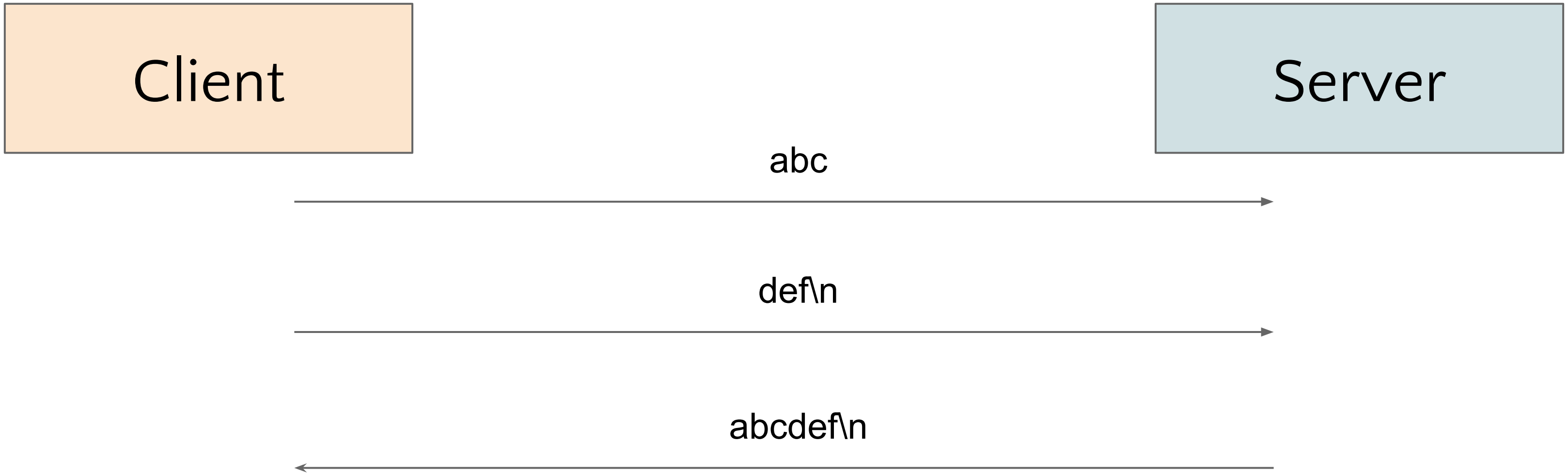
CS3211 Tutorial 9

Asynchronous Programming in Rust
Simon J – T5

Adapted From Sriram's Slides

Today's task: A line echo server



A “default” implementation: threads

```
fn main() -> std::io::Result<()> {
    let port = std::env::args()
        .nth(1)
        .map(|s| s.parse().unwrap())
        .unwrap_or(50000u16);
    let listener = TcpListener::bind(SocketAddr::from(([127, 0, 0, 1], port)))?;
    loop {
        let (socket, _) = listener.accept()?;
        thread::spawn(move || {
            eprintln!("Accepted connection");
            std::mem::drop(handle_client(socket));
            eprintln!("Connection ended");
        });
    }
```

Get port from command line args

Create a TCP listener (std lib)

Loop forever, and for each loop…

Spawn a new thread for each new connection

Each handles reading until newline, then writing back

A “default” implementation: threads

```
fn main() -> std::io::Result<()> {
    let port = std::env::args()
        .nth(1)
        .map(|s| s.parse().unwrap())
        .unwrap_or(50000u16);
    let listener = TcpListener::bind(SocketAddr::from(([127, 0, 0, 1], port)))?;
    loop {
        let (socket, _) = listener.accept()?;
        thread::spawn(move || {
            eprintln!("Accepted connection");
            std::mem::drop(handle_client(socket));
            eprintln!("Connection ended");
        });
    }
```

Why do we need the move inside the thread::spawn? [p]

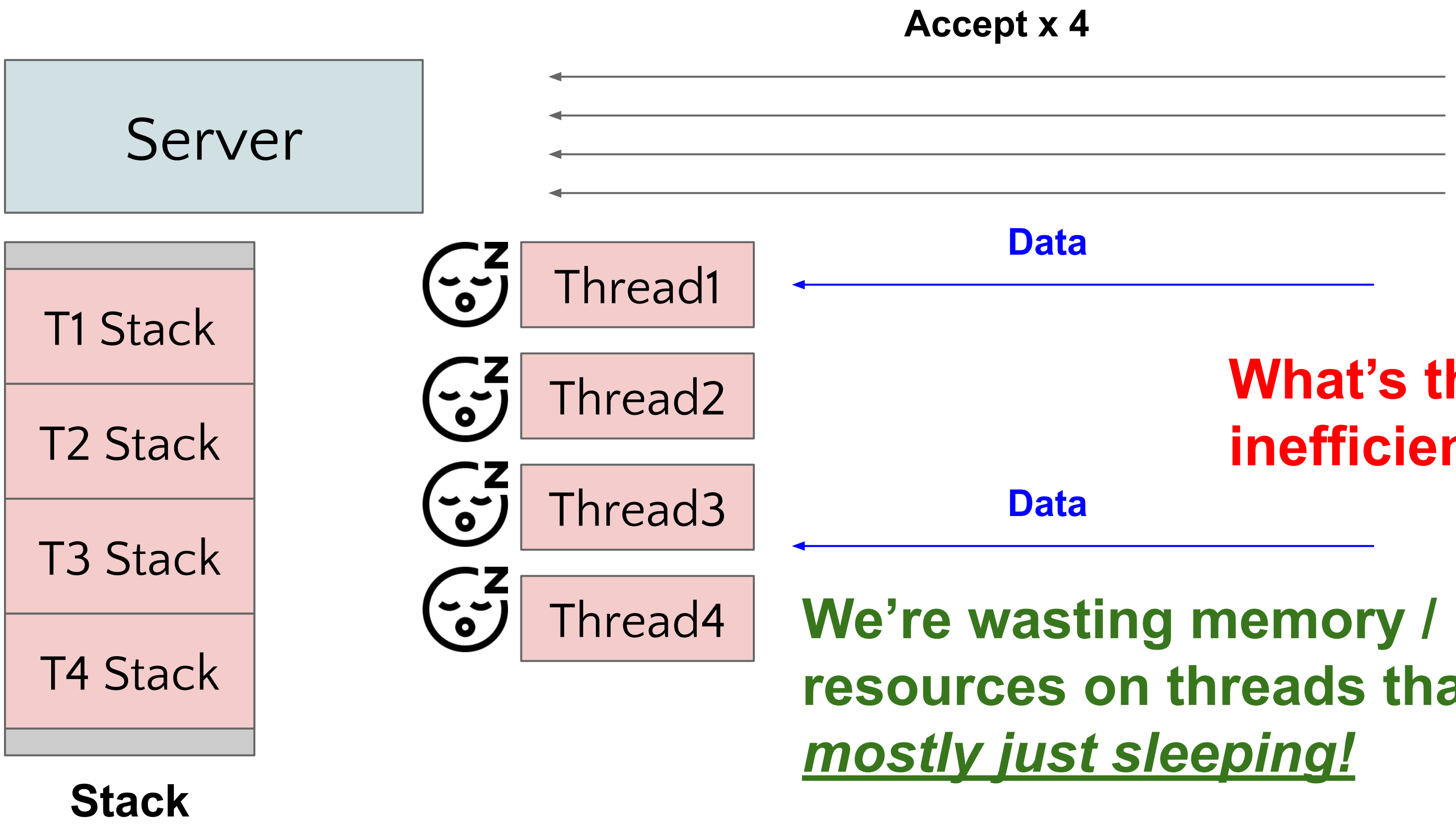
A “default” implementation: threads

```
fn main() -> std::io::Result<()> {
    let port = std::env::args()
        .nth(1)
        .map(|s| s.parse().unwrap())
        .unwrap_or(50000u16);
    let listener = TcpListener::bind(SocketAddr::from(([127, 0, 0, 1], port)))?;
    loop {
        let (socket, _) = listener.accept()?;
        thread::spawn(move || {
            eprintln!("Accepted connection");
            std::mem::drop(handle_client(socket));
            eprintln!("Connection ended");
        });
    }
```

None of the options are true: concern is that the *thread* may outlive the *socket* variable without the move.

So we let the thread take ownership of it.

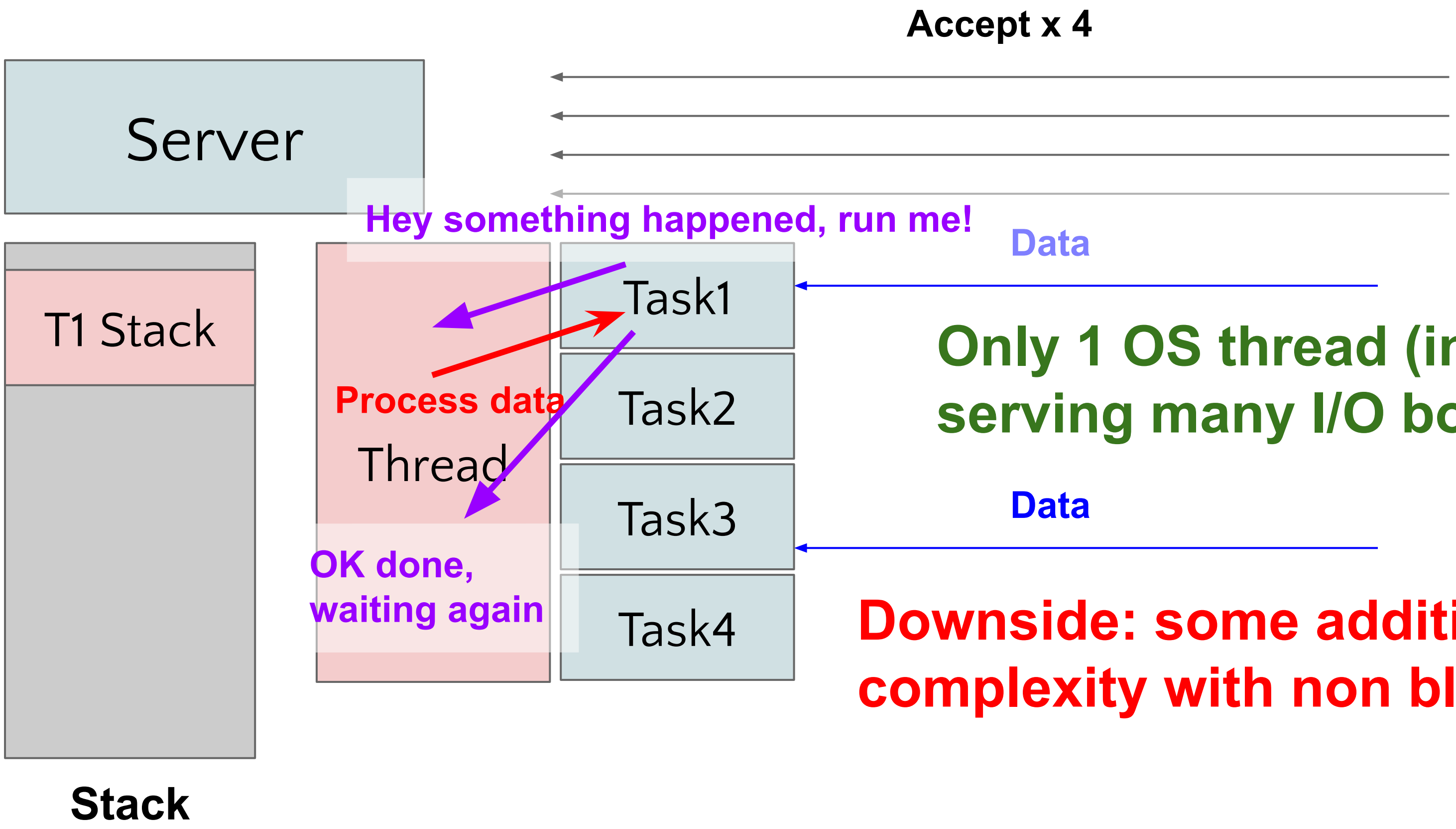
How does the thread implementation look?



