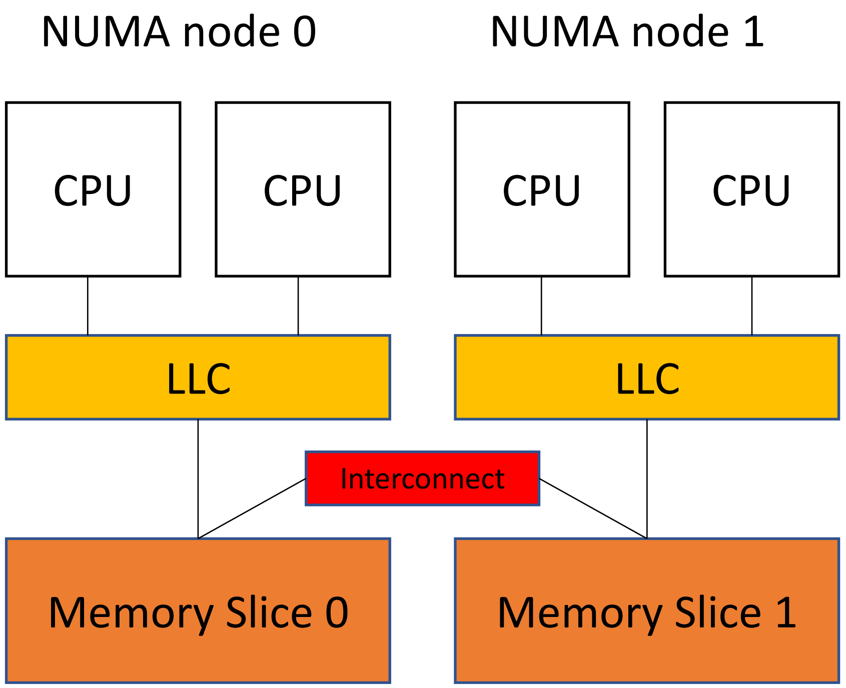
CSE113: Parallel Programming

Jan. 28, 2022

- **Topics**:
  - RW mutexes
  - Hierarchical aware locks
  - Impact of real world data conflicts



Announcements

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

- Homework 2 is due next Friday
  - People are making good progress on part 1
  - Today’s lecture will get you through the rest
  - After Monday you can start sharing results (not code)

Announcements

- Schedule:
  - Starting Module 3 next week: Concurrent data structures!
- Midterm assigned Feb. 7:
  - Available for 1 week, not timed
  - Designed to take ~3 hours
  - open book, open note, open slide
  - Do not discuss at all with classmates
  - You can use google, but ***do not*** google questions exactly, or ask on stackoverflow

Returning to in-person

- Monday’s synchronous lecture will be in-person!
  - Kresge 327
  - Record lectures and post them after
  - Quizes (attendance) will maintain the same format, please do them!

Today’s Quiz

- Due Monday by class time

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?

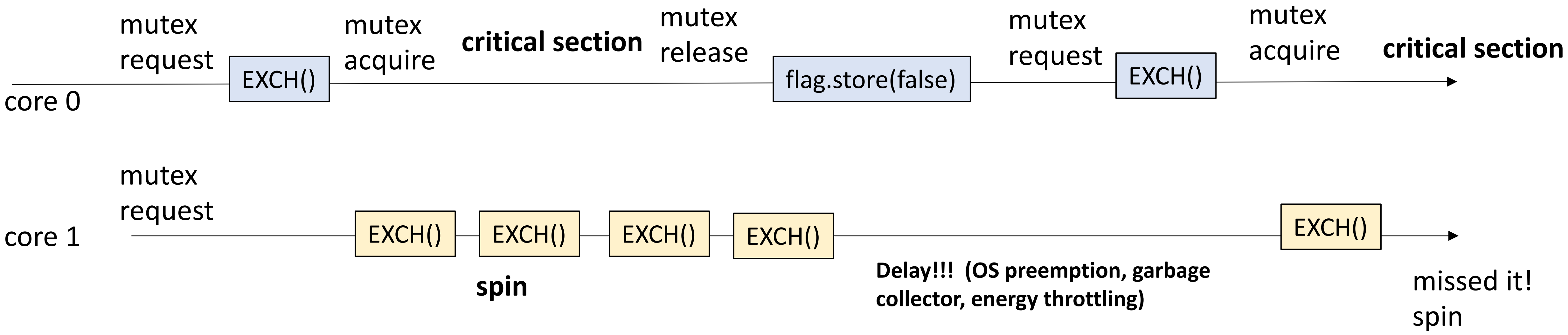
Review

Fairness of RMW locks

are EXCH/CAS locks starvation free?

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

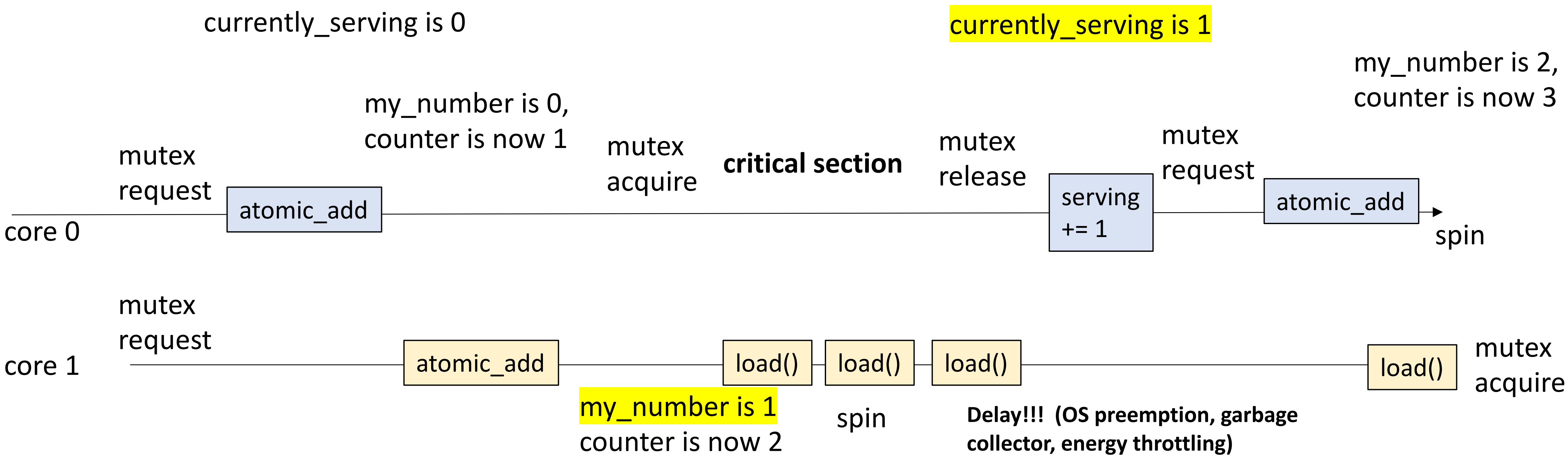
```
void unlock() {
  flag.store(false);
}
```



are Ticket locks are fair?

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

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



Mutex optimizations

Optimizations: relaxed peeking

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

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

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**

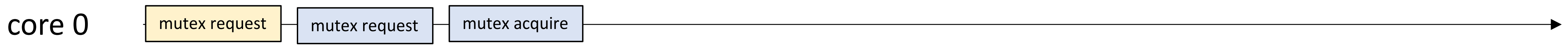
Say threads 0 and 1 are executing concurrently

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**

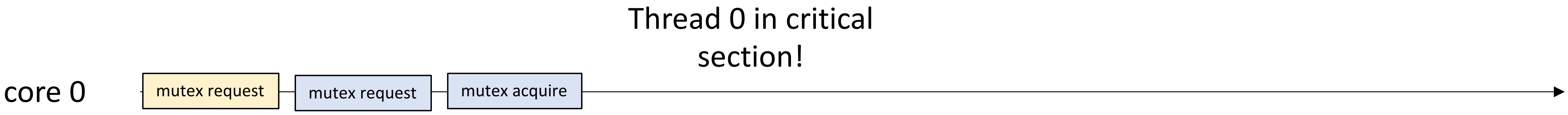
Say threads 0 and 1 are executing concurrently



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**

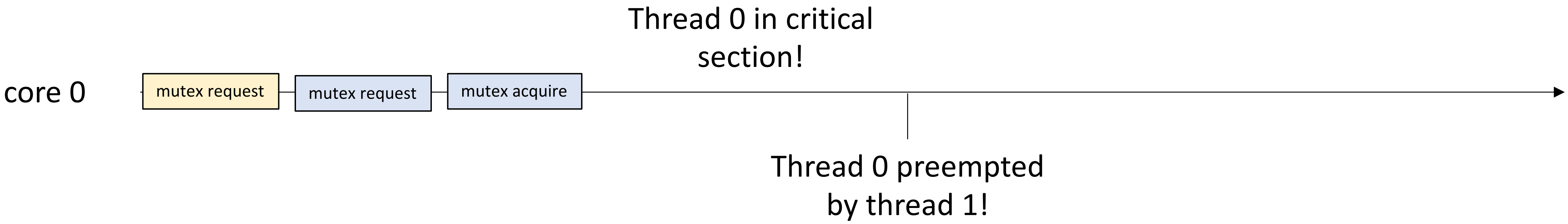
Say threads 0 and 1 are executing concurrently



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**

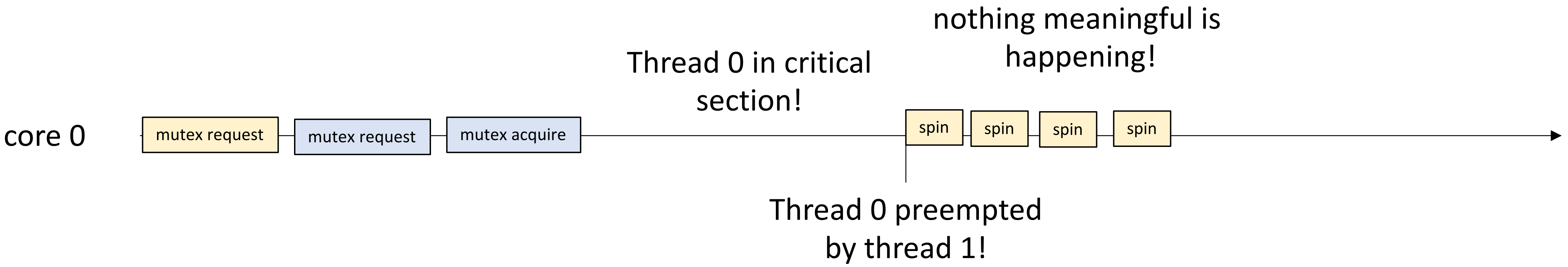
Say threads 0 and 1 are executing concurrently



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**

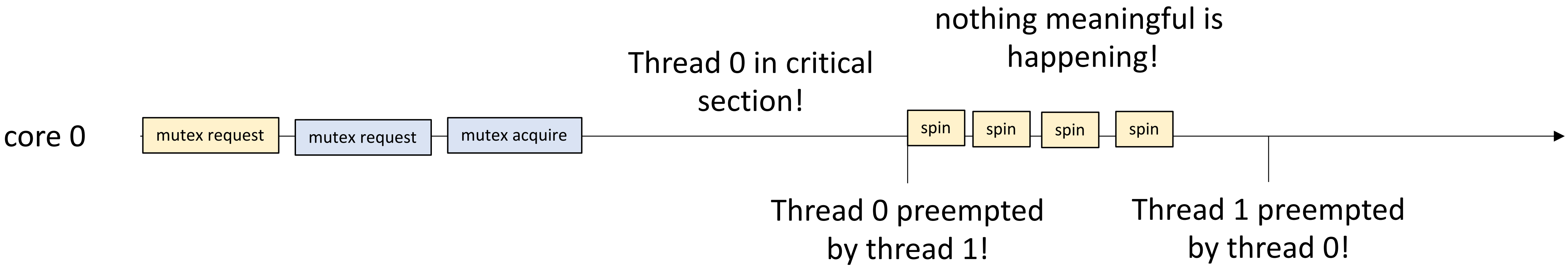
Say threads 0 and 1 are executing concurrently



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**

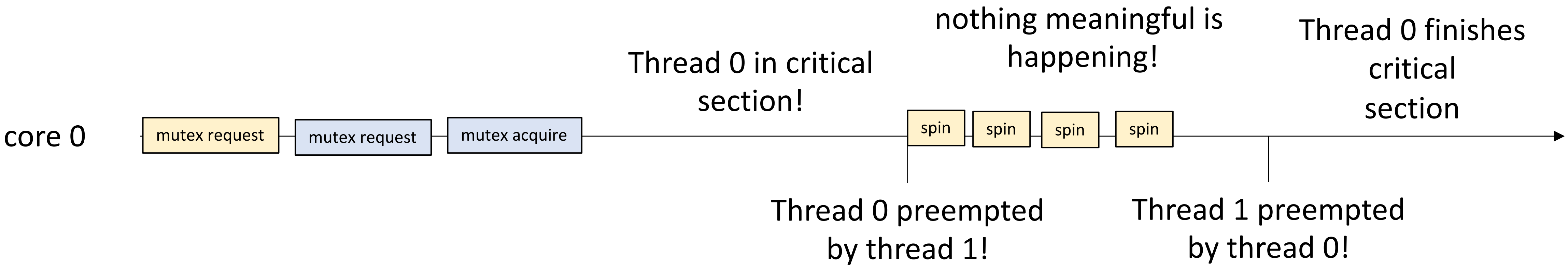
Say threads 0 and 1 are executing concurrently



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**

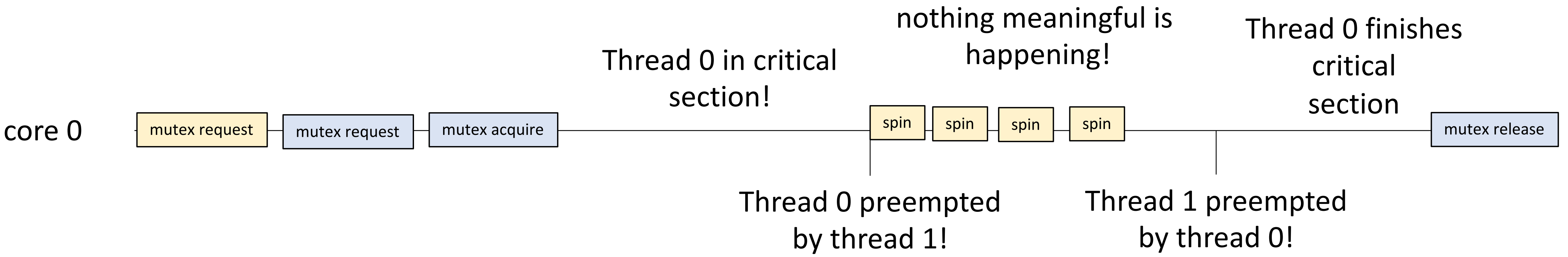
Say threads 0 and 1 are executing concurrently



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**

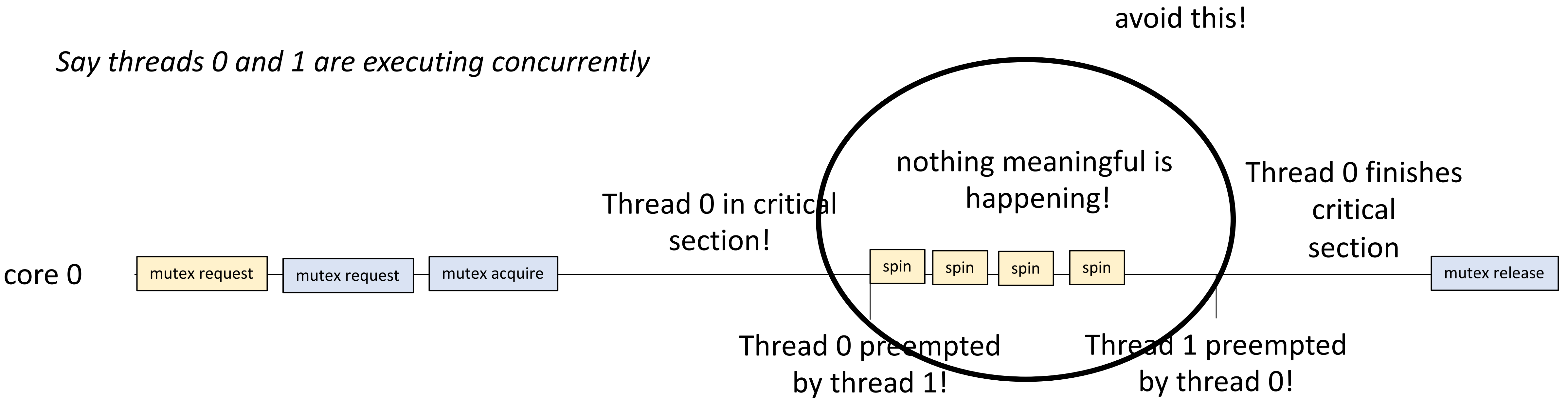
Say threads 0 and 1 are executing concurrently



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**

Say threads 0 and 1 are executing concurrently



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);
  }
}
```

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);
}
```

Example: UI refresh

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

New material!

