Implementing WeenixOS

Mario Camacho

Spring 2023

1 Introduction

For my capstone project, I implemented the Weenix operating system as part of CS169 throughout the Spring 2023 semester, which involved completing five major parts of an operating system. The five major parts are: Processes and Threads, Drivers, the Virtual File System, the System V File System, and Virtual Memory, which were all implemented in C. The semester-long project improved my ability to read and debug large code-bases, think about how different parts of an operating system interact, and helped me understand how to better work with low-level systems when writing, reading, and debugging code.

2 Implementation

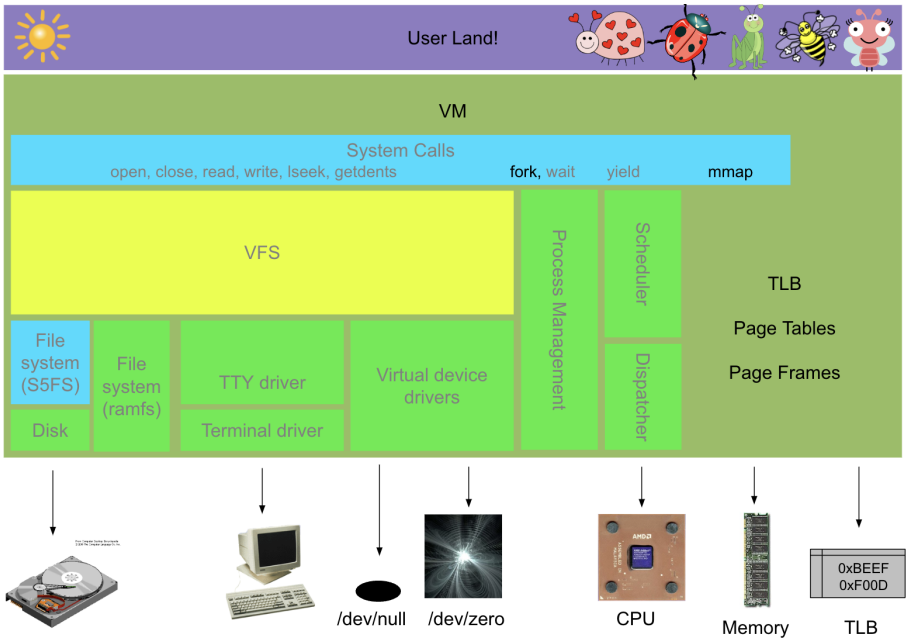

Figure 1: Weenix Architecture

2.1 Processes and Threads

To begin, we provided the basic building blocks for the Weenix operating system: threads, processes, and synchronization primitives. I implemented a simple FIFO scheduler for context switching, process, and thread cleanup, as well as the `waitpid` system call.

2.2 Drivers

For Drivers, we implemented the operating system's `tty`, which allowed interaction with the terminal, the SATA drivers that work with the disk, and the `/dev/null` and `/dev/zero` memory devices. I used a simple circular buffer that acts as the line discipline for the terminal. At this point, we are now able to use `kshell`, a `sh` like environment that now allows us to use commands.

2.3 Virtual File System

The virtual file system(VFS) provides a common interface between the operating system kernel and the various file systems. For VFS, I implemented the various system calls for files such as `open`, `read`, `write`, and others. In addition to this, we worked with virtual nodes to implement file locking and file reference counts for proper use and cleanup.

2.4 System V File System

The System V File System(S5FS) is a simple file system to interact with the Virtual File System we previously implemented. S5FS introduces the concepts of inodes, disk blocks, and writing to disk. It also extensively works with memory objects and page frames. Memory object represents some data source, which could be a file, and page frames are used to reference the blocks of data. All of this together allows us to read and write to the disk as well as use the system calls from VFS.

2.5 Virtual Memory

Finally, I implemented Virtual Memory to allow us to run binaries in userland. This turned Weenix into a more whole operating system since before this point we were running everything entirely within the kernel. Specifically, I implemented the logic for a virtual address space, shadow objects(for copy-on-write logic), anonymous objects(used for memory objects like the stack), system calls that connect kernel and userland(e.g. `fork`, `mmap`, and `brk`), and handling page faults to page in data into memory for address lookups. Implementing Virtual Memory was the hardest part since any bugs from the previous parts hide behind many layers of code making it difficult to debug.

3 Conclusion

WeenixOS is now able to run binaries in userland and support dynamic linking! :)

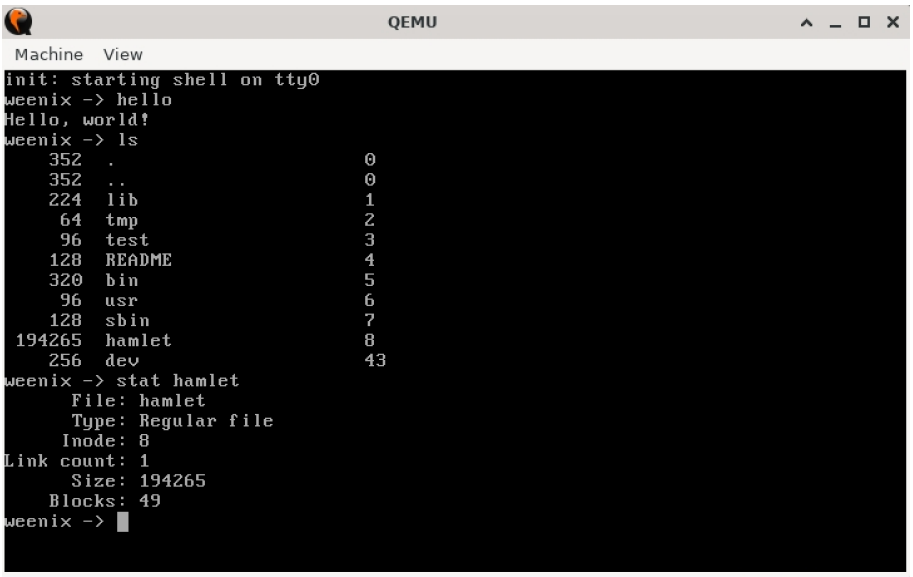

Figure 2: Weenix shell