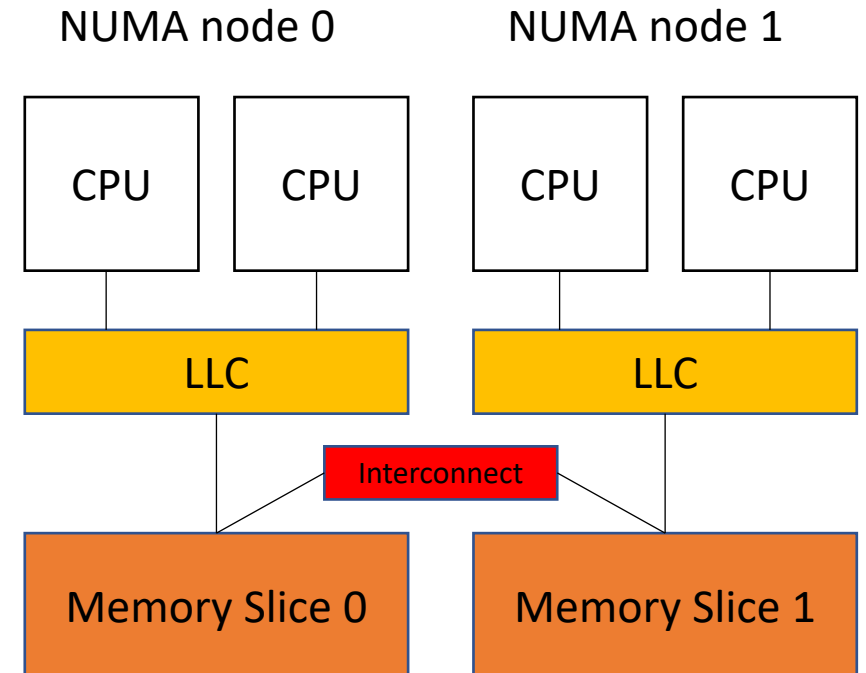


CSE113: Parallel Programming

April 22, 2021

- **Topic:** Specialized Mutexes
 - Optimizing Locks - backoff, hierarchical
 - Checking code for data conflicts



Announcements

- Homework due today by midnight
 - Reese has office hours after lecture
 - Gan had them earlier today
 - Check out my office hour recording from yesterday
 - You should have many resources if you need help!
- New homework is posted tonight by midnight.

Announcements

- **Midterm**

- Posted in 1 week
- Inclusive up to the lecture that day
- Asynchronous - you have one week
- Do not ask each other questions about the test or discuss it with each other until it is due.
- You can ask us clarifying questions in private piazza threads or teach mailing list.
- Estimated time is 2 hours. ***Excluding*** studying and checking answers!

Quiz

- Canvas Quiz

Quiz

- Canvas Quiz
- Discuss answers

Experiments From Last Week

- Look at the mutex throughput benchmark
 - You have a similar wrapper for your homework

Experiments From Last Week

- Look at the mutex throughput benchmark
 - You have a similar wrapper for your homework
- Throughput of ticket lock vs CAS lock

Experiments From Last Week

- Look at the mutex throughput benchmark
 - You have a similar wrapper for your homework
- Throughput of ticket lock vs CAS lock
- Adding relaxed peeking

Lecture Schedule

- Optimizations
 - Backoff
 - Hierarchical aware locks
 - Further reading
- RW locks:
 - Implementation
 - C++
- Checking for data conflicts

Lecture Schedule

- **Optimizations**

- **Backoff**

- Hierarchical aware locks

- Further reading

- RW locks:

- Implementation

- C++

- Checking for data conflicts

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

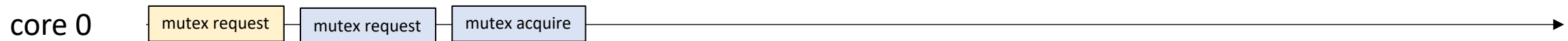
core 0



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**

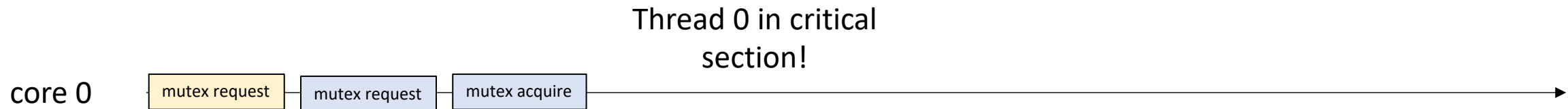
Say threads 0 and 1 are executing concurrently



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**

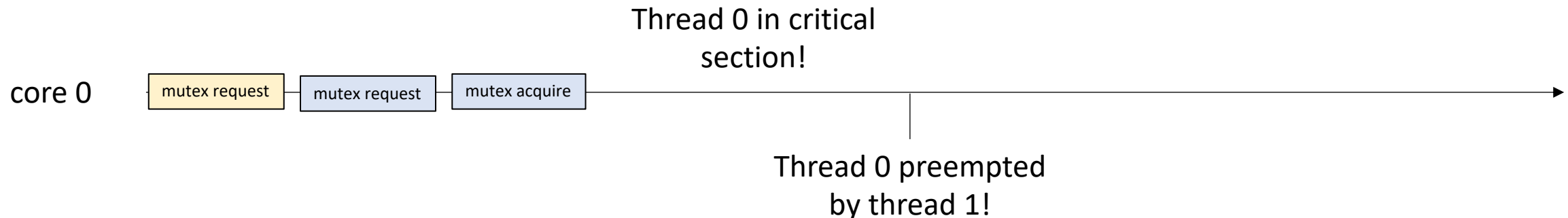
Say threads 0 and 1 are executing concurrently



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**

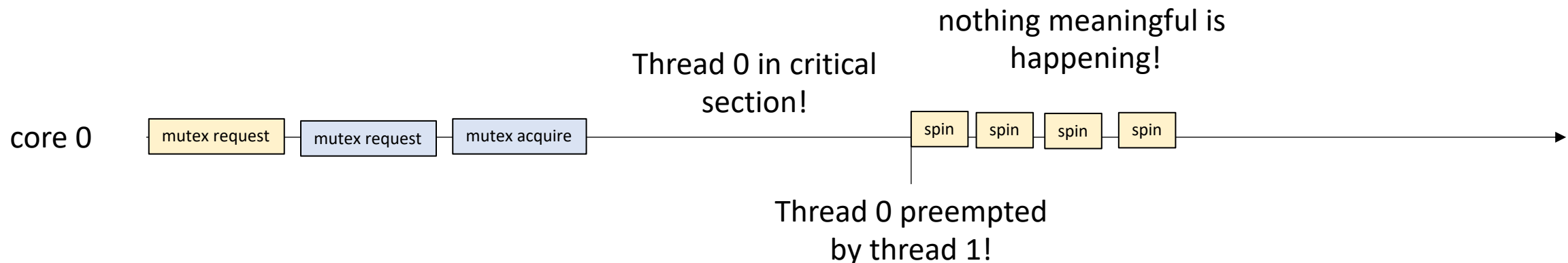
Say threads 0 and 1 are executing concurrently



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**

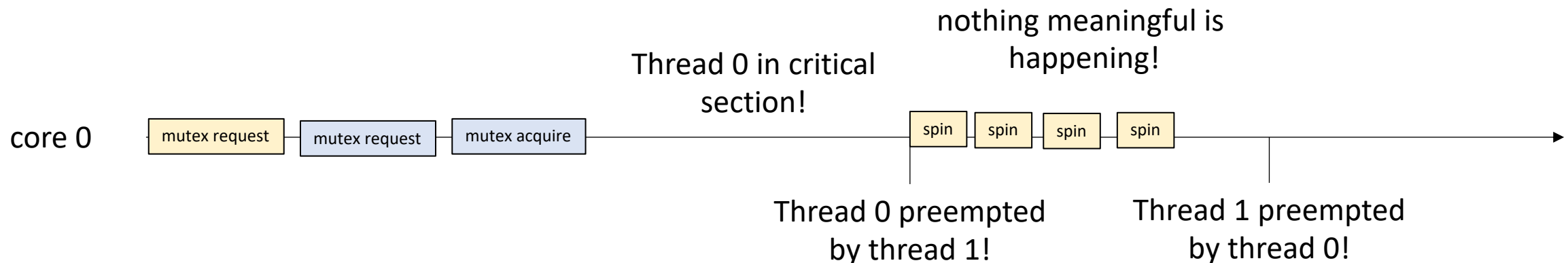
Say threads 0 and 1 are executing concurrently



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**

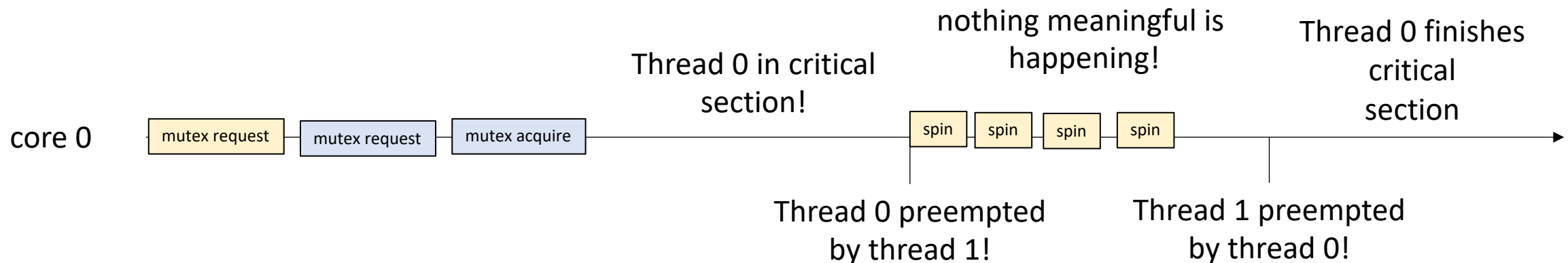
Say threads 0 and 1 are executing concurrently



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**

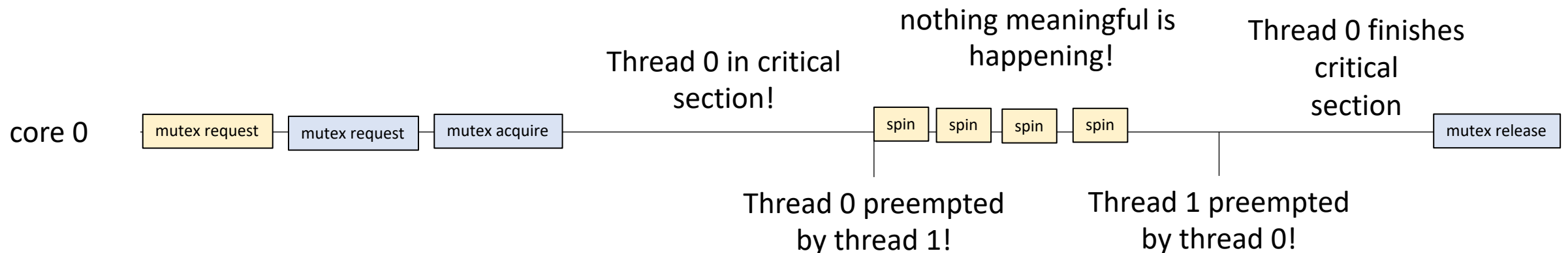
Say threads 0 and 1 are executing concurrently



Optimizations: backoff

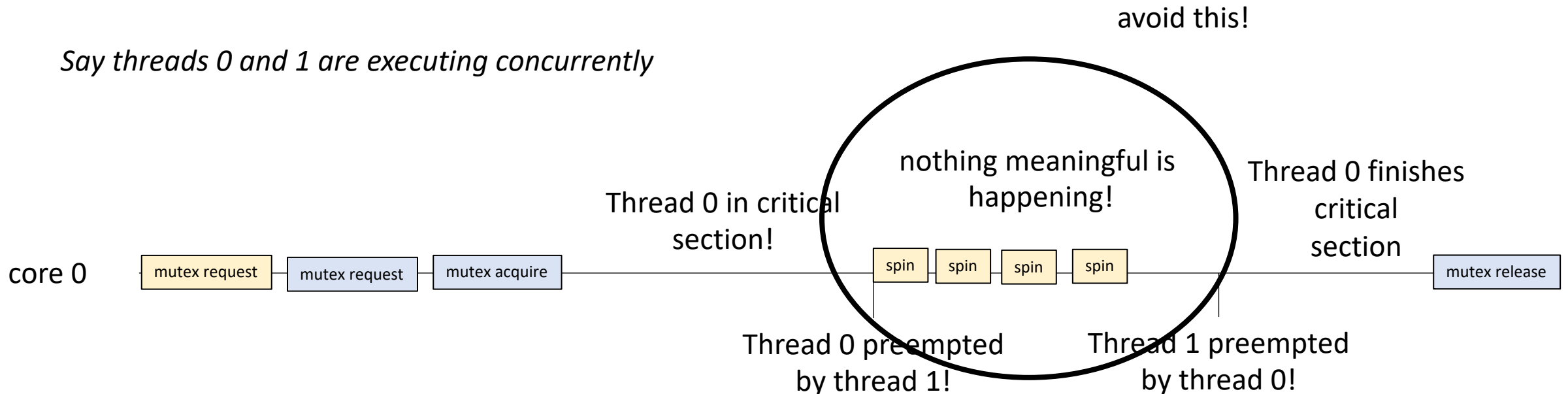
- Two issues remain:
 - Loads still cause bus traffic (even if its not as bad as RMWs)
 - **In non-parallel systems, concurrent threads can get in the way of progress**

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 (and implemented one that is just as efficient in many cases!)

