UNIVERSITY OF BIRMINGHAM

The implications of an operating system written in Rust

Author:

Daniel Benton

Cyber Security MSc

2007018

Supervisor:

Ian Batten

University of Birmingham

Date: 18/09/23

Abstract

This research paper delves into the unexplored potential of employing the Rust programming language for the development and evaluation of a microkernel-based operating system, with a focus on memory safety. Traditional monolithic kernels face inherent vulnerabilities related to memory management, often resulting in system crashes and security breaches. The study seeks to mitigate such challenges by leveraging Rust's robust safety features, including zero-cost abstractions, move semantics, guaranteed memory safety, and data-race-free threads. Through the creation of a kernel, user space, and an interconnecting API, the project aims to demonstrate Rust's capability to alleviate common low-level vulnerabilities, thereby contributing to the broader discourse on system security and reliability. The research findings aim to provide a comprehensive understanding of the practical and theoretical advantages of using Rust in system-level programming, particularly in kernel development.

Contents

1 Introduction

As computer systems continue to evolve and become more complex, the need for reliable and robust software has never been more critical, this has also been of concern since the creation of complex networked systems. With an ever-increasing array of interconnected devices sharing information, the margin for error, particularly at the kernel level of operating systems, is rapidly shrinking. Central to this is the need for memory safety, a vital attribute for ensuring the reliability and security of an operating system, particularly in its kernel, the central core of an operating system that has complete control over everything in the system.

The kernel is the heart of an operating system, and its integrity directly affects the stability and safety of the whole system. Traditional monolithic kernels have been susceptible to bugs and vulnerabilities due to their large code base and intricate interdependencies, and these issues can lead to severe consequences such as system crashes, data corruption, and security breaches. Consequently, the advent of microkernel architecture, which emphasises minimalism and modularity, has presented a promising solution to these issues. Microkernels confine the essential features of the kernel to a minimal set of functionalities, thus reducing the attack surface and making it easier to manage and verify.

The Rust programming language, known for its unique approach to memory management, offers exciting possibilities in this space. Rust's "zero-cost abstractions", "move semantics", "guaranteed memory safety", and "threads without data races" promise to reduce many common programming errors, especially those related to memory safety. However, despite its increasing popularity, there is limited research on the application of Rust in creating microkernels and evaluating its impact on memory safety.

This project seeks to fill this research gap by creating a kernel, user space and an interconnecting API in Rust and comprehensively evaluating its memory safety. The project aims to leverage Rust's safety guarantees to develop a system that can potentially mitigate common low-level vulnerabilities related to memory management. Hopefully, this research will provide valuable insights into the practicality and viability of using Rust in system-level programming, specifically in the context of kernel development, and contribute to the broader discourse on improving systems security and reliability.

2 Background

2.1 The Boot proccess

The process by which an x86 system initiates its operations upon being powered on is multifaceted, involving a series of intricate steps to ensure that the hardware is ready and the correct software is executed [11]. Initially, upon power-up, the system conducts a Power-On Self-Test (POST), a self-diagnostic process that evaluates the integrity of the hardware components and confirms their operational status. Following the POST, control is transferred to the system firmware, which can be either the Basic Input/Output System (BIOS) [33] or the Unified Extensible Firmware Interface (UEFI) [55]. This firmware then seeks the bootstrap loader on the designated boot device, often representing the preliminary stage of a two-part loading sequence, as observed in many Linux systems with the use of the Grand Unified Bootloader (GRUB) [26] [12]. Once the bootstrap loader is situated in memory, it identifies and loads the operating system kernel. Upon successful kernel initialisation, the reins of the system are handed over to the operating system, marking the culmination of the booting process.

The BIOS, a longstanding firmware used in PCs, is responsible for hardware testing and the initiation of the operating system from a boot device. Traditionally, BIOS systems employ the Master Boot Record (MBR) as their boot sector format, [2] encompassing both the boot loader and the partition table. The BIOS interface, being 16-bit, has inherent limitations in terms of booting speed and hardware accessibility. Additionally, its compatibility is restricted to booting from drives of 2.2TB or less due to its reliance on MBR.

The Master Boot Record (MBR) is a critical data structure situated in the initial sector of a hard drive, [52] instrumental in booting the device and loading the operating system into memory. Formed during disk partitioning, a process that logically divides the drive into multiple units, the MBR contains the disk's partition table, pinpointing bootable (active) partitions. It further encompasses details about partition characteristics, such as type, size, and associated file systems. The MBR is tasked with initiating the computer's boot process through an executable code termed the "bootloader". Standard MBR configurations support 512-byte sectors, with a general disk size limitation of two terabytes. Vulnerabilities in the MBR, which takes charge post-BIOS, can expose systems to threats, including malware that alters boot sequences or ransomware that relocates the MBR. Thus, rigorous examination of the MBR is vital to identify and mitigate potential malicious intrusions.

Contrastingly, the UEFI emerges as a contemporary specification outlining a software interface between the operating system and platform firmware, envisioned as a replacement for the BIOS. [73] UEFI systems are characterised by their use of the GUID Partition Table (GPT), which not only accommodates larger drives but also offers a more extensive partitioning scheme. Unlike the text-based BIOS, UEFI can potentially offer a graphical environment even before the operating system is loaded, enhancing user interaction during the boot process. UEFI is inherently designed for rapid hardware initialisation, often bypassing certain tests, thereby facilitating quicker boot times. One of its paramount features is the Secure Boot mechanism, however, the level of security is questioned [7], ensuring the execution of only digitally signed, trusted software during the boot process.

In essence, while the BIOS has been required in computer systems for numerous decades, the UEFI emerges as a modern counterpart, addressing many of the inherent limitations of the BIOS. This includes offering enhanced boot times, bolstered security measures, and seamless integration with contemporary hardware and software configurations.

2.1.1 The Multiboot Standard

The Multiboot Specification, initially propagated by the Free Software Foundation, outlines a cross-platform standard for the booting process of computer systems. The Multiboot Specification emerged out of a necessity for standardisation in the early days of operating system development, when each new operating system had to contend with creating its unique bootloader. The introduction of Multiboot2 extended support for more complex booting scenarios, including advanced memory mapping and modular kernels. This standard seeks to facilitate the booting of operating systems from any form of storage media [50], removing the need for every new operating system to possess its distinct bootloader. Under the Multiboot framework, the bootloader remains responsible for loading the kernel and an initial RAM disk into memory, while the kernel becomes accountable for its initialization.

An important feature of the Multiboot standard is its ability to load multiple kernel images, allowing for more sophisticated multi-operating system scenarios and even certain configurations of kernel-level debugging. In addition, the standard provides an interface between the bootloader and the operating system, enabling the commnication of system information, such as memory maps and graphics mode settings.

From a technical standpoint, the Multiboot standard defines a series of requirements for the kernel to be compliant [72]. The kernel image must include a Multiboot header, usually at the beginning of the text segment, comprising several fields that dictate boot loader behavior. These fields can specify various boot-time parameters, such as the required and optional features, initial video mode, and header length. The standard outlines a mechanism for the bootloader to pass this information to the kernel, ensuring a seamless transition from the booting phase to the operating system's initialisation.

By adhering to the Multiboot standard, kernel developers can be abstracted away from the complexities of writing a bootloader from scratch or contending with the intricacies of platform-specific boot methods, focusing instead on the kernel's primary functionalities, while also adhering to well used standards.

2.1.2 Target Specification

In the Rust programming ecosystem, the package manager and build system, Cargo, facilitates the compilation for various target systems using the –target parameter. Each target is characterised by a 'target triple,' a descriptor that encapsulates the CPU architecture, vendor, operating system, and Application Binary Interface (ABI). While Rust offers a plethora of predefined target triples, including those tailored for Android or WebAssembly platforms, this project might necessitate specialised configurations. [17]

For this project, the inherent requirements, such as the absence of a conventional underlying OS, rendered the pre-existing target triples unsuitable. To address this, Rust's extensible design was leveraged to define a custom target, articulated through a JSON configuration [69]. This configuration, encapsulates numerous parameters, including architecture specifications, endianness, pointer width, and linker settings. Of paramount importance is the stipulation that the entry point adheres to the _start nomenclature, irrespective of the host operating system's conventions.

The capacity to define custom target configurations highlights Rust's adaptability, catering to bespoke system requirements. This flexibility, however, forces special care with configuration to ensure that the resultant binaries are suitable with the intended hardware and software environment, highlighting the implications of such customisations in system-level projects.

2.2 QEMU

QEMU is a versatile machine emulator that has the capability to mimic multiple types of CPUs, such as x86, PowerPC, ARM, and Sparc. It works on various host systems, including Linux, Windows, and Mac OS X. This tool allows for the running of a target operating system and its software within a virtual environment. A key feature of QEMU is its ability to run one operating system on top of another, making it useful for operations across different systems, debugging, and mimicking specific embedded devices. [8]

QEMU is organised into several parts, including a CPU emulator, a set of virtual devices like screens and hard drives, generic devices that bridge the gap between virtual and real hardware, machine descriptions, debugging tools, and a user interface.

QEMU's dynamic translator converts target CPU instructions to the host system's language in real-time and stores the converted code for later use. Unlike other dynamic translators that are difficult to adapt to new systems, QEMU makes this easier by using pre-made machine code fragments from the GNU C Compiler.

The dynamic translation in QEMU relies on breaking down target CPU instructions into simpler 'micro-operations.' These are small operations that are easier to read and take up less space. A tool called 'dyngen'

processes these micro-operations when compiling, creating a dynamic code generator. This generator then forms a function in real-time by combining several micro-operations. QEMU is efficient because it can set fixed parameters for these micro-operations during the compiling stage, using placeholder code adjustments to achieve this. [57]

In summary, QEMU is a complex mix of dynamic translation and emulation, providing a versatile platform for operations across different operating systems, debugging, and testing different architectures.

2.3 Operating systems and the Kernel

2.3.1 Monolithic kernels

A monolithic kernel represents a classical approach to operating system design where the entire operating system, encompassing not just the core functions but also device drivers, file system management, and system service routines, operates in a singular address space. This contrasts with microkernel designs where these functions are abstracted into separate processes running in user space. [60]

One of the inherent strengths of the monolithic design is its potential for high performance, mainly because of the direct function calls within the singular address space, as opposed to the message passing mechanisms frequently observed in microkernels. Moreover, the development can be more straightforward, given the direct access to internal structures of the OS. As well as being more comfortable with more developers. [49]

The architecture of a monolithic kernel is characterised by its cohesiveness and inter-communication. All of its modules, whether they are to do with process management, memory management, or I/O operations, reside in the kernel space, making inter-process communication more direct and, often, faster due to the absence of context switches typically required for user space communications. [28]

In the context of software development, the monolithic kernel approach poses both advantages and disadvantages. One advantage is the reduced overhead for developers, who do not need to worry about inter-process communication complexities between kernel and user space. This simplifies debugging and can speed up the development process. On the other hand, the lack of modularity makes it challenging to isolate faults during debugging, and the system may become less maintainable as it grows in complexity.

However, this design also introduces challenges. The complete integration means that a single module's failure, such as a device driver error, could compromise the stability and security of the entire system. Furthermore, updates or modifications to a particular module force the recompilation of the entire kernel, potentially introducing downtime or system reboots.

While monolithic kernels have been critiqued for their susceptibility to system-wide failures from a single module, it is important to note that modern implementations often incorporate techniques to mitigate this risk [62]. Features such as loadable kernel modules allow for greater flexibility and reduce the need for frequent system reboots, offering a middle-ground between the rigidity of traditional monolithic kernels and the modularity of microkernels.

In summary, a monolithic kernel offers a consolidated approach to operating system design, trading off modularity and fault isolation for direct communication and potentially enhanced performance. While its architecture has been the foundation of many traditional operating systems, contemporary system design often evaluates it against other paradigms, like microkernels, to determine the optimal balance between performance, stability, and maintainability. The debate between monolithic and microkernel architectures extends to various factors such as security implications, real-time capabilities, and the suitability for different types of hardware. For instance, the monolithic design's tight integration can, in some cases, offer less surface for security vulnerabilities, although this comes at the cost of lower fault isolation.

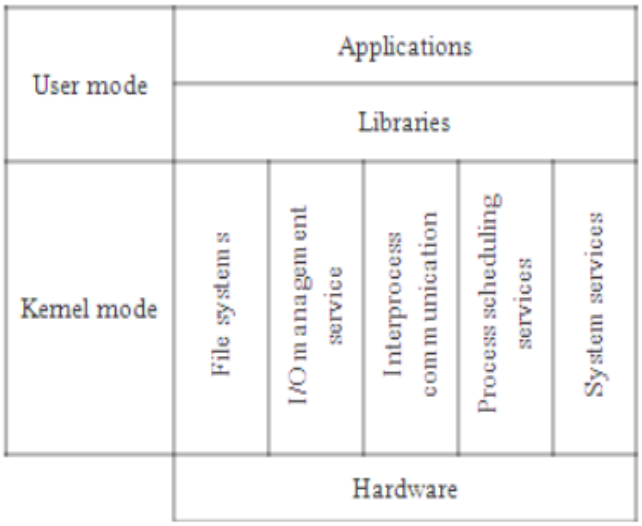


Figure 1: Structure of a Monolithic Kernel

2.3.2 Microkernels

A microkernel represents a specific approach to operating system architecture, emphasising minimalism by relegating most services, such as filesystem operations and networking, to user space as independent daemons or services. The kernel's primary responsibilities in this paradigm are limited to fundamental services, encompassing physical memory allocation, scheduling, and inter-process communication (IPC). [44] [28]

The theoretical advantages of a microkernel design are many. By placing a majority of the functionalities to preemptible user-space entities, the system potentially enhances its responsiveness. This is due to the reduced need for transitioning into the kernel, thus minimising context-switching overheads. Additionally, by minimsing the codebase in the kernel space, the system's stability is inherently bolstered, as well as ease of patching. For multi-CPU systems, microkernels offer simplifications in re-entrancy protection and are better adapted for asynchronous operations. Moreover, in distributed operating systems, the transparent utilisation of services becomes feasible, regardless of the service provider's location on the same machine or an external one. [44]

However, the microkernel approach is not without challenges. The architecture's reliance on extensive messaging and frequent context-switching can introduce performance overheads, potentially causing microkernels to be less performant than their monolithic counterparts. Furthermore, while the microkernel's resilience to individual service failures is touted, practical implications, such as data loss from a filesystem crash, remain. Addressing these challenges necessitates vast design considerations, stressing that stability and efficiency are not just byproducts of the architecture but require deliberate design efforts. In terms of software development, the microkernel architecture provides a fertile ground for modular design and parallel development efforts. Different teams can work concurrently on separate services without affecting the core kernel functionality. This division of labor can accelerate the development cycle and potentially lead to more robust system services, as each can be developed and tested in isolation. Implementations, such as AmigaOS,[5] offer insights into the potential and pitfalls of microkernels. Despite being one of the fastest due to its unique messaging system, the initial versions of AmigaOS lacked memory protection, negating the stability benefits typically associated with microkernels. Another aspect worthy of discussion is the security implications of the microkernel design. The modular nature of this architecture allows for better isolation between different system services, making it easier to apply security policies [36]. However, the increased inter-process communication could be exploited as a potential attack vector, thereby requiring additional security mechanisms to safeguard these communications.

Furthermore, the microkernel design introduces complexities in system booting processes. Given that filesystems and storage device drivers are managed outside the kernel, initial booting procedures require alternative strategies, like the bootloader loading a RAM disk image encompassing the kernel and extra support files. The adaptability of the microkernel architecture extends to its utility in embedded systems and real-time operating systems (RTOS). With minimal core functionality residing in the kernel space, microkernels can be easily tailored to meet the specific constraints of these specialized applications. However, the performance overhead due to frequent context-switching must be carefully managed to ensure real-time constraints are met.

In summary, while the microkernel paradigm offers a modular and potentially more stable architecture, its practical implementation demands strict design and consideration of performance trade-offs. Some modern

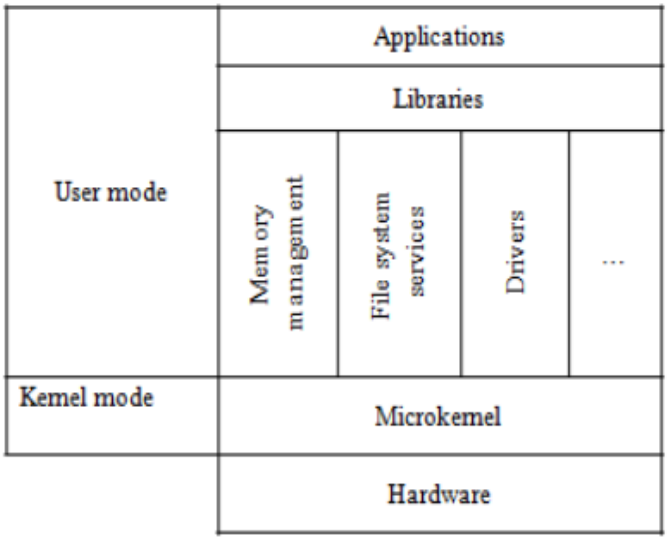


Figure 2: Structure of a Microkernel

operating systems even combine features from both microkernels and monolithic kernels, trying to find an optimal balance between modularity, performance, and stability.

2.3.3 Hybrid Kernels

A hybrid kernel represents a combination in operating system design, bringing together elements from both monolithic and microkernel architectures [25]. While a monolithic kernel consolidates all core services within the kernel space, and a microkernel delegates most to user space, the hybrid kernel integrates selected services into the kernel space for optimised performance. By doing so, it aims to harness the efficiency and direct hardware interaction of monolithic kernels while benefiting from the modularity and fault isolation characteristic of microkernels. This fusion trys to strike a balance, mitigating the individual weaknesses of both paradigms and capitalising on their respective strengths, offering an architecture that attempts to be both performant and resilient. One notable feature of hybrid kernels is their flexibility in design choices. Depending on the specific requirements of a system or application, developers can selectively move certain services between user and kernel space. This adaptability allows for a more dynamic allocation of resources, potentially leading to performance improvements without sacrificing system stability.

Another important point to consider is the increased complexity associated with hybrid kernel architectures. Unlike monolithic or microkernel designs, which adhere to a more uniform structure, hybrid kernels require meticulous planning and implementation to ensure that the benefits of both foundational architectures are successfully harnessed. This complexity extends to areas like debugging, testing, and maintenance, posing challenges that are somewhat unique to this architectural paradigm. [28]

The hybrid kernel design also impacts system security considerations. While it allows for more granular control over security policies, similar to a microkernel, it also inherits the vulnerabilities associated with integrating more services into the kernel space, akin to a monolithic kernel. Therefore, security measures in a hybrid kernel environment must be more comprehensive, accounting for both the vulnerabilities and strengths inherent to each contributing architecture.

2.3.4 Exokernels

An exokernel stands as a novel approach to operating system design, [20] emphasising the minimalism of the kernel layer by pushing most management tasks to the application or library level. Unlike traditional kernels which abstract hardware, the exokernel securely interfaces with resources, granting applications more direct access to hardware. This paradigm shift significantly affects memory management: instead of the kernel choosing memory allocation, applications have greater discretion in managing their own memory, resulting in potentially finer-grained control and optimised performance. However, this places a heightened responsibility on applications to manage memory fairly.

One distinctive feature of the exokernel design lies in its focus on enabling more efficient use of hardware resources. By reducing the layers of abstraction between applications and hardware, exokernels allow software to

make more informed decisions about resource allocation. This direct interaction often leads to improved performance and resource utilisation, as it eliminates the overhead associated with more generic resource management strategies commonly found in traditional kernel architectures.

The increased flexibility and direct hardware access come at the cost of a steeper learning curve for developers. Writing applications that interact directly with hardware requires an deep understanding of the system, making development potentially more time-consuming and error-prone. This complexity also translates into security implications, as poor application-level management of resources could lead to vulnerabilities.

Another consideration in exokernel design is its impact on portability. Traditional kernel architectures often provide a uniform interface to applications, abstracting away hardware specifics. In contrast, exokernels, by allowing direct hardware interaction, could result in applications that are more tightly coupled to specific hardware configurations. Therefore, achieving software portability becomes a more complex endeavor in an exokernel environment.

2.4 Memory management in operating systems

2.4.1 Identity Mapping

Identity mapping in memory management represents a unique intersection of simplicity and complexity, offering distinct advantages and corresponding limitations. Its core concept lies in a one-to-one correspondence between virtual and physical addresses, where each virtual address directly maps to an identical physical address. [20]

The advantages of identity mapping are multifaceted. Firstly, its simplicity is evident in the direct correspondence between the virtual and physical address spaces, which prevents the need for complex calculations or lookup tables [27]. This straightforward correlation simplifies the translation process, making it particularly valuable in environments where fast and direct access to memory is required, such as real-time systems. Secondly, this simplicity can lead to performance improvements by reducing the latency involved in memory access. The direct relationship between addresses can improve efficiency, especially in high-performance computing or systems that require rapid access to memory. Lastly, the transparency in memory layout enabled by identity mapping helps in debugging and development processes. In kernel development and low-level programming, this clarity can be important in identifying and resolving issues related to memory access and allocation [24].

However, one of the most significant challenges posed by identity mapping is its potential to create security vulnerabilities due to the direct exposure of physical memory. In this configuration, each virtual address is directly mapped to a corresponding physical address, leaving little to no room for abstraction or obfuscation. An attacker who gains unauthorised access to the virtual address space could, in theory, manipulate or extract data directly from the physical memory [27]. This poses a severe security risk, particularly in environments where data confidentiality and integrity are paramount, such as in financial institutions or healthcare systems. The direct correlation between virtual and physical addresses in identity mapping essentially opens a pathway for attackers to bypass some of the traditional layers of security mechanisms, putting the system at increased risk of exploitation.

In conclusion, identity mapping serves as an effective yet complex aspect of memory management. Its blend of simplicity, performance, and transparency must be carefully weighed against potential security risks, scalability challenges, inefficient address space utilisation, and compatibility issues with virtualisation. The decision to employ identity mapping should be predicated on a comprehensive understanding of the specific system's requirements, constraints, and goals, recognising that it may necessitate a combination of other memory management techniques to achieve an optimal balance. The exploration of identity mapping thus provides a valuable insight into the battle between efficiency and complexity in memory management.

2.4.2 Virtual Memory

Virtual memory stands as a pivotal innovation, [19] acting as a vital bridge between the physical constraints of hardware and the expansive demands of modern software. By abstracting the underlying physical memory, virtual memory creates an illusion of a larger and more accessible memory space, allowing programs to operate beyond the actual physical memory available. The concept of virtual memory is embedded in the idea of separating a program's view of memory (the virtual address space) from the actual physical storage (the physical address space). This separation allows the operating system to manage memory more flexibly, enabling features such as multitasking and memory protection.

Virtual memory operates through a combination of hardware and software mechanisms. The Memory Management Unit (MMU) plays a central role in translating virtual addresses into corresponding physical addresses [15]. This translation process is guided by data structures such as page tables or segment tables, depending on the particular memory management scheme.

Virtual memory was not always a standard feature in operating systems. Its incorporation marked a significant shift in paradigms, transitioning from constrained, single-task systems to the flexible, multitasking

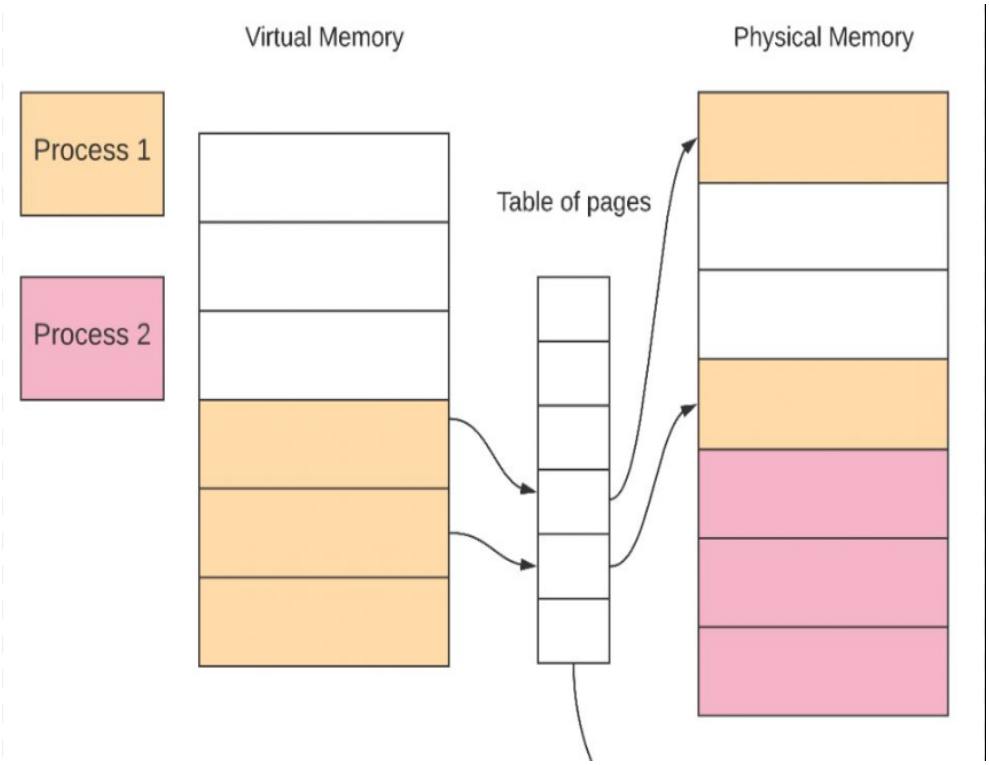


Figure 3: Virtual to physical memory

environments we're familiar with today. This development has been instrumental in the progress of complex software applications and systems.

One of the fundamental principles underlying virtual memory is paging. In a paged virtual memory system, both the virtual and physical address spaces are divided into fixed-size blocks known as pages [3]. A page table maintains the mapping between virtual pages and physical page frames, allowing the operating system to swap pages between physical memory and secondary storage, such as a hard disk. This swapping enables the execution of programs that exceed the available physical memory, effectively expanding the usable memory space.

Segmentation, another key aspect of virtual memory, provides a more granular control over memory organisation. By dividing memory into variable-sized segments, each corresponding to a logical unit of a program [3], segmentation offers a more natural alignment with the structure of high-level programming constructs.

The implementation of virtual memory brings forth several significant advantages. It facilitates multitasking by allowing multiple processes to share the physical memory [13], each operating within its isolated virtual address space. Virtual memory also enhances security by providing memory isolation, preventing one process from accessing the memory space of another. Furthermore, it simplifies programming by offering a consistent and large virtual address space, regardless of the underlying physical memory configuration.

Implementing virtual memory in a Rust-based operating system presents unique opportunities and challenges. Rust's focus on memory safety and zero-cost abstractions can simplify the complexities involved in memory management. For instance, Rust's ownership model can assist in preventing race conditions when multiple processes access the page table. However, the language's strict safety checks may pose challenges in implementing low-level features inherent to virtual memory management.

Effective management of virtual memory also necessitates the use of sophisticated page replacement algorithms. Algorithms such as First-In-First-Out (FIFO) and Least Recently Used (LRU) determine which pages are evicted from physical memory when space is needed, affecting system performance. The choice of algorithm can vary based on the specific requirements and workload characteristics of the system.

However, the translation between virtual and physical addresses introduces overhead, potentially affecting performance [24]. Effective management of virtual memory requires careful balancing of factors such as page size, swapping algorithms, and memory allocation strategies to mitigate potential inefficiencies. While virtual memory enhances security through features like memory isolation, it is not impervious to security risks. Techniques such as side-channel attacks can exploit the data remnants in shared physical memory, posing a potential risk. Security considerations are, therefore, crucial to the design and management of virtual memory systems.

In conclusion, by abstracting and extending memory, virtual memory has reshaped the way memory management systems are designed, operated, and utilised.

2.4.3 Segmentation

Memory segmentation represents a fundamental concept in the architecture of computer systems, serving as an approach to memory management that divides the virtual memory space into distinct segments. Unlike the contiguous and monolithic approach found in early computer systems, segmentation offers a more nuanced and flexible way to organize memory, allowing various parts of a program to reside in separate segments. This division facilitates both the logical organisation of data and the implementation of protection mechanisms.

In a segmented memory system, memory is divided into segments that may vary in size and function [3]. Each segment corresponds to a specific type of data or functionality, such as code, data, stack, or heap. A segment descriptor typically defines each segment, containing information about the segment's base address, length, access rights, and other attributes [45]. Segment descriptors serve as metadata structures that offer detailed information about each memory segment. These descriptors typically include the segment's base address, its length or limit, access rights, and other attributes like segment type and privilege level. The base address indicates where the segment starts in memory, while the length specifies its size. Access rights define what operations (read, write, execute) are permitted on that segment. Additionally, attributes like segment type can indicate whether the segment contains code, data, or stack information, while privilege levels can impose restrictions on which CPU privilege levels can access the segment. These descriptors are crucial for both the operating system and the hardware to manage memory segmentation accurately and securely. They allow the Memory Management Unit (MMU) to perform address translation and access checks, enabling the system to enforce memory protection and isolation policies effectively. This will be particularly important in this project, as code running an area defined by a user descriptor, will not be able to access memory marked by a kernel descriptor.

The benefits of memory segmentation are various. By separating different types of data and code into distinct segments, segmentation allows for more natural alignment with the structure of high-level programming languages. This alignment promotes a more understandble mapping between the programmer's conceptual view of data and its physical representation in memory. Furthermore, segmentation provides mechanisms for isolating different parts of a program, enhancing the system's security by preventing unauthorised access between segments. This isolation can also improve the stability of the system by containing errors within individual segments, thereby limiting their potential impact. [16]

While segmentation offers enhanced security through isolation of program segments, it is not without vulnerabilities. Incorrectly configured segment descriptors or poor protection mechanisms can lead to unauthorised memory access, posing security risks. Understanding these risks is essential for implementing robust memory protection in any system.

