Concurrent programming in Rust

4SE05

Léopold Clément

Télécom Paris

2025-04-04

Outline

Concurency in Rust

Why use Rust

> *For example, introducing parallelism in Rust is a relatively low-risk operation: the compiler will catch the classical mistakes for you.*
>
> *— The rust book, foreword*

> *By leveraging ownership and type checking, many concurrency errors are compile-time errors in Rust rather than runtime errors.*
>
> *— The rust book, chapter 16*

Two types of concurrency

Two types of concurrency

- Threaded concurrency
  - Based on threads, like in C
  - Good for compute-bound workloads,

Two types of concurrency

- Threaded concurrency
  - Based on threads, like in C
  - Good for compute-bound workloads,
- Asynchronous concurrency
  - Based on futures / promise / task / async functions
  - Seen in Python, JavaScript/TypeScript
  - Good for IO-bound workloads,
  - Used in embassy (cf 4SE02)

Concurency in Rust

Safty guarantiees in Rust

The borrow checker

The borrow checker quarantees that there is either only one mutable reference to a value, or multiple inmutable.

The borrow checker

The borrow checker quarantees that there is either only one mutable reference to a value, or multiple inmutable.

Raw object cannot be shared between thread and be mutable.

The borrow checker

The borrow checker quarantees that there is either only one mutable reference to a value, or multiple inmutable.

Raw object cannot be shared between thread and be mutable.

To share object between threads and have mutability, you need to use some synchronisation primitive.

The borrow checker

The borrow checker quarantees that there is either only one mutable reference to a value, or multiple inmutable.

Raw object cannot be shared between thread and be mutable.

To share object between threads and have mutability, you need to use some synchronisation primitive.

Sharing value with `Rc`

`Rc` is a smart pointer.

It can be cloned, and count the number of time was cloned.

It allow for an value to have a lifetime long enough for all the time it is used.

```
fn rc_to_thread() {
    let a = Rc::from(1);
    let b = a.clone();
    let j = spawn(move || println!("val = {}", *b));
    j.join();
}
```

Sharing value with Rc, errors!

```
error[E0277]: `Rc<i32>` cannot be sent between threads safely
   --> src/main.rs:90:19
    |
90  |     let j = spawn(move || println!("val = {}", *b));
    |             ----- -------^^^^^^^^^^^^^^^^^^^^^^^^^
    |             |     |
    |             |     `Rc<i32>` cannot be sent between threads safely
    |             |     within this `{closure@src/main.rs:90:19: 90:26}`
    |             required by a bound introduced by this call
    |
   = help: within `{closure@src/main.rs:90:19: 90:26}`, the trait `Send`
is not implemented for `Rc<i32>`
```

The `Send` and `Sync` trait

The compiler implements those trait when possible :

The `Send` and `Sync` trait

The compiler implements those trait when possible :

`Send` The value can be moved between threads.

The `Send` and `Sync` trait

The compiler implements those trait when possible :

`Send` The value can be moved between threads.

`Sync` Reference to the value can be moved between threads.

Sharing value with `Arc`

To share the ownership of a value of type `T` across threads boundaries, you might use `Arc<T>` instead of `Rc<T>`.

Sharing value with `Arc`

To share the ownership of a value of type `T` across threads boundaries, you might use `Arc<T>` instead of `Rc<T>`.

Here the counter is atomic, so the compiler add the `Send` trait and the `Arc` can be moved between threads.

Sharing value with `Arc`

To share the ownership of a value of type `T` across threads boundaries, you might use `Arc<T>` instead of `Rc<T>`.

Here the counter is atomic, so the compiler add the `Send` trait and the `Arc` can be moved between threads.

Shared ownership does not permit mutability, `Arc` can only be deref into inmutable reference.

Primitives

Most of the primitives are defined in the `std` module.

Primitives

Most of the primitives are defined in the `std` module.

