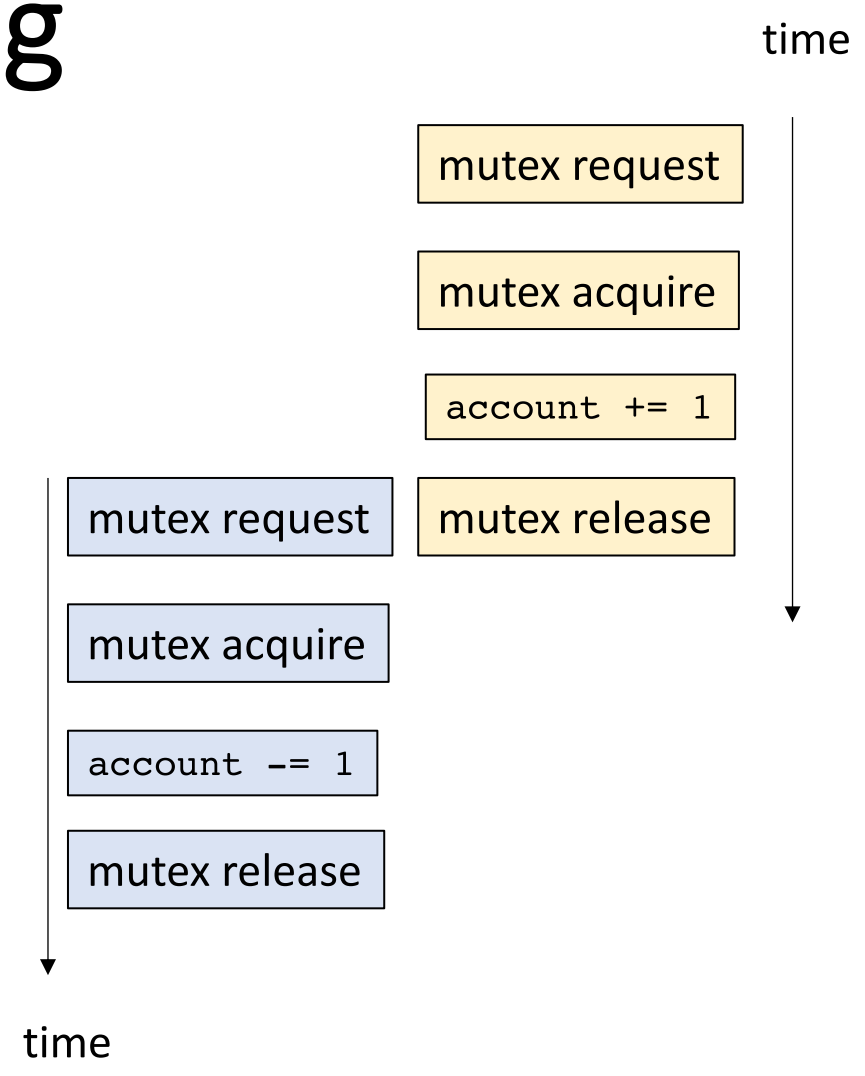
CSE113: Parallel Programming

Jan. 24, 2022

- **Topics**:
  - Mutex implementations



Announcements

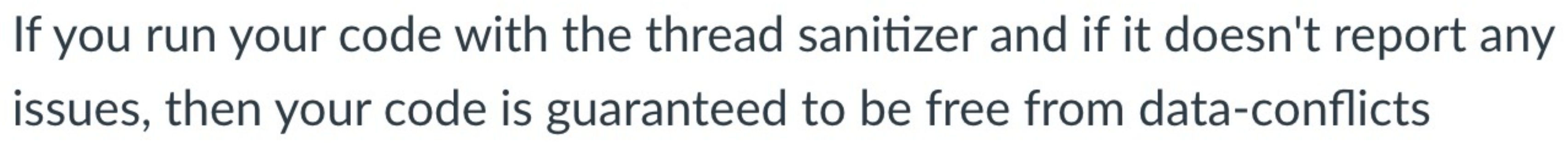
- Homework 1 was due on Friday
  - I hope things went smoothly (they seemed to on my end)

- Homework 2 was released on Friday
  - Part 1 is possible to get started on
  - Part 2 and 3 will be possible after Wednesday (or Friday)

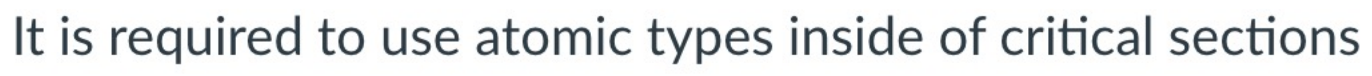
Today’s Quiz

- Please do it!
- Due by midnight on tomorrow

Previous quiz

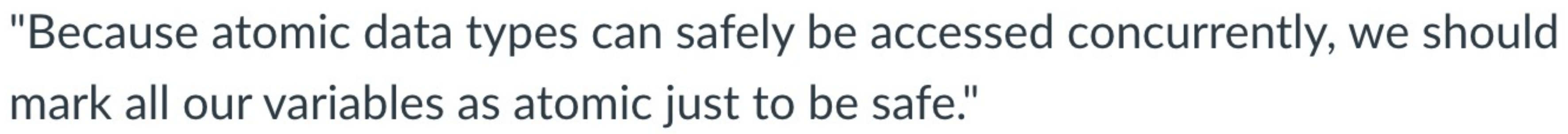

If you run your code with the thread sanitizer and if it doesn't report any issues, then your code is guaranteed to be free from data-conflicts

Previous quiz



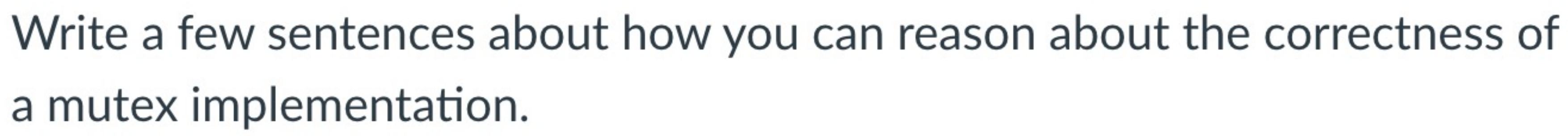
It is required to use atomic types inside of critical sections

Previous quiz



"Because atomic data types can safely be accessed concurrently, we should mark all our variables as atomic just to be safe."

Previous quiz



Write a few sentences about how you can reason about the correctness of a mutex implementation.

Review

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.

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

concurrent execution
mutex request
mutex request
time

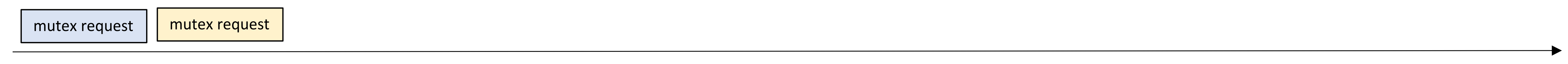
Properties of mutexes

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



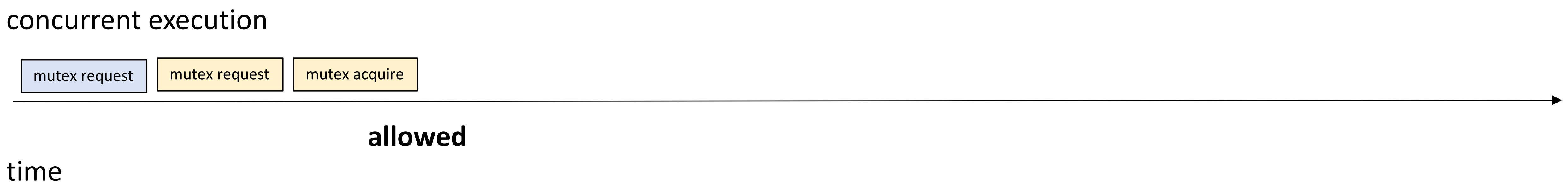
time

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

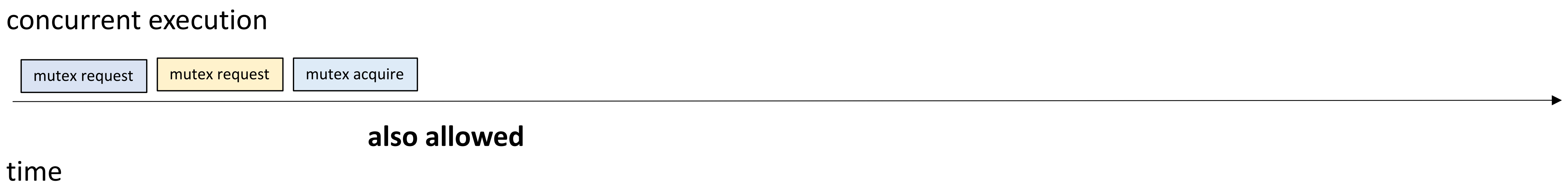


Properties of mutexes

Three properties

- **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

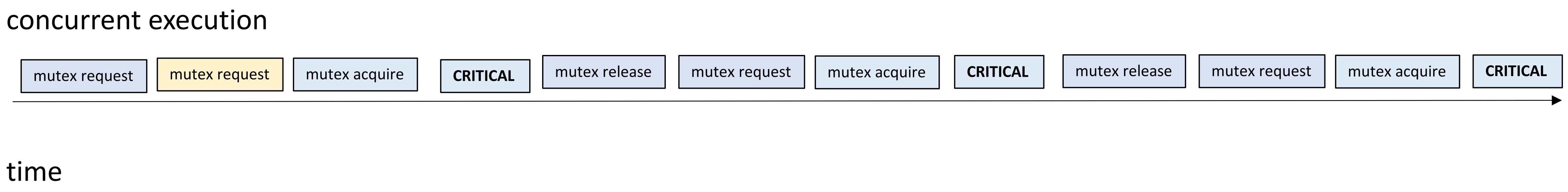


Properties of mutexes

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

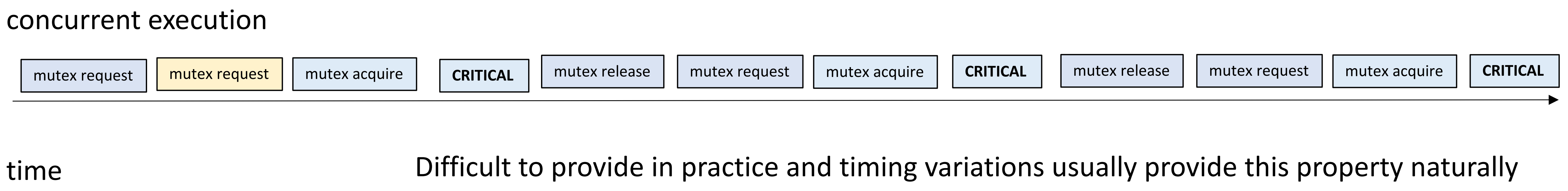


Properties of mutexes

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



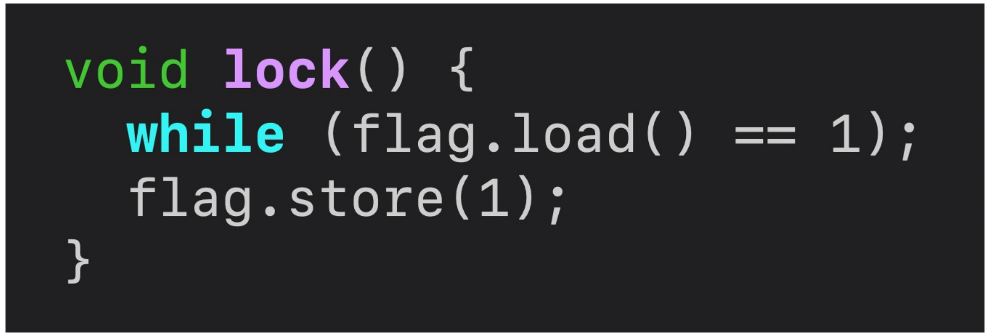
Difficult to provide in practice and timing variations usually provide this property naturally

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.

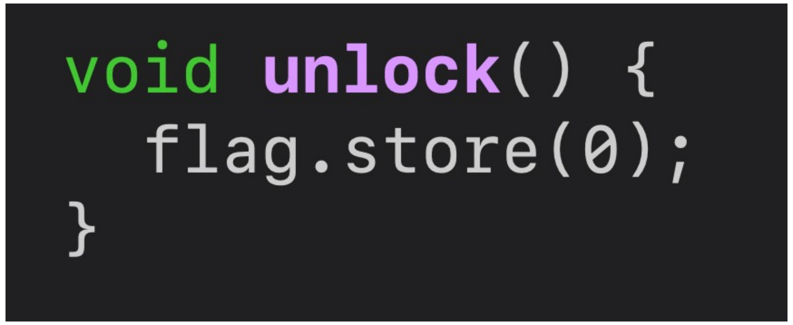
Mutex Implementations



```
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



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

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

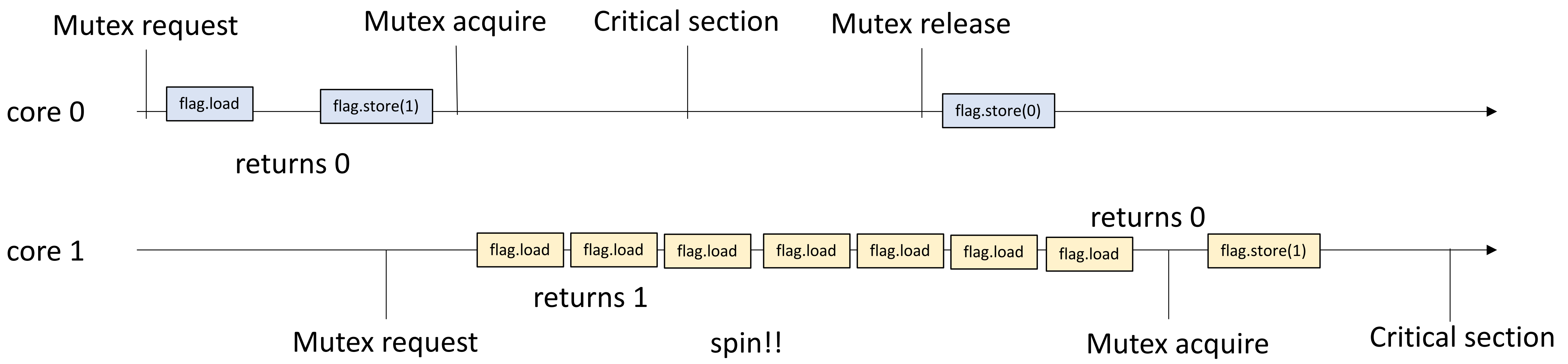
Analysis

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

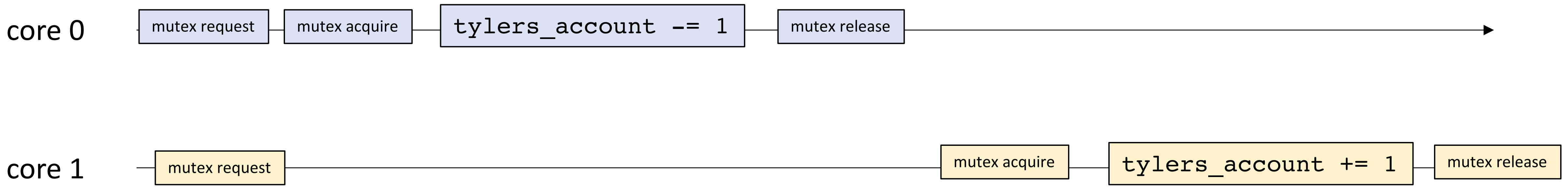
```
void unlock() {
  flag.store(0);
}
```

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

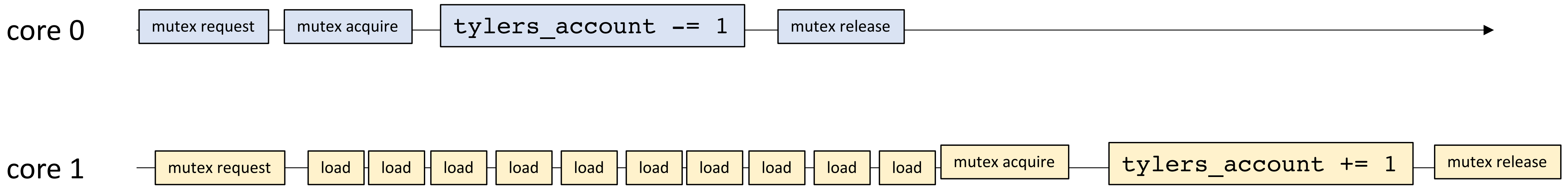
Thread 1:
```
m.lock();
m.unlock();
```



Quick aside



What is actually happening here?


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

However, this mutex attempt is buggy

Analysis

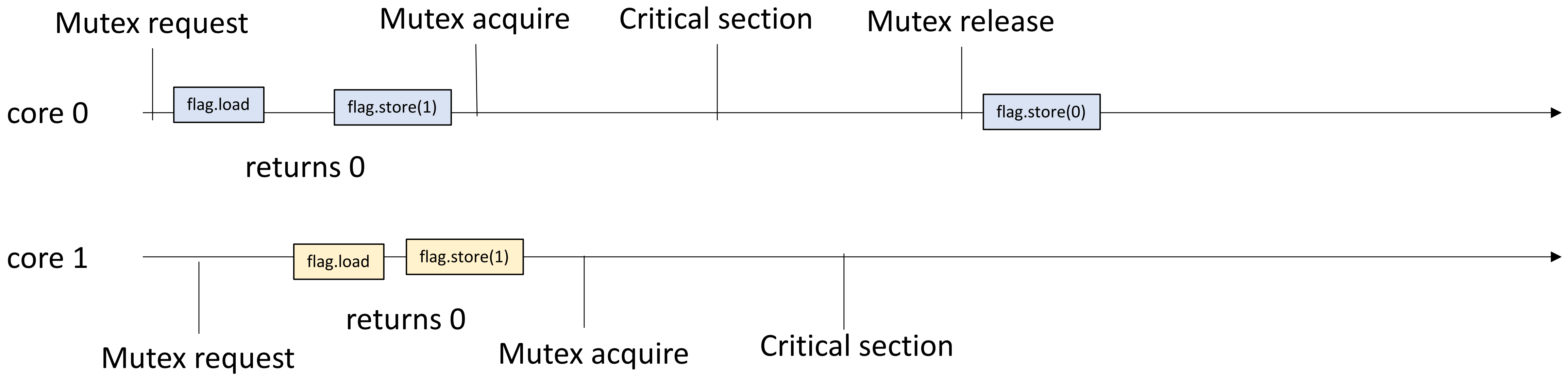
```
void lock() {
  while (flag.load() == 1);
  flag.store(1);
}
```

```
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!



To the lecture!

- Continuing mutex implementations

- RMW mutex implementations

Mutex Implementations

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

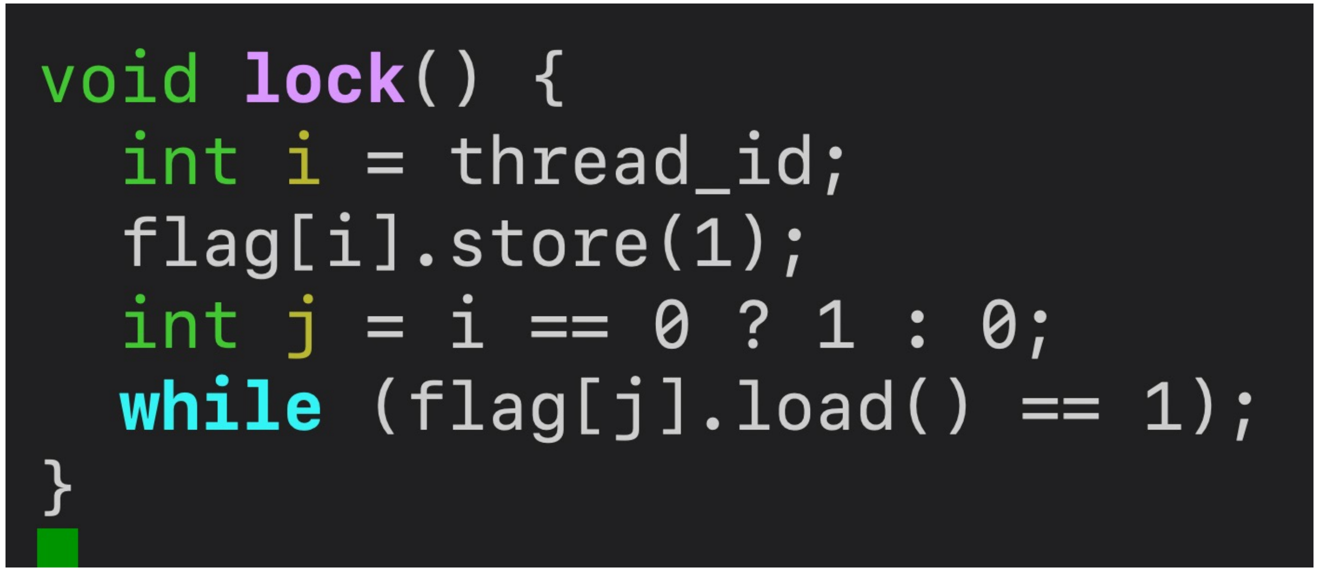
Mutex Implementations

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

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

  void lock();
  void unlock();

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

both initialized to 0

two flags this time

Mutex Implementations



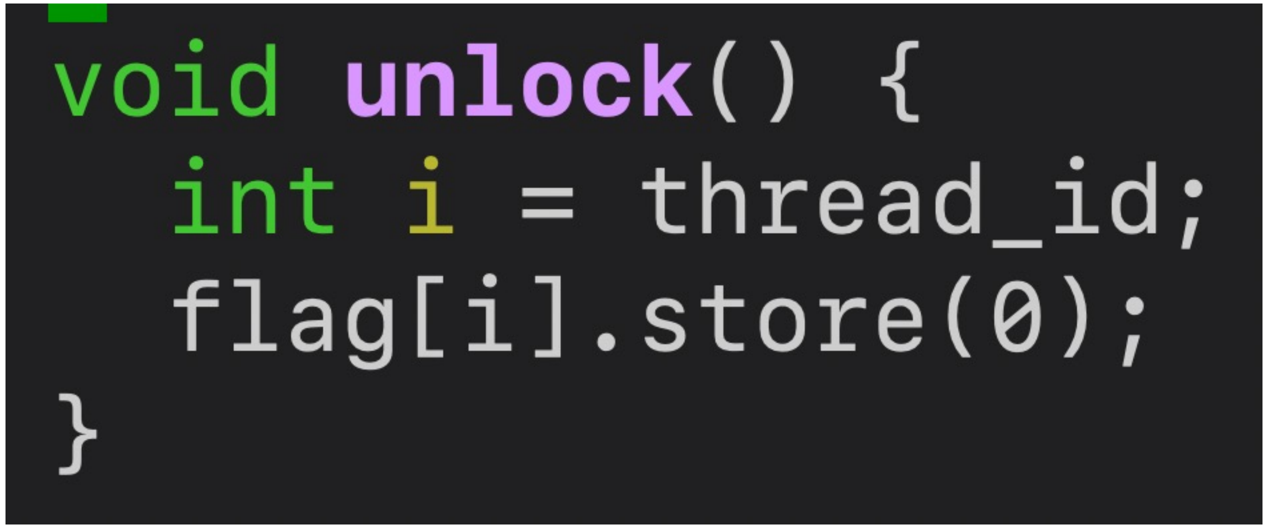
```
void lock() {
  int i = thread_id;
  flag[i].store(1);
  int j = i == 0 ? 1 : 0;
  while (flag[j].load() == 1);
}
```

Thread id (0, or 1)

Mark your intention to take the lock

Wait for other thread to leave the critical section

Mutex Implementations

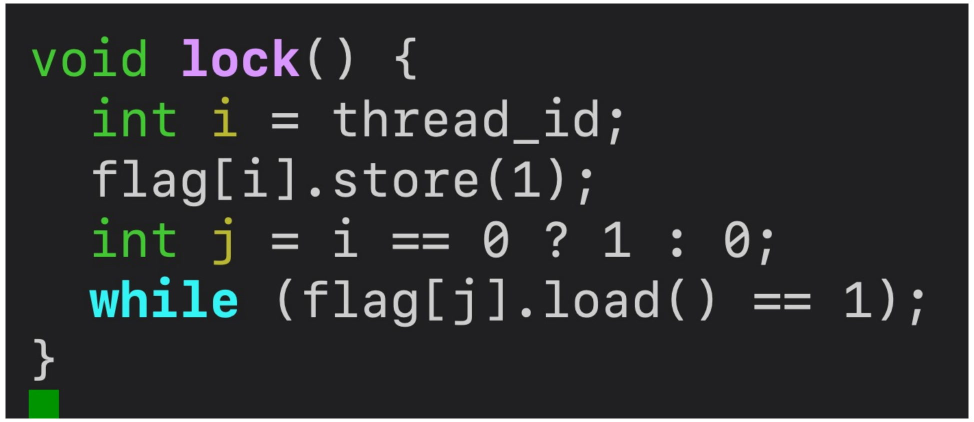


```
void unlock() {
  int i = thread_id;
  flag[i].store(0);
}
```

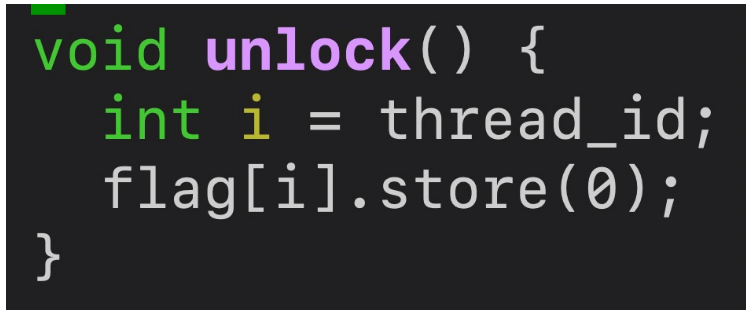
Thread id (0, or 1)

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

Analysis



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



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

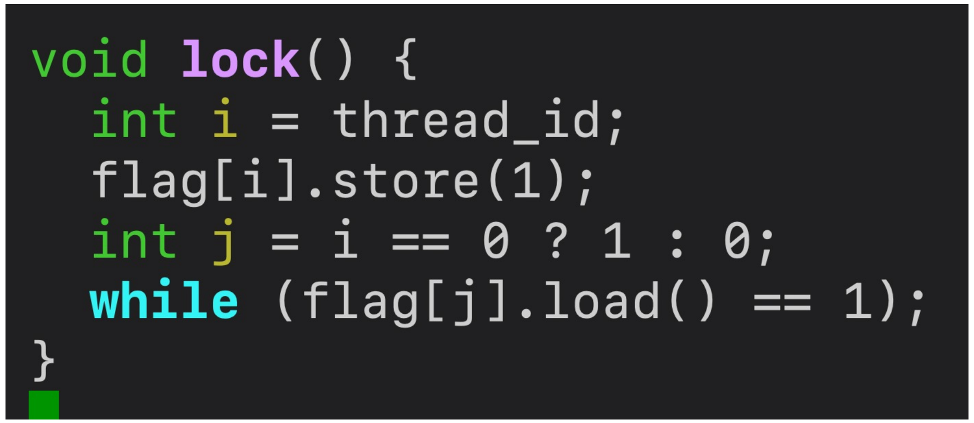
Thread 0:

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

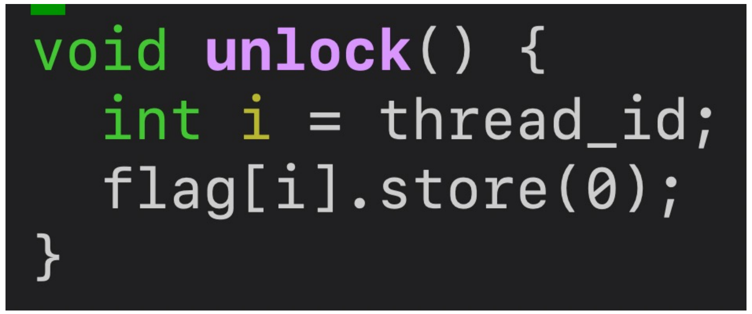
Thread 1:

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

Analysis



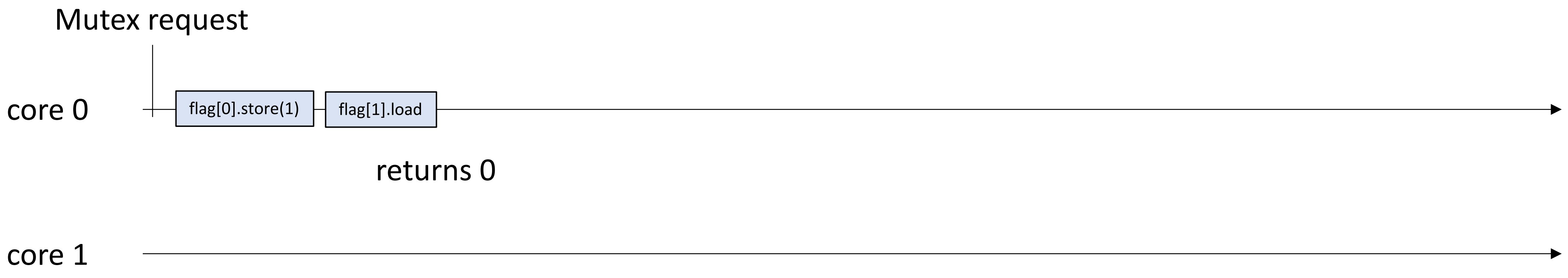
```
void lock() {
  int i = thread_id;
  flag[i].store(1);
  int j = i == 0 ? 1 : 0;
  while (flag[j].load() == 1);
}
```



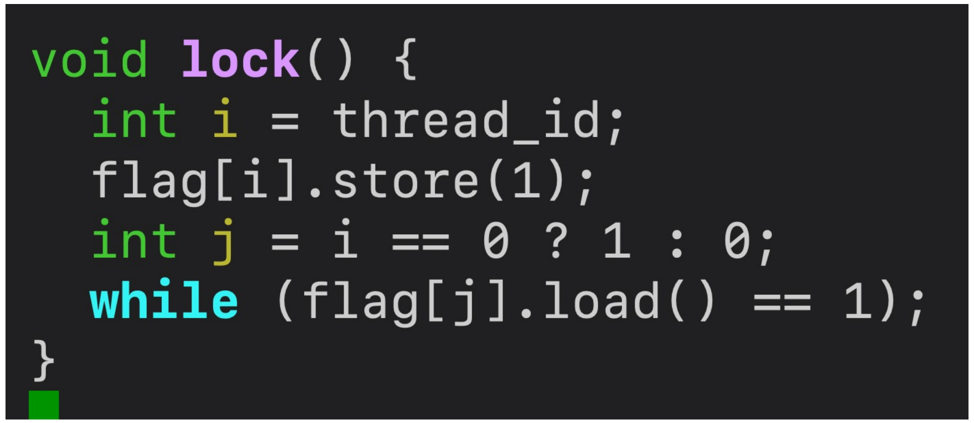
```
void unlock() {
  int i = thread_id;
  flag[i].store(0);
}
```

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

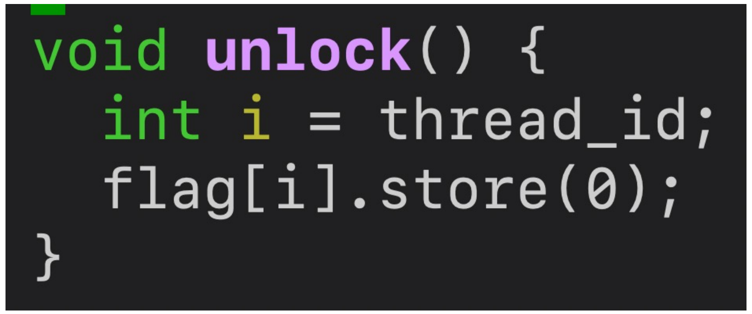
Thread 1:
```
m.lock();
m.unlock();
```



Analysis



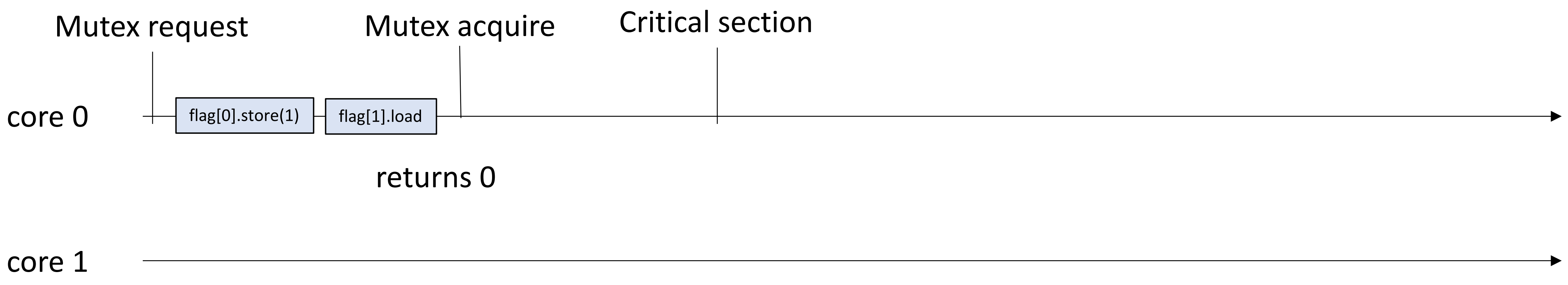
```
void lock() {
  int i = thread_id;
  flag[i].store(1);
  int j = i == 0 ? 1 : 0;
  while (flag[j].load() == 1);
}
```



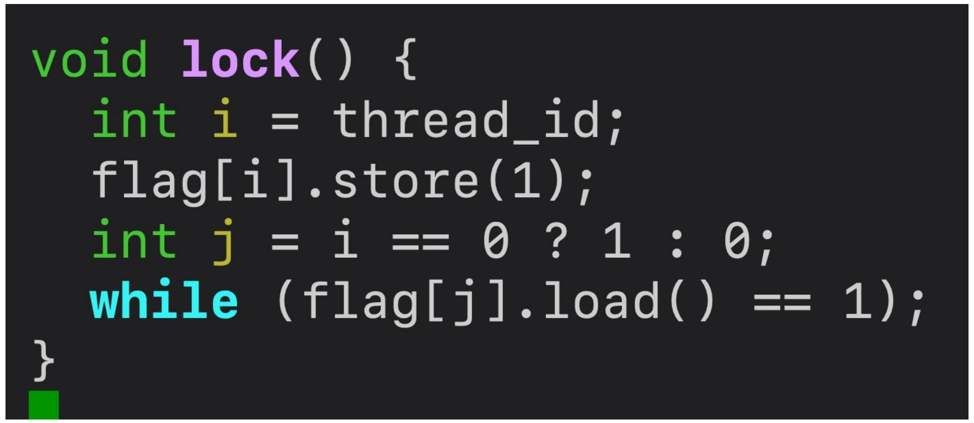
```
void unlock() {
  int i = thread_id;
  flag[i].store(0);
}
```

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

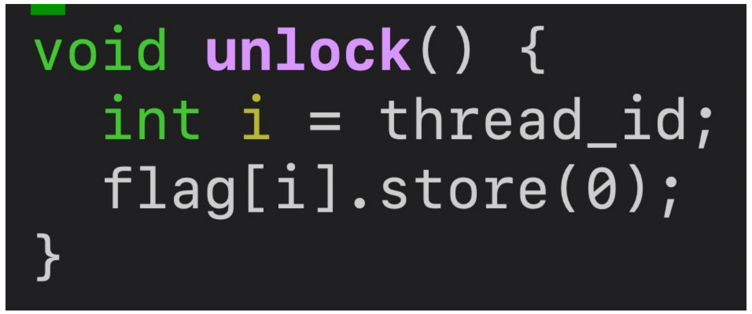
Thread 1:
```
m.lock();
m.unlock();
```



Analysis



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



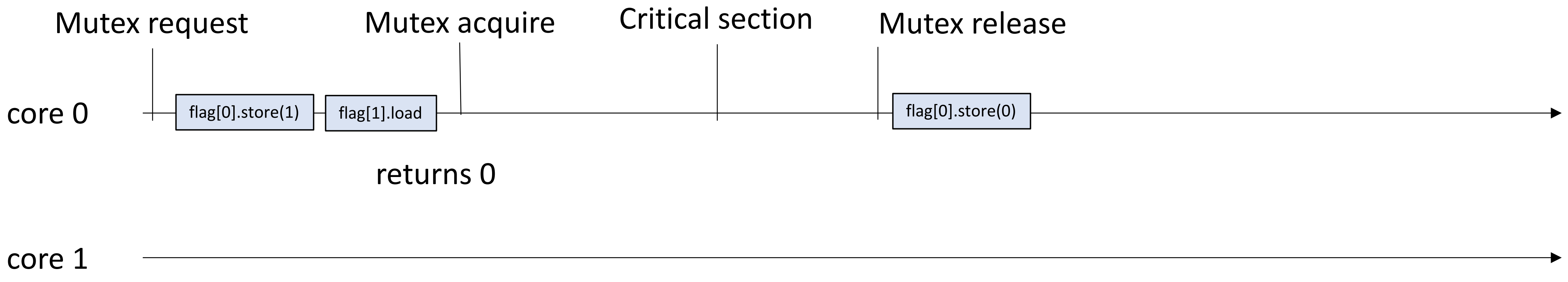
```
void unlock() {
  int i = thread_id;
  flag[i].store(0);
}
```

Thread 0:

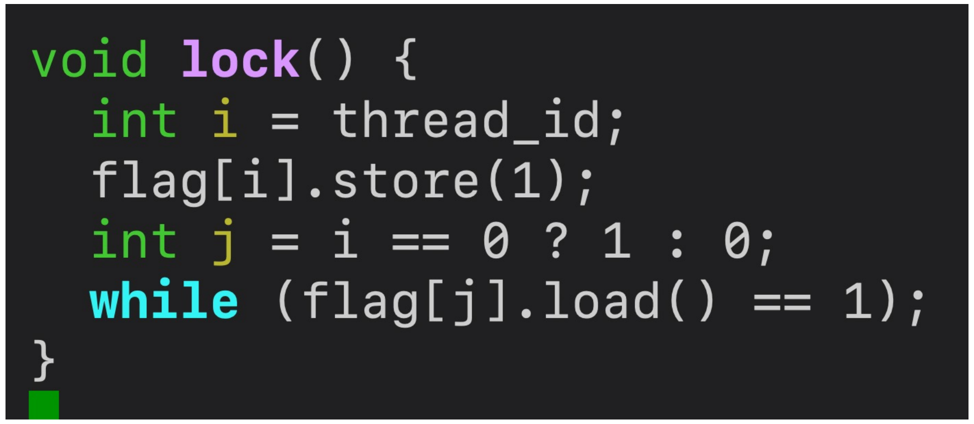
```
m.lock();
m.unlock();
```

Thread 1:

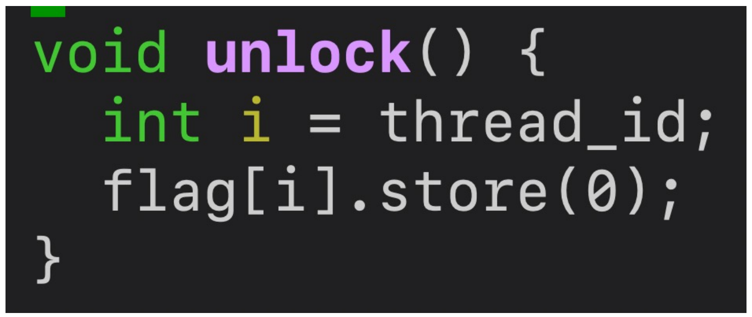
```
m.lock();
m.unlock();
```



Analysis



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

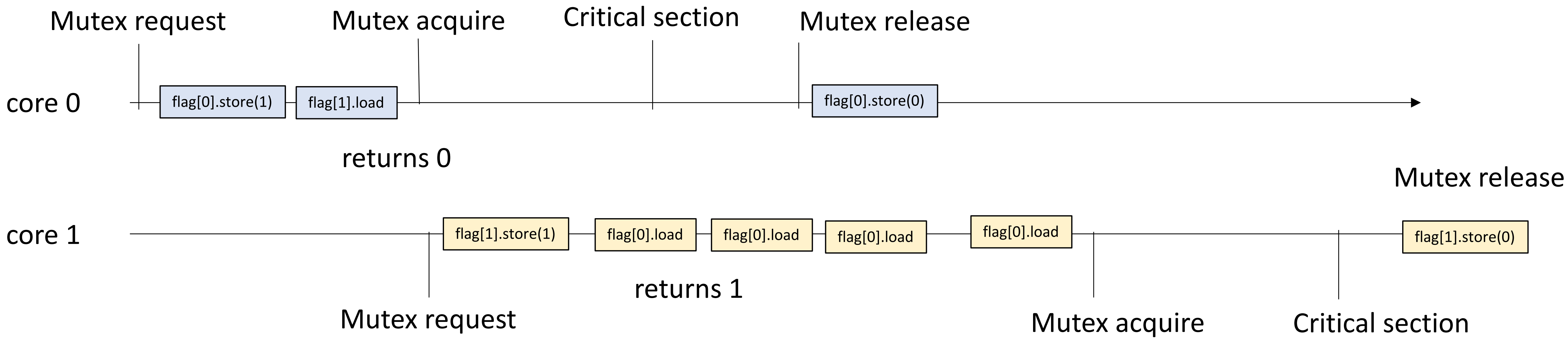


```
void unlock() {
  int i = thread_id;
  flag[i].store(0);
}
```

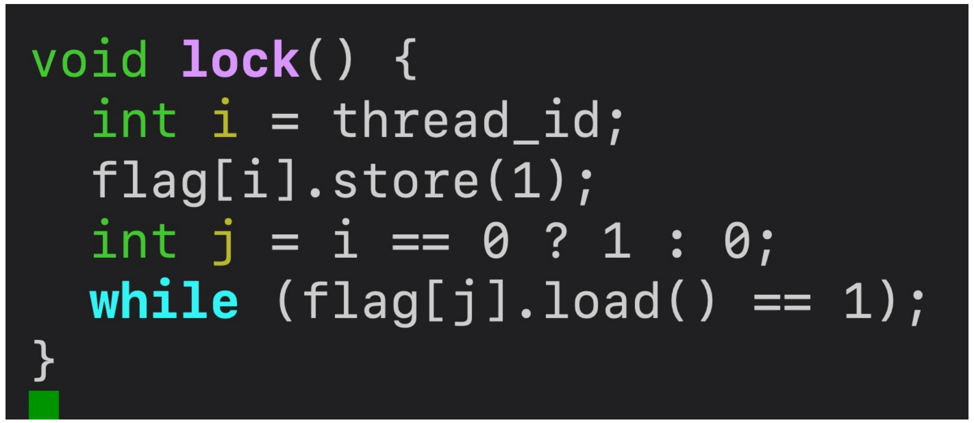
Thread 0:
```
m.lock();
m.unlock();
```

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

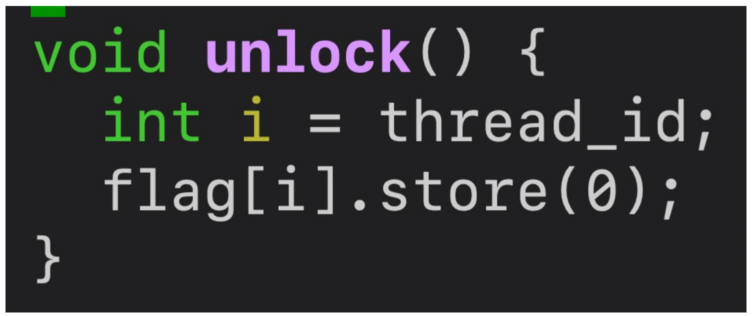
critical sections do not overlap!



Analysis



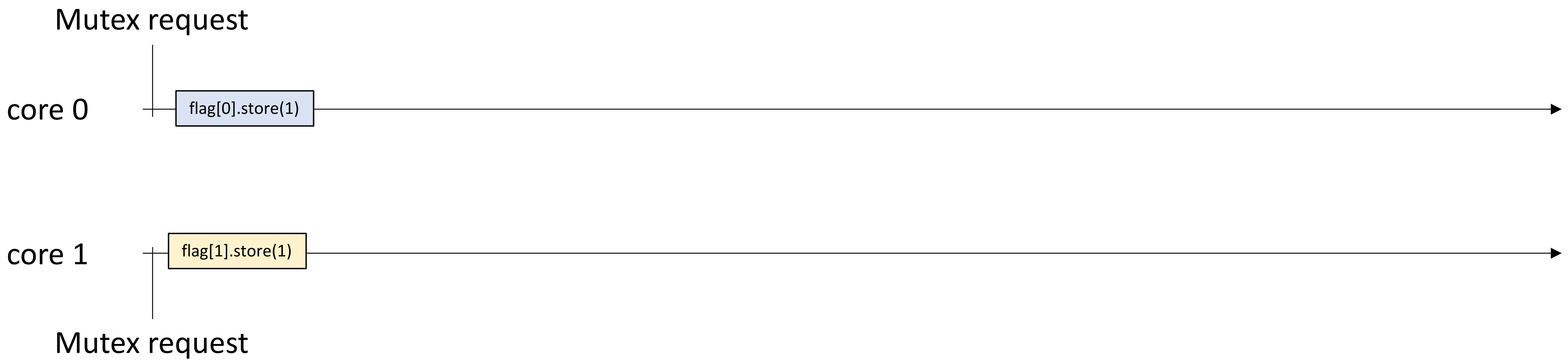
```
void lock() {
  int i = thread_id;
  flag[i].store(1);
  int j = i == 0 ? 1 : 0;
  while (flag[j].load() == 1);
}
```



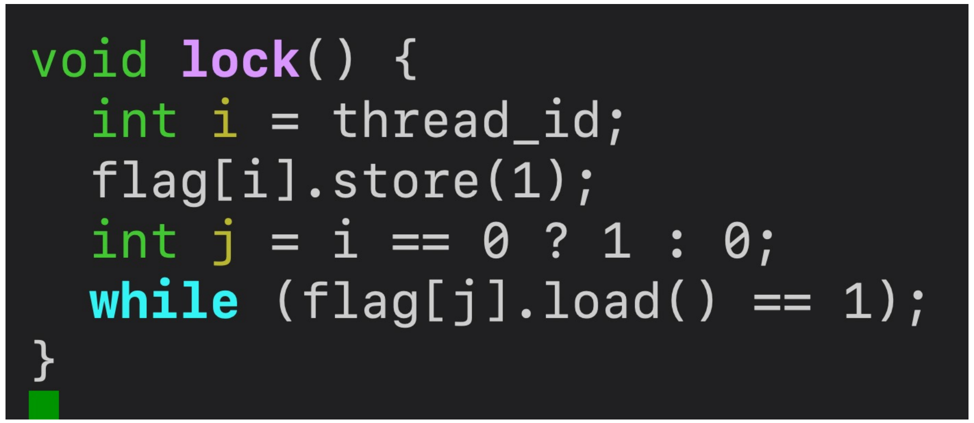
```
void unlock() {
  int i = thread_id;
  flag[i].store(0);
}
```

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

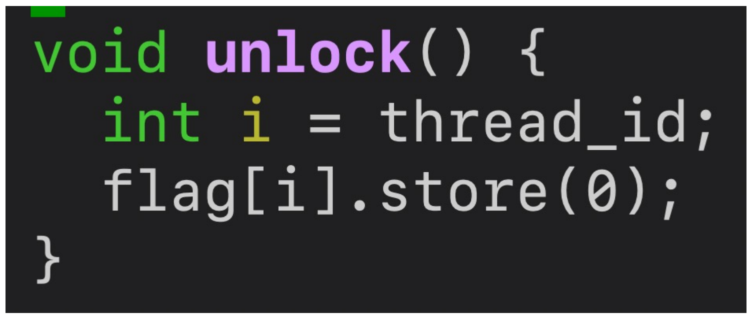
Thread 1:
```
m.lock();
m.unlock();
```



Analysis



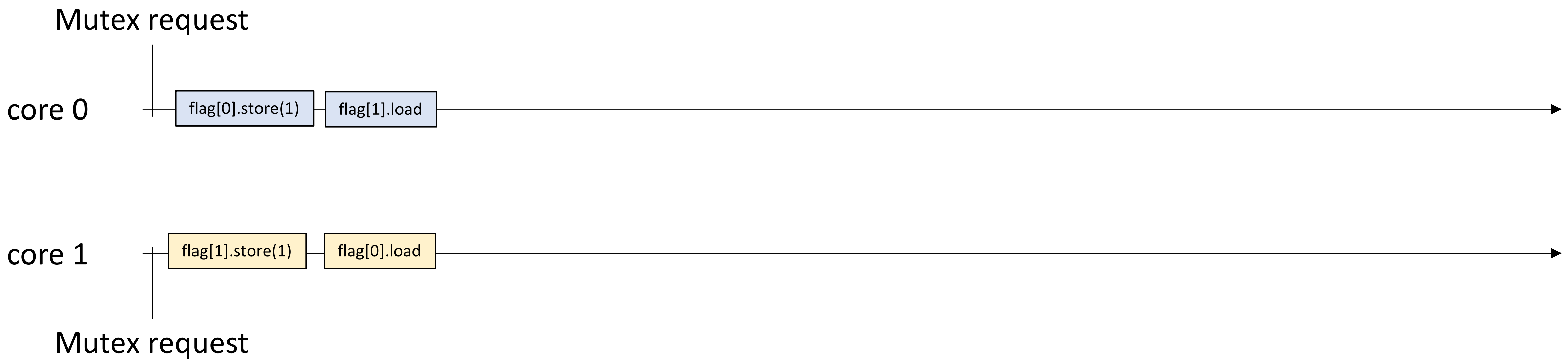
```
void lock() {
  int i = thread_id;
  flag[i].store(1);
  int j = i == 0 ? 1 : 0;
  while (flag[j].load() == 1);
}
```



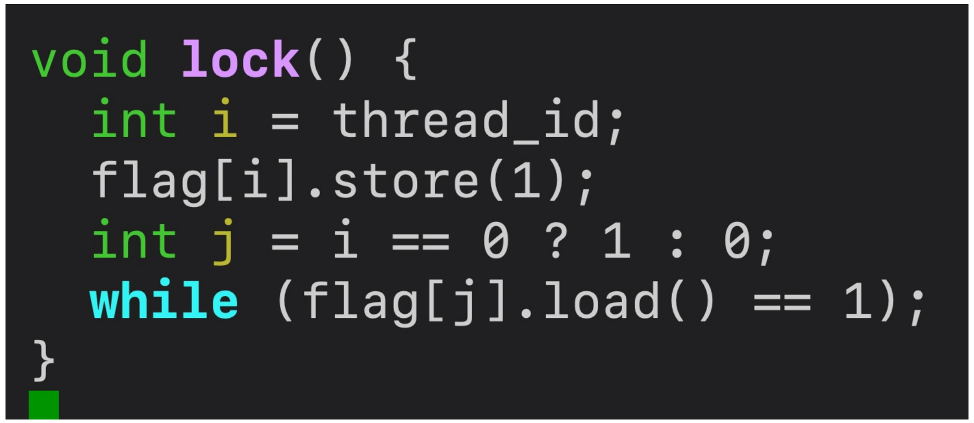
```
void unlock() {
  int i = thread_id;
  flag[i].store(0);
}
```

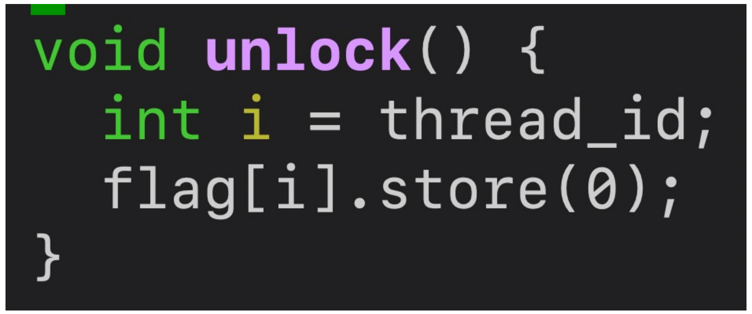
Thread 0:
```
m.lock();
m.unlock();
```

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



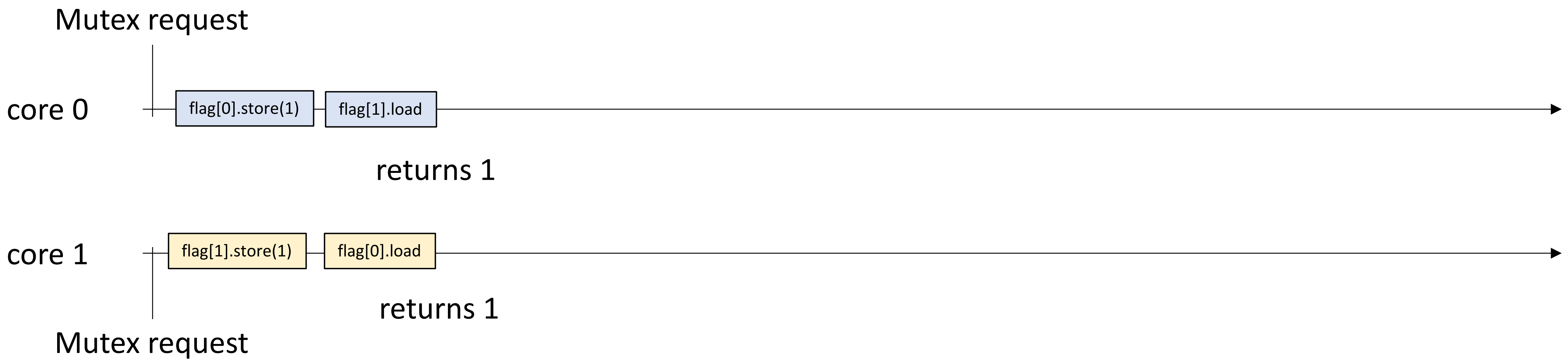
Analysis


```
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

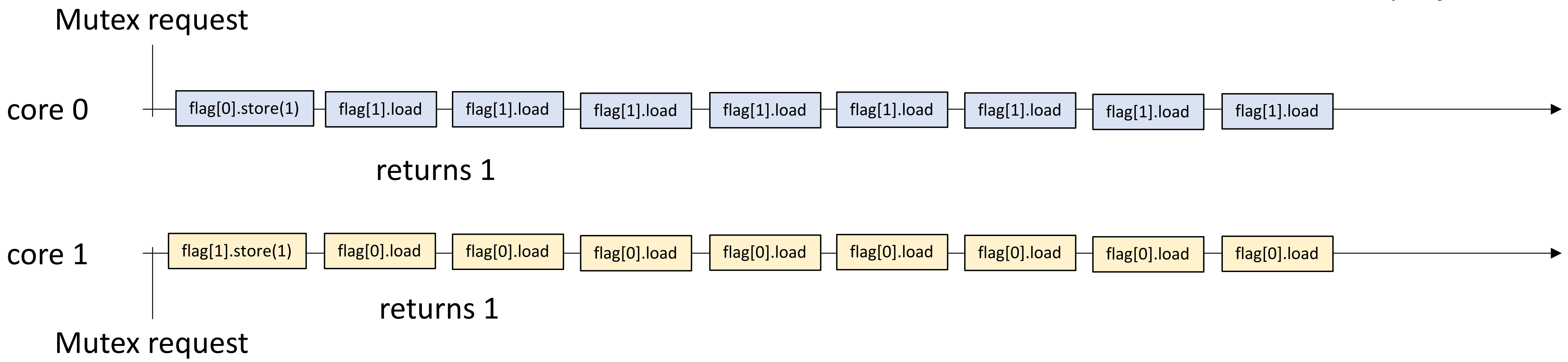
```
void lock() {
  int i = thread_id;
  flag[i].store(1);
  int j = i == 0 ? 1 : 0;
  while (flag[j].load() == 1);
}
```

```
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!



