Threads vs Coroutines

Understanding the Concurrency Models of C++

CONOR SPILSBURY

Threads vs. Coroutines

Understanding the Concurrency Models of C++

Bloomberg

Engineering

CppCon 2025
September 18th, 2025

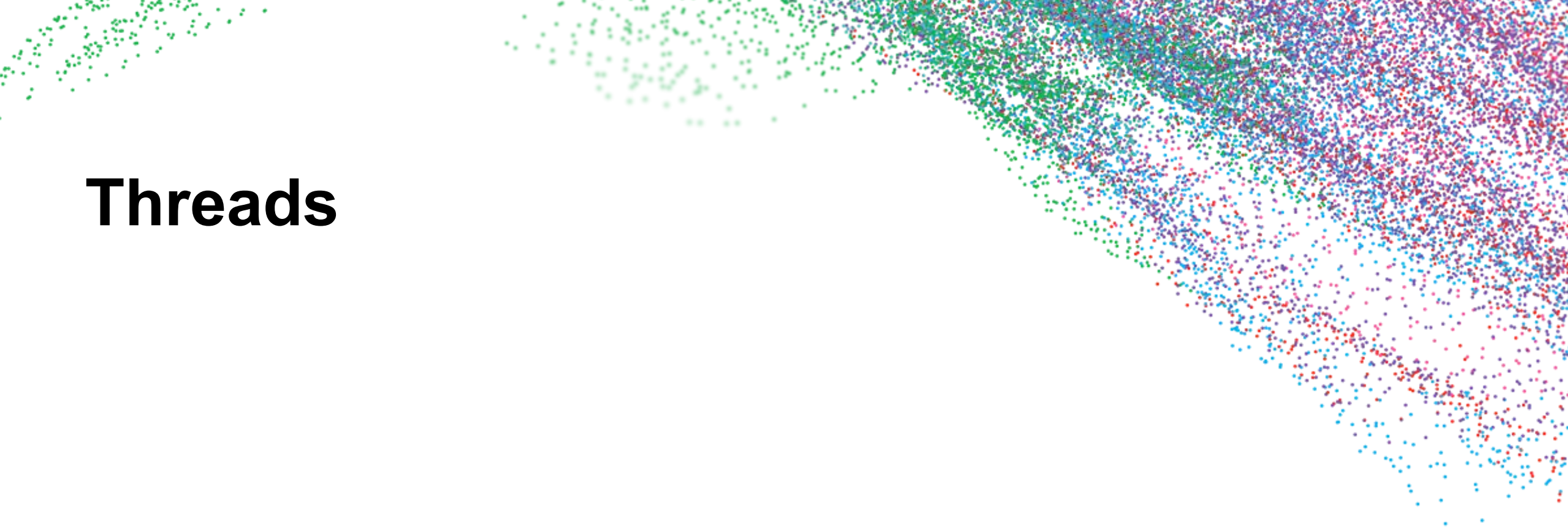
Conor Spilsbury
Senior Software Engineer

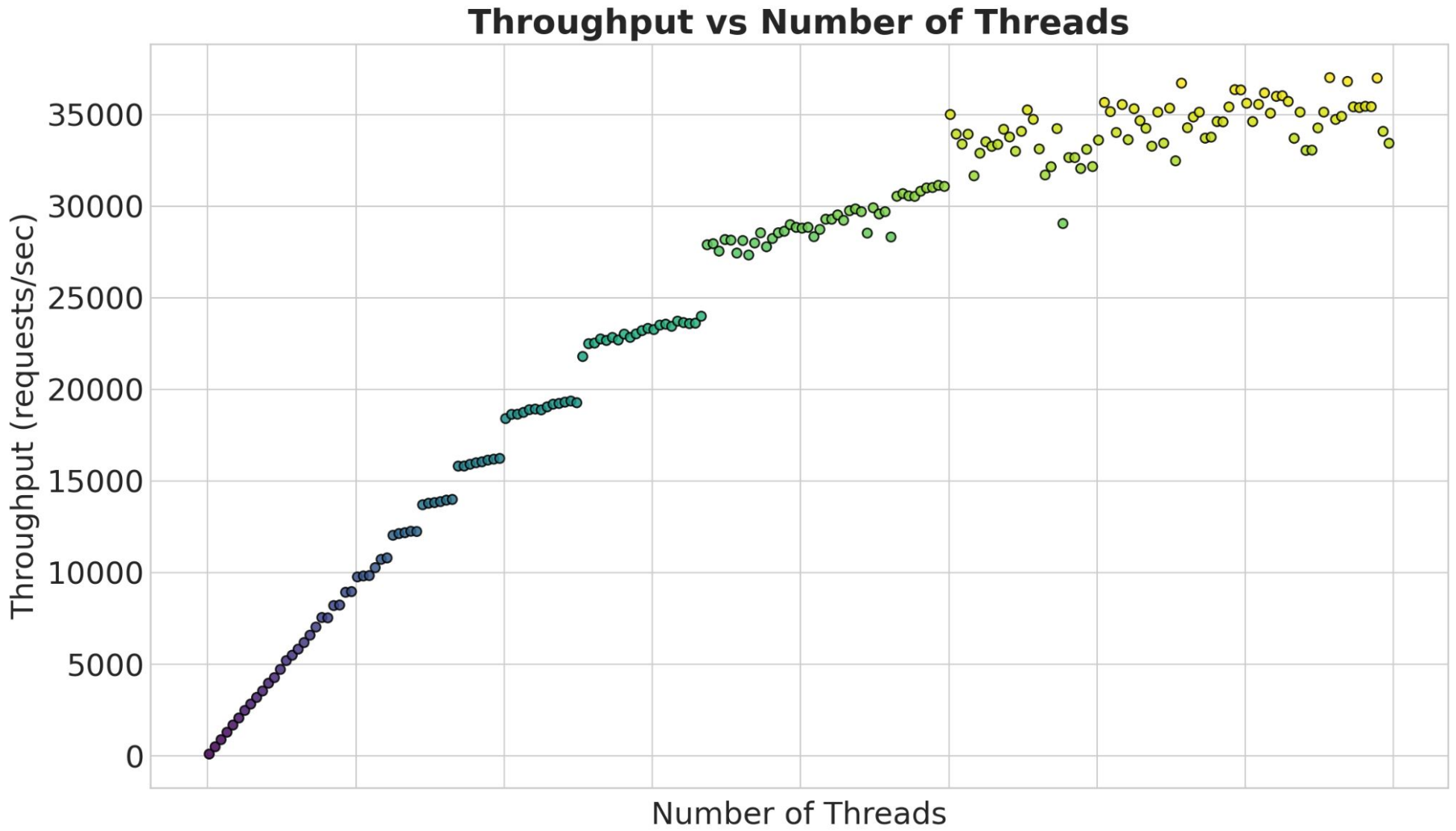
TechAtBloomberg.com

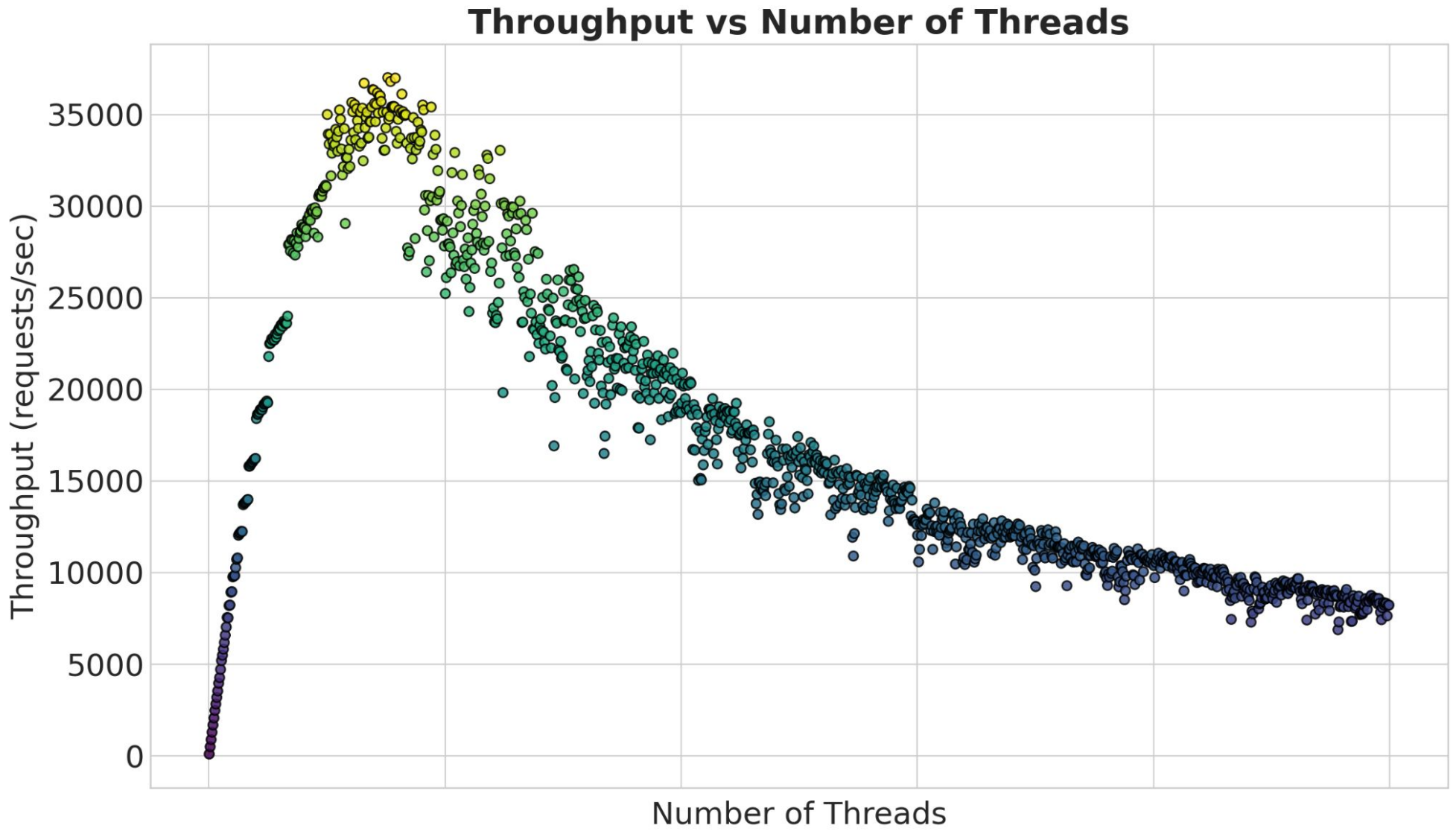
Design Decisions

- Application requirements shape concurrency decisions
- Threads vs. coroutines: limits of OS scheduling vs. cooperative scheduling
- Understand how to choose the right tool

