Improving Low-Level Isolation of Containers: Leveraging Microkernel Design

B. R. P. Perera

2024

Improving Low-Level Isolation of Containers: Leveraging Microkernel Design

B. Ravin Perera
Index No : 19001126

Supervisor: Dr. C. I. Keppitiyagama
Co-supervisor: Mr. Tharindu Wijethilake

May 2024

Submitted in partial fulfillment of the requirements of the B.Sc in Computer Science Final Year Project (SCS4224)

Declaration

I certify that this dissertation does not incorporate, without acknowledgement, any material previously submitted for a degree or diploma in any university and to the best of my knowledge and belief, it does not contain any material previously published or written by another person or myself except where due reference is made in the text. I also hereby give consent for my dissertation, if accepted, be made available for photocopying and for interlibrary loans, and for the title and abstract to be made available to outside organizations.

Candidate Name: B. R. P. Perera

[illegible]

. . . .

Signature of Candidate Date: 29/09/2024

This is to certify that this dissertation is based on the work of B. R. P. Perera under my supervision. The thesis has been prepared according to the format stipulated and is of acceptable standard.

Co- Supervisor's Name: Mr. Tharindu Wijethilake

[illegible]

. . . .

Signature of Supervisor

Date: 29/09/2024

Tharindu

Signature of Co-Supervisor

Date: 29/09/2024

Abstract

The container is a concept in virtualization that groups code and dependencies into a single isolated unit. It leverages the operating system's kernel features to manage and run processes within its isolated environment. While current implementations offer seamless integration and enhanced performance, they do come with inherent limitations. The design and architecture of the kernel serve as a critical factor in enhancing key container characteristics, such as isolation, owing to its dependency on kernel functionality. While it affects the level of isolation provided within a container, it also affects the isolation provided among containers. A deeper understanding of container implementations underscores the importance of shifting the primary focus from containers themselves to the underlying kernel and its inherent strengths. The majority of contemporary container engines are optimized for monolithic kernels, which, by definition, prioritize performance over isolation. In contrast, microkernels are designed to provide higher levels of isolation at the cost of performance. It is important to explore how the capabilities of microkernels and the requirements of containers could interact to collectively find a stronger position in terms of isolation. This research investigates the potential of microkernels to emulate container environments, focusing on file system isolation and comparing performance against monolithic implementations. Leveraging GNU Hurd, based on the GNU Mach microkernel, the study employs Subhurd and translators to establish a container environment for evaluating performance and file system isolation against Linux-based containers.

Preface

In completing the requirements for my B.Sc in Computer Science Final Year Project (SCS4224), this research marks a significant step in my academic journey. This work is built on a thorough examination of existing literature, where I carefully reviewed relevant publications that contributed to the study. Following this, I engaged in the design and implementation stages, primarily relying on the web documentation of GNU Hurd. This section reflects my independent effort and technical skills in developing and executing the proposed solution, although I received assistance from the GNU Hurd community when encountering technical challenges. These efforts were greatly influenced by and connected to the web documentation of GNU Hurd, which aided in learning necessary concepts and commands. At the heart of this thesis are the original findings presented in the results section. Through extensive experimentation and analysis, I have provided new insights to the field, while properly acknowledging relevant research material.

Acknowledgement

I am deeply grateful to the University of Colombo School of Computing (UCSC) for providing me with the opportunity to pursue this research endeavor. It has been invaluable in both my academic and personal development. I extend my sincere appreciation to Dr. C. I. Keppitiyagama, my supervisor, and Mr. Tharindu Wijethilake, my co-supervisor at UCSC, for their expert guidance and unwavering support throughout this journey. Special thanks are also due to Samuel Thibault, Professor at Université de Bordeaux and a contributor to GNU Hurd, for his assistance and insights during critical moments. Lastly, I express my gratitude to my friends and family for their encouragement and support, which has been a constant source of strength.

Table of Contents

List of Figures

List of Tables

Chapter 1 - Introduction

1.1 Background

Virtualization is a technology that allows for the creation and operation of virtual instances or representations of computer resources, such as servers, operating systems, storage devices, or network resources. At its core, virtualization separates the logical or virtual layer from the underlying physical infrastructure [1]. This separation allows multiple virtualized environments to run on a single physical server, each with its own set of applications and software dependencies. Isolation is a fundamental concept in virtualization that ensures the separation and independence of these instances, preventing them from interfering with one another [2]. When discussing virtualization, it's important to talk about the two main technologies used in the process, Virtual Machines and Containers. Virtual machines are heavier in terms of resource usage and performance but provide better isolation for the environment by introducing a dedicated operating system and kernel to work with. Containers, on the contrary, are lightweight and provide a much smaller level of isolation utilizing the host operating system kernel features [3]. The expected level of isolation from a virtualized environment can vary depending on the specific virtualization technology and configuration used. However, while virtualization provides strong isolation between virtual instances, it's important to note that achieving absolute isolation is challenging, especially in containers due to the underlying hardware and software complexities involved such as process isolation, memory isolation, file system isolation, network isolation, resource isolation, etc [2]. Improving the isolation of containers to a level that is competitive with VMs while still retaining its lightweight nature can be a game changer. It may be a challenging feat, but making advancements in at least one of the mentioned areas can make a significant impact on the overall result.

1.1.1 Containers

Containers are a fundamental concept in virtualization, serving as encapsulated units that bundle together code and its dependencies. They leverage the capabilities of the underlying operating system's kernel to manage and isolate processes effectively. By utilizing features like namespaces and cgroups, containers can enforce restrictions on resource utilization such as CPU, memory, disk I/O, and network access [4]. This approach allows for the creation of virtualized environments tailored specifically to individual applications, ensuring that each container contains all the necessary components and libraries to run the software it holds. Despite the complexity of container technologies like Docker, their functionality ultimately relies on a few key kernel features. However, this dependency on the kernel means that the stability of the underlying operating system is crucial for maintaining the promised isolation and security of containers. Any vulnerabilities or faults within the kernel can potentially compromise the entire container infrastructure, leading to errors and issues across the system. Furthermore, the level of isolation provided by containers is inherently limited by the capabilities of the kernel itself [5]. In monolithic kernel architectures, where components are tightly interconnected, the risk of errors spreading between containers is heightened. A single error affecting the kernel could potentially impact multiple containers due to their coupled nature. This is where the concept of microkernel architecture comes into play. By decoupling components related to containers, microkernel architectures reduce the risk of errors propagating between containers. This architectural approach enhances reliability and isolation, addressing some of the limitations inherent in traditional monolithic kernel designs.

1.1.2 Microkernels

A microkernel is a minimalist operating system kernel that focuses solely on essential services. Most operating system functions are implemented as user-level programs, called "servers," running outside the kernel [6]. This design enhances isolation between components interacting with the kernel and increases process isolation. Microkernels align closely with container expectations, offering better isolation by decoupling processes and system services from the kernel. This decoupling enables containers to isolate themselves more effectively and eliminate interfering dependencies. Leveraging the architectural advantages of microkernels, if the kernel can isolate system services related to each container, container isolation could be significantly improved at a lower level.

1.2 Gap and Research Questions

The area of containerization has not greatly extended to the realms of microkernels and no standard implementation of containers is popular within the industry with respect to that particular kernel architecture. It is important to address and explore how the capabilities of microkernels and the characteristics of containers could interact to collectively find a stronger position in terms of container isolation. The research intends to fill that gap by initiating a discussion about their relationships and serve as a starting point to encourage conversation around the topic.

The research primarily revolves around exploring the following research questions.

- Can a microkernel's detached nature with its system services be utilized to enhance low-level isolation of containers?
- Does implementation of containers on a microkernel-based architecture suffer in terms of performance when compared to Linux-based containers?

1.3 Research Aim and Objectives

The research aims to investigate the potential of implementing containers on a microkernel to improve isolation using the architectural advantages and special characteristics provided by its kernel design. The research process intends to achieve the following objectives.

- Explore the architecture of microkernels and its support for containers
- Implement a primitive version of containers using features provided a microkernel based operating system
- Explore and analyze the functionalities provided by the kernel to isolate file system-related services for each container
- Measure the performance of containers and compare them with a monolith

based container implementation

1.4 Research Scope

The research will primarily cover the following tasks.

- Explore the topics of microkernels and containers
- Explore the relationship between the isolation mechanisms of microkernels and the isolation of containers
- Implement a primitive version of a container using features provided by GNU Hurd.
- Explore the capabilities of GNU Hurd to isolated the file system of each container.
- Analyze the overall isolation provided by the aforementioned setup with respect to the file system.
- Measure the performance of the aforementioned setup to analyze limitations
- Suggest future work and improvements to the topic

Implementing an entire container engine that could support the most commonly found features in platforms such as Docker can be extremely challenging. This is why the research aims to work with a setup that closely mimics the isolation provided by modern day containers. Containers tend to use many different kernel services other than file systems and analyzing all these services and how GNU Hurd handles isolation in each of these areas can be difficult and time-consuming. Due to the strict timeline of the research, the project will only focus on the file system of a container and any further improvements to design will be considered as future work.

1.5 Significance of the project

Achieving robust isolation within and among containers poses a significant challenge. While absolute isolation is elusive, advancements in areas contributing to isolation can pave the way for progress. This research focuses on enhancing file system isolation through microkernel technology, thereby bolstering container safety and resilience. Future advancements in this field hold promise for improving isolation in other critical container areas. Developers must prioritize designs that minimize kernel dependency to mitigate the risk of kernel exploitation, which could compromise all containers on a host. One significant drawback of microkernels is their overhead in Interprocess Communication (IPC). Upon conducting a performance comparison, the preference for monolithic architecture over microkernel architecture becomes evident due to the notable difference. However, considering the rapid pace of advancements in computing technology, there is reason to be optimistic about the future and the vast computational power that will be available. Many research endeavors exploring AI or higher-end cryptographic encryption standards operate under the assumption of expecting increased computational power in the future. Nonetheless, it's never too early to delve into these topics in detail. This pattern has been evident in the exploration of neural networks, developments in virtual reality, and similar research where theoretical advancements often preceded the availability of requisite capabilities to demonstrate their value. Yet, throughout the history of technology, one truth remains constant: superior architectures constrained solely by technological limitations eventually dominate once technology catches up. This has clearly been demonstrated by the meteoric rise of microservice architecture due to improvements in network latency. With an optimistic view regarding the future of computing, it's reasonable to anticipate that microkernels will establish themselves as a standard due to their architectural superiority. When that time arrives, containers and other virtualization technologies will need to adapt to the shift in architecture, and this research will serve as a solid foundation for moving in that direction.

1.6 Research Methodology and Evaluation Criteria

The research will revolve around the design science research methodology. Although the larger research aim encompasses a design problem (which will be discussed in much detail during the design section 3), it is embedded with multiple knowledge problems. As mentioned in [7], the process shall consist of three cycles. The relevance cycle which will take the identified problems, requirements, and evaluation criteria from the environment and pass them to the design science research. Then, the design cycle will generate the appropriate artifacts matching the provided requirements and evaluation criteria. The rigor cycle will add the newly found knowledge and artifacts to the knowledge base. The key components related to the three cycles are provided below.

- Stakeholders
  - Users of containers and virtualization
  - Researchers related to the area of research
  - Operating system developers
- Requirements
  - Improvements to isolation in containers at a lower level closer to the operating system's kernel
  - Identify the relationship between the inherent isolation provided by the design of microkernels and the isolation sought by containers at a lower level
  - Establish a discussion about the use of microkernels to improve isolation in virtualized environments
  - Evaluate and analyze the performance and isolation difference between monolith and microkernel implementations of containers
- Artifacts
  - A primitive version of container implementation utilizing the characteristics and features of a microkernel based operating system
  - An addition of comprehensive knowledge regarding the relationship between microkernels and container isolation with respect to isolation into the related field
- Evaluation criteria
  - The system shall be tested for performance and efficiency targeting the file structure using benchmarking tools
  - An analysis based on isolation will be conducted on the microkernel based container environment setup
- Additions to the knowledge base
  - A research paper providing a detailed explanation of the methods, find-

ings, and conclusions of the research

The research process shall also satisfy the design science guidelines as follows.

- Design as an artifact
  - The problem is well-defined and has a set of well-defined requirements. The artifact consists of a model and instantiation
- Problem relevance
  - The problem revolves around technology and has business relevance as discussed in the "Significance of the project" section.
- Design Evaluation
  - The methodology shall consist of strict evaluation criteria and shall use rigorous methods to maintain consistency and integrity
- Research contributions
  - The research will contain verifiable contributions and references where appropriate and will always give credit to relevant parties when deemed necessary
- Research rigor
  - Evaluations will always be conducted as accurately as possible
  - The methods used for evaluation will be transparent and would be reproducible
- Design as a search process
  - The research will undergo review once completed analyzing limitations and future work that could eventually improve the artifact to better meet the requirements of the stakeholders
- Communication as research
  - The research process and methods involved will be disclosed publicly through one or more research papers or other common mediums of research documentation at the end of the process to communicate the findings

A thorough explanation regarding the research design is provided in the design section 3.

Chapter 2 - Literature Review

2.1 Virtualization

2.1.1 What is virtualization?

Virtualization initially emerged in the 1960s as a solution to manage time-sharing systems. During this era, the cost of purchasing individual mainframe units for each user was prohibitive. As a cost-effective alternative, a single mainframe was shared among multiple users, with each accessing it during designated time slots. To safeguard user privacy, their sessions needed to be kept separate, a challenge that virtualization addressed effectively [8]. The widespread adoption of virtualization was primarily driven by industry needs rather than academic research. Hence, it is useful to define virtualization from an industry standpoint [9].
Some definitions:

- The separation of a service request from the underlying physical delivery of that service - VMWare [10].
- The abstraction of the computer hardware, that is, hiding the physical computer from the way in which it is used - Intel [11].

Broadly speaking, virtualization can be perceived as an abstraction that delineates the physical attributes of computing resources, creating a distinction between hardware and software components [9].

2.1.2 Characteristics of virtualization

While virtualization finds applications across various domains, its primary utilization is within the realm of cloud computing. This preference stems from its capacity to streamline the provisioning and management of computing infrastructure, thereby minimizing both cost and complexity [12]. Cloud computing encompasses an amalgamation of hardware, storage, networks, interfaces, and services, facilitating the provision of computing resources as a service. Within the domain of cloud computing, virtualization exhibits three principal characteristics [13].

1. Partitioning

A single physical system should be able to support various software systems by partitioning various resources

2. Isolation

   Each virtualized system should be able to work in isolation without affecting other systems or the host system. This should also isolate data among systems.

3. Encapsulation

   A virtualized system should be able to package the software system into a single file or bundle. This helps in portability and safety as each bundle is separate and avoids interference.

2.1.3 Types of virtualization

Historically, virtual machines utilizing hypervisors have been the predominant form of virtualization solution. However, advancements in software technologies over time have introduced a diverse array of alternative solutions, each emphasizing distinct aspects of virtualization [14]. Presented below are two of the most notable solutions in this regard.

1. Virtual Machines (Full virtualization)

   A virtual machine (VM) serves as a simulated representation of a physical computer system, commonly referred to as a guest, while the actual physical machine hosting it is denoted as the host. Unlike direct interaction with physical hardware, a VM requires mediation through a lightweight software layer known as a hypervisor to facilitate communication with the underlying physical infrastructure. The hypervisor assumes the responsibility of allocating physical computing resources such as processors, memory, and storage to each VM, ensuring their segregation to prevent interference between them. Employing a hypervisor on a physical computer or server, often termed a bare metal server, enables the separation of the operating system and applications from the underlying hardware. Subsequently, the physical machine can partition itself into multiple independent “virtual machines,” each capable of executing its own operating system and applications autonomously while leveraging shared resources managed by the hypervisor, including memory, RAM, and storage [15].

2. Containers (Operating System layer virtualization)

   This concept, also known as Single Kernel Image (SKI) or container-based virtualization, operates by concurrently executing multiple instances of the same operating system (OS). Consequently, the virtualization occurs at the level of the host OS rather than at the hardware level. All virtual machines (VMs) utilize an identical virtualized OS image, referred to as the virtual machine image herein. This streamlined approach simplifies system administration by enabling administrators to allocate resources such as memory, CPU, and disk space both during VM instantiation and dynamically during runtime. Operating system-layer virtualization proves to be more efficient than alternative virtualization methods, albeit lacking complete isolation. However, as VMs share the kernel with the host OS, compatibility necessitates that the guest OS matches the host OS, thereby precluding scenarios such as running Windows atop Linux [16].

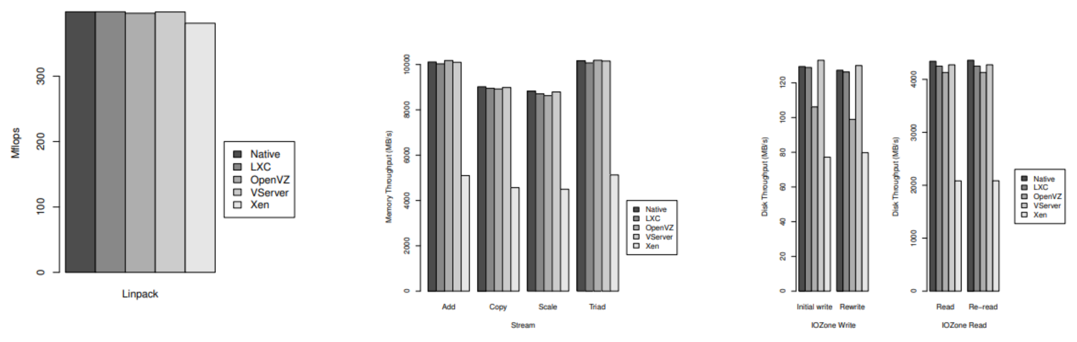
The classification of containerization, involving the utilization of containers, as a form of virtualization is a topic of ongoing debate within the community. However, upon closer examination of virtualization definitions, it appears to align with the overarching concept of virtualization. Increasingly, numerous researchers are acknowledging operating system-based virtualization, including containerization, as integral components of the broader virtualization paradigm [14][16][13].

2.1.4 Comparision between virtual machines and containers

Performance

Virtual machines (VMs) and containers represent two prevalent technologies utilized in the deployment and execution of applications. While both afford a degree of isolation and segregation between applications and the underlying host system, they exhibit disparities, particularly concerning performance. VMs furnish comprehensive isolation by simulating an entire operating system along with virtual hardware, thereby enabling each VM to execute distinct operating systems and application sets. Nonetheless, this isolation entails performance trade-offs as each VM encompasses an entire kernel alongside its associated processes, resulting in

substantial consumption of system resources and consequent performance degradation. Notably, the overhead becomes notably pronounced when multiple VMs operate on a single physical host.
Conversely, containers offer a lightweight form of isolation by leveraging the same operating system kernel as the host system. This design facilitates rapid startup and shutdown times, typically within seconds, in contrast to VMs which may take minutes. By virtue of sharing the underlying operating system, containers incur diminished overhead and utilize fewer system resources relative to VMs, thereby enhancing performance and resource efficiency. However, containers exhibit certain limitations in comparison to VMs. For instance, containers lack the capability to execute distinct operating systems or kernels, and they do not furnish complete isolation between the host and the container due to their shared kernel. Consequently, this shared environment poses potential security vulnerabilities and raises concerns pertaining to data privacy. Moreover, containers may prove unsuitable for applications necessitating stringent isolation or dependencies on specific hardware components [17]. The Figure 2.1 provides a comparison on performance between difference virtualization techniques.



(a) Computing performance using Linpack for matrices of order 3000.

(b) Memory throughput using STREAM.

(c) Disk throughput using IOZone.

Figure 2.1: Performance metrics [17]

Security

As previously discussed, virtual machines and containers present distinct security paradigms. VMs afford a robust level of isolation between the host system and the guest operating system housed within the VM. Each VM operates with its

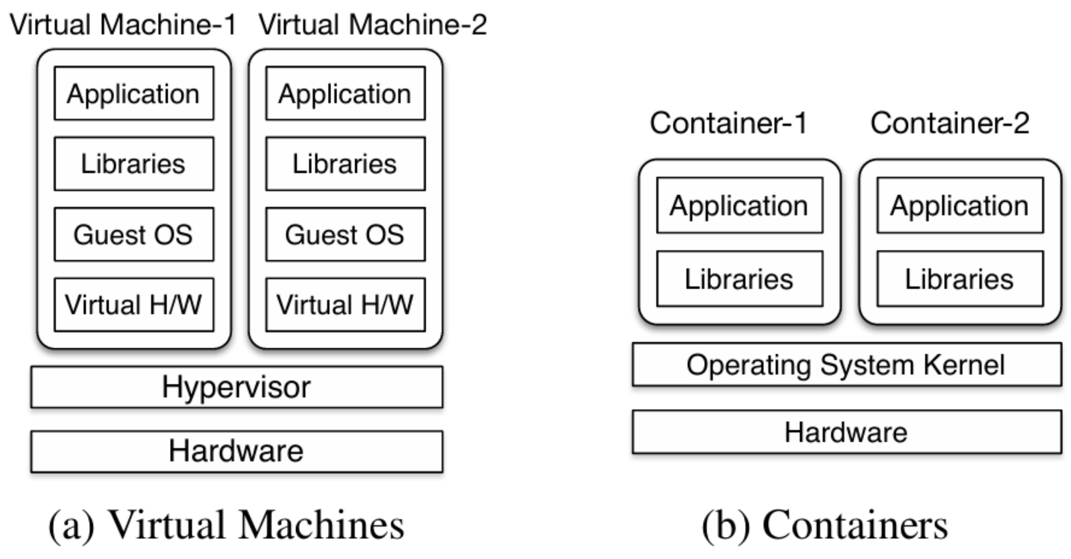


Figure 2.2: Implementation comparison [18]

own virtual hardware, encompassing network interfaces, disk drives, and assorted devices. This comprehensive isolation complicates unauthorized access for an attacker who has compromised one VM, thereby mitigating the risk of intrusion into other VMs or the host system. Additionally, VMs facilitate granular control over security policies, encompassing firewall configurations and access management.