Lecture Schedule

- Optimizations
 - Backoff
 - Hierarchical aware locks
 - Further reading
- RW locks:
 - Implementation
 - C++
- Checking for data conflicts

Lecture Schedule

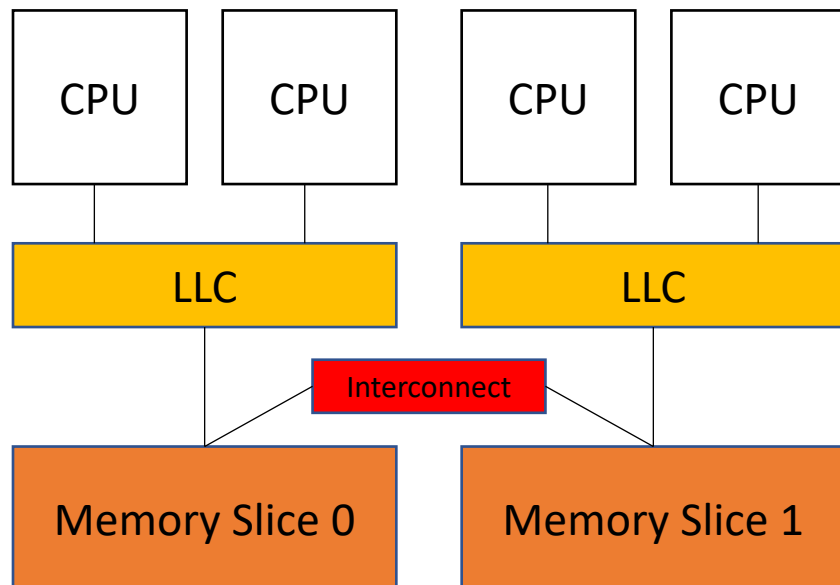
- Optimizations
 - Backoff
 - **Hierarchical aware locks**
 - Further reading
- RW locks:
 - Implementation
 - C++
- Checking for data conflicts

Optimization: Hierarchical locks

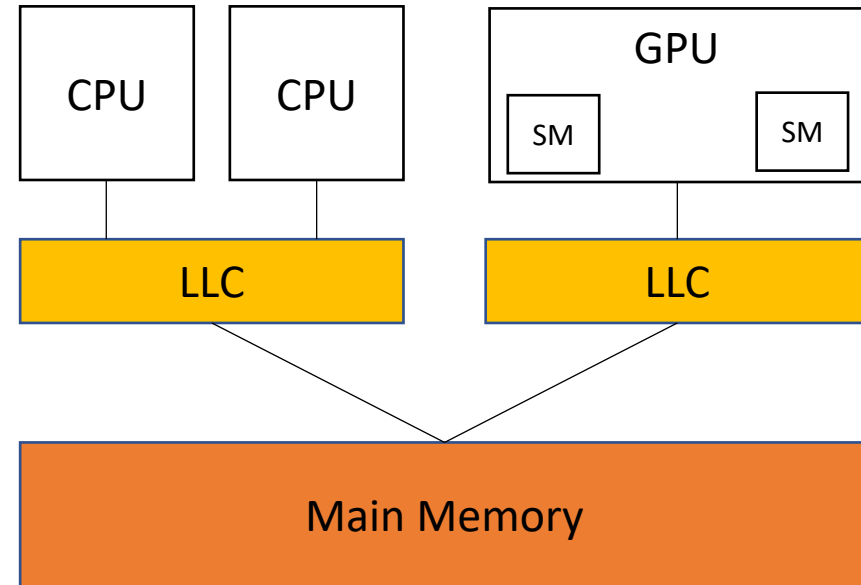
- NUMA (non-uniform memory access) systems
- heterogeneous systems (CPU GPU)

Discrete GPUs communicate through PCIE

For example: Large server nodes

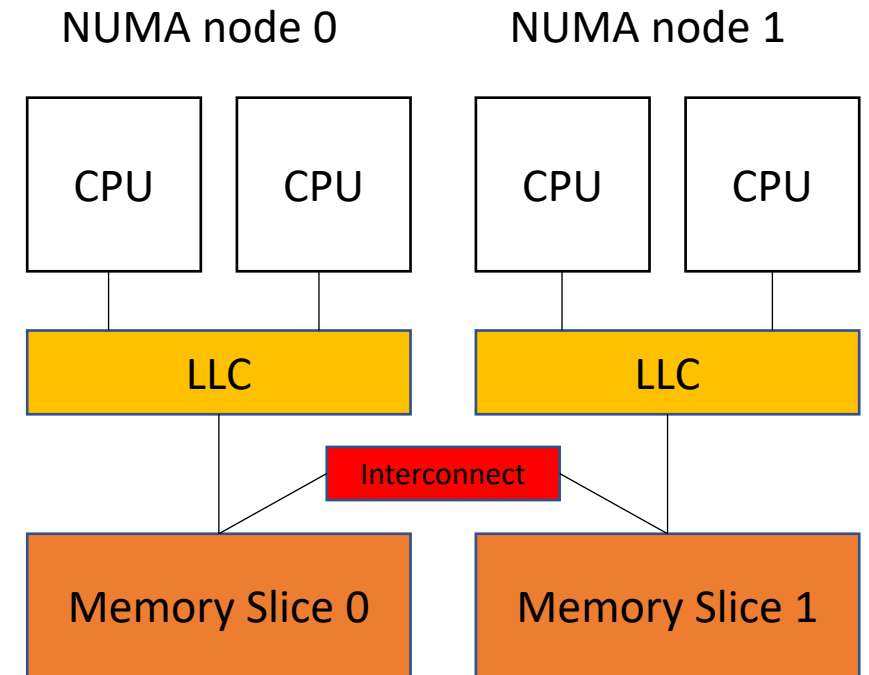


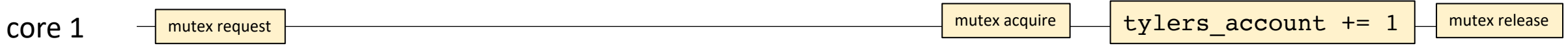
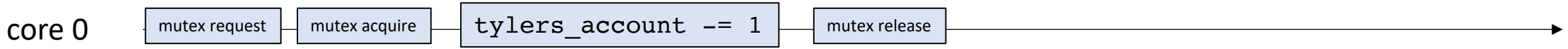
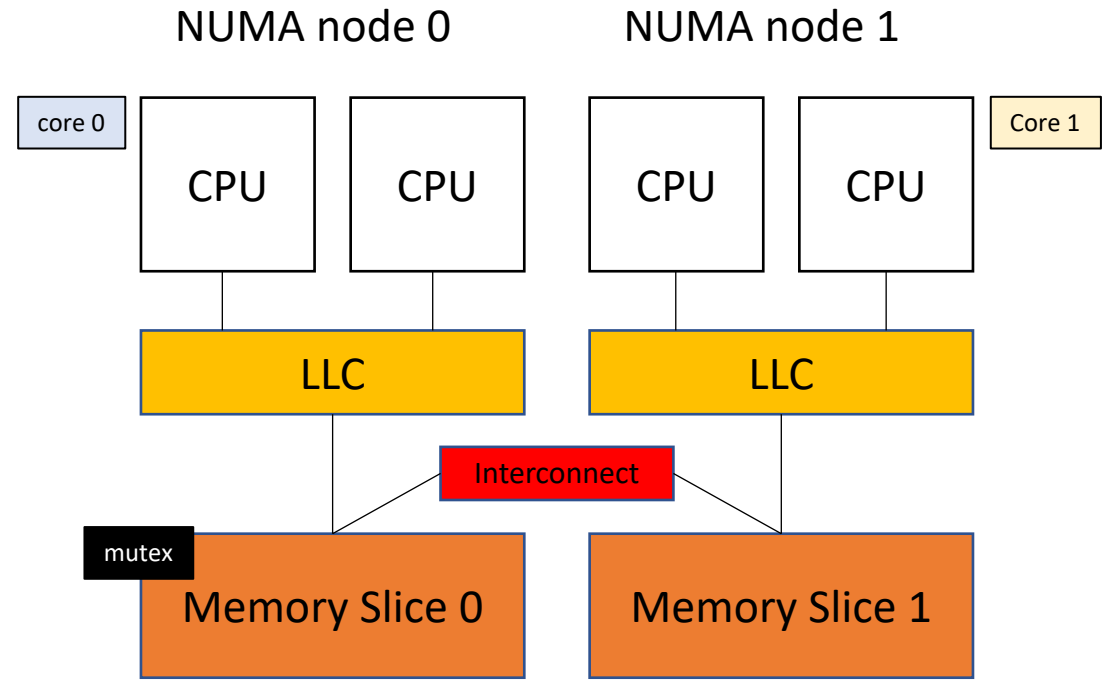
For example: SoCs like Iphone

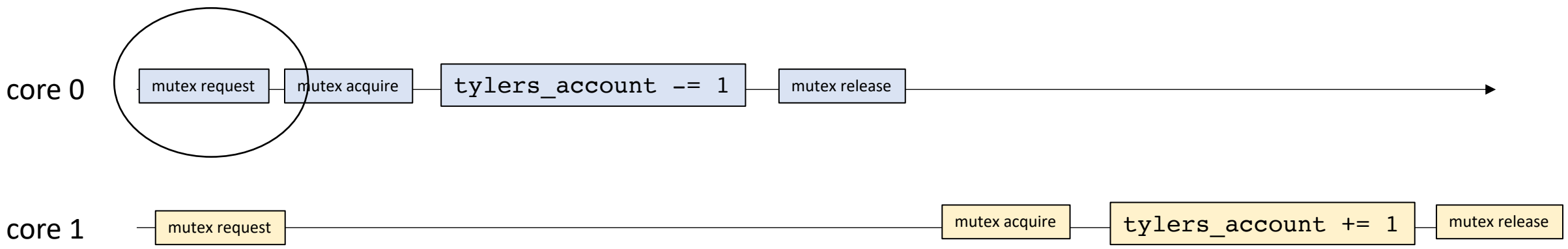
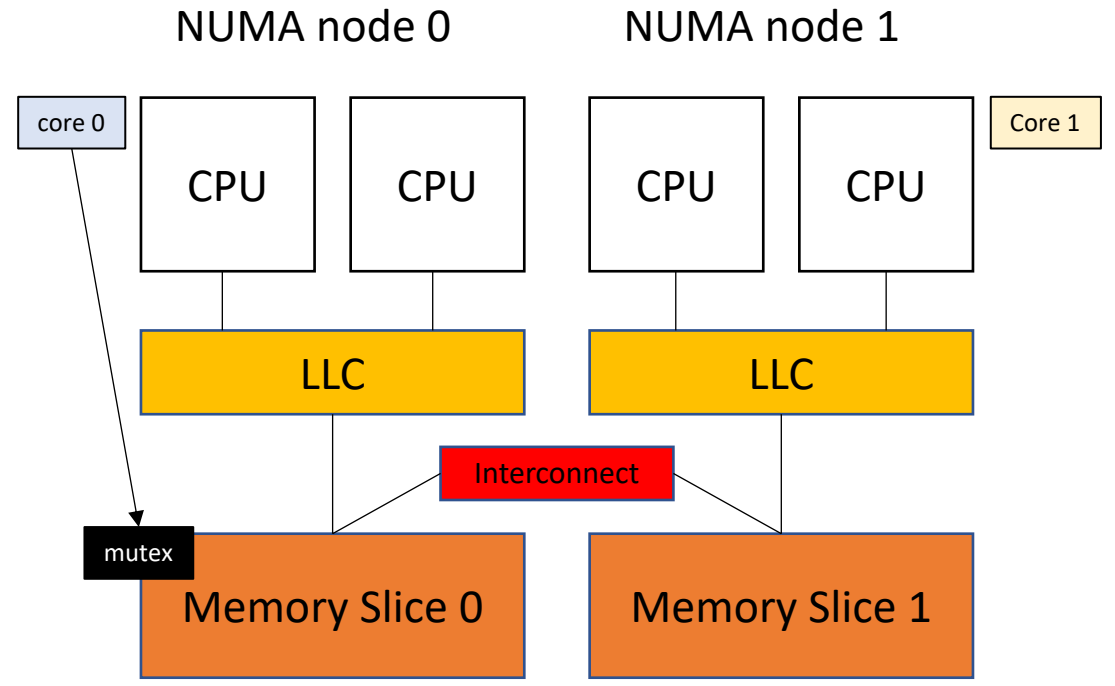


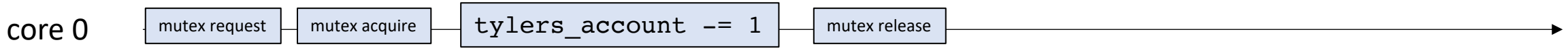
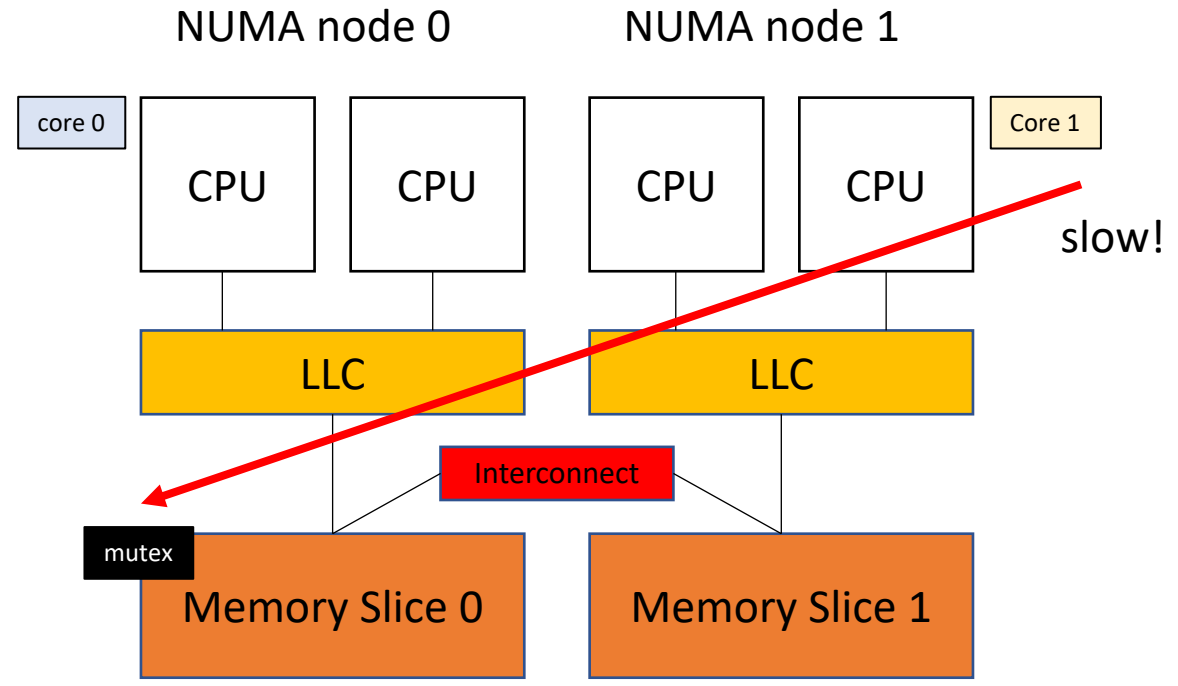
Optimization: Hierarchical locks

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



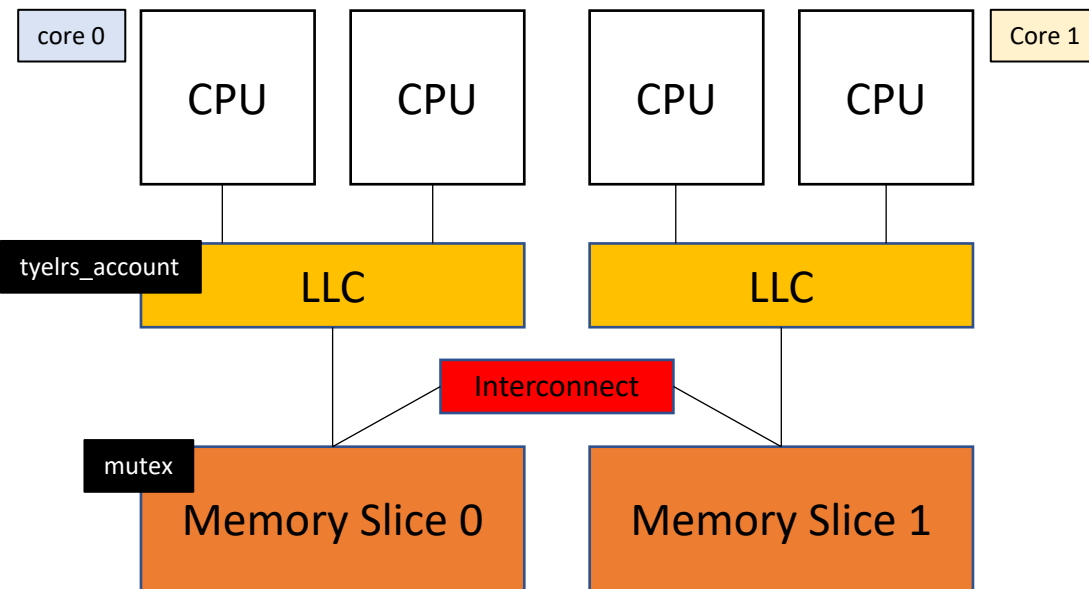




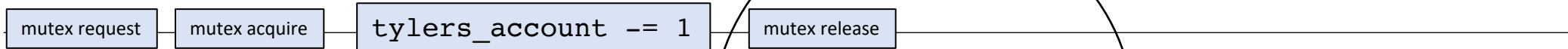


NUMA node 0

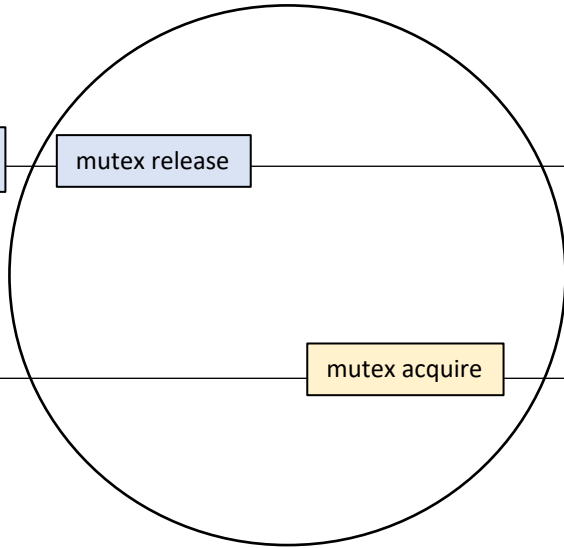
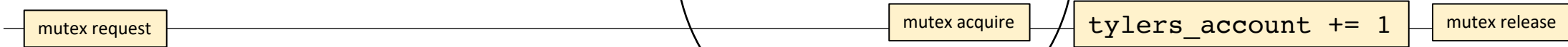
NUMA node 1



