Coroutines and Asynchronous Code

Advanced Systems Programming (M) 2018-2019 – Laboratory Exercise 6
Dr Colin Perkins, School of Computing Science, University of Glasgow

Introduction

The Advanced Systems Programming (M) course considers several advanced topics in systems programming. In lecture 7 we reviewed the concepts of coroutines and asynchronous programming, using examples in both Python 3 and Rust. This laboratory exercise aims to reinforce that understanding, and to begin to explore the Tokio library that provides asynchronous I/O for the Rust programming language. This is the final laboratory exercise for the course.

Coroutines and Asynchronous Programming

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 implementation 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.

Having understood the concepts of asynchronous functions and `async/await` in Python, review the Tokio library for the Rust programming languages (`https://tokio.rs/`). Consider to what extent Tokio provides the same asynchronous programming abstractions as Python, and to what extent they differ. To what extent is a `Future` in Tokio equivalent to an `async` function in Python, and how do the `poll()` operations on `Future` instances relate to the `next()` and `send()` operations on Python `async` functions? How do the features provided by the Tokio runtime relate to those provided by the Python runtime?

Asynchronous programming is highly beneficial for single-threaded languages such as JavaScript, and has clear benefits in languages such as Python that are not aggressively multithreaded and that can efficiently manage, and garbage collect, coroutine objects in a syntactically lightweight manner. Its implementation in Rust–the Tokio library–is more complex, due to the need to be explicit about types and because it's difficult to reflect data ownership patterns in the types for mutually recursive coroutine execution. When do you think asynchronous I/O makes sense for Rust–do you believe the complexity is worthwhile for the benefits?