

IMPROVING LOW-LEVEL ISOLATION OF CONTAINERS INSIDE MICROKERNEL

C.J. PIETERSZ

2025



Improving Low-Level Isolation of Containers Inside Microkernel

C. J. Pietersz

Index No: **20001347**

Supervisor: **Dr. Chamath I. Keppetiyagama**

May 2025

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



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, to be made available for photocopying and for interlibrary loans, and for the title and abstract to be made available to outside organizations.

Candidate Name : C.J. Pietersz



.....

Signature of Candidate

Date: 29-06-2025

This is to certify that this dissertation is based on the work of Mr. **C.J. Pietersz** under my supervision. The thesis has been prepared according to the format stipulated and is of acceptable standard.

Principle/Co-Supervisor's Name : Dr. Chamath I. Keppetiyagama



.....

Signature of Supervisor

30-06-25

Date:

Acknowledgment

I would like to express my sincere gratitude to my supervisor, Dr. Chamath Kepingiyagama and co-supervisor, Mr. Tharindu Wijethilake for their valuable advices, guidance and feedbacks throughout this research. Special thanks to Mr. Ravin Perera for providing insightful perspectives that enhanced this work.

Finally, I wish to extend my heartfelt appreciation to my parents and Miss. S.H.P.S. Fernando for their unwavering support and encouragement, which greatly contributed to the completion of this work.

Table of Contents

1	Introduction	2
1.1	Background	3
1.2	Isolation	4
1.2.1	Importance of Isolation in Virtualization	5
1.2.2	Improving Isolation	6
1.3	Gap and Research Questions	6
1.4	Research Aims and Objectives	7
1.5	Research Scope	7
1.5.1	In Scope	7
1.5.2	Out Scope	8
1.6	Significance of the Research	8
1.7	Research Methodology and Evaluation Criteria	9
2	Literature Review	13
2.1	Virtualization	13
2.1.1	What is Virtualization	13
2.1.2	Key Features of Virtualization	14
2.1.3	Types of Virtualization	14
2.2	Containerization	16
2.2.1	Containerization Technologies	18
2.2.2	Container Security	19
2.2.3	Microkernel-based Containerization	20

2.3	Operating System Kernels	21
2.3.1	Monolithic Kernel Approach	21
2.3.2	Microkernel Approach	22
2.3.3	Hybrid Kernel Approach	26
2.4	Reliable Operating Systems	27
2.4.1	Current Operating System Issues	27
2.4.2	Solutions	28
3	Design	31
3.1	Selection of a Suitable Environment	31
3.1.1	Main Characteristics of GNU/Hurd & GNU/Mach	31
3.1.2	Setting Up a Working GNU/Hurd System	35
3.2	Containerization Within GNU/Linux	36
3.3	Containerization Within GNU/Hurd	38
3.3.1	Current Features	39
3.4	Initial Design - Handle Namespaces in the Kernel	41
3.5	Second Design Approach - Handle Namespaces in the Proc Server . . .	44
3.6	Analyse Isolation	45
3.7	Inherent Isolation Within GNU/Mach	45
4	Implementation	47
4.1	Environment Setup	47
4.1.1	Run the Disk Image using Qemu	48
4.1.2	Expanding the Disk Image Size	48
4.1.3	Compilation of GNU/Mach	49
4.1.4	Compilation of GNU/Hurd	50
4.1.5	Compilation of GNU/Mig	51
4.1.6	Compilation of GNU/Glibc	51
4.1.7	Challenges and Troubleshooting	51
4.1.8	Reproducibility	54

4.2	Implementation of First Design Approach	55
4.3	Implementation of Second Design Approach	56
4.3.1	Namespace Implementation	58
4.3.2	Enforce process Isolation	62
4.3.3	Design Decision	76
5	Results & Evaluation	77
5.1	Setup Environments	77
5.2	Isolation Analysis	77
5.2.1	Qualitative Isolation	78
5.2.2	Quantitative Isolation	83
5.3	Performance Comparison	84
5.3.1	CPU	85
5.3.2	File System	87
5.3.3	Memory	89
6	Conclusions	93
6.1	Limitations	93
6.2	Future Works	95

List of Figures

2.1	Application Deployment using traditional, hypervisor and container architecture (Bhardwaj and Krishna 2021)	17
2.2	Simple architecture of a modular monolithic system (Tanenbaum and Woodhull 1997)	22
3.1	Subhurd Architecture	41
3.2	Design Approach 1	42
3.3	Design Approach 2	44
4.1	Hurd Architecture (R. Espinola 2009)	52
4.2	/GNU/Hurd system failure during boot due to simple kernel changes . .	56
4.3	Simple Isolation	68
4.4	ls /proc behavior on simple isolation	69
4.5	Enhanced Isolation	73
5.1	Setup two namespaces to monitor filesystem behavior	80
5.2	Failure of namespace 1 filesystem	80
5.3	Re-invoke namespace 1 filesystem	81
5.4	Setup two namespaces to monitor procfs behavior	81
5.5	Failure of namespace 1 procfs	82
5.6	Re-invoke namespace 1 & 2 procfs translators	82
5.7	CPU Performance Comparison: Linux vs Hurd	85
5.8	Linux: Impact of Enhance namespace isolation on cpu performance . .	86
5.9	Hurd: Impact of Enhance namespace isolation on cpu performance . . .	87

5.10 Application Overhead:Linux vs Hurd	88
5.11 App Overhead across scenarios	88
5.12 Memory Bandwidth: Linux vs Hurd namespace	89
5.13 Namespace Isolation on Memory Bandwidth: hurd	90
5.14 Namespace Isolation on Memory Bandwidth: linux	90
5.15 Memory Latency: Linux vs Hurd namespace	91
5.16 Namespace Isolation on Memory Latency: hurd	91
5.17 Namespace Isolation on Memory Latency: linux	92

List of Tables

3.1	Key Containerization Features in Linux Kernel	38
3.2	Comparison of Containerization Features	39
4.1	Comparison: Namespace Design inside Kernel Space vs User Space . .	57
5.1	CPU Information of the Host Laptop	78
5.2	Test Environment Configuration: Linux vs GNU/Hurd	85

List of Acronyms

IPC Inter Process Communication

VM Virtual Machine

VMM Virtual Machine Monitor

LXC Linux Containers

KVM Kernel-based Virtual Machine

QEMU Quick Emulator

IOT Internet of Things

OS Operating System

CPU Central Processing unit

HPC High-Performance Computing

RPC Remote Procedure Calls

ARM Advanced RISC Machine

SKI Single Kernel Image

NS NAMESPACE

Abstract

Namespace isolation plays a major role in modern containerization, enabling secure and efficient process separation in operating systems like Linux. This research introduces a novel approach to implementing namespace awareness in the GNU/Hurd, a system traditionally lacking such mechanisms due to its minimalist design. Two design approaches were evaluated: embedding namespace awareness within the Mach microkernel versus implementing it externally in Hurd's user-space servers. The latter was chosen, aligning with Hurd's philosophy of excluding policies from the kernel and adhering to minimalist design, thus maintaining system modularity & flexibility. In this study, a new `unshare` command is proposed on the selected design, achieved through targeted modifications to Hurd's `proc` server & message passing interfaces, to enable process namespace isolation. The implemented `unshare` command successfully isolates process namespaces, and when combined with `chroot`, it achieves enhanced isolation comparable to Linux container mechanisms. Using the same design methodology, further namespace isolations can be developed with additional research and optimization. Performance evaluations demonstrate that the isolation introduces acceptable overheads in Hurd, with memory bandwidth and latency impacts higher than Linux's. . Specifically, memory bandwidth and latency impacts are 7–15% and 5.4%, respectively, compared to Linux's 0–1.3% and 1.0%. These results validate the feasibility of namespace isolation in micro-kernels, showcasing Hurd's potential for container-like functionality, and pave the way for future research into broader namespace support.

Chapter 1

Introduction

Microkernels, characterized by their minimalist architectural design and strict separation of concerns, provide a robust framework for isolating core system functionalities such as memory management, scheduling, and Inter Process Communication (IPC) from user-space applications (Tanenbaum and Woodhull 1997). This inherent isolation, which removes device drivers, file systems, and other utilities from kernel mode to user mode, has long been regarded as a security advantage over monolithic kernels, where such components are tightly integrated (Tanenbaum, Herder, and Bos 2006). Given this foundation, a critical question arises: can the isolation capabilities of microkernels be leveraged to support containerization, a paradigm that fundamentally relies on isolation? This research addresses this question by designing, implementing and evaluating namespace awareness in the GNU/Hurd Operating System (OS), a system traditionally lacking such mechanisms. Adhering to the proposed design we developed a new `unshare` command that utilizes implemented Remote Procedure Calls (RPC) call by modifying Hurd's `proc` server, enabling process namespace isolation, and combined it with `chroot` to achieve enhanced isolation akin to Linux container mechanisms. Two design approaches were considered: embedding namespace awareness within the Mach microkernel versus implementing it externally in Hurd's user-space servers. The latter was adopted, preserving Hurd's minimalist philosophy. Performance evaluations using `cputest`, `tinymembench`, and `fs_mark` benchmarks reveal that Hurd achieves effective iso-

lation with acceptable trade-offs. Memory bandwidth and latency impacts are 7–15% and 5.4%, respectively, compared to Linux’s 0–1.3% and 1.0%. Although implementation is limited to process namespaces due to time constraints, this approach demonstrates the feasibility of container-like isolation in microkernels and lays the groundwork for future research into broader namespace support. By showcasing the potential of microkernels, comparing design approaches, and analyzing performance trade-offs across memory, computational, and filesystem operations, this work contributes to the ongoing research on the future of containerization in alternative kernel architectures.

1.1 Background

The GNU project was started by *Richard Stallman* in 1983 to give a free OS and supported software to all people just like Air is free. Because of that almost every utility, program(extended) in the UNIX system is developed by worldwide developers for free use. But the main part missed was the kernel for the OS. That’s when new microkernels and monolithic kernels emerged like *Trix*, *Mach*, Linux. Gnu project first used different types of kernels and all were not fully developed yet. In 1981 they decided to have Linux kernel developed by a student named *Linus Torvalds* over the GNU/Mach microkernel which was developed by some expert developers but wasn’t fully developed - (Tanenbaum, Herder, and Bos 2006). After that, all the GNU software and hardware created to support Linux and Linux developed rapidly while Gnu/Hurd which uses Gnu/mach as the underline kernel slowed down the development due to the lack of contribution. So, the world of OSs has been built around monolithic and hybrid kernels. Researchers have long debated the merits of monolithic kernels versus microkernels. Since we now have efficient and faster hardware support, enthusiasts all over the world are thinking about the revolution of powerful secure microkernels.

Despite being revolutionary, the earlier microkernels discussed above frequently experienced performance overhead, less development contribution, and limited adoption - (Tanenbaum and Woodhull 1997). While the GNU Hurd remains in microkernel

philosophy, its special purposes had a limited impact on the OS mainstream. However, the underlying ideas behind microkernels have remained fascinating to researchers and developers.

Early virtualization improved with the rise of multiprogramming and OSs since the 1950s, with the sole purpose of having safely shared physical machine resources between several processes, but it expanded into an isolated chunk of shared resources from the host machine. However, all implementations of virtualization (Virtual Machine (VM)s & Containers) expose a surface more than necessary to the underlying kernel. Therefore the features of the OS kernel directly affect the security of the virtualized environment - (Randal 2020). Traditional containerization solutions, while powerful, face inherent limitations in security and isolation in distributed environments because of the dependency on current kernel architectures. This is where the microkernel re-enters the scene, with its inherent isolation potentially offering a solution.

1.2 Isolation

In the realm of virtualization and containerization, isolation is broadly understood as the prevention of unintended interactions between distinct systems or processes, encompassing data sharing, resource access, or behavioral interference. There isn't an accepted standard definition for isolation. Hakamian and Rahmani define a properly isolated virtualized system as one that confines programs within their own boundaries, preventing them from affecting other systems or the host (Hakamian and Rahmani 2015). This research adopts a similar perspective, defining isolation as the capability to restrict a process's interactions across namespaces in the GNU/Hurd microkernel, ensuring that processes within a namespace cannot access or influence processes, filesystems, or resources in another namespace, whether maliciously or inadvertently. However, isolation is a multifaceted concept, spanning security, performance, and fault propagation dimensions, which renders comprehensive measurement and comparison challenging (Popek and Goldberg 1974).

This study specifically focuses on process namespace isolation within GNU/Hurd, capitalizing on the microkernel’s inherent architectural advantages to enhance container-like isolation. Unlike monolithic kernels like Linux, which expose a large attack surface through approximately 360-441 system calls (Torvalds 2025), microkernels such as GNU/Mach minimize the kernel’s role, delegating functionalities like filesystem management and process handling to user-space servers. This design reduces the attack surface, as GNU/Mach employs around 197 message-passing interfaces (GNU Project 2025), which are less numerous and more dynamic than Linux’s system calls, thereby limiting potential attack vector. Moreover, microkernel isolation provides qualitative benefits, including tolerance to parent namespace failures and unchanged kernel behavior, ensuring that namespace-specific failures do not propagate system-wide (Popek and Goldberg 1974). In Hurd, these advantages are realized through features such as isolated filesystems per namespace, local and shared system servers per namespace, and the use of translators like `procfs` to enforce namespace boundaries, as demonstrated in the proposed design. These attributes position microkernels as a promising foundation for containerization.

1.2.1 Importance of Isolation in Virtualization

Isolation is a cornerstone of virtualization, as articulated by Popek and Goldberg, who defined a virtualized system as an “efficient, isolated duplicate of a real computer machine” (ibid.). This significance has intensified with the advent of containerization, where isolation directly influences security, performance, and fault containment. Inadequate isolation can lead to severe consequences, such as fault propagation, where a failure in one container disrupts others due to shared kernel dependencies. Microkernels like GNU/Hurd address these risks by isolating system components in user space, reducing shared areas and dependencies between processes, aligning with established methods for enhancing isolation.

1.2.2 Improving Isolation

Enhancing isolation involves minimizing shared areas and dependencies between systems, as exemplified by microkernel designs. In this research, namespace awareness in Hurd's proc server, combined with `chroot`, isolates filesystems and system servers per namespace, reducing cross-namespace interactions. Future improvements could integrate security frameworks like `seccomp` or SELinux-inspired policies to further confine processes, leveraging microkernel modularity to enforce mandatory access controls while preserving performance and fault tolerance (*namespace_selinux(8) - Linux manual page 2025*).

1.3 Gap and Research Questions

Containerization inside microkernels has not evolved to a stage that can be used to isolate code and dependencies into a single container and there aren't any implementations of containers popular within the industry leveraging this kernel architecture. Perera (2024) has highlighted containerization like technology inside Gnu/Hurd OS by leveraging subhurd system but, with several limitations. Such as, this research primarily concentrated on file system isolation, the isolation analysis doesn't delve deeply into drivers & other utilities, the research does not extensively explore high technical aspects of Gnu/Hurd OS and its system servers & subhurd has its own system servers replicating hurd system servers, make this design more like a VM ignoring lightweightness of containers. So according to that, additional isolation using subhurd hasn't been explored and implementation doesn't touch the Mach microkernel as well as Hurd servers which may cause to exhibit additional performance limitations. It is important to address and explore how the architectural advancement of microkernels can be applied to containers to gain greater isolation. The research intends to fill that gap by continuing the discussion initiated by Perera (ibid.) about the relationship between containers and microkernels. The research primarily revolves around exploring the following research questions.

- **What is the suitable architecture to implement namespaces inside microker-**

nels?

- **What are the advantages/disadvantages of hurd-based containers compared to Linux-based containers?**

1.4 Research Aims and Objectives

The research aims to investigate the potential of implementing containers inside the Hurd OS to improve isolation using the architectural advantages and special characteristics provided by its kernel design, specifically using in-kernel features. The research process intends to achieve the following objectives.