core 0

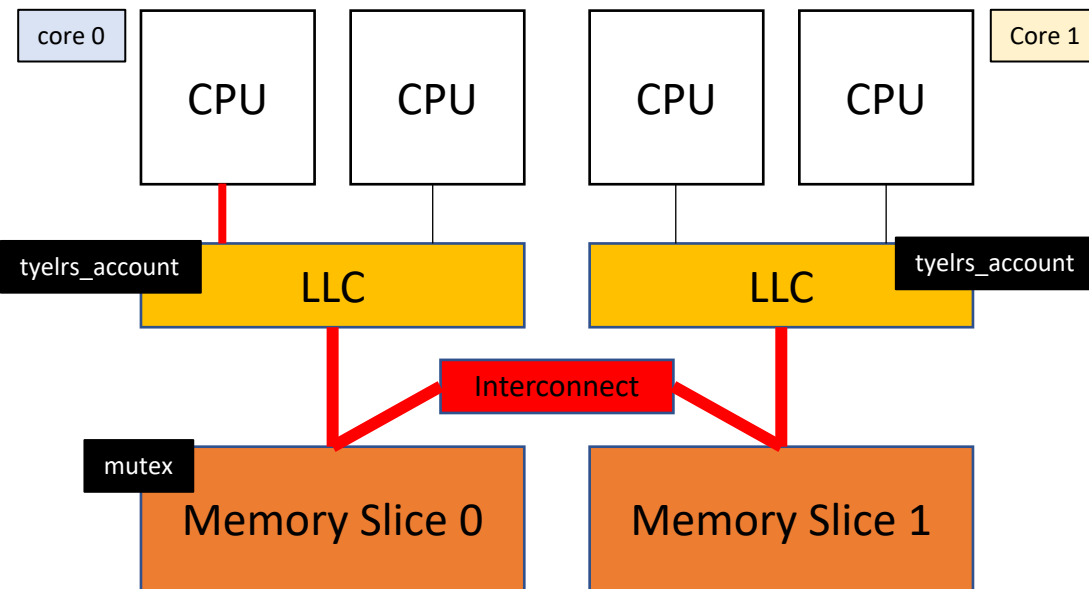


core 1

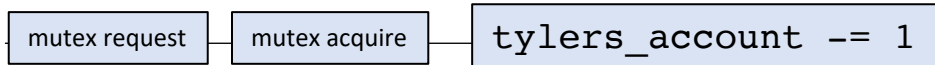


NUMA node 0

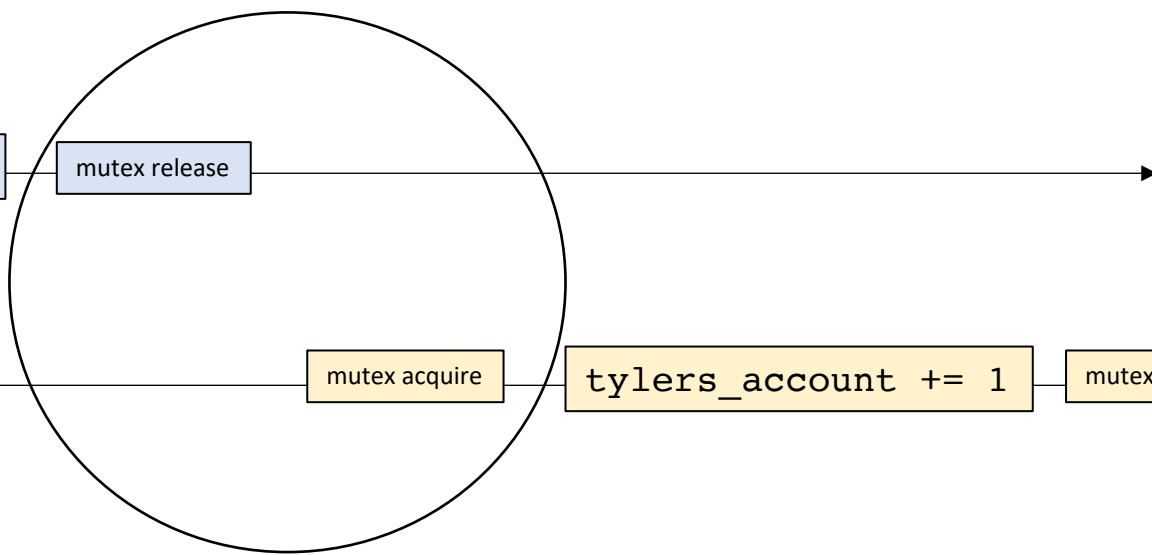
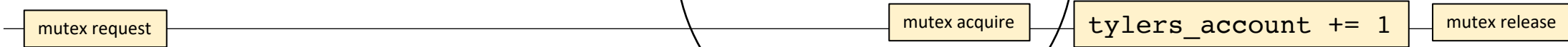
NUMA node 1

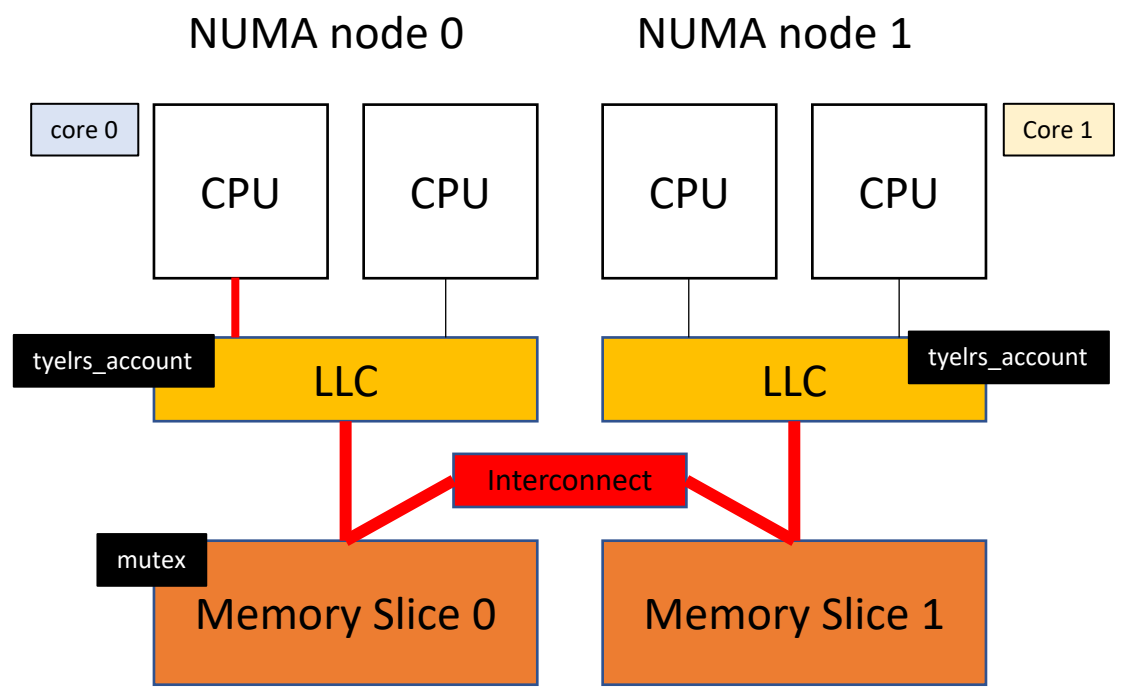


core 0

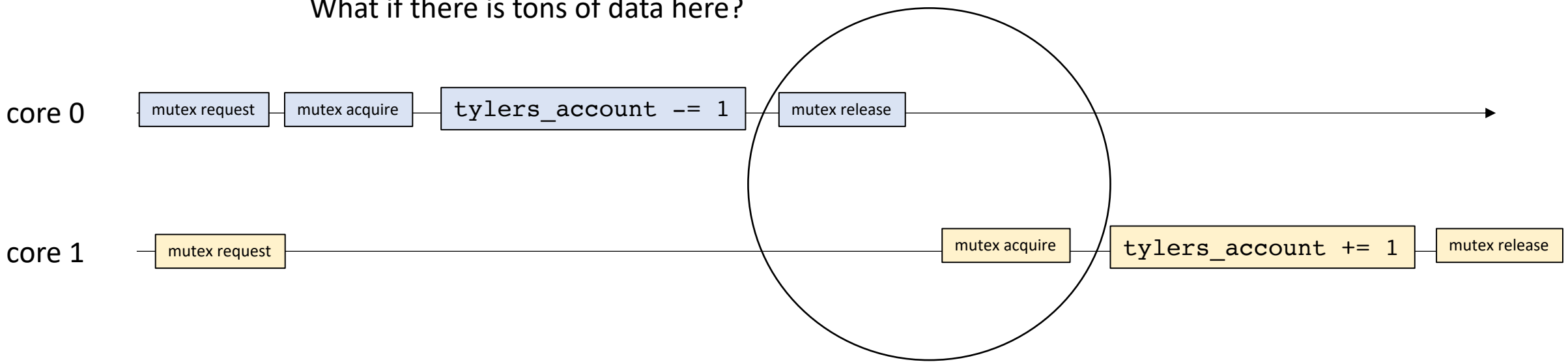


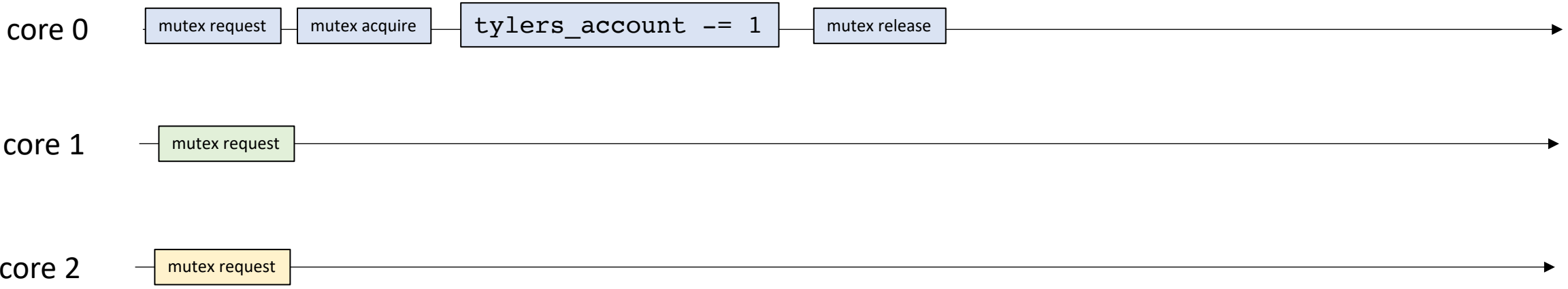
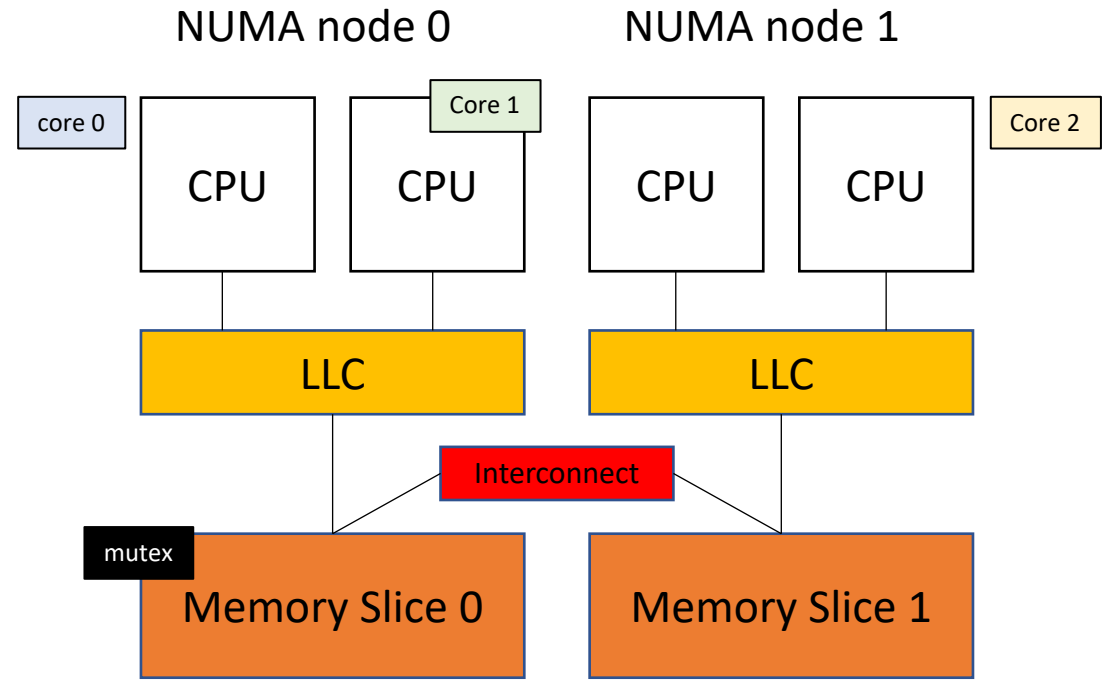
core 1

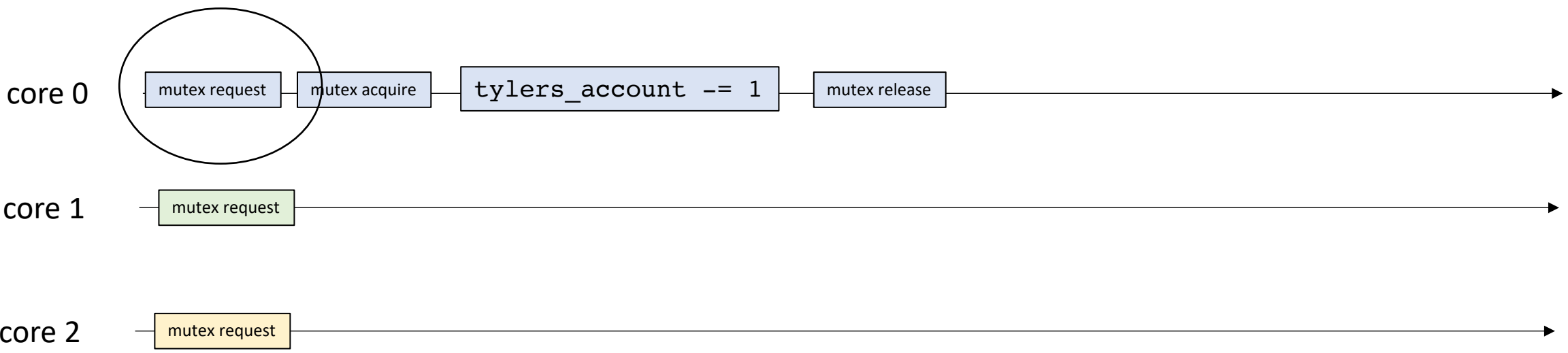
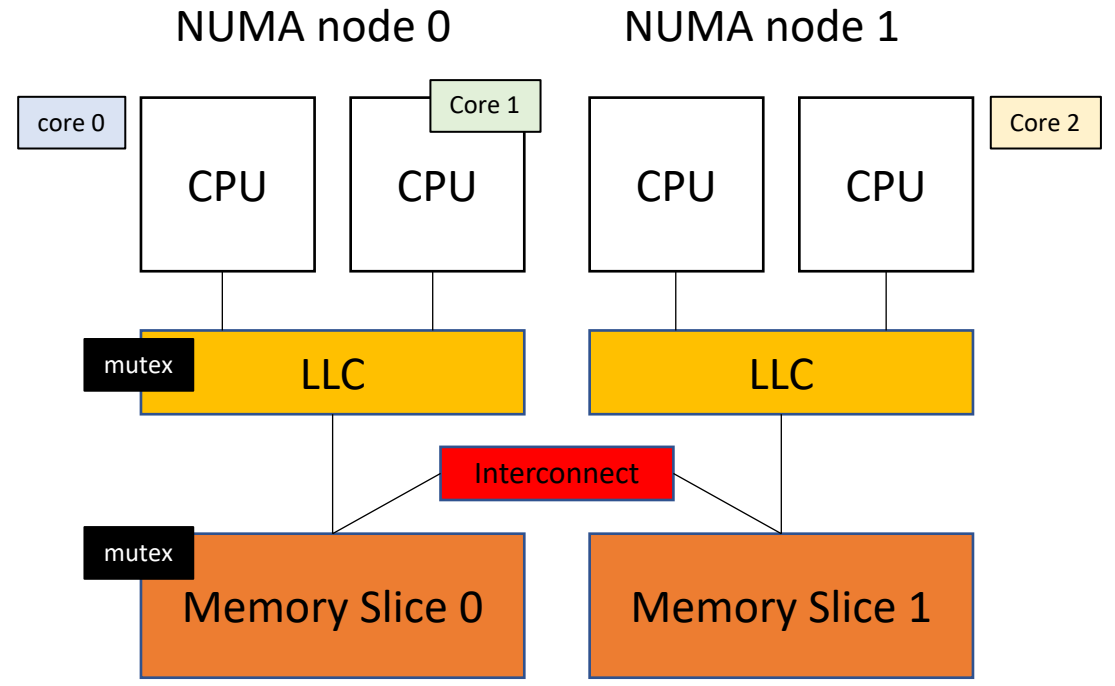




