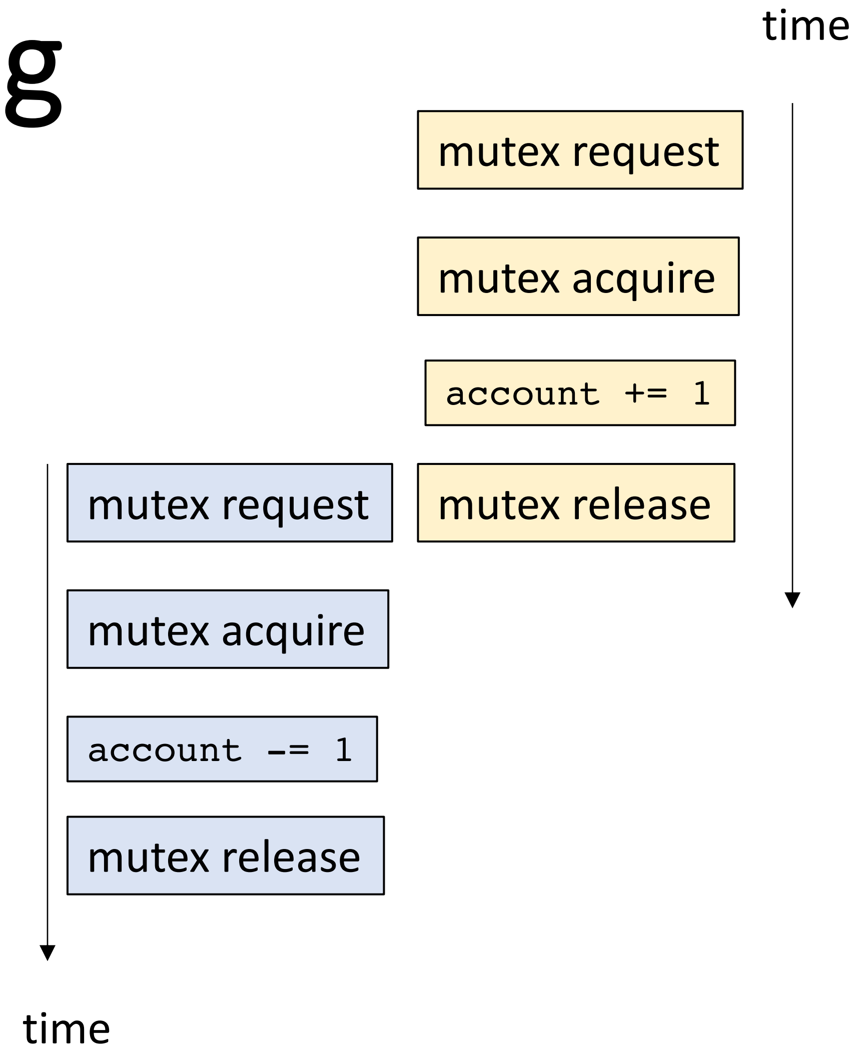
CSE113: Parallel Programming

April 13, 2021

- **Topic**: Introduction to Mutual Exclusion
  - Reasoning about concurrent programs
  - Mutual exclusion properties
  - Multiple mutexes



Announcements

- No more asynchronous lectures planned

- Homework 1 is posted:
  - Due April 22

- My office hours are on Wednesday, 3 - 5 PM
  - TAs have office hours daily
  - They are more helpful with tool flows (docker, VSCode)

- *New module: Mutual Exclusion!*

Lecture Schedule

- Canvas Quiz
- Notes on homework
- Reasoning about concurrency
- Mutual exclusion
- Multiple Mutexes

Lecture Schedule

- **Canvas Quiz**
- Notes on homework
- Reasoning about concurrency
- Mutual exclusion
- Multiple mutexes

Quiz

- Publishing quiz on canvas:
  - Open for 5 minutes

Quiz

- Publishing quiz on canvas:
  - Open for 5 minutes

- Go over questions

Lecture Schedule

- Canvas Quiz
- **Notes on homework**
- Reasoning about concurrency
- Mutual exclusion
- Multiple mutexes

Homework

- Demo on terminal

Lecture Schedule

- Canvas Quiz
- Notes on homework
- **Reasoning about concurrency**
- Mutual exclusion
- Multiple mutexes

Embarrassingly parallel

Embarrassingly parallel

Embarrassingly parallel

From Wikipedia, the free encyclopedia

In parallel computing, an **embarrassingly parallel** workload or problem (also called **embarrassingly parallelizable**, **perfectly parallel**, **delightfully parallel** or **pleasingly parallel**) is one where little or no effort is needed to separate the problem into a number of parallel tasks.[1] This is often the case where there is little or no dependency or need for communication between those parallel tasks, or for results between them.[2]

For this class: A multithreaded program is ***embarrassingly parallel*** if there are no ***data-conflicts.***

A ***data conflict*** is where one thread writes to a memory location that another thread reads or writes to concurrently and without sufficient ***synchronization***.

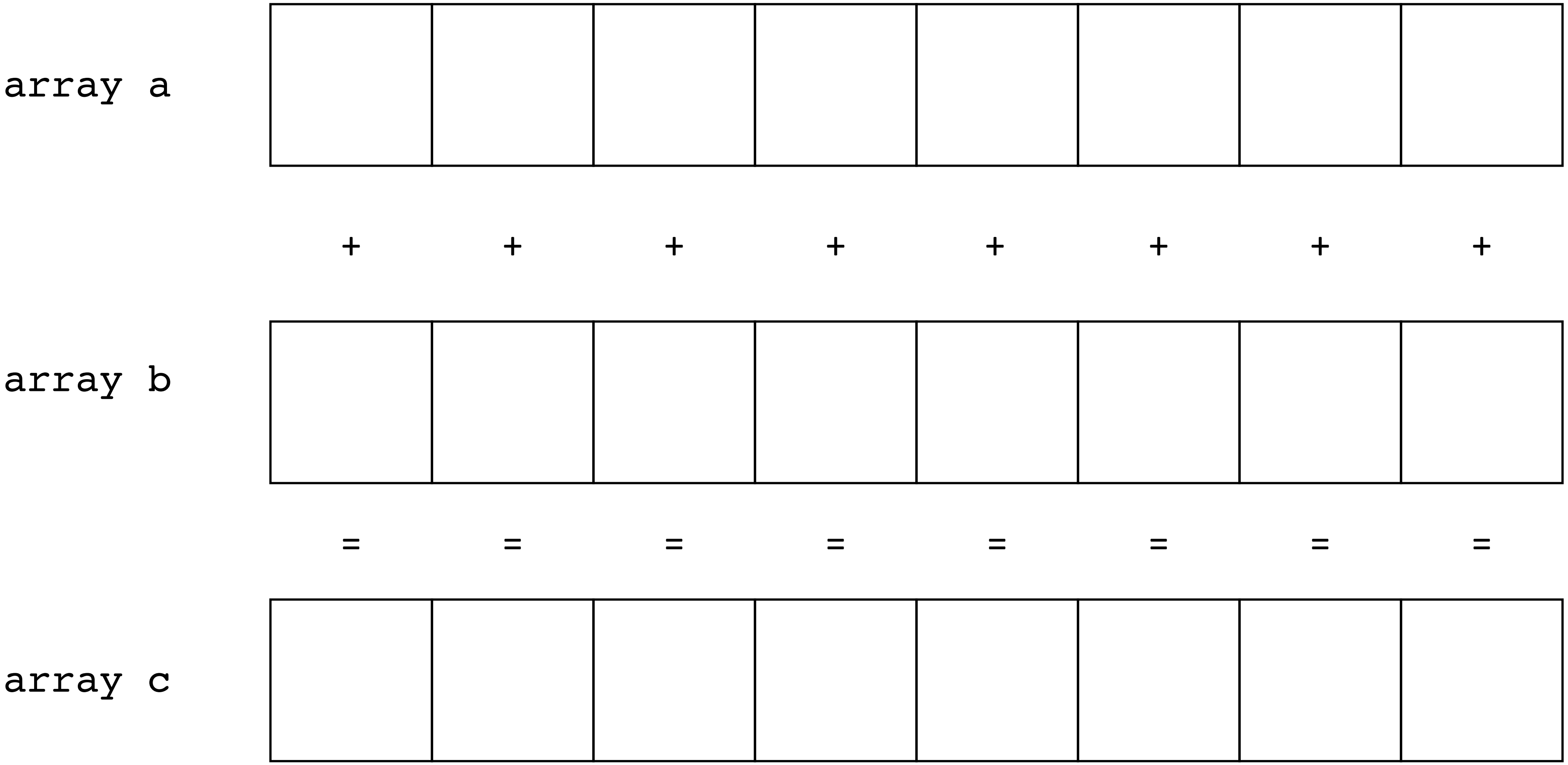
Embarrassingly parallel

- Consider the following program:

There are 3 arrays: `a, b, c.`

We want to compute `c[i] = a[i] + b[i]`

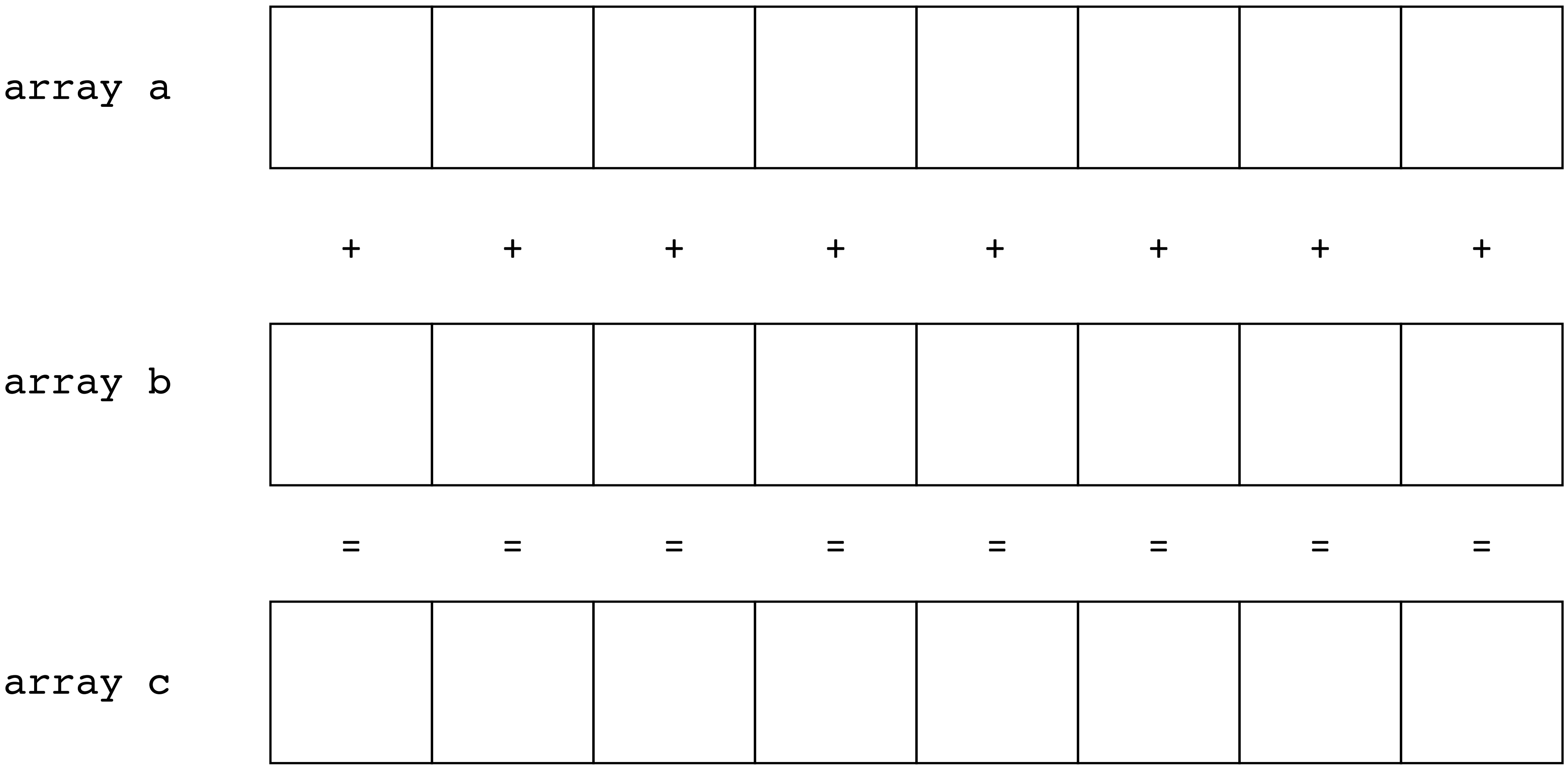
Embarrassingly parallel



Embarrassingly parallel

Computation can easily be divided into threads

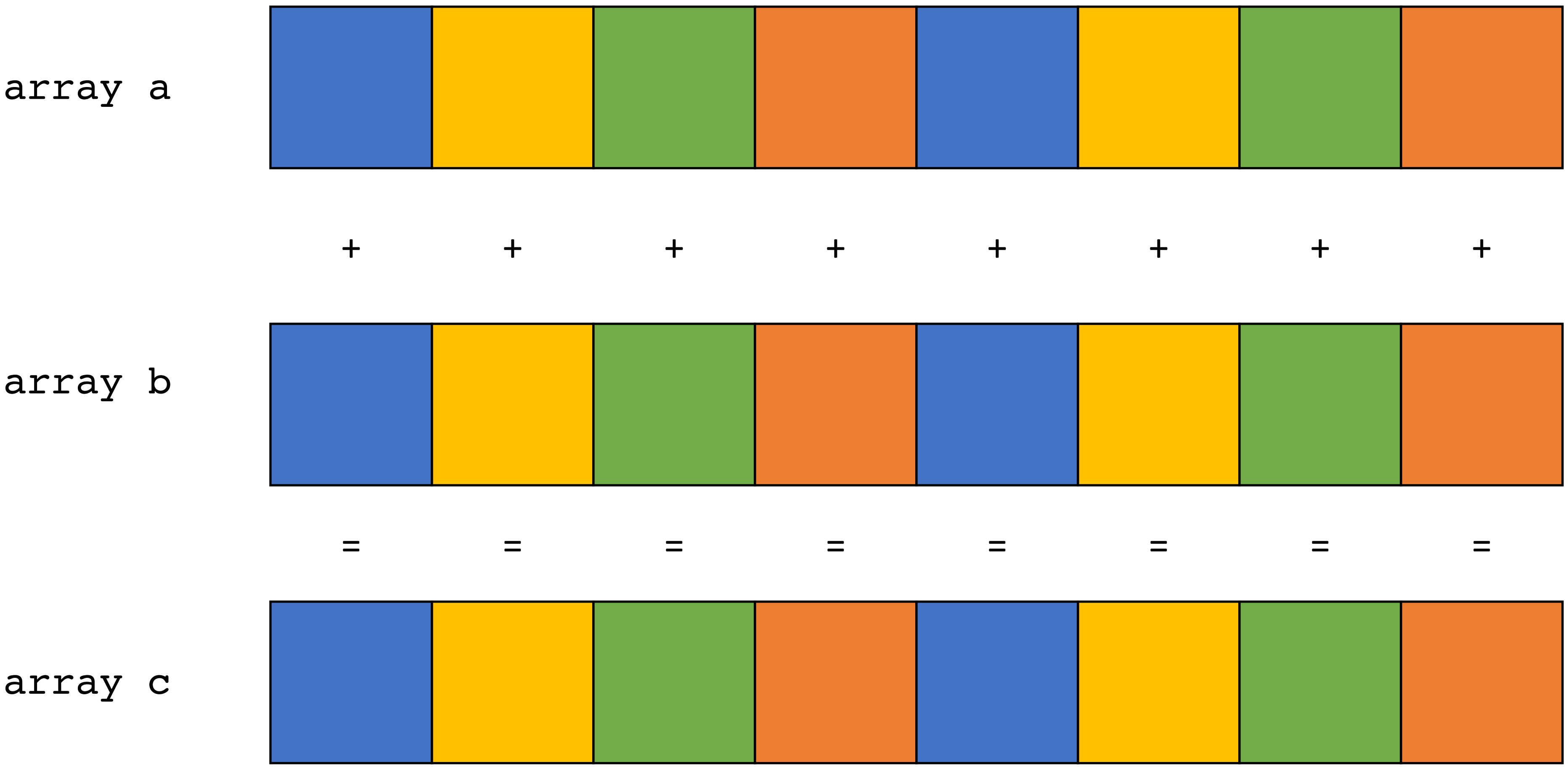
Thread 0 - Blue
Thread 1 - Yellow
Thread 2 - Green
Thread 3 - Orange



Embarrassingly parallel

Computation can easily be divided into threads

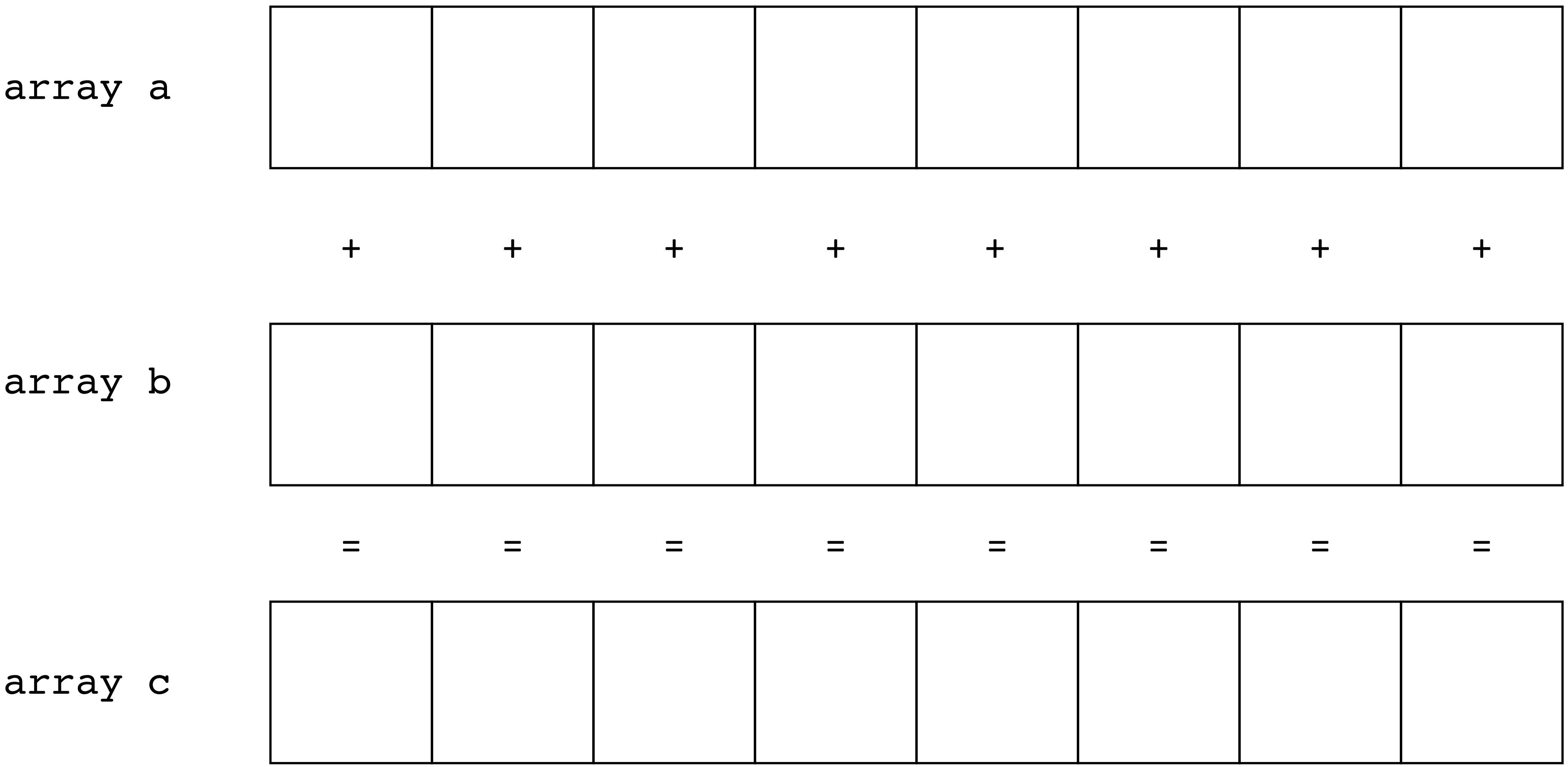
Thread 0 - Blue
Thread 1 - Yellow
Thread 2 - Green
Thread 3 - Orange



Embarrassingly parallel

Computation can easily be divided into threads

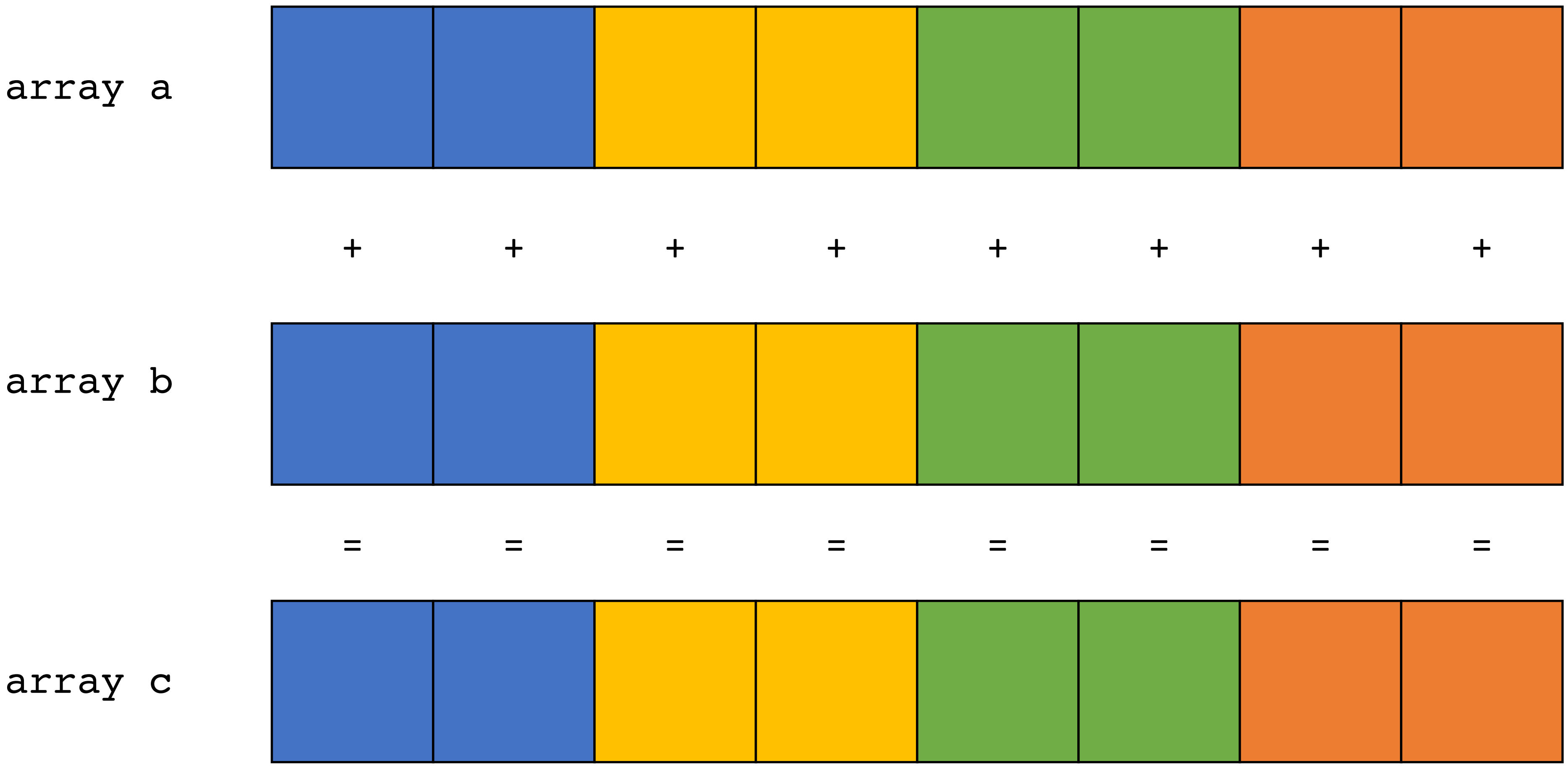
Thread 0 - Blue
Thread 1 - Yellow
Thread 2 - Green
Thread 3 - Orange



Embarrassingly parallel

Computation can easily be divided into threads

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



Embarrassingly parallel

- The different parallelization strategies will probably have different performance behaviors.

