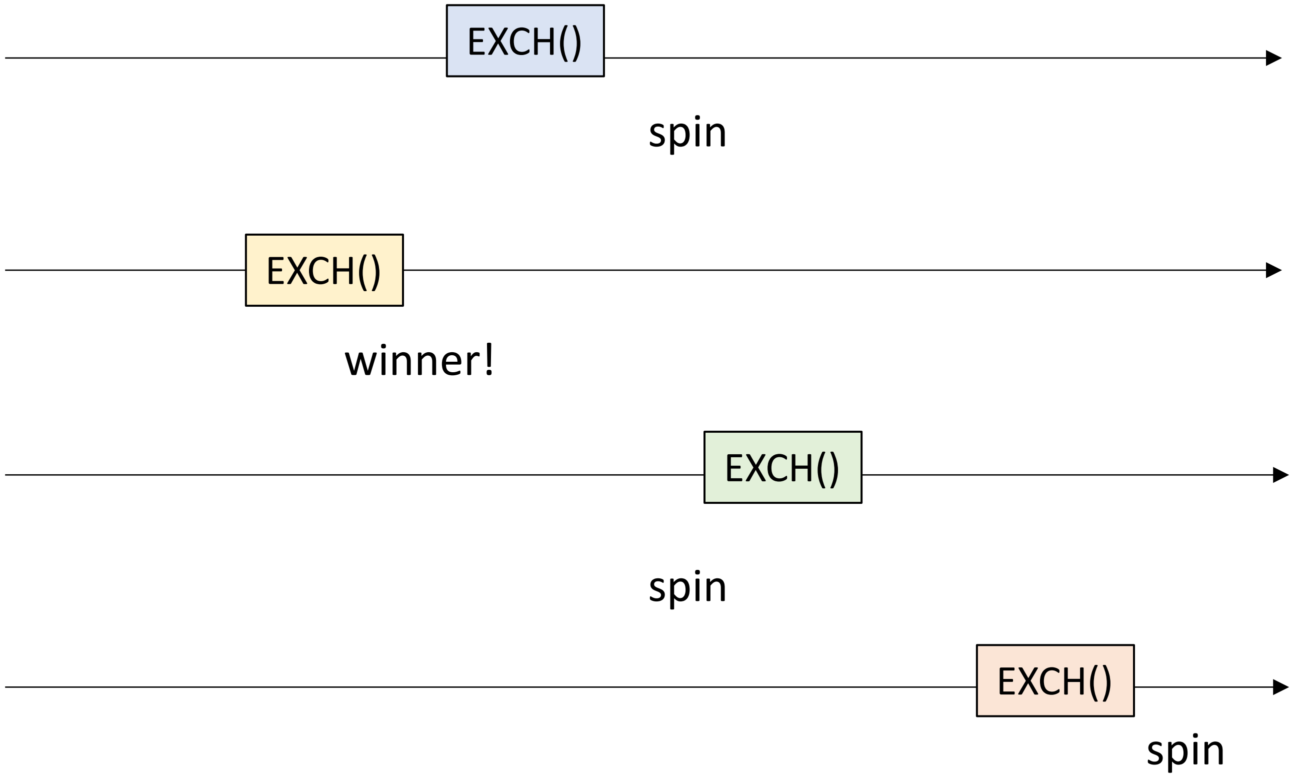
CSE113: Parallel Programming

Jan. 26, 2022

- **Topics**:
  - RMW mutex implementations
    - Fairness
    - Relaxed Peeking
    - Backoff



Announcements

- We are starting to grade HW 1, expect grades by the time HW 2 is due (potentially sooner)
    - Ask about issues early
    - In some cases you might be asked about performance issues

- Homework 2 was released on Friday
    - Hoping to get through all material to get through all of it by today!

Returning to in-person

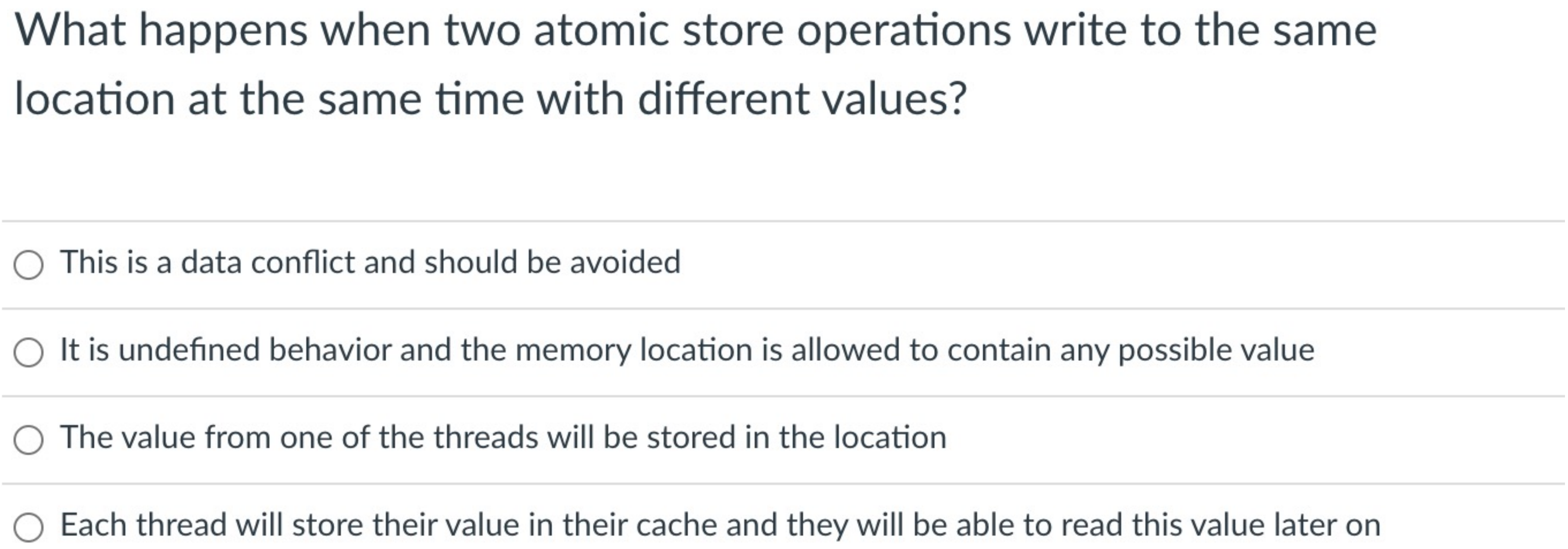
- We will discuss at the end of the class
  - Gives us time to go over questions/comments

Today’s Quiz

- Please do it!

- Due by midnight tomorrow

Previous quiz



What happens when two atomic store operations write to the same location at the same time with different values?

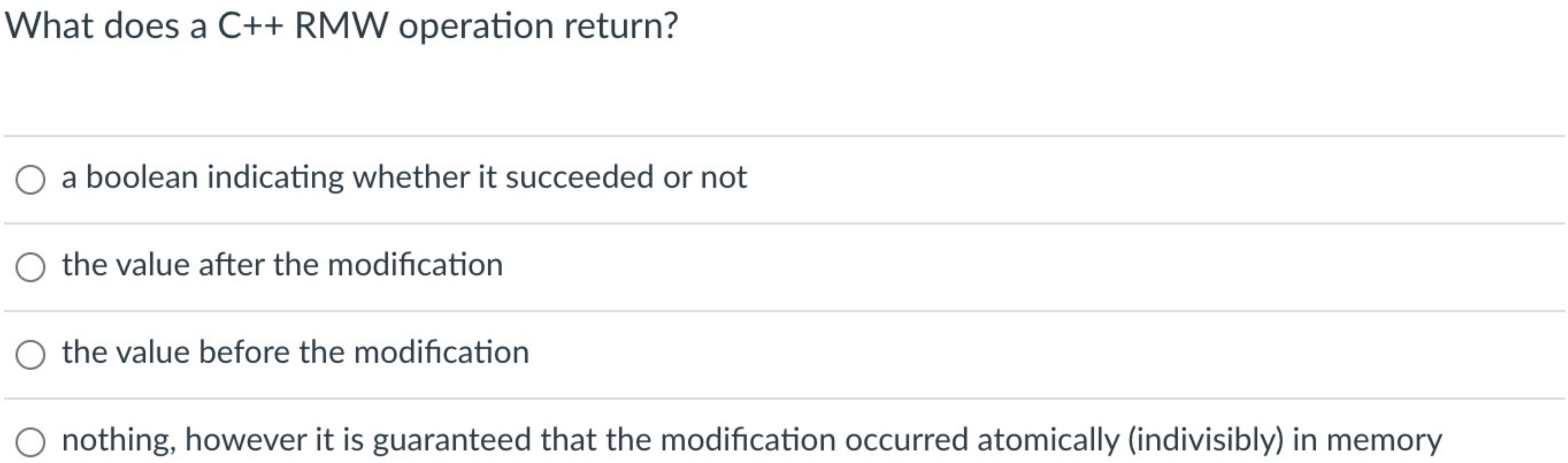
○ This is a data conflict and should be avoided

○ It is undefined behavior and the memory location is allowed to contain any possible value

○ The value from one of the threads will be stored in the location

○ Each thread will store their value in their cache and they will be able to read this value later on

Previous quiz



What does a C++ RMW operation return?

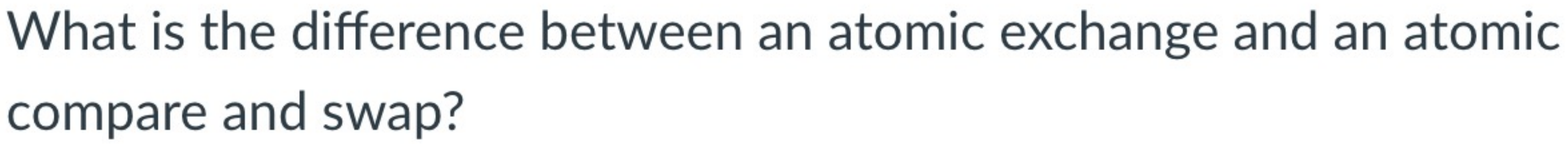
○ a boolean indicating whether it succeeded or not

○ the value after the modification

○ the value before the modification

○ nothing, however it is guaranteed that the modification occurred atomically (indivisibly) in memory

Previous quiz



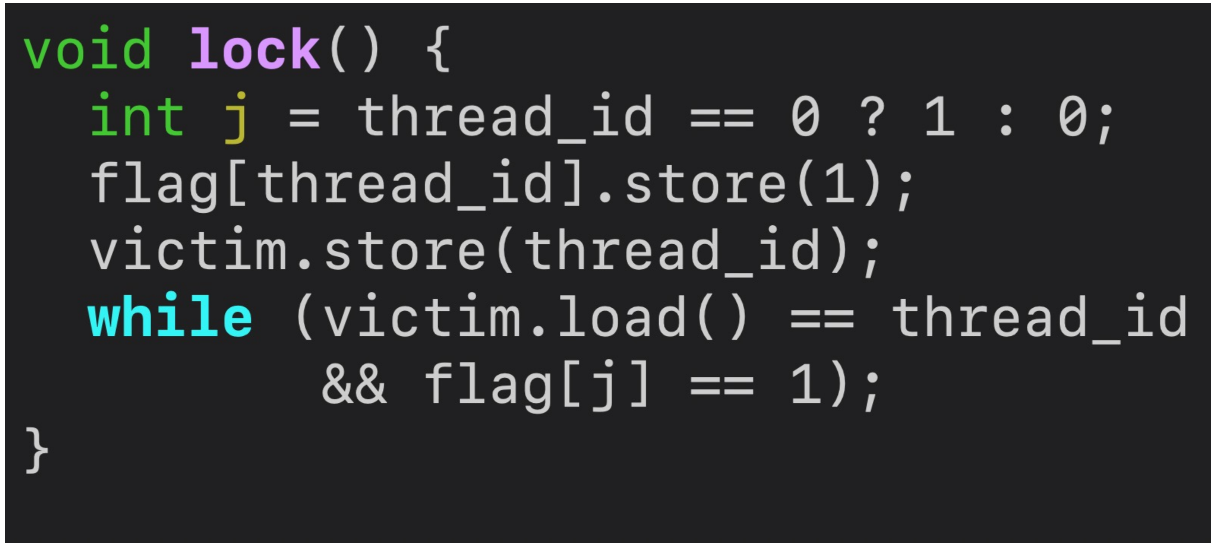
What is the difference between an atomic exchange and an atomic compare and swap?

Previous quiz

Discuss a few trade-offs between RMW mutexes and the simpler load/store mutexes (e.g. peterson's lock).

Review

Peterson’s 2 threaded mutex



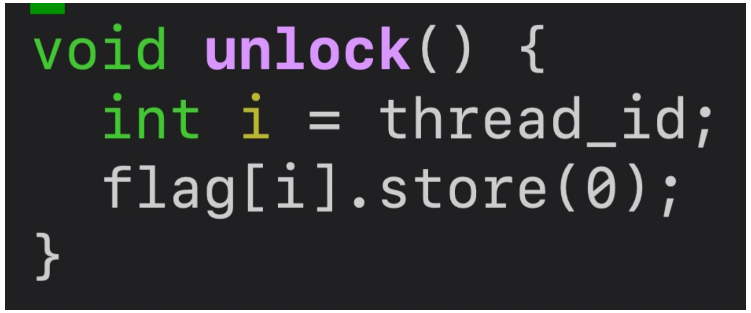
```
void lock() {
  int j = thread_id == 0 ? 1 : 0;
  flag[thread_id].store(1);
  victim.store(thread_id);
  while (victim.load() == thread_id
         && flag[j] == 1);
}
```

j is the other thread

Mark ourself as interested

volunteer to be the victim in case of a tie

Spin only if:
there was a tie in wanting the lock,
and I won the volunteer raffle to spin



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

mark ourselves as uninterested

RMWs for mutexes

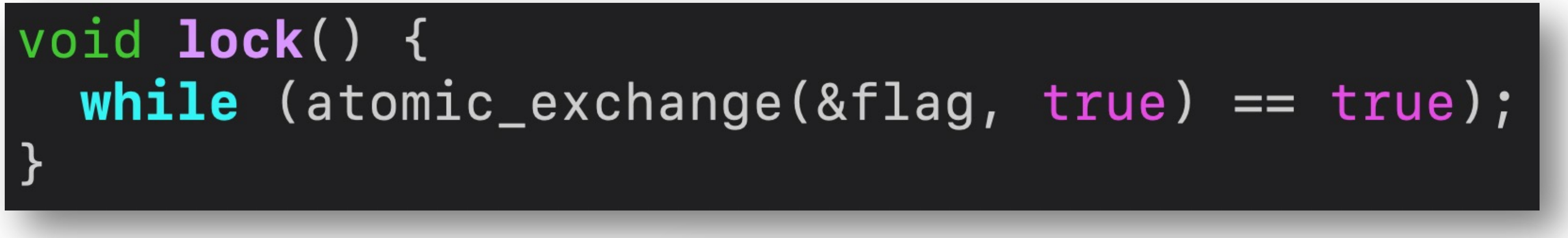
First example: Exchange Lock

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

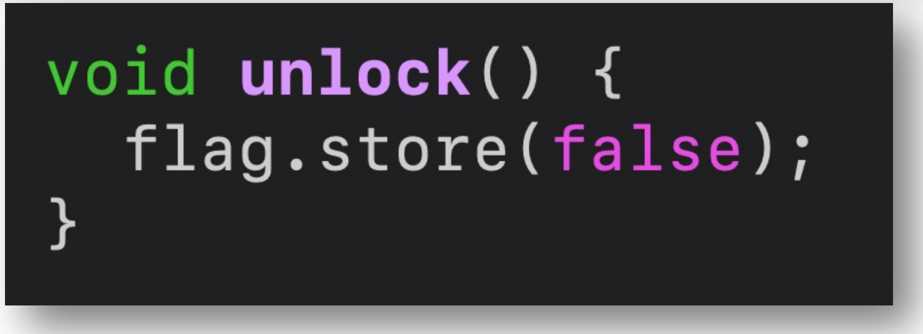
Loads the value at a and stores the value in v at a. Returns the value that was loaded.

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

Analysis



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



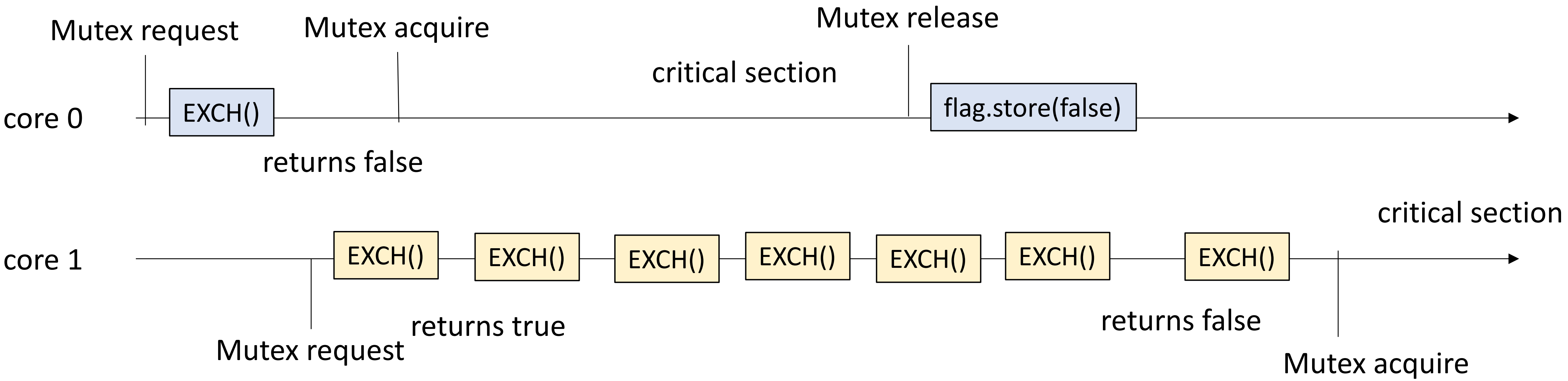
```
void unlock() {
  flag.store(false);
}
```

Thread 0:

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

Thread 1:

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



Mutex request

Mutex acquire

critical section

Mutex release

core 0

EXCH()

flag.store(false)

returns false

critical section

core 1

EXCH()

EXCH()

EXCH()

EXCH()

EXCH()

EXCH()

EXCH()

returns true

returns false

Mutex request

Mutex acquire

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);
```

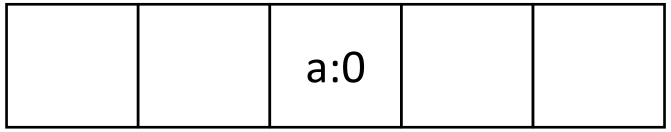
Checks if value at a is equal to the value at `expected`. If it is equal, swap with `replace`.
returns `True` if the values were equal. `False` otherwise.
`expected` is passed by reference: the previous value at a is returned

Most versatile RMW: Compare-and-swap

```
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);
```



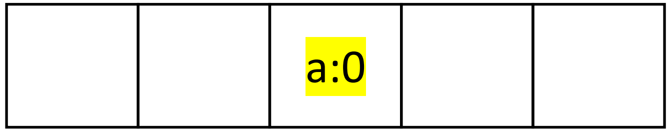
a:0

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);
```



a:0

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);
```

		a:6		

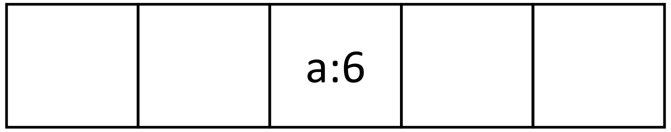
Most versatile RMW: Compare-and-swap

```
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



a:6

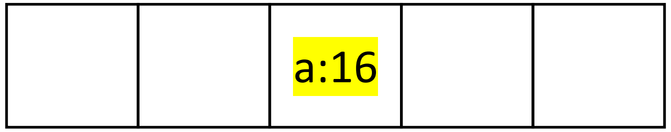
Most versatile RMW: Compare-and-swap

```
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;
}
```

next example

thread 0:

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

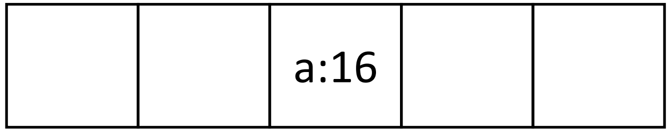


Most versatile RMW: Compare-and-swap

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

thread 0:

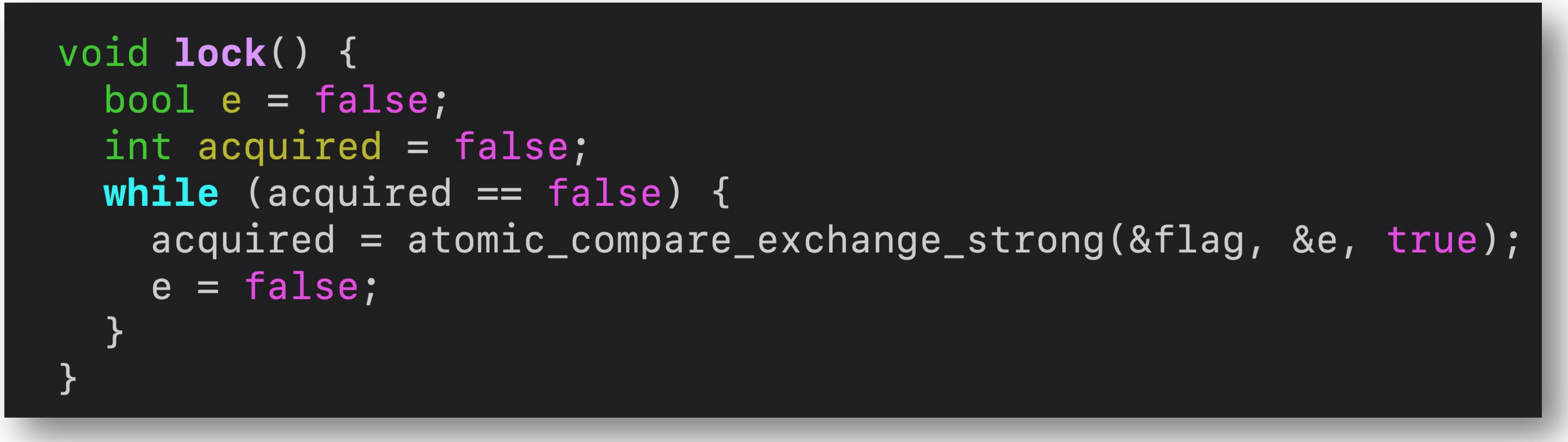
```
// some atomic int address a
int e = 0;
bool s = atomic_CAS(a,&e,6);
```



