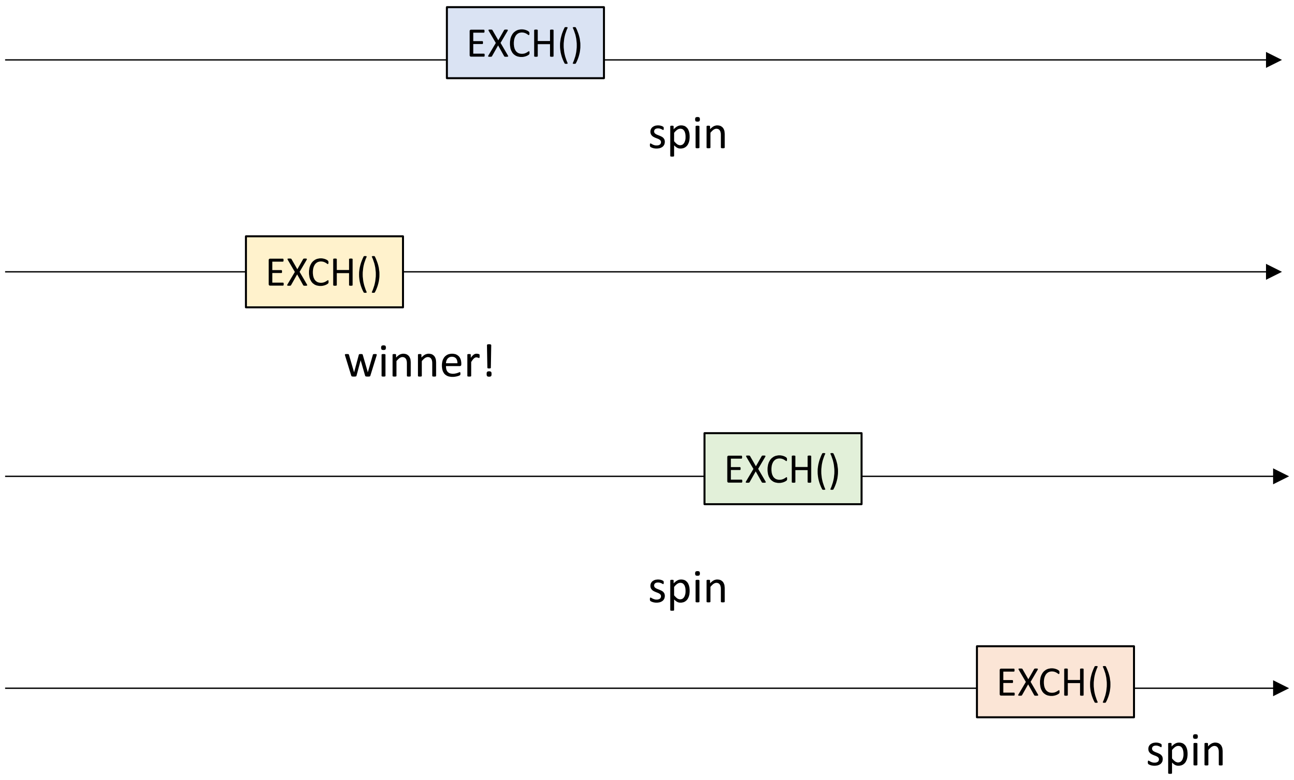
CSE113: Parallel Programming

April 20, 2021

- **Topic**: Practical Mutual Exclusion
  - Atomic RMW locks
  - Optimizing locks



Announcements

- Homework due on Thursday at Midnight
  - Gan set up a submission link on Canvas
  - Questions:
    - Visit a few from Canvas
  - Should Gan do office hours on Thursday instead of Friday?
    - 10 - 11 AM on Thursday

- My office hours are on Wednesday
  - Unfortunately do not have many on Thursday

Announcements

- Next homework:
  - Assigned by midnight Thursday

- Sign up for Piazza:
  - Lots of good discussions
  - Only 50 of you have

- I trust things are going well on Discord...

Announcements

- A little bit out of sync with the book
  - We'll do first half of Chapter 7 today
  - Next lecture will be last of Chapter 7 and some of Chapter 8
  - Still on track to start Module 3 (concurrent data structures) in 1 week

Quiz

- Canvas Quiz
- Peterson’s algorithm is starvation free: true or false
- atomic operations provide mutual exclusion

Quiz

- Canvas Quiz
- Go over answers

Before we start: C++ `lock_guard`

- C++ `lock_guard` - Pretty cool!
- Uses C++ constructor and destructor

recall this snippet of code.
What was the issue?

Tyler's coffee addiction:

```
m.lock();
tylers_account -= 1;
if (tylers_account < -100) {
  printf("warning!\n");
  return;
}
m.unlock();
return;
```

Before we start: C++ `lock_guard`

- C++ `lock_guard` - Pretty cool!
- Uses C++ constructor and destructor

recall this snippet of code.
What was the issue?

We need to unlock
at two places

Tyler's coffee addiction:

```
m.lock();
tylers_account -= 1;
if (tylers_account < -100) {
  printf("warning!\n");
  m.unlock();
  return;
}
m.unlock();
return;
```

Before we start: C++ `lock_guard`

- C++ `lock_guard` - Pretty cool!
- Uses C++ constructor and destructor

recall this snippet of code.
What was the issue?

We need to unlock
at two places

Tyler's coffee addiction:

```
m.lock();
tylers_account -= 1;
if (tylers_account < -100) {
  printf("warning!\n");
  m.unlock();
  return;
}
m.unlock();
return;
```

Before we start: C++ `lock_guard`

- C++ `lock_guard` - Pretty cool!
- Uses C++ constructor and destructor

recall this snippet of code.
What was the issue?

Or we use lock_guard

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

```
lock_guard<mutex> lck(m);
tylers_account -= 1;
if (tylers_account < -100) {
  printf("warning!\n");
  return;
}
return;
```

Before we start: C++ `lock_guard`

- C++ `lock_guard` - Pretty cool!
- Uses C++ constructor and destructor

A template on a mutex type (i.e. implements lock and unlock).

Tyler's coffee addiction:

```
lock_guard<mutex> lck(m);
tylers_account -= 1;
if (tylers_account < -100) {
  printf("warning!\n");
  return;
}
return;
```

recall this snippet of code.
What was the issue?

Or we use lock_guard

Before we start: C++ `lock_guard`

A template on a mutex type (i.e. implements lock and unlock).

- C++ `lock_guard` - Pretty cool!
- Uses C++ constructor and destructor

Tyler's coffee addiction:

```
lock_guard<mutex> lck(m);
tylers_account -= 1;
if (tylers_account < -100) {
  printf("warning!\n");
  return;
}
return;
```

recall this snippet of code.
What was the issue?

Or we use lock_guard

pass the mutex into the constructor.
No other methods!

Before we start: C++ `lock_guard`

- C++ `lock_guard` - Pretty cool!
- Uses C++ constructor and destructor

A template on a mutex type (i.e. implements lock and unlock).

Tyler's coffee addiction:

```
lock_guard<mutex> lck(m);
tylers_account -= 1;
if (tylers_account < -100) {
  printf("warning!\n");
  return;
}
return;
```

recall this snippet of code. What was the issue?

Or we use lock_guard

Constructor locks the mutex, destructor unlocks it.

pass the mutex into the constructor. No other methods!

Before we start: C++ `lock_guard`

- C++ `lock_guard` - Pretty cool!
- Uses C++ constructor and destructor

Writing to files can throw exceptions, if we don't handle the exception, then the system could deadlock.

Tyler's coffee addiction:

```
lock_guard<mutex> lck(m);
tylers_account -= 1;
if (tylers_account < -100) {
  printf("warning!\n");
  log_warning_to_file();
  return;
}
return;
```

recall this snippet of code.
What was the issue?

Or we use lock_guard

Constructor locks the mutex, destructor unlocks it.

pass the mutex into the constructor.
No other methods!

Lecture Schedule

- Atomic RMW mutexes
  - Exchange
  - CAS
  - Ticket

- Optimizations
  - Relaxed peeking
  - Backoff

Lecture Schedule

- **Atomic RMW mutexes**
  - Exchange
  - CAS
  - Ticket

- Optimizations
  - Relaxed peeking
  - Backoff

From previous lecture: Peterson’s mutex

- Peterson’s algorithm: 2 threaded mutex implementation
  - 2 flag values
  - 1 victim
- We used primitives: atomic loads and stores
- Proof in book:
  - If you only use loads and stores, synchronizing N threads requires O(N) memory

From previous lecture: Peterson's mutex

- Peterson's algorithm: 2 threaded mutex implementation
  - 2 flag values
  - 1 victim

- We used primitives: atomic loads and stores

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

  void lock();
  void unlock();


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

From previous lecture: Peterson's mutex

- Peterson's algorithm: 2 threaded mutex implementation
  - 2 flag values
  - 1 victim

- We used primitives: atomic loads and stores

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

  void lock();
  void unlock();


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

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

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

From previous lecture: Peterson’s mutex

- Peterson’s algorithm
  - 2 threads
  - 2 flag values

- Generalizations:
  - Filter lock - N threaded Peterson’s algorithm (uses 2*N memory)
  - Bakery lock - N threaded fair mutex (uses 2*N memory)
  - Implementations in the Book! Chapter 3

From previous lecture: Peterson’s mutex

- Peterson’s algorithm
  - 2 threads
  - 2 flag values

- Implementing Peterson’s was difficult because of loads/stores interleaving!

From previous lecture: Peterson's mutex

- Peterson's algorithm
  - 2 threads
  - 2 flag values

- Implementing Peterson's was difficult because of loads/stores interleaving!

- But what if there was another way...

Buggy Mutex implementation

```
#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

Buggy Mutex implementation

```
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

Buggy Mutex implementation

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

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

Buggy Mutex implementation: Analysis

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

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

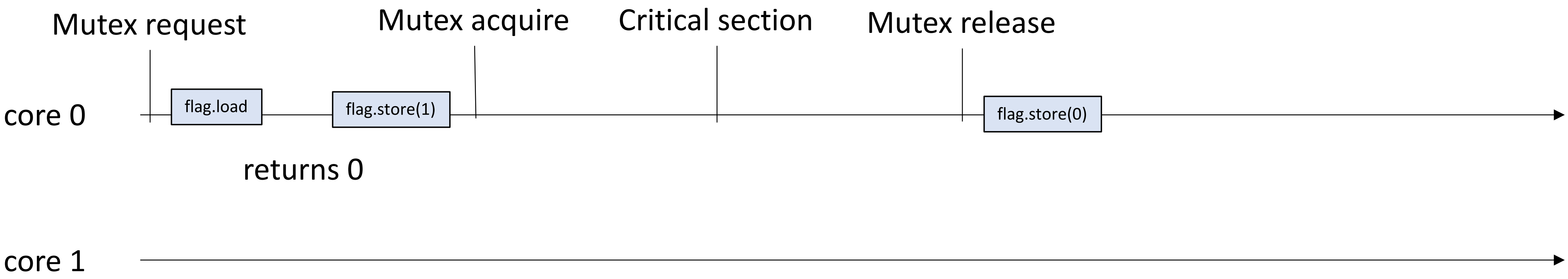
Thread 0:

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

Thread 1:

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

Lets try another interleaving



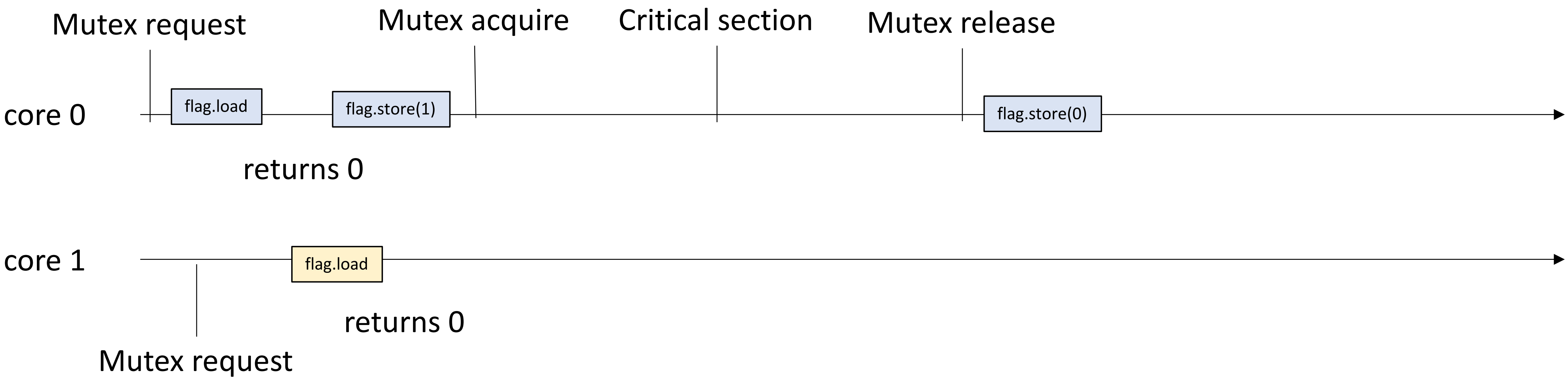
Buggy Mutex implementation: Analysis

```
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();
```



Buggy Mutex implementation: Analysis

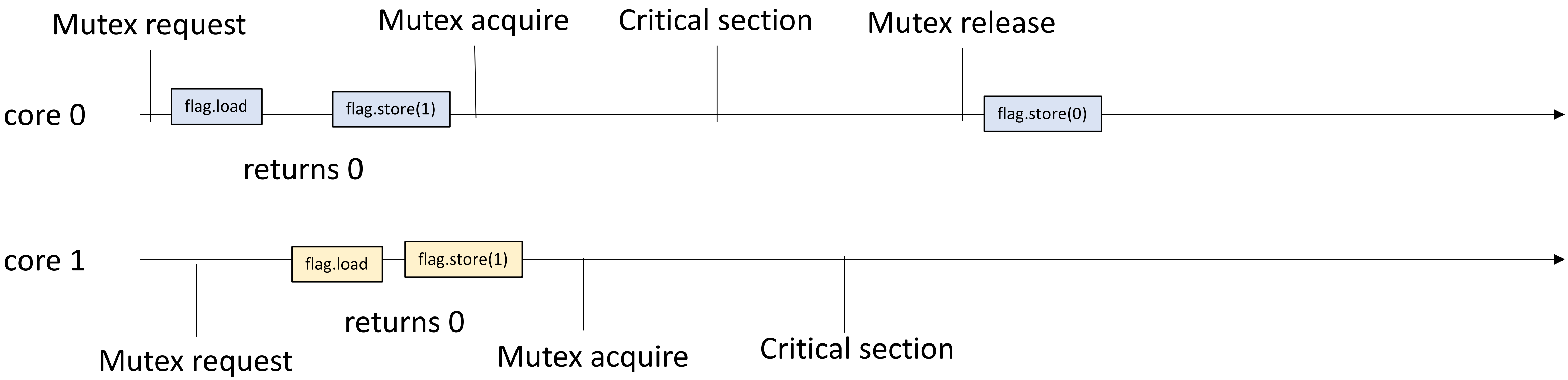
```
void lock() {
  while (flag.load() == 1);
  flag.store(1);
}
```

```
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

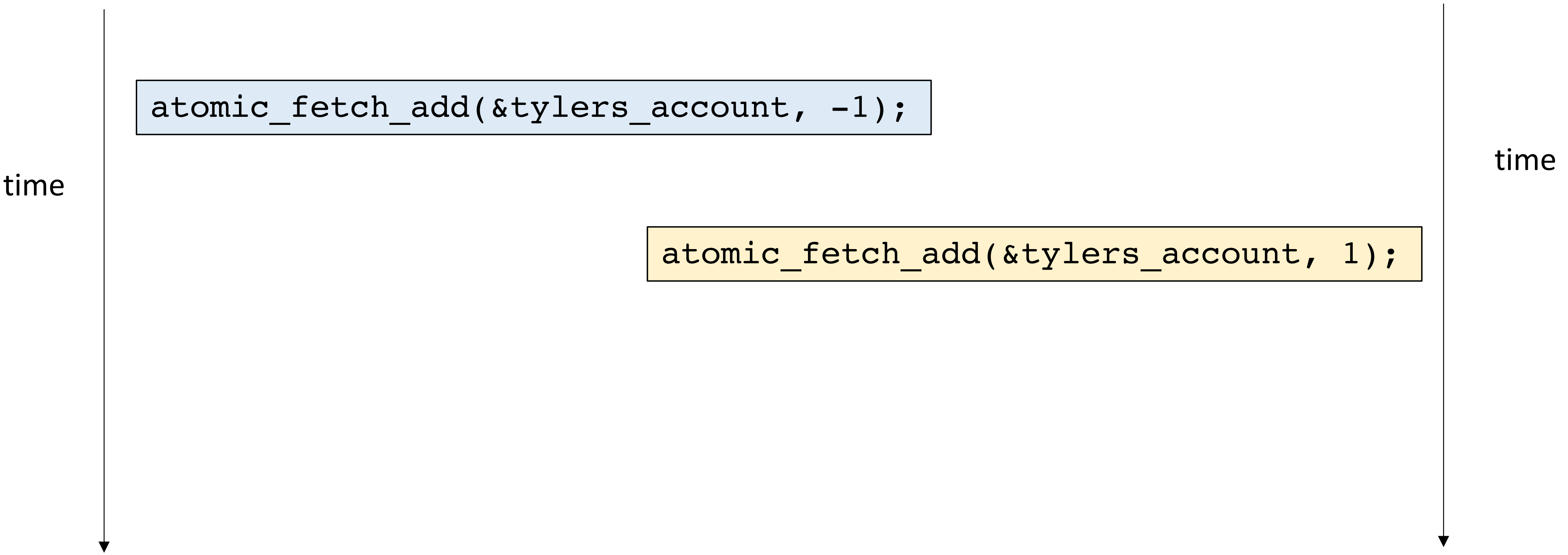
lock-free accounts

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);
```