Schedule

- Reader-Write (RW) mutexes
- Hierarchical aware locks
- Impact of data-races

Reader-Writer Mutex

Global variable: `int tylers_account`

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

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

Reader-Writer Mutex

Global variable: `int tylers_account`

But what happens more frequently than either of those things?

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

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

Reader-Writer Mutex

Global variable: `int tylers_account`

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

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

which of these operations can safely be executed concurrently?

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

But what happens more frequently than either of those things?

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

<u>Different actors accessing it concurrently</u>

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
```

Schedule

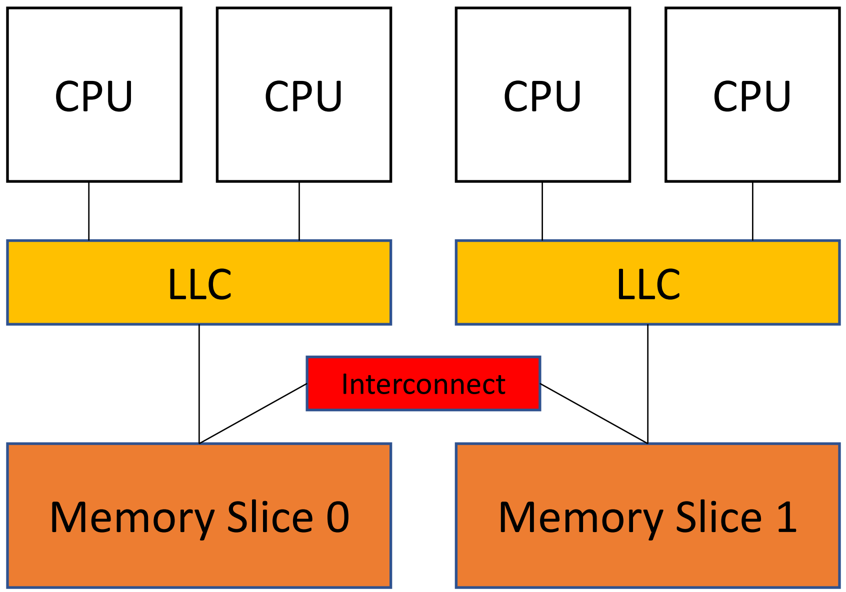
- Reader-Write (RW) mutexes
- Hierarchical aware locks
- Impact of data-races

Optimization: Hierarchical locks

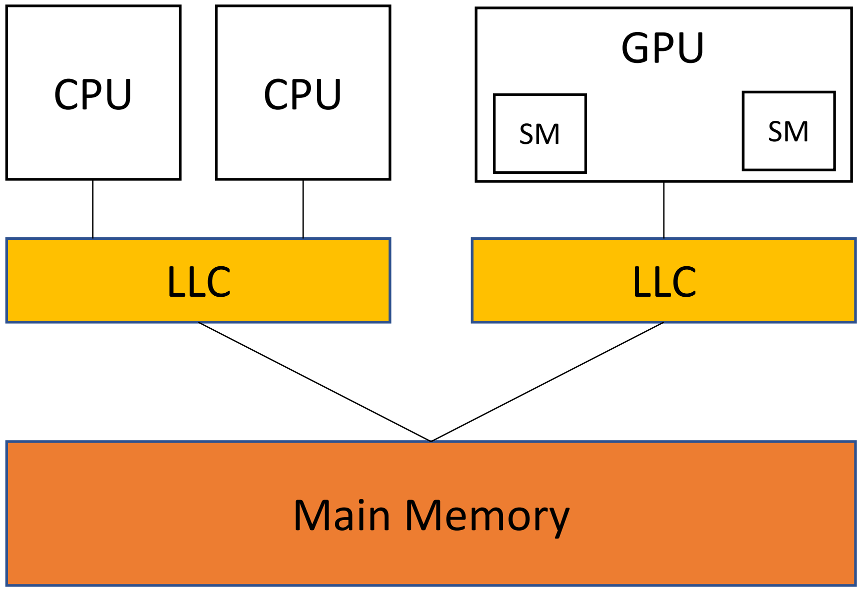
- NUMA (non-uniform memory access) systems
- heterogeneous systems (CPU GPU)