16

false

CAS lock

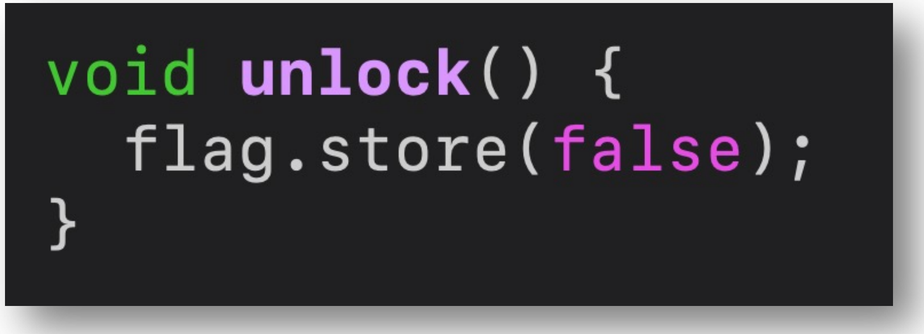


```
void lock() {
  bool e = false;
  int acquired = false;
  while (acquired == false) {
    acquired = atomic_compare_exchange_strong(&flag, &e, true);
    e = false;
  }
}
```

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);
}
```

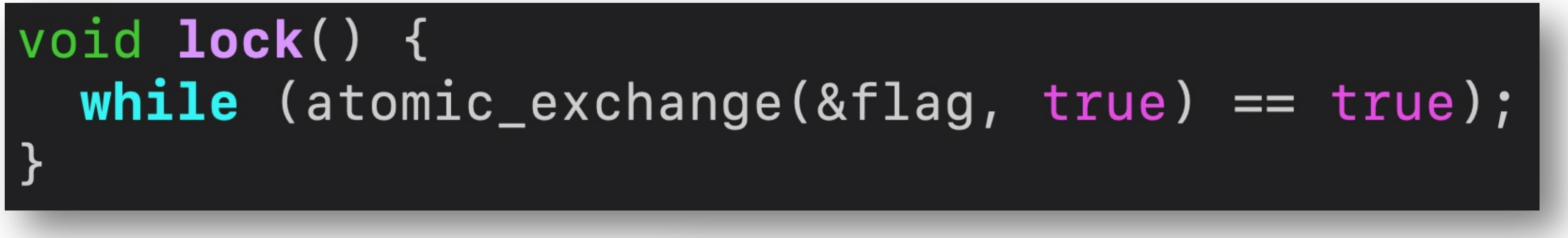
Unlock is simple! Just store false back

Schedule

- **Fairness of RMW locks**
- Optimization of RMW locks
- RW mutexes

Starvation

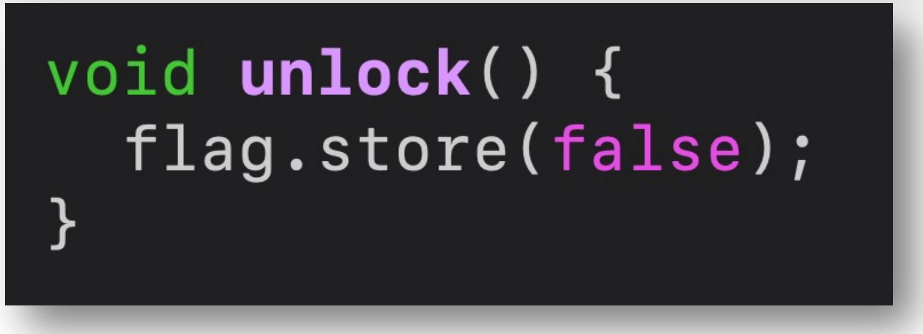
- Are these RMW locks fair?



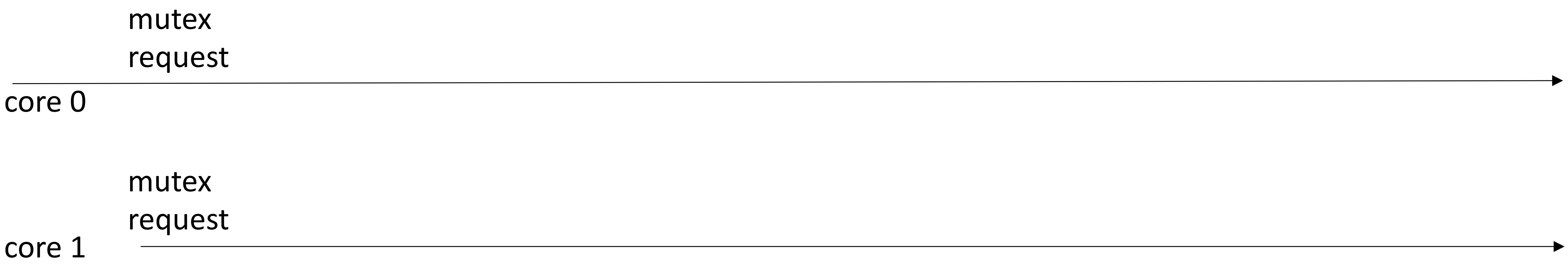
```
void lock() {
  while (atomic_exchange(&flag, true) == true);
}
```

Analysis

Is this mutex starvation Free?



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



mutex request

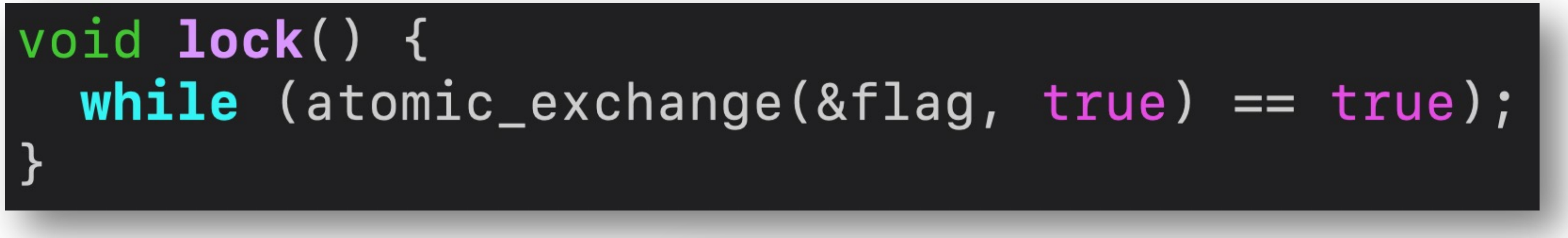
core 0

mutex request

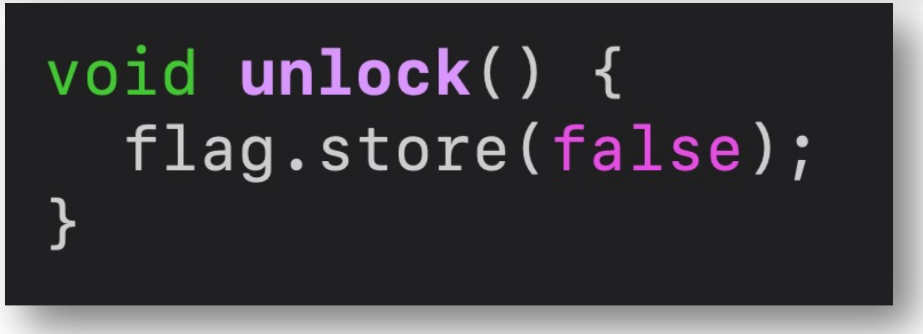
core 1

Analysis

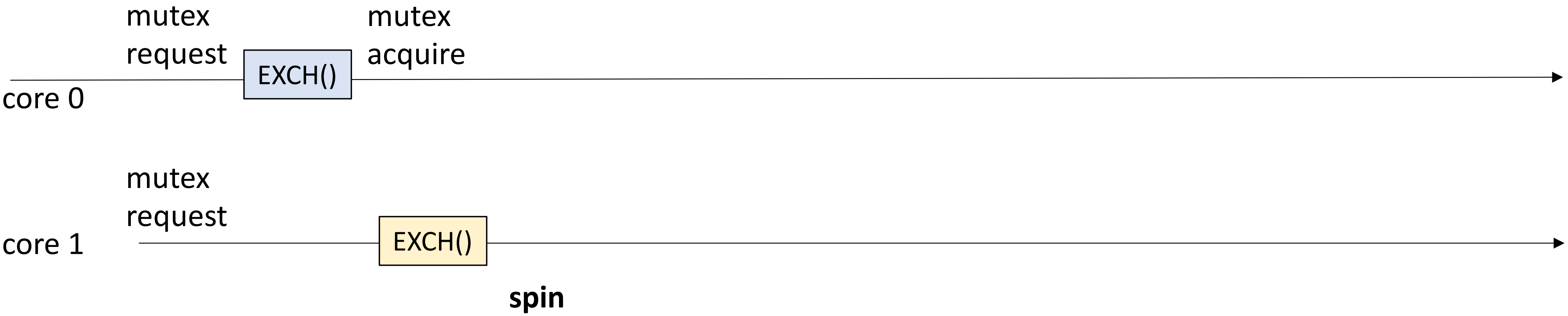
Is this mutex starvation Free?



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



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



mutex request

EXCH()

mutex acquire

core 0

mutex request

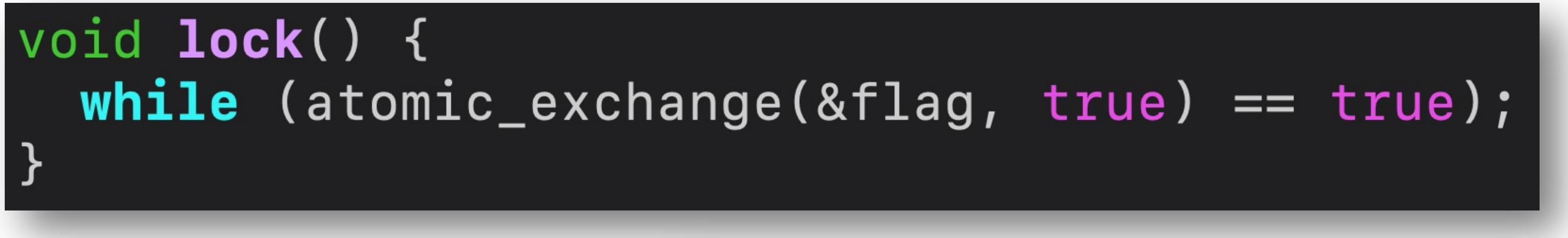
core 1

EXCH()

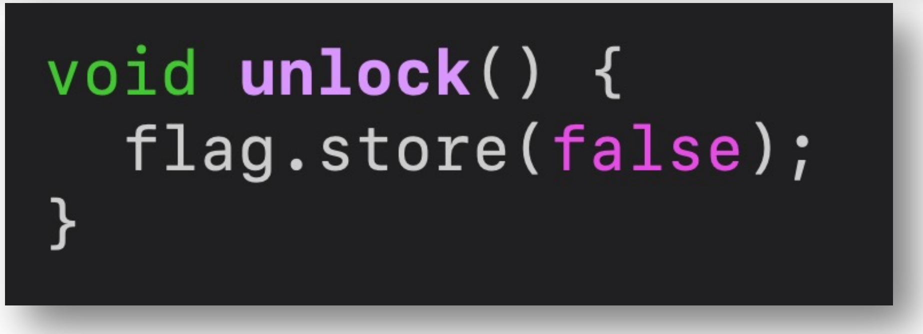
spin

Analysis

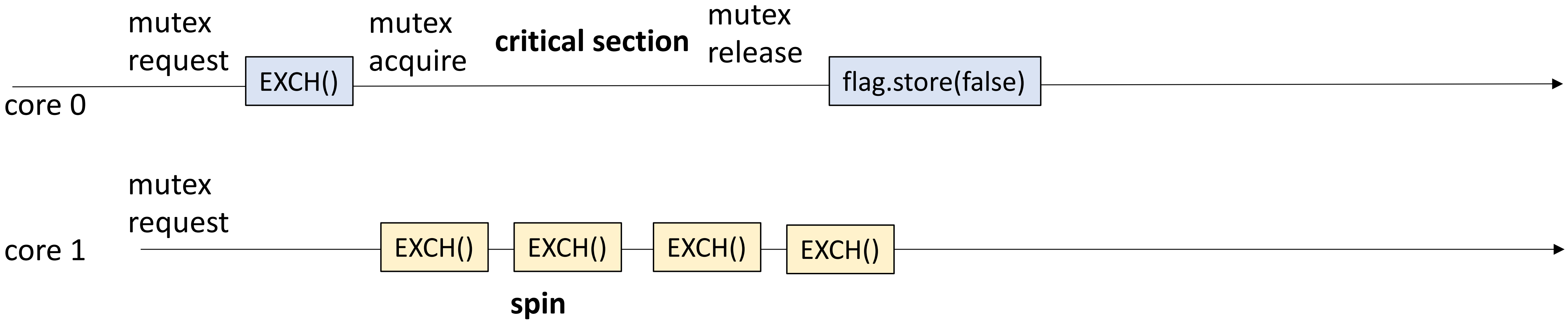
Is this mutex starvation Free?



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

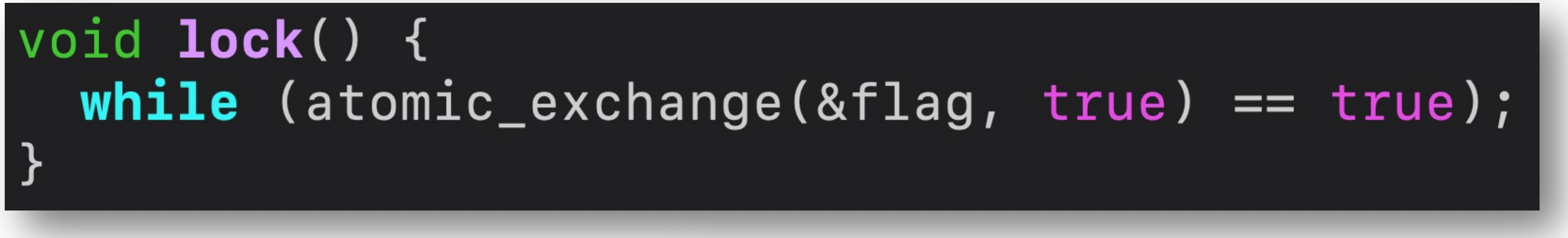


```
void unlock() {
  flag.store(false);
}
```

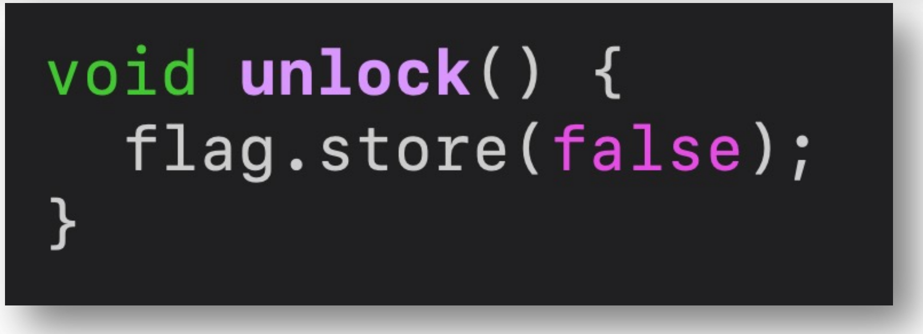


Analysis

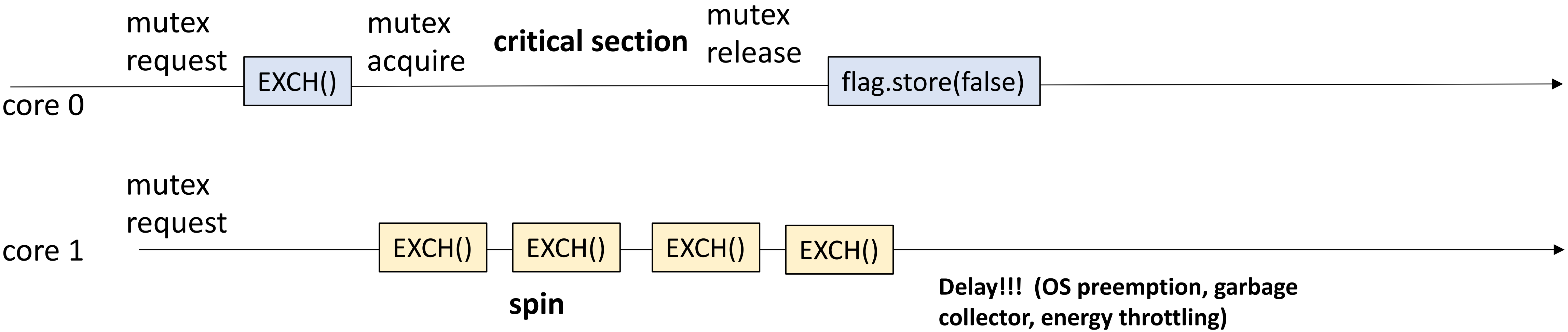
Is this mutex starvation Free?



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

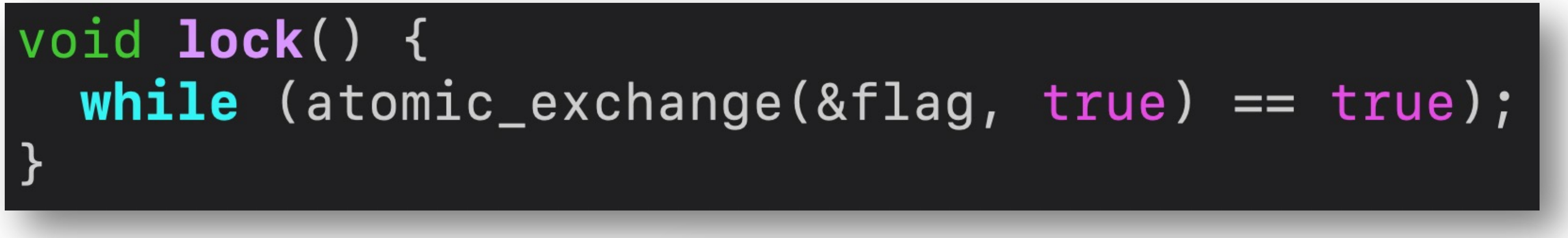


```
void unlock() {
  flag.store(false);
}
```



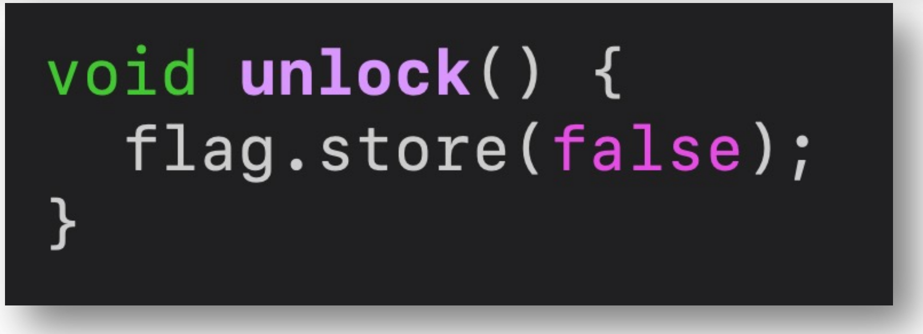
Delay!!! (OS preemption, garbage collector, energy throttling)

Analysis

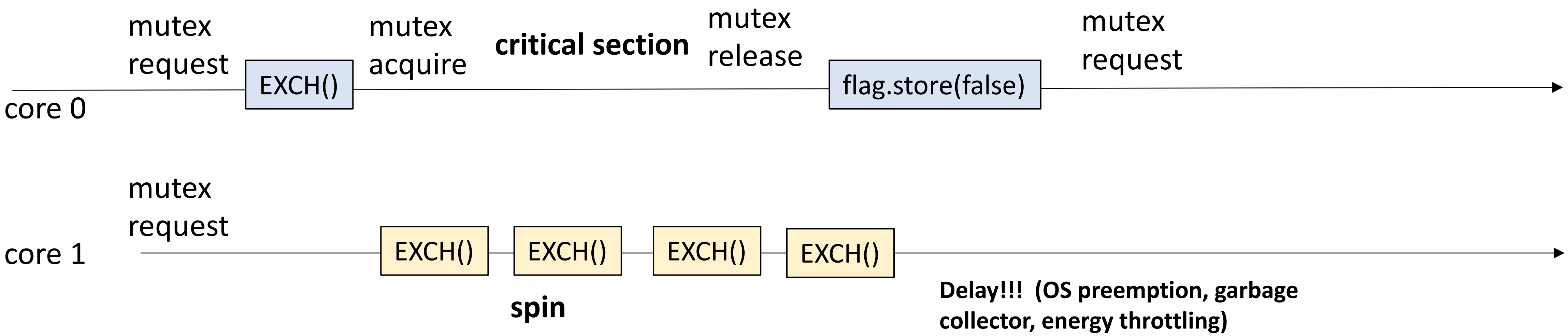