- std
  - atomics
  - thread
  - sync
    - mutex
    - channel
  - future
  - task

Primitives

Most of the primitives are defined in the `std` module.

- std
  - atomics
  - thread
  - sync
    - mutex
    - channel
  - future
  - task

You can build more complex primitives from basic primitives, or using unsafe block.

Atomics

Atomics lives in `std::sync::atomics`.

```
use std::sync::atomic::{AtomicU16, Ordering};

let atomic = AtomicU16::new(0);
assert_eq!(atomic.load(Ordering::Relaxed), 0);
atomic.store(1, Ordering::Relaxed);
assert_eq!(atomic.fetch_add(10, Ordering::Relaxed), 1);
assert_eq!(atomic.swap(100, Ordering::Relaxed), 11);
assert_eq!(atomic.load(Ordering::Relaxed), 100);
```

Atomics

Atomics lives in `std::sync::atomics`.

```
use std::sync::atomic::{AtomicU16, Ordering};

let atomic = AtomicU16::new(0);
assert_eq!(atomic.load(Ordering::Relaxed), 0);
atomic.store(1, Ordering::Relaxed);
assert_eq!(atomic.fetch_add(10, Ordering::Relaxed), 1);
assert_eq!(atomic.swap(100, Ordering::Relaxed), 11);
assert_eq!(atomic.load(Ordering::Relaxed), 100);
```

The ordering is the same as in C++ 20

Atomics : Ordering

`Relaxed`: only guarantee atomicity and modification order consistency,

`AcqRel`: also, things that append before/after a store in one threads append before/after a read in another thread,

`SeqCst`: also, there is a single total modification order of all atomic operations that are so tagged

Atomics : Creating a mutex

```
pub struct SpinLockMutex{
    locked : AtomicBool,
}
impl SpinLockMutex {
    pub fn new() -> Self{
        SpinLockMutex { locked: AtomicBool::new(false) }
    }
    pub fn lock(&self) -> (){
        loop {
            let old_state = AtomicBool::swap(&self.locked, true, Ordering::AcqRel);
            if !old_state {return;}
        }
    }
    pub fn unlock(&self) -> (){
        AtomicBool::store(&self.locked, false, Ordering::AcqRel);
    }
}
```

Interior Mutability

```rust
let atomic = AtomicU16::new(0);

pub fn swap(&self, val: bool, order: Ordering) -> bool
pub fn lock(&self) -> ()
pub fn unlock(&self) -> ()
```

Interior Mutability

```rust
let atomic = AtomicU16::new(0);

pub fn swap(&self, val: bool, order: Ordering) -> bool
pub fn lock(&self) -> ()
pub fn unlock(&self) -> ()
```

In Rust, some type allows its value to be mutated even on not mutable instances.

Interior Mutability with `Cell`

`Cell<T>` is a container used to create interior mutability.

```
pub fn set(&self, val: T)
pub fn replace(&self, val: T) -> T
pub fn swap(&self, other: &Cell<T>)
```

`Cell` is not `Send` : it is not atomic from other threads, only from its own thread.

Concurrent primitive for threaded concurrency

Concurrent primitive for threaded concurrency

Threads

Thread model

Rust threads are os threads.

Thread model

Rust threads are os threads.

- If the main thread panics, the program stops.

Thread model

Rust threads are os threads.

- If the main thread panics, the program stops.
- If another thread panics, nothing happens.

Thread model

Rust threads are os threads.

- If the main thread panics, the program stops.
- If another thread panics, nothing happens.
- If the main thread terminates before the other, the other threads are stopped.

Kinds of functions in Rust

`FnOnce` Can be called once.

Kinds of functions in Rust

`FnOnce` Can be called once.

`Fn` Can be called multiples time.

Kinds of functions in Rust

`FnOnce` Can be called once.

`Fn` Can be called multiples time.

`FnMut` Can be called multiples time and mutate borrowed value.

Kinds of functions in Rust

`FnOnce` Can be called once.

`Fn` Can be called multiples time.