However, the management of variable-sized segments can lead to fragmentation, where the memory becomes divided into small, non-contiguous blocks that are difficult to utilise efficiently. This fragmentation can result in wasted memory and increased complexity in memory allocation and deallocation processes. Moreover, the overhead associated with maintaining segment descriptors and performing segment-based address translation can introduce additional latency in memory access. [19]

In a Rust-based operating system, memory segmentation can interact in interesting ways with the language's core principles, such as ownership and borrowing. Rust's strict type-checking and its emphasis on memory safety could simplify some of the complexities involved in segment management. For instance, segment descriptors could be represented as strongly-typed structs, offering additional safety guarantees. On the flip side, Rust's safety mechanisms might introduce challenges when implementing certain low-level manipulations commonly associated with segmentation.

An example of memory segmentation can be found in the Intel x86 architecture, where segmentation was used in 16-bit and 32-bit modes. In this architecture, segment registers hold the segment descriptors, and logical addresses are formed by combining a segment selector with an offset within the segment. The segment selector points to the segment descriptor, which defines the base address and attributes of the segment, while the offset specifies the location within the segment. This combination allows for a two-dimensional addressing scheme that provides both flexibility and control. Memory segmentation is not unique to Intel architectures. Other architectures like ARM or SPARC have their own approaches to segmentation or opt for alternative memory management schemes. However, the principles remain consistent: the division of memory into varying segments to facilitate data organization and protection. The choice of segmentation versus other methods often depends on specific architectural goals and constraints.

In conclusion, memory segmentation offers a flexible and logical way to organise memory, aligning closely with the structures and requirements of high-level programming languages. Its benefits in terms of security, stability, and organisation are tempered by challenges related to fragmentation and overhead. While memory segmentation remains a foundational concept, it is worth noting that its usage has diminished in modern computer architectures, especially with the advent of 64-bit computing. Many contemporary systems have transitioned to using paging or a combination of paging and segmentation, as these methods often provide more efficient and simpler solutions to memory management.

Global Descriptor Table

Address	Content
GDTR Offset + 0	Null
GDTR Offset + 8	Entry 1
GDTR Offset + 16	Entry 2
GDTR Offset + 24	Entry 3
...	...

Figure 4: The Global Descriptor Table example

2.4.4 The GDT

The Global Descriptor Table (GDT) emerges as an important mechanism for memory segmentation and task management, particularly within x86 and x86-64 architectures [59]. Functioning as a data structure, the GDT holds an array of segment descriptors, each of which holds important information about a specific memory segment, including its base address, limit, and access permissions. [53]

The creation of the GDT can be traced back to the beginning of segmented memory models, where the operating system and applications operate within distinct, non-overlapping segments of memory [66]. In such an environment, the GDT serves as a centralised data structure that allows the dynamic allocation and management of these segments. Each entry in the GDT is a segment descriptor, 8 bytes in length for x86 and 16 bytes for x86-64 architectures, containing fields that represent the properties of the segment it represents.

A segment selector, usually stored in a segment register, is used to index into the GDT. This selector contains a 13-bit index, a Table Indicator bit to specify whether the selector refers to the GDT or a Local Descriptor Table (LDT), and a 2-bit Requestor Privilege Level (RPL) that indicates the privilege level of the selector, serving as a means to enforce access control policies. By combining these elements, the segment selector enables access to the appropriate segment descriptor within the GDT, thereby facilitating the process of memory segmentation and task switching [66] .

One of the most important features of the GDT is its role in facilitating system-level operations, such as task switching and inter-process communication. Specialised segment descriptors, known as Task State Segment (TSS) descriptors, are often stored in the GDT to manage hardware-level context switches between tasks [66]. These descriptors encapsulate the state of a task, including register values and stack pointers, enabling seamless transitions between different execution contexts.

Despite its apparent utility, the mechanism introduces an additional layer of complexity in memory management, requiring careful control between the GDT and other components like the Memory Management Unit (MMU). Moreover, the overhead associated with accessing and modifying the GDT could potentially affect system performance, especially when frequent changes are required.

While the Global Descriptor Table (GDT) has been a necessity in segment-based memory management schemes, especially in x86 architectures, its role has been somewhat replaced by the incorporation of paging mechanisms. The shift toward paging as the primary mode of memory management is motivated by several factors. First and foremost, paging offers a more flexible and scalable solution for handling large and sparse address spaces. Unlike segmentation, which requires each segment to be contiguous in memory, paging allows for the non-contiguous allocation of memory, thereby mitigating issues of fragmentation. Moreover, paging mechanisms generally introduce less overhead in terms of both memory and computational resources, as they rely on fixed-size pages and simplified data structures like page tables for address translation.

However, it would be wrong to suppose that the GDT has been rendered entirely outdated. Although its role in complete memory management has weakened, the GDT continues to serve specific, albeit more specialised, functions in modern systems. For instance, it remains an essential component for certain system-level operations, such as task switching and privilege-level management. The GDT also facilitates the setting of specific segment attributes, like access permissions, that can enhance system security.

Furthermore, in modern x86-64 architectures, the GDT is often used in a minimalistic fashion during system initialisation. Even though the system may transition to a paging-based memory model for general operation, the GDT is still utilized to establish the initial operating environment. This utilisation exemplifies the subtle and complex interplay between historical architectures and modern architectures.

In summary, while paging mechanisms have largely replaced the GDT for the main task of memory management, the GDT has not been entirely rejected. Its continuing relevance in specific contexts highlights the

complex balance between legacy systems and evolving technological requirements.

2.4.5 4-level page tables

The concept of page tables emerges as a fundamental structure that provides the translation between virtual and physical memory addresses. Page tables are integral to the virtual memory system, serving as an intermediary that provides a mapping mechanism between virtual memory addresses, as used by programs and applications, and the corresponding physical addresses within the computer's main memory or RAM.[22]

The page table is typically organised in a hierarchical structure, often including multiple levels, each containing entries that point to another level of the table or directly to a physical address [24]. Each entry within the page table corresponds to a virtual page and contains information such as the physical page frame number, along with other attributes like access permissions and status bits.

The utilisation of page tables enables several crucial functions within modern computer systems [16]. Firstly, it allows for the abstraction of memory, enabling programs to function as if they have access to a continuous and large address space, irrespective of the underlying physical memory's actual configuration or availability. This abstraction enables the efficient utilisation of memory, allowing for multi-tasking and complex memory management schemes.

Secondly, page tables play a key role in memory protection. By controlling access permissions at the page level, page tables prevent unauthorised access to specific regions of memory, thereby enhancing the overall security of the system.

The hierarchical nature of page tables appears as a strong solution to the complex problem of translating virtual addresses to physical addresses. This hierarchical structure, often referred to as multi-level page tables, is predicated on the partitioning of the virtual address into several segments, each corresponding to a different level of the page table.

A hierarchical page table system generally consists of two or more levels, each representing a different portion of the virtual address. The uppermost level is known as the root, and each subsequent level represents a further subdivision of the virtual address space. This tiered approach efficiently manages large virtual address spaces, reducing memory requirements for the page tables themselves.

In a typical multi-level paging system, say a four-level one, you have different layers of tables—PML4, PDPT, PDT, and PT, each serving as a directory to the next. When translating a virtual address to a physical one [24], the system goes through each of these tables in sequence to arrive at the physical address. Each table contains entries that point to the next table, and the final table points to the actual frame in physical memory. The challenge arises when you want to modify these tables. The CPU can only translate virtual addresses to physical addresses, but these tables themselves are in physical memory.[40]

This is where the concept of recursive page tables comes is important. The idea is to configure the last entry of the level-4 page table (PML4) to point back to itself. Now, when trying to access this recursive entry, the MMU (Memory Management Unit) will walk through the page tables as usual, but when it reaches the recursive entry, it is directed back to the PML4 table. This recursive loop virtually tricks the system into treating a certain range of virtual addresses as a direct map to the table's physical addresses.

Although the MMU believes it's looking at virtual addresses, what it's actually doing is manipulating the page tables directly. This is incredibly useful for operations like page allocation or deallocation, where you need to modify the tables themselves.

Eventually, the MMU will reach an address that directly corresponds to the physical address of the PML4 table. So, if you know how to calculate the offsets, you can simply use virtual addresses to read or modify the actual page tables. This is incredibly handy because you can manipulate the page tables as if you were dealing with an array in your codes.

The process of translating a virtual address into a physical address within this hierarchical structure is sequential and systematic [31]. For example, a system that uses a two-level page table, where a virtual address is divided into three segments: the first for the root level, the second for the next level, and the third for the offset within the page.

Root Level: The first segment of the virtual address is used as an index into the root-level table, pointing to the entry that maps to the second-level table.
Second Level: The second segment of the virtual address is used as an index into the second-level table, pointing to the entry that contains the physical address of the desired page frame.
Offset: The third segment of the virtual address represents the offset within the physical page, identifying the exact location of the required data within the physical memory.

The hierarchical nature of page tables therefore enables an efficient mechanism for translating virtual addresses to physical addresses. It provides a scalable solution that can adapt to varying sizes of virtual address

spaces, allowing for the fine-grained control and abstraction that are important in modern memory management schemes. This tiered approach illustrates the balance between efficiency, flexibility, and complexity inherent in the design and operation of contemporary computer systems.

However, the process of translating virtual addresses to physical addresses through page tables can introduce overhead, as multiple table lookups may be required. To mitigate this, many systems employ additional hardware, such as Translation Lookaside Buffers (TLBs) [32], which cache recently used page table entries to expedite the translation process.

In conclusion, page tables represent a sophisticated solution to the challenges of memory management within systems. They allow for the efficient abstraction and utilisation of memory, provide robust protection mechanisms, and form the core of the virtual memory system that underpins modern operating systems. Their implementation and utilisation are emblematic of the complex interplay of efficiency, flexibility, and security.

2.4.6 Higher half kernel

A Higher Half Kernel refers to the practice of mapping the kernel's virtual address space to the higher portion of the available addressable range. [52] This means that the kernel resides in the upper half of the virtual address space, while user-space programs are allocated to the lower half. The separation between the kernel and user space is typically defined at the midpoint of the addressable range.

The use of a Higher Half Kernel offers several advantages. Primarily, it facilitates a clear separation between kernel-space and user-space memory [30], enhancing both security and stability. By confining the kernel to a distinct and protected region of the address space, the risk of user-space applications accessing or altering kernel memory is minimised. This isolation helps prevent potential system crashes or security breaches that could come from unauthorised access to kernel structures .

Furthermore, the Higher Half Kernel approach simplifies the management of virtual memory. By allocating the kernel to the higher portion of the address space, a consistent and standardised memory layout can be maintained across different processes. This aids in the implementation of features like virtual memory and multitasking, as it allows for more straightforward context switching between user processes while preserving the kernel's position in memory.

2.4.7 Local and static variables, heap allocation

Memory management is completely relied on, providing the foundational infrastructure upon which programs execute and interact. Within this framework, the concepts of local and static variables, as well as heap allocation, take on critical roles in defining a program's memory footprint and operational behavior.

Local variables are ephemeral in nature, coming into existence when their enclosing function or block is invoked and ceasing to exist upon its termination [70]. They are typically stored in the program's stack, a region of memory characterised by its Last-In-First-Out (LIFO) organisation. Each function invocation generates a new stack frame, a contiguous block of memory that houses local variables, among other data. The stack's inherent locality and deterministic allocation and deallocation patterns make it highly efficient but also limited in size and scope. Local variables are best suited for temporary storage and are naturally tied to the function's lifecycle.

Conversely, static variables are different from this, persisting throughout the program's execution. Unlike local variables, which are reinitialised upon each function call, static variables keep their state across invocations. They are generally allocated in the data segment of a program's memory, a region distinct from the stack and dedicated to long-lived variables. The data segment is initialised at program startup and remains largely immutable in size, rendering static variables suitable for maintaining state or caching values between function calls.

Heap allocation, on the other hand, introduces an entirely different aspproach to memory management. The heap is a region of memory reserved for dynamic allocation, allowing variables to be allocated and deallocated at runtime. Unlike the stack and data segment, the heap is not bound by compile-time size constraints or function lifecycles. Variables allocated on the heap, have lifetimes that must be explicitly managed by the programmer [54], generally through system calls like malloc and free in C or C++. This flexibility comes at the cost of complexity, as improper heap management can lead to issues such as memory leaks or fragmentation.

The contradiction between stack and heap allocation represents the trade-off between efficiency and flexibility. While stack allocation, with its fixed-size frames and automatic deallocation, offers a fast and straightforward method for managing memory, it lacks the adaptability and scope of heap allocation, which allows for dynamic memory resizing and long-term data storage.

Each serves a unique role, optimised for specific use-cases and performance requirements, yet all are integral to the functioning of computer systems.

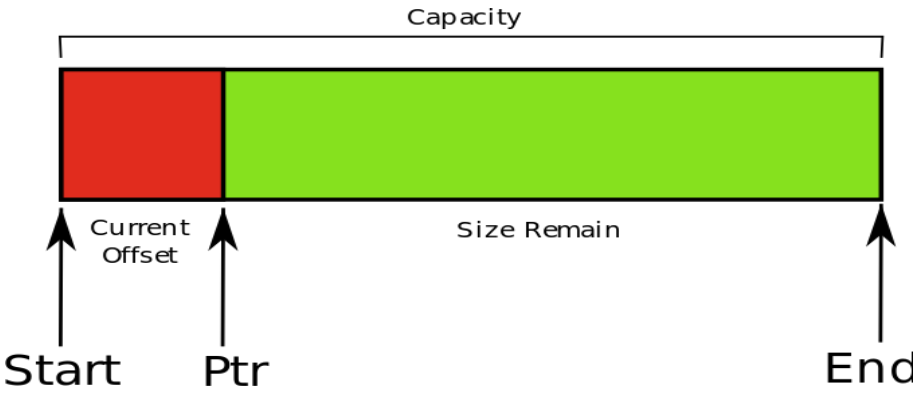


Figure 5: Bump allocator

2.4.8 Linked list, Bump, Slab and Fixed sized allocators

Among the many strategies used to manage memory allocation, linked list, bump, slab, and fixed-block allocators have pivotal roles, each offering a distinct blend of advantages and complexities.

Linked List Allocators

Linked list allocators appear as an effective technique, offering a blend of adaptability and simplicity. Rooted in the foundational data structure of linked lists, this allocator strategy provides a dynamic mechanism for managing memory allocation and deallocation, catering to the variable size and unpredictable lifetime of memory blocks in contemporary systems [24].

The core architecture of a linked list allocator consists of a linked list where each node represents a free block of memory. The node typically contains metadata, such as the size of the block and pointers to adjacent free blocks. These pointers form the linked structure, enabling the traversal and modification of the list. The overhead incurred due to the storage of this metadata is a noteworthy aspect, as it slightly reduces the usable memory space.

Within the context of linked list allocators, various allocation strategies like First-Fit, Best-Fit, and Worst-Fit can be employed. In the First-Fit approach, the allocator traverses the list and allocates the first block that satisfies the size requirement. Best-Fit, on the other hand, searches for the smallest block that meets the requirement, thereby minimizing wasted space. Worst-Fit seeks the largest available block, with the rationale that this will leave the largest remaining free block after the allocation [34]. Each strategy presents its own set of trade-offs between speed and memory utilisation efficiency.

An important challenge posed by linked list allocators is memory fragmentation [34]. Over time, the allocation and deallocation of blocks of varying sizes can lead to a fragmented memory space, where small, unusable gaps between allocated blocks prevent the efficient use of memory. Compaction techniques can mitigate this issue but often at the cost of increased computational overhead.

To further optimise memory usage, linked list allocators often implement coalescing and splitting mechanisms [43]. Coalescing involves merging adjacent free blocks into a single, larger block, thereby reducing fragmentation and making more effective use of available memory. Splitting, conversely, involces dividing a large free block into smaller portions to satisfy a specific allocation request, optimising the memory fit but potentially increasing fragmentation.

The operational complexity of linked list allocators is generally linear, O(n), due to the need to traverse the list to find suitable blocks for allocation or deallocation. However, certain optimisations, such as ordered free lists or segregated storage, can be introduced to improve performance [24].

In conclusion, linked list allocators serve as a versatile yet straightforward mechanism for dynamic memory management, aptly balancing the complexities and unpredictabilities inherent in modern computer systems. While their flexibility is a strength, it comes at the cost of potential fragmentation and computational overhead.

Bump Allocators

In stark contrast to the flexibility of linked list allocators, bump allocators display simplicity and speed. Operating on a "bump-the-pointer" principle, these allocators maintain a single pointer that indicates the start of the free memory region [**empty citation**]. Allocation is performed by simply advancing this pointer by the size of the requested block, making the allocation operation exceedingly fast. However, this speed comes at the cost of deallocation functionality; bump allocators generally do not support freeing individual blocks, leading to potential memory wastage.

Slab Allocators

Originating from the observation that many systems frequently allocate and deallocate fixed-size objects, slab allocators present a specialised methodology designed to minimise the overhead and fragmentation.

The fundamental building block of a slab allocator is the "slab," a contiguous region of memory divided into equal-sized blocks, each capable of holding one object [9]. A slab can exist in one of three states: free, in which all blocks are unallocated; partial, where some but not all blocks are allocated; and full, where all blocks are in use. Slabs are grouped into "caches," [10] [4], each dedicated to a specific object type or size, eliminating the need for complex calculations or lookup tables during the allocation process.

One of the important attributes of slab allocators is the pooling and reuse of objects. When an object is deallocated, it is not returned to the general-purpose memory pool. Instead, it remains in its slab, readily available for quick reallocation. This reuse mechanism not only accelerates the allocation process but also enhances cache locality, which can lead to improved system performance [24].

While slab allocators are optimised for object-based allocation [9], their rigid structure can lead to internal fragmentation, particularly when the object sizes do not align well with the underlying hardware's memory page size. To combat this, some slab allocators employ a technique known as "colorisation," which involves varying the starting offsets of objects within slabs to optimise cache utilisation, thereby ameliorating the impact of internal fragmentation.

In practical applications, slab allocators are often used in operating system kernels, databases, and real-time systems, where the predictable allocation and deallocation behavior is necessary. For instance, the Solaris operating system was one of the first to implement a slab allocator to manage kernel objects [9], achieving remarkable improvements in speed and memory utilisation.

While slab allocators excel in terms of speed and object reusability, they do introduce some level of complexity. The management of slabs and caches demands additional metadata, which consumes memory. Moreover, the initialisation of slab caches can be computationally intensive, although this is generally a one-time cost.

In conclusion, slab allocators stand as a persuasive yet specialised aspect of memory management. Their focus on fixed-size object allocation and deallocation minimises overhead and enhances performance but does so at the cost of increased complexity and potential internal fragmentation.

Fixed-Block Allocators

The foundation of a fixed-size block allocator lies in its division of memory into uniform, fixed-size blocks. This eliminates the need for complex calculations or metadata searches during the allocation process. A bitmap or a similar data structure is commonly used to track the allocation status of each block, enabling faster allocation and deallocation without significant overhead [56].

One of the most effective advantages of fixed-size block allocators is their efficiency in both allocation and deallocation. The use of a bitmap or a free-list ensures that the allocator can quickly determine the location of a free block, usually in constant time (O(1)). This speed is particularly beneficial in real-time or embedded systems, where deterministic behavior is essential.

In contrast to the single block sizes accommodated by a singular linked list in traditional linked list allocators, the fixed-size block allocator adopts a more broad strategy. Specifically, it uses separate linked lists for each distinct size class, eliminating the complexities associated with variable block sizes [37].

To illustrate, a system where block sizes of 16, 64, and 512 bytes are utilised. In this scenario, the allocator would instantiate three linked lists within the memory, each dedicated to one of these block sizes. Instead of maintaining a singular head pointer, the system employs multiple, designated head pointers, each of which points to the first unused block in its corresponding size class.

One of the most important advantages of this approach is the removal of the need to store block sizes within each node of the list. Given that all nodes within a particular list are uniform in size, the size information is specified by the head pointer's name. This not only reduces the overhead of storing additional metadata but also streamlines the allocation process. The uniformity of block sizes within each list has another benefit: it causes each element within a list equally suitable for fulfilling an allocation request. As such, the allocator can execute an allocation operation with efficiency, following a sequence of straightforward steps.

While fixed-size block allocators do well in terms of speed and predictability, they are not immune to internal fragmentation [68]. The granularity of fixed-size blocks implies that any allocated but unused space within a block leads to wasted memory. The severity of this fragmentation is inversely proportional to the block size, making the choice of block size an important design decision.

Fixed-size block allocators are frequently used in scenarios where the allocation size is predictable and the overhead of a more flexible allocator cannot be justified [68]. Examples include real-time systems, embedded applications, and certain high-performance computing tasks where memory allocation patterns are well-understood and stable.

The decision to use a fixed-size block allocator in this project was mainly influenced by the requirement for fast and deterministic memory allocation and deallocation. Given the project's focus on evaluating memory usage in a custom kernel, the fixed-size block allocator's predictability and low overhead make it an ideal choice.

It enables the system to perform allocations with minimal computational cost, allowing for a more accurate and focused evaluation of memory utilisation patterns.

2.5 Exceptions and interrupts

In the complex ecosystem of computer systems architecture, the mechanisms for handling exceptions and interrupts stand as an important component, central to both the system's reliability and efficiency. These mechanisms act as the foundation between hardware events and software responses, ensuring that the system can gracefully handle unexpected conditions and external inputs.

At the most fundamental level, exceptions and interrupts serve as signals that change the normal flow of execution, for different reasons and from different sources [29]. Exceptions are generally internal events triggered by the CPU when a strange condition is encountered during instruction execution, such as division by zero or invalid memory access. Interrupts, are external events often initiated by peripheral devices like disk drives and keyboards to signal the completion of an I/O operation or other state changes.

Interrupts, originating largely from outside the CPU, are typically classified based on their source and urgency; Hardware Interrupts are signals from peripheral devices indicating a change in state or the completion of an operation [46]. Examples include keyboard presses, disk I/O completion, and network packet arrivals. Software Interrupts often deliberately invoked by programs or the operating system, these interrupts facilitate transitions from user mode to kernel mode, enabling system calls and other privileged operations. Non-Maskable Interrupts (NMIs): These are high-priority interrupts that cannot be disabled, serving as critical signals that demand immediate attention, such as hardware failures or emergency shutdowns.

Timer Interrupts are generated by the system's timer, these interrupts are essential for preemptive multitasking, enabling the operating system to regain control and potentially switch tasks.

2.5.1 Interrupt descriptor table (IDT)

The Interrupt Descriptor Table (IDT) plays a crucial role as the gateway between these signaling events and their corresponding handling routines. In system memory, the IDT is an array of descriptors that map each exception or interrupt vector to a specific handler function within the operating system kernel. When an exception or interrupt occurs, the CPU utilizes the IDT to identify the appropriate handler function, thereby decoupling the hardware signaling mechanism from the software-level response.

Upon the invocation of an interrupt or exception handler, the state of the CPU must be preserved to enable a seamless return to normal execution [46]. This is achieved through the construction of an Interrupt Stack Frame (ISF), a data structure that captures the critical CPU state, including register values and program counter, at the moment the interrupt or exception was triggered. The ISF is pushed onto the system stack, providing a snapshot of the CPU state that can be restored upon the completion of the interrupt or exception handling routine [52]. The handling of exceptions and interrupts involves a sequence of steps. Upon the occurrence of an exception or interrupt, the CPU first suspends its current execution and consults the IDT to locate the relevant handler function. Once identified, the CPU constructs the ISF and pushes it onto the system stack. The handler function is then invoked to address the exception or process the interrupt. Finally, the CPU state is restored from the ISF, and execution resumes from the point of interruption [29]. In multi-threaded or multicore environments, the management of exceptions and interrupts becomes complex, requiring mechanisms for synchronisation and potentially nested handling. The IDT and ISF remain integral in these scenarios, serving as the elements upon which additional layers of management and synchronisation are built.

2.6 Process Management

2.6.1 Context switching

In the realm of multitasking operating systems, context switching serves as a cornerstone, providing the illusion of concurrent execution for multiple processes. This operation is an example of the differences between software and hardware, involving a coordinated sequence of actions that transitions the CPU from the context of one process to another [41]. Two key components that facilitate this transition are the Interrupt Stack Frame (ISF) and page tables [24].

Context switching involves the preservation of the complete execution state of the currently running process [61], followed by the restoration of the saved state for the next process to be executed. This state includes, CPU register values, program counter, stack pointers, and memory access information. A disruption, often in the form of a timer interrupt or a system call, triggers the context switch, prompting the operating system's scheduler to determine the next process to be executed.

The Interrupt Stack Frame plays a pivotal role in preserving the CPU state during a context switch. When the context switch is triggered, the CPU automatically constructs an ISF, which encapsulates critical state

information, such as the program counter and various register values. This ISF is pushed onto the stack of the currently executing process, ensuring that the CPU state can be meticulously restored when the process resumes execution. The ISF serves as a snapshot, capturing the precise state of the CPU at the moment of interruption, thereby enabling a seamless transition between different process contexts.

Page tables, the controllers of virtual memory systems, add another layer of complexity to context switching. The page table of a process defines the mapping between its virtual and physical memory addresses [35]. During a context switch, it becomes crucial to replace the current process's page table with the page table of the next process. This is typically achieved by updating the page table base register to point to the new page table, ensuring that memory references by the newly scheduled process are correctly translated to their corresponding physical addresses. Consequently, the role of the page table extends beyond just memory management; it becomes an essential element in the context switching mechanism.

Given the low-level manipulations involved, context switching is a critical section of code that often requires careful synchronisation, especially in multi-threaded or multi-core systems. The use of the ISF and page tables must be atomic to prevent inconsistencies and ensure that the switch is both reliable and efficient.

In summary, context switching represents a complex orchestration of hardware and software components, designed to provide the foundational support for multitasking in modern computer systems. The Interrupt Stack Frame and page tables stand as crucial architectural elements in this operation, facilitating the preservation and restoration of process-specific states, thereby enabling the CPU to transition smoothly between different execution contexts.

2.6.2 Scheduling algorithms

The scheduling algorithms employed by the operating system serve as a critical factor of overall system performance and responsiveness [39]. These algorithms govern the allocation of CPU time among competing processes, trying to balance a myriad of objectives such as fairness, efficiency, and throughput. Among the most fundamental scheduling algorithms are First-Come, First-Serve (FCFS), Round Robin, and Priority Scheduling, each with its own unique set of advantages and limitations.

First-Come, First-Serve is arguably the simplest and most straightforward scheduling algorithm. Operating on a queuing model, FCFS allocates CPU time to processes in the order they arrive, providing a fair and transparent mechanism. However, its simplicity can be a double-edged sword, leading to the "convoy effect," where short processes wait for long processes to release the CPU, thereby increasing wait times and reducing system efficiency.

In contrast, the Round Robin algorithm introduces time-slicing to alleviate some of the limitations of FCFS [39]. In this model, each process is allocated a fixed time quantum, after which it is moved to the end of the queue, and the CPU scheduler picks the next process in the ready queue. This ensures that all processes get a fair share of the CPU and that no single process hogs the resource. However, the choice of time quantum is critical; too short a quantum leads to excessive context switching, while too long a quantum leaves the system back in an FCFS model.

Priority Scheduling adds a layer of complexity by assigning a priority level to each process, and the CPU is allocated based on these priorities. High-priority processes are executed before lower-priority ones, providing a mechanism to ensure that critical tasks are not starved of CPU time. While this model offers greater control, it is susceptible to "starvation," where low-priority processes may never get scheduled if high-priority processes continually demand the CPU.

Moreover, variations of these basic algorithms exist, often combining features to form hybrid scheduling algorithms like the Multi-Level Queue Scheduling, which partitions the ready queue into several separate queues, each with its own scheduling algorithm, or the Completely Fair Scheduler used in Linux, which aims to maintain a balance of 'fair' CPU usage across all processes.

In summary, scheduling algorithms play an important role in shaping the behavior and performance of computer systems. While First-Come, First-Serve offers simplicity and fairness, its inefficiencies are mitigated by the time-slicing approach of Round Robin. Priority Scheduling, on the other hand, introduces a mechanism for preferential treatment of processes but requires careful design to avoid starvation. These algorithms, either in their pure forms or as hybrids, form the bedrock upon which multitasking operating systems are built, each contributing its own blend of advantages and challenges to the intricate landscape of CPU scheduling.

2.6.3 IPC

Inter-Process Communication (IPC) serves as a foundational mechanism, enabling discrete processes to coordinate their activities and exchange information. Through a variety of techniques, IPC provides a structured framework for data sharing and process synchronisation, enhancing both the functionality and performance of the system. Among the IPC methods are message passing, shared memory, pipes, remote procedure calls, and rendezvous systems, each offering its own unique advantages and trade-offs [67] [38].

Message passing facilitates clear separation between interacting processes, reducing the likelihood of data corruption and simplifying debugging. However, the overhead involved in the packaging, sending, and unpacking of messages can impact performance, particularly in high-throughput or low-latency systems. In contrast, shared memory offers a more direct avenue for IPC, as it allows multiple processes to access a common memory region. While efficient, shared memory introduces complexities surrounding synchronization, often necessitating additional mechanisms like semaphores or mutexes [38].

Pipes, commonly used in Unix and Unix-like operating systems, offer a unidirectional or bidirectional channel through which data can be read and written, much like file operations. Though generally limited to parent-child or sibling processes, pipes offer a simple yet effective means of IPC for specific use-cases. Remote Procedure Calls (RPCs) expand the IPC landscape to include distributed systems, abstracting much of the complexity associated with message passing to enable seamless interaction between processes on different machines.

Additionally, rendezvous systems represent another form of IPC that emphasizes strong synchronization between processes. In a rendezvous system, both the sending and receiving processes are blocked until the message exchange is complete, thereby creating a "meeting point" or "rendezvous" between them. This synchronous or blocking form of communication ensures robust synchronisation but can incur performance costs if either process is delayed. It is particularly useful in real-time or tightly-coupled systems where synchronization is a critical requirement.