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

time

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
- When we have simpler implementations, we can focus on performance

Lecture Schedule

- **Atomic RMW mutexes**
  - **Exchange**
  - CAS
  - Ticket

- Optimizations
  - Relaxed peeking
  - Backoff

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

```cpp
#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);
}
```

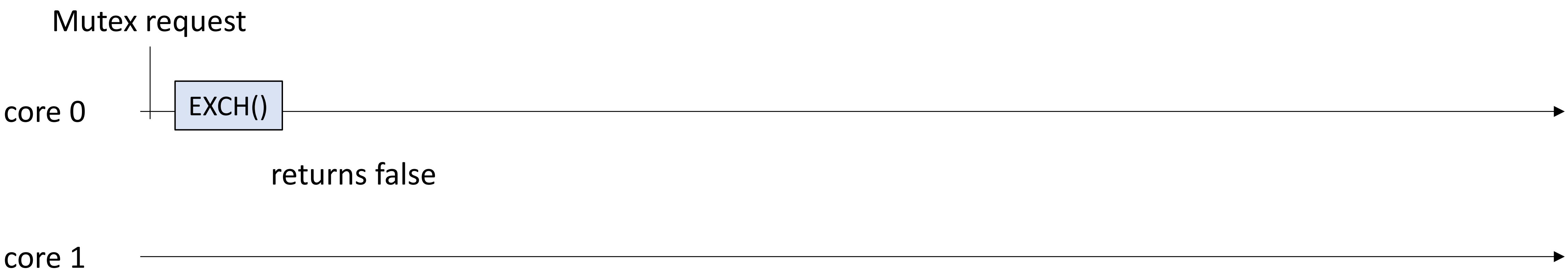
```
void unlock() {
  flag.store(false);
}
```

Thread 0:

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

Thread 1:

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



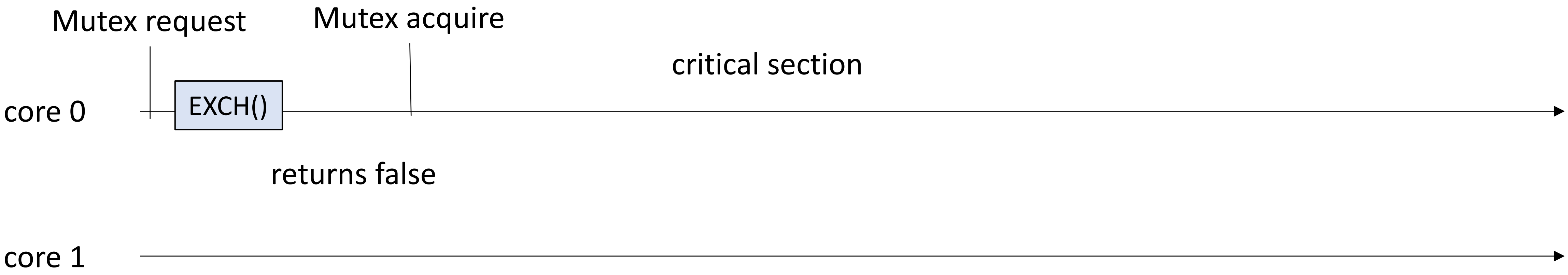
Analysis

```
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();
```



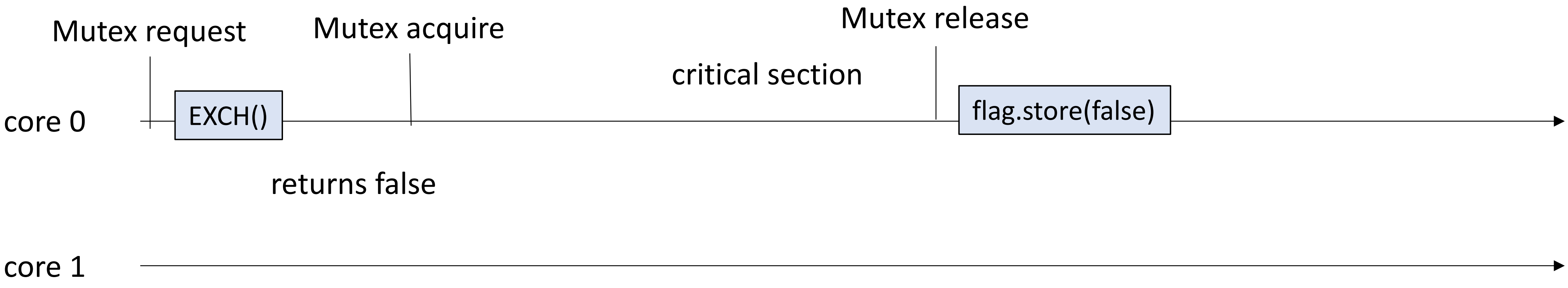
Analysis

```
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();`



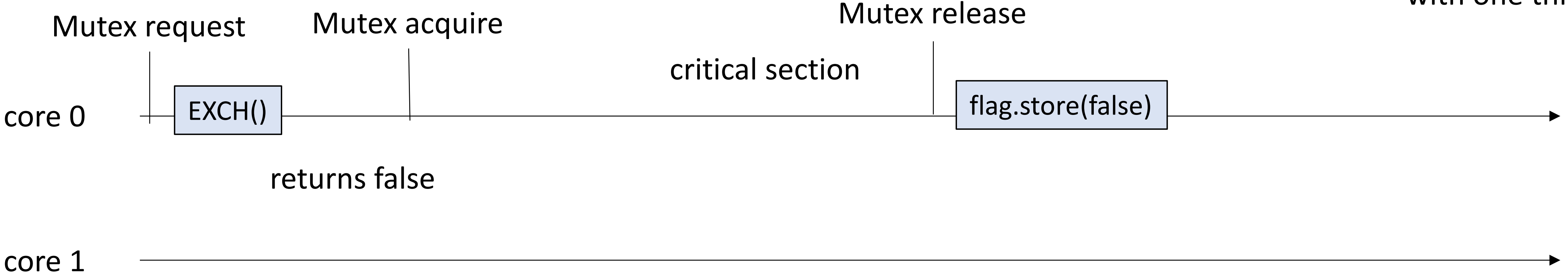
Analysis

```
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();
```



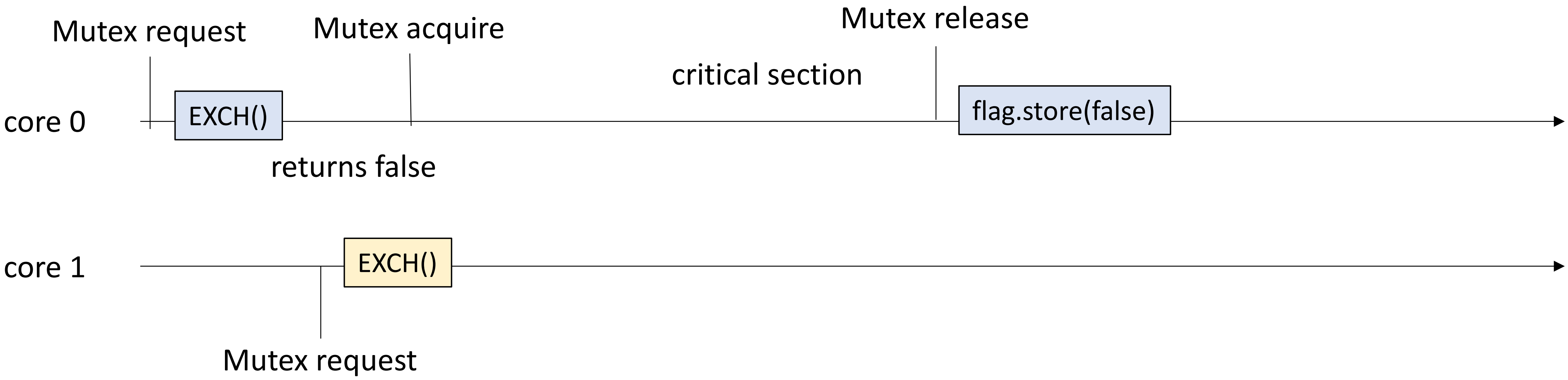
mutex works with one thread

Analysis

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

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

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



Analysis

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

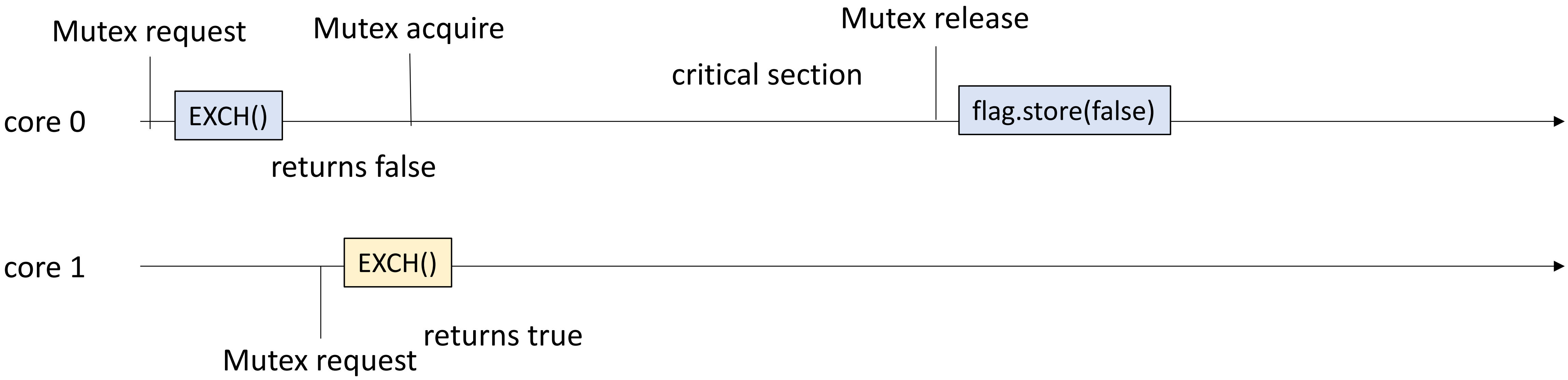
```
void unlock() {
  flag.store(false);
}
```

Thread 0:

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

Thread 1:

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



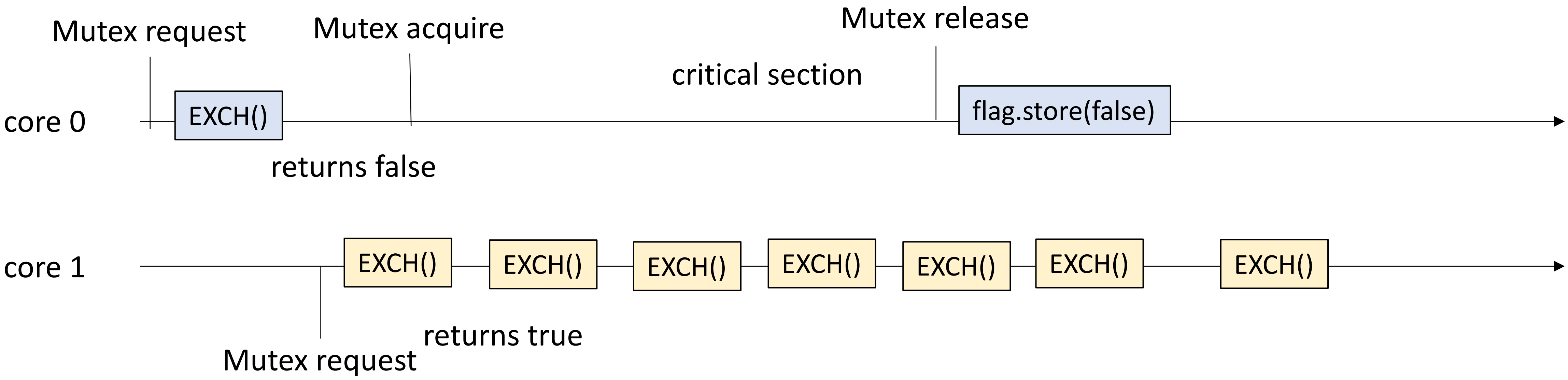
Analysis

```
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();
```



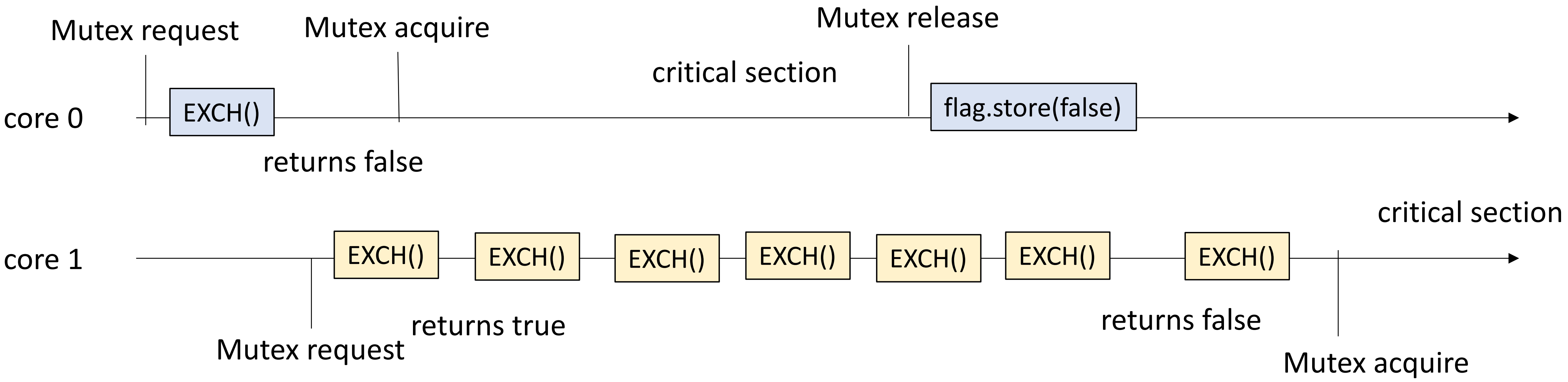
Analysis

```
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

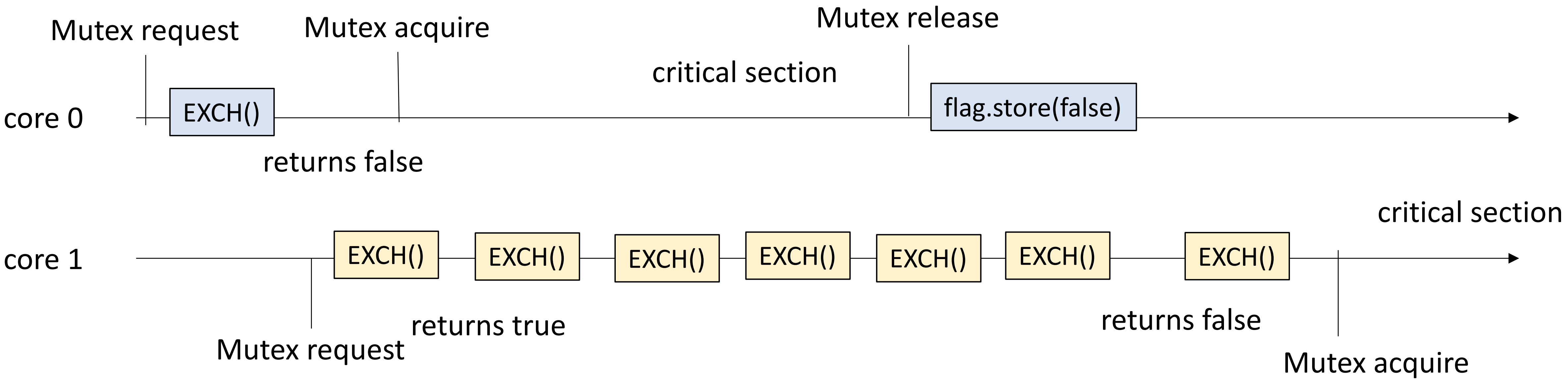
```
void lock() {
  while (atomic_exchange(&flag, true) == true);
}
```

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

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

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

what about interleavings?



