OS and Parallelism

Andrew DeOrio

Agenda

- Motivation
- Processes
- Threads
- Synchronization
  - Atomic operations
- Summary

- Theme: How to run multiple things at the same time

Distributed systems

- Distributed system: Multiple computers cooperating on a task

- MapReduce: Distributed system for compute
  - Run a program that would be too slow on one computer
- Google File System: Distributed system for storage
  - Store more data than fits on one computer

Distributed system implementation

- How are distributed systems implemented?
- Threads and processes for parallelization
  - Today
- Networking for communication
  - Next time

Not parallel: sequential program

- The Flask development server is a sequential program
- Handles one request at a time

```
@app.route("/")
def index():
    context = # slow database read
    return flask.render_template("index.html", **context)
```

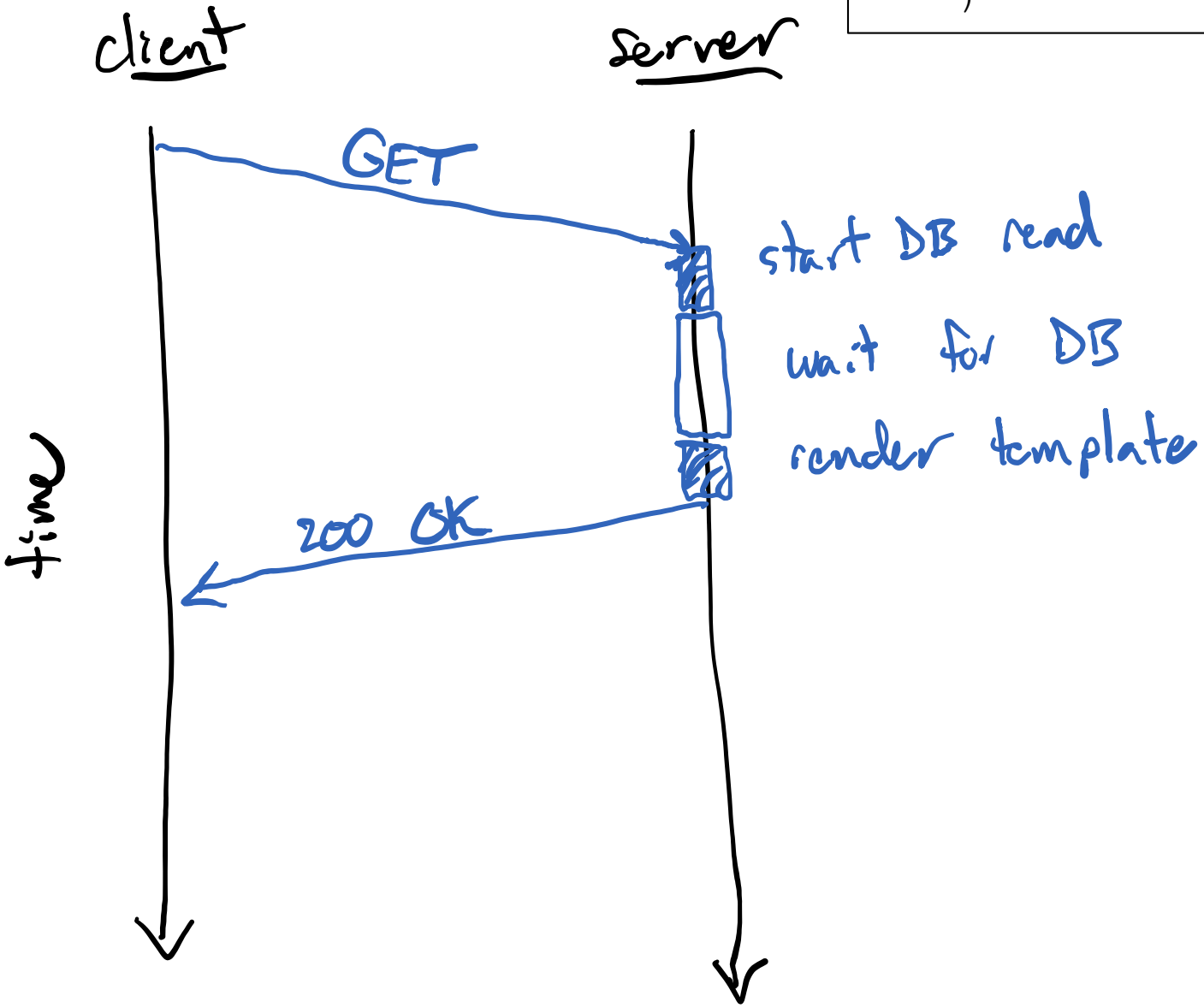
```
$ FLASK_APP=hello flask run
 * Running on http://127.0.0.1:5000/
```

- First, we'll look at a one request from one client

```
$ curl localhost:5000
<html><body>Hello World!</body></html>
```

One request

```
@app.route("/")
def index():
    context = # slow database read
    return flask.render_template(
        "index.html", **context
    )
```

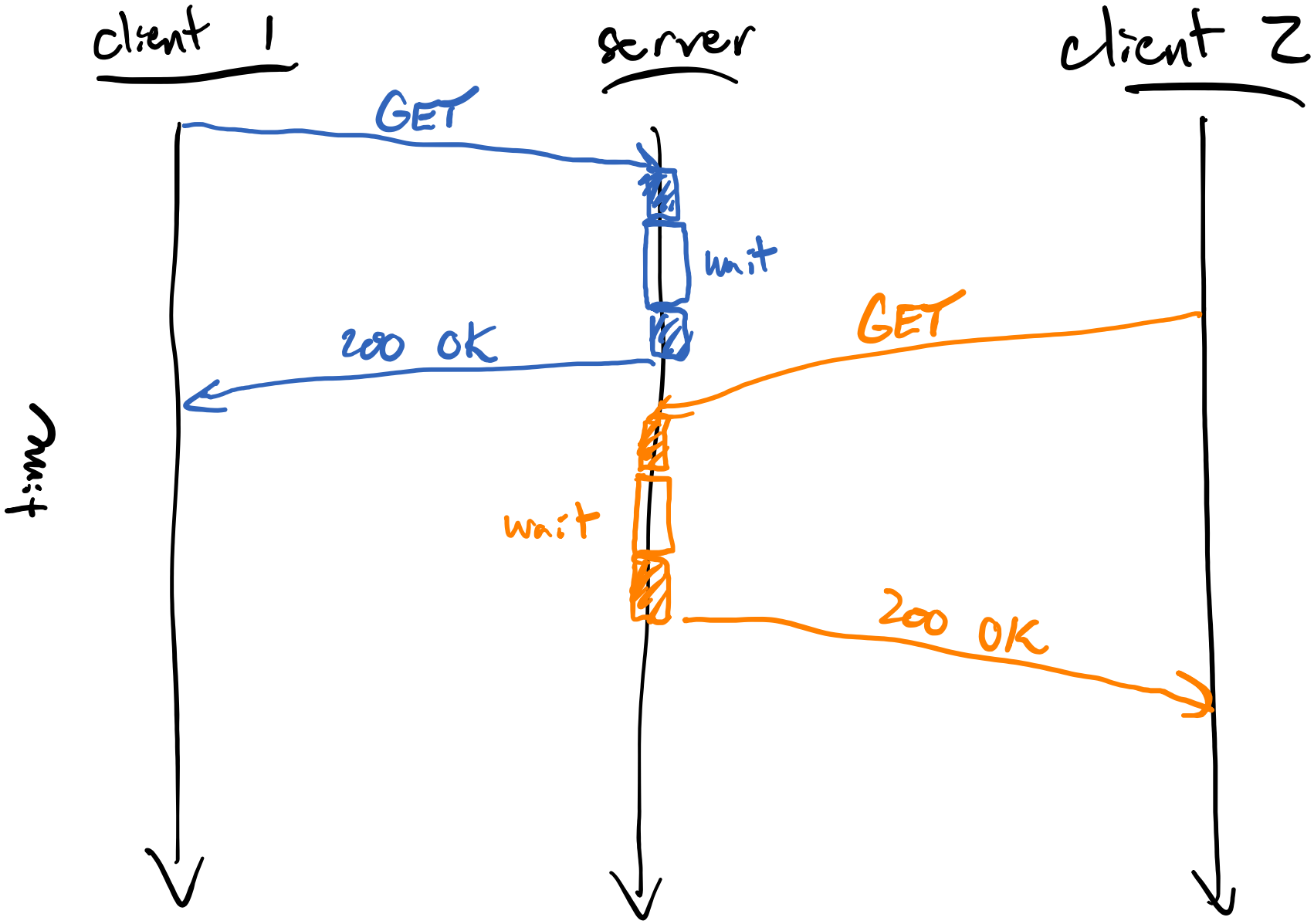


Serial requests

- Two serial requests from two clients

- Request from client 1
- Start client 1 database read
- Wait for database
- Render client 1 template
- Response to client 1
- Request from client 2
- Start client 2 database read
- Wait for database
- Render client 2 template
- Response to client 2