In contrast, containers share the underlying operating system kernel with other containers and the host system. Although containers offer a degree of isolation, they do not match the security robustness of VMs due to their partial isolation from both other containers and the host system. The exploitation of a single container by an attacker can compromise the integrity of the entire system. Furthermore, as containers share the same kernel, any vulnerabilities at the kernel level can potentially impact all containers hosted on the system. Despite these limitations, containers utilize namespaces and cgroups to furnish rudimentary security measures. Namely, namespaces segregate processes and resources within containers, thereby limiting an attacker's access scope, while cgroups regulate CPU, memory, and disk I/O utilization to prevent resource monopolization. Additionally, Seccomp filters bolster security by thwarting system calls that could execute malicious code.

A notable advantage of containers pertains to their resistance against configuration drift, a phenomenon characterized by the stagnation and obsolescence of configurations due to prolonged environment uptime. Unlike VMs, which frequently encounter configuration drift owing to infrequent shutdowns, containers maintain configuration consistency more effectively. Despite their comparative security vulnerabilities, containers offer a lightweight and agile deployment approach for applications. Nevertheless, in scenarios involving sensitive data or critical applications, VMs remain the preferred choice due to their superior isolation and security assurances. Consequently, users often opt to deploy containers atop VMs to enhance security, even at the expense of anticipated performance levels [19]. The Figure 2.2 demonstrates the differences in both architectures.

2.2 Isolation

There isn't a widely agreed-upon definition for isolation in virtualized systems, but different authors tend to imply it in similar ways. Broadly speaking, isolation is the absence of the ability of a system A to interact with a different system B. This interaction could involve sharing needed information, stealing data, or negatively affecting the intended behavior of one or both systems, either intentionally or unintentionally. Hakamian suggests that a properly isolated virtualized system should be able to confine programs within themselves, preventing them from affecting other virtualized systems or the host system [20]. This explanation implies that interactions can occur through various means like interprocess communication, networking, or memory access. It's also evident that if a system A can have x such interactions and a similar system B only has y (y is a subset of x) such interactions, then system B is more isolated than system A.

2.2.1 Importance of isolation in virtualization

Isolation stands out as a paramount attribute of virtualization, as underscored by Popek and Goldberg's definition of a virtualized system as "an efficient, isolated duplicate of a real computer machine" dating back to 1974 [21]. With the evolution of technology and the proliferation of new virtualization methodologies, coupled with the escalating demand for virtualization, the significance of isolation has only amplified, particularly in domains such as security, performance, and fault propagation. Examining instances where systems have suffered compromise owing to inadequate isolation sheds light on the criticality of this aspect.

Security

Virtualized environments are susceptible to side-channel attacks, where information can be compromised through indirect means. For instance, when a guest VM utilizes a GPU in pass-through mode and is subsequently shut down, there exists a potential scenario wherein another guest VM may be launched and assigned the same GPU. Under certain conditions, remnants of data may persist in the GPU memory, allowing the new guest VM to access information left behind by a previous guest [22]. Additionally, in scenarios where multiple virtualized systems share common memory cache or hardware resources, one system may be capable of profiling these shared units to extract information pertaining to other systems. This vulnerability was notably exemplified in the well-known “Bernstein’s attack” on the AES (Advanced Encryption Standard) [23]. In addition to instances of data theft, there have been occurrences where a virtualized system gains unauthorized access to other virtualized systems by exploiting vulnerabilities and utilizing read commands [24]. Notably, attacks on DRAMs utilizing disturbance errors, known as rowhammer attacks, have been observed. These attacks can be leveraged to manipulate data stored in RAM, subsequently facilitating the alteration of executable programs residing in other virtualized systems [25].

Performance

Every virtualized system incorporates a manager within the host system to oversee the virtualized environments. If one virtualized system has the ability to disrupt the manager’s execution, it impacts all other virtualized systems. Likewise, if a virtualized system overwhelms the manager with tasks, it may impede crucial operations related to other virtualized systems. Another tactic involves excessive utilization of shared physical resources like CPU, memory, or I/O, thereby depriving other virtualized systems of efficient access to those resources [26].

Fault propagation

Faults are inherent in any system, but effective management necessitates their containment within the system where they occur. The phenomenon of transferring the repercussions of a fault from one system to other external systems is termed fault propagation [27]. Within a virtualized environment, fault propagation poses significant risks due to the extensive scale of such systems. A single host may host numerous virtualized systems of varying natures, some being malicious, others crucially requiring stability, and some less critical. The potential for a fault in one system to propagate to another system carries considerable implications, both disastrous and unforeseen. As previously discussed, fault propagation can occur either voluntarily (maliciously) or involuntarily. [28] notes the heightened vulnerability of post-quantum cryptographic algorithms, particularly those based on learning with errors problems, to such attacks. Resource sharing is a common scenario for fault propagation, where interruptions or compromises to shared resources by a single virtualized system render them unusable or inaccessible to other systems. [29] highlights a multitude of such faults pertaining to file systems.

Upon reviewing the aforementioned instances and considering the comprehensive definition of isolation, it becomes apparent that many of the discussed attacks stem primarily from insufficient isolation measures. Enhancements in isolation can bolster a system's resilience against such attacks, underscoring its significance for any system, whether virtualized or not.

2.2.2 Methods of improving isolation

Though various methods exist for enhancing isolation within a system, they ultimately revolve around fundamental concepts. If a system can manage these concepts effectively through certain mechanisms, it can bolster its isolation.

1. Reduce shared areas between systems [30]

   When two systems interact, potentially compromising isolation, minimizing the size of the interface available for interaction can enhance isolation. For instance, consider reducing the Trusted Computing Base. This concept is demonstrated in Figure 2.3

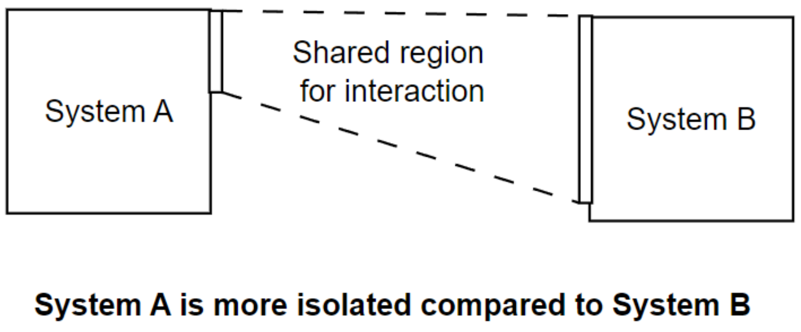

System A
Shared region for interaction
System B
System A is more isolated compared to System B

Figure 2.3: Reduce shared areas between systems

2. Diminish dependence among systems [30]

   When one system relies on another, its isolation is compromised. This interdependence may arise from reliance on availability, message passing for information exchange, or utilization of shared resources controlled by another system. This is demonstrated in Figure 2.4.

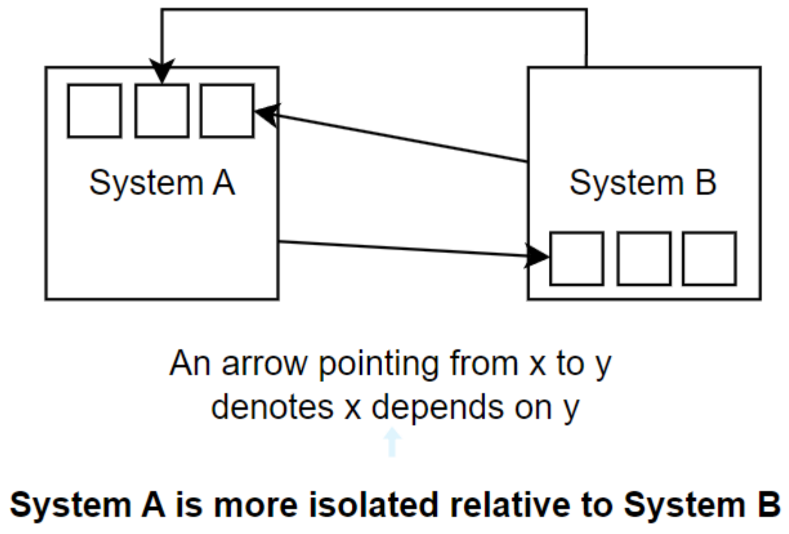

System A
System B
An arrow pointing from x to y denotes x depends on y
System A is more isolated relative to System B

Figure 2.4: Diminish dependence among systems

If a system succeeds in enhancing the factors mentioned above, it can be inferred that the system's isolation has technically been heightened.

2.3 Containers

The container is a concept in virtualization that groups code and dependencies into a single unit. A container uses the operating system's kernel features to separate processes and set restrictions on how resources like CPU, memory, disk I/O, and network are utilized. This makes it possible to create a virtualized environment at the application level. The containers consist of all the necessary dependencies and system libraries to run the software that it consists of. This is also true regardless of which environment the container is running on, either Windows or Linux [4].
The concept of containers embodies a framework that leverages operating system (OS) virtualization techniques to establish a virtualized environment. Unlike hardware virtualization, which focuses on virtualizing physical hardware, operating system virtualization entails the virtualization of the OS kernel. This kernel-level virtualization is pivotal for container abstraction, encompassing the allocation of CPU shares, memory, network I/O, and file system isolation to each container. Various allocation strategies, including dedicated, shared, and best effort, may be supported akin to hardware virtualization. In certain instances, the underlying OS kernel may emulate a distinct OS kernel version for processes operating within a container [18]. This functionality is frequently utilized to facilitate backward OS compatibility or to emulate diverse OS application programming interfaces (APIs), as evidenced in LX branded zones [31] on Solaris and in the execution of Linux applications on Windows [32]. Numerous OS virtualization techniques have been developed, including Solaris Zones, BSD-jails, and Linux LXC. The recent advent of Docker, a container platform akin to LXC but distinguished by its layered filesystem and additional software engineering advantages, has reignited interest in container-based virtualization for data centers and cloud environments [18].

2.3.1 Evolution of container technologies

While Docker and Kubernetes are often synonymous with containers, containerization is the culmination of numerous technologies and kernel features. A core objective of containerization is to achieve significant process isolation. This section explores the evolution of key technologies that provide this isolation. The

1979 release of Unix version 7 introduced “chroot,” allowing users to change the root directory for a process and its children to any directory within the filesystem [33]. This restricted process access to the file system beyond the designated “fake root” folder, introducing filesystem namespace isolation. However, isolation was limited to the file system structure, as processes still shared resources and network namespaces. In 2000, “FreeBSD Jails” addressed these limitations. Jails offered filesystem namespace isolation but also isolated the network namespace and userspace. This meant even if an isolated process gained root access, it could not affect processes and resources outside the jail [34]. The year 2001 saw the introduction of “Linux VServer,” a virtualization technology that utilized patched Linux kernels to add functionalities for limiting and isolating resource usage [35]. Around the same time, “OpenVZ” emerged, another virtualization technology offering similar functionalities with the addition of inter-process communication isolation [36]. These patches were later integrated into the official kernel between 2006 and 2013. Notably, user namespaces were introduced, allowing users within a specific namespace to create processes with privileges limited to that namespace [37]. In 2004, Solaris introduced “Zones,” a feature that grouped processes, allowing them to signal and observe only other processes within the same group. Zones also offered resource limitation and separate filesystem namespaces per group [38]. This concept later made its way into the Linux kernel as “Control Groups” (cgroups). Another area of development focused on adding functionalities to control and restrict system calls for enhanced isolation. The 2001 Linux kernel module “SubDomain” introduced access control functionalities for a selected number of system calls within processes [39]. Shortly after, SELinux (Security Enhanced Linux) was proposed, but its strict implementation led to rejection [39]. However, it inspired the creation of the Linux Security Module (LSM) framework, enabling the loading of security policies as kernel modules for a more generalized approach [40]. In 2005, ”seccomp” patches were added to the Linux kernel, limiting the number of system calls a process could make. While an improvement, it wasn’t sufficient. AppArmor, a system for dynamically filtering system calls using Berkeley Packet Filter (BPF), was later introduced [41].

The Rise of Modern Containers

Modern containers offer robust resource control and isolated environments. The 2008 Linux Containers (LXC) project combined key Linux kernel features like cgroups, namespaces, and capabilities to create a tool for managing and creating system containers. Research has explored improvements to LXC's file system isolation using SELinux policies and compared its resource isolation and management functionalities to those of Linux VServer and OpenVZ [42]. Docker, a popular container management platform launched in 2013, built upon LXC. Docker later developed its own implementation, libcontainer, as a substitute for LXC. Both LXC and Docker with libcontainer utilize the same underlying Linux kernel features (cgroups, capabilities, namespaces) and exhibit similar performance across various metrics, with LXC performing slightly better on random writes due to its use of a union filesystem [42]. Research has explored various aspects of container security, including dynamically generating AppArmor rules and security assessments based on attack goals. Docker's "runc" project, launched later, aimed to establish a vendor-neutral container runtime specification maintained by the Open Container Initiative (OCI). Runc utilizes the same isolation mechanisms (namespaces and cgroups) as Docker and supports security features like SELinux, AppArmor, and seccomp. It is noted to be more minimal compared to the Docker runtime [42]. In 2016, CoreOS and Docker merged their container image formats into a more vendor-neutral specification maintained by OCI. This standardization within the container ecosystem is expected to lead to improved interoperability and security [42].

2.3.2 Technology Stack of Containers

The container technology stack encompasses bare-metal hardware, underpinned by a host operating system (OS) comprising a kernel, and further augmented by a container engine operating atop the host OS, thereby enabling containerization. Containers, in essence, are processes within the host OS, yet they maintain isolation from each other through pivotal kernel features such as namespaces, cgroups, and capabilities [14]. A comprehensive examination of these kernel features alongside the container engine is warranted. The Figure 2.5 shows an indepth look at the architecture of a container stack.

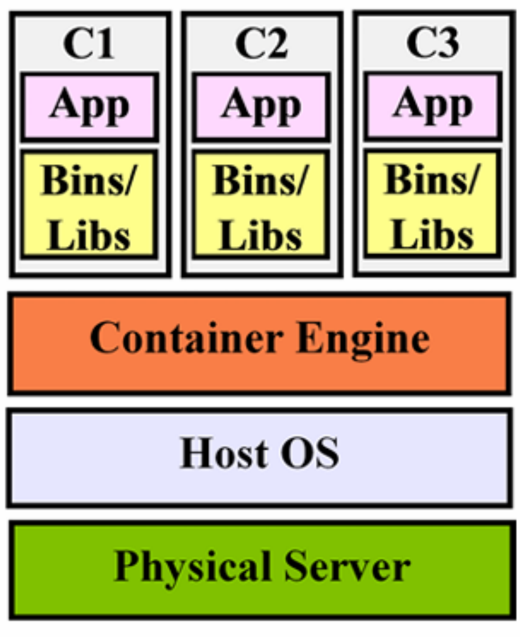


Figure 2.5: Container Technology Stack [14]

Container Engine

The container engine assumes the duties of creating, configuring, and deploying containers, while also overseeing their productivity and health. Serving as a mediator between containers and the underlying operating system, the engine provisions essential resources, including storage, network connectivity, and CPU and memory allocations, utilizing the capabilities of the underlying kernel. Well-known container engines comprise Docker, rkt, and LXC. Leveraging container engines empowers developers and operators to maintain a uniform and dependable performance of applications across various underlying infrastructures, facilitated by the container engine acting as an interface for system administrators [4].

Namespaces

“Namespaces” is a functionality offered by the Linux kernel that allows for the creation of isolated environments within the operating system. They essentially provide a way to partition system resources such as network interfaces and file systems making resources within a particular namespace only visible to the processes running in that namespace. Container implementations use this feature to provide a degree of isolation among containers while still running on the same host machine. Each namespace is associated with a particular type of system resource, such as network interfaces or process IDs, and is identified by a namespace name. When a new process is created within a namespace, it is only able to interact with the resources associated with the respective namespace. This makes it possible to create multiple containers on the same host, each with its own network namespace, file system namespace, and other resources [18].

The Linux kernel provides several types of namespaces, including:

1. PID namespaces - provides the illusion of a separate process ID space for each container
2. Network namespaces - provides a separate network stack for each container, including its own network devices, IP addresses, and routing table
3. Mount namespaces - provides a separate view of the file system for each container, allowing each container to have its own root file system
4. IPC namespaces - provides a separate inter-process communication (IPC) namespace for each container, so that processes running in one container cannot communicate with processes running in another container using IPC mechanisms such as System V IPC or POSIX message queues
5. UTS namespaces - provides a separate hostname and domain name for each container

Control Groups (Cgroups)

Cgroups are a feature in the Linux kernel that enables fine-grained control over system resources such as CPU, memory, and disk I/O. They allow processes to be organized into hierarchical groups, and limits can be set on the number of resources each group can use. This provides a way to ensure that one process or group of processes does not consume all the available system resources, which could lead to a degraded system or even a crash. The hierarchy of a cgroup is represented as a tree, with each node representing a resource control group. The top-level node is the root cgroup, which represents the entire system. Each child node in the tree can have child nodes, and so on, creating a hierarchy. Resource allocation is done using the notion of limits and quotas. A “limit” is a maximum amount of a resource that can be allocated to a particular cgroup, while a “quota” is the minimum. In the context of containerization, cgroups are a critical component for achieving resource isolation. Each container is associated with one or more cgroups that specify the resource limits for the processes running inside the container. This enables containers to operate as if they have their own dedicated resources, even though they are running on the same host as other containers and possibly other processes. For instance, a container can be assigned a specific amount of CPU time, memory, and network bandwidth, and any processes running inside the container will be limited to those resource constraints. This allows for the creation of multi-tenant environments where multiple containers can run on the same host without interference [43].

Capabilities

Linux capabilities constitute an integral component embedded within the Linux kernel, facilitating precise management of individual process privileges. Traditionally, Unix-like systems, including Linux, adhered to a binary privilege model, wherein processes were granted either complete root (superuser) access or restricted user privileges. This binary model posed substantial security vulnerabilities, as processes with root privileges could access all system resources and execute any operation.

The advent of capabilities heralded a transformative shift in this paradigm by enabling nuanced privilege allocation. Rather than endowing processes with unrestricted root access, capabilities allow for the tailored assignment of specific privileges tailored to the requirements of each process. This shift enhances security by limiting the potential attack surface and mitigating the repercussions of compromised processes.

Furthermore, capabilities facilitate adherence to the principle of least privilege, which advocates for granting processes only the essential privileges requisite for their designated tasks. For instance, instead of endowing an entire process with root privileges solely to bind to low-numbered ports, the capability to bind to ports can be selectively delegated. This approach ensures that the process operates with

diminished privileges, thereby minimizing associated risks [44].
The above section broadly speaks of the kernel features used in linux based containers as that is the most prominent form of containers used today [14].

2.3.3 Isolation in containers

When assessing the isolation capabilities of containers, it is evident that containers are not commonly perceived as highly isolated within the community. Google articulates this perspective in their blog, asserting, "A container isn't a strong security boundary. They provide some restrictions on access to shared resources on a host, but they don't necessarily prevent a malicious attacker from circumventing these restrictions." [45] Dan Walsh from Red Hat echoes a similar sentiment, stating, "Containers do not contain. Stop assuming that Docker and the Linux kernel protect you from malware." [46] Despite containers employing the aforementioned technology stack to furnish a substantial degree of isolation within a container, the potential impacts that could arise among containers, as discussed in the isolation section, are notably high. Enhancing isolation emerges as a pivotal strategy to fortify the containment capabilities of containers. Consequently, examining the implementation of containers underscores the pivotal role of the host kernel in determining the level of isolation provided to the container. Rather than seeking solutions that operate atop the host operating system, it is imperative to focus on improving the host kernel in terms of architecture and design.

2.4 Operating system kernels

The kernel constitutes the central component of the operating system, responsible for executing a myriad of crucial tasks. Operating at the highest privilege level within the system, the kernel undertakes essential operations such as scheduling, memory management, and inter-process communication, necessitating such elevated privileges for their execution. However, the specific functionalities embedded within the kernel are contingent upon the architectural choices and design principles adopted by the developer. Let us delve into examination of two prevalent kernel architectures [47].