Analysis

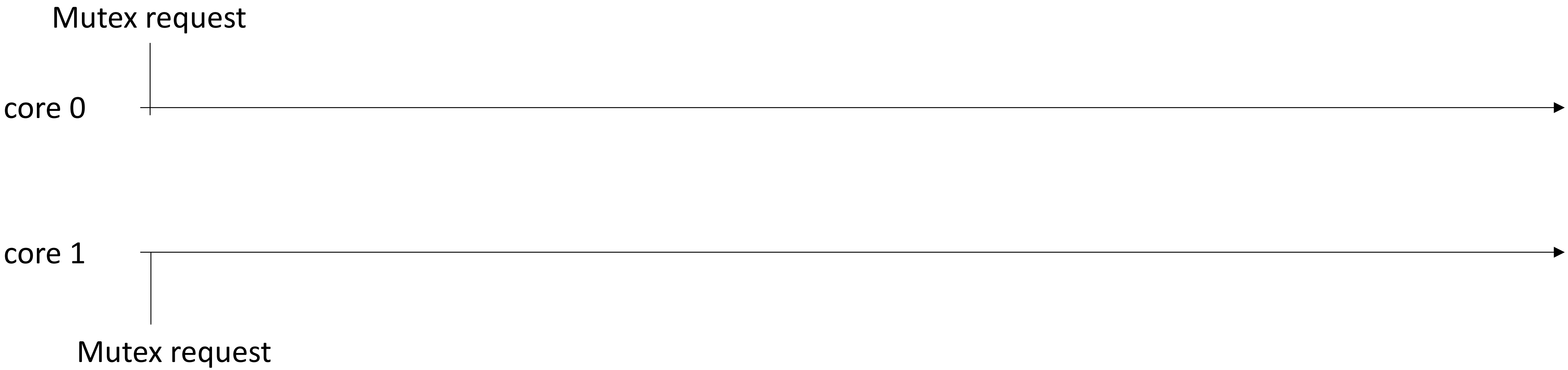
```
void lock() {
  while (atomic_exchange(&flag, true) == true);
}
```

```
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);
}
```

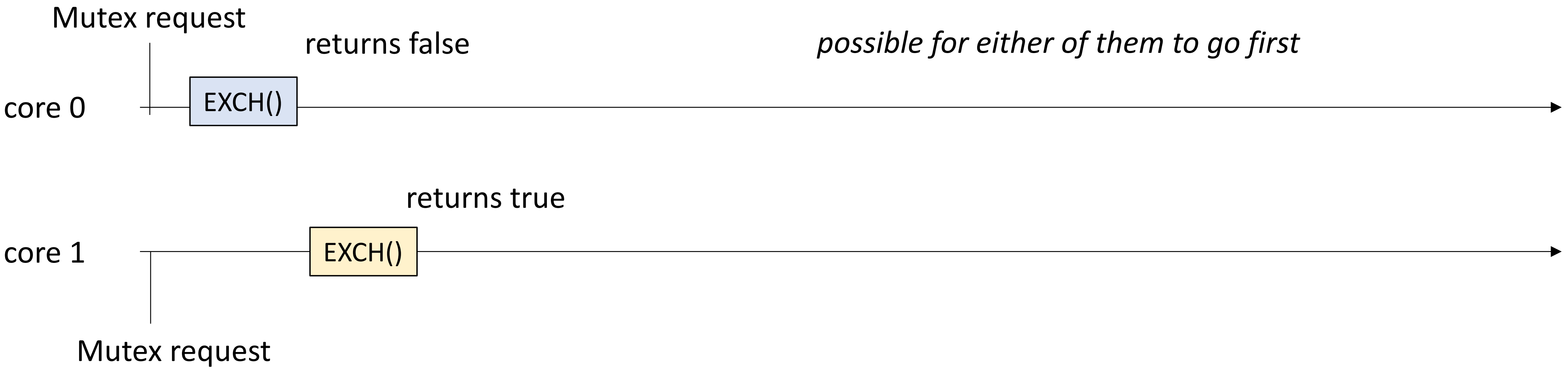
Thread 0:

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

Thread 1:

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

recall RMWS can't overlap!



Analysis

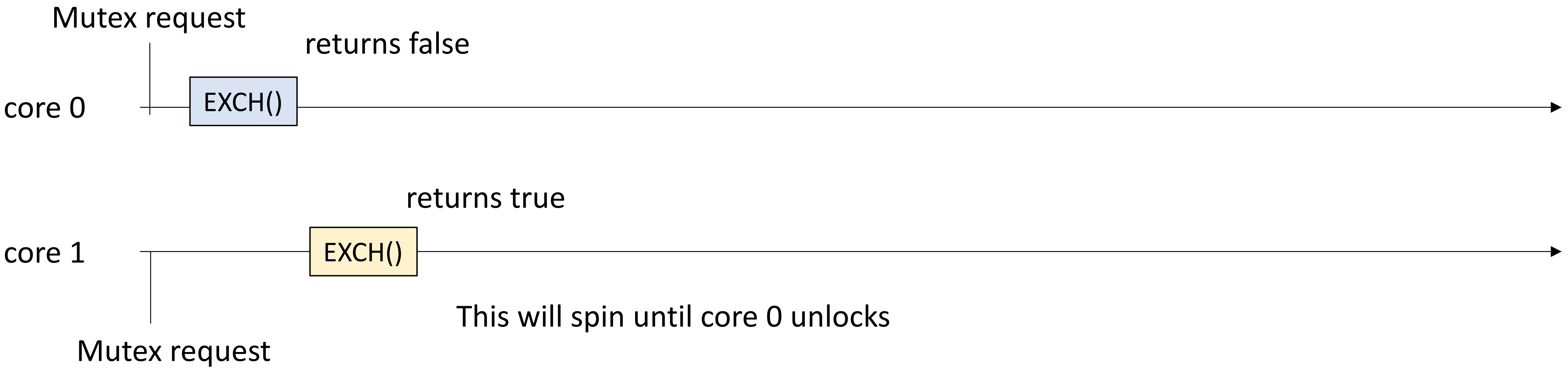
```
void lock() {
  while (atomic_exchange(&flag, true) == true);
}
```

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

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

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

recall RMWS can't overlap!



Analysis

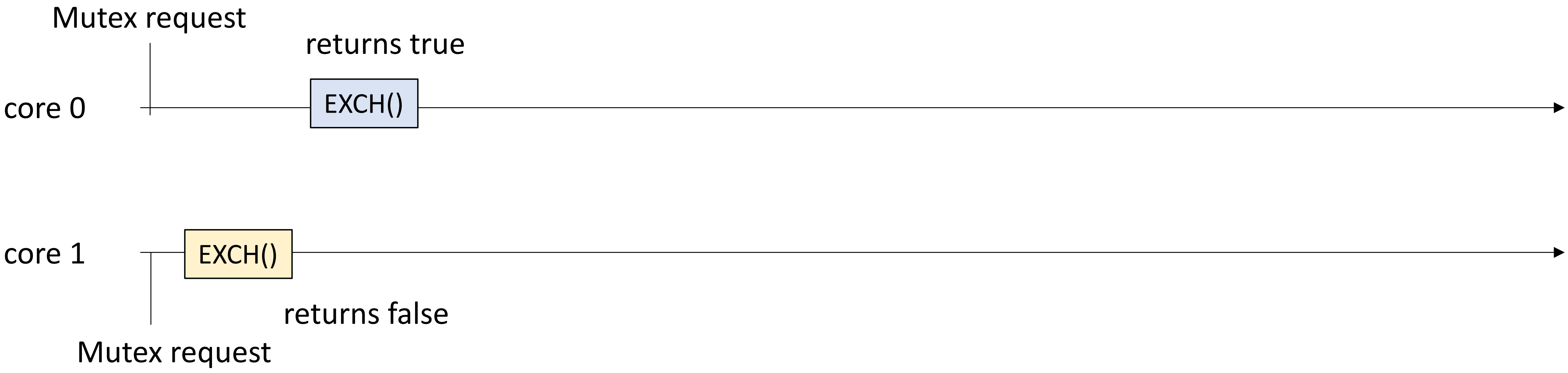
```
void lock() {
  while (atomic_exchange(&flag, true) == true);
}
```

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

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

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

recall RMWS can't overlap!



Analysis

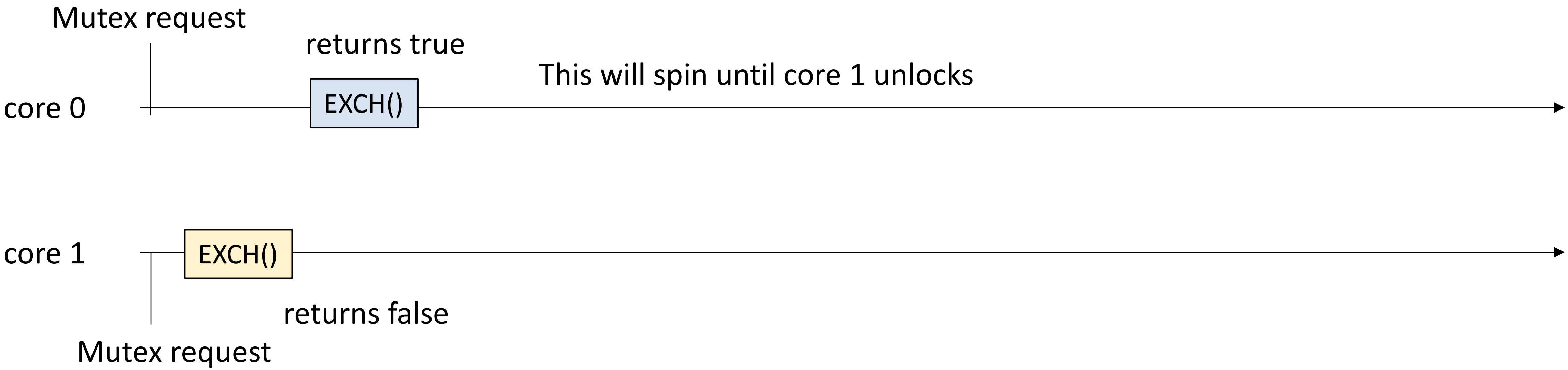
```
void lock() {
  while (atomic_exchange(&flag, true) == true);
}
```

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

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

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

recall RMWS can't overlap!

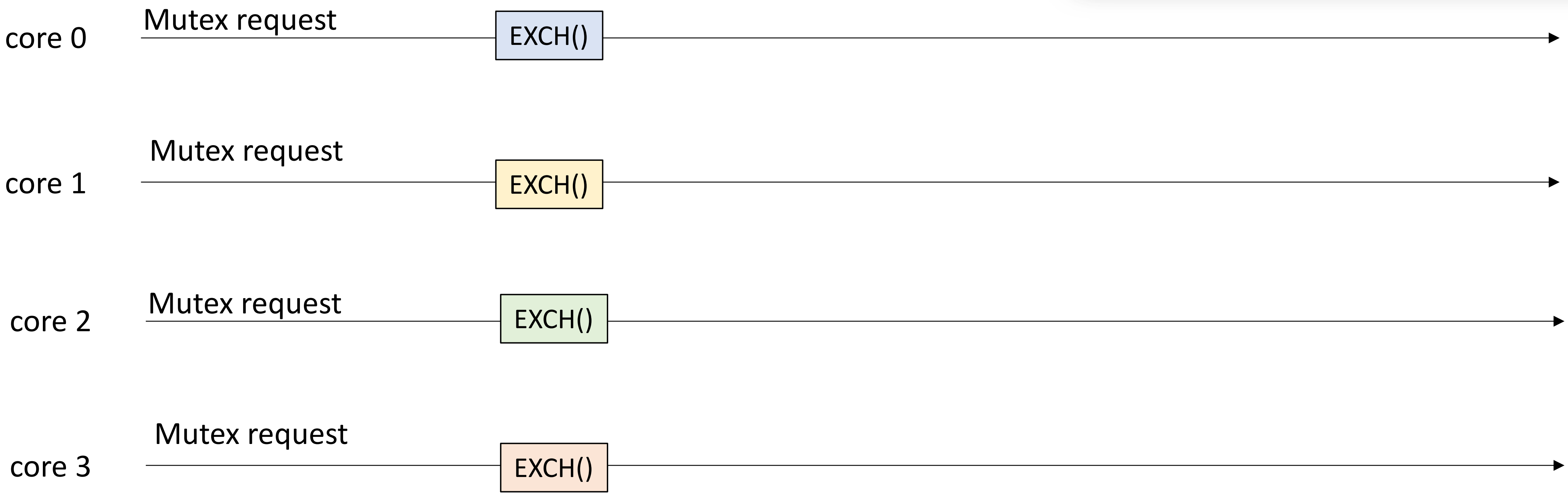


Analysis

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

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

what about 4 threads?



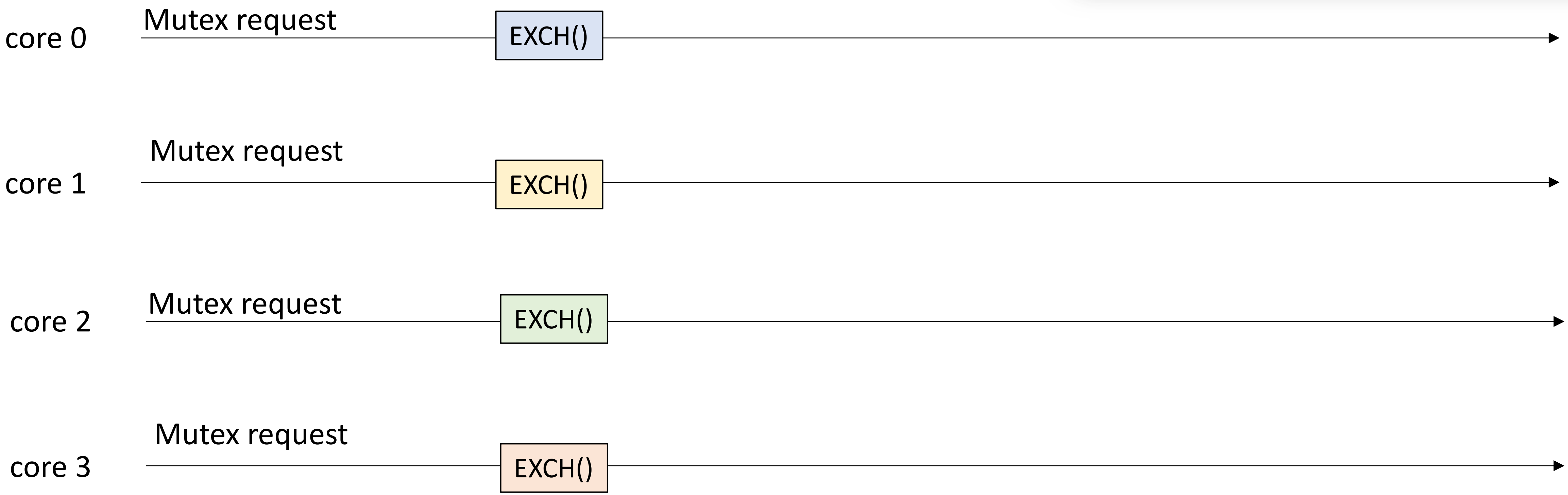
Analysis

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

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

what about 4 threads?

atomic operations can't overlap



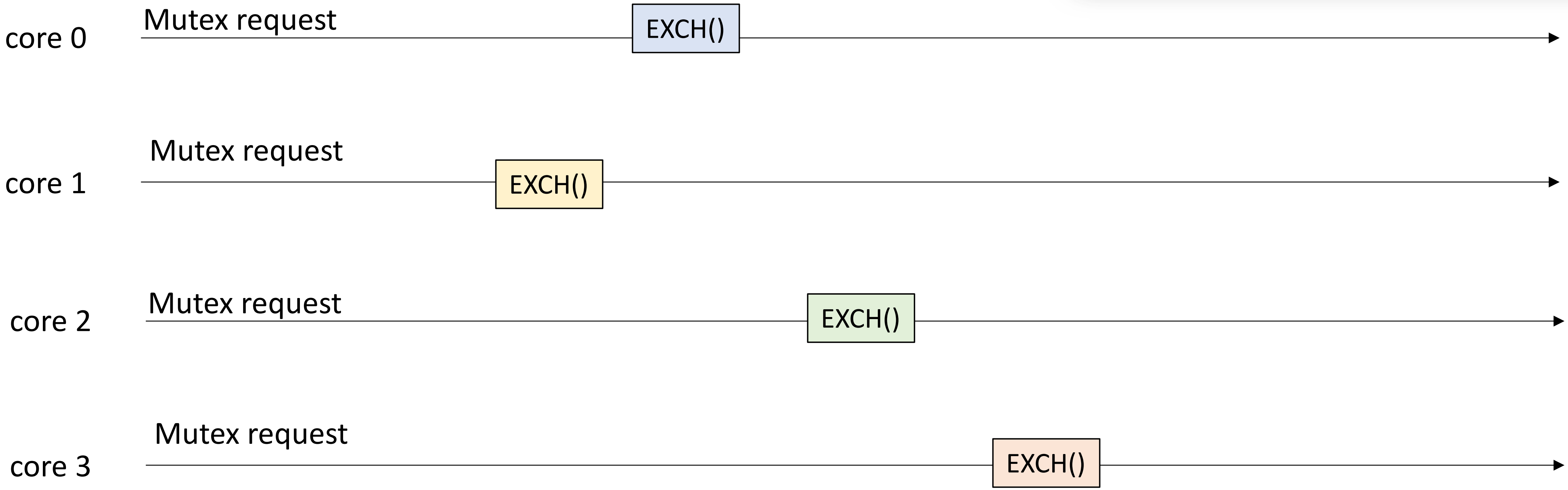
Analysis

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

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

what about 4 threads?

