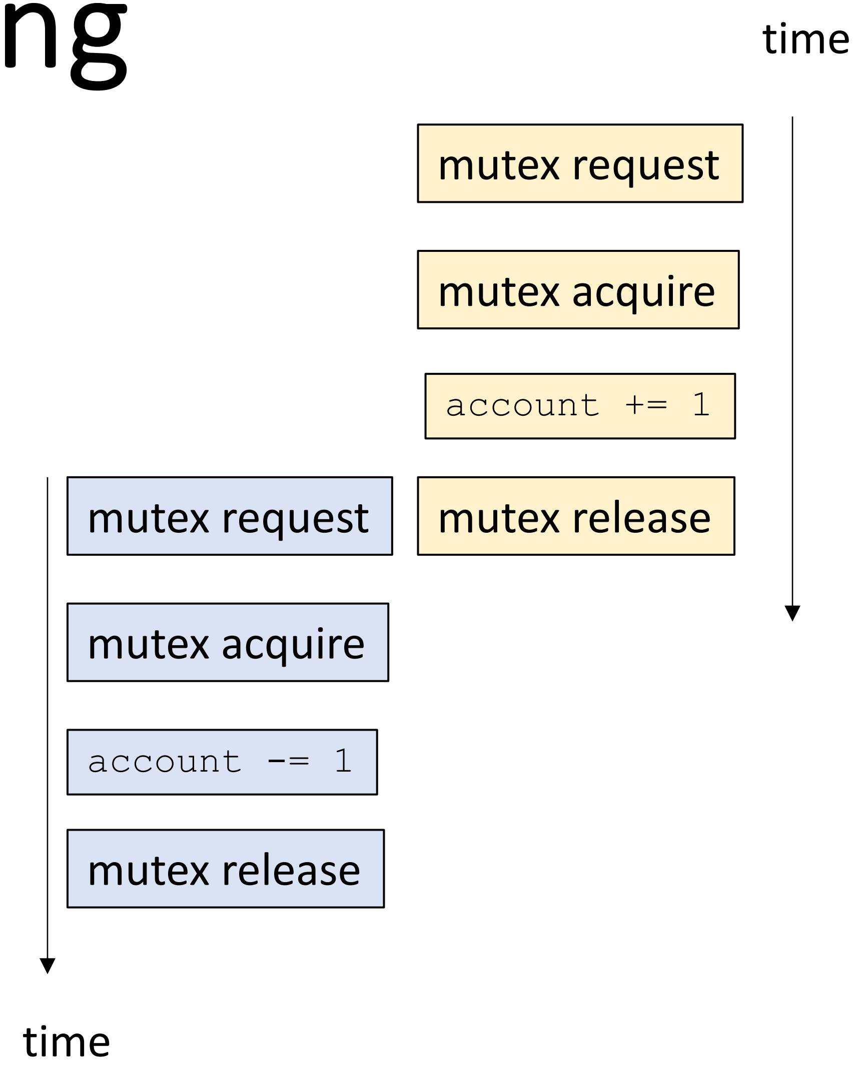
CSE113: Parallel Programming

- **Topics**:
  - RMW mutex implementations



Announcements

- Third lecture in Module 2
- HW 1 should be in today.
- HW 2 was last Thursday. You can start on part 1, probably part 2 by end of today.
- Office hours available. Start early!

Announcements

- Office hour etiquette
  - Some people are going without help. Let’s do round robin.

Announcements

- Midterm is in next week, Oct 22
  - In-person test
  - 3 pages of notes front and back (but no memorization questions)
  - 10% of your grade
  - 5 or 6 short answer questions

Previous quiz + review

Previous quiz

Which one of the answers is NOT a property of mutexes?

- ○ Deadlock Freedom
- ○ Mutual Exclusion
- ○ Deterministic Execution
- ○ Starvation Freedom

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.

Previous quiz

We should aim to make mutual exclusion regions as short as possible because of the caching overhead of locks.

- ○ True
- ○ False

Mutex Performance

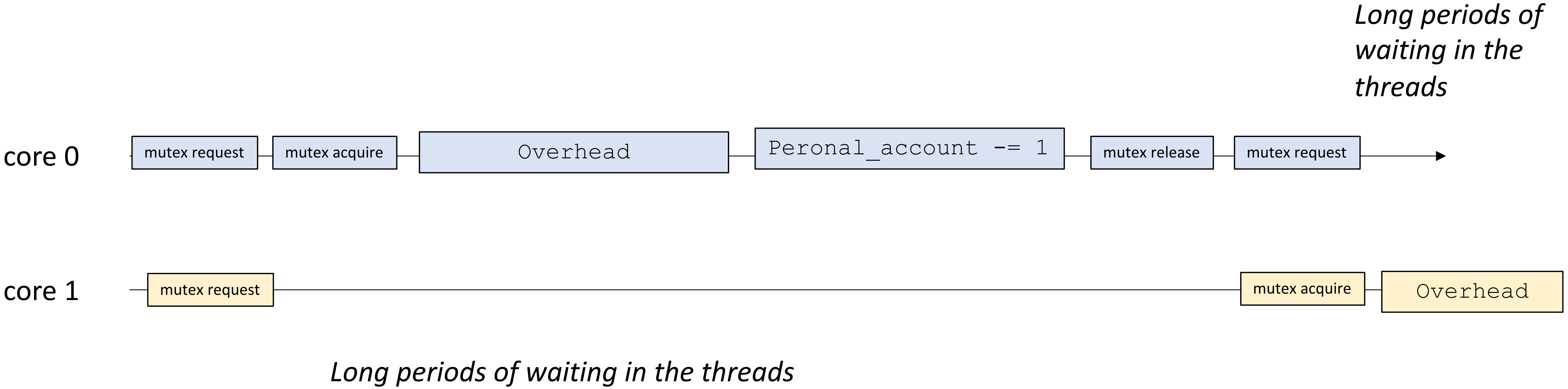
Try to keep mutual exclusion sections small!

Code example with overhead

Mutex Performance

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

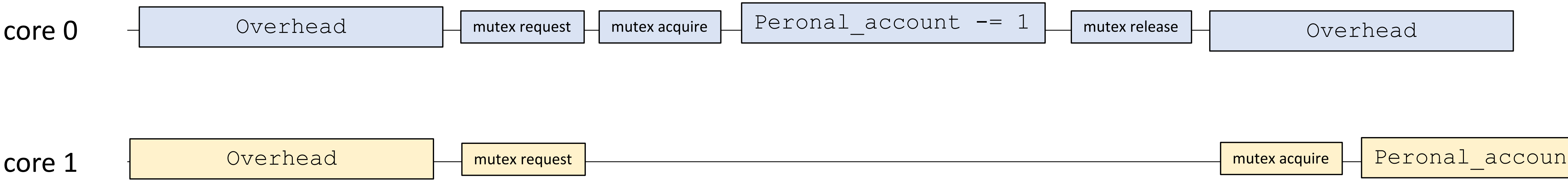
Code example with overhead



Mutex Performance

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

Code example with overhead



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

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

- ○ True
- ○ False

Previous quiz

It is required to use atomic types inside of critical sections

- ○ True
- ○ False

Our primitive instructions

- Types: atomic_int

- Interface (C++ provides overloaded operators):
  - load
  - store

- Properties:
  - loads and stores will always go to memory.
  - compiler memory fence
  - hardware memory fence

Previous quiz

Write 1 or 2 sentences about whether you agree or disagree with the following sentence and why:

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

Atomic properties

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

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

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

Previous quiz

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

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 be victim

Mutex Implementations

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

mark ourselves as uninterested

previous flag issue

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

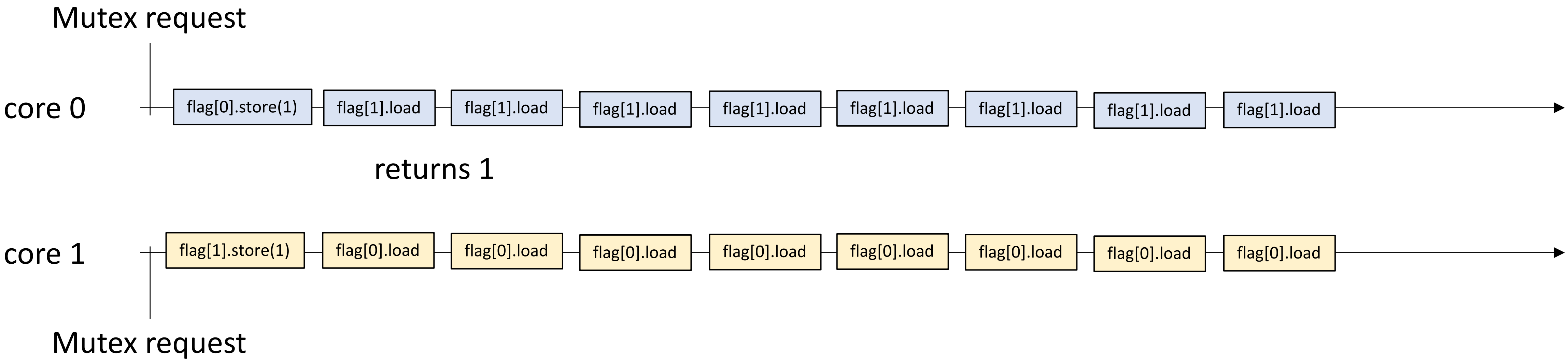
```
void unlock() {
  int i = thread_id;
  flag[i].store(0);
}
```

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

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

how does petersons solve this?

Both will spin forever!



Tie breaking with victim

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

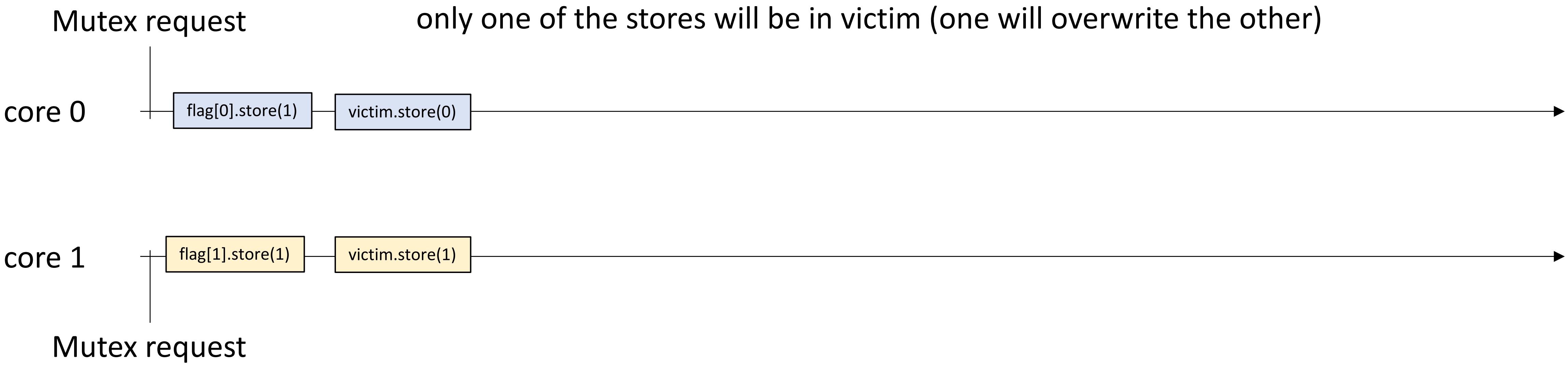
```
void unlock() {
  int i = thread_id;
  flag[i].store(0);
}
```

Thread 0:

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

Thread 1:

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



Tie breaking with victim

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

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

Thread 0:

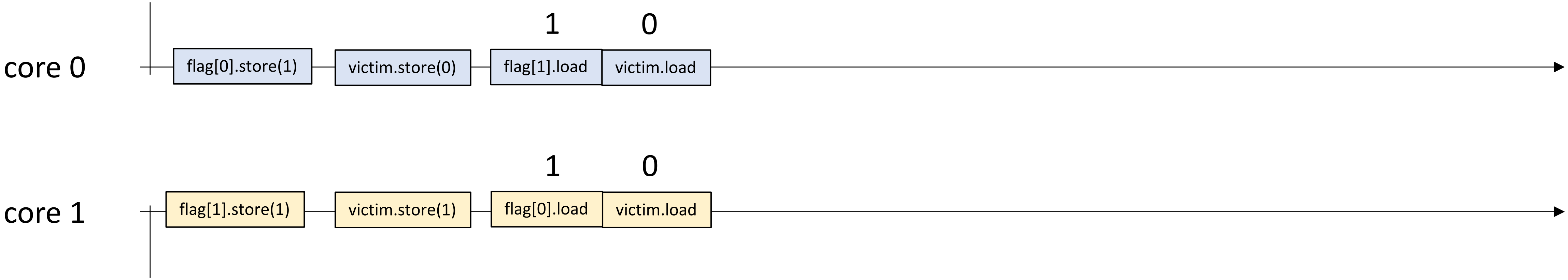
```
m.lock();
m.unlock();
```

Thread 1:

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

Mutex request

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



Mutex request

Tie breaking with victim

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

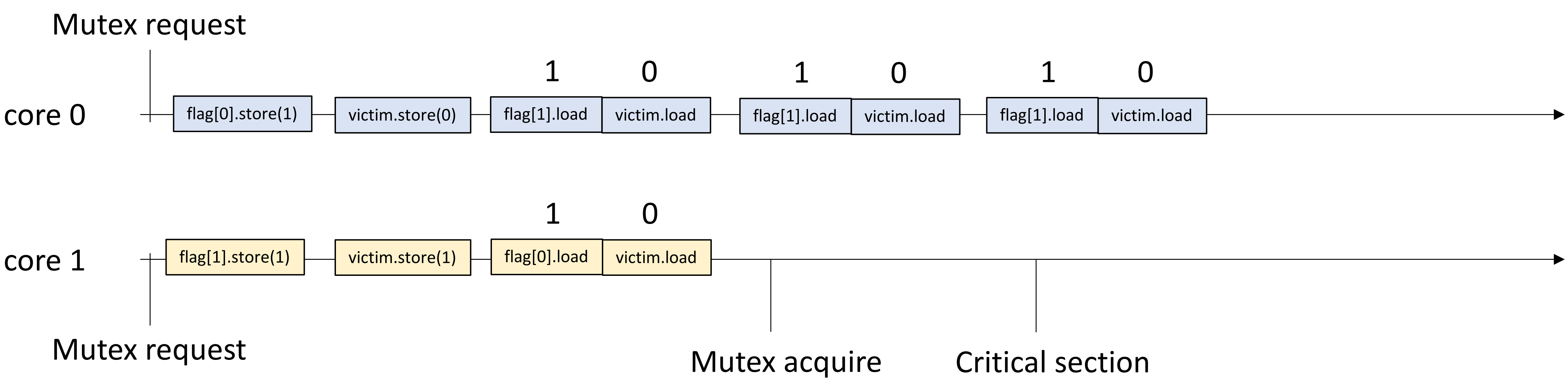
```
void unlock() {
  int i = thread_id;
  flag[i].store(0);
}
```