2.6.4 Multithreading

Multithreading stands as a crucial advancement, serving to maximise CPU utilisation and enhance system responsiveness. Multithreading introduces the concept of concurrent execution within a single process, enabling multiple threads to share resources while operating in parallel. Among the popular techniques employed to create multithreading systems are preemptive and cooperative threading models [1], thread pooling, and data-level and task-level parallelism, each offering a unique set of characteristics and challenges.

Preemptive multithreading, one of the most commonly used models, allows the operating system to forcibly interrupt a running thread to give way to another, ensuring a fair distribution of CPU time among all threads [47]. This model provides robustness and system responsiveness but introduces complexities such as the potential for race conditions, requiring careful synchronisation through mechanisms like mutexes and semaphores. Cooperative multithreading, on the other hand, relies on threads to voluntarily yield control of the CPU, either when they reach a predetermined point or when they are idling. While this approach simplifies synchronization, it poses the risk of thread starvation if a thread fails to relinquish control.

Thread pooling serves as another technique, designed to minimise the overhead associated with thread creation and destruction by reusing a pool of threads. This is particularly advantageous for handling short-lived tasks in high-throughput systems, as it alleviates the performance impact of frequent thread creation and termination. However, thread pooling requires careful management to ensure that the optimal number of threads is maintained for the specific workload, avoiding both underutilisation and overutilisation of system resources.

Data-level parallelism and task-level parallelism represent two approaches to multithreading that focus on the nature of the tasks being parallelised. Data-level parallelism involves breaking a large dataset into smaller chunks and processing them concurrently, making it well-suited for vector and matrix operations. Task-level parallelism, conversely, decomposes a problem into discrete, often independent tasks that can be executed in parallel, making it ideal for complex algorithms and workflows that can be partitioned into smaller sub-problems.

Furthermore, emerging paradigms like speculative multithreading seek to exploit parallelism in irregular and data-dependent algorithms by speculatively executing code paths, although this introduces its own set of challenges related to rollback and state consistency.

2.6.5 Kernel and User mode

In the architecture of modern computer systems, the distinction between kernel mode and user mode emerges as a key design paradigm that plays a crucial role in ensuring both system stability and security. This dual-mode operation, which traces its origins back to the early days of operating systems, serves to segregate the responsibilities and privileges according to the operating system and user-level applications [63].

Historically, the beginning of multi-user operating systems necessitated a mechanism to protect the system and its resources from error prone or malicious behavior. Early computing systems operated in what could be considered a singular mode, lacking the specialised protections we take for granted today.

In kernel mode, the operating system uses unrestricted access to all system resources and hardware, including privileged CPU instructions and unrestricted memory access. This unrestricted access is essential for the operating system to perform tasks such as memory management, task scheduling, and hardware interaction. However, this omnipotence also carries inherent risks, as a faulty or malicious kernel-level operation could lead to system-wide instability or compromise.

Conversely, user mode serves as a constrained environment where user-level applications execute. In this mode, direct access to hardware and certain CPU instructions is prohibited. Applications running in user mode are restricted to a subset of the CPU's instruction set and can only access a limited portion of memory, thereby safeguarding the system's integrity. Any attempt to perform a restricted operation, such as accessing hardware directly, triggers an exception that transfers control to the operating system, which can then decide whether to grant the request, terminate the application, or take other appropriate actions.

The transition between these two modes is carefully controlled and usually occurs through specific system calls [24], which act as gateways for user-level applications to request services from the operating system. Upon invocation of a system call, the CPU switches from user mode to kernel mode, enabling the execution of privileged operations. Once the operation is complete, control is returned to the calling application, and the CPU reverts to user mode, thereby maintaining the sanctity of the dual-mode operational paradigm.

2.7 Rust Programming language and Memory safety

In the ever-evolving landscape of programming languages, Rust has emerged as a popular one, gathering attention for its unique approach to systems programming. Designed with the explicit goals of performance, reliability, and most notably, memory safety, Rust offers a rich set of features that distinguishes it from its predecessors and contemporaries. While its roots can be traced to the broader lineage of systems programming languages, Rust's emphasis on preventing memory errors such as null pointer dereferencing and buffer overflows sets it apart, thereby addressing some of the long-standing challenges in software development.

The history of programming languages is packed with efforts to balance the often orthogonal goals of performance and safety. Early languages like C and Assembly offered high performance and low-level access to computer hardware but provided little in the way of memory safety, resulting in an ecosystem where errors could easily lead to system crashes or vulnerabilities. Languages such as Java and Python prioritised safety and ease of use but often at the expense of raw performance.

What sets Rust apart is its ownership model [6], a feature that enforces memory safety without needing a garbage collector. This model requires that every piece of data have a single 'owner' and sets strict rules on borrowing and lifetime, which are checked at compile-time. As a result, many of the issues that come with languages like C and C++, such as data races and dangling pointers, are prevented, without incurring the runtime overhead typically associated with managed languages.

Additionally, Rust's strong type system and emphasis on concurrency further contribute to its robustness [58], making it particularly well-suited for the development of performance-critical applications like operating systems, database engines, and even browser components. Another noteworthy aspect of Rust is its thriving open-source community, which has led to a large ecosystem of libraries and tools, further easing the development process. The language has also been used by multiple technology companies and has seen adoption in a variety of domains, from web assembly to embedded systems, signaling its growing importance in the contemporary programming landscape. As well as the recent approval of using Rust in the Linux Kernel.

In summary, Rust stands as a unique language in the large field of programming languages, offering a novel approach that combines performance and safety, enabled by features like its ownership model and strong type system. In doing so, it addresses some of the most persistent challenges that have been present since the early days of systems programming, offering a modern alternative that learns from both the triumphs and tribulations of its predecessors.

Here's a simple C code snippet that attempts to access an element outside the bounds of an array:

```c
#include <stdio.h>

int main() {
    int arr[3] = {1, 2, 3};
    int index = 4;  // Out of bounds index
    int value = arr[index];  // Undefined behavior
    printf("Value: %d\n", value);
    return 0;
}
```

Now the equivalent code written in Rust:

```rust
fn main() {
    let arr = [1, 2, 3];
    let index = 4;  // Out of bounds index
    let value = arr[index];  // Compile-time error
    println!("Value: {}", value);
}
```

In the C code snippet, the attempt to access an array element at an out-of-bounds index results in undefined behavior. Such instances pose a significant risk, as they can lead to memory corruption, data leakage, and in worst-case scenarios, remote code execution vulnerabilities. This lack of boundary checks in C is a well-known caveat and has been the root cause of numerous security incidents.

The Rust language, with its emphasis on memory safety, prevents such risky operations at compile-time. By refusing to compile code that would result in out-of-bounds access, Rust eradicates a broad category of potential security vulnerabilities. This illustrates Rust's innovative ownership model and strong type system, which together enforce memory safety without the need for a garbage collector, allowing for both safe and efficient code.

Here's a C code snippet that suffers from a buffer overflow vulnerability:

```
    #include <stdio.h>
#include <string.h>

int main() {
    char buffer[10];  // Fixed-size buffer
    strcpy(buffer, "This is a buffer overflow");  // Overflowing the buffer
    printf("Buffer: %s\n", buffer);
    return 0;
}
```

In this example, the program uses the strcpy function to copy a string into a buffer that is not large enough to hold it. This results in a buffer overflow, leading to undefined behavior and potential security vulnerabilities, such as code injection or data corruption.

Rust handles a similar situation:

```
fn main() {
    let mut buffer = [0u8; 10];  // Fixed-size buffer
    let data = b"This is a buffer overflow";  // Data to be copied
    buffer.copy_from_slice(&data[0..10]);  // Copy data into buffer, ensuring no overflow
    println!("Buffer: {:?}", &buffer);
}
```

The C code snippet presents a classic case of a buffer overflow vulnerability, a notorious issue that has been the root cause of numerous security incidents over the years. By allowing data to be written past the end of an allocated buffer, C allows a range of exploits, including arbitrary code execution and unauthorised data access. This comes from C's design philosophy, which prioritises performance and low-level access to hardware but lacks robust mechanisms for ensuring memory safety.

In contrast, Rust's design incorporates safeguards against such vulnerabilities. By enforcing length checks either at compile-time or runtime, Rust eliminates a broad category of potential security risks associated with buffer overflows. This is part and parcel of Rust's broader emphasis on memory safety, enabled by its strong type system and ownership model. When compared to C, Rust provides a more secure environment for developing systems software, without sacrificing performance.

The contrasting behaviors of these code snippets highlight the trade-offs between performance and safety in systems programming languages. While C offers low-level access and high performance, it lacks the safeguards to prevent memory-related errors. Rust, on the other hand, strikes a balance by offering low-level control coupled with strong safety guarantees, thereby representing a significant advancement in the field of safe systems programming.

2.8 Threat models against Non-memory safe kernels

In the realm of cyber security, the concept of threat modeling serves as an approach to identify, manage, and mitigate potential vulnerabilities. When it comes to standard, non-memory safe kernels—often developed in languages like C and Assembly that lack strict memory safety features—the range of threat models encompasses several key categories. Among these are buffer overflows, privilege escalation, arbitrary code execution, and denial-of-service attacks, each representing its own unique set of challenges and historical developments.

Buffer overflow attacks, one of the most notorious forms of security vulnerabilities, have been exploited since the late 1980s [21]. They occur when an excess amount of data is written into a fixed-size buffer, overwriting adjacent memory. This can lead to unpredictable behavior, from data corruption to the execution of malicious code. The Morris Worm, one of the first computer worms distributed via the internet, exploited a buffer overflow vulnerability, marking it as one of the earliest instances of such an attack.

Privilege escalation is another significant threat model that targets the permissions and access controls within a system. This type of attack aims to gain unauthorised access to resources by exploiting vulnerabilities in the kernel or associated software [14]. Since the kernel operates with the highest level of privilege, any compromise at this level can lead to full system control, making privilege escalation a particularly pernicious form of attack. Incidents involving privilege escalation have been a recurring theme in the history of cybersecurity, highlighting the critical importance of robust access control mechanisms.

Arbitrary code execution threats involve the injection and execution of unauthorised code within a system. These attacks often leverage existing vulnerabilities, such as buffer overflows or insecure system calls [23], to execute code with the same privileges as the compromised application or service. In the context of a non-memory safe kernel, the lack of strict memory safety checks provides ample opportunities for such vulnerabilities to exist, thereby increasing the risk of arbitrary code execution.

Denial-of-service (DoS) attacks, while generally less catastrophic than the aforementioned threats, can still have a large impact on system performance and availability. By overwhelming system resources or exploiting vulnerabilities to induce system crashes, DoS attacks aim to make services unavailable, thereby disrupting normal operations. In kernels lacking memory safety features, resource exhaustion can be exacerbated by the exploitation of memory leaks or buffer overflows, making the system more susceptible to DoS attacks.

In summary, the threat landscape against non-memory safe kernels is complex, ranging from buffer overflows and privilege escalation to arbitrary code execution and denial-of-service attacks. Each of these threat models presents its own set of challenges and has been the focus of extensive research and mitigation efforts. The historical tendency to prioritise performance over security has resulted in an overload of vulnerabilities, highlighting the growing importance of memory-safe programming paradigms.

3 Related Work

Memory Safety for Low-Level Software/Hardware Interactions

https://www.usenix.org/legacy/event/sec09/tech/full_papers/criswell.pdf

The paper in focus presents an analysis and set of solutions aimed at enhancing memory safety in operating systems, particularly those that are not designed with memory safety as a core feature. It addresses a critical gap in current memory safety practices by observing the behavior of low-level software-hardware interactions such as memory-mapped I/O, MMU configurations, and context switching. The paper argues that existing systems, even those that claim to provide a 'safe execution environment,' often fail to account for the vulnerabilities introduced by these low-level interactions. This gap in safety measures can result in exploitable vulnerabilities, negating the guarantees promised by traditional safety checking systems.

To fix this, the authors introduce a set of program analysis and runtime instrumentation techniques that extend the functionalities of a compiler-based virtual machine called Secure Virtual Architecture (SVA). They demonstrate that their modifications to SVA required only minimal changes to the kernel (just around 100 lines of code) and incurred negligible performance overhead. Importantly, these techniques were proven to effectively prevent known memory safety violations specific to low-level Linux operations that would otherwise bypass existing safety mechanisms.

Firstly, it presents novel mechanisms that ensure low-level kernel-hardware interactions do not violate the assumptions made by memory safety checkers; secondly, it demonstrates the practical applicability of these mechanisms through an implementation in the SVA system; and thirdly, it evaluates the effectiveness and efficiency of these mechanisms, showing they add little overhead while successfully preventing real-world exploits.

This paper provides invaluable insights. As I delve into the specifics of memory management, understanding the low-level interactions and the vulnerabilities they may introduce will offer a broader perspective. The paper's techniques can serve as a guide to design the system's safeguards, especially when dealing with the hardware-software interface. Moreover, their empirical approach to evaluating memory safety can inform the methodology I may employ to test the memory safety of the kernel.

Towards Linux Kernel memory safety

https://onlinelibrary.wiley.com/doi/epdf/10.1002/spe.2638?saml_referrer

The paper called "Toward Linux Kernel Memory Safety" talks about two ways to make the Linux kernel more secure and reliable. It also looks at how these changes affect the system's speed and ease of use. The authors believe that issues with memory safety in the kernel are a big problem. Attackers can take advantage of these issues to take over the system. To fix this, the authors suggest two solutions: a new kind of reference counter and a project to make the kernel protect itself.

The new reference counter is made to stop a type of attack that tricks the system by making numbers overflow. The authors show that adding this feature doesn't slow the system down much, but it does make it more secure.

The self-protection project adds multiple security features directly into the main Linux kernel. This makes many different kinds of devices more secure right from the start. The authors share what they learned from adding these features to the main Linux kernel and give advice that could help others do the same.

Even though this paper doesn't talk about using Rust to build a kernel, it does offer useful information. It shows what problems need solving and gives examples of how to solve them. This could be helpful for anyone looking to make a more secure and reliable kernel in Rust, as it emphasizes the need for memory safety and suggests ways to achieve it.

RedLeaf: Isolation and Communicationin a Safe Operating System

"https://www.usenix.org/system/files/osdi20-narayanan_vikram.pdf"

The paper delves into the potential of using Rust, a programming language designed for system-level tasks that offers strong memory safety features, to build a modern operating system kernel. The authors introduce RedLeaf, an entirely new operating system developed in Rust, and assess its speed and security features.

A significant part of this paper is the development and testing of two device drivers, one for a 10Gbps Ethernet and another for PCIe-connected SSDs. The aim was to examine how Rust's safety mechanisms, which come at zero cost to performance, affect these low-latency subsystems. The findings suggest that Rust's safety features have a minimal impact on the performance of these device drivers, reinforcing that Rust is a viable option for creating efficient kernel components.

RedLeaf is designed with a focus on practicality and speed, aiming to serve as a full-featured alternative to mainstream kernels like Linux. The microkernel offers a streamlined interface for managing isolated domains, thread execution, scheduling, and interrupt handling, as well as memory management. It employs memory management techniques akin to those in Linux. Each domain in RedLeaf operates its own memory allocator and requests memory directly from the kernel's buddy allocator.

Additionally, the paper introduces a new concept of lightweight fine-grained isolation, executed through Rust's type system and ownership model. This allows for the development of isolated domains with minimal overhead, offering strong safety guarantees without the need for hardware-based isolation methods.

In summary, the paper offers important insights into the challenges and advantages of using Rust to develop an operating system kernel. It establishes that Rust's built-in safety features do not hinder performance to a significant extent and introduces a new method for creating isolated, efficient, and secure kernel components.

From L3 to seL4 What Have We Learnt in 20 Years of L4 Microkernels?

"https://sigops.org/s/conferences/sosp/2013/papers/p133-elphinstone.pdf"

The paper "From L3 to seL4: What Have We Learnt in 20 Years of L4 Microkernels" offers a comprehensive look at the development and transformation of L4 microkernels over two decades. It primarily focuses on the key lessons garnered from years of microkernel design and how these insights have shaped the current generation of L4 microkernels, most notably seL4.

The study emphasizes the core philosophies that have consistently guided the design of L4 microkernels, notably the principles of minimalism and high inter-process communication (IPC) performance. The authors assert that despite the changes and advancements over the years, these fundamental principles have remained integral to design choices.

In examining the progression of L4 microkernels, the paper delves into the attributes that define modern L4 kernels and how these relate to the original design specifications outlined by Liedtke. It also explores how these essential features have stood the test of time while pointing out divergent design decisions in other current L4 iterations.

The discussion then shifts to seL4, a groundbreaking version that has taken the L4 model to new heights. Notably, seL4 is the first operating system kernel to have a fully verified implementation and has undergone thorough analysis for worst-case execution times. The authors underscore the pivotal role of seL4 in shaping the new generation of L4 microkernels.

In summary, the paper offers invaluable perspectives on the maturation of L4 microkernels and the lessons gleaned from years of microkernel design. Though it doesn't directly address kernel development in Rust, it lays a robust foundation for comprehending the guiding principles and methods that influence microkernel design, which could be beneficial for kernel development in any language, including Rust.

https://cs.brown.edu/media/filer_public/47/83/47833fec-2224-4836-a6c0-41d552213818/lightalex.pdf

This paper by explores the feasibility of implementing a Unix-like operating system kernel in Rust, a modern programming language that offers safety and type systems that are not present in C, the language traditionally used for kernel development. The paper provides a detailed account of the author's experience in creating a basic kernel that supports multiple processes, drivers, and a virtual file system, using the Weenix operating system as a starting point.

The paper begins with an introduction to the Weenix OS and the Rust language, highlighting the advantages of Rust over C in terms of safety and expressiveness. The author then describes the organization of the Reenix kernel, including the booting process, memory and initialization, and process management. Throughout the paper, the author compares the performance of Rust and C implementations of the kernel, noting where Rust's safety and type systems helped or hindered the development process.

One of the key contributions of this paper is its demonstration that a Unix-like operating system kernel can be built using Rust. The author concludes that Rust offers several advantages over C, including simpler implementation of support code and improved safety and expressiveness. However, the author also notes that there were still some rough edges in Rust's implementation of low-level systems programming, and that some aspects of the development process were more difficult in Rust than in C. However, due to recent developments in the supporting crates of Rust, some of these worries have been solved.

Overall, this paper provides a valuable resource for developers interested in kernel development in Rust. It offers a detailed account of the challenges and successes of implementing a kernel in Rust, and provides insights into the strengths and weaknesses of Rust as a language for systems programming. By reading this paper, developers can gain a better understanding of the potential benefits and limitations of Rust for kernel development, and can use this knowledge to inform their own development projects.

Ownership is theft

https://dl.acm.org/doi/epdf/10.1145/2818302.2818306

The paper investigates the practicality of constructing a Unix-like operating system kernel in Rust, a contemporary programming language endowed with safety features and robust type systems absent in C, the conventional language for kernel development. The paper draws on the author's hands-on experience of developing a rudimentary kernel that accommodates multiple processes, drivers, and a virtual file system, using the Weenix operating system as a foundational framework.

Starting with an overview of the Weenix operating system and the Rust programming language, the paper accentuates the merits of Rust over C, particularly focusing on aspects of safety and language expressiveness. The author then elaborates on the architecture of the Reenix kernel, delineating its booting mechanism, memory management, and process control. Throughout this discussion, comparative performance metrics between Rust and C kernel implementations are provided, with specific attention to how Rust's safety and type systems either facilitated or complicated the development.

A notable contribution of this paper is its empirical evidence demonstrating the feasibility of building a Unix-like kernel in Rust. The author concludes that Rust confers several advantages over C, such as simplified support code implementation and enhanced safety measures. Despite this, the author acknowledges certain limitations in Rust's suitability for low-level systems programming, although some of these concerns have been alleviated through updates in Rust's supporting libraries.

In summary, the paper serves as an invaluable guide for developers intrigued by the prospect of kernel development in Rust. It furnishes a nuanced narrative of the complexities and triumphs encountered during the kernel's implementation in Rust, thus offering insights into Rust's capabilities and constraints for systems programming. This comprehensive account allows developers to make more informed decisions in their kernel development endeavors, taking into consideration the benefits and drawbacks of employing Rust.

Exploring Rust for unikernel development

https://dl.acm.org/doi/abs/10.1145/3365137.3365395

The paper delves into the intricacies of constructing an embedded operating system using Rust, a modern language explicitly designed for systems programming with a focus on safety. The authors elucidate the merits and obstacles of employing Rust for the development of an operating system, especially within the confines of embedded systems characterized by limited hardware capabilities and event-driven architectures.

The authors identify three primary challenges that arise when using Rust for an embedded kernel. First, Rust's automated memory management is not tailored for the specialized requirements of hardware resources

and device drivers. Second, the language's ownership model hampers resource sharing between closures and the broader kernel code. Third, the use of closures necessitates dynamic memory allocation, an operation that is problematic in the resource-constrained environment of embedded systems. The authors detail their creative solutions to these issues, which often involved broadening the trusted computing base of the operating system to bypass certain restrictions imposed by Rust's type system.

In a bid to enhance Rust's applicability to event-driven embedded platforms, the authors introduce a proposed language feature termed "execution contexts." This innovation aims to furnish Rust with the means to facilitate safe memory sharing, particularly in scenarios where hardware limitations or specific execution models can reliably mitigate concurrency-related risks.

In summary, the paper stands as a valuable resource, offering both a nuanced account of the real-world challenges of utilizing Rust in embedded operating systems and practical solutions to those challenges. Moreover, the proposed language feature, if implemented, could significantly advance Rust's utility for ensuring memory safety in event-driven platforms.

4 Design

The Design chapter serves as the architectural blueprint of the kernel developed for this project, offering a meticulous breakdown of its essential components and the rationale behind various design choices. It aims to provide a comprehensive understanding of how the kernel is structured to achieve its objectives, particularly focusing on memory management and security features. Leveraging Rust as the programming language, this chapter will elucidate how its unique features contribute to the kernel's memory safety and performance. Each section will delve into specific aspects of the kernel, from low-level hardware interactions to high-level abstractions, substantiating the design decisions with empirical data and relevant academic literature. As we navigate through this chapter, we will unravel the complexities of creating a memory-efficient and secure kernel, thereby setting the stage for the subsequent Implementation and Evaluation chapters.

4.1 Approach

In the approach to designing the kernel for this project, an extensive preparatory phase was undertaken to foster a deep understanding of existing systems, particularly those written in both traditional and memory-safe languages. A considerable amount of time was initially devoted to studying the Linux kernel, given its ubiquity and the wealth of academic and practical insights it offers. This initial exploration involved not just a review of existing literature but also a detailed examination of the Linux kernel's codebase, dissecting its architecture, memory management paradigms, and security measures.

Simultaneously, a thorough investigation was carried out on various open-source operating systems developed in Rust, the language chosen for this project. By diving into the codebases of these Rust-based systems, a nuanced understanding was gained regarding the unique challenges and advantages of employing Rust for kernel development. This duality of perspectives—gleaning insights from a well-established kernel like Linux and emerging Rust-based systems—provided a balanced and comprehensive foundation upon which the design of the project was conceived.

This multifaceted approach was instrumental in identifying the most pertinent features and techniques to be incorporated into the project. It informed the decision-making process at various design junctures, from selecting memory management strategies to implementing security features, ensuring that the resulting kernel would be both robust and efficient. By juxtaposing the traditional methods employed by Linux with the innovative techniques facilitated by Rust, the design approach aimed to synthesize the best of both worlds, crafting a kernel that is not only functionally rich but also ingrained with state-of-the-art memory safety measures. This preparatory work laid the groundwork for the iterative design and development process that followed, the details of which are elaborated upon in the subsequent sections.

4.2 Functional requirements

Booting

(R1) Initialization of Hardware: The system must initialize all essential hardware components such as the CPU, memory, and I/O devices during the boot process.

(R2) Load Bootloader: The system must be capable of locating and executing the bootloader from non-volatile storage.

(R3) BIOS/UEFI Interface: The system must interface correctly with either BIOS or UEFI firmware to facilitate the boot process.

(R4) Initial RAM checks must be performed to ensure that the available memory is both sufficient and error-free.

(R5) The bootloader must locate and load the kernel into memory.

(R6) Pass Control to Kernel: Once loaded, control must be passed seamlessly from the bootloader to the kernel.

(R7) User-Space Initialization: After the kernel is loaded, the system must initialize the user space, setting up the environment for user-level applications.

(R8) Logging and Diagnostics: The system must log key steps in the boot process for diagnostic purposes and must have the capability to display these logs in case of boot failure.

(R9) Error Handling: In case of failure at any step, the system must provide a meaningful error message and should attempt to enter a recovery or safe mode.

Kernel

(R11) Process Management: The kernel must be capable of creating, scheduling, and terminating processes, as well as managing their state transitions.

(R12) Memory Management: The kernel must allocate and deallocate memory as required by processes, including both stack and heap memory.

(R13) Inter-Process Communication (IPC): Support for some form of IPC mechanism, such message queues, semaphores, shared memory, RPCs or Rendezvous should be present.

(R14) Error Handling: The kernel must have robust error-handling mechanisms to gracefully manage errors at runtime.

(R15) Resource Allocation: The kernel needs to efficiently allocate system resources like CPU time and memory to various processes based on scheduling algorithms.

(R17) Logging and Auditing: The kernel must maintain logs for system events and security-relevant transactions.

(R18) Interrupt Handling: The kernel should be capable of handling hardware and software interrupts, dispatching them appropriately.

(R19) System Monitoring: Facilities for monitoring system performance, such as CPU utilization and memory usage, should be available.

(R20) Modularity and Extensibility: The kernel architecture should be modular to allow for easy extension or modification of functionalities.

(R21) System Calls: A set of system calls must be implemented to allow user-space programs to interact with the kernel and utilize its services.

(R22) Input: The system should be able to receive input from I/O, such as a keyboard, and either present those inputs or react accordingly.

User space

(R1) ELF File Loading: The system must be capable of loading ELF files from storage into memory and correctly initializing them for execution.

(R) Thread Creation: The user space must support the creation of multiple threads, each with its own stack and program counter.

(R) Memory Isolation: Each user thread must operate in its own isolated memory space, including separate stack and heap regions, to ensure process security and integrity.

(R) System Calls: The user space should provide an interface to system calls, allowing user threads to interact with the kernel for services like file I/O, networking, resource allocation and message sending.

(R) Process Scheduling: The user space should interface with the kernel's scheduling algorithm to enable pre-emptive or cooperative multitasking between user threads.

(R) Inter-Thread Communication: Facilities must be provided for threads within the same process to communicate and synchronize, such as through message passing or shared memory.

(R) Signal Handling: User threads should be able to register and handle signals for events like termination, suspension, and other thread-specific conditions.

(R) Resource Cleanup: Upon thread termination or process exit, the system must reclaim all allocated resources like memory and file descriptors.

(R) Standard Libraries: Standard libraries for common tasks like string manipulation, mathematical operations, and data structure manipulation should be available in the user space.

(R) Error Handling: Robust error-handling mechanisms should be in place to catch and report runtime errors in the user space, such as illegal memory access or division by zero.

(R) Dynamic Memory Allocation: User threads must be able to dynamically allocate and deallocate memory from the heap during runtime.

4.3 Non-functional requirements

There are several non-functional requirements that should be considered during the course of the project. Mainly looking at the usability and performance of the system.

- Performance: The system should be optimized for high throughput and low latency. Metrics like boot time, context-switch time, and I/O operation speed could be used to gauge performance.
- Scalability: The system should be capable of efficiently utilizing resources as the number of user threads and processes increases.
- Reliability: The system should be designed to operate without failure for extended periods of time, and any failures should be isolated to minimize impact.
- Maintainability: The codebase should be well-organized, well-commented, and adhere to coding standards to allow for easy maintenance and upgrades.
- Version Control: The system should support versioning to manage changes, updates, and compatibility issues.

4.4 High-level Architecture

In this section, an architecture diagram is presented to explain the core functionalities of the system during runtime. It should be noted that the diagram has been deliberately simplified to emphasise the most critical components and their interactions. The objective is to provide a high-level understanding of how the system operates, especially focusing on runtime behavior.

While the diagram serves as an overview, it is worth mentioning that it does not capture all relationships and dependencies within the system. For instance, almost all components in the system are reliant on the memory allocator for dynamic resource allocation; however, such ubiquitous dependencies are not explicitly shown in the diagram. This omission is intended to maintain the diagram's clarity and focus on illustrating the primary architecture rather than detailing every interaction.

The simplified architecture diagram thus serves as an initial guide to the system's core functionalities, providing a foundational understanding upon which more complex relationships and functionalities can be explored. Hopefully, in the following sections, larger clarification will be provided.

The architecture of the system under discussion is compartmentalized into three fundamental sections: the kernel, user space, and an API. Each of these sections plays a distinct yet interrelated role in the orchestration of the system's functionalities.

