Coroutines and Asynchronous Code

Advanced Systems Programming (H) 2021-2022 – Laboratory Exercise 5
Dr Colin Perkins, School of Computing Science, University of Glasgow

1 Introduction

The Advanced Systems Programming (H) course uses the Rust programming language (`https://rust-lang.org/`) to illustrate several advanced topics in systems programming. Lecture 8 discusses the use of coroutines and asynchronous programming for lightweight concurrency, using examples in both Python and Rust. This laboratory exercise aims to reinforce that understanding, and begins to explore Futures and asynchronous I/O in the Rust programming language. **This is a formative exercise and is not assessed.**

2 Asynchronous Programming in Python

One of the programming languages that has well-developed support for asynchronous programming is Python. This support has been gradually developing over several releases, culminating in Python 3.7 which has a fully developed asynchronous I/O subsystem. This provides `async` functions as coroutines, with an `await` operation to yield control to another coroutine while blocked for I/O. The Python interpreter provides an event loop and runtime to support this.

Read the tutorial at `https://realpython.com/async-io-python/`, and in particular the section entitled "The 10,000-Foot View of Async IO". This reviews the concepts of asynchronous programming and coroutines, as an alternative concurrency mechanism. Make sure you understand the concept of coroutines, and how they differ from threads, as a means of supporting concurrent execution. Be aware of the difference between concurrent and parallel execution.

Then, read the section entitled "The `asyncio` Package and `async`/`await`" in the tutorial. If you have Python 3.7 installed on your system, and are familiar with Python programming, follow along with the examples and check that they work as described and that you understand the code presented. The goal is for you to understand how asynchronous code is structured, and how control flow passes between asynchronous functions when the `await` operations are performed.

Finally, read the section entitled "The Event Loop and `asyncio.run()`", paying special attention to point "#1: Coroutines don't do much on their own until they are tied to the event loop". Read the presentation at `http://www.dabeaz.com/coroutines/Coroutines.pdf`, slides 15–26, to reinforce this material, nothing that `async` functions are implemented as coroutines, with `await` calls mapping onto `yield` operations after scheduling non-blocking I/O requests. Review your understanding of the following:

- `async` functions compile to coroutines. These are represented by heap allocated objects in the runtime. A coroutine object performs no action unless explicitly called by the runtime, and does not have its own thread of control.
- The runtime provides an event loop and task executor, that can drive the execution of asynchronous functions. It does this by calling the `next()` or `send()` methods of the coroutine object, which in turn execute the next fragment of the `async` function.
- When executed by the runtime, an `async` function will only proceed until the next `await` statement. At that point, it schedules some I/O operation to complete in a non-blocking manner, then transfers control back to the runtime. The runtime then resumes executing the next available `async` function that it not blocked. Eventually the I/O completes, and the asynchronous function is marked as non-blocked, and will resume execution at some later time.
- An `async` function explicitly relinquishes control by calling `await`. It cannot otherwise be pre-empted, and will execute until it calls `await` irrespective of what other tasks exist in the system. The `async` functions are cooperatively scheduled.

3 Asynchronous Programming in Rust

The book Asynchronous Programming in Rust (`https://rust-lang.github.io/async-book/`) discusses futures, coroutines, and asynchronous programming in Rust. You should read it along with this handout, to help you understand the material.

3.1 Futures in Rust

Asynchronous programs in Rust are structured as a set of *futures*. Each future represents an asynchronous computation that will eventually return a value. In contrast to a *thread*, which starts executing independently once it is spawned, a future is more passive and operates within the context of a runtime. The runtime periodically polls the future, allowing it to make some progress, until it is complete. A future will only make progress when polled by the runtime; it does not have an active thread of control. Every future implements the `Future` trait from the Rust standard library. This is defined in `std::future::Future` as follows:

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

The `poll()` method advances the execution of the `Future`. The runtime calls this method when it has reason to believe the future is ready to make progress. The `poll()` method is passed a context, `cx`, that holds ongoing state, and returns an instance of the `Poll` enum. The `Poll` enum is defined in the Rust standard library as:

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

If the computation is not yet complete, the `poll()` methods returns `Poll::Pending` to indicate that the runtime should `poll()` again at some later time. Alternatively, if the computation is complete, `poll()` returns `Ready(Self::Output)` (where `Self::Output` is the return type from the future, defined as an associated type to be specified by implementations of the `Future` trait).

The `self` parameter of the `poll()` method has the unusual type of `Pin<&mut Self>`. That is, it takes a mutable reference to itself that is pinned in place – it can't be moved in memory. As with any other `&mut` reference, this allows the object to be mutated but requires that it be the only reference to the object. The additional restriction, that the object can't be moved, is needed to ensure references to the object stored within the context remain valid between calls to `poll()`.

It is possible to implement the `Future` trait directly, and pass the resulting object to an asynchronous runtime to be executed, but this is rarely done. Rather, futures in Rust are usually returned as the result of an **asynchronous function**.

You should never call `poll()` method of a `Future` directly. This method is called by the asynchronous runtime system, to manage the execution of the `Future`.

