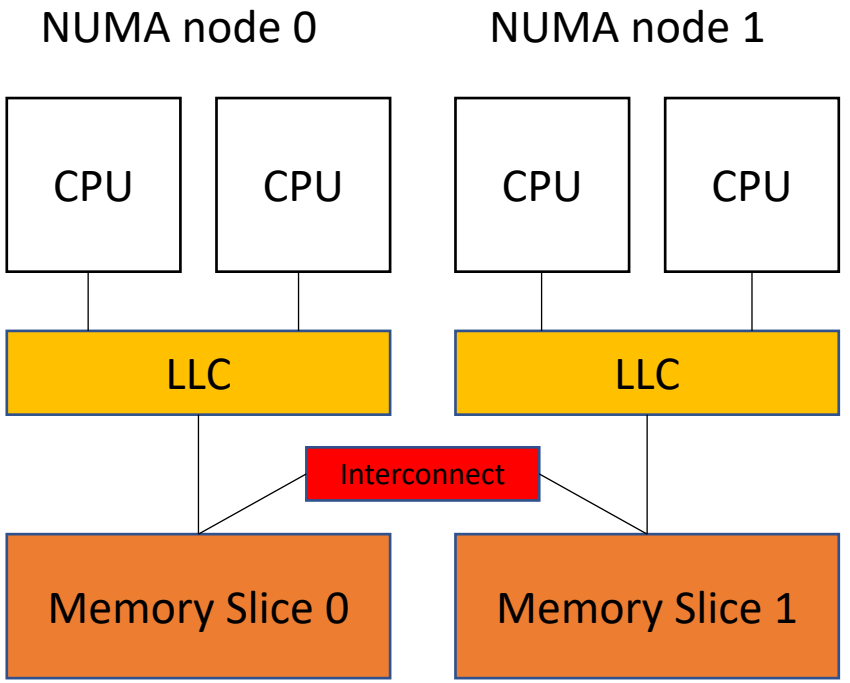
CSE113: Parallel Programming

Feb 6, 2023

- **Topics**:
  - Specialized Mutexes
    - Reader Writer Mutexes
    - NUMA aware mutexes



Announcements

- Homework 2 is out
  - Due on Thursday, with 4 days of free late days
  - You will have everything you need after today for the entire assignment
  - Hopefully you have already started
  - Let us know your throughput numbers and if you are not passing tests when you think you should be
- Ask for help in office hours or piazza if needed

Announcements

- Working on grading HW 1, expect scores within a week

- Midterm is coming next week
  - Take home
  - Released on Monday
  - Due on Friday
  - Open book, open internet, open notes
    - Do not discuss test with each other
    - Do not google for specific answers
    - Do not use chatGPT

Announcements

- The midterm and HW 3 will be out at the same time. Please budget your time accordingly.

- This is the last day on module 2
  - We will move to module 3: “concurrent data structures” next lecture

Previous quiz

Previous quiz

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

- ◯ True
- ◯ False

Previous quiz

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

Previous quiz

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

Previous quiz

Why is the compare-and-swap operation required after the relaxed peeking sees that the mutex is available?

```
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);
  }
}
```

Review

An optimized mutex

```
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);
  }
}
```

A fair RMW mutex

```
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;
};
```

Could you do relaxed peeking in this mutex?

Could you yield in this mutex?

Try lock

- API that allows 1-shot attempt at acquiring the mutex

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

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

Example for UI refresh where lock should be taken once every screen refresh: every 16 ms.