atomic operations can't overlap



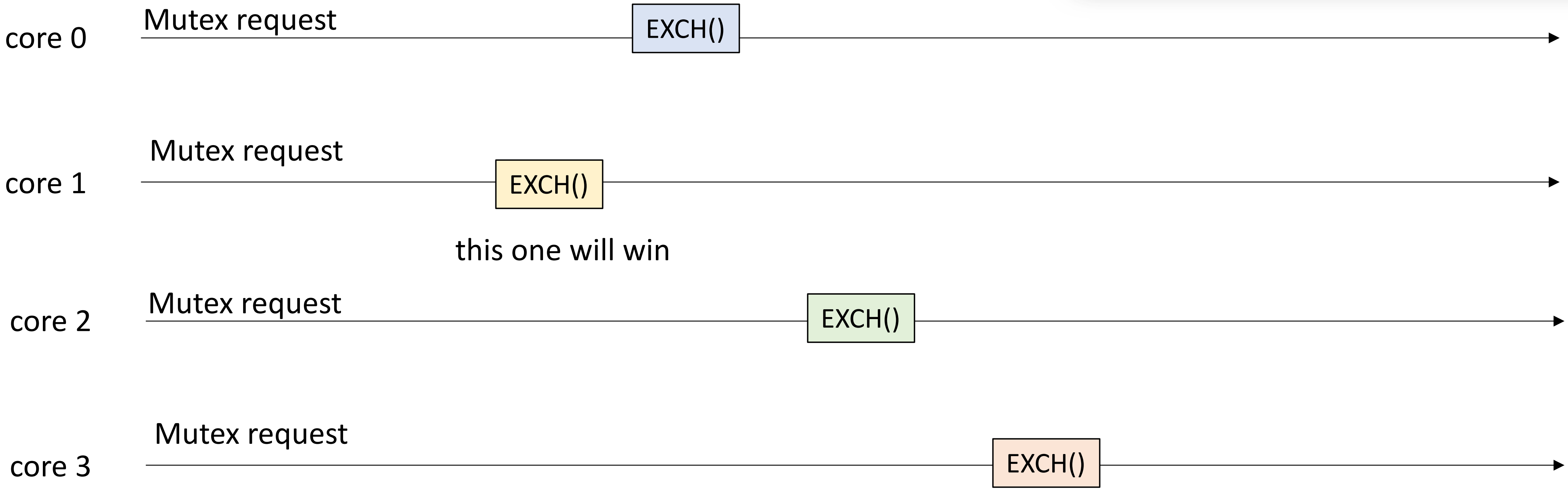
Analysis

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

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

what about 4 threads?

atomic operations can't overlap



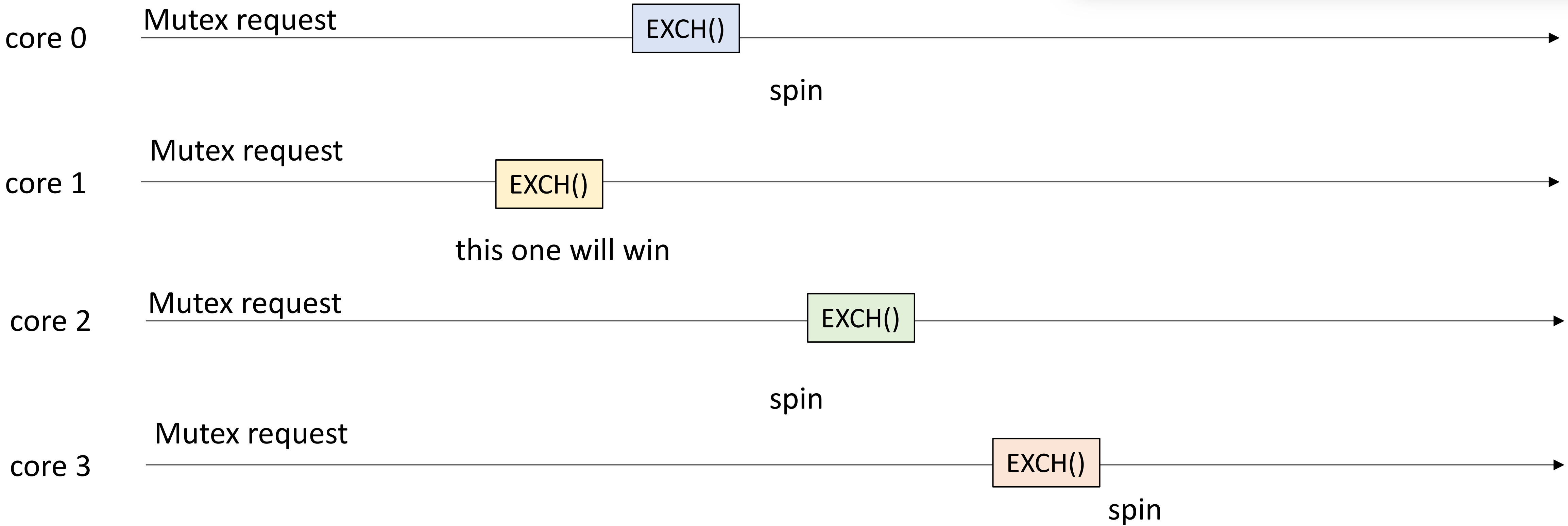
Analysis

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

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

what about 4 threads?

atomic operations can't overlap



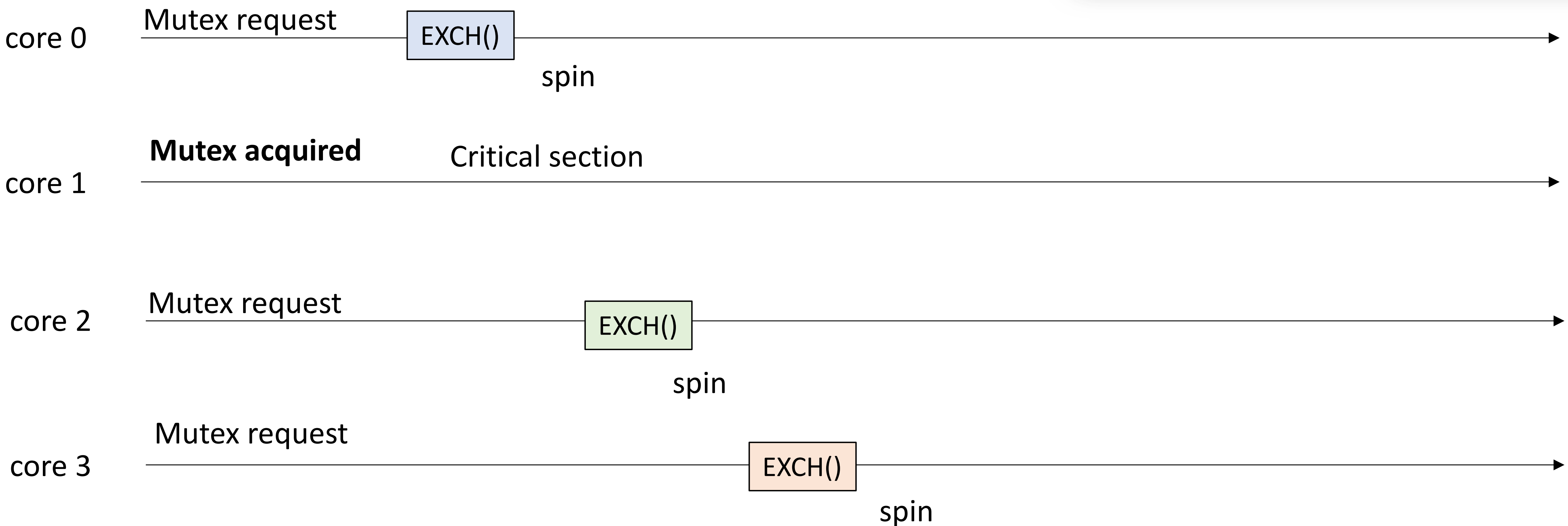
Analysis

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

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

what about 4 threads?

atomic operations can't overlap



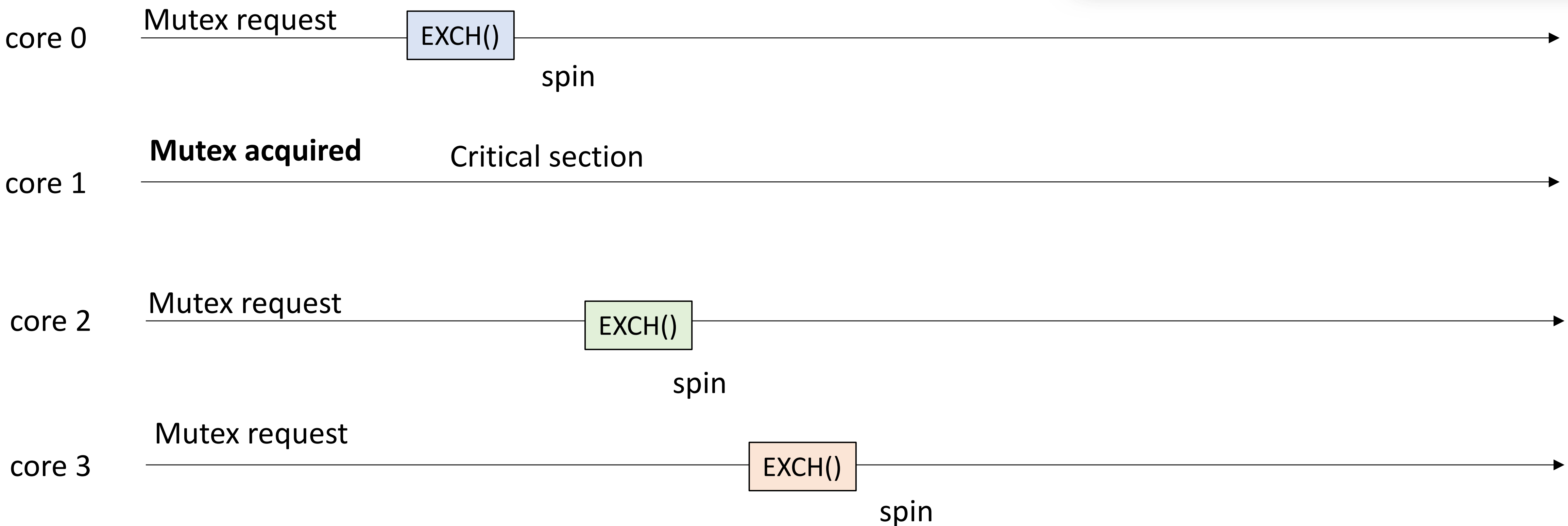
Analysis

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

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

what about 4 threads?

atomic operations can’t overlap



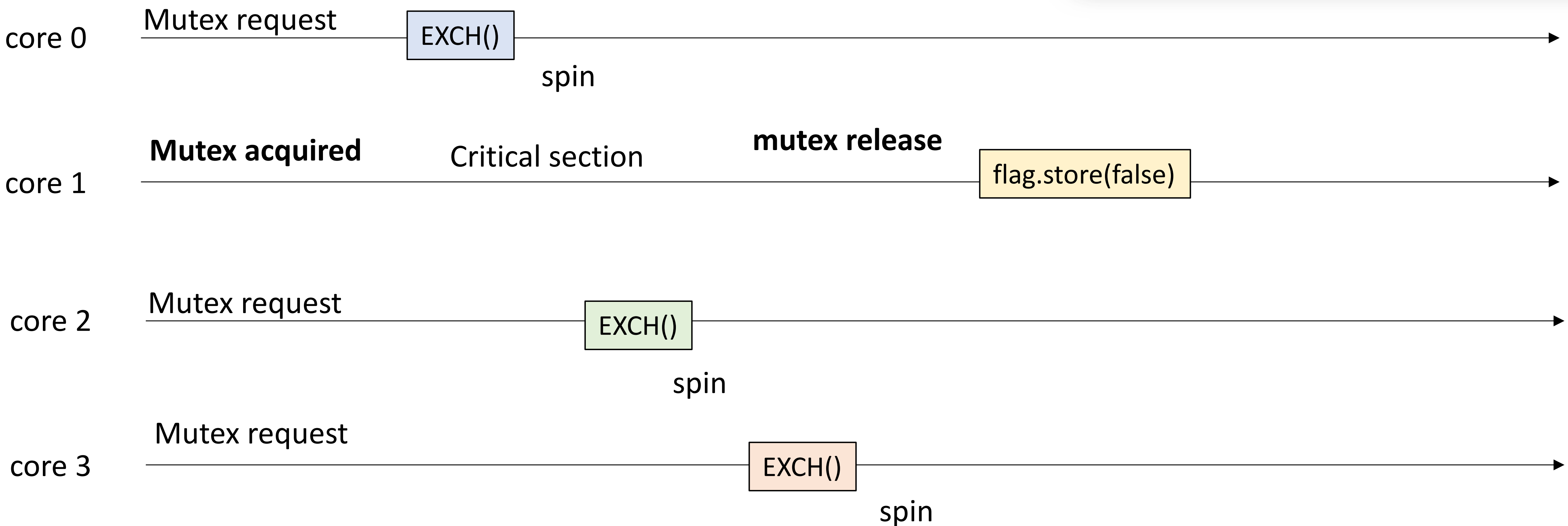
Analysis

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

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

what about 4 threads?

atomic operations can’t overlap



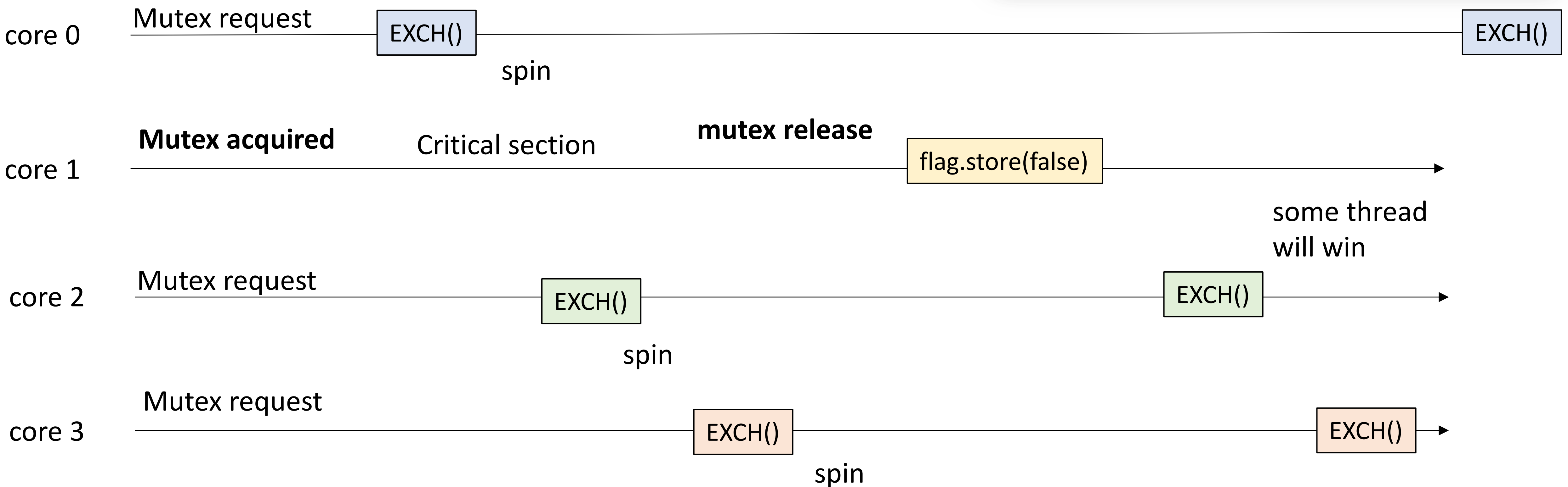
Analysis

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

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

what about 4 threads?

