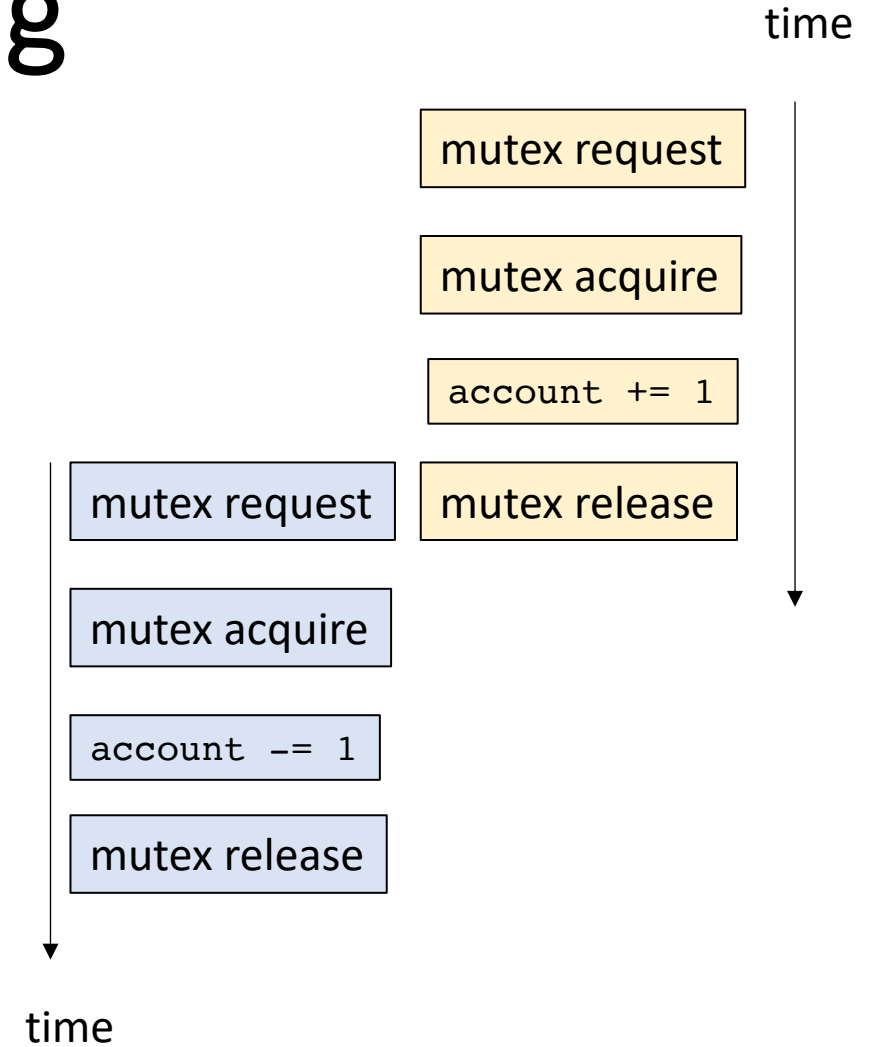


CSE113: Parallel Programming

Jan. 14, 2022

- **Topics:**

- Review SPMD programming model
- Intro to mutual exclusion
 - Different types of parallelism
 - Data conflicts
 - Protecting shared data



Announcements

- Starting module 2
- No class on Monday (enjoy your long weekend!)
- Homework 1 is released
 - We have covered everything you need in class
- Office hours/mentoring hours/piazza
 - lots of support available

Announcements

- Homework 1 notes:
 - No assigned speedup required. You should get a noticeable speedup from ILP
 - You can start to share results on Monday. Everyone's results will be slightly different
 - Sometimes you cannot account for small differences
 - We should be running the code for more iterations

Today's Quiz

- Make sure to do it!

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.

Previous quiz

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

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.

Review

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) {  
    for (int i = 0; i < a_size; i++) {  
        a[i]++;  
    }  
}
```

*lets do this in parallel!
each thread increments different
elements in the array*

SPMD programming model

```
void increment_array(int *a, int a_size, int tid, int num_threads) {  
    for (int i = 0; i < a_size; i++) {  
        a[i]++;  
    }  
}
```

The function gets a thread id and the number of threads

SPMD programming model

```
void increment_array(int *a, int a_size, int tid, int num_threads) {  
    for (int i = 0; i < a_size; i++) {  
        a[i]++;  
    }  
}
```

*A few options on how to split up the work
lets do round robin*

SPMD programming model

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

SPMD programming model

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



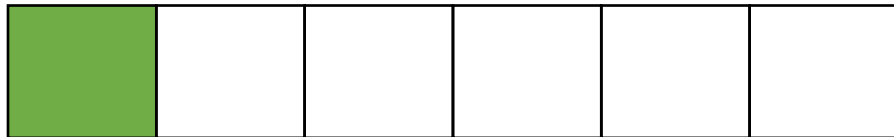
array a

Assume 2 threads
lets step through thread 0
i.e.
tid = 0
num_threads = 2

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]++;  
    }  
}
```

iteration 1 computes index 0



array a

Assume 2 threads
lets step through thread 0
i.e.
tid = 0
num_threads = 2

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]++;  
    }  
}
```

iteration 2 computes index 2



array a

Assume 2 threads
lets step through thread 0
i.e.
tid = 0
num_threads = 2

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]++;  
    }  
}
```

iteration 3 computes index 4



array a

Assume 2 threads
lets step through thread 0
i.e.
tid = 0
num_threads = 2

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

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



array a

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]++;  
    }  
}
```

iteration 1 computes index 1



array a

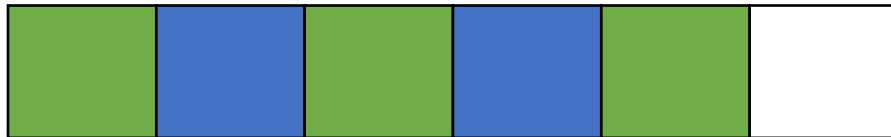
switch to thread 1

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

iteration 2 computes index 3



array a

switch to thread 1

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

iteration 3 computes index 5



array a

switch to thread 1

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;
```

```
}
```

On to the lecture!

Lecture Schedule

- Introduction to thread-level parallelism
- Data conflicts
- Mutual exclusion

Lecture Schedule

- **Introduction to thread-level parallelism**
- Data conflicts
- Mutual exclusion

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

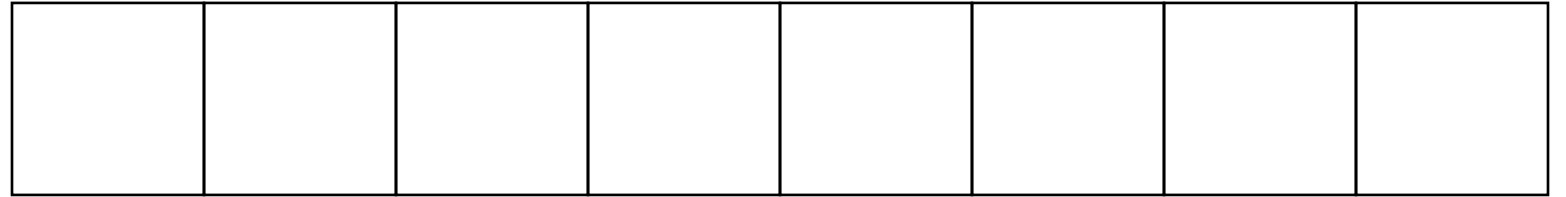
- Consider the following program:

There are 3 arrays: a , b , c .