Thread 0:

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

Thread 1:

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



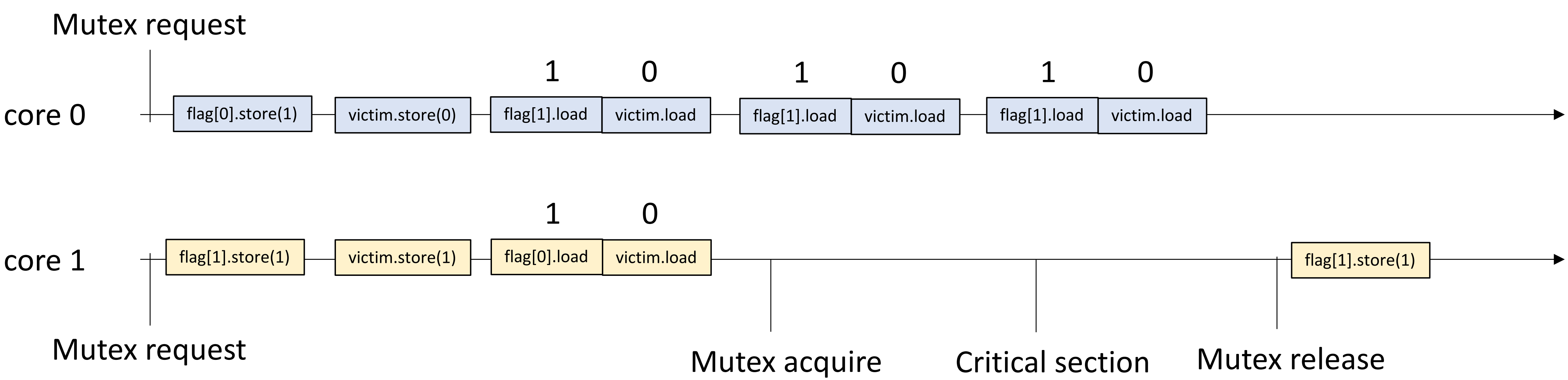
Tie breaking with victim

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

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

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

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



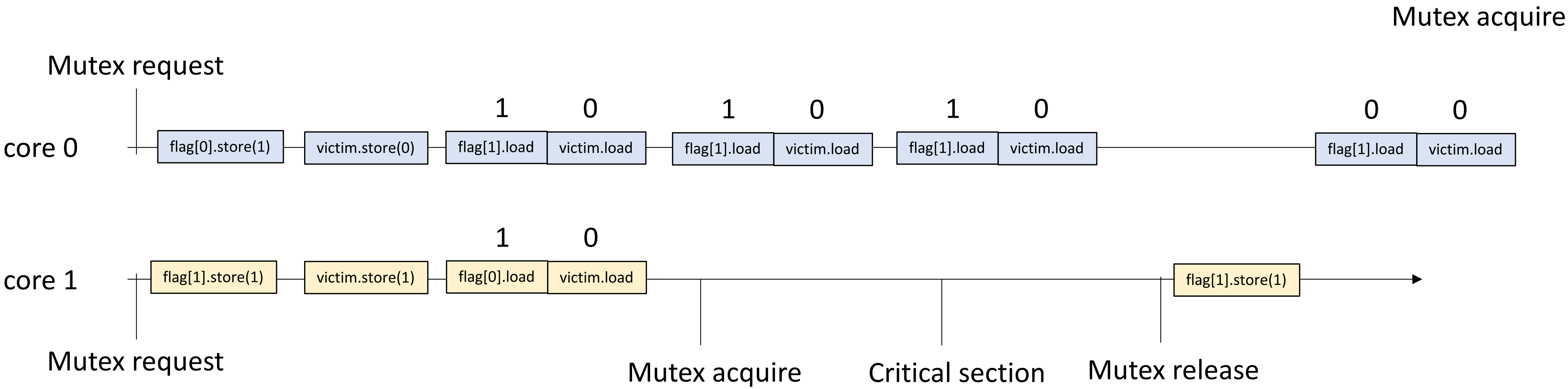
Tie breaking with victim

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

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

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

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



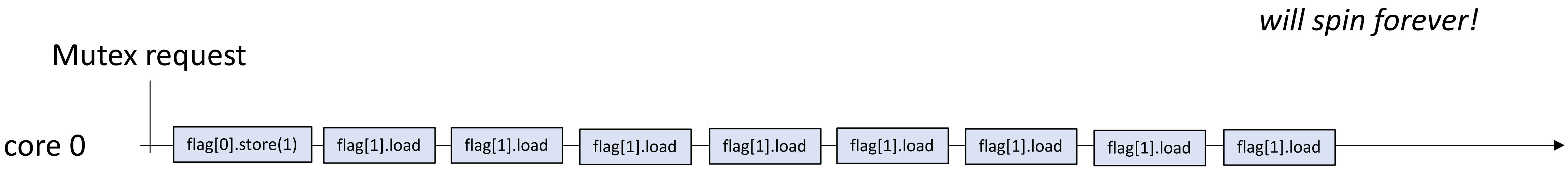
previous victim issue

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

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

Thread 0:

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



previous flag issue

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

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

Thread 0:

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

Mutex request

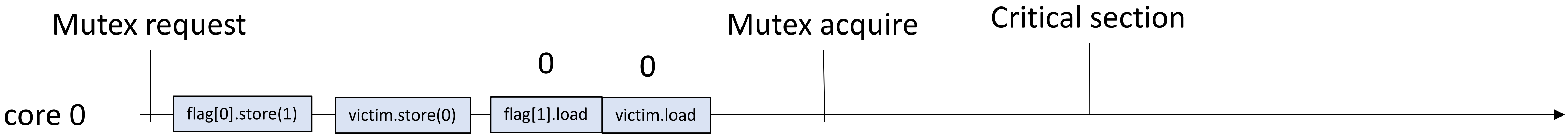
previous flag issue

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

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

Thread 0:

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



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

New material

Historical perspective

- These locks are academically interesting: they can be implemented with plain loads and stores

- However, they are not very performant compared to modern solutions
  - Your HW will show this

- 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

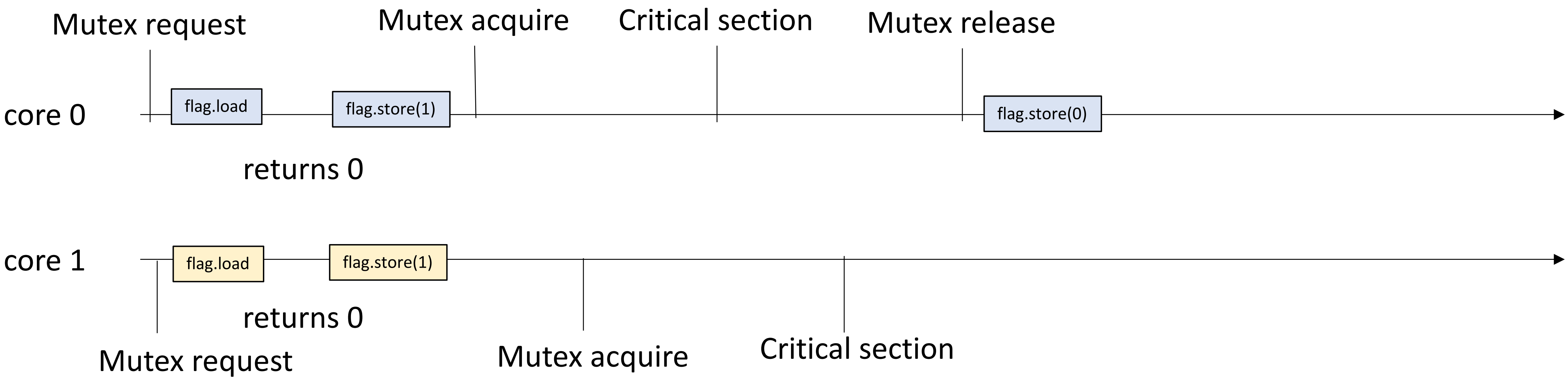
```
void lock() {
  while (flag.load() == 1);
  flag.store(1);
}
```

```
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_fetch_add

Recall the lock free account

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

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

lock-free accounts

Tyler's coffee addiction:

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

Tyler's employer

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

time

time

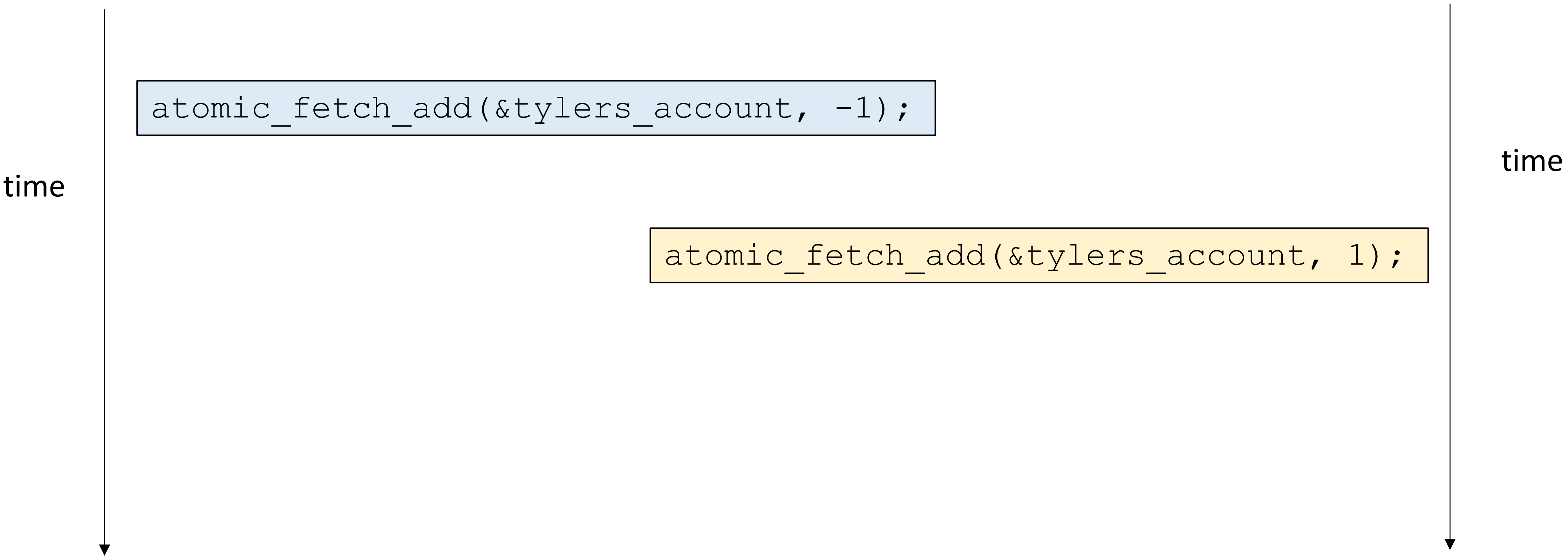
lock-free accounts

Tyler's coffee addiction:

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

Tyler's employer

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



lock-free accounts

Tyler's coffee addiction:

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

Tyler's employer

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

time

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

time

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

lock-free accounts

Tyler's coffee addiction:

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

Tyler's employer

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

time

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

cannot interleave!

time

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

lock-free accounts

Tyler's coffee addiction:

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

Tyler's employer

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

time

cannot interleave!

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

time

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

either way, account breaks even at the end!

atomic_fetch_add

Recall the lock free account

```
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
}
```

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.

They operate on atomic types.

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

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

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

  void lock();
  void unlock();

private:
  atomic_bool flag;
};
```

Lets make a mutex with just one atomic bool!

First example: Exchange Lock

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

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

  void lock();
  void unlock();

private:
  atomic_bool flag;
};
```

Lets make a mutex with just one atomic bool!

initialized to `false`

one atomic flag

First example: Exchange Lock

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

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

  void lock();
  void unlock();

private:
  atomic_bool flag;
};
```

Lets make a mutex with just one atomic bool!

initialized to `false`

one atomic flag

main idea:

The flag is false when the mutex is free.

The flag is true when some thread has the mutex.

First example: Exchange Lock

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

First example: Exchange Lock

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

So what’s going on?

First example: Exchange Lock

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

So what's going on?

