## Build systems and tutorials

I looked at the files in a C build system from RPi4OS and Phillip Opperman's Rust OS tutorial to get explanations of what files to look for and what their purpose was to build a `kernel8.img` file. The main files to worry about were the assembly `*.s` files and the platform specific linker `*.ld` files. The `*.s` assembly is prepended to the assembled Rust kernel; it contains the bootloader code to setup and load the kernel entry point. The `*.ld` files linked (nonexistent in our case) OS libraries for a target architecture and OS and ABI.

After a bit of extra setup not necessary in C, Rust has out-of-the-box support for building to a bare-metal target. The `rustc` compiler supports the "target triple" `aarch64-unknown-none` without needing to download any extra toolchains manually, and without using Makefiles and specifying linker files. There exists a tool to prepend an `x86` bootloader to a kernel built this way, but none I could find for `AArch64`. For now, I decided to not work on this and use the build system for Raspberry Pi 3 Model B+ from CS3210 from Georgia Tech with a few minor edits. These included changes to the command line arguments from `raspi3` to `raspi4`, `target-cpu` to `cortex-a72`, starting address of the kernel to `0x80000`, and GPIO base register physical address (for memory-mapped IO) to `0xFE000000`. This corresponds to slight differences from RPi3+ to RPi4, including a default "low-memory" mode.

An interesting point is that a lot of Rust code written for bare-metal can be in "unsafe" blocks with weaker static safety guarantees than Rust written with OS libraries (so it was not much better than just using C). This was usually done to emulate C (for example, using raw pointers used in C instead of smart pointers); otherwise, a lot of Rust's static safety features were still available even without the Rust standard library when building for bare-metal for other code (lifetimes, smart pointers, robust error handling, etc). This was an interesting exercise in learning a bit about what the OS offers a given language through `syscalls`, and what is innate to the language itself.

## Deliverables

CS3210 also has some great labs for Rust and the Rasberry Pi architecture, which I decided to work on since they were actually perfect for getting a better understanding of bare metal Rust OS development.

I ended up following their first lab to get my morse code program on the RPi4 and breadboard working, and I did some work following their second lab.

## Next steps

Do some more of the CS3210 labs. Start looking at what I can do with Redox and Theseus.

## Final deliverable

The next progress report will be the final one. Additionally, a demo presentation of all the work is in order. The demo could be in the format of a detailed writeup, and/or a live presentation.