Mutex Implementations

Third attempt

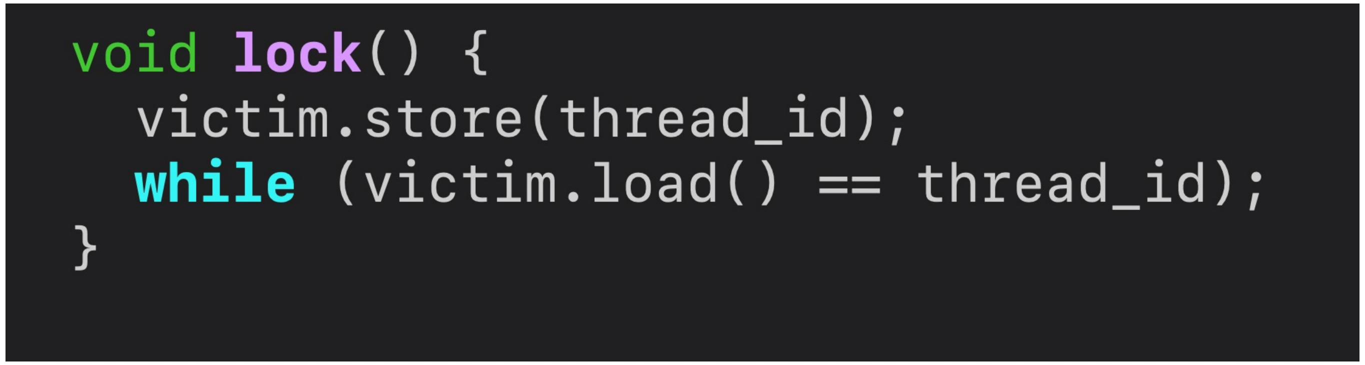
Mutex Implementations

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

  void lock();
  void unlock();


private:
  atomic_int victim;
};
```

initialized to -1

back to a single variable

Mutex Implementations

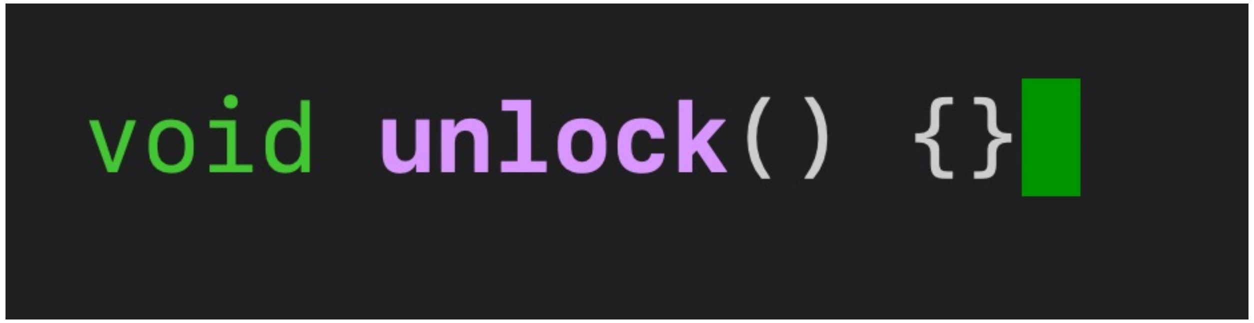


```
void lock() {
  victim.store(thread_id);
  while (victim.load() == thread_id);
}
```

Volunteer to be the victim

Victims only job is to spin

Mutex Implementations



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

No unlock!

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

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

Thread 0:

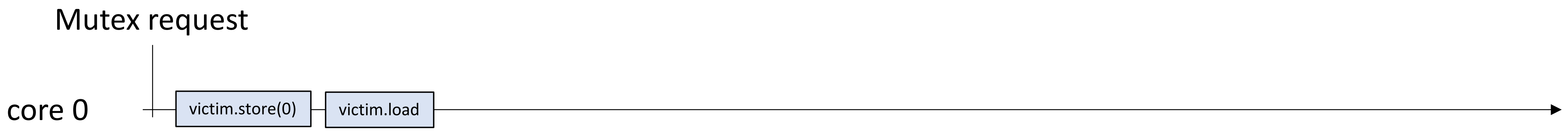
```
m.lock();
m.unlock();
```

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

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

Thread 0:

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



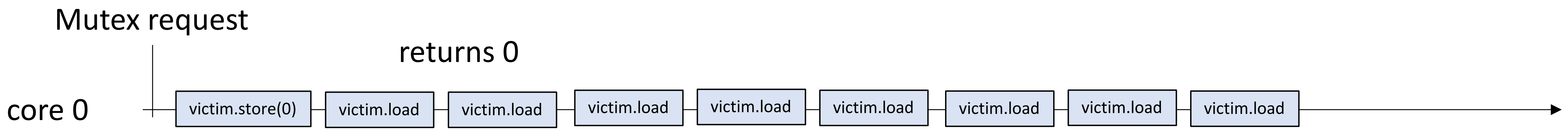
```
void lock() {
  victim.store(thread_id);
  while (victim.load() == thread_id);
}
```

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

Thread 0:

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

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



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

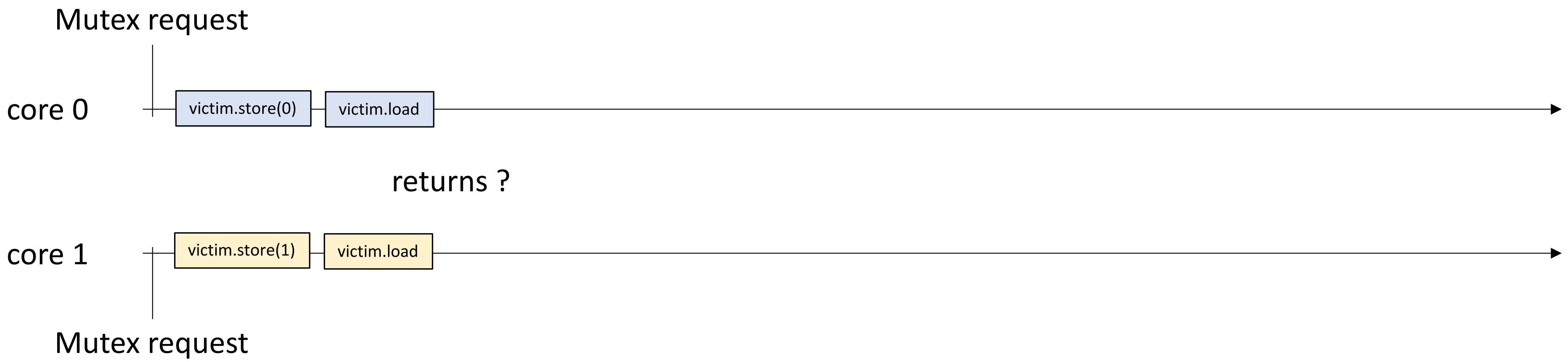
```
void unlock() {}
```

Thread 0:

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

Thread 1:

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

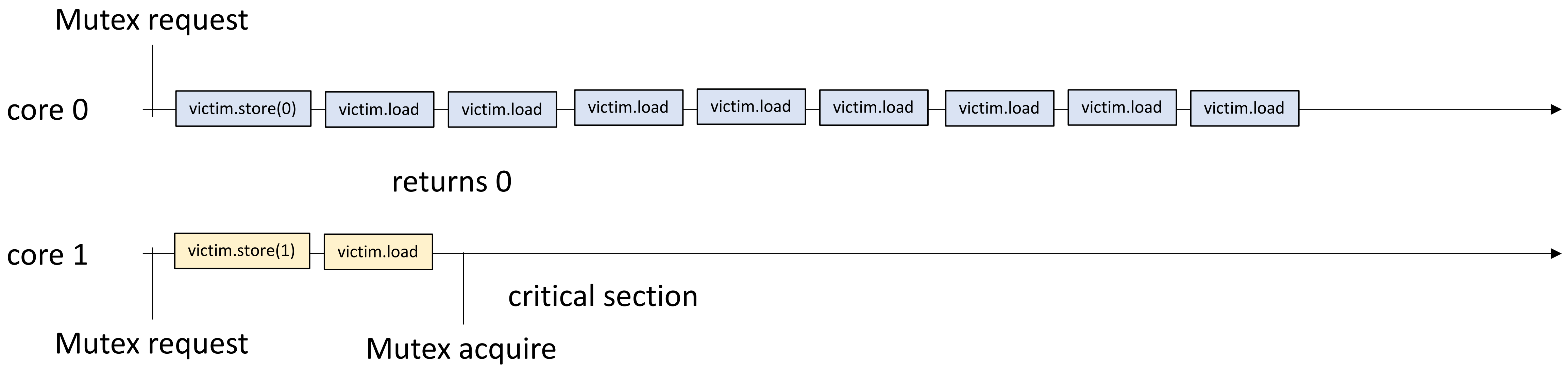


```
void lock() {
  victim.store(thread_id);
  while (victim.load() == thread_id);
}
```

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

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

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

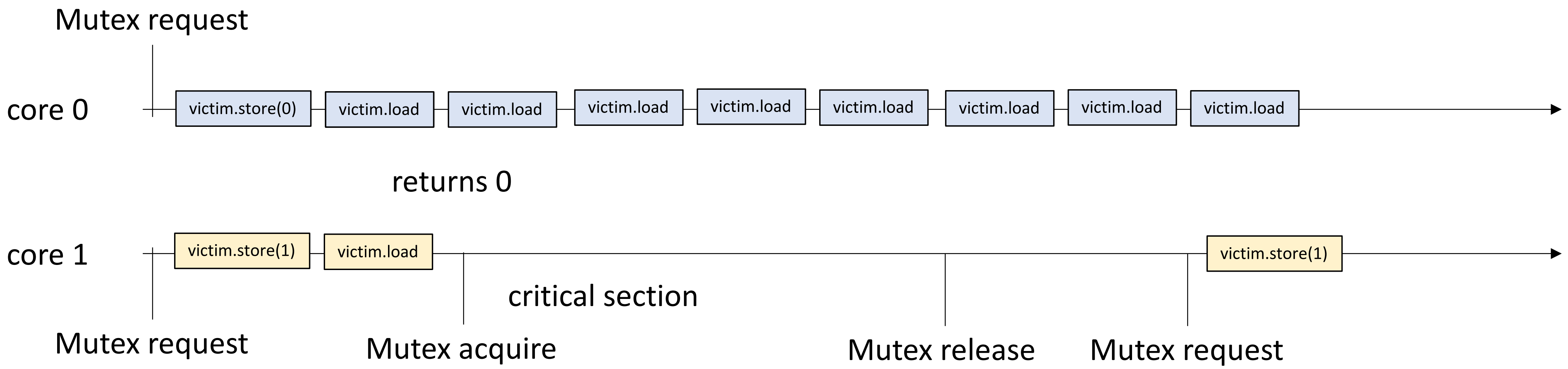


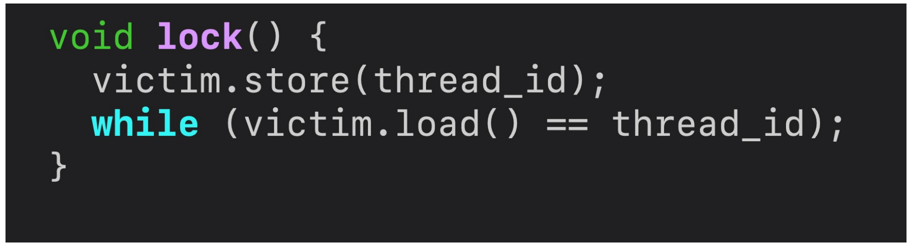
```
void lock() {
  victim.store(thread_id);
  while (victim.load() == thread_id);
}
```

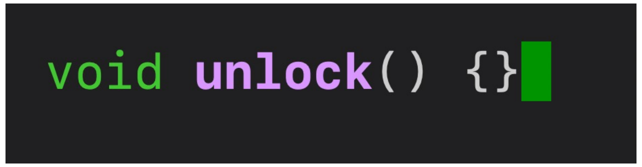
```
void unlock() {}
```

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

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

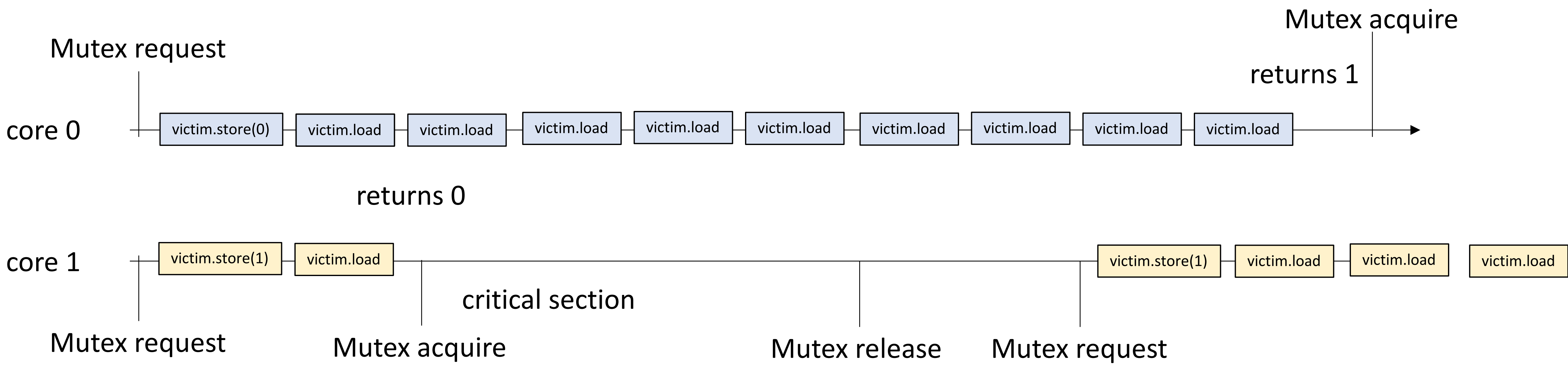



```
void lock() {
  victim.store(thread_id);
  while (victim.load() == thread_id);
}
```


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

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

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



Mutex Implementations

Finally, we can can make a mutex that works:

Use flags to mark interest

Use victim to break ties

Called the **Peterson Lock**

Mutex Implementations

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

  void lock();
  void unlock();


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

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);
}
```

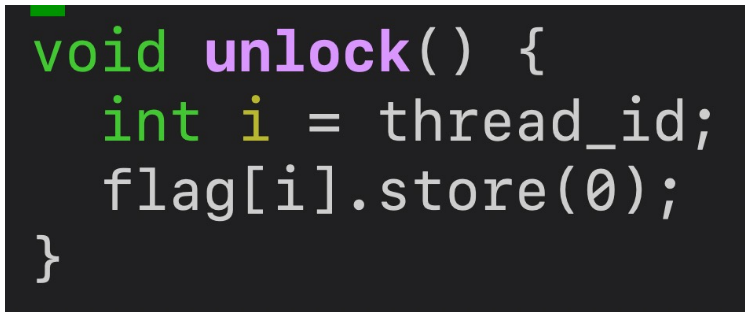
j is the other thread

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

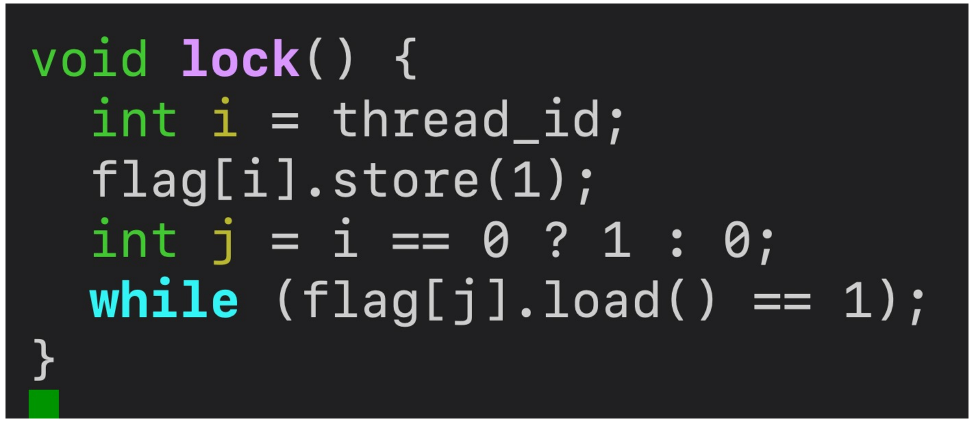
Mutex Implementations

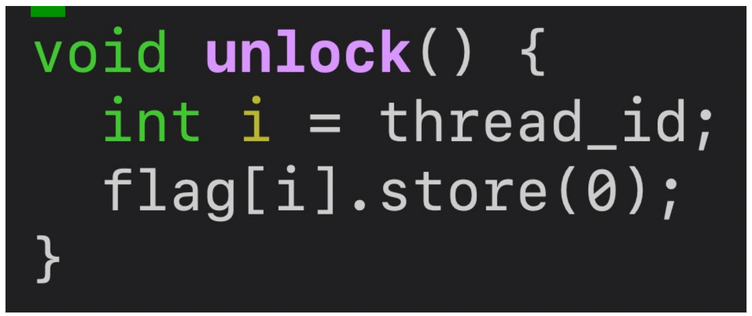


```
void unlock() {
  int i = thread_id;
  flag[i].store(0);
}
```

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);
}
```

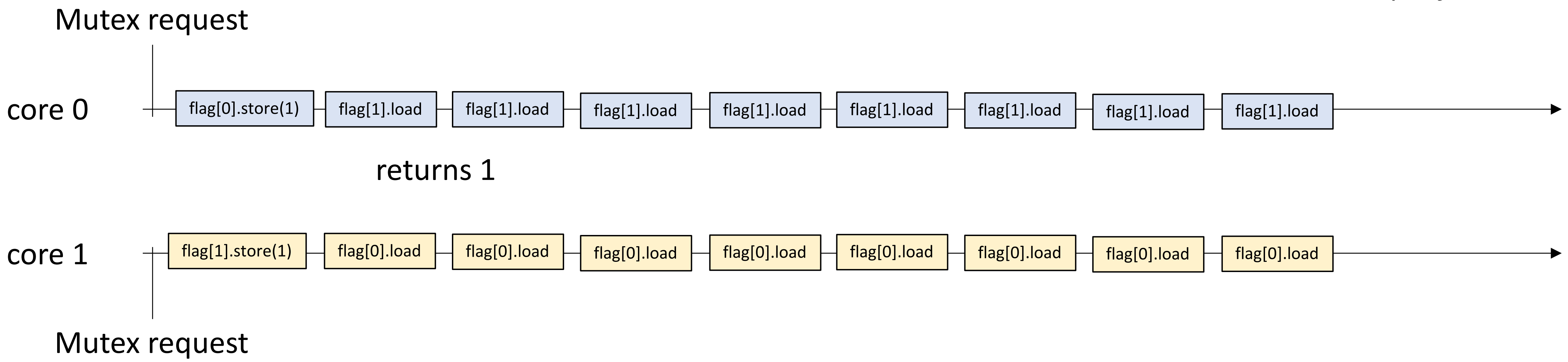

```
void unlock() {
  int i = thread_id;
  flag[i].store(0);
}
```

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

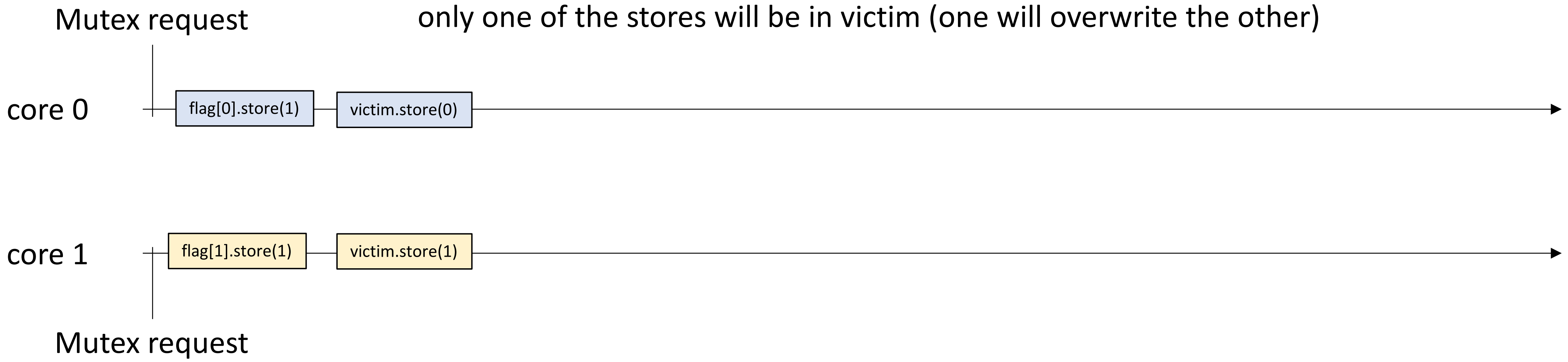
```
void lock() {
  int j = thread_id == 0 ? 1 : 0;
  flag[thread_id].store(1);
  victim.store(thread_id);
  while (victim.load() == thread_id
         && flag[j] == 1);
}
```