Two cases:

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

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

First example: Exchange Lock

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

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

Analysis

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

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

Thread 0:

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

Thread 1:

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

core 0

core 1

Analysis

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

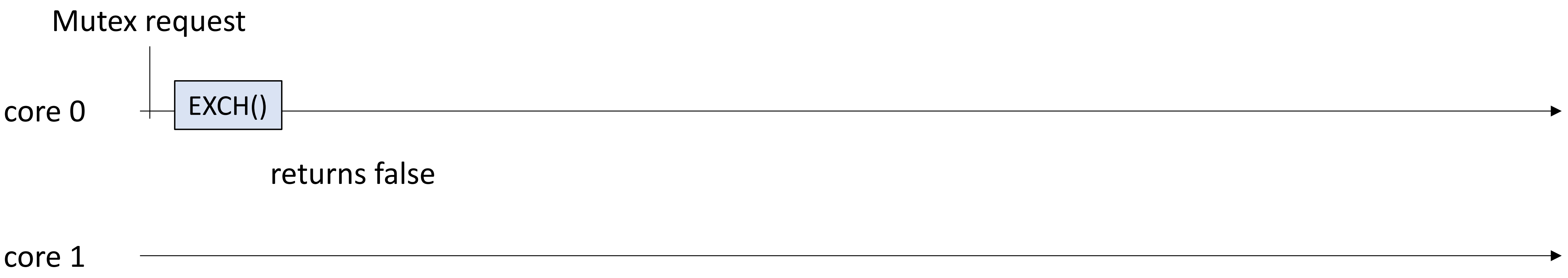
```
void unlock() {
  flag.store(false);
}
```

Thread 0:

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

Thread 1:

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



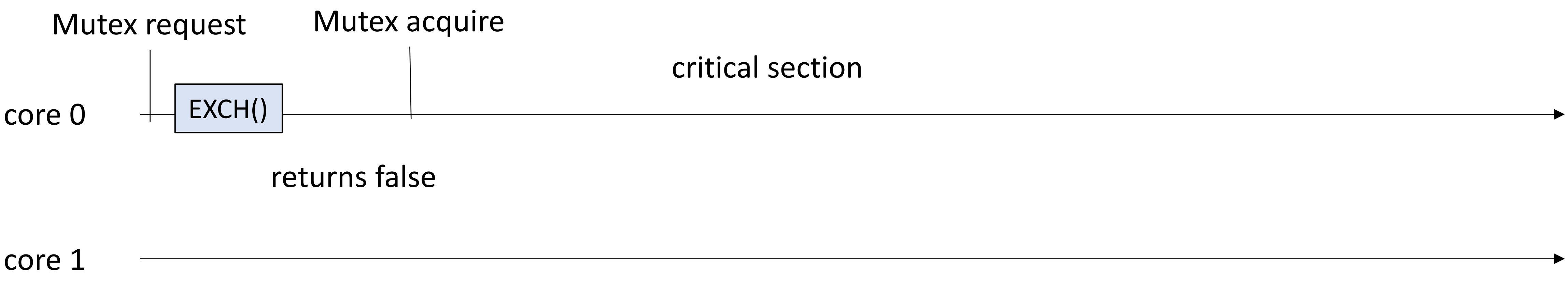
Analysis

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

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

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

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



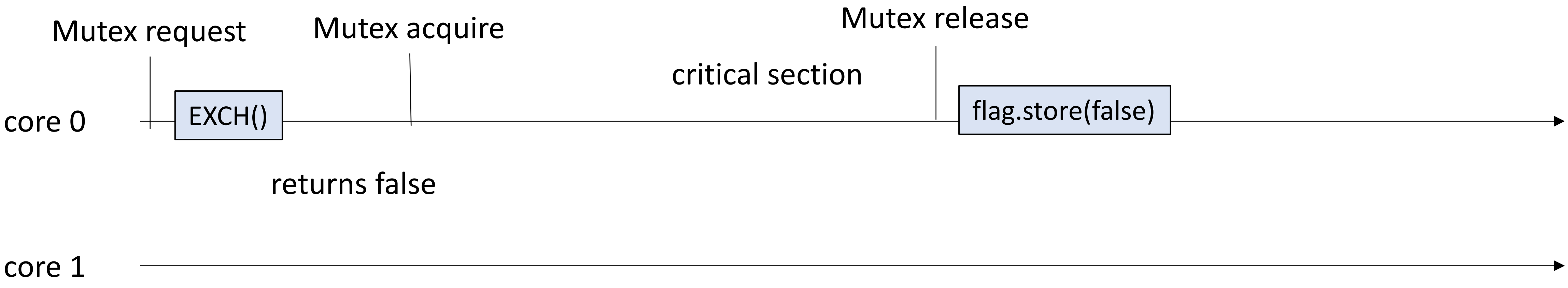
Analysis

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

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

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

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



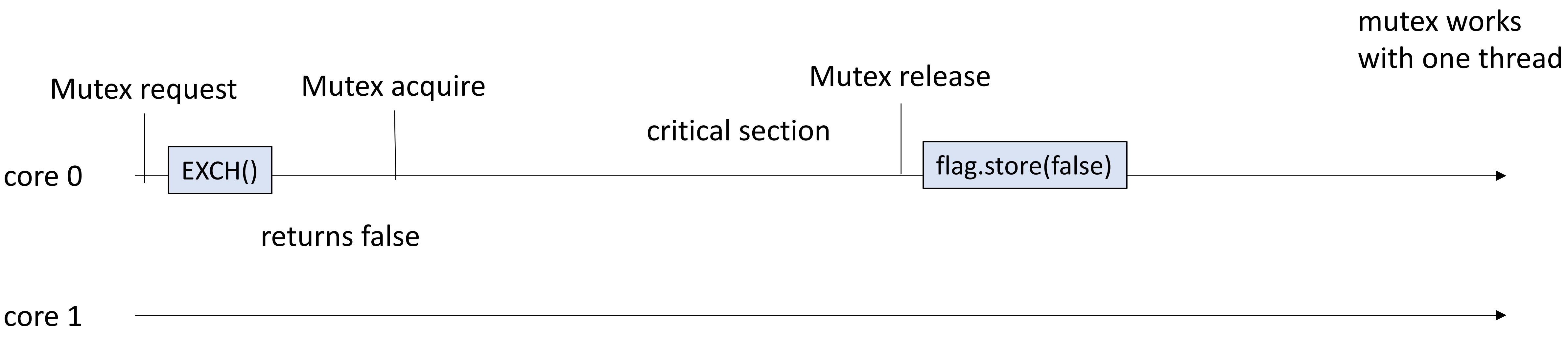
Analysis

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

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

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

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



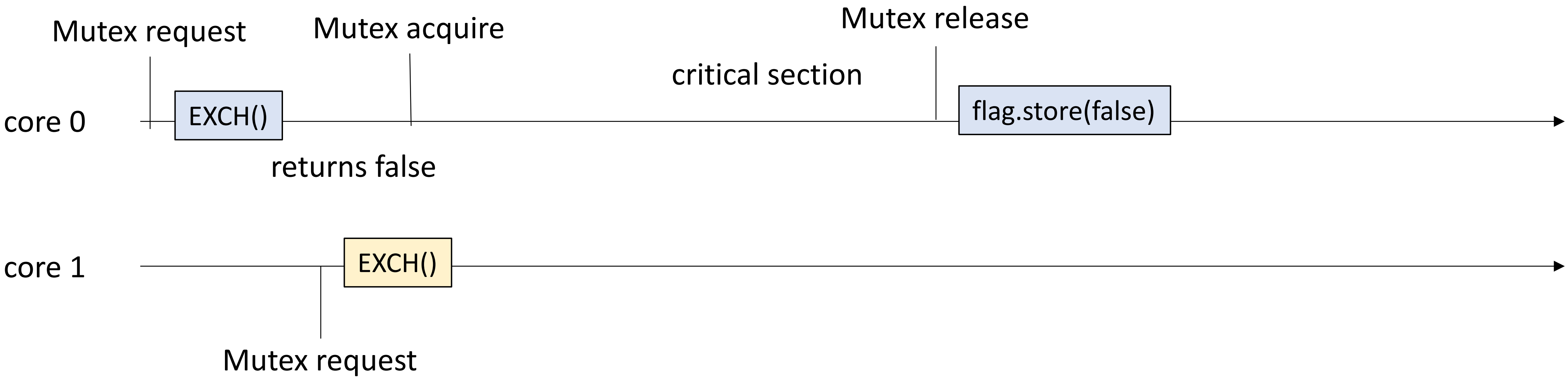
Analysis

```
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();
```



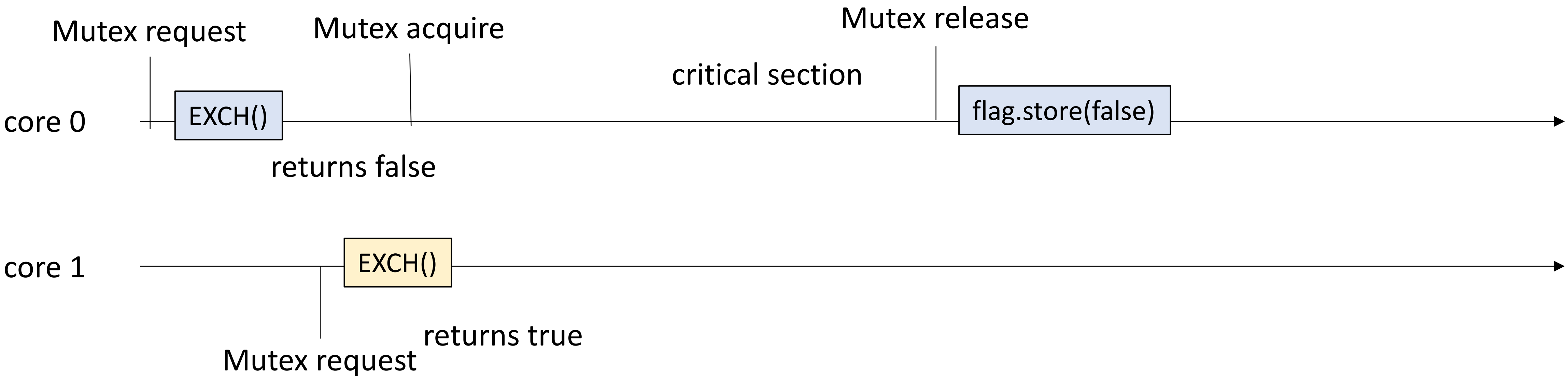
Analysis

```
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();
```



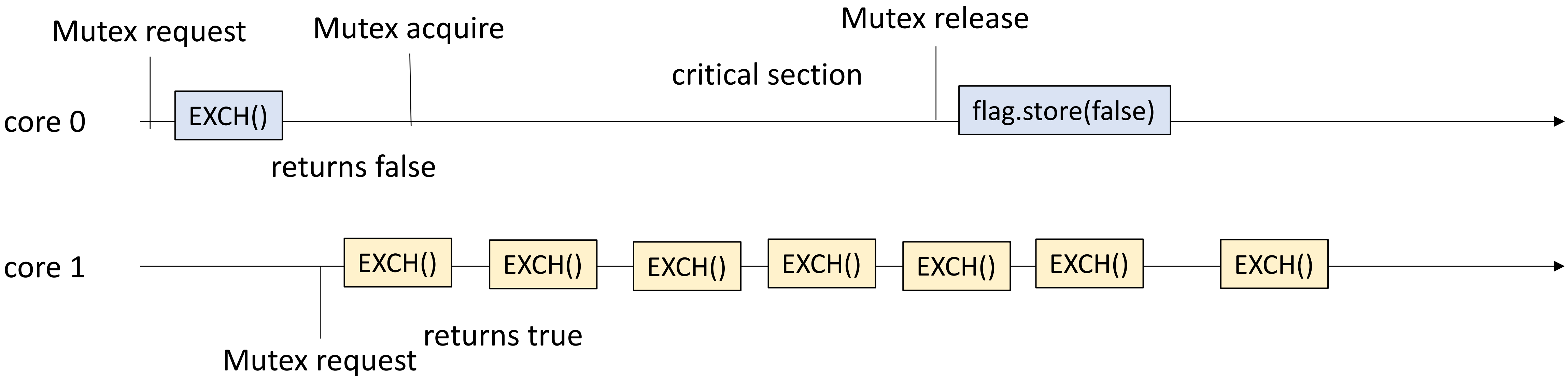
Analysis

```
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();
```