- But they are both embarrassingly parallel solutions to the problem

- There is lots of research into making these types of programs go fast!
    - but this module will focus on programs that require synchronization

Embarrassingly parallel

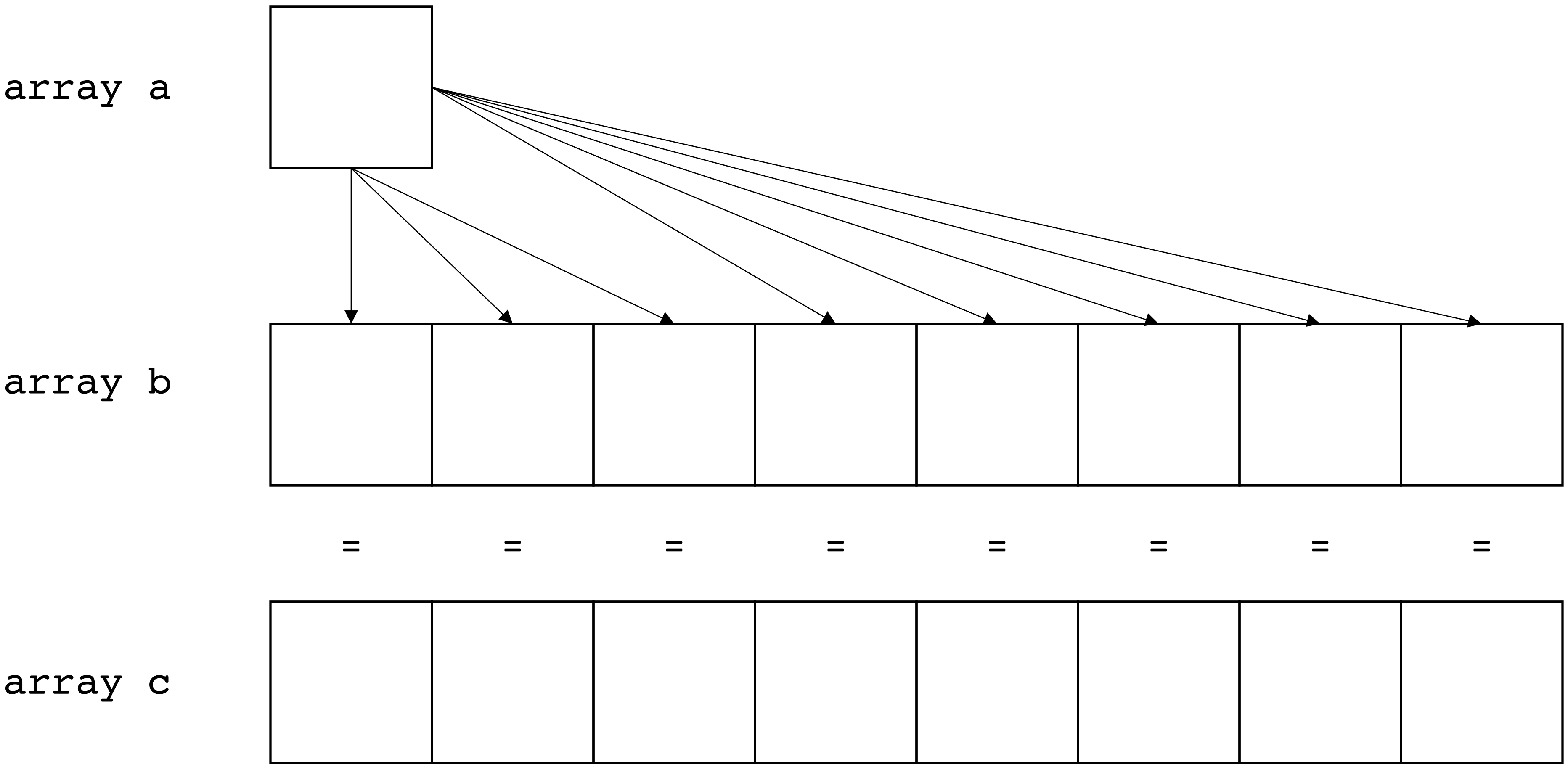
- Next Program

There are 3 arrays: `a, b, c.`

We want to compute `c[i] = a[0] + b[i]`

Embarrassingly parallel

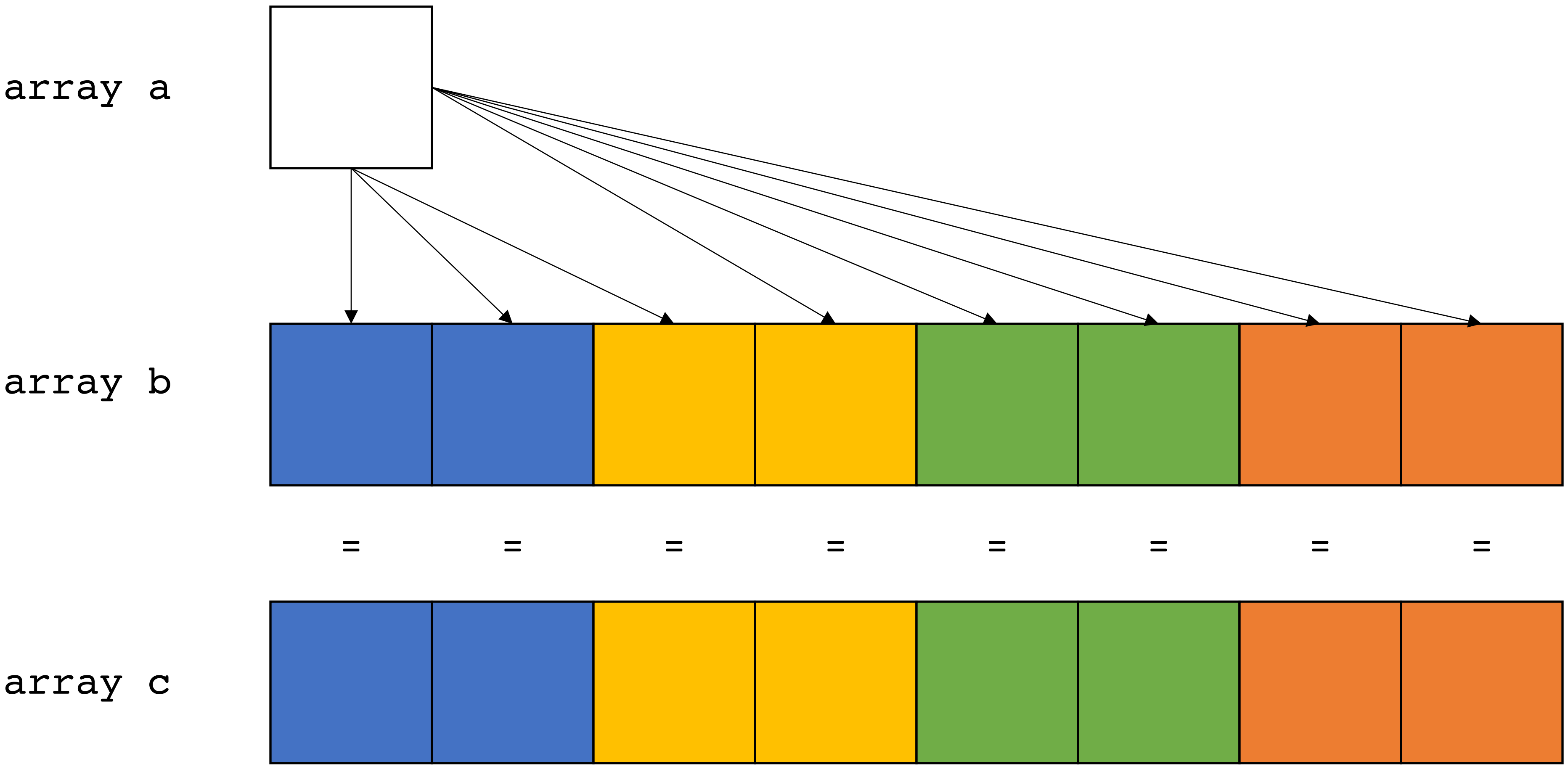
Thread 0 - Blue
Thread 1 - Yellow
Thread 2 - Green
Thread 3 - Orange



is this problem embarrassingly parallel?

Embarrassingly parallel

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

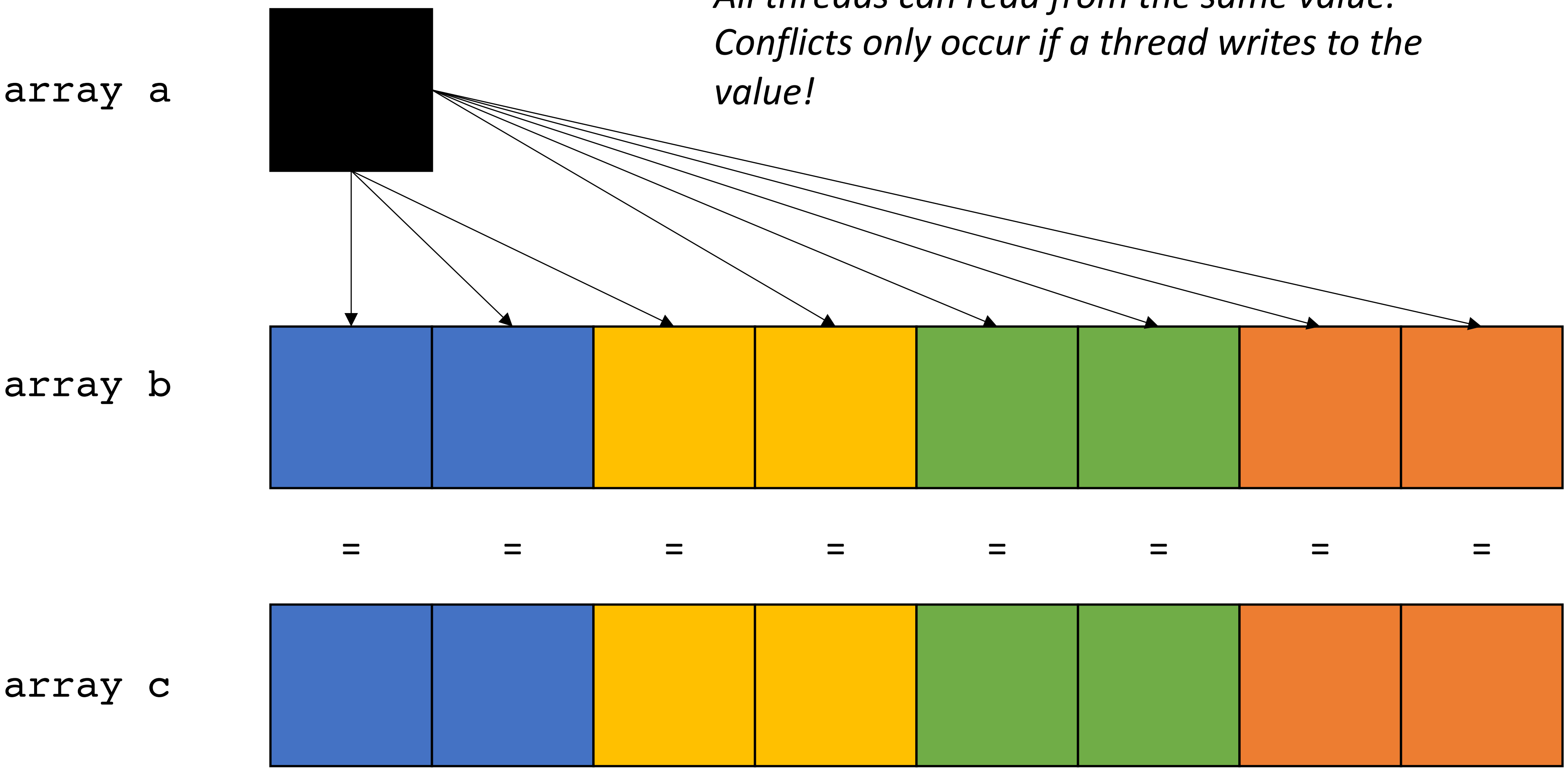


is this problem embarrassingly parallel?

Embarrassingly parallel

All threads can read from the same value. Conflicts only occur if a thread writes to the value!

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



is this problem embarrassingly parallel?

Embarrassingly parallel

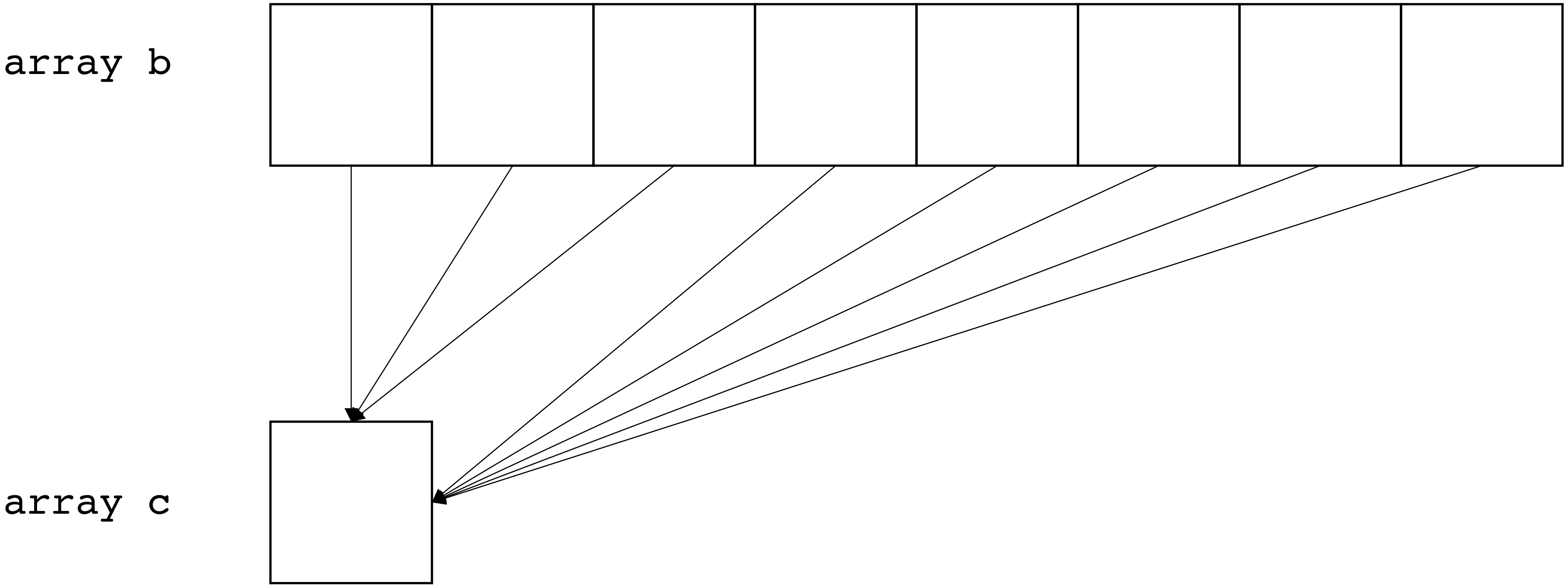
- Next Program

There are 2 arrays: `b, c`

We want to compute `c[0] = b[0] + b[1] + b[2] ...`

Embarrassingly parallel

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

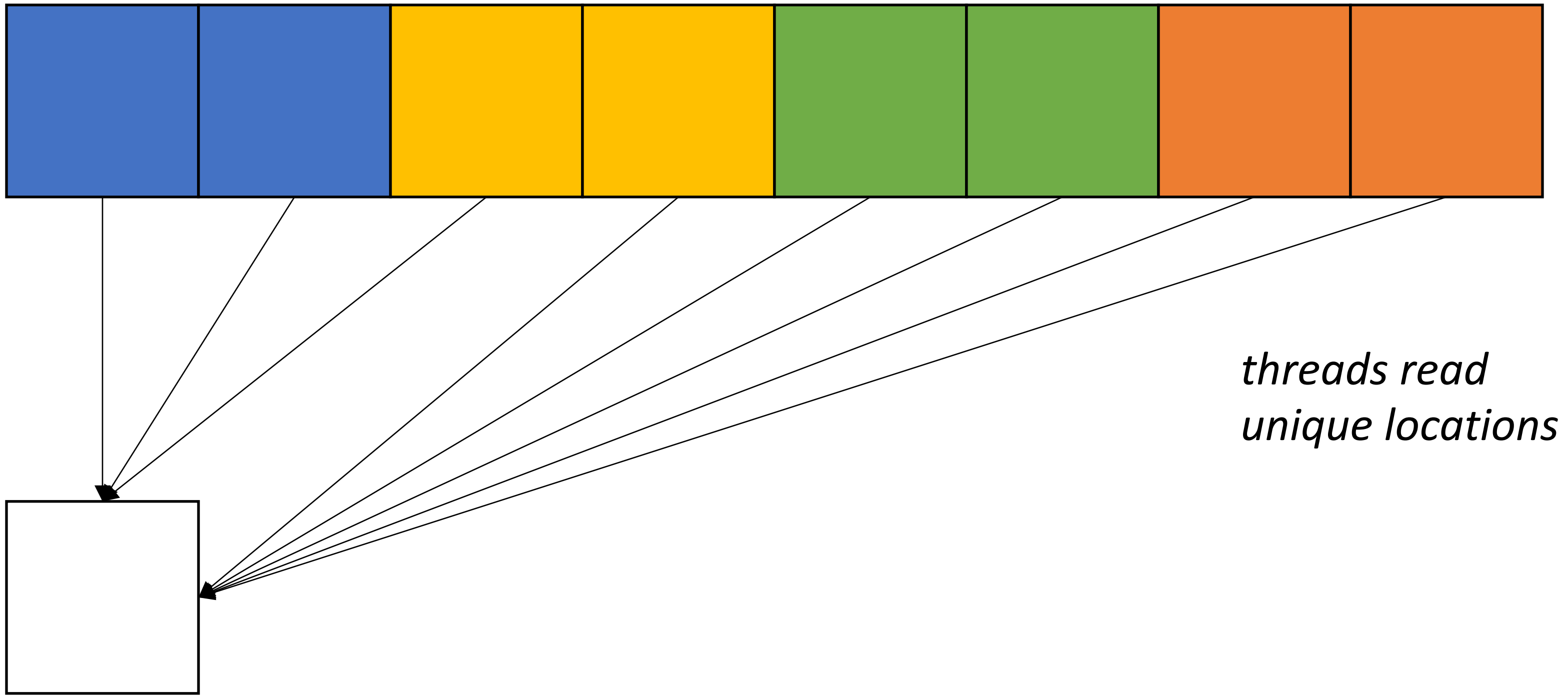


is this problem embarrassingly parallel?

Embarrassingly parallel

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

array b



array c

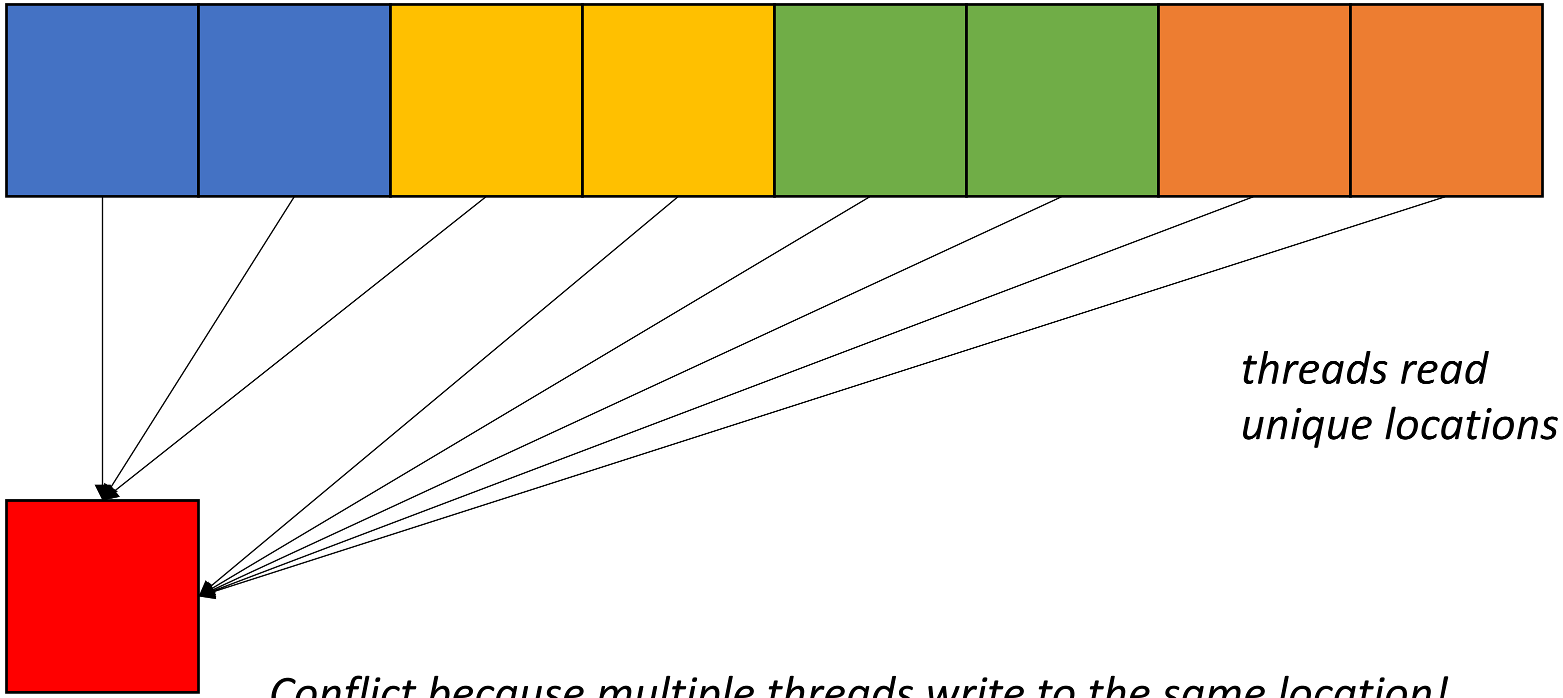
threads read unique locations

is this problem embarrassingly parallel?

