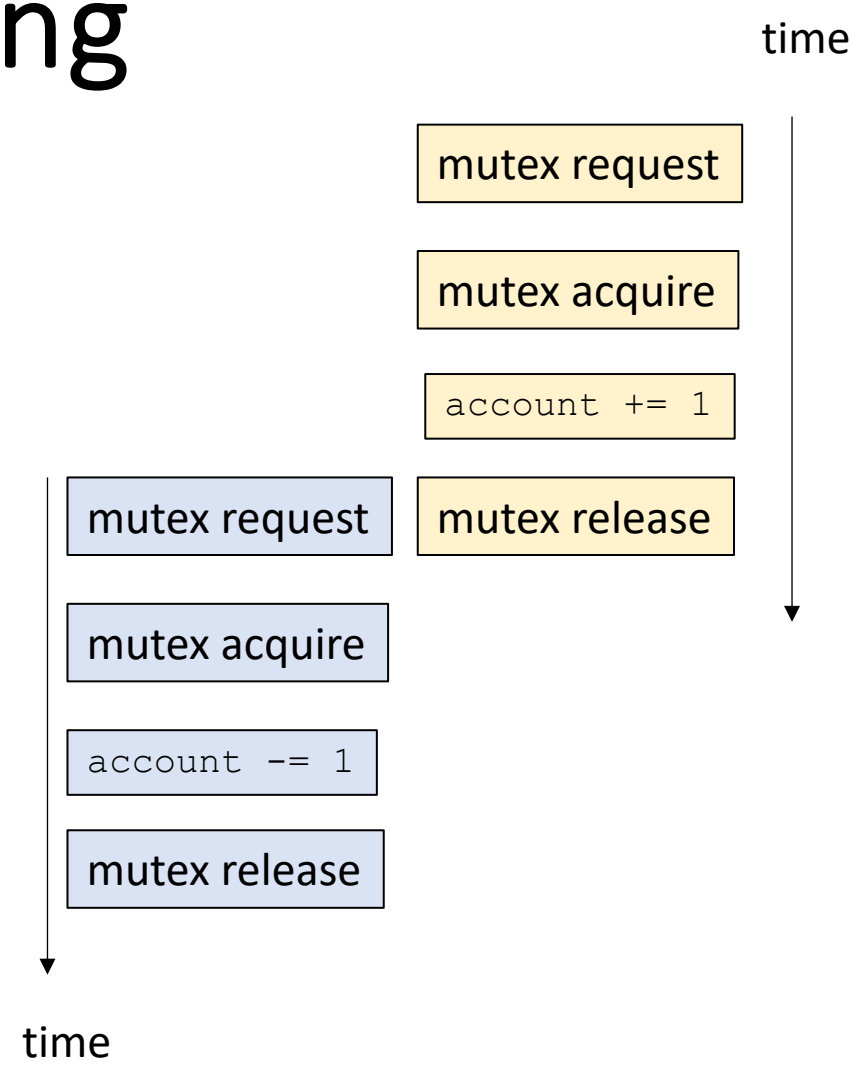
CSE113: Parallel Programming

Feb. 5, 2024

- **Topics**:
  - RMW optimizations
  - Specialized RMWs



Announcements

- Last planned lecture in Module 2
  - Next moving on to Module 3: concurrent data structures

- Working on HW 1 grades now
  - Giving a 0 if we have issues with your repo
  - Please contact us to resolve the issue!

Announcements

- HW 2 was released, with an announcement on the parts you could start

- Will have everything you need today to finish the homework
  - Extended the deadline: it is due in 1 week + 3 late days
  - We will release HW 3 next wednesday

Announcements

- Midterm is in 1 week (Feb 12)
  - In-person test
  - 3 pages of notes front and back (but no memorization questions)
  - 10% of your grade
  - 5 or 6 short answer questions (e.g., how to reverse a linked list but with parallel programming questions)

Previous quiz + review

Previous quiz + review

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

Mutex Implementations

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

  void lock();
  void unlock();


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

Initially:
No victim and no threads are interested in the critical section

flags and victim

Mutex Implementations

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

j is the other thread

Mark ourself as interested

volunteer to be the victim in case of a tie

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

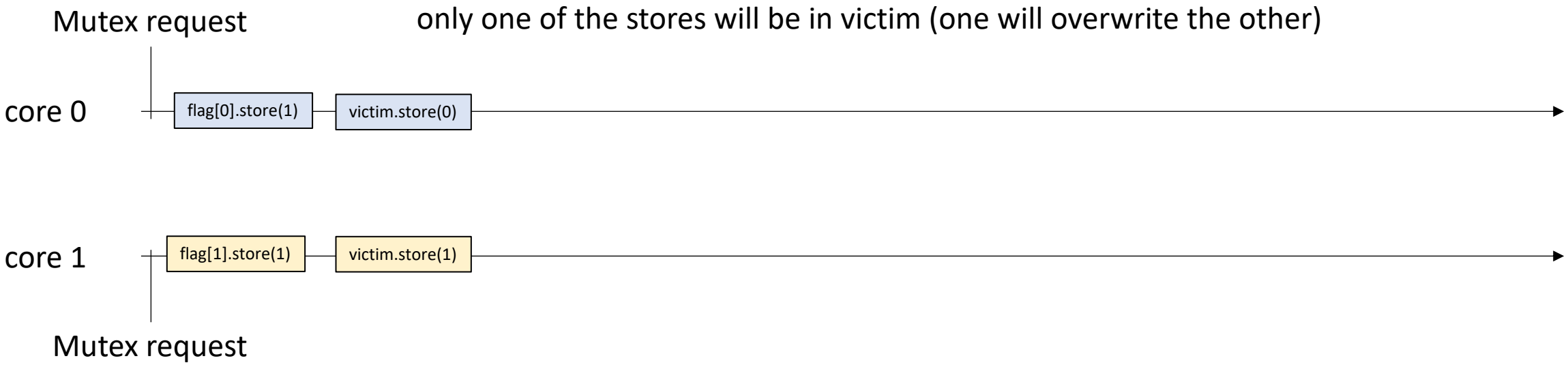
Tie breaking with victim

```
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 quiz + review

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

atomic_fetch_add

Recall the lock free account

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

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

Previous quiz + review

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

Exchange