2.4.1 Microkernel architecture

Microkernels, dating back to 1969, represent the initial kernel architecture [48]. A defining characteristic of microkernels is their streamlined design, focusing solely on essential functionalities such as inter-process communication, memory management, and scheduling. These functions operate within the kernel mode, while the remainder functions in the user mode. Unlike being implemented as a single extensive process, the microkernel is divided into multiple processes known as Servers. Ideally, only these Servers possess elevated privileges necessary for their designated tasks. Each Server operates independently within its own address space, ensuring separation from the system [49]. This modular approach prevents errors from affecting the entire system, as they are confined to the specific process in which they occur. Additionally, the modularization facilitates the seamless exchange of Servers without disrupting the entire system. However, the reliance on inter-process communication for communication introduces overhead compared to direct function calls, leading to increased context switches compared to a monolithic kernel. Consequently, these context switches result in notable latency, impacting overall performance negatively [6].

2.4.2 Monolith kernel architecture

In contrast to a microkernel, the monolithic kernel encompasses a broader array of functions. This includes numerous services operating in kernel mode, such as device drivers, dispatchers, schedulers, virtual memory management, all forms of inter-process communication (beyond simple IPC), and the (virtual) file system. System calls are also managed within kernel mode, with only applications executing in user mode. Unlike the divided nature of a microkernel, the monolithic kernel is implemented as a single process running within a unified address space. All kernel services operate within this shared address space, simplifying communication between them. This unified structure allows kernel processes to directly call all functions, akin to programs in user space. This capability facilitates improved performance and a simpler kernel implementation. However, a drawback is that a crash or bug in one module operating in kernel mode can potentially crash the entire system [49].

2.4.3 Comparison of isolation between architectures

The magnitude of monolithic kernel sizes has expanded exponentially over time. This growth stems from an inherent architectural limitation of monolithic kernels: to incorporate additional features, the kernel itself must expand. The accompanying figure 2.6 illustrates the progressive expansion of the Linux codebase over time. As the kernel size increases, so too does the prevalence of vulnerabilities. Consequently, reducing the Trusted Computing Base (TCB) of the kernel becomes crucial, allowing the most essential kernel components to remain compact. However, in monolithic architecture, the entire kernel constitutes the TCB, resulting in continual expansion [50]. In contrast, microkernels maintain a small footprint and tend not to experience significant growth over time. This characteristic enables microkernels to maintain a consistently small TCB. Moreover, a smaller kernel size facilitates formal verification, a process that has proven successful for microkernels like seL4 in eliminating vulnerabilities. This emphasis on a streamlined kernel, responsible solely for critical tasks and isolated from non-essential functions that can be delegated to user space, yields notable benefits. Research indicates that by formally verifying the kernel, vulnerabilities found in Linux can be significantly reduced in severity, with nearly 40% of vulnerabilities potentially eliminated altogether [50].

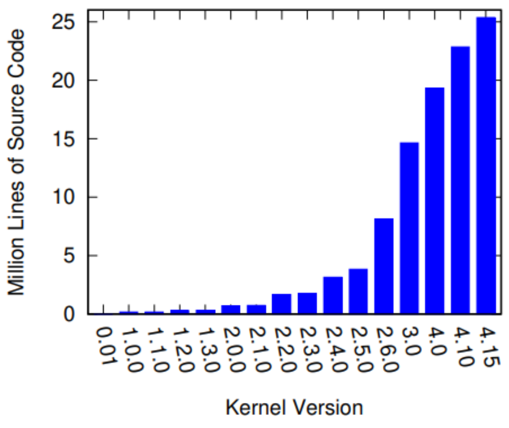


Figure 2.6: Linux kernel code size growth [50]

Microkernels demonstrate superior isolation, particularly in the realm of logical memory. In monolithic kernels, both kernel and user-level processes inhabit the same address space. This configuration poses a significant security risk, as a vulnerability within a user process could potentially grant access to the virtual memory of the kernel. Microkernels, on the other hand, adopt a distinct approach. They maintain separate address spaces for the kernel and user-level processes, establishing a barrier that mitigates the risk of unauthorized access. Communication between these distinct processes is facilitated through Inter-Process Communication (IPC) mechanisms [51].

Microkernels excel in isolating functionalities and access, ensuring that each user process is provided with a specific interface and a restricted set of functionalities. This prevents the compromise of one process from affecting a larger portion of the system. In the event of a single process failure, the kernel remains unaffected, and other processes continue to operate normally [51] [52]. In terms of access control, monolithic kernels typically employ coarse-grained mechanisms reliant on system calls and file system permissions to regulate process activities. Granting root access to a process effectively circumvents most security measures, providing unrestricted access to the entire system, including the ability to mount new file systems. In contrast, microkernels often utilize capability-based security. Under this framework, a process can only utilize a capability if explicitly granted. This capability-based approach enables finer-grained control over process activities [53]. Upon examining the distinguishing features of microkernels, it becomes evident that they prioritize isolation to a much greater extent compared to monolithic kernels, establishing a robust position in this regard. Nevertheless, design choices made by monolithic kernels offer support in alternative areas, such as enhanced performance.

Chapter 3 - Design

The study is separated into four important parts.

1. Selection of a suitable environment.
2. Setup a container-like system within the microkernel-based operating system.
3. Analyze the improvements in isolation provided to the file system of the container
4. Performance comparison between the two implementations.

3.1 Selection of a suitable environment

The research necessitates access to an operating system founded on a microkernel architecture. While numerous microkernels exist, locating a well-established operating system equipped with the necessary tools for configuring a container-like environment and conducting tests poses a considerable challenge. Among the array of microkernels available-L4, seL4, GNU Mach, Minix, to name a few-GNU Mach was selected as the preferred microkernel due to the existence of a notably matured operating system built upon it, known as GNU Hurd. Backed by Debian, GNU Hurd offers the convenience of Debian's robust package manager, "apt," and is fully integrated with the popular and powerful GNU toolset. These attributes position GNU Hurd as a promising choice for the research objectives at hand.
Now, let's delve into some key characteristics of GNU Hurd as outlined in its online documentation [54].

3.1.1 Important characteristics of GNU Hurd

Hurd is multi-server system

Several operating systems are built upon Mach, but they share the same drawbacks as a monolithic kernel because they are implemented as a single process operating atop the kernel. This single process delivers all the services typically offered by a monolithic kernel. This approach may not seem particularly logical, except for the potential ability to run multiple isolated single servers on the same machine.

Such systems are often referred to as single-server systems. However, the Hurd stands out as the sole practical multi-server system built on Mach. Within the Hurd, numerous server programs handle distinct services provided by the operating system. These servers operate as Mach tasks and communicate via Mach's message passing facilities. While each server may offer only a fraction of the system's functionality, collectively they construct a comprehensive and operational POSIX-compatible operating system. The Figure 3.1 shows the architecture of GNU Hurd.

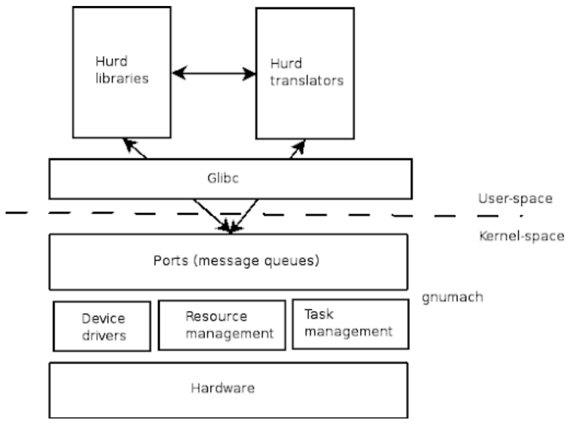


Figure 3.1: GNU Hurd architecture [55]

Mach ports

Inter-process communication within Mach operates on the concept of ports. A port serves as a message queue, facilitating one-way communication channels. Alongside a port, a corresponding port right (a type of capability) is required, which may take the form of a send right, receive right, or send-once right. Depending on the type of port right possessed, users can either send messages to the server, receive messages from it, or transmit a single message [52]. For each port, there exists precisely one task holding the receive right, while there can be zero or multiple senders. The send-once right proves beneficial for clients anticipating a response message. These clients can allocate a send-once right to the reply port along with the message. The kernel ensures that at some point, a message will be received on the reply port, which may involve a notification indicating that the server has relinquished the send-once right. Imagine the kernel queue as a reservoir for messages, capable of holding a substantial number of them. However, there's a bottleneck: when the queue hits capacity, any attempt to send new messages is put on hold until space becomes available. To mitigate this, there's a timeout mechanism in place to allow for interruption if needed. A receive right acts as a backstage pass to a specific queue, granting its holder the privilege to extract messages from it and even generate send rights. Conversely, send and send-once rights provide access to a queue, empowering the holder to add messages to it. In the case of a send-once right, only a single message can be inserted. Each addition to the queue through these rights is similar to invoking a unique capability. Ports, on the other hand, are similar to managed objects within the kernel's purview. They're meticulously guarded resources, requiring the appropriate port right for any interaction. Notably, ports are automatically relinquished when no corresponding port rights are held. Mach maintains a ledger of port rights allocated to each task. However, within a task, threads interact with ports using simplified local names without concerning themselves with the intricacies of port rights. Each task is allocated its own private cache of port rights, termed a port address space, ensuring a segregated context for managing port-related operations. Consider the scenario where a port send right is acquired. Using the associated port name, messages are dispatched to the port – be it one or multiple, depending on the nature of the right. These messages typically queue up for processing. When the recipient, say a server task, wishes to peruse its incoming messages, it exercises its port receive right, facilitating Inter-Process Communication (IPC). A noteworthy capability is the delegation of port rights within messages, similar to passing the baton to another party. Upon dequeuing such a message, the recipient inherits the associated rights. Message delivery adheres strictly to order, ensuring reliability. If messages 1 and 2 are dispatched sequentially, they're guaranteed to be received in the same order, despite potential interleaving by other threads. Due to their fortified nature, ports are globally unique, making them ideal candidates for system-wide references. This is exemplified in the Remote Procedure Call (RPC) system utilized by GNU Hurd, where methods are invoked on port-based references, facilitated by the MIG tool. It's crucial to note that invoking operations on a port doesn't entail ceding control to the recipient. Rather, it initiates an asynchronous process, similar to casting a message into the ether and awaiting a response. Particularly in RPC setups, including a reply port using a send-once right enables synchronization, allowing the sender to await the response, blocking on the receive port until it arrives. Understanding the message format is not necessary to utilize Mach IPC effectively. The Mach interface generator, mig, abstracts the intricacies of composing, sending, and receiving messages, presenting users with an interface that resembles a function call. In reality, the message could traverse a network to a server operating on a separate computer. The set of remote procedure calls provided by a server constitutes its public interface [56]. A figure 3.2 show the enqeueing and dequeuing process related to mach ports.

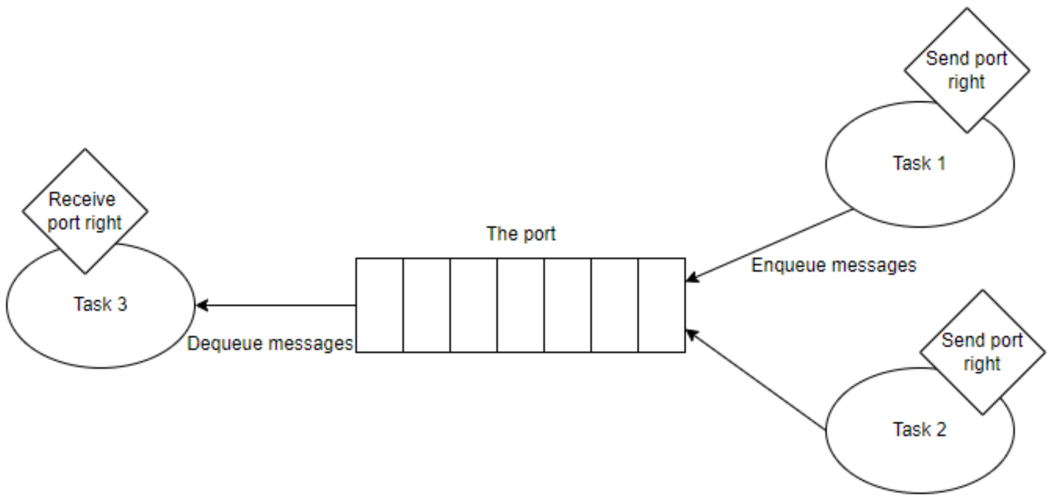


Figure 3.2: GNU Mach ports

Capabilities

When talking about Mach ports, we should also touch on capabilities in the context of GNU Mach and what they stand for.
A capability serves as a secure reference, combining both reference and protection attributes. Acting as a pointer to an object, it possesses safeguards against forgery, ensuring its integrity. In essence, a capability not only identifies the referenced object but also carries the authority to manipulate it. The fusion of designation and authorization within capabilities streamlines delegation processes [56]. Consider a scenario where program instance A intends to instruct program B to utilize a specific file for data storage. If A and B operate within distinct trust domains,

such as having different User IDs (UIDs), conveying solely the file's name to B poses risks. B must ascertain that granting access doesn't inadvertently empower A to manipulate the file under B's authority, thereby guarding against potential security breaches similar to the "confused deputy problem [57]." Moreover, without a contextual naming framework, a string sent by A to identify the file may resolve to an unintended object. By tethering designation and authorization inseparably, capability-based systems circumvent such dilemmas. Systems built upon capability-based architectures adhere to the principle of least privilege, ensuring that entities possess solely the permissions essential for their tasks. Typically, a capability mechanism finds its implementation within the software realm, often within the operating system kernel, particularly in microkernel architectures. The computational overhead associated with software-based implementations is minimal when compared to hardware alternatives [56].
In GNU Mach, a capability does not refer to a server, instead references a mach port. The capabilities allow the tasks to refer and perform actions on those ports [52].

Translators

Most servers are interacted with through file openings. Typically, when a file is opened, a server creates a port associated with that file, owned by the server overseeing the directory containing the file. For instance, a disk-based filesystem typically manages numerous ports, each representing an open file or directory. Upon file opening, the server generates a new port, links it to the file, and then furnishes the port to the requesting program. However, a file may have a translator affiliated with it. In such cases, instead of returning its own port pertaining to the file's contents, the server executes a translator program linked to the file. This translator receives a port to the actual file contents and is tasked with providing a port back to the original user to finalize the open operation. This mechanism is employed for mounting by associating a translator with each mount point. When a program accesses the mount point, the translator—often a program understanding the disk format of the mounted filesystem—is invoked and provides a port to the program. Once initiated, the translator need not be rerun unless it terminates;

the parent filesystem retains a port to the translator for subsequent requests. File owners can link a translator to a file without requiring special permissions, meaning any program can be designated as a translator. However, the system's functionality relies on the translator accurately implementing the file protocol. Nonetheless, the Hurd is designed so that the worst-case outcome is an interruptible hang. One approach to using translators is to access hierarchically structured data using the file protocol. For instance, the complexity of the user interface in the ftp program can be abstracted away. Users only need to recognize that a specific directory signifies FTP and can use standard file manipulation commands (e.g., ls or cp) to access the remote system, rather than learning a new set of commands. Similarly, a basic translator could simplify the complexity of tar or gzip. While transparent access might entail some additional overhead, it would offer convenience.

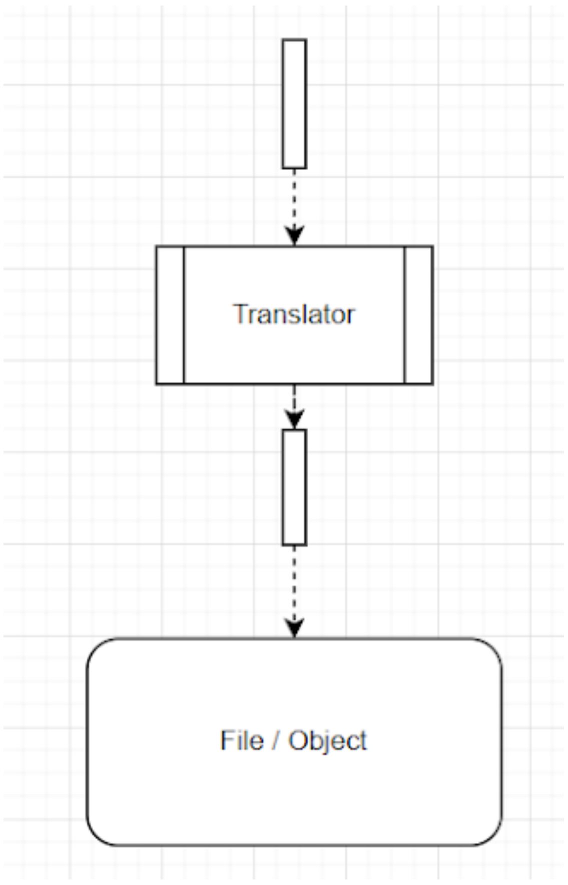
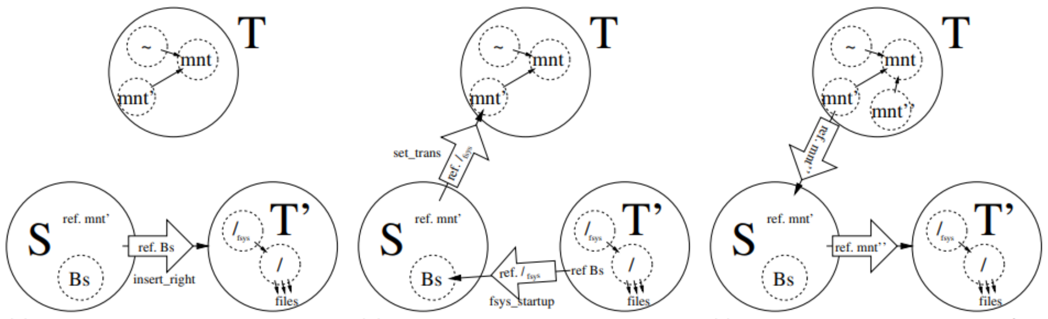


Figure 3.3: GNU Hurd translator

Translators are mounted for the file system using the `settrans` program. The following line will start the ftpfs translator and attach it to the node `~/mnt` and then pass the remaining arguments to the translator.

```
$settrans -a ~/mnt /hurd/ftpfs username:password@site.org/~
```

Figure 3.4 shows the underlying implementations of the `settrans` program. Initially, the `settrans` program acquires a handler to the associated mount point. Following this, it instantiates the program and, prior to initiating its operation, generates a port and inserts a send right to it into the bootstrap capability slot of the process. Subsequently, once this setup is complete, the translator is invoked. Upon detecting the bootstrap capability, the translator returns a reference to its `fsys` object to the `settrans` program. This reference will be essential for the parent object in the forthcoming discussion on the `dir_lookup` implementation. Following this, `settrans` proceeds to invoke the `file_set_translator` on the mount point while supplying the previously obtained `fsys` reference. Additionally, `settrans` acquires a reference handler for the mount point and transfers it to the translator program, which will serve as a reference to its parent if required in the future [52].



(a) `settrans` S has opened `~/mnt` and created the new translator instance, T′. `settrans` now inserts the bootstrap capability Bs into the bootstrap capability slot of T′. `settrans` starts T' running.

(b) Upon finding the bootstrap capability Bs, T′ invokes `fsys_startup`, passing a reference to its `fsys` object. S invokes `file_set_translator` to mount T′ on T.

(c) T installs the `fsys` object of T′. S obtains and returns a reference to an unauthenticated handle for the mount point to T′.

Figure 3.4: The `settrans` program implementation [52]

3.2 Setup a container within GNU Hurd

As GNU Hurd remains in active development and has yet to achieve widespread commercial adoption as a fully functional operating system, it lacks a pre-existing container implementation. This absence can be attributed to the limited interest in utilizing the microkernel architecture as a viable solution for containers. Consequently, prominent container platforms like Docker do not offer a readily available solution compatible with GNU Hurd. Given that Docker and similar platforms rely heavily on Linux-specific kernel features such as namespaces and cgroups—features not present or not functioning as intended in GNU Hurd—the straightforward porting of existing implementations is unfeasible. Consequently, to achieve container-like functionality within GNU Hurd, an environment closely aligned with the container concept must be constructed. The following section outlines the initial attempt to develop such a solution from scratch.

Before we delve deep into the proposed architectures for the experimental setup, it is important to understand some of the core components of GNU Hurd.

3.2.1 Essential components of GNU Hurd

In a Hurd system, you'll find at least the Mach kernel, along with an auth server, an exec server, a proc server, a password server, and a file system server. Let's explore each of these components in much detail as documented by the GNU Hurd Documentation and related resources [56].

File System Server

The file system server is one of the most crucial core servers of GNU Hurd as it implements two very important functionalities.

Function as the nameserver We've explored the significance of ports in Mach and their role as communication endpoints. But how exactly do we locate a port to a desired server? In Mach, a dedicated nameserver is essential for this task. A task can obtain a port to a server with send rights by querying the nameserver, assuming the intended server has already registered with the nameserver beforehand. However, in Hurd, there's no separate nameserver; instead, the file system server assumes this role. This setup is feasible because Hurd always maintains a root file system for POSIX compatibility. The file system server offers the `hurd_file_name_lookup` RPC to fulfill this function.