`FnMut` Can be called multiples time and mutate borrowed value.

```
let mut x = 0;
let add = |y| {x + y}; //Fn
let mut inc_by = |y| {x += y}; //FnMut

x: String = String::from("Léopold");
let concat_to_x = |y: String| add(x, &y); //FnOnce
                                          // x is consumed by add
```

Spawning and joining a thread

```rust
let t = std::thread::spawn(move || {
    println!("Hi from the thread");
});
println!("Hi from main");
t.join().unwrap();
```

Spawning and joining a thread

```rust
let t = std::thread::spawn(move || {
    println!("Hi from the thread");
});
println!("Hi from main");
t.join().unwrap();
```

Always `join` a thread.

Move value inside threads, wrong

```
let name = Arc::new("Leopold");
let hello_from_inside =|x:usize, name| {println!("Hi {} from {}", name,
x);};
let t1 = spawn(move || hello_from_inside(1, name));
let t2 = spawn(move || hello_from_inside(2, name));
```

```
error[E0382]: use of moved value: `name`
```

Move value inside threads, correct

Il faut cloner explicitement les `Arc`

```
    let name = Arc::new("Leopold");
    let hello_from_inside =|x:usize, name| {println!("Hi {} from {}",
name, x);};
    let tmp = name.clone(); // clone the Arc
    let t1 = spawn(move || hello_from_inside(1, tmp)); //tmp move, not
name
    let tmp = name.clone();
    let t2 = spawn(move || hello_from_inside(2, tmp));
```

Communication between threads

There are two kind of communication:

Communication between threads

There are two kind of communication:

- Shared state
  - use `mutex`
  - every threads can read/write

Communication between threads

There are two kind of communication:

- Shared state
  - use `mutex`
  - every threads can read/write
- Message passing
  - use `channel`
  - some threads are writer, some are reader

Concurrent primitive for threaded concurrency

Mutex

Usecase

A `mutex` allow to share a value between thread. Only one thread may access the value at a time.

Usecase

A `mutex` allow to share a value between thread. Only one thread may access the value at a time. In Rust, `Mutex` is a wrapper around a non-atomic value. Each thread has a reference to the `Mutex`, but only on at a time has a reference to the value.

Lifecycle of a Mutex

```
let mut threads : Vec<JoinHandle<_>> = Vec::new();
let counter = Arc::new(Mutex::new(0));
for idx_thread in 0..N_MAX{
    let counter = counter.clone();
    threads.push(spawn(move || {
        for _ in 0..idx_thread{
            let mut c = counter.lock().unwrap();
            *c += 1;}
      }));
    }
for t in threads.into_iter(){t.join().unwrap();}
assert_eq!(*counter.lock().unwrap(), (N_MAX-1)*(N_MAX)/2);
```

Taking a Mutex

```
fn lock(&self) -> LockResult<MutexGuard<'_, T>>;
fn try_lock(&self) -> TryLockResult<MutexGuard<'_, T>>;

LockResult<T> = Result<T, PoisonError<T>>;
TryLockResult<T> = Result<T, TryLockError<T>>;
```

The mutex might be poisoned, and can’t be taken.

Taking a Mutex

```
fn lock(&self) -> LockResult<MutexGuard<'_, T>>;
fn try_lock(&self) -> TryLockResult<MutexGuard<'_, T>>;
```

```
LockResult<T> = Result<T, PoisonError<T>>;
TryLockResult<T> = Result<T, TryLockError<T>>;
```

The mutex might be poisoned, and can’t be taken.

There are two interfaces :

- blocking : `lock`
- non-blocking : `try_lock`

Releasing a Mutex

The mutex is released a the end of the context where it was locked.

You can also use `std::mem::drop`.

Releasing a Mutex

The mutex is released a the end of the context where it was locked.

You can also use `std::mem::drop`.

```
for _ in 0..idx_thread{
  let mut c = counter.lock().unwrap();
  *c += 1;
  } // counter is released here
```

Poisoned Mutex

If a thread panic while holding a mutex, the mutex will be poisoned. A poisoned mutex can not be taken by anyone.

A mutex can be tested for poison with `is_poisoned` and cured with `clear_poison`. This only affect the mutex, the inner data might still be corrupted.

Concurrent primitive for threaded concurrency

Channel `mpsc`

Usecase

A `Channel` allow to pass messages between context.

The base implementation allow multiple writer and one reader. The channel has an infinite capacity.

Creating a channel

A channel for value of type `T` is represented by its ends, the `Sender<T>` (`tx`) and the `Receiver<T>` (`rx`).

```
let (tx, rx) = std::sync::mpsc::channel();
```

The `Sender` can be cloned, to create multiple producer.

Sending through the channel

```
fn send(&self, t: T) -> Result<(), SendError<T>>
```

Return an error only if the `rx` is dropped. `Ok` doesn’t mean that the message is received. This will never block.

Receiving through the channel

```
fn recv(&self) -> Result<T, RecvError>
```

Will block if no message in the channel. Will return an `Err` if the `Sender` is disconnected.

Receiving through the channel

```
fn recv(&self) -> Result<T, RecvError>
```

Will block if no message in the channel. Will return an `Err` if the `Sender` is disconnected.

There is a non-clocking interface, `try_recv` and a timout interface `recv_timeout`.

```
fn recv_timeout(&self, timeout: Duration) -> Result<T, RecvTimeoutError>
fn try_recv(&self) -> Result<T, TryRecvError>
```

Receiving through the channel

```
fn recv(&self) -> Result<T, RecvError>
```

Will block if no message in the channel. Will return an `Err` if the `Sender` is disconnected.

There is a non-clocking interface, `try_recv` and a timout interface `recv_timeout`.

```
fn recv_timeout(&self, timeout: Duration) -> Result<T, RecvTimeoutError>
fn try_recv(&self) -> Result<T, TryRecvError>
```

The `Receiver` can also be turned into a iterator:

```
for msg in rx.iter() {
  ...
}
```

Async/await, `Future` and cooperative concurrence

Async/await, `Future` and cooperative concurrence

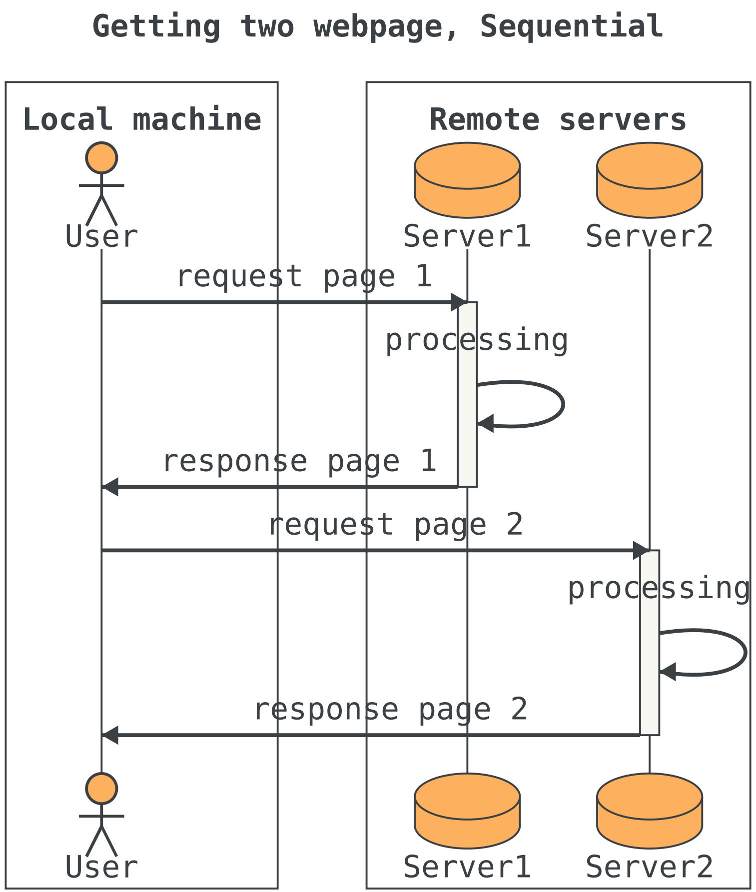
Getting two pages at the same time

Problem stattement

How to get the content of two web page ?

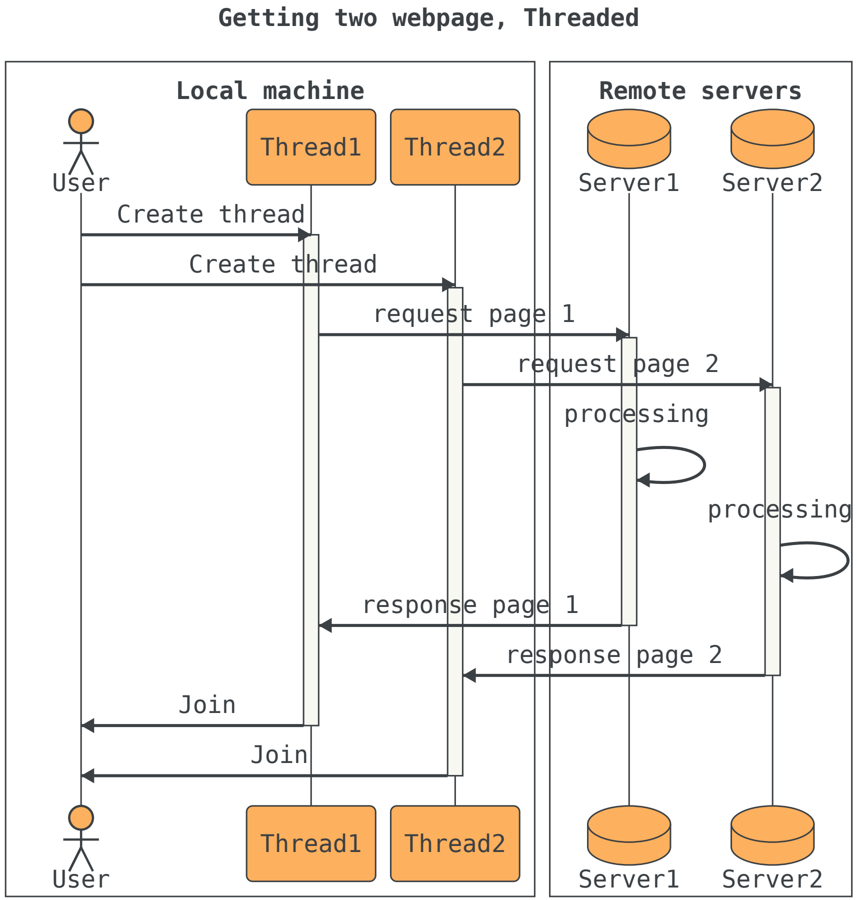
- as fast as possible
- without using too much ressources