```
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;
}
```

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

Previous quiz + review

CAS and Exchange locks are not starvation free, but starvation is so rare that it does not matter in practice

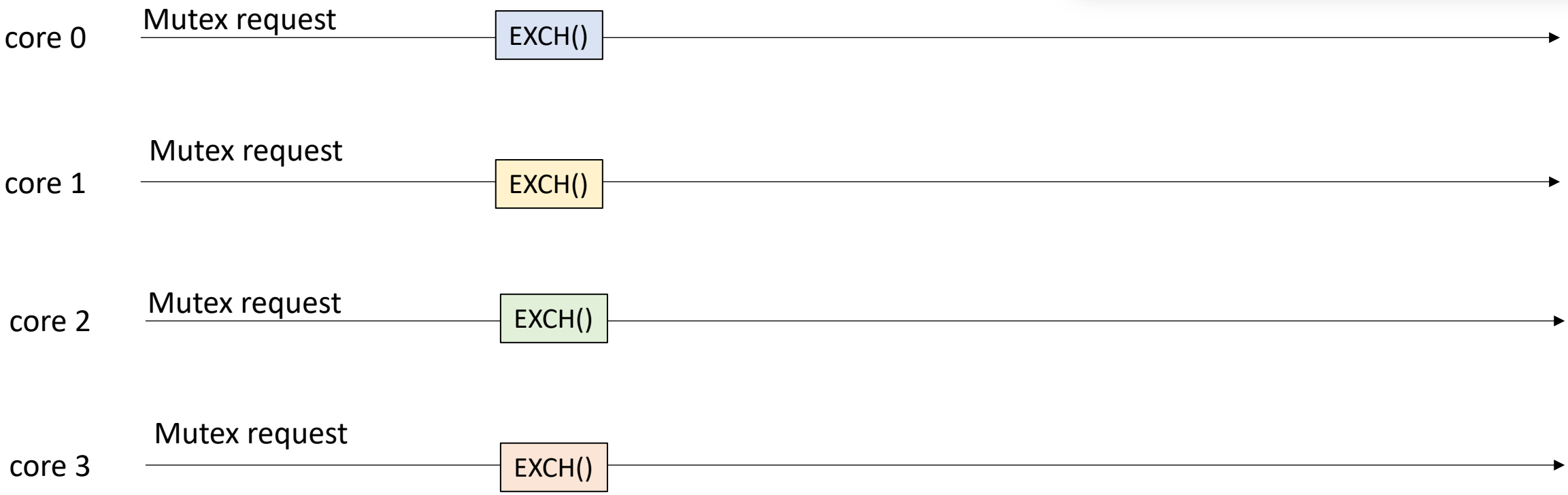
- ◯ True
- ◯ False

Analysis

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

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

what about 4 threads?



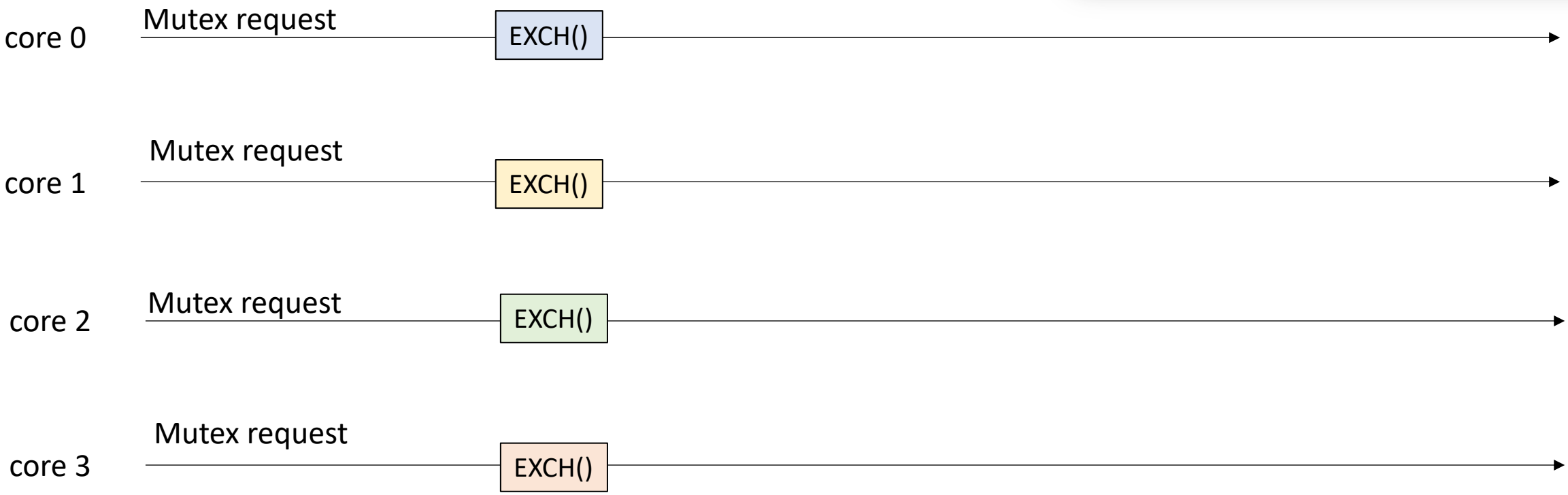
Analysis

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

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

what about 4 threads?

atomic operations can't overlap



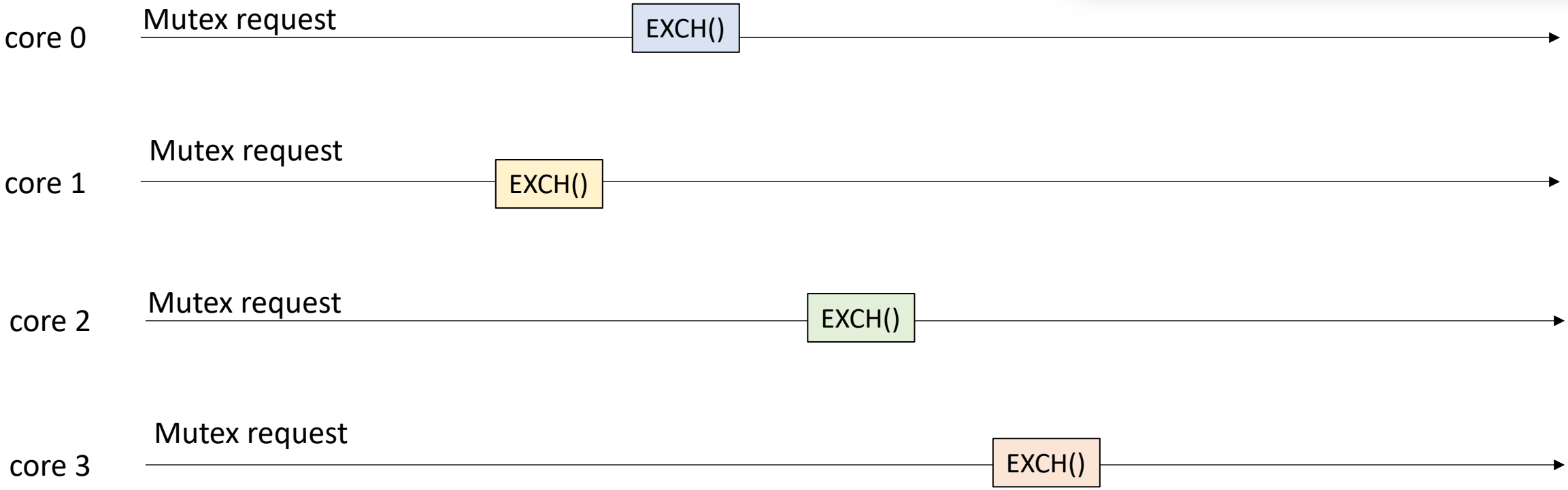
Analysis

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

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

what about 4 threads?

atomic operations can’t overlap



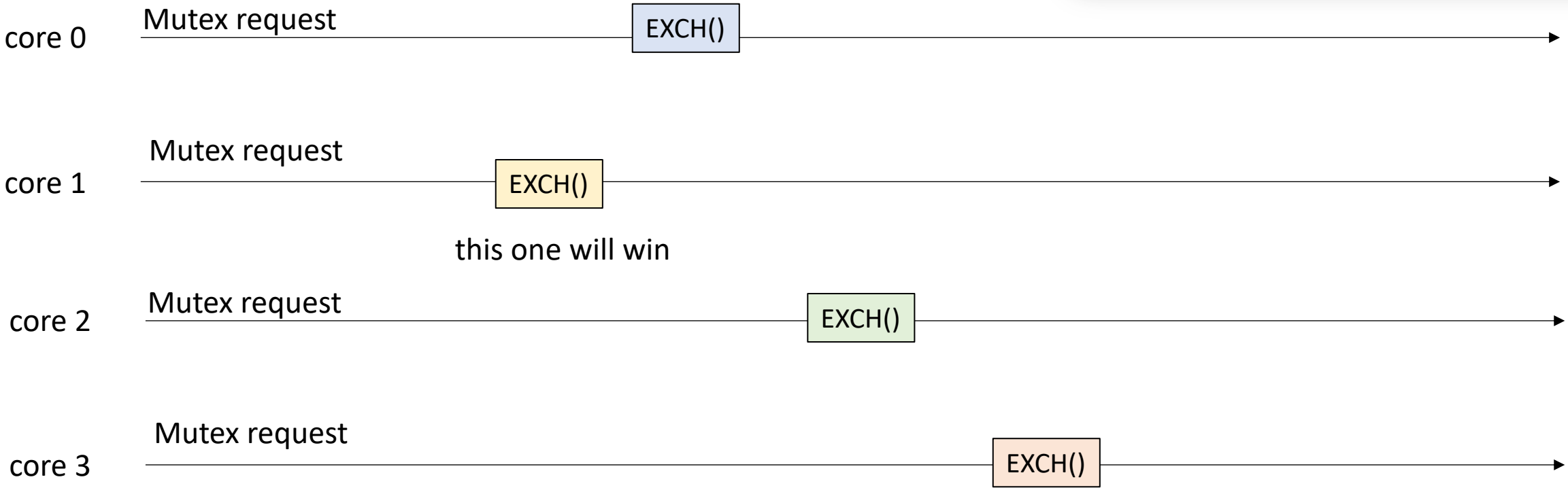
Analysis

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

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

what about 4 threads?

atomic operations can't overlap



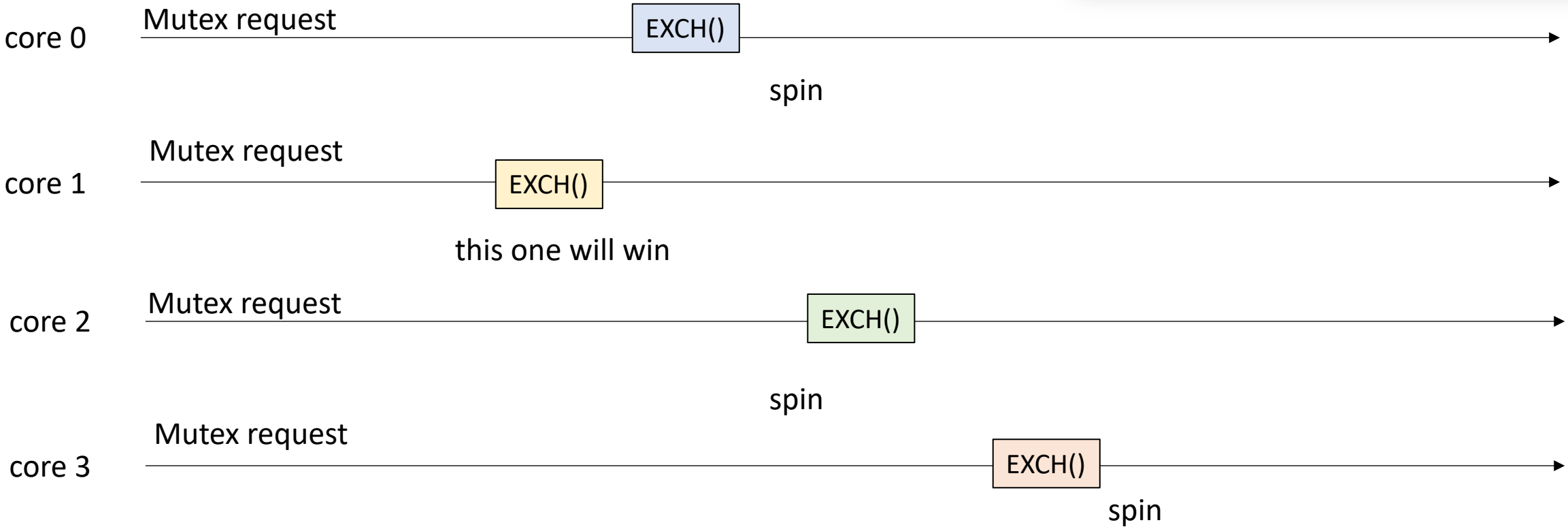
Analysis

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

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

what about 4 threads?

atomic operations can't overlap



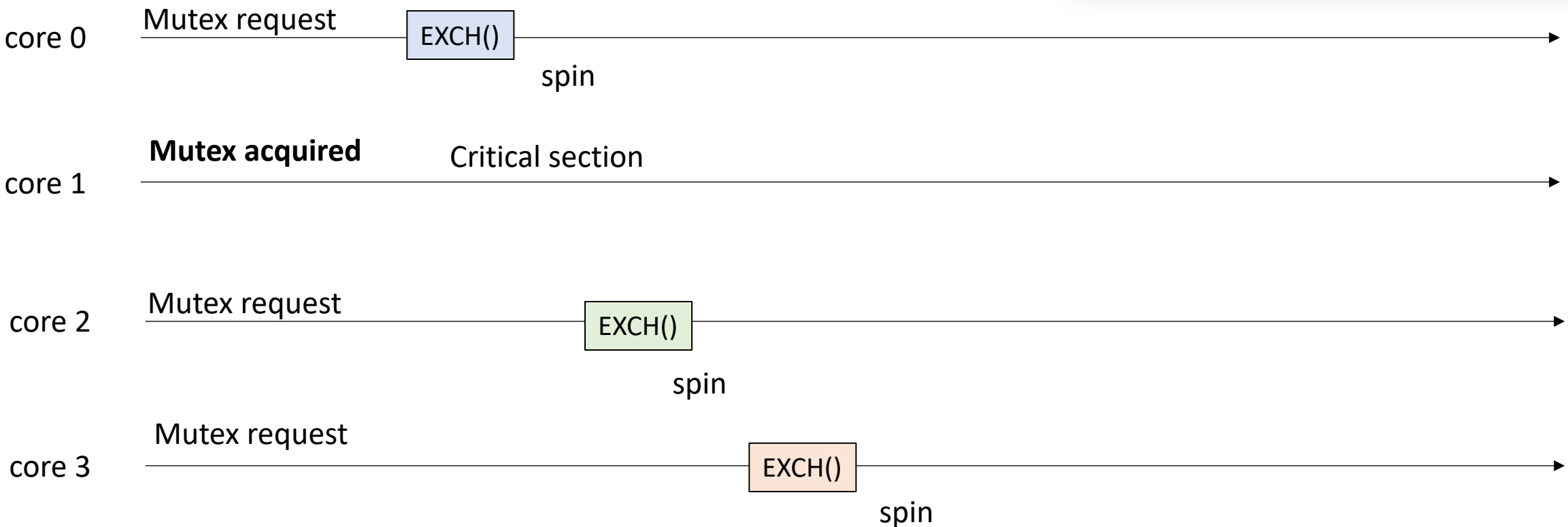
Analysis

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

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

what about 4 threads?

atomic operations can't overlap



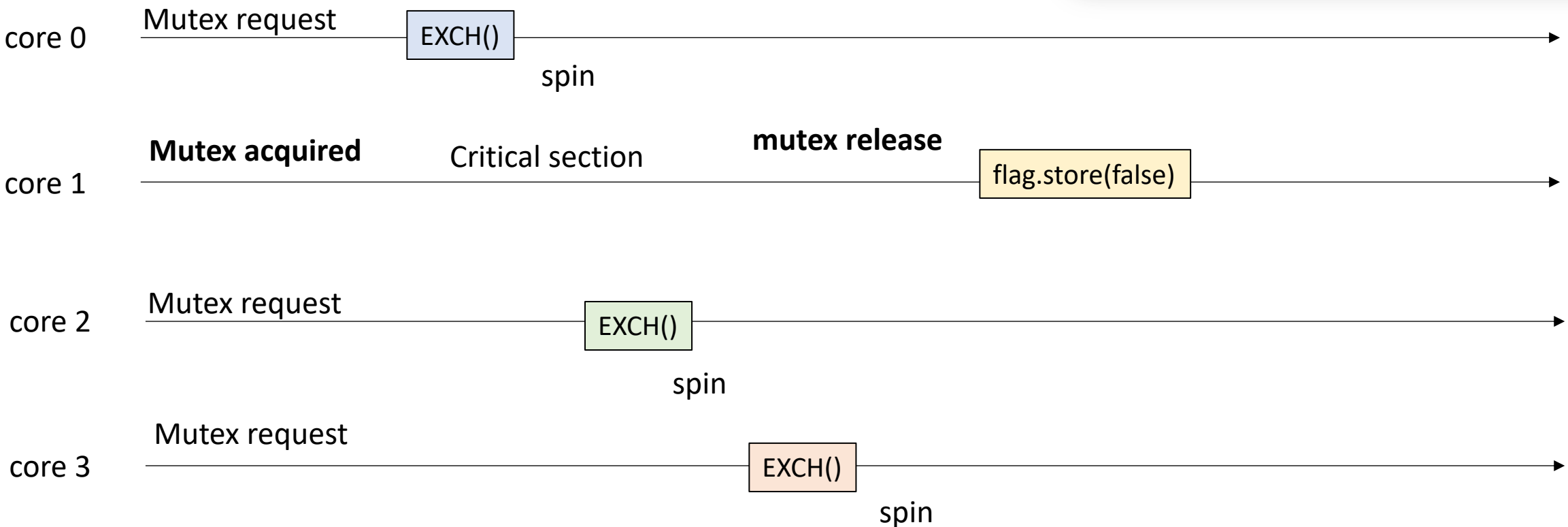
Analysis

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

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

what about 4 threads?

atomic operations can't overlap



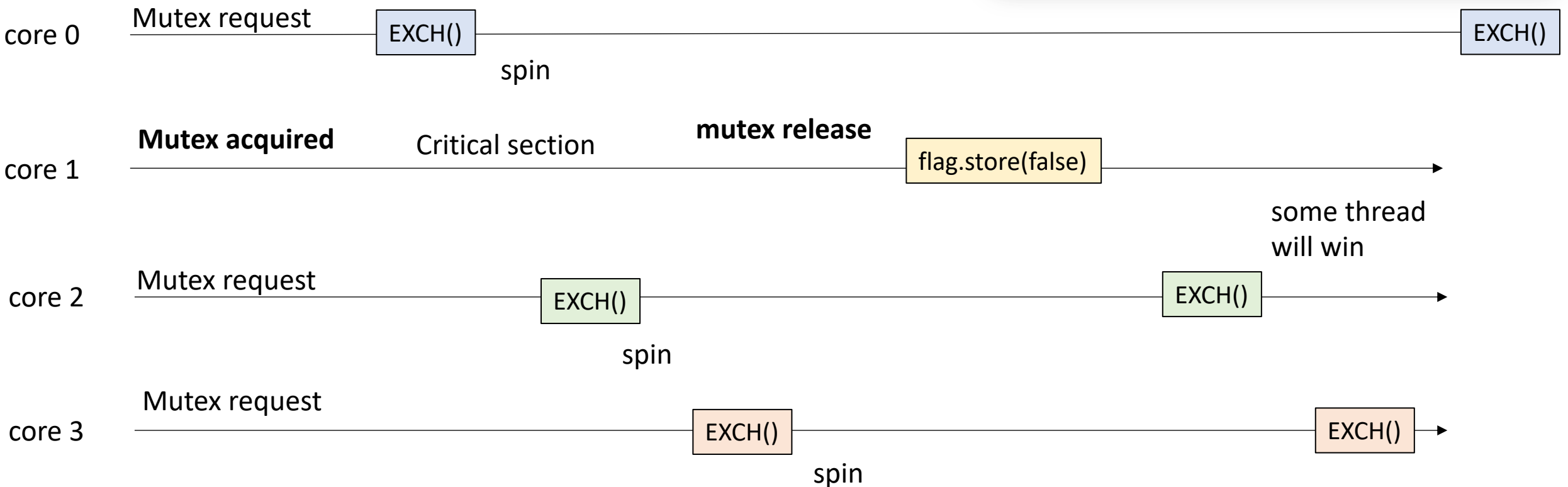
Analysis

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

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

what about 4 threads?

atomic operations can't overlap



Previous quiz + review

Which of the following locks have required a RMW atomic for unlocking?

- ◯ CAS lock
- ◯ Exchange lock
- ◯ Ticket lock
- ◯ all of the above
- ◯ none of the above

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

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.

Previous quiz + review

discuss some of the trade-offs between a fair mutex and unfair mutex

Previous quiz + review

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

New material

Optimizing mutexes

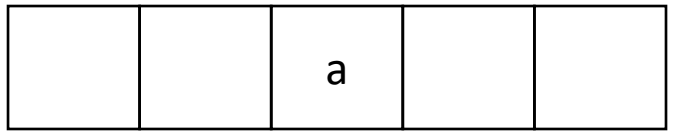
RMW implementations

Pessimistic Concurrency

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

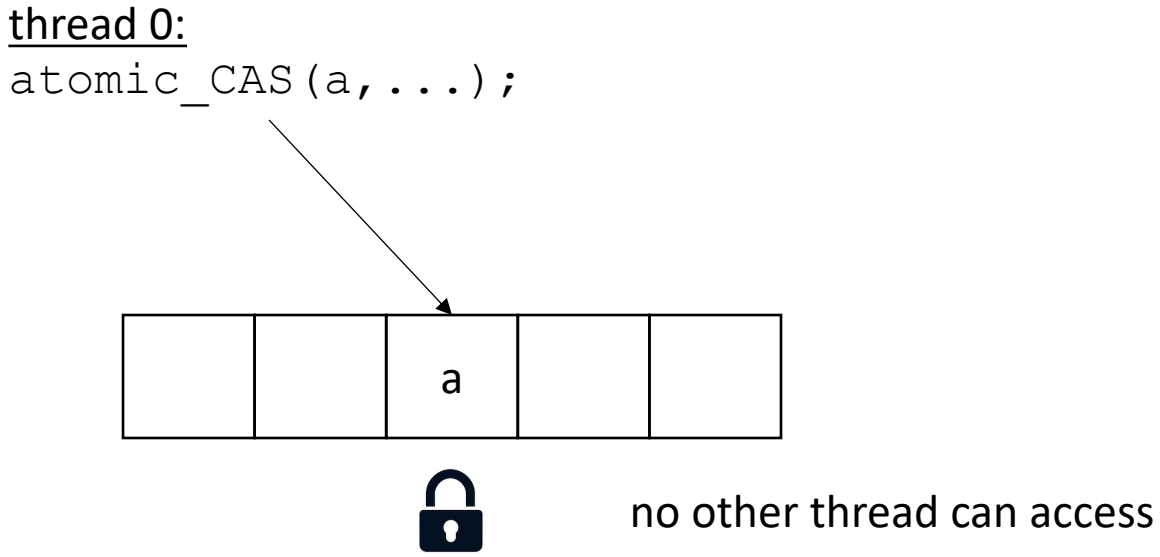
thread 0:

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



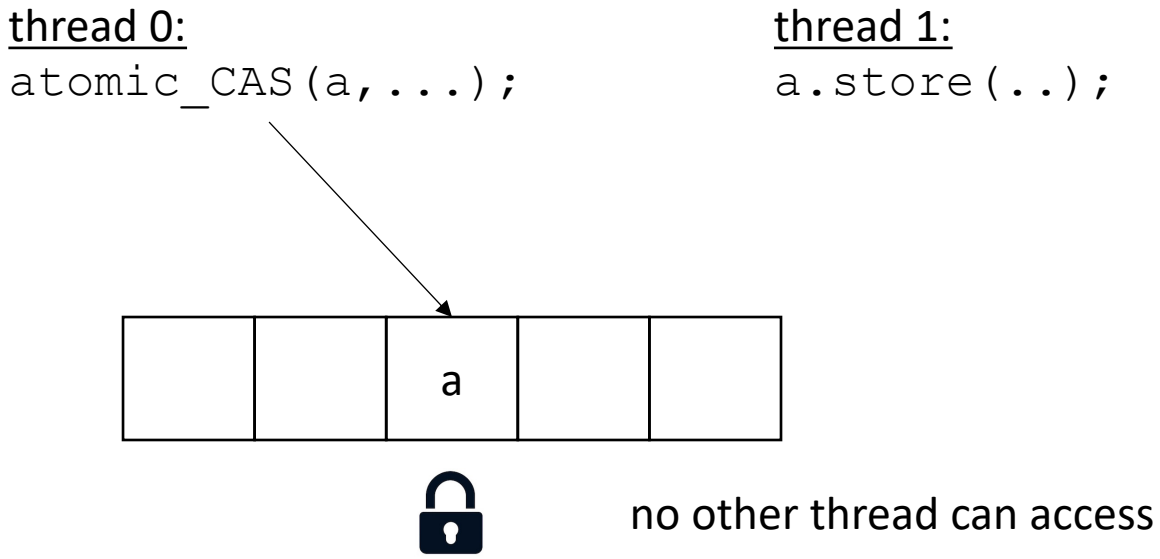
Pessimistic Concurrency

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



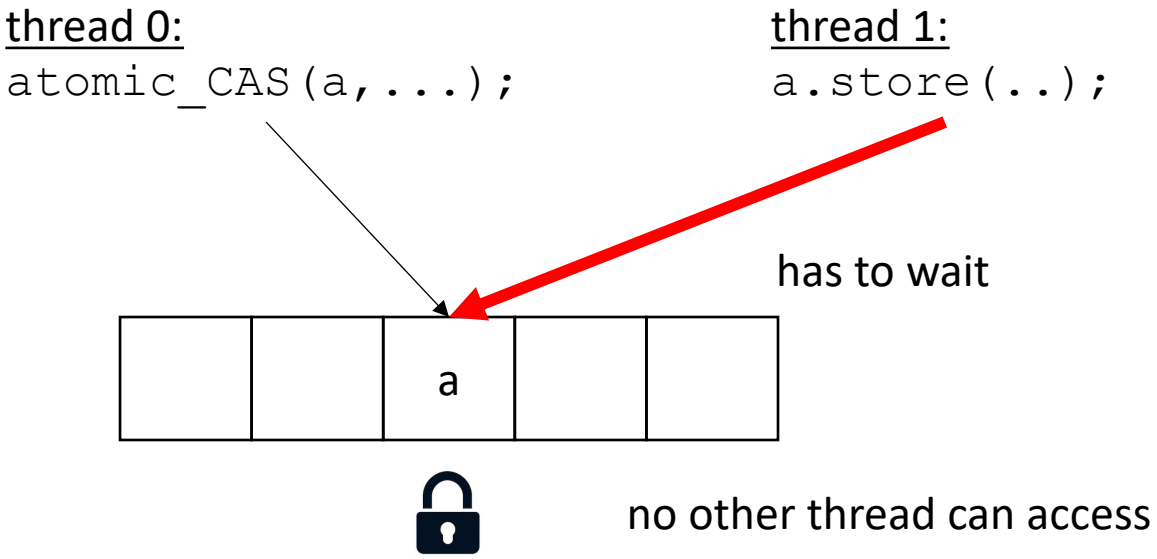
Pessimistic Concurrency

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



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

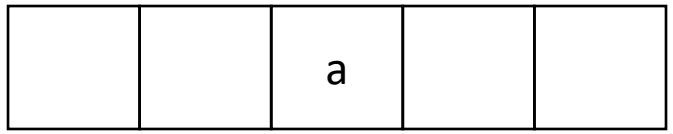


Optimistic Concurrency

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

thread 0:

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



T0_exclusive = 0

RMWs are expensive!

- Can we reduce the number of RMWs that we do?

Optimizations: relaxed peeking

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

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

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

Optimizations: relaxed peeking

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

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

Optimizations: relaxed peeking

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

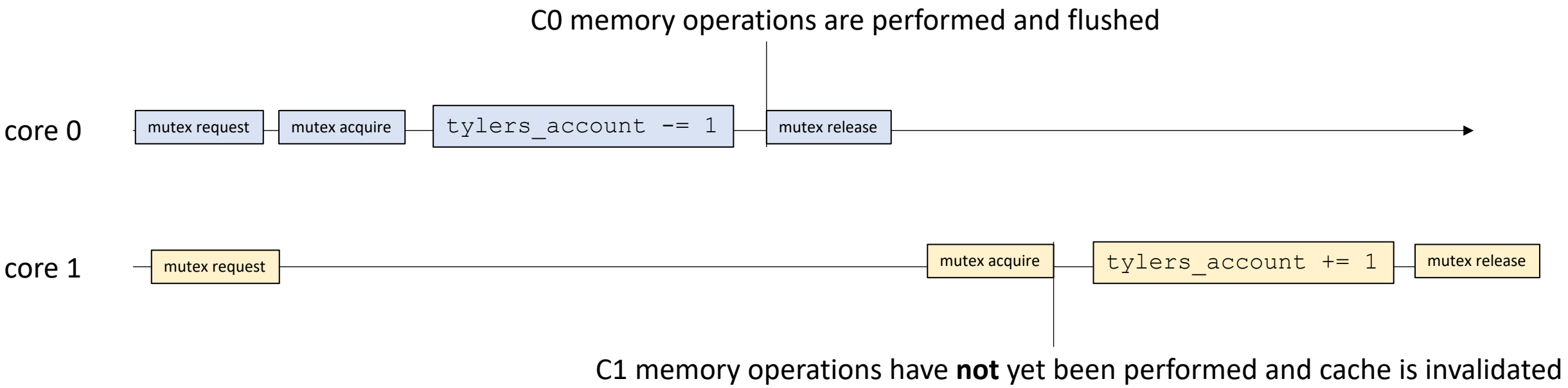
```
void lock() {
  bool e = false;
  bool acquired = false;
  while (!acquired) {
    while (flag.load() == true);
    e = false;
    acquired = atomic_compare_exchange_strong(&flag, &e, true);
  }
}
```

Optimizations: relaxed peeking

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

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

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



Relaxed atomics

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

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

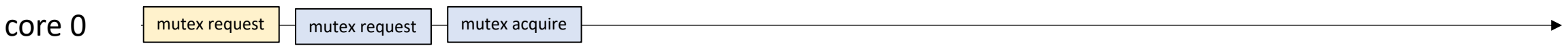
- Even using relaxed peeking, two issues remain:
  - Loads still cause bus traffic (even if its not as bad as RMWs)
  - **In non-parallel systems, concurrent threads can get in the way of progress**

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**

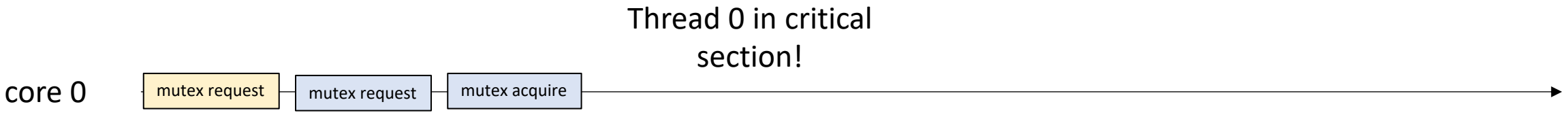
Say threads 0 and 1 are executing concurrently



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**

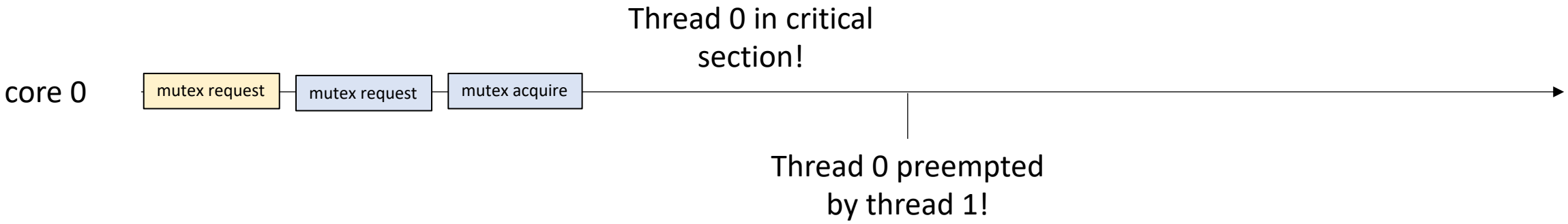
Say threads 0 and 1 are executing concurrently



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**

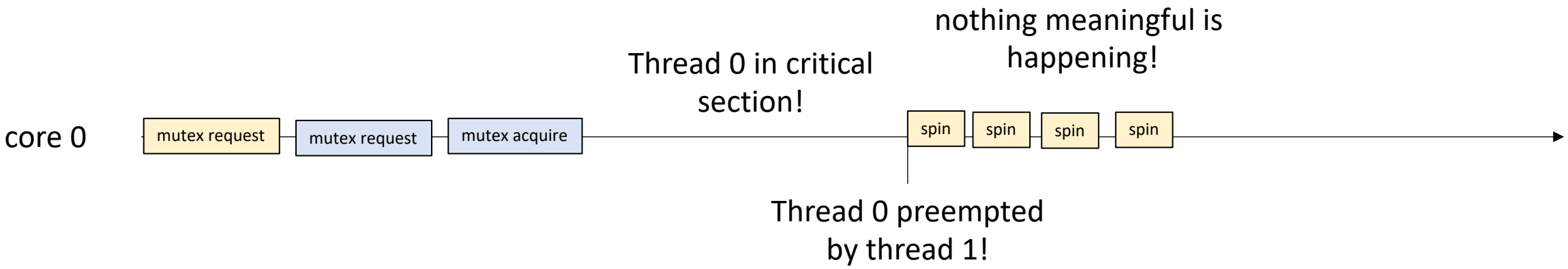
Say threads 0 and 1 are executing concurrently



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**

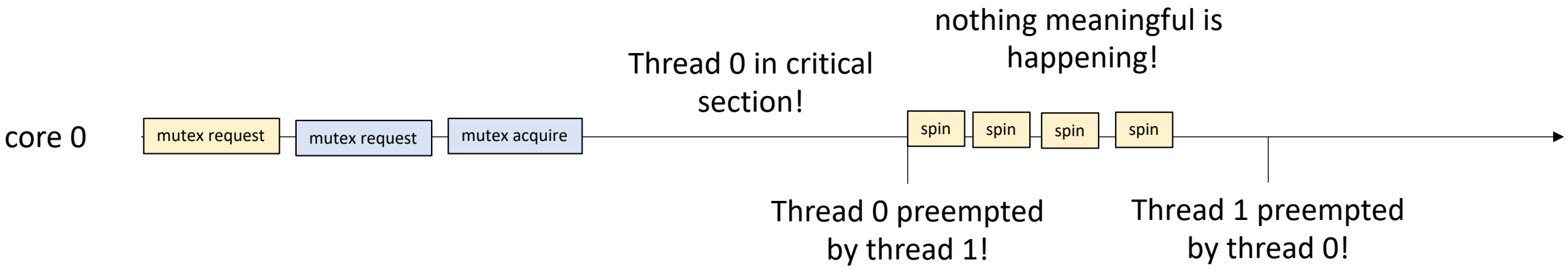
Say threads 0 and 1 are executing concurrently



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**

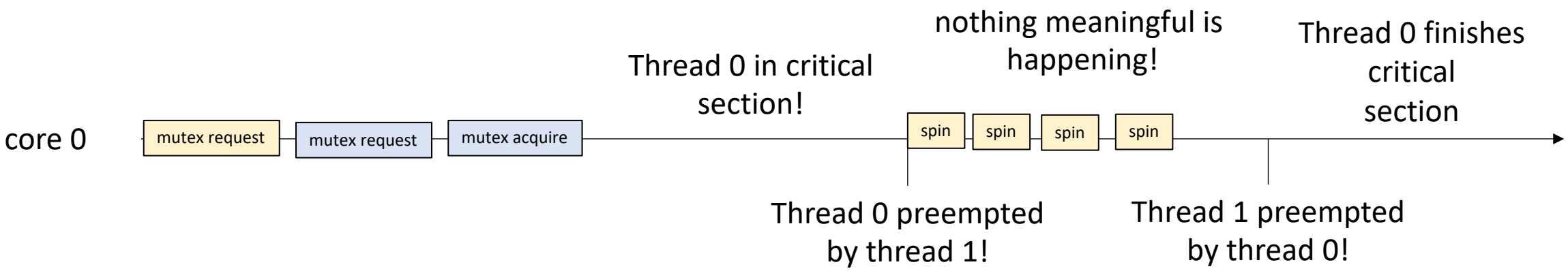
Say threads 0 and 1 are executing concurrently



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**

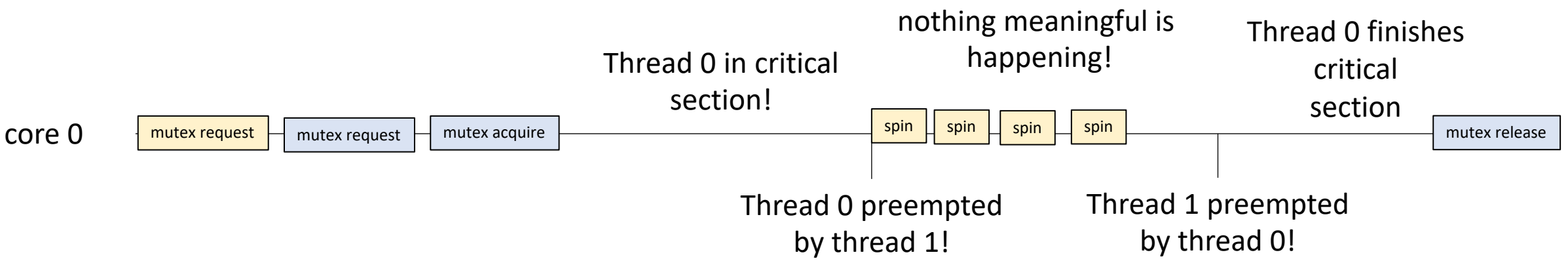
Say threads 0 and 1 are executing concurrently



Optimizations: backoff

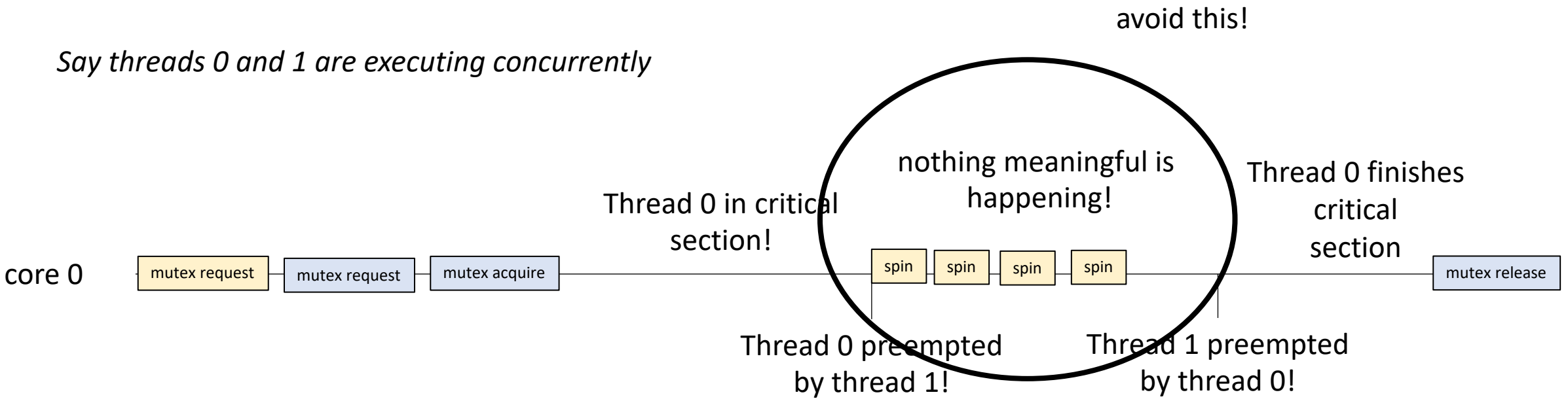
- Two issues remain:
  - Loads still cause bus traffic (even if its not as bad as RMWs)
  - **In non-parallel systems, concurrent threads can get in the way of progress**

Say threads 0 and 1 are executing concurrently



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**



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

Reader-Writer Mutex

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`

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

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

But what happens more frequently than either of those things?

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
```