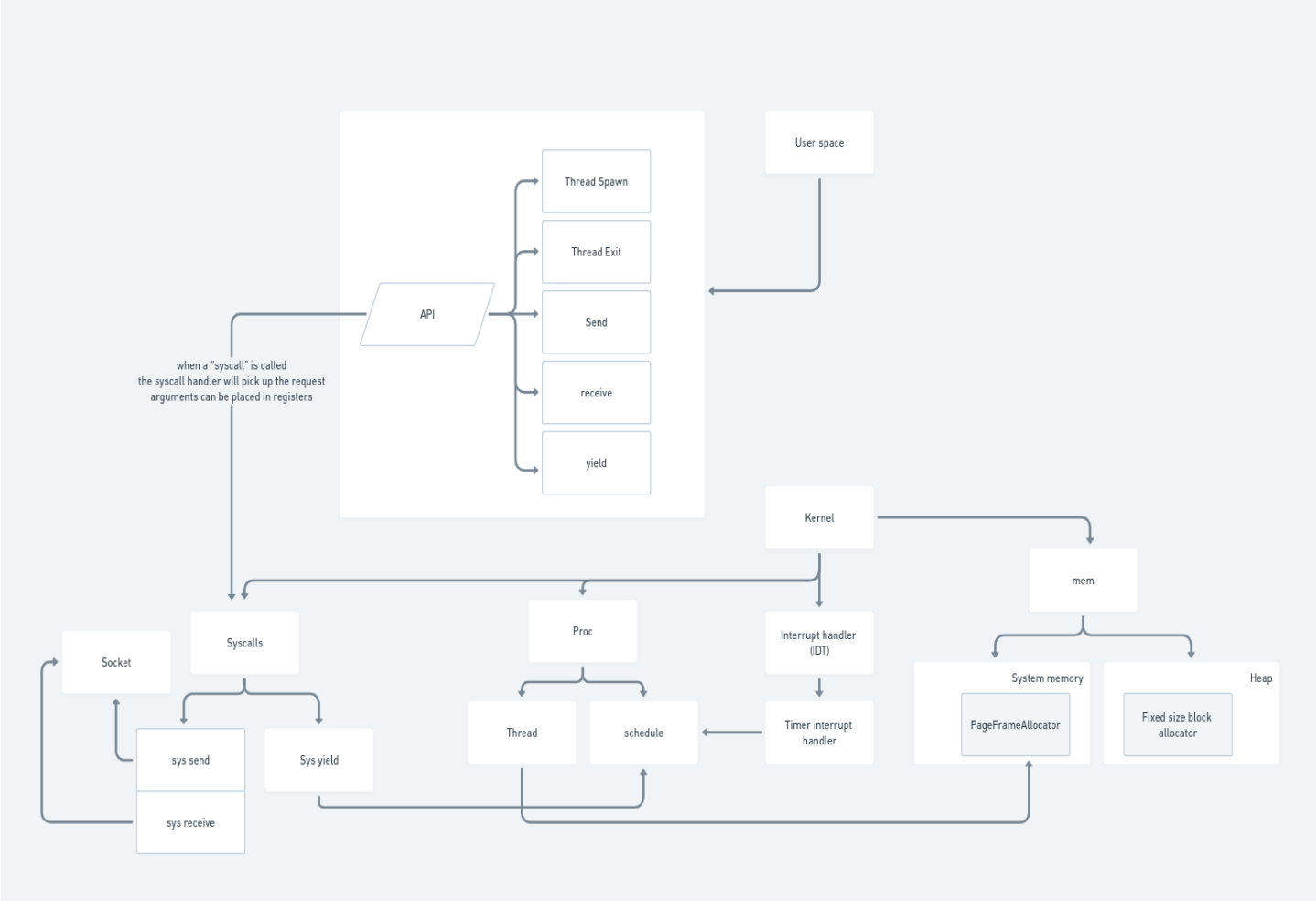


Figure 6: High-level architecture

The kernel serves as the center of the entire system, assuming responsibility for an array of core operations. These operations encompass system booting, memory allocation, process management, and scheduling. Additionally, the kernel is vested with the role of handling interrupts and system calls, as well as loading user binaries into the system. It is imperative to understand that the kernel operates at a privileged level, directly interfacing with the hardware and serving as the foundation upon which higher-level functionalities are built.

Conversely, the user space is essentially a program written in the Rust programming language and compiled into Executable and Linkable Format (ELF). It is noteworthy that ELF serves as the standard file format for the majority of UNIX systems, thereby aligning with conventional practices. The user space is intrinsically reliant on the API section, establishing the latter as a critical intermediary between the kernel and user space.

The API section functions as a controlled gateway to the system's resources, mediating interactions between the user space and the kernel. This is done through a suite of system calls, which are made available to the user space. When a system call is invoked, specific values, including the system call number, are placed into registers, ending in the execution of the "syscall" instruction. Upon invocation, the system call handler within the kernel is triggered, which the kernel informs the CPU where that function lives, retrieving the system call number. This handler, in turn, invokes the corresponding system call, utilizing values stored in particular registers as arguments.

It is worth highlighting that this approach of restricted access to system resources via an API suite is aligned with contemporary architectural designs in computer systems. The model ensures a balance between accessibility and security, allowing user space programs to interact with the system while maintaining robust security protocols.

Thus, this architecture comprising the kernel, user space, and API not only describes the system's structural organization but also describes the interdependencies and data flow that govern its runtime behavior.

The appendix contains large scale call graphs that mostly show how each function within each class is called. Some of the calls are missing, potentially due to the power of the automatic tool that I used, called Crabviz: https://github.com/chanhx/crabviz

4.5 Memory-management

In the architecture of the kernel's memory management system, two principal components are integral: a Page Frame Allocator and a Fixed-Size Block Allocator. The Fixed-Size Block Allocator is primarily responsible for heap memory allocation as well as actually allocating virtual frames, while the Page Frame Allocator serves a more comprehensive role, notably in formulating a 4-level recursive page table based on the system's boot information.

The selection of a 4-level recursive page table for memory management was a deliberate decision, driven by the need for both granularity and flexibility in memory allocation. This approach affords a fine-grained control over memory mapping, enabling nuanced allocation and deallocation of memory spaces—whether they are small or large. The hierarchical structure inherent to the 4-level design allows for facile adjustments in memory mapping, thereby accommodating the dynamic needs of modern systems, particularly those with expansive RAM. Moreover, the intricacies of multi-level memory accesses, which could be a potential drawback, are mitigated by the capabilities of contemporary CPUs. Specifically, hardware features such as Translation Lookaside Buffers (TLBs) efficiently cache multi-level table lookups, thereby reducing the associated performance overhead. This recursive nature of the table also provides the added advantage of facilitating debugging processes; it enables straightforward table-walking mechanisms, thus offering clear visibility into the state of allocated memory.

4.6 Process Management and Scheduling

The architecture for managing processes within the kernel is built upon a model consisting of a wait queue and a running thread architecture. This dual-faceted approach serves as the cornerstone of the kernel's process management, enabling a robust and efficient scheduling mechanism.

Each process within this architectural framework is a parent to a child thread, serving as a structural unit that encapsulates the individual tasks to be executed. The incorporation of a child thread for each process not only modularizes the scheduling but also simplifies the implementation of inter-process communications and synchronization mechanisms.

Uniqueness is a key attribute for these threads, necessitating the provision of a unique identifier for each. This unique identification system allows for precise referencing and manipulation of threads, which is essential for an array of operations, ranging from scheduling to debugging.

A critical aspect of each thread's data structure is the storage of its own context, effectively capturing the state of the CPU at any given time. This context storage serves as a snapshot of the thread's computational state, enabling the kernel to pause and resume threads as needed, thereby facilitating preemptive multitasking.

In addition to the context, each thread maintains references to specific memory locations where its kernel stack and user stack reside. The distinction between these two stacks is crucial for the proper functioning of the system. While the user stack handles memory needs related to the user-space operations of the thread, the kernel stack plays a critical role during the process of thread switching. Specifically, the kernel stack is leveraged to store the thread's context during such transitions. This ensures a seamless switch, preserving the state of the outgoing thread while preparing the CPU to accommodate the incoming thread's computational needs.

In summary, the design of the process management system is a nuanced orchestration of various components—wait queues, running threads, unique identifiers, context storage, and dual-stack architecture—each serving a distinct purpose but collectively contributing to a cohesive and efficient process management framework.

5 Implementation

5.1 Environment Setup and Libraries

5.1.1 IDE

In the course of developing my kernel, I opted to use Neovim [48] as my primary text editor, a decision driven by several compelling factors. First and foremost, Neovim offers an unparalleled level of customisation and extensibility, which allowed me to tailor my development environment to the specific needs of kernel development in Rust. This is particularly crucial for a project of this complexity, where seamless navigation through the codebase and efficient code manipulation can significantly expedite the development process.

Another advantage of Neovim is its lightweight nature, ensuring that system resources are predominantly allocated to the compilation and debugging processes, rather than the text editor itself. This becomes increasingly important when working when working on my "slow" Ubuntu laptop. Additionally, Neovim's robust plugin architecture allowed me to integrate essential tools directly into the editor, from syntax highlighting and code linting to Git integration. This not only streamlined my workflow but also reduced the context-switching overhead, thereby enhancing productivity.

Lastly, Neovim's focus on facilitating asynchronous operations minimized any lag during code autocompletion and other IO-bound tasks, which is crucial for maintaining a smooth and responsive development experience. s

In summary, the choice of Neovim significantly contributed to the efficacy and efficiency of my development process. Its performance-oriented architecture and user-centric design provided an optimized environment that was perfectly suited to the rigorous demands of kernel development in Rust.

5.1.2 Cargo and rustc

In the realm of my kernel development project, two unavoidable tools that have been central to my workflow are Cargo and rustc, the Rust package manager and compiler, respectively. Both have played a pivotal role in shaping the development environment and influencing the overall success of the project.

Cargo [64], Rust's native package manager, offers a plethora of features that expedite the development cycle. It automates many of the mundane tasks associated with project management, from dependency resolution to the compilation process, allowing me to focus on the core logic of my kernel. The facility to easily manage dependencies is particularly useful in the context of this project, as it streamlines the integration of external libraries, ensuring that they are seamlessly incorporated into the build process. This is done within a Cargo.toml file, which easily allows you to manage and add dependencies and their versions from a central location, which is a massive advantage over languages such as Python that require entire virtual environment setups like Conda and venv. Furthermore, Cargo's support for custom build scripts and 'features' allows for a high degree of customization, enabling me to fine-tune the compilation process according to the unique requirements of kernel development. This was particularly important for this project, as creating a custom build script allowed me to define arguments such as what addresses the user text and data segment should reside.

On the other hand, rustc serves as the main compiler of the Rust language, translating high-level Rust code into machine code that can be executed by the hardware. It provides a robust set of optimization features, which are essential for achieving the level of performance required in system-level programming. One of its standout features is the meticulous error reporting, which not only identifies issues but also offers insightful suggestions for rectification. This has been invaluable in debugging the intricate components of my kernel. [17]

However, rustc is not without its limitations. The compiler is relatively slow compared to some other languages, which can be a bottleneck in a large project. This is somewhat mitigated by incremental compilation, but the issue still persists, especially in the context of full rebuilds [71].

In conclusion, both Cargo and rustc have proven to be invaluable assets in the development of my Rust-based kernel. While they each have their own set of advantages and limitations, their synergistic combination

has provided a robust and efficient development environment. Cargo's automation and dependency management have streamlined the build process, while rustc's powerful optimization and error reporting features have significantly aided in producing a high-performance and secure kernel.

5.1.3 Unlinking from the standard library and creating a target

A kernel, by its very nature, operates without the use of an underlying operating system. Thus, our initial step is to craft a Rust executable that excludes the standard library, enabling us to run Rust code directly on the hardware "bare-metal". In Rust, this is accomplished by employing the **![no_std]** attribute at the beginning of the main source file [17]. With this attribute in place, we've essentially severed our code's dependency on any OS-specific functionality, such as file handling, networking, or threading, provided by the standard library. Now, we're able to write a kernel that leverages Rust's intrinsic features like pattern matching, iterators, and the ownership model, while avoiding any undefined behavior or memory safety issues.

The absence of the standard library renders the default panic behavior undefined, as it typically relies on OS-specific functionalities for stack unwinding or aborting the program. Therefore, implementing a custom **[panic_handler]** allows the system to define a controlled behavior during panics, ensuring that the system reacts in a predictable manner. This is particularly critical for a kernel, where unexpected behavior can lead to severe consequences. The custom panic handler serves as the sole mechanism for handling runtime errors and ensuring system integrity.

Typical Rust binaries initiate execution with a C runtime library, known as crt0, which prepares the environment by establishing the stack and setting the appropriate registers. Subsequently, the Rust runtime takes over, performing additional tasks like configuring stack overflow guards, before finally transferring control to the main function [17].

However, in the context of our freestanding executable, there is neither a C runtime nor a Rust runtime to perform these initialisations. As a result, relying on the usual main function is unfeasible, since there is no underlying runtime to invoke it. The absence of these runtimes necessitates the definition of a custom entry point to kickstart the kernel. To achieve this, we utilize the **![no_main]** attribute to inform the Rust compiler that we intend to bypass the standard entry point chain. We then implement our own function, marking it with **[no_mangle]** and extern "C" to ensure proper naming and calling convention, respectively [17].

The custom entry point kernel_main serves as the very first function invoked either by the operating system or the bootloader. Given its unique role, it is diverging in nature, signified by its ! return type, meaning it is not expected to return. We can also use the bootloader crate to define an exact entry point, which essentially remaps a function with a name other than "_start" to a name that we can pick, in this case, I chose "kernel_main". This custom entry point is not just a mere convention but a vital construct for enabling the kernel to function independently of any runtime or operating system.

In the construction of this kernel, the concept of a "target triple" plays an essential role. A target triple is a descriptor consisting of three key components: the CPU architecture, the vendor, and the operating system, sometimes extended with the ABI (Application Binary Interface). Target triples act as a bridge between the code and the system it is intended to run on, guiding the compiler and the linker during the build process.

However, the kernel doesn't fit any of the pre-defined target triples, largely because it runs on bare metal with no underlying operating system. Rust offers the flexibility to define a custom target triple through a JSON file. This JSON file contains a myriad of fields that configure both the LLVM backend and the Rust frontend. These fields specify various attributes like data layout, linker flavor, and architectural features, which are crucial for generating compatible machine code.

```
{
    "llvm-target": "x86_64-unknown-none",
    "data-layout": "e-m:e-i64:64-f80:128-n8:16:32:64-S128",
    "arch": "x86_64",
    "target-endian": "little",
    "target-pointer-width": "64",
    "target-c-int-width": "32",
    "os": "none",
    "executables": true,
    "linker-flavor": "ld",
    "linker": "ld",
    "panic-strategy": "abort",
    "disable-redzone": true,
    "features": "-mmx,-sse,+soft-float"
}
```

For instance, we use "llvm-target": "x86_64-unknown-none" and "os": "none" to indicate that our code will run on a 64-bit x86 architecture without an underlying OS. Creating this custom target specification in a JSON file enables precise control over how our kernel will be built and how it will interact with the hardware.

The compiler_builtins crate provides the low-level, architecture-specific functionalities that are implicitly required by Rust's compiler for various operations, particularly around arithmetic and memory. For memory, this crate supplies implementations of functions like memcpy, memset, and memcmp, which are essential for memory allocation, deallocation, and manipulation at the most basic level. These functions are core to any system, and when working on a kernel or a bare-metal environment, linking with compiler _builtins ensures that these fundamental routines are available for the main code to use.

5.1.4 Booting in QEMU

In the process of creating a bootable operating system kernel, I found that the process is divided into several intricate but manageable steps. The first critical task involved compiling my Rust-based kernel into a binary, a standard binary format that's both executable and linkable [69]. The file serves as an intermediate representation of the kernel that can be processed further.

Realising the complexity of bootloaders, I opted to use the bootloader crate to avoid reinventing the wheel. This crate, provides a basic BIOS bootloader. It's designed to initialise the CPU and load the kernel into memory, bridging the gap between the system's hardware and my kernel.

To streamline the otherwise difficult task of linking the kernel with the bootloader post-compilation, I utilized a specialized tool known as bootimage. This tool attempts to compile both the kernel and the bootloader. More importantly, it links these two components to produce a unified, bootable disk image. The resulting image is now a bootable image that can be loaded in various ways.

Upon booting, the bootloader starts by setting up the CPU environment and loading the kernel into memory. Control is then transferred to the kernel, allowing it to take over the system's operations. When it came to testing the compiled kernel, I faced a choice. One option was to deploy it on actual hardware, a process that, while thorough, is laborious and time-consuming. The other alternative, which I ultimately selected, was to use a virtual machine for emulating the kernel's behavior. Specifically, I chose QEMU, the Quick Emulator, as my virtualization solution.

QEMU offered several compelling advantages that made it the ideal choice for my project. First and foremost, it provides a fast and efficient way to emulate various hardware architectures, including but not limited to x86 and ARM. This flexibility allowed me to simulate how my kernel would behave on different hardware platforms without the need for physical devices.

Secondly, QEMU integrates seamlessly with the debugging tools, offering features like breakpoint support and introspection. This is invaluable when you're dealing with low-level kernel code, where bugs can be cryptic and challenging to identify. With QEMU, I could run my kernel in a controlled environment and easily examine its behavior at any point in time. During the development of the kernel I could create a remote GDB session to get a dump of all register values, this was useful, not perfect. GDB only partially supported the analysis of the binary as it is a custom target, so stepping through the code was not available. Despite this, it was still useful for understanding the state of the system after a failure.

Additionally, using QEMU eliminated the risk of damaging my physical machine. In the early stages of kernel development, crashes are not just expected; they are almost a certainty. Running my still-in-progress kernel on a real machine could have resulted in anything from minor glitches to complete system failure. QEMU provided an isolated sandbox where my kernel could crash and reboot as many times as needed without any repercussions.

Finally, the convenience factor cannot be overstated. Using QEMU, I could effortlessly switch between my development environment and the emulated machine without leaving my workspace. The ability to run a command and instantly see my kernel boot up (or crash) was not just satisfying; it was a huge boost to my productivity.

For these reasons, QEMU was the perfect fit for my project, allowing for rapid iterations in a safe, controlled, and versatile environment.

5.2 Essential libraries and crates

There were several crates that made the developement of the system much easier and in fact possible. This is a short description of the particularly important ones.

5.2.1 Bootloader crate and bootimage

In the course of my operating system development, the bootloader crate emerged as an indispensable asset, providing a rudimentary bootloader to link with the kernel image. The bootimage tool facilitated the compilation

of both the bootloader and the kernel, subsequently linking them to produce a bootable image. Notably, the bootloader crate also provides a BootInfo struct, an important element in initialising the kernel's memory architecture. This struct not only stored the base address of the page tables generated during the bootimage process but also encapsulated additional information. The provision of such pre-configured, high-level abstractions by the bootloader crate streamlined the initialisation process, allowing for greater focus on core development objectives.

5.2.2 x86_64 crate

In the architectural design of my kernel, the x86 crate served as a robust utility, offering an array of helper functions and types that substantially eased the implementation process. The crate provides many things, the Port type was particularly useful, enabling the kernel to interface directly with the QEMU emulation environment through signal transmission. Furthermore, the crate provided types for representing the Interrupt Descriptor Tables, and not only facilitated their instantiation but also managed their loading into the appropriate CPU registers. It also offered types for Interrupt Stack Frames, simplifying the handling of hardware interrupts. The x86 crate provided a comprehensive set of tools that mitigated the complexity of low-level system programming, enabling a more streamlined development experience. It also provided different types to represent memory, including virtual addresses, physical addresses. Along with page table types that can be implemented use the Rust trait system.

5.2.3 UART crate

The UART crate provides a SerialPort type, which allowed the system to output values from outside of QEMU, mainly to the host running the system, in my case, a terminal session. This was invaluable to the progression of the project as it meant that I could debug and log out various aspects of the system methodically and completely. It allows allowed the creation of the Memory Logger and stack monitor. Which could be piped to a log file and processed afterwards. The crate also allows you to call specific interrupts for testing. [69]

5.2.4 lazy_static

In the development of my kernel, the *lazy_static* crate became a resource for managing static variables, particularly those requiring intricate initialisation procedures. Normally, the Rust language necessitates that static variables be initialized at compile-time, constraining their complexity and limiting their use for more involved operations. The lazy_static crate circumvents this limitation by deferring the initialisation until the variable's first access, effectively enabling the use of static variables with runtime-dependent initialisations. This capability proved invaluable for components such as complex data structures and synchronization primitives, which could not be instantiated through Rust's native static construct, an example of this would be the wait queue used to store all of the threads waiting to be scheduled as well as the different sockets required for communication, these will be described more. Therefore, the *lazy_static* crate was a critical tool, allowing for the incorporation of advanced features and functionalities into my kernel without compromising on performance or safety.

5.2.5 ASM

In building my kernel, the *asm* crate was extremely valuable for allowing inline assembly code to be executed. Usually, Rust's safety rules limit direct interactions with hardware or specific CPU instructions, but the asm crate lets us bypass these restrictions. This was crucial for implementing system calls and handling interrupts, which require direct hardware interactions. By using the asm crate, I could perform these lower-level operations while keeping them safely contained within designated unsafe blocks of code. Therefore, the asm crate was essential for the kernel's advanced functionalities. Significant checks were made for all assembly created to ensure their safety and intention, however, this could potentially be an attack avenue.

5.2.6 Object

The object crate significantly streamlined the task of parsing and validating ELF files, which are essential for loading user programs. The crate provides access to the entry point of the ELF file, serving as the starting point for code execution. This feature was indispensable as it allowed me to seamlessly transfer control to the user program once its required resources—such as page tables, stacks, and heaps—were then allocated. Thus, the object crate not only simplified the handling of ELF files but also integrated smoothly with the memory management components of my kernel. It can be noted that the loading of the ELF could have been done manually, but this crate significantly abstracted this to the point of it not being a concern, another advantage of using Rust.

5.3 Memory Management

5.3.1 Memory Logger

The memory_logger.rs file plays an important role in the project by providing detailed insights into the memory management system of the kernel. It is built around the central MemoryLogger struct, which holds various fields to record key statistics such as total memory, used memory, and free memory, as well as counts of successful and failed allocations and deallocations. The logger is capable of capturing intricate details about memory regions, encapsulated in the MemoryRegion struct, which documents the start and end addresses of a given region.

Upon instantiation via its constructor, *MemoryLogger::new()*, all fields are initialised to default values, establishing a baseline for the logging activity. The logger offers a method, *setTotalMemory*, to set the total available memory, which is crucial for subsequent calculations and logging.

Two essential methods, *log_allocation* and *log_deallocation*, are responsible for logging the activities of memory allocation and deallocation, respectively. These methods accept MemoryRegion objects and sizes to update internal fields such as used_memory and free_memory. Additionally, they update counts of successful or failed operations. If an allocation is successful, the logger updates its peak memory usage if the new used_memory surpasses the previous peak. These functions are called many times during the runtime of the system, and may introduce some overhead to the system, however, adding some sort of global flag would be possible to turn the debugging on and off.

The logger employs my 'serial_println!' macro to print these statistics to the serial output, a feature that is invaluable for debugging and performance tuning. The logs can be parsed and analyzed to generate graphs or insights, thereby aiding in the evaluation and optimization of the memory management system. These logs contain information about the memory management from the moment the logger is initialised, which is near the beginning all the way to the end of the system.

Another method, log_stats, is designed to print or store the current memory usage statistics, providing a snapshot of the system's memory state at any given point in time. This is a useful method to call during exception handling to get an idea of whatm memory looks like at that exact moment.

5.3.2 VGA Memory

I explored the VGA text buffer's characteristics, using [69] as a resource. It is essentially a two-dimensional array mapped in memory, where each cell corresponds to a screen character and its associated metadata. The metadata includes the foreground and background colors, the ASCII character itself, and a blinking option. The VGA text buffer resides at a fixed memory-mapped I/O address: *0xb8000*. This mapping means that any write to this address would manipulate the VGA hardware directly, updating the screen contents. Interestingly, the VGA text buffer allows for straightforward memory operations, simplifying the task.

I started by defining an enum for the available VGA colors in Rusts, explicitly specifying the color values to match those supported by the VGA text buffer. I then created a ColorCode structure, encapsulating both the foreground and background colors into a single byte. The structure utilizes bit manipulation techniques to encode the color information correctly.

To write to the 2-dimensional buffer, I implemented a Writer struct, which maintains the current cursor position and color code. The struct also holds a mutable reference to the actual VGA buffer. I implemented methods like write_byte and write_string to handle single ASCII characters and strings, respectively. I also took care of line wrapping and newlines within these methods. Additionally, implementing the builtin formatting type for this Writer struct automatically means that special characters and types other than "String" can be written to the screen.

In conclusion, the VGA text buffer module provided a simple yet effective way to output information, logs and interact with the user. This module is used by a custom "println" macro, which is the standard way for Implementing this module in Rust also allowed me to benefit from strong type checks and memory safety features, thus ensuring robustness. As a result, I was not only able to print text to the screen but also lay the groundwork for more advanced features like error messages and user prompts.

5.3.3 Allocating memory

In the design of the memory management for this Rust-based kernel, multiple key components deserve attention. The core aspects include the MemoryInfo struct, the PageFrameAllocator struct, and the functions that manipulate these structures, such as *init*, *allocate_frame*, and *deallocate_frame*.

5.3.4 Recursive page tables

In the context of my custom kernel's memory management system, I opted for the use of recursive page tables as described in the previous background section. The choice of this approach is instrumental in simplifying

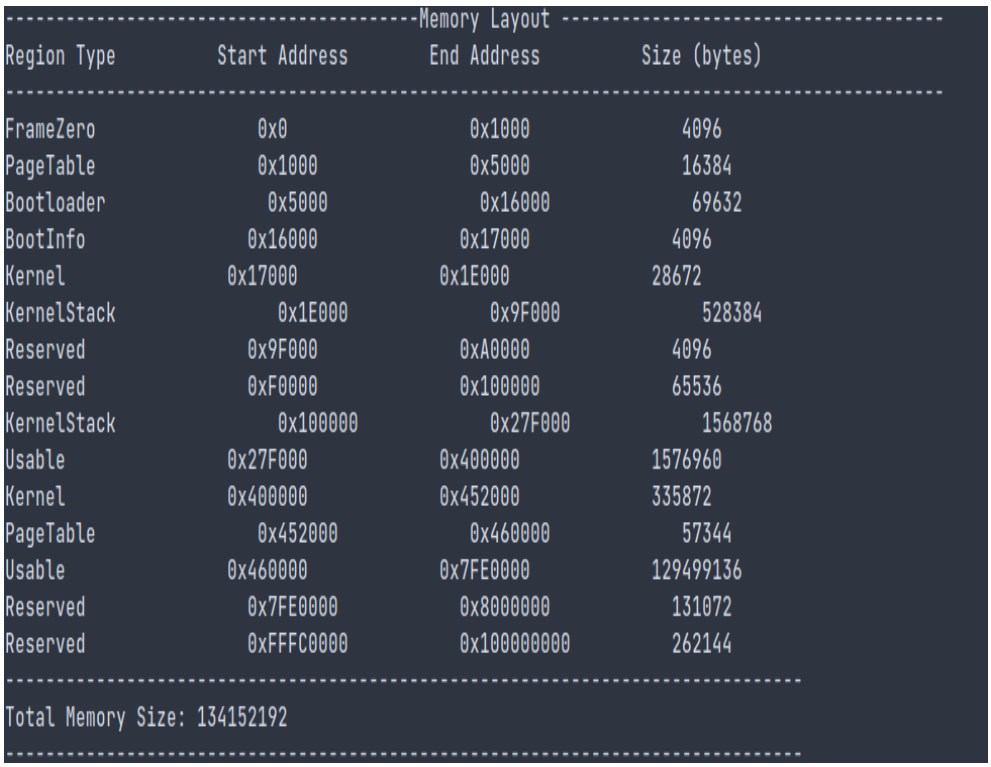


Figure 7: Memory Region types and ranges

the access to memory locations and their corresponding management. Upon system initialisation, the critical first step is to acquire the address of the highest-level page table, also known as the level 4 page table. This is possible through reading the CR3 register, which inherently points to this table. The physical and virtual addresses obtained are then fed back into the memory management system. This operation is conducted within a function named *active_level_4_table*, which itself is called inside of the memory module's *init* function.

It is important to note that this step is contingent on the prior mapping of all physical memory to their corresponding virtual addresses. The bootloader crate, incorporated into the system, takes responsibility for this mapping. When the bootloader transitions control to my kernel, it passes along vital information related to virtual memory. Specifically, this information is encapsulated within a structure termed BootInfo. Among the data fields of this structure is the offset for physical memory in the virtual memory space. By adding this offset to the initial physical address obtained from the CR3 register, we are able to accurately pinpoint the address of the level 4 page table.

Having gained access to this page table, the next critical step involves traversing through the hierarchical structure of page tables, thereby enabling the modification and access of memory at specific virtual addresses. To facilitate this operation, I have designed a suite of helper functions that simplify the traversal process. These functions not only make the system more developer-friendly but also allow for the comprehensive output of table contents and memory region types. Such output serves as a valuable diagnostic tool for understanding the memory layout within the system.

The function *allocate_pages_mapper* plays a pivotal role in the dynamic allocation and mapping of pages within the memory management system. It takes five parameters: a mutable reference to a FrameAllocator instance, a mutable reference to a Mapper instance, a starting virtual address (start_addr), the size of the memory to be allocated (size), and the flags that need to be associated with the pages (flags). This design allows the function to be highly adaptable and extensible, accommodating different allocator and mapper implementations while providing fine-grained control over the allocation process.

The function first calculates the range of pages that need to be allocated. To accomplish this, it utilizes the *Page::containing_address* method to identify the page corresponding to the starting and ending virtual addresses. Once these pages are identified, the function generates an inclusive range of pages to allocate. This range calculation is a crucial step, as it dictates the amount of physical memory that will be mapped to the virtual address space.

Subsequently, the function enters a loop to allocate and map each individual page in the calculated range. Within this loop, the FrameAllocator is invoked to allocate a new frame of memory. Should the frame allocation fail, an error of type *MapToError::FrameAllocationFailed* is returned, providing diagnostic information that aids

in pinpointing issues related to frame allocation. Assuming successful allocation, the function then proceeds to perform the actual mapping from the virtual page to the physical frame. This is done using the *Mapper::map_to* method, which is called in an unsafe block due to its potential to violate memory safety guarantees if misused.

In addition to *allocate_pages_mapper*, another function named *allocate_pages* exists, designed to work specifically with a given level 4 page table. This function acts as a wrapper, facilitating the usage of a specific level 4 table during page allocation. It accesses global memory information and creates an *OffsetPageTable* mapper, which is then passed along with the frame allocator to *allocate_pages_mapper*. By doing this, *allocate_pages* adds an additional layer of abstraction, allowing the system to work seamlessly with any level 4 page table. The *OffsetPageTable* type comes from the x86 crate and provides the functionality for the physical mapping of memory with the *map_to* function.