What's the inefficiency here?

We're wasting memory / resources on threads that are *mostly just sleeping!*

What about an alternative



Only 1 OS thread (in this eg) but serving many I/O bound clients!

Downside: some additional complexity with non blocking IO

Threads vs Async-Await (by Jim Blandy)

Metric	Rust Threads (Kernel /OS threads)	Rust Async-Await (Tokio runtime)	Async Improvement
Creation time			
Context switch time (between tasks)			
Memory use (small task)			

https://github.com/jimblandy/context-switch
Jim Blandy is the author of O'Reilly's Programming Rust.

How do we transform threaded / blocking code to async code?

How to transform to async?

Original per-client handler

```rust
fn handle_client(stream: TcpStream) -> std::io::Result<()> {
    let mut reader = BufReader::new(stream);
    let mut buf: Vec<u8> = Vec::new();
    loop {
        let size = reader.read_until(b'\n', &mut buf)?;
        if size == 0 || buf[size - 1] != b'\n' {
            break;
        }
        reader.get_mut().write_all(&buf[..size])?;
        buf.clear();
    }
    Ok(())
}
```

Buffered reader to read fron the TCP stream efficiently

Read until \n - <u>a blocking call</u>

Write everything back to client - <u>another blocking call</u>

How to transform to async?

With Tokio

```rust
use tokio::io::{AsyncBufReadExt, AsyncWriteExt, BufReader};
use tokio::net::{TcpListener, TcpStream};
```

async fn now

```rust
async fn handle_client(stream: TcpStream) -> std::io::Result<()> {
    let mut reader = BufReader::new(stream);
    let mut buf: Vec<u8> = Vec::new();
    loop {
        let size = reader.read_until(b'\n', &mut buf).await?;
        if size == 0 || buf[size - 1] != b'\n' {
            break;
        }
        reader.get_mut().write_all(&buf[..size]).await?;
        buf.clear();
    }
    Ok(())
}
```

tokio version of BufReader

We can call await on it as it returns a Future

Similarly, write method returns a Future

How to transform to async?

With Tokio (main function)

What's wrong with this code? [p]

#[tokio::main] macro - run this function on the tokio runtime

```
#[tokio::main]
async fn main() -> std::io::Result<()> {
    let port = std::env::args()
        .nth(1)
        .map(|s| s.parse().unwrap())
        .unwrap_or(50000u16);
    let listener = TcpListener::bind(SocketAddr::from(([127, 0, 0, 1], port))).await?;
    loop {
        let (socket, _) = listener.accept().await?;
        handle_client(socket).await?;
    }
}
```

Tokio's own TcpListener returns a <u>Future</u> that we can await on for binding

Similarly, await for someone to accept the connection, then await the handling of the data from the client

How to transform to async?

With Tokio

```
#[tokio::main]
async fn main() -> std::io::Result<()> {
    let port = std::env::args()
        .nth(1)
        .map(|s| s.parse().unwrap())
        .unwrap_or(50000u16);
    let listener = TcpListener::bind(SocketAddr::from(([127, 0, 0, 1], port))).await?;
    loop {
        let (socket, _) = listener.accept().await?;
        handle_client(socket).await?;
    }
}
```

There is only 1 task, and it gets blocked for each new client. Even worse than the std::thread code!

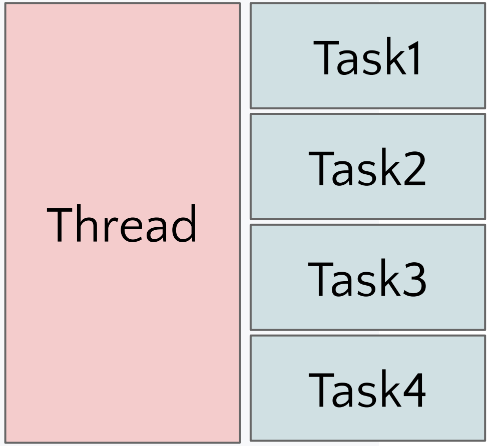
How to transform to async?

With Tokio

Now we spawn a new tokio task for each new client! They can be scheduled to run by the tokio runtime.

Note the async closure here.

```rust
#[tokio::main]
async fn main() -> std::io::Result<()> {
    let port = std::env::args()
        .nth(1)
        .map(|s| s.parse().unwrap())
        .unwrap_or(50000u16);
    let listener = TcpListener::bind(SocketAddr::from(([127, 0, 0, 1], port))).await?;
    loop {
        let (socket, _) = listener.accept().await?;
        tokio::spawn(async move {
            eprintln!("Accepted connection");
            std::mem::drop(handle_client(socket).await);
            eprintln!("Connection ended");
        });
    }
}
```



Why this question?

- Async code (in simple cases) is **not too different** from normal code!

- **Compiler handles** much **heavy lifting** on our behalf

- Massive **performance gain** if we do this for the right problems!

```
#[tokio::main]
async fn main() -> std::io::Result<()> {
    let port = std::env::args()
        .nth(1)
        .map(|s| s.parse().unwrap())
        .unwrap_or(50000u16);
    let listener = TcpListener::bind(SocketAddr::from(([127, 0, 0, 1], port))).await?;
    loop {
        let (socket, _) = listener.accept().await?;
        tokio::spawn(async move {
            eprintln!("Accepted connection");
            std::mem::drop(handle_client(socket).await);
            eprintln!("Connection ended");
        });
    }
}
```

Synchronization between async tasks (H2O problem)

H2O Problem

```cpp
struct WaterFactory3 {
  std::counting_semaphore<> oxygenSem;
  std::counting_semaphore<> hydrogenSem;
  std::barrier<> barrier;

  WaterFactory3() : oxygenSem{1}, hydrogenSem{2}, barrier{3} {}

  void oxygen(void (*bond)()) {
    oxygenSem.acquire();  // Lets at most one oxygen through
    barrier.arrive_and_wait();
    bond();
    oxygenSem.release();  // We are done, let the next oxygen in
  }

  void hydrogen(void (*bond)()) {
    hydrogenSem.acquire();  // Lets at most two hydrogen through
    barrier.arrive_and_wait();
    bond();
    hydrogenSem.release();  // We are done, let the next hydrogen in
  }
};
```

Rusty H2O: Step 1: Converting the struct

Note – we're not following the tutorial solution, trying to do a direct mapping to C++ instead (also more idiomatic) [credits: Walter]

```cpp
struct WaterFactory3 {
  std::counting_semaphore<> oxygenSem;
  std::counting_semaphore<> hydrogenSem;
  std::barrier<> barrier;
```

```rust
use tokio::sync::{Barrier, Semaphore};
struct WaterFactory {
    o_sem: Semaphore,
    h_sem: Semaphore,
    barrier: Barrier,
}
```

Rusty H2O: Step 2: Initialization

Initialization is also pretty similar!

```
WaterFactory3() : oxygenSem{1}, hydrogenSem{2}, barrier{3} {}
```

```
impl WaterFactory {
// WaterFactory is fine too, but Self allows changing names
    fn new() -> Self {
        Self {
                o sem: Semaphore::new(1),
                h sem: Semaphore::new(2),
                barrier: Barrier::new(3)
            }
    }
}
```

Rusty H2O: Step 3: Implementation

Implementation is relatively straightforward...

```
void oxygen(void (*bond)()) {
   oxygenSem.acquire();  // Lets at most one oxygen through
   barrier.arrive_and_wait();
   bond();
   oxygenSem.release();  // We are done, let the next oxygen in
}

void hydrogen(void (*bond)()) {
   hydrogenSem.acquire();  // Lets at most two hydrogen through
   barrier.arrive_and_wait();
   bond();
   hydrogenSem.release();  // We are done, let the next hydrogen in
}
```

Rusty H2O: Step 3: Implementation

Implementation is relatively straightforward…

```
fn oxygen(&self, bond: impl FnOnce()) {
    let _ = self.o_sem.acquire(); // RAII
    self.barrier.wait();
    bond();
}

fn hydrogen(&self, bond: impl FnOnce()) {
    let _ = self.h_sem.acquire(); // RAII
    self.barrier.wait();
    bond();
}
```

Rusty H2O: Step 4: Async!

Now we make it async! Great!

```
async fn oxygen(&self, bond: impl FnOnce()) {
    let _ = self.o_sem.acquire(); // RAII
    self.barrier.wait();
    bond();
}

async fn hydrogen(&self, bond: impl FnOnce()) {
    let _ = self.h_sem.acquire(); // RAII
    self.barrier.wait();
    bond();
}
```

Rusty H2O: Step 4

Async Runner!

Many pointers to WaterFactory now (each hydrogen/oxygen function)

Clone the shared pointer to waterfactory (increase ref count)

Pass bond function as argument

Join all the task handles to force them to execute concurrently

```rust
use futures::future::join_all;
use std::sync::Arc;
use tokio::sync::{Barrier, Semaphore};

fn bond(s: &str) {
    println!("bond {s}!");
}

#[tokio::main]
async fn main() {
    let n = 10;
    let f = Arc::new(WaterFactory::new());

    let hs = (0..n * 2).map(|i| {
        let f = f.clone();
        tokio::spawn((|| async move {
            f.hydrogen(|| bond(&format!("h{i}"))).await;
        })())
    });
    let os = (0..n).map(|i| {
        let f = f.clone();
        tokio::spawn((|| async move {
            f.oxygen(|| bond(&format!("o{i}"))).await;
        })())
    });

    join_all(Iterator::chain(hs, os)).await;
}
```

Rusty H2O: Step 5: Fixing Async…

`cargo run --bin task2-struct-1`

There are issues with our implementation…

```
async fn oxygen(&self, bond: impl FnOnce()) {
    let _ = self.o_sem.acquire(); // RAII
    self.barrier.wait();
    bond();
}

async fn hydrogen(&self, bond: impl FnOnce()) {
    let _ = self.h_sem.acquire(); // RAII
    self.barrier.wait();
    bond();
}
```

Rusty H2O: Step 5: Fixing Async…

`cargo run --bin task2-struct-2`

There are issues with our implementation… no await! More?

```
async fn oxygen(&self, bond: impl FnOnce()) {
    let _ = self.o_sem.acquire().await.unwrap(); // RAII
    self.barrier.wait().await;
    bond();
}

async fn hydrogen(&self, bond: impl FnOnce()) {
    let _ = self.h_sem.acquire().await.unwrap(); // RAII
    self.barrier.wait().await;
    bond();
}
```

Rusty H2O: Step 5: Fixing Async…

`cargo run --bin task2-struct`

There are issues with our implementation… dropping the permit prematurely!

(we release() the semaphore automatically when the permit is dropped, previously dropped immediately)

https://github.com/rust-lang/rust/issues/10488

```
async fn oxygen(&self, bond: impl FnOnce()) {
    let _permit = self.o_sem.acquire().await.unwrap(); // RAII
    self.barrier.wait().await;
    bond();
}

async fn hydrogen(&self, bond: impl FnOnce()) {
    let _permit = self.h_sem.acquire().await.unwrap(); // RAII
    self.barrier.wait().await;
    bond();
}
```

Now this is correct!

https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=ab9c1a9f853f914bda4bcda932645a18

Why this question?

- Async code (even in more complex cases) is **not too different** from normal code!

- As usual, may need to handle **shared pointer / reference counter** stuff as async **tasks still can be moved across threads!**

But… how does all this magic work?