New material for the lecture

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
```

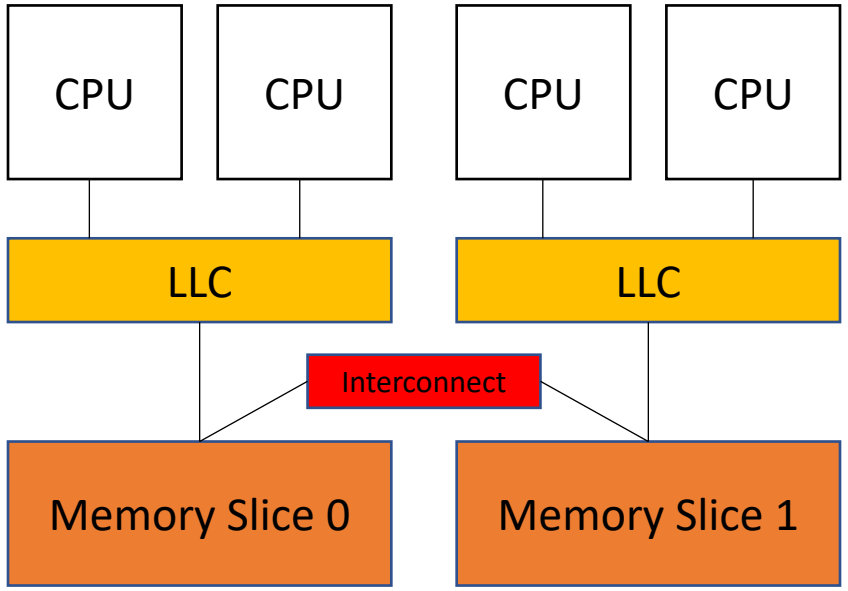
Optimization: Hierarchical locks

Optimization: Hierarchical locks

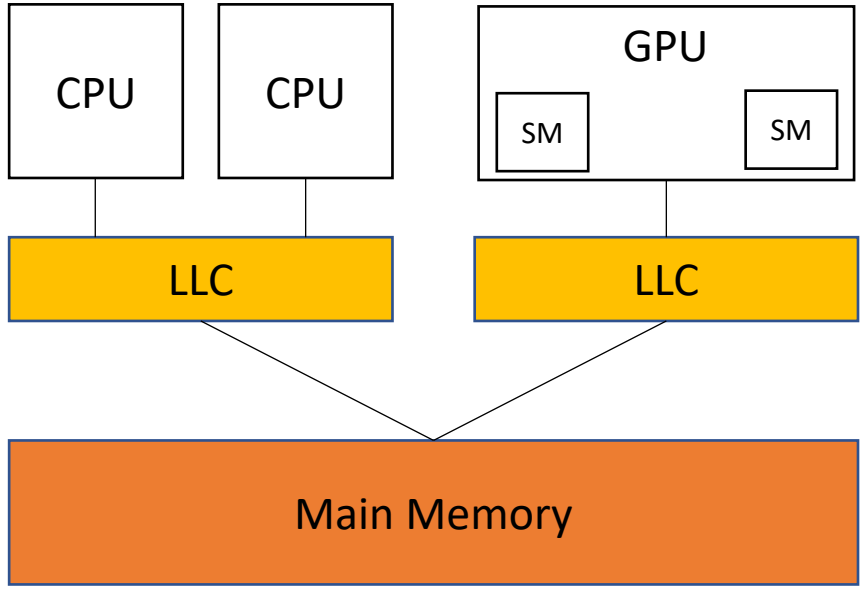
- NUMA (non-uniform memory access) systems
- heterogeneous systems (CPU GPU)

Discrete GPUs communicate through PCIE

For example: Large server nodes

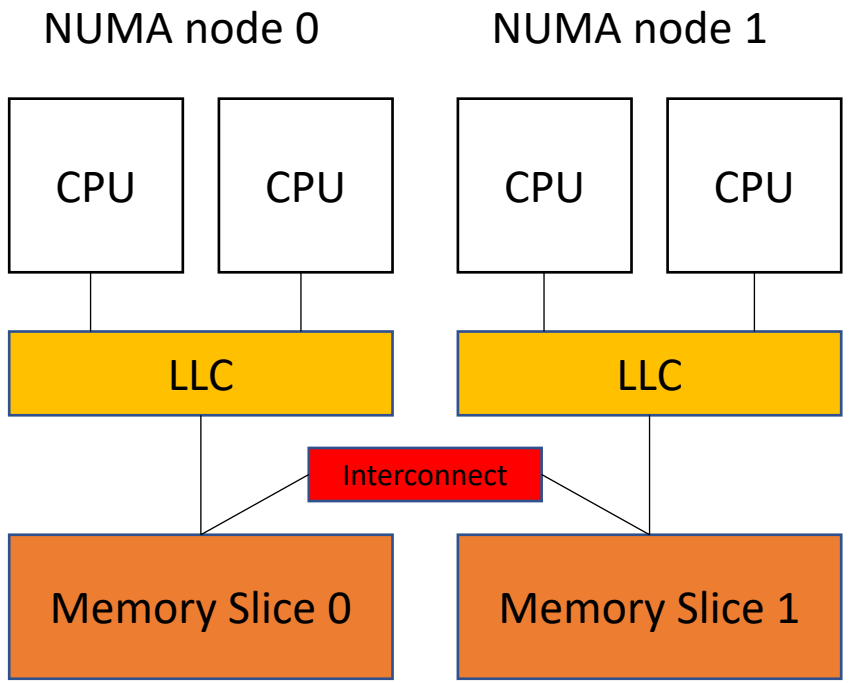


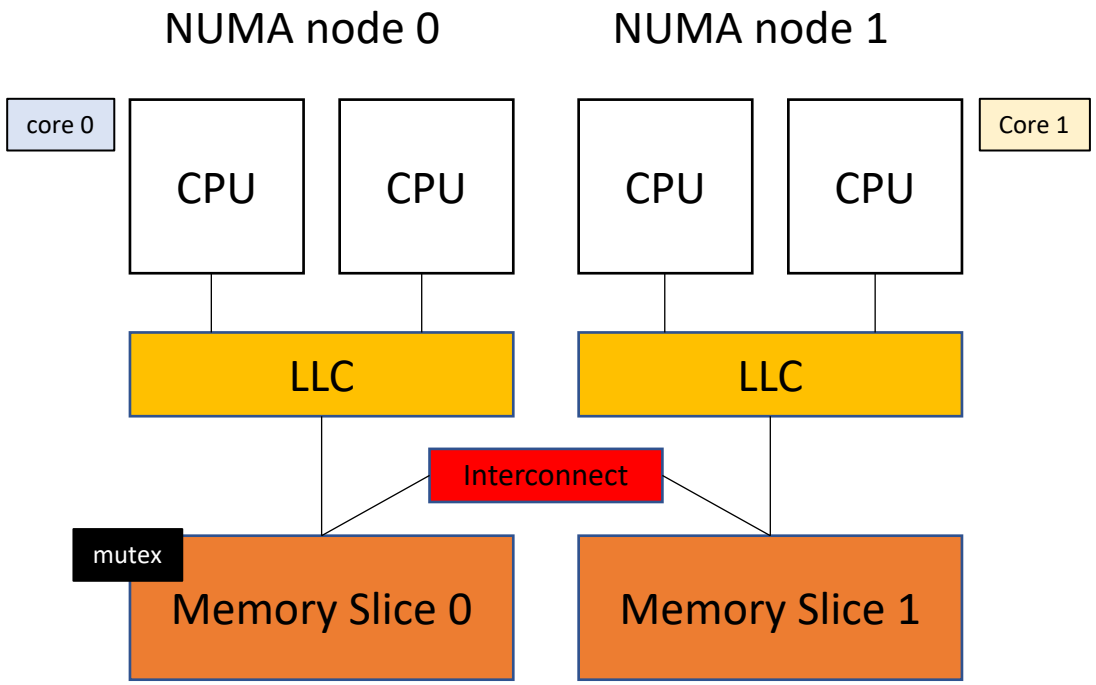
For example: SoCs like Iphone

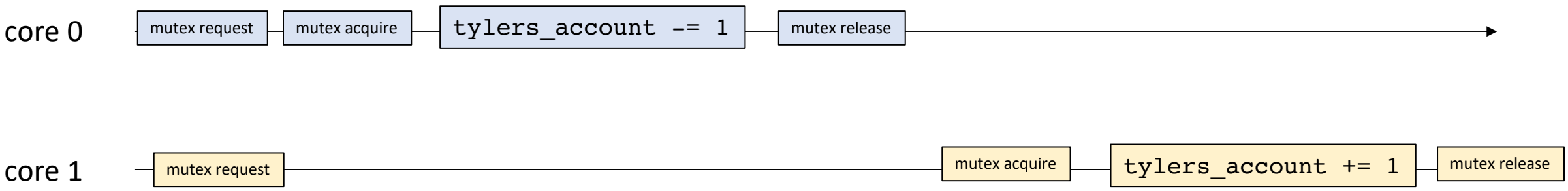


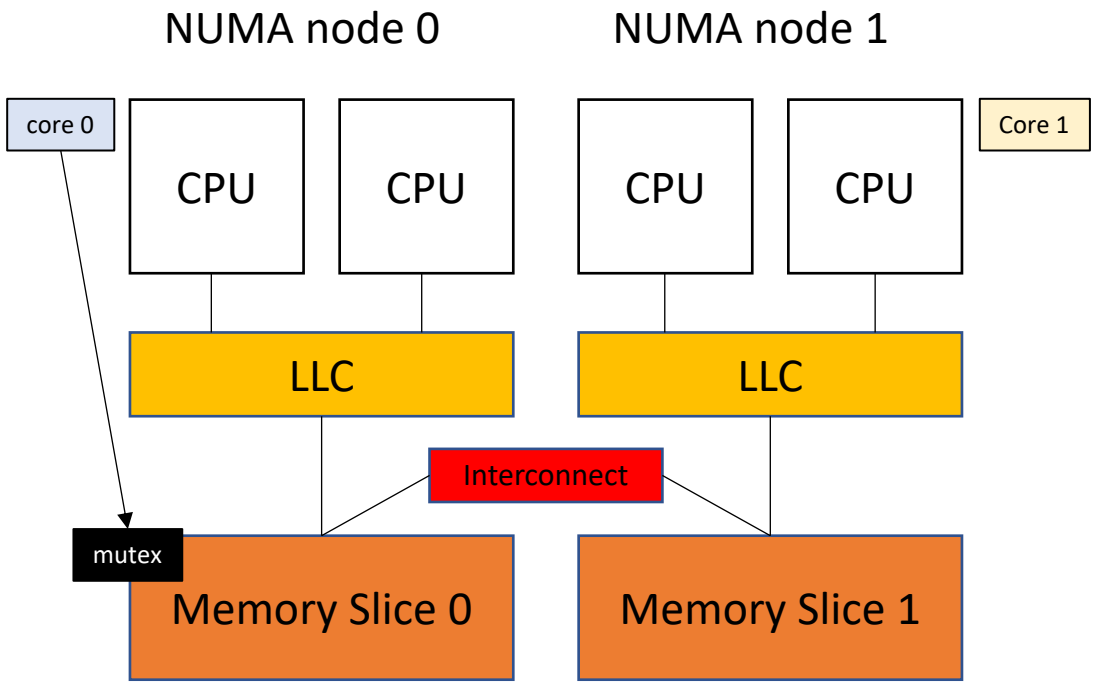
Optimization: Hierarchical locks

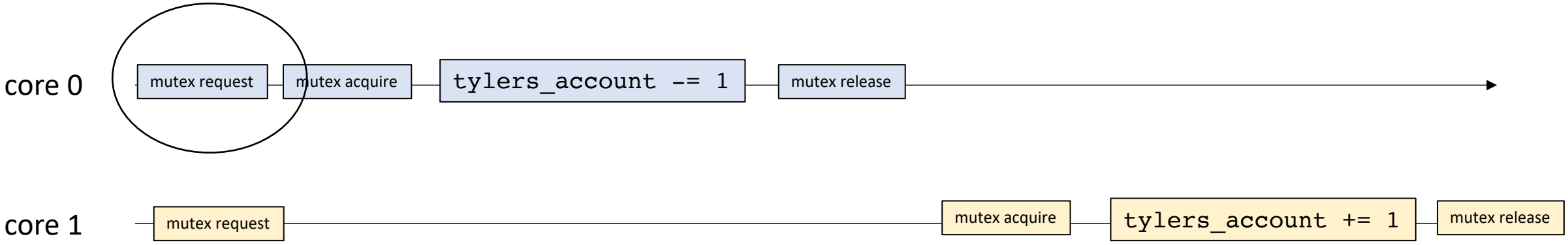
- Any sort of communication is very expensive:
  - Spinning triggers expensive coherence protocols.
  - cache flushes between NUMA nodes is expensive (transferring memory between critical sections)

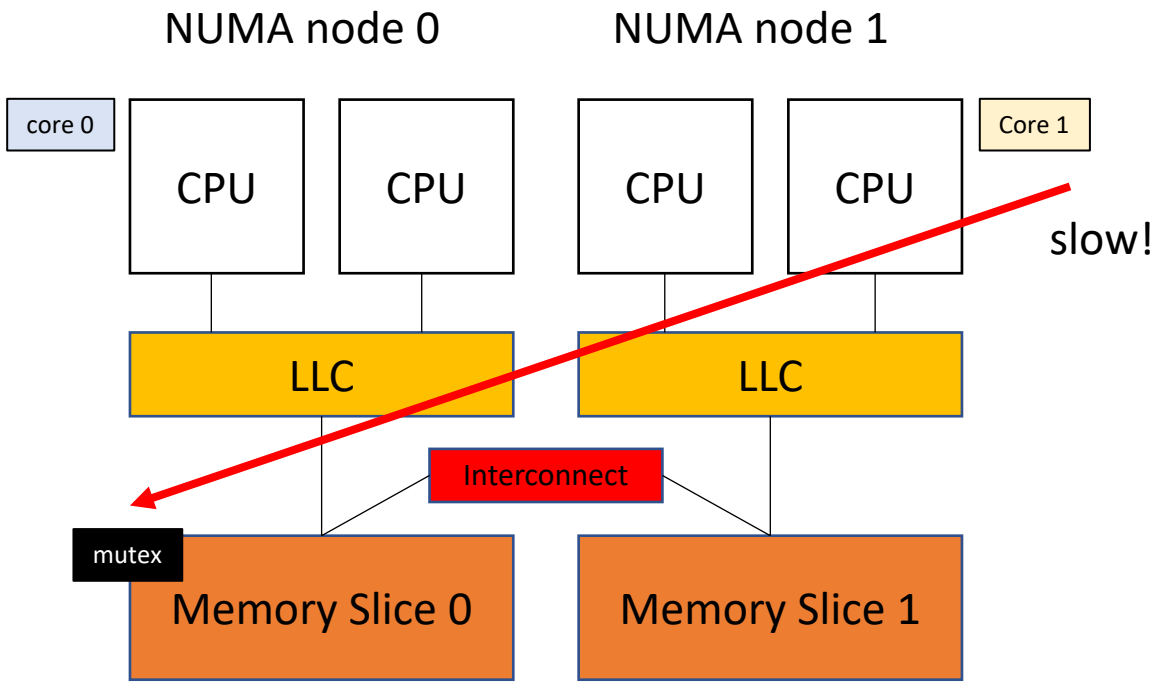


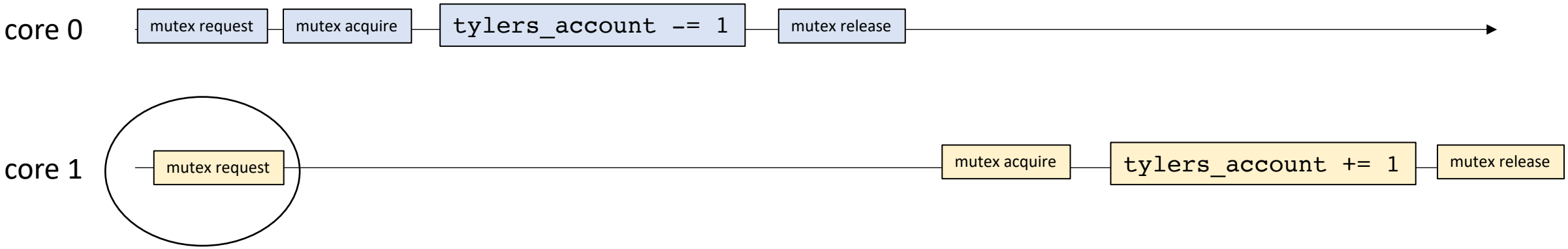

NUMA node 0
NUMA node 1
core 0
CPU
CPU
CPU
CPU
Core 1
LLC
LLC
Interconnect
mutex
Memory Slice 0
Memory Slice 1

