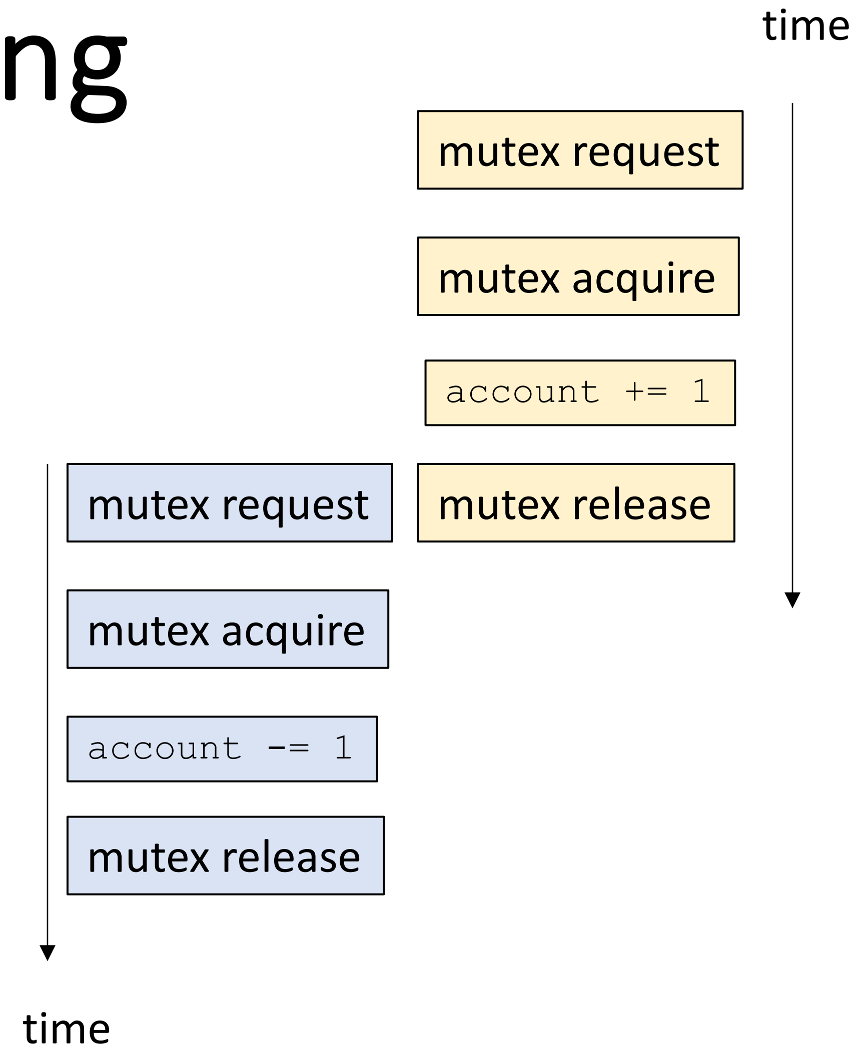
CSE113: Parallel Programming

- **Topics**:
  - Intro to mutual exclusion
    - Different types of parallelism
    - Data conflicts
    - Protecting shared data



Announcements

- Homework due on Oct 15
  - You have everything you need to get it done
  - Three free late days, nothing accepted after that
  - Plenty of office hours remaining to get help
  - Work on your design doc before asking for help
  - We do not answer questions on the weekend

- Starting on Module 2 today!

Announcements

- Homework 1 notes:
  - No assigned speedup required. You should get a noticeable speedup from ILP
  - You can start to share results on your personal machines. Everyone’s results will be slightly different
  - Sometimes you cannot account for small differences
    - Run your code for more iterations and take an average

Previous quiz

Previous quiz

How many elements of type double can be stored in a cache line?

- [ ] 1
- [ ] 2
- [ ] 4
- [ ] 8
- [ ] 16
- [ ] 32

Previous quiz

Instructions with the following property should be placed as far apart as possible in machine code:

- ☐ Instructions that compute floating point values
- ☐ Instructions that load from memory
- ☐ Instructions that depend on each other
- ☐ Instructions that perform the same operation

Previous quiz

What does ILP stand for?

- [ ] Interleaved Language Program
- [ ] Instruction Level Parallelism
- [ ] Interpreted Latency Pipeline

Previous quiz

C++ threads are initialized with a function argument where they will start execution, but they must be explicitly started with the "launch" command.

- ○ True
- ○ False

Previous quiz

The "join" function for a C++ thread causes the thread to immediately exit.

- ○ True
- ○ False

Previous quiz

A thread that is launched will eventually exit by itself and there is no need for the main thread to keep track of the threads it launches.

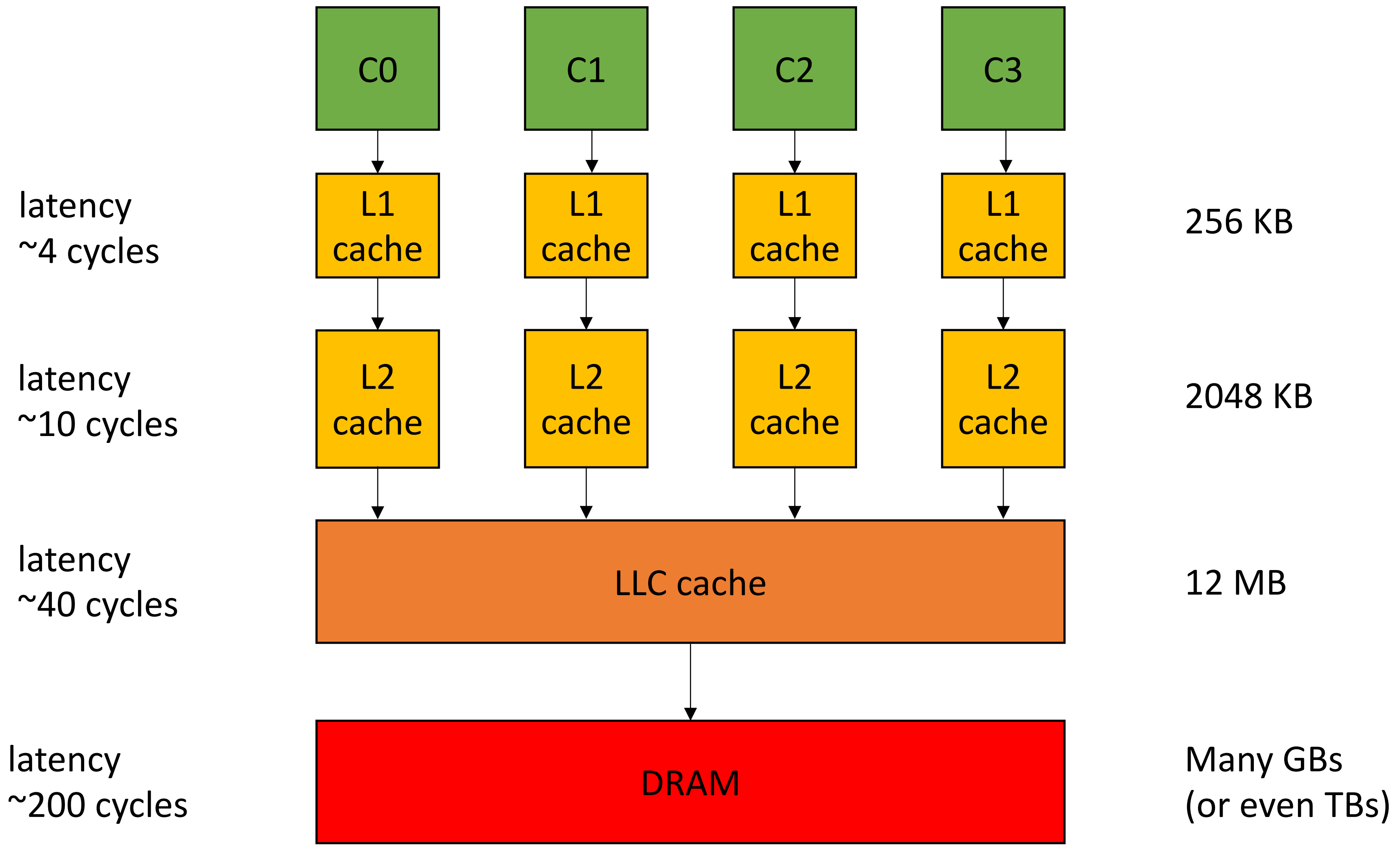
- ○ True
- ○ False

Previous quiz

In 2 or 3 sentences, explain the difference between instruction level parallelism and thread parallelism

Review

Caches

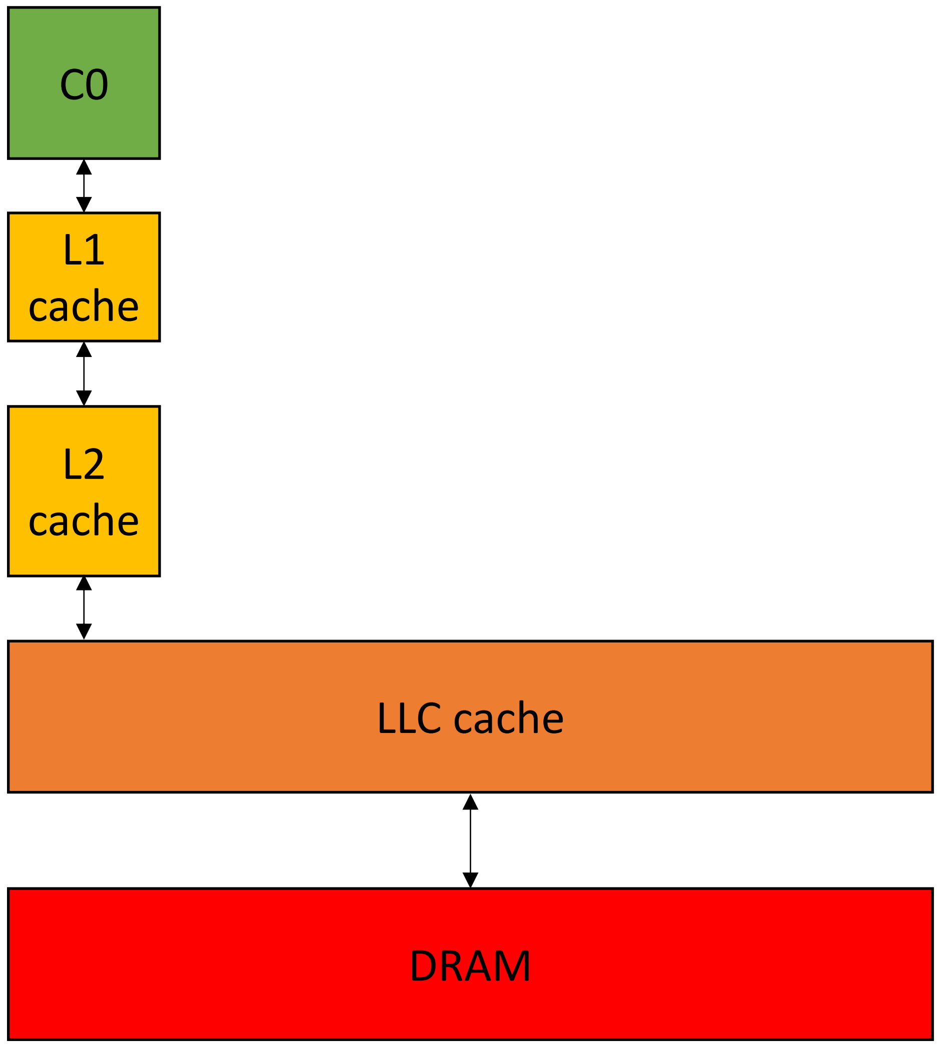

C0
C1
C2
C3
latency
~4 cycles
L1 cache
L1 cache
L1 cache
L1 cache
256 KB
latency
~10 cycles
L2 cache
L2 cache
L2 cache
L2 cache
2048 KB
latency
~40 cycles
LLC cache
12 MB
latency
~200 cycles
DRAM
Many GBs
(or even TBs)

Caches

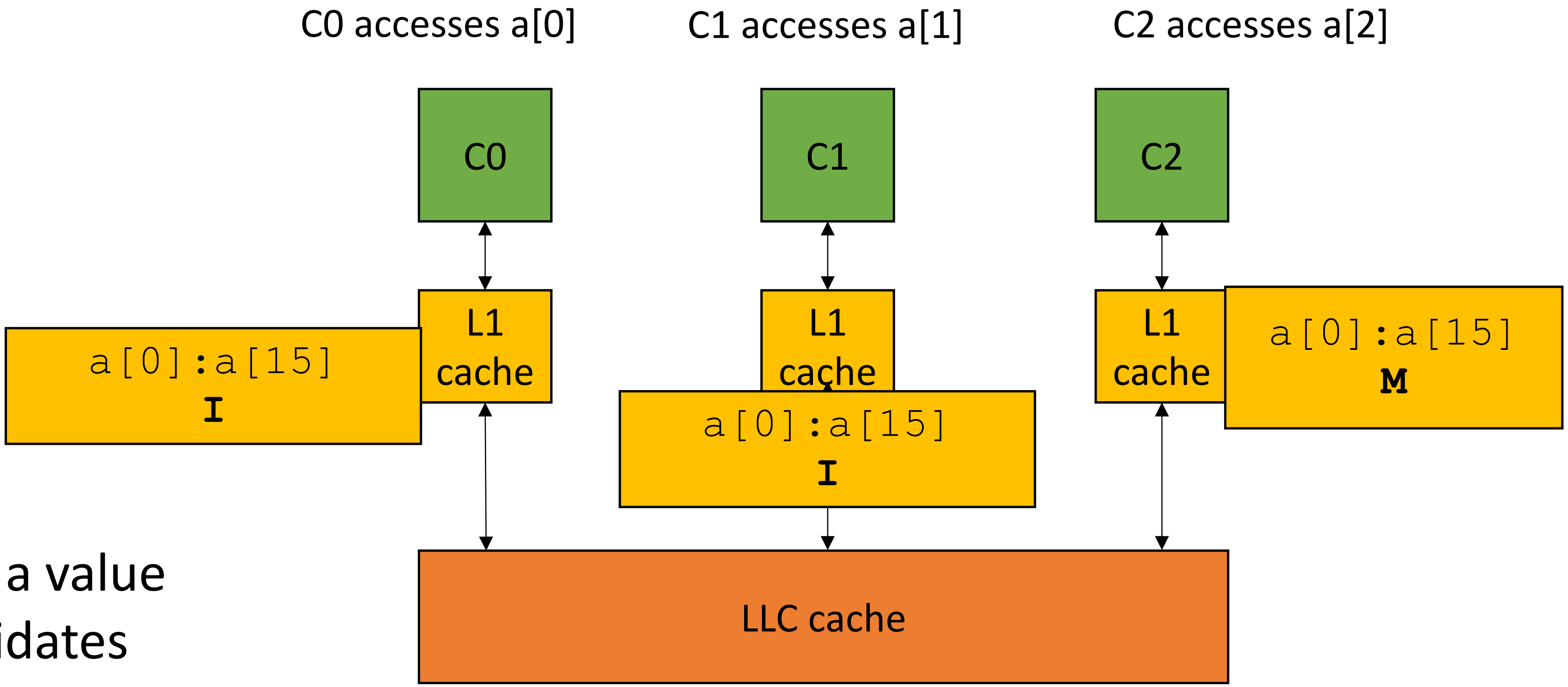
```
int increment(int *a) {
    a[0]++;
}
```

```
%5 = load i32, i32* %4        4 cycles
%6 = add nsw i32 %5, 1        1 cycles
store i32 %6, i32* %4         4 cycles
```

9 cycles!



Cache Coherence and False Sharing



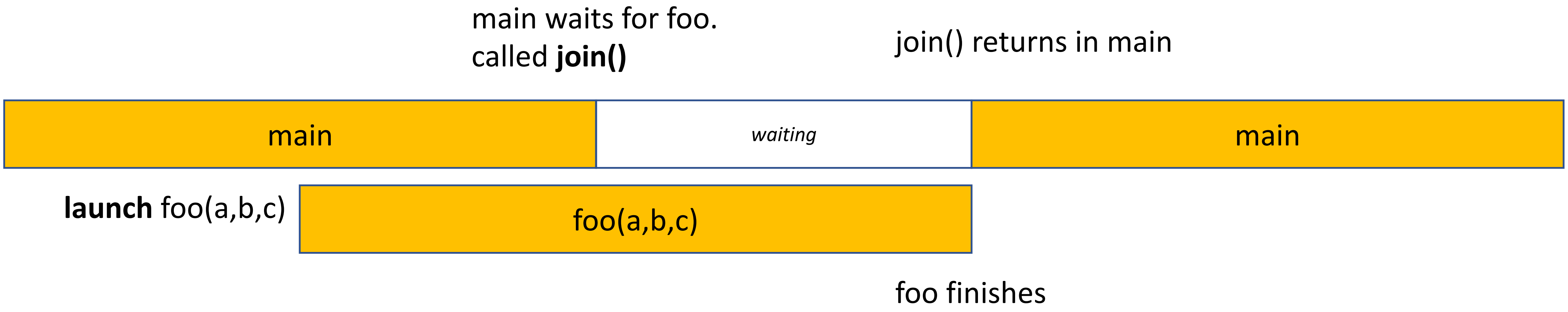
when one core modifies a value in the cache line, it invalidates everyone else's cache line.

This is called ***False Sharing***

```cpp
#include <thread>
using namespace std;

void foo(int a, int b, int c) {
  // some foo code
}

int main() {
  // some main code
  thread thread_handle (foo,1,2,3);
  // code here runs concurrently with foo
  thread_handle.join();
  return 0;
}
```



```cpp
#include <thread>
#include <iostream>
using namespace std;

void foo(int a, int b, int *c) {
  // return a + b;
  *c = a + b;
}

int main() {
  // some main code
  int ret = 0;
  thread thread_handle (foo,1,2, &ret);
  // code here runs concurrently with foo
  cout << ret << endl;
  thread_handle.join();
  return 0;
}
```

What if....

SPMD programming model

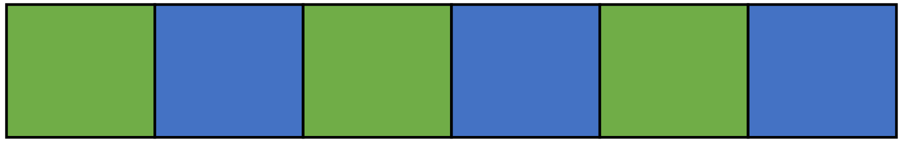
- Same program, multiple data
- Main idea: many threads execute the same function, but they operate on different data.
- How do they get different data?
  - each thread can access their own thread id, a contiguous integer starting at 0 up to the number of threads

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

iterations computed by thread 1



array a

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

SPMD programming model

```
void increment_array(int *a, int a_size, int tid, int num_threads);


#define THREADS 8
#define A_SIZE 1024
int main() {
  int *a = new int[A_SIZE];
  // initialize a
  thread thread_ar[THREADS];

  for (int i = 0; i < THREADS; i++)
    thread_ar[i] = thread(increment_array, a, A_SIZE, i, THREADS);
  for (int i = 0; i < THREADS; i++)
    thread_ar[i].join();

  delete[] a;
  return 0;
}
```

New material

Concurrency vs. Parallelism

- Abstract tasks:
  - In the abstract: a sequence of computation
  - *Given an input, produces an output*

Concurrency vs. Parallelism

- Abstract tasks:
  - In the abstract: a sequence of computation
  - *Given an input, produces an output*

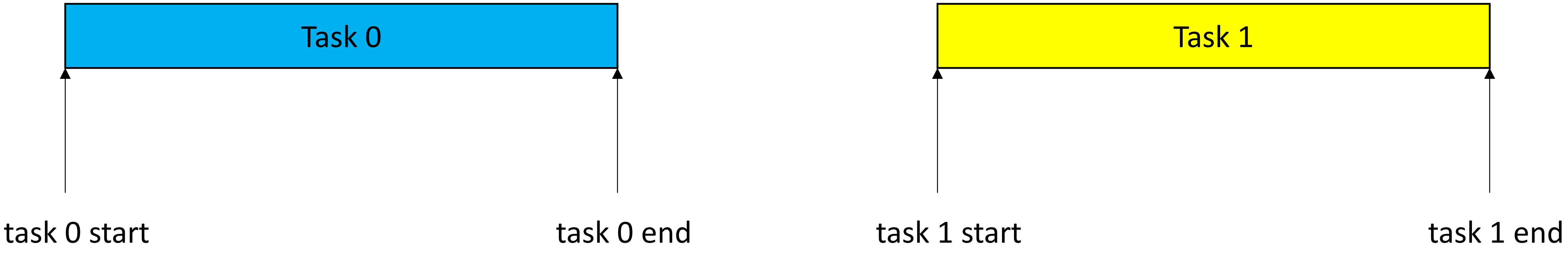
- Concrete tasks:
  - Application (e.g. Spotify and Chrome)
  - Function
  - Loop iterations
  - Individual instructions

coarse

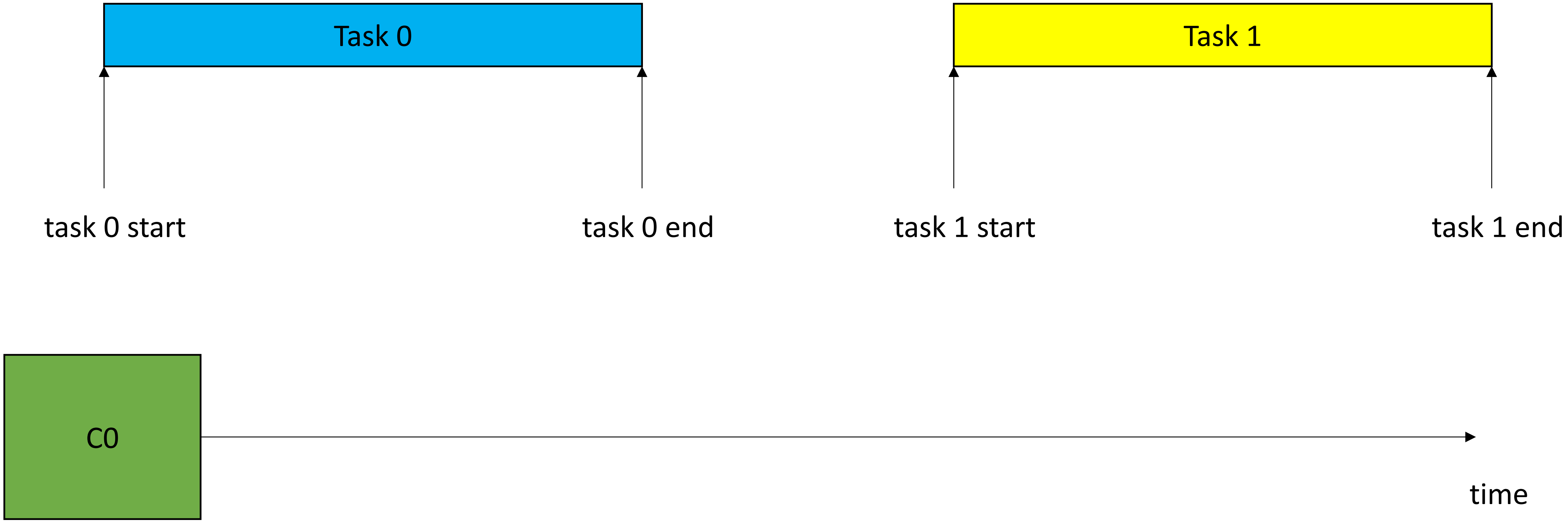
granularity

fine

Concurrency vs. Parallelism

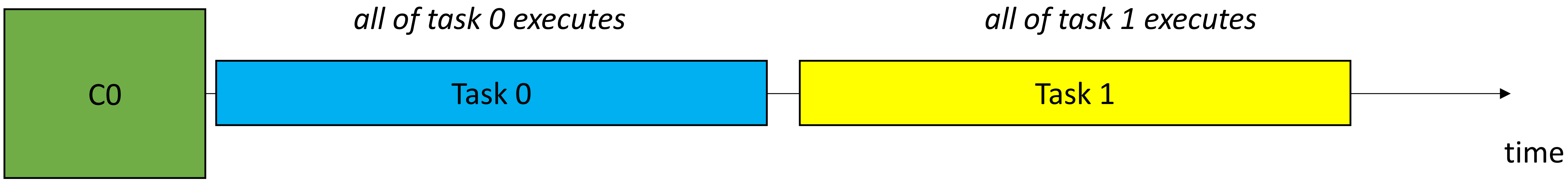


Concurrency vs. Parallelism



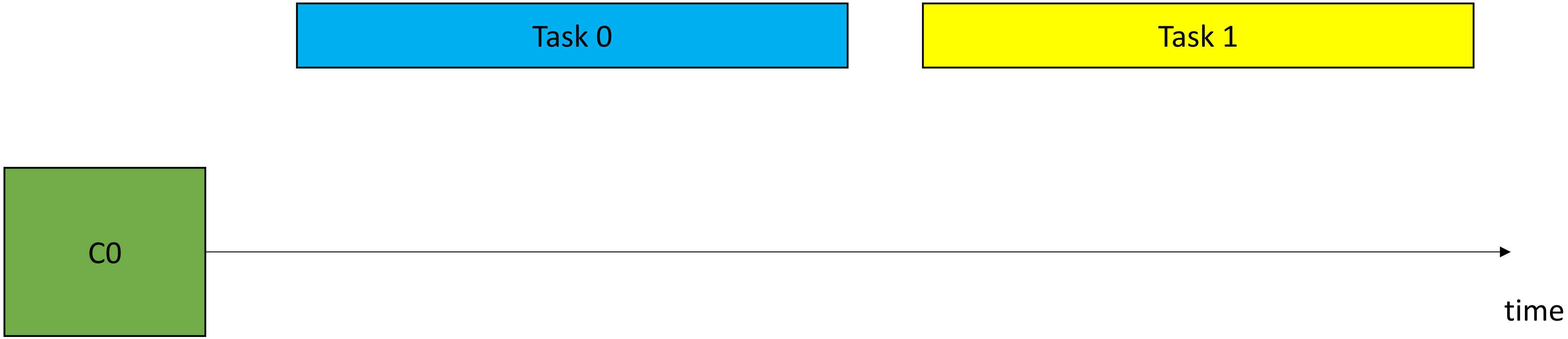
Concurrency vs. Parallelism

Sequential execution
Not concurrent or parallel



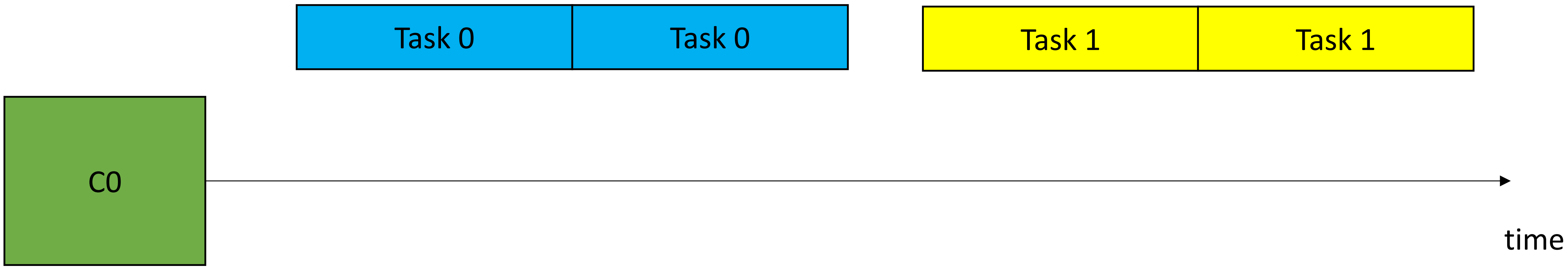
Concurrency vs. Parallelism

The OS can preempt a thread
(remove it from the hardware resource)