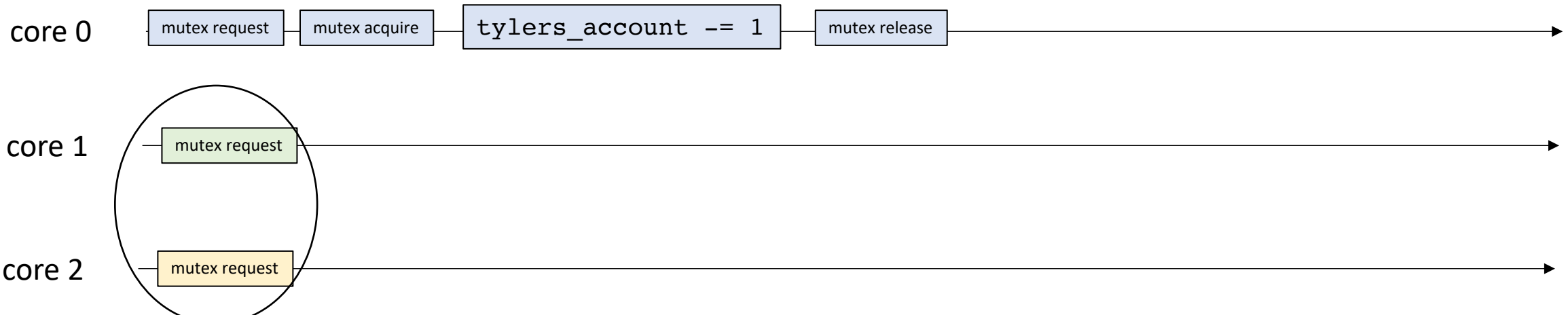
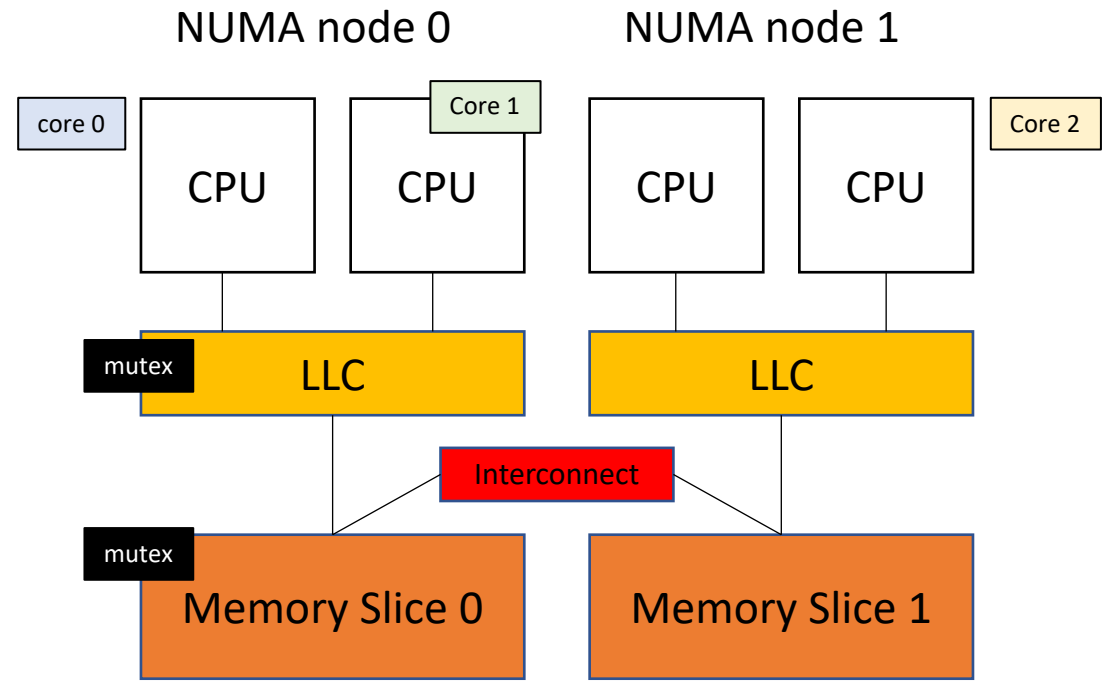
What if there is tons of data here?

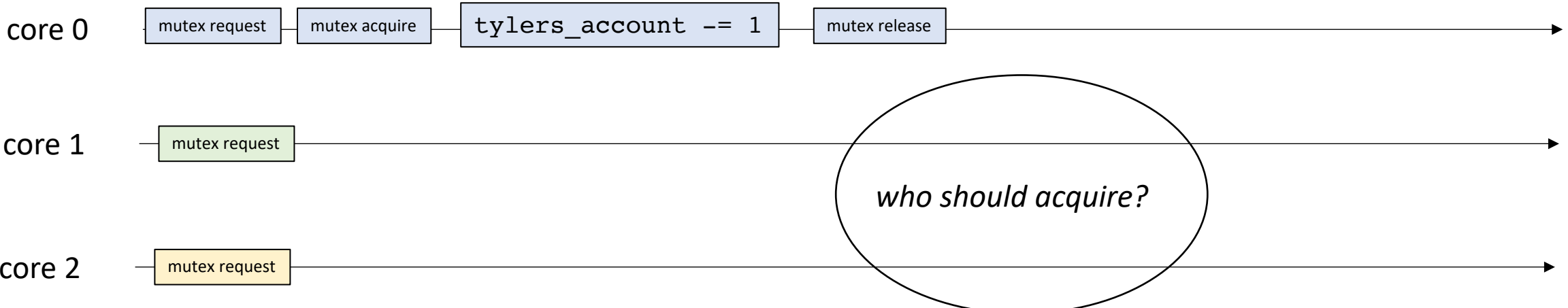
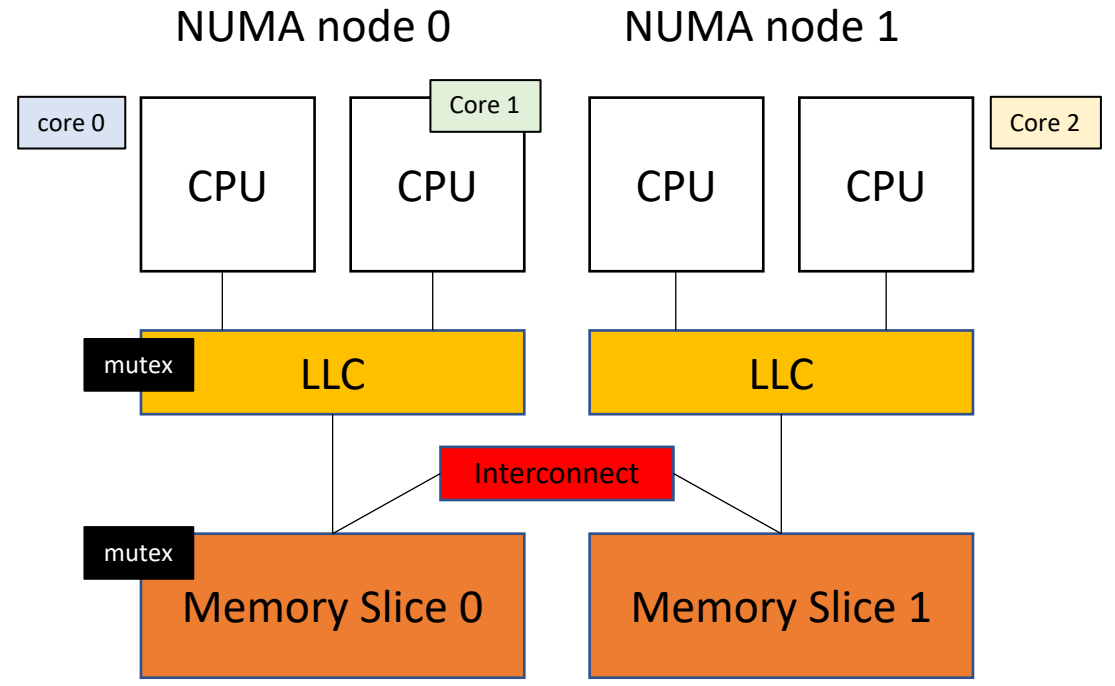




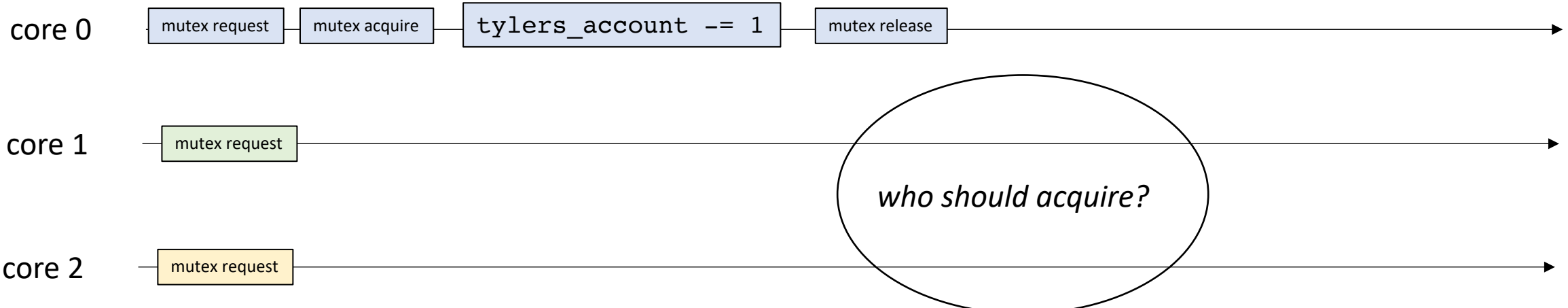
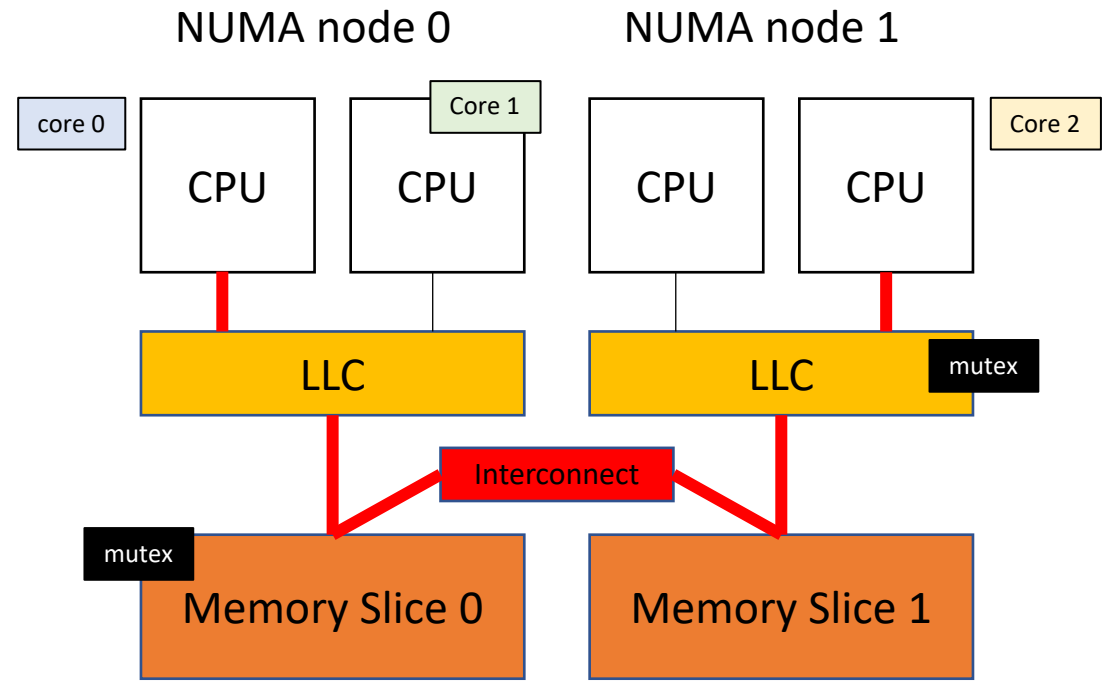