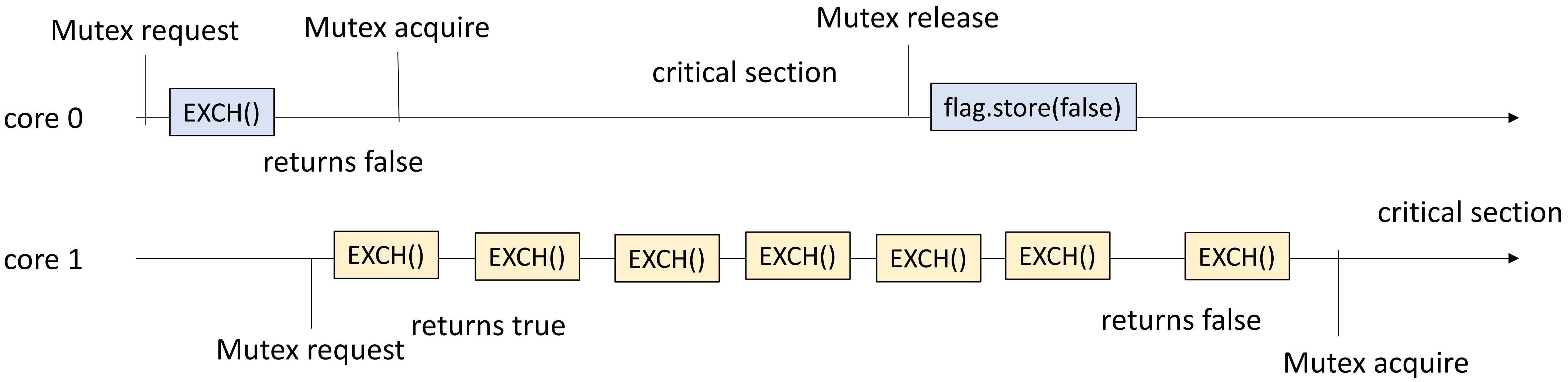
Analysis

```
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

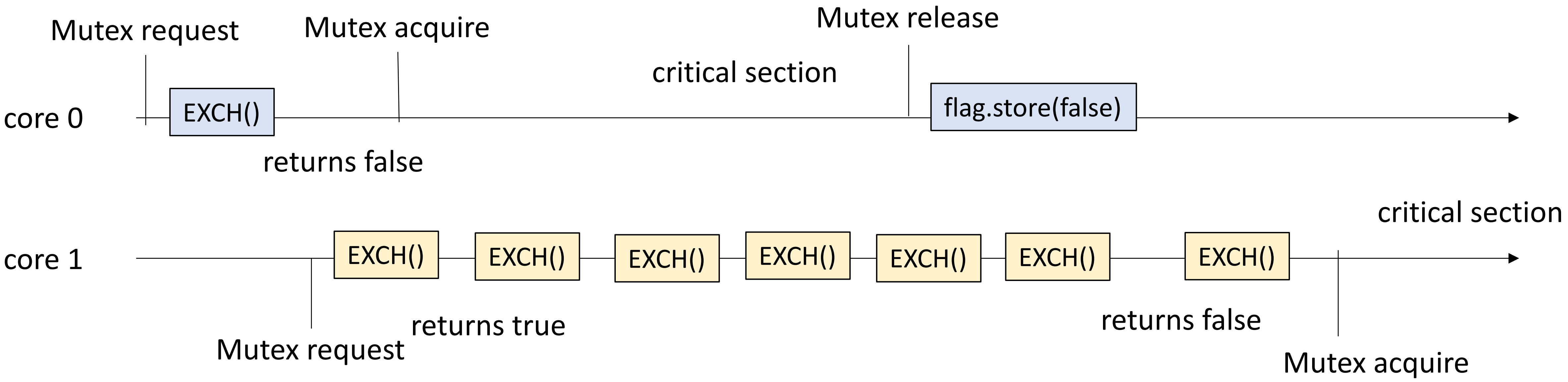
```
void lock() {
  while (atomic_exchange(&flag, true) == true);
}
```

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

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

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

what about interleavings?

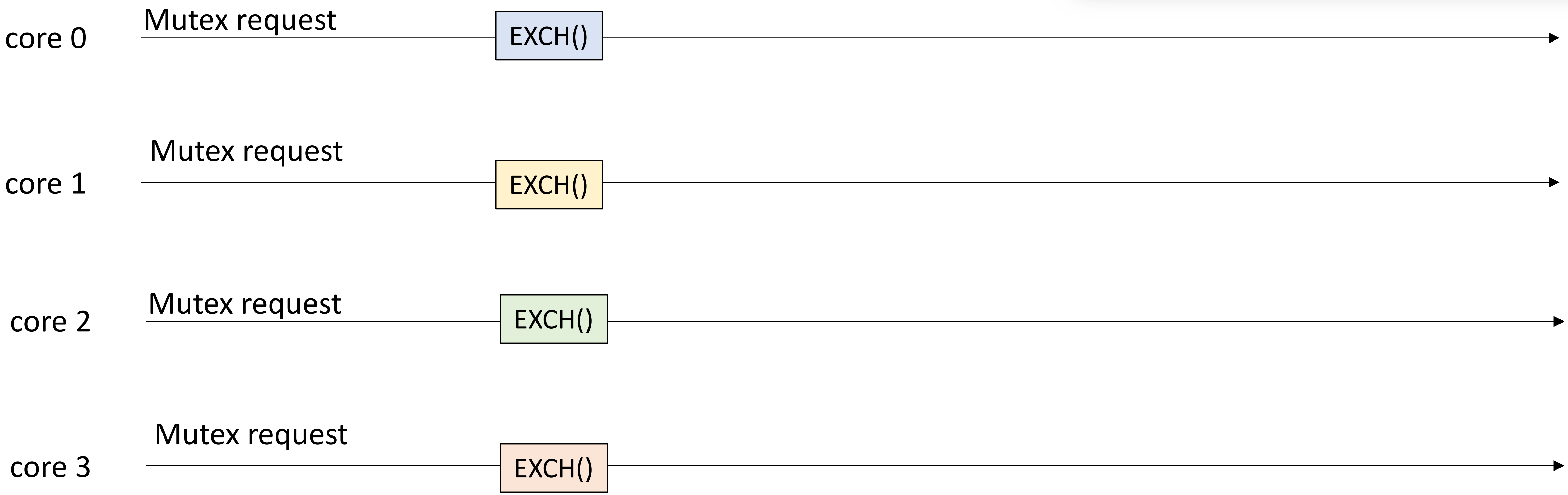


Analysis

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

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

what about 4 threads?



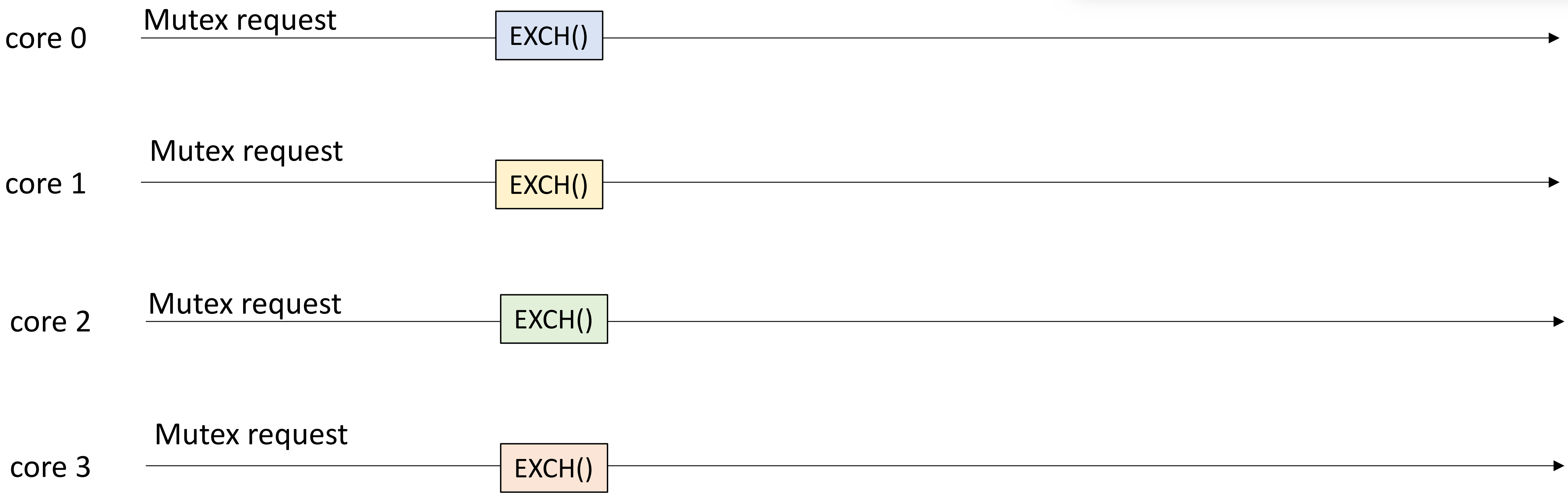
Analysis

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

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

what about 4 threads?

atomic operations can't overlap



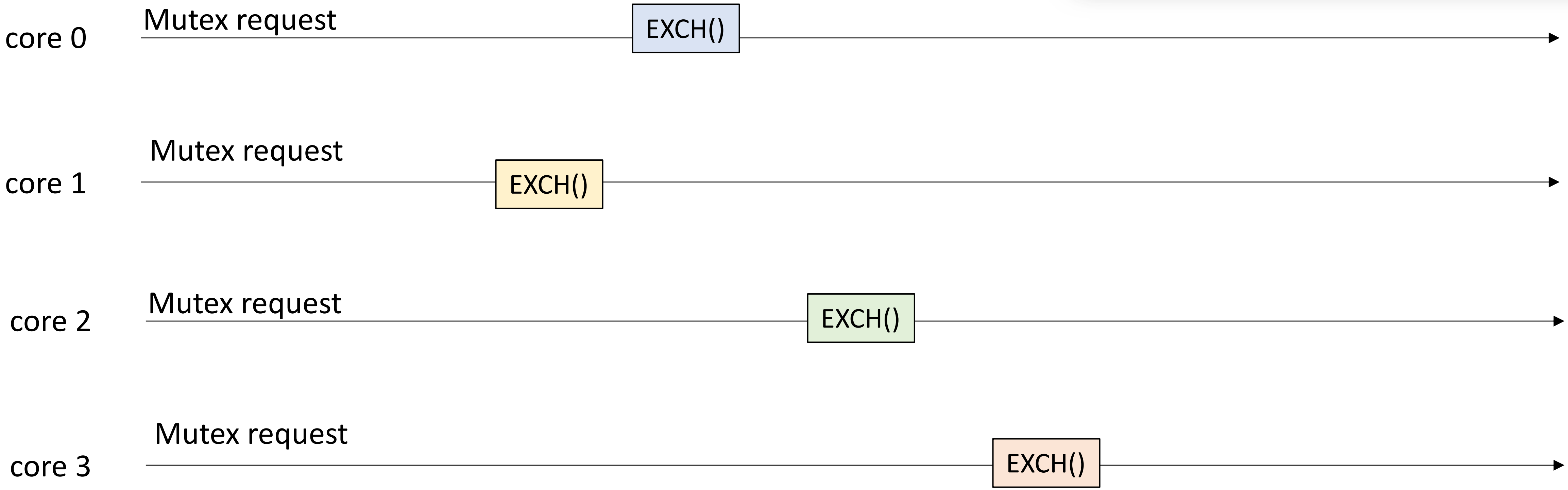
Analysis

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

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

what about 4 threads?

atomic operations can't overlap



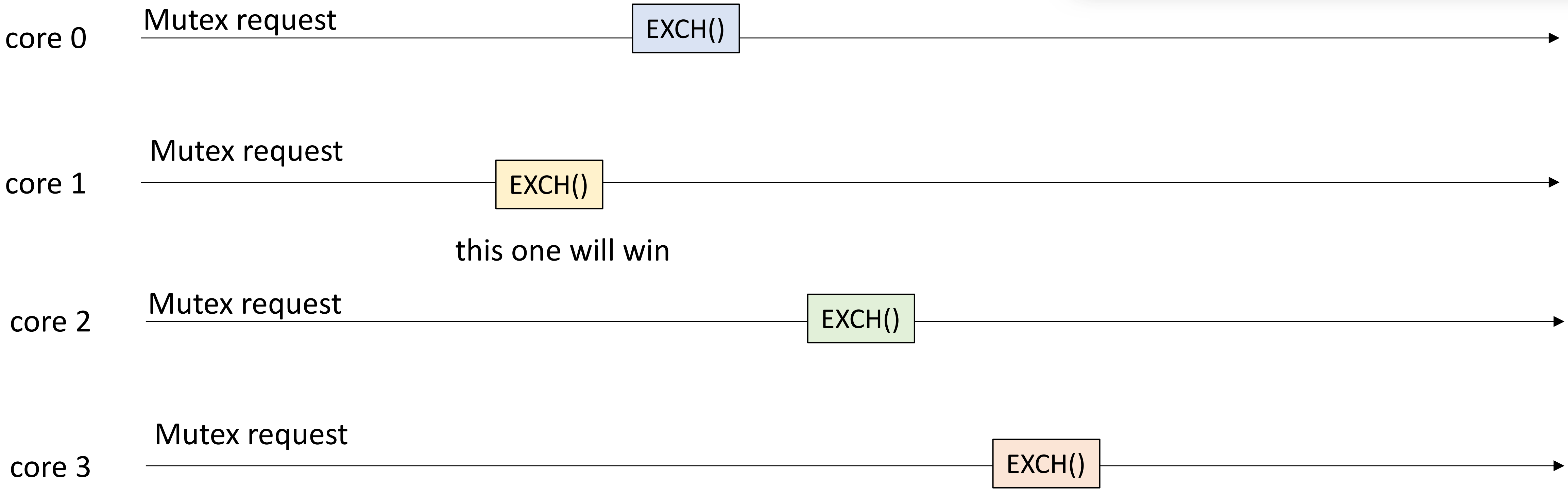
Analysis

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

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

what about 4 threads?

atomic operations can’t overlap



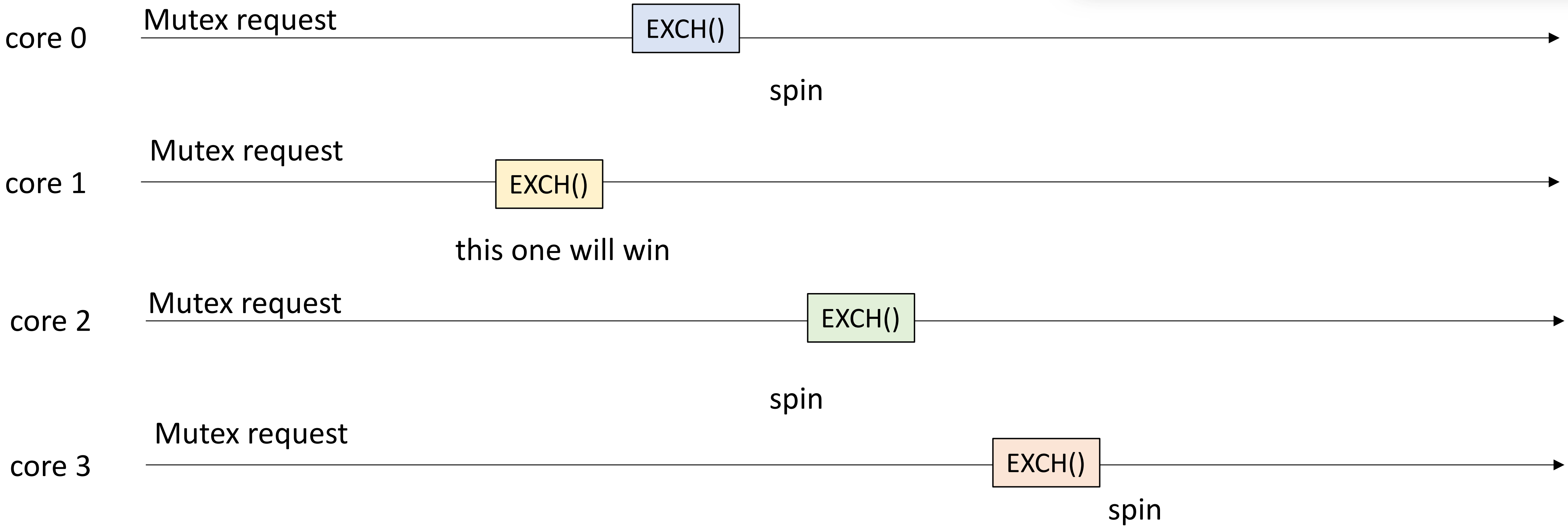
Analysis

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

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

what about 4 threads?

atomic operations can't overlap



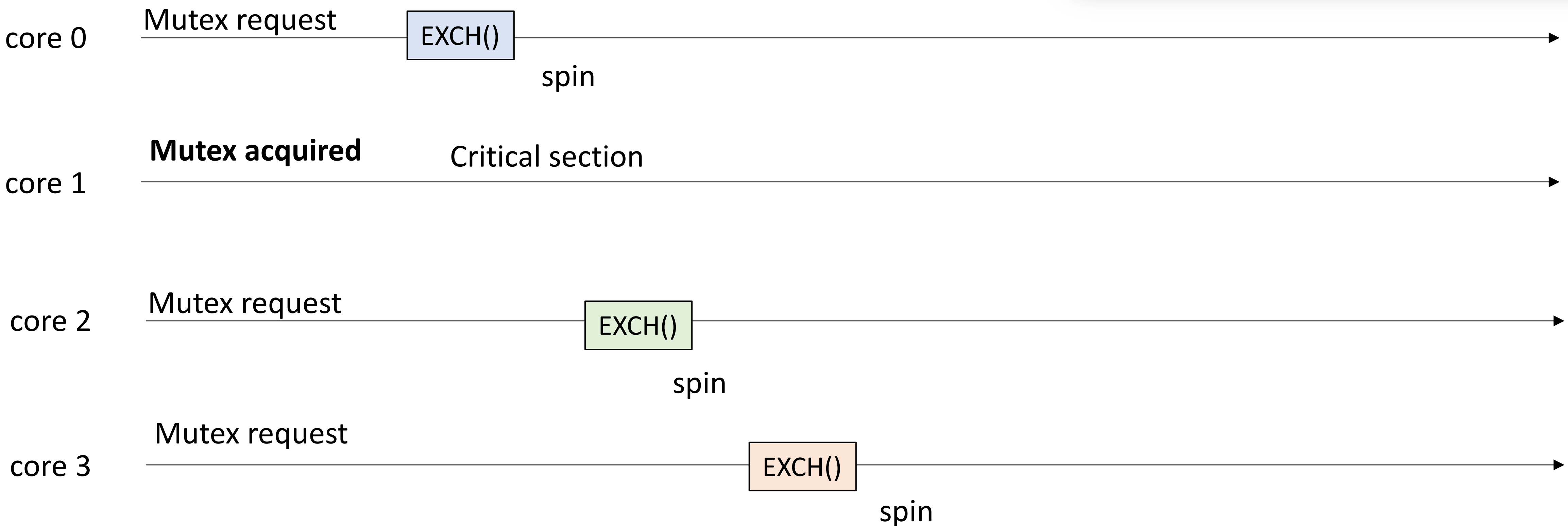
Analysis

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

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

what about 4 threads?

atomic operations can't overlap



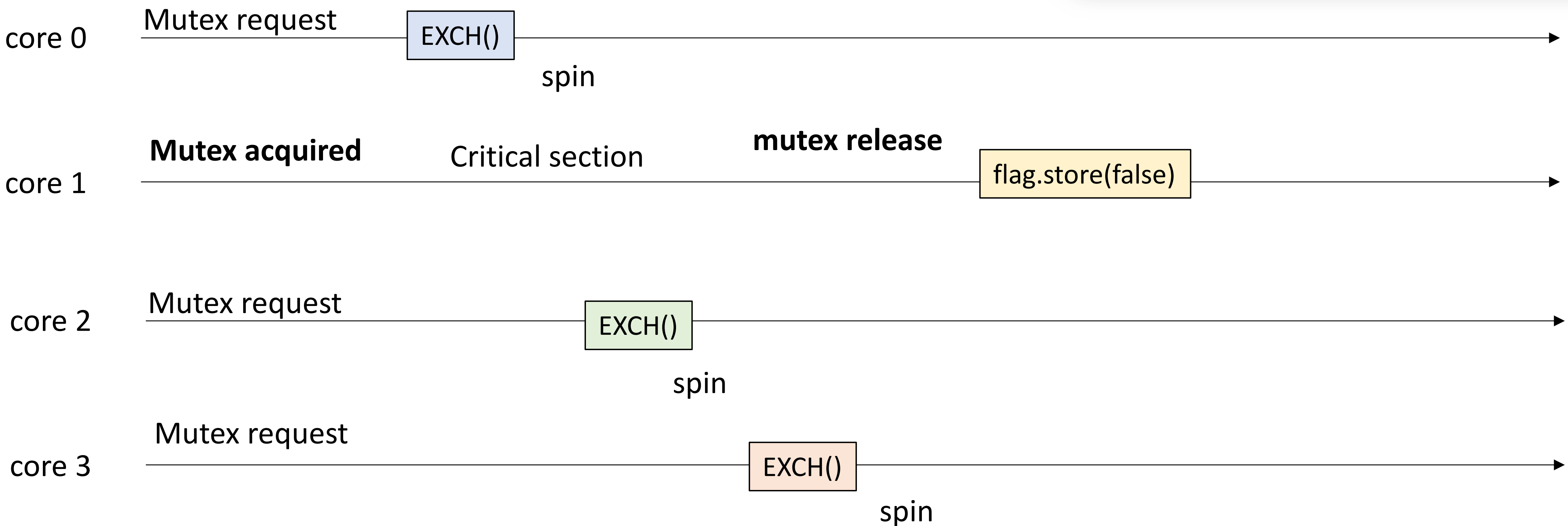
Analysis

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

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

what about 4 threads?

atomic operations can't overlap



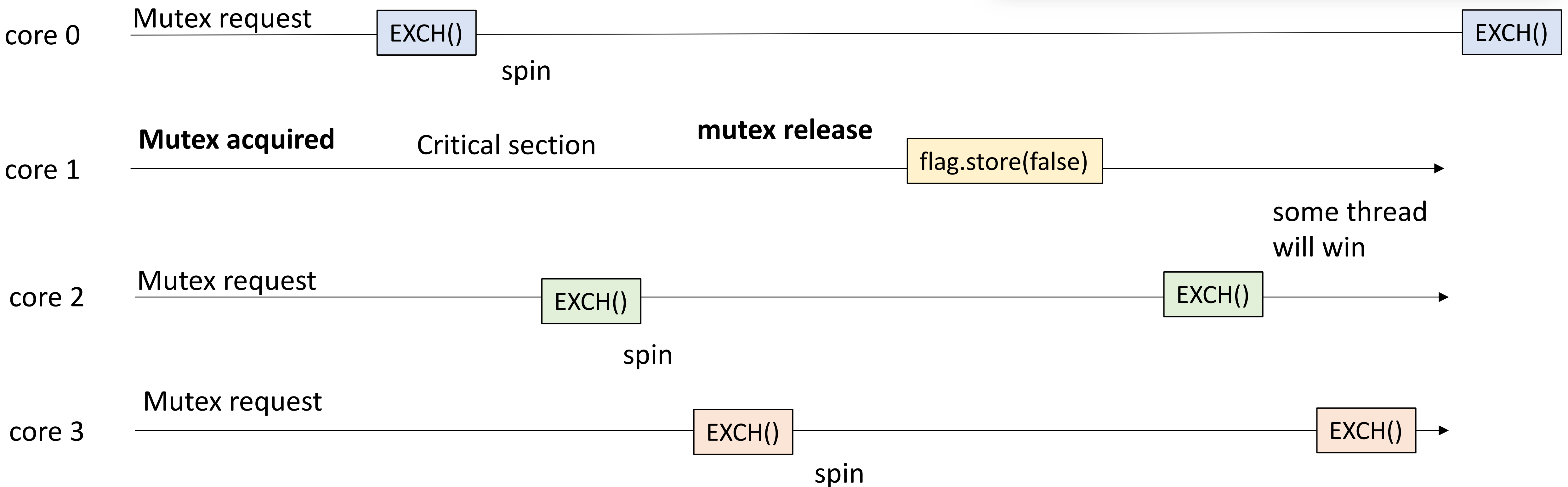
Analysis

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

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

what about 4 threads?

atomic operations can't overlap



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)

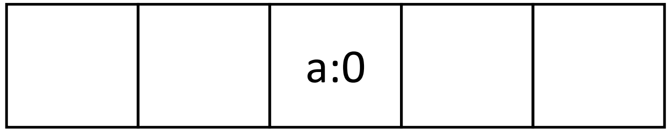
```
bool atomic_compare_exchange_strong(atomic *a, value *expected, value replace) {
   value tmp = a.load();
   if (tmp == *expected) {
     a.store(replace);
     return true;
   }
   *expected = tmp;
   return false;
}
```

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;
}
```