(thanks again to Walter)

Scenario: Add To Inbox

```
async fn add_to_inbox(...) -> Result<(), Error> {
  let msg = load_message(email).await?;
  let user = get_user(id).await?;
  user.verify_has_space(&msg)?;
  user.add_to_inbox(msg).await
}
```

How to allow context switching?

How does OS std::thread context switch?

"Stackful threads"

Stores and loads (to/from memory)
- CPU registers
- Program counter (PC)
- Stack Pointer (SP)…

Eg: asm/ptrace.h

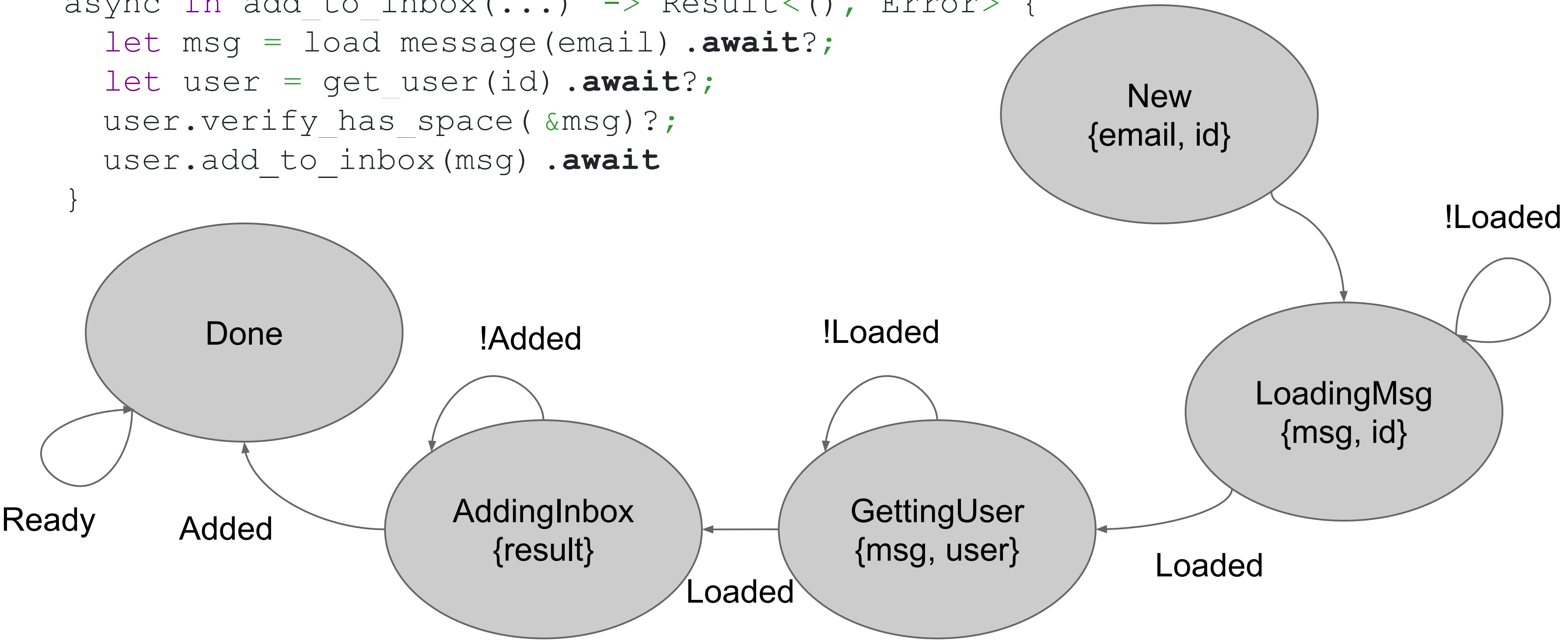
Can we do better?

```c
struct pt_regs {
/*
 * C ABI says these regs are callee-preserved. They aren't saved on kernel entr
 * unless syscall needs a complete, fully filled "struct pt_regs".
 */
	unsigned long r15;
	unsigned long r14;
	unsigned long r13;
	unsigned long r12;
	unsigned long bp;
	unsigned long bx;
/* These regs are callee-clobbered. Always saved on kernel entry. */
	unsigned long r11;
	unsigned long r10;
	unsigned long r9;
	unsigned long r8;
	unsigned long ax;
	unsigned long cx;
	unsigned long dx;
	unsigned long si;
	unsigned long di;
/*
 * On syscall entry, this is syscall#. On CPU exception, this is error code.
 * On hw interrupt, it's IRQ number:
 */
	unsigned long orig_ax;
/* Return frame for iretq */
	unsigned long ip;
	unsigned long cs;
	unsigned long flags;
	unsigned long sp;
	unsigned long ss;
/* top of stack page */
};
```

Alternative: State Machine

Store minimal state with union!

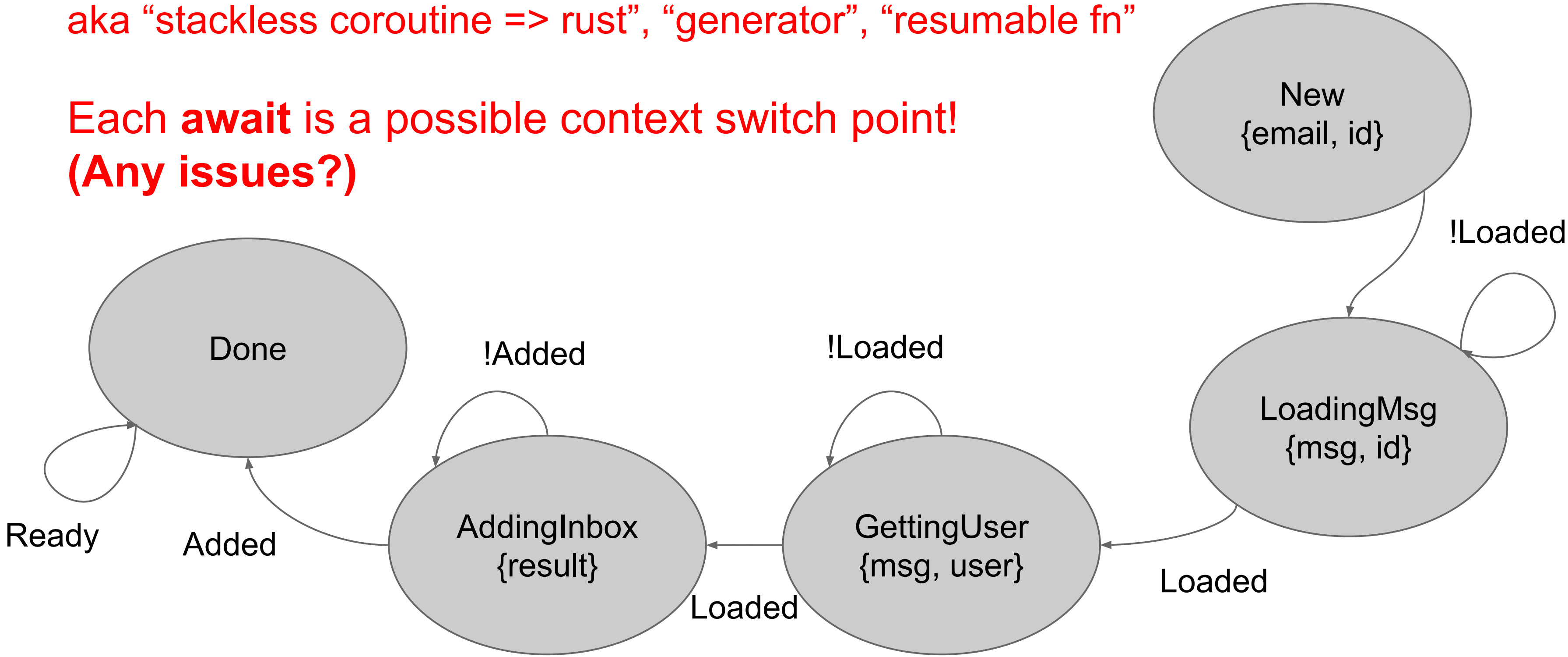
```
async fn add_to_inbox(...) -> Result<(), Error> {
  let msg = load_message(email).await?;
  let user = get_user(id).await?;
  user.verify_has_space(&msg)?;
  user.add_to_inbox(msg).await
}
```



State Machine

aka “stackless coroutine => rust”, “generator”, “resumable fn”

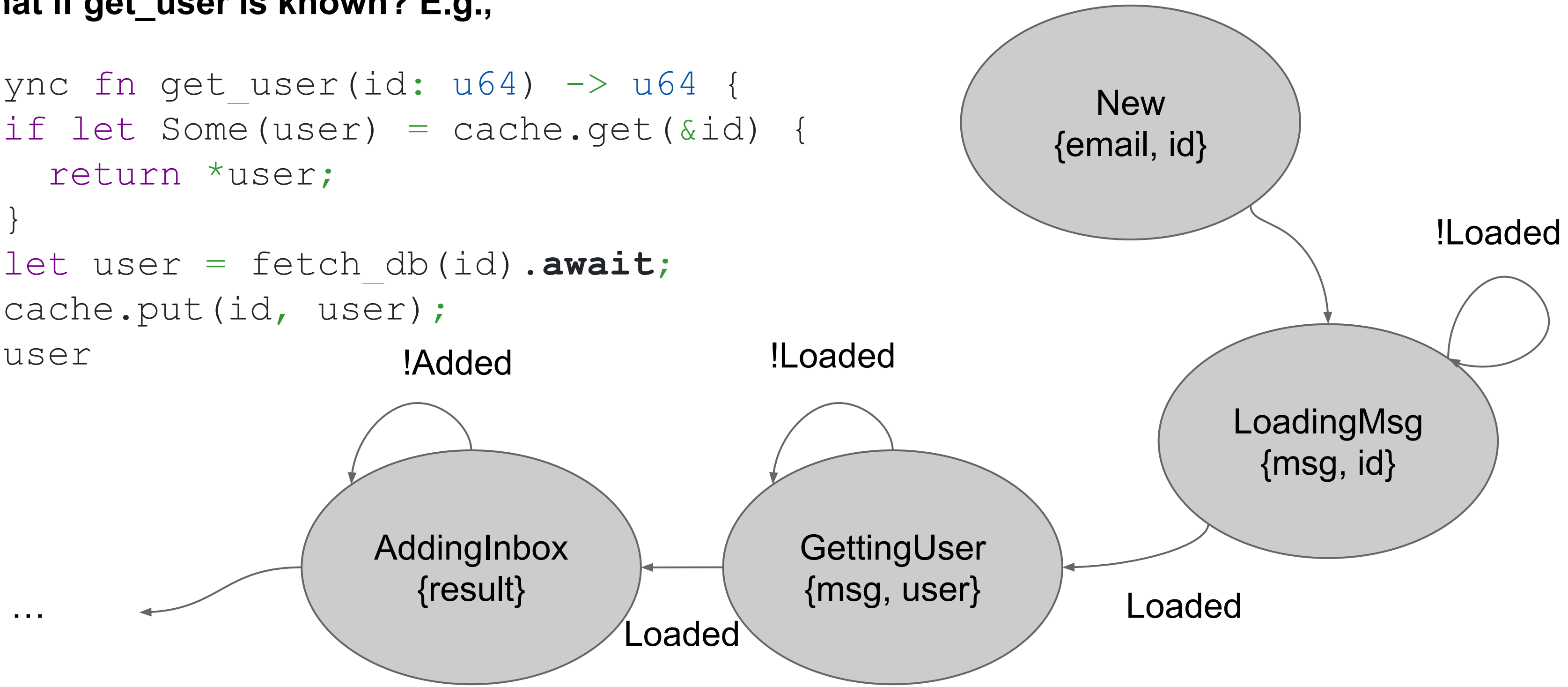
Each **await** is a possible context switch point!
(Any issues?)