Discrete GPUs communicate through PCIE

For example: Large server nodes

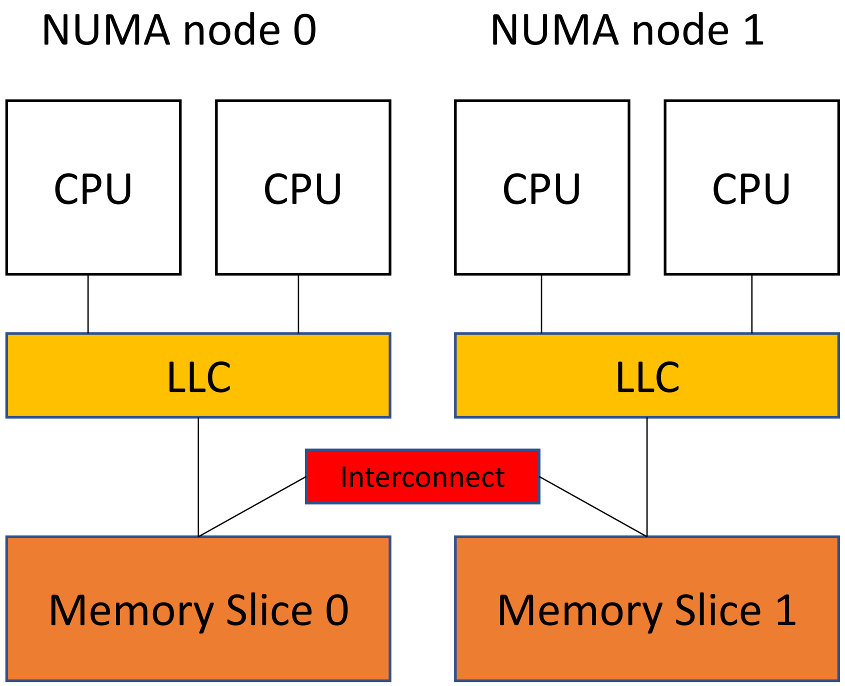


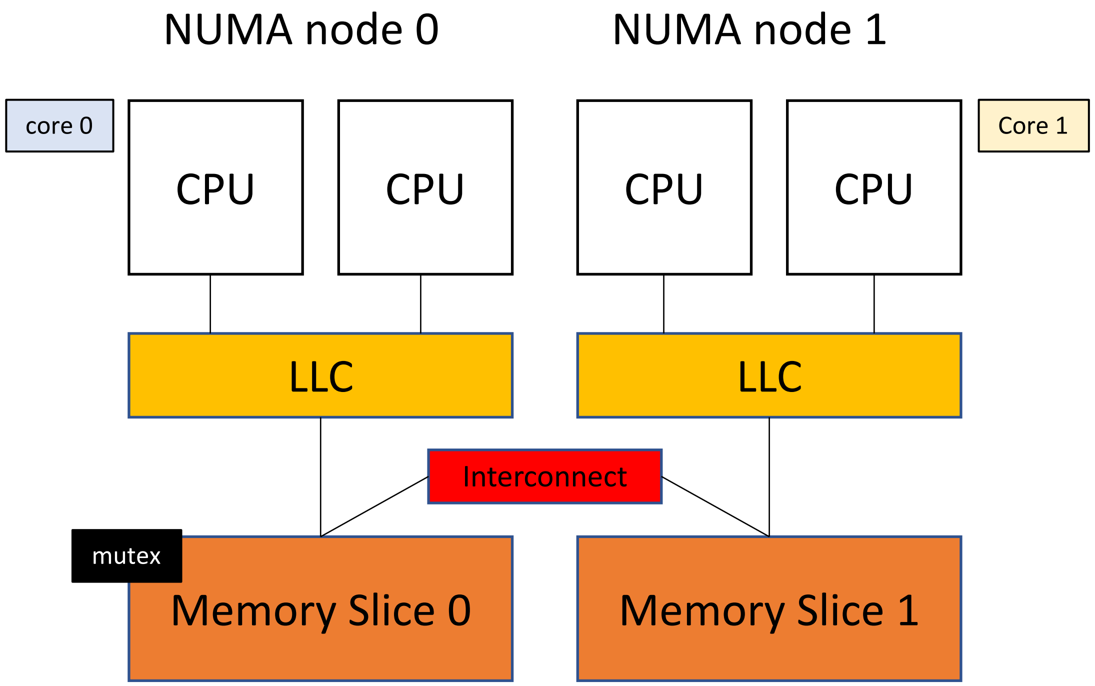
For example: SoCs like Iphone

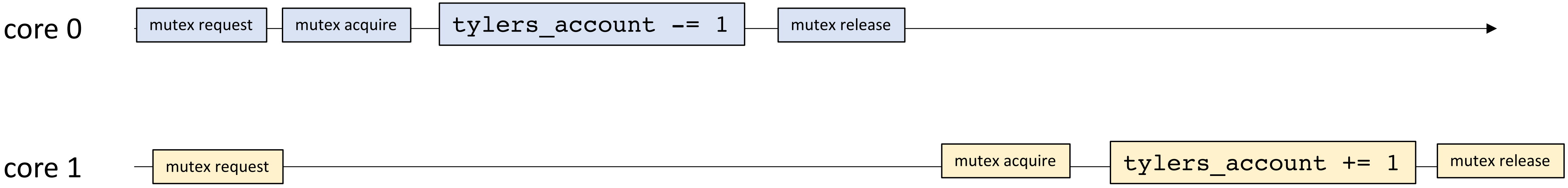


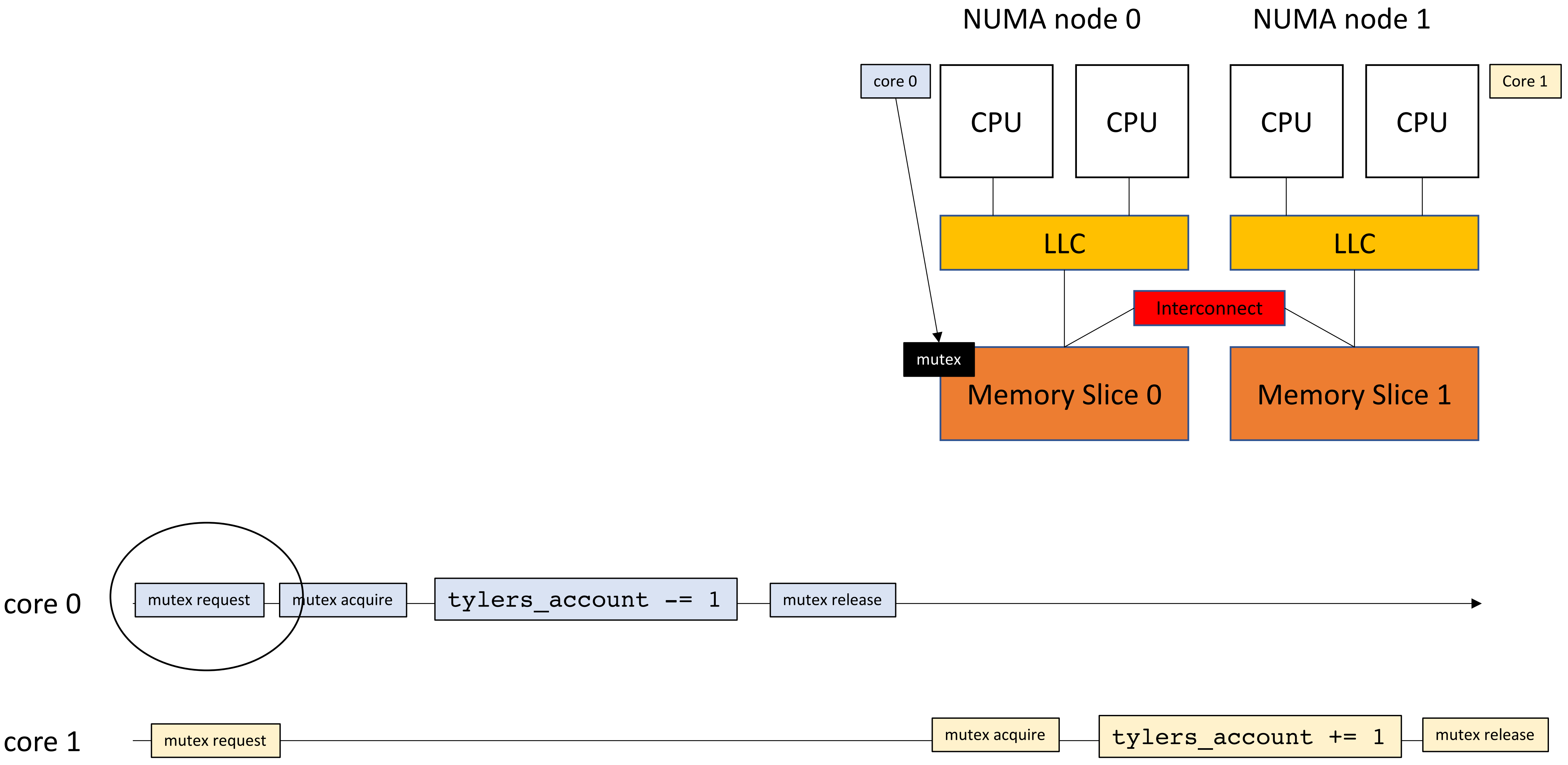
Optimization: Hierarchical locks

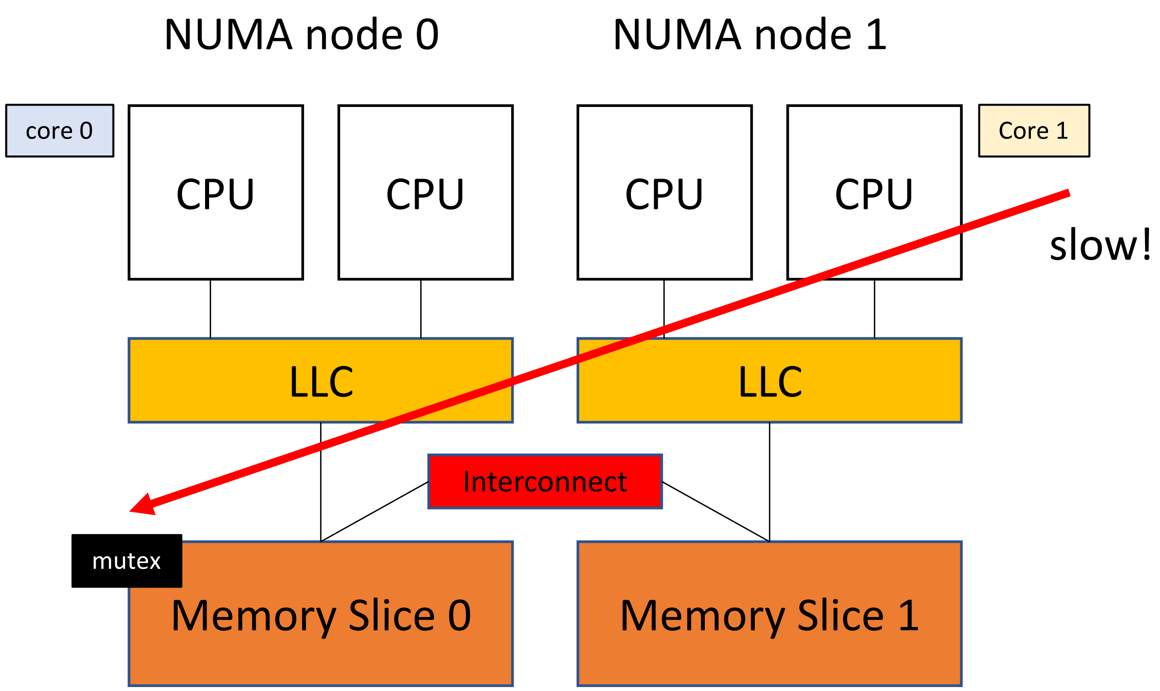
- Any sort of communication is very expensive:
  - Spinning triggers expensive coherence protocols.
  - cache flushes between NUMA nodes is expensive (transferring memory between critical sections)

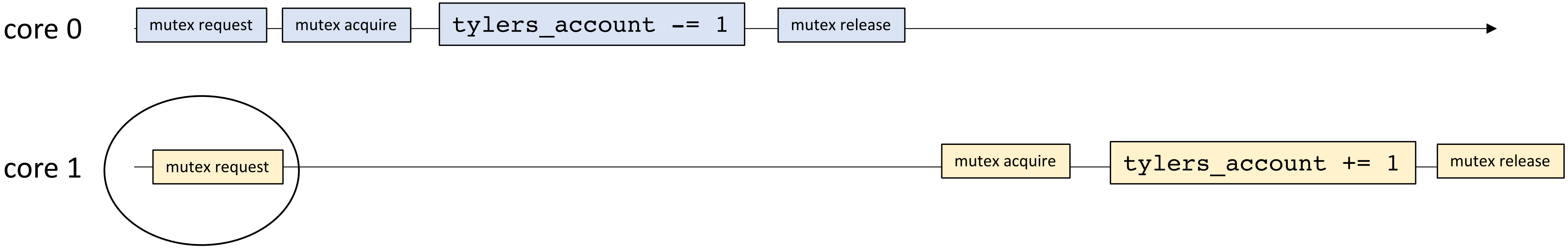


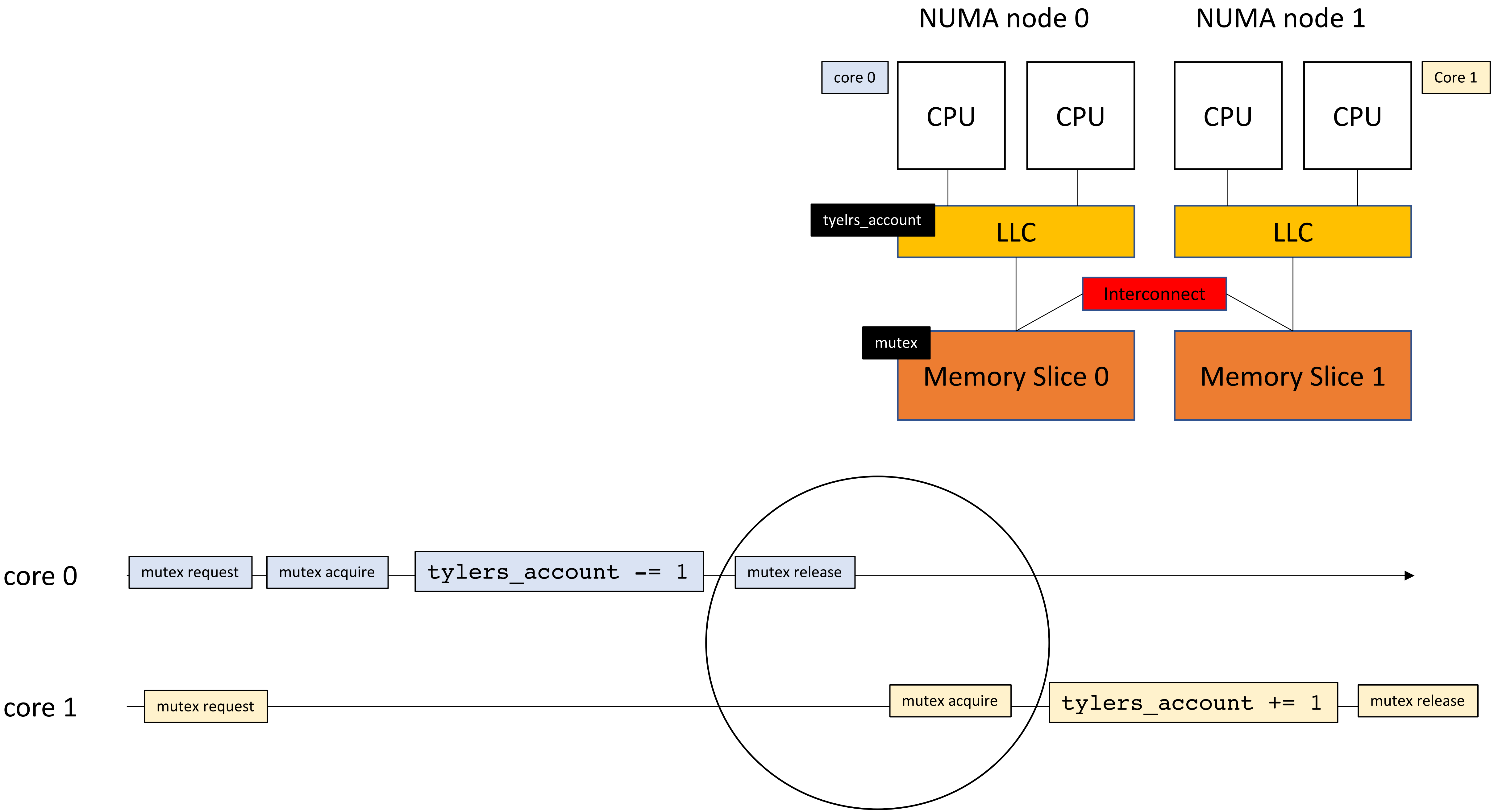

NUMA node 0
NUMA node 1
core 0
CPU
CPU
CPU
CPU
Core 1
LLC
LLC
Interconnect
mutex
Memory Slice 0
Memory Slice 1

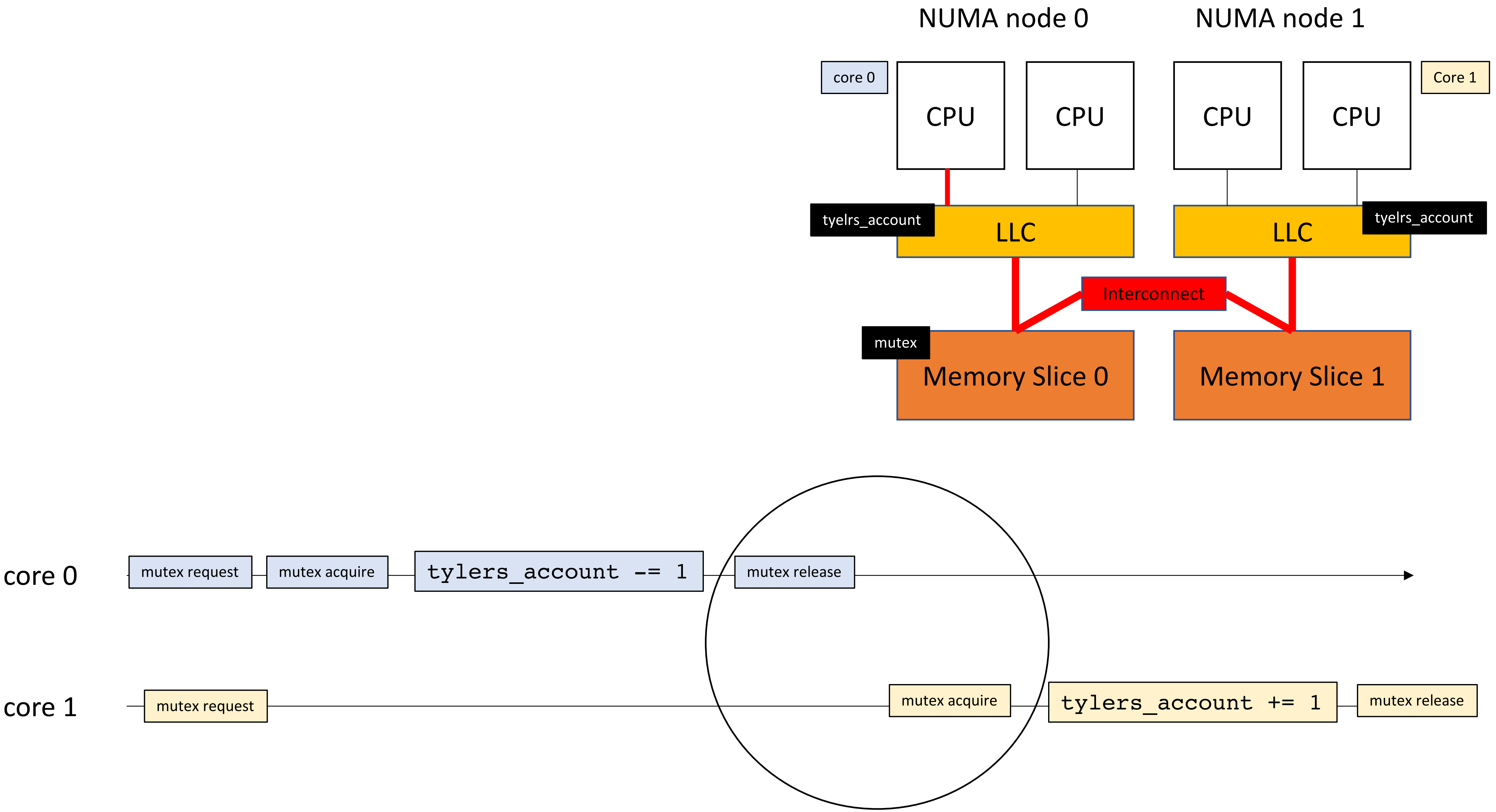

core 0
mutex request
mutex acquire
tylers_account -= 1
mutex release
core 1
mutex request
mutex acquire
tylers_account += 1
mutex release

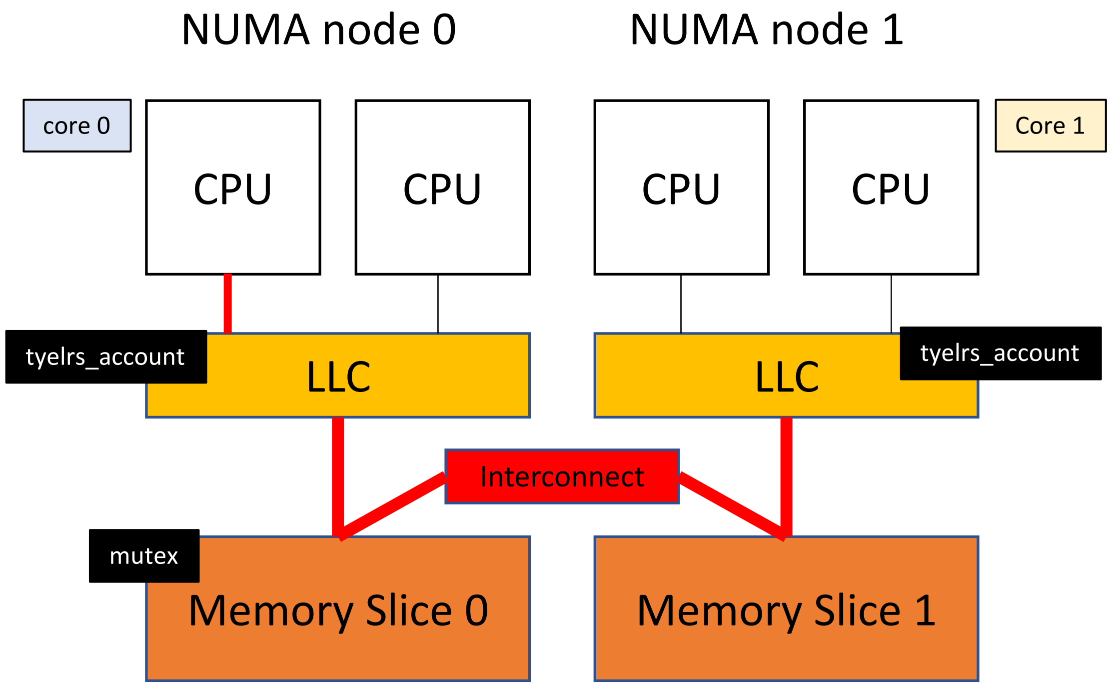

NUMA node 0
NUMA node 1
core 0
CPU
CPU
CPU
CPU
Core 1
LLC
LLC
Interconnect
mutex
Memory Slice 0
Memory Slice 1
core 0
mutex request
mutex acquire
tylers_account -= 1
mutex release
core 1
mutex request
mutex acquire
tylers_account += 1
mutex release


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