- Design the architecture for the modifications and addition of services such as namespaces into the Gnu/Hurd to support containers at low levels.
- Add the necessary improvements to the Gnu/Hurd OS and Gnu/Mach according to the designed architecture.
- Implement a primitive version of containers using implemented kernel features such as namespaces to isolate processes.
- Assess the isolation provided by the implemented namespace inside the Gnu/Hurd.
- Assess the performance, attack surface, and security of implemented namespaces inside the Gnu/Hurd.
- Evaluate results by discussing limitations, issues, advantages, and disadvantages using the outcomes.

1.5 Research Scope

1.5.1 In 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 monolithic kernels.
- Explore the capabilities of Hurd OS to isolate the processes of each container.
- Design an architecture to implement namespaces in Hurd OS.
- Implement a primitive version of a container using features provided by GNU/Hurd.
- Analyze the overall isolation provided by the aforementioned setup concerning the process isolation.
- Measure the performance of the aforementioned setup to analyze potential & limitations.
- Suggest future work and improvements to the topic.

1.5.2 Out Scope

The capability of implementing an entire container engine that could support many features in platforms such as Docker can be extremely challenging due to the time constraints of the research. Popular container engines use many different kernel services other than process isolation and analyzing all these services and how much microkernel handles isolation in each area can be difficult and time-consuming. Due to the strict timeline of the research, the project will only focus on the process namespace isolation of a container which is the first step in namespace isolation and any further improvements to design will be considered as future work.

1.6 Significance of the Research

Strong isolation between and within containers is a major challenge due to sharing the same kernel features underlined. Although perfect isolation is unachievable, advance-

ments in areas such as microkernel architecture that have the potential to contribute to the inherent isolation can be a game changer. Although, the same isolation in VMs can't be achievable using containers, using microkernel architectural advancement to gain better isolation since less kernel-level privilege is used can be beneficial. The goal of this research is to improve process namespace isolation inside a microkernel, by ensuring the safety and resilience of containers. Progress in this area in the future could lead to better isolation in other important namespace isolation technologies. Prioritizing designs that limit kernel interference is imperative for developers to reduce the danger of kernel exploitation, which has the risk of compromising all containers on a single host. Microkernels' overhead in IPC is one of their main drawbacks. A performance comparison of many research papers reveals that monolithic design is preferred over microkernel architecture because of this significant difference. However, considering the advancements in technology, it is reasonable to be optimistic about the huge computational power that will be available in the future. Hence, keeping aside the performance limitations of microkernels and implementing future advancements will prove their usefulness in the future. This trend has been seen in research on neural networks, virtual reality, and related fields where theoretical advances have frequently come before the necessary tools to prove their usefulness. Looking ahead to the future of computing with optimism, it makes sense to assume that because of their superior architecture, microkernels will become the standard. When that time comes, this research will provide a strong basis for shifting containers and other virtualization technologies to adjust to the change in architecture.

1.7 Research Methodology and Evaluation Criteria

The design science research methodology will be the main focus of the research. Three cycles will comprise the process, as stated in Hevner (2007). The relevance cycle will transfer the requirements, evaluation criteria, and problems from the environment to the design science research. The appropriate artifacts that meet the specified requirements

and evaluation criteria will then be produced by the design cycle. The newly discovered information and artifacts will be added to the knowledge base by the rigor cycle. The key aspects applicable to the three cycles are listed below.

- Stakeholders
 - Researchers related to the area of research
 - Primary users of containers and virtualization
 - Industries that use containers and virtualization
- Requirements
 - Improvements to isolation in containers at a lower level closer to the microkernel
 - Reduction of potential vulnerabilities and faults in the kernel that can affect the performance or reliability of the container infrastructure hosted on top of it
 - improvements to the microkernel at the lower level to support container isolation and reliability.
 - Identify the relationship between the inherent isolation provided by the microkernel architecture and the current isolation provided by monolithic kernels at lower-level
- Artifacts
 - A primitive version of container implementation utilizing or implementing the services and features of the Mach microkernel
 - An addition of comprehensive knowledge regarding the relationship between microkernels and container isolation to the related field
- Evaluation criteria

- The result will be compared with metrics of existing Linux-based systems for comparison
- Additions to the knowledge base
 - A research paper providing a detailed explanation of the methods, findings, 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 which is used by large tech companies and therefore has business relevance to it.
- Design Evaluation
 - The methodology consists 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 necessary 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

Chapter 2

Literature Review

2.1 Virtualization

2.1.1 What is Virtualization

In the 1960s, virtualization emerged as a method to manage time-sharing systems, addressing the high costs of providing individual mainframes for each user. During that period, the expense of buying separate mainframe units was impractical, leading to the practice of sharing a single mainframe among multiple users who accessed it in allocated time segments. To protect user privacy, it was essential to isolate their sessions, a problem that virtualization effectively resolved (Crosby and Brown 2006). The broad acceptance of virtualization got popular largely from industrial demands rather than academic exploration, making it valuable to define virtualization from an industry perspective (Sharma and Park 2011).

Some definitions for virtualization:

- The separation of a service request from the underlying physical delivery of that service - VMWare (VMWARE 2007).
- The abstraction of the computer hardware, that is, hiding the physical computer from the way in which it is used - Intel (Uhlig et al. 2005).

Virtualization can generally be understood as an abstraction that separates the physi-

cal properties of computing resources, establishing a clear division between hardware and software elements (Sharma and Park 2011).

2.1.2 Key Features of Virtualization

Although virtualization is applied in numerous fields, its most significant use lies in cloud computing. This preference arises from its ability to simplify the allocation and administration of cloud computing infrastructure, thereby reducing both expenses and operational difficulties (Bhardwaj and Krishna 2021). Cloud computing integrates hardware, storage, networks, and services to deliver computing resources as a service. In the context of cloud computing, virtualization demonstrates three main characteristics (Tamane 2015).

1. **Partitioning:** A single physical system should be able to run multiple software systems by partitioning various resources
2. **Isolation:** Each virtualized system can exist in isolation without affecting other systems or the host system. This should also isolate data among those systems.
3. **Encapsulation:** A virtualized system should be able to package the software into a single file or bundle. This improves portability and safety as each bundle is separate and less dependent from others.

2.1.3 Types of Virtualization

Historically, virtual machines that used hypervisors have been the major virtualization solution. However, advancements in technologies over time have paved the way to various alternative solutions, each highlighting distinct aspects of virtualization (Bhardwaj and Krishna 2021). These are the two main solutions that is highly used by many industries over the years.

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 (IBM 2025).

A VM acts as an emulated version of a physical computer system, often called a guest, with the physical machine it operates on referred to as the host. Rather than directly accessing physical hardware, a VM relies on a lightweight software component known as a hypervisor to mediate interactions with the host system. The hypervisor controls the distribution of physical resources, such as processors, memory, and storage, to each VM, ensuring their isolation to avoid interference among them. By deploying a hypervisor on a physical computer or server (commonly known as a bare metal server), the operating system and applications are decoupled from the hardware. This allows the physical machine to divide itself into several independent VMs, each able to run its own operating system and applications independently while utilizing shared resources like memory, RAM, and storage, all managed by the hypervisor (ibid.).

2. Containers (Operating System layer virtualization)

This concept, also known as Single Kernel Image (SKI) or container-based virtualization, operates by concurrently executing applications on top of a single

kernel image separately using OS level policies & restrictions. Consequently, the virtualization occurs at the level of the host OS rather than at the hardware level. All VMs use an virtualized image (bundled applications & libraries), referred to as the virtual machine image herein. This 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 & lightweight than alternative virtualization methods, though it doesn't provide complete isolation. However, as VMs share the kernel with the host OS, compatibility necessitates that the guest virtual image matches the host virtual image, making scenarios like running Windows on a Linux host unfeasible (Landaeta 2024).

The categorization of containerization, which involves the use of containers, as a type of virtualization remains a topic of debate within the community. However, a detailed analysis of virtualization definitions suggests that it fits within the broader virtualization framework. A growing number of researchers are recognizing operating system-based virtualization, including containerization, as essential elements of the wider virtualization landscape (Bhardwaj and Krishna 2021; Landaeta 2024; Tamane 2015).

2.2 Containerization

Sharing physical resources between several users and applications is considered one of the major reasons for initiating current cloud services. Each guest OS in the cloud (installed in each user) thinks it controls the whole computer & resources and doesn't know they are residing in a virtualized environment and share the hardware resources (Central Processing unit (CPU), memory, cache, devices) - (Watada et al. 2019). Two main technologies implement virtualization, which are virtualization-based hypervisors & virtualization-based containers. Virtualization using hypervisors is considered to have less performance than using containers because, hypervisor technology installs a new OS (guest) on top of the host OS, while containers share the same OS kernel as in figure 2.1.

Therefore, containers are smaller. Containerization technology uses virtualization-based containers to isolate from the other applications - (Bentaleb et al. 2022).

The main reason to use containerization is it contains all dependencies & OS libraries needed for its application so that the application can run in any environment without concern about dependencies or libraries (portability & platform independence) - (Prins 2022). While at present containerization is used vastly by not only large tech companies but also developers, and students to develop and deploy their products, Let's take a look at the history of this technology to get an idea about their performance, reliability & security.

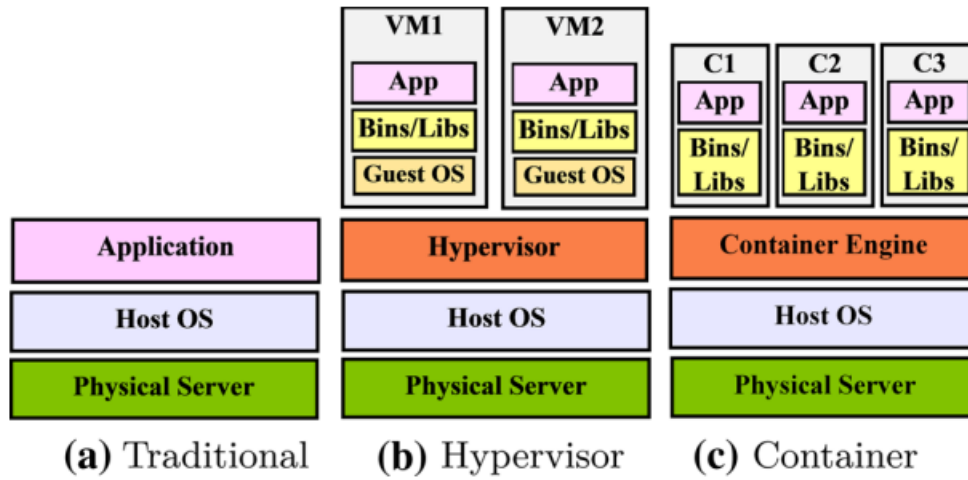


Figure 2.1: Application Deployment using traditional, hypervisor and container architecture (Bhardwaj and Krishna 2021)

The complexity of the system software increased with the multiprogramming because concurrent processes interacted with other processes and started to share multiple hardware resources like memory. In response to this, a privileged kernel was introduced to the system software responsible for managing every other process and resource (Randal 2020). While improving this isolation, the origin of the containers began with the introduction of the chroot system called in the seventh edition of UNIX OS by *Bell Labs*. filesystem namespace isolation introduced by chroot influenced the development of containers. Containers are not using solely chroot, it uses **namespaces**, **cgroups**, **seccomp**, and **capabilities** to provide isolation (ibid.).

Term capability in theoretical work introduced by Dennis and Van Horn (1966), de-

defined the set of memory segments a process allowed to read, write, or execute. The **early capability systems** prioritized security over complexity. The security working group of the POSIX standards project began extending the POSIX.1 standard, which introduced a capability feature. POSIX capabilities (a set of flags that determine whether a process is allowed to perform specific actions, etc) are entirely different implementations of early capabilities, but similar concepts were there in both - (Randal 2020). FreeBSD added jails to the system Kamp and Watson (2000), which isolated filesystem **namespaces** using chroot in an improved manner. It isolated the filesystem, processes, and network resources inside a jail so that the process has the root access privilege inside the jail but can't perform operations beyond the jail. Biederman and Networx (2006) proposed expanding namespaces isolation in the Linux OS to isolate process IDs, IPC, and network stack. Between 2006 and 2007, the process containers feature was introduced to the Linux kernel, later renamed to **cgroups** (control groups) - which introduced resource limiting, prioritization, accounting, and control features for processes. **Seccomp** is a feature added to the Linux kernel to restrict processes so that it can only run limited system calls to work with already open file handles and to terminate a process attempting to run any other system call. In 2008, to create low-level containers, Linux Containers (LXC) combined cgroups, namespaces, and capabilities from the Linux kernel. (Randal 2020).

2.2.1 Containerization Technologies

Container-based virtualization technology has been widely used in several industries. In this chapter, Let's discuss briefly some popular container technologies: Docker, and Singularity, each focusing on different use cases.

Docker

Docker is a lightweight container-based virtualization platform. Docker extended functionality from LXC, called libcontainer (ibid.). The performance of LXC and Docker was contrasted by Morabito, Kjällman, and Komu (2015) and found both are roughly equal in CPU performance, disk I/O, and network I/O except in random writes(LXC

performed 30% better). Docker container uses namespaces, and cgroups to provide process & resource isolation. Namespaces (user, net, PID, mnt, cgroup) will limit the user's space and provide isolated Linux kernel resources. creating, running, and managing containers easier with the help of Docker. Docker images can be built using .Dockerfile in the project folder, which contains commands to initialize and run the image. There is a container registry called Docker Hub that can be used to share and save images (can manage different versions). To run microservices using several containers (multi-containers) and manage those containers, we can use docker-compose. (Bentaleb et al. 2022)

Singularity

While docker focuses on applicability in the industry, Singularity focuses on the portability of the containers, optimized for High-Performance Computing (HPC) environments. HPCs are usually used in scientific research, engineering simulations, etc. Singularity is a pattern, which allows users to deploy and create their execution environments. Singularity can be used to configuration of namespaces for containers and minimize the number of virtualized namespaces. Singularity allows untrusted users to run untrusted containers safely. The security design of the singularity uses kernel security modules like SELinux, and AppArmor and can load security policies at runtime. (ibid.)

2.2.2 Container Security

Containerization comes with some security benefits. providing "secure-by-default" approach container management engines support the same isolation techniques as the host OS. So, Security permissions will be managed by the engine itself to check communication between containers - (Prins 2022). Technology doesn't come with 100% security & reliability. All the current VMs and containers are weak abstractions, providing underlying software surface or hardware than it is supposed to the outer environment through the OS kernel. Security research conducted in 2018 showed that isolation of virtualized environments (guests) can be easily broken using some hardware vulnerabilities related

to speculative execution like Spectre Meltdown, Foreshadow, L1TF, and variants. The large companies using virtualization in cloud-based environments know these security vulnerabilities & risks, but they ignore them because of the benefits like flexibility, performance, cost, portability, and customer reach. (Randal 2020). Implementation of Docker, Quick Emulator (QEMU)+Kernel-based Virtual Machine (KVM), and Xen on Advanced RISC Machine (ARM) hardware architecture has been done by Raho et al. (2015). Adversaries can take over all containers residing on the same host by subverting the Linux kernel. Linux isolation is strict, but it's not secure in front of ARP poisoning. The kernel provides this security model using kernel security models: Seccomp, Apparmor, and SELinux.(Watada et al. 2019).

Considering these security & isolation issues, more secure and reliable solutions are strongly needed!

2.2.3 Microkernel-based Containerization

Till now containers and microkernels have been discussed separately. Containerization is very popular with monolithic OSs like Linux, Windows, and Apple Mac but not with microkernel-based OSs like Gnu Hurd, Minix, or SeL4. The existence of microkernel-based containerization technologies currently in use is discussed below.

To overcome some container security shortcomings discussed in Chapter 2.2.2, many researchers proposed using unikernels (minimal OSs focused on security used for special purposes & are not general-purpose complete OSs) to create or manage containers. But these can be used for special needs like secure military vehicle cybersecurity - (Prins 2022).