Pros: Optimize across functions!

What if get_user is known? E.g.,

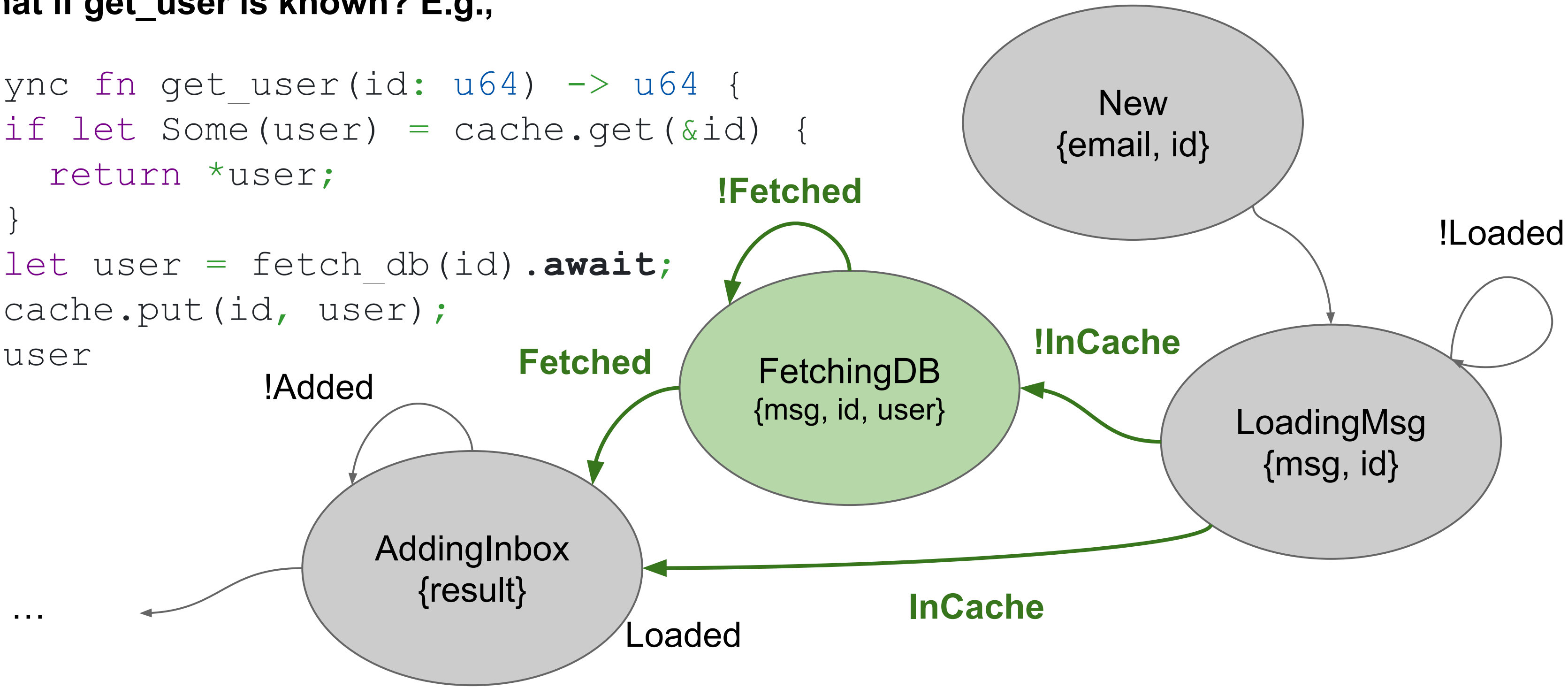
```
async fn get_user(id: u64) -> u64 {
  if let Some(user) = cache.get(&id) {
    return *user;
  }
  let user = fetch_db(id).await;
  cache.put(id, user);
  user
}
```



Pros: Optimize across functions!

What if get_user is known? E.g.,

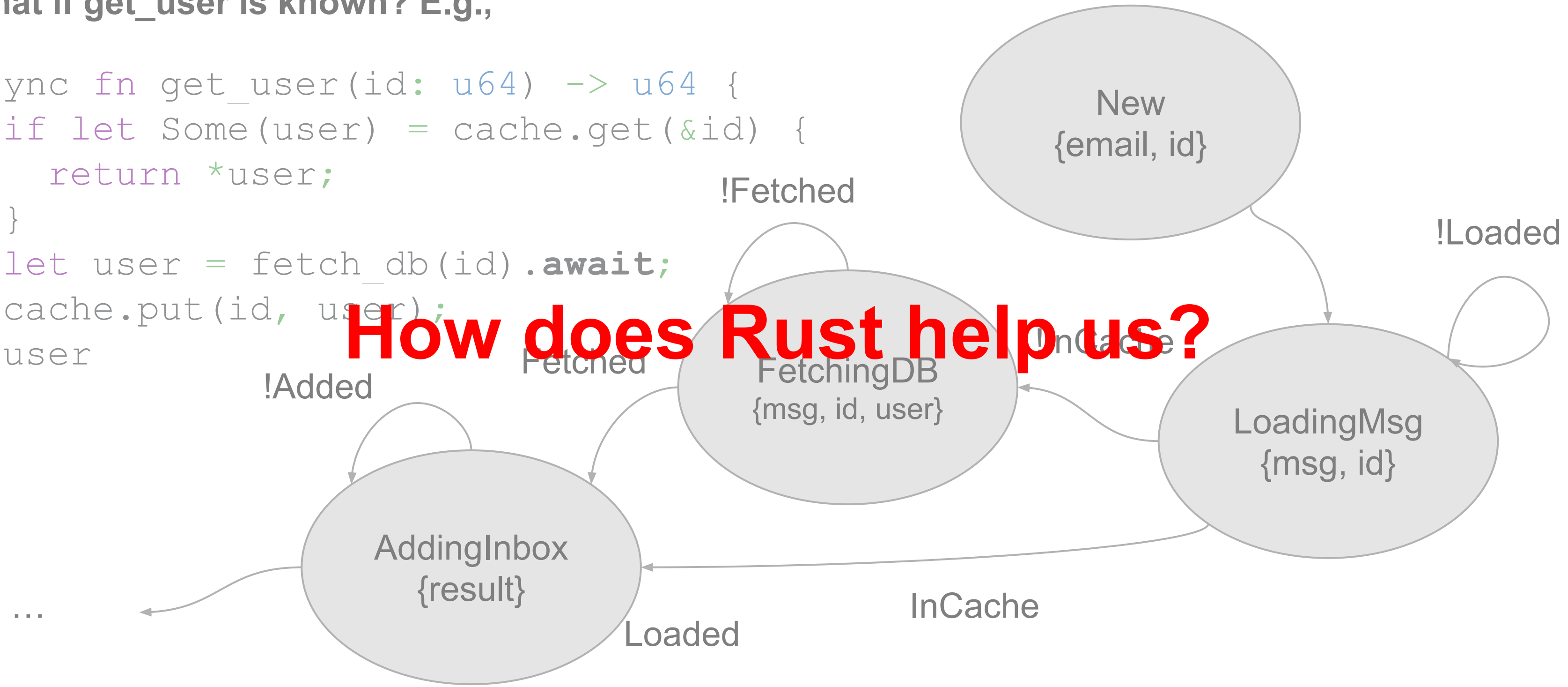
```
async fn get_user(id: u64) -> u64 {
  if let Some(user) = cache.get(&id) {
    return *user;
  }
  let user = fetch_db(id).await;
  cache.put(id, user);
  user
}
```



Pros: Optimize across functions!

What if get_user is known? E.g.,

```
async fn get_user(id: u64) -> u64 {
  if let Some(user) = cache.get(&id) {
    return *user;
  }
  let user = fetch_db(id).await;
  cache.put(id, user);
  user
}
```



How does Rust help us?

Compile-Time State Machine

Original

```
async fn add_to_inbox(...)
    -> Result<(), Error> {
  let msg = load_message(email) .await?;
  let user = get_user(id) .await?;
  user.verify_has_space( &msg)?;
  user.add_to_inbox(msg) .await
}
```

Desugars into...

```
enum AddToInboxState {
  New          {email: String, id: u64},
  LoadingMsg   {msg : Option<String>, id:
u64},
  GettingUser  {msg: String, user: Option<u64>},
  AddingInbox {result : Option<Result>},
}

struct AddToInbox {
  state: AddToInboxState
}
```

Compile-Time State Machine

```
impl Future for AddToInbox {
  type Self::Output = Result<(), Error>;
  fn poll(..., cx) -> Poll<Self::Output> {
    match self.state {
      // Union of New, LoadingMsg, …
      // Can only get (email, id) if New!
      New(email, id) => {

        // Callee is non-blocking, eg event queue
        // When done, callee uses cx.wake()
        load_message(email, cx);

        // If loading, msg is None
        // Else,       msg is Some(msg),
                       set by cx.wake()
        self.state = LoadingMsg{msg: None, id: id};
        Poll::Pending
      },
      ...
    }
  }
}
```

Original

```
async fn add_to_inbox(...)
    -> Result<(), Error> {
  let msg = load_message(email) .await?;
  let user = get_user(id) .await?;
  user.verify_has_space( &msg)?;
  user.add_to_inbox(msg) .await
}
```

Desugars into…

```
enum AddToInboxState {
  New          {email: String, id: u64},
  LoadingMsg   {msg : Option<String>, id:
u64},
  GettingUser  {msg: String, user: Option<u64>},
  AddingInbox {result : Option<Result>},
}

struct AddToInbox {
  state: AddToInboxState
}
```

Compile-Time State Machine

```
impl Future for AddToInbox {
  type Self::Output = Result<(), Error>;
  fn poll(..., cx) -> Poll<Self::Output> {
    match self.state {
      ...
      AddingInbox => {
        if let Some(res) = self.result {
          // Wake my caller!
          cx.wake();

          // Also an enum Poll { Pending, Ready(T) }
          Poll::Ready(res)
        } else {
          Poll::Pending
        }
      },
    }
  }
}
```

Original

```
async fn add_to_inbox(...)
    -> Result<(), Error> {
  let msg = load_message(email).await?;
  let user = get_user(id).await?;
  user.verify_has_space(&msg)?;
  user.add_to_inbox(msg).await
}
```

Desugars into…

```
enum AddToInboxState {
  New          {email: String, id: u64},
  LoadingMsg   {msg: Option<String>, id:
u64},
  GettingUser  {msg: String, user: Option<u64>},
  AddingInbox {result: Option<Result>},
}

struct AddToInbox {
  state: AddToInboxState
}
```

Compile-Time State Machine

Original

```
async fn add_to_inbox(...)
    -> Result<(), Error> {
  let msg = load_message(email) .await?;
  let user = get_user(id) .await?;
  user.verify_has_space( &msg)?;
  user.add_to_inbox(msg) .await
}
```

Desugars into...

```
enum AddToInboxState {
  New          {email: String, id: u64},
  LoadingMsg   {msg : Option<String>, id:
u64},
  GettingUser  {msg: String, user: Option<u64>},
  AddingInbox {result : Option<Result>},
}

struct AddToInbox {
  state: AddToInboxState
}
```

```
impl Future for AddToInbox {
  type Self::Output = Result<(), Error>;
  fn poll(..., cx) -> Poll<Self::Output> {
    match self.state {
      ...
      AddingInbox => {
        if let Some(res) = self.result {
          // Wake my caller!
          cx.wake();

          // Also an enum Poll { Pending, Ready(T) }
          Poll::Ready(res)
        } else {
          Poll::Pending
        }
      },
    }
  }
}
```

Union requires 1 heap allocation vs n allocations per callback!