Concurrency vs. Parallelism

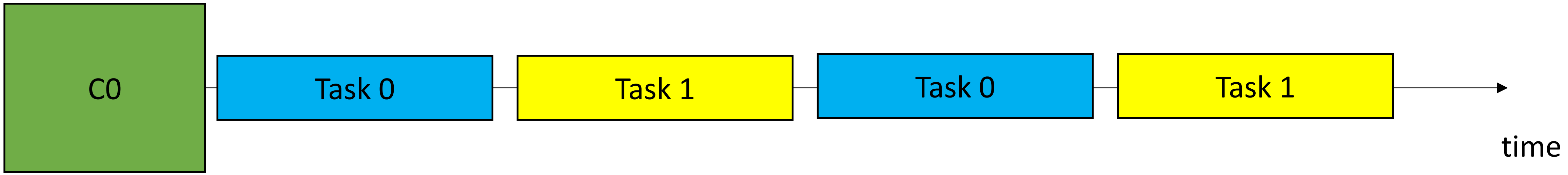
The OS can preempt a thread
(remove it from the hardware resource)



Concurrency vs. Parallelism

The OS can preempt a thread
(remove it from the hardware resource)

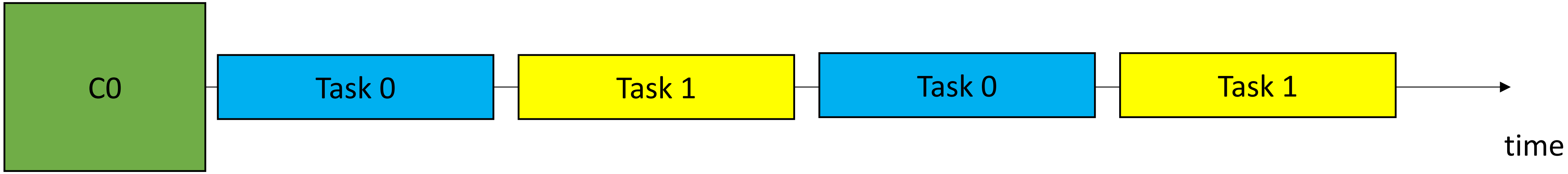
tasks are interleaved on the same processor



Concurrency vs. Parallelism

- Definition:
  - 2 tasks are **concurrent** if there is a point in the execution where both tasks have started and neither has ended.

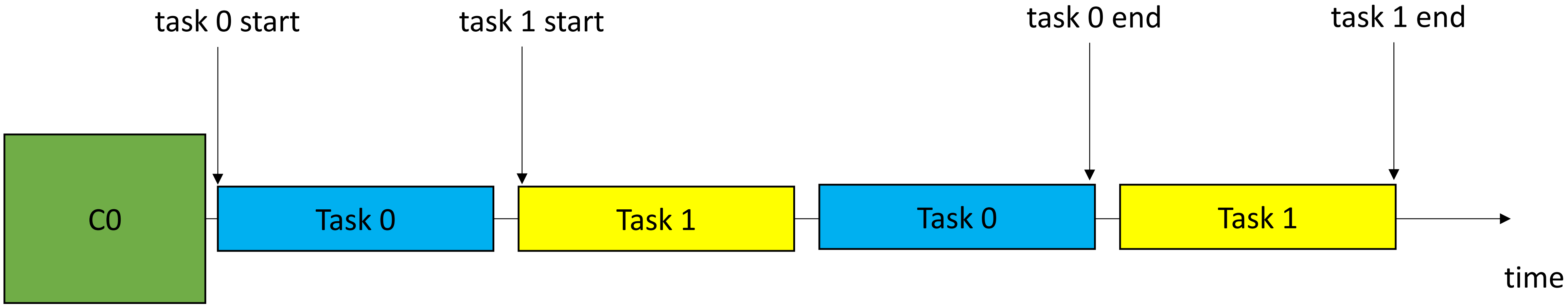
The OS can preempt a thread
(remove it from the hardware resource)



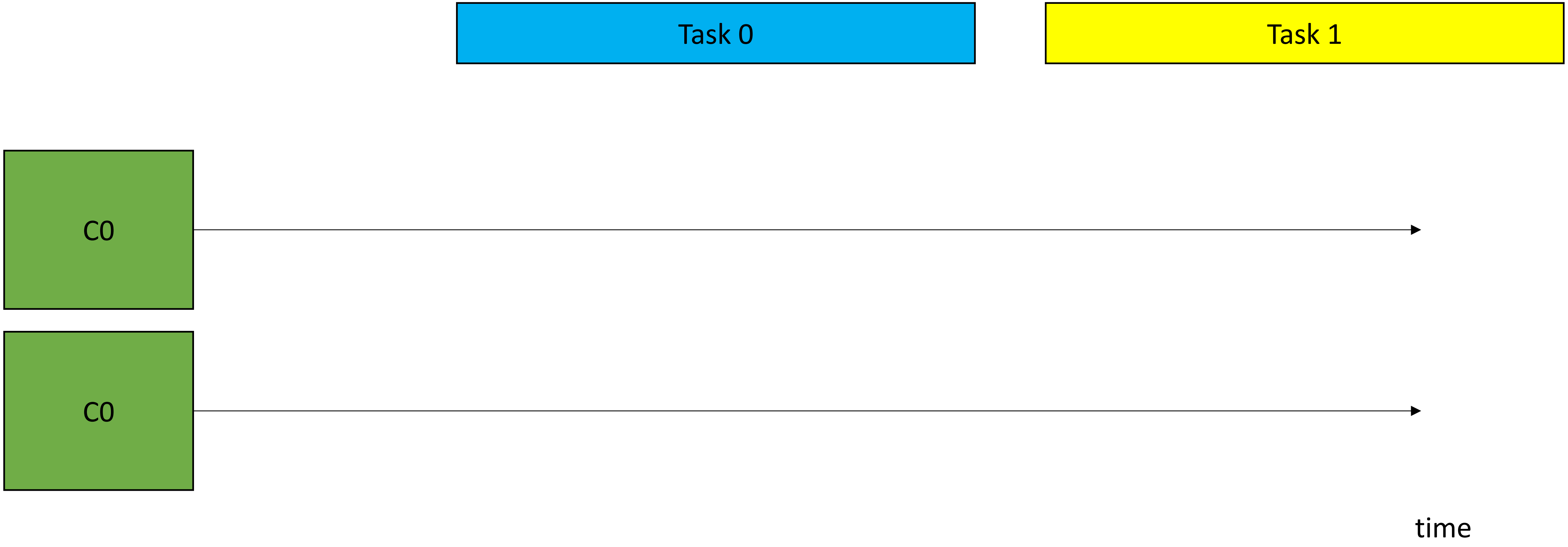
Concurrency vs. Parallelism

- Definition:
  - 2 tasks are **concurrent** if there is a point in the execution where both tasks have started and neither has ended.

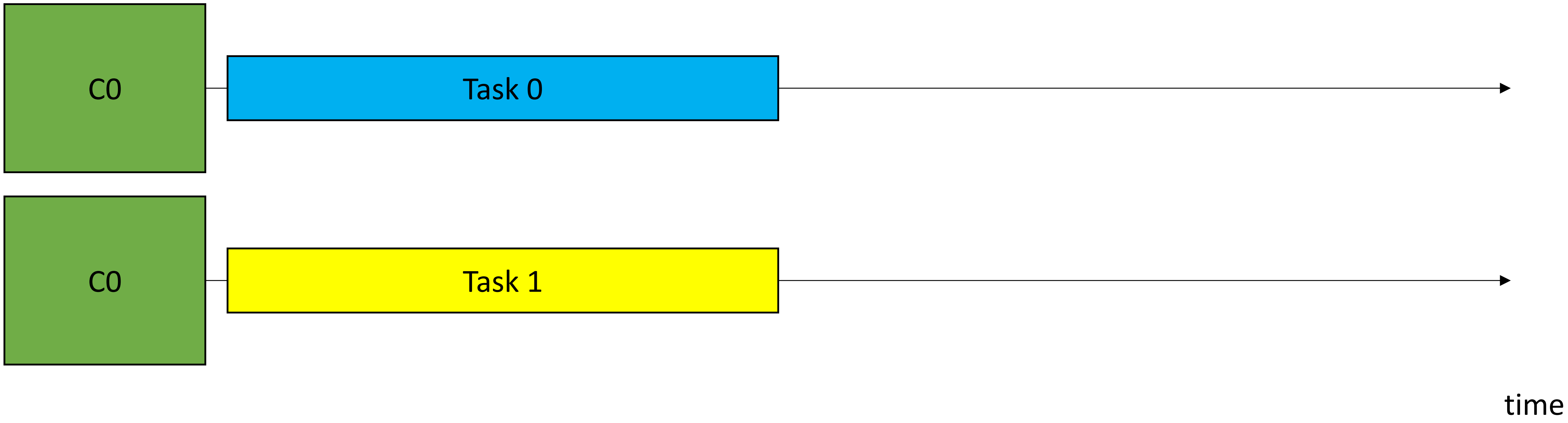
The OS can preempt a thread
(remove it from the hardware resource)



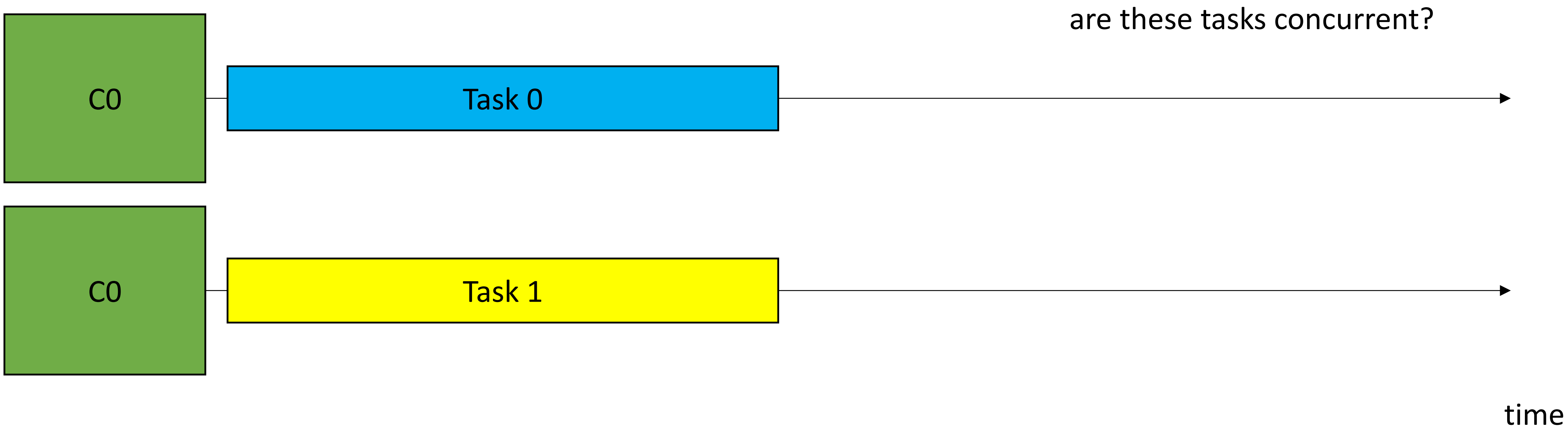
Concurrency vs. Parallelism



Concurrency vs. Parallelism



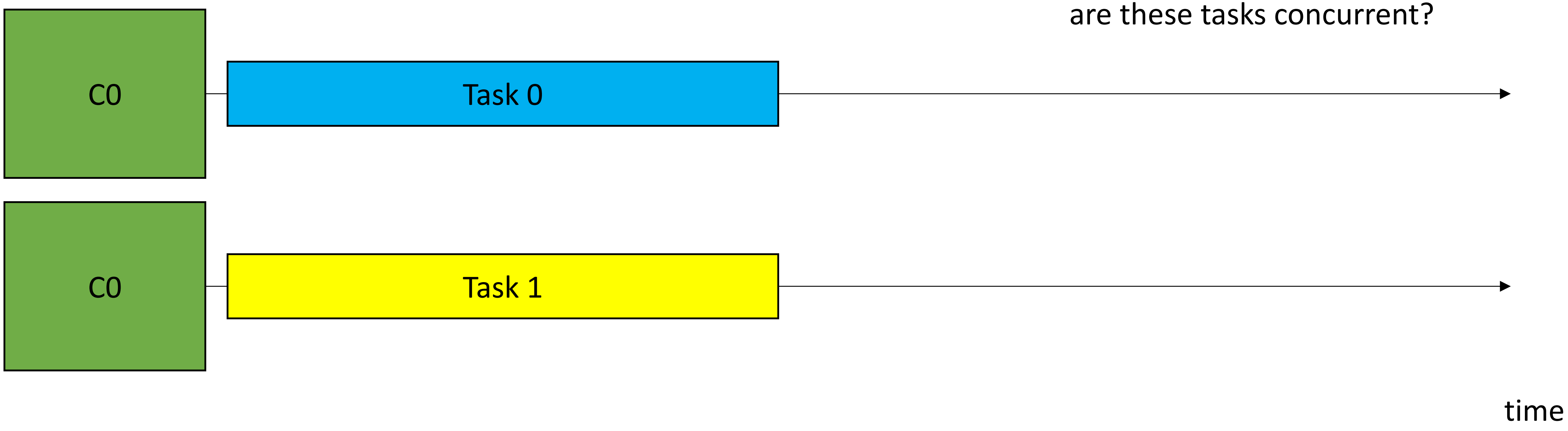
Concurrency vs. Parallelism



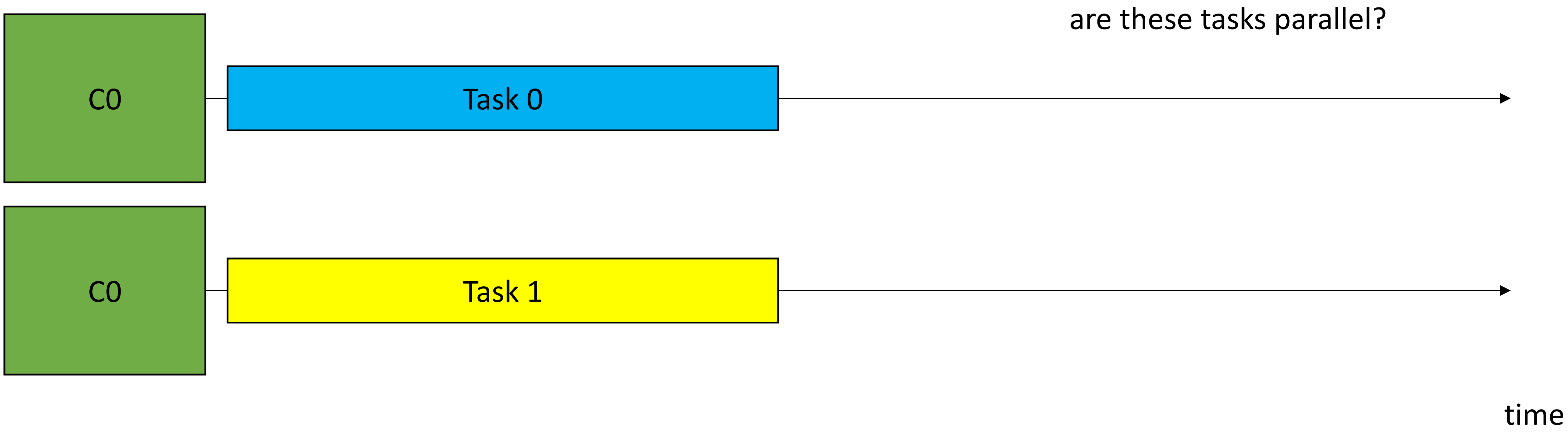
Concurrency vs. Parallelism

- 2 tasks are **concurrent** if there is a point in the execution where both tasks have started and neither has ended.

are these tasks concurrent?

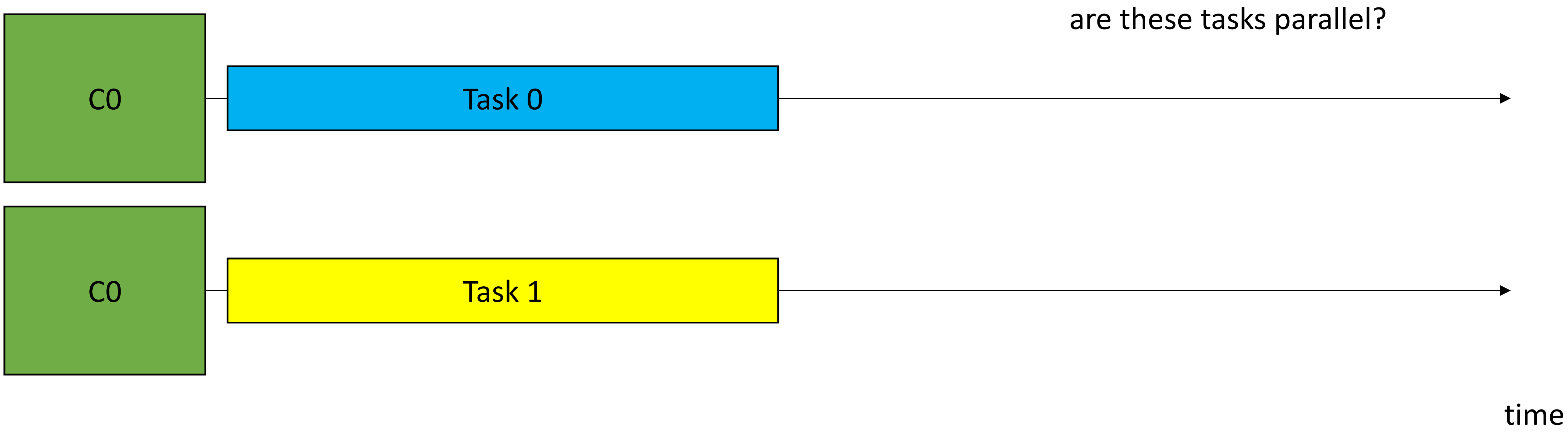


Concurrency vs. Parallelism



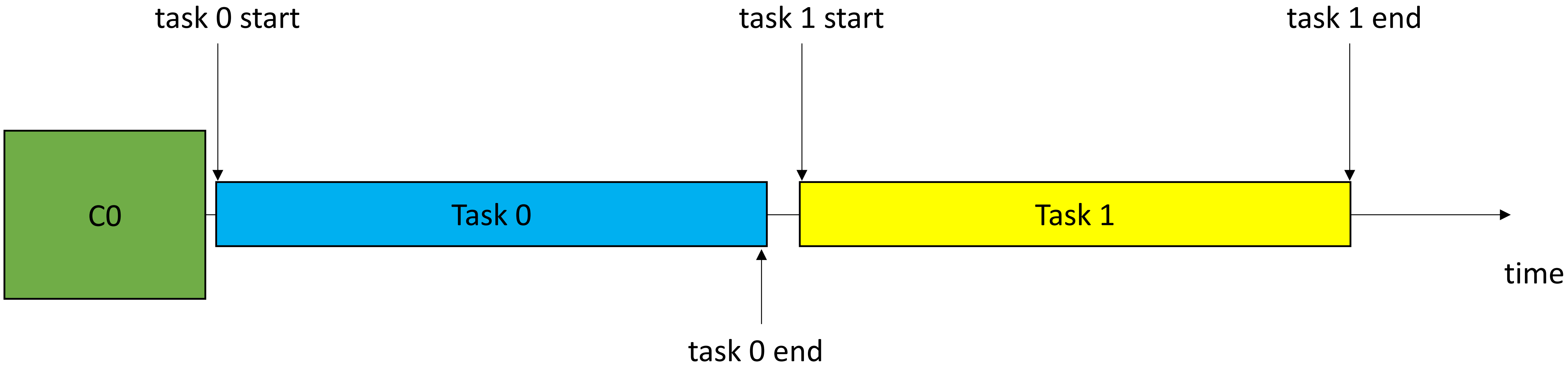
Concurrency vs. Parallelism

- Definition:
  - An execution is **parallel** if there is a point in the execution where computation is happening simultaneously



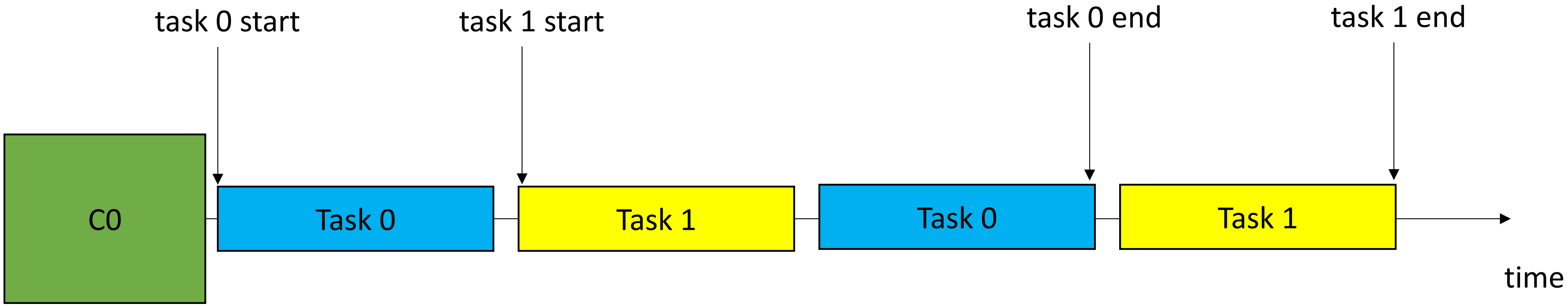
Concurrency vs. Parallelism

- Examples:
  - Neither concurrent or parallel (sequential)



