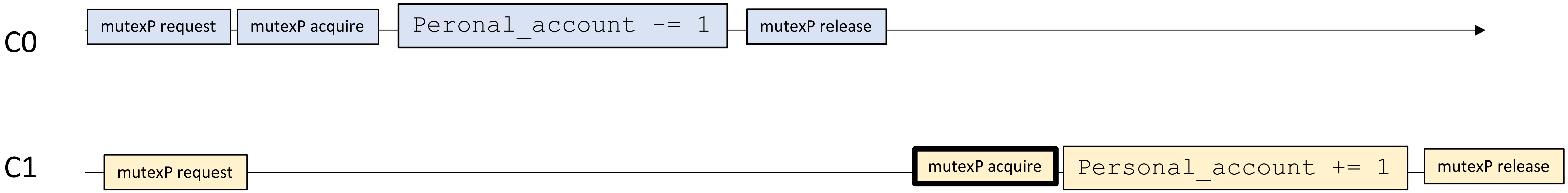
Possible for the mutex message to travel through the cache before the personal_account information!



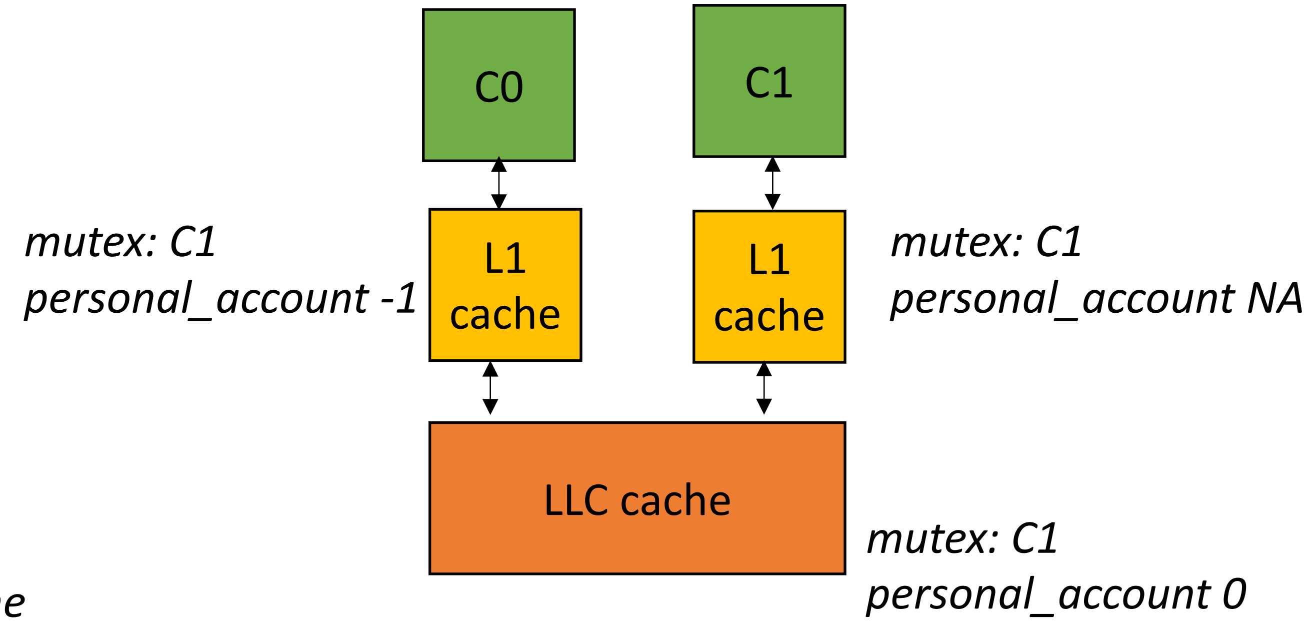
- Memory Fence (or Memory Barrier)



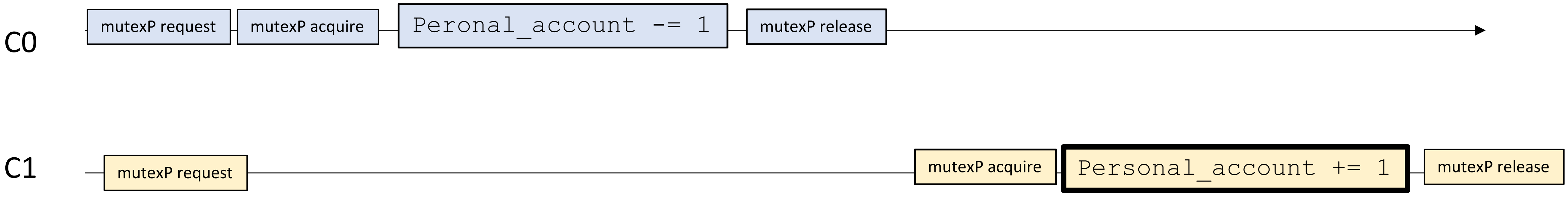
Possible for the mutex message to travel through the cache before the personal_account information!



- Memory Fence (or Memory Barrier)



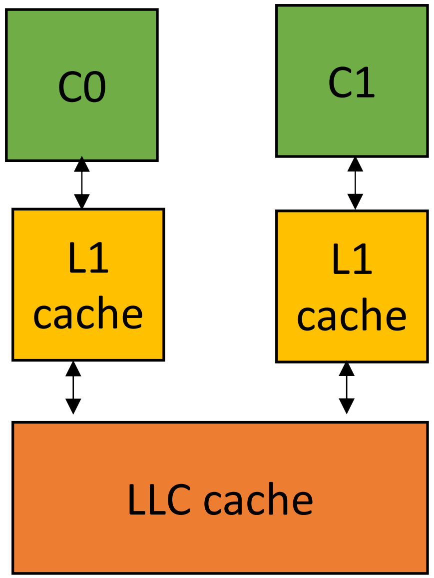
Possible for the mutex message to travel through the cache before the personal_account information!



- Memory Fence (or Memory Barrier)

stale value!

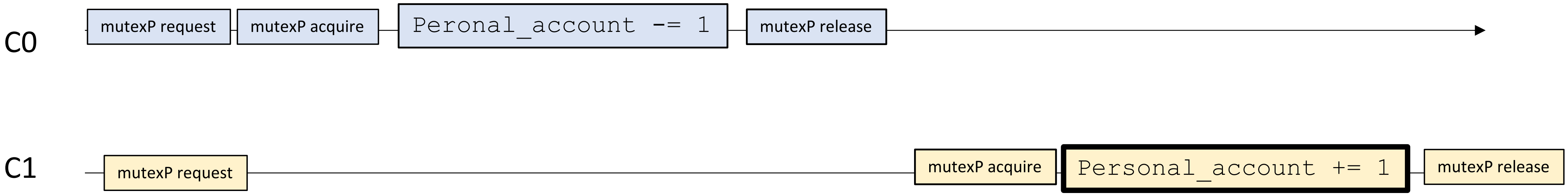
mutex: C1
personal_account -1



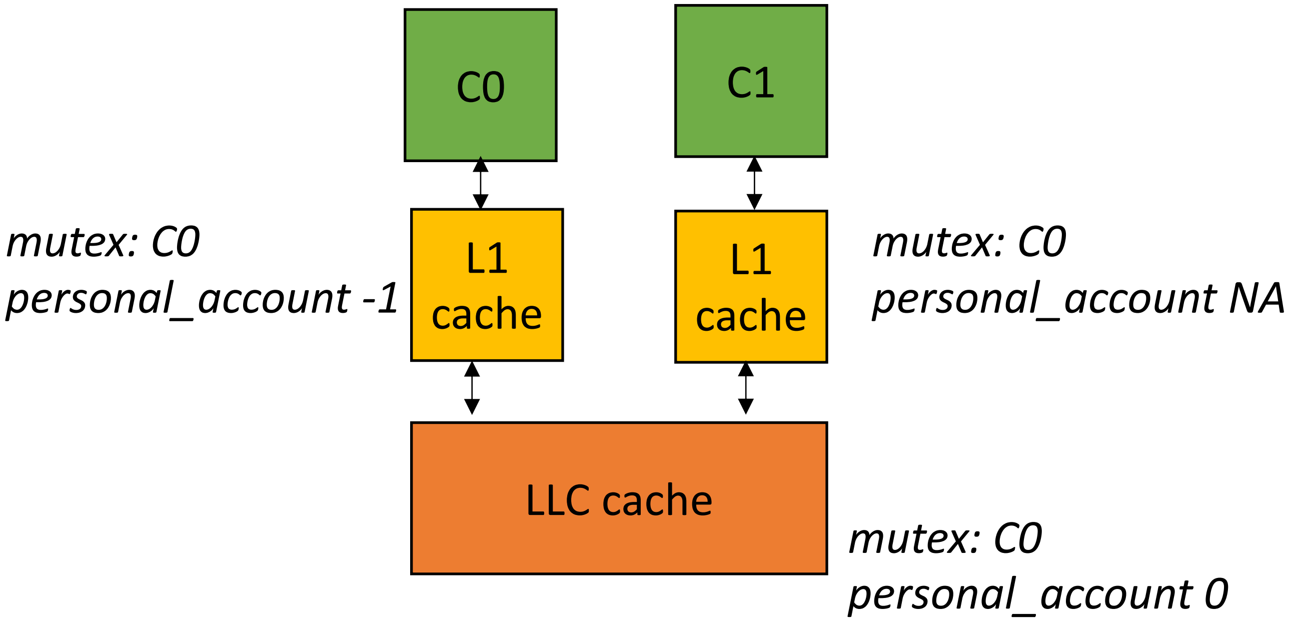
mutex: C1
personal_account 0

mutex: C1
personal_account 0

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

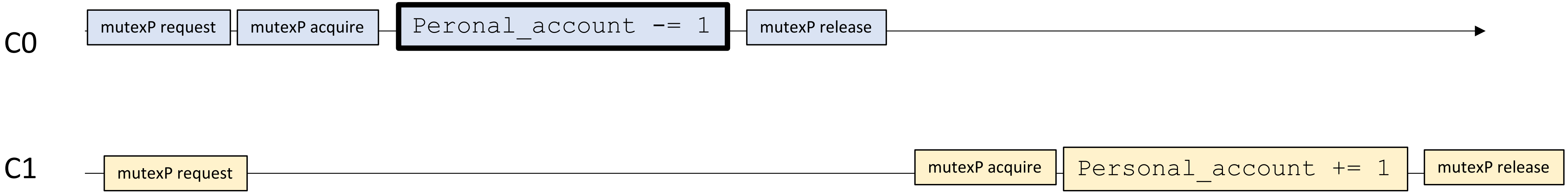


- Memory Fence (or Memory Barrier)