Compile-Time State Machine

```
impl Future for AddToInbox {
  type Self::Output = Result<(), Error>;
  fn poll(..., cx) -> Poll<Self::Output> {
    match self.state {
      ...
      AddingInbox => {
        if let Some(res) = self.result {
          // Wake my caller!
          cx.wake();

          // Also an enum Poll { Pending, Ready(T) }
          Poll::Ready(res)
        } else {
          Poll::Pending
        }
      },
    }
  }
}
```

Who calls poll()?

How does this all run?

Original

```
async fn add_to_inbox(...)
    -> Result<(), Error> {
  let msg = load_message(email) .await?;
  let user = get_user(id) .await?;
  user.verify_has_space( &msg)?;
  user.add_to_inbox(msg) .await
}
```

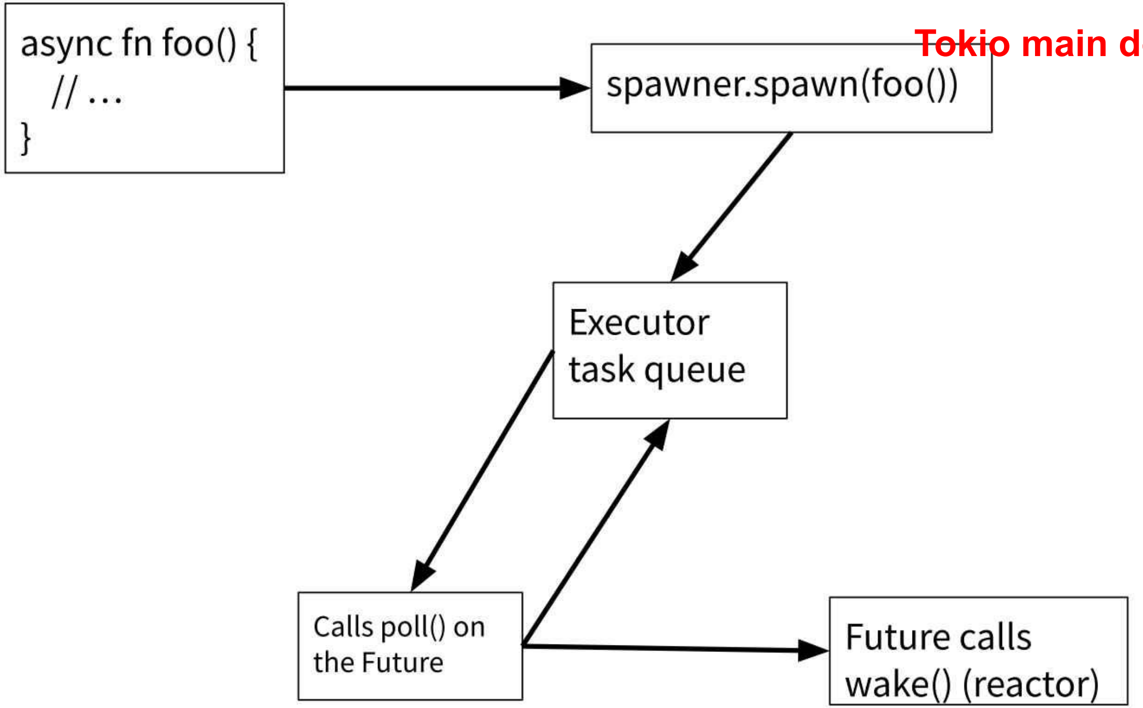
Desugars into…

```
enum AddToInboxState {
  New          {email: String, id: u64},
  LoadingMsg   {msg : Option<String>, id:
u64},
  GettingUser  {msg: String, user: Option<u64>},
  AddingInbox {result : Option<Result>},
}

struct AddToInbox {
  state: AddToInboxState
}
```

Super mega TLDR

(https://eventhelix.com/rust/rust-to-assembly-async-await/)



Summary

- Async-await is a great way to have many, many minimal-overhead tasks that exploit maximum concurrency
- Rust efficiently compiles async functions into state machines – no better way to do it yourself
- Tokio executes these async functions as tasks
- Comes with usual caveats: sharing memory safely, RAII, etc

Feedback Exercise

- NUS-wide student feedback is open now
- Please let me know what I should keep / what I should change!

See you next week!

Be sure to come because I am going to bribe you guys

Lifecycle of a Task (thanks Walter)

Lifecycle of a Task

```
async fn foo() {
  // …
}
```

Lifecycle of a Task

```
async fn foo() {
  // …
}
```

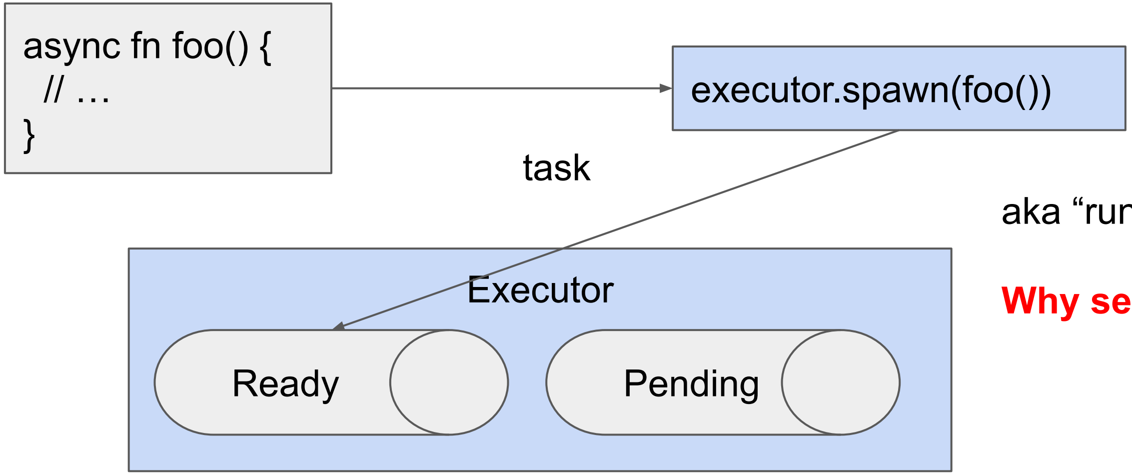
→ `executor.spawn(foo())`

Eg tokio::spawn, async_std::task::spawn

Executor may **run immediately until .await**

How does executor schedule many Futures?

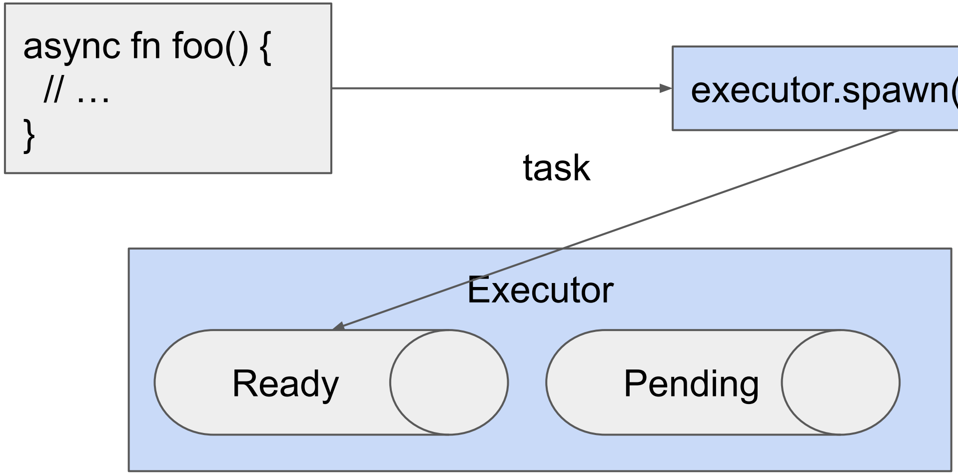
Lifecycle of a Task



aka “runtime”, “scheduler”, “event loop”

Why separate executor?

Lifecycle of a Task

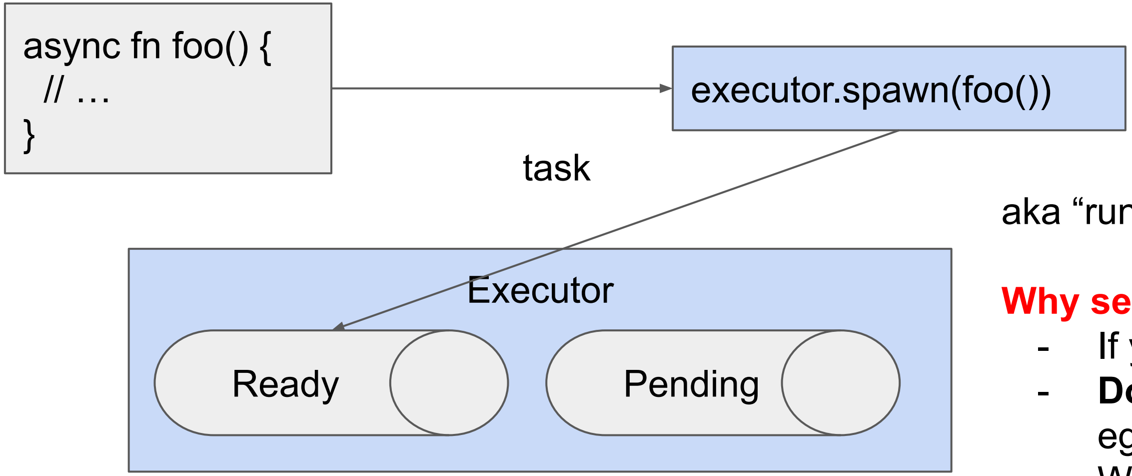


aka “runtime”, “scheduler”, “event loop”

Why separate executor?

- If you don’t use async, you **don’t pay for it**
- **Don’t want to alloc** / unbounded queue
  eg embedded systems
- Want cache locality for compute
  eg **thread queues**
- Want performance (eg latency, throughput)
  eg **work stealing, optimal scheduling**

Lifecycle of a Task



aka “runtime”, “scheduler”, “event loop”

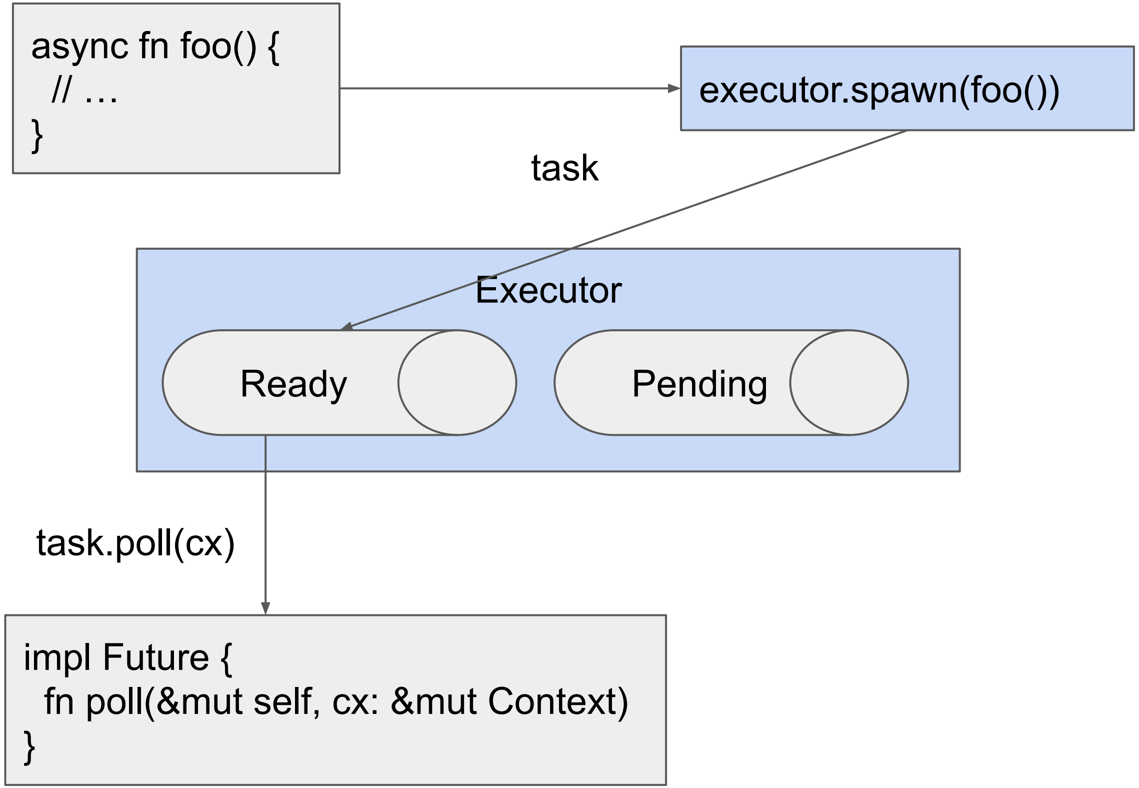
Why separate executor?