The counterpart to the allocation functions in the memory management system is the *deallocate_page* function. This function serves the critical role of deallocating a specified range of pages, effectively freeing up physical memory that was previously allocated. It takes three parameters: a mutable reference to a Mapper instance, the starting virtual address (address), and the size of the memory region to be deallocated (size). Similar to its allocation counterpart, the function begins by calculating the inclusive range of pages that need to be deallocated. This is achieved through the *Page::containing_address* method, just as in the allocation functions. The function then iterates over this range, invoking the *Mapper::unmap* method to unmap each virtual page from its corresponding physical frame. Successful unmapping leads to flushing the mapping to ensure the changes take effect immediately. However, if the unmapping process fails, an error message is output to the serial console for diagnostic purposes.

The frame allocator is not just limited to memory allocation functions; it also includes utilities for translating between virtual and physical addresses. Leveraging Rust's expressive syntax, it becomes straightforward to locate the next available 'free' frame, thanks to the utility of Rust's nth() function.

However, just access and traversal do not constitute a full-fledged memory management solution. To enable dynamic allocation and deallocation of memory, it was essential to implement a *FrameAllocator* [65], encapsulated in the PageFrameAllocator struct. This allocator is specifically engineered to manage physical frames of memory dynamically. One of its key helper functions iterates over the memory regions, marking each frame as either 'free' or 'not-free' within a boolean array labeled *frame_usage*. Although this design approach streamlines allocation and deallocation operations, it also imposes a static limit on the number of frames that can be managed, restricted by the size of the *frame_usage* array.

The *allocate_frame* function within the *PageFrameAllocator* struct serves a specialised role in the memory management system—allocating individual physical frames of memory. Unlike the *allocate_pages_mapper* function, which focuses on mapping a range of virtual pages to physical frames, *allocate_frame* is concerned solely with the allocation of physical frames. It operates within the *FrameAllocator* trait, taking no arguments and returning an *Option¡PhysFrame¿* to indicate the success or failure of the allocation operation.

The function begins by looking for the next usable frame in the memory map, starting from the index pointed to by the next field. It employs the *usable_frames().nth(self.next)* method to fetch the nth usable frame. Once a candidate frame is identified, its availability is verified using the *is_frame_free(frame)* method. If the frame is free, it is marked as used by invoking *mark_frame_as_used(frame)*, and the next index is incremented to prepare for the next allocation.

In case no free frames are available, the function outputs an error message and returns None, signaling the failure of the allocation operation, this error is then handled by the calling method. The contrast between *allocate_frame* and *allocate_pages_mapper* lies in their scope and purpose: while the former is dedicated to the allocation of physical frames, the latter is geared toward the mapping of virtual pages to these frames, thereby providing a more complete memory management solution.

While these functionalities provide a foundation for memory management, they are not exhaustive. The system will require additional features, such as the ability to allocate a user stack and to switch between kernel and user-level page tables. These aspects, however, are more suited for discussion within the context of process management, which constitutes another critical part of this kernel.

In summary, the high-level memory management system of my system, underpinned by the PageFrameAllocator and a series of helper functions, offers a combination of flexibility, efficiency, and diagnostic capability. However, its design is extensible, able to accommodate further complexities as the kernel matures.

5.3.5 Heap Allocation

The heap memory management system is initialised through the *init_heap* function, which takes mutable references to a *Mapper* and a *FrameAllocator*. This function aims to create a contiguous block of virtual memory pages mapped to physical frames, which will act as the heap.

The heap's starting address and size are predetermined constants, denoted as *HEAP_START* and *HEAP_SIZE*. These constants define the virtual address space that will be reserved for the heaps.

The *init_heap* function starts by calculating the range of virtual pages required for the heap. It employs *VirtAddr::new* to create a *VirtAddr* object, specifying the heap's starting address. This object is then used to calculate the virtual address of the heap's end point, ensuring that the range is inclusive. The *Page::containing_address* function is invoked to convert these virtual addresses into Page types. Subsequently, *Page::range_inclusive* creates a range of pages between these start and end points.

Upon establishing the range of virtual pages, the function iterates over this range to map each virtual page to a physical frame. The *FrameAllocator* allocates a physical frame, and the *MapToError::FrameAllocationFailed* error is returned if the allocation fails. The page table flags PRESENT and WRITABLE are set to indicate that the page is both present in physical memory and writable.

The *map_to* function, which is part of the *Mapper* trait, is used to establish the mapping from virtual pages to physical frames. This operation is performed within an unsafe block, as it directly manipulates the hardware's page tables. The flush method is invoked to ensure that these changes are written immediately.

Finally, the heap is initialized by calling the *init* method on the *ALLOCATOR*, a global static instance of *FixedSizeBlockAllocator* wrapped in a locked state. The allocator is initialised with the heap's starting address and size. This operation is done within an unsafe block due to the direct manipulation of memory that may affect other parts of the system.

The *FixedSizeBlockAllocator* serves as a custom memory allocator designed for fixed-size block allocations. The primary advantage of using fixed-size blocks is that it can significantly reduce memory fragmentation, thus enhancing the efficiency of memory utilisation. When an allocation of bytes is needed, it will round those up to the nearest block size and allocate that amount of memory. This allocator is implemented as a struct containing an array of list heads, one for each predefined block size, and a fallback allocator. Essentially, rather than just one linked list, we have a list of linked list pointers. These pointers point to an unused block of memory for its corresponding block size. This is further described in the background section.

The predefined block sizes, stored in the constant array *BLOCK_SIZES*, are set as powers of two. This design choice ensures better alignment and more efficient memory usage. Each element in the *list_heads* array is an *Optional* reference to a ListNode, which in turn may reference the next node in a linked list. These linked lists serve as free lists for each block size, effectively managing available blocks of memory.

The allocator also contains a *fallback_allocator* of type *linked_list_allocator::Heap*, which is used for block sizes that do not align with any of the predefined sizes. This provides a general-purpose allocation mechanism for unusual requirements.

The *fallback_allocation* function is a helper method to allocate memory using the fallback allocator. If the allocation fails, it returns a *None*, signaling the failure and this is handled in the calling function, if it is not handled then there will be a compile time error, thus enforcing correct handling.

The core logic for memory allocation and deallocation is implemented through the *GlobalAlloc* trait. The *alloc* function checks whether the block size required by the Layout fits any of the predefined sizes. If it does, it searches the corresponding free list for an available block. If no block is available or if the size does not match any predefined sizes, it falls back to the *fallback_allocator*.

The *dealloc* function first rounds up to the nearest block size the amount of memory that needs to be freed and then uses *find_block_index* to find the appropriate free list for the block being deallocated. It then adds this block back to the free list, making it available for future allocations. If the block size is not predefined, it uses the *fallback_allocator* for deallocation.

With the setup of the heap, all kernel programs can use data structures that are dynamic, such as *Box* and *Vec*. There are several other designs for allocators, however after trying to implement them I came across several issues and inconsistencies. Notably, deallocating with other allocators just did not work, and after testing I would quickly fill the heap space and fault, this will be due to implementation errors and not the language itself. Choosing the fixed block approach was simple enough but also efficient, striking a good balance.

5.4 Hardware interface and interrupts

Interrupts and interrupt service routines are essential for a functional kernel. The x86_64 crate provides an *InterruptDescriuptorTable* struct, which contains all of the necessary registers and defines which exceptions need handler functions, causing compile time errors when a function was missing. Rust provides support for writing a function that takes an x86 stack frame, which allows us to switch to this state and handle the exception. Before handling the exception however, we must save the current state of the processor so that when we return from the exception handling we can load the state of the system before the exception. It saves this state by storing the stack pointer and instruction pointer, then reads the address of the interrupt handler defined by the IDT. This is described more in the background section.

For the kernel to know which interrupt to call from within the IDT, the system must tell it where the IDT is. Luckily, the struct provided by the x86 crate provides a *load* method, but it does mean that you have to define the *IDT* with a static lifetime, which means that it exists for the entirety of the kernel. Otherwise, as

soon as it has gone out scope in the stack, the system would not be able to access it. We can use a lazy static type again, which is it's own crate, that provides mutex access to this static IDT reference.

The *x86* crate also provides an *ISF* struct that we can populate when an exception occurs. Creating handler functions meant that I could output any debug messages whenever one of these exceptions occurs. There is a timer interrupt handler, breakpoint handler, double fault handler, page fault handler (which is particularly important) and a general protection fault handler, as well as divide by a zero handler and others . Most of these just output a debug message and use log the state of the stack out. However, all memory related ones terminate the thread and call a function to drop it's page tables, the stack space itself goes out of scope when it is dropped, so this memory is freed.

I created an ISF (interupt stack frame) struct that represents the CPU when an interrupt occurs. Which is used extensivley when a switch is needed, this contains more data than the crate's version, which is necessary for being able to return back to the complete state. The *launch_thread* function uses assembly to switch CPU context to a new thread. Inline assembly is used to manually set CPU registers and stack pointers, essentially launching the new thread. The *interrupt_wrap* macro serves to generate a wrapper function for each interrupt handler, ensuring CPU state is saved and restored correctly, this was inspired by the Moros OS, which uses the kind of wrapper macro in several places to make switching threads far easier. It does this by first disabling interrupts using the *cli* instruction, then pushes all general purpose registers on to the stack, it then calls the handler via a *call* instruction. When this returns, it restores all the stored registers to their original state and calls the *iretq* instruction to return from the interrupt to the interrupted thread. The *interrupt_wrap* macro automates the process of saving and restoring the CPU state, thereby allowing the timer interrupt handler to focus solely on the logic needed to handle the timer event. This makes the code cleaner and less error-prone. I will discuss more in the process management section about the timer handler, as it plays a key role in the scheduling of threads.

The Task State Segment (TSS) and Interrupt Stack Table (IST) are constructs for handling task-switching and interrupt delivery, respectively. These components play pivotal roles in ensuring both the integrity and efficiency of interrupt handling—a fundamental requirement for any modern operating system.

The TSS, or Task State Segment, is a special data structure that the CPU consults during task switches and when interrupts or exceptions occur. Historically, the TSS contained a plethora of information needed for task switching. However, in 64-bit mode, its role has been significantly pared down, yet it remains essential for specifying interrupt stacks. In the context of this project, the TSS is used to define different stacks for different types of interrupts to switch to. For instance, a double fault, a severe condition where an exception occurs while trying to call an exception handler, is configured to use a different stack. This is crucial because a double fault typically occurs when the original stack is in an inconsistent or unusable state. Having a seperate stack for each interrupt makes returning back to the original state more explicit and also prevents any injection of values into a privileged interrupt handler.

Complementing the TSS is the Interrupt Stack Table (IST), a feature that allows each entry in the Interrupt Descriptor Table (IDT) to specify its own unique stack. When an interrupt or exception occurs, the CPU can automatically switches to the stack defined in the IST, thus isolating the interrupt context from the user or kernel stack. This isolation is particularly beneficial for handling nested or concurrent interrupts efficiently.

IDT structure is populated with various interrupt handlers, each specified to use a particular stack from the IST. For instance, the double fault handler is configured with set_stack_index(gdt::DOUBLE_FAULT_IST_INDEX), ensuring that it uses a dedicated stack.

In summary, while both TSS and IST deal with specifying stacks for handling interrupts and exceptions, the TSS offers a more general mechanism for specifying which types of interrupts should use which stacks. IST, meanwhile, provides a finer level of control by allowing unique stacks for each individual interrupt or exception. Both mechanisms aim to improve the system's reliability and robustness by ensuring that interrupts and exceptions are handled in an isolated manner, but they operate at different levels of granularity.

5.4.1 Hardware support with PICs

The capability to accurately and efficiently handle hardware interrupts is indispensable for system robustness and functionality. A important element in achieving this objective is the Programmable Interrupt Controller (PIC), a hardware device that serves as an intermediary between various hardware components and the Central Processing Unit (CPU). The PIC's role is to route hardware interrupts, such as those generated by a keyboard or a mouse, to the appropriate interrupt handler routines executed by the CPU.

An implementation facilitating this is provided in the *PIC8259* crate, which offers a two-level hierarchy of PICs. This architecture essentially extends the number of hardware interrupt lines by abstracting a single interrupt line into an additional eight interrupt lines, thereby increasing the system's capability to handle a broader array of individual hardware interrupts. This two-level PIC architecture, also commonly known as "cascading," allows for greater system extensibility and is particularly beneficial in complex systems where the

number of hardware components that can generate interrupts exceeds the available interrupt lines on a single PIC.

In this specific implementation, the Programmable Interrupt Controller (PIC) is managed through the *ChainedPics* struct from the *pic8259* crate. The chained PICs are statically instantiated as a mutex-protected global object, using *spin::Mutex* for synchronisation. The name chained is in reference to the two-level hierarchy. This mutex protection ensures that access to the PICs is serialized, thus preventing race conditions in a multi-threaded environment. The offsets for the first and second PICs are set to *PIC_1_OFFSET* and *PIC_2_OFFSET*, respectively.

The hardware interrupts are mapped to their corresponding indices in the Interrupt Descriptor Table (IDT) through an enumeration, InterruptIndex. Each hardware interrupt, such as Timer and Keyboard, is assigned a unique index based on its PIC offset. In the timer_handler and keyboard_handler_inner functions, the end of the interrupt is signaled to the PIC with the *notify_end_of_interrupt* method. This step is essential for acknowledging the interrupt, allowing the PIC to queue further interrupts. Overall, this implementation leverages Rust's type safety and concurrency primitives to manage the PIC efficiently and safely.

5.5 Process Management

When it comes to thread scheduling, each thread in a system will have its own stack. As the scheduler transitions control from one thread to another, the stack pointer must be updated to point to the new thread's stack. While this operation can be manually performed by saving and restoring stack pointers during the context switch, using the TSS to manage separate stacks in the system is sused to manage separate stacks for each thread. When a thread context switch occurs, the scheduler updates the TSS with the new thread's stack pointer, streamlining the process of stack management. This removes the need for manually saving and restoring stack pointers during the context switch, enhancing both efficiency and safety.

Now that the system has a way to save and load the state of the CPU it is possible to schedule threads. Firstly though, I needed to create a Thread object. Below, is the struct that defines a Thread.

```
pub struct Thread {
    /// A unique identifier for the thread. This ID is typically assigned sequentially
    /// and can be used to differentiate threads from one another.
    thread_id: u64,

    /// A reference to the parent process of the thread. This provides context for
    /// the thread's execution and access to shared resources with other threads of
    /// the same process.
    process: Arc<RwLock<Process>>,

    /// The memory region reserved for the thread's kernel stack. The kernel stack
    /// is used for operations that occur in kernel mode, separate from the user stack.
    /// Used to store the Context of a thread on switch
    kernel_stack: Vec<u8>,

    /// The end address of the kernel stack. Given that stacks grow downwards in memory,
    /// this represents the top of the stack, and it's where new items would be pushed.
    /// Actual address that is placed in the TSS
    kernel_stack_end: u64,

    /// The memory address where the thread's context is stored. The context includes
    /// information like register values which are necessary to resume the thread's
    /// execution after it has been paused (e.g., during a context switch).
    context: u64,

    /// The end address of the user stack. Similar to the kernel stack end, but for
    /// operations that the thread performs in user mode. New items are pushed onto the
    /// user stack at this address.
    user_stack_end: u64,

    /// A pointer to the physical memory location of the page table that manages memory
    /// translations for this thread. This is essential for virtual memory and ensuring
    /// the thread accesses the right memory locations.
    page_table_phys: u64,
```

```
    /// Indicates the type of the thread (e.g., Kernel or User). This can be used to
    /// determine the privileges and operations the thread is allowed to perform.
    thread_type: ThreadType,
}
```

On a single core system, only a single process can run at any one time. Having a reference to this running thread, called *RUNNING_THREAD*, allows functions to be able to either change it or monitor, that is why I have a static reference to a running thread which can be accessed kernel wide. To be able to switch between different threads the system needs a data structure to hold a list of every thread that isn't the running thread, or another way of describing it is all threads that are waiting to be the running thread. So the system also implements a circular queue called *WAIT_QUEUE*. Both of these system variables are wrapped in a spin lock and lazy static, meaning that only one thread can access a reference to these at a time, removing the risk of data races.

The system supports spawning two types of threads, a kernel thread and a user thread. Spawning a kernel thread is slightly simpler so I will describe that first. The function *spawn_kernel_thread* serves as a critical component for thread management. This function essentially bootstraps a new kernel thread, setting up its initial context, and inserting it into the scheduling queue for eventual execution. The function's operations can be split into multiple discrete yet interconnected steps, each contributing to the overarching goal of thread instantiation and management.

5.5.1 Kernel Threads

Firstly, the function creates a new Thread object. Within the confines of this object, two separate stacks, a kernel stack and a user stack, are allocated within the same memory region. This is a strategic decision, aimed at encapsulating the stacks within a single memory region for easier management. The starting and ending addresses of these stacks are subsequently stored for later use.

```
// Allocate kernel stack and user stack within the same memory region
let kernel_stack = Vec::with_capacity(KERNEL_STACK_SIZE + USER_STACK_SIZE);
let kernel_stack_start = VirtAddr::from_ptr(kernel_stack.as_ptr());
let kernel_stack_end = (kernel_stack_start + KERNEL_STACK_SIZE).as_u64();
let user_stack_end = kernel_stack_end + (USER_STACK_SIZE as u64);
```

Next, the function generates a unique identifier for the thread using the *unique_id* function. This ID serves as the primary key for thread identification within the system, thereby enabling more effective thread management and debugging. It is important that these are not duplicated, so the the counter that generates the unique ID is also wrapped in a mutex lock.

The Thread object is also populated with a Process object that contains the physical address of the page table and a list of sockets associated with the thread. These sockets provide inter-thread communication capabilities, essential for a multitasking environment, which will be described in more detail soon. The thread is also placed inside of a Box so that they can be handled within the wait queue in neater fashion.

```
// Initialize the thread object
Box::new(Thread {
    thread_id: uid,
    process: Arc::new(RwLock::new(Process {
        page_table_physaddr: 0,
        sockets: sockets.drain(..).map(|h| Some(h)).collect(),
    })),
    kernel_stack,
    kernel_stack_end,
    context: kernel_stack_end - INTERRUPT_CONTEXT_SIZE as u64,
    user_stack_end,
    page_table_phys: 0,
    thread_type: ThreadType::Kernel,
})
```

Following the initialisation of the Thread object, the method proceeds to set the thread's initial state by modifying its Interrupt Stack Frame (ISF). The instruction pointer is set to point to the function that the thread will execute upon its turn in the CPU, this is passed as an argument to the method. This essentially provides

the thread with its purpose or task within the system.

```
// Set the instruction pointer to the function to be executed by the thread
// provides the thread with a function to run
context.instruction_pointer = function as usize;
```

The processor flags are set next, with a specific flag enabling hardware interrupts. This is an essential step, as enabling interrupts is crucial for the scheduler to preempt the thread when required. The segment selectors for the code and data segments are also set at this point, aligning with the current Global Descriptor Table (GDT) settings. This was a major issue during development as these flags were not set, meaning that the timer interrupt could not interrupt the thread while running, thus not enabling switching to other threads.

```
// Set processor flags; 0x200 enables interrupts
context.flags = 0x200;

// Set segment selectors for code and data
let (code_selector, data_selector) = gdt::get_kernel_segments();
context.code_segment = code_selector.0 as usize;
context.stack_segment = data_selector.0 as usize;
```

The stack pointer for the thread is then set to the end of the user stack, preparing it for its first execution cycle. At this point, the function also employs debugging and monitoring functionalities to keep track of the thread's state, essential for system maintenance and troubleshooting.

```
// Monitor the new thread (for debugging or logging)
monitor(&new_thread);

// Print thread details (for debugging or logging)
new_thread.print_details();
```

Finally, the newly minted thread is inserted into the scheduling queue by invoking *schedule_thread*, marking it as ready for execution, all this function does it place the newly formed thread to the front of the wait queue. It must do this while interrupts are turned off. The function concludes by returning the unique thread ID, which can be used for further reference or management operations.

5.5.2 Scheduling

Before discussing user threads, I implemented the scheduler following the implementation of the kernel threads, so that I could test whether thread creation in general and scheduling would work. This meant that I could create kernel functions to be used in threads.

The *schedule* function serves as the crux of my thread scheduling logic, invoked by the timer interrupt handler to perform a context switch between threads. Its primary objective is to relinquish the CPU from the currently running thread and allocate it to the next thread in line, effectively implementing a round-robin scheduling algorithm. The function accepts a single argument, *context_addr*, which represents the address of the saved context of the currently executing thread. It returns the address of the saved context of the subsequent thread to be executed.

Upon invocation, the function first obtains write locks on the *WAITING_QUEUE* and the *RUNNING_THREAD* global variables. The function then invokes *update_current_thread* to modify the saved context of the currently running thread and moves it to the end of the *WAITING_QUEUE*. This operation ensures that each thread gets its fair share of CPU time in a round-robin manner.

```
// Lock the running queue and the current thread for writing.
let mut waiting_queue = WAIT_QUEUE.write();
let mut running_thread = RUNNING_THREAD.write();

// Update the current thread and move it to the back of waiting queue.
update_current_thread(context_addr, &mut waiting_queue, &mut running_thread);

// Pop the next thread from the front of the running queue.
```

```
*running_thread = waiting_queue.pop_front();
```

Subsequently, the function dequeues the next thread from the *WAITING_QUEUE* and updates the *RUNNING_THREAD* to point to it. Before transferring control, the function performs crucial preparatory steps. If the thread to be scheduled is not a kernel thread (denoted by a non-zero *page_table_phys* field), the function switches to that thread's page table using *memory::switch_to_pagetable*. Additionally, the interrupt stack table is updated via the *gdt::set_interrupt_stack_table* function to use the new thread's kernel stack, thus ensuring that any future interrupts will be handled in the context of the newly scheduled thread.

```
match running_thread.as_ref() {
    Some(thread) => {
        // Set the interrupt stack (kernel stack) for the next scheduled thread.
        gdt::set_interrupt_stack_table(gdt::TIMER_INTERRUPT_INDEX as usize,
            VirtAddr::new(thread.kernel_stack_end)); // context of next thread

        // If this isn't a kernel thread, switch to its page table.
        if thread.page_table_phys != 0 {
            memory::switch_to_pagetable(thread.page_table_phys);
        }

        // Return the saved context address for the next thread.
        thread.context as usize
    },
    None => 0  // If there's no thread to schedule, return 0. i.e, before the first
    // thread is spawned but there is still a timer interrupt
}
```

In summary, the schedule function is instrumental in orchestrating the CPU's multitasking capabilities. It fairly updates the context of the currently running thread, dequeues the next waiting thread, and sets it up for execution, thereby achieving a seamless transition and equitable distribution of CPU resources. If there are no threads to be scheduled— a condition possible before the first thread has been spawned but a timer interrupt occurs— the function returns zero. Further improvements to this could be made, including adding a priority field to the Thread struct and using priority and round robin scheduling.

5.5.3 User threads

The function *spawn_user_thread* allows the kernel to create threads in user space by incorporating three critical operations: setting up the thread's memory space, parsing the provided ELF (Executable and Linkable Format) binary, and initializing the thread's execution context.

The function begins by validating the supplied binary against the ELF format's specifications. An ELF file is expected to start with specific magic bytes, and the *validate_binary* function checks for this conformance. If the binary is not an ELF file, the function returns an error, thereby ensuring that only compatible binaries are processed further.

Once the binary's legitimacy is decided, it undergoes parsing through the *parse_elf_binary* function. This step is vital as it deciphers the structure of the binary, which includes segment information, entry points, and other crucial metadata. If the parsing fails, an error is returned, aborting the thread creation process.

```
// Validate the binary format
if !validate_binary(bin_file) {
    return Err("Invalid binary format");
}

   // Parse the binary using object crate
let obj_result = parse_elf_binary(bin_file);
if obj_result.is_err() {
    return Err("Failed to parse ELF");
}
let obj = obj_result.unwrap();
```

Subsequent to the parsing phase, the function proceeds to create a new page table for the user thread. This page table is initialized to include only kernel pages, in line with the principle of least privilege. The physical address of this new page table is stored for future reference.

Following this, heap memory is allocated for the user thread. This allocation is pivotal for dynamic memory requirements that may arise during the thread's execution.

The function then switches to the newly created user page table. The function *load_binary_into_memory* loads an ELF (Executable and Linkable Format) binary into a user thread's memory space. It takes three arguments: the ELF file to be loaded, the physical address of the user's page table, and a pointer to the user's page table. The function iterates over each segment in the ELF binary, validates the segment's memory range to ensure it falls within the prescribed user code range, and reserves corresponding memory in the user's page table. Memory reservation is performed with specific *PageTableFlags* that mark the memory as present, writable, and user-accessible. Once the memory is successfully allocated, the function switches to the user's page table and copies the segment data into the allocated memory.

The loading process is meticulous in its safety checks: it validates the segment data size to prevent buffer overflows and ensures that memory allocation succeeds before proceeding. In case of any anomalies, such as segment data exceeding the allocated size, failure to allocate memory, or segment addresses falling outside the permissible range, the function returns an error. Several of these helper methods are provided by the *Object* crate one of the most important features of using this crate is the generation of a reference to the entry point to the binary file.

Following the successful loading of the binary, the thread's execution context is initialized via the *initialise_thread* function, which performs almost identical operations as spawning a kernel thread. This function sets the instruction pointer to the entry point of the ELF object, aligns the stack, and performs other necessary initialisations, and adds it to the waiting queue along with any other threads. Upon successful completion, a unique thread ID is generated and returned.

```
// Switch to the user pagetable and setup the memory segments
// based on the parsed ELF object.
return switch_pagetable_and_execute(user_page_table_physaddr, || {
    load_binary_into_memory(&obj, user_page_table_physaddr, user_page_table_ptr)?;

    let new_thread_id = initialise_thread(obj.entry(),
                user_page_table_physaddr, user_page_table_ptr, params);
    match new_thread_id {
            Ok(new_id) => {
                // Return the thread ID as Result<u64, &str>
                Ok(new_id)
            },
            Err(e) => {
                // Forward the error
                Err(e)
            }
        }
}).map_err(|e| "Error creating user thread");
```

If any error occurs during the thread's creation, it is appropriately forwarded, and the function terminates with a failure state. The transition from kernel mode to user mode in my system involves a series of steps, especially focused on the manipulation of the Global Descriptor Table (GDT). Initially, the GDT is populated with kernel-specific entries for code and data segments. Additionally, I modified the GDT with entries for user code and user data segments, thereby segregating the memory space and access permissions for kernel and user-level operations. The *Selector* struct is updated to include these new user-specific selectors, which essentially encapsulate the offsets into the GDT for these segments. Specifically, the value in the Code Segment (CS) register governs the privilege level, commonly referred to as the "ring level". This is what will be checked for each memory access, if a user thread attempts to write to a kernel thread, it will cause a page fault, this is tested and shown in the evaluation section.

The function *free_user_stack* is designed to deallocate a user thread stack, given the virtual address of the stack's end. This function is crucial for reclaiming memory resources when a user-mode thread terminates and is particularly important for a system that wants to acheive memory safety. The function performs several key operations to achieve this. It first calculates the last page address of the stack by subtracting 1 from the given end address, stored in the variable *addr*. It then accesses the current level 1 page table containing this address via the function *active_level_1_table_containing*. The function retrieves the global memory information, encapsulated in the *MEMORY_INFO object*, to interact with the frame allocator. It calculates the range of indices in the page table that correspond to the stack pages. The index *iend* is obtained from the last page address. Finally, it iterates through the page table entries that correspond to the stack pages. For each entry, it checks if the page is writable (hence uniquely allocated for this stack). If it is, the frame is deallocated and

given back to the boolean list of available memory. Regardless, the page table entry is then cleared.

The *Drop* trait is implemented for the Thread struct to ensure that when a thread object goes out of scope or is explicitly dropped, its associated user-mode stack is deallocated. Implemented for a trait is a feature of Rust that is similar to using an object's interface. The drop method of this trait calls the *free_user_stack* function, passing the virtual address of the stack's end, which is stored in the Thread object as *self.user_stack_end*.

```
impl Drop for Thread {
    fn drop(&mut self) {
        memory::free_user_stack(VirtAddr::new(self.user_stack_end));

        monitor(&self);
        println!("[!] - Thread {} deallocated user stack at {:#x}", self.thread_id,
                        self.user_stack_end);
        serial_println!("[!] - Thread {} deallocated user stack at {:#x}",
                        self.thread_id, self.user_stack_end);
    } // thread is now out of scope and is 'freed'
}
```

5.6 System calls

In the architecture of an operating system, system calls serve as a critical interface that enables user-level processes to request services provided by the kernel. These services range from file operations to process control and inter-process communication. System calls act as a controlled gateway, facilitating the safe and secure transition from user mode to kernel mode, thus preserving the system's integrity and stability. They are quintessential for achieving the abstraction and isolation properties that are fundamental to a multitasking, multi-user environment.

When a user-level application necessitates a resource or a service that is managed by the kernel, it cannot directly access it due to the privilege level restrictions imposed by modern CPUs. Direct access to hardware resources or sensitive data structures by user-level processes could result in system instability or security vulnerabilities. System calls provide a structured and secure mechanism for these processes to request and utilize kernel-level services. They do so by triggering a software interrupt, which prompts the CPU to switch from user mode to kernel mode and transfer control to a predefined location in the kernel code. The kernel then validates the request, performs the necessary operations, and returns control to the user-level process, often via another context switch.

In essence, system calls are indispensable for encapsulating the complexities and critical operations of the kernel, exposing only a set of well-defined interfaces to user-level applications. They provide the means for controlled interaction between user space and kernel space, ensuring that all operations adhere to the principles of security and system integrity. Thus, system calls are not merely a feature but a necessity in any modern operating system kernel, underpinning the core functionalities and ensuring a harmonious coexistence of multiple applications and services.

To implement system calls, I first set up the necessary infrastructure to enable and handle system calls. I created a new file *syscalls.rs* where I defined some constants related to *Model Specific Registers (MSRs)*, which are used for syscall control [53]. The *init* function initialises these MSRs and prepares the CPU for system call handling, essentially allowing the *syscall* instruction to be called, and directing it to the handler. Model Specific Registers (MSRs) are special-purpose registers that are used for enabling or configuring low-level CPU features. These are not general-purpose registers like EAX, EBX, etc., but they are specifically designed to control certain CPU features or to get status from the CPU. Accessing these MSRs is usually done through the RDMSR and WRMSR assembly instructions, which read and write to these registers respectively. Various changes to these registers enable system calls, and the *syscall* instruction.