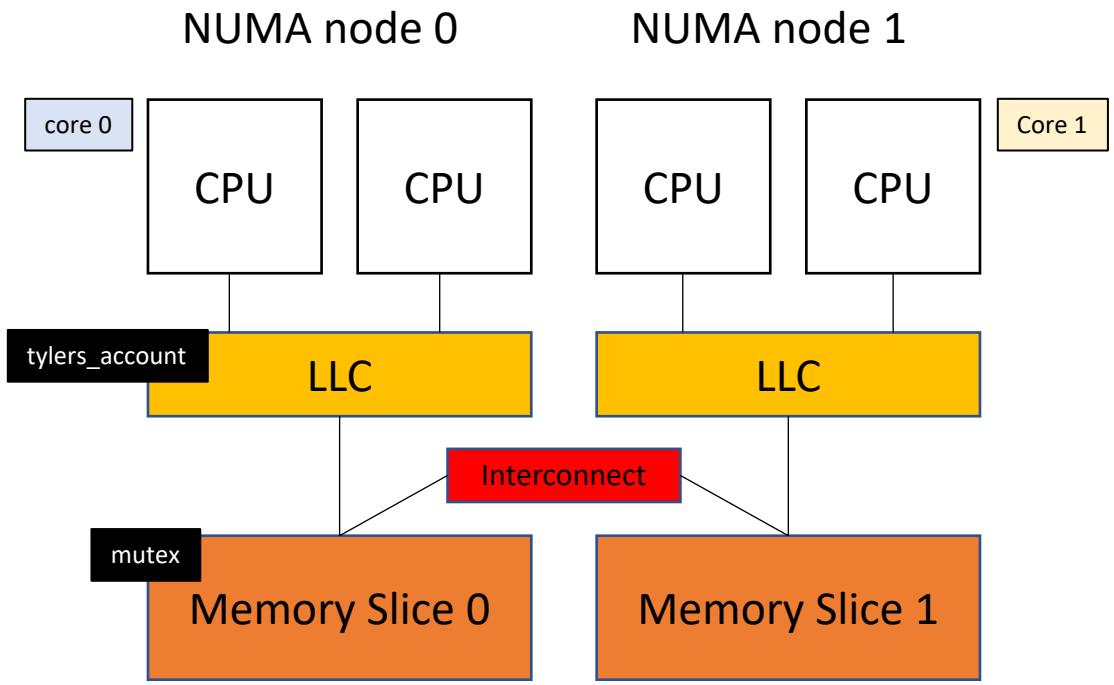

core 0
mutex request
mutex acquire
tylers_account -= 1
mutex release
core 1
mutex request
mutex acquire
tylers_account += 1
mutex release

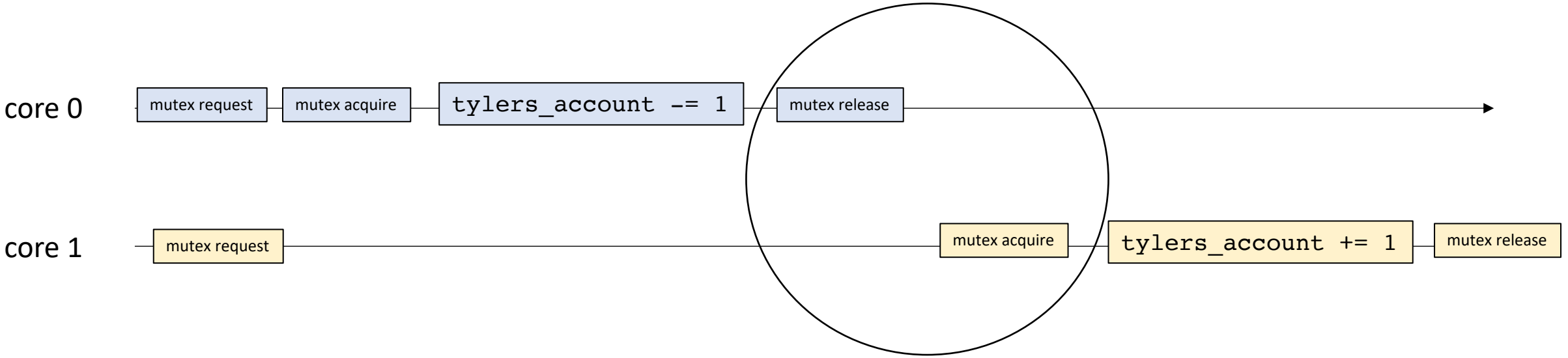

NUMA node 0
NUMA node 1
core 0
CPU
CPU
CPU
CPU
Core 1
LLC
LLC
Interconnect
mutex
Memory Slice 0
Memory Slice 1

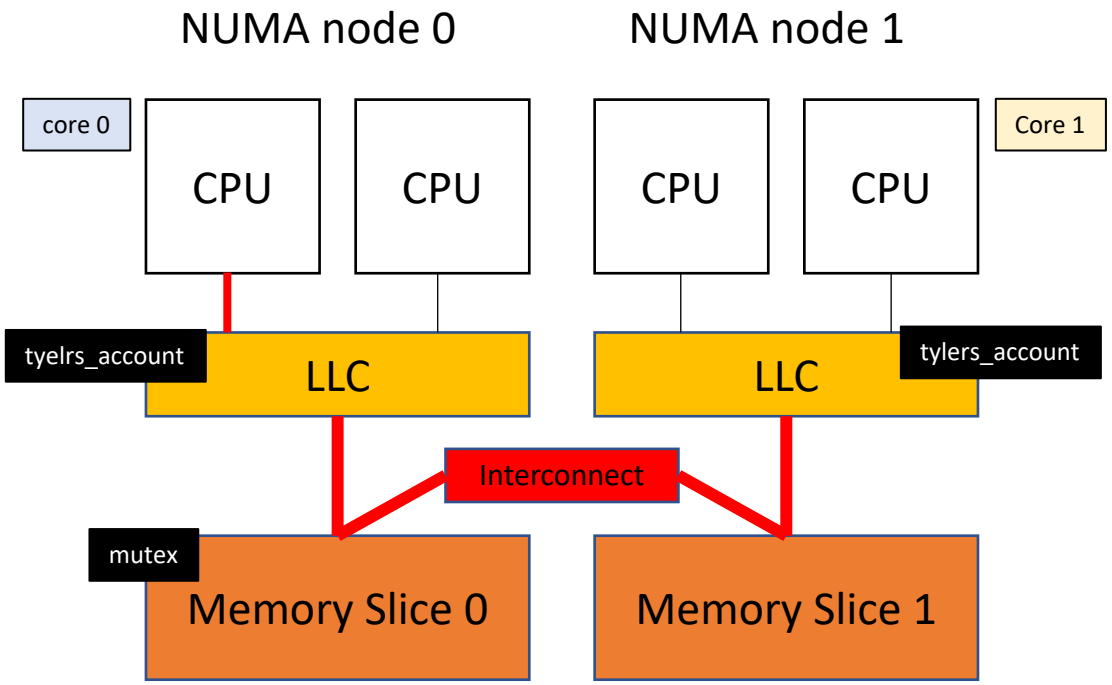

core 0
mutex request
mutex acquire
tylers_account -= 1
mutex release
core 1
mutex request
mutex acquire
tylers_account += 1
mutex release

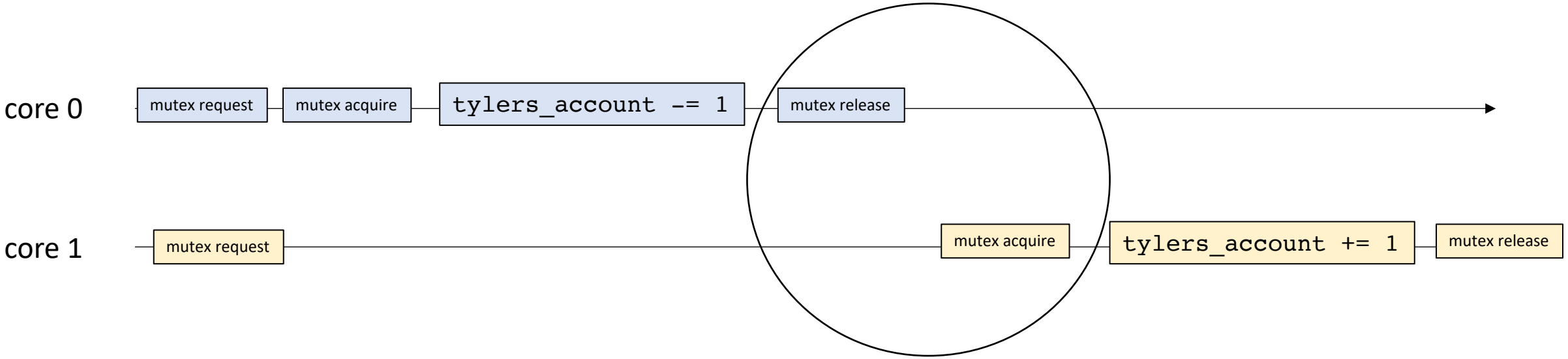

NUMA node 0
NUMA node 1
core 0
CPU
CPU
CPU
CPU
Core 1
slow!
LLC
LLC
Interconnect
mutex
Memory Slice 0
Memory Slice 1

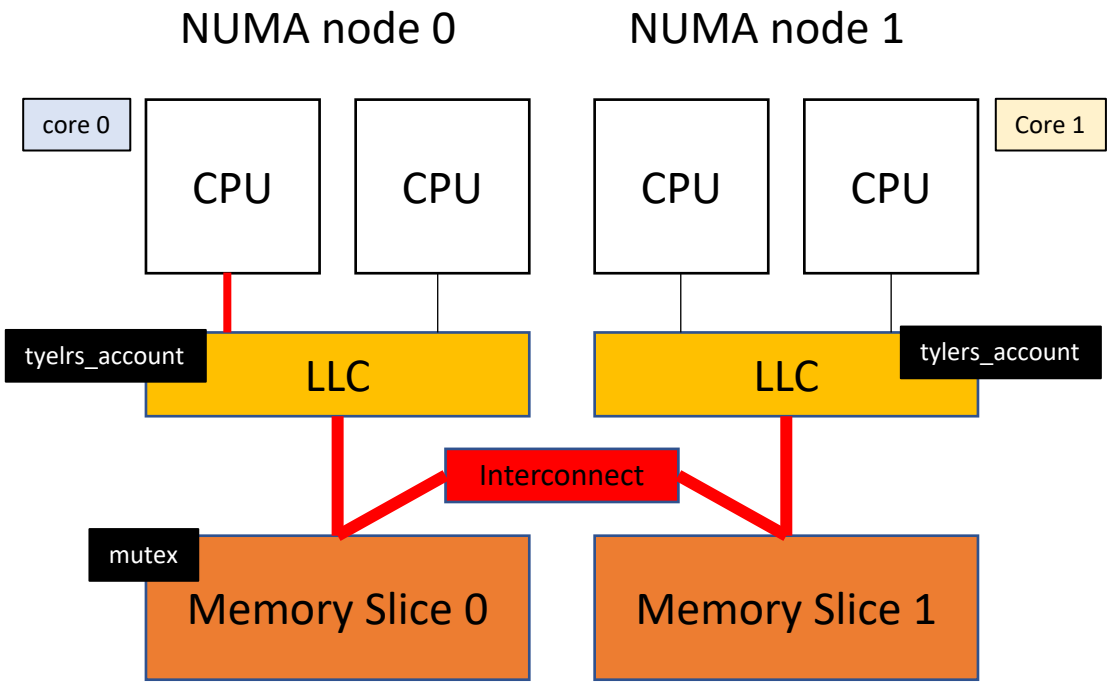

core 0
mutex request
mutex acquire
tylers_account -= 1
mutex release
core 1
mutex request
mutex acquire
tylers_account += 1
mutex release


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

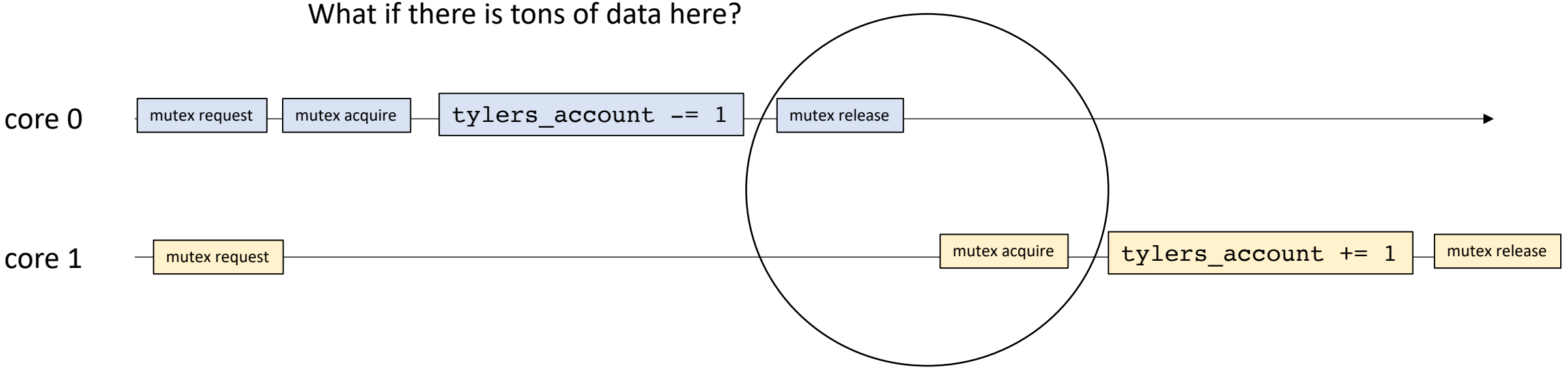

core 0
mutex request
mutex acquire
tylers_account -= 1
mutex release
core 1
mutex request