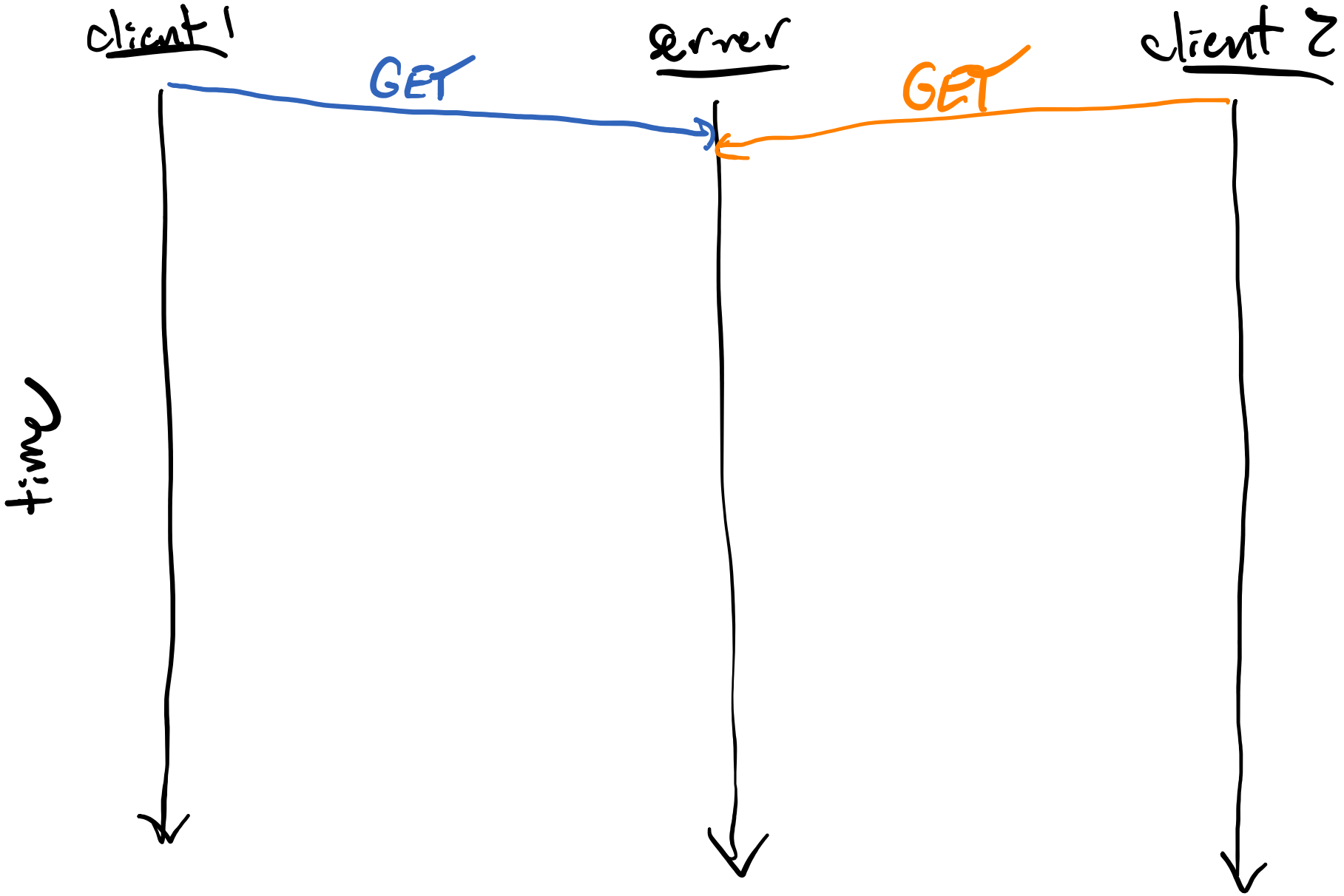
Serial requests



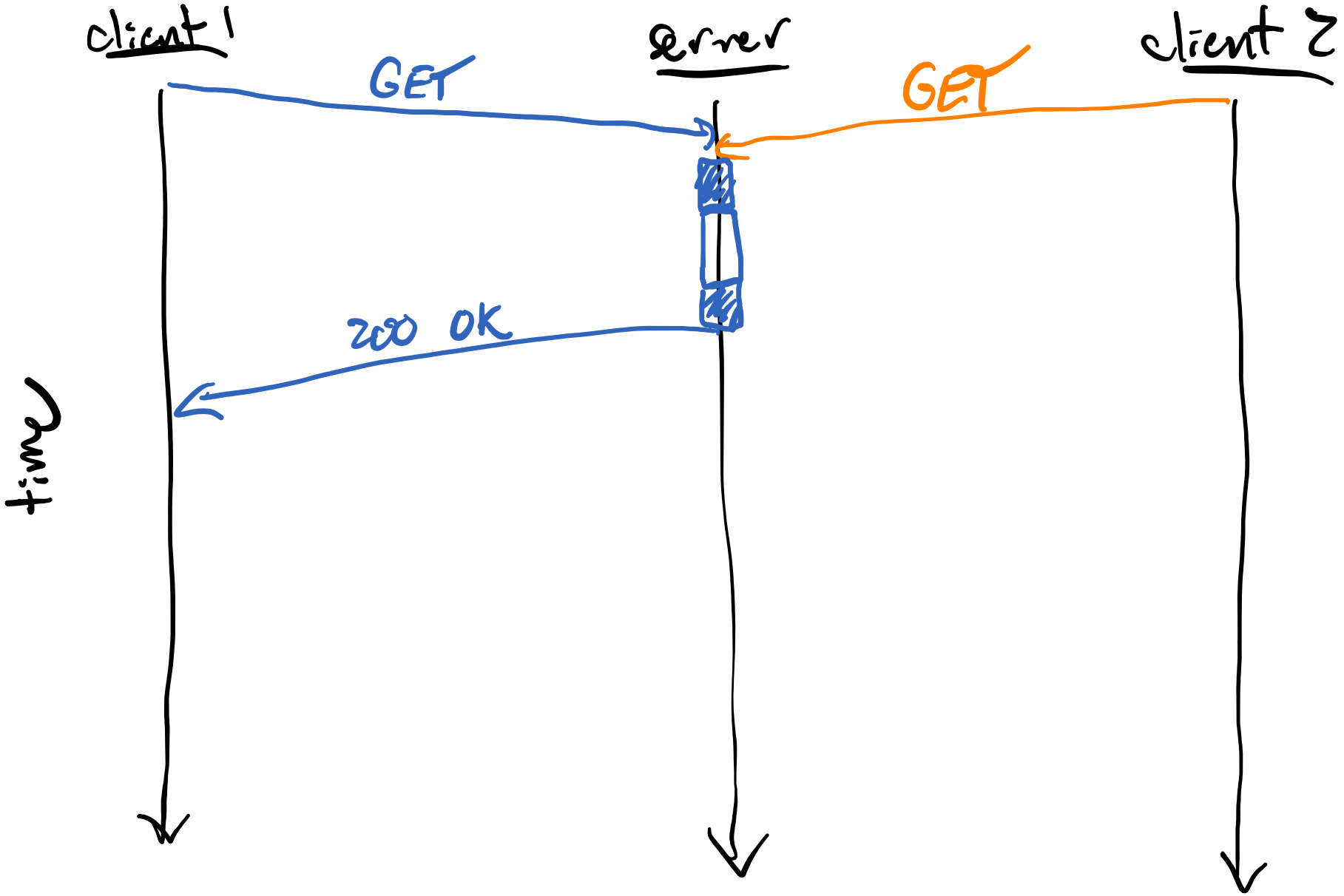
Concurrent requests

- Two concurrent requests from two clients
- **Request from client 1**
- **Request from client 2**
- Start client 1 database read
- Wait for database
- Render client 1 template
- Response to client 1
- Start client 2 database read
- Wait for database
- Render client 2 template
- Response to client 2

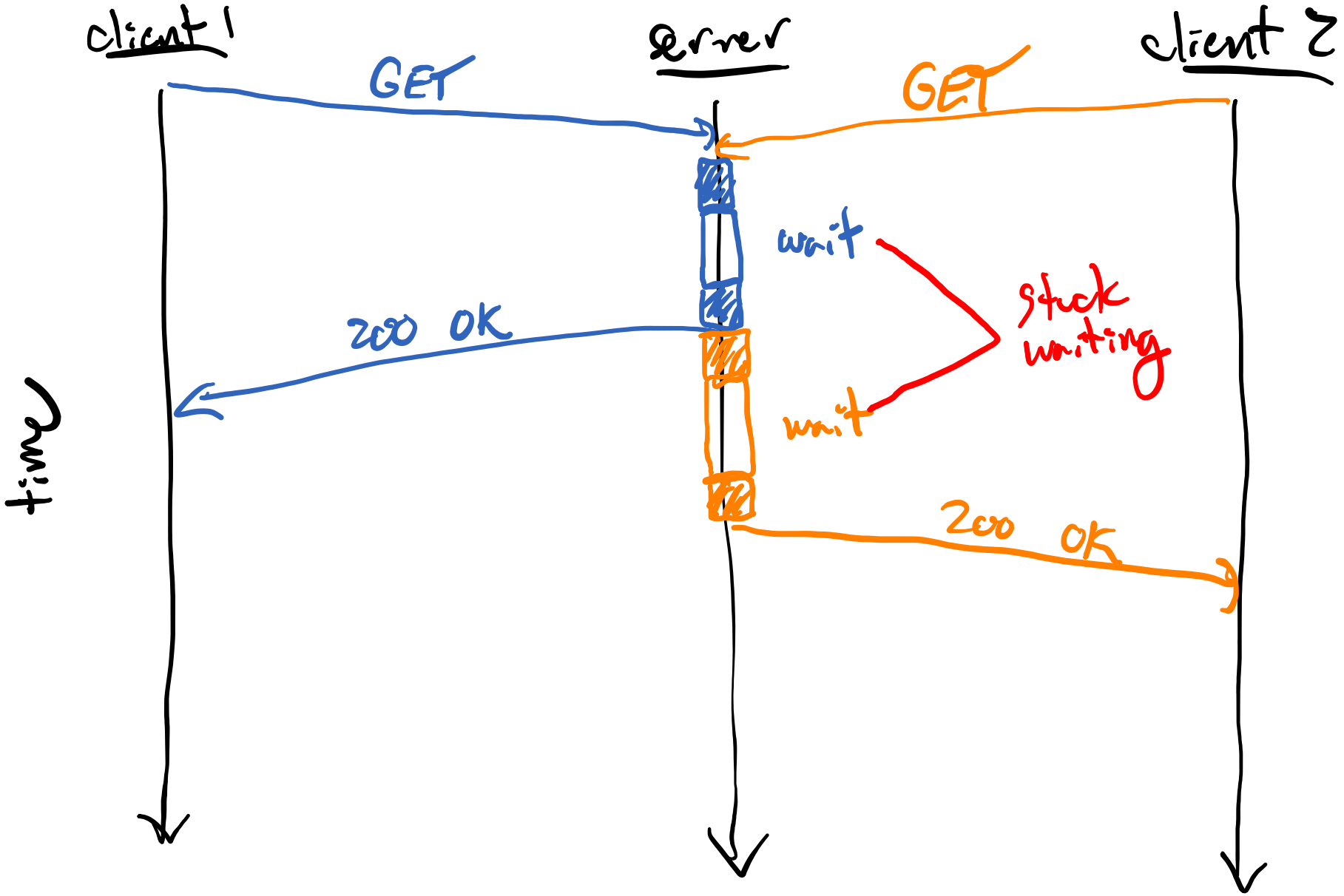
Concurrent requests



Concurrent requests



Concurrent requests



Problem with sequential programs

- Problem with sequential programs when:
- Multiple things happening at once
  - Two concurrent requests in this example
- Slow resource
  - Database read in this example