```
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);
}
```

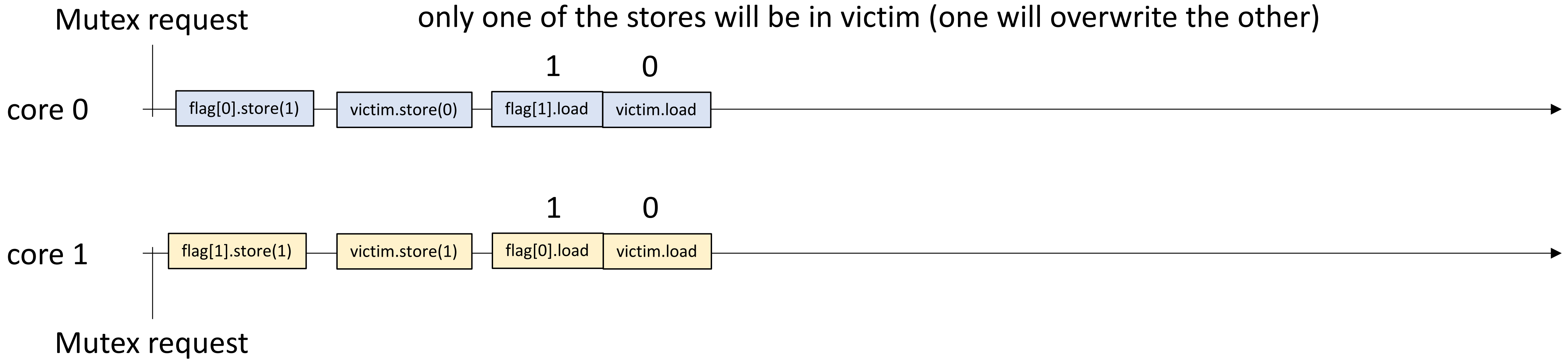
Thread 0:

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

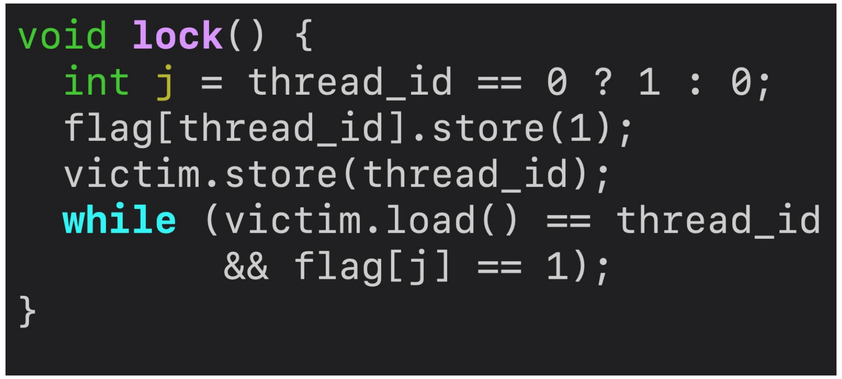
Thread 1:

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

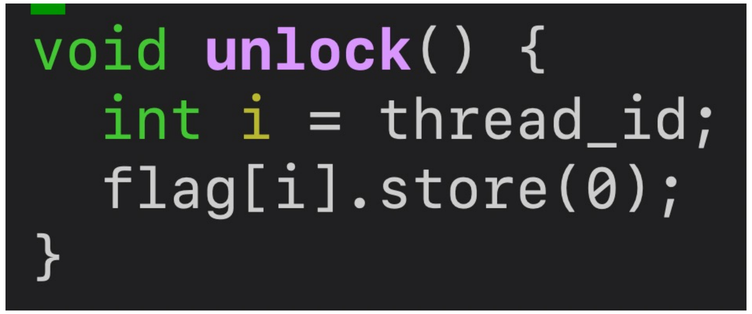
only one of the stores will be in victim (one will overwrite the other)



Tie breaking with victim



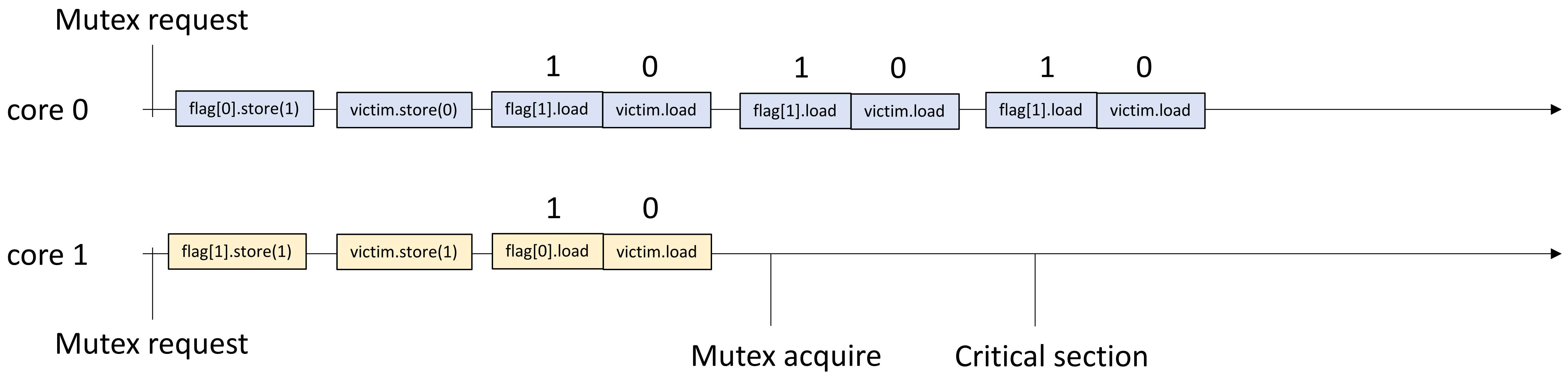
```
void lock() {
  int j = thread_id == 0 ? 1 : 0;
  flag[thread_id].store(1);
  victim.store(thread_id);
  while (victim.load() == thread_id
         && flag[j] == 1);
}
```



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

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

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



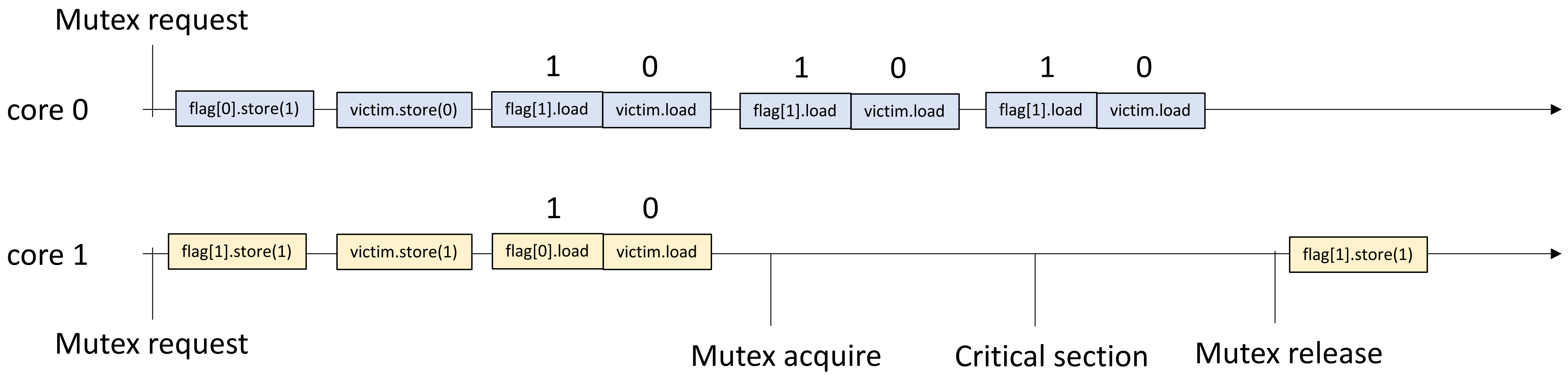
Tie breaking with victim

```
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);
}
```

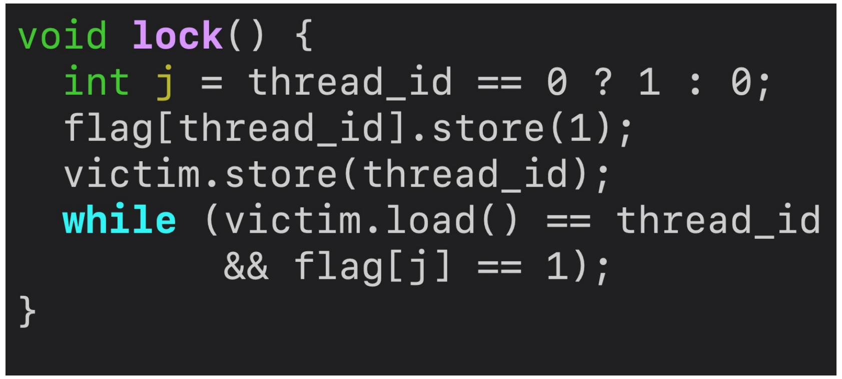
```
void unlock() {
  int i = thread_id;
  flag[i].store(0);
}
```

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

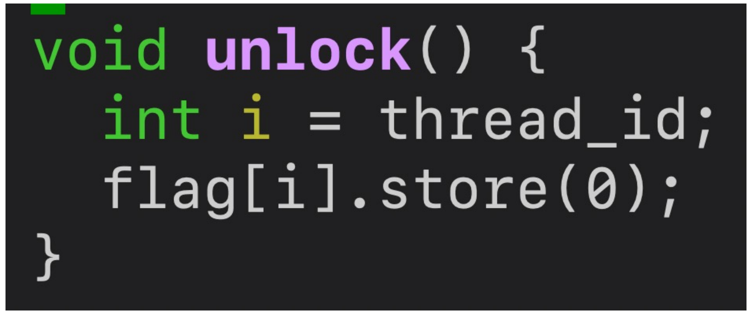
Thread 1:
```
m.lock();
m.unlock();
```



Tie breaking with victim



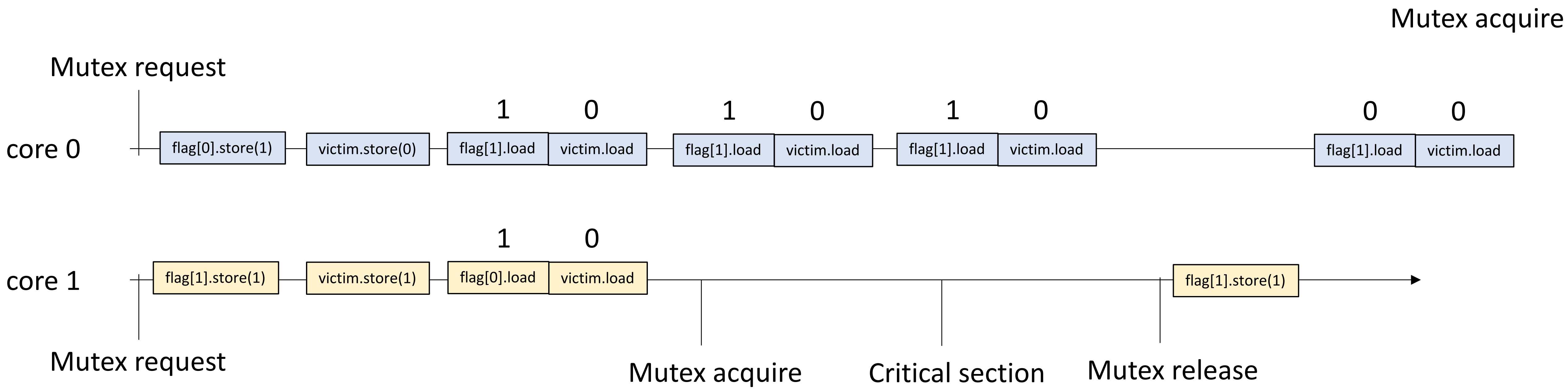
```
void lock() {
  int j = thread_id == 0 ? 1 : 0;
  flag[thread_id].store(1);
  victim.store(thread_id);
  while (victim.load() == thread_id
         && flag[j] == 1);
}
```



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

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

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



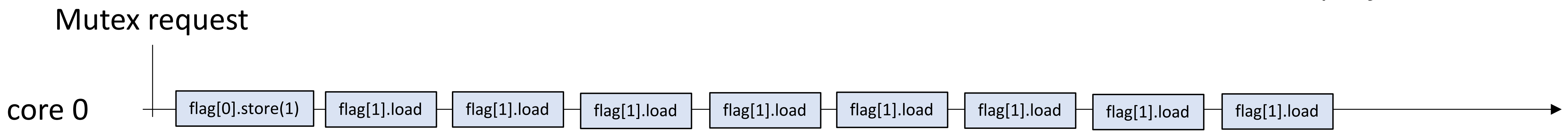
previous victim issue

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

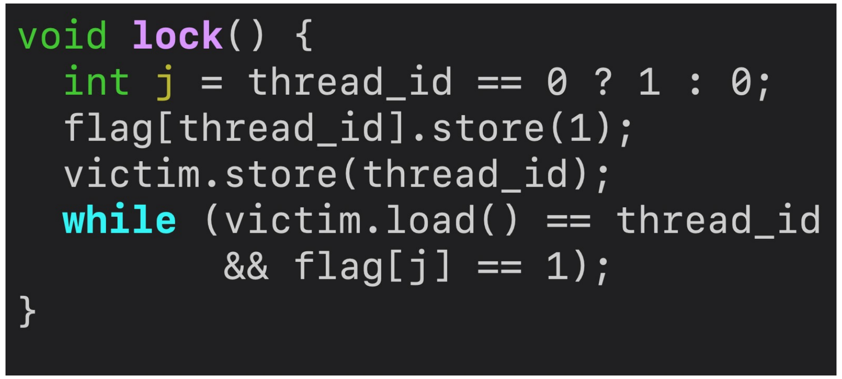
```
void unlock() {}
```

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

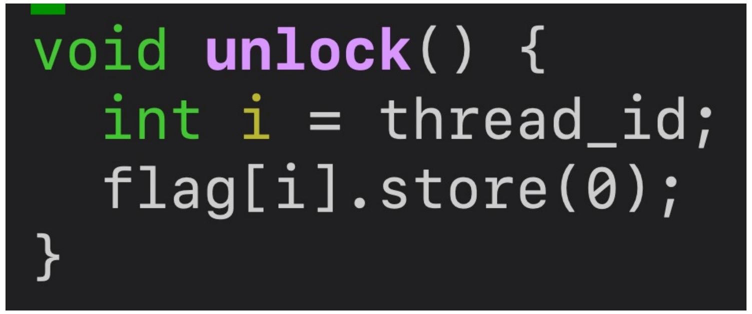
will spin forever!



previous flag issue



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

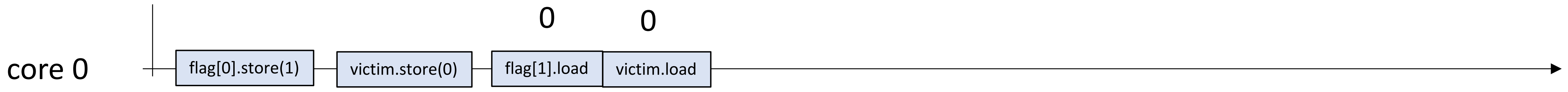


```
void unlock() {
   int i = thread_id;
   flag[i].store(0);
}
```

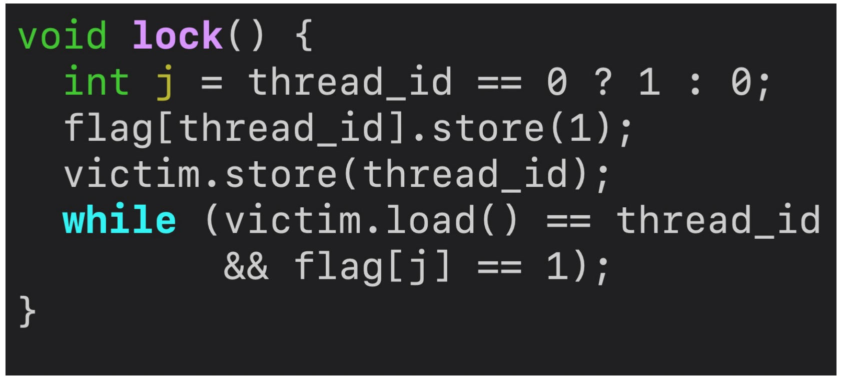
Thread 0:

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

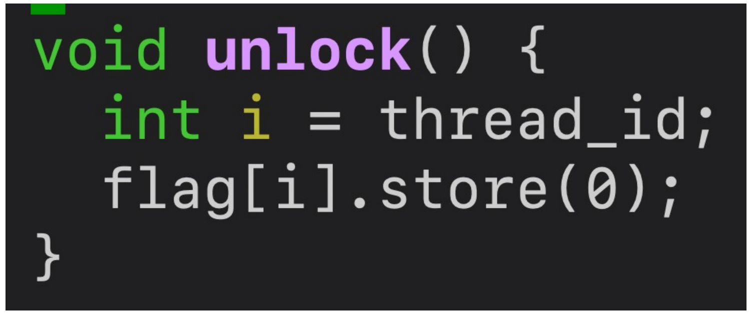
Mutex request



previous flag issue



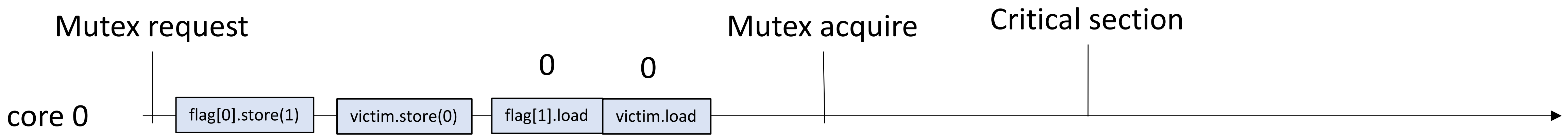
```
void lock() {
  int j = thread_id == 0 ? 1 : 0;
  flag[thread_id].store(1);
  victim.store(thread_id);
  while (victim.load() == thread_id
         && flag[j] == 1);
}
```



```
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

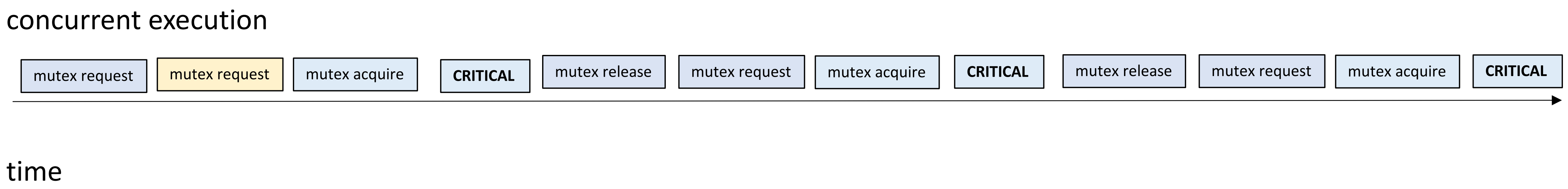
This lock satisfies the two critical properties

- Mutual exclusion
- Deadlock freedom
- *More formal proof given in the textbook*
- *How might we test it?*

What about starvation

recall the starvation property:

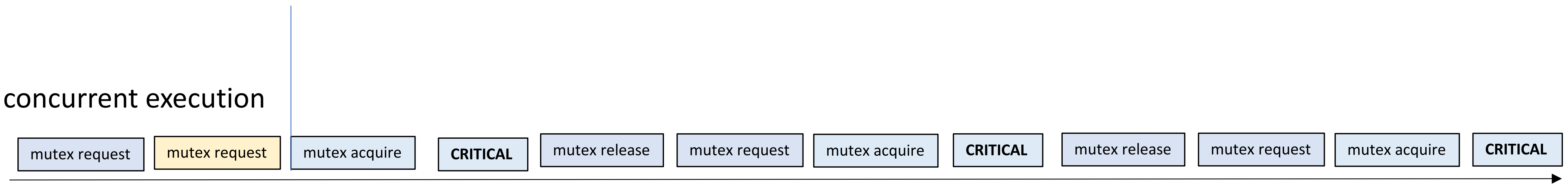
Thread 1 (yellow) requests the mutex but never gets it



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



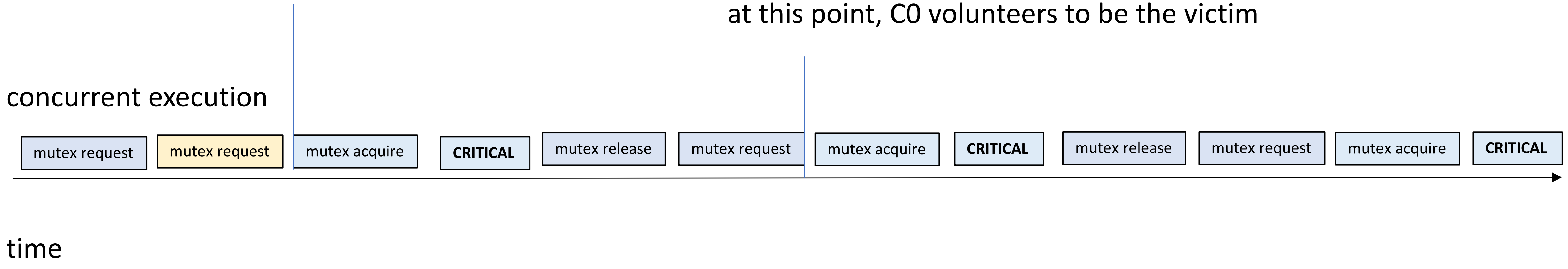
time

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);
}
```

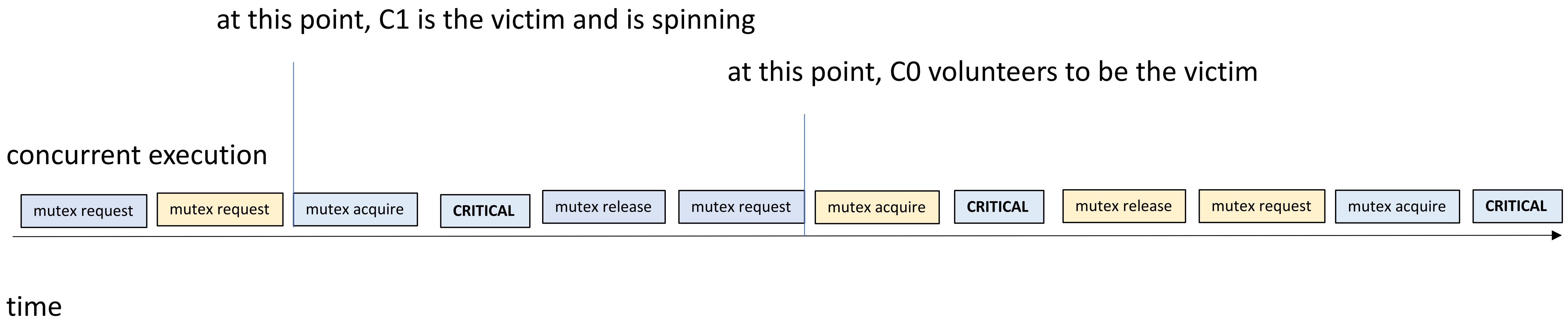
at this point, C1 is the victim and is spinning

at this point, C0 volunteers to be the victim



What about starvation

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



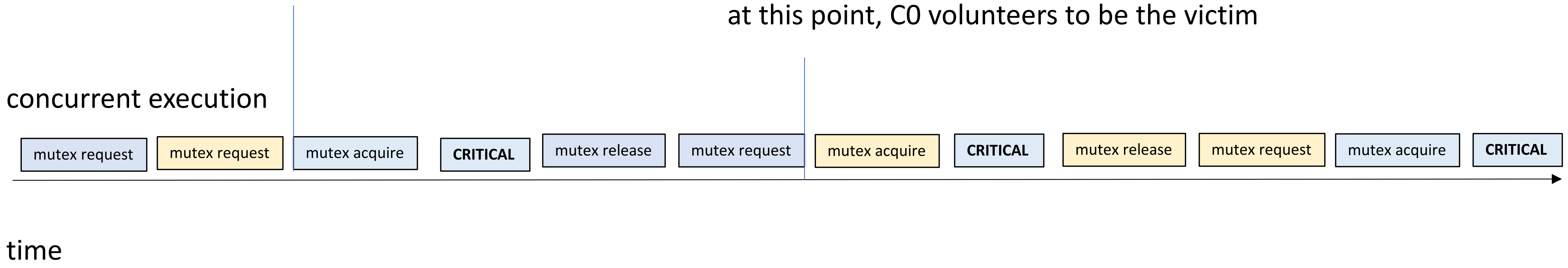
What about starvation

Threads take turns in petersons algorithm. It is starvation free

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

at this point, C1 is the victim and is spinning

at this point, C0 volunteers to be the victim



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

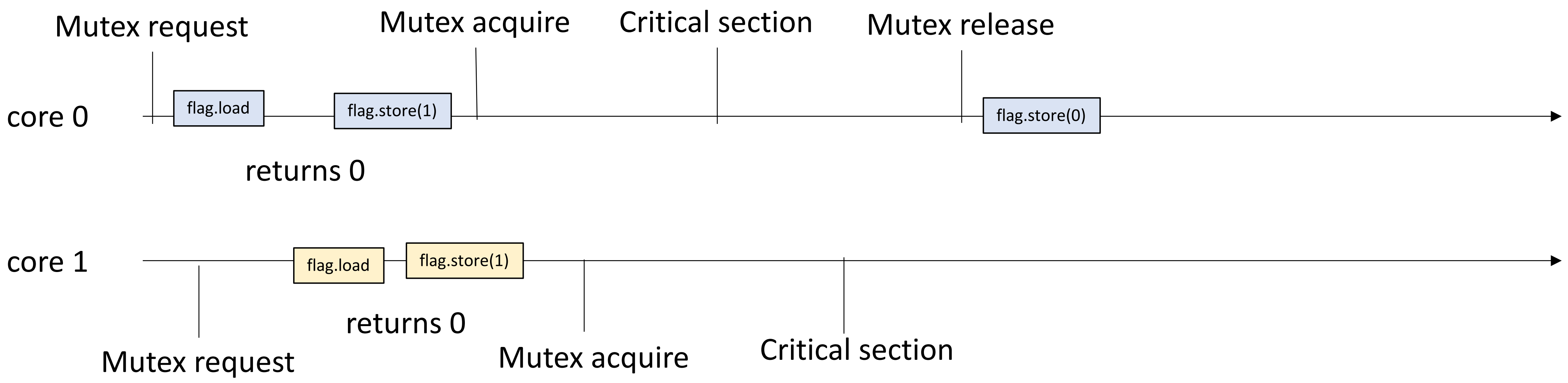
```
void lock() {
  while (flag.load() == 1);
  flag.store(1);
}
```