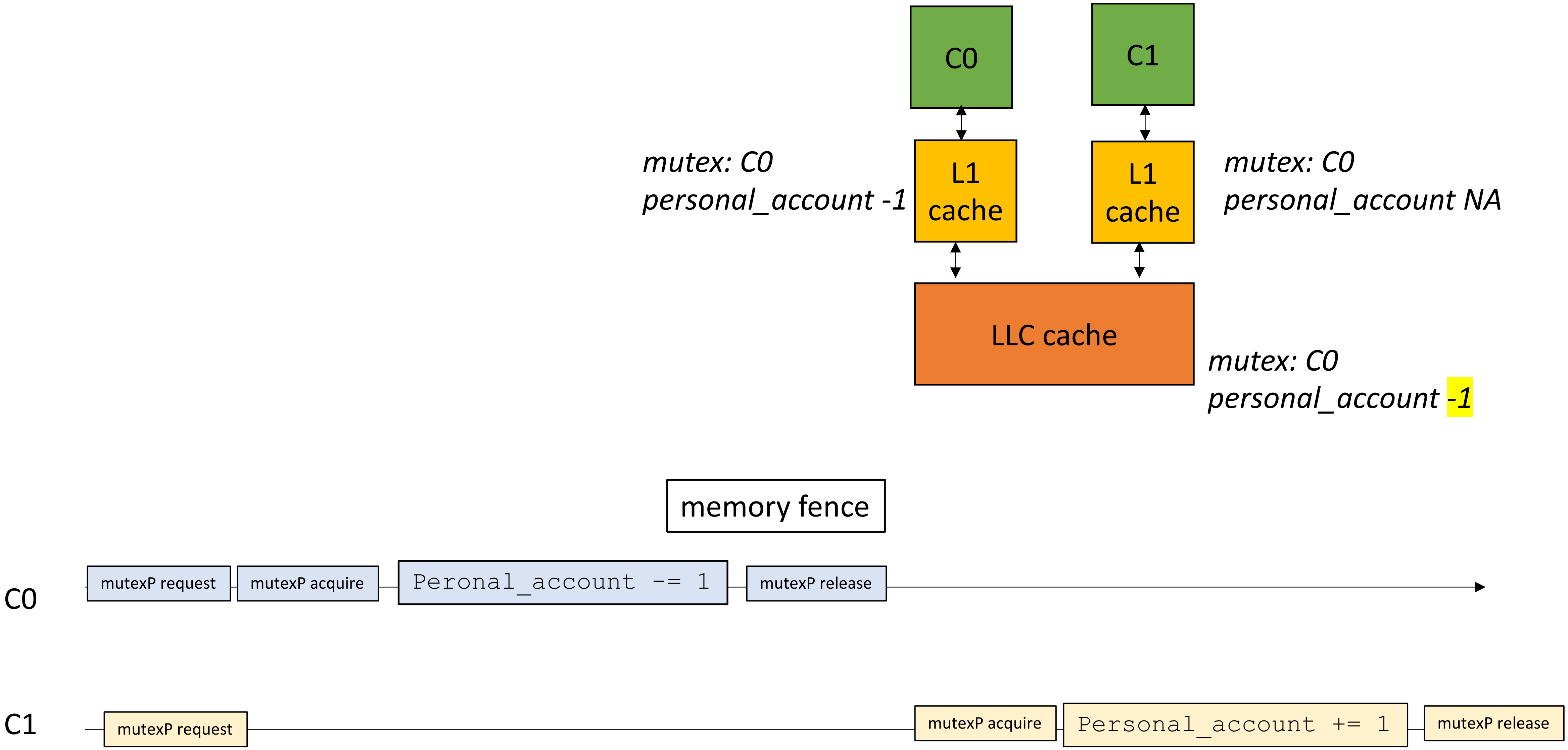


rewind

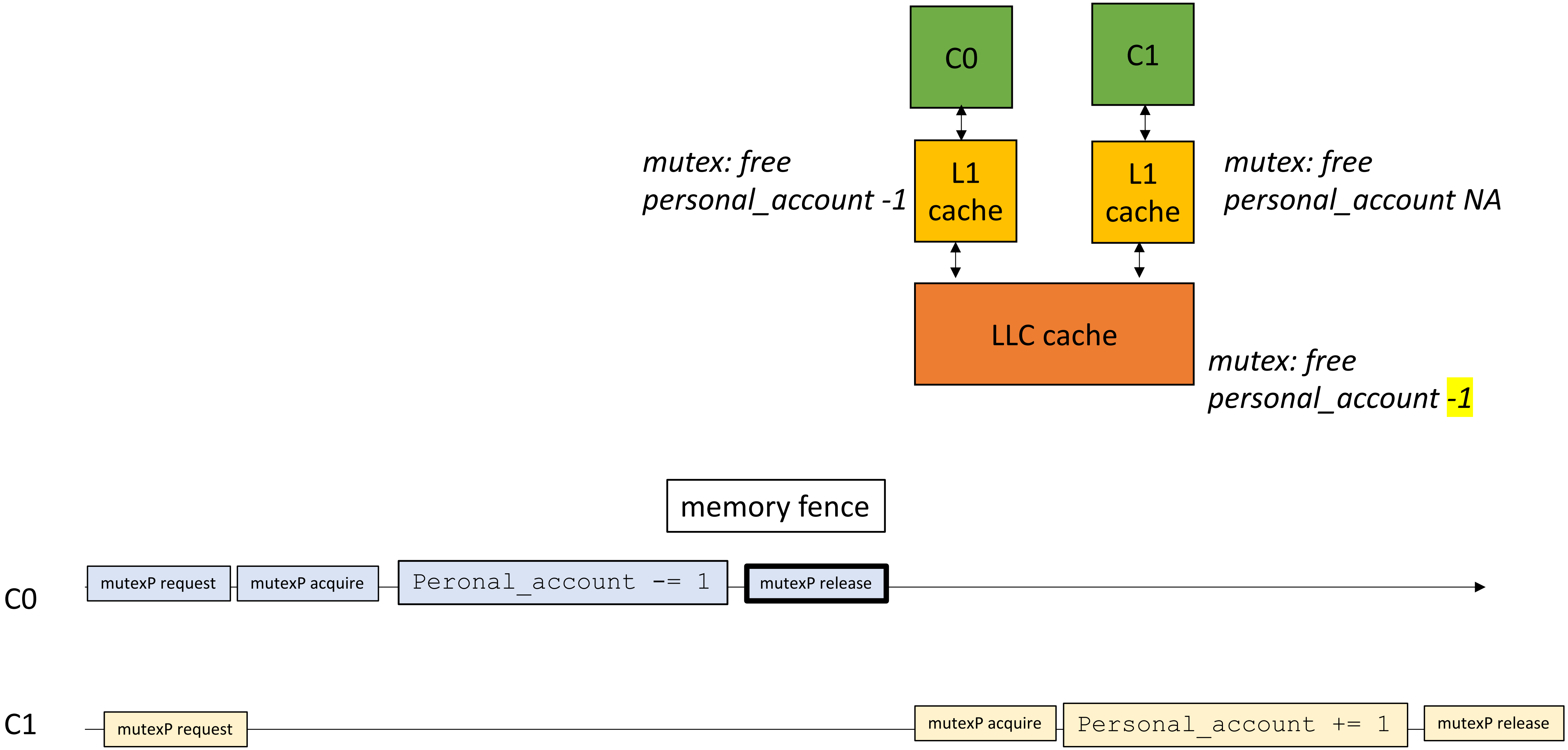
memory fence



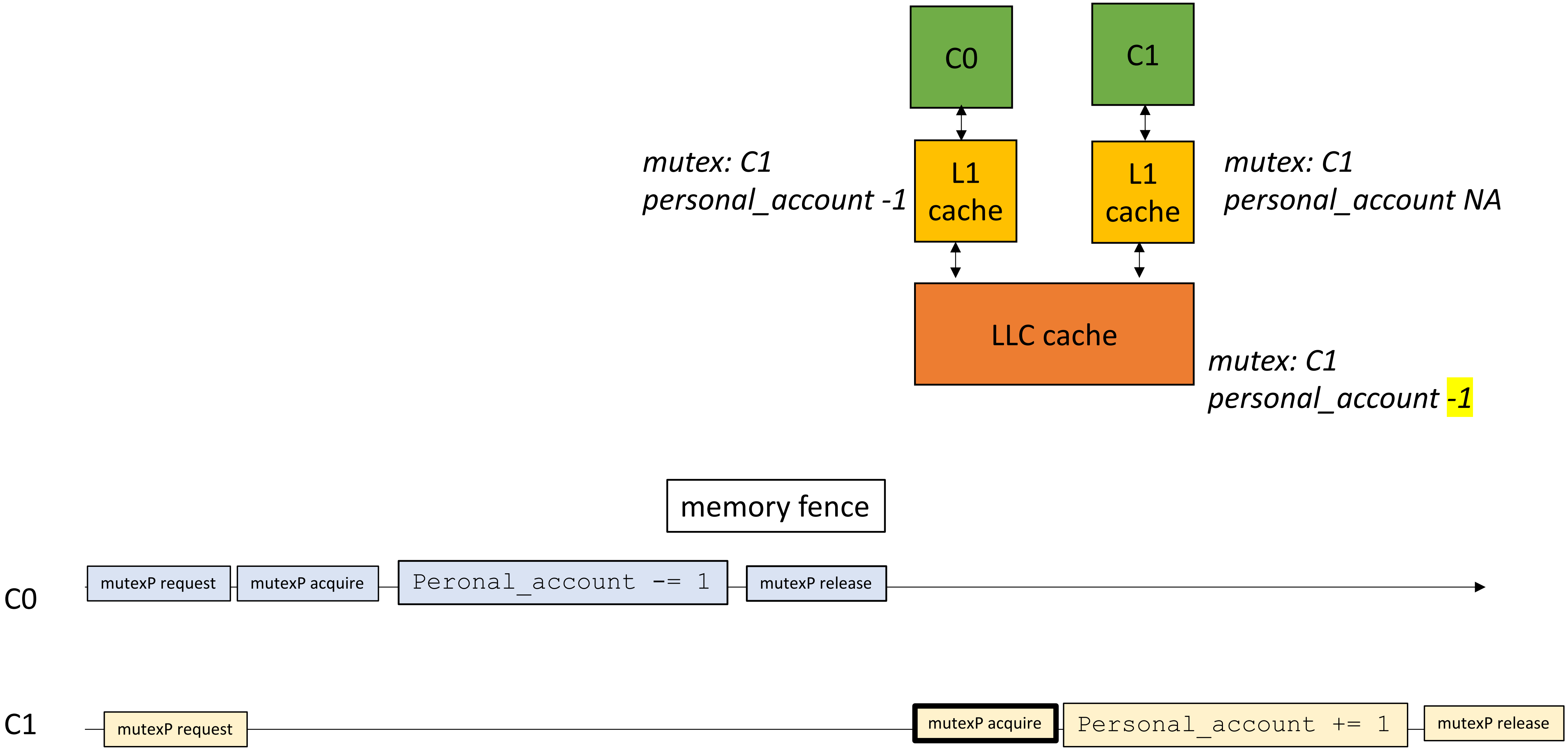
- Memory Fence (or Memory Barrier)



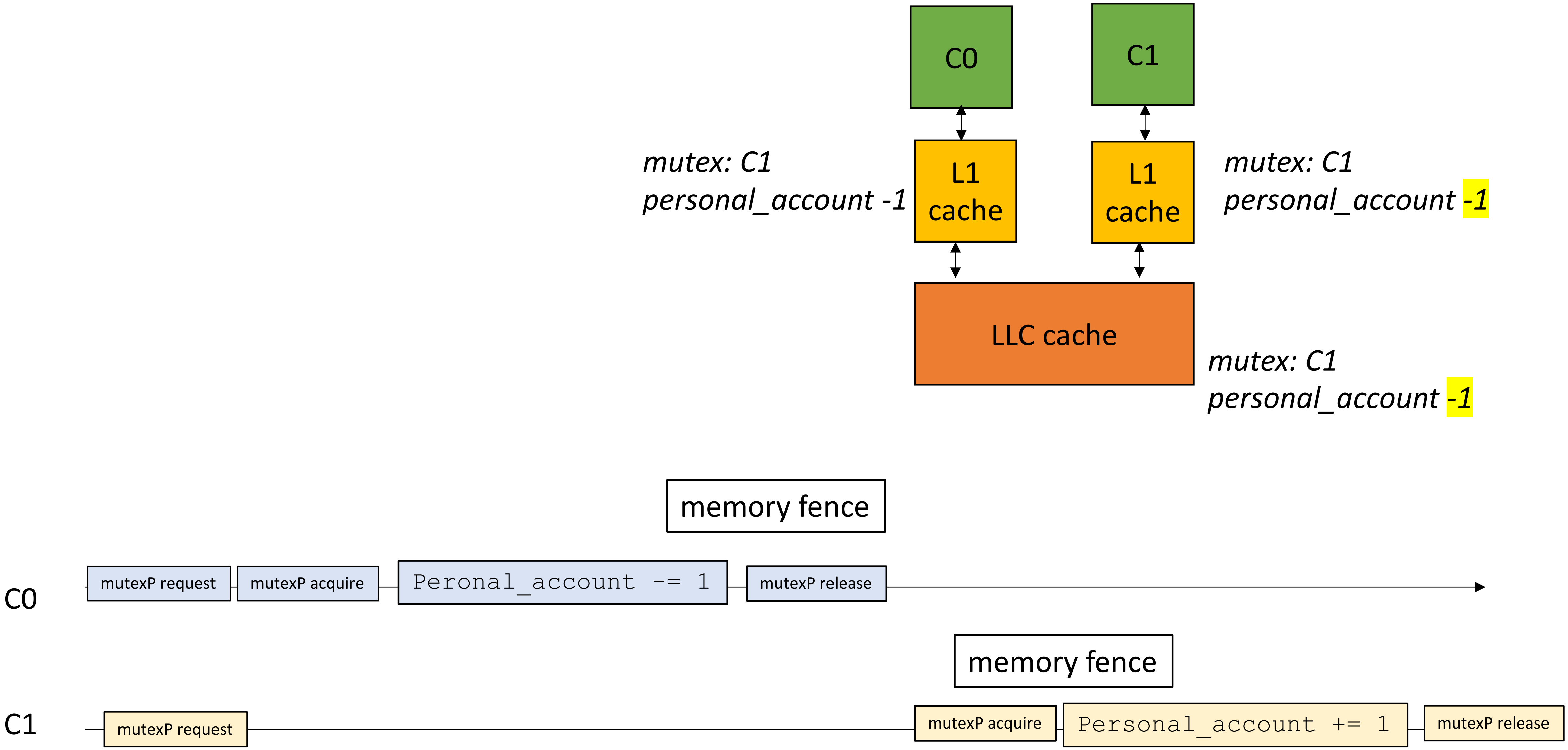
- Memory Fence (or Memory Barrier)



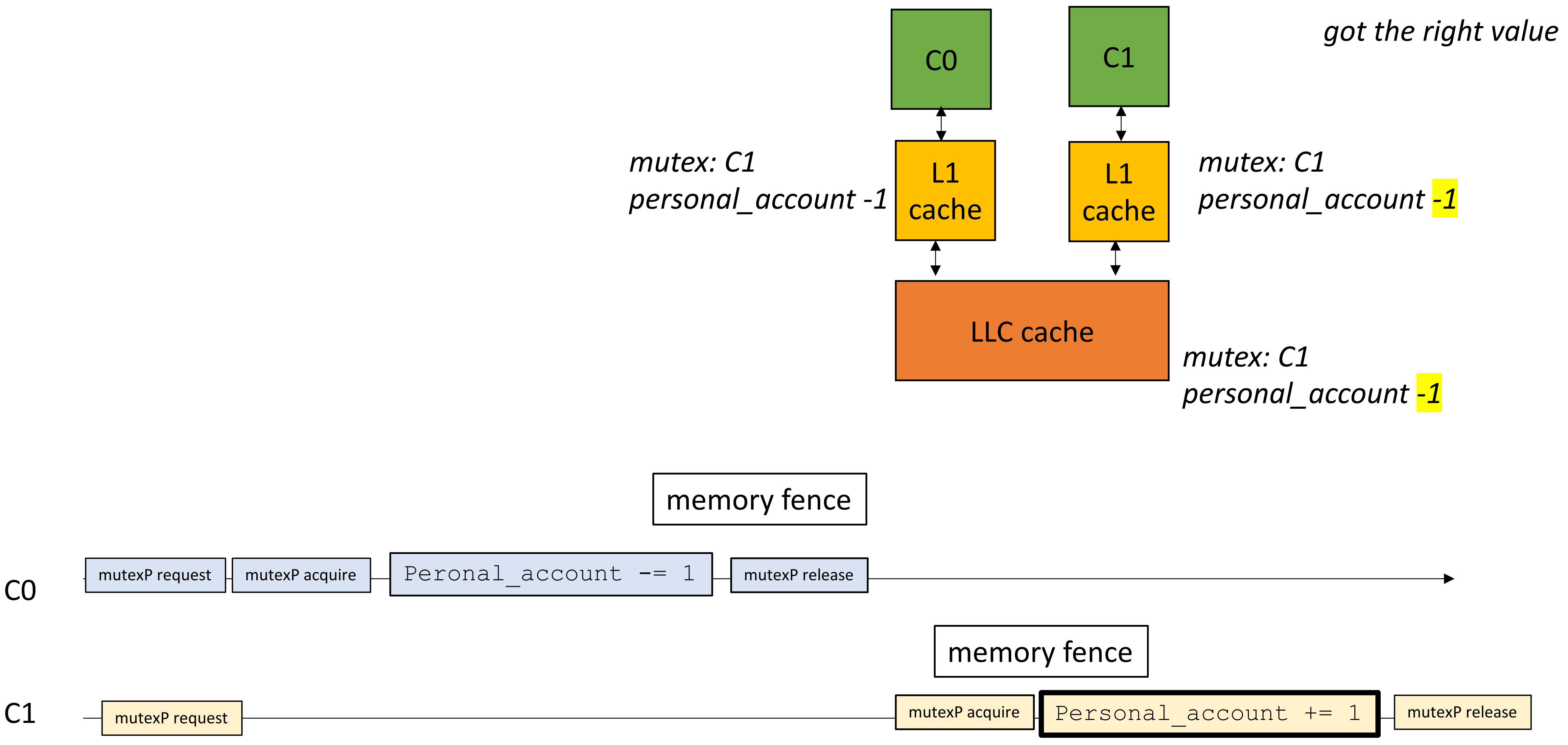
- Memory Fence (or Memory Barrier)