- Our program spends a lot of time waiting
- Solution: Work on both requests at the same time

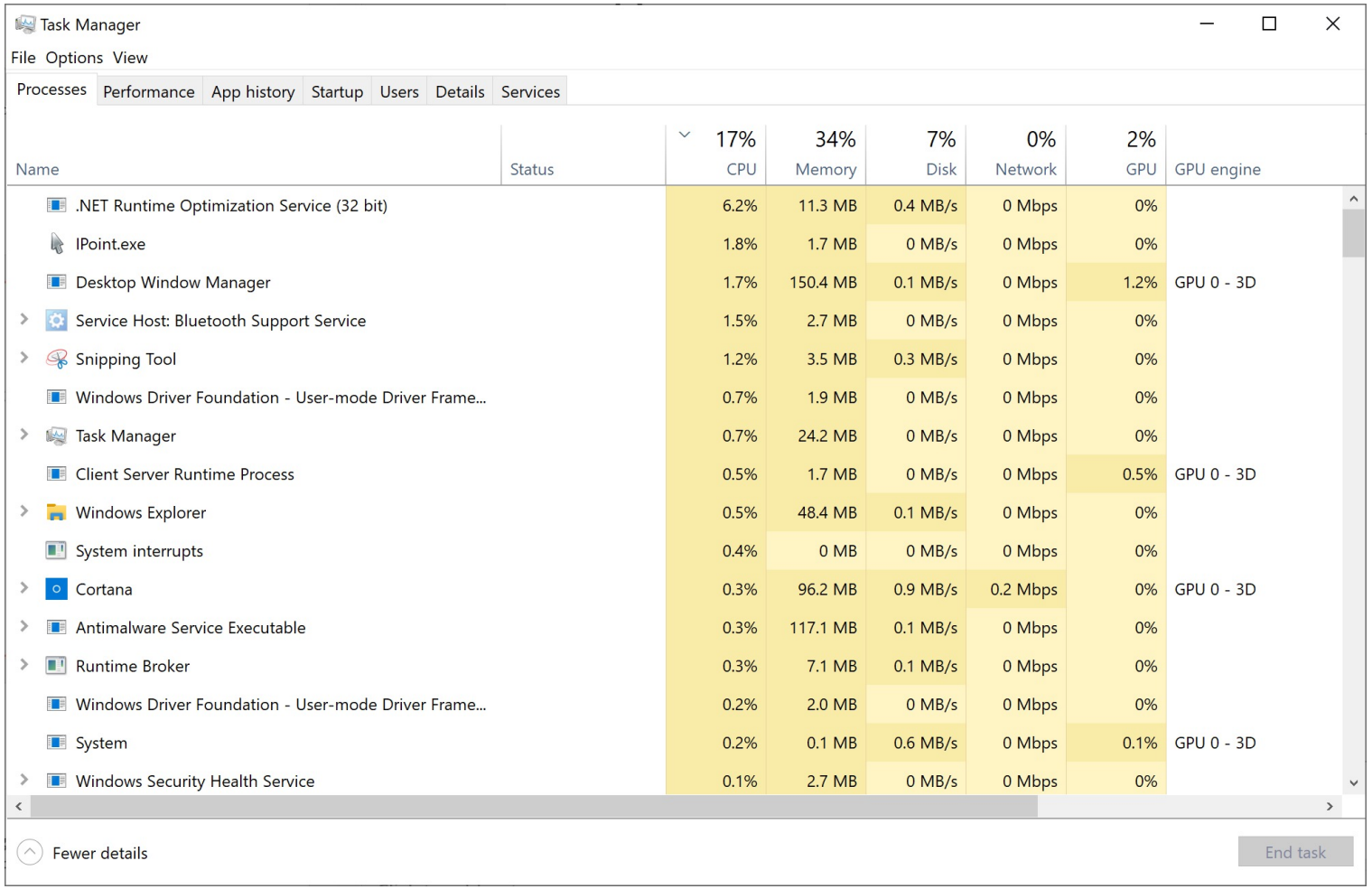
- Later today, we'll solve this problem with threads, but first we need to learn about processes

Agenda

- Motivation
- **Processes**
- Threads
- Synchronization
  - Atomic operations
- Summary

Processes

- A process is a running program



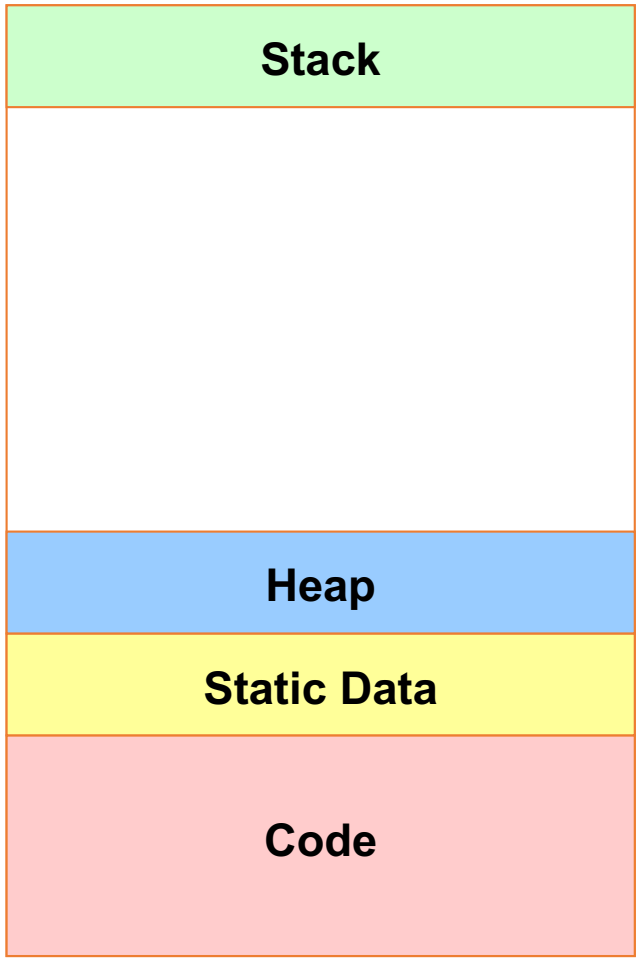
The process abstraction

- The *Process* is an OS abstraction for execution
- A *process* is a program in execution
  - Programs are static entities with potential for execution

- Process consists of:
  - A unique process ID (PID)
  - An address space (memory)
  - 1 or more threads (sequences of computation)
  - Some other resources (file handles, open sockets,...)

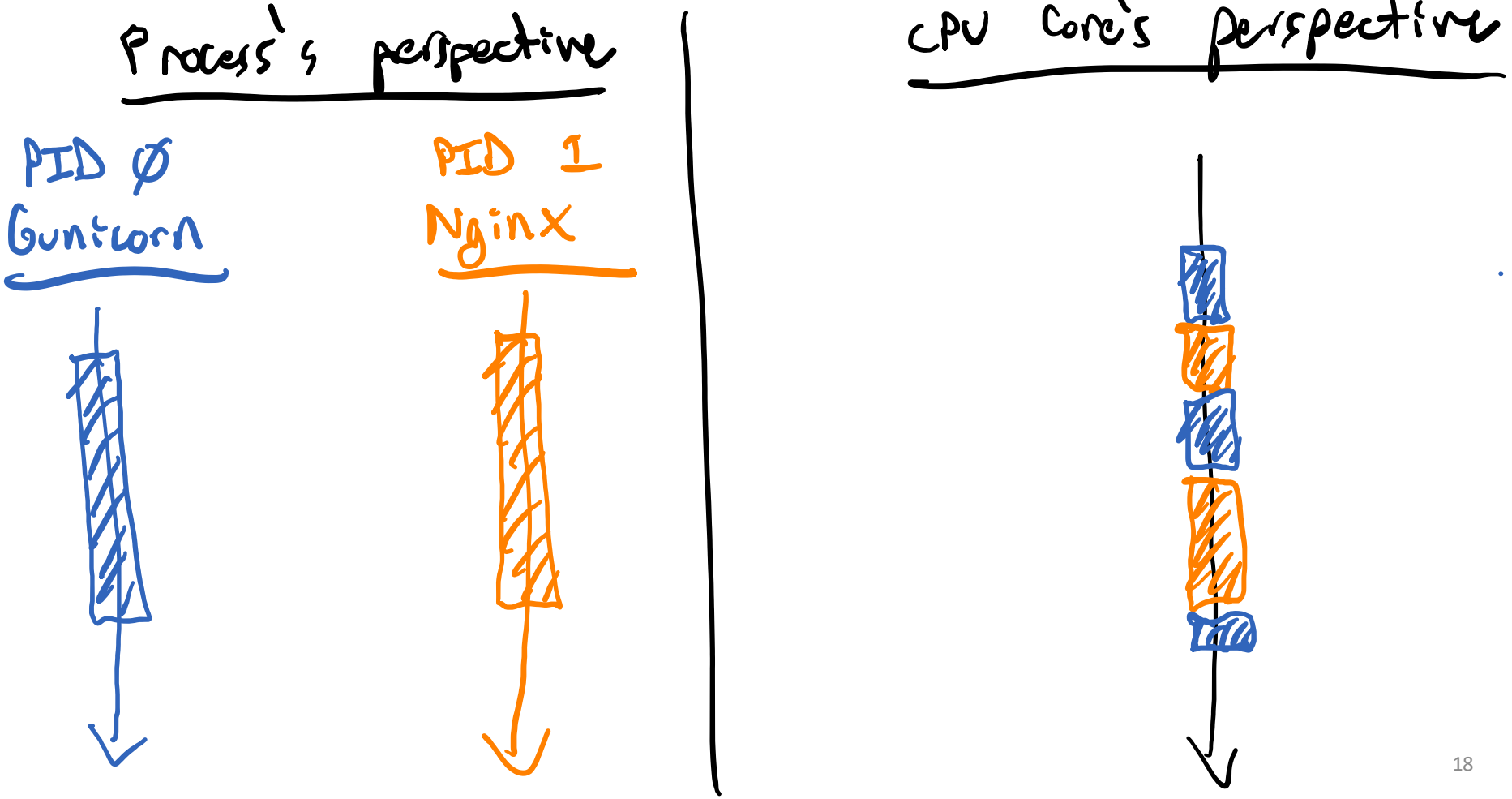
Process address space

- Each process gets its own address space
- Process can only write to its own address space



