

Técnicas de Programación Concurrente I

Programación Asíncrona

Ing. Pablo A. Deymonnaz
pdeymon@fi.uba.ar

Facultad de Ingeniería
Universidad de Buenos Aires



Si una aplicación va creando muchos threads, cada uno puede tener 100kb de stack.

Puede ser un problema la demanda de memoria.

Tareas asincrónicas de Rust

- ▶ Se puede usar **Tareas asincrónicas de Rust** para intercalar tareas en un único thread o en un pool de threads.
- ▶ Son mucho más livianas que los threads.
- ▶ Más rápidas de crear, más eficiente de pasarle el control a ellas.
- ▶ Menor overhead de memoria.

→ se puede tener miles o decenas de miles en un programa.
El código asincrónico luce como el de threads, salvo que las operaciones que bloquean, se manejan diferente.

Ejemplo versión sincrónica:

```
use std::{net, thread};
let listener = net::TcpListener::bind(address)?;

for socket_result in listener.incoming() {
    let socket = socket_result?;
    let groups = chat_group_table.clone();
    thread::spawn(|| {
        log_error(serve(socket, groups));
    });
}
```

Tareas asincrónicas de Rust (cont.)

Ejemplo versión **asincrónica**:

```
use async_std::{net, task};
let listener = net::TcpListener::bind(address).await?;

let mut new_connections = listener.incoming();
while let Some(socket_result) = new_connections.next().await {
    let socket = socket_result?;
    let groups = chat_group_table.clone();
    task::spawn(async {
        log_error(serve(socket, groups).await);
    });
}
}
```

Conceptos de programación asincrónica

- ▶ *futures*
- ▶ funciones asincrónicas
- ▶ expresiones *await*
- ▶ tareas (*tasks*)
- ▶ executors: *block_on* y *spawn_local*
- ▶ tipo *Pin*

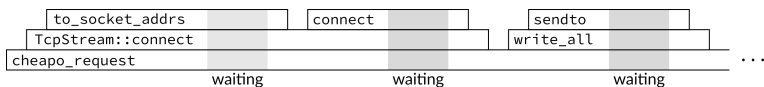
Ejemplo sincrónico

```
use std::io::prelude::*;
use std::net;

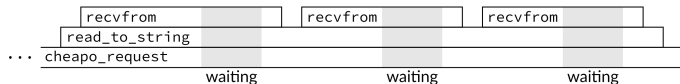
fn cheapo_request(host: &str, port: u16, path: &str)
-> std::io::Result<String>
{
    let mut socket = net::TcpStream::connect((host, port))?;
    let request =
        format!("GET {} HTTP/1.1\r\nHost: {}\r\n\r\n", path, host);
    socket.write_all(request.as_bytes())?;
    socket.shutdown(net::Shutdown::Write)?;

    let mut response = String::new();
    socket.read_to_string(&mut response)?;
    Ok(response)
}
```

Ejecución de las funciones en el tiempo



(continued from above)



La mayor parte del tiempo, el thread principal está bloqueado a la espera de system calls.

Para evitar esto, un thread debe poder tomar otras tareas mientras espera que la system call se complete.

Rust introduce el trait `std::future::Future`:

```
trait Future {
    type Output;
    // por ahora, interpretar 'Pin<&mut Self>'
    // como '&mut Self'.
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>)
        -> Poll<Self::Output>;
}

enum Poll<T> {
    Ready(T),
    Pending,
}
```

Futures (cont.)

- ▶ Representa una operación sobre la que se puede testear si se completó.
- ▶ El método *poll* nunca bloquea.
- ▶ Si la operación se completó, retorna: *Poll::Ready(output)* (*output* es el resultado final de la operación).
- ▶ Si no se completó, retorna *Pending*.
- ▶ Modelo **piñata** de la programación asincrónica: lo único que se puede hacer con un future es golpearlo con *poll* hasta que caiga el valor.
- ▶ SO proveen system calls como interfaz para hacer *poll*.

Futures (cont.)

Versión sincrónica:

```
fn read_to_string(&mut self, buf: &mut String)
    -> std::io::Result<usize>;
```

Versión asincrónica:

```
fn read_to_string(&mut self, buf: &mut String)
    -> impl Future<Output = Result<usize>>;
```

Futures (cont.)

- ▶ Cada vez que es pollado, avanza todo lo que puede.
- ▶ El Future almacena lo necesario para realizar el pedido hecho por la invocación.
- ▶ Crate **async-std**: Provee versiones async de las facilidades de I/O de la std (incluyendo un trait Read asincrónico).

Performance:

- ▶ La arquitectura async de Rust está diseñada para ser eficiente.
- ▶ Se llama a poll solamente cuando vale la pena (algo debe retornar *Ready*, o progresar al objetivo).

Funciones Async y Expresiones Await

```
use async_std::io::prelude::*;
use async_std::net;

async fn cheapo_request(host: &str, port: u16, path: &str)
-> std::io::Result<String>
{
    let mut socket =
        net::TcpStream::connect((host, port)).await?;
    let request =
        format!("GET {} HTTP/1.1\r\nHost: {}\r\n\r\n", path, host);
    socket.write_all(request.as_bytes()).await?;
    socket.shutdown(net::Shutdown::Write)?;
    let mut response = String::new();
    socket.read_to_string(&mut response).await?;
    Ok(response)
}
```

Funciones Async y Expresiones Await

- ▶ Invocar una función *async* retorna inmediatamente, antes de que comience a ejecutarse el cuerpo de la función.
- ▶ Se obtiene un *Future* del valor, que contiene todo lo necesario para que la función pueda ejecutarse (argumentos, espacio para variables locales, etc).
- ▶ El tipo específico es generado al momento de compilar.
- ▶ Al ejecutar `poll` por primera vez sobre el retorno, se ejecuta el cuerpo de la función hasta el primer `await`.
- ▶ Si no se completó, retorna *Pending* y toda la función devuelve ese valor/estado.
- ▶ La expresión `await` toma ownership del future y hace el `poll`.
- ▶ Si está *Ready*, el valor final del future es el valor devuelto en la expresión `await`, y continúa.
- ▶ Caso contrario, retorna *Pending* a su función que lo invocó.

Funciones Async y Expresiones Await

- ▶ La siguiente invocación a `poll` sobre la función `cheapo_request`, continuará desde el punto donde estaba el `future connect`.
- ▶ El `Future` almacena el punto donde debe retomarse en el siguiente `poll` y el estado local.
- ▶ Las expresiones `await` tienen la capacidad de continuar, se pueden usar solamente en funciones `sync`.

La función `async` retorna un `Future`, alguien debe esperar por el valor.

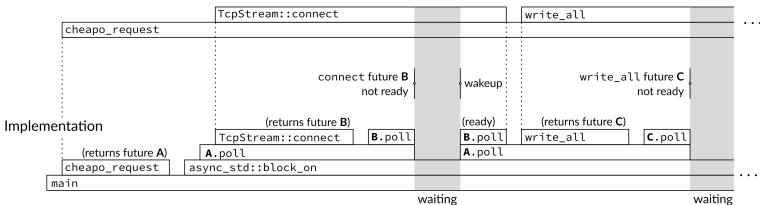
```
fn main() -> std::io::Result<()> {  
    use async_std::task;  
  
    let response =  
        task::block_on(cheapo_request("example.com", 80, "/"))?;  
    println!("{}", response);  
    Ok(())  
}
```

block_on

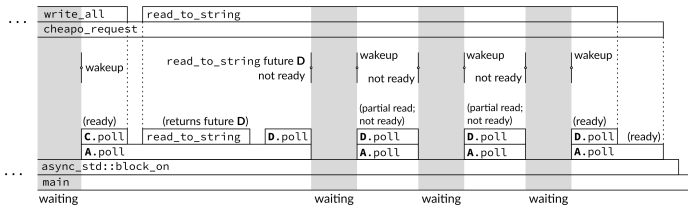
- ▶ **block_on** es una función sincrónica que produce el valor final de la función asincrónica.
- ▶ Es un adaptador del mundo asincrónico al sincrónico.
- ▶ No debe usarse en una función async (bloquea a todo el thread).
- ▶ **block_on** conoce cuánto hacer sleep hasta hacer poll de nuevo.

block_on

Simplified view



(continued from above)



Crear tareas async

- ▶ `async_std::task::spawn_local` recibe un future y lo agrega a un pool que realizará polling en el `block_on` (soportado solamente en *unstable* del crate). Es análogo a `spawn` de `thread`.
- ▶ Los lifetimes de las variables deben ser `static`, porque debe poder ejecutarse hasta el final del programa.
- ▶ Todas las ejecuciones pueden realizarse en un único thread. Una llamada asíncrona ofrece la apariencia de una única llamada a función que se ejecuta hasta que se completa. Pero es realizada por una serie de llamadas sincrónicas al método `poll`, que retorna rápidamente, hasta que se completa.

Crear tareas async

- ▶ El cambio de una tarea a otra ocurre solamente en las expresiones **await** → un cómputo grande en una función no daría lugar a la ejecución de otras tareas (diferencia con threads).
- ▶ Existe **async_std::task::spawn**: crea la tarea y la coloca en el pool de threads dedicado a hacer poll de los futures.
- ▶ No necesita ejecutar **block_on** para que sea polleada.

- ▶ **`async_std::task::yield_now`** favorece el paralelismo. Retorna un future para pasar el control a otra tarea.
- ▶ Se usa: `async_std::task::yield_now().await;`
- ▶ La primera vez que se hace poll, retorna Pending, la siguiente vez retorna Ready(()).
- ▶ **`async_std::task::spawn_blocking`**: coloca la tarea en otro thread del SO (para realizar un cómputo pesado), dedicado a tareas bloqueantes.

Cuándo usar código Async en Rust

- ▶ Las tareas asincrónicas usan menos memoria. Los threads de Linux pueden usar desde 20 kb de memoria.
- ▶ Las tareas asincrónicas se crean más rápido. En Linux, threads 15 μ s, tareas 300 ns.
- ▶ Los cambios de contexto son más rápidos con tareas asincrónicas que con threads.

- ▶ **Programming Rust: Fast, Safe Systems Development**, 2nd Edition, Jim Blandy, Jason Orendorff, Leonora F. S. Tindall. 2021.