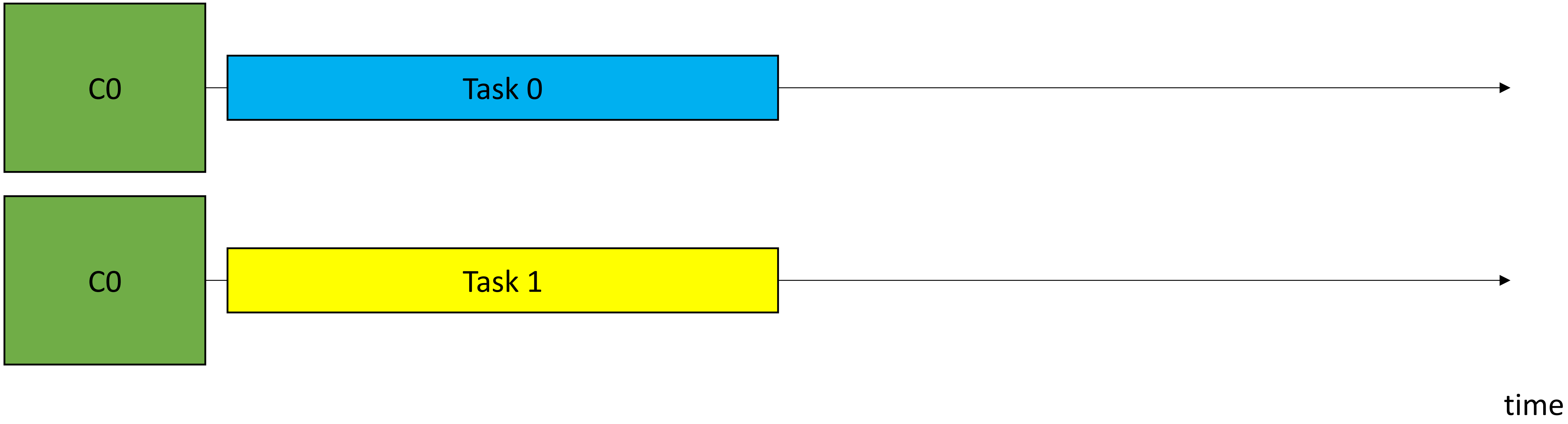
Concurrency vs. Parallelism

- Examples:
  - Concurrent but not parallel



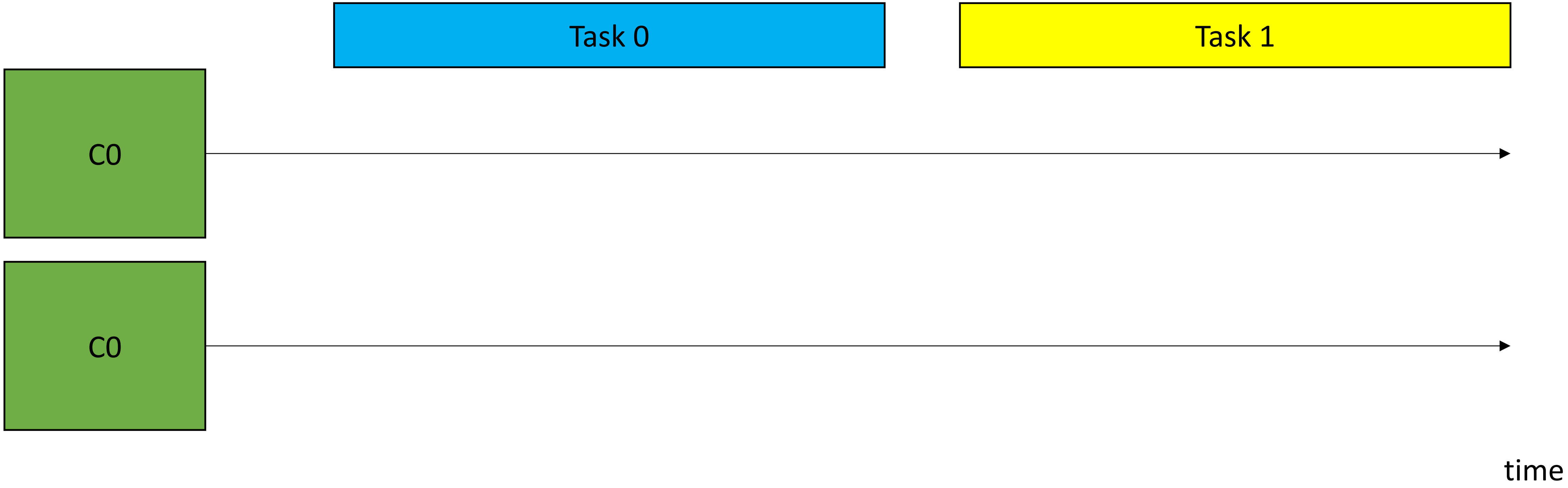
Concurrency vs. Parallelism

- Examples:
  - Parallel and Concurrent



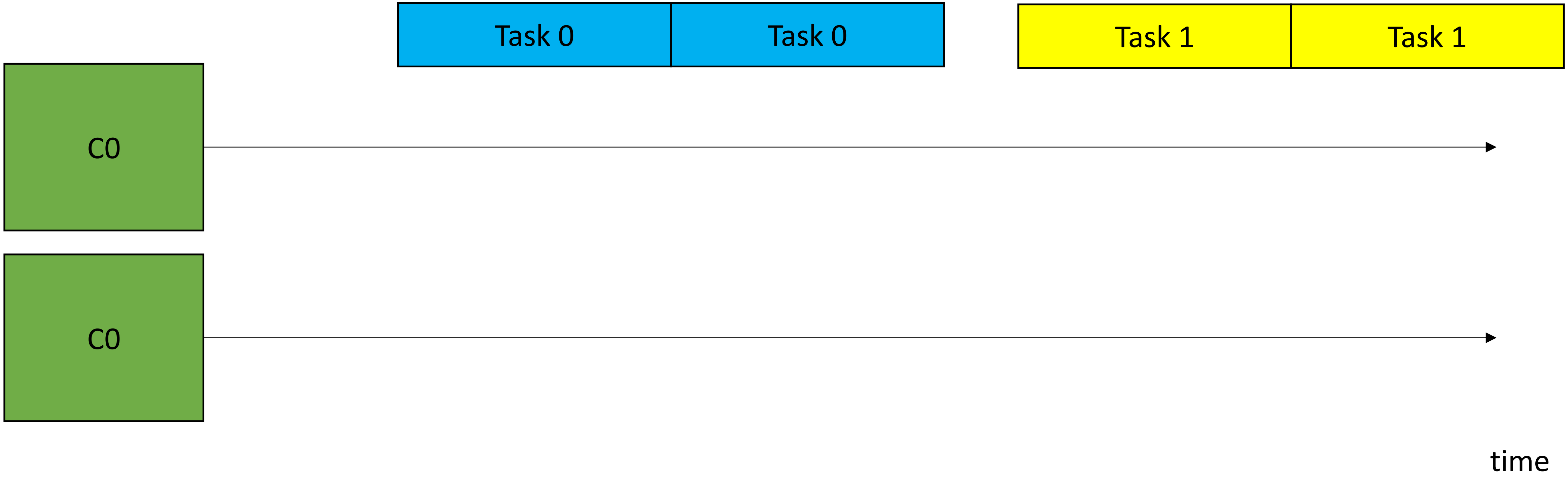
Concurrency vs. Parallelism

- Examples:
  - Parallel but not concurrent?



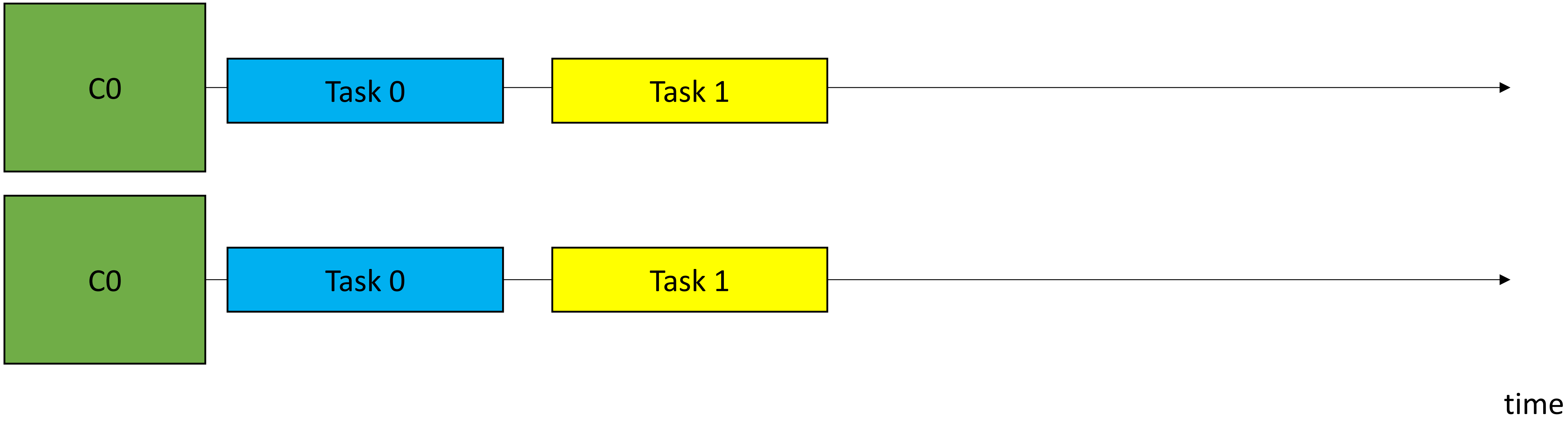
Concurrency vs. Parallelism

- Examples:
  - Parallel but not concurrent?



Concurrency vs. Parallelism

- Examples:
  - Parallel execution but task 0 and task 1 are not concurrent?



Concurrency vs. Parallelism

- In practice:
  - Terms are often used interchangeably.
  - *Parallel programming* is often used by high performance engineers when discussing using parallelism to accelerate things
  - *Concurrent programming* is used more by interactive applications, e.g. event driven interfaces.

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***.

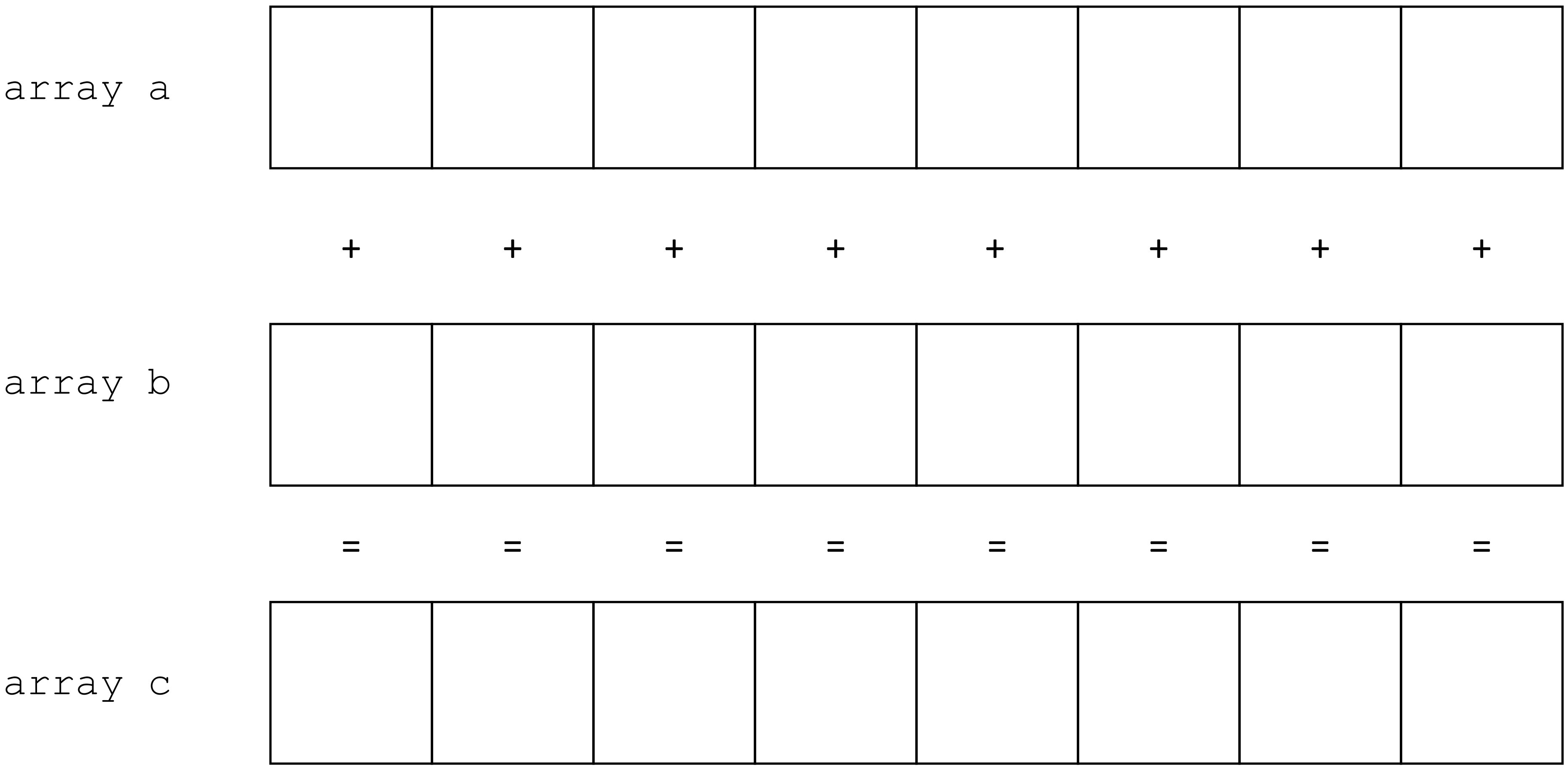
Embarrassingly parallel

- Consider the following program:

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

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

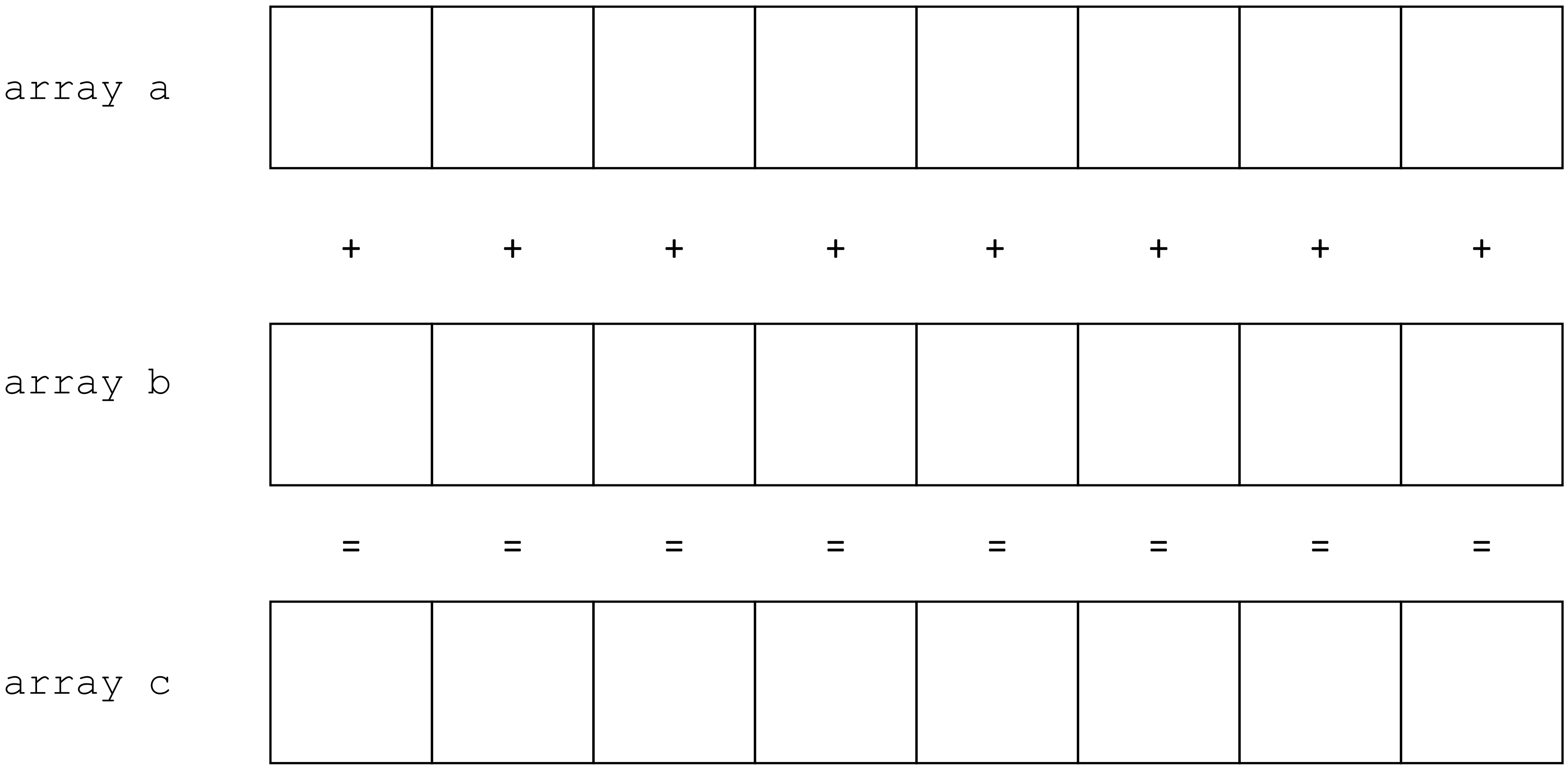
Embarrassingly parallel



Embarrassingly parallel

Computation can easily be divided into threads

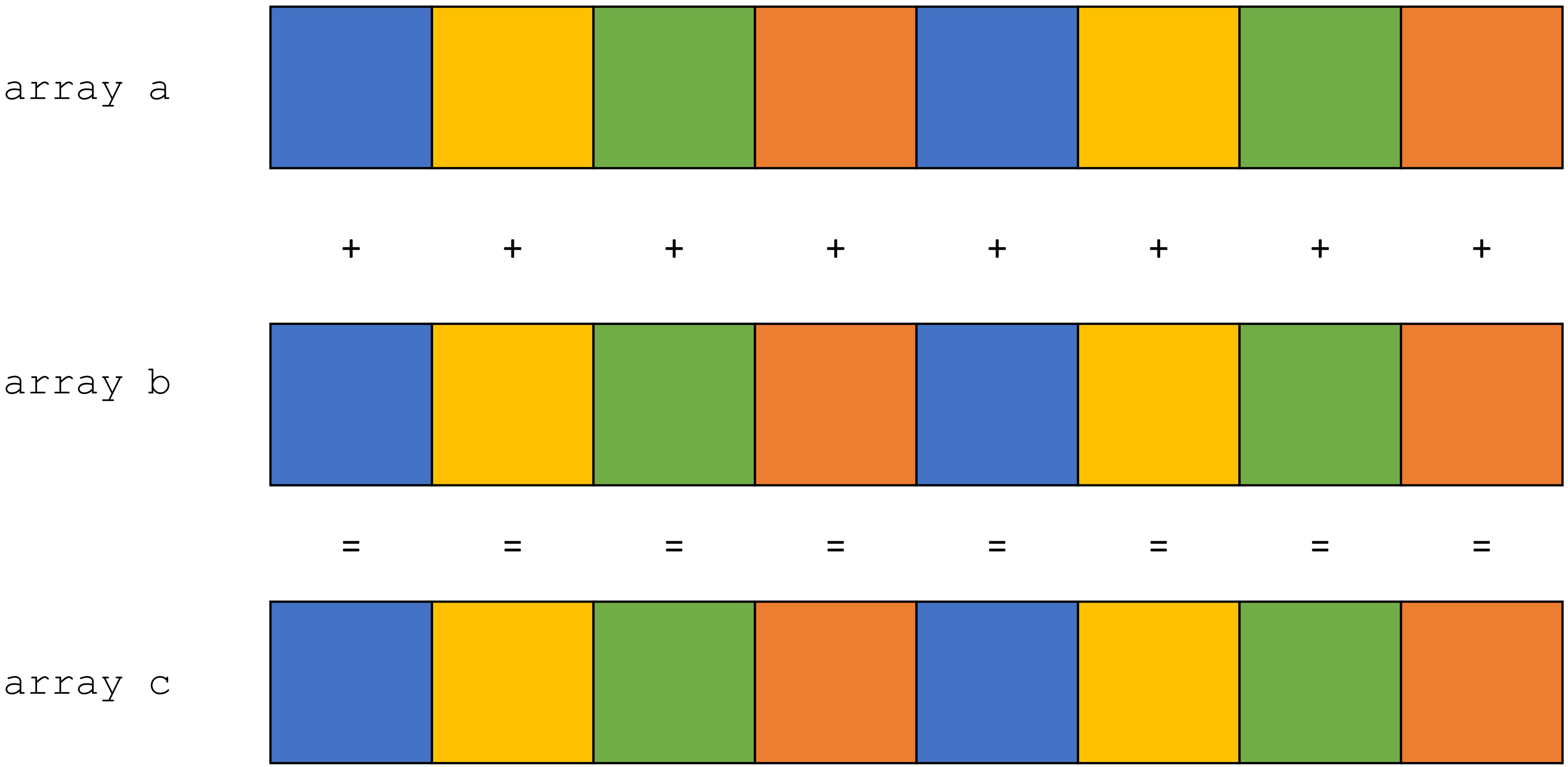
Thread 0 - Blue
Thread 1 - Yellow
Thread 2 - Green
Thread 3 - Orange



Embarrassingly parallel

Computation can easily be divided into threads

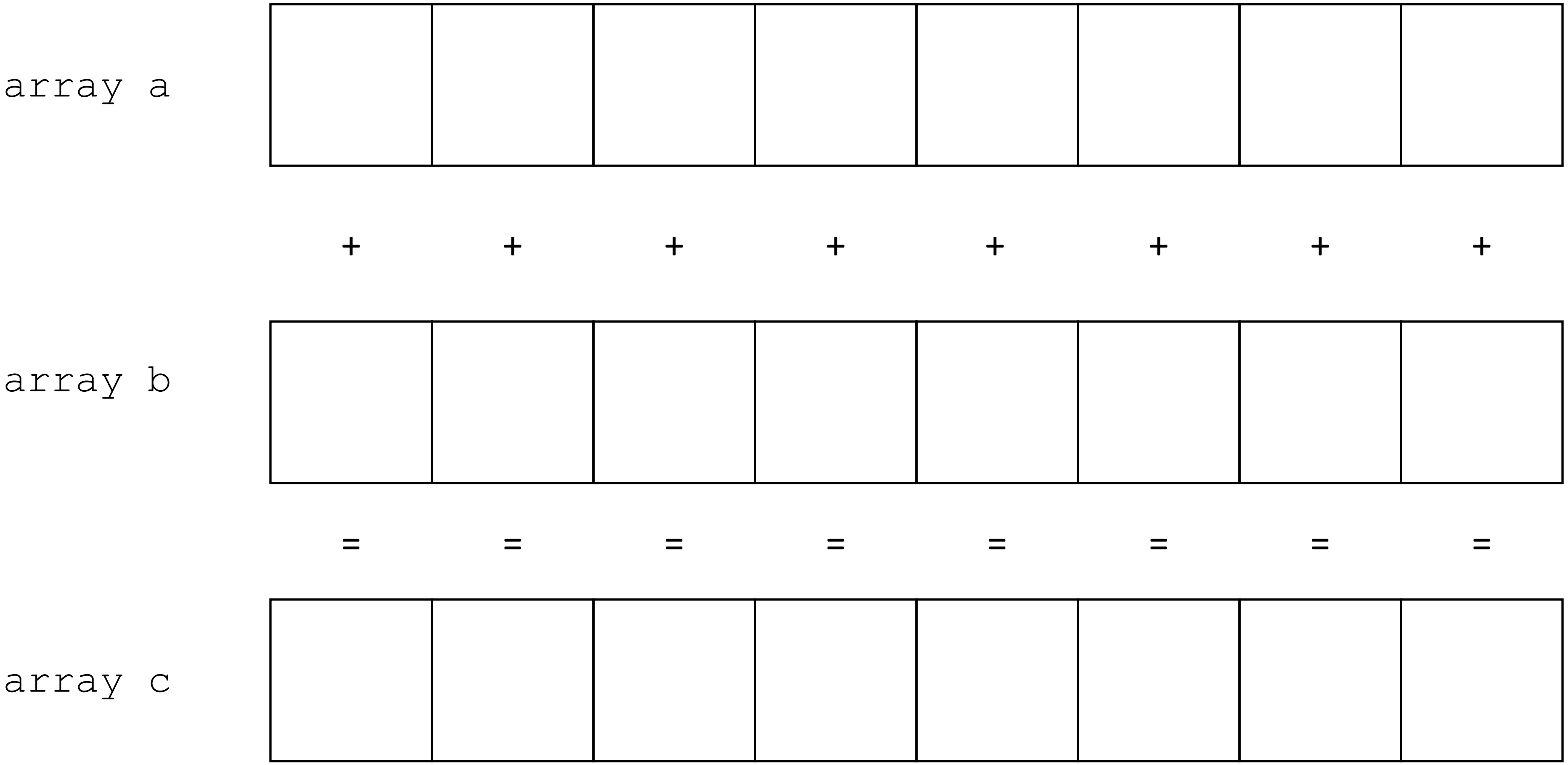
Thread 0 - Blue
Thread 1 - Yellow
Thread 2 - Green
Thread 3 - Orange



Embarrassingly parallel

Computation can easily be divided into threads

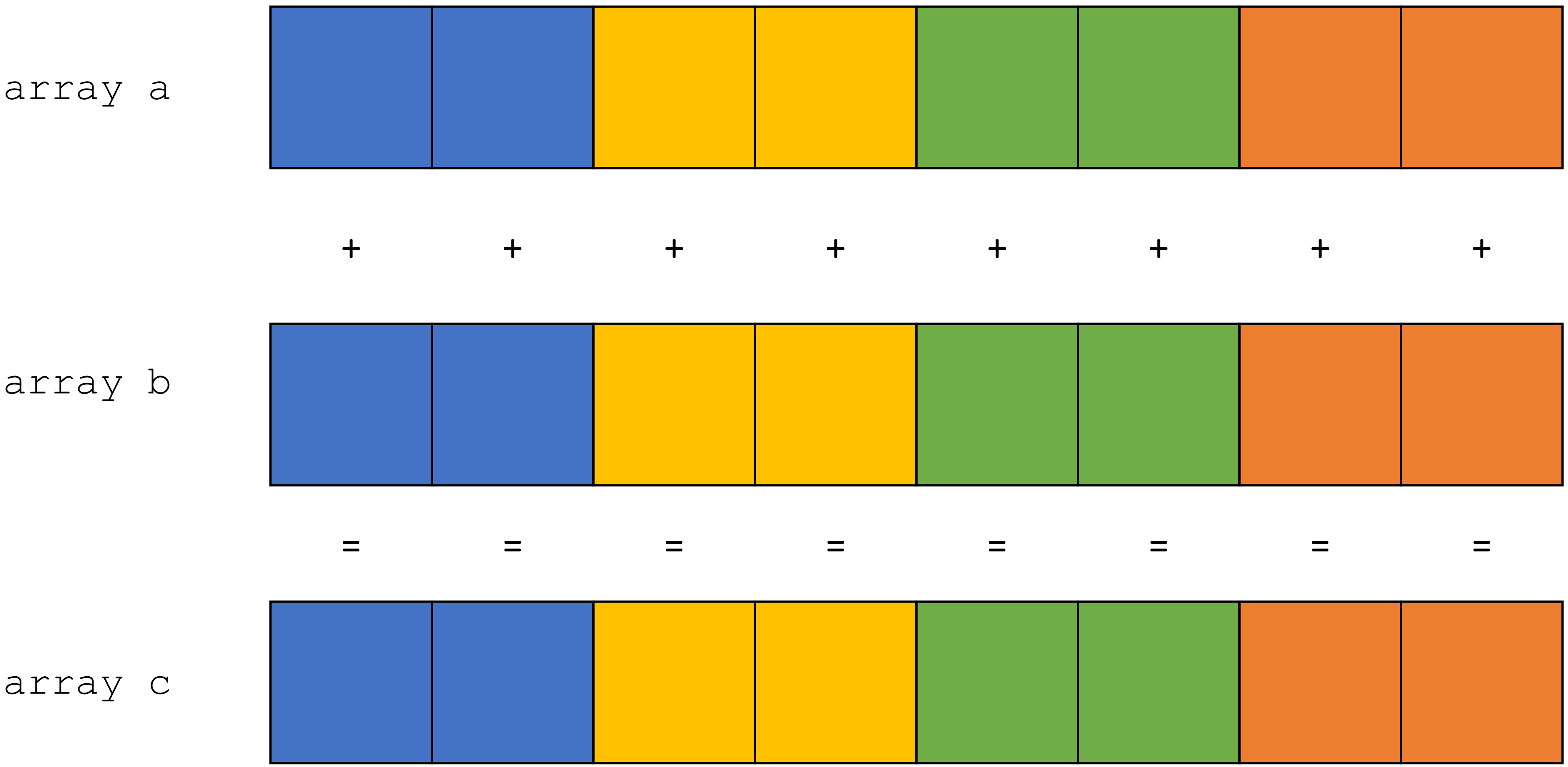
Thread 0 - Blue
Thread 1 - Yellow
Thread 2 - Green
Thread 3 - Orange



