The Pluggable Interrupt OS: Writing a Kernel in Rust

Gabriel J. Ferrer
Department of Mathematics and Computer Science
Hendrix College
<ferrer@hendrix.edu>

Rust-Edu Workshop 2022
https://rust-edu.org/workshop

Abstract

As Rust is a suitable language for writing bare-metal code, it is a viable competitor with C and C++ as a programming language of instruction for the Operating Systems course. The popular blog Writing an OS in Rust [5] provides stellar resources for instructors wishing to teach students how to create a kernel using Rust. In this paper, I describe a crate I developed called the Pluggable Interrupt OS [4]. Building on the resources from the aforementioned blog Writing an OS in Rust, this crate enables students to easily create a functioning kernel for the x86 architecture. A key aspect of its simplicity is presenting students with an API that renders the `unsafe` keyword unnecessary by presenting straightforward abstractions for setting up interrupts. Several assignments employed in the classroom built upon this crate are described.

1 Introduction

The Rust programming language is an appealing language of instruction for an Operating Systems course. Most Operating Systems courses use the C or C++ languages. Like these languages, a Rust executable can run on the bare metal in the absence of an operating system, thus enabling its use to create one. David Evans gives a persuasive argument for the superiority of Rust over other alternatives, particularly C and C++, for an Operating Systems course. [2].

The blog Writing an OS in Rust [5] gives lots of specific details for writing Rust code that interacts with x86 hardware, specifically the VGA buffer and the interrupt handlers. In terms of abstraction, the blog presents code to implement the `print!` and `println!` macros using the VGA buffer, but it presents no additional abstractions for setting up interrupt handlers.

In order to facilitate student projects by providing useful abstractions for bare-metal programming, I created the Pluggable Interrupt OS crate [4]. My goals for the crate are as follows:

- Enable students to create a 100% Rust bare-metal program.
- No `unsafe` blocks in student code.
- Provide straightforward abstractions for setting up timer and keyboard interrupts, as well as kernel startup code and background code.

The rest of this paper is organized as follows. We begin by describing the curricular context in which this crate was developed. Next, we examine the `HandlerTable struct`, the abstraction provided by the crate for setting up handlers for interrupts, startup, and background code. We next examine the setup of a cooperative multitasking kernel that shows how to integrate interrupt code with background code. We then discuss our classroom experiences and conclude with a discussion of future directions for the crate.

2 Curricular Context

Our curriculum requires all Computer Science majors and minors to take a two-course introductory sequence: Foundations of Computer Science and Data Structures. Once students complete Data Structures, they can

take any of our upper level courses, including Operating Systems. Students learn Python in Foundations of Computer Science and Java in Data Structures. Operating Systems is the only course in which we teach Rust.

Prior to attempting bare-metal programming, students gain experience with Rust over a series of assignments of gradually increasing difficulty and sophistication. In the first set of assignments, students implement each of the following Unix command line utilities as a Rust executable:

- `ls` - list a directory
- `rm` - remove a file
- `mv` - rename a file
- `cp` - copy a file
- `head` - see the first n lines of a file
- `wc` - count words, lines, and characters in a file
- `fgrep` - search for a text string in a file
- `sort` - sort the lines of a file
- `wget` - download a web page

Students then build two larger programs in Rust (inspired by assignments from David Evans's course [3]):

- Unix shell, including pipelines, I/O redirection, and background processes.
- Web server, spawning a separate thread for each HTTP request.

3 Setting Up Interrupt Handlers

As the primary goal of the crate is to enable students to easily set up interrupt handlers, the `HandlerTable` data type, shown in Figure 1, is provided to make it possible to do so clearly and concisely. Its four attributes control all programmer-specified aspects of the system's behavior:

- The `timer` function, if present, is invoked by the timer interrupt handler.
- The `keyboard` function, if present, is invoked by the keyboard interrupt handler.
- The `startup` function, if present, is invoked when the system begins.
- The `cpu_loop` function runs continuously whenever there is no interrupt to be handled. (If no function is specified, it executes the x86 `hlt` instruction.)

```
pub struct HandlerTable {
    timer: Option<fn()>,
    keyboard: Option<fn(DecodedKey)>,
    startup: Option<fn()>,
    cpu_loop: fn() -> !
}
```

Figure 1: The `HandlerTable` data type

None of the projects listed in Section 2 requires students to use the `unsafe` keyword. Teaching students to make disciplined use of `unsafe` in addition to the other complications involved with bare-metal programming

was more material than I felt possible to cover in the time available. This is why building the crate to abstract away from `unsafe` was an important goal.

As the crate's interrupt handling code is under 200 lines of code, it is feasible to explain it to students in detail. Here are three examples of using `unsafe` to interact with hardware that are straightforward to explain during class:

- Invoking the instruction to assign the interrupt handlers to the interrupt controller must take place within an `unsafe` block.
- Reading the key that triggered a keyboard interrupt requires accessing a specific memory location. That memory access can only occur within an `unsafe` block.
- Students can see where the `timer` and `keyboard` functions from `HandlerTable` are invoked, followed by the `unsafe` code to inform the interrupt controller that the handler function has completed its work.

Figure 2 shows a "Hello, World!" employment of the crate (included as `main.rs`). It prints a `.` character on each `timer` interrupt, and prints each character typed on a `keyboard` interrupt. Using the builder pattern, the programmer specifies the handler function for each attribute, starting execution with the `.start()` method.

```
fn startup() {println!("Hello, world!");}

fn tick() {print!(".");}

fn key(key: DecodedKey) {
    match key {
        DecodedKey::Unicode(character) => print!("{}", character),
        DecodedKey::RawKey(key) => print!("{:?}", key),
    }
}

#[no_mangle]
pub extern "C" fn _start() -> ! {
    HandlerTable::new()
        .keyboard(key)
        .timer(tick)
        .startup(startup)
        .start()
}
```

Figure 2: Hello, World!

The `.start()` method does the following:

- Runs the `startup` function.
- Stores the `HandlerTable` in a spin lock `Mutex`, accessible to the low-level interrupt handling code.
- Initializes and starts the timer and keyboard interrupts.
- Runs the `cpu_loop` function.

Once `.start()` has completed, the `cpu_loop` code runs indefinitely, unless an interrupt handler is triggered. Once a handler is triggered, control transfers to the handler function. Once the handler function exits, control resumes where it left off in the `cpu_loop` code. Note that in the Figure 2 example, since there is no `cpu_loop` code, all activity happens solely in the interrupt handlers.

4 Integrating Timer, Keyboard, and Ongoing Computation

Because interrupts are invoked asynchronously, it is necessary to have a strategy for concurrent access to data structures shared by the handlers and CPU code. Figure 3 shows one strategy for integrating timer and keyboard events with an ongoing computation. This is part of the skeletal code for one of the course assignments, in which students create a cooperative multitasking kernel with four processes in the VGA buffer, as shown in Figure 4. `AtomicCell` objects (from the `crossbeam` crate [1]) wrap both the most recent keystroke as well as the count of timer interrupts. The `cpu_loop()` function has a `Kernel` object as a local variable that can be given a keystroke, asked to draw itself, or asked to execute an instruction. It constantly checks the `AtomicCell` objects that store the ticks and recent keystrokes, and updates the `Kernel` accordingly.

```
static ref LAST_KEY: AtomicCell<Option<DecodedKey>> = AtomicCell::new(None);
static ref TICKS: AtomicCell<usize> = AtomicCell::new(0);

fn cpu_loop() -> ! {
    let mut kernel = Kernel::new();
    let mut last_tick = 0;
    kernel.draw();
    loop {
        if let Some(key) = LAST_KEY.load() {
            LAST_KEY.store(None);
            kernel.key(key);
        }
        let current_tick = TICKS.load();
        if current_tick > last_tick {
            last_tick = current_tick;
            kernel.draw();
        }
        kernel.run_one_instruction();
    }
}

fn tick() {TICKS.fetch_add(1);}

fn key(key: DecodedKey) {LAST_KEY.store(Some(key));}
```

Figure 3: Integrating Timer and Keyboard with Background Computation

5 Classroom Assignments

Students were given three assignments in which to use the Pluggable Interrupt OS. The first assignment was to create a simple bare-metal video game. The second assignment was to create a video game kernel in which the user picks a student-created game from the previous assignment to play. Students learn about process management as the user can pause games, start new instances of games, and resume games.

The third assignment was to create the four-window cooperative multitasking operating system depicted in Figure 4, building on the code template introduced in Section 4. Two applications are provided: a program to compute π via a power series, and an interactive program to compute an average. The first application is designed for running in the background, while the second emphasizes I/O. The student is to divide the VGA buffer into four windows, each of which can run one of the two applications.

About half of the students in the course successfully completed the video game and video game kernel assignments, and about a third of the students completed the cooperative multitasking operating system. There seem to have been two major obstacles to completing the assignments:

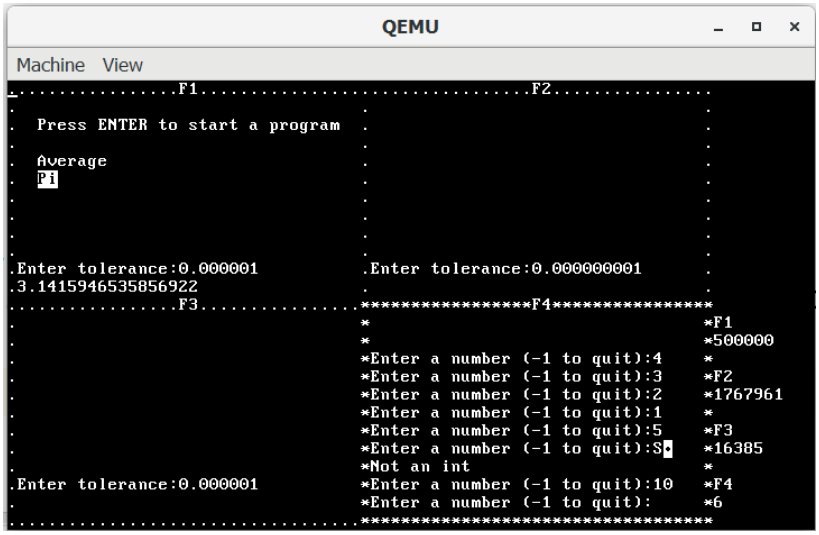

Figure 4: Screenshot of Cooperative Multitasking Kernel

- It was possible to earn a good grade in the course without successfully completing them.
- In spite of the simplifications provided by the Pluggable Interrupt OS, the code template for the cooperative multitasking assignment was still over 200 lines of code. The code template over-specified the design; next time, I plan to provide less code and create more conceptual space for students to work out their own designs.

6 Conclusion

The overall design goals for the Pluggable Interrupt OS crate were met. However, successful classroom implementation may require further effort to improve explanation of the assignments. Changing the course grading criteria to more strongly incentive the effort necessary to write the kernel projects may also be helpful.

The crate's capabilities do enable students to create a bare-metal program that can legitimately be considered a kernel. But two key additional capabilities would expand pedagogical options considerably:

- Mechanisms to enable pre-emeptive multitasking.
- Introduce a mechanism for persistent storage, thus enabling the creation of a file system.

References

[1] Alex Crichton, Jeehoon Kang, Aaron Turon, and Taiki Endo. `crossbeam`. `https://crates.io/crates/crossbeam`. Retrieved August 11, 2022.

[2] David Evans. Using Rust for an undergraduate OS course. `http://rust-class.org/0/pages/using-rust-for-an-undergraduate-os-course.html`, 2013. Retrieved August 20, 2022.

[3] David Evans. cs4414: Operating systems. `http://rust-class.org/`, 2014. Retrieved August 20, 2022.

[4] Gabriel Ferrer. Pluggable interrupt OS. `https://crates.io/crates/pluggable_interrupt_os`. Retrieved August 20, 2022.

[5] Philipp Oppermann. Writing an OS in Rust. `https://os.phil-opp.com`. Retrieved August 20, 2022.