Understanding Over Guesswork

Andrew Hobden Yvonne Coady

Abstract

We discuss the merits of instructing operating systems courses using the modern systems programming language Rust. (TODO: Longer, better, stronger, faster)

Contents

The State of the OS Course

Concurrency, parallelism, memory management, process scheduling, mutexes, system calls, filesystems, and architectural considerations are all commonly taught concepts in Operating Systems courses. These topics can be a struggle to understand, even for determined students, due to their complex, low-level characteristics. Students are typically asked to use C or C++ to accomplish assignments on the above tasks.

Instructors may also find themselves struggling, as these assignments can be difficult to create, and sometimes nearly impossible to evaluate. Instructors and their markers desire assignments which are simple enough to fit into a few files, demonstrate understanding of failure modes, can be tested effectively in a preferrably automated fashion, and show students the caveats of their attempts to solve the problem. In many cases, a tradeoff is necessary. Building an interactive shell is a common, and much loved, assignment in which instructors must balance the number of features required with the time provided. Features such as pipes, background tasks, tab-completion, and environment variables are all desirable and interesting to implement, but contribute greatly to the complexity of the code, as well as the amount of time it takes to evaluate.

At the University of Victoria evaluation is typically done through an interactive demo involving the student and one of the teaching assistants. This method gives the student a chance to explain qualities and characteristics of their code, demonstrate it's features, and explain any possible bugs which may be discovered as the code is tested. Testing is often light and relatively incomplete due to time and technical constraints. Tools like `valgrind` and `clang` (with all warnings and lints flagged on) help in evaluation, but are not always used. We have found this to be a better method than the marker grading the student without any input from them, as it encourages a certain level of independence and creativity in the students, it also encourages students to improve their ability to explain their code. This method also gives the student immediate feedback that they may use on further assignments, rather than hearing back sometimes weeks later, often well into the next assignment.

Challenges

```
TODO: Survey some OS instructors to figure out what challenges
they face
```

Finding a Better Place

TODO: Introduce Rust and briefly mention other alternatives. Perhaps a comparison chart.

Introducting Rust

TODO: A high-level, plain english explaination of Rust and how it came to be.

Language History

Rust Basics

TODO: Introduce Rust basics and syntax.

Types

On the Lack of `null`

Borrow and Move Semantics

Traits

Safety as a First-Class Goal

TODO: Discuss "safety" and define it clearly, show how Rust accomplishes this.

Zero-cost Abstractions

TODO: Discuss why Rust's abstractions are different than, say, Java's, and why this is appropriate for systems.

Composition over Inheritance

TODO: Talk about how traits resonate with the UNIX philosophy and why this matters.

Static Analysis at the Core

TODO: Discuss the power of static analysis.

Tooling

TODO: Discuss tooling, testing, and automation.

A Comparison with C

TODO: A few examples where C falls apart and the equivilent Rust code works, or gives a compiler error.

Tackling Common Problems

TODO: Show a few solutions to problems like Dining Philosophers and do a qualitative evaluation of the differences in what they teach.

Future Work

References