



# Async Rust, a fondo

Escribamos un async runtime desde cero

por Jorge Prendes

## Notes:

Hola a todos, mi nombre es Jorge, y hoy vengo a hablarles sobre async Rust.

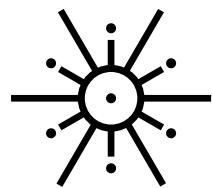
El objetivo de esta presentación es entender como funciona async/await en Rust.

Usar async/await es un poco mágico, esparcimos un poco de async y un poco de await por acá y por allá, y funciona.

Pero a final de cuentas son solo transformaciones que hace el compilador, no hay nada mágico.

Para poder entender bien qué hacen tokio y el compilador cuando usamos async/await, vamos a escribir un runtime async desde cero.

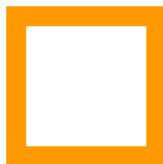
# Desde cero? Para qué???



Tokio



Smol



Embassy



## Notes:

En el ecosistema de Rust hay varios runtimes async, algunos de los más conocidos son tokio, smol, y embassy.

Cada uno tiene sus particularidades, por ejemplo:

- Tokio es el más completo, tiene muchas características y es el más usado, pero también es bastante opinionado.
- Smol es menos opinionado, y está diseñado para que juegue bien con otros runtimes async.
- Embassy está diseñado para ser usado en entornos embebidos, como microcontroladores.

Si me preguntan: ¿Debería escribir mi propio runtime async, o usar uno existente?

La respuesta está clara: Usá uno existente, salvo que tengas una razón muy específica para no hacerlo.

Entonces, para qué vamos a escribir un runtime async desde cero?



# Hyperlight



## Notes:

Bueno, **mi** motivación es que estoy trabajando en un proyecto llamado [hyperlight](#).

Hyperlight es una librería que te permite crear micro-VMs muy rápido (en milisegundos).

El compromiso, es que las micro-VMs son muy limitadas, no tienen sistema operativo, el código corre básicamente bare-metal.

Eso significa que no podemos usar ni tokio ni smol, porque ambos usan el sistema operativo.

Embassy sería una opción, pero Embassy está diseñado para microcontroladores, mientras que en nuestro caso no es un microcontrolador, no tenemos las mismas limitaciones de memoria, CPU, etc.

# Por la ciencia!



mini-tokio



## Notes:

La otra razón es por la ciencia.

Para aprender, ya que entendiendo como funcionan las cosas que usamos podemos hacer mejor uso de ellas.

# Capítulo I

## La magia `async fn`

---

### Notes:

Como mencioné antes, usar `async/await` es un poco mágico, esparcimos un poco de `async` y un poco de `await` por acá y por allá, y funciona.

# De fn a async fn

```
1 fn saludar(nombre: String) -> String {  
2     format!("hola {}!")  
3 }  
4  
5 fn main() {  
6     let res =  
7         saludar("Rust en español".into());  
8     println!("{}");  
9 }
```



## Notes:

Empezamos con un código muy simple

- una función `saludar` que recibe un nombre y devuelve un saludo.
- y una función `main` que llama a `saludar` y muestra el resultado.

# De fn a async fn

```
1  async fn saludar(nombre: String) -> String {  
2      format!("hola {}!")  
3  }  
4  
5  fn main() {  
6      let res =  
7          saludar("Rust en español".into());  
8      println!("{}");  
9  }
```



## Notes:

Luego, convertimos `saludar` en una función `async fn`.

# De fn a async fn

```
1  async fn saludar(nombre: String) -> String {  
2      format!("hola {}!")  
3  }  
4  
5  fn main() {  
6      let res =  
7          saludar("Rust en español".into()).await;  
8      println!("{}");  
9  }
```



## Notes:

Para poder obtener el resultado de `saludar` en `main`, tenemos que usar `await`.

# De fn a async fn

```
1  async fn saludar(nombre: String) -> String {  
2      format!("hola {}!")  
3  }  
4  
5  async fn main() {  
6      let res =  
7          saludar("Rust en español".into()).await;  
8      println!("{}");  
9  }
```



## Notes:

Pero para poder usar `await` en `main`, tenemos que convertir `main` en una función `async fn`.

`await` solo puede usarse dentro de funciones `async fn`.

# De fn a async fn

```
1  async fn saludar(nombre: String) -> String {  
2      format!("hola {}!")  
3  }  
4  
5  #[tokio::main]  
6  async fn main() {  
7      let res =  
8          saludar("Rust en español".into()).await;  
9      println!("{}");  
10 }
```



## Notes:

Pero Rust no sabe cómo ejecutar funciones `async fn` por sí mismo, para eso necesita ayuda de un runtime async.

Así que usamos la macro `#[tokio::main]` para convertir ese async `main` en un `main` normal. La macro va a transformar el `main` async en un `main` normal que arranca el runtime de tokio y ejecuta el `main` async dentro del runtime.

Y esto funciona, es súper simple, pero... ¿qué está pasando realmente? todo esto es un poco mágico.

Esto también nos lleva a otro problema, que es lo que se suele llamar **coloring**.

Para poder convertir `saludar` en async tuvimos que convertir todo el stack que llama a `saludar` en async también.

Esto significa que código Rust que **sí** sea async es básicamente incompatible con código Rust que **no** sea async.

Hay soluciones, por ejemplo tokio provee varias utilidades para hacer de puente entre los dos mundos.

Pero no es ideal, especialmente comparado con lenguajes como Go, donde no hay distinción entre código async y código normal.

Como es de esperarse, hay ventajas y desventajas en ambos enfoques, pero eso es una discusión para otro día.

# async fn a dieta

```
1 async fn saludar(nombre: String) -> String {  
2     format!("hola {}!")  
3 }
```



## Notes:

Todo esto de `async` es lo que se suele llamar "sugar syntax", es decir, azúcar sintáctica.

Son cosas del lenguaje que mejoran (muchísimo) la experiencia de escribir código, la ergonomía, mantenibilidad, pero que no son estrictamente necesarias.

Técnicamente podemos obtener el mismo resultado sin usar `async/await`.

Podrían argumentar que Rust es azúcar sintáctica de assembly, pero eso es una discusión para otro día.

Para entender que hace el compilador cuando escribimos `'async fn'`, vamos a reescribir `'saludar'` sin usar el azúcar sintáctico, vamos a poner nuestro `'async fn'` a dieta.

La función `'saludar'` que tenemos acá no es una función mágica, es una función normal.

Pero si la llamamos, no devuelve un `'String'` como nos promete acá.

# async fn a dieta

```
1 use core::future::Future;
2
3 fn saludar(
4     nombre: String
5 ) -> impl Future<Output = String> {
6     format!("hola {}!")  
7 }
```



## Notes:

Lo que hace el compilador es transformar la función para devolver un objeto anónimo que implementa el trait `Future`.

El problema ahora es que `String` no implementa `Future`.

Pero hay una solución fácil.

# async fn a dieta

```
1 use core::future::Future;
2
3 fn saludar(
4     nombre: String
5 ) -> impl Future<Output = String> {
6     async move {
7         format!("hola {}!")  
8     }
9 }
```



## Notes:

Usamos lo que se llama un bloque `async` .

Lo que hace un bloque async es crear una estructura anónima que implementa `Future` , y que ejecuta el código dentro del bloque async.

Esto funciona, pero... sigue siendo bastante mágico!

# async {} a dieta

```
1 use core::future::Future;
2
3 fn saludar(
4     nombre: String
5 ) -> impl Future<Output = String> {
6     async move {
7         format!("hola {}!")  
8     }
9 }
```



## Notes:

Por suerte, el bloque `async` también es azúcar sintáctica, así que también lo podemos poner a dieta!

Como dijimos antes, el bloque `async` se traduce a una estructura que implementa `Future`.

# async {} a dieta

```
1 use core::future::Future;
2
3 fn saludar(
4     nombre: String
5 ) -> impl Future<Output = String> {
6     async move {
7         format!("hola {}", nombre)
8     }
9 }
10
11 struct Saludar(String);
```



## Notes:

Así que el primer paso es crear una estructura para representar el bloque `async`.

Creamos una estructura `Saludar` que contiene el nombre al que debemos saludar.

# async {} a dieta

```
1 use core::future::Future;
2
3 fn saludar(
4     nombre: String
5 ) -> impl Future<Output = String> {
6     Saludar(nombre)
7 }
8
9 struct Saludar(String);
```



## Notes:

Y devolvemos esa estructura en lugar del bloque async.

# async {} a dieta

```
1 use core::future::Future;
2
3 fn saludar(
4     nombre: String
5 ) -> impl Future<Output = String> {
6     Saludar(nombre)
7 }
8
9 struct Saludar(String);
10
11 impl Future for Saludar {
12 }
```



## Notes:

Pero ahora nos comprometimos a que `Saludar` implementara `Future`, así que hagamos eso.

# async {} a dieta

```
1 use core::future::Future;
2
3 fn saludar(
4     nombre: String
5 ) -> impl Future<Output = String> {
6     Saludar(nombre)
7 }
8
9 struct Saludar(String);
10
11 impl Future for Saludar {
12     type Output = String;
13 }
```



## Notes:

El trait `Future` tiene un tipo asociado `Output`, que es el tipo que se devuelve cuando hacemos `await`.

En este caso, `Output` es `String`, porque `saludar` devuelve un `String`.

# async {} a dieta

```
1 use core::future::Future;
2 use core::pin::Pin;
3 use core::task::{Context, Poll};
4
5 fn saludar(
6     nombre: String
7 ) -> impl Future<Output = String> {
8     Saludar(nombre)
9 }
10
11 struct Saludar(String);
12
13 impl Future for Saludar {
14     type Output = String;
15
16     fn poll(
17         self: Pin<&mut Self>,
18         _ctx: &mut Context<'_>
19     ) -> Poll<String> {
20     }
21 }
```



## Notes:

Y por último, nos queda implementar el único método del trait `Future`, que es `poll`.

Este es el método que se llama para avanzar la ejecución del futuro.

El futuro por si solo no hace nada, tenemos que llamar `poll` para que haga algo.

# async {} a dieta

```
1 use core::future::Future;
2 use core::pin::Pin;
3 use core::task::{Context, Poll};
4
5 fn saludar(
6     nombre: String
7 ) -> impl Future<Output = String> {
8     Saludar(nombre)
9 }
10
11 struct Saludar(String);
12
13 impl Future for Saludar {
14     type Output = String;
15
16     fn poll(
17         self: Pin<&mut Self>,
18         _ctx: &mut Context<'_>
19     ) -> Poll<String> {
20         let nombre = &self.0;
21         Poll::Ready(format!("hola {}!"))
22     }
23 }
```



## Notes:

En este caso, `poll` simplemente devuelve `Poll::Ready` con el saludo.

No se preocupen mucho por los detalles de poll, ya los vamos a ver bien en detalle en un ratito.

Y esto... básicamente funciona!

El código compila, y tokio es capaz de ejecutar este futuro sin problemas.

## main a dieta

```
1 #[tokio::main]
2 async fn main() {
3     let res =
4         saludar("Rust en español".into()).await;
5     println!("{}",&res);
6 }
```



### Notes:

Muy bien, `saludar` ya lo tenemos a dieta.

Ahora vamos a hacer lo mismo con `main`, que por ahora sigue siendo muy mágico.

# main a dieta

```
1 async fn main() {  
2     let res =  
3         saludar("Rust en español".into()).await;  
4     println!("{}",&res);  
5 }
```



## Notes:

Lo primero que hacemos es sacar la macro `#[tokio::main]`.

## main a dieta

```
1 fn main() {  
2     let res =  
3         saludar("Rust en español".into()).await;  
4     println!("{}:?", res);  
5 }
```



### Notes:

Con lo cual también tenemos que sacar el `async` de `main`.

# main a dieta

```
1 fn main() {  
2     let res =  
3         saludar("Rust en español".into());  
4     println!("{}",&res);  
5 }
```



## Notes:

Con lo cual, ya no podemos usar `await`.

# main a dieta

```
1 fn main() {  
2     let fut = saludar("Rust en español".into());  
3     println!("{}{:?}");  
4 }
```



## Notes:

Esto implica que `res` ya no es el resultado de `saludar`, sino que es el futuro que devuelve `saludar`. Así que lo renombramos.

# main a dieta

```
1 use core::future::Future;
2
3 fn main() {
4     let fut = saludar("Rust en español".into());
5
6     let res = fut.poll(&mut ctx);
7
8     println!("{}:?", res);
9 }
```



## Notes:

Ahora tenemos un problema, cómo obtenemos nuestro string?

Vamos a tener que llamar a `poll` para avanzar la ejecución del futuro.

# main a dieta

```
1 use core::future::Future;
2 use core::task::Poll;
3
4 fn main() {
5     let fut = saludar("Rust en español".into());
6
7     let res = match fut.poll(&mut ctx) {
8         Poll::Ready(ret) => ret,
9         _ => todo!(),
10    };
11
12    println!("{}{:?}", res);
13 }
```



## Notes:

Recordemos que el método `poll` devuelve un tipo `Poll`.

Este tipo `Poll` es un enum, en nuestro caso estamos interesados en la variante `Poll::Ready`, que es la que contiene nuestropreciado string.

# main a dieta

```
1 use core::future::Future;
2 use core::task::Poll;
3
4 fn main() {
5     let fut = saludar("Rust en español".into());
6
7     let res = match fut.poll(&mut ctx) {
8         Poll::Ready(ret) => ret,
9         Poll::Pending => todo!(),
10    };
11
12    println!("{}{:?}", res);
13 }
```



## Notes:

La otra variante de `Poll` es `Poll::Pending`, que indica que el futuro no está listo todavía.

`Poll::Ready` indica que la ejecución del futuro terminó y tenemos el resultado.

`Poll::Pending` indica que el futuro no terminó todavía, y vamos a tener que llamar `poll` más tarde para que el futuro pueda seguir avanzando.

Imagínense por ejemplo, que estamos recibiendo datos de la red, llamamos `poll`, nuestro futuro procesa los datos que llegaron, pero no llegaron todos los datos todavía, así que devuelve `Poll::Pending`, y vamos a tener que volver a llamarlo mas tarde para que pueda seguir procesando el resto de los datos que faltan.

# main a dieta

```
1 use core::future::Future;
2 use core::task::Poll;
3
4 fn main() {
5     let fut = saludar("Rust en español".into());
6
7     let res = loop {
8         match fut.poll(&mut ctx) {
9             Poll::Ready(ret) => ret,
10            Poll::Pending => todo!(),
11        }
12    };
13
14    println!("{}{:?}", res);
15 }
```



## Notes:

La clave está en que vamos a tener que llamar `poll` repetidamente hasta que el futuro devuelva `Poll::Ready`. Así que lo metemos todo en un loop.

# main a dieta

```
1 use core::future::Future;
2 use core::task::Poll;
3
4 fn main() {
5     let fut = saludar("Rust en español".into());
6
7     let res = loop {
8         match fut.poll(&mut ctx) {
9             Poll::Ready(ret) => ret,
10            Poll::Pending => continue,
11        }
12    };
13
14    println!("{}{:?}", res);
15 }
```



## Notes:

Si `poll` devuelve `Pending`, simplemente seguimos el loop y volvemos a llamar `poll`.

# main a dieta

```
1 use core::future::Future;
2 use core::task::Poll;
3
4 fn main() {
5     let fut = saludar("Rust en español".into());
6
7     let res = loop {
8         match fut.poll(&mut ctx) {
9             Poll::Ready(ret) => break ret,
10            Poll::Pending => continue,
11        }
12    };
13
14    println!("{}{:?}", res);
15 }
```



## Notes:

Cuando `poll` devuelve `Ready`, quiere decir que terminamos, así que rompemos el loop y devolvemos el resultado.

# main a dieta

```
1 use core::future::Future;
2 use core::task::{Context, Poll, Waker};
3
4 fn main() {
5     let fut = saludar("Rust en español".into());
6
7     let mut ctx =
8         Context::from_waker(Waker::noop());
9
10    let res = loop {
11        match fut.poll(&mut ctx) {
12            Poll::Ready(ret) => break res,
13            Poll::Pending => continue,
14        }
15    };
16
17    println!("{}:?", res);
18 }
```



## Notes:

Ok, ok, todo muy lindo, pero esto no compila.

Si se acuerdan de la *signature* del método `poll`, que les dije que se preocupen todavía, ahora es el momento de prestarle atención.

El método `poll` recibe un parámetro, que es un contexto.

Un contexto es un objeto que contiene información sobre el entorno en el que se está ejecutando el futuro.

Hoy en día el tipo `Context` es básicamente un *wrapper* alrededor de un objeto Waker. Tiene 2 funciones:

- Un constructor `from\_waker`, que crea un contexto a partir de un waker.
- Un método `waker`, que devuelve una referencia al waker asociado al contexto.

Así que sí, básicamente un paquete bonito envolviendo un waker.

El waker va a ser el mecanismo que vamos a usar para que el futuro nos avise cuando pueda seguir avanzando.

Pero por ahora no se preocupen demasiado, vamos a verlo en detalle más adelante.

Lo importante ahora es que podemos crear un waker "tonto" que no hace nada con `Waker::noop` de la librería estándar.

Y podemos usar ese waker para crear el contexto que necesitamos para llamar a `poll`.

# main a dieta

```
1 use core::future::Future;
2 use core::pin::pin;
3 use core::task::{Context, Poll, Waker};
4
5 fn main() {
6     let fut = saludar("Rust en español".into());
7
8     let mut ctx =
9         Context::from_waker(Waker::noop());
10    let mut fut = pin!(fut);
11
12    let res = loop {
13        match fut.as_mut().poll(&mut ctx) {
14            Poll::Ready(ret) => break res,
15            Poll::Pending => continue,
16        }
17    };
18
19    println!("{}:?", res);
20 }
```



## Notes:

Muy bien, ya compila? No!

El método `poll` recibe `self` como un `Pin<&mut Self>`, y nosotros estamos intentando llamar a `poll` en un `&Self`.

Las razones por las que `poll` recibe `self` como un `Pin<&mut Self>` son un poco complejas, pero básicamente es para asegurarse de que el futuro no se mueva en memoria mientras está siendo ejecutado.

A nosotros nos afecta en que tenemos que "pinnear" nuestro futuro antes de poder llamar a `poll`.

Y eso lo podemos hacer con la macro `pin!` de la librería estándar.

# main a dieta

```
1 use core::future::Future;
2 use core::pin::pin;
3 use core::task::{Context, Poll, Waker};
4
5 fn main() {
6     let fut = saludar("Rust en español".into());
7
8     let mut ctx =
9         Context::from_waker(Waker::noop());
10    let mut fut = pin!(fut);
11
12    let res = loop {
13        match fut.as_mut().poll(&mut ctx) {
14            Poll::Ready(ret) => break ret,
15            Poll::Pending => continue,
16        }
17    };
18
19    println!("{}{:?}{}", "res:", res);
20 }
```



## Notes:

Ok, ahora si, esto compila, y lo podemos usar para ejecutar nuestro futuro. 🎉

Nos podemos dar una palmadita en la espalda.

Y nada de esto es mágico! es solo código Rust normal y corriente, nada de async/await.

# Capítulo II

## El trait Future

---

### Notes:

Ok, ya sabemos que un `async fn` es una función normal que devuelve un objeto que implementa el trait `Future`. Incluso implementamos una función `saludar` que devuelve un futuro, y una función `main` que ejecuta ese futuro. Pero ese futuro es muy simple, ni bien lo llamamos ya está listo, no hace nada interesante. Vamos a hacer un futuro un poco más interesante, y aprender un poco más sobre el trait `Future` en el camino.

# Durmiendo

```
1 use std::time::Duration;
2
3 async fn dormir(t: Duration) {
4     todo!()
5 }
```



## Notes:

Ok, vamos a hacer un futuro que duerma por un tiempo determinado.

Tenemos una función `dormir`, que recibe una duración de cuánto tiempo dormir.

# Durmiendo

```
1 use std::time::{Duration, Instant};  
2  
3 struct Dormir {  
4     hora: Instant  
5 }  
6 async fn dormir(t: Duration) {  
7     todo!()  
8 }
```



## Notes:

`dormir` en `async`, pero vamos a reescribirla como una función normal.

Para eso, primero creamos una estructura `Dormir` que va a set nuestro futuro.

# Durmiendo

```
1 use std::time::{Duration, Instant};  
2  
3 struct Dormir {  
4     hora: Instant  
5 }  
6 fn dormir(t: Duration) -> Dormir {  
7     todo!()  
8 }
```



---

## Notes:

Y hacemos que `dormir` devuelva un `Dormir`.

# Durmiendo

```
1 use std::time::{Duration, Instant};  
2  
3 struct Dormir {  
4     hora: Instant  
5 }  
6 fn dormir(t: Duration) -> Dormir {  
7     let hora = Instant::now() + t;  
8     Dormir { hora }  
9 }
```



## Notes:

La estructura `Dormir` necesita saber hasta qué hora tiene que dormir, así que calculamos la hora actual más la duración, y la guardamos en `hora`.

Y ahora, nuestro contrato, `Dormir` tiene que implementar el trait `Future`.

# Durmiendo

```
1 impl Future for Dormir {  
2 }
```



---

## Notes:

Ok, implementemos el trait `Future` para `Dormir`.

# Durmiendo

```
1 impl Future for Dormir {  
2     type Output = ();  
3 }
```



## Notes:

El tipo asociado `Output` es `()` , porque `dormir` cuando era una función async no devolvía nada.

# Durmiendo

```
1 impl Future for Dormir {
2     type Output = ();
3
4     fn poll(
5         self: Pin<&mut Self>,
6         _ctx: &mut Context
7     ) -> Poll<()> {
8         }
9     }
```



## Notes:

Ahora lo interesante, el método `poll`.

# Durmiendo

```
1 impl Future for Dormir {
2     type Output = ();
3
4     fn poll(
5         self: Pin<&mut Self>,
6         _ctx: &mut Context
7     ) -> Poll<()> {
8         let hora = Instant::now();
9
10        if self.hora < hora {
11            todo!()
12        } else {
13            todo!()
14        }
15    }
16 }
```



## Notes:

Lo primero que tenemos que hacer, es ver si ya es hora de despertar.

Obtenemos la hora actual, y la comparamos con la hora de despertarse.

# Durmiendo

```
1 impl Future for Dormir {
2     type Output = ();
3
4     fn poll(
5         self: Pin<&mut Self>,
6         _ctx: &mut Context
7     ) -> Poll<()> {
8         let hora = Instant::now();
9
10        if self.hora < hora {
11            Poll::Ready(())
12        } else {
13            todo!()
14        }
15    }
16 }
```



## Notes:

Si ya es hora de despertarse, devolvemos `Poll::Ready(())`, indicando que el futuro terminó de ejecutarse.

# Durmiendo

```
1 impl Future for Dormir {
2     type Output = ();
3
4     fn poll(
5         self: Pin<&mut Self>,
6         _ctx: &mut Context
7     ) -> Poll<()> {
8         let hora = Instant::now();
9
10        if self.hora < hora {
11            Poll::Ready(())
12        } else {
13            Poll::Pending
14        }
15    }
16 }
```



## Notes:

Y ahora la parte interesante.

Si todavía no es hora de despertarse, devolvemos `Poll::Pending`, indicando que el futuro no terminó todavía.

Pero...

# Durmiendo

```
1 impl Future for Dormir {
2     type Output = ();
3
4     fn poll(
5         self: Pin<&mut Self>,
6         ctx: &mut Context
7     ) -> Poll<()> {
8         let hora = Instant::now();
9
10        if self.hora < hora {
11            Poll::Ready(())
12        } else {
13            ctx.waker().wake_by_ref();
14            Poll::Pending
15        }
16    }
17 }
```



## Notes:

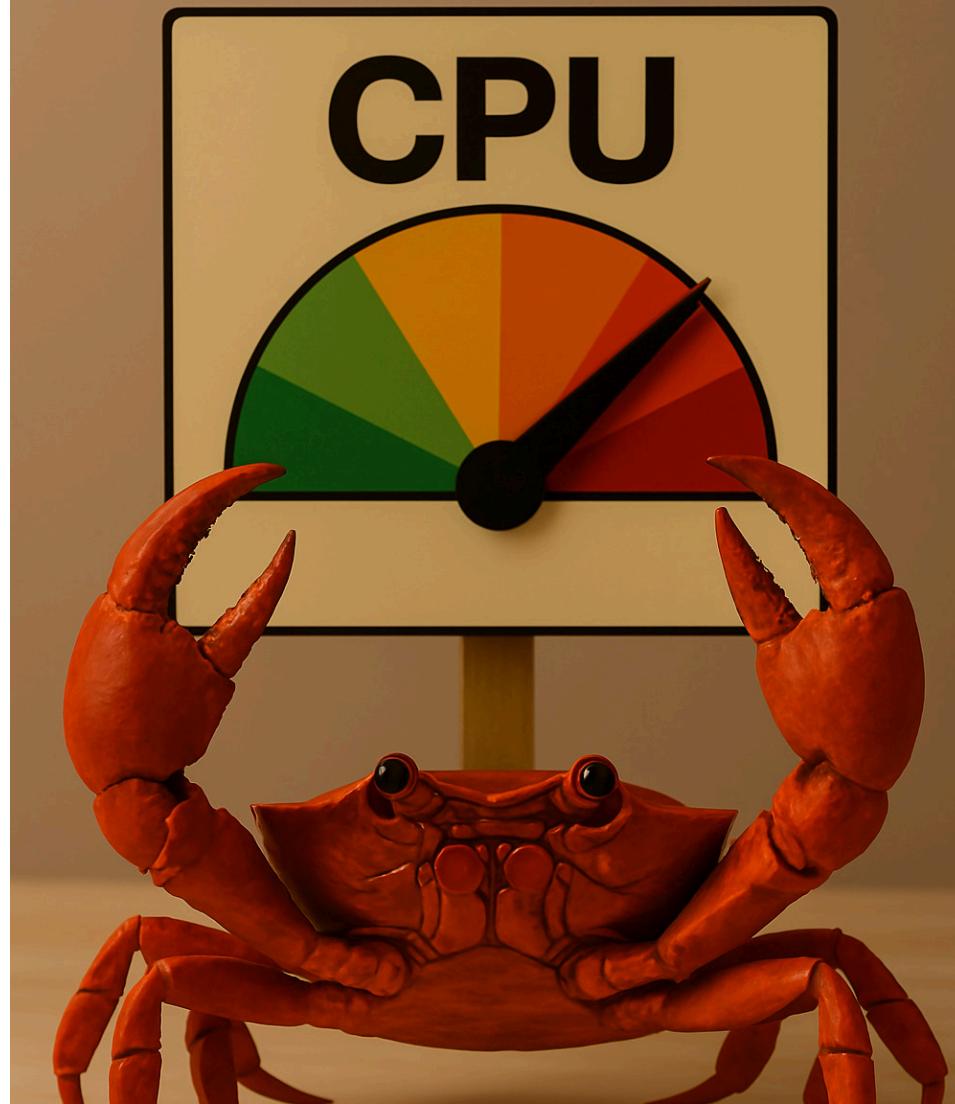
Hay un contrato隐式 en el trait `Future` que tenemos que cumplir.

Cuando un futuro devuelve `Poll::Pending`, tiene que asegurarse que alguien en algún momento, va a usar el waker para indicar que el futuro puede seguir avanzando.

Nosotros en este caso hacemos algo muy simple, simplemente llamamos a `wake\_by\_ref` en el waker antes de devolver `Poll::Pending`.

# Durmiendo

```
1 impl Future for Dormir {
2     type Output = ();
3
4     fn poll(
5         self: Pin<&mut Self>,
6         ctx: &mut Context
7     ) -> Poll<()> {
8         let hora = Instant::now();
9
10        if self.hora < hora {
11            Poll::Ready(())
12        } else {
13            ctx.waker().wake_by_ref();
14            Poll::Pending
15        }
16    }
17 }
```

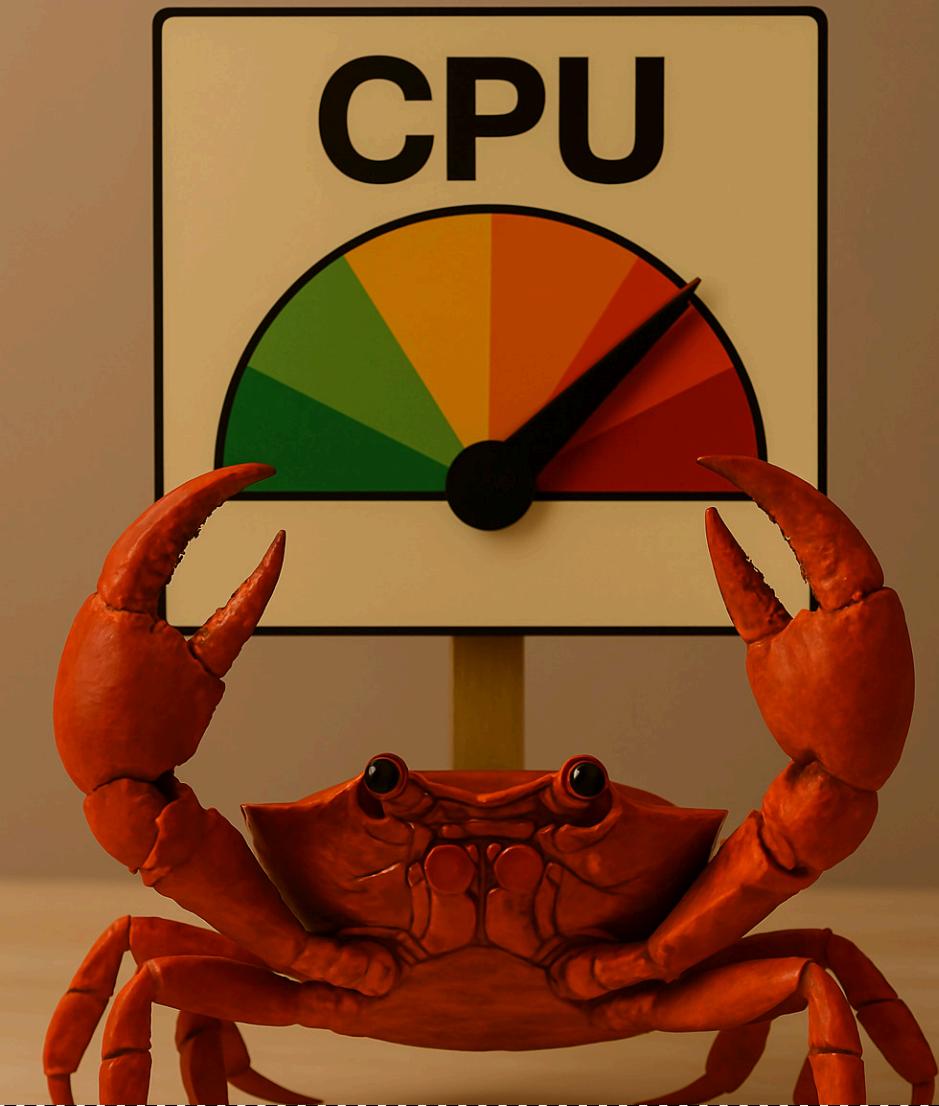
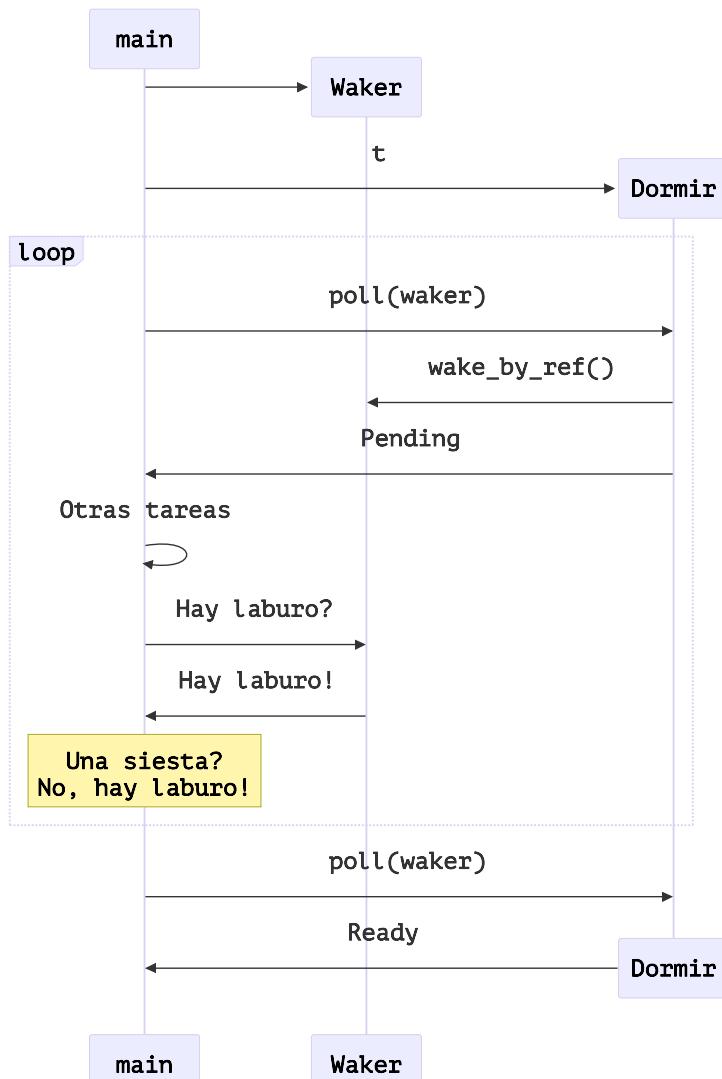


## Notes:

Esta implementación, *teóricamente* funciona.

Pero tiene un problema importante, y es que está quemando ciclos de CPU inútilmente.

Para entender por qué, veamos un diagrama de secuencia de lo que está pasando.



## Notes:

1. `main` crea un `'Waker'` para poder llamar a `'poll'`.
2. `main` también crea `'Dormir'` (por medio de la función `'dormir'`) con el tiempo que tiene que dormir.
3. Luego `main` llama a `'poll'` en `'Dormir'` pasandole el `'waker'`.
4. `'Dormir'` chequea la hora, ve que todavía no es hora de despertarse, así que llama a `'wake_by_ref'` en el waker, y devuelve `'Poll::Pending'`.
5. En este momento, `main` puede hacer otras tareas, pero `'Dormir'` (porque devolvió `'Pending'`) todavía no terminó, así que `main` tiene que volver a llamar en algún momento.
6. Para decidir cuándo llamar a `'poll'` de nuevo, `main` le pregunta al waker si `'Dormir'` tiene laburo para hacer, es decir, si el futuro puede seguir avanzando.
7. Cuando `'Dormir'` llamó a `'wake_by_ref'`, eso le indica al `'Waker'` que `'Dormir'` tiene laburo para hacer, así que inmediatamente le dice a `main` que sí, que hay laburo.
8. Si no hubiese laburo, acá `main` se podría hacer una siesta hasta que haya nuevo laburo. Pero como si hay laburo para hacer, `main` vuelve a llamar a `'poll'` en `'Dormir'` inmediatamente.

Y esto se repite una y otra vez, hasta que finalmente es hora de despertarse.

Una vez que hayamos quemado suficientes ciclos de CPU, finalmente es hora de despertarse, `main` llama a `'poll'` una última vez, y esta vez `'Dormir'` devuelve `'Poll::Ready()'`.

# Durmiendo

```
1 impl Future for Dormir {
2     type Output = ();
3
4     fn poll(
5         self: Pin<&mut Self>,
6         ctx: &mut Context
7     ) -> Poll<()> {
8         let hora = Instant::now();
9
10        if self.hora < hora {
11            Poll::Ready(())
12        } else {
13            ctx.waker().wake_by_ref();
14            Poll::Pending
15        }
16    }
17 }
```



## Notes:

La solución a este problema es bastante simple.

# Durmiendo

```
1 impl Future for Dormir {
2     type Output = ();
3
4     fn poll(
5         self: Pin<&mut Self>,
6         ctx: &mut Context
7     ) -> Poll<()> {
8         let hora = Instant::now();
9
10        if self.hora < hora {
11            Poll::Ready(())
12        } else {
13            let waker = ctx.waker().clone();
14            let t = self.hora - hora;
15            std::thread::spawn(move || {
16                std::thread::sleep(t);
17                waker.wake_by_ref();
18            });
19            Poll::Pending
20        }
21    }
22 }
```

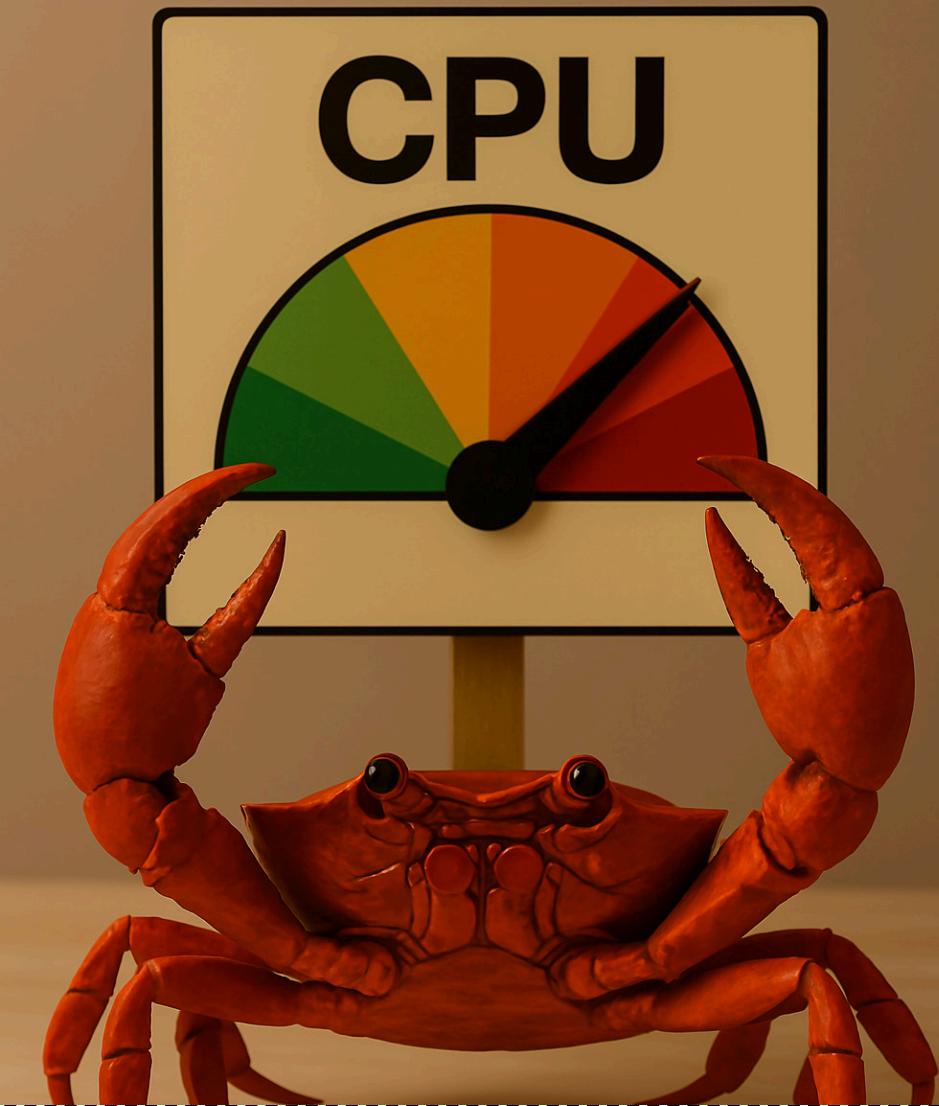
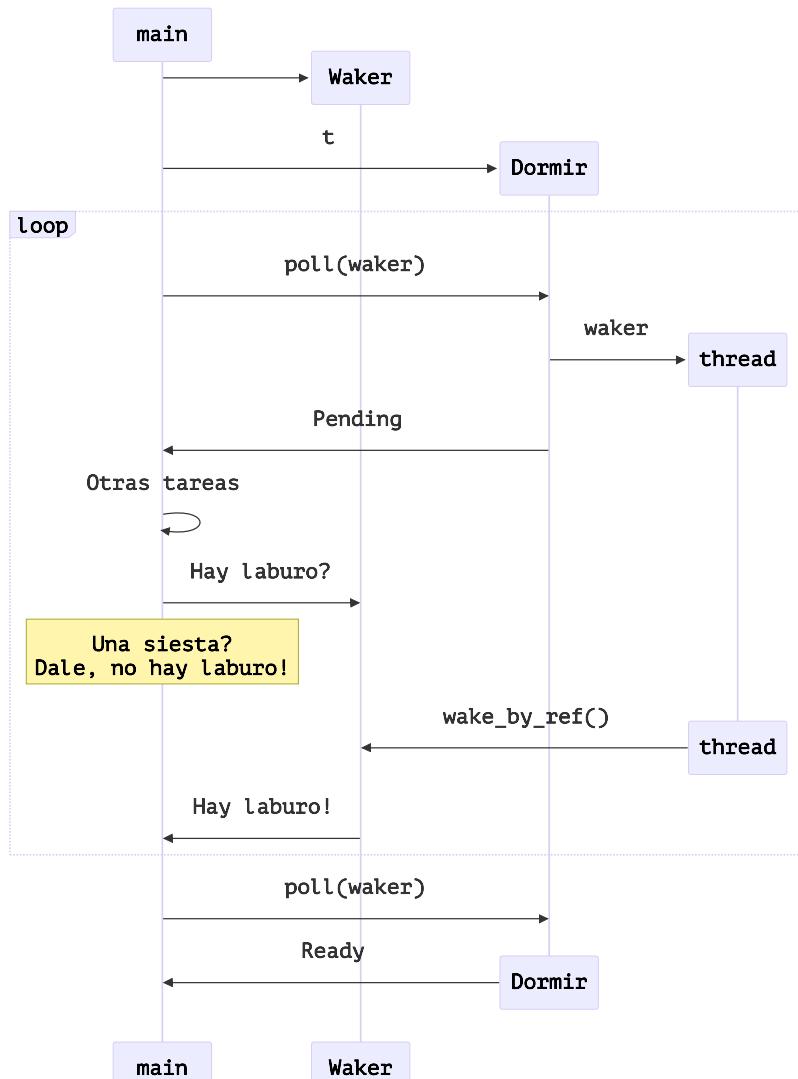


## Notes:

En lugar de llamar a `wake\_by\_ref` inmediatamente, lo que hacemos es crear un thread que va a llamar a `wake\_by\_ref` después de que haya pasado el tiempo que falta para despertarse.

Calculamos el tiempo que falta para despertarse, y creamos un thread que duerme ese tiempo, y luego llama a `wake\_by\_ref`.

Easy Peasy!



## Notes:

Veamos el diagrama de secuencia actualizado.

Basicamente es igual que antes hasta que main llama a `poll` en `Dormir`.

Cuando `Dormir` ve que todavía no es hora de despertarse, crea un thread que va a llamar a `wake\_by\_ref` después de que haya pasado el tiempo que falta para despertarse.

Y devuelve `Poll::Pending`.

Como antes, acá `main` puede hacer otras tareas.

También como antes, `Dormir` todavía no terminó, así que `main` va a tener que volver a llamar a `poll` en algún momento.

Y devuelta como antes, para decidir cuándo llamar a `poll` de nuevo, `main` le pregunta al waker si `Dormir` tiene laburo para hacer.

**A diferencia de antes**, el waker todavía no recibió la llamada a `wake\_by\_ref`, así que le dice a `main`: "Vos tranqui, hacete una siesta, y cuando haya laburo te aviso".

Así que `main` se hace una siesta.

Eventualmente, el thread que creó `Dormir` termina de dormir, y llama a `wake\_by\_ref` en el waker.

Esto le indica al waker que `Dormir` tiene laburo para hacer, así que el waker le avisa a `main` que hay laburo.

`main` se despierta, y vuelve a llamar a `poll` en `Dormir`.

Finalmente, es hora de despertarse, `Dormir` devuelve `Poll::Ready()`, y todo termina felizmente.

Y esto funciona, y no quema ciclos de CPU inútilmente.

Pero tiene *otro* problema...

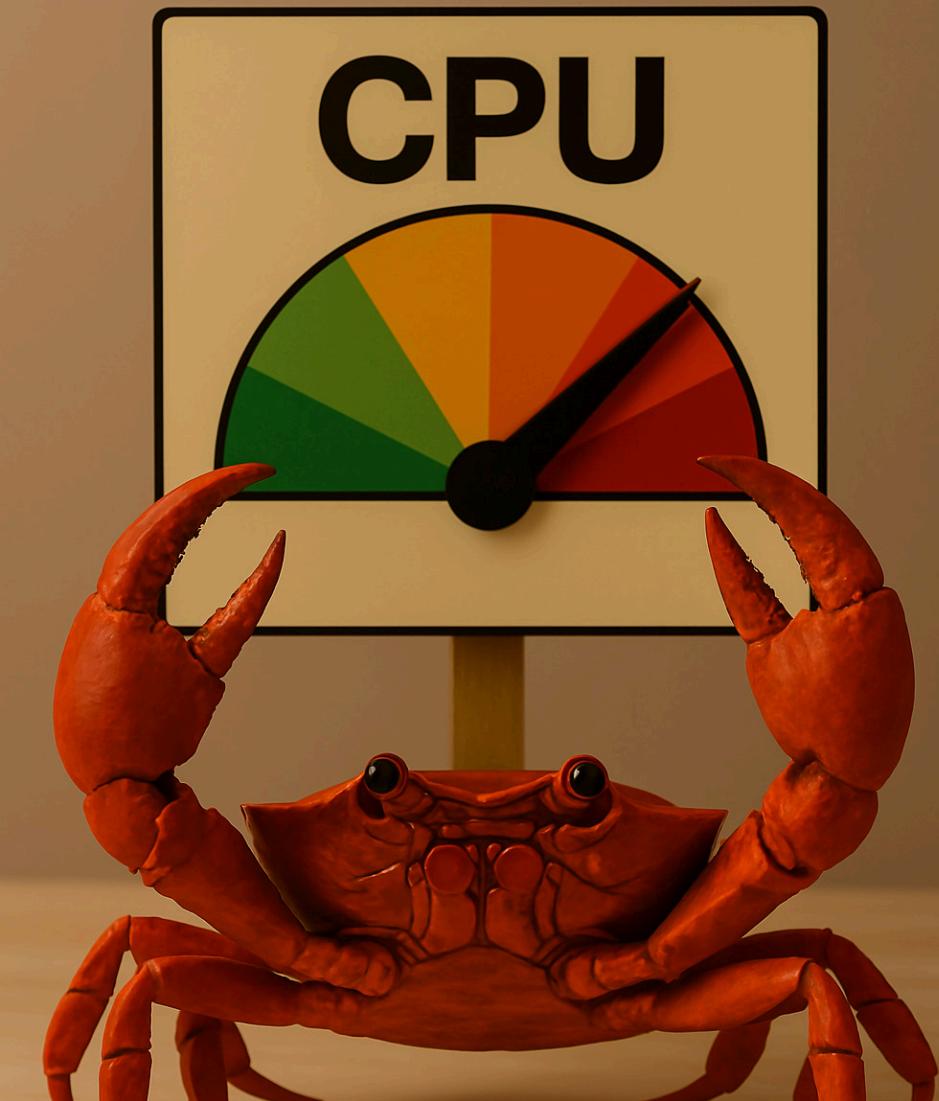
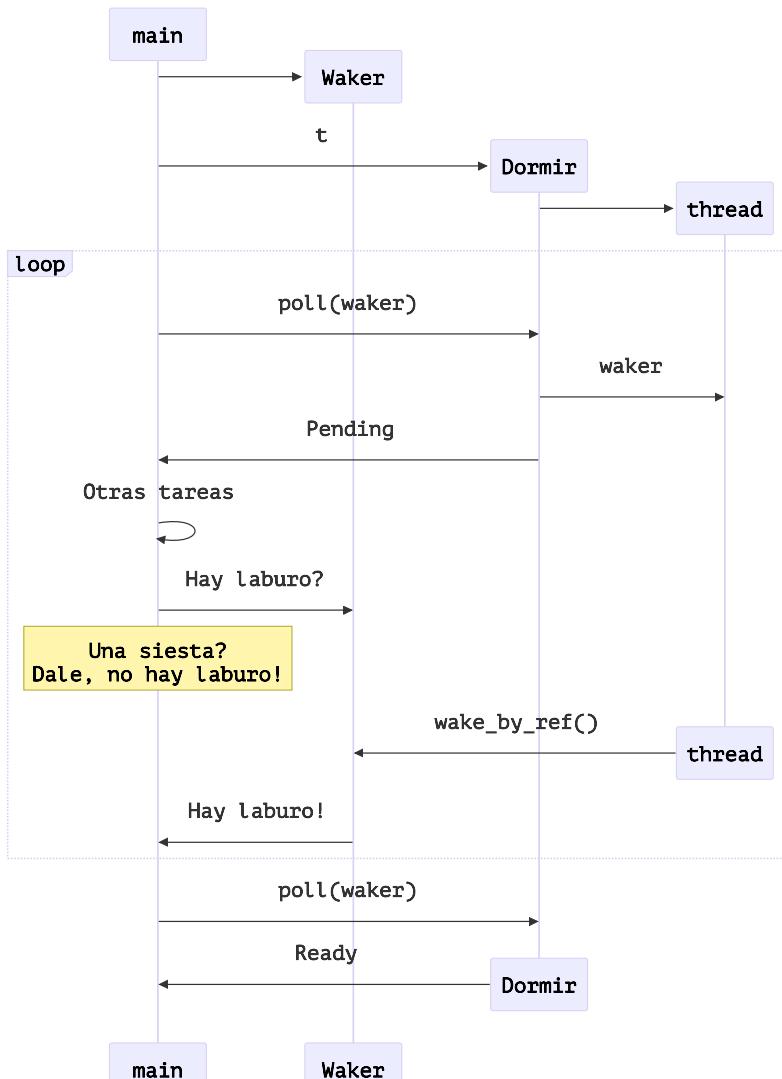
Si bien el runtime **puede** hacerse una siesta, no **tiene** que hacerlo.

El runtime puede decidir no hacer una siesta, y seguir llamando a `poll` repetidamente, como el runtime que implementamos nosotros en main.

El problema ahora, además de que el runtime está quemando ciclos, es que `poll` está siendo llamado miles de veces por segundo, y cada vez que se llama a `poll`, se crea un thread nuevo.

Y al sistema operativo no le va a gustar que le creemos miles de threads por segundo.

Por suerte la solución a este problema es bastante simple.



# Durmiendo

```
1 use std::time::{Duration, Instant};  
2  
3 struct Dormir {  
4     hora: Instant  
5 }  
6 fn dormir(t: Duration) -> Dormir {  
7     let hora = Instant::now() + t;  
8     Dormir { hora }  
9 }
```



## Notes:

Todo muy lindo, pero vamos a ver cómo implementarlo.

# Durmiendo

```
1 use std::time::{Duration, Instant};  
2  
3 struct Dormir {  
4     hora: Instant  
5 }  
6 fn dormir(t: Duration) -> Dormir {  
7     let hora = Instant::now() + t;  
8     std::thread::spawn(move ||{  
9         std::thread::sleep(t);  
10        todo!()  
11    });  
12    Dormir { hora }  
13 }
```



## Notes:

Como dijimos, movemos la creación del thread fuera del loop, y lo ponemos en la función `dormir`.

Tenemos que ver que hacemos cuando el thread se despierte...

# Durmiendo

```
1 use std::time::{Duration, Instant};  
2  
3 struct Dormir {  
4     hora: Instant  
5 }  
6 fn dormir(t: Duration) -> Dormir {  
7     let hora = Instant::now() + t;  
8     let waker = todo!();  
9     std::thread::spawn(move ||{  
10         std::thread::sleep(t);  
11         todo!()  
12     });  
13     Dormir { hora }  
14 }
```



## Notes:

Por lo pronto, necesitamos el *placeholder* para el waker.

# Durmiendo

```
1 use std::time::{Duration, Instant};  
2  
3 struct Dormir {  
4     hora: Instant  
5 }  
6 fn dormir(t: Duration) -> Dormir {  
7     let hora = Instant::now() + t;  
8     let waker = Waker::noop().clone();  
9     std::thread::spawn(move ||{  
10         std::thread::sleep(t);  
11         todo!()  
12     });  
13     Dormir { hora }  
14 }
```



## Notes:

Por suerte podemos crear un waker que no haga nada con `Waker::noop`, y usar eso como placeholder.

# Durmiendo

```
1 use std::time::{Duration, Instant};  
2  
3 struct Dormir {  
4     hora: Instant  
5 }  
6 fn dormir(t: Duration) -> Dormir {  
7     let hora = Instant::now() + t;  
8     let waker = Waker::noop().clone();  
9     let waker = Arc::new(Mutex::new(waker));  
10    std::thread::spawn(move ||{  
11        std::thread::sleep(t);  
12        todo!()  
13    });  
14    Dormir { hora }  
15 }
```



## Notes:

Para poder compartirlo con el thread, y poder actualizarlo desde `poll`, lo ponemos dentro de un `Arc<Mutex<\_>>`.

# Durmiendo

```
1 use std::time::{Duration, Instant};  
2  
3 struct Dormir {  
4     hora: Instant  
5 }  
6 fn dormir(t: Duration) -> Dormir {  
7     let hora = Instant::now() + t;  
8     let waker = Waker::noop().clone();  
9     let waker = Arc::new(Mutex::new(waker));  
10    std::thread::spawn(move ||{  
11        std::thread::sleep(t);  
12        waker.lock().unwrap().wake_by_ref();  
13    });  
14    Dormir { hora }  
15 }
```



## Notes:

Y desde el thread, cuando se despierte, simplemente bloqueamos el mutex, obtenemos el waker, y llamamos a `wake\_by\_ref`.

# Durmiendo

```
1 use std::time::{Duration, Instant};  
2  
3 struct Dormir {  
4     hora: Instant,  
5     despertador: Arc<Mutex<Waker>>,  
6 }  
7 fn dormir(t: Duration) -> Dormir {  
8     let hora = Instant::now() + t;  
9     let waker = Waker::noop().clone();  
10    let waker = Arc::new(Mutex::new(waker));  
11    let despertador = waker.clone();  
12    std::thread::spawn(move ||{  
13        std::thread::sleep(t);  
14        waker.lock().unwrap().wake_by_ref();  
15    });  
16    Dormir { hora, despertador }  
17 }
```



## Notes:

Para poder actualizar el waker desde `poll`, nos guardamos una copia del `Arc<Mutex<\_>>` dentro de la estructura `Dormir`, y lo vamos a llamar `despertador`.

# Durmiendo

```
1 impl Future for Dormir {
2     type Output = ();
3
4     fn poll(
5         self: Pin<&mut Self>,
6         ctx: &mut Context
7     ) -> Poll<()> {
8         let hora = Instant::now();
9
10        if self.hora < hora {
11            Poll::Ready(())
12        } else {
13            todo!()
14        }
15    }
16 }
```



## Notes:

Y ahora, vamos a la implementación de `poll` .

# Durmiendo

```
1 impl Future for Dormir {
2     type Output = ();
3
4     fn poll(
5         self: Pin<&mut Self>,
6         ctx: &mut Context
7     ) -> Poll<()> {
8         let hora = Instant::now();
9
10        if self.hora < hora {
11            Poll::Ready(())
12        } else {
13            Poll::Pending
14        }
15    }
16 }
```



## Notes:

Cuando vemos que todavía no es hora de despertarse, devolvemos `Poll::Pending`.

Pero antes de devolver `Pending`, tenemos que actualizar el waker que está en `despertador`.

# Durmiendo

```
1 impl Future for Dormir {
2     type Output = ();
3
4     fn poll(
5         self: Pin<&mut Self>,
6         ctx: &mut Context
7     ) -> Poll<()> {
8         let hora = Instant::now();
9
10        if self.hora < hora {
11            Poll::Ready(())
12        } else {
13            *self.despertador.lock().unwrap() =
14                ctx.waker().clone();
15            Poll::Pending
16        }
17    }
18 }
```



## Notes:

Así que bloqueamos el mutex, y clonamos el waker que recibimos en `poll` dentro del waker que está en `despertador`.

# Durmiendo

```
1 impl Future for Dormir {
2     type Output = ();
3
4     fn poll(
5         self: Pin<&mut Self>,
6         ctx: &mut Context
7     ) -> Poll<()> {
8         let hora = Instant::now();
9
10        if self.hora < hora {
11            Poll::Ready(())
12        } else {
13            *self.despertador.lock().unwrap()
14                .clone_from(ctx.waker());
15            Poll::Pending
16        }
17    }
18 }
```



## Notes:

Hay una pequeña optimización que podemos hacer acá, y es usar `clone\_from`.

Waker tiene una implementación de `clone\_from` que evita hacer la copia si el waker ya era el mismo.

# El Waker

```
1 fn main() {
2     let fut = saludar("Rust en español".into());
3
4     let mut ctx =
5         Context::from_waker(Waker::noop());
6     let mut fut = pin!(fut);
7
8     let res = loop {
9         match fut.as_mut().poll(&mut ctx) {
10             Poll::Ready(ret) => break ret,
11             Poll::Pending => continue,
12         }
13     };
14 }
```



## Notes:

Muy bien, ya tenemos un futuro que duerme de forma eficiente.

Pero nuestro runtime en `main` no aprovecha esa eficiencia, y sigue quemando ciclos de CPU inútilmente.

Es uno de esos runtimes que mencionamos antes, que decide nunca hacerse una siesta, y seguir llamando a `poll` repetidamente.

Así que actualicemos `main` para permitirle hacerse una siesta.

# El Waker

```
1 struct ChannelWaker(Sender<()>);  
2  
3 fn main() {  
4     let fut = saludar("Rust en español".into());  
5  
6     let mut ctx =  
7         Context::from_waker(Waker::noop());  
8     let mut fut = pin!(fut);  
9  
10    let res = loop {  
11        match fut.as_mut().poll(&mut ctx) {  
12            Poll::Ready(ret) => break ret,  
13            Poll::Pending => continue,  
14        }  
15    };  
16 }
```



## Notes:

Hasta ahora venimos usando el `Waker::noop`.

Este waker no hace nada, y no le podemos preguntar si hay trabajo para hacer.

Para arreglar esto, vamos a implementar nuestro propio waker.

Vamos a usar un canal, un `mpsc::channel`.

El canal tiene dos extremos, un transmisor, `Sender`, y un receptor, `Receiver`.

Para que nuestro waker pueda avisarnos de que hay trabajo para hacer, le vamos a pasar el transmisor, así nos puede avisar enviando un mensaje por el canal.

# El Waker

```
1 use std::task::Wake;
2
3 struct ChannelWaker(Sender<()>);
4
5 impl Wake for ChannelWaker {
6     fn wake(self: std::sync::Arc<Self>) {
7         todo!()
8     }
9 }
10
11 fn main() {
12     let fut = saludar("Rust en español".into());
13
14     let mut ctx =
15         Context::from_waker(Waker::noop());
16     let mut fut = pin!(fut);
17
18     let res = loop {
19         match fut.as_mut().poll(&mut ctx) {
20             Poll::Ready(ret) => break ret,
21             Poll::Pending => continue,
22         }
23     };
24 }
```



## Notes:

Para poder obtener un `std::task::Waker` a partir de nuestro `ChannelWaker`, la manera más fácil es implementando el trait `Wake`.

Este trait requiere implementar un solo método, `wake` (como mínimo).

El método `wake` recibe `self` como un `Arc<Self>`, así el waker puede ser clonado y compartido entre varios futuros.

Cuando el futuro llame `wake\_by\_ref` en el waker, el waker va a llamar a este método `wake`.

# El Waker

```
1 use std::task::Wake;
2
3 struct ChannelWaker(Sender<()>);
4
5 impl Wake for ChannelWaker {
6     fn wake(self: std::sync::Arc<Self>) {
7         let _ = self.0.send(());
8     }
9 }
10
11 fn main() {
12     let fut = saludar("Rust en español".into());
13
14     let mut ctx =
15         Context::from_waker(Waker::noop());
16     let mut fut = pin!(fut);
17
18     let res = loop {
19         match fut.as_mut().poll(&mut ctx) {
20             Poll::Ready(ret) => break ret,
21             Poll::Pending => continue,
22         }
23     };
24 }
```



## Notes:

Sabiendo esto, es fácil, cuando se llame a `wake`, simplemente enviamos un mensaje por el transmisor.

# El Waker

```
1 use std::task::Wake;
2
3 struct ChannelWaker(Sender<()>);
4
5 fn main() {
6     let fut = saludar("Rust en español".into());
7
8     let mut ctx =
9         Context::from_waker(Waker::noop());
10    let mut fut = pin!(fut);
11
12    let res = loop {
13        match fut.as_mut().poll(&mut ctx) {
14            Poll::Ready(ret) => break ret,
15            Poll::Pending => continue,
16        }
17    };
18 }
```



## Notes:

Concentremosnos ahora en nuestro runtime en `main`.

# El Waker

```
1 use std::task::Wake;
2
3 struct ChannelWaker(Sender<()>);
4
5 fn main() {
6     let fut = saludar("Rust en español".into());
7
8     let (tx, rx) = channel();
9
10    let mut ctx =
11        Context::from_waker(Waker::noop());
12    let mut fut = pin!(fut);
13
14    let res = loop {
15        match fut.as_mut().poll(&mut ctx) {
16            Poll::Ready(ret) => break ret,
17            Poll::Pending => continue,
18        }
19    };
20 }
```



## Notes:

Para poder crear nuestro waker, necesitamos crear un canal, y obtenemos el transmisor y el receptor.

# El Waker

```
1 use std::task::Wake;
2
3 struct ChannelWaker(Sender<()>);
4
5 fn main() {
6     let fut = saludar("Rust en español".into());
7
8     let (tx, rx) = channel();
9     let waker = ChannelWaker(tx);
10
11    let mut ctx =
12        Context::from_waker(Waker::noop());
13    let mut fut = pin!(fut);
14
15    let res = loop {
16        match fut.as_mut().poll(&mut ctx) {
17            Poll::Ready(ret) => break ret,
18            Poll::Pending => continue,
19        }
20    };
21 }
```



## Notes:

Y usamos el transmisor para crear nuestro `ChannelWaker`.

# El Waker

```
1 use std::task::Wake;
2
3 struct ChannelWaker(Sender<()>);
4
5 fn main() {
6     let fut = saludar("Rust en español".into());
7
8     let (tx, rx) = channel();
9     let waker = ChannelWaker(tx);
10    let waker = Arc::new(waker).into();
11
12    let mut ctx =
13        Context::from_waker(Waker::noop());
14    let mut fut = pin!(fut);
15
16    let res = loop {
17        match fut.as_mut().poll(&mut ctx) {
18            Poll::Ready(ret) => break ret,
19            Poll::Pending => continue,
20        }
21    };
22 }
```



## Notes:

Para poder convertir nuestro `ChannelWaker` en un `std::task::Waker`, lo ponemos dentro de un `Arc`, y llamamos `.``into()`. Así es como el trait `Wake` nos permite crear un `Waker` a partir de nuestro `ChannelWaker`.

# El Waker

```
1 use std::task::Wake;
2
3 struct ChannelWaker(Sender<()>);
4
5 fn main() {
6     let fut = saludar("Rust en español".into());
7
8     let (tx, rx) = channel();
9     let waker = ChannelWaker(tx);
10    let waker = Arc::new(waker).into();
11
12    let mut ctx = Context::from_waker(&waker);
13    let mut fut = pin!(fut);
14
15    let res = loop {
16        match fut.as_mut().poll(&mut ctx) {
17            Poll::Ready(ret) => break ret,
18            Poll::Pending => continue,
19        }
20    };
21 }
```



## Notes:

Y ahora que tenemos nuestro waker...

En lugar de crear el contexto con el `Waker::noop`, usamos nuestro nuevo `waker`.

# El Waker

```
1 use std::task::Wake;
2
3 struct ChannelWaker(Sender<()>);
4
5 fn main() {
6     let fut = saludar("Rust en español".into());
7
8     let (tx, rx) = channel();
9     let waker = ChannelWaker(tx);
10    let waker = Arc::new(waker).into();
11
12    let mut ctx = Context::from_waker(&waker);
13    let mut fut = pin!(fut);
14
15    let res = loop {
16        match fut.as_mut().poll(&mut ctx) {
17            Poll::Ready(ret) => break ret,
18            Poll::Pending => {
19                let _ = rx.recv();
20            },
21        }
22    };
23 }
```



## Notes:

Y por último, cuando `poll` devuelve `Poll::Pending`, en lugar de simplemente seguir el loop, usamos el receptor para esperar hasta que haya un mensaje.

Cuando haya un mensaje, significa que el waker fue despertado, así que volvemos a llamar a `poll`.

Cuando llamamos `recv`, si no hay nada para recibir, el thread de nuestro runtime se va a dormir hasta que haya algo para recibir, y evitamos quemar ciclos de CPU inútilmente.

Y listo! Le acabamos de enseñar a nuestro runtime a hacerse una siesta cuando no hay nada para hacer.

Y básicamente... eso es todo.

Esto es un runtime que puede ejecutar nuestros futuros, de forma relativamente eficiente.

# Demo!



básica



avanzada



## Notes:

Lo que me gustaría hacer ahora es mostrarles una demo de todo esto andando.

Hay dos versiones de la demo, una versión básica, y una versión avanzada.

La única diferencia entre las dos versiones, es el futuro que le estamos pasando al runtime.

En el caso es un futuro simple, que duerme a imprime un mensaje secuencialmente.

En el caso de la versión avanzada, el futuro crea varios futuros que duermen e imprimen mensajes en paralelo.

Pero el runtime es exactamente el mismo en ambos casos.

Algo para destacar es que los futuros que estamos ejecutando están siendo creados por el compilador.

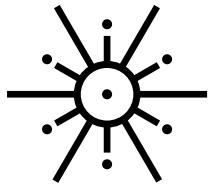
Nosotros nos encargamos de implementar los futuros *fundamentales*, y usamos el compilador para componer nuestros futuros y crear futuros de más alto nivel.

En el caso de la versión avanzada, el futuro que estamos ejecutando es bastante complejo, y en este caso no es solo futuros creados por el compilador, si no que también estamos usando futuros del crate `futures` que al igual que el compilador, está componiendo nuestros futuros fundamentales para crear futuros de más alto nivel.

Y todo eso en menos de 100 líneas de código!

La verdad que cuando yo me metí a hacer esto, no esperaba que fuera tan sencillo.

# Los profesionales



Tokio



Smol



Embassy



## Notes:

Y bueno, esto es un runtime muy simple, tiene varias limitaciones.

Por ejemplo, crea un nuevo thread por cada futuro que duerme, lo cual no es muy eficiente.

Lo único que sabe hacer es dormir, no sabe hacer I/O, ni nada por el estilo.

Pero la idea de este ejercicio no es hacer un runtime para producción, sino entender cómo funcionan los futuros y los runtimes.

En nuestro caso usamos un `mpsc::channel` para permitirle al runtime hacerse una siesta, pero esto puede ser bastante limitante.

Por ejemplo, `tokio` usa un mecanismo mucho más sofisticado basado en `epoll` (al menos en Linux, y mecanismos similares en otros sistemas operativos). Esto le permite a tokio decirle al sistema operativo "me interesan este tipo de eventos en estos archivos, me voy a dormir, y despertarme cuando pase alguna de esas cosas que me interesa".

Eso le permite a tokio, entre otras cosas, hacer I/O de multiples archivos en un solo thread, y de forma muy eficiente.

En el caso de smol, smol utiliza un thread-pool para ejecutar los futuros. Es más limitado que tokio, pero también es mucho más simple, y puede usar todos los mecanismos de la librería estándar de rust para hacer I/O en lugar de tener que hacer todo desde cero.

Finalmente, Embassy es distinto, porque no tiene un sistema operativo que pueda usar. Embassy usa lo que se llama un HAL (Hardware Abstraction Layer) para interactuar con el hardware. El HAL es la interfaz entre Embassy y el hardware, y Embassy provee implementaciones del HAL para varios microcontroladores. El HAL le da a Embassy acceso a funcionalidades de hardware como timers, puertos de I/O, interrupciones de hardware, etc, que Embassy usa para implementar su runtime.



```
fn el_fin() {  
    println!("chau!");  
}
```

## Notes:

Y bueno, eso es todo por hoy!