mutex acquire
tylers_account += 1
mutex release

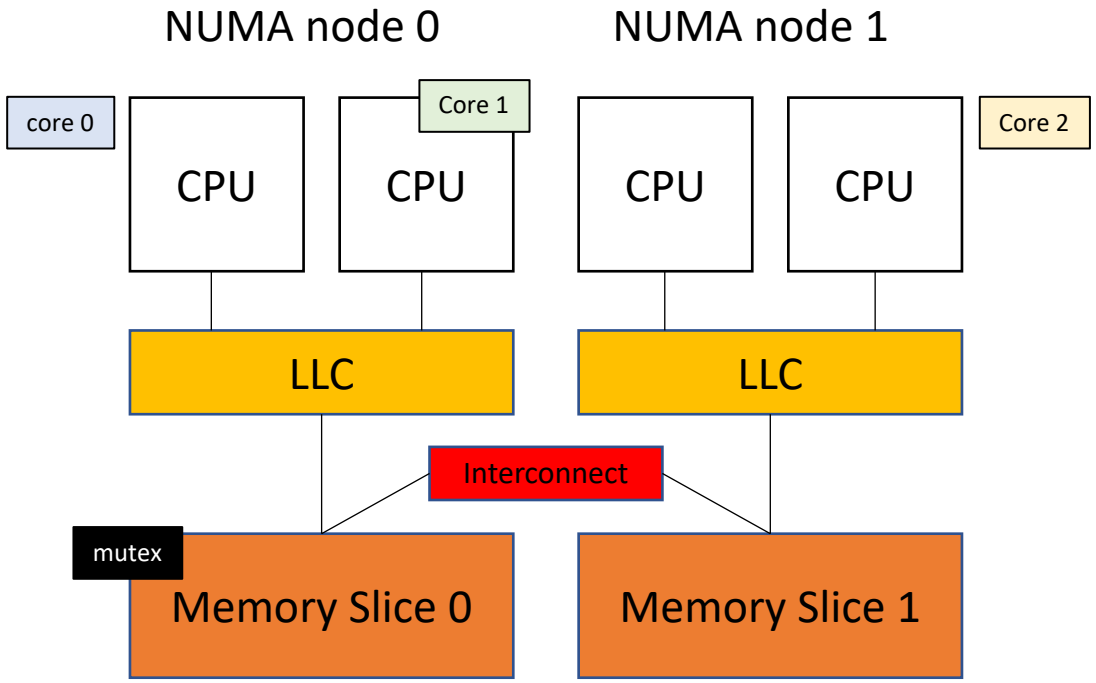

NUMA node 0
NUMA node 1
core 0
CPU
CPU
CPU
CPU
Core 1
tyelrs_account
LLC
LLC
tylers_account
Interconnect
mutex
Memory Slice 0
Memory Slice 1

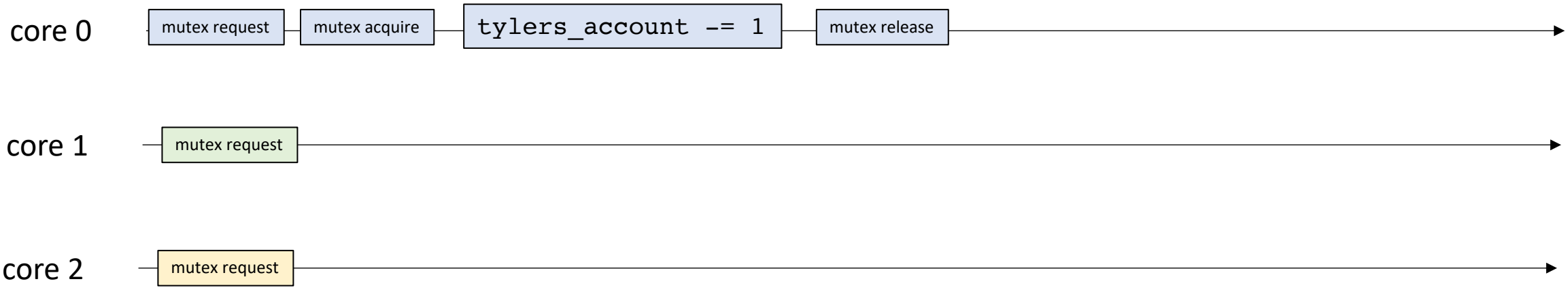

core 0
mutex request
mutex acquire
tylers_account -= 1
mutex release
core 1
mutex request
mutex acquire
tylers_account += 1
mutex release

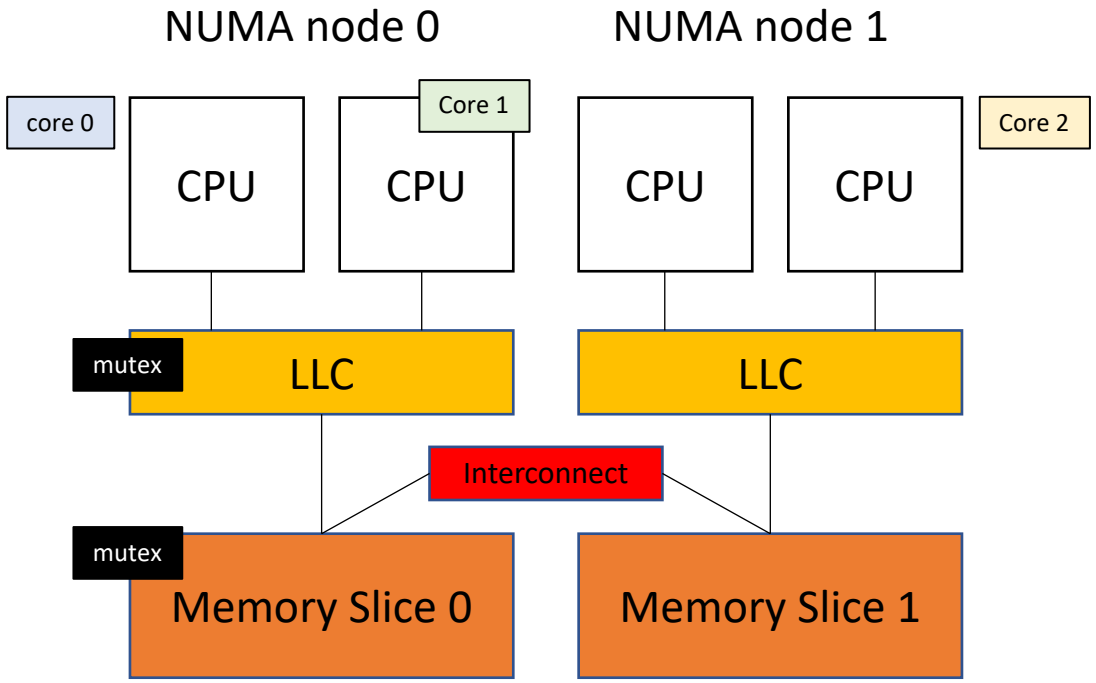


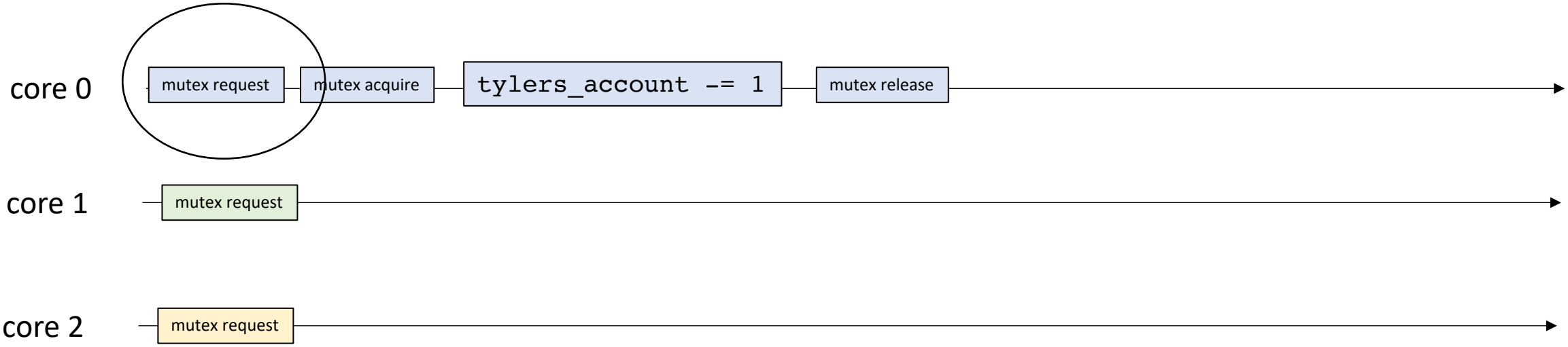
What if there is tons of data here?

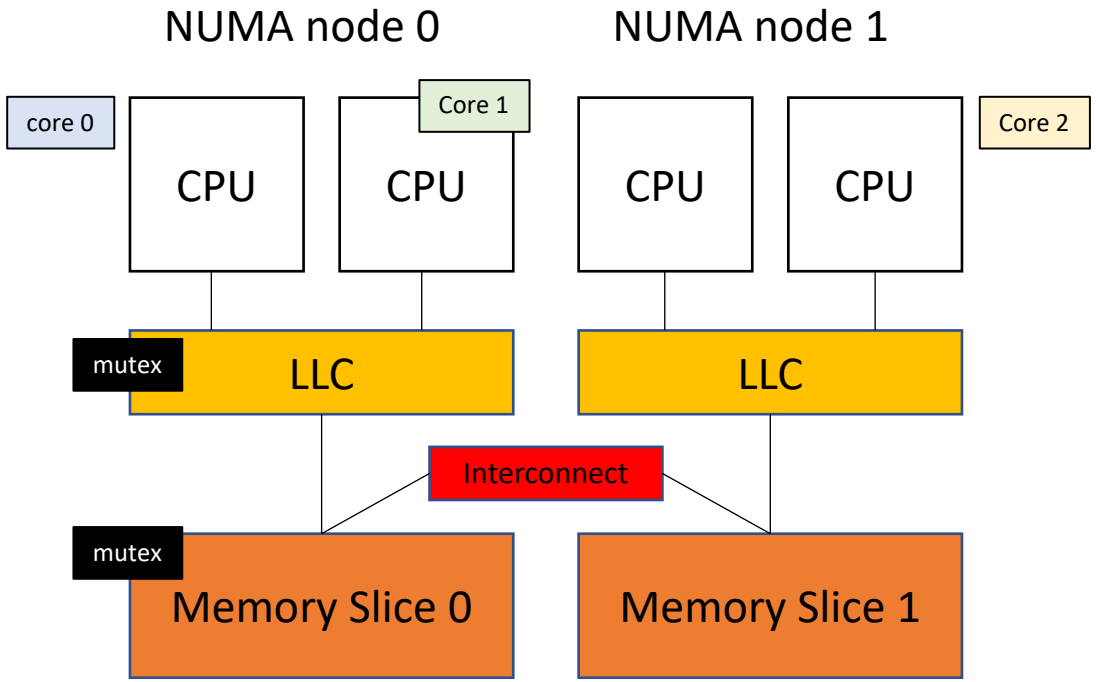


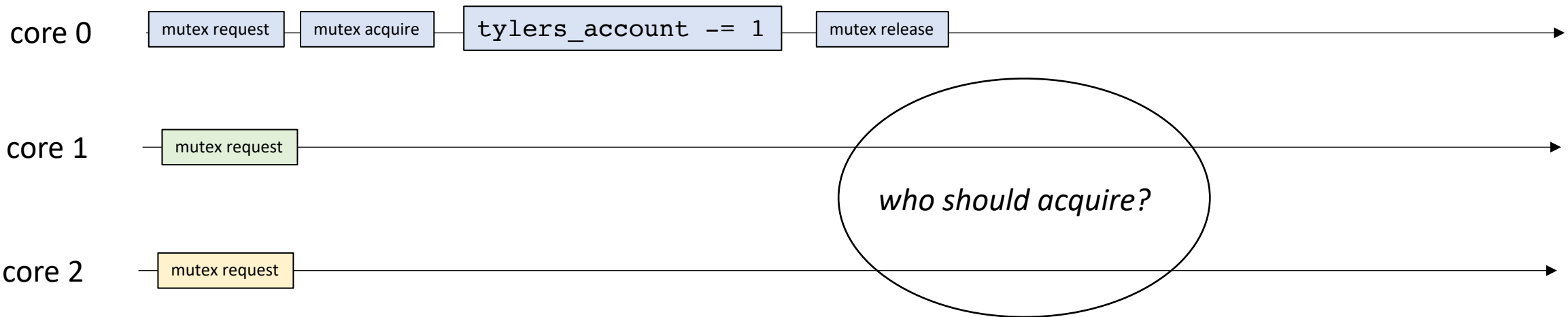

NUMA node 0
NUMA node 1
core 0
Core 1
Core 2
CPU
CPU
CPU
CPU
LLC
LLC
Interconnect
mutex
Memory Slice 0
Memory Slice 1


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

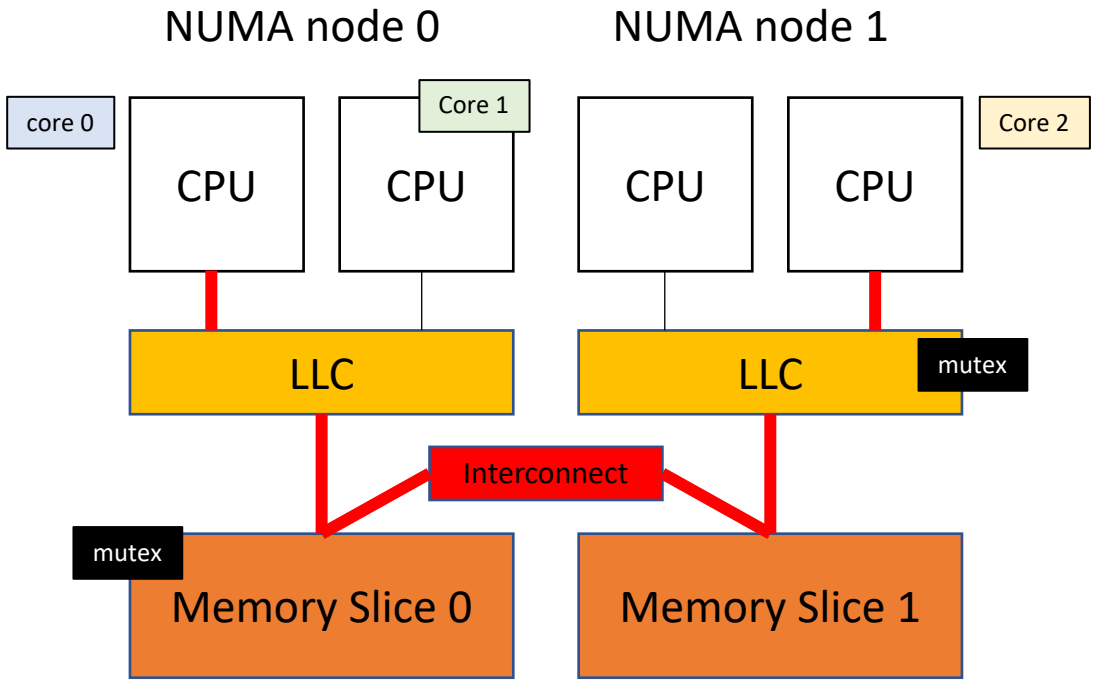

NUMA node 0
NUMA node 1
core 0
CPU
Core 1
CPU
CPU
CPU
Core 2
mutex
LLC
LLC
Interconnect
mutex
Memory Slice 0
Memory Slice 1

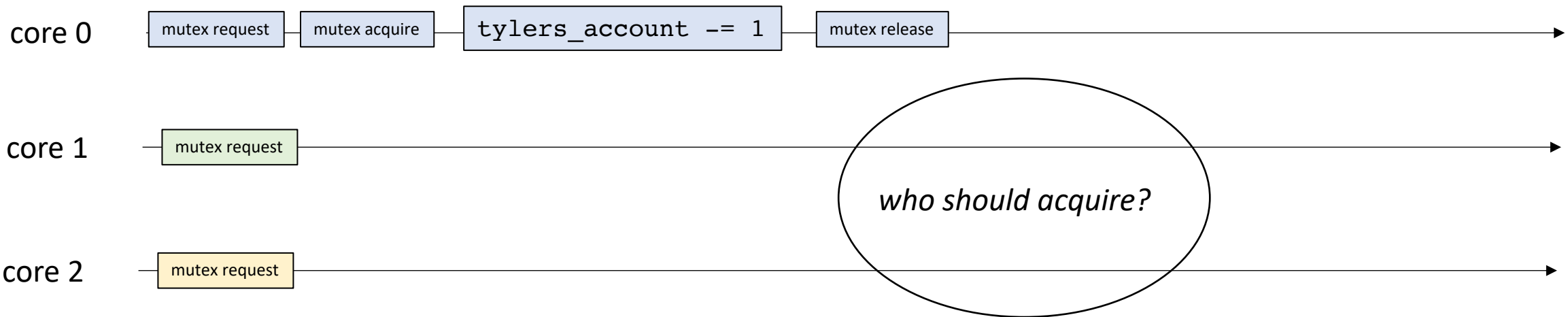

core 0
mutex request
mutex acquire
tylers_account -= 1
