

17-350 / 17-650

Lecture 09: Asynchrony in Rust

Jonathan Aldrich

Sep 24, 2026



Lecture Plan

- From parallelism to **concurrency**, from threads to asynchronous tasks
- Futures from first principles: `poll`, `Waker`, executors
- `async fn` as a state machine, and why that requires `Pin`
- Tokio in practice: `spawn`, `Send`, and a lock held across `.await`
- `select!`, timers, channels, and cancellation
- The actor pattern and networking
- Where this is going: HW4 (to be released Tuesday)

First of three lectures on distributed systems: async today, CRDTs on Tuesday, collaborative text and testing next Thursday.

Where We Are: From Parallelism to Concurrency

- Lectures 07–08 and HW3: **parallelism**
 - Every thread is *busy computing* (tracing rays)
 - More cores means we finish sooner
- HW4: a replica in a replicated key-value store / text editor
 - Almost never computing, mostly **waiting**:
 - for a message from a peer
 - for the next gossip timer tick
 - for the user's next edit
 - Several waits outstanding at once; react to **whichever happens first**

Concurrency ≠ Parallelism

	Concurrency	Parallelism
Meaning	many things <i>in progress</i> at once	many things <i>executing</i> at once
Typical bottleneck	waiting (I/O, timers, users)	CPU
Goal	responsiveness, many simultaneous waits	throughput, speedup
Rust tools	async/await , tokio (today)	threads, <code>thread::scope</code> , <code>chunks_mut</code> , atomics (Lectures 07–08)

- They're independent: async code can run on **one** thread (HW4's test simulator) or on **many** (tokio's default runtime)

Option 1: A Thread per Activity

```
for peer in peers {  
    thread::spawn(move || loop {  
        let msg = read_message(&peer);    // blocks this thread until data arrives  
        handle(msg);  
    });  
}
```

- Simple, fine for a handful of peers
- Doesn't scale to thousands of concurrent waits (the "C10K problem"):
 - each thread reserves its own stack (Rust: 2MB of virtual memory)
 - the kernel context-switches every time one blocks or wakes
 - creating threads isn't free
- Most of these threads spend most of their time **blocked, doing nothing**

Option 2: An Event Loop with Callbacks

```
loop {  
    // Ask the OS which sockets/timers are ready (epoll / kqueue / IOCP)  
    for event in wait_for_ready_events() {  
        match event {  
            Readable(sock) => on_readable(sock, &mut conn_state[sock]),  
            TimerFired(t)  => on_timer(t, &mut timer_state[t]),  
        }  
    }  
}
```

- Efficient: one thread, many connections, no blocked threads
- But larger operations get shredded into callbacks
 - start a long operation, call read on a file, get called back when the read succeeds, etc.
 - what would have been **local variables must be hand-packed into structs** between events
- A hand-written state machine for every protocol ("callback hell")

async/await: Write It Like Threads, Run It Like an Event Loop

```
async fn serve_peer(mut stream: TcpStream) {  
    loop {  
        let msg = read_message(&mut stream).await; // pause here; don't block  
        handle(msg);  
    }  
}
```

- Reads like the blocking, thread-per-connection version
- The compiler turns it into the event-loop state machine for you

An `.await` is a point where this task (think: a lightweight thread) may pause so the OS thread can run something else, and later resume exactly where it left off.

The Future Trait

```
pub trait Future {  
    type Output;  
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output>;  
}  
  
pub enum Poll<T> {  
    Ready(T),  
    Pending,  
}
```

- A future is a value representing a computation that may not be done yet
- **Someone** repeatedly calls `poll`:
 - makes some computational progress, if possible, then returns:
 - `Ready(v)` : done, here's the result
 - `Pending` : not yet, poll me again later
- A **task** is a top-level future owned and polled by an executor (created by `spawn`); the futures it `.await`s are nested inside it
- (`Pin` and `Context` in a moment)

Futures Are Lazy

- Creating a future **does nothing**; it only makes progress when polled
 - Unlike JavaScript promises or Java's `CompletableFuture`, which start running as soon as they're created

```
async fn notify_peer(key: &str) { /* ... */ }

fn oops() {
    notify_peer("apples");           // creates a future... and drops it. Nothing happens.
}
```

```
warning: unused implementer of `Future` that must be used
= note: futures do nothing unless you `.await` or poll them
```

- Common first bug: forgetting `.await` (or `spawn` for independent tasks)

The Waker Contract

- Each call to `poll` receives a `Context`; `cx.waker()` is the `Waker` for the task being polled
 - purpose: lets the future tell the executor "poll me again"
- If `poll` returns `Pending`, the future **must arrange** for `cx.waker().wake()` to be called when it can make progress
 - then the `Future` will be polled again soon
 - otherwise nobody ever polls it again, and the task **hangs silently**
- Whoever polls the future (the **executor**) is then free to go do other work, or sleep, until some waker fires

A Hand-Written Future: Delay (1/2), the state

```
pub struct Delay {
    name: &'static str,      // for demo printouts
    dur: Duration,
    polls: u32,              // for demo printouts
    /// `None` until the first poll: futures are lazy, so nothing starts
    /// (no thread, no timer) until someone polls us.
    timer: Option<Arc<Mutex<TimerState>>>,
}

/// Shared between the Delay and the helper thread that times it.
struct TimerState {
    done: bool,              // set by the helper thread when time is up
    waker: Option<Waker>,    // who to wake when that happens
}
```

A Hand-Written Future: Delay (2/2), poll

```
impl Future for Delay {
    type Output = ();
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<()> {
        let this = self.get_mut(); // OK because Delay is Unpin (more soon)
        match &this.timer {
            None => { // first poll: start the timer
                let timer = Arc::new(Mutex::new(TimerState {
                    done: false, waker: Some(cx.waker().clone()) }));
                let for_thread = Arc::clone(&timer);
                let dur = this.dur;
                thread::spawn(move || {
                    thread::sleep(dur);
                    let mut state = for_thread.lock().unwrap();
                    state.done = true;
                    if let Some(waker) = state.waker.take() { waker.wake(); }
                });
                this.timer = Some(timer);
                Poll::Pending // allowed: we arranged a wake
            }
            Some(timer) => { // polled again, e.g., by select!
                let mut state = timer.lock().unwrap();
                if state.done { Poll::Ready(()) }
                else { state.waker = Some(cx.waker().clone()); Poll::Pending }
            }
        }
    }
}
```

Real runtimes use one timer thread and a timer wheel, not a thread per timer. This version just makes the contract visible.

An Executor in Ten Lines

```
struct ThreadWaker(thread: Thread);

impl Wake for ThreadWaker {
    fn wake(self: Arc<Self>) { self.0.unpark(); }    // waking = unpark the executor
}

/// Run one future to completion on the current thread.
pub fn block_on<F: Future>(fut: F) -> F::Output {
    let mut fut = pin!(fut);                          // pinned; never moves again
    let waker = Waker::from(Arc::new(ThreadWaker(thread::current())));
    let mut cx = Context::from_waker(&waker);
    loop {
        match fut.as_mut().poll(&mut cx) {
            Poll::Ready(value) => return value,
            Poll::Pending => thread::park(),           // sleep until a waker fires
        }
    }
}
```

- There's nothing magic in an executor: it's a loop that polls, then sleeps until woken

Demo: delay-future

```
1. Futures are lazy
   created a Delay; no poll has happened, so no timer is running
   ...300 ms later, now handing it to block_on
       [lazy] poll #1
       [lazy] poll #2
   finished 801 ms after creation ( $\approx 300 + 500$ : the timer started at the first poll)
```

- 800 ms, not 500: **nothing happened until the first poll**
- Exactly two polls: **Pending** (timer armed, waker stored), then **Ready** after the wake

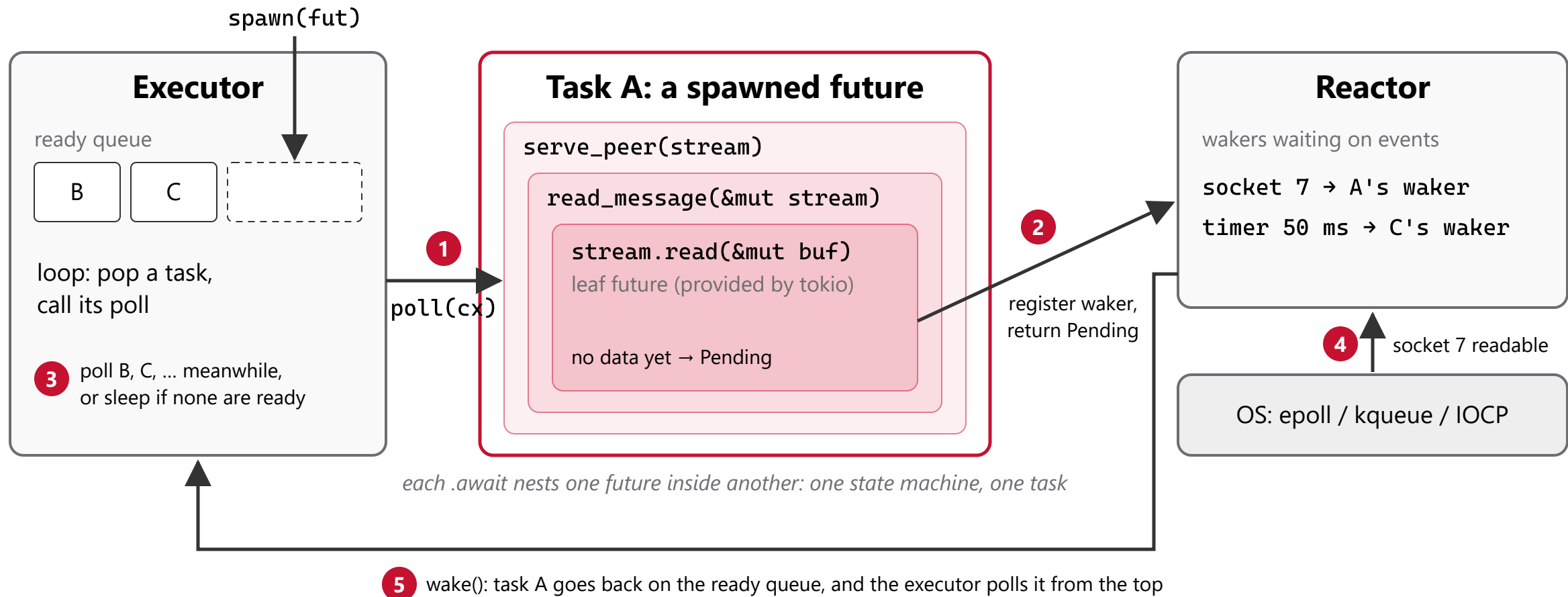
[Live demo] `cargo run -p delay-future` (in `09-async/demos/`)

Executors and Reactors

A real runtime has two cooperating halves:

1. The **executor** pops task A off its ready queue and calls `poll`
 2. Task A tries to read a socket that has no data yet, so it registers its waker with the **reactor** and returns `Pending`
 3. The executor polls other ready tasks, or sleeps if there are none
 4. The OS reports the socket readable (`epoll` / `kqueue` /IOCP), and the reactor calls task A's waker
 5. `wake()` puts task A back on the ready queue, and we're back at step 1
- The **reactor** turns OS readiness events into `wake()` calls
 - The **executor** turns `wake()` calls into `poll` calls

How It Fits Together



future = any value with `poll` · **task** = a top-level future the executor owns · **waker** = a handle that re-queues its task

Tokio = Executor + Reactor + Async Std

- **Executor:** multi-threaded and **work-stealing**
 - one worker per core, each with a queue of ready tasks; idle workers steal from busy ones
 - the same idea as HW3's dynamic tile queue, applied to tasks
- **Reactor:** I/O readiness and timers
- **Async versions of std:** `tokio::net`, `tokio::time`, `tokio::sync`, `tokio::fs`, ...
- Rust's standard library defines `Future` but ships **no runtime**. You choose one; tokio is the de facto standard.

async fn Is a State Machine

```
async fn sync_once(peer: &Peer) -> usize {  
    let request = build_request();  
    let reply = peer.send(request).await;    // await point 1  
    let ops = decode(reply);  
    peer.ack().await;                        // await point 2  
    ops.len()  
}
```

- Calling `sync_once(&p)` runs **none** of the body. It returns a future (a state machine) positioned at the start
- Each `poll` runs the body until the next `.await` that returns `Pending`
- What gets saved across a pause? **Exactly the locals still needed afterward**

Roughly What the Compiler Generates

```
enum SyncOnce<'a> {
    Start          { peer: &'a Peer },
    AwaitingReply  { peer: &'a Peer, fut: SendFuture }, // `request` was moved into send
    AwaitingAck    { ops: Vec<Op>, fut: AckFuture },    // `peer`, `reply` no longer needed
    Done,                                                // finished; holds nothing
}

// poll(): a `match` on the current state, advancing until a sub-future is Pending
// Start          => build request; fut = peer.send(request); go to AwaitingReply
// AwaitingReply  => match fut.poll(cx) {
//     Pending      => return Pending,          // state is saved
//     Ready(reply) => { ops = decode(reply); fut = peer.ack();
//                     go to AwaitingAck } }
// AwaitingAck    => match fut.poll(cx) {
//     Pending      => return Pending,
//     Ready(_)     => { let n = ops.len(); go to Done; return Ready(n) } }
// Done           => panic!("`async fn` resumed after completion")
```

- It's the hand-packed struct from the event-loop slide, written for you by the compiler
- Nested `async fn` calls nest their state machines inline

Future Size = Largest Suspended State

An enum is as large as its largest variant (Lecture 03), and the same holds for futures:

<code>async fn</code> (from the <code>delay-future</code> demo)	<code>size_of_val(&fut)</code>
two awaits, small locals	56 bytes
<code>[u8; 4096]</code> live across an <code>.await</code>	4152 bytes
same buffer, but dead before the <code>.await</code>	56 bytes

- **Zero-cost:** no heap allocation per `async fn` call. A whole tree of nested futures is one flat, statically sized value
- Allocation happens only when you **spawn** a task (it gets boxed) or explicitly `Box::pin`

Recursive `async fn` needs `Box::pin`: otherwise the state machine would contain itself, an infinitely sized type, just like a recursive enum without a `Box`.

Why Pin? Futures That Point Into Themselves

```
async fn example() {  
    let buf = [0u8; 1024];  
    let slice = &buf[..10];           // a borrow of a local...  
    something().await;                // ...held across an await  
    use_it(slice);  
}
```

- Both `buf` and `slice` are saved in the **same** state-machine variant
- So `slice` points **into the future itself**
- If the future were moved (memcpy'd elsewhere) between polls, `slice` would **dangle**
- Ordinary Rust forbids self-referential structs; `async fn` creates them all the time. `Pin` makes this sound.

Pin as a Safety Argument

The same structure as our `unsafe` write-ups (HW2, HW3, Lecture 08):

- **Hazard:** moving a self-referential future after its first poll leaves dangling internal pointers
- **Invariant:** all such futures are pinned. Once a value is pinned, it never moves again until it is dropped
- **Establish:** `Box::pin(fut)` (on the heap) or `pin!(fut)` (on the stack, shadowing the original so it can't be used again). Executors pin every task they run
- **Maintain:** `Pin<&mut T>` **never hands out** `&mut T` (which would allow `mem::swap`, a move) unless `T: Unpin`. The escape hatch, `get_unchecked_mut`, is `unsafe`

`Unpin` is an auto trait, like `Send` and `Sync`: most types (`i32`, `String`, `Vec`, our `Delay`) don't care about moving. Compiler-generated futures are `!Unpin`.

In Practice: When Do You Write `pin!` ?

- Rarely. `.await` and `spawn` handle pinning for you
- The common case: polling the **same** future repeatedly by **reference**, e.g., one deadline across many loop iterations

```
let deadline = tokio::time::sleep(Duration::from_secs(5));
tokio::pin!(deadline);           // shadows `deadline`: now a Pin<&mut Sleep>

loop {
    tokio::select! {
        _ = &mut deadline      => break,           // same timer every iteration
        Some(msg) = rx.recv()  => handle(msg),
    }
}
```

- Forget the `pin!` and the compiler says "`PhantomPinned` cannot be unpinned ... consider using the `pin!` macro". This is the fix it means

Tokio Basics

```
#[tokio::main]                // builds a multi-threaded runtime; block_on(main)
async fn main() {
    let handle: JoinHandle<u64> = tokio::spawn(async {
        fetch_count_from_peer().await
    });
    // ... do other things concurrently ...
    let count = handle.await.unwrap(); // Err if the task panicked
}
```

```
pub fn spawn<F>(future: F) -> JoinHandle<F::Output>
where
    F: Future + Send + 'static,
    F::Output: Send + 'static,
```

- `spawn` creates an **independent task**, which the runtime polls concurrently with everything else
- Look at those bounds...

`spawn` Requires `Send + 'static`, Again

	<code>thread::spawn</code> (Lecture 07)	<code>tokio::spawn</code>
<code>'static</code>	the thread may outlive its caller	the task may outlive its caller
<code>Send</code>	the closure moves to another thread	the task may resume on a different worker thread after any <code>.await</code>

- Work stealing means a task can be polled on core 1, pause, and be polled again on core 3
- A future is `Send` only if **every value held across an `.await` is `Send`**
 - That includes references: a `&T` held across an `.await` is `Send` only if `T: Sync` (Lecture 07)
 - So spawned *generic* async code needs bounds like `Op: Send + Sync + 'static`
- It's the same compile-time race-freedom argument, now applied to tasks

Live Demo: A Lock Held Across `.await`

```
fn broken(counts: Arc<std::sync::Mutex<Counts>>, key: &'static str) -> JoinHandle<()> {
    tokio::spawn(async move {
        let mut guard = counts.lock().unwrap();
        *guard.entry(key.to_string()).or_default() += 1;
        notify_peer(key).await;           // `guard` is still alive here
    })
}
```

- A perfectly reasonable-looking task: bump a count, then tell a peer
- What will the compiler say?

[Live demo] `cargo build -p not-send --features broken`

What the Compiler Says

```
error: future cannot be sent between threads safely
      tokio::spawn(async move {
        -----^
        |         let mut guard = counts.lock().unwrap();
        |         ...
        |     })
        -----^ future created by async block is not `Send`

= help: within `{async block}`, the trait `Send` is not implemented for
      `std::sync::MutexGuard<'_, HashMap<String, u64>>`
note: future is not `Send` as this value is used across an await
      let mut guard = counts.lock().unwrap();
      ----- has type `std::sync::MutexGuard<'_, HashMap<String, u64>>`
               which is not `Send`
      notify_peer(key).await;
                        ^^^^^ await occurs here, with `mut guard` maybe used later
note: required by a bound in `tokio::spawn`
      F: Future + Send + 'static,
                        ^^^^ required by this bound in `spawn`
```

- A std mutex must be unlocked **by the thread that locked it**, and after the `.await` we might be on a different thread
- Holding a lock across an arbitrarily long pause is a latency and deadlock hazard anyway

Excerpted from rustc 1.93 output.

Let's fix this with `drop(guard)`; does this fix the problem?

```
tokio::spawn(async move {  
    let mut guard = counts.lock().unwrap();  
    *guard.entry(key.to_string()).or_default() += 1;  
    drop(guard); // released before the await!  
    notify_peer(key).await;  
})
```

[Live demo] `cargo build -p not-send --features try-drop`

Let's fix this with `drop(guard)`; does this fix the problem? no!

```
tokio::spawn(async move {  
    let mut guard = counts.lock().unwrap();  
    *guard.entry(key.to_string()).or_default() += 1;  
    drop(guard); // released before the await!  
    notify_peer(key).await;  
})
```

```
note: future is not `Send` as this value is used across an await  
    |           notify_peer(key).await;  
    |           ^^^^^^ await occurs here, with `mut guard` maybe used later
```

- **Still rejected** (rustc 1.93). The compiler decides what's held across an `.await` by **scope**, not by moves, and `guard` is still in scope
- The check is **conservative**: sound, but not perfectly precise

[Live demo] `cargo build -p not-send --features try-drop`

Three Fixes That Work

```
{  
    let mut guard = counts.lock().unwrap();  
    *guard.entry(key.to_string()).or_default() += 1;  
}  
notify_peer(key).await;  
// Fix 1: the scope ends, so the guard is gone
```

Fix	4 tasks × 10 ms notify
1. Scope the guard in a block, so it drops before the .await	~16 ms
2. Don't share at all: one task owns the map (the <i>actor pattern</i> , coming up)	~16 ms
3. tokio::sync::Mutex, whose guard <i>may</i> be held across .await	~62 ms

- Fix 3 compiles, but the lock now **serializes the notifications**. It's sometimes necessary, and it's a smell as a default

[Live demo] `cargo run -p not-send`

Don't Block the Executor

- Tokio's scheduling is **cooperative**: a task gives up its worker thread **only at an `.await`**
 - Lecture 07's OS threads are **preemptive**: the kernel interrupts them
- A blocking call inside a task stalls **every other task on that worker**

Instead of...	Use
<code>std::thread::sleep(d)</code>	<code>tokio::time::sleep(d).await</code>
<code>std::net::TcpStream</code> , blocking <code>std::fs</code>	<code>tokio::net</code> , <code>tokio::fs</code>
a long CPU-bound loop	<code>tokio::task::spawn_blocking(...)</code> , or OS threads as in HW3

- Rule of thumb: an async task should reach an `.await` often, within roughly 10–100 μ s of work

Tokio's Channels

Channel	Shape	Role in HW4
<code>mpsc</code>	many senders → one receiver, a queue	an actor's mailbox ; local edits arrive this way
<code>oneshot</code>	exactly one value, once	the reply to a request
<code>watch</code>	many readers see the latest value only	publishing replica snapshots
<code>broadcast</code>	every message to every receiver	(not needed)

- `send(...).await` on a bounded `mpsc` waits when the channel is full (backpressure); `try_send` never waits
- A `recv()` returning `None` means every sender is gone, so shut down
- `watch` : the writer calls `send_replace(v)` ; readers call `borrow()` , or `changed().await` to wait for a new value
 - `borrow()` holds a lock, like a `MutexGuard` . Clone what you need and **don't hold it across an `.await`**
 - In a spawned task the compiler rejects it (not `Send`); anywhere else (`main` , the HW4 simulator's hosts) it compiles and **can deadlock**

Timers

```
// Pause this task (not the thread) for 50 ms
tokio::time::sleep(Duration::from_millis(50)).await;

// A periodic tick, e.g., "gossip every 100 ms"
let mut gossip = tokio::time::interval(Duration::from_millis(100));
gossip.tick().await; // the first tick completes immediately

// Give up on a slow peer
match tokio::time::timeout(Duration::from_millis(500), TcpStream::connect(addr)).await {
    Ok(Ok(stream)) => { /* connected */ }
    Ok(Err(e))      => { /* connection failed */ }
    Err(_elapsed)  => { /* took longer than 500 ms */ }
}
```

- `timeout` is a future holding **two** futures, yours and a `sleep`, and it polls **both** each time it's polled: whichever finishes first wins, and the other is dropped. (`select!`, `next`, generalizes this)
- To bound a whole multi-step exchange (connect, send, await the reply), wrap it in one async block: `timeout(d, async { ... }).await`

select! : Wait for Whichever Comes First

The main loop of a replica (from the `gossip-toy` demo, abbreviated):

```
let mut sensor = interval(Duration::from_millis(300));
let mut gossip = interval(Duration::from_millis(200));
loop {
    tokio::select! {
        _ = sensor.tick(), if readings_left > 0 => { /* take a local reading */ }
        Some(msg) = inbox.recv() => { /* merge a peer's value */ }
        _ = gossip.tick() => { /* send my value to a peer */ }
        _ = &mut shutdown => break,
    }
}
```

- Polls all branch futures; the **first to complete** wins, and its body runs
- Each pass makes **new** branch futures; the losers are dropped, but the state lives in `inbox`, `sensor`, `gossip`, outside the loop.
`&mut shutdown` keeps *one* future across passes
- This is the shape of HW4 Part 3's node: local edits, peer messages, a gossip timer

select! Details

- `, if cond` is a **precondition**: when false, that branch is disabled for this iteration
- **Pattern mismatch disables the branch**: `Some(msg) = inbox.recv()` ignores `None` (the channel closed) instead of running the body
- If several branches are ready at once, tokio picks one **at random**, for fairness
 - `biased;` as the first line makes it check branches in order instead
 - (Remember "random" when we get to testing next Thursday)
- Bodies run **one at a time, inside the loop**: while a body is running (or awaiting), no other branch is being serviced
 - Keep bodies short, and don't await slow things there

Cancellation Is Just `drop`

- When one `select!` branch wins, the other branch futures are **dropped**, and never resume (the next pass of the loop makes new ones)
- The same happens when a `timeout` expires, or when you call `JoinHandle::abort()`
 - Careful: merely *dropping* a `JoinHandle` does **not** cancel a spawned task. The task keeps running, detached
- Rust has no separate cancellation API: the future stops at its last `.await`, and destructors run (sockets close, guards release)
- Contrast with threads: you **can't** safely kill a thread from outside
- With futures it's always possible, which also means it can happen at any `.await`

Cancel Safety

A future is **cancel-safe** if dropping it mid-way loses nothing.

```
// NOT cancel-safe as a select! branch:
async fn read_msg(s: &mut TcpStream) -> io::Result<Vec<u8>> {
    let len = s.read_u32().await?;           // if cancelled after this...
    let mut buf = vec![0; len as usize];
    s.read_exact(&mut buf).await?;          // ...the length prefix is gone, and the
    Ok(buf)                                  // stream is in the wrong position to restart
}
```

- `mpsc::Receiver::recv` is cancel-safe: either you got the message, or it's still in the channel
- Rule: before putting a future in a `select!` branch, check the "Cancel safety" section of its tokio docs
- For network messages, use a **framed stream** (two slides ahead), whose `next()` is cancel-safe

The Actor Pattern

```
enum Command {
    Increment { key: String },
    Get { key: String, reply: oneshot::Sender<u64> },
}

fn spawn_counter_actor() -> mpsc::Sender<Command> {
    let (tx, mut rx) = mpsc::channel::<Command>(32);
    tokio::spawn(async move {
        let mut counts = Counts::new(); // owned by this task alone
        while let Some(cmd) = rx.recv().await {
            match cmd {
                Command::Increment { key } => *counts.entry(key).or_default() += 1,
                Command::Get { key, reply } =>
                    { let _ = reply.send(counts.get(&key).copied().unwrap_or(0)); }
            }
        }
        // all senders gone: exit
    });
    tx
}
```

- One task **owns** the state; everyone else sends it messages
- A request that needs an answer carries a `oneshot::Sender` for the reply

Actor vs. Shared State

	Actor (one owner task)	<code>Arc<Mutex<State>></code>
Locks	none	every access
Held across <code>.await</code> ?	impossible	must be scoped carefully
Where invariants are maintained	one place: the actor loop	every task that locks
Cost	a message per operation	contention on the lock

- Lecture 08 made the module the **trusted computing base** for an invariant; here the actor task plays the same role
- HW4's replica has an invariant about its operation log. With an actor, it's maintained in exactly one loop
- HW4 lets you choose, and asks you to justify the choice

Networking: Listen, Accept, Forward

```
async fn accept_loop(listener: TcpListener, mailbox: mpsc::Sender<Gossip>) {
    loop {
        let Ok((stream, _peer)) = listener.accept().await else { continue };
        let mailbox = mailbox.clone();
        tokio::spawn(async move {                                // one task per connection
            let mut frames = FramedRead::new(stream, LengthDelimitedCodec::new());
            while let Some(Ok(frame)) = frames.next().await {
                match serde_json::from_slice::<Gossip>(&frame) {
                    Ok(msg) => if mailbox.send(msg).await.is_err() { break },
                    Err(e)  => eprintln!("dropping malformed frame: {e}"),
                }
            }
        });
    }
}
```

- The actor **never touches a socket**: handler tasks decode messages and forward them into its mailbox
- Sending: `TcpStream::connect(addr)`, done in its own spawned task, never inside the actor loop

TCP Is a Byte Stream: Frame Your Messages

- TCP delivers **bytes**, not **messages**: one write may arrive as two reads, and two writes may arrive as one read
- Fix: **length-delimited framing**, a 4-byte length followed by that many bytes

```
#[derive(Serialize, Deserialize)]
struct Gossip { from: usize, max_seen: u64 }

let mut frames = FramedWrite::new(stream, LengthDelimitedCodec::new());
frames.send(Bytes::from(serde_json::to_vec(&msg)?)).await?;
```

```
wire format: [u32 length = 25] {"from":0,"max_seen":123}
```

- `serde` generates the (de)serialization; `serde_json` because you can **read** the messages when debugging
- HW4 provides these helpers, so you won't write marshalling code by hand

Demo: gossip-toy

Three replicas, each taking 5 sensor readings and gossiping the **largest value seen**; the merge is `max`.

```
node 2: local reading 955 (new max)
node 1: learned max 955 from node 2
node 0: learned max 955 from node 1

final values: [955, 955, 955]
all 3 replicas converged to 955
```

- The same actor code runs over in-memory channels and over TCP
- Lost or duplicated gossip is harmless: `max(a, a) = a`
- **For Tuesday:** what if we wanted the **total number** of readings instead? Why doesn't `+` work as the merge?

[Live demo] `cargo run -p gossip-toy`, then `cargo run -p gossip-toy --bin tcp`

Async or Threads?

Workload	Use
Many concurrent waits : network, timers, users	async (tokio)
CPU-bound data parallelism	threads, <code>thread::scope</code> , <code>chunks_mut</code> (HW3)
Both	async for coordination; <code>spawn_blocking</code> or a thread pool for heavy compute

- Async isn't faster at computing, only at **waiting for many things**
- If your program never waits on I/O, you probably don't need async

Summary

- A future is a value you `poll`; `Pending` comes with a promise to `wake`
- An executor is a loop; a reactor turns OS events into wakes
- `async fn` compiles to an **enum state machine** that holds exactly the locals live across each `.await`
- `Pin` is an encapsulated-`unsafe` invariant: pinned things don't move
- `spawn` needs `Send + 'static` for the same reasons as `thread::spawn`
- `select!` waits for the first of several events; **cancellation is** `drop`
- **Actors** keep state and its invariants in one task

Next Time, and HW4

- HW3 is due Tuesday. HW4 is released Tuesday and due Oct 20
- HW4: a local-first replicated store
 - Part 3 is a `select!`-loop node like today's, syncing replicas over framed TCP
- **Tuesday:** when replicas **disagree** (two people edit the same thing at once), what should happen?
 - Eventual consistency and **CRDTs**: merges that always converge
- **Thursday:** collaborative text, and how to test all of this deterministically

Demo code: [26fa/lectures/09-async/demos/](#) (see its README)