```
void lock() {
  while (atomic_exchange(&flag, true) == true);
}
```

Is this mutex starvation Free?



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



core 0

mutex request

EXCH()

mutex acquire

critical section

mutex release

flag.store(false)

mutex request

core 1

mutex request

EXCH()

EXCH()

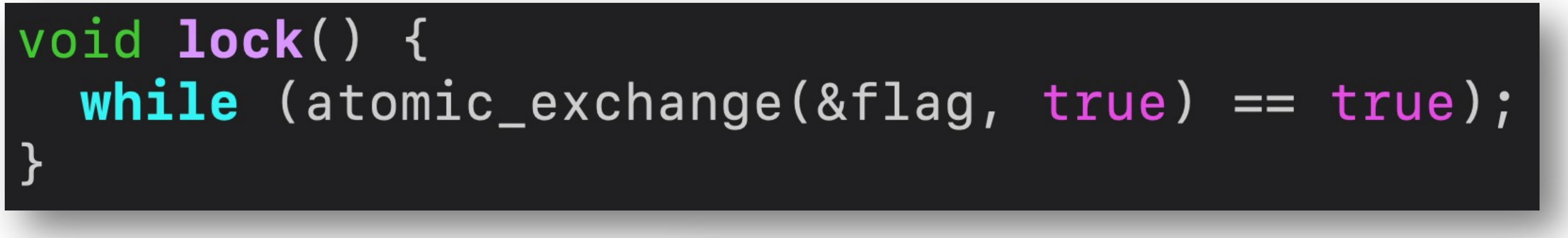
EXCH()

EXCH()

spin

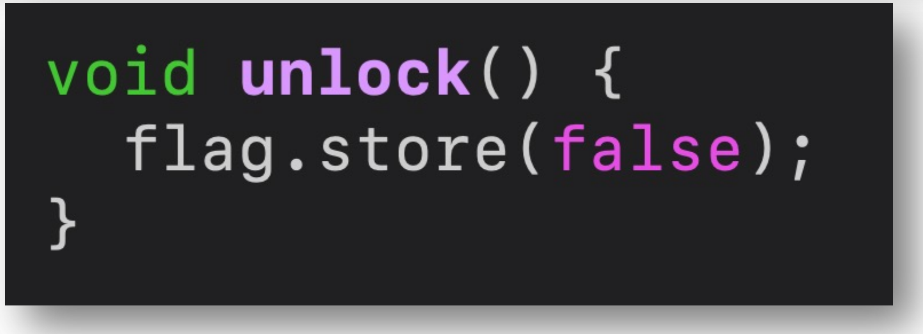
Delay!!! (OS preemption, garbage collector, energy throttling)

Analysis

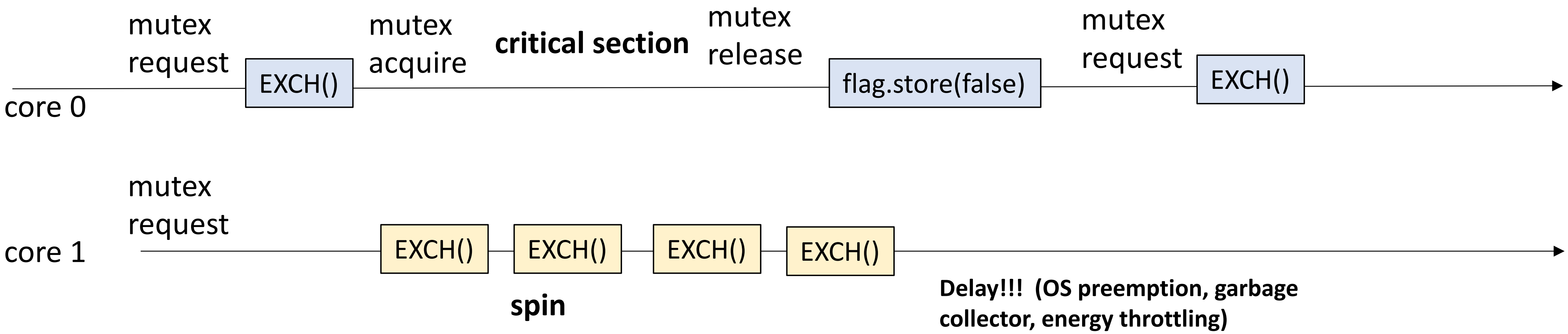


```
void lock() {
  while (atomic_exchange(&flag, true) == true);
}
```

Is this mutex starvation Free?



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



core 0

mutex request

EXCH()

mutex acquire

critical section

mutex release

flag.store(false)

mutex request

EXCH()

core 1

mutex request

EXCH()

EXCH()

EXCH()

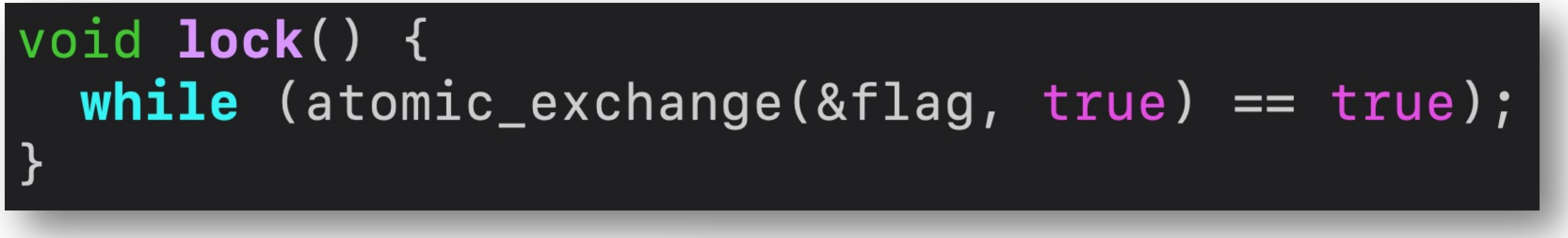
EXCH()

spin

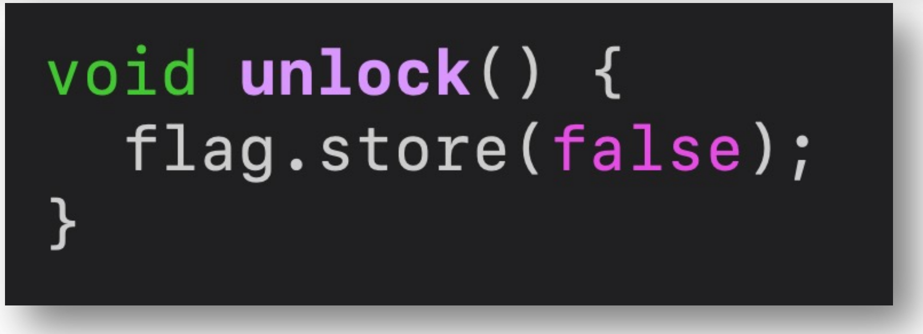
Delay!!! (OS preemption, garbage collector, energy throttling)

Analysis

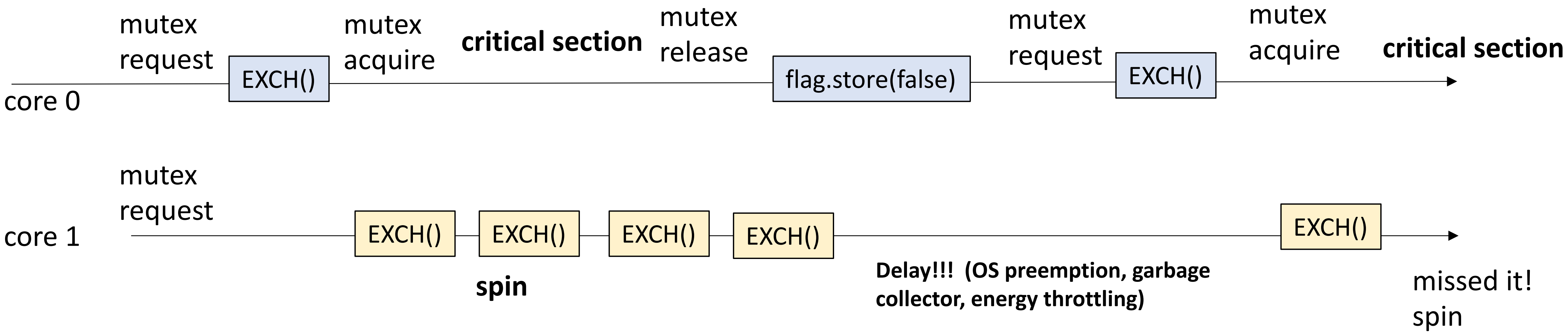
Is this mutex starvation Free?



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



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



core 0

mutex request

EXCH()

mutex acquire

critical section

mutex release

flag.store(false)

mutex request

EXCH()

mutex acquire

critical section

core 1

mutex request

EXCH()

EXCH()

EXCH()

EXCH()

spin

Delay!!! (OS preemption, garbage collector, energy throttling)

EXCH()

missed it! spin

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!

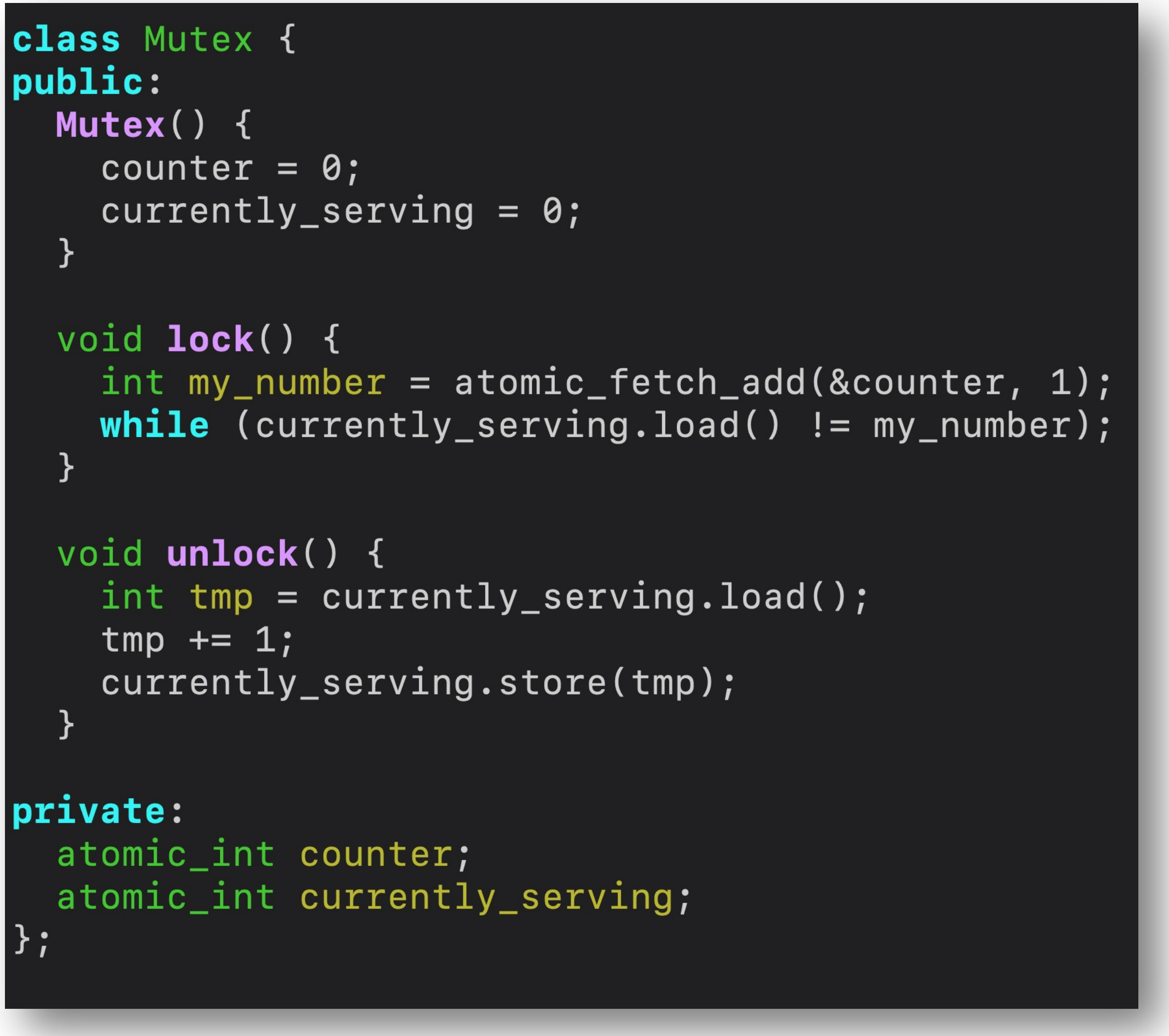
How can we make this more fair?

- 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!

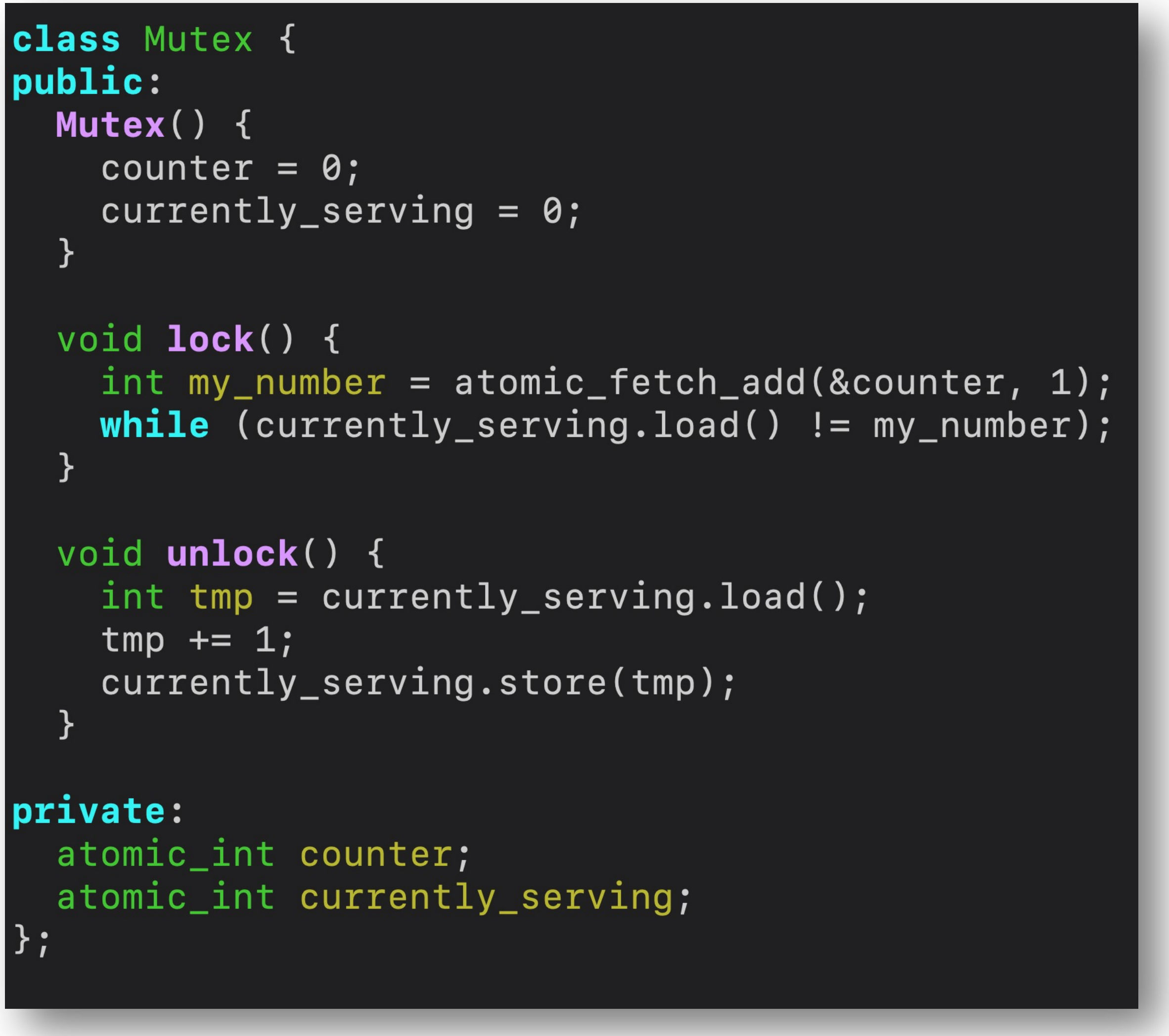
Ticket lock



```
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

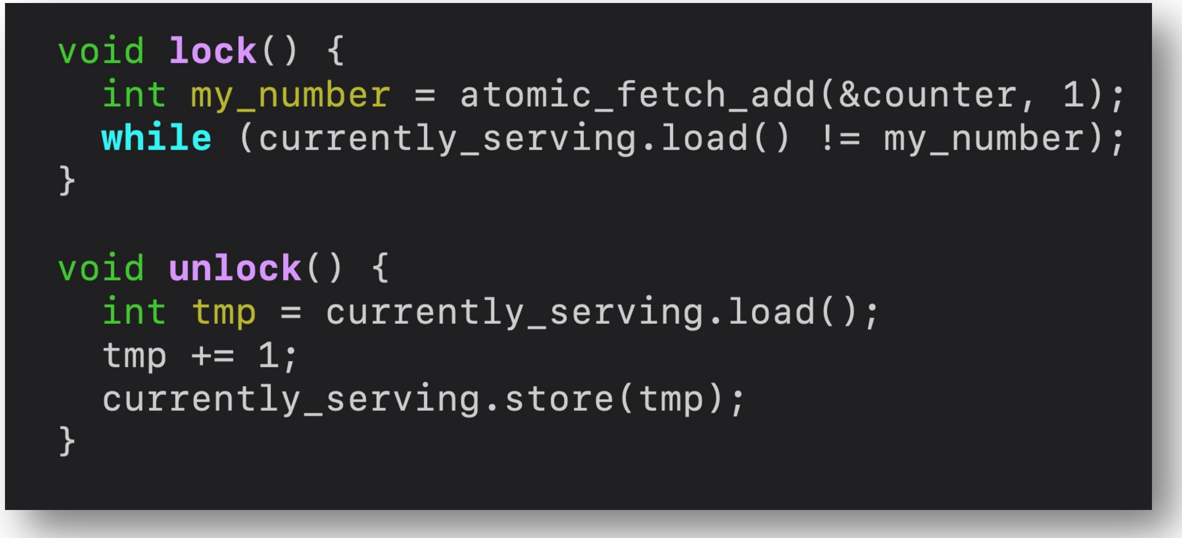
Get a unique number

Spin while your number isn't being served

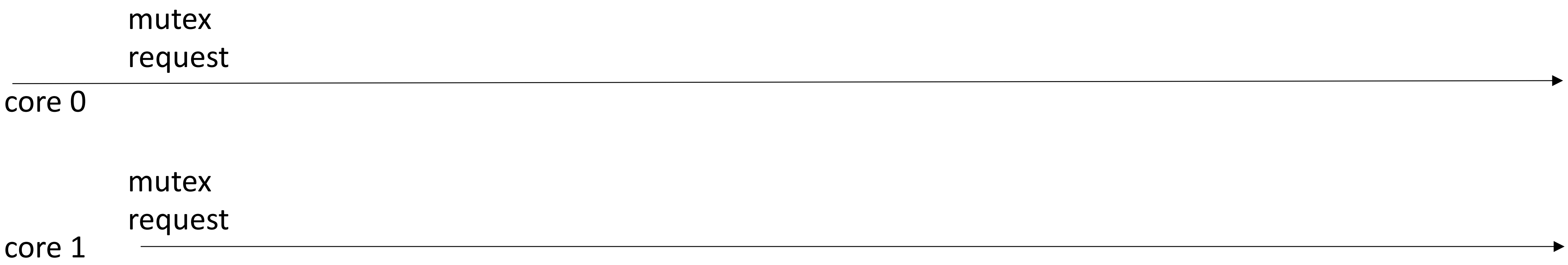
To release, increment the number that's currently being served.

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);
}
```



mutex
request

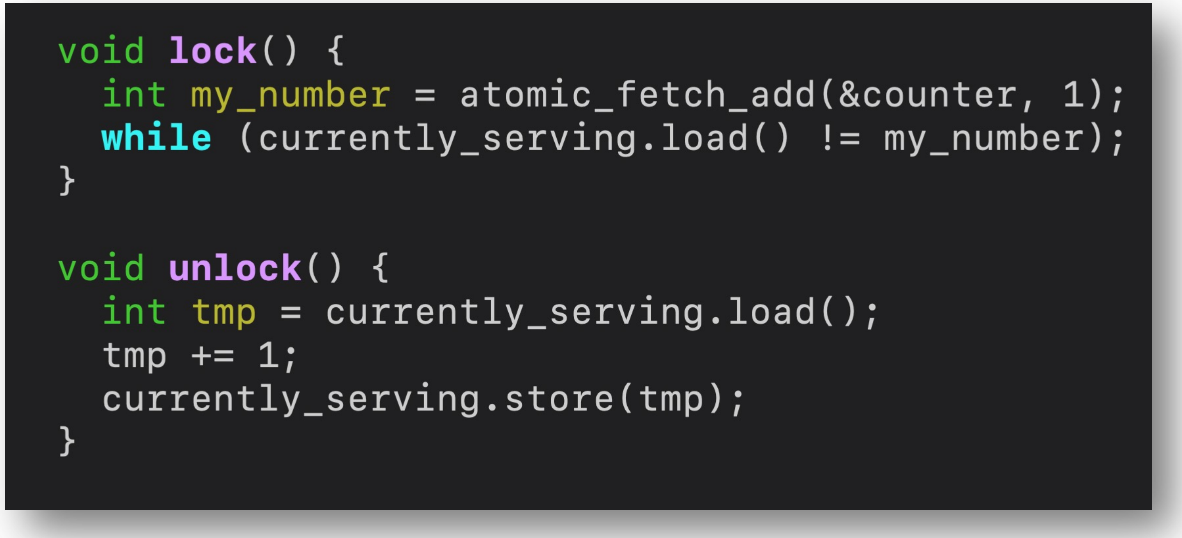
core 0

mutex
request

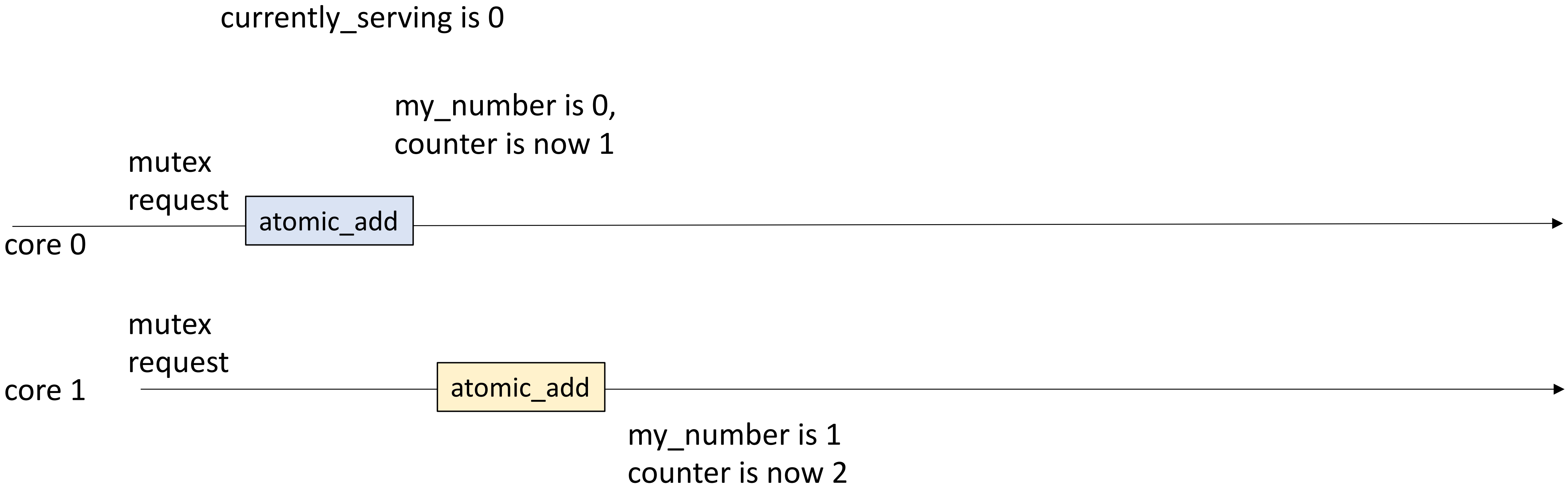
core 1

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);
}
```