mutex release
core 1
mutex request
core 2
mutex request


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

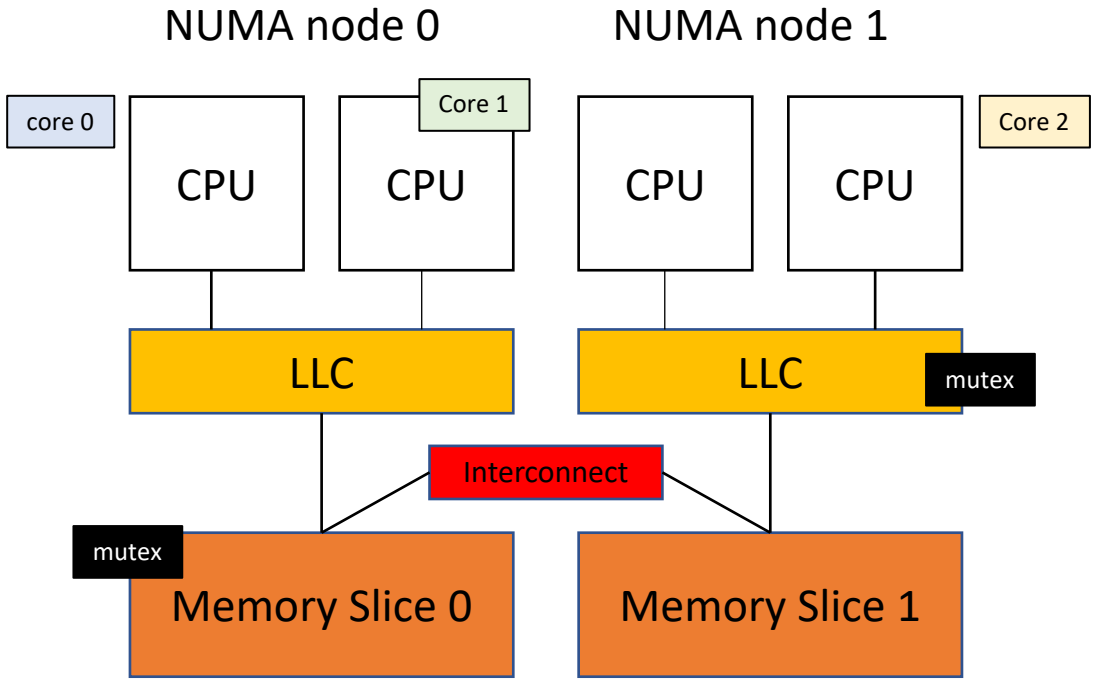

core 0
mutex request
mutex acquire
tylers_account -= 1
mutex release
core 1
mutex request
who should acquire?
core 2
mutex request

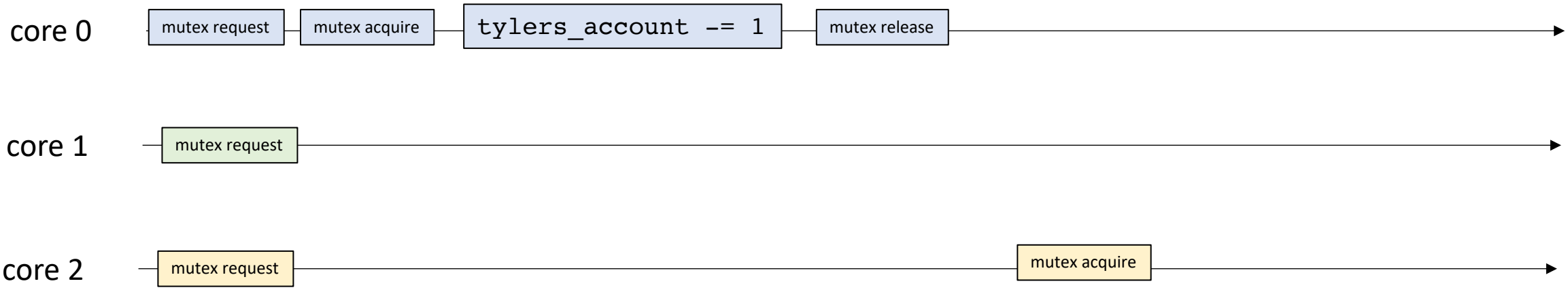
If core 2 acquires first communication must go through the interconnect



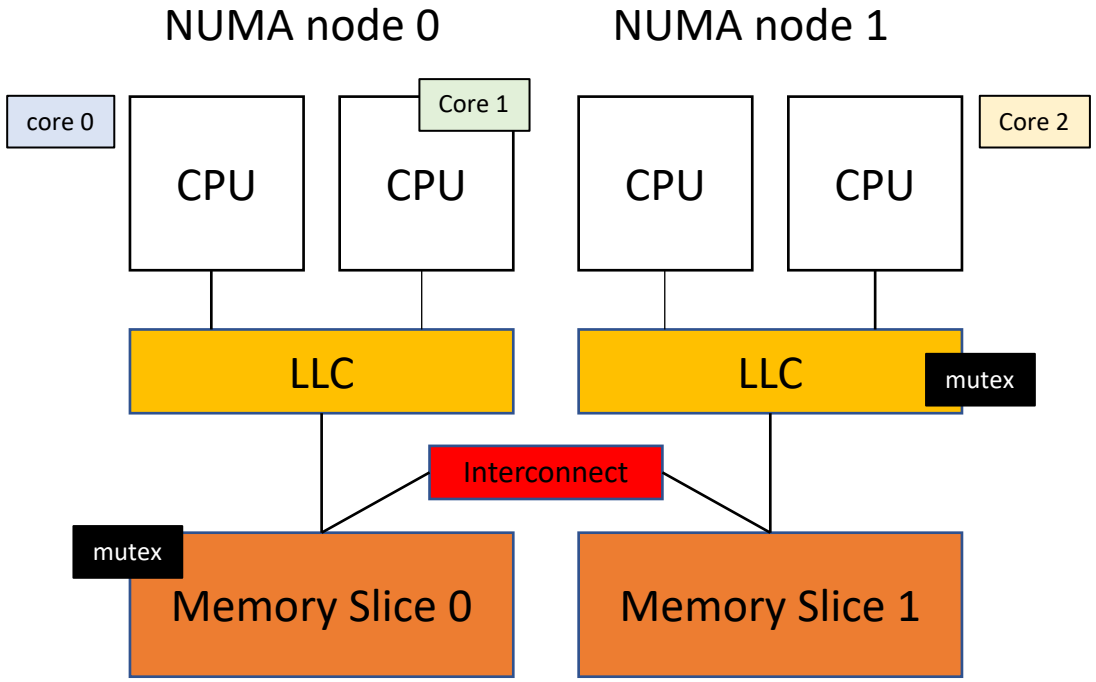


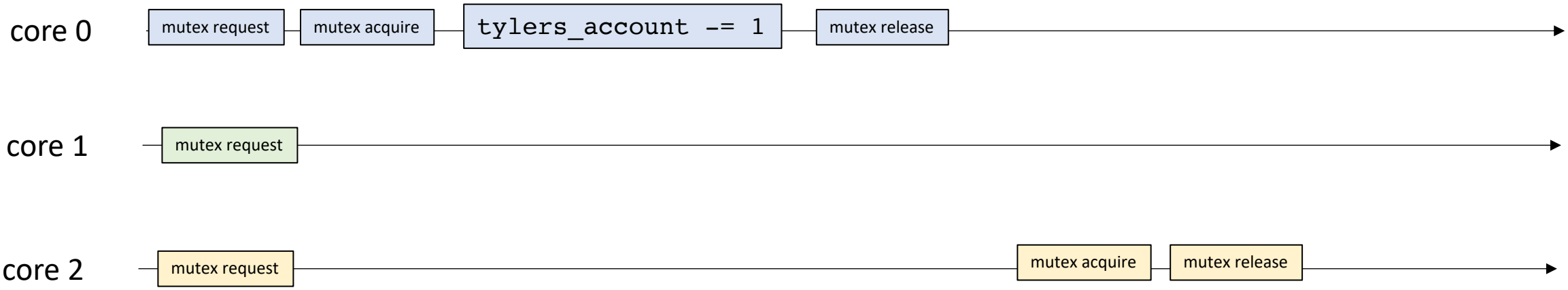
If core 2 acquires first communication must go through the interconnect





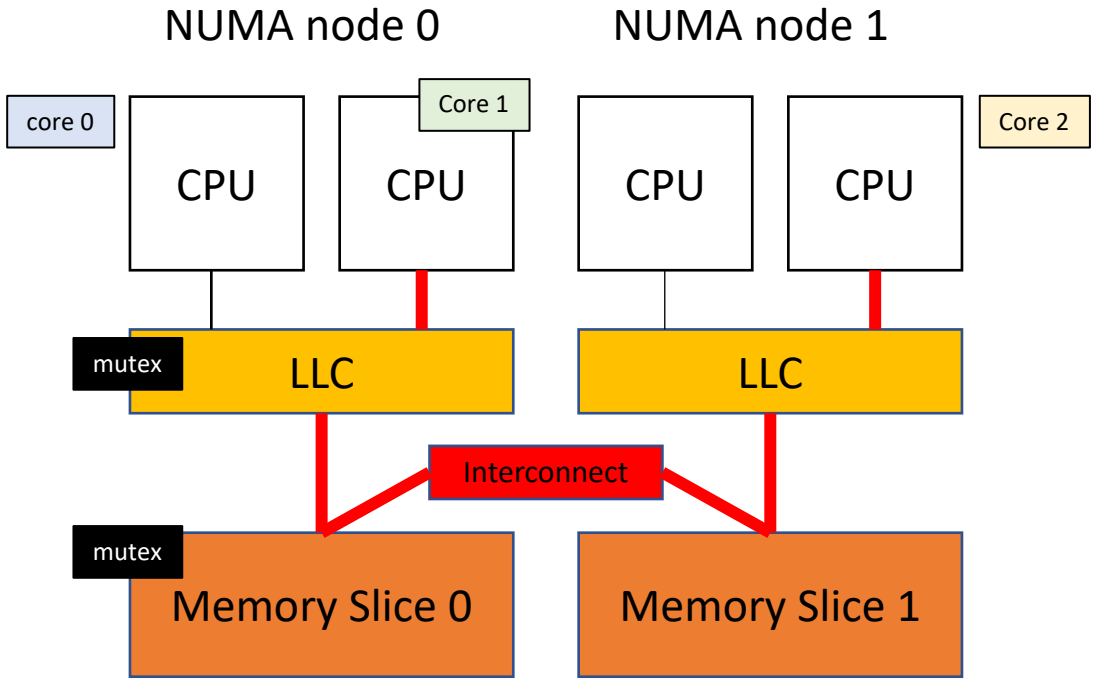
If core 2 acquires first communication must go through the interconnect

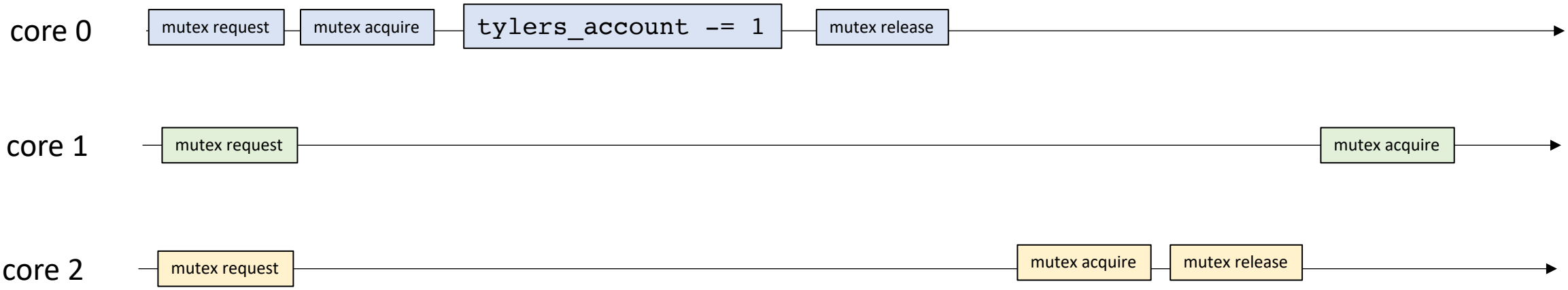




If core 2 acquires first communication must go through the interconnect

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

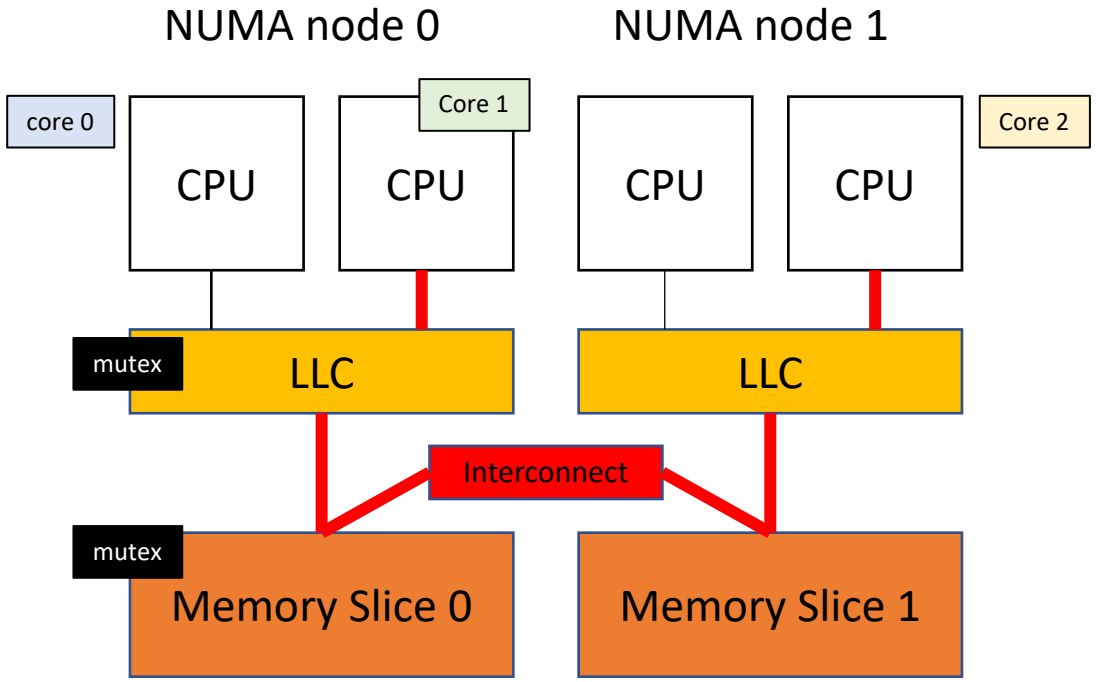


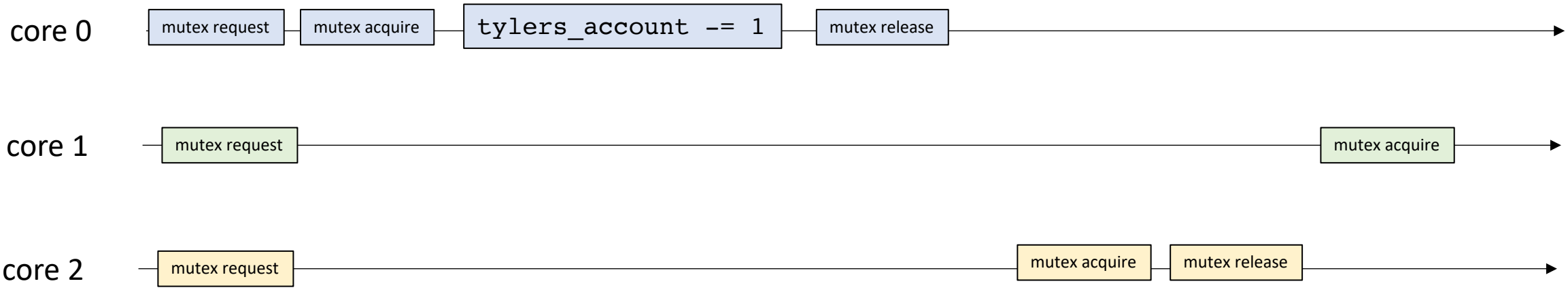


Two trips through the interconnect!!

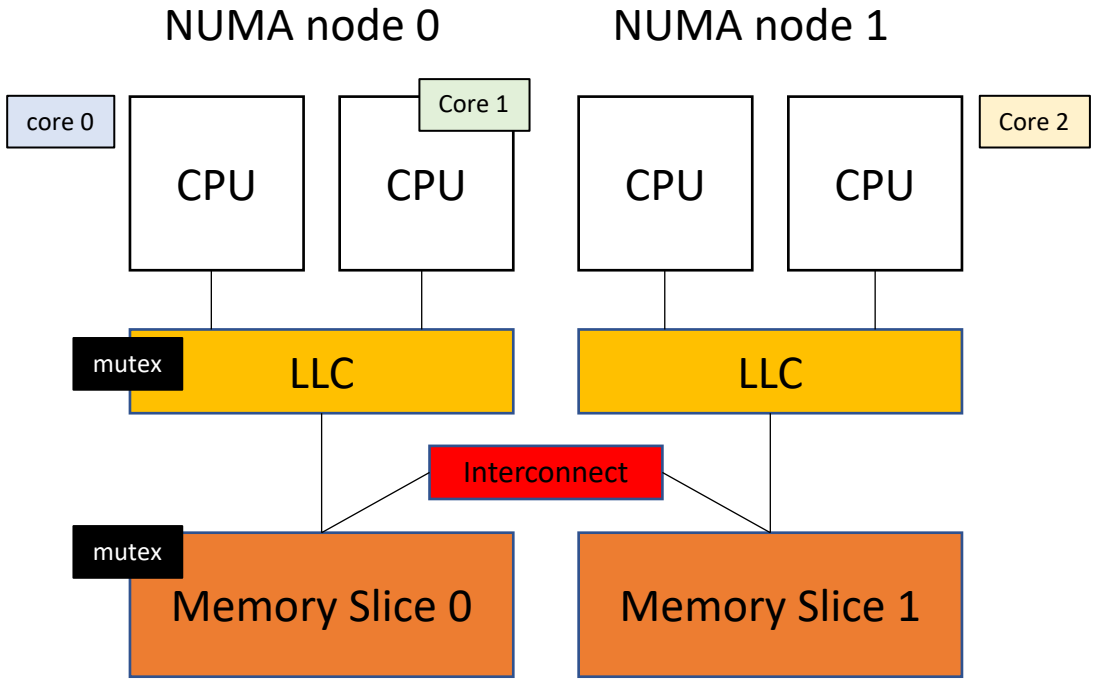
If core 2 acquires first communication must go through the interconnect