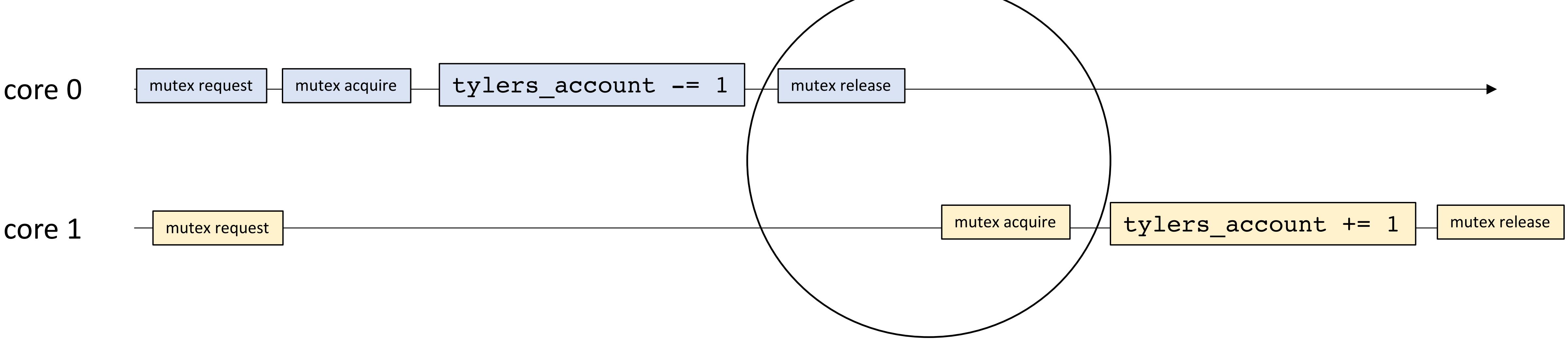

core 0
mutex request
mutex acquire
tylers_account -= 1
mutex release
core 1
mutex request
mutex acquire
tylers_account += 1
mutex release

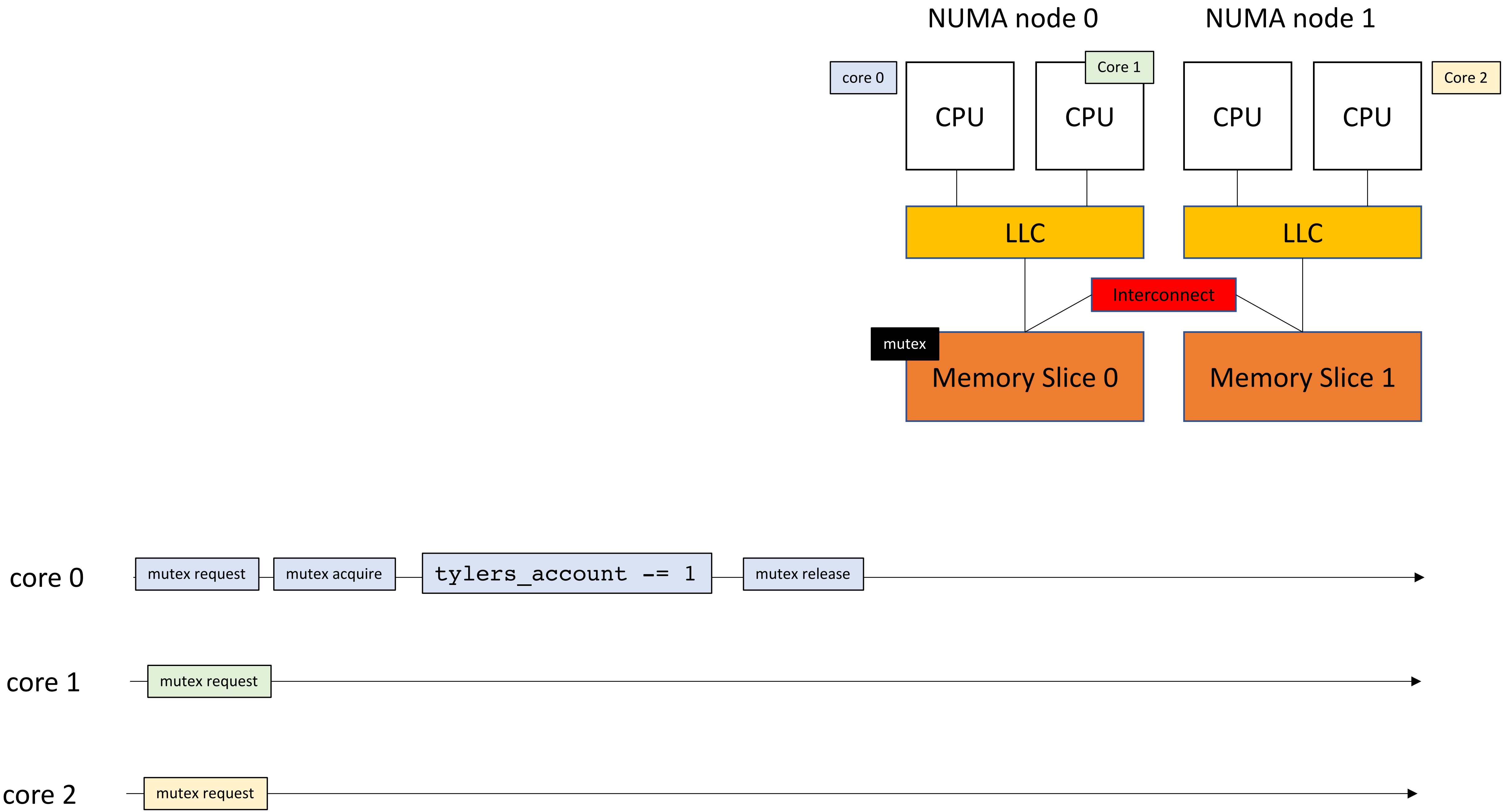

NUMA node 0
NUMA node 1
core 0
CPU
CPU
CPU
CPU
Core 1
tyelrs_account
LLC
LLC
Interconnect
mutex
Memory Slice 0
Memory Slice 1
core 0
mutex request
mutex acquire
tylers_account -= 1
mutex release
core 1
mutex request
mutex acquire
tylers_account += 1
mutex release

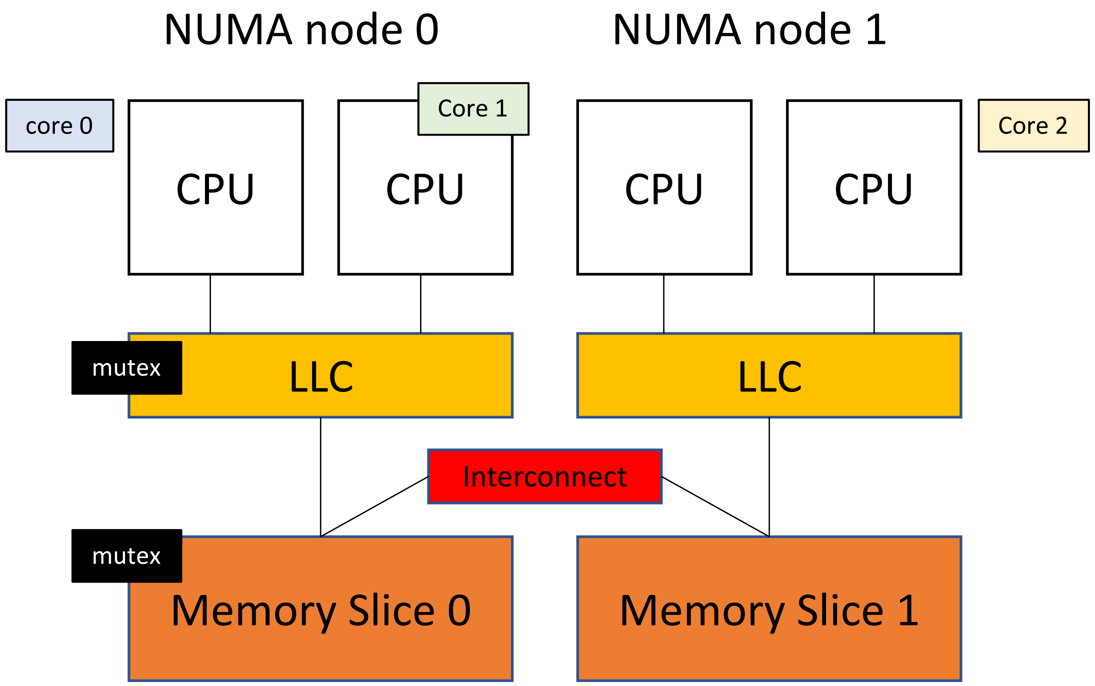

NUMA node 0
NUMA node 1
core 0
CPU
CPU
CPU
CPU
Core 1
tyelrs_account
LLC
LLC
tyelrs_account
Interconnect
mutex
Memory Slice 0
Memory Slice 1
core 0
mutex request
mutex acquire
tylers_account -= 1
mutex release
core 1
mutex request
mutex acquire
tylers_account += 1
mutex release

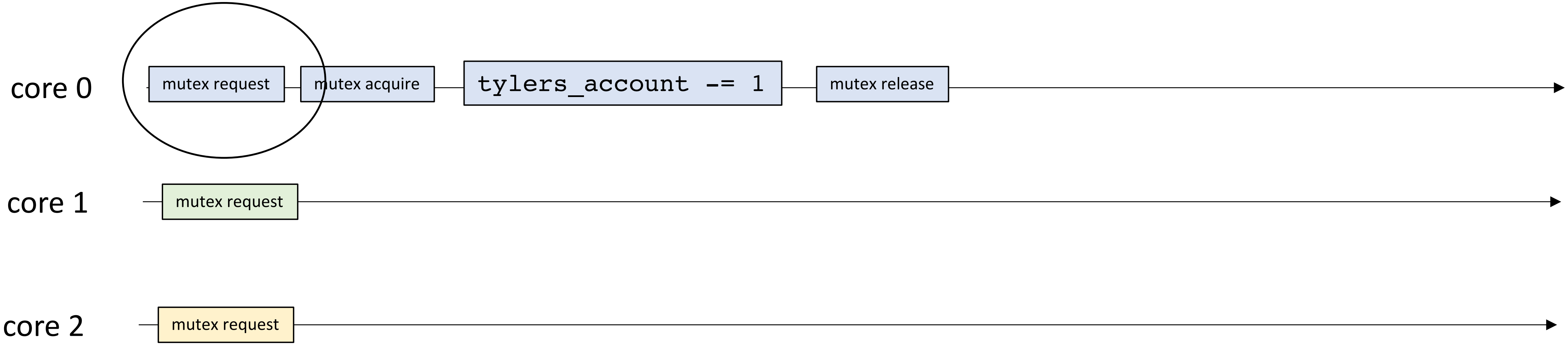


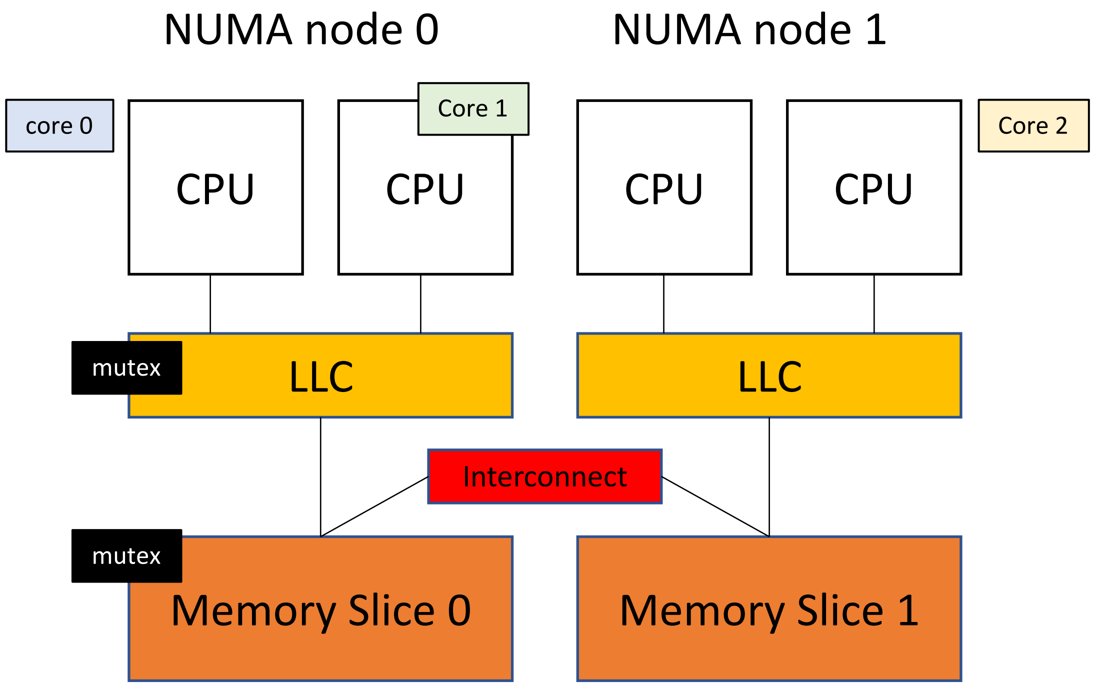
What if there is tons of data here?



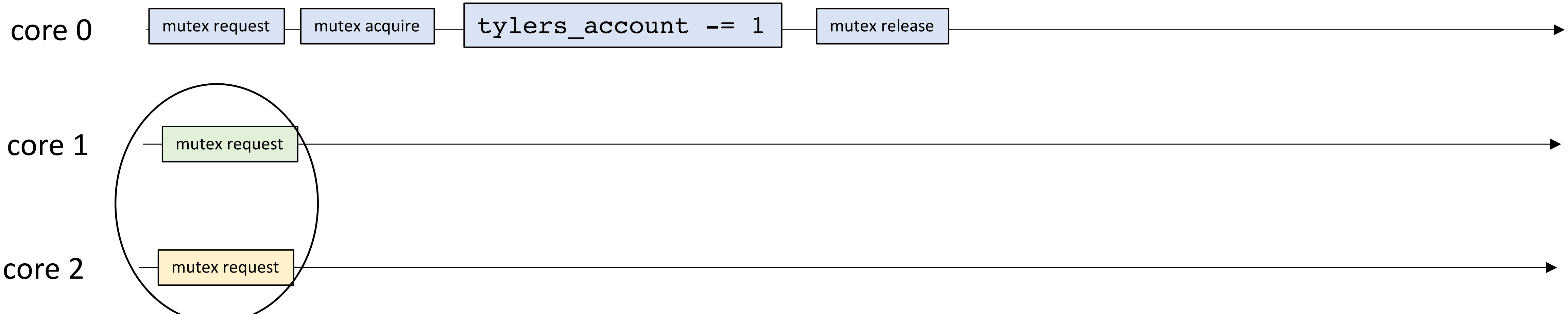

NUMA node 0
NUMA node 1
core 0
Core 1
Core 2
CPU
CPU
CPU
CPU
LLC
LLC
Interconnect
mutex
Memory Slice 0
Memory Slice 1
core 0
mutex request
mutex acquire
tylers_account -= 1
mutex release
core 1
mutex request
core 2
mutex request

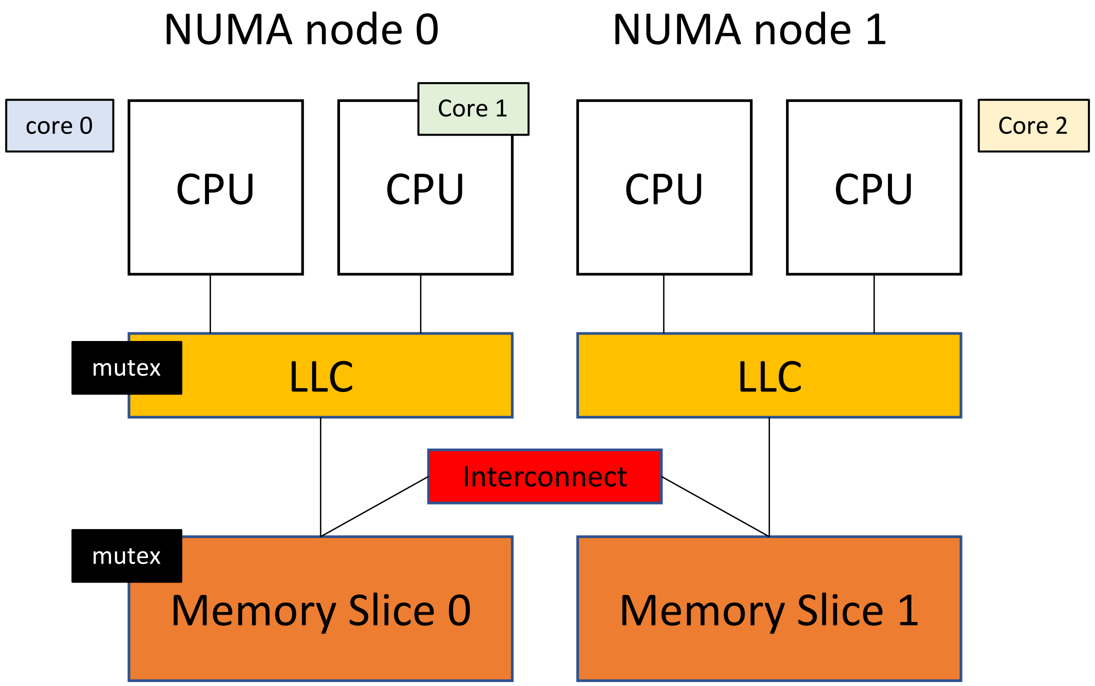

NUMA node 0
NUMA node 1