Prins (ibid.) have discussed containerization with SeL4 microkernel. SeL4 is considered a secured microkernel since its formal verification, without sacrificing performance, means it has strong isolation. Here, SeL4 Virtual Machine Monitor (VMM) is used to separate criticality levels into separate VMs, and then each VM can run various containers by using SeL4 security benefits. While these containers had nearly native performance, the boot time of VMs and the initializing time of the container engine make

it slower. However, native support for namespaces and isolation in SeL4 microkernel has greater security advantages toward containerization.

2.3 Operating System Kernels

OSs are required by Central Processing unit (CPU)s to manage their resources and abstract hardware from applications that are running on them and the users who use those applications. So, mainly OSs perform two basic functions, extending the machine and managing resources as stated by Tanenbaum and Woodhull (1997). An OS is a group of system software applications that has many functions like management of the memory units, management of the process creation to termination, handling IPC, and managing input/output(I/O) devices (Isaac et al. 2021). A CPU has privilege modes that are typically named from 0 to 3. 0 is the most privileged mode (kernel mode) of all and 3 is the least privileged level (user mode). An OS kernel is the set of processes that are running in the most privileged mode where it can control all the hardware and execute any instruction that the machine's hardware has the capability to run. Every other software process runs in user mode. The kernel is identified as the heart of the OS. There are two major approaches to designing a kernel architecture, namely the monolithic kernel architecture approach, and the microkernel architecture approach. Let's discuss the above-mentioned kernel architecture solutions in detail.

2.3.1 Monolithic Kernel Approach

The most common approach by far, has no standard structure, subtitled as “The Big Mess” by Tanenbaum and Woodhull (1997). The OS is a collection of procedures, each one can call any of the other ones whenever it wants to. Each procedure has a well-defined interface that other procedures can call. As the name suggests, there is no information hiding except for structures designed using modules or packages, in which much of the information is encapsulated inside modules, and only the designated entry points are visible to other modules. Using modules, it is possible to design the system

with a little structure like the example in figure 2.2.

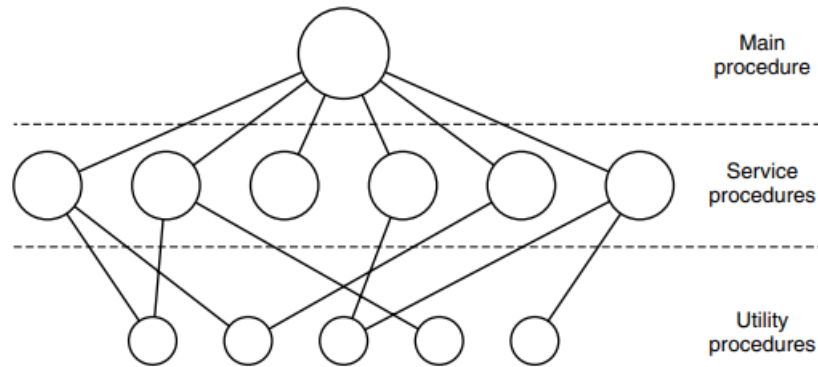


Figure 2.2: Simple architecture of a modular monolithic system (Tanenbaum and Woodhull 1997)

Even if every module producing these procedures is separate from the whole, the code integration is very tight and difficult to perform effectively. Since all the modules run in the same address space, an error in one module may affect the whole system. More modern monolithic kernels such as Linux, FreeBSD, and Solaris can dynamically load (and unload) executable modules at runtime. This modularity of the kernel is not at the kernel architecture level. Instead, it is at the binary (image) level. No one should conflict modular monolithic kernels with the architectural level modularity inherent in microkernels - (Stankov and Spasov 2006). Due to this “System as a whole” structure, monolithic kernels can work near userland processes, thus it can predict resource usage patterns, and more components directly interact with scheduler, memory & process management units. So, the system can have a higher-level view of how users and processes use resources. (Walfield and Brinkmann 2007).

2.3.2 Microkernel Approach

As the demand for the OS grows, the kernel becomes larger and more complex, the microkernel approach solves this problem by providing only a minimal set of services. Even though we don’t hear much about this architecture, it has been there since the 1980s. Researchers and developers didn’t pay attention much because of its inherent

performance overhead over the inherent security it provided at that time. After the GNU project decided to choose Linux as its kernel, All the software and hardware support for Linux started emerging. Also, the researchers have focused on Linux. That's where enthusiasts who are interested in microkernels have to slow down the development due to low contribution. But with the people realizing the true benefits of higher security and reliability, people are starting to develop OSs around microkernels. Therefore, we can see microkernels are used in several products: mobile phones, embedded devices, distributed systems, Internet of Things (IOT) devices, and mission-critical systems like medical devices.

Unlike monolithic kernels, microkernels have a very minimalistic design, providing only the most essential OS functions ensuring enhanced security, modularity, isolation, and adaptability. These systems can be expanded with more services without modification to the core. The kernel's reduced size and straightforward design simplifies security and accuracy checking. Based on their design principles and development we can identify several microkernel generations. 1st generation establishes the core concepts and limited functionality will be there in the kernel mainly focusing on IPC and memory management. 2nd generations were focused on the limitations of 1st generation and improved performance, security, and modularity. 3rd generation microkernels can work with practical applications like real-time systems and try to improve usability. To understand a handful of trends in the microkernel-based OSs design let's review several microkernel-based OSs and underline components.

Microkernel-based Operating Systems

1. Mach Microkernel

The effectiveness of three microkernels: Amoeba, Mach, and Chorus was compared by Tanenbaum (1995). Comparison has been conducted focusing on Memory Management, process management, and IPC. Similarities and differences in the components of the three microkernels were assessed. Mach, considered as a 1st generation microkernel, was first developed by the University of Rochester as

RIG and then by Carnegie-Mellon University (CMU), It was not a microkernel at first but then developed into a much smaller microkernel. Because of limitations in interprocess communication, Mach experienced a performance overhead. Mach kernel ran on a variety of supercomputers. Mach threads are managed by the kernel and provide extensive support for multiprocessor systems and are based on processes, threads, memory objects, ports, and messages. Despite each microkernel being developed by a different group, all three have numerous similarities.(Tanenbaum 1995)

2. L4 Microkernel

German National Research Center for Information Technology developed the L4 kernel in 1995. which can be considered as 2nd generation microkernel. The development of L4 microkernel was discussed over 20 years. Performance is improved by designing a better synchronized IPC approach. IPC has rich semantics and the minimality design was supportive of gaining the performance.(Elphinstone and Heiser 2013)

3. L4RE Microkernel

The L4re microkernel is a later version of the original L4 microkernel. The system follows the same design principle as the L4 family. Any component in the kernel cannot be removed from the kernel space. if so, it'll affect the proper functionality of the system. At the lowest level of the L4re microkernel is the Fiasco-OC microkernel. it runs in the highest privilege mode. L4re microkernel is compatible and scalable in embedded devices and High-Performance Computing (HPC). So, it was used by secure systems, mobile phones, and the automobile industry. (Rana et al. 2023)

4. SeL4 Microkernel

This highly secure microkernel was considered as a 3rd generation microkernel. The design goal of this kernel is to use this in security and safety-critical systems

Rana et al. (2023). The implementation of this microkernel was formally machine-checked and proven for functional correctness and the formal model applies integrity and confidentiality in Klein et al. (2014). seL4 uses capabilities for access control.

5. GNU Hurd Microkernel

Gnu Hurd is an open-source, Unix-compatible microkernel-based OS that has continued development till now. But not completely suitable for day-to-day usage due to lack of contribution thus slow development. In Walfield and Brinkmann (2007), critique the architecture of the Gnu Hurd and assess it in terms of user environment focusing on security. Hurd consists of a set of objects: system services represented as kernel objects. Capabilities are used to authorize access to objects. The system uses Remote Procedure Calls (RPC) to communicate between servers and clients. Also, this paper discusses Hurd's architecture in different aspects like structure, security, abstraction, etc. Hurd architecture values both security and flexibility. The Hurd OS has used Mach 3.0 as the core once. but due to some compatibility issues, Hurd tried different microkernels and now it uses its microkernel. But, seems some shortcomings are still there to remedy.

6. Minix 3 Microkernel

Andrew S. Tanenbaum wrote the Minix OS, compatible with Unix but the underlying design is different from one another. In 2004, he & some of his students developed Minix 3, a major redesign of the Minix system by restructuring the microkernel and increasing reliability & modularity. It is an open-source project and was developed for use in both PCs and embedded systems. Minix 3 provides three primitives for communication: send, receive, send & wait, which are C library procedures. scheduler uses a multilevel queuing system. (Tanenbaum and Woodhull 1997). Although Minix 3 hasn't released a stable version since 2014, It is a great source to learn about OSs from the start.

7. QNX Neutrino Microkernel

QNX Neutrino is a vastly used proprietary real-time microkernel-based OS in embedded devices due to its reliability, security, and performance. In Stankov and Spasov (2006), a comprehensive comparison was done between QNX Neutrino and Linux-based distribution: RedHat Embedded. The use of microkernel in QNX gives inherent distributed characteristics and real-time behavior to the OS. In this comparison, the conclusion is that QNX neutrino is well-suited for real-time OSs because of its performance, security, better memory management, and real-time supportiveness. Still, in performance, both OSs had a fast performance.

2.3.3 Hybrid Kernel Approach

A hybrid kernel is another type of OS kernel that combines characteristics of both microkernels and monolithic kernels. Modern OSs use this approach to acquire better security than monolithic OSs & performance than microkernel-based OSs. That is, they are using a microkernel as the core of their kernel to handle essential tasks and integrate the kernel with some additional functionalities like virtual memory management, networking, and security mechanisms. Remaining services like device drivers can run as user processes separate from the kernel. However, in this approach, not all device drivers will be thrown out from the kernel space. So, hybrid kernels are a kind of compromise between security and performance while introducing some complexity in their design. For example:

- Apple Mac OS uses a hybrid kernel method by integrating with a modified Mach microkernel.
- Android uses a hybrid Linux kernel.
- Windows uses Windows NT (influenced by microkernel design) as the kernel

2.4 Reliable Operating Systems

"Improving reliability can also improve security" is a popular phrase stated by Tanenbaum, Herder, and Bos (2006). Current computers are not reliable due to sudden crashes, implored users to download some emergency software updates from the internet, etc. While TV sets, DVD recorders, MP3 players, and other software-laden electronic devices are reliable and secure, computers are not! A few major reasons are computers are flexible, anyone can change the software, the IT industry is immature, users haven't much knowledge about security or reliability, etc. To make computers reliable, the main thing we need to delve into in this case is the OS. Because even application programs contain many flaws, if the OS is reliable & secure, bugs in other programs can do only a limited harm to the system. (ibid.)

2.4.1 Current Operating System Issues

As Tanenbaum, Herder, and Bos (ibid.) says, Current OSs have two features that make them unreliable and insecure: "They are huge and have very poor fault isolation". Currently, the Linux kernel has about 20 million lines of code, and the Windows kernel is more than twice the size of the Linux kernel. And day by day they are getting even bigger. As the code itself gets bigger and bigger the error count of that code will be very high. Typically, about 70 percent of the OS consists of device drivers. Because of this size, no single person can understand the whole system can be an issue when it comes to debugging and further improvements. The second issue comes with binding all possible modules or functionality with the OS: fault isolation. Although the kernel consists of modules, they all live together in the same container. If one module or procedure fails or crashes the whole container will be affected. For example, if a virus or worm infects one procedure, it can easily spread to others. In early times people didn't have much need for security and reliability but performance. With the change in human needs and behaviors, now we are in an era where people consider security and reliability more.

2.4.2 Solutions

To this reliability problem, there are 4 approaches discussed in Tanenbaum, Herder, and Bos (2006). The noticeable thing in these solutions is that 3 of those 4 approaches use microkernels! Let's discuss those approaches.

Armored Operating Systems

This approach called the *Nooks project*, is designed to improve the reliability of existing OSs like Linux and Windows without affecting their monolithic structure by making device drivers, the core reason for the problem less dangerous. *Nooks* tries to do this by wrapping each device driver in a layer of protective software (wrapper) to form a kind of protection domain. *Nooks* project's goal is to protect the kernel against buggy device driver failures and recover automatically when a driver fails by performing as few changes as possible to the existing drivers and kernel. When a driver loads into the kernel, there are several functions provided by the driver to the kernel and vice-versa. *Nooks* provides wrappers for all those such functions and then each function call from the kernel to drivers and vice-versa will go through a wrapper and will be checked and monitored for validity. Upon successful completion of the calling request (the operation will be performed to a copy of the kernel object), the isolation manager in the wrapper will change the kernel accordingly. In this way, a driver failure will not directly affect to the kernel. After a failure happens, the user-mode recovery agent runs and tries to restart the driver using recorded old logs. While this method is a proven better way to catch driver failures, it's not perfect. because still drivers are running in kernel mode and thus can perform any privileged instruction it should not execute, and *Nooks* have to write a large number of wrappers for each driver and those wrappers can contain faults.(ibid.).

ParaVirtual Machines

This approach tries to use virtualization to provide isolation and prevention from device driver failures. This approach was implemented by a research group at the *University of Karlsruhe* using the L4 microkernel as the host kernel. Instead of only one OS, this

uses two or more OSs in a virtual manager by slightly modifying Linux into L4Linux to match with paravirtualization (Old OSs were not developed enough to fully virtualize). With each OS thinking it has the entire machine to itself, this approach gained better isolation. The most interesting part here was this system uses one Linux machine to run the user applications while one or more other Linux machines run device drivers. By doing that, if a device driver fails, only the affected VM will crash, and other VMs will continue to work. An additional advantage of this is no need to modify the pool of device drivers. However, the kernel has to be modified slightly to achieve integration, and handle interrupts, and IPC. While this approach will have a performance overhead of about 3 to 8 percent, theoretically, it should provide greater reliability than a single OS because VMs can restart to their initial state after a crash happens. (Tanenbaum, Herder, and Bos 2006)

MultiServer Operating Systems

Even though the first two approaches focused on modifying legacy OSs, the latter approaches focus on a different architecture. This approach uses Minix microkernel as the core of the system and every other service and device driver will serve as separate servers (multi-server) by eliminating the buggy device driver problem. By executing device drivers as the user processes, one failure in one device driver will not crash the system because drivers are not in the kernel space, they can't perform privileged instructions directly without the kernel. Using a reincarnation server, it kills failed, crashed, buggy device drivers and restarts them. The use of separate instruction and data spaces will ensure no injected code or error can crash the system. While the performance loss that user-mode drivers cause is less than 10 percent when compared with the monolithic OSs, the Minix microkernel approach provides more reliable features like minimum code lines to minimize errors, self-healing properties, etc. (ibid.)

Language-Based Protection

This approach from a Microsoft research, called singularity is implemented using a new type-safe language named Sing#. All processes can run in a single virtual address space because language safety tightly constrains the system and processes. The compiler will not allow a process to meddle with the other process's data and eliminates kernel traps and context switches. This will provide both safety and efficiency to the OS. Each process is a closed entity and has its own code, memory, runtime, libraries, and garbage collector. That will ensure isolation in the system. Loadable modules like device drivers must run as separate processes because unverified foreign code can corrupt the system. This approach also consists of a microkernel (implemented using sing#) and a set of processes, all running in a common virtual address space. All process-to-process communication uses point-to-point bidirectional channels because sing# supports channels in the language. (Tanenbaum, Herder, and Bos 2006)

Even though microkernels have long been considered as lower performance than monolithic kernels, observing all four approaches and discussions, we can see microkernel approaches are more suitable than monolithic kernels when it comes to isolation, reliability, and security. And that is exactly what we expect in containerization technologies today!