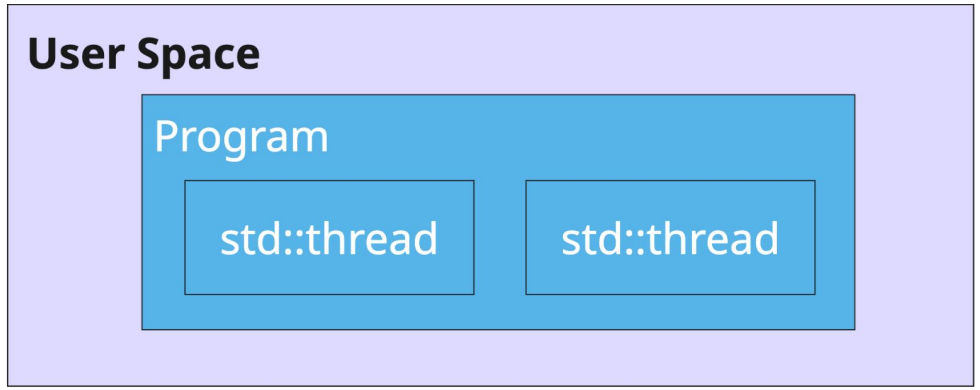
Threads



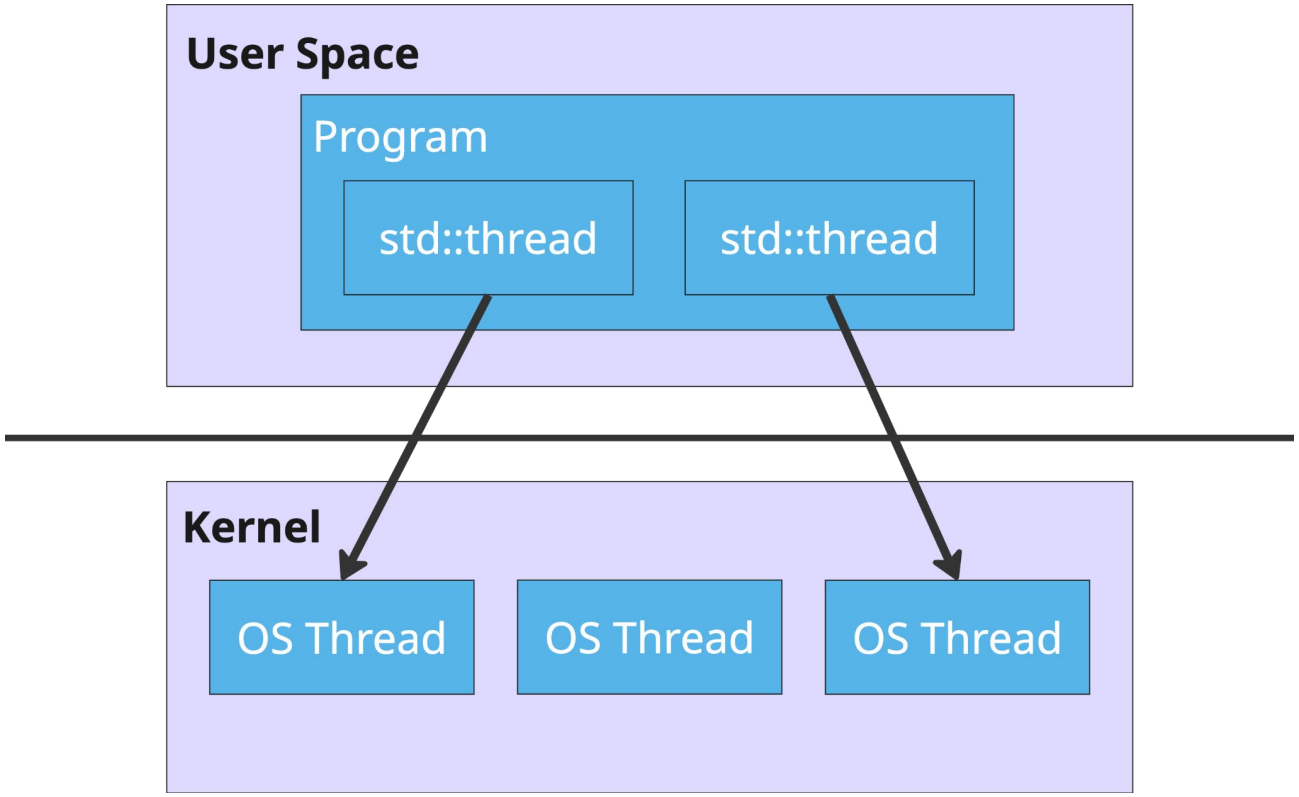

Throughput vs Number of Threads
Throughput (requests/sec)
0
5000
10000
15000
20000
25000
30000
35000
Number of Threads


Throughput vs Number of Threads
Throughput (requests/sec)
35000
30000
25000
20000
15000
10000
5000
0
Number of Threads

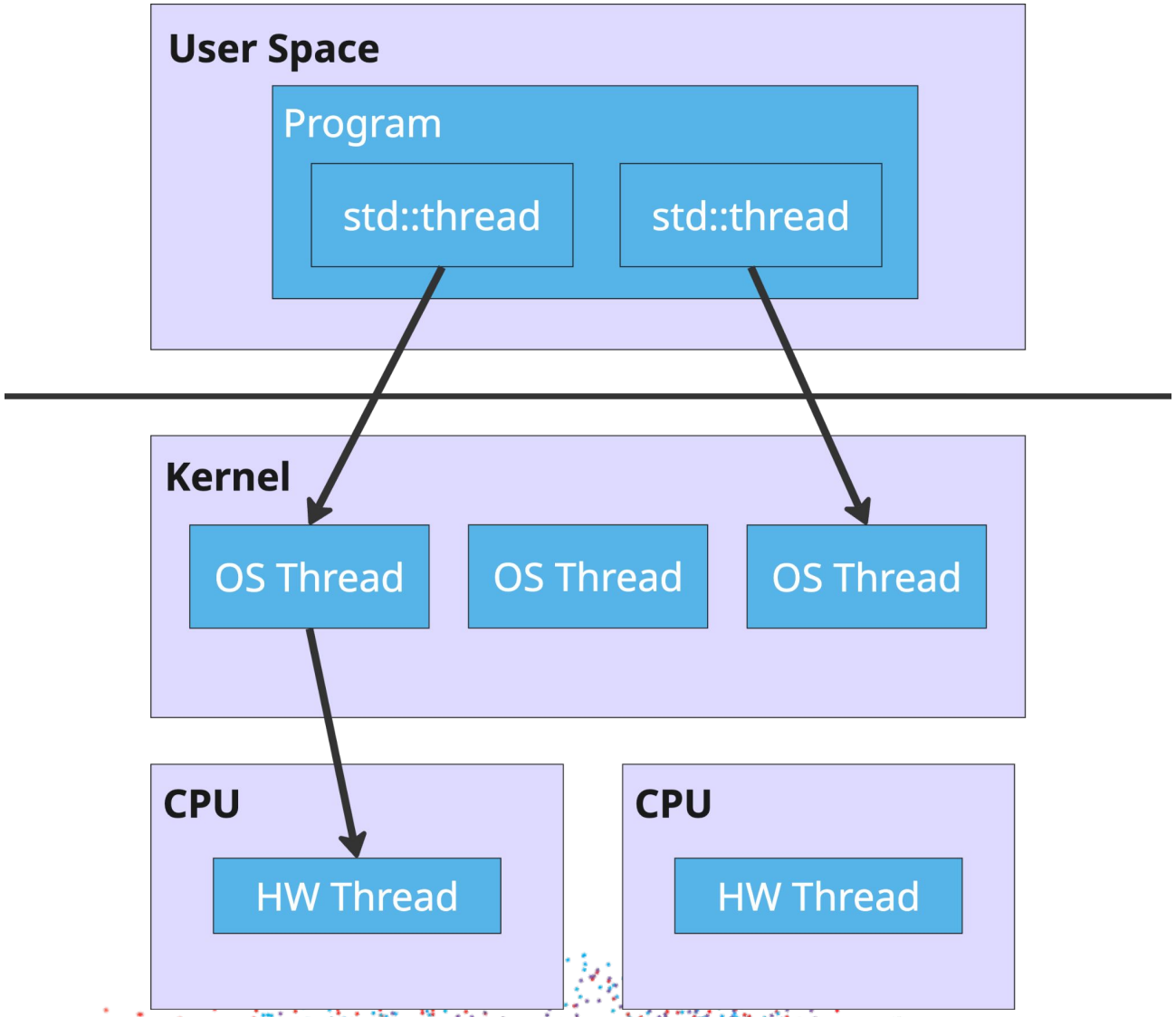
What Are User and Kernel Space?



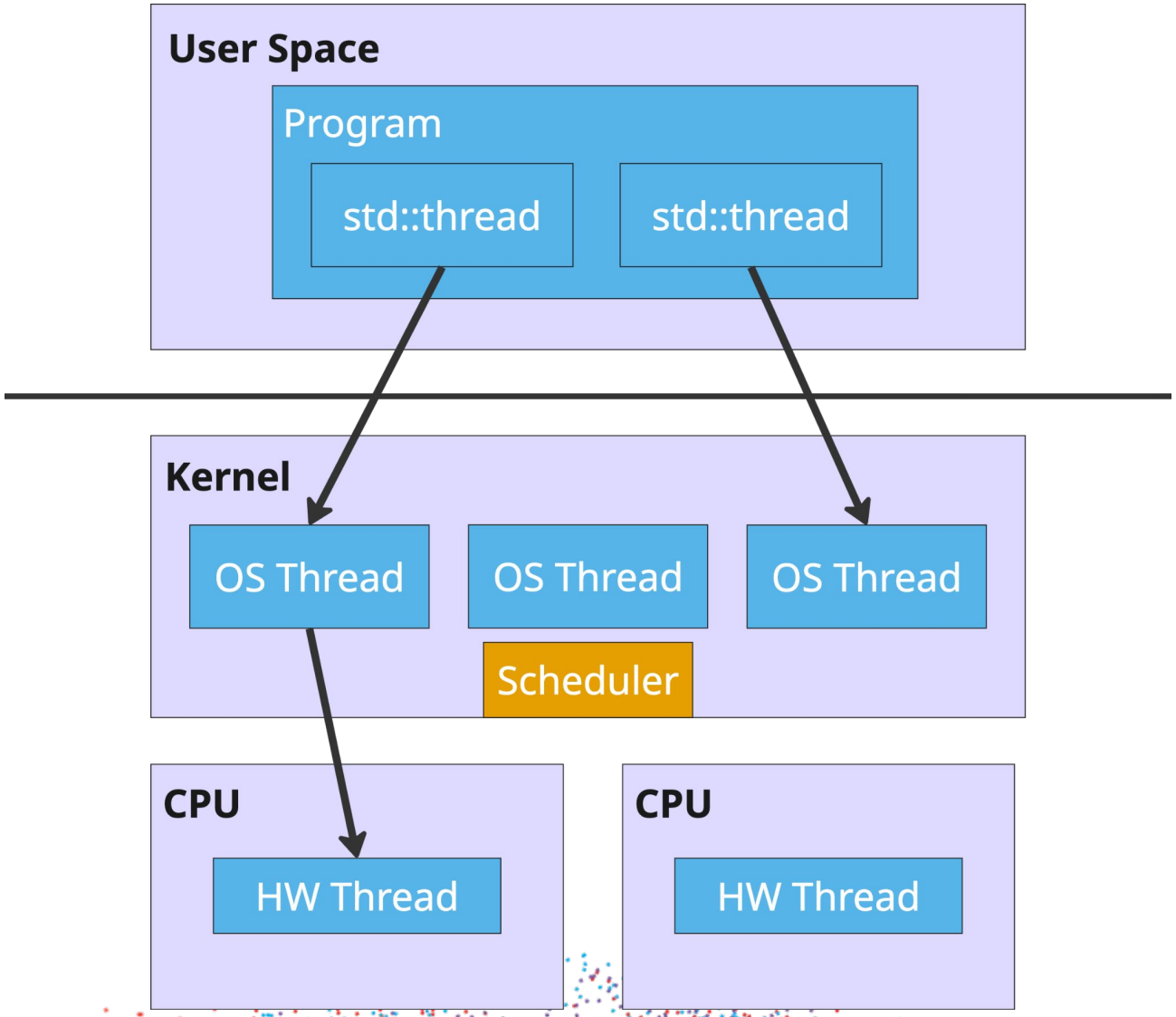
What Are User and Kernel Space?