<u>thread 0:</u>

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

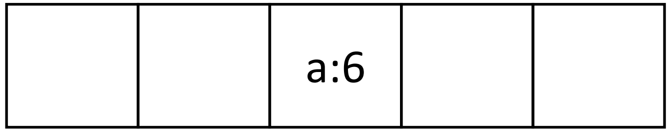
Most versatile RMW: Compare-and-swap

```
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;
}
```

<u>thread 0:</u>

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

true



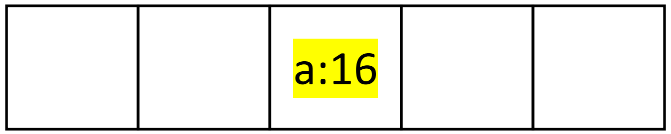
Most versatile RMW: Compare-and-swap

```
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);
```



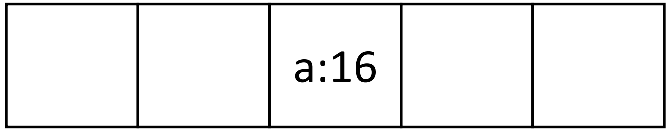
Most versatile RMW: Compare-and-swap

```
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;
}
```

<u>thread 0:</u>

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

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;
  }
}
```

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

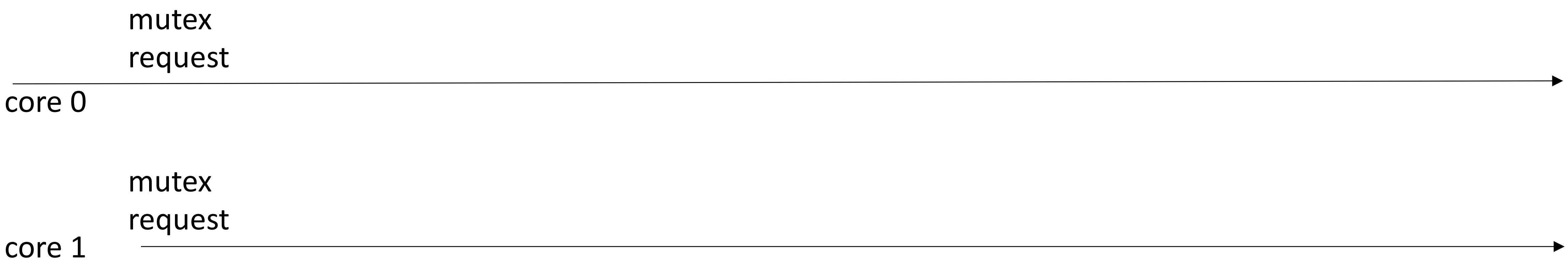
- Are these RMW locks fair?

Analysis

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

Is this mutex starvation Free?

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

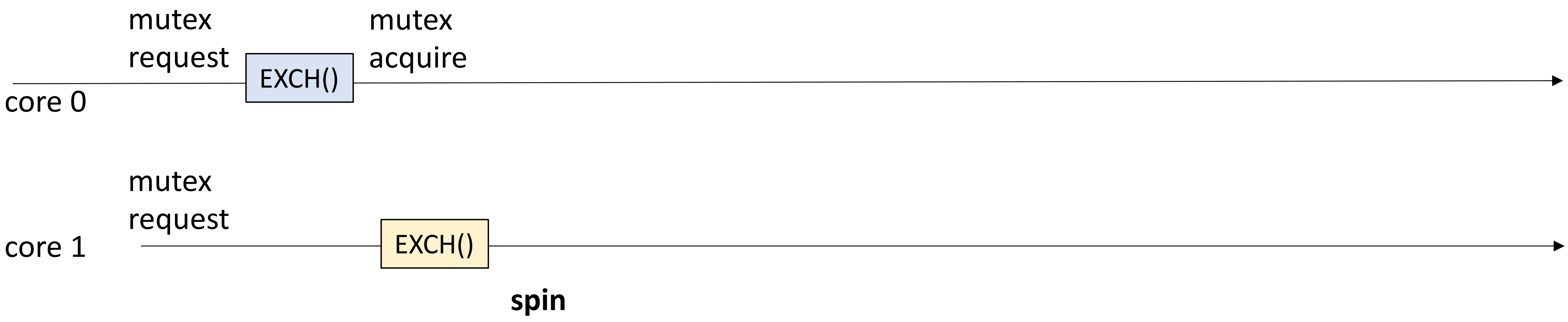


Analysis

Is this mutex starvation Free?

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

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

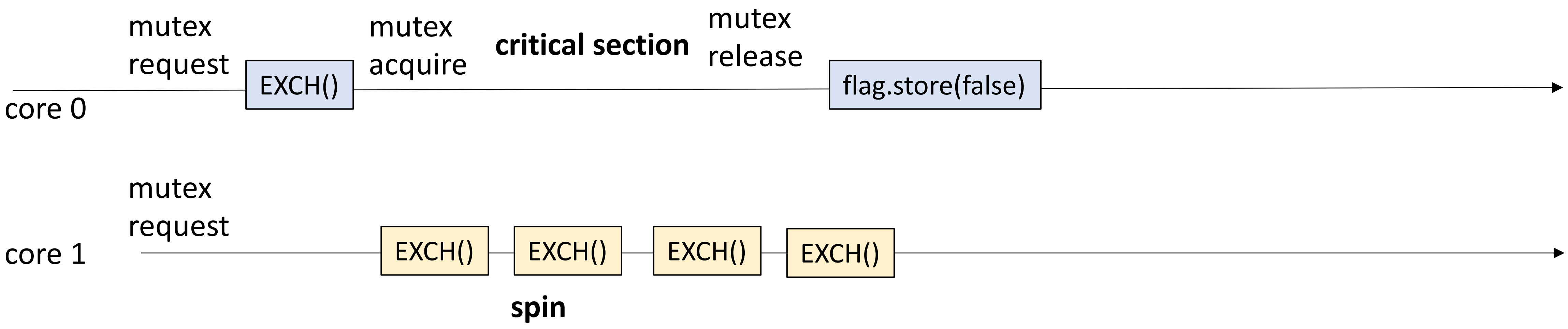


Analysis

Is this mutex starvation Free?

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

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

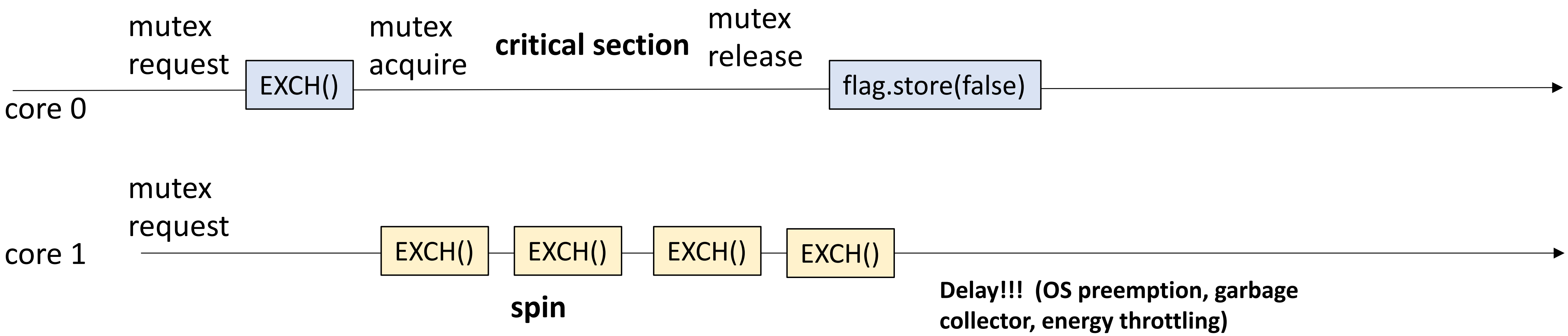


Analysis

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

Is this mutex starvation Free?

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

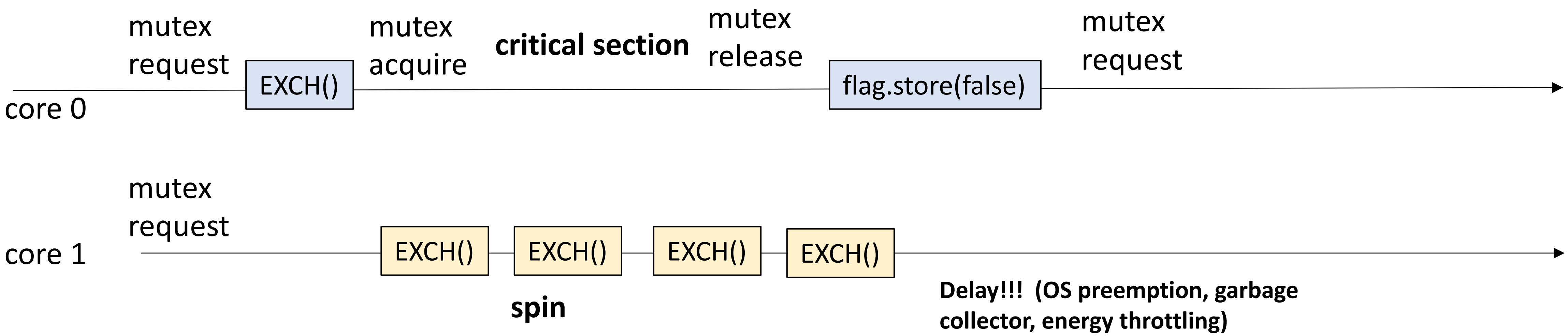


Analysis

Is this mutex starvation Free?

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

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

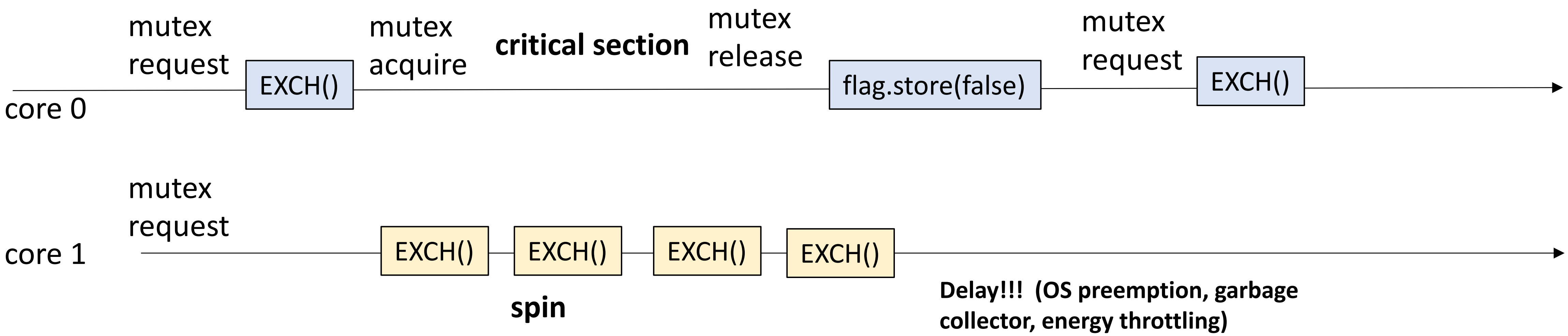


Analysis

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

Is this mutex starvation Free?

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

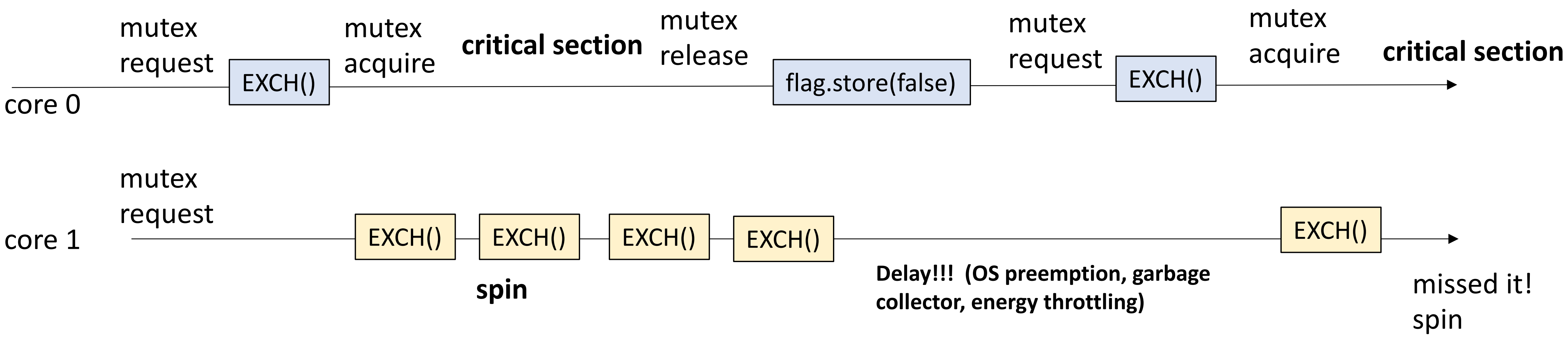


Analysis

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

Is this mutex starvation Free?

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



How about in practice?

- Code demo

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!

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

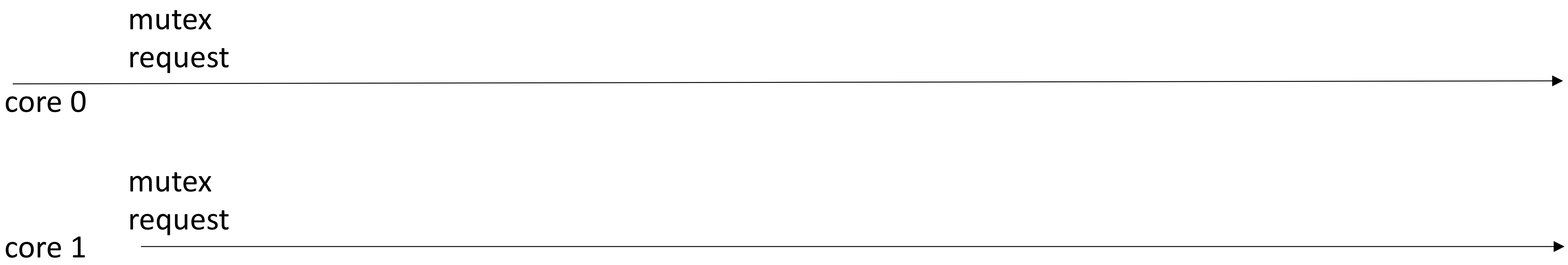
To release, increment the number that's currently being served.

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);
}
```

