Asynchronous programming

SSP-RS — Safe System Programming (in Rust)

Samuel Tardieu Stefano Zacchiroli

2023-11-21

Blocking, non-blocking and asynchronous I/O

What does a typical program do?

A typical computer program can do one or more of the following:

- Input data from the user
- Output data to the user
- Read data from storage
- Write data to storage
- Read data from a network
- Send data to a network
- Compute something

Splitting common actions into tasks

Some actions are composed of several smaller tasks. For example:

- Read from storage: send the data address to the storage driver, *wait for the data to have been retrieved*, then read the data
- Query a database: send the request to the database, *wait for the result*, then read the result
- Make a HTTP request: send the request to the server, *wait for the result*, then read the result
- Answer a HTTP request: parse the request headers, *wait for the request body to be fully received*, compute the result and send it

One common denominator between those actions is: at some point there is a need to wait for data which is not instantly ready.

Waiting should not waste CPU cycles if the computer has other things to do.

Wait times and parallelism

Input/output (I/O) services are provided by the operating system. When a thread does a *blocking* request (a request whose result is not immediately available), the OS will mark it as *waiting* and will not schedule it until the result is available.

A multithreaded web server answering requests by querying a database might work as follow:

- The server binds a listening socket to the port it want to receive requests on (for example 443).
- The server **waits** for a connection to happen.
- When an incoming connection arrives, the server spawns a new thread to handle this connection (while itself resuming its listening):
  - The thread **waits** for the request to be sent.
  - The thread sends a query to the database server and **waits** for the result.
  - The thread sends the result to the connection, and resumes **waiting** for the next request.

On Linux, the default thread size is 10 MiB. If a server is popular, it might have to deal with tenths of thousands of simultaneous connections, meaning that as many threads would be created, each one with its stack $\Rightarrow$ this is not scalable.

Non-blocking I/O

On modern systems, every I/O operation happens through a *file descriptor*. A file descriptor can correspond to an open file, a network socket, a special peripheral (standard input and output), etc.

Reading from, or writing to a file descriptor are by default *potentially blocking* operations: if an operation cannot be completed immediately, the kernel suspends the thread, then resumes it when the operation has been completed.

File descriptors can be configured in **non-blocking** mode: all operations will return immediately, with error code `EWOULDBLOCK` if an operation is attempted and would result in blocking the calling thread.

A typical way to use non-blocking mode is:

- check whether the operation (send data, receive data) would be possible without blocking;
- then when it is, perform the operation.

The OS provides tools (`select()`, `poll()`) to check on the readiness of one or more file descriptors at the same time with regard to read and write operations.

Web server with non-blocking I/O

A web server using non-blocking I/O could:

- setup a `struct pollfd` object to retrieve the state of:
  - the socket it listens for new connections to arrive on (read mode)
  - all sockets it is waiting for data to arrive on (read mode)
  - all sockets it is waiting for data to be written to (write mode)
- `poll()` this object: the kernel will block the calling thread until one of the polled file descriptors is ready (some data can be read from or written to it), or until a *timeout* occurs
- do what is requested on the corresponding file descriptor (read and write some data, maybe not all, in which case the file descriptor will be polled again)
- start a new thread when actual work has to be done (for example to produce the content of a web page), or even do the work in the current thread if it is short enough

In this scenario, new threads are spawned only to do actual work, not for every incoming connection. Thread pools can even be used to avoid having too many threads alive at the same time.

However, it complicates the program structure compared to a more linear execution form.

How is non-blocking I/O done?

In C:

- A file can be opened with the `O_NONBLOCK` flag, or the flag can be set on any existing file descriptor `fd` using `fcntl(fd, SET_FL, O_NONBLOCK)`.
- The system calls `select()` and `poll()` can poll several file descriptors.
- Other OS-specific methods (`pselect()` and `ppoll()` on Linux, `kqueue` on BSD) offer finer grained control.

In Rust:

- Direct bindings can be used to interface with the C library's `select()` and `poll()`.
- The `mio` (*Metal I/O*) crate offers higher-level types and functions to perform those operations.

Asynchronous I/O

Non-blocking I/O is powerful, but tedious:

- one only knows that *some bytes* can be read or written from the file descriptor;
- the operation must be restarted with the rest of the data to read or write.

Asynchronous I/O is different:

- requests to read or write data always return immediately (as in non-blocking I/O);
- the I/O subsystem uses the provided buffer to store the received data or to get the data to write to the file descriptor;
- the requester can query whether an operation is terminated or in progress;
- the I/O subsystem can send a *signal* when the operation is complete (data fully read or fully written, or an error occurred);
- when an operation is complete, the requester can then use the data (in case of a read without error) or free the buffer (in case of a write).

Asynchronous I/O and callbacks

Asynchronous I/O are often used along with a callback subsystem. A simple design could be:

- A signal handler is in charge of running previously registered callbacks when an asynchronous operation terminates. It keeps a mapping of `operation` → `callback`.
- When the signal arises, for every key the signal handler checks which operations are complete and calls their callback, removing them from the mapping.
- Just before calling the asynchronous operation, the user registers a callback with the signal handler.

This way, any client of this callback subsystem can have some code execute when an asynchronous operation is complete even though all completions are materialized using a single signal.

Variations are possible, for example by registering two callbacks (in case of success and in case of error).

Asynchronous I/O and callbacks: an example

An example in Javascript using `node.js` shows a callback being registers for the `fs.readFile()` operation:[1]

```javascript
const fs = require("fs");
fs.readFile("/file.md", (err, data) => {
  if (err) throw err;
  console.log(data);
});
moreWork(); // will run before console.log
```

`fs.readFile()` takes care of registering the anonymous function given as second argument with the builtin signal handler before starting the operation.

[1] Source nodejs.org

Asynchronous I/O and callbacks: another example

Another example[2] shows how much chained asynchronous I/O operations disrupt the program flow:

```
const fs = require("fs");
fs.readFile("/file.md", (readFileErr, data) => {
  if (readFileErr) throw readFileErr;
  console.log(data);
  fs.unlink("/file.md", (unlinkErr) => {
    if (unlinkErr) throw unlinkErr;
  });
});
```

Callbacks are a great functionality, but force the programmer's (and the reader's) mind to switch into "parallel/asynchronous mode". Such programs are harder to review, and bugs may be harder to find when they happen in the middle of a callback chain.

[2] Also from nodejs.org

The fault is not Javascript’s, it is the callbacks’ one

Rust has a `Future` type to represent the result of a not-yet-ready operation. Combinators allow the programmer to use a callback-like style:

```rust
// Using fictitious asynchronous operations
let fut = open("/file.md")
    .or_else(|e| panic!("open error: {e}"))
    .and_then(|fd|
        fd.read_file()
            .or_else(|e| panic!("read_file error: {e}"))
            .and_then(|data|
                unlink("/file.md").or_else(|e| panic!("unlink error: {e}")).and_then(|_| data)
            )
    );
// Later, run request on an executor (here we only have one future to run, we could have
// several of them)
let data = executor.block_on(fut);
```

Again, callbacks are great, but they can make the code unreadable when deeply chained.

Asynchronous I/O pitfalls

In addition to the “callback chain mess”, there are other ways asynchronous I/O can go wrong:

- The user might reuse the buffer being asynchronously written, for example to prepare the next request.
- The user might read incomplete data from a buffer, before the read operation has terminated.

Those pitfalls exist in blocking I/O as well, but require the buffer to be accessible from multiple threads at once, a situation which is usually avoided.

The main difficulty with asynchronous I/O is that the programmer has to structure the code around the asynchronicity.

There must be a better way.

Futures and async/await

Our goal

We want to transform this code, which builds a `Future` using combinators:

```rust
let fut = open("/file.md")
    .or_else(|e| panic!("open error: {e}"))
    .and_then(|fd|
        fd.read_file()
            .or_else(|e| panic!("read_file error: {e}"))
            .and_then(|data|
                unlink("/file.md").or_else(|e| panic!("unlink error: {e}")).and_then(|_| data)
            )
    );
let data = executor.block_on(fut);
```

into something like

```rust
let fut = async {
    let data = open("/file.md").await?.read_file().await?;
    unlink("/file.md").await?;
    Ok(data)
};
let data = executor.block_on(fut)?;
```

Note on the syntax

```rust
let fut = async {
    let data = open("/file.md").await?.read_file().await?;
    unlink("/file.md").await?;
    Ok(data)
};
let data = executor.block_on(fut)?;
```

- `async { … }` creates an asynchronous block, *i.e.,* an anonymous `Future`.
- `async { … }` captures variables from its environment as needed, like a closure.
- `async move { … }` transfers all captured variables into the asynchronous block.
- `.await` in an asynchronous block waits for a `Future` to terminate.
- `.await?` is just `.await` followed by `?`, this is not a special syntax. It is useful if the `Future` being awaited returns a `Result` or an `Option`.

The program flow starts to look familiar and resembles a traditional blocking flow.

Asynchronous functions

get_and_delete() will return a Future containing a file content and delete it, or an error:

```rust
pub fn get_and_delete(name: String) -> impl Future<Output = Result<String, std::io::Error>> {
    async move {                    // move is needed here to capture `name` into the future
        let data = open(&name).await?.read_file().await?;
        unlink(&name).await?;
        Ok(data)
    }
}
```

This can be written even more concisely. This async fn desugars into the code above:

```rust
pub async fn get_and_delete(name: String) -> Result<String, std::io::Error> {
    let data = open(&name).await?.read_file().await?;
    unlink(&name).await?;
    Ok(data)
}
```

In both (equivalent) cases, calling get_and_delete(file_name) will return a Future. Future in Rust are lazy, and must be passed to an executor in order to progress.

The Future type in Rust

The `std::future::Future` trait represents a computation whose result will be available later:[3]

```rust
trait Future {
  type Output;
  fn poll(self: Pin<&mut Self>, context: &mut Context<'_>) -> Poll<Self::Output>;
}
```

The first call to `poll()` starts the computation. `poll()` returns:

- `Poll::Ready(result)`: the result is available without delay
- `Poll::Pending`: the result is not available yet, try again later

While it would be possible to call `poll()` repeatedly until the result is ready, it would be most inefficient. The `Context` contains a callback (a *waker*) that the operation started in the background must call to indicate that progress has been made (*e.g.*, when it is complete).

[3] Pretend you do not see the `Pin` type at the moment, and consider it as a smart pointer.

A `Future` unrolled

The code in `poll()` for a future `fut` returning the first 100 bytes of a file might:

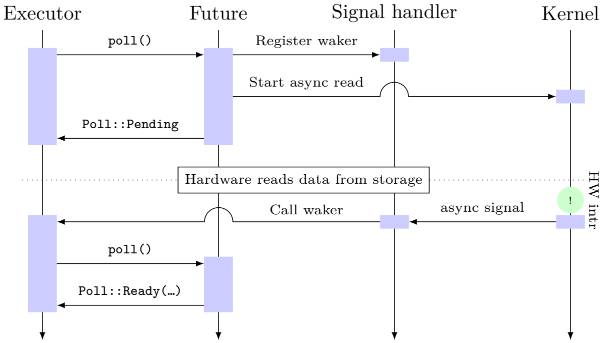
- prepare a buffer to store the content;
- register with the asynchronous I/O signal handler the association between asynchronous read operation and the callback passed in the context;
- start the asynchronous read of the file;
- return `Poll::Pending`.

When the asynchronous read terminates, the operating system sends the asynchronous I/O signal, which in turn checks the associations and find that the asynchronous read is complete. It calls the callback, which tells the executor that `fut.poll(context)` should be called again. This call will:

- see that the asynchronous operation has completed;
- return `Poll::Ready(buffer)` with the 100 first bytes of the file (at most).

The executor is in charge

The executor is in charge of polling the `Future` when it is useful to do so.



A minimal executor contains:

```rust
/// Execute `fut` to completion
fn block_on<T, F>(fut) -> T
  where F: Future<Output = T>
```

This future may itself launch sub-futures by calling their `poll()` method from its own `poll()` method, passing them the same waker. As soon as one sub-future progresses, the top-level future `poll()` method will be called by the executor.

`Future` and parallelism

The `futures` crate contains utilities to help working with futures, such as:

- `join(fut1, fut2)`: return a `Future` returning a couple `(r1, r2)` with the result of both futures
- `select(fut1, fut2)`: return a `Future` returning a `Either` value. `Either::Left(r1)` if `fut1` terminates first, `Either::Right(r2)` if `fut2` terminates first.

If `fut_iter` is an iterator producing `Future` objects with the same type:

- `join_all(fut_iter)`: return a `Future` returning a `Vec` with the result of all futures
- `select_all(fut_iter)`: return a `Future` returning the result of the first terminating future along with its index

Parallelism example

`get_and_delete_many()` works on several files in parallel:

```rust
pub async fn get_and_delete(name: String) -> Result<String, std::io::Error> {
    let data = open(&name).await?.read_file().await?;
    unlink(&name).await?;
    Ok(data)
}

pub async fn get_and_delete_many(names: Vec<String>) -> Result<Vec<String>, std::io::Error> {
    let results: Vec<Result<String, std::io::Error>> =
        join_all(names.into_iter().map(get_and_delete)).await;
    // An iterator of `Result<T, E>` can be collected into `Result<Vec<T>, E>`
    results.into_iter().collect()
}
```

It should be noted that:

- `names.into_iter().map(get_and_delete)`[4] is an iterator producing futures
- `join_all()` produces a future from the iterator, and we await it

[4] The last part is equivalent to `.map(|name| get_and_delete(name))` through η-conversion

What about lifetimes?

`name` was captured in `get_and_delete()` because `async fn` desugars into a function returning a `async move` block. But what if the name was given as a `&str`?

The naïve version without using `async fn` fails to compile:

```rust
pub fn get_and_delete(name: &str) -> impl Future<Output = Result<String, std::io::Error>> {
    async {                          // no `move` needed to capture `name` as references implement Copy
        let data = open(name).await?.read_file().await?;
        unlink(name).await?;
        Ok(data)
    }
}
```

```
error[E0700]: hidden type for `impl Future<Output = …>` captures lifetime that does
  not appear in bounds
```

As expected from Rust, `async` and `async move` build an anonymous `Future` whose lifetime cannot exceed the lifetime of captured parameters, just like a closure does.

Solving the lifetime issue

There are several ways this particular lifetime issue can be tackled. One obvious way would be to make a new copy of the name and capture it inside the `async` block:

```rust
pub fn get_and_delete(name: &str) -> impl Future<Output = Result<String, std::io::Error>> {
    let name: String = name.to_owned();
    async move {                             // `move` needed to capture `name` inside the block
        let data = open(&name).await?.read_file().await?;
        unlink(&name).await?;
        Ok(data)
    }
}
```

In this case, the returned future does not reference anything from the environment, and its lifetime is `'static`: it will leave until the end of the program (or until it gets consumed), even if the original `name` is deallocated.

This solution wouldn't work with `async fn` though: the `String` was copied with `.to_owned()` before creating the `async move` block. In an `async fn`, the whole function body is put in the `async move` block while desugaring, the early copy cannot be done this way.

Solving the lifetime issue (con't)

Another solution would be to accept that the returned `Future` is constrained by the input parameter lifetime, and represent it into the function output type [5]:

```rust
pub fn get_and_delete(name: &str) -> impl Future<Output = Result<String, std::io::Error>> + '_ {
    async {
        let data = open(name).await?.read_file().await?;
        unlink(name).await?;
        Ok(data)
    }
}
```

The future will have to be used or dropped before the end of `&str` lifetime. `async fn` will also automatically add this lifetime restriction to the anonymous output type it desugars into. This works:

```rust
pub async fn get_and_delete(name: &str) -> Result<String, std::io::Error> {
    let data = open(name).await?.read_file().await?;
    unlink(name).await?;
    Ok(data)
}
```

[5] According to lifetime elision rules, `'_` in an output type represents the lifetime of `&self`/`&mut self` when it is the first function parameter, or the unique input lifetime otherwise, in this case the lifetime of `&str`.

Solving the lifetime issue (the end)

Using `&[&str]` instead of `Vec<String>` in `get_and_delete_many()` may be interesting[6]:

```rust
pub async fn get_and_delete_many(names: &[&str]) -> Result<Vec<String>, std::io::Error> {
    let results: Vec<Result<String, std::io::Error>> =
        join_all(names.iter().map(|name| get_and_delete(name))).await;
    results.into_iter().collect()
}
```

Not using `async fn` requires naming the one input lifetime that will be copied into the output:

```rust
pub fn get_and_delete_many<'a>(names: &'a [&str]) ->
    impl Future<Output = Result<Vec<String>, std::io::Error>> + 'a
{
    async {
        let results: Vec<Result<String, std::io::Error>> =
            join_all(names.iter().map(|name| get_and_delete(name))).await;
        // An iterator of `Result<T, E>` can be collected into `Result<Vec<T>, E>`
        results.into_iter().collect()
    }
}
```

[6] Note how `.map(get_and_delete)` cannot be used here as the type of the items (`&&str`) does not match the requires type (`&str`). The compiler inserts an auto `*` before `name` in the function call: `get_and_delete(*name)`.

How does the compiler build a `Future` from an `async` block?

The use case will be the following simple `async move` block (`filename` is a `&str`):

```rust
async move {
    let file = open(filename).await?;
    let data = file.read_file().await?;
    file.close().await?;
    Ok(data)
}
```

The idea is to transform it into an object implementing `Future`. `poll()` will be called: each time there is a `.await`, `poll()` will return `Poll::Pending` if the awaited future is not immediately ready. The next time `poll()` will be called, the work must resume at the right point.

In the theory of computation, this is called a *finite-state machine* (or *FSM*).

`async` block hand-waving: the FSM

⚠ *The following three slides are a schematic view (or some hand-waving) of what happens when an `async` block is transformed. The real process is different, but works roughly the same way.*

The compiler builds a state machine to preserve the state and all variables alive accross each wait point and will implement `Future` on it:

```
enum $AsyncBlock42<'filename> {
    Start { filename: &'filename str },
    AwaitLine2 { $future: OpenFut },    // OpenFut is the type returned by `open()`
    AwaitLine3 { file: File, $future: ReadFileFut },
    AwaitLine4 { data: String, $future: CloseFut },
}
```

`filename` is a captured variable; since it is a reference type, it requires an explicit lifetime. It is only stored in `Start` because it is no longer used after the `await` point at line 2. `file` however is alive accross the `await` points at line 3.

If a variable is alive accross several `await` point, it must always end up at the same offset from the beginning of the structure for reasons that will be explained later.

async block hand-waving: poll()

```
impl Future for $AsyncBlock42<'_> {   // `'filename` will never be referenced, use `'_`
    type Output = Result<String, std::io::Error>;
    fn poll(self: Pin<&mut Self>, context: &mut Context<'_>) -> Poll<Self::Output> {
        match self.as_mut() {
            &mut $AsyncBlock42::AwaitLine3 { ref mut file, ref mut $future } => {
                if let Poll::Ready(x) = $future.poll(context) { // `file.read_file()` is completed
                    let x = match x {                             // Expansion of `?` from source
                        Ok(x) => x,
                        Err(x) => return Poll::Ready(Err(x.into())),
                    };
                    let data = x;                                 // From source
                    let $future = file.close();                   // From source
                    *self.as_mut() = $AsyncBlock42::AwaitLine4 { data, $future }; // Progress
                    self.poll(context)                            // Poll the next future
                } else {
                    Poll::Pending                                 // Wait again
                }
            }
            … // Other variants not shown here
        }
    }
}
```

`async` block hand-waving: final expansion

So this code

```rust
pub async fn get_file_md(filename: &str) -> Result<String, std::io::Error> {
    let file = open(filename).await?;
    let data = file.read_file().await?;
    file.close().await?;
    Ok(data)
}
```

will produce

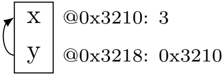
```rust
pub fn get_file_md(filename: &str) -> $AsyncBlock42<'_> {  // Copy the input lifetime as 'filename
    $AsyncBlock42::Start { filename }                      // Capture `filename`
}

enum $AsyncBlock42<'filename> { … }

impl Future for $AsyncBlock42<'_> {
    type Output = Result<String, std::io::Error>;
    fn poll(self: Pin<&mut Self>, context: &mut Context) -> Poll<Self::Output> { … }
}
```

Self-referential structures

Rust is a language which likes to move objects around. By default, when ownership is transferred, the underlying object can move around (for example when storing a field in a `struct`).

It is always safe to use `std::mem::replace()`: any mutable reference to a fixed-size object can be used to replace the object with another one of the same type.

But what if an object could store a reference to itself? Would such an object be safe to move? For example, if an object stored at address `0x3210` in memory contained a `x` field initialized to `3u64` and a `y` field of type `&u64` pointing to this `x` field?



@0x3210: 3

@0x3218: 0x3210

Fortunately, even though it is possible to build such an object, Rust lifetimes and borrowing rules will prevent it from moving at all[7].

[7] An example follows in a few slides.

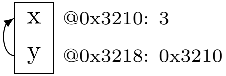
async block and self-referential structures

Now what about this async block?

```
1  async {
2      let x = 3;
3      do_something().await;
4      let y = &x;
5      do_something_else().await;
6      f(y)
7  }
```

x and &y are both alive accross the await point at line 5: they will be stored in the FSM, and y (in reality the y field) references the x field. This self-referential structure is built by the compiler, which can do much more than the user, and is not burdened by lifetime considerations.

However, once stored at an address (*e.g.,* 0x3210) where it will be used by the FSM, it must never move again, or y will not point to the right place. How can this be achieved?



Preventing an object from moving

What can make an object move in safe Rust? As discussed already:

1. The owner of an object can transfer it. The object may move in the process.
2. The owner of a mutable reference to an object can `std::mem::replace()` this object with another one. Not only will the referenced object move, but the owner can now transfer it and move it again (item 1).
3. Any smart-pointer implementing `DerefMut` can be used to obtain a mutable reference to the object. See item 2.

To prevent an object from moving, one must ensure that its owner cannot move it, and that no mutable reference or smart pointer implementing `DerefMut` exist or can be obtained in safe Rust on this object. However, to be useful, the object must still be accessible through non-mutable references, and maybe even mutably through `unsafe` functions.

`Pin` is the solution to this problem.

The `Pin` wrapper over pointers

If `P<T>` is a smart-pointer or a reference over `T` (for example `Box<T>`, `&mut T`), `Pin<P<T>>` is a smart-pointer which wraps an existing pointer or smart-pointer:

- If `P<T>` implements `Deref<Target = T>`, `Pin<P<T>>` also does.
- If `T` implements `Unpin` only: if `P<T>` implements `DerefMut`, `Pin<P<T>>` also does.
- If `T` implements `Unpin` only: `.into_inner()` can be used to retrieve the inner `P<T>`.

As a consequence:

- If `T` implements `Unpin`, `Pin` is mostly transparent: `Pin<Box<T>>` will act as a `Box<T>`, `Pin<&mut T>` will act as a `&mut T`.
- If `T` does not implement `Unpin`, `Pin` acts more like a non-mutable reference: `Pin<Box<T>>` and `Pin<&mut T>` will implement `Deref` but not `DerefMut`. Only operations requiring a non-mutable access (`&self` methods, but not `&mut self` methods) can be used.
- If `T` does not implement `Unpin`, once a mutable reference (the only one) or a pointer implementing `DerefMut` (the only one) has been wrapped into a `Pin`, it can never be unwrapped.

The Unpin marker trait

The Unpin marker trait is automatically implemented by almost all Rust types, as it is not possible to build a self-referential structure other than locally in safe Rust, and this structure cannot ever be moved:

```rust
1  struct SelfRef<'a> {
2      x: u64,
3      y: Option<&'a u64>,
4  }
5
6  fn main() {
7      let mut s = SelfRef { x: 3, y: None };
8      s.y = Some(&s.x);   // `s` borrows itself
9      let t = s;          // Error: cannot move out of `s` because it is borrowed (line 8)
10 }
```

However, as soon as the compiler builds a self-referential data structure when transforming an async block into a FSM, it marks this structure as !Unpin. If a reference or smart-pointer to an object of this type is placed into a Pin, the object will never move again until it gets destroyed.

Explaining `poll()` signature

The signature of the `poll()` function for the `Future` trait contains `Pin<&mut Self>`:

```
trait Future<T> {
    type Output;
    fn poll(self: Pin<&mut Self>, context: &mut Context<'_>) -> Poll<Self::Output>;
}
```

When writing the `poll()` method for a type implementing `Future`:

- either the type is provably `Unpin`, as most types are: `self` may be freely used as a mutable reference to the object;
- or the type is not provably `Unpin`: the accessible operations on the objects are similar as those through a non-mutable reference, or `unsafe` must be used.

But in what case could a user type implementing `Future` not be provably `Unpin`? Through generics! Storing an object of an arbitrary type (this possibly `!Unpin`) into a structure will make it `!Unpin`. Indeed, if a structure moves, its fields will also move; having one (possibly) unmovable field makes the structure unmovable.

Dealing with a !Unpin in poll()

Using unsafe allows to still get a mutable reference to self in poll() even though the type is potentially !Unpin: it becomes the programmer's responsibility to uphold Rust invariants, such as not moving the object or its possibly Unpin fields.

For example, implementing join(fut1, fut2) as a future returning the results of both futures fut1 and fut2 might imply designing a type like:

```rust
struct Join<T1, T2, F1: Future<Output = T1>, F2: Future<Output = T2>> {
  fut1: F1,
  fut2: F2,
  …
}
```

Since F1 and F2 might be !Unpin, Join cannot be proven to be Unpin in all cases. In poll(), it will require using unsafe to get access to &mut self.fut1 and &mut self.fut2, and put them in Pin wrappers themselves in order to call their respective poll() method. They will not have moved, the invariant is upheld.

Now that futures are thoroughly explained, it is time to use them in practice.

Executors and libraries

Executors

The standard library does not contain any executor. The most used executors are:

- `Tokio`: a modular full-fledged executor coming with bells and whistles
- `async-std`: an asynchronous version of the Rust standard library
- `smol`: a small and fast asynchronous runtime

In addition to the basic `block_on()` service, those executors offer many additional services:

- I/O (file and network) versions of the common operations — using a blocking operation in a `Future` (or an `async` block) would block the entire thread and defeat the purpose
- timer centered functions, to implement delays, and timeouts

Tokio will be briefly presented through the next slides. Its `full` feature flag can be used during development before trimming down the features.

Tokio

Tokio comes with a handy attribute allowing to define an asynchronous `main()` function. Similarly, a `#[tokio::test]` attribute decorates asynchronous test functions in lieu of `#[test]`.

```rust
#[tokio::main]
async fn main() {
    do_something().await;
}
```

Tokio comes with drop-in replacements of the standard library I/O types and modules:

```rust
use tokio::net::{TcpListener, TcpStream};

#[tokio::main]
async fn main() {
    let listener = TcpListener::bind("127.0.0.1:6379").await.unwrap();
    loop {      // Accept then process one incoming connection at a time
        let (socket, _) = listener.accept().await.unwrap();
        process(socket).await;
    }
}

async fn process(socket: TcpStream) { … }
```

Tokio: spawning futures

Tokio also provides `tokio::spawn()`, which runs a future independently of the current one, thus resembling the thread model:

```rust
use tokio::net::{TcpListener, TcpStream};

#[tokio::main]
async fn main() {
    let listener = TcpListener::bind("127.0.0.1:6379").await.unwrap();
    loop {      // Accept an incoming connection and spawn a new future to handle it
        let (socket, _) = listener.accept().await.unwrap();
        tokio::spawn(process(socket));    // Run independently
    }
}

async fn process(socket: TcpStream) { … }
```

Spawning a future will call its `poll()` automatically as long as it does not terminate. The result will be lost, as spawning a future makes it live its own life. Channels can be used to communicate in this scenario.

Tokio: synchronization tools

The Tokio crate contains several asynchronous channel models:

- `mpsc`: multi-producer single-consumer, similar to the one in the standard library
- `oneshot`: single value channel, both the sender and the receiver are consumed after use
- `broadcast`: multiple-producer multiple-consumer, every reader sees every message
- `watch`: single-producer multiple-consumer, every reader sees only the last value produced

Also, Tokio offers alternatives to the common synchronization tools. The Tokio variants are asynchronous and will not block the whole thread: `Mutex`, `RwLock`, etc.

futures: the extension crate

The futures crate augments the standard library functionalities:

- The FutureExt trait adds combinators (map(), filter(), etc.) to the Future trait.
- The TryFutureExt trait adds combinators (and_then(), or_else(), etc.) to the Future trait when its output type is a Result.

Importing those traits into direct visibility using a use clause is enough to get those new methods available on any Future object.

The crate also contains standalone functions to join() and select() futures (simultaneous execution), or to build an anonymous future type from a poll()-like function:

```rust
use futures::future::poll_fn;
use futures::task::{Context, Poll};

fn hello(_cx: &mut Context<'_>) -> Poll<&'static str> {
    Poll::Ready("Hello, World!")
}

let hello_future = poll_fn(hello);
```

Conclusion

In this class, the following points were made:

- Blocking requests paralyze an entire thread even during waiting times.
- Non-blocking I/O can free up threads but require more involvement from the programmer.
- Asynchronous I/O are of a higher level than non-blocking I/O but can lead to a callback mess.
- Futures combined with `async`/`await` lead to a more sequential program flow.
- The Rust compiler transforms `async`/`await` blocks into a finite state machine (FSM).
- The Rust compiler tries very hard to ensure that the programmer will not attempt to use unmovable data, thanks to the `Pin` wrapper type and the `Unpin` marker trait.
- Executors are not bundled with the Rust standard library. They come in various shapes and forms. External crates add functionalities to the existing futures.