What Are User and Kernel Space?



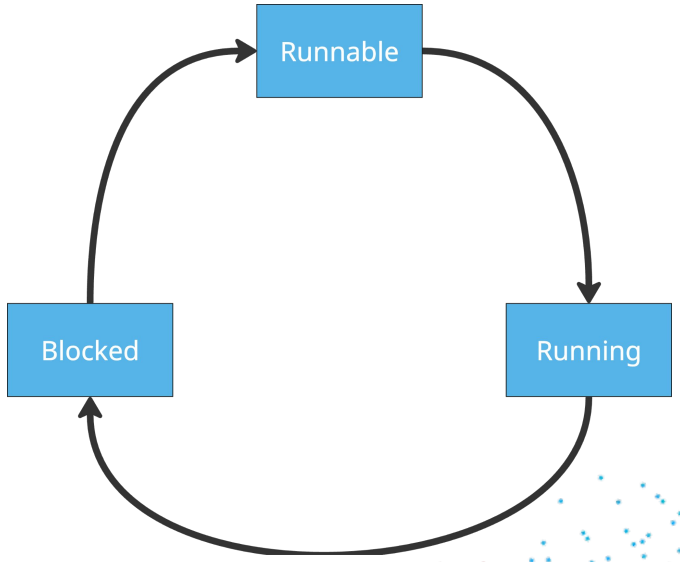
What Are User and Kernel Space?

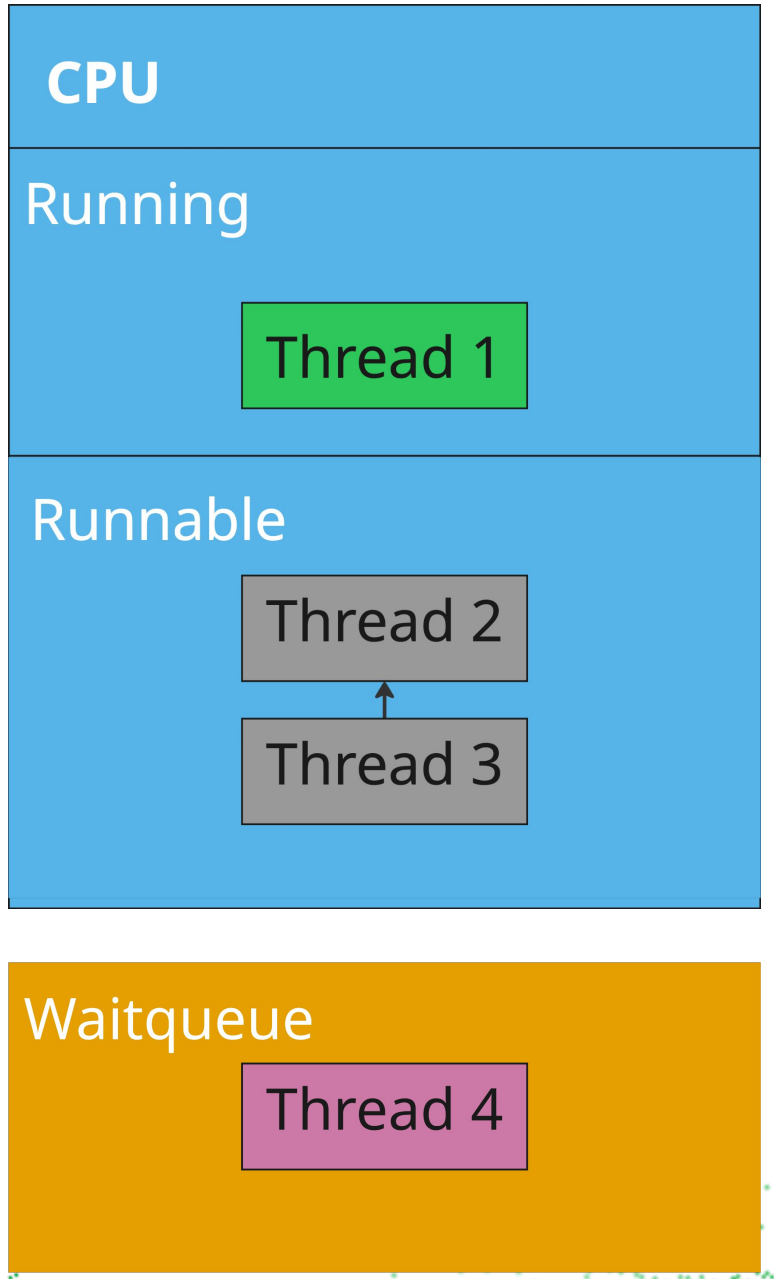


Thread State

At any moment, a thread is either:

- RUNNING on a CPU
- RUNNABLE, waiting to be scheduled
- BLOCKED, on a wait queue (I/O, futex, timers)





Thread Scheduling

- The *Scheduler* decides which thread runs on which core and for how long

```
std::thread a([](){ /* work a */ });
std::thread b([](){ /* work b */ });
```

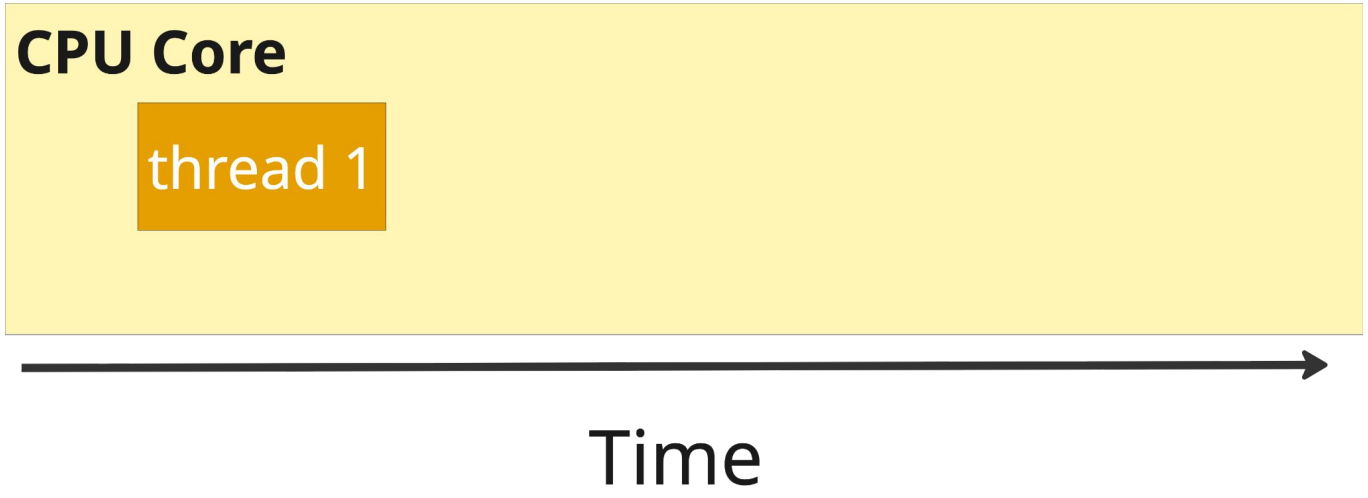
- Not deterministic as to whether a or b will run first
- Execution can be interleaved

Context Switches

- Involuntary (preemption)
- Voluntary (e.g., block on I/O)