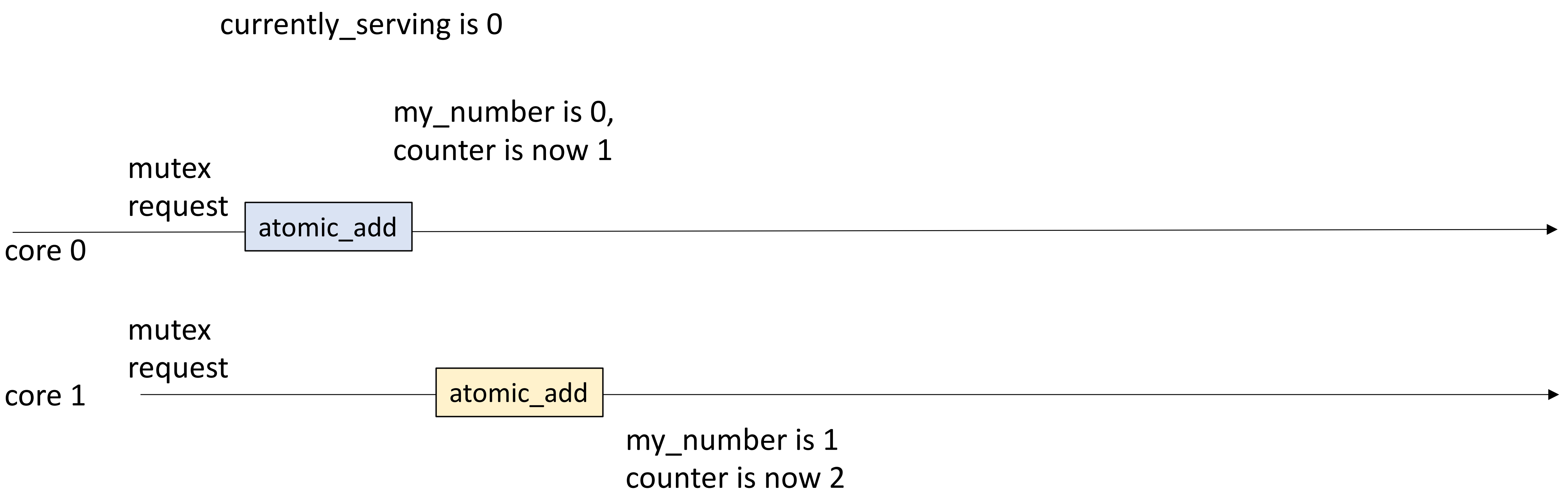


Analysis

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);
}
```

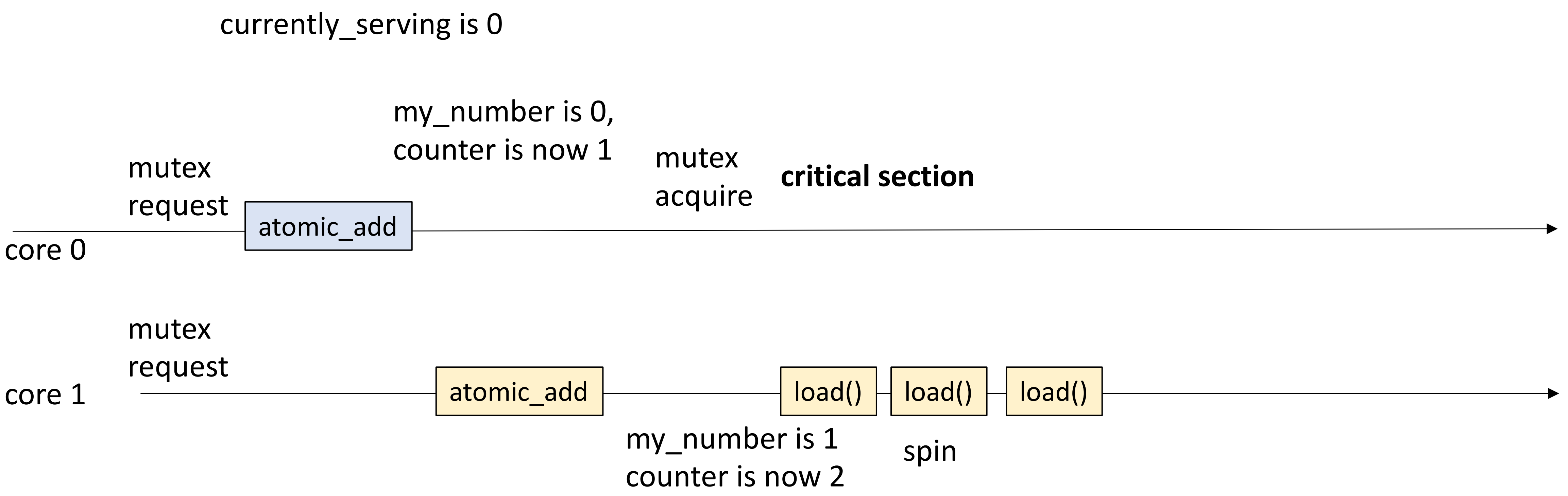


Analysis

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);
}
```

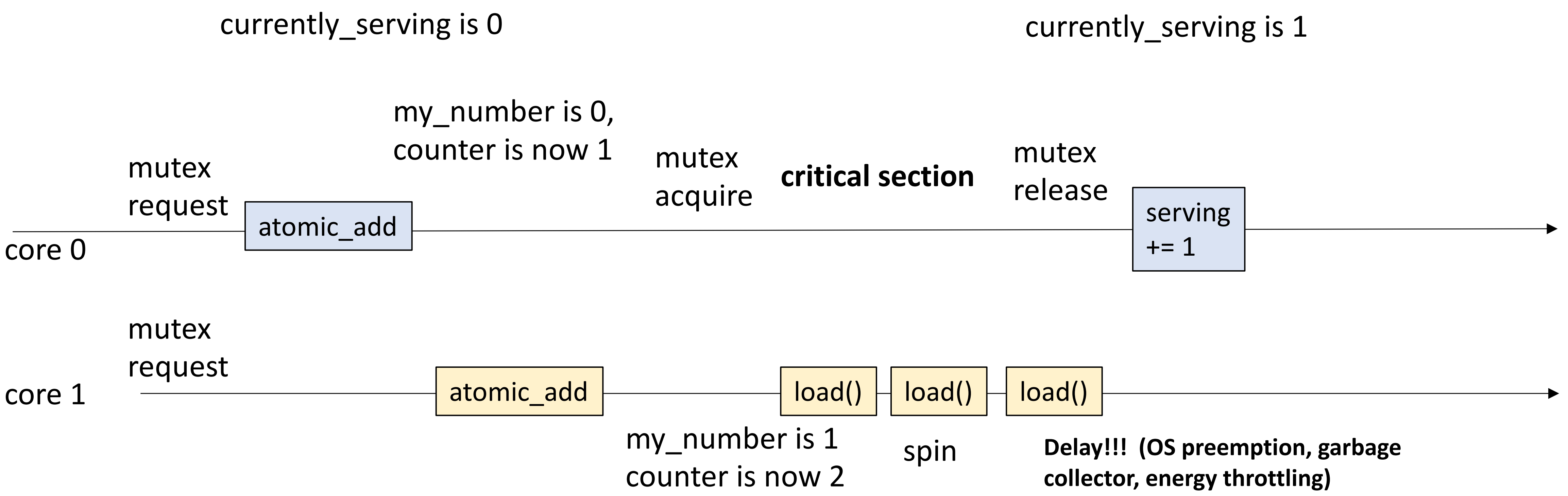


Analysis

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);
}
```

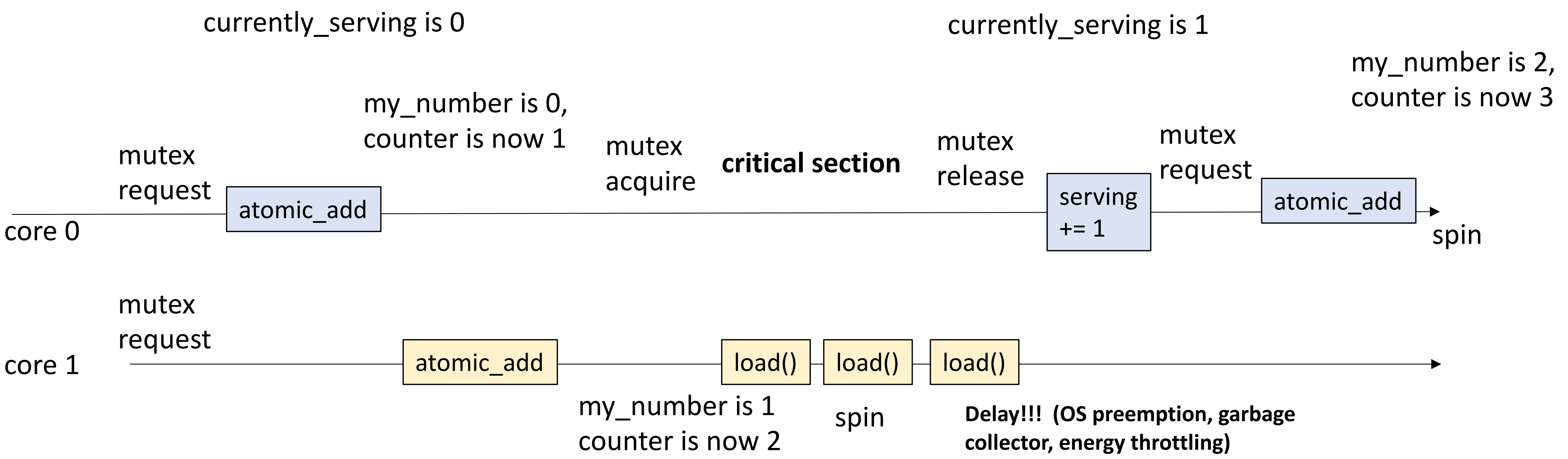


Analysis

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);
}
```