Ideally core 2 accesses the mutex less frequently than core 1

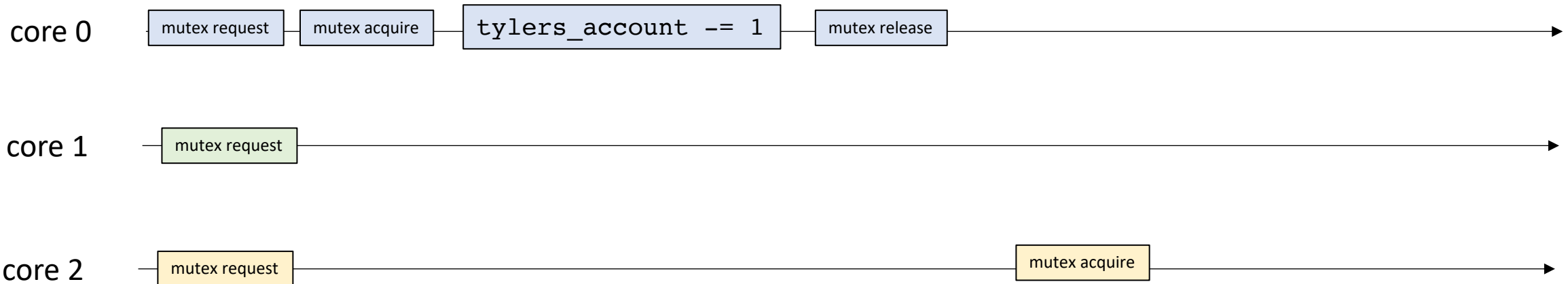
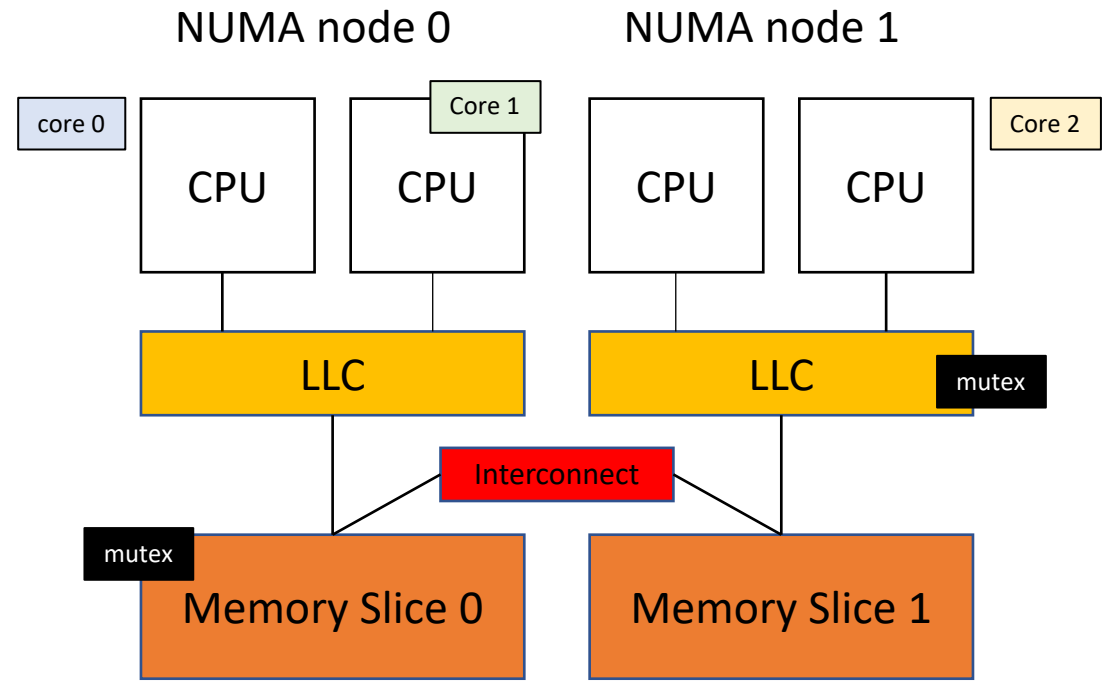




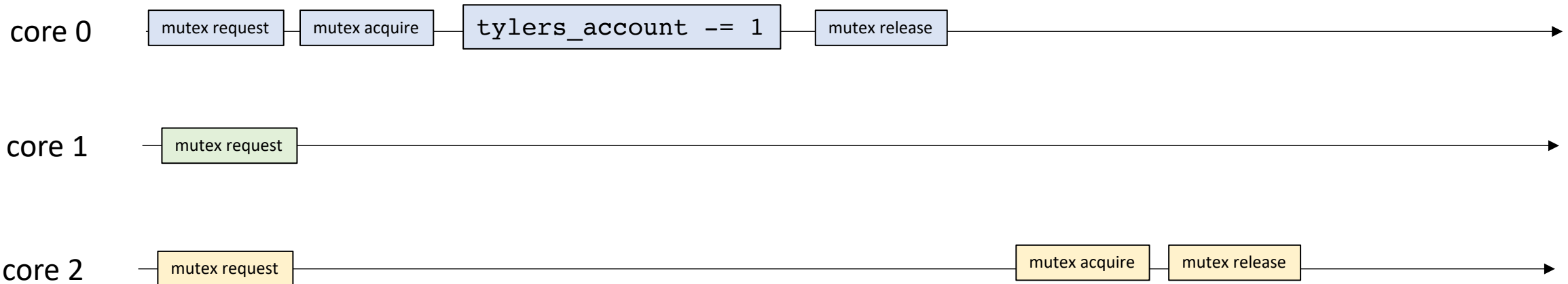
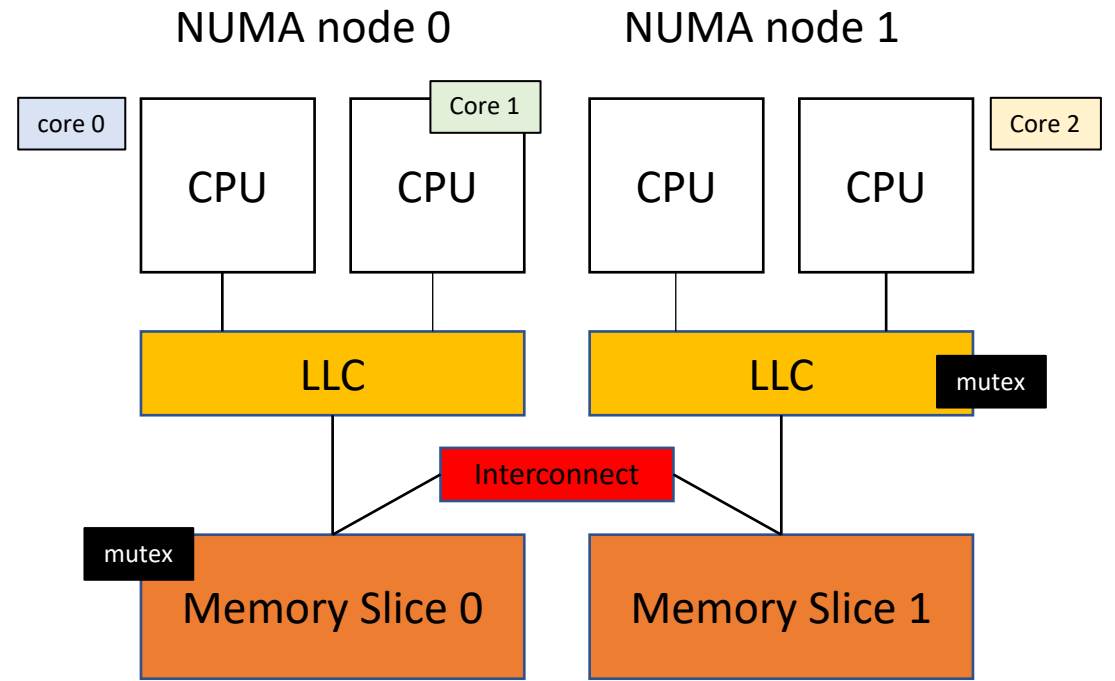
If core 2 acquires first communication must go through the interconnect



If core 2 acquires first communication must go through the interconnect

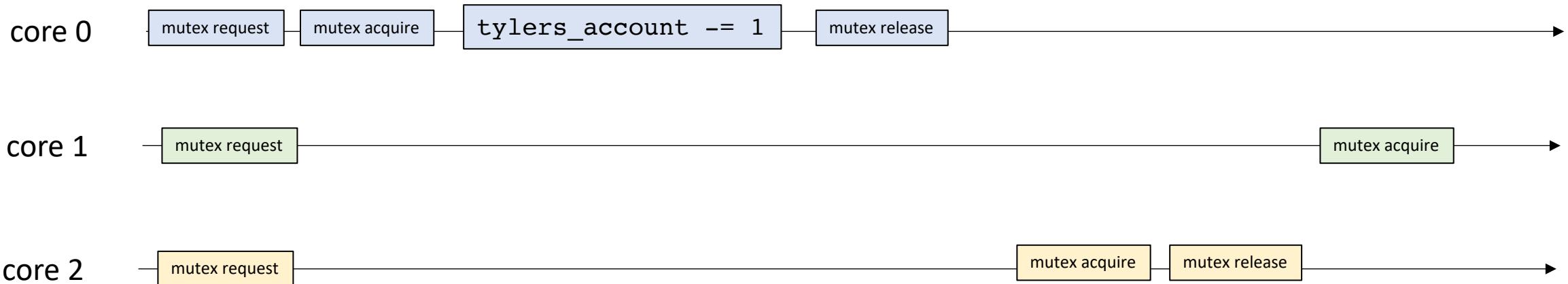
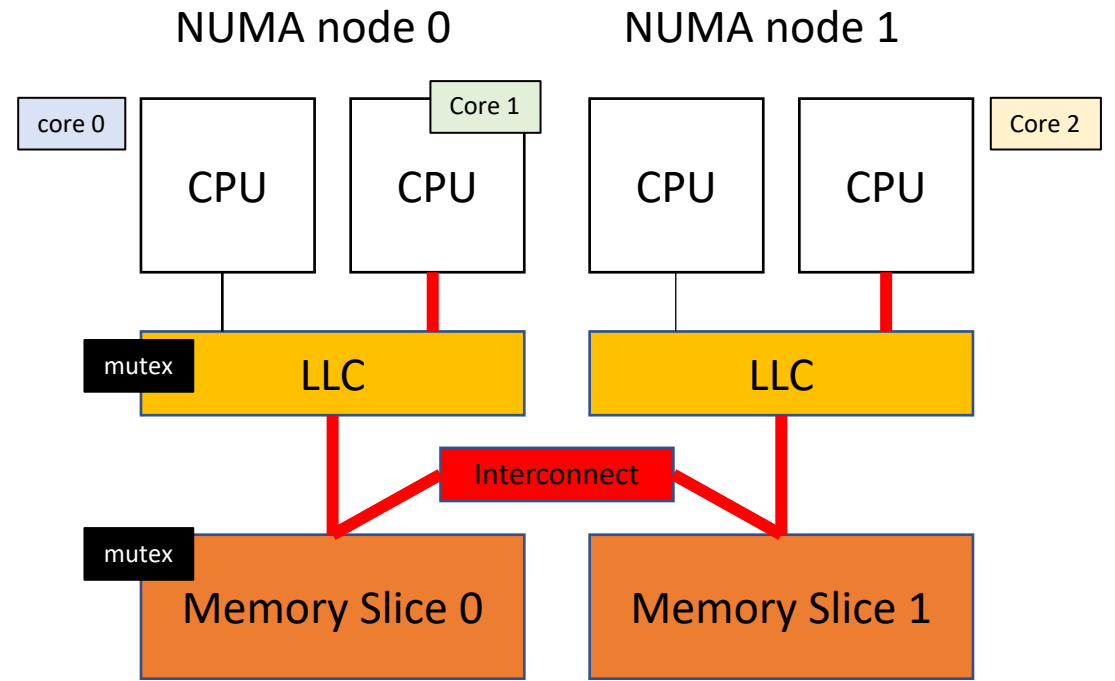


If core 2 acquires first communication must go through the interconnect



If core 2 acquires first communication must go through the interconnect

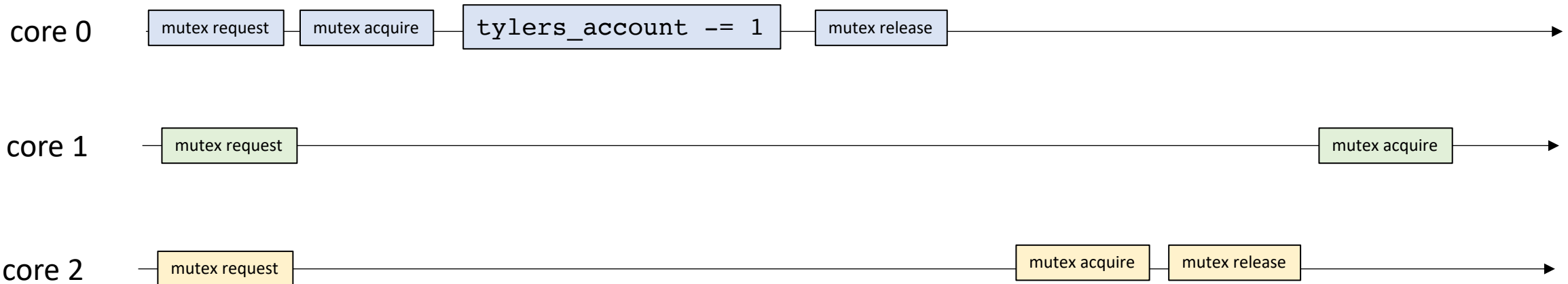
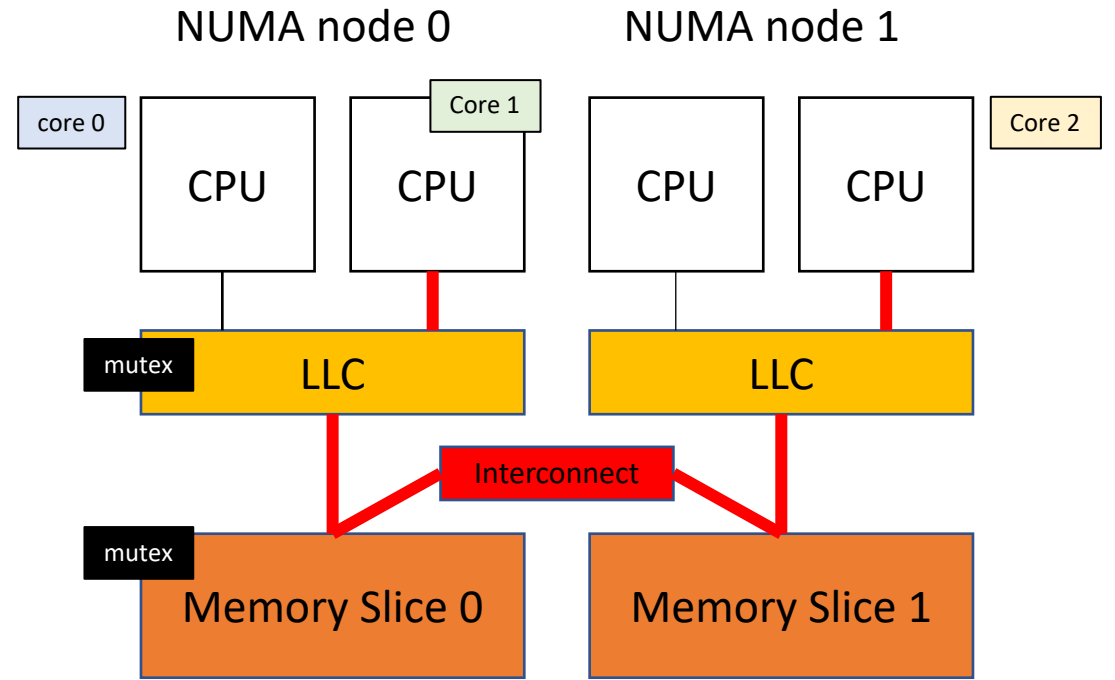
When core 1 finally acquires, it requires another expensive trip through the interconnect