Sequential



- simple
- slow
- use only one stack

Threaded



- use the os for context switching
- fast
- use three stacks

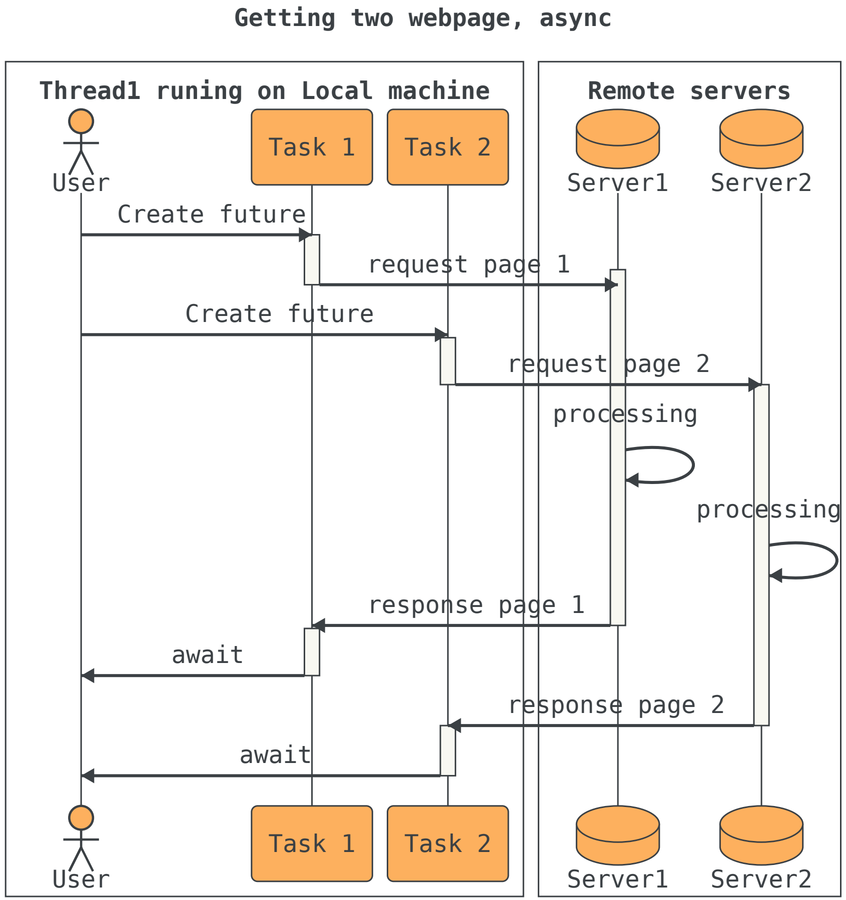
Issue with threads

- Compute vs IO bound:
  - here, we are IO-bound
  - adding compute time through threads is not usefull

Issue with threads

- Compute vs IO bound:
  - here, we are IO-bound
  - adding compute time through threads is not usefull
- Scaling to many tasks:
  - one thread by tasks $\Longrightarrow$ one stack by task $\Longrightarrow$ heavy memory usage
  - could we use only one the main thread and distribute compute time to tasks ?

Asynchronous tasks



- use a local algorithm for context switching
- fast
- use one stack

Task, future, executor

- tasks, and their sub-tasks:
  - are represented as the promise of a future value (that value can be the unit type if ther is no return value)
  - can do a little progress at a time, stoping when they can not progress anymore or are finised
- an algorithm, the executor:
  - run on the main thread
  - distribute compute time at each task

Async/await, `Future` and cooperative concurrence

Implementation in Rust

Futures

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

The context of a task is stored in the other member of the struct. `poll` must be nonblocking.

Resolving a future

- Resolving a future is polling it until it return `Ready`.
- `Future` implementation use a `Waker` to signal that the future is ready to be polled.
  - this allow for task to only wakeup when progress is possible.
- Unlike `Javascript` or `C#`, the task polling is done by the program, not by the runtime/vitual machine.

Implementing `Future`

- `Future` should be implemented as state machine,
- `Future` should not start their work before the first `poll`,
- `Future` should use the `Waker` to be called again,
- if a `Future` contain another `Future`, it should call the `poll` of the children each time it is called,
- `poll` must be quick

A delay future

```
struct Delay {when: Instant}
impl Future for Delay {
    type Output = &'static str;
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>)-> Poll<&'static str>
    {   if Instant::now() >= self.when {
            println!("Hello world");
            Poll::Ready("done")
        } else {
            let waker = cx.waker().clone();// Get a handle to the waker
            let when = self.when;
            thread::spawn(move || { // Spawn a timer thread.
                let now = Instant::now();
                if now < when {thread::sleep(when - now);}
                waker.wake();
            });
            Poll::Pending
      }}}
```

