Writing Your Own Operating System from Scratch

Tom Suter

Supervisors:
Peter Skrotzky
Alan Metzler

November 5, 2023

Contents

1 Understanding Operating Systems

1.1 Acronyms and abbreviations

ABI	Application binary interface
API	Application programming interface
BIOS	Basic input/output system
CPU	Central processing unit
FCFS	First come first serve
FIFO	First in first out
GDT	Global descriptor table
GPU	Graphics processing unit
HDD	Hard disk drive
IDT	Interrupt descriptor table
KiB	Kibibyte (1024 bytes)
libc	C standard library
LIFO	Last in first out
MMU	Memory management unit
OS	Operating system
RAM	Random access memory
ROM	Read only memory
SSD	Solid state drive
Syscall	System call
VFS	Virtual file system
PIT	Programmable interrupt timer
PIC	Programmable interrupt controller

1.2 What is an OS?

An operating system (OS) is a foundational software component that serves as the intermediary between the hardware and the applications running on a computer system. It plays a pivotal role in managing and controlling various aspects of a computer's functionality, ensuring that users and applications can interact with the hardware effectively, securely, and reliably. I will explore the functions of an OS and how the protection ring model, with its distinct privilege levels (rings), helps govern access to system resources. Additionally, I will delve into Microsoft's experimental OS, Singularity, which pushed the boundaries of traditional OS design by running all code in the most privileged ring, ring 0.

The boot process initiates with the BIOS[1], followed by the bootloader, which loads the OS. Initially, the OS boots in 16-bit real mode, a legacy mode that provides a simple environment for early initialization. Subsequently, an assembly stub or bootloader transitions the system to 32-bit protected mode, providing memory protection. Later, the OS may further switch to 64-bit long mode[2], offering support for more extensive memory addressing and advanced features. To facilitate this transition, key elements like GDT, Page Table, and code execution setup need to be appropriately configured[3].

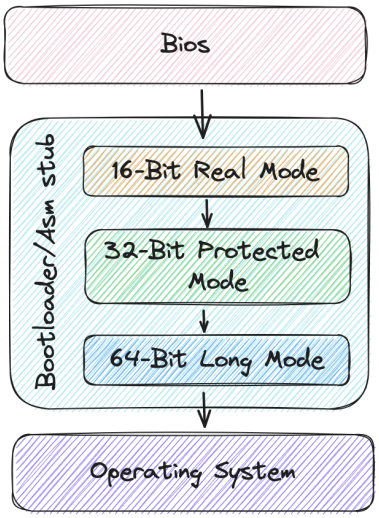

Figure 1: Boot process

At its core, an operating system serves as the backbone of modern computing by orchestrating the allocation of hardware resources, managing processes, providing file and device access, and ensuring security and stability. It does so by interfacing with various components of the computer system, including

[1] Osdev.org, 2019a
[2] Oppermann, 2015a
[3] thomasloven, 2019

the CPU, memory, storage devices, input/output (I/O) peripherals, and network interfaces. Through a well-defined and controlled set of interfaces and abstractions, the OS abstracts the complexity of hardware, making it easier for software applications to interact with the underlying resources[4].

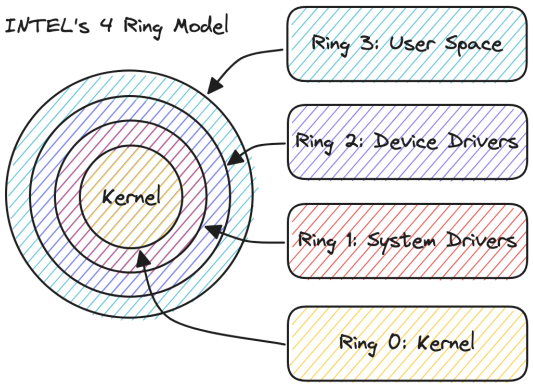

Figure 2: Intel's ring model

The protection ring model One fundamental concept in understanding how an OS operates is the protection ring model. In this model, the system's privileges are divided into different rings, typically numbered from 0 to 3. Each ring represents a different level of privilege, with ring 0 being the most privileged and ring 3 being the least privileged[5].

Ring 0 (Kernel): Ring 0 is the innermost and most privileged level in the protection ring model, As this is where, the OS kernel operates. The kernel has unrestricted access to hardware resources and can execute sensitive and critical operations, such as managing memory, scheduling processes, handling hardware interrupts, and interacting directly with hardware devices. It serves as the gatekeeper to system resources, ensuring that tasks are performed securely and efficiently[6].

Ring 1 (System drivers): Ring 1 is typically reserved for system drivers, which are software components responsible for managing and controlling system-level devices and functions. These drivers operate with a higher level of privilege than user-level applications (Ring 3) but are restricted from directly accessing certain kernel functions and hardware resources. They pro-

[4]GeeksforGeeks, 2018a
[5]GeeksforGeeks, 2020
[6]Pearsonitcertification.com, 2013

vide an additional layer of isolation and protection, preventing errant drivers from compromising the stability of the kernel[7].

Ring 2 (Device drivers): Ring 2, if implemented, is designated for device drivers that are less critical than system drivers but more privileged than user-level code. However, not all protection ring models include this level, as its usage can vary depending on the OS design[7].

Ring 3 (User space): Also known as user-land, Ring 3 represents the least privileged level in the protection ring model, where user-level applications run. Applications running in ring 3 have the most restricted access to hardware resources and must make system calls to the kernel in order to perform privileged operations like file access, memory management, and I/O operations. This level of isolation ensures that user applications cannot directly interfere with the kernel or other processes[7].

It is worth noting that while the protection ring model provides a robust security framework, some experimental OS designs challenge this convention. For instance, Microsoft's experimental OS, Singularity, is known for running all code at Ring 0. Singularity adopts a fundamentally different approach to security by leveraging advanced programming languages and techniques to ensure software isolation and reliability. This approach eliminates the traditional distinction between user and kernel space, focusing on fine-grained, language-based security mechanisms. While Singularity remains an experimental project, it has sparked discussions about alternative OS design paradigms that challenge traditional notions of privilege and security models[8].

Kernel design paradigms

When discussing OS kernel design, several paradigms come into play, including microkernels, monokernels, and unikernels, each with their own strengths and weaknesses.

[7]Pearsonitcertification.com, 2013

[8]Microsoft Research, 2023

- **Microkernels** are designed to minimize the size and complexity of the kernel by delegating most OS functions to user-level processes or servers. They prioritize modularity and separation of components, enhancing system reliability and maintainability. Examples of microkernel-based operating systems include QNX and Minix[9].

- **Monokernels** are more traditional in their approach, encompassing a single, unified kernel that handles a broad range of tasks, including device drivers and system services. While this design may sacrifice some modularity, it can provide higher performance and efficiency. Popular monokernel-based OSs include Linux and Windows[9].

- **Unikernels** are specialized, minimalistic kernels tailored to run a single application or service. They aim to optimize resource utilization and security by stripping away unnecessary components. Unikernels are well-suited for cloud computing and virtualization environments where lightweight, purpose-built OS instances are advantageous[10].

In summary, an operating system is a fundamental software layer that plays a crucial role in computer systems by managing resources, enforcing security, and facilitating communication between users, applications, and hardware. The Protection Ring Model serves as a fundamental security mechanism, categorizing and protecting different levels of system privileges. Meanwhile, innovations like Microsoft's Singularity challenge traditional paradigms by reimagining OS design. Additionally, various kernel design paradigms, such as microkernels, monokernels, and unikernels, offer distinct trade-offs in terms of modularity, performance, and specialization. These diverse approaches collectively contribute to the evolution and versatility of operating systems, shaping the way we interact with and utilize computers in an ever-changing technological landscape.

[9] Bigelow and Lulka, 2022
[10] Unikernel.org, 2023

1.3 What does an OS have to implement?

To achieve its task as an intermediary as well as its managing functions, an OS must implement a wide range of functionalities, each serving a specific purpose. This part delves into the fundamental components and responsibilities that an OS has to implement Networking, Memory Management, Scheduling, File System Services, Device Management, Authentication and Security, Error Handling, CPU Time Management, and Software Layers for Applications to Run On. Together, those form the backbone of any modern operating system, ensuring that the computer system operates effectively, securely, and reliably[11].

Memory management
Memory management is a critical aspect of computer systems that plays a key role in ensuring the efficient and reliable operation of modern computing devices. It encompasses a wide array of concepts and mechanisms, all designed to effectively handle the allocation, organization, and utilization of a computer's memory resources.

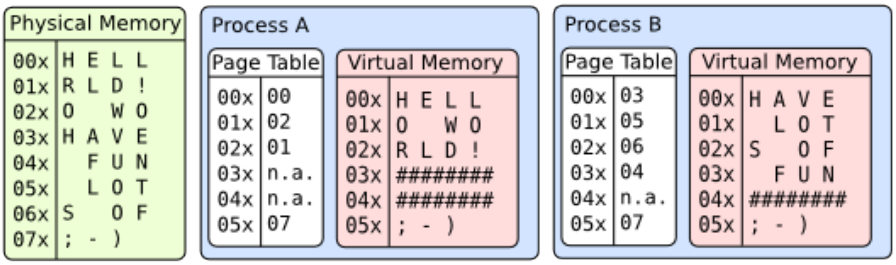

Figure 3: Virtual memory model[12]

At its core, memory management is vital because it enables a computer to make the most of its memory resources, ensuring that programs run smoothly, effectively, and without the risk of data corruption or system crashes. It is like the brain of a computer, orchestrating the allocation and deallocation of memory spaces as needed by various processes and applications. Memory in a computer is organized into physical and virtual addresses. Physical mem-

[11] GeeksforGeeks, 2018a
[12] Jhawthorn, (2006) Virtual memory

ory refers to the actual hardware memory chips installed in the computer. In contrast, virtual memory is an abstraction that provides a more extensive address space than the physical memory available. Virtual memory allows the operating system to use secondary storage, such as a hard drive, as an extension of physical memory. This enables powerful multitasking, as processes can share the same physical memory while having their own virtual memory spaces[13].

One essential concept in memory management is paging. Paging divides physical memory into fixed-size blocks called pages, usually the size of 4 KiB, and divides the logical memory (used by programs) into equally sized blocks called frames. The operating system keeps track of which pages are stored in which frames. When a program needs to access a specific memory location, the MMU translates the virtual address to a physical address using this mapping. Paging simplifies memory allocation, allows for easy swapping of pages in and out of physical memory, and facilitates memory protection[13].

The distinction between the stack and the heap is another crucial aspect of memory management. The stack is used for the storage of function call information, local variables, and control flow data. It is a highly organized and efficient structure with a fixed size Memory allocation and deallocation follow a strict last-in, first-out (LIFO)[14] order. In contrast, the heap is used for dynamic memory allocation, where the size and lifetime of memory are not known in advance. Managing the heap involves more complex algorithms and data structures to keep track of allocated and deallocated memory blocks[15].

Effective memory management helps prevent common issues like memory leaks that can lead to program crashes or security vulnerabilities. It also plays a role in optimizing system performance by minimizing the overhead associated with memory allocation and deallocation[16].

Therefore, memory management is the backbone of computer systems, ensuring the orderly allocation and utilization of memory resources. It involves the use of physical and virtual addresses, paging mechanisms, and the care-

[13]GeeksforGeeks, 2021
[14]GeeksforGeeks, 2018b
[15]Oppermann, 2019b
[16]Osdev.org, 2023c

ful management of the stack and heap. By efficiently managing memory, computer systems can deliver optimal performance, stability, and security, making memory management a fundamental pillar of modern computing.

Networking
Networking is a key component of modern computing, enabling communication between devices and systems via local and global networks. An operating system must incorporate networking capabilities to facilitate data transfer, internet connectivity, and communication among diverse applications and devices. This involves managing network protocols, configuring network interfaces, handling data packets, and providing APIs for network communication. Additionally, the OS is responsible for resolving domain names to IP addresses (DNS resolution), managing network connections (establishing, maintaining, and terminating connections), and ensuring data integrity and security through protocols like TCP/IP and encryption[17].

Effective networking support is crucial for tasks such as web browsing, email communication, file sharing and online gaming. The OS must balance performance, security, and user-friendliness when implementing networking features, as users expect seamless connectivity while maintaining their privacy and security.

Scheduling
Scheduling is the process of determining which processes or threads should run on a CPU core and for how long. An OS must implement scheduling algorithms to effectively manage the allocation of CPU time to different processes. Scheduling ensures fair distribution of processing resources, responsiveness, and adept utilization of CPU cores in a multitasking environment.

Common scheduling algorithms include FCFS also known as FIFO, Round Robin, Priority Scheduling, and Multilevel Queue Scheduling. Each algorithm has its advantages as well as its disadvantages, and the OS must choose the most appropriate one based on the system's requirements and priorities[18].

[17]Osdev.org, 2019b
[18]Tutorialspoint.com, 2023

Scheduling plays a critical role in maintaining system responsiveness and overall performance. A capable scheduling system ensures preventing one misbehaving process from monopolizing the CPU and ensures that time-critical tasks are executed promptly.

File system

File system services play a fundamental role in the realm of computing, serving as the backbone for managing and organizing data across various storage devices, including HDDs and SSDs. Operating systems rely on file systems to provide a structured and effective means of handling files and directories, encompassing tasks such as creating, reading, writing, deleting, organizing data, and managing file permissions and access control[19].

One noteworthy aspect of file systems is the diversity in their designs, each tailored to specific use cases and requirements. Some named file systems include FAT32, NTFS, APFS, Btrfs, and Ext4, each of them with their unique features and characteristics. Understanding the distinctions among these named file systems is essential for making informed decisions when selecting a file system for a particular storage medium or operating environment.

FAT32 (File Allocation Table 32) is a relatively straightforward file system known for its compatibility across various operating systems and devices. However, it has limitations, including a maximum file size of 4 GB and a lack of support for advanced features such as encryption and journaling, making it more suitable for basic storage needs[20].

NTFS (New Technology File System) is a proprietary file system developed by Microsoft, offering enhanced capabilities compared to FAT32. It supports large file sizes, robust security features, file compression, and journaling, making it a robust choice for Windows-based systems and environments where data integrity and access control are paramount[21].

APFS (Apple File System) is specifically designed for macOS and iOS devices, focusing on efficiency and data management. It features copy-on-write

[19] Osdev.org, 2023b
[20] C-Bit.org, 2020
[21] Osdev.org, 2023d

snapshots, native encryption, and optimization for SSDs, contributing to improved performance and reliability in Apple's ecosystem[22].

BTRFS (B-tree File System) is a Linux-native file system that emphasizes data integrity and flexibility. It offers features like built-in RAID, snapshots, and transparent compression, making it suitable for servers and systems where resilience and scalability are critical[23].

Ext4 (Fourth Extended File System) is an evolution of the Ext3 file system and is popular in the Linux world. It maintains compatibility with Ext3 while introducing enhancements such as support for larger file sizes and improved file system management. Ext4 is a dependable choice for Linux-based systems[24].

The choice of file system has a profound impact on various aspects of data management, including file size limits, data integrity, support for encryption and compression, and compatibility with different operating systems. Selecting the most appropriate file system for a particular scenario is essential to ensure efficient data storage, retrieval, and security.

Effective file system services are crucial for data stability and organization. They enable users and applications to reliably store and retrieve information while ensuring data security and integrity.

Device management
Device management is the process of controlling and interacting with hardware devices connected to a computer. It includes input/output devices (e.g., keyboards, mice, monitors), storage devices (e.g., hard drives, USB drives), and communication devices (e.g., network adapters, Bluetooth devices). An OS must implement device management to facilitate communication between software and hardware components[25].

[22]Hutchinson, L, 2016
[23]Kernel.org, 2023a
[24]Kernelnewbies.org, 2017
[25]javatpoint.com, 2021

Device drivers are software components that bridge the gap between the OS and hardware devices, enabling the OS to send commands to devices and receive data from them. The OS must detect, install, and manage these device drivers, ensuring that hardware components function correctly[26].

Proper management of devices is crucial to ensure hardware compatibility, device configuration, and the preservation of system stability. In the absence of adequate device management, the operating system would struggle to efficiently communicate with hardware components.

Application layer

The OS application layer, as depicted in the diagram, is important because it provides a bridge between user-level software applications and the underlying hardware resources of a computer system. It is a critical component of the operating system architecture and serves as the intermediary layer that facilitates communication, resource management, and the execution of user programs[27].

The OS application layer, as depicted in the diagram, plays a pivotal role in providing a bridge between user-level software applications and the underlying hardware resources of a computer system. It is a critical component of the operating system architecture and serves as the intermediary layer that facilitates communication, resource management, and the execution of user programs.

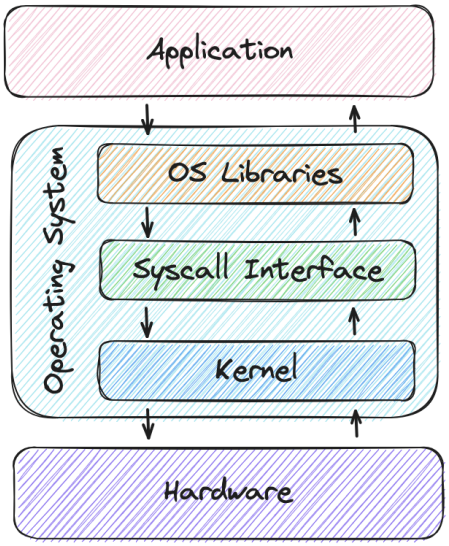

Figure 4: Application layer model

At the outermost level, we have the "User" and "Application" components. These represent the end-users and the software applications they interact with.

[26]javatpoint.com, 2021
[27]Tutorialspoint.com, 2021

User-level applications include everything from word processors and web browsers to video games and scientific simulations. These applications rely on the services and functionalities provided by the OS application layer to perform their tasks effectively[28].

Moving inward, we encounter the OS application layer itself, which comprises several key components. The "OS libraries" consist of pre-written code modules and libraries that offer commonly used functions and services to applications as API. These libraries simplify development by providing high-level abstractions and standardized interfaces for tasks such as file I/O, network communication, and graphical user interfaces[28].

The "Syscall interface" acts as the gatekeeper between user-level applications and the kernel, enforcing access controls and ensuring that system calls adhere to predefined interfaces. System calls allow user-level programs to request services or access resources provided by the OS kernel, such as managing processes, file operations, or network communication[28].

As mentioned earlier, the protection ring model typically consists of multiple privilege levels (rings), with Ring 0 being the most privileged (kernel mode) and Ring 3 being the least privileged (user mode). The OS kernel runs in Ring 0, while user-level applications run in Ring 3. This strict separation prevents user-level code from directly accessing or tampering with critical system resources[28].

The OS application layer ensures that user-level programs can make controlled requests to the kernel, thereby leveraging the full potential of the hardware without compromising system stability or security. By providing a structured and controlled interface to the underlying resources, the OS application layer enables the creation of diverse applications while maintaining the integrity of the overall system. It also plays a crucial role in resource management, process scheduling, and security enforcement, thereby contributing to the efficient and secure operation of the computer system[28].

[28]Tutorialspoint.com, 2021

In summary, the OS application layer serves as a vital intermediary, allowing user-level applications to interact with the kernel and hardware resources in a controlled and secure manner. It adheres to the protection ring model, which helps maintain the integrity and security of the system by segregating different levels of privileges. This architectural design ensures that the OS can deliver the necessary services and maintain system stability while enabling a rich environment of user software.

Authentication and security
Authentication and security are important concerns for operating systems. An OS must implement security measures to protect user data, prevent unauthorized access, and ensure the integrity of system resources. This includes user authentication through methods such as passwords, biometrics[29], or multifactor authentication (MFA)[30].

A robust authentication and security framework is vital for safeguarding user privacy, preventing data breaches, and maintaining the overall integrity of the system.

Error handling
Error handling is a critical aspect of OS design, encompassing various mechanisms to detect, respond to, and recover from errors or exceptional conditions that may occur during the execution of computer programs. Three essential components of error handling in an OS are CPU exceptions, hardware interrupts, and double faults[31].

CPU exceptions, also known as traps or exceptions, are events that occur during the execution of a program and disrupt the normal flow of instructions. These exceptions can be caused by various factors, such as division by zero, invalid memory access, or the attempt to execute privileged instructions in user mode. When a CPU exception occurs, the processor transfers control to a specific exception handler routine provided by the OS[31].

[29]Trick, 2022
[30]Keerthi Chinthaguntla, 2020
[31]Oppermann, 2018a

Exception handlers are responsible for diagnosing the cause of the exception, taking appropriate actions to resolve or mitigate the issue, and allowing the program to continue execution if possible. For example, a divide-by-zero exception might trigger the OS to smoothly terminate the malfunctioning process, preventing it from crashing the entire system[31].

Hardware interrupts are signals generated by external hardware devices to request the attention of the CPU. These interrupts are used for various purposes, such as input/output (I/O) operations completion, timer events, or hardware errors. When a hardware interrupt occurs, the CPU temporarily suspends its current execution and transfers control to a specific interrupt handler routine, also provided by the OS[32].

Interrupt handlers are essential for managing hardware interactions and ensuring timely responses to external events. For instance, a keyboard interrupt handler would process keystrokes and pass them to the appropriate application, allowing users to interact with the system. Similarly, a timer interrupt handler may be responsible for preemptive multitasking, allowing the OS to switch between running processes to ensure fairness and responsiveness[32].

A double fault is a specific type of exception that occurs when the CPU attempts to handle one exception but encounters another exception while doing so. Double faults often indicate severe system issues that may result in a system crash or instability. They can be triggered by a variety of conditions, such as a page fault during the handling of another exception or a stack overflow in the exception handler itself[33].

Handling double faults is particularly challenging for the OS, as it must ensure the system's stability while dealing with the underlying issue that caused the initial fault. Typically, double fault handlers are designed to log diagnostic information, halt the system, and potentially initiate a reboot to prevent data corruption or further damage[33].

[32]Oppermann, 2018c
[33]Oppermann, 2018b

To summarize, error management in an operating system entails managing CPU exceptions, hardware interrupts, and double faults to maintain system stability and reliability. These mechanisms ensure that the OS can respond to unexpected events, protect user data, and prevent a single faulty program or hardware component from causing a catastrophic failure of the entire system. Effective error handling is a critical aspect of OS design, contributing to the overall robustness and resilience of the computing environment. Effective error handling ensures that the OS can smoothly handle unexpected situations, reducing downtime and improving the overall user experience.

2 Implementation process

2.1 Languages and tooling

Creating an operating system from scratch is a challenging endeavor that requires careful consideration of language and tooling choices. In my journey to develop an operating system, I opted for Rust as my primary programming language, driven by its unique combination of memory safety, execution speed, and my proficiency in it. Additionally, Rust's high-level abstractions with low-level control and extensibility using assembly were compelling factors in my decision-making process.

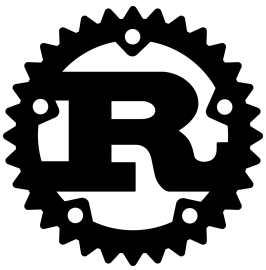

Figure 5: Rust Logo[34]

One of the primary reasons I chose Rust as the cornerstone of my operating system project was its remarkable focus on memory safety without compromising execution speed. This is akin to the performance of C. Memory safety is paramount in operating system development. A single memory-related error can lead to system crashes or vulnerabilities. Rust's ownership system and strict borrowing rules empower developers to write code that is less prone to memory-related issues, making it an excellent choice for systems programming[35].

Another critical advantage of Rust is its lack of overhead associated with a garbage collector. Traditional programming languages like Java or C# rely on garbage collection to manage memory, which introduces runtime overhead. In contrast, Rust's ownership model allows for deterministic memory management without runtime penalties, ensuring predictable and productive memory usage[35].

Furthermore, my proficiency in Rust played a significant role in my decision. Being proficient in a language expedites development, reduces the likelihood of bugs, and enables me to focus on the intricate details of operating system design and functionality rather than wrestling with the language itself.

[34]Rust-lang.org, 2023b
[35]Rust-lang.org, 2018

One of Rust's defining features is its ability to provide high-level abstractions while retaining low-level control. This duality is crucial in operating system development, as it allows me to work with abstractions when convenient while dropping to a lower level when necessary, such as when interacting with hardware or writing assembly code for specific tasks[36].

Rust's ecosystem and the power of Cargo, its package manager and build tool, were also compelling factors. Cargo simplifies dependency management and automates tasks like compilation, testing, and documentation generation[38]. Furthermore, Rust's seamless integration with LLVM and the GNU Assembler (GAS) meant that I could efficiently cross-compile without the need to install an external assembler, streamlining the development process[39].

Figure 6: Cargo Logo[37]

To managing the build process of my operating system, I chose Cargo Make[40], a versatile build tool that combines the flexibility of a traditional makefile with the convenience and integration of Cargo. This choice allowed me to define custom build steps, automate complex tasks, and maintain a clean and organized project structure.

Figure 7: Nix Logo[41]

Setting up my development environment was a crucial step in this journey. To achieve consistency and reproducibility, I turned to Nix, a powerful and declarative package manager[42]. With Nix, I could create a configuration that specified all the dependencies and tools required for my project. This meant I could quickly set up my development environment on any machine where Nix was installed, ensuring that my development environment remained consistent across different platforms.

[36] Rust-lang.org, 2018
[37] Rust-lang.org, 2023b
[38] Rust-lang.org, 2021
[39] Rust-lang.org, 2023a
[40] sagiegurari, 2017
[41] Tim Cuthbertson, 2017
[42] Nixos.org, 2023

While Nix served as an excellent tool, I did not directly use it as my build system. This was due to some challenges I encountered with the immutability of build environments when attempting to forbid Cargo from downloading crates. Instead, I relied on Cargo Make for build automation and Nix for environment provisioning, striking a balance between the two.

Finally, as my text editor of choice, I embraced the productivity and extensibility of Neovim (NVim). The Vim key bindings enhanced my productivity, while NVim's extensive plugin ecosystem allowed me to tailor my development environment to suit my specific needs[44]. Additionally, its minimalistic design and low resource usage made it a compelling choice over more resource-intensive editors like Visual Studio Code.

Figure 8: Neovim Logo[43]

My decision to choose Rust as the primary programming language, along with the supporting toolchain and environment, was driven by the language's memory safety, execution speed, my proficiency, and its ecosystem. By combining Rust with the right tooling, I embarked on a fulfilling journey to develop my operating system, confident in the choices I made to ensure efficiency, reliability, and maintainability throughout the project.

2.2 What have I implemented?

In the pursuit of developing an operating system, the project was driven by the aspiration to achieve specific goals. The latter encompassed both the minimum expectations and extended functionalities, all within the context of fundamental software objectives. The minimum goals included the ability to boot the operating system and print information, while the extended objectives involved memory management, text I/O, error handling, and scheduling. The overall objective was to build an operating system that

[43] Jason Long, 2015
[44] Neovim.io, 2023

not only fulfilled these project-specific goals but also aligned with the overarching software goals for a functioning and comprehensive OS.

Memory management

At the heart of this project lies the vital concept of memory management, a pivotal component in any operating system, as it is responsible for the efficient allocation and safeguarding of memory resources. The approach opted for, hinged upon the implementation of a recursive 4-level page table, a widely adopted and versatile technique in the realm of virtual memory systems.

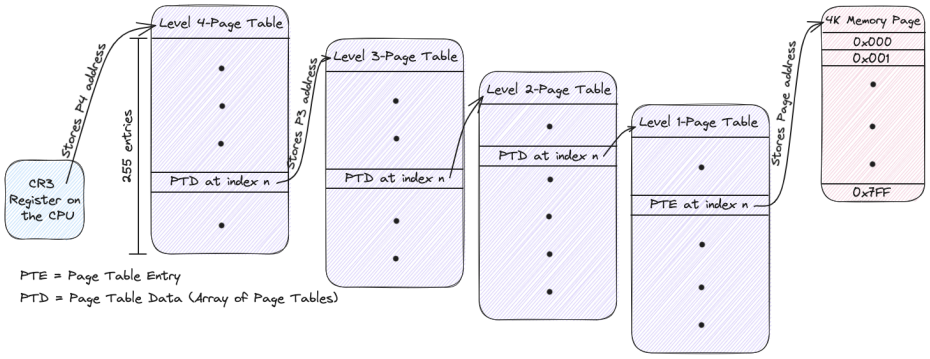

Figure 9: 4-level paging

A recursive page table works by establishing a hierarchical structure of page tables, akin to a tree-like configuration. This design simplifies the intricate task of managing memory through recursive traversal. Each page table at different hierarchical levels serves a specific purpose, contributing to the efficient memory address translation process. Page tables 1 to 4 are structured in a hierarchical manner, with each page table containing entries for the next level. Notably, the level-1 page table only has page entries and plays a fundamental role in the virtual-to-physical address translation process[45].

Central to this mechanism is the CPU's CR3 register, which stores the address of the level-4 page table. This register serves as a critical reference point for the operating system, enabling it to efficiently traverse the hierarchical page tables and to determine the actual physical address associated with a given virtual address[46].

[45] GeeksforGeeks. 2019

[46] Oppermann, 2019b

The utilization of recursive page tables results in a comprehensive and effective memory management system. It not only grants fine-grained control over memory allocation and protection, but also simplifies the translation of virtual addresses to their corresponding physical addresses. By successfully implementing this memory management strategy, the project laid a robust foundation for resource management within the operating system[47]. This achievement enhances the OS's capability to concurrently run multiple processes, all while maintaining the integrity and efficiency of data management.

Scheduling

Scheduling, another essential component of the project, plays a crucial role in determining how the operating system allocates and manages CPU resources among multiple processes. Scheduling is pivotal in achieving multitasking and fairness in process execution[48]. The project successfully integrated a cooperative multitasking mechanism using Rust's futures, allowing the OS to allocate CPU time to different processes efficiently. This approach ensures that no single process monopolizes system resources[48], promoting a balanced and responsive system. The implementation of cooperative multitasking significantly enhances system responsiveness and overall performance, bringing the project closer to achieving the goal of a fully functional operating system.

Error handling

Error handling is a crucial and foundational aspect of ChadOS, my operating system's design. It plays a pivotal role in ensuring system reliability, encompassing the identification, diagnosis, and recovery from errors or exceptional conditions that may arise during program execution[49]. To achieve robust error handling, ChadOS leverages Rust's powerful Option and Result types, complemented by the panic! function and my custom eprintln! macro for error reporting. Additionally, the wprintln! macro is used for handling warnings. These mechanisms collectively fortify the OS's stability and guarantee a seamless and graceful response to unforeseen events, reducing the risk of system crashes and data corruption.

[47] Oppermann, 2019b
[48] Tutorialspoint.com, 2023
[49] Oppermann, 2018a

Text I/O
Text input/output (I/O) was a project goal that aimed at enabling the OS to effectively communicate with users and other applications[50]. Remarkably, not only was the input goal met but it was definitely surpassed. The OS now supports a total of 8 different keyboard layouts, greatly enhancing its versatility. However, it's worth noting that there is room for improvement in handling certain special characters like tabs.

On the output front, the project didn't just meet the goal; it went above and beyond. The implementation of dynamic color switching allows ChadOS to print in 16 different foreground and background colors, offering an array of visual options for user interaction. Furthermore, the addition of various print methods, such as wprintln for warnings and kprintln for kernel messages, adds flexibility and clarity to the communication process. In addition, the implementation of printing via serial ports further enhances the OS's communication capabilities. This accomplishment signifies a substantial leap toward creating a more interactive and communicative operating system, with the potential for further enhancements in text I/O capabilities in the future.

Testing
To complement these specific project goals, a test suite was devised. This test suite assumed a critical role in the development process and included a systematic evaluation of the implemented features to expose potential bugs. Furthermore, it ensured that the realized goals were not only met but were met with robustness and reliability. Testing was carried out rigorously to validate the functionality and stability of the implemented components, highlighting the importance of thorough quality assurance in OS development[51].

Custom shell
A notable and groundbreaking addition to the project was the development of CheapShell, a custom shell that introduces a unique and captivating dimension to the project. Although not initially part of the original project objectives, CheapShell was conceived to enhance the project's innovation. What truly sets CheapShell apart from traditional shells is its distinctive approach.

[50]Uic.edu, 2023
[51]Akhtar, 2023

Beyond the traditional concept of opening executable files with respective arguments at runtime[52], CheapShell stands out by supporting the chaining of commands via Posix-like pipes, offering advanced functionality. Furthermore, CheapShell empowers users to define functions within the usr_bin module and register these functions as custom programs during the compile-time process, a novel approach to user-defined functions that distinguishes CheapShell from conventional shells[51].

Time
Additionally, similar to the implementation of the custom shell, an innovative feature was incorporated into the OS that was not initially part of the project's goals. The project introduced a timekeeping mechanism that enables the OS to manage time-related functions. This functionality includes the ability to implement sleep durations and retrieve the system's uptime. These time-related features were achieved by setting up handlers for the PIT, showcasing the project's adaptability and commitment to enhancing the OS's capabilities beyond the initial objectives.

Realized project objectives
In addition to successfully meeting the core project goals, the project not only exceeded expectations but also ventured well beyond the initial objectives. The goals encompassed memory management, cooperative multitasking, robust error handling, and text I/O enhancements. These objectives led to the development of a responsive operating system with extensive error-handling capabilities. Moreover, the OS's I/O features went beyond basic functionality, providing support for 8 keyboard layouts and dynamic color printing. The inclusion of various print methods for clarity and the implementation of printing via serial ports expanded the OS's communication abilities. Furthermore, the creation of a test suite ensured that all realized goals were not only met but were executed with impressive robustness and reliability. The innovative introduction of CheapShell, enabling advanced command chaining and user-defined functions at compile time, significantly elevated the OS's capabilities, reinforcing its status as an extraordinary and multifunctional system. Additionally, the implementation of time-related functions, such as sleep and uptime retrieval, went absolutely above and beyond, further enhancing the

[52]Aosabook.org, 2023

OS's versatility and utility, mirroring the innovative spirit demonstrated by the inclusion of CheapShell.

2.3 Methodology

In my approach to implementing various aspects of the operating system, I initially drew inspiration from Phil Oppermann's blog, "Writing an OS in rust." This resource served as a valuable starting point, offering insights into creating a basic freestanding Rust binary using the bootloader crate for OS bootstrapping. The blog provided guidance on fundamental OS components like output handling, exception management, and memory management[53]. However, I made significant changes and improvements to adapt it to my project's needs.

One of the early decisions I made was to switch bootloaders multiple times. While the initial blog used the bootloader crate, which generated .bin files, I opted to transition to different bootloaders, including Grub2[54] and Limine[55]. These transitions involved extensive exploration of the OSDev Wiki and examination of various kernel codebases. Although my initial attempts were met with challenges due to a lack of essential knowledge, these experiences pushed me to refine my understanding and make informed decisions.

In the first major rewrite (Rewrite 1), I focused on enhancing the existing codebase from the blog. One notable improvement was the introduction of a centralized configuration module, 'src/cfg.rs'. Additionally, I introduced various printing methods, such as eprintln! for error reporting, inspired by Rust's built-in eprintln! macro. Recognizing the importance of providing clear warnings, I also created wprintln!. To streamline kernel message printing, I devised kprintln!, inspired by similar functionalities in C-based kernels and the Linux kernel[56]. Moreover, I introduced the rprint! macro, allowing for dynamic text replacement within a line, resembling modern shell behavior.

A significant evolution in my methodology was the transition of my build system. Starting as a Haskell program that checked for dependencies, it

[53] Oppermann, 2022
[54] Dubbs, 2018
[55] Limine-bootloader, 2023
[56] Kernel.org, 2023b

evolved into a shell script, then a Makefile, and finally, I adopted Nix to set up the development environment. Cargo Make became my tool of choice for building the OS, streamlining the build process.

Additionally, my approach to time implementation was guided by a need for precision and accuracy. Initially, I faced challenges with time calculations, resulting in miscalculated intervals where minutes appeared to be 24 seconds longer. In my quest for a solution, I stumbled upon Bran's kernel development series[57], where he meticulously crafted a 32-bit kernel in C. While exploring his time implementation, I recognized an opportunity to reimagine and adapt it to my Rust-based system, tailored to work seamlessly with my interrupt handlers. This endeavor culminated in the development of my time implementation, which aimed to deliver accurate timekeeping within the operating system.

Later, I embarked on the task of amalgamating the most advantageous elements of the three rewrites, all while incorporating my distinct enhancements. The foundation for text output and paging was drawn from the blog but underwent significant optimization to bolster performance. When it came to exception handling, I delved into a multitude of kernel sources for inspiration, taking cues from k4dos, MOROS, various C-based kernels, Hermit-rs, Redox, and the blog. My exception handling approach, similar in spirit to MOROS and the blog, featured notable improvements, including the utilization of more suitable data structures and performance enhancements such as the implementation of Lazy initialization for the Programmable Interrupt Controller (PIT). Notably, I made the PIT more efficient by removing unnecessary mutexes, resulting in enhanced performance and code clarity. Additionally, I restructured the codebase by relocating global variables to the config file and segregating the handler functions, further enhancing code organization and readability.

Another significant enhancement involved the heap allocator. While the blog initially employed a bump allocator, I delved into research and experimentation with different allocators like jemalloc, dlmalloc, galloc, and slab-malloc. Ultimately, I adopted galloc from the good_memory_allocator crate, boasting nine times the efficiency of the linked_list allocator used in the bump alloca-

[57]Bran, 2023

tor[58]. To ensure flexibility, I introduced a feature-based switch to allow users to choose between galloc and the original bump allocator.

In the realm of scheduling, I adhered to the blog's approach, employing Rust's futures to implement cooperative multitasking. While my changes were relatively minimal, I relocated sensitive options to the global configuration, ensuring maintainability and ease of use.

The keyboard implementation, inspired by MOROS's design, featured some key modifications. I introduced the ability to replace handler functions, making the console more adaptable. Additionally, I expanded keyboard layout support to a total of eight layouts and provided an abstraction layer for users to change layouts without delving into the implementation details.

For testing purposes, I developed a testing suite based on the test! macro, ensuring uniformity in test cases. Each test followed a consistent naming scheme, making it easy to identify their purpose and assertions.

One of the most innovative aspects of my project was the user shell. Unlike traditional shells, I introduced a novel concept allowing users to define custom functions in Rust and register them as programs at compile-time. This unique approach, unseen in mainstream OSs, empowered users to unleash their creativity without the need for complex disk drivers, file systems, or syscall interfaces. The user shell relied on a HashMap to store program names and corresponding function pointers, ensuring uniformity in function identity. To standardize data exchange between programs, I followed the Unix philosophy, treating text as the universal data type. Macros like parse! facilitated parsing text into desired data types. Additionally, I introduced POSIX-like pipes, enabling command chaining by passing output as input to subsequent commands. The user API offered access to core OS functions, heap-allocated types, assembly, and screen printing, simplifying the process of creating custom program functions. I provided examples and recommended defining user programs in the usr_bin directory for clarity[59].

[58] MaderNoob, 2022
[59] Opengroup.org, 2018

During a bug-fix phase, I made numerous minor improvements, addressing issues, enhancing resource management, and boosting speed. Code documentation and organization received attention, and I leveraged macros to reduce code repetition. For instance, the kinit (kernel init) macro simplified function initialization, reducing the number of lines and enhancing code maintainability.

In summary, my methodology for implementing various components of the operating system involved a comprehensive study of existing resources while actively making significant changes and improvements to adapt them to my project's needs. These enhancements, along with innovative additions like the user shell, demonstrate my commitment to surpassing expectations and contributing unique features to the OS development landscape.

3 Challenges and problems

Throughout the course of this project, I grappled with a multitude of challenges that shed light on the disparities between the pursuit of knowledge and its practical application. One of the initial hardships I faced was the scarcity of comprehensive information. In my prior experience, I had grown accustomed to having a multitude of sources at my disposal for research and troubleshooting. However, certain aspects of the project, such as the intricate realm of SSD detection, lacked the documentation and information I had come to rely on. This information gap presented a significant roadblock, necessitating extensive efforts to acquire the necessary insights.

Moreover, I grappled with the challenge of comprehending theoretical concepts while being without the practical proficiency necessary for their effective implementation. The educational curriculum had primarily emphasized Python, a language ill-suited for the intricacies of OS development, leaving me with a significant deficit in the fundamental OS concepts and skills. Consequently, I embarked on a steep learning curve, compelling me to extensively research and consult an array of references to independently acquire the requisite knowledge and skills. For instance, I undertook the formidable task of grasping the Assembly language, renowned as one of the most challenging programming languages to master[60]. I pushed my boundaries to achieve a level of proficiency that enabled me to both understand and write Assembly code, a vital skill for various components of the operating system

The complexity of certain aspects of the project posed another significant hurdle. Some areas of information proved to be so intricate that they demanded a profound knowledge base for meaningful comprehension. Confronted with these intricate concepts, I embarked on an extensive learning journey to bridge the knowledge gap and make these complex concepts more accessible.

Additionally, the transition from a 32-bit to a 64-bit operating system architecture presented its own set of challenges. It required a careful evaluation of which concepts from the former could seamlessly integrate into the latter, ensuring compatibility and optimal performance of the 64-bit OS.

[60]devmio - Software Know-How, 2018

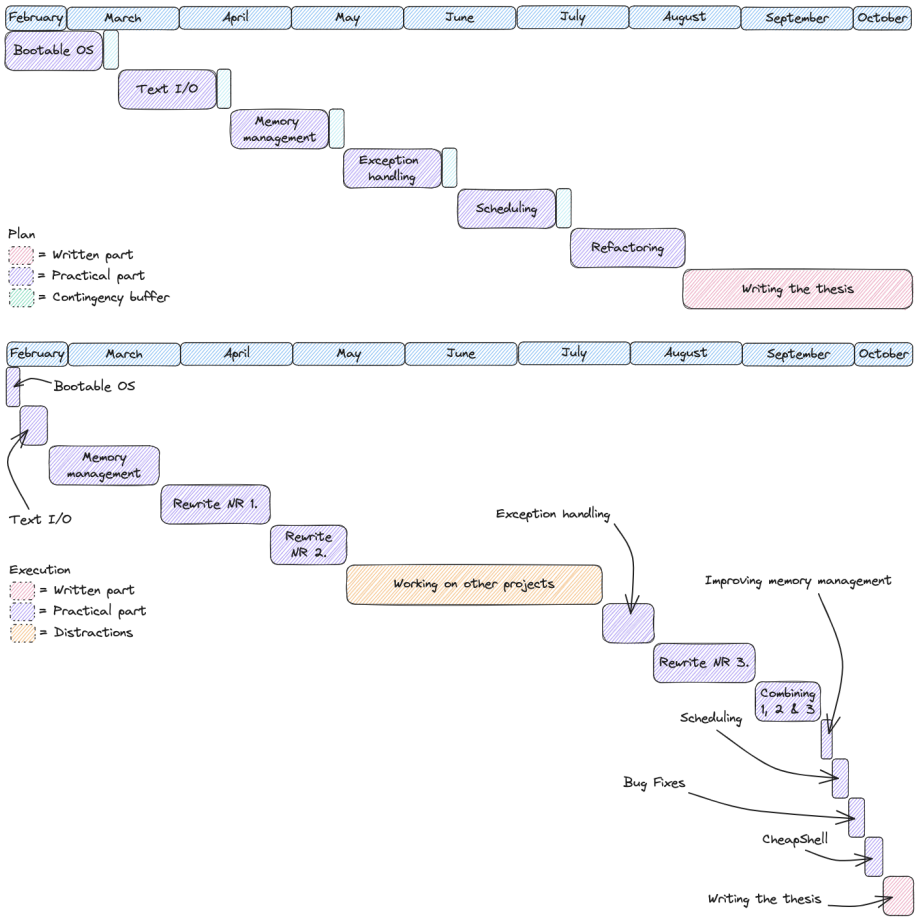

Figure 10: Time (mis)management

On the project planning front, my initial optimism, marked by evenly distributed sub-goals within the project timeline, collided with the unpredictability of reality. The actual project timeline revealed moments of intense productivity interspersed with prolonged pauses and diversions, resulting in periods of stress and inefficiency. This discrepancy between the envisioned plan and the actual progression underscored the need for more flexible and realistic project planning.

Compatibility issues further complicated the project. My initial guidance from a blog post detailing OS creation in Rust aimed to lay a solid foundation. However, as my project scope expanded to include insights from osdev articles, ensuring compatibility with the existing codebase presented unfore-

seen challenges. At one point, I had to discard an entire month's worth of code due to its incompatibility, emphasizing the necessity for thorough evaluation before making substantial changes.

Another persistent issue revolved around the desire to implement new features while existing components remained incomplete. The allure of exploring uncharted territories often overshadowed the essential need to solidify the project's foundation. This challenge highlighted the importance of striking a balance between progress and comprehensive refinement of existing elements.

In conclusion, the diverse challenges I encountered not only tested my problem-solving skills but also underscored the intricate nature of OS development. The scarcity of readily available information, the disconnection between theoretical knowledge and practical implementation, the complexity of certain concepts, the transition to a 64-bit architecture, the need for flexible project planning, compatibility issues, and the struggle to balance new features with foundational refinement all served as valuable lessons. These experiences continue to inform and shape the ongoing development of the project, reflecting the adaptability, in-depth research, and balanced approach between theory and practice essential for successful OS development.

4 Reflecting on the project

4.1 What could I have done better?

Reflecting on the OS development project, there are notable areas in which improvements could have been made in terms of the overall efficiency and effectiveness of the development process. The project exhibited characteristics reminiscent of the "Dizz Tracked[61]" and "Cowboy Coder[62]" styles described in the OS dev wiki, which, while having some advantages, also presented challenges.

One aspect that could have been managed more effectively was the tendency to become "Dizz Tracked[61]." I occasionally found myself getting sidetracked into exploring new compilers, linkers, or unrelated coding endeavors. While this diversification can fend off boredom, it also has the potential to divert valuable time and resources away from the primary project goal. A more focused and disciplined approach, emphasizing the core project objectives, could have mitigated these distractions and increased productivity.

My inclination towards the "Cowboy Coder[62]" style was also evident, especially during the initial stages of the project. To jump head first into coding without thorough planning or a deep understanding of the underlying theory led to certain challenges. The development process was occasionally marked by a lack of strategic foresight, which resulted in multiple rewriting of the kernel in search of a "better" solution. This lack of a coherent plan and strategy occasionally hindered progress and led to incomplete or abandoned projects.

Furthermore, the fluctuating motivation and shifting focus of the "Cowboy Coder[62]" style sometimes disrupted the project's continuity. It was not uncommon for the project's direction to suddenly veer towards unrelated areas, like transitioning from bootloader development to GUI design. The lack of a clear, long-term vision often resulted in a fragmented approach to problem-solving, focusing on trivial issues without a comprehensive understanding of the overarching goals.

[61] Osdev.org, 2022a
[62] Osdev.org, 2020

Moreover, the tendency to dive into coding without fully grasping the big picture often resulted in forum queries that lacked context and clarity. These inquiries may have posed difficulties for both the developer that sought assistance and the individuals trying to provide guidance.

To enhance the project's efficiency and cohesiveness, a more structured approach to project planning and goal setting, with a clear vision of the project's overall scope and requirements, could have been beneficial. This approach would have helped to reduce the distractions stemming from side projects and provide a strong foundation for consistent progress. Additionally, a well-defined development strategy that considers the bigger picture and problem-solving at a higher level would mitigate the need for frequent code rewrites and enhance the project's long-term sustainability.

In conclusion, even though the "Dizz Tracked[63]" and "Cowboy Coder[64]" styles offer certain advantages, such as new ideas and interesting insights they also present challenges in terms of efficiency, focus, and continuity. By incorporating a more structured and strategic approach, future OS development projects can be developed more cohesively and productively, which would ultimately result in a more successful and fulfilling outcome.

4.2 Prospects for the future

With innovative developments on the horizon, the future of operating systems holds promising prospects. One exciting avenue is the enhancement of system libraries, such as the C library (libc), to offer greater functionality and compatibility. Implementing advanced networking capabilities will enable seamless communication in an increasingly connected world. The integration of device drivers for the detection, reading, and writing to external drives ensures comprehensive hardware support. Furthermore, the adoption of will bolster data storage and management, promoting efficiency and reliability. Changing the bootloader will enable more fine grain control. These future endeavors signify a continued evolution in operating systems, aligning them with the ever-expanding demands of modern computing.

[63]Osdev.org, 2022a
[64]Osdev.org, 2020

Adding a libc

One of the primary considerations is the implementation of an application layer and a shell. This step would significantly increase the usability of the operating system as it allows users to run their programs and interact with the system more intuitively. To achieve this, porting a C-standard library and cross-compiling the GNU binutils are crucial. Numerous options for portable C standard libraries are available, and each has its advantages and limitations, as shown by Osdev:

Glibc

- GPL license
- Should be absolutely complete (even has all the bloat)
- Supports almost every architecture
- Generates large programs
- Is not written with anything other than Linux in mind, making it a hard port.

Musl

- MIT license
- No kernel portability layer, uses the Linux system calls directly. You can add your own layer between musl and the kernel to translate Linux system calls into native system calls, which is the method used by midipix
- A full set of math and printf functions
- Support for about 1200 functions
- Many system calls needs to be implemented as it assumes you are a full Linux

NewLib

- The license is unrestricted (not GPL or LGPL), but each file likely has a different copyright notice.
- Requires threading, so is more appropriate for a runtime library
- About 400 functions supported

mlibc

- MIT license
- Supports C11, POSIX and various linux and glibc extensions
- Highly modular (can turn off extensions if not wanted)
- Extremely easy to port to your OS
- Supports a large variety of ports (bash, gcc, wayland, xorg and many more)
- Supports dynamic and static linking

(Osdev.org, 2014)

I recommend Mlibc as it emerges as the most suitable choice for several compelling reasons. To begin with, its MIT license grants significant freedom without the constraints of more restrictive licenses. However, like all libcs there is a need to implement a syscall api. Its high modularity allows for customization, thus letting developers selectively enable or disable extensions based on specific project requirements. Notably, Mlibc's seamless portability to different operating systems sets it apart from other options, making it an adaptable and versatile option. Moreover, with extensive support for ports including essential components like bash, gcc, Wayland, and Xorg, Mlibc ensures compatibility with a wide range of software. Combined with support for both dynamic and static linking, this solidifies Mlibc's position as the most suitable C library.

Implementing networking

Next, implementing networking capabilities is a crucial step for expanding the project's functionality[65]. The choice between following the OS Dev wiki's networking article and utilizing the smoltcp Rust library depends on factors such as ease of their respective implementation as well as long-term maintenance considerations. The smoltcp library could be an excellent choice, as it offers support and simplification in networking development.

[65] Osdev.org, 2019b

Adding device drivers
The development of device drivers constitutes a pivotal stage in empowering the operating system to seamlessly identify and establish efficient communication with external hardware components. Notably, this task is indispensable since the current state of the OS lacks the mechanisms required for detecting, reading from, and writing to external drives, which are essential for interfacing with physical file systems. By undertaking this essential endeavor, the OS can significantly augment its functionality, thereby enhancing its adaptability and versatility in interactions with a diverse array of external devices. This augmentation is a pivotal stride towards enabling the OS to reach its full potential in bridging the gap between software and hardware, ultimately facilitating a more comprehensive and effective computing experience[66].

Adding a file system
Opting for VFS as the initial file system is a prudent decision due to its simplicity and straightforward implementation. Its uncomplicated design streamlines the development process, and thus makes it an ideal choice for a foundational file system. Moreover, VFS serves as a versatile bridge, enabling seamless interaction with files from various file system formats. This adaptability enhances the system's capabilities and sets a strong foundation for future expansions and enhancements in the realm of file management[67].

Switching the bootloader
The plan to transition from the existing bootloader, the bootloader crate to alternatives such as Grub2 or Limine holds significant merit. The decision to change bootloaders revolves around the desire to exercise greater control over critical aspects, particularly memory management. In the conventional system, memory management is primarily handled by the bootloader, or it is embedded within the boot assembly stub. By opting for a different bootloader, the project gains a heightened degree of control over the implementation of memory management, allowing for fine-grained customization.

Furthermore, an essential consideration in favor of changing bootloaders is the need to create an ISO image. The current bootloader crate solely gener-

[66]Osdev.org, 2023a
[67]Tldp.org, 2023

ates .bin images. Whereas these are suitable for booting purposes, they may appear unconventional and perplexing to users. Shifting to a real bootloader like Grub2 or Limine paves the way for the development of a small program that facilitates the transfer of the OS from a bootable storage device (e.g., USB) to a hard disk. A genuine bootloader provides enhanced control over memory addresses, simplifying the determination of the OS size and starting address for efficient copying—a practice commonly employed by major operating systems.

Grub2 presents an appealing choice, as it offers support for multiboot and multiboot2, along with flexible assembly boot stub requirements. It also permits the integration of modules, facilitates UEFI and BIOS boot options, and boasts compatibility with a wide array of file system formats. Grub2's boot process commences in 16 Real mode assembly, necessitating the subsequent switch to 32-bit and then 64-bit modes[68].

On the other hand, Limine caters to the needs of hobby OS developers, streamlining much of the intricate work and reducing the assembly prerequisites. It also allows for modules and supports various boot methods, including UEFI and BIOS[69]. Notably, Limine differentiates itself by adhering to a minimalistic philosophy, avoiding unnecessary bloat:

> The idea with Limine is to remove the responsibility of parsing filesystems and formats, aside from the bare minimum necessities (eg: FAT*, ISO9660), from the bootloader itself. It is a needless duplication of efforts to have bootloaders support all possible filesystems and formats, and it leads to massive, bloated bootloaders as a result (eg: GRUB2). (Limine-bootloader, 2019a)

Additionally, Limine offers a direct path to a 64-bit setup with higher-half kernel support, simplifying kernel development and deployment.

I would recommend Limine over Grub2 due to its minimalistic design and focus on simplicity. Limine is purpose-built for hobby OS developers, streamlining the process and reducing the need for extensive assembly programming. Its lean and efficient nature aligns with the needs of smaller-scale OS projects

[68] Dubbs, 2018
[69] Limine-bootloader, 2023

like ChadOS, providing essential features without unnecessary bloat. Limine offers a straightforward and uncluttered approach, making it a preferred choice. This minimalistic approach enhances clarity and control.

5 Conclusion

In conclusion, the journey to creating an operating system from scratch has been a challenging yet rewarding endeavor. Key decisions were made with regard to the choice of programming language, tooling, and the development environment to ensure the efficiency, reliability, and maintainability of the project. Rust emerged as the primary programming language due to its focus on memory safety without sacrificing execution speed, a crucial aspect in operating system development. Its lack of overhead associated with a garbage collector and high-level abstractions with low-level control made it a well-suited choice. My proficiency in Rust further expedited the development process.

The choice of tooling and environment was just as critical. Rust's ecosystem combined with the power of Cargo, its package manager and build tool, simplified dependency management and streamlined the development process. Cargo Make, a versatile build tool, was selected to manage the build process, as it provides flexibility and automation while maintaining project organization. Nix played a crucial role in maintaining a consistent and reproducible development environment, ensuring that the development setup could be easily replicated on different platforms.

The text editor of choice, Neovim, provided productivity and extensibility, making it an ideal companion for the project. Its minimalistic design and low resource usage made it a compelling alternative to more resource-intensive editors.

The realization of project objectives showcased the importance of memory management, error handling, and scheduling in the development of a functional operating system. These achievements aligned the project with broader software goals and highlighted its commitment to reliability and multitasking.

I want to emphasize that this project has been significantly more challenging compared to other A-level thesis projects I've encountered. With a codebase comprising 4.3k lines, it stands as one of the most demanding and intricate theses. However, I can affirm that this experience has been incredibly

enlightening. The journey was marked by formidable challenges, including a scarcity of accessible information, bridging the gap between theory and practical implementation, grappling with complexity in specific areas, and managing compatibility issues when expanding the project's scope. These challenges underscored the paramount importance of adaptability, rigorous research, and a harmonious integration of theory and practice in the realm of OS development.

With regard to the future, the project has promising prospects. Enhancing system libraries such as the libc, implementing networking capabilities, adding device drivers, and incorporating a file system will expand the project's functionality and usability. Changing the bootloader to a minimalistic one like Limine, will offer greater control and simplicity in memory management and deployment. These future endeavors align with the ever-expanding demands of modern computing and signify a continued evolution in operating systems.

Together with the realization of project objectives and the anticipation of future developments, the choice of Rust as the programming language and the chosen tooling and environment, have set a strong foundation for the ongoing development of this operating system project. The challenges encountered have provided valuable lessons and insights, shaping the project's future direction.

6 Appendix

Project integrity
To ensure the integrity of the source code and to demonstrate that it has not been altered after the project deadline, a zip archive of the codebase was created. The zip archive was then hashed using the SHA-256 hash algorithm, resulting in a unique and fixed hash value. Any changes made to the source code would result in a different hash value. This approach serves as a tamper-evident measure, providing evidence that the code remains unchanged beyond the specified project deadline. To check the hash of the zip file, enter '`sha256sum ChadOS.zip`' in a UNIX terminal.

```
SHA-256 HASH: 00bb26a64b047ae48206ef94b0130322af8cda804678ec05ff7159d04beff1e3
```

Acknowledgements
I would like to extend my sincere gratitude to Corinna Geng for her invaluable proofreading assistance, ensuring the quality and clarity of my work. Special thanks to Phil Opp for his excellent blog on writing an OS in Rust, which laid a strong foundation for my project. Additionally, I want to express my heartfelt appreciation to Peter Skrotzky for his constant support and the many scientific discussions that have enriched this work. I would also like to extend my warm thanks to Alan Metzler for agreeing to be my counter-reader. As my long-time English teacher, he has been a motivating influence in my use of English in everyday life and in this work. Lastly, I am immensely thankful to the OSDev Wiki, which provided an extensive wealth of information on a wide array of pertinent subjects, guiding me throughout the OS development journey. The contributions and support of everyone named have been instrumental in the project's success, and I deeply appreciate their guidance.

7 References

- Advanced Micro Devices Inc. (2013). AMD64 Architecture Programmer's Manual Volume 4: 128-Bit and 256-Bit Media Instructions. [online] Available at: https://bluewaters.ncsa.illinois.edu/liferay-content/document-library/amd_4_26568.pdf [Accessed 2. Nov. 2023]

- Akhtar, H. (2023). Introduction to Code Based Testing and its Importance — BrowserStack. [online] BrowserStack. Available at: https://www.browserstack.com/guide/code-based-testing [Accessed 5 Nov. 2023].

- Aosabook.org. (2023). The Architecture of Open Source Applications (Volume 1)The Bourne-Again Shell. [online] Available at: https://aosabook.org/en/v1/bash.html#:~:text=Bash%20processing%20is%20much%20like,and%20collects%20its%20return%20status. [Accessed 5 Nov. 2023].

- Apple Developer Documentation. (2023). About Apple File System — Apple Developer Documentation. [online] Available at: https://developer.apple.com/documentation/foundation/file_system/about_apple_file_system [Accessed 12 Oct. 2023].

- Apple File System Reference Developer Contents. (n.d.). Available at: https://developer.apple.com/support/downloads/Apple-File-System-Reference.pdf [Accessed 12 Oct. 2023].

- Archlinux.org. (2022a). Ext4 - ArchWiki. [online] Available at: https://wiki.archlinux.org/title/ext4 [Accessed 12 Oct. 2023].

- Archlinux.org. (2022b). Limine - ArchWiki. [online] Available at: https://wiki.archlinux.org/title/Limine [Accessed 12 Oct. 2023].

- Archlinux.org. (2022c). Limine - ArchWiki. [online] Available at: https://wiki.archlinux.org/title/Limine [Accessed 12 Oct. 2023].

- Bigelow, S.J. and Lulka, J. (2022). kernel. [online] Data Center. Available at: https://www.techtarget.com/searchdatacenter/definition/kernel [Accessed 2 Nov. 2023].

- Bran. (2023). Bran's Kernel Development Tutorial on Bona Fide OS Developer. [online] Available at: http://www.osdever.net/tutorials/view/brans-kernel-development-tutorial [Accessed 5 Nov. 2023].
- C-Bit.org. (2020). Limitations of the FAT32 File System in Windows XP - 31.07.2020 10:19:21 (CEST). [online] Available at: https://web.archive.org/web/20200731081921/http://c-bit.org/kb/314463/EN-US/ [Accessed 11 Oct. 2023].
- Cisco. (2023). Duo Multi-Factor Authentication (MFA). [online] Available at: https://www.cisco.com/c/en/us/products/security/what-is-multi-factor-authentication.html [Accessed 3 Nov. 2023].
- devmio - Software Know-How. (2018). POLL RESULTS: Assembly is officially crowned the language with the steepest learning curve. [online] Available at: https://devm.io/programming/most-difficult-programming-languages-152590 [Accessed 5 Nov. 2023].
- Die.net. (2023). ustar(1): unique standard tape archiver - Linux man page. [online] Available at: https://linux.die.net/man/1/ustar [Accessed 12 Oct. 2023].
- Dubbs, B. (2018). GNU GRUB - GNU Project - Free Software Foundation (FSF). [online] Gnu.org. Available at: https://www.gnu.org/software/grub/ [Accessed 5 Nov. 2023].
- GeeksforGeeks. (2018a). Functions of Operating System. [online] Available at: https://www.geeksforgeeks.org/functions-of-operating-system/ [Accessed 2 Nov. 2023].
- GeeksforGeeks. (2018b). LIFO Last In First Out approach in Programming. [online] Available at: https://www.geeksforgeeks.org/lifo-last-in-first-out-approach-in-programming/ [Accessed 2 Nov. 2023].
- GeeksforGeeks. (2019). Multilevel Paging in Operating System. [online] Available at: https://www.geeksforgeeks.org/multilevel-paging-in-operating-system/ [Accessed 13 Oct. 2023].
- GeeksforGeeks. (2020). Protection Ring. [online] Available at: https://www.geeksforgeeks.org/protection-ring/ [Accessed 12 Oct. 2023].

- GeeksforGeeks. (2021). Memory Management in Operating System. [online] Available at: https://www.geeksforgeeks.org/memory-management-in-operating-system/ [Accessed 2 Nov. 2023].
- GeeksforGeeks. (2023). Stack Vs Heap Data Structure. [online] Available at: https://www.geeksforgeeks.org/stack-vs-heap-data-structure/ [Accessed 2 Nov. 2023].
- Gnu.org. (2022). Binutils - GNU Project - Free Software Foundation. [online] Available at: https://www.gnu.org/software/binutils/ [Accessed 11 Oct. 2023].
- Gnu.org. (2023). GNU GRUB Manual 2.06: Features. [online] Available at: https://www.gnu.org/software/grub/manual/grub/html_node/Features.html [Accessed 12 Oct. 2023].
- Hutchinson, L. (2016). Digging into the dev documentation for APFS, Apple's new file system. [online] Ars Technica. Available at: https://arstechnica.com/gadgets/2016/06/digging-into-the-dev-documentation-for-apfs-apples-new-file-system/ [Accessed 11 Oct. 2023].
- Ibm.com. (2023). IBM Z Multi-Factor Authentication — IBM. [online] Available at: https://www.ibm.com/products/ibm-multifactor-authentication-for-zos [Accessed 3 Nov. 2023].
- Jason Long. (2015). File:Neovim-mark.svg - Wikipedia. [online] Available at: https://en.m.wikipedia.org/wiki/File:Neovim-mark.svg [Accessed 2 Nov. 2023].
- JasonGerend (2023). NTFS overview. [online] Microsoft.com. Available at: https://learn.microsoft.com/en-us/windows-server/storage/file-server/ntfs-overview [Accessed 12 Oct. 2023].
- javatpoint.com. (2021). Device Management in Operating System - javatpoint. [online] Available at: https://www.javatpoint.com/device-management-in-operating-system [Accessed 2 Nov. 2023].
- Jhawthorn. (2006). File:Virtual memory.png - OSDev Wiki. [online] Available at: https://wiki.osdev.org/File:Virtual_memory.png [Accessed 12 Oct. 2023].

- Jmu.edu. (2021). 2.4. System Call Interface — Computer Systems Fundamentals. [online] Available at: https://w3.cs.jmu.edu/kirkpams/OpenCSF/Books/csf/html/Syscall.html [Accessed 3 Nov. 2023].
- Keerthi Chinthaguntla (2020). Setting up multi-factor authentication on Linux systems. [online] Enable Sysadmin. Available at: https://www.redhat.com/sysadmin/mfa-linux [Accessed 3 Nov. 2023].
- Kernel.org. (2023a). BTRFS — The Linux Kernel documentation. [online] Available at: https://docs.kernel.org/filesystems/btrfs.html. [Accessed 11 Oct. 2023].
- Kernel.org. (2023b). Message logging with printk — The Linux Kernel documentation. [online] Available at: https://www.kernel.org/doc/html/next/core-api/printk-basics.html [Accessed 5 Nov. 2023].
- Kernel.org. (2023c). Page Table Management. [online] Available at: https://www.kernel.org/doc/gorman/html/understand/understand006.html [Accessed 13 Oct. 2023].
- Kernelnewbies.org. (2017). Ext4 - Linux Kernel Newbies. [online] Available at: https://kernelnewbies.org/Ext4 [Accessed 12 Oct. 2023].
- Limine-bootloader. (2019a). PHILOSOPHY.md at v5.x-branch · limine-bootloader/limine. [online] GitHub. Available at: https://github.com/limine-bootloader/limine/blob/v5.x-branch/PHILOSOPHY.md [Accessed 12 Oct. 2023].
- Limine-bootloader. (2019b). PROTOCOL.md at v5.x-branch · limine-bootloader/limine. [online] GitHub. Available at: https://github.com/limine-bootloader/limine/blob/v5.x-branch/PROTOCOL.md [Accessed 12 Oct. 2023].
- Limine-bootloader. (2023). Limine. [online] Available at: https://limine-bootloader.org/ [Accessed 12 Oct. 2023].
- MaderNoob. (2022). good_memory_allocator — Rust memory management library // Lib.rs. [online] Available at: https://lib.rs/crates/good_memory_allocator [Accessed 5 Nov. 2023].

- Microsoft Research. (2023). Singularity - Microsoft Research. [online] Available at: https://www.microsoft.com/en-us/research/project/singularity/ [Accessed 11 Oct. 2023].
- Mit.edu. (2023). Design and Implementation of the Second Extended Filesystem. [online] Available at: http://web.mit.edu/tytso/www/linux/ext2intro.html [Accessed 11 Oct. 2023].
- Neovim.io. (2023). Home - Neovim. [online] Available at: https://neovim.io/ [Accessed 3 Nov. 2023].
- Nixos.org. (2023). Nix & NixOS — Reproducible builds and deployments. [online] Available at: https://nixos.org/ [Accessed 3 Nov. 2023].
- Opengroup.org. (2018). Shell Command Language. [online] Available at: https://pubs.opengroup.org/onlinepubs/9699919799/utilities/V3_chap02.html [Accessed 5 Nov. 2023].
- Opengroup.org. (2019). 9. Application Layer : ArchiMate 3.1 Specification. [online] Available at: https://pubs.opengroup.org/architecture/archimate31-doc/chap09.html [Accessed 3 Nov. 2023].
- Oppermann, P. (2015a). A minimal Multiboot Kernel — Writing an OS in Rust (First Edition). [online] Phil-opp.com. Available at: https://os.phil-opp.com/multiboot-kernel/ [Accessed 2 Nov. 2023].
- Oppermann, P. (2015b). Page Tables — Writing an OS in Rust (First Edition). [online] Phil-opp.com. Available at: https://os.phil-opp.com/page-tables/ [Accessed 13 Oct. 2023].
- Oppermann, P. (2018a). CPU Exceptions — Writing an OS in Rust. [online] Phil-opp.com. Available at: https://os.phil-opp.com/cpu-exceptions/ [Accessed 3 Nov. 2023].
- Oppermann, P. (2018b). Double Faults — Writing an OS in Rust. [online] Phil-opp.com. Available at: https://os.phil-opp.com/double-fault-exceptions/ [Accessed 3 Nov. 2023].
- Oppermann, P. (2018c). Hardware Interrupts — Writing an OS in Rust. [online] Phil-opp.com. Available at: https://os.phil-opp.com/hardware-interrupts/ [Accessed 3 Nov. 2023].

- Oppermann, P. (2019a). Heap Allocation — Writing an OS in Rust. [online] Phil-opp.com. Available at: https://os.phil-opp.com/heap-allocation/ [Accessed 2 Nov. 2023].
- Oppermann, P. (2019b). Introduction to Paging — Writing an OS in Rust. [online] Phil-opp.com. Available at: https://os.phil-opp.com/paging-introduction/ [Accessed 3 Nov. 2023].
- Oppermann, P. (2022). Writing an OS in Rust. [online] Phil-opp.com. Available at: https://os.phil-opp.com/ [Accessed 5 Nov. 2023].
- Osdev.org. (2014). GRUB - OSDev Wiki. [online] Available at: https://wiki.osdev.org/GRUB [Accessed 12 Oct. 2023].
- Osdev.org. (2019a). BIOS - OSDev Wiki. [online] Available at: https://wiki.osdev.org/BIOS [Accessed 2 Nov. 2023].
- Osdev.org. (2019b). Network Stack - OSDev Wiki. [online] Available at: https://wiki.osdev.org/Network_Stack [Accessed 12 Oct. 2023].
- Osdev.org. (2020). Dizz Tracked - OSDev Wiki. [online] Available at: https://wiki.osdev.org/Dizz_Tracked [Accessed 13 Oct. 2023].
- Osdev.org. (2022a). Cowboy Coder - OSDev Wiki. [online] Available at: https://wiki.osdev.org/Cowboy_Coder [Accessed 13 Oct. 2023].
- Osdev.org. (2022b). Ext2 - OSDev Wiki. [online] Available at: https://wiki.osdev.org/Ext2 [Accessed 11 Oct. 2023].
- Osdev.org. (2023a). Device Management - OSDev Wiki. [online] Available at: https://wiki.osdev.org/Device_Management [Accessed 5 Nov. 2023].
- Osdev.org. (2023b). File Systems - OSDev Wiki. [online] Available at: https://wiki.osdev.org/File_Systems [Accessed 2 Nov. 2023].
- Osdev.org. (2023c). Memory management - OSDev Wiki. [online] Available at: https://wiki.osdev.org/Memory_management [Accessed 5 Nov. 2023].

- Osdev.org. (2023d). NTFS - OSDev Wiki. [online] Available at: https://wiki.osdev.org/NTFS [Accessed 12 Oct. 2023].
- Osdev.org. (2023e). USTAR - OSDev Wiki. [online] Available at: https://wiki.osdev.org/USTAR [Accessed 12 Oct. 2023].
- Osdev.org. (2023f). What Order Should I Make Things In? - OSDev Wiki. [online] Available at: https://wiki.osdev.org/What_Order_Should_I_Make_Things_In [Accessed 13 Oct. 2023].
- Pearsonitcertification.com. (2013). Security Architecture — CISSP Exam Cram: Security Architecture and Models — Pearson IT Certification. [online] Available at: https://www.pearsonitcertification.com/articles/article.aspx?p=1998558&seqNum=3 [Accessed 2 Nov. 2023].
- RealWorldCyberSecurity (2020). Negative Rings in Intel Architecture: The Security Threats That You've Probably Never Heard Of. [online] Medium. Available at: https://medium.com/swlh/negative-rings-in-intel-architecture-the-security-threats-youve-probably-never-heard-of-d725a4b6f831 [Accessed 12 Oct. 2023].
- Reddit.com. (2023). How does APFS compare to NTFS?. [online] Available at: https://www.reddit.com/r/osx/comments/6gkofg/how_does_apfs_compare_to_ntfs/ [Accessed 11 Oct. 2023].
- Remzi H. (2018). Paging: Introduction. [online] Available at: https://pages.cs.wisc.edu/~remzi/OSTEP/vm-paging.pdf [Accessed 13 Oct. 2023].
- Rust-lang.org. (2018). Available at: https://www.rust-lang.org/ [Accessed 3 Nov. 2023].
- Rust-lang.org. (2021). Hello, Cargo! - The Rust Programming Language. [online] Available at: https://doc.rust-lang.org/book/ch01-03-hello-cargo.html [Accessed 3 Nov. 2023].
- Rust-lang.org. (2023a). Inline assembly - The Rust Reference. [online] Available at: https://doc.rust-lang.org/reference/inline-assembly.html [Accessed 3 Nov. 2023].

- Rust-lang.org. (2023b). Rust Foundation. [online] Available at: https://foundation.rust-lang.org/policies/logo-policy-and-media-guide/ [Accessed 11 Oct. 2023].
- sagiegurari. (2017). cargo-make 0.3.35 - Docs.rs. [online] Available at: https://docs.rs/crate/cargo-make/0.3.35 [Accessed 3 Nov. 2023].
- smoltcp-rs. (2023). smoltcp - Rust. [online] Available at: https://docs.rs/smoltcp/latest/smoltcp/ [Accessed 12 Oct. 2023].
- Sudo. (2014). Sudoers Manual. [online] Available at: https://www.sudo.ws/docs/man/sudoers.man/ [Accessed 3 Nov. 2023].
- thomasloven (2019). mittos64/doc/3_Activate_Long_Mode.md at master · thomasloven/mittos64. [online] GitHub. Available at: https://github.com/thomasloven/mittos64/blob/master/doc/3_Activato_Long_Mode.md [Accessed 2 Nov. 2023].
- Tim Cuthbertson. (2017). nixos-artwork/logo/nix-snowflake.svg at master · NixOS/nixos-artwork. [online] Available at: https://github.com/NixOS/nixos-artwork/blob/master/logo/nix-snowflake.svg [Accessed 2 Nov. 2023].
- Tldp.org. (2023). A tour of the Linux VFS. [online] Available at: https://tldp.org/LDP/khg/HyperNews/get/fs/vfstour.html [Accessed 5 Nov. 2023].
- Trick, C. (2022). How to Secure Your Operating System (OS). [online] Trentonsystems.com. Available at: https://www.trentonsystems.com/blog/secure-your-operating-system [Accessed 3 Nov. 2023].
- Tutorialspoint.com. (2023). Operating System - Process Scheduling. [online] Available at: https://www.tutorialspoint.com/operating_system/os_process_scheduling.htm [Accessed 2 Nov. 2023].
- Uic.edu. (2023). Operating Systems: I/O Systems. [online] Available at: https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/13_IOSystems.html [Accessed 5 Nov. 2023].

- Unikernel.org. (2023). Unikernels - Rethinking Cloud Infrastructure. [online] Available at: `http://unikernel.org/` [Accessed 2 Nov. 2023].
- Washington.edu. (2013). CSE 451, Introduction to Operating Systems, Spring 2013. [online] Available at: `https://courses.cs.washington.edu/courses/cse451/13sp/projects/project3.html` [Accessed 11 Oct. 2023].