atomic operations can’t overlap



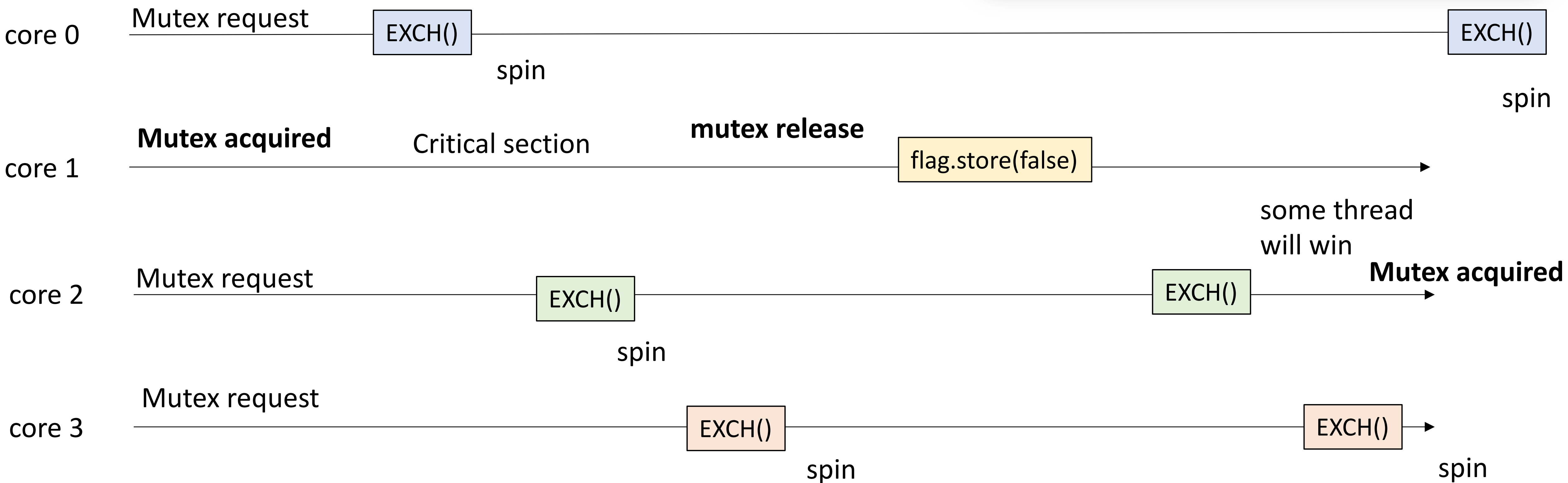
Analysis

```
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?

Lecture Schedule

- **Atomic RMW mutexes**
  - **Exchange**
  - **CAS**
  - Ticket
- Optimizations
  - Relaxed peeking
  - Backoff

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)

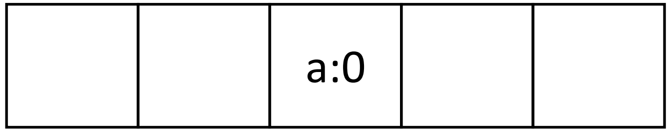
```
bool atomic_compare_exchange_strong(atomic *a, value *expected, value replace) {
   value tmp = a.load();
   if (tmp == *expected) {
     a.store(replace);
     return true;
   }
   *expected = tmp;
   return false;
}
```

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;
}
```

<u>thread 0:</u>

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

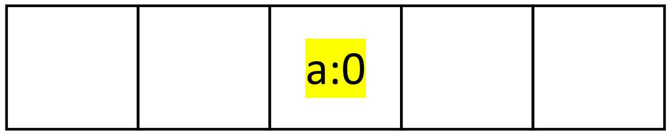


Most versatile RMW: Compare-and-swap

```
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);
```



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);
```

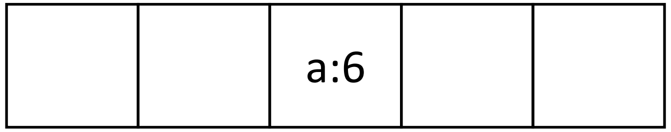
Most versatile RMW: Compare-and-swap

```
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);
```

true

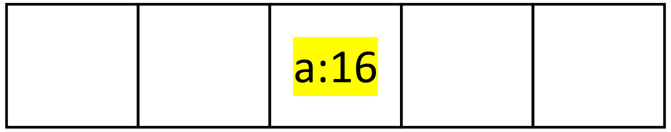


Most versatile RMW: Compare-and-swap

```
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;
}
```

<u>thread 0:</u>

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

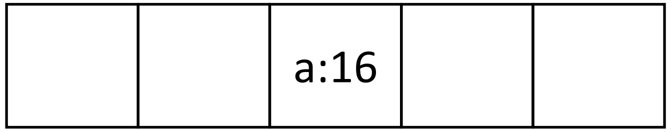


Most versatile RMW: Compare-and-swap

```
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);
```



16

false

CAS lock

```
#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

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

bool try_lock() {
  bool e = false;
  return atomic_compare_exchange_strong(&flag, &e, true);
}
```

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: versatile

- Why do I say it is versatile?

CAS: versatile

- Why do I say it is versatile?

- We can implement ANY other RMW using CAS!

Implementing atomic_fetch_add

```
int atomic_fetch_add(atomic_int *a, value v) {
  // implement me using CAS
}
```

Implementing atomic_fetch_add

```
int atomic_fetch_add(atomic_int *a, value v) {
  int old_val = a->load();
  int new_val = old_val + v;
  atomic_compare_exchange(a, &old_val, new_val);
}
```

Implementing atomic_fetch_add

```
int atomic_fetch_add(atomic_int *a, value v) {
  do {
    int old_val = a->load();
    int new_val = old_val + v;
    bool success = atomic_compare_exchange(a, &old_val, new_val);
  } while (!success)
}
```

Implementing atomic_fetch_add

could be any operation!

```
int atomic_fetch_add(atomic_int *a, value v) {
  do {
    int old_val = a->load();
    int new_val = old_val + v;
    bool success = atomic_compare_exchange(a, &old_val, new_val);
  } while (!success)
}
```

Implementing RMWs with CAS

- Gives you access to a wide range of operations!
  - `atomic_fetch_add` for float (not often provided)
  - You have to be careful with bit casting

- Why might this be difficult to implement?
  - Not provided in C++
  - Not provided for GPUs either (generally)
  - But very useful, especially for reduction and flow algorithms

How is CAS implemented?

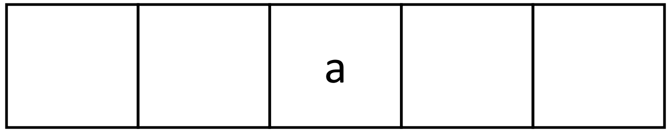
- X86 has an actual instruction
- ARM and POWER are load linked store conditional
- Show Godbolt example

Pessimistic Concurrency

- X86 has an actual instruction: lock the memory location
- Known as **Pessimistic Concurrency**
- Assume conflicts will happen and defend against them from the start

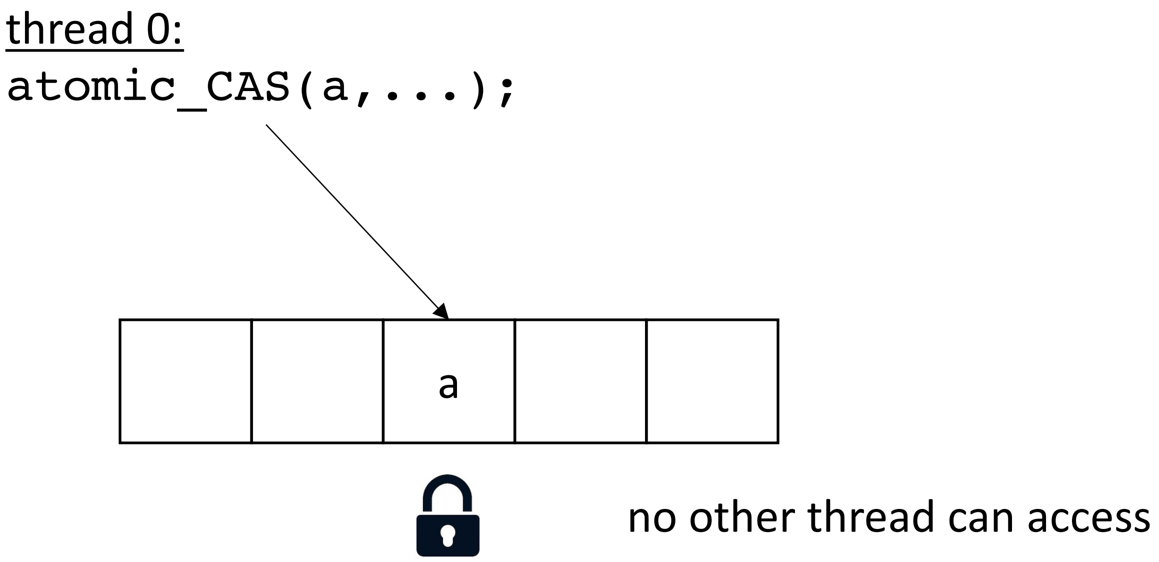
thread 0:

```
atomic_CAS(a,...);
```



Pessimistic Concurrency

- X86 has an actual instruction: lock the memory location
- Known as **Pessimistic Concurrency**
- Assume conflicts will happen and defend against them from the start

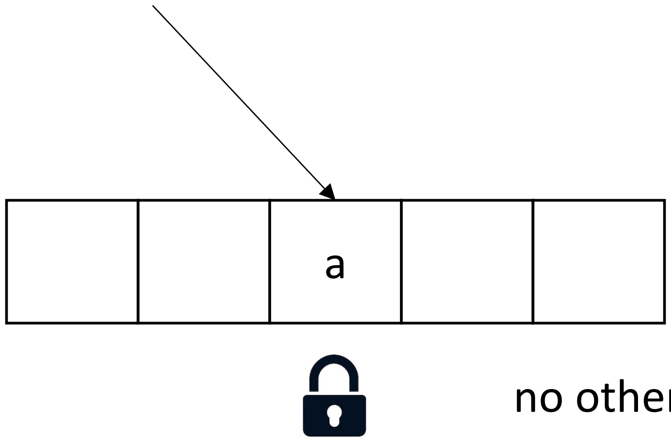


Pessimistic Concurrency

- X86 has an actual instruction: lock the memory location
- Known as **Pessimistic Concurrency**
- Assume conflicts will happen and defend against them from the start

thread 0:
`atomic_CAS(a,...);`

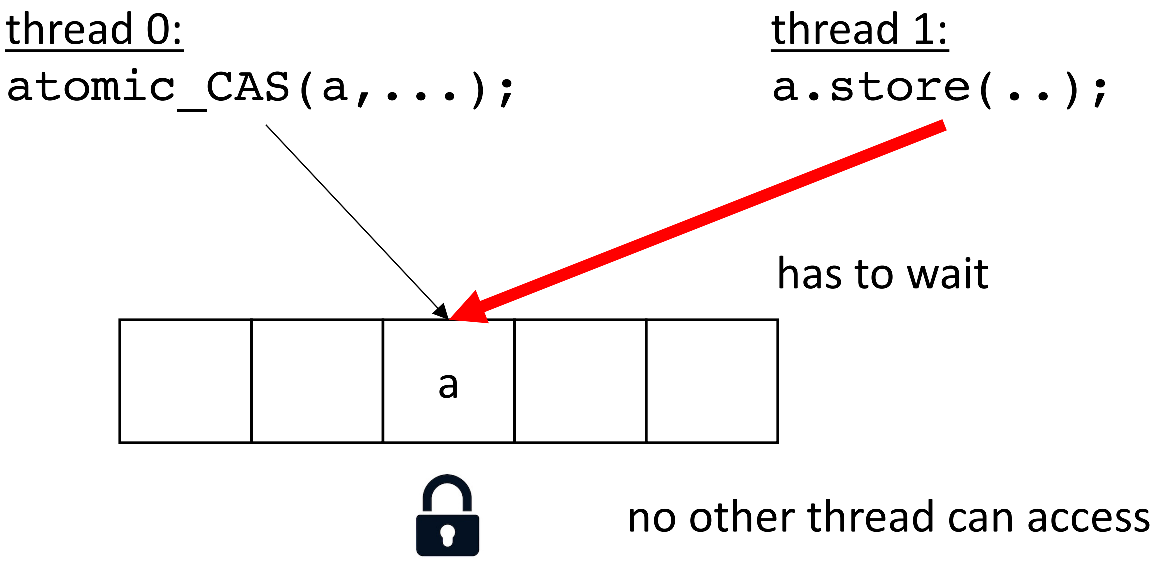
thread 1:
`a.store(..);`



no other thread can access

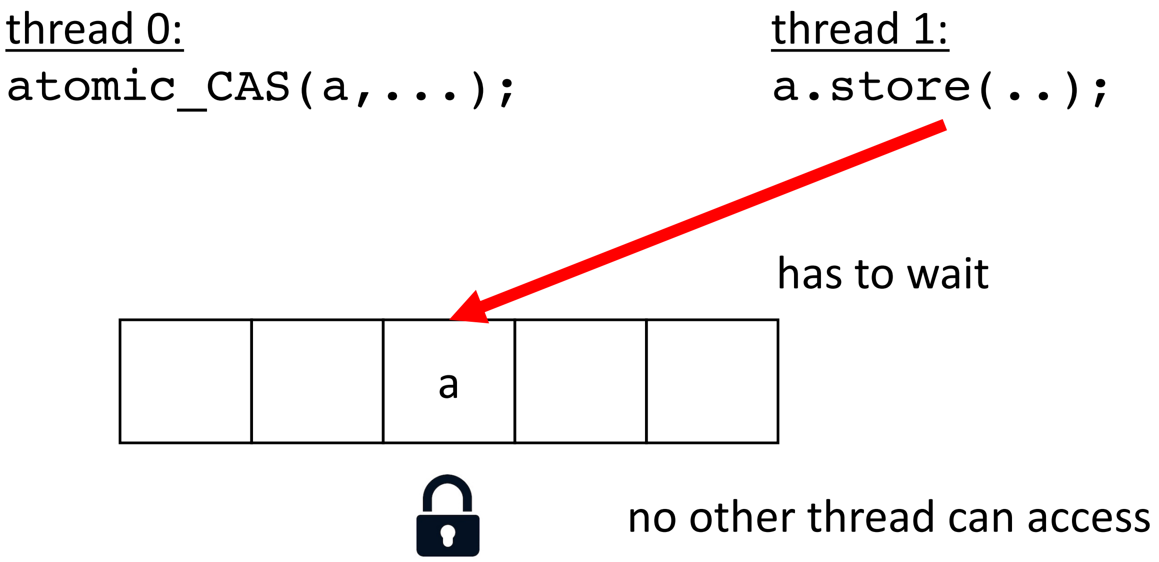
Pessimistic Concurrency

- X86 has an actual instruction: lock the memory location
- Known as **Pessimistic Concurrency**
- Assume conflicts will happen and defend against them from the start



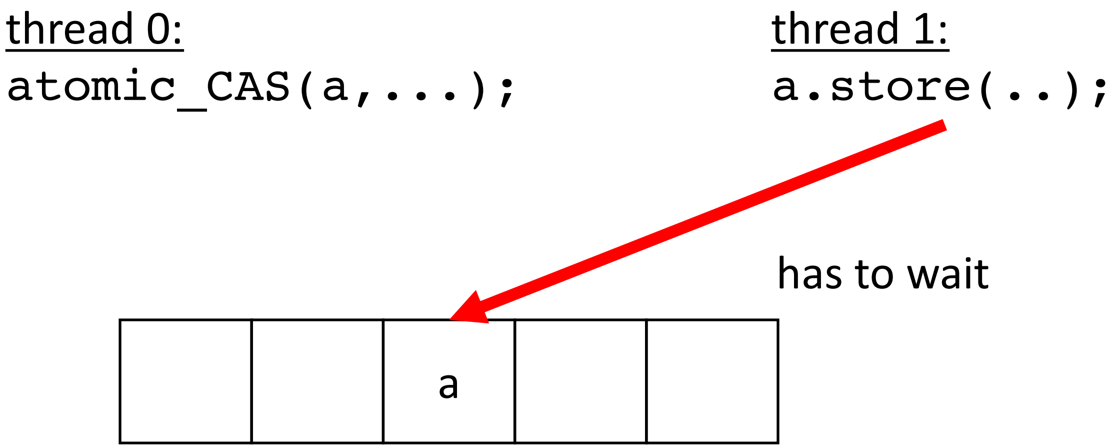
Pessimistic Concurrency

- X86 has an actual instruction: lock the memory location
- Known as **Pessimistic Concurrency**
- Assume conflicts will happen and defend against them from the start



Pessimistic Concurrency

- X86 has an actual instruction: lock the memory location
- Known as **Pessimistic Concurrency**
- Assume conflicts will happen and defend against them from the start

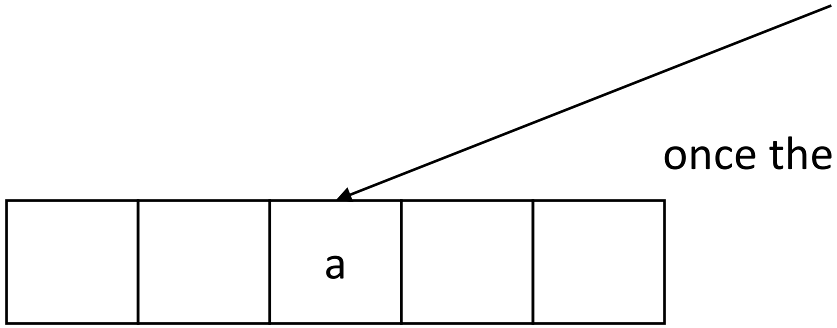


Pessimistic Concurrency

- X86 has an actual instruction: lock the memory location
- Known as **Pessimistic Concurrency**
- Assume conflicts will happen and defend against them from the start

thread 0:
```
atomic_CAS(a,...);
```

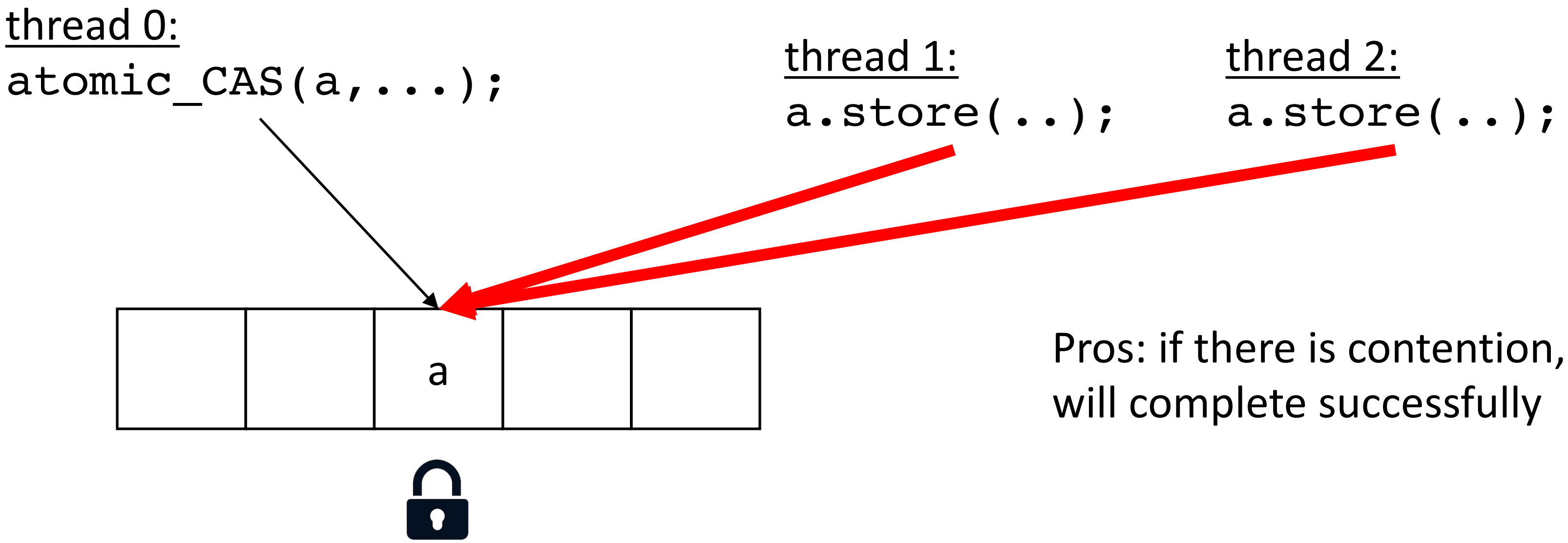
thread 1:
```
a.store(..);
```



once the lock is released then we can access

Pessimistic Concurrency

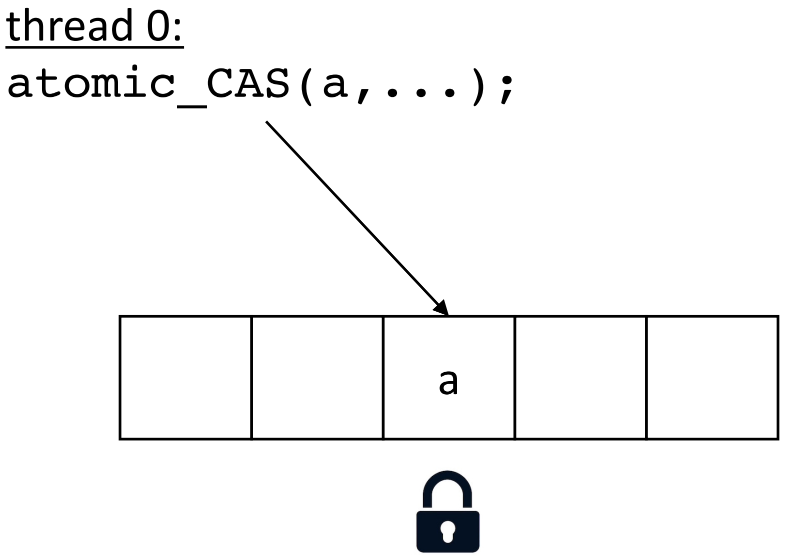
- X86 has an actual instruction: lock the memory location
- Known as **Pessimistic Concurrency**
- Assume conflicts will happen and defend against them from the start



Pros: if there is contention, the CAS will complete successfully

Pessimistic Concurrency

- X86 has an actual instruction: lock the memory location
- Known as **Pessimistic Concurrency**
- Assume conflicts will happen and defend against them from the start



Cons: if no other threads are contending, lock overhead is high

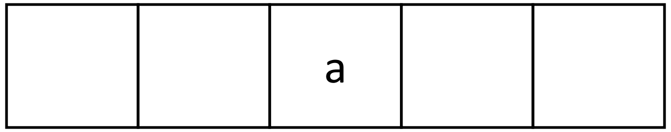
Optimistic Concurrency

- ARM has load/store exclusive
- Known as **Optimistic Concurrency**
- Assume ***no*** conflicts will happen. Detects and reacts to them.

thread 0:

```
tmp = load_exclusive(a,...);
tmp += 1;
store_exclusive(a, tmp);
```

For this example consider an atomic increment

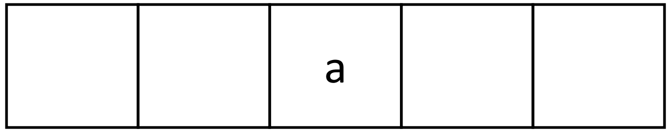


Optimistic Concurrency

- ARM has load/store exclusive
- Known as **Optimistic Concurrency**
- Assume ***no*** conflicts will happen. Detects and reacts to them.

thread 0:

```
tmp = load_exclusive(a,...);
tmp += 1;
store_exclusive(a, tmp);
```



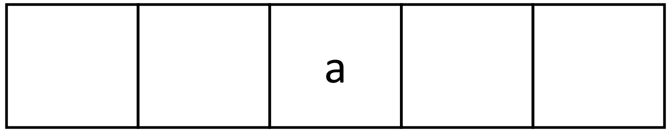
T0_exclusive = 1

Optimistic Concurrency

- ARM has load/store exclusive
- Known as **Optimistic Concurrency**
- Assume ***no*** conflicts will happen. Detects and reacts to them.

thread 0:

```
tmp = load_exclusive(a,...);
tmp += 1;
store_exclusive(a, tmp);
```



T0_exclusive = 1

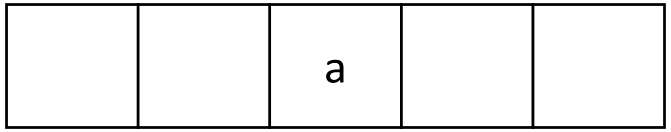
Optimistic Concurrency

- ARM has load/store exclusive
- Known as **Optimistic Concurrency**
- Assume ***no*** conflicts will happen. Detects and reacts to them.

thread 0:

```
tmp = load_exclusive(a,...);
tmp += 1;
store_exclusive(a, tmp);
```

before we store, we have to check if there was a conflict.



T0_exclusive = 1

Optimistic Concurrency

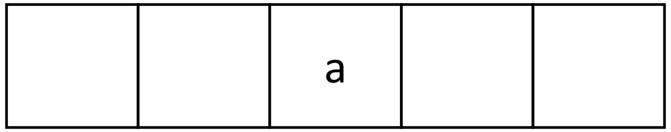
- ARM has load/store exclusive
- Known as **Optimistic Concurrency**
- Assume ***no*** conflicts will happen. Detects and reacts to them.

<u>thread 0:</u>

```
tmp = load_exclusive(a,...);
tmp += 1;
store_exclusive(a, tmp);
```

<u>thread 1:</u>

```
a.store(...)
```



Optimistic Concurrency

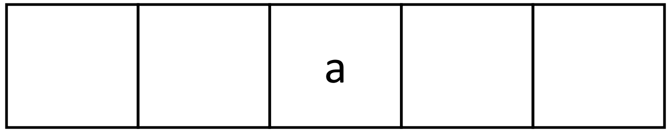
- ARM has load/store exclusive
- Known as **Optimistic Concurrency**
- Assume ***no*** conflicts will happen. Detects and reacts to them.

thread 0:

```
tmp = load_exclusive(a,...);
tmp += 1;
store_exclusive(a, tmp);
```

thread 1:

```
a.store(...)
```



T0_exclusive = 1

Optimistic Concurrency

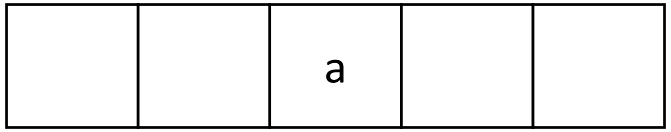
- ARM has load/store exclusive
- Known as **Optimistic Concurrency**
- Assume ***no*** conflicts will happen. Detects and reacts to them.

thread 0:

```
tmp = load_exclusive(a,...);
tmp += 1;
store_exclusive(a, tmp);
```

thread 1:

```
a.store(...)
```



T0_exclusive = 1

Optimistic Concurrency

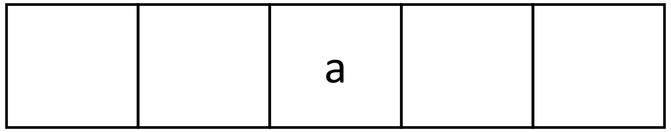
- ARM has load/store exclusive
- Known as **Optimistic Concurrency**
- Assume ***no*** conflicts will happen. Detects and reacts to them.

thread 0:

```
tmp = load_exclusive(a,...);
tmp += 1;
store_exclusive(a, tmp);
```

thread 1:

```
a.store(...)
```



T0_exclusive = 0

Optimistic Concurrency

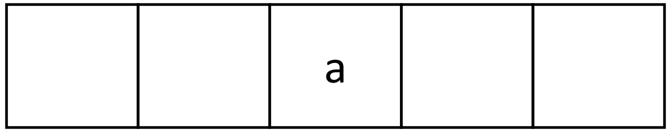
- ARM has load/store exclusive
- Known as **Optimistic Concurrency**
- Assume ***no*** conflicts will happen. Detects and reacts to them.

thread 0:

```
tmp = load_exclusive(a,...);
tmp += 1;
store_exclusive(a, tmp);
```

thread 1:

```
a.store(...)
```



T0_exclusive = 0

Optimistic Concurrency

- ARM has load/store exclusive
- Known as **Optimistic Concurrency**
- Assume ***no*** conflicts will happen. Detects and reacts to them.

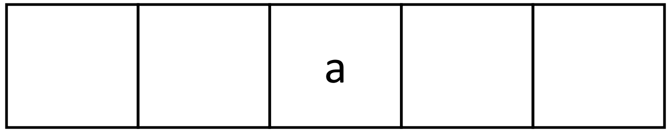
thread 0:

```
tmp = load_exclusive(a,...);
tmp += 1;
store_exclusive(a, tmp);
```

thread 1:

```
a.store(...)
```

can't store because our exclusive bit was changed, i.e. there was a conflict!



T0_exclusive = 0

Optimistic Concurrency

- ARM has load/store exclusive
- Known as **Optimistic Concurrency**
- Assume ***no*** conflicts will happen. Detects and reacts to them.

thread 0:

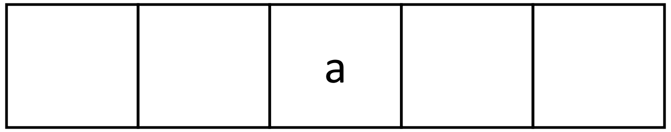
```
tmp = load_exclusive(a,...);
tmp += 1;
store_exclusive(a, tmp);
```

thread 1:

```
a.store(...)
```

can’t store because our exclusive bit was changed, i.e. there was a conflict!

solution: loop until success:



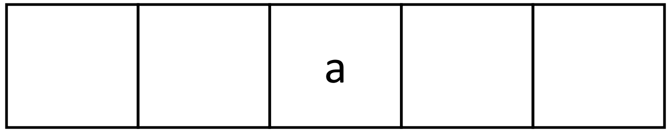
T0_exclusive = 0

Optimistic Concurrency

- ARM has load/store exclusive
- Known as **Optimistic Concurrency**
- Assume ***no*** conflicts will happen. Detects and reacts to them.

thread 0:

```
do {
tmp = load_exclusive(a,...);
tmp += 1;
} while(!store_exclusive(a, tmp));
```



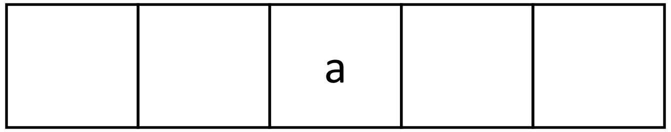
T0_exclusive = 0

Optimistic Concurrency

- ARM has load/store exclusive
- Known as **Optimistic Concurrency**
- Assume ***no*** conflicts will happen. Detects and reacts to them.

<u>thread 0:</u>

```
do {
tmp = load_exclusive(a,...);
tmp += 1;
} while(!store_exclusive(a, tmp));
```



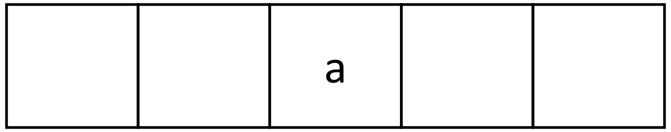
T0_exclusive = 1

Optimistic Concurrency

- ARM has load/store exclusive
- Known as **Optimistic Concurrency**
- Assume ***no*** conflicts will happen. Detects and reacts to them.

<u>thread 0:</u>

