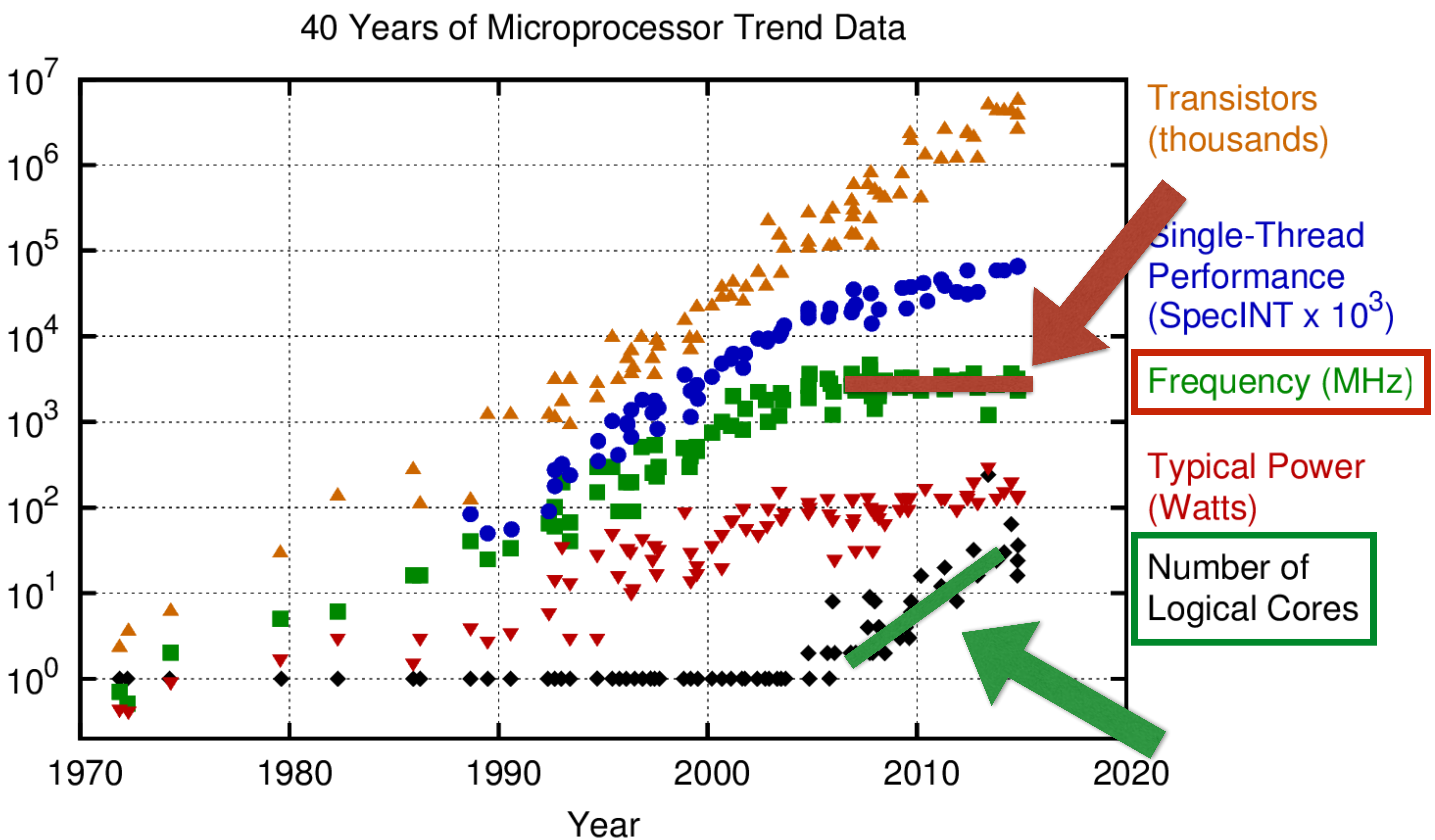
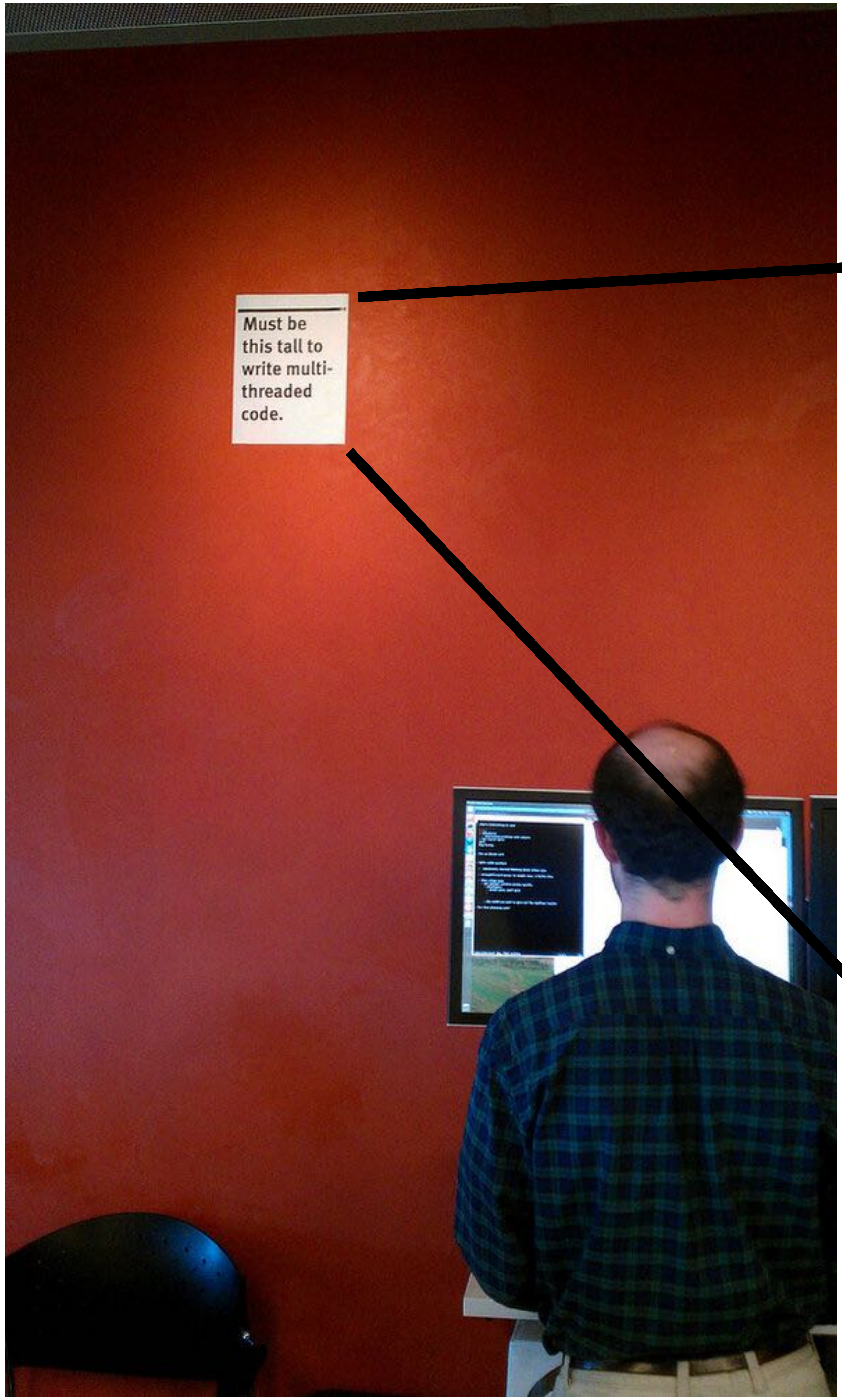
Concurrency in Rust

Alex Crichton



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp


Bug 631527
Parallelize selector matching
Get help with this page
NEW Assigned to dzbarsky
Status (NEW bug with no priority)
Product: Core
Component: CSS Parsing and Computation
Status: NEW
Reported: 7 years ago
Reported: 7 years ago



Must be this tall to write multi-threaded code.

Fearless Concurrency with Rust

Apr 10, 2015 • Aaron Turon

The Rust project was initiated to solve two thorny problems:

- How do you do safe systems programming?
- How do you make concurrency painless?

Initially these problems seemed orthogonal, but to our amazement, the solution turned out to be identical: **the same tools that make Rust safe also help you tackle concurrency head-on**.

Memory safety bugs and concurrency bugs often come down to code accessing data when it shouldn't. Rust's secret weapon is *ownership*, a discipline for access control that systems programmers try to follow, but that Rust's compiler checks statically for you.