Operating system process scheduling

- Each CPU core can run one process at a time
- Every ~1-10ms, the OS can switch



Process abstraction

- On a single-core machine
  - Processes take turns
- On a multi-core machine
  - Processes run in parallel
  - Processes still take turns because there are usually many more processes than cores

- Recall that the *Process* is an OS abstraction for execution
- A process is an illusion of parallelism provided by OS

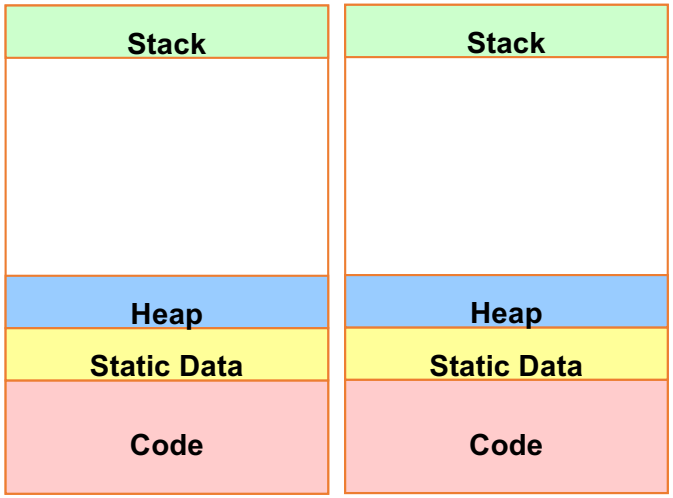
When are processes useful?

- Processes are most useful when:

- Multiple things happening at once
- Different programs running on the same machine
- Same program running on different machines

When are processes not useful?

- Back to our problem with sequential programs
  - Two concurrent requests with slow database operations
- Solution: Two processes
- Two processes can handle two requests concurrently
- Problem: Higher memory overhead
  - Copy of entire address space
  - They don't really need separate code, etc.

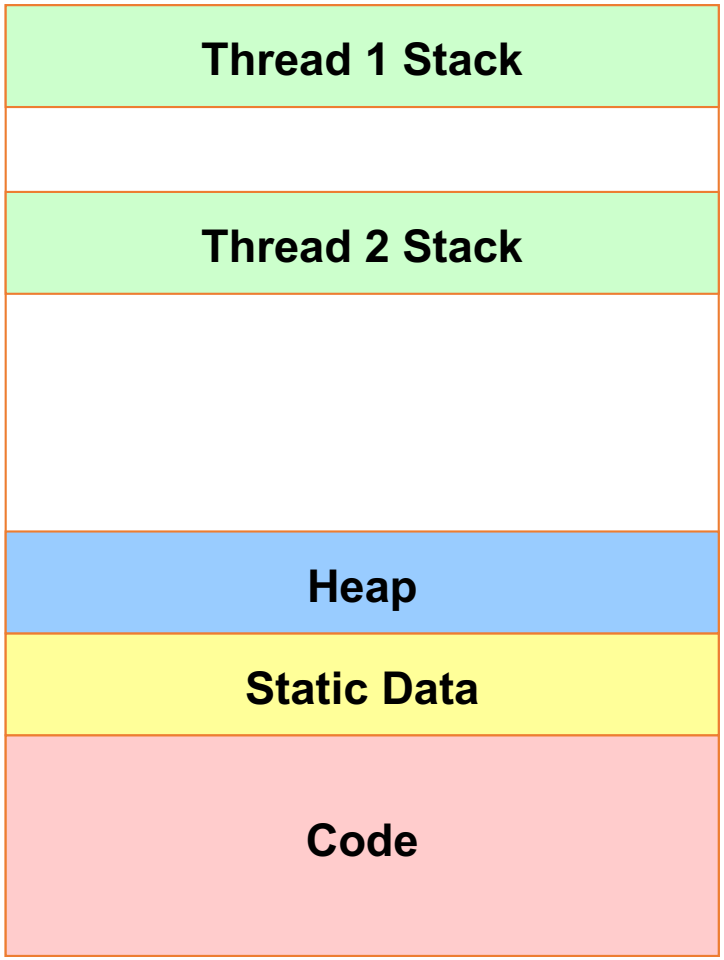


Agenda

- Motivation
- Processes
- **Threads**
- Synchronization
  - Atomic operations
- Summary

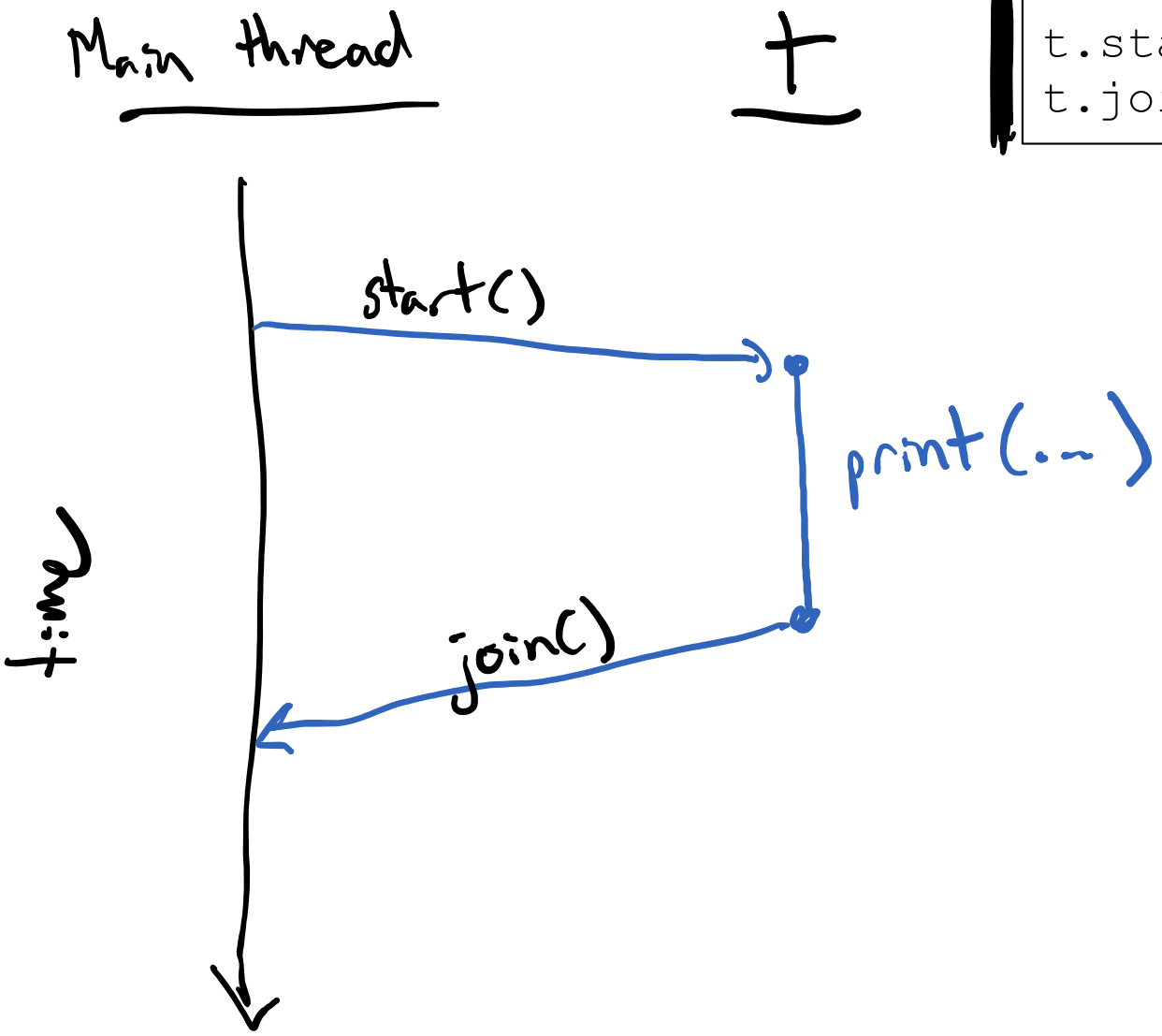
Threads

- *Threads* are multiple functions in one process running simultaneously
- Share heap, static data, code
- Lower overhead than processes



