Distributed Systems Lab 04

Asynchronous programming

During the previous lab, we presented basic ideas behind a so-called event-driven shared-nothing architecture and discussed how to implement them as an executor system on top of operating system threads. This lab presents asynchronous programming in Rust, which should facilitate more efficient implementations of the executor system. We will use files contained in this package, which you should download and extract locally.

Moreover, this lab features no Small Assignment. Instead, you will be required to solve the first Large Assignment.

Learning Section

Asynchronous programming

Traditionally, computations within a single thread of execution are synchronous: consecutive instructions are executed by the CPU one by one (some CPUs do implement out-of-order execution, but it is largely hidden from the programmers), and a call to a function requires completing the execution of the function before the subsequent instruction is executed.

However, in some applications, this approach is cumbersome. For instance, in the event-driven shared-nothing architecture, it requires splitting the functionality of a module into multiple handlers and storing and restoring some state when subsequent messages are sent and received. Moreover, in the executor system, a careless assignment of modules to executors may result in degraded performance, for instance, when the workload is not balanced between the threads optimally.

In such an application, one may use the **asynchronous programming** paradigm. It introduces a concept of computations whose result is initially unknown and provides means of executing them asynchronously and continuing the main computations when the result is ready. Asynchronous programming is supported by multiple modern programming languages under various names, including a *future*, a *promise*, a *delay*, a *deferred*, and the like.

Example asynchronous programming task

Asynchronous programming, in practice, is often seen as a complementary approach to operating-system-thread-based concurrency. Imagine you would be writing a function to download a couple of websites, using a library `fetch` function. In the simplest approach, you could start with a sequential implementation like:

```
fn download_two() -> (Data, Data) {
    let site_1 = fetch("http://a.com");
    let site_2 = fetch("http://b.com");
    (site_1, site_2)
}
```

Because synchronous system calls are blocking while waiting for I/O operations, the simplest approach to introduce concurrency is to spawn multiple operating system threads, each waiting for its I/O operation to finish:

```rust
fn download_two() -> (Data, Data) {
    let site_1 = thread::spawn(|| fetch("http://a.com"));
    let site_2 = thread::spawn(|| fetch("http://b.com"));
    (site_1.join().unwrap(), site_2.join().unwrap())
}
```

However, this approach is quickly limited by the overhead of memory for each thread state and *context switching* between them. You may be aware, for instance, from Network Programming classes, that an efficient approach is to use non-blocking *asynchronous* I/O operations. This approach, however, is complex to modularize. A naive implementation could look like:

```rust
fn download_two() -> (Data, Data) {
    let site_1: AsyncDownloader<Data> = fetch_async("http://a.com");
    let site_2: AsyncDownloader<Data> = fetch_async("http://b.com");
    while let Ready(fd) = libc::poll([site_1.get_fd(), site_2.get_fd()]) {
        // Dispatch the event to downloader structs. ...
        if fd == site_1.get_fd() {site_1.progress()} else {site_2.progress()};
    }
    (site_1.join(), site_2.join())
}
```

Apart from other issues in the code above, one stands out: introduction of the event loop to the function both obfuscate its flow a prevents further reuse as asynchronous, e.g., to implement `download_four`. Therefore, it is clear proper modularization of the code above requires an asynchronous framework or even a paradigm. Ideally, we would like to simply write code like:

```rust
async fn download_two() -> (Data, Data) {
    let site_1 = fetch_async("http://a.com");
    let site_2 = fetch_async("http://b.com");
    join!(site_1, site_2)
}
async fn download_four() -> ((Data, Data), (Data, Data)) {
    join!(download_two(), download_two())
}
```

In the code above, `download_four` preserves the ability to download four sites concurrently. Where did the event loop go? It is extracted to an *executor* for driving the computation, as discussed later. The `fetch_async` needs to express (partial) computation, as could be done by the above `AsyncDownloader` struct. This function could look in pseudocode like this:

```rust
async fn fetch_async(url: &str) -> Data {
    let mut response = sys::url_get_async(url).await;
    if let Redirect(new_url) = response {
        response = fetch_async(new_url).await;
    };
    response.parse() // synchronous
}
```

Fully-fledged examples of downloading multiple websites using asynchronous programming in Rust are provided in `examples/asychronous.rs`. Unless you are familiar with this programming paradigm, we suggest only browsing through these examples initially. Finally, remember that the *paradigm* is more general than merely supporting I/O-bound tasks — in fact, the System in the Assignment doesn't do any I/O.

Below, we discuss the details of asynchronous programming in Rust step-by-step.

Asynchronous programming in Rust

In Rust, asynchronous code is based on the **`Future`** trait:

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

We may look at this trait twofold: 1. The trait represents a `value: Output` that will be available in the future. 2. The trait represents an asynchronous *computation* that can ultimately produce a `value: Output`.

Every call to the `poll()` method tries to make progress on the computation and returns its status: - `Poll::Pending`, when the computation is not yet finished and `poll()` should be called again, - `Poll::Ready(Output)` otherwise.