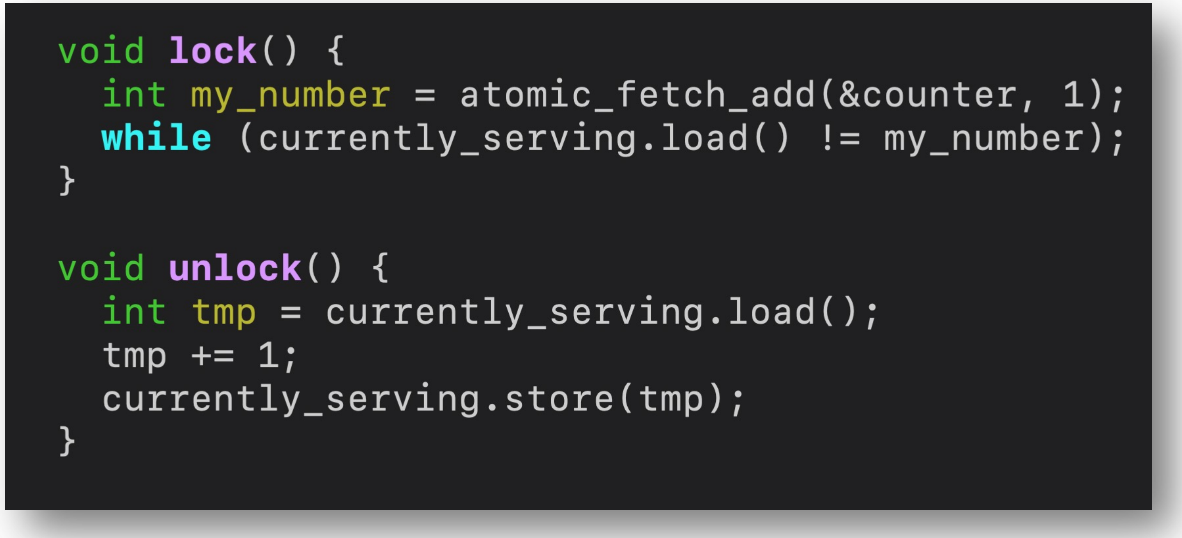
currently_serving is 0

my_number is 0,
counter is now 1

my_number is 1
counter is now 2

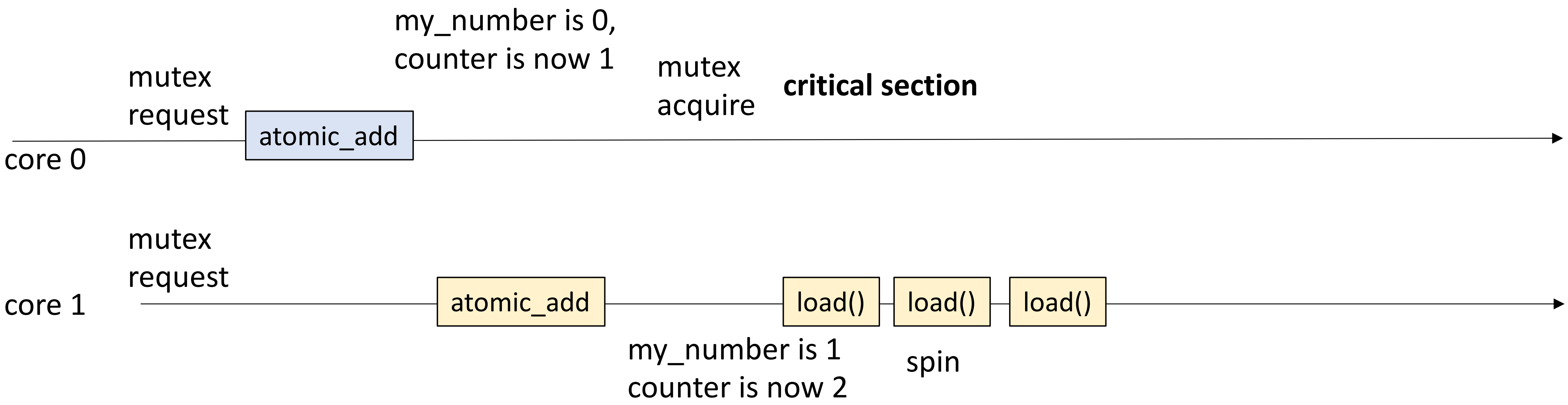
Analysis

Is this mutex starvation Free?



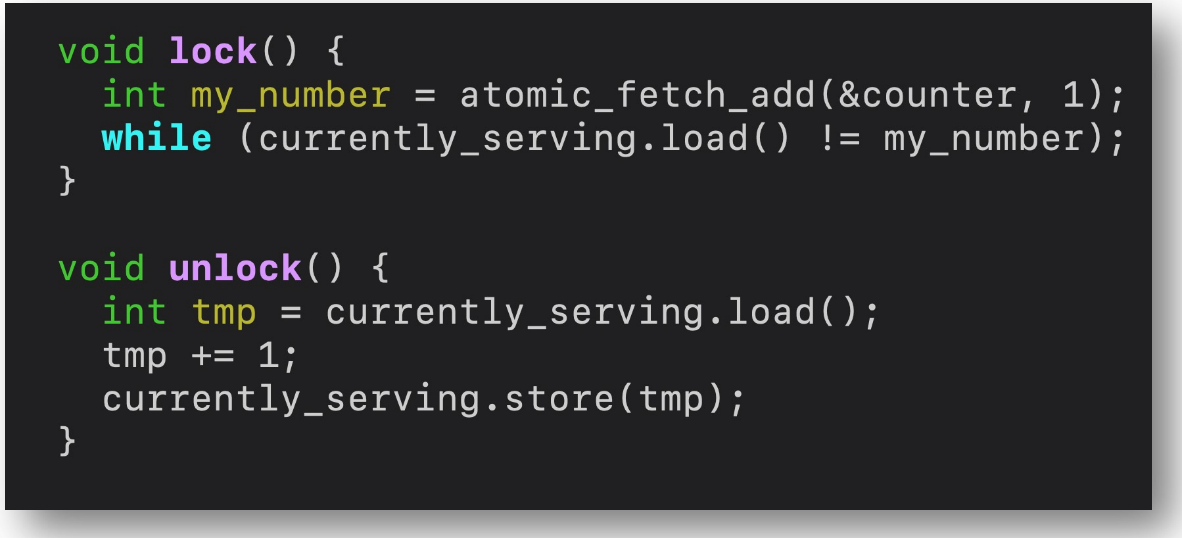
```
void lock() {
  int my_number = atomic_fetch_add(&counter, 1);
  while (currently_serving.load() != my_number);
}

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

currently_serving is 0

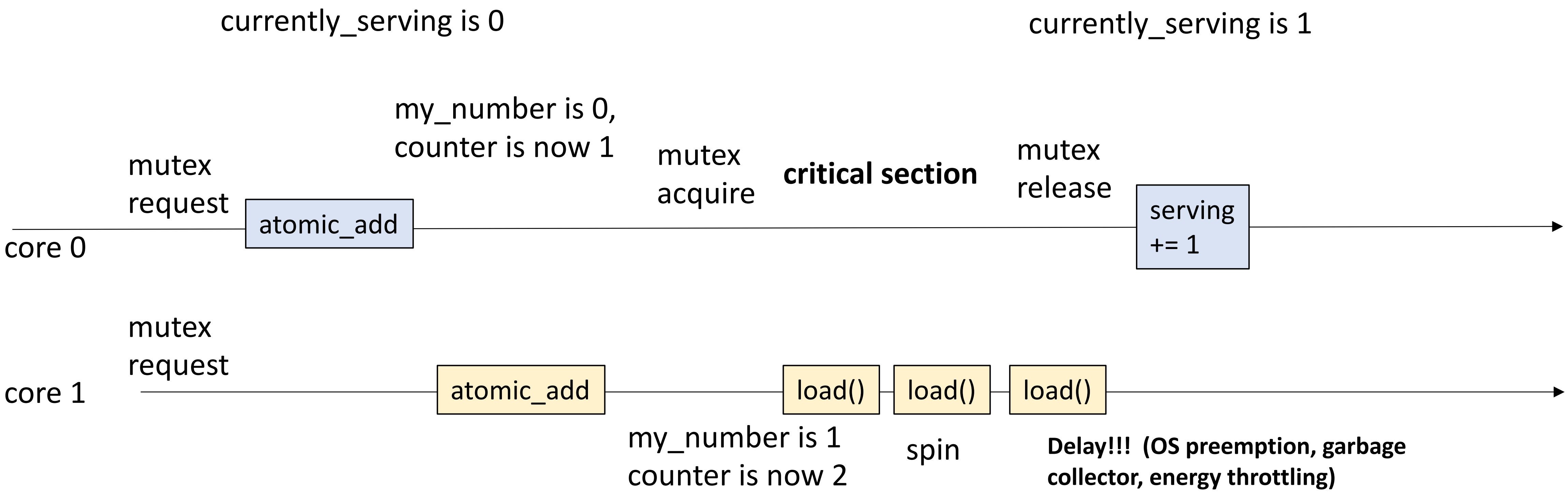


Analysis

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);
}
```



currently_serving is 0

currently_serving is 1

my_number is 0,
counter is now 1

mutex acquire

critical section

mutex release

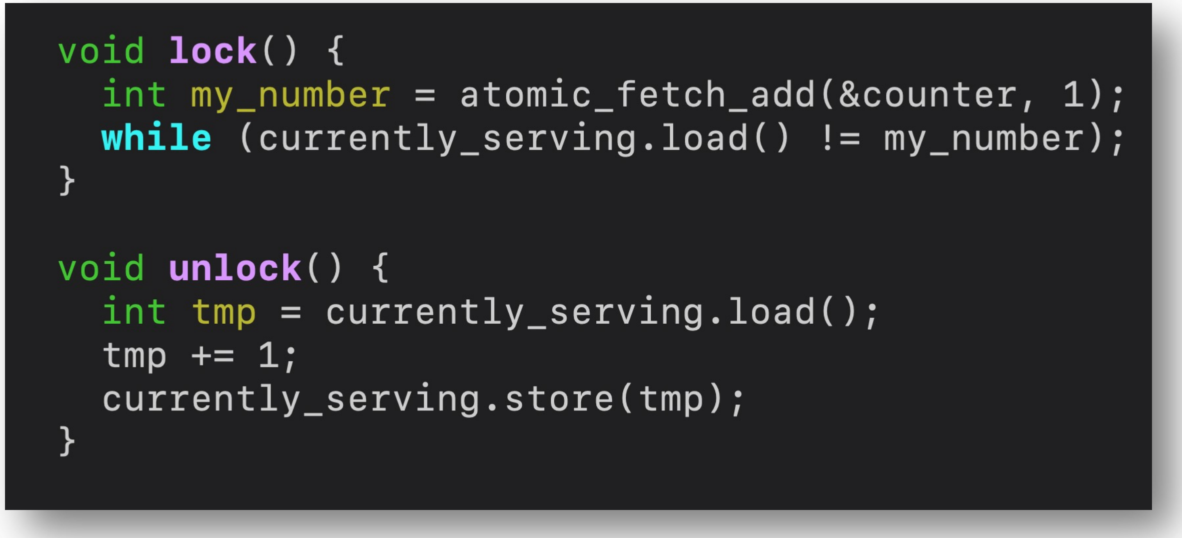
my_number is 1
counter is now 2

spin

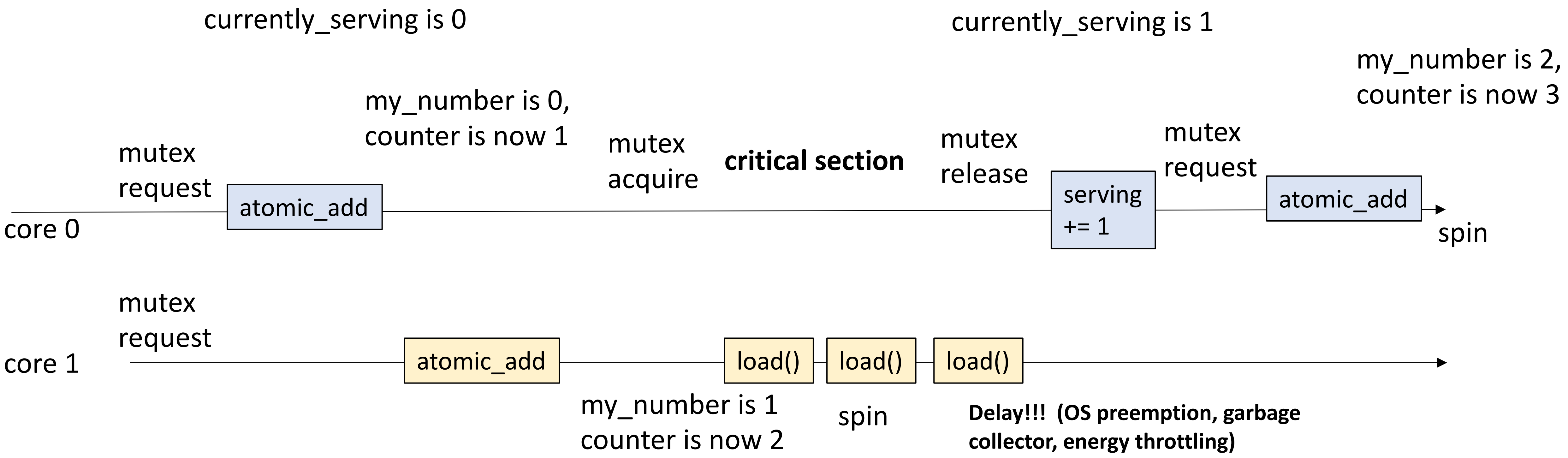
Delay!!! (OS preemption, garbage collector, energy throttling)

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

currently_serving is 1

my_number is 2,
counter is now 3

my_number is 0,
counter is now 1

mutex
request

mutex
acquire

critical section

mutex
release

mutex
request

core 0

atomic_add

serving
+= 1

atomic_add

spin

mutex
request

core 1

atomic_add

load()

load()

load()

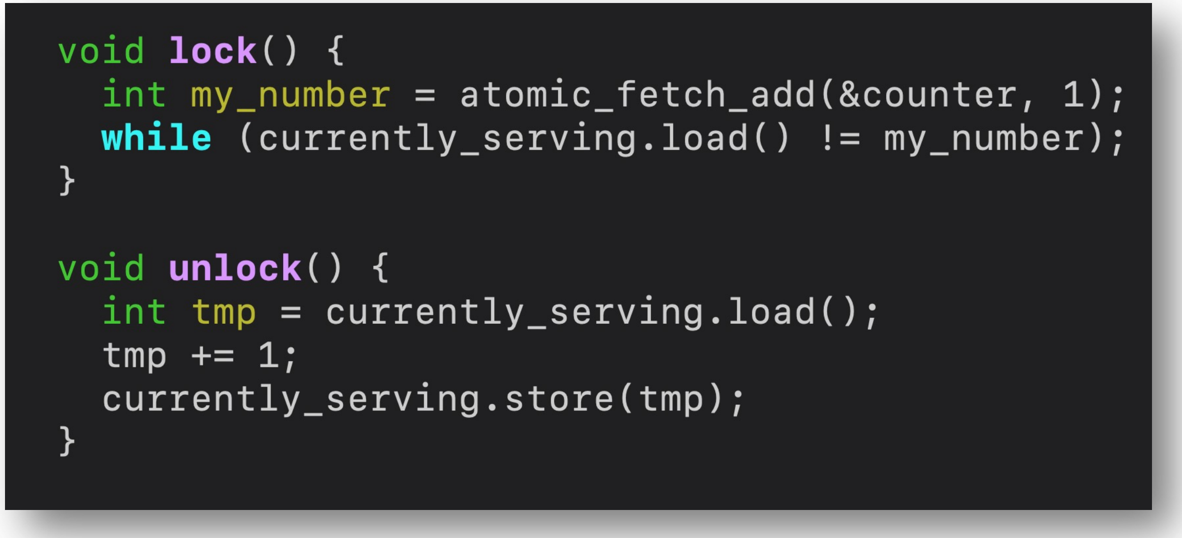
my_number is 1
counter is now 2

spin

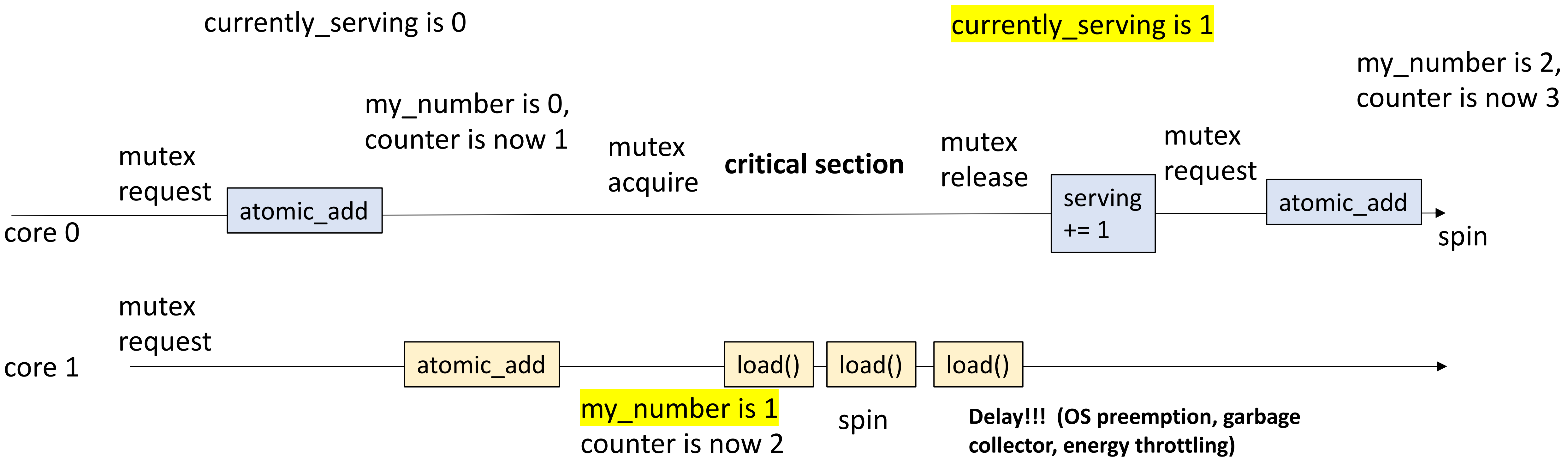
Delay!!! (OS preemption, garbage collector, energy throttling)

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

currently_serving is 1

core 0

mutex request

atomic_add

my_number is 0,
counter is now 1

mutex acquire

critical section

mutex release

serving += 1

mutex request

atomic_add

my_number is 2,
counter is now 3

spin

core 1

mutex request

atomic_add

my_number is 1
counter is now 2

load()

load()

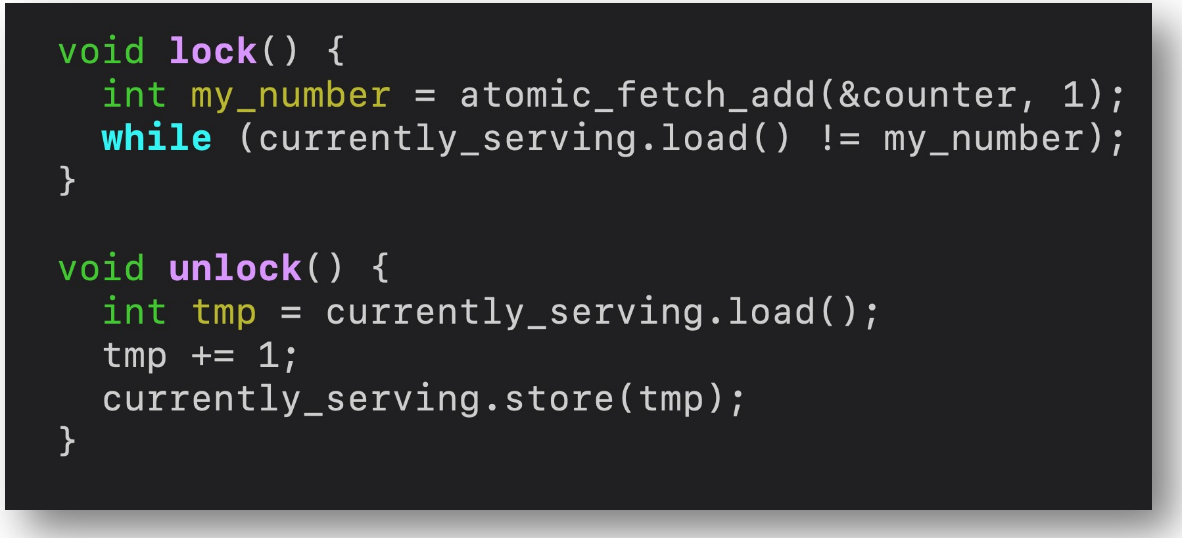
load()

spin

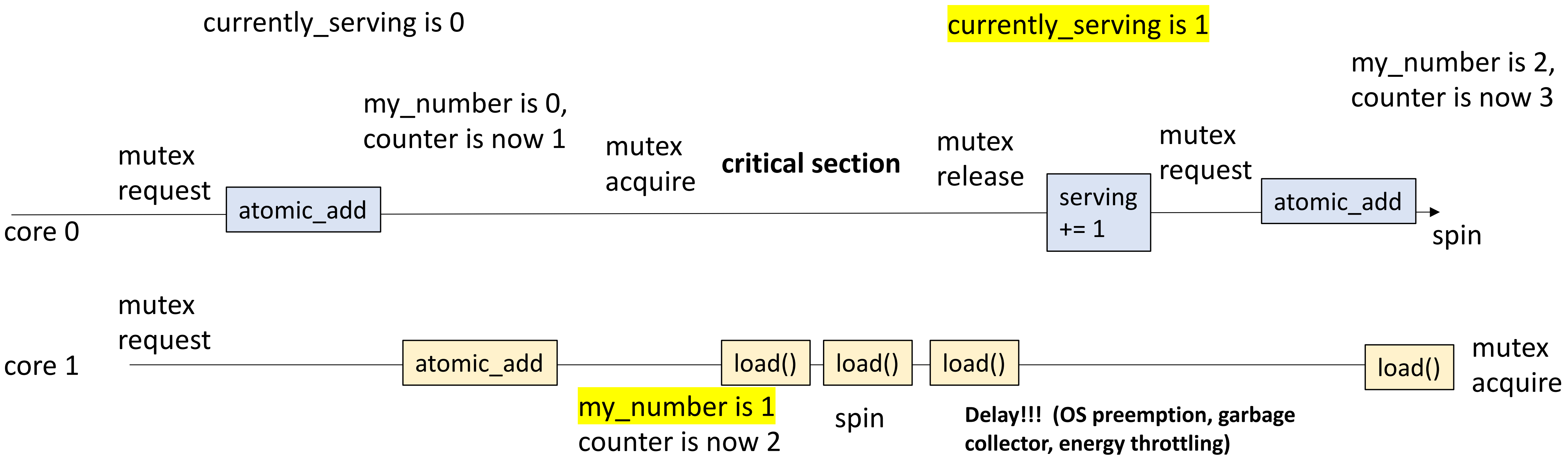
Delay!!! (OS preemption, garbage collector, energy throttling)

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

currently_serving is 1

core 0

mutex request

atomic_add

my_number is 0, counter is now 1

mutex acquire

critical section

mutex release

serving += 1

mutex request

atomic_add

my_number is 2, counter is now 3

spin

core 1

mutex request

atomic_add

my_number is 1

counter is now 2

load()

load()

load()

spin

Delay!!! (OS preemption, garbage collector, energy throttling)

load()

mutex acquire

Fair but at what cost?

