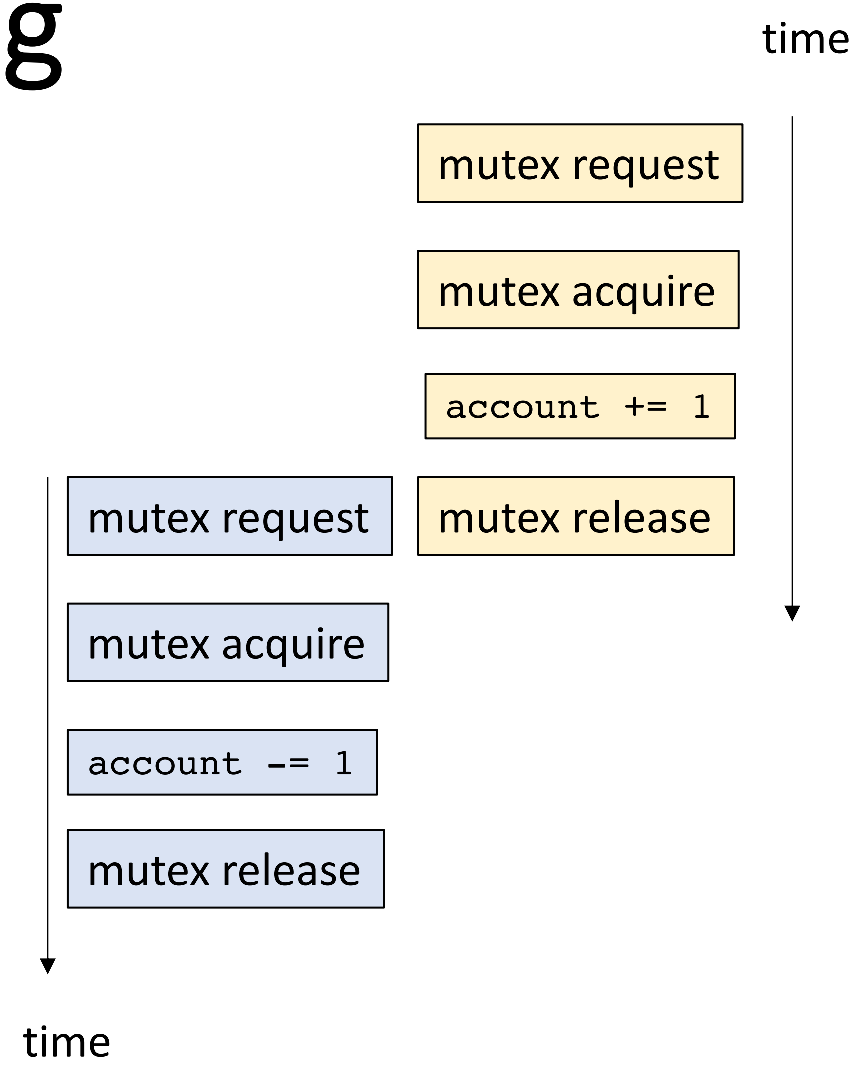
CSE113: Parallel Programming

Jan. 19, 2022

- **Topics**:
  - Mutual exclusion examples
  - Multiple mutexes
  - Mutex properties
  - Atomic operation properties



Announcements

- Hope everyone had a nice holiday weekend!

- Homework 1 is due on Friday!
  - Due at midnight, but instructor support ends at 5 PM
  - Ask questions on Piazza, come to office/tutoring hours
    - Reese has 1.5 hours today
    - I have 2 hours tomorrow
    - Tim has 1 hour tomorrow
    - Sanya has 1 hour on Friday

- Homework 2 is assigned on Friday

Announcements

- Homework 1 notes:
    - No assigned speedup required. You should get a noticeable speedup from ILP
        - AMD processors are being a little strange on part 2
        - Please note the processor in your write-up
    - You can start to share results now. Everyone's results will be slightly different
    - Sometimes you cannot account for small differences
        - We should be running the code for more iterations

Today's Quiz

- Didn't save ☹ I will update it after class. It will be ready by 3:00 PM
- Due by midnight the next day, or on Fridays, it is due before the next class
- From what I hear, the plan is to go back in person as planned 🎉

Previous quiz

A data conflict is when two threads access the same memory location.

True	25 respondents	**45 %**
False	31 respondents	55 %

Previous quiz

How many interleavings are possible with 3 threads, each them executing 1 event?

Previous quiz

How many extra arguments are required to turn a function into an SPMD function?

Previous quiz

Write a few sentences about how you can remove data-conflicts from your program. We have mentioned a few ways in class, but feel free to mention other ways you can think of!

Review

SPMD programming model

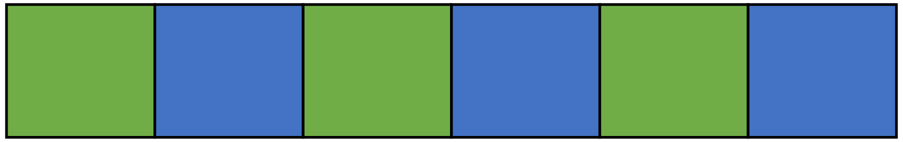
```
void increment_array(int *a, int a_size, int tid, int num_threads) {
    for (int i = tid; i < a_size; i+=num_threads) {
        a[i]++;
    }
}
```

SPMD programming model

```
void increment_array(int *a, int a_size, int tid, int num_threads) {
   for (int i = tid; i < a_size; i+=num_threads) {
       a[i]++;
   }
}
```

switch to thread 1

iteration 3 computes index 5



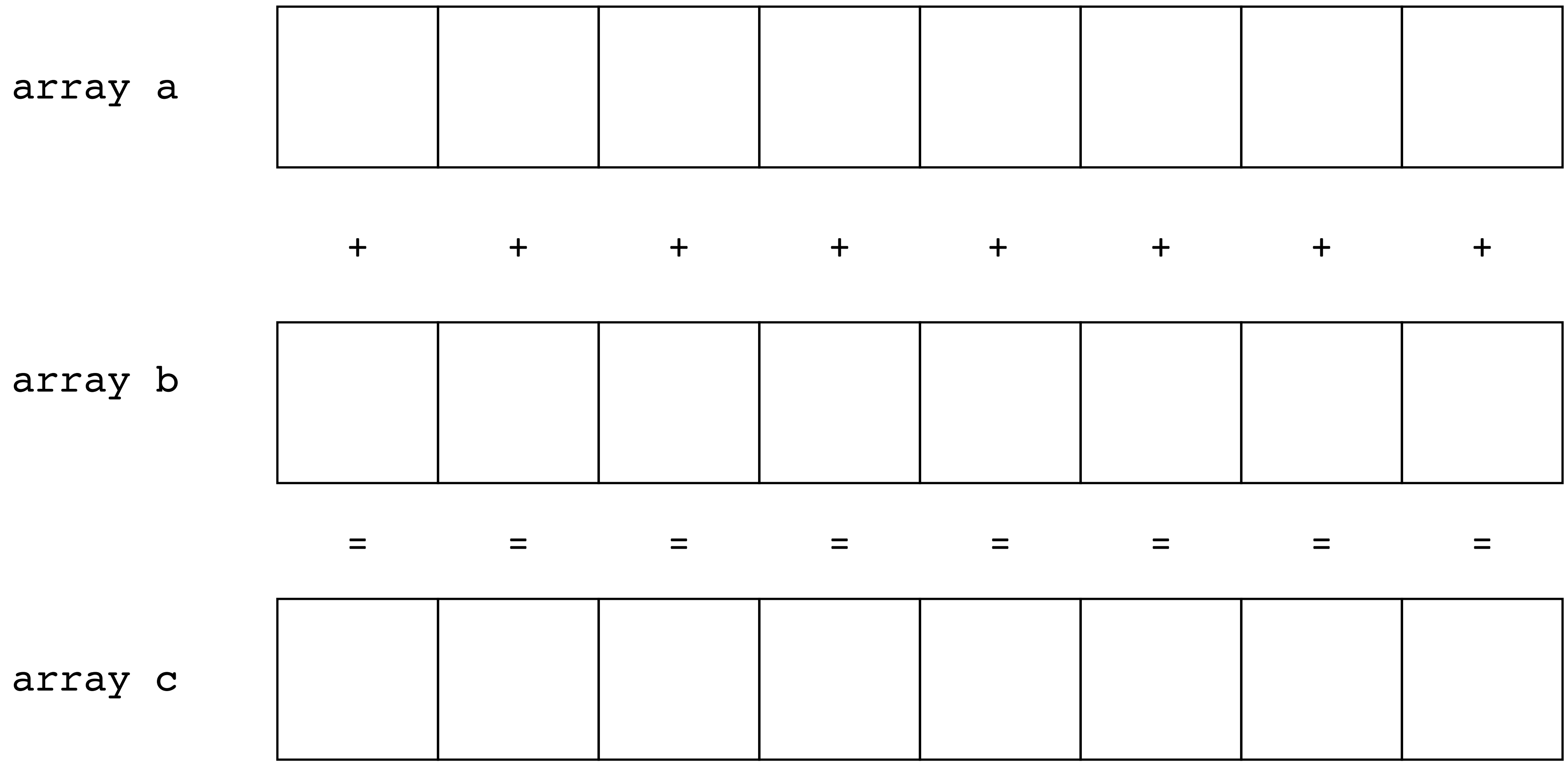
array a

Assume 2 threads
lets step through thread 1
i.e.
tid = 1
num_threads = 2

Embarrassingly parallel

Computation can easily be divided into threads

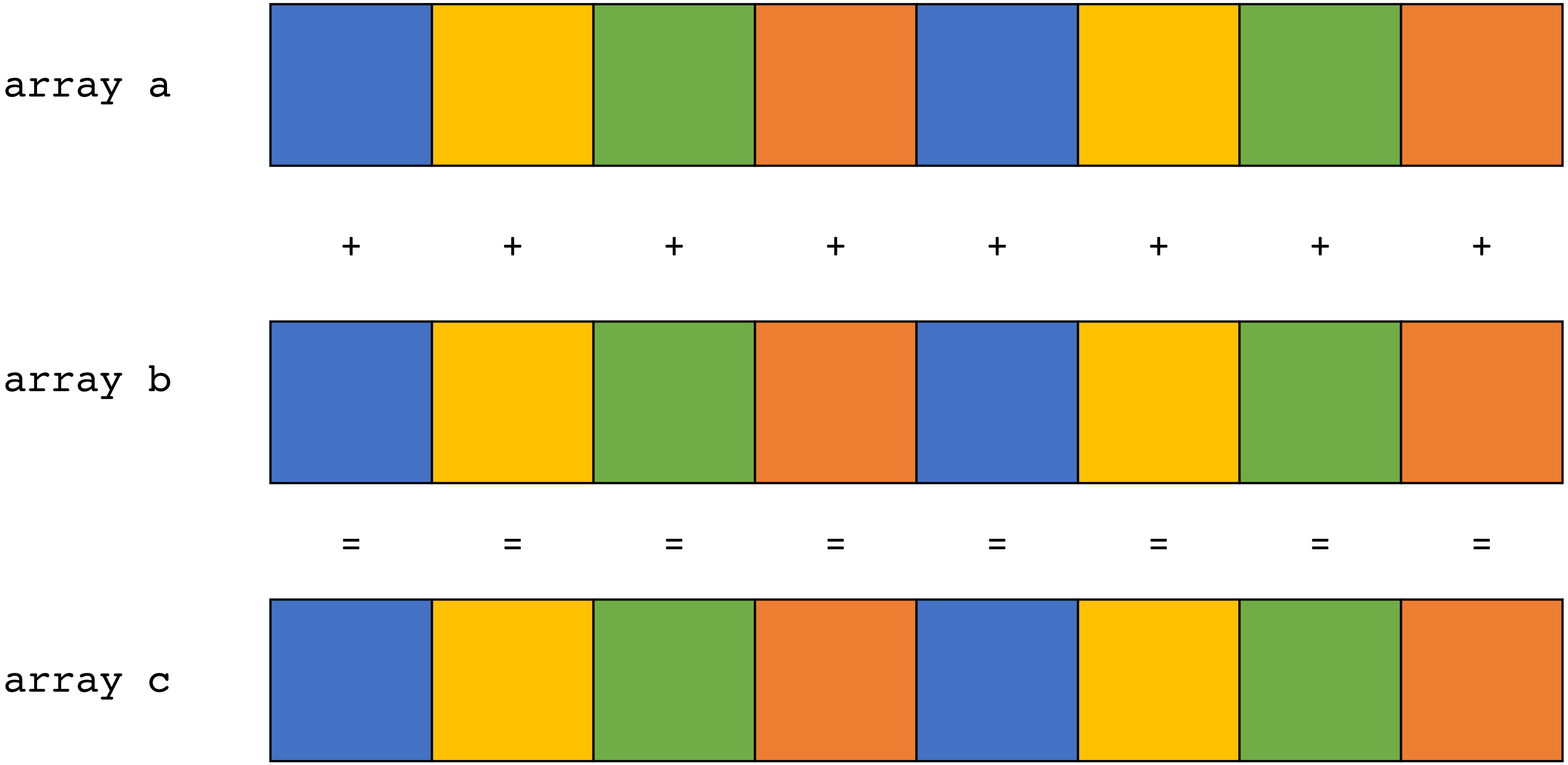
Thread 0 - Blue
Thread 1 - Yellow
Thread 2 - Green
Thread 3 - Orange



Embarrassingly parallel

Computation can easily be divided into threads

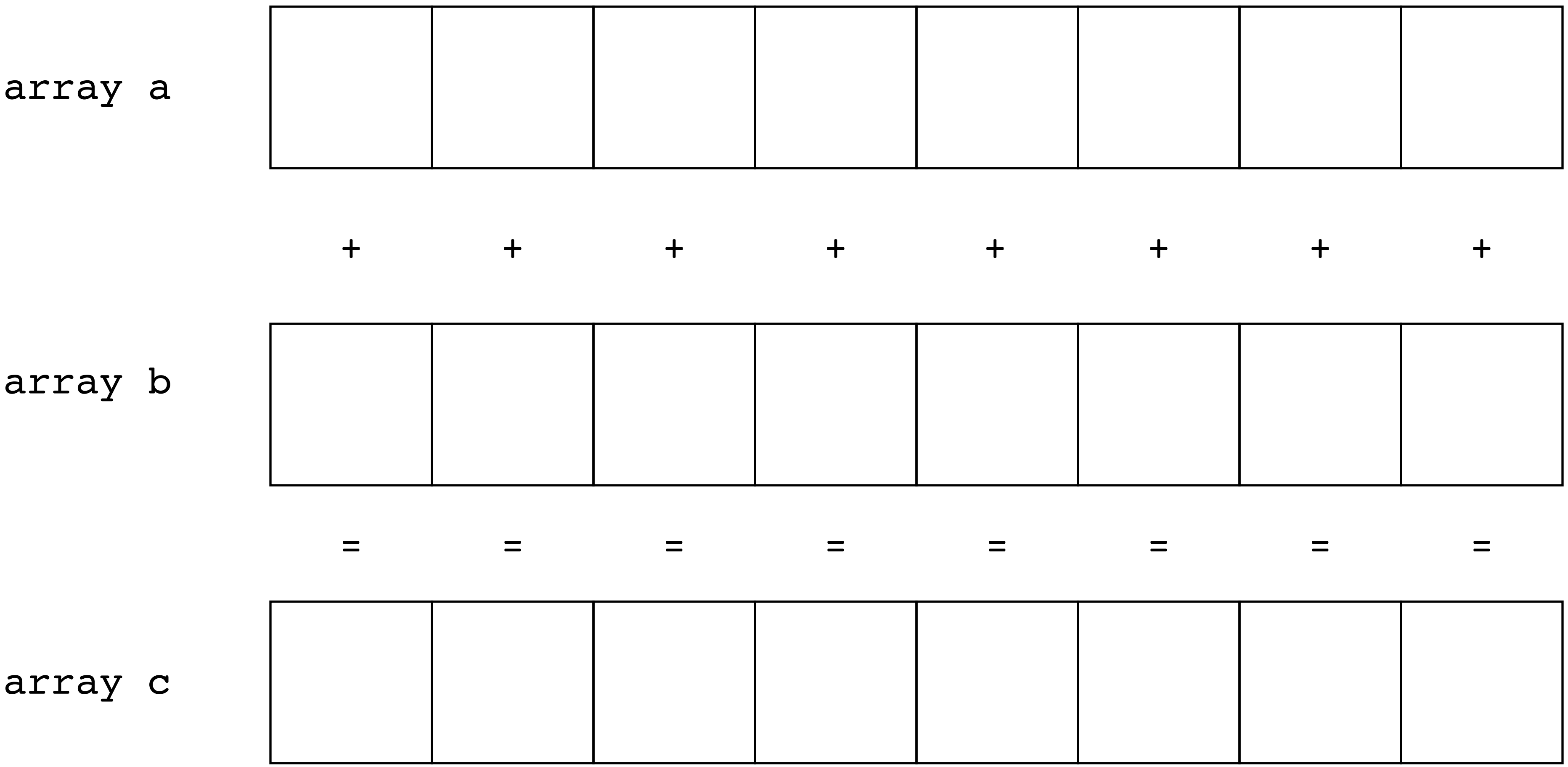
Thread 0 - Blue
Thread 1 - Yellow
Thread 2 - Green
Thread 3 - Orange



Embarrassingly parallel

Computation can easily be divided into threads

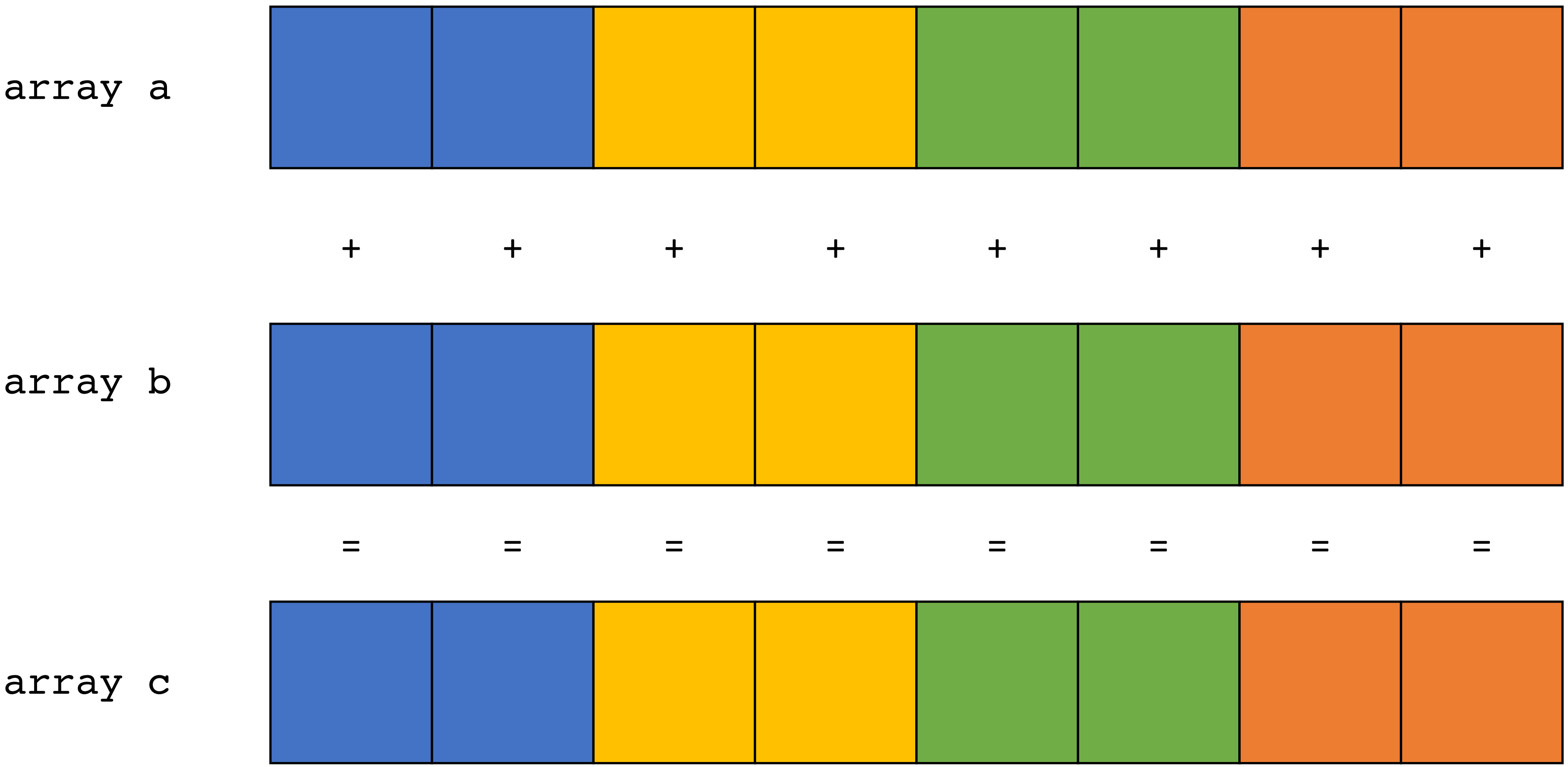
Thread 0 - Blue
Thread 1 - Yellow
Thread 2 - Green
Thread 3 - Orange