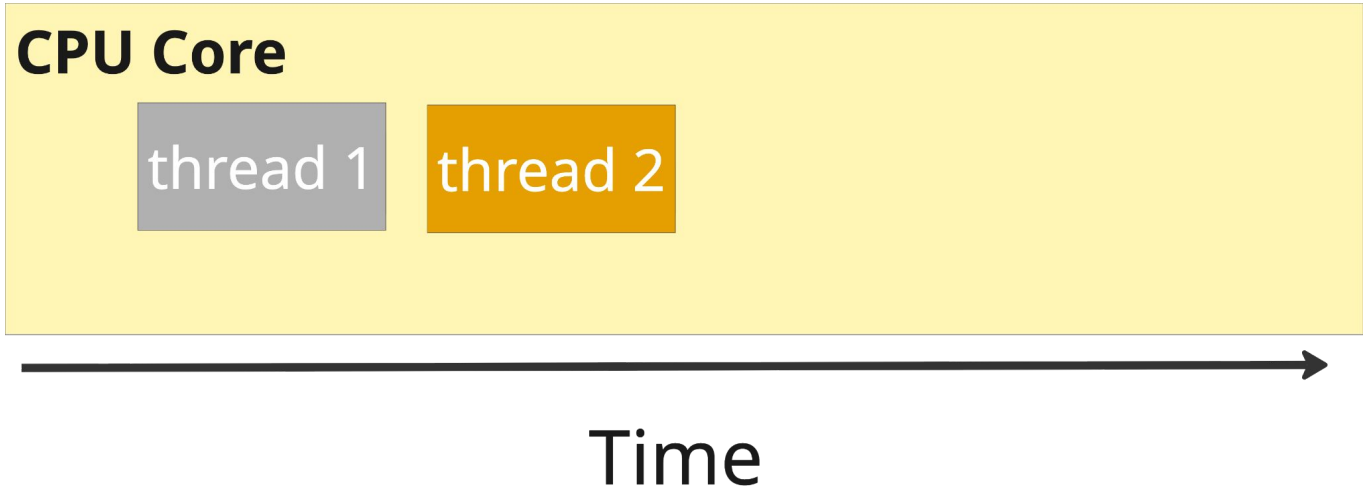
Preemption

- Time Slicing



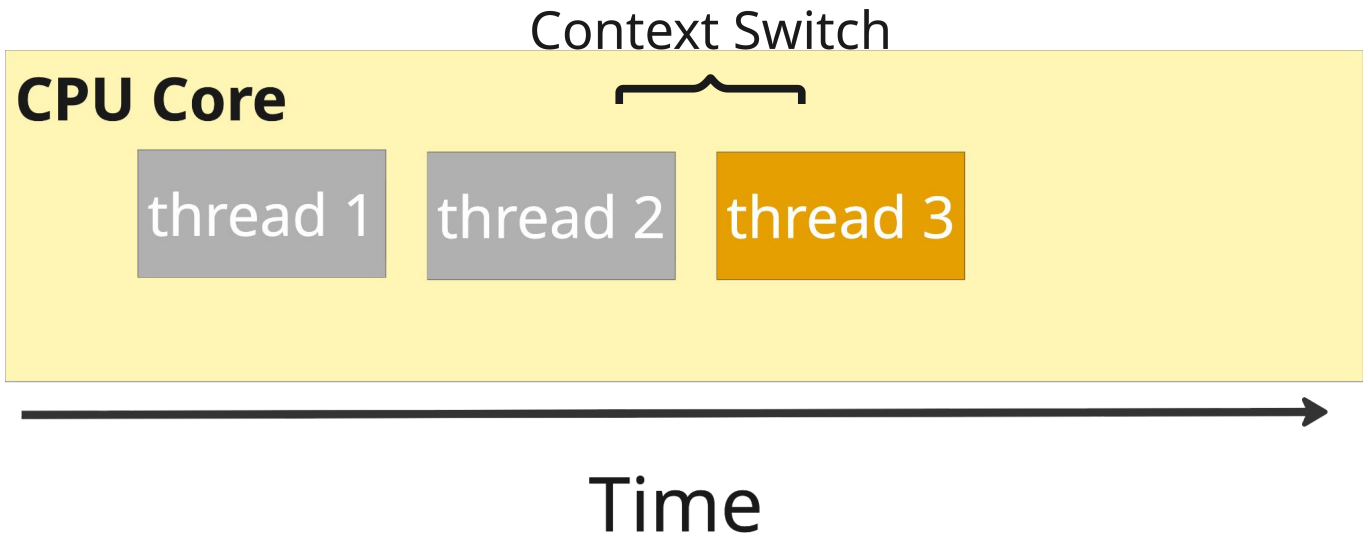
Preemption

- Time Slicing



Preemption

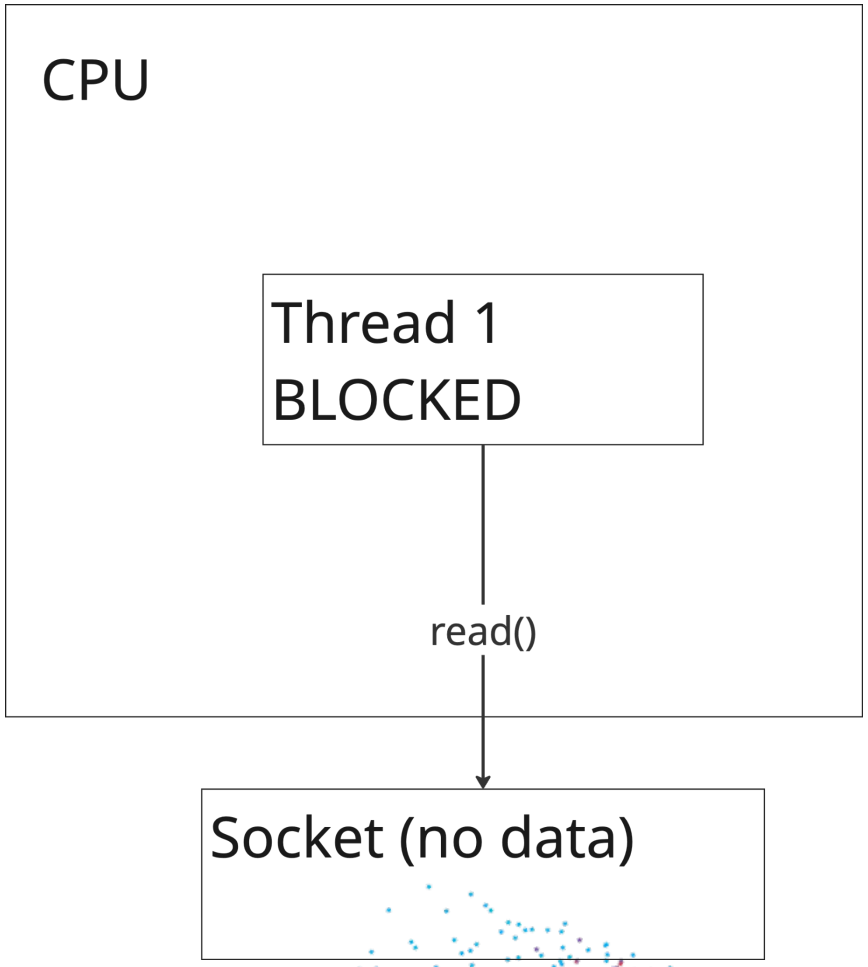
- Time Slicing



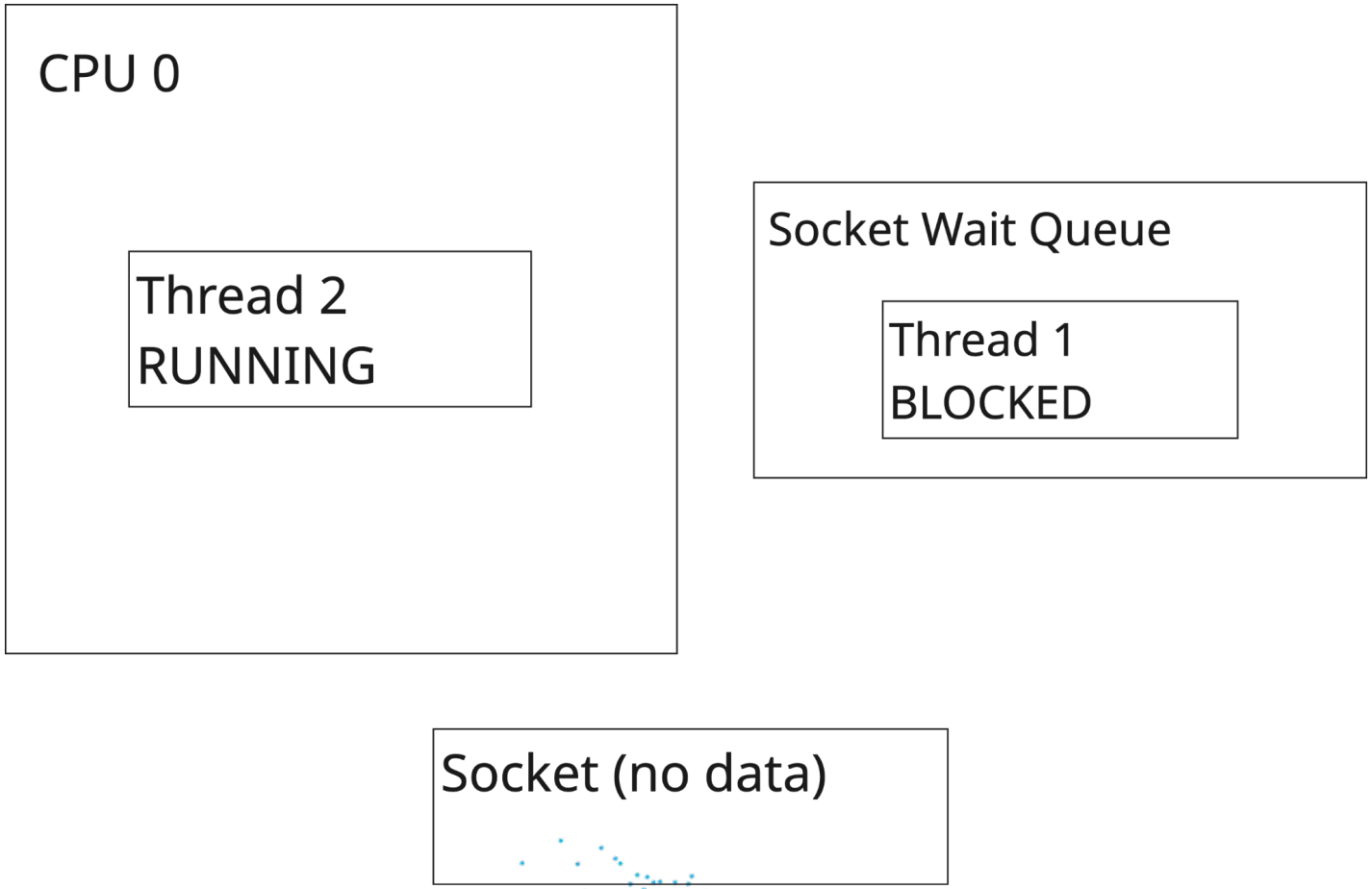
Blocking on I/O

```
int result = read(fd, buffer, sizeof(buffer));
```

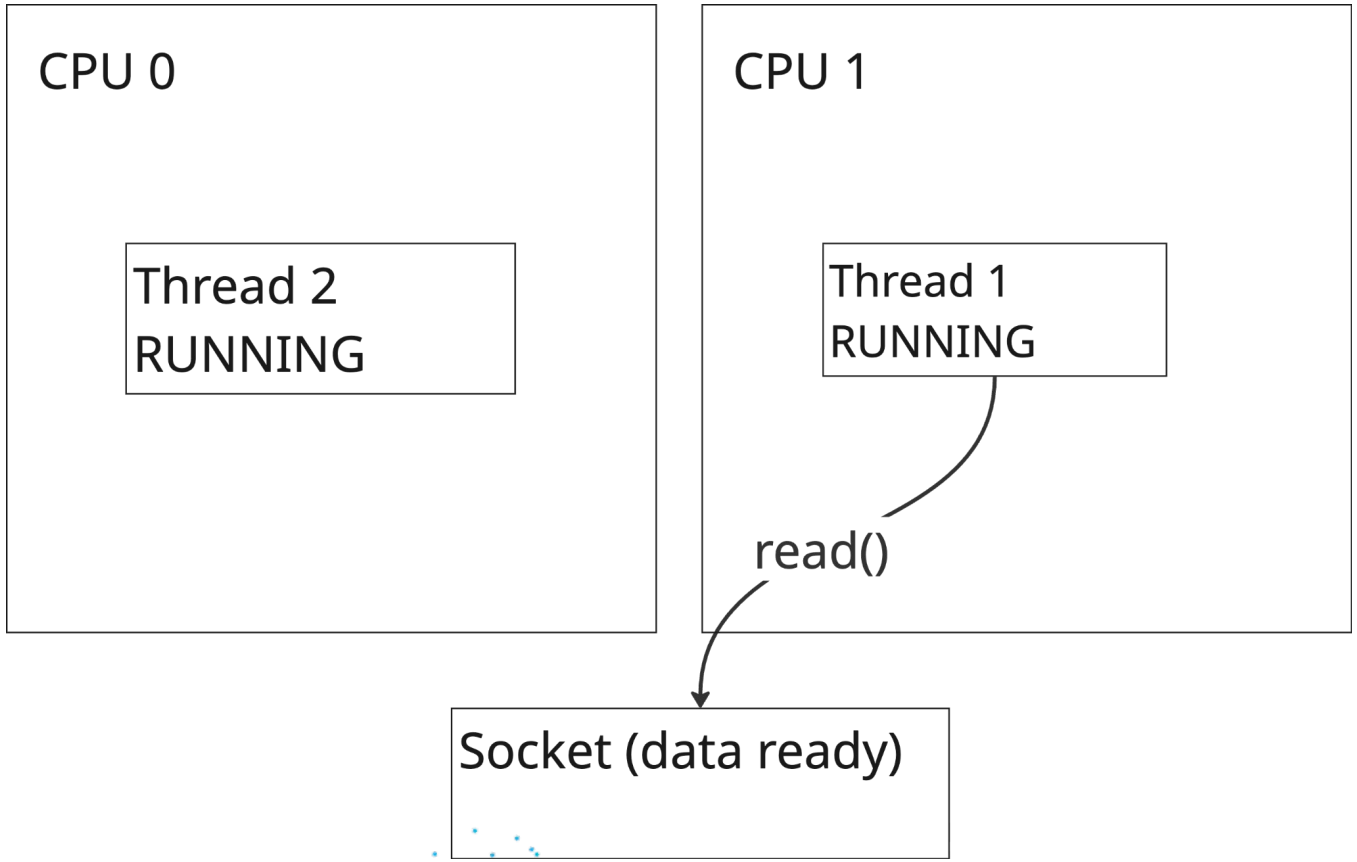
Blocking on I/O (what happens to the thread)