```
do {
tmp = load_exclusive(a,...);
tmp += 1;
} while(!store_exclusive(a, tmp));
```



T0_exclusive = 1

Pros: very efficient when there is no conflicts!

Cons: conflicts are very expensive!

Spinning thread might starve (but not indefinitely) if other threads are constantly writing.

Back to mutexes…

- Speaking of starvation:

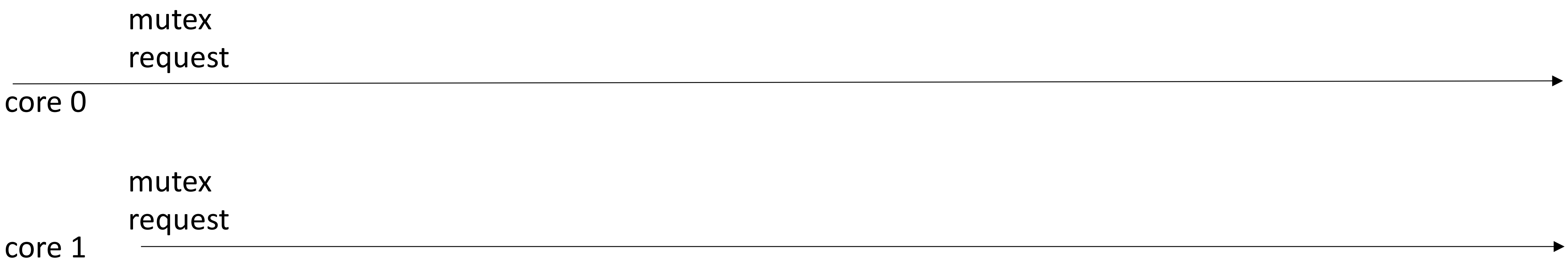
- Are the Exchange lock or Spin lock starvation free?

Analysis

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

Is this mutex starvation Free?

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

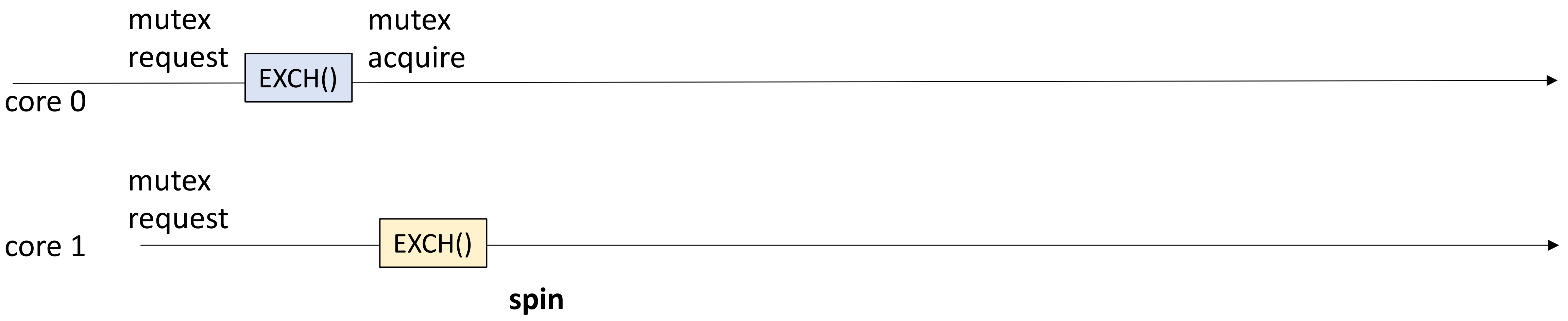


Analysis

Is this mutex starvation Free?

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

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

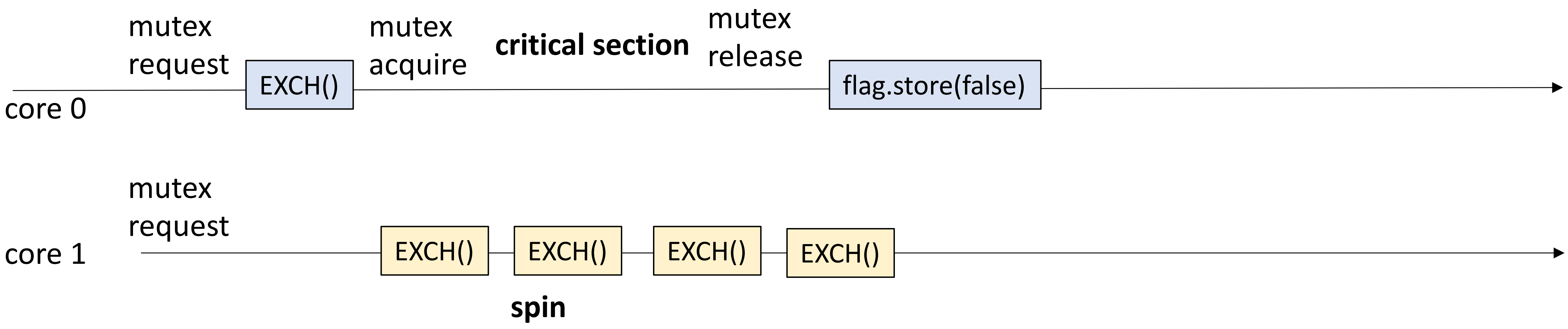


Analysis

Is this mutex starvation Free?

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

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

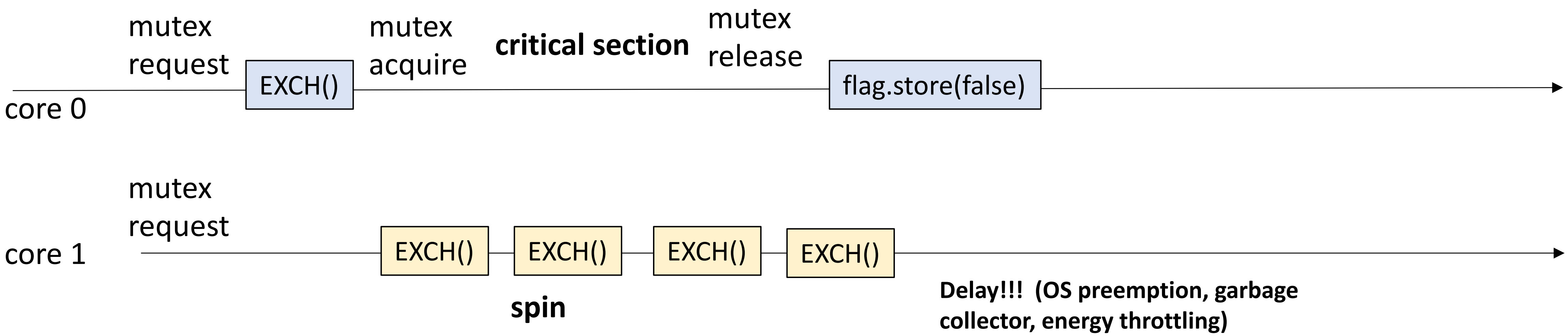


Analysis

Is this mutex starvation Free?

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

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

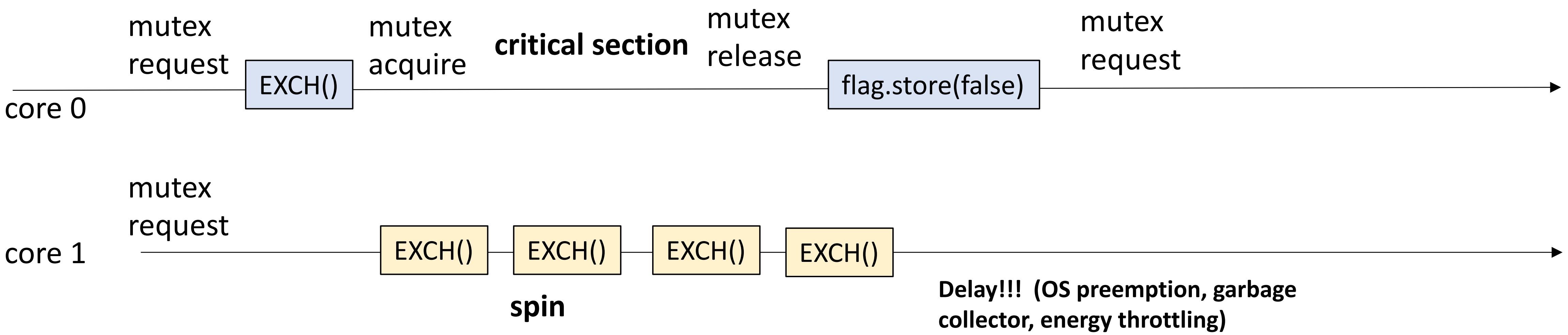


Analysis

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

Is this mutex starvation Free?

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

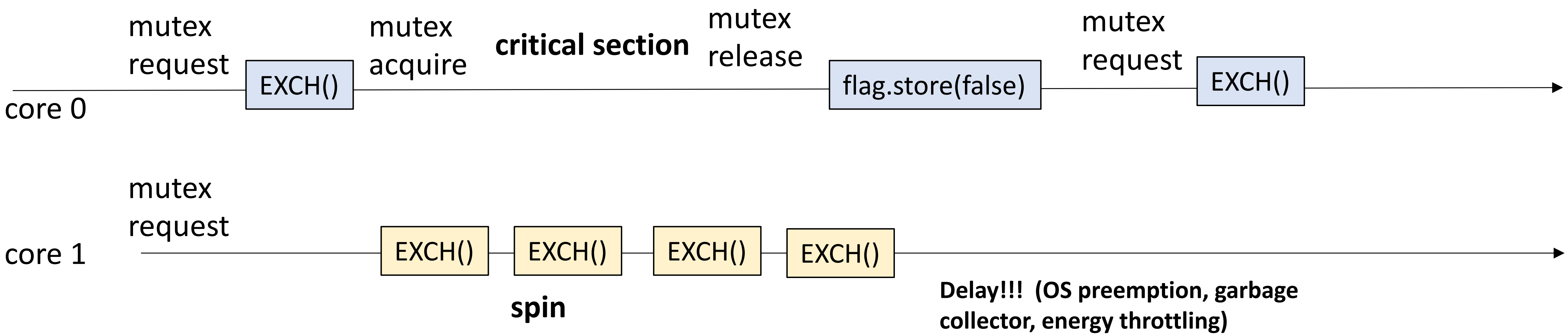


Analysis

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

Is this mutex starvation Free?

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

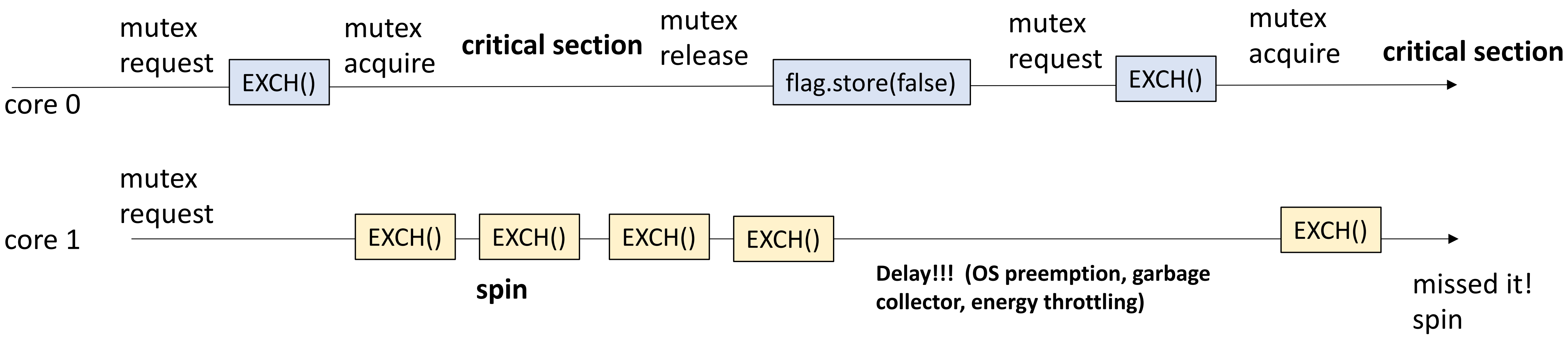


Analysis

Is this mutex starvation Free?

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

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



How does this look in practice

- Try it

How can we make this more fair?

- Use a different atomic instruction:
  - `int atomic_fetch_add(atomic_int *a, int v);`

We've seen this one before!

How can we make this more fair?

- Use a different atomic instruction:
  - `int atomic_fetch_add(atomic_int *a, int v);`

We've seen this one before!
intuition: take a ticket

like at Zoccoli's!

Lecture Schedule

- **Atomic RMW mutexes**
  - **Exchange**
  - **CAS**
  - **Ticket**

- Optimizations
  - Relaxed peeking
  - Backoff

Ticket lock

```
class Mutex {
public:
  Mutex() {
    counter = 0;
    currently_serving = 0;
  }

  void lock() {
    int my_number = atomic_fetch_add(&counter, 1);
    while (currently_serving.load() != my_number);
  }

  void unlock() {
    int tmp = currently_serving.load();
    tmp += 1;
    currently_serving.store(tmp);
  }

private:
  atomic_int counter;
  atomic_int currently_serving;
};
```

- Ticket lock: instead of 1 bit, we need an integer for the counter.
- The mutex also needs to track of which ticket is currently being served

Ticket lock

```
class Mutex {
public:
  Mutex() {
    counter = 0;
    currently_serving = 0;
  }

  void lock() {
    int my_number = atomic_fetch_add(&counter, 1);
    while (currently_serving.load() != my_number);
  }

  void unlock() {
    int tmp = currently_serving.load();
    tmp += 1;
    currently_serving.store(tmp);
  }

private:
  atomic_int counter;
  atomic_int currently_serving;
};
```

- Ticket lock: instead of 1 bit, we need an integer for the counter.
- The mutex also needs to track of which ticket is currently being served

Get a unique number

Spin while your number isn’t being served

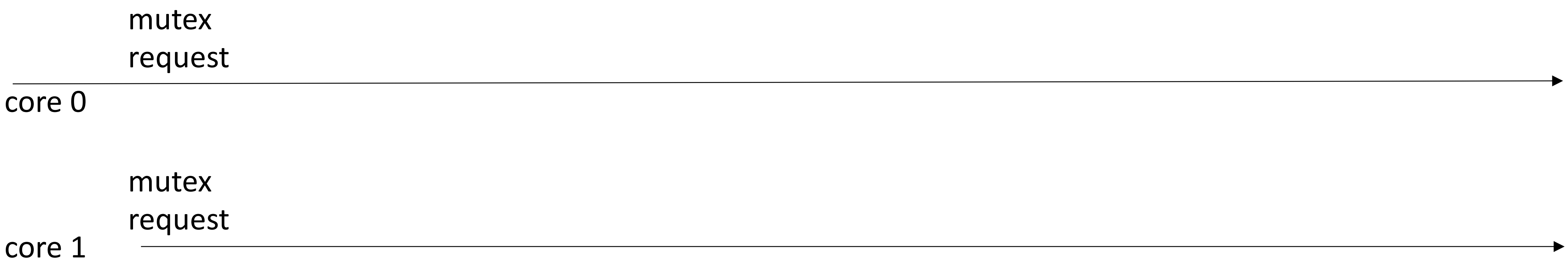
To release, increment the number that’s currently being served.

Analysis

Is this mutex starvation Free?

```
void lock() {
  int my_number = atomic_fetch_add(&counter, 1);
  while (currently_serving.load() != my_number);
}

void unlock() {
  int tmp = currently_serving.load();
  tmp += 1;
  currently_serving.store(tmp);
}
```

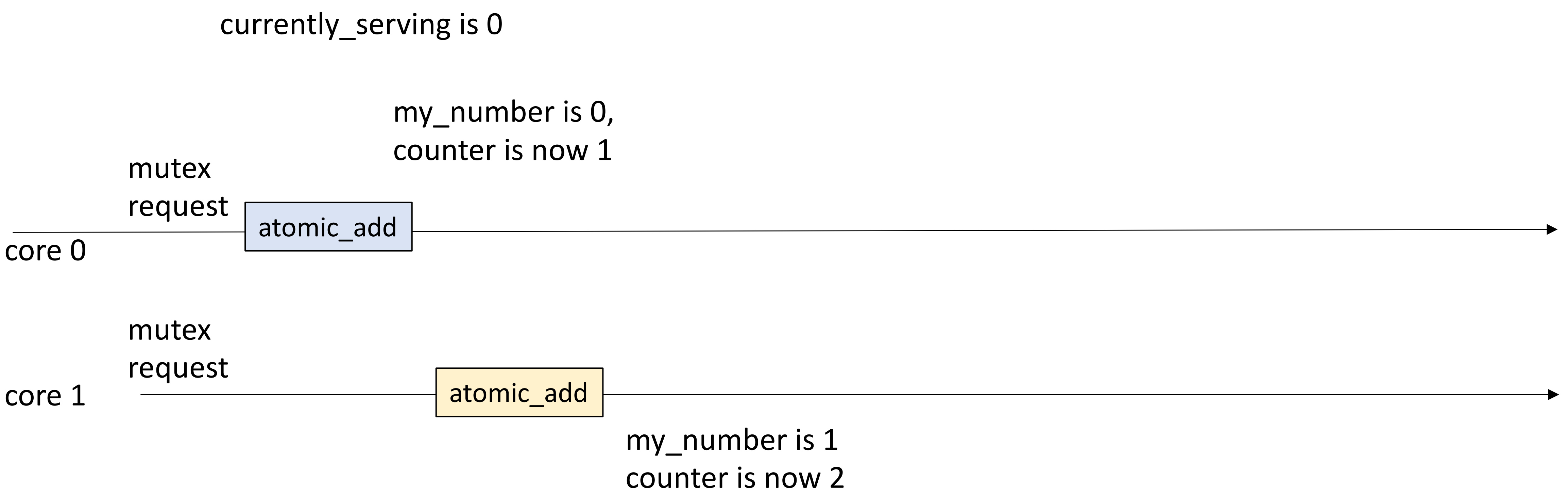


Analysis

Is this mutex starvation Free?