Chapter 3

Design

3.1 Selection of a Suitable Environment

Access to an operating system based on a microkernel architecture is required for the research. Even though there are many microkernels, it can be difficult to find a reliable operating system that has the tools required to set up a container-like environment and run tests. Due to the presence of a highly developed operating system called GNU/Hurd, GNU/Mach was chosen as the preferred microkernel among the variety of microkernels that were available, including L4, seL4, GNU/Mach, Minix, and others. Supported by Debian, GNU/Hurd is completely integrated with the well-known GNU toolset and provides the ease of use of Debian's strong package manager, "apt." These qualities make GNU/Hurd a reasonable choice for the current research goals in this research. ("GNU Hurd/ Documentation" 2024)

3.1.1 Main Characteristics of GNU/Hurd & GNU/Mach

Hurd is a multi-server system

Many operating systems are based on Mach, but they are implemented as a single process running on top of the kernel, they have the same limitations as a monolithic kernel. All of the services that a monolithic kernel would normally provide are provided by this one process. Aside from the possibility of running several separate, isolated servers on

a single computer, this strategy might not seem very sensible. These kinds of systems are frequently called single-server systems. The Hurd, however, is unique in that it is the only workable multi-server system based on Mach. Numerous server programs manage various operating system services within the Hurd. These servers use Mach's message-passing capabilities to communicate and function as Mach tasks. Even though each server only provides a small portion of the system's capabilities, taken as a whole, they create a complete and functional operating system that is compatible with POSIX (R. Espinola 2009).

Mach ports

Within Mach, IPC is based on the idea of ports. A port facilitates one-way communication channels by acting as a message queue. It is required to have a corresponding port right (a capability) in addition to a port. This right can be send, receive, or send-once. Users can transmit a single message, receive messages from the server, or send messages to the server, depending on the type of port right they possess. There is exactly one task that holds the receive right for every port, but there may be zero or more senders. For clients who are waiting for a response message, the send-once right is advantageous. Along with the message, these clients have the ability to assign a send-once right to the reply port. The kernel makes sure that a message—which may contain a notification that the server has set up the send-once right—will eventually arrive on the reply port (Walfield and Brinkmann 2007).

Capabilities

We should discuss capabilities in relation to GNU/Mach and what they represent when discussing Mach ports. By fusing reference and protection features, a capability acts as a secure reference. It serves as a pointer to an object and is protected from forgery, guaranteeing its integrity. Essentially, a capability carries the power to manipulate the referenced object in addition to identifying it. Delegation procedures are streamlined when designation and authorization are combined within capabilities (“GNU Hurd/

Documentation” 2017).

Components of GNU/Hurd

- 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 can we locate a port to a selected server? This task requires a dedicated nameserver in Mach. If the intended server has previously registered with the nameserver, a task can query the nameserver to obtain a port to a server with send rights. But in Hurd, the file system server serves as the nameserver instead of having a separate one. To do this, the file system server provides the Hurd file name lookup RPC.

- * **Function as a data source** - The file system server manages pathname resolution and provides a port to the associated file (node) when a path is accessed, as was previously mentioned. When compared to a single nameserver, this method has several advantages. First of all, it makes it possible to limit access to a server by using standard Unix permissions on directories. Access can be managed by properly configuring a parent directory’s permissions and making sure there are no other ways to access a server port. But the ramifications go far beyond that. It is noteworthy that a pathname indicates a server port rather than a file. Because of this flexibility, the server can choose to serve static data from a regular file. As an alternative, data can be dynamically generated by a server. A server connected to /dev/random, for example, might provide new random data with each io read() on its port, whereas a server connected to /dev/fortune might provide a fresh fortune cookie with each open(). Other servers provide virtual information or a combination of

both, whereas a traditional filesystem server serves data as it is stored on disk. Every time a remote procedure is called, the server is in charge of making sure the data is consistent and useful. If this isn't done, the results could be confusing and different from what the user expected. (Perera 2024)

- Auth Server

- Identity-Based Access Control (IBAC) authorizes access based on user identity. As a result, a subject must identify itself to an object under such a system in order to gain access to it. In Unix, the majority of servers and the identity manager are located in the same trust domain. In the Hurd environment, this is different, which creates a significant difficulty: the object's managing server must be able to verify user identities, but it must also be prohibited from exploiting them. The Hurd's auth server solves this problem by enabling a three-way handshake that permits mutually insecure cooperation and sharing without the need for prior collaboration. Every port that connects to this server is used to identify a user and is linked to an ID block that includes both available user IDs and group IDs as well as sets of effective user IDs and group IDs. (ibid.)

- Password Server

- After receiving a Unix password, the password server issues a new authentication port and runs with root privileges. To ensure POSIX compliance, the IDs linked to the authentication port match the Unix user and group IDs. Additionally, this system incorporates support for shadow passwords. The password server, which runs as root and is located at /servers/password, exchanges Unix passwords with the auth server and verifies them using the password or shadow file. Many applications make use of this server, eliminating the necessity of directly managing passwords.

(ibid.)

- Proc server

- The process server is crucial for preserving system integrity because it acts as a central location for organizing information about the system. Opting out means giving up the Hurd system's POSIX-like appearance, even though its use is not required. Four main services are provided by this server: First of all, it controls crucial host-level information like the hostname, hostid, and system version that the Mach kernel does not handle. Second, preserving sessions and process groups, makes POSIX functionalities easier. Thirdly, by giving each task a distinct process ID (PID), it creates a direct mapping between Mach tasks and Hurd processes. 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. (Perera 2024)

3.1.2 Setting Up a Working GNU/Hurd System

A functional GNU/Hurd system can be established through various approaches, with the most accessible and widely recommended method involving the use of a prebuilt disk image. This method significantly reduces setup complexity and enables quick access to a working system, making it especially suitable for experimentation and development.

Using a Prebuilt Image

Among the available options, the prebuilt disk image approach offers the most straightforward experience. These images are prepared in advance and designed to work seamlessly within virtual machine environments like QEMU. By eliminating the need for manual configuration, this method minimizes the technical barriers typically associated with installing Hurd. The Debian GNU/Hurd project maintains these images and provides clear guidance on their use. For the i386 architecture, a stable version is available, while a 64-bit version exists in pre-release form. Once the image is obtained and run in a virtual environment, users are presented with a ready-to-use Hurd system, making it possible to explore its features with minimal setup effort (Debian Project 2025).

Alternative Methods

In contrast, other installation methods, such as using the Debian Installer or tools like crosshurd, require a more involved setup process. The Debian Installer method involves creating installation media from ISO images and proceeding through a traditional install routine, including manual partitioning and system configuration. Similarly, the crosshurd method sets up the system over the internet and typically involves multiple steps like system compilation and network setup. These alternatives, while flexible and more customizable, are generally more time-consuming and error-prone, making them less suitable for newcomers or quick evaluations.

Overall, the prebuilt image method remains the most efficient and reliable way to get started with GNU/Hurd. Its simplicity and ease of use, especially when run in a virtualized environment, allow users to bypass many common setup challenges and begin exploring the system with confidence.

3.2 Containerization Within GNU/Linux

Linux kernel containerization relies on namespaces, cgroups, chroot, and system calls like unshare, pivot_root, and mount. These features isolate processes, manage resources,

and create container-like environments, forming the foundation for tools like Docker and LXC.

- Namespace:

Isolate process views using `unshare` and `clone` system calls. Code in `kernel/nsproxy.c` and `include/linux/nsproxy.h` manages namespace creation (Torvalds 2025), (The Linux Kernel Community 2025b).

- PID Namespace: Isolates process IDs, ensuring processes in different namespaces have independent PID spaces. The `init` process in a PID namespace has PID 1.
- Network Namespace: Isolates network stacks, allowing separate network interfaces, IP addresses, and routing tables.
- Mount Namespace: Isolates filesystem mount points, enabling different views of the filesystem hierarchy.
- User Namespace: Isolates user and group IDs, allowing a process to be root inside a namespace without root privileges on the host.
- UTS Namespace: Isolates hostname and domain name, allowing containers to have unique identifiers.
- IPC Namespace: Isolates System V IPC objects and POSIX message queues.
- Time Namespace: Isolates system clocks, introduced in Linux 5.6 (2020).
- Cgroup Namespace: Isolates cgroup hierarchies, ensuring processes see only their cgroup subtree.

- Cgroups: (The Linux Kernel Community 2025a)

Control resource allocation (e.g., CPU, memory) via `cgroupfs`. Code in `kernel/cgroup/` handles cgroup operations (Torvalds 2025).

- usage: Processes are added to cgroups by writing their PIDs to `/sys/fs/cgroup/<controller>/tasks`.

– Example: Limit CPU usage.

- Chroot and Filesystem Isolation:

chroot and pivot_root system calls (code in fs/namespace.c) isolate filesystem views (Torvalds 2025).

(Hallyn 2013)

Table 3.1: Key Containerization Features in Linux Kernel

Feature	Mechanism	System Calls
Process Isolation	Namespaces	clone, unshare, setns
Resource Control	Cgroups	(via cgroupfs)
Filesystem Isolation	Chroot, Pivot_root	chroot, pivot_root, mount
Security	Seccomp, Capabilities	seccomp, capset

Listing 3.1: Create a new process namespace

```
1 #!/bin/bash
2 sudo unshare --fork --pid --mount-proc /bin/bash
3 # this will hide all other processes in the system from
4 # the processes inside the namespace
```

(Kerrisk 2025)

3.3 Containerization Within GNU/Hurd

After having a thorough research (chapter 2), we can state confidently GNU/Hurd lacks native containerization like Linux, but its microkernel design allows for isolation via tasks and translators, where each task operates in its own address space with dedicated resources. (Loepere 1992). Implementing container-like behavior, including unshare and jail-like features, requires a careful design, with no standardized tools currently available. GNU/Hurd uses the Mach microkernel, where system services are user-space servers called translators, providing inherent isolation for tasks and resources (gnu.org/ 2003b), which offers potential for container-like environments.

Table 3.2: Comparison of Containerization Features

Feature	Linux	GNU/Hurd
Native Container Support	Yes (namespaces, cgroups)	No
Unshare-like Behavior	Yes (unshare syscall)	No
Performance	High	Lower
Hardware Support	Broad	Limited

3.3.1 Current Features

This section highlights some fetures GNU/Hurd provides currently that complements container-like environment. While it's important to note that these features are not namespace equivalent, Linux uses these type of functionality to implement full containerization.

Hurd Jails

Creating a task with restricted filesystem translator (ext2fs) and limited device access can approximate a jail, similar to FreeBSD's jails. This involves setting up a new disk image & setting up ext2fs translator to make the disk image accessible as a filesystem in hurd. Then limit its interaction with the system, enforcing resource boundaries manually as shown in listing 3.2 with the use of chroot (GNU Project 2016a).

Listing 3.2: Jail-like environment (ibid.)

```

1      #!/bin/bash
2      mkdir container
3      dd if=/dev/zero of=container.img bs=1M count=100
4      mke2fs container.img
5      settrans -a container /hurd/ext2fs --writable container.img
6      mkdir -p container/{bin,lib}
7      cp /usr/bin/bash container/bin/
8      cp /lib/i386-gnu/libtinfo.so.6 container/lib/
9      cp /lib/i386-gnu/libc.so.0.3 container/lib/
10     cp /lib/ld.so container/lib/
11     cp /lib/i386-gnu/libmachuser.so.1 container/lib/
12     cp /lib/i386-gnu/libhurduser.so.0.3 container/lib/
13     chroot container /bin/bash

```

SubHurd

GNU/Hurd's inherent support of isolation paved the way to build subhurd, a concept that is experimental and community-driven, with no standardized tools available. Further can be refers to running a subset of Hurd services or a lightweight Hurd instance within a larger Hurd system, leveraging the microkernel's ability to create nested or sandboxed environments. Subhurd primarily have introduced to debugging core servers (proc server, etc) so that developers can attach to system servers using GDB. This happens specially because, no one can start a translator without bootstrap process that runs at the begining of the hurd boot process (gnu.org/ 2003a). So, GDB into system translators cannot be done directly and it returns translator didn't run using bootstrap message to avoid deadlocks (gnu.org/ 2017).

While this looks like a container environment inside Hurd, primary focus of subhurd which is debugging the hurd servers have overlooked lightweightness & performace. That solves the concern why subhurd have replicated many servers including proc server by running boot process (bootstrap) equivalent as shown in boot command in figure 3.3. Therefore, when a process inside a subhurd requests a service from proc server, main proc server will definetely receive this request but without handling the request by itself, proc server queries the proc servers inside each deployed subhurd and forward the request to the relevant proc server. While this is a good solution to debug system servers by replicating them, subhurd shouldn't consider as a container equivalent since primary and important reason of containers was not considered when designing subhurds which is lightweightness & performace. Even though GNU/Hurd has lesser performance compared to Linux as a microkernel based OS, When designing something performace critical, lightweightness should be a major concern of the design. This design error in-terms of lightweightness inside subhurd lead to the proposed design in section 3.5.

Listing 3.3: Creating a subhurd environment (ibid.)

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

```

4 sudo settrans -a /mnt /hurd/ext2fs /dev/hd1s2
5 sudo boot --kernel-command-line="fastboot_root=pseudo-root" /dev/
  hd1s2

```

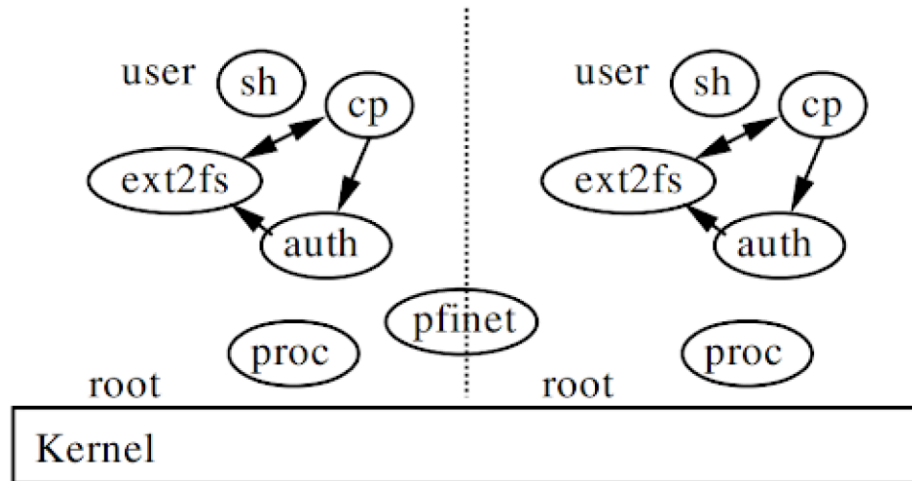


Figure 3.1: Subhurd Architecture

3.4 Initial Design - Handle Namespaces in the Kernel

GNU/Mach uses a port mechanism to IPC, this approach is designed to alter this port mechanism to spread namespace awareness into the other servers by combining a byte-size message into IPC. But, to do so, the kernel should maintain & manage namespaces like the Linux kernel to some extent. As we can see in figure 3.2, the namespace manager in Kernel has the responsibility to create namespaces and bind resources(processes, file systems, mount points, network interfaces) to the created namespace. Each created namespace will have a unique number (eg: 00000001). Any process IPC will go through this namespace manager to append the unique number of the respective namespace at the end of the IPC message so that any receiving end server can check the last byte to retrieve the namespace of the associated process quickly. And any restriction can be enforced at the kernel level. For example, if a process in namespace 'ns1' wants a port sent right to a port associated with a process in namespace 'ns2', the kernel can deny it as a security of namespace isolation mechanism. Since this research mainly focused on