- If you don’t use async, you **don’t pay for it**
- **Don’t want to alloc** / unbounded queue
  eg embedded systems
- Want cache locality for compute
  eg **thread queues**
- Want performance (eg latency, throughput)
  eg **work stealing, optimal scheduling**

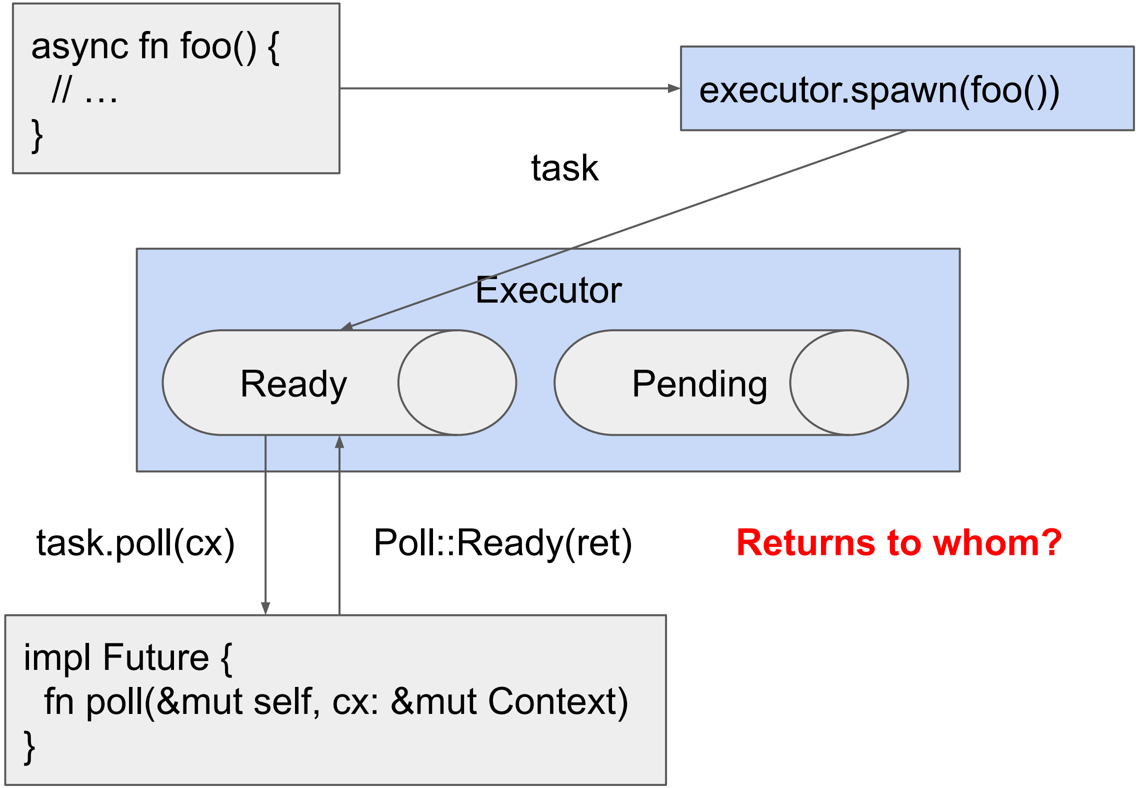
That’s why .await is lazy

- Eager execution allocates new tasks
- Lazy execution executes within parent fn

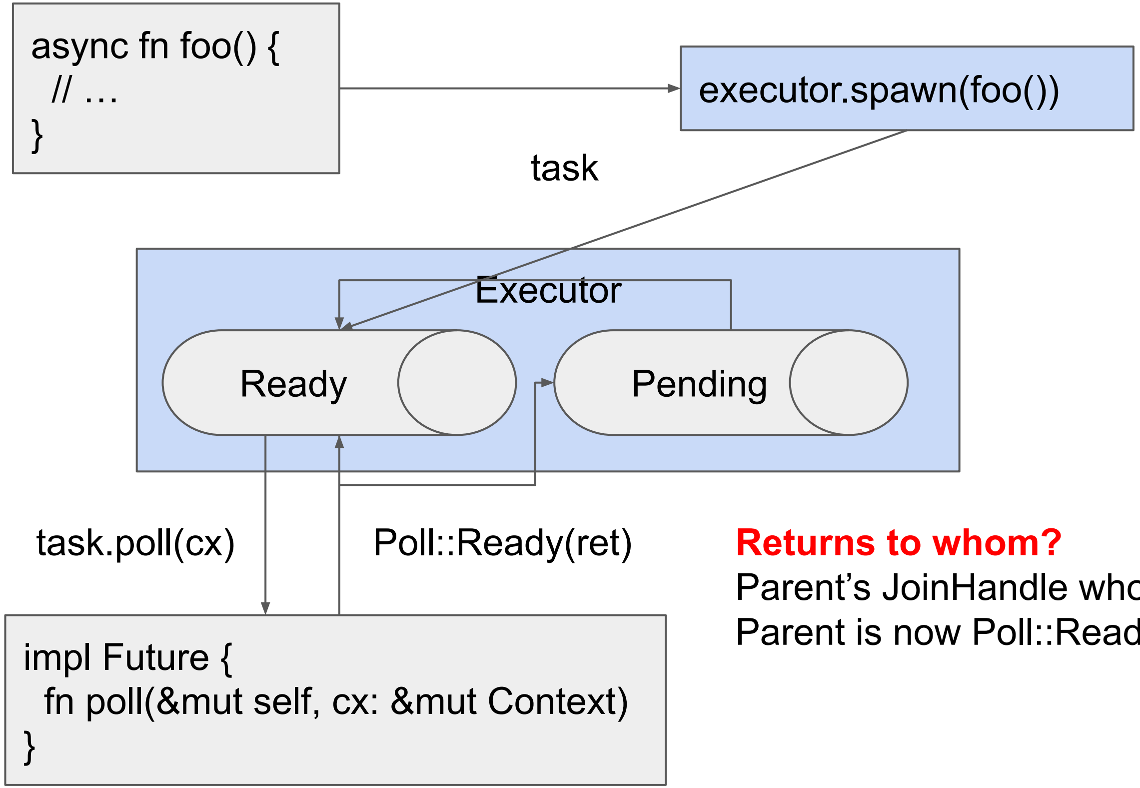
Lifecycle of a Task



Lifecycle of a Task

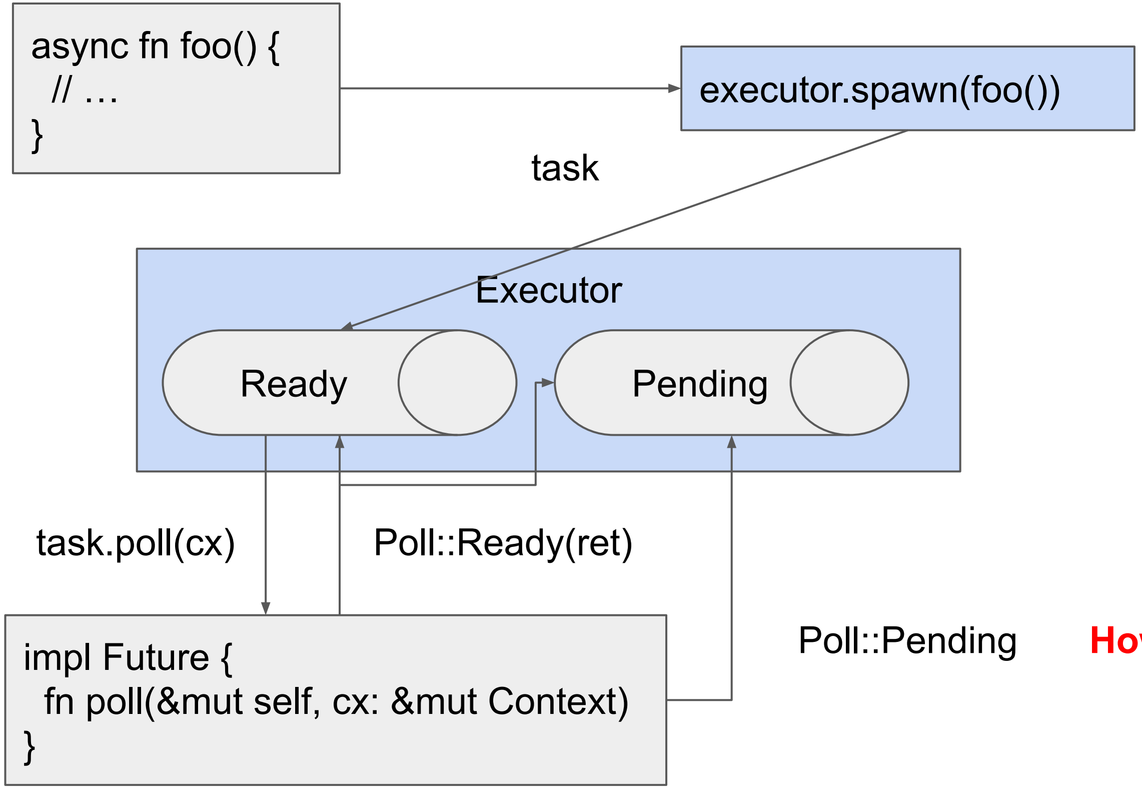


Lifecycle of a Task



Returns to whom?
Parent's JoinHandle who .await this Future
Parent is now Poll::Ready!