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

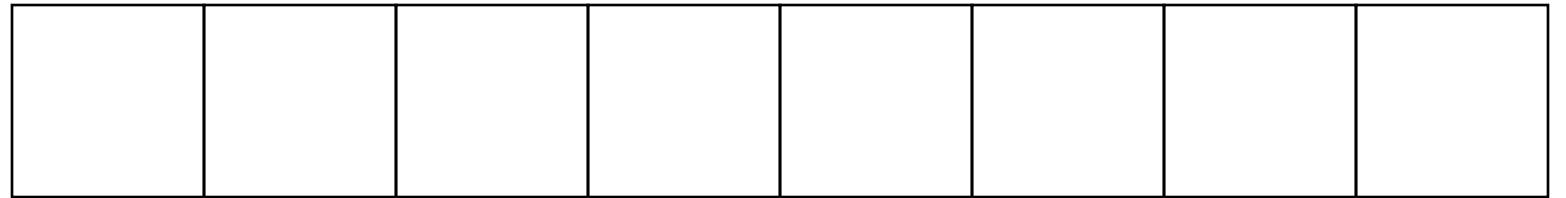
Embarrassingly parallel

array a



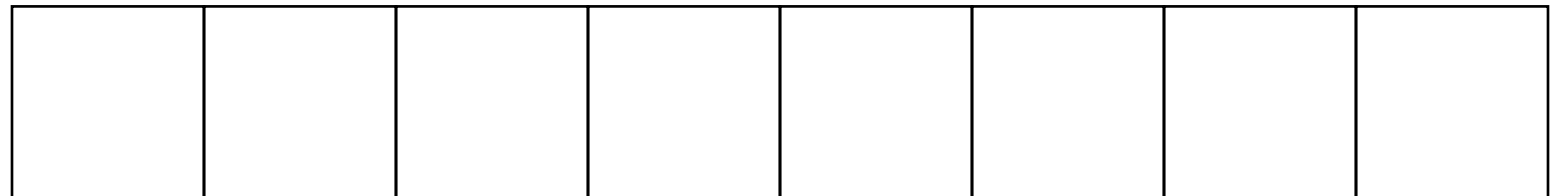
+ + + + + + + +

array b



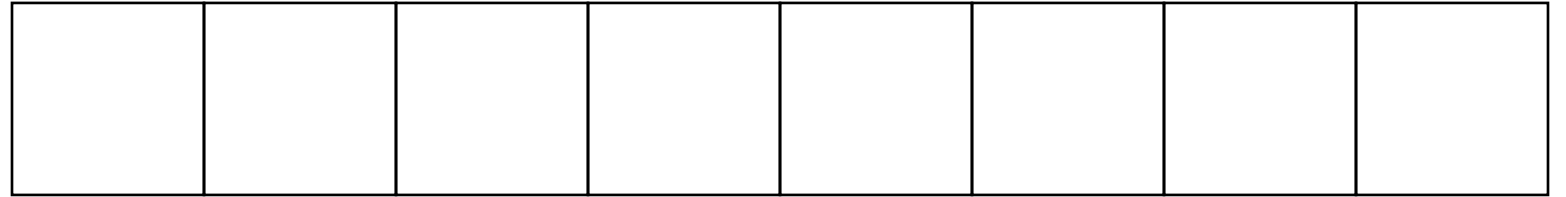
= = = = = = = =

array c



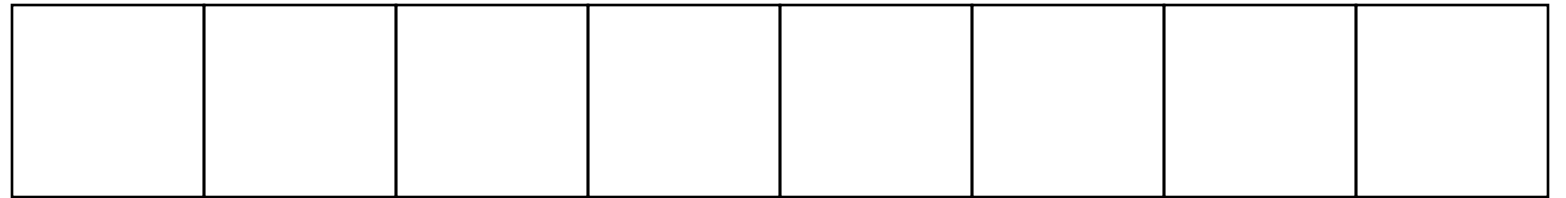
Embarrassingly parallel

array a



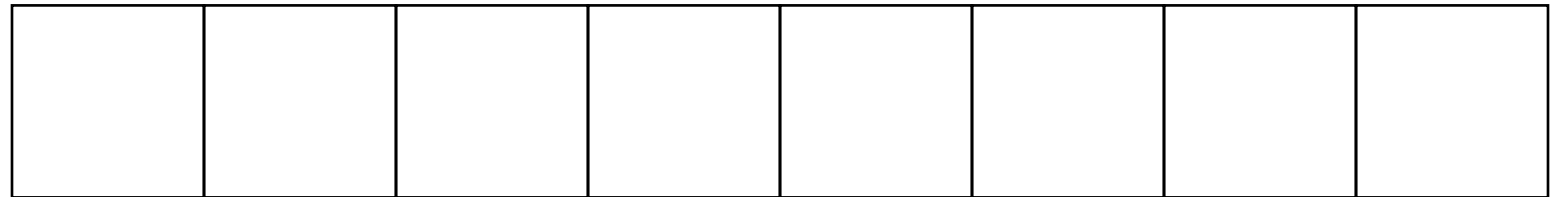
+ + + + + + + +

array b



= = = = = = = =

array c

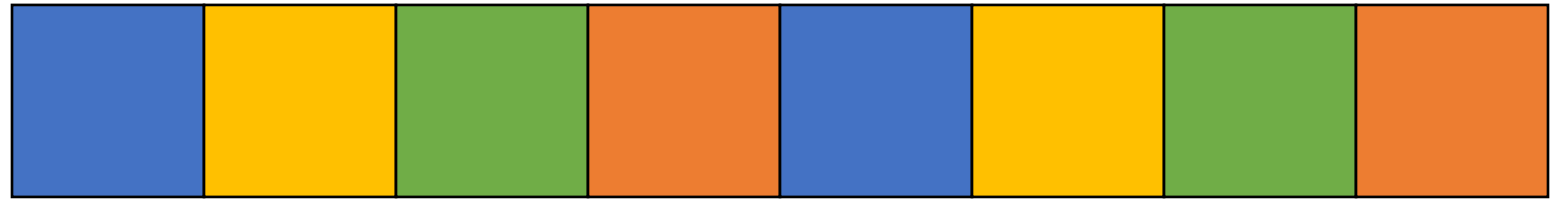


Computation
can easily be
divided into
threads

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

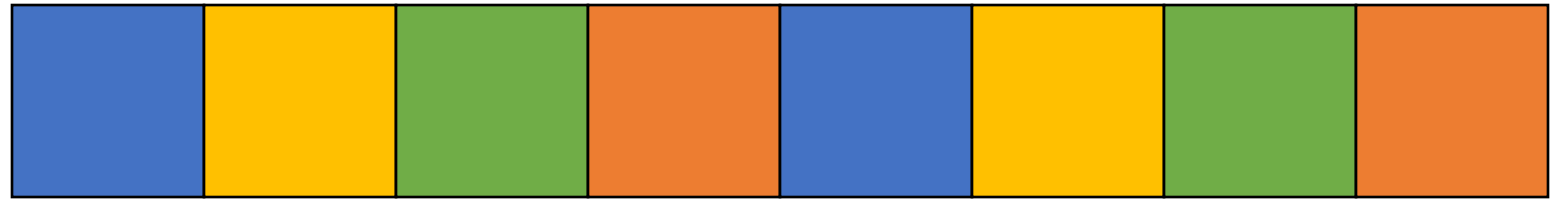
Embarrassingly parallel

array a



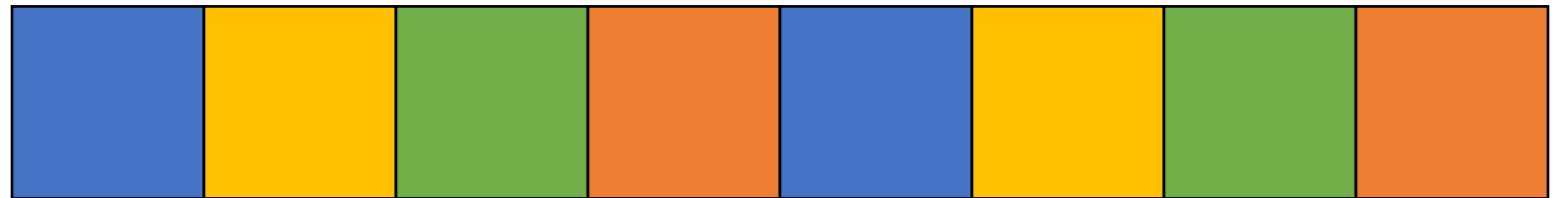
+ + + + + + + +

array b



= = = = = = = =

array c

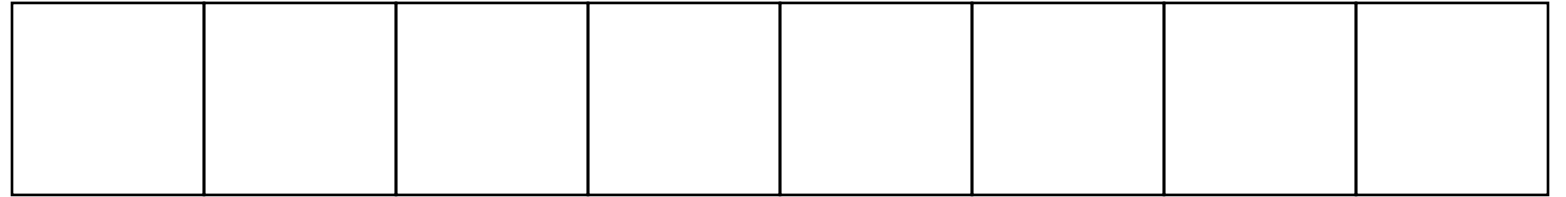


Computation
can easily be
divided into
threads

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

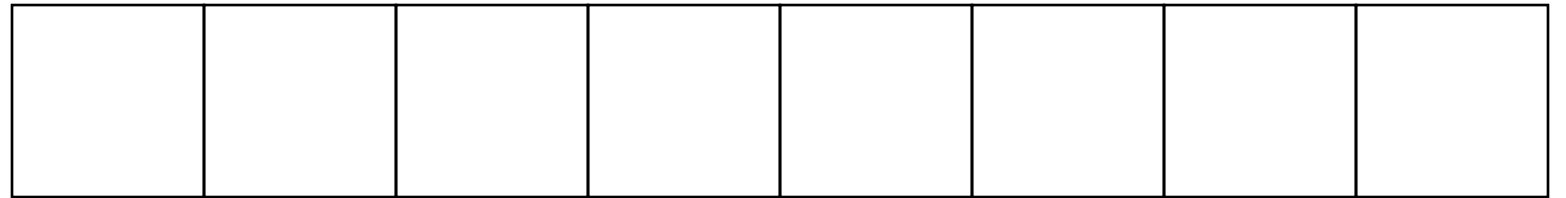
Embarrassingly parallel

array a



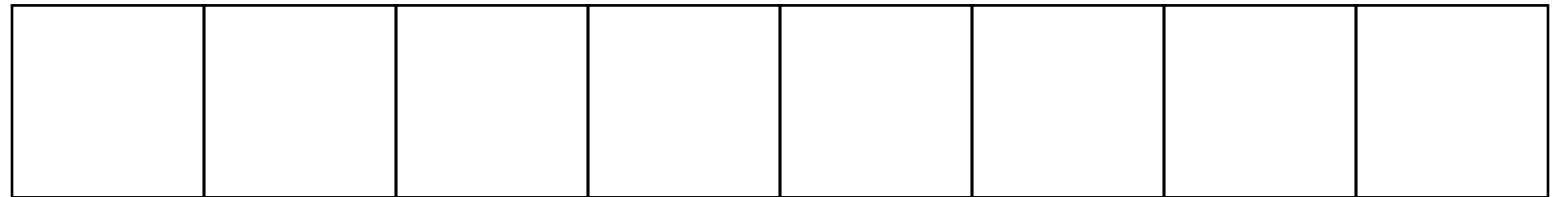
+ + + + + + + +

array b



= = = = = = = =

array c



Computation
can easily be
divided into
threads

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

Embarrassingly parallel

array a



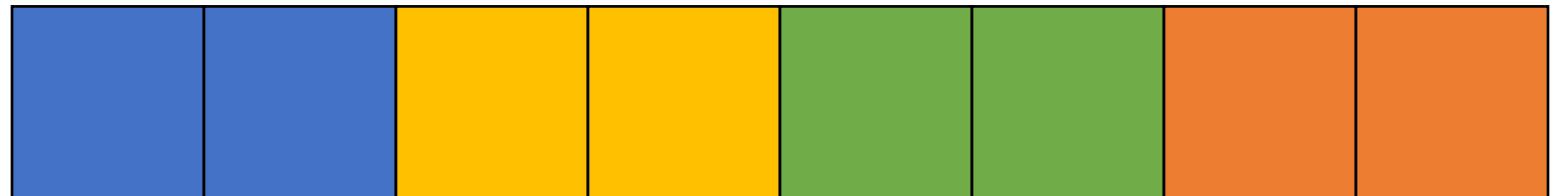
+ + + + + + + +

array b



= = = = = = = =

array c



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

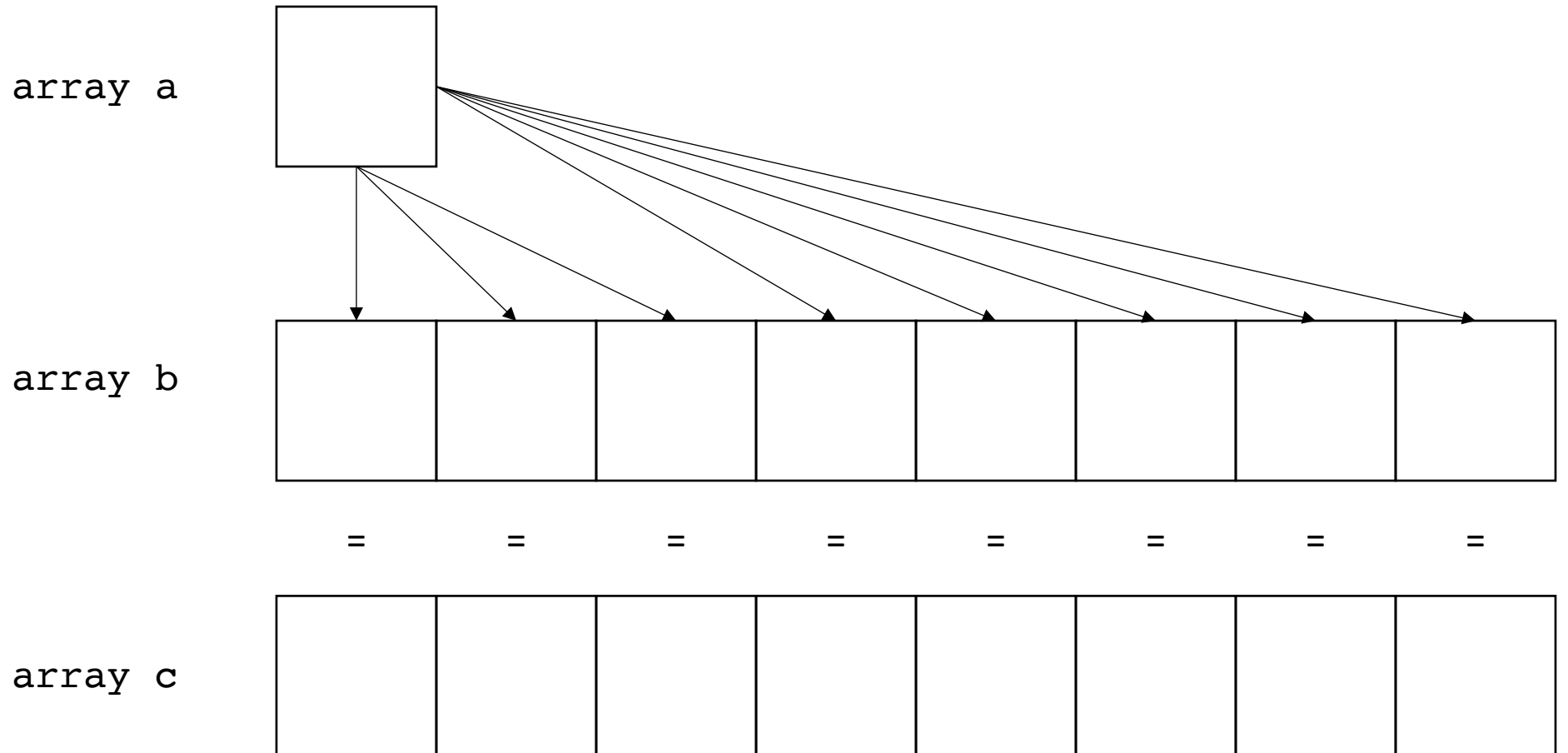
- Next Program

There are 3 arrays: a , b , c .

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

Embarrassingly parallel

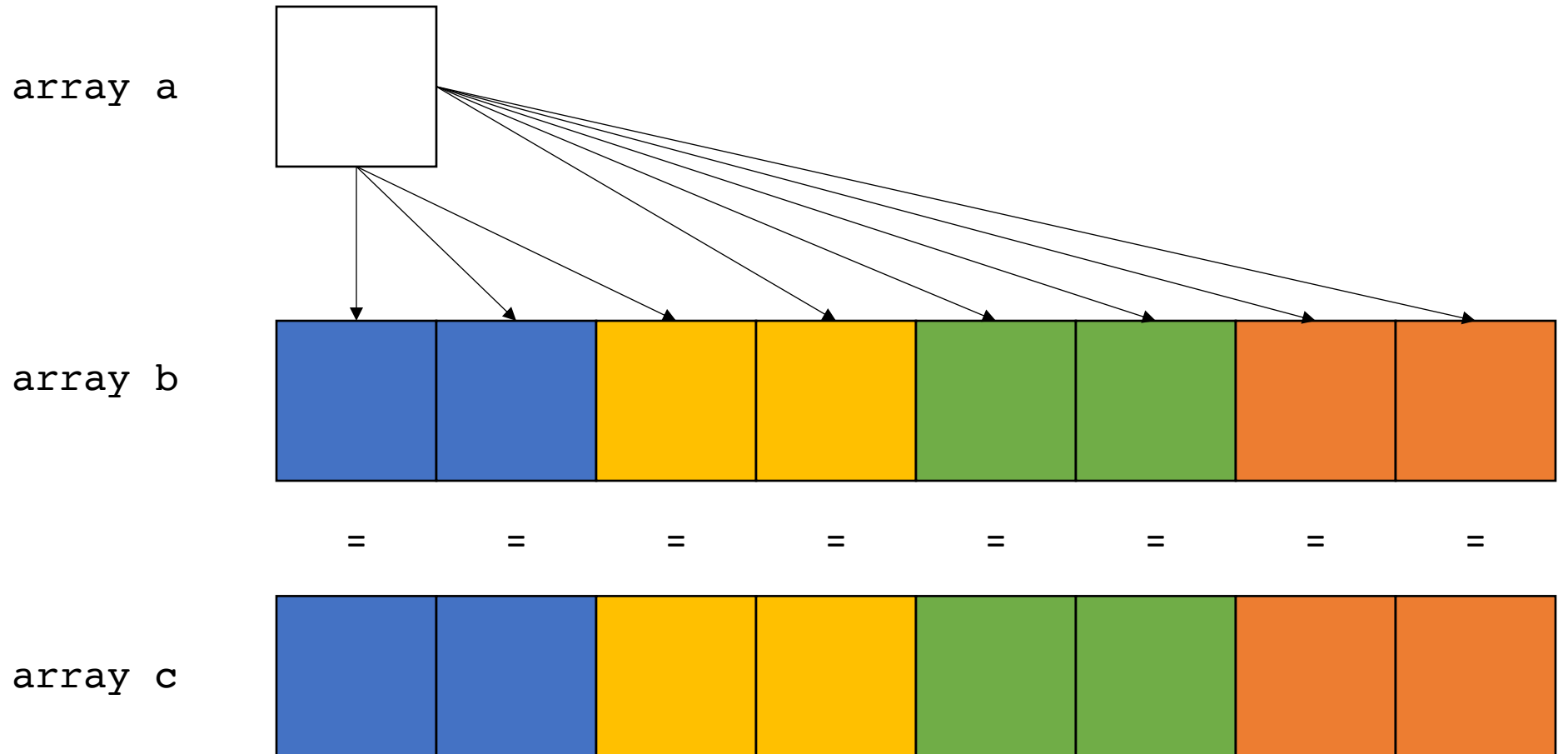
- Thread 0 - Blue
- Thread 1 - Yellow
- Thread 2 - Green
- Thread 3 - Orange



*is this problem
embarrassingly
parallel?*

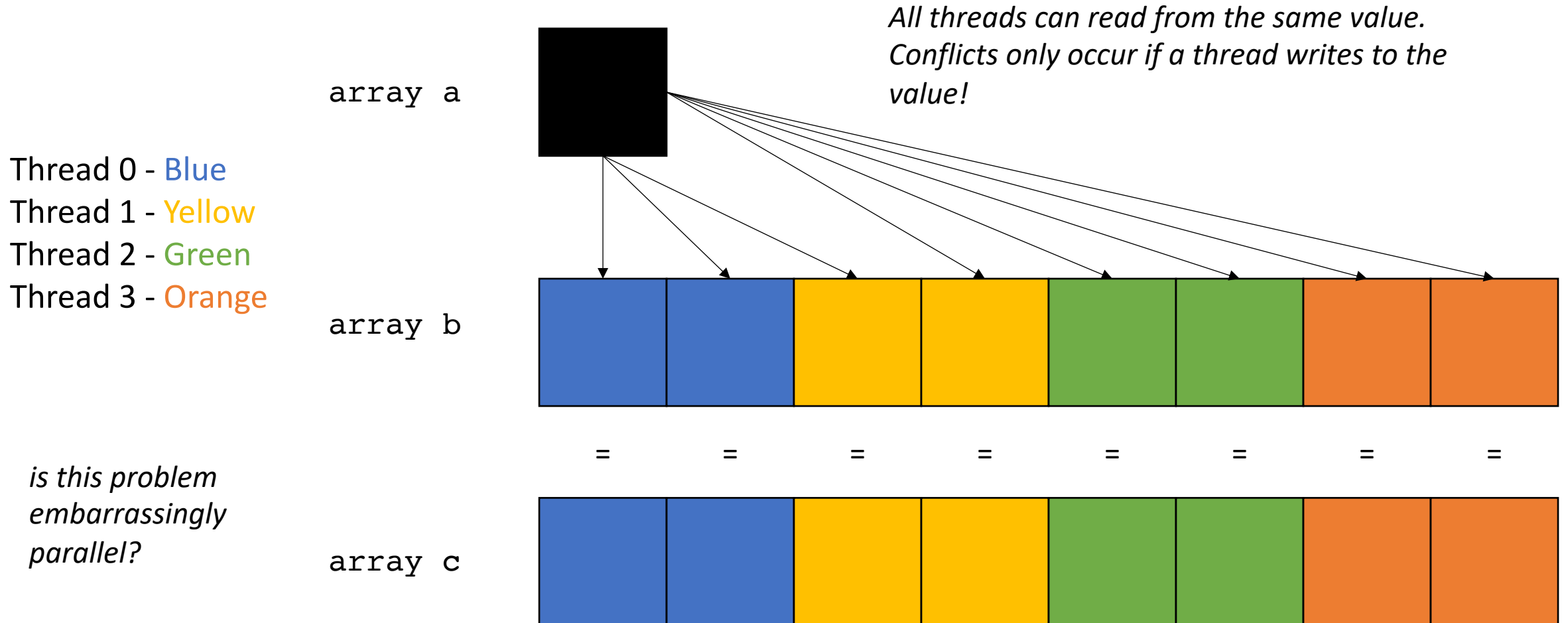
Embarrassingly parallel

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



*is this problem
embarrassingly
parallel?*

Embarrassingly parallel



Embarrassingly parallel

- Next Program

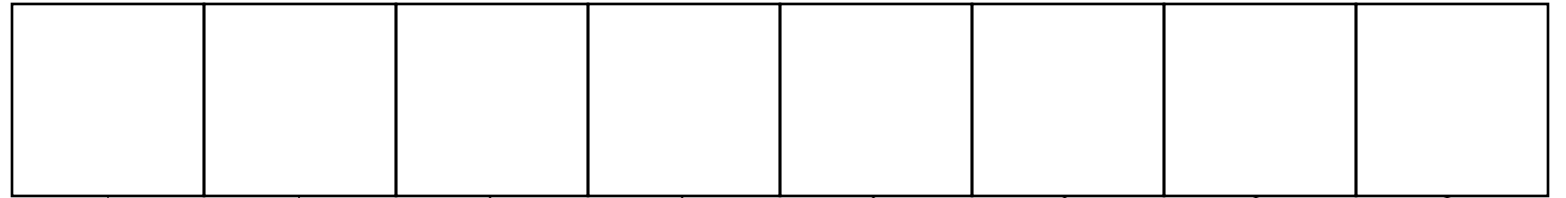
There are 2 arrays: b , c

We want to compute $c[0] = b[0] + b[1] + b[2] \dots$

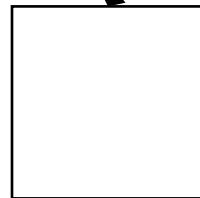
Embarrassingly parallel

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

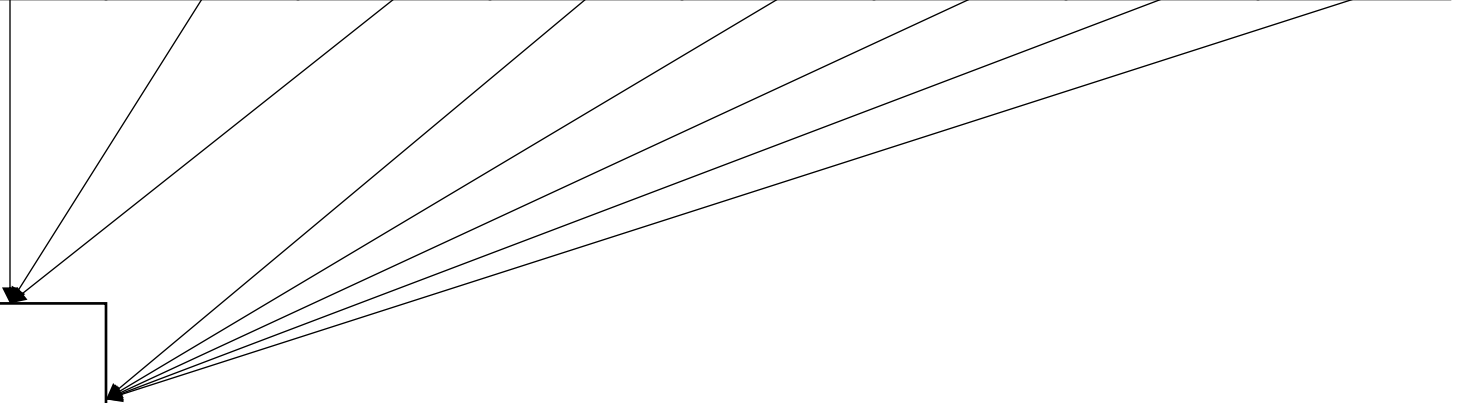
array b



array c



*is this problem
embarrassingly
parallel?*



Embarrassingly parallel

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

array b



*is this problem
embarrassingly
parallel?*

array c

*threads read
unique locations*

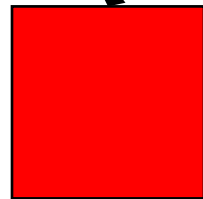
Embarrassingly parallel

- 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?*

Conflict because multiple threads write to the same location!

Embarrassingly parallel

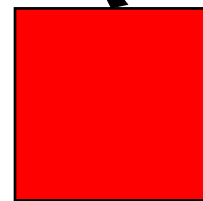
Note: Reductions have some parallelism in them, as seen in your homework.

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

array b



array c



*threads read
unique locations*

*is this problem
embarrassingly
parallel?*

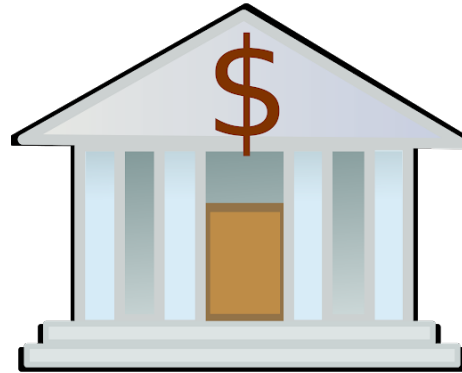
Conflict because multiple threads write to the same location!

We need a way how to safely share memory

- *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

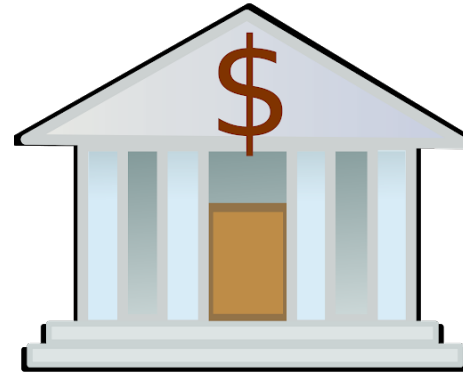


My account: \$\$



We need a way how to safely share memory

- Bank



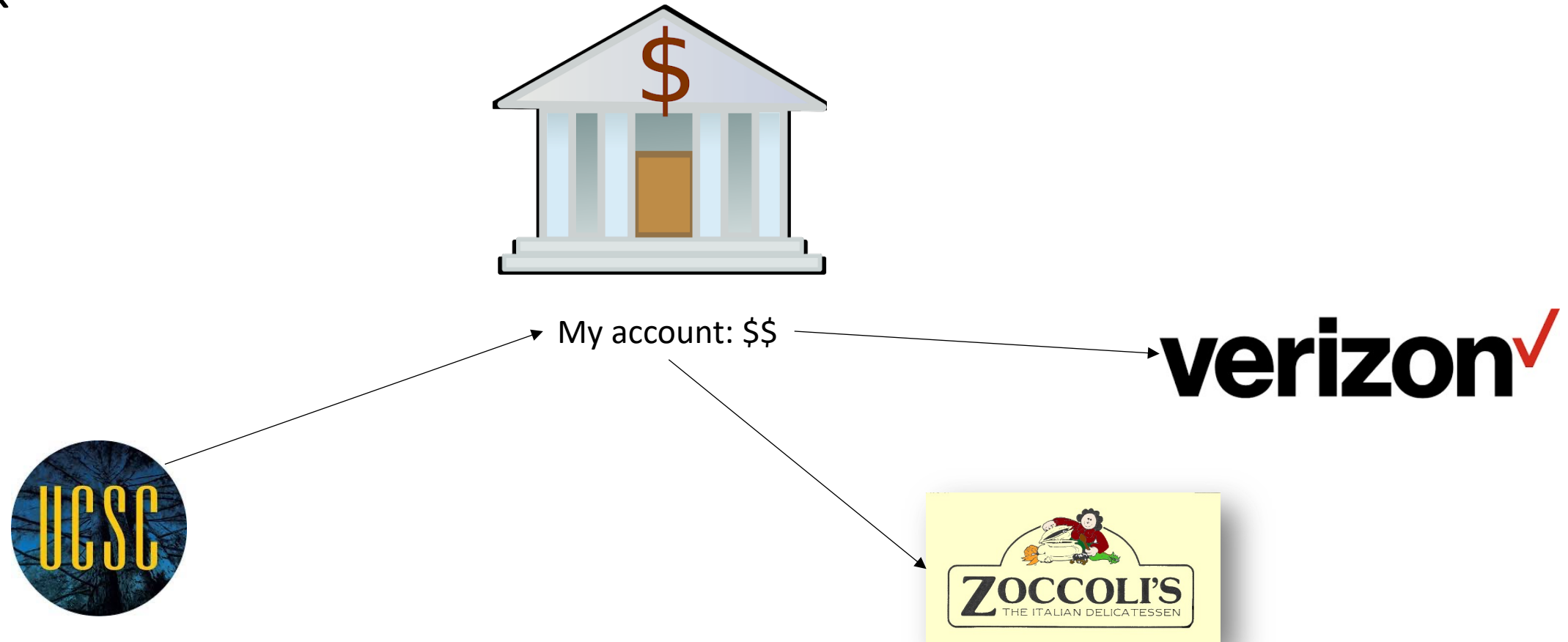
My account: \$\$

verizon✓



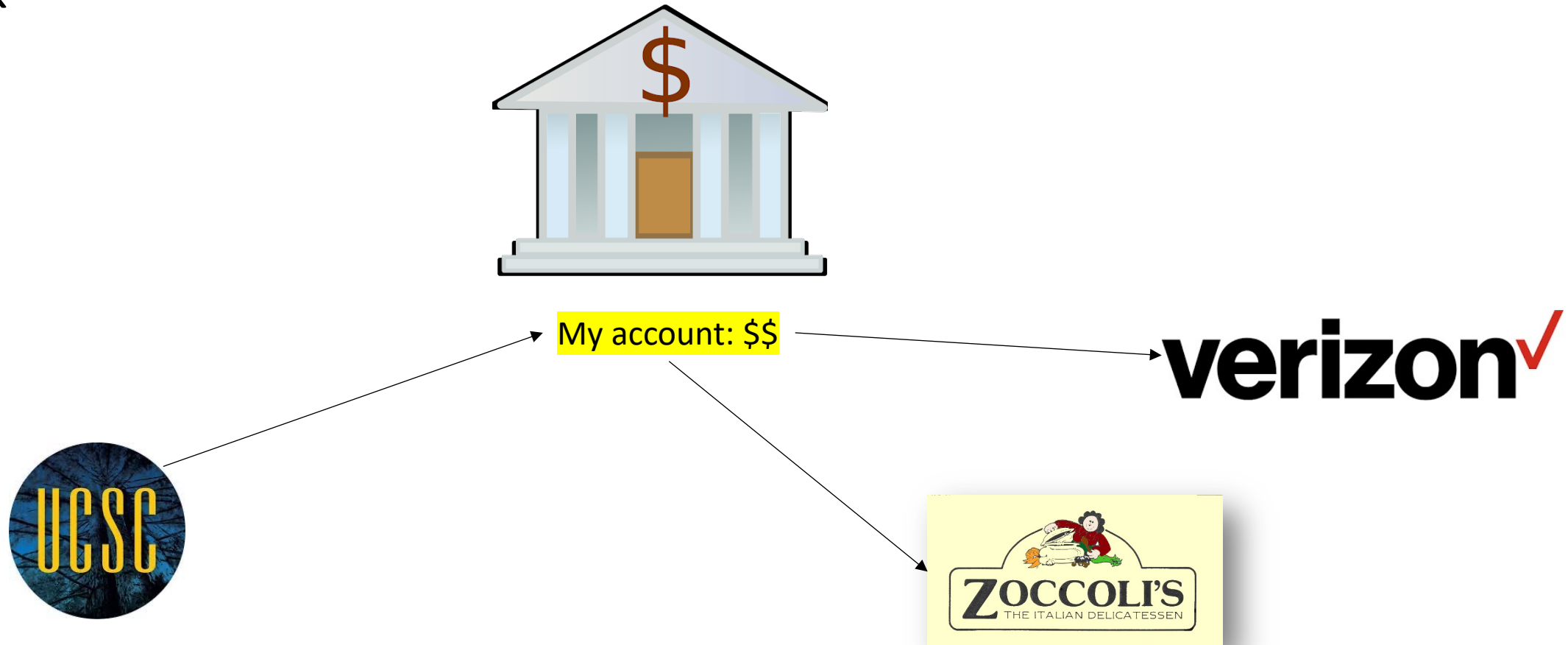
We need a way how to safely share memory