What Rust has to offer

Strong safety guarantees…

No seg-faults, no data-races, expressive type system.

…without compromising on performance.

No garbage collector, no runtime.

Goal:

Confident, productive systems programming

Concurrency?

Rust?

Libraries

Futures

What’s concurrency?

In computer science, concurrency is a property of systems in which several computations are executing simultaneously, and potentially interacting with each other.

Getting our feet wet

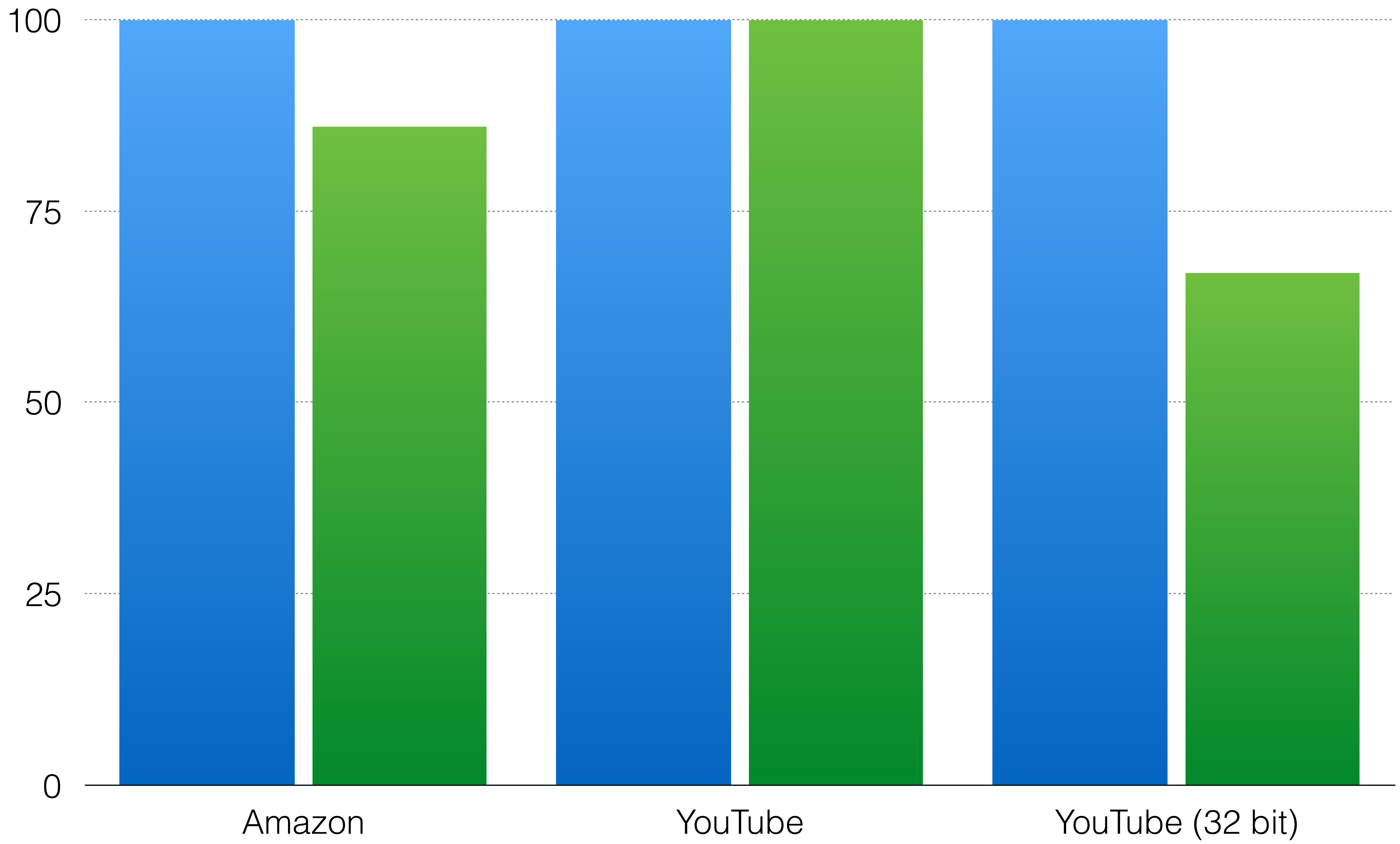
```
// What does this print?
int main() {
    int pid = fork();
    printf("%d\n", pid);
}
```

Concurrency is hard!

- Data Races
- Race Conditions
- Deadlocks
- Use after free
- Double free

Exploitable!

Concurrency is nice!



Concurrency?

Rust?