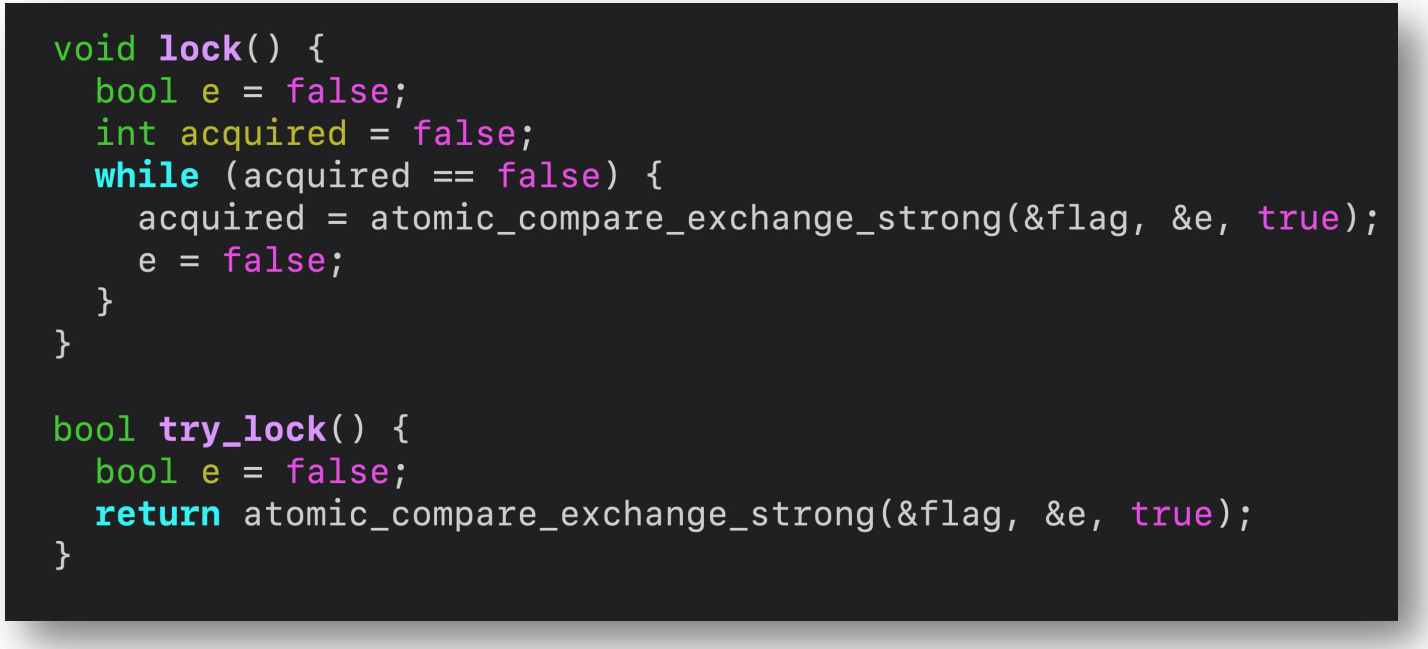
- Example

Schedule

- Fairness of RMW locks
- **Optimization of RMW locks**
- RW mutexes

Optimizations: relaxed peeking

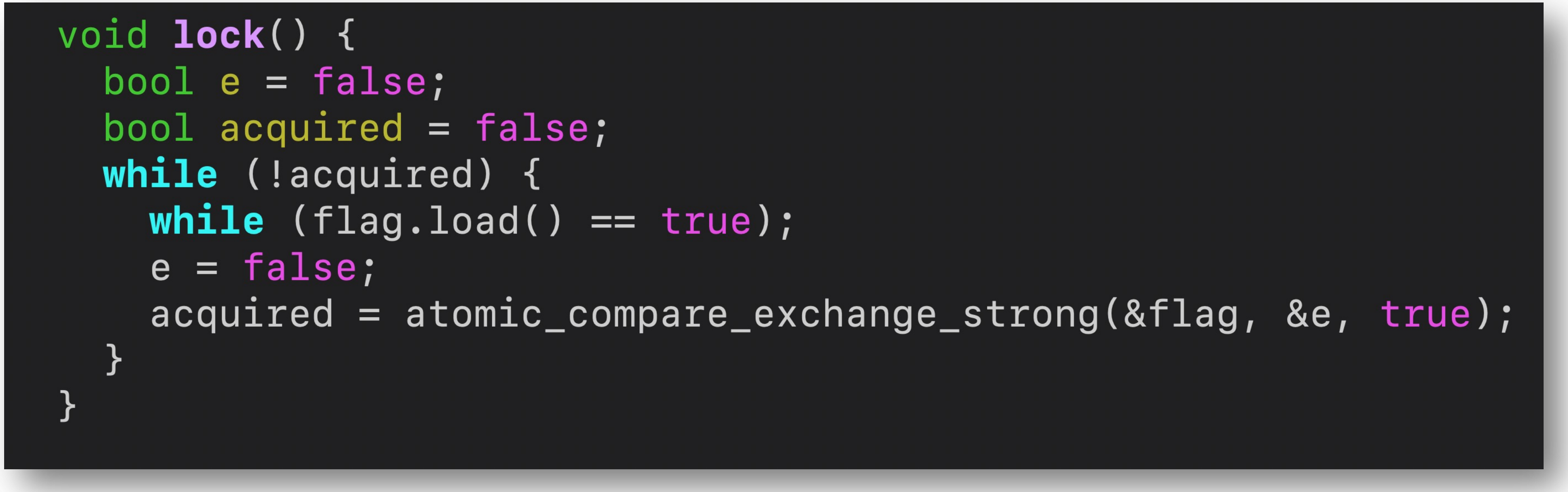
- Relaxed Peeking
    - the Writes in RMWs cost extra; rather than always modify, we can do a simple check first



```
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

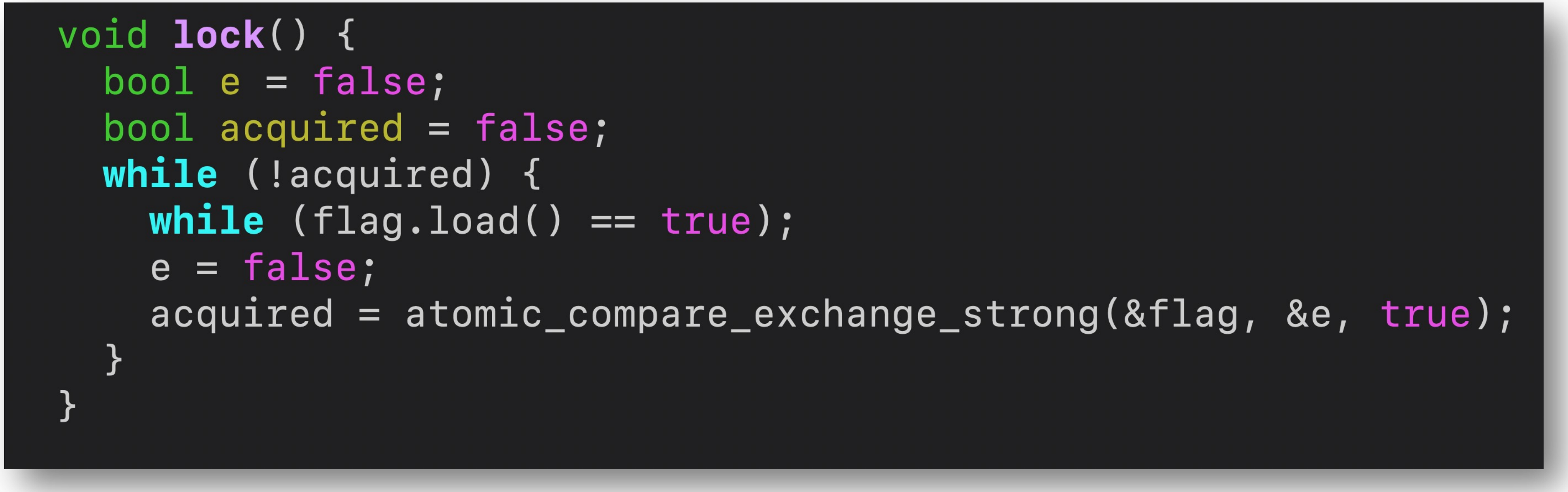
- Relaxed Peeking
  - the Writes in RMWs cost extra; rather than always modify, we can do a simple check first



```
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

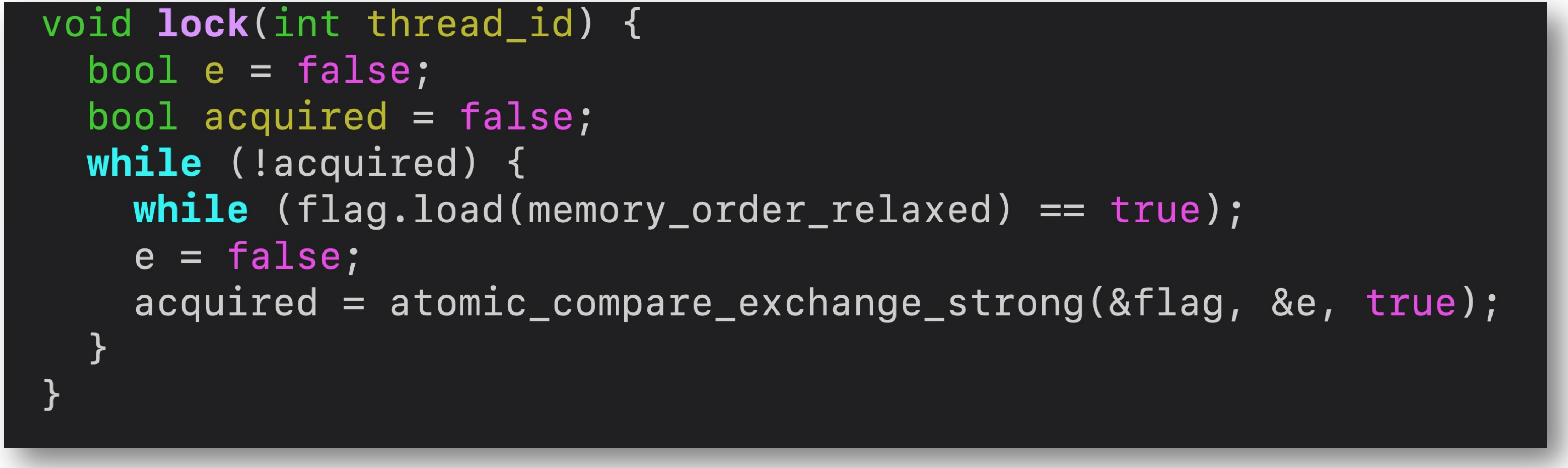
- What about the load in the loop? Remember the memory fence? Do we need to flush our caches every time we peek?
- We only need to flush when we actually acquire the mutex



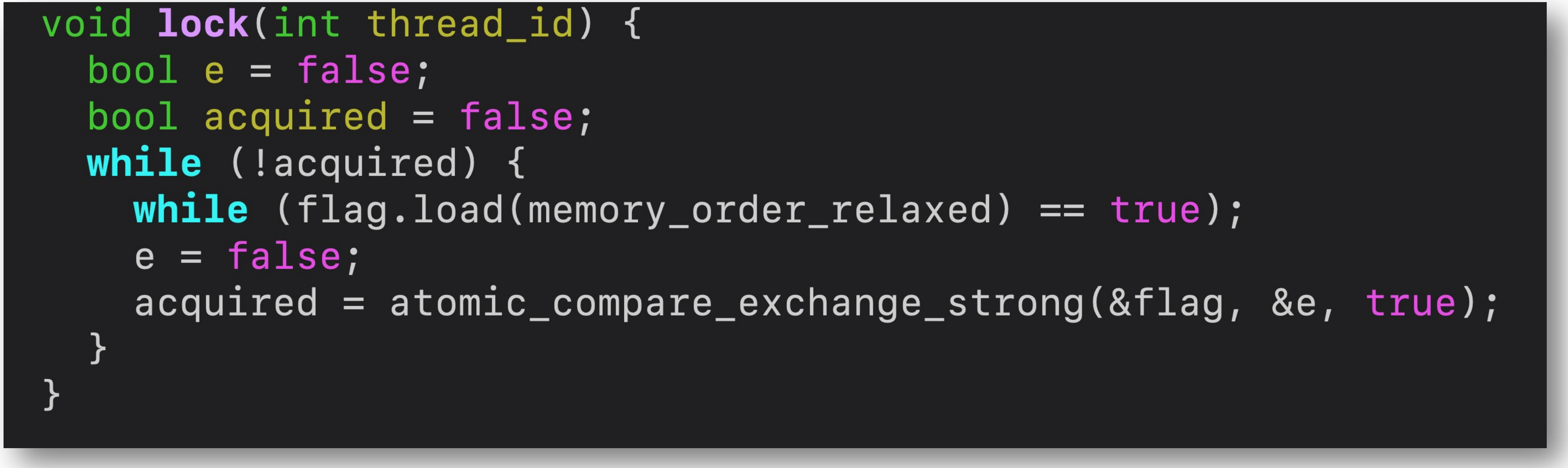
```
void lock() {
  bool e = false;
  bool acquired = false;
  while (!acquired) {
    while (flag.load() == true);
    e = false;
    acquired = atomic_compare_exchange_strong(&flag, &e, true);
  }
}
```

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

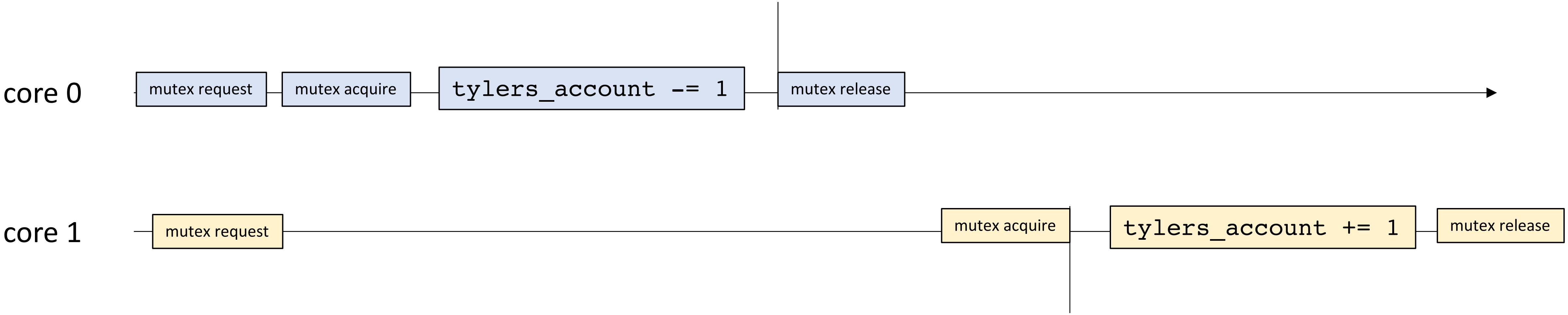


```
void lock(int thread_id) {
  bool e = false;
  bool acquired = false;
  while (!acquired) {
    while (flag.load(memory_order_relaxed) == true);
    e = false;
    acquired = atomic_compare_exchange_strong(&flag, &e, true);
  }
}
```



```
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);
  }
}
```

C0 memory operations are performed and flushed



core 0

mutex request

mutex acquire

`tylers_account -= 1`

mutex release

core 1

mutex request

mutex acquire

`tylers_account += 1`

mutex release

C1 memory operations have **not** yet been performed and cache is invalidated

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.

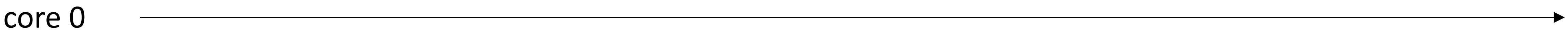
Optimizations: backoff

- Even using relaxed peeking, two issues remain:
    - Loads still cause bus traffic (even if its not as bad as RMWs)
    - In non-parallel systems, concurrent threads can get in the way of progress

Optimizations: backoff

- Even using relaxed peeking, two issues remain:
  - Loads still cause bus traffic (even if its not as bad as RMWs)
  - **In non-parallel systems, concurrent threads can get in the way of progress**

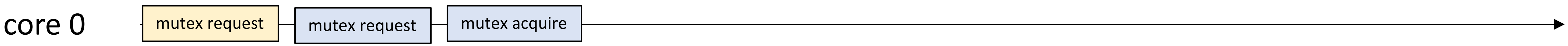
Say threads 0 and 1 are executing concurrently



Optimizations: backoff

- Even using relaxed peeking, two issues remain:
  - Loads still cause bus traffic (even if its not as bad as RMWs)
  - **In non-parallel systems, concurrent threads can get in the way of progress**

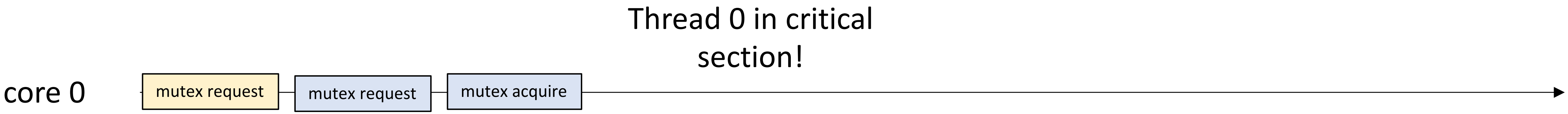
Say threads 0 and 1 are executing concurrently



Optimizations: backoff

- Even using relaxed peeking, two issues remain:
  - Loads still cause bus traffic (even if its not as bad as RMWs)
  - **In non-parallel systems, concurrent threads can get in the way of progress**

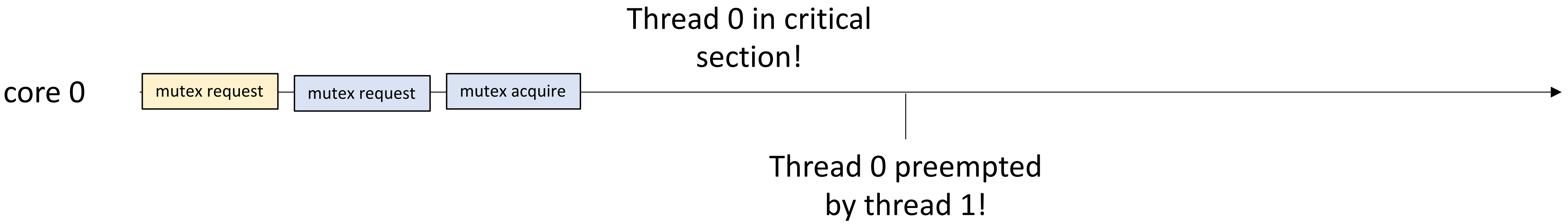
Say threads 0 and 1 are executing concurrently



Optimizations: backoff

- Even using relaxed peeking, two issues remain:
  - Loads still cause bus traffic (even if its not as bad as RMWs)
  - **In non-parallel systems, concurrent threads can get in the way of progress**

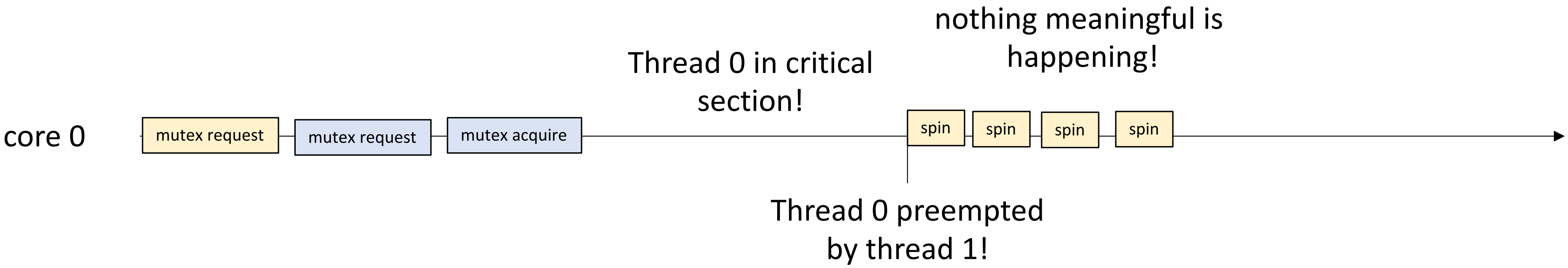
Say threads 0 and 1 are executing concurrently


Thread 0 in critical section!
core 0
mutex request
mutex request
mutex acquire
Thread 0 preempted by thread 1!

Optimizations: backoff

- Two issues remain:
  - Loads still cause bus traffic (even if its not as bad as RMWs)
  - **In non-parallel systems, concurrent threads can get in the way of progress**

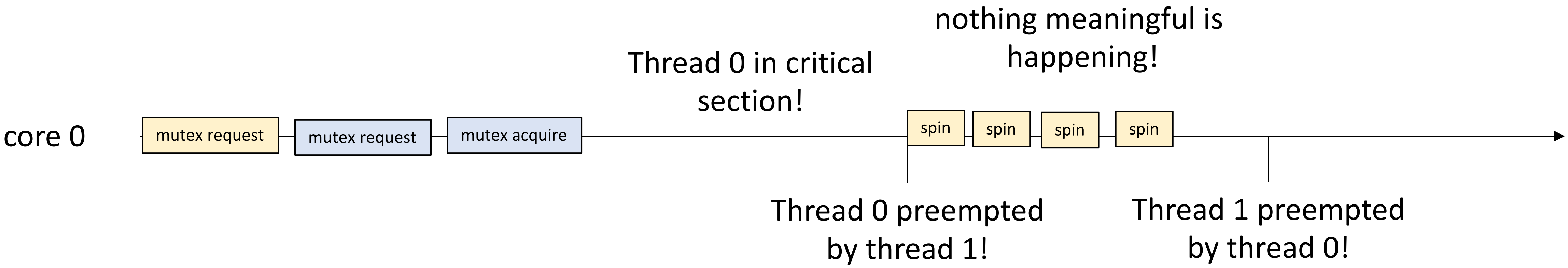
Say threads 0 and 1 are executing concurrently


nothing meaningful is happening!
Thread 0 in critical section!
core 0
mutex request
mutex request
mutex acquire
spin
spin
spin
spin
Thread 0 preempted by thread 1!