Upon the invocation of the *syscall* instruction, the *handle_syscall* function undertakes multiple responsibilities. In a typical x86-64 operating system, the GS base register is used to point to a per-CPU or per-thread data structure when executing in kernel mode. This data structure often contains information that is critical to the currently executing thread or CPU, such as pointers to the kernel stack or other processor-specific data. When a *syscall* is invoked, the operating system typically needs to switch from using the user-mode GS base to the kernel-mode GS base, because the data that the kernel needs is different from the data that the user-level code uses.

The *SWAPGS* instruction facilitates this switch in an efficient manner and is important for my implementation. Prior to entering kernel mode, the kernel-mode GS base is stored in the hidden register. When a *syscall* is made, executing *SWAPGS* swaps the user-mode GS base with the kernel-mode GS base, thus setting up the GS segment for kernel-mode execution. The same instruction can be used again to swap back to the user-mode GS

base when returning from the *syscall*, making the transition seamless. Following this assembly instruction the *handle_syscall* function saves the context of the CPU, which involves pushing the current state of all relevant registers onto the stack. This step is important because the execution of the system call may modify these registers, necessitating their restoration prior to returning control to user-space.

Following context saving, the function switches from the user stack to a kernel stack. This stack transition serves a dual purpose: first, it enhances system security by preventing any potential stack overflow vulnerabilities that could otherwise compromise the kernel; second, it utilises the kernel stack's generally larger size to accommodate more extensive data or computational requirements. The stack switching is orchestrated through precise assembly instructions that manipulate the relevant segment registers and pointers.

After these preparatory steps, the *handle_syscall* function delegates the actual system call execution to another Rust function, termed *dispatch_syscall*. This function contains the logic for the identification and execution of the specific system call requested, based on the system call number passed in. Essentially, *dispatch_syscall* acts as a dispatcher that routes the system call to its appropriate handler function, such as read, write, or any other kernel service.

Subsequent to the execution of the system call, the *handle_syscall* function resumes its role to restore the original context. It pops the saved state of all registers from the stack and prepares to return to user-space. The function accomplishes this return through the *sysretq* assembly instruction, which transitions the CPU back to user-mode and resumes user-space execution.

In *dispatch_syscall*, I use the *syscall ID* to identify which system call is being made. For example, if the *syscall ID* is 2, it calls *sys_write*, a function that writes a string to the console. The other syscalls like *sys_receive*, *sys_send*, and *sys_yield* are implemented in a similar manner. They serve different purposes like inter-process communication and yielding the CPU to other processes.

```
// Dispatch to the appropriate syscall handler based on the syscall ID.
// The syscall ID is masked with 0xFF to get the actual ID value.
match syscall_id & 0xFF {
    0 => process::fork_current_thread(context),  // Fork the current thread
    1 => process::exit_current_thread(context),  // Exit the current thread
    2 => sys_write(arg1 as *const u8, arg2 as usize),  // Write to a socket
    3 => sys_receive(context_ptr, arg1),  // Receive a message from a socket
    4 => sys_send(context_ptr, syscall_id, arg1, arg2, arg3),  // Send a message
    5 => sys_send(context_ptr, syscall_id, arg1, arg2, arg3),  // Send and recieve
    9 => sys_yield(context_ptr),  // Yield the processor
    _ => println!("Unknown syscall {:?} {} {} {}", context_ptr, syscall_id, arg1, arg2)
// Unknown syscall
}
```

The syscall handlers often interact with the scheduler and might even switch the currently running thread based on the system call's semantics. For example, *sys_send* and *sys_receive* perform inter-process communication, and based on the message passing, they may reschedule threads. I will go in to more detail in the next section on how this works.

This setup allows user-level processes to interact with the kernel securely, requesting services like writing to the console or performing inter-process communication, all while maintaining system integrity and security. For a user thread to spawn another user thread we can call the syscall instruction after placing the appropriate syscall ID into the correct registers.

In the context of memory management for user threads, the capability to deallocate individual stacks is a prerequisite for effective resource utilisation. However, when the execution of all threads within a given process reaches completion, it becomes important to also deallocate the associated page tables to fully reclaim memory resources. The architecture of these page tables is inherently recursive, a characteristic that the system capitalizes upon to facilitate their deallocation.

To accomplish this, the system iterates through each entry in the page table, deallocating the corresponding frame in physical memory and thereby releasing these resources back into the system's available pool. This recursive traversal of page table entries ensures a comprehensive deallocation of the page tables associated with the terminated process. And was also a source of difficulty during development, as debugging this recursive process was especially difficult.

There exists a critical prerequisite to this operation: the system must transition to the kernel's page table prior to initiating the deallocation of a thread's page table. The failure to do so—i.e., attempting to remove a thread's own page table while within the context of that thread—would precipitate system failures. This preemptive switch to the kernel page table is, therefore, a crucial step in ensuring the integrity and stability of

the system during the memory deallocation process. This error was one that I tackled for a large amount of time.

```
// When a Process is dropped (no longer in use), ensure the associated memory structures a
impl Drop for Process {
    fn drop(&mut self) {
        if self.page_table_physaddr == memory::active_pagetable_physaddr() {
            memory::kernel_mode();
        }
        memory::free_user_pagetables(self.page_table_physaddr);
        println!("[!] - Process with page table at {:#x} deallocated its user page tables"
    }
}
```

5.6.1 User heap

Linux employs two primary system calls (syscalls) for heap memory allocation: *brk()* and mmap() [42]. Both syscalls offer a persistent "heap" space that remains intact across function calls. From a user-space perspective, heap allocation typically occurs through calls to malloc() or new [Reference].

The brk syscall functions by modifying the "breakpoint," which serves as a boundary between the heap and the stack within the virtual memory layout. The heap is situated at an address above the program code and expands upwards, while the stack initiates at a high address and grows downwards. The area separating the heap and the stack is termed as an unmapped guard page. By shifting this guard page upwards, additional space is allocated for the heap, albeit at the cost of reducing space for the stack.

The mmap() syscall, on the other hand, maps a page-aligned region of memory, which can subsequently be released (unmapped) back to the operating system. For this, I chose a range of memory that will be reserved for user space heaps.

In my implementation the allocator is initially provided with a fixed-size heap, achieved by mapping a single frame and marking the remainder as read-only. This approach employs a demand-paging mechanism, where the page fault handler allocates frames as they are accessed. Upon program termination, these frames are reclaimed. The *create_on_demand_pages* is designed to facilitate the on-demand allocation of pages in the memory, a strategy that aligns with modern demand-paging mechanisms [51]. The function takes three primary arguments: a pointer to the Level 4 page table (*level_4_table_ptr*), the starting virtual address for the pages to be mapped (*start_addr*), and the size of the memory region that should be mapped (size).

Upon invocation, the function first retrieves global memory information via a mutable reference, specifically targeting the frame allocator within this global context. A mutable reference to the Level 4 page table is then obtained, based on the provided pointer and the physical memory offset. Subsequently, an *OffsetPageTable* mapper is instantiated to manage the address translation process.

To decide the range of pages that require mapping, the function calculates the ending address based on the provided *start_addr* and size. Using these calculated boundaries, the function establishes a range of inclusive pages that should be mapped to frames. A single frame is then allocated for this entire range of pages, ensuring that a failure to allocate would result in an error being returned and forcing handling.

Following successful frame allocation, the function proceeds to map each page within the specified range to the allocated frame. During this operation, the page table flags are set to indicate that the pages are present, writable, and accessible by the user. This is executed within an unsafe block due to the direct manipulation of memory, and the page tables are subsequently flushed to commit these changes.

As a final step, the function updates the flags for the first page in the specified range to make it writable. This operation essentially designates this page as the 'owner' of the frame, which is important for managing frame deallocation. Like previous operations, this too necessitates the flushing of the page tables to ensure the changes are immediately effective.

On successful execution, the function returns a Result type indicating success, thereby offering a robust mechanism for dynamic memory allocation in user space.

5.7 User binaries

User programs are not linked against the kernel code. This means that they do not have access to the shared resources that the kernel supplies. This is important in terms of security, as it means that the user program is completely independent of the kernel, but can commuincate and request resources, using the API's system calls, which return the action of resources with the correct privilege that the user program should have.

Cargo allows the separation of the two systems into separate packages, which compile down to their own binaries. For the user program, this is an independent ELF binary. Which satisfies the spawn user thread

functionality in the kernel. This required a restructure of the project to accommodate this Rust feature but Rust's file based package manager made this straightforward.

The user program does not need it's own allocator, as this is defined in the API. This gets defined as the global allocator, but as this is in it's own rust package, it does not conflict or use the kernel's global allocator. For this, I simply used the linked list allocator crate as it was simpler and easier than re-implementing the fixed size block one that the kernel uses. This is all wrapped inside of the kernel's allocator anyway.

The structure of a user program is similar to how the kernel looked like in the beginning, with a no standard, no mangle, entry point definition of a start/main function.

```
#[no_mangle]
fn main() {
    let _main_thread_id = recv_id();
    println!("\n\nUser land program usage:
Input: h - (show these prompts again)\n
Input: b - (calls 3 threads incrementing an individual counter)\n
Input: r - (calls a function that recurs and allocated some space in  the stack)\n
Input: m - (calls a function that attempts to write to a kernel stack (malicious!)\n
Input: number up to 4 - (atomically incremenet a counter in parallel (number times))\n
Input: e - (allocate some space on the heap and then free)\n
        ");



    loop{
        let message = syscall::receive(0).unwrap(); // recv from keyboard interrupt handle
        let value = match message {
            Message::Packet(_, value, _) => value,
            _ => 0
        };
        let ch = char::from_u32(value as u32).unwrap();

        if ch == 'x' {
            println!("[!] - Exiting");
            break;
        } else if ch == 'b' {
            call_basic_threads(3);
        } else if ch == 'r' {
            call_recursive_threads();
        } else if ch == 't' {
            call_recursive_too_deep_threads()
        } else if ch == 'm' {
            call_malicious_thread();
        } else if ch == 'e' {
            call_heap_alloc();
        } else if ch == 'n'{
            null_safety_demo();
        } else if ch == '1' || ch == '2' || ch == '3' || ch == '4' {
            let ch_value = ch as u64 - '0' as u64;  // Convert char to its ASCII value and
            println!("{}", ch_value);
            call_basic_threads(ch_value);
        }
        else if ch == 'h' {
            println!("\n\nUser land program usage:
Input: h - (show these prompts again)\n
Input: b - (calls 3 threads incrementing an individual counter)\n
Input: r - (calls a function that recurs and allocated some space in  the stack)\n
Input: m - (calls a function that attempts to write to a kernel stack (malicious!)\n
Input: number up to 4 - (atomically incremenet a counter in parallel (number times))\n
Input: e - (allocate some space on the heap and then free)\n
        ");
```

```
        }

        syscall::send(1, message).unwrap(); // send to vga_listener
    }
}
```

The functions that are called following a particular key press make the foundation of the testing for the system. This operates as a very rudimental shell, allowing me to combine different function combinations. As well as stopping the process of having to restart the system every time. As you can see, this file is quite small an this is mainly due to the abstraction the API provides.

5.8 IPC with Sockets

After creating a package that runs independently from the kernel, within its own memory space, the system needs to be able to communicate between its threads. This includes a user-to-user thread, as well as kernel-to-kernel, and importantly, user-to-kernel and kernel-to-user thread communication.

The Inter-Process Communication (IPC) mechanism in this system is implemented using a message-passing paradigm, encapsulated in the Socket abstraction. The Socket object has four distinct states: Empty, Sending, Receiving, and SendReceiving, each serving a specific role in the message transfer process. Messages themselves are represented by the Message enum, which currently supports a single format, Packet, consisting of three 64-bit values. The data that can be part of a message are abstracted by the Data enum, which includes simple 64-bit values and references to other sockets.

The send_message method is an important function that attempts to send a message through the socket. If the socket is in the Empty state, it transitions to the Sending state and waits for a receiver. Should another sender attempt to use the socket while it is already in the Sending state, an error is returned to the calling thread. In case the socket is in the Receiving state, the message is immediately passed to the waiting thread, and both the sender and receiver threads are returned. The method also has provisions for restricted communication, where it allows messages to be sent only to a specific thread, identified by its thread ID.

The receive method sets the socket to a Receiving state if it is Empty, or retrieves a message if one is already waiting in a Sending state. In case the socket is already in a Receiving state, an error is returned to the calling thread. Similarly, the send_receive method attempts to implement a request-reply pattern by sending a message and immediately setting the socket to wait for a reply.

Each method returns a tuple containing optional sender and receiver threads, which enables greater flexibility in managing thread states outside the Socket abstraction. Moreover, the methods utilize a series of unsafe blocks to directly manipulate memory, underscoring the low-level nature of these operations. These methods are designed to be atomic operations, ensuring thread safety in multi-threaded environments.

The Socket abstraction also provides auxiliary methods for inspecting its state, peeking into the message if available, and resetting it to an Empty state.

The first Socket type that was implemented was to support taking keyboard interrupts and passing them to the user space, otherwise the kernel would hold on to these interrupts and the user program would not be aware that there was some input. To do this, the keyboard socket is created in kernel space so that it has access to the interrupts. When spawning a user thread, the socket is passed to the user program, meaning that it can call sys_send and sys_receive, with the socket ID. These system calls place a message into the socket for a receiving thread to retrieve respectively.

5.9 API

To facilitate the development of user programs and to enhance system efficiency, I created an abstraction layer which was incorporated into the project's architecture. This layer serves as an intermediary between user-level applications and the kernel, encapsulating the intricate details of system calls and CPU register manipulations. This architectural choice was done through the creation of another Rust package, which is integrated into the larger project ecosystem, in a similar way to how user programs were defined, the API was added to the cargo workspace, meaning that it was compiled independently from the kernel.

This abstraction layer offers a series of pre-defined methods, designed to eliminate the need for user applications to explicitly handle low-level system interactions, such as setting up the CPU registers and calling the *syscall* instruction. For instance, user programs seeking to engage in inter-process or inter-thread communication can simply invoke 'send' or 'receive' functions provided by this API. This not only simplifies the codebase of individual user programs but also enhances code maintainability and reduces the likelihood of errors that could arise from manual handling of system calls.

```
// Function to send a message to a file socket
// Takes a socket and a Message as arguments.
// Returns a Result indicating success or failure.
pub fn send(socket: u32, message: Message) -> Result<(), u64> {
    // Match on the type of message to send.
    match message {
        Message::Packet(message_data1, message_data2, message_data3) => {
            // Initialize a variable to hold the error code.
            let mut error_code: u64;

            // Execute inline assembly to perform the syscall for sending a message.
            unsafe {
                asm!("syscall",
                     in("rax") 4 + ((socket as u64) << 32),
                     in("rdi") message_data1,
                     in("rsi") message_data2,
                     in("rdx") message_data3,
                     lateout("rax") error_code,
                     out("rcx") _,
                     out("r11") _);
            }

            // Check if there was an error during the syscall.
            if error_code == 0 {
                return Ok(());
            }

            // Return the error code.
            Err(error_code)
        },
        // If the message type is not supported, return an error.
        _ => return Err(0)
    }
}
```

Moreover, the API further contributes to system efficiency by incorporating a thread yield function. This function provides user threads with the capability to voluntarily cede control of the CPU, enabling more effective scheduling of processor time. In doing so, the system can achieve a higher degree of multitasking and resource optimization. This becomes particularly beneficial in scenarios that involve complex computations or require real-time responsiveness.

In addition to simplifying system calls and CPU register interactions, the API plays a crucial role in abstracting memory management tasks, further streamlining the development process for user-level applications. Specifically, the API incorporates a memory allocator that allows user programs to dynamically allocate and deallocate memory without the need to implement these functionalities explicitly. By integrating this allocator into the abstraction layer, user applications are relieved of the complex task of managing memory, again, attempting to remove users from having to correctly implement memory management. This not only reduces the code complexity for user programs but also establishes a unified approach to memory management across the system. Through this unified approach, the system can achieve better memory utilisation and potentially reduce overheads associated with different memory management strategies. Thus, the incorporation of a memory allocator within the API reflects a holistic approach to simplifying user-level programming while optimizing system-level performance.

In summary, the abstraction layer serves dual roles: it simplifies the user-level programming experience and also contributes to the optimisation of system-level operations. Through its integration, the project achieves a balance between ease of development and operational efficiency, allowing multiple different user programs to be created. Currently there is only one program, however, more can be made, for example, a user shell program, a user networked program and others. With a user thread being able to launch a new user thread, you can have one main program that manages all of these programs. As well as several programs that can provide an abstract interface for user programs to use, such as a network stack and file system.

6 Evaluation

6.1 Methodology

Using the user space programs allowed me to develop tests of the system, which will be discussed below. These tests are aimed to test memory allocation and safety specifically, to demonstrate the uses of Rust as a systems development language. The memory and stack monitor, which are integrated loggers provide important test results that can be viewed and analysed used typical means, including python scripting to concisely draw conclusions on the state of the system after running different tests.

6.2 Testing

6.2.1 Incrementing counters

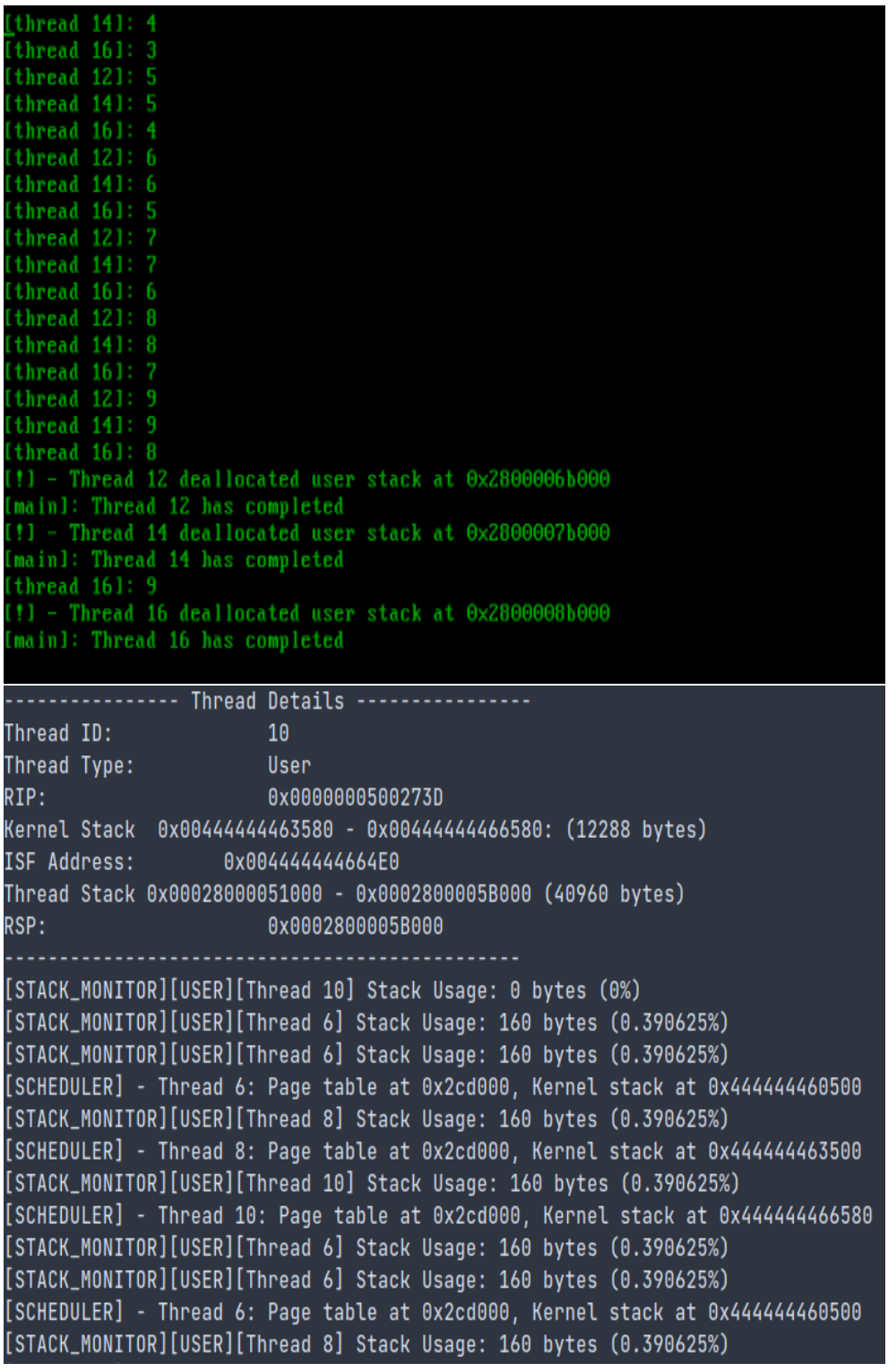
In the evaluation phase of my project, I paid particular attention to the kernel's capabilities in thread management, memory safety, and the scheduling of a reliable operating system. I began by running a single thread that incremented a counter up to 10, which served as a baseline for assessing the kernel's basic functionality and reliability including it's ability to output to the screen as well as generate logs, the format of these logs allowed me to develop additional scripts to automate the analysis process. The counter incremented sequentially as expected, providing initial evidence of a well-functioning system. The logs following this run showed the allocation of the stack frames and the subsequent deallocation following the thread's termination.

To further explore the system's multitasking capabilities and the effectiveness of the scheduler, a round robin scheduler, I scaled the experiment to include more concurrently running threads, each responsible for incrementing its own counter. The successful interleaving of outputs from these threads demonstrated the scheduler's ability to allocate CPU time efficiently among multiple tasks. The threads are spawned in the same order that they are spawned in the code, however the order of which threads outputte their incremented value alters, this is also an expected behaviour. It was noteworthy that each thread was able to perform its counting operation without interfering with the others, reinforcing the effectiveness of the scheduler in orchestrating the execution of multiple threads. This test was not merely a demonstration of concurrent execution but also a practical evaluation of the scheduler's fairness and task-switching mechanisms. This was tested up to 10 threads running concurrently, with no issues found.

Moreover, the scheduler log was incorporated to provide insights into the memory regions allocated to each thread. Each thread was found to be operating within its designated memory space, underscoring the kernel's capability for safe and effective memory management. Therefore, this evaluation serves as a robust validation of the kernel's multitasking, scheduling, and memory safety features. It also strengthens the argument for Rust's efficacy in operating system development, especially in comparison to languages like C that lack built-in memory safety mechanisms as well it's comparitive performance. Below are some examples of the threads running and interleaving.

```
[!] - Thread spawned with ID: 6
[thread 6]: 0
[thread 6]: 1
[thread 6]: 2
[thread 6]: 3
[thread 6]: 4
[thread 6]: 5
[thread 6]: 6
[thread 6]: 7
[thread 6]: 8
[thread 6]: 9
[!] - Thread 6 deallocated user stack at 0x2800003b000
[main]: Thread 6 has completed
```

```
---------------- Thread Details ----------------
Thread ID:              6
Thread Type:            User
RIP:                    0x0000000500273D
Kernel Stack  0x0044444445D500 - 0x00444444460500: (12288 bytes)
ISF Address:        0x00444444460460
Thread Stack 0x00028000031000 - 0x0002800003B000 (40960 bytes)
RSP:                    0x0002800003B000
------------------------------------------------
[STACK_MONITOR][USER][Thread 6] Stack Usage: 0 bytes (0%)
[STACK_MONITOR][USER][Thread 6] Stack Usage: 0 bytes (0%)
[STACK_MONITOR][USER][Thread 6] Stack Usage: 160 bytes (0.390625%)
[SCHEDULER] - Thread 6: Page table at 0x2cd000, Kernel stack at 0x444444460500
[STACK_MONITOR][USER][Thread 6] Stack Usage: 160 bytes (0.390625%)
[STACK_MONITOR][USER][Thread 6] Stack Usage: 160 bytes (0.390625%)
[SCHEDULER] - Thread 6: Page table at 0x2cd000, Kernel stack at 0x444444460500
```



In the course of my evaluation, I employed a Python script to generate graphs from the system logs, adding a quantitative dimension to my assessment. The logs in question were generated during a system run, capturing critical information about each thread's activities and the subsequent deallocation of resources. Specifically, after the completion of each thread's task, the logs indicated that the memory allocated for the stack was returned to the system without issues. This successful returning of memory serves as another benefit to the efficacy of Rust's Drop trait in managing the lifecycle of each thread.

The Drop trait ensures that resources are cleaned up when an object goes out of scope, which, in the context of this system, means that each thread's stack memory is properly deallocated. By examining the logs and the resultant graphs, one can conclude that the Drop trait provides a robust mechanism for automatic resource management in a multi-threaded environment. This not only confirms the integrity of the memory management system but also highlights the advantages of using Rust for kernel development, particularly its inherent focus on safety and resource management.

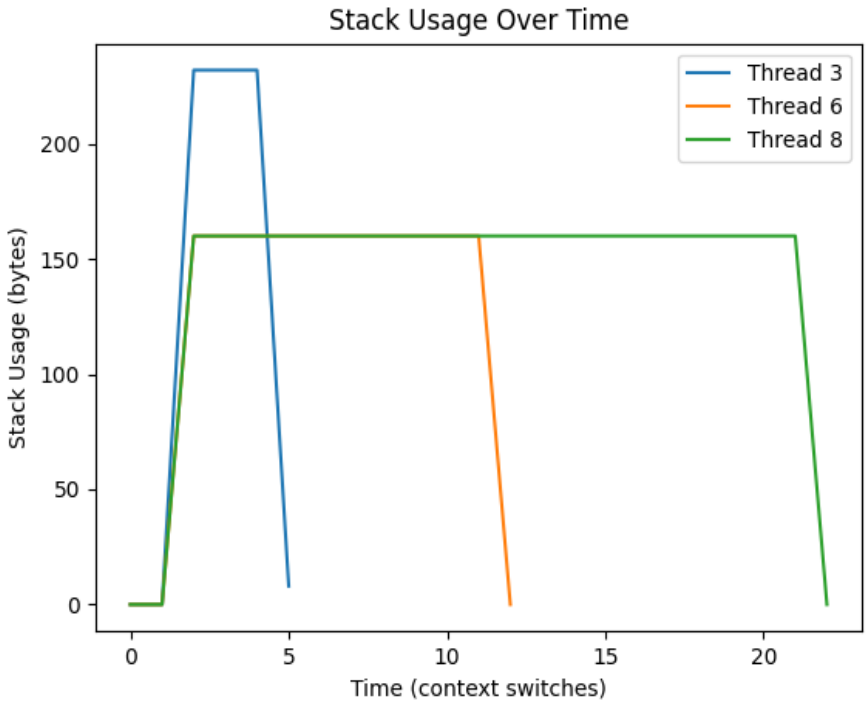


Figure 8: Two threads interleaving and dropping memory

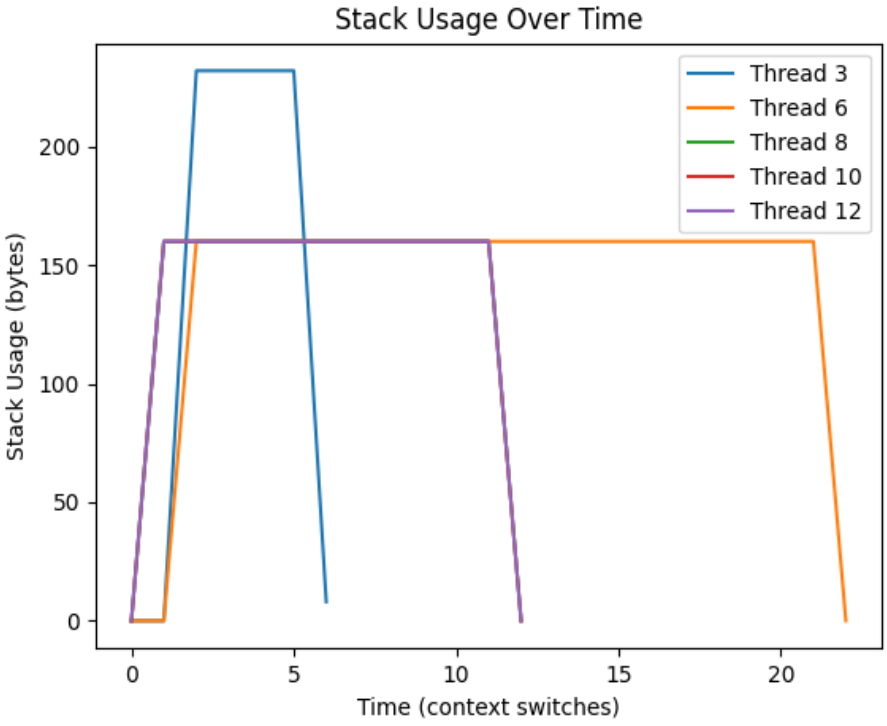


Figure 9: Three threads interleaving and dropping memory

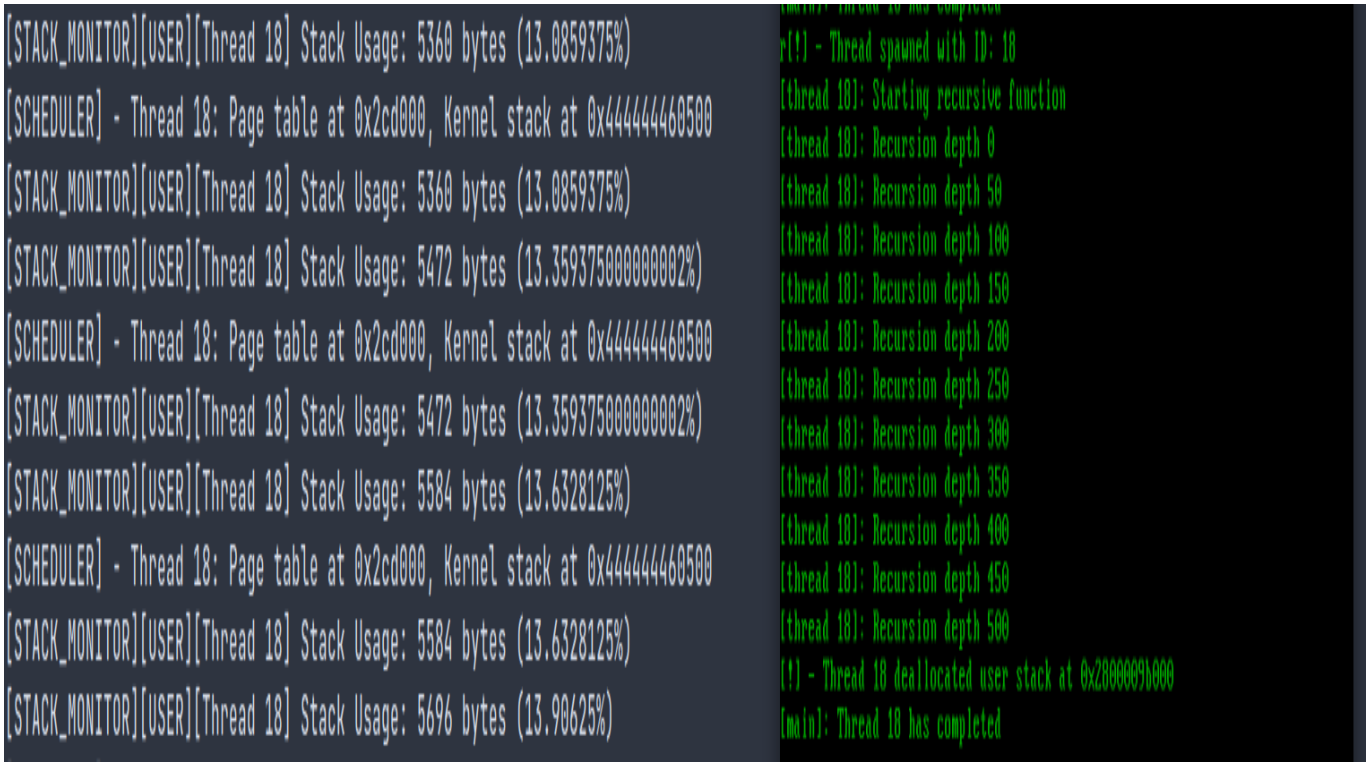
6.3 Filling the stack

In the realm of evaluating the operating system's performance and reliability, particularly in terms of memory management, the handling of stack overflow situations is a crucial test. To test this, I created a function with the ability to recursively allocate bytes on the user stack. Under standard operating conditions, this function does not come close to overflowing the stack's capacity, typically utilising under 15% of the allocated space. This serves as an initial validation of the system's competent management of memory under routine circumstances. As well as a small sanity check to ensure the efficacy of the monitoring system, similar to the standard incrementing counter in the previous test.