- Bank



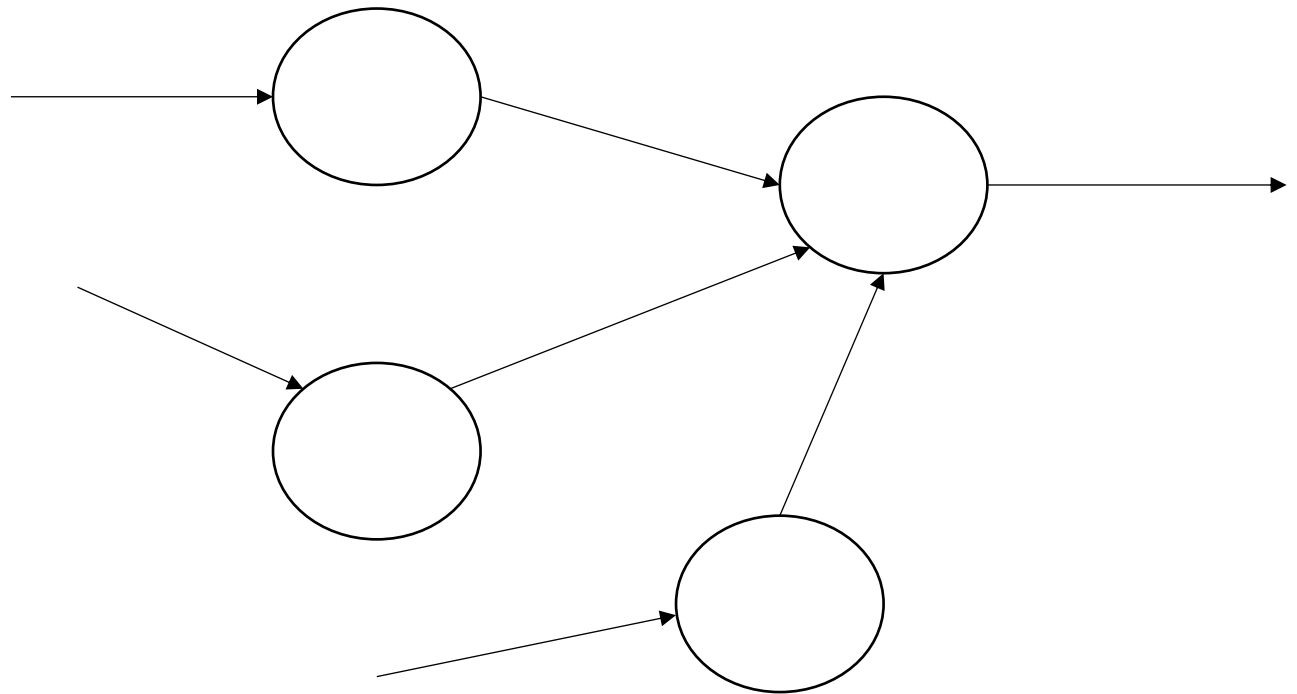
We need a way how to safely share memory

- Bank



We need a way how to safely share memory

- Graph algorithms

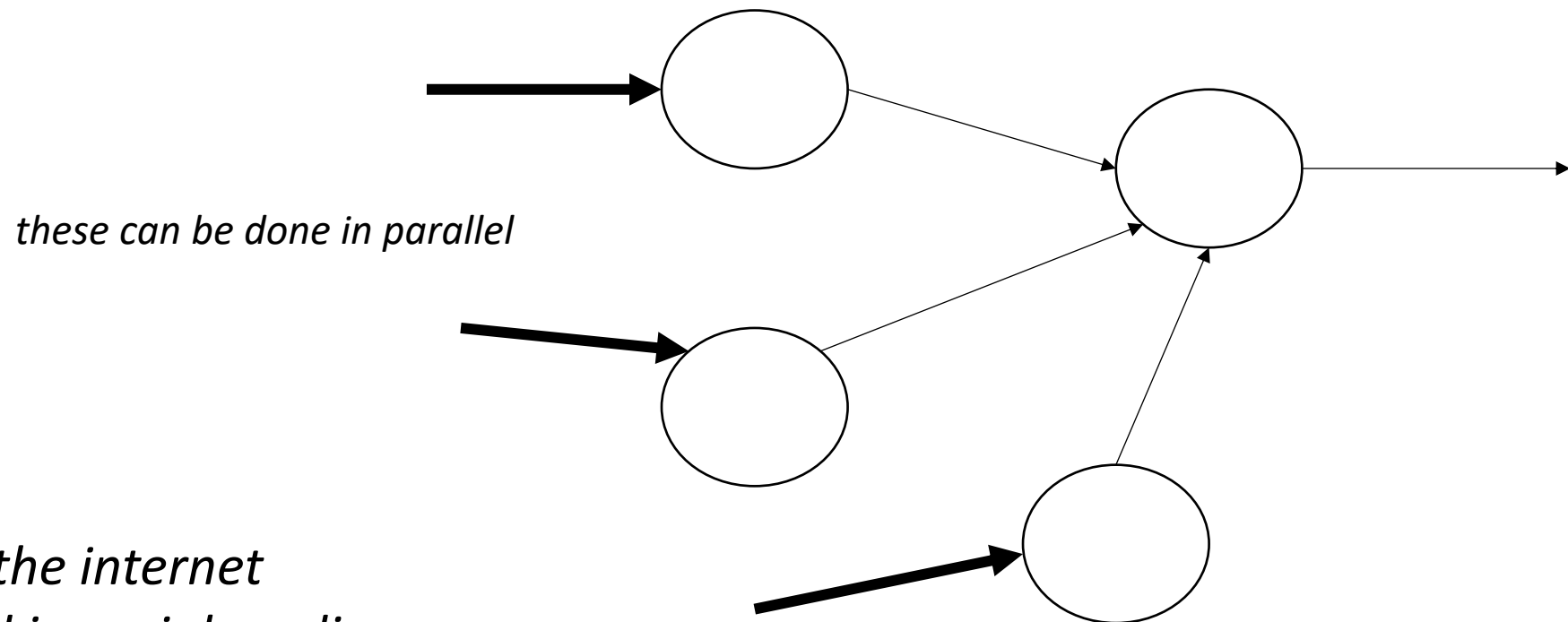


Examples:

*Ranking pages on the internet
information spread in social media*

We need a way how to safely share memory

- Graph algorithms



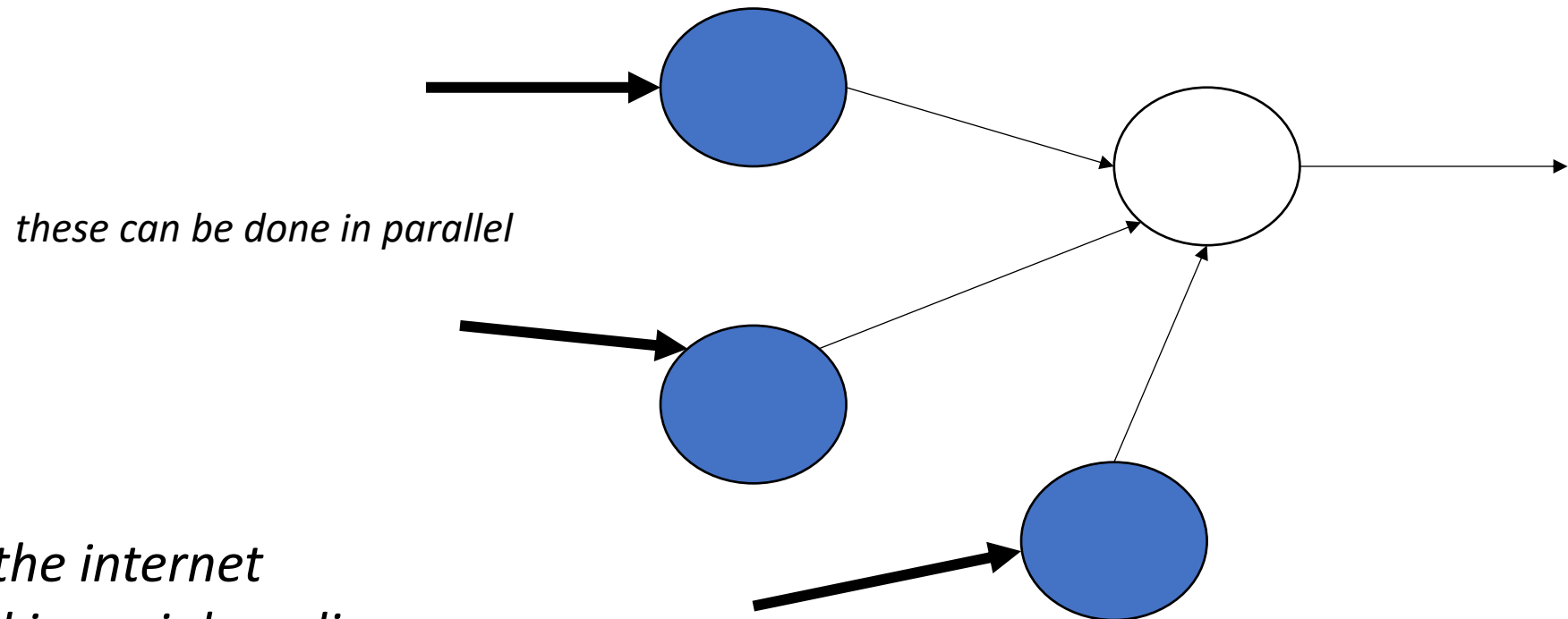
Examples:

Ranking pages on the internet

information spread in social media

We need a way how to safely share memory

- Graph algorithms



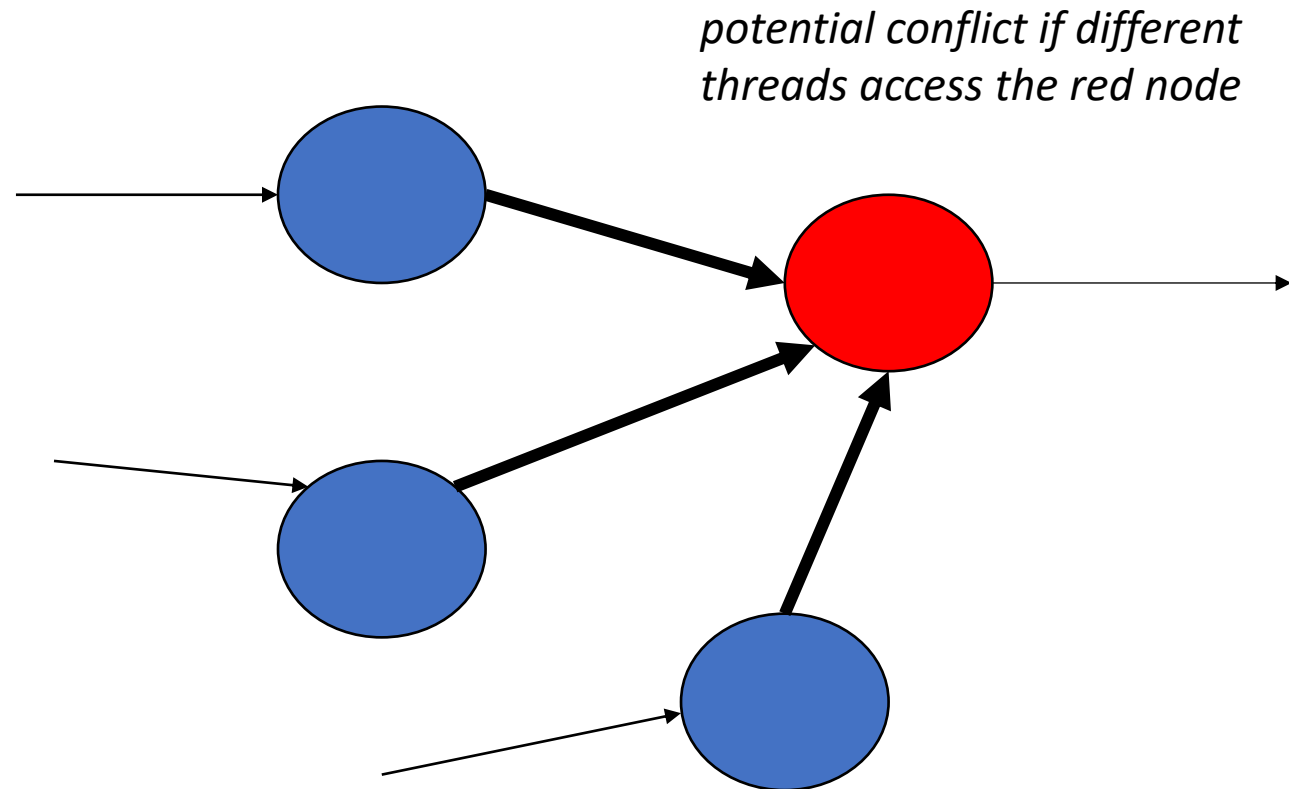
Examples:

Ranking pages on the internet

information spread in social media

We need a way how to safely share memory

- Graph algorithms



Examples:

*Ranking pages on the internet
information spread in social media*

We need a way how to safely share memory

- Machine Learning

"Dot Product"

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix} = \begin{bmatrix} 58 \\ \end{bmatrix}$$

Lots of machine learning is some form of matrix multiplication

We need a way how to safely share memory

- Machine Learning

conflict!

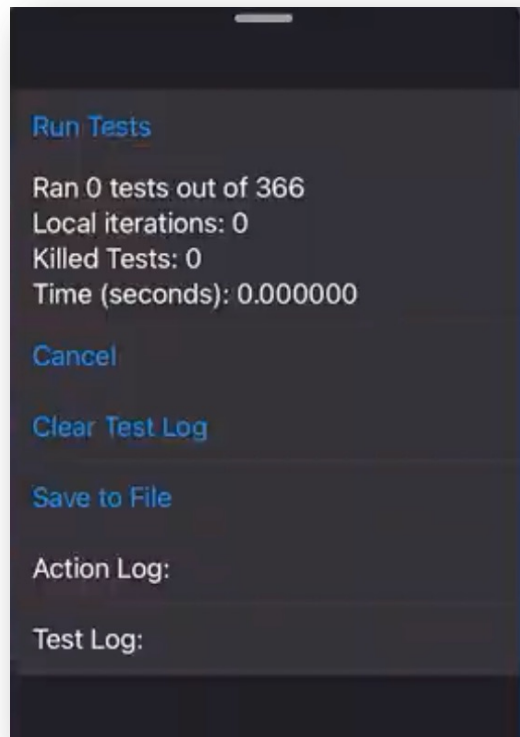
"Dot Product"

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix} = \begin{bmatrix} 58 & \dots \\ \dots & \dots \end{bmatrix}$$

Lots of machine learning is some form of matrix multiplication

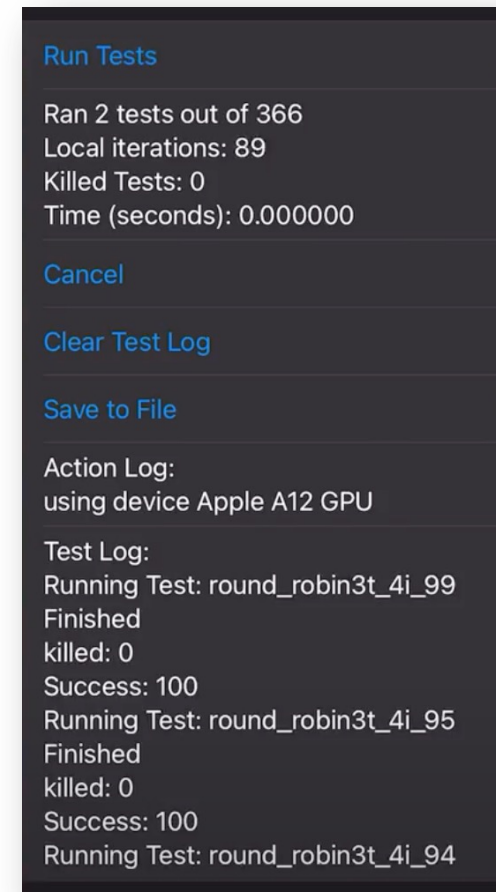
We need a way how to safely share memory

- User interfaces



*background process
that provides progress
updates to the UI.*

*UI updates must be
synchronized!!*



Lecture Schedule

- Introduction to thread-level parallelism
- **Data conflicts**
- Mutual exclusion

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.