Embarrassingly parallel

Computation can easily be divided into threads

Thread 0 - Blue
Thread 1 - Yellow
Thread 2 - Green
Thread 3 - Orange



Reasoning about parallel programs

Programs to events:

Tyler's coffee addiction:

```
for (int i = 0; i < HOURS; i++) {
   tylers_account -= 1;
}
```

time ↓

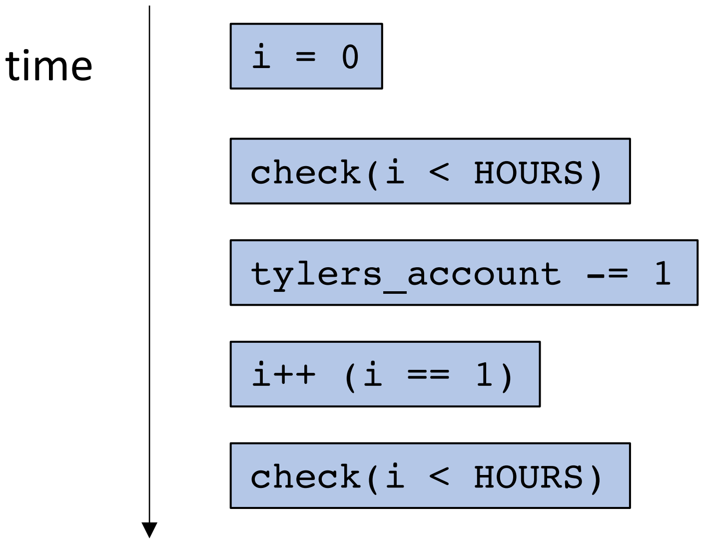
```
i = 0
check(i < HOURS)
tylers_account -= 1
i++ (i == 1)
check(i < HOURS)
tylers_account -= 1
i++ (i == 2)
check(i < HOURS)
tylers_account -= 1
```

Tyler's employer

```
for (int j = 0; j < HOURS; j++) {
   tylers_account += 1;
}
```

time ↓

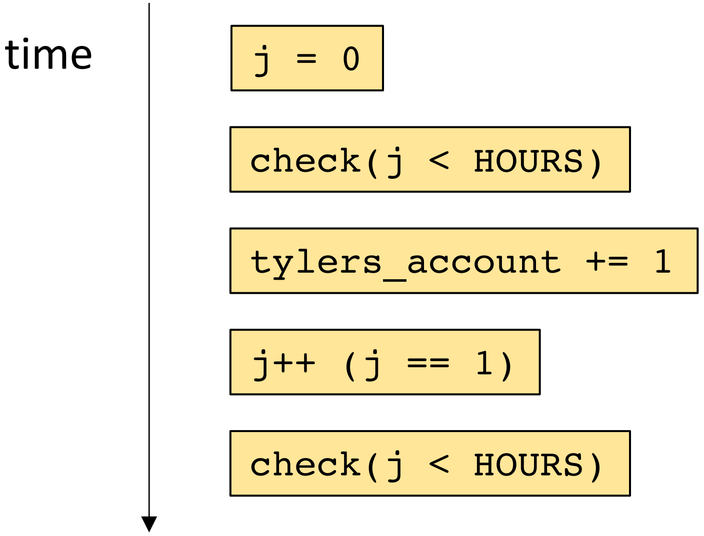
```
j = 0
check(j < HOURS)
tylers_account += 1
j++ (j == 1)
check(j < HOURS)
tylers_account += 1
j++ (j == 2)
check(j < HOURS)
tylers_account += 1
```



How many possible interleavings?
Combinatorics question:

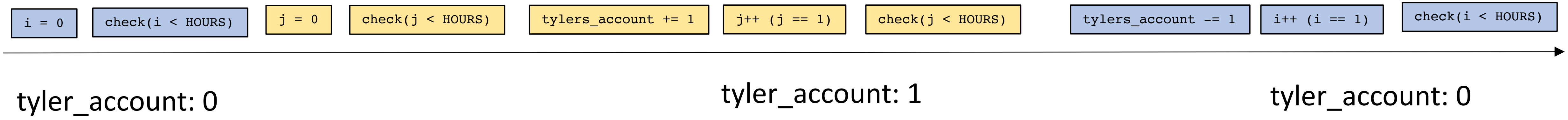
if Thread 0 has N events
if Thread 1 has M events

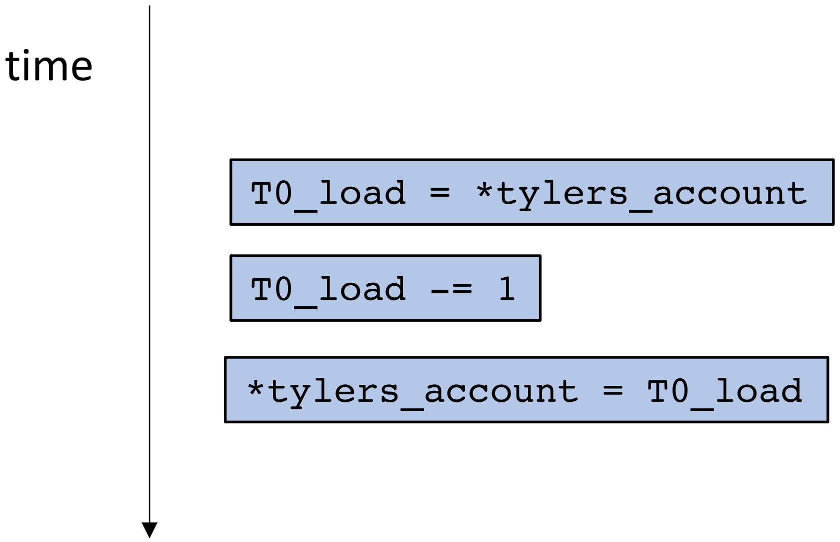
$$\frac{(N+M)!}{N!\,M!}$$

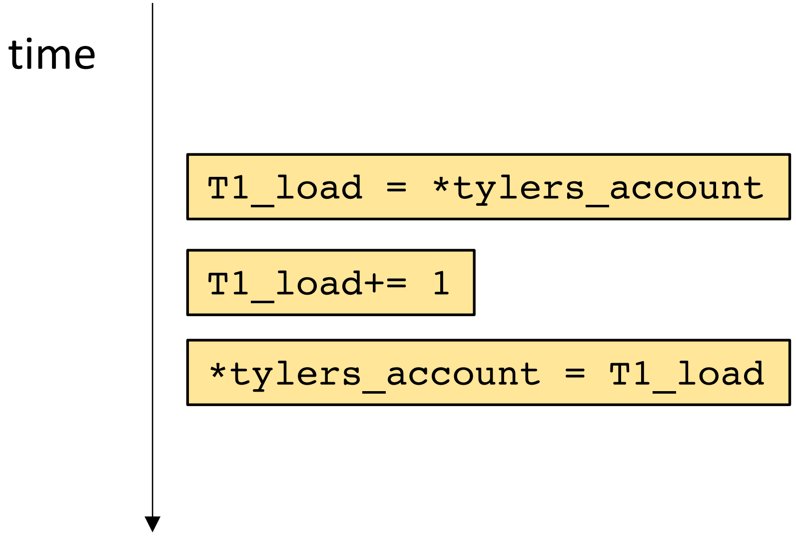


Concurrent execution

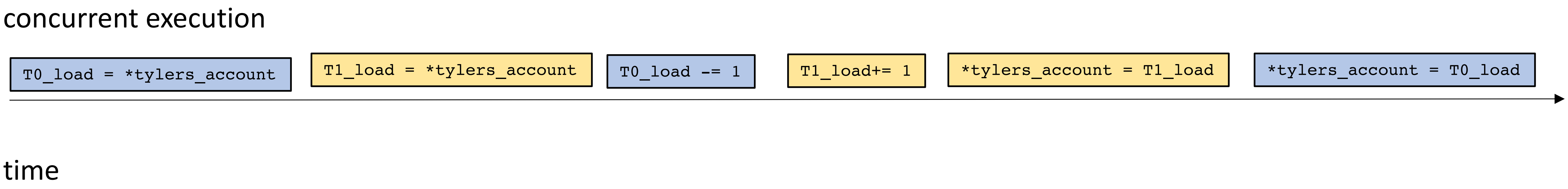
in our example there are 252 possible interleavings!







tylers_account has -1 at the end of this interleaving!



How to reason about our programs

- We don’t want data conflicts
  - Requires reasoning about the compiler and machine. Not portable and extremely error prone
  - Technically undefined in C++ and Java
- View simpler versions of the program
  - e.g. one loop iteration
- High-level properties
  - Final value in the account after execution

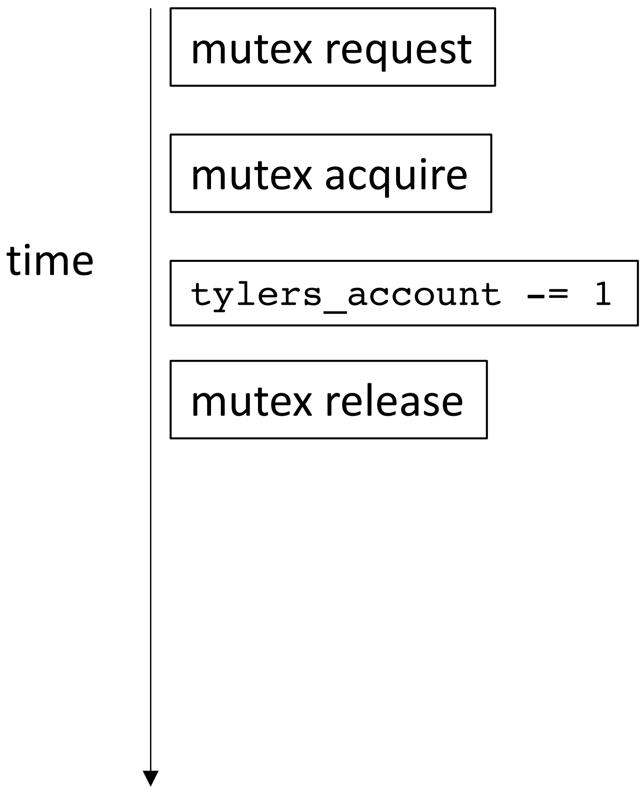
Mutex events

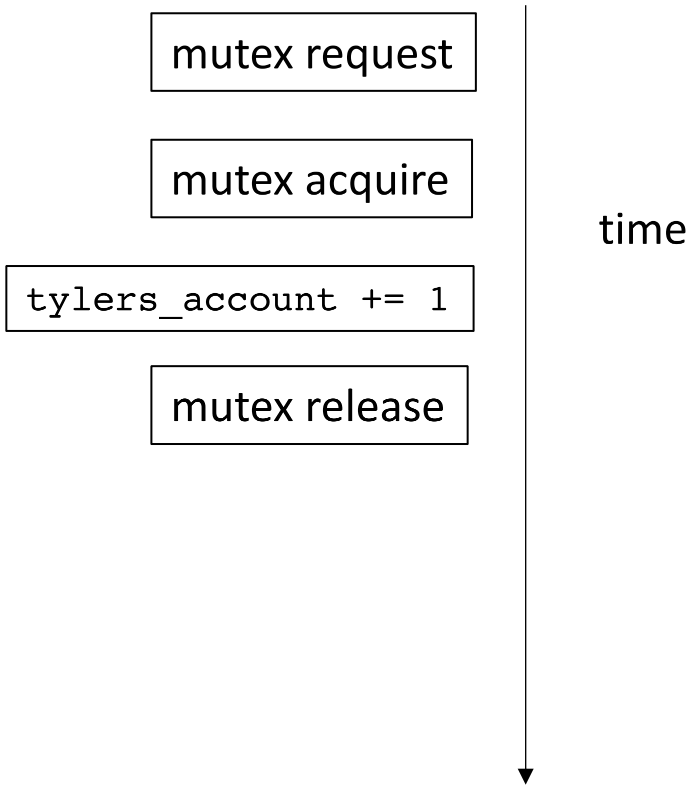
Tyler's coffee addiction:

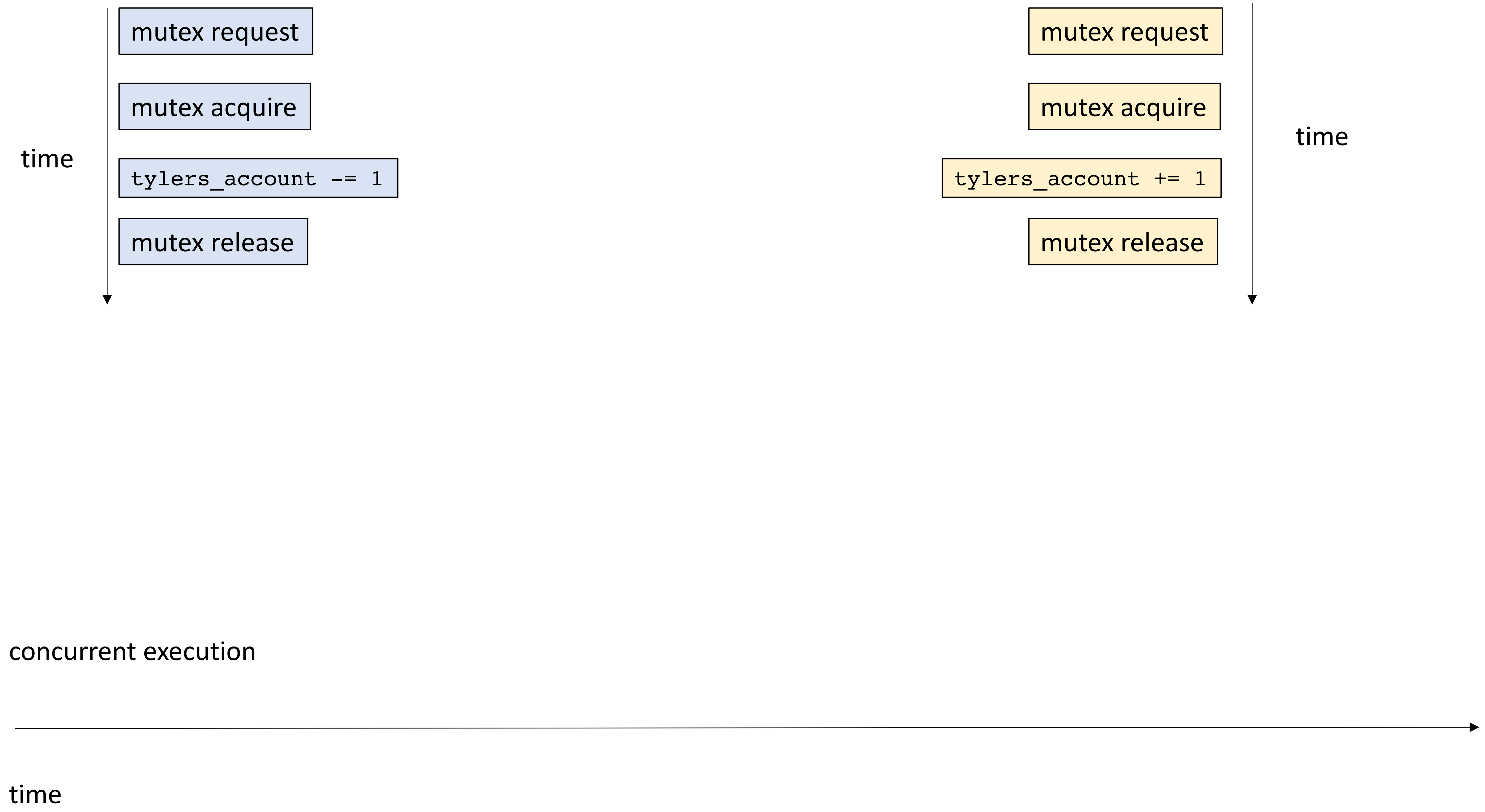
```
m.lock();
tylers_account -= 1;
m.unlock();
```

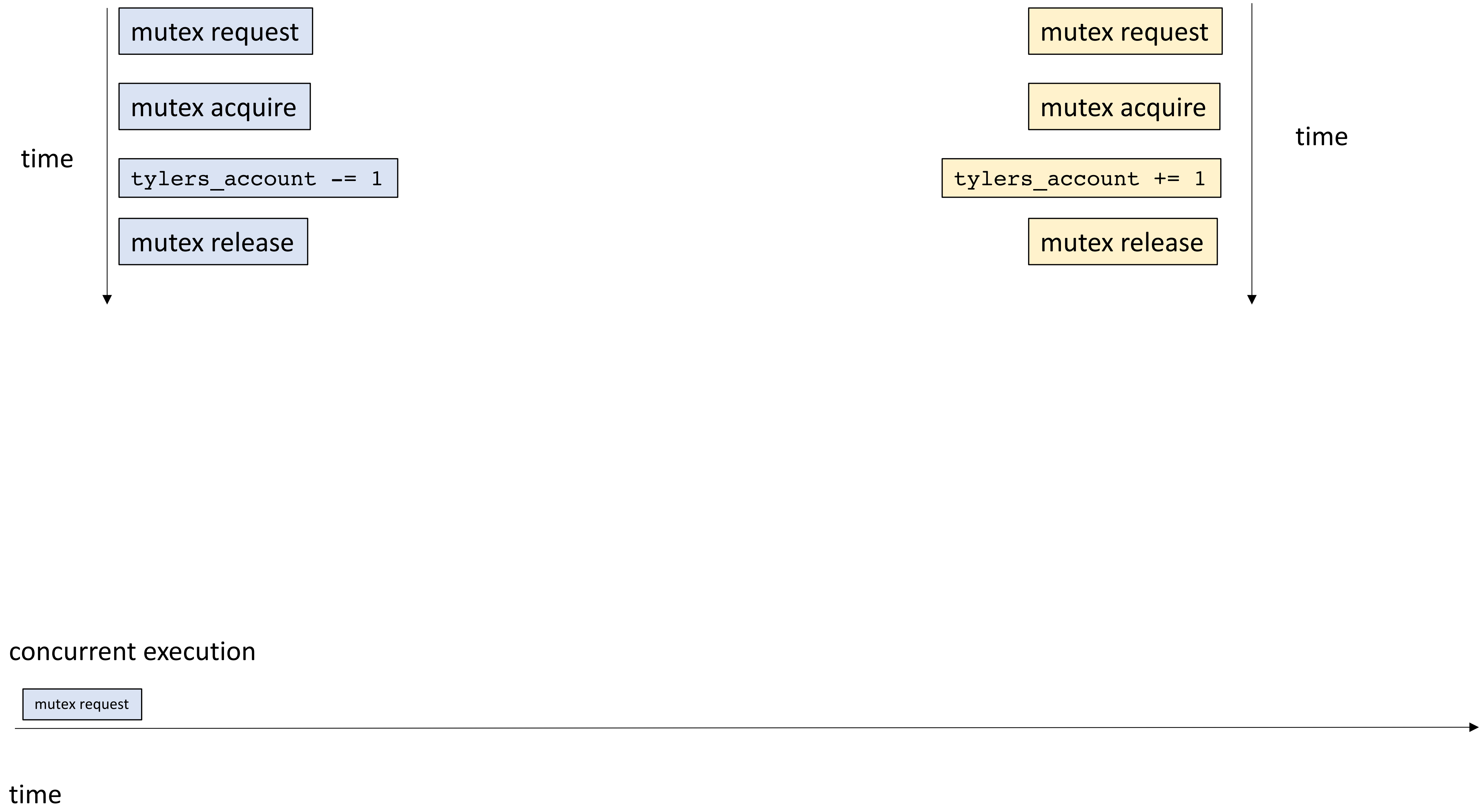
Tyler's employer

```
m.lock();
tylers_account += 1;
m.unlock();
```