An example of the `hurd_file_name_lookup` is given below.

```
mach_port_t identity;
mach_port_t pwserver;
kern_return_t err;

pwserver = hurd_file_name_lookup("/servers/password");
err = password_check_user (pwserver,0 /* root */, "supass",&identity);
```

Pathname resolution **-** Most tasks interact with the kernel via the GNU C library (glibc) embedded within each task. This library not only facilitates communication through RPC but also ensures POSIX compliance, allowing developers to access kernel services seamlessly. One such service is the ability to acquire a port to a server, accomplished through a concept called “Pathname resolution.” However, the C library doesn’t maintain a comprehensive list of all available servers. As discussed earlier, glibc must communicate with the file system (acting as the nameserver) to traverse through all servers and locate the desired port. Path resolution doesn’t assume an implicit root; instead, it’s always relative to an explicitly referenced object. Typically, applications resolve most names relative to the capability stored in their root directory capability slot or their current working directory slot. These slots are usually populated by the parent process during process creation, effectively creating a single global namespace. Frequently, the dir lookup procedure doesn’t directly yield a capability pointing to the resolved object. Instead, it generates a new entity called a **handle**, which encompasses session state and points to the resolved object. Initially, the C library queries the root filesystem server about the provided filename, assuming the user intends to resolve an absolute path, using the `dir_lookup` RPC. If the filename corresponds to a regular file or directory on the filesystem, the root filesystem server simply returns a port to itself and notes the association with the specified file or directory. However, if a prefix of the full path matches a known server path, the root filesystem server returns a port to this server along with the remaining unresolved pathname. The C library then retries and queries the other server about the remaining path component. Eventually, the library either determines that the remaining path cannot be resolved by the last server in the list or successfully obtains a valid port to the desired server. The figure 3.5 demonstrates this process.

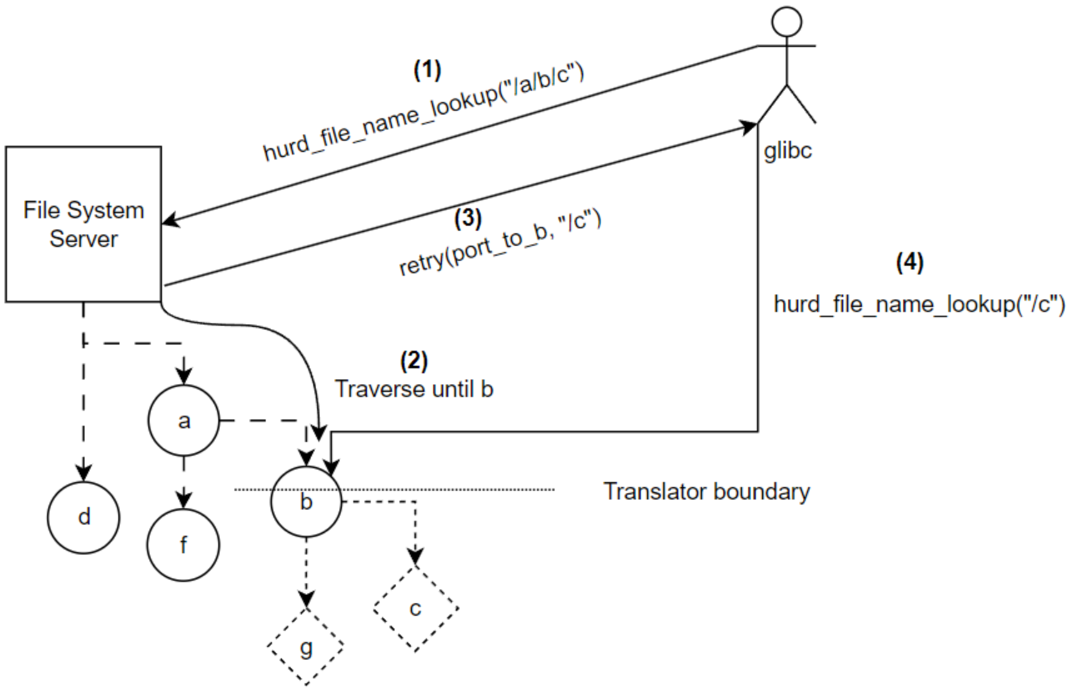


Figure 3.5: Pathname resolution

Function as a data source As previously discussed, the file system server handles pathname resolution and, when accessing a path, furnishes a port to the corresponding file (node). This approach offers significant advantages over a single nameserver. Firstly, it enables the utilization of standard Unix permissions on directories to restrict access to a server. By appropriately setting the permissions of a parent directory and ensuring no other means to obtain a server port exist, access can be controlled. However, the implications extend much further. Notably, a pathname doesn't directly point to a file; rather, it denotes a port of a server. This flexibility means that serving static data from a regular file is merely one option for the server. Alternatively, a server can generate data dynamically. For instance, a server linked with `/dev/random` could furnish fresh random data with every `io_read()` on its port, while one associated with `/dev/fortune` could deliver a new fortune cookie with every `open()`. While a conventional filesystem server serves data as stored on disk, other servers offer virtual information or a combi-

nation of both. The server is responsible for ensuring consistency and usefulness of data with each remote procedure call. Failure to do so may lead to results that diverge from user expectations and cause confusion. This mechanism facilitates the development of various applications, such as an NFS client or an FTP filesystem. For example, an FTP filesystem could seamlessly integrate FTP servers into the filesystem, allowing programs to access files using the standard POSIX file interface without needing to implement multiple network protocols.
Listed below are some of the file system translators that are currently implemented and undergoing development.

- Store based filesystems
  - ext2fs
  - ufs
  - isofs (iso9660, RockRidge, GNU extensions)
  - fatfs (under development)
- Network file systems
  - nfs
  - ftpfs
- Miscellaneous
  - hostmux
  - usermux
  - tmpfs (under development)

Auth Server

Identity-Based Access Control (IBAC) relies on user identity for access authorization. Therefore, when a subject seeks access to an object governed by such a system, it must reveal its identity to the object. In Unix, both the identity manager and most servers reside within the same trust domain. However, this differs in the Hurd environment, leading to a notable dilemma: while the server managing the object needs to inspect user identities, it must not be allowed to exploit them. To address this challenge, the Hurd's auth server facilitates a three-way handshake, enabling mutually distrustful collaboration and sharing without prior coordination. Each port to this server serves to identify a user and is associated with an ID block, encompassing sets of effective user IDs and group IDs, as well as available user IDs and group IDs (any of which may be empty).

The auth server provides three key services:

1. Boolean operations on a given set of authentication ports, resulting in a third port representing the union of user and group IDs.
2. The root user (UID 0) can generate any authentication port.
3. A set of Remote Procedure Calls (RPCs) to establish identity and exchange information among users.

When a service seeks to authenticate a user, it communicates with its designated trusted auth server. If the user is associated with a different auth server, the transaction will be unsuccessful. This necessitates a framework where all users are compelled to utilize the same auth server, ensuring that the interface is structured to accommodate any secure operation. In the event of possessing two identities, they can be merged to request an identity comprising the unions of the sets from the auth server. Additionally, it is possible to create a new identity consisting solely of subsets of an existing identity. However, expanding sets is restricted unless the user holds superuser privileges, indicated by possessing the user ID 0. However, no restrictions are imposed; any user can develop a program implementing the authentication protocol and it among their processes.

Three-way handshake The three-way handshake is used to establish a trusted connection between the user and the server. The protocol is as follows.

- Step 1: A user desires a server to be aware of its IDs.
- Step 2: The user initiates a reauthentication request to the server, including a rendezvous port.
- Step 3: Both parties convey this port to the authentication system. The user employs `auth_user_authenticate`, while the server utilizes `auth_server_authenticate`. Additionally, the server provides a new port to the authentication system.
- Step 4: The authentication system correlates these two requests and matches rendezvous ports .
- Step 5: The user obtains the new port (provided by the server) from the authentication system, and the server receives information regarding the user

thereby enabling the server to gain insight into the user's IDs.

The figure 3.6 shows the above process in action.

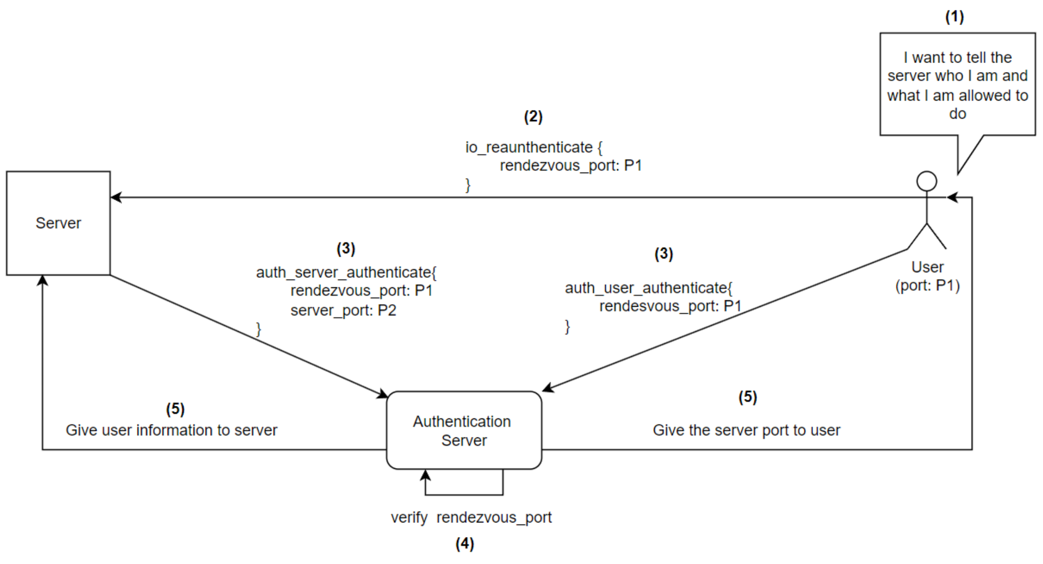

(1)
I want to tell the server who I am and what I am allowed to do
(2)
io_reaunthenticate { rendezvous_port: P1 }
Server
User (port: P1)
(3)
auth_server_authenticate{ rendezvous_port: P1 server_port: P2 }
(3)
auth_user_authenticate{ rendesvous_port: P1 }
(5)
Give user information to server
Authentication Server
(5)
Give the server port to user
verify rendezvous_port
(4)

Figure 3.6: Three-way handshake

Password Server

The password server operates with root privileges and issues a fresh authentication port upon receipt of a Unix password. The IDs associated with the authentication port align with the Unix user and group IDs to maintain POSIX compliance. Additionally, support for shadow passwords is integrated within this system. Situated at `/servers/password` and operating as root, the password server exchanges Unix passwords with the auth server, authenticating them against the password or shadow file. Various utilities leverage this server, thereby obviating the need to manage passwords directly, a task that necessitates privileged access like `setuid`.

Proc server

The process server serves as a central repository for organizing system information, playing a crucial role in maintaining system integrity. While its use is not mandatory, opting out means sacrificing the POSIX-like appearance of the Hurd system. This server offers four primary services: Firstly, it manages essential host-level data not handled by the Mach kernel, such as the hostname, hostid, and system version.

Secondly, it facilitates POSIX functionalities by maintaining sessions and process groups. Thirdly, it establishes a direct mapping between Mach tasks and Hurd processes, assigning each task a unique process ID (pid). Processes can register message ports with the server, accessible to any requesting program. Additionally, the process server allows processes to disclose their current command-line arguments and environment variables, enabling ps-like programs and facilitating information manipulation. Moreover, the server supports process collections for managing multiple process message ports simultaneously. It also provides mechanisms for converting between pids, process server ports, and Mach task ports while ensuring port security. While the default system process server is unavoidable, users have the flexibility to run additional process servers with non-superuser privileges to implement specific features as needed.

Exec server

The exec server plays a crucial role in program execution. Execution of a program happens through the `execve` call (similar to Linux). The implementation of this call is divided among three components. Firstly, the library handles the marshaling of argument and environment vectors, then sends a message to the file server containing the program to be executed. The file server checks execute permissions and performs any necessary modifications during the exec call. For instance, if the file is marked `setuid` and the file server has the capability, it may adjust the user identification of the new image. It also determines whether programs with access to the previous task should retain access to the new one. If permissions are being augmented or if an unreadable image is being executed, the exec operation must occur in a new Mach task to uphold security. Once the policy for the new image is established, the file system invokes the exec server to load the task. Utilizing the Binary File Descriptor (BFD) library, this server loads the image, supporting a wide range of object file formats. Additionally, it handles scripts starting with `#!` by executing them through the specified program. Moreover, the standard exec server examines the environment of the new image. If the environment contains a variable `EXECSERVERS`, it utilizes the programs specified there as exec servers instead of the default system ones. However, this is not applied to execs requested to remain secure by the file server. Upon initiation, the new image commences operation within the GNU C Library, which communicates with the exec server to obtain arguments, environment, umask, current directory, etc. None of this additional state is exclusive to the file or exec servers; programs have the flexibility to utilize it differently if desired.

Now that the necessary background concepts have been reviewed, let's explore the designs for the test setup in detail.

3.2.2 Design 1

As detailed in the literature review, the concept of containers originated with BSD jails, aiming to confine processes within a designated part of the file system using chroot, followed by additional kernel features to enhance process isolation and impose restrictions [34]. Similarly, the idea of implementing such isolation in GNU Hurd can be considered. As previously discussed, the file system server in GNU Hurd plays a pivotal role, particularly in pathname resolution, effectively serving as the authority governing namespaces. By closely examining the chroot program in Hurd, it becomes evident that the concept of namespace isolation is already ingrained in the system. For POSIX compatibility, a `dir_lookup` implementation is required to resolve the special name dot-dot (..) strictly at the parent directory and not within a translator. Consider the scenario depicted in Figure 3.4. When a process invokes `dir_lookup` on the capability denoting `home/mnt`, with dot-dot specified as the path for resolution, the `ftpfs` instance responds with a retry message. This message contains a capability denoting the `/home/mnt` object on the parent translator, with the path rewritten as dot-dot. However, the traditional Unix `chroot` mechanism mandates that a directory appears as a root to a group of processes, altering the meaning of dot-dot. To address this, Hurd offers the more versatile `file_reparent` mechanism, which resolves this issue without necessitating superuser privileges. This mechanism creates a unique handle where lookups of dot-dot lead to the provided capability instead of its parent, signified by a special void capability indicating the directory's root appearance. Handles derived from a reparented node maintain this property, ensuring that lookups initiated from a reparented node remain at the highest possible level in the hierarchy [52]. With this implementation, chroot in Hurd now mirrors its usage in a Linux environment, effectively establishing a new namespace for the process and its child processes, confining them within a jail-like environment. All that remains is to set up a new file system server, `chroot` into it, and designate the new file system translator as the root node for the subsequent executed process. This server will manage all file system operations for that process and its descendants. Similar to a typical jail setup, all associated programs and libraries must be relocated into the jail to ensure proper functioning of the processes. Since many core components of GNU Hurd operate in user space, files related to servers and translators must also be moved into the jail. Tools like Debootstrap can facilitate this task. The initial design, as illustrated in Figure 3.7, demonstrates this concept. However,

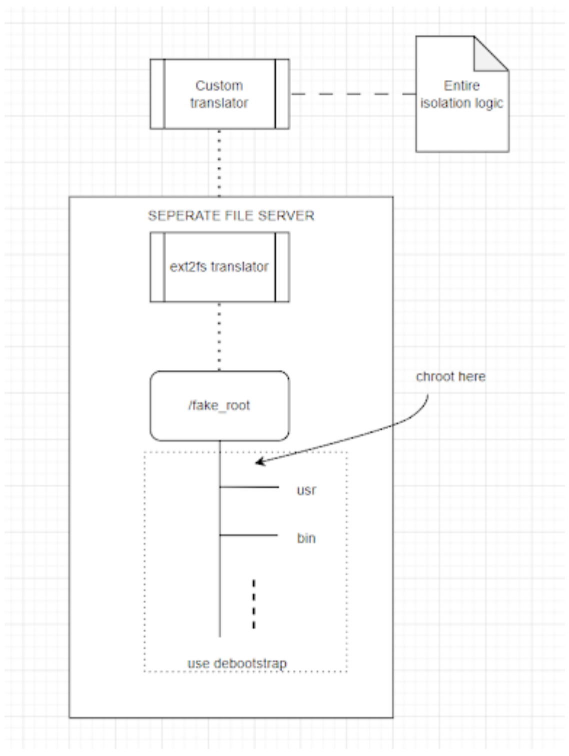


Figure 3.7: Initial architecture for design 1

the process remains non-isolated in terms of other processes and resource usage. Nonetheless, this issue could be addressed by launching new instances of the proc, auth, password, and exec servers. Given that these servers can operate alongside existing ones, this approach poses no challenge for the host environment. Thus, instead of implementing isolation measures externally (as depicted in Figure 3.7), a new set of instances can be initiated from within the jail. This approach should prevent processes within the jail from accessing the outer host environment. This revised architecture is illustrated in Figure 3.8.

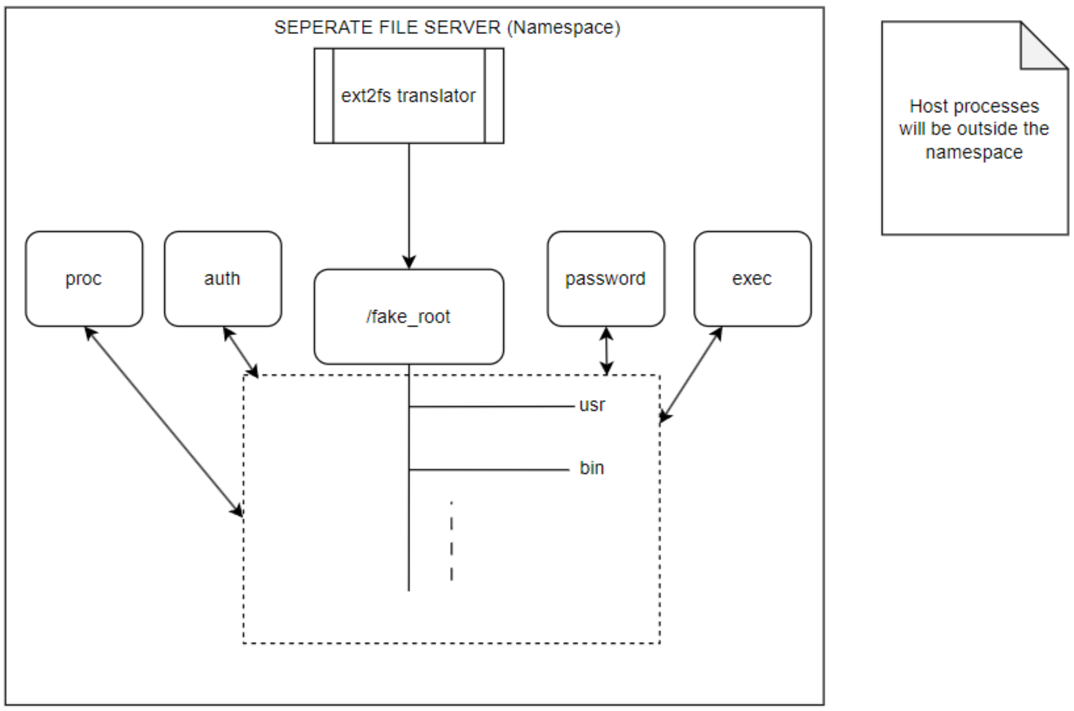

SEPERATE FILE SERVER (Namespace)
ext2fs translator
proc
auth
/fake_root
password
exec
usr
bin
Host processes will be outside the namespace

Figure 3.8: Revised architecture for design 1

Although this revised design provides resource isolation, GNU Hurd lacks an effective resource monitoring system to enforce resource constraints. Consequently, this design cannot replicate the functionality of control groups found in Linux.

Unfortunately, the proposed design encountered several challenges during implementation. The anticipated level of process isolation was not achieved as expected. Moreover, configuring shared resources like packet forwarding interfaces proved to be cumbersome. Given that the research primarily focuses on file system isolation, the complexity of managing network isolation seemed disproportionate to the research objectives. A comprehensive discussion on the implementation and the encountered challenges will be provided in the implementation section 4. Consequently, due to these reasons, further implementation of the design was paused as efforts were redirected toward exploring better alternatives.