- 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

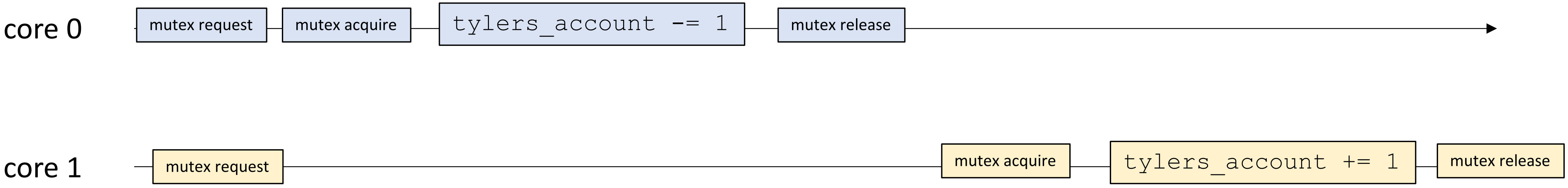
ARM and PowerPC let cache values flow out-of-order
GPUs let caches flow out-of-order

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

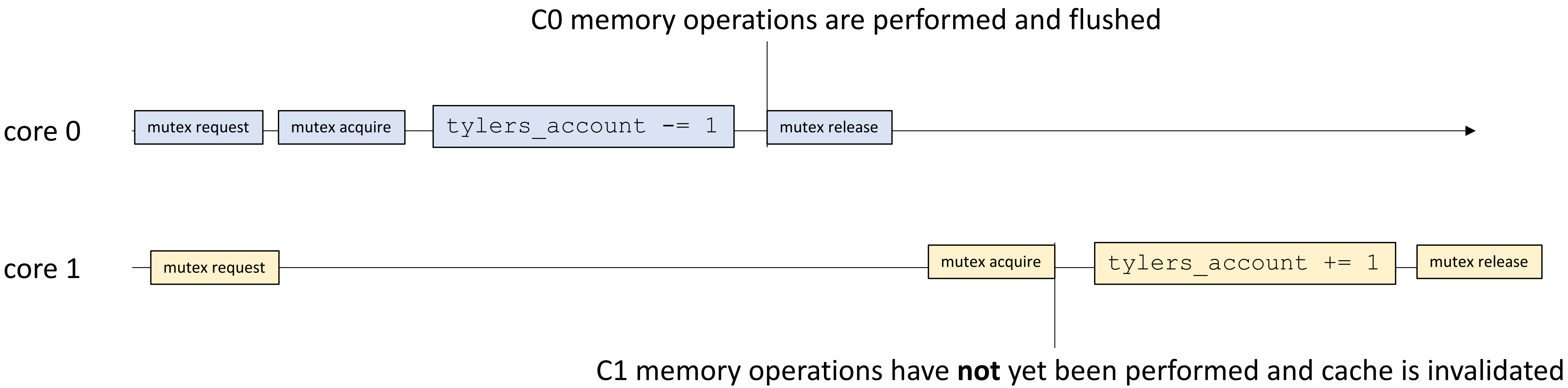
- What do those fences (compiler and memory) give us?
- Atomics were designed so that we can implement things like mutexes!



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 acquire the mutex, you wait until it is set to 0, then you store 1 in it.
  - To release 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

What’s 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);
}
```

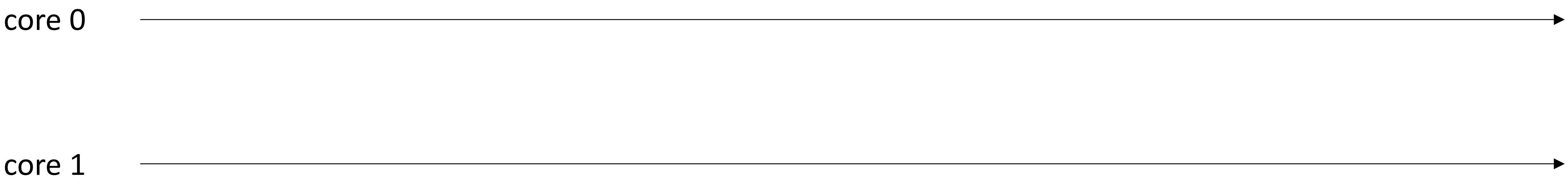
```
void unlock() {
  flag.store(0);
}
```

Thread 0:

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

Thread 1:

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



Analysis

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

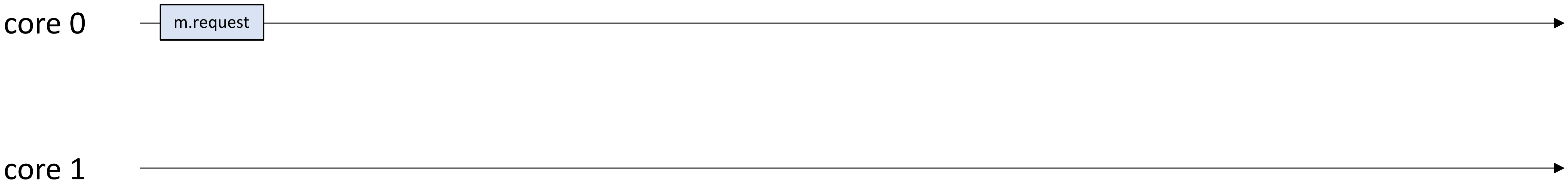
```
void unlock() {
  flag.store(0);
}
```

Thread 0:

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

Thread 1:

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



Analysis

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

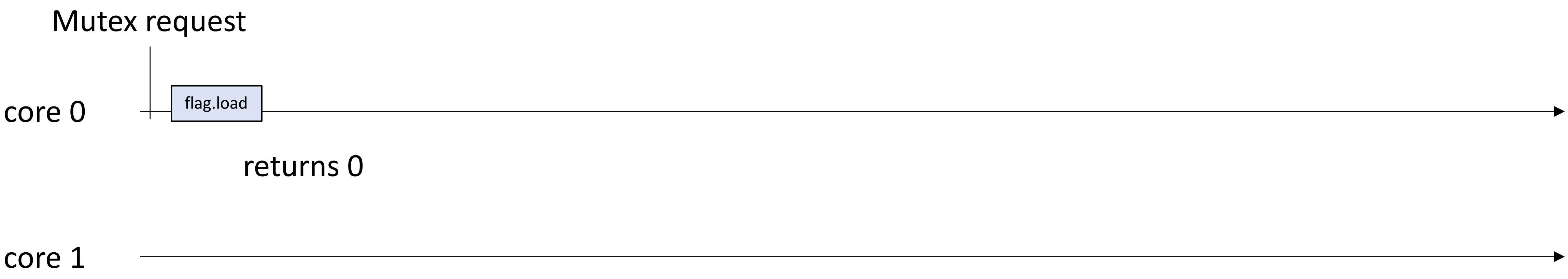
```
void unlock() {
  flag.store(0);
}
```

Thread 0:

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

Thread 1:

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



Analysis

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

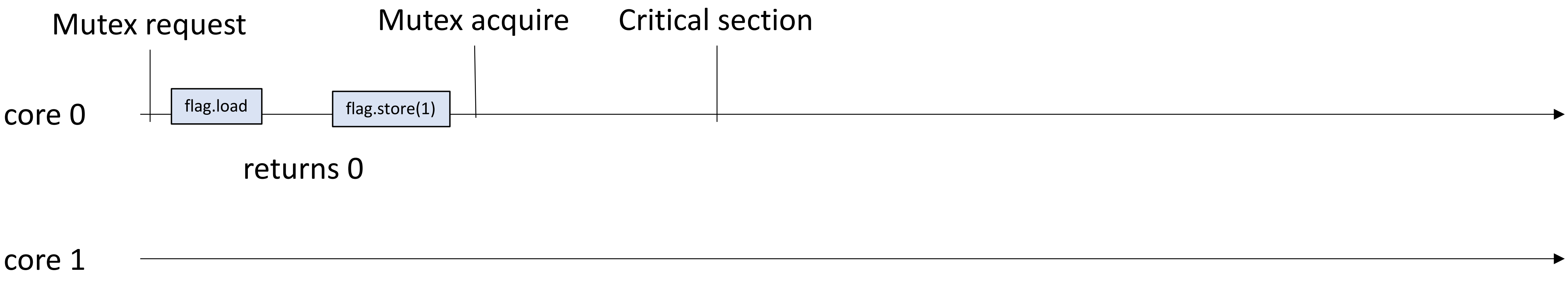
```
void unlock() {
  flag.store(0);
}
```

Thread 0:

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

Thread 1:

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



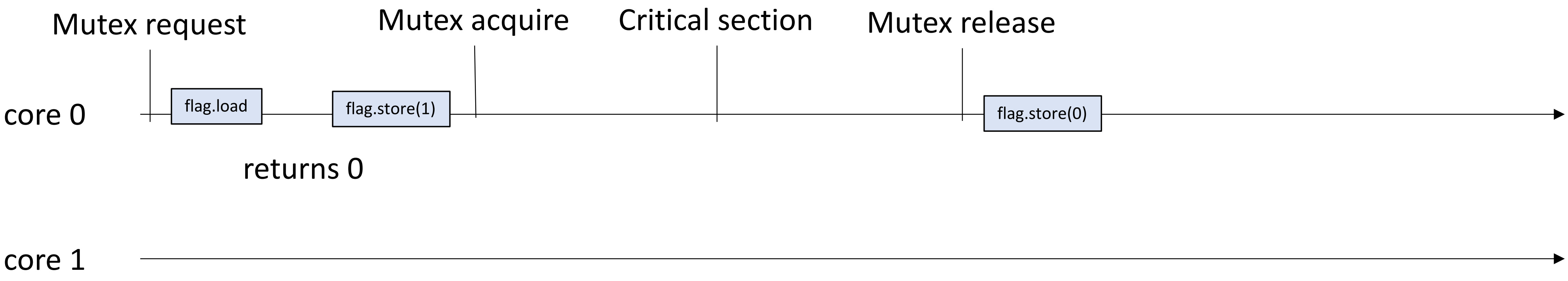
Analysis

```
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);
}
```

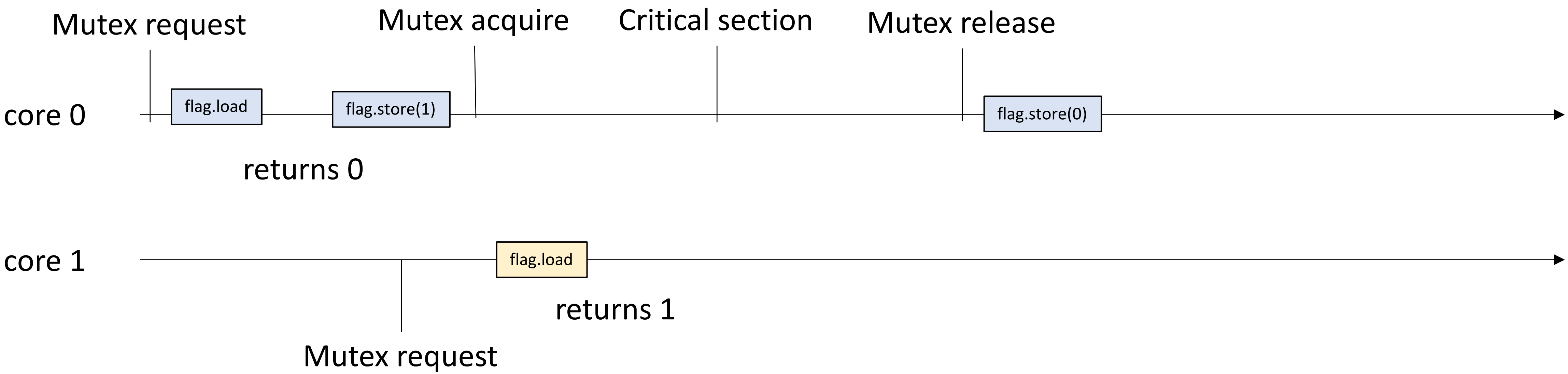
```
void unlock() {
  flag.store(0);
}
```

Thread 0:

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

Thread 1:

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



Analysis

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

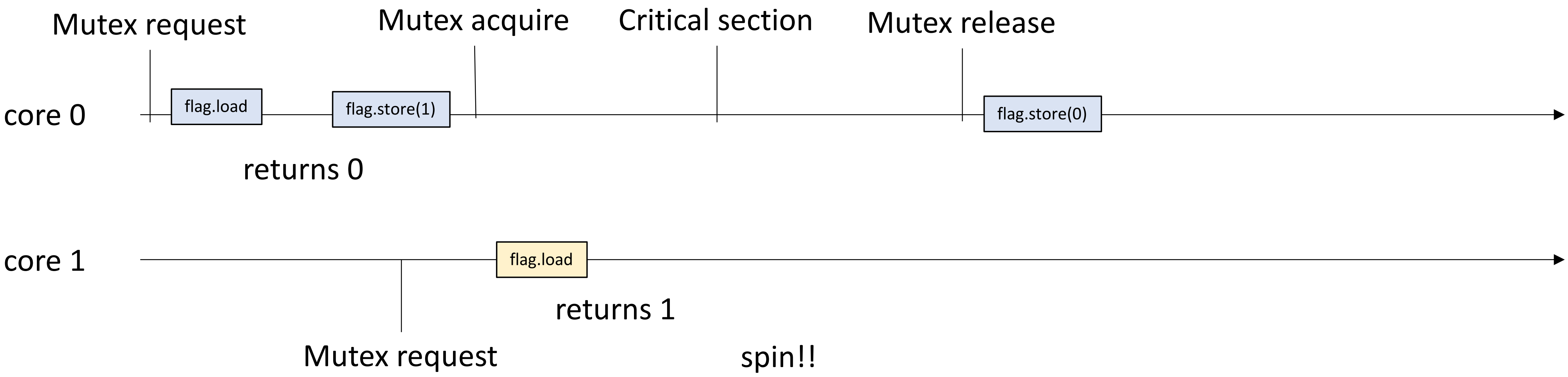
```
void unlock() {
  flag.store(0);
}
```

Thread 0:

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

Thread 1:

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



Analysis

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

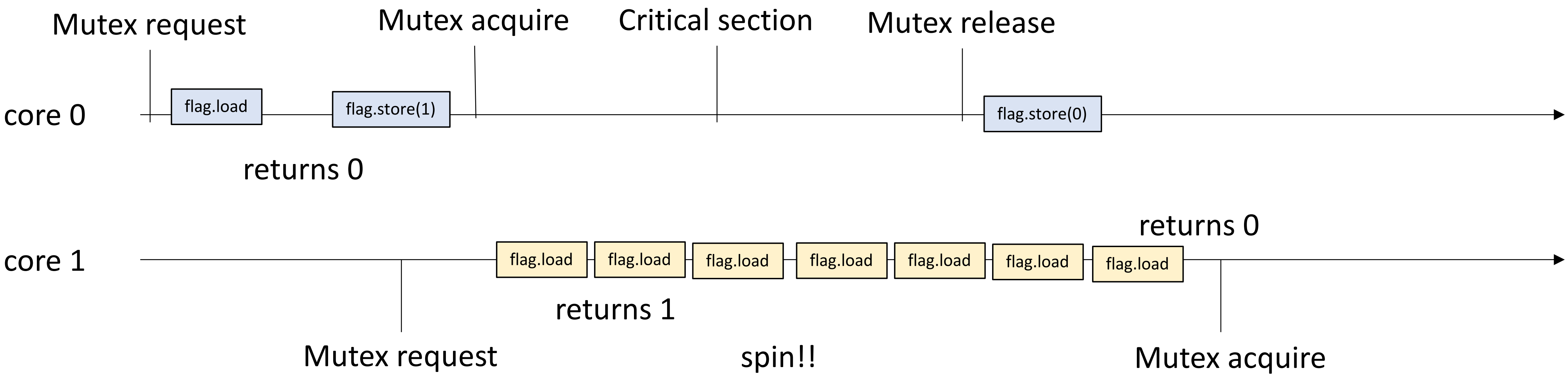
```
void unlock() {
  flag.store(0);
}
```

Thread 0:

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

Thread 1:

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



Analysis

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

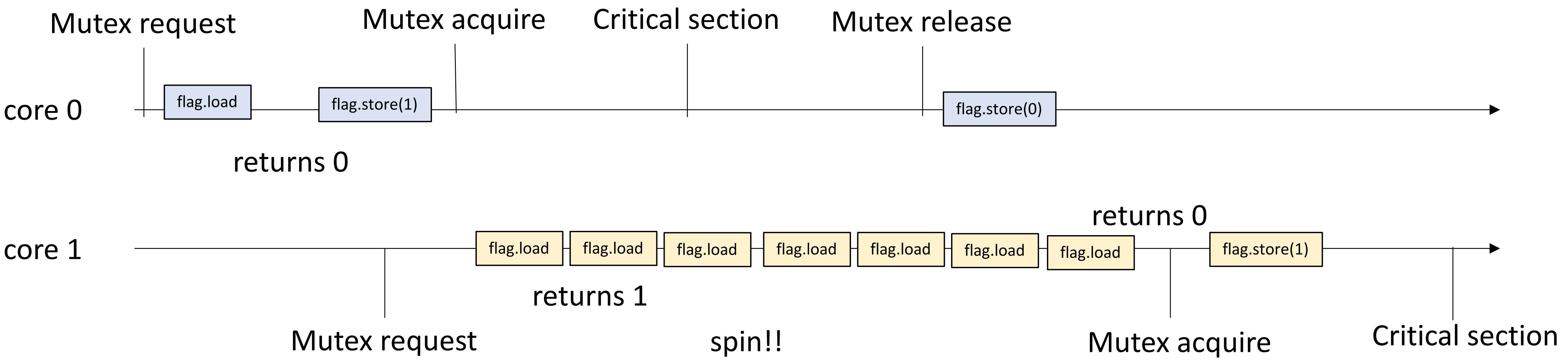
```
void unlock() {
  flag.store(0);
}
```

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);
}
```