core 0
Core 1
Core 2
CPU
CPU
CPU
CPU
mutex
LLC
LLC
Interconnect
mutex
Memory Slice 0
Memory Slice 1

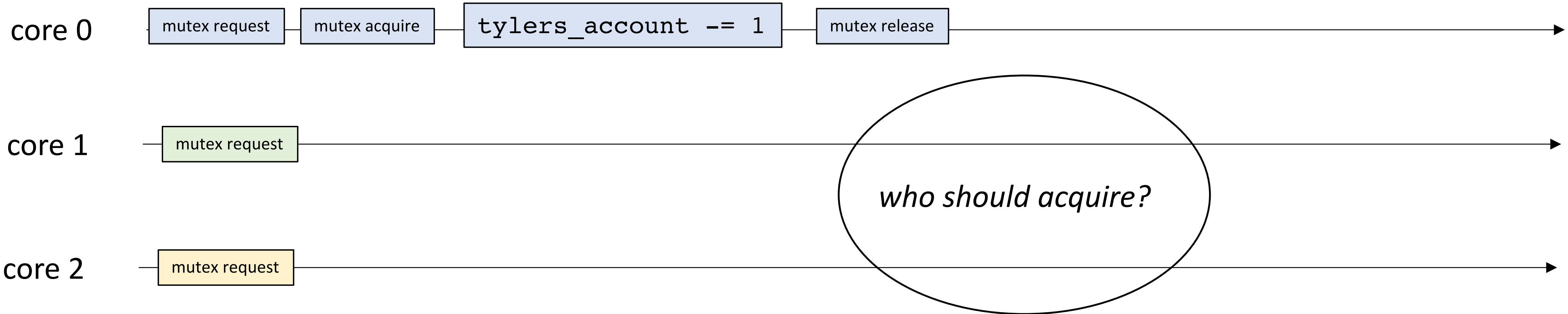

core 0
mutex request
mutex acquire
tylers_account -= 1
mutex release
core 1
mutex request
core 2
mutex request



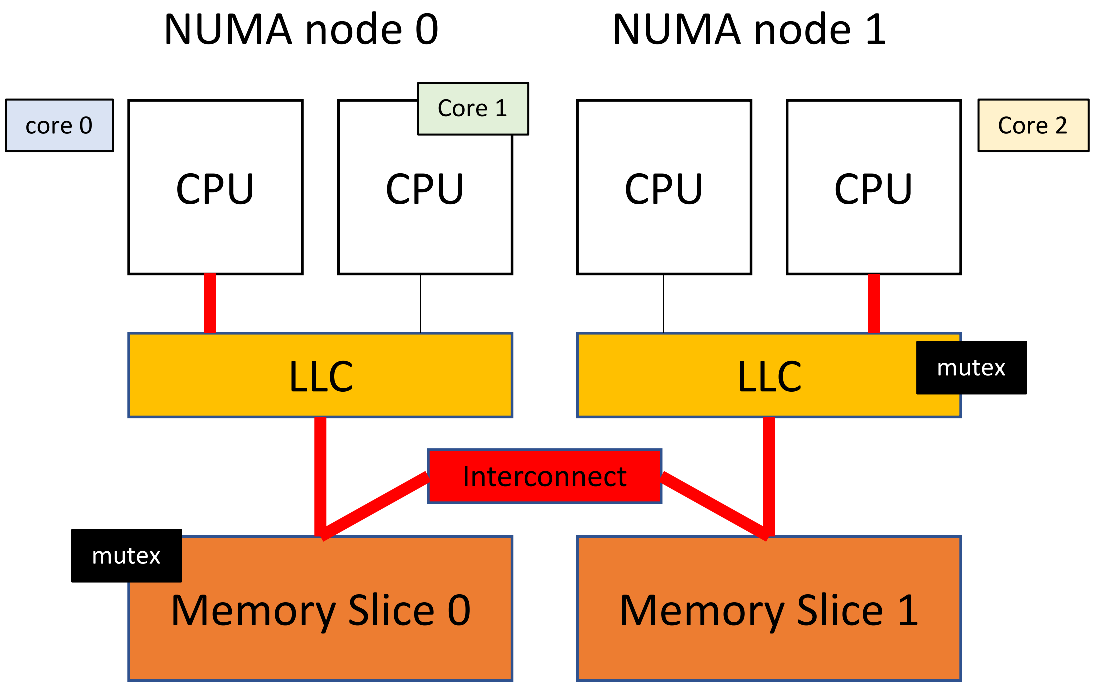
Ideally core 2 accesses the mutex less frequently than core 1

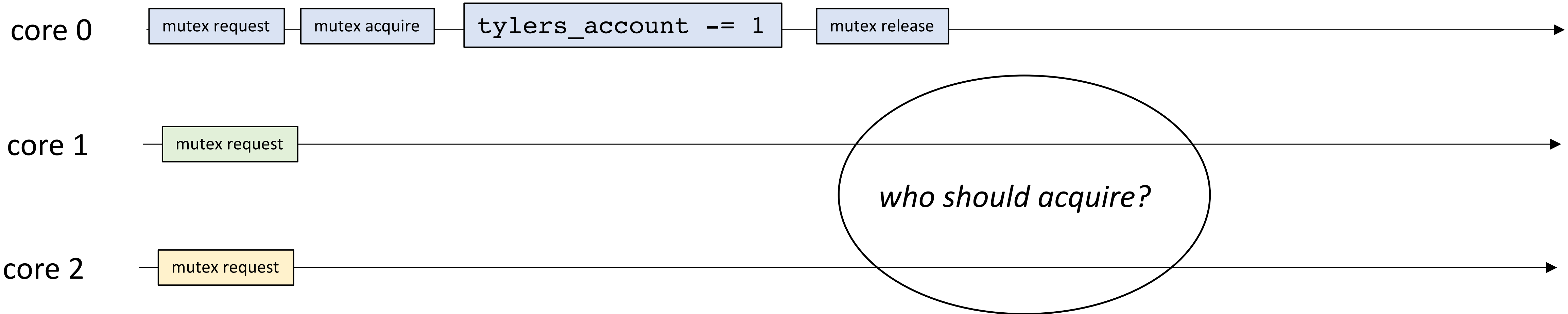



NUMA node 0
NUMA node 1
core 0
Core 1
Core 2
CPU
CPU
CPU
CPU
mutex
LLC
LLC
Interconnect
mutex
Memory Slice 0
Memory Slice 1

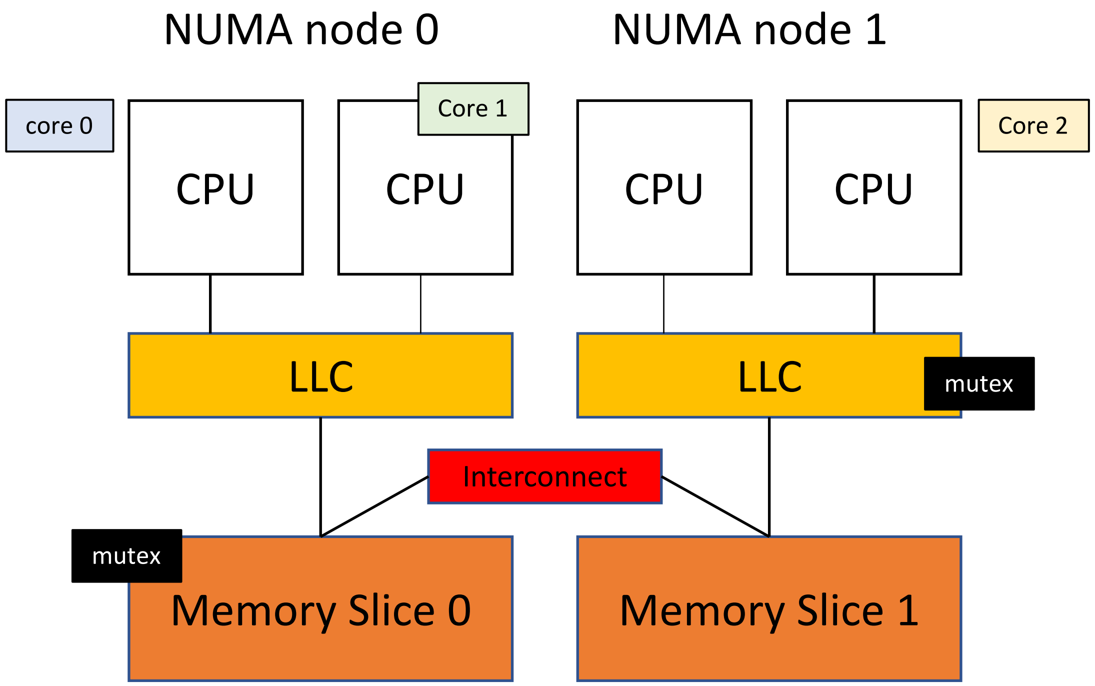

core 0
mutex request
mutex acquire
tylers_account -= 1
mutex release
core 1
mutex request
who should acquire?
core 2
mutex request

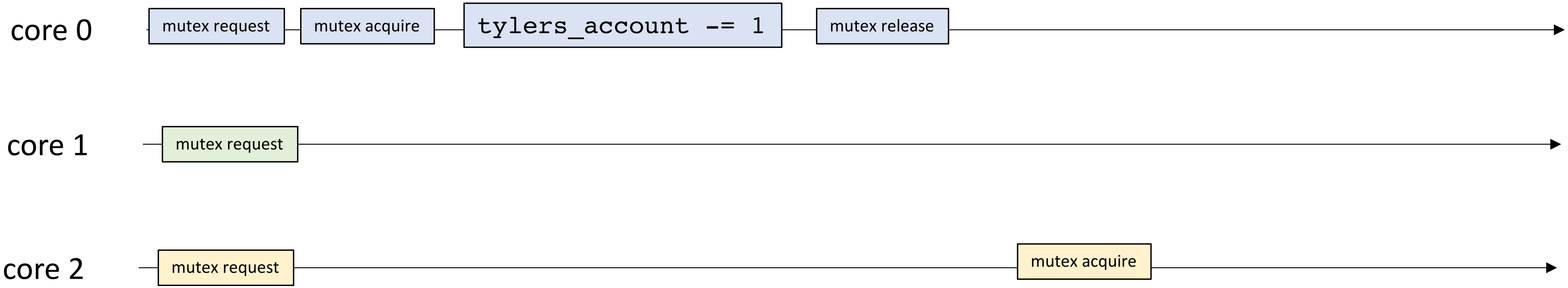
If core 2 acquires first communication must go through the interconnect



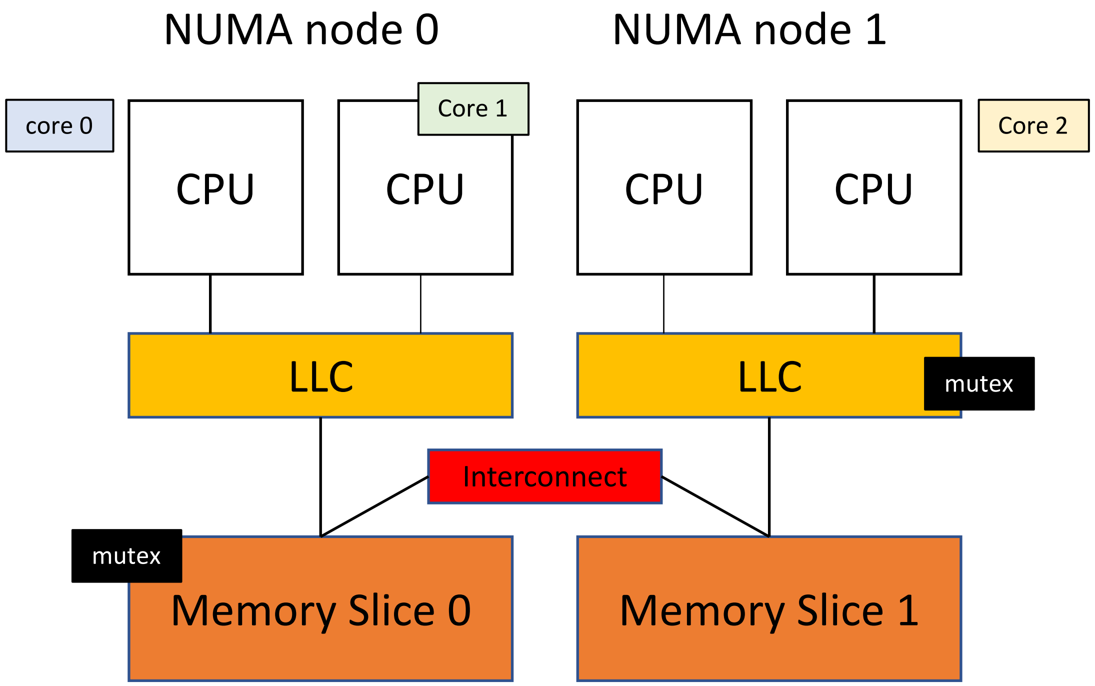


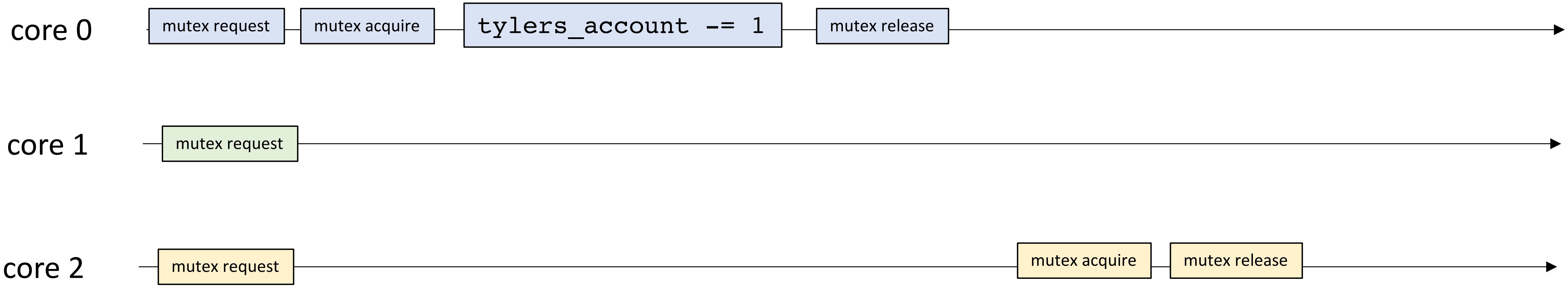
If core 2 acquires first communication must go through the interconnect

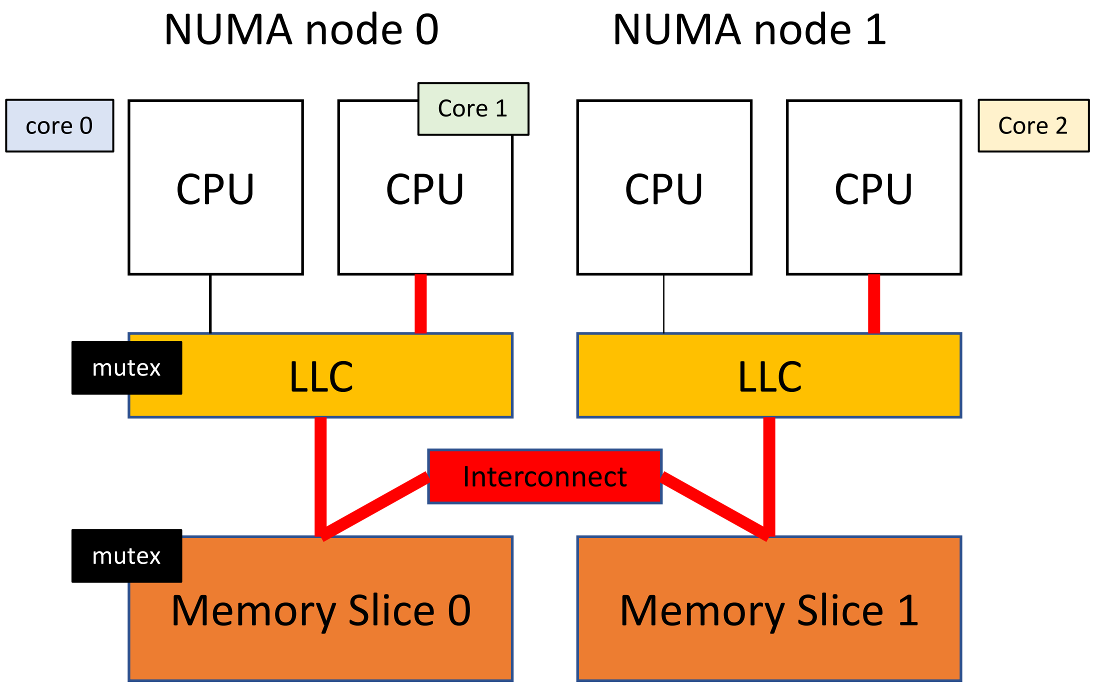




If core 2 acquires first communication must go through the interconnect

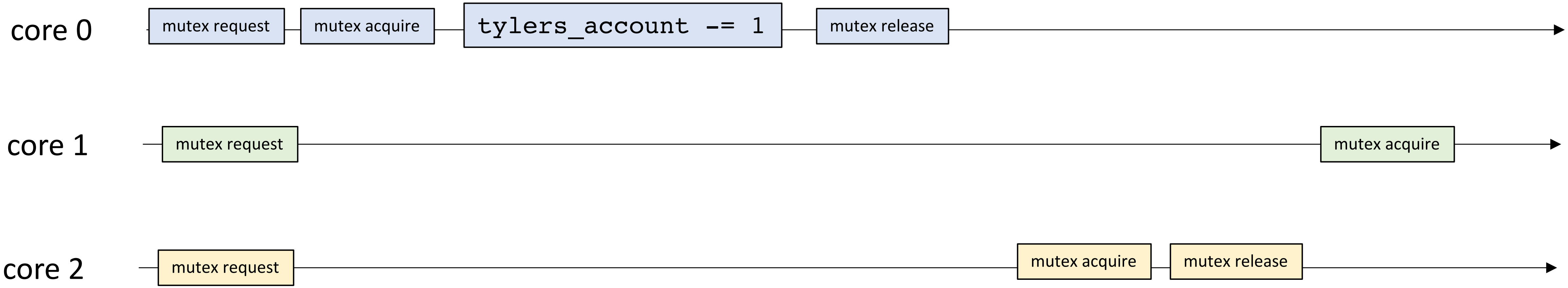






If core 2 acquires first communication must go through the interconnect

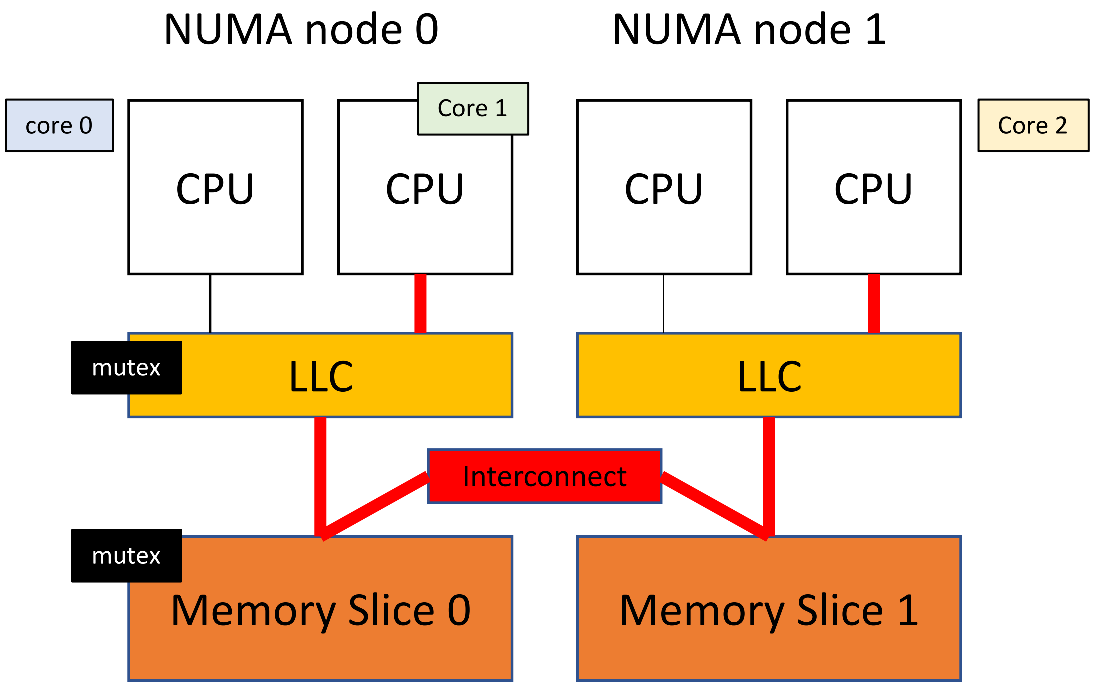