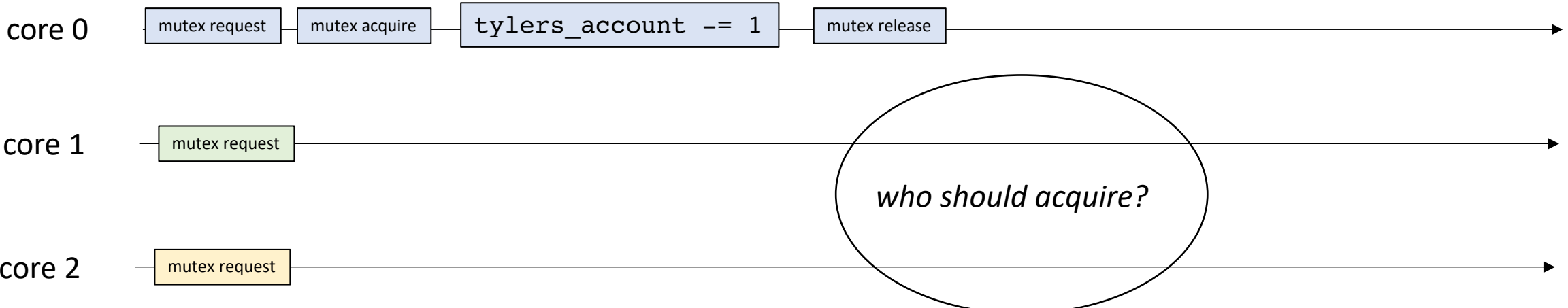
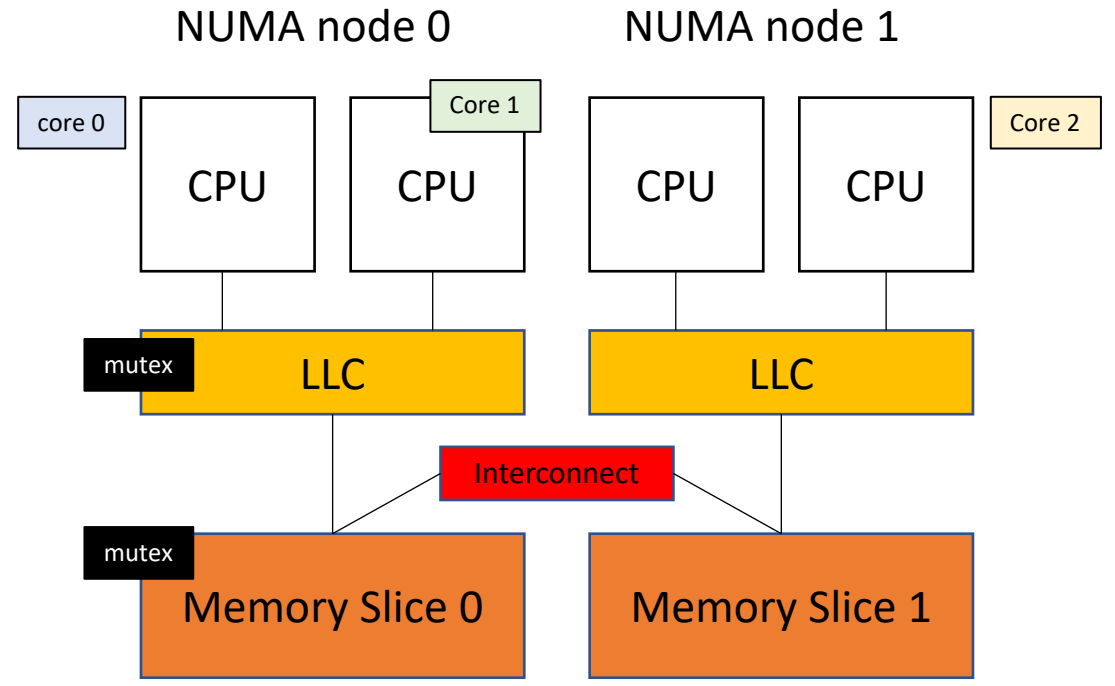
Two trips through the interconnect!!

If core 2 acquires first communication must go through the interconnect

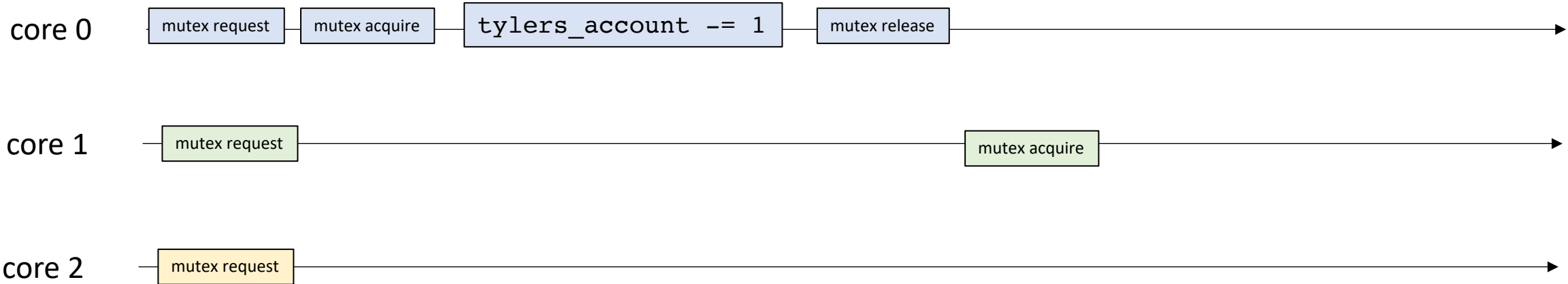
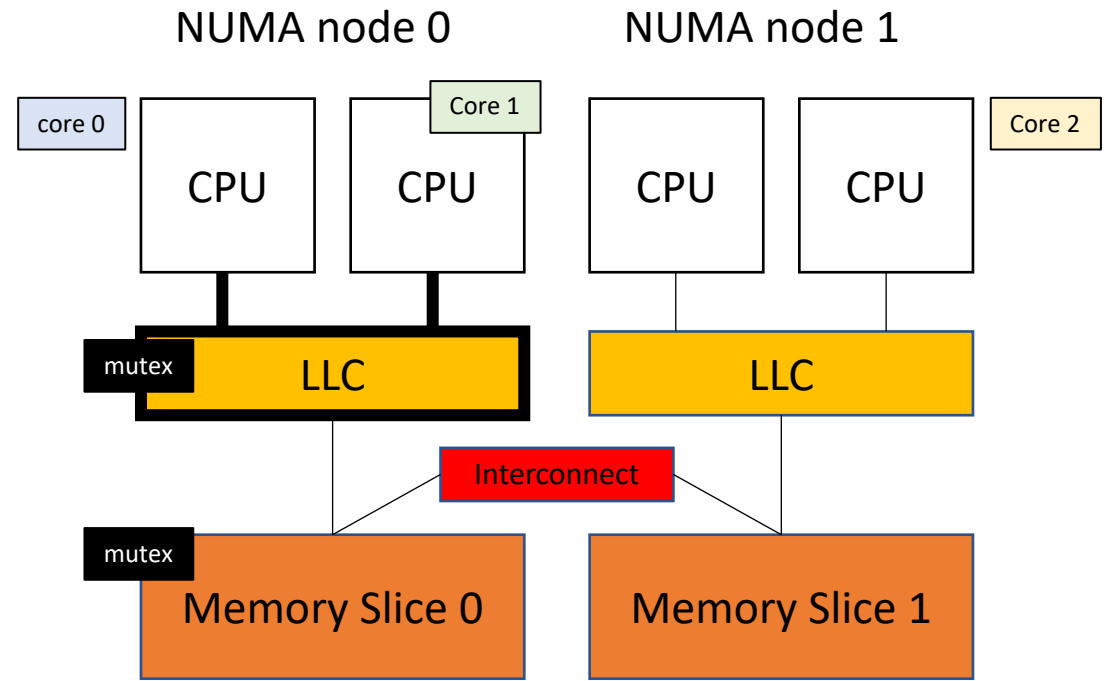
When core 1 finally acquires, it requires another expensive trip through the interconnect



Lets go back in time and make a different decision!

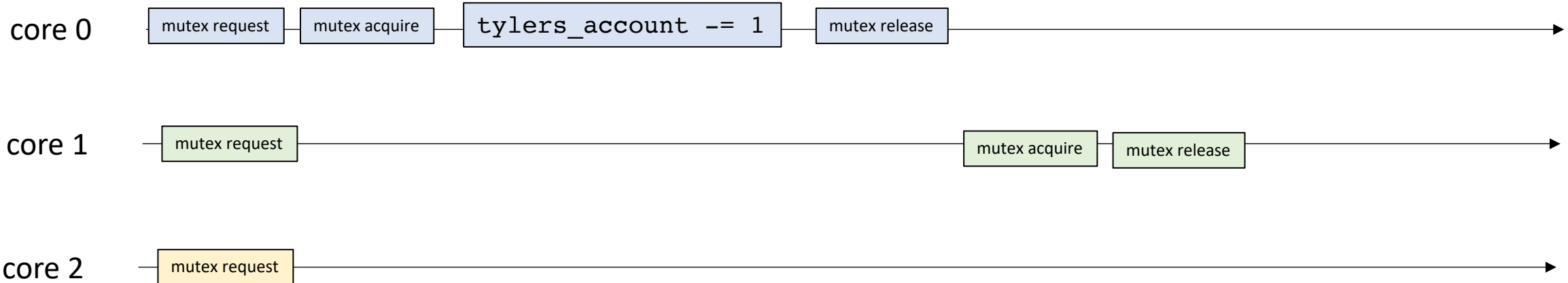
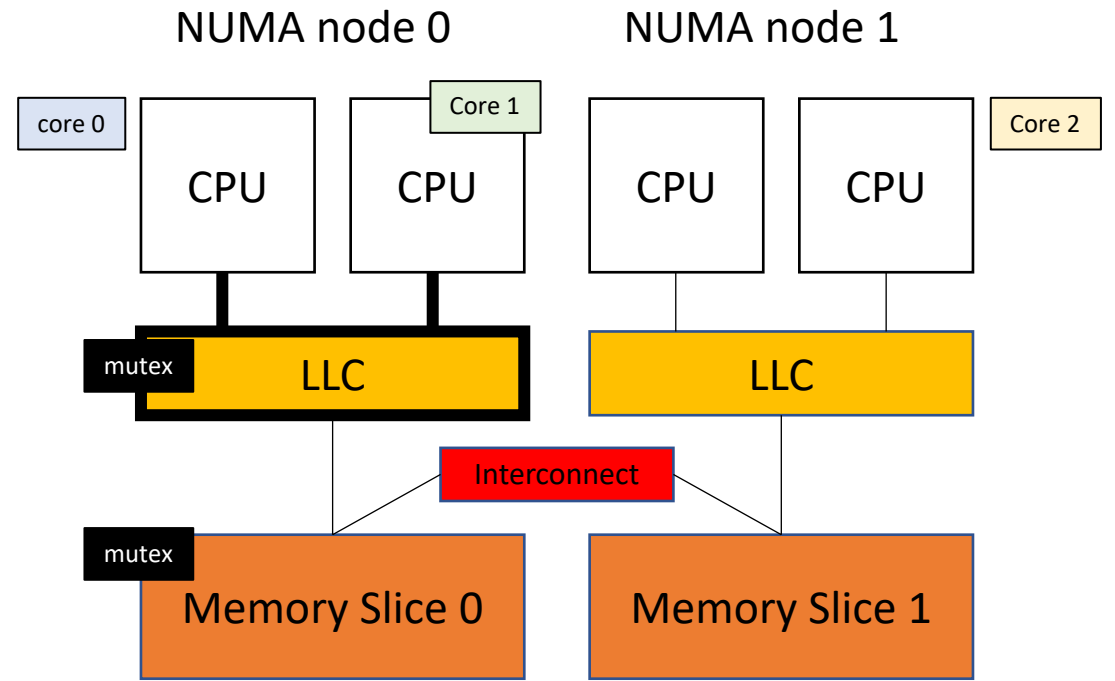


If core 1 acquires first communication can occur through the LLC of NUMA node 0



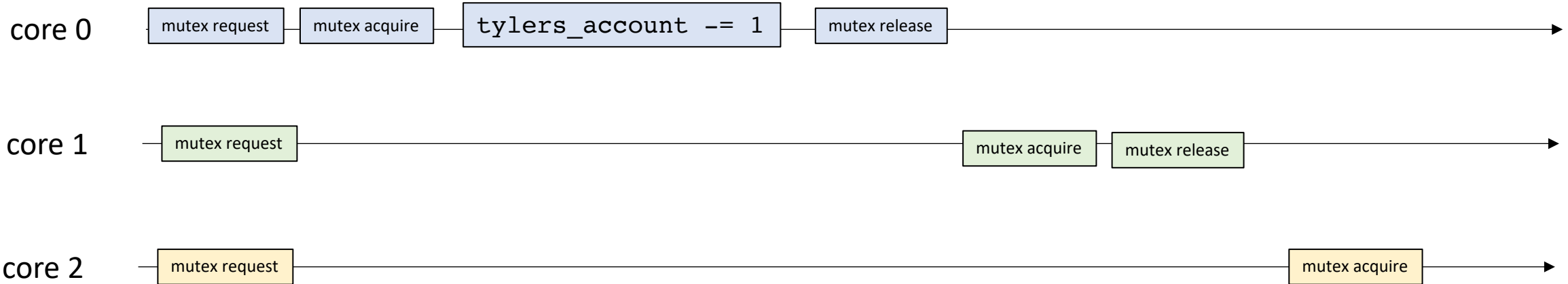
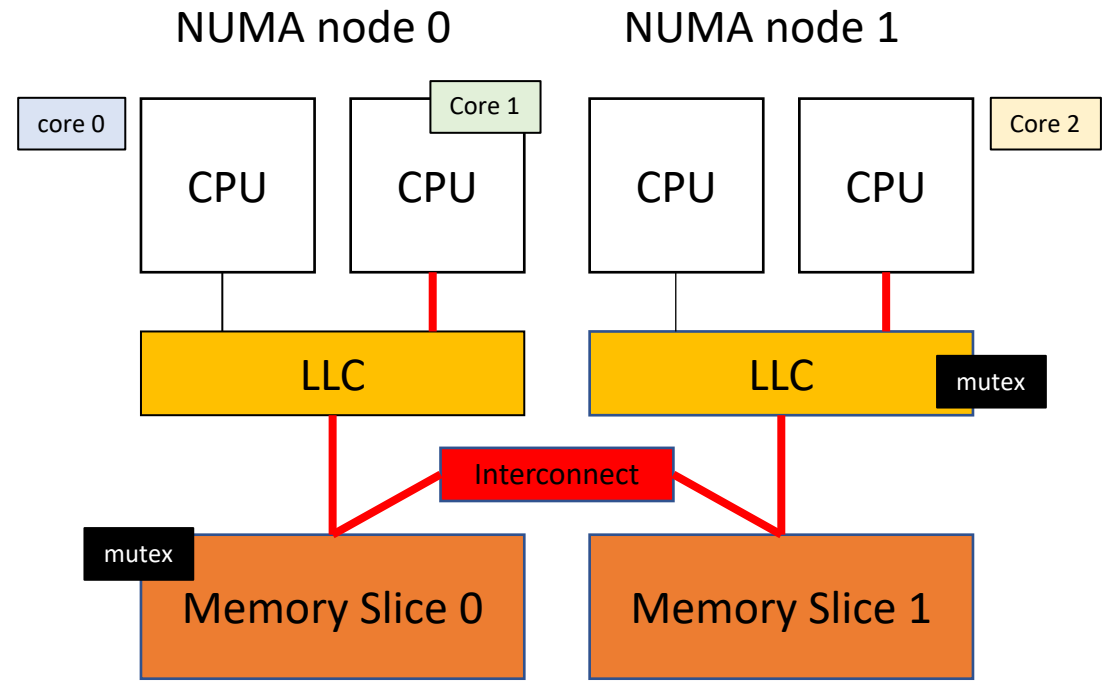
If core 1 acquires first communication can occur through the LLC of NUMA node 0