```
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
}
```

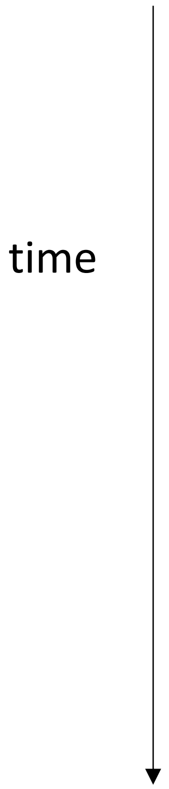
lock-free accounts

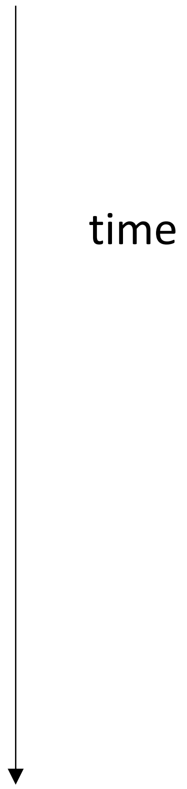
Tyler's coffee addiction:

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

Tyler's employer

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





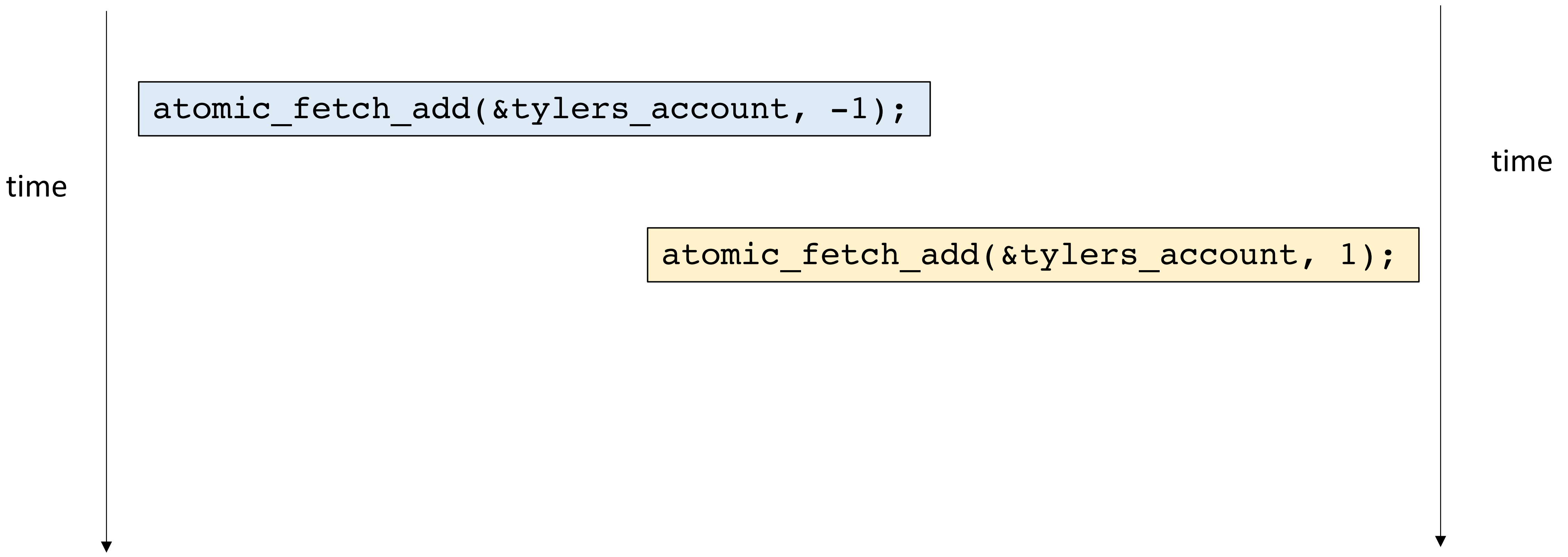
lock-free accounts

Tyler's coffee addiction:

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

Tyler's employer

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


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

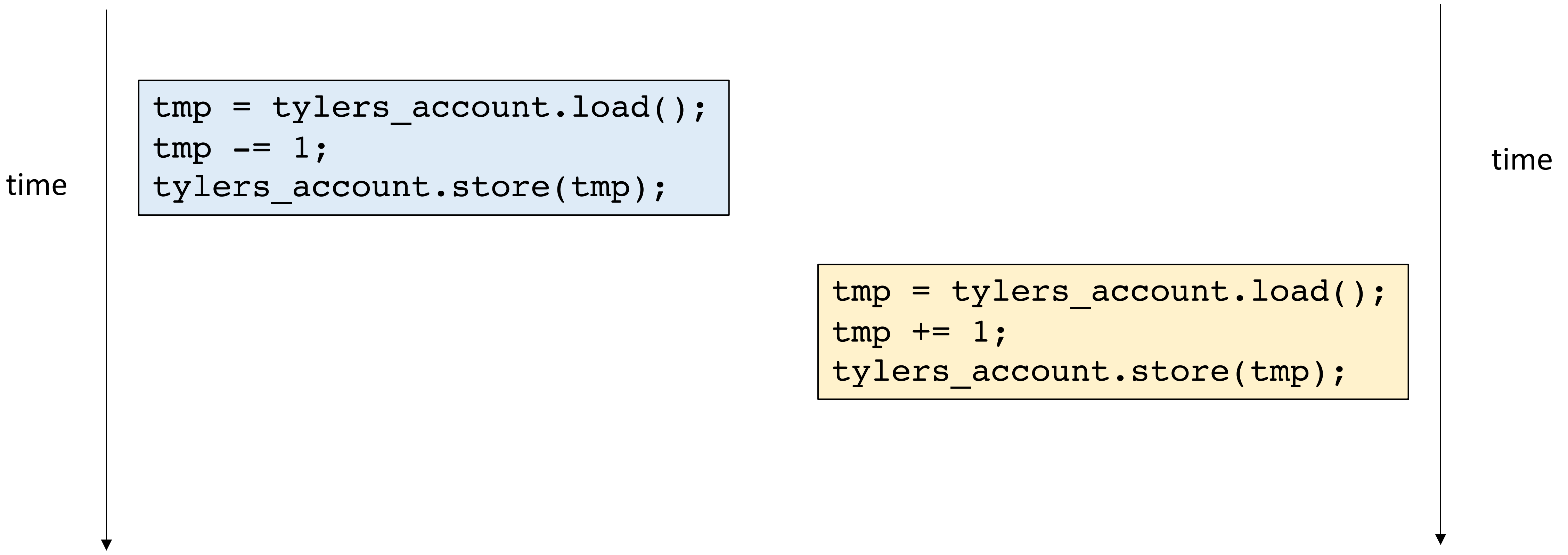
lock-free accounts

Tyler's coffee addiction:

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

Tyler's employer

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



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

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

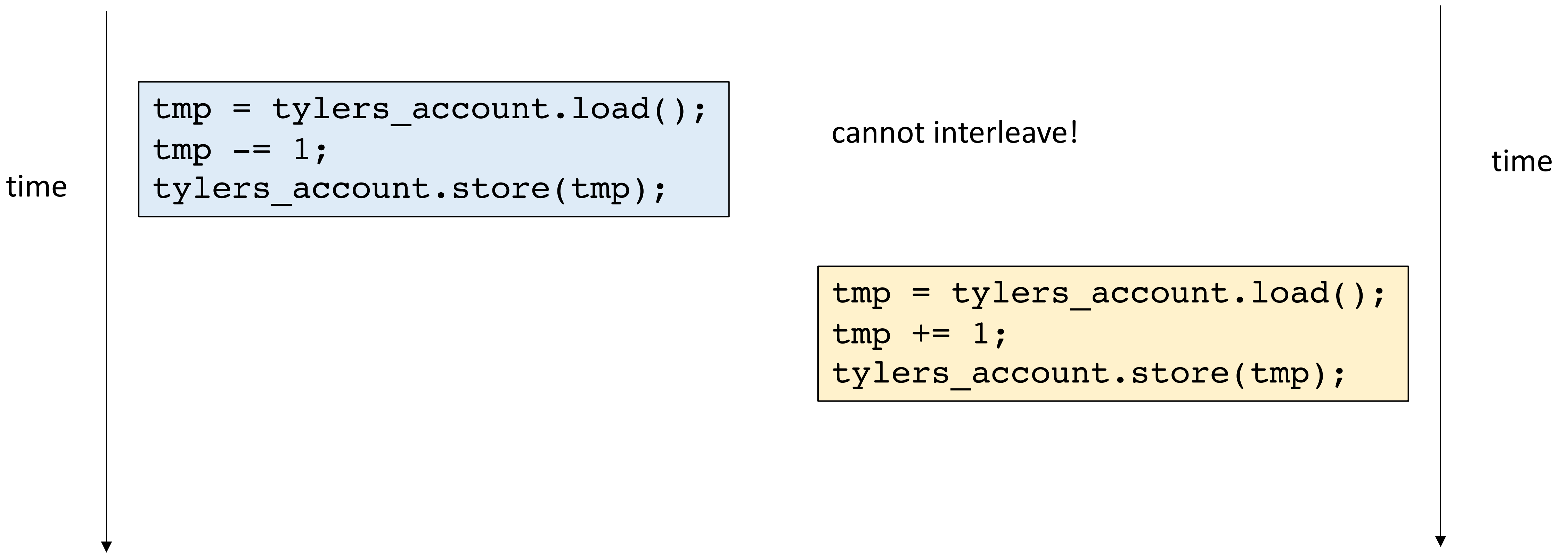
lock-free accounts

Tyler's coffee addiction:

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

Tyler's employer

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



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

```
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!

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

time

either way, account breaks even at the end!

RMW-based locks

- A few simple RMWs enable lots of interesting mutex implementations
- When we have simpler implementations, we can focus on performance

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

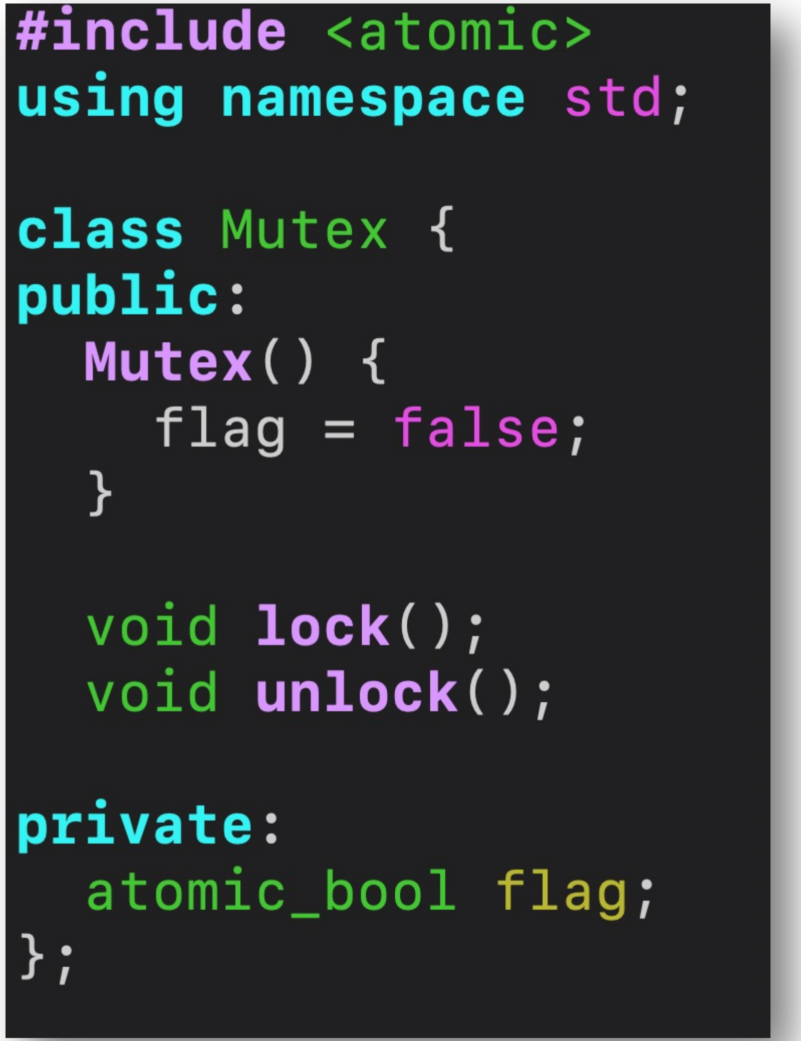
```
value atomic_exchange(atomic *a, value v);
```

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;
}
```

no "modify" step!

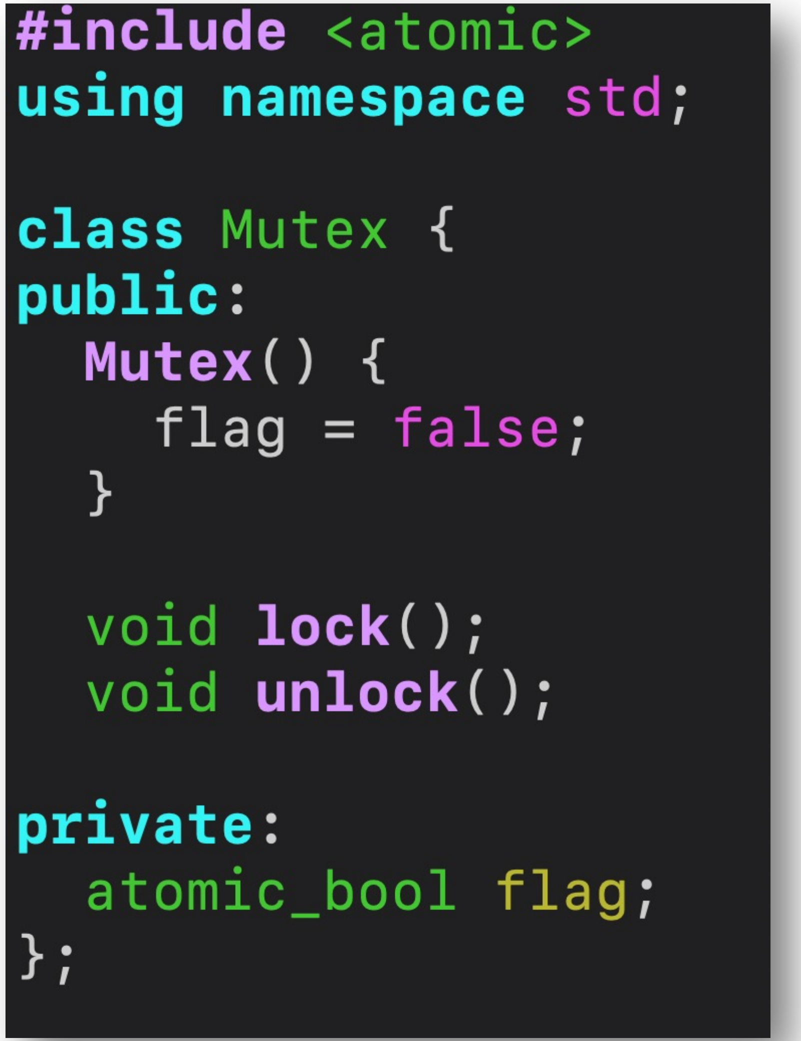
First example: Exchange Lock



```
#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



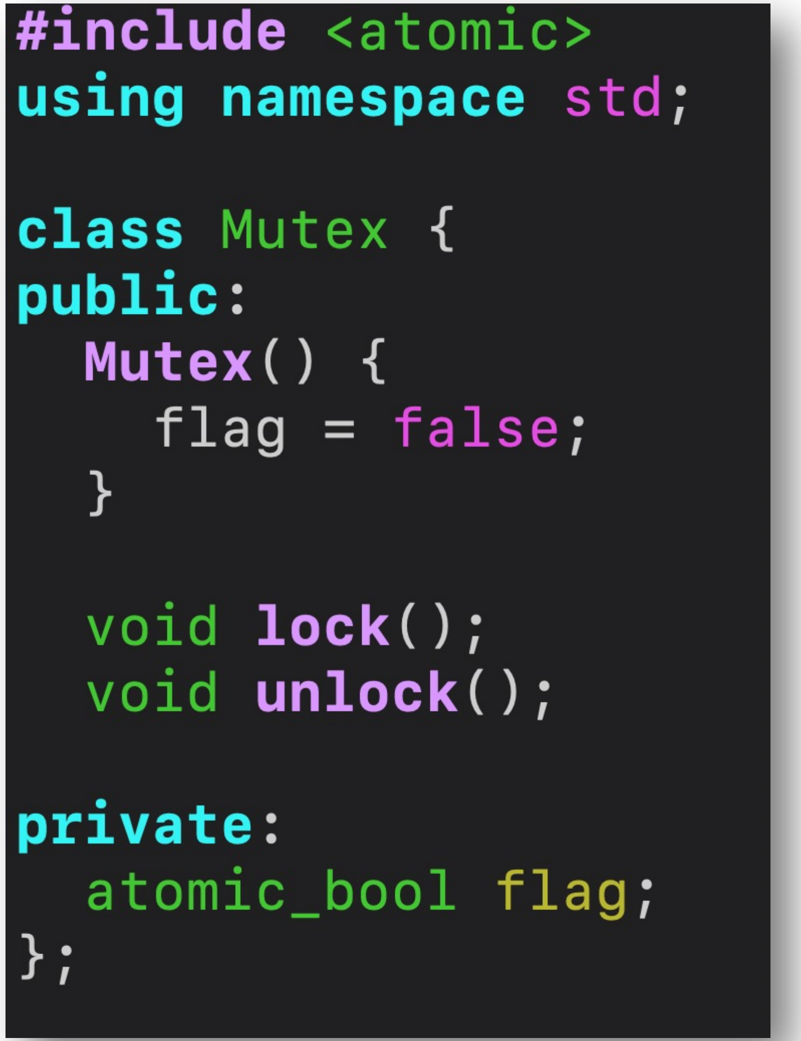
```
#include <atomic>
using namespace std;

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`

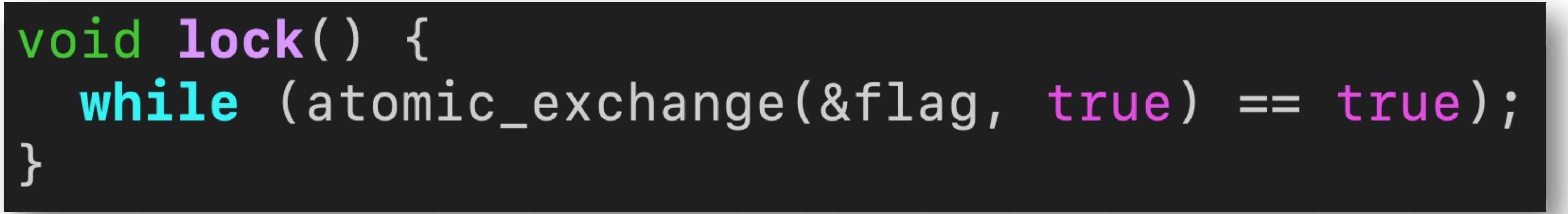
one atomic flag

main idea:

The flag is false when the mutex is free.

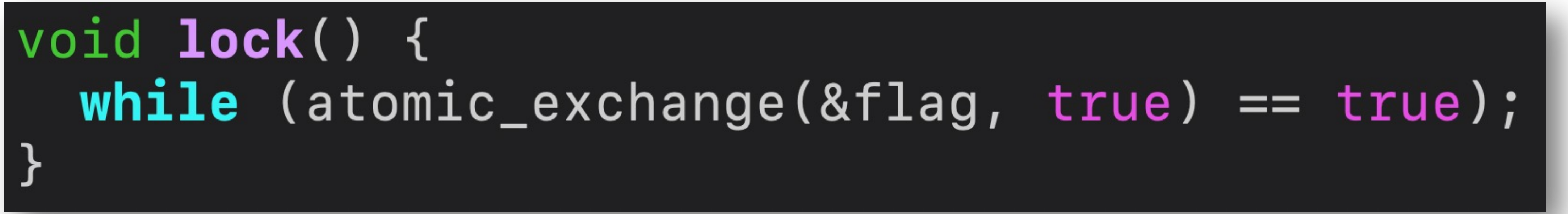
The flag is true when some thread has the mutex.

First example: Exchange Lock



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

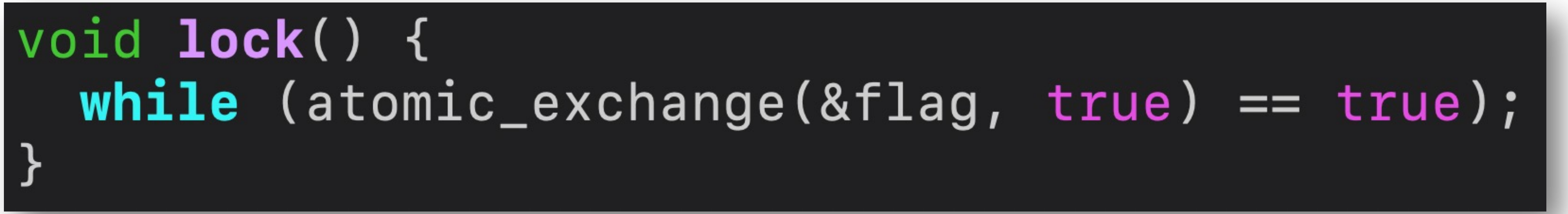
First example: Exchange Lock



```
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

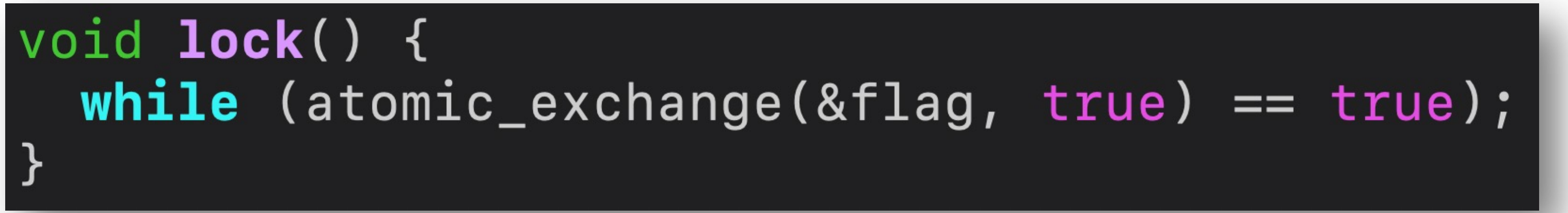
mutex is taken: the value loaded is true, we put the SAME value back (true). The returned value is true, so we spin.

First example: Exchange Lock

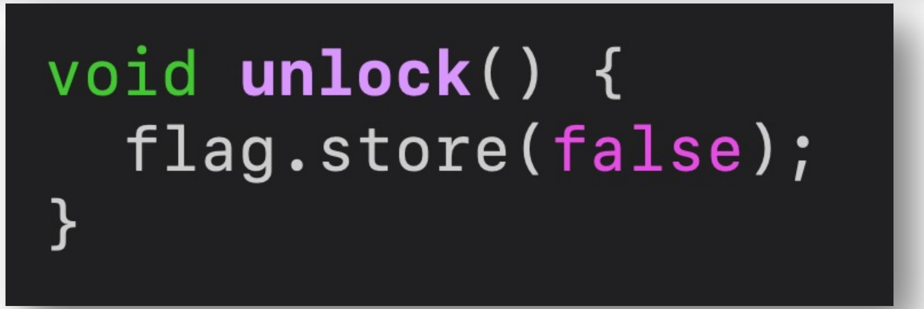
```
void unlock() {
  flag.store(false);
}
```

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

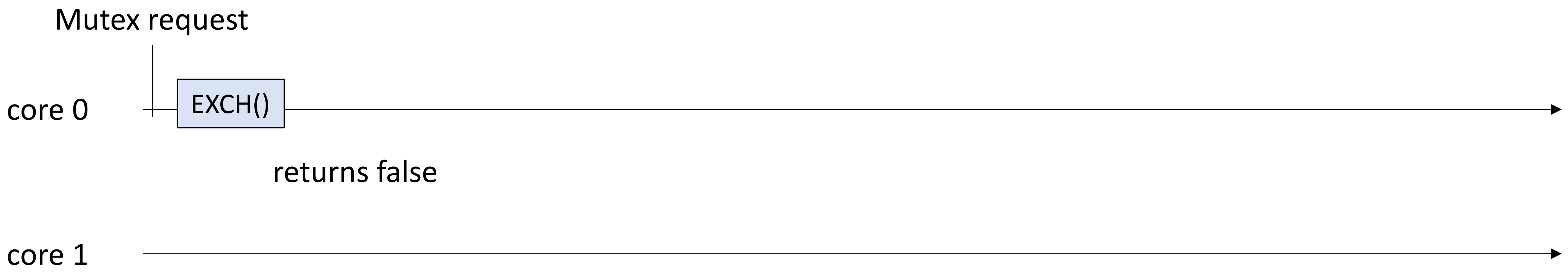
Analysis

```
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();
```



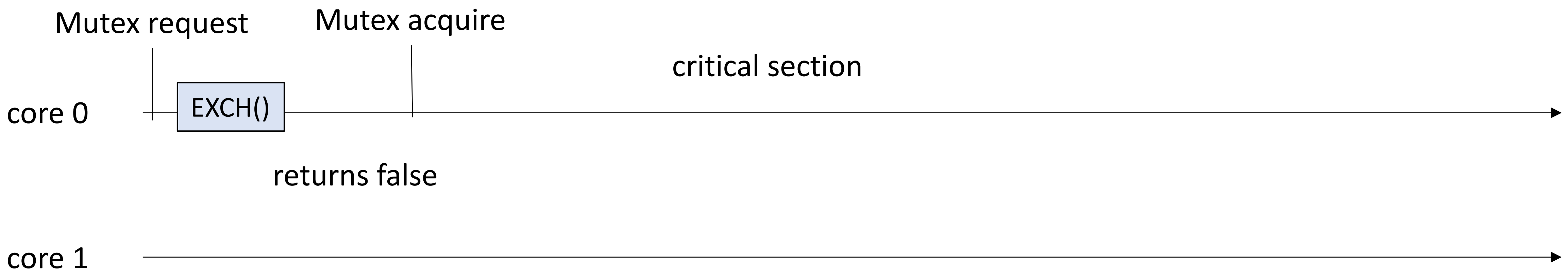
Analysis

```
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();
```



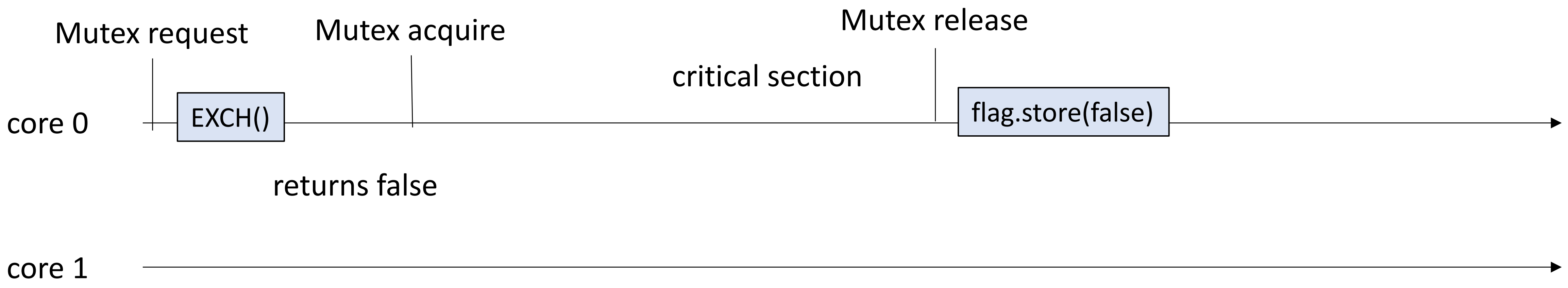
Analysis

```
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();
```



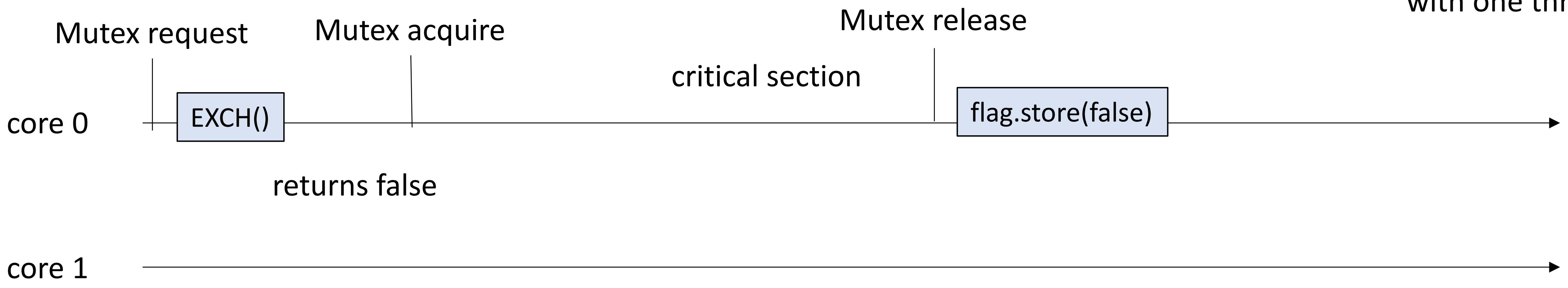
Analysis

```
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);
}
```

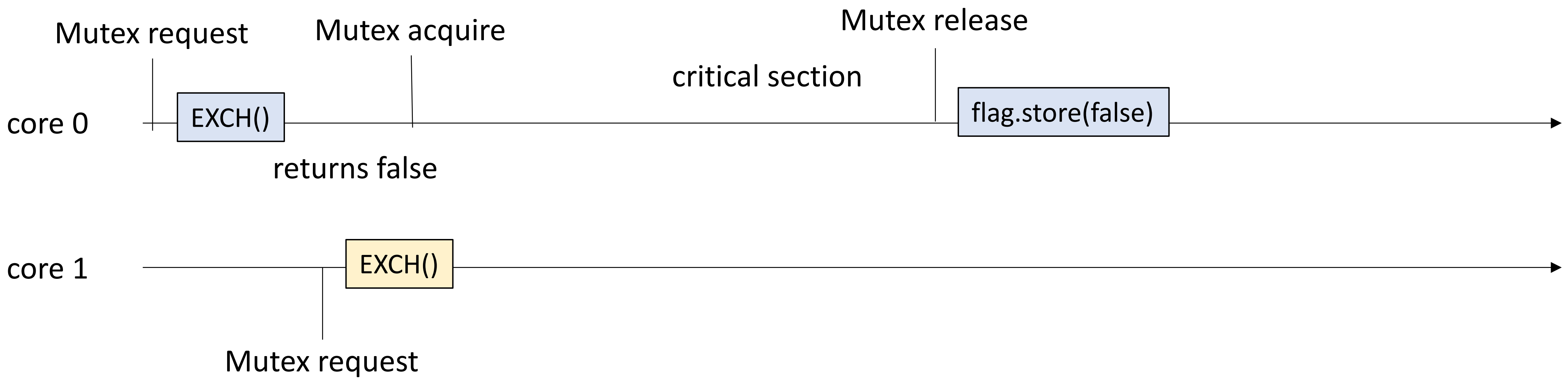
```
void unlock() {
  flag.store(false);
}
```