Embarrassingly parallel

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

array b



threads read unique locations

is this problem embarrassingly parallel?

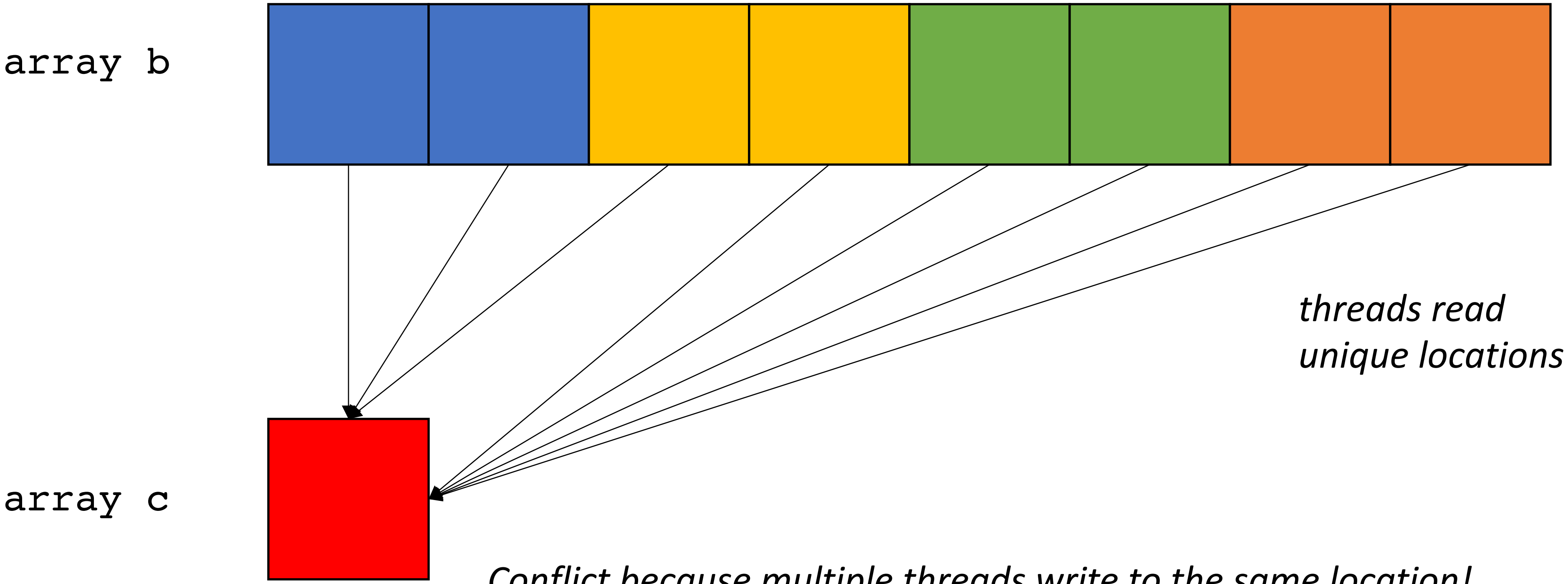
array c

Conflict because multiple threads write to the same location!

Embarrassingly parallel

Note: Reductions have some parallelism in them, as seen in your homework.

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



is this problem embarrassingly parallel?

Conflict because multiple threads write to the same location!

We need a way how to safely share memory

- *Most applications are not embarrassingly parallel*

We need a way how to safely share memory

- Bank

My account: $$

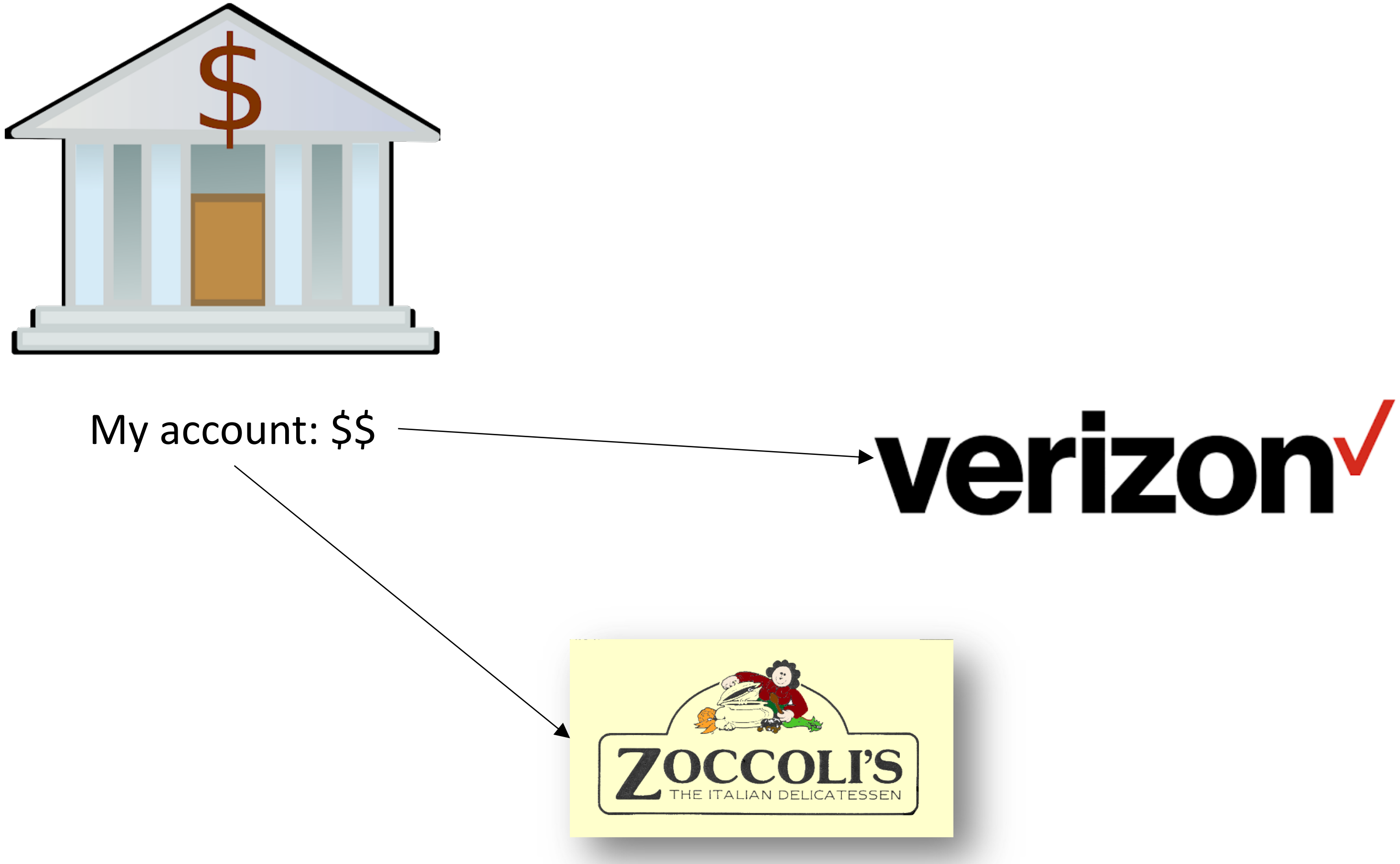
We need a way how to safely share memory

- Bank

My account: $$

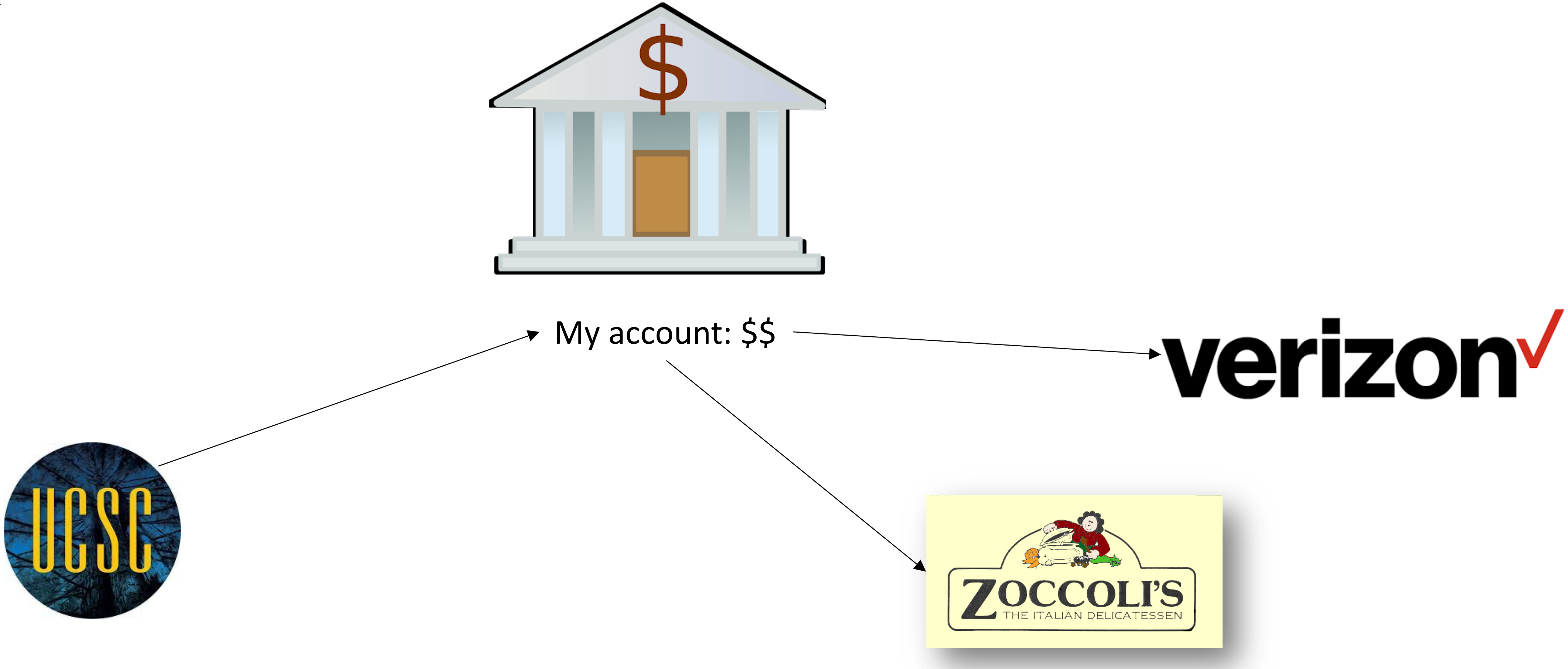
We need a way how to safely share memory

- Bank



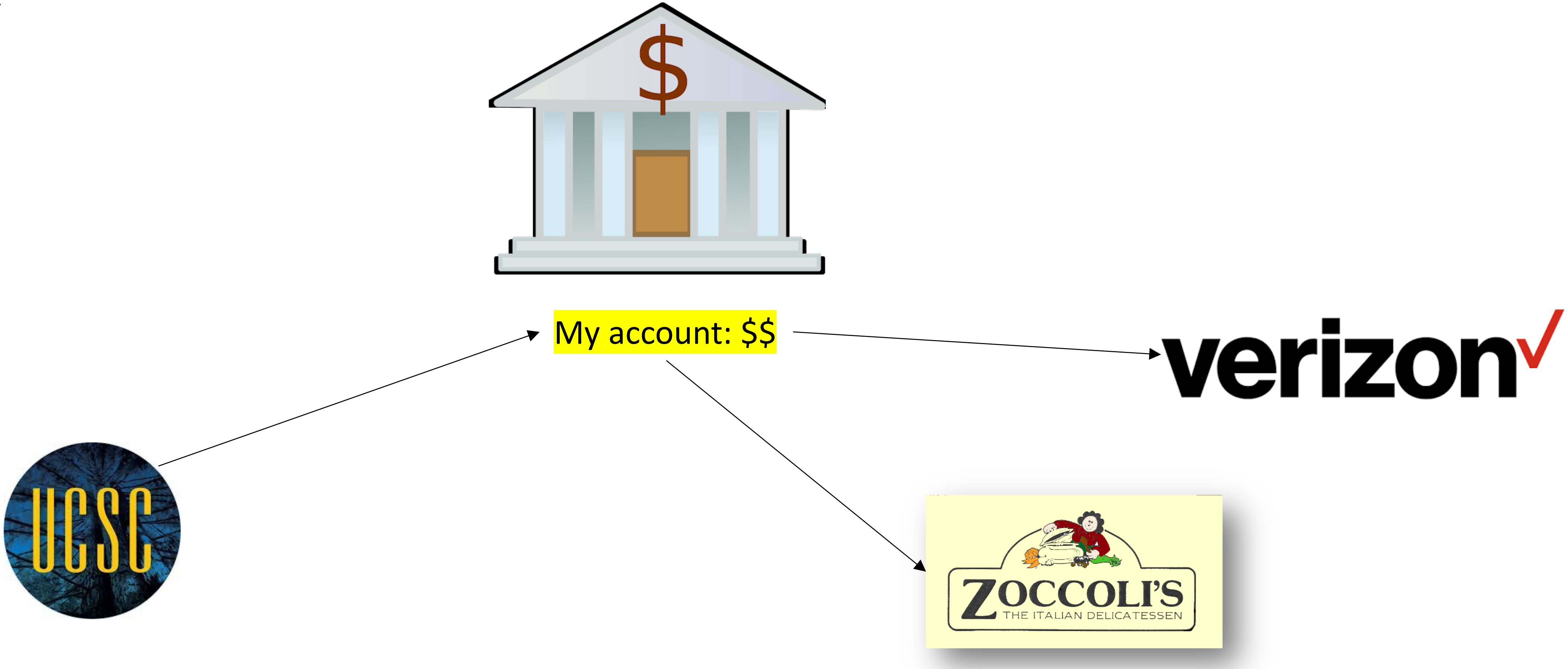
We need a way how to safely share memory

- Bank



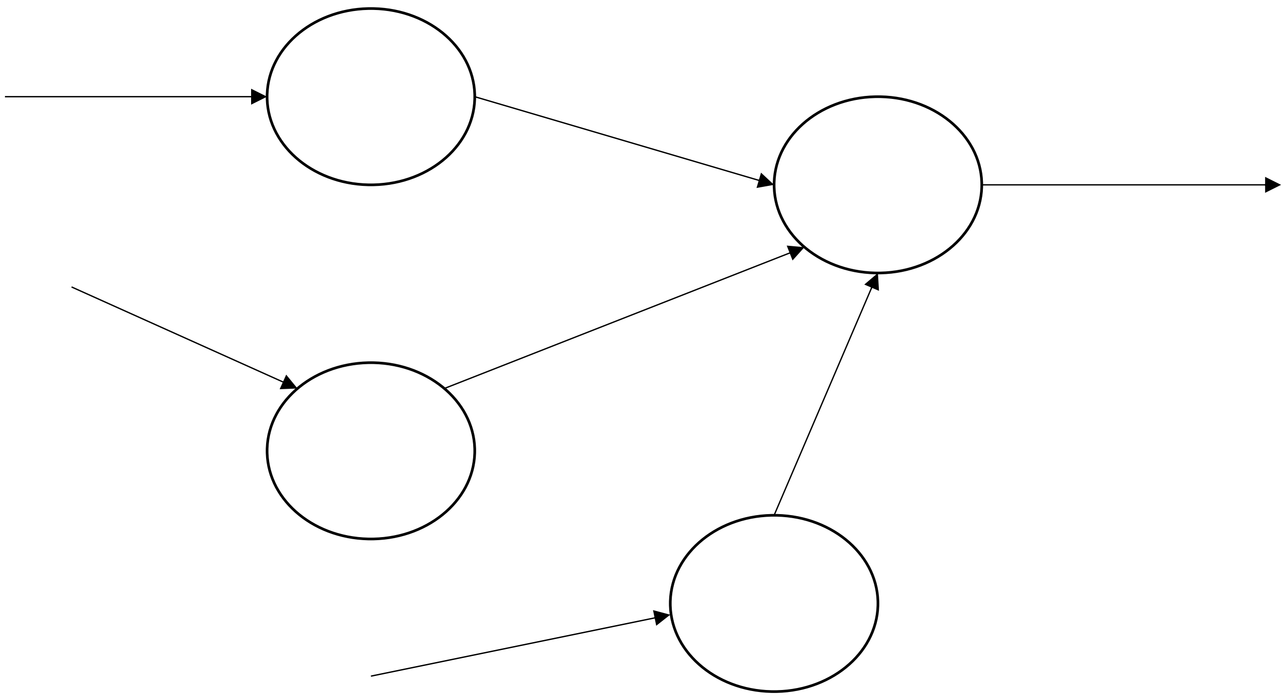
We need a way how to safely share memory