Note: lock_guard in C++

- *Resource Acquisition Is Initialization* or RAII
- Defined in header <mutex>
- owns a mutex for the duration of a scoped block.

https://en.cppreference.com/w/cpp/thread/lock_guard

Note: lock_guard in C++

```
volatile int g_i = 0;
std::mutex g_i_mutex;

void safe_increment(int iterations) {
  const std::lock_guard< std::mutex > lock(g_i_mutex);
  while (iterations-- > 0) {
    g_i = g_i + 1;
  }
}
```

https://en.cppreference.com/w/cpp/thread/lock_guard

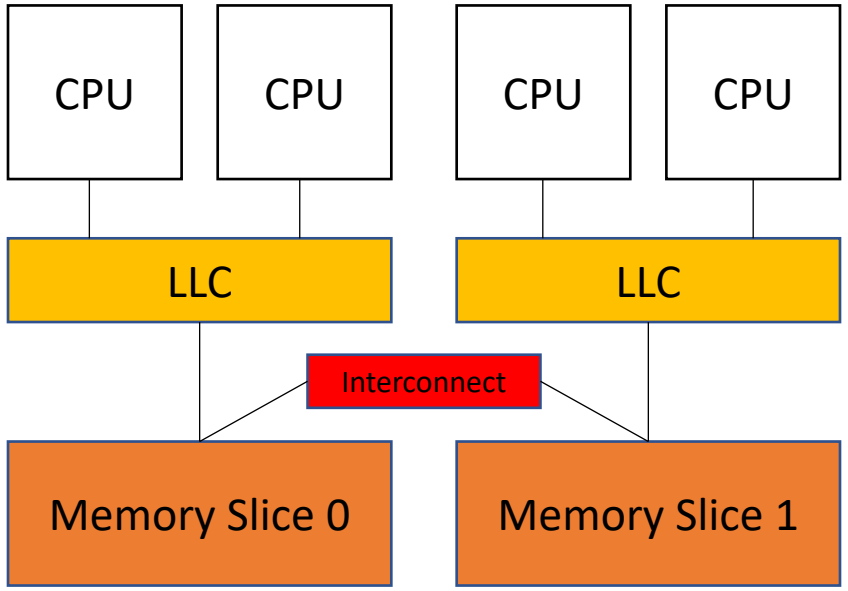
Optimization: Hierarchical locks

Optimization: Hierarchical locks

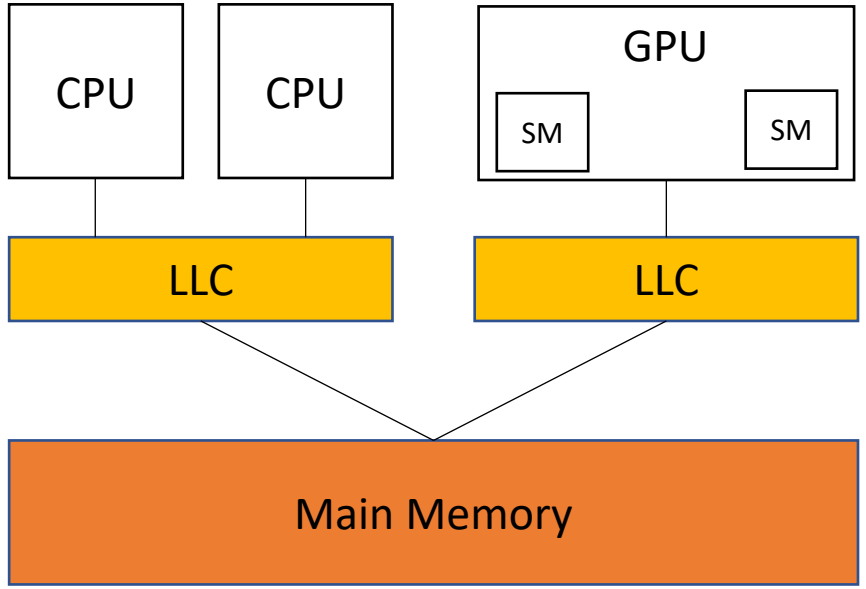
- NUMA (non-uniform memory access) systems
- heterogeneous systems (CPU GPU)

Discrete GPUs communicate through PCIE

For example: Large server nodes

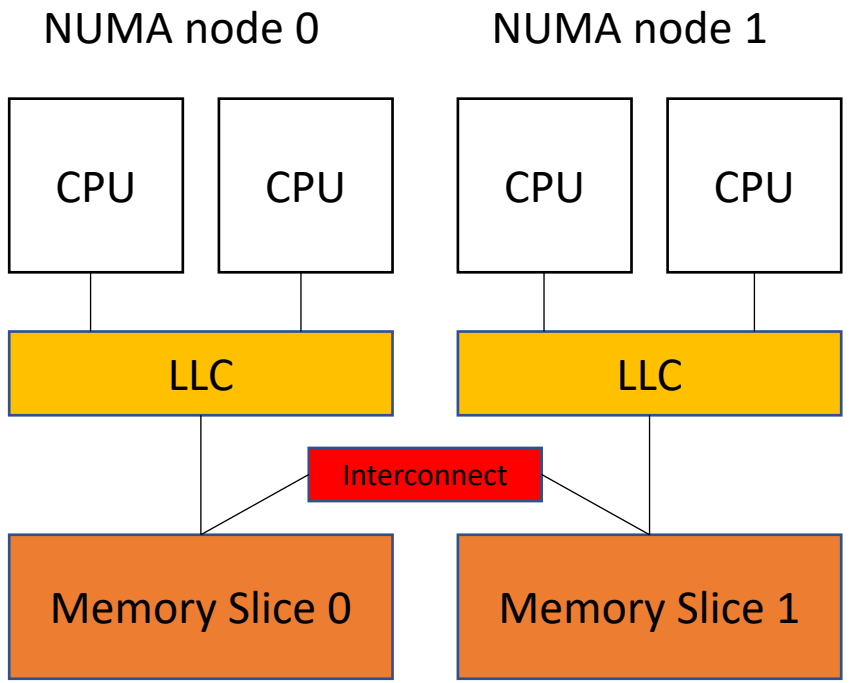


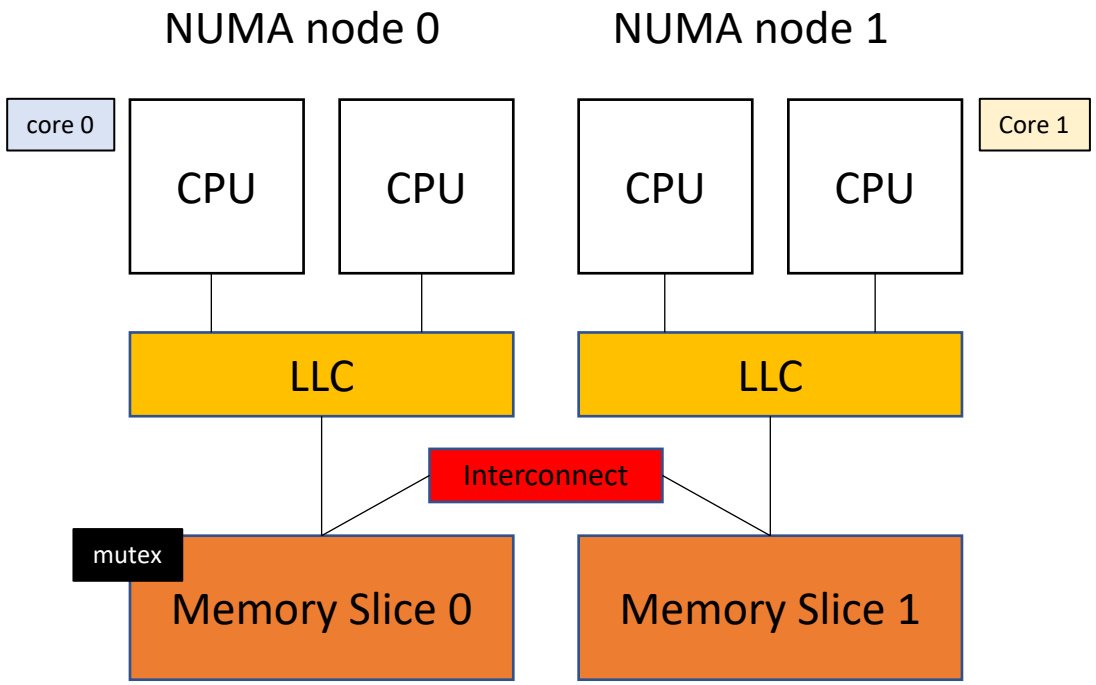
For example: SoCs like Iphone

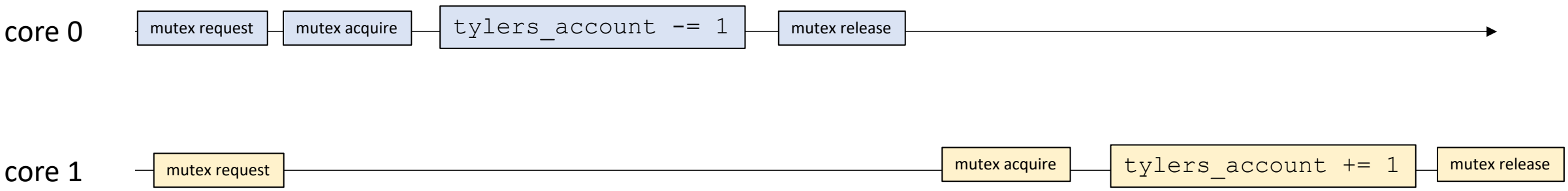


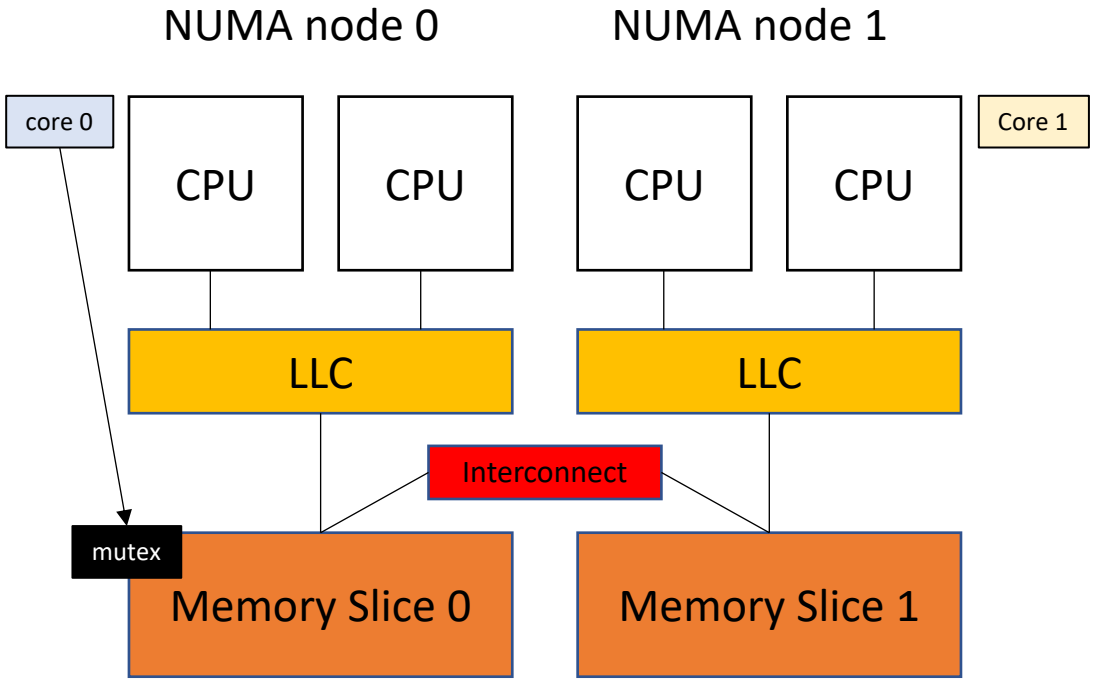
Optimization: Hierarchical locks

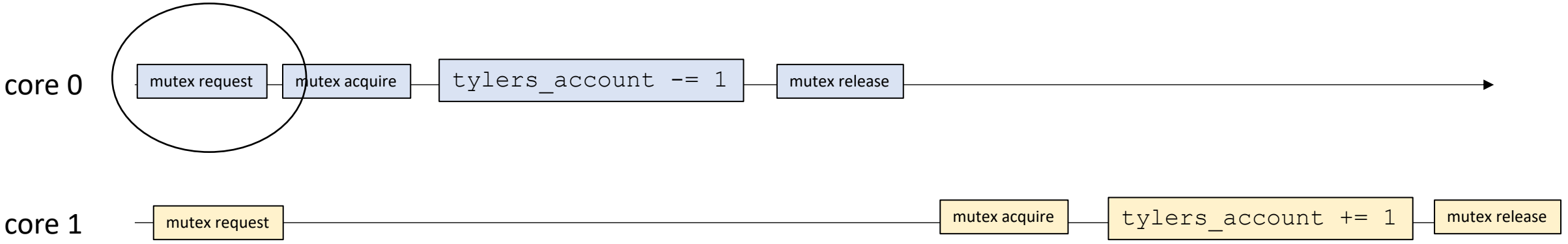
- Any sort of communication is very expensive:
  - Spinning triggers expensive coherence protocols.
  - cache flushes between NUMA nodes is expensive (transferring memory between critical sections)