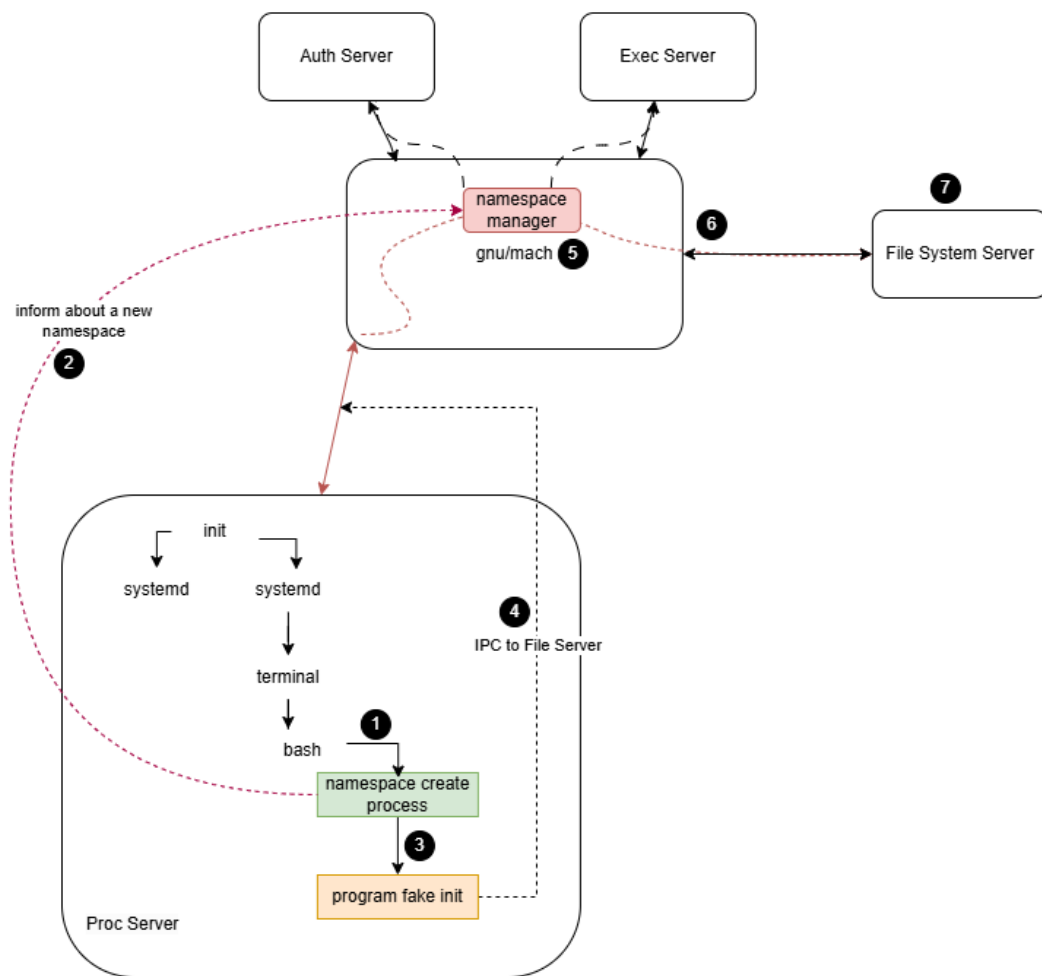


Figure 3.2: Design Approach 1

process isolation, the design here in figure 3.2 shows an example usage in a proc server. This approach has several main steps from 1 to 7 as you can see in figure 3.2:

1. user issues the command to initiate process isolation along with the initial process(file) to be executed. (same kind as the ‘unshare‘ command in Unix).
2. above command will fork & create the given initial process & create a namespace inside mach and bind the given process to that namespace.
 - This binding is done only considering process isolation. Therefore any process in any namespace still could see the entire file system.
3. after forking, the given initial process is a child process of the namespace create

process.

- Now created child process is in a namespace and it can only see & work with processes inside the same namespace. However, other hurd system servers are also processes. Every process irrespective of its namespace should be able to reach those servers to call operating system functionalities. this is one point microkernel design defers from monolithic kernel design. Therefore by default, every namespace (process group) has all the hurd system server processes.

4. If the initial process wants to access a file in the file system server, It sends an IPC through a port associated with the file system server.

- First this process has to request a send port right from the Mach kernel. mach kernel (namespace manager) permits it because the file system server is in the same namespace by default as mentioned.

5. Mach kernel (namespace manager) altering the IPC sent by the initial process above to append the ending byte indicating the namespace that process belongs to.

6. namespace manager forward altered IPC to the file system server.

7. file system server can give the requested file according to the defined file system isolation based on the namespace that the process belongs to.

- File system isolation can be defined & implemented here. For example, if the retrieved namespace is associated with a mounted file system (file system isolation), the file system server only give permission to files in that associated file system not to the entire file system.

Therefore, by studying the above steps in figure 3.2, we can clearly see mach kernel managing namespaces but awareness is delegated to other Hurd operating system servers

3.5 Second Design Approach - Handle Namespaces in the Proc Server

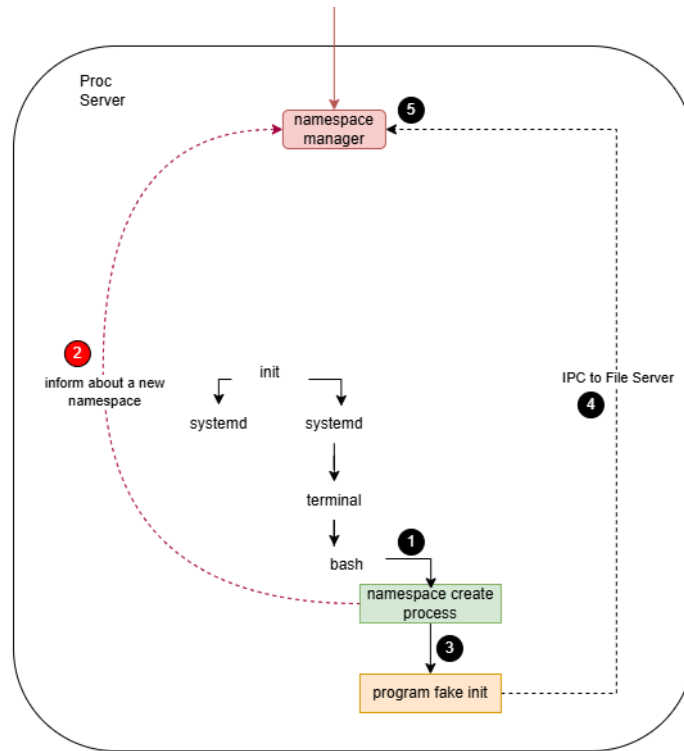


Figure 3.3: Design Approach 2

As shown in figure 3.3, the namespace awareness applied to the proc server which is fundamentally responsible for process related policies. This is solely done not only to reduce the kernel complexity but also to consider the container characteristics. Because containerization is primarily separating a group of processes from others in a system (a responsibility of the proc server). All the other namespace isolation can be achieved by delegating proc server namespace awareness into respective system server. The main steps in the second design approach have changed like this:

1. user issues namespace create command ('unshare')
2. namespace create command creates the given process by forking. Then creates a new namespace & assigns the forked process to it.

- Here kernel is not aware of namespaces, proc server creates processes as

usual using mach kernel. But, inside the proc server, there is an abstraction layer that handles namespace awareness. proc server (namespace manager) creates namespaces and process groups.

3. forked new process is a child process of namespace create process.
4. when forked child process needs access to a specific file in the file system, it sends IPC to file server through namespace manager.
 - namespace manager will filter every process IPC based on its namespace so that a process in a namespace is not aware of other processes not in the same namespace.
5. namespace manager alters the IPC to have a new property called namespace in RPC sent from the proc server.
 - Proc server will send namespace information to other servers through RPC.

By studying above design, we can see that kernel is not aware of namespaces and therefore micro-kernel complexity is minimal as before. But, namespace awareness is still delegate to the other system servers through IPC mechanism.

3.6 Analyse Isolation

3.7 Inherent Isolation Within GNU/Mach

- Tasks and Threads: Mach organizes execution into tasks (which encapsulate virtual address spaces) and threads (which are schedulable units within tasks). This separation ensures that each task operates in its own protected memory space (gnu.org 2008-11-13b).
- Ports and Capabilities: Inter-process communication (IPC) in Mach is facilitated through ports, which are protected message queues. Access to these ports is

controlled via capabilities, ensuring that only authorized tasks can communicate or access specific resources (gnu.org 2008-11-13a).

- Message-Oriented IPC: All interactions with kernel services and other tasks occur through message passing, reinforcing isolation by preventing direct access to another task's memory or resources (ibid.).

In monolithic architectures, a significant drawback lies in the integration of file system logic within the kernel. This ensures that if a kernel crash happened 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. Therefore employing a microkernel architecture, such as Mach, can enhance the isolation of containers by ensuring that a failure in one container does not affect the kernel or other containers. While microkernels inherently offer extensive isolation across their systems, our examination will concentrate solely on process namespace isolation.

Chapter 4

Implementation

4.1 Environment Setup

Setting up the environment for developing and testing process namespaces in GNU/Hurd is a critical step that lays the foundation for this research. This process is inherently complex due to several factors:

- **Interdependent Components:** GNU Mach, GNU Hurd, GNU MIG, and GNU C Library are tightly coupled. Building or modifying one often necessitates rebuilding the others to maintain compatibility.
- **Limited Documentation:** The lack of up-to-date and comprehensive documentation can lead to a trial-and-error approach, consuming significant time and resources.
- **Cross-Compilation Requirements:** Given that GNU/Hurd primarily targets the i386 architecture, setting up a cross-compilation environment on modern systems adds another layer of complexity.
- **Sparse Community Support:** GNU/Hurd has fewer community resources and forums for troubleshooting, making problem-solving more challenging.

Components Overview and used versions:

- GNU Mach: The microkernel responsible for low-level tasks such as memory management and inter-process communication. (v1.8+git20230526)
- GNU Hurd: A collection of servers running on top of GNU Mach that implement higher-level operating system functionalities. (v0.9.git20230520)
- GNU MIG (Mach Interface Generator): A tool that generates code to facilitate communication between the microkernel and user-space servers. (v1.8+git20230520)
- GNU Libc: The C standard library, which in the context of GNU/Hurd, includes specific adaptations to interface correctly with the Hurd servers. (v2.36)

4.1.1 Run the Disk Image using Qemu

As we discussed in Design phase (section 3.1.2), we can obtain an GNU/Hurd pre-built image easily without doing the hassle of cross compiling or installing from scratch. Even though we can use direct qemu command to run the image, it's better to use a script that we can handle parameters ourselves easily as shown in appendix.

This script allows to change the hurd image, set the ssh port, ram storage easily. for example: `./runHurd hurd-deb.img 2222 4` command will run the hurd image named `hurd-deb.img` opening port 2222 to SSH with 4GB of ram storage.

4.1.2 Expanding the Disk Image Size

As development progresses, you may find the default disk image size insufficient. Here is an example shell commands to increase the size of hurd image named `ready14-deb-7-8G.img` by 4GB (Debian GNU/Hurd Maintainers 2023).

Listing 4.1: Resize the GNU/Hurd Disk Image

```
1 #!/bin/bash
2 qemu-img resize -f raw ready14-deb-7-8G.img +4G
3 fdisk -l ready14-deb-7-8G.img # to see where partition 2 starts
```

After execution of the above command, look for the **start sector** of the **second partition**.

Listing 4.2: modify the partition layout

```
1 parted ready14-deb-7-8G.img
2 # Inside parted
3 resizepart 2 100% # Expand partition 2 to use all available space
4 quit
```

Listing 4.3: attach the loop device & resize the filesystem

```
1 # attach the loop device
2 sudo losetup -o $((512*<start_sector>)) /dev/loop0 ready14-deb-7-8G.
   img
3 sudo e2fsck -f /dev/loop0 # require before resize2fs
4 sudo resize2fs /dev/loop0 # resize the filesystem
5 sudo losetup -d /dev/loop0 # Detach the Loop Device
```

Replace **<start_sector>** with the partition's start sector

Listing 4.4: verify changes

```
1 # boot into hurd
2 df -h
```

4.1.3 Compilation of GNU/Mach

as we disussed in design chapter, to implement namespace awareness inside the GNU/Mach directly, we have to compile and install the GNU/Mach microkernel from the sources.

Listing 4.5: verify changes

```
1 #!/bin/bash
2 # GNU Mach
3 mkdir gnumach && cd gnumach
4 chown _apt:root .
5 apt-get source gnumach
```

```

6  cd gnumach-*/
7  sudo apt-get build-dep gnumach
8  sudo apt install debhelper quilt
9  dh_quilt_patch
10 # can employ new changes into the source code
11 dpkg-buildpackage -us -uc -b -rfakeroot
12 sudo dpkg -i \
13 ../gnumach-image-1.8-486_1.8+git20230526-2_hurd-i386.deb \
14 ../gnumach-dev_1.8+git20230526-2_hurd-i386.deb \
15 ../gnumach-common_1.8+git20230526-2_hurd-i386.deb
16 sudo update-grub
17 sudo reboot

```

4.1.4 Compilation of GNU/Hurd

Compiling and installing GNU/Hurd system is crucial in this research since, to implement each new designs proposed, we have to change the hurd system servers.

Listing 4.6: verify changes

```

1  mkdir hurd
2  chown _apt:root hurd
3  apt-get source hurd
4  cd hurd-*/
5  # install all dependancies for hurd
6  sudo apt-get build-dep hurd
7  # apply deb patches
8  sudo apt install debhelper quilt
9  dh_quilt_patch
10 # build deb packages
11 dpkg-buildpackage -us -uc -nc -b -rfakeroot
12 # install new hurd deb packages
13 dpkg -i ../hurd_*.deb ../hurd-libs0.3_*.deb ../hurd-dev_*.deb
14 reboot

```

4.1.5 Compilation of GNU/Mig

To compile GNU/Mach and GNU/Hurd we need the support of GNU/Mig which is used to generate RPC stubs and skeletons to facilitate communication between the microkernel and user-space servers.

Listing 4.7: verify changes

```
1 #!/bin/bash
2 sudo apt source mig
3 sudo apt build-dep mig # get all dependancies
4 cd mig-1.8+git...
5 dpkg-buildpackage -us -uc -b -rfakeroot
6 cd ../
7 dpkg -i mig_1.8+git...
8 mig -a # verify
```

4.1.6 Compilation of GNU/Glibc

Compilation of Glibc library will be crucial when we have implemented new calls into hurd servers or microkernel so that it provides the general interface to user level programs to use those internal calls, acting as a standard wrapper as shown in figure 4.1.

Listing 4.8: verify changes

```
1 #!/bin/bash
2 # Cause: CET (Control-flow Enforcement Technology) is not supported
   on Hurd.
3 ../glibc-2.36/configure --prefix= --disable-werror --disable-cet
4 make
5 make install
```

4.1.7 Challenges and Troubleshooting

Building container-like functionality in GNU/Hurd, especially with a focus on enhancing namespace awareness and leveraging concepts like subhurds, involves several challenges.

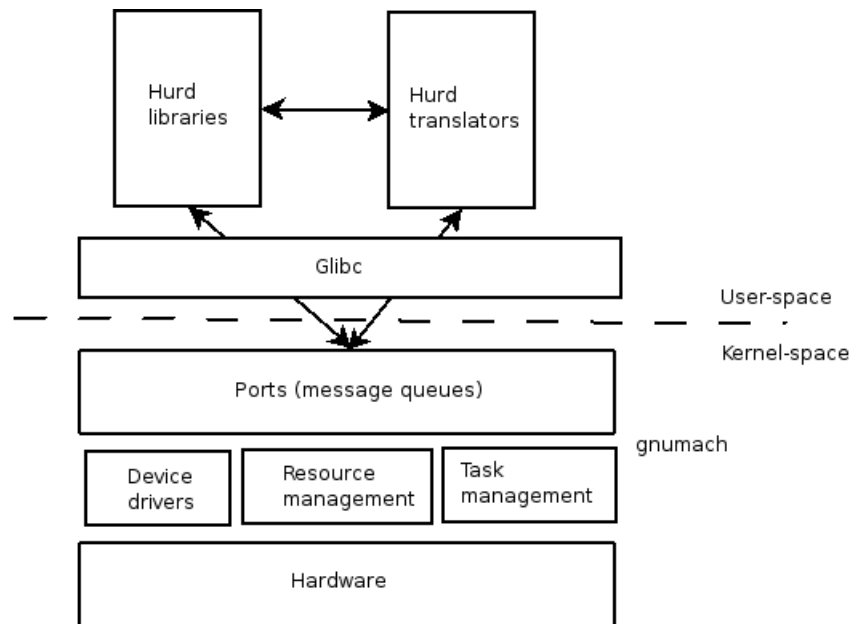


Figure 4.1: Hurd Architecture (R. Espinola 2009)

These stem from the system's unique microkernel architecture, limited community support, and the experimental nature of Hurd. These are the challenges, and troubleshooting strategies, identified after critically examining the development process for a more comprehensive understanding.