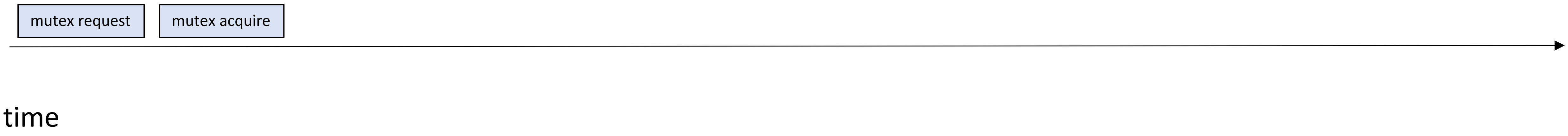


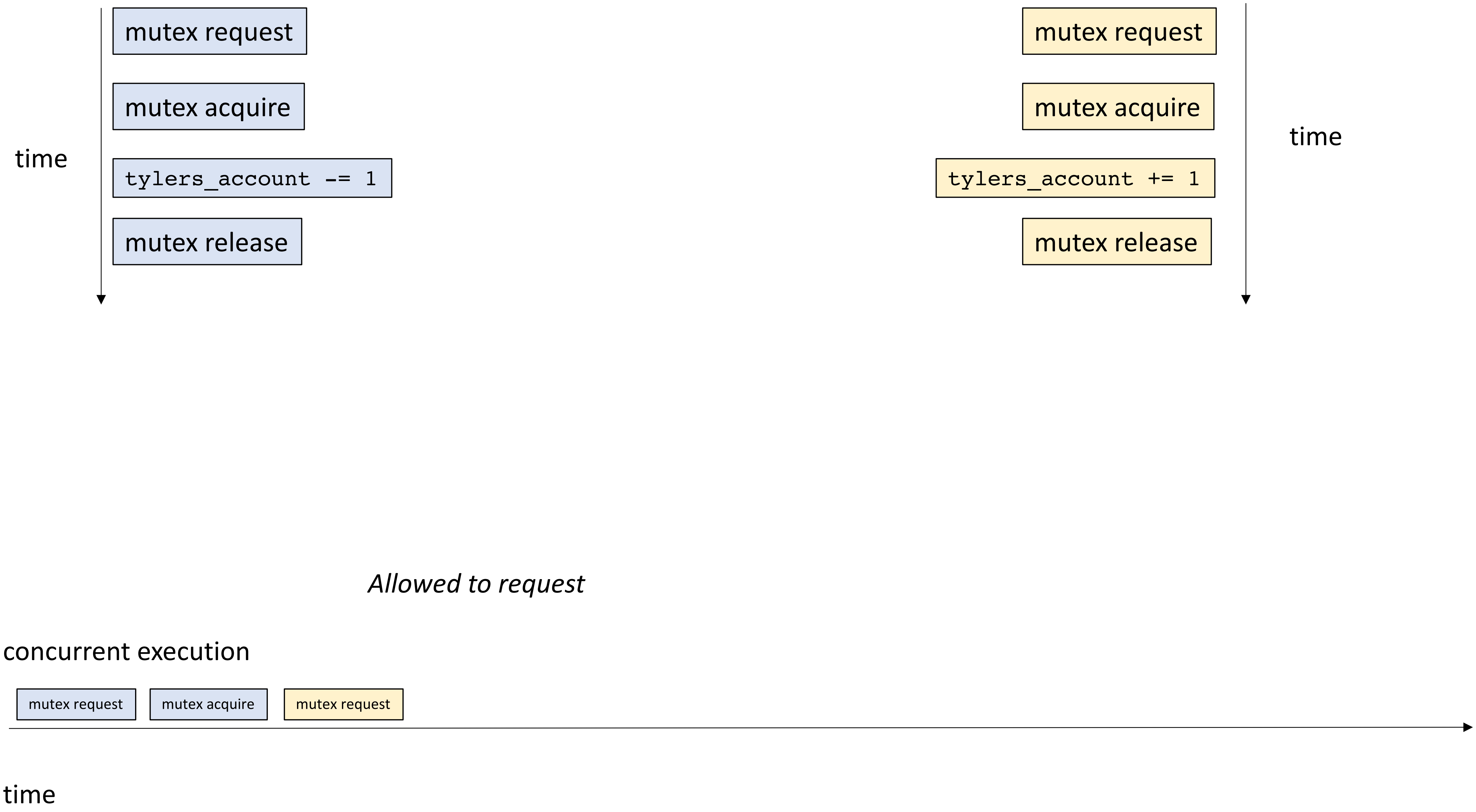

mutex request
mutex acquire
tylers_account -= 1
mutex release
time
mutex request
mutex acquire
tylers_account += 1
mutex release
time
concurrent execution
time


mutex request
mutex acquire
time
tylers_account -= 1
mutex release
mutex request
mutex acquire
time
tylers_account += 1
mutex release
concurrent execution
mutex request
time

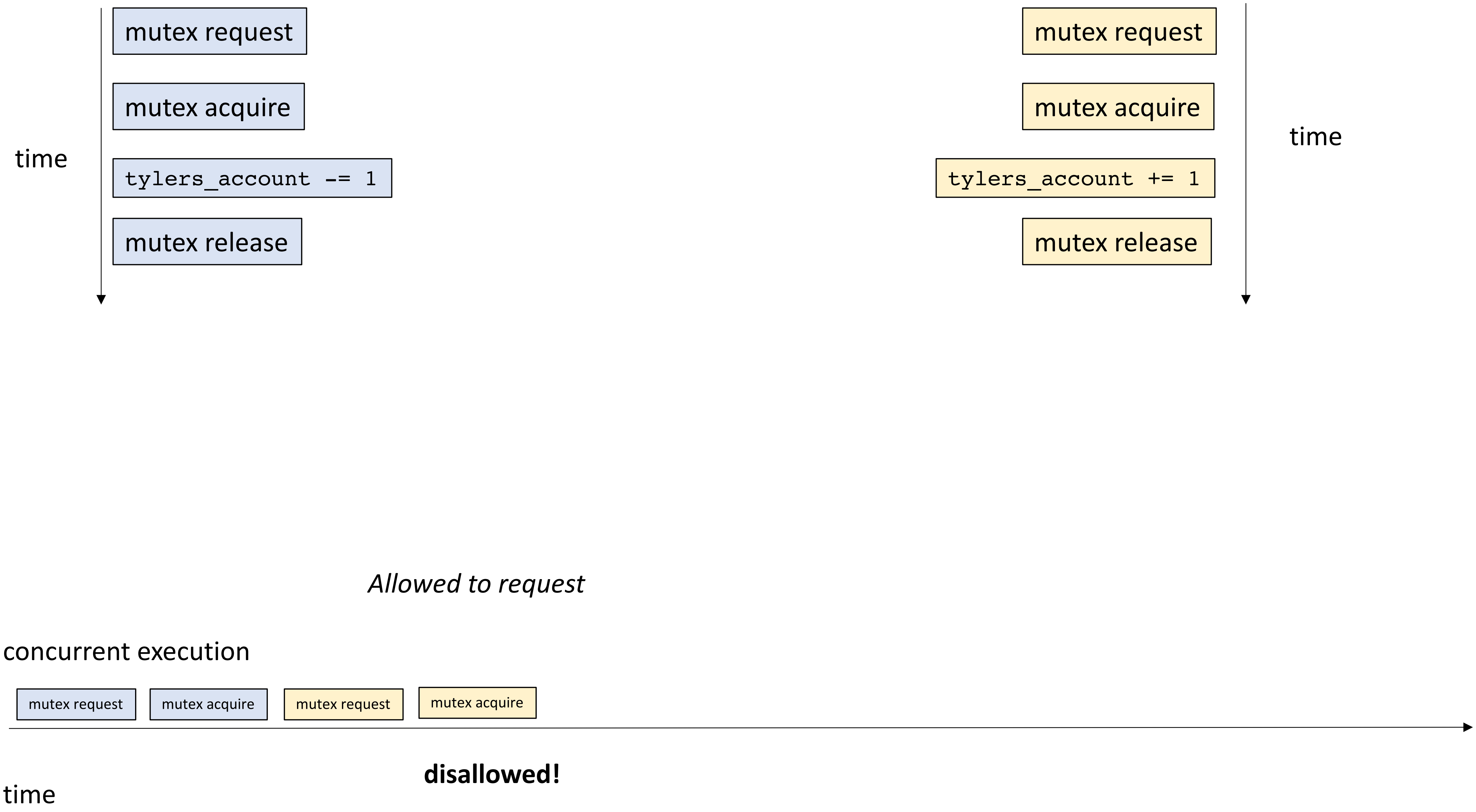
at this point, thread 0 holds the mutex.
another thread cannot acquire the mutex until thread 0 releases the mutex
also called the ***critical section.***

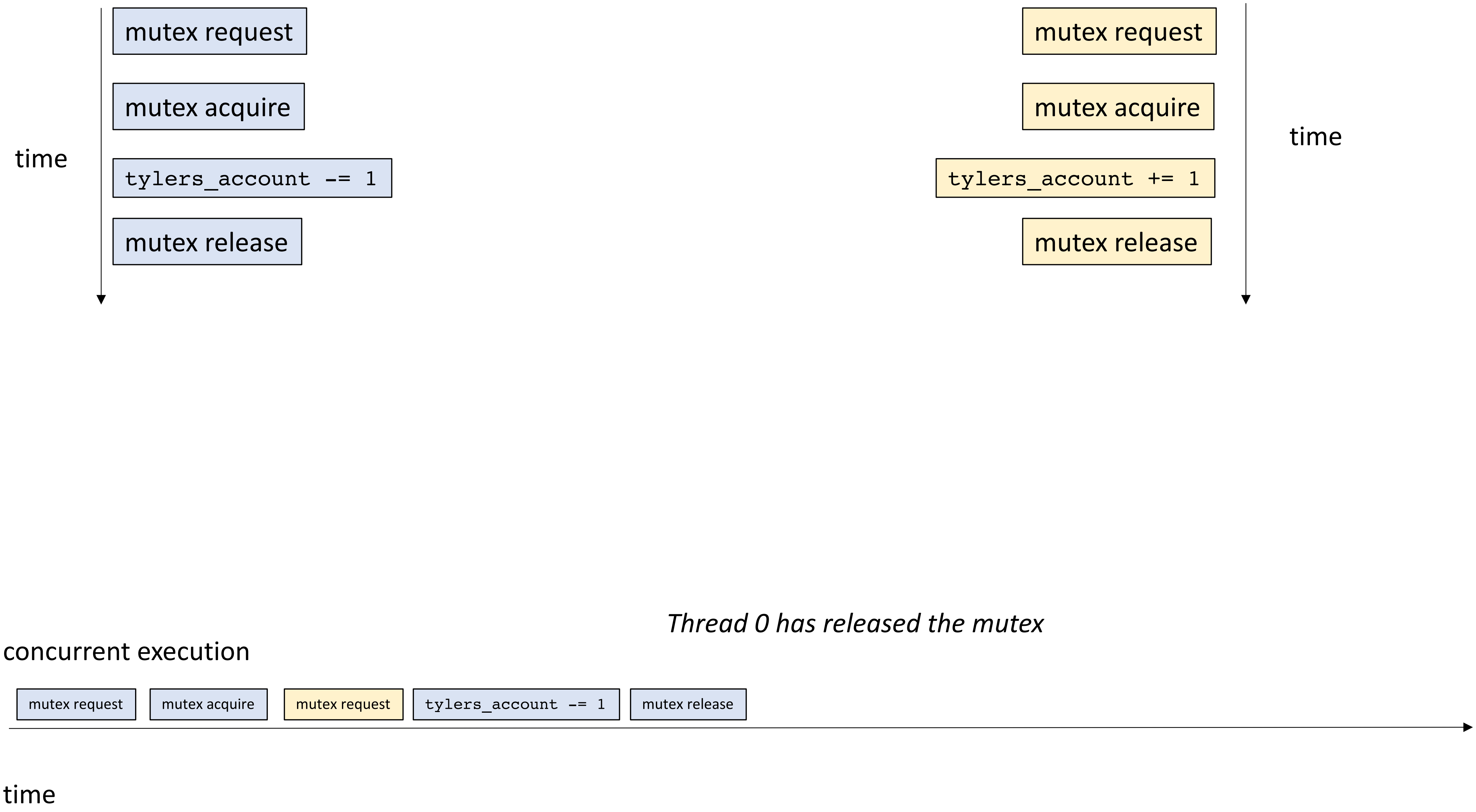
concurrent execution



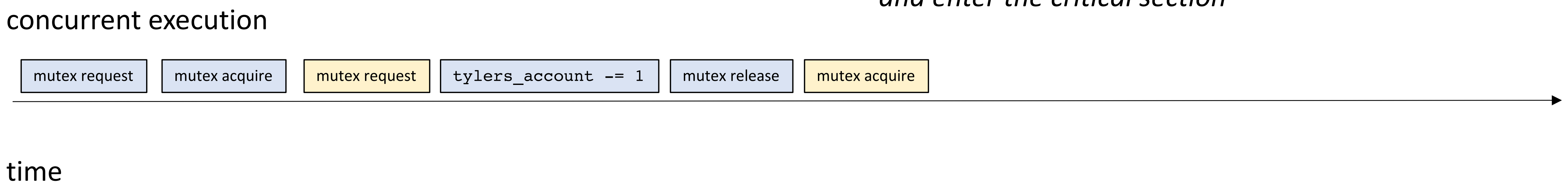


mutex request
mutex acquire
time
tylers_account -= 1
mutex release
mutex request
mutex acquire
time
tylers_account += 1
mutex release
Allowed to request
concurrent execution
mutex request
mutex acquire
mutex request
time


mutex request
mutex acquire
tylers_account -= 1
mutex release
time
mutex request
mutex acquire
tylers_account += 1
mutex release
time
Allowed to request
concurrent execution
mutex request
mutex acquire
mutex request
mutex acquire
disallowed!
time

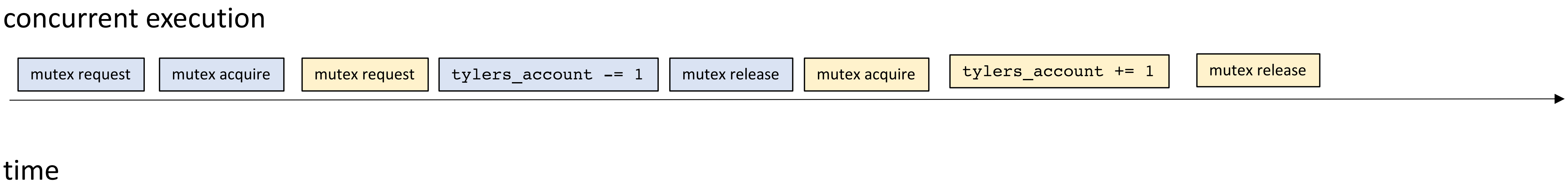

mutex request
mutex acquire
tylers_account -= 1
mutex release
time
mutex request
mutex acquire
tylers_account += 1
mutex release
time
Thread 0 has released the mutex
concurrent execution
mutex request
mutex acquire
mutex request
tylers_account -= 1
mutex release
time

Thread 1 can take the mutex and enter the critical section



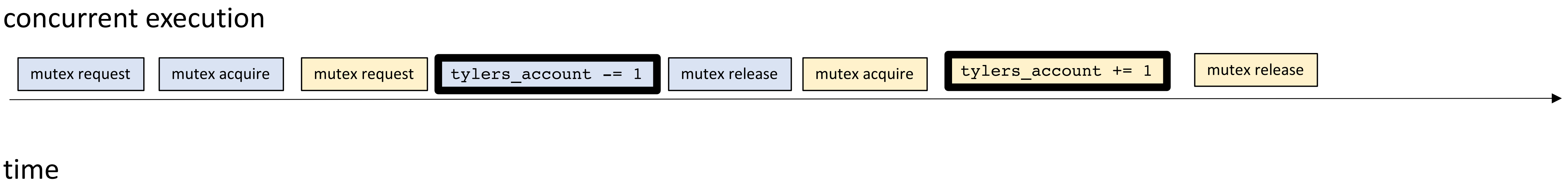
A mutex restricts the number of allowed interleavings
Critical section are mutually exclusive: i.e. they cannot interleave

Thread 1 can take the mutex
and enter the critical section



It means we don't have to think about 3 address code

Thread 1 can take the mutex and enter the critical section



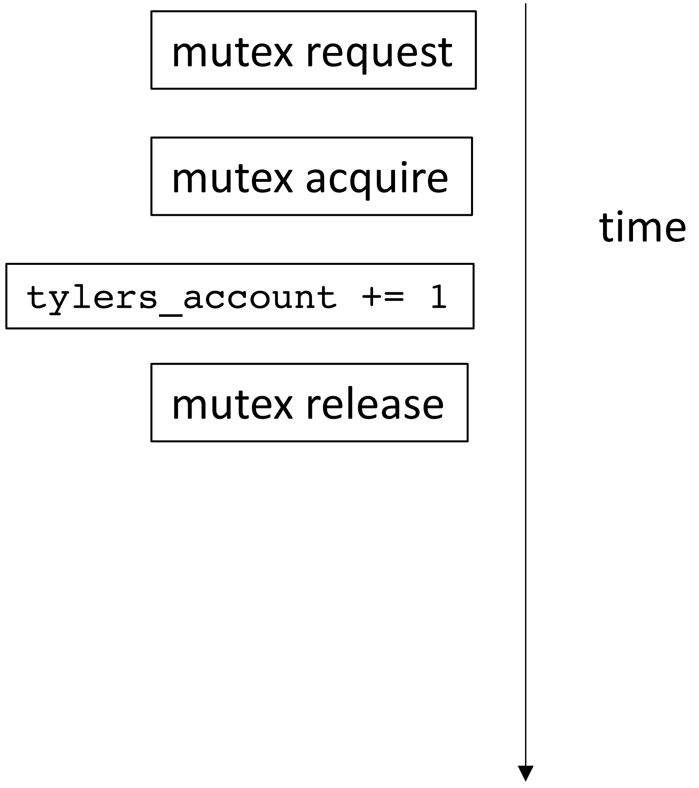
Make sure to unlock your mutex!

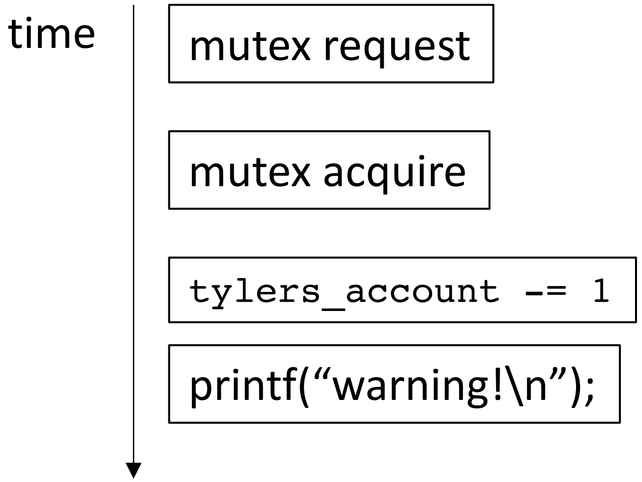
Tyler's coffee addiction:

```
m.lock();
tylers_account -= 1;
if (tylers_account < -100) {
  printf("warning!\n");
  return;
}
m.unlock();
return;
```

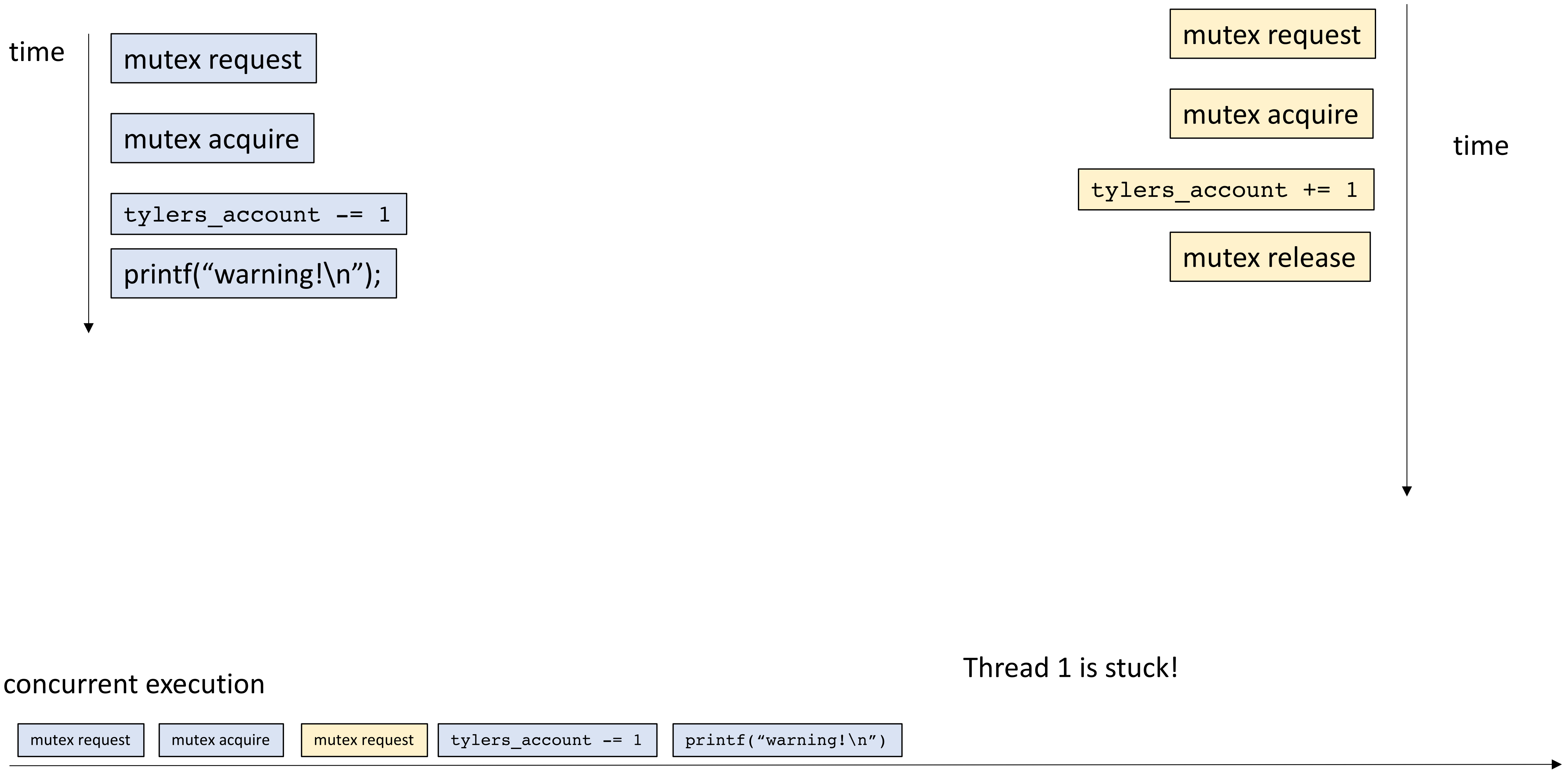
Tyler's employer

```
m.lock();
tylers_account += 1;
m.unlock();
```





say tylers_account is -1000


time
mutex request
mutex acquire
tylers_account -= 1
printf(“warning!\n”);
mutex request
mutex acquire
tylers_account += 1
mutex release
time
Thread 1 is stuck!
concurrent execution
mutex request
mutex acquire
mutex request
tylers_account -= 1
printf(“warning!\n”)

On to the lecture!

Lecture schedule

- Mutex performance considerations

- Multiple mutexes

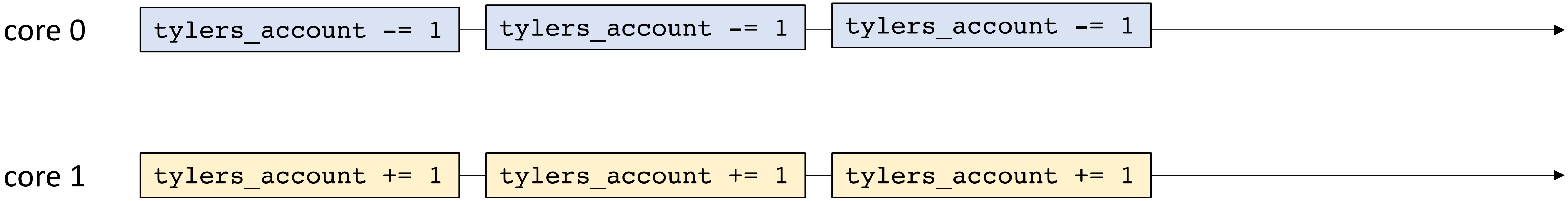
Mutex Performance

- What about timing?
  - Overhead of acquiring/releasing mutex
  - Cache flushing (heavier weight than coherence)
  - Reduces parallelism

Mutex Performance

- What about timing?
  - Overhead of acquiring/releasing mutex
  - Cache flushing (heavier weight than coherence)
  - Reduces parallelism

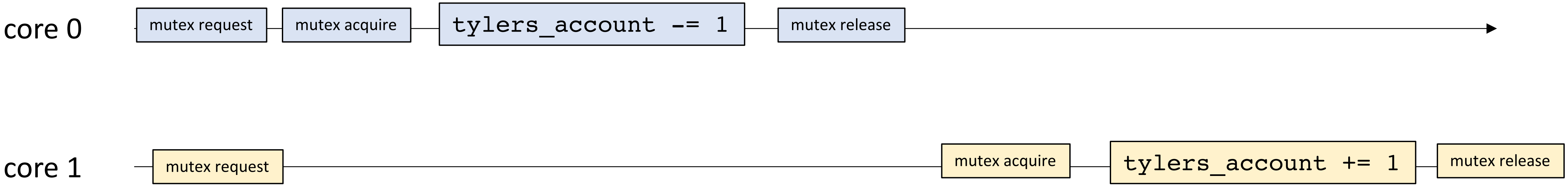
in a parallel system without the mutex



Mutex Performance

- What about timing?
  - Overhead of acquiring/releasing mutex
  - Cache flushing (heavier weight than coherence)
  - Reduces parallelism

in a parallel system with the mutex



Long periods of waiting in the threads

Code example

Mutex Performance

try to keep mutual exclusion sections small!

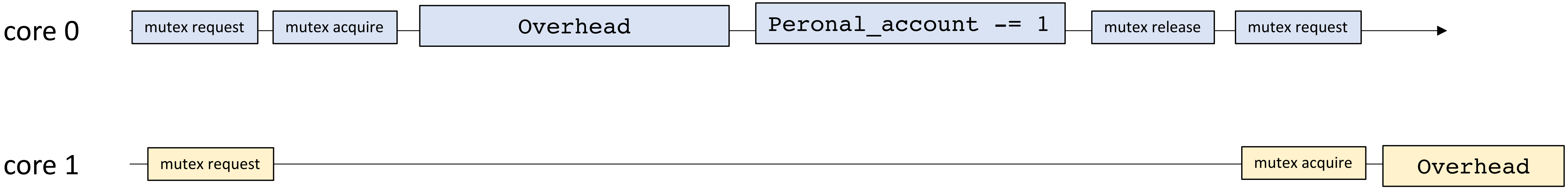
Code example with overhead

Mutex Performance

Try to keep mutual exclusion sections small! Protect only data conflicts!

Code example with overhead

Long periods of waiting in the threads

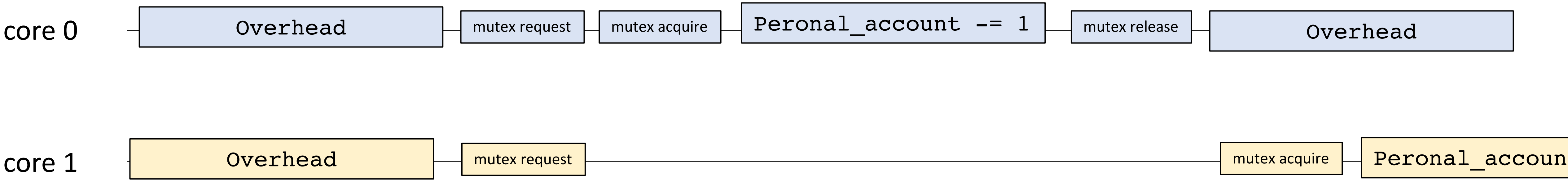


Long periods of waiting in the threads

Mutex Performance

Try to keep mutual exclusion sections small! Protect only data conflicts!

Code example with overhead



overlap the overhead (i.e. computation without any data conflicts)

Mutex alternatives?

Other ways to implement accounts?

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

```
atomic_fetch_add(atomic_int * addr, int value) {
   int tmp = *addr; // read
   tmp += value;    // modify
   *addr = tmp;     // write
}
```

other operations: max, min, etc.

Modify these programs to use atomic RMWs

<u>*Tyler's coffee addiction:*</u>

```
m.lock();
tylers_account -= 1;
m.unlock();
```

<u>*Tyler's employer*</u>

```
m.lock();
tylers_account += 1;
m.unlock();
```

time

time

Modify these programs to use atomic RMWs

Tyler's coffee addiction:

~~m.lock();~~
`tylers_account -= 1;`
~~m.unlock();~~

Tyler's employer

~~m.lock();~~
`tylers_account += 1;`
~~m.unlock();~~

time

time

Modify these programs to use atomic RMWs

Tyler's coffee addiction:

```
tylers_account -= 1;
```

Tyler's employer

```
tylers_account += 1;
```

time

time

Modify these programs to use atomic RMWs

Tyler's coffee addiction:

```
atomic_fetch_add(&tylers_account, -1);
```

Tyler's employer

```
atomic_fetch_add(&tylers_account, 1);
```

time

time

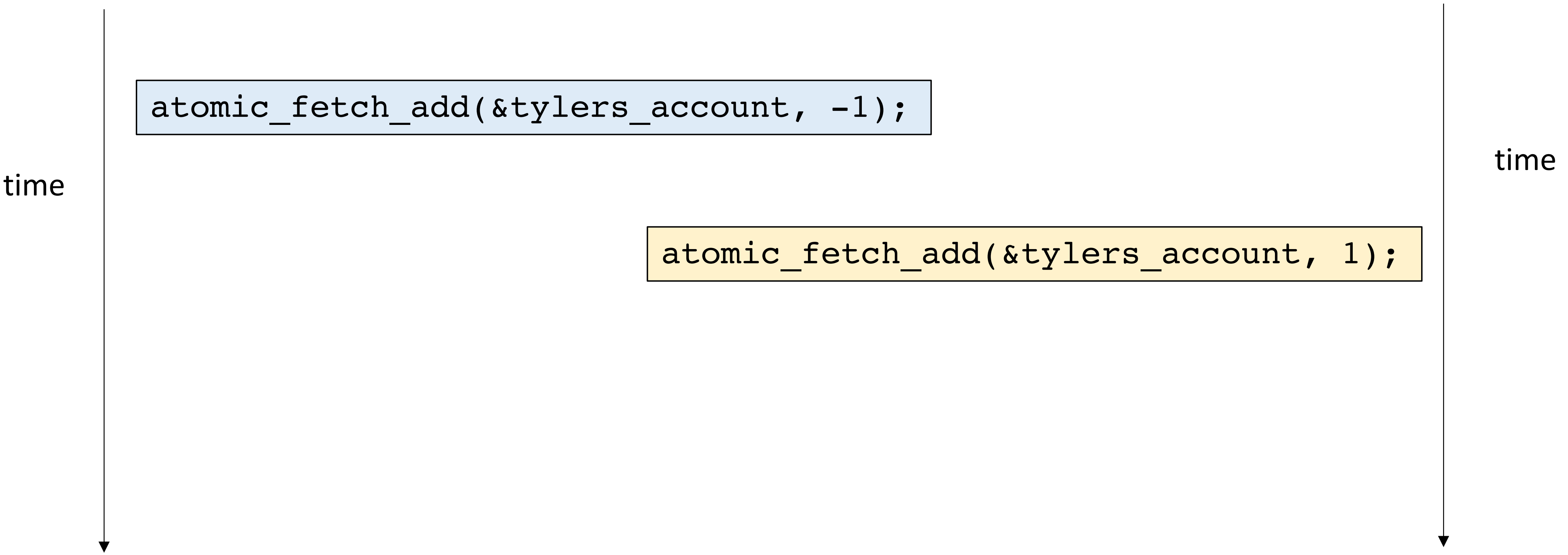
Modify these programs to use atomic RMWs

Tyler's coffee addiction:

```
atomic_fetch_add(&tylers_account, -1);
```

Tyler's employer

```
atomic_fetch_add(&tylers_account, 1);
```



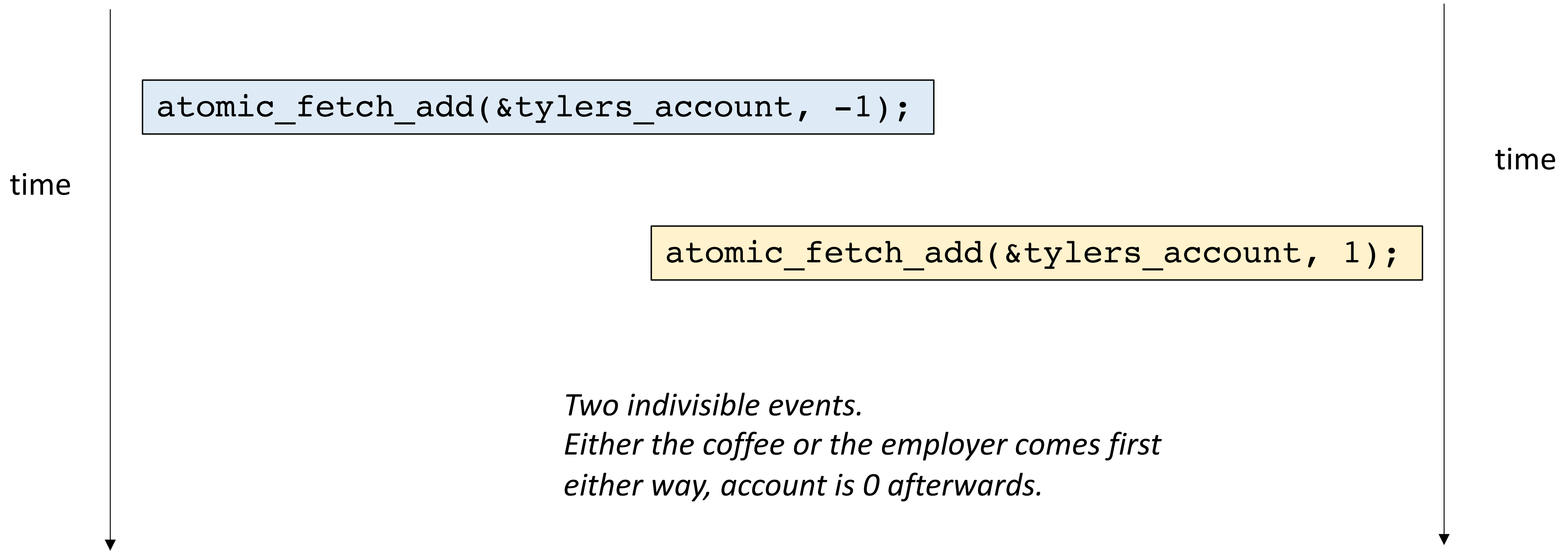
Modify these programs to use atomic RMWs

Tyler's coffee addiction:

```
atomic_fetch_add(&tylers_account, -1);
```

Tyler's employer

```
atomic_fetch_add(&tylers_account, 1);
```



Two indivisible events.
Either the coffee or the employer comes first
either way, account is 0 afterwards.

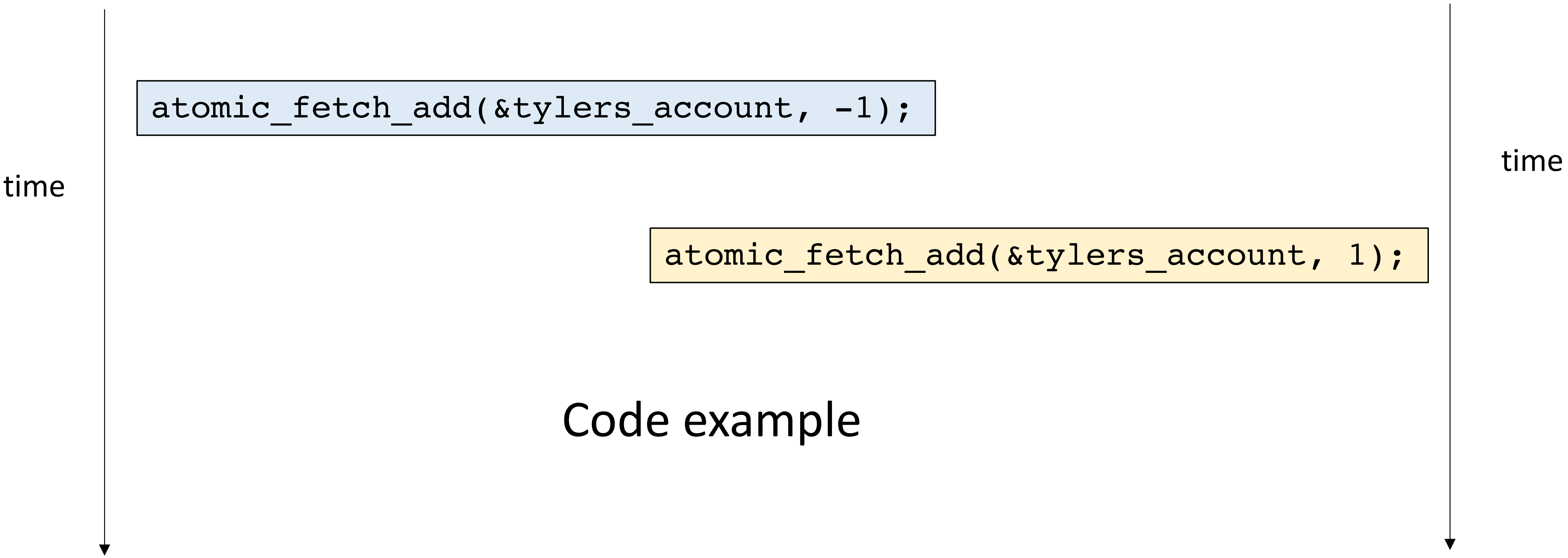
Modify these programs to use atomic RMWs

Tyler's coffee addiction:

```
atomic_fetch_add(&tylers_account, -1);
```

Tyler's employer

```
atomic_fetch_add(&tylers_account, 1);
```



Code example

Atomic RMWs

Pros? Cons?

Atomic RMWs

Pros? Cons?

Not all architectures support RMWs (although more common with C++11)

Limits critical section (what if account needs additional updating?)

atomic types need to propagate through the entire application

Multiple mutexes

Lets say I have two accounts:

- Business account
- Personal account

- Need to protect both of them using a mutex
  - Easy, we can just the same mutex
  - Show implementation

Multiple mutexes

Lets say I have two accounts:

- Business account
- Personal account

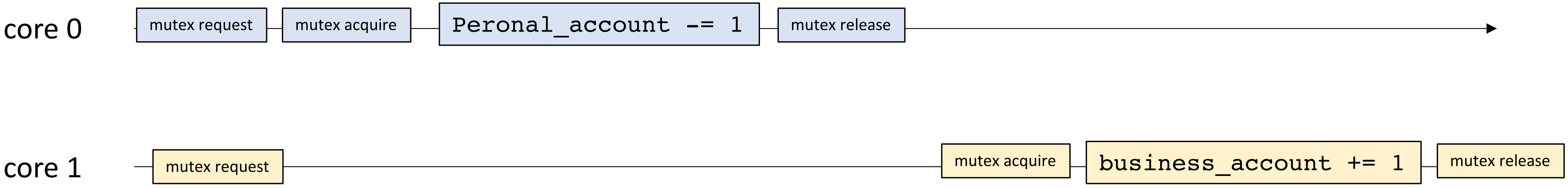
- No reason individual accounts can’t be accessed in parallel

Multiple mutexes

Lets say I have two accounts:

- Business account
- Personal account

- No reason individual accounts can't be accessed in parallel



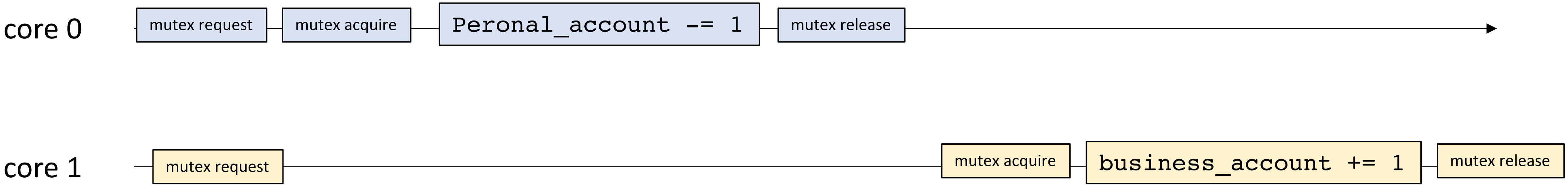
Long periods of waiting in the threads

Multiple mutexes

Mutexes are objects. We can create multiple versions of them to protect different shared data.

`MutexP` for personal account
`MutexB` for business account

Critical sections across different mutexes can overlap

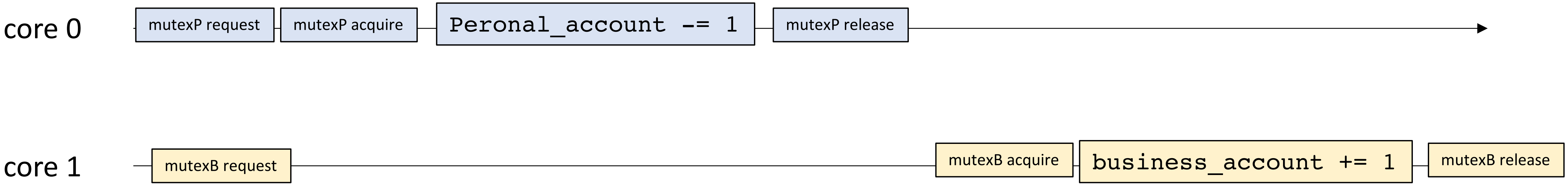


Multiple mutexes

Mutexes are objects. We can create multiple versions of them to protect different shared data.

`MutexP` for personal account
`MutexB` for business account

Critical sections across different mutexes can overlap

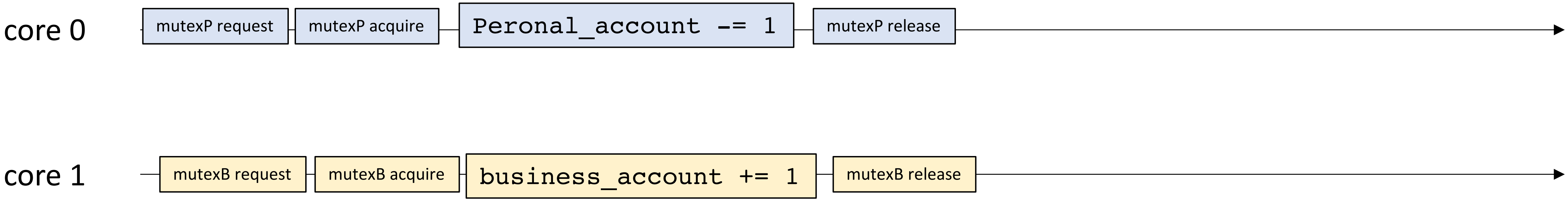


Multiple mutexes

Mutexes are objects. We can create multiple versions of them to protect different shared data.

`MutexP` for personal account
`MutexB` for business account

Critical sections across different mutexes can overlap

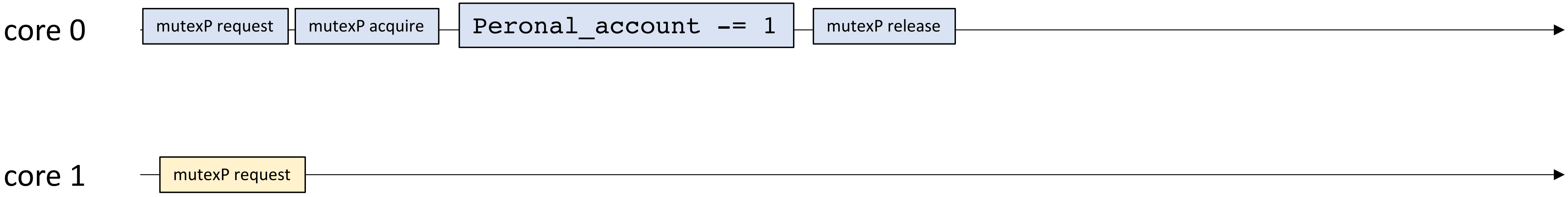


Multiple mutexes

Mutexes are objects. We can create multiple versions of them to protect different shared data.

`MutexP` for personal account
`MutexB` for business account

Critical sections across different mutexes can overlap

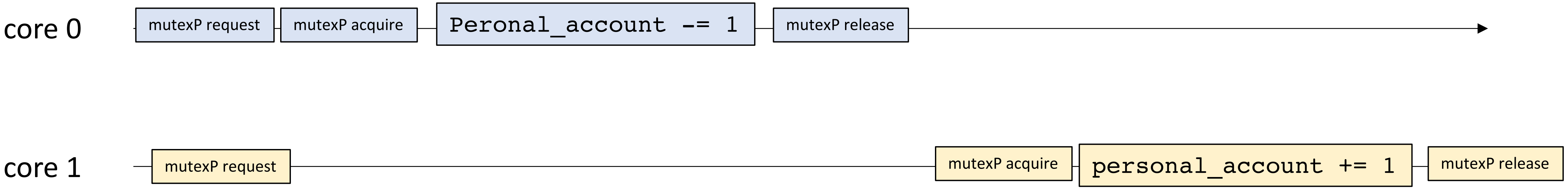


Multiple mutexes

Mutexes are objects. We can create multiple versions of them to protect different shared data.

`MutexP` for personal account
`MutexB` for business account

Critical sections across different mutexes can overlap



Multiple mutexes

Mutexes are objects. We can create multiple versions of them to protect different shared data.

`MutexP` for personal account
`MutexB` for business account

Critical sections across different mutexes can overlap

Code example

Managing multiple mutexes

Consider this increasingly elaborate scheme

My accounts start being audited by two agents:

- UCSC
- IRS

- They need to examine the accounts at the same time. They need to acquire both locks

Managing multiple mutexes

Consider this increasingly elaborate scheme

My accounts start being audited by two agents:

- UCSC
- IRS

- **Code example**

Multiple mutexes

- Our program deadlocked! What happened?

Multiple mutexes

- Our program deadlocked! What happened?

Multiple mutexes

- Our program deadlocked! What happened?

Multiple mutexes

- Our program deadlocked! What happened?

Multiple mutexes

- Our program deadlocked! What happened?

Multiple mutexes

- Our program deadlocked! What happened?

Multiple mutexes

- Our program deadlocked! What happened?

Multiple mutexes

- Our program deadlocked! What happened?

IRS has the personal mutex and won’t release it until it acquires the business mutex.
UCSC has the business mutex and won’t release it until it acquires the personal mutex.

This is called a deadlock!

Multiple mutexes

- Our program deadlocked! What happened?

- Fix: Acquire mutexes in the same order

- Proof sketch by contradiction
  - Thread 0 is holding mutex X waiting for mutex Y
  - Thread 1 is holding mutex Y waiting for mutex X

Assume the order that you acquire mutexes is X then Y
Thread 1 cannot hold mutex Y without holding mutex X.
Thread 1 cannot hold mutex X because thread 0 is holding mutex X
Thus the deadlock cannot occur

Multiple mutexes

- Our program deadlocked! What happened?
- Fix: Acquire mutexes in the same order

Double check with testing

- Proof sketch by contradiction
  - Thread 0 is holding mutex X waiting for mutex Y
  - Thread 1 is holding mutex Y waiting for mutex X

Assume the order that you acquire mutexes is X then Y
Thread 1 cannot hold mutex Y without holding mutex X.
Thread 1 cannot hold mutex X because thread 0 is holding mutex X
Thus the deadlock cannot occur

Programming with mutexes can be HARD!

make sure all data conflicts are protected with a mutex

keep critical sections small

balance between having many mutexes (provides performance) but gives the potential for deadlocks

Towards Implementations

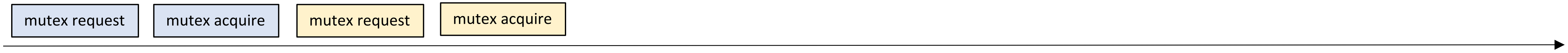
Properties of mutexes

Three properties

- **Mutual exclusion** - Only 1 thread can hold the mutex at a time. Critical sections cannot interleave

Other threads are allowed to request, but not acquire until the thread that has acquired the mutex releases it.

concurrent execution



disallowed!

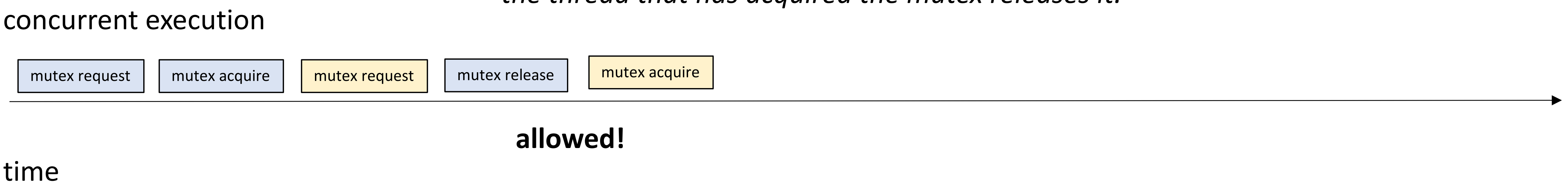
time

Properties of mutexes

Three properties

- **Mutual exclusion** - Only 1 thread can hold the mutex at a time. Critical sections cannot interleave

Other threads are allowed to request, but not acquire until the thread that has acquired the mutex releases it.



Properties of mutexes

Three properties

- **Deadlock Freedom -** If a thread has requested the mutex, and no thread currently holds the mutex, the mutex must be acquired by one of the requesting threads

Properties of mutexes

Three properties

- **Deadlock Freedom -** If a thread has requested the mutex, and no thread currently holds the mutex, the mutex must be acquired by one of the requesting threads

Program cannot hang here
Either thread 0 or thread 1 must acquire the mutex

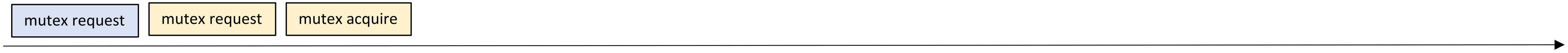
Properties of mutexes

Three properties

- **Deadlock Freedom -** If a thread has requested the mutex, and no thread currently holds the mutex, the mutex must be acquired by one of the requesting threads

Program cannot hang here
Either thread 0 or thread 1 must acquire the mutex

concurrent execution



allowed

time

Properties of mutexes

Three properties

- **Deadlock Freedom -** If a thread has requested the mutex, and no thread currently holds the mutex, the mutex must be acquired by one of the requesting threads

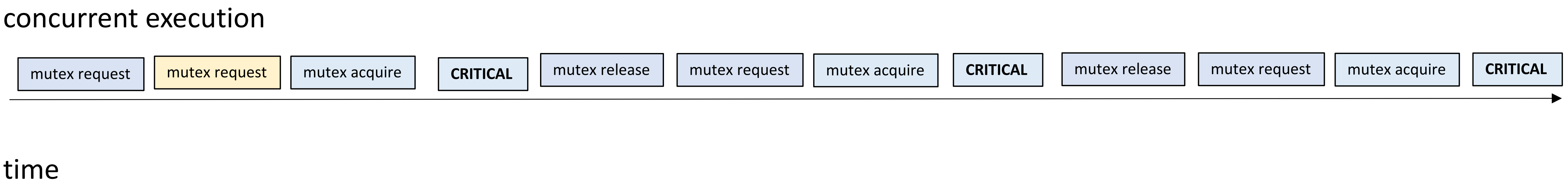
Program cannot hang here
Either thread 0 or thread 1 must acquire the mutex

Properties of mutexes

Three properties

- **Starvation Freedom** (*Optional*) - A thread that requests the mutex must eventually obtain the mutex.

Thread 1 (yellow) requests the mutex but never gets it



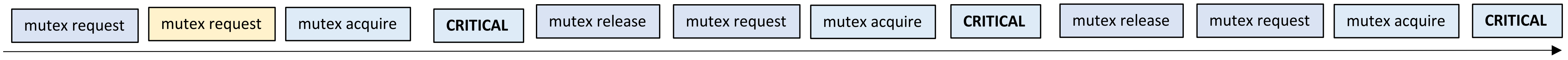
Properties of mutexes

Three properties

- **Starvation Freedom** (*Optional*) - A thread that requests the mutex must eventually obtain the mutex.

Thread 1 (yellow) requests the mutex but never gets it

concurrent execution



time

Difficult to provide in practice and timing variations usually provide this property naturally

Properties of mutexes

Recap: three properties

- **Mutual Exclusion**: Two threads cannot be in the critical section at the same time

- **Deadlock Freedom**: If a thread has requested the mutex, and no thread currently holds the mutex, the mutex must be acquired by one of the requesting threads

- **Starvation Freedom** (*optional*): A thread that requests the mutex must eventually obtain the mutex.

Building blocks

- Memory reads and memory writes
  - later: read-modify-writes
- We need to guarantee that our reads and writes actually go to memory.
  - And other properties we will see soon
- To do this, we will use C++ atomic operations

A historical perspective

- Adding concurrency support to a programming language is hard!
- The memory model defines how threads can safely share memory

- Java tried to do this,

wikipedia

> The original Java memory model, developed in 1995, was widely perceived as broken, preventing many runtime optimizations and not providing strong enough guarantees for code safety. It was updated through the Java Community Process, as Java Specification Request 133 (JSR-133), which took effect in 2004, for Tiger (Java 5.0).[1][2]

Brian Goetz (2019)

> It is worth noting that broken techniques like double-checked locking are still broken under the new memory model, a

A historical perspective

- How is C++?

- Has issues (imprecise, not modular)
  - but at least considered safe
  - Specification makes it difficult to reason about all programs
  - Open problem!

- Luckily mutexes (and their implementations) avoid the problematic areas of the language!

Our primitive instructions

- Types: atomic_int

- Interface (C++ provides overloaded operators):
  - load
  - store

- Properties:
  - loads and stores will always go to memory.
  - compiler memory fence
  - hardware memory fence

Atomic properties

- loads and stores will always go to memory

- Compiler example, performance difference

Atomic properties

- loads and stores will always go to memory
- Compiler example, performance difference
- Compiler makes reasoning about parallel code hard, but big performance improvements:
  - O(2048) vs. O(1)

Atomic properties

- Compiler Fence

- Compiler can be aggressive with memory operations:
  - For non-atomic memory locations, the following optimizations are valid

Atomic properties

- Compiler Fence

- Compiler can be aggressive with memory operations:
  - For non-atomic memory locations, the following optimizations are valid

```
a[i] = 0;
a[i] = 1;
```

can be optimized to:

```
a[i] = 1;
```

Atomic properties

- Compiler Fence

- Compiler can be aggressive with memory operations:
  - For non-atomic memory locations, the following optimizations are valid

```
a[i] = 0;
a[i] = 1;
```

can be optimized to:

```
a[i] = 1;
```

```
x = a[i];
x2 = a[i];
```

can be optimized to:

```
x = a[i];
x2 = x;
```

Atomic properties

- Compiler Fence

- Compiler can be aggressive with memory operations:
  - For non-atomic memory locations, the following optimizations are valid

```
a[i] = 0;
a[i] = 1;
```

can be optimized to:

```
a[i] = 1;
```

```
x = a[i];
x2 = a[i];
```

can be optimized to:

```
x = a[i];
x2 = x;
```

```
a[i] = 6;
x = a[i];
```

can be optimized to:

```
x = 6;
```

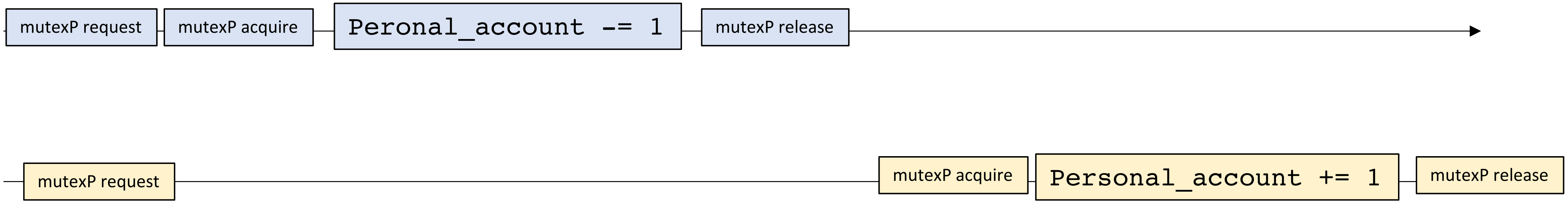
Atomic properties

- Compiler Fence
- Compiler can be aggressive with memory operations:
  - For non-atomic memory locations, the following optimizations are valid
- And many others... especially when you consider mixing with other optimizations
  - Very difficult to understand when/where memory accesses will actually occur in your code

Atomic properties

- Compiler Fence

Compiler cannot keep personal_account in a register past the mutex

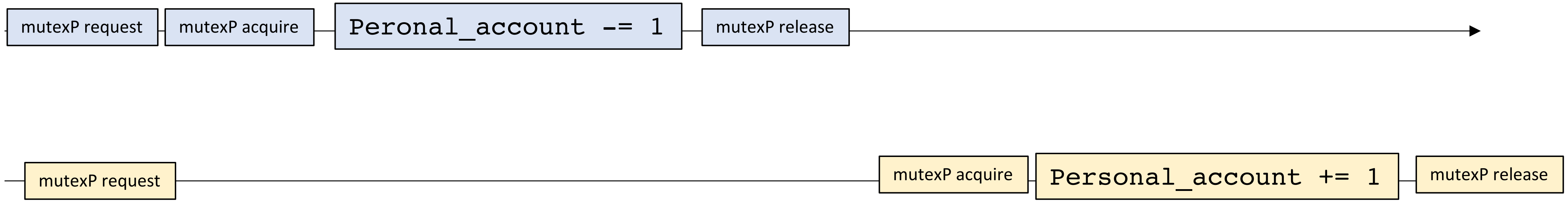


because this thread needs to see the updated view

Atomic properties

- Compiler Fence

what can go wrong if the compiler doesn't write values to memory?

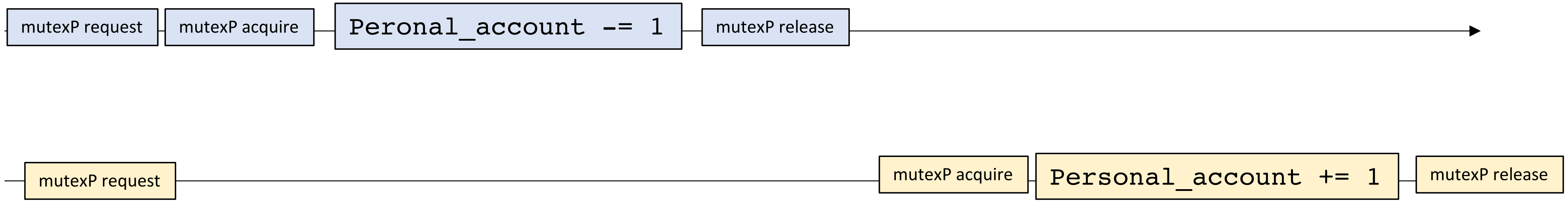


Atomic properties

- Compiler Fence

what can go wrong if the compiler doesn't write values to memory?

initially personal_account is 0

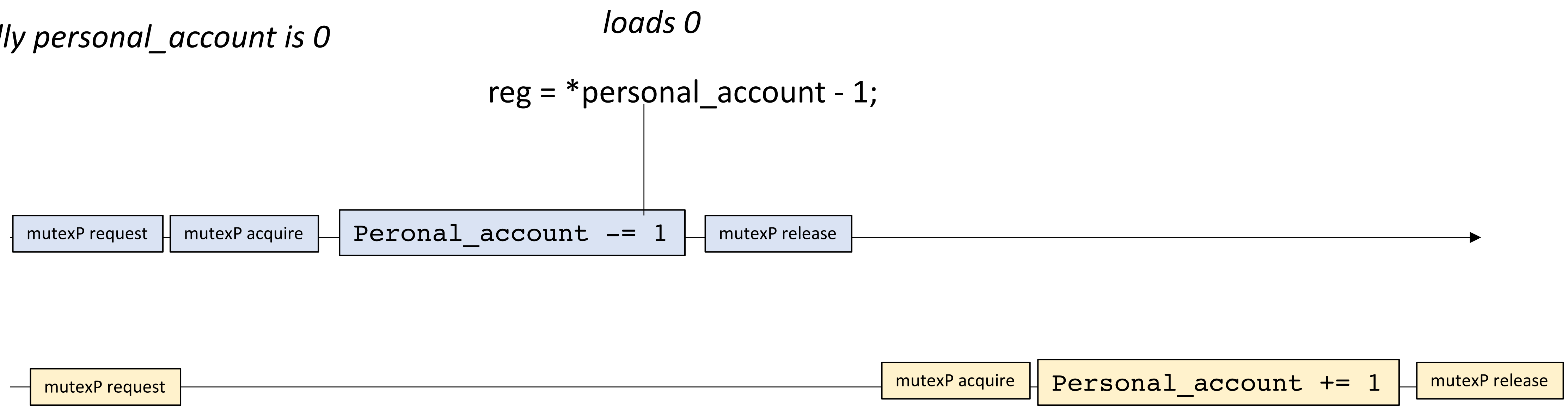


Atomic properties

- Compiler Fence

what can go wrong if the compiler doesn't write values to memory?

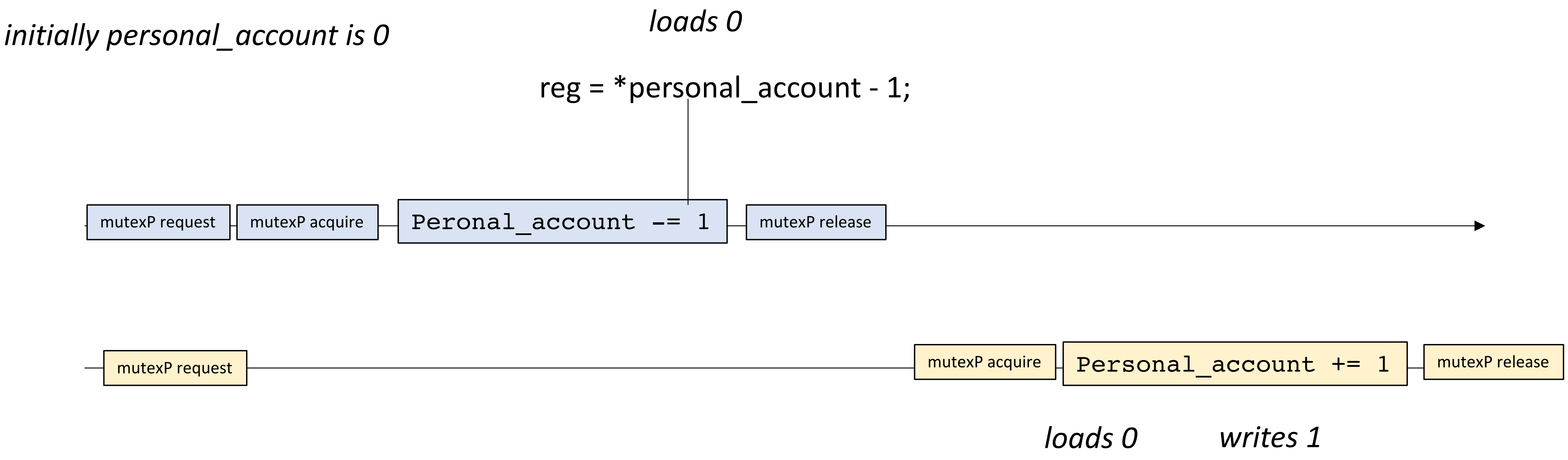
initially personal_account is 0



Atomic properties

- Compiler Fence

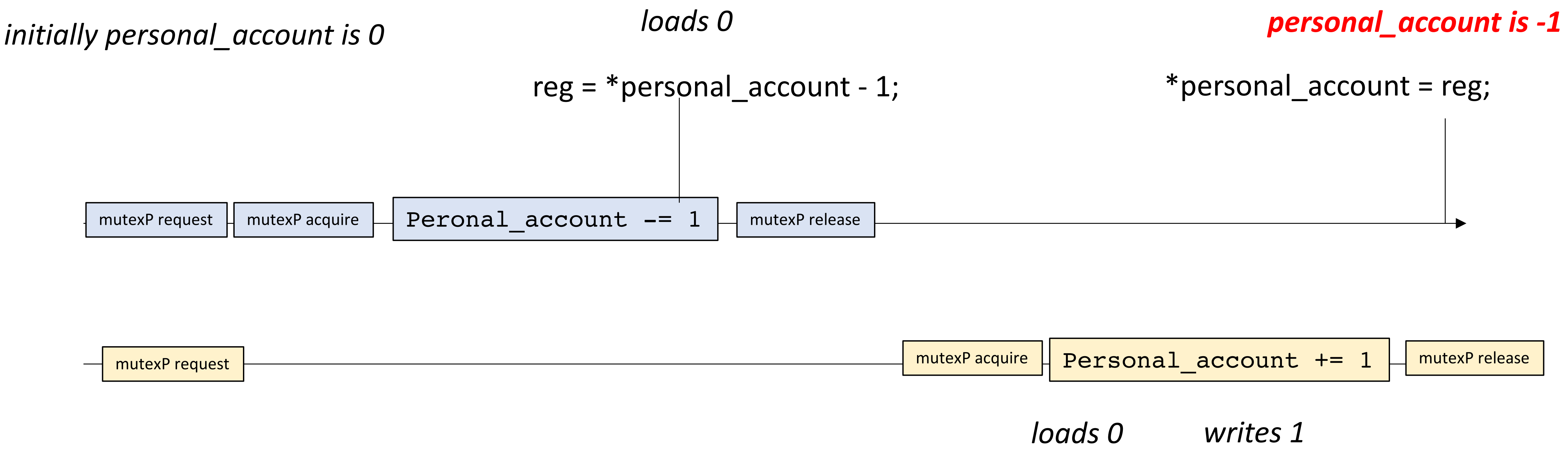
what can go wrong if the compiler doesn't write values to memory?



Atomic properties

- Compiler Fence

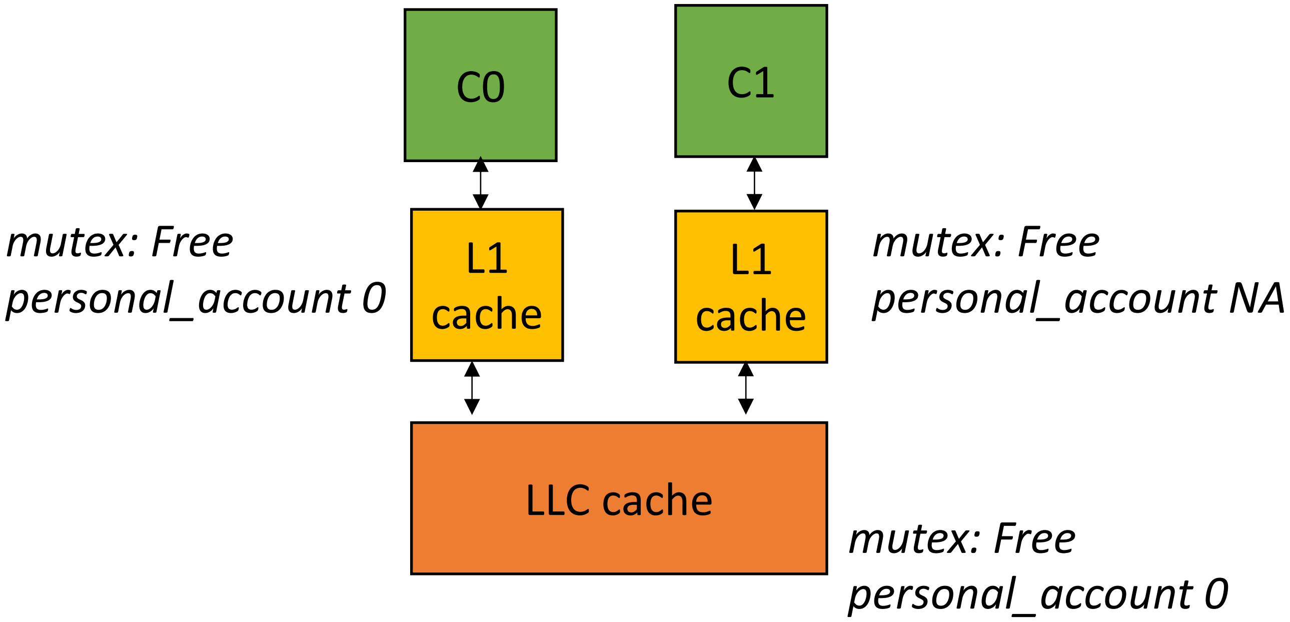
what can go wrong if the compiler doesn't write values to memory?

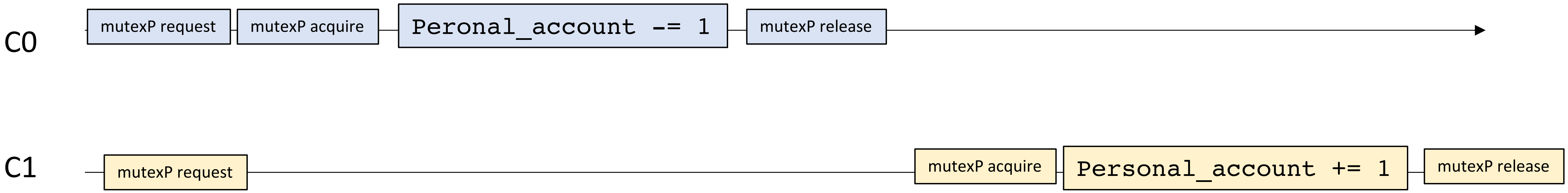


Atomic properties

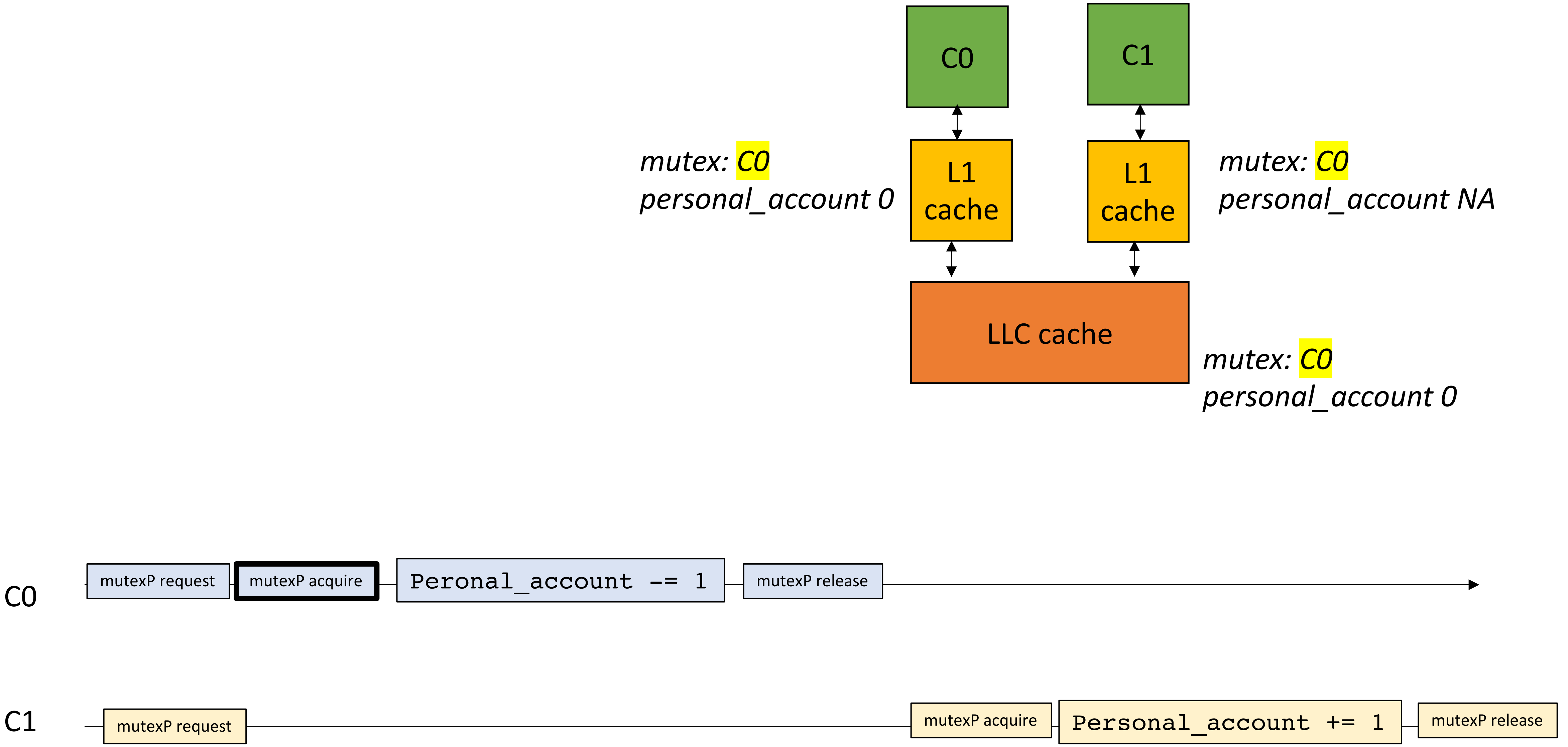
- Also provides a memory barrier

- Memory Fence (or Memory Barrier)

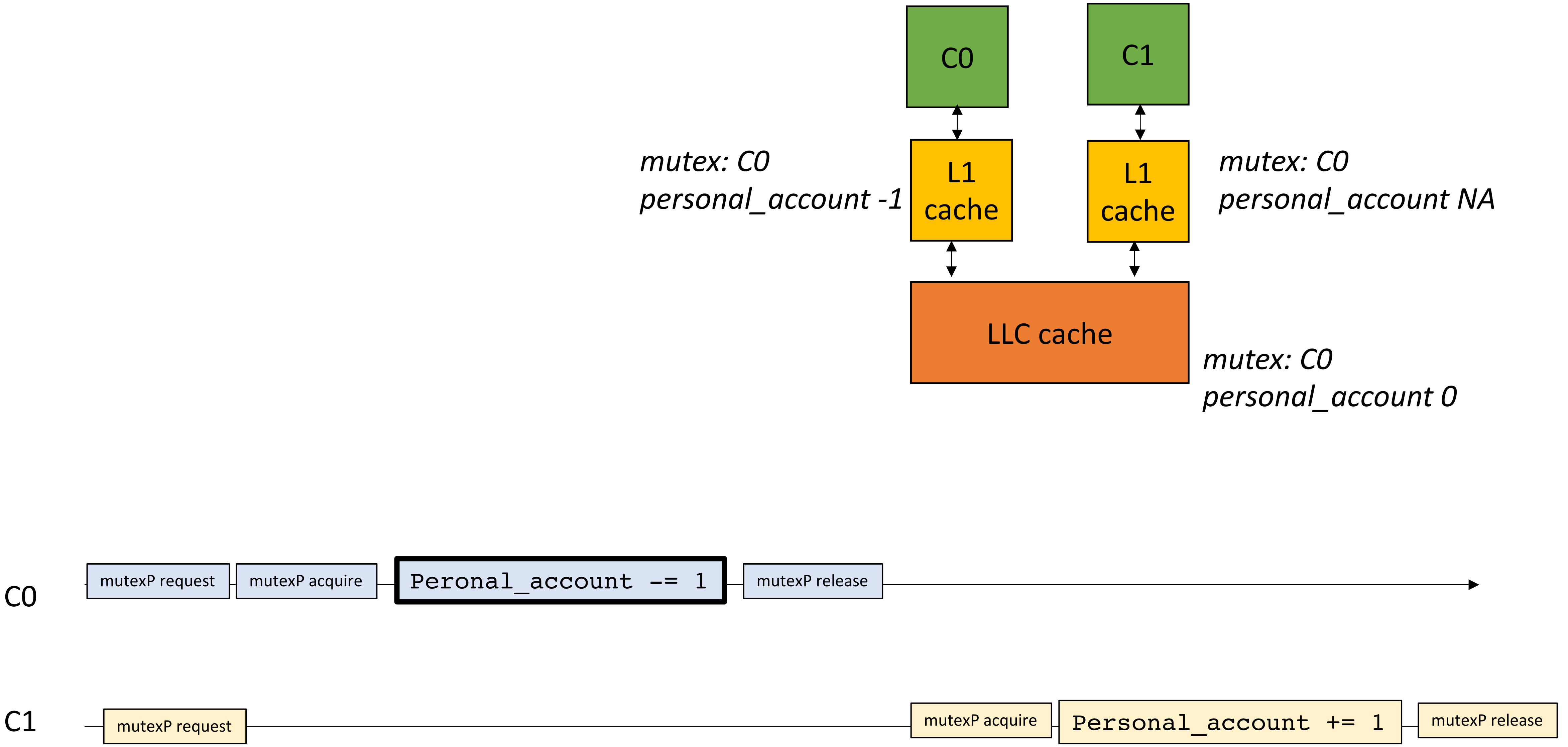




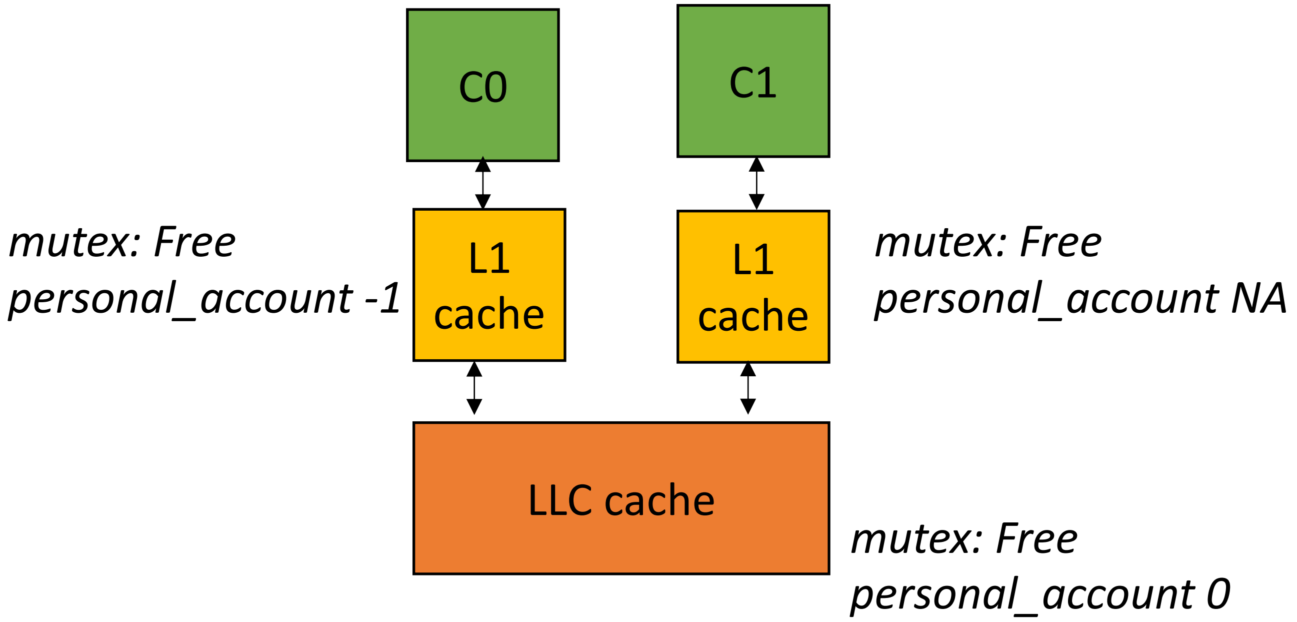
- Memory Fence (or Memory Barrier)

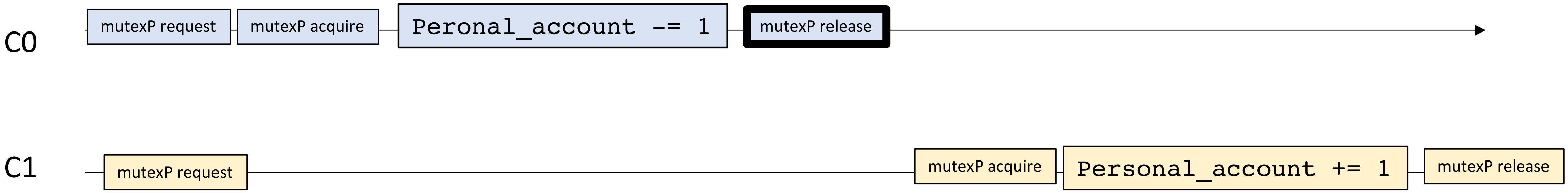


- Memory Fence (or Memory Barrier)

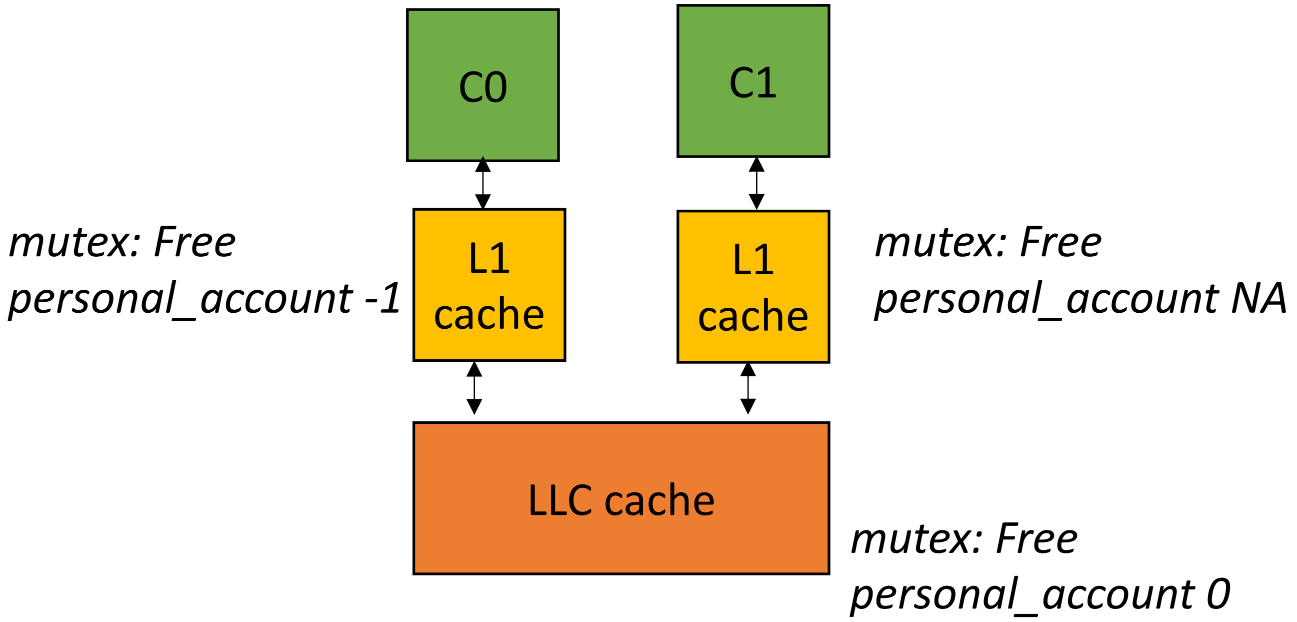


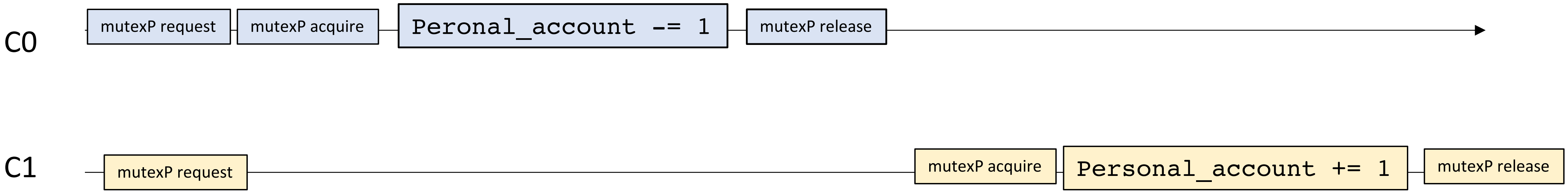
- Memory Fence (or Memory Barrier)



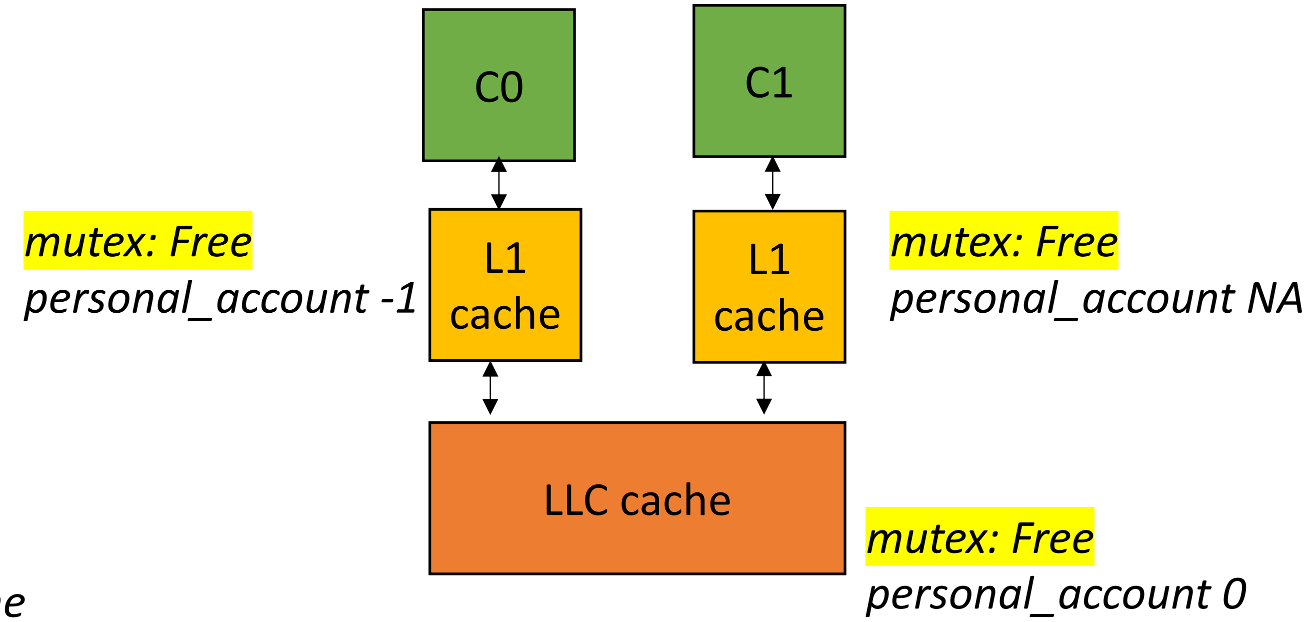


- Memory Fence (or Memory Barrier)

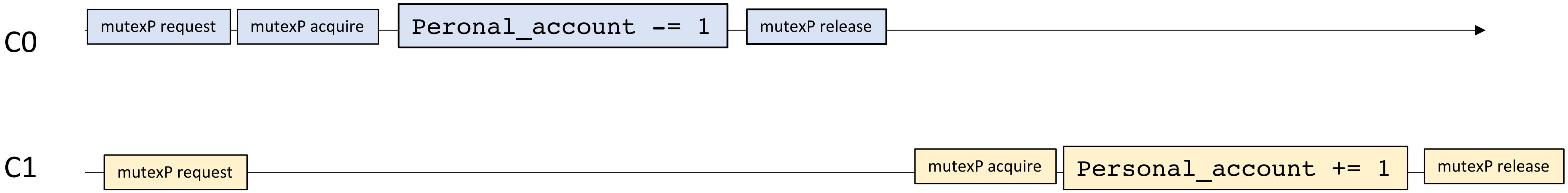




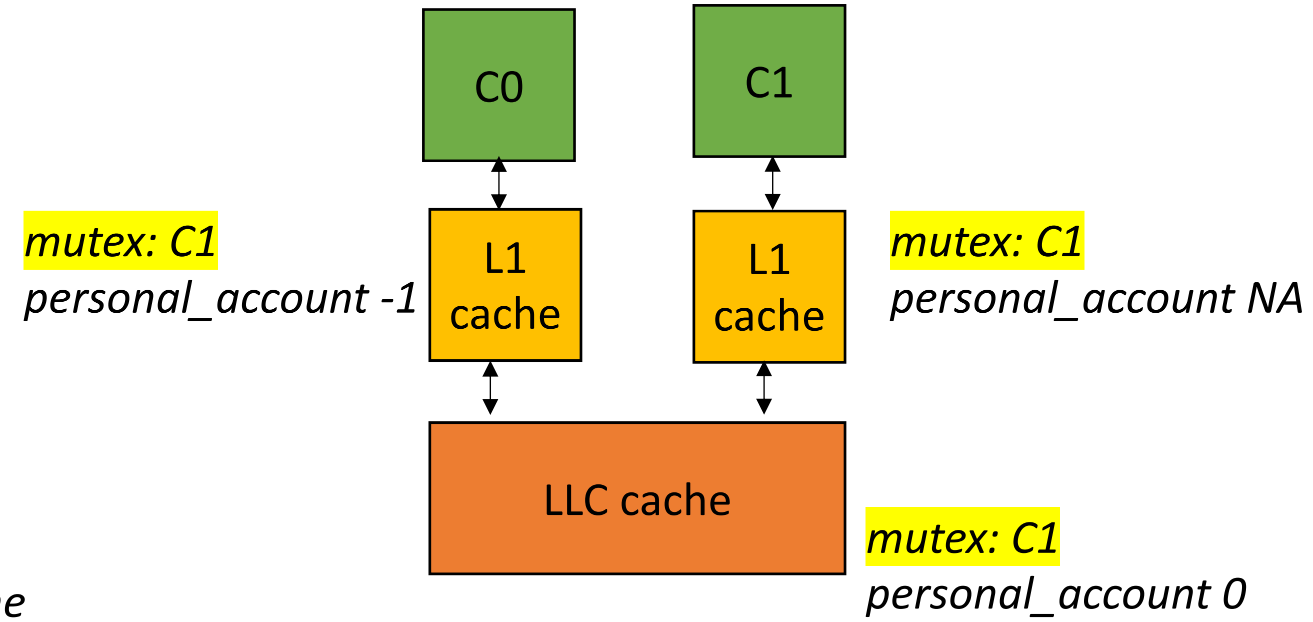
- Memory Fence (or Memory Barrier)



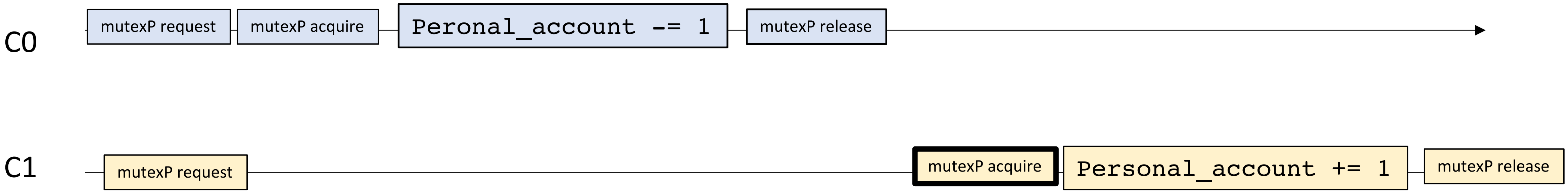
Possible for the mutex message to travel through the cache before the personal_account information!



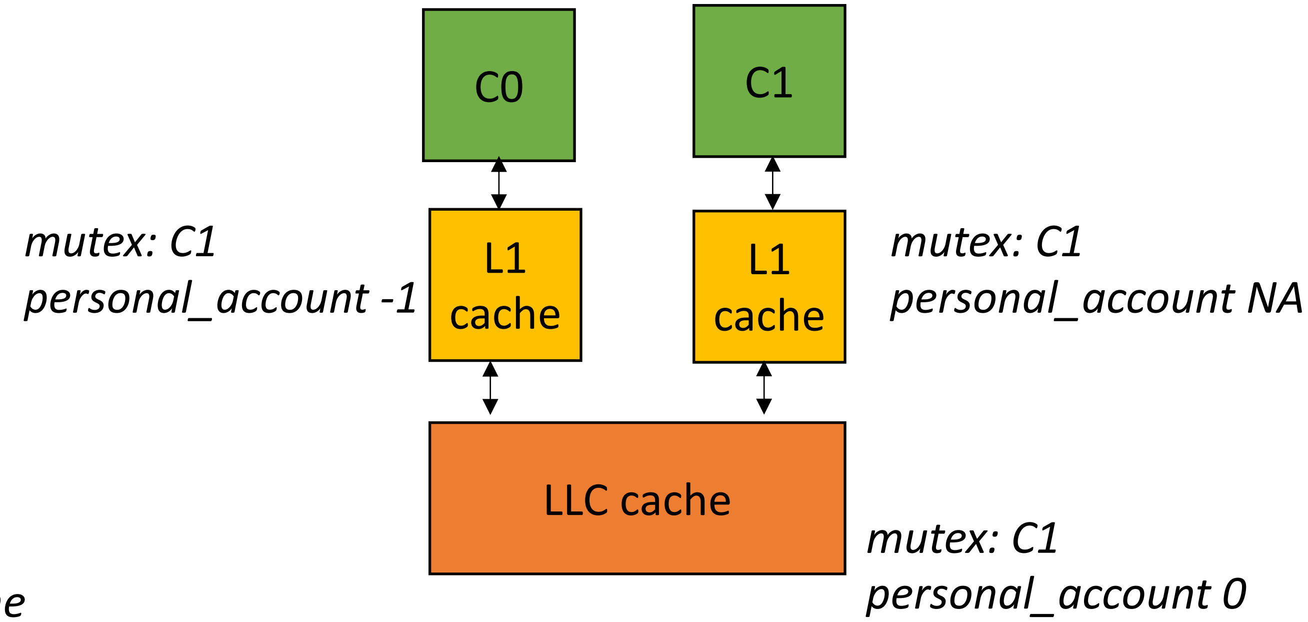
- Memory Fence (or Memory Barrier)



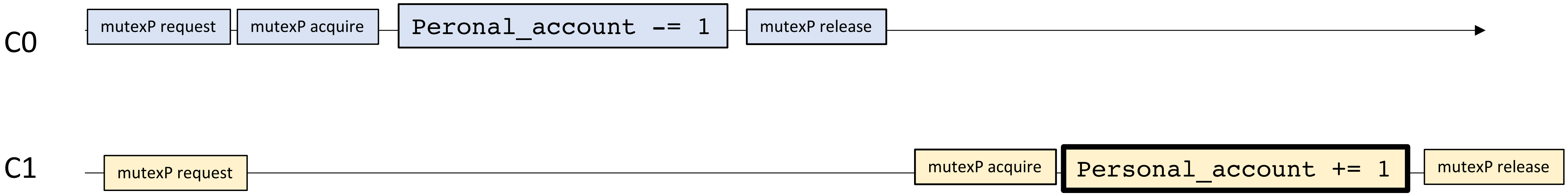
Possible for the mutex message to travel through the cache before the personal_account information!



- Memory Fence (or Memory Barrier)

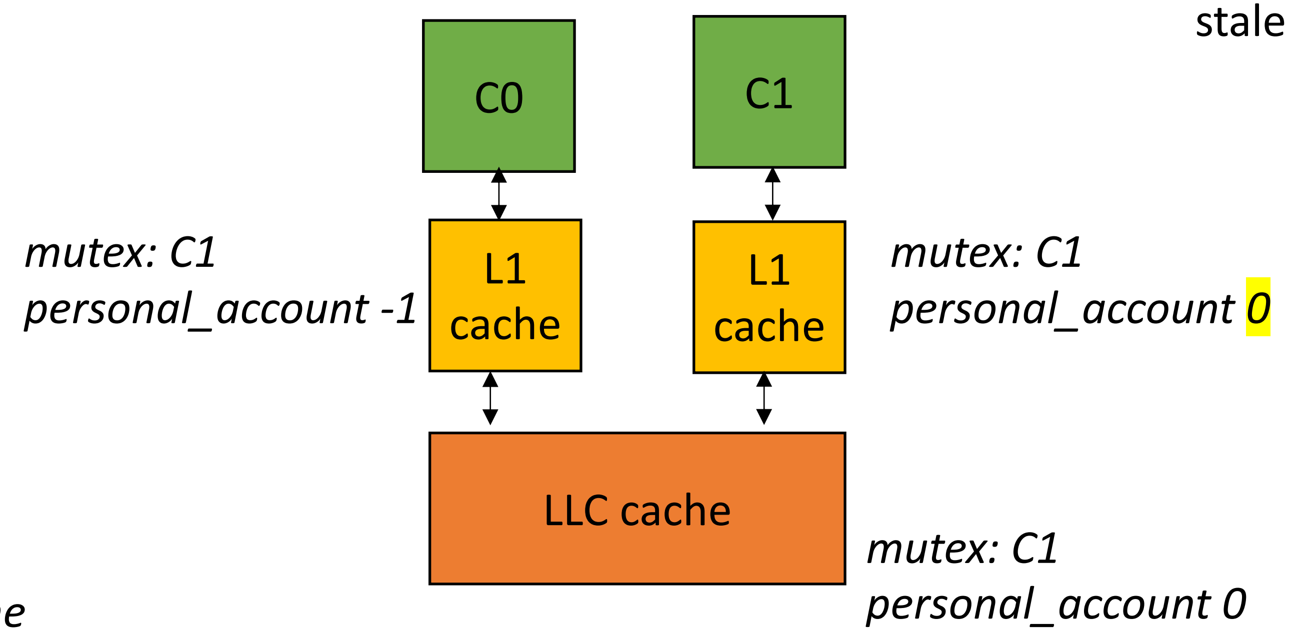


Possible for the mutex message to travel through the cache before the personal_account information!

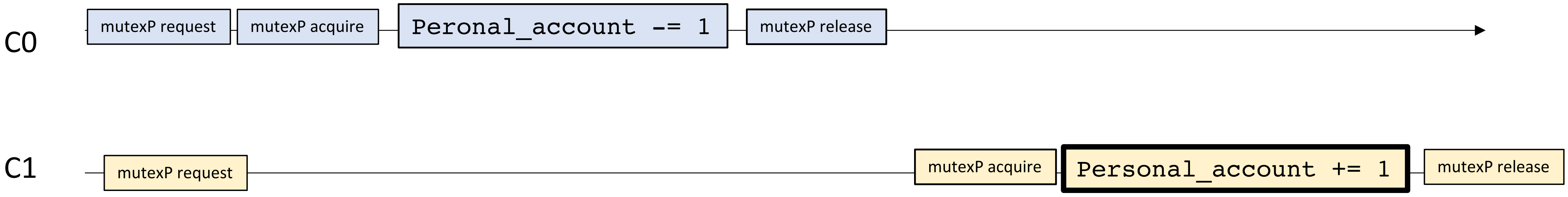


- Memory Fence (or Memory Barrier)

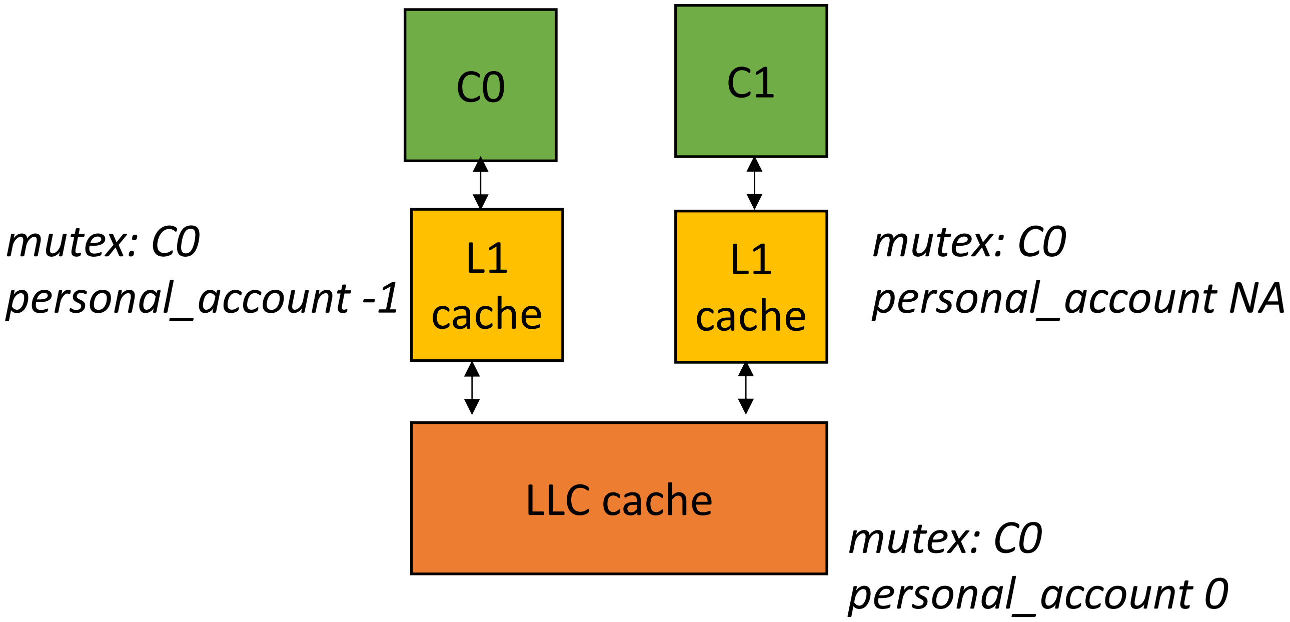
stale value!



Possible for the mutex message to travel through the cache before the personal_account information!

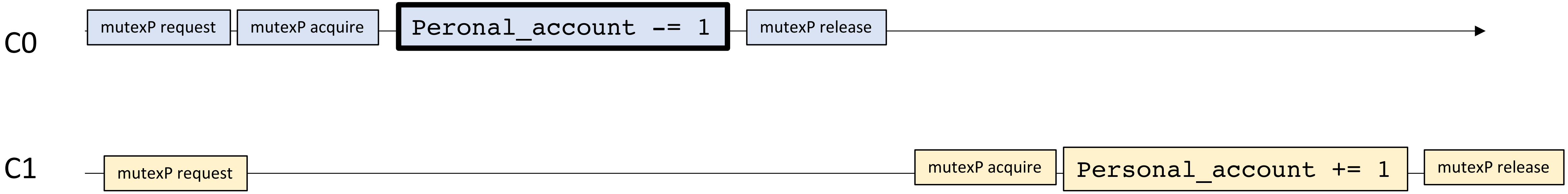


- Memory Fence (or Memory Barrier)

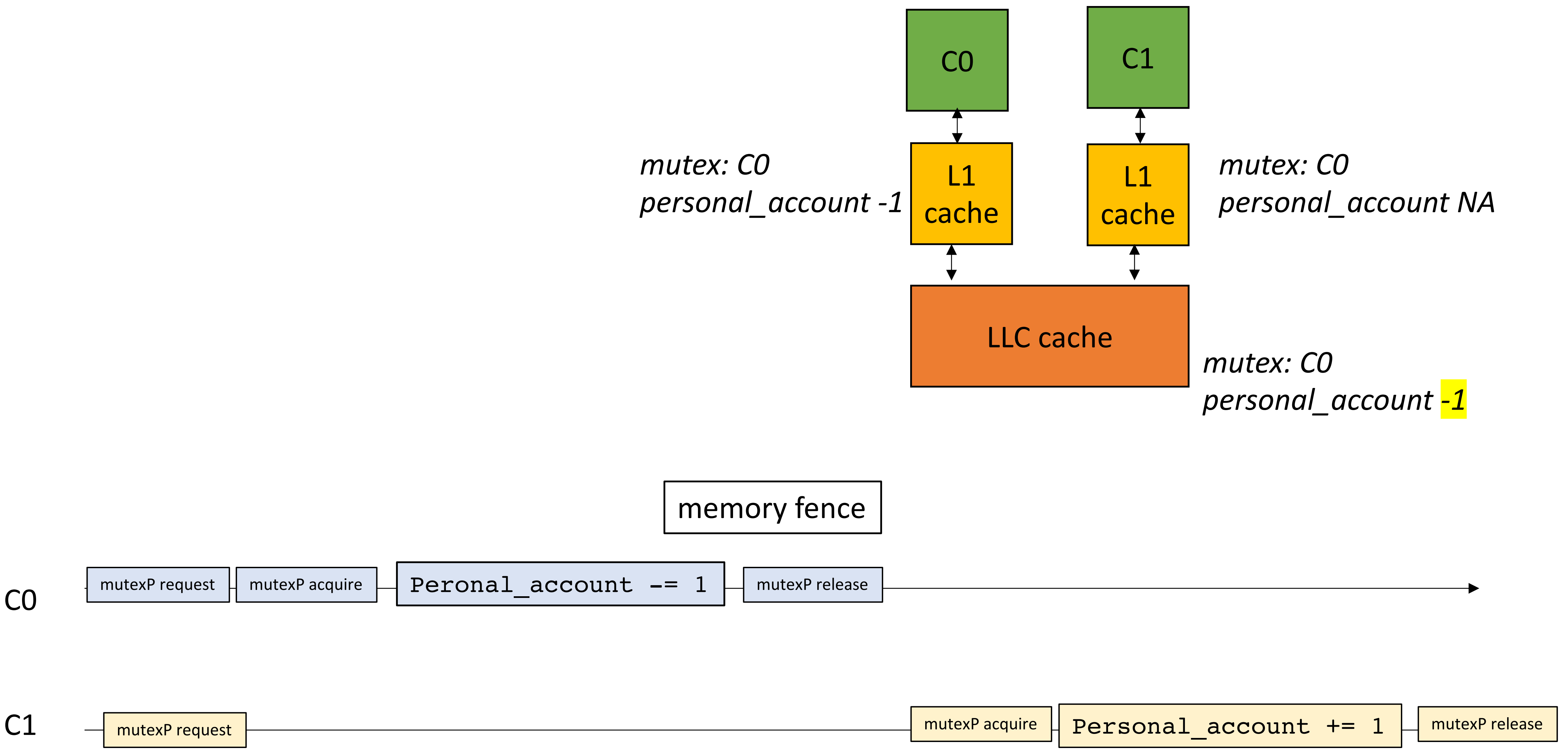


rewind

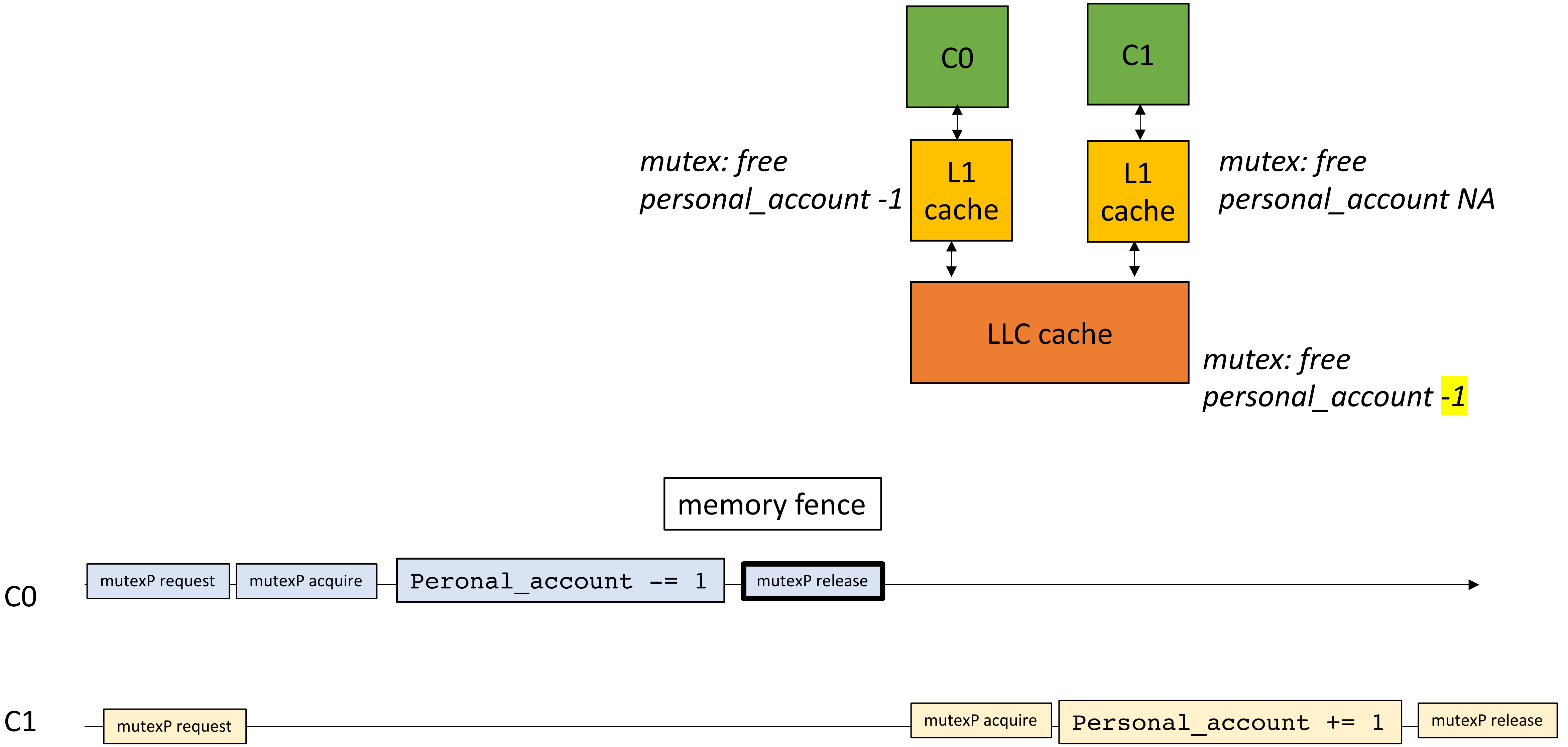
memory fence



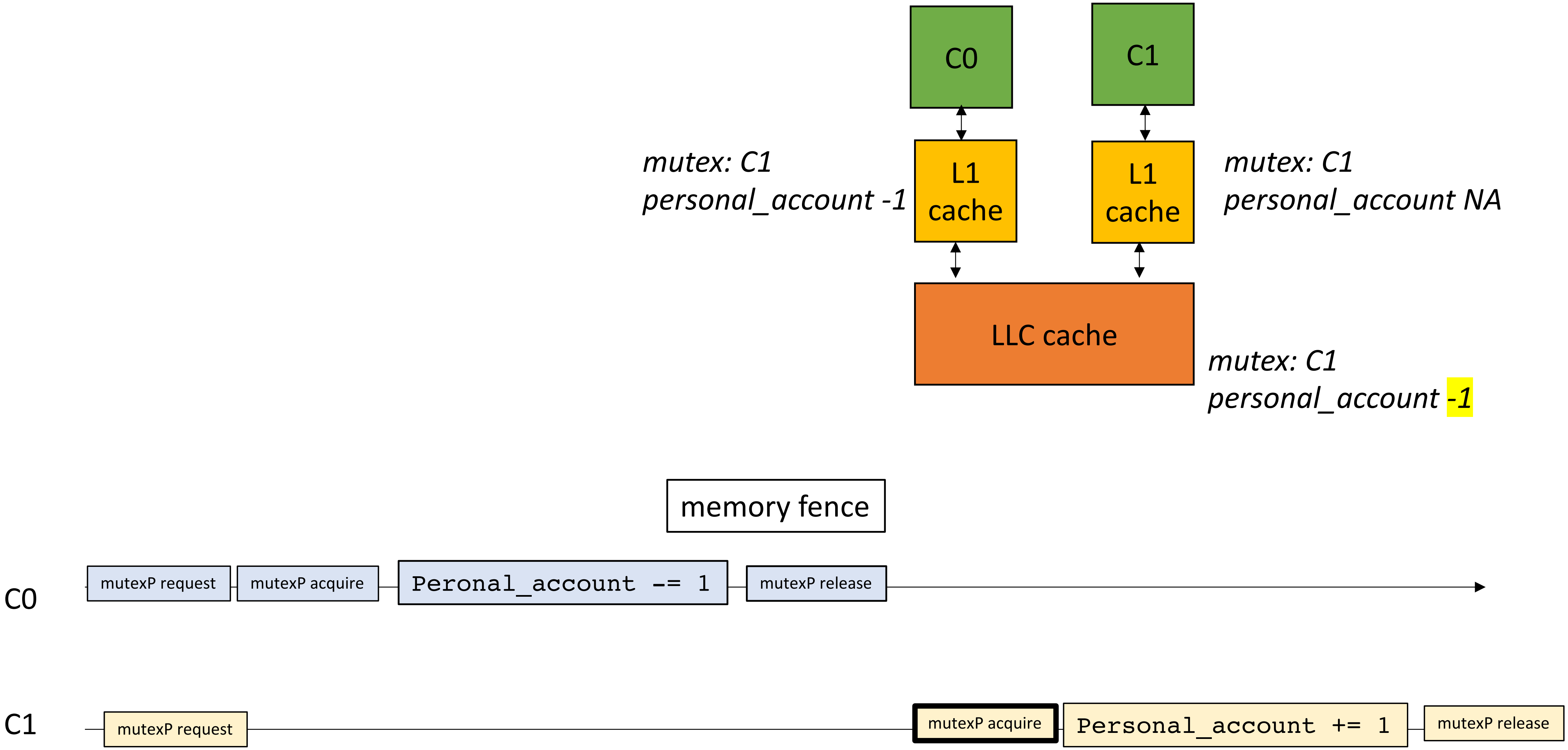
- Memory Fence (or Memory Barrier)



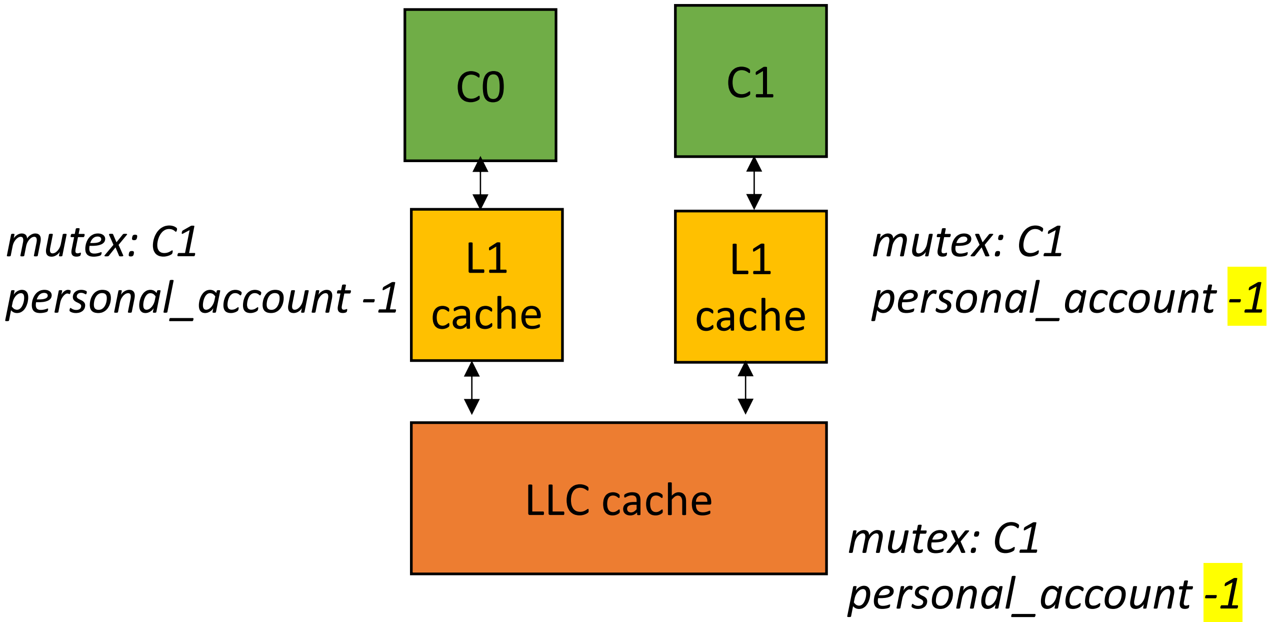
- Memory Fence (or Memory Barrier)

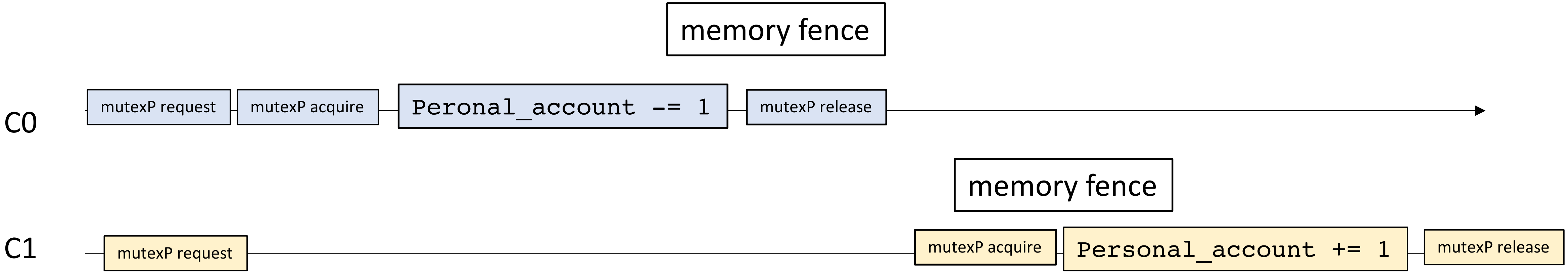


- Memory Fence (or Memory Barrier)

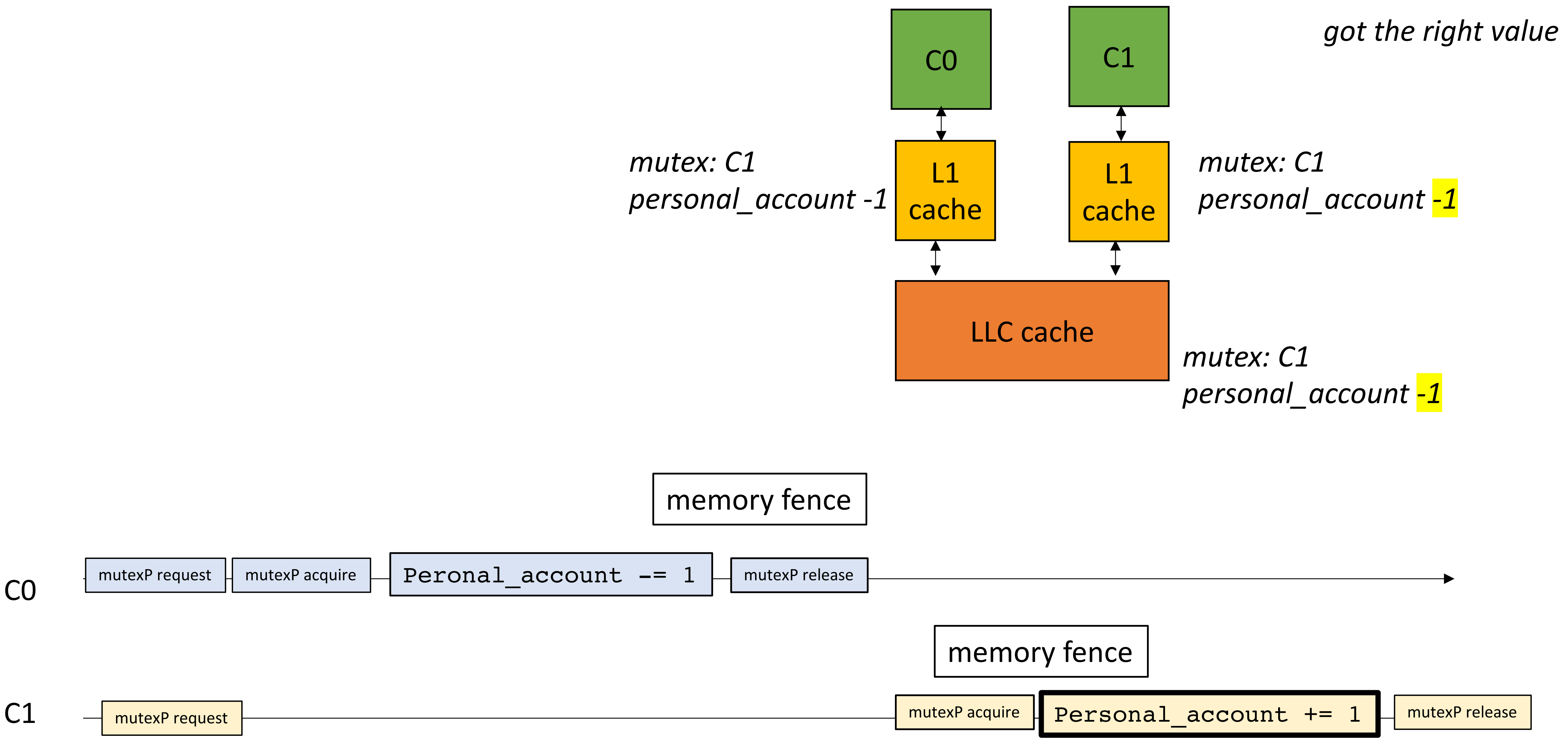


- Memory Fence (or Memory Barrier)





- Memory Fence (or Memory Barrier)



- **Memory Fence (or Memory Barrier)**

different architectures have different memory barriers

Intel X86 naturally manages caches in order

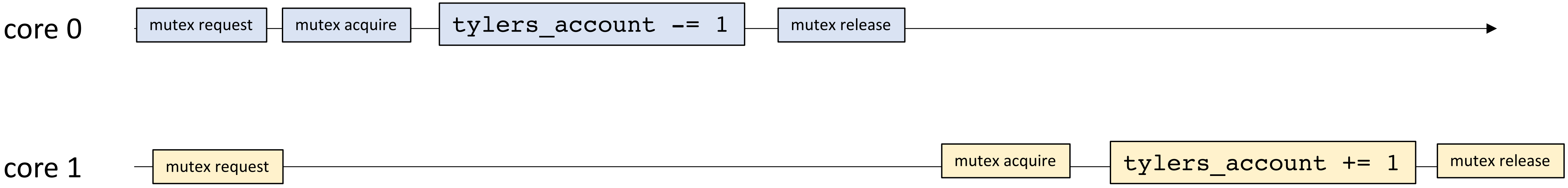
ARM and PowerPC let cache values flow out-of-order
GPUs let caches flow out-of-order

RISC-V has two models:
  more like x86: easier to program
  more like ARM: faster and more energy efficient

For mutexes, atomics will naturally handle the memory fences for us!

Atomics

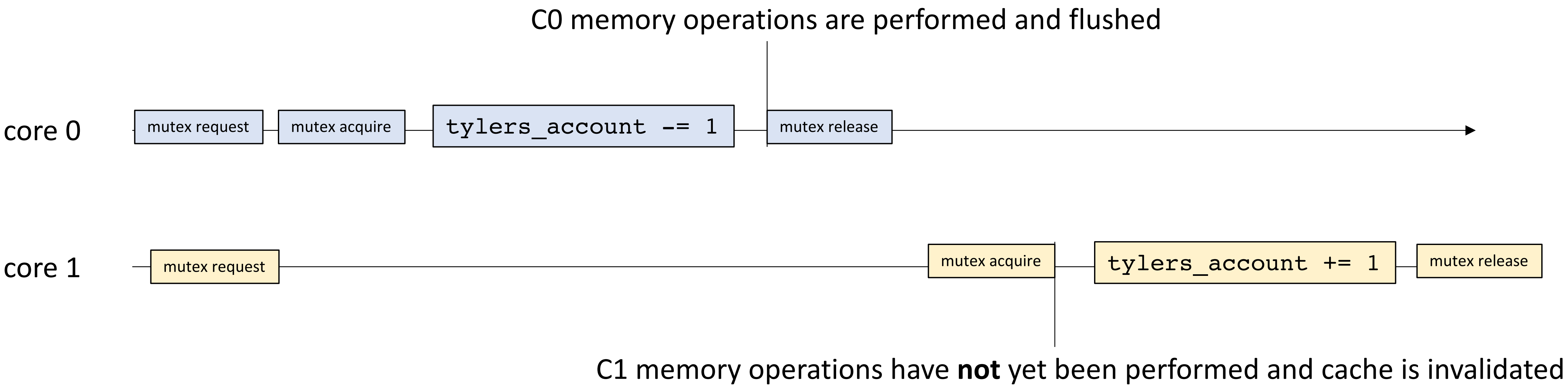
- What do those fences (compiler and memory) give us?
- Atomics were designed so that we can implement things like mutexes!



Atomics

- What do those fences (compiler and memory) give us?

- Atomics were designed so that we can implement things like mutexes!



Thanks!

- Next time:
  - Work on a simple mutex implementation using atomics

- Work on your homework and use office hours, piazza and tutors

- Do the quiz!