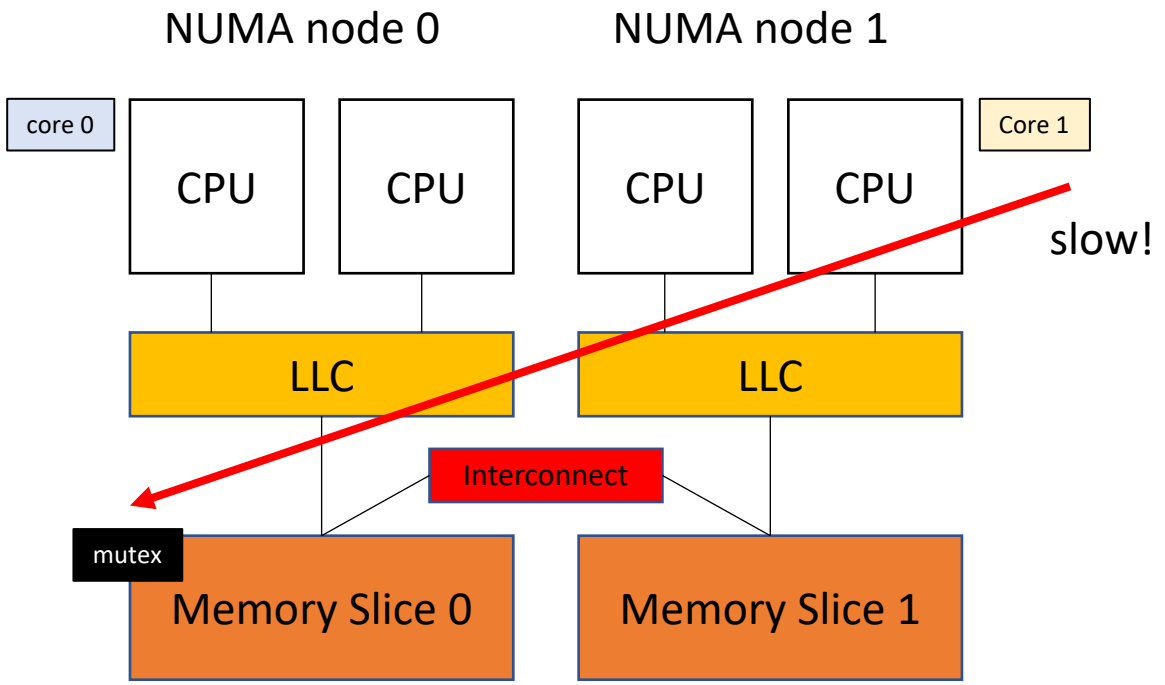


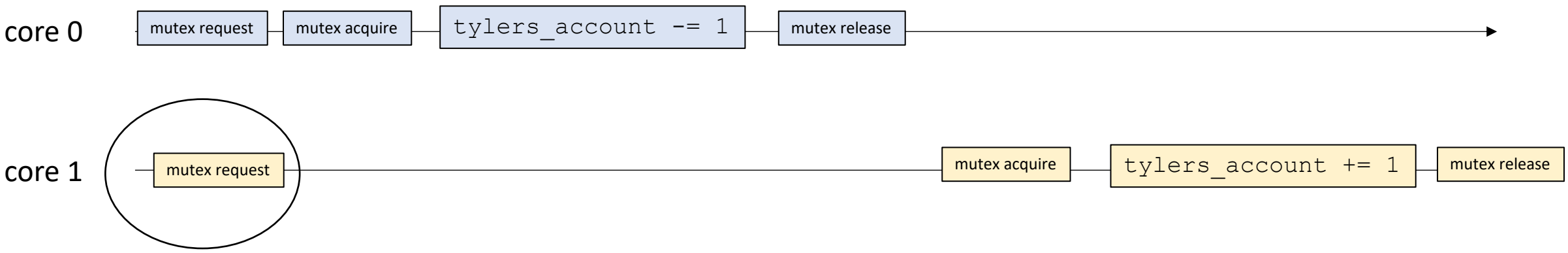

NUMA node 0
NUMA node 1
core 0
CPU
CPU
CPU
CPU
Core 1
LLC
LLC
Interconnect
mutex
Memory Slice 0
Memory Slice 1

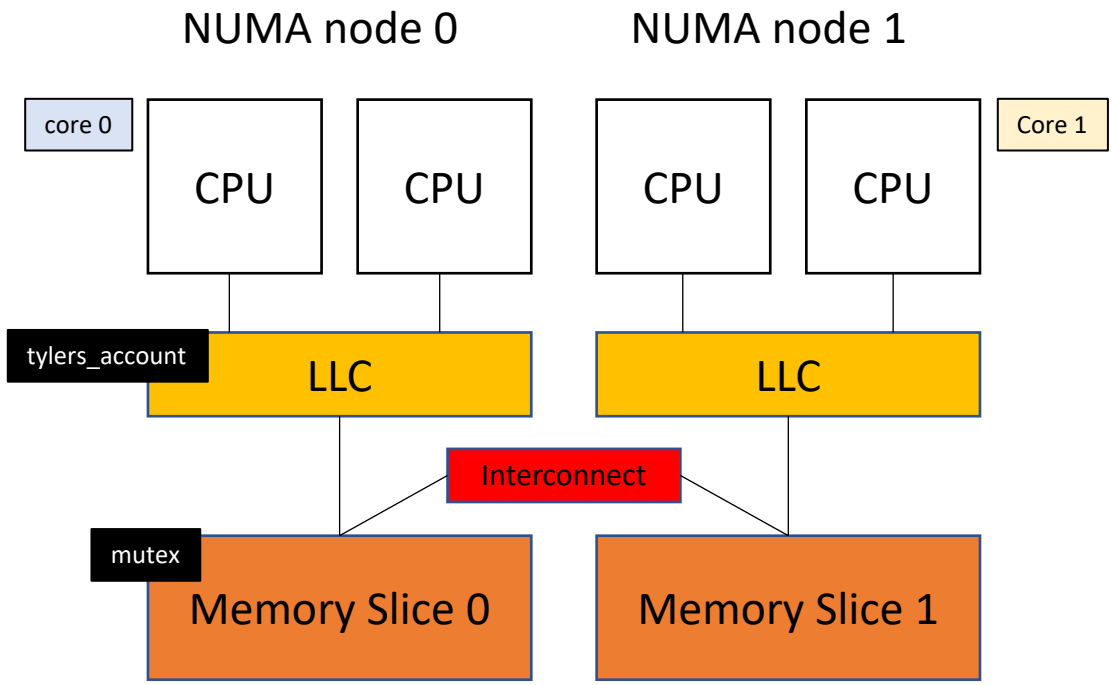

core 0
mutex request
mutex acquire
tylers_account -= 1
mutex release
core 1
mutex request
mutex acquire
tylers_account += 1
mutex release

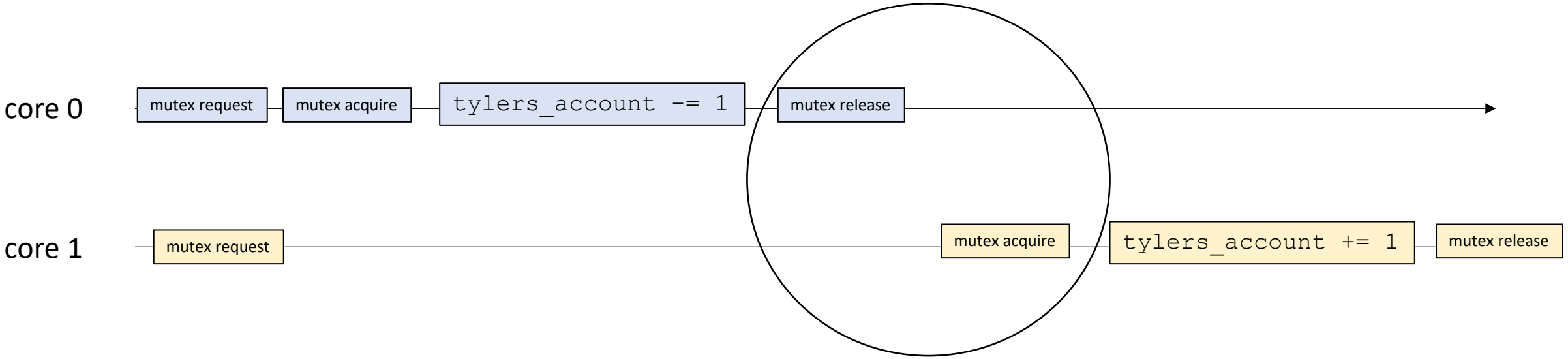

NUMA node 0
NUMA node 1
core 0
CPU
CPU
CPU
CPU
Core 1
LLC
LLC
Interconnect
mutex
Memory Slice 0
Memory Slice 1

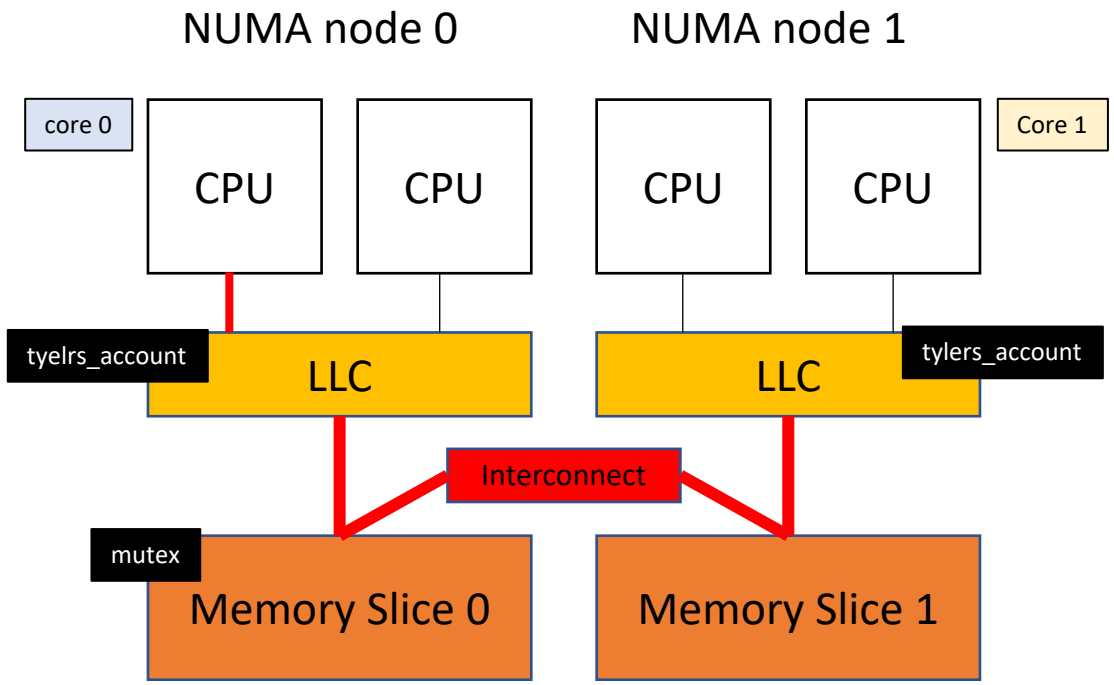

core 0
mutex request
mutex acquire
tylers_account -= 1
mutex release
core 1
mutex request
mutex acquire
tylers_account += 1
mutex release

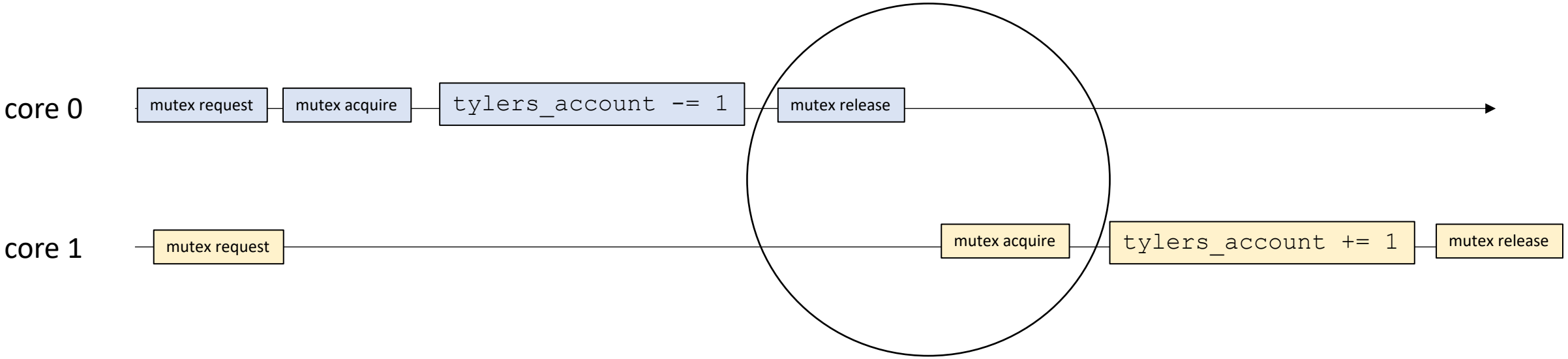

NUMA node 0
NUMA node 1
core 0
CPU
CPU
CPU
CPU
Core 1
slow!
LLC
LLC
Interconnect
mutex
Memory Slice 0
Memory Slice 1

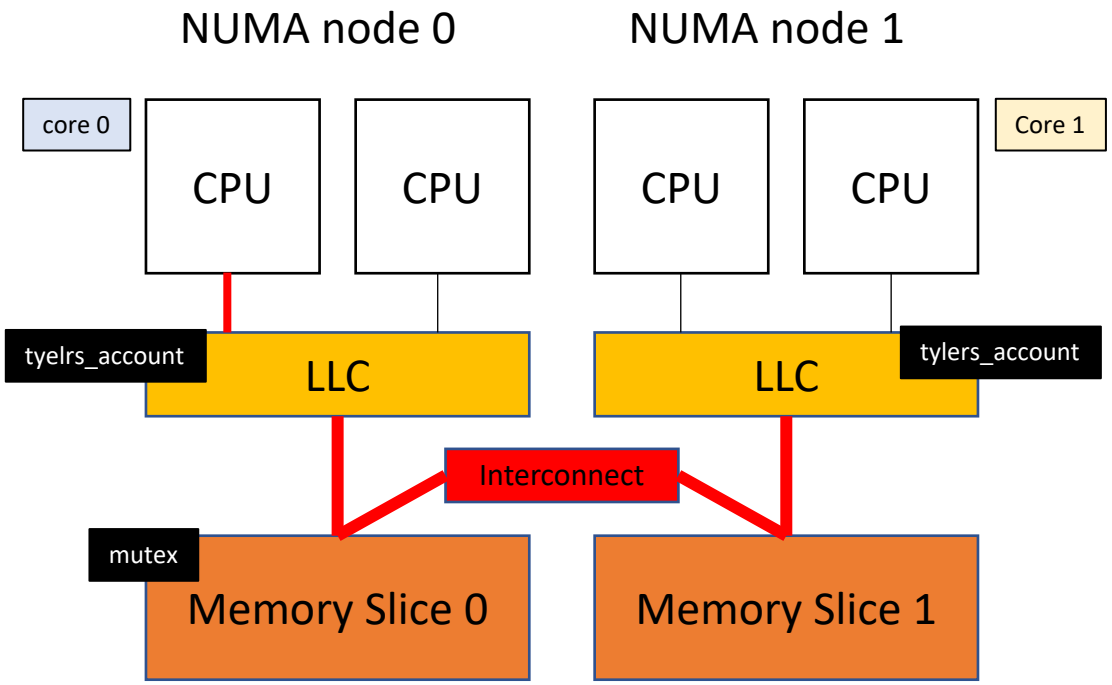

core 0
mutex request
mutex acquire
tylers_account -= 1
mutex release
core 1
mutex request
mutex acquire
tylers_account += 1
mutex release


NUMA node 0
NUMA node 1
core 0
CPU
CPU
CPU
CPU
Core 1
tylers_account
LLC
LLC
Interconnect
mutex
Memory Slice 0
Memory Slice 1

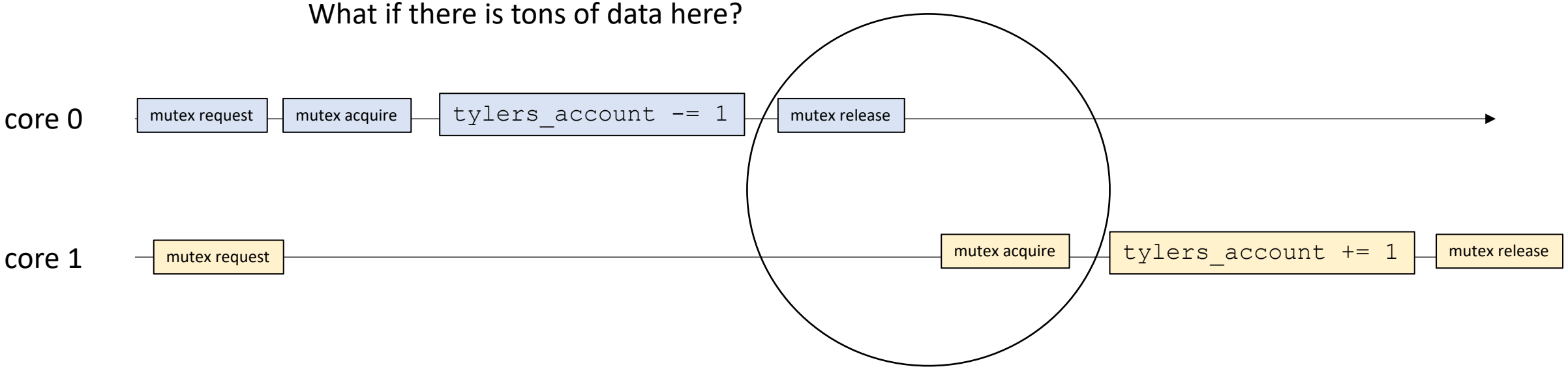

core 0
mutex request
mutex acquire
tylers_account -= 1
mutex release
core 1
mutex request
mutex acquire
tylers_account += 1
mutex release