Async function

```rust
// fn slow_getter(&str) -> impl Future<i32>;
async fn slow_get_add(r: &str, n: i32) -> i32 {
  let r = get_slow(r).await;
  r + n
}
```

Async function

```rust
// fn slow_getter(&str) -> impl Future<i32>;
async fn slow_get_add(r: &str, n: i32) -> i32 {
  let r = get_slow(r).await;
  r + n
}
```

`async` marks the function, the compiler will turn it into the function `fn slow_getter(&str, i32) -> impl Future<i32>`. The associated structure implementiong `Future` will also be generated. The `.await` marks that the future should poll the `get_slow` future.

Multiple Futures to await

```
async fn get_plus_twice_bad(r1: &str, r2: &str, n: i32) -> (i32, i32) {
  let p1 = slow_get_add(r1, n).await;
  let p2 = slow_get_add(r2, n).await;
  (p1, p2)
}
async fn get_plus_twice(r1: &str, r2: &str, n: i32) -> (i32, i32) {
  let pair = join(slow_get_add(r1, n), slow_get_add(r2, n));
  pair.await
}
```

Futures will only start to execute if there are awaited/polled.

Executor

The executor is the object that handle polling the tasks.

Executor

The executor is the object that handle polling the tasks.

There is no runtime/executor in `std`. The main ones are:

- `Tokio` for hosted environement
  - include io async function for IO (network and filesystem)
  - awaitable synchronisation primitive (mutex, channel)
- `Embassy` for embedded environement
  - include async hardware abstraction layer

Tokio, main task

```rust
#[tokio::main]
async fn main() -> Result<()> {
    let mut client = client::connect("127.0.0.1:6379").await?;

    client.set("hello", "world".into()).await?;

    let result = client.get("hello").await?;

    println!("got value from the server; result={:?}", result);

    Ok(())
}
```

Tokio, creating tasks

```rust
#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    let listener = TcpListener::bind("127.0.0.1:8080").await?;
    loop {
        let (mut socket, _) = listener.accept().await?;
        let handle = tokio::spawn(async move {
            let mut buf = [0; 1024];
            loop {// In a loop, read data from the socket and write the data back.
                let n = match socket.read(&mut buf).await {
                    Ok(0) => return, // socket closed
                    Ok(n) => n,
                    Err(e) => {
                        eprintln!("failed to read from socket; err = {:?}", e);
                        return;}};
                if let Err(e) = socket.write_all(&buf[0..n]).await {
                    eprintln!("failed to write to socket; err = {:?}", e);
                    return;
}}});}}
```

When spawned with `spawn`, tasks are eagerly executed.

Good practice for async

- No blocking in async functions!

Good practice for async

- No blocking in async functions!
- Do not hold a `mutex` across a `await`,

Good practice for async

- No blocking in async functions!
- Do not hold a `mutex` across a `await`,
- use the `join!` macro to await multiple future at the same time.
- use the crate `futures`:
  - the `join` futures to await multiple future at the same time,
  - an awaitable `Mutex`

Conclusion

Threads vs Futures

- Threads
  - Compute-bound program
  - expensive to start
  - can exploit multi-hart computer
  - relies on the OS
  - communication via channel or mutex
  - can block
- Futures
  - IO-bound program
  - cheap to start
  - might exploit multi-hart computer
  - relies on a user provided executor

Threads vs Futures

- should not block

Using the best of both world

Using the best of both world

- multithreaded executor :
  - use multiple thread to `poll` multiple future at the same time,
  - it is default in tokio,

Using the best of both world

- multithreaded executor :
  - use multiple thread to `poll` multiple future at the same time,
  - it is default in tokio,
- using a thread to resolve long calculation
  - the future create/acquire a thread, launch a calculation and return `pending`,
  - when the thread terminate, it wake the future and give the result
  - can also use a pool of thread

Questions ?