However, to test the case of overflowing, I created a similar function to a test more extreme use-case, which can be seen in the listing below. Specifically, I manipulated the recursion depth of the function to ensure that it would exceed the stack's capacity, essentially forcing the system into a corner where it would have to deal with an overflow. The goal was to ascertain how the system would respond when the stack is not just full but is also attempting to write beyond its last allocated page. In traditional operating systems, particularly those developed in languages like C or C++, such an event could trigger undefined behavior, leading to system instability or even security vulnerabilities.

Contrastingly, this system handled this fault gracefully. It detected the overflow condition, triggered a page fault as expected, and then proceeded to exit the thread responsible for the overflow. The system did this without crashing or causing collateral damage to other running processes, thus highlighting Rust's innate capabilities for secure memory management. Importantly, the system also cleanly deallocated the memory allocated for that particular overflowing stack. This exemplifies the strength of Rust's Drop trait, which ensures that resources are reclaimed effectively, even when a program doesn't behave as expected.

The evidence for these can be found be found in the following series of screenshots that look at analysing the logs at the point of the fault. The first screenshot portrays the system effortlessly managing the stack under a low-stress recursion test. The second screenshot, in contrast, captures the moment of the page fault triggered by the stack overflow, validating the system's robustness and fault-tolerance. The combination of these tests not only provides a view of the system's memory management capabilities but also positions it as uniquely resilient compared to existing operating systems. The screenshots for stack usage over time also depict how the memory is freed when a thread exits. The larger orange line in the last screenshot looks different due to the much larger scall the stack uses bytes, however, the same amount of memory is dropped at the end.



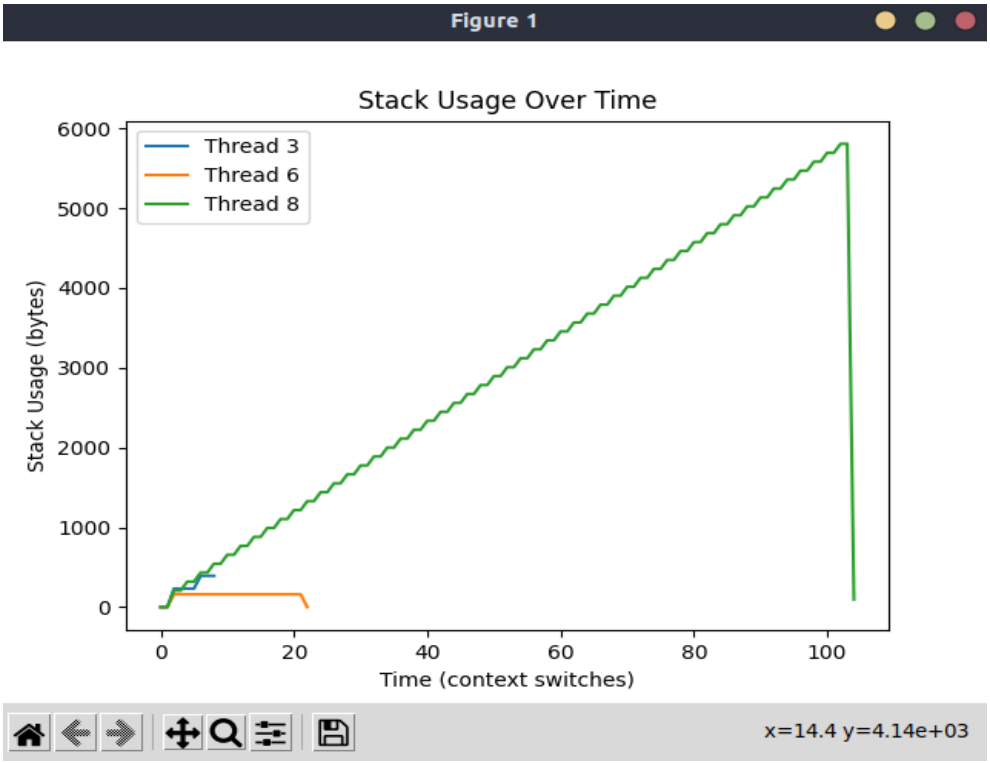


Figure 10: A recursion with a shallow depth and subsequent memory dropping

Below is the code that will be run to test allocating more than one hundred percent of the stack space. And it's following output.

```
fn recursive_too_deep_function(depth: usize, thread_id: u64) {
    // Print the current depth to monitor the recursion
    if depth % 50 == 0 { // print every 50th depth to avoid flooding the console
        println!("[thread {}]: Recursion depth {}", thread_id, depth);
    }

    // Artificially use up some stack space
    let _array: [u64; 1000] = [0; 1000]; // 100 include_bytes!("

    syscall::thread_yield();
    // Recursive call
    if depth < 500 { // Limiting the depth to prevent infinite recursion
        recursive_too_deep_function(depth + 1, thread_id);
    }
}
```

6.3.1 Allocating and deallocating

In an assessment of an operating system's robustness and efficiency, the effective management of memory resources is an important consideration. I deployed a Python script with the log from the memory logger to meticulously analyse the system's memory usage and deallocation patterns. While the script effectively identified regions that were not deallocated upon the system's completion, a deeper analysis revealed that this was not indicative of a memory leak, an issue often common in systems programmed in languages like C.

These persistently allocated regions, rather, are intentionally mapped to the kernel space and are designed to remain allocated throughout the system's runtime. Such allocation is by design and serves essential system functions, rather than representing an oversight or inefficiency. It's crucial to differentiate this kind of sustained allocation from a memory leak, which would signify an unintended consumption of memory resources and result in gradual system degradation. The regions of memory that should be deallocated are any frames that are mapped within the user space, from analysing the logs and using the BootInfo struct to determine memory region types, no user space memory remained deallocated.

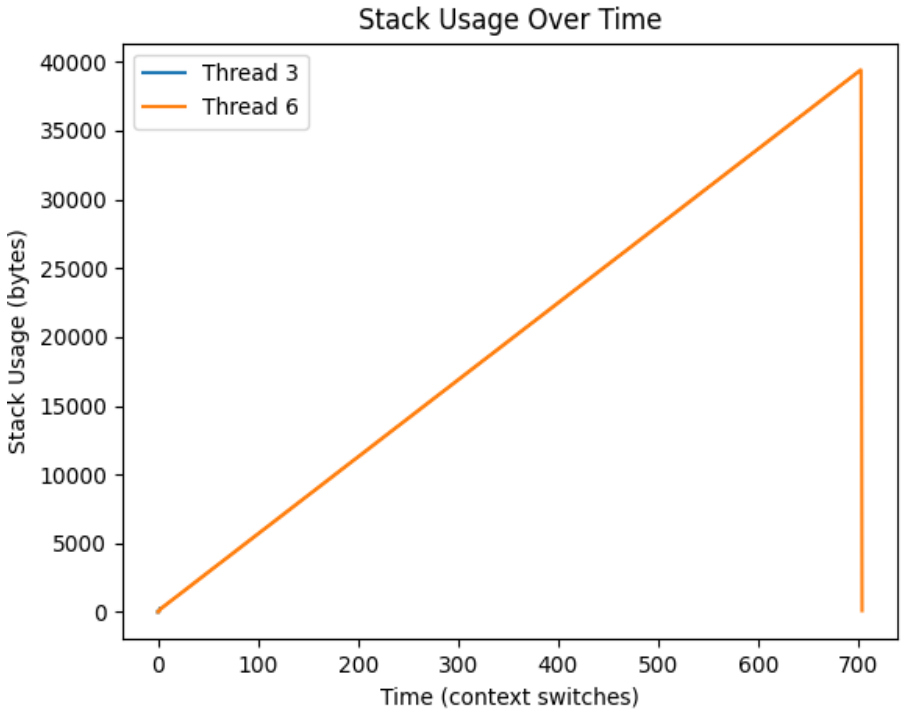


Figure 11: Attempting to fill the stack past 100% and it's subsequent dropping of that threads memory

```
[SCHEDULER] - Thread 6: Page table at 0x2cd000, Kernel stack at 0x444444460500
[STACK_MONITOR][USER][Thread 6] Stack Usage: 40528 bytes (98.9453125%)
[STACK_MONITOR][USER][Thread 6] Stack Usage: 40640 bytes (99.21875%)
[SCHEDULER] - Thread 6: Page table at 0x2cd000, Kernel stack at 0x444444460500
[STACK_MONITOR][USER][Thread 6] Stack Usage: 40640 bytes (99.21875%)
[STACK_MONITOR][USER][Thread 6] Stack Usage: 40752 bytes (99.4921875%)
[SCHEDULER] - Thread 6: Page table at 0x2cd000, Kernel stack at 0x444444460500
[STACK_MONITOR][USER][Thread 6] Stack Usage: 40752 bytes (99.4921875%)
[STACK_MONITOR][USER][Thread 6] Stack Usage: 40864 bytes (99.765625%)
[SCHEDULER] - Thread 6: Page table at 0x2cd000, Kernel stack at 0x444444460500
[STACK_MONITOR][USER][Thread 6] Stack Usage: 40864 bytes (99.765625%)
Exiting thread 6
```

Figure 12: Stack monitor log of stack usage

```
Region ('0x0000000000293000', '0x0000000000294000') was not deallocated
Region ('0x000000000002B0000', '0x000000000002B1000') was not deallocated
Region ('0x00000000002A3000', '0x00000000002A4000') was not deallocated
Region ('0x00000000002C2000', '0x00000000002C3000') was not deallocated
Region ('0x00000000002A6000', '0x00000000002A7000') was not deallocated
Region ('0x0000000000296000', '0x0000000000297000') was not deallocated
Region ('0x00000000002C1000', '0x00000000002C2000') was not deallocated
Region ('0x00000000002C3000', '0x00000000002C4000') was not deallocated
Region ('0x00000000002C7000', '0x00000000002C8000') was not deallocated
Region ('0x00000000002BB000', '0x00000000002BC000') was not deallocated
Region ('0x00000000002B2000', '0x00000000002B3000') was not deallocated
Region ('0x0000000000298000', '0x0000000000299000') was not deallocated
Region ('0x00000000002B8000', '0x00000000002B9000') was not deallocated
Region ('0x00000000002BE000', '0x00000000002BF000') was not deallocated
Region ('0x00000000002AE000', '0x00000000002AF000') was not deallocated
Region ('0x0000000000285000', '0x0000000000286000') was not deallocated
Region ('0x000000000028F000', '0x0000000000290000') was not deallocated
```

Figure 13: Regions not deallocated during the run of the program, which belong to the kernel

In the Linux operating system, a C-based system, the kernel also retains certain memory regions for its internal use. However, the lack of memory safety guarantees in C means that improper deallocations or buffer overflows could compromise these regions, leading to potential system instability or security risks. Rust's strong typing and ownership model make it less likely that such regions would be improperly accessed or deallocated, enhancing the system's overall reliability.

The screenshot below displays a list of regions that were not deallocated, affirming that these areas are indeed part of the kernel's own space. While at a glance this may raise concerns, understanding the context and function of these allocations stops such worries and points to the system's well-considered memory management strategy. Therefore, while these regions remain allocated, they do so to serve their intended purpose, rather than as a symptom of a memory leak or system inefficiency. With more development time, a graceful exit signal to the kernel could be created which would successfully unmap and deallocate all memory before system shutdown.

6.3.2 Memory region reuse

The system operates in a manner that is similar to a queue when it comes to the allocation and deallocation of memory spaces dedicated to threads, this is due to the operation '.next()' on the list of memory regions. More explicitly, when a thread deallocated its allocated memory, that particular region does not get reassigned instantaneously to the next available thread, it remains at the front of the list. Rather, the system continues to allocate fresh memory regions from the available pool until the point where no more free memory exists. Once this limit is reached, the system reverts to the strategy of using memory regions at the front of the list in a circular manner. This serves as tangible proof of effective memory deallocation and resource recycling.

This behavior becomes especially important when viewed through the lens of Rust's capabilities and memory safety guarantees. Rust's ownership model and type system play a pivotal role in ensuring that memory is managed in a safe and predictable manner. These features of the language mitigate the risk of common pitfalls like buffer overflows, data races, and other concurrency errors, which are often prevalent in systems written in languages that do not offer the same level of memory safety.

The queuing behavior of the system also matches with Rust's philosophy of resource safety. By ensuring that memory regions are genuinely deallocated and are available for reuse, the system exemplifies the language's commitment to both performance and safety. This is of paramount importance in operating systems where memory is a constrained resource and its effective management is crucial for system stability and security. No explicit 'free' call was made to deallocate a thread's memory, Rust's scoping rules manage this instead. Therefore, the system not only validates the effectiveness of its own memory management strategies but also helps to prove

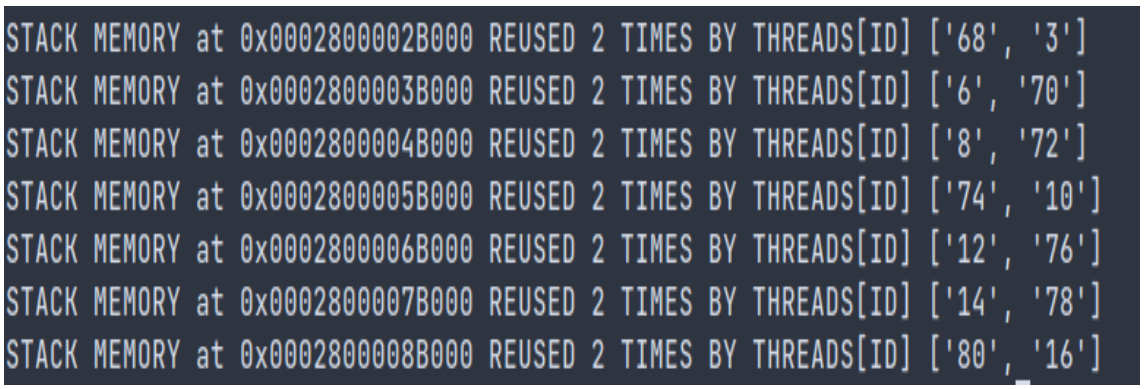


Figure 14: Reuse of 'old' memory regions for fresh threads

Rust's robustness as a language for secure, high-performance systems development.

6.3.3 Writing to a kernel thread's memory space from a User thread

Another important test is the attempt to write into a kernel thread's memory space from a user-level thread. This was to test the system's capacity to uphold memory boundaries and safety regulations, a challenge that operating systems frequently encounter. As anticipated, the test triggered a page fault, prompting the system to initiate its fail-safe mechanisms. The user-level thread responsible for the unauthorised memory access was terminated, and its allocated memory was successfully reclaimed by the system.

This outcome shows the robustness of the system's memory management and error-handling routines, but also a demonstration of the advantages of using Rust as the backbone for systems programming. Rust's strong type system and ownership model enforce strict rules concerning data access and manipulation, creating an additional layer of memory safety. In a less rigorously designed language environment, an unauthorized memory access could potentially lead to catastrophic failures and vulnerabilities, including system crashes or security vulnerabilities such as privilege escalation.

The occurrence of the page fault, and the system's handling of it, shows that the boundaries set by Rust's type system are not merely theoretical constructs but are actively enforced at runtime. This creates a safety net that operates as a last line of defense against a wide array of common programming errors and security threats. Thus, the test serves to validate not only the system's capability to securely manage memory but also Rust's efficacy in serving as a tool that inherently prioritizes and enforces secure coding practices.

6.4 Test omissions

The choice of programming language can significantly influence the scope and nature of the required testing. Using Rust as the foundational language for this kernel affords a set of safety guarantees that remove the need for an entire suite of tests commonly essential in systems developed in languages such as C or C++. Specifically, the language's strict ownership model and strong type system preempt a range of issues related to memory safety, reducing the test cases required to validate these aspects.

For instance, in a Linux-based system developed primarily in C, extensive tests are needed to check for buffer overflows, null pointer dereferences, and use-after-free errors, among other vulnerabilities. These issues represent real-world security threats that have been exploited to compromise systems. Contrastingly, Rust's compile-time checks mitigate these risks from the onset, leaving a whole subset of memory-related tests redundant. In essence, many tests that would be classified as mandatory in a C-based environment become redundant in a Rust environment.

Buffer overflow tests, for instance, are designed to ensure that an application does not write data beyond the boundaries of allocated buffers, a common vulnerability that can lead to arbitrary code execution. Tests for null pointer dereferences aim to ascertain that the system does not attempt to access memory locations through uninitialized or null pointers, which can lead to undefined behavior and system crashes. Similarly, use-after-free tests are implemented to prevent the system from accessing memory that has already been deallocated, another vector for arbitrary code execution.

In contrast, Rust's robust type system and strict ownership model offer built-in defenses against these vulnerabilities, alleviating the need for exhaustive testing. The language enforces strict checks at compile-time that prevent buffer overflows. For example, Rust's slice type, which is a more flexible version of arrays, provides

```
EXCEPTION: PAGE FAULT
Accessed Address: VirtAddr(0x444444450284)
Error Code: PROTECTION_VIOLATION | CAUSED_BY_WRITE | USER_MODE
InterruptStackFrame {
    instruction_pointer: VirtAddr(
        0x50017b0,
    ),
    code_segment: 51,
    cpu_flags: 0x202,
    stack_pointer: VirtAddr(
        0x2800003af70,
    ),
    stack_segment: 43,
}
Page fault error: Error: Unexpected table flags
```

Figure 15: Correctly faulting from an illegal page access attempt

safe access to array elements, ensuring that any attempt to access data outside the allocated memory is caught during compilation. This negates the need for runtime tests to validate against buffer overflows.

Null pointer dereferences are also mitigated in Rust through its *Option* type, which makes the absence of value explicit and forces the developer to check the result of the Option, otherwise it will cause a compile-time error. Unlike in C, where a null pointer can be unwittingly dereferenced, Rust requires an explicit handling of the Option type, therefore removing null dereference issues at compile-time.

Likewise, the language's borrowing and ownership principles are designed to prevent use-after-free errors. In Rust, once the ownership of a variable is transferred or the variable goes out of scope, it is not accessible, thereby invalidating any dangling references. The borrow checker ensures that references to variables obey two key rules: you can't have mutable and immutable references simultaneously, and you can't have a mutable reference while the variable is being accessed. This prevents simultaneous read and write operations on memory locations, thereby eliminating race conditions and associated vulnerabilities.

Furthermore, Rust's "zero-cost abstractions" mean that these safety features are enforced without incurring a performance penalty, making it an attractive alternative for systems where both safety and performance are critical. This not only streamlines the testing process but also enhances the overall reliability and security of the system, a noteworthy advantage that could shift future systems development paradigms.

Therefore, the language choice of Rust inherently brings about a more focused and efficient evaluation phase, allowing for a concentration on higher-order functionality and performance optimization. It provides a compelling argument for the adoption of Rust in the development of secure, high-performance systems, fundamentally altering what is deemed necessary in the evaluation of kernel and system robustness.

6.4.1 Data races

The responsibility for ensuring thread safety when accessing shared resources largely falls on the programmer. Mutexes and spinlocks are commonly employed to serialise access to shared data, but their proper use is not enforced by the language itself. For instance, a simplistic C example that uses a pthread mutex to protect shared data:

```
pthread_mutex_t lock;
int shared_resource;

void update_resource(int value) {
    pthread_mutex_lock(&lock);
    shared_resource = value;
```

```
    pthread_mutex_unlock(&lock);
}
```

In this example, the pthread mutex lock is used to ensure exclusive access to the shared_resource. While this approach is effective, it relies entirely on the discipline of the developer to consistently lock and unlock the mutex around every access of the shared resource. Failure to do so can result in data races, a class of bug that is notoriously difficult to diagnose and fix.

Rust, on the other hand, elevates the concept of mutability and ownership to the language level, providing compile-time checks for data races. Concurrent access to shared state in Rust is commonly managed through abstractions like Mutex or RwLock, which are part of the standard library. These abstractions are designed to be safe, meaning they incorporate the lock semantics into the type system:

```
use std::sync::Mutex;
let shared_resource = Mutex::new(0);

// In some thread
let mut data = shared_resource.lock().unwrap();
*data += 1;
```

In this Rust example, shared_resource is wrapped in a Mutex, and the lock method returns a unique reference to the data. Rust's ownership rules ensure that at most one thread can hold this unique reference at any given time, thereby eliminating the possibility of data races. If another thread attempts to lock the mutex while it's already locked, it will be blocked until the lock is available.

Moreover, Rust's Drop trait ensures that the lock is automatically released when the unique reference goes out of scope, reducing the likelihood of deadlocks, a common pitfall when manually managing locks. This is a stark contrast to C, where forgetting to release a lock can lead to a deadlock situation that halts the entire system.

For instance, a wait queue is maintained to keep track of threads that are waiting for their turn to be executed by the processor. The mutual exclusion provided by the lock ensures that this queue is accessed in a serialised manner, preventing race conditions that could otherwise corrupt the queue's state.

Additionally, a reference to the thread that is currently running is encapsulated behind a lock. This is imperative because the scheduler, or any other system component that wishes to manipulate the running thread, must first acquire this lock. This guarantees that the thread's state is not simultaneously altered by multiple entities, thereby ensuring its consistent management.

Another resource that benefits from such locking is the counter used for generating unique thread identifiers. By locking this counter, the system ensures that each thread receives a distinct identifier, eliminating the risk of identifier duplication which could lead to erroneous behavior.

Interprocess communication (IPC) is also a facet that requires careful synchronisation. Access to the socket used for IPC is controlled through a lock, allowing only one process at a time to send or receive messages. This ensures that the messages are not corrupted or lost, providing a robust communication mechanism between processes.

Moreover, memory management involves the manipulation of a list of memory regions available for allocation and deallocation. Locking is essential here to prevent multiple threads from simultaneously allocating or deallocating the same memory region, which would lead to undefined behavior and system instability.

Lastly, the functions responsible for actual memory allocation are also protected by locks. This prevents scenarios where multiple allocations could potentially be directed to the same memory region, a situation that would not only lead to data corruption but also compromise the integrity of the system's memory layout.

In summary, Rust's type system and standard library provide robust, language-level guarantees that effectively prevent data races and deadlocks. These features make it significantly easier to write concurrent code that is both safe and efficient, a combination that is challenging to achieve in languages that lack such built-in safety mechanisms.

6.5 Comparison

In the course of developing this system, one of the largest advantages I observed was the abstraction capabilities that Rust offers over C. Rust's type system and ownership semantics, for instance, allowed me to express complex logic with a degree of conciseness and clarity that would have been more difficult in C. This, in turn, reduced the cognitive load during development, enabling me to focus on the higher-level architecture of the system rather than getting entangled in the intricacies of language aspects such as pointer arithmetic.

Rust's syntactic sugar further simplified the development process. Features such as pattern matching, the Option and Result types, and iterators provide more expressive ways to handle control flow and data

manipulation. These features made the codebase more readable and maintainable. In contrast, accomplishing the same functionality in C would require verbose error-handling code, explicit/manual memory management, and often a reliance on external libraries to provide similar abstractions.

For example, in C, the absence of sum types often leads to the use of struct variants and flags to indicate state, making the code harder to read and prone to errors. In Rust, however, the use of enums for sum types makes it easier to understand what states a system can be in, thus reducing the likelihood of invalid state transitions. This is particularly beneficial in a complex system like an operating system kernel, where state management is critical. Examples of this were the Socket type, used for IPC, where the state of the Socket is key to the runtime of the system. Using implementations structs follows a more object oriented approach which for certain resources makes development much simpler in comparison.

Moreover, Rust's package management system, Cargo, streamlined the process of integrating external libraries, which often comes with additional challenges in C due to the lack of a standard package manager. This made it easier to incorporate sophisticated algorithms and data structures into the system, thereby reducing the time spent on implementation and testing.

In Linux, reference counting is used to keep track of the number of references to a resource, such as a file or a network socket. When this count drops to zero, the resource is automatically deallocated, thus ensuring that it is neither leaked nor used after it has been freed. While effective, this approach has its drawbacks. Incorrectly updating the reference count can result in resource leaks or premature deallocation, leading to undefined behavior. Moreover, managing these counts can be error-prone and necessitates additional code for incrementing and decrementing the counts, adding to the complexity and potential for error.

In contrast, Rust's ownership model, combined with its borrowing and lifetime mechanisms, provides a more elegant solution to resource management without requiring explicit reference counts. Borrowing rules ensure that references to the resource are valid as long as they are in use. This eliminates the need for manual deallocation and provides strong compile-time guarantees against use-after-free errors [18], thus reducing the risk of resource leaks and security vulnerabilities, which is another clear advantage.

In summary, Rust's abstractions and syntactic sugar not only made the development process more efficient but also resulted in a more robust and maintainable system. These features allowed me to abstract away many of the lower-level details that are often stumbling blocks in C, facilitating a smoother and faster development experience. ref counters

6.5.1 Runtime efficiency

In the context of this project, I observed that Rust's programming model offers advantages in runtime efficiency over traditional C-based systems like Linux. One of the most clear points is that Rust's strong compile-time checks substantially minimize the need for runtime checks, a common source of overhead in C-based systems. This reduces the need for runtime validations to catch these errors, thereby enhancing the system's overall performance [18].

Moreover, the absence of a garbage collector in Rust is a key factor contributing to its runtime efficiency. While garbage collection can simplify memory management, and is required for a system to run, it introduces a layer of unpredictability in system performance, particularly in real-time or high-availability systems. Rust's features automates memory management without the unpredictability of garbage collection, resulting in more consistent performance characteristics. [71]

Additionally, Rust's "zero-cost abstractions" philosophy ensures that higher-level constructs do not impose a performance penalty.These constructs generate code that is as efficient as their hand-optimized C counterparts, allowing developers to write expressive yet performant code.

6.6 Discussion

A more accurate discussion of performance would further highlight advantages of Rust, however, a reliable timing system was something that I was trying to develop nearer to the end of the project, but I would find inconsistent metrics that were difficult to interpret. With a more reliable timer system, quantitative comments on how quickly and efficiently memory management and process management could be made. I faced issues with this due to the variable nature of the timer interrupts in the system, without access to dedicated hardware for timing I had to rely on the uniformity of a timer interrupt.

The scope of an operating system is very large, features such as a file system, networking, GUI and extensions to the user facing API could create a more feature rich system, however time constraints prevented significant development on these. However, some small progress was made on creating some form of a network stack, by accessing the PCI devices available to the system, which including the network card provided by the QEMU interface, but due to the size of such a network stack, it became out of scope for this. A file system would involve interfacing with a RAM disk, and implementing structs to represent a file and a directory. This is a very

common task and a potential port of other filesystems, such as ext3 or ext4 could be feasible following some modifications to the kernel space. This would likely need to be implemented in user space, which resembles a micro kernel architecture more than a monolithic one.

With the development of a user space and an API, the ease of creating user programs is greatly increased. Creating a shell program that just pulls characters from a keyboard socket would allow a more typical interaction with the system. Due to user programs being independent Rust projects, that only need to use the API package to run within the system, this lays the foundation for further development which I plan on doing.

7 Conclusion

In light of the critical need for enhancing memory safety in operating systems, particularly at the kernel level, this project was aimed to explore the use of the Rust programming language in kernel development. The objective was to create a microkernel architecture, along with a user space and an API interlinking them, all implemented in Rust. The primary focus was on rigorously evaluating the memory safety attributes of the system, drawing on Rust's built-in safety guarantees. This project successfully met its objectives, offering a practical implementation that not only highlights the merits of using Rust in system-level programming but also underscores its potential for substantially reducing low-level vulnerabilities associated with memory management. Therefore, this research fits in to the ongoing discourse on elevating systems security and reliability, and it validates Rust's suitability for such critical applications.