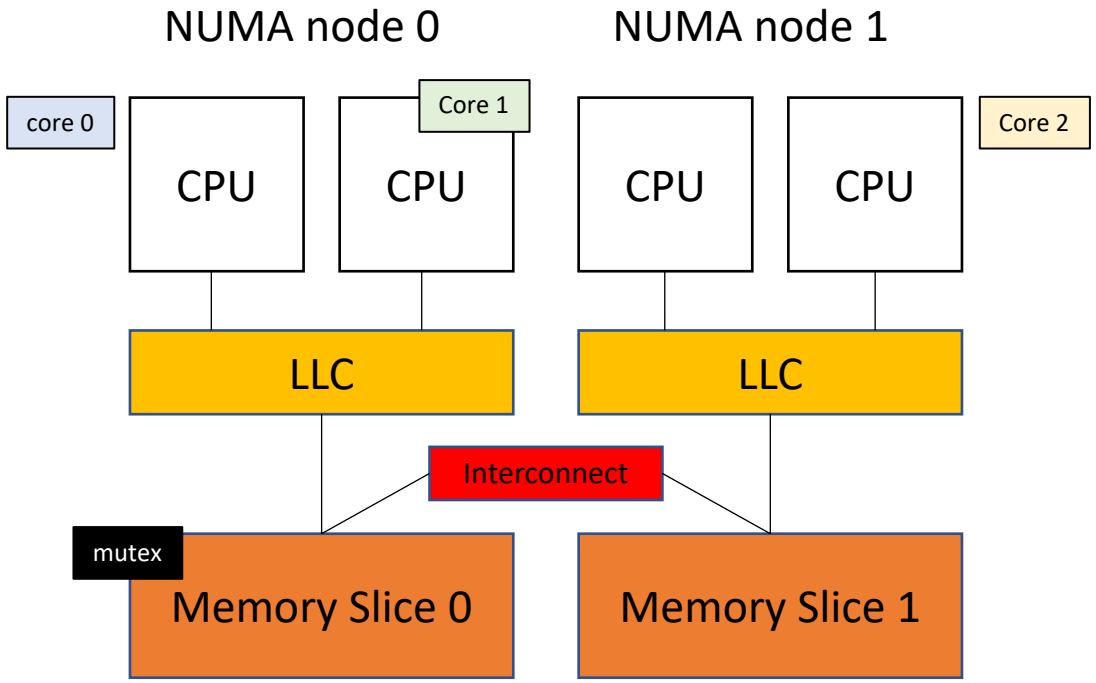

NUMA node 0
NUMA node 1
core 0
CPU
CPU
CPU
CPU
Core 1
tyelrs_account
LLC
LLC
tylers_account
Interconnect
mutex
Memory Slice 0
Memory Slice 1

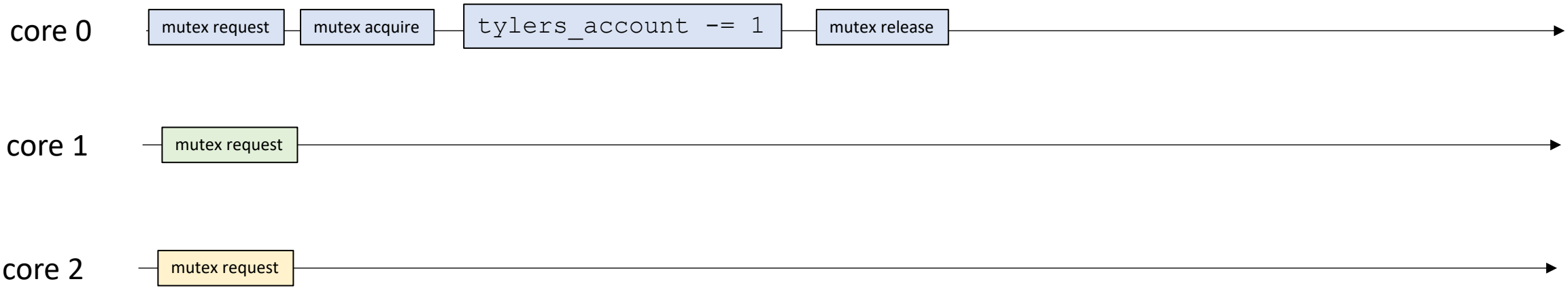

core 0
mutex request
mutex acquire
tylers_account -= 1
mutex release
core 1
mutex request
mutex acquire
tylers_account += 1
mutex release

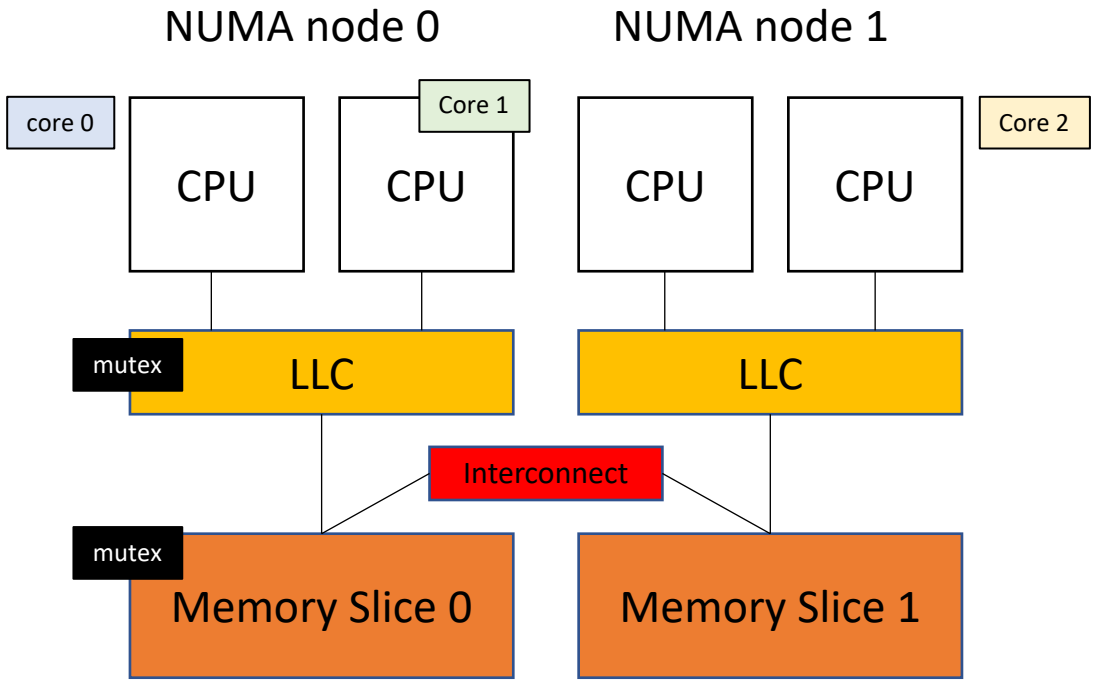


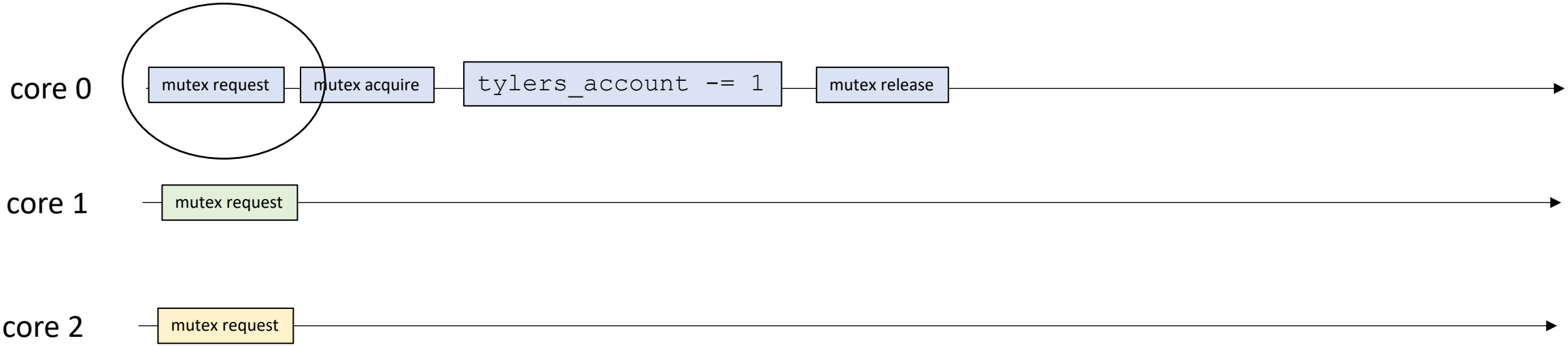
What if there is tons of data here?

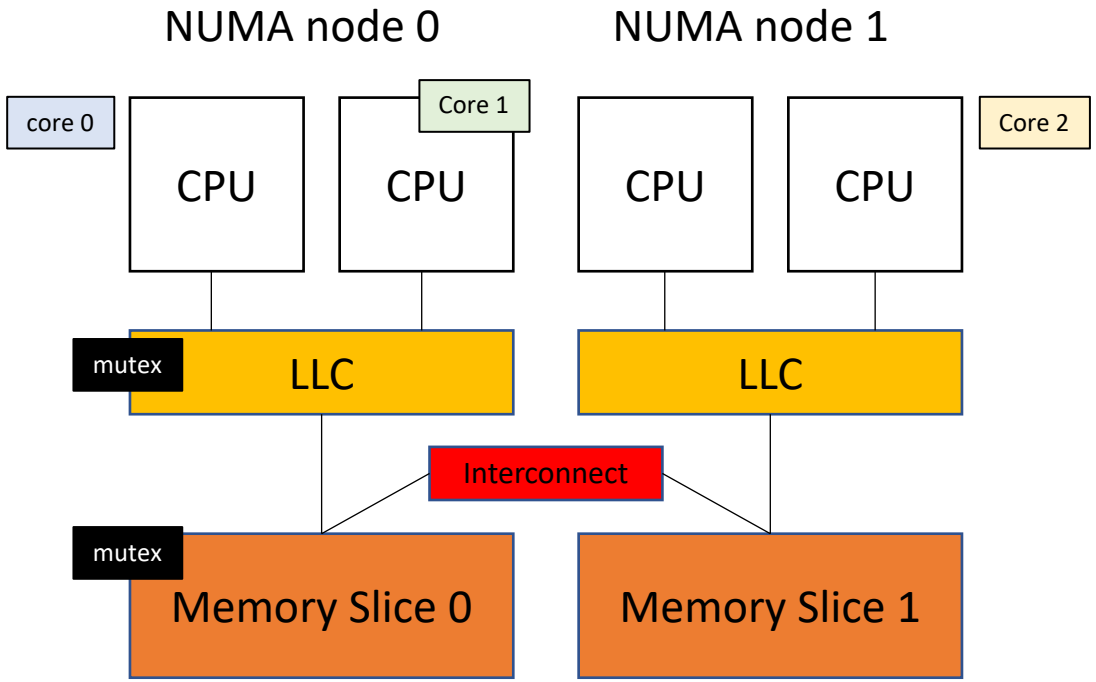


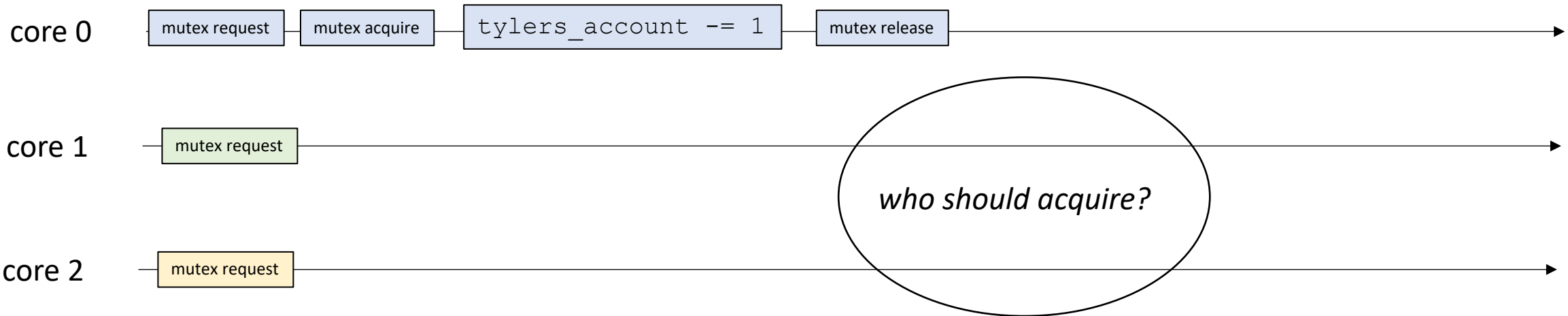

NUMA node 0
NUMA node 1
core 0
Core 1
Core 2
CPU
CPU
CPU
CPU
LLC
LLC
Interconnect
mutex
Memory Slice 0
Memory Slice 1


core 0
mutex request
mutex acquire
tylers_account -= 1
mutex release
core 1
mutex request
core 2
mutex request

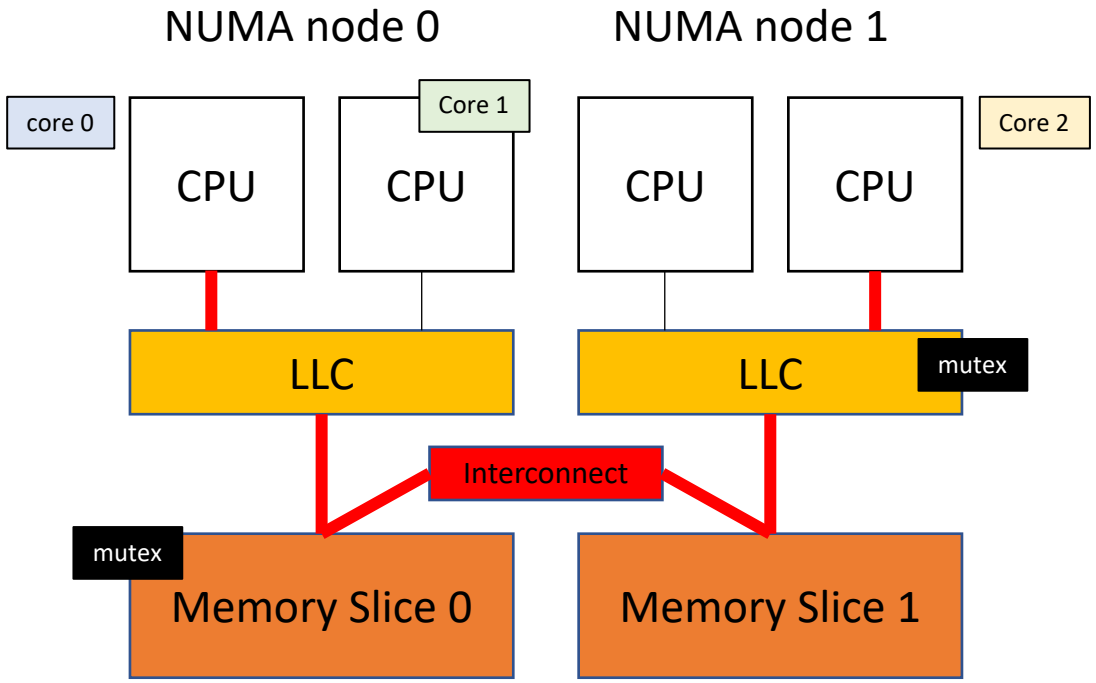

NUMA node 0
NUMA node 1
core 0
CPU
Core 1
CPU
CPU
CPU
Core 2
mutex
LLC
LLC
Interconnect
mutex
Memory Slice 0
Memory Slice 1

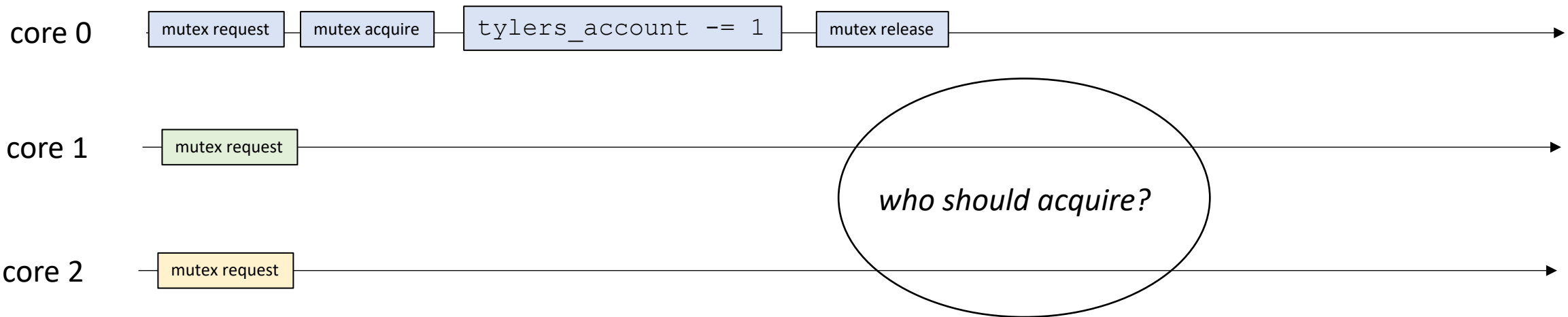

core 0
mutex request
mutex acquire
tylers_account -= 1
mutex release
core 1
mutex request
core 2
mutex request


NUMA node 0
NUMA node 1
core 0
Core 1
Core 2
CPU
CPU
CPU
CPU
mutex
LLC
LLC
Interconnect
mutex
Memory Slice 0
Memory Slice 1

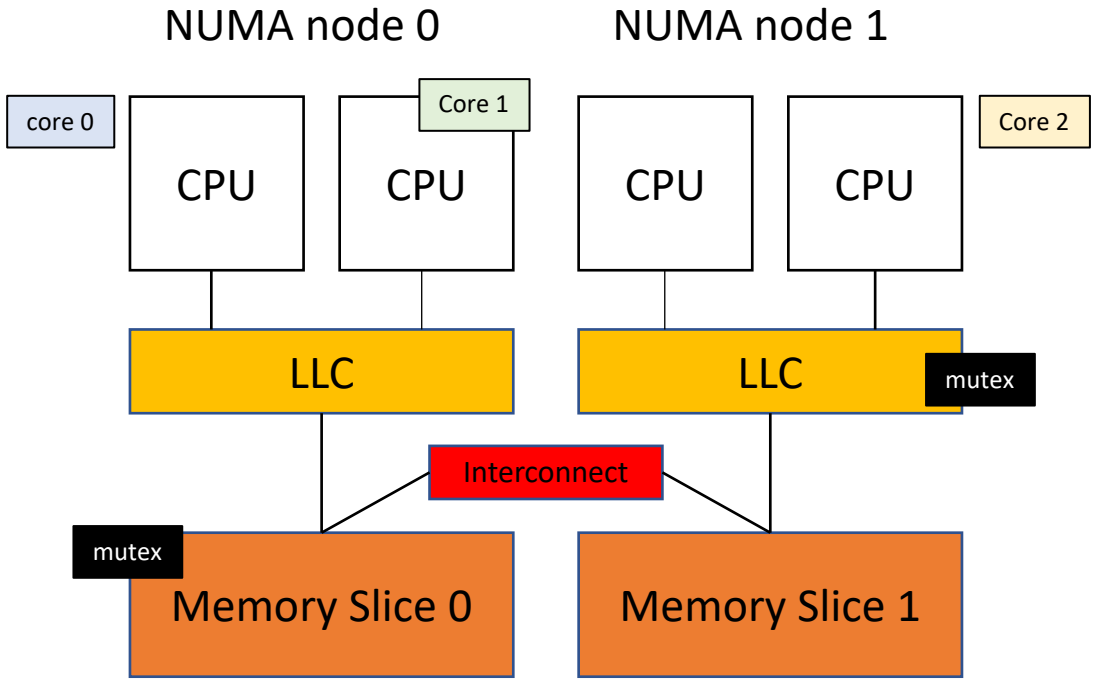

core 0
mutex request
mutex acquire
tylers_account -= 1
mutex release
core 1
mutex request
who should acquire?
core 2
mutex request

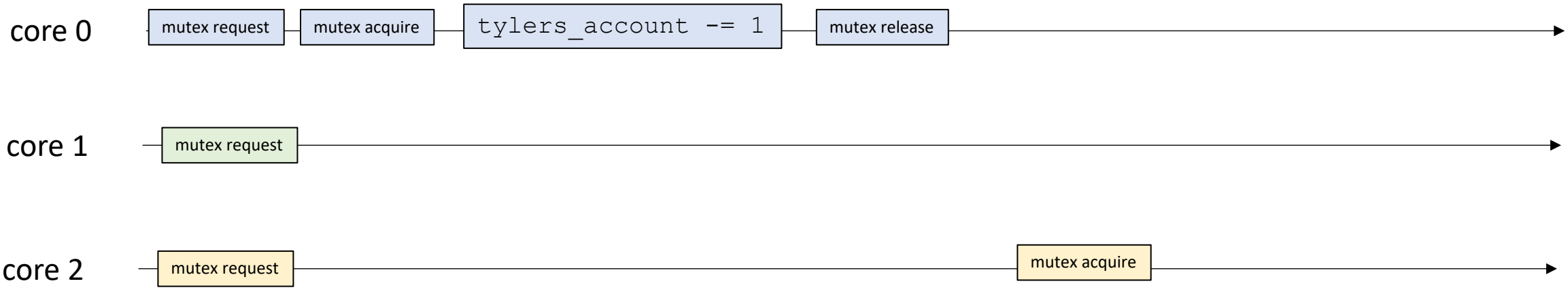
If core 2 acquires first communication must go through the interconnect