Blocking on I/O (what happens to the thread)

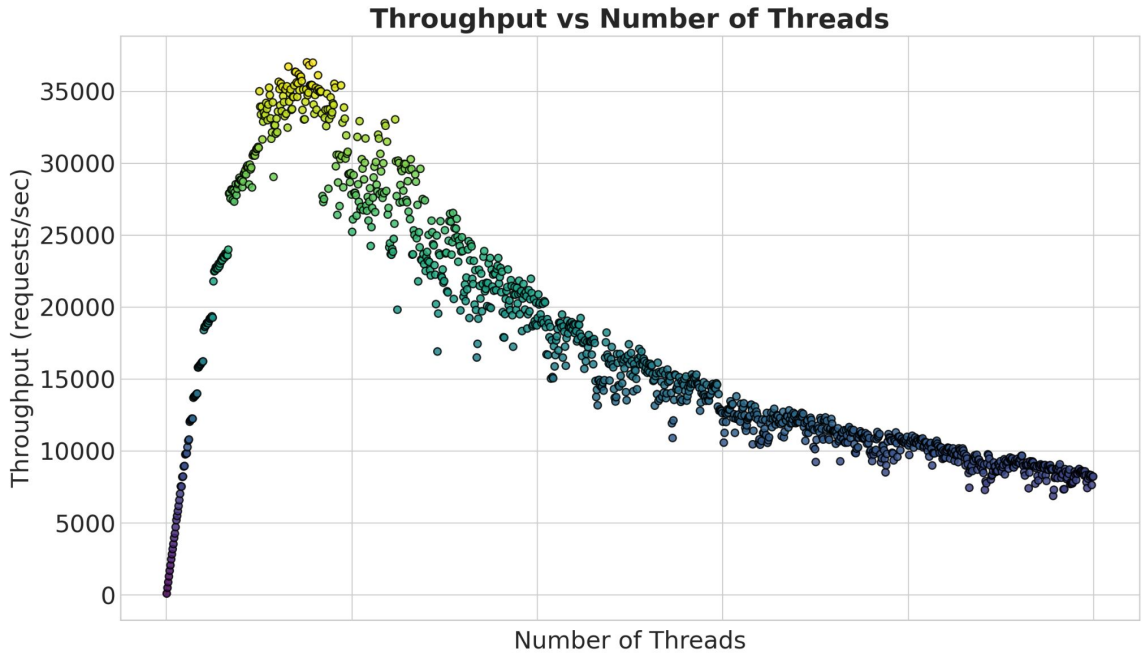


Blocking on I/O (what happens to the thread)



Where the cost really comes from

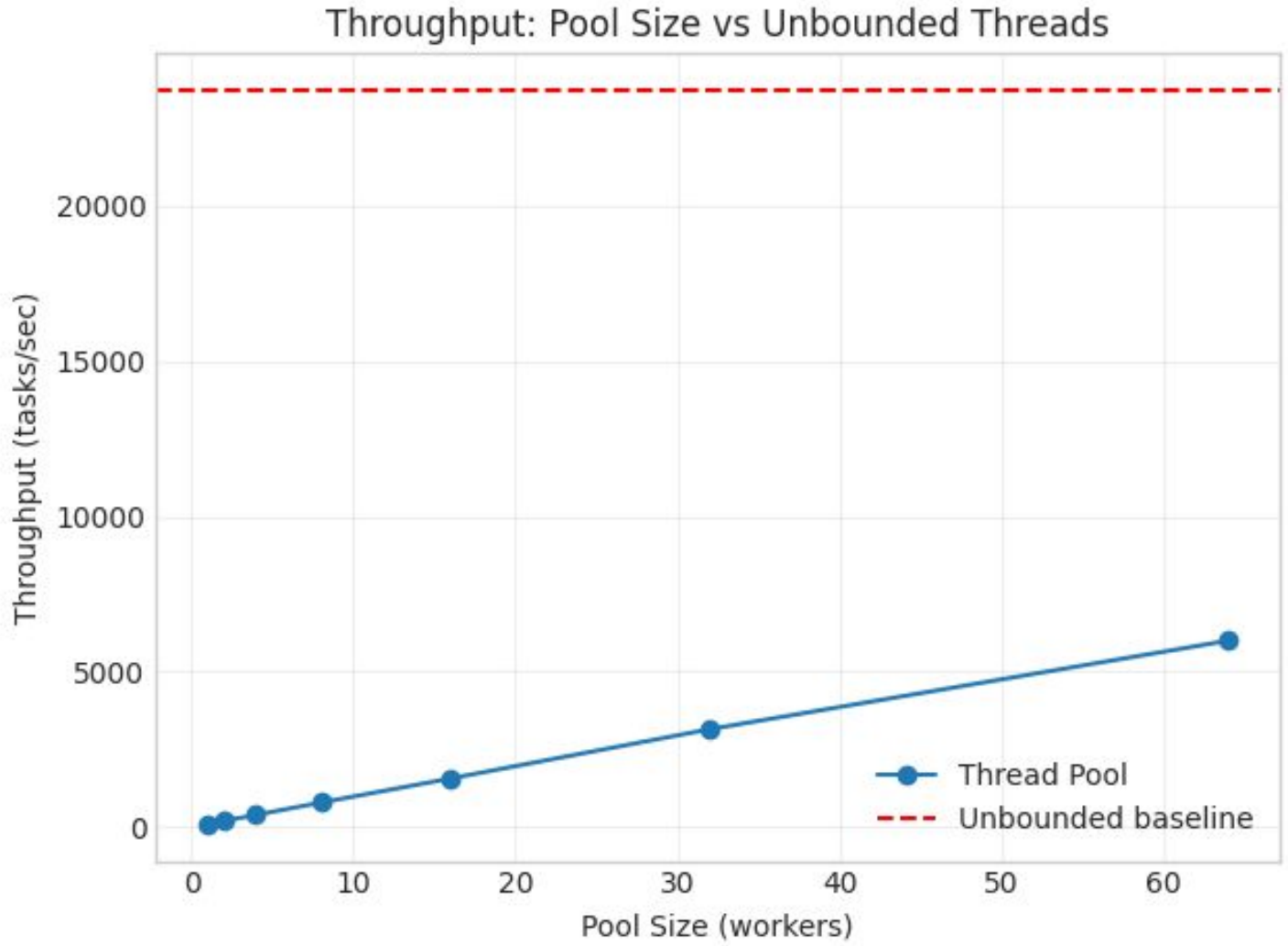
- Context switches prevent thread from completing its work until it is next scheduled
- The more threads, the more the scheduler tries to enforce fairness
- Oversubscription



Thread Pools

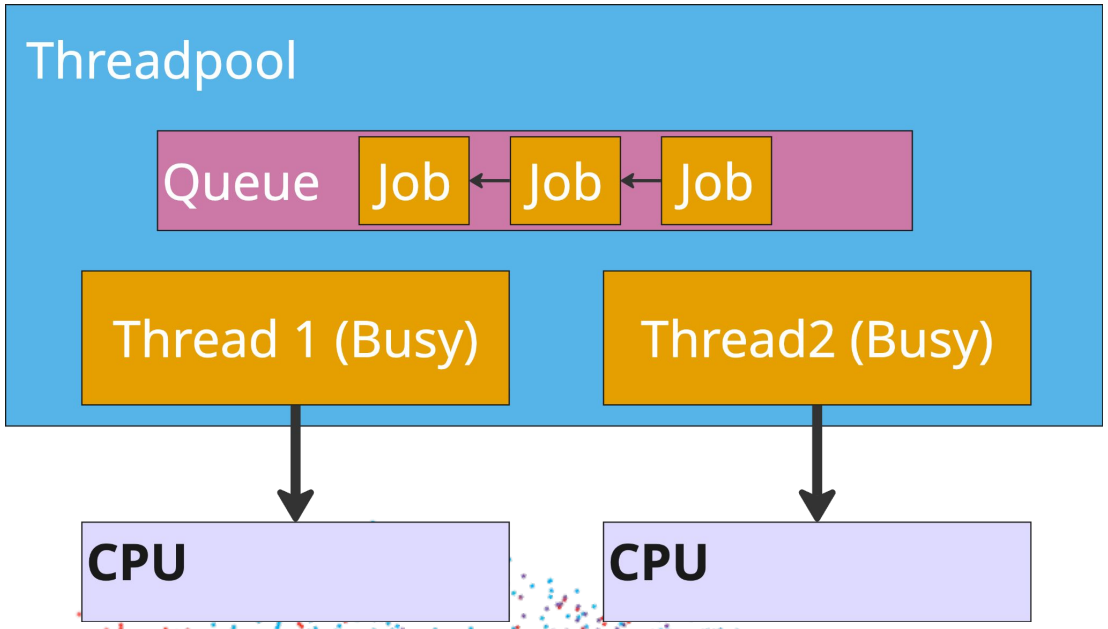
- Limit the number of threads
- Push work items into a queue
- Worker threads in pool pick up work and execute it

Thread Pools



Thread Pools

- *Parallelism* equal to the number of threads in our pool
- But each thread runs to completion, no interleaving
- Jobs are sitting in the queue to our pool
- Now limiting our *concurrency*
- Does not solve problem of blocking



Parallelism vs. Concurrency

Parallelism: Multiple tasks running at the exact same time

Concurrency: Multiple tasks in progress during the same time period, but not necessarily running at the same instant.