Compiling using original gnu source code & version conflicts

Compiling GNU/Hurd components like GNU Mach, the Hurd servers, or the proc server often requires using the original source code from the GNU repositories. However, version mismatches between components—such as GNU Mach, Hurd, and the GNU C Library can lead to compilation failures or runtime errors. For instance, hurd OS freezes at "start acpi:" after modifying mach_msg_header_t highlights how a change in one component (Mach) can break compatibility with others (Hurd servers).

Troubleshooting:

- **Version Pinning:** Ensure all components are from compatible versions. Use the exact commit hashes or tags and verify compatibility with the glibc version of the Debian GNU/Hurd environment. Check the Debian GNU/Hurd port page for recommended versions.

- **Dependency Tracking:** Use `dpkg -l | grep hurd` and `dpkg -l | grep gnumach` to list installed versions, ensuring alignment.
- **Build Isolation:** Compile in a clean environment to avoid interference from system-wide libraries or mismatched headers.
- **Using Debian repositories:** As in subsection 4.1.3 and following other subsections about compiling sources, it's very convenient to use debian repositories directly as they have some patches which will help to stabilize the build.

Installing glibc library

The GNU C Library (glibc) is a critical dependency for GNU/Hurd, providing essential system calls and runtime support. Installing or upgrading glibc can be challenging because Hurd relies on a specific (older) ported, version of glibc. Upgrading glibc to a newer version may break Hurd's servers or introduce bugs due to differences in system call implementations or Mach-specific extensions. For example, creating and defining a new RPC will involve update glibc executable on the new RPC. Otherwise other processes cannot use the defined new RPC. An incompatible glibc could cause segmentation faults or unexpected behavior.

Troubleshooting:

- **Use Hurd-Compatible glibc:** Stick to the glibc version provided by Debian GNU/Hurd. Avoid manual upgrades unless explicitly supported by the Hurd team.
- **Copy binaries directly:** Try to use specific library binaries by copying them into the standard library directory if that updation is enough for the implementation without trying to compile entire glibc library.

Potential Data Loss Due to System Crashes

Developing on GNU/Hurd, carries the risk of system crashes. Custom code changes can be lost if not properly backed up. Additionally, Hurd's limited stability means crashes during development are more frequent, especially when experimenting with kernel changes.

Troubleshooting:

- **Version Control:** Use Git to track all changes.
- **Frequent Backups:** Before rebuilding or rebooting, back up modified source directories. (using scp like commands or even image level backups)
- **Incremental Testing:** Apply changes incrementally and test after each step to isolate crash causes.

4.1.8 Reproducibility

Reproducibility is a cornerstone of reliable system development, ensuring that builds can be consistently replicated. The following measures were undertaken to achieve reproducible builds:

Controlled Build Environment

- **Host System:** All builds were performed on a Debian 12 host, providing a stable and consistent environment.
- **Toolchain Consistency:** Specific versions of compilers, linkers, and related tools were used, and their versions documented, to prevent discrepancies arising from toolchain variations.

Verification

- **Repeat Builds:** Multiple builds were performed in clean environments to confirm that the process yields identical results, reinforcing the reproducibility of the setup.

By adhering to these practices, the development process for the GNU/Hurd environment achieves a high degree of reproducibility.

4.2 Implementation of First Design Approach

First design approach tries to implement namespace awareness within the kernel and delegate this awareness into the other part of the OS (system servers) eventually as explained in section 3.4. While process/task isolation (preventing one process from accessing or interfering with another) is a mechanism of an operating system, Process Namespace Isolation (process abstraction of isolating one or a group of processes from others) is a policy of an operating system. For example, while process abstraction is completely unknown to the mach kernel, proc server tracks & manage process metadata and relations and provides the interface that userland tools use. The kernel even doesn't know what a pid is.

/GNU/Hurd as a microkernel based operating system emphasizes minimality and clear division between mechanisms(kernel level) and policies(user level). Therefore trying to implement namespace awareness within the kernel is not the correct design principle (Levin et al. 1975).

"A concept is tolerated inside the microkernel only if moving it outside the kernel would prevent the implementation of the system's required functionality." (Liedtke 1995)

But in the other hand, implementing the namespace awareness within the kernel might increase the performance since less IPCs are involved. If the namespace awareness solely inside the kernel, still users are interacting with the proc server to perform process level operations. That means users have to use system servers to perform their relevant operations. Therefore there need to be a mechanism to spread namespace awareness into the system servers. That will definitely involve IPC calls again. Therefore the performance advantage can be expected from using less IPCs in this design will not have a significant difference.

Further more, as discussed above, this design have to delegate namespace awareness

from kernel to the other system servers. That process is very complex and error prone since changing kernel affects all the other part of the hurd system overall. Therefore without implementing namespace awareness into the other servers and services of the hurd system first, kernel modifications cannot apply. Otherwise it causes the system to fail at the booting as provided in the figure 4.2, when trying to change a struct of the kernel which is `mach_msg_header_t`. Since this research only focuses on the process namespace awareness, dealing with all the system servers and implmenting all the other namespaces functionalities are out of scope and time frame not allows it.

```

done
scsi : 0 hosts.
scsi : detected total.
Partition check (DOS partitions):
  hd0: hd0s1 hd0s2
com0: at atbus0, port = 3f8x, spl = 6u, pic = 4. (DOS COM1)
com 2 out of range
lpr0: at atbus1, port = 378x, spl = 6d, pic = 7.
RTC time is 2025-04-08 14:59:19
module 0: acpi --host-priv-port=${host-port} --device-master-port=${device-port}
  --next-task=${pci-task} $(acpi-task=task-create) $(task-resume)
module 1: pci-arbiter --next-task=${disk-task} $(pci-task=task-create)
module 2: rumpdisk --next-task=${fs-task} $(disk-task=task-create)
module 3: ext2fs --readonly --multiboot-command-line=${kernel-command-line} --ex
ec-server-task=${exec-task} -T typed ${root} $(fs-task=task-create)
module 4: exec /hurd/exec $(exec-task=task-create)
5 multiboot modules
task loaded: acpi --host-priv-port=1 --device-master-port=2 --next-task=3
task loaded: pci-arbiter --next-task=1
task loaded: rumpdisk --next-task=1
task loaded: ext2fs --readonly --multiboot-command-line=root=part:2:device:hd0 -
-exec-server-task=1 -T typed part:2:device:hd0
task loaded: exec /hurd/exec

start acpi:

```

Figure 4.2: /GNU/Hurd system failure during boot due to simple kernel changes

Because of the incorrectness of the design and mentioned issues above, this approach of designing namespaces inside the kernel concluded as incorrect. Table 4.1 summarizes the comparison between implementing namespace awareness inside the kernel and outside the kernel.

4.3 Implementation of Second Design Approach

Second design approach tries to implement namespace awareness mainly inside proc server and other system servers which are related to each type of namespace isolation without employing kernel modifications unless it's required as discussed in section 3.5. As pointed out in the section 4.2, namespace isolation is a policy and implementing

Table 4.1: Comparison: Namespace Design inside Kernel Space vs User Space

Aspect	Kernel Implementation	OS Server (User-space) Implementation
Speed (Performance)	Faster – direct memory access, fewer context switches.	Slower – due to IPC overhead, context switches, and possibly extra data copying.
Flexibility	Harder to change or update.	More modular and flexible. Can be replaced or updated without touching the kernel.
Security/Stability	Bug in kernel equals to system crash or exploit.	Crashing user-space server doesn't bring down the whole system.
Philosophical Alignment (with Hurd)	Violates Hurd's microkernel design philosophy.	Matches Hurd's vision: policy outside, mechanism inside.

such policies outside the kernel is a great design principle which emphasizes minimality design principle in microkernel based operating systems (Levin et al. 1975). Furthermore, end to end argument in networks suggests that certain functions are best implemented at the application level, where the full context is available, rather than at lower levels of the system.

"The function in question can completely and correctly be implemented only with the knowledge and help of the application standing at the endpoints of the communication system. Therefore, providing that questioned function as a feature of the communication system itself is not possible." (Saltzer, Reed, and Clark 1984)

In the context of microkernel architectures, such as GNU/Hurd, this argument supports the design choice of implementing only essential mechanisms within the kernel, while delegating policies to user-space servers. For example, the Mach microkernel provides fundamental mechanisms like inter-process communication and memory management, but higher-level abstractions like process management are handled by user-space servers like `proc`. Therefore the separation of namespace awareness from the kernel level in this design approach, perfectly aligns with the End-to-End Argument by ensuring that policy decisions, which may vary between applications, are implemented where the necessary context and flexibility exist (outside the kernel).

4.3.1 Namespace Implementation

As we discussed in section 3.2, Linux uses CLONE system call underneath the unshare command to create new namespaces. Since this research mainly focuses on process namespaces, to create such process namespace using the help of proc server, we need to define a function inside proc server, so that user processes can call that function to tell proc server to create a new namespace using the caller process (process called that function) as the first process of that namespace. But calling such function inside proc server is not straight forward as in linux since in GNU/Hurd, proc server is a separate process. Since it's a system server, using GNU/Mach ports capabilities and GNU/Mig defined RPC stubs, user programs can call exposed functions of the proc server. Source files related to the proc server can be found in a directory called proc inside the hurd source code.

Before defining the new RPC, new struct was created as showed in listing 4.9 inside proc/proc.h to represent a namespace inside the proc server so that applying improvements & new features could be easier. Even though namespace inheritance is not a concern in this research scope, a field to keep track of the parent namespace also can be added allowing future improvements.

```
1  struct namespace {
2      pid_t ns_id;           // Unique ID for the namespace
3      struct proc *root_proc; // First process in the namespace
4      struct proc *proc_list; // List of processes in this
                               namespace
5      // struct namespace *parent_ns; // Parent namespace, NULL for
                               root namespace
6  };
```

Listing 4.9: Namespace struct

Namespace Create Functionality

There is a function that creates the init process proc structure namely create_init_proc in proc/mgt.c. To prevent unintentional conflicts, a line was added to set init proc namespace

as a NULL since, host system processes considered doesn't have a namespace. hence NULL. Then, a new function to create a namespace was defined in proc/mgt.c as shown in listing 4.10.

```
1 error_t
2 S_proc_create_namespace(struct proc *callerp, task_t task) {
3     struct proc *shadow;
4     if (! callerp)
5         return EOPNOTSUPP;
6     shadow = task_find (task);
7     if (shadow)
8     {
9         /* found the fake_init process */
10        // create a namespace
11        struct namespace *ns = allocate_namespace (shadow);
12        if(!ns)
13            return EOPNOTSUPP;
14        else
15            shadow->p_namespace = ns;
16    }
17    else {
18        // task not found
19        return EOPNOTSUPP;
20    }
21    return 0;
22 }
23 struct namespace *
24 allocate_namespace (struct proc *fake_init)
25 {
26     struct namespace *ns;
27     /* all other processes inherit from fake_init here. */
28     size_t size = sizeof (struct namespace);
29     ns = malloc (size);
30     if (!ns)
31         return NULL;
32     memset (ns, 0, sizeof(&ns));
```

```

33 // assign pid of the fake init for temporary
34 ns->ns_id = ((fake_init)->p_pid);
35 ns->root_proc = fake_init;
36 ns->proc_list = NULL;
37 ns->parent_ns = NULL;
38 return ns;
39 }

```

Listing 4.10: Function to create a namespace

Namespace inherit Functionality

Even though, now proc server allows to create a namespace, a logic to inherit namespace from parent process to child process is needed. When created a new task, mach kernel notifies the proc server about the newly added task to handle process level abstractions. `S_mach_notify_new_task` function inside `proc/mgt.c` is responsible to handle such notifies from mach kernel. Therefore the namespace inherent logic has implemented as shown in listing 4.11 inside the `S_mach_notify_new_task` function.

```

1  /** S_mach_notify_new_task function */
2  // task is the new task created by mach kernel
3  childp = task_find_nocreate (task);
4  if (! childp)
5  {
6      mach_port_mod_refs (mach_task_self (), task,
7                          MACH_PORT_RIGHT_SEND, +1);
8      childp = new_proc (task);
9      /* by charith pietersz */
10     inherit_namespace(parentp, childp);
11     /* end charith pietersz */
12 }
13 /** end of S_mach_notify_new_task function */
14 /* Ensure child process inherits namespace from parent */
15 void inherit_namespace(struct proc *parent, struct proc *child)
16 {

```

```

16     if (parent->p_namespace)
17     {
18         // allocated a namespace
19         child->p_namespace = parent->p_namespace;
20     }
21     else
22     {
23         // allocated a NULL namespace
24         child->p_namespace = NULL; // Root namespace
25     }
26 }

```

Listing 4.11: Inherit namespaces

Define a New RPC

Still the defined function `S_proc_create_namespace` is just a function. Proc server doesn't expose this function to others to call. This is where GNU/Mig plays a crucial role by letting define new RPCs using specific standard method. All the process related RPC routines were defined in `hurdf/process.defs`. Newly defined RPC reply and request routines were added in the exact order of these routines were defined in `process.defs`, `hurdf/process_reply.defs` and `hurdf/process_request.defs` accordingly. Therefore new routines to the `S_proc_create_namespace` function inside proc server were defined in those three files respecting the order as shown in listing 4.12.

```

1     // hurdf/process.defs
2     routine proc_create_namespace (
3         process: process_t;
4         task: task_t);
5     // hurdf/process_request.defs
6     simpleroutine proc_create_namespace_request (
7         process: process_t;
8         ureplyport reply: reply_port_t;
9         task: task_t);
10    // hurdf/process_reply.defs

```



```
11 skip; /* proc_create_namespace */
```

Listing 4.12: Define RPC routines

During compilation, GNU/Mig generates client & server functions related to these routines defined in .defs files and at the end of the compilation, process.h header file is created containing function declarations to the RPCs defined in process.defs. After the successful installation of the hurd source, this process.h header file will move into the /usr/include/hurd/ directory.

But still after successful updation of these header files about the new function proc_create_namespace, function calls from user processes fails at run time because, libc.so.0.3 library which provides the POSIX like interface to the user space, doesn't expose function call to the new RPC defined. This is where we have to compile the GNU/Glibc. During compilation of the GNU/libc, it notices changes in hurd libraries, and uses new header files like process.h to create new POSIX like interface so that user space programs can call the new RPC function. After the successful updation of libc.so.0.3 library, we can invoke proc_create_namespace function within user space programs.

4.3.2 Enforce process Isolation

Unshare Client

With the given namespace awareness into the proc server, user programs can use proc_create_namespace function by including process.h header file into the code base. There should be a client program that uses the new RPC to create process namespaces. Just like unshare command in linux. To experience namespace creation as same as linux, client program named as unshare.

Basically, unshare client perform these tasks:

- fork and create a new child
- new child is expected to be the fake init function inside the process namespace.