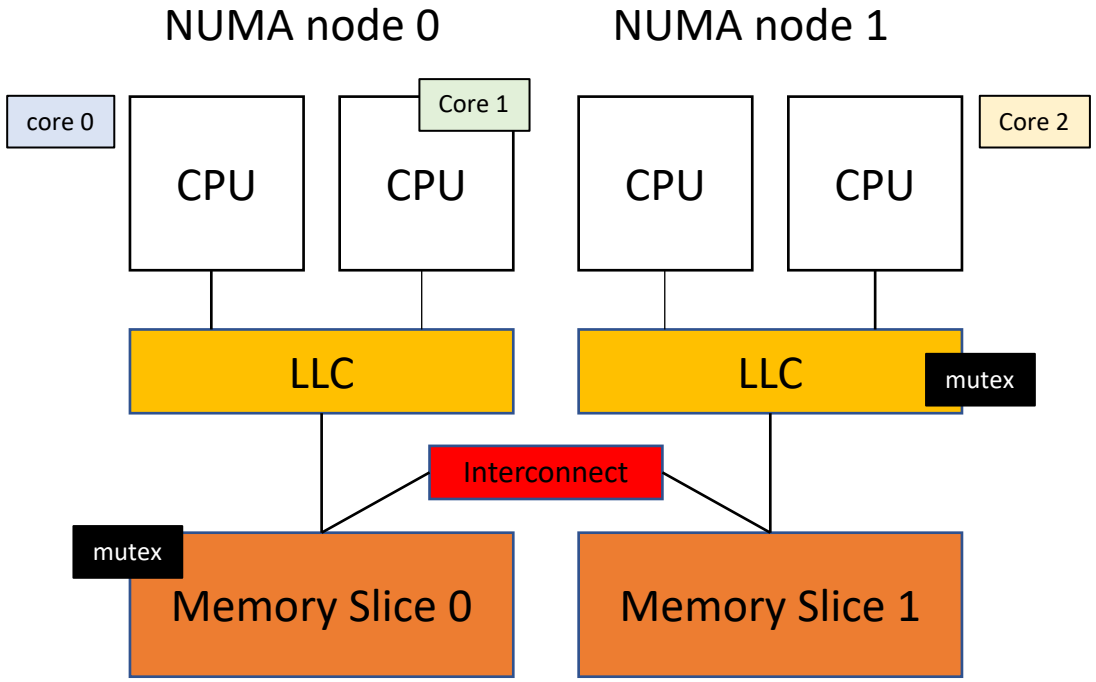


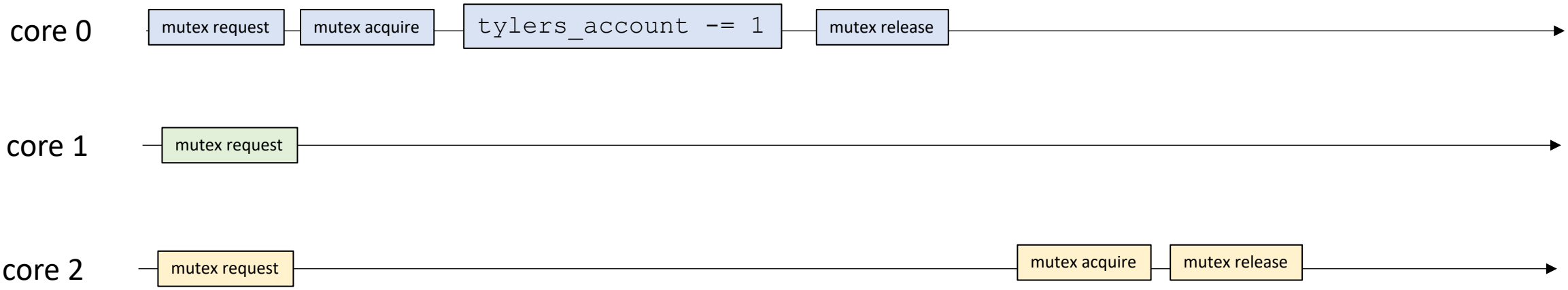
If core 2 acquires first communication must go through the interconnect





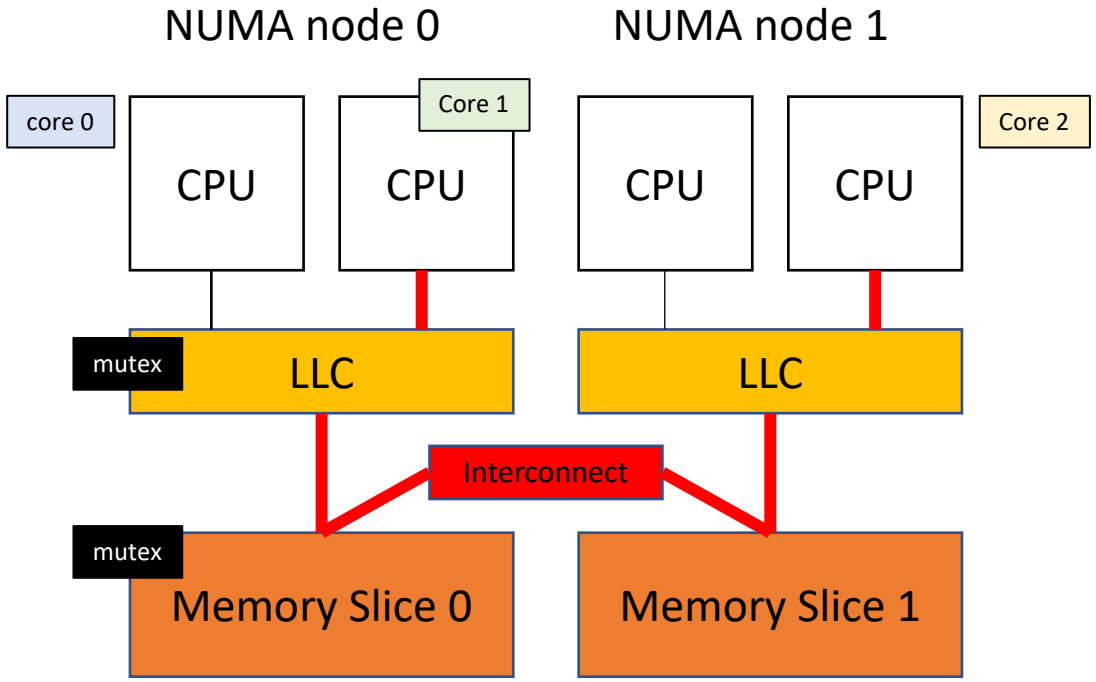
If core 2 acquires first communication must go through the interconnect

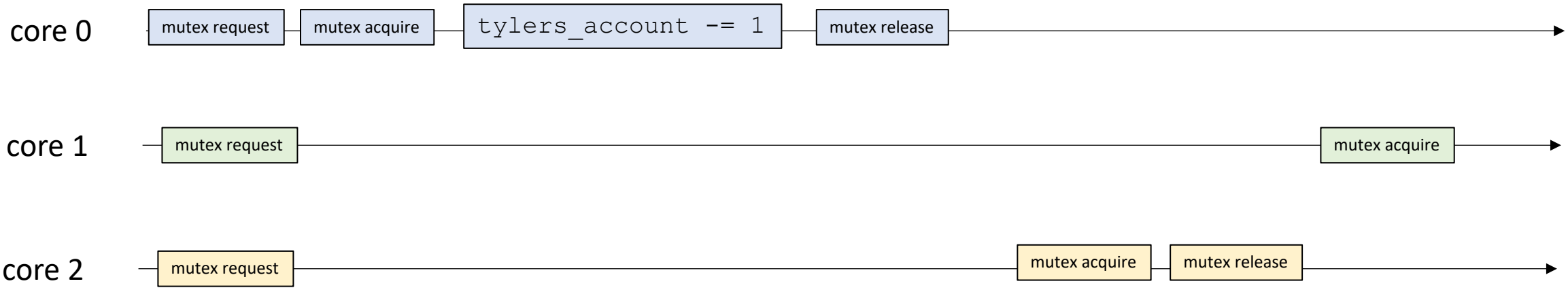




If core 2 acquires first communication must go through the interconnect

When core 1 finally acquires, it requires another expensive trip through the interconnect

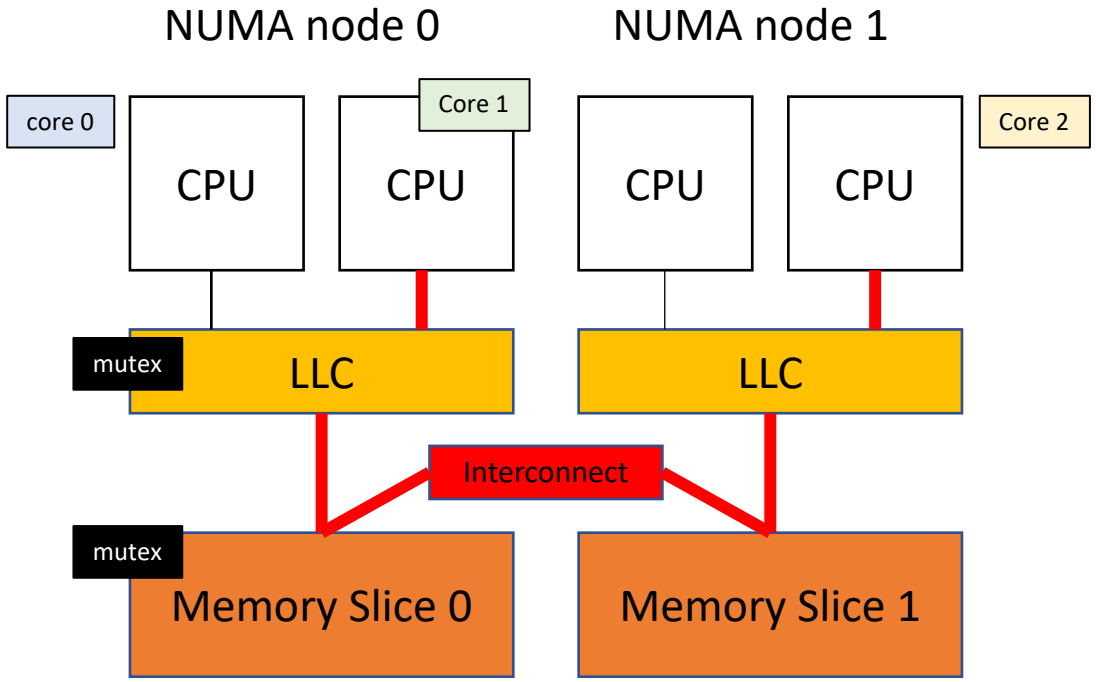


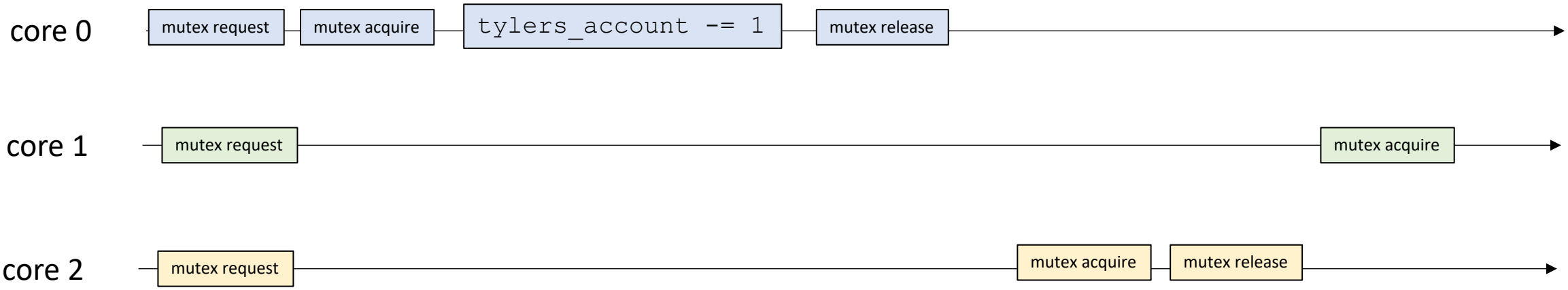


Two trips through the interconnect!!

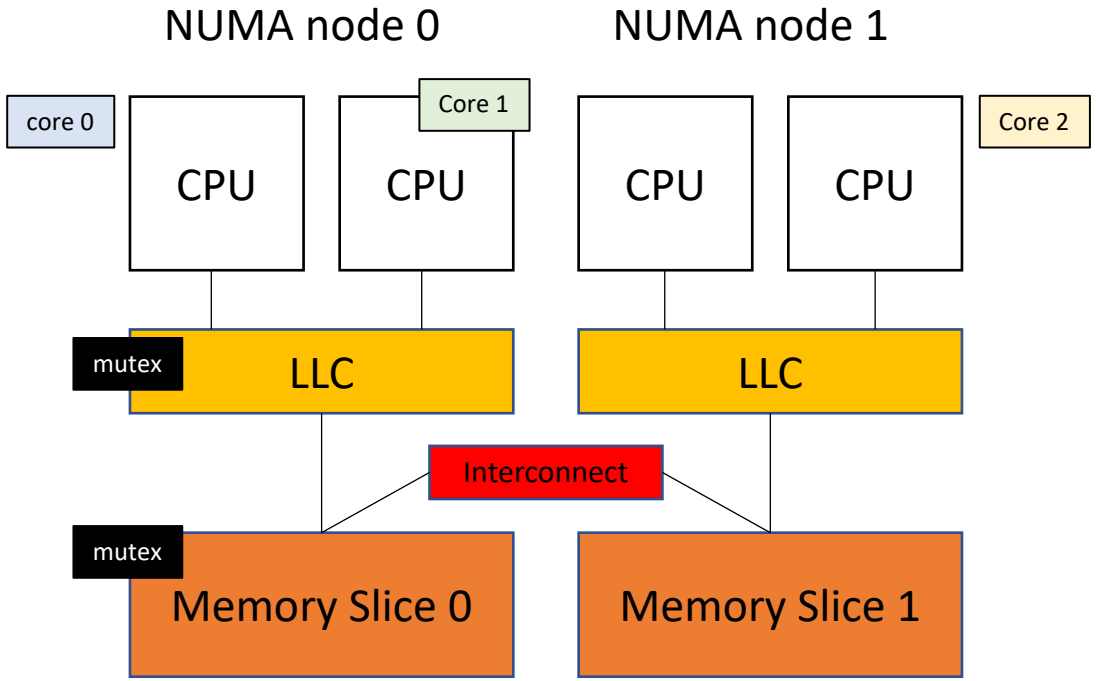
If core 2 acquires first communication must go through the interconnect

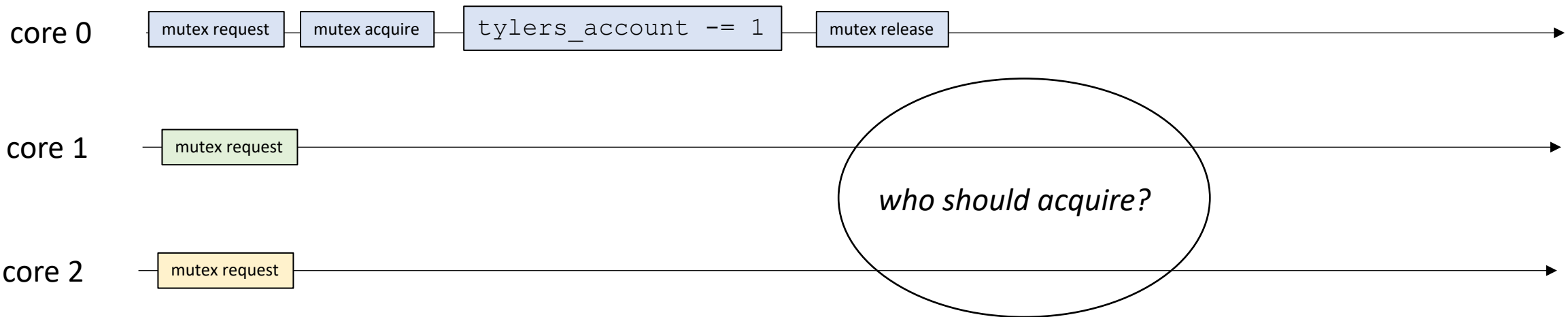
When core 1 finally acquires, it requires another expensive trip through the interconnect



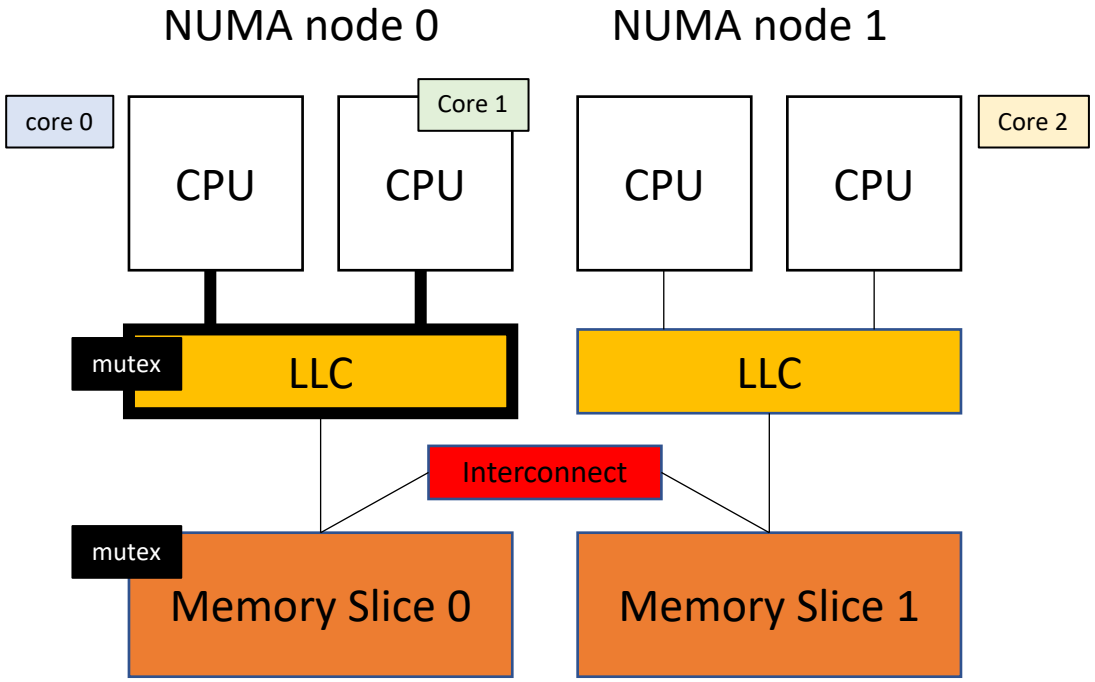


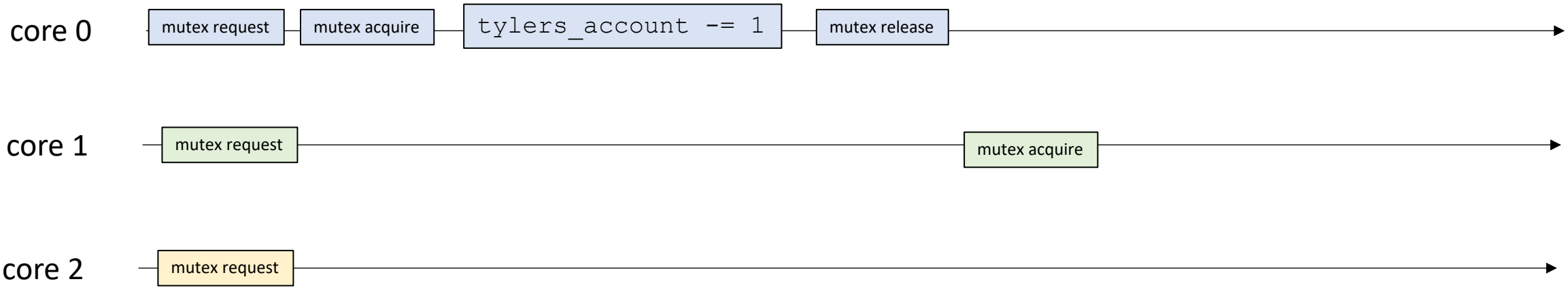
Lets go back in time and make a different decision!