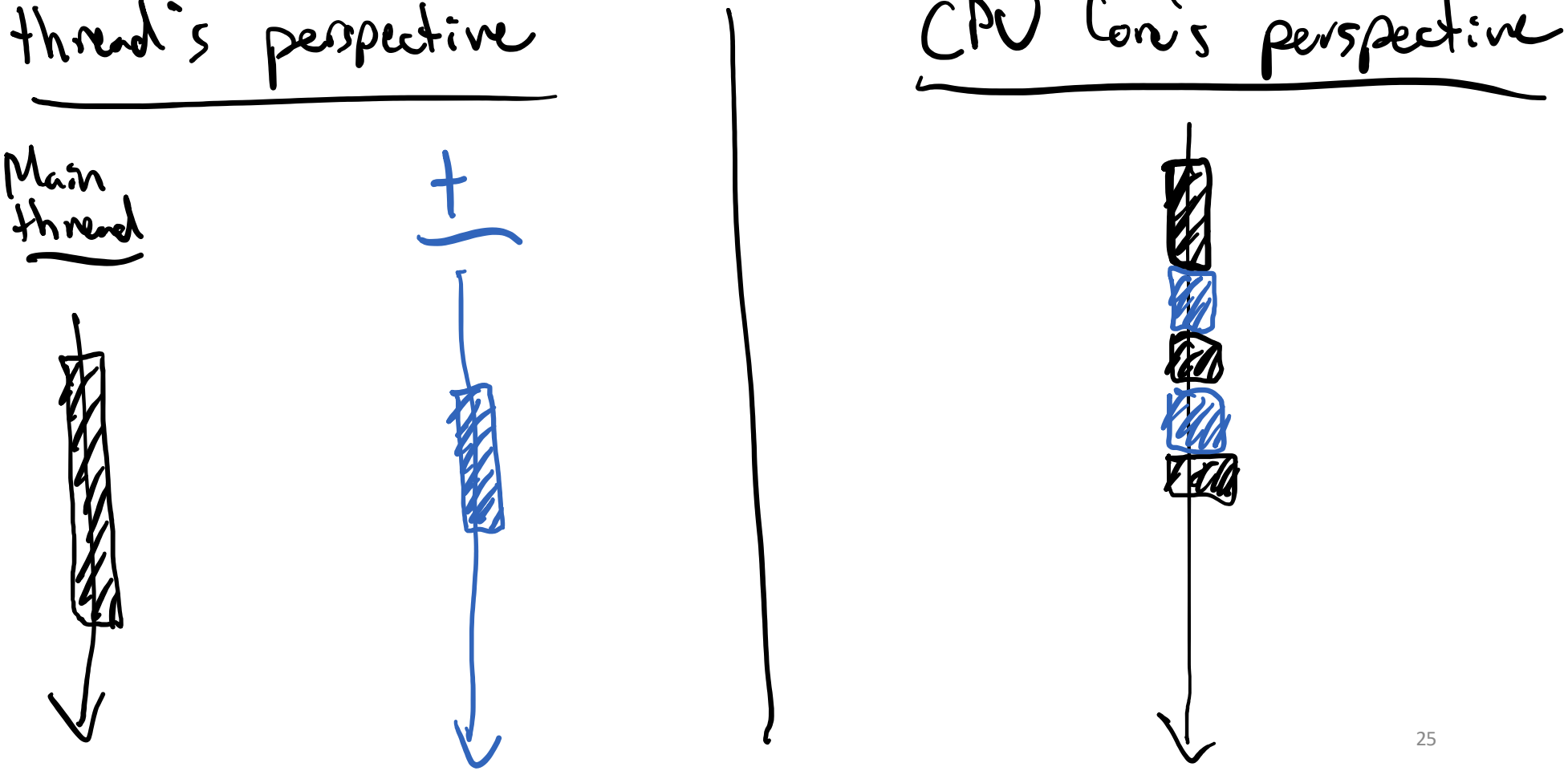
Thread example

```
from threading import Thread

def hello():
    print("hello world")

t = Thread(target=hello)
t.start()
t.join()
```



Operating system thread scheduling

- Each CPU core can run one thread at a time
- Every ~1-10ms, the OS can switch



Thread abstraction

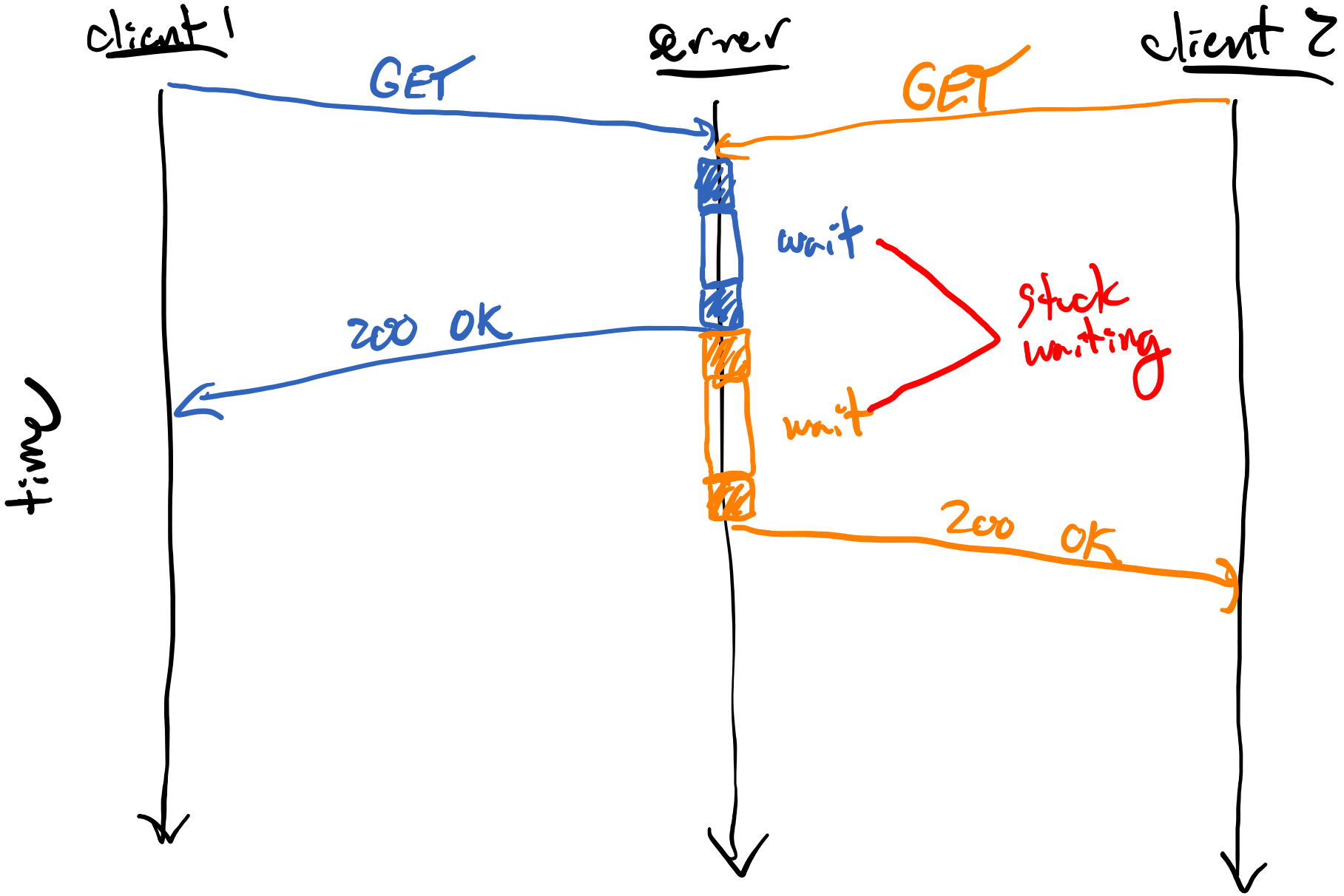
- On a single-core machine
  - Threads take turns
- On a multi-core machine
  - Threads run in parallel*
  - Threads still take turns because there are usually more threads than cores

- Threads have a lot in common with Processes
  - Both provide an illusion of parallelism
  - Both can provide true parallelism on the right hardware

- Threads share memory
  - Less overhead
  - Possible security risk, depending on the application

*In Python, threads cannot run in parallel due to the Global Interpreter Lock

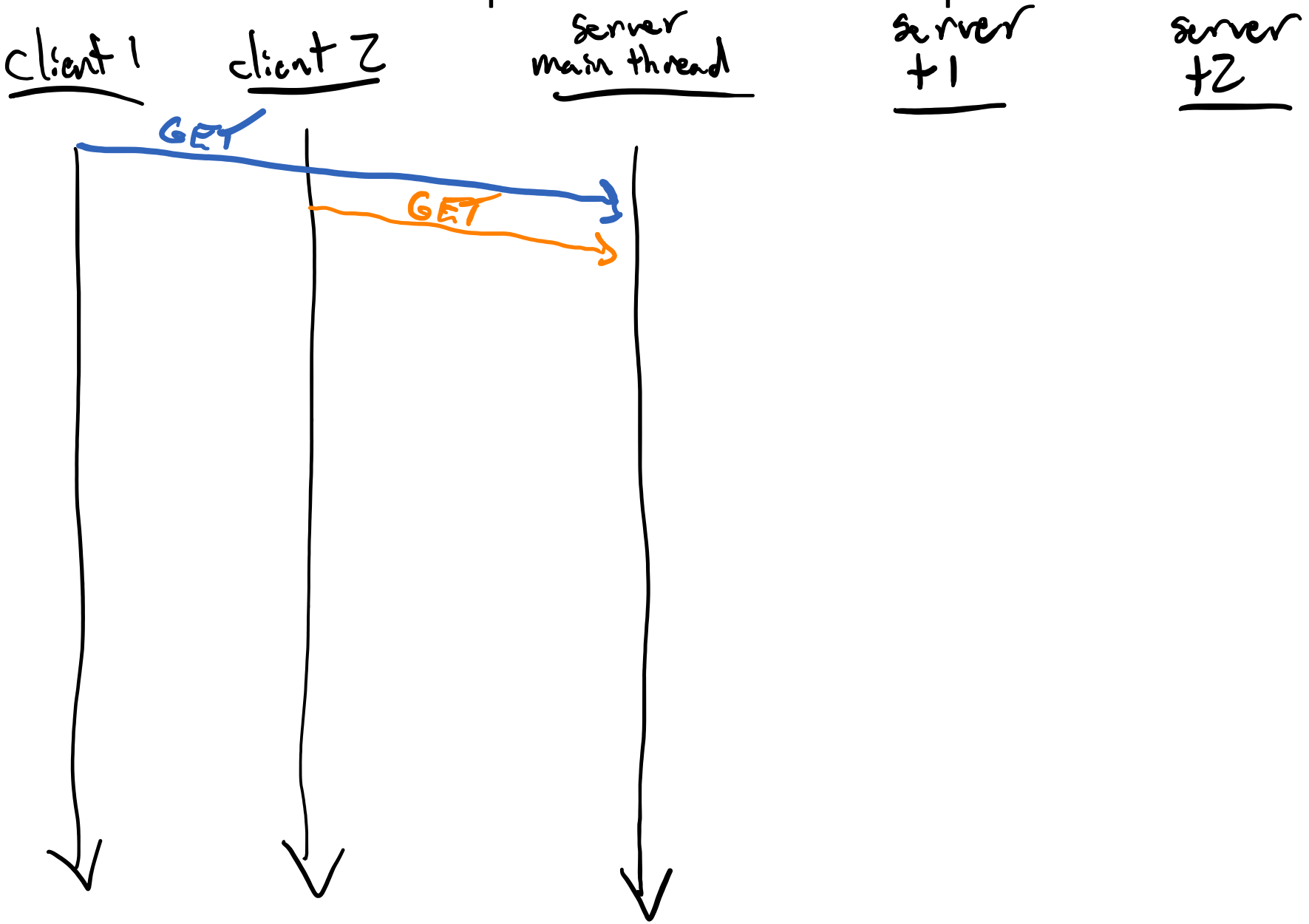
Recall: concurrent requests with 1 thread