Optimizations: backoff

- Two issues remain:
  - Loads still cause bus traffic (even if its not as bad as RMWs)
  - **In non-parallel systems, concurrent threads can get in the way of progress**

Say threads 0 and 1 are executing concurrently

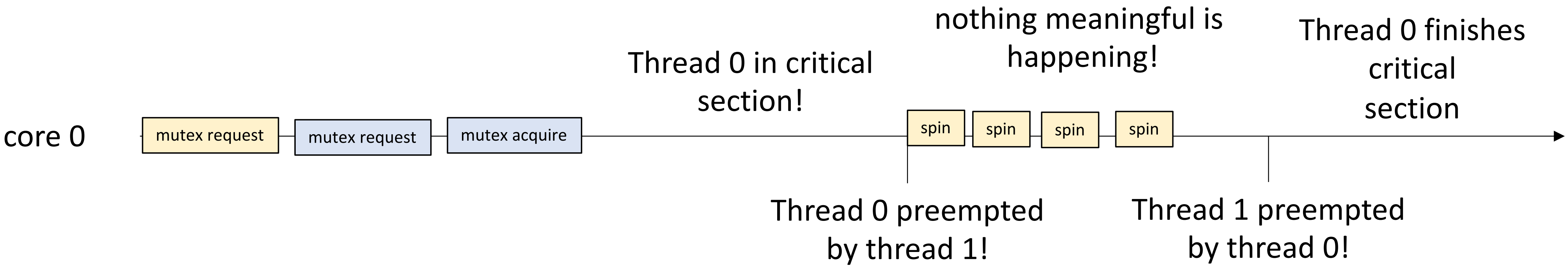


nothing meaningful is happening!
Thread 0 in critical section!
core 0
mutex request
mutex request
mutex acquire
spin
spin
spin
spin
Thread 0 preempted by thread 1!
Thread 1 preempted by thread 0!

Optimizations: backoff

- Two issues remain:
  - Loads still cause bus traffic (even if its not as bad as RMWs)
  - **In non-parallel systems, concurrent threads can get in the way of progress**

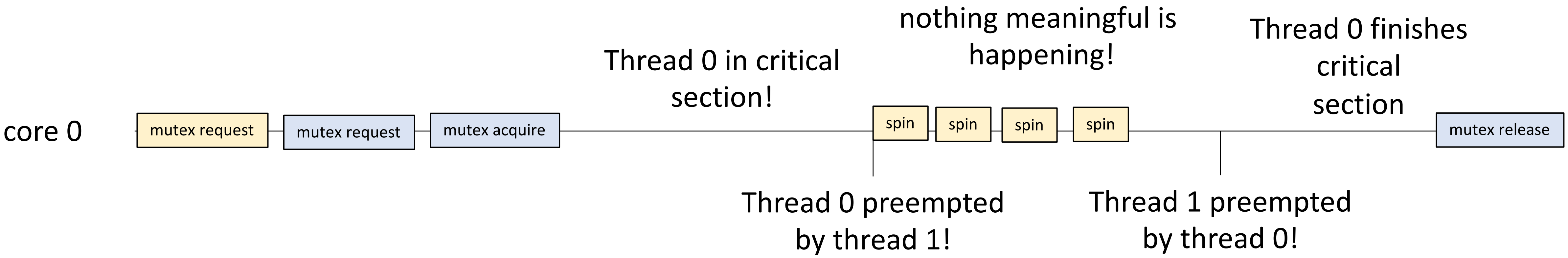
Say threads 0 and 1 are executing concurrently


core 0
mutex request
mutex request
mutex acquire
Thread 0 in critical section!
spin
spin
spin
spin
nothing meaningful is happening!
Thread 0 finishes critical section
Thread 0 preempted by thread 1!
Thread 1 preempted by thread 0!

Optimizations: backoff

- Two issues remain:
  - Loads still cause bus traffic (even if its not as bad as RMWs)
  - **In non-parallel systems, concurrent threads can get in the way of progress**

Say threads 0 and 1 are executing concurrently

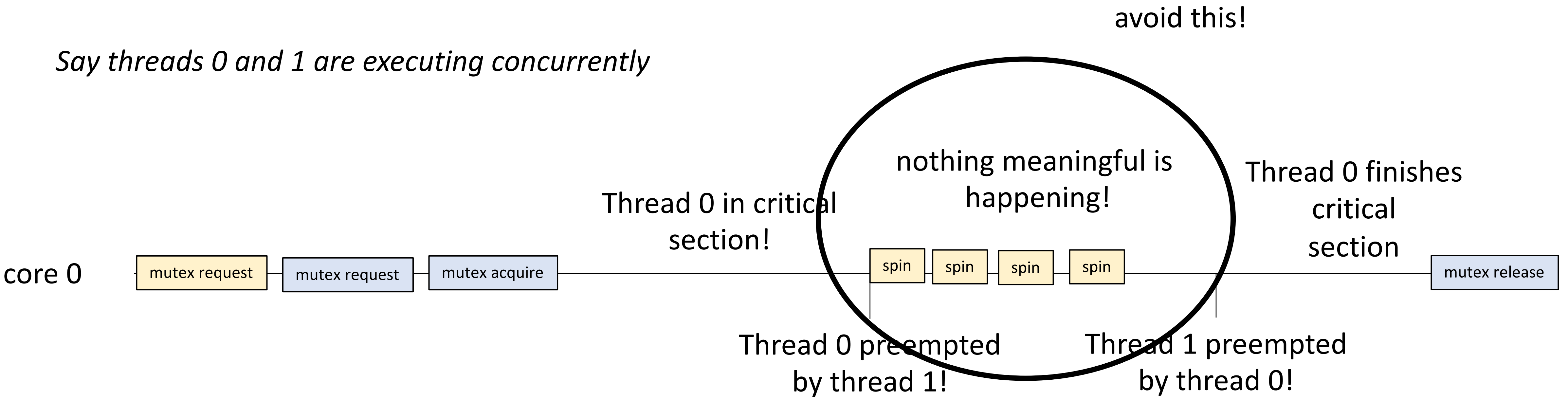


core 0
mutex request
mutex request
mutex acquire
Thread 0 in critical section!
spin
spin
spin
spin
nothing meaningful is happening!
Thread 0 preempted by thread 1!
Thread 1 preempted by thread 0!
Thread 0 finishes critical section
mutex release

Optimizations: backoff

- Two issues remain:
  - Loads still cause bus traffic (even if its not as bad as RMWs)
  - **In non-parallel systems, concurrent threads can get in the way of progress**

Say threads 0 and 1 are executing concurrently


avoid this!
core 0
mutex request
mutex request
mutex acquire
Thread 0 in critical section!
nothing meaningful is happening!
spin
spin
spin
spin
Thread 0 finishes critical section
mutex release
Thread 0 preempted by thread 1!
Thread 1 preempted by thread 0!

Optimizations: backoff

- C++
  - `this_thread::yield();`

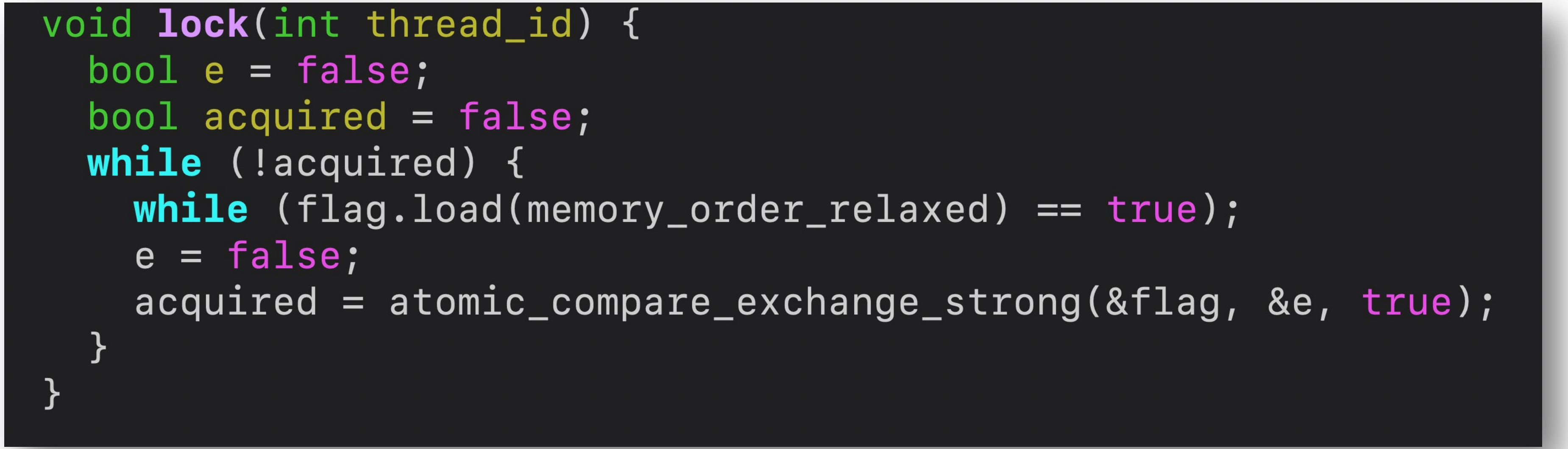
- Hints to the operating system that we should take a break while other threads (potentially the threads that have the mutex) get scheduled.

Optimizations: backoff

where do we put it?

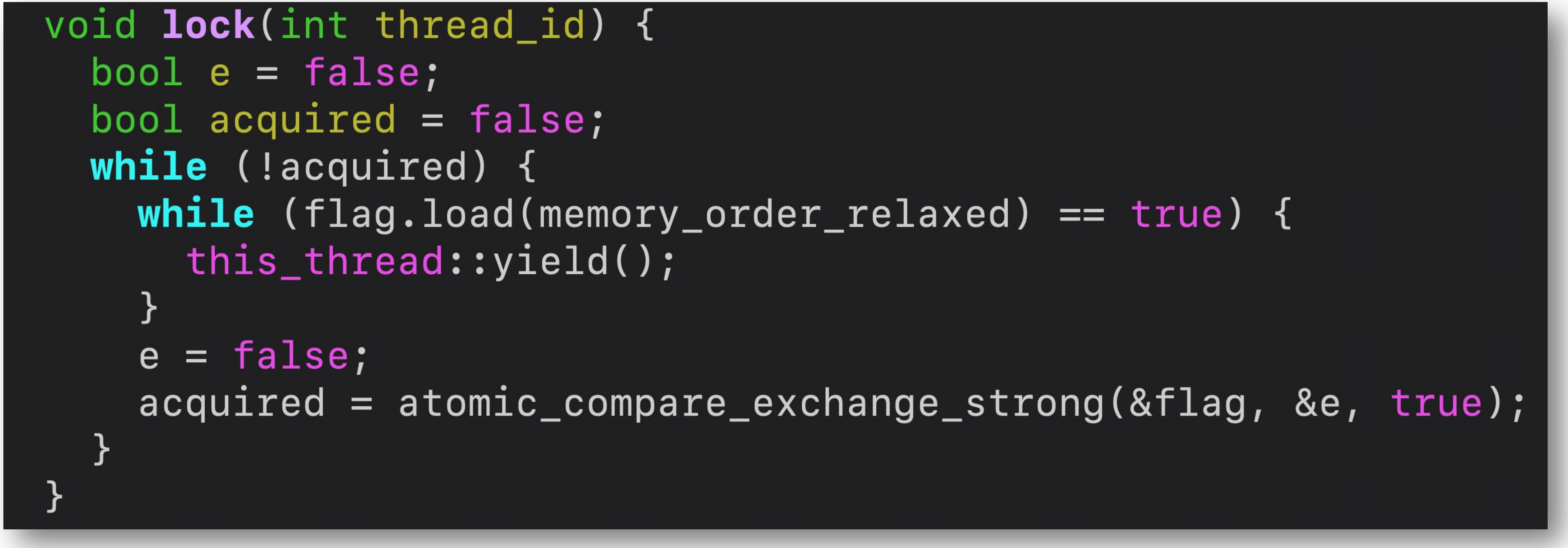
- C++
  - `this_thread::yield();`

- Hints to the operating system that we should take a break while other threads (potentially the threads that have the mutex) get scheduled.



```
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);
  }
}
```

Optimizations: backoff



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

Demo

- Example in terminal

Optimizations: backoff

- Other backoff strategies: sleeping
    - `this_thread::sleep_for(10ms);`
    - Finer control over sleep time

- Exponential backoff:
    - Every time the thread wakes up, sleep for 2x as long

- Tuned sleep time:
    - Keep track of a sleep time.
    - Every time you spin, increase the sleep time (remember for next spin)
    - If you acquire, reduce the sleep time

Optimizations: when to use them

- **Spinning** is useful for short waits on non-oversubscribed systems

- **Sleeping** is useful for regular tasks
  - tasks occur at set frequencies
  - critical sections take roughly the same time
  - In these cases, sleep times can be tuned

- **Yielding** is useful for oversubscribed systems, with irregular tasks
  - On modern systems, yield is usually sufficient!

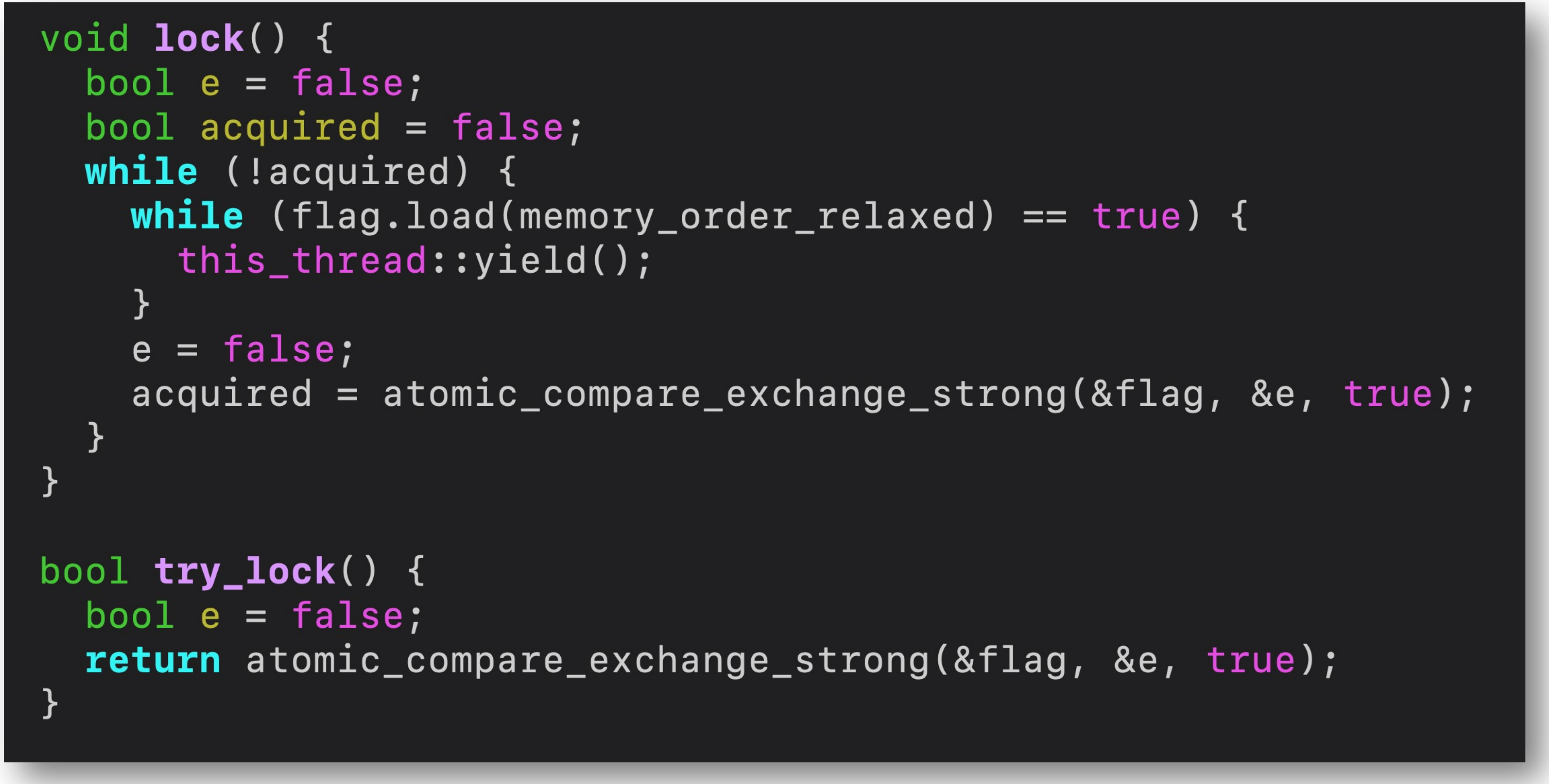
Optimizations: when to use them

- When to use what optimization?
  - Start with C++ mutex, then
  - microbenchmark
  - profile

- Sometimes we want our own custom backoff strategies.
  - We can optimize around existing mutexes!

try_lock

- another common mutex API method: try_lock()
- one-shot mutex attempt (implementation defined)
- You can then implement your own sleep/yield strategy around this

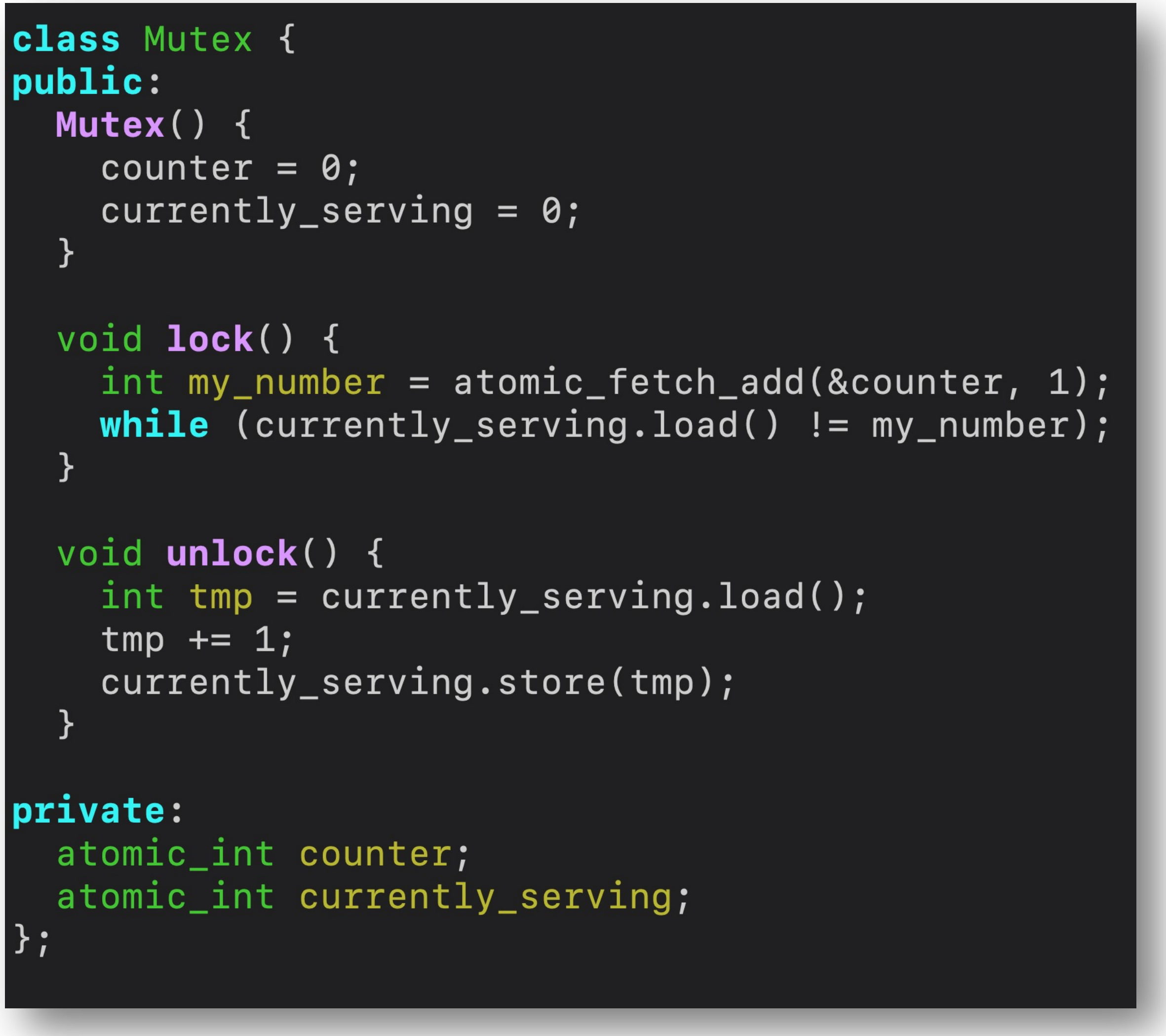


```
void lock() {
  bool e = false;
  bool acquired = false;
  while (!acquired) {
    while (flag.load(memory_order_relaxed) == true) {
      this_thread::yield();
    }
    e = false;
    acquired = atomic_compare_exchange_strong(&flag, &e, true);
  }
}

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

try_lock

- straightforward with CAS and exchange mutex

- What about 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;
};
```

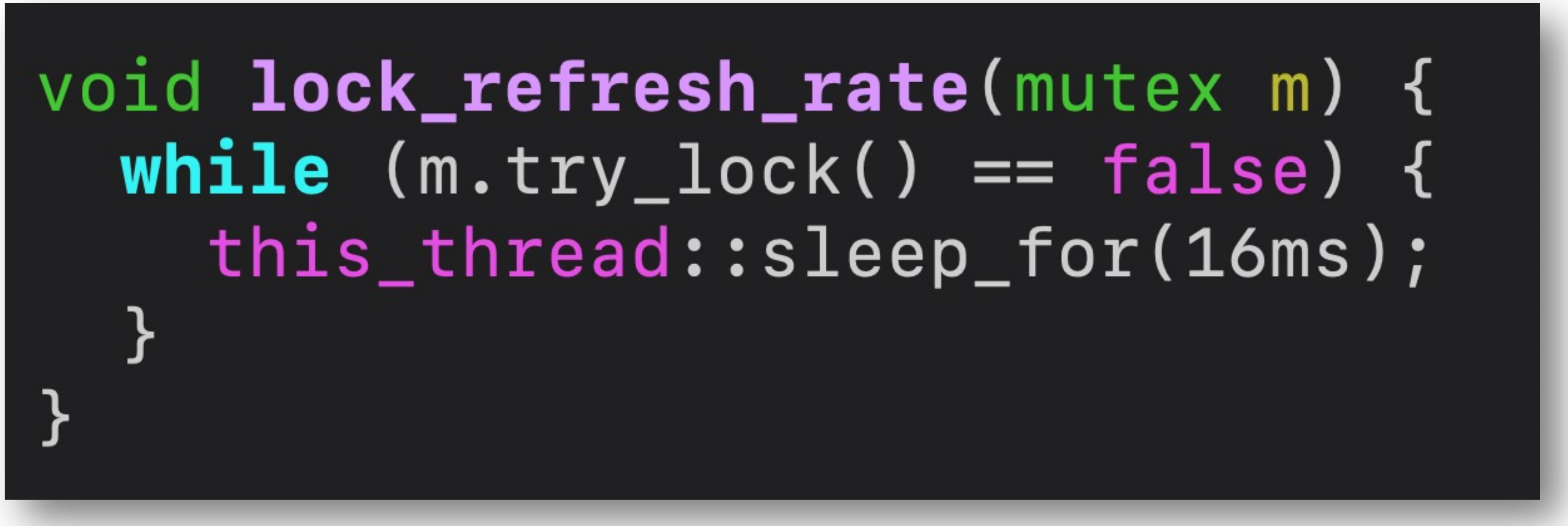
Example: UI refresh

- Screen refreshes operate at ~60 FPS.

