Async in Rust

Say you want to wait for an event…

… without blocking other tasks.

Example of events:

- connection to a database
- response to your TCP packet
- a push on a button

Solution 1: threads (bad)

```
std::thread::spawn(|| {
    // run the GET request
    let handler = get(URL);

    // wait until it comes back
    loop {
        std::thread::sleep(std::time::Duration::from_millis(10));
        if handler.is_ready() {
            break;
        }
    }

    // look at the result
    let request = handler.get_request().expect("request was not ready");

    // ...
}
```

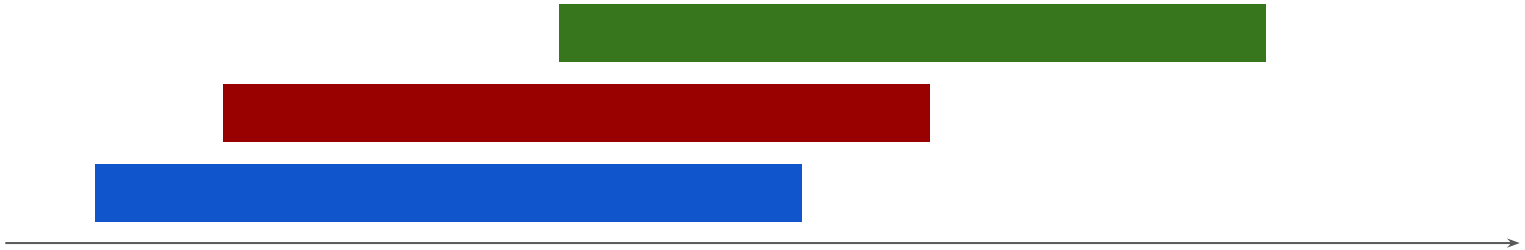
Solution 2: async (good)

```
// web request
let res = reqwest::get("http://httpbin.org/get").await;

// SQL connection
let conn = SqliteConnection::connect("sqlite::memory:").await?;

// wait for button
async_input.wait_for_high().await;
```

Async vs Threads



Async vs Threads

Threads
Heavy
Suited to CPU bounds
Work in parallel
Preemptive multitasking

Async vs Threads

Threads	Async
Heavy	Lightweight
Suited to CPU bounds	Suited to IO bounds
Work in parallel	Jumps between tasks
Preemptive multitasking	Cooperative multitasking

Example with sqlx

```
use sqlx::{FromRow, SqlitePool};

#[derive(Debug, FromRow)]
struct User {
    id: i64,
    name: String,
    password: String,
    is_admin: bool,
}
```

Example with sqlx

```
#[async_std::main]
async fn main() -> Result<(), sqlx::Error> {
    // Connect to or create the SQLite database
    let pool = SqlitePool::connect("sqlite::memory:").await?;
    let sql_script = include_str!(concat!("../data/build_database.sql"));
    sqlx::query(sql_script).execute(&pool).await?;
```

Example with sqlx

```rust
#[async_std::main]
async fn main() -> Result<(), sqlx::Error> {
    // Connect to or create the SQLite database
    let pool = SqlitePool::connect("sqlite::memory:").await?;
    let sql_script = include_str!(concat!("../data/build_database.sql"));
    sqlx::query(sql_script).execute(&pool).await?;

    // Insert a user
    let name = "alice";
    let password = "secret_hash";
    let is_admin = true;

    sqlx::query!(
        "INSERT INTO USER (name, password, is_admin) VALUES (?, ?, ?)",
        name,
        password,
        is_admin
    )
    .execute(&pool)
    .await?;
```

Example with sqlx

```rust
#[async_std::main]
async fn main() -> Result<(), sqlx::Error> {
    // Connect to or create the SQLite database
    let pool = SqlitePool::connect("sqlite::memory:").await?;
    let sql_script = include_str!(concat!("../data/build_database.sql"));
    sqlx::query(sql_script).execute(&pool).await?;

    // Insert a user
    let name = "alice";
    let password = "secret_hash";
    let is_admin = true;

    sqlx::query!(
        "INSERT INTO USER (name, password, is_admin) VALUES (?, ?, ?)",
        name,
        password,
        is_admin
    )
    .execute(&pool)
    .await?;
```

```rust
    // Query users
    let users: Vec<User> = sqlx::query_as::<_, User>(
          "SELECT * FROM USER"
       )
       .fetch_all(&pool)
       .await?;

    println!("{:?}", users);

    Ok(())
}
```

Example with sqlx

```rust
#[async_std::main]
async fn main() -> Result<(), sqlx::Error> {
    // Connect to or create the SQLite database
    let pool = SqlitePool::connect("sqlite::memory:").await?;
    let sql_script = include_str!(concat!("../data/build_database.sql"));
    sqlx::query(sql_script).execute(&pool).await?;

    // Insert a user
    let name = "alice";
    let password = "secret_hash";
    let is_admin = true;

    sqlx::query!(
        "INSERT INTO USER (name, password, is_admin) VALUES (?, ?, ?)",
        name,
        password,
        is_admin
    )
    .execute(&pool)
    .await?;

    // Query users
    let users: Vec<User> = sqlx::query_as::<_, User>(
        "SELECT * FROM USER"
    )
    .fetch_all(&pool)
    .await?;

    println!("{:?}", users);

    Ok(())
}
```

Note to self: show some compiler errors

Example with embassy on a chip 1/2

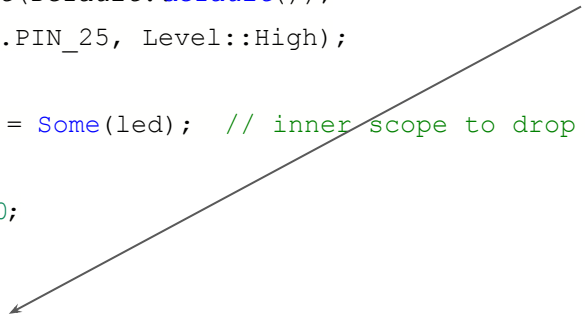
```
#[embassy_executor::task(pool_size =2)]
async fn toggle_led(led: &'static LedType, delay: Duration) {
    let mut ticker = Ticker::every(delay);
    loop {
        {
            let mut led_unlocked = led.lock().await;
            if let Some(pin_ref) = led_unlocked.as_mut() {
                pin_ref.toggle();
            }
        }
        ticker.next().await;
    }
}
```

Example with embassy on a chip 2/2

```rust
#[embassy_executor::main]
async fn main(spawner: Spawner) {
    let p = embassy_rp::init(Default::default());
    let led = Output::new(p.PIN_25, Level::High);
    {
        *(LED.lock().await) = Some(led);  // inner scope to drop the mutex
    }
    let dt = 100 * 1_000_000;
    let k = 1.003;

    unwrap!(spawner.spawn(toggle_led(&LED, Duration::from_nanos(dt))));
    unwrap!(spawner.spawn(toggle_led(&LED, Duration::from_nanos((dt as f64 * k) as u64))));
}
```

multitasking without an OS!



Rules of thumb of async

- do not wait too much across await points
  - do not use the “usual” sleep !
  - do not run long tasks !
- use async locks
  - “usual” locks may deadlock in async context

Why not using synchronous locks

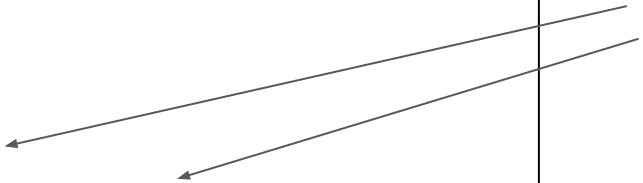
```
use tokio::time::{delay_for, Duration};
use std::sync::Mutex;

async fn work(mutex: &Mutex<i32>) {
    let mut v = mutex.lock().unwrap();
    delay_for(Duration::from_millis(100)).await;
    *v += 1;
}

#[tokio::main]
async fn main() {
    let mutex = Mutex::new(0);

    tokio::join!(work(&mutex), work(&mutex));

    println!("{}", *mtx.lock().unwrap());
}
```



Look, a deadlock!

Side quest 1: how did the compiler know about my table?

(time to show some build scripts)

(more info about build scripts next week)

Side quest 2: how is a future able to reference data inside itself?

(let’s talk about Pin (https://doc.rust-lang.org/std/pin/))

A task has to store its state when it is awaited. It may lead to using self references.

But what happens when you move the structure? (these references break)

Hence, you must pin these self-referential structs in memory.