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

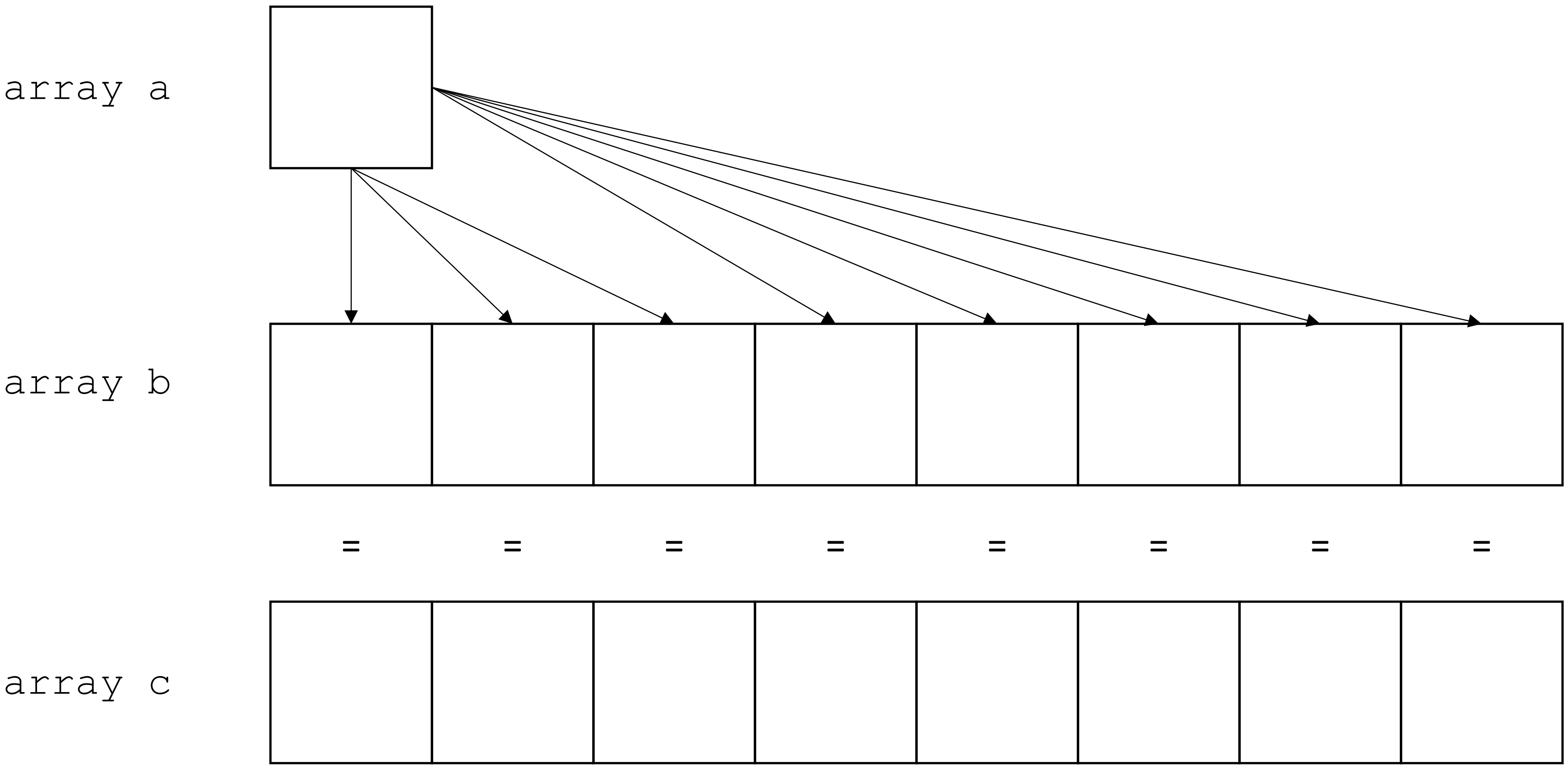
- Next Program

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

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

Embarrassingly parallel

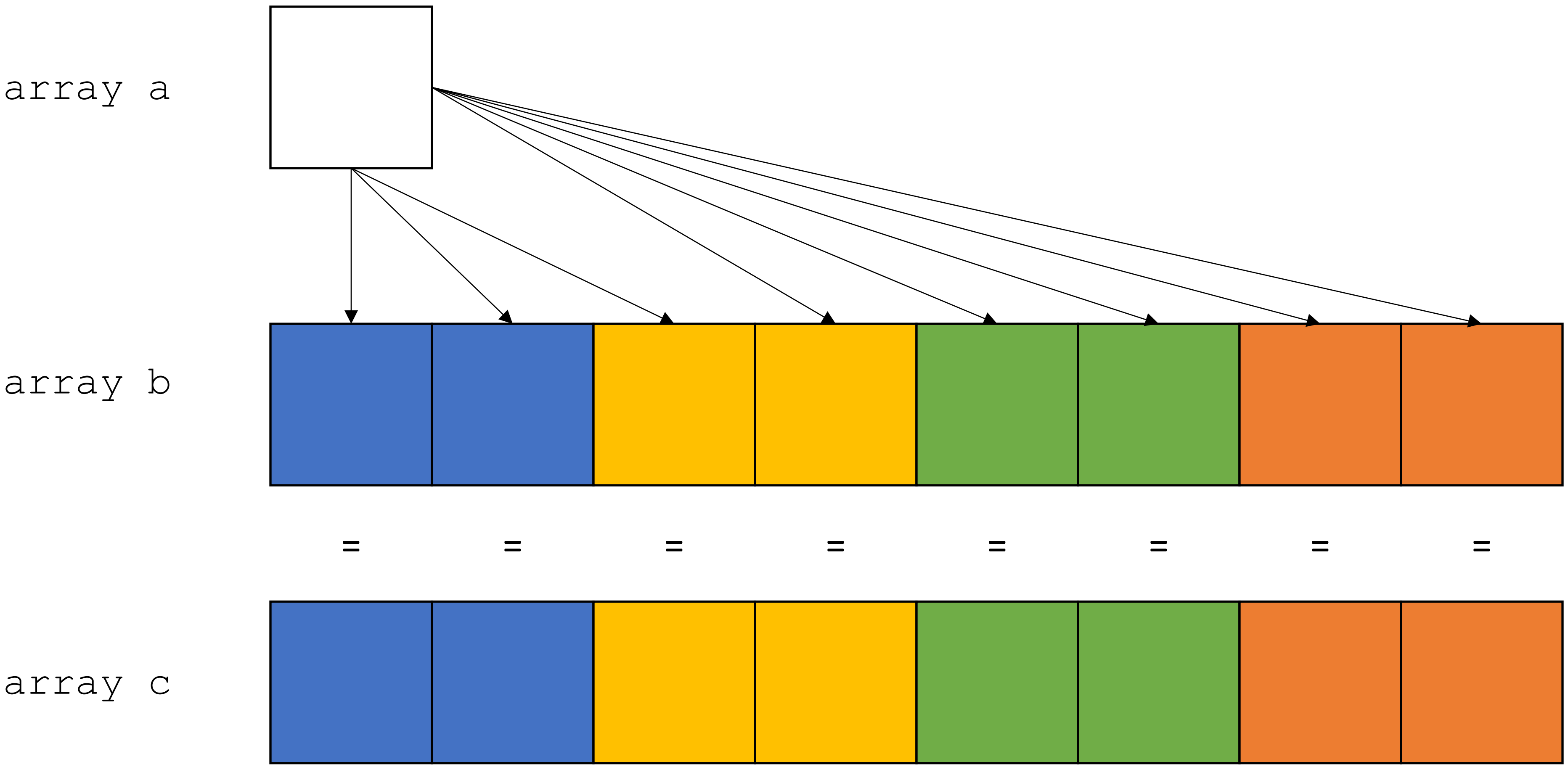
Thread 0 - Blue
Thread 1 - Yellow
Thread 2 - Green
Thread 3 - Orange



is this problem embarrassingly parallel?

Embarrassingly parallel

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

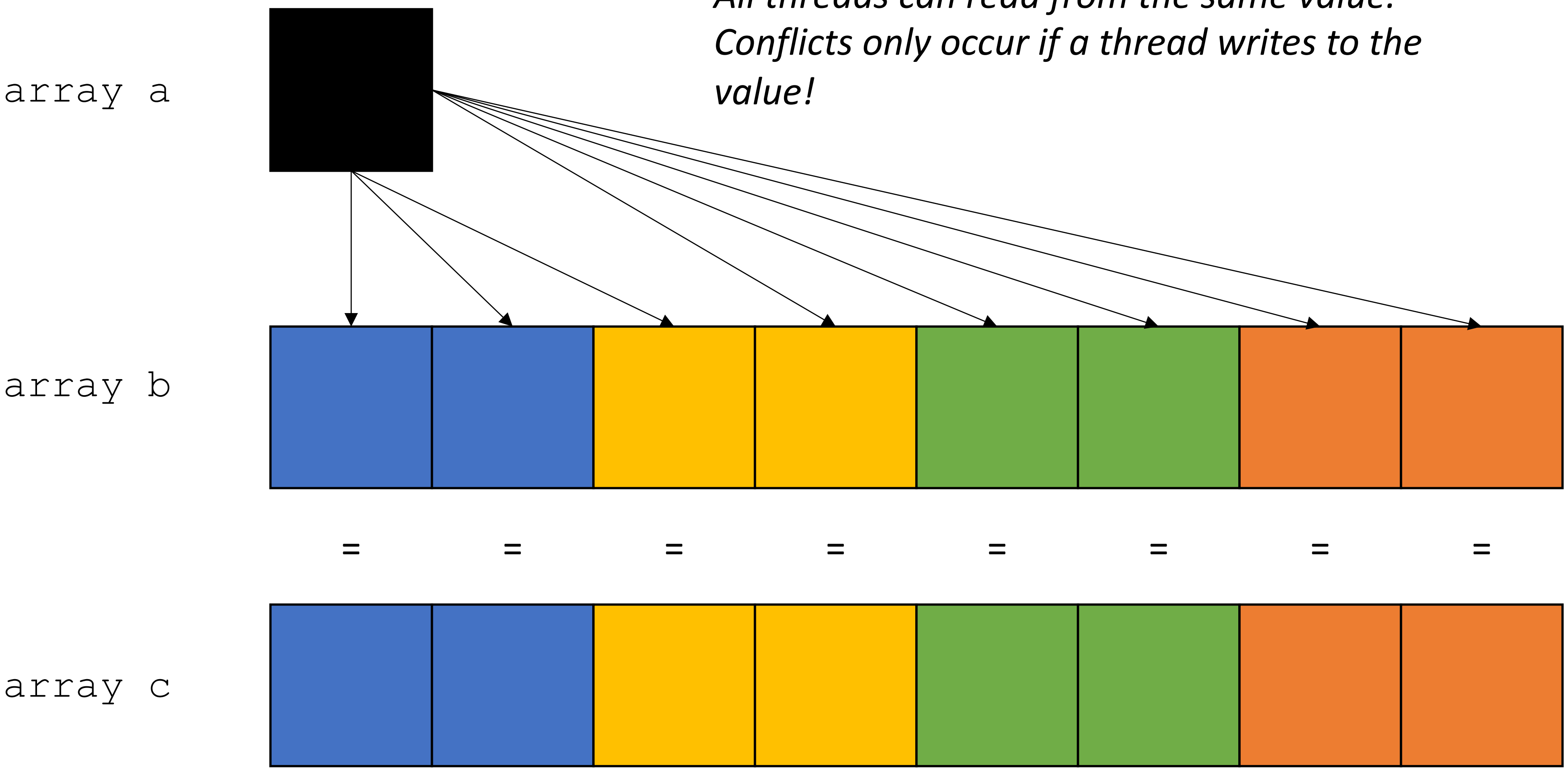


is this problem embarrassingly parallel?

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

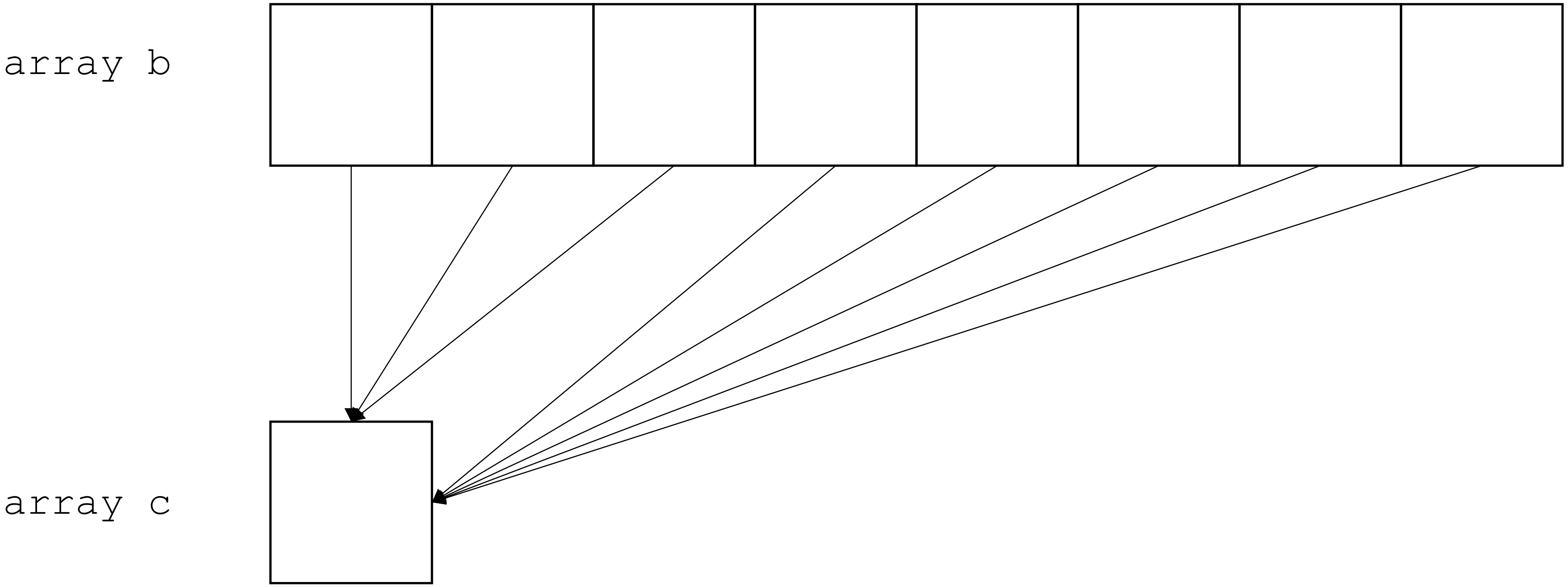
- Next Program

There are 2 arrays: `b, c`

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

Embarrassingly parallel

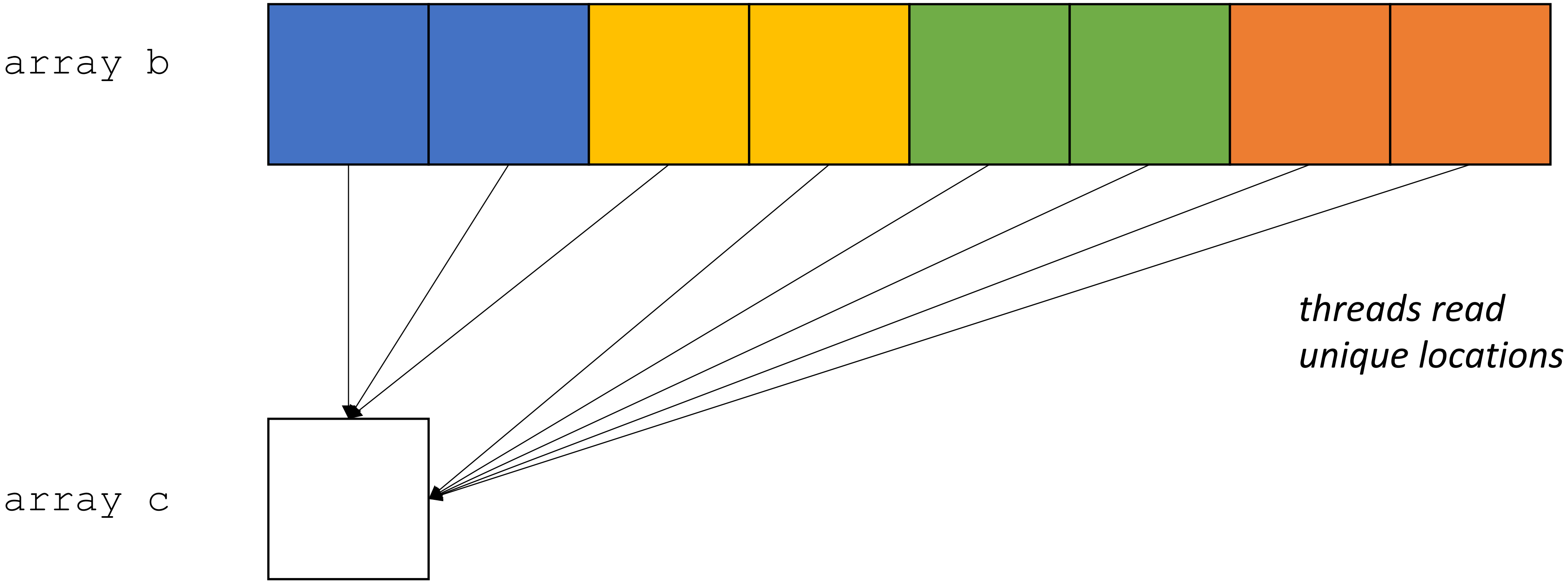
Thread 0 - Blue
Thread 1 - Yellow
Thread 2 - Green
Thread 3 - Orange



is this problem embarrassingly parallel?

Embarrassingly parallel

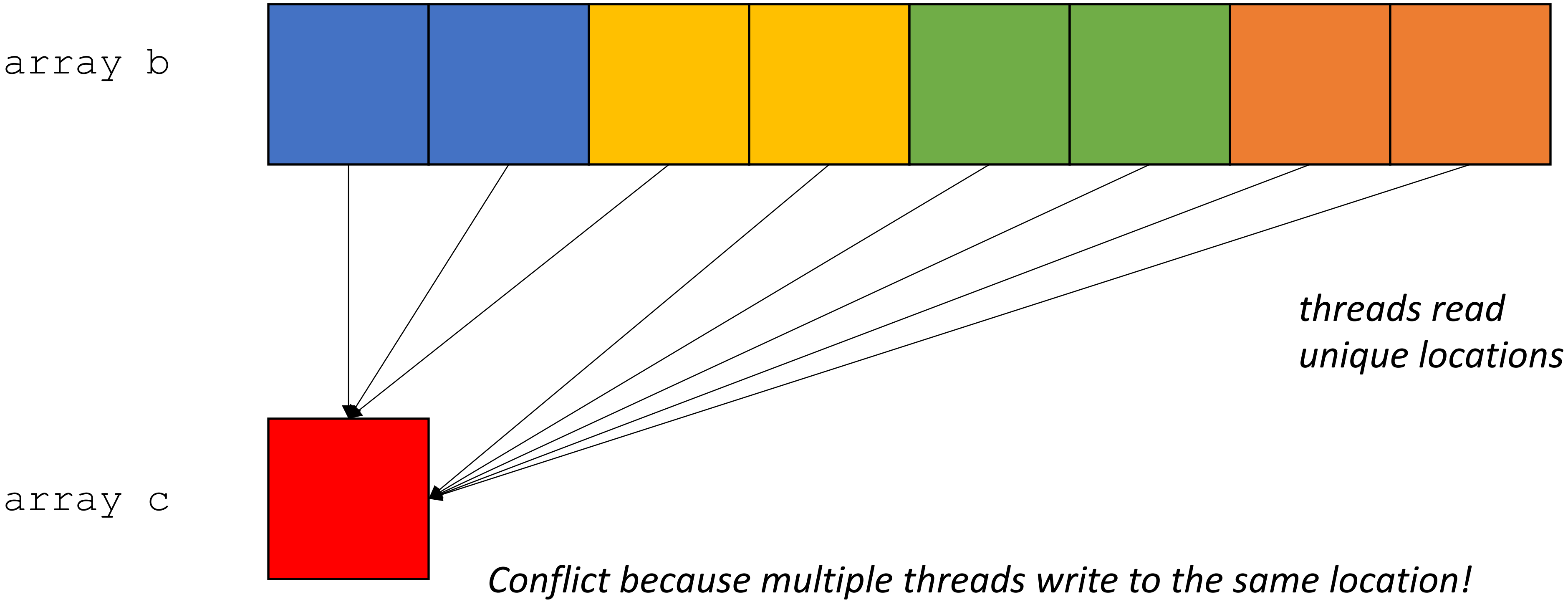
Thread 0 - Blue
Thread 1 - Yellow
Thread 2 - Green
Thread 3 - Orange



is this problem embarrassingly parallel?

Embarrassingly parallel

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



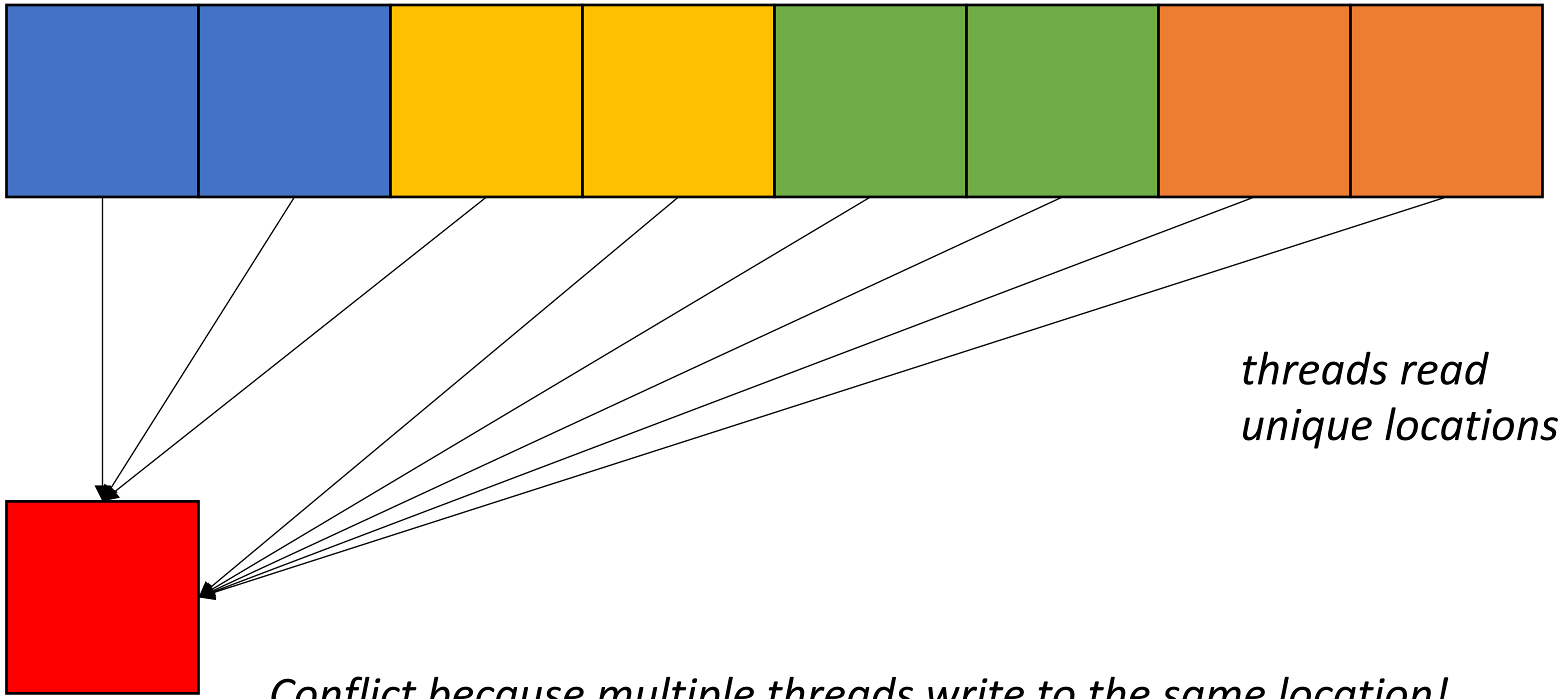
is this problem embarrassingly parallel?

Embarrassingly parallel

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

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?

Conflict because multiple threads write to the same location!