3.2.3 Design 2

Subhurds represent a concept within the Hurd operating system designed to accommodate the execution of multiple logical systems within a single kernel. The concept arises from the necessity to debug the Hurd servers and boot process while operating within a running Hurd system. This approach enables the user to place the test program or boot code within a subhurd and then monitor it through the host system using tools like ps and gdb. This approach also circumvents potential deadlocks that may arise when attempting to debug a server instance that halts during debugging (Ex: Proc server). Furthermore, Subhurds can be utilized to debug the primary Hurd system itself, including its root filesystem, although requiring a privileged Subhurd for such tasks. This bidirectional debugging capability contributes to the overall robustness and maintainability of the Hurd operating system [56]. To initiate a Subhurd, the boot process is triggered by executing a command similar to boot `/dev/sd1s1`, prompting the loading of various components such as the `ext2fs.static` filesystem handler and Hurd servers. This sequence ultimately results in the establishment of a Subhurd environment, facilitating user login. The functionality of Subhurds hinges on the capability to run the Hurd operating system unaltered within its confines. Upon booting, the `/bin/boot` utility instigates a secondary set of Hurd servers, effectively creating a setup reminiscent of the stock GNU Mach configuration. Notably, the `/bin/boot` utility responsible for initiating Subhurds is impressively compact, comprising a mere 2,606 lines of source code, inclusive of the bootscript parser and various stubs. This efficiency underscores the streamlined nature of the Subhurd architecture and simplicity of starting a Subhurd.

Although initially intended for streamlined debugging purposes, upon closer examination, Subhurds exhibit characteristics similar to containers. Notably, each Subhurd spawns a fresh instance of its core servers upon booting, rendering them distinct and isolated entities from one another while still sharing the host kernel. This isolation lends itself well to utilization as containers in our context. Subhurds boast individual proc servers, file system servers, and other associated servers, thereby affording both file system and process isolation. These attributes are scrutinized and validated in the implementation phase. Owing to these inherent characteristics, Subhurds emerged as the optimal environment for implementing a container-like solution on GNU Hurd. Most importantly, Subhurds also address the challenges encountered in Design 1 regarding process isolation and network configuration. The overall architecture of a Subhurd is given in Figure 3.9

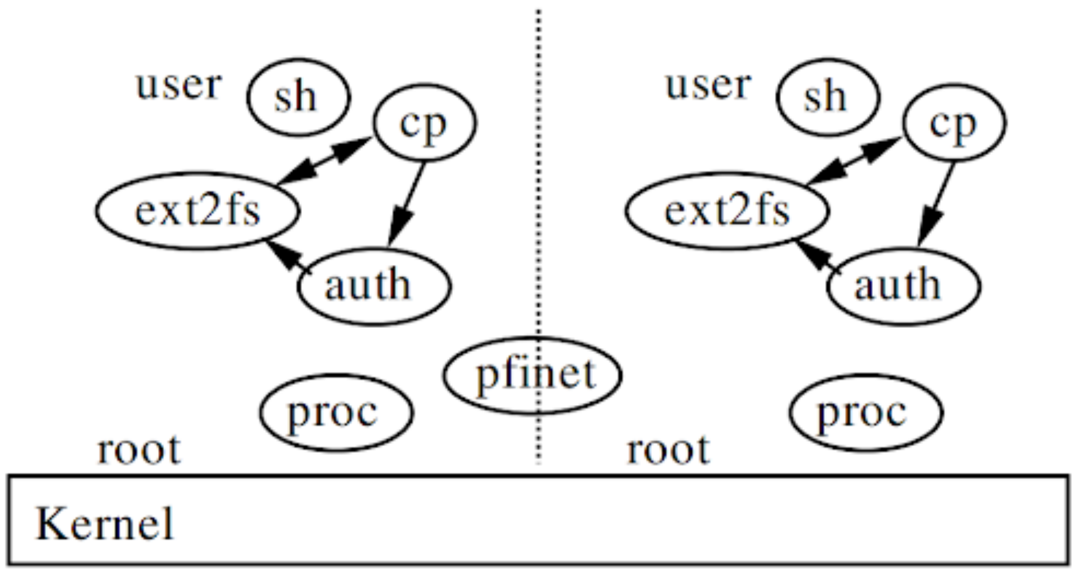


Figure 3.9: Architecture of a Subhurd

3.3 Isolation Analysis

The third phase of the study will delve into the pivotal aspect of the research, which involves assessing the enhancements in isolation provided by microkernels on container file systems. While microkernels inherently offer extensive isolation across their systems, our examination will concentrate solely on file systems. Employing the definition of isolation and methodologies for enhancing it (discussed in the literature review section 2.2), we will scrutinize how Subhurds exhibit superior isolation compared to conventional container implementations, owing to the characteristics of their underlying microkernel. The analysis will delve deeply into the functionality of GNU Hurd servers, elucidating their roles and how Subhurds leverage the microkernel architecture to enhance isolation. Specifically, we will explore how the implementation reduces shared regions among components and diminishes dependencies among systems—two key factors determining the degree of isolation, as discussed in the literature review 2.2.

In monolithic architectures, a significant drawback lies in the integration of file system logic within the kernel. This entails that if a container manager induces a kernel crash due to a file system logic bug, all containers will cease to function. However, microkernels relocate this logic to user space via servers, thereby insulating the kernel from such failures. To test this resilience, we will conduct an experiment by concurrently running two Subhurds and observing how a crash in one file system server affects the other.

3.4 Performance comparison

This section conducts a series of fundamental benchmarking tests to assess the performance of CPU, memory, and I/O within the implemented environment. These tests will be duplicated within a Docker container, and the outcomes will be closely analysed for comparison.

Chapter 4 - Implementation

4.1 Test Environment Setup

Since GNU Hurd doesn't operate on a Linux kernel, it's not feasible to switch kernels within a Linux distribution. To run GNU Hurd, one must either utilize a dedicated system or employ a virtualization platform like VMWare, Virtualbox, or QEMU. Running GNU Hurd directly on hardware is discouraged due to inadequate driver support for most commercially available hardware. Hence, virtualization becomes the sole option. QEMU was selected as the virtualization platform for its flexibility and endorsement by the official Debian guide [58]. Combining QEMU with KVM typically offers better performance compared to the Windows native alternative. Therefore, QEMU was run on WSL2 to leverage KVM capabilities. Although there's a slight performance drop due to WSL2 usage [59], it doesn't significantly affect the comparison, as performance benchmarks related to the monolith implementation will be conducted in Docker, which also utilizes WSL2 underneath. The performance of GNU Hurd will indeed be affected by the use of QEMU, but this impact must be overlooked because there are no better alternatives available. Details regarding the device hardware and software versions (relative to WSL2) are provided below. Table 4.1 shows information regarding the Memory, Table 4.2 about CPU and Table 4.3 about WSL.

QEMU information

```
QEMU emulator version 4.2.1 (Debian 1:4.2-3ubuntu6.27) Copyright (c) 2003-
2019 Fabrice Bellard and the QEMU Project developers
```

	total	**used**	**free**	**shared**	**buff/cache**	**available**
Mem:	7627	1299	2543	4	3784	6031
Swap:	2048	1	2046			

Table 4.1: Memory information

Attribute	**Value**
Architecture	x86_64
CPU(s)	12
Thread(s) per core	2
Core(s) per socket	6
Model name	AMD Ryzen 5 4600H with Radeon Graphics
Virtualization	AMD-V
Hypervisor vendor	Microsoft
Virtualization type	full
L1d cache	192 KiB
L1i cache	192 KiB
L2 cache	3 MiB
L3 cache	4 MiB

Table 4.2: CPU information

Attribute	**Value**
WSL version	2.1.5.0
Kernel version	5.15.146.1-2
WSLg version	1.0.60
MSRDC version	1.2.5105
Direct3D version	1.611.1-81528511
DXCore version	10.0.25131.1002-220531-1700.rs-onecore-base2-hyp
Windows version	10.0.22631.3447

Table 4.3: WSL and Ubuntu (running in WSL) Information

Instead of compiling from source, the GNU Hurd prepared CD image was chosen for its convenience. The `20230608` release was utilized. The following command enables booting into the Hurd system using QEMU.

```
sudo qemu-system-x86_64 --enable-kvm -hda debian-hurd-20230608.img
    -m 4G -net nic -net user,hostfwd=tcp:127.0.0.1:2222-:22
```

The `--enable-kvm` option enables KVM support in QEMU, optimizing it for kernel-based virtualization. A memory limit of 4GB was set, following the recommendation in the official guide, and port 2222 was selected for establishing a TCP connection to access the system securely via SSH through the WSL terminal. Upon booting up the system, users could log in as either root or demo. The demo user was chosen to evaluate the capabilities of a regular account, though it was added to the sudo group to perform privileged actions. To update the system, newer sources were added to the `/etc/apt/sources.list` file, and the system was upgraded using the apt package manager. To access the latest Debian repositories, the following sources were added.

```
deb http://deb.debian.org/debian-ports unstable main
deb-src http://deb.debian.org/debian unstable main
deb http://deb.debian.org/debian-ports unreleased main
```

4.2 Implementing Design 1

With a functional GNU Hurd system in place, the implementation of Design 1 can commence. The initial and crucial phase involves achieving namespace isolation. To accomplish this, a new file system server must be established. This can be done utilizing the `mke2fs` program in Hurd, which generates an ext2 file system. However, before proceeding, an empty block file is necessary to serve as the logical data store. The command below demonstrates how to create a block file of approximately 1GB in size.

```
dd if=/dev/zero of=./container_storage bs=1M count=1000
```

This command generates a 1GB block filled with zeroes. Subsequently, this block file can be employed to establish an ext2 file system. The subsequent command utilizes the block storage file as input to transform it into an ext2 file system. It allocates group tables, writes inode tables, superblocks, and other metadata as necessary.

```
sudo mke2fs container_storage
```

With a correctly created ext2 file system in place, a translator is necessary to access this storage. GNU Hurd employs the `ext2fs` translator located in `/hurd/ext2fs` to perform the translation between the block storage and the process. Additionally, this translator handles the crucial pathname resolution on this node. The following command creates and binds an `ext2fs` translator to the block file.

```
    sudo settrans -c container /hurd/ext2fs $PWD/container_storage
```

Executing the following command will associate the `container_storage` with a container node (creating it if it's absent due to the `-c` argument), and then embed an ext2fs translator within it. Subsequently, any action carried out on the container (e.g., cd) will be interpreted by the translator and implemented on the `container_storage` accordingly. For example, the `cd` command could be utilized to navigate the block file as if it was mounted to the file system via the container node. Ensuring that the location of the block file given to the `settrans` command is an absolute address is crucial for the translator to function correctly.

Now that there's a valid file system linked to a node and a functional translator executing a file system server, the subsequent step involves transferring the essential dependency files for the container's operation into the file system. As discussed in the design section 3, unlike a typical jail in a Linux environment where relocating essential programs from the `/bin` directory and their associated dependencies from the `/lib` directory suffices to enable program execution within the jail, this is not the case in GNU Hurd. In Linux, most critical functionalities reside within the kernel and are thus shared with the jail. However, in Hurd, these functionalities are predominantly located in user space as servers. Manually identifying the crucial server programs and locating their dependencies is exceedingly time-consuming. Therefore, a tool named Debootstrap comes into play for this task.

Debootstrap is a versatile tool crafted to install a Debian base system into a subdirectory of an existing system, obviating the need for an installation CD by accessing a Debian repository. It can also operate from a different operating system, allowing users to install Debian onto a separate partition from a running system, such as Gentoo. Furthermore, it facilitates the creation of a rootfs (root file system) for a machine with a different architecture, a process known as "cross-debootstrapping [60]." Since GNU Hurd is supported by Debian, the Hurd base image is available in their repositories.

Despite Debootstrap boasting a plethora of functionalities, only the capability to download a base image from a repository URL will be utilized. According to web documentation, Debootstrap has been successfully ported to function on GNU Hurd provided proper sources are provided in the sources.list file. As per the documentation, the following command is intended to access the Debian repositories and retrieve the base image into the jail (`container` directory), with the target set to `sid` to prompt Debootstrap to fetch the stable release from various other alternatives [58].

```
sudo debootstrap sid container
```

However, the Debootstrap process fails displaying the following error.

```
E: Invalid Release file, no entry for main/binary-hurd-i386/Packages
```

Examining the Debootstrap log found in
`$PWD/container/debootstrap/debootstrap.log` didn't yield significant insights either. It merely shows how the process abruptly terminated.

```
...
gpgv:  using RSA key 4CB50190207B4758A3F73A796ED0E7B82643E131
gpgv:  Good signature from "Debian Archive Automatic Signing Key
(12/bookworm)   <ftpmaster@debian.org>"
```

The Debian Keyring stores the OpenPGP keys of Debian Developers, who have full upload privileges to the Debian archives. Debootstrap verifies these keys when examining the Release file. In case Debootstrap encountered difficulties accessing the correct Release file from the repository due to a keyring issue, the keyring were installed using the following command.

```
sudo apt install debian-ports-archive-keyring
```

Upon further investigation, a review of a previous Google Spaces conversation between two Hurd developers revealed that the parameters passed to Debootstrap needed to be updated as follows.

```
sudo debootstrap
    --keyring=/usr/share/keyrings/debian-ports-archive-keyring.gpg
    --extra-suites=unreleased sid containers
    http://deb.debian.org/debian-ports/
```

It was unfortunate that this was not updated in the documentation.
Now, the Debootstrap process successfully acquired the Release file, fetched dependencies, and downloaded the required packages. Following the download process, Debootstrap proceeds to install the packages, which occurs in three stages.

1. Installing the core packages
2. Configuring the core packages
3. Installing the base packages

Debootstrap smoothly proceeded through the initial two stages but encountered a blocker due to a dependency error in the `cron-daemon-common` package, which is closely related to the cron package. Cron, a widely-used program in UNIX environments, is responsible for scheduling tasks based on time. Upon analyzing the Debootstrap logs, it became evident that `cron-daemon-common` relies on `systemd` and its associated packages—`systemd-standalone-sysusers` and `systemd-sysusers`. However, these components are specific to the `systemd` software suite, primarily utilized in Linux-based systems. Given that GNU Hurd doesn't incorporate `systemd` in its implementation, it not possible to access these packages within Hurd. Consequently, excluding the installation of this package (`cron-daemon-common`), along with its dependencies, seems feasible, as providing the required dependencies (`systemd`) isn't an option. Debootstrap offers an argument (`--exclude`) precisely for this purpose.

```
sudo debootstrap
    --keyring=/usr/share/keyrings/debian-ports-archive-keyring.gpg
    --extra-suites=unreleased --exclude=cron,cron-daemon-common sid
    containers http://deb.debian.org/debian-ports/
```

However, despite the exclusion of the package from the list, Debootstrap disregarded the exclusion. This behavior seemed to be an unintended bug within the program. To address the issue, a thorough investigation into the inner workings of Debootstrap was necessary. It was discovered that when given a specific target, the program attempts to execute a predefined script file located in the `/usr/share/debootstrap/scripts` directory. Upon noticing the script related to the target `sid` was the debian-common script, which contains commands for installing base packages, it was observed that the script employs `dpkg` for package installation within the designated directory. Given that `dpkg` offers an argument to ignore dependencies (`--ignore-depends`), the script was modified accordingly to disregard the related dependencies during the installation of any packages.

```
..
....
p; smallyes " |
in_target dpkg --force-overwrite --force-confold --skip-same-version
--ignore-depends=systemd,systemd-standalone-sysusers,systemd-sysusers
--install $(debfor $predep) base=$(without "$base" "$predep")
....
..
```

However, this approach proved futile as the program persisted in attempting to install the packages related to `cron` and continued to fail due to missing dependencies. A final effort was made to resolve the issue by employing an alternative method. Instead of downloading the packages from the repository each time, Debootstrap offers the option of downloading the packages into a tar file. This `tar` file can then be utilized later for package installation instead of fetching them directly from the repository. To implement this, the program necessitates an empty directory for temporarily storing the downloaded content (`temp`) and a designated location to store the packages (`hurd_base.tar`). The command to execute this functionality is as follows.

```
sudo debootstrap --verbose
    --keyring=/usr/share/keyrings/debian-ports-archive-keyring.gpg
    --extra-suites=unreleased --make-tarball=$PWD/hurd_base.tar sid
    temp http://deb.debian.org/debian-ports/
```

Next, the downloaded packages can be used in the installation process with the following command. Additionally, the command includes the `--exclude` argument, which is configured to disregard the `cron` and `cron-daemon-common packages`.

```
sudo debootstrap --verbose
    --keyring=/usr/share/keyrings/debian-ports-archive-keyring.gpg
```

```
        --unpack-tarball=$PWD/hurd_base.tar --extra-suites=unreleased
        --exclude=cron,cron-daemon-common sid container
```

This approach resolved the issue, allowing all three stages to be completed successfully without encountering any problems.

With all the dependencies inside the jail, chroot should be able to initiate a shell within it. Chroot will also trigger the `file_reparent` mechanism to set up the node for dot-dot pathname resolution, effectively establishing an isolated file system namespace within the jail. The following command can be invoked to run the chroot program on the container node.

```
    sudo chroot container
```

This grants the user access to a shell capable of executing processes with file system isolation. Nonetheless, the new jail merely contains a minimal base image of Hurd and requires updating to acquire the latest packages and utilities essential for configurations (e.g., `ifconfig`, `nano`). However, upon examination, it was discovered that the jail lacked necessary components for an update, including sources, certificates, and the keyring. Given the absence of essential tools such as a text editor, the sources were manually appended through the terminal as outlined below.

```
echo "deb http://deb.debian.org/debian-ports unstable main" | tee -a
    /etc/apt/sources.list
echo "deb-src http://deb.debian.org/debian unstable main" | tee -a
    /etc/apt/sources.list
echo "deb http://deb.debian.org/debian-ports unreleased main" | tee -a
    /etc/apt/sources.list
```

The required certificates were transferred from the outer environment to the jail using the following command.

```
    sudo cp -r /etc/ssl ~/container/etc
```

Then, using the `apt` tool and its `--fix-broken install` feature, the Debian keyring was installed.

With the system now operational and fully updated, our attention shifts to imple-

menting process isolation by initiating a new instance of the proc server. However, this is where significant challenges arise. While a new instance of the proc server can coexist with the currently running one, users face a hurdle in replicating a new instance identical to the original proc server. To achieve this, users must custom-build a proc server from scratch, if they need to overcome whatever constraints of the current proc server implementation. This obstacle stems from a design change by the Hurd team, whereby the proc server's initiation, originally separate, was later integrated into the init process. This decision was made due to the proc server's requirement for elevated privileges and other design considerations. Consequently, users logged into the system are unable to execute the necessary logic to initiate a new instance of the proc server, as the init process operates solely during the booting sequence. While the documentation advises users to use firm-links to connect the jail process to the host servers, this approach enlarges the issue by further reducing isolation.
I was also noticed that upon returning to the jail following a reboot, the jail lacked network access because the Packet Forwarding interface was already in use by the host. One potential explanation for this issue could be the configuration process during Debootstrap. When new servers are installed in the jail system, they are configured to operate within the current system. However, this configuration is subsequently overridden when the host system reboots, as the host boot sequence is unaware of the presence of a chroot jail.

Because of the challenges posed by process isolation limitations and the complexities involved in manually reconfiguring the servers after each system reboot, this implementation was paused. Although the research primarily concentrates on file system isolation, which the current implementation appears to achieve, the lack of substantial similarity between the current jail implementation and the container concept prompted the exploration of better alternatives.

4.3 Implementing Design 2

The second approach involves repurposing an established debugging environment to fulfill the isolation requirements anticipated by a basic container. We'll now

explore the process of setting up and evaluating a Subhurd to determine if the environment offers the required isolation to align with the research objectives.
To begin with, like a chroot environment, a Subhurd necessitates a filesystem, preferably ext2, containing all the required dependencies, including the servers. While Debootstrap could serve this purpose, the documentation suggests employing a prepared CD image. Given the challenges faced earlier with Debootstrap and the availability of essential utilities in the image file, opting for the prepared CD image for the Subhurd seems prudent.
Since the Subhurd operates within the original Hurd environment and shares the kernel, it's advisable to download the image file into the Hurd system itself rather than externally. This can be achieved with the following command.

```
sudo wget https://cdimage.debian.org/cdimage/ports/latest/\
hurd-i386/debian-hurd-20230608.img.tar.gz
```

The command above fetches the latest Hurd release, `20230608`. After downloading, the tar file must be decompressed to acquire the intended image file. The following command accomplishes this task.

```
    tar -xvzf debian-hurd-20230608.img.tar.gz
```

An error arises because of the file size limitation within Hurd, restricting files to a maximum size of 2GB (an inherent limitation in the implementation). Hence, the image file needs to be supplied externally. The downloaded and decompressed image file (subhurd1.img) can be presented as an external disk to Qemu using the following command.

```
    sudo qemu-system-x86_64 --enable-kvm -hda debian-hurd-20230608.img
        -m 4G -hdb subhurd1.img -net nic -net
        user,hostfwd=tcp:127.0.0.1:2222-:22
```

Now that the image has been linked to the system as a device, the next step is to allocate an ext2 translator to interact with the image. This can be accomplished using the following command, similar to the implementation in design 1.

```
    sudo settrans -a /mnt /hurd/ext2fs /dev/hd1s2
```

The Hurd image (subhurd1.img) is naturally divided into two partitions.

1. Partition 1 - Swap
2. Partition 2 - Image contents

Therefore when setting the translator, it is important to set it to the second partition. This is reflected in the command given above.

There are two types of translators in GNU Hurd.