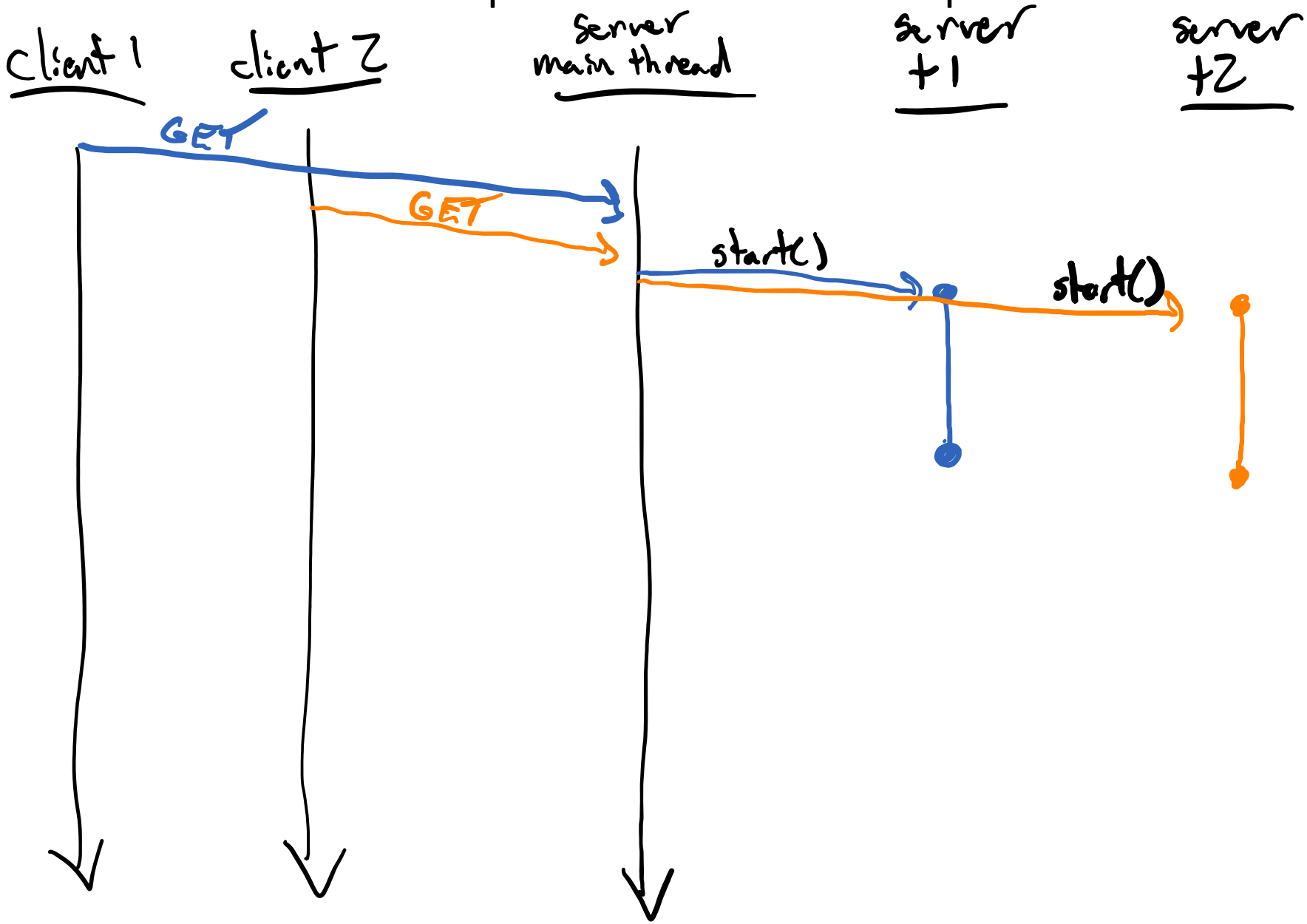
Concurrent requests with multiple threads

- Two concurrent requests from two clients
  - Note: other interleavings are possible
- Request from client 1, assign to thread 1
- Request from client 2, assign to thread 2
- Start client 1 database read
- Start client 2 database read
- **Wait for client 1 database read**
- **Wait for client 2 database read**
- Render client 1 template
- Response to client 1
- Render client 2 template
- Response to client 2

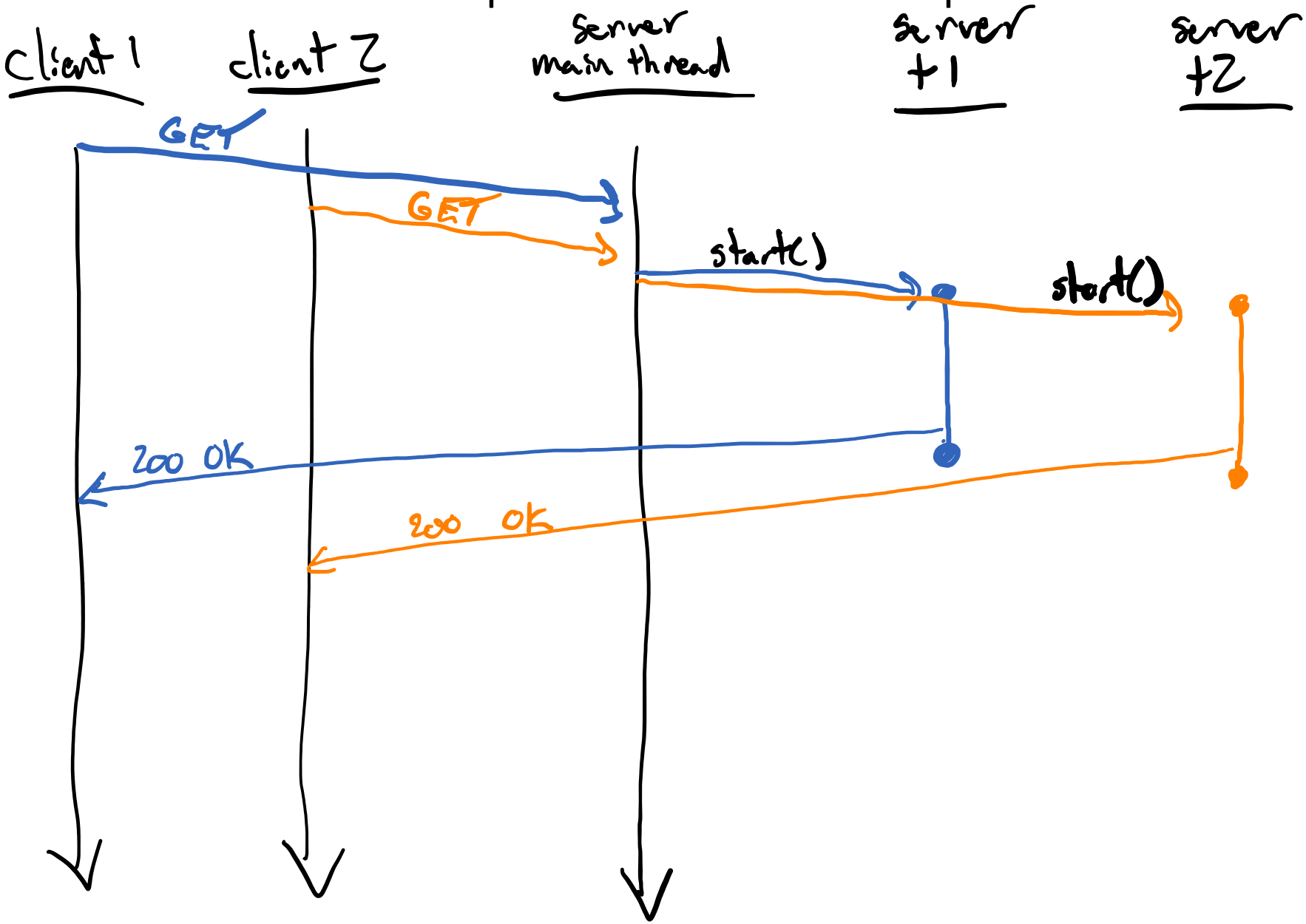
Concurrent requests with multiple threads