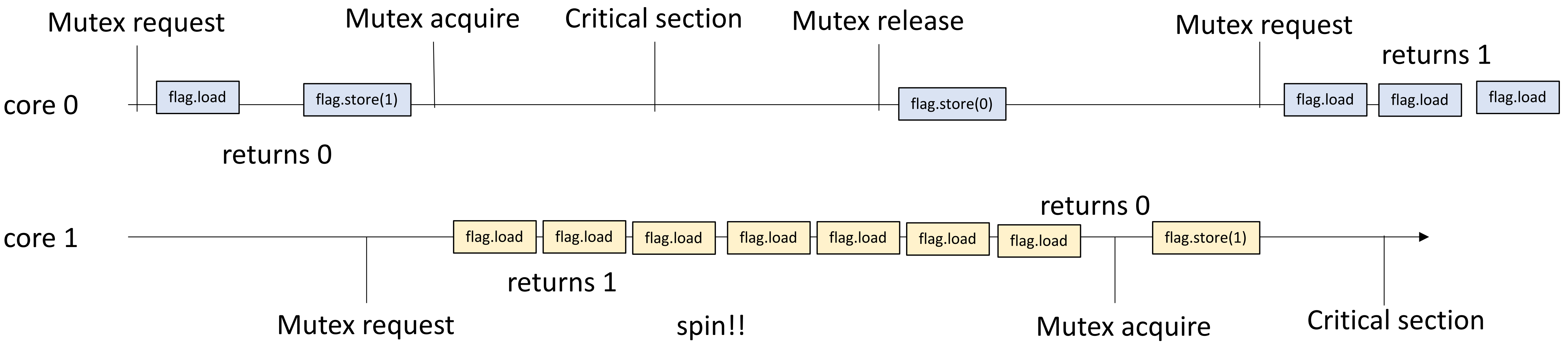
Thread 0:

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

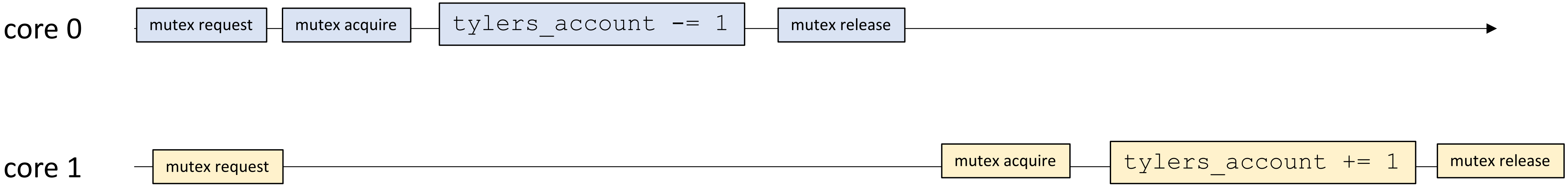
Thread 1:

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

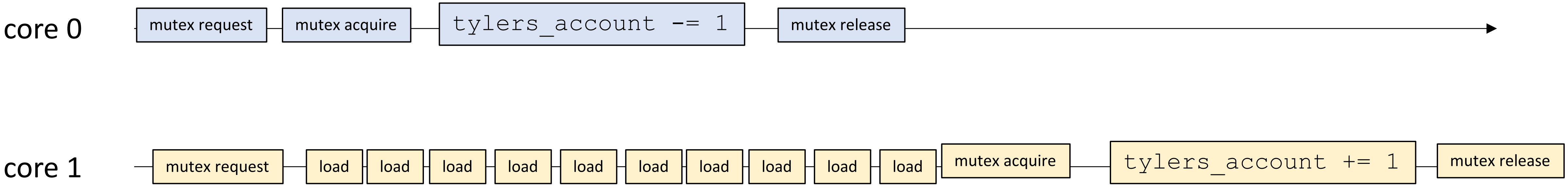
Mutual Exclusion property!
critical sections do not overlap!



Recall our previous analysis. What was core 1 probably doing?



Recall our previous analysis. What was core 1 probably doing?



Analysis

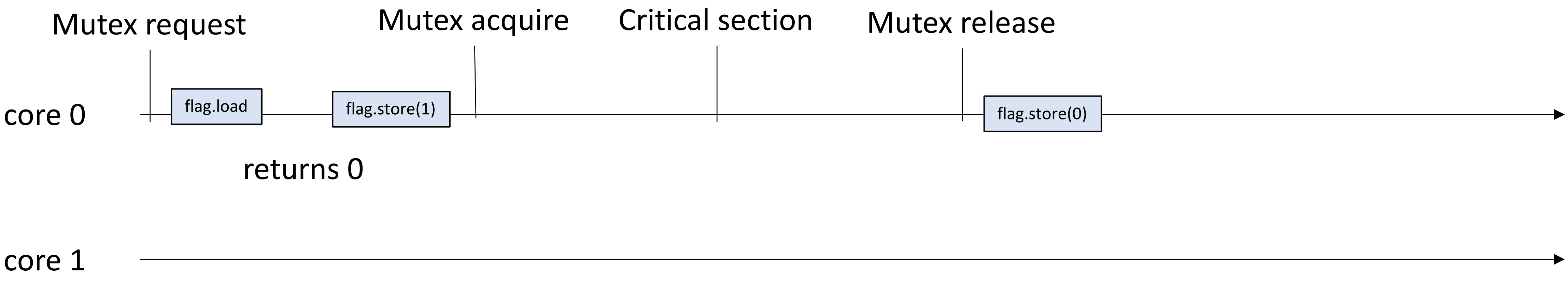
```
void lock() {
  while (flag.load() == 1);
  flag.store(1);
}
```

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

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

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

Lets try another interleaving



Analysis

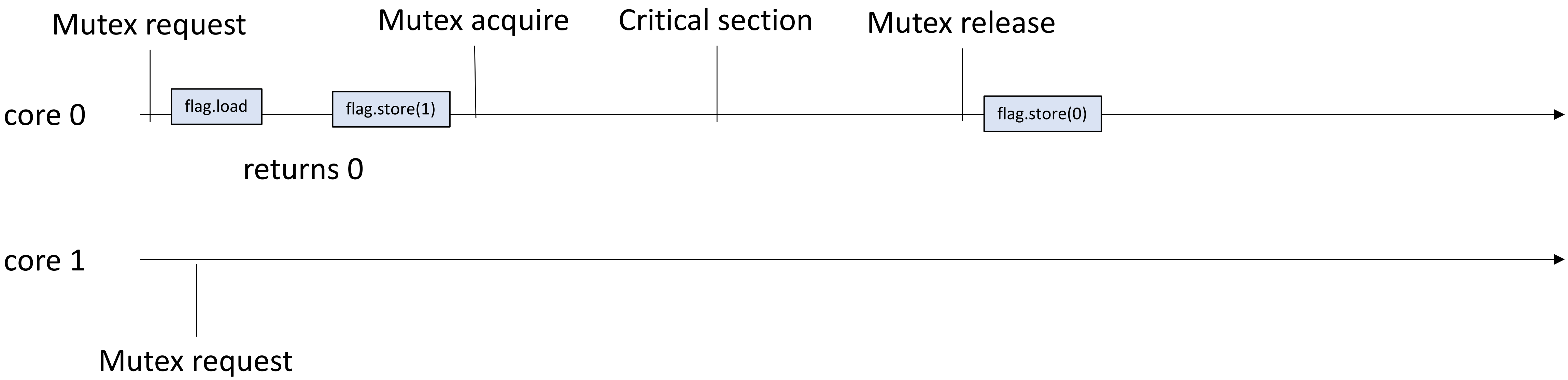
```
void lock() {
  while (flag.load() == 1);
  flag.store(1);
}
```

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

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

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

Enter at almost the same time



Analysis

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

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

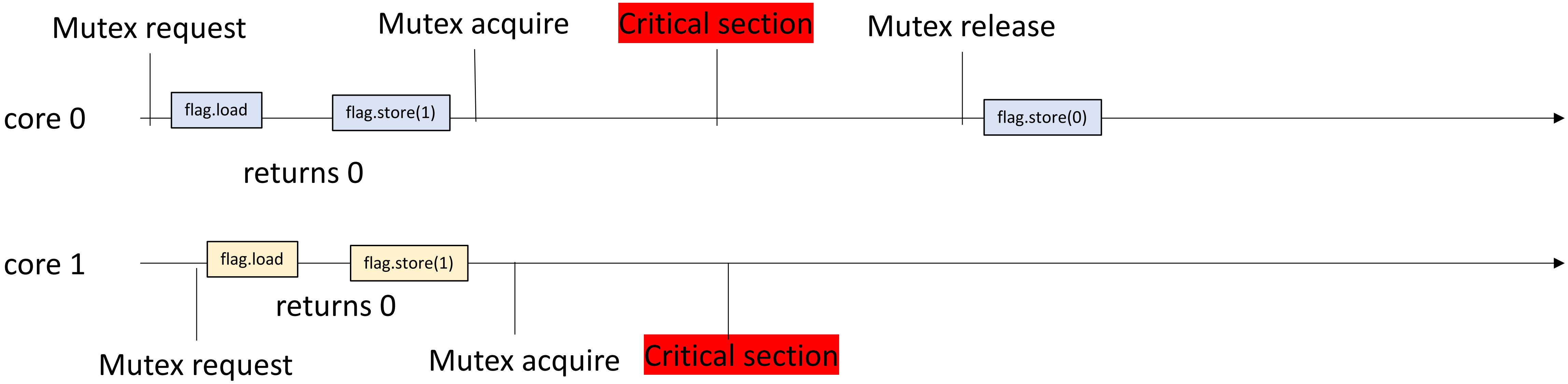
Thread 0:

```
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);
}
```

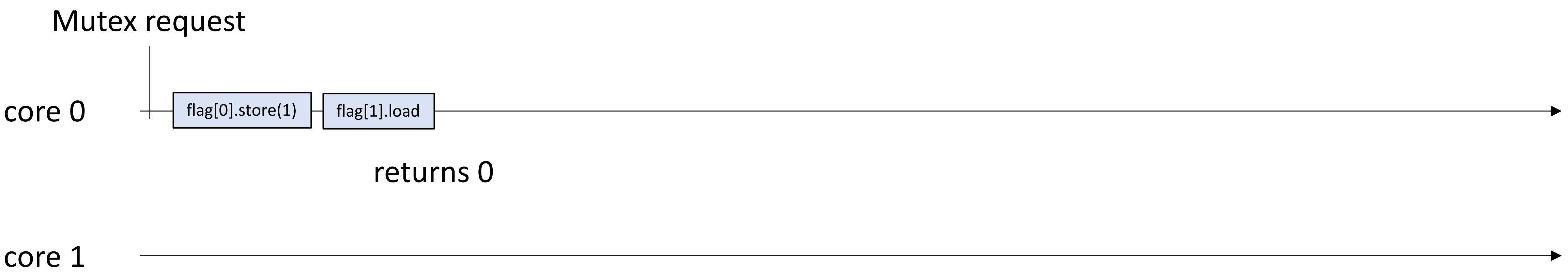
```
void unlock() {
  int i = thread_id;
  flag[i].store(0);
}
```