Libraries

Futures

Zero-cost abstractions

+

Memory safety & data-race freedom

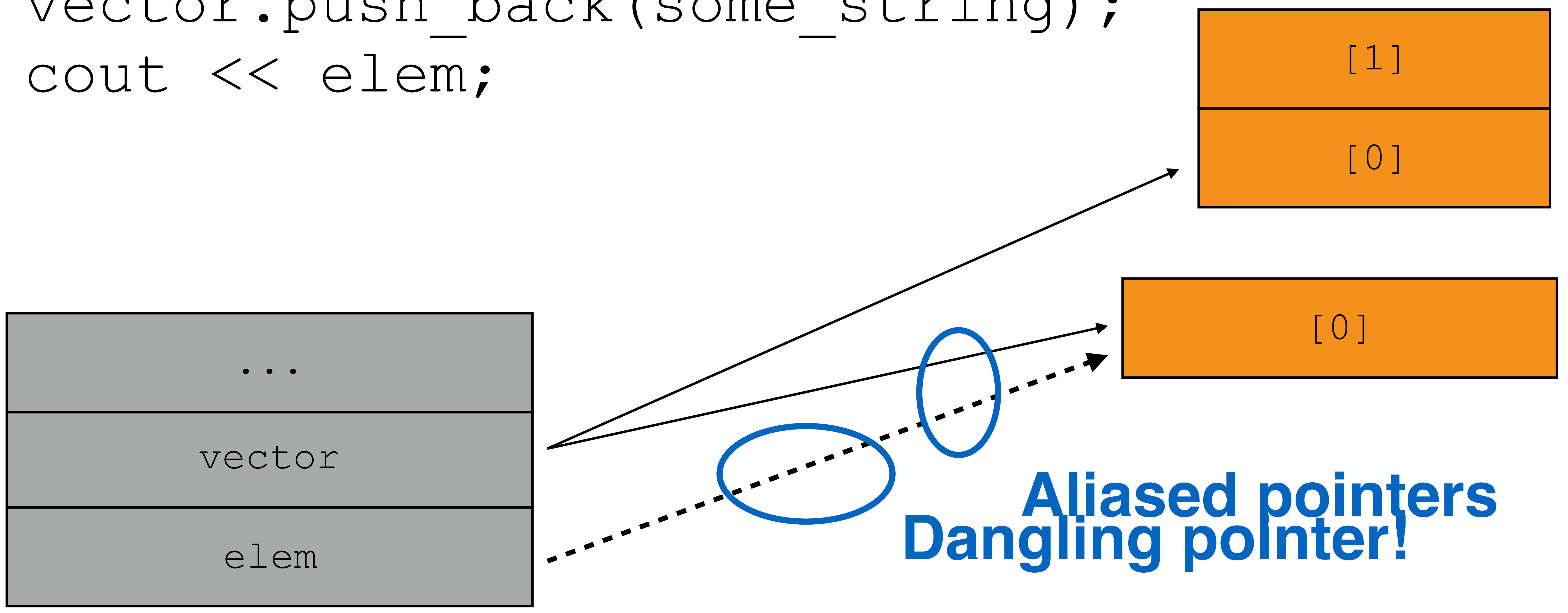
=

Confident, productive systems programming

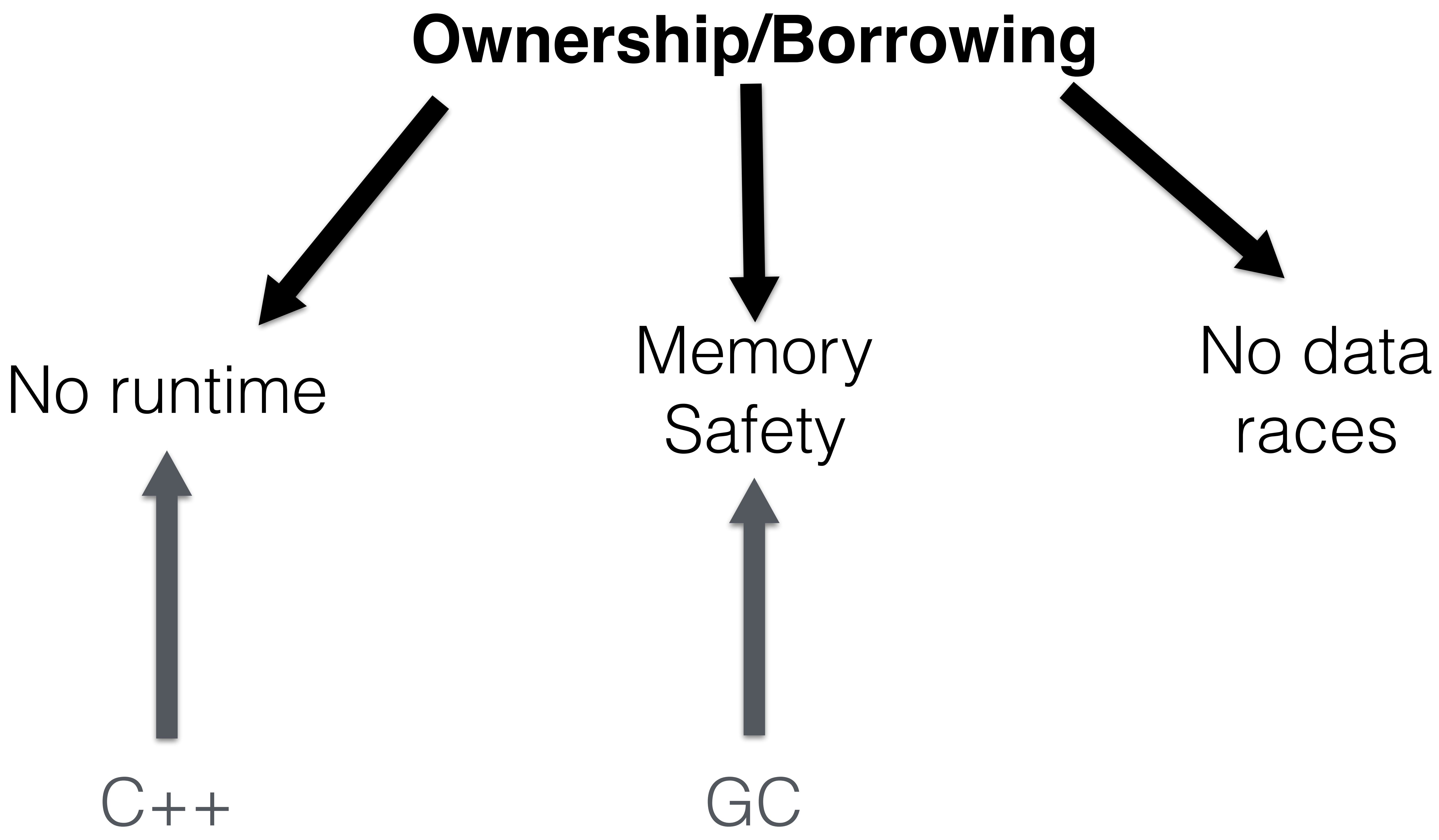
What's **safety**?

```
void example() {
  vector<string> vector;
  // ...
  auto& elem = vector[0];
  vector.push_back(some_string);
  cout << elem;
}
```

Mutation



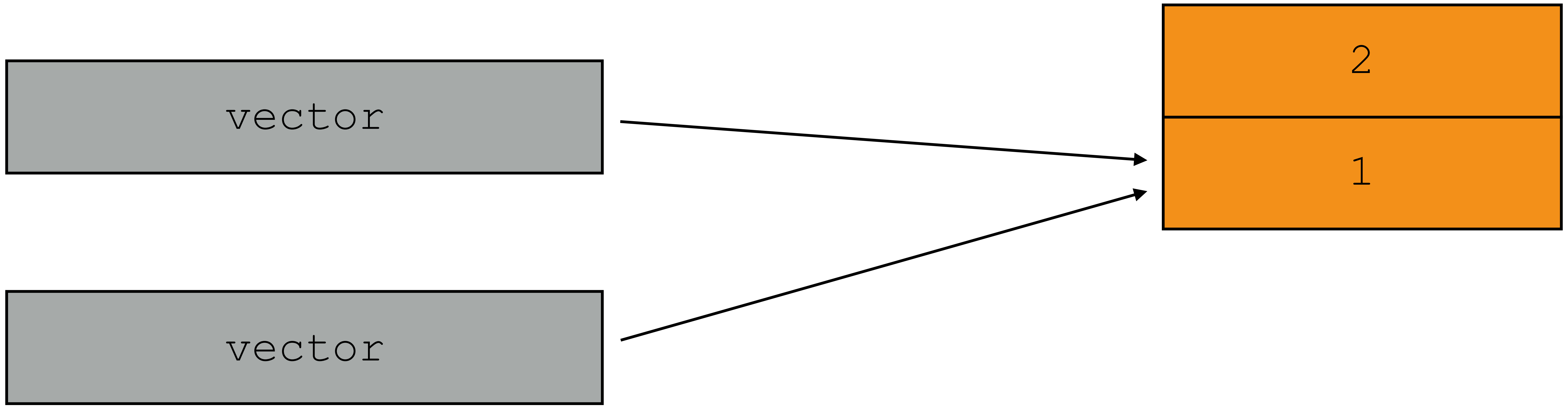
Rust’s Solution



Ownership

```
fn main() {
    let mut v = Vec::new();
    v.push(1);
    v.push(2);
    take(v);
    // ...
}
```

```
fn take(v: Vec<i32>) {
    // ...
}
```



Ownership

```
fn main() {
    let mut v = Vec::new();
    v.push(1);
    v.push(2);
    take(v);
    // ...
}
```

```
fn take(v: Vec<i32>) {
    // ...
}
```

Ownership

```rust
fn main() {
    let mut v = Vec::new();
    v.push(1);
    v.push(2);
    take(v);
    v.push(3);
}
```

```rust
fn take(v: Vec<i32>) {
    // ...
}
```

Ownership

```
fn main() {
    let mut v = Vec::new();
    v.push(1);
    v.push(2);
    take(v);
    v.push(3);
}
```

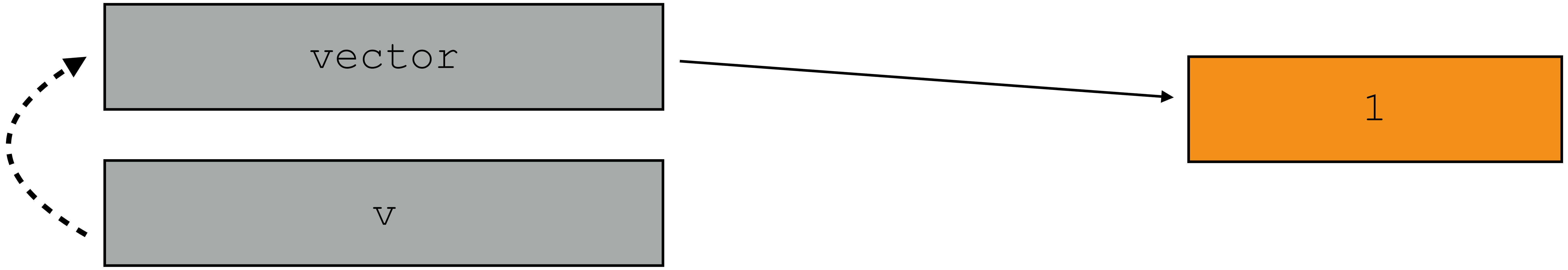
```
fn take(v: Vec<i32>) {
    // ...
}
```

error: use of moved value `v`

Borrowing

```
fn main() {
    let mut v = Vec::new();
    push(&mut v);
    read(&v);
    // ...
}
```

```
fn push(v: &mut Vec<i32>) {
    v.push(1);
}
```



Safety in Rust

- Rust *statically* prevents aliasing + mutation
- Ownership prevents double-free
- Borrowing prevents use-after-free
- Overall, no segfaults!

Data races

Sound familiar?

Sharing

\+ Mutation

\+ No ordering

Data race

Concurrency?

Rust?

Libraries

Futures

Library-based concurrency

Originally: Rust had message passing built into the language.

Now: library-based, multi-paradigm.