- Bank



We need a way how to safely share memory

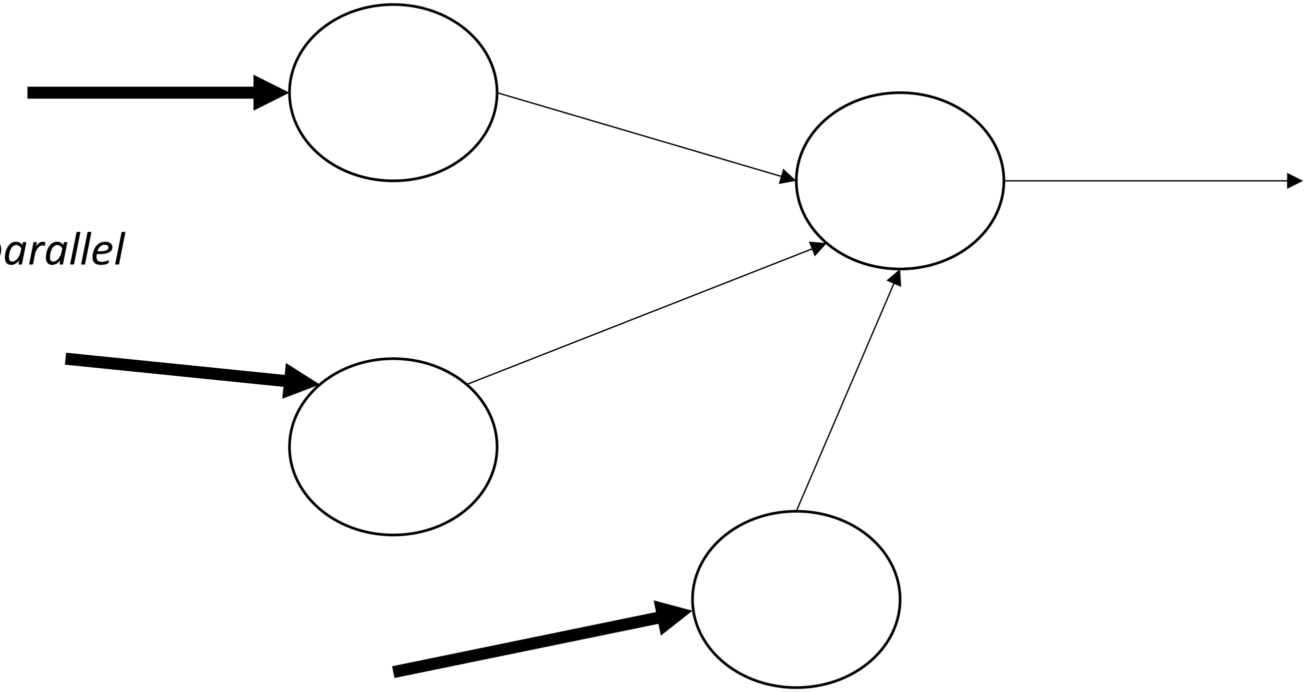
- Graph algorithms



Examples:
Ranking pages on the internet
information spread in social media

We need a way how to safely share memory

- Graph algorithms

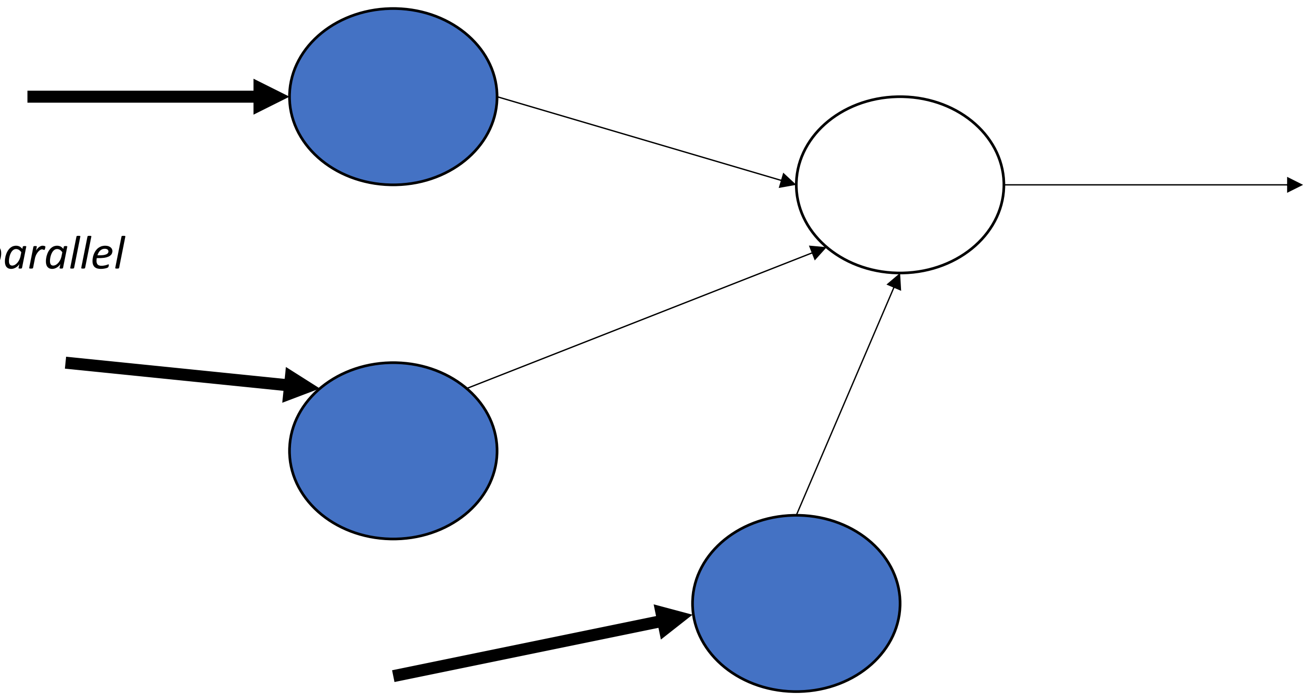


these can be done in parallel

Examples:
Ranking pages on the internet
information spread in social media

We need a way how to safely share memory

- Graph algorithms



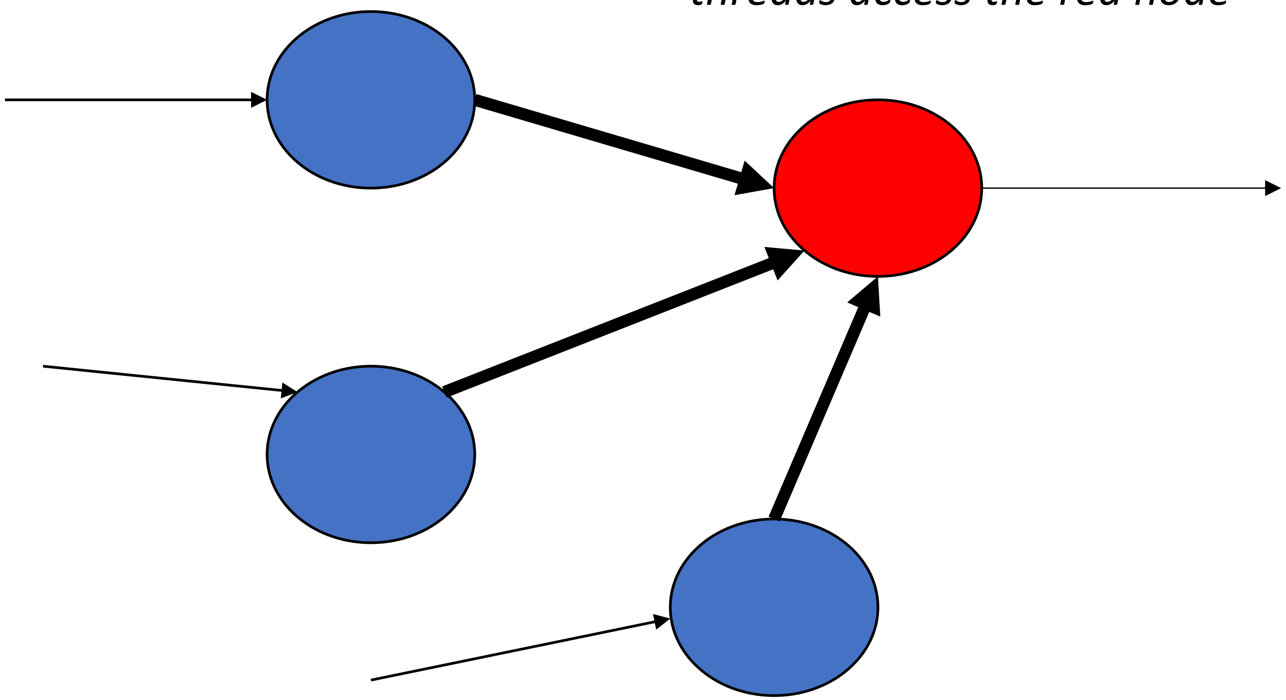
these can be done in parallel

Examples:
Ranking pages on the internet
information spread in social media

We need a way how to safely share memory

- Graph algorithms

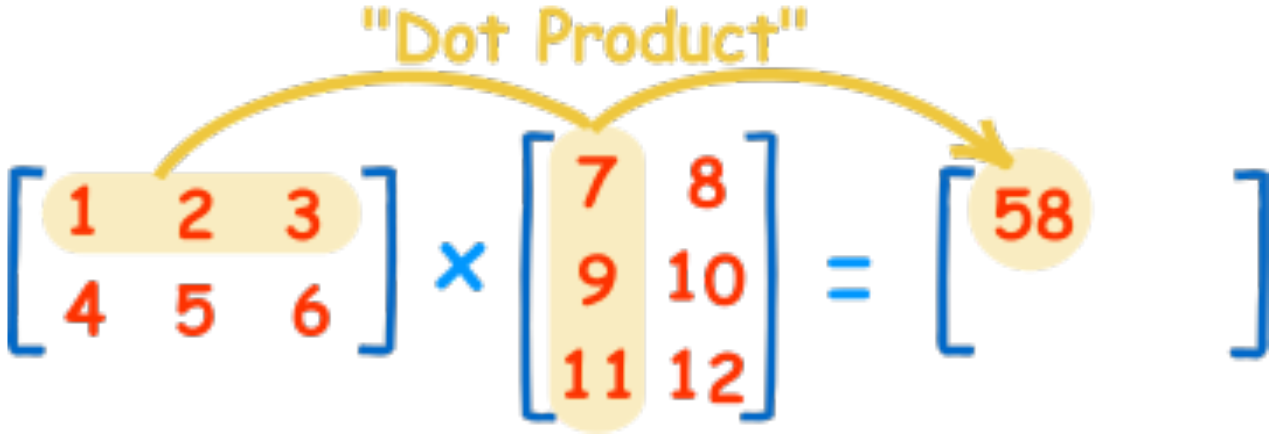
potential conflict if different threads access the red node



Examples:
Ranking pages on the internet
information spread in social media

We need a way how to safely share memory

- Machine Learning



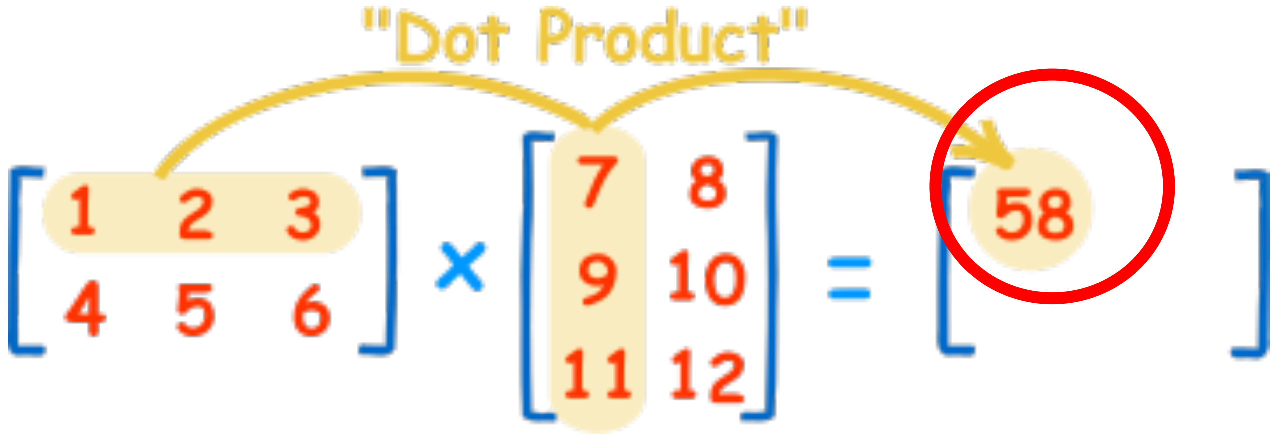
Lots of machine learning is some form of matrix multiplication

image from: https://www.mathsisfun.com/

We need a way how to safely share memory

- Machine Learning

conflict!

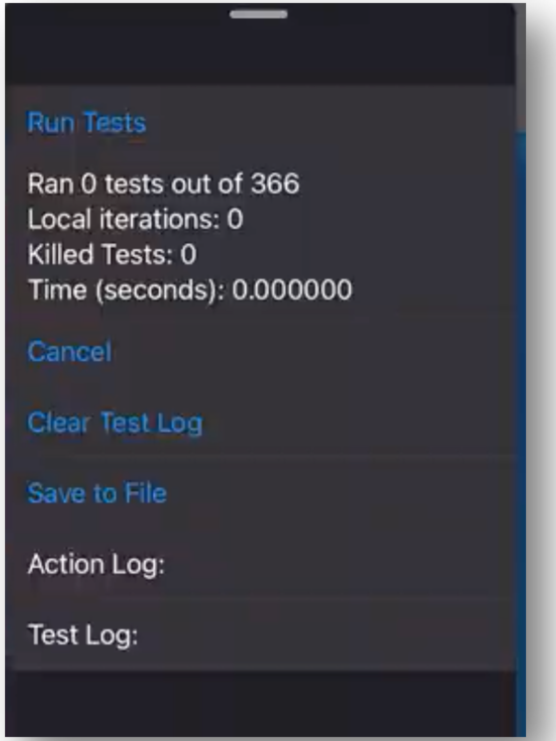


Lots of machine learning is some form of matrix multiplication

image from: https://www.mathsisfun.com/

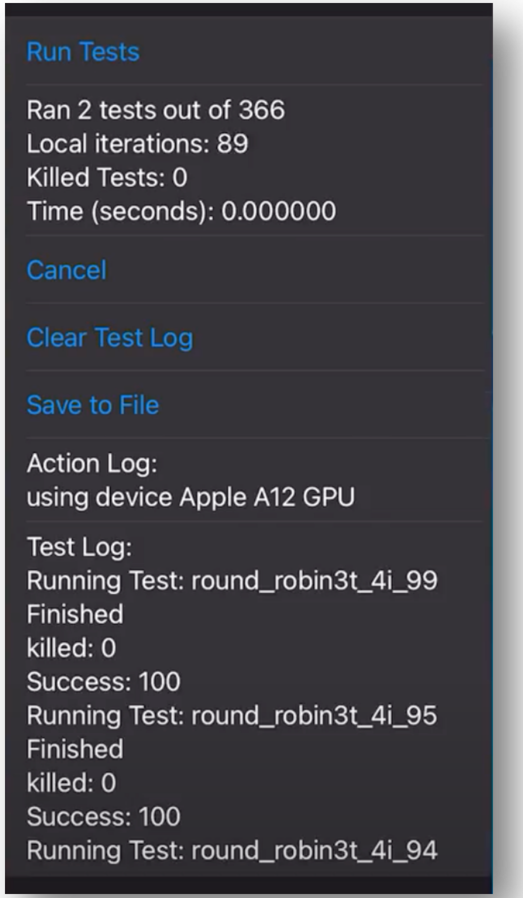
We need a way how to safely share memory

- User interfaces



background process that provides progress updates to the UI.

UI updates must be synchronized!!



Dangers of conflicts

- We will illustrate using a running bank account example

Sequential bank scenario

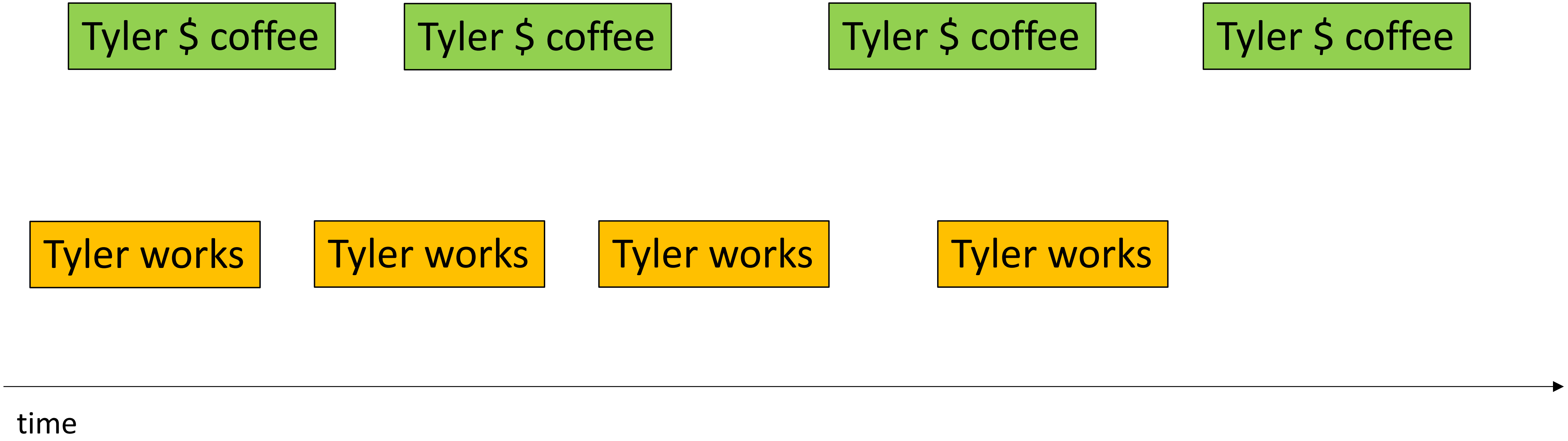
- UCSC deposits $1 in my bank account after every hour I work.
- I buy a cup of coffee ($1) after each hour I work.
- I work 1M hours (which is actually true).
- *I should break even*
- **C++ code**

Concurrent bank scenario

- UCSC contracts me to work 1M hours.
- My bank is so impressed with my contract that they give me a line of credit. i.e. I can overdraw as long as I pay it back.
- UCSC deposits $1 in my bank account **after** every hour I work.
- I budget $1M to spend on coffee **during** work.
- C++ code

Concurrent bank scenario

This sets up a scheme where I buy coffee concurrently with working



Reasoning about concurrency

- What is going on?
- We need to be able to reason more rigorously about concurrent programs

A thread is a sequential program

Tyler's coffee addiction:

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

Tyler's employer

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

A thread is a sequential program

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

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

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

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

The execution of a program gives rise to events

Important distinction between program and events

A thread is a sequential program

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;
}
```

A thread is a sequential program

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
```