- Active Translators
- Passive Translators

An active translator denotes a translator process currently running, as specified in the earlier command. Active translators can be established and removed using the settrans `-a` command, with the `-a` option indicating the intention to modify the active translator. On the other hand, a passive translator is configured and modified using the same syntax as an active translator, minus the `-a` option. All the properties discussed earlier regarding active translators also apply to passive translators. However, there is a notable difference: passive translators are not initialized yet. This approach is sensible because typically, you don't want a partition to be mounted unless you actually access files on it, or the network brought up unless there is traffic, and so on. Instead, when a passive translator is first accessed, it is automatically retrieved from the inode, and an active translator is launched on top of it using the stored command line. This mirrors the functionality of the Linux automounter but is an inherent aspect of the system, not an additional feature requiring manual setup. Therefore, setting passive translators postpones the initiation of the translator task until it's genuinely needed. Moreover, if the passive translator transitions to active but unexpectedly terminates, such as during a system reboot, it is automatically restarted the next time the inode is accessed. Another distinction is that active translators can cease to exist or become lost. Once the active translator process terminates, such as during a system reboot, it cannot be recovered. In contrast, passive translators are persistent and remain in the inode across reboots until they are modified with the `settrans` program or the inodes to which they are attached are deleted. Consequently, there is no need to maintain a configuration file with mount points. Additionally, even if a passive

translator is set, it is still possible to establish a different active translator. Only when the translator is automatically initiated because there was no active translator at the time the inode was accessed is the passive translator activated.
While passive translators provide significant advantages, we opt to establish an active translator for the `/mnt` node. This decision stems from the persistent nature of passive translators, as we aim to ensure that the translator does not remain dormant and unexpectedly activate when interaction with the Subhurd is not desired. Furthermore, setting an active translator guarantees that the currently active translator aligns with our intended choice and has not been replaced by a different active translator in the interim.
After successfully attaching the translator, the next step is to boot the Subhurd, accomplished with the following command.

```
sudo boot /dev/hd1s2
```

It is after this we run into our first issue. The boot sequence fails and the following error is displayed.

```
/hurd/mach-defpager: panic: (default pager):/hurd/mach-defpager: ../../mach-defpager/main.c:200: panic: Assertion '0' failed.
/hurd/startup: Crashing system; essential task mach-defpager died
startup: notifying random of reboot...done
startup: notifying ext2fs.static pseudo-root of reboot...done
INIT: version 3.06 booting
INIT: No inittab.d directory found
Using makefile-style concurrent boot in runlevel S.
startup: rebooting Mach (flags 0)...
Would reboot the system. Bye.
```

After extensive research, a bug was discovered to be caused by the pager of the host system. Given that GNU Hurd actively utilizes virtual memory, the kernel employs a pager, known as `mach-defpager` in GNU Hurd, for paging tasks. Typically, the kernel anticipates and operates with a single pager for these tasks. However, when booting the Subhurd, the boot program assumes it's initializing a new GNU

Hurd instance, as the image file used is identical to that of a GNU Hurd prepared CD image. Consequently, it attempts to create a new `mach-defpager` instance, resulting in a crash due to the existence of the current pager. This issue stems from a flaw in the implementation. Nonetheless, since the Subhurd isn't meant to function as an entirely distinct operating system and the kernel is expected to be shared once successfully initiated, we can bypass the creation of the pager. To modify the script associated with this operation, we need to access the content of the image file. Since we've already mounted the partition as an ext2 file system, we can navigate to it using the `cd` command. The relevant code can be found in the `/etc/hurd/runsystem.sysv` script. We can now proceed to comment out the line responsible for running the `mach-defpager` instance. The modified section of the script is provided below.

```
...
# Start the default pager. It will bail if there is already one running.
#/hurd/mach-defpager          #Commented to stop the defpager execution
...
```

After making this adjustment, the Subhurd should progress through the boot process as anticipated. However, a different error related to the file system check arises. This outcome is anticipated, and the documentation offers instructions on resolving this issue. The error occurs because the image assumes the file system is situated at `/dev/hd0s1`. Instead of utilizing the simple boot program, a new argument is supplied to enable `fastboot`, circumventing the file system check and granting access to a temporary shell. The command to execute a `fastboot` is provided below.

```
sudo boot --kernel-command-line="fastboot root=pseudo-root"
    /dev/hd1s2
```

After executing the above command, we can use the shell provided by the Subhurd to change the `fstab` file to match the expected location. This can be achieved by executing the three commands given below.

```
settrans -c /dev/pseudo-root /hurd/storeio pseudo-root
echo /dev/pseudo-root / ext2 defaults 0 1 >/etc/fstab
```

```
halt
```

Executing these commands will modify the `fstab` file situated in `/etc/fstab` and halt the Subhurd. Subsequently, the Subhurd can be restarted using the standard `boot` command, disregarding `fastboot`.

Upon booting the Subhurd, users encounter a login prompt requiring a user named "root" for authentication. It's important to note that although this user is labeled "root," its privileges are limited compared to a typical UID 0 root user. Instead, it operates as a virtual root user within the Subhurd environment.

Before proceeding to utilize the Subhurd, a virtual network interface needs to be set up. To accomplish this, we exit the Subhurd by issuing the `halt` command and return to the original host environment. From there, we utilize a translator called the `eth-multiplexer`. This network multiplexer facilitates the creation of virtual interfaces and facilitates traffic routing between virtual interfaces and an actual Ethernet interface. It relies on the Berkeley Packet Filter library (libbpf). Below are the commands required to establish a virtual network interface. It's crucial to ensure that there are no active SSH or similar connections when executing these commands, and it's advisable to run them using the root account.

The command provided below links the eth-multiplexer to the network device.

```
sudo settrans -c /dev/eth0m /hurd/eth-multiplexer
    --interface=/dev/eth0
```

Once that is done, the following three commands can be executed to create a new virtual network interface.

```
sudo ifdown /dev/eth0
sudo sed -i -e s#/dev/eth0#/dev/eth0m/0# /etc/network/interfaces
sudo ifup /dev/eth0m/0
```

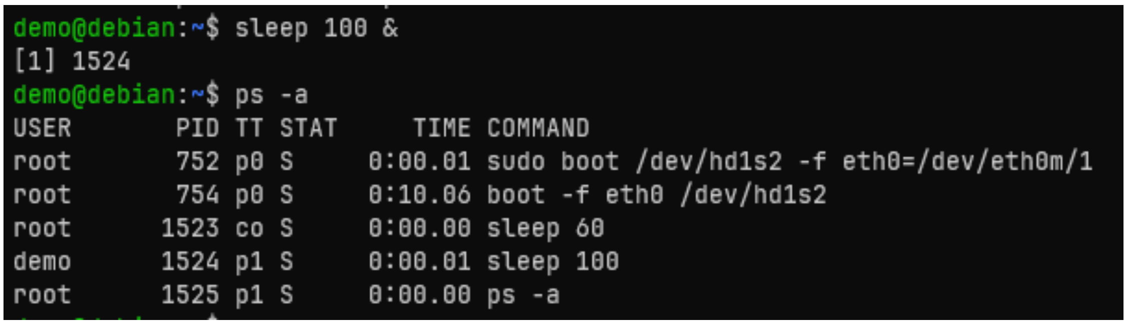
After this process is finished, the Subhurd can be booted, utilizing the created virtual network interface for its operations. This can be accomplished with the following command.

```
sudo boot /dev/hd1s2 -f eth0=/dev/eth0m/1
```

4.3.1 Isolation analysis

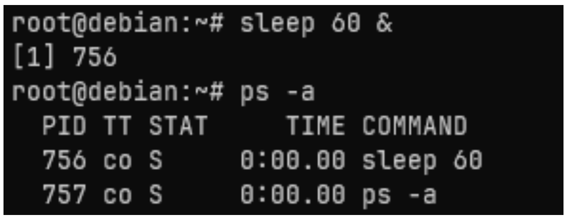
We should assess the effectiveness of the new container-like environment in isolating processes, file systems, and networks. In theory, since Subhurds initiate a completely separate cluster of core servers, these attributes should be achieved. However, it's crucial to verify if these properties are upheld in practical testing.

Process isolation can be assessed by checking whether a process running on the main host is detectable from within the Subhurd environment. This can be accomplished using the ps utility. Figure 4.1 illustrates the execution of the `ps` command in the host system, while Figure 4.2 depicts its execution within the Subhurd. As anticipated, the sleep command from the host is not observable within the Subhurd, confirming the expected process isolation. Conversely, the sleep command initiated within the Subhurd is visible in the host environment, which aligns with expectations and ensures process isolation.



```
demo@debian:~$ sleep 100 &
[1] 1524
demo@debian:~$ ps -a
USER       PID TT STAT     TIME COMMAND
root       752 p0 S     0:00.01 sudo boot /dev/hd1s2 -f eth0=/dev/eth0m/1
root       754 p0 S     0:10.06 boot -f eth0 /dev/hd1s2
root      1523 co S     0:00.00 sleep 60
demo      1524 p1 S     0:00.01 sleep 100
root      1525 p1 S     0:00.00 ps -a
```

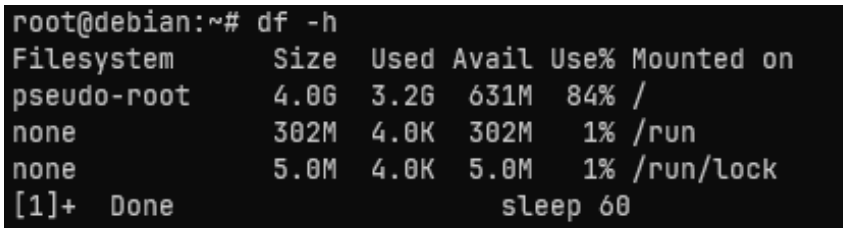
Figure 4.1: Processes listed within the host environment



```
root@debian:~# sleep 60 &
[1] 756
root@debian:~# ps -a
  PID TT STAT     TIME COMMAND
  756 co S     0:00.00 sleep 60
  757 co S     0:00.00 ps -a
```

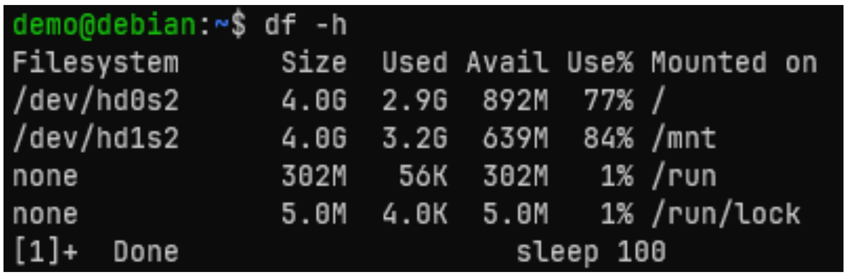
Figure 4.2: Processes listed within the Subhurd

File system isolation can be examined by using the `df` utility. Figure 4.3 illustrates the file system mounted within the Subhurd, while Figure 4.4 displays the file systems mounted within the host. By comparing the mount points and devices, it becomes evident that the file systems are isolated. Additionally, verifying this isolation can be achieved by attempting a dot-dot operation from within the Subhurd's root to access its parent directory, which fails as anticipated.



```
root@debian:~# df -h
Filesystem      Size  Used Avail Use% Mounted on
pseudo-root     4.0G  3.2G  631M  84% /
none            302M  4.0K  302M   1% /run
none            5.0M  4.0K  5.0M   1% /run/lock
[1]+  Done                    sleep 60
```

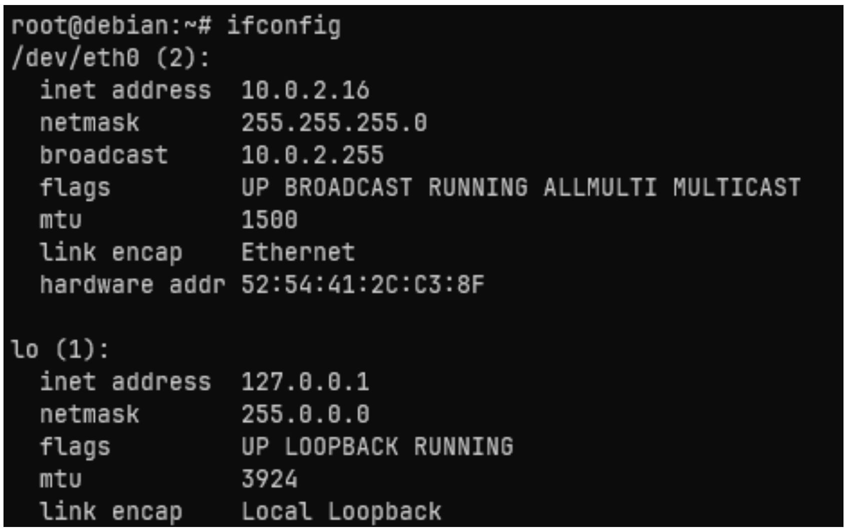
Figure 4.3: File Systems within the Subhurd



```
demo@debian:~$ df -h
Filesystem      Size  Used Avail Use% Mounted on
/dev/hd0s2      4.0G  2.9G  892M  77% /
/dev/hd1s2      4.0G  3.2G  639M  84% /mnt
none            302M   56K  302M   1% /run
none            5.0M  4.0K  5.0M   1% /run/lock
[1]+  Done                    sleep 100
```

Figure 4.4: File Systems within the host environment

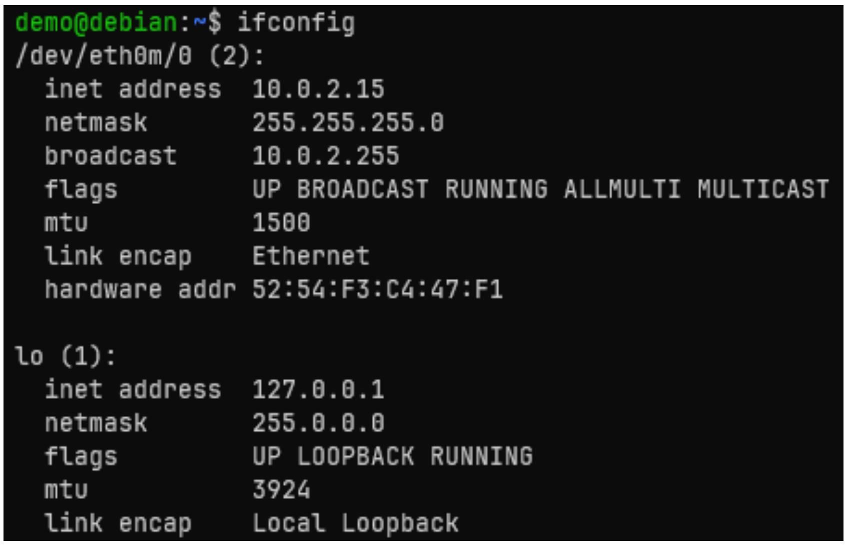
Network isolation can be assessed by employing the `ifconfig` tool and contrasting the interfaces. Figure 4.5 illustrates the network interfaces utilized by the Subhurd, while Figure 4.6 displays the network interfaces employed by the host. It is evident that the Subhurd solely possesses access to a virtual network interface, whereas the host employs the original interface. This confirms network isolation.



```
root@debian:~# ifconfig
/dev/eth0 (2):
  inet address  10.0.2.16
  netmask       255.255.255.0
  broadcast     10.0.2.255
  flags         UP BROADCAST RUNNING ALLMULTI MULTICAST
  mtu           1500
  link encap    Ethernet
  hardware addr 52:54:41:2C:C3:8F

lo (1):
  inet address  127.0.0.1
  netmask       255.0.0.0
  flags         UP LOOPBACK RUNNING
  mtu           3924
  link encap    Local Loopback
```

Figure 4.5: Network interfaces within the Subhurd



```
demo@debian:~$ ifconfig
/dev/eth0m/0 (2):
  inet address  10.0.2.15
  netmask       255.255.255.0
  broadcast     10.0.2.255
  flags         UP BROADCAST RUNNING ALLMULTI MULTICAST
  mtu           1500
  link encap    Ethernet
  hardware addr 52:54:F3:C4:47:F1

lo (1):
  inet address  127.0.0.1
  netmask       255.0.0.0
  flags         UP LOOPBACK RUNNING
  mtu           3924
  link encap    Local Loopback
```

Figure 4.6: Network interfaces within the host environment

In the broader context, this implemented environment closely resembles a container.

Chapter 5 - Results and Evaluation

The results section addresses two main components. Initially, it explores the evaluation of file system isolation within a container solution based on a microkernel architecture. Subsequently, it presents a performance comparison between the two container implementations across CPU, memory, and I/O metrics.

5.1 Analysis on file system isolation

The analysis of file system isolation within subhurds constitutes a critical aspect of understanding the security and performance implications of microkernel-based operating systems like GNU Hurd. Inherent to the design philosophy of microkernels is the imperative to minimize the Trusted Computing Base (TCB), a principle that directly correlates with the level of isolation achievable within the system. Within this paradigm, meticulous attention to the design and implementation of file system interfaces plays a pivotal role in ensuring robust isolation between user-level processes and kernel services. This section delves into an in-depth analysis of various aspects of file system isolation within subhurds, shedding light on both theoretical underpinnings and practical implications for security, performance, and system resilience. Given below is an overview of the topics cover throughout the analysis

1. Separating of the file system server from the kernel.
2. Minimizing of shared resources
3. Extensibility of the virtualized environment
4. Tighter interfaces with strict access control

5.1.1 Separating of the file system server from the kernel

Given that microkernels communicate through Inter-Process Communication (IPC), this ensures that the server maintains a well-defined interface and a separation of concerns [49]. The functionality of the file system server remains contained within itself, with only the necessary functionality present. This significant reduction in the Trusted Computing Base (TCB) of the file system enhances isolation by minimizing shared regions. Additionally, it diminishes dependencies on other components; errors occurring in different servers are less likely to affect the functionality of the file system server, thus enhancing isolation further.

In contrast, in a monolithic implementation, the file system logic is interspersed among other components, which can lead to kernel misuse of the file system, causing unforeseen issues. This expanded TCB and loosely defined interface contribute substantially to decreased isolation. This disparity becomes more pronounced when comparing the nature of the address spaces in both architectures, which greatly impacts error propagation and fault tolerance of the file system and related components. The monolithic architecture employs a unified address space, wherein the kernel resides within the processes' address space upon execution. This design choice is primarily driven by the necessity to execute kernel code during system calls without necessitating a context switch, thereby enhancing performance. However, this approach, while advantageous for performance, presents significant drawbacks in terms of isolation. Integrating the kernel within the same address space as potentially malicious processes is suboptimal, particularly concerning the isolation of critical operating system components such as the file system [49].

A vulnerability in either a kernel component or an executing process could lead to an attacker gaining access to the entire kernel, thereby facilitating an attack. Consequently, this leaves the file system susceptible to exploitation. In contrast, within a microkernel architecture, the kernel's address space is distinct and can only be accessed through Inter-Process Communication (IPC). This segregation significantly mitigates the risk of kernel attacks, consequently reducing the likelihood of misusing highly privileged kernel components to compromise the file system[49].

Hence, the separation in address spaces yields two key benefits: first, it prevents a bug within the file system from impacting other components, and second, it shields the file system from bugs in disparate components, including the kernel. This separation enhances isolation by reducing shared regions and dependencies.

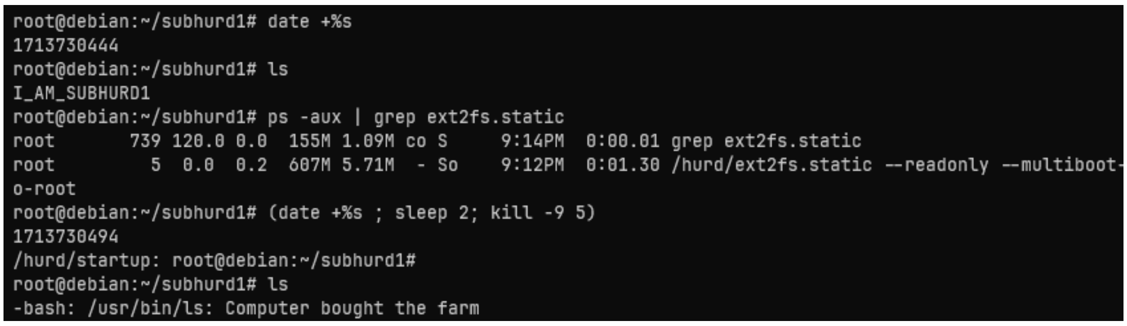
Furthermore, the separation of the file system from the kernel yields additional advantages. By relocating the file system logic from the kernel to a smaller component with a well-defined interface and a secure protocol, the logic within the server becomes amenable to formal verification. This rigorous verification process serves to minimize system bugs and ensures the expected functionality of the file system. These characteristics and isolation benefits are evident within the kernel, but how do they translate to the realm of containerization? As previously discussed, the file system holds significant importance within a microkernel architecture. In the case of Hurd, it serves as the primary authority responsible for global namespace management, including pathname resolution, and functions as a data store. Given the criticality of its role, ensuring the reliability and security of this particular server is paramount.

Containers, or similar environments such as a Subhurd, executed on the host operating system, facilitate the execution of programs of diverse natures. Consequently, meticulous exposure of the file system through an extremely stringent interface employing secure protocols becomes crucial in enhancing the isolation of the server, thus ensuring the security of the container that relies on its functionality.