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);
}
```

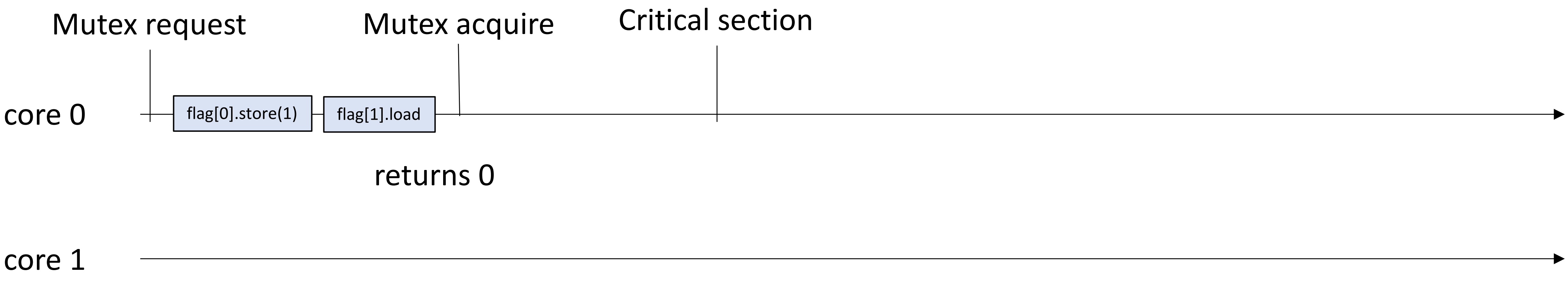
```
void unlock() {
  int i = thread_id;
  flag[i].store(0);
}
```

Thread 0:

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

Thread 1:

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



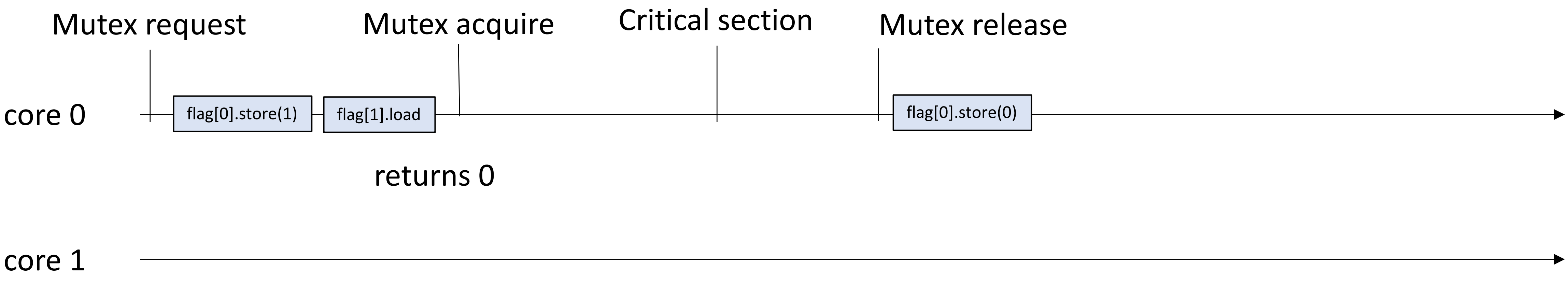
Analysis

```
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);
}
```

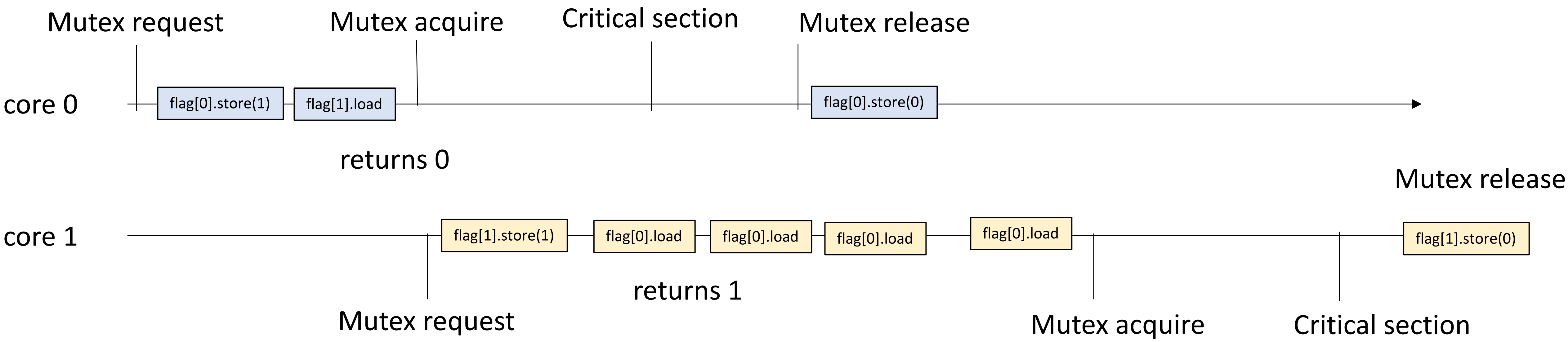
Thread 0:

```
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();
```

Enter at almost the same time

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);
}
```

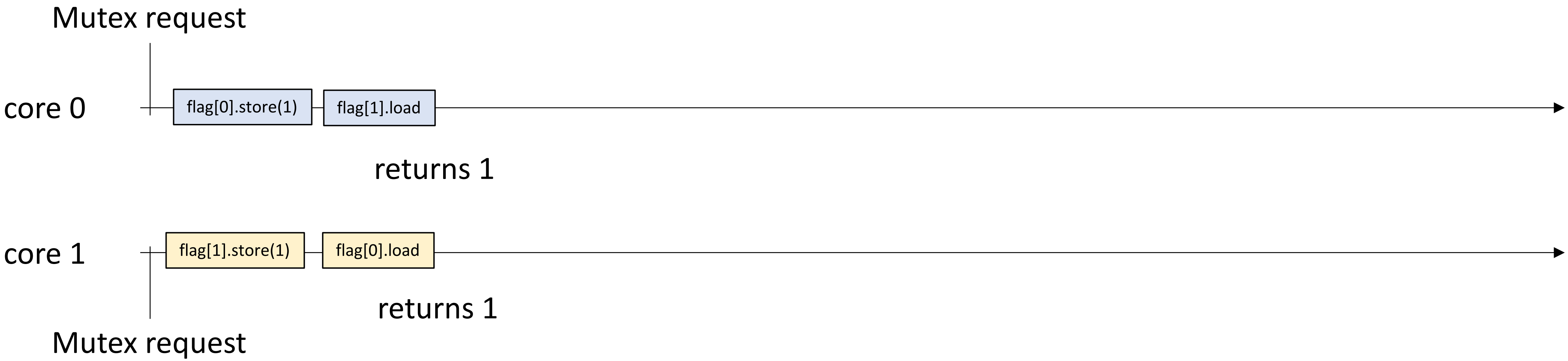
```
void unlock() {
  int i = thread_id;
  flag[i].store(0);
}
```

Thread 0:

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

Thread 1:

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



Analysis

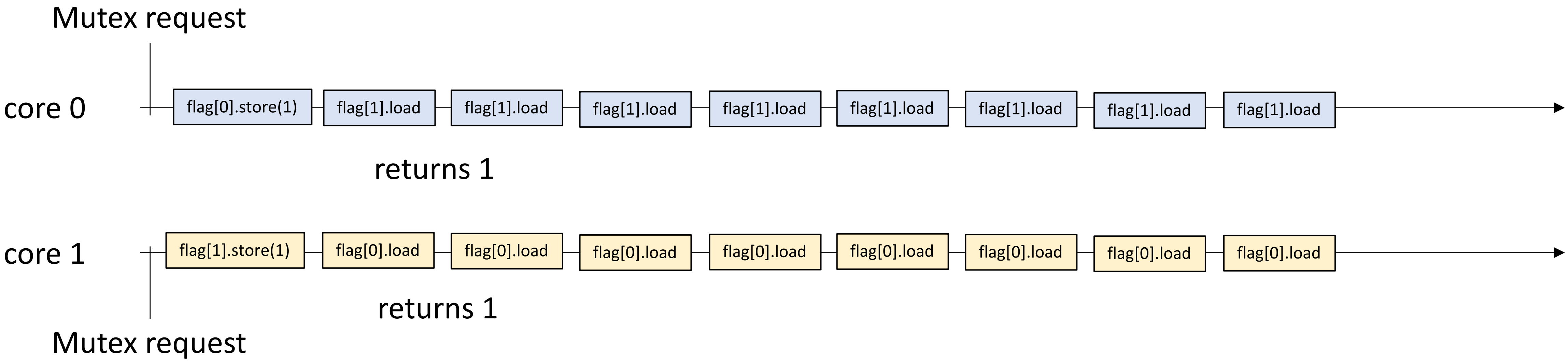
```
void lock() {
  int i = thread_id;
  flag[i].store(1);
  int j = i == 0 ? 1 : 0;
  while (flag[j].load() == 1);
}
```

```
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

concurrent execution

time

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();
```

Mutex request

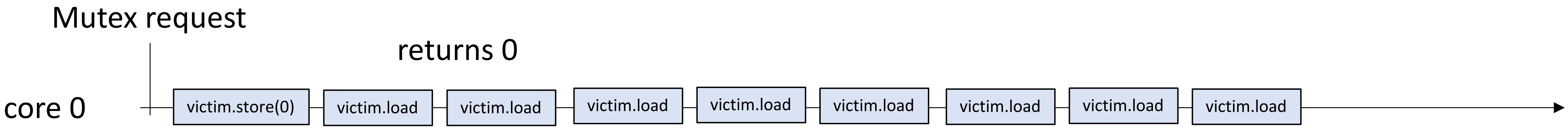
```
void lock() {
  victim.store(thread_id);
  while (victim.load() == thread_id);
}
```

```
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();
```

Enter at almost the same time

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

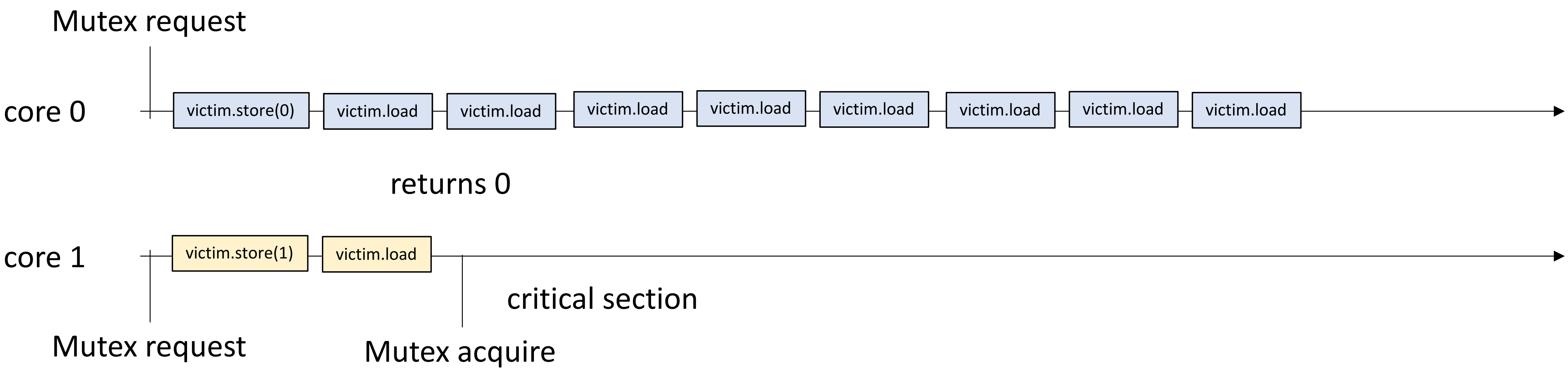
```
void unlock() {}
```

Thread 0:

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

Thread 1:

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



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

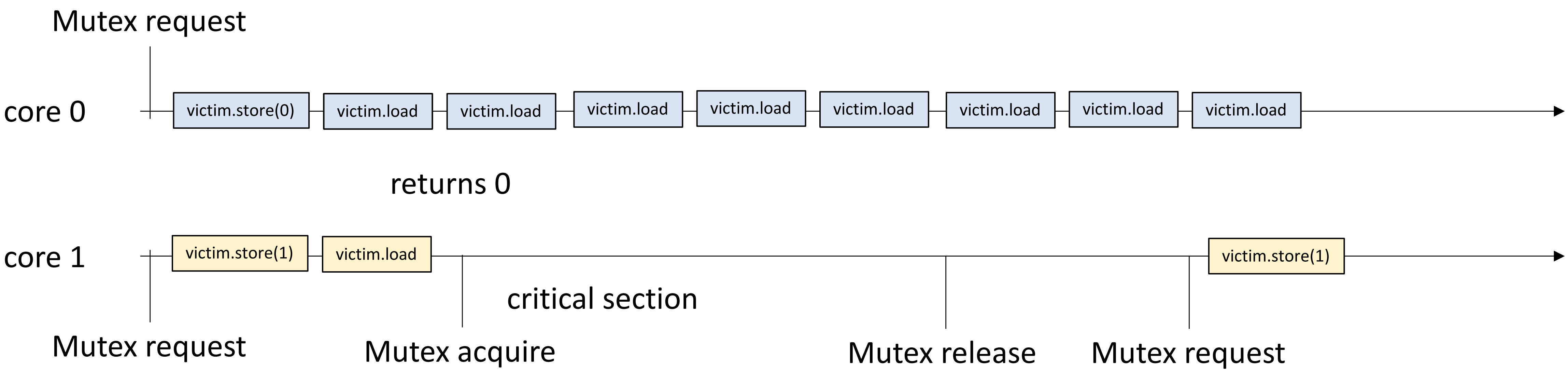
```
void unlock() {}
```

Thread 0:

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

Thread 1:

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



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

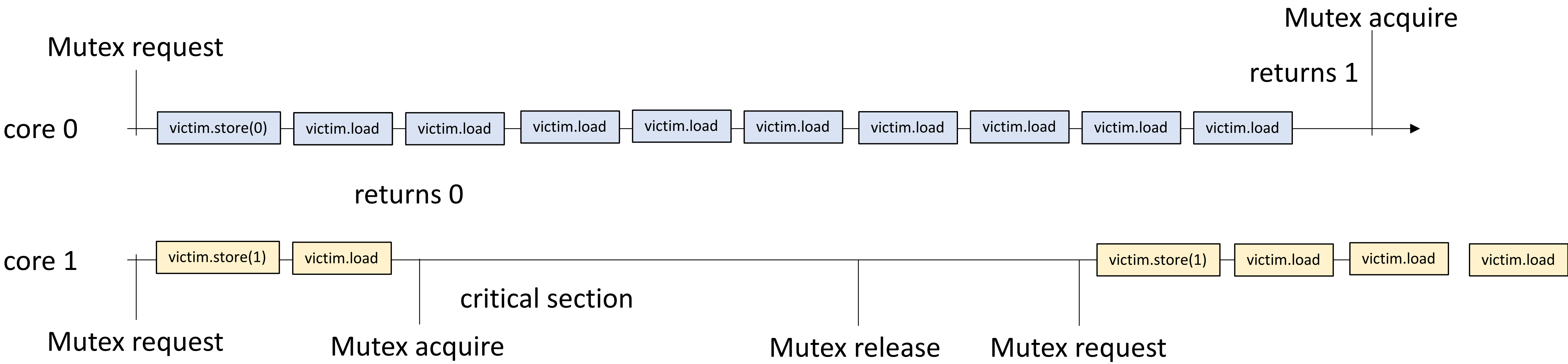
```
void unlock() {}
```

Thread 0:

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

Thread 1:

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



Mutex Implementations

Implementation with flags works when they do not request at the same time

Implementation with victim works when they request at the same time

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 the other thread wants the lock as well,
and I am the victim

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:

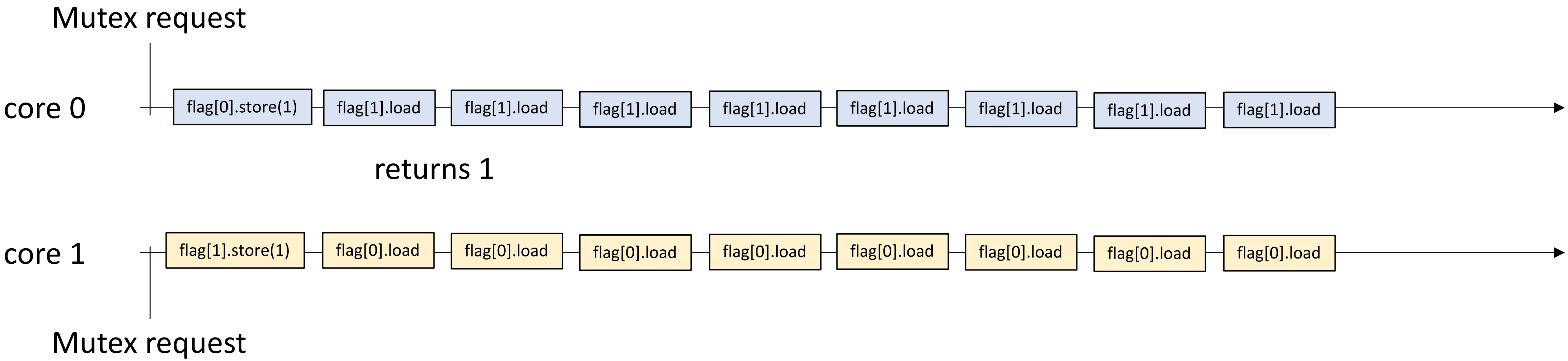
```
m.lock();
m.unlock();
```

Thread 1:

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

How does Peterson 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:

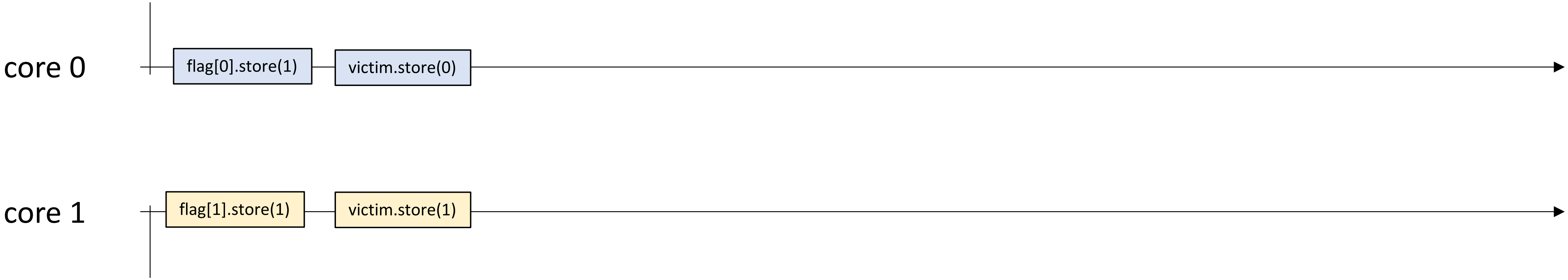
```
m.lock();
m.unlock();
```

Thread 1:

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

Mutex request

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



Mutex request

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);
}
```

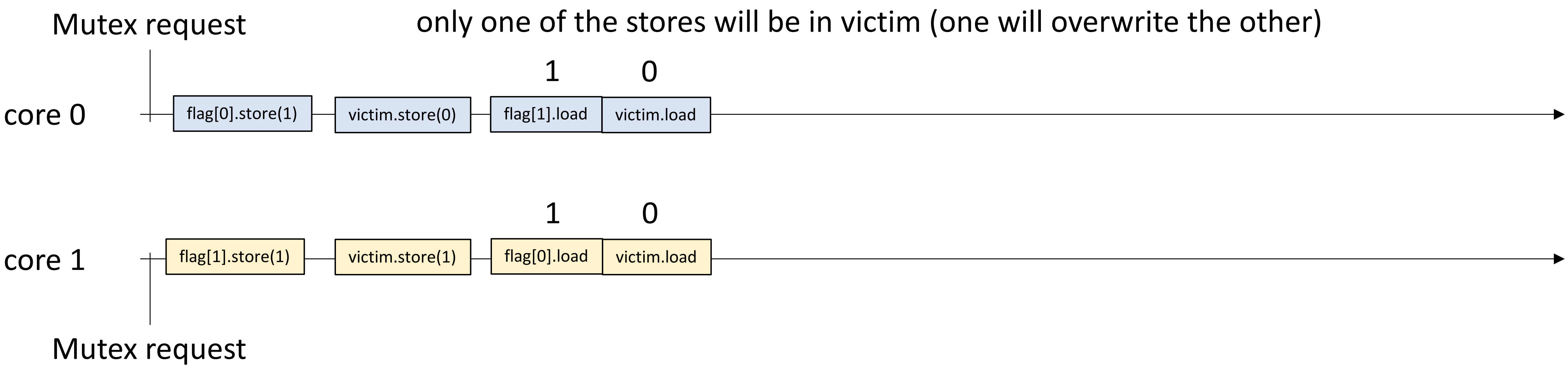
```
void unlock() {
  int i = thread_id;
  flag[i].store(0);
}
```

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);
}
```

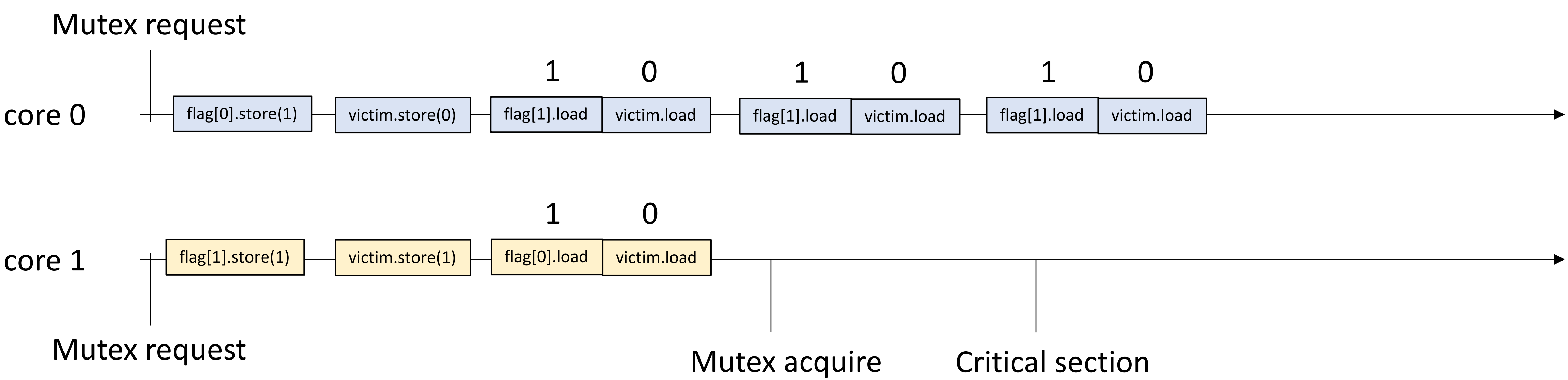
```
void unlock() {
  int i = thread_id;
  flag[i].store(0);
}
```

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);
}
```

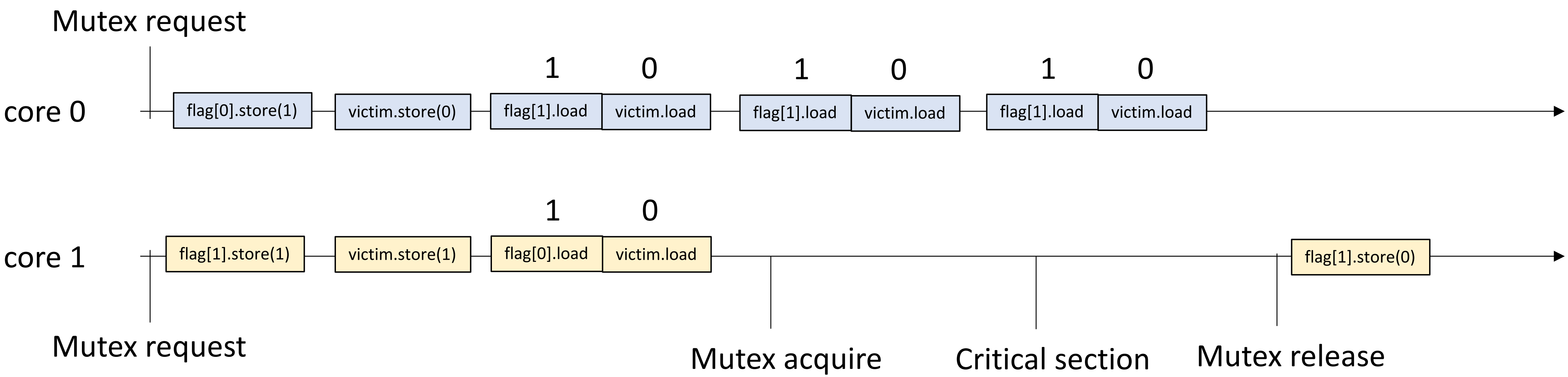
```
void unlock() {
  int i = thread_id;
  flag[i].store(0);
}
```

Thread 0:

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

Thread 1:

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



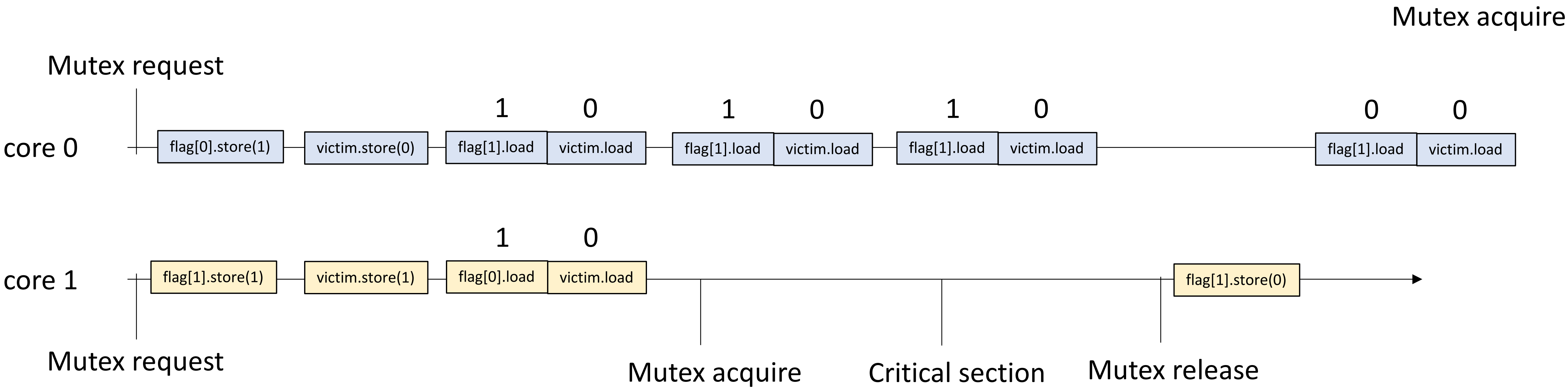
Tie breaking with victim

```
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();
```



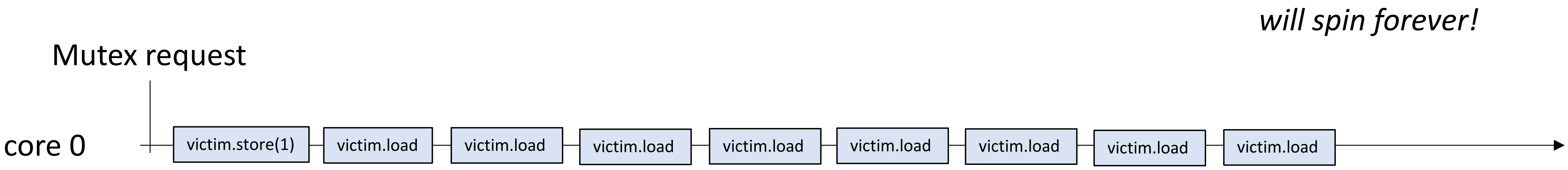
Previous victim issue

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

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

Thread 0:

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



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

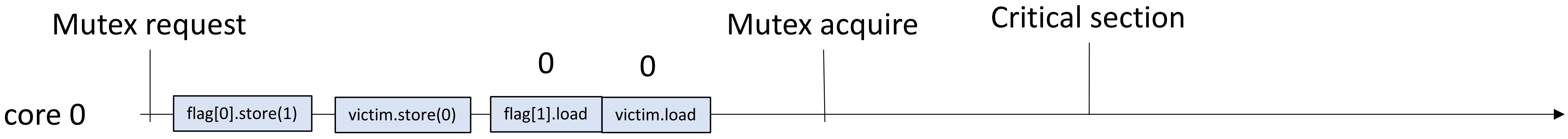
previous flag issue

```
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

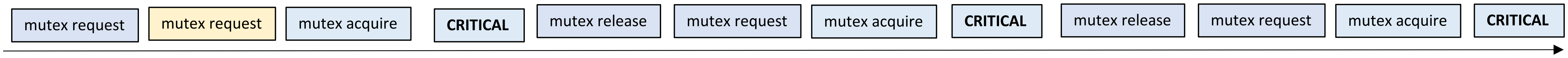
- Mutual exclusion
- Deadlock freedom
- *More formal proof given in the textbook*

What about starvation

recall the starvation property:

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

concurrent execution

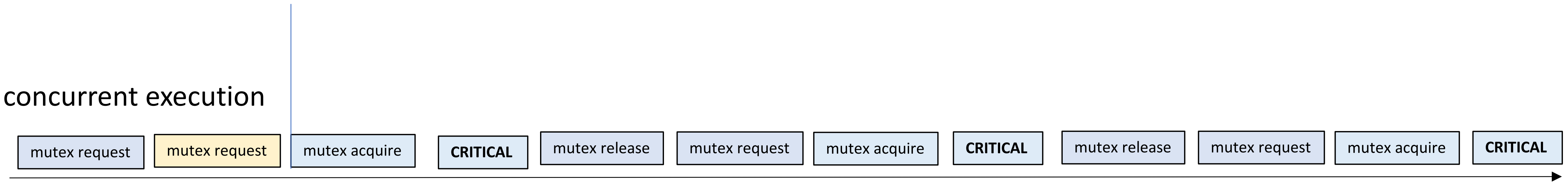


time

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);
}
```

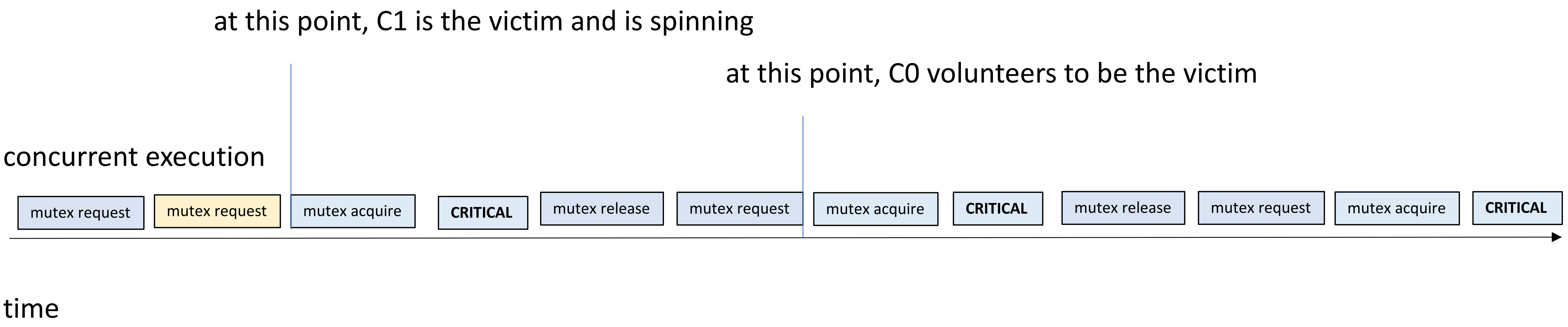
at this point, C1 is the victim and is spinning



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);
}
```



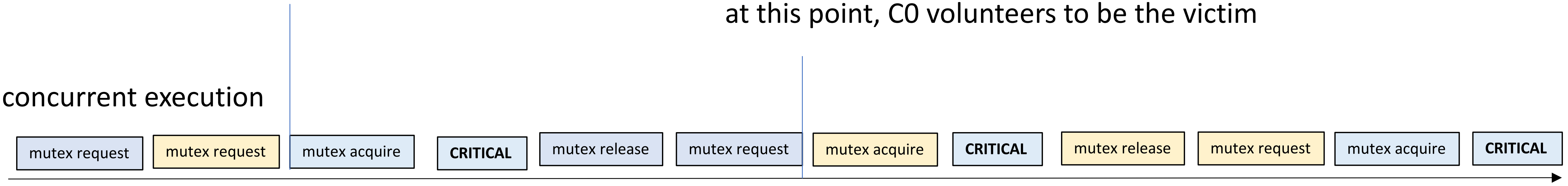
What about starvation

```
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

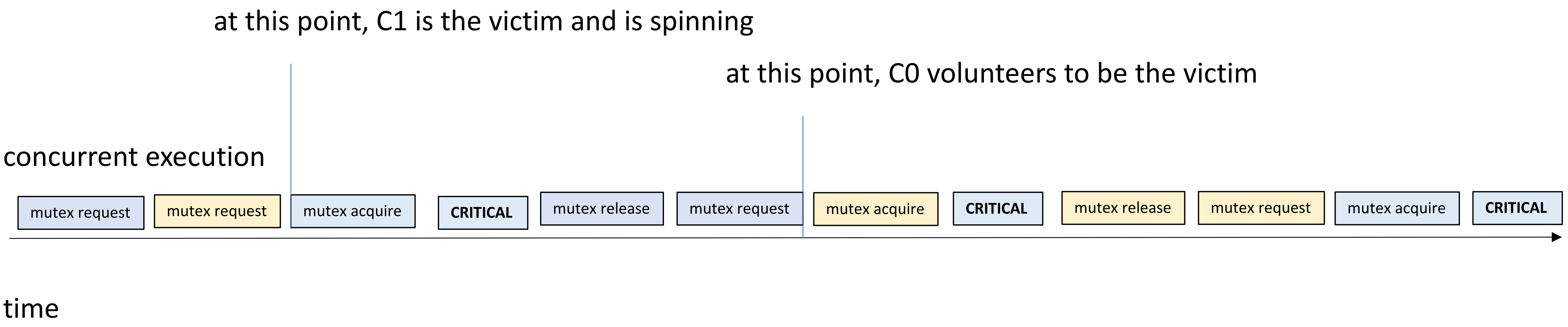


time

What about starvation

Threads take turns in Peterson 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, part 1 of your homework!)

Check implementations

- Thread sanitizer provided in Clang
- Checks for “data races”
  - Generally can help you check if you’ve used mutexes correctly (protecting all shared memory accesses).
  - Also: If you don’t implement your mutexes correctly, you will probably have data races
  - This should hold for your next assignments too
  - Can also check for deadlock based on lock inversion
- Checking tool: if you pass, it doesn’t mean your code is correct

Check implementations

- Why not run all the time with thread sanitizer? Overhead!

Back to Mutex Implementations

Peterson only works with 2 threads.

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

Historical perspective

- These locks are not very performant compared to modern solutions
  - Your HW will show this

- However, they are academically interesting: they can be implemented with plain loads and stores

- We will now turn our attention to more performant implementations that use RMWs

Start by revisiting our first mutex implementation

- 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.
- Let's remember why it was buggy

Buggy Mutex implementation: Analysis

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

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

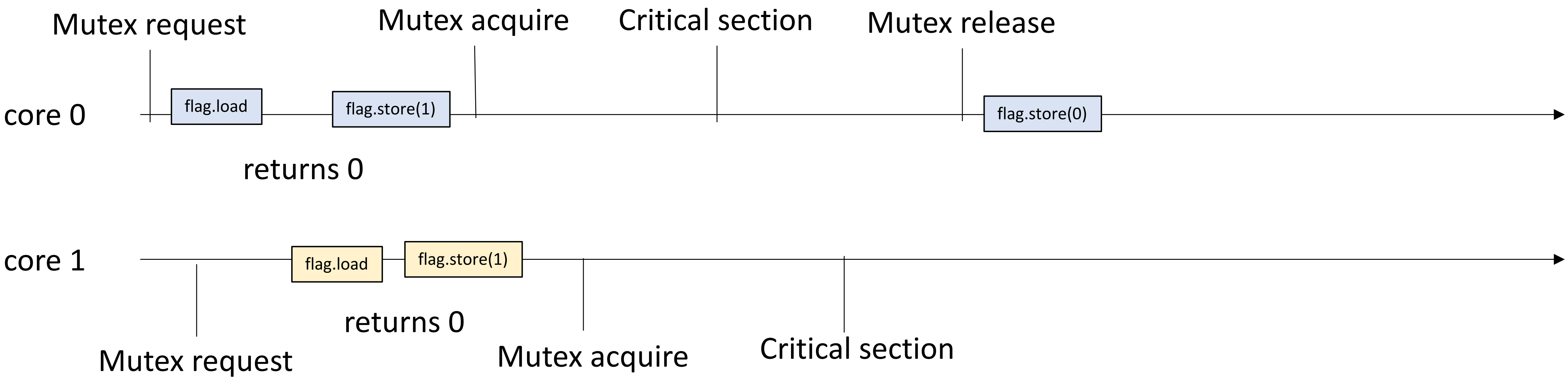
Thread 0:

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

Thread 1:

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

Critical sections overlap! This mutex implementation is not correct!



What went wrong?

- The load and stores from two threads interleaved
  - What if there was a way to prevent this?

What went wrong?

- The load and stores from two threads interleaved
  - What if there was a way to prevent this?

- Atomic RMWs
  - operate on atomic types (we already have atomic types)
  - recall the non-locking bank accounts:
    `atomic_fetch_add(`**`atomic`**` *a, `**`value`**` v);`

What is a RMW

A read-modify-write consists of:

- ***read***
- ***modify***
- ***write***

done atomically, i.e. they cannot interleave.

Typically returns the value (in some way) from the read.

atomic_fetch_add

Recall the lock free account

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
}
```

atomic_fetch_add

Recall the lock free account

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

```
int atomic_fetch_add(atomic_int * addr, int value) {
   int stash = *addr; // read
   int new_value = value + stash;    // modify
   *addr = new_value;     // write
   return stash;      // return previous value in the memory location
}
```

lock-free accounts

Tyler's coffee addiction:

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

Tyler's employer

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

time

time

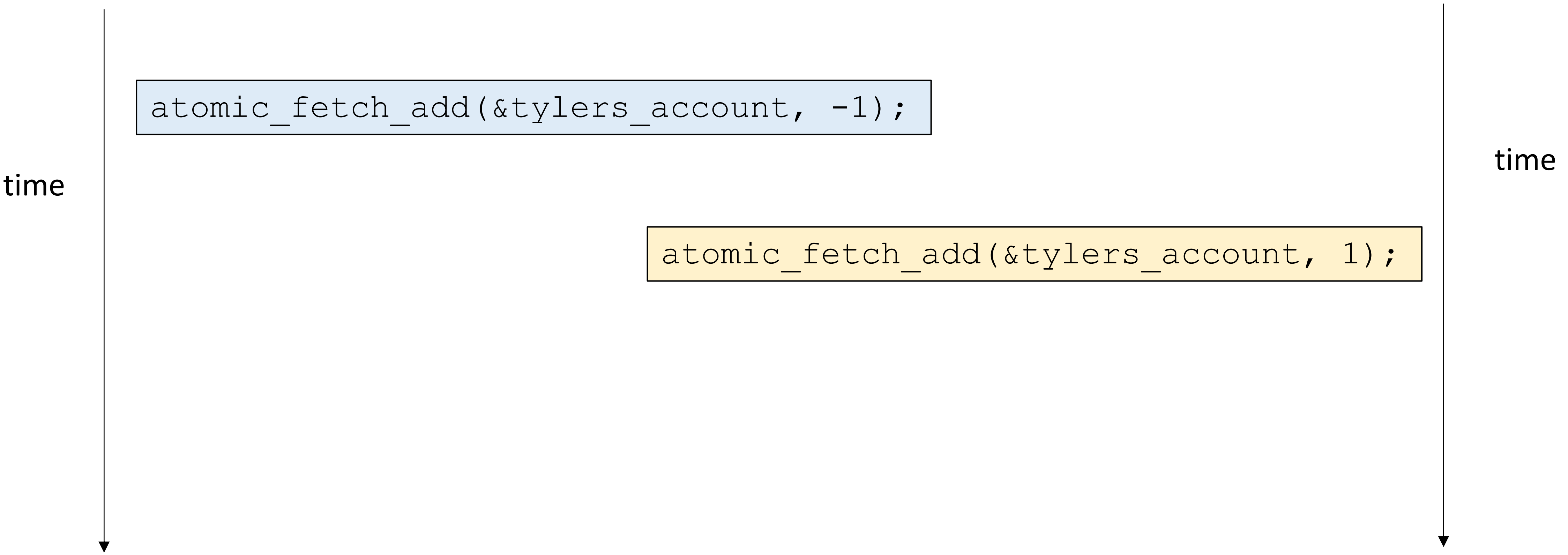
lock-free accounts

Tyler's coffee addiction:

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

Tyler's employer

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



lock-free accounts

Tyler's coffee addiction:

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

Tyler's employer

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

time

```
tmp = tylers_account.load();
tmp -= 1;
tylers_account.store(tmp);
```

time

```
tmp = tylers_account.load();
tmp += 1;
tylers_account.store(tmp);
```

lock-free accounts

Tyler's coffee addiction:

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

Tyler's employer

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

time

```
tmp = tylers_account.load();
tmp -= 1;
tylers_account.store(tmp);
```

cannot interleave!

time

```
tmp = tylers_account.load();
tmp += 1;
tylers_account.store(tmp);
```

lock-free accounts

Tyler's coffee addiction:

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

Tyler's employer

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

time

cannot interleave!

```
tmp = tylers_account.load();
tmp += 1;
tylers_account.store(tmp);
```

time

```
tmp = tylers_account.load();
tmp -= 1;
tylers_account.store(tmp);
```

either way, account breaks even at the end!

RMW-based locks

- A few simple RMWs enable lots of interesting mutex implementations

First example: Exchange Lock

- Simplest atomic RMW will allow us to implement an:
- N-threaded mutex with 1 bit!

First example: Exchange Lock

```
value atomic_exchange(atomic *a, value v);
```

Loads the value at `a` and stores the value in `v` at `a`. Returns the value that was loaded.