Thread 0:

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

Thread 1:

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



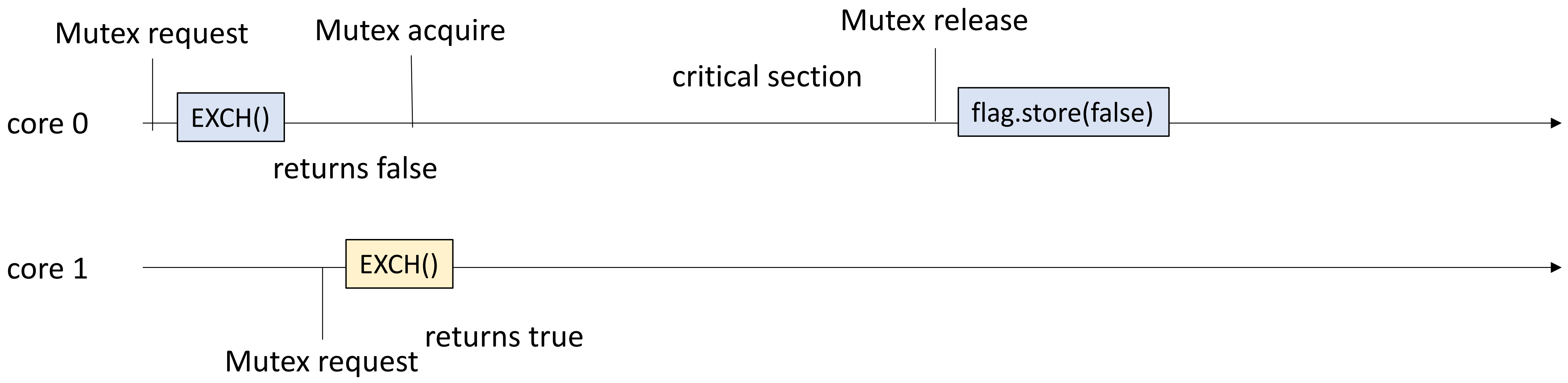
Analysis

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

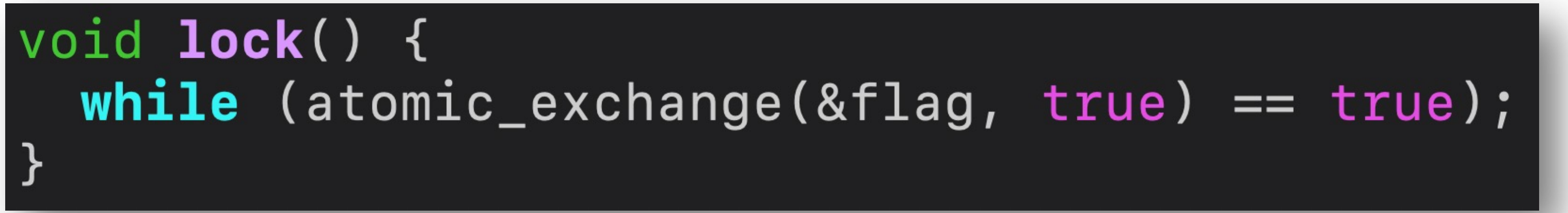
```
void unlock() {
  flag.store(false);
}
```

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

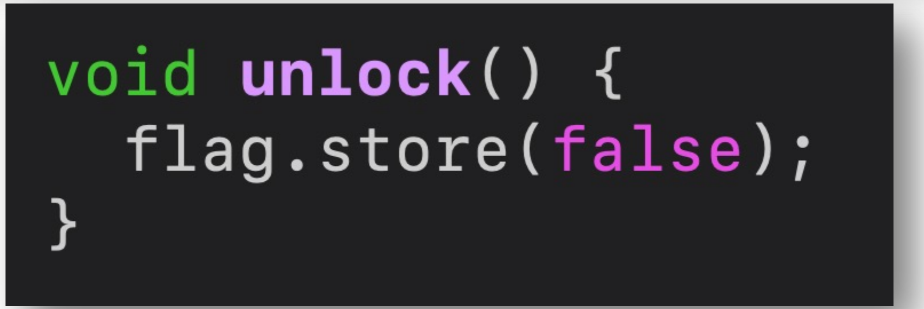
Thread 1:
```
m.lock();
m.unlock();
```



Analysis



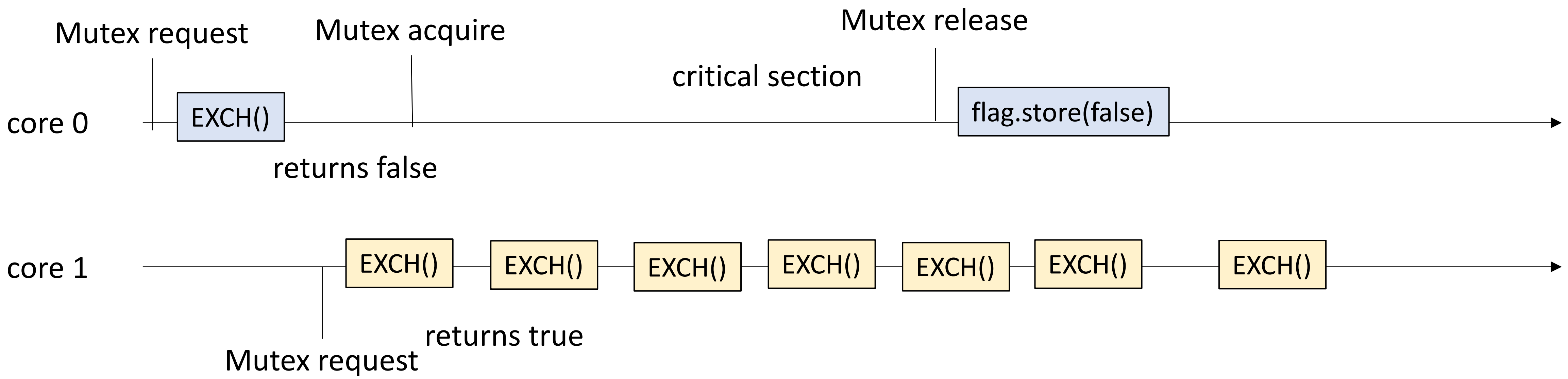
```
void lock() {
  while (atomic_exchange(&flag, true) == true);
}
```



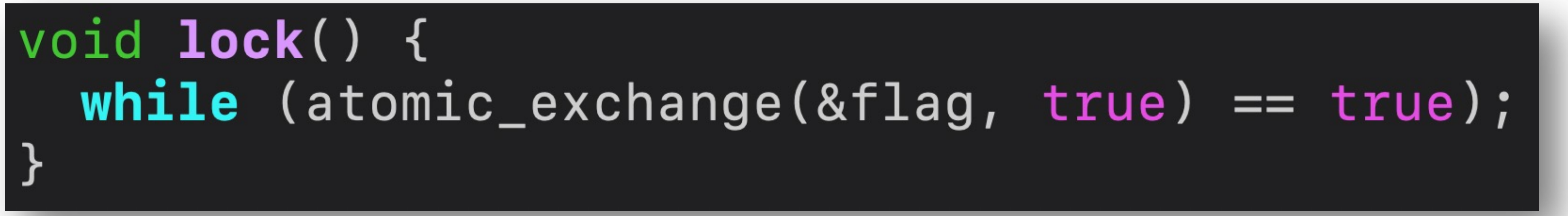
```
void unlock() {
  flag.store(false);
}
```

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

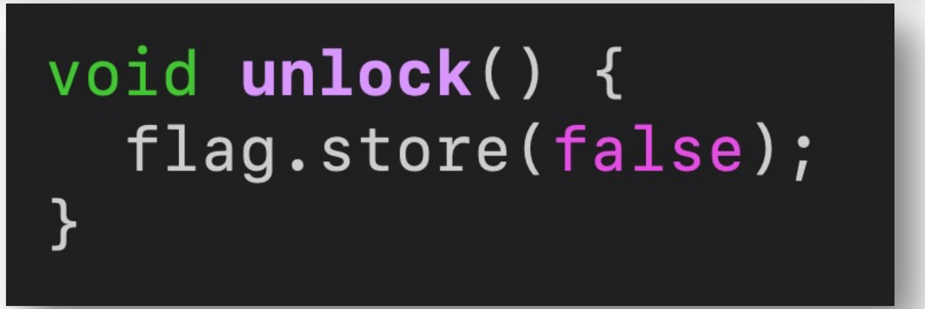
Thread 1:
```
m.lock();
m.unlock();
```



Analysis



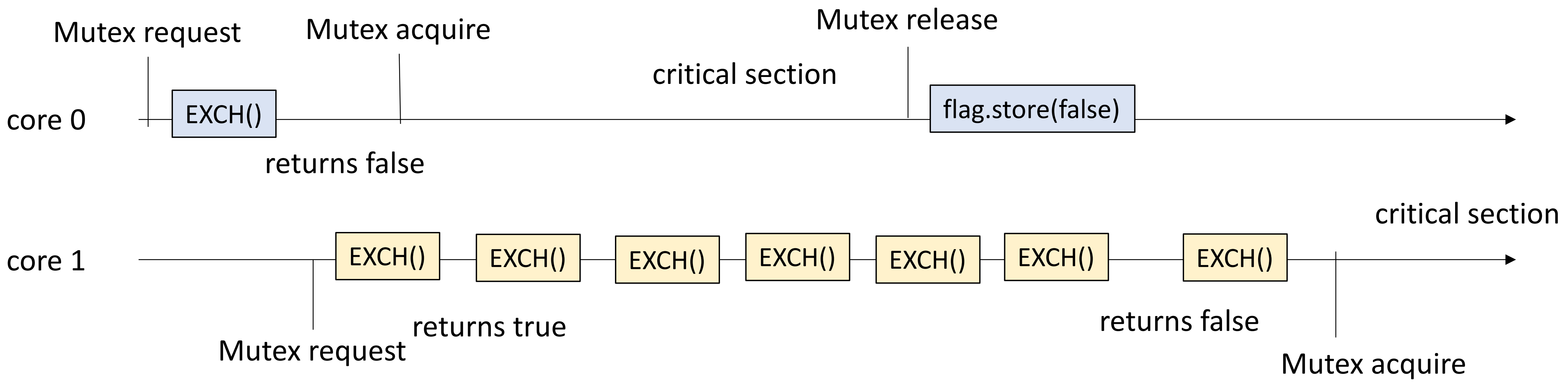
```
void lock() {
  while (atomic_exchange(&flag, true) == true);
}
```



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

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

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



Analysis

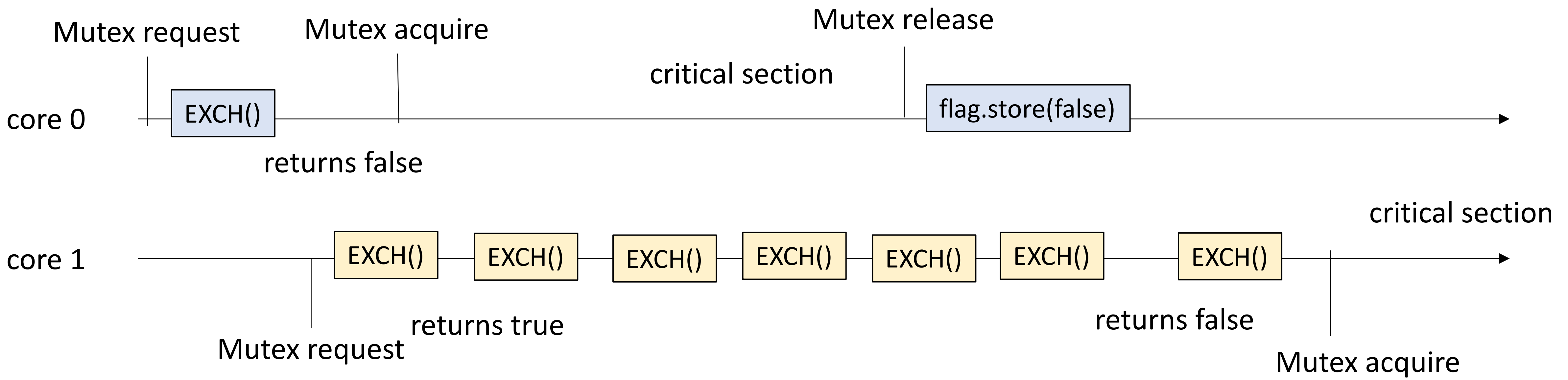
```
void lock() {
  while (atomic_exchange(&flag, true) == true);
}
```

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

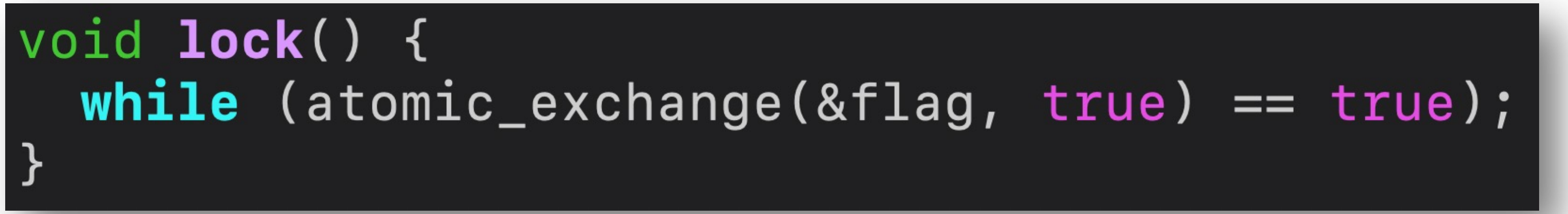
Thread 0:
```
m.lock();
m.unlock();
```

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

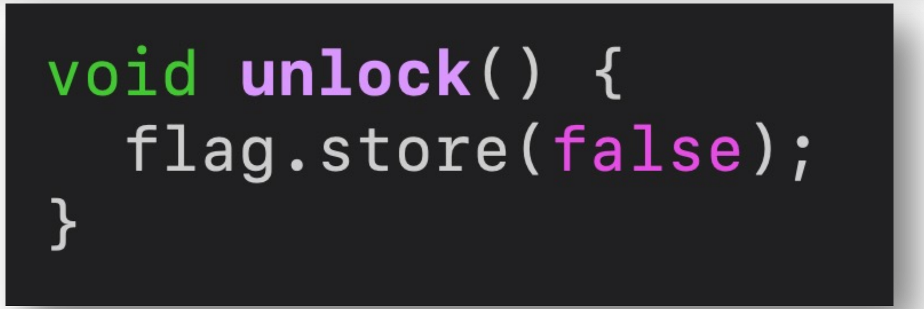
what about interleavings?



Analysis



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

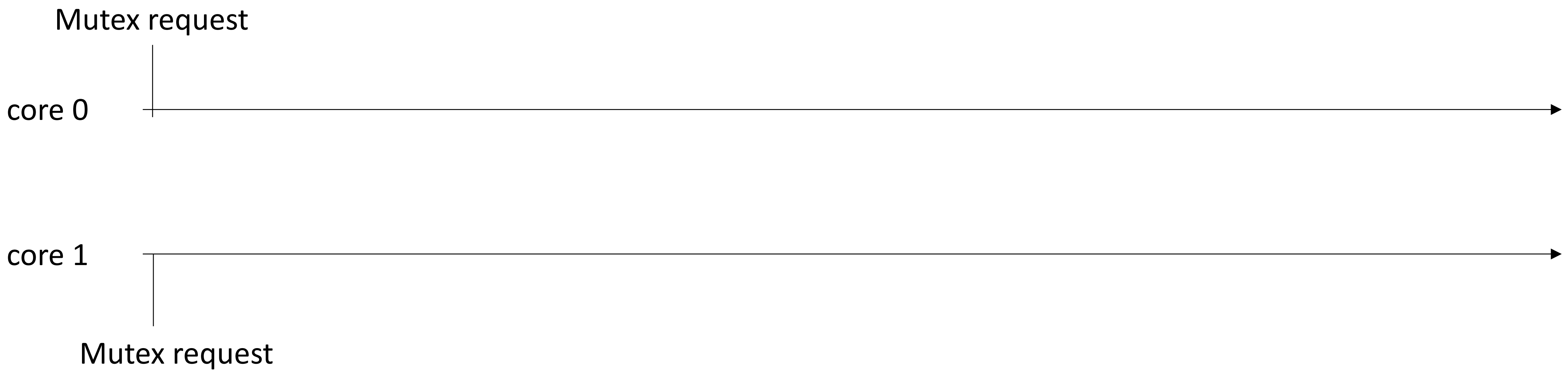


```
void unlock() {
  flag.store(false);
}
```

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

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

what about interleavings?



Analysis

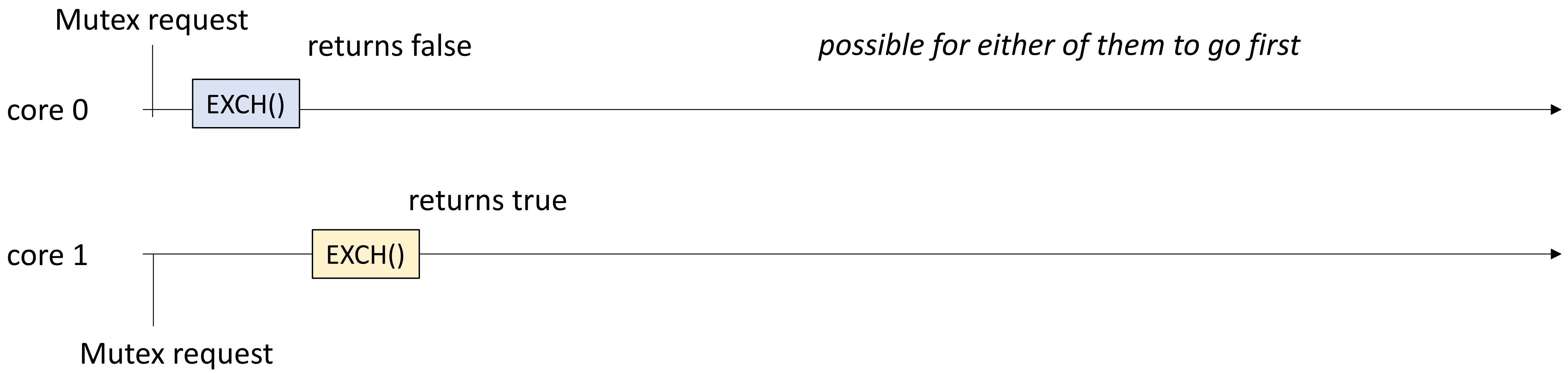
```
void lock() {
  while (atomic_exchange(&flag, true) == true);
}
```

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

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

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

recall RMWS can't overlap!



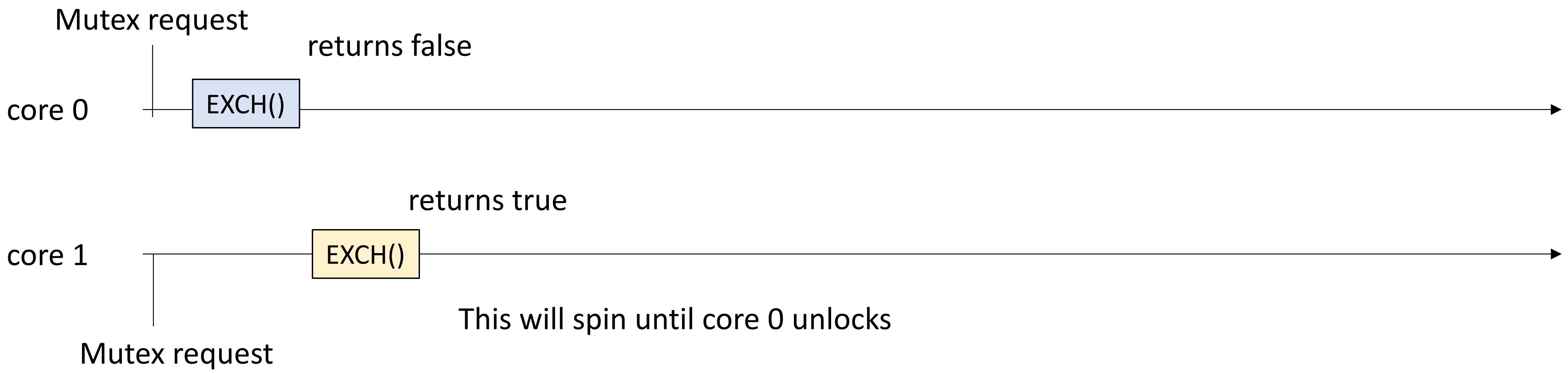
Analysis

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

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

```
Thread 0:          Thread 1:
m.lock();          m.lock();
m.unlock();        m.unlock();
```

recall RMWS can't overlap!



Analysis

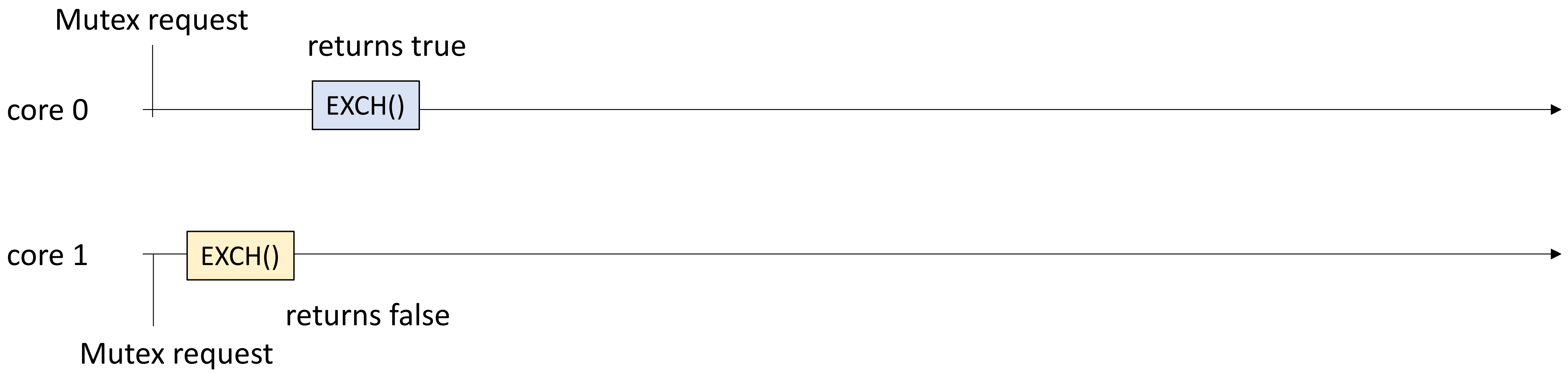
```
void lock() {
  while (atomic_exchange(&flag, true) == true);
}
```

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

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

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

recall RMWS can't overlap!



Analysis

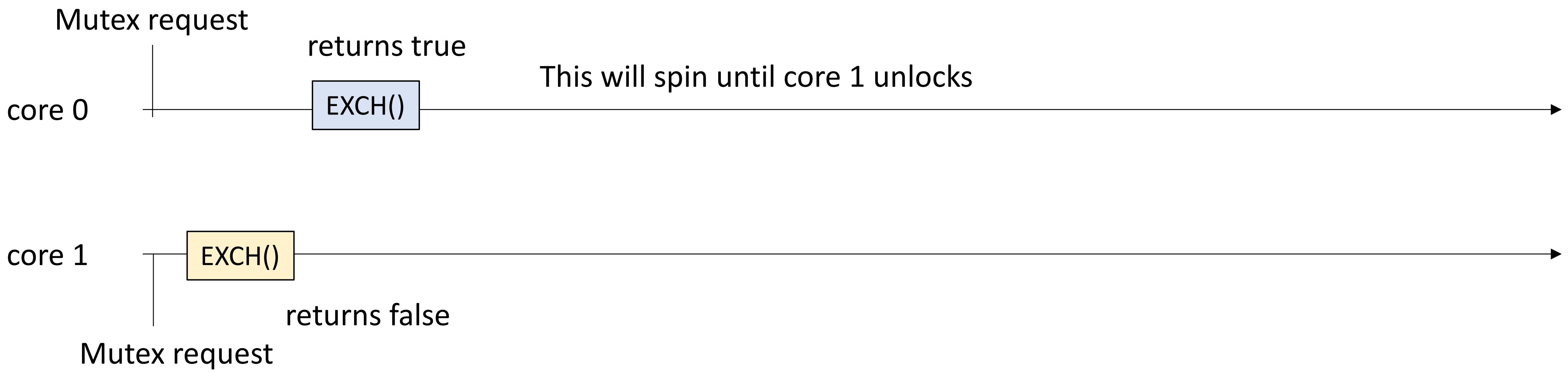
```
void lock() {
  while (atomic_exchange(&flag, true) == true);
}
```

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

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

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

recall RMWS can't overlap!

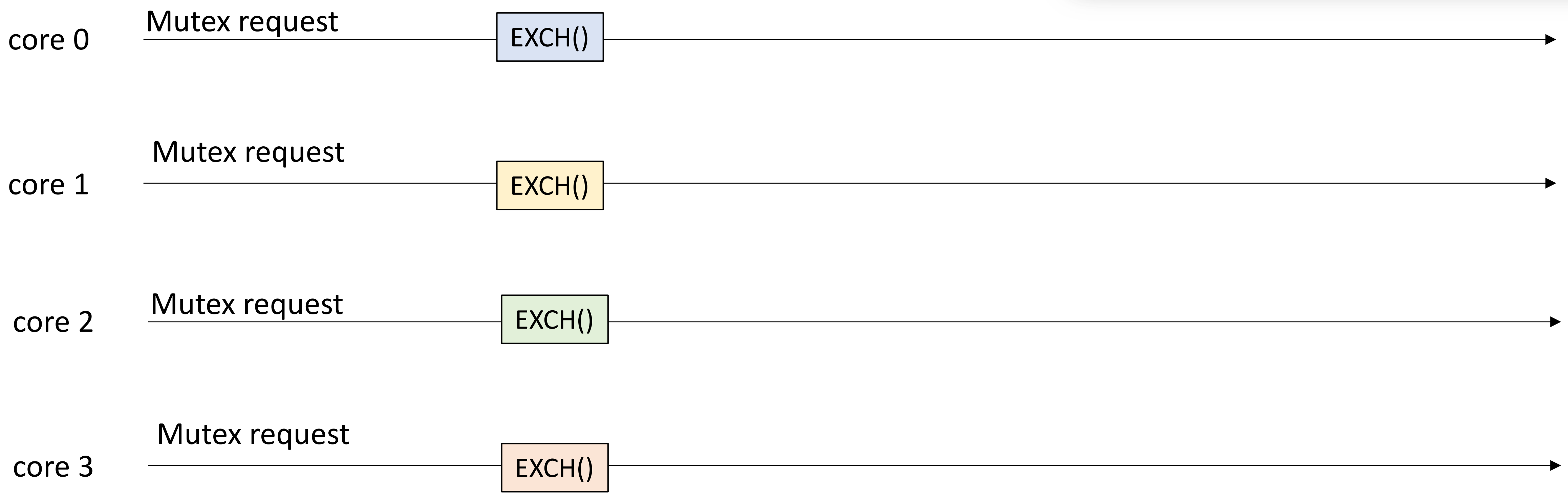


Analysis

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

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

what about 4 threads?