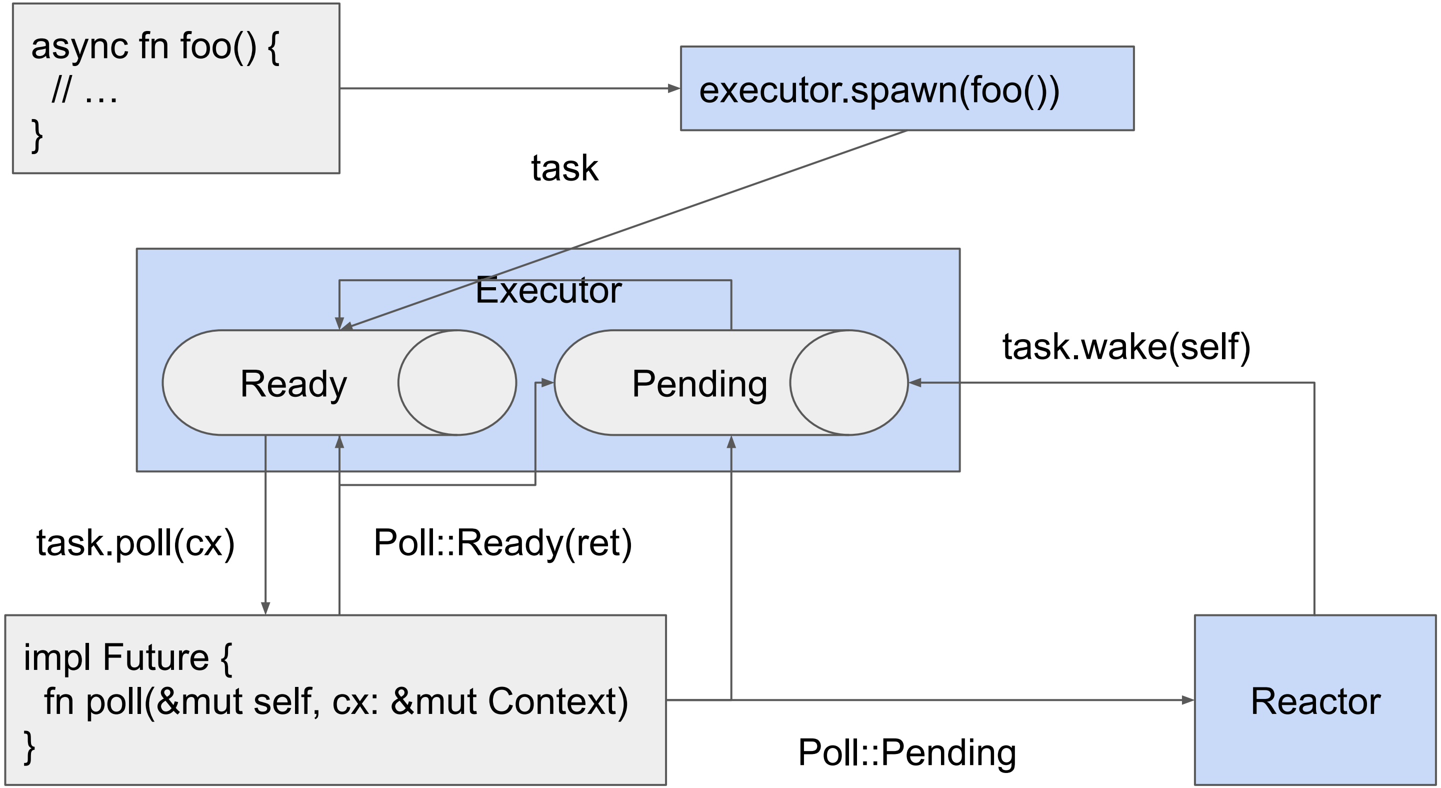
Lifecycle of a Task



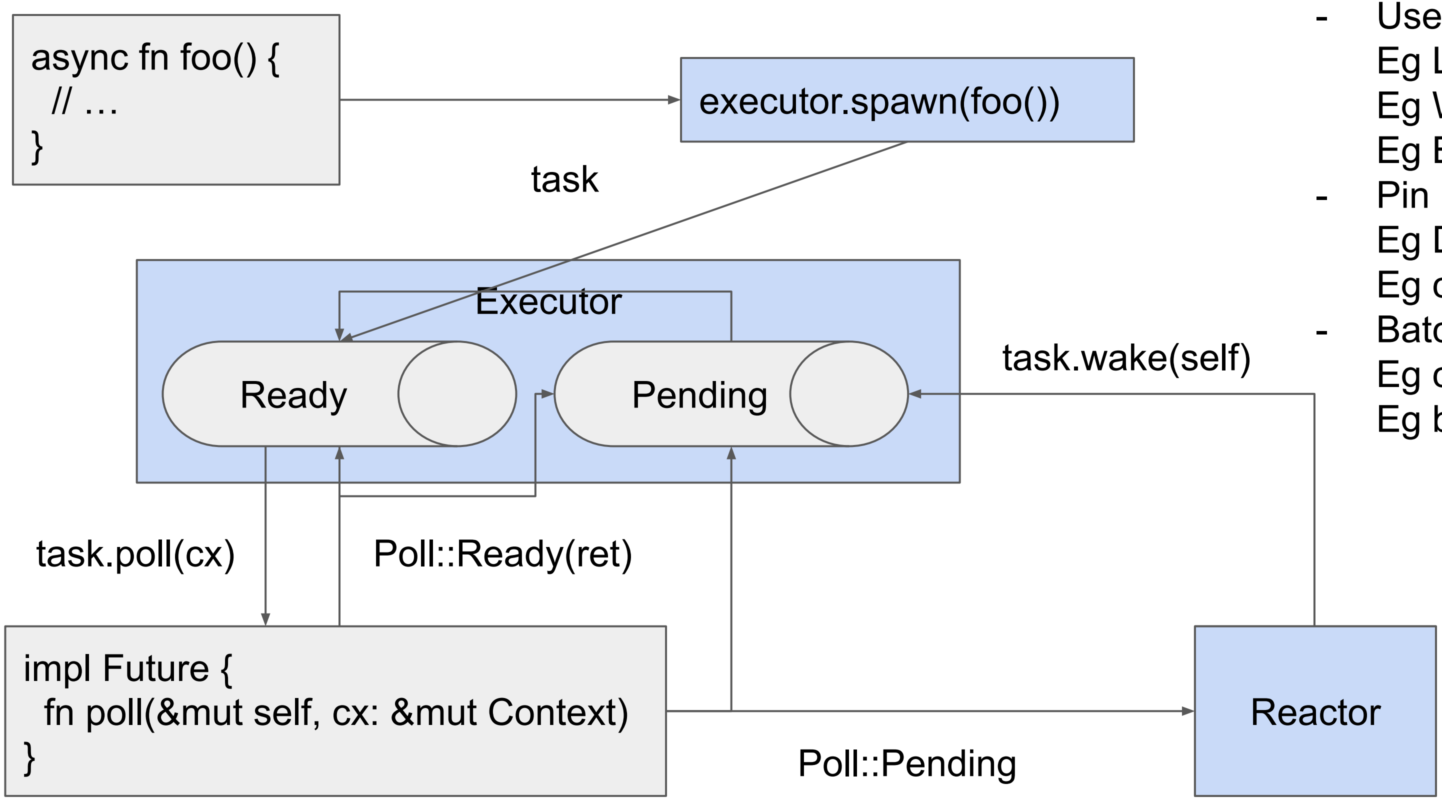
How to tell if ready?

Lifecycle of a Task

Why separate reactor?