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

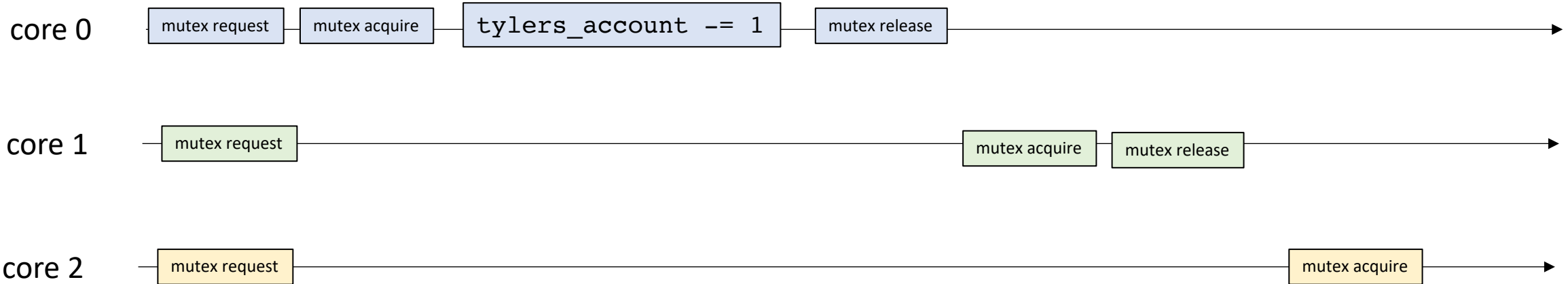
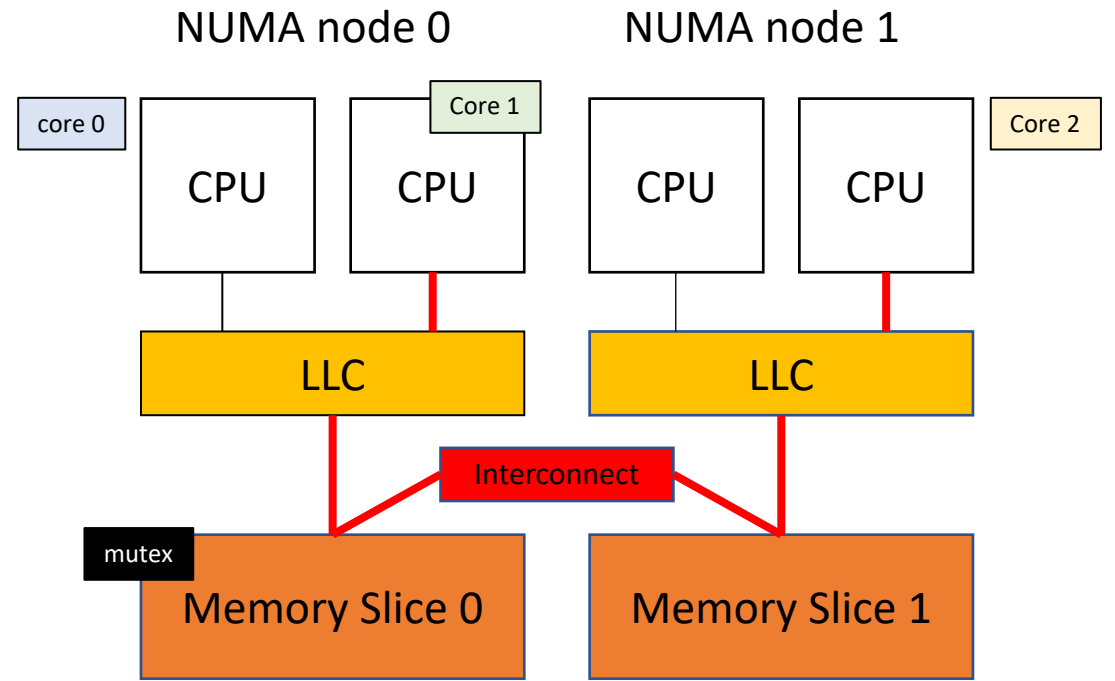
When core 2 finally acquires it requires an expensive trip through the interconnect



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?

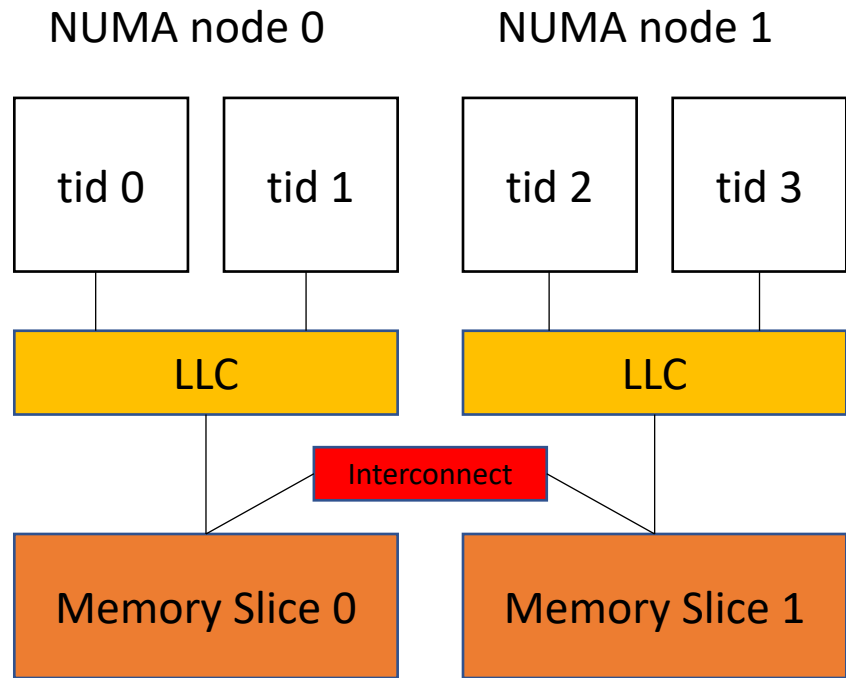
```
void lock(int thread_id) {
    int e = -1;
    int acquired = false;
    while (acquired == false) {
        acquired = atomic_compare_exchange_strong(&m_owner, &e, thread_id);
        e = -1;
    }
}
```

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):