First example: Exchange Lock

```
value atomic_exchange(atomic *a, value v);
```

Loads the value at `a` and stores the value in `v` at `a`. Returns the value that was loaded.

```
value atomic_exchange(atomic *a, value v) {
  value tmp = a.load();
  a.store(v);
  return tmp;
}
```

First example: Exchange Lock

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

class Mutex {
public:
  Mutex() {
    flag = false;
  }

  void lock();
  void unlock();

private:
  atomic_bool flag;
};
```

Lets make a mutex with just one atomic bool!

First example: Exchange Lock

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

class Mutex {
public:
  Mutex() {
    flag = false;
  }

  void lock();
  void unlock();

private:
  atomic_bool flag;
};
```

Lets make a mutex with just one atomic bool!

initialized to `false`

one atomic flag

First example: Exchange Lock

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

class Mutex {
public:
  Mutex() {
    flag = false;
  }

  void lock();
  void unlock();

private:
  atomic_bool flag;
};
```

Lets make a mutex with just one atomic bool!

initialized to `false`

one atomic flag

main idea:

The flag is false when the mutex is free.

The flag is true when some thread has the mutex.

First example: Exchange Lock

```
void lock() {
  while (atomic_exchange(&flag, true) == true);
}
```

First example: Exchange Lock

```
void lock() {
  while (atomic_exchange(&flag, true) == true);
}
```

So what's going on?

First example: Exchange Lock

```
void lock() {
  while (atomic_exchange(&flag, true) == true);
}
```

So what's going on?

Two cases:

mutex is free: the value loaded is false. We store true. The value returned is false, so we don't spin

mutex is taken: the value loaded is true, we put the SAME value back (true). The returned value is true, so we spin.

First example: Exchange Lock

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

Unlock is simple: just store false to the flag, marking the mutex as available.

Analysis

```
void lock() {
  while (atomic_exchange(&flag, true) == true);
}
```

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

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

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

core 0

core 1

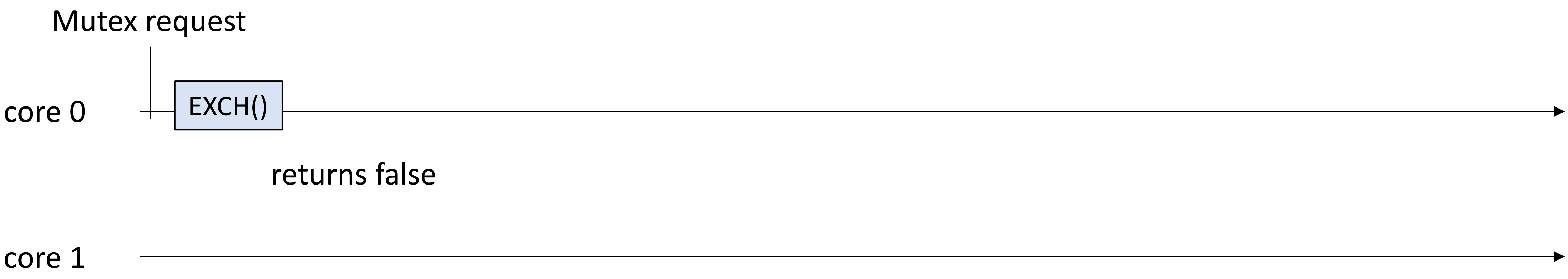
Analysis

```
void lock() {
  while (atomic_exchange(&flag, true) == true);
}
```

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

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

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



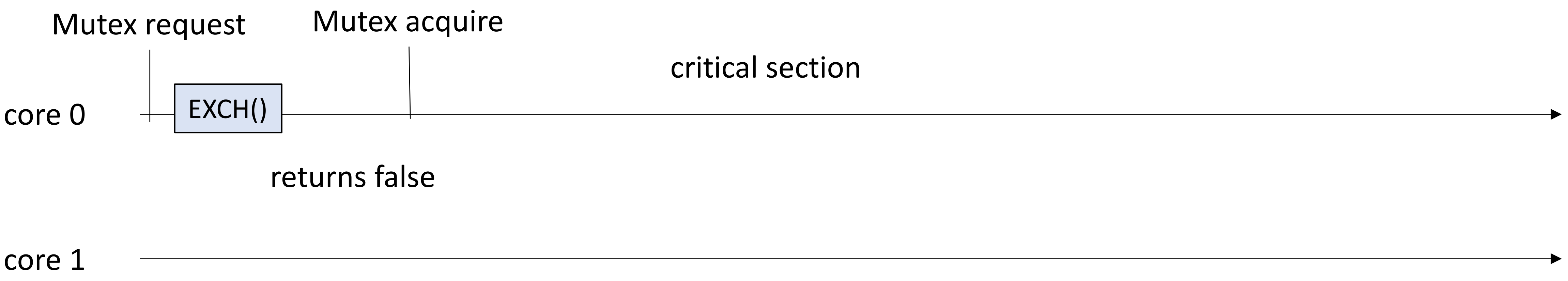
Analysis

```
void lock() {
  while (atomic_exchange(&flag, true) == true);
}
```

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

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

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



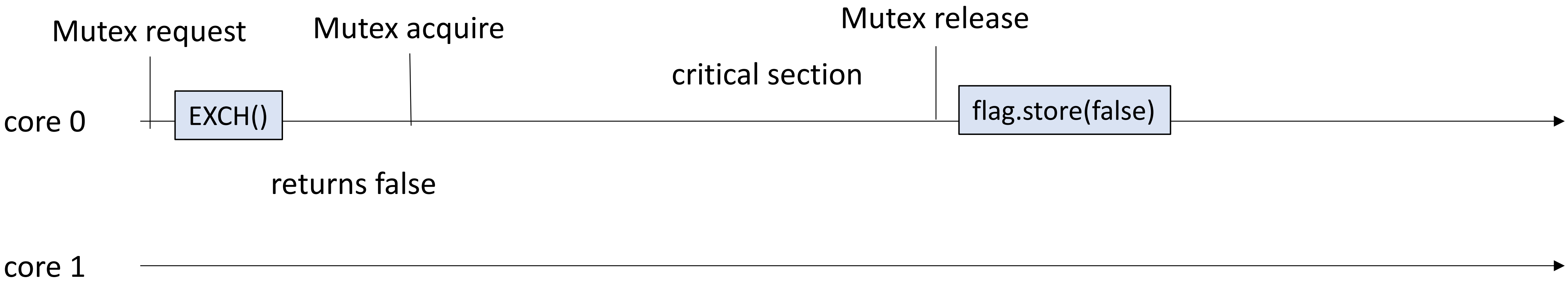
Analysis

```
void lock() {
  while (atomic_exchange(&flag, true) == true);
}
```

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

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

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



Analysis

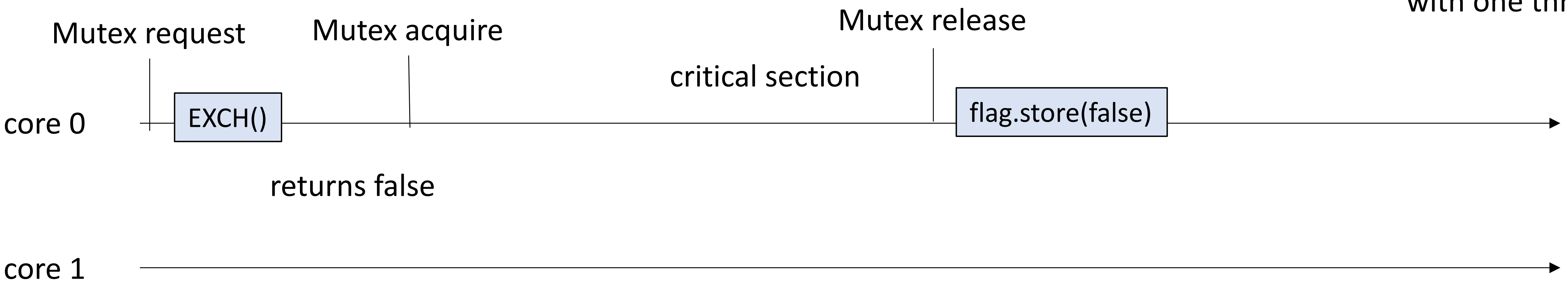
```
void lock() {
  while (atomic_exchange(&flag, true) == true);
}
```

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

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

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

mutex works
with one thread



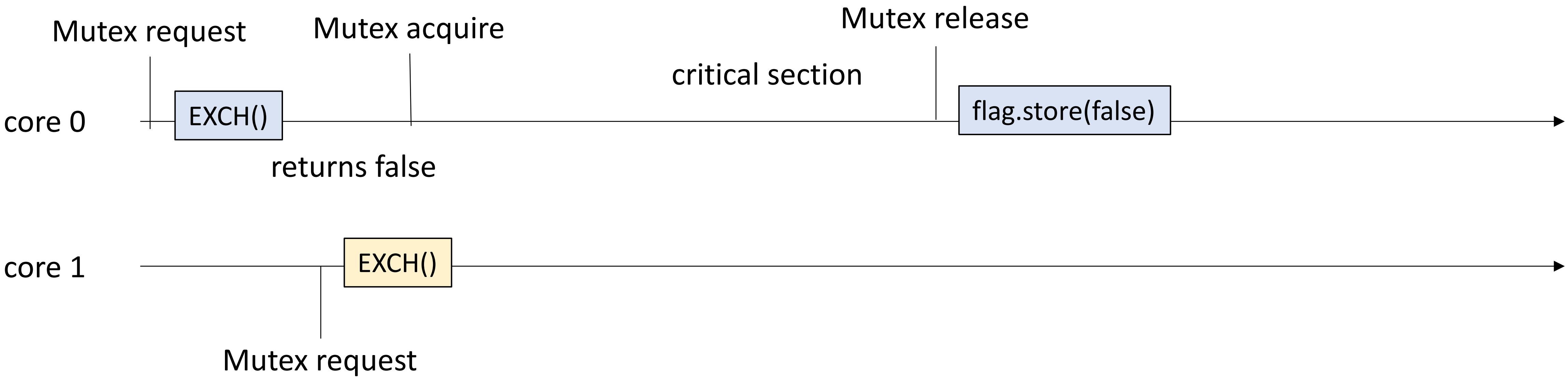
Analysis

```
void lock() {
  while (atomic_exchange(&flag, true) == true);
}
```

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

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

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



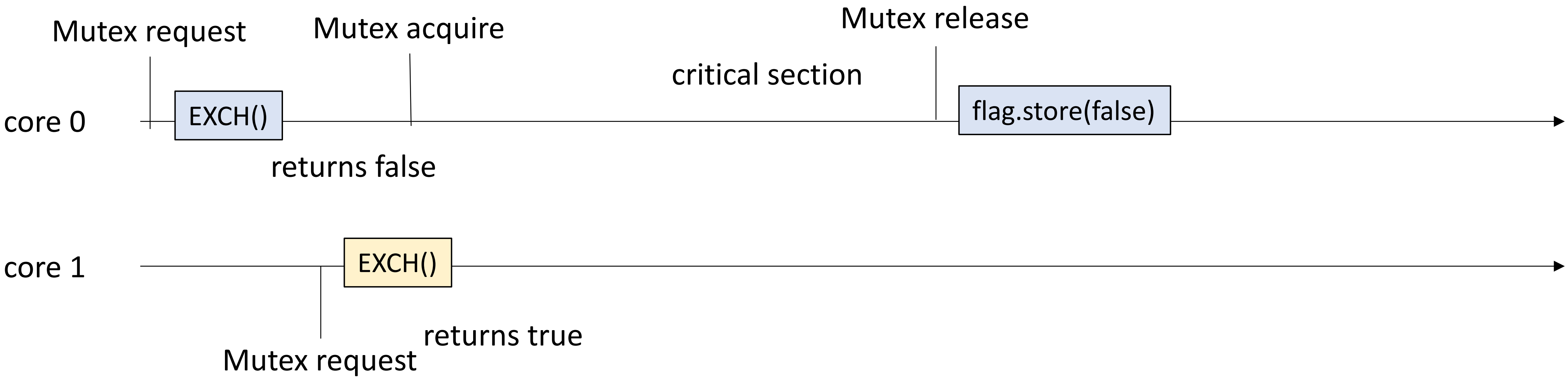
Analysis

```
void lock() {
  while (atomic_exchange(&flag, true) == true);
}
```

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

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

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



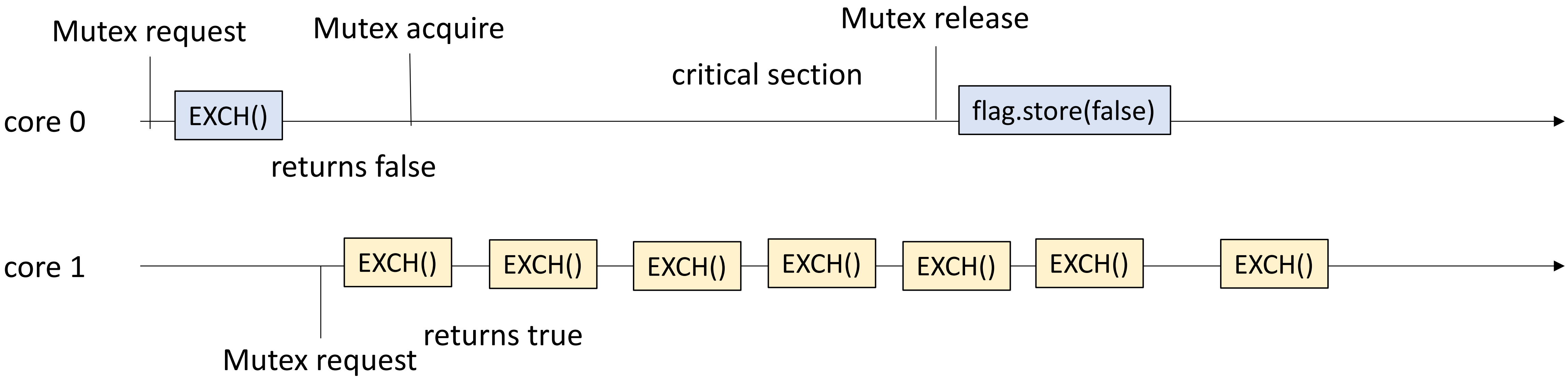
Analysis

```
void lock() {
  while (atomic_exchange(&flag, true) == true);
}
```

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

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

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



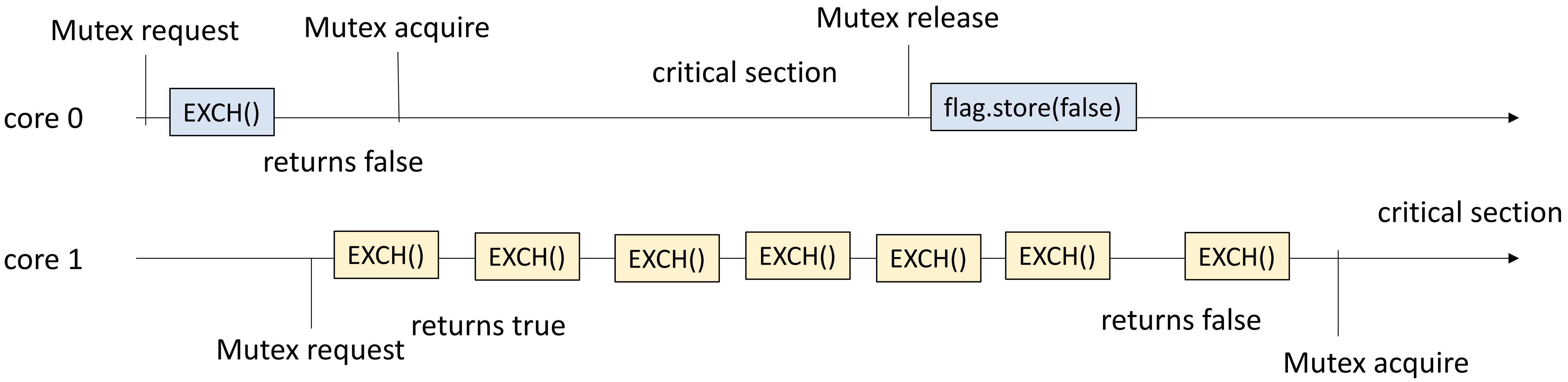
Analysis

```
void lock() {
  while (atomic_exchange(&flag, true) == true);
}
```

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

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

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



Analysis

```
void lock() {
  while (atomic_exchange(&flag, true) == true);
}
```

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

Thread 0:

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

Thread 1:

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

what about interleavings?