Concurrency

Advanced Systems Programming (M) 2018-2019 – Laboratory Exercise 5
Dr Colin Perkins, School of Computing Science, University of Glasgow

Introduction

The Advanced Systems Programming (M) course uses the Rust programming language (`https://rust-lang.org/`) to illustrate several advanced topics in systems programming. A key feature needed to make effective use of a modern multicore processor is safe concurrency. This laboratory session reviews concurrent programming in Rust, to reinforce the material covered in Lecture 6, and as preparation for the second assessed exercise.

Concurrency in Rust

To reinforce the material covered in Lecture 6, review the following sections of the online Rust book, try the code samples, and ensure you understand the material:

- **Closures** are anonymous functions that take parameters and can borrow, or take ownership of, local variables in their environment. Rust uses closures to represent functions that can be executed by a concurrent thread. The Rust book discusses closures in Chapter 13: `https://doc.rust-lang.org/book/ch13-01-closures.html`. Review that material. Make sure you understand the difference between closures that borrow their environment and those that take ownership of their environment.
- **Threads** are the mechanism used in Rust to run concurrent code. The Rust standard library provides a `spawn` function that runs a closure in a separate thread. Read `https://doc.rust-lang.org/book/ch16-01-threads.html` to understand how threads are created in Rust. In particular, pay special attention to how threads work with `move` closures to pass ownership of data to the newly created thread, in a way that prevents data races between the creating and created threads.
- **Message Passing** provides a safe mechanism for inter-thread communication in Rust. Read the section on message passing in the online Rust book, `https://doc.rust-lang.org/book/ch16-02-message-passing.html`, to understand how multiple producer single consumer channels can be used with Rust to allow message passing between threads. Again, pay attention to how ownership of data is transferred between threads.

The essential goal is to understand how Rust spawns threads to execute code in closures, and how it transfers ownership of data into those threads. Be sure to consider what guarantees are provided for concurrent code in Rust. In particular, while Rust provides freedom from data races consider whether this means that there are no race conditions in Rust code. Furthermore, consider whether Rust code is immune to deadlocks.