$$\text{thread_id} / 2$$

$$\text{thread_id} / \text{NUM_NUMA_NODES}$$

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;
    }
}
```

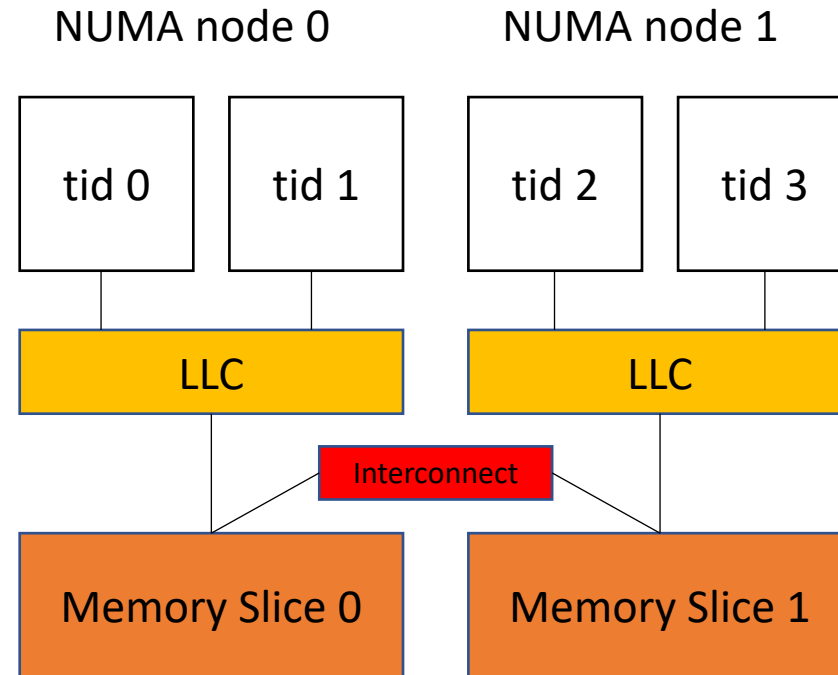
Starvation?

- Tune sleep times. You shouldn't starve the other nodes!
- Demo
- 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:

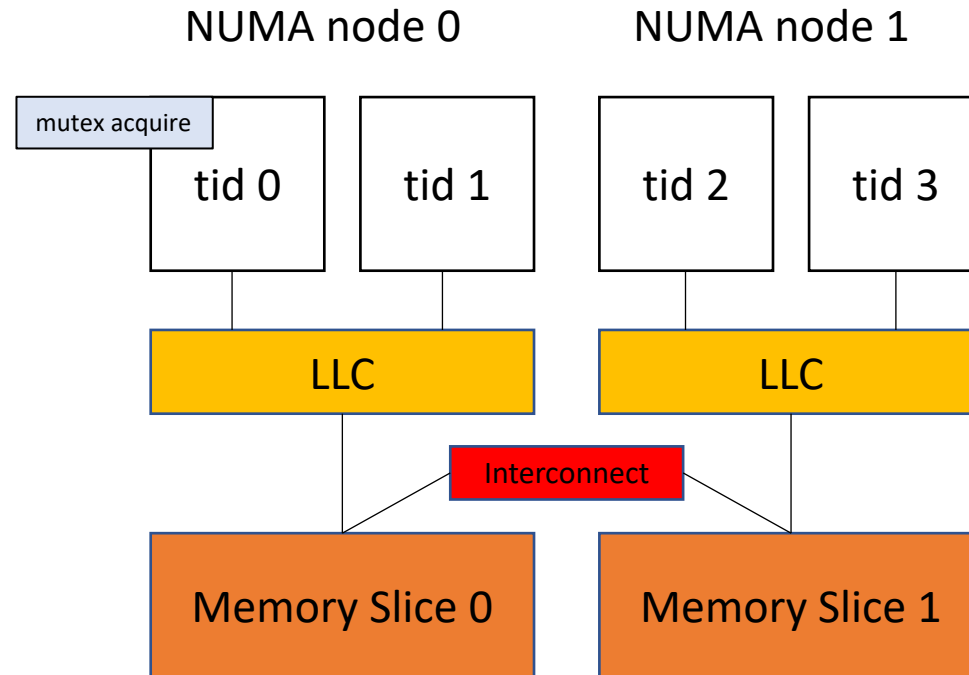
Mutex counter:
Local_Com: 0



Example:

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

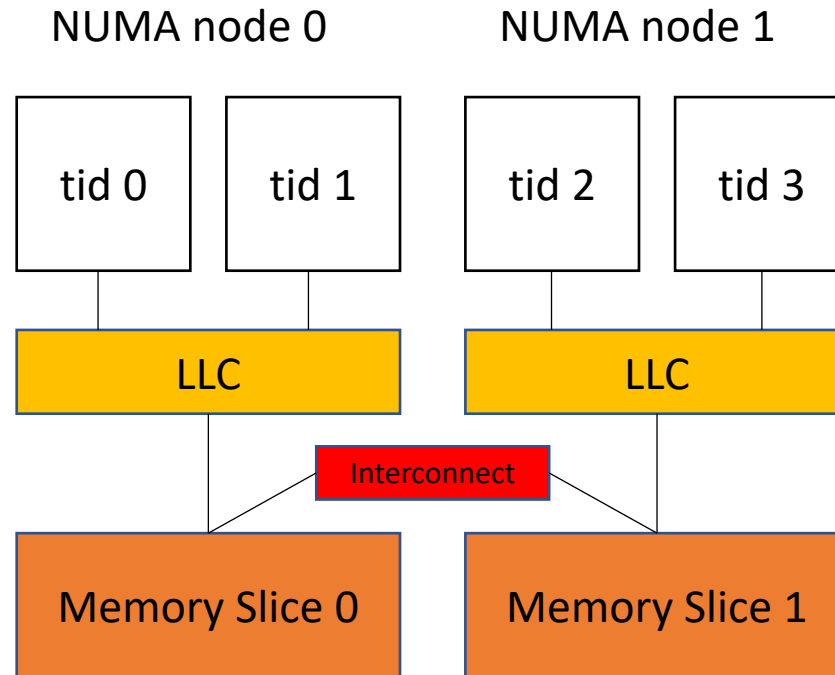
Mutex counter:
Local_Com: 1



Example:

tid 0:
tid 1:
tid 2:

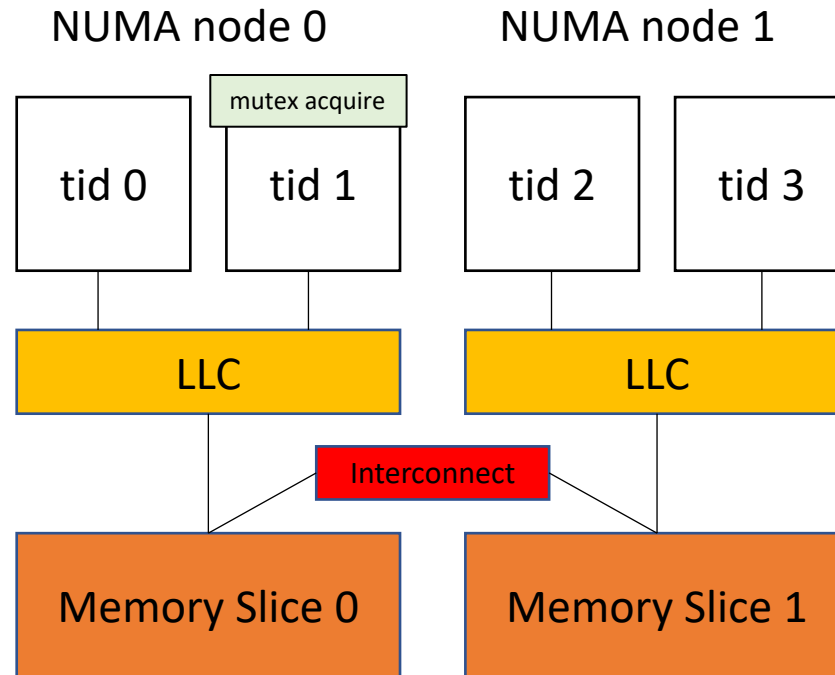
Mutex counter:
Local_Com: 1



Example:

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

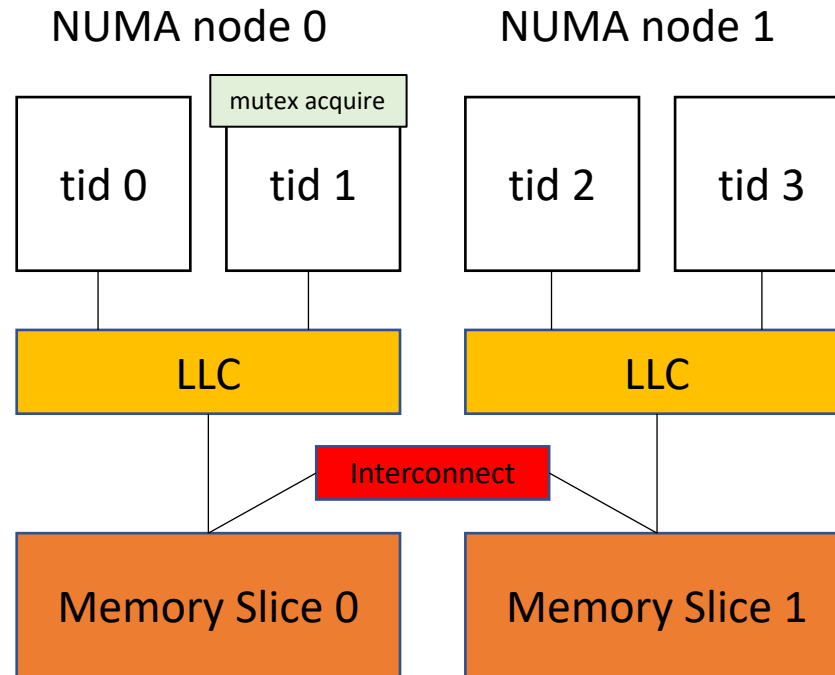
Mutex counter:
Local_Com: 2



Example:

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

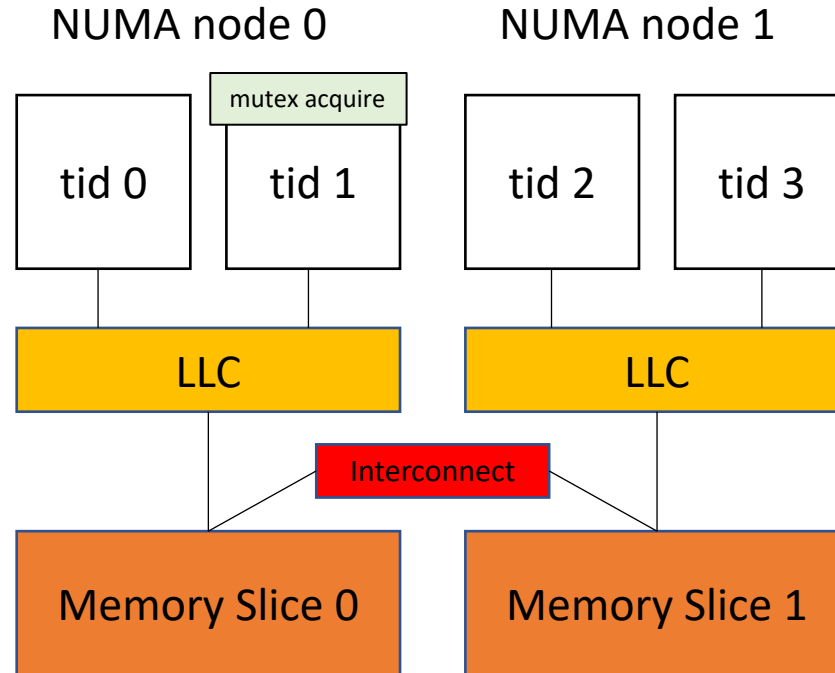
Mutex counter:
Local_Com: 2



Example:

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

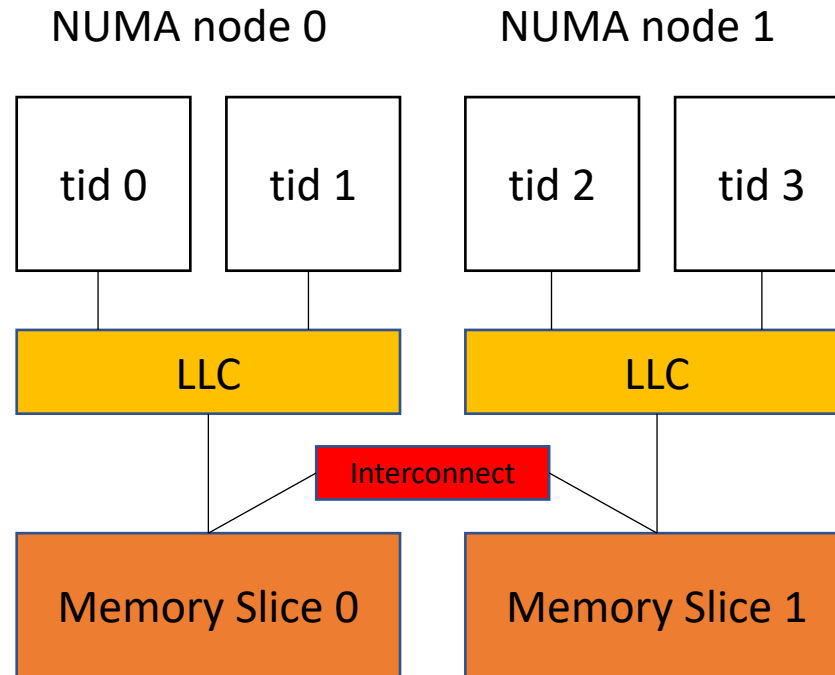
Mutex counter:
Local_Com: 2



Example:

tid 0:
tid 1:
tid 2:

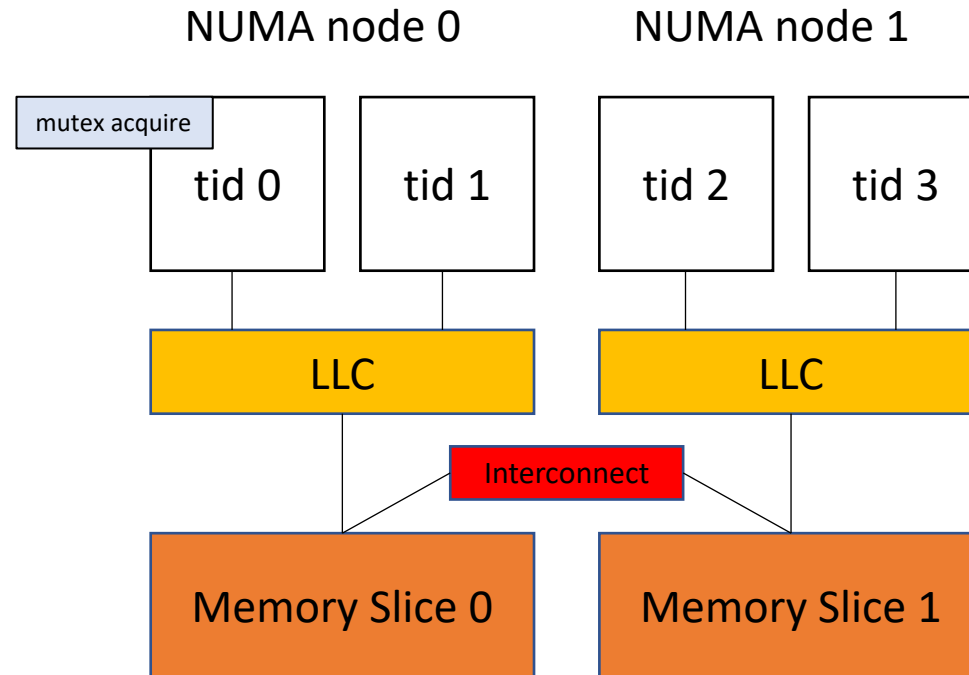
Mutex counter:
Local_Com: 1



Example:

tid 0: acquired
tid 1: sleep $1 \text{ ms} * \text{Local_Com} = 3 \text{ ms}$
tid 2: sleep 100 ms

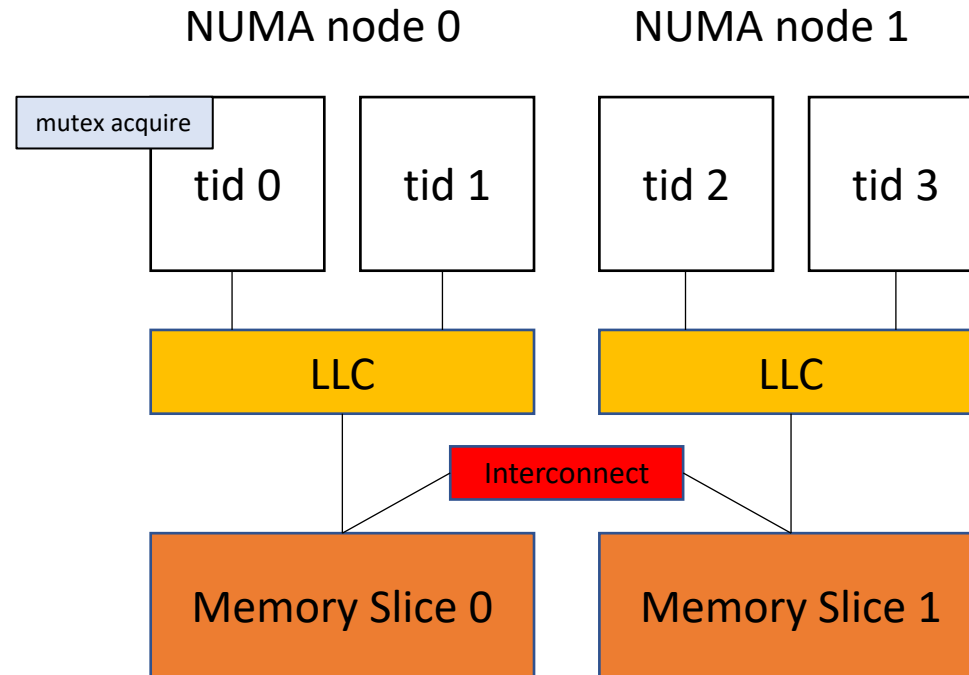
Mutex counter:
Local_Com: 3



Example:

tid 0: acquired
tid 1: sleep $1 \text{ ms} * \text{Local_Com} = 3 \text{ ms}$
tid 2: sleep 100 ms

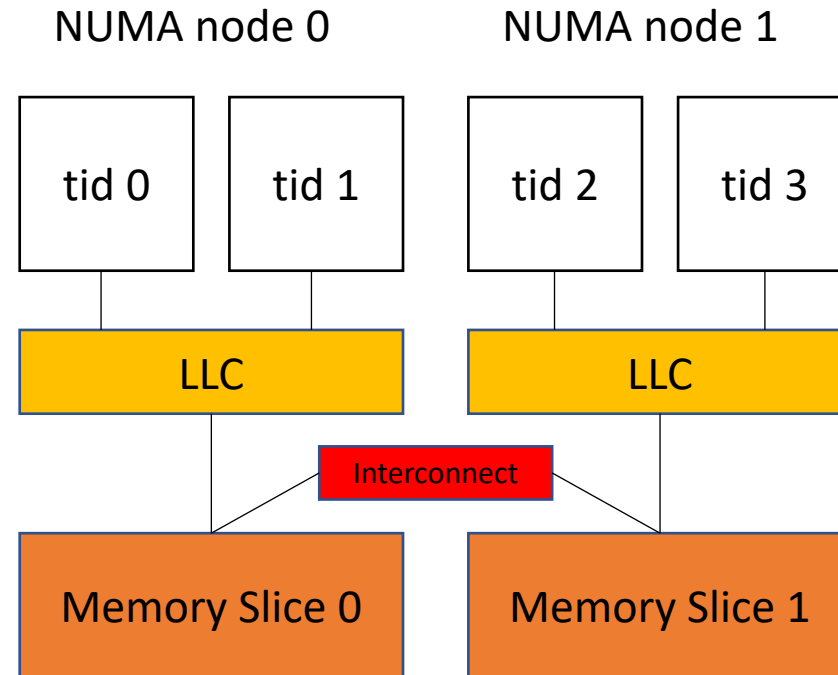
Mutex counter:
Local_Com: 3



Example:

tid 0:
tid 1:
tid 2:

Mutex counter:
Local_Com: 3

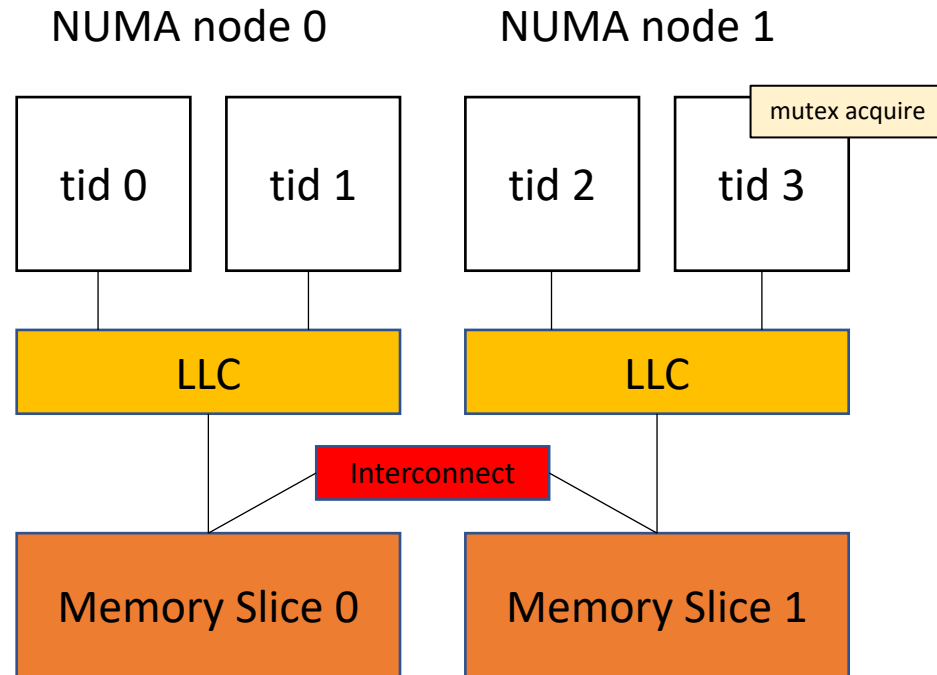


Example:

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

Perspective

- Keep in mind that the book was published nearly 10 years ago
- Synchronization costs have changed!

My experience:

Impact of lock implementation had over 100x impact on Fermi Nvidia GPUs (circa 2010)

Impact of lock implementation had less than 2x on Maxwell Nvidia GPUs (circa 2016)

These days many devices have efficient coherence protocols. The optimizations we discussed in class will give you good performance on most of today's devices.

BUT: Maybe history will repeat itself with RISC-V chips?! Princeton DECADES chip was very sensitive...

Lecture Schedule

- Optimizations
 - Backoff
 - Hierarchical aware locks
 - Further reading
- RW locks:
 - Implementation
 - C++
- Checking for data conflicts

Lecture Schedule

- Optimizations
 - Backoff
 - Hierarchical aware locks
 - Further reading
- **RW locks:**
 - Implementation
 - C++
- Checking for data conflicts

Reader-Write Mutex

Global variable: `int tylers_account`

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

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

Reader-Write 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-Write 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;  
}
```

which of these operations can safely be executed concurrently?

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

Different actors accessing it concurrently

Credit monitors

Accountants

Personal

Reader-Write 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-Write Mutex

Global variable: `int tylers_account`

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

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

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

Reader-Write 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-Write 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;  
}
```

writer = false
num_readers = 0

```
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 = 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;  
}
```

writer = true
num_readers = 0

```
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 = true
num_readers = 0

```
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();  
}
```

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;  
}
```

writer = False
num_readers = 0

```
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();  
}
```


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;  
}
```

writer = False
num_readers = 1

```
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();  
}
```

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;  
}
```

writer = False
num_readers = 2

```
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 = 2

```
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();  
}
```

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?

writer = False
num_readers = 1

```
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 = 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;  
}
```

writer = False
num_readers = 0

```
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();  
}
```

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
```

Lecture Schedule

- Optimizations
 - Backoff
 - Hierarchical aware locks
 - Further reading
- RW locks:
 - Implementation
 - C++
- Checking for data conflicts

Lecture Schedule

- Optimizations
 - Backoff
 - Hierarchical aware locks
 - Further reading
- RW locks:
 - Implementation
 - C++
- **Checking for data conflicts**

Data conflicts

- Data conflicts are undefined
 - Compiler can do crazy things
 - rare interleavings cause bugs that are extremely rare
- Your code should use mutexes to avoid data conflicts!
- What happens when you don't?

Horrible data conflicts in the real world

Therac 25: a radiation therapy machine

- Between 1987 and 1989 a software bug caused 6 cases where radiation was massively overdosed
- Patients were seriously injured and even died.
- Bug was root caused to be a data conflict.
- <https://en.wikipedia.org/wiki/Therac-25>

Horrible data conflicts in the real world

2003 NE power blackout

- second largest power outage in history: 55 million people were effected
- NYC was without power for 2 days, estimated 100 deaths
- Root cause was a data conflict
- https://en.wikipedia.org/wiki/Northeast_blackout_of_2003

But checking for data conflicts is hard...

- Tools are here to help (Professor Flanagan is famous in this area)

How do they work?

- Two approaches
- **Happens-before:** build a partial order of mutex lock/unlocks. Any memory access that can't be ordered in this partial order is a conflict.
- **Lockset:** Every shared memory location has is associated with a set of locks. Refine the lockset for every access and evaluate the final result.

Dynamic Analysis

- Thread sanitizer:
 - a compiler pass built into Clang
 - About 10x overhead when you run the program
 - Identifies data conflicts
 - deadlocks
- Examples

Static Analysis

- Facebook Infer:
 - Statically checks for many issues (memory safety, assertions)
 - Can check for races in concurrent classes
 - Main support is for Java, although they claim support for C++

Current state of data conflicts

- A recent tool:
 - Checks for C++ races
 - Scales to large programs
- Reports:
 - Chrome has 6 unresolved data-conflicts
 - Firefox has 52 unresolved data-conflicts
- Difficult to fix! 6.7 million lines of code in Chrome



Summary

- Avoid data conflicts! They can cause serious bugs that trigger very very rarely. (heisenbugs).
 - Better to use too many mutexes than not enough
- Use tools to help you!
 - Infer can help with Java
 - Thread sanitizer helps with C++

End of lecture

- End of mutex module
 - My favorite question: “is mutex an open question”
 - If you have any unanswered questions about mutexes, please ask in Piazza or Canvas!
- Next week: starting concurrent data structures:
 - Lectures will follow the same pattern:
 - First we do specification
 - Next we do implementations
 - Finally we do a use-case: Load balancing

Good luck on the homework!!
You got this!