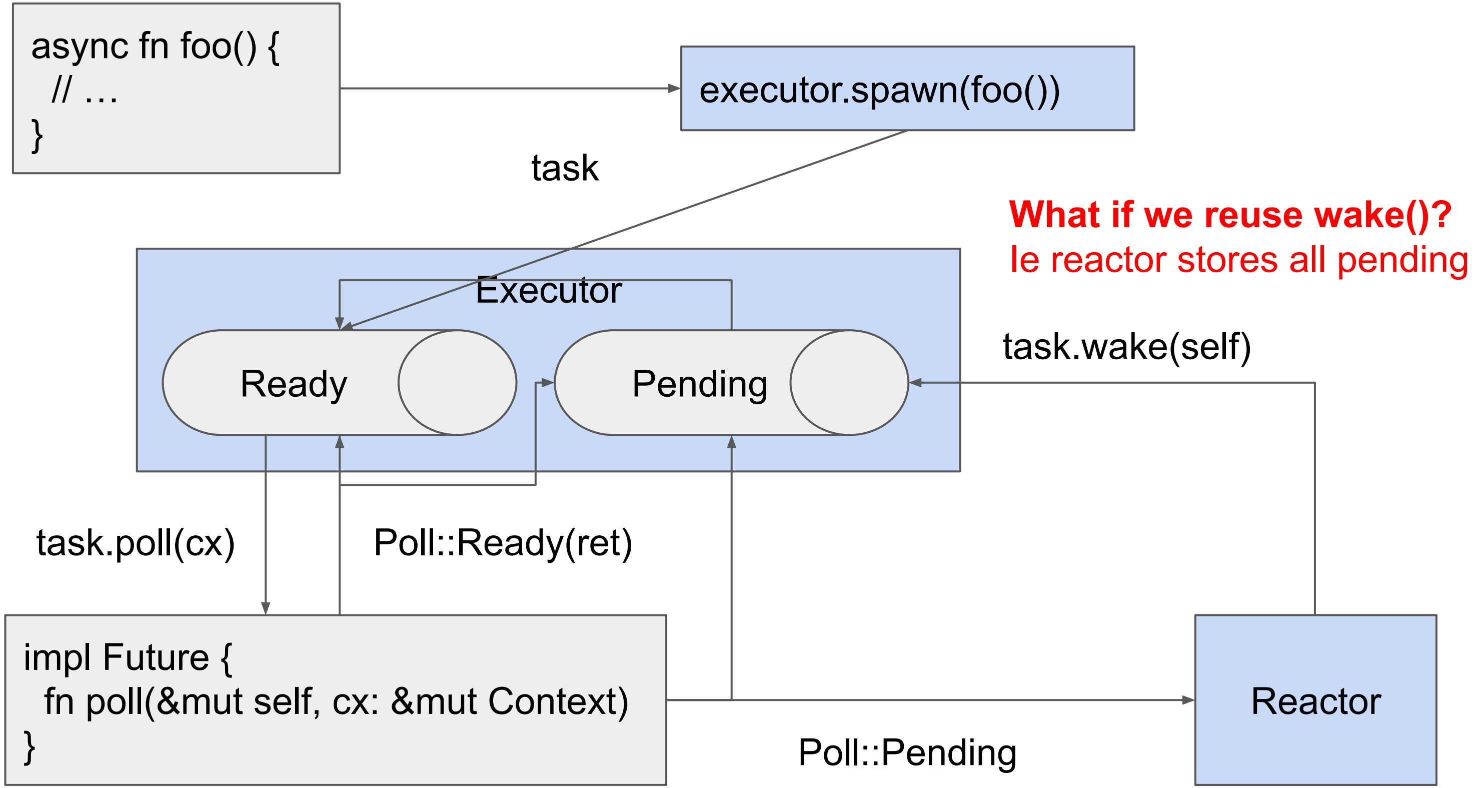
Lifecycle of a Task



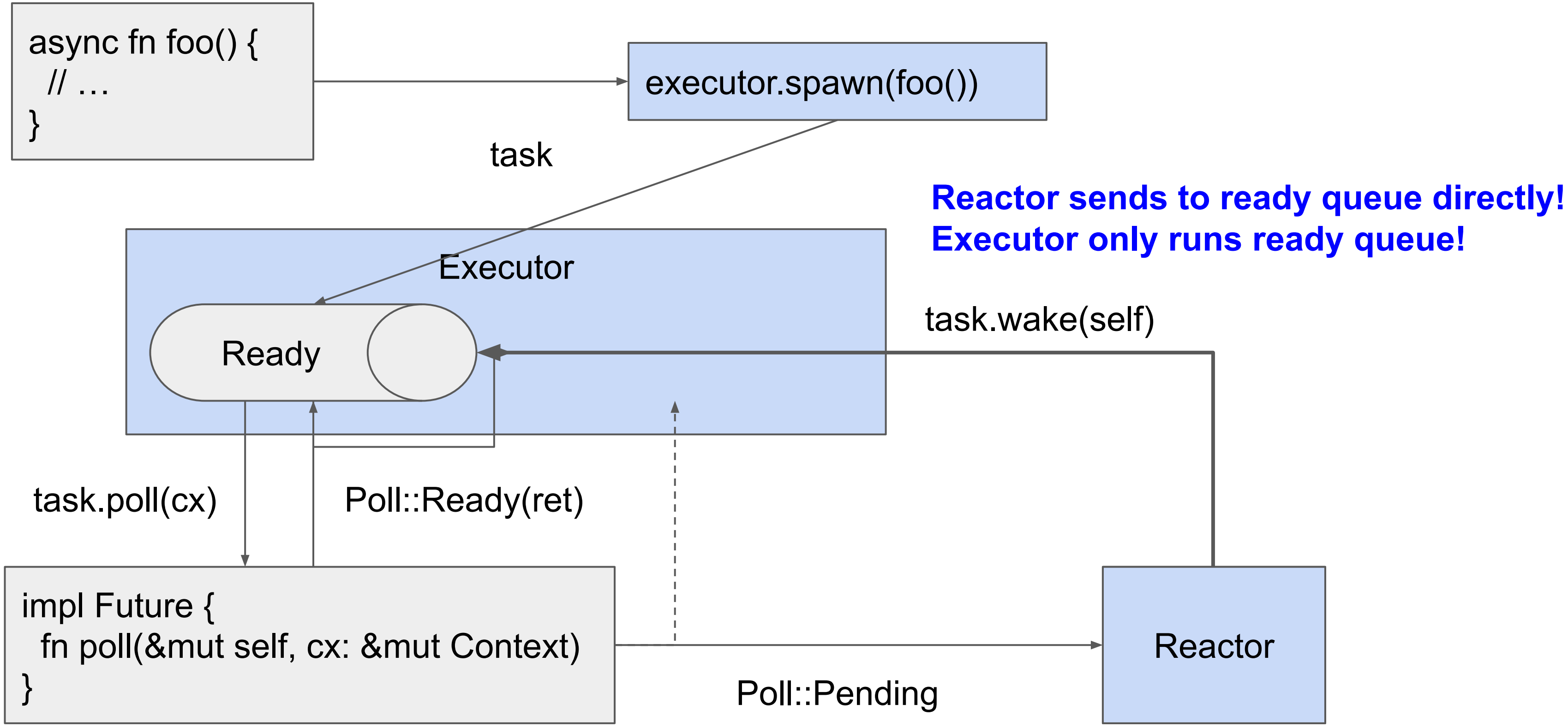
Why separate reactor?

- Use your own API
  Eg Linux epoll, io_uring()
  Eg Windows IOCP
  Eg Bare Metal GPIO
- Pin resource
  Eg DPDK, kernel bypass
  Eg core pinning, arena
- Batch operations
  Eg connection pool, caching
  Eg batch send/recv

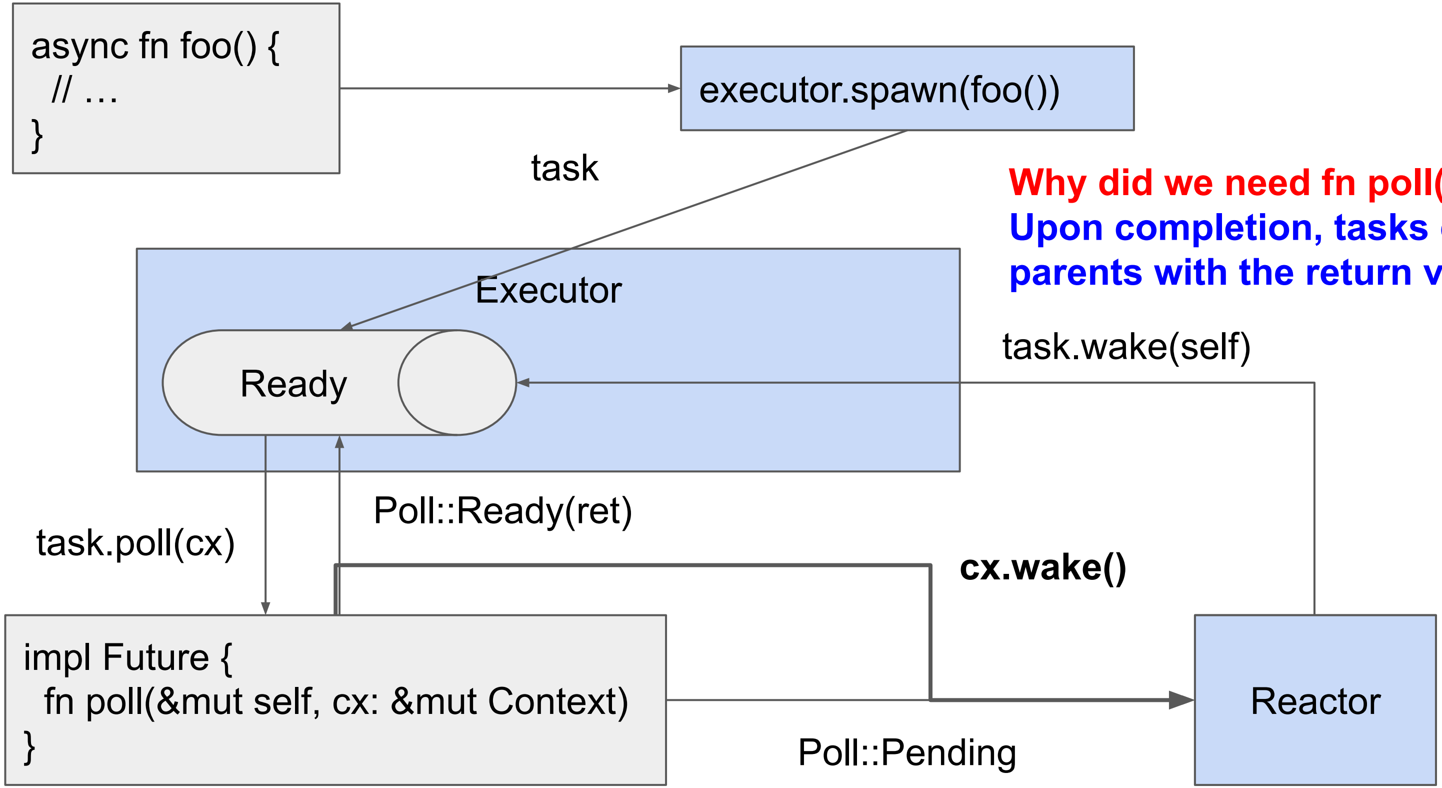
Lifecycle of a Task



Lifecycle of a Task



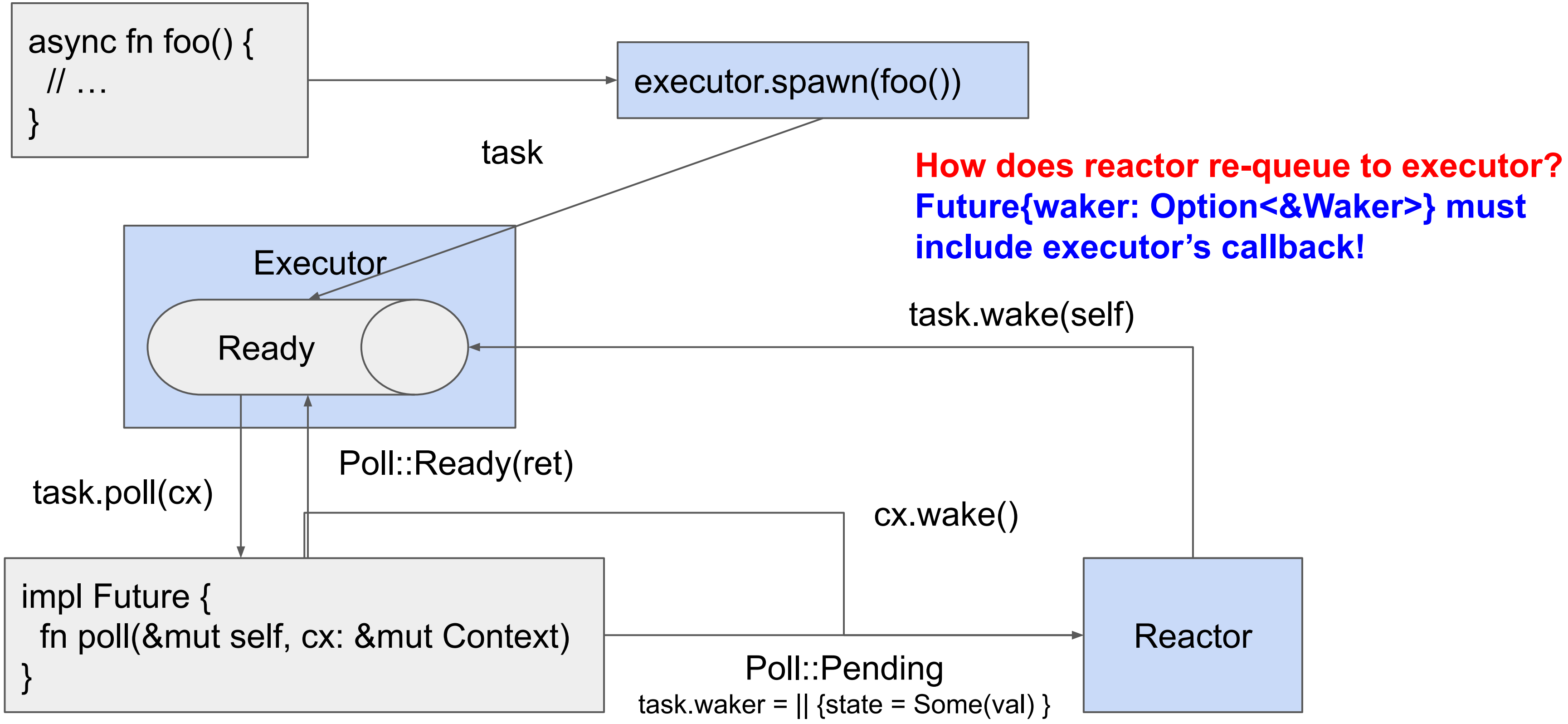
Lifecycle of a Task



Why did we need fn poll(.., cx)?

Upon completion, tasks can wake() their parents with the return value

Lifecycle of a Task



Interesting Behavior in Tokio (Sharing data across .await points)

Extra: sending data in tokio

- Let's run this innocent looking code

https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=d195f0f6b16cda5ee6441279187481cf

https://tokio.rs/tokio/tutorial/shared-state

Extra: sending data in tokio

- At every **await** point – Tokio executor can choose to run this task on a **different thread:** can be moved!
  - Every “alive” variable needs to be “Send” (movable to another thread!)
  - MutexGuard from std is not!

```rust
async fn foo(state: ShareState) {
    let mut st = state.lock().unwrap();

    bar(&mut st).await;
}
```

```
error: future cannot be sent between threads safely
   --> src/main.rs:13:17
    |
13  |       tokio::spawn(async move {
    |  _________________^
14  | |          foo(st).await;
15  | |      });
    | |_____^ future created by async block is not `Send`
    |
    = help: within `[async block@src/main.rs:13:17: 15:6]`, the trait `Send` is not
note: future is not `Send` as this value is used across an await
   --> src/main.rs:21:17
    |
19  |      let mut st = state.lock().unwrap();
    |          ------ has type `std::sync::MutexGuard<'_, State>` which is not `Send`
20  |
21  |      bar(&mut st).await;
    |                   ^^^^^^ await occurs here, with `mut st` maybe used later
22  | }
    | - `mut st` is later dropped here
note: required by a bound in `tokio::spawn`
   --> /playground/.cargo/registry/src/github.com-1ecc6299db9ec823/tokio-1.27.0/src/
    |
163 |         T: Future + Send + 'static,
    |                     ^^^^ required by this bound in `spawn`
    |
```

Extra: sending data in tokio

- Solution: use tokio Mutex which is Send-able

https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=2b406c1489d69c58a27650b334a35aa5

```
use tokio::sync::Mutex;
```

```
async fn foo(state: ShareState) {
    let mut st = state.lock().await;

    bar(&mut st).await;
}
```

https://tokio.rs/tokio/tutorial/shared-state