This principle also extends to error propagation and fault tolerance. Since each container possesses its own file system isolated within its designated address space, a vulnerability occurring within the file system of a single container does not propagate to other containers. Similarly, a fault occurring in a separate component within a container has minimal impact on the container's ability to utilize the file system server. This capability to contain faults and enhance fault tolerance can be readily demonstrated through the use of two Subhurds, as follows.

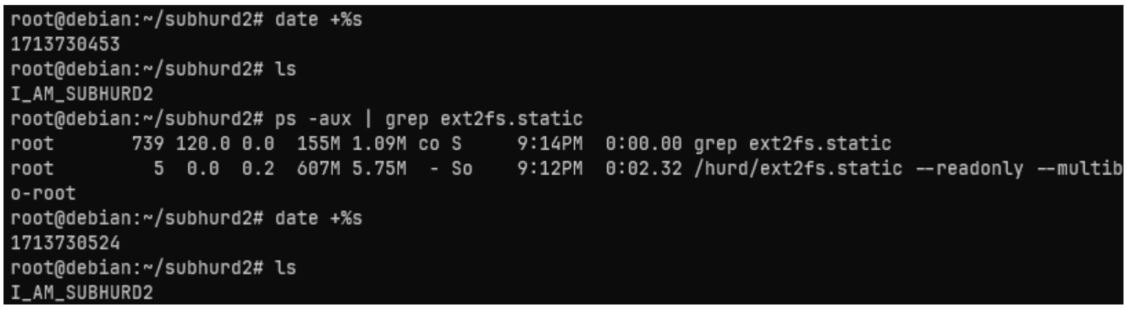
The experiment aims to terminate the file system server of one Subhurd and observe whether the file system server of the other Subhurd remains operational. In Figure 5.1, the diagrams demonstrates the process involving the identification of the PID related to the file system server of Subhurd 1 and terminating it. Timestamps are provided to demonstrate that the termination occurred prior to executing the ls command on Subhurd 2. Figure 5.2 illustrates that Subhurd 2 remains unaffected by the crash of Subhurd 1's file system server. Finally, Figure 5.3 displays the outcome with only one file system server remaining (that of Subhurd 2).

Note: The ext2fs.static file system types are associated with Subhurds, while the host employs an ext2 file system type. Therefore, the grep tool was instructed to search for the text "ext2fs.static". Additionally, it's important to mention that the error message "Computer bought the farm," typically indicative of a translator



```
root@debian:~/subhurd1# date +%s
1713730444
root@debian:~/subhurd1# ls
I_AM_SUBHURD1
root@debian:~/subhurd1# ps -aux | grep ext2fs.static
root       739 120.0 0.0  155M 1.09M co S     9:14PM  0:00.01 grep ext2fs.static
root         5  0.0  0.2  607M 5.71M  - So    9:12PM  0:01.30 /hurd/ext2fs.static --readonly --multiboot-
o-root
root@debian:~/subhurd1# (date +%s ; sleep 2; kill -9 5)
1713730494
/hurd/startup: root@debian:~/subhurd1#
root@debian:~/subhurd1# ls
-bash: /usr/bin/ls: Computer bought the farm
```

Figure 5.1: Killing the file system server of Subhurd 1



```
root@debian:~/subhurd2# date +%s
1713730453
root@debian:~/subhurd2# ls
I_AM_SUBHURD2
root@debian:~/subhurd2# ps -aux | grep ext2fs.static
root       739 120.0 0.0  155M 1.09M co S     9:14PM  0:00.00 grep ext2fs.static
root         5  0.0  0.2  607M 5.75M  - So    9:12PM  0:02.32 /hurd/ext2fs.static --readonly --multib
o-root
root@debian:~/subhurd2# date +%s
1713730524
root@debian:~/subhurd2# ls
I_AM_SUBHURD2
```

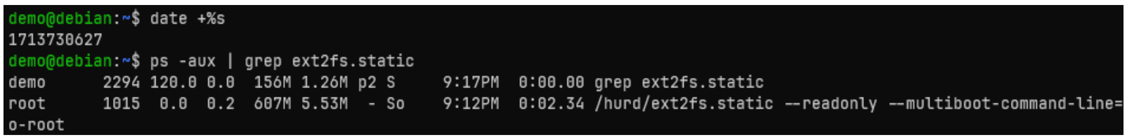
Figure 5.2: Observing the functionality of the file system server of Subhurd 2

crash, is equivalent to an “Oops” message [56].

In a monolithic system, a scenario similar to the experiment described above, where a glitch in the file system code leads to a crash, would result in the entire kernel collapsing, consequently affecting the entire cluster of containers [49].

5.1.2 Minimizing of shared resources

Microkernels inherently place their file system-related logic in user space as a process that can be initiated and duplicated as required. This was exemplified several times in the implementation section. This capacity to encapsulate and isolate task-related logic while scaling horizontally offers certain advantages. For instance, non-shareable resources within the file system are dedicated to specific instances (e.g., an array of UID stored within the file system server accessed with a mutex lock). Such resources often pose issues when multiple processes attempt to utilize the same non-shareable resources simultaneously, as access to these resources is blocked. However, when the logic can be horizontally scaled, similar to how file system servers in Hurd can be initiated as needed, there is less contention and



```
demo@debian:~$ date +%s
1713730627
demo@debian:~$ ps -aux | grep ext2fs.static
demo      2294 120.0 0.0  156M 1.26M p2 S     9:17PM  0:00.00 grep ext2fs.static
root      1015  0.0  0.2  607M 5.53M  - So    9:12PM  0:02.34 /hurd/ext2fs.static --readonly --multiboot-command-line=
o-root
```

Figure 5.3: List of operating file systems seen by the host

congestion to access these types of resources, which can enhance performance and reduce the impact on a process using the file system from another process. This effect is particularly pronounced in the context of containers. As observed during the implementation of the Subhurd, each container (or Subhurd) receives a dedicated file system server to handle tasks specific to it. This implies that the utilization frequency of the file system server of one container will not affect the performance of another. This reduction in interdependence between file system servers and containers can be viewed as an enhancement in isolation compared to the monolithic counterpart, where the file system logic is shared among all containers. This concept is illustrated in Figure 5.4.

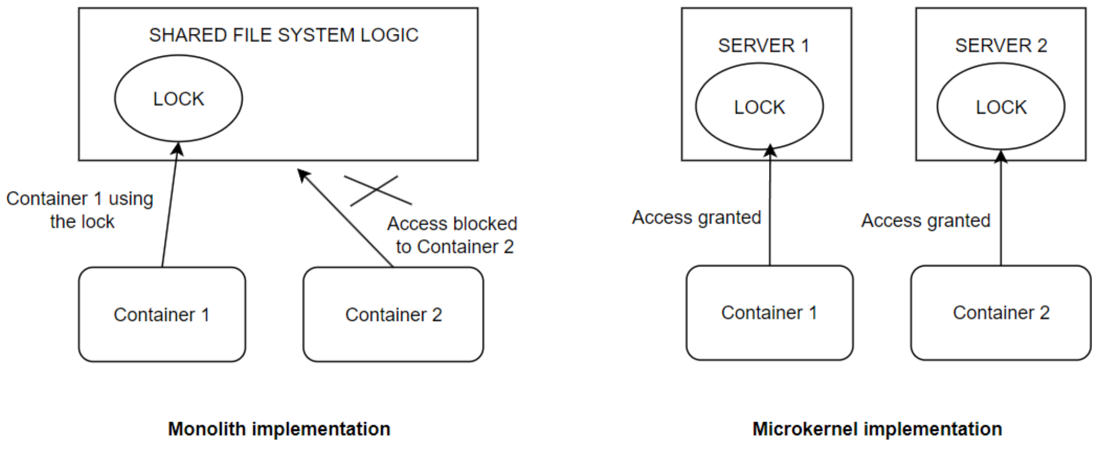


Figure 5.4: Minimization of non-sharable resources in a microkernel architecture

Additionally, increasing the pool of available resources while decreasing the number of processes requesting them should significantly decrease the likelihood of encountering deadlocks, thus providing better isolation against deadlocks.

While the isolation provided could enhance performance, reducing the overall sharing of resources could also enhance security. When the same resource is shared among multiple processes, these resources become vulnerable to side-channel attacks. These attacks could potentially extract information about recently accessed files, access rights, and other metadata stored in the file system logic related to a particular user. Numerous incidents involving cryptographic algorithms and cache systems have underscored the risks associated with such attacks. For instance, a scenario where a container could extract information about another container in the system using a side-channel attack could have devastating consequences[23][22][25].

5.1.3 Extensibility of the containerized system

While extensibility may not initially appear closely tied to isolation, its relevance becomes evident upon closer examination. Decreasing reliance on the host system represents an enhancement of isolation. Essentially, this implies that when a virtualized system becomes good at fulfilling requirements while reducing dependency on the specific implementation details of the host system, its isolation has been improved. The objective of any virtualization platform and technique is to provide an experience similar to that of a fully dedicated system. As these two systems converge in terms of functionalities, the mutual restrictions and dependencies diminish, thereby enhancing independence alongside isolation [9]. Consequently, even if changes occur in the file system implementation, a container remains insulated from these changes as it wasn't reliant on them to begin with.
One such functionality demonstrating this principle is the ability to connect a file system of any type to a container and utilize it. Subhurds facilitates this capability, as utilizing a translator is not a privileged action. Users retain the freedom to opt for a completely different file system than the one provided by the underlying kernel, perhaps selecting one that better suits their tasks or preferences. Furthermore, this feature enables the testing of custom file systems from within a container, thereby fostering experimentation and innovation. The monolithic counterpart has recently recognized the value of separation and isolation, endeavoring to incorporate userspace file systems like `FUSE` and introduce concepts such as Docker Volumes to enhance file system capabilities [61] [62]. However, these implementations still necessitate privileged access or a system capability akin to `CAP_SYS_ADMIN` to function effectively. Granting such formidable power to an unknown user solely for the purpose of mounting a new file system presents a notable

security risk. Rootless containers have been proposed as a solution, which involves adding the user `docker` to the `sudo` group. While this mitigates the issue to some extent, it essentially involves introducing a potentially exploitable user into a highly privileged group. One of the primary reasons for this issue lies in the centralized file system logic, tasked with managing various file system types. Allowing users to integrate any file system introduces the potential for complications within the file system logic, potentially surpassing its capabilities. As previously discussed, this places the entire kernel at risk, consequently jeopardizing the integrity of the entire cluster of containers. However, microkernels do not encounter this issue. If a container misuses this functionality and inadvertently causes a problem or bug, only that specific container will fail, leaving the rest of the containers unaffected. This principle extends to drivers associated with file systems and related hardware as well. In monolithic architectures, drivers are integrated into the kernel, making it challenging to select a set of drivers optimized for a particular file system implementation. Instead, reliance is placed on the kernel developer's choice, which may not necessarily align with individual requirements. This dependency of containers on their kernel is absent in the microkernel architecture. Since drivers exist in userspace as regular processes, users have the flexibility to initiate new drivers and utilize them as needed, thereby alleviating reliance on kernel-specific choices.

5.1.4 Tighter interfaces with strict access control

Microkernels implement servers as processes with interfaces, which is partly necessitated by the communication mode being Remote Procedure Call (RPC). However, this setup provides an advantage when considering process isolation. In the case of Hurd, there is stringent control over who and what level of access is granted on its underlying storage, accomplished through Identity-Based Access Control (IBAC). Hurd servers manage access to objects based on the identity of the subject. While the policy is similar to Unix, the mechanism differs significantly. In Hurd, identities are treated as first-class objects, meaning a single process may possess multiple identities or none at all, managed by the auth server. The auth server also supports programs in the implementation of IBAC by furnishing an authentication mechanism that enables programs to securely expose identities to others in a verifiable manner [52].

As identities are first-class objects, a process can hold access to any number of User IDs (UIDs) and Group IDs (GIDs), or none at all. Furthermore, because they are merely objects denoted by capabilities, a process can relinquish its authority to an identity by dropping the capability referencing it, a process known as discretionary authority reduction. This technique enables applications to operate with reduced authority, thereby mitigating the potential damage caused by bugs or attackers. Applications requiring access to a fixed number of resources at startup, and subsequently not needing the authority granted by an identity, can leverage this approach. For example, a network server needing to bind to a TCP port below 1024, but not requiring superuser authority otherwise, can operate without any UIDs or GIDs after binding to the port.

This practice of tightening interfaces with stringent access control measures, which diminishes shared interfaces and interactions, thereby enhancing isolation, can also be implemented in file system-related servers. For instance, a document viewer, after opening a user-specified file, could nullify the identity object to mitigate the impact of a malicious macro, effectively eliminating a potential vulnerability isolationg the file system against such attacks.

5.2 Performance comparison

Performance comparisons were conducted between a single Subhurd environment and a Docker Lubuntu image (running on WSL). Since many popular tools are designed for the Linux kernel, smaller open-source benchmarks compatible with Hurd's glibc interface were utilized. It's important to highlight that GNU Hurd operates on QEMU with KVM, and the performance metrics may vary when running on a bare-metal Hurd. Each test was executed five times (CPU benchmark three times) at various instances, and the average values were considered for analysis.

5.2.1 CPU

Figure 5.5 illustrates the duration taken by each environment to execute a benchmark tasked with calculating the first 5,761,455 primes [63]. As anticipated, Subhurd exhibits suboptimal performance, attributed to its constrained optimization within the kernel and the overhead of inter-process communication (IPC).

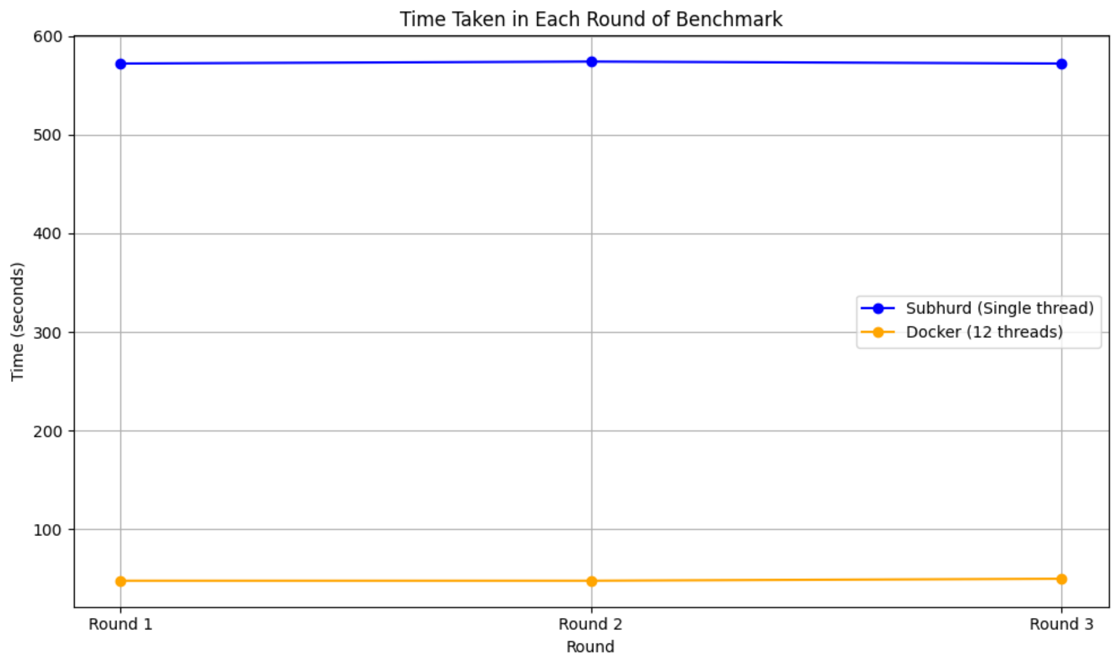


Figure 5.5: Time taken to calculate 5,761,455 primes

5.2.2 File System

The benchmark utilized in the assessment aimed to generate new files with names of 40 bytes in length and content of 10,240 bytes. A total of 5,000 such files were generated to evaluate the performance and effectiveness of the file system logic and interfaces [64].

Figure 5.6 depicts the comparison of files created per second. It is evident from the data that the Subhurd performs comparably well with the Docker implementation in this specific metric.

Figure 5.7 illustrates the comparison of application overhead, which denotes the duration the application remained idle while other components were occupied with their respective tasks. As anticipated, the Subhurd exhibits poor performance

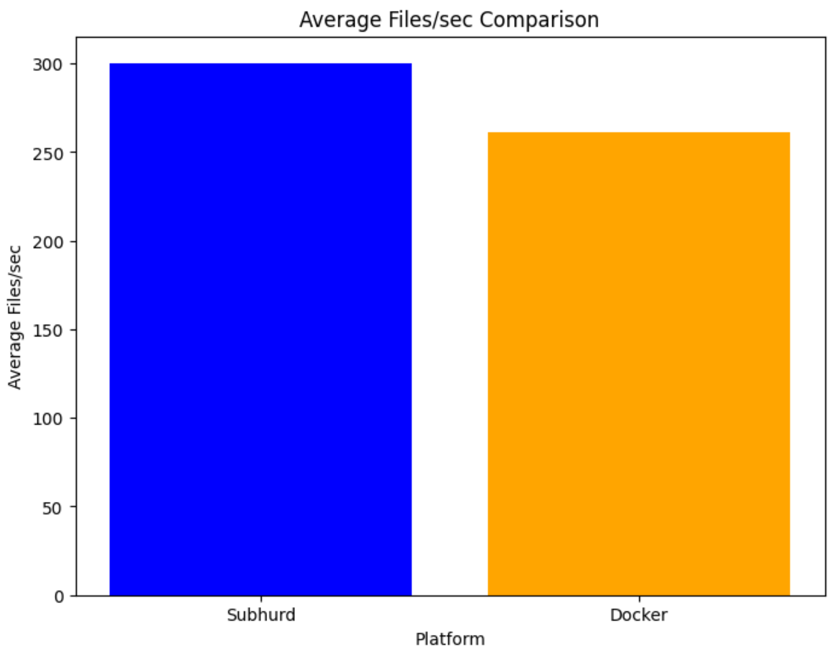


Figure 5.6: File created per second

owing to the utilization of message passing as the communication mechanism.

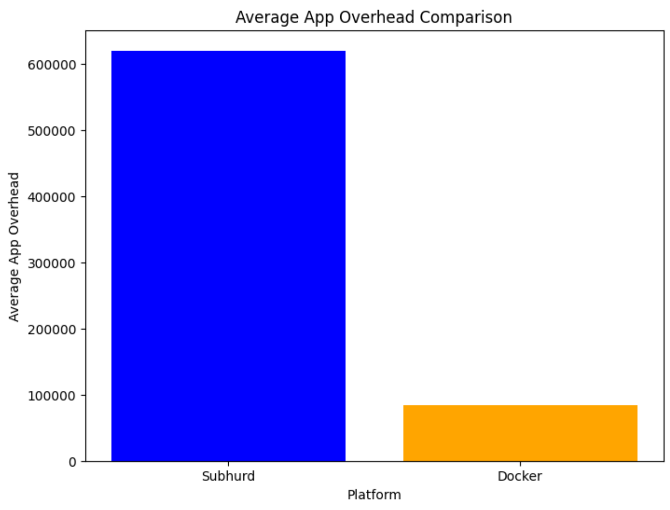


Figure 5.7: Application overhead

In Figure 5.8, the latency in executing each system call during the file creation process is depicted. Once more, the Subhurd demonstrates a consistent underperformance, attributable to its reliance on the IPC mechanism.

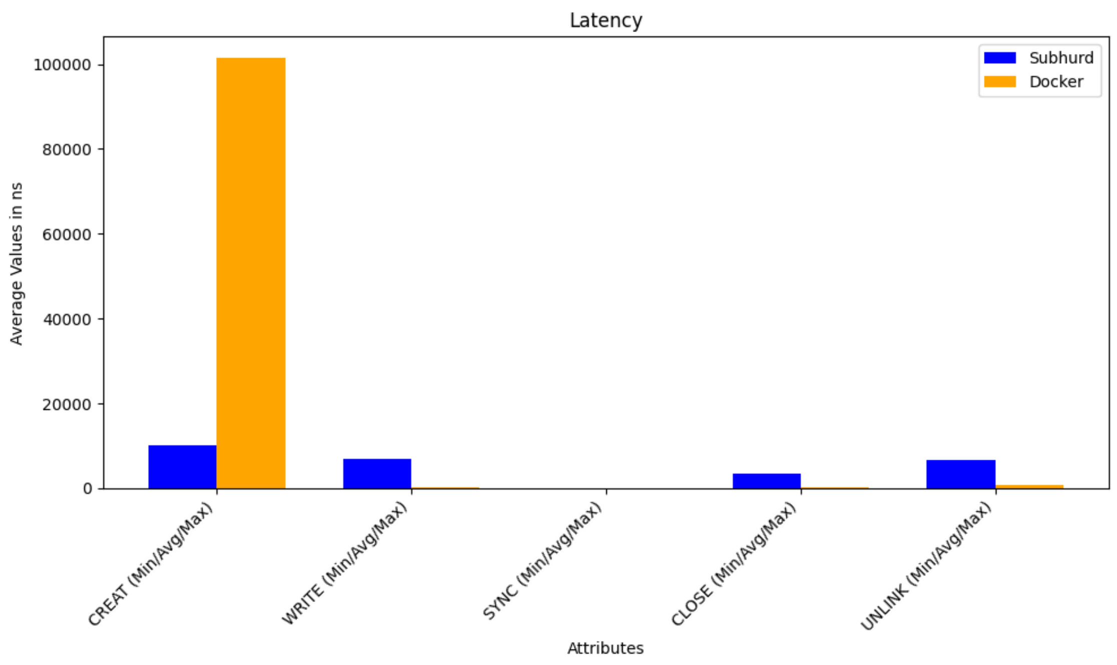


Figure 5.8: System call latency

5.2.3 Memory

The benchmark conducts a sequence of memory allocations and records the transfer speeds. Additionally, it reads blocks of varying sizes from memory and measures the latency in receiving these expected blocks. Figure 5.9 illustrates the transfer speeds for different memory allocation and deallocation tasks, revealing Docker container's superior performance over the Subhurd [65].