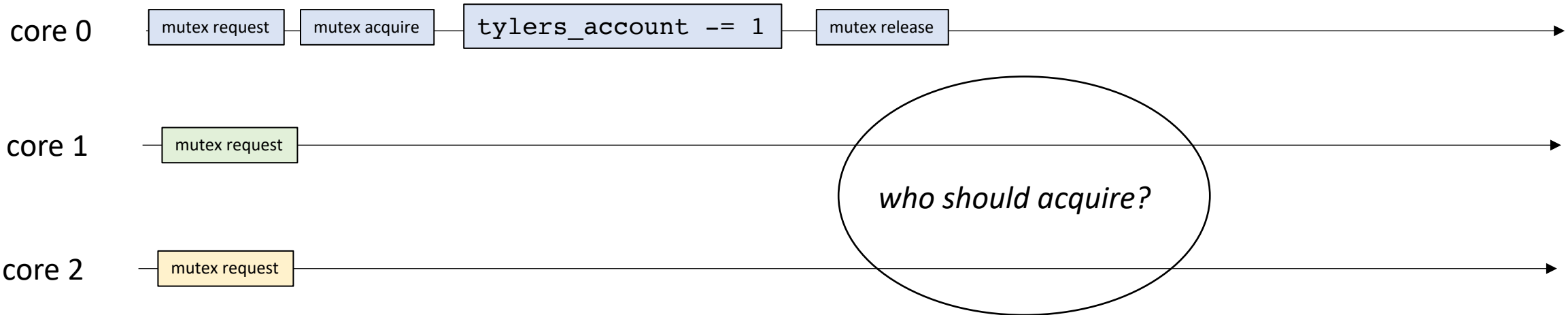
When core 1 finally acquires, it requires another expensive trip through the interconnect



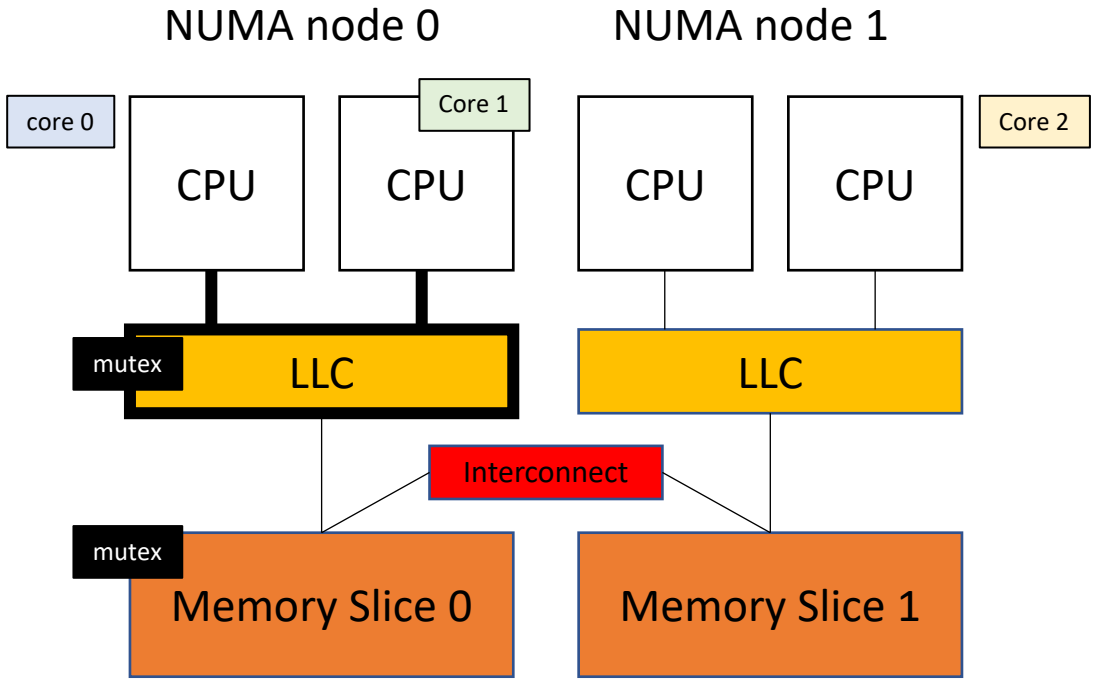


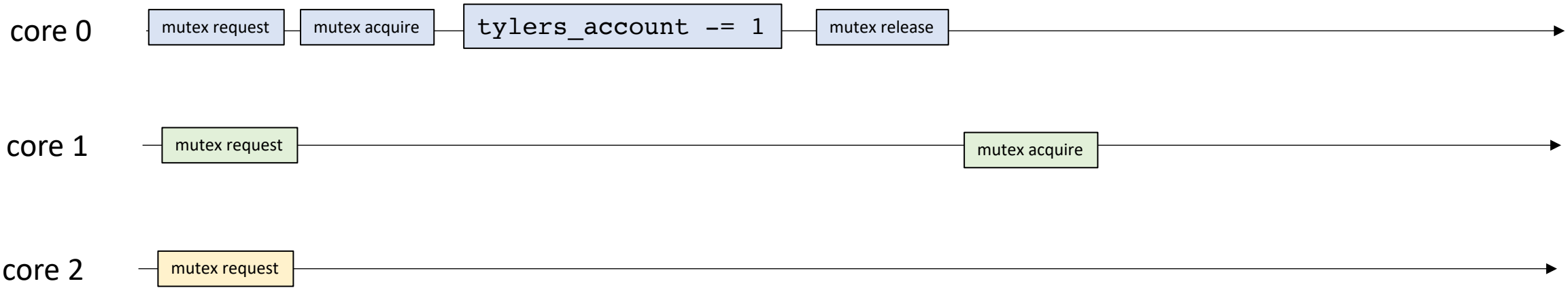
Lets go back in time and make a different decision!





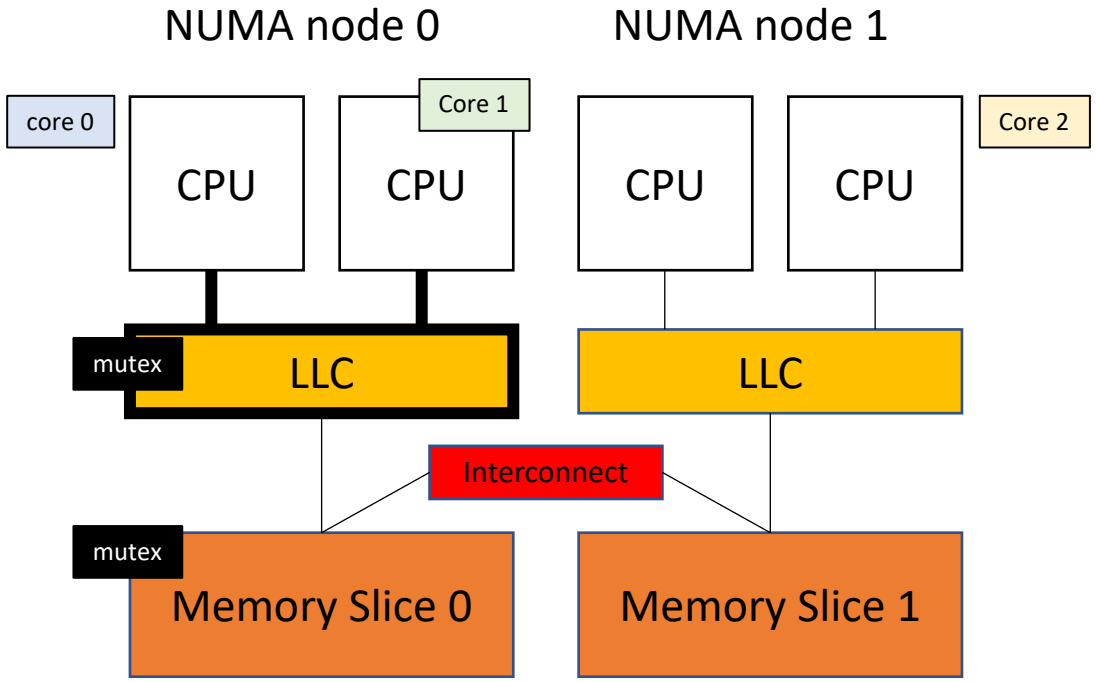
If core 1 acquires first communication can occur through the LLC of NUMA node 0

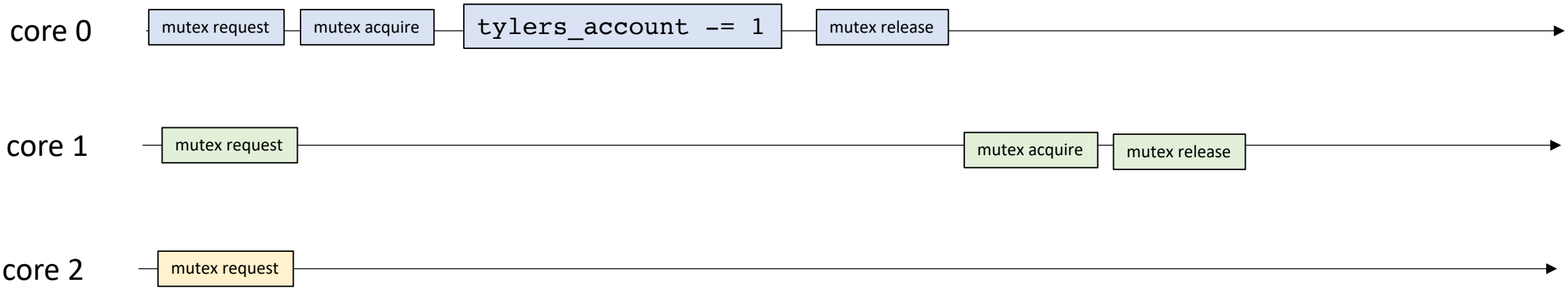




If core 1 acquires first communication can occur through the LLC of NUMA node 0

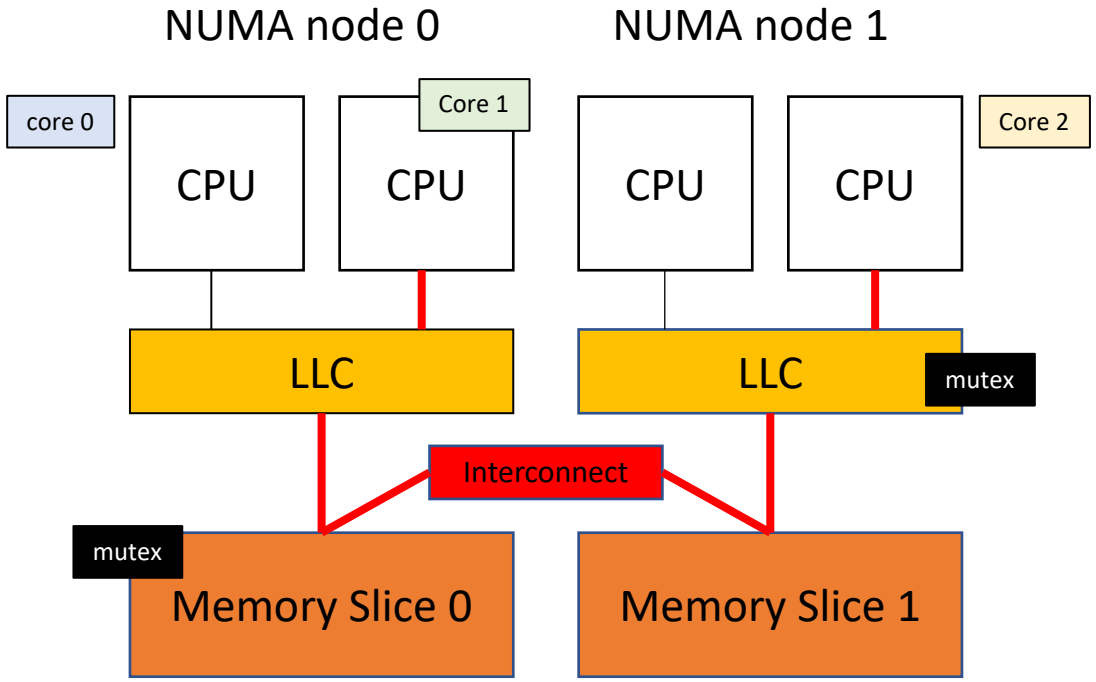
When core 2 finally acquires it requires an expensive trip through the interconnect

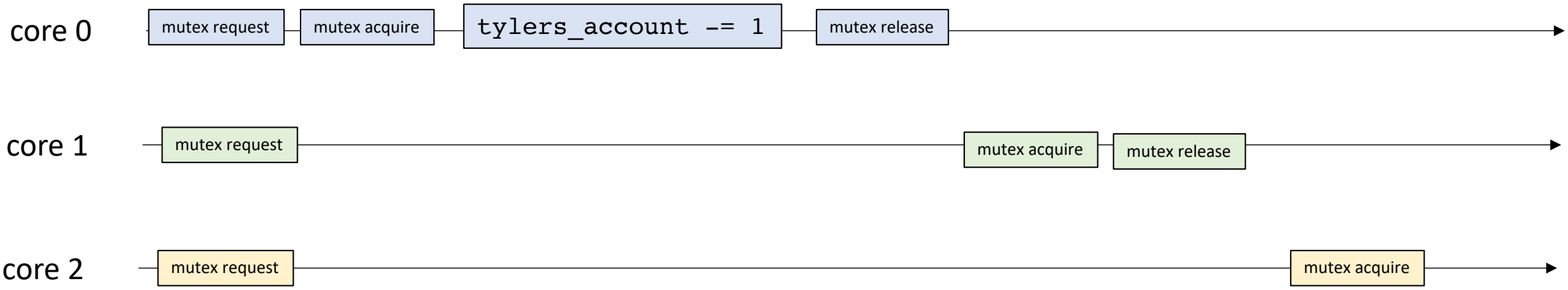




If core 1 acquires first communication can occur through the LLC of NUMA node 0

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

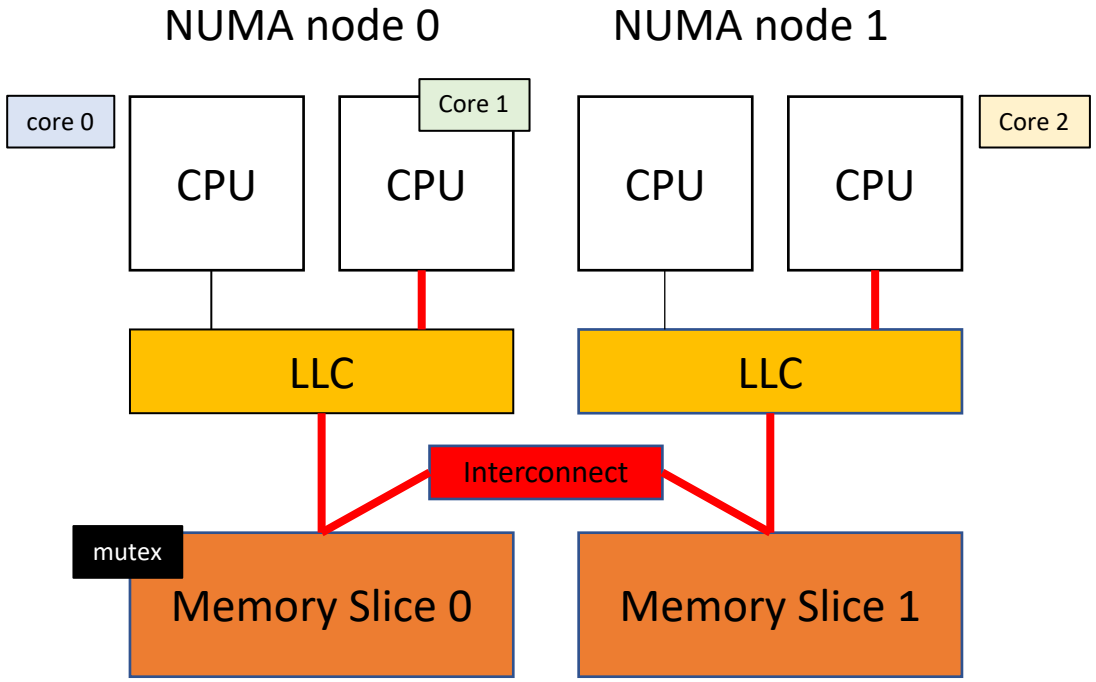


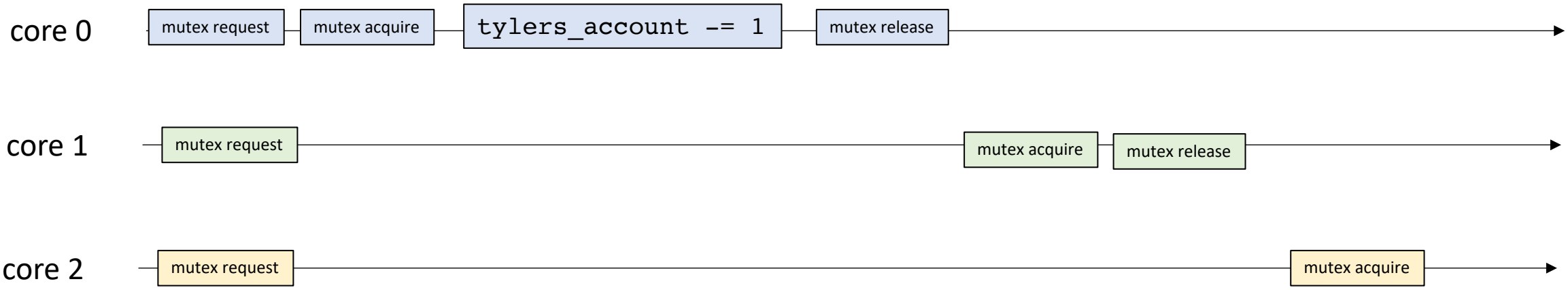


Only 1 trip through the interconnect

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;
  }
}
```