3.2 Asynchronous Functions

Asynchronous functions in Rust use similar syntax to Python. For example, an asynchronous function `get_response()` that takes a `Connection` as a parameter and returns a `Response` might be written:

```
async fn get_response(c : Connection) -> Response {
    ...
}
```

The Rust compiler automatically transforms such an asynchronous function into a state machine that implements the `Future` trait. For example, the above function is transformed internally by the compiler into:

```
fn get_response(c : Connection) -> impl Future<Output=Response> {
    ...
}
```

Every asynchronous function is therefore equivalent to a normal function that returns an implementation of the `Future` trait (the syntax `impl Future<...>` indicates that the function returns something that implements the trait `Future`, but does not specify the precise type). Note that the transformation is done as part of the program compilation, and is not visible to the programmer. An `async fn` is merely syntactic sugar to define a function that returns a `Future`. A runtime is still needed in order to execute the asynchronous function.

Asynchronous functions therefore obscure the actual code being generated, hiding the use of futures. The syntax is simpler, but there is a hidden code transformation that needs explanation. Do you think this is an appropriate trade-off? Discuss with the lecturer or lab demonstrator to ensure you understand the transformation is that being performed.

3.3 Awaiting Asynchronous I/O

The power of asynchronous functions is that rather than blocking while performing an I/O request, they can await a result. This suspends the asynchronous function, allowing some other code to execute while waiting for the I/O to complete. For example, the `get_response()` function defined about could be called as:

```
...
let resp = get_response(c).await;
...
```

The call to `.await` asks the runtime to asynchronously evaluate the future returned by the `get_response()` method and return a result when ready.

When the `.await` operation is performed, the asynchronous runtime calls the `poll()` method on the `Future` returned by the `get_response()` call. If the future returns `Ready(Response)` then the call completes and returns the result, setting the value of `resp` and allowing the code to continue executing.

However if the future returns `Pending`, then the runtime will put this future onto a queue of tasks to be woken up later, and then switch to executing another future that has now woken up. Then, at some later time, when the underlying asynchronous I/O operation is believed to be complete, the runtime will call the `poll()` method on the `Future` again, repeating until it returns `Ready(Response)`.

Calling `.await` potentially results in a context switch away from the current task, while asynchronous I/O is performed, allowing something else to run. The asynchronous runtime libraries support only *cooperative* multitasking: task switches only occur on calls to `.await`.

3.4 Asynchronous Runtime Libraries

Calling `.await` tells the runtime to temporarily suspend the current task and allow another task to execute. At the top level, though, it's still necessary to start the runtime and pass it an initial asynchronous function to execute. At the time of this writing, there are two well-known asynchronous runtime support libraries for Rust: **Tokio** and **async-std**. Several other asynchronous runtime support libraries also exist in less well-developed states. Of these, Tokio is the most popular and perhaps best developed.

When using Tokio (`https://crates.io/crates/tokio`), it is possible to declare `main()` to be an asynchronous function, provided it is labelled with a particular attribute:

```
#[tokio::main]
async fn main() {
    println!("Hello world");
}
```

The compiler rewrites such a function to return a `Future` as described earlier. The `#[tokio::main]` annotation causes the compiler to also generate a standard `main()` function that does nothing but initialise the Tokio runtime then call the asynchronous `main()` function repeatedly until it returns `Ready`.

The Tokio library also provides asynchronous versions of TCP and UDP sockets, file system operations, etc. It is important to use these instead of the standards, synchronous, blocking version of these calls. Calling a synchronous I/O operation will cause the entire runtime to block, stopping all asynchronous tasks, and so is to be avoided (unfortunately, the compiler cannot warn about this).

4 Formative Exercise

The second assessed exercise for this course required you to write a multi-threaded simple HTTP client. This performs a DNS lookup for a domain, then starts parallel connections to each address (IPv4 or IPv6) of the domain, continuing with the first connection to succeed and discarding the rest.

To practise use of asynchronous functions in Rust, write a similar program that uses asynchronous DNS lookups and TCP sockets to perform the same task. This program should perform a DNS lookup, asynchronously start connection attempts to each possible address, accept the first connection to succeed, send an HTTP request, and print the HTTP response. This can be implemented using Tokio, `async-standard`, or any other asynchronous runtime library you choose.

Once you implementation is complete, consider the following questions:

- Is the asynchronous version of this program is simpler and/or more efficient than the version you implemented using multiple threads with synchronous (blocking) I/O.
- To what extent the use of multiple asynchronous runtime libraries fragments the Rust ecosystem. Do you think it problematic that asynchronous code requires alternative runtime support libraries, and that there are competing libraries that provide their own I/O abstractions?
- How does asynchronous programming in Rust compare to asynchronous programming in Python? Does the asynchronous function abstraction make equal sense in the two languages?

This is a formative exercise and is not assessed. You do not need to submit your code or your answers to the questions, but may want to discuss them with the lecturer or lab demonstrator to confirm your understanding.

- + -