- Assume a situation where there is mutex for the screen buffer. It can be updated by one thread, once per frame.

- We know that the sleep will be ~16ms

Example: UI refresh



```
void lock_refresh_rate(mutex m) {
  while (m.try_lock() == false) {
    this_thread::sleep_for(16ms);
  }
}
```

try_lock

- C++ provides a try_lock for their mutex operation

- We have now covered the entire C++ mutex object

Schedule

- Fairness of RMW locks
- Optimization of RMW locks
- **RW mutexes**

Reader-Writer Mutex

Global variable: `int tylers_account`

```
void buy_coffee() {
    tylers_account--;
}
```

```
void get_paid() {
    tylers_account++;
}
```

Reader-Writer Mutex

Global variable: `int tylers_account`

But what happens more frequently than either of those things?

```
void buy_coffee() {
    tylers_account--;
}
```

```
void get_paid() {
    tylers_account++;
}
```

Reader-Writer Mutex

Global variable: `int tylers_account`

```
void buy_coffee() {
    tylers_account--;
}
```

```
void get_paid() {
    tylers_account++;
}
```

which of these operations can safely be executed concurrently?

Remember the definition of a data-conflict: at least one write

But what happens more frequently than either of those things?

```
int check_balance() {
    return tylers_account;
}
```

Different actors accessing it concurrently

Credit monitors

Accountants

Personal

Reader-Writer Mutex

Global variable: `int tylers_account`

```
void buy_coffee() {
    tylers_account--;
}
```

```
void get_paid() {
    tylers_account++;
}
```

But what happens more frequently than either of those things?

```
int check_balance() {
    return tylers_account;
}
```

No reason why this function can’t be called concurrently. It only needs to be protected if another thread calls one of the other functions.

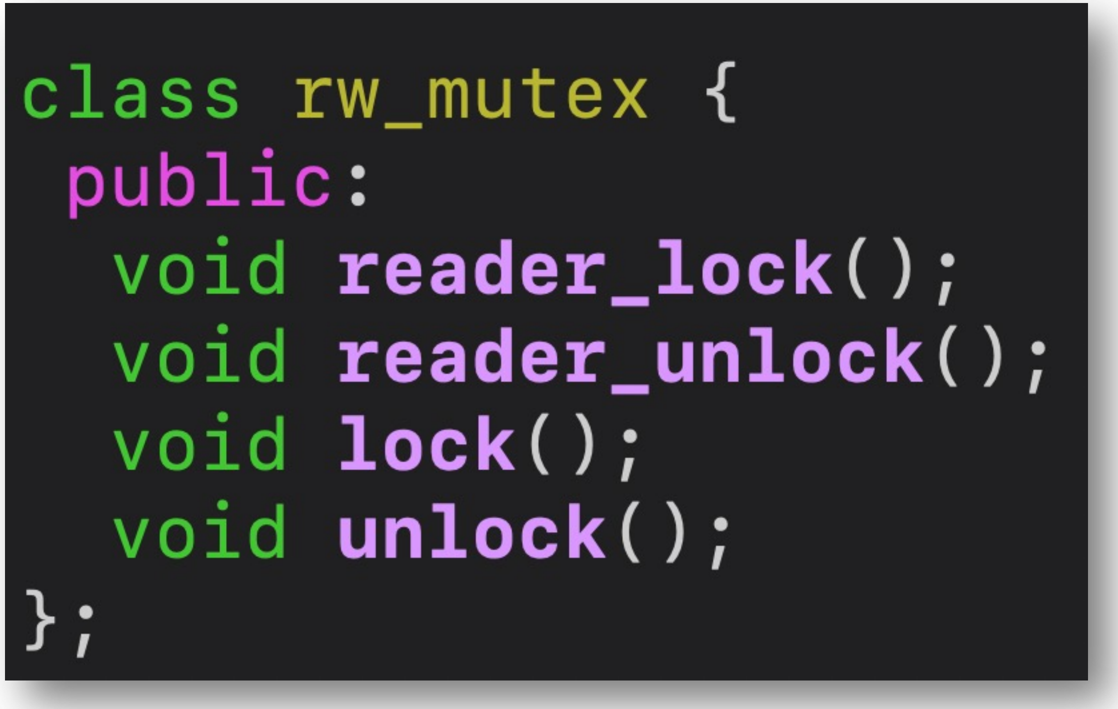
Reader-Writer Mutex

- different lock and unlock functions:
    - Functions that only read can perform a “read” lock
    - Functions that might write can perform a regular lock

    - regular locks ensures that the writer has exclusive access (from other reader and writers)

    - but multiple reader threads can hold the lock in reader state

Reader-Writer Mutex



```
class rw_mutex {
 public:
  void reader_lock();
  void reader_unlock();
  void lock();
  void unlock();
};
```

Reader-Writer Mutex

Global variable: `int tylers_account`

```
void buy_coffee() {
    tylers_account--;
}
```

```
void get_paid() {
    tylers_account++;
}
```

```
int check_balance() {
    return tylers_account;
}
```

Reader-Writer Mutex

Global variable: `int tylers_account`

```
void buy_coffee() {
    m.lock();
    tylers_account--;
    m.unlock();
}
```

```
void get_paid() {
    m.lock();
    tylers_account++;
    m.unlock();
}
```

```
int check_balance() {
    return tylers_account;
}
```

Reader-Writer Mutex

Global variable: `int tylers_account`

```
void buy_coffee() {
    m.lock();
    tylers_account--;
    m.unlock();
}
```

```
void get_paid() {
    m.lock();
    tylers_account++;
    m.unlock();
}
```

```
int check_balance() {
    m.reader_lock();
    int t = tylers_account;
    m.reader_unlock();
    return t;
}
```

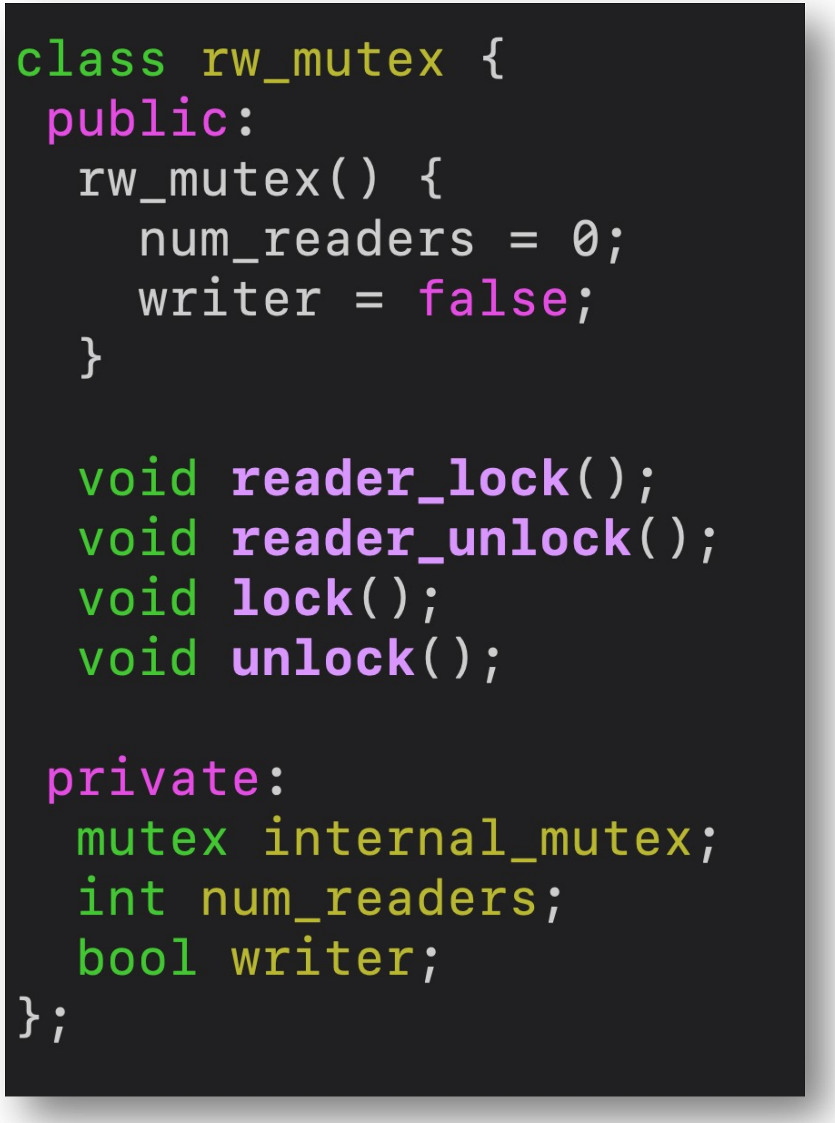
Reader-Writer Mutex Implementation

- Primitives that we built the previous mutexes with:
  - atomic load, atomic store, atomic RMW

- We have a new tool!
  - Regular mutex!

Reader-Writer Mutex Implementation

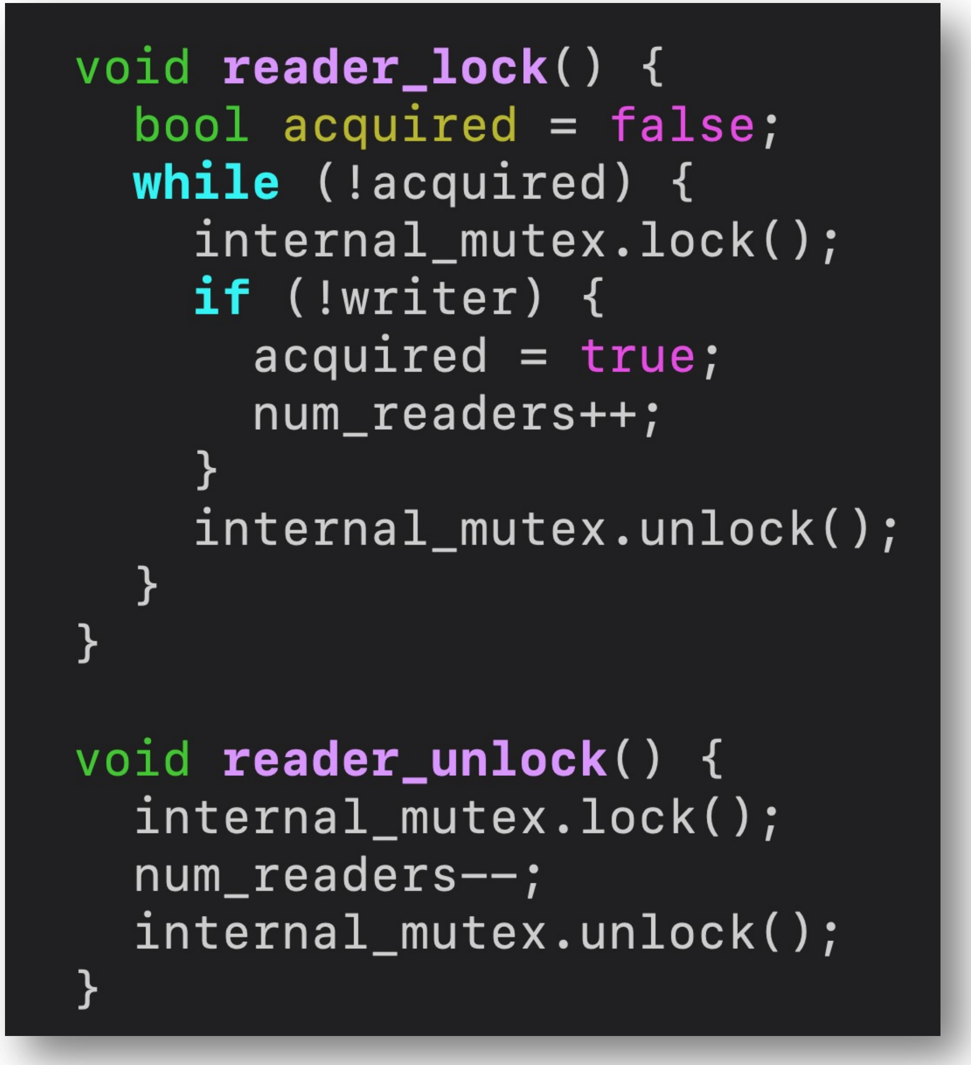
- We will use a mutex internally.
- We will keep track of how many readers are currently “holding” the mutex.
- We will keep track of if a writer is holding the mutex.



```
class rw_mutex {
 public:
  rw_mutex() {
    num_readers = 0;
    writer = false;
  }

  void reader_lock();
  void reader_unlock();
  void lock();
  void unlock();

 private:
  mutex internal_mutex;
  int num_readers;
  bool writer;
};
```

Reader-Writer Mutex Implementation

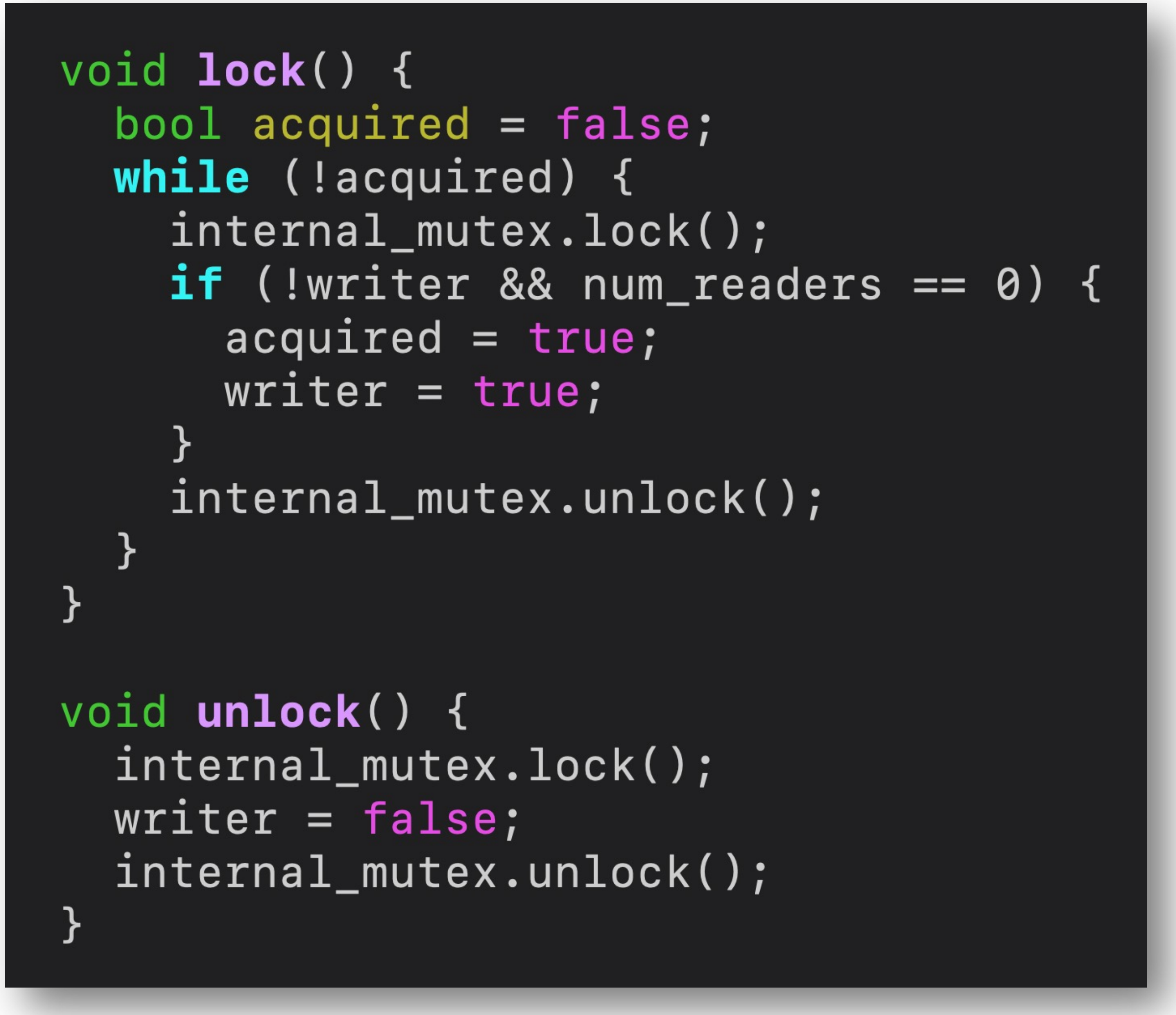
- Reader locks



```
void reader_lock() {
  bool acquired = false;
  while (!acquired) {
    internal_mutex.lock();
    if (!writer) {
      acquired = true;
      num_readers++;
    }
    internal_mutex.unlock();
  }
}

void reader_unlock() {
  internal_mutex.lock();
  num_readers--;
  internal_mutex.unlock();
}
```

Reader-Writer Mutex Implementation

- Regular locks



```
void lock() {
  bool acquired = false;
  while (!acquired) {
    internal_mutex.lock();
    if (!writer && num_readers == 0) {
      acquired = true;
      writer = true;
    }
    internal_mutex.unlock();
  }
}

void unlock() {
  internal_mutex.lock();
  writer = false;
  internal_mutex.unlock();
}
```

Thread 0

```
void buy_coffee() {
    m.lock();
    tylers_account--;
    m.unlock();
}
```

Thread 1

```
void get_paid() {
    m.lock();
    tylers_account++;
    m.unlock();
}
```

Thread 2

```
int check_balance() {
    m.reader_lock();
    int t = tylers_account;
    m.reader_unlock();
    return t;
}
```

Thread 3

```
int check_balance() {
    m.reader_lock();
    int t = tylers_account;
    m.reader_unlock();
    return t;
}
```

writer = false
num_readers = 0

Thread 0

```
void buy_coffee() {
    m.lock();
    tylers_account--;
    m.unlock();
}
```

Thread 1

```
void get_paid() {
    m.lock();
    tylers_account++;
    m.unlock();
}
```

Thread 2

```
int check_balance() {
    m.reader_lock();
    int t = tylers_account;
    m.reader_unlock();
    return t;
}
```

Thread 3

```
int check_balance() {
    m.reader_lock();
    int t = tylers_account;
    m.reader_unlock();
    return t;
}
```

writer = false
num_readers = 0

Thread 0

```
void buy_coffee() {
    m.lock();
    tylers_account--;
    m.unlock();
}
```

Thread 1

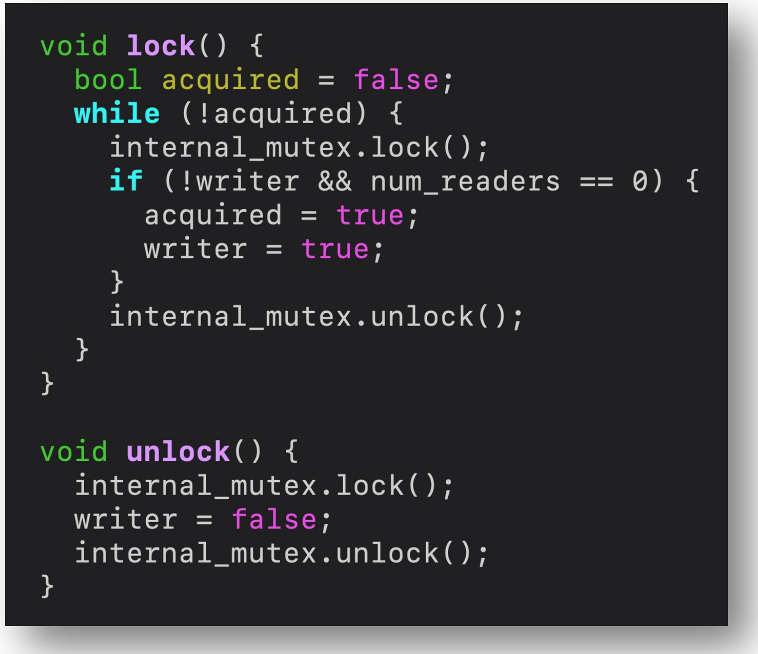
```
void get_paid() {
    m.lock();
    tylers_account++;
    m.unlock();
}
```

Thread 2

```
int check_balance() {
    m.reader_lock();
    int t = tylers_account;
    m.reader_unlock();
    return t;
}
```

Thread 3

```
int check_balance() {
    m.reader_lock();
    int t = tylers_account;
    m.reader_unlock();
    return t;
}
```



```
void lock() {
  bool acquired = false;
  while (!acquired) {
    internal_mutex.lock();
    if (!writer && num_readers == 0) {
      acquired = true;
      writer = true;
    }
    internal_mutex.unlock();
  }
}

void unlock() {
  internal_mutex.lock();
  writer = false;
  internal_mutex.unlock();
}
```

writer = false
num_readers = 0

Thread 0

```
void buy_coffee() {
    m.lock();
    tylers_account--;
    m.unlock();
}
```

Thread 1

```
void get_paid() {
    m.lock();
    tylers_account++;
    m.unlock();
}
```

Thread 2

```
int check_balance() {
    m.reader_lock();
    int t = tylers_account;
    m.reader_unlock();
    return t;
}
```

Thread 3

```
int check_balance() {
    m.reader_lock();
    int t = tylers_account;
    m.reader_unlock();
    return t;
}
```

writer = true
num_readers = 0

Thread 0

```
void buy_coffee() {
    m.lock();
    tylers_account--;
    m.unlock();
}
```