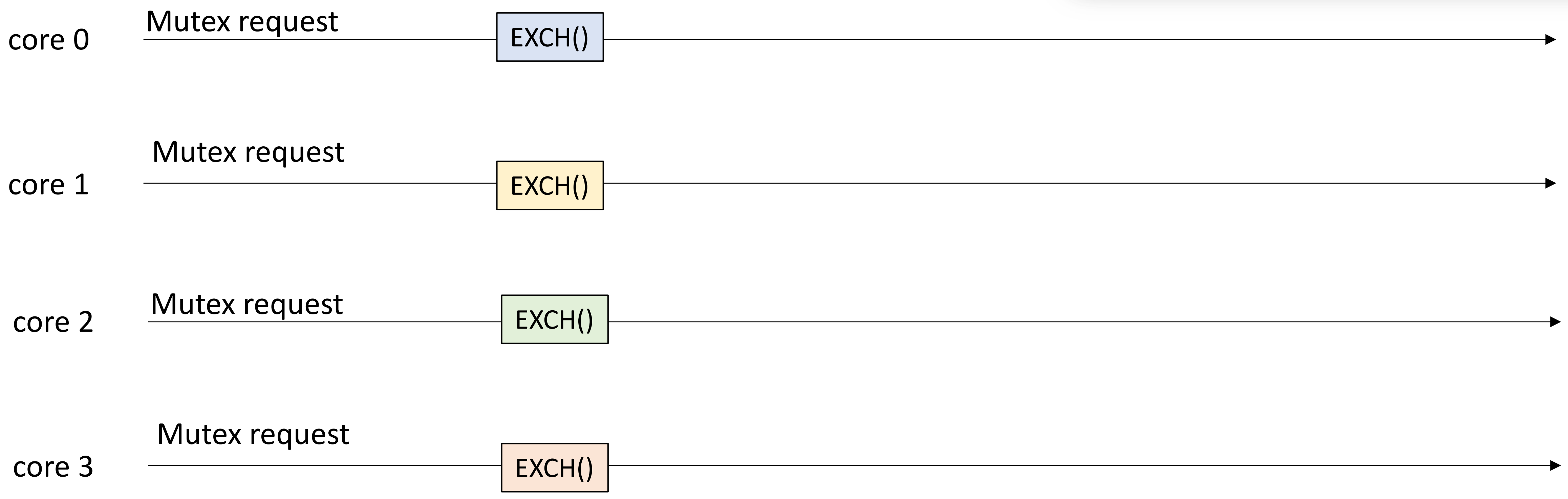
Analysis

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

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

what about 4 threads?

atomic operations can't overlap



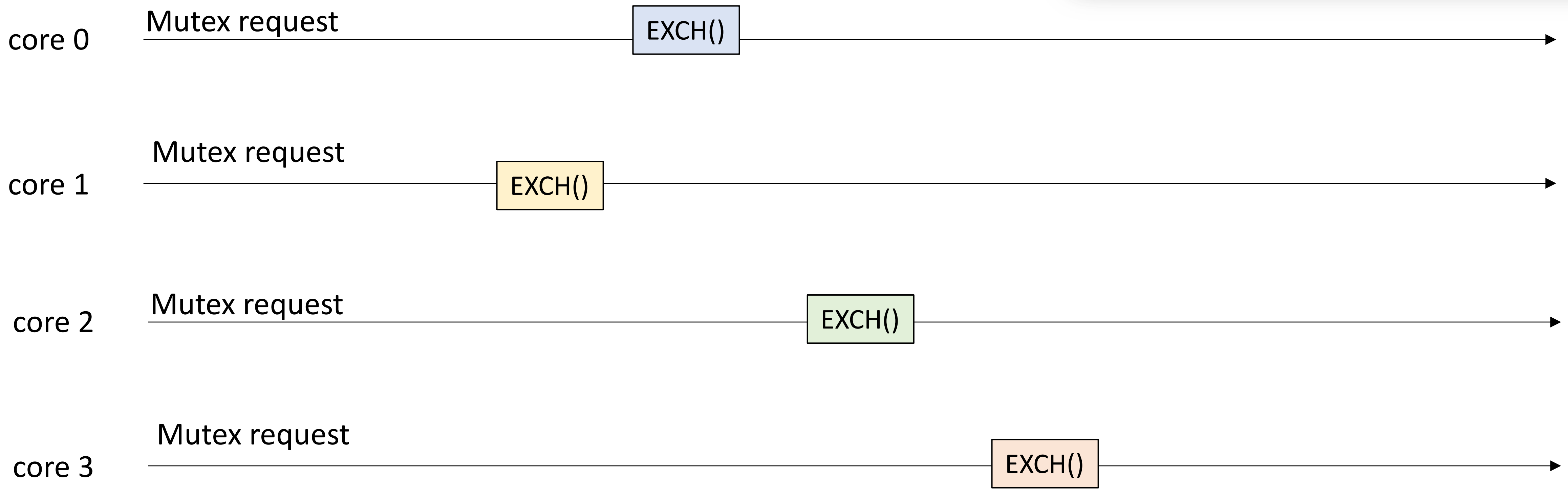
Analysis

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

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

what about 4 threads?

atomic operations can't overlap



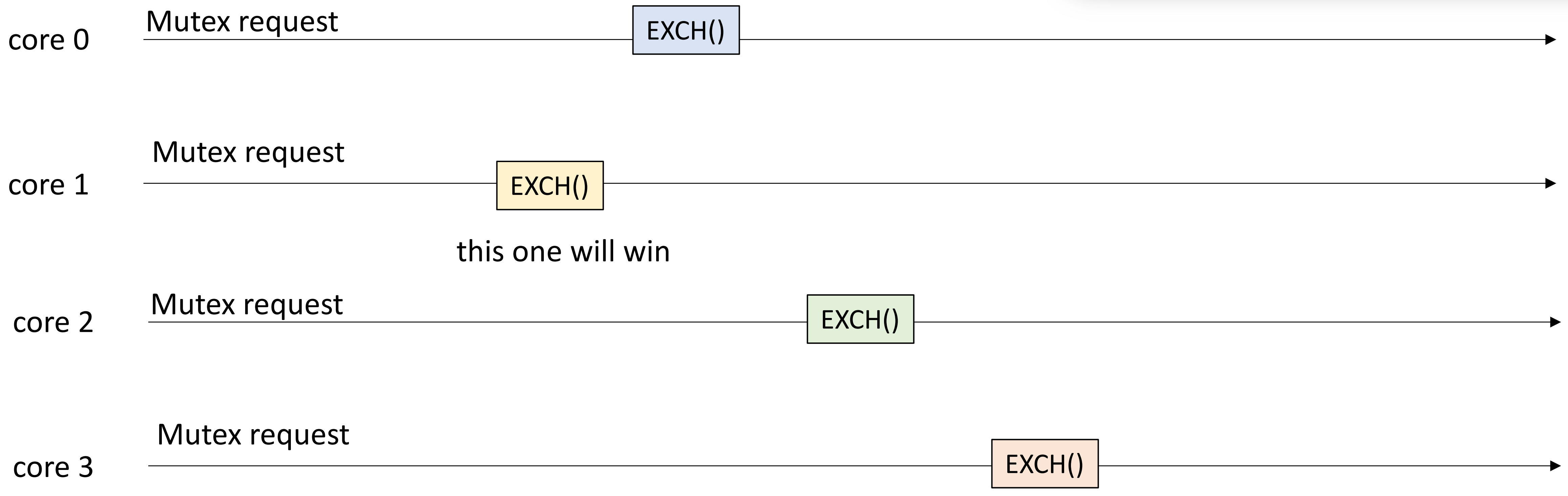
Analysis

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

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

what about 4 threads?

atomic operations can't overlap



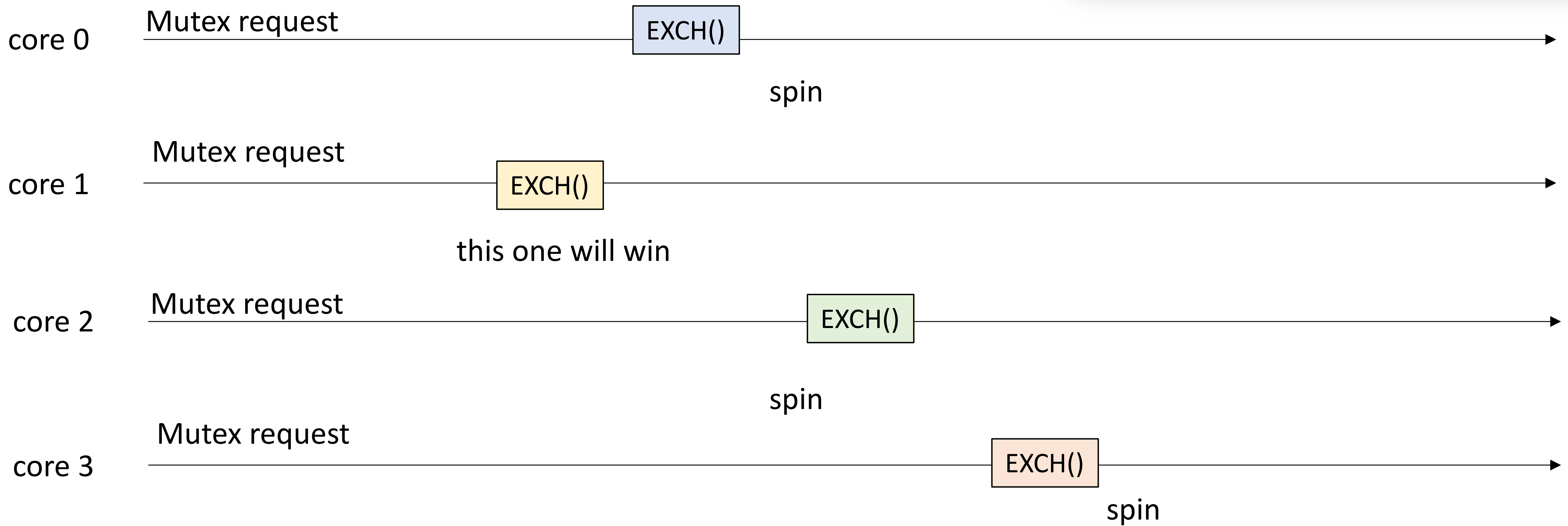
Analysis

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

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

what about 4 threads?

atomic operations can't overlap



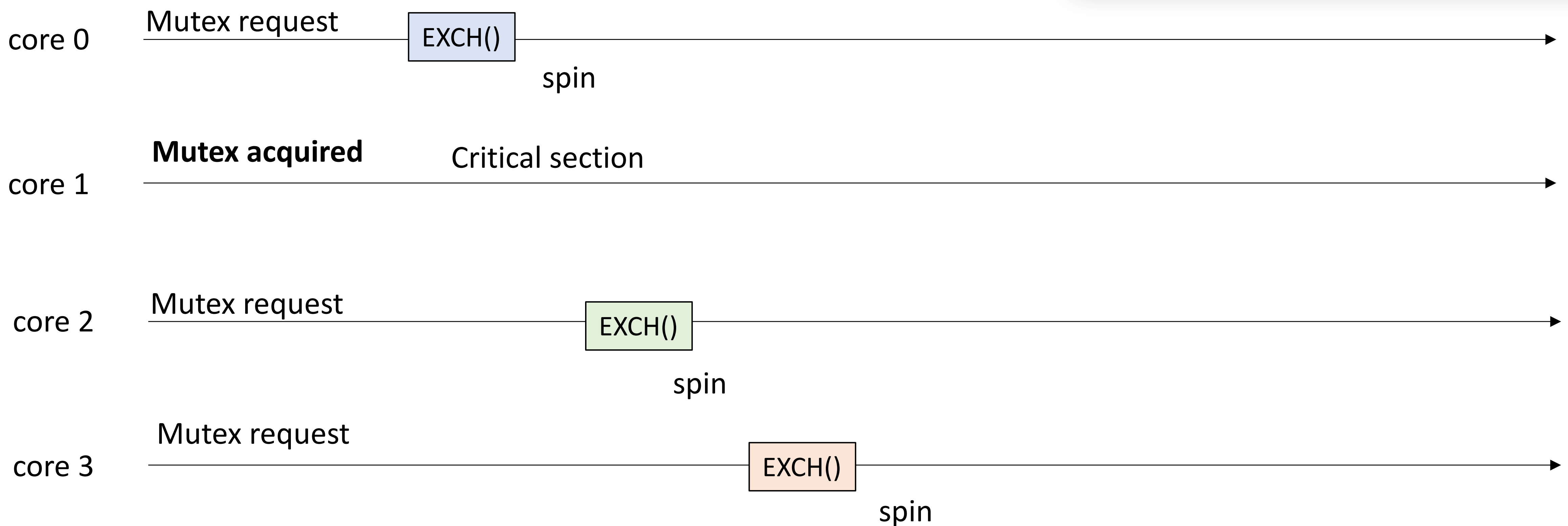
Analysis

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

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

what about 4 threads?

atomic operations can't overlap



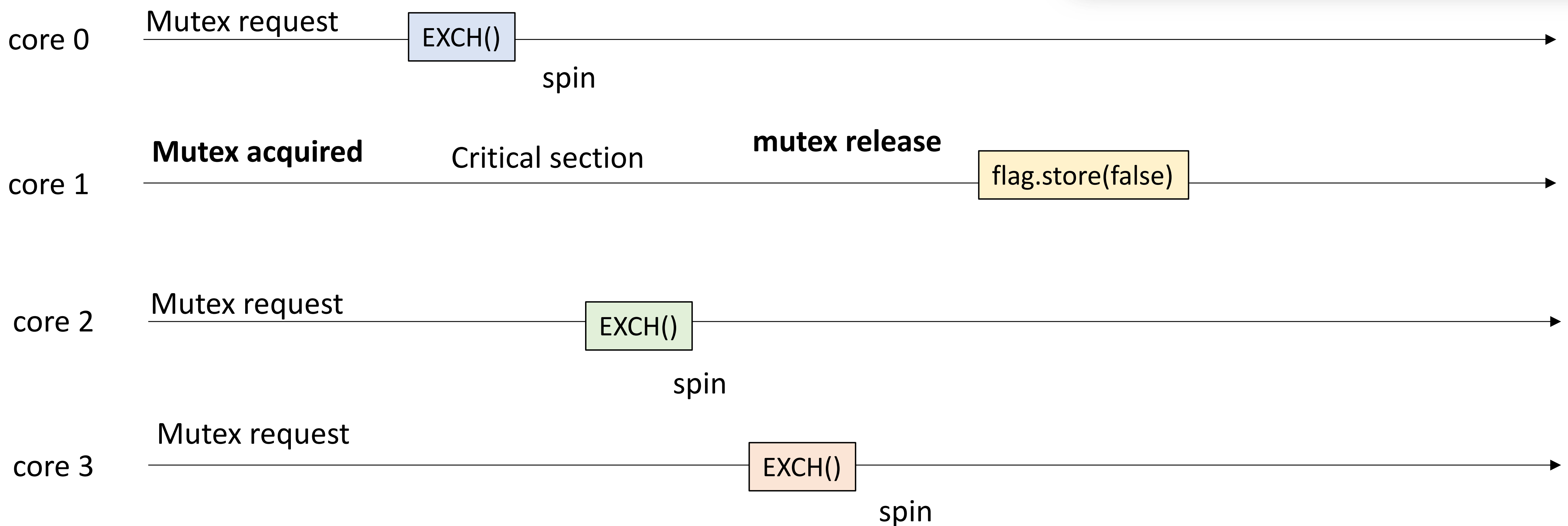
Analysis

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

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

what about 4 threads?

atomic operations can't overlap



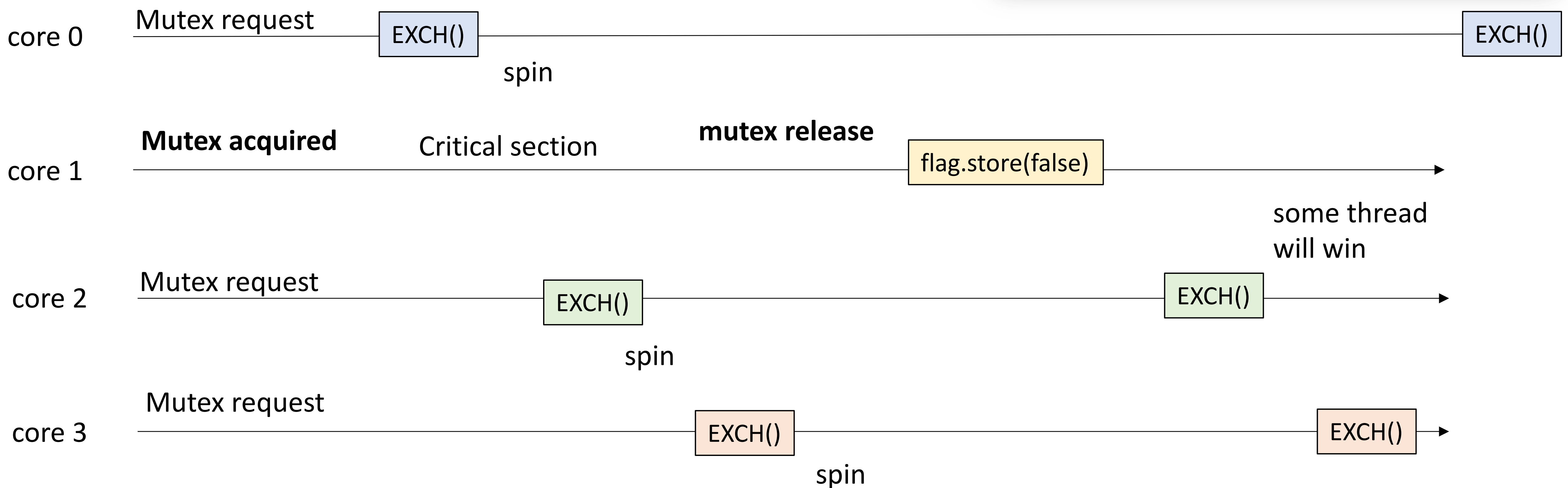
Analysis

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

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

what about 4 threads?

atomic operations can't overlap



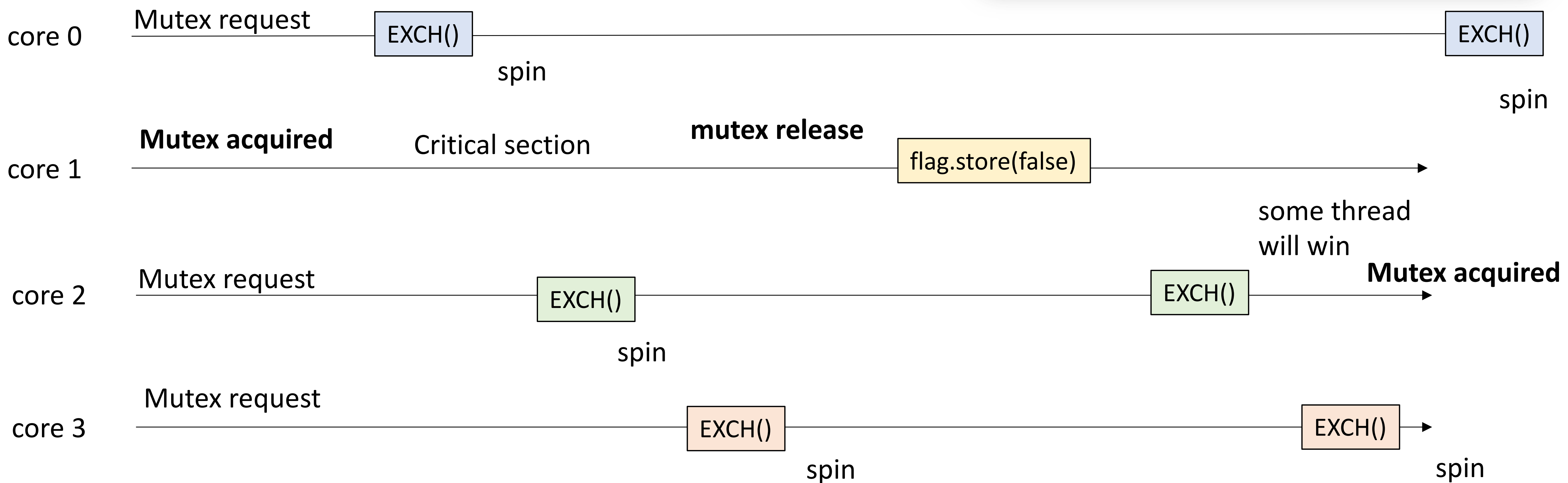
Analysis

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

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

what about 4 threads?

atomic operations can't overlap



First example: Exchange Mutex

- Questions?

Most versatile RMW: Compare-and-swap

- Exchange was the simplest RMW (no modify)
- Most versatile RMW: Compare-and-swap (CAS)

```
bool atomic_compare_exchange_strong(atomic *a, value *expected, value replace);
```

Most versatile RMW: Compare-and-swap

- Exchange was the simplest RMW (no modify)
- Most versatile RMW: Compare-and-swap (CAS)

```
bool atomic_compare_exchange_strong(atomic *a, value *expected, value replace);
```

Checks if value at `a` is equal to the value at `expected`. If it is equal, swap with `replace`. returns `True` if the values were equal. `False` otherwise.

Most versatile RMW: Compare-and-swap

- Exchange was the simplest RMW (no modify)

- Most versatile RMW: Compare-and-swap (CAS)

```
bool atomic_compare_exchange_strong(atomic *a, value *expected, value replace);
```

Checks if value at `a` is equal to the value at `expected`. If it is equal, swap with `replace`.
returns `True` if the values were equal. `False` otherwise.
`expected` is passed by reference: the previous value at a is returned

Most versatile RMW: Compare-and-swap

- Exchange was the simplest RMW (no modify)

- Most versatile RMW: Compare-and-swap (CAS)

```
bool atomic_compare_exchange_strong(atomic *a, value *expected, value replace) {
   value tmp = a.load();
   if (tmp == *expected) {
     a.store(replace);
     return true;
   }
   *expected = tmp;
   return false;
}
```

Most versatile RMW: Compare-and-swap

- Exchange was the simplest RMW (no modify)

we will discuss this soon!

- Most versatile RMW: Compare-and-swap (CAS)

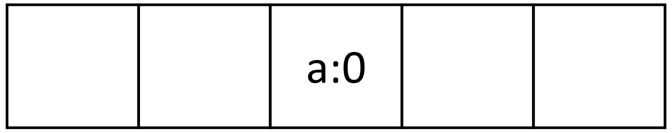
```
bool atomic_compare_exchange_strong(atomic *a, value *expected, value replace) {
   value tmp = a.load();
   if (tmp == *expected) {
     a.store(replace);
     return true;
   }
   *expected = tmp;
   return false;
}
```

Most versatile RMW: Compare-and-swap

```
bool atomic_compare_exchange_strong(atomic *a, value *expected, value replace) {
   value tmp = a.load();
   if (tmp == *expected) {
     a.store(replace);
     return true;
   }
   *expected = tmp;
   return false;
}
```

thread 0:

```
// some atomic int address a
int e = 0;
bool s = atomic_CAS(a,&e,6);
```

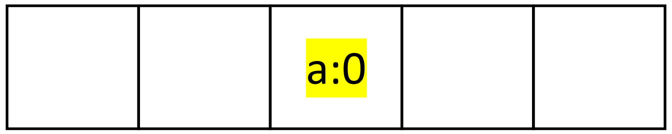


Most versatile RMW: Compare-and-swap

```
bool atomic_compare_exchange_strong(atomic *a, value *expected, value replace) {
   value tmp = a.load();
   if (tmp == *expected) {
     a.store(replace);
     return true;
   }
   *expected = tmp;
   return false;
}
```

thread 0:

```
// some atomic int address a
int e = 0;
bool s = atomic_CAS(a,&e,6);
```

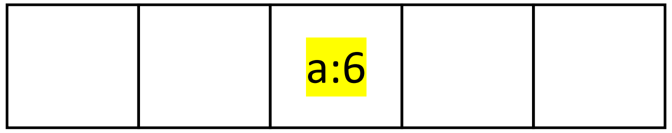


Most versatile RMW: Compare-and-swap

```
bool atomic_compare_exchange_strong(atomic *a, value *expected, value replace) {
   value tmp = a.load();
   if (tmp == *expected) {
     a.store(replace);
     return true;
   }
   *expected = tmp;
   return false;
}
```

<u>thread 0:</u>

```
// some atomic int address a
int e = 0;
bool s = atomic_CAS(a,&e,6);
```



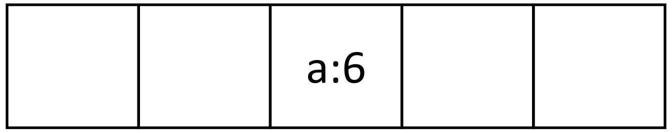
Most versatile RMW: Compare-and-swap

```
bool atomic_compare_exchange_strong(atomic *a, value *expected, value replace) {
   value tmp = a.load();
   if (tmp == *expected) {
     a.store(replace);
     return true;
   }
   *expected = tmp;
   return false;
}
```

thread 0:

```
// some atomic int address a
int e = 0;
bool s = atomic_CAS(a,&e,6);
```

true



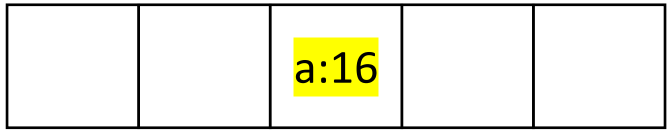
Most versatile RMW: Compare-and-swap

```
bool atomic_compare_exchange_strong(atomic *a, value *expected, value replace) {
   value tmp = a.load();
   if (tmp == *expected) {
     a.store(replace);
     return true;
   }
   *expected = tmp;
   return false;
}
```

next example

thread 0:

```
// some atomic int address a
int e = 0;
bool s = atomic_CAS(a,&e,6);
```

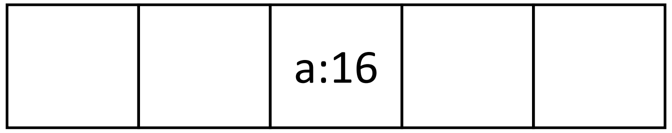


Most versatile RMW: Compare-and-swap

```
bool atomic_compare_exchange_strong(atomic *a, value *expected, value replace) {
   value tmp = a.load();
   if (tmp == *expected) {
     a.store(replace);
     return true;
   }
   *expected = tmp;
   return false;
}
```

thread 0:

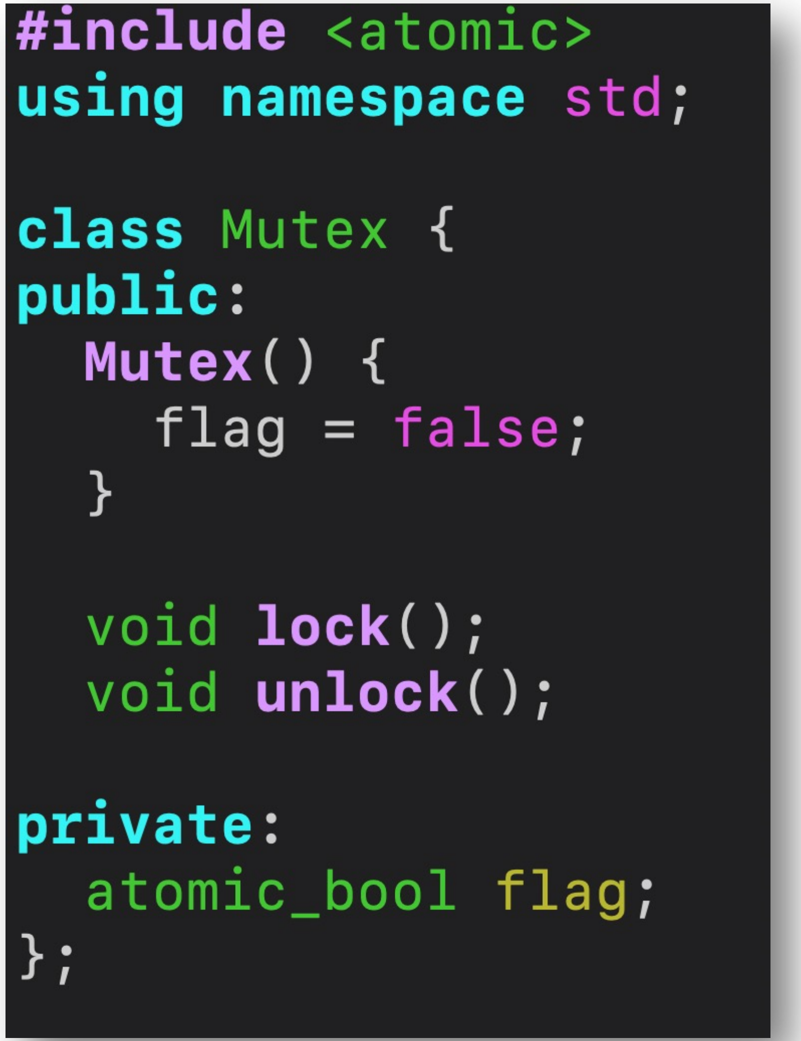
```
// some atomic int address a
int e = 0;
bool s = atomic_CAS(a,&e,6);
```



16

false

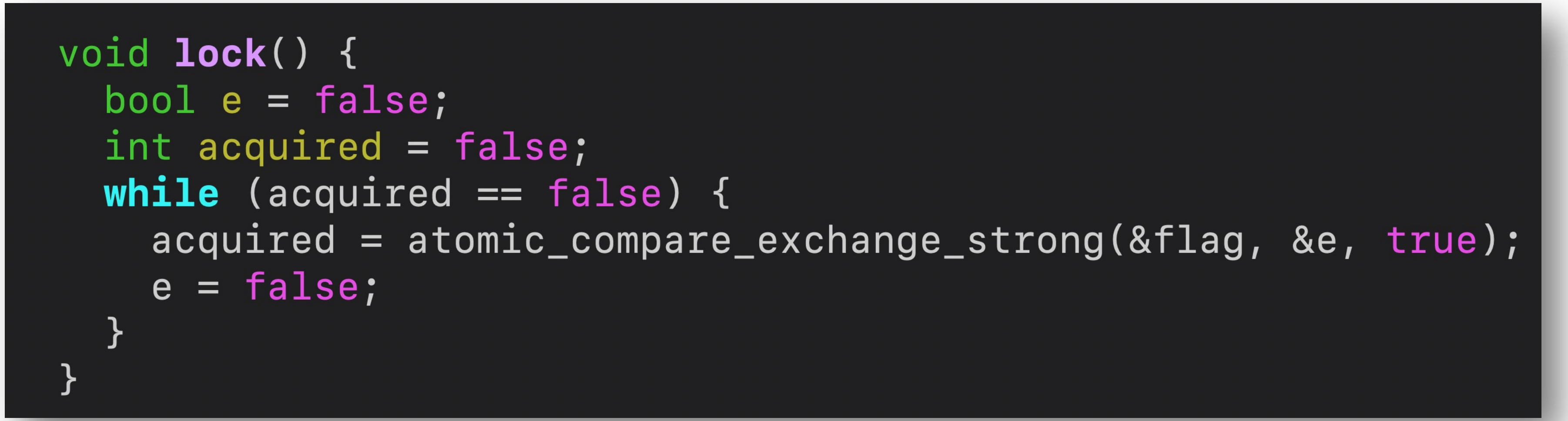
CAS lock



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

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

  void lock();
  void unlock();

private:
  atomic_bool flag;
};
```

Pretty intuitive: only 1 bit required again:

CAS lock

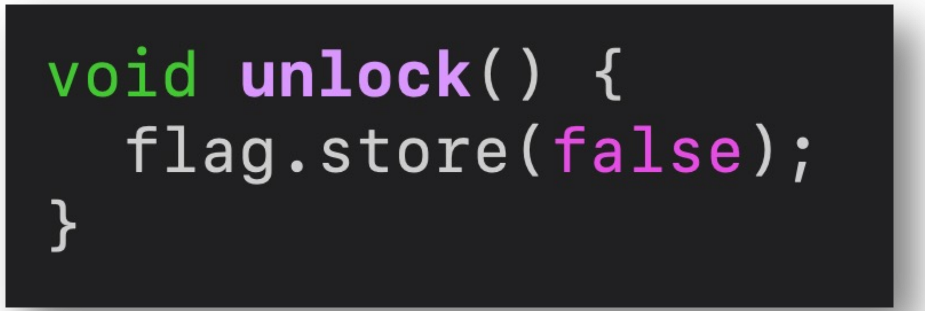


```c
void lock() {
  bool e = false;
  int acquired = false;
  while (acquired == false) {
    acquired = atomic_compare_exchange_strong(&flag, &e, true);
    e = false;
  }
}
```

Check if the mutex is free, if so, take it.

compare the mutex to free (false), if so, replace it with taken (true). Spin while the thread isn’t able to take the mutex.

CAS lock



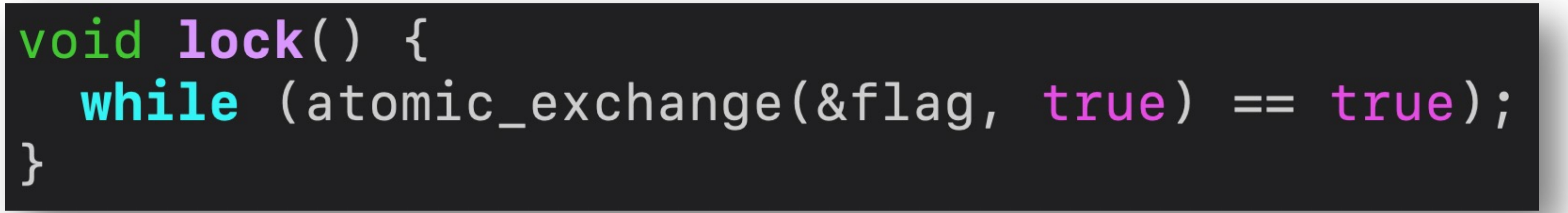
```
void unlock() {
  flag.store(false);
}
```

Unlock is simple! Just store false back

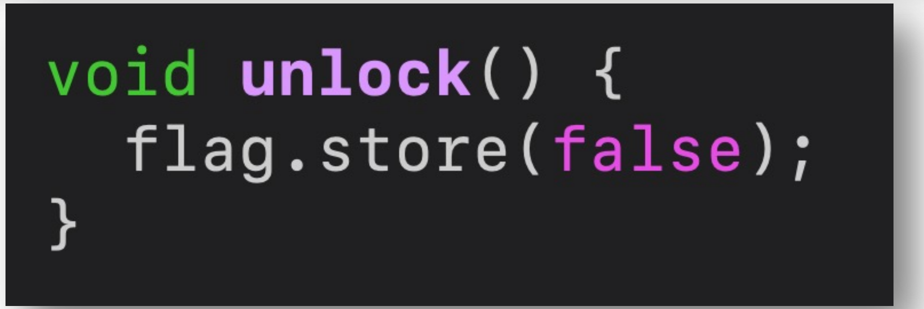
Starvation

- Are these RMW locks fair?

Analysis

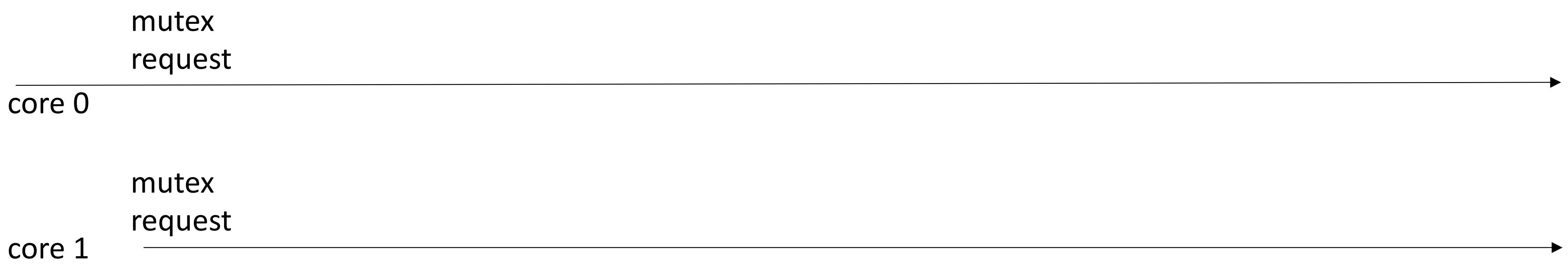


```
void lock() {
  while (atomic_exchange(&flag, true) == true);
}
```



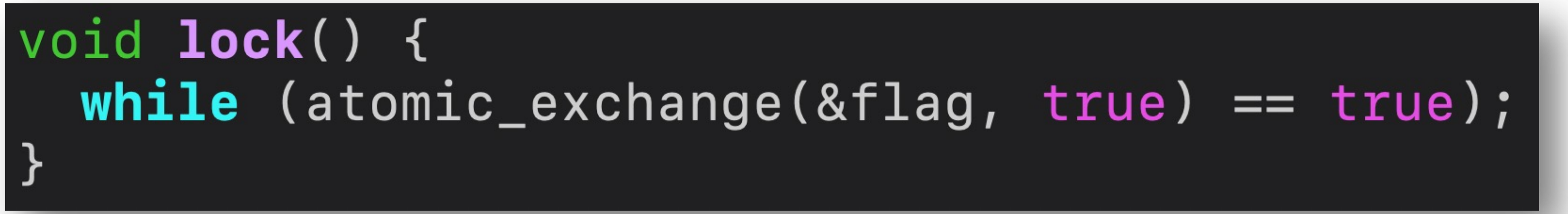
```
void unlock() {
  flag.store(false);
}
```

Is this mutex starvation Free?

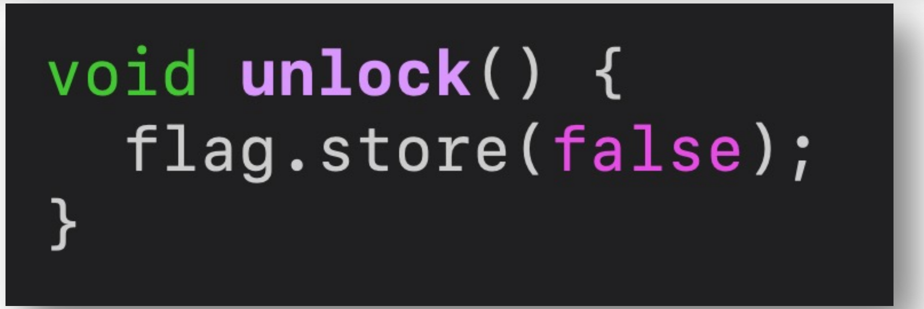


Analysis

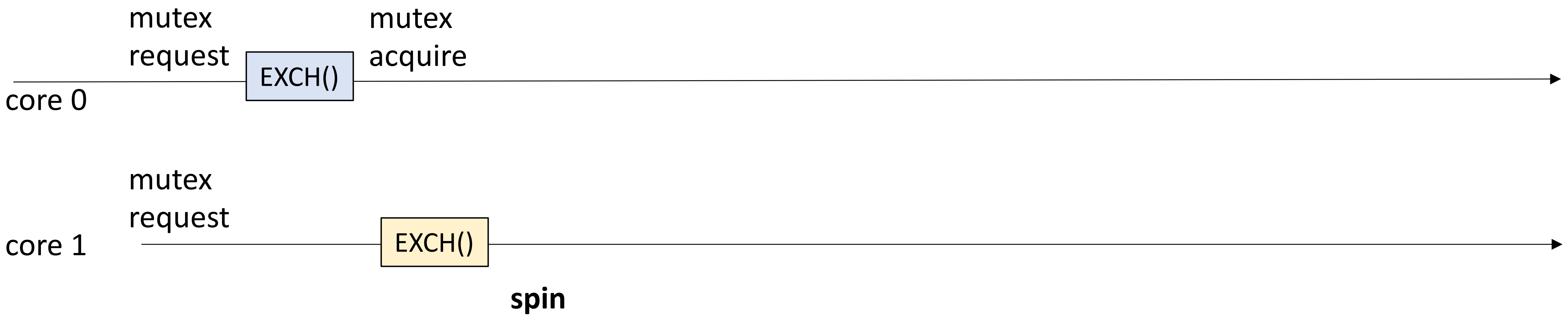
Is this mutex starvation Free?



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



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

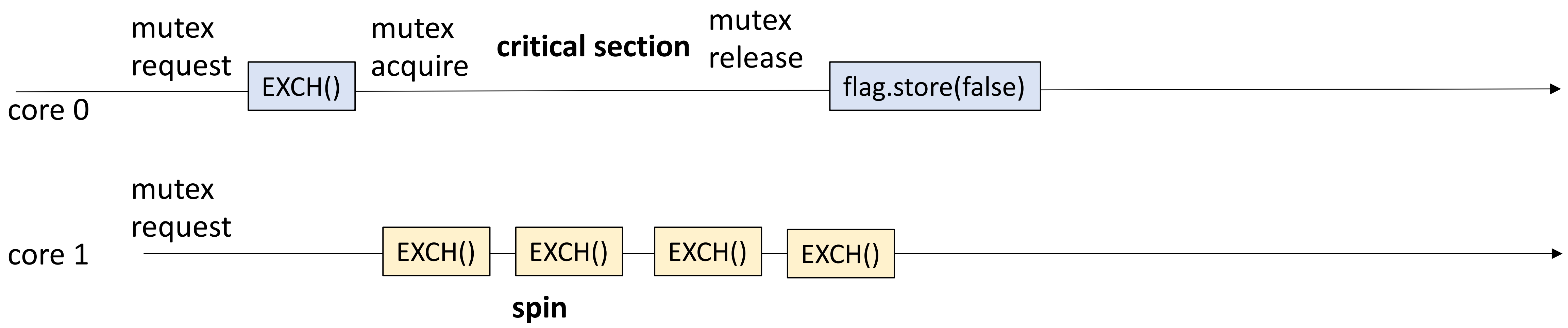


Analysis

Is this mutex starvation Free?

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

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

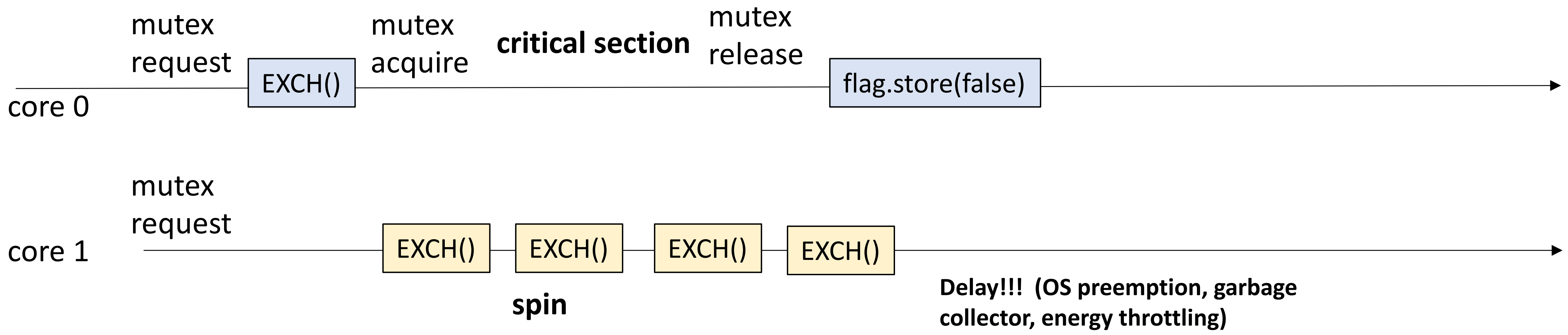


Analysis

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

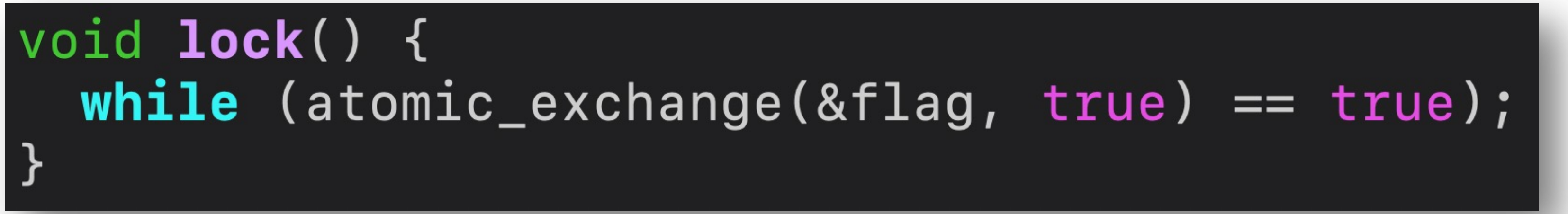
```
void unlock() {
  flag.store(false);
}
```

Is this mutex starvation Free?

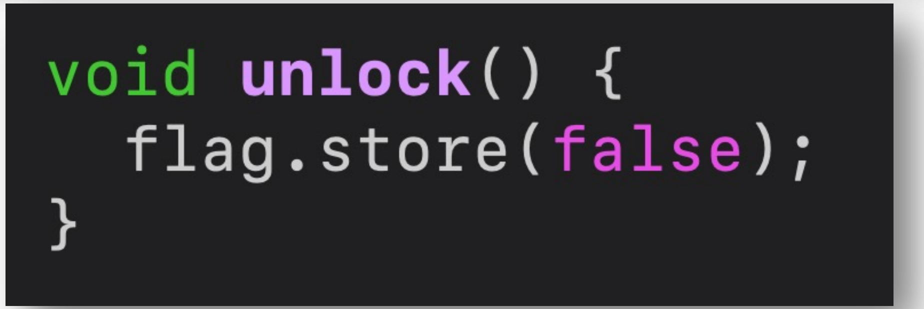


Analysis

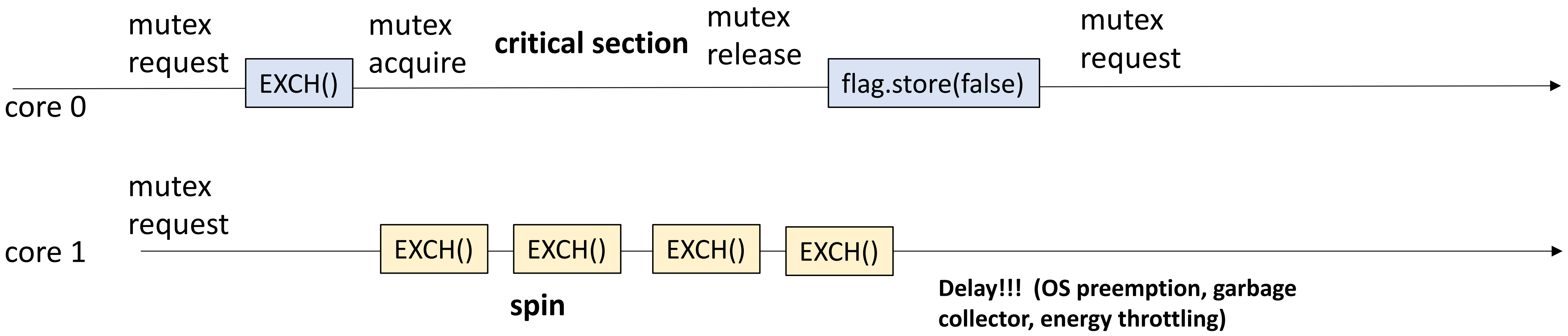
Is this mutex starvation Free?



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



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

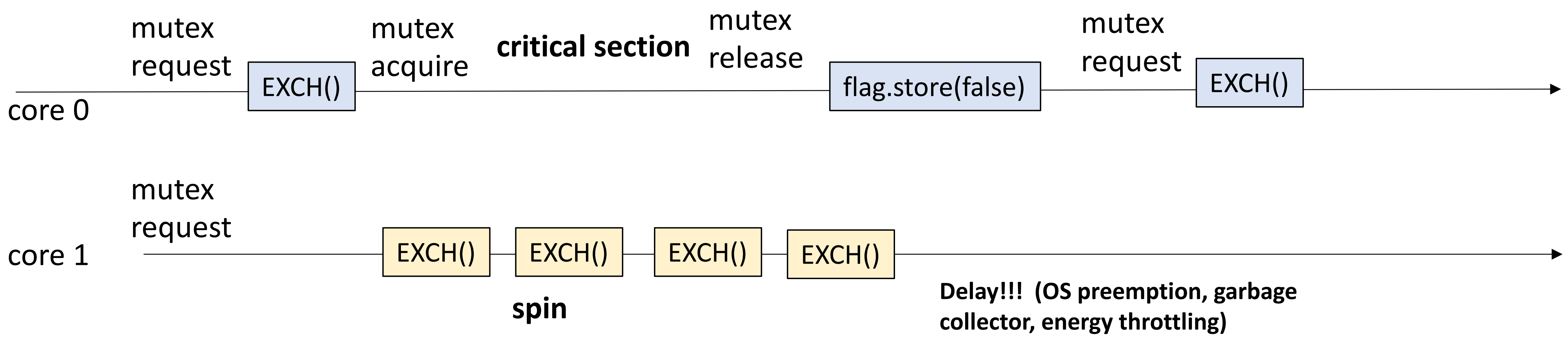


Analysis

Is this mutex starvation Free?

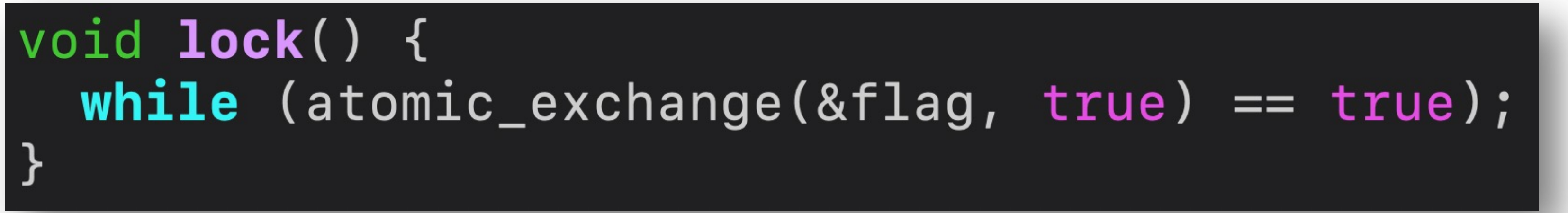
```
void lock() {
  while (atomic_exchange(&flag, true) == true);
}
```

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

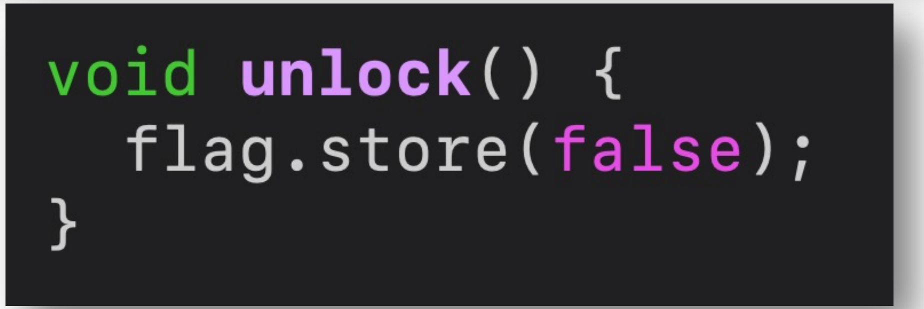


Analysis

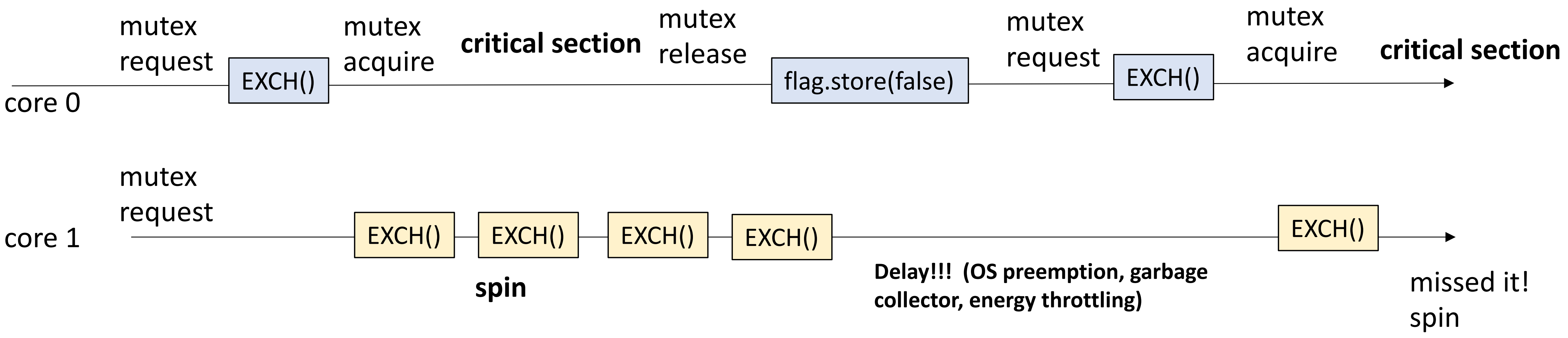
Is this mutex starvation Free?



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



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



How about in practice?

- Code demo

Thanks!

- Next time:
    - A fair RMW lock
    - optimizations (yield)
    - Reader-Writer locks

- Start on HW 2 part 1

- Do the quiz please!