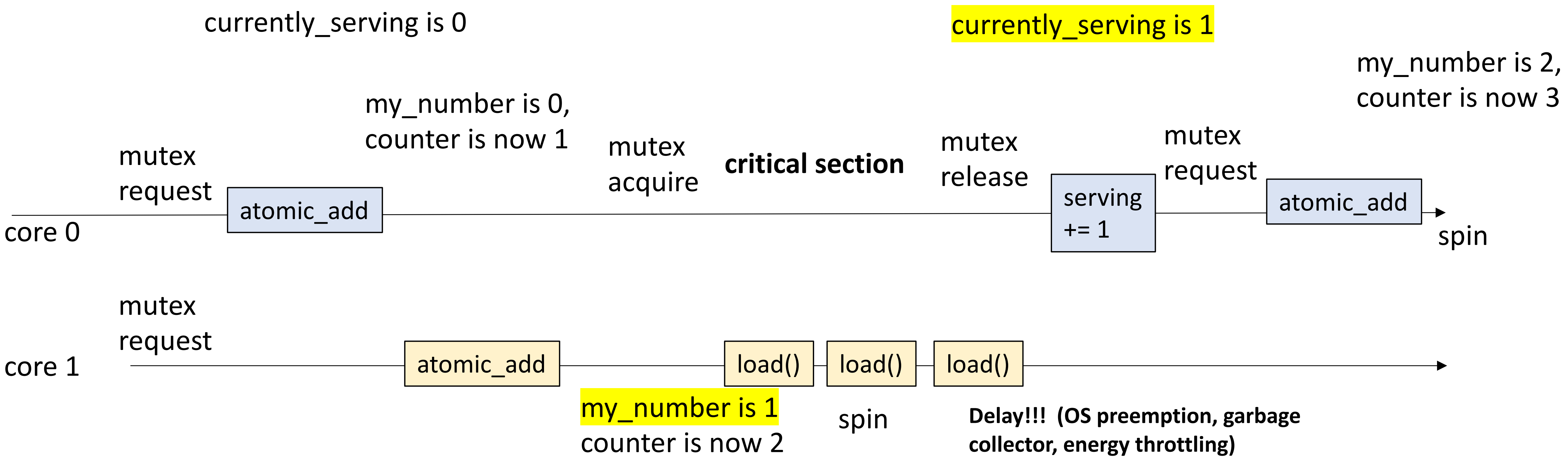


Analysis

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);
}
```

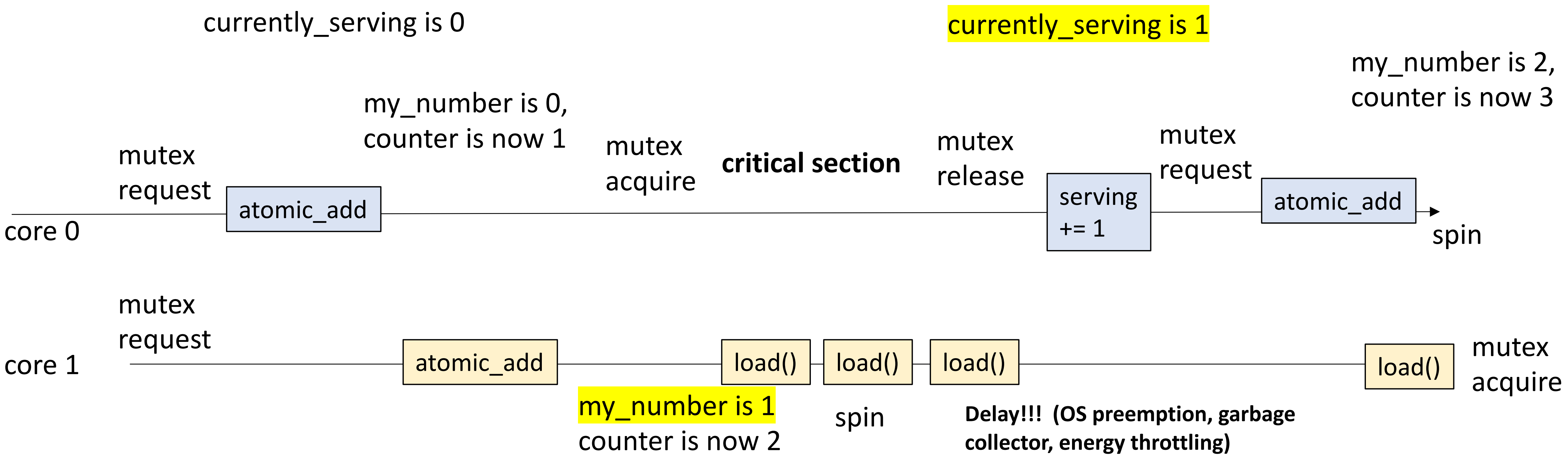


Analysis

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?

- Code Example

Optimizations

How is CAS (and others) implemented?

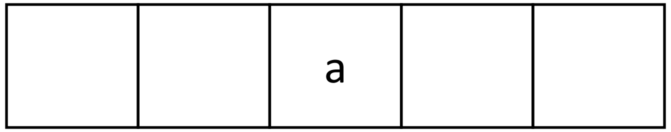
- X86 has an actual instruction
- ARM and POWER are load linked and store conditional

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

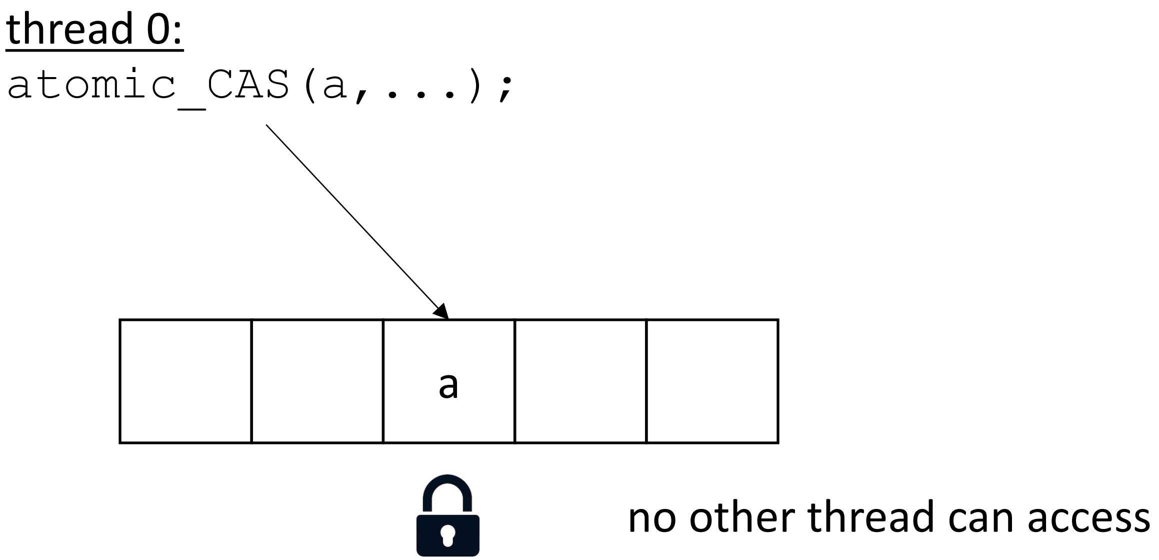
<u>thread 0:</u>

```
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

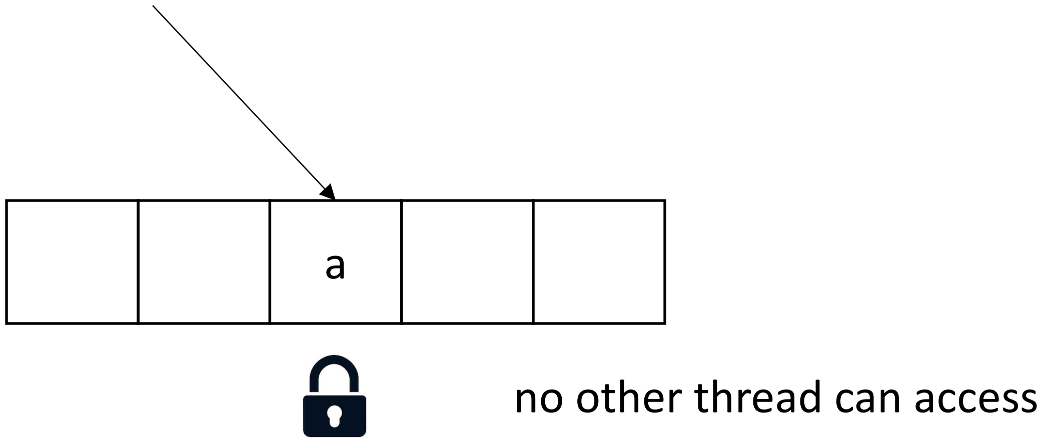
- X86 has an actual instruction: lock the memory location
- Known as **Pessimistic Concurrency**
- Assume conflicts will happen and defend against them from the start

thread 0:

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

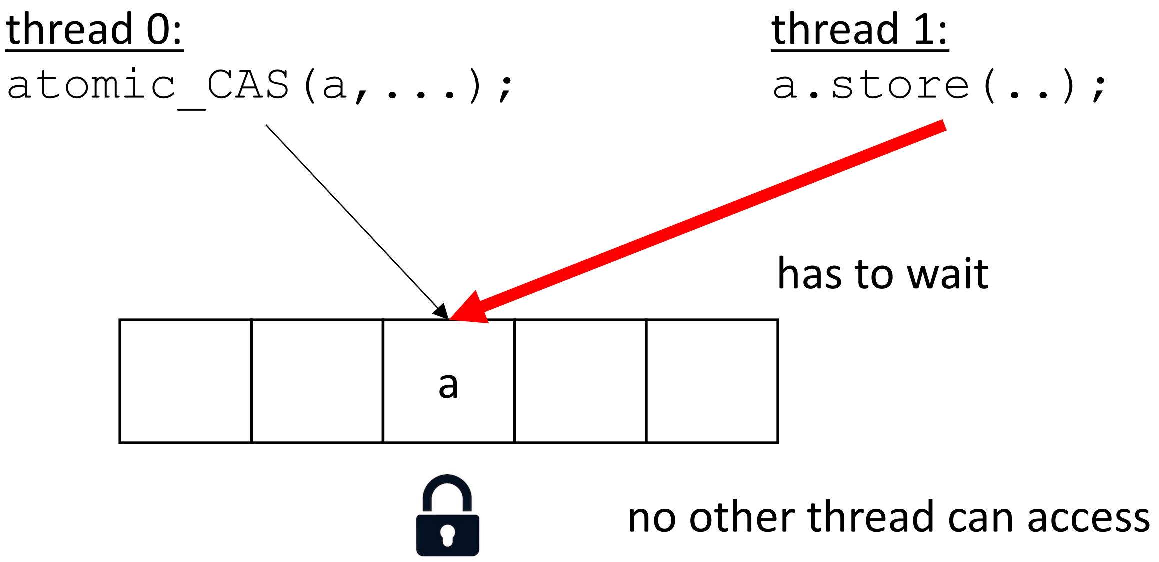
thread 1:

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



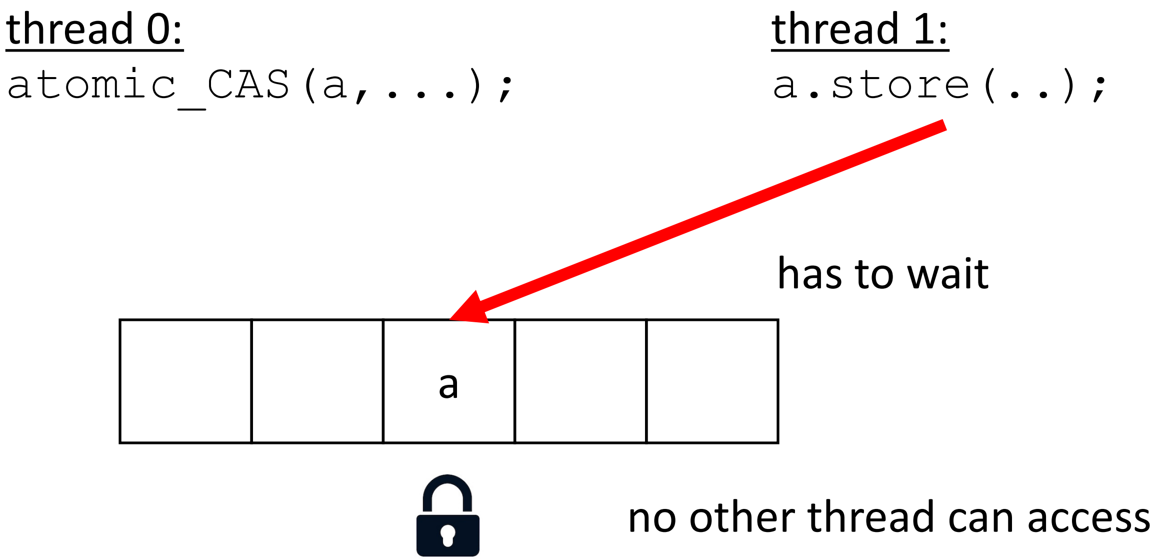
Pessimistic Concurrency

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



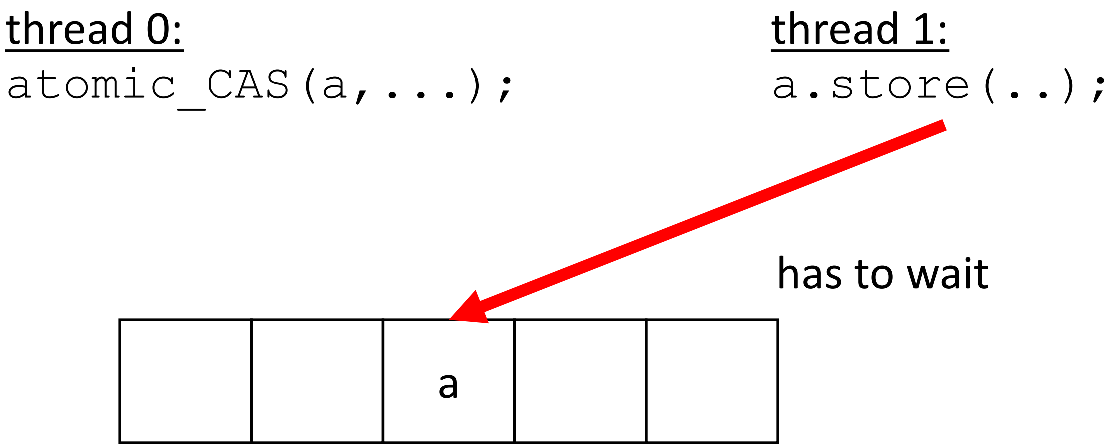
Pessimistic Concurrency

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



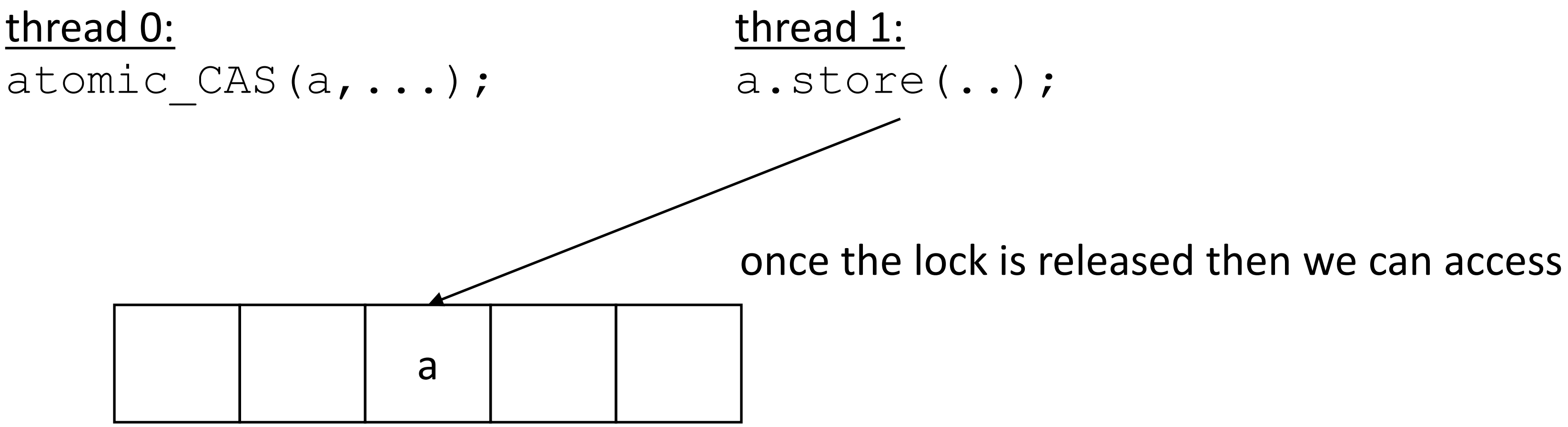
Pessimistic Concurrency

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



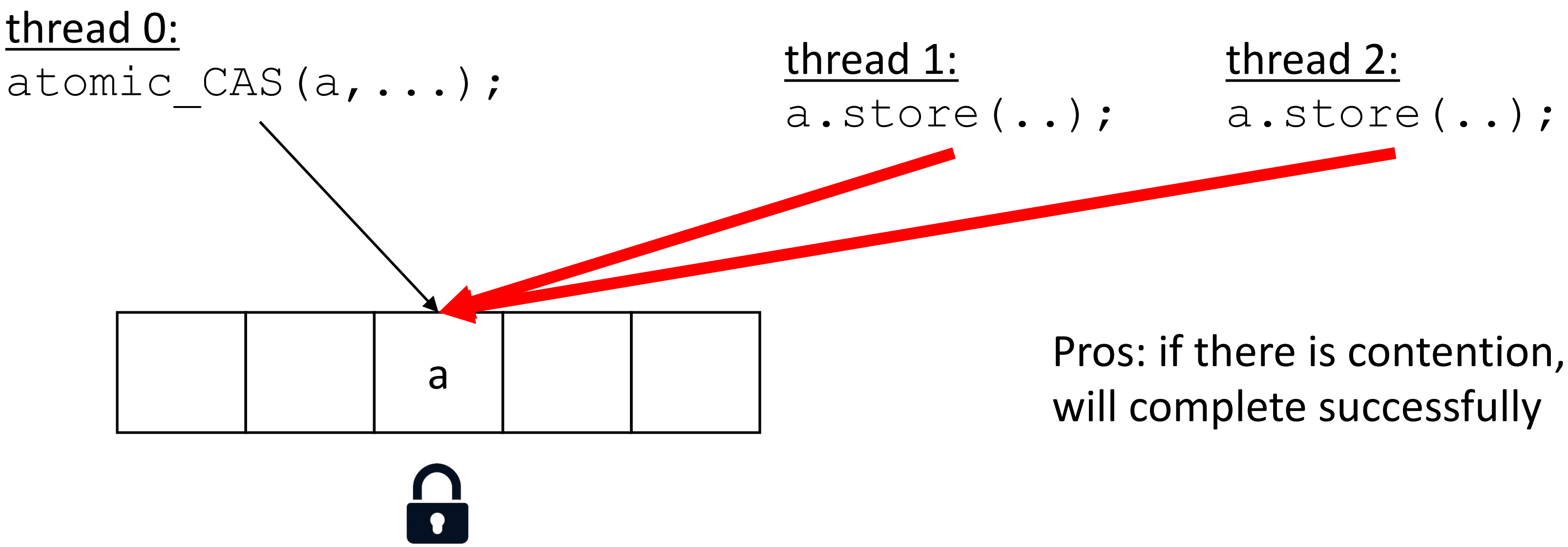
Pessimistic Concurrency

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



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



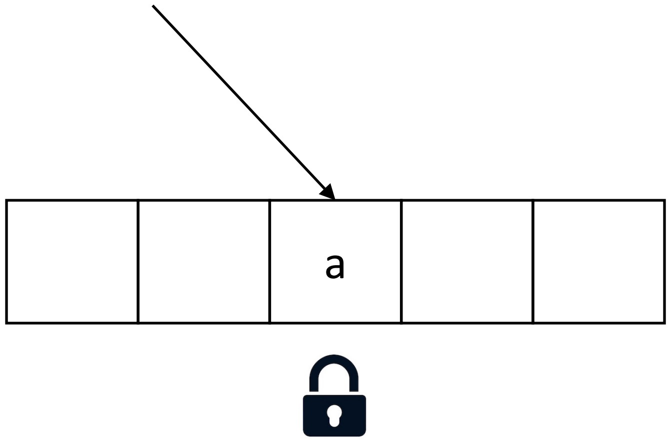
Pros: if there is contention, the CAS will complete successfully

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

<u>thread 0:</u>

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



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

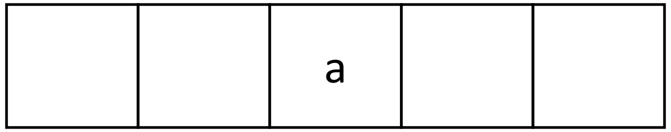
Optimistic Concurrency

- 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

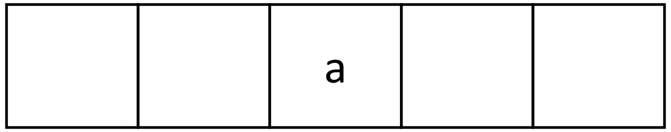


Optimistic Concurrency

- 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);
```



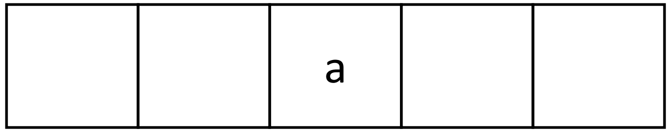
T0_exclusive = 1

Optimistic Concurrency

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

<u>thread 0:</u>

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



T0_exclusive = 1

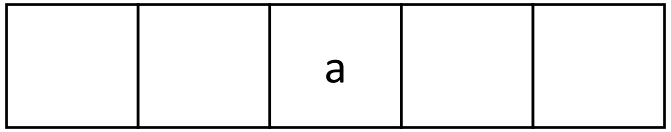
Optimistic Concurrency

- 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

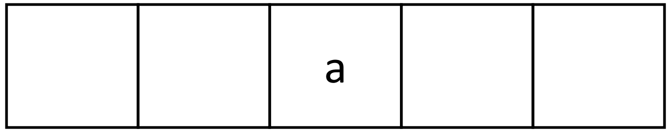
- ARM has load/store exclusive
- Known as **Optimistic Concurrency**
- Assume ***no*** conflicts will happen. Detects and reacts to them.

thread 0:

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

thread 1:

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



Optimistic Concurrency

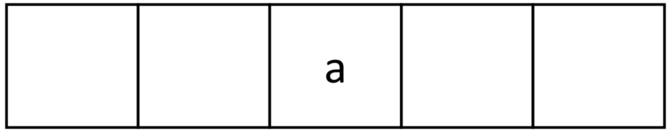
- ARM has load/store exclusive
- Known as **Optimistic Concurrency**
- Assume ***no*** conflicts will happen. Detects and reacts to them.

thread 0:

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

thread 1:

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



T0_exclusive = 1

Optimistic Concurrency

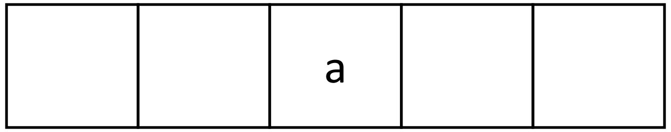
- ARM has load/store exclusive
- Known as **Optimistic Concurrency**
- Assume ***no*** conflicts will happen. Detects and reacts to them.

thread 0:

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

thread 1:

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



T0_exclusive = 1

Optimistic Concurrency

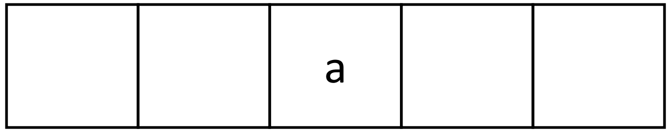
- ARM has load/store exclusive
- Known as **Optimistic Concurrency**
- Assume ***no*** conflicts will happen. Detects and reacts to them.

<u>thread 0:</u>

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

<u>thread 1:</u>

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



T0_exclusive = 0

Optimistic Concurrency

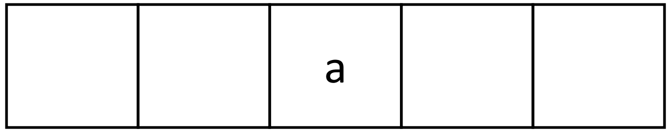
- ARM has load/store exclusive
- Known as **Optimistic Concurrency**
- Assume ***no*** conflicts will happen. Detects and reacts to them.

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.

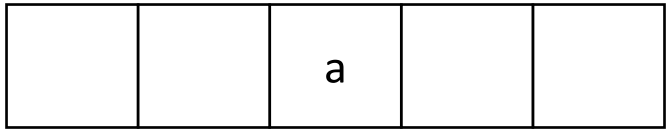
thread 0:

```
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

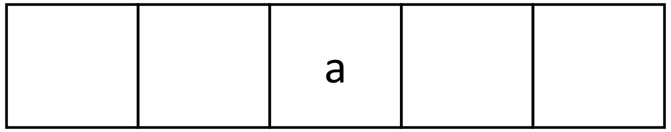
- ARM has load/store exclusive
- Known as **Optimistic Concurrency**
- Assume ***no*** conflicts will happen. Detects and reacts to them.

thread 0:

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

thread 1:

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



T0_exclusive = 0

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

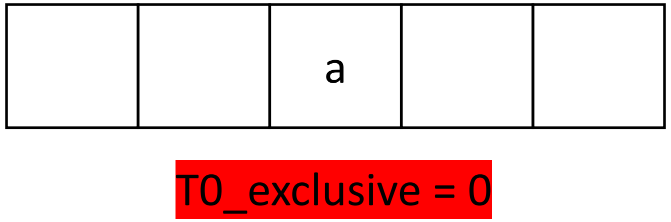
solution: loop until success:

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));
```

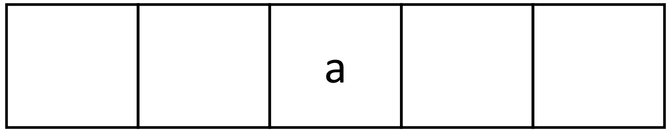


Optimistic Concurrency

- 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));
```



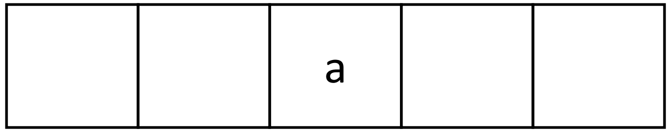
T0_exclusive = 1

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 = 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.

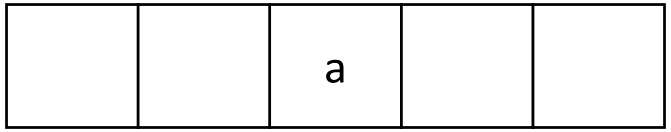
Optimistic Concurrency

- 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));
```

ARM implements all atomics this way!



T0_exclusive = 1