Thread 1

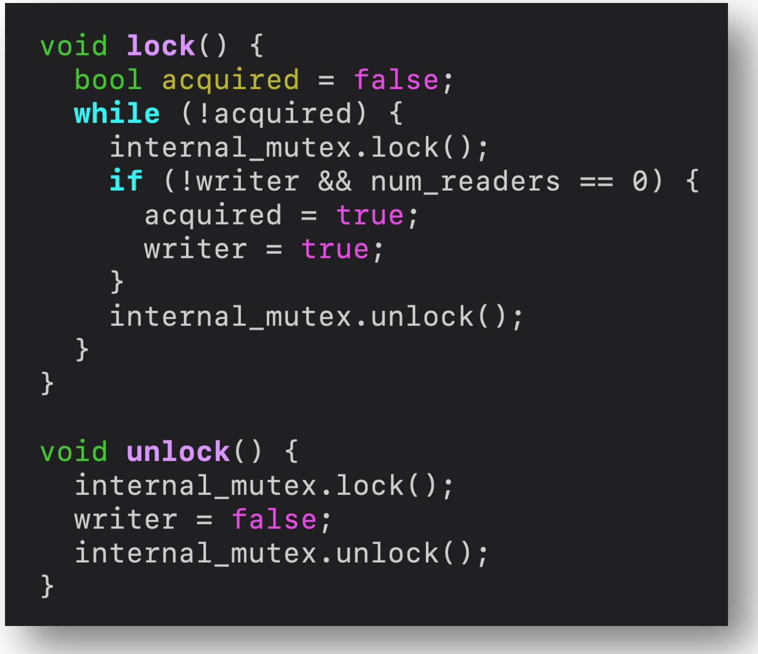
```
void get_paid() {
    m.lock();
    tylers_account++;
    m.unlock();
}
```

Thread 2

```
int check_balance() {
    m.reader_lock();
    int t = tylers_account;
    m.reader_unlock();
    return t;
}
```

Thread 3

```
int check_balance() {
    m.reader_lock();
    int t = tylers_account;
    m.reader_unlock();
    return t;
}
```



```
void lock() {
  bool acquired = false;
  while (!acquired) {
    internal_mutex.lock();
    if (!writer && num_readers == 0) {
      acquired = true;
      writer = true;
    }
    internal_mutex.unlock();
  }
}

void unlock() {
  internal_mutex.lock();
  writer = false;
  internal_mutex.unlock();
}
```

writer = true
num_readers = 0

Thread 0

```
void buy_coffee() {
    m.lock();
    tylers_account--;
    m.unlock();
}
```

Thread 1

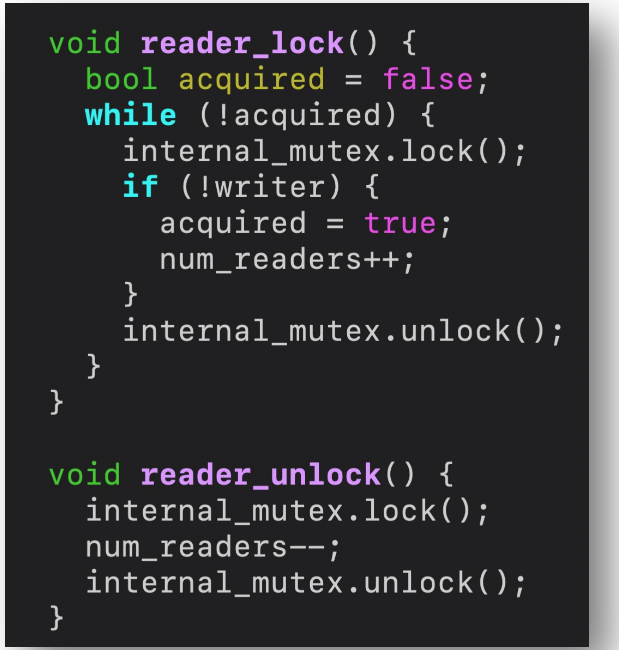
```
void get_paid() {
    m.lock();
    tylers_account++;
    m.unlock();
}
```

Thread 2

```
int check_balance() {
    m.reader_lock();
    int t = tylers_account;
    m.reader_unlock();
    return t;
}
```

Thread 3

```
int check_balance() {
    m.reader_lock();
    int t = tylers_account;
    m.reader_unlock();
    return t;
}
```



```
void reader_lock() {
  bool acquired = false;
  while (!acquired) {
    internal_mutex.lock();
    if (!writer) {
      acquired = true;
      num_readers++;
    }
    internal_mutex.unlock();
  }
}

void reader_unlock() {
  internal_mutex.lock();
  num_readers--;
  internal_mutex.unlock();
}
```

writer = true
num_readers = 0

reset!

Thread 0

```
void buy_coffee() {
    m.lock();
    tylers_account--;
    m.unlock();
}
```

Thread 1

```
void get_paid() {
    m.lock();
    tylers_account++;
    m.unlock();
}
```

Thread 2

```
int check_balance() {
    m.reader_lock();
    int t = tylers_account;
    m.reader_unlock();
    return t;
}
```

Thread 3

```
int check_balance() {
    m.reader_lock();
    int t = tylers_account;
    m.reader_unlock();
    return t;
}
```

writer = False
num_readers = 0

Thread 0

```
void buy_coffee() {
    m.lock();
    tylers_account--;
    m.unlock();
}
```

Thread 1

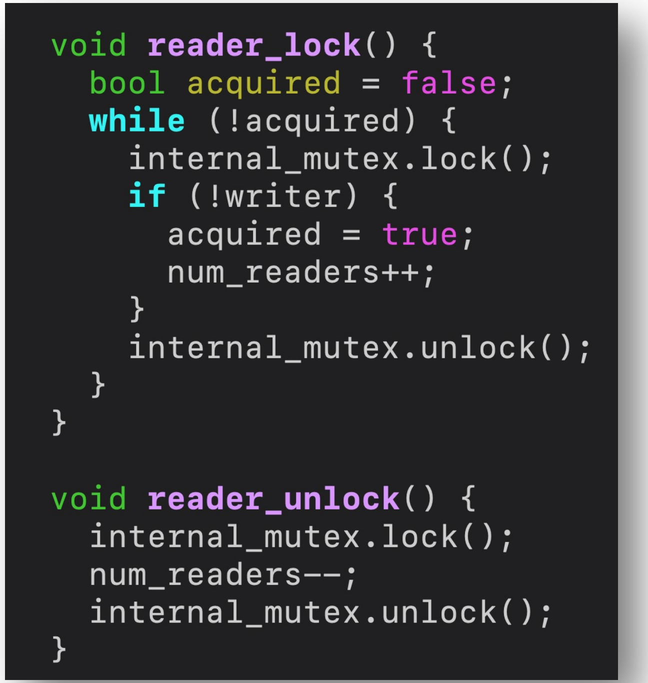
```
void get_paid() {
    m.lock();
    tylers_account++;
    m.unlock();
}
```

Thread 2

```
int check_balance() {
    m.reader_lock();
    int t = tylers_account;
    m.reader_unlock();
    return t;
}
```

Thread 3

```
int check_balance() {
    m.reader_lock();
    int t = tylers_account;
    m.reader_unlock();
    return t;
}
```



```
void reader_lock() {
  bool acquired = false;
  while (!acquired) {
    internal_mutex.lock();
    if (!writer) {
      acquired = true;
      num_readers++;
    }
    internal_mutex.unlock();
  }
}

void reader_unlock() {
  internal_mutex.lock();
  num_readers--;
  internal_mutex.unlock();
}
```

writer = False
num_readers = 0

Thread 0

```
void buy_coffee() {
    m.lock();
    tylers_account--;
    m.unlock();
}
```

Thread 1

```
void get_paid() {
    m.lock();
    tylers_account++;
    m.unlock();
}
```

Thread 2

```
int check_balance() {
    m.reader_lock();
    int t = tylers_account;
    m.reader_unlock();
    return t;
}
```

Thread 3

```
int check_balance() {
    m.reader_lock();
    int t = tylers_account;
    m.reader_unlock();
    return t;
}
```

writer = False
num_readers = 1

Thread 0

```
void buy_coffee() {
    m.lock();
    tylers_account--;
    m.unlock();
}
```

Thread 1

```
void get_paid() {
    m.lock();
    tylers_account++;
    m.unlock();
}
```

Thread 2

```
int check_balance() {
    m.reader_lock();
    int t = tylers_account;
    m.reader_unlock();
    return t;
}
```

Thread 3

```
int check_balance() {
    m.reader_lock();
    int t = tylers_account;
    m.reader_unlock();
    return t;
}
```

writer = False
num_readers = 1

Thread 0

```
void buy_coffee() {
    m.lock();
    tylers_account--;
    m.unlock();
}
```

Thread 1

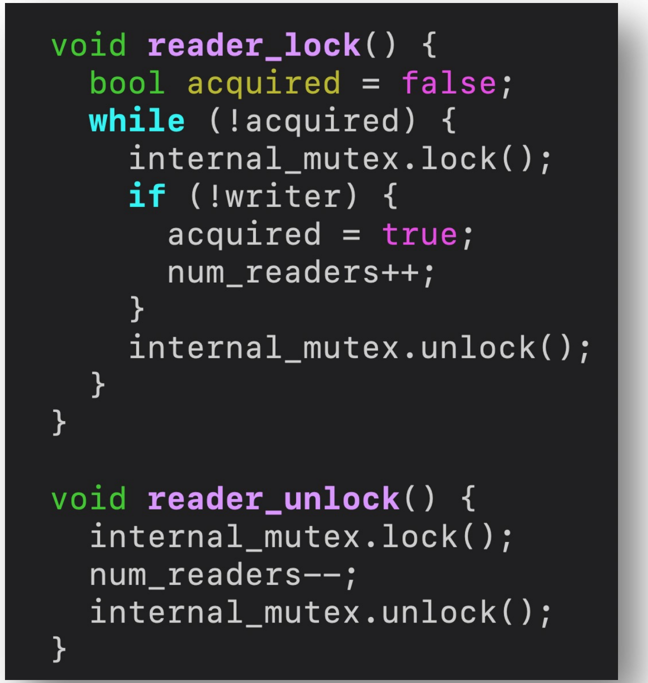
```
void get_paid() {
    m.lock();
    tylers_account++;
    m.unlock();
}
```

Thread 2

```
int check_balance() {
    m.reader_lock();
    int t = tylers_account;
    m.reader_unlock();
    return t;
}
```

Thread 3

```
int check_balance() {
    m.reader_lock();
    int t = tylers_account;
    m.reader_unlock();
    return t;
}
```



```
void reader_lock() {
  bool acquired = false;
  while (!acquired) {
    internal_mutex.lock();
    if (!writer) {
      acquired = true;
      num_readers++;
    }
    internal_mutex.unlock();
  }
}

void reader_unlock() {
  internal_mutex.lock();
  num_readers--;
  internal_mutex.unlock();
}
```

writer = False
num_readers = 1

Thread 0

```
void buy_coffee() {
    m.lock();
    tylers_account--;
    m.unlock();
}
```

Thread 1

```
void get_paid() {
    m.lock();
    tylers_account++;
    m.unlock();
}
```

Thread 2

```
int check_balance() {
    m.reader_lock();
    int t = tylers_account;
    m.reader_unlock();
    return t;
}
```

Thread 3

```
int check_balance() {
    m.reader_lock();
    int t = tylers_account;
    m.reader_unlock();
    return t;
}
```

writer = False
num_readers = 2

Thread 0

```
void buy_coffee() {
    m.lock();
    tylers_account--;
    m.unlock();
}
```

Thread 1

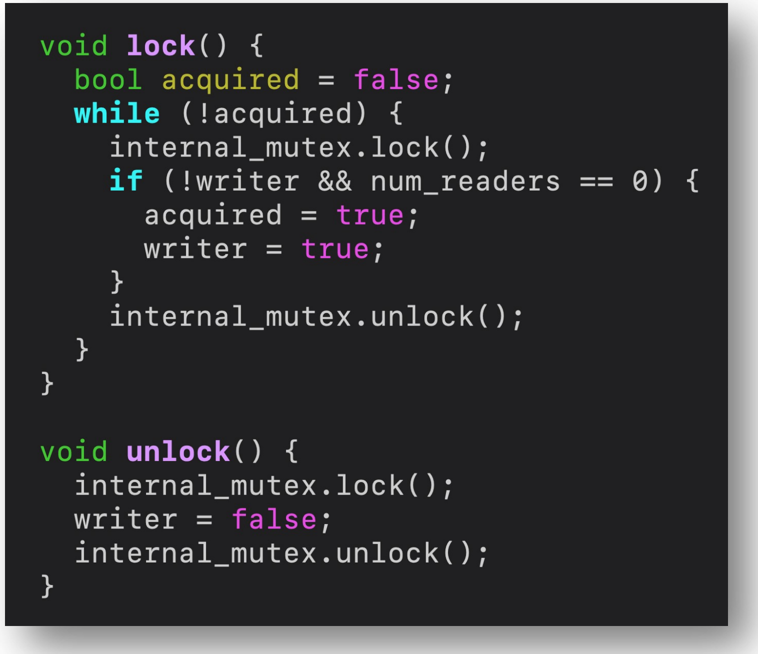
```
void get_paid() {
    m.lock();
    tylers_account++;
    m.unlock();
}
```

Thread 2

```
int check_balance() {
    m.reader_lock();
    int t = tylers_account;
    m.reader_unlock();
    return t;
}
```

Thread 3

```
int check_balance() {
    m.reader_lock();
    int t = tylers_account;
    m.reader_unlock();
    return t;
}
```



```
void lock() {
  bool acquired = false;
  while (!acquired) {
    internal_mutex.lock();
    if (!writer && num_readers == 0) {
      acquired = true;
      writer = true;
    }
    internal_mutex.unlock();
  }
}

void unlock() {
  internal_mutex.lock();
  writer = false;
  internal_mutex.unlock();
}
```

writer = False
num_readers = 2

Thread 0

```
void buy_coffee() {
    m.lock();
    tylers_account--;
    m.unlock();
}
```

Thread 1

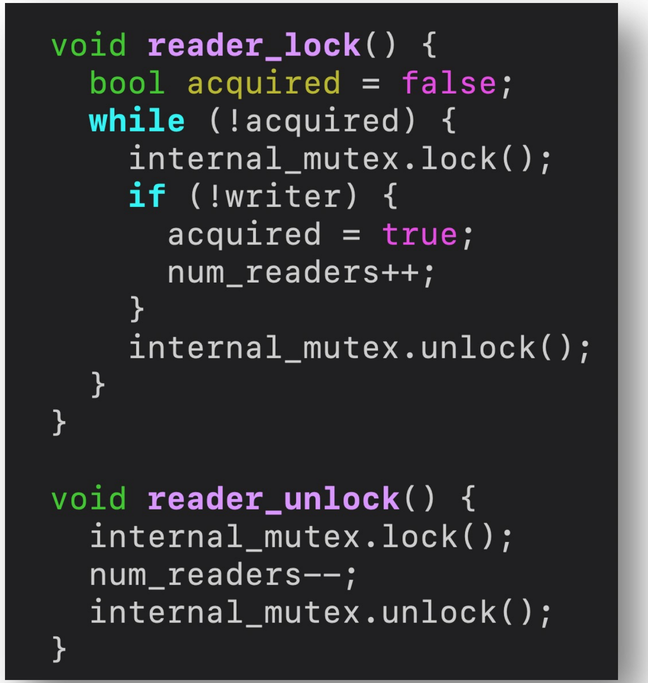
```
void get_paid() {
    m.lock();
    tylers_account++;
    m.unlock();
}
```

Thread 2

```
int check_balance() {
    m.reader_lock();
    int t = tylers_account;
    m.reader_unlock();
    return t;
}
```

Thread 3

```
int check_balance() {
    m.reader_lock();
    int t = tylers_account;
    m.reader_unlock();
    return t;
}
```



```
void reader_lock() {
  bool acquired = false;
  while (!acquired) {
    internal_mutex.lock();
    if (!writer) {
      acquired = true;
      num_readers++;
    }
    internal_mutex.unlock();
  }
}

void reader_unlock() {
  internal_mutex.lock();
  num_readers--;
  internal_mutex.unlock();
}
```

writer = False
num_readers = 2

Thread 0

```
void buy_coffee() {
    m.lock();
    tylers_account--;
    m.unlock();
}
```

Thread 1

```
void get_paid() {
    m.lock();
    tylers_account++;
    m.unlock();
}
```

Thread 2

```
int check_balance() {
    m.reader_lock();
    int t = tylers_account;
    m.reader_unlock();
    return t;
}
```

Thread 3

```
int check_balance() {
    m.reader_lock();
    int t = tylers_account;
    m.reader_unlock();
    return t;
}
```

writer = False
num_readers = 1

Thread 0

```
void buy_coffee() {
    m.lock();
    tylers_account--;
    m.unlock();
}
```

Thread 1

```
void get_paid() {
    m.lock();
    tylers_account++;
    m.unlock();
}
```

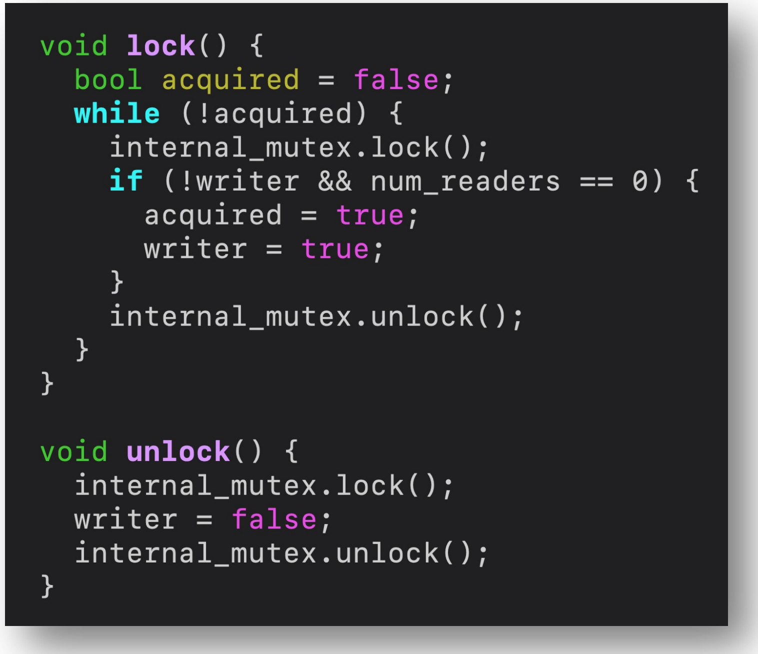
Thread 2

```
int check_balance() {
    m.reader_lock();
    int t = tylers_account;
    m.reader_unlock();
    return t;
}
```

Thread 3

```
int check_balance() {
    m.reader_lock();
    int t = tylers_account;
    m.reader_unlock();
    return t;
}
```

can we lock yet?



```
void lock() {
  bool acquired = false;
  while (!acquired) {
    internal_mutex.lock();
    if (!writer && num_readers == 0) {
      acquired = true;
      writer = true;
    }
    internal_mutex.unlock();
  }
}

void unlock() {
  internal_mutex.lock();
  writer = false;
  internal_mutex.unlock();
}
```

writer = False
num_readers = 1

Thread 0

```
void buy_coffee() {
    m.lock();
    tylers_account--;
    m.unlock();
}
```

Thread 1

```
void get_paid() {
    m.lock();
    tylers_account++;
    m.unlock();
}
```

Thread 2

```
int check_balance() {
    m.reader_lock();
    int t = tylers_account;
    m.reader_unlock();
    return t;
}
```

Thread 3

```
int check_balance() {
    m.reader_lock();
    int t = tylers_account;
    m.reader_unlock();
    return t;
}
```

writer = False
num_readers = 1

Thread 0

```
void buy_coffee() {
    m.lock();
    tylers_account--;
    m.unlock();
}
```

Thread 1

```
void get_paid() {
    m.lock();
    tylers_account++;
    m.unlock();
}
```

Thread 2

```
int check_balance() {
    m.reader_lock();
    int t = tylers_account;
    m.reader_unlock();
    return t;
}
```

Thread 3

```
int check_balance() {
    m.reader_lock();
    int t = tylers_account;
    m.reader_unlock();
    return t;
}
```

writer = False
num_readers = 0

Thread 0

```
void buy_coffee() {
    m.lock();
    tylers_account--;
    m.unlock();
}
```

Thread 1

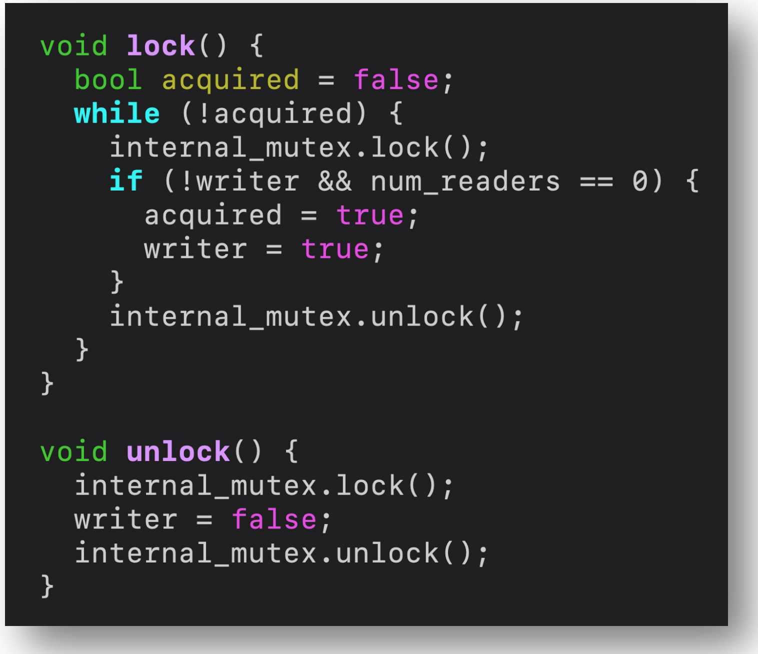
```
void get_paid() {
    m.lock();
    tylers_account++;
    m.unlock();
}
```

Thread 2

```
int check_balance() {
    m.reader_lock();
    int t = tylers_account;
    m.reader_unlock();
    return t;
}
```

Thread 3

```
int check_balance() {
    m.reader_lock();
    int t = tylers_account;
    m.reader_unlock();
    return t;
}
```



```
void lock() {
  bool acquired = false;
  while (!acquired) {
    internal_mutex.lock();
    if (!writer && num_readers == 0) {
      acquired = true;
      writer = true;
    }
    internal_mutex.unlock();
  }
}

void unlock() {
  internal_mutex.lock();
  writer = false;
  internal_mutex.unlock();
}
```

writer = False
num_readers = 0

Reader Writer lock

- This implementation potentially starves writers
  - The common case is to have lots of readers!

- Think about ways how an implementation might be more fair to writers.

How this looks in C++

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

shared_mutex m;

m.lock_shared()   // reader lock
m.unlock_shared() // reader unlock
m.lock()          // regular lock
m.unlock()        // regular unlock
```

Next week

- Planning on last mutex lecture
  - More specialized examples
  - Optimistic vs. pessimistic concurrency
- Work on HW 2! You now have everything you need to complete it!
  - Parts of next lectures might help with part 2.