You typically do not call this method directly, but it is called by an *executor driving the Future*. Of course, implementing an executor of asynchronous code as a simple loop, which repeatedly polls futures would be extremely inefficient. For this reason, `Context` is passed to a `poll()` call. It contains a special `Waker` type, which is used to notify the executor when some event happens so that the future can make progress: for instance, data has been read from disk.

The `poll()` method requires also a reference to `Self`, but wrapped in a `Pin` struct. `Pin` stands for **pinned pointer** and prevents the pointed value from being moved in memory. (Unless it implements the `Unpin` trait, which marks types that can be moved safely.) Pinning futures makes it safe to create references to values inside asynchronous code. Internally in Rust, asynchronous code is backed by an associated state machine holding intermittent state. Whenever the future makes progress, a transition to its next state occurs. You may also think about it in terms of coroutines, which may interleave execution with another code:

```rust
let mut something = 42; // A value on the stack frame
let x = &mut something; // And a reference to it
yield // the control flow returns up the stack and will return later
*x = 7; // The pointer inside `x` would be invalid if the stack relocated.
```

Futures can be easily created and manipulated using the futures crate. For example:

```rust
fn simple_future() {
    // Crate a future that is immediately ready with a value,
    // which is then transformed:
    let future = futures::future::ready(7).map(|x| x * 2);

    // Run the future to completion in the current thread:
    let result = futures::executor::block_on(future);

    println!("Result of the simple future: {}", result);
}
```

Futures can be combined and joined into new futures to implement more complex asynchronous computations (see `examples/futures.rs` for examples). However, creating complex computations this way is cumbersome and has several limitations.

Instead of combining futures functionally, we can ask the compiler to transform imperative-style code into a future. We do this with an `async` block:

```rust
// The return value is the same as in a typical block expression.
// It has an anonymous type as below:
let future/*: impl std::future::Future<Output=i32> + Sized */ = async {
    println!("I'm an async block!");
    42
};
```

The same transformation may be applied to a whole function. To create an asynchronous function, the **`async`** keyword is required. The function's return type becomes a future. For example:

```rust
async fn async_function() {
    println!("I'm an async function.");
}
// This is essentially another way to write:
fn async_function() -> impl std::future::Future<Output = ()> {
    async {
        println!("I'm an async function.");
    }
}

fn run_async_function() {
    // Create a future by calling an asynchronous function:
    let future = async_function();

    // Run the future to completion in the current thread:
    futures::executor::block_on(future);
}
```

Futures are the most useful, of course, when there are multiple asynchronous computations that can be executed concurrently, and we can express complex control flow. Within asynchronous code we may use the `await` expression to "unwrap" a future into its output. It suspends the current function until the execution of the `awaited` future is completed and its result is ready. In other words, our auto-generated future delegates `Future::pool` to the `awaited` one until it completes. When the current function is blocked, the asynchronous executor tries to run another one. For instance:

```rust
async fn action1_step1() -> String {
    String::from("Step 1 of Action 1")
}

async fn action1_step2(str: String) {
    println!("Step 2 of Action 1 follows {}", str);
}

// By using `.await` we do not block the thread until each step
// is completed. Contrarily, we make it possible to advance
// the other action when the futures are not completed:
async fn action1() {
    // Run the first step and wait until it completes:
    let partial_result = action1_step1().await;

    // Run the second step (using the result of the first step)
    // and wait until it completes:
    action1_step2(partial_result).await;
}
```

```rust

async fn action2() { /* ... */ }

fn run_asynchronous_actions() {
    // Join two futures into a new future:
    let future = futures::future::join(action1(), action2());

    // Run the future to completion on the current thread:
    futures::executor::block_on(future);
}
```

Full versions of the above examples are provided in `examples/futures.rs`

Tokio

Although the `Future` trait and the `async`/`await` syntax are a part of the Rust standard library, as of now there is no standard asynchronous executor. This is visible in the previous examples, where an executor from the `futures` crate is used. There are also multiple other crates that implement asynchronous executors. During this course, we will use the tokio crate, which provides a full-featured asynchronous runtime, implementing:

- Tools for working with asynchronous tasks: synchronization primitives, channels, sleeps, timeouts, …
- APIs for asynchronous IO: TCP/UDP sockets, filesystem operations, …
- Runtimes for executing asynchronous code (a runtime is, simplifying, an advanced asynchronous executor).