Concurrent bank scenario

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



time

Reasoning about concurrency

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

Code demo

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

Tyler's coffee addiction:

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

Tyler's employer

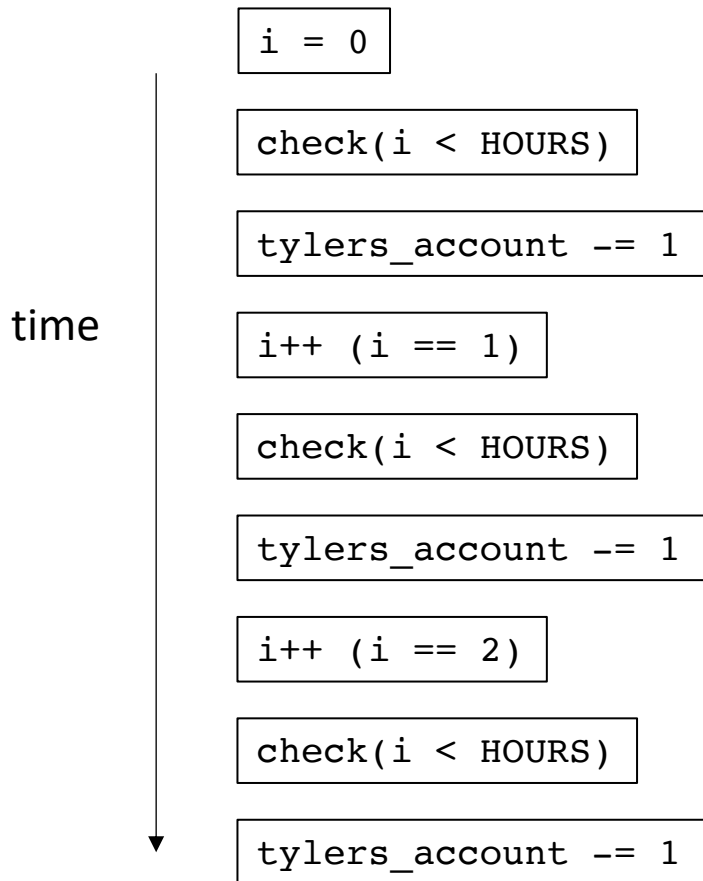
```
for (int j = 0; j < HOURS; j++) {  
    tylers_account += 1;  
}
```

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;  
}
```



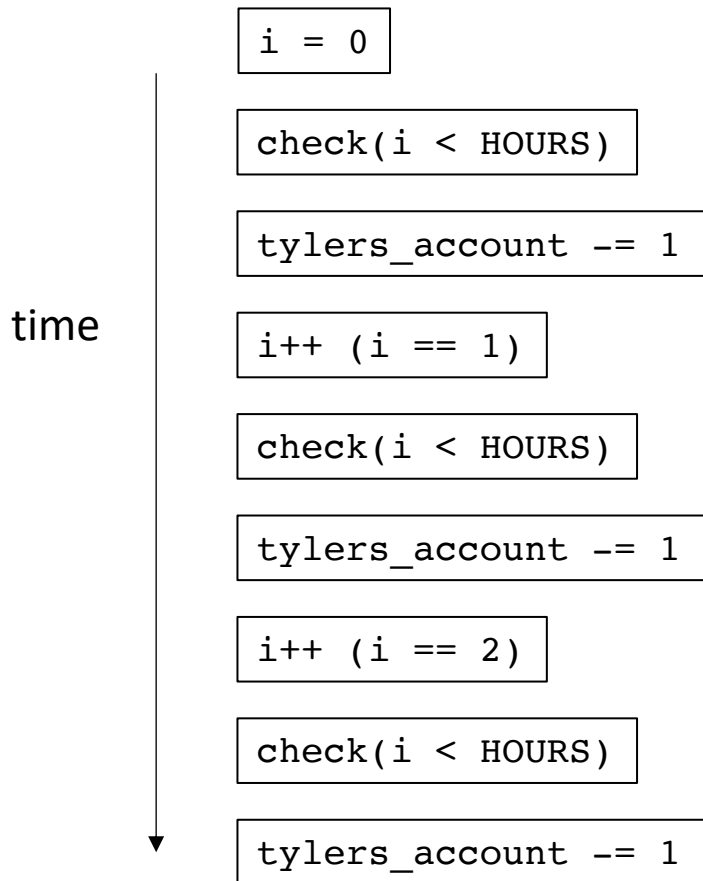
Tyler's employer

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

A thread is a sequential program

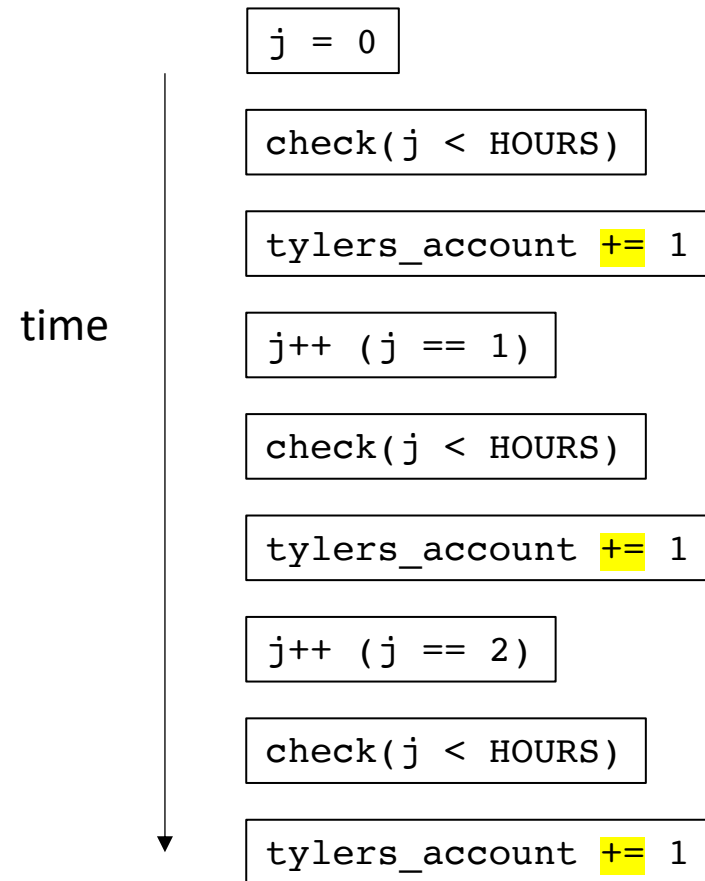
Tyler's coffee addiction:

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



Tyler's employer

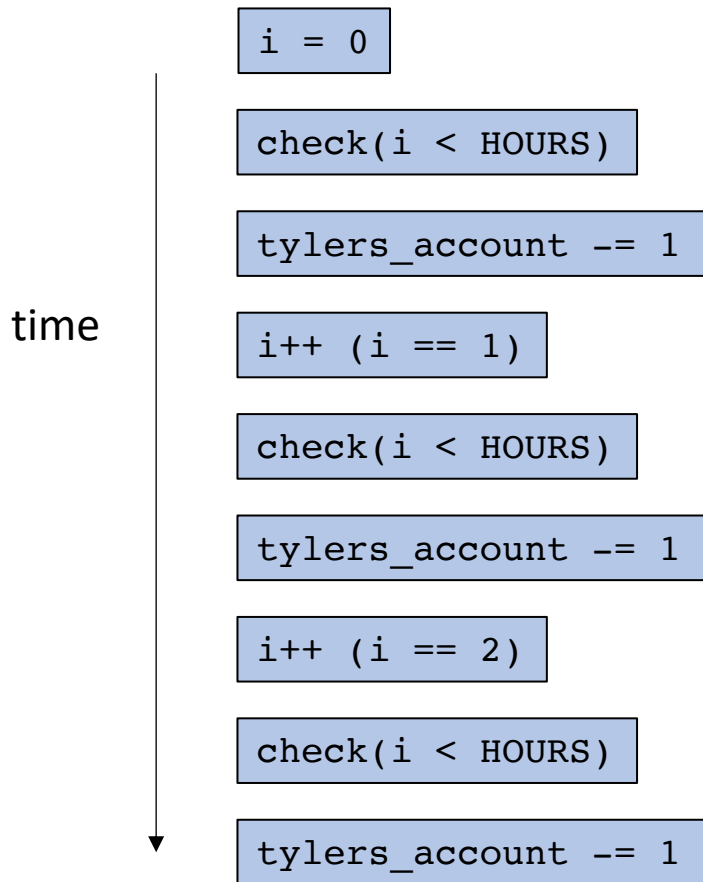
```
for (int j = 0; j < HOURS; j++) {  
    tylers_account += 1;  
}
```



A thread is a sequential program

Tyler's coffee addiction:

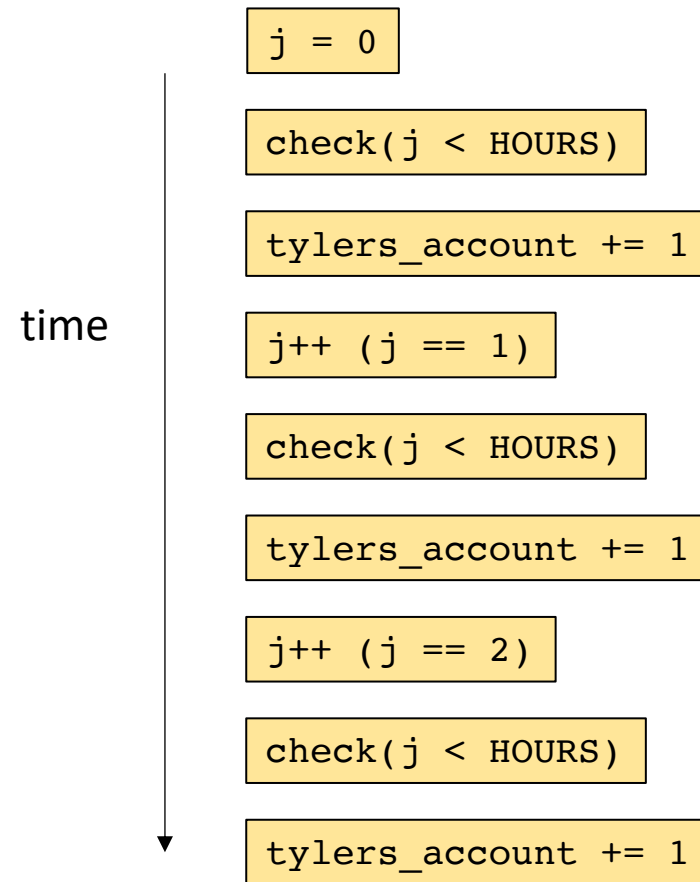
```
for (int i = 0; i < HOURS; i++) {  
    tylers_account -= 1;  
}
```



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

Tyler's employer

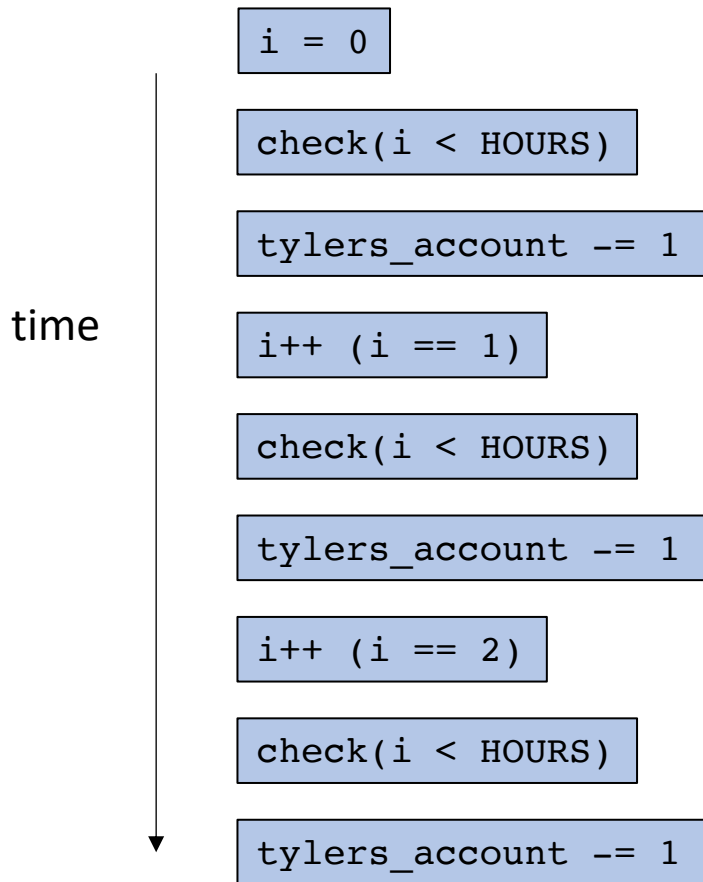
```
for (int j = 0; j < HOURS; j++) {  
    tylers_account += 1;  
}
```



A thread is a sequential program

Tyler's coffee addiction:

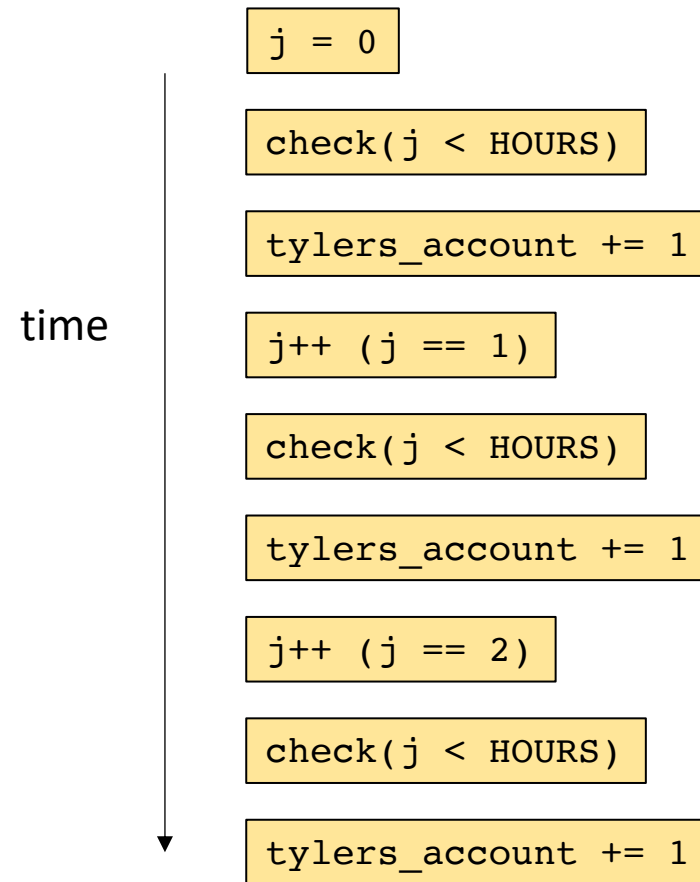
```
for (int i = 0; i < HOURS; i++) {  
    tylers_account -= 1;  
}
```



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

```
i = 0
```

```
check(i < HOURS)
```

```
tylers_account -= 1
```

```
i++ (i == 1)
```

```
check(i < HOURS)
```

time

```
j = 0
```

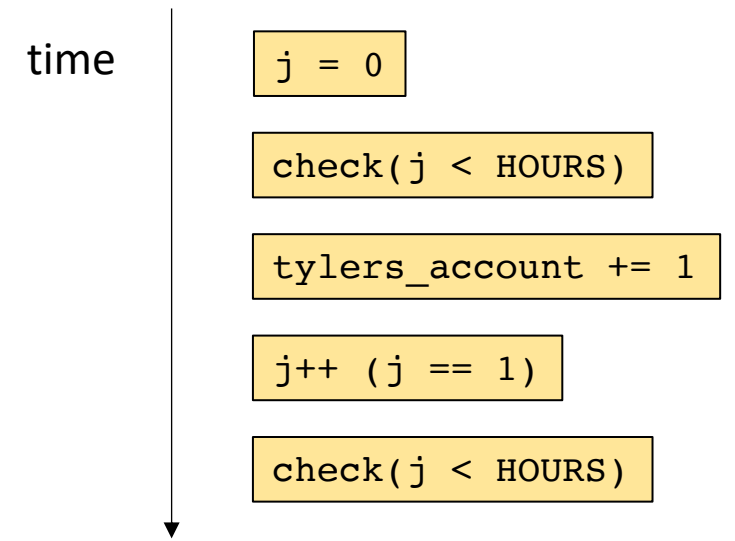
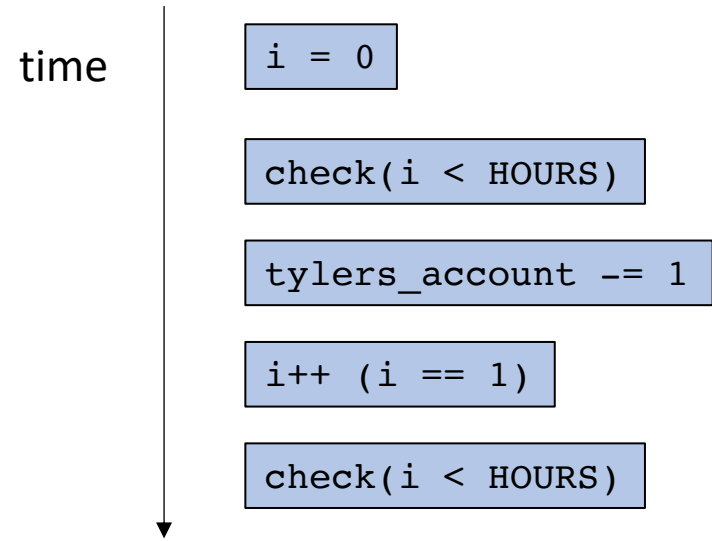
```
check(j < HOURS)
```

```
tylers_account += 1
```

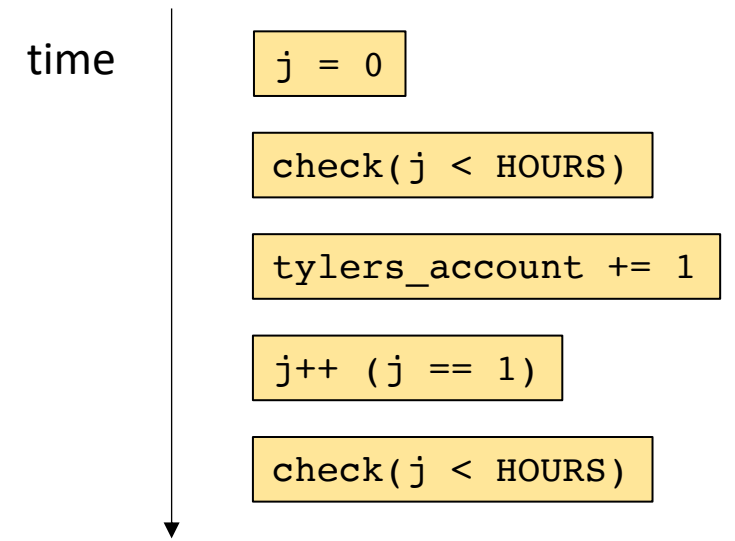
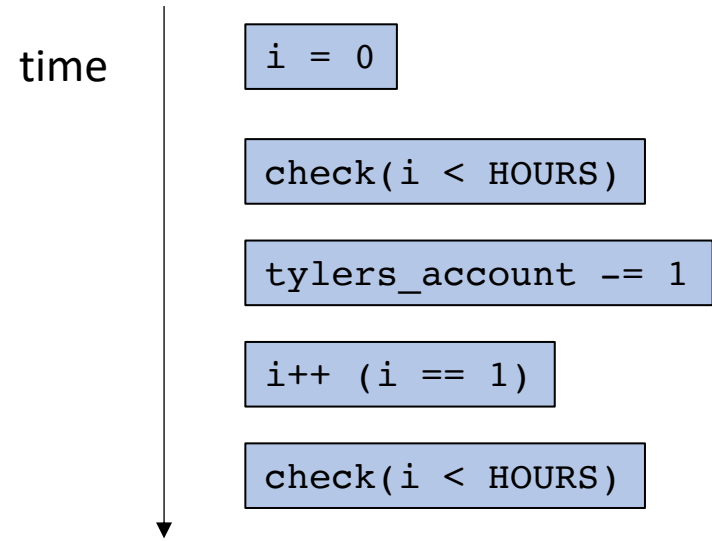
```
j++ (j == 1)
```

