Part Four: Everything in Rust

Implementation
My implementation uses Arc<Mutex<Vec>> as the memory region to write and read to, providing data race protection. The memory file struct containing this region implements the Operations trait that contains read, write, seek, open, and release methods for file management.

Difficulties
I had extreme difficulty compiling the kernel with Rust – despite following all of the steps, it took multiple tries to compile the kernel (each of which took 9+ hours). After going through two 20 hour compile runs and multiple 9+ hour runs on both my Mac and the Zoo, I was finally able to get the kernel to compile and load (each 9+ hour run compiled a kernel that would hang when booting up on qemu).

Is there any difference in performance compared to the program written in C? Why?
The Rust implementation seems to take a little time – perhaps due to the memory safety and data race protections that are enforced.

What difference does your Rust-based implementation make regarding the data race problem you had to deal with in Part 3? Does Rust compiler help? If so, how?

The kernel itself provides protection through Arc<Mutex<_>> in comparison to the C code that used mutexes in the test implementation.

1. Rust's ownership and borrowing rules prevent data races at compile-time by ensuring that mutable references are not aliased.
2. The usage of Arc<Mutex<_>> ensures that only one thread at a time can access the critical section, preventing data races and ensuring thread safety.
3. Rust compiler enforces these guarantees.

Now that both the test module and Memory driver are implemented in safe Rust, how does the Rust compiler help solve the data race issues from Part 3?
By wrapping the file handle in a Mutex protected by an Arc (atomic reference counting), Rust ensures:

- Only one thread can access the file at a time.
- Any access to the file is safe from data races.

The compile-time checking ensures that:

- The Mutex must be locked to access the data, preventing unsynchronized access.
- The lock is released properly, avoiding deadlocks.

AI
Chat GPT wrote the entire test file for this, and a combination of Perplexity Pro, Claude, and ChatGPT helped me write the kernel module.