When using Tokio, one usually does not create and manage futures oneself, as in the previous examples. Instead, **tasks** are used. They are non-blocking units of execution to be executed asynchronously. Tasks are created by **spawning** futures, which makes them being executed asynchronously by a runtime. Tokio provides two types of runtime. A **single-threaded runtime** executes all tasks within the current thread. A **multi-threaded runtime** executes tasks on an automatically created thread pool. In other words, Tokio tasks implement a concept called *asynchronous green-threads*. A correctly coded program can be run on both runtimes without requiring any changes other than selecting the other type of the runtime. An example follows:

```rust
async fn async_function() {
    println!("I am a task created from the async function.")
}

// The `#[tokio::main]` macro creates a runtime, and
// calls `block_on()` passing to it the function as a future:
#[tokio::main]
async fn main_task() {
    println!("I am the main task.");

    let task1 = tokio::spawn(async {
        println!("I am a new task created from the async block.");
    });

    let task2 = tokio::spawn(async_function());

    let task3 = tokio::spawn(async {
        println!("I am a new task.");
        tokio::spawn(async {
            println!("I am a new task spawned by the new task.");
```

```
        });
    });

    // All above tasks are being now executed asynchronously.

    // Spawning a task returns a handle, which allows of waiting
    // for its completion:
    task1.await.unwrap();
    task2.await.unwrap();
    task3.await.unwrap();
}

fn tokio_example() {
    // Although `main_task()` is prefixed with `async`, the below
    // call is synchronous. The `#[tokio::main]` macro wraps the
    // asynchronous function with a synchronous initialization
    // of the Tokio runtime, and calls `block_on()` passing
    // the function as a future:
    main_task();
}
```

More examples, with a detailed commentary, are provided in `examples/tokio.rs`.

Execution of tasks is scheduled (within a single thread or within a thread pool) by the runtime in a manner called **cooperative multitasking**: execution is switched to another task when the current task explicitly yields the execution. It means that one should avoid using blocking calls or writing long synchronous blocks of code (i.e., long parts without `.await`) and one should spawn new tasks for potentially long-running actions. (In particular `println!` is a blocking operation.) Otherwise, the gain from using asynchronous programming is greatly diminished. In practice, Tokio should be able to handle any number of tasks.

Cancellation safety

The execution of a future can be cancelled rather trivially: by simply dropping the future when there is no `poll` on it in progress. While this is always safe in the sense of Rust's safety guarantees, not all futures guarantee a correct behavior if they are cancelled. For example, cancelling a future that acquired a semaphore could cause a deadlock by never releasing the semaphore.

Asynchronous channels

Tokio provides asynchronous channels in `tokio::sync::mpsc`. The usage of these channels is similar to the usage of `crossbeam`. An example is provided in `examples/tokio_channels.rs`.

Traits with asynchronous methods

Rust supports asynchronous functions in traits, however, with some limitations. One of the remaining ones is that these traits are not object safe (i.e., you cannot use `dyn Trait`). This issue is shared with traits using `impl Trait` in the return position (RPITIT):

```
trait Rpitit {
    fn debugable(&self) -> impl Future<Output=??>;
}
```

You can read more about both historical and remaining issues in this blog post (complications #1 & #2 are resolved).

Fortunately, there is the async-trait crate, which provides the `#[async_trait::async_trait]` macro that makes asynchronous functions always return a concrete type: a heap-allocated trait objects. The downside is that this solution requires a heap allocation and dynamic dispatch. An example is provided in `examples/async_trait.rs`.

Asynchronous closures

Using asynchronous callbacks, especially closures, can be a little tricky (in Rust 1.82).

In the case of storing asynchronous closures as callbacks that should be run when some action is completed, a concrete type is required. An asynchronous closure is a closure that returns a future. This future implements the trait `Future<Output = Ret>`, where `Ret` is the type obtained by `await`ing the future. Its exact type is not known, so a trait object of type `Box<dyn Future<Output = Ret>>` is necessary. Moreover, a future can be self-referential but `Box` does not guarantee that it will not be moved. Therefore, `Pin` is also needed, and the return type of the closure becomes `Pin<Box<dyn Future<Output = Ret>>>`. The closure itself also has to be a trait object, so the complete type of an asynchronous *call-once* closure, which accepts a parameter of type `Arg` is:

```
Box<dyn FnOnce(Arg) -> Pin<Box<dyn Future<Output = Ret>>>>
```

Such a closure can be created as follows:

```
let closure = Box::new(move |arg| {
    Box::pin(async move {
        /* ... */
    })
});
```

The first `move` is needed to transfer the ownership of any captured value to the closure, and the second `move` is needed to transfer the ownership to the returned future. A full example is provided in `examples/async_closure.rs`.

Asynchronous closures that may be called multiple times (`Fn` and `FnMut`) are currently very limited in stable Rust. This is because `Fn` and `FnMut` cannot express returning a future that contains a reference to the closure. This can happen if we remove the second `move` in the snippet above. You can read more about these issues in the Async Closures RFC and in this blog post.

Additional Homework

If you wish to read in depth about asynchronous code in Rust, we recommend the Asynchronous Programming in Rust book. Some students also recommended the video tutorial by Jon Gjengset, and his Crust of Rust series in general if you want to learn even more about Rust.

If you want to learn how the asynchronous computing works in Rust, we recommend reading the Async/Await blog entry by Philipp Oppermann. The entry demonstrates also how asynchronous programming can be used to handle keyboard interrupts, and how to implement a basic asynchronous executor.

Authors: F. Plata, K. Iwanicki, M. Banaszek, W. Ciszewski, M. Matraszek.