A thread is a sequential program

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

color code events.
coffee thread is blue
payment thread is yellow

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

A thread is a sequential program

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
```

Any interleaving of the events is a valid execution of the concurrent program!

Tyler's employer

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

time

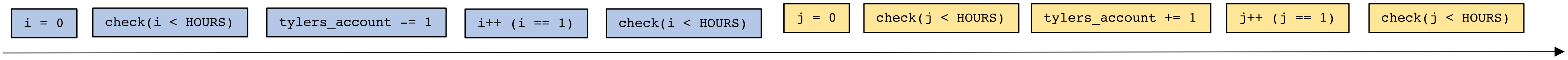
```
j = 0
check(j < HOURS)
tylers_account += 1
j++ (j == 1)
check(j < HOURS)
tylers_account += 1
j++ (j == 2)
check(j < HOURS)
tylers_account += 1
```

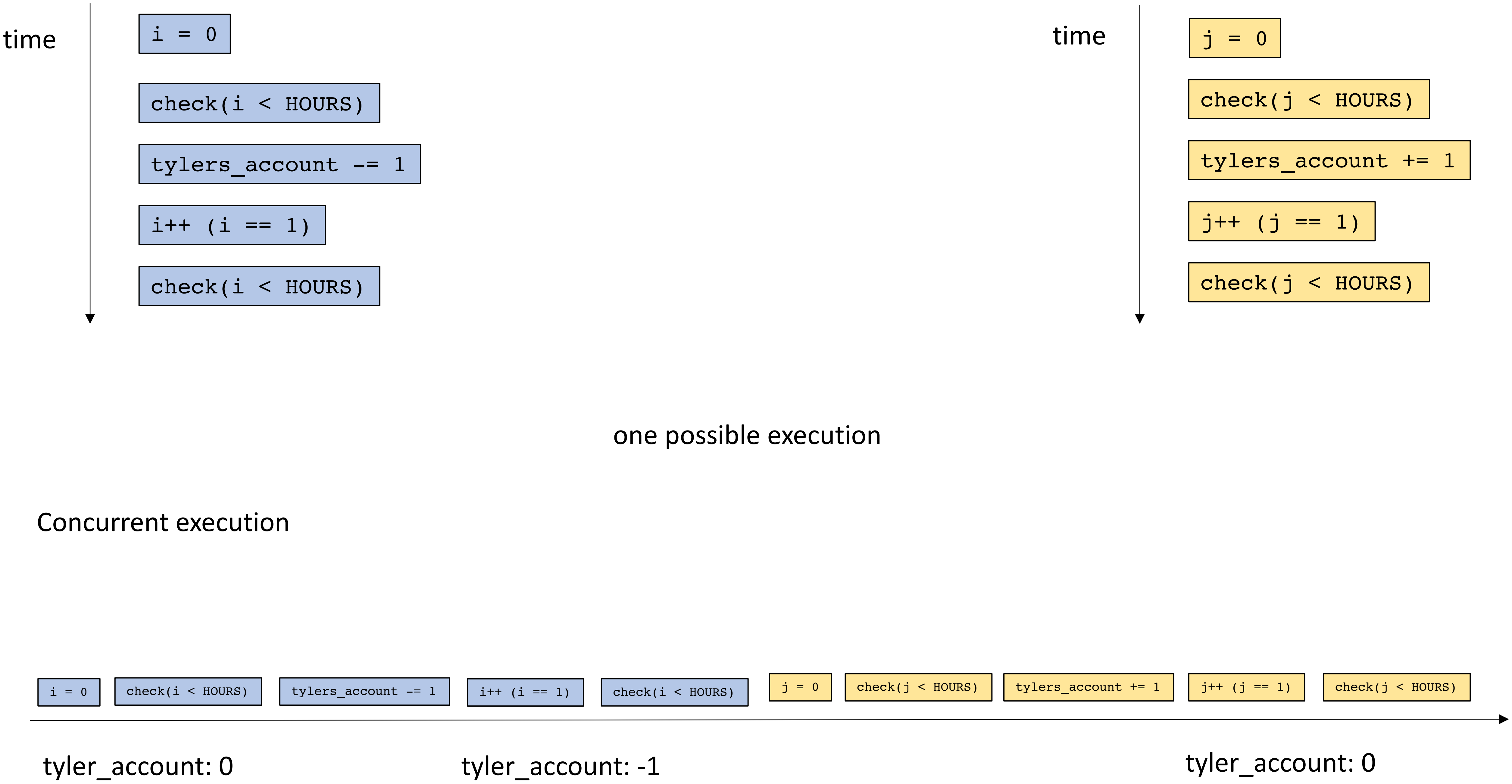
consider just one loop iteration

Concurrent execution

one possible execution

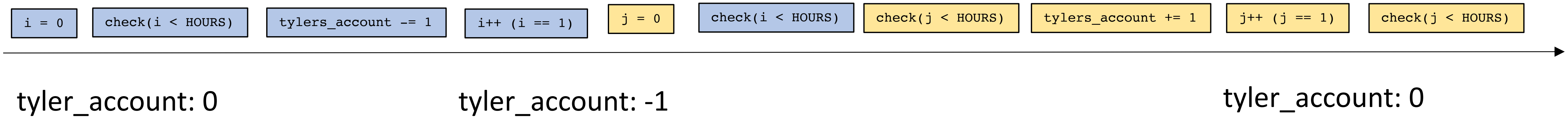
Concurrent execution




time
i = 0
check(i < HOURS)
tylers_account -= 1
i++ (i == 1)
check(i < HOURS)
time
j = 0
check(j < HOURS)
tylers_account += 1
j++ (j == 1)
check(j < HOURS)
one possible execution
Concurrent execution
i = 0
check(i < HOURS)
tylers_account -= 1
i++ (i == 1)
check(i < HOURS)
j = 0
check(j < HOURS)
tylers_account += 1
j++ (j == 1)
check(j < HOURS)
tyler_account: 0
tyler_account: -1
tyler_account: 0

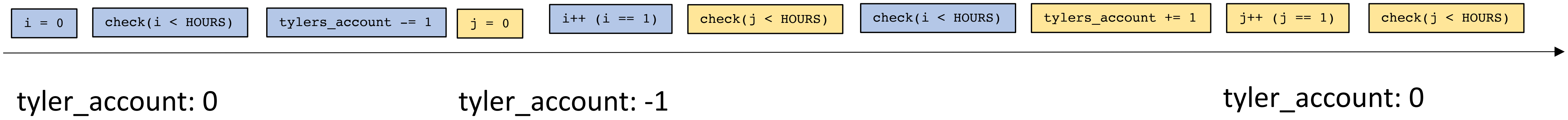
Another possible execution

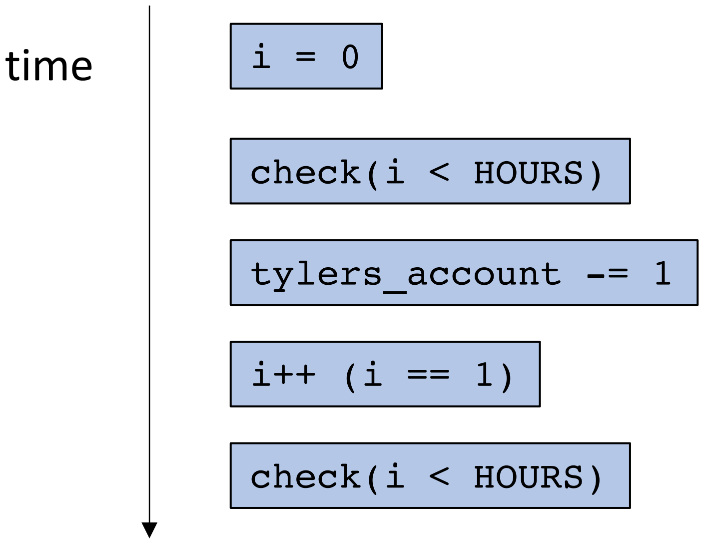
Concurrent execution

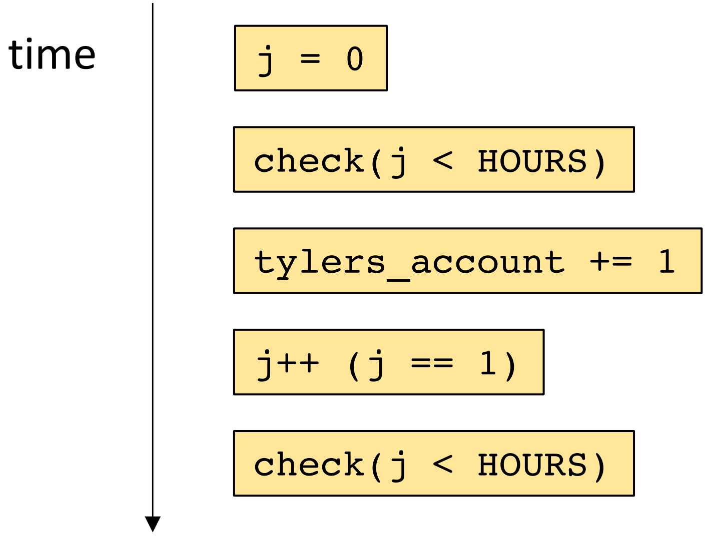


Another possible execution

Concurrent execution

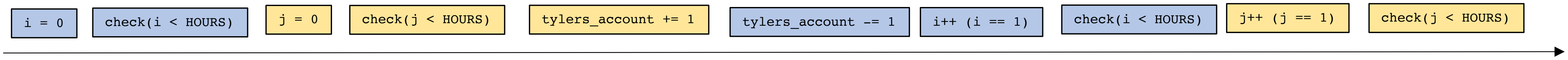


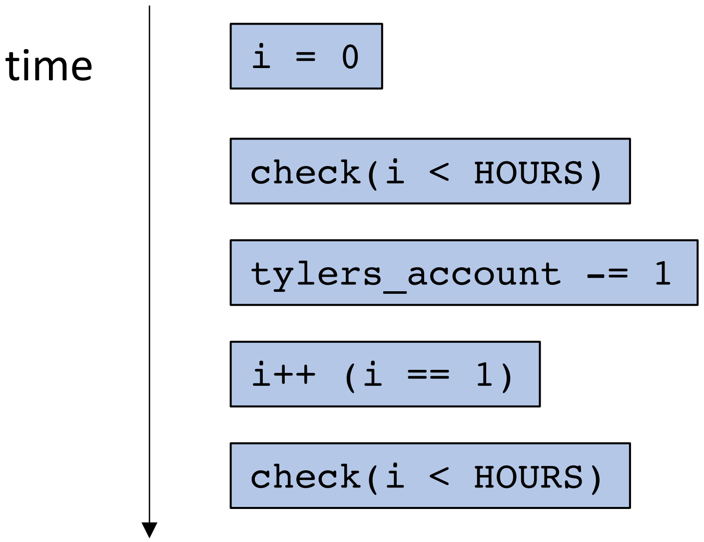


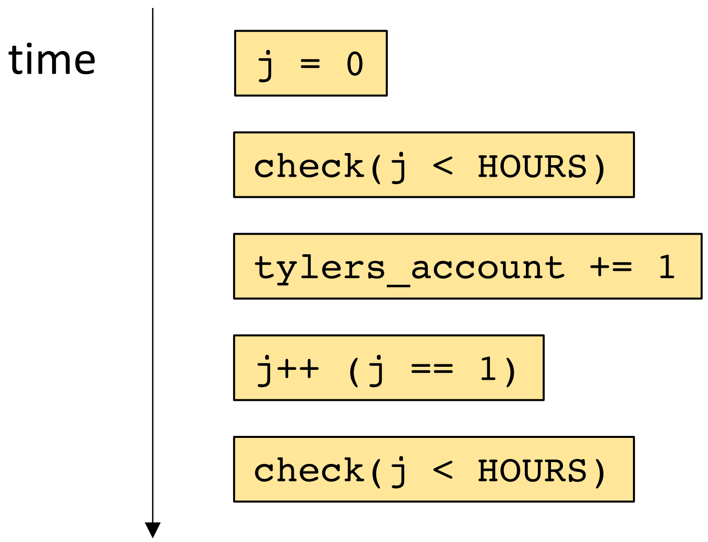


Another possible execution

Concurrent execution



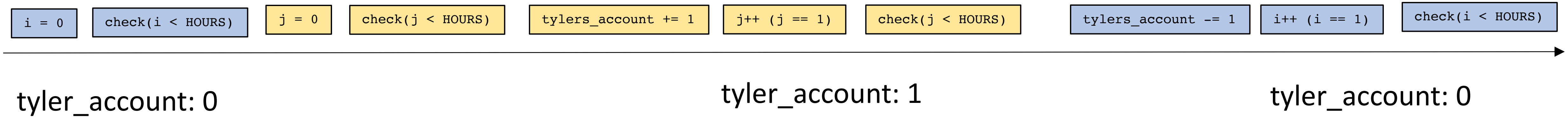


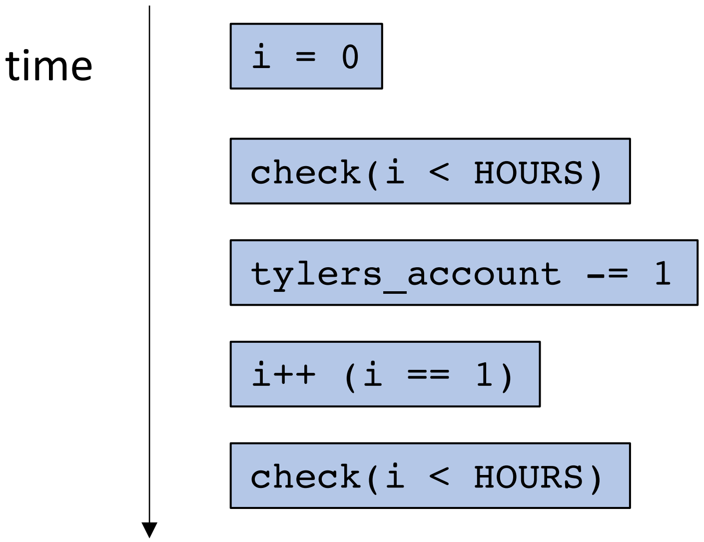


Another possible execution

This time my account isn’t ever negative

Concurrent execution

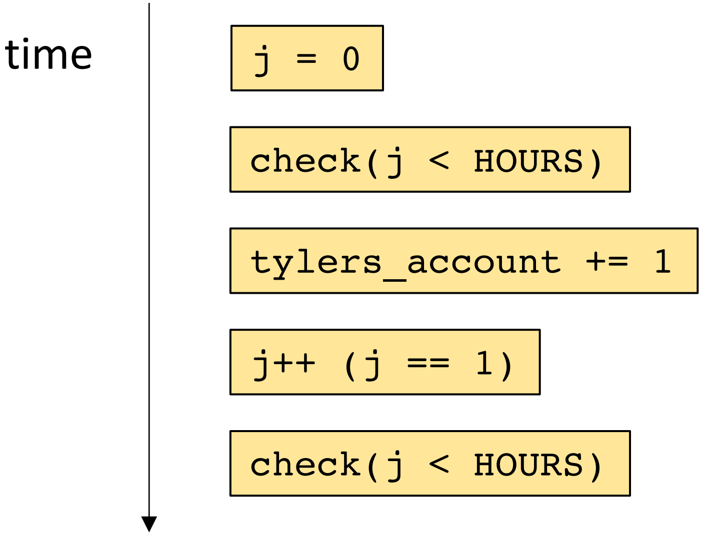




How many possible interleavings?
Combinatorics question:

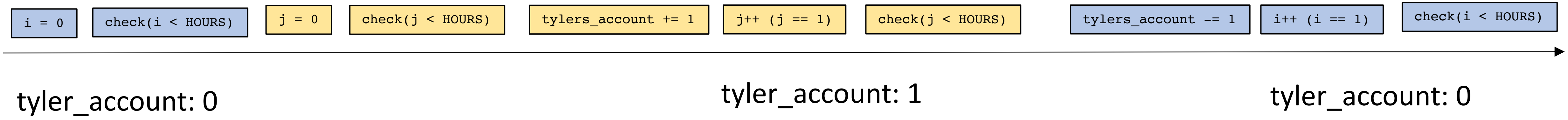
if Thread 0 has N events
if Thread 1 has M events

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



Concurrent execution

in our example there are 252 possible interleavings!

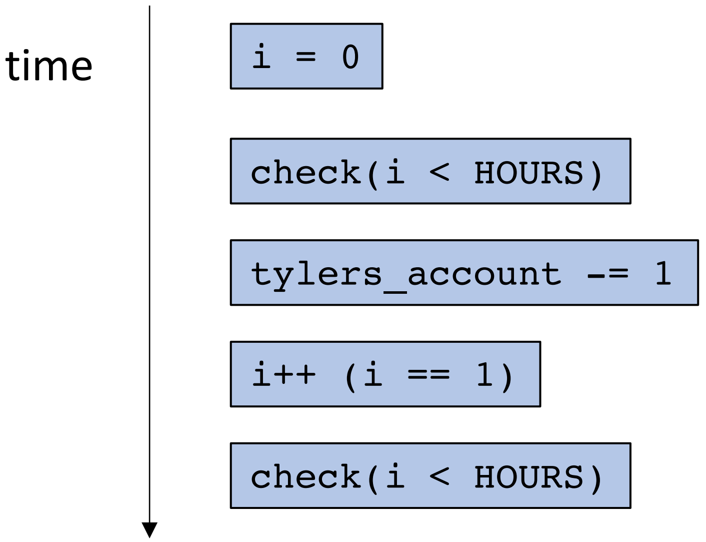


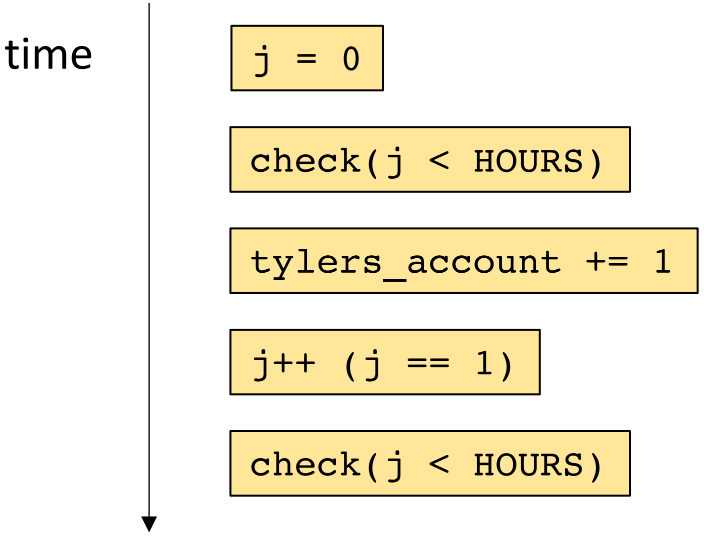
Reasoning about concurrency

- Not feasible to think about all interleavings!
  - Lots of interesting research in pruning, testing interleavings (Professor Flanigan)
  - Very difficult to debug

- Think about smaller instances of the problem, reason about the problem as a whole.
  - Tyler spends a total of $1M on coffee
  - Tyler gets paid a total of $1M
  - The balance should be 0!

- **Reduce the problem**: *If there's a problem we should be able to see it in a single loop iteration.*

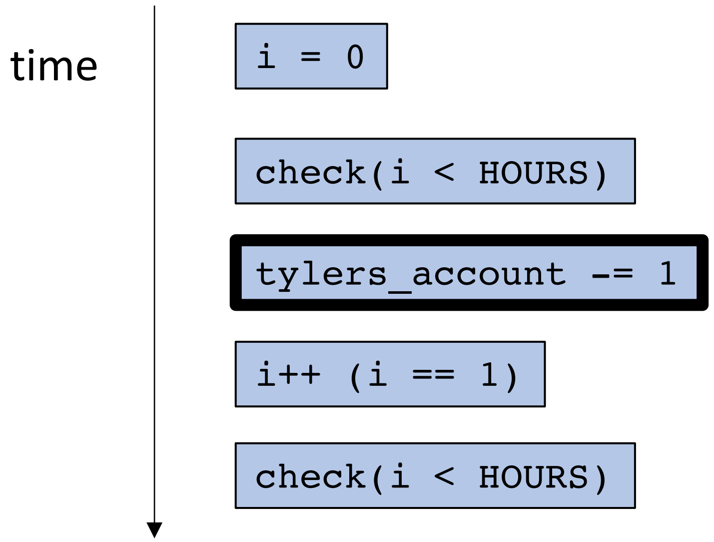


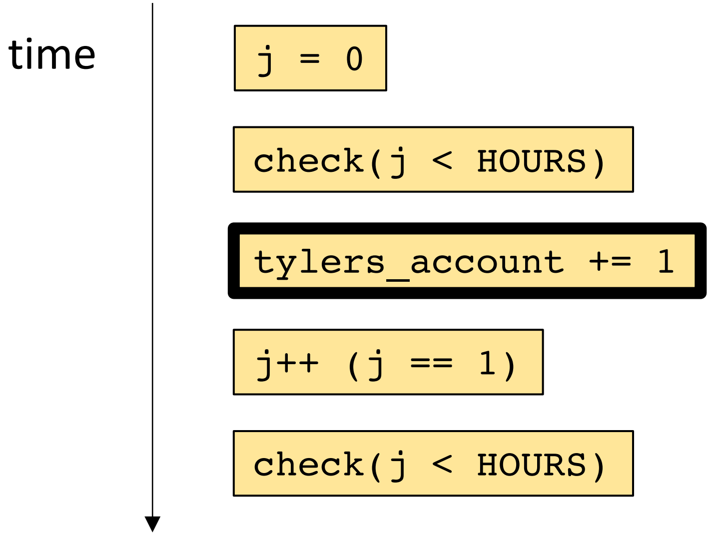


Lets get to the bottom of our money troubles:

For any interleaving, both of the increase and decrease must happen in some order.

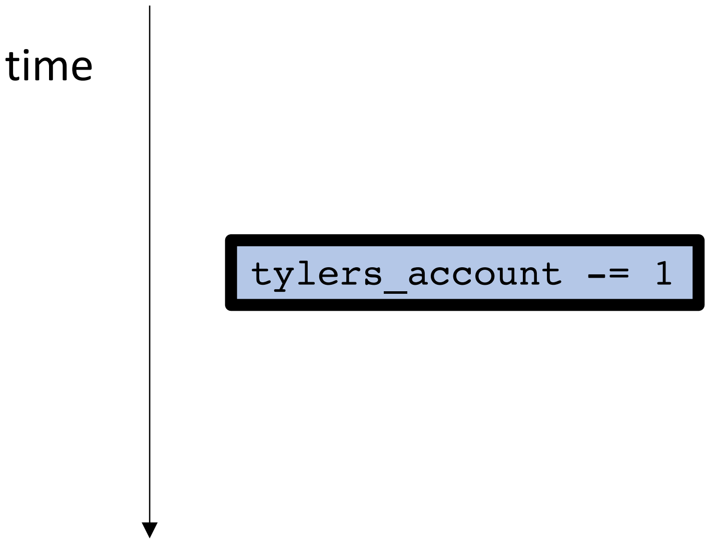
So there isn’t an interleaving that will explain the issue.

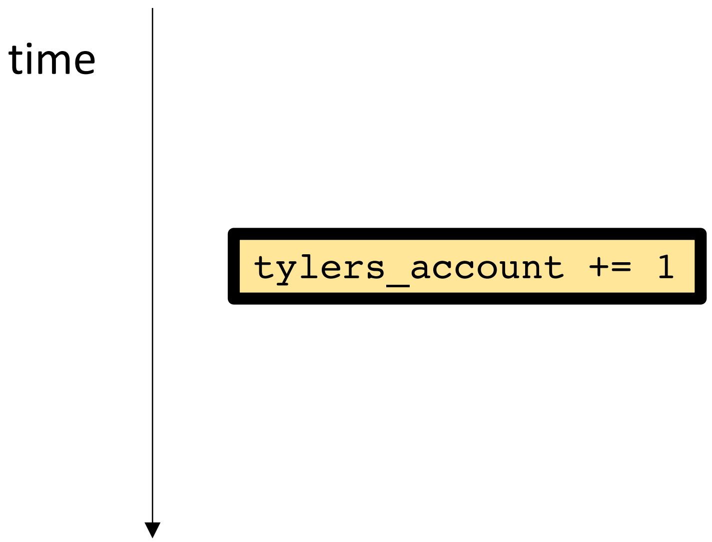




concurrent execution

time





Remember 3 address code...

concurrent execution

time

time

```
T0_load = *tylers_account
T0_load -= 1
*tylers_account = T0_load
```

```
tylers_account -= 1
```

time

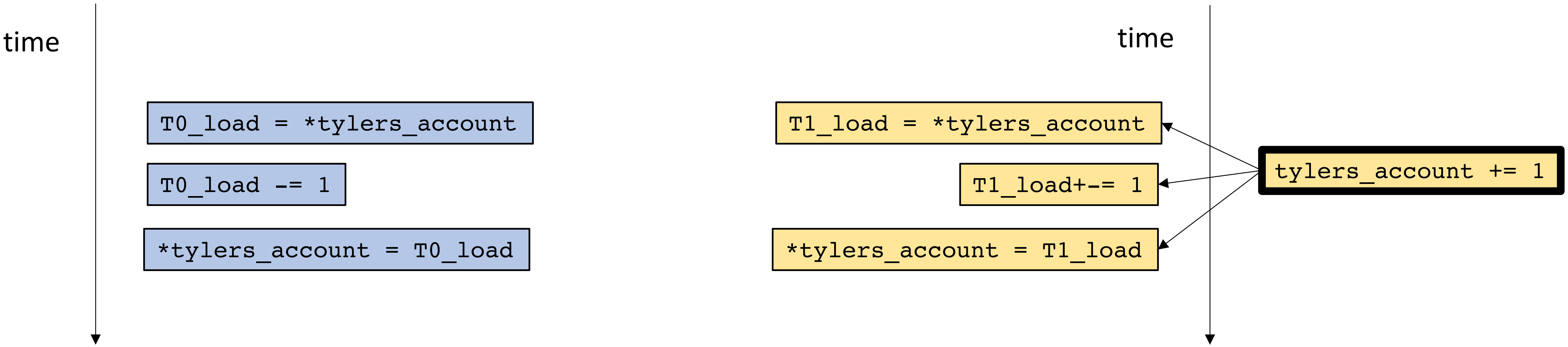
```
tylers_account += 1
```

this line of code needs to be expanded

Remember 3 address code...

concurrent execution

time



Remember 3 address code...

concurrent execution

time

time

```
T0_load = *tylers_account
T0_load -= 1
*tylers_account = T0_load
```

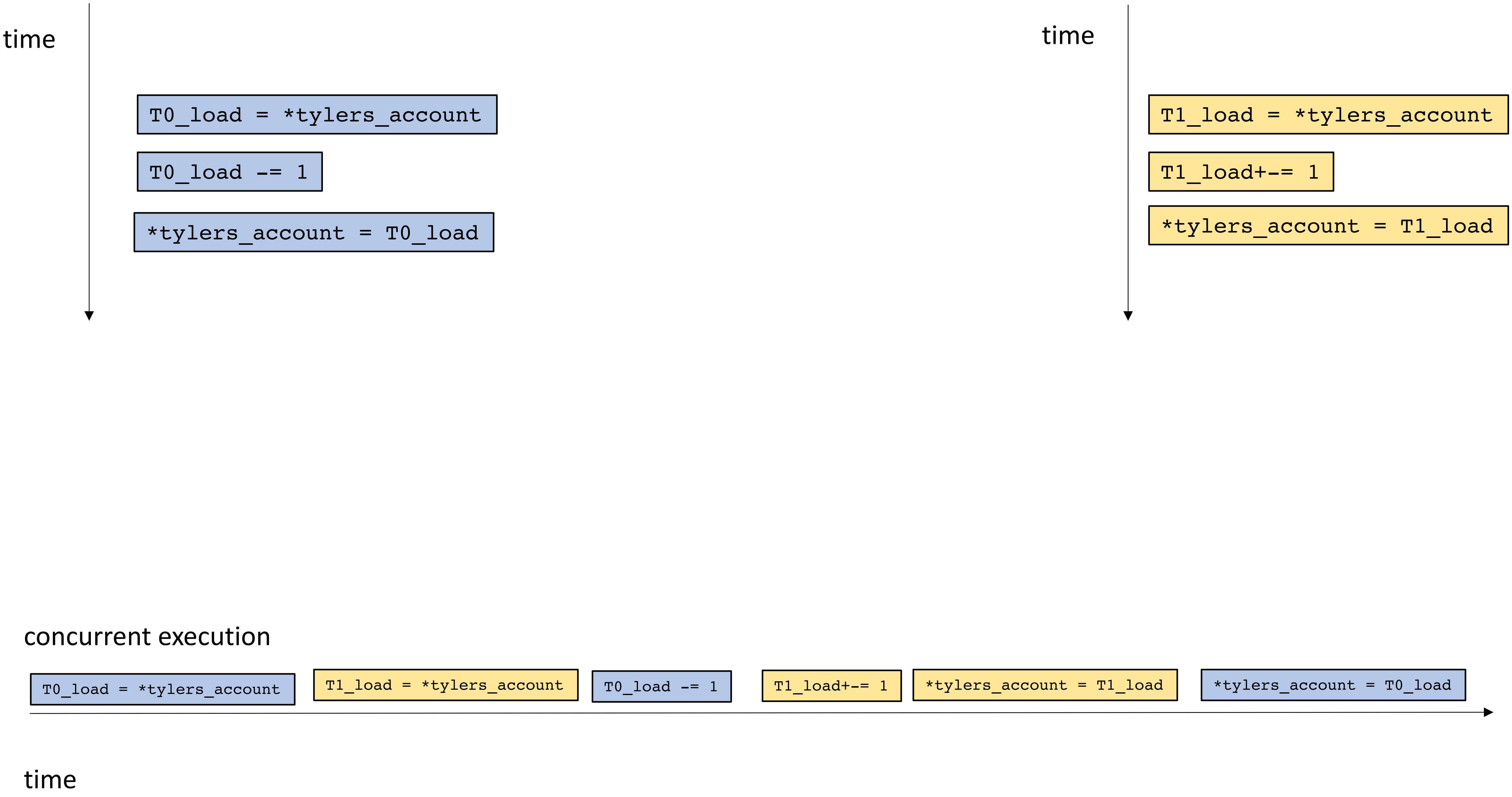
time

```
T1_load = *tylers_account
T1_load+-= 1
*tylers_account = T1_load
```

What if we interleave these instructions?

concurrent execution

time


time
T0_load = *tylers_account
T0_load -= 1
*tylers_account = T0_load
time
T1_load = *tylers_account
T1_load+-= 1
*tylers_account = T1_load
concurrent execution
T0_load = *tylers_account
T1_load = *tylers_account
T0_load -= 1
T1_load+-= 1
*tylers_account = T1_load
*tylers_account = T0_load
time

time

```
T0_load = *tylers_account
T0_load -= 1
*tylers_account = T0_load
```

time

```
T1_load = *tylers_account
T1_load+-= 1
*tylers_account = T1_load
```

tylers_account has -1 at the end of this interleaving!

concurrent execution

```
T0_load = *tylers_account
T1_load = *tylers_account
T0_load -= 1
T1_load+-= 1
*tylers_account = T1_load
*tylers_account = T0_load
```

time

What now?

- Data conflicts lead to many different types of issues, not just strange interleavings.
  - Data tearing
  - Instruction reorderings
  - Compiler optimizations

- Rather than reasoning about data conflicts, we will protect against them using ***synchronization***.

Synchronization

- A scheme where several actors agree on how to safely share a resource during concurrent access.

- Must define what “safely” means.

- Example:
  - Two neighbors sharing a yard between a dog and cat
  - Sharing refrigerator with roommates
  - An account balance that is written to and read from
  - Chapter 1 in text book

Lecture Schedule

- Canvas Quiz
- Notes on homework
- Reasoning about concurrency
- **Mutual exclusion**
- Multiple mutexes

Mutexes

- A Synchronization object to protect against data conflicts

Simple API:

```
lock()
unlock()
```

- Before a thread accesses the shared memory, it should call `lock()`
- When a thread is finished accessing the shared data, it should call `unlock()`

A thread is a sequential program

Tyler's coffee addiction:

```
tylers_account -= 1;
```

Tyler's employer

```
tylers_account += 1;
```

assume a global mutex object `m`

protect the account access with the mutex

A thread is a sequential program

Tyler's coffee addiction:

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

Tyler's employer

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

assume a global mutex object `m`

protect the account access with the mutex

A thread is a sequential program

Tyler's coffee addiction:

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

Tyler's employer

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

time

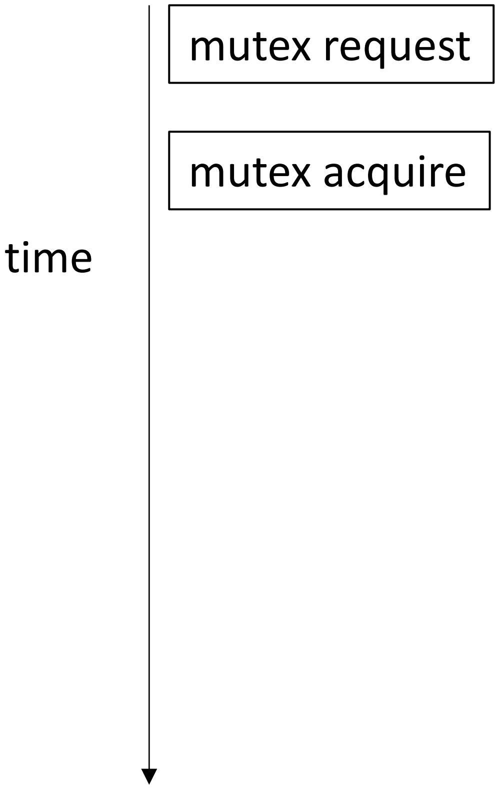
A thread is a sequential program

Tyler’s coffee addiction:

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

Tyler’s employer

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



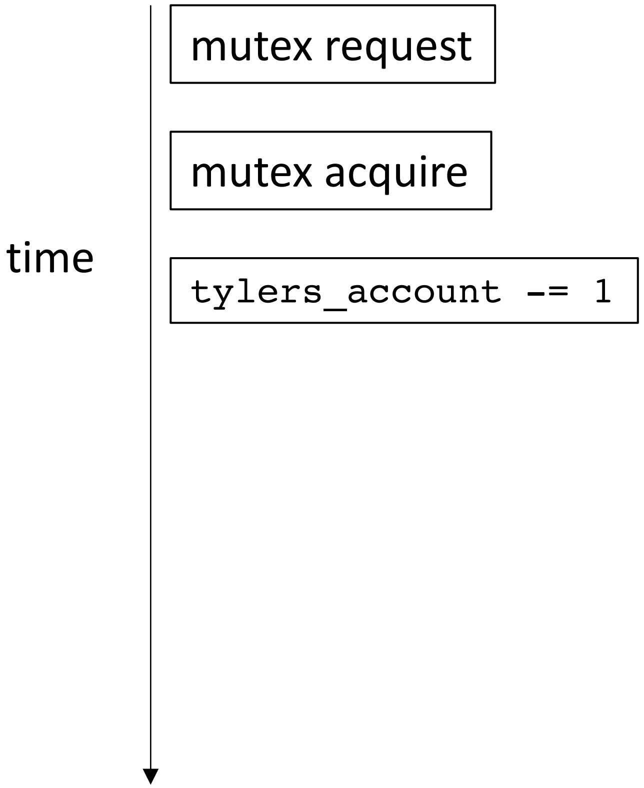
A thread is a sequential program

Tyler's coffee addiction:

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

Tyler's employer

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



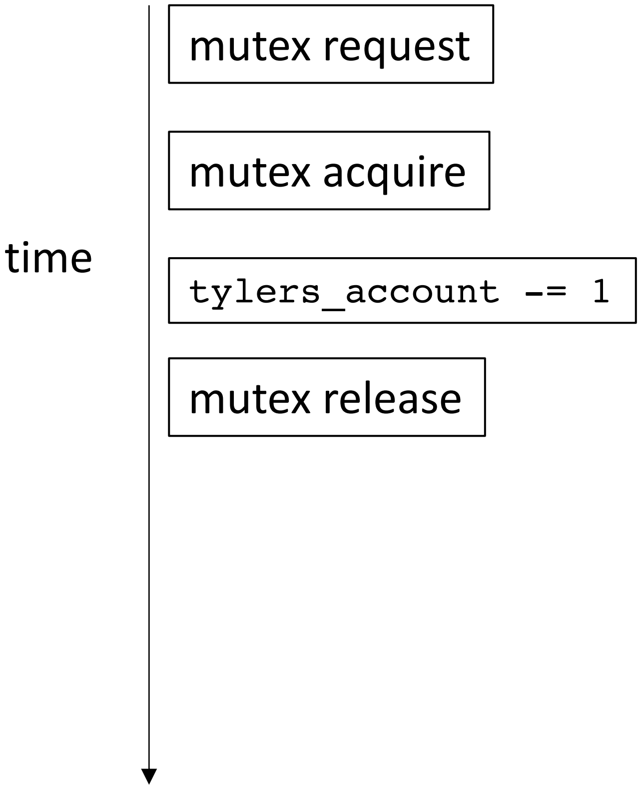
A thread is a sequential program

Tyler's coffee addiction:

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

Tyler's employer

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



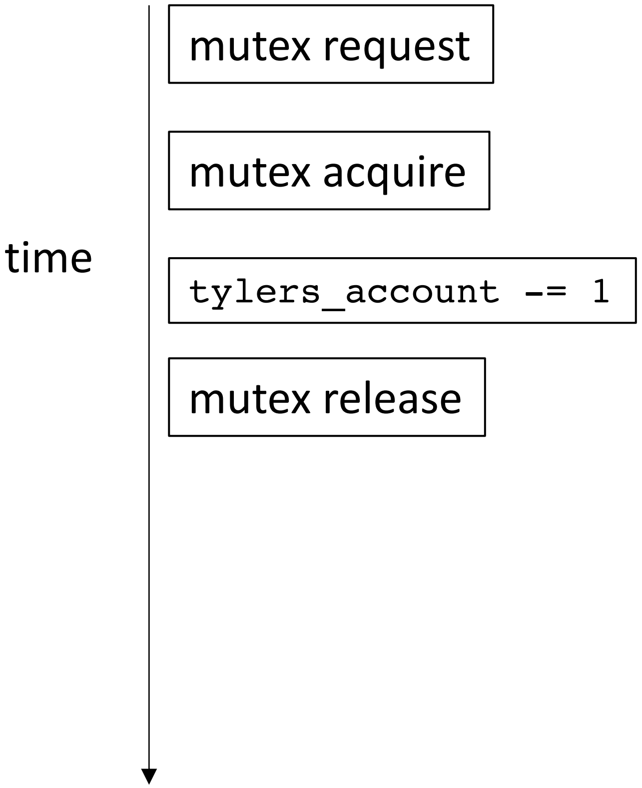
A thread is a sequential program

Tyler's coffee addiction:

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

Tyler's employer

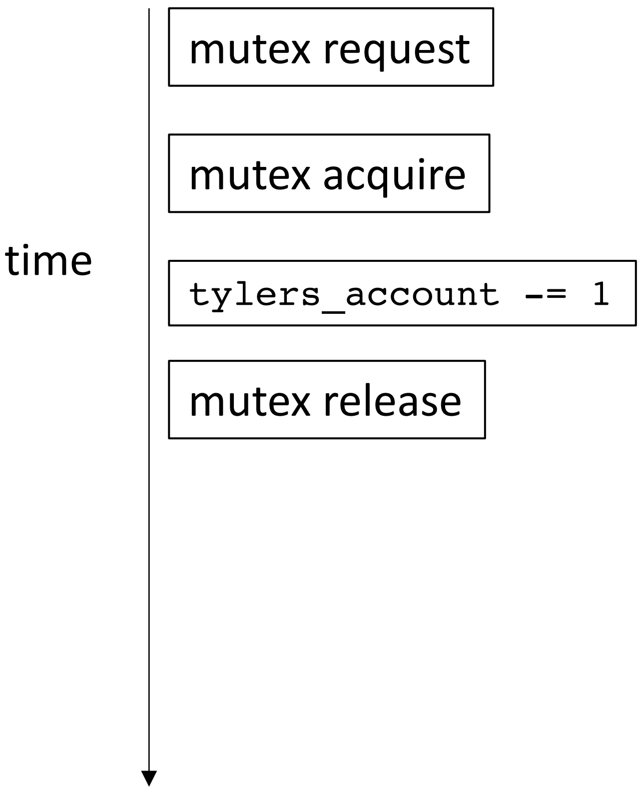
```
m.lock();
tylers_account += 1;
m.unlock();
```



A thread is a sequential program

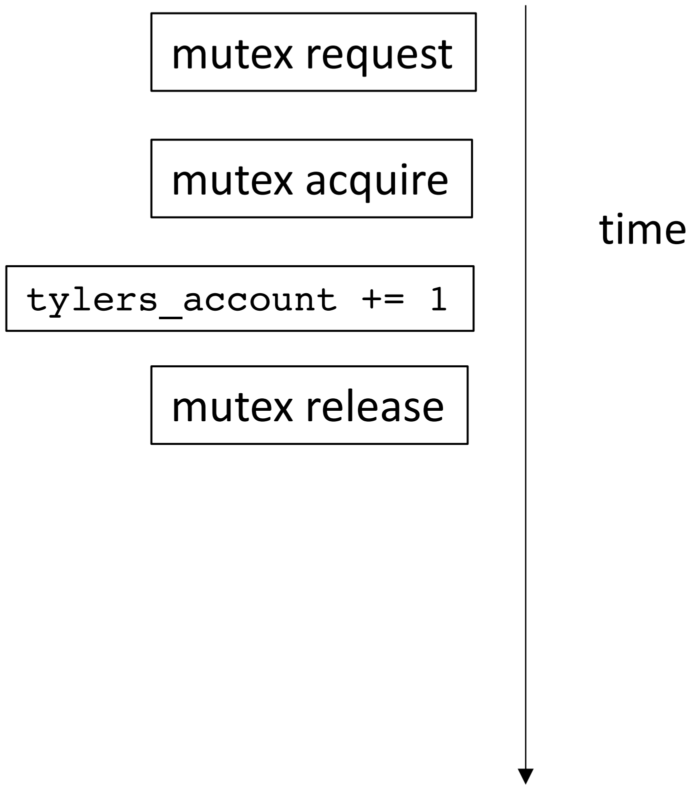
Tyler's coffee addiction:

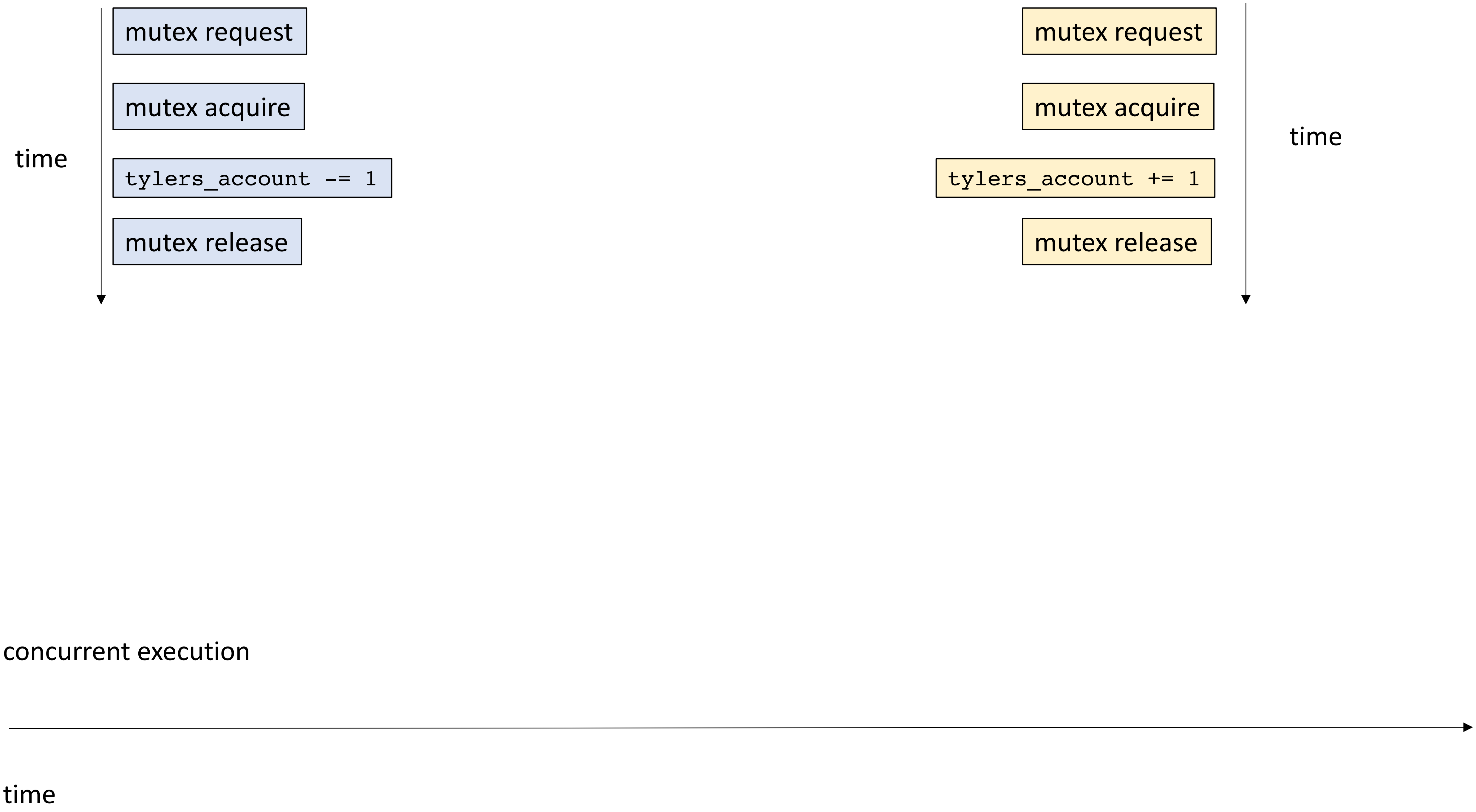
```
m.lock();
tylers_account -= 1;
m.unlock();
```



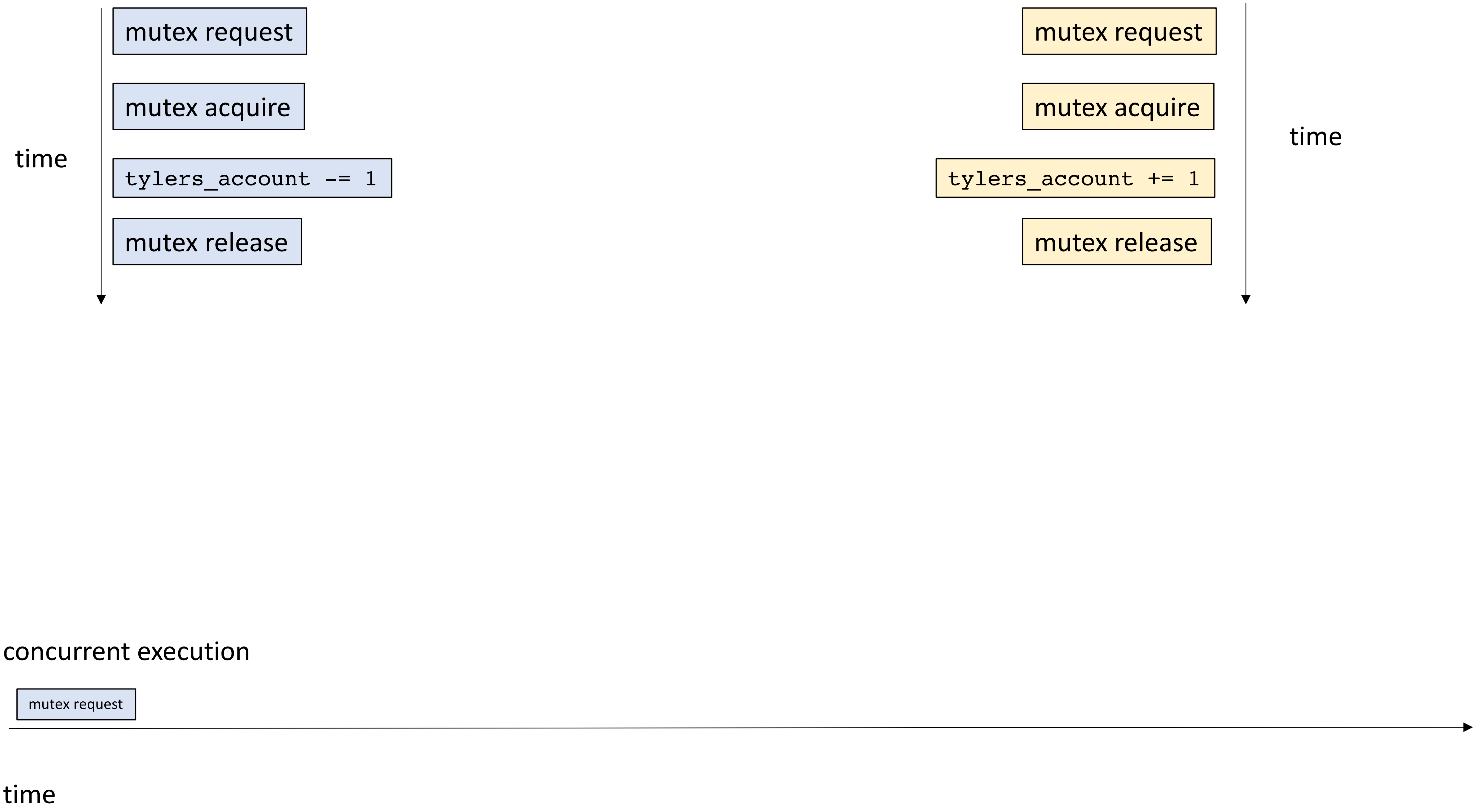
Tyler's employer

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



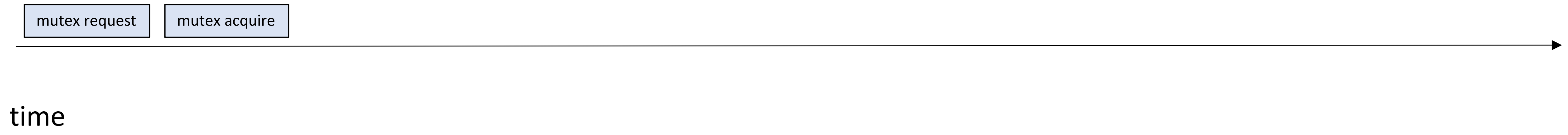


mutex request
mutex acquire
time
tylers_account -= 1
mutex release
mutex request
mutex acquire
time
tylers_account += 1
mutex release
concurrent execution
time


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

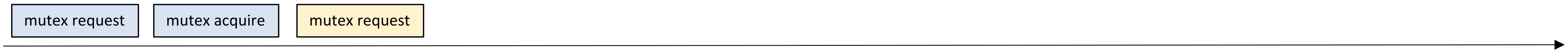
at this point, thread 0 holds the mutex.
another thread cannot acquire the mutex until thread 0 releases the mutex
also called the ***critical section.***

concurrent execution

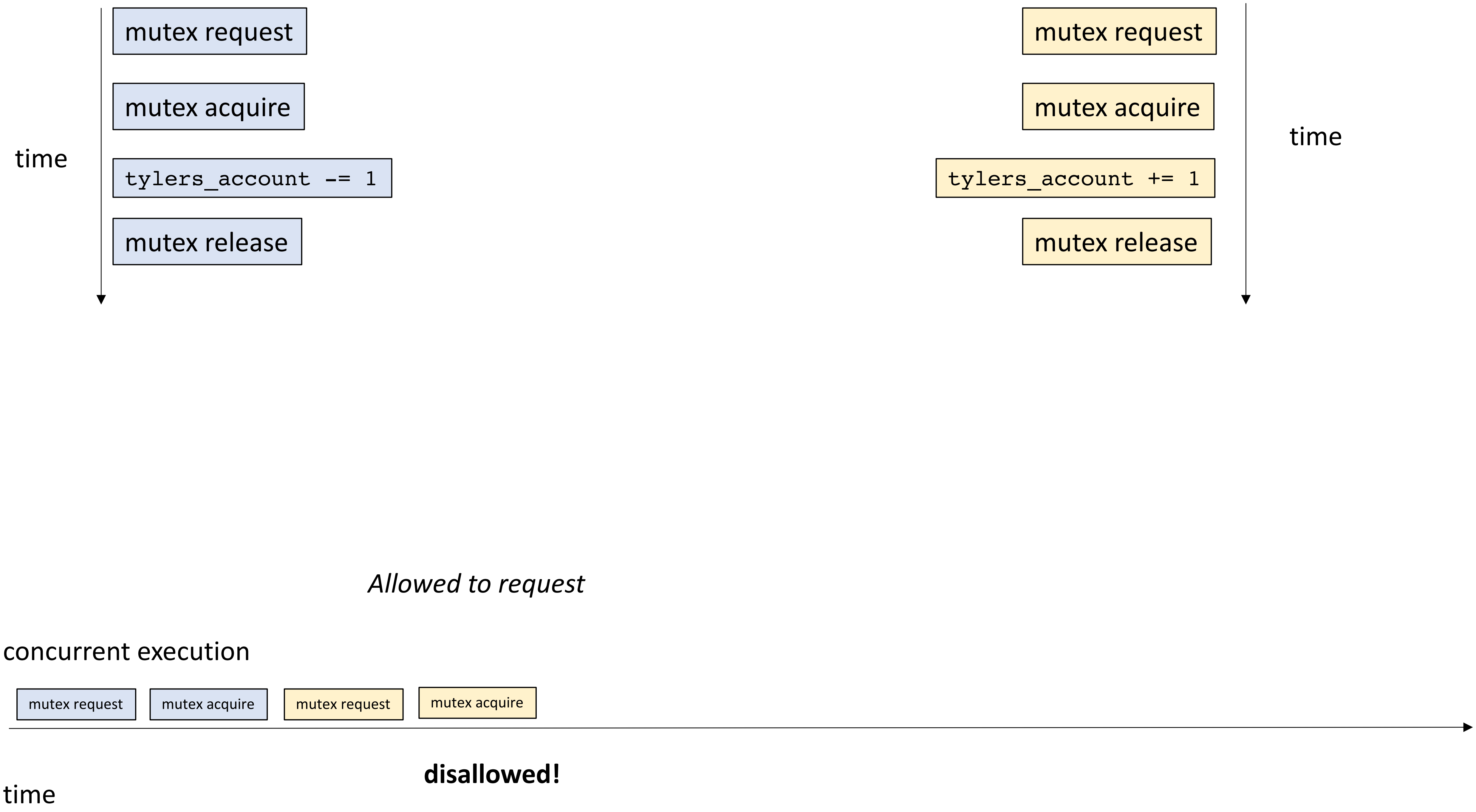


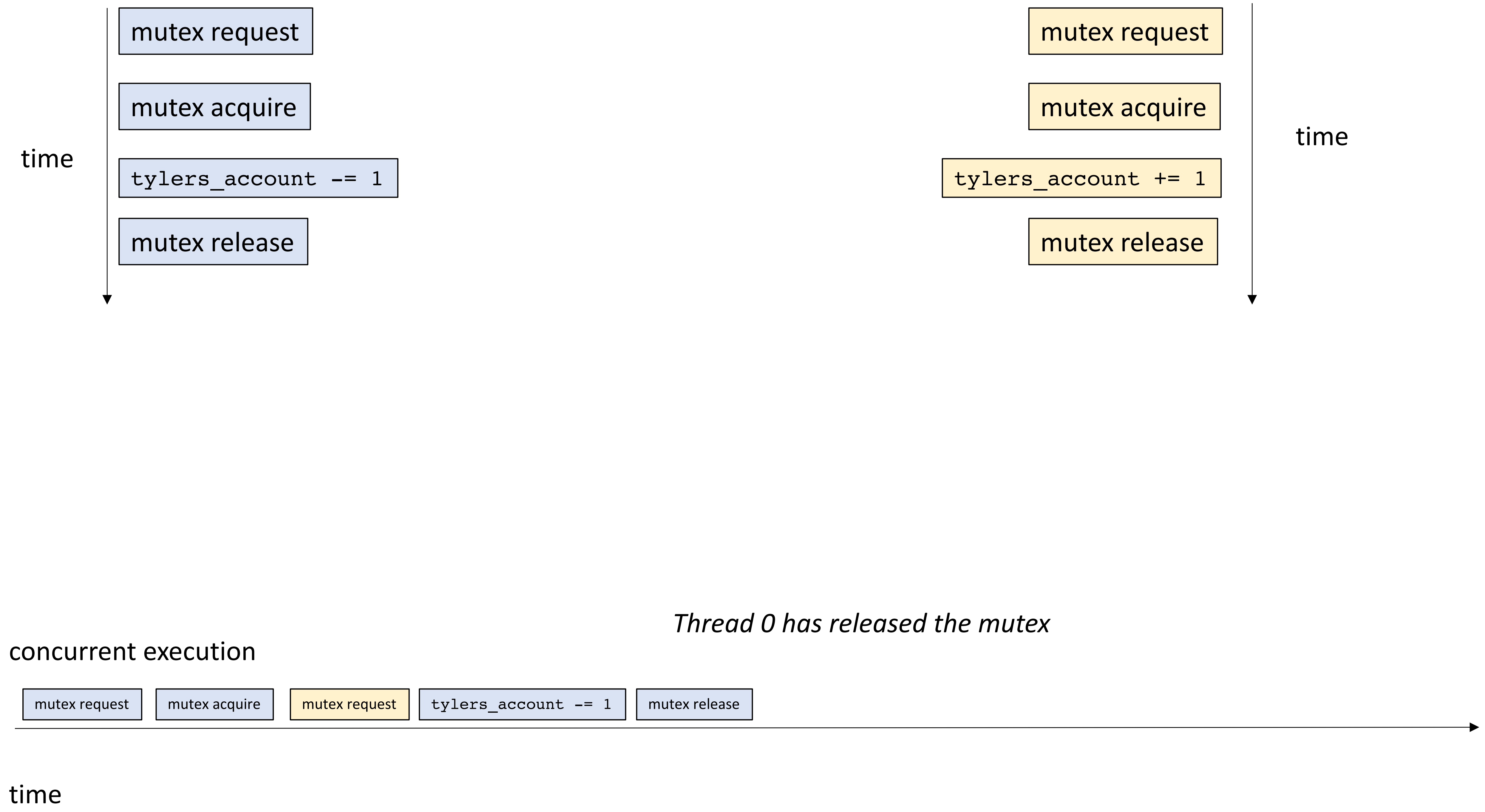
Allowed to request

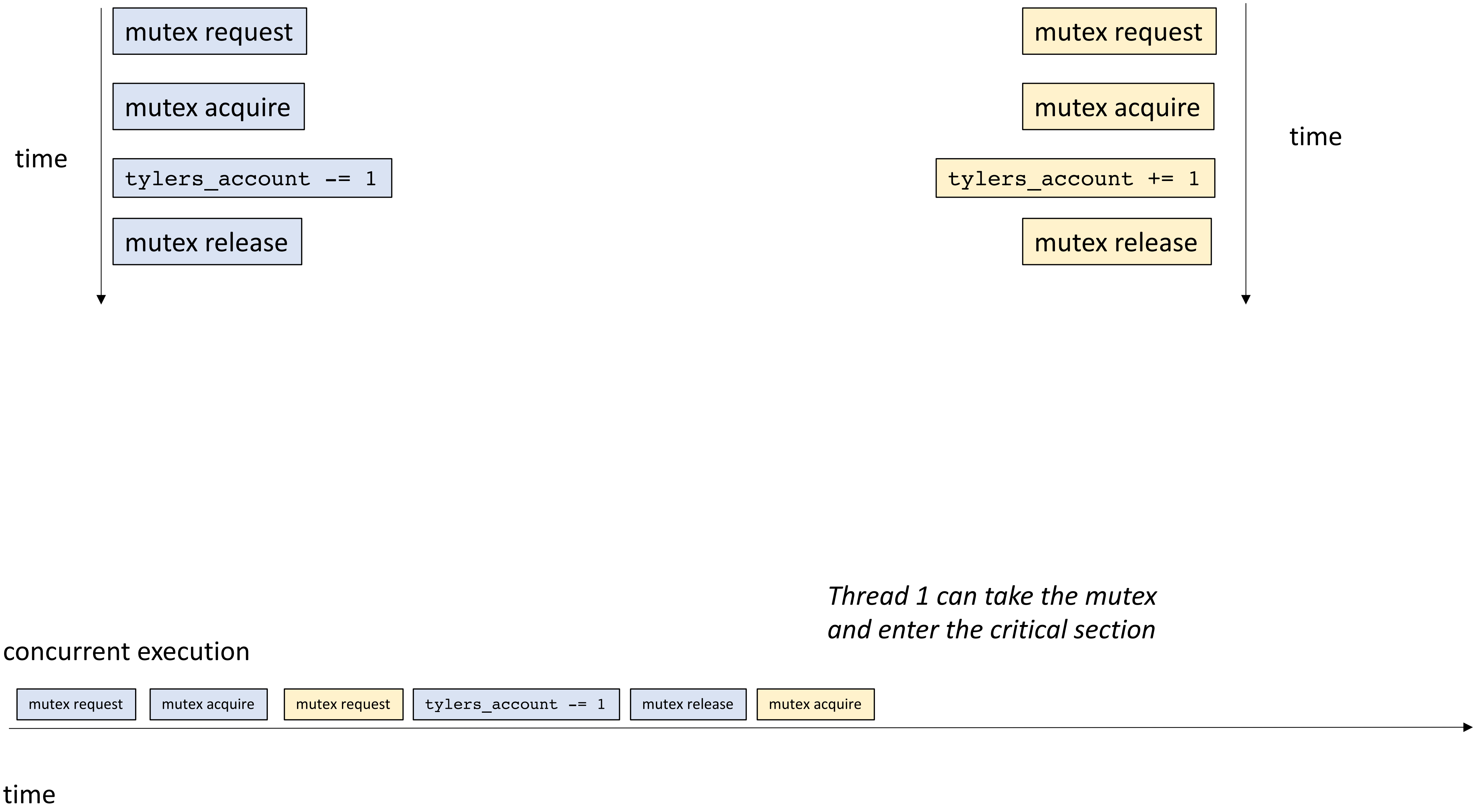
concurrent execution



time


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
mutex acquire
disallowed!
time

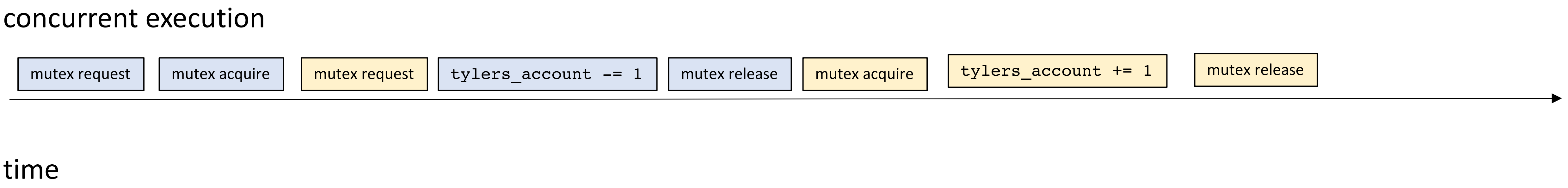

mutex request
mutex acquire
tylers_account -= 1
mutex release
time
mutex request
mutex acquire
tylers_account += 1
mutex release
time
Thread 0 has released the mutex
concurrent execution
mutex request
mutex acquire
mutex request
tylers_account -= 1
mutex release
time



time
mutex request
mutex acquire
tylers_account -= 1
mutex release
mutex request
mutex acquire
tylers_account += 1
mutex release
time
Thread 1 can take the mutex
and enter the critical section
concurrent execution
mutex request
mutex acquire
mutex request
tylers_account -= 1
mutex release
mutex acquire
time

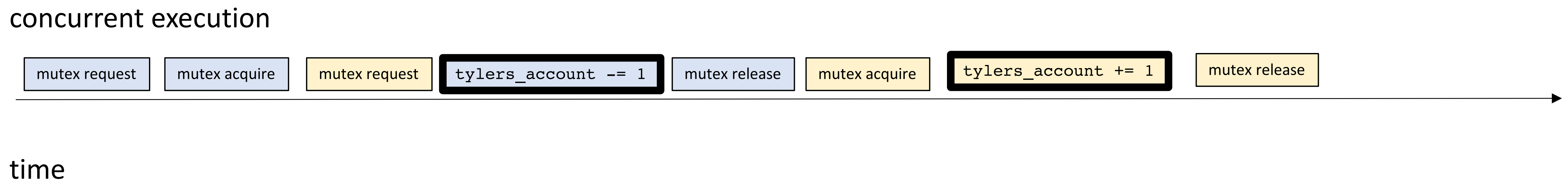
A mutex restricts the number of allowed interleavings
Critical section are mutually exclusive: i.e. they cannot interleave

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



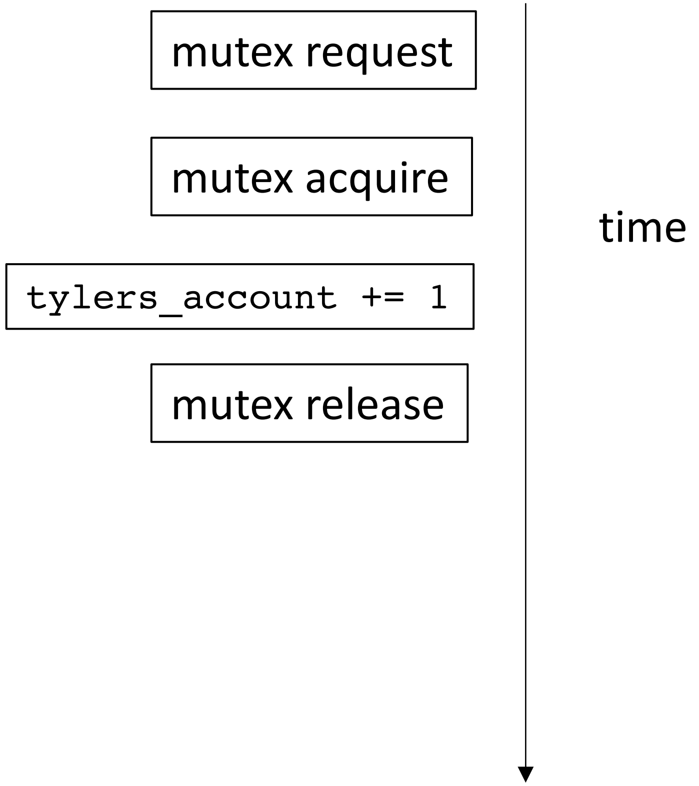
Make sure to unlock your mutex!

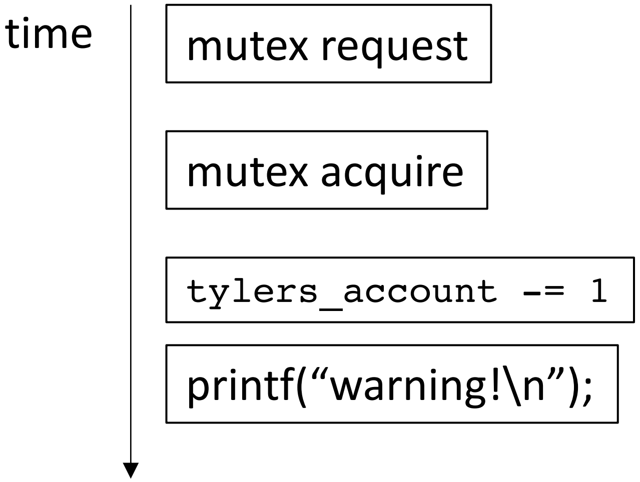
Tyler's coffee addiction:

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

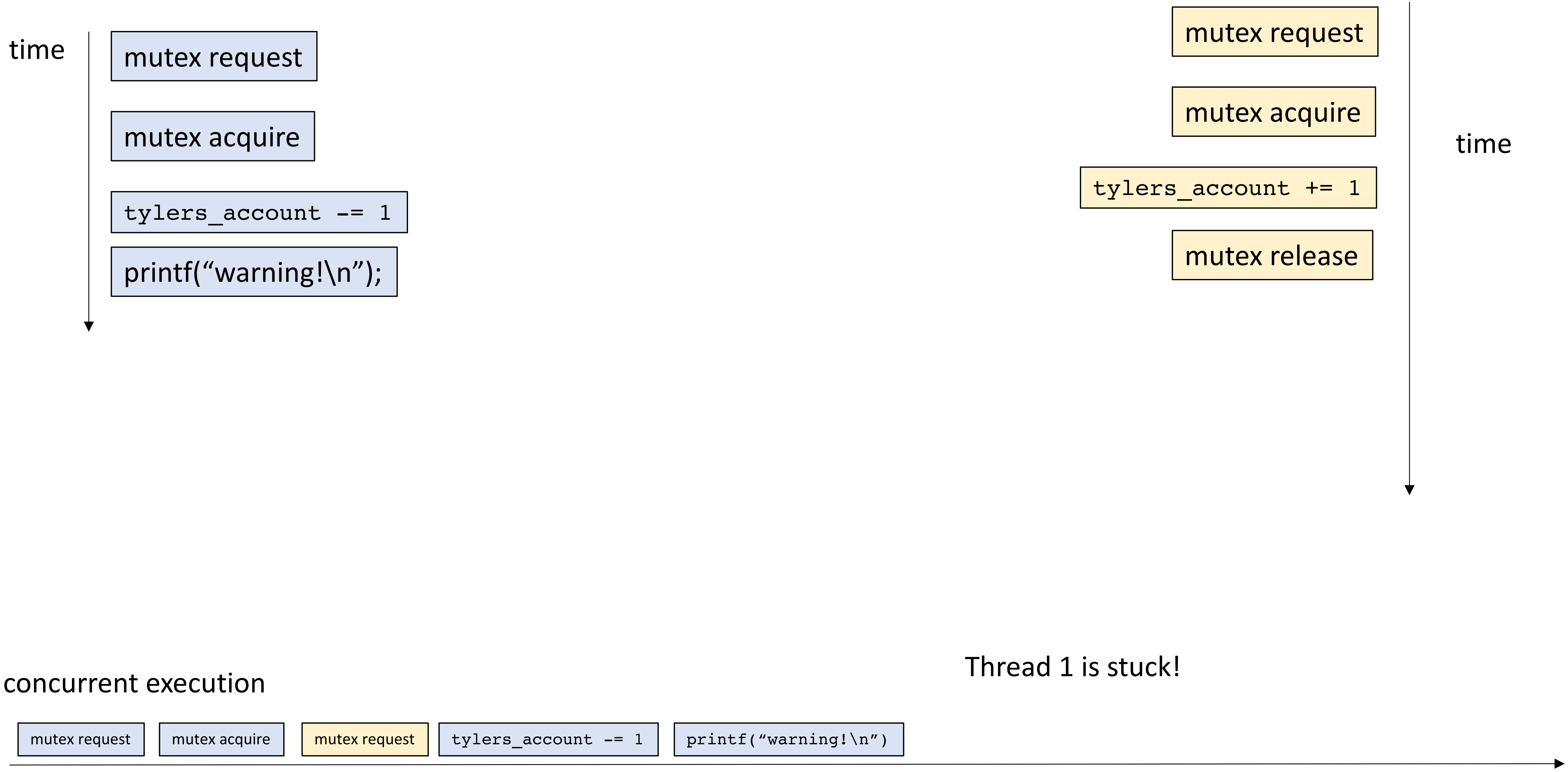
Tyler's employer