- child process calls to `proc_create_namespace` function
- after successful creation of process namespace, execute `exec` to replace child process with the `bin/bash` providing an interactive shell to the users.

```
1  #include <sys/types.h>
2  #include <unistd.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <sys/wait.h>
6  #include <hurd.h>
7  #include <hurd/process.h>
8  #include <mach/mach.h>
9  #include <error.h>
10
11 mach_port_t get_procserver() {
12     return getproc(); // Returns the 'procserver' port for the
13                        // calling process
14 }
15
16 int main()
17 {
18     int id = fork();
19     if(id == 0)
20     {
21         pid_t pid = getpid();
22         mach_port_t procserver = get_procserver();
23         if (procserver == MACH_PORT_NULL) {
24             printf("Failed to get procserver\n");
25             exit(1);
26         }
27         task_t task;
28         error_t err;
29         err = proc_pid2task (procserver, pid, &task);
30         if (err)
31         {
```

```

31         error(0, err, "Getting task for the child process
           : %lu\n",pid);
32         exit(1);
33     }
34     // call to create namespace function
35     err = proc_create_namespace(procserver, task);
36     if (err) {
37         printf("RPC call failed: %d\n", err);
38         exit(1);
39     }
40     execl("/bin/bash", "bash", NULL);
41     // if exec failed
42     perror("execl failed");
43     exit(1);
44 }
45 else if(id > 0)
46 {
47     pid_t pid = getpid();
48     int status;
49     // Wait for the child to terminate
50     waitpid(id, &status, 0);
51     exit(0);
52 }
53 else{
54     printf("fork failed");
55     exit(1);
56 }
57 return 0;
58 }

```

Listing 4.13: Unshare client program

Simple Isolation

Current implementation of the process namespace in proc server, only has the namespace awareness, not any isolation enforced using the given namespace awareness. Therefore executing unshare client is not significant & useful. To enforce full isolation, every process related function inside proc server have to use the given process namespace awareness. But, this is extensive work for this research as it is enough to show this method can enforce full isolation. Those, tiny yet vast amount of changes can be modified later as updates to this work. But after reading the proc server source code thoroughly, we found out that there is a functions that is used by many programs like ps, procs to know about other processes which is getallpids function. Therefore enforcing process namespace isolation to this function is the current best & easy option to see the effectiveness of this design. Current getallpids function returns all the process ids in the system using an assembly function called prociterate. Even though there is proc* p (proc struct pointer of the caller process) as a parameter of this function as you can see in listing 4.14, comments have specifically mentioned they don't use it right now. But as of this new namespace awareness, proc* p is a crucial parameter because, using caller p namespace information, we can filter the processes that should be return. That is what exactly done in modified getallpids function shown in listing 4.14.

```
1 kern_return_t
2 S_proc_getallpids (struct proc *p,
3                    pid_t **pids,
4                    mach_msg_type_number_t *pidslen)
5 {
6     if (!p)
7     {
8         return EINVAL;
9     }
10    else
11    {
12        struct ns_filter caller;
13        caller.ns = p->p_namespace;
```

```

14     caller.count = 0;
15     add_tasks(0);
16     /* Count number of processes in the same namespace */
17     prociterate(count_ns_up, &caller);
18     if (caller.count > *pidslen)
19     {
20         *pids = mmap (0, caller.count * sizeof (pid_t),
21                     PROT_READ|PROT_WRITE,MAP_ANON, 0, 0);
22         if (*pids == MAP_FAILED)
23             return ENOMEM;
24     }
25     caller.pids = *pids;
26     prociterate(store_ns_pid, &caller);
27     *pidslen = caller.count;
28     return 0;
29 }

```

Listing 4.14: getallpids function modification

Another important point to mention is, current getallpids function uses two callback functions to feed into prociterate function to get different type of process information. These two callback functions are store_pid (get pids) & count_up (get pid count). In modified version of getallpids function, we had to use new callback functions namely store_ns_pid & count_ns_up that has namespace awareness. These functions returns same namespace process information only. Collectively, with the help of these important callback functions as shown in listing 4.15, modified getallpids correctly returns only the same namespace pids to the caller process.

```

1 struct ns_filter {
2     struct namespace *ns; // Pointer to the namespace (can be NULL)
3     int count;            // Counter for the number of processes
4     pid_t *pids;          // Pointer to store PIDs
5 };
6 static void

```

```

7 count_ns_up (struct proc *p, void *data)
8 {
9     struct ns_filter *caller = (struct ns_filter *)data;
10    if (!caller->ns)
11    {
12        /* If caller is in the main namespace (NULL ns), include all
13         */
14        caller->count++;
15        return;
16    }
17    /* If the process is in the same namespace, filter by namespace
18     */
19    if (p->p_namespace && caller->ns->ns_id == p->p_namespace->ns_id)
20    {
21        caller->count++;
22        return;
23    }
24    // printf("caller_ns:%d, processID:%d\n", caller->ns->ns_id, p->
25    p_pid);
26    // namespace doesn't match
27    return;
28 }
29
30 static void
31 store_ns_pid(struct proc *p, void *data)
32 {
33     struct ns_filter *filter = (struct ns_filter *)data;
34     if (!filter->ns) {
35         /* If caller is in the main namespace (NULL ns), include all
36          */
37         *filter->pids++ = p->p_pid;
38         return;
39     }
40     /* If the process is in the same namespace, store its PID */
41     if (p->p_namespace && filter->ns->ns_id == p->p_namespace->ns_id)
42     {

```

```

37     *filter->pids++ = p->p_pid;
38     return;
39 }
40 // namespaces doesn't match
41 return;
42 }

```

Listing 4.15: Define new callback function for getallpids function

With the enforced isolation into the getallpids function, after successful compilation & installation, we can see simple process namespace isolation by executing unshare client as shown in figure 4.3. As you can see now processes inside a namespace can't access information about processes outside of the same namespace. ps command returns only the namespace processes. In figure 4.3, we can clearly see the behavior of ps command changed in two scenarios (inside and outside namespace).

```

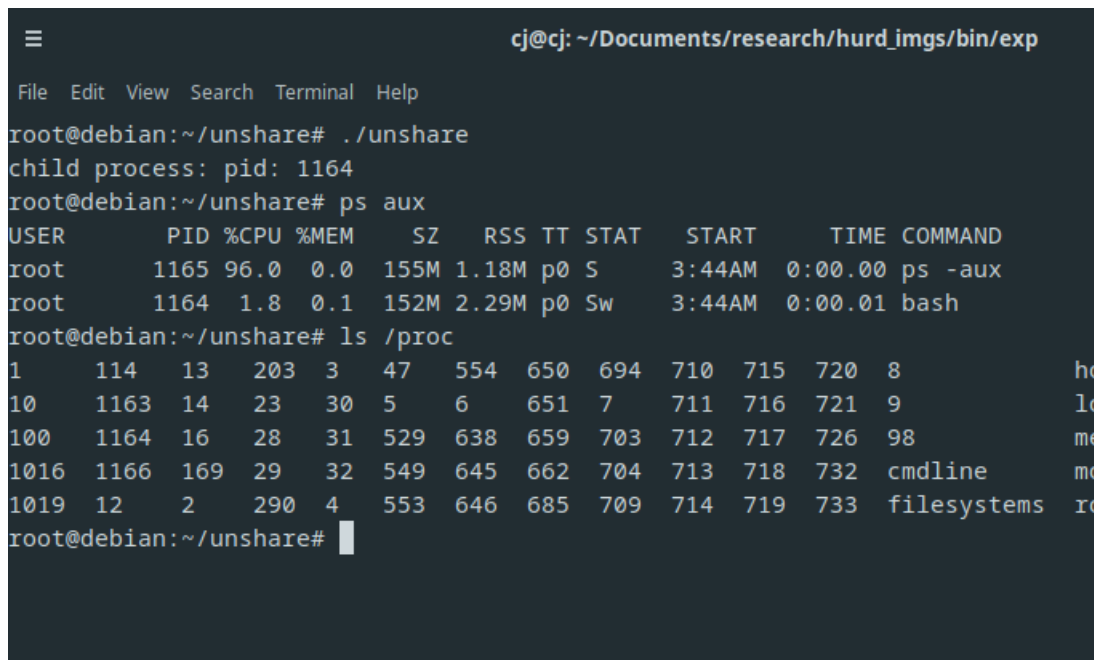
File Edit View Search Terminal Help
root@debian:~/unshare# ./unshare
child process: pid: 1154
root@debian:~/unshare# ps aux
USER      PID %CPU %MEM    SZ   RSS TT  STAT   START     TIME COMMAND
root      1155 43.9  0.0  155M 1.18M p0  S    3:28AM  0:00.02 ps -aux
root      1154 17.8  0.1  152M  2.3M p0  Sw   3:28AM  0:00.01 bash
root@debian:~/unshare#
exit
Child process terminated with status 0
From parent (1153)
root@debian:~/unshare#
root@debian:~/unshare#
root@debian:~/unshare#
root@debian:~/unshare#
root@debian:~/unshare# ps aux
USER      PID %CPU %MEM    SZ   RSS TT  STAT   START     TIME COMMAND
root      1156 120.0 0.0  155M 1.32M p0  S    3:28AM  0:00.00 ps -aux
root         10  0.6  0.0  157M  696K -  S<o  1:00PM  0:00.05 /hurd/auth
root          9  0.3  0.0  141M  1.08M -  S<o  1:00PM  0:01.13 /hurd/exec
root       554  0.1  0.1  217M  3.12M -  S<o  1:00PM  0:02.47 /hurd/netdde
root          8  0.1  0.2  591M  7.28M -  S<o  1:00PM  3:16.97 ext2fs --readonly --multiboot-c
root          2  0.0  0.6  147M 19.5M -  Sp   1:00PM  0:00.03 /hurd/startup root=part:2:device
root          3  0.0  1.8  951M 55.9M ?  D<fmo 1:00PM  0:00.22 gnumach root=part:2:device:hd0
root          4  0.0  0.0  150M  1.05M ?  R<mo  1:00PM  0:12.86 /hurd/proc
root          5  0.0  0.1  154M  1.94M -  S<o  1:00PM  0:00.16 acpi --host-priv-port=1 --devic
root          6  0.0  0.8  155M 25.9M -  S<o  1:00PM  0:00.07 pci-arbiter --next-task=1
root          7  0.0  0.6  147M 19.6M -  S<o  1:00PM  0:00.12 rumpdisk --next-task=1
root       646  0.0  0.1  256M  1.92M -  S<o  1:00PM  0:00.00 /hurd/console
root        12  0.0  0.0  166M  940K -  S<o  1:00PM  0:00.29 /hurd/random --seed-file /var/L
root        13  0.0  0.0  157M  760K -  S<o  1:00PM  0:00.24 /hurd/term /dev/console device
root        14  0.0  0.0  166M  1.08M -  S<o  1:00PM  0:00.11 /hurd/pflocal
root       645  0.0  0.0  173M  1.21M -  S<o  1:00PM  0:00.01 /hurd/term /dev/tty1 hurdio /de
root       529  0.0  0.1  182M  2.34M -  S<o  1:00PM  0:00.87 /hurd/pfinet -6 /servers/socket
root       650  0.0  0.2  201M  6.36M -  S<fo  1:01PM  0:00.18 /bin/console --daemonize -d cu
root       651  0.0  0.0  156M  1.28M -  S<o  1:01PM  0:00.01 /hurd/storpio --no-cache mem

```

Figure 4.3: Simple Isolation

Concern about Procfs

Above mentioned simple isolation isolate the processes inside a namespace from accessing other processes outside of the namespace. But, as shown in figure 4.4, performing `ls` command on `/proc` directory reveals all the process information inside the system to the processes even if we are executing `ls` process inside a namespace. That's why aforementioned isolation referred to as the simple isolation.



```

cj@cj: ~/Documents/research/hurd_imgs/bin/exp
File Edit View Search Terminal Help
root@debian:~/unshare# ./unshare
child process: pid: 1164
root@debian:~/unshare# ps aux
USER      PID %CPU %MEM    SZ   RSS TT STAT   START    TIME COMMAND
root      1165  96.0   0.0  155M  1.18M p0  S    3:44AM   0:00.00 ps -aux
root      1164   1.8   0.1  152M  2.29M p0  Sw   3:44AM   0:00.01 bash
root@debian:~/unshare# ls /proc
1      114   13    203   3     47    554   650   694   710   715   720   8      ho
10     1163  14    23    30    5     6     651   7     711   716   721   9      lo
100    1164  16    28    31    529   638   659   703   712   717   726   98     me
1016   1166  169   29    32    549   645   662   704   713   718   732   cmdline mc
1019   12    2     290   4     553   646   685   709   714   719   733   filesystems ro
root@debian:~/unshare#
```

Figure 4.4: `ls /proc` behavior on simple isolation

This behavior is anticipated. `procfs` translator is mounted on `/proc` directory to provide process related information to the user using special utility program called `settrans`. As mentioned above, even though `procfs` uses `getallpids` function, `getallpids` function uses namespace of the caller process (`procfs` in this case) to filter processes that should be return. It's important to notice that `procfs` process is not a process in the created namespace. Therefore `procfs` process doesn't have namespace (hence `NULL`) and that's why `getallpids` function returns all the available process ids to the `procfs`. `procfs` in-turns return all the process information to the user or `ls` process (caller of the `proc fs`) which breaks the isolation at some point. To show that this design provides process isolation, we atleast need to isolate process visibility. Because, if a process can't name another process, it can't get access to that process.

As a solution to enforce process information isolation, we can't put procfs into the namespace since it will affect to the other users and namespaces. Because, all others will only see the processes inside the namespace of procfs. Even global namespace (NULL) processes will only see processes inside the namespace of the procfs not all processes of the system which is unintentional and incorrect behavior in a concurrent system. we have one of two solutions to enforce better isolation:

- **procfs namespace awareness:** Provide namespace awareness to the procfs translator so that procfs aware of the namespace of the caller process(ls process in ls /proc scenario).
- **Invoke another procfs:** create another procfs translator process in current namespace on different directory that is specific to the current namespace rather than /proc.

procfs namespace awareness

Let's consider ls /proc as an example. To procfs to aware of namespace of the caller (ls) process, procfs need to know task_id or process_id of the caller. In /GNU/Hurd unlike linux, /proc behavior is handled by procfs and the procfs process is mount to /proc at the booting process. We can remove and set the procfs process using specific utility program named settrans as shown in listing 4.16.

Listing 4.16: remount procfs on /proc

```
1 settrans -fg /proc # remove current procfs translator
2 settrans -ca /proc /hurd/procfs --compatible # remount procfs on /
  proc
```

To access pid information of the caller, the function which is responsible to invoke procfs to return process information when executing ls /proc need to be found out. That was accomplished by tracing RPC calls of ls /proc using rpctrace command (rpctrace ls /proc).

Even though RPC trace is over 120 lines, after careful read of the trace, only few

important lines are need to discuss here to get an overall idea. The relevant interaction with /proc starts here:

```
1 7<--40(pid2497)->dir_lookup ("proc" 128 0) = 0 1 "" 52<--25(
    pid2497)
```

- Port 7 is the root directory (/) port for ls (PID 2497).
- dir_lookup ("proc" ...) resolves /proc, returning port 52.

```
1 52<--25(pid2497)->io_stat_request () = 0 {23 8 0 113388 ...}
2 7<--40(pid2497)->dir_lookup ("proc/" 2097161 0) = 0 1 "/" 52<--47(
    pid2497)
3 52<--47(pid2497)->dir_lookup ("/" 2097161 0) = 0 1 "" 54<--25(
    pid2497)
4 54<--25(pid2497)->dir_readdir (0 -1 0) = 0 "*\0\0\0..." 79
```

- Port 52 is the /proc node port, used for io_stat_request and further dir_lookup calls.
- The caller (PID 2497) interacts with /proc via port 52, which is managed by /hurd/procfs (PID 2446).

netfs_S_dir_lookup in libnetfs is the RPC handler for dir_lookup calls, which ls uses to access /proc. This is the entry point where /hurd/procfs receives requests. After extensive research and attempts, the fact that task id or task port id of the caller doesn't receives to the dir_lookup function but needed data & user credentials (uid, gid, etc) can be access through procfs.

2nd attempt was carried out to get the pid or tid of the caller process through mach_msg_header_t in netfs_S_dir_lookup function. This attempt also failed since mach_msg_header_t doesn't contain caller process information. Therefore conclusion was accessing caller process information is impossible without changing RPC call itself (which have to change how ls works behind the scene) or modifying the mach kernel to include caller process tid in mach_msg_header_t. Changing the RPC call of the

dir_lookup, will affect all the utility applications that are using it. Therefore the 3rd attempt was to modify the mach kernel to add task id of the caller process when passing the RPC from caller to receiver (ls process to procfs process in this scenario). There are two main source files in gnumach source turned out to be important in this modification.