We need a way how to safely share memory

- *Many 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

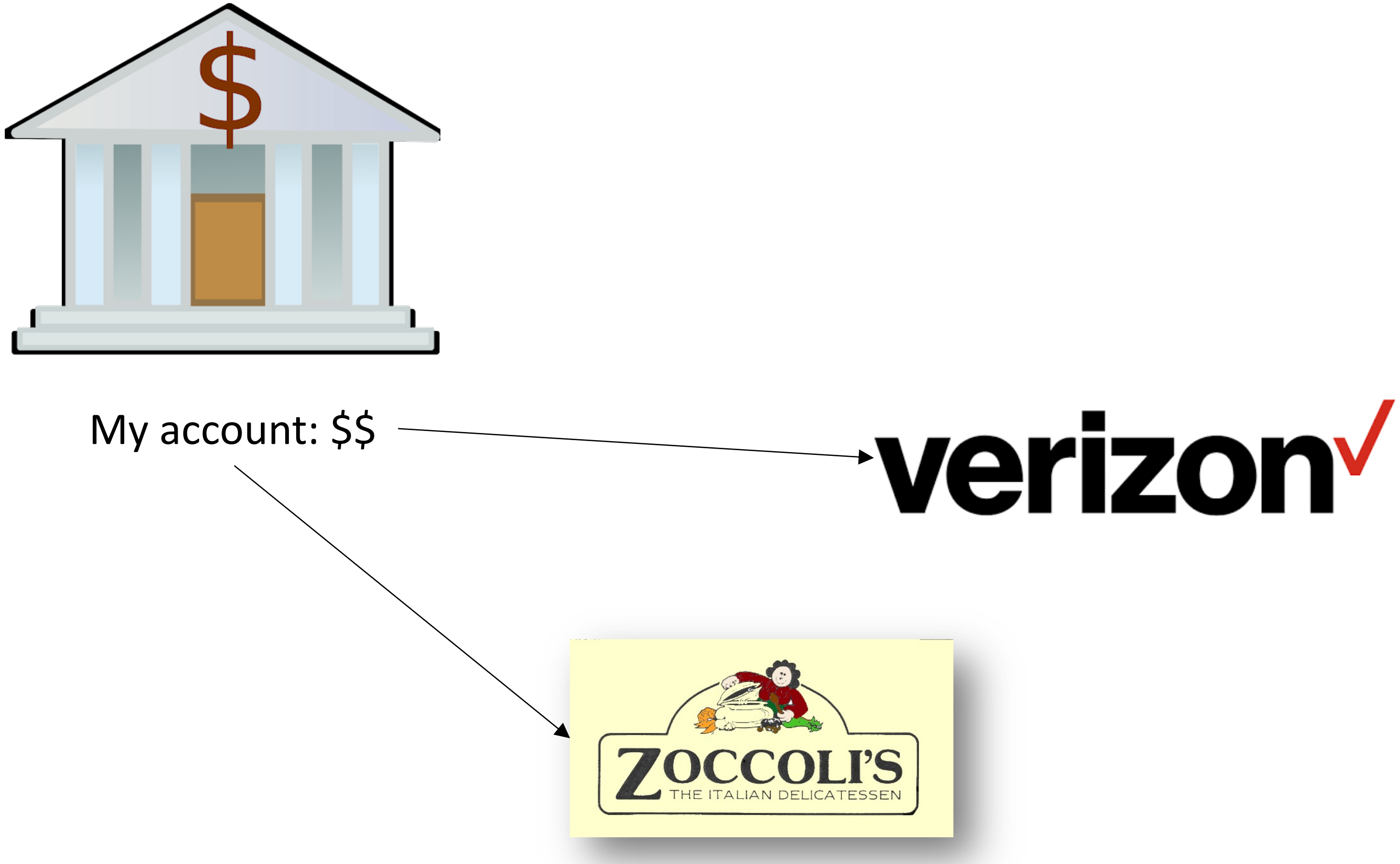
- Bank

My account: $$

ZOCCOLI'S
THE ITALIAN DELICATESSEN

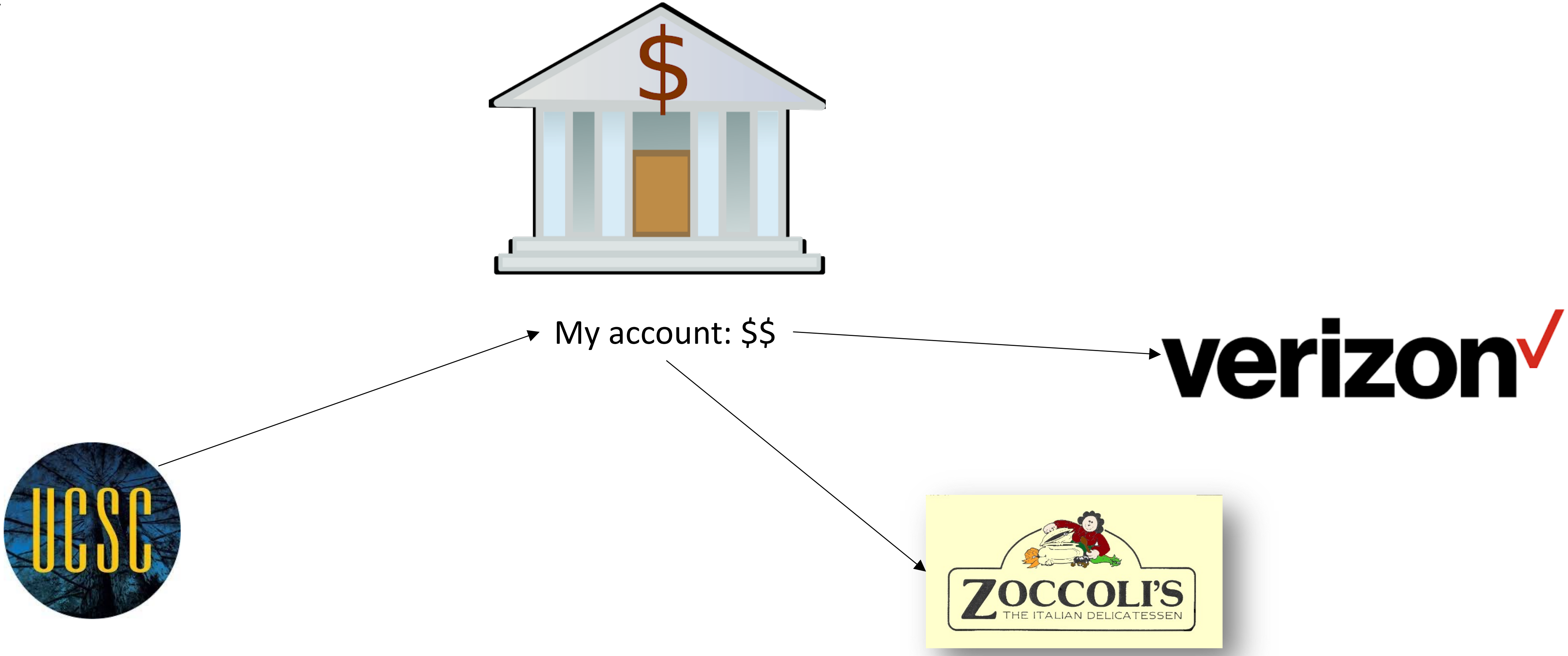
We need a way how to safely share memory

- Bank



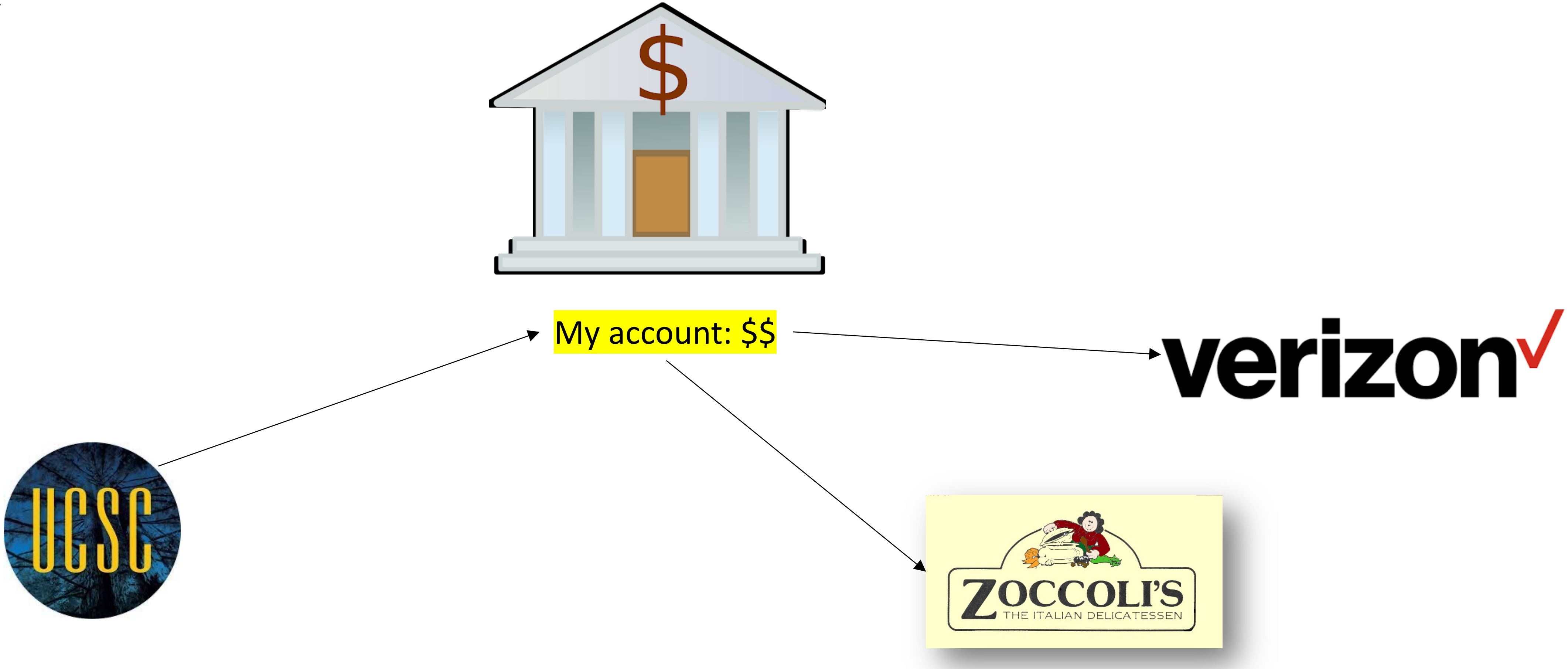
We need a way how to safely share memory

- Bank



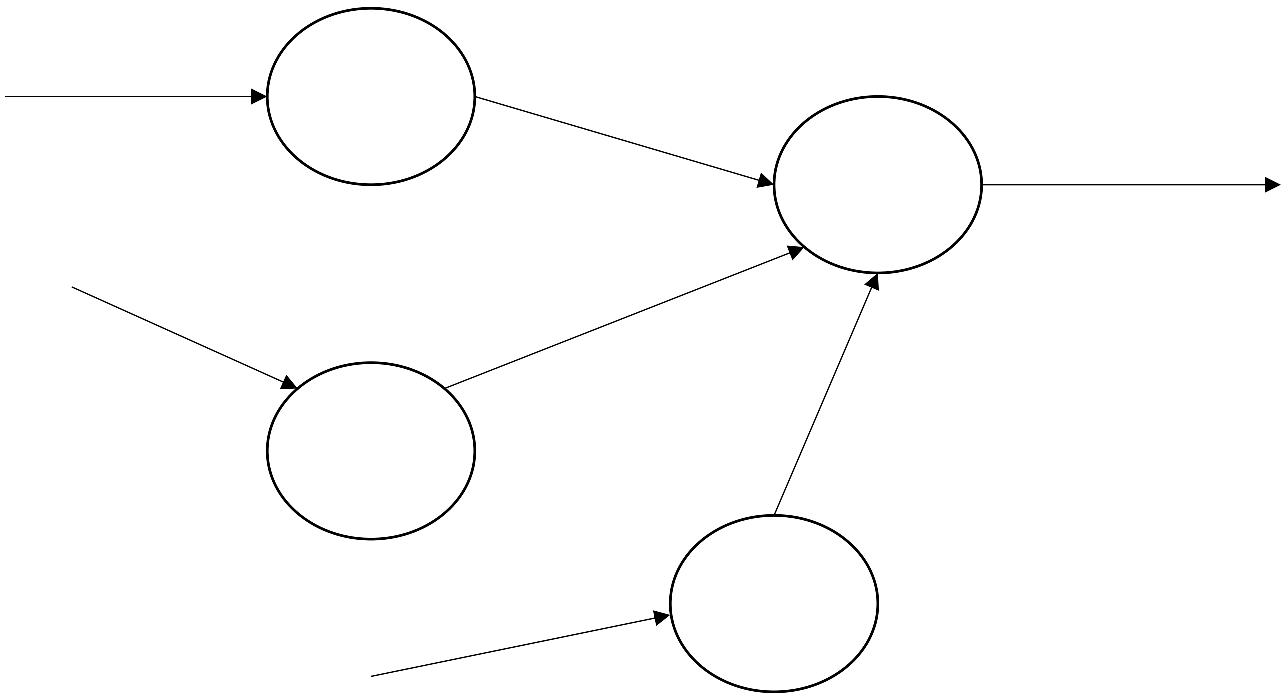
We need a way how to safely share memory

- Bank



We need a way how to safely share memory

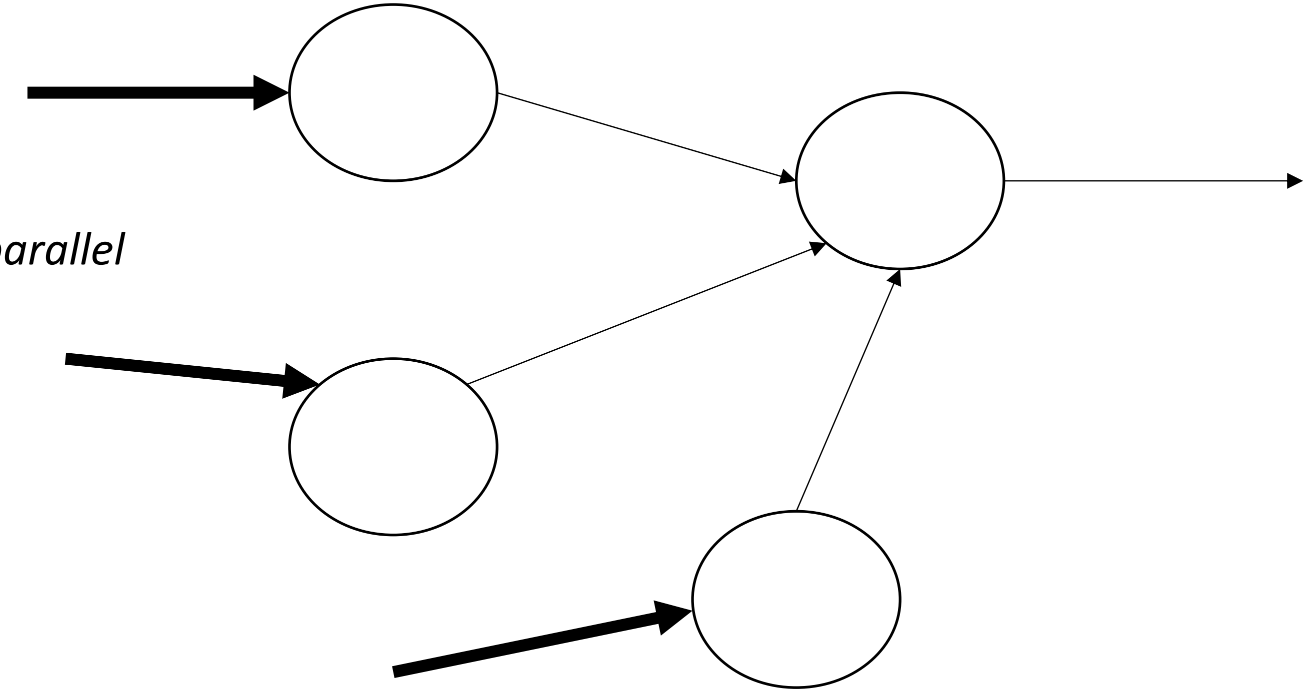
- Graph algorithms



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

We need a way how to safely share memory

- Graph algorithms

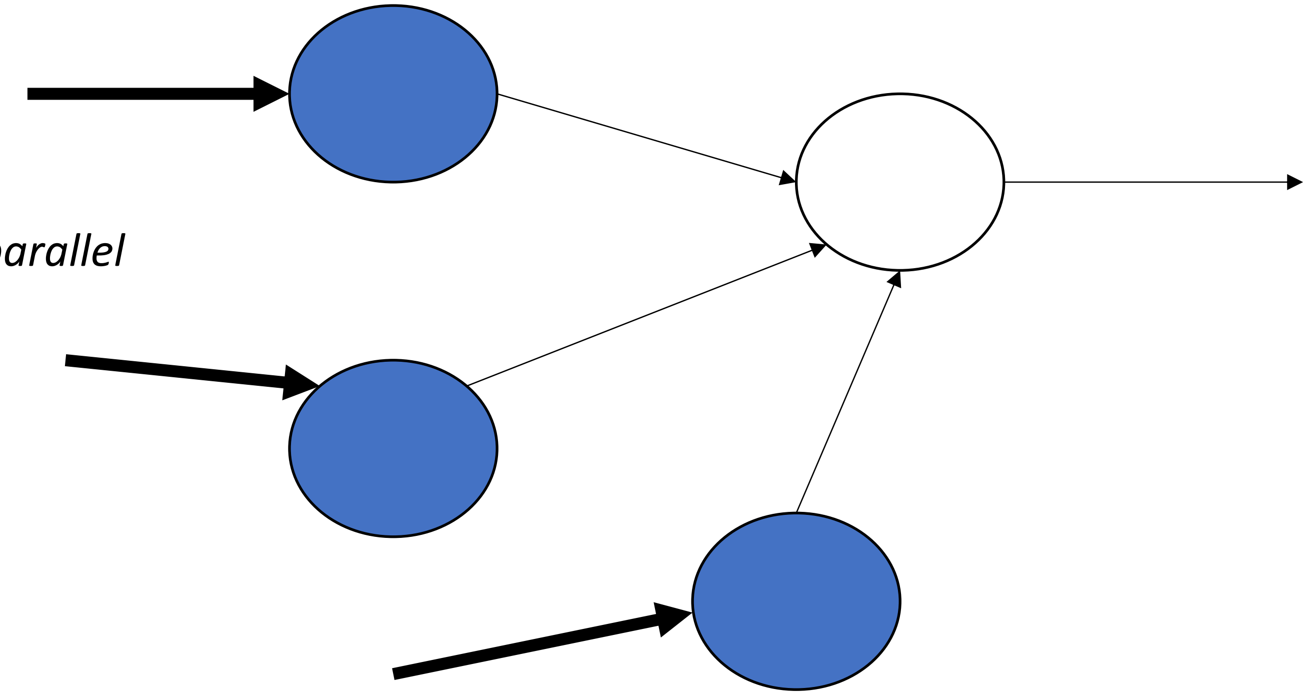


these can be done in parallel

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

We need a way how to safely share memory

- Graph algorithms



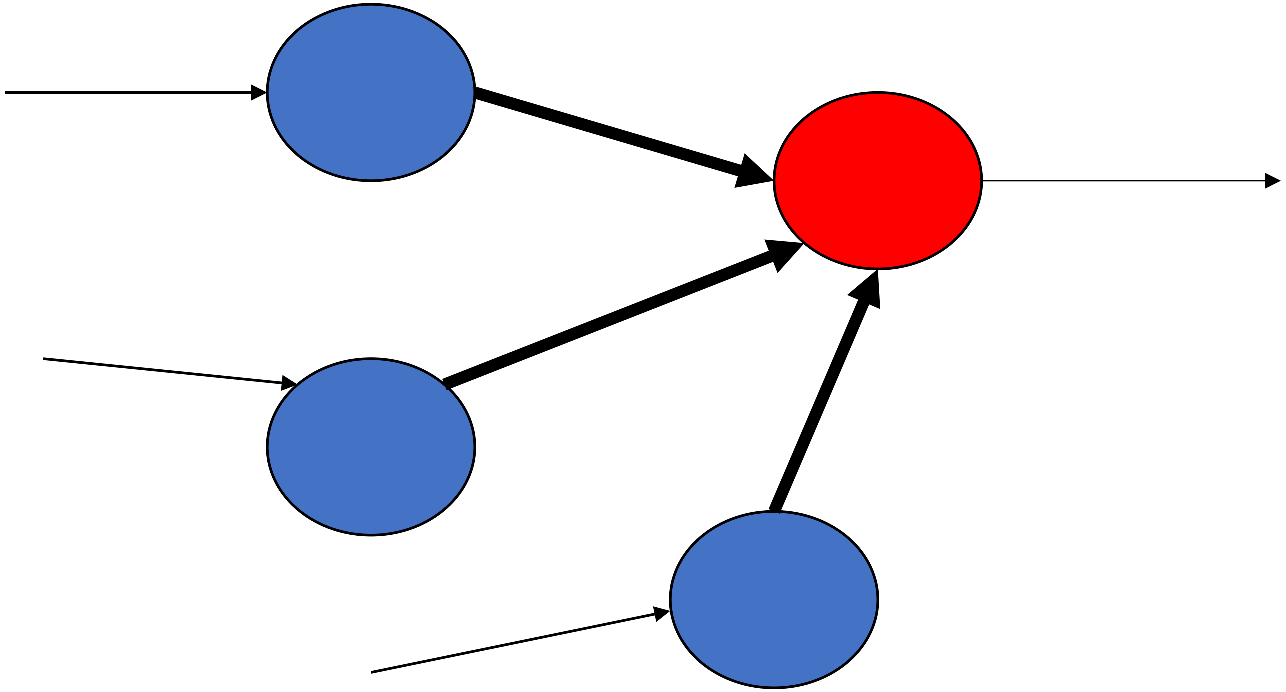
these can be done in parallel

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

We need a way how to safely share memory

- Graph algorithms

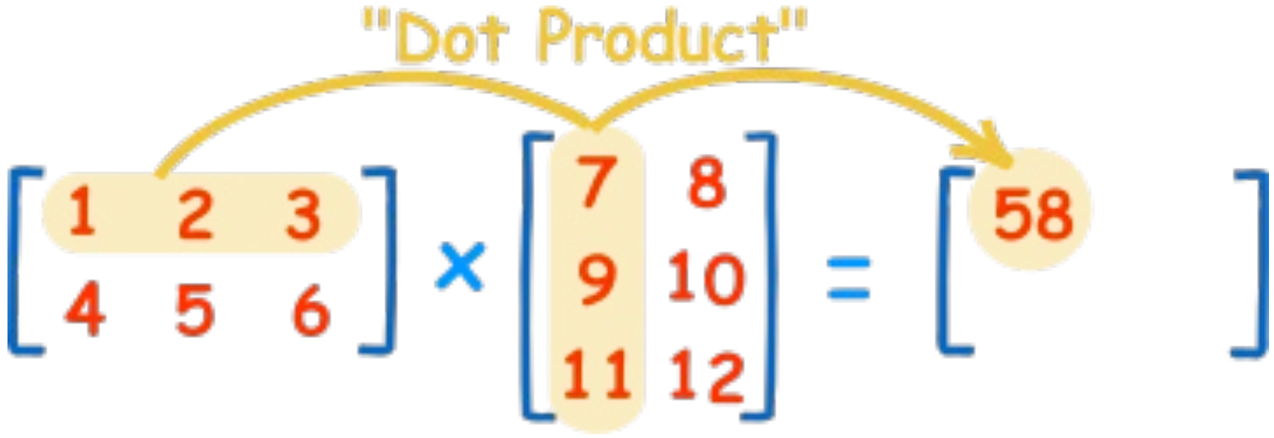
potential conflict if different threads access the red node



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

We need a way how to safely share memory

- Machine Learning



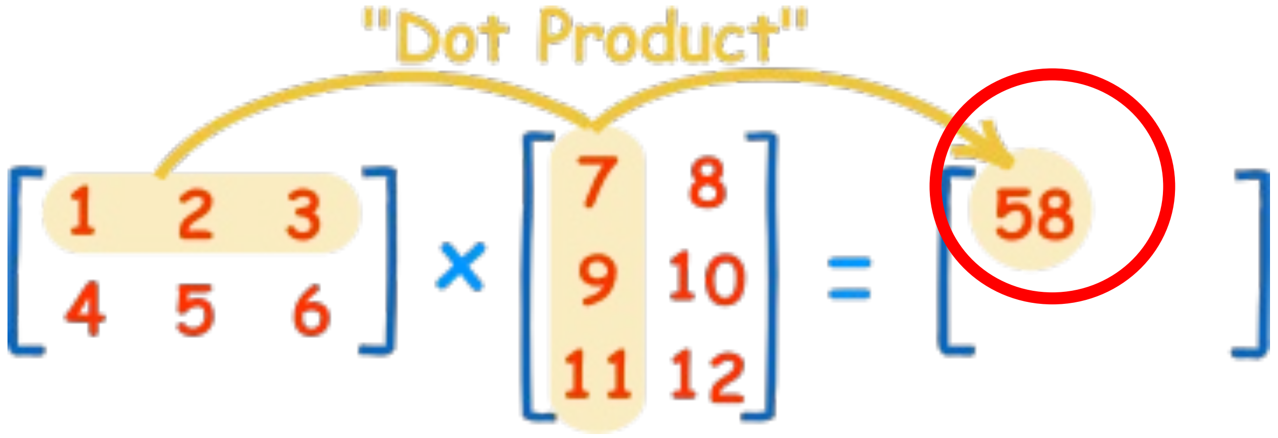
Lots of machine learning is some form of matrix multiplication

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

We need a way how to safely share memory

- Machine Learning

conflict!

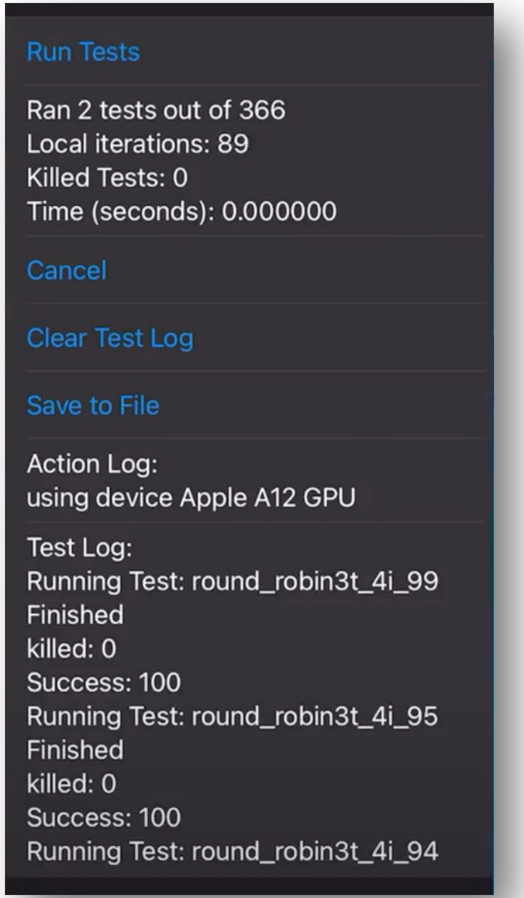


Lots of machine learning is some form of matrix multiplication

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

We need a way how to safely share **resources**

- User interfaces



background process that provides progress updates to the UI.