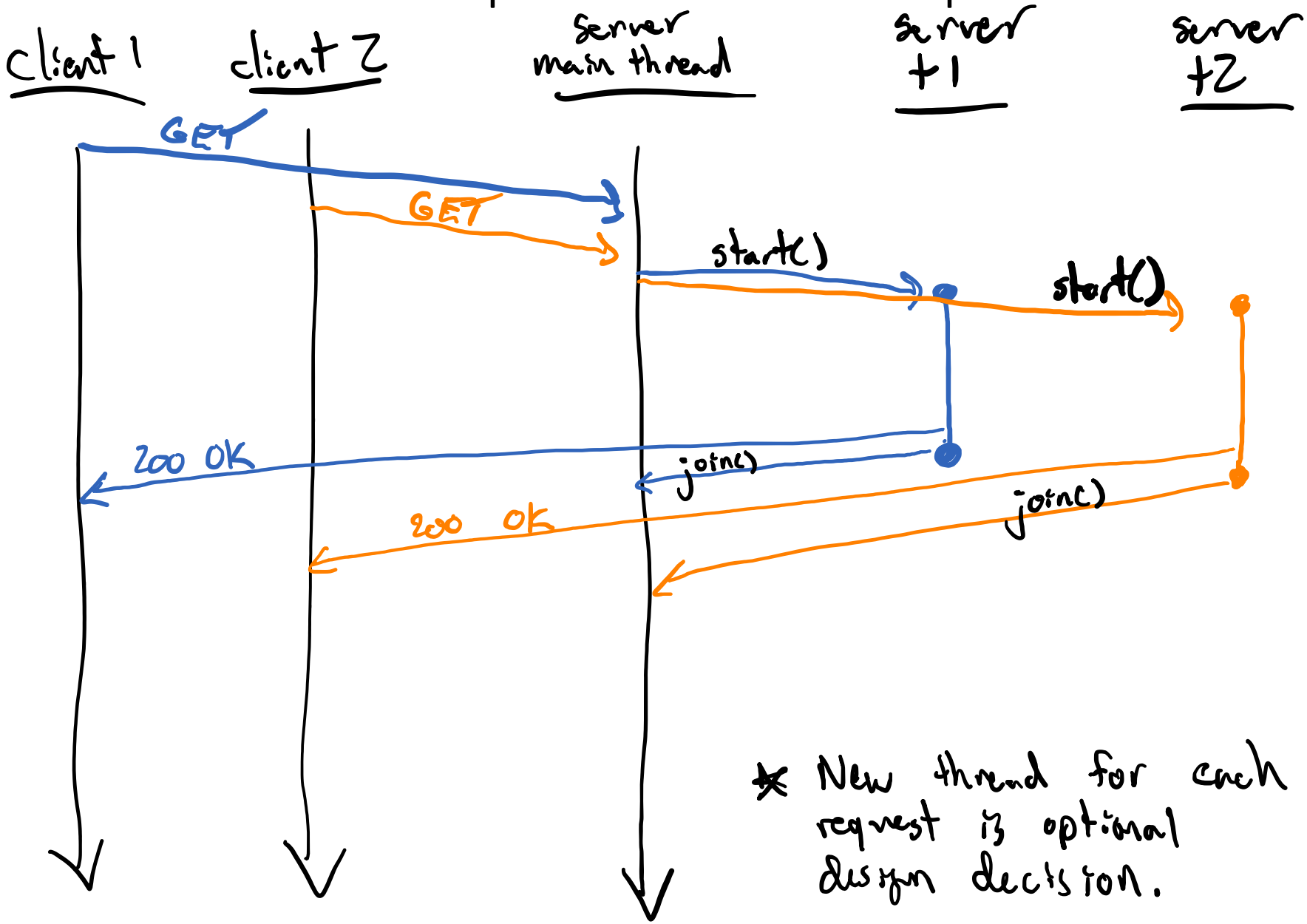
Concurrent requests with multiple threads



Concurrent requests with multiple threads



Concurrent requests with multiple threads



When are threads useful?

- Multiple things happening at once
  - Within one program
- May need to share data in memory
- Usually some slow resource
  - Network response
  - Disk I/O
  - Reading a sensor
  - Human input

Benefits of threads

- Simpler programming model
  - The illusion of a dedicated CPU per thread
  - State for each thread (local variables)

- OS takes care of CPU sharing
  - Other threads can progress while one thread waits for I/O

When to use processes vs. threads

- **Processes** have separate address spaces
  - Useful when there is not complete trust
  - Useful to limit the damage caused by buggy code
  - Needed for execution on different computers

- **Threads** share an address space
  - Useful for lower memory overhead
  - Can share data

Agenda

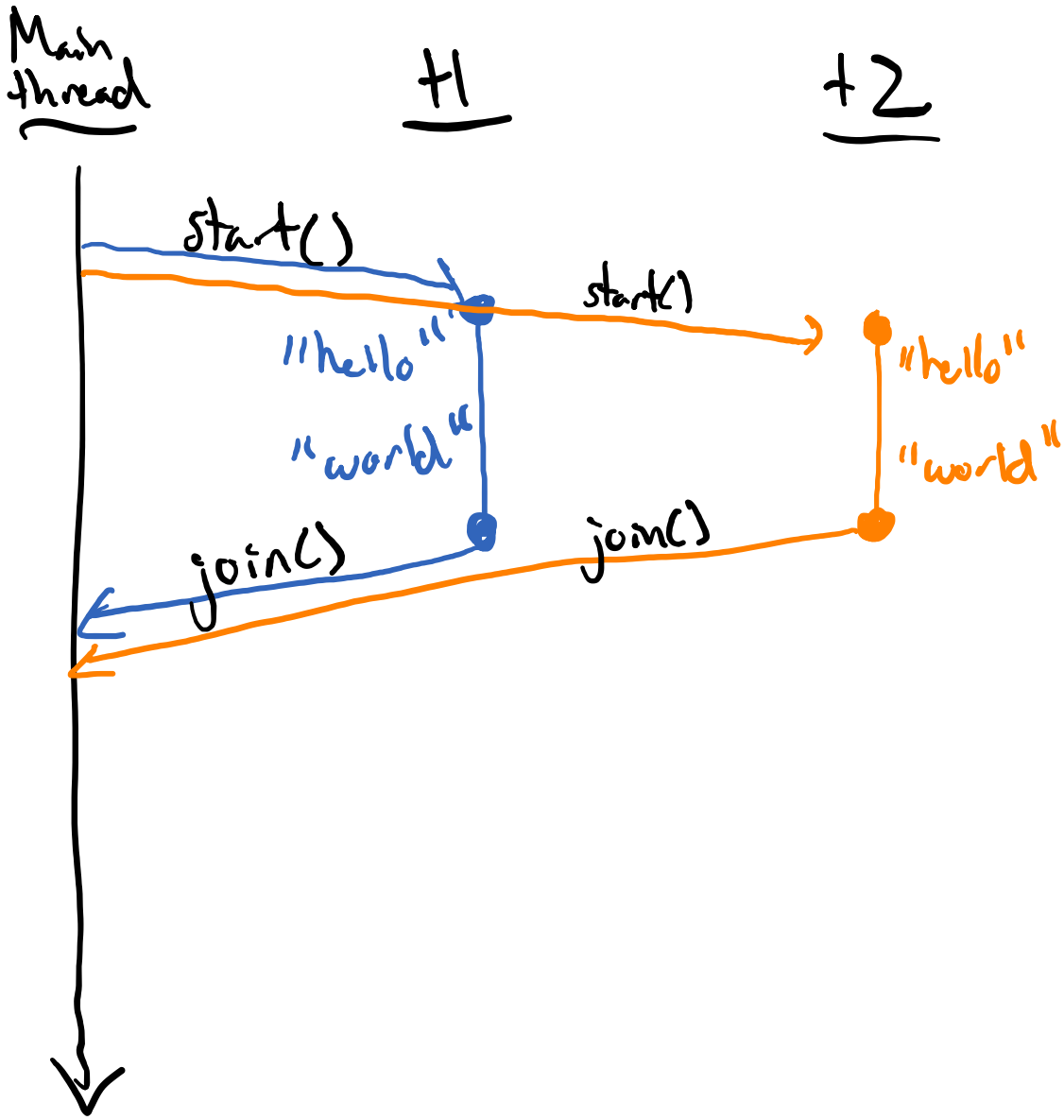
- Motivation
- Processes
- Threads
- **Synchronization**
  - Atomic operations
- Summary

Cooperating threads

- What happens when threads cooperate?
- What are these threads sharing? The output stream.
- What is the output of this code?

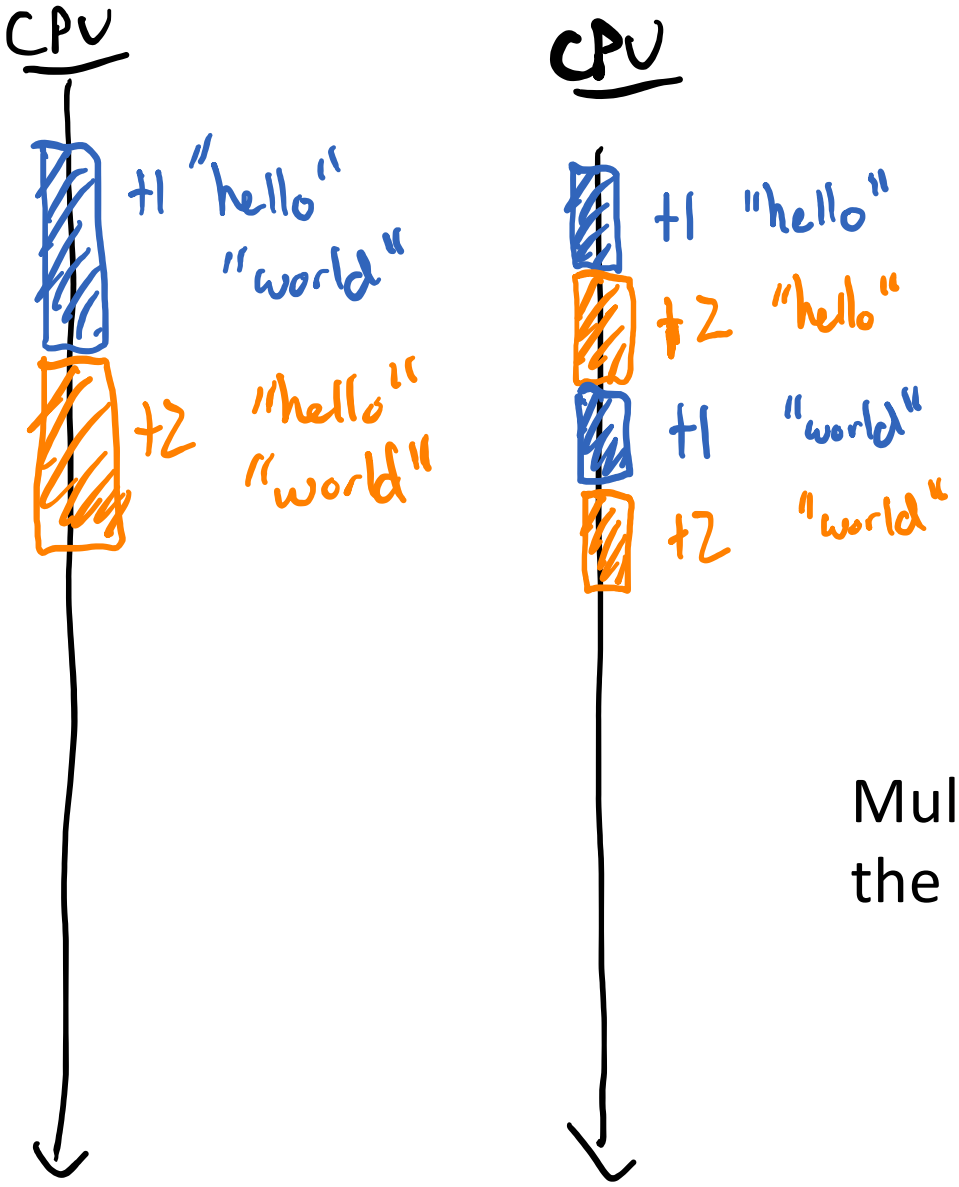
```
def hello():
    print("hello")
    print("world")

t1 = Thread(target=hello)
t2 = Thread(target=hello)
t1.start()
t2.start()

t1.join()
t2.join()
```