```
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")

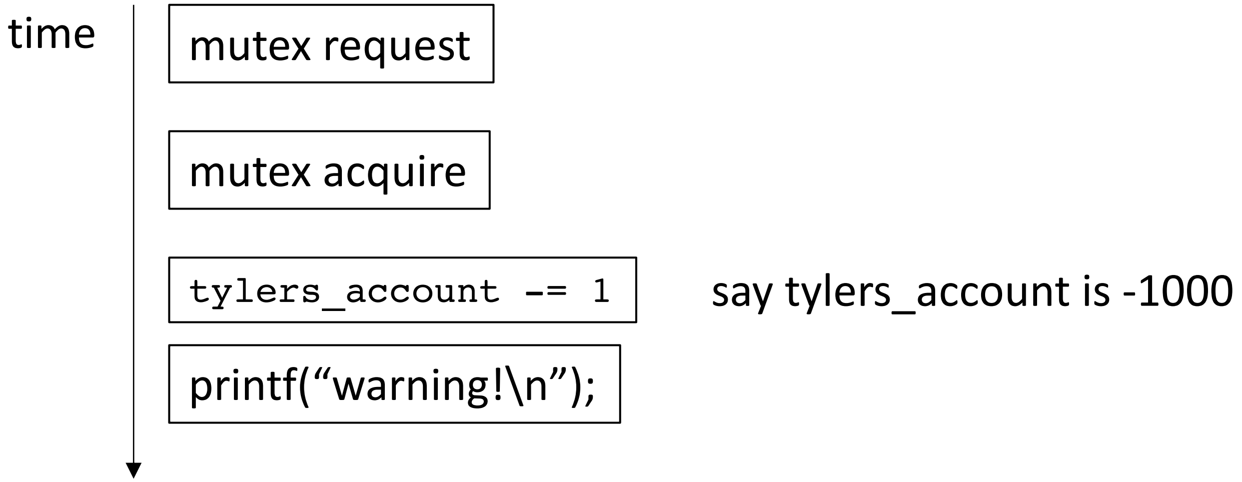
Mutexes

- C++ provides a mutex. Example

Make sure to unlock your mutex!

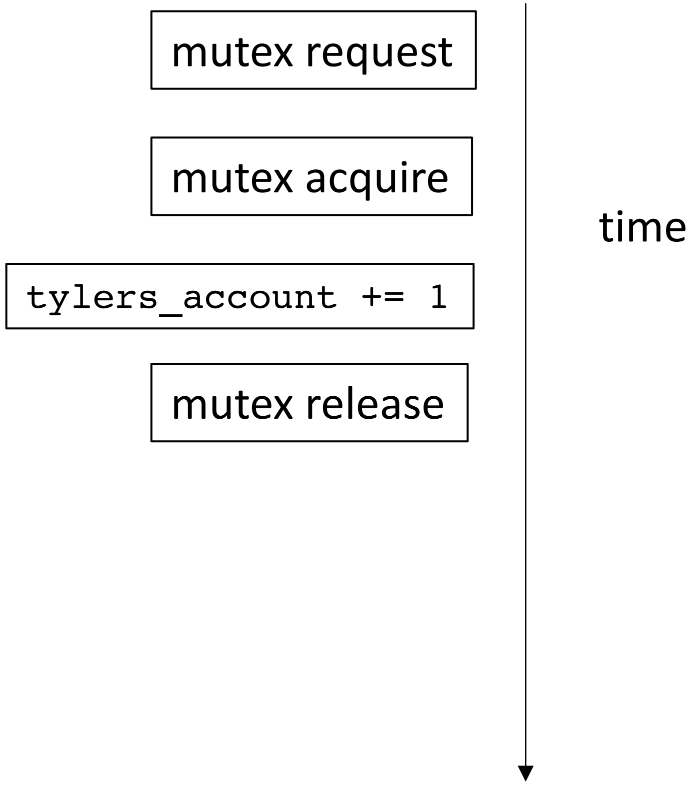
Tyler's coffee addiction:

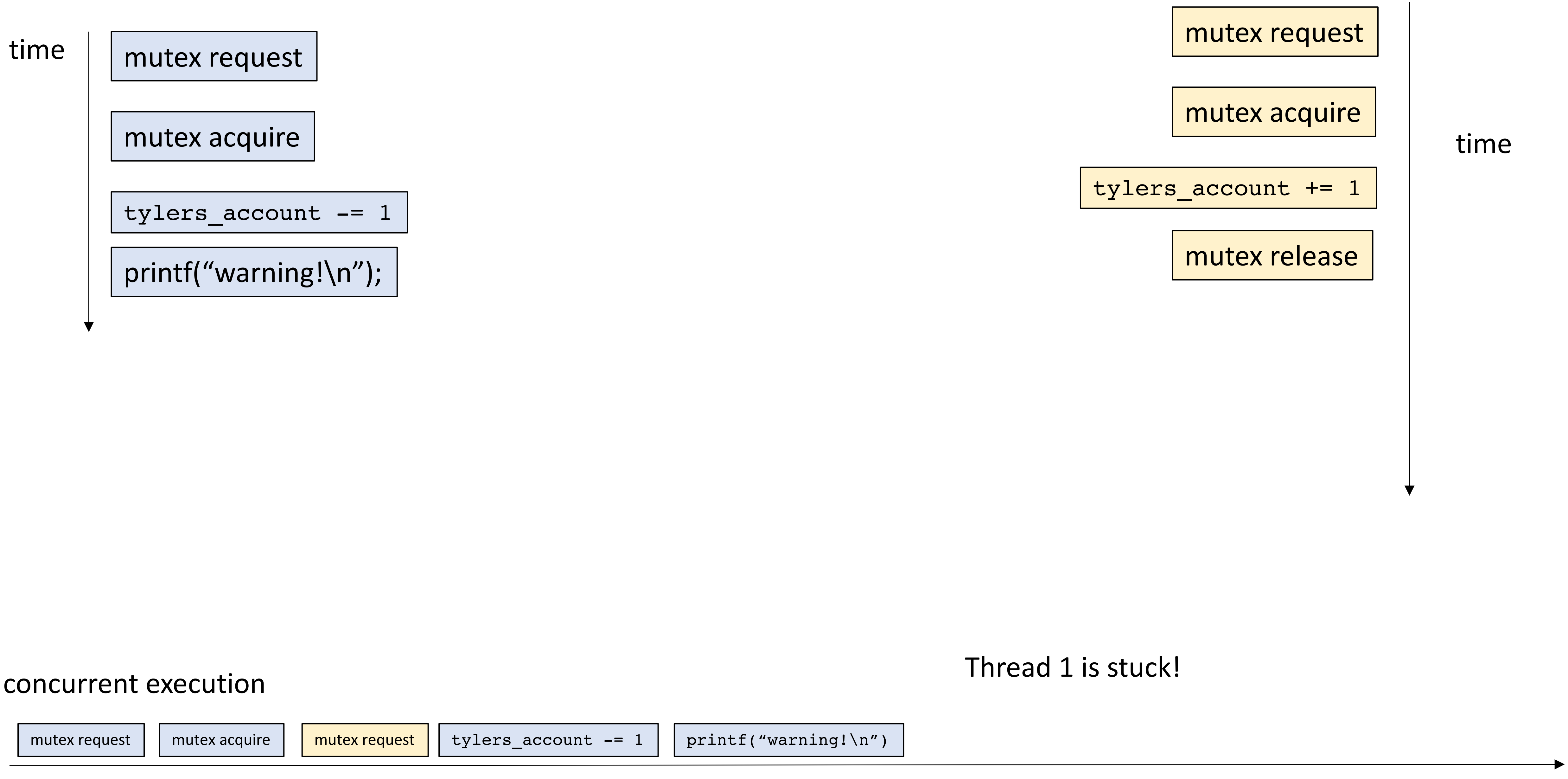
```
m.lock();
tylers_account -= 1;
if (tylers_account < -100) {
  printf("warning!\n");
  return;
}
m.unlock();
return;
```



Tyler's employer

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




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”)

Example

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”)`

Mutexes

- What about timing?