UI updates must be synchronized!!

https://drive.google.com/file/d/1JVQTQsrKhpksgVAM1yaMQkyohfDtWsSI/view?usp=sharing

Dangers of conflicts

- We will illustrate using a running bank account example

Sequential bank scenario

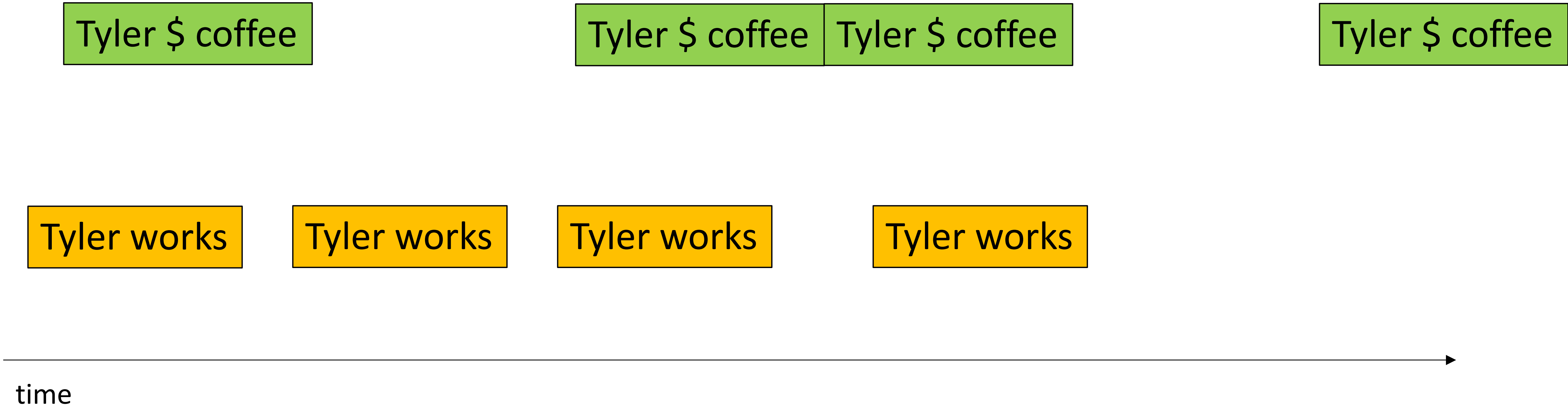
- UCSC deposits $1 in my bank account after every hour I work.
- I buy a cup of coffee ($1) after each hour I work.
- I work 1M hours (which is actually true).
- *I should break even*
- C++ code

Concurrent bank scenario

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

Concurrent bank scenario

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



Code demo

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

<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;
}
```

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;
}
```

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;
}
```

```
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
```

time

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!

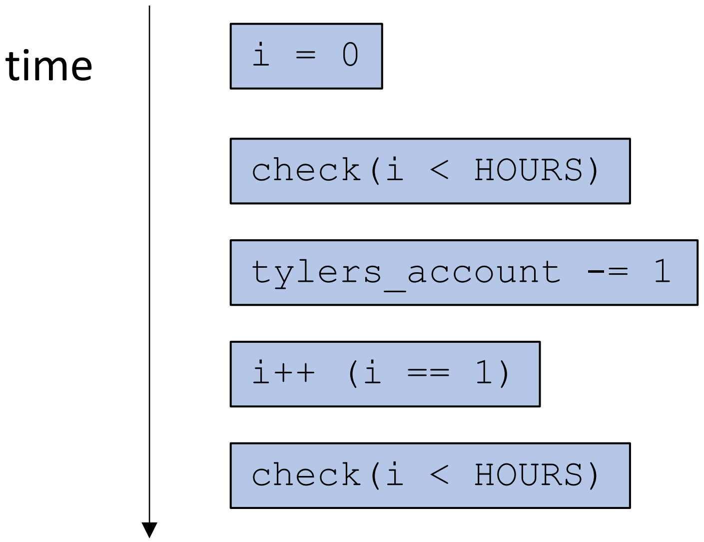
Tyler's employer

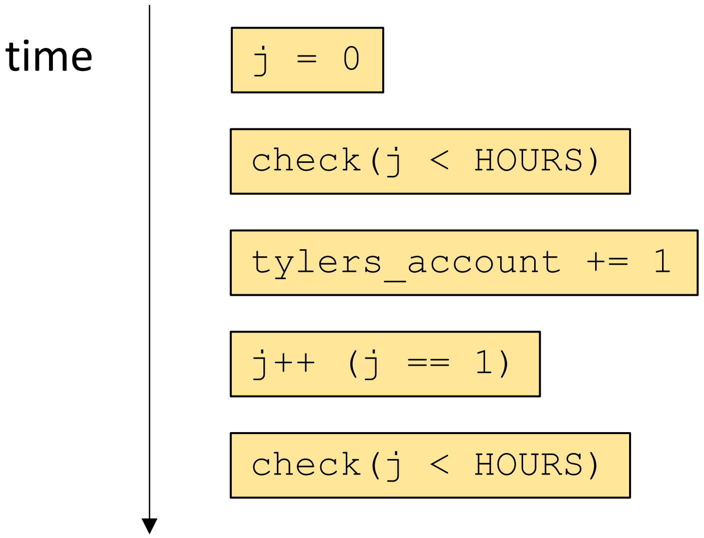
```
for (int j = 0; j < HOURS; j++) {
   tylers_account += 1;
}
```

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

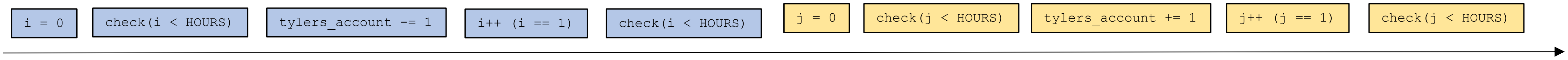




Concurrent execution

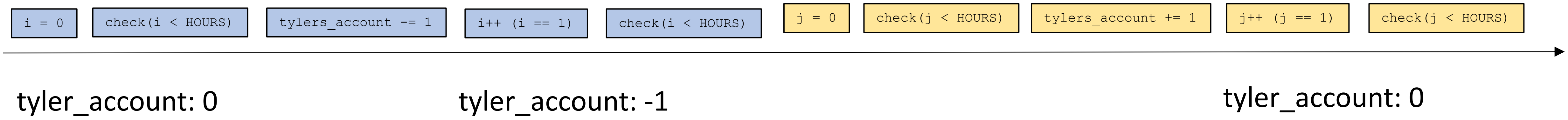
one possible execution

Concurrent execution



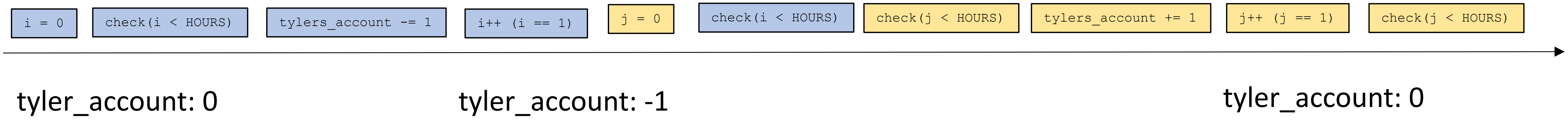
one possible execution

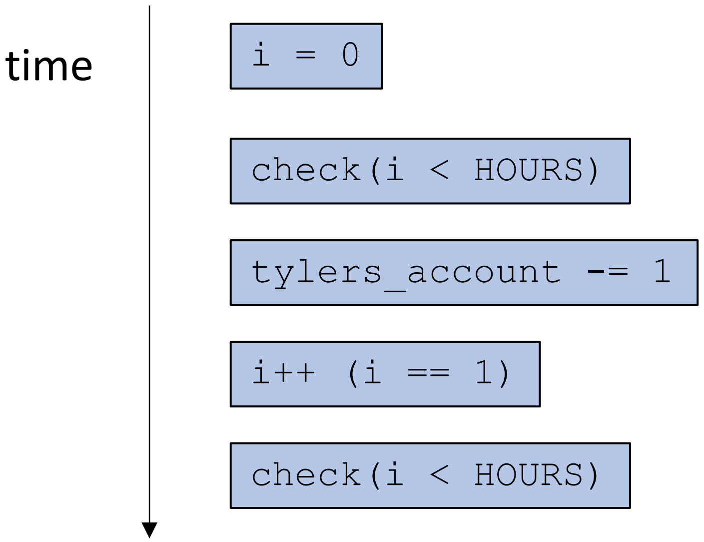
Concurrent execution

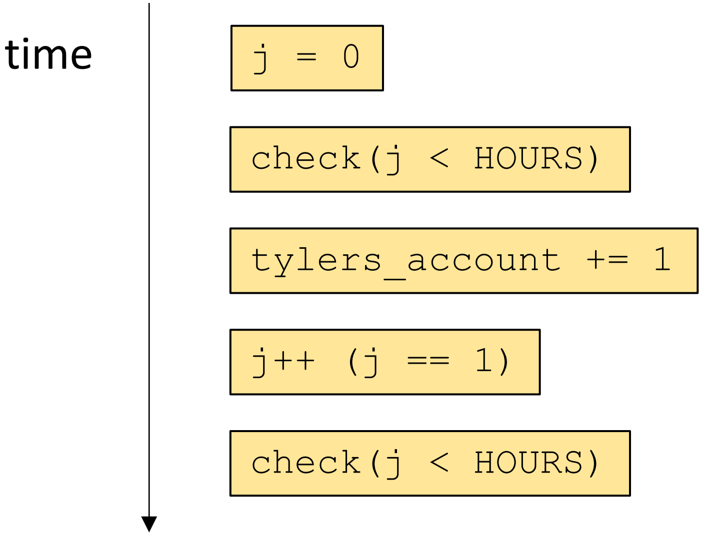


Another possible execution

Concurrent execution

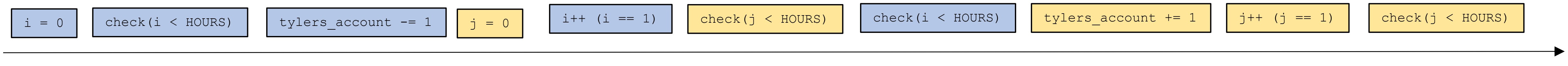






Another possible execution

Concurrent execution



tyler_account: 0

tyler_account: -1

tyler_account: 0

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)
```

Another possible execution

This time my account isn't ever negative

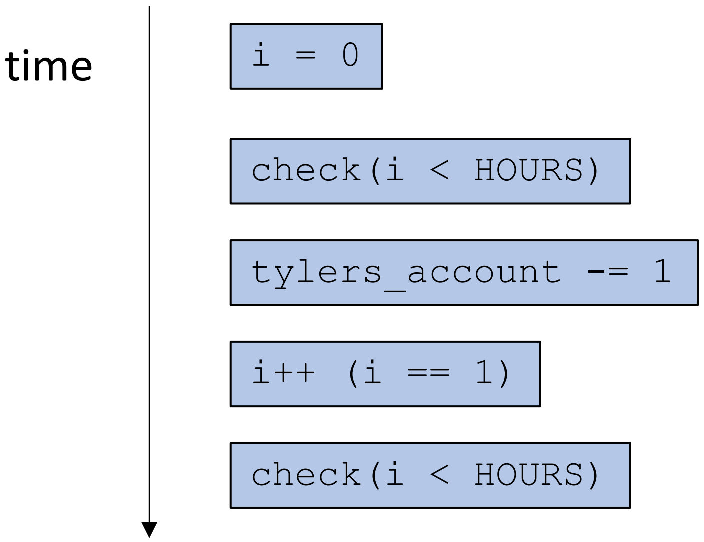
Concurrent execution

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

tyler_account: 0

tyler_account: 1

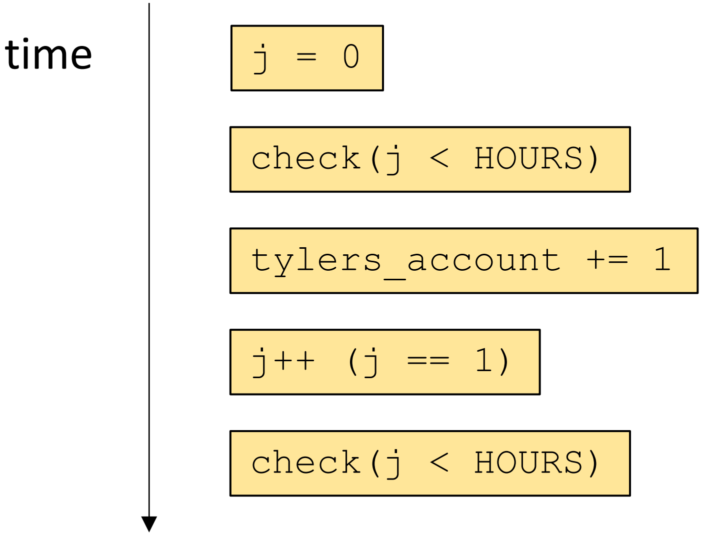
tyler_account: 0



How many possible interleavings?
Combinatorics question:

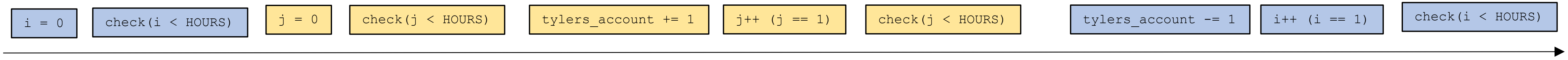
if Thread 0 has N events
if Thread 1 has M events

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



Concurrent execution

in our example there are 252 possible interleavings!



tyler_account: 0

tyler_account: 1

tyler_account: 0

Reasoning about concurrency

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

- Think about smaller instances of the problem

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

time

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

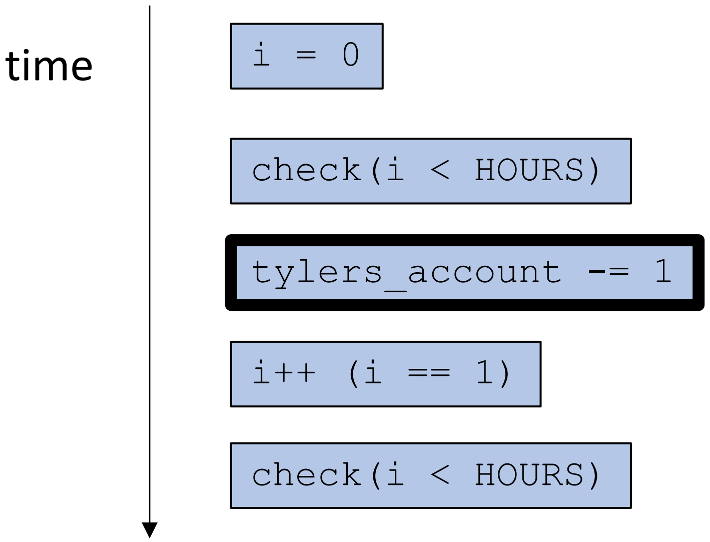
time

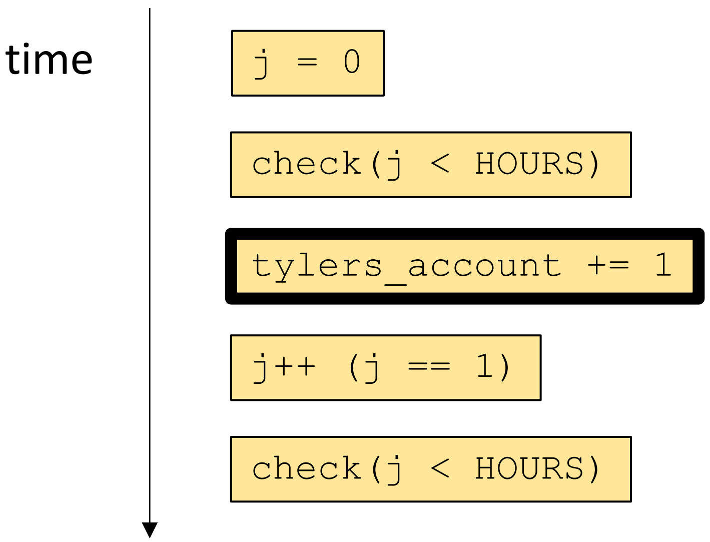
```
j = 0
check(j < HOURS)
tylers_account += 1
j++ (j == 1)
check(j < HOURS)
```

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





concurrent execution

time

time

`tylers_account -= 1`

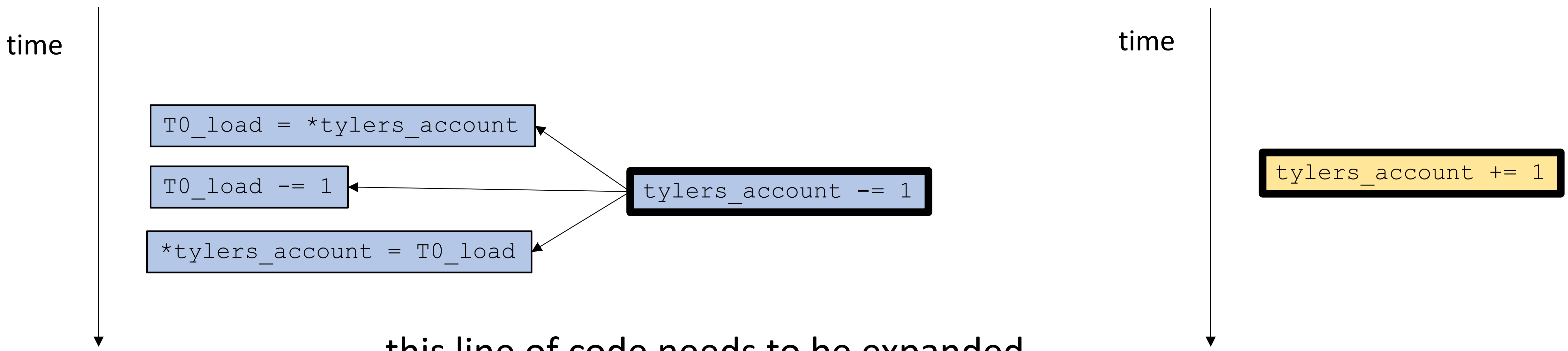
time

`tylers_account += 1`

Remember 3 address code...

concurrent execution

time

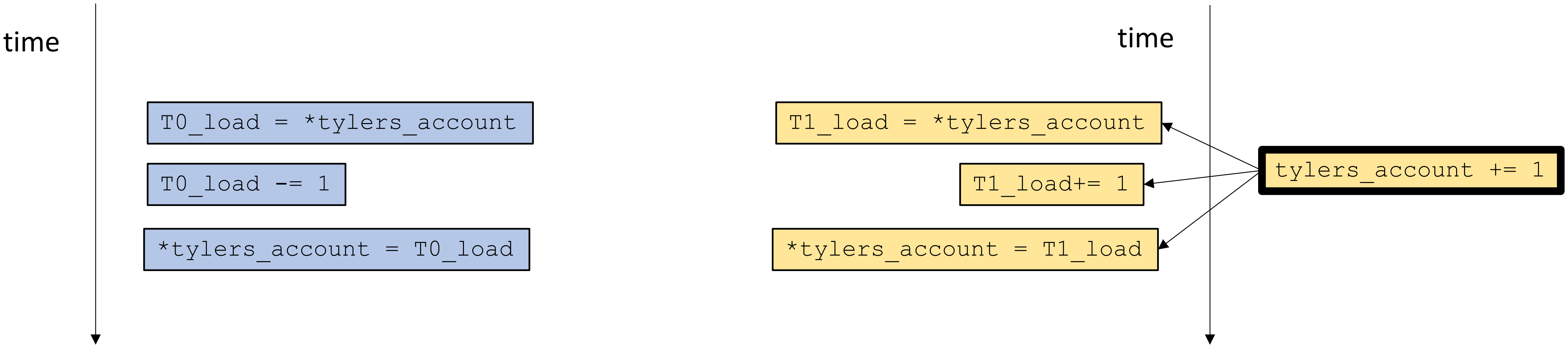


this line of code needs to be expanded

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
```

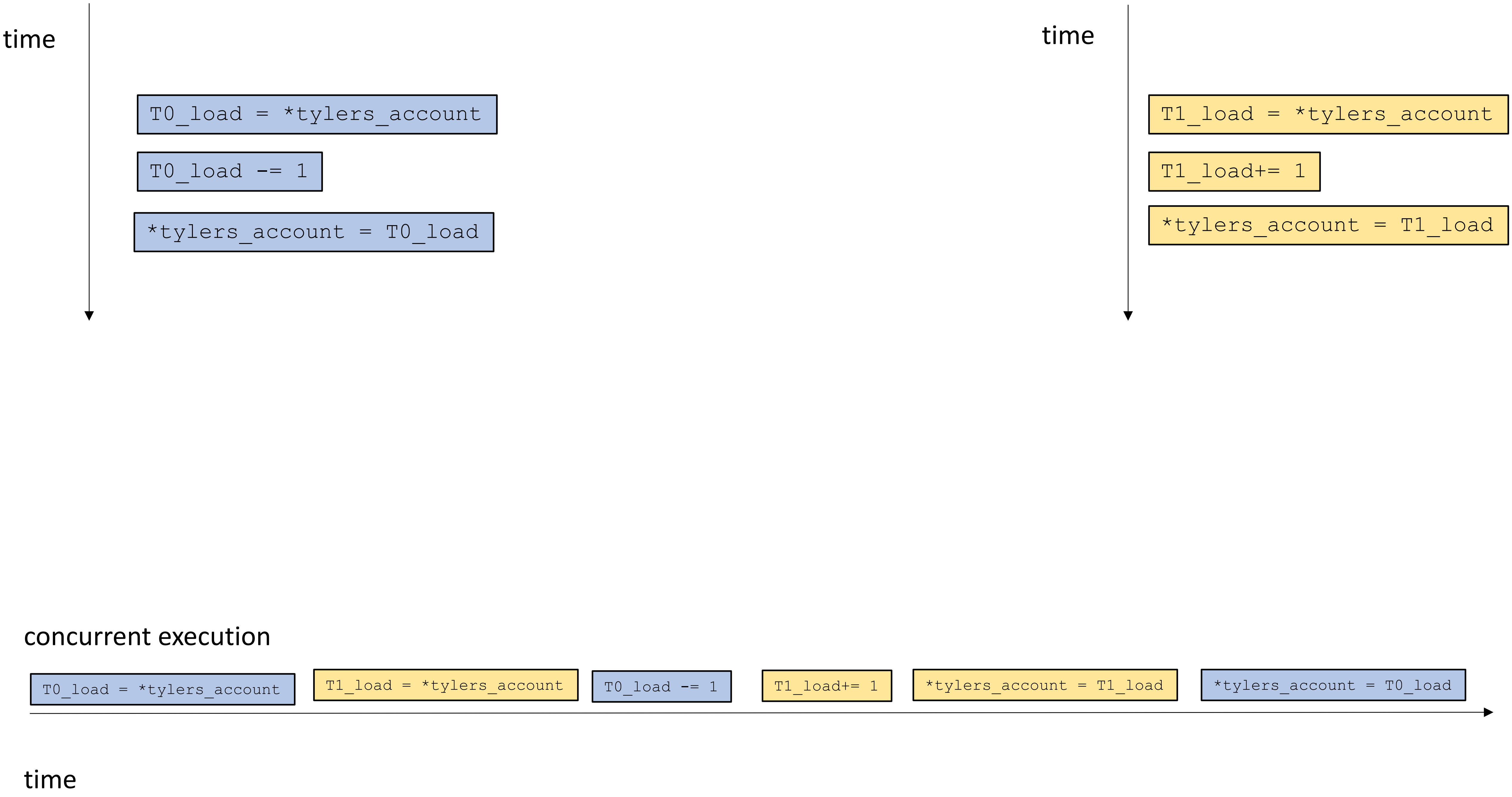
time

```
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
  - More described in Chapter 1 in text book

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

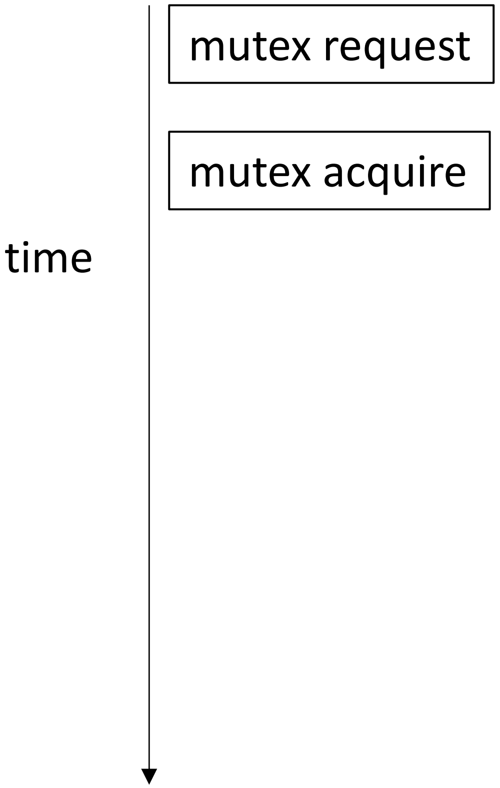
A thread is a sequential program

Tyler's coffee addiction:

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

Tyler's employer

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



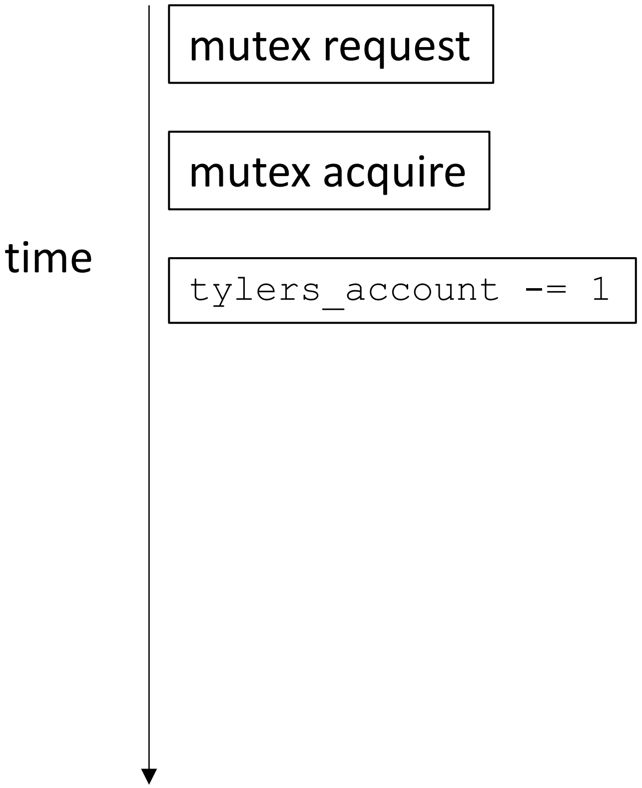
A thread is a sequential program

Tyler's coffee addiction:

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

Tyler's employer

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



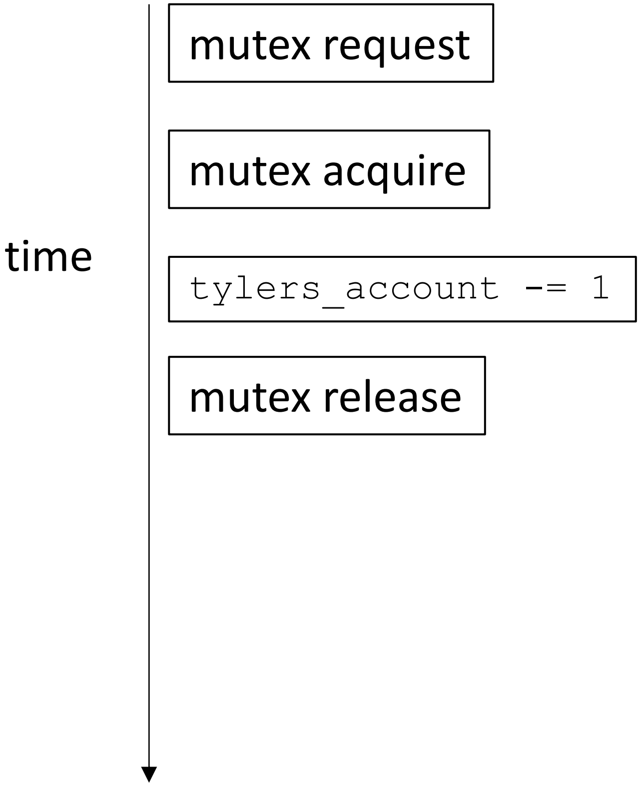
A thread is a sequential program

Tyler's coffee addiction:

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

Tyler's employer

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



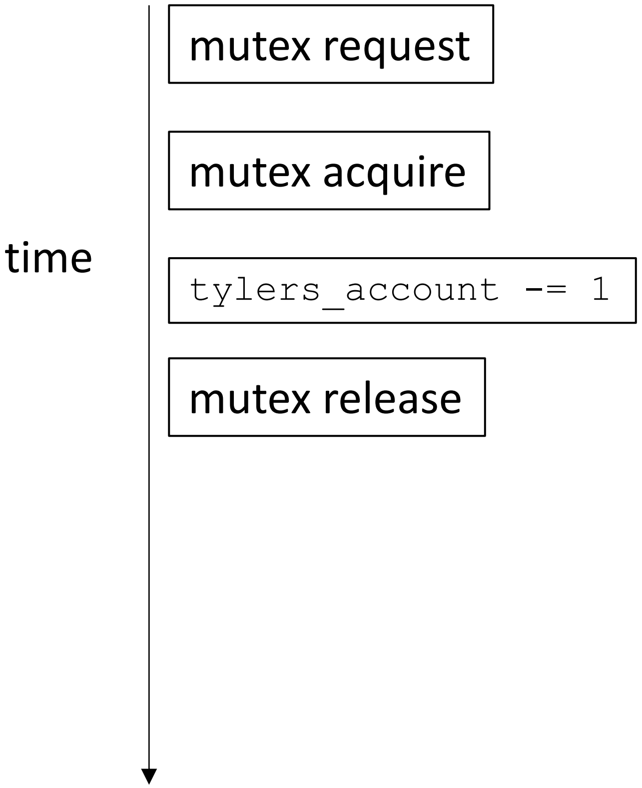
A thread is a sequential program

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

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

<u>Tyler's employer</u>

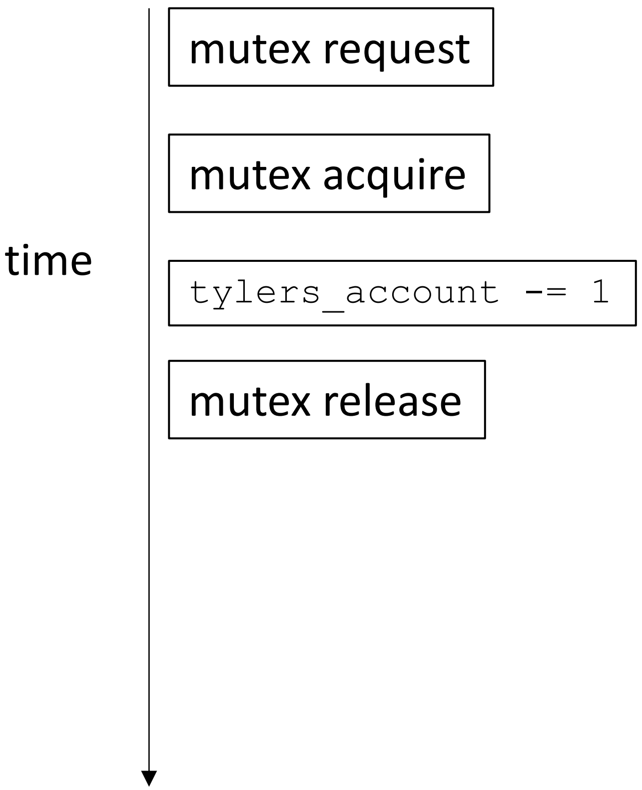
```
m.lock();
tylers_account += 1;
m.unlock();
```



A thread is a sequential program

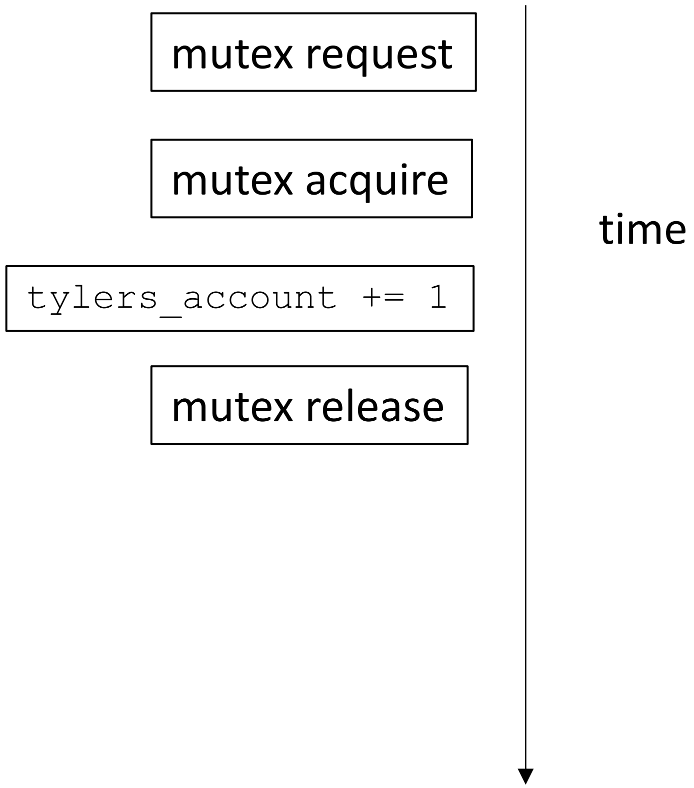
Tyler's coffee addiction:

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



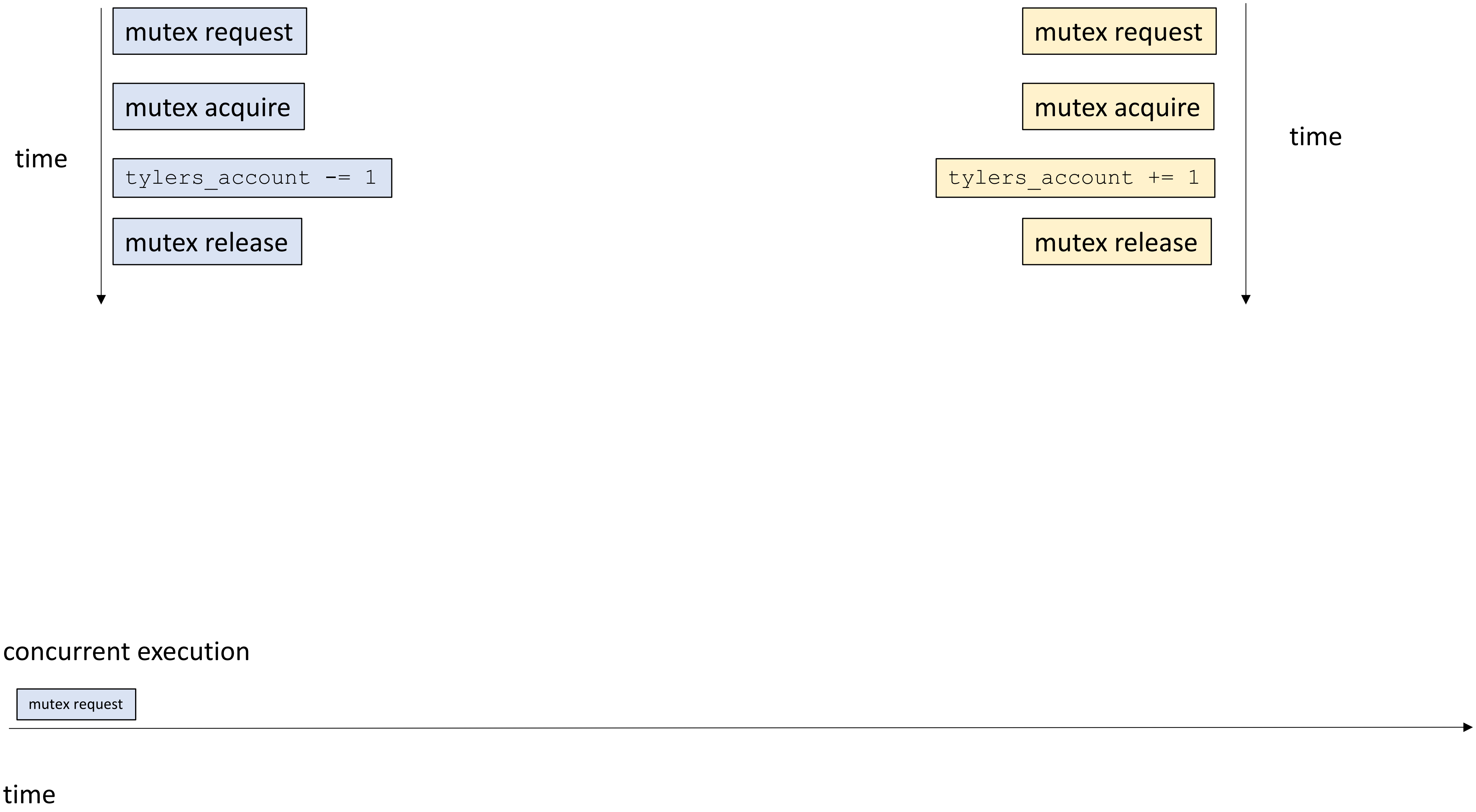
Tyler's employer

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



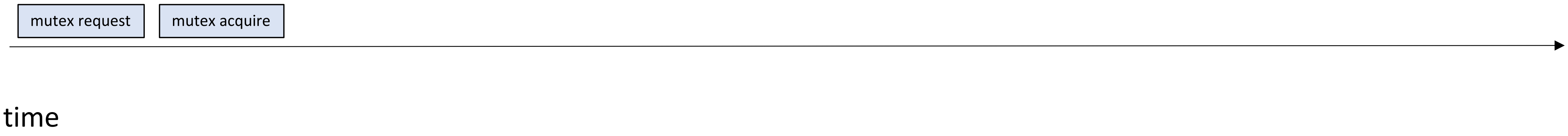
concurrent execution

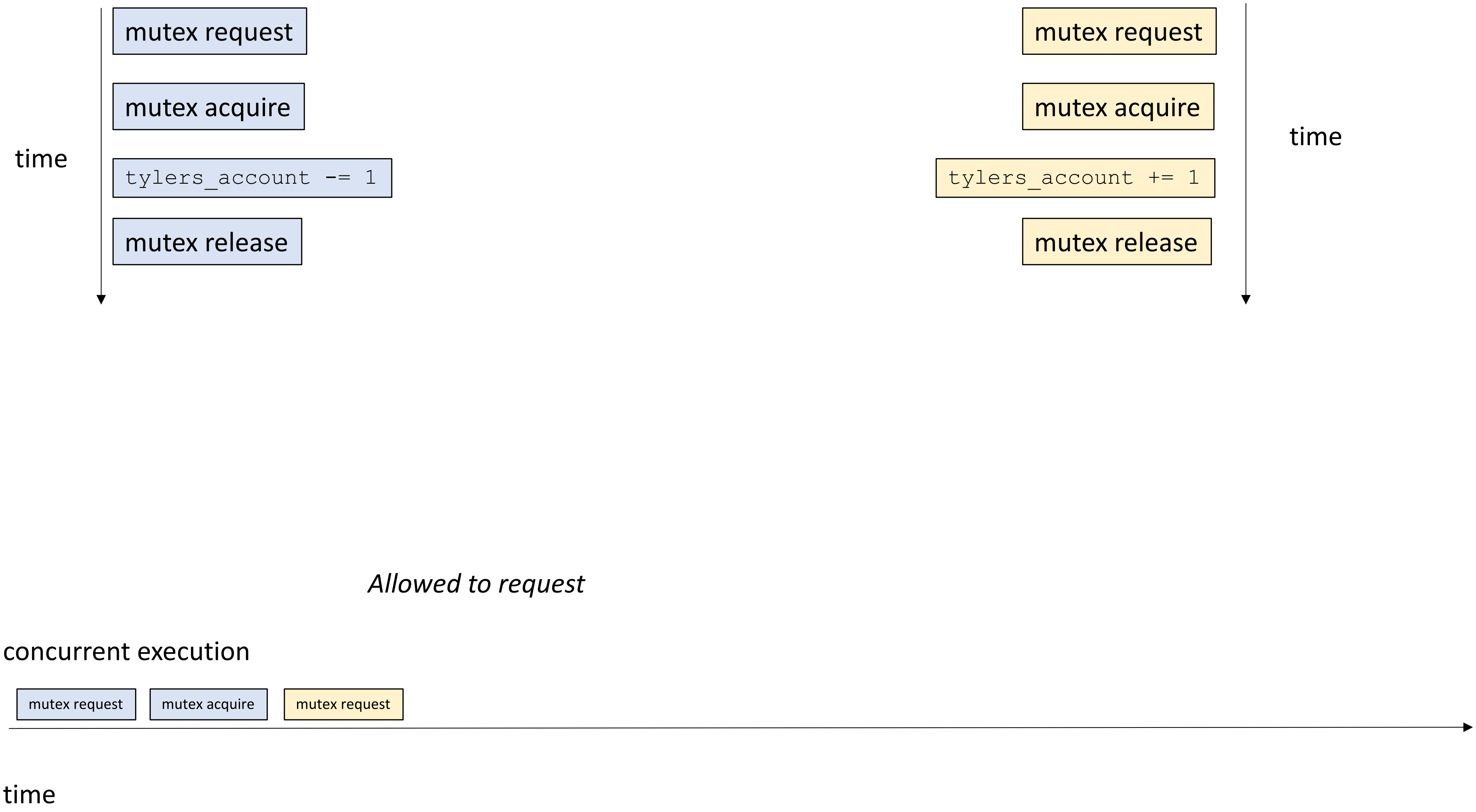
time


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
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
time

mutex request

mutex acquire

time

`tylers_account -= 1`

mutex release

mutex request

mutex acquire

time

`tylers_account += 1`

mutex release

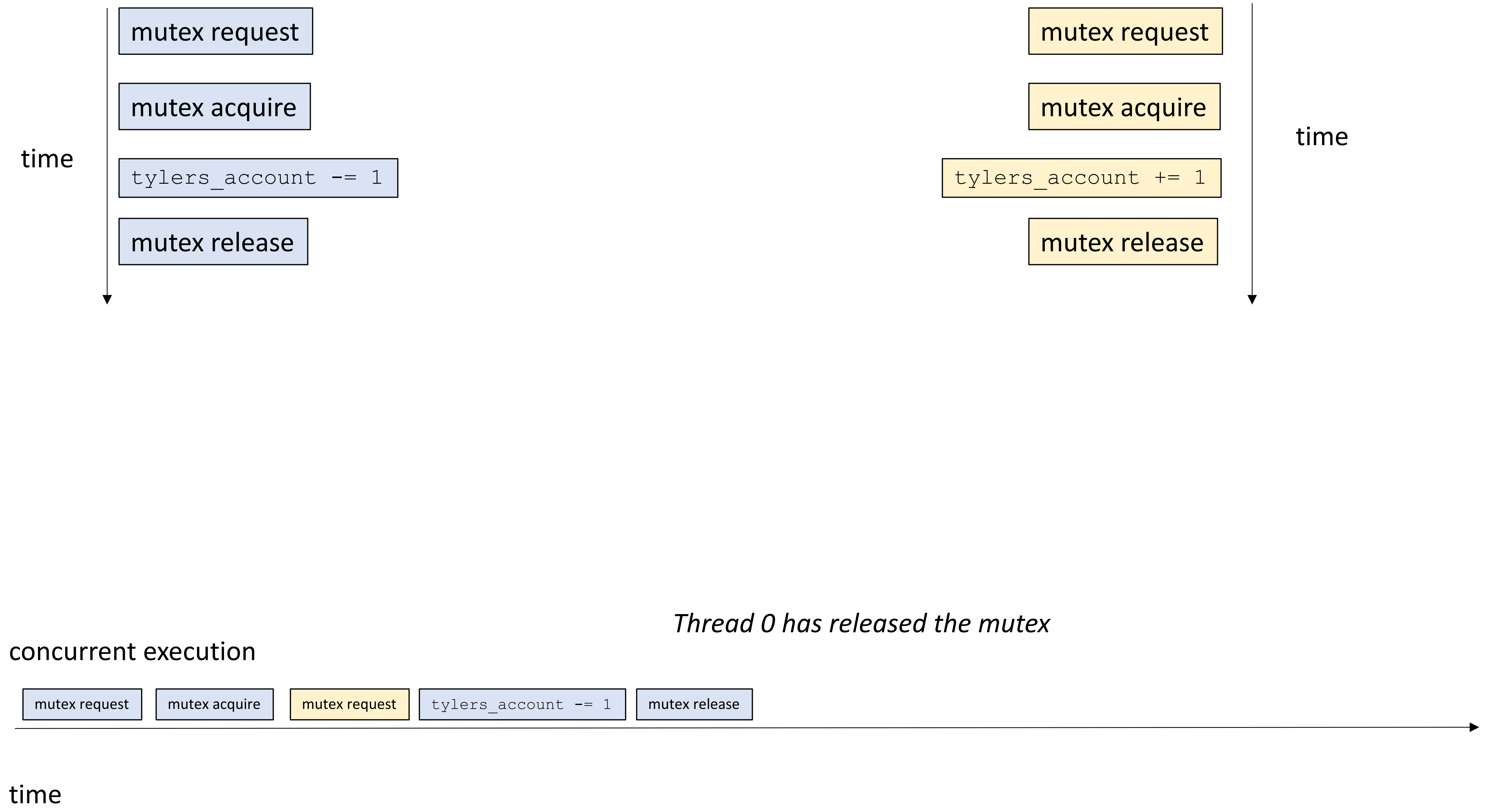
Allowed to request

concurrent execution

mutex request | mutex acquire | mutex request | mutex acquire

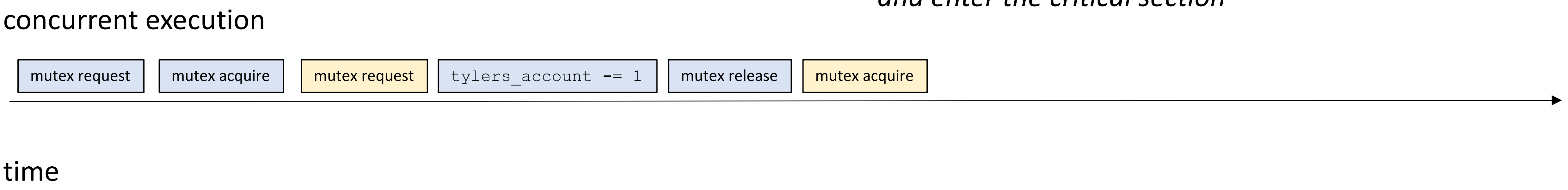
disallowed!

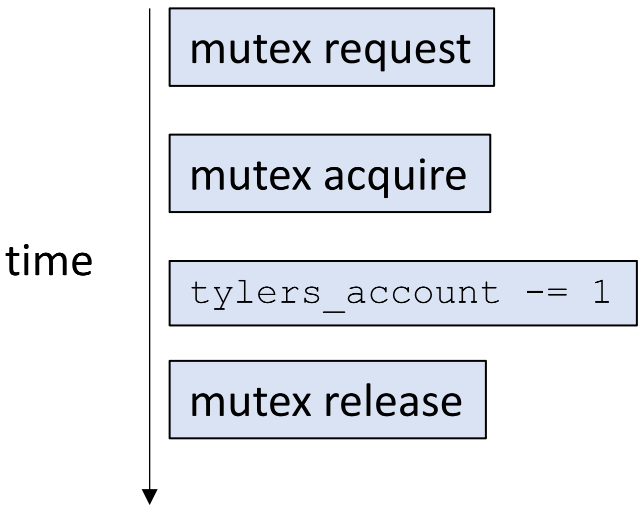
time

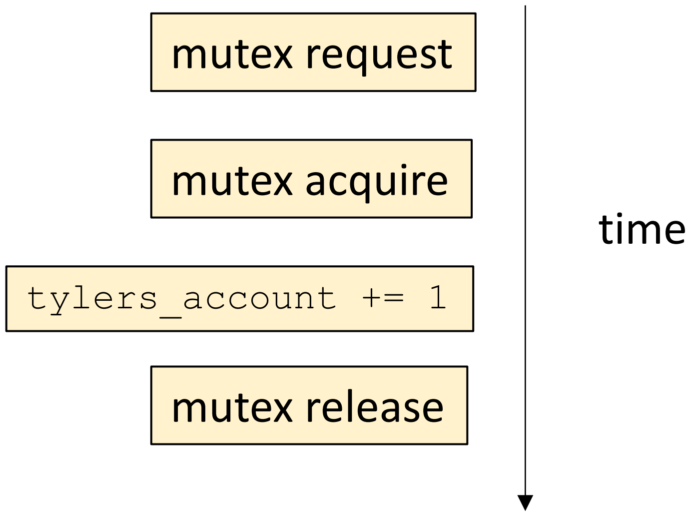


mutex request
mutex acquire
time
tylers_account -= 1
mutex release
mutex request
mutex acquire
time
tylers_account += 1
mutex release
Thread 0 has released the mutex
concurrent execution
mutex request
mutex acquire
mutex request
tylers_account -= 1
mutex release
time

Thread 1 can take the mutex and enter the critical section

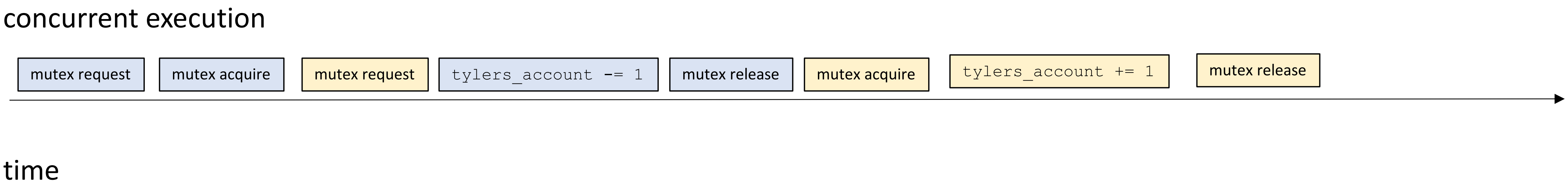






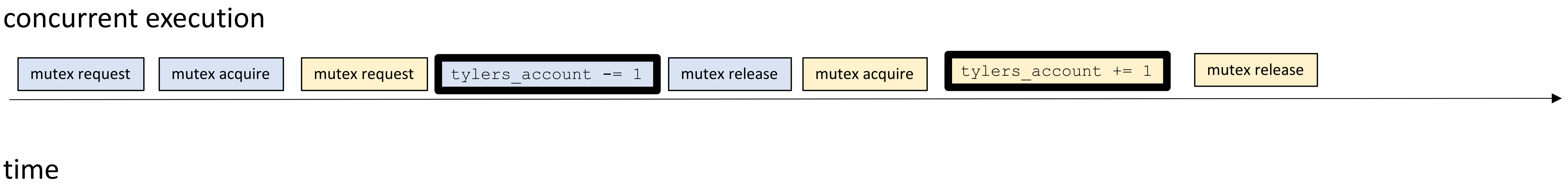
A mutex restricts the number of allowed interleavings
Critical section are mutually exclusive: i.e. they cannot interleave

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



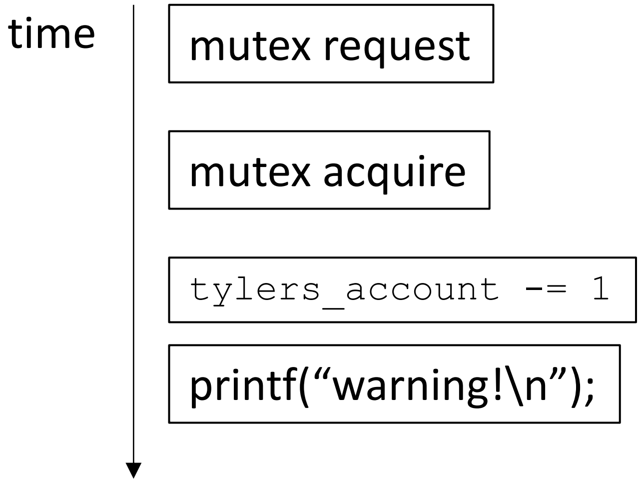
Make sure to unlock your mutex!