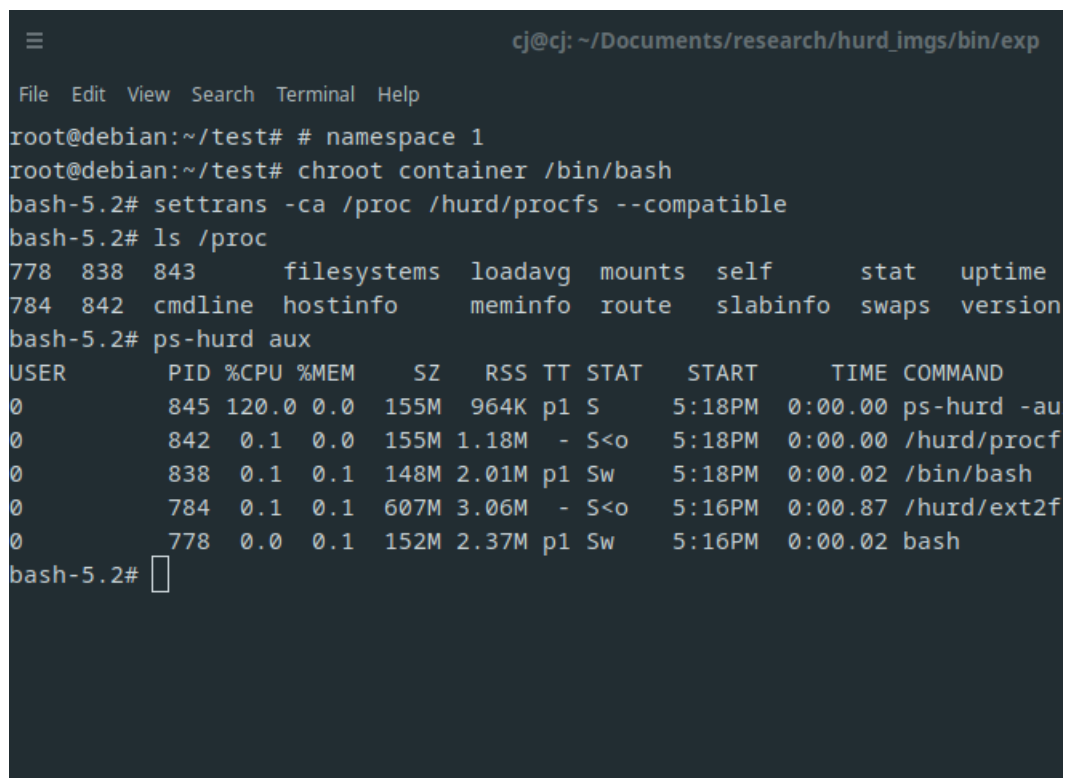
- include/mach/message.h: Define the mach_msg_header_t struct that needs to be modified to contain mach_port_t field(sender's task port).
- ipc/ipc_kmsg.c: Defines functions for message construction and port handling. mach_msg_send (in mach_msg.c) invokes ipc_kmsg_copyin in ipc/ipc_kmsg.c which invokes ipc_kmsg_copyin_header function. This is where the user-space msg_header_t gets translated into the kernel's ipc_kmsg_t structure, including port rights and header fields.

Modifying mach_msg_header_t to contain an additional field of type mach_port_t & ipc_kmsg_copyin_header function to fill the new field of mach_msg_header_t with caller task port, ensures receiver process has the task id of the caller process. Even though this seems practically feasible, applying this will handover the tid of another process to any 3rd party user application which might be a security concern. After having several failed compilations since assertion failures due to the size change in mach_msg_header_t, new kernel binary was installed to the system. But during bootup process, even though mach kernel loaded correctly, hurd system servers and services gets freezed no matter what changes applied to the kernel. This suggests changing something simple part of the kernel will affect all the other components using that modified part. Therefore the procfs namespace awareness attempt concluded as a failure due to security issues as well as changing mach_msg_header_t affects other components like a chain reaction.

Invoke another procfs

Current procfs process is mounted to /proc during bootup to handle dir lookups. Since the attempt to apply process namespace awareness into the procfs failed, the option to having one procfs process to handle lookup requests from different process namespaces

is not feasible. Therefore this attempt is to having a new procfs process to each namespace to handle their lookup requests locally without bothering the procfs in global namespace(NULL). This doesn't means we have to use settrans to mount another procfs to /proc. Because global procfs process is already mounted on /proc. This is where we can use the jail behavior using chroot as shown in section 3.3.1 to have different filesystem to each namespace. By combining this filesystem isolation with implemented process namespace isolation, we can enforce better process isolation. After having a shell inside the jail using the chroot command, /proc is different from real /proc. Now we can use settrans to mount the procfs into fake /proc. This will nicely isolate other processes from namespace processes as shown in figure 4.5.



```

cj@cj: ~/Documents/research/hurd_imgs/bin/exp
File Edit View Search Terminal Help
root@debian:~/test# # namespace 1
root@debian:~/test# chroot container /bin/bash
bash-5.2# settrans -ca /proc /hurd/procfs --compatible
bash-5.2# ls /proc
778 838 843      filesystems  loadavg  mounts  self      stat  uptime
784 842 cmdline  hostinfo  meminfo  route  slabinfo swaps  version
bash-5.2# ps-hurd aux
USER      PID %CPU %MEM    SZ   RSS TT  STAT   START    TIME COMMAND
0          845 120.0 0.0   155M 964K p1 S    5:18PM  0:00.00 ps-hurd -au
0          842  0.1  0.0   155M 1.18M - S<o  5:18PM  0:00.00 /hurd/procfs
0          838  0.1  0.1   148M 2.01M p1 Sw   5:18PM  0:00.02 /bin/bash
0          784  0.1  0.1   607M 3.06M - S<o  5:16PM  0:00.87 /hurd/ext2fs
0          778  0.0  0.1   152M 2.37M p1 Sw   5:16PM  0:00.02 bash
bash-5.2# 

```

Figure 4.5: Enhanced Isolation

Why this works:

- Invoking new procfs process inside the process namespace assigns the relevant namespace to the procfs. Therefore the new procfs is now a process inside the namespace.
- For each request procfs receives will be handled exactly same as before. But, now

proc server knows this procfs is in a namespace and reveals only what it needs to know.

Enhanced Isolation & Final Usage

Final usage of the implemented process namespace isolation, can be demonstrate using the listing 4.17. Since this case doesn't mount procfs inside the isolated filesystem, ls /proc returns nothing. Listing 4.18 shows the commands to execute to copy required binaries, of procfs & settrans and setting up a new procfs process to serve lookup requests from the user processes.

Listing 4.17: Process isolation using unshare

```
1 #!/bin/bash
2 # configure a container
3 mkdir container
4 dd if=/dev/zero of=container.img bs=1M count=100
5 mke2fs container.img
6 settrans -a container /hurd/ext2fs \
7     --writable \
8     container.img
9 mkdir -p container/{bin,lib,sbin,etc,usr,var,tmp,dev,proc,hurd}
10 # copy bash binary and associated libraries
11 cp /usr/bin/bash container/bin/
12 cp /lib/i386-gnu/libtinfo.so.6 container/lib/
13 cp /lib/i386-gnu/libc.so.0.3 container/lib/
14 cp /lib/ld.so.1 container/lib/
15 cp /lib/i386-gnu/libmachuser.so.1 container/lib/
16 cp /lib/i386-gnu/libhurduser.so.0.3 container/lib/
17 cp /lib/ld.so container/lib/
18 # copy ls,ps-hurd,unshare binaries & associated libraries
19 cp ./unshare container/bin/
20 cp /bin/ps-hurd container/bin/
21 cp /usr/bin/ls container/bin/
22 cp /lib/i386-gnu/libps.so.0.3 container/lib/
23 cp /lib/i386-gnu/libpthread.so.0.3 container/lib/
```

```

24 cp /lib/i386-gnu/libihash.so.0.3 container/lib/
25 cp /lib/i386-gnu/libcrypt.so.1 container/lib/
26 cp /lib/i386-gnu/libfshelp.so.0.3 container/lib/
27 cp /lib/i386-gnu/libshouldbeinlibc.so.0.3 container/lib/
28 cp /lib/i386-gnu/libports.so.0.3 container/lib/
29 cp /lib/i386-gnu/libnetfs.so.0.3 container/lib/
30 cp /lib/i386-gnu/libiohelp.so.0.3 container/lib/
31 # enter into the jail using chroot
32 chroot container /bin/bash
33 ps-hurd aux
34 unshare
35 ps-hurd aux

```

Listing 4.18: Procfs isolation using unshare

```

1 #!/bin/bash
2 # copy hurd server binaries - only procfs is enough for now
3 cp -a /hurd/* container/hurd/
4 # copy settrans binary
5 cp /usr/bin/settrans container/bin/
6 # enter into the jail using chroot
7 chroot container /bin/bash
8 ps-hurd aux
9 unshare
10 ps-hurd aux
11 settrans -ca /proc /hurd/procfs --compatible
12 ls /proc

```

Isolation measurements

The proposed design introduces namespace awareness into the GNU/Hurd microkernel by modifying the proc server, establishing a critical foundation for process namespace isolation. While this implementation does not implement the full functionality of process isolation, it represents a pivotal component by enabling the proc server to recognize and manage namespace boundaries for processes. Specifically, the design allows for the

isolation of namespace-specific processes by ensuring that operations such as process enumeration (via the `getallproc` function) are namespace-aware, restricting visibility to processes within the same namespace. Further isolation mechanisms, such as preventing a process from terminating another outside its namespace (through a namespace-aware kill function), can be incrementally implemented by extending namespace awareness to the other exposed functionalities of the proc server. However, due to the constrained timeframe of this research, a comprehensive implementation of such application-level isolation and security mechanisms was not feasible. Developing these features is a complex and time-intensive process, potentially requiring the integration of advanced security components like in Linux (seccomp within the proc server to enforce fine-grained policies).

However pivotal role in proc server in this design, complements GNU/Mach inherent isolation. As stated in 3.7, GNU/Mach provides isolated IPCs and message passing between processes using port rights & capabilities. Therefore providing process namespace isolation is enough to leverage IPC namespace isolation since processes are not visible beyond namespaces makes IPCs automatically cannot pass between processes of different namespaces. Also, enhancing namespace isolations like auth & network can be done in the same way `procfs` was enhanced(examining pros and cons): either invoking each Inside a namespace or enforcing namespace awareness into them.

4.3.3 Design Decision

In designing namespace awareness for GNU/Hurd, we have logically chosen the design that implements namespace awareness primarily within the proc server rather than embedding it inside the Mach kernel. This choice aligns with Hurd's minimalist design philosophy, which excludes policies from the kernel, ensuring modularity and flexibility for future extensions while maintaining the microkernel's separation of concerns.

Chapter 5

Results & Evaluation

5.1 Setup Environments

Since many popular tools are designed for the Linux kernel, smaller open-source benchmarks compatible with Hurd's glibc interface were utilized. Setting Up tinymembench and c-cpu-bench is exactly the same as instructed in their git repositories (linux compatible). The source code of fs_mark have to ported into GNU/hurd to configure on the system correctly. (ssvb 2025), (freshe 2025), (Bacik 2025)

Since comparisons have to be made between linux namespaces and designed GNU/hurd namespaces, linux server is used inside a Qemu environment with the same settings applied to the hurd instance as shown in table 5.2. Also, the same host machine (figure 5.1) was used to instantiate both linux and hurd images.

5.2 Isolation Analysis

The main focus of this reasearch is to improve container isolation using the microkernel architecture. While assess and compare isolation is subjective as we discussed in section 1.2, the comparisons & assessments was done qualitatively as well as quantitatively between linux namespaces and GNU/hurd namespaces to get insights on how the design of GNU/hurd namespaces can improve isolation.

It is worth noting that Linux's own namespace isolation framework evolved incremen-

Table 5.1: CPU Information of the Host Laptop

Attribute	Value
Architecture	x86_64
CPU op-mode(s)	32-bit, 64-bit
Address Sizes	39 bits physical, 48 bits virtual
Byte Order	Little Endian
CPU(s)	8
Threads per Core	2
Cores per Socket	4
Socket(s)	1
Model Name	11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz
CPU Min MHz	400.0000
CPU Max MHz	4700.0000
Virtualization	VT-x
L1d Cache	192 KiB (4 instances)
L1i Cache	128 KiB (4 instances)
L2 Cache	5 MiB (4 instances)
L3 Cache	12 MiB (1 instance)
NUMA Node(s)	1
NUMA Node0 CPU(s)	0-7

tally over more than a decade (Smalley 2016), (Smalley 2019b), with features like PID, mount, and network namespaces introduced and refined progressively. Similarly, the isolation landscape in microkernels like Hurd is a broader and multi domain that cannot be fully analyzed or realized in the early stages of this research. The proposed design, not limited to process namespaces. It establishes a modular and extensible foundation where other namespaces can be implemented in the future.

5.2.1 Qualitative Isolation

These are the major isolation improvements that can be achieved using the proposed design because of the microkernel architecture and the design itself.

Isolated filesystems from one another

In GNU/hurd, file system server (ext2fs) is a separate process unlike in linux. Designing and implementing GNU/hurd namespaces, will inherit the advantage of GNU/hurds' ability to remain unharmed even if the filesystem was failed due to an unintentional behavior. But the important point here is, combining process namespace creation with the

chroot assigns a new filesystem (ext2fs) to the created namespace. Therefore, failures of internal filesystem of a process namespace will not affect to the host machines' filesystem or other namespaces' filesystems. While this is the case in theory, can be shown by creating two different namespaces and monitoring the behavior of each including the host system during a filesystem failure.

In figure 5.1, two namespaces were created. There is a separate filesystem process running inside each namespace. The filesystem (ext2fs) process of NAMESPACE (NS) 1 is 969 and NS2 is 882. Then from the host system, pid 969 was killed to simulate filesystem failure. Figure 5.2 shows the behavior of the NS1 after filesystem failure. The host system and NS2 are not affected by the failure of NS1 filesystem. This shows that the filesystem of each namespace is isolated from one another and the host system. The failed filesystem process can be re-invoked to get the continuous service as shown in figure 5.3.

Linux namespaces doesn't have this kind of behavior since failure of the filesystem will crash the entire kernel & other namespaces in the system. This is a significant advantage of the proposed design over linux namespaces since it can be used to create a better isolated environment for applications.

Local system servers per namespace

As presented in section 4.3.2: Using the proposed design, major services like procfs can be invoke in each namespace to have clean and clear separation of services while maintaining GNU/hurd security mechanisms (processes shouldn't have other processes' proc information). This clear separation of the design improves isolation since these services will be served by processes inside the namespace, not from host system or kernel (like linux namespaces). By doing so, any failure of such services only affects to the processes inside the namespace not others. Namespace can create failed translator process again to get the continuous service. This can be shown by creating two namespaces and monitoring the behavior of each namespace including the host system during a procfs failure.

```

cj@cj: ~/Documents/research/hurd_imgs/bin/exp
File Edit View Search Terminal Help
root@debian:~/test# chroot container /bin/bash
bash-5.2# ps-hurd aux
USER      PID %CPU %MEM    SZ   RSS TT STAT   START    TIME COMMAND
0          832 120.0 0.0   155M 960K p1 S    4:42PM  0:00.00 ps-hurd -aux
0          831 11.0  0.1   148M 1.98M p1 Sw   4:41PM  0:00.01 /bin/bash
0          769  0.0  0.1   607M 3.05M  - S<o   4:37PM  0:01.03 /hurd/ext2fs --writable container.img
0          763  0.0  0.1   152M 2.38M p1 Sw   4:36PM  0:00.02 bash
bash-5.2# # namespace 1
bash-5.2# 

```

```

cj@cj: ~/Documents/research/hurd_imgs/bin/exp
File Edit View Search Terminal Help
root@debian:~/unshare# chroot container /bin/bash
bash-5.2# