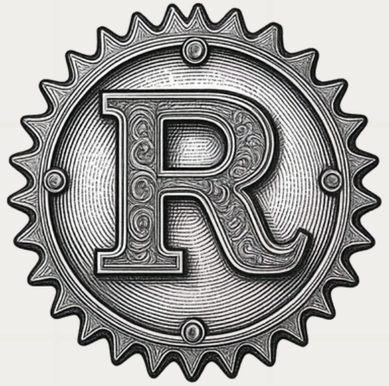
PROGRAMACIÓN CONCURRENTE: TUTORIAL - RUST

Presentado por:
Oscar Leonardo Riveros Perez
Nicolás Martínez López
Cristian Camilo Barrera
Andrés Felipe Rojas Aguilar



The Rust Programming Language

LENGUAJES DE PROGRAMACIÓN - 2025966

ENLACE TUTORIAL

HISTORIA DE RUST

Graydon Hoare, trabajador de Mozilla, empieza el diseño en 2006 después de que el ascensor de su conjunto fallara por un problema de software (vivía en el piso 21)

2006 — Graydon inicia el desarrollo en su tiempo libre

2009 — Graydon presenta Rust en Mozilla y consigue patrocinio

Enero 20 del 2012 — Primer lanzamiento público, Rust 0.1

2013 — Se elimina el recolector de basura

Mayo 15 del 2015 — Lanzamiento de la primera versión estable, Rust 1.0

2016 — Motor de navegador web Servo, por Mozilla, en Rust

2017-2020 — Múltiples aplicaciones en empresas de tecnología

2021 — Se establece la Rust Foundation

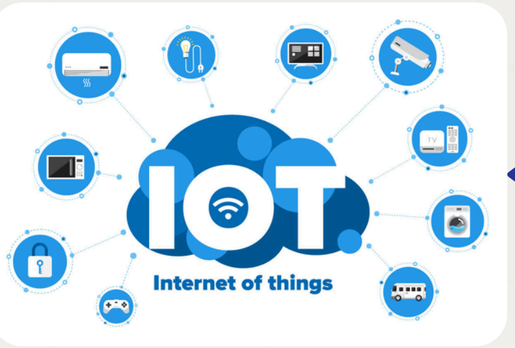
APLICACIONES

Microsoft

Linux Kernel

Sistemas Operativos

aws



Redes

Backend Cloud

Concurrencia Segura

Web Assembly / Web

ferrous systems

ACTIX

Deno

CLOUDFLARE

Figma

¿POR QUÉ RUST ES IMPORTANTE?

Rust combina alto rendimiento con seguridad de memoria y concurrencia segura.

SEGURIDAD DE MEMORIA

- Previene null pointers y buffer overflows
- Verificación en compilación
- Sin Garbage Collector

CONCURRENCIA SEGURA

- Prevención de data races
- Thread safety en compile-time
- Ideal para sistemas concurrentes

ALTO RENDIMIENTO

- Rendimiento cercano a C/C++
- Control eficiente de recursos
- Zero-cost abstractions

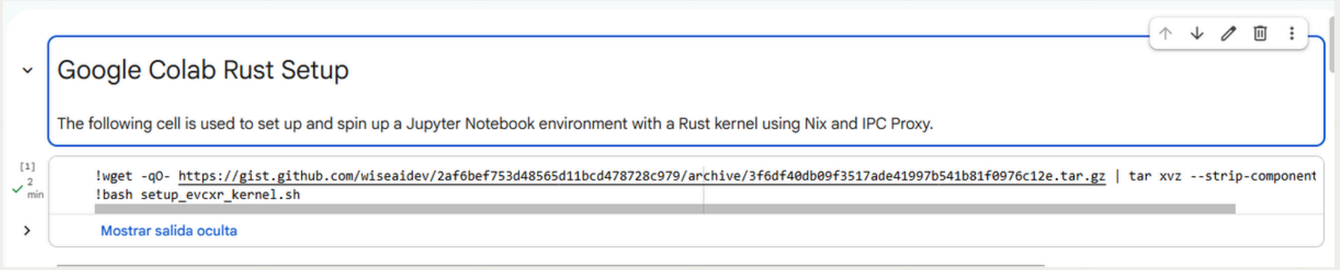
ECOSISTEMA MODERNO

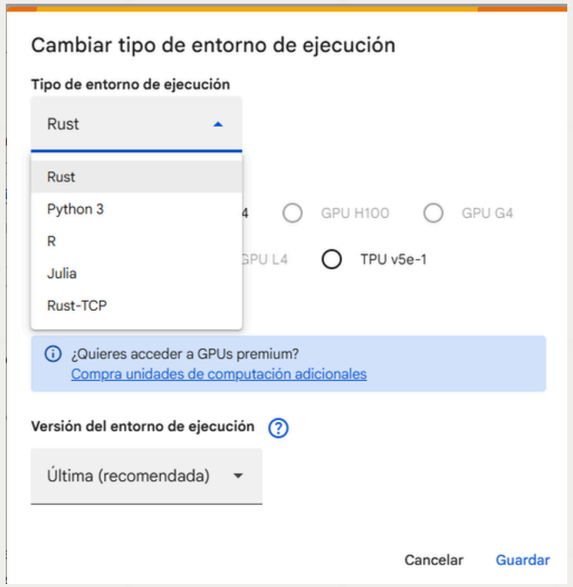
- Cargo y crates
- Tooling integrado
- Adopción creciente en la industria

Rust fue diseñado para resolver uno de los problemas más difíciles de los sistemas concurrentes: las condiciones de carrera (data races), detectándolas durante la compilación en lugar de durante la ejecución.

EJECUCIÓN DE RUST EN GOOGLE COLLAB (PYTHON 3)

Existe una documentación para tratar de **instalar/ejecutar** el **kernel de Rust** en **Google Collab**, sin embargo, actualmente no está funcionando dicha implementación **[y]**. https://github.com/evcxr/evcxr/tree/main/evcxr_jupyter





EJECUCIÓN DE RUST EN GOOGLE COLLAB (PYTHON 3)

Otra forma de poder ejecutar el código de **Rust** en **Google Collab** es usando la sintaxis de **cell magic** en jupyter **(%%)** donde se define una función de python que compile el código, lo ejecute y muestre errores de sintaxis.

Para ello se ha de instalar **Rust** en el entorno de **Collab**, ademas de instalar la dependencia **Tokio** para concurrencia:

```
!curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh -s -- -y
import os
os.environ['PATH'] += ":/root/.cargo/bin"
!cargo new rust_playground
!cd rust_playground && cargo add tokio --features full
```

***Nota:** **Cargo** es el gestor de paquetes oficial de **Rust**, por ende, es importante importarlo en el entorno de **Collab**

EJECUCIÓN DE RUST EN GOOGLE COLLAB (PYTHON 3)

Una vez se cuenta con el compilador de **Rust** en el entorno de **Collab**, se genera una **función magic** para ejecutar código de **Rust** cuando se usa el identificador de celda mágica **%%rust**:

```python
from IPython.core.magic import register_cell_magic
import subprocess

@register_cell_magic
def rust(line, cell):
    # Escribir código
    with open(
        "rust_playground/src/main.rs",
        "w"
    ) as f:
        f.write(cell)

    # Ejecutar cargo run
    process = subprocess.Popen(
        ["cargo", "run"],
        cwd="rust_playground",
        stdout=subprocess.PIPE,
        stderr=subprocess.STDOUT,
        text=True,
        bufsize=1
    )

    # Mostrar salida en tiempo real
    for linea in process.stdout:
        print(linea, end="")

    process.wait()
```

Prueba de ejecución:

```
%%rust
fn saludar(nombre: &str) {
    println!("Hola, {} desde Rust!", nombre);
}

fn main() {
    saludar("Colab");
}
```

```
   Compiling rust_playground v0.1.0 (/content/rust_playground)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.40s
     Running `target/debug/rust_playground`
Hola, Colab desde Rust!
```

***Nota:** Dado que el compilador/kernel sigue siendo **Python**, este mostrará errores de sintaxis en el resaltado del código.

SISTEMA DE TIPOS

Declaración de Variables y Mutabilidad

En Rust las variables son **inmutables por defecto**, lo que significa que una vez asignado un valor, este no puede modificarse, la base para declarar es haciendo uso de la palabra reservada let.

```
3 fn main() {
4     let x = 10;
5     println!("{}", x);
6 }
```

```
3 fn main() {
4     let x = 10;
5     x = 56;
6     println!("{}", x);
7 }
```

Variables Mutables

Para permitir modificaciones se utiliza la palabra clave mut.

```
2 fn main() {
3     let mut contador = 0;
4     println!("{}", contador);
5
6     contador += 1;
7     println!("{}", contador);
8
9
10    contador = 56;
11
12    println!("{}", contador);
13 }
```

SISTEMA DE TIPOS

Declaración de Variables y Mutabilidad

Inferencia de Tipos

Rust deduce automáticamente el tipo de una variable.

```rust
fn main() {
    let mut mi_variable = 10; // Rust infiere i32
    println!("Valor inicial: {}", mi_variable);

    // Esto causará un error de compilación
    // mi_variable = "Hola";
    // println!("Después de asignar string: {}", mi_variable);

    // Esto también causará un error de compilación
    // mi_variable = 20.5;
    // println!("Después de asignar float: {}", mi_variable);
```

```rust
%%rust

fn main() {
    let edad = 20;       // Rust infiere i32 por defecto
    let nombre = "Ana"; // Rust infiere &str

    println!("Edad: {}", edad);
    println!("Nombre: {}", nombre);
}
```

Especificación Explícita de Tipos

También puedes especificar el tipo de una variable explícitamente usando la sintaxis variable: tipo.

```rust
%%rust

fn main() {
    let edad: u32 = 20;        // Entero sin signo de 32 bits
    let promedio: f64 = 4.5;  // Número de punto flotante de 64 bits

    println!("Edad (u32): {}", edad);
    println!("Promedio (f64): {}", promedio);
}
```

SISTEMA DE TIPOS

Declaración de Variables y Mutabilidad

Tipos Primitivos en RUST

Categoría	Tipo	Descripción
Enteros	`i8`	Entero con signo de 8 bits
	`i16`	Entero con signo de 16 bits
	`i32`	Entero con signo de 32 bits (por defecto para enteros)
	`i64`	Entero con signo de 64 bits
	`i128`	Entero con signo de 128 bits
	`isize`	Entero con signo del tamaño de la arquitectura (p. ej., 64 bits en x86-64)
	`u8`	Entero sin signo de 8 bits
	`u16`	Entero sin signo de 16 bits
	`u32`	Entero sin signo de 32 bits
	`u64`	Entero sin signo de 64 bits
	`u128`	Entero sin signo de 128 bits
	`usize`	Entero sin signo del tamaño de la arquitectura (p. ej., 64 bits en x86-64)
Punto Flotante	`f32`	Número de punto flotante de precisión simple (32 bits)
	`f64`	Número de punto flotante de doble precisión (64 bits, por defecto para flotantes)
Booleanos	`bool`	`true` o `false`
Caracteres	`char`	Un carácter Unicode escalar (cuatro bytes)
Tuplas	`(T, U, ..)`	Colección de valores de diferentes tipos de longitud fija
Arrays	`[T; N]`	Colección de valores del mismo tipo de longitud fija
Slices	`&[T]`	Referencia a una parte de una colección (como un `Vec` o un `Array`)
Strings	`&str`	Slice de cadena inmutable (referencia a datos UTF-8 válidos)
	`String`	Cadena de caracteres UTF-8 mutable y propia
Unit Type	`()`	Un tipo con un único valor, el tipo

SISTEMA DE TIPOS

Declaración de Funciones

Sintaxis General

```
fn nombre_funcion(parametro: Tipo) -> TipoRetorno {
    // código
}
```

Funciones Sin Retorno

Una función sencilla que no devuelve ningún valor explícito (devuelve el tipo () unit type implícitamente).

```
%%rust

fn saludar(nombre: &str) {
    println!("Hola {} desde la función!", nombre);
}

fn main() {
    saludar("Carlos");
    saludar("Ana");
}
```

Funciones Con Retorno

```
%%rust

fn sumar(a: i32, b: i32) -> i32 {
    a + b // La última expresión se retorna implícitamente sin 'return'
}

fn main() {
    let resultado = sumar(5, 3);
    println!("El resultado de la suma es: {}", resultado);

    let otro_resultado = sumar(10, -4);
    println!("Otro resultado: {}", otro_resultado);
}
```

```
   Compiling rust_playground v0.1.0 (/content/rust_playground)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.13s
     Running `target/debug/rust_playground`
El resultado de la suma es: 8
Otro resultado: 6
```

Funciones Con Multiples Retornos

```
%%rust

fn multiplicar(x: i32, y: i32, z: i32) -> i32 {
    x * y * z
}

fn main() {
    let producto = multiplicar(2, 4, 5);
    println!("El producto es: {}", producto);
}
```

```
   Compiling rust_playground v0.1.0 (/content/rust_playground)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.14s
     Running `target/debug/rust_playground`
El producto es: 40
```

SISTEMA DE TIPOS

Declaración de Funciones

Funciones Lambda (Closures)

- Funciones anónimas
- Permiten escribir código más compacto
- Muy usadas con iteradores (map, filter)
- Pueden capturar variables externas

Sintaxis General → |parámetros| expresión

```
%%rust
fn main() {
    let numeros = vec![1, 2, 3, 4];

    let cuadrados: Vec<i32> = numeros
        .iter()
        .map(|x| x * x)
        .collect();

    println!("{:?}", cuadrados);
}
```

```
   Compiling rust_playground v0.1.0 (/content/rust_playground)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.20s
     Running `target/debug/rust_playground`
[1, 4, 9, 16]
```

Sin

```
%%rust

fn saludar(nombre: &str) {
    println!("{} {}", mensaje, nombre); // ERROR
}
```

```
   Compiling rust_playground v0.1.0 (/content/rust_playground)
error[E0425]: cannot find value `mensaje` in this scope
 --> src/main.rs:3:23
  |
3 |     println!("{} {}", mensaje, nombre); // ERROR
  |                        ^^^^^^^ not found in this scope

error[E0601]: `main` function not found in crate `rust_playground`
 --> src/main.rs:4:2
  |
4 | }
  |  ^ consider adding a `main` function to `src/main.rs`

Some errors have detailed explanations: E0425, E0601.
For more information about an error, try `rustc --explain E0425`.
error: could not compile `rust_playground` (bin "rust_playground") due to 2 previous errors
```

Con

```
%%rust

fn main() {
    let mensaje = "Hola";

    let saludar = |nombre| {
        println!("{} {}", mensaje, nombre);
    };

    saludar("Clase");
}
```

```
   Compiling rust_playground v0.1.0 (/content/rust_playground)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.12s
     Running `target/debug/rust_playground`
Hola Clase
```

SISTEMA DE TIPOS

Declaración de Funciones

Tipado Explícito de Parámetros
Todos los parámetros de una función deben declarar su tipo.

Tipado Explícito de Retorno
El tipo de retorno de la función debe especificarse explícitamente después de ->. Si la función no devuelve nada, implícitamente devuelve () (el unit type).

Retorno Implícito
La última expresión en el cuerpo de la función puede retornarse sin la palabra clave return;. Si terminas una línea con un punto y coma, se convierte en una sentencia y no devolverá un valor, a menos que sea el tipo ().

Importancia en la Concurrencia
Las funciones reciben la propiedad o referencias de los datos, lo que permite que el compilador verifique accesos seguros cuando múltiples hilos utilizan información compartida.

SISTEMA DE TIPOS

Tipos Personalizados

Structs - Estructuras

Los structs permiten agrupar datos relacionados en una única estructura, lo que es útil para crear tipos complejos que modelan entidades del mundo real.

```
Nombre: Juan
Edad: 25
Edad de Ana después de un año: 31
```

```rust
%%rust

// Definición de un struct Persona
struct Persona {
    nombre: String,
    edad: u32,
}

fn main() {
    // Creación de una instancia de Persona
    let persona = Persona {
        nombre: String::from("Juan"),
        edad: 25,
    };

    // Acceso a los campos del struct
    println!("Nombre: {}", persona.nombre);
    println!("Edad: {}", persona.edad);

    // También se puede crear de forma mutable
    let mut persona_mutable = Persona {
        nombre: String::from("Ana"),
        edad: 30,
    };

    persona_mutable.edad += 1;
    println!("Edad de Ana después de un año: {}", persona_mutable.edad);
}
```

SISTEMA DE TIPOS

Tipos Personalizados

Enums - Enumeraciones

Los enums permiten definir un tipo que puede ser uno de varios estados posibles, lo que es ideal para situaciones donde una variable puede tomar un conjunto finito de valores.

```rust
// Definición de un enum simple
enum EstadoUsuario {
    Activo,
    Inactivo,
    Suspendido,
}

// Enums con datos asociados (tuple structs o anonymous structs)
enum Mensaje {
    Quit, // Sin datos
    Echo(String), // Con un String
    Mover { x: i32, y: i32 }, // Con un struct anónimo
    CambiarColor(i32, i32, i32), // Con tres i32
}

fn procesar_estado(estado: EstadoUsuario) {
    match estado {
        EstadoUsuario::Activo => println!("El usuario está activo."),
        EstadoUsuario::Inactivo => println!("El usuario está inactivo."),
        EstadoUsuario::Suspendido => println!("El usuario ha sido suspendido."),
    }
}

```

```rust
fn main() {
    // Uso de un enum simple
    let usuario1 = EstadoUsuario::Activo;
    let usuario2 = EstadoUsuario::Inactivo;
    procesar_estado(usuario1);
    procesar_estado(usuario2);

    // Uso de enums con datos asociados
    let mensaje1 = Mensaje::Echo(String::from("¡Hola!"));
    let mensaje2 = Mensaje::Mover { x: 10, y: 20 };
    let mensaje3 = Mensaje::CambiarColor(255, 0, 128);

    match mensaje1 {
        Mensaje::Echo(texto) => println!("Mensaje de eco: {}", texto),
        _ => (),
    }

    match mensaje2 {
        Mensaje::Mover { x, y } => println!("Mover a X: {}, Y: {}", x, y),
        _ => (),
    }

    match mensaje3 {
        Mensaje::CambiarColor(r, g, b) => println!("Cambiar color a R:{}, G:{}, B:{}", r, g, b),
        _ => (),
    }
}
```

SISTEMA DE TIPOS

Tipos Personalizados

impl - Implementación de Métodos

Podemos asociar funciones (métodos) a nuestros structs y enums usando bloques impl. Estos métodos pueden operar sobre los datos de la instancia.

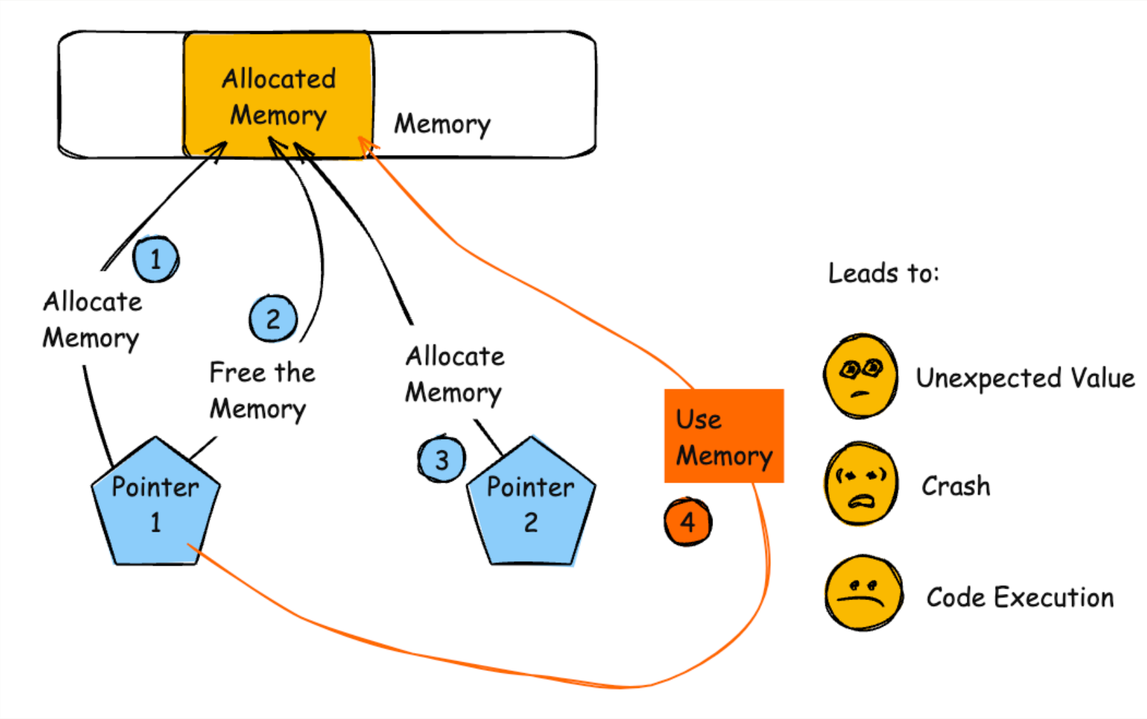
```rust
struct Rectangulo {
    ancho: u32,
    alto: u32,
}

// Implementación de métodos para el struct Rectangulo
impl Rectangulo {
    // Un método que calcula el área. Toma una referencia inmutable a self (&self).
    fn area(&self) -> u32 {
        self.ancho * self.alto
    }

    // Un método que crea un nuevo Rectangulo (método asociado, no necesita self)
    fn cuadrado(lado: u32) -> Rectangulo {
        Rectangulo {
            ancho: lado,
            alto: lado,
        }
    }
}

```

```rust
fn main() {
    let r1 = Rectangulo {
        ancho: 10,
        alto: 5,
    };

    // Llamada al método de instancia
    println!("El área del rectángulo es: {}", r1.area());

    let r2 = Rectangulo::cuadrado(7);
    println!("El área del cuadrado es: {}", r2.area());
}
```

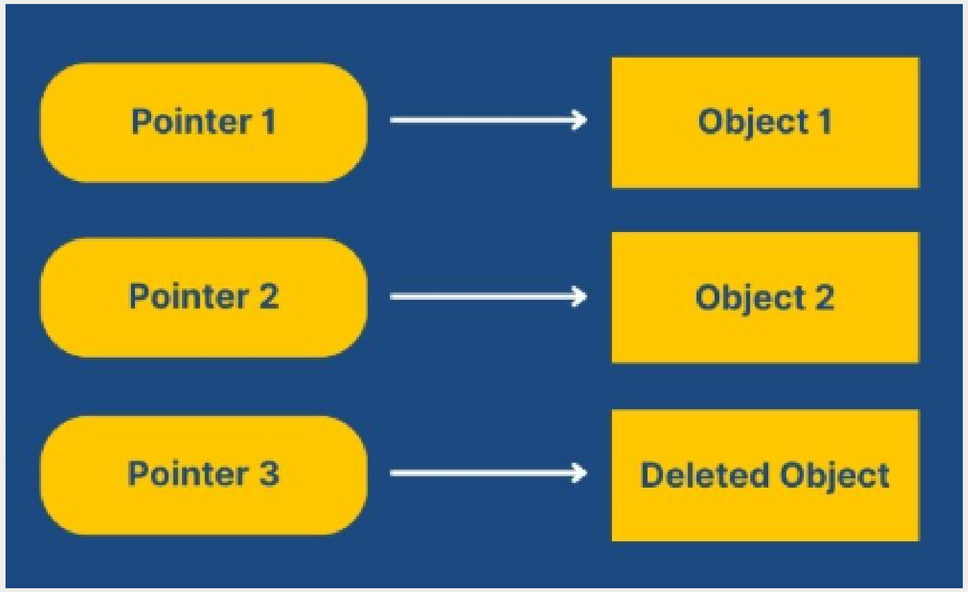
MANEJO DE MEMORIA

Bugs por gestión manual de memoria

USE AFTER FREE



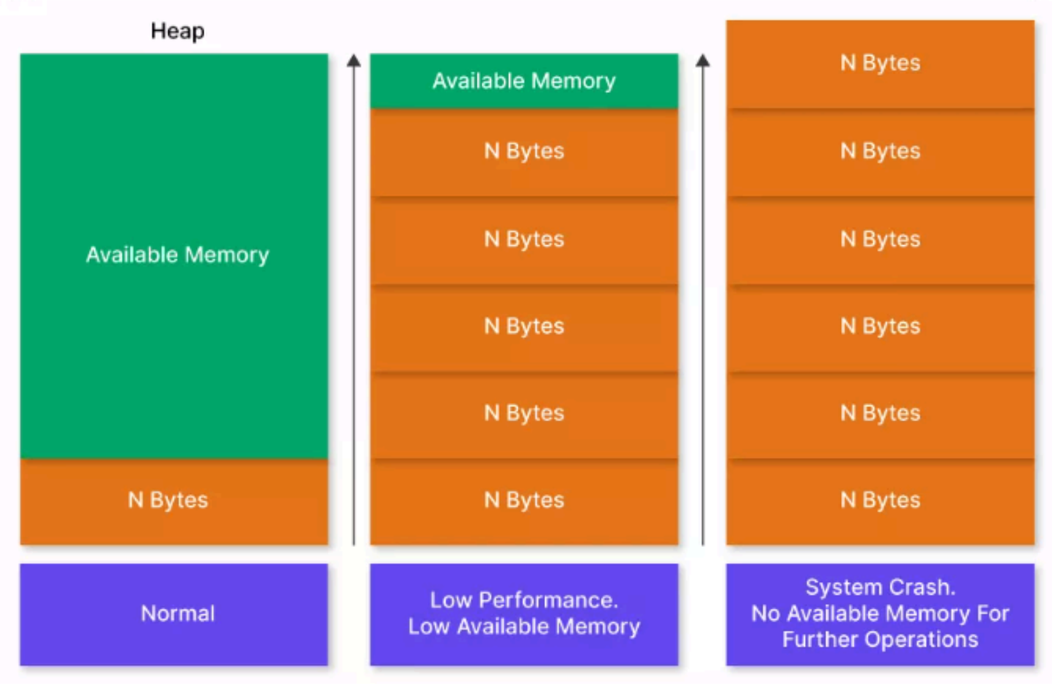
DANGLING POINTER



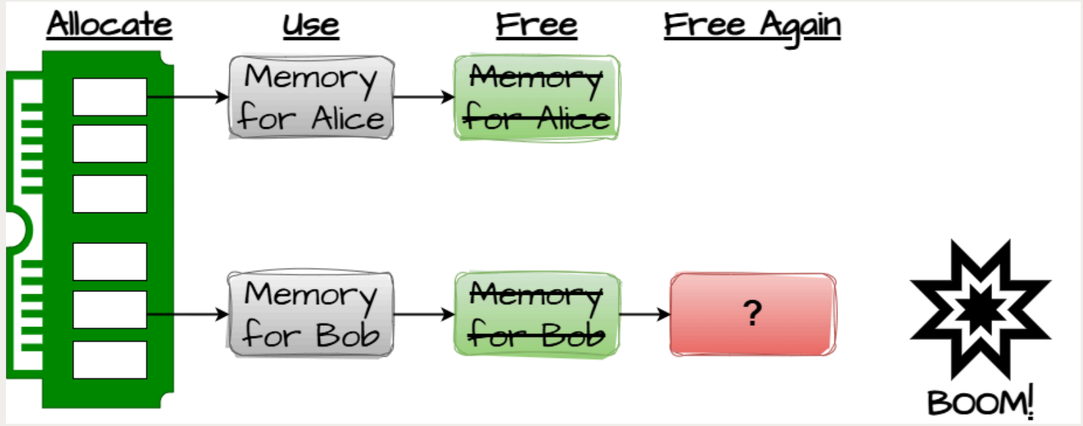
MANEJO DE MEMORIA

Bugs por gestión manual de memoria

MEMORY LEAK



DOUBLE FREE



MANEJO DE MEMORIA

Ownership:

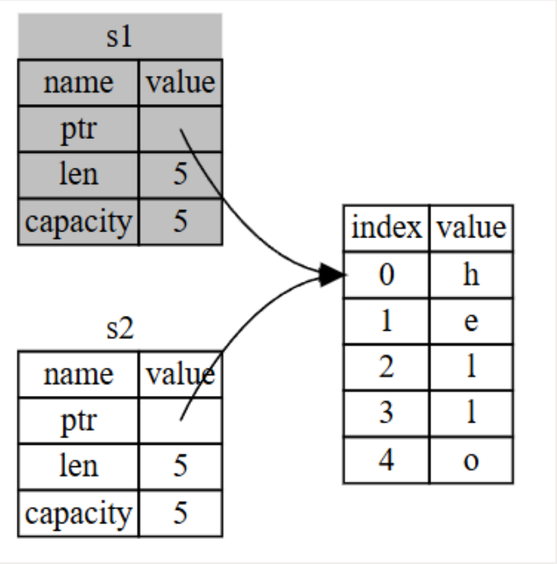
¿Cómo permitirle a los programadores controlar la memoria ellos mismos, sin que esto cause bugs?

Todo valor tiene una variable que es su dueño, y solo uno. Cuando sale del scope el valor se elimina **automáticamente**

La propiedad (ownership) se pasa de una variable a otra (un **move**) y la referencia de la variable anterior queda invalidada.
También ocurre al llamar funciones

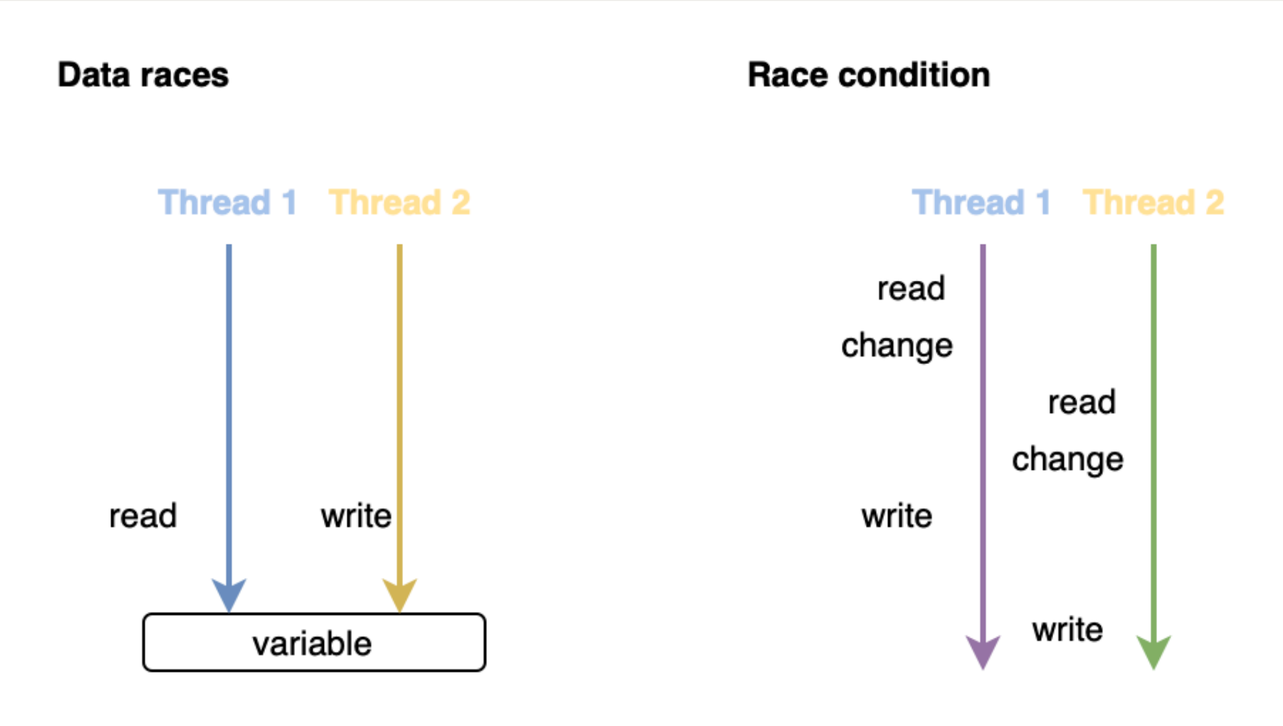
Detectado en **tiempo de compilación**

```
fn main() {
    let s1 = String::from("Hello");
    let s2 = s1;

    // println!("{}", s1); Error:
    // value borrowed after move

    println!("{}", s2);
}
```



MANEJO DE MEMORIA

Bugs por concurrencia



Cuando se tienen múltiples hilos, el orden de ejecución no es determinista y puede ocurrir que

- El resultado final cambie dependiendo del orden (**race condition**)
- Un hilo lector lea un valor desactualizado (stale read) o el escritor sobreescriba antes de lo debido (**data race**)

MANEJO DE MEMORIA

Borrowing:

En lugar de recibir la propiedad de un valor, lo podemos **pedir prestado** mediante una **referencia**

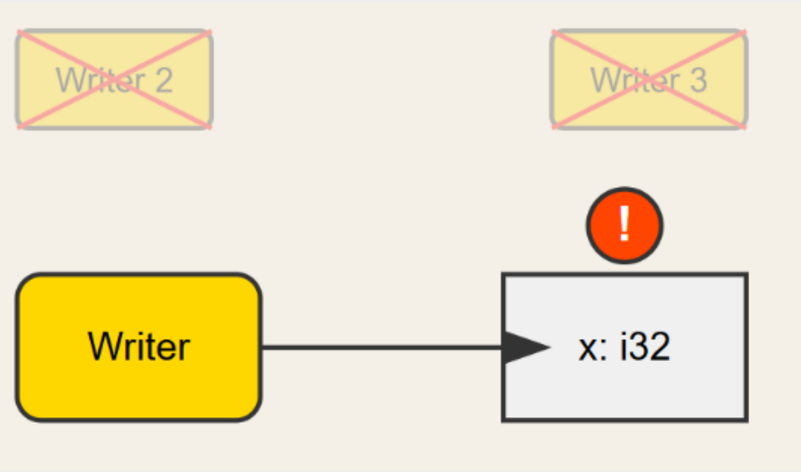
BORROW INMUTABLE

Un valor puede tener múltiples lectores, pero sin ningún escritor al mismo tiempo

Dueño: biblioteca
Lectores: múltiples
Escritores: ninguno

BORROW MUTABLE

Solo puede haber un escritor, y no puede modificar el valor mientras los lectores lo quieran leer



Verificado en **tiempo de compilación**

MECANISMOS DE CONCURRENCIA

Hilos:

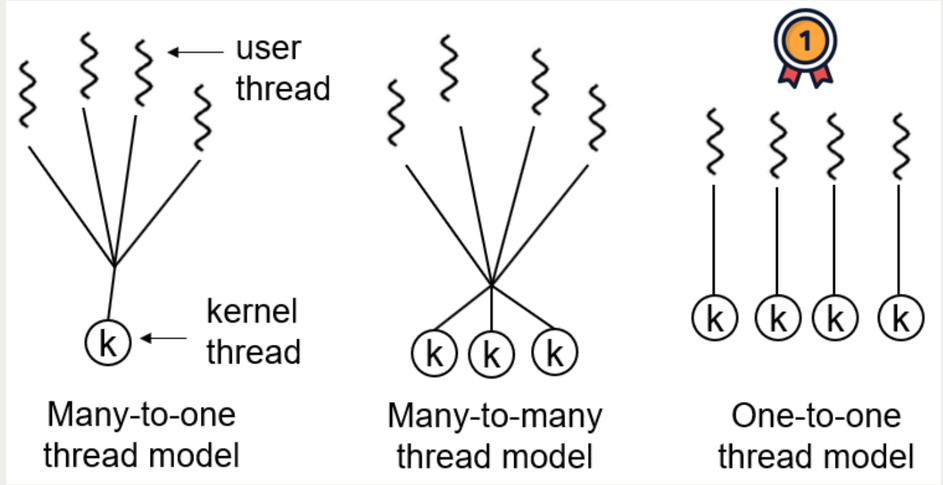
En **Rust** la librería estándar **Threads** le solicita al S.O la creación de **hilos de kernel** ya que implementa el modelo 1:1 **[Z]**, por lo tanto, este esquema de **concurrencia** es potencialmente **paralelo**.

Sintaxis:

- Importar la librería estándar: **use std::thread;**
- Invocar a un hilo: **thread::spawn(f)**
- Obtener el resultado de un hilo: **join()**
- Pausar un hilo: **thread::sleep(time)**
- Ceder el recurso del hilo: **thread::yield_now()**
- Obtener información del hilo: **thread::current()**

***Nota:** Dado que **join()** maneja tanto el **resultado** como el **error** del **contexto del hilo**, se usa el método **join().unwrap()** para obtener el **resultado** y manejar automáticamente el **error**.

[z] https://doc.rust-lang.org/stable/book/ch16-01-threads.html



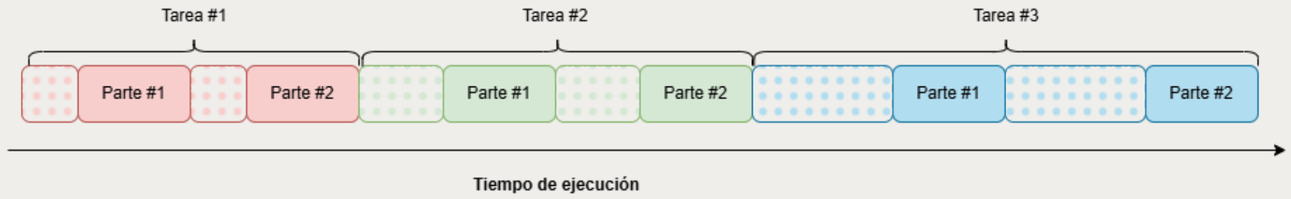
***Nota:** El parámetro **f** del método **spawn**, puede ser una **función declarada** o una **función lambda**.

MECANISMOS DE CONCURRENCIA

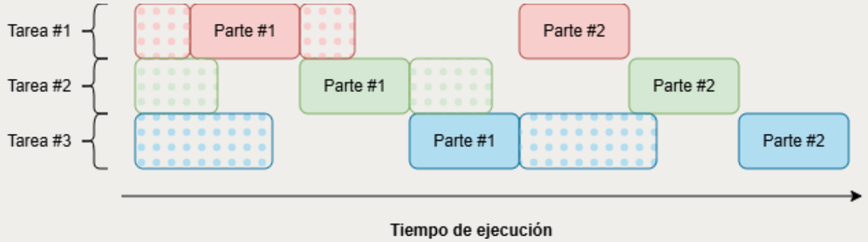
Hilos:

Ejemplo:

Tarea secuencial:



Tarea concurrente:



MECANISMOS DE CONCURRENCIA

Hilos:

Ejemplo:

```
%%rust
use std::thread;
use std::time::{Duration, Instant};

fn calculo_pesado(limite: u64) {
    let mut acumulador: f64 = 0.0;

    for i in 0..limite {
        acumulador = acumulador * (i as f64);
    }
}

fn ejecutar_tarea(i: u64, n_partes: u64, carga: u64) {
    for j in 0..n_partes {
        // --- Simulando operacion I/O ---
        println!("Tarea {:02} [{:?}] -> Requiriendo datos en disco", i, thread::current().id());
        thread::sleep(Duration::from_millis((100 * i * (n_partes - j)) % 10000));

        // --- PARTE n ---
        println!("Tarea {:02} [{:?}] -> Iniciando PARTE {:02}", i, thread::current().id(), j + 1);
        calculo_pesado(carga);
    }

    println!("Tarea {:02} [{:?}] Trabajo finalizado", i, thread::current().id());
}
```

```
fn main() {
    let inicio_programa = Instant::now();
    let mut handles = vec![];

    let n_tareas = 10;
    let n_partes = 5;
    let carga = 50_000_000;

    for i in 0..n_tareas {
        let handle = thread::spawn(move || ejecutar_tarea(i, n_partes, carga));
        handles.push(handle);
    }

    // Esperar a que todos terminen
    for h in handles {
        h.join().unwrap();
    }

    let duracion_total = inicio_programa.elapsed();

    println!("--- Todas las tareas han finalizado. ---");
    println!("DURACIÓN TOTAL DEL PROGRAMA: {:?}", duracion_total);
}
```

***Nota:** El entorno de **collab** ofrece únicamente 1 **core** con 2 **threads**.

MECANISMOS DE CONCURRENCIA

Hilos:

Ejemplo:

```
...     Compiling rust_playground v0.1.0 (/content/rust_playground)
        Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.27s
         Running `target/debug/rust_playground`
    Tarea 01 [ThreadId(3)] -> Requiriendo datos en disco
    Tarea 00 [ThreadId(2)] -> Requiriendo datos en disco
    Tarea 00 [ThreadId(2)] -> Iniciando PARTE 01
    Tarea 02 [ThreadId(4)] -> Requiriendo datos en disco
    Tarea 03 [ThreadId(5)] -> Requiriendo datos en disco
    Tarea 06 [ThreadId(8)] -> Requiriendo datos en disco
    Tarea 04 [ThreadId(6)] -> Requiriendo datos en disco
    Tarea 07 [ThreadId(9)] -> Requiriendo datos en disco
    Tarea 08 [ThreadId(10)] -> Requiriendo datos en disco
    Tarea 09 [ThreadId(11)] -> Requiriendo datos en disco
    Tarea 05 [ThreadId(7)] -> Requiriendo datos en disco
    Tarea 01 [ThreadId(3)] -> Iniciando PARTE 01
    Tarea 00 [ThreadId(2)] -> Requiriendo datos en disco
    Tarea 00 [ThreadId(2)] -> Iniciando PARTE 02
    Tarea 02 [ThreadId(4)] -> Iniciando PARTE 01
    Tarea 03 [ThreadId(5)] -> Iniciando PARTE 01
    Tarea 04 [ThreadId(6)] -> Iniciando PARTE 01
    Tarea 05 [ThreadId(7)] -> Iniciando PARTE 01
    Tarea 01 [ThreadId(3)] -> Requiriendo datos en disco
    Tarea 06 [ThreadId(8)] -> Iniciando PARTE 01
```

```
    Tarea 02 [ThreadId(4)] Trabajo finalizado
    Tarea 04 [ThreadId(6)] -> Requiriendo datos en disco
    Tarea 08 [ThreadId(10)] -> Requiriendo datos en disco
    Tarea 04 [ThreadId(6)] -> Iniciando PARTE 05
    Tarea 07 [ThreadId(9)] -> Iniciando PARTE 04
    Tarea 05 [ThreadId(7)] -> Requiriendo datos en disco
    Tarea 09 [ThreadId(11)] -> Requiriendo datos en disco
    Tarea 03 [ThreadId(5)] Trabajo finalizado
    Tarea 05 [ThreadId(7)] -> Iniciando PARTE 05
    Tarea 06 [ThreadId(8)] -> Requiriendo datos en disco
    Tarea 08 [ThreadId(10)] -> Iniciando PARTE 04
    Tarea 06 [ThreadId(8)] -> Iniciando PARTE 05
    Tarea 04 [ThreadId(6)] Trabajo finalizado
    Tarea 09 [ThreadId(11)] -> Iniciando PARTE 04
    Tarea 07 [ThreadId(9)] -> Requiriendo datos en disco
    Tarea 05 [ThreadId(7)] Trabajo finalizado
    Tarea 07 [ThreadId(9)] -> Iniciando PARTE 05
    Tarea 08 [ThreadId(10)] -> Requiriendo datos en disco
    Tarea 06 [ThreadId(8)] Trabajo finalizado
    Tarea 09 [ThreadId(11)] -> Requiriendo datos en disco
    Tarea 08 [ThreadId(10)] -> Iniciando PARTE 05
    Tarea 07 [ThreadId(9)] Trabajo finalizado
    Tarea 08 [ThreadId(10)] Trabajo finalizado
    Tarea 09 [ThreadId(11)] -> Iniciando PARTE 05
    Tarea 09 [ThreadId(11)] Trabajo finalizado
    --- Todas las tareas han finalizado. ---
    DURACIÓN TOTAL DEL PROGRAMA: 29.79427011s
```

MECANISMOS DE CONCURRENCIA

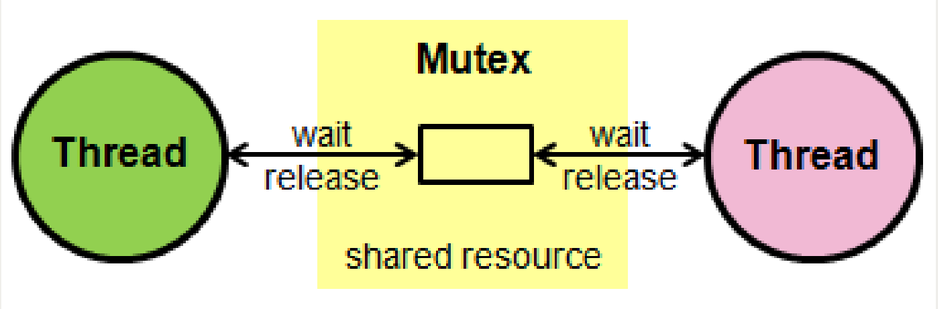
Bloqueos (Mutex):

En **Rust**, **Mutex<T>** es un mecanismo de **sincronización** basado en **exclusión mutua** que permite coordinar el acceso concurrente a **memoria compartida** entre múltiples **hilos [j]**.

[j] https://doc.rust-lang.org/stable/book/ch16-03-shared-state.html

Sintaxis:

- Importar la librería estándar: **use std::sync::Mutex;**
- Crear un dato protegido con un mutex: **Mutex::new(T)**
- Adquirir el candado: **lock()**
- Acceder a un mutex adquirido: *
- Liberar el recurso adquirido: Salir del scope



***Nota:** Dado que **lock()** maneja tanto el **mutex** como el **error**, por lo tanto, se usa el método **lock().unwrap()** para obtener el **mutex** y manejar automáticamente el **error**.

***Nota:** El mutex contiene una **referencia** al **dato;** por ende, se ha de **desreferenciar** para acceder al **dato** que protege.

MECANISMOS DE CONCURRENCIA

Contador de referencias atómicas (ARC):

En **Rust**, cuando se usa una **función** para ejecutarse en **hilos**, se da **propiedad** de todos los **recursos** usados en la función; por ende, utiliza la **referencia compartida** en los **mutex** para su uso en **múltiples hilos [j]**.

Sintaxis:

- Importar la librería estándar: `use std::sync::Arc;`
- Crear un dato compartido con referencias atómicas: `Arc::new(T)`
- Adquirir una referencia del dato: `Arc::clone(&var)`
- Acceder al dato: *

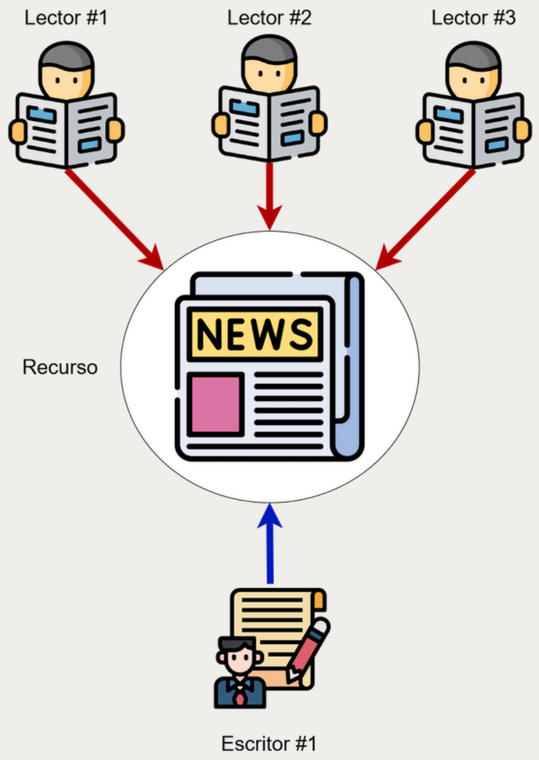
[j] https://doc.rust-lang.org/stable/book/ch16-03-shared-state.html

***Nota:** Dado que se usará en conjunto con **mutex** y no con un **primitivo simple**, no se ha de **desreferenciar** el dato; se puede acceder a los métodos de **mutex** normalmente.

MECANISMOS DE CONCURRENCIA

Bloqueos (Mutex) + ARC:

Ejemplo:



Todos los involucrados quieren **acceder al recurso**, pero es posible que el **escritor** cambie el contenido a medida que un **lector** lo vaya consultando.

Por ello se usa **Mutex** para que solo un usuario **acceda** al recurso a la vez.

MECANISMOS DE CONCURRENCIA

Bloqueos (Mutex) + ARC:

Ejemplo:

Definicion del lector

```rust
%%rust
use std::sync::{Arc, Mutex};
use std::thread;
use std::time::Duration;

fn lector(id: usize, dato: Arc<Mutex<i32>>) {
    for _ in 0..3 {
        let num = dato.lock().unwrap();

        println!("Lector {} lee valor: {}", id, *num);
        thread::sleep(Duration::from_millis(500));
    }
}
```

Definicion del escritor

```rust
fn escritor(dato: Arc<Mutex<i32>>) {
    for _ in 1..=3 {
        {
            let mut num = dato.lock().unwrap();

            *num += 10;
            println!("Escritor modifica valor a: {}", *num);
        }

        thread::sleep(Duration::from_millis(700));
    }
}
```

MECANISMOS DE CONCURRENCIA

Bloqueos (Mutex) + ARC:

Ejemplo:

```rust
fn main() {
    let dato = Arc::new(Mutex::new(0));

    let mut handles = vec![];

    // Lectores
    for i in 0..3 {
        let dato_clonado = Arc::clone(&dato);
        let handle = thread::spawn(move || lector(i, dato_clonado));
        handles.push(handle);
    }

    // Escritor
    let dato_clonado = Arc::clone(&dato);
    let handle_escritor = thread::spawn(move || escritor(dato_clonado));

    handles.push(handle_escritor);

    // Esperar todos los hilos
    for handle in handles {
        handle.join().unwrap();
    }
}
```

```
...     Compiling rust_playground v0.1.0 (/content/rust_playground)
        Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.29s
         Running `target/debug/rust_playground`
Lector 0 lee valor: 0
Lector 0 lee valor: 0
Lector 0 lee valor: 0
Escritor modifica valor a: 10
Lector 1 lee valor: 10
Lector 1 lee valor: 10
Lector 1 lee valor: 10
Escritor modifica valor a: 20
Lector 2 lee valor: 20
Lector 2 lee valor: 20
Lector 2 lee valor: 20
Escritor modifica valor a: 30
```

MECANISMOS DE CONCURRENCIA

Cola de mensajes (Channels):

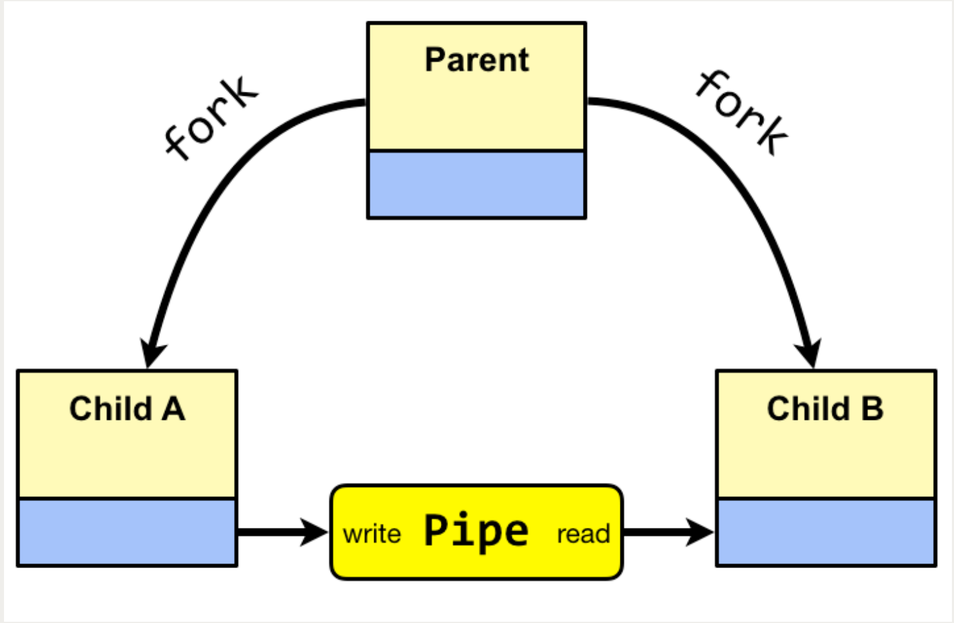
En **Rust**, los **channels** son mecanismos de comunicación entre **hilos** que permiten **enviar y recibir** datos de forma segura en entornos **concurrentes**, evitando la necesidad de usar **Mutex<T>** para compartir memoria **[g]**.

Sintaxis:

- Importar la librería estándar: **use std::sync::mpsc;**
- Crear un canal de comunicación: **mpsc::channel()**
- Enviar datos por el canal: **send(T)**
- Recibir datos del canal: **recv()**

***Nota:** Es importante tener en cuenta que, al crear el **canal**, se retornan dos **variables**. Uno para la **transmisión** y otro para la **recepción**.

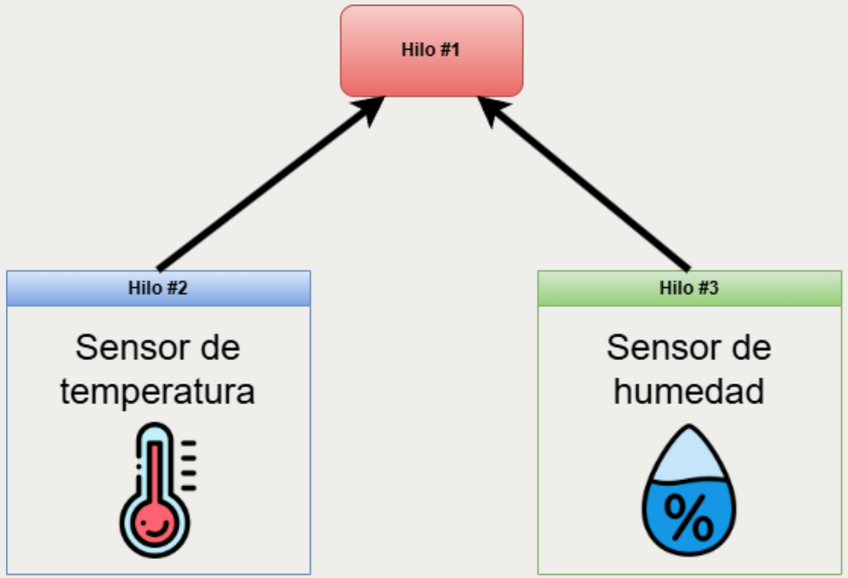
https://doc.rust-lang.org/stable/book/ch16-02-message-passing.html



MECANISMOS DE CONCURRENCIA

Cola de mensajes (Channels):

Ejemplo:



El hilo principal crea **dos hilos concurrentes** encargados de ejecutar continuamente **tareas de sensado**.

Cada vez que uno de los hilos **captura** un nuevo **dato**, este es **mostrado** en consola y posteriormente **enviado** al hilo principal mediante un **channel**, permitiendo la comunicación segura entre **hilos** sin necesidad de **compartir memoria** directamente.

MECANISMOS DE CONCURRENCIA

Cola de mensajes (Channels):

Ejemplo:

```
%%rust
use std::sync::mpsc;
use std::thread;
use std::time::Duration;

fn sensor_temperatura(tx: mpsc::Sender<(String, i32)>) {
    let temperaturas = [22, 24, 27, 31, 29];

    for temp in temperaturas {
        println!("[Sensor Temperatura {:?}] Nueva lectura: {}°C", thread::current().id(), temp);

        tx.send(("Temperatura".to_string(), temp)).unwrap();
        thread::sleep(Duration::from_millis(700));
    }
}
```

```
fn sensor_humedad(tx: mpsc::Sender<(String, i32)>) {
    let humedades = [40, 45, 50, 48, 43];

    for hum in humedades {
        println!("[Sensor Humedad {:?}] Nueva lectura: {}%", thread::current().id(), hum);

        tx.send(("Humedad".to_string(), hum)).unwrap();
        thread::sleep(Duration::from_millis(500));
    }
}
```

```
fn main() {
    let (tx, rx) = mpsc::channel();

    // Sensor temperatura
    let tx_temp = tx.clone();
    let h1 = thread::spawn(move || {
        sensor_temperatura(tx_temp);
    });

    // Sensor humedad
    let tx_hum = tx.clone();
    let h2 = thread::spawn(move || {
        sensor_humedad(tx_hum);
    });

    // Cerrar sender principal
    drop(tx);

    // Centro de monitoreo
    for (tipo, valor) in rx {
        println!("[CENTRAL] Dato recibido -> {}: {}", tipo, valor);
    }

    h1.join().unwrap();
    h2.join().unwrap();

    println!("--- Monitoreo finalizado ---");
}
```

MECANISMOS DE CONCURRENCIA

Cola de mensajes (Channels):

Ejemplo:

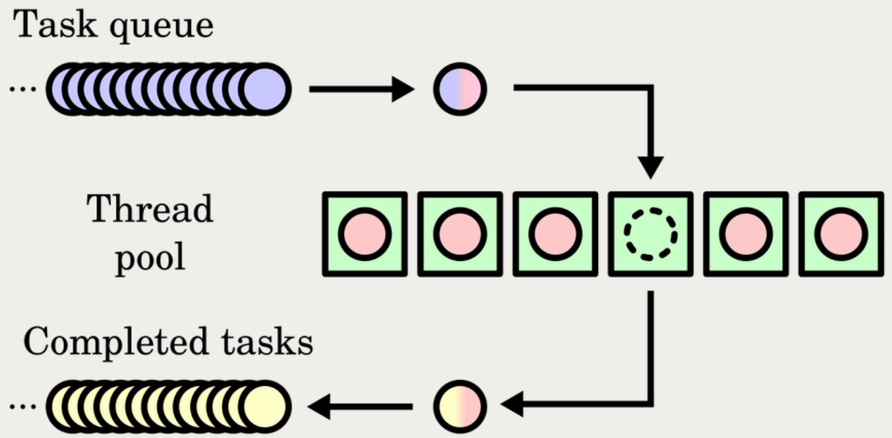
```
   Compiling rust_playground v0.1.0 (/content/rust_playground)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.27s
     Running `target/debug/rust_playground`
[Sensor Humedad ThreadId(3)] Nueva lectura: 40%
[Sensor Temperatura ThreadId(2)] Nueva lectura: 22°C
[CENTRAL] Dato recibido -> Humedad: 40
[CENTRAL] Dato recibido -> Temperatura: 22
[Sensor Humedad ThreadId(3)] Nueva lectura: 45%
[CENTRAL] Dato recibido -> Humedad: 45
[Sensor Temperatura ThreadId(2)] Nueva lectura: 24°C
[CENTRAL] Dato recibido -> Temperatura: 24
[Sensor Humedad ThreadId(3)] Nueva lectura: 50%
[CENTRAL] Dato recibido -> Humedad: 50
[Sensor Temperatura ThreadId(2)] Nueva lectura: 27°C
[CENTRAL] Dato recibido -> Temperatura: 27
[Sensor Humedad ThreadId(3)] Nueva lectura: 48%
[CENTRAL] Dato recibido -> Humedad: 48
[Sensor Humedad ThreadId(3)] Nueva lectura: 43%
[CENTRAL] Dato recibido -> Humedad: 43
[Sensor Temperatura ThreadId(2)] Nueva lectura: 31°C
[CENTRAL] Dato recibido -> Temperatura: 31
[Sensor Temperatura ThreadId(2)] Nueva lectura: 29°C
[CENTRAL] Dato recibido -> Temperatura: 29
--- Monitoreo finalizado ---
```

MECANISMOS DE CONCURRENCIA

Programa asíncrono (async + await):

En **Rust**, **async** y **await**, mediante la librería **Tokio**, permiten ejecutar **tareas concurrentes [t]**. Tokio facilita la **comunicación segura** entre tareas usando **canales asíncronos**, evitando compartir memoria directamente con **Mutex<T>**.

https://doc.rust-lang.org/stable/book/ch17-01-futures-and-syntax.html



Sintaxis:

- Importar tokio: **use tokio;**
- Declarar una función asíncrona: **async fn**
- Esperar una operación asíncrona: **.await**
- Crear una tarea concurrente: **tokio::spawn()**

***Nota:** Es importante tener en cuenta que se ha de usar la línea especial **#[tokio::main]** para ejecutar el **runtime asíncrono**.

MECANISMOS DE CONCURRENCIA

Programa asíncrono (async + await):

Ejemplo:

```rust
#[tokio::main]
async fn main() {
    // Inicia medición de tiempo
    let inicio = Instant::now();

    let mut tareas = Vec::new();

    // Crear 100 000 tareas concurrentes
    for i in 0..100_000 {
        tareas.push(tokio::spawn(cliente(i)));
    }

    // Esperar a que todas terminen
    for tarea in tareas {
        tarea.await.unwrap();
    }

    // Tiempo total transcurrido
    let duracion = inicio.elapsed();

    println!("Todos los clientes terminaron");
    println!("Tiempo total: {:?}", duracion);
}
```

```rust
use tokio::time::{sleep, Duration, Instant};

async fn cliente(id: u32) {
    println!("Cliente {} conectado", id);
    sleep(Duration::from_millis(100)).await;
    println!("Cliente {} desconectado", id);
}
```

```
Cliente 2999331 desconectado
Cliente 2999332 desconectado
Cliente 2999337 desconectado
Cliente 2999334 desconectado
Cliente 2999335 desconectado
Cliente 2999336 desconectado
Cliente 2999339 desconectado
Cliente 2999338 desconectado
Cliente 2999340 desconectado
Todos los clientes terminaron
Tiempo total: 75.389178115s
```

El modelo de **Thread pool** es útil para realizar **tareas cortas** de manera **concurrente** sin recurrir al **overhead** masivo de declarar múltiples **hilos de kernel**.

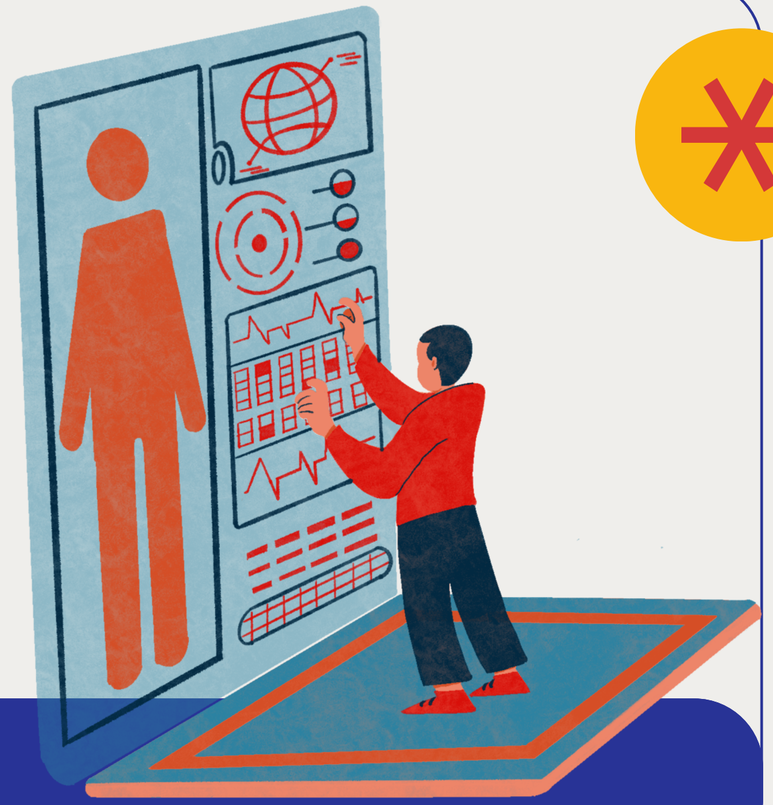
VENTAJAS Y DESVENTAJAS

VENTAJAS PRINCIPALES

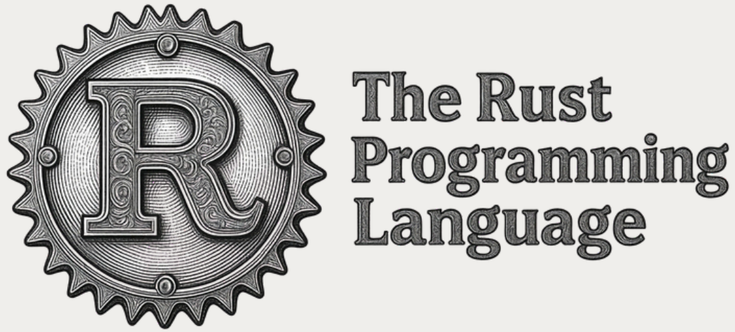
Seguridad + Rendimiento
Prevención de bugs → Reduce significativamente vulnerabilidades
Concurrencia confiable → Productividad en sistemas paralelos
Ecosistema creciente → Crates.io, adopción industrial

DESVENTAJAS REALES

Curva de aprendizaje empinada → 3-6 meses para productividad
Tiempos de compilación lentos (mejorando)
Complejidad cognitiva → Lifetimes, trait bounds
Ecosistema inmaduro en algunos dominios (GUI, async)



CONCLUSIONES



- **Paradigma transformador:** Seguridad de memoria sin garbage collector
- **Garantías en compilación**: Errores atrapados antes de producción
- **Concurrencia sin miedo**: Data races imposibles en código seguro
- **Rendimiento C/C++**: Sin sacrificar expresividad moderna

RUST NO SOLO ES UN LENGUAJE ES UNA PRUEBA DE CONCEPTO DE QUE PODEMOS TENER SEGURIDAD Y RENDIMIENTO SIMULTÁNEAMENTE[1]

MUCHAS GRACIAS