Domain Knowledge

- The domain knowledge lives in our application
- The OS does not know what is needed
- We know what behaviour we want to see
- Let’s start to improve what we have control over

Event Loops

- One thread

- Single loop for:
  - Wait for event
  - Dispatch event to handler

Event Loops (Slide code example)

```
class EventLoop {
 std::mutex mutex_;
 std::condition_variable cv_;
 std::queue<Task> ready_queue_;

 void run() {
   while (!runq.empty()) {
     auto task = ready_queue_.pop()
     task();  // do small, non-blocking work
   }
   cv_.wait();
 }
}
```

Non-Blocking

- Don’t wait for the work to be finished
- Need some way to figure out if it is finished
- Kernel has select/poll/epoll

…But

- The code gets more challenging
- Fixes how we wait, but not how we write the waiting code

Coroutines

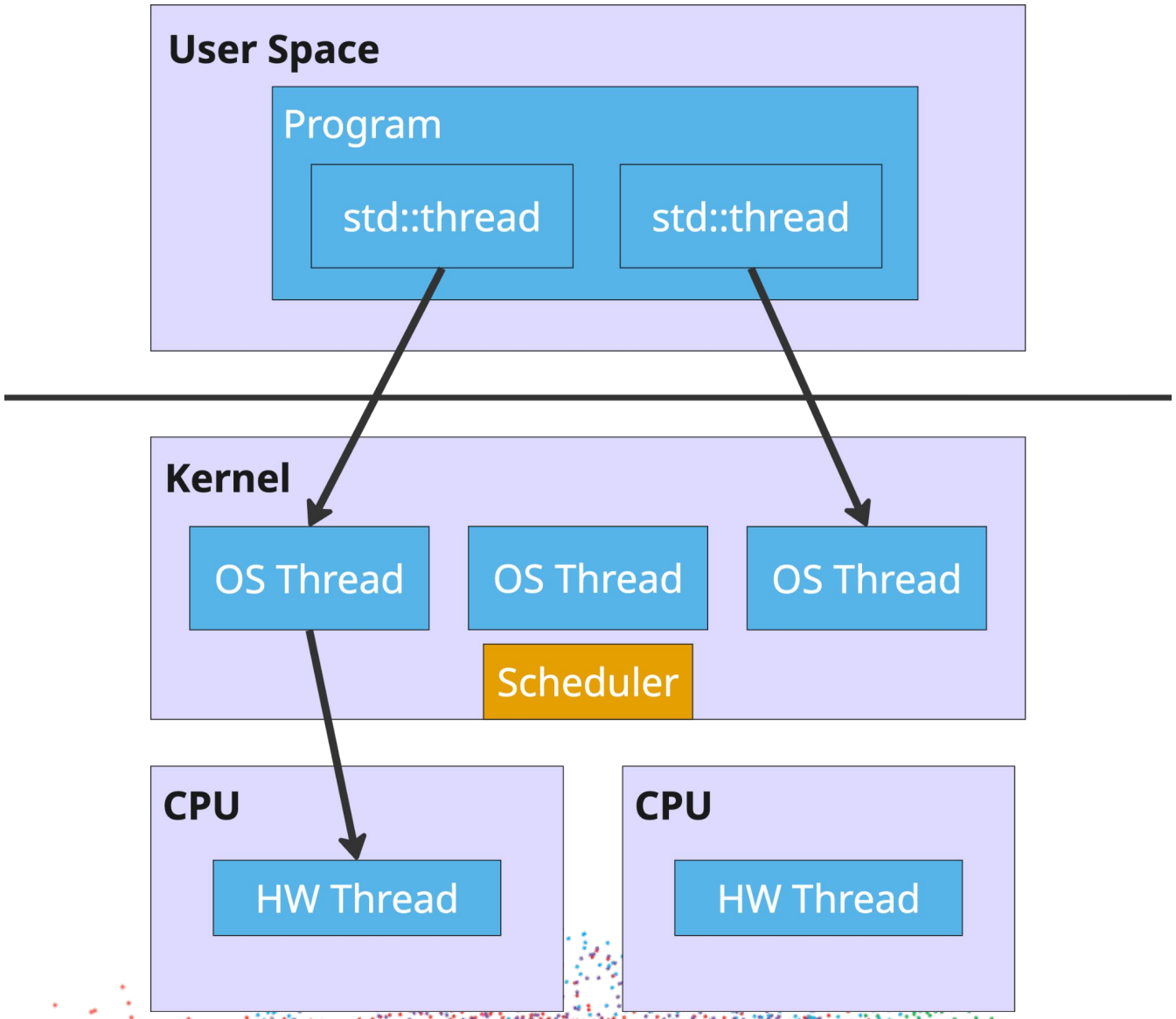
What is a Coroutine?

- A function that can suspend itself (and be resumed)

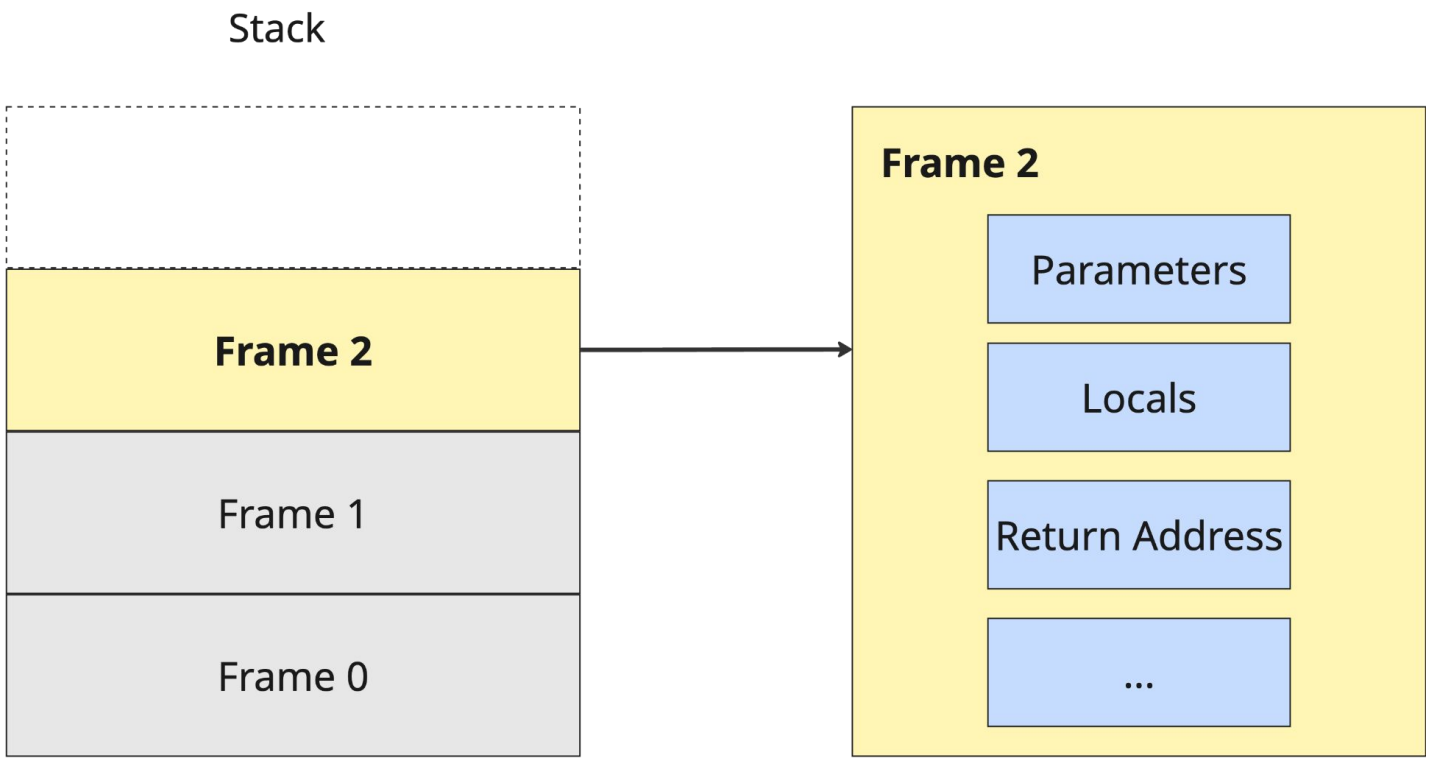
```
co_await async_read();
```

- Suspends function until there is a result from “async_read”