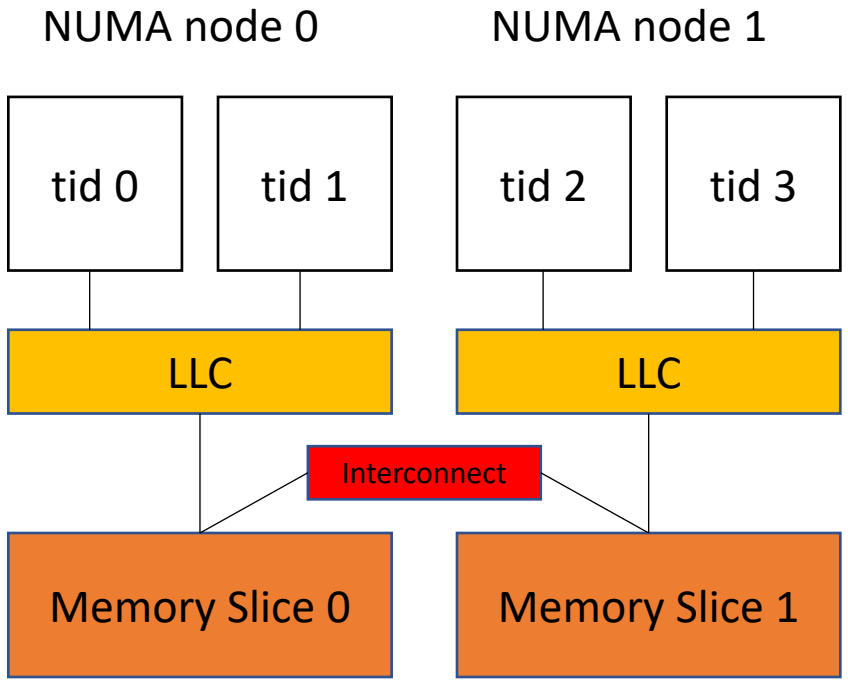
We have a new lock

- 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?

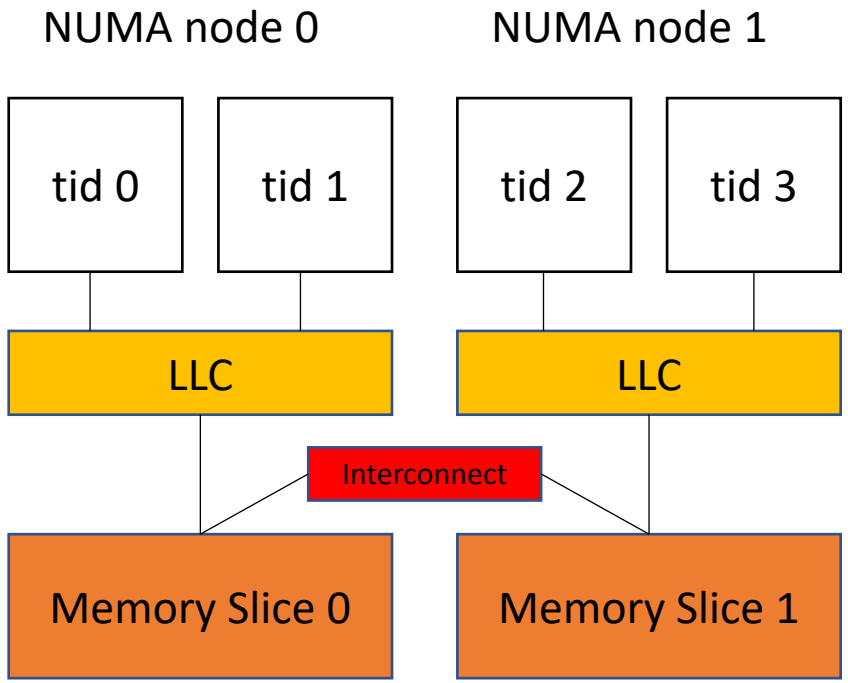
- Tune sleep times. You shouldn’t starve the other nodes!

- 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:

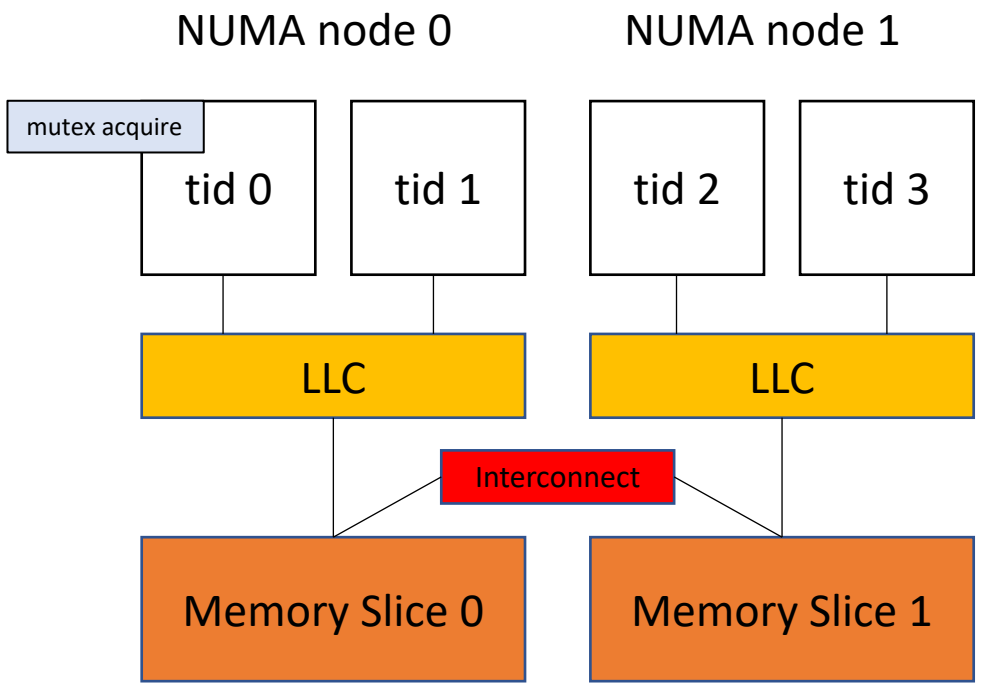
Mutex counter:
Local_Com: 0



Example:

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

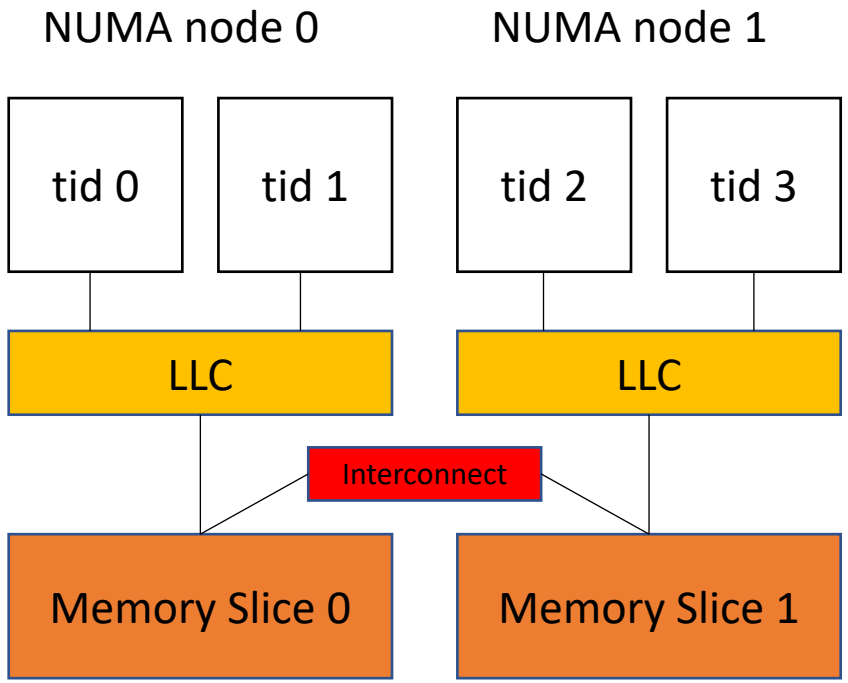
Mutex counter:
Local_Com: 1



Example:

tid 0:
tid 1:
tid 2:

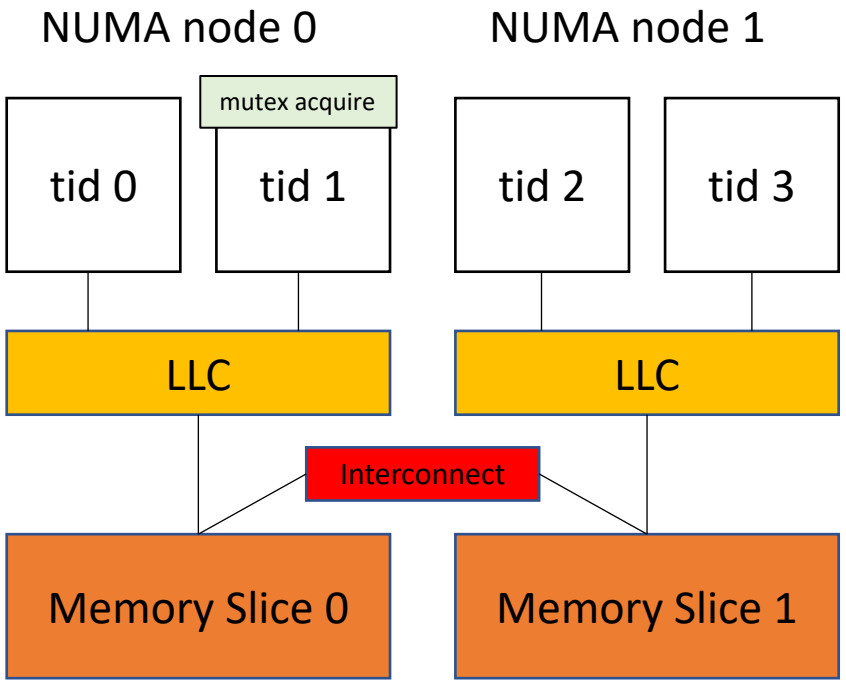
Mutex counter:
Local_Com: 1



Example:

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

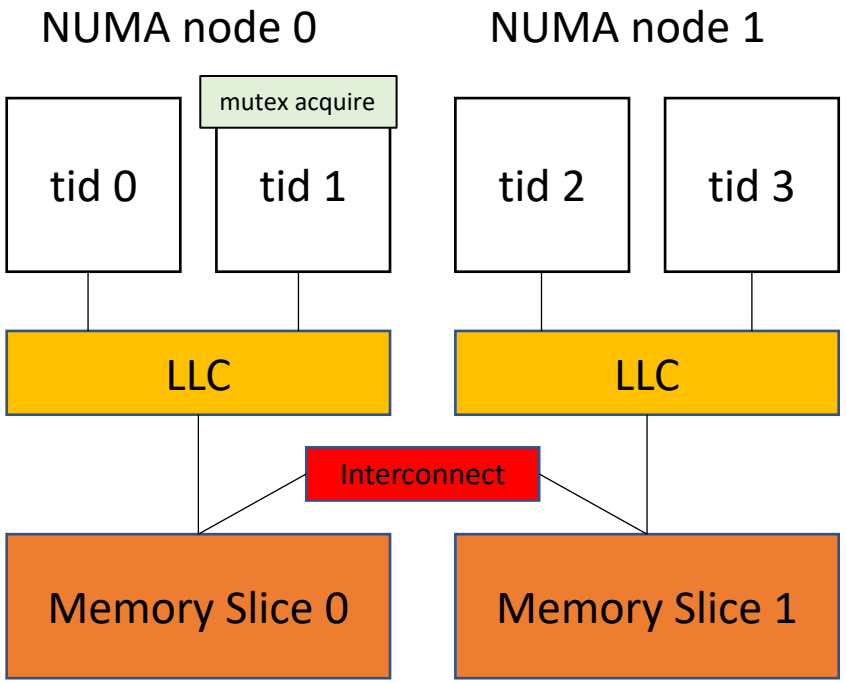
Mutex counter:
Local_Com: 2



Example:

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

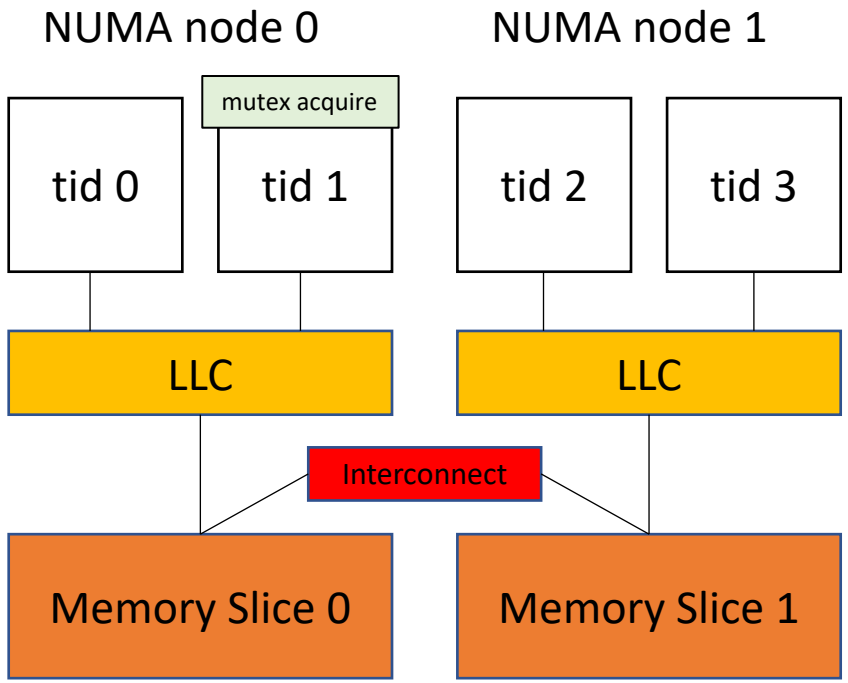
Mutex counter:
Local_Com: 2



Example:

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

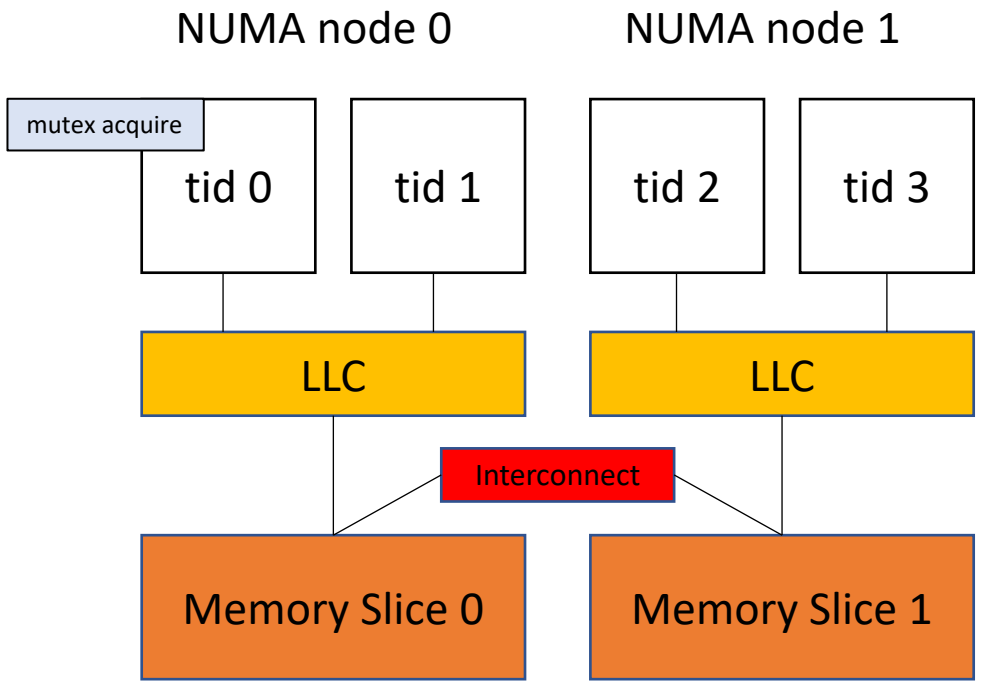
Mutex counter:
Local_Com: 2



Example:

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

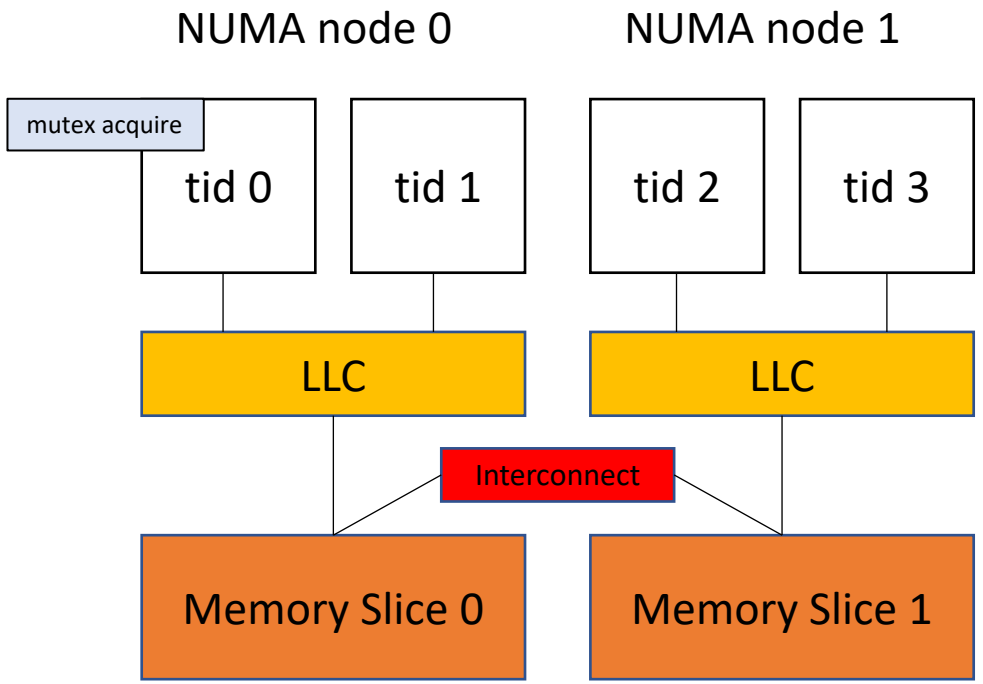
Mutex counter:
Local_Com: 3



Example:

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

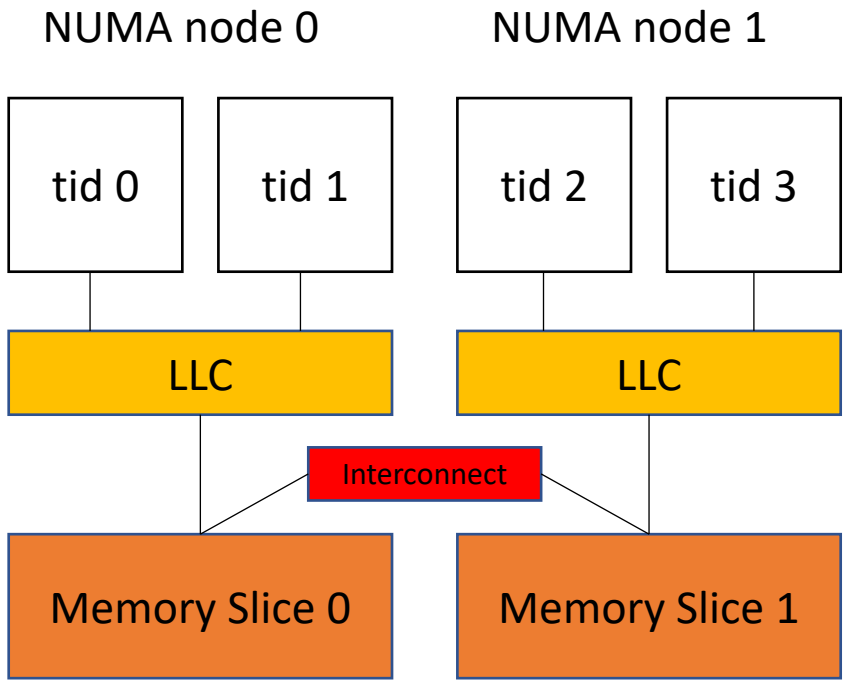
Mutex counter:
Local_Com: 3



Example:

tid 0:
tid 1:
tid 2:

Mutex counter:
Local_Com: 3

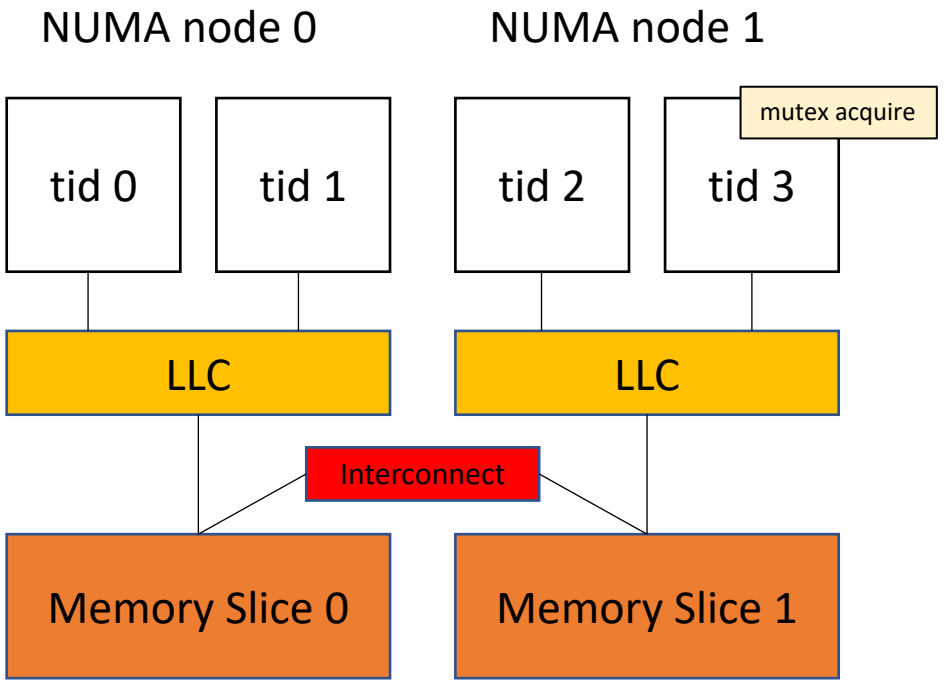


Example:

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!

Next lecture

- Starting on Module 3
- Work on HW 2! You now have everything you need to complete it!