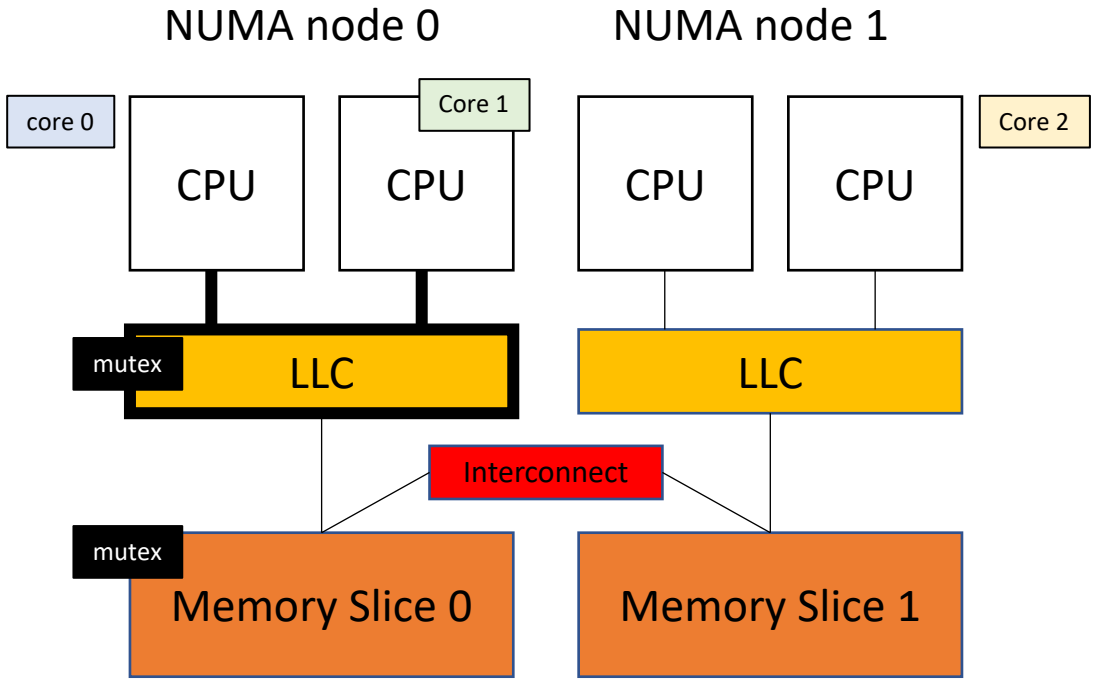
If core 1 acquires first communication can occur through the LLC of NUMA node 0

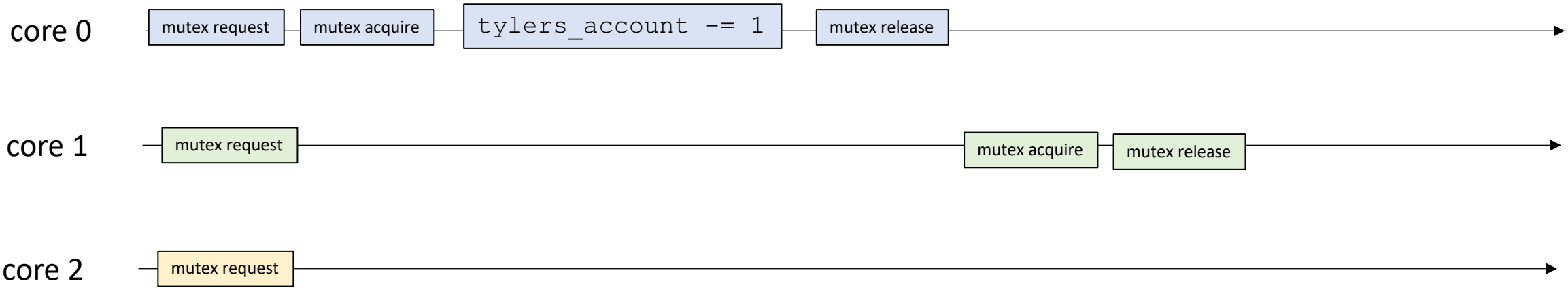




If core 1 acquires first communication can occur through the LLC of NUMA node 0

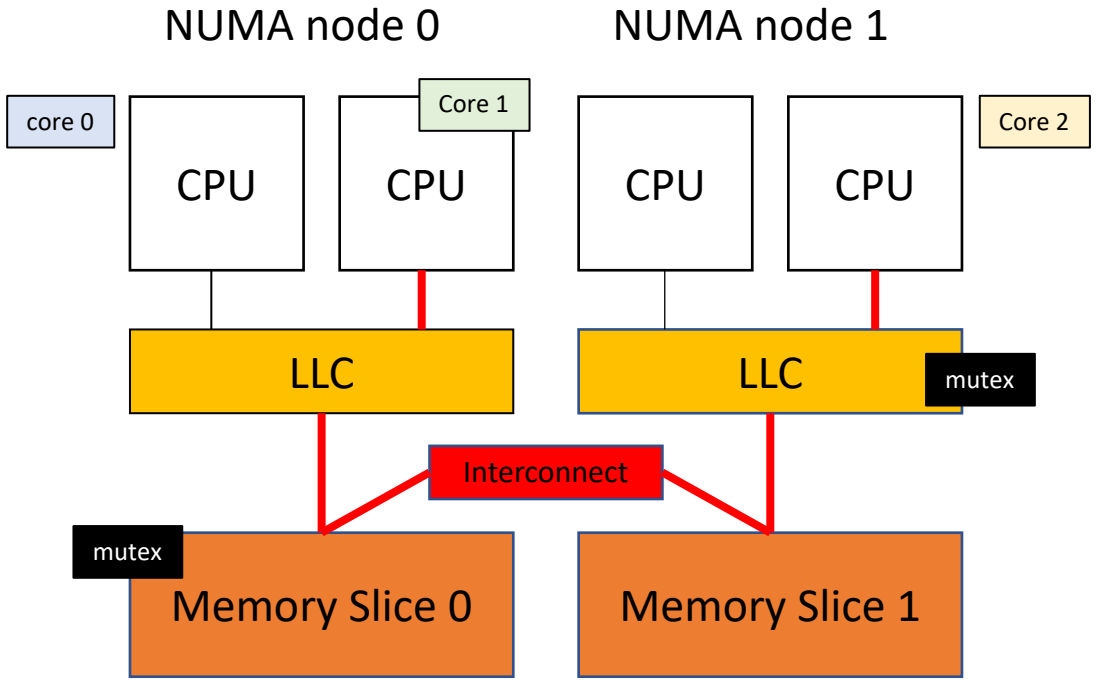
When core 2 finally acquires it requires an expensive trip through the interconnect

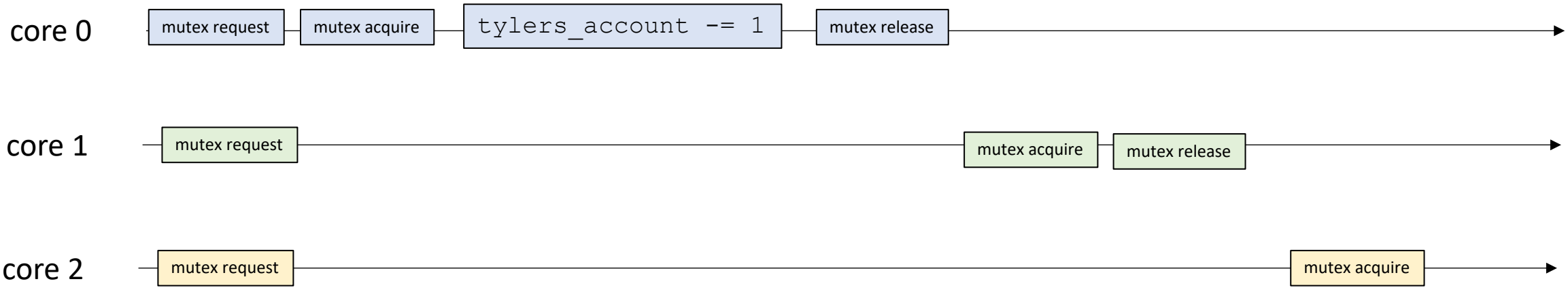




If core 1 acquires first communication can occur through the LLC of NUMA node 0

When core 2 finally acquires it requires an expensive trip through the interconnect

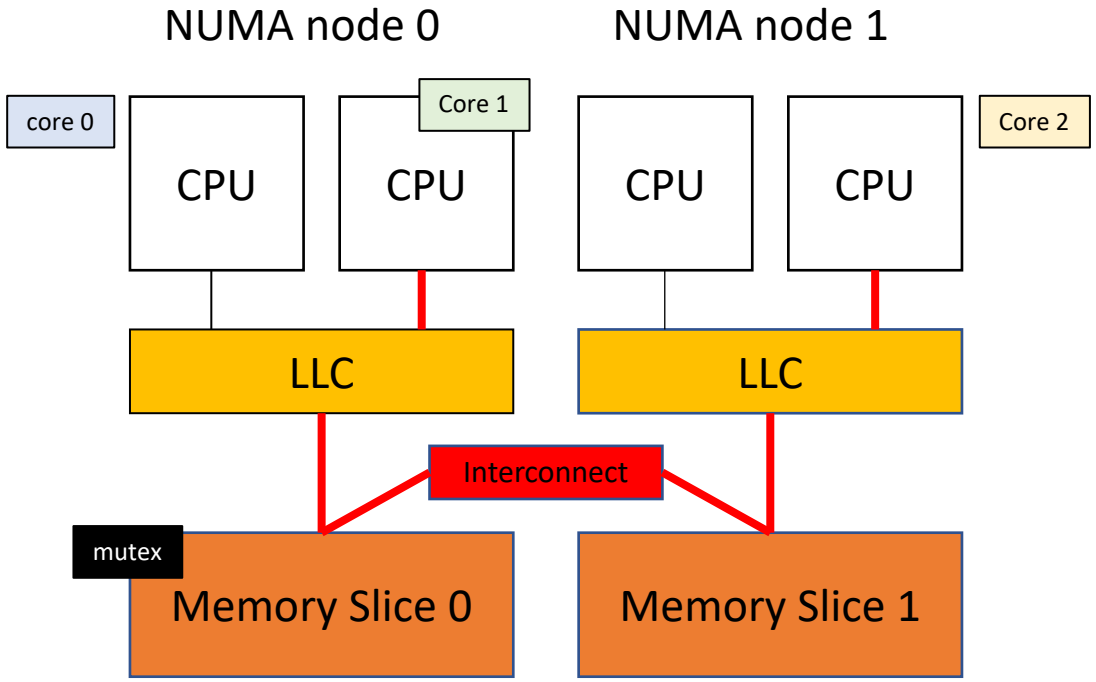


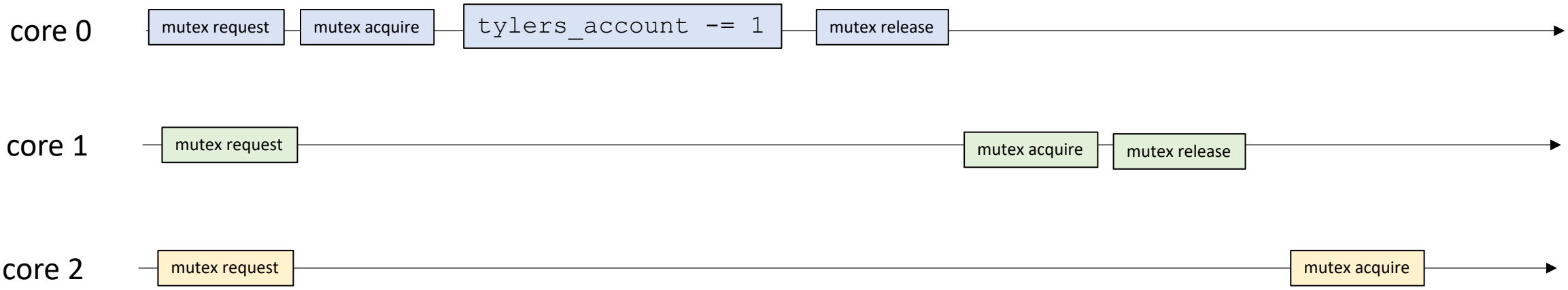


Only 1 trip through the interconnect

If core 1 acquires first communication can occur through the LLC of NUMA node 0

When core 2 finally acquires it requires an expensive trip through the interconnect





Hierarchical locks

- If thread T in NUMA node N holds the mutex:
  - the mutex should prioritize other threads in NUMA node N to acquire the mutex when T releases it.

- We will do this in two steps:
  - Slightly modify the CAS mutex
  - Add targeted sleeping

Hierarchical locks

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

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

  void lock();
  void unlock();

private:
  atomic_bool flag;
};
```

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

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

  void lock();
  void unlock();

 private:
  atomic_int m_owner;
};
```

New CAS lock

the value of -1 means the mutex is available

In the new mutex, we switch from a flag to an int.

Hierarchical locks

main idea is that threads put their thread ids in the mutex

No longer possible with exchange lock!

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

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

  void lock();
  void unlock();

 private:
  atomic_int m_owner;
};
```

the value of -1 means the mutex is available

In the new mutex, we switch from a flag to an int.

new lock: we attempt to put our thread id in the mutex when we lock.

```
void lock(int thread_id) {
  int  e = -1;
  int acquired = false;
  while (acquired == false) {
    acquired = atomic_compare_exchange_strong(&m_owner, &e, thread_id);
    e = -1;
  }
}
```

previously we didn't require a thread id. We just used true and false

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

Unlock is boring as usual

```
void unlock() {
  m_owner.store(-1);
}
```

We have a new lock

- But there isn’t any hierarchy yet.

- What value is in ‘e’ after a failed lock attempt?

```
void lock(int thread_id) {
  int  e = -1;
  int acquired = false;
  while (acquired == false) {
    acquired = atomic_compare_exchange_strong(&m_owner, &e, thread_id);
    e = -1;
  }
}
```

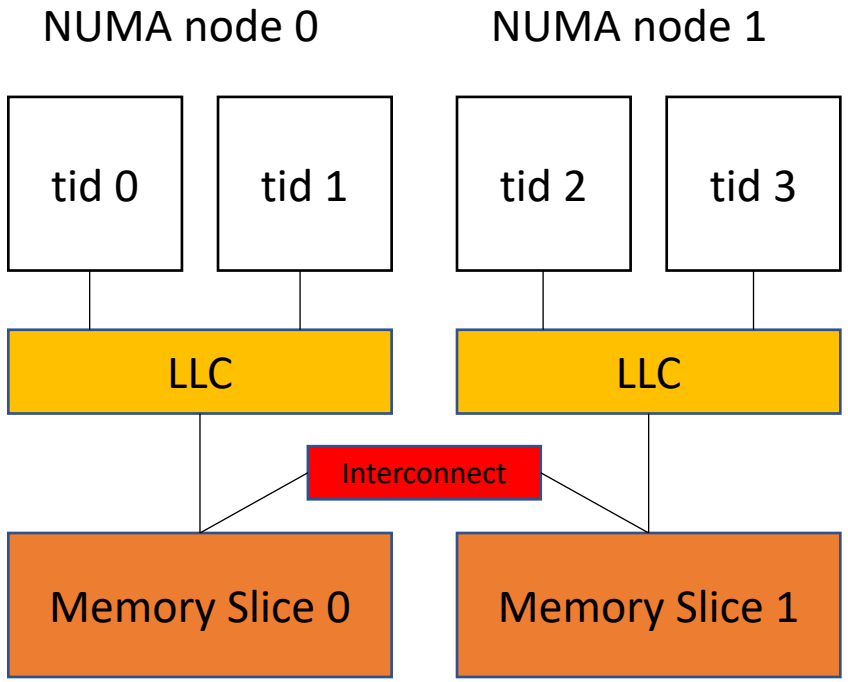
We have a new lock

- But there isn’t any hierarchy yet.

- What value is in ‘e’ after a failed lock attempt?