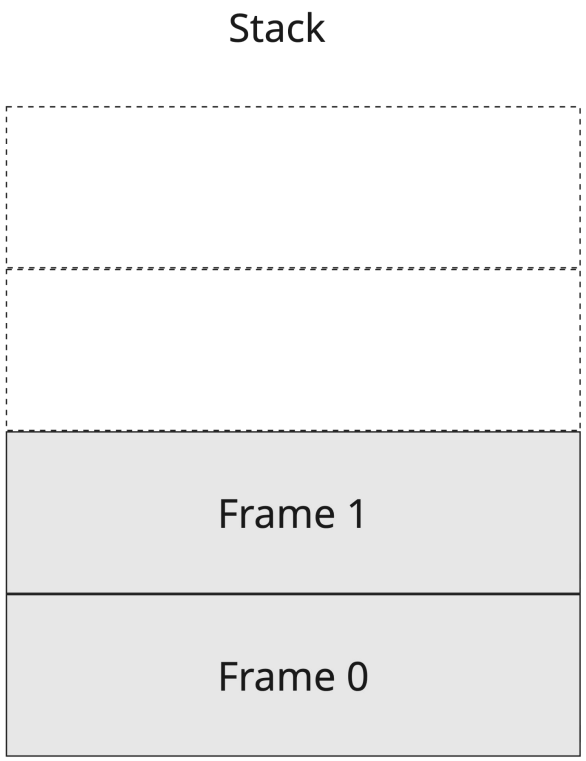
User, Kernel, and CPUs revisited



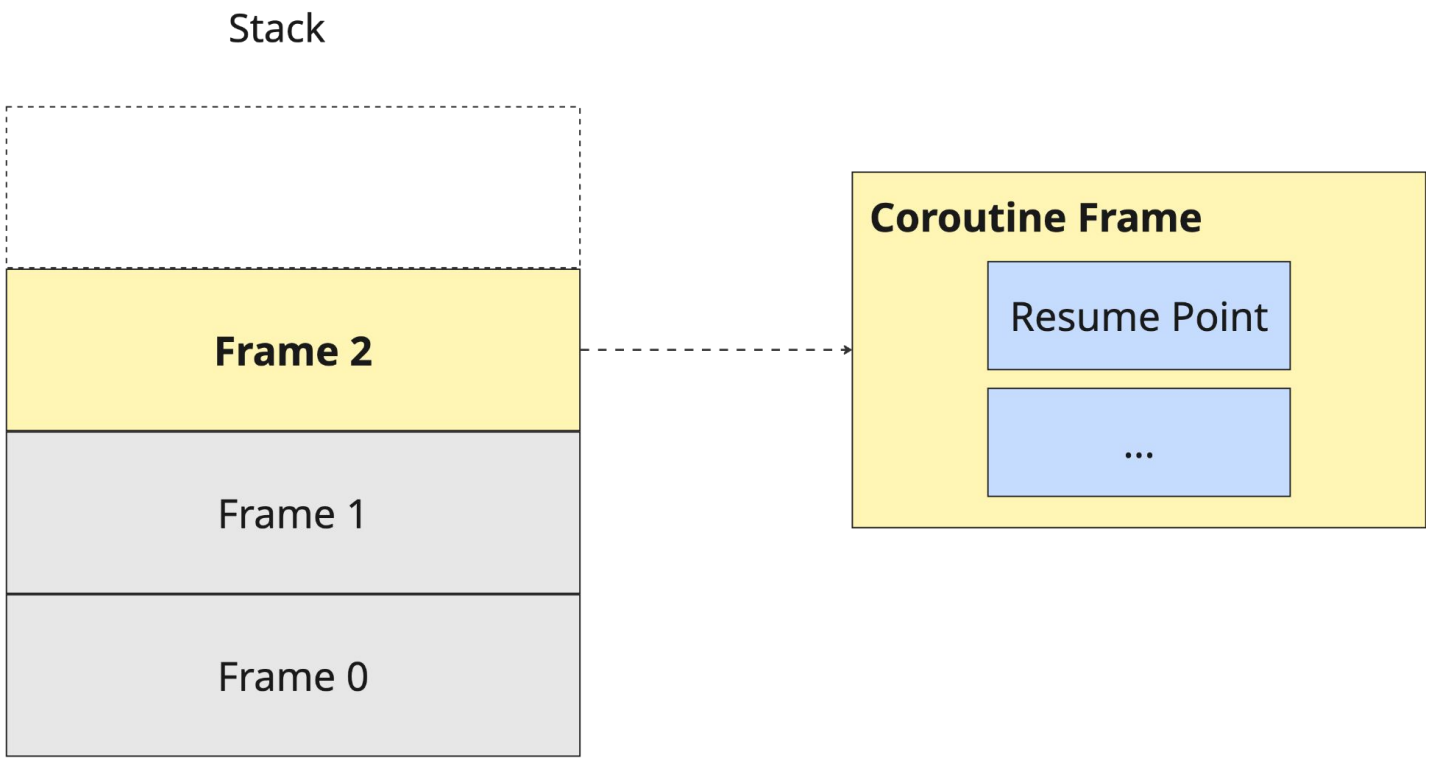
Function Calling



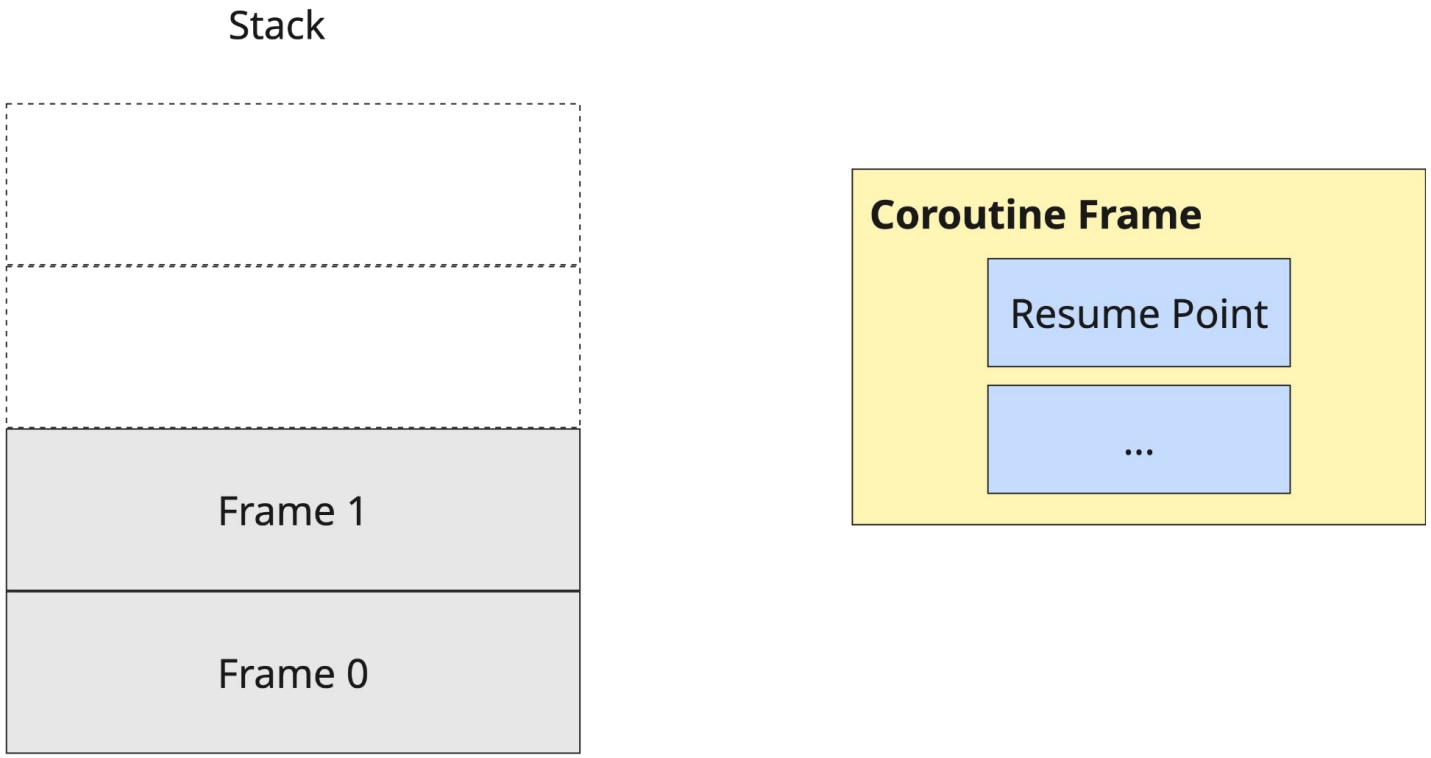
Function Calling



Coroutine Frames

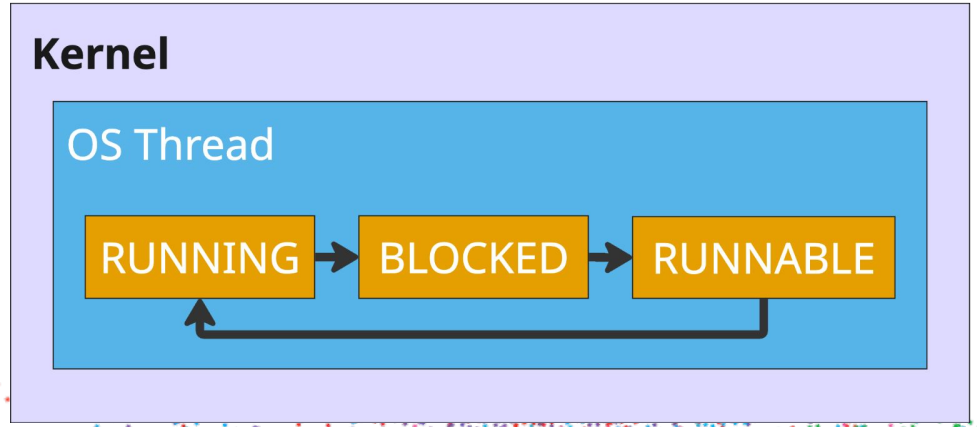


Coroutine Frames



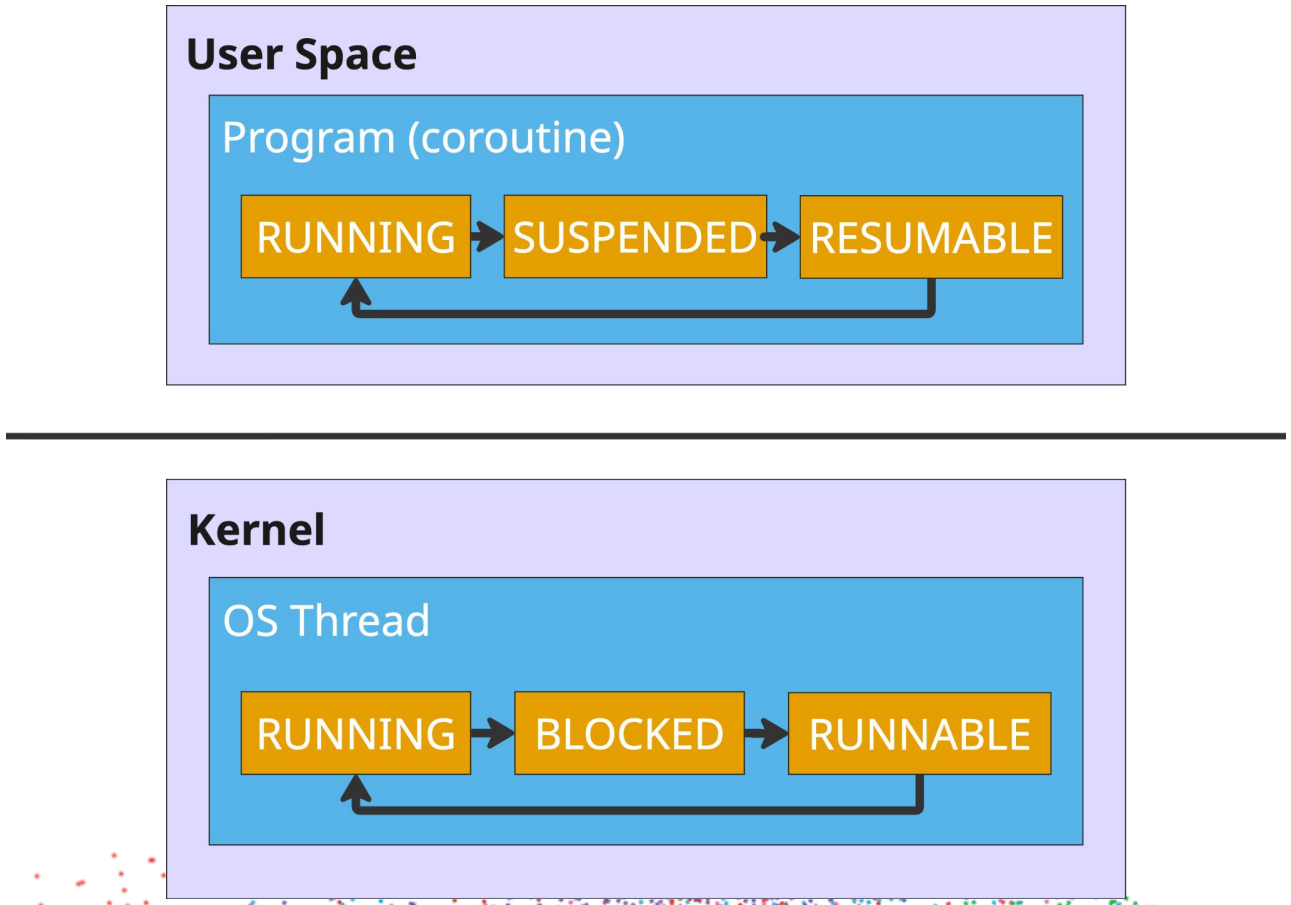
The switching is in User Space!

- Compiler generates state machine
- Cost of “Context Switching” comparable to function call
- Our application chooses when to run it



The switching is in User Space!

- Compiler generates state machine
- Cost of “Context Switching” comparable to function call
- Our application chooses when to run it



Scheduling

Return to our event loop from earlier.

```
void work_func() { /* work */ }


int main() {
 EventLoop loop;
 loop.post(work_func);
 loop.run();
}
```

Scheduling

Return to our event loop from earlier.

```
Task coroutine(EventLoop& loop) {
 co_await Awaitable{loop}; // suspends coroutine, event loop keeps running
}

int main() {
 EventLoop loop;
 coroutine(loop); // call our coroutine function
 loop.run();
}
```

What is wrong with this?

```
Task coroutine(...) {
  …
  BlockingRead(...);
  …
}
```

What is wrong with this?

```
Task coroutine(...) {
  co_await BlockingRead(...);
}
```

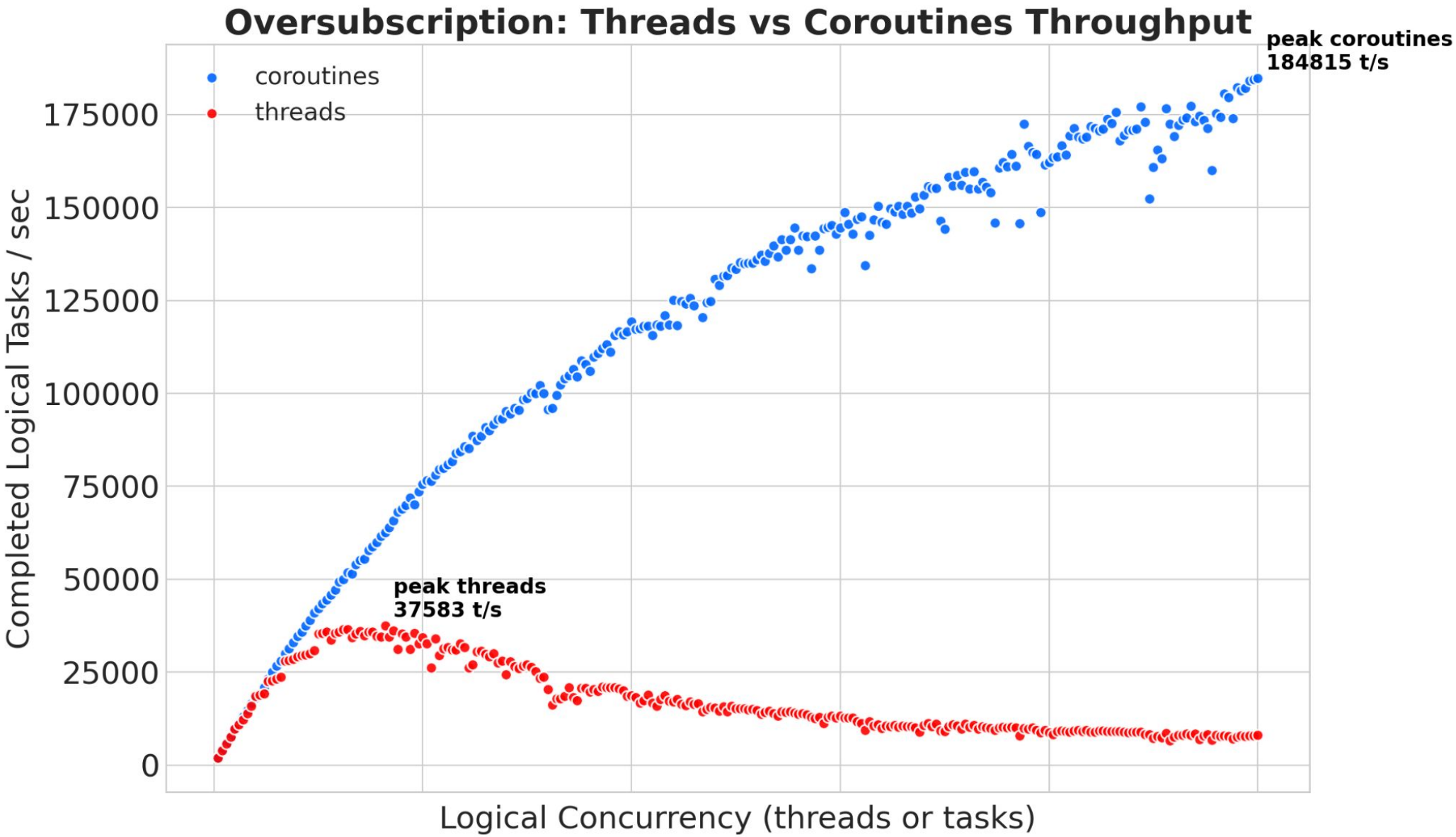
- Starvation!
- **Cooperative** multitasking: everyone must play nice!

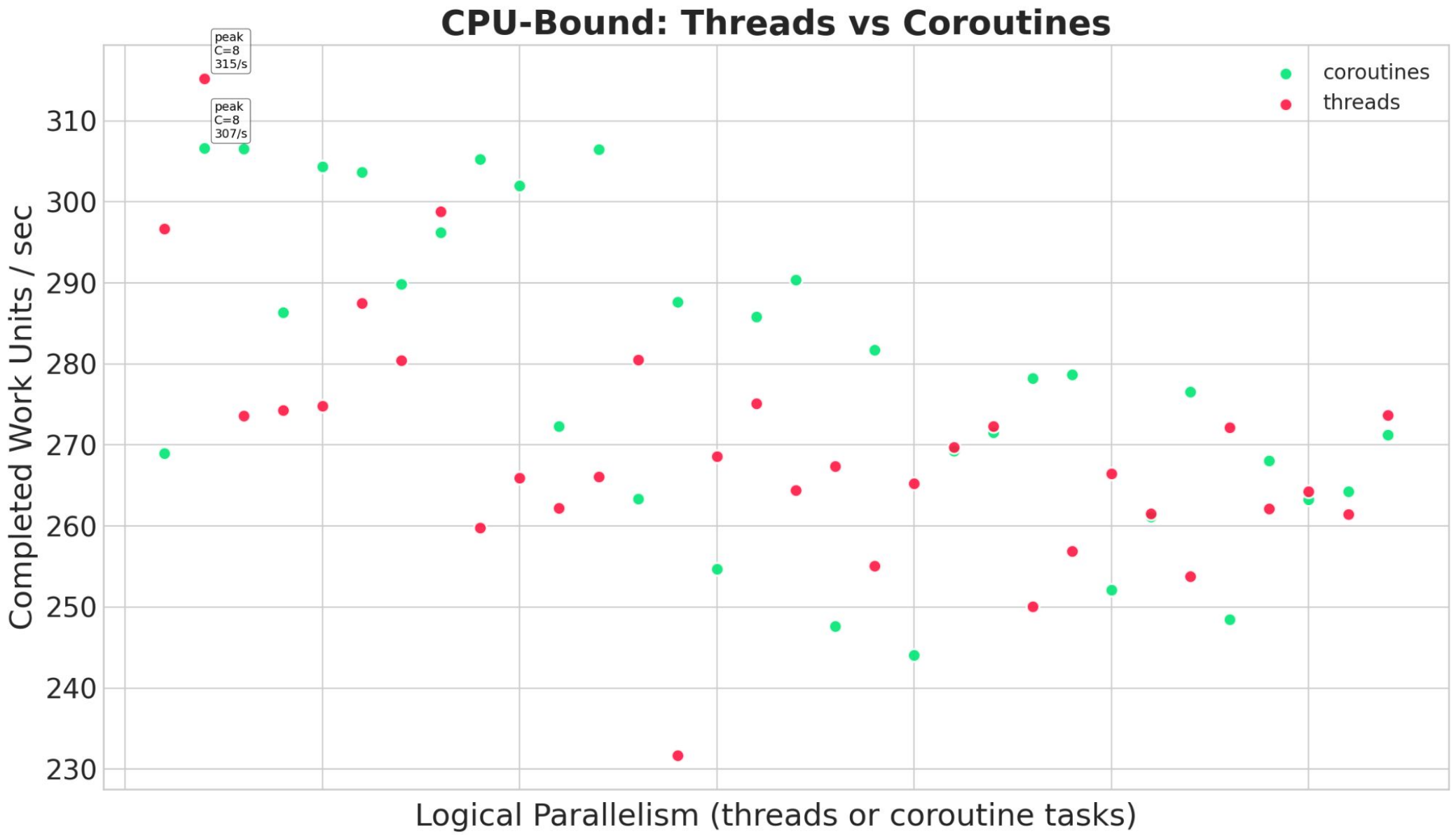
The type of work matters

- The application we are building is doing “work”
- I/O bound work?
- CPU bound work?

I/O Bound and Non-Blocking Work

- Bottleneck is *waiting for data* from outside the CPU
- I/O bound work involves a lot of waiting
- Coroutines can suspend cheaply while waiting
- Leaves the CPU to do other things
- E.g. reading/writing from a socket, waiting for a database query to complete, calling a remote API over the network.


Oversubscription: Threads vs Coroutines Throughput
coroutines
threads
peak coroutines
184815 t/s
peak threads
37583 t/s
Completed Logical Tasks / sec
175000
150000
125000
100000
75000
50000
25000
0
Logical Concurrency (threads or tasks)


CPU-Bound: Threads vs Coroutines
peak
C=8
315/s
peak
C=8
307/s
coroutines
threads
Completed Work Units / sec
310
300
290
280
270
260
250
240
230
Logical Parallelism (threads or coroutine tasks)

CPU Bound Work

- Threads (and threadpools) if bottleneck is computation
- Program already has the data, no waiting involved
- Examples: working with large in-memory datasets, or parsing or compressing a large file

Combination

- Coroutines can orchestrate I/O between stage
- Hand off any CPU work to the thread pool

```
Task handle_request(...) {
    auto data = co_await async_read(...); // I/O wait
    auto result = co_await schedule_on(cpu, process(data)); // CPU work
    co_await async_write(..., result); // I/O again
}
```

C++ 26 and std::execution

- Composable building blocks:
  - “what to do” (your work)
  - “where to do it” (the scheduler/executor)
- Make the *I/O-bound vs CPU-bound* distinction explicit

```
auto task = async_read(s)
          | stdexec::on(io)     // wait for I/O
          | stdexec::then(process)
          | stdexec::on(cpu)    // crunch numbers
          | stdexec::then(async_write);
```

Summing Up

- **Threads**
  - OS decides which thread runs on which CPU
  - Neccessary for parallelism up to N cores, concurrency beyond that
  - Preemptive, does not scale
  - Expensive (stack, kernel state)
- **Coroutines:**
  - Scheduled in user space
  - Cooperative: ensure tasks yield
  - Executors/Schedulers

Thank you!

https://www.bloomberg.com/careers
https://bloomberg.avature.net/careers/JobDetail/13661

TechAtBloomberg: “Meet the C++ Guild” article