The key findings of this project were the clear advantages over similar languages in terms of the ease of development and the powerful abstractions provided by the design of Rust. The powerful compile time prevention of problems that are prolific in the other systems programming was surprising, even with the knowledge from the outset that some of these would be prevented. Over the course of the project, the errors that would be prevented before even compiling really consolidated my opinion that Rust provides significant advantages to systems, whether they are completely written in Rust or whether kernel drivers in Rust are being implemented into a C based system such as Linux.

This has a significant impact on the security of system, preventing programmers from being capable of introducing fatal vulnerabilities into a system. Of course, a completely secure operating system C is possible, but considerably difficult to create and design for any developer, to the point of there being very few examples. Linux itself has an incredible backlog of vulnerabilities, even with the continual community development and attention. Rust, and its design, create a different scope for system design, where some of the features that are considered are not relevant or even make sense for Rust.

This does not mean that the development of such Rust systems are without difficulties, but these are not due to the language specifically. Typical low level issues, such as comprehensive debugging and the low level nature of memory management can still seem random without being able to debug the entire memory space for every error. However, these are issues are not language based. One large issue that was difficult to resolve for a good portion of the project was the freeing of the page tables for user processes. The system would repeatedly attempt to Drop a page table for a user process after it finished. This concerned me and made me doubtful of Rust, however it turned out to be the way that I was using the recursive freeing of the page tables, I was attempting to remove the page table for every removal of each page within the table. This example shows that system performance and security is still tied to the competency of the programmer, but Rust significantly shifts the scope of the errors away from system critical bugs.

In conclusion, this project served as both an intriguing and challenging foray into the world of low-level systems development, particularly focusing on the security implications of memory management. While this project proved to me the power of Rust, it equally gave insight into the issues that common systems development faces and how changing the language just shifts the design questions to a new space. In my opinion, Rust provides a strong starting point for questioning the design of other older languages, and how we can change the scope of a language to fit specific scenarios more appropriately. Additionally, the use of a language's design, which is aimed to reduce the developer management of memory successfully increases the security of a system.

References

[1] Haitham Akkary and Michael A Driscoll. "A dynamic multithreading processor". In: *Proceedings. 31st Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE. 1998, pp. 226–236.

[2] Ghania Al Sadi. "Analyzing Master Boot Record for Forensic Investigations". In: *International Journal of Applied Information Systems* 10 (Apr. 2016), pp. 22–26. DOI: 10.5120/ijais2016451541.

[3] Timo O Alanko and A Inkeri Verkamo. "Segmentation, paging and optimal page sizes in virtual memory". In: *Performance Evaluation* 3.1 (1983), pp. 13–33.

[4] SLAB Allocator. "Linux Kernel Internals". In: ().

[5] *AmigaOS*. https://www.amigaos.net/.

[6] Abhiram Balasubramanian et al. "System programming in rust: Beyond safety". In: *Proceedings of the 16th workshop on hot topics in operating systems*. 2017, pp. 156–161.

[7] Vladimir Bashun et al. "Too young to be secure: Analysis of UEFI threats and vulnerabilities". In: *14th Conference of Open Innovation Association FRUCT*. 2013, pp. 16–24. DOI: 10.1109/FRUCT.2013.6737940.

[8] Fabrice Bellard. "QEMU, a Fast and Portable Dynamic Translator". In: *2005 USENIX Annual Technical Conference (USENIX ATC 05)*. Anaheim, CA: USENIX Association, Apr. 2005. URL: https://www.usenix.org/conference/2005-usenix-annual-technical-conference/qemu-fast-and-portable-dynamic-translator.

[9] Jeff Bonwick et al. "The slab allocator: An object-caching kernel memory allocator." In: *USENIX summer*. Vol. 16. Boston, MA, USA. 1994.

[10] Jeff Bonwick, Jonathan Adams, and Wonsuk Song. "Magazines and Vmem: Extending the Slab Allocator to". In: (2017).

[11] H. Chang, Ramesh Karne, and A.L. Wijesinha. "Insight into the x86-64 bare PC application boot/load/run methodology". In: (Jan. 2013), pp. 31–36.

[12] David Cooper et al. *BIOS Protection Guidelines*. en. 2011-04-29 00:04:00 2011. DOI: https://doi.org/10.6028/NIST.SP.800-147. URL: https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=908423.

[13] Robert C Daley and Jack B Dennis. "Virtual memory, processes, and sharing in Multics". In: *Communications of the ACM* 11.5 (1968), pp. 306–312.

[14] Lucas Davi et al. "Privilege escalation attacks on android". In: *Information Security: 13th International Conference, ISC 2010, Boca Raton, FL, USA, October 25-28, 2010, Revised Selected Papers 13*. Springer. 2011, pp. 346–360.

[15] Peter J Denning. "Virtual memory". In: *ACM Computing Surveys (CSUR)* 2.3 (1970), pp. 153–189.

[16] Peter J. Denning. "Virtual Memory". In: *ACM Comput. Surv.* 2.3 (Sept. 1970), pp. 153–189. ISSN: 0360-0300. DOI: 10.1145/356571.356573. URL: https://doi.org/10.1145/356571.356573.

[17] *Embedded Rust Book*. https://docs.rust-embedded.org/book/intro/install.html.

[18] Mehmet Emre et al. "Translating C to safer Rust". In: *Proceedings of the ACM on Programming Languages* 5.OOPSLA (2021), pp. 1–29.

[19] D. R. Engler, M. F. Kaashoek, and J. O'Toole. "Exokernel: An Operating System Architecture for Application-Level Resource Management". In: *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*. SOSP '95. Copper Mountain, Colorado, USA: Association for Computing Machinery, 1995, pp. 251–266. ISBN: 0897917154. DOI: 10.1145/224056.224076. URL: https://doi.org/10.1145/224056.224076.

[20] D. R. Engler, M. F. Kaashoek, and J. O'Toole. "Exokernel: An Operating System Architecture for Application-Level Resource Management". In: *SIGOPS Oper. Syst. Rev.* 29.5 (Dec. 1995), pp. 251–266. ISSN: 0163-5980. DOI: 10.1145/224057.224076. URL: https://doi.org/10.1145/224057.224076.

[21] James C Foster. *Buffer overflow attacks*. 2005.

[22] Jayneel Gandhi et al. "Efficient Memory Virtualization: Reducing Dimensionality of Nested Page Walks". In: *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 2014, pp. 178–189. DOI: 10.1109/MICRO.2014.37.

[23] Thanassis Giannetsos et al. "Arbitrary code injection through self-propagating worms in von neumann architecture devices". In: *The Computer Journal* 53.10 (2010), pp. 1576–1593.

[24] Mel Gorman. *Understanding the Linux virtual memory manager*. Prentice Hall Upper Saddle River, 2004.

[25] Dylan Griffiths and Dwight Makaroff. "Hybrid vs. monolithic OS kernels: a benchmark comparison". In: *Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research*. 2006, 30–es.

[26] *GRUB*. https://www.gnu.org/software/grub/.

[27] Swapnil Haria, Mark D Hill, and Michael M Swift. "Devirtualizing memory in heterogeneous systems". In: *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. 2018, pp. 637–650.

[28] Harshvardhan Harshvardhan and Shridhar Irabatti. "Study of Kernels in Different Operating Systems in Mobile Devices". In: *Journal of Online Engineering Education* 14.1s (2023), pp. 21–26.

[29] Makoto Higashi et al. “An effective method to control interrupt handler for data race detection”. In: *Proceedings of the 5th Workshop on Automation of Software Test*. 2010, pp. 79–86.

[30] *Higher half Kernel*. https://wiki.osdev.org/Higher_Half_Kernel.

[31] Jerry Huck and Jim Hays. “Architectural support for translation table management in large address space machines”. In: *Proceedings of the 20th annual international symposium on computer architecture*. 1993, pp. 39–50.

[32] Bruce L Jacob and Trevor N Mudge. “A look at several memory management units, TLB-refill mechanisms, and page table organizations”. In: *ACM SIGPLAN Notices* 33.11 (1998), pp. 295–306.

[33] J. Jex. “Flash memory BIOS for PC and notebook computers”. In: *[1991] IEEE Pacific Rim Conference on Communications, Computers and Signal Processing Conference Proceedings*. 1991, 692–695 vol.2. DOI: 10.1109/PACRIM.1991.160834.

[34] Fethullah KARABİBER, Ahmet SERTBAŞ, et al. “Dynamic memory allocator algorithms simulation and performance analysis”. In: *IU-Journal of Electrical & Electronics Engineering* 5.2 (2005), pp. 1435–1441.

[35] Avi Kivity et al. “kvm: the Linux virtual machine monitor”. In: *Proceedings of the Linux symposium*. Vol. 1. 8. Dttawa, Dntorio, Canada. 2007, pp. 225–230.

[36] Gerwin Klein et al. “Comprehensive formal verification of an OS microkernel”. In: *ACM Transactions on Computer Systems (TOCS)* 32.1 (2014), pp. 1–70.

[37] Kenneth C Knowlton. “A fast storage allocator”. In: *Communications of the ACM* 8.10 (1965), pp. 623–624.

[38] Leslie Lamport. *On interprocess communication*. Digital Equipment Corporation Systems Research Center, 1985.

[39] Joseph YT Leung. *Handbook of scheduling: algorithms, models, and performance analysis*. CRC press, 2004.

[40] Ted G. Lewis, Brian J. Smith, and Marilyn Z. Smith. “Dynamic Memory Allocation Systems for Minimizing Internal Fragmentation”. In: *Proceedings of the 1974 Annual ACM Conference - Volume 2*. ACM ’74. New York, NY, USA: Association for Computing Machinery, 1974, pp. 725–728. ISBN: 9781450378505. DOI: 10.1145/1408800.1408893. URL: https://doi.org/10.1145/1408800.1408893.

[41] Chuanpeng Li, Chen Ding, and Kai Shen. “Quantifying the cost of context switch”. In: *Proceedings of the 2007 workshop on Experimental computer science*. 2007, 2–es.

[42] *Linux kernel man pages*. https://man7.org/linux/man-pages/.

[43] Miguel Masmano et al. “TLSF: A new dynamic memory allocator for real-time systems”. In: *Proceedings. 16th Euromicro Conference on Real-Time Systems, 2004. ECRTS 2004.* IEEE. 2004, pp. 79–88.

[44] Rami Matarneh. “Multi Microkernel Operating Systems for Multi-Core Processors”. In: *Journal of Computer Science* 5.7 (July 2009), pp. 493–500. DOI: 10.3844/jcssp.2009.493.500. URL: https://thescipub.com/abstract/jcssp.2009.493.500.

[45] Alastair JW Mayer. “The architecture of the Burroughs B5000: 20 years later and still ahead of the times?” In: *ACM SIGARCH Computer Architecture News* 10.4 (1982), pp. 3–10.

[46] Pedro Mejia-Alvarez et al. “Interrupt handling in classic operating systems”. In: *Interrupt Handling Schemes in Operating Systems* (2018), pp. 15–25.

[47] Mario Nemirovsky and Dean Tullsen. *Multithreading architecture*. Springer Nature, 2022.

[48] *Neovim*. https://neovim.io/.

[49] Bojan Novković and Marin Golub. “Improving monolithic kernel security and robustness through intra-kernel sandboxing”. In: *Computers Security* 127 (2023), p. 103104. ISSN: 0167-4048. DOI: https://doi.org/10.1016/j.cose.2023.103104. URL: https://www.sciencedirect.com/science/article/pii/S0167404823000147.

[50] Yoshinori Okuji et al. “The multiboot specification”. In: (Sept. 2023).

[51] *On-Demand paging*. https://en.wikipedia.org/wiki/Demand_paging.

[52] *OS Dev Wiki*. https://wiki.osdev.org/.

[53] *OS Dev Wiki*. https://wiki.osdev.org/Expanded_Main_Page/.

[54] Jihyun Park et al. “Memory corruption detecting method using static variables and dynamic memory usage”. In: *Proceedings of the 13th International Workshop on Automation of Software Test*. 2018, pp. 46–52.

[55] Greig Paul and James Irvine. “Take control of your PC with UEFI secure boot”. In: *Linux Journal* 2015 (2015), p. 1. URL: https://api.semanticscholar.org/CorpusID:54959221.

[56] Isabelle Puaut. “Real-time performance of dynamic memory allocation algorithms”. In: *Proceedings 14th Euromicro Conference on Real-Time Systems. Euromicro RTS 2002*. IEEE. 2002, pp. 41–49.

[57] *QEMU*. https://www.qemu.org/.

[58] Boqin Qin et al. “Understanding memory and thread safety practices and issues in real-world Rust programs”. In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2020, pp. 763–779.

[59] Bo Qu and Zhaozhi Wu. “Kernel experiment series for operating system course teaching”. In: *2011 IEEE International Symposium on IT in Medicine and Education*. Vol. 2. IEEE. 2011, pp. 12–16.

[60] Benjamin Roch. “Monolithic kernel vs . Microkernel”. In: URL: https://api.semanticscholar.org/CorpusID:17918683.

[61] Lina Sawalha, MP Tull, and RD Barnes. “Hardware thread-context switching”. In: *Electronics letters* 49.6 (2013), pp. 389–391.

[62] Ivan Stankov and Grisha Spasov. “Discussion of microkernel and monolithic kernel approaches”. In: *International Scientific Conference Computer Science*. sn. 2006.

[63] *Switching modes*. https://www.ibm.com/docs/en/aix/7.2?topic=performance-mode-switching.

[64] *The Rust Book*. https://doc.rust-lang.org/cargo/.

[65] *Theseus OS*. https://www.theseus-os.com/Theseus/doc/frame_allocator/index.html.

[66] Rich Uhlig et al. “Intel virtualization technology”. In: *Computer* 38.5 (2005), pp. 48–56.

[67] Aditya Venkataraman and Kishore Kumar Jagadeesha. “Evaluation of inter-process communication mechanisms”. In: *Architecture* 86 (2015), p. 64.

[68] Kiem-Phong Vo. “Vmalloc: A general and efficient memory allocator”. In: *Software: Practice and Experience* 26.3 (1996), pp. 357–374.

[69] *Writing an OS in Rust*. https://os.phil-opp.com/.

[70] Yves Younan, Frank Piessens, and Wouter Joosen. “Protecting global and static variables from buffer overflow attacks”. In: *2009 International Conference on Availability, Reliability and Security*. IEEE. 2009, pp. 798–803.

[71] Yuchen Zhang et al. “Towards understanding the runtime performance of rust”. In: *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 2022, pp. 1–6.

[72] Jianwen Zheng, Xiaochao Li, and Donghui Guo. “A novel multiboot framework for embedded system”. In: *2008 2nd International Conference on Anti-counterfeiting, Security and Identification*. 2008, pp. 344–347. DOI: 10.1109/IWASID.2008.4688422.

[73] Vincent Zimmer, Michael Rothman, and Suresh Marisetty. *Beyond BIOS: Developing with the Unified Extensible Firmware Interface, Third Edition*. 3rd. Berlin, DEU: De—G Press, 2017. ISBN: 1501514784.

8 Appendix

8.0.1 GitLab repository

The code for this project can be found here:

https://git.cs.bham.ac.uk/projects-2022-23/dxb834

8.1 Installing and running the system

After cloning the repository to your local system there are a few things that need to be installed and configured to run the system. Firstly, installing Rust is important, this can be done by curling in this command, or by using other package managers:

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

Following this, installing and running the bootimage tool will allow you to run the makefile in the root directory of the project. This will create a kernel image that can be used on physical hardware or booting in QEMU. To install QEMU you can head to this website https://www.qemu.org/download/. Currently, the system automatically tries to boot in QEMU using the cargo runner feature.

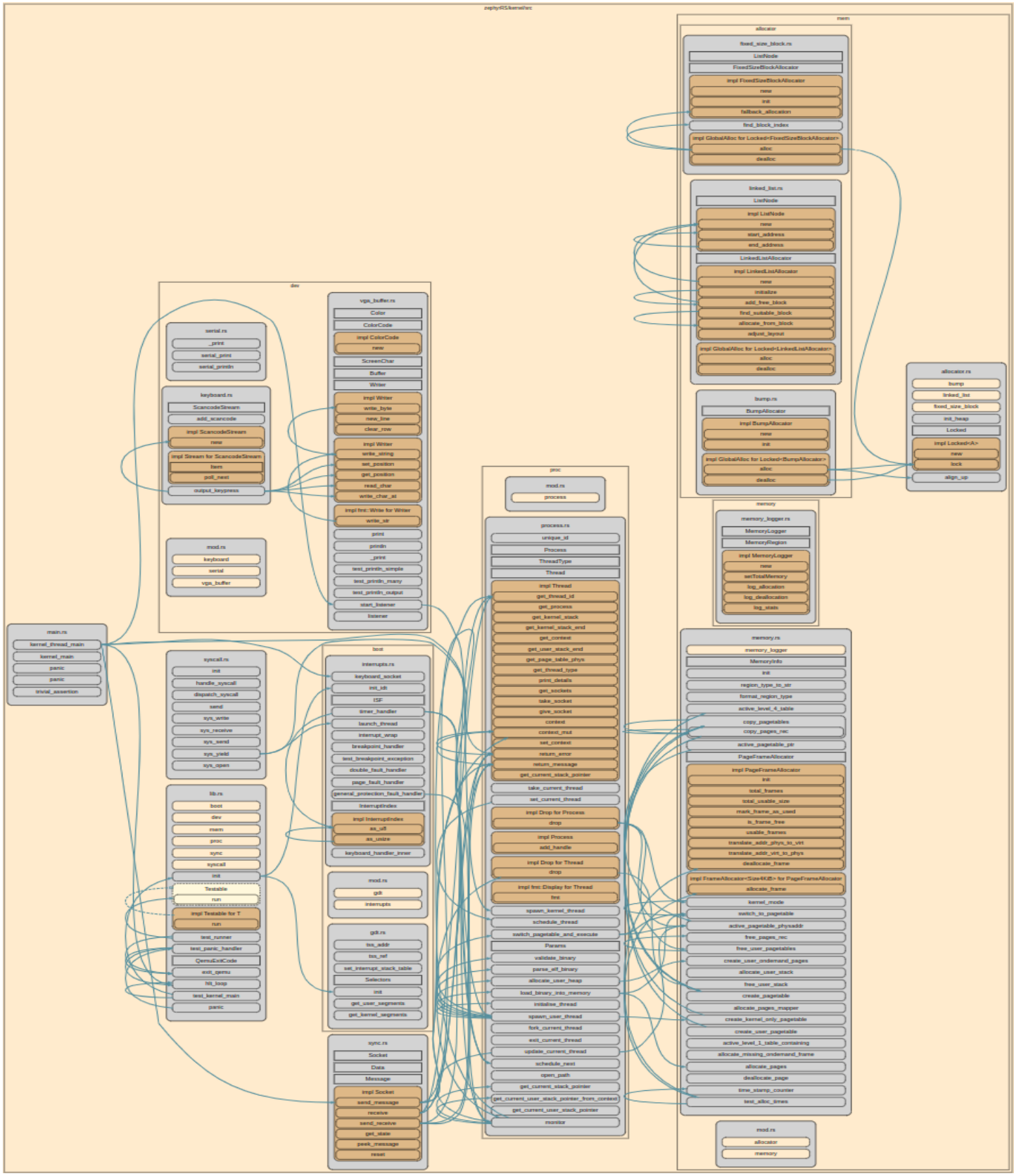
8.2 Code use

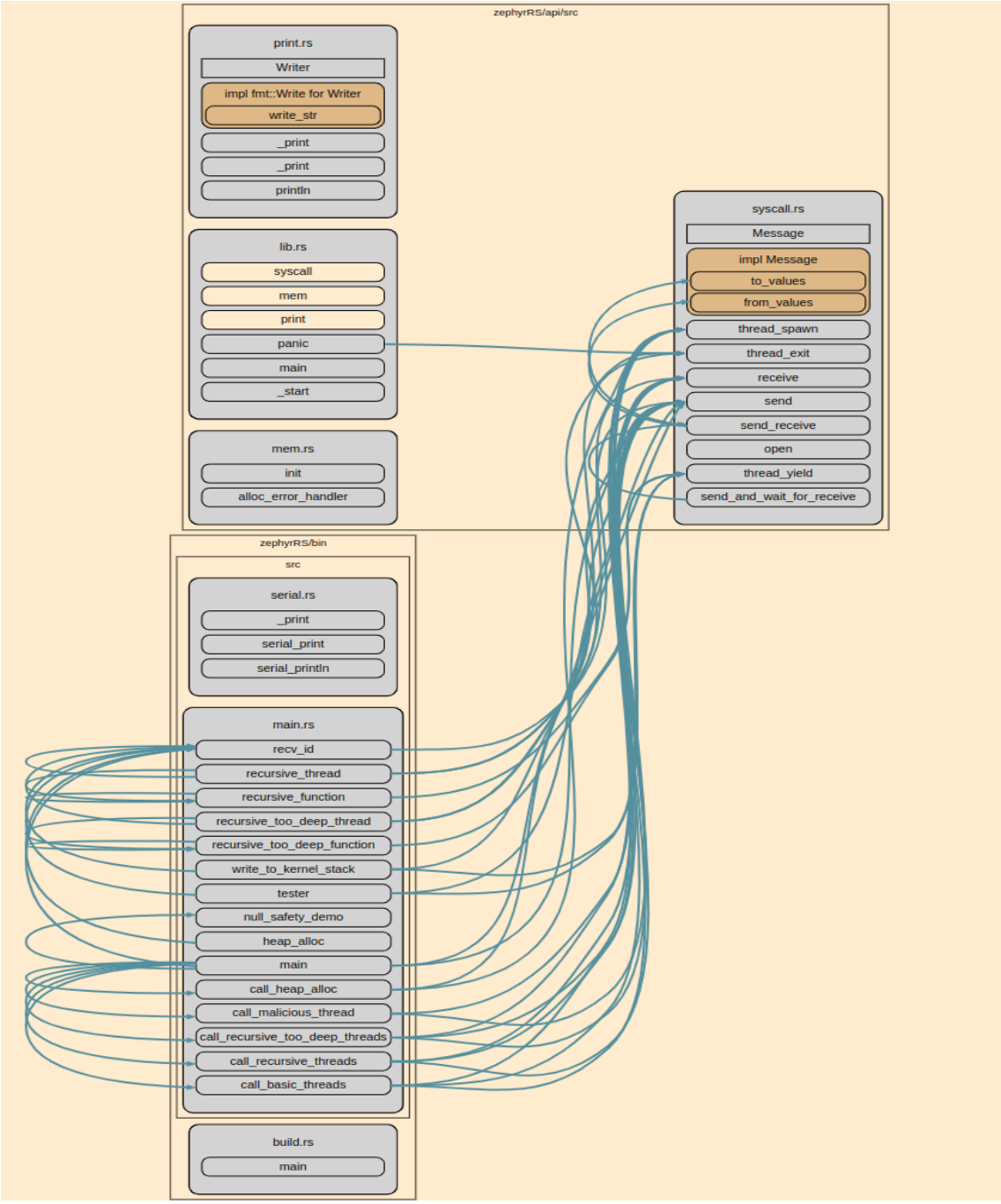
There were many resources that were instrumental in the development in this project. Flosse's comparison of open-source Rust operating systems served as a foundational reference for various aspects of kernel design. Kerla OS was instrumental in shaping the structure of the process module and offered valuable approaches to thread termination. Insights into heap allocator design and user space segregation were gleaned from Poplar OS, while MorosOS provided comprehensive knowledge on frame allocation, interrupt handling, and the intricacies of system calls, including CPU argument placement. FelixOS also contributed crucial guidance on system call management.

In addition to these specific projects, the Redox OS served as a comprehensive guide to constructing a scalable and maintainable Rust codebase, supported by extensive documentation. Lastly, the foundational resource "Writing an OS in Rust" was invaluable in this project's early stages, offering a thorough walkthrough of creating a freestanding binary, booting it in QEMU, and elucidating the merits of different memory allocators in Rust.

8.3 File structure

This project can be split into three main parts. The 'kernel; directory contains a 'src' folder that contains all of the different modules involved in making the kernel, including the memory management in memory.rs and process management process.rs, effort was made to name the files appropriately. The 'bin' folder in the root directory contains the user space programs. Each one of these contains their own Cargo package manager. The same can be said for the 'api' folder, this contains the user facing library that provides the system calls. The root directory defines the workspaces within this project, which is also used for building the project. There also includes some python files that were used to generate the graphs and images for the evaluation, as well as a script to filter the singular log file into seperate more managable files.


zephyrRS/kernel/src
mem
allocator
fixed_size_block.rs
ListNode
FixedSizeBlockAllocator
impl FixedSizeBlockAllocator
new
init
fallback_allocation
find_block_index
impl GlobalAlloc for Locked<FixedSizeBlockAllocator>
alloc
dealloc
linked_list.rs
ListNode
impl ListNode
new
start_address
end_address
LinkedListAllocator
impl LinkedListAllocator
new
initialize
add_free_block
find_suitable_block
allocate_from_block
adjust_layout
impl GlobalAlloc for Locked<LinkedListAllocator>
alloc
dealloc
bump.rs
BumpAllocator
impl BumpAllocator
new
init
impl GlobalAlloc for Locked<BumpAllocator>
alloc
dealloc
allocator.rs
bump
linked_list
fixed_size_block
init_heap
Locked
impl Locked<A>
new
lock
align_up
dev
vga_buffer.rs
Color
ColorCode
impl ColorCode
new
ScreenChar
Buffer
Writer
impl Writer
write_byte
new_line
clear_row
impl Writer
write_string
set_position
get_position
read_char
write_char_at
impl fmt::Write for Writer
write_str
print
println
_print
test_println_simple
test_println_many
test_println_output
start_listener
listener
serial.rs
_print
serial_print
serial_println
keyboard.rs
ScancodeStream
add_scancode
impl ScancodeStream
new
impl Stream for ScancodeStream
Item
poll_next
output_keypress
mod.rs
keyboard
serial
vga_buffer
proc
mod.rs
process
process.rs
unique_id
Process
ThreadType
Thread
impl Thread
get_thread_id
get_process
get_kernel_stack
get_kernel_stack_end
get_context
get_user_stack_end
get_page_table_phys
get_thread_type
print_details
get_sockets
take_socket
give_socket
context
context_mut
set_context
return_error
return_message
get_current_stack_pointer
take_current_thread
set_current_thread
impl Drop for Process
drop
impl Process
add_handle
impl Drop for Thread
drop
impl fmt::Display for Thread
fmt
spawn_kernel_thread
schedule_thread
switch_pagetable_and_execute
Params
validate_binary
parse_elf_binary
allocate_user_heap
load_binary_into_memory
initialise_thread
spawn_user_thread
fork_current_thread
exit_current_thread
update_current_thread
schedule_next
open_path
get_current_stack_pointer
get_current_user_stack_pointer_from_context
get_current_user_stack_pointer
monitor
memory
memory_logger.rs
MemoryLogger
MemoryRegion
impl MemoryLogger
new
setTotalMemory
log_allocation
log_deallocation
log_stats
memory.rs
memory_logger
MemoryInfo
init
region_type_to_str
format_region_type
active_level_4_table
copy_pagetables
copy_pages_rec
active_pagetable_ptr
PageFrameAllocator
impl PageFrameAllocator
init
total_frames
total_usable_size
mark_frame_as_used
is_frame_free
usable_frames
translate_addr_phys_to_virt
translate_addr_virt_to_phys
deallocate_frame
impl FrameAllocator<Size4KiB> for PageFrameAllocator
allocate_frame
kernel_mode
switch_to_pagetable
active_pagetable_physaddr
free_pages_rec
free_user_pagetables
create_user_ondemand_pages
allocate_user_stack
free_user_stack
create_pagetable
allocate_pages_mapper
create_kernel_only_pagetable
create_user_pagetable
active_level_1_table_containing
allocate_missing_ondemand_frame
allocate_pages
deallocate_page
time_stamp_counter
test_alloc_times
mod.rs
allocator
memory
main.rs
kernel_thread_main
kernel_main
panic
panic
trivial_assertion
syscall.rs
init
handle_syscall
dispatch_syscall
send
sys_write
sys_receive
sys_send
sys_yield
sys_open
lib.rs
boot
dev
mem
proc
sync
syscall
init
Testable
run
impl Testable for T
run
test_runner
test_panic_handler
QemuExitCode
exit_qemu
hlt_loop
test_kernel_main
panic
boot
interrupts.rs
keyboard_socket
init_idt
TSF
timer_handler
launch_thread
interrupt_wrap
breakpoint_handler
test_breakpoint_exception
double_fault_handler
page_fault_handler
general_protection_fault_handler
InterruptIndex
impl InterruptIndex
as_u8
as_usize
keyboard_handler_inner
mod.rs
gdt
interrupts
gdt.rs
tss_addr
tss_ref
set_interrupt_stack_table
Selectors
init
get_user_segments
get_kernel_segments
sync.rs
Socket
Data
Message
impl Socket
send_message
receive
send_receive
get_state
peek_message
reset


zephyrRS/api/src
print.rs
Writer
impl fmt::Write for Writer
write_str
_print
_print
println
lib.rs
syscall
mem
print
panic
main
_start
mem.rs
init
alloc_error_handler
syscall.rs
Message
impl Message
to_values
from_values
thread_spawn
thread_exit
receive
send
send_receive
open
thread_yield
send_and_wait_for_receive
zephyrRS/bin
src
serial.rs
_print
serial_print
serial_println
main.rs
recv_id
recursive_thread
recursive_function
recursive_too_deep_thread
recursive_too_deep_function
write_to_kernel_stack
tester
null_safety_demo
heap_alloc
main
call_heap_alloc
call_malicious_thread
call_recursive_too_deep_threads
call_recursive_threads
call_basic_threads
build.rs
main