Cooperating threads




```
def hello():
    print("hello")
    print("world")

t1 = Thread(target=hello)
t2 = Thread(target=hello)
t1.start()
t2.start()

t1.join()
t2.join()
```

Cooperating threads




```
def hello():
    print("hello")
    print("world")

t1 = Thread(target=hello)
t2 = Thread(target=hello)
t1.start()
t2.start()

t1.join()
t2.join()
```

Multiple ways to interleave the operations from t1 and t2

Many possible outputs

```
def hello():
    print("hello")
    print("world")

t1 = Thread(target=hello)
t2 = Thread(target=hello)
t1.start()
t2.start()
```

- Possible outputs

```
hello
world
hello
world
```

```
hello
world
hello
world
```

- Impossible outputs

```
world
hello
world
hello
```

```
world
world
hello
hello
```

Unpredictable order

- When threads share data, the output can be unpredictable
- OS decides when threads take turns
- Different timing -> different thread order - > different output

Side note

- To replicate this example, use this code
- Intentionally slow `print()` causes CPU to switch threads more often

- We're also relying on Python print function's line buffering to keep the characters "hello" and "world" together
  - More in this SO post

```
import time
import random
from threading import Thread

def slow_print(text):
    if random.randint(0, 1):
        time.sleep(0.01)
    print(text)

def hello():
    slow_print("hello")
    slow_print("world")

t1 = Thread(target=hello)
t2 = Thread(target=hello)
t1.start()
t2.start()

t1.join()
t2.join()
```

Shared data

```
def hello():
    print("hello")
    print("world")

t1 = Thread(target=hello)
t2 = Thread(target=hello)
t1.start()
t2.start()

t1.join()
t2.join()
```

- Insight: Shared data
- The output buffer in this example

- Ordering within a thread is sequential
- Many ways to merge multiple threads together into global order

- Problem: Some global orderings may produce incorrect results
- Solution: Locks

Agenda

- Motivation
- Processes
- Threads
- Synchronization
  - **Atomic operations**
- Summary

Atomic operations

- An *atomic* operation appears to the other threads as if it happened instantaneously
  - No code from other threads can occur in between

- In our previous example, `print()` behaved atomically*

*The `print()` output was line buffered, which means different lines of output cannot be interleaved (more detail).

Atomic operations in Python

- The following operations are atomic in Python:
  - Update, insert, read, etc. of dictionaries, lists
  - Reads/writes of built-in types
  - Read and assignment of instance variables

- Safe to perform these operations simultaneously from different threads

- Combinations of atomic operations are not atomic
  - Safe to execute x = 1 and x = 0 in different threads
  - Not safe to execute x = x + 1 in different threads!

Locks

- How can we make sure "hello world" is printed by each thread?
- Limit when threads can take turns

- A *Lock* is for mutual exclusion
  - Also called a *mutex*

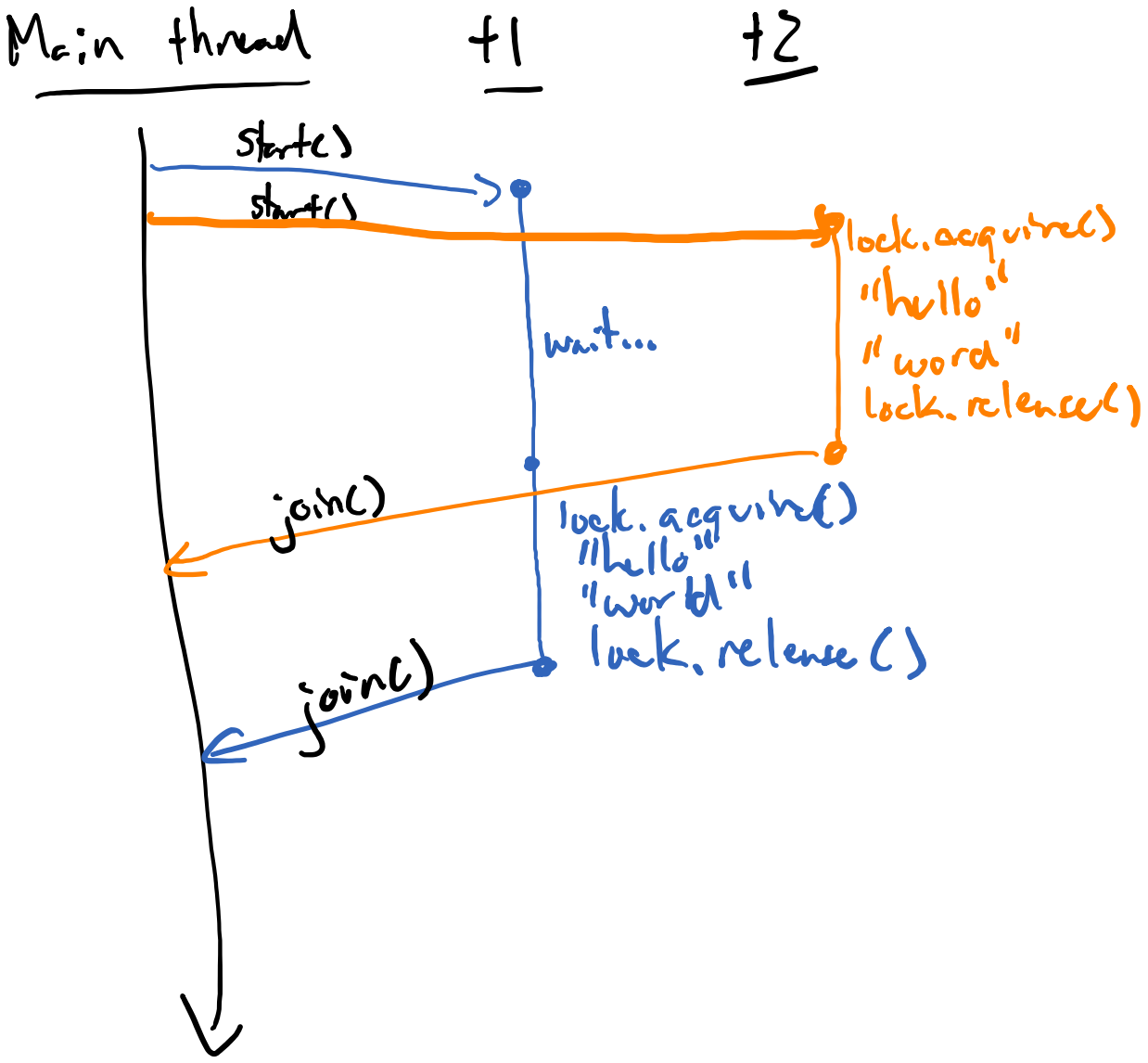
- A *Lock* lets the programmer limit when threads take turns

Locks

- How can we make sure "hello world" is printed by each thread?
- Limit when threads can take turns

```
import time
import random
from threading import current_thread,
Thread, Lock

def hello(lock):
    name = current_thread().name
    lock.acquire()
    print(f"{name} hello")
    print(f"{name} world")
    lock.release()

lock = Lock()
t1 = Thread(target=hello, args=(lock,))
t2 = Thread(target=hello, args=(lock,))
t1.start()
t2.start()

t1.join()
t2.join()
```

Locks


Main thread
t1
t2
start()
start()
lock.acquire()
"hello"
"word"
lock.release()
wait...
join()
lock.acquire()
"hello"
"world"
lock.release()
join()

Agenda

- Motivation
- Processes
- Threads
- Synchronization
  - Atomic operations
- **Summary**

Summary

- Parallelization is important for implementing distributed systems
- **Processes** and **threads** are used to write parallel programs
- With parallel programs, need to think about **synchronization** of data
- To synchronize data, we use **atomic operations**

Threads vs. processes

- **Threads** have lower overhead than processes
  - Threads share the same memory image (code and data) and other elements of the process state (open file handles, current directory, etc.) This reduces their resource footprint and the number of cycles needed for context switches between threads in the same process.

- **Processes** provide separate address spaces
  - Needed to run a different executable
  - Needed when threads are on different computers
  - Useful to create firewalling for security reasons
  - Useful to limit the damage caused by buggy code

Threads vs. asynchronous programming

- *Threads* are multiple functions in one process running simultaneously
  - OS controls when functions take turns
- *Asynchronous programming* is functions interleaved with one another in a single thread of control
  - Programmer controls when functions take turns

- Both implement concurrency
- Both are illusions of parallelism
  - Threads can provide true parallelism with the right hardware and software

Threads and processes in EECS 485

- You'll write a distributed system in Project 4
  - MapReduce framework
- Processes for Main server and each Worker
- Threads for simultaneous communication and computation

- Won't need locks in EECS 485
  - Take advantage of Python's existing atomic operations

Next time

- How are distributed systems implemented?
- Threads and processes for parallelization
  - Today
- Networking for communication
  - Next time