When core 1 finally acquires, it requires another expensive trip through the interconnect

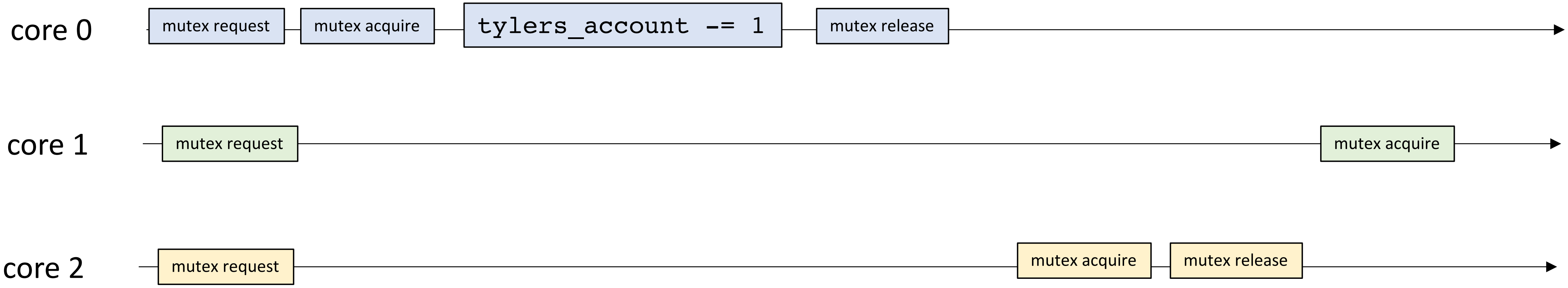


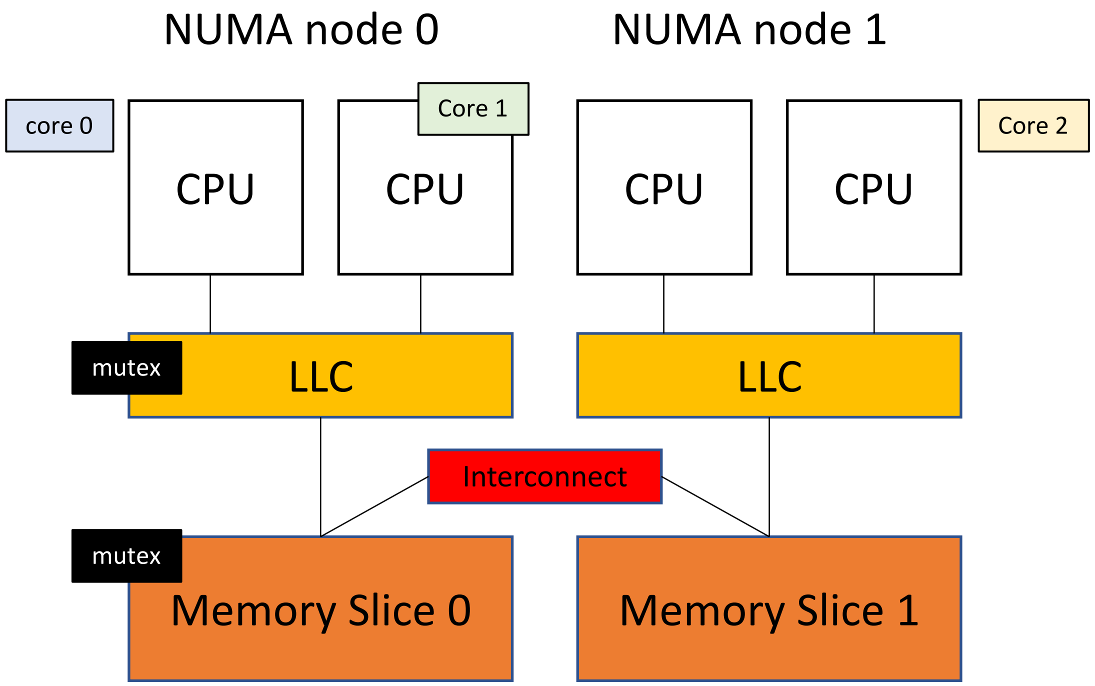
Two trips through the interconnect!!

If core 2 acquires first communication must go through the interconnect

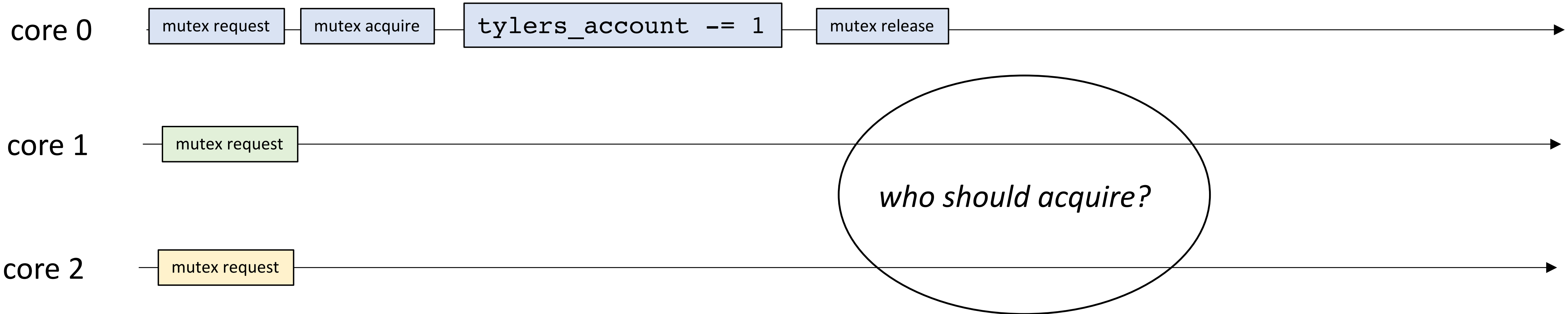
When core 1 finally acquires, it requires another expensive trip through the interconnect



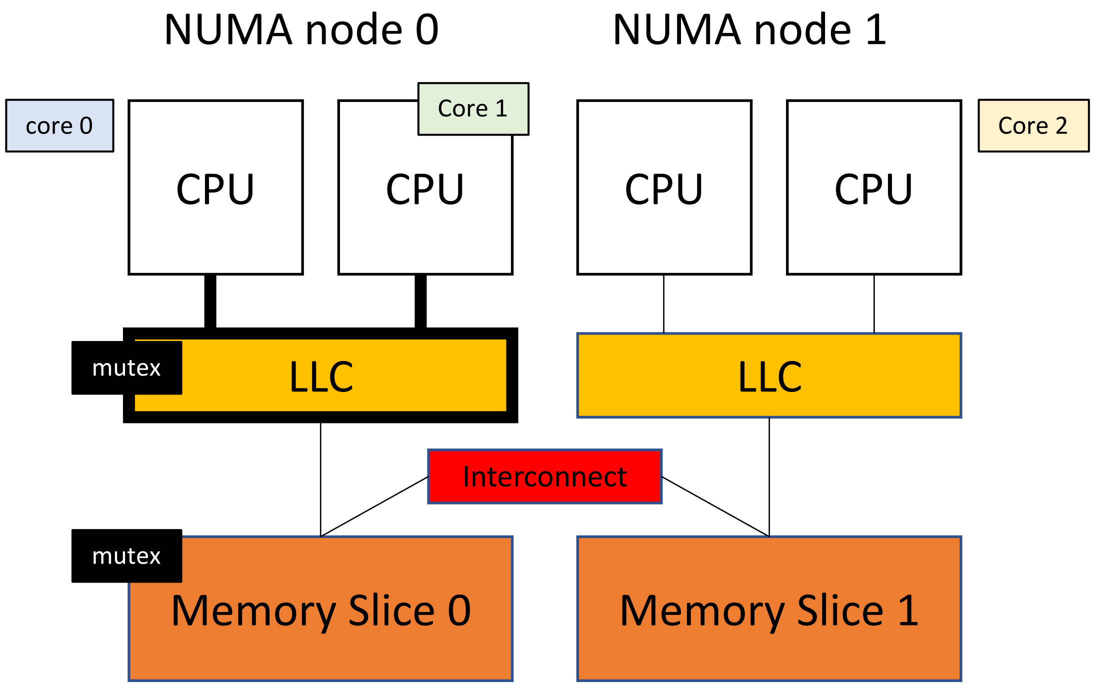


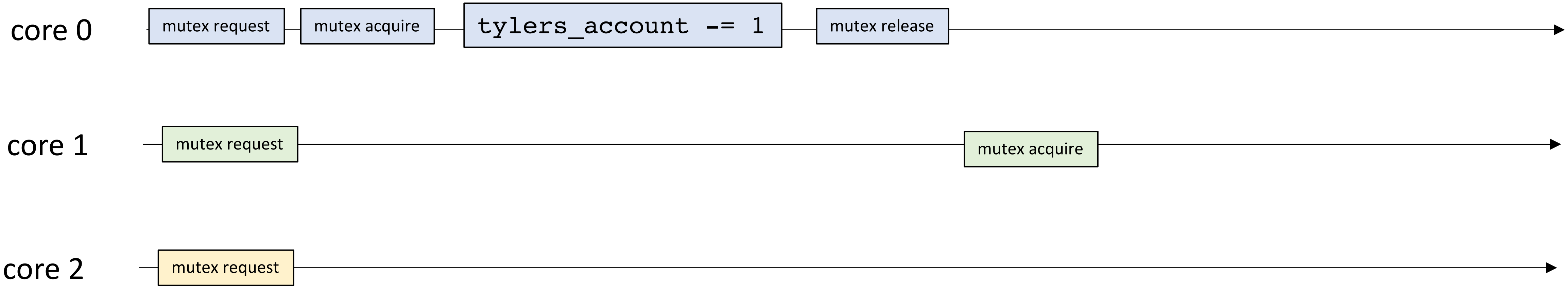


Lets go back in time and make a different decision!



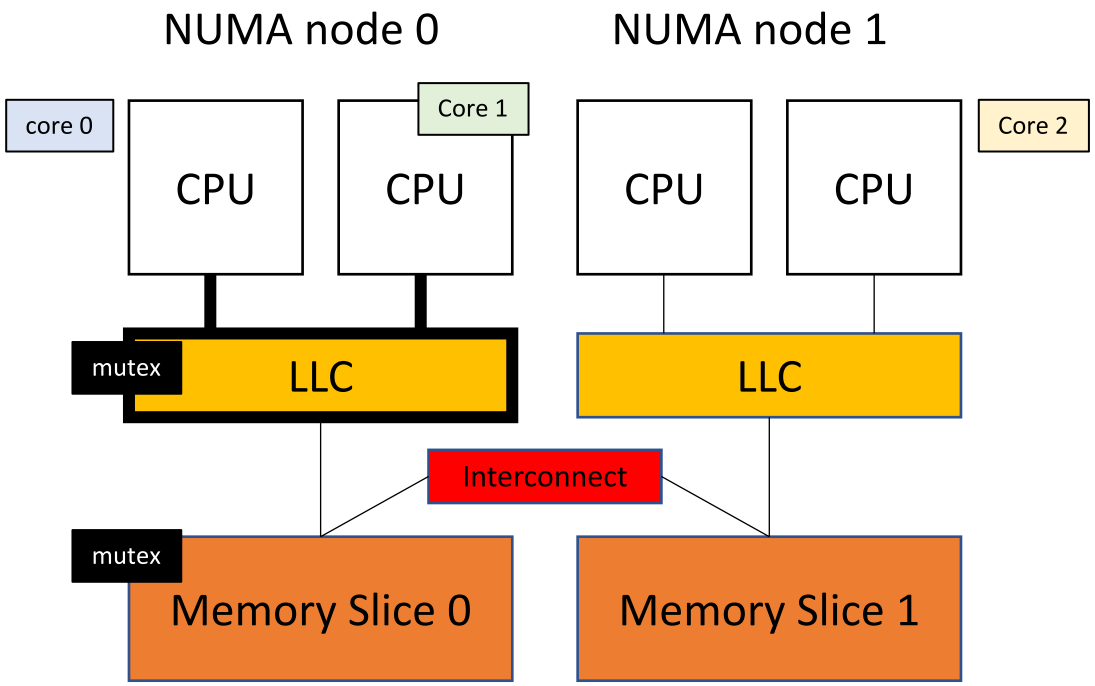
If core 1 acquires first communication can occur through the LLC of NUMA node 0

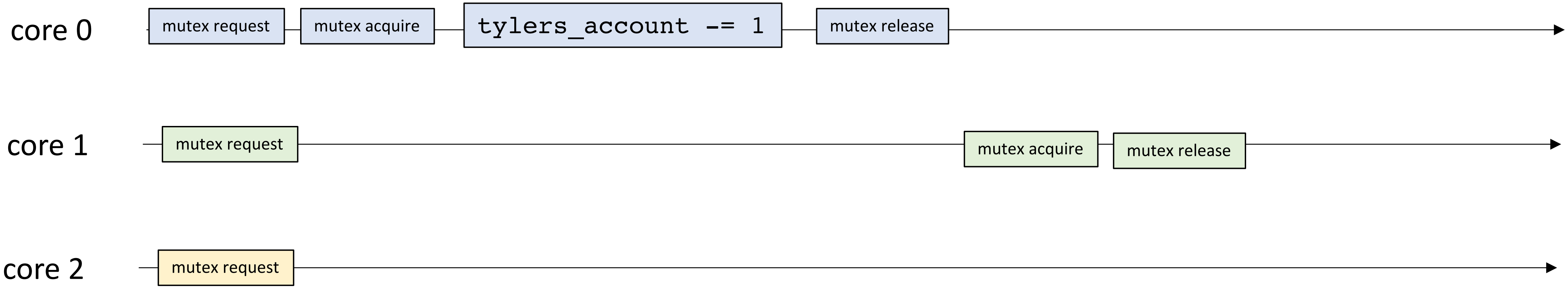




If core 1 acquires first communication can occur through the LLC of NUMA node 0

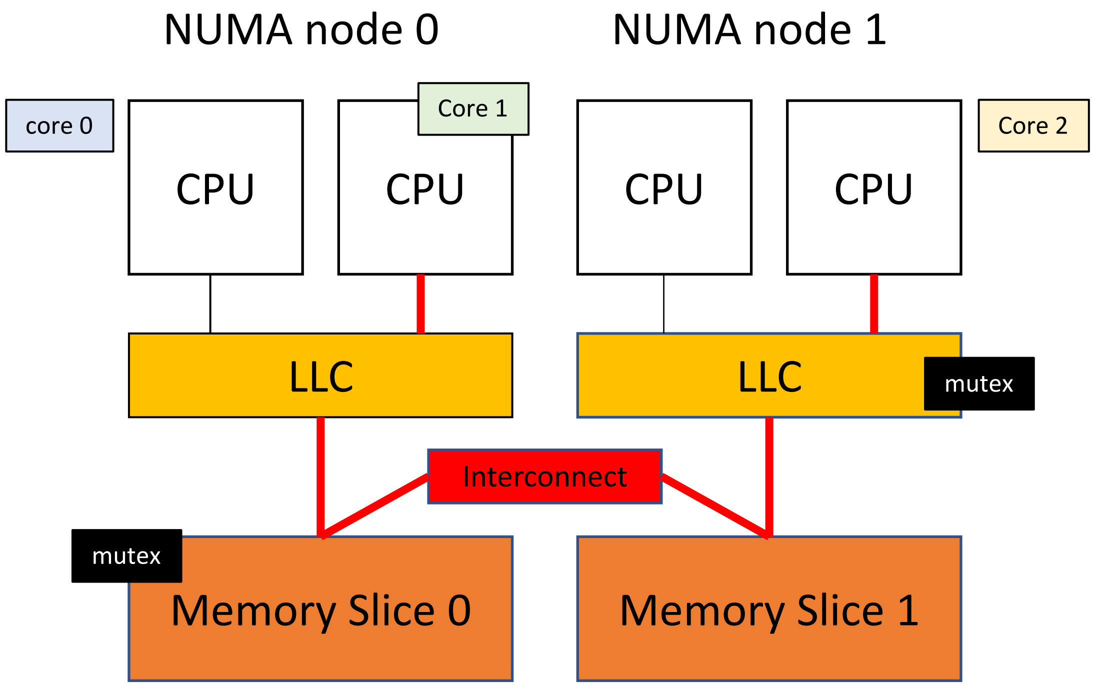
When core 2 finally acquires it requires an expensive trip through the interconnect

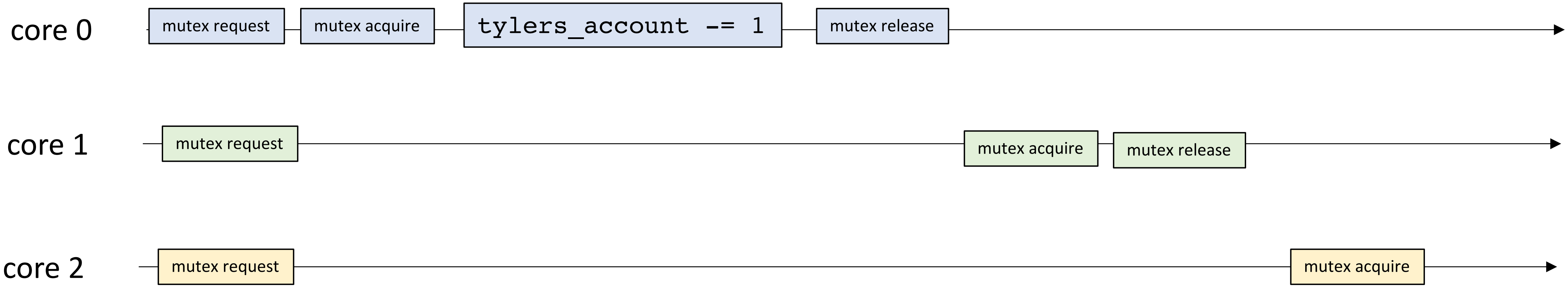




If core 1 acquires first communication can occur through the LLC of NUMA node 0

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

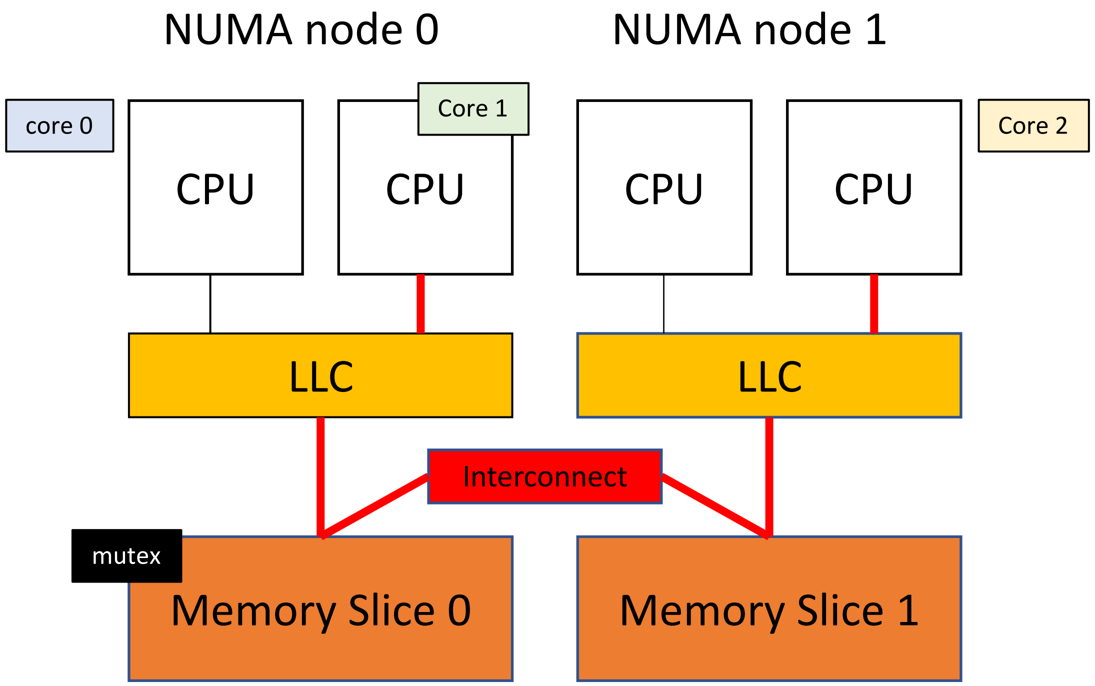


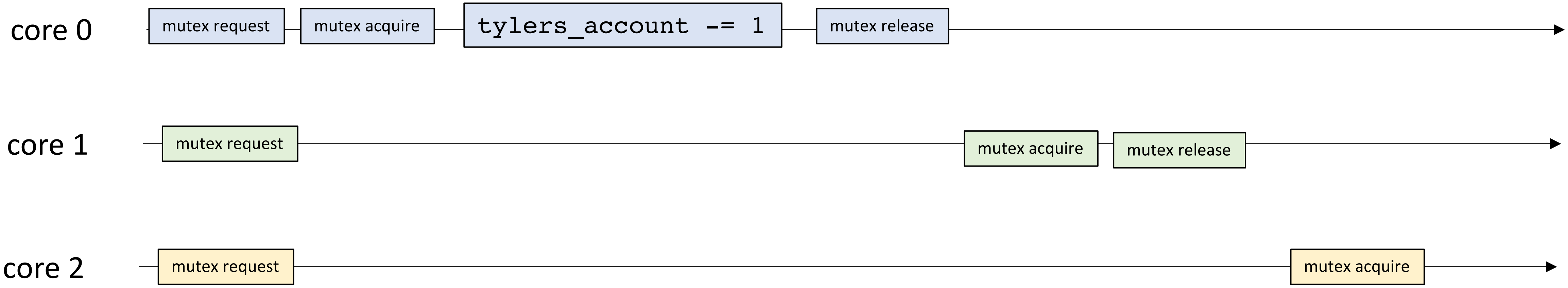


Only 1 trip through the interconnect

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;
  }
}
```

