I Project Description

I made a minimal x86-64 kernel in Rust, largely following the Writing an OS in Rust blog by Philipp Oppermann.

The project is hosted on GitHub at https://github.com/CordlessCoder/os. Information on building and running the kernel, a more detailed description of the implementation, and what I want to implement next is available in the repo.

II Implementation

Following the blog series, I created a bootable freestanding Rust binary and used `bootimage` to build the bootloader.

I started by designing what I think is a convenient API for writing output to the VGA Text (Mode 3) output.

```
fn test_println() {
    println!(fgcolor = Blue, "Hello from the VGA console!");
}
```

To implement it, I need to allow safe global access to the VGA output writer, so I needed a mutual exclusion primitive. This was a great opportunity to write my own spinlock-backed Mutex, taking inspiration from what I learned from the Rust Atomics and Locks book by Mara Bos.

I then created the Color API as a nice wrapper over the cryptic VGA color encoding scheme, as well as implemented the underlying byte-encoding.

I then moved on to implementing the fundamental operations on the VGA buffer using volatile accesses to ensure the changes are externally observable.

With all the components ready, I built the `println!` macro. I then did the same for QEMU serial output to be used in tests and for debugging.

To make the kernel operational I also added interrupt handlers following the blog, as otherwise it would immediately double and triple-fault, causing an immediate hardware reset.

I then implemented paging and a memory frame allocator, which allowed me to get to the fun part of implementing my own heap allocator.

I started with a simple bump allocator. While it does allocate memory extremely quickly, it cannot properly reuse deallocated memory until all allocations have been freed, which makes it fundamentally incompatible with long-lived allocations like ones you find in a kernel. I wrote a test with a long-lived allocation to showcase that, which it failed.

To remedy this, I wrote a more complex, freelist-backed allocator which is one of the rare cases outside lock-free data structures where a linked list truly is the most optimal solution 😆 . I then implemented merging of adjacent free nodes by keeping the free list sorted and added allocation statistics to track memory usage and allocation patterns in the kernel.

With memory allocation, interrupts and output handled, I had everything I needed to start working on an async runtime. I built an async executor similar to the one described in the blog, but using a `BtreeSet` to represent the tasks that need to be woken, instead of a queue to avoid an infinite loop if a task is woken multiple times for each time it is polled, which is common when a task registers itself for multiple Wakers.

Afterwards I implemented a global clock driven by the PIT and implemented efficient interrupt-driven sleep for tasks.

Now tasks can display output and wait, but that isn't very useful without an ability to handle input, so I wrote an interrupt-driven Keypress Stream as well. This allows tasks to efficiently and conveniently respond to keyboard input without busy spinning.

With input, output and the ability to perform actions based on the time, I had everything I need to build a simple game. So, I built a very simple shell to be able to select what to run and then made simple snake and flappy bird clones.

III Results

I now have a working kernel that can run simple games and most no_std Rust code!

A showcase of the kernel is available on youtube at https://youtu.be/GBPCbMECyOs.

IV What I learned

I've been interested in the lower-level intricacies of how a modern computer operates for quite some time, but I've generally not dared to go below the syscall interface provided by Linux. I also write a lot of code related to concurrency atop cooperatively scheduled coroutines (via Rust async) but have never actually written an async executor/runtime myself.

This project allowed me to learn how an async executor works, how memory allocators track memory and the different approaches to memory mapping using paging in kernels.

I am proudest of my freelist memory allocator because it's the first "proper" memory allocator I've built to date, and of the timer logic because time handling is something I've always left up to the standard library/runtime of the languages I use.

All in all, I've really enjoyed writing a kernel and am really proud of the amount of progress I've made in the 3 weeks I've worked on it.

V Future work

There's still a lot to be done until my kernel can successfully run Doom, Notable obstacles are VGA Graphics Mode, a user space, file system, syscalls and a libc port.