```
void lock(int thread_id) {
  int  e = -1;
  int acquired = false;
  while (acquired == false) {
    acquired = atomic_compare_exchange_strong(&m_owner, &e, thread_id);
    e = -1;
  }
}
```

we know what thread currently owns the mutex!



Given a thread ID, we can compute the NUMA node ID of the thread using integer division (floor):

```
thread_id / 2
```

```
thread_id / THREADS_PER_NUMA_NODE
```

GPUs give this as a builtin

Hierarchical lock

- We know our thread id (passed in)
- We know the thread id of the thread that owns the mutex (returned in ‘e’)

- Check if we are in the same NUMA node as the thread that owns the mutex.
  - if not, sleep for a long time
  - else sleep for a short time

```
void lock(int thread_id) {
  int e = -1;
  bool acquired = false;
  while (acquired == false) {
    acquired = atomic_compare_exchange_strong(&m_owner, &e, thread_id);

    if (thread_id/2 != e/2) {
      this_thread::sleep_for(10ms);
    }
    else {
      this_thread::sleep_for(1ms);
    }
    e = -1;
  }
}
```

Starvation?

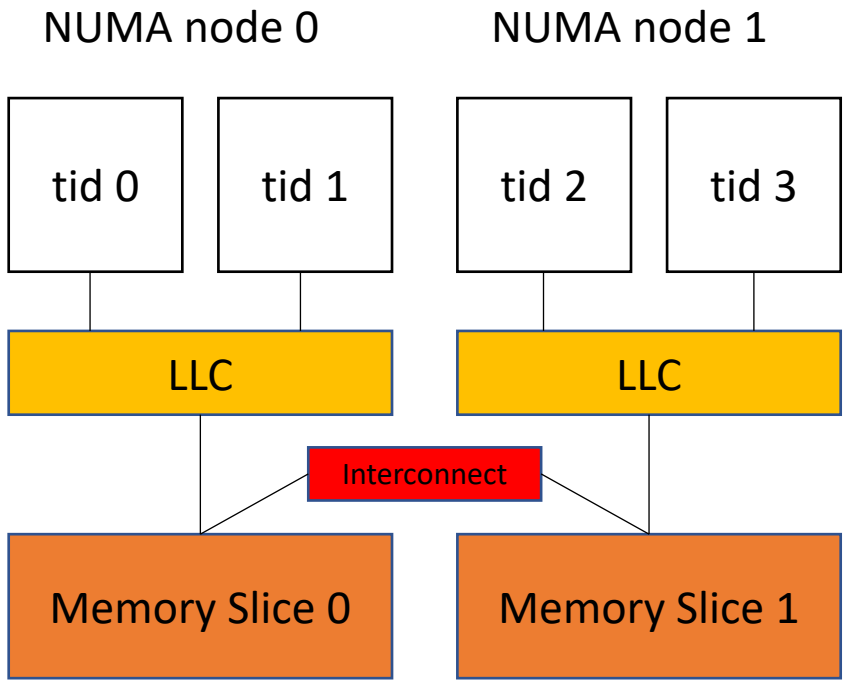
- Tune sleep times. You shouldn’t starve the other nodes!

- Advanced: have internal mutex state that counts how long the mutex has stayed with in the NUMA node.

Example:

tid 0:
tid 1:
tid 2:

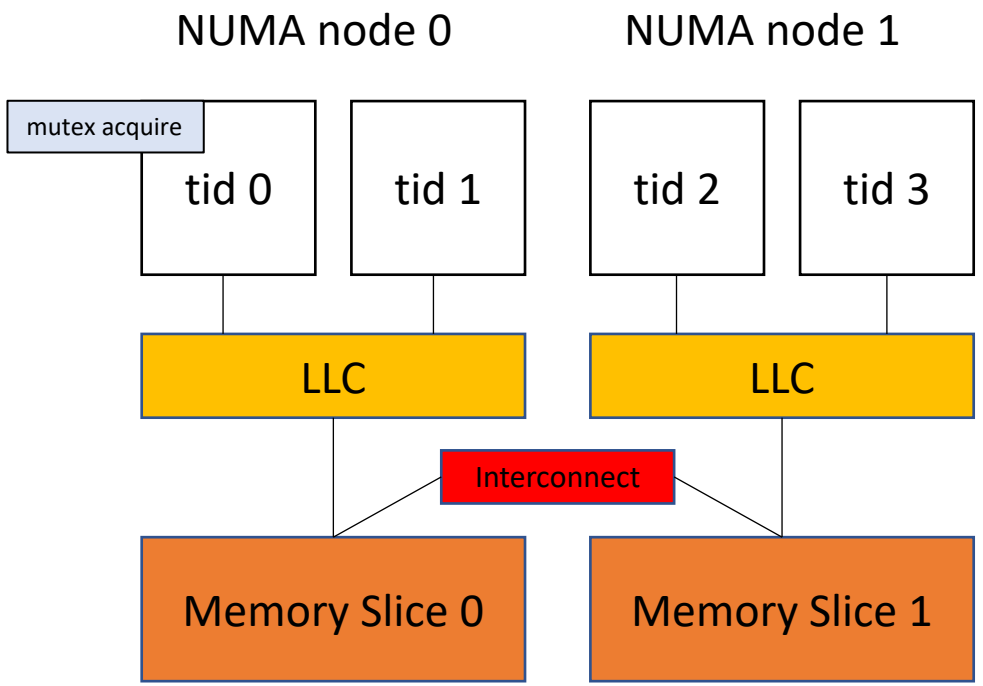
Mutex counter:
Local_Com: 0



Example:

tid 0: Acquired
tid 1: sleep 1 ms
tid 2: sleep 100 ms

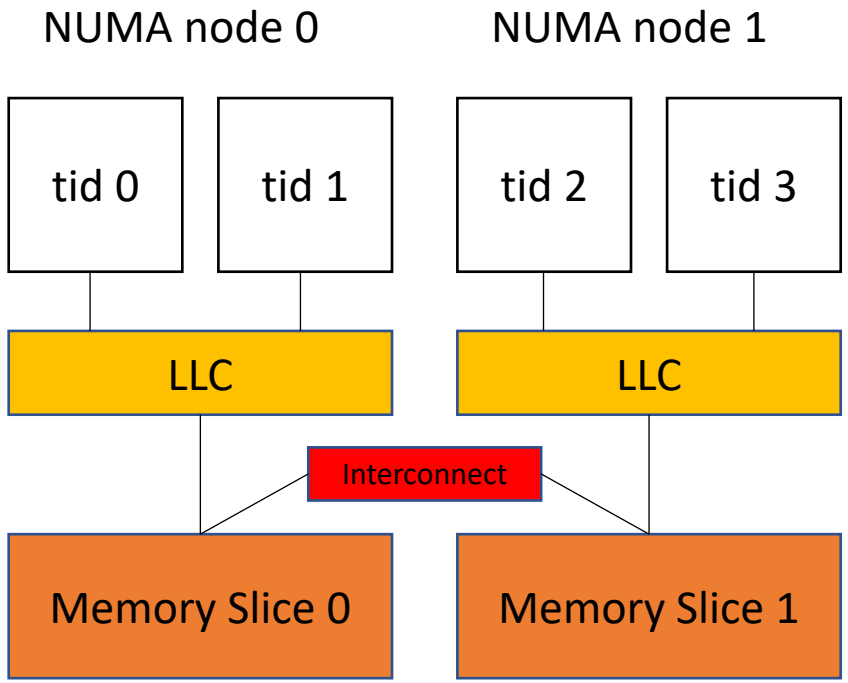
Mutex counter:
Local_Com: 1



Example:

tid 0:
tid 1:
tid 2:

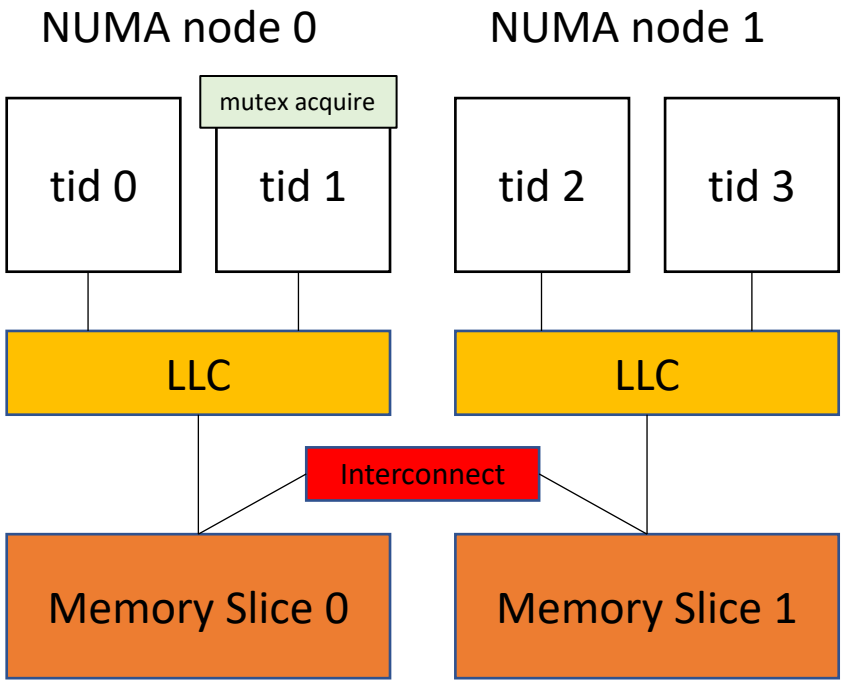
Mutex counter:
Local_Com: 1



Example:

tid 0: sleep 1 ms
tid 1: acquired
tid 2: sleep 100 ms

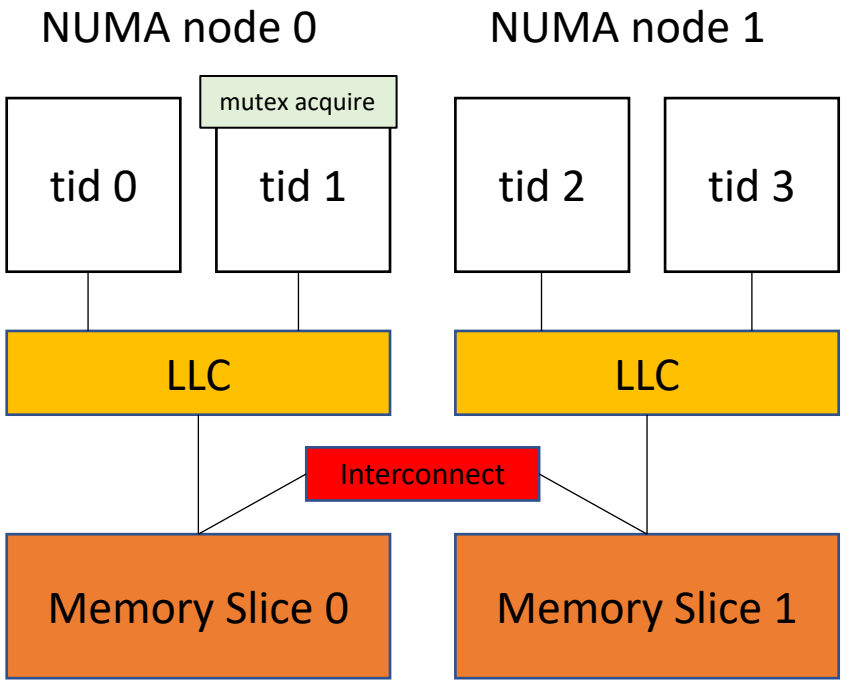
Mutex counter:
Local_Com: 2



Example:

tid 0: sleep 1 ms
tid 1: acquired
tid 2: sleep 100 ms

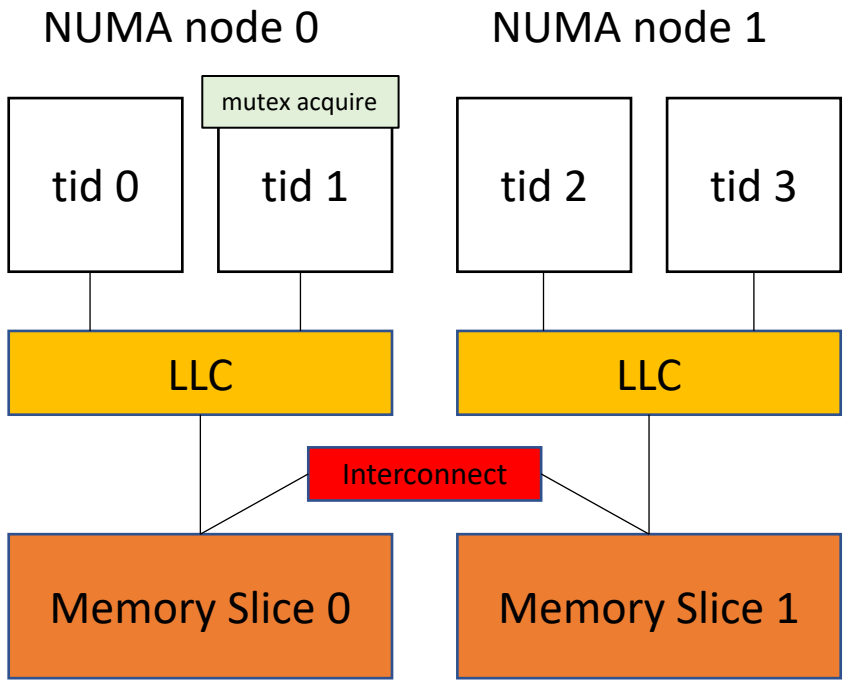
Mutex counter:
Local_Com: 2



Example:

tid 0: sleep 1 ms * Local_Com = 2 ms
tid 1: acquired
tid 2: sleep 100 ms

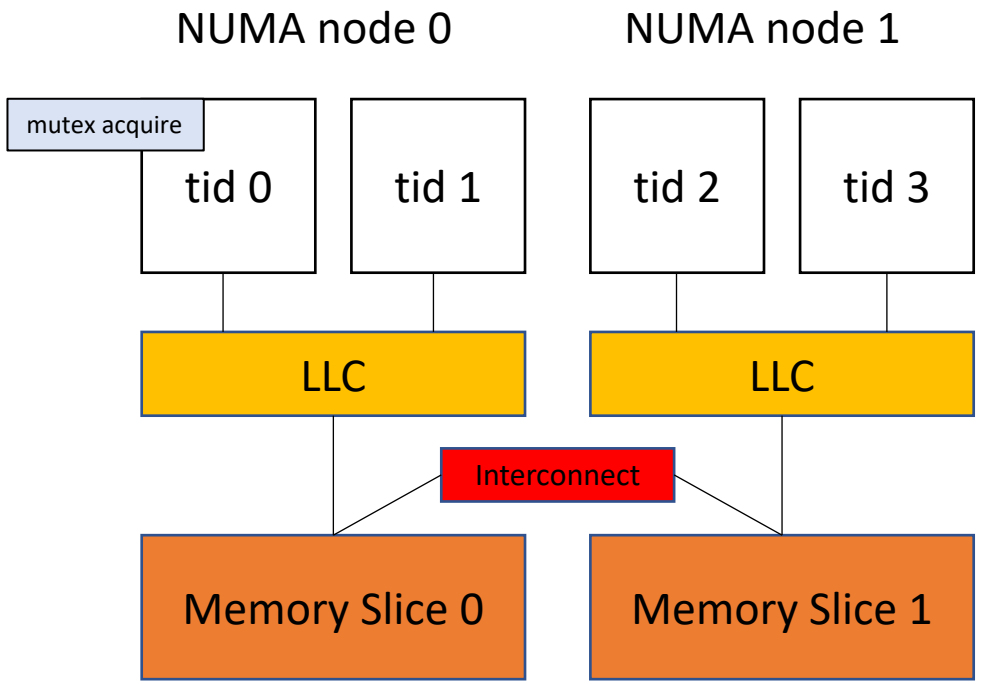
Mutex counter:
Local_Com: 2



Example:

tid 0: acquired
tid 1: sleep 1 ms * Local_Com = 3 ms
tid 2: sleep 100 ms

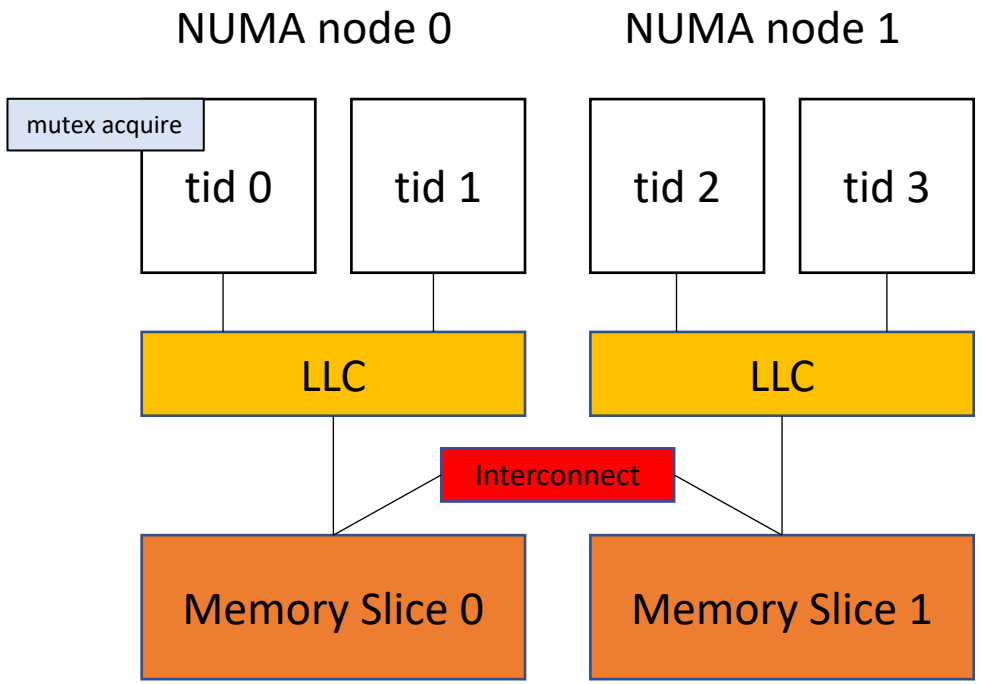
Mutex counter:
Local_Com: 3



Example:

tid 0: acquired
tid 1: sleep 1 ms * Local_Com = 3 ms
tid 2: sleep 100 ms

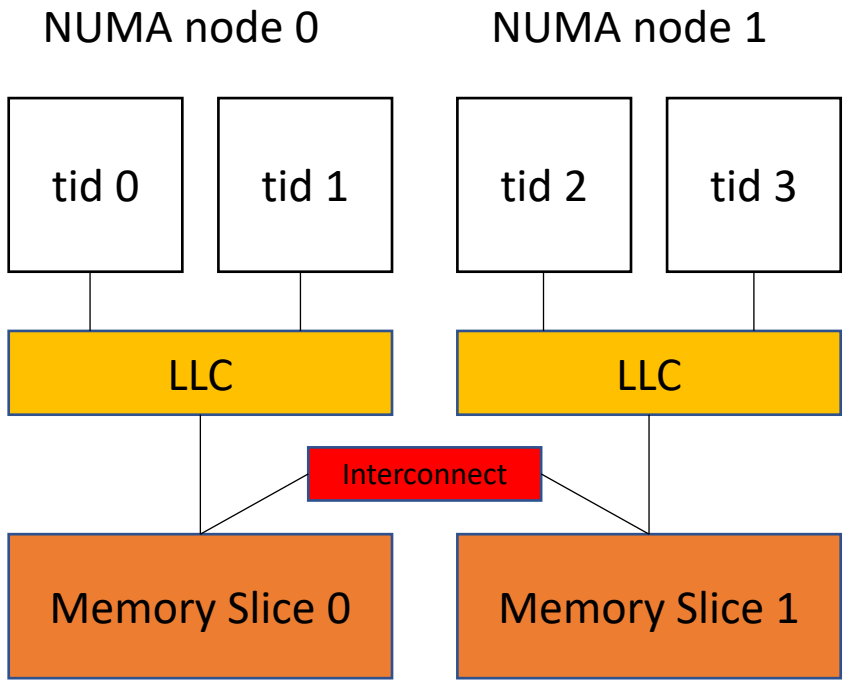
Mutex counter:
Local_Com: 3



Example:

tid 0:
tid 1:
tid 2:

Mutex counter:
Local_Com: 3

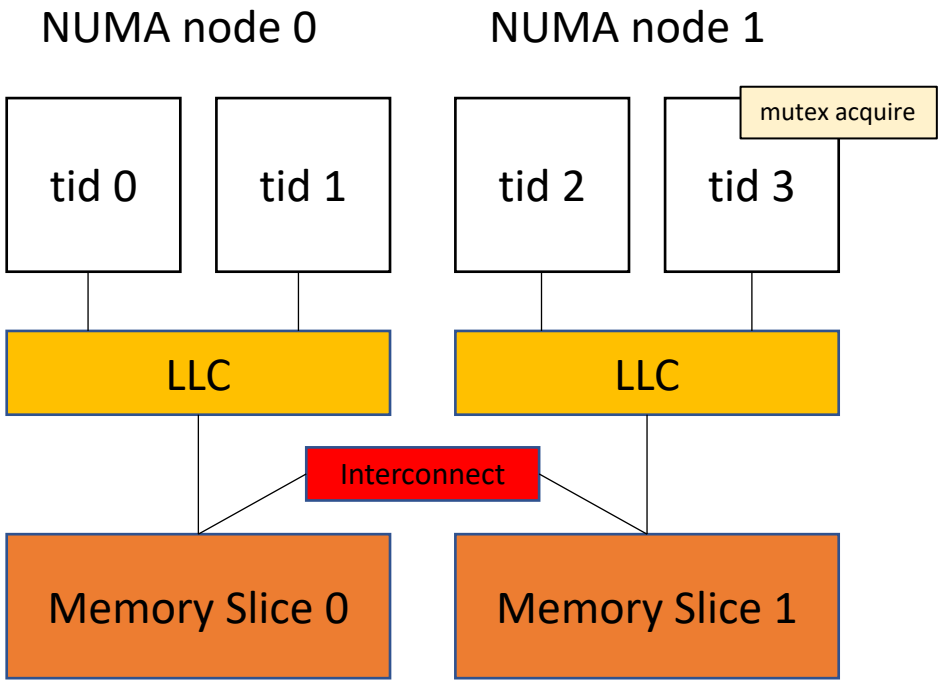


Example:

tid 0:
tid 1:
tid 2:

Mutex counter:
Local_Com: 1

reset because
we moved across nodes



Further reading

- More elaborate schemes:
  - Queue locks - spinning on different cache lines
  - Composite locks - combining queue locks and RMW locks
  - Fair hierarchical locks

- Read in the book to learn more!