```
check(j < HOURS)
```

consider just one loop iteration

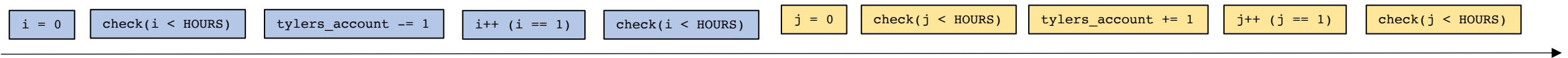


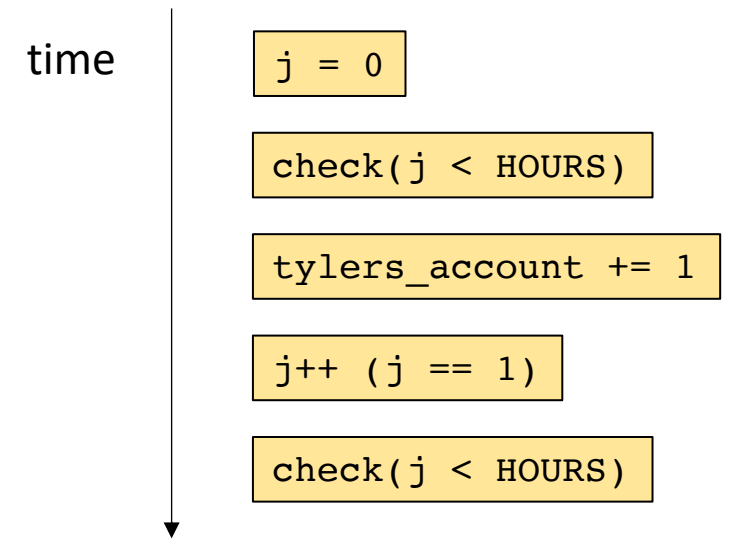
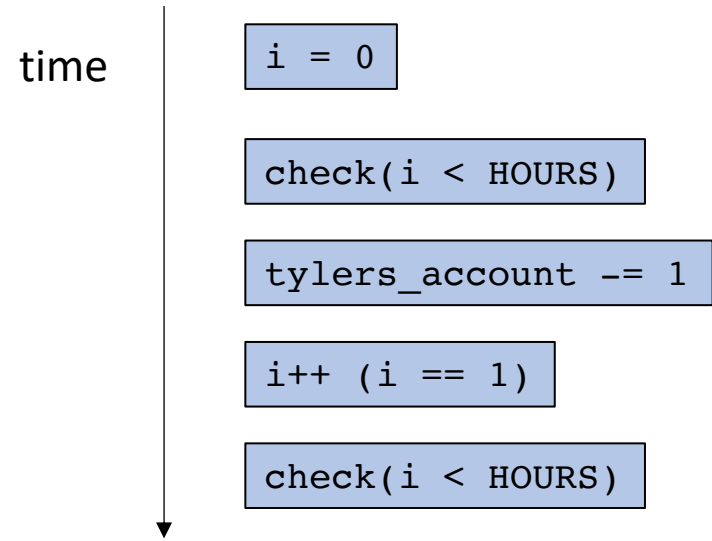
Concurrent execution



one possible execution

Concurrent execution





one possible execution

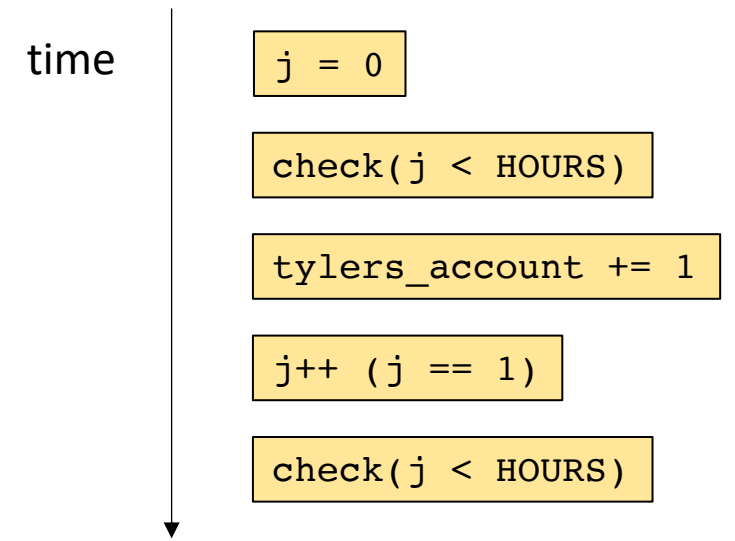
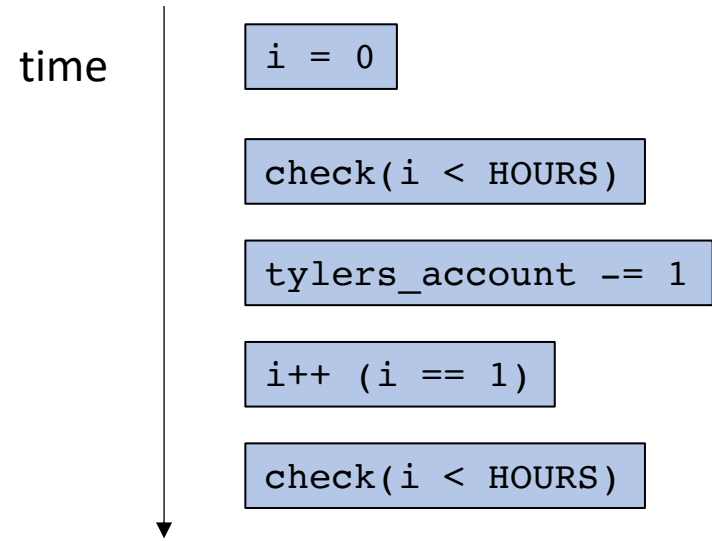
Concurrent execution



tyler_account: 0

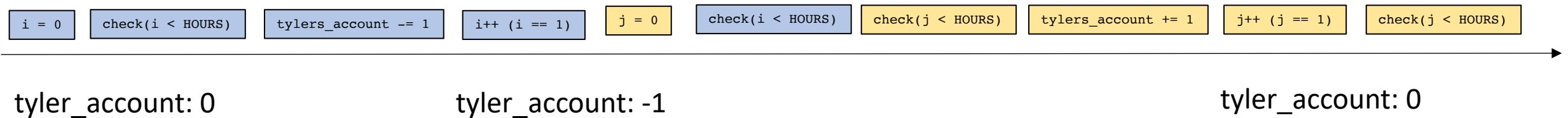
tyler_account: -1

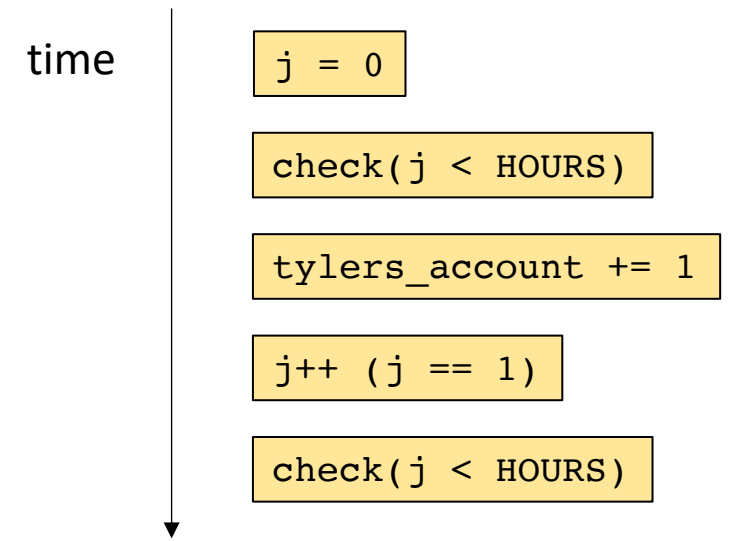
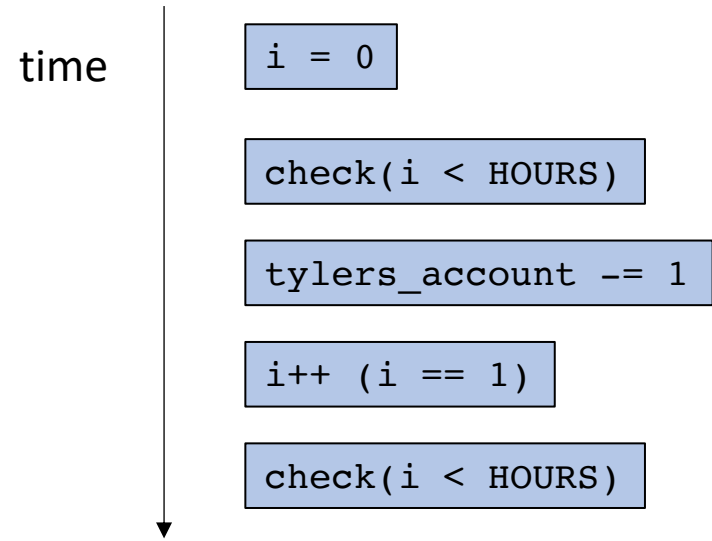
tyler_account: 0



Another possible execution

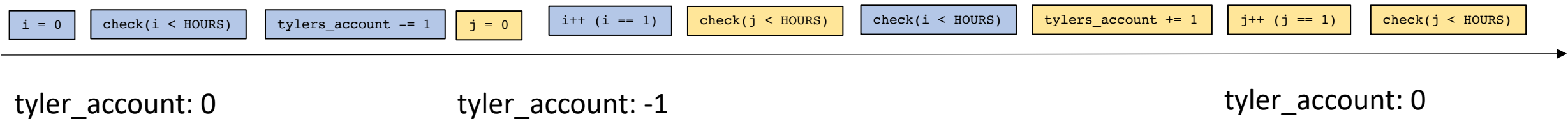
Concurrent execution

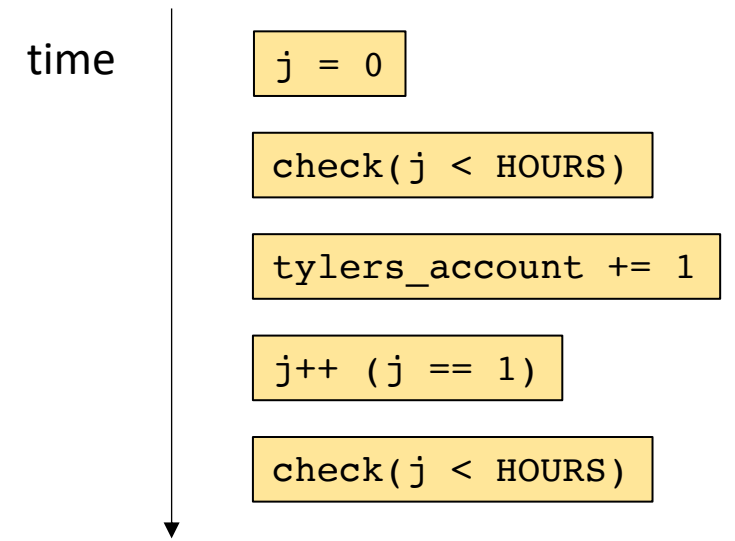
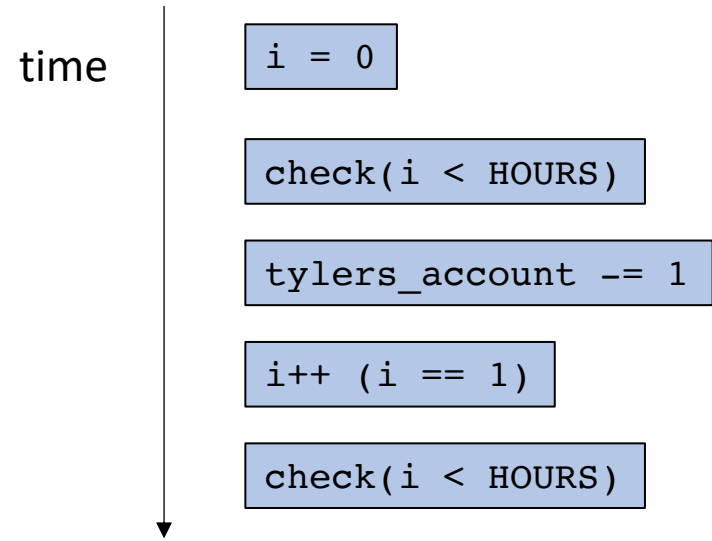




Another possible execution

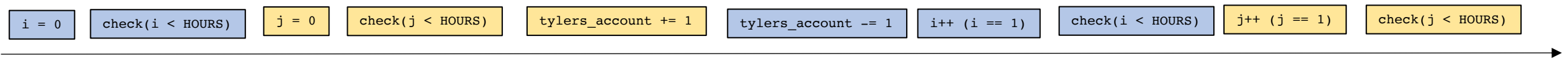
Concurrent execution

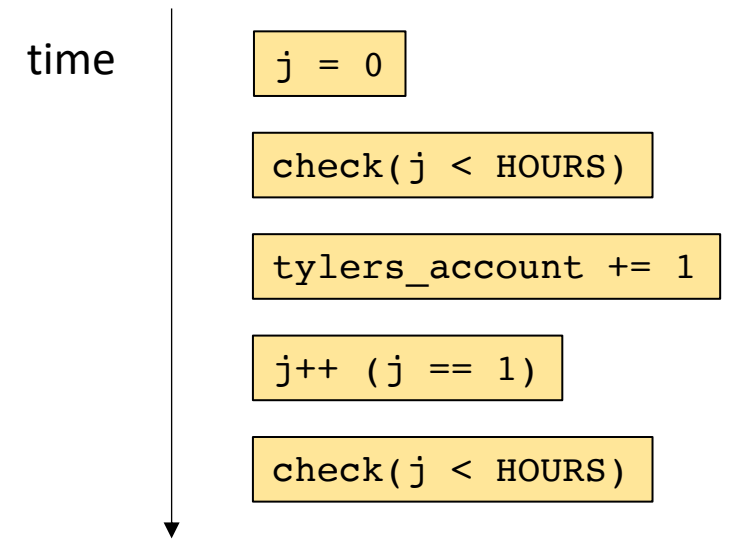
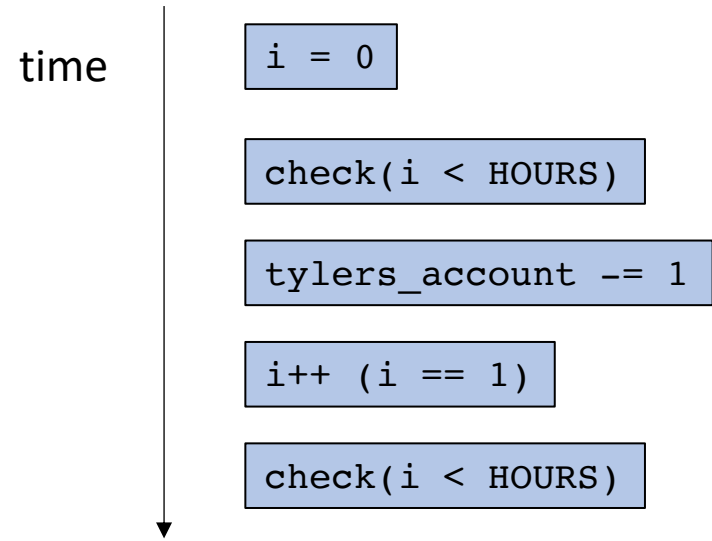




Another possible execution

Concurrent execution

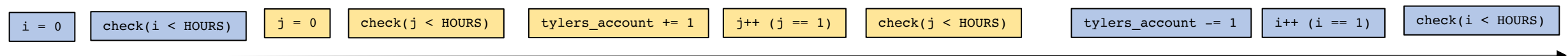




Another possible execution

This time my account isn't ever negative

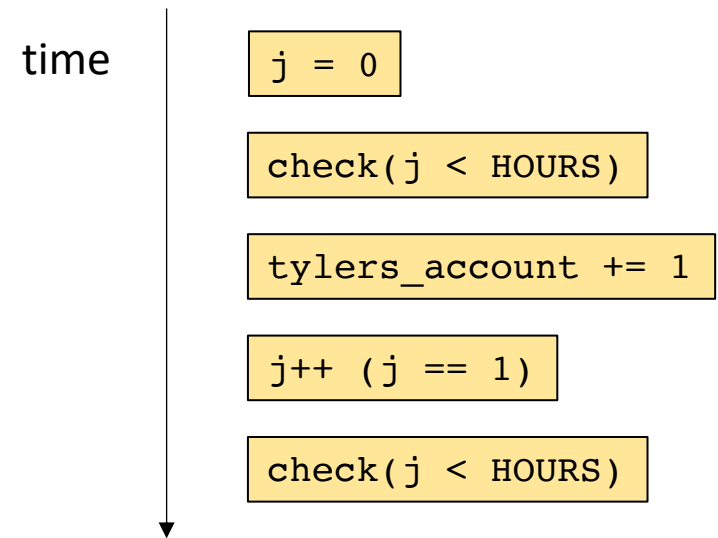
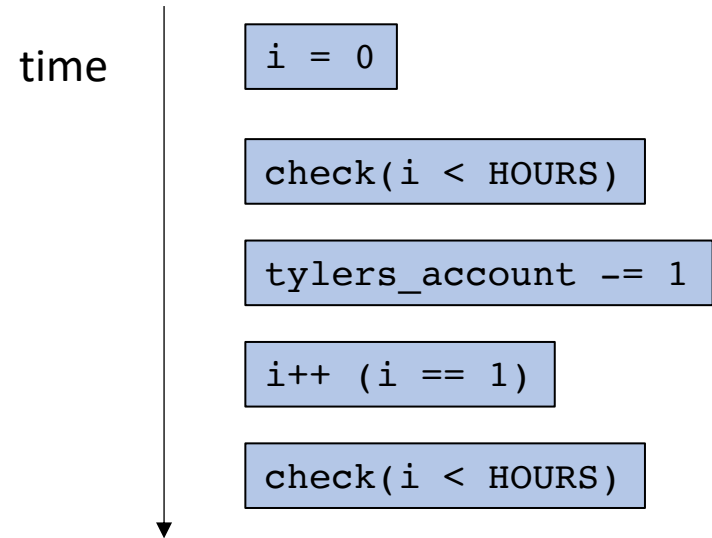
Concurrent execution



tyler_account: 0

tyler_account: 1

tyler_account: 0



How many possible interleavings?

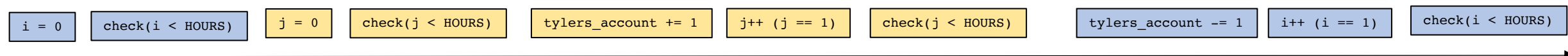
Combinatorics question:

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!



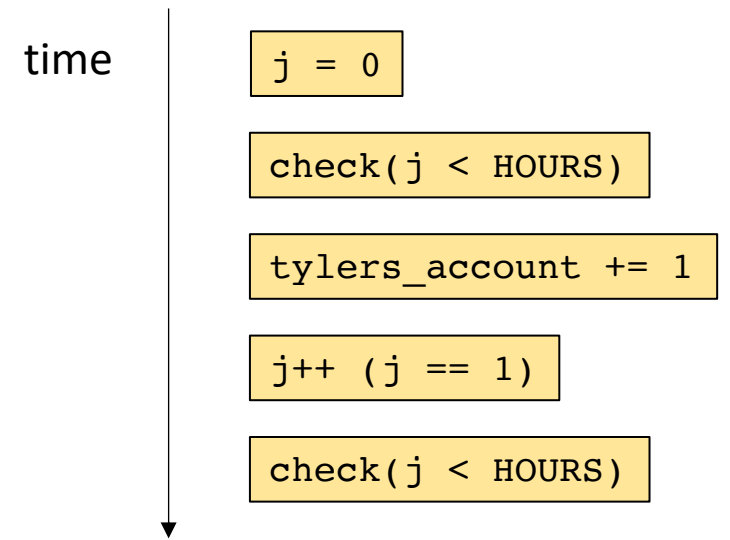
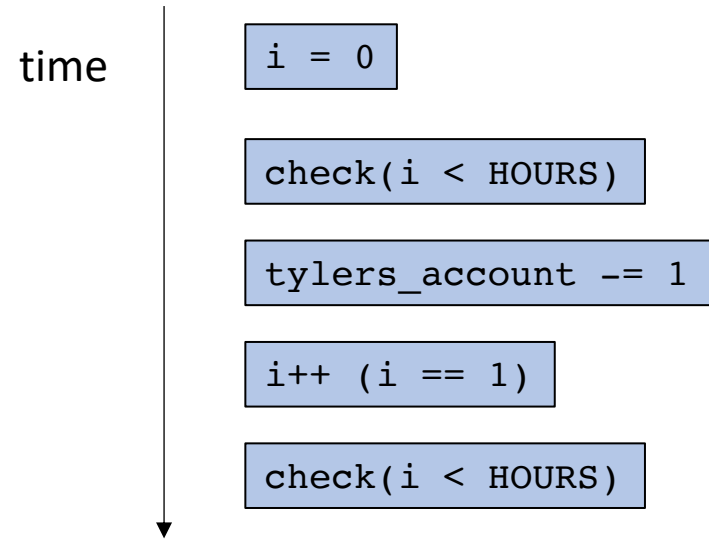
tyler_account: 0

tyler_account: 1

tyler_account: 0

Reasoning about concurrency

- Not feasible to think about all interleavings!
 - Lots of interesting research in pruning, testing interleavings (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:

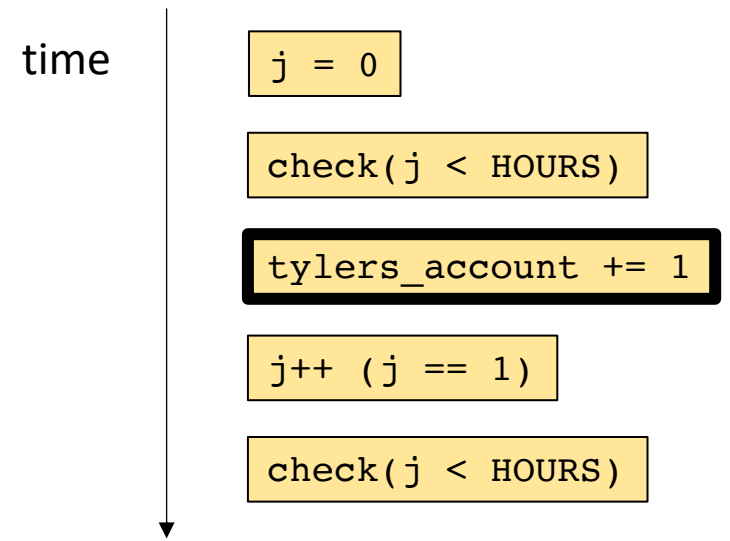
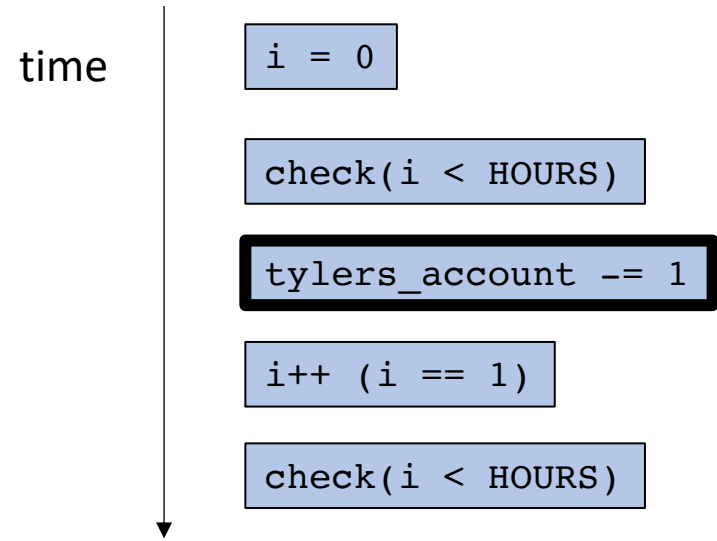
For any interleaving, both of the increase and decrease must happen in some order.

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

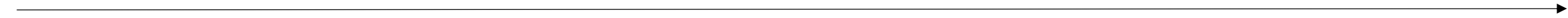
concurrent execution



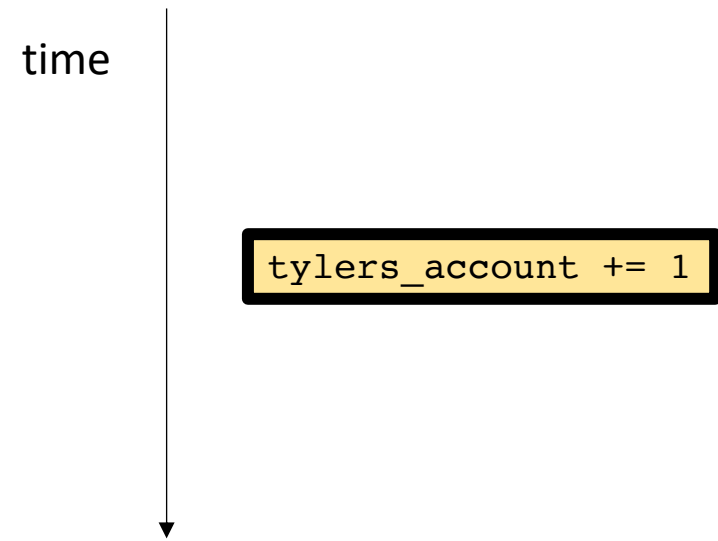
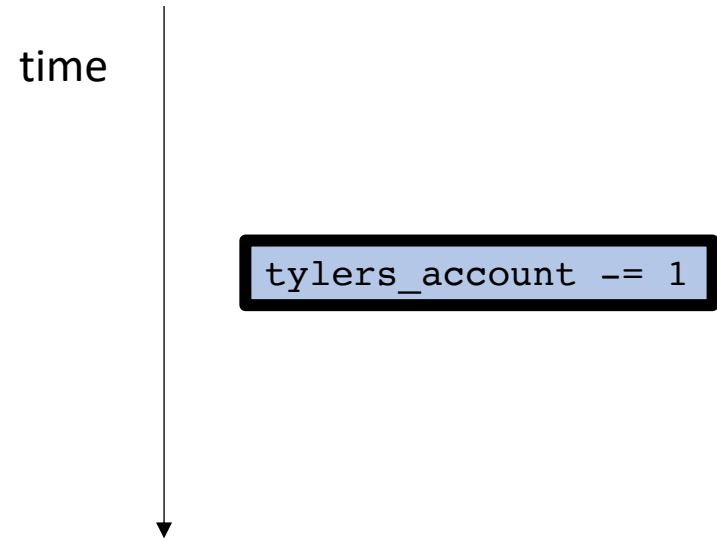
time



concurrent execution



time

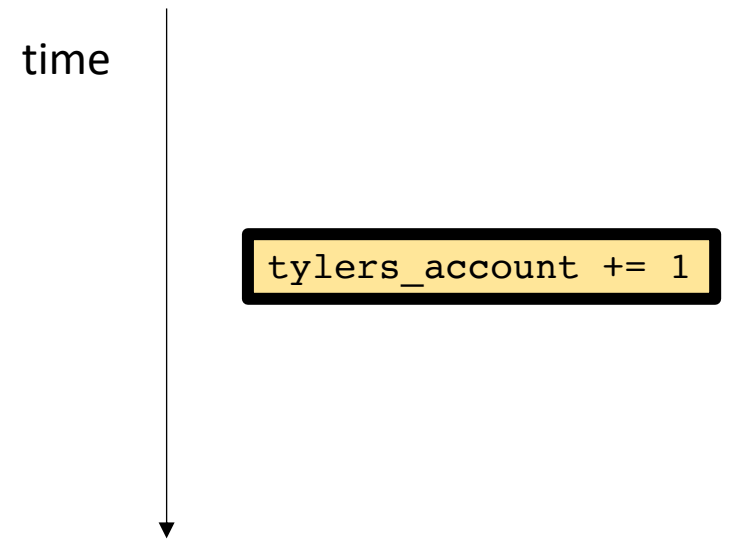
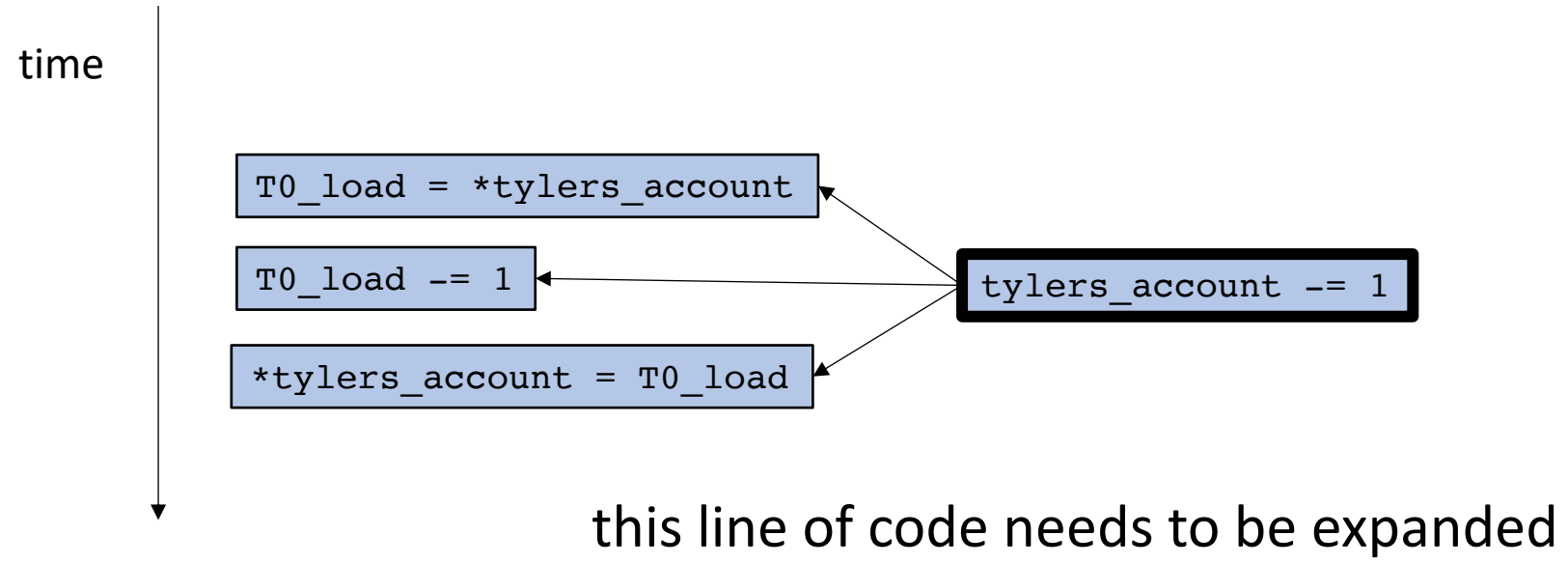


Remember 3 address code...

concurrent execution

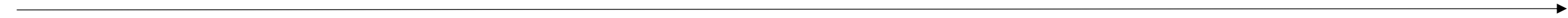


time

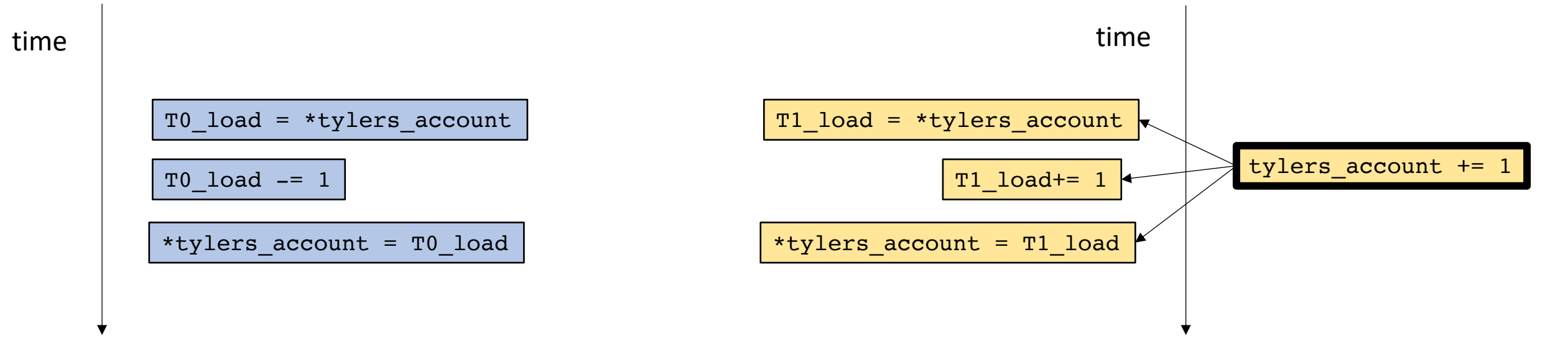


Remember 3 address code...

concurrent execution

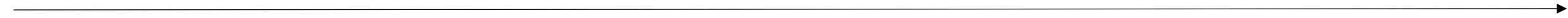


time

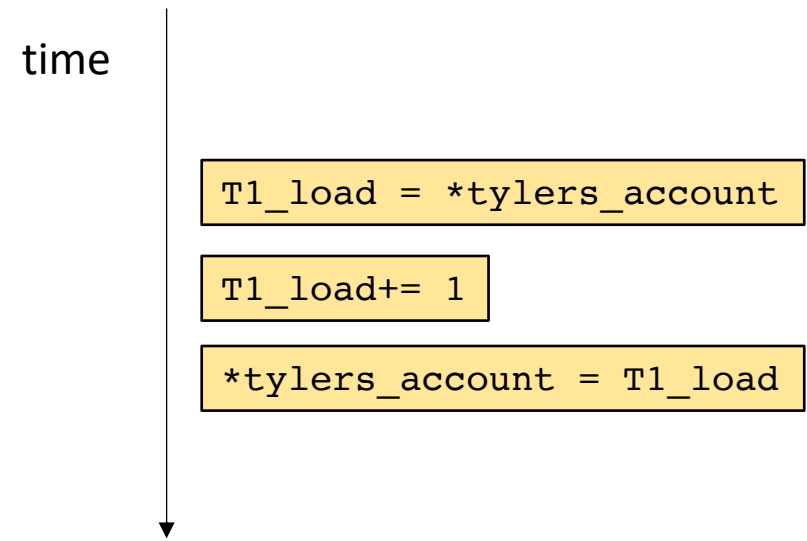
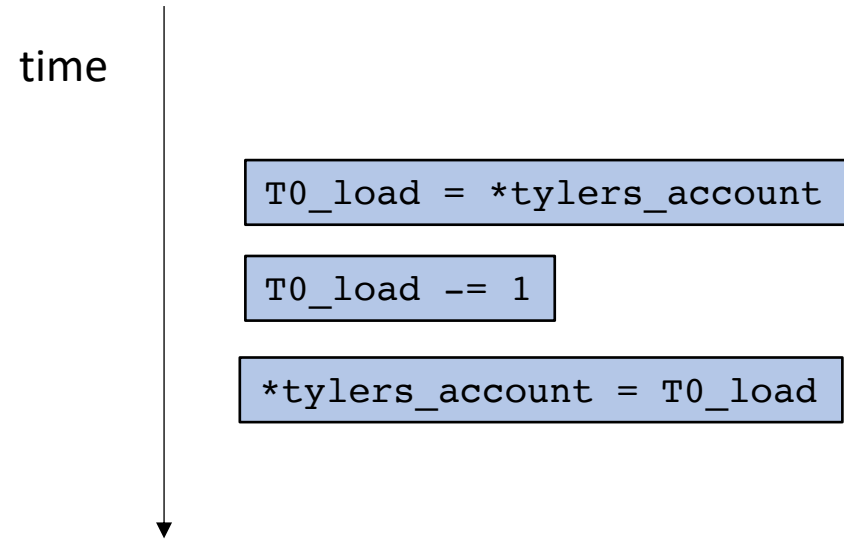


Remember 3 address code...

concurrent execution



time

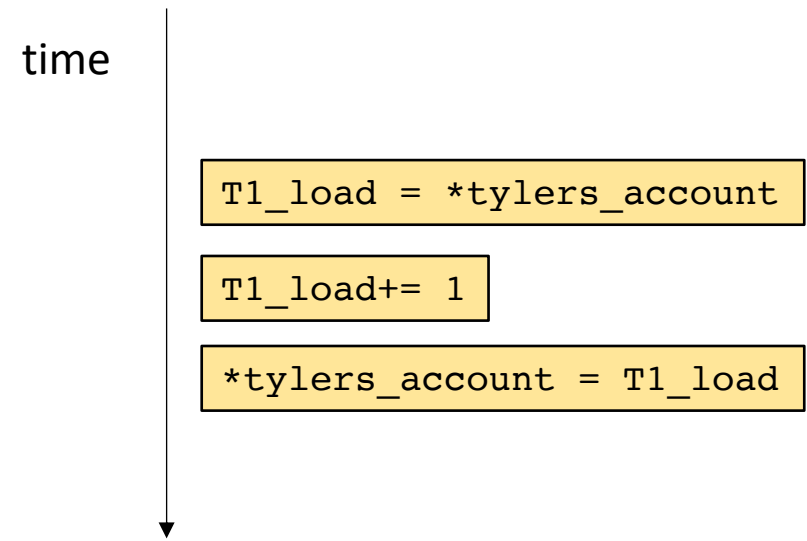
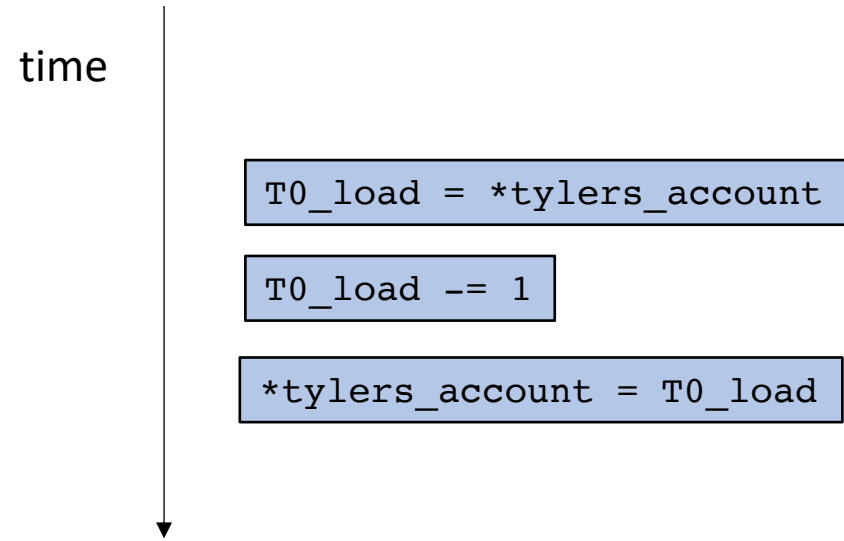


What if we interleave these instructions?

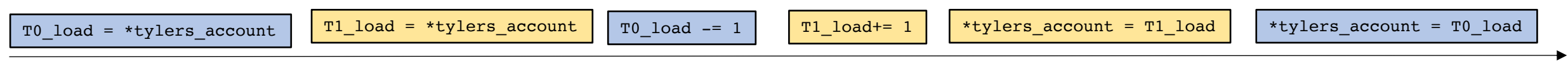
concurrent execution



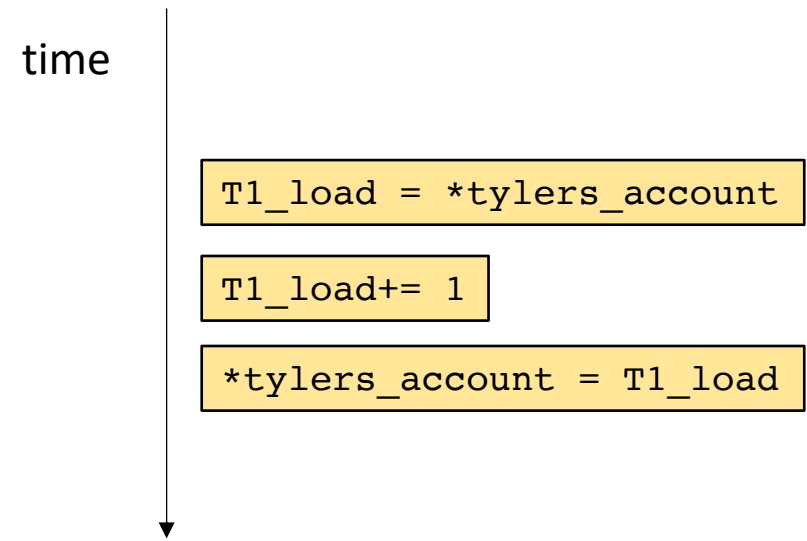
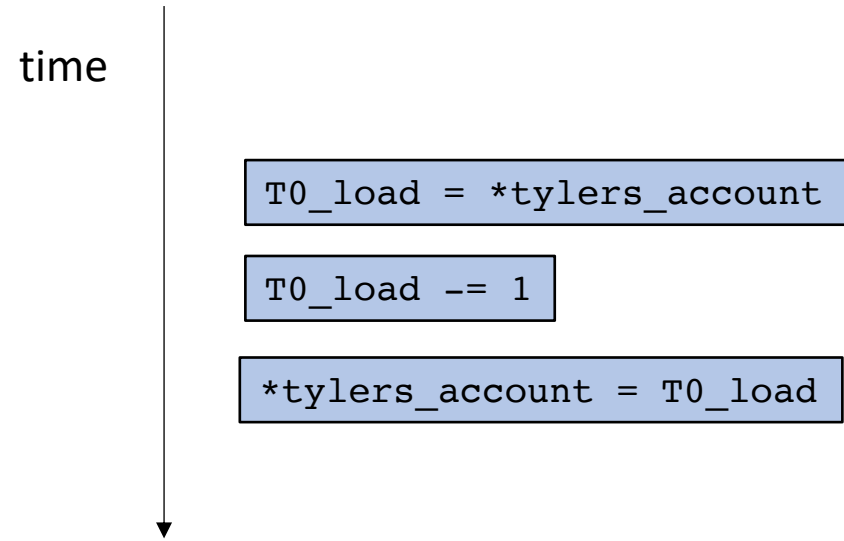
time



concurrent execution

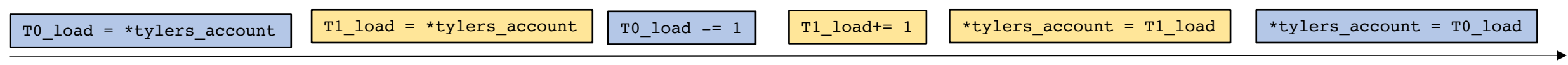


time



tylers_account has -1 at the end of this interleaving!

concurrent execution



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***.

Lecture Schedule

- Introduction to thread-level parallelism
- Data conflicts
- **Mutual exclusion**

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

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



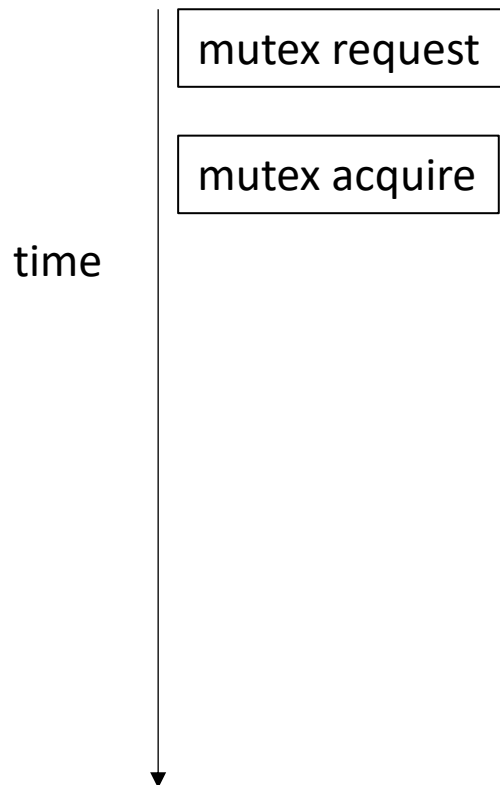
A thread is a sequential program

Tyler's coffee addiction:

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

Tyler's employer

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



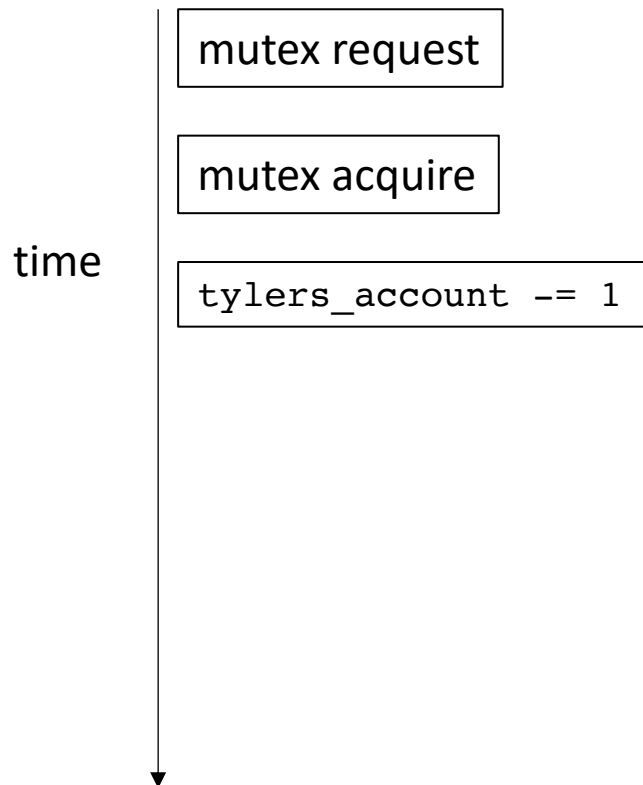
A thread is a sequential program

Tyler's coffee addiction:

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

Tyler's employer

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



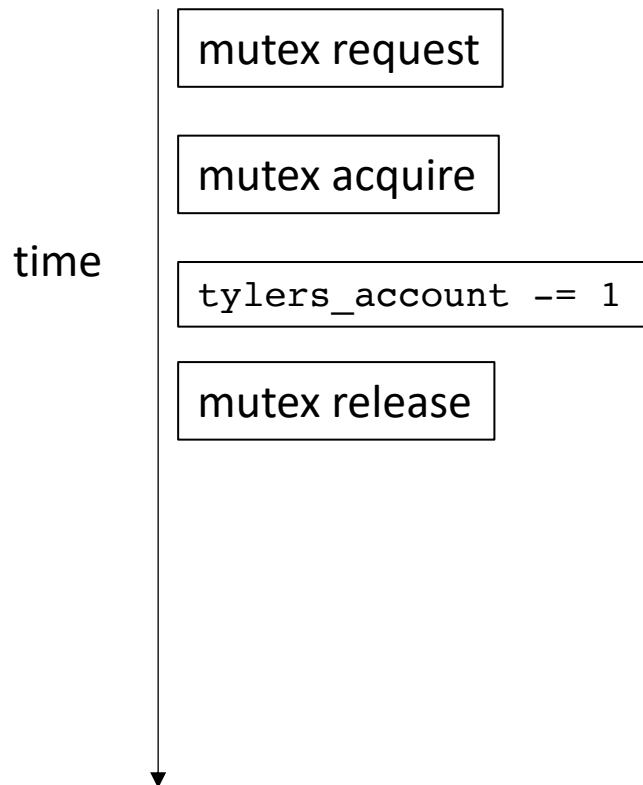
A thread is a sequential program

Tyler's coffee addiction:

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

Tyler's employer

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



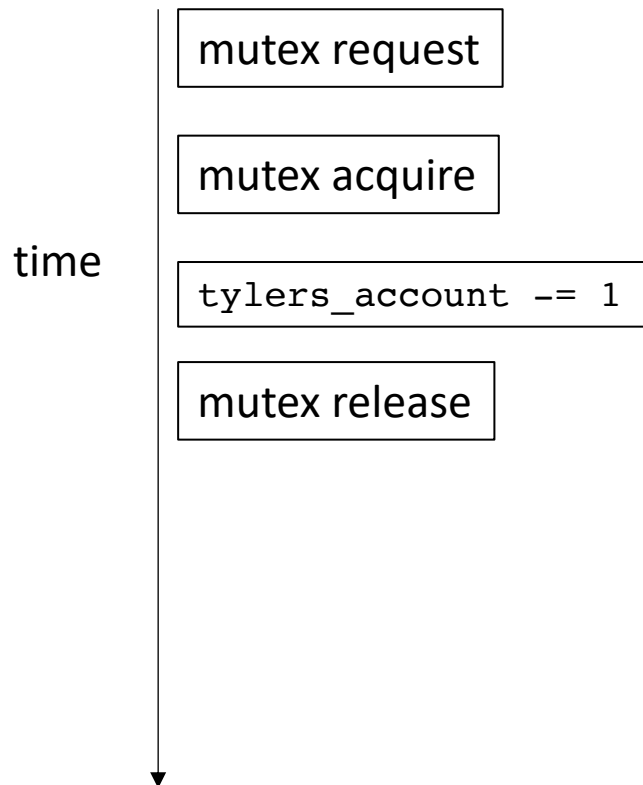
A thread is a sequential program

Tyler's coffee addiction:

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

Tyler's employer

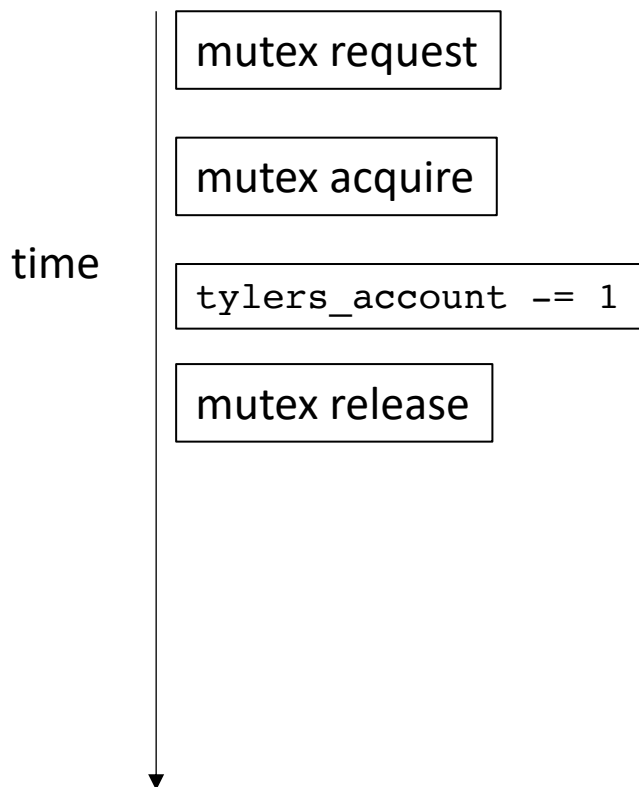
```
m.lock();  
tylers_account += 1;  
m.unlock();
```



A thread is a sequential program

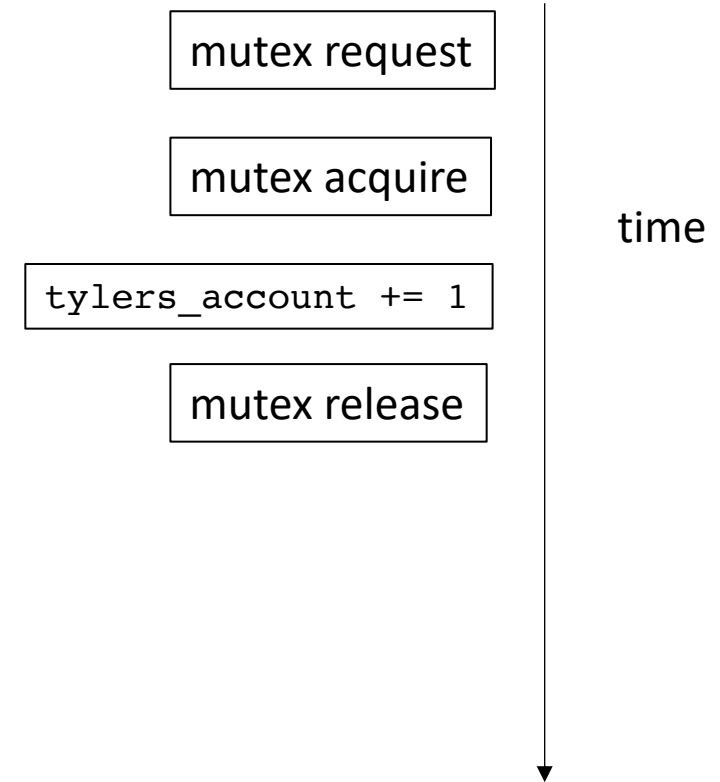
Tyler's coffee addiction:

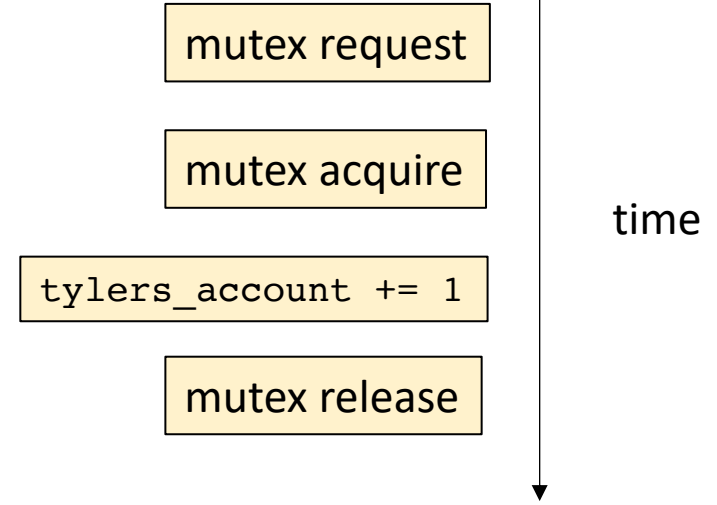
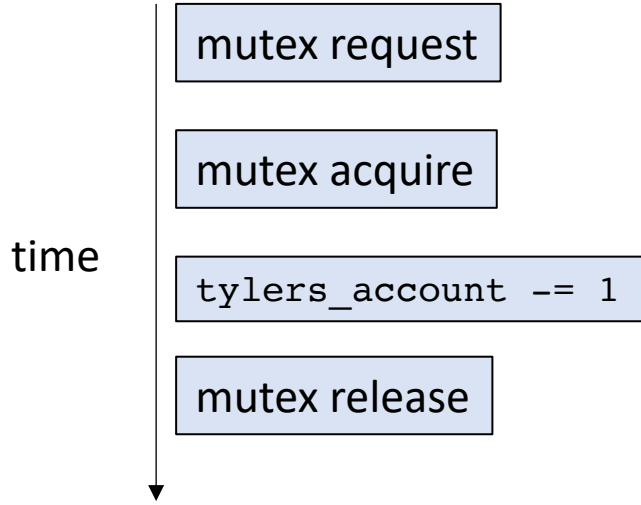
```
m.lock();  
tylers_account -= 1;  
m.unlock();
```



Tyler's employer

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

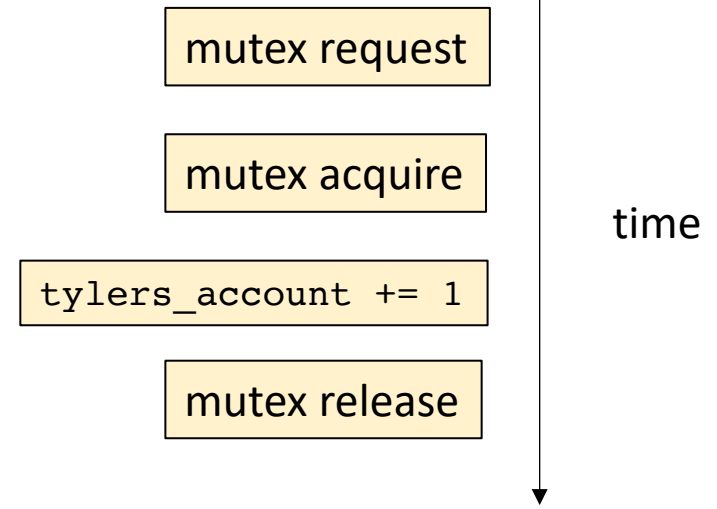
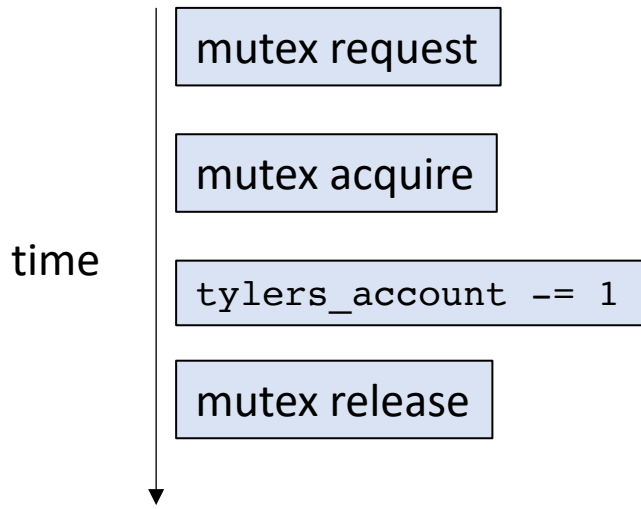




concurrent execution



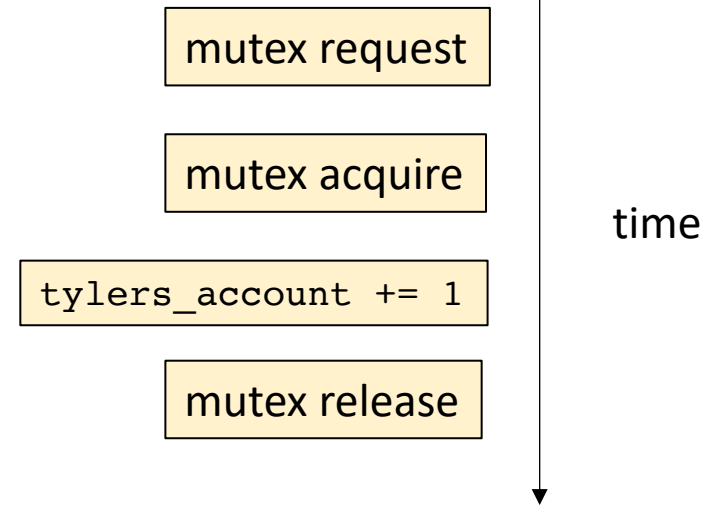
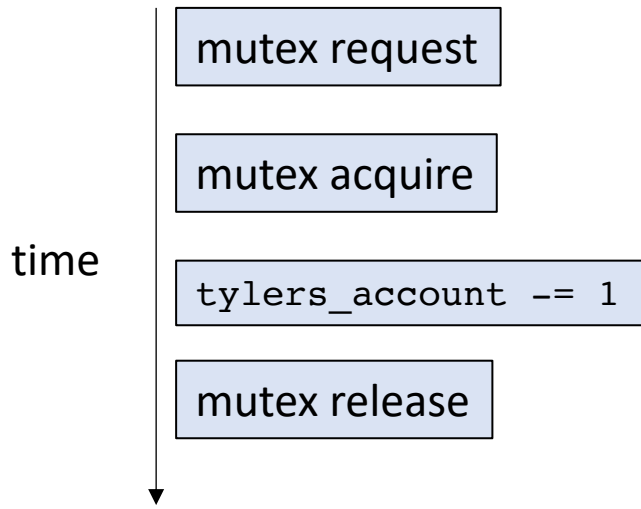
time



concurrent execution



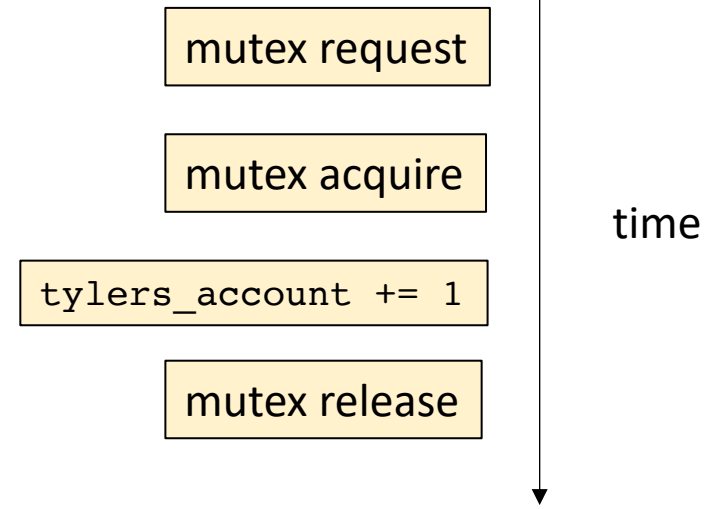
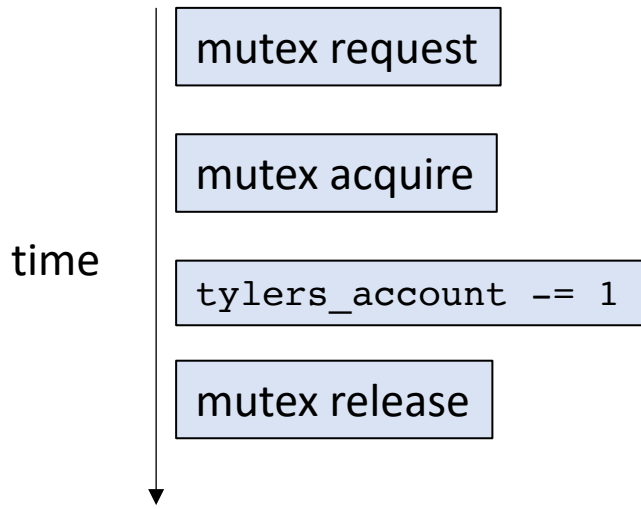
time



*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



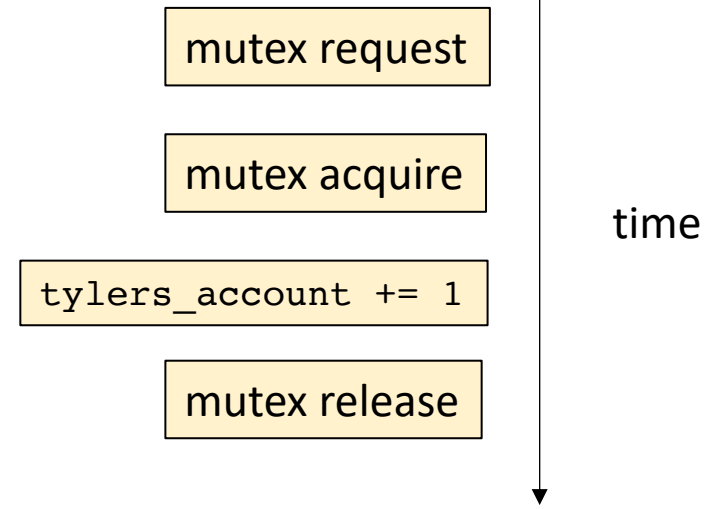
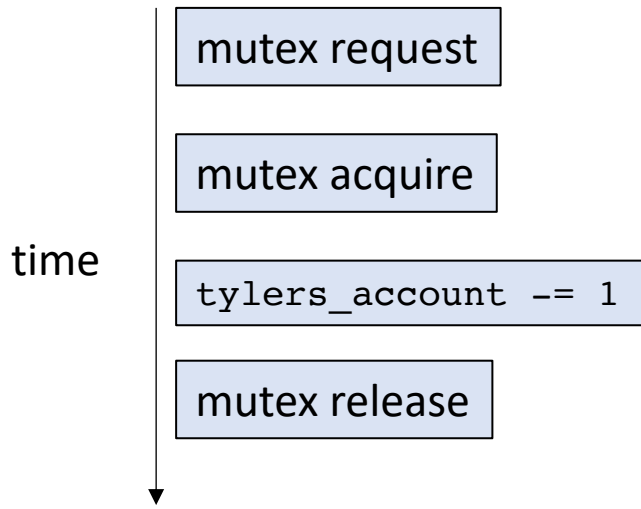


Allowed to request

concurrent execution



time



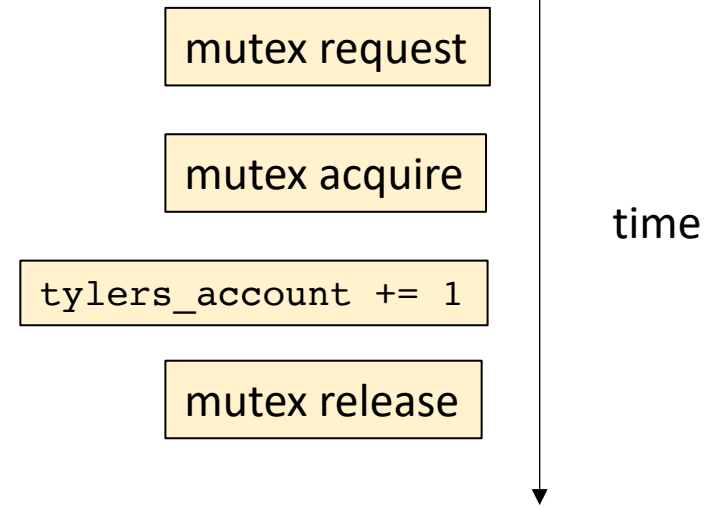