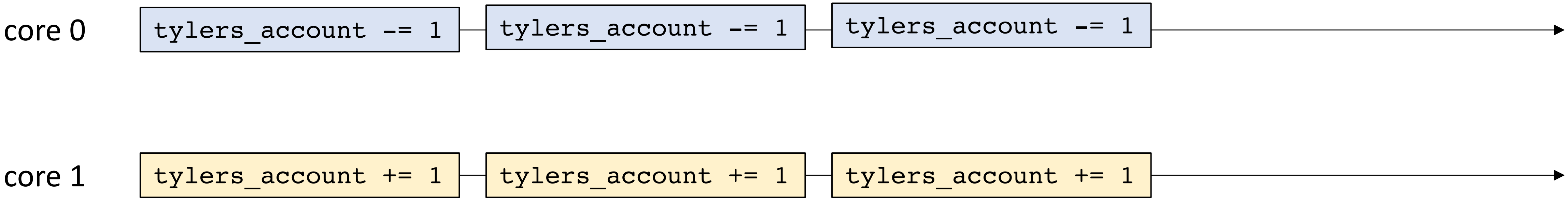
Mutexes

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

Mutexes

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

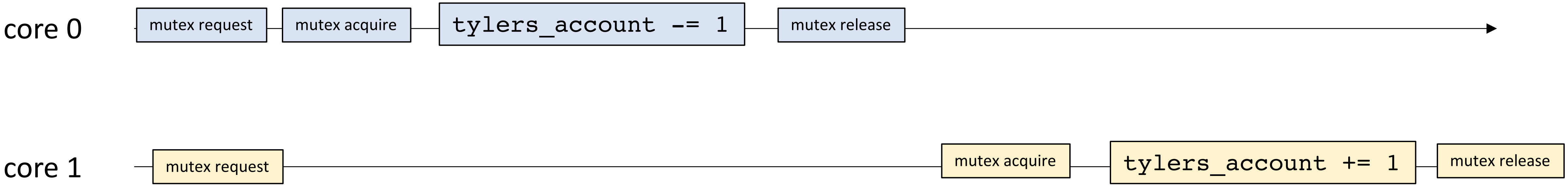
in a parallel system without the mutex



Mutexes

- 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

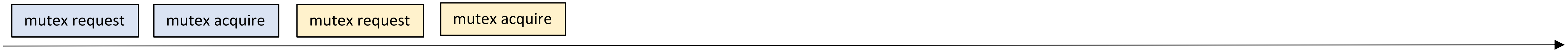
Properties of mutexes

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!

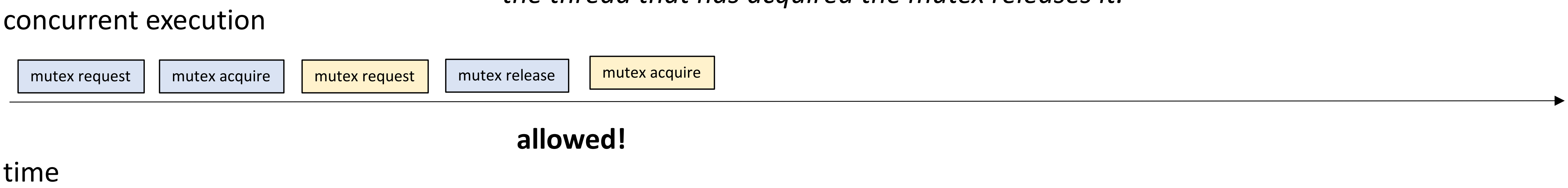
time

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

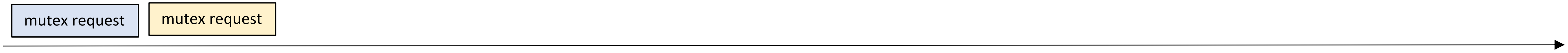
Properties of mutexes

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

concurrent execution



time

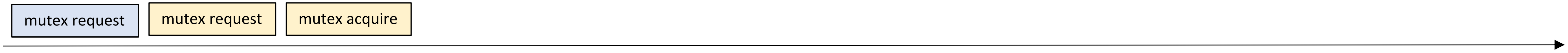
Properties of mutexes

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

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

concurrent execution

also allowed

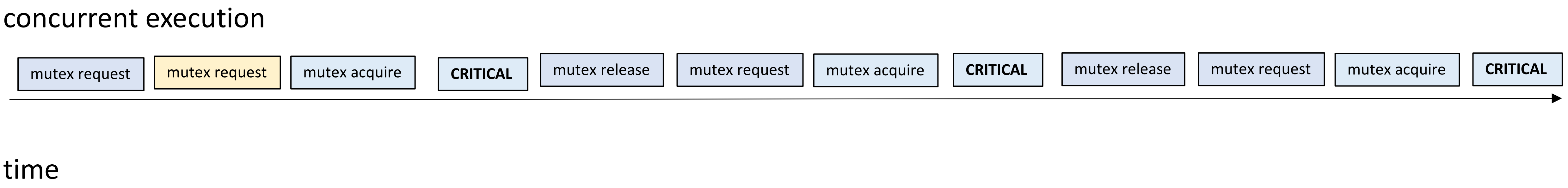
time

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



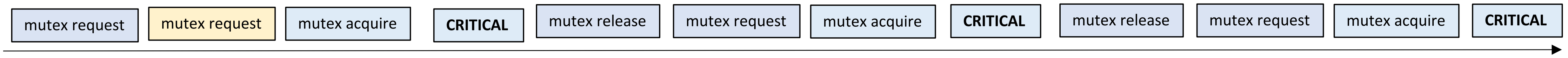
Properties of mutexes

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.

Lecture Schedule

- Canvas Quiz
- Notes on homework
- Reasoning about concurrency
- Mutual exclusion
- **Multiple mutexes**

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

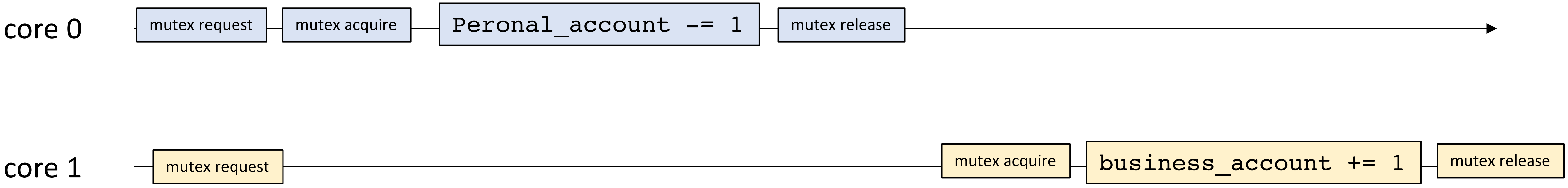
- No reason individual accounts can’t be accessed in parallel

Multiple mutexes

Lets say I have two accounts:

- Business account
- Personal account

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



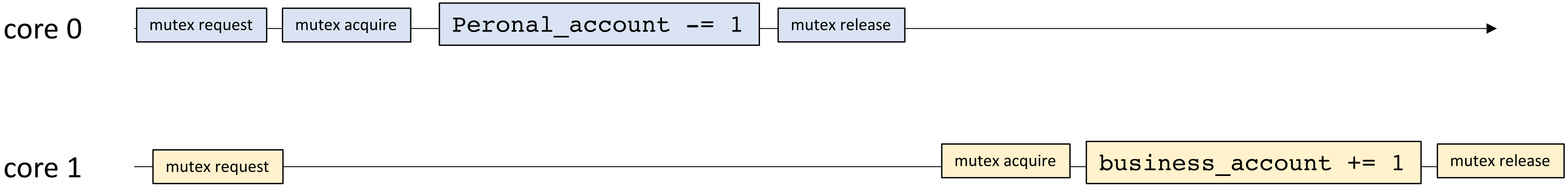
Long periods of waiting in the threads

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

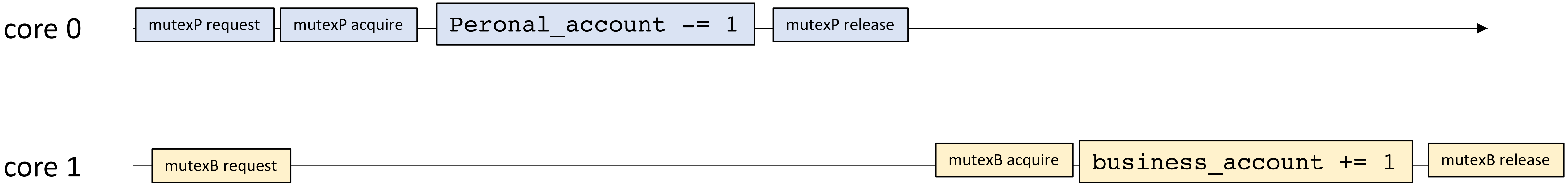


Multiple mutexes

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

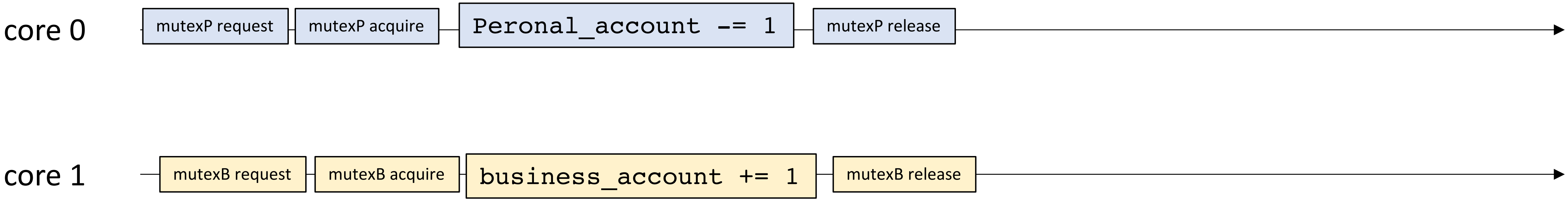


Multiple mutexes

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

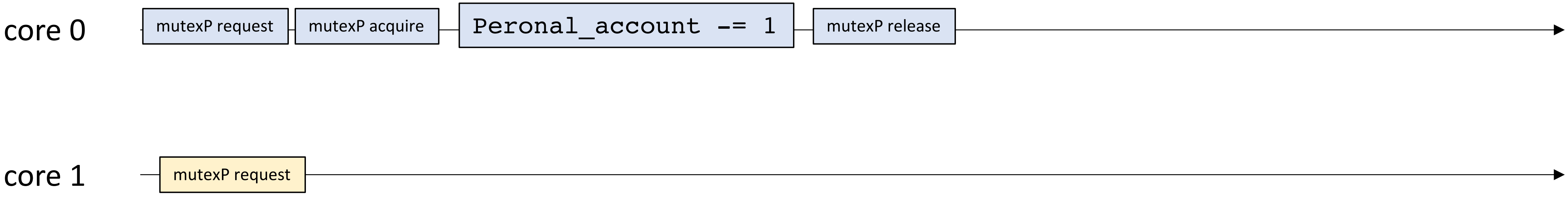


Multiple mutexes

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

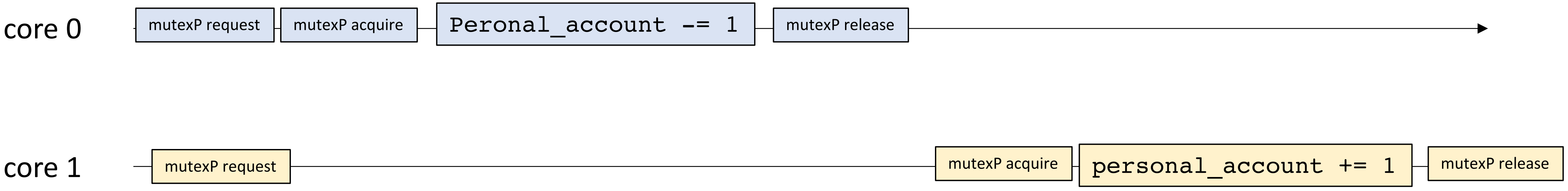


Multiple mutexes

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

Introducing mutual exclusion

Today isn't about performance, but try to keep mutual exclusion sections small!

Code example with overhead

Programming with mutexes is 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

We haven’t even talked about implementations!

Atomic RMWs

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

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:

~~`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;
```

time

Tyler's employer

```
tylers_account += 1;
```

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

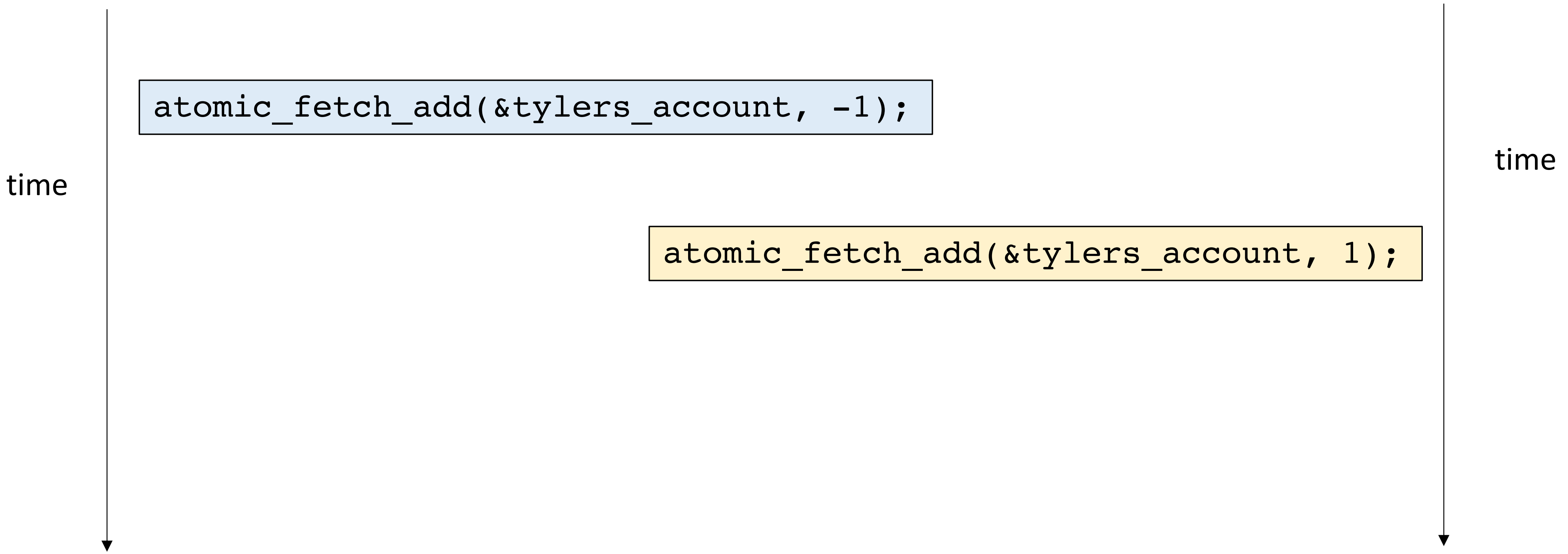
Modify these programs to use atomic RMWs

Tyler's coffee addiction:

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

Tyler's employer

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



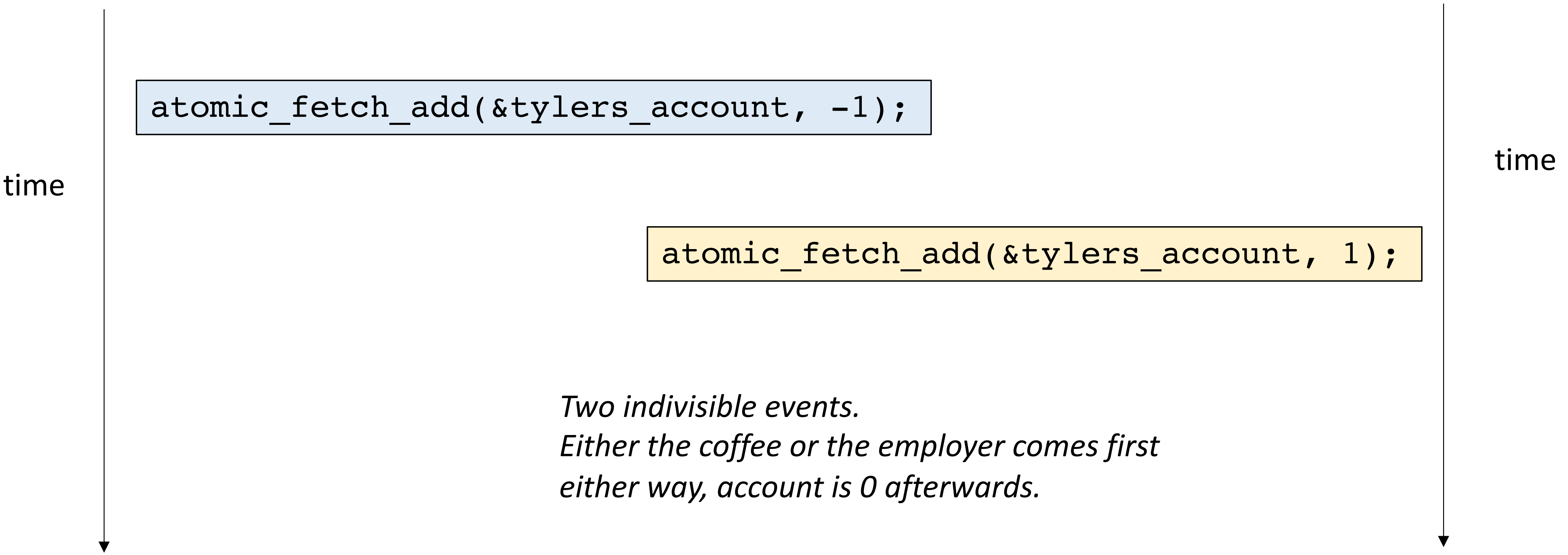
Modify these programs to use atomic RMWs

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.

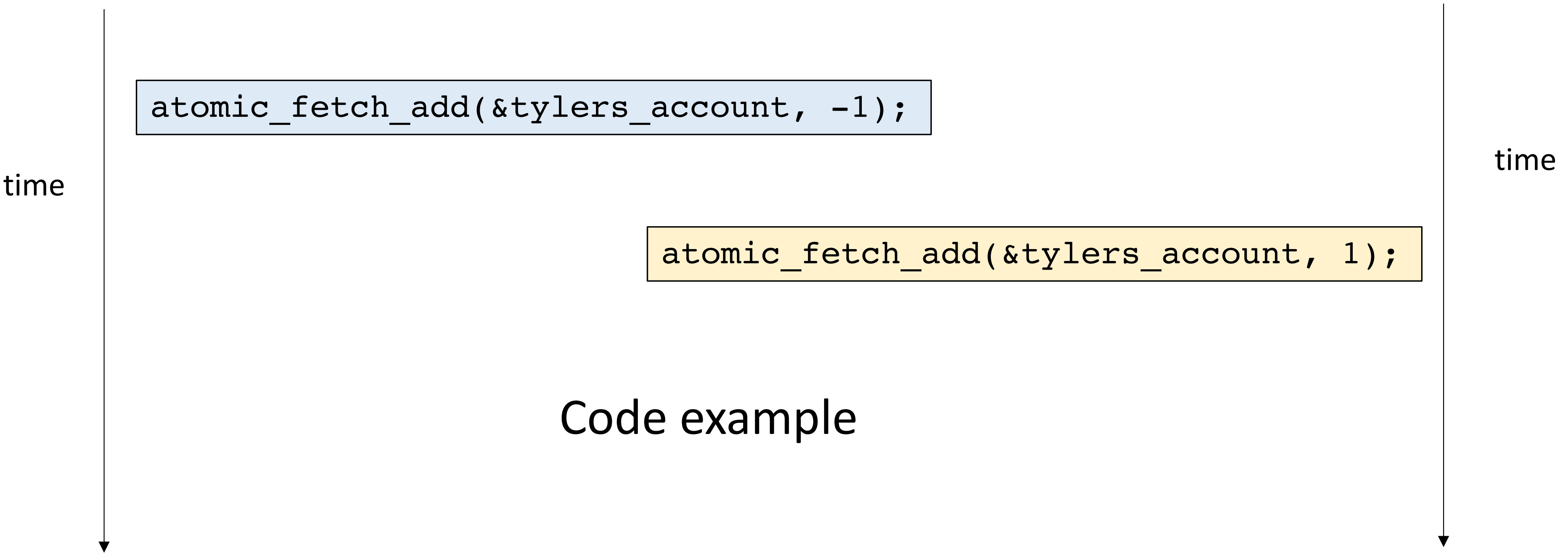
Modify these programs to use atomic RMWs

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

Finish

- Next two classes: Implementing mutexes
  - Reasoning about correctness
  - Reasoning about fairness
  - Reasoning about performance
- Final class in module:
  - specialized mutexes