```
void lock() {
  int my_number = atomic_fetch_add(&counter, 1);
  while (currently_serving.load() != my_number);
}

void unlock() {
  int tmp = currently_serving.load();
  tmp += 1;
  currently_serving.store(tmp);
}
```



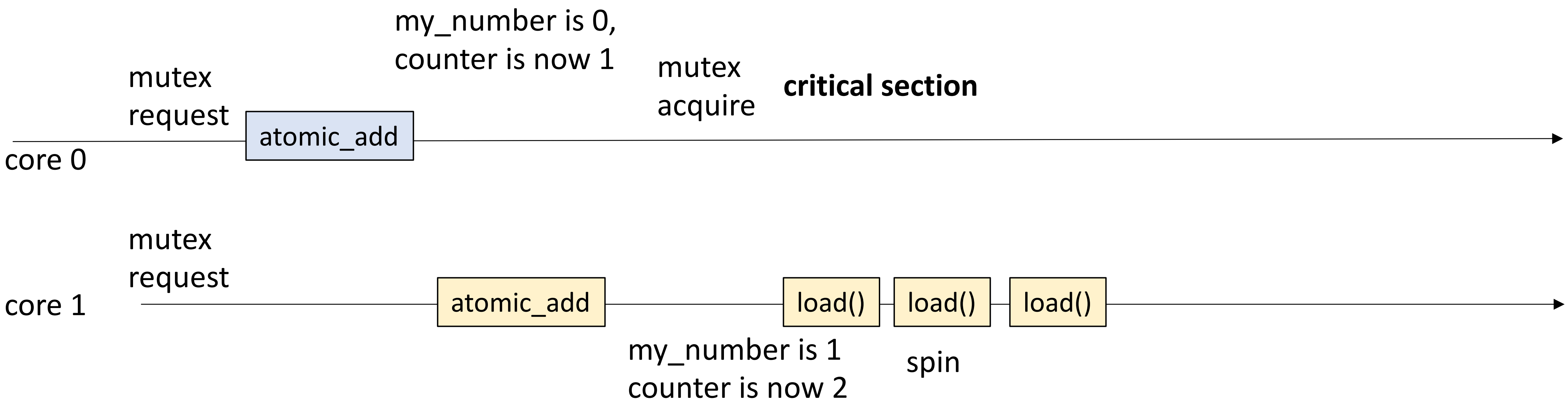
Analysis

Is this mutex starvation Free?

```
void lock() {
  int my_number = atomic_fetch_add(&counter, 1);
  while (currently_serving.load() != my_number);
}

void unlock() {
  int tmp = currently_serving.load();
  tmp += 1;
  currently_serving.store(tmp);
}
```

currently_serving is 0

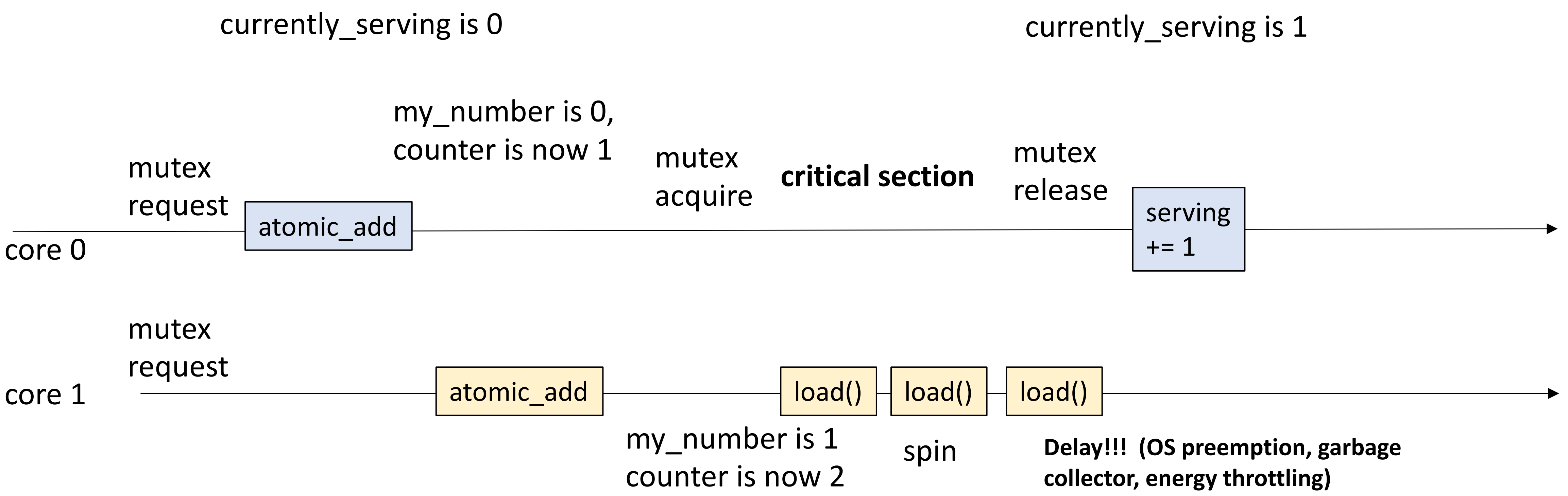


Analysis

Is this mutex starvation Free?

```
void lock() {
  int my_number = atomic_fetch_add(&counter, 1);
  while (currently_serving.load() != my_number);
}

void unlock() {
  int tmp = currently_serving.load();
  tmp += 1;
  currently_serving.store(tmp);
}
```

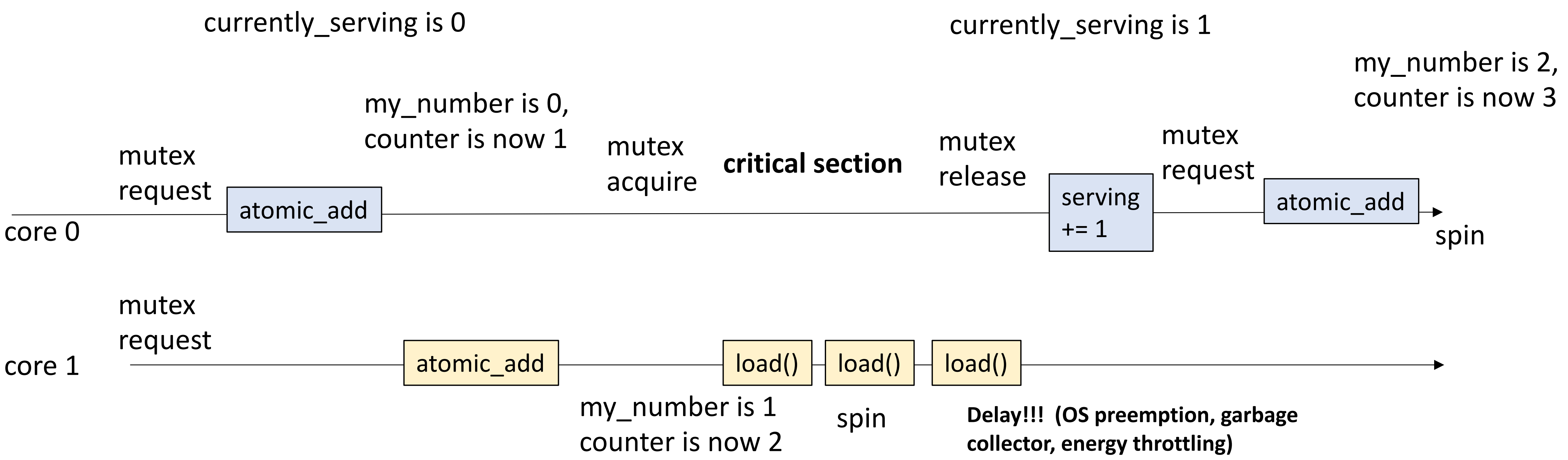


Analysis

Is this mutex starvation Free?

```
void lock() {
  int my_number = atomic_fetch_add(&counter, 1);
  while (currently_serving.load() != my_number);
}

void unlock() {
  int tmp = currently_serving.load();
  tmp += 1;
  currently_serving.store(tmp);
}
```

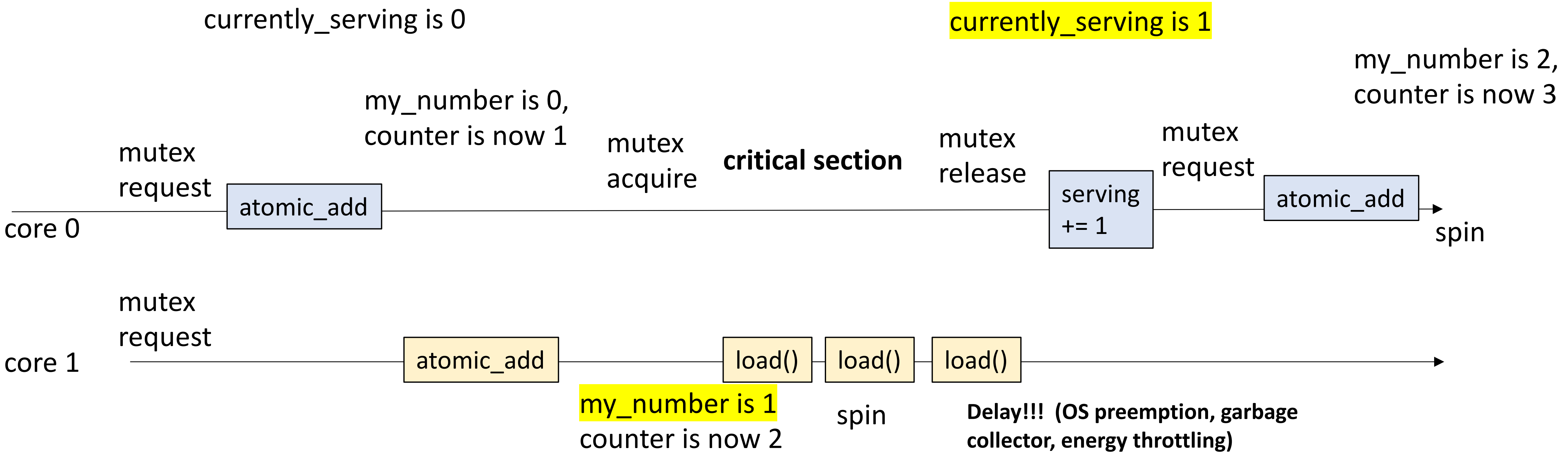


Analysis

Is this mutex starvation Free?

```
void lock() {
  int my_number = atomic_fetch_add(&counter, 1);
  while (currently_serving.load() != my_number);
}

void unlock() {
  int tmp = currently_serving.load();
  tmp += 1;
  currently_serving.store(tmp);
}
```

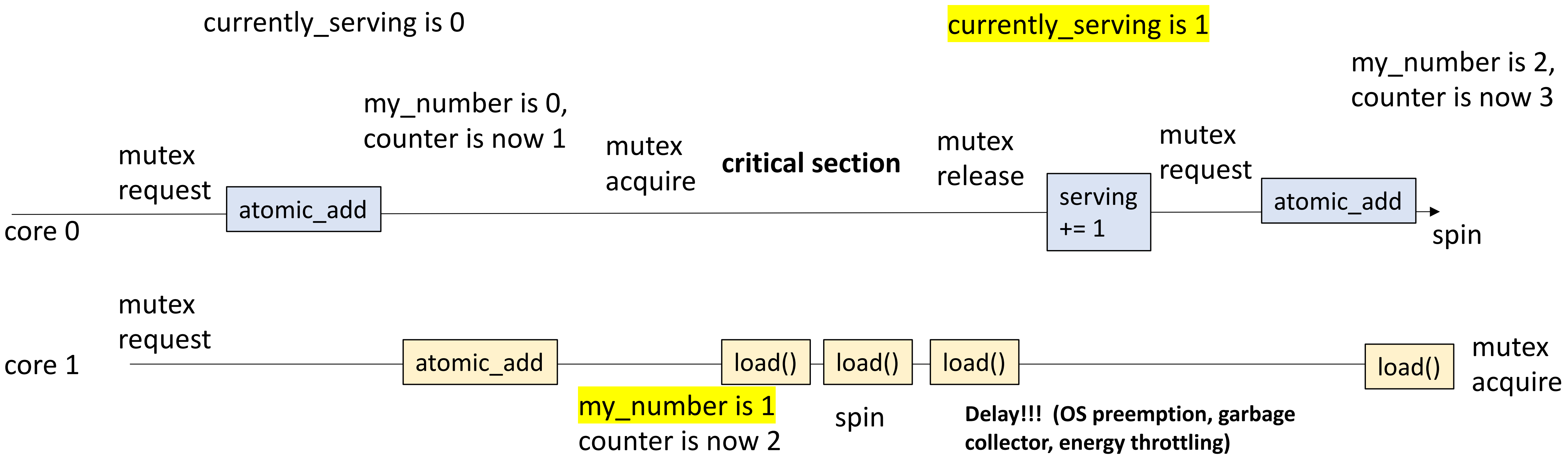


Analysis

Is this mutex starvation Free?

```
void lock() {
  int my_number = atomic_fetch_add(&counter, 1);
  while (currently_serving.load() != my_number);
}

void unlock() {
  int tmp = currently_serving.load();
  tmp += 1;
  currently_serving.store(tmp);
}
```



Fair but at what cost?

- Example

Lecture Schedule

- Atomic RMW mutexes
  - Exchange
  - CAS
  - Ticket

- Optimizations
  - Relaxed peeking
  - Backoff

Lecture Schedule

- Atomic RMW mutexes
  - Exchange
  - CAS
  - Ticket

- **Optimizations**
  - **Relaxed peeking**
  - Backoff

Optimizations: relaxed peeking

- Relaxed Peeking
  - the Writes in RMWs cost extra; rather than always modify, we can do a simple check first

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

bool try_lock() {
  bool e = false;
  return atomic_compare_exchange_strong(&flag, &e, true);
}
```

Optimizations: relaxed peeking

- Relaxed Peeking
  - the Writes in RMWs cost extra; rather than always modify, we can do a simple check first

```
void lock() {
  bool e = false;
  bool acquired = false;
  while (!acquired) {
    while (flag.load() == true);
    e = false;
    acquired = atomic_compare_exchange_strong(&flag, &e, true);
  }
}
```

Optimizations: relaxed peeking

- What about the load in the loop? Remember the memory fence? Do we need to flush our caches every time we peek?
- We only need to flush when we actually acquire the mutex

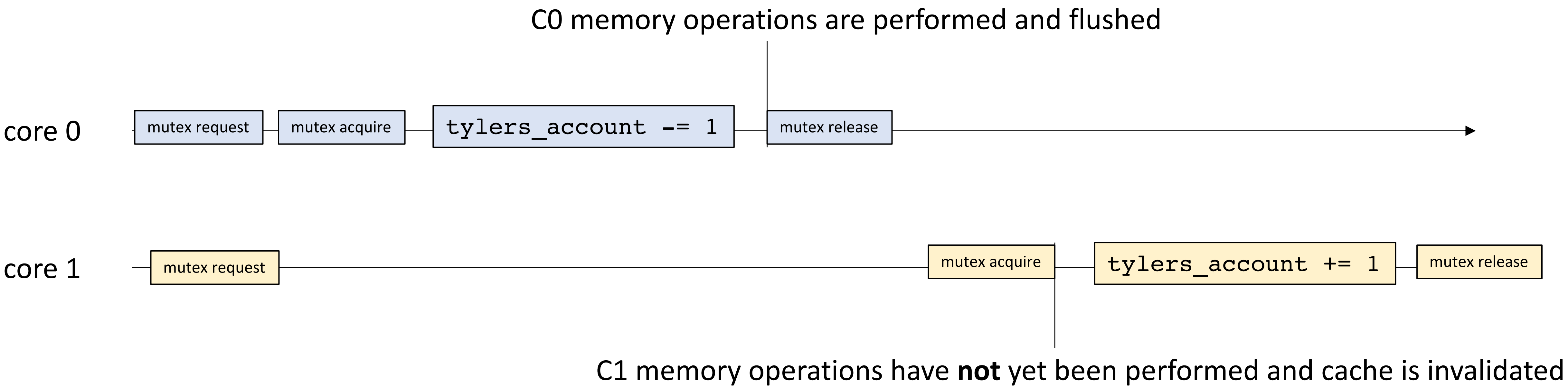
```
void lock() {
  bool e = false;
  bool acquired = false;
  while (!acquired) {
    while (flag.load() == true);
    e = false;
    acquired = atomic_compare_exchange_strong(&flag, &e, true);
  }
}
```

Optimizations: relaxed peeking

- What about the load in the loop? Remember the memory fence? Do we need to flush our caches every time we peek?
- We only need to flush when we actually acquire the mutex

```
void lock(int thread_id) {
  bool e = false;
  bool acquired = false;
  while (!acquired) {
    while (flag.load(memory_order_relaxed) == true);
    e = false;
    acquired = atomic_compare_exchange_strong(&flag, &e, true);
  }
}
```

```
void lock(int thread_id) {
  bool e = false;
  bool acquired = false;
  while (!acquired) {
    while (flag.load(memory_order_relaxed) == true);
    e = false;
    acquired = atomic_compare_exchange_strong(&flag, &e, true);
  }
}
```



Relaxed atomics

- Enter expert mode!
  - explicit atomics with relaxed semantics
  - Beware! they do not provide a memory fence!
  - Only use when a memory fence is issued later before leaving your mutex implementation. Good for “peeking” before you actually execute your RMW.

Demo

- Example in terminal