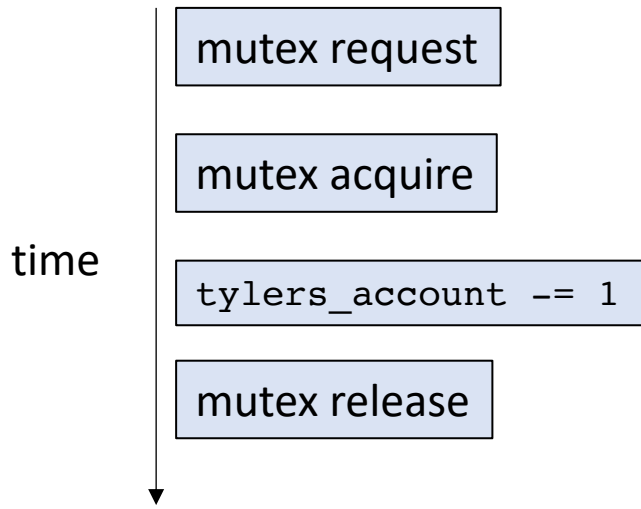
Allowed to request

concurrent execution



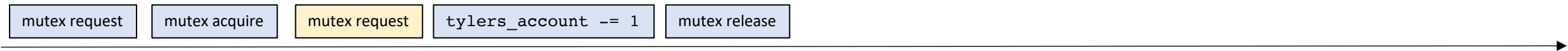
disallowed!

time

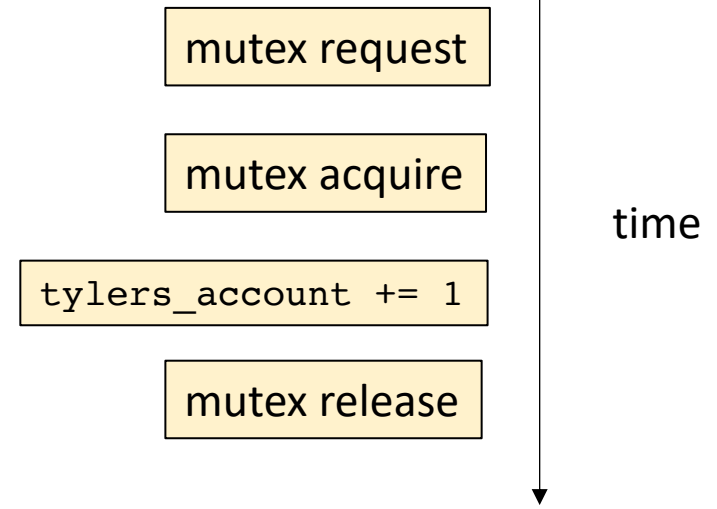
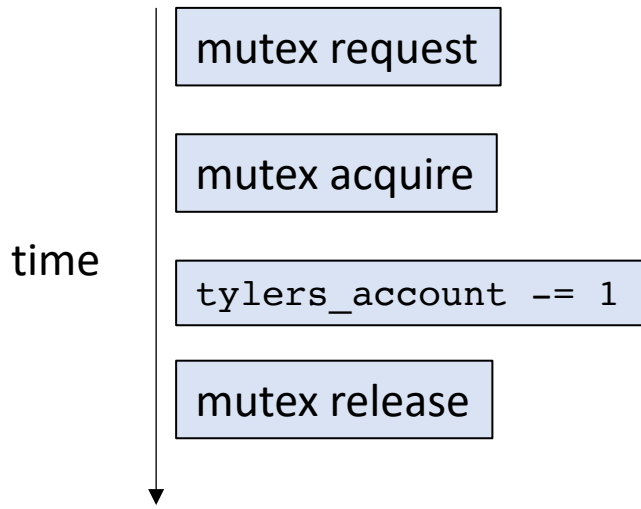


Thread 0 has released the mutex

concurrent execution

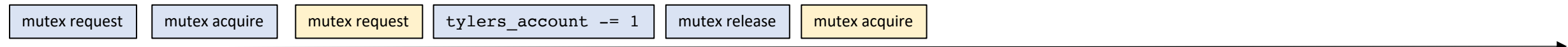


time

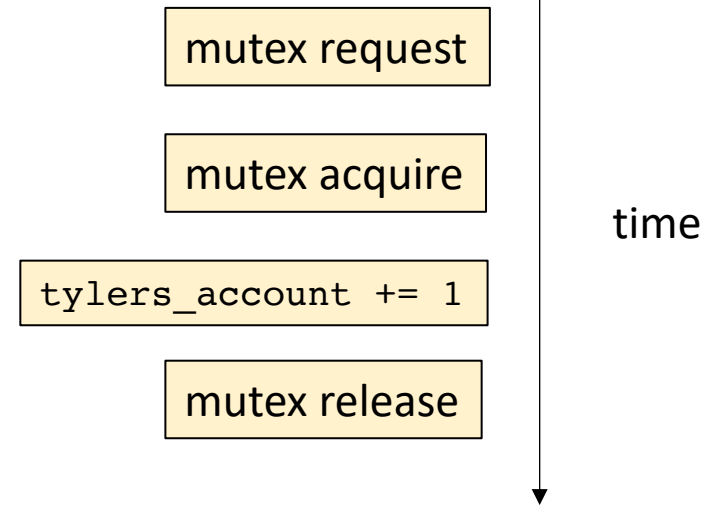
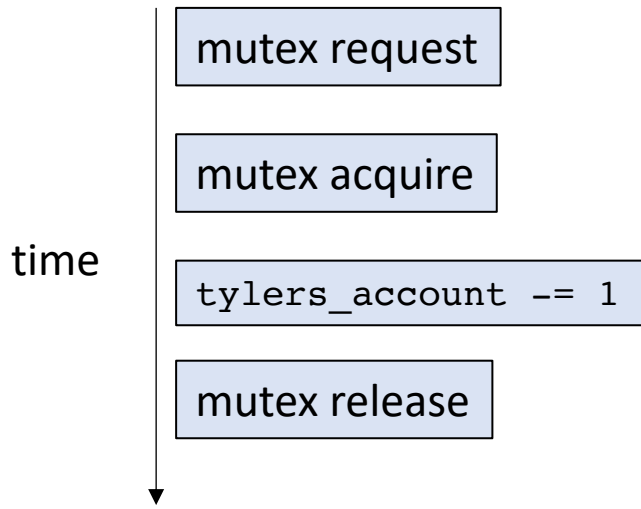


Thread 1 can take the mutex and enter the critical section

concurrent execution



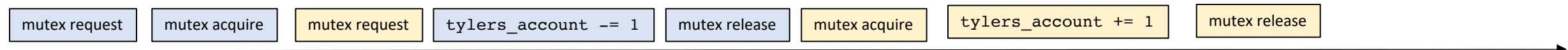
time



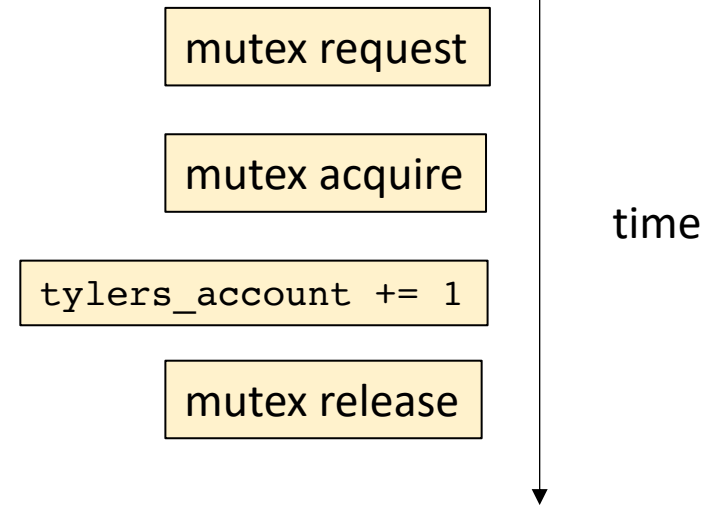
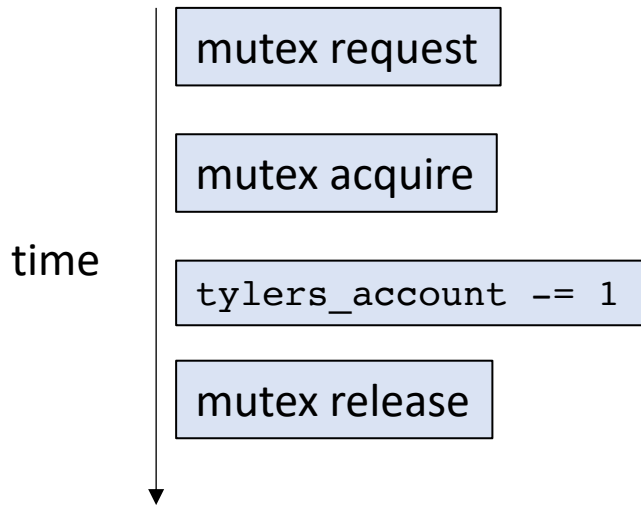
***A mutex restricts the number of allowed interleavings
 Critical section are mutually exclusive: i.e. they cannot interleave***

*Thread 1 can take the mutex
 and enter the critical section*

concurrent execution



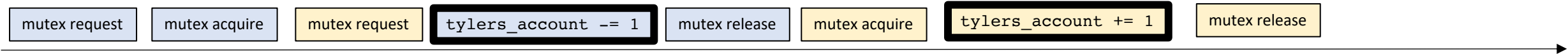
time



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

Thread 1 can take the mutex and enter the critical section

concurrent execution

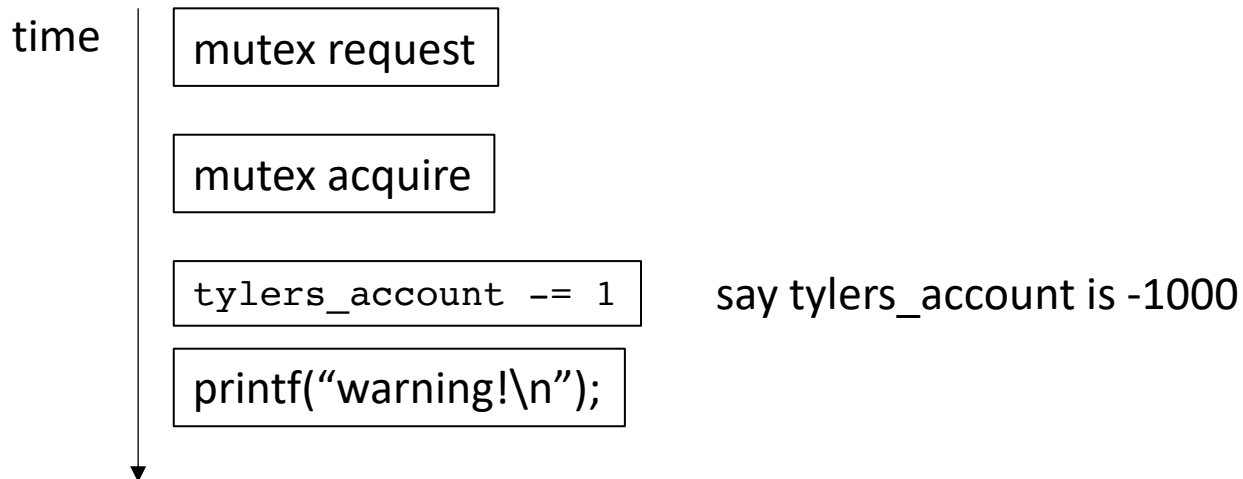


time

Make sure to unlock your mutex!

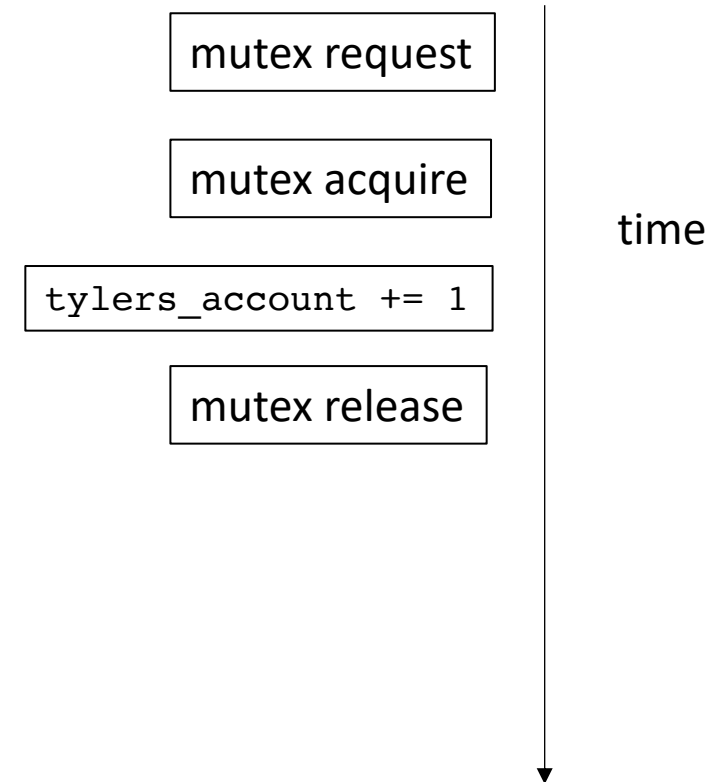
Tyler's coffee addiction:

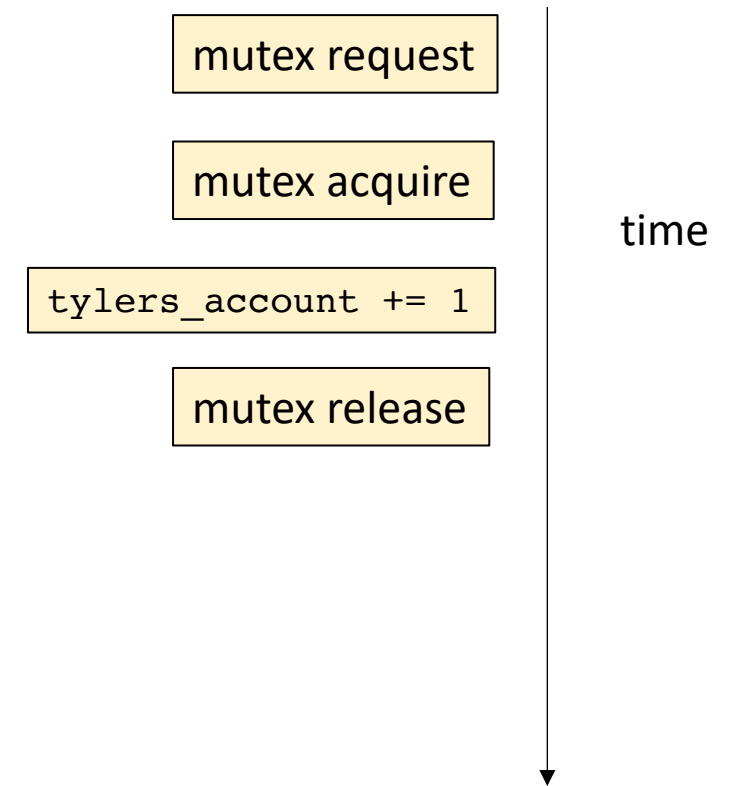
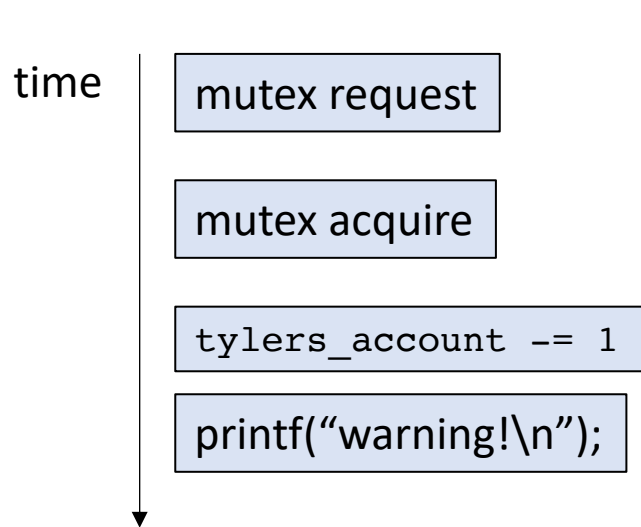
```
m.lock();
tylers_account -= 1;
if (tylers_account < -100) {
    printf("warning!\n");
    return;
}
m.unlock();
return;
```



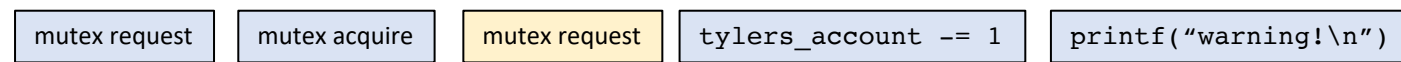
Tyler's employer

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





concurrent execution



Thread 1 is stuck!

Thanks!

- Next time:
 - Formal properties of mutual exclusion
 - Using multiple mutexes
 - Atomic RMWs
- Enjoy the long weekend!
- Get started on HW 1!
- Do the quiz!