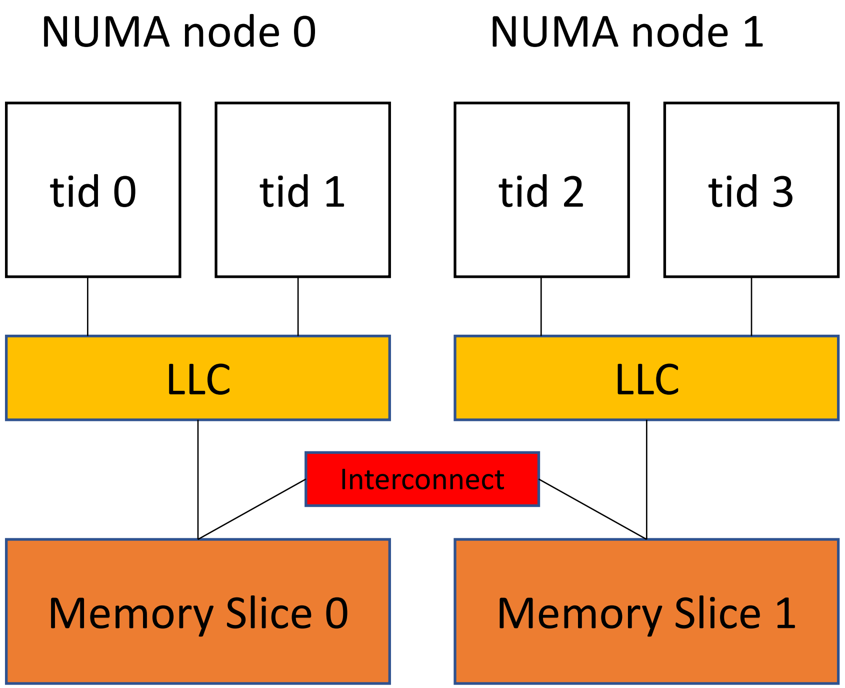
We have a new lock

- 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;
  }
}
```

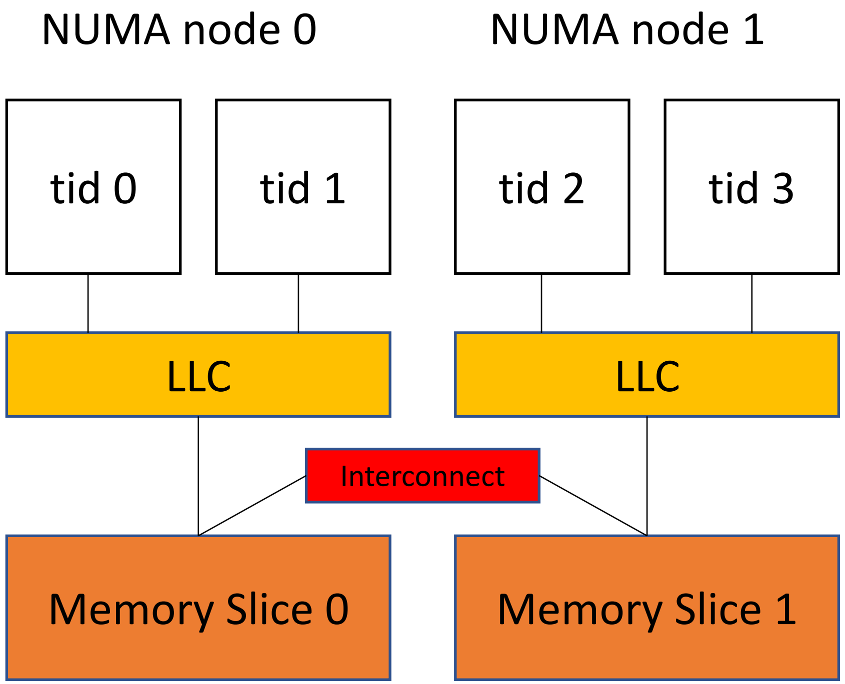
Starvation?

- 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:

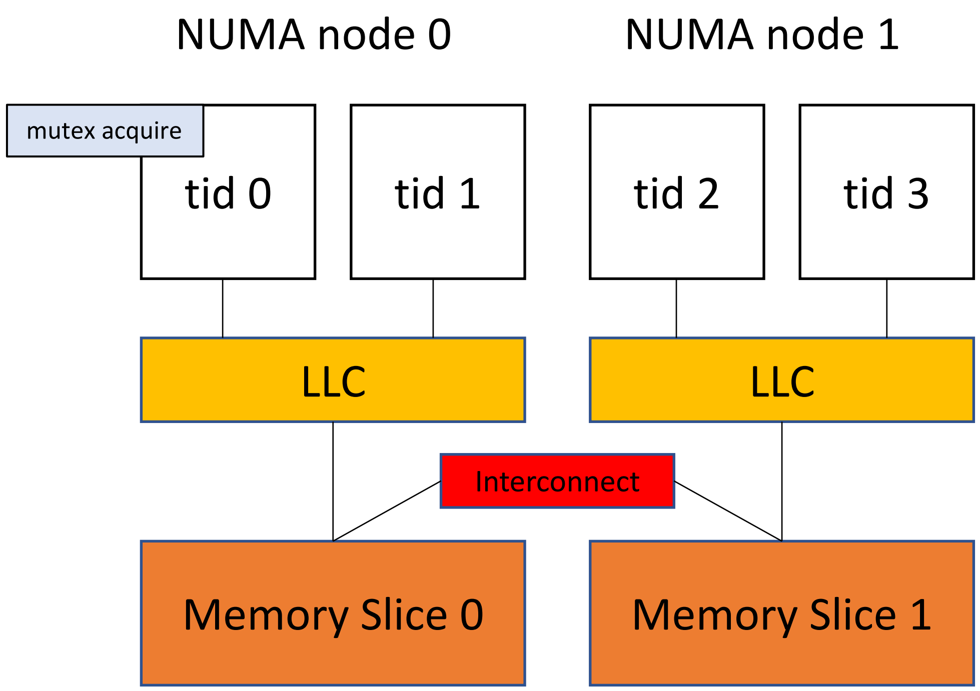
Mutex counter:
Local_Com: 0



Example:

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

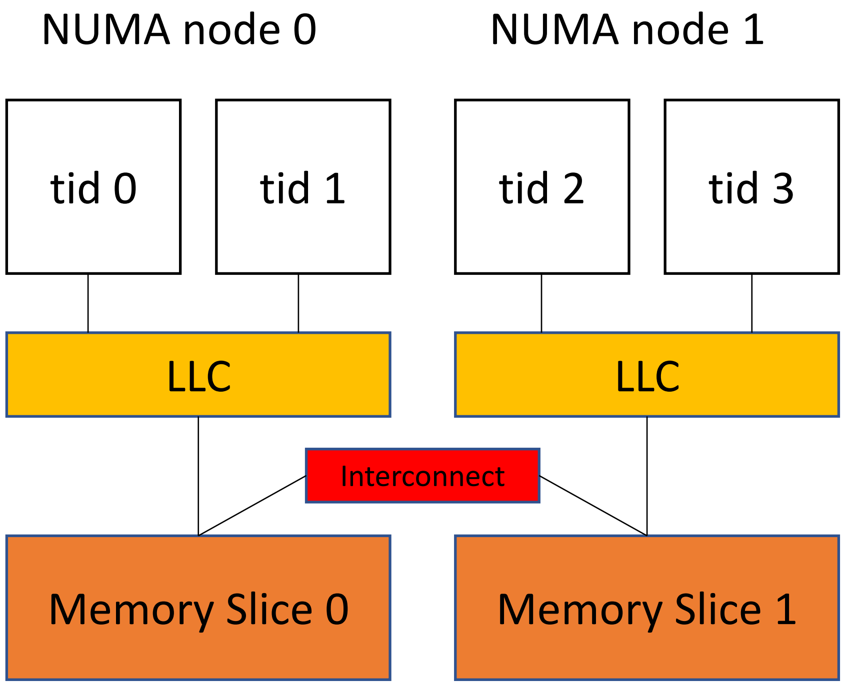
Mutex counter:
Local_Com: 1



Example:

tid 0:
tid 1:
tid 2:

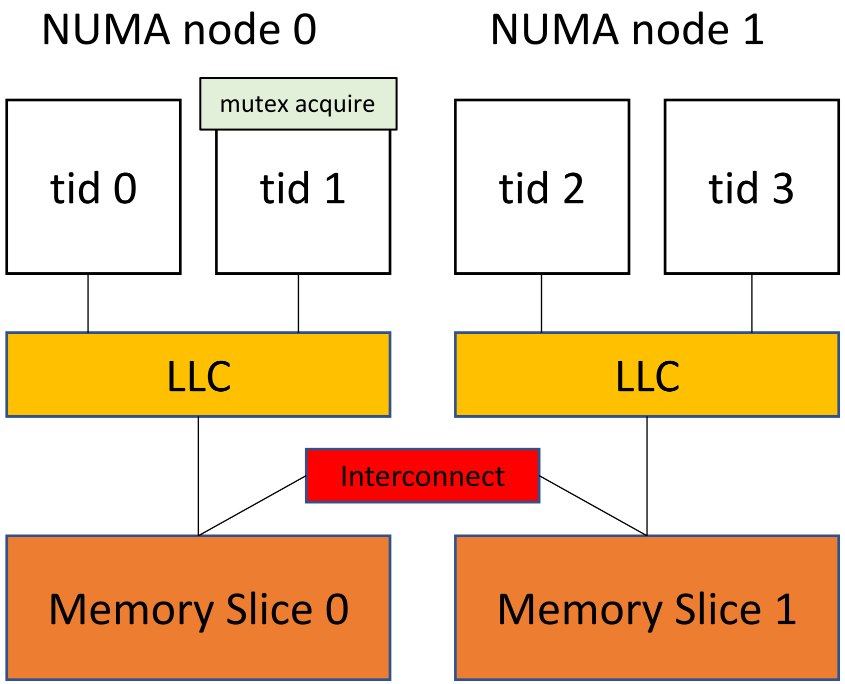
Mutex counter:
Local_Com: 1



Example:

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

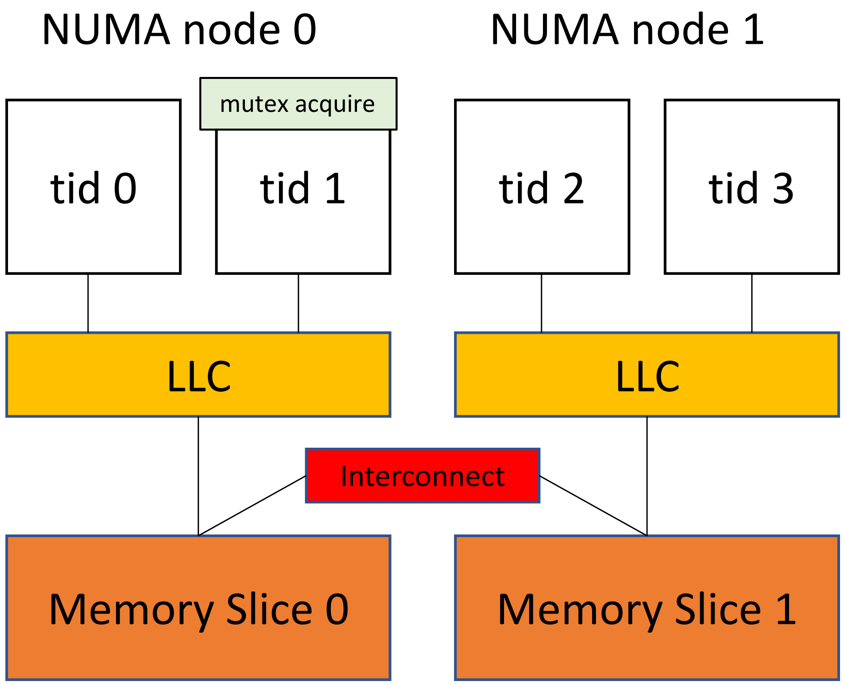
Mutex counter:
Local_Com: 2



Example:

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

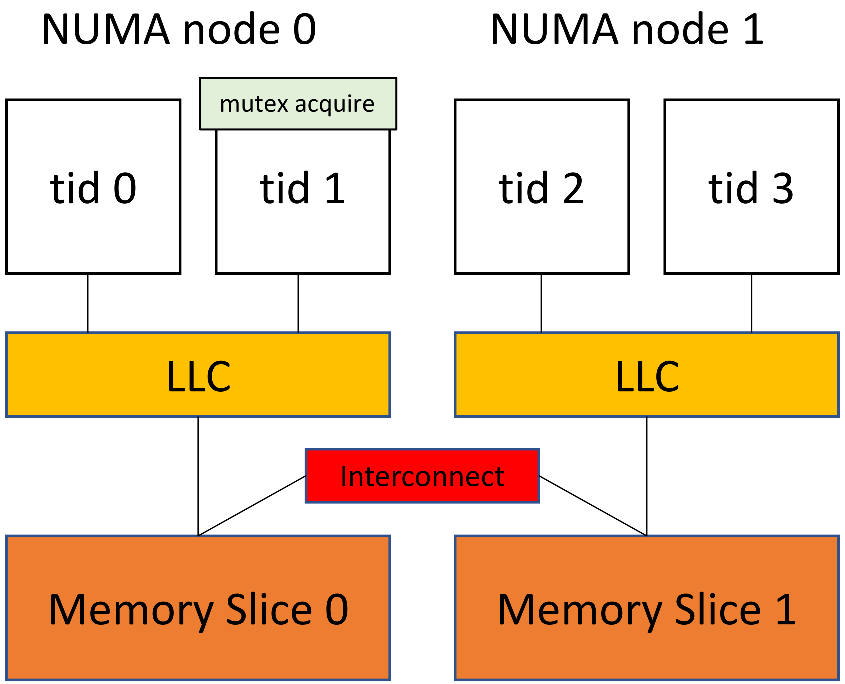
Mutex counter:
Local_Com: 2



Example:

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

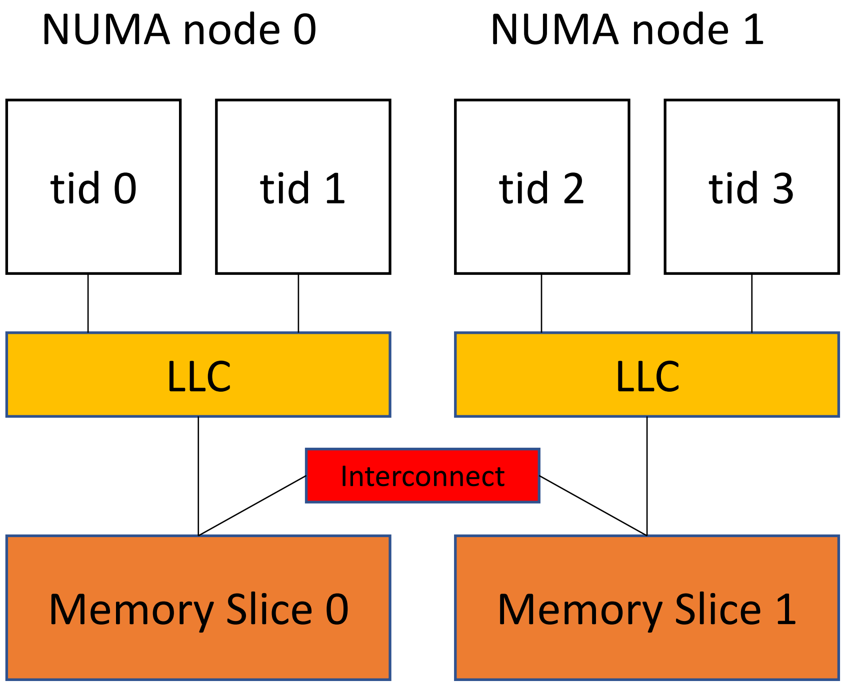
Mutex counter:
Local_Com: 2



Example:

tid 0:
tid 1:
tid 2:

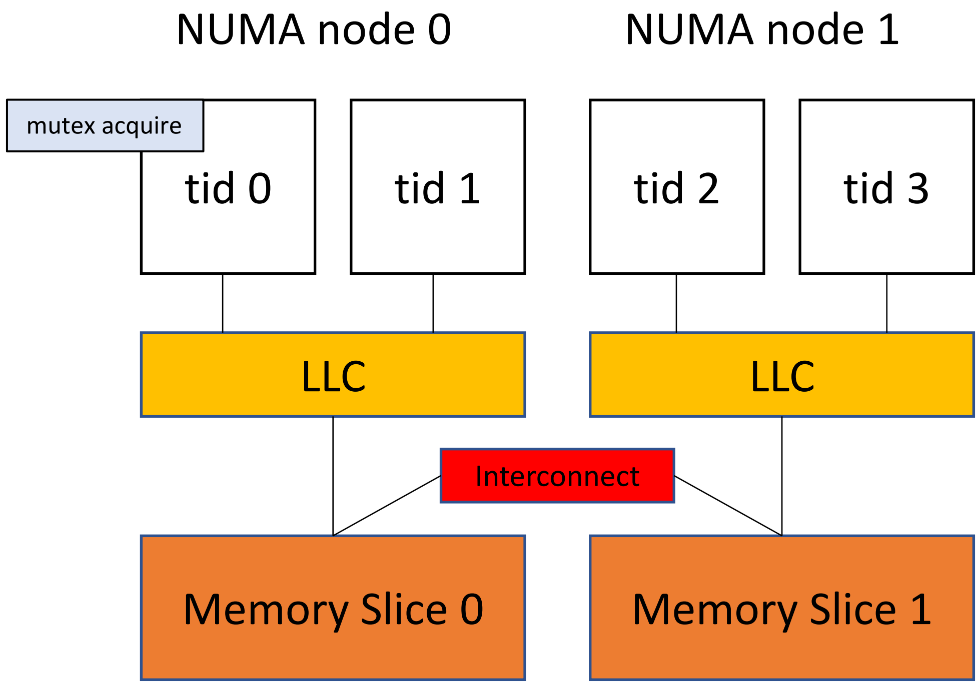
Mutex counter:
Local_Com: 1



Example:

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

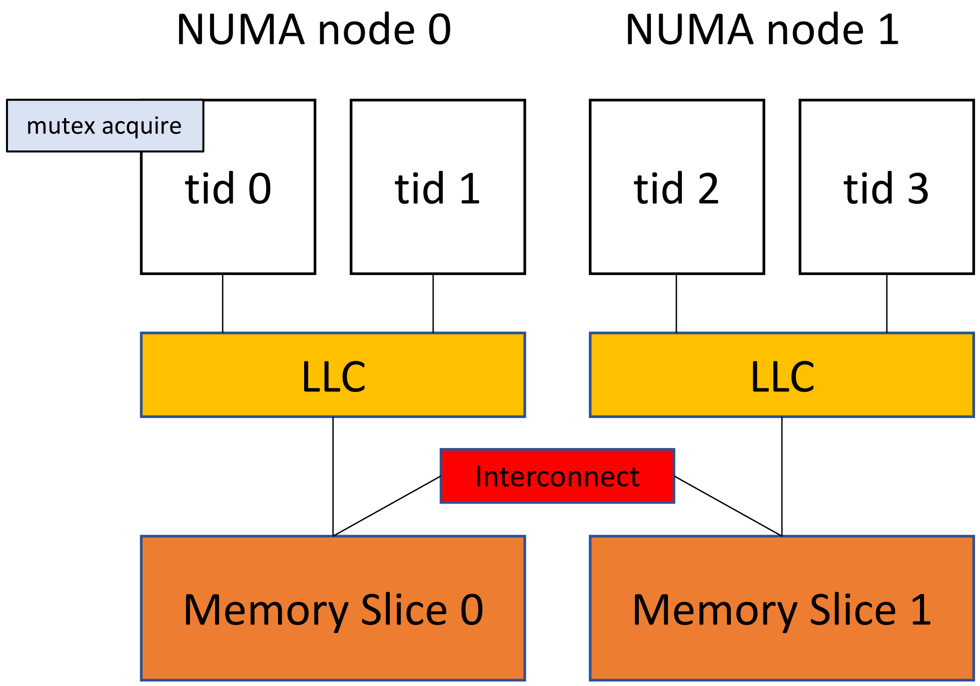
Mutex counter:
Local_Com: 3



Example:

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

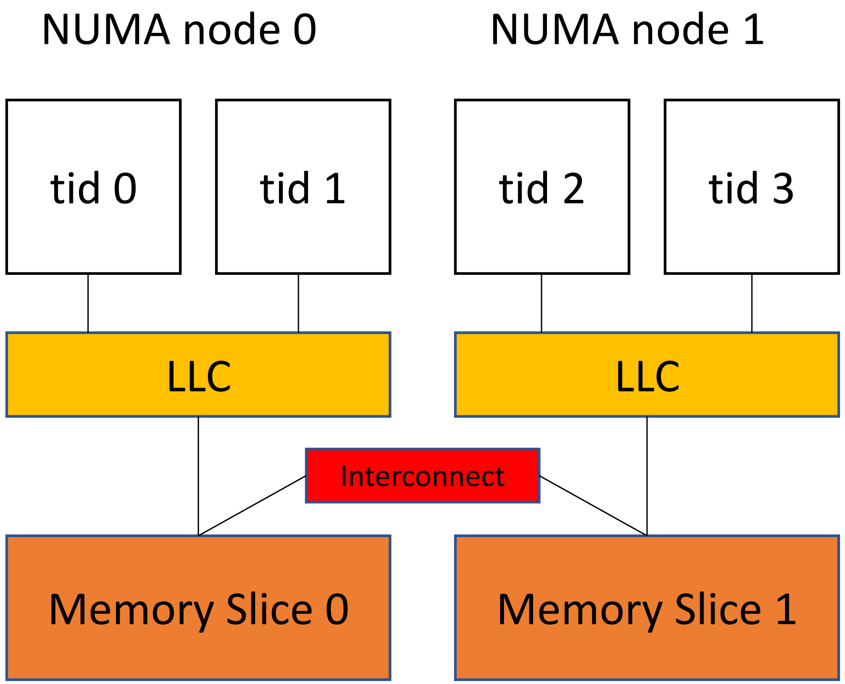
Mutex counter:
Local_Com: 3



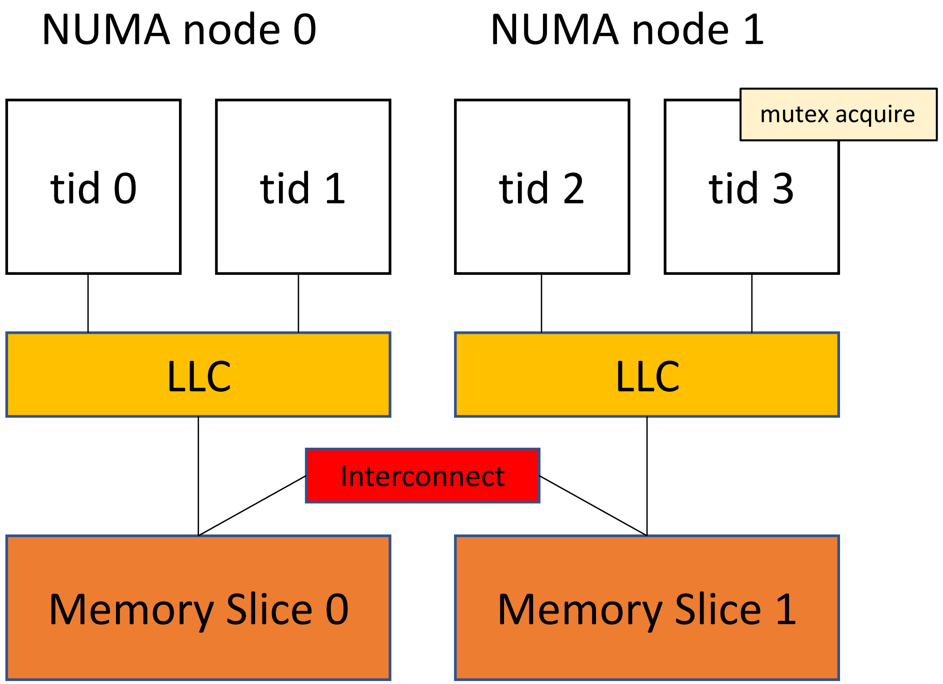
Example:

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

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?!

Schedule

- Reader-Write (RW) mutexes
- Hierarchical aware locks
- Impact of data-races

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

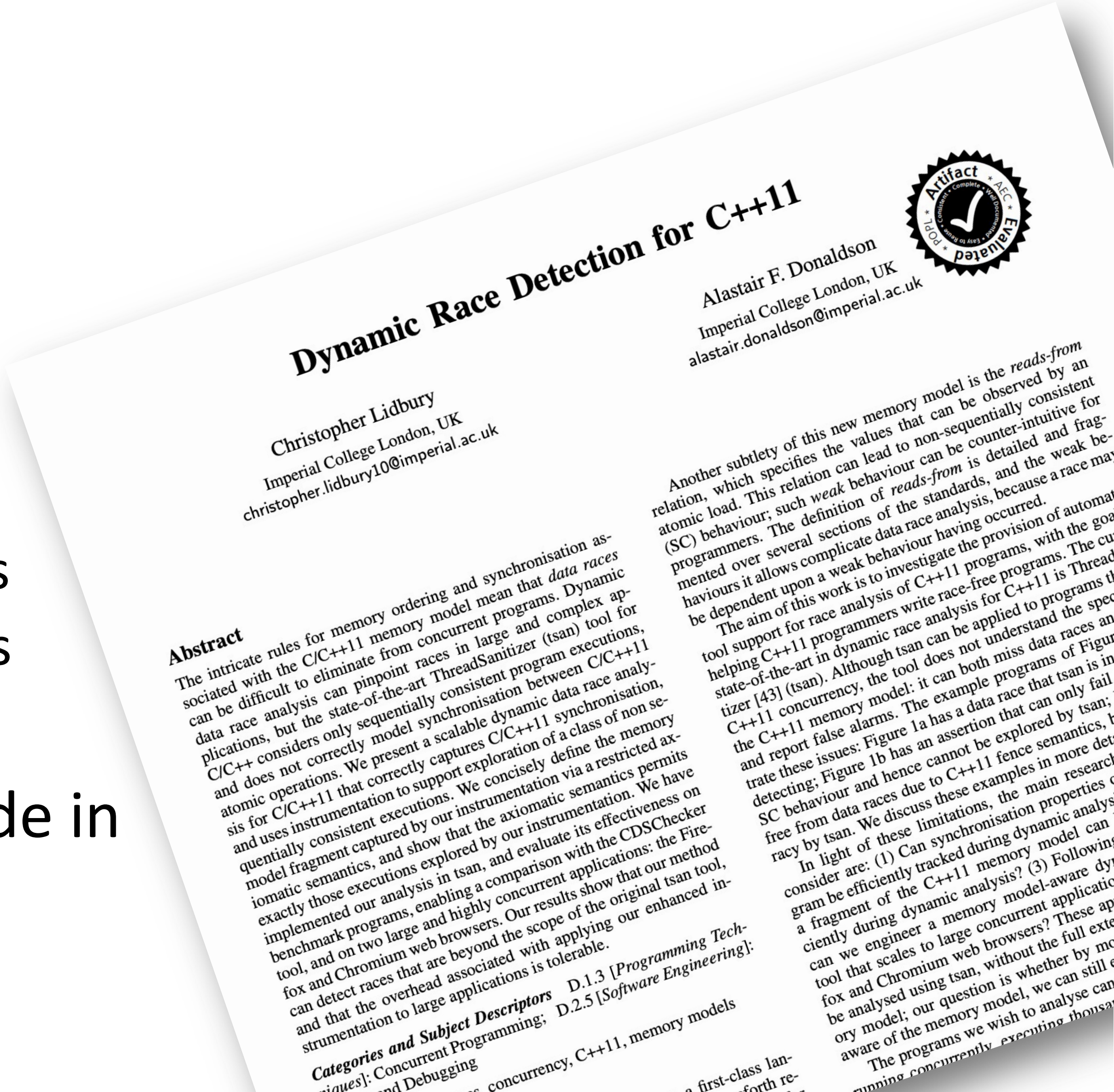
- Facebook Infer:
  - Statically checks for many issues (memory safety, assertions)
  - Can check for races in concurrent classes
  - Main support is for Java, although they claim support for C++

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 very rarely. (heisenbugs).
  - Better to use too many mutexes than not enough

- Use tools to help you!
  - Infer can helps with Java
  - Thread sanitizer helps with C++

Next week

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