Libraries leverage **ownership and borrowing** to avoid data races.

std::thread

```
let loc = thread::spawn(|| {
    “world”
});
println!(“Hello, {}!”,
          loc.join().unwrap());
```

std::thread

```
let mut dst = Vec::new();
thread::spawn(move || {
    dst.push(3);
});
dst.push(4);
```

error: use after move

std::thread

```
let mut dst = Vec::new();
thread::spawn(|| {
    dst.push(3);
});
dst.push(4);
```

error: value doesn't live long enough

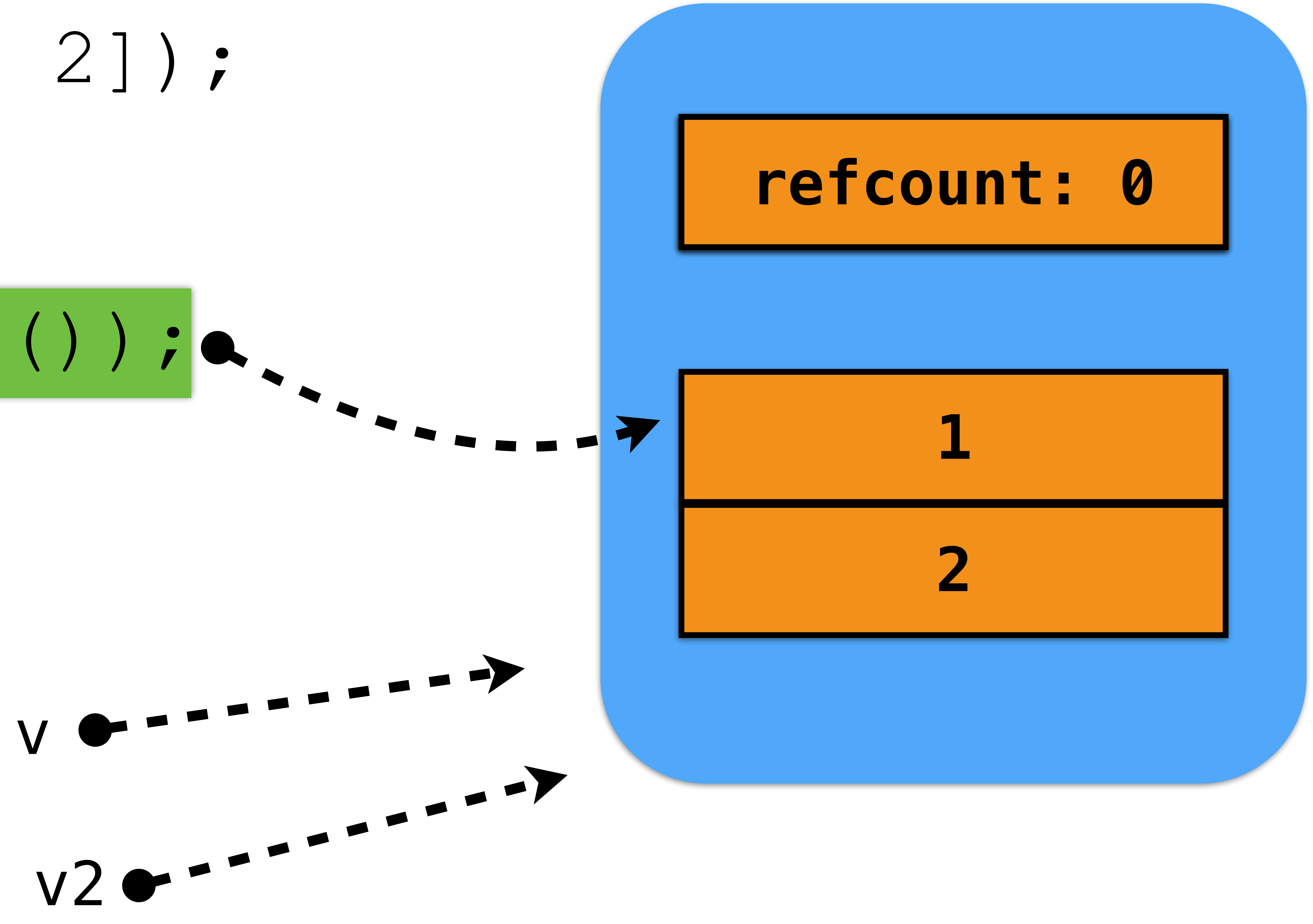
std::thread

```
let v = Rc::new(vec![1, 2, 3]);
let v2 = v.clone();
thread::spawn(move || {
    println!("{}", v.len());
});
another_function(v2.clone());
```

error: Rc<T> can't be sent across threads

std::sync::Arc

```
let v = Arc::new(vec![1, 2]);
let v2 = v.clone();
thread::spawn(move || {
    println!("{}", v.len());
});
another_function(&v2);
```



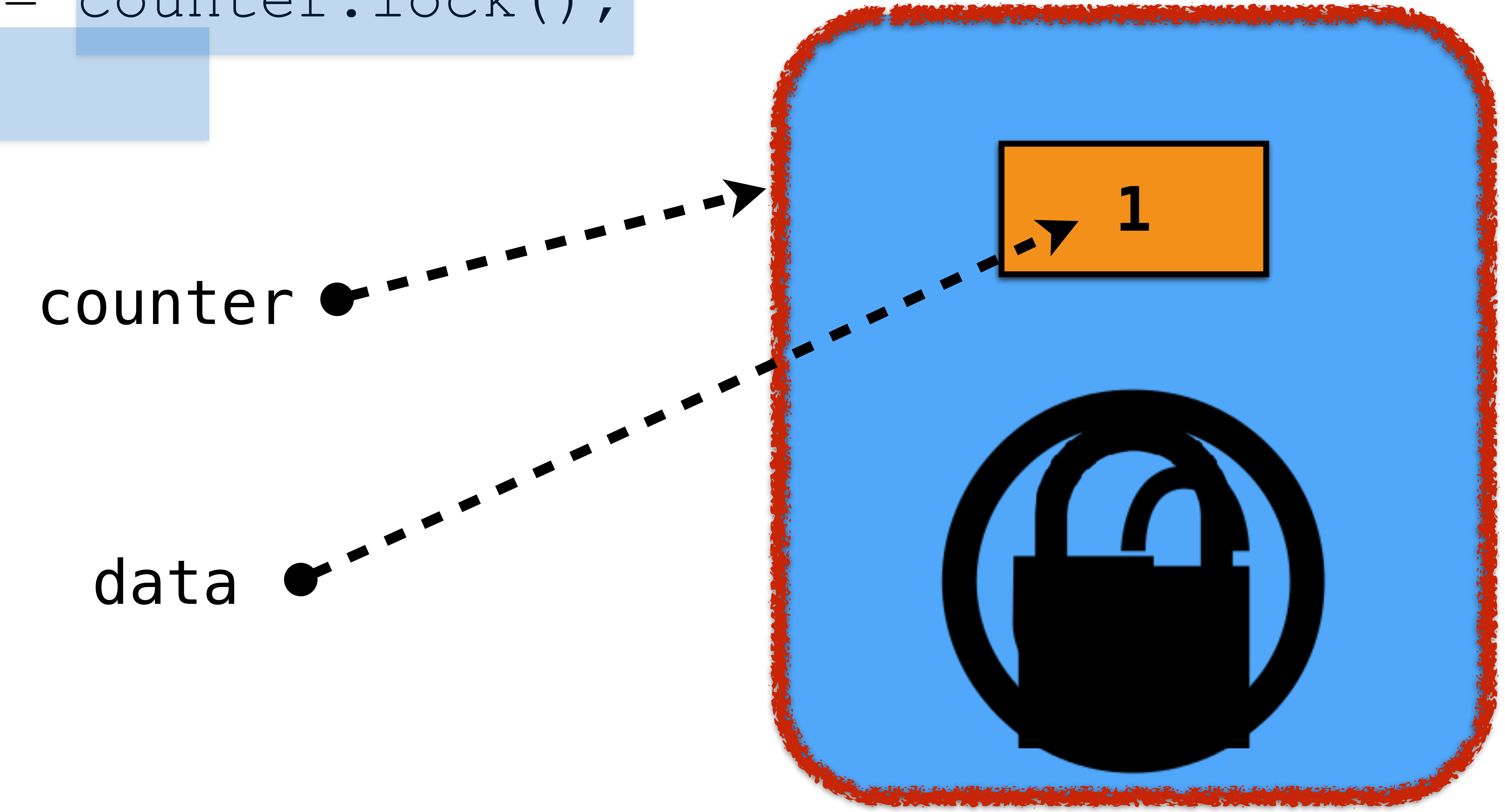
std::sync::Arc

```
let v = Arc::new(vec![1, 2]);
let v2 = v.clone();
thread::spawn(move || {
    v.push(3);
});
another_function(&v2);
```

error: cannot mutate through shared reference

std::sync::Mutex

```
fn sync_inc(counter: &Mutex<i32>) {
  let mut data: Guard<i32> = counter.lock();
  *data += 1;
}
```



