Part 5: Everything in Rust

Implementation

I have two rust kernel modules – rust_mymem which contains the memory driver, and rust_test which contains the test implementation.

rust_mymem

- Initializes the memory region as a global static variable
- Creates RustMemoryDriver to interface with the region
- RustMemoryDriver implements pub read and pub write for the test module to read and write to the memory region

rust_test

- Initializes RustMemoryDriver
- Records the latency of reading and writing with different byte sizes
- Records count after multi thread operation

Difficulties

I was led on a wild goose chase by AI :(– everything online told me that in order to include external functions in rust kernel modules you have to use the syntax pub "C" extern. This led all of my makes to fail, which was unfortunate also because for most of the makes my entire kernel was recompiling as well. Thank you to In for helping me fix the error!

AI

AI was supremely unhelpful during this problem set – in fact, it was detrimental. I spent 80% of my time debugging bugs from AI generated Makefiles and extern code – it seems there isn't much information online about rust kernel modules interfacing with each other, which also did not help my debugging :(. However, the RustMemoryDriver was helped along by Claude, as were some parts of the test file.

Why does the inserting order matter here?

The inserting order matters because the test module uses functions from the memory module, so the memory module must be inserted before the test module.

Is there any performance difference compared to the program running in the user space (Part 4)? Why?

Rust Kernel Space		
Size	**Write Time (µs)**	**Read Time (µs)**
1	112.8	112.8
64	6.7	22.1
1024	5.1	13.1
65535	34.1	162.8

Rust User Space		
Size	**Write Time (µs)**	**Read Time (µs)**
1	6150	4488
64	339	314
1024	343	340
65535	513	532

Userspace seems to take much more time than kernel space did – which makes sense, considering now that the program is executing in userspace it no longer needs to interface with the kernel through time consuming system calls but can rather directly access memory.

What difference does your Rust-based, all-inside-kernel implementation make regarding the data race problem you had to deal with in Parts 3 and 4? How does the Rust compiler help address the problem? How does having the test module in the kernel make any difference?
All of the code protecting against data races is now inside the kernel and well established as the Rust compiler is very careful about preventing these kinds of harmful operations.

What tool(s) are used by Rust for Linux to generate "bindings" for a kernel module to access the function and data structures from a Rust kernel module?
Rust for Linux looks for the binding crate which contains information about the C API. It obtains access to this crate through bindgen.