Figure 5.10 depicts the reading of blocks from memory, with the Subhurd displaying the anticipated poor performance.

In general, the Subhurd exhibits poor performance across various metrics, attributable to its reliance on Inter-Process Communication, limited optimizations due to inadequate development contributions, and the additional overhead introduced by Qemu emulation.

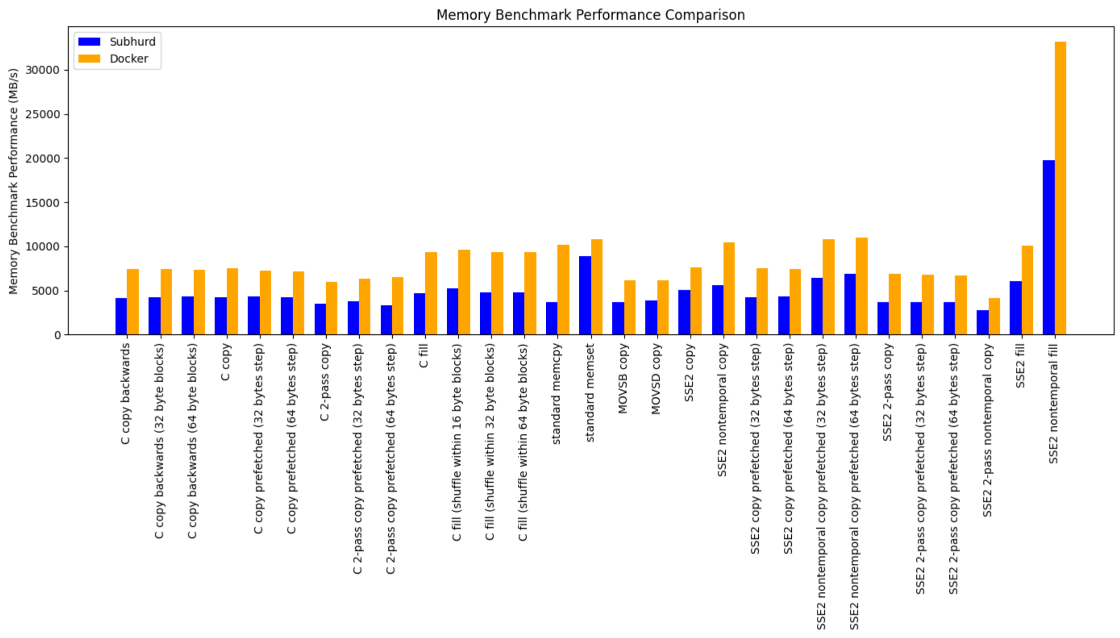


Figure 5.9: Memory transfer speeds

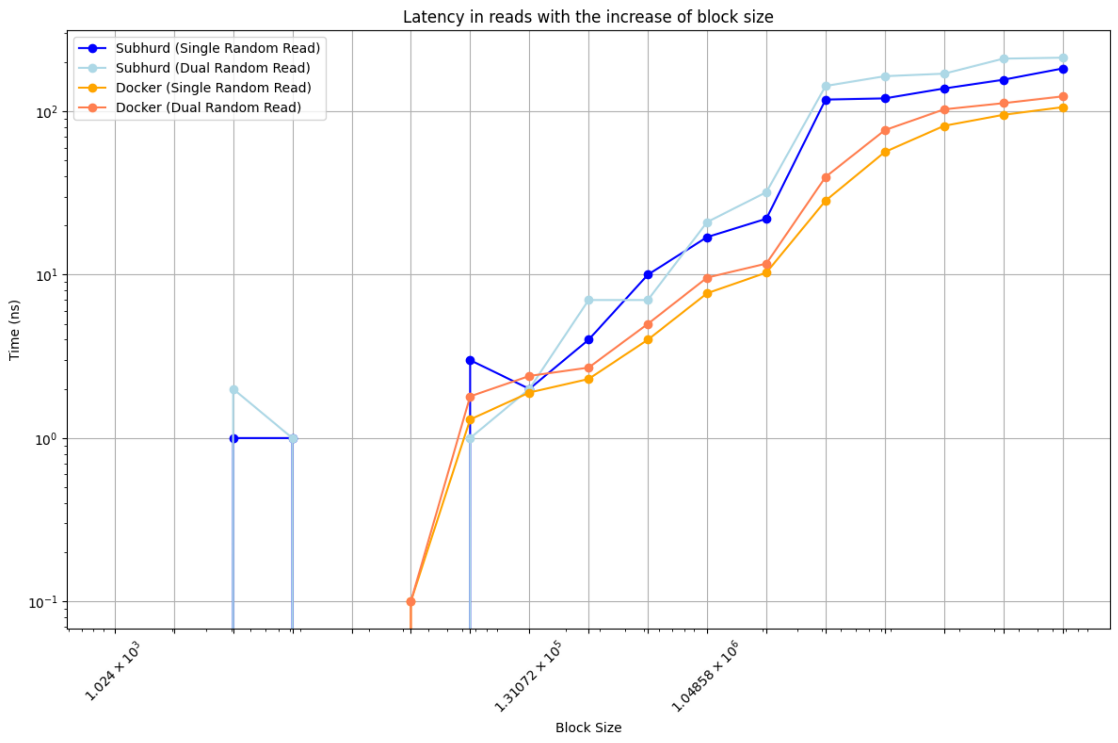


Figure 5.10: Reading speed

Chapter 6 - Conclusions

The research has effectively achieved its objectives by investigating and establishing a correlation between microkernels and containers. Additionally, it successfully implemented a rudimentary version of containers utilizing the functionalities offered by the microkernel-based operating system (GNU Hurd). Furthermore, the study delved into and evaluated the capabilities provided by microkernels to enhance file system isolation. Finally, a performance comparison was conducted between the two environments to address the research questions.
The findings indicate that:

- The decoupled architecture of microkernels can indeed be leveraged to enhance the low-level isolation of containers.
- However, the implementation of containers in a microkernel-based operating system exhibits performance limitations.

Overall, this research lays a strong foundation for further exploration into the discussed topics.

6.1 Limitations

- The research primarily concentrated on file system isolation, offering only a broader perspective on additional isolation mechanisms
- The research revolves around implementing GNU Hurd rather than presenting a generalized theoretical model.
- The isolation analysis does not delve deeply into drivers and other firmware-related aspects that could impact file system isolation.
- The research does not extensively explore the highly technical aspects of GNU Hurd.

6.2 Future Work

- Delving into other aspects of a microkernel-based operating system, such as networking and security, and investigating how the inherent features of a microkernel can enhance those aspects when running a container implementation.
- Identifying potential enhancements for GNU Hurd to address some of the challenges encountered during the implementation of a container.

Bibliography

[1] Fernando, F. Freitag, and Navarro, "A summary of virtualization techniques," 2012.

[2] K. Khajehei, "Role of virtualization in cloud computing," *International Journal of Advance Research in Computer Science and Management Studies*, vol. 2, no. 4, p. 0, 2014.

[3] R. K. Barik, R. K. Lenka, K. R. Rao, and D. Ghose, "Performance analysis of virtual machines and containers in cloud computing," in *2016 International Conference on Computing, Communication and Automation (ICCCA)*, 2016, pp. 1204–1210. DOI: `10.1109/CCAA.2016.7813925`.

[4] V. G. da Silva, M. Kirikova, and G. Alksnis, "Containers for virtualization: An overview," *Applied Computer Systems*, vol. 23, no. 1, pp. 21–27, 2018. DOI: `doi:10.2478/acss-2018-0003`. [Online]. Available: `https://doi.org/10.2478/acss-2018-0003`.

[5] C. G. Kominos, N. Seyvet, and K. Vandikas, "Bare-metal, virtual machines and containers in openstack," in *2017 20th Conference on Innovations in Clouds, Internet and Networks (ICIN)*, 2017, pp. 36–43. DOI: `10.1109/ICIN.2017.7899247`.

[6] J. Gu, X. Wu, W. Li, *et al.*, "Harmonizing performance and isolation in microkernels with efficient intra-kernel isolation and communication," in *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, 2020, pp. 401–417.

[7] A. R. Hevner, "A three cycle view of design science research," *Scandinavian journal of information systems*, vol. 19, no. 2, p. 4, 2007.

[8] S. Crosby and D. Brown, "The virtualization reality: Are hypervisors the new foundation for system software?" *Queue*, vol. 4, no. 10, pp. 34–41, Dec. 2006, ISSN: 1542-7730. DOI: `10.1145/1189276.1189289`. [Online]. Available: `https://doi.org/10.1145/1189276.1189289`.

[9] S. Sharma and Y. Park, "Virtualization: A review and future directions executive overview," *American Journal of Information Technology*, vol. 1, pp. 1–37, May 2011.

[10] VMware, Inc., *Understanding full virtualization, paravirtualization, and hardware assist*, VM Technical Resources, Retrieved Apr. 7, 2024 from `https://www.vmware.com/techpapers/2007/understanding-full-virtualization-paravirtualizat-1008.html`, Nov. 2007.

[11] R. Uhlig, G. Neiger, D. Rodgers, *et al.*, "Intel virtualization technology," *Computer*, vol. 38, no. 5, pp. 48–56, 2005. DOI: `10.1109/MC.2005.163`.

[12] A. Bhardwaj and C. R. Krishna, "Virtualization in cloud computing: Moving from hypervisor to containerization—a survey," *Arabian Journal for Science and Engineering*, vol. 46, no. 9, pp. 8585–8601, 2021.

[13] S. Tamane, "A review on virtualization: A cloud technology," *International Journal on Recent and Innovation Trends in Computing and Communication*, vol. 3, no. 7, pp. 4582–4585, 2015.

[14] A. Bhardwaj and C. R. Krishna, “Virtualization in cloud computing: Moving from hypervisor to containerization—a survey,” *Arabian Journal for Science and Engineering*, vol. 46, no. 9, pp. 8585–8601, 2021.

[15] IBM, *What is a virtual machine (vm)?* [Online]. Available: `https://www.ibm.com/topics/virtual-machines`.

[16] R. Chopra, “A review paper on virtualization,” *International Journal of Innovative Research in Computer Science & Technology*, vol. 10, no. 2, pp. 131–135, 2022.

[17] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. De Rose, “Performance evaluation of container-based virtualization for high performance computing environments,” in *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, IEEE, 2013, pp. 233–240.

[18] P. Sharma, L. Chaufournier, P. Shenoy, and Y. C. Tay, “Containers and virtual machines at scale: A comparative study,” in *Proceedings of the 17th International Middleware Conference*, ser. Middleware ’16, Trento, Italy: Association for Computing Machinery, 2016, ISBN: 9781450343008. DOI: `10.1145/2988336.2988337`. [Online]. Available: `https://doi.org/10.1145/2988336.2988337`.

[19] M. Compastié, R. Badonnel, O. Festor, and R. He, “From virtualization security issues to cloud protection opportunities: An in-depth analysis of system virtualization models,” *Computers & Security*, vol. 97, p. 101 905, 2020.

[20] A. Hakamian and A. Rahmani, “Evaluation of isolation in virtual machine environments encounter in effective attacks against memory,” *Security and Communication Networks*, vol. 8, Oct. 2015. DOI: `10.1002/sec.1374`.

[21] G. J. Popek and R. P. Goldberg, “Formal requirements for virtualizable third generation architectures,” *Commun. ACM*, vol. 17, no. 7, pp. 412–421, Jul. 1974, ISSN: 0001-0782. DOI: `10.1145/361011.361073`. [Online]. Available: `https://doi.org/10.1145/361011.361073`.

[22] C. Maurice, C. Neumann, O. Heen, and A. Francillon, “Confidentiality issues on a gpu in a virtualized environment,” in *Financial Cryptography and Data Security*, Springer, 2014, pp. 119–135.

[23] D. J. Bernstein, “Cache-timing attacks on aes,” 2005.

[24] F. Serna, *The info leak era on software exploitation*, Retrieved Apr. 7, 2024 from `https://media.blackhat.com/bh-us-12/Briefings/Serna/BH_US_12_Serna_Leak_Era_Slides.pdf`, 2012.

[25] M. Talbi, *Attacking a co-hosted vm: A hacker, a hammer and two memory modules*, Retrieved Apr. 7, 2024 from `https://www.stormshield.com/news/attacking-co-hosted-vm-hacker-hammer-two-memory-modules/`, 2017.

[26] M. Pearce, S. Zeadally, and R. Hunt, “Virtualization: Issues, security threats, and solutions,” *ACM Computing Surveys (CSUR)*, vol. 45, no. 2, pp. 1–39, 2013.

[27] X. Diao, M. Pietrykowski, F. Huang, C. Mutha, and C. Smidts, "An ontology-based fault generation and fault propagation analysis approach for safety-critical computer systems at the design stage," *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, vol. 36, e1, 2022. DOI: `10.1017/S0890060421000342`.

[28] S. Kundu, S. Chowdhury, S. Saha, A. Karmakar, D. Mukhopadhyay, and I. Verbauwhede, "Carry your fault: A fault propagation attack on side-channel protected lwe-based kem," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2024, pp. 844–869, Mar. 2024. DOI: `10.46586/tches.v2024.i2.844-869`.

[29] J. Sun, S. Li, J. Xu, and J. Huang, "The security war in file systems: An empirical study from a vulnerability-centric perspective," *ACM Trans. Storage*, vol. 19, no. 4, Oct. 2023, ISSN: 1553-3077. DOI: `10.1145/3606020`. [Online]. Available: `https://doi.org/10.1145/3606020`.

[30] Z. Huang, "A comparative study on the performance isolation of virtualization technologies," Arizona State University, Tech. Rep., 2019.

[31] *How to create a Linux container without a VM (LX Branded Zones) - SmartOS Documentation — wiki.smartos.org*, `https://wiki.smartos.org/lx-branded-zones/`, [Accessed 08-04-2024].

[32] craigloewen-msft, *What is Windows Subsystem for Linux — learn.microsoft.com*, `https://learn.microsoft.com/en-us/windows/wsl/about`, [Accessed 08-04-2024].

[33] M. D. M. B. W. Kernighan, *UNIX Time-Sharing System: UNIX Programmer's Manual*, 7th ed. Bell Telephone Laboratories Incorporated, Murray Hill, NJ., 1979, vol. 2.

[34] P.-H. Kamp and R. N. Watson, "Jails: Confining the omnipotent root," in *Proceedings of the 2nd International SANE Conference*, vol. 43, 2000, p. 116.

[35] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors," in *Proceedings of the 2Nd ACM SIGOPS/EuroSys european conference on computer systems 2007*, 2007, pp. 275–287.

[36] Y. Huang, A. Stavrou, A. K. Ghosh, and S. Jajodia, "Efficiently tracking application interactions using lightweight virtualization," in *Proceedings of the 1st ACM workshop on Virtual machine security*, 2008, pp. 19–28.

[37] D. Price and A. Tucker, "Solaris zones: Operating system support for consolidating commercial workloads.," in *LISA*, vol. 4, 2004, pp. 241–254.

[38] C. Cowan, S. Beattie, G. Kroah-Hartman, C. Pu, P. Wagle, and V. D. Gligor, "Subdomain: Parsimonious server security.," in *LISA*, 2000, pp. 355–368.

[39] P. A. Loscocco and S. Smalley, "Integrating flexible support for security policies into the linux operating system.," in *USENIX Annual Technical Conference, FREENIX Track*, 2001, pp. 29–42.

[40] C. Wright, C. Cowan, J. Morris, S. Smalley, and G. Kroah-Hartman, "Linux security module framework," in *Ottawa Linux Symposium*, vol. 8032, 2002, pp. 6–16.

[41] J. Krude and U. Meyer, “A versatile code execution isolation framework with security first,” in *Proceedings of the 2013 ACM workshop on Cloud computing security workshop*, 2013, pp. 1–10.

[42] A. Randal, “The ideal versus the real: Revisiting the history of virtual machines and containers,” *ACM Computing Surveys (CSUR)*, vol. 53, no. 1, pp. 1–31, 2020.

[43] X. Gao, Z. Gu, Z. Li, H. Jamjoom, and C. Wang, “Houdini’s escape: Breaking the resource rein of linux control groups,” in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1073–1086.

[44] *Finding Out Which Linux Capabilities a Process Needs to Work — Baeldung on Linux — baeldung.com*, `https://www.baeldung.com/linux/process-needed-capabilities`, [Accessed 11-04-2024].

[45] *Exploring container security: An overview — Google Cloud Blog — cloud.google.com*, `https://cloud.google.com/blog/products/gcp/exploring-container-security-an-overview`, [Accessed 11-04-2024].

[46] D. J. Walsh, *Are Docker containers really secure? — opensource.com*, `https://opensource.com/business/14/7/docker-security-selinux`, [Accessed 11-04-2024].

[47] A. Tanenbaum, *Modern Operating Systems*. Pearson Prentice Hall, 2009, ISBN: 9780138134594. [Online]. Available: `https://books.google.lk/books?id=3PM5ngEACAAJ`.

[48] P. B. Hansen, “Rc 4000 software: Multiprogramming system,” in *Classic Operating Systems: From Batch Processing To Distributed Systems*, Springer, 1969, pp. 237–281.

[49] P. Bitterling, *Operating system kernels*, 2010.

[50] S. Biggs, D. Lee, and G. Heiser, “The jury is in: Monolithic os design is flawed,” in *Asia-Pacific Workshop on Systems (APSys). Korea: ACM SIGOPS*, 2018.

[51] M. Rana and S. Baul, “A survey on microkernel based operating systems and their essential key components,” *SSRN Electronic Journal*, Jun. 2023. DOI: `10.2139/ssrn.4467406`.

[52] N. H. Walfield and M. Brinkmann, “A critique of the gnu hurd multi-server operating system,” *ACM SIGOPS Operating Systems Review*, vol. 41, no. 4, pp. 30–39, 2007.

[53] D. Vemuri and W. Alhamdani, “Measures to improve security in a microkernel operating system,” Sep. 2011. DOI: `10.1145/2047456.2047460`.

[54] *Documentation — darnassus.sceen.net*, `https://darnassus.sceen.net/~hurd-web/documentation/`, [Accessed 16-04-2024].

[55] R. Espinola, *Stallman’s Dream: GNU Hurd*, `https://raulespinola.wordpress.com/2009/02/12/el-sueno-de-stallman-gnu-hurd/`, [Accessed 16-04-2024].

[56] G. Hurd, *Gnu documentation*, 2023. [Online]. Available: `https://www.gnu.org/software/hurd/documentation.html`.

[57] *The confused deputy problem - AWS Identity and Access Management — docs.aws.amazon.com*, https://docs.aws.amazon.com/IAM/latest/UserGuide/confused-deputy.html, [Accessed 18-04-2024].

[58] *Debian – Debian GNU/Hurd 2014; Configuration — debian.org*, https://www.debian.org/ports/hurd/hurd-install, [Accessed 20-04-2024].

[59] *WSL2 vs Linux (HPL HPCG NAMD) — pugetsystems.com*, https://www.pugetsystems.com/labs/hpc/wsl2-vs-linux-hpl-hpcg-namd-2354/, [Accessed 20-04-2024].

[60] Debian, *Debootstrap*, https://wiki.debian.org/Debootstrap, [Accessed 21-04-2024].

[61] *FUSE &x2014; The Linux Kernel documentation — kernel.org*, https://www.kernel.org/doc/html/next/filesystems/fuse.html, [Accessed 22-04-2024].

[62] *Volumes — docs.docker.com*, https://docs.docker.com/storage/volumes/, [Accessed 22-04-2024].

[63] *GitHub - freshe/c-cpu-bench: Simple CPU bench / test — github.com*, https://github.com/freshe/c-cpu-bench, [Accessed 22-04-2024].

[64] *GitHub - josefbacik/fs$_m$ark: A file system benchmark tool — github.com*, https://github.com/josefbacik/fs_mark, [Accessed 22-04-2024].

[65] *GitHub - ssvb/tinymembench: Simple benchmark for memory throughput and latency — github.com*, https://github.com/ssvb/tinymembench, [Accessed 22-04-2024].