Tyler's coffee addiction:

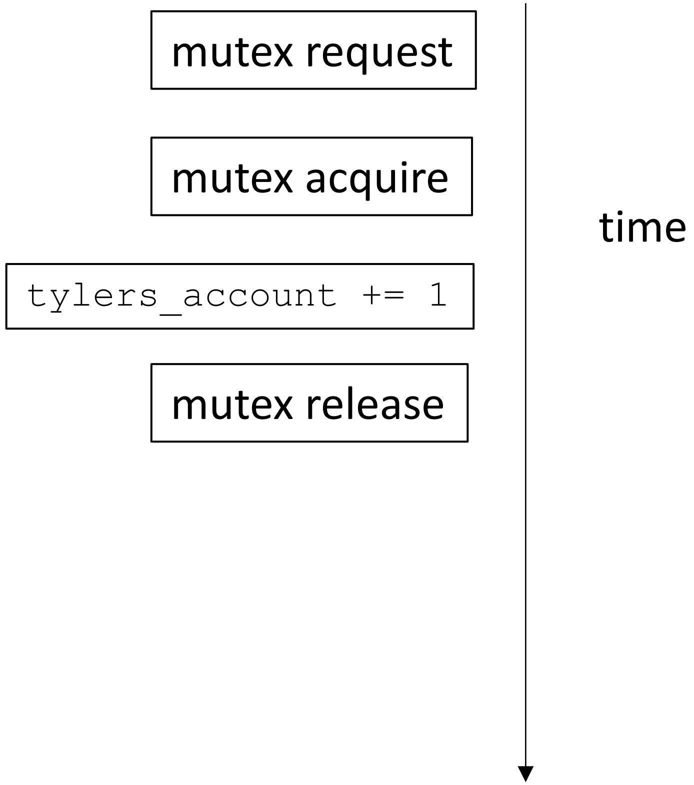
```
m.lock();
tylers_account -= 1;
if (tylers_account < -100) {
  printf("warning!\n");
  return;
}
m.unlock();
return;
```

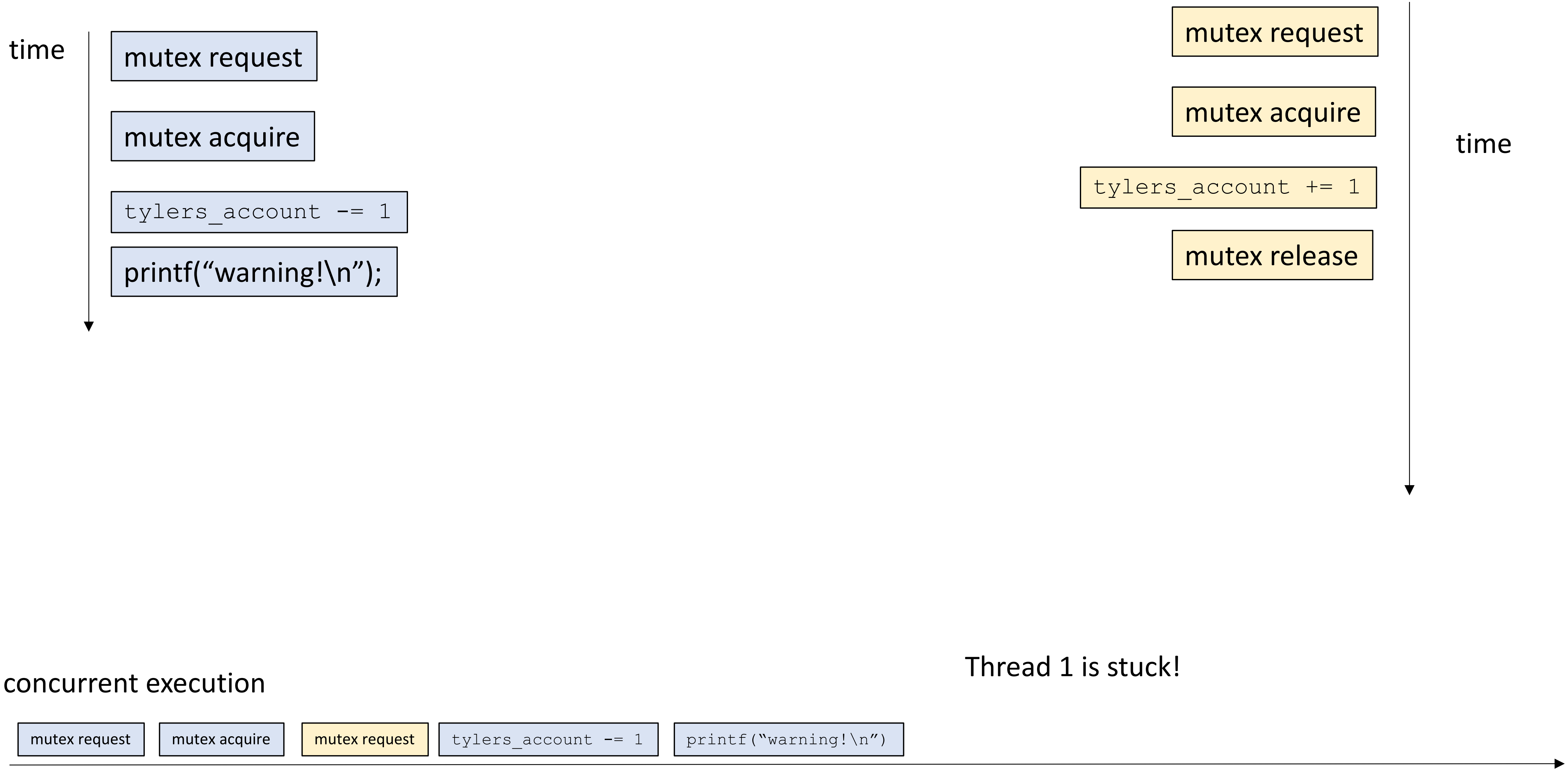
Tyler's employer

```
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”)

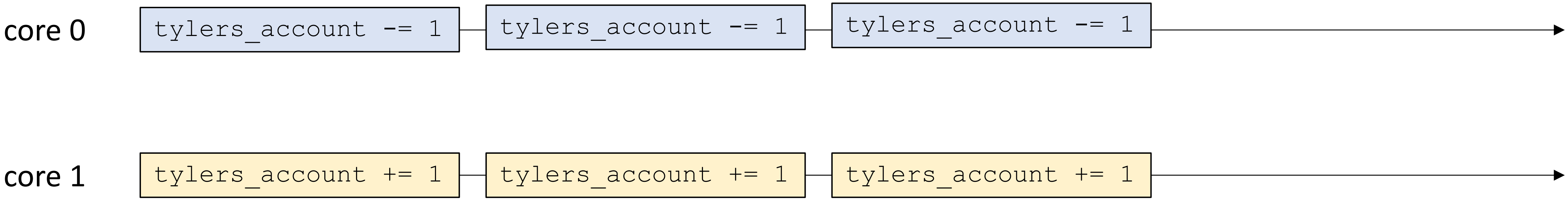
Mutex Performance

- 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

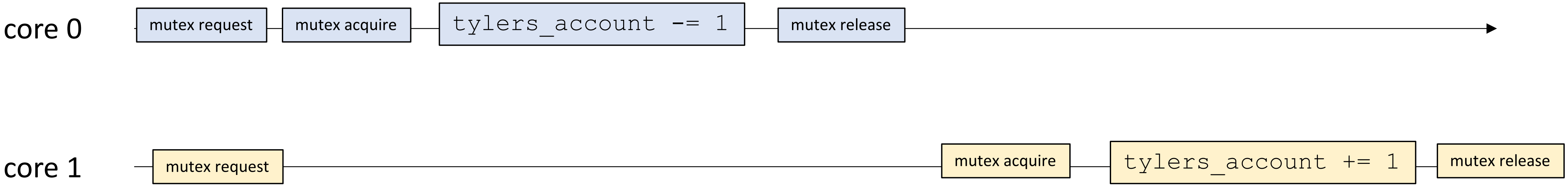
in a parallel system without the mutex



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

Mutex Performance

Try to keep mutual exclusion sections small!

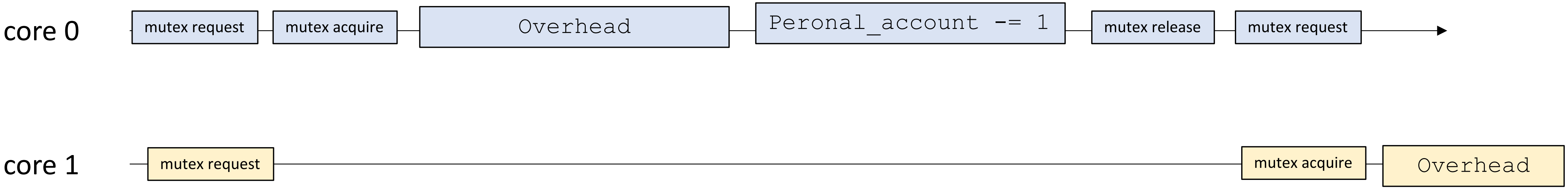
Code example with overhead

Mutex Performance

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

Code example with overhead

Long periods of waiting in the threads

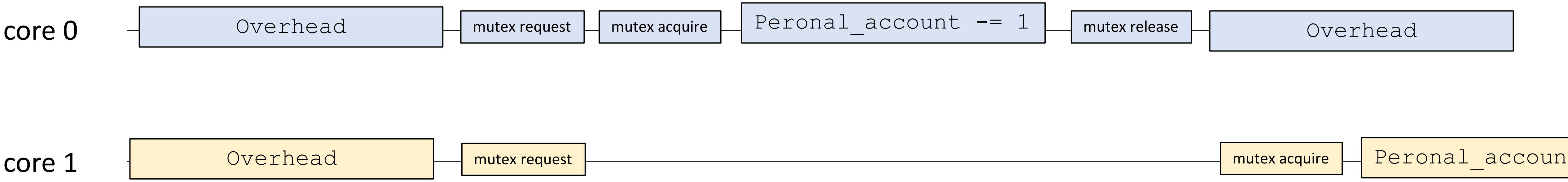


Long periods of waiting in the threads

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

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

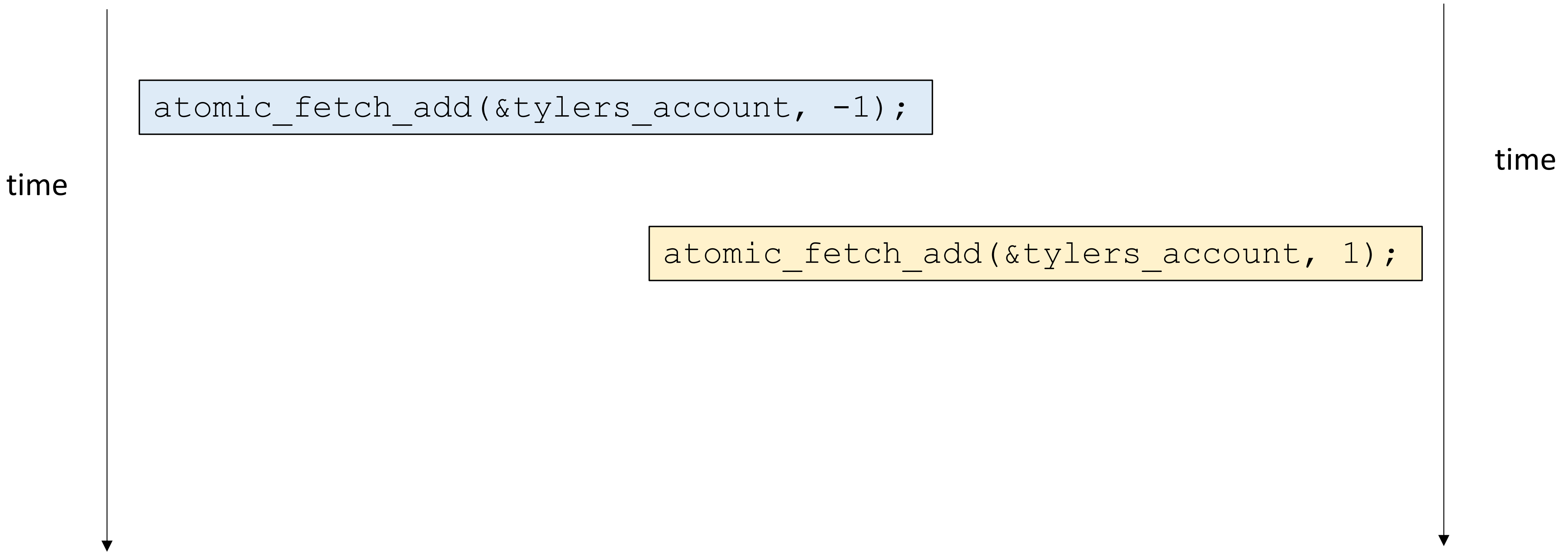
Modify these programs to use atomic RMWs

Tyler's coffee addiction:

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

Tyler's employer

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



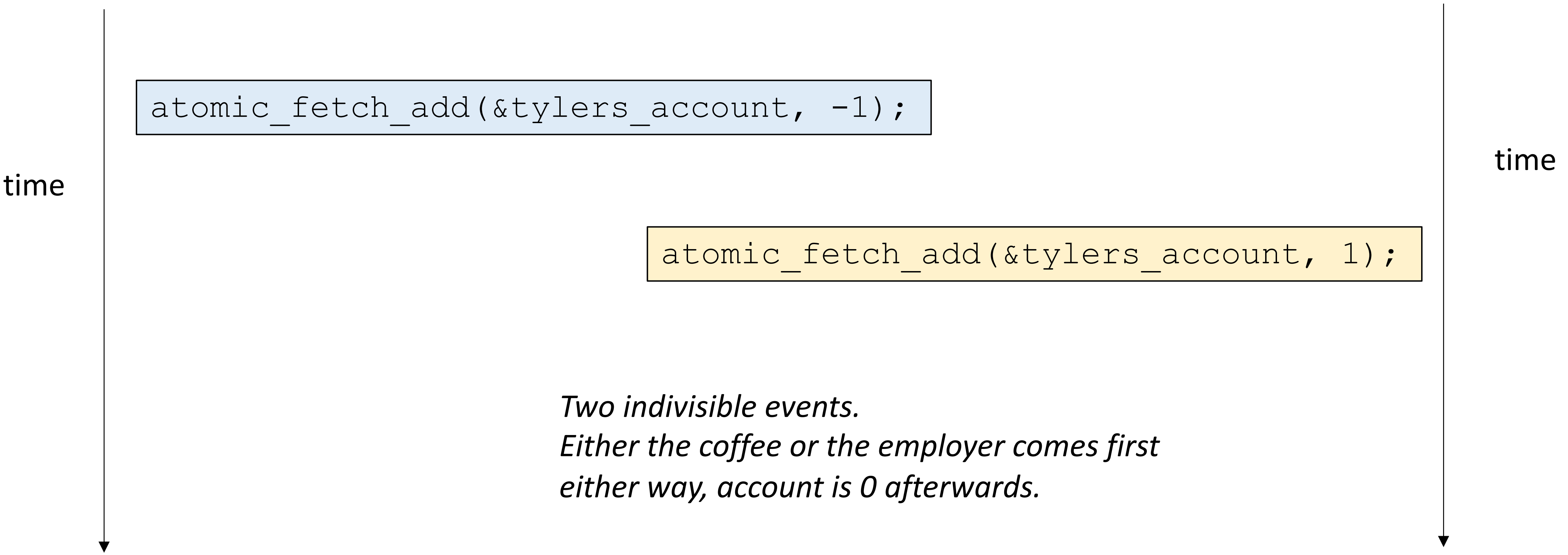
Modify these programs to use atomic RMWs

Tyler's coffee addiction:

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

Tyler's employer

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



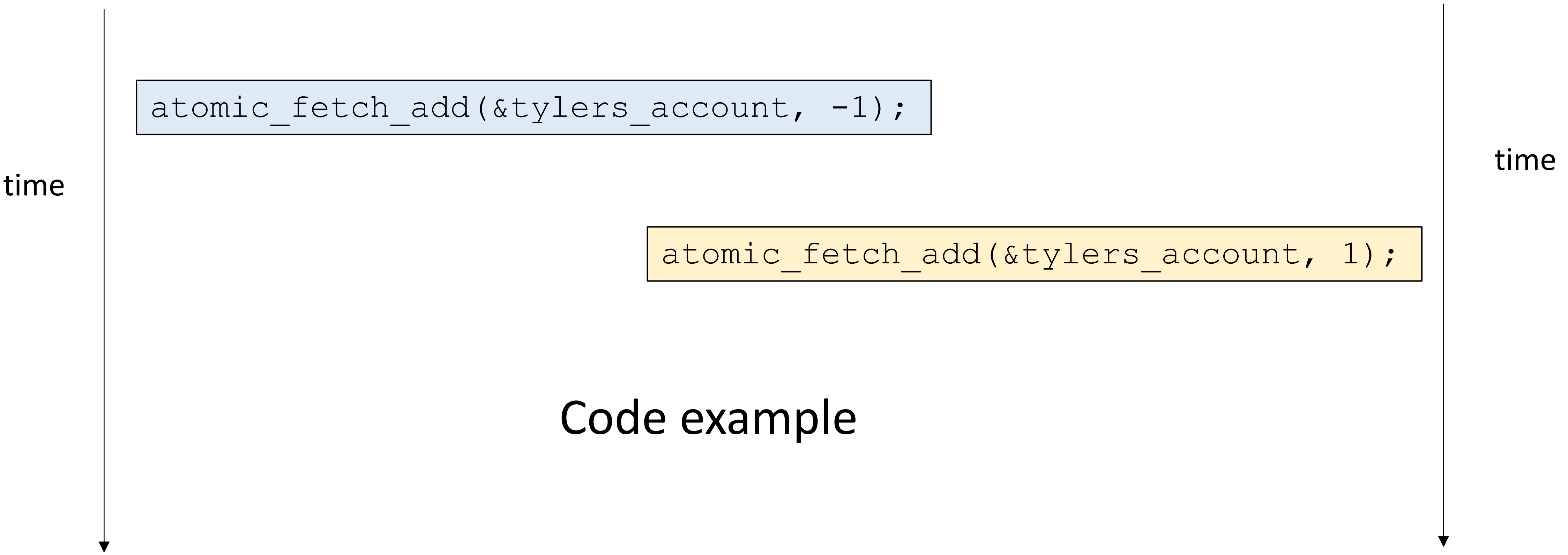
Modify these programs to use atomic RMWs

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?)

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

Multiple mutexes

Lets say I have two accounts:

- Business account
- Personal account

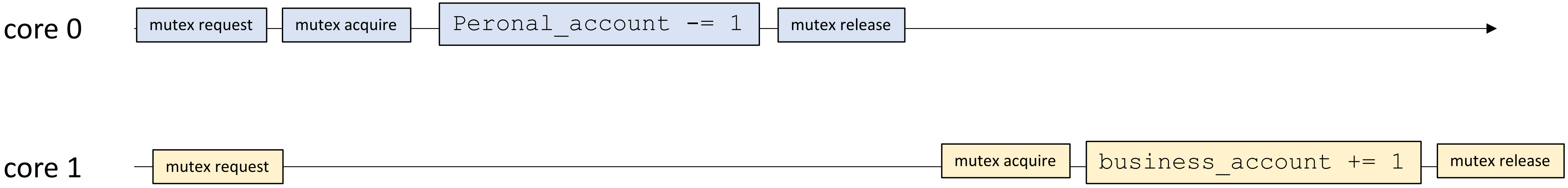
- No reason individual accounts can’t be accessed in parallel

Multiple mutexes

Lets say I have two accounts:

- Business account
- Personal account

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



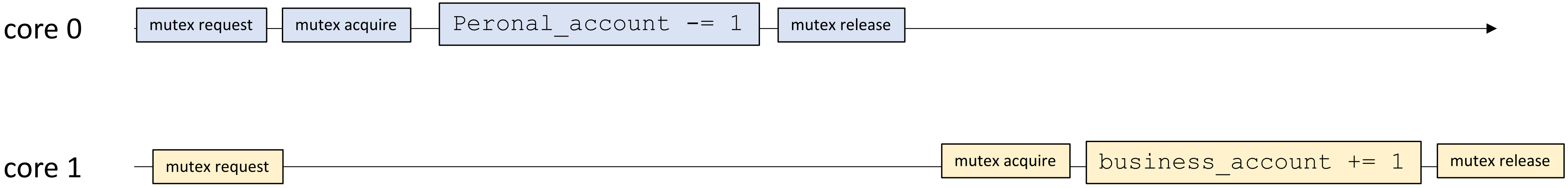
Long periods of waiting in the threads

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

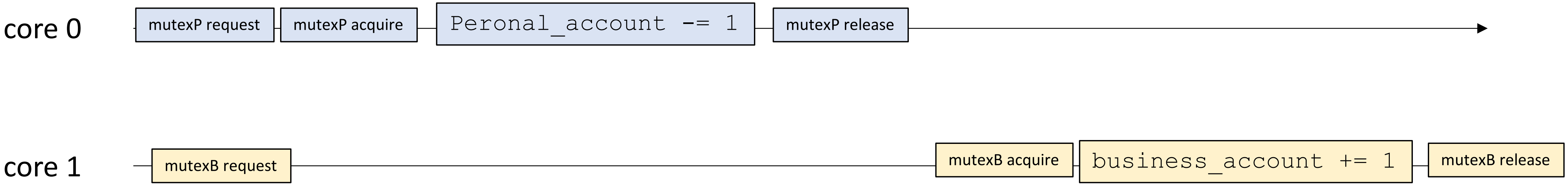


Multiple mutexes

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

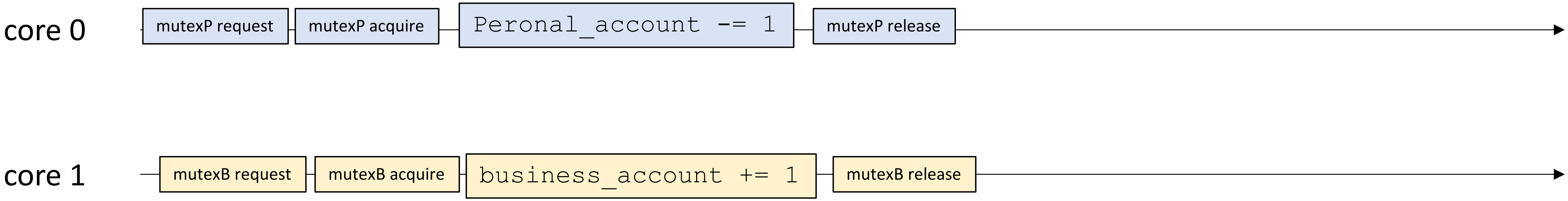


Multiple mutexes

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

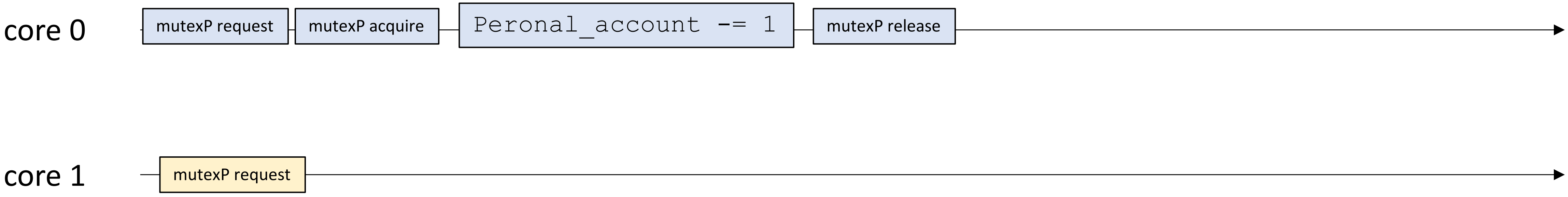


Multiple mutexes

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

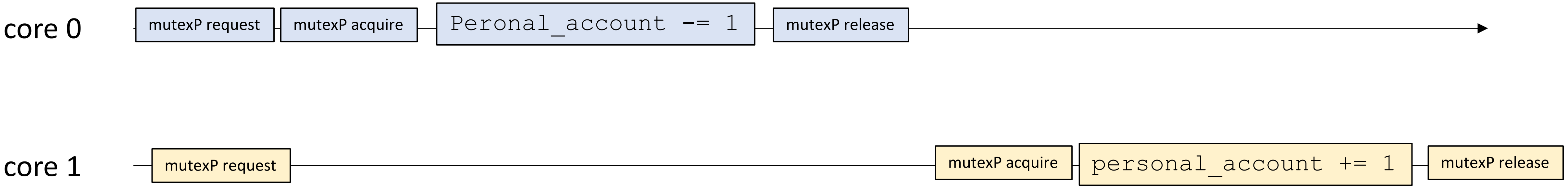


Multiple mutexes

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



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

```
void irs_audit() {
  for (int i = 0; i < NUM_AUDITS; i++) {
    tylers_personal_account_mutex.lock();
    tylers_business_account_mutex.lock();

    AUDIT(tylers_personal_account, tylers_business_account);

    tylers_personal_account_mutex.unlock();
    tylers_business_account_mutex.unlock();
  }
}
```

Multiple mutexes

- Our program deadlocked! What happened?

```
void ucsc_audit() {
  for (int i = 0; i < NUM_AUDITS; i++) {
    tylers_business_account_mutex.lock();
    tylers_personal_account_mutex.lock();

    AUDIT(tylers_personal_account, tylers_business_account);

    tylers_personal_account_mutex.unlock();
    tylers_business_account_mutex.unlock();
  }
}
```

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 0 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

Thanks!

- Next time:
  - Implementing Mutexes