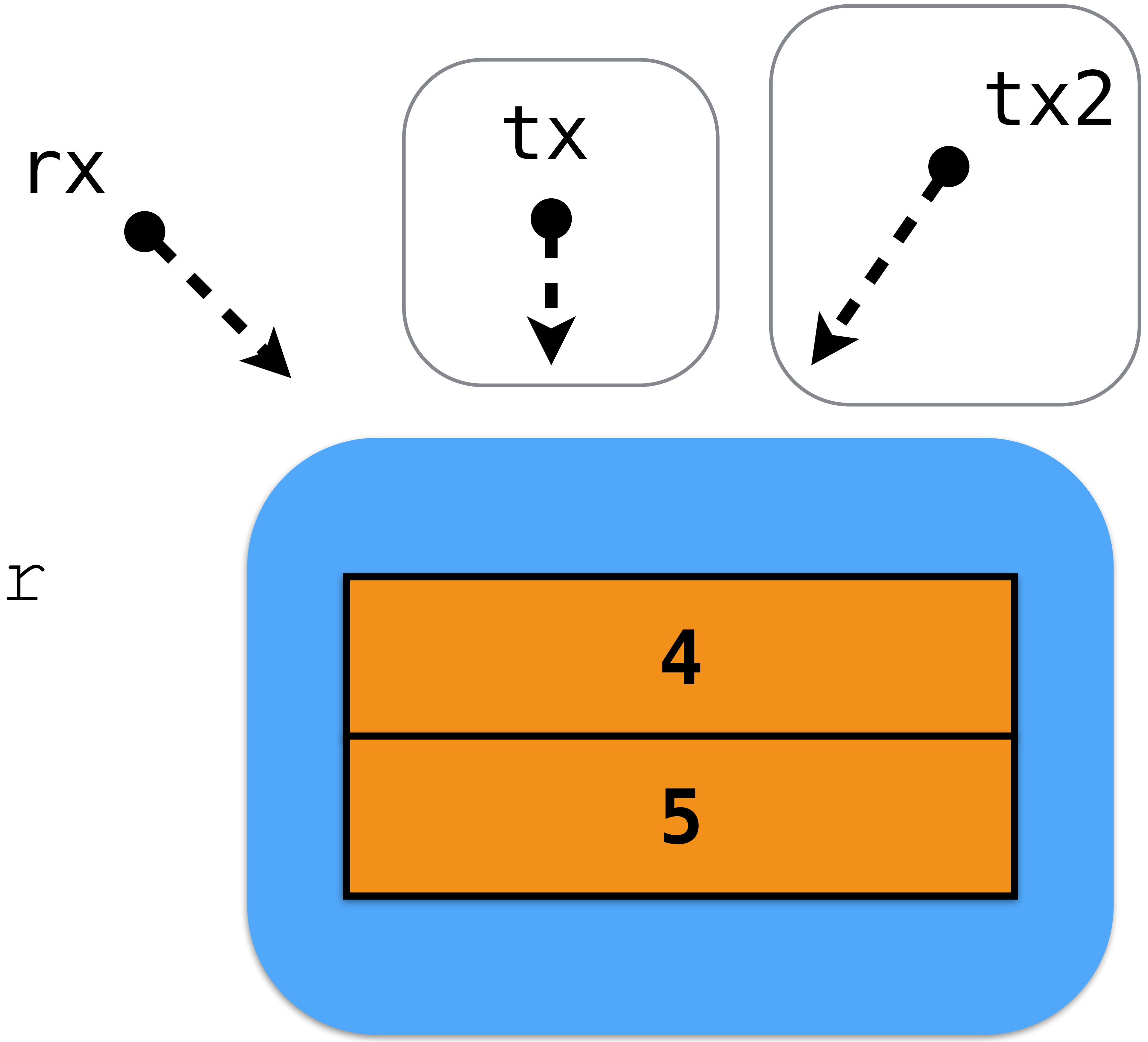
std::sync::atomic::*

```
let number = AtomicUsize::new(10);
let prev = number.fetch_add(1, SeqCst);
assert_eq!(prev, 10);
let prev = number.swap(2, SeqCst);
assert_eq!(prev, 11);
assert_eq!(number.load(SeqCst), 2);
```

std::sync::mpsc

```
let (tx, rx) = mpsc::channel();
let tx2 = tx.clone();
thread::spawn(move || tx.send(5));
thread::spawn(move || tx2.send(4));

// Prints 4 and 5 in an unspecified order
println!("{:?}", rx.recv());
println!("{:?}", rx.recv());
```



rayon

```
fn sum_of_squares(input: &[i32]) -> i32 {
    input.iter()
         .map(|&i| i * i)
         .sum()
}
```

rayon

```
use rayon::prelude::*;

fn sum_of_squares(input: &[i32]) -> i32 {
    input.par_iter()
         .map(|&i| i * i)
         .sum()
}
```

rayon

```
use rayon::prelude::*;

fn sum_of_squares(input: &[i32]) -> i32 {
    let mut cnt = 0;
    input.par_iter()
        .map(|&i| {
            cnt += 1;
            i * i
        })
        .sum()
}
```

error: `cnt` cannot be shared concurrently

crossbeam

- Epoch-based memory reclamation
- Easy translation of algorithms that require GC
- Work stealing deque
- MPMC queues

100% Safe

- Everything you just saw is foolproof
- No segfaults
- No data races
- No double frees…

Concurrency?

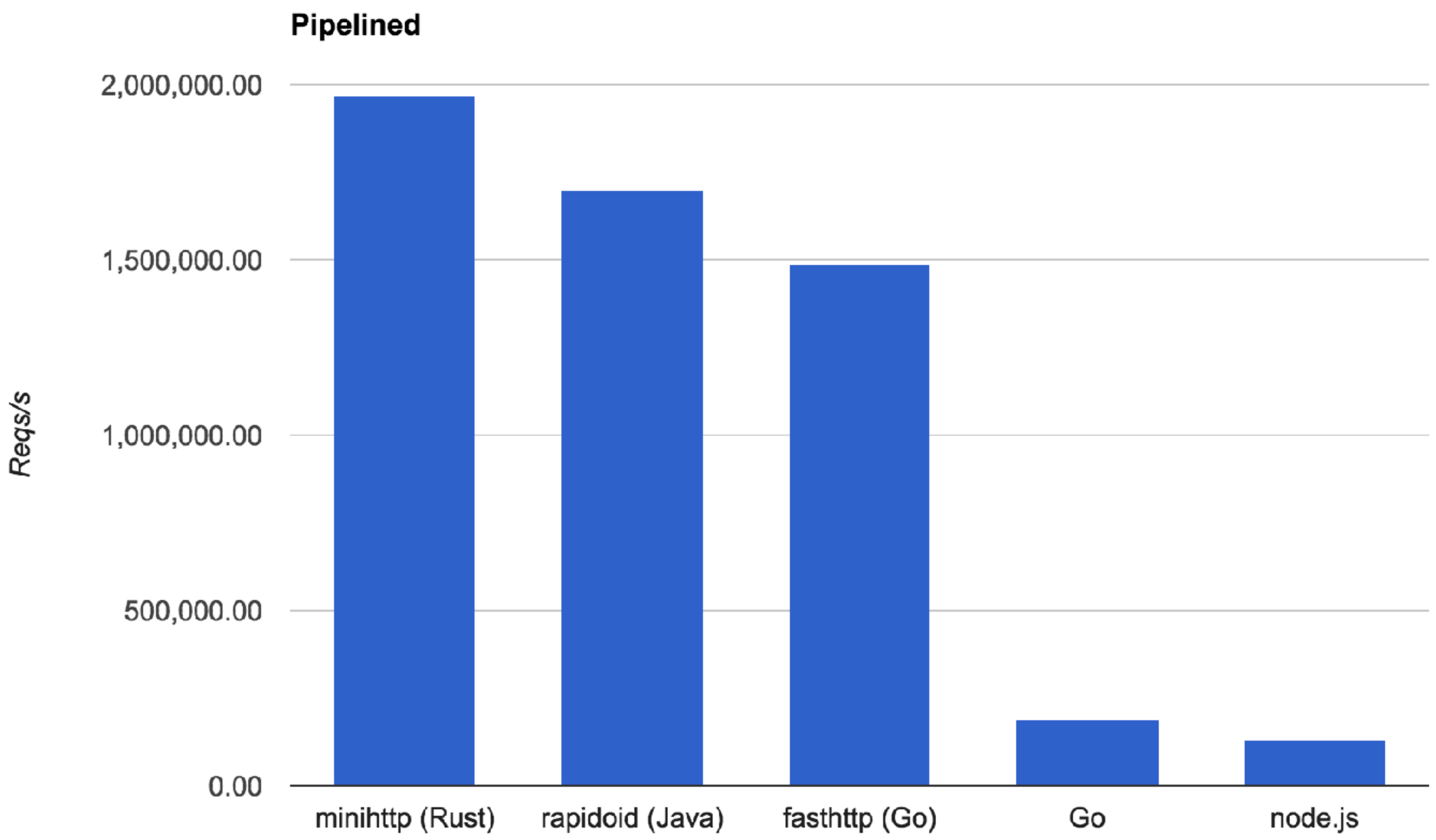
Rust?

Libraries

Futures

Concurrent I/O

- Rust-the-language fuels shared-memory parallelism
- Rust-the-ecosystem fuels other forms of concurrency
- **Futures** are the core foundation for async I/O in Rust


Pipelined
Reqs/s
2,000,000.00
1,500,000.00
1,000,000.00
500,000.00
0.00
minihttp (Rust)
rapidoid (Java)
fasthttp (Go)
Go
node.js

What is async I/O?

```
// blocking
let amt = socket.read(buffer);
assert_eq!(amt, 4);

// async
let amt = socket.read(buffer);
assert_eq!(amt, EWOULDBLOCK);
```

What is async I/O?

- No I/O ever blocks
- Application receives bulk lists of events
- User responsible for dispatching events

Async I/O can be hard

User: I’d like the contents of
https://www.rust-lang.org, please!

Kernel: file descriptor 5 is ready for reading

Futures

- Sentinel for a value “being computed”
- Internally captures state to produce a value
- Allows composing different kinds of computations

Futures and async I/O

User: I'd like the contents of
https://www.rust-lang.org, please!

Library: OK, here's a `Future<Vec<u8>>`

Futures without I/O

```
let result = pool.spawn(|| {
    fibonacci(100)
});

println!("calculating…");
get_some_coffee();
run_an_errand();

let result = result.wait();
println!("fib(100) = {}", result);
```

Rust’s solution: Tokio

cross-platform async I/O

Tokio is the fusion of `mio` *and* `futures`

you can build futures with a TCP socket

Rust’s solution: Tokio

- Futures powered by TCP/UDP/…
- Works on all major platforms
- Implementation of an *event loop*

Futures today

- Traits for one value (Future), many values (Stream), or pushing values (Sink)
- “oneshot channels” - computing an in-memory value asynchronously
- “mpsc channels” - like `std::sync::mpsc`
- Integration with crates such as rayon

Futures today

```
#[async]
fn fetch(client: hyper::Client, url: String)
    -> Result<Vec<u8>, hyper::Error>
{
    let response = await!(client.get(&url));
    if !response.status().is_ok() {
        return Err(…)
    }
    let mut body = Vec::new();
    #[async]
    for chunk in response.body() {
        body.extend(chunk);
    }
    Ok(body)
}
```

Tokio today

- Crates such as tokio-core, tokio-proto, tokio-service
- Considering a `tokio` crate (tokio-rs/tokio-rfcs#2)
- Supports TCP, UDP, Unix sockets, Named pipes, processes, signals, HTTP, HTTP/2 (coming soon!), websockets, …
- Deployed to production in a number of companies

Implementing Futures

```
struct Future<T> {
    // ...
}
```

What if I have a more efficient implementation?

```
trait Future {
    type Item;

    // ...
}
```

```
trait Future {
    type Item;

    // ...
}
```

“A future is a sentinel for something being computed”

```
trait Future {
    type Item;

    fn schedule<F>(self, F)
        where F: FnOnce(T);
}
```

Can’t do virtual dispatch!

```rust
fn compute(&self, key: u32)
    -> impl Future<Item=u32>
{
    if let Some(v) = self.cache.get(&key) {
        future::ok(v)
    } else {
        self.compute_slow(key)
    }
}
```

Different types!

```
fn compute(&self, key: u32)
    -> impl Future<Item=u32>
{
    if let Some(v) = self.cache.get(&key) {
        future::ok(v)
    } else {
        self.compute_slow(key)
    }
}
```

```
fn compute(&self, key: u32)
    -> impl Future<Item=u32>
{
    if let Some(v) = self.cache.get(&key) {
        Either::A(future::ok(v))
    } else {
        Either::B(self.compute_slow(key))
    }
}
```

Assuming:

```
impl<A, B> Future for Either<A, B>
    where A: Future, B: Future
```

```
fn compute(&self, key: u32)
    -> impl Future<Item=u32>
{
    if let Some(v) = self.cache.get(&key) {
        Either::A(future::ok(v))
    } else if ... {
        Either::B(self.compute_slow(key))
    } else {
        // ...
    }
}
```

Either::AA ?

```
fn compute(&self, key: u32)
    -> Box<Future<Item=u32>>
{
    if let Some(v) = self.cache.get(&key) {
        Box::new(future::ok(v))
    } else if ... {
        Box::new(self.compute_slow(key))
    } else {
        // ...
    }
}
```

Virtual dispatch!

```
trait Future {
    type Item;

    fn schedule<F>(self, F)
        where F: FnOnce(T);
}
```

Can’t do virtual dispatch!

```
trait Future {
    type Item;

    fn schedule(&mut self,
                f: Box<FnOnce(T)>);
}
```

```
trait Future {
    type Item;

    fn schedule(&mut self,
                f: Box<FnOnce(T)>);
}
```

Servers as futures

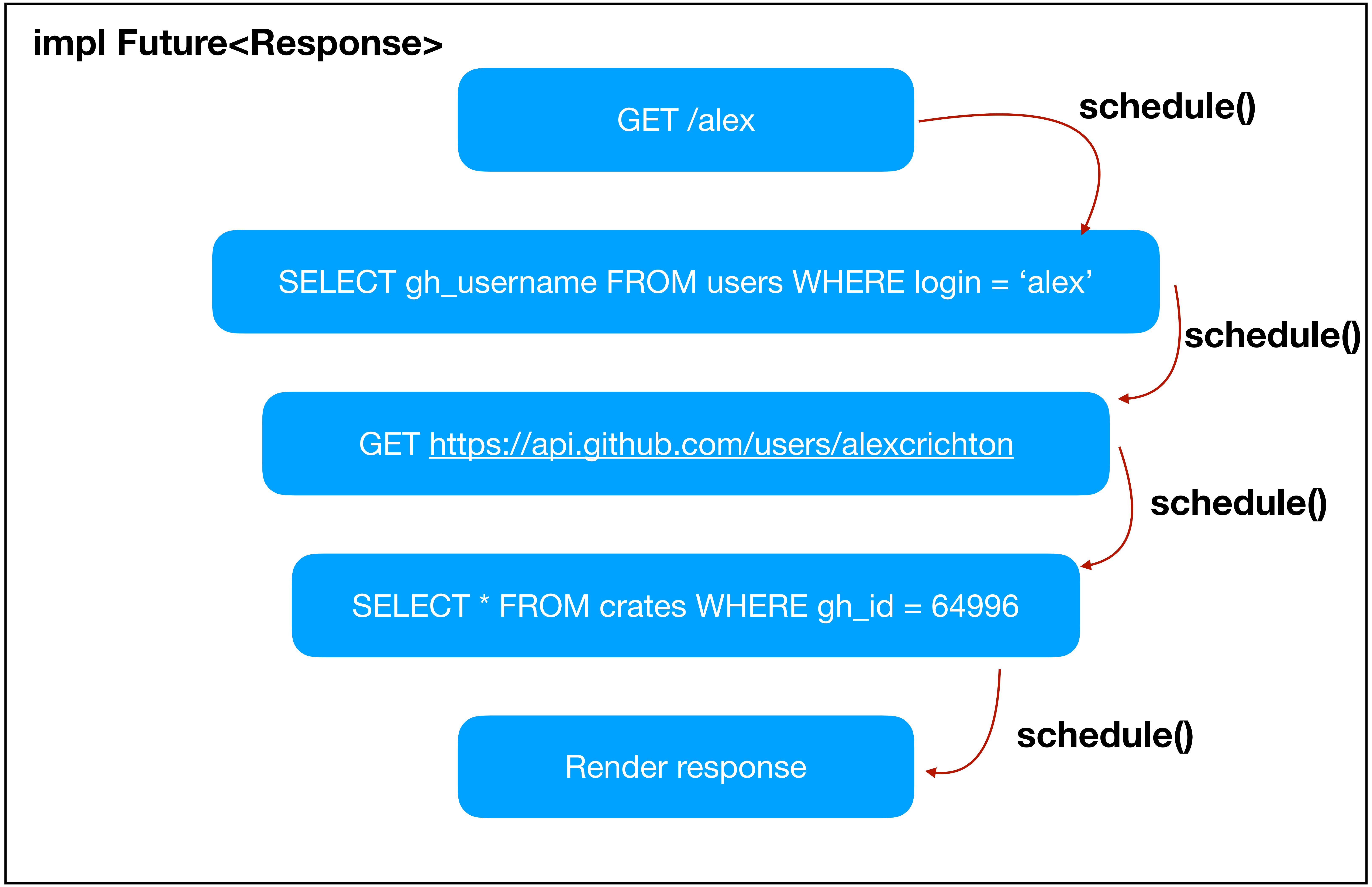
```
fn process(req: Request)
    -> impl Future<Response>
{
    // ...
}
```

Servers as futures

- Receive a request
- Load user from the database
- Query user's GitHub username
- Look up username in database
- Render a response

Futures

Servers as futures



Servers as futures

- Server responses may want to be entirely a future
- Futures are frequently composed of many futures
- Futures internally make many state transitions
- **Each state transition is allocating a callback**

Futures and threading

```
trait Future {
    type Item;

    fn schedule(&mut self,
                f: Box<FnOnce(T)>);
}
```

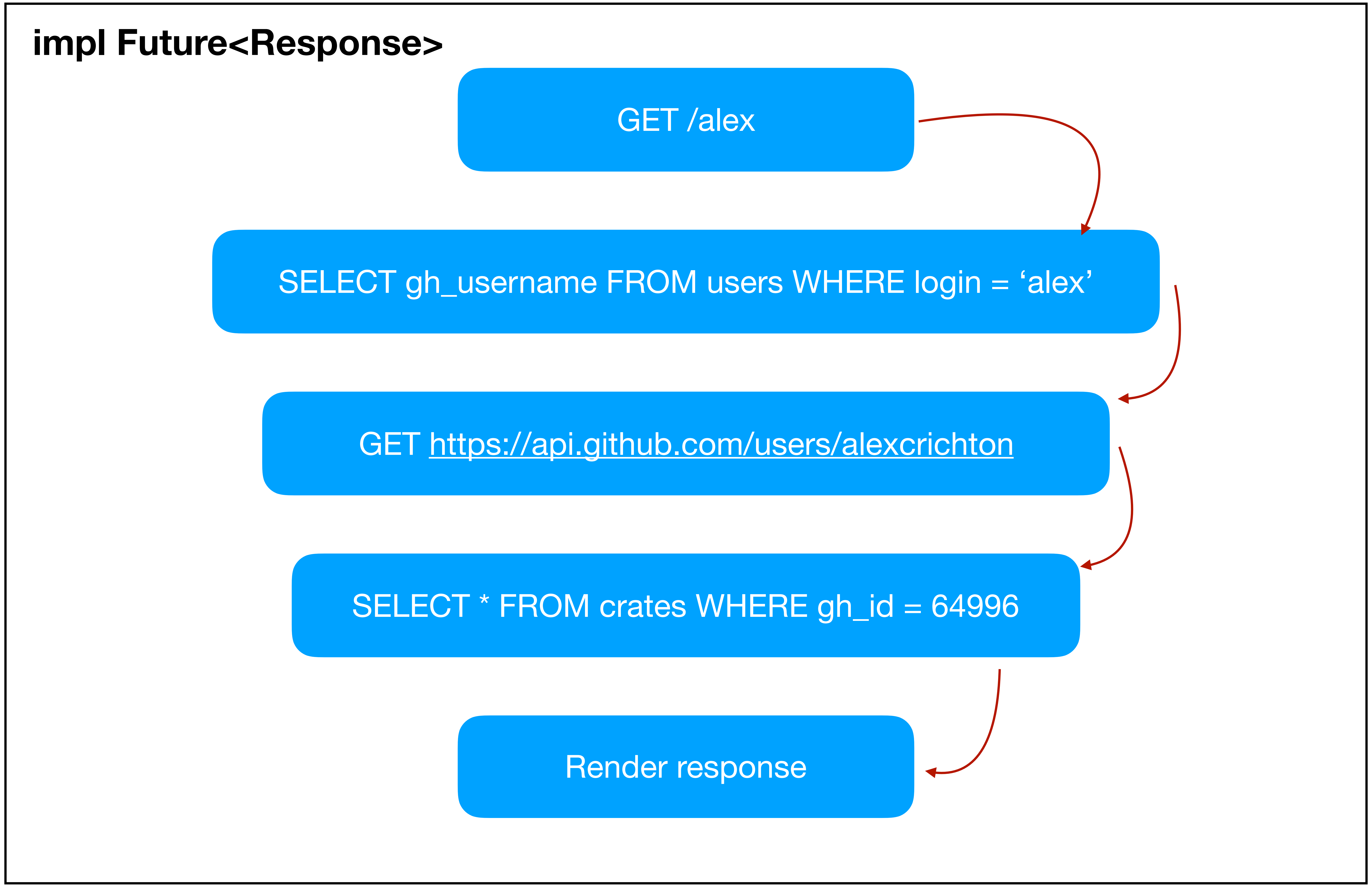
Send? Sync?

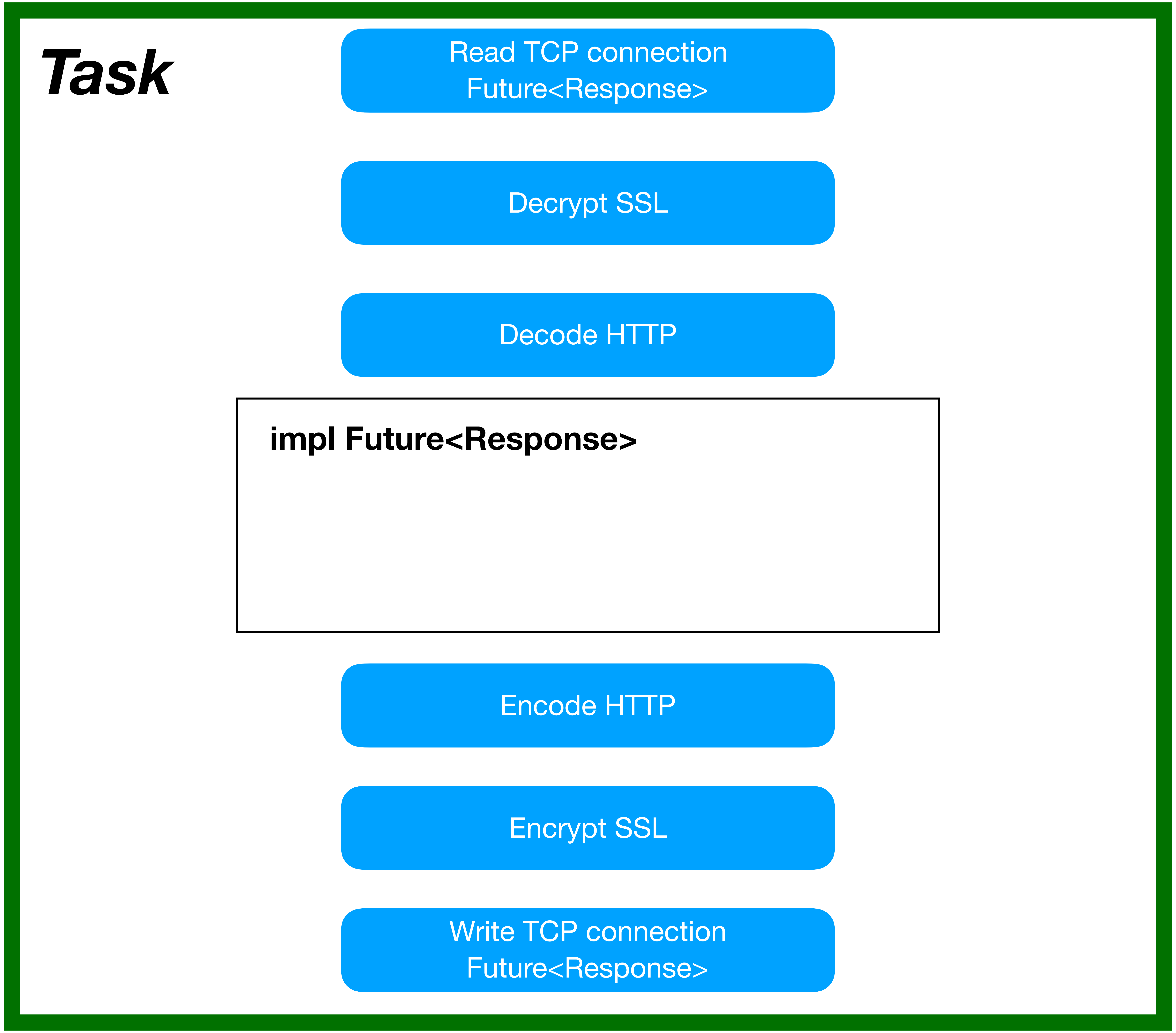
Drawbacks of callbacks

- Expensive at runtime, forced allocation per state transition
- Gymnastics with threading
  - synchronization may be required where not necessary
  - may require multiple “threadsafe future” and “non-threadsafe future” traits

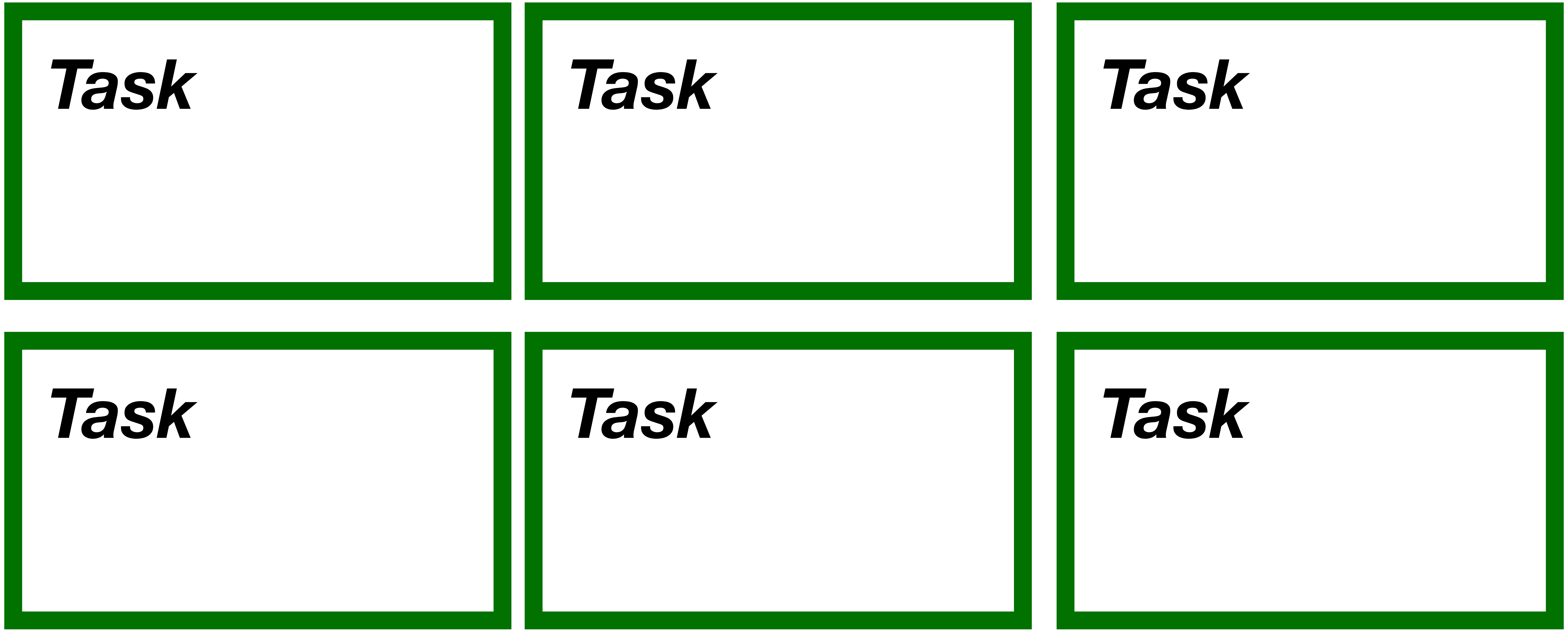
Constraints so far

- Futures must be a trait
- The Future trait must allow virtual dispatch
- State transitions need to be cheap
- Threadsafe futures shouldn’t require gymnastics


impl Future<Response>
GET /alex
SELECT gh_username FROM users WHERE login = ‘alex’
GET https://api.github.com/users/alexcrichton
SELECT * FROM crates WHERE gh_id = 64996
Render response


Task
Read TCP connection
Future<Response>
Decrypt SSL
Decode HTTP
impl Future<Response>
Encode HTTP
Encrypt SSL
Write TCP connection
Future<Response>

One Server



Futures and Tasks

- A *Task* is composed of many futures
- All futures internal to a *task* track the same task
- Tasks are a unit of concurrency
- Very similar to green/lightweight threads!

```
trait Future {
    type Item;

    // …
}
```

“A future is a sentinel for something being computed”

```
trait Future {
    type Item;

    fn poll(&mut self)
        -> Option<Self::Item>;
}
```

Poll-based futures

- A return value of `None` means “not ready”
- Returning `Some` resolves a future
- If you see `None`, when do you later call `poll`?

Poll-based futures

- **Futures are owned by one task**
- If a future isn't ready, the **task** needs to know when to try again
- A return value of `None` **automatically** means "I'll notify you later"

```
struct Timeout {
    // ...
}
```

```rust
struct Timeout {
    when: Instant,
    timer: Timer,
}
```

When our timeout fires

Library support for this timeout

```rust
// provided by another library
impl Timer {
    fn defer(&self,
             at: Instant,
             f: impl FnOnce()) {
        // ...
    }
}
```

```
impl Future for Timeout {
    type Item = ();

    fn poll(&mut self) -> Option<()> {
        // ...
    }
}
```

```
impl Future for Timeout {
    type Item = ();

    fn poll(&mut self) -> Option<()> {
        if self.when < Instant::now() {
            return Some(())
        }
        // ...
    }
}
```

```rust
use futures::task;

impl Future for Timeout {
    type Item = ();

    fn poll(&mut self) -> Option<()> {
        if self.when < Instant::now() {
            return Some(())
        }
        let task = task::current();
        self.timer.defer(self.when, || {
            task.notify();
        });
        None // not ready yet
    }
}
```

Constraints

- ✓ `trait Future`
- ✓ Virtual dispatch
- Cheap
- Threadsafe

```
trait Future {
    type Item;

    fn poll(&mut self)
        -> Option<Self::Item>;
}
```

Constraints

- ✓ `trait Future`
- ✓ Virtual dispatch
- ✓ Cheap
- Threadsafe

```
mod task {
    fn current() -> Task {
        // just an Arc bump
    }

    impl Task {
        fn notify(&self) {
            // just a queue
        }
    }
}
```

Constraints

- `trait Future`
- Virtual dispatch
- Cheap
- Threadsafe

```
fn assert<T: Send + Sync>() {}

assert::<Task>();
```

I/O and Polling futures

- In Tokio **everything** is a future
- Futures may eventually need to wait for I/O
- Tokio’s job is to route I/O notifications to tasks

I/O and Polling futures

```
// async
let amt = socket.read(buffer);
assert_eq!(amt, EWOULDBLOCK);
```

- I/O all returns immediately
- “would block” turns into notifying a task
- “would block” translated to `NotReady`

I/O and Polling futures

```
trait AsyncRead: Read {
    // empty!
}

trait AsyncWrite: Write {
    fn shutdown(&mut self)
        -> Poll<(), io::Error>;
}
```

Marker traits for the current future’s task is scheduled to receive a notification when the object is ready again

I/O and Polling futures

```
impl<E: Read> Read for PollEvented<E>{
    fn read(&mut self, buf: &mut [u8])
        -> io::Result<usize>
    {
        if self.poll_read().is_not_ready() {
            return Err(wouldblock())
        }
        let r = self.get_mut().read(buf);
        if is_wouldblock(&r) {
            self.need_read();
        }
        return r
     }
}
```

Tokio’s Event Loop

Kernel: file descriptor 5 is ready for reading

Tokio: Ok, let me go wake up that task.

Tokio’s Event Loop

- Responsible for blocking the current thread
- Dispatches events received from the kernel
- Translates I/O notifications to task notifications

doc.rust-lang.org/stable/book

users.rust-lang.org

github.com/alexcrichton/futures-rs