Implementing Weenix OS

Alexander Yu

Spring 2020

1 Introduction

For my capstone, I implemented Weenix OS, a single-processor operating system, as part of CS 169, taught by Professor Doeppner. This assignment spanned the entire semester and was split into 5 major segments: Processes, Drivers, Virtual File System, S5 File System, and Virtual Memory. The project was written in C. My implementation of Weenix can run numerous user binaries and can support dynamic linking.

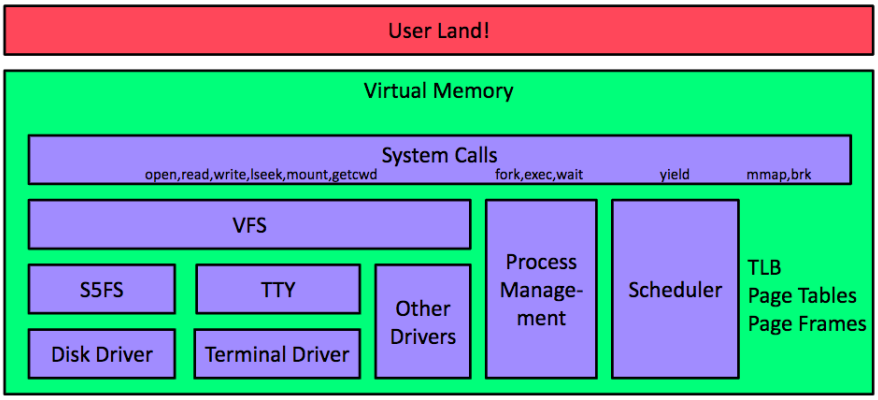


Figure 1: Weenix Architecture

2 Implementation

2.1 Processes

For Processes, I implemented the underlying structs for processes and threads. I also implemented a simple FIFO thread scheduler and process-related system calls such as `waitpid` and `exit`. Some important concepts I learned in this segment were how context switching worked and how processes were able to "sleep" and "wait" on each other.

2.2 Drivers

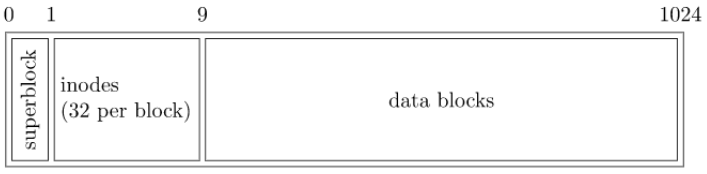
For Drivers, I worked on the operating system's terminal interface which consisted of reading from and writing to the operating system's TTYs. Additionally, I wrote code that interacted with the disk driver, `/dev/null` and `/dev/zero`. From this segment, it was interesting to learn that the kernel treats the terminal simply as a file in which it can perform file-related system calls on.

2.3 Virtual File System

For Virtual File System, I implemented the virtual API for all file-system related system calls (e.g `open`, `read`, `write`, `mkdir`, etc). In terms of the operating system architecture, this is necessary to provide an interface for the kernel to interact with different types of file-systems. In this segment, I learned how the kernel uses vnodes (virtual nodes) to implement the file hierarchy, file-locking, and file reference counts.

2.4 S5 File System

For S5 File System, I implemented the same interface from Virtual File System, but this time for an actual file system that had inodes and disk blocks. This segment helped me understand how a file system organizes its underlying data blocks and keeps track of free and used blocks.



The default disk layout for Weenix.

2.5 Virtual Memory

Lastly, for Virtual Memory, I implemented process virtual memory which included writing logic for the virtual address space, handling page faults, anonymous objects, shadow objects, memory mapped files, and memory-related system calls (e.g `mmap`, `brk`, and `fork`). All of this was necessary to run user code on Weenix OS. This part of the project tied the rest of the segments together and was the last step in completing a fully functioning operating system. It was by far the hardest part of the entire project.

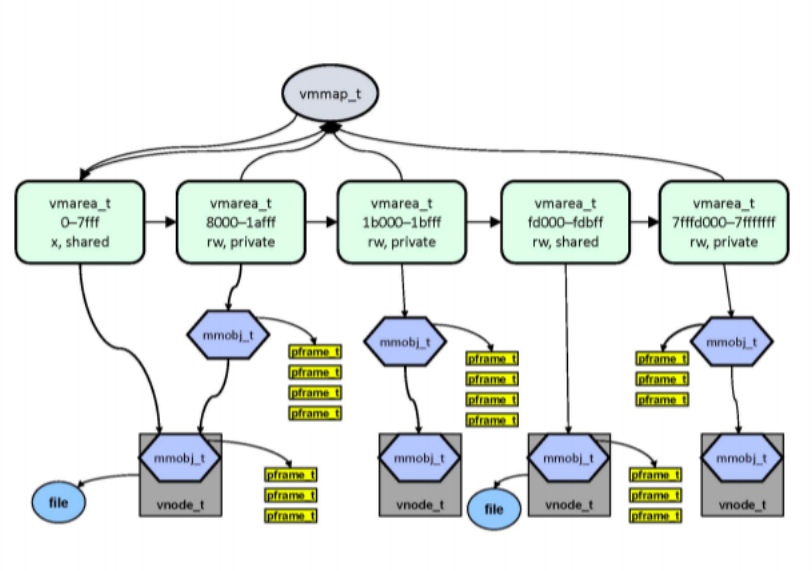


Figure 2: Example process virtual memory space

3 Conclusion

Overall, I thoroughly enjoyed working on Weenix OS. As a result of this capstone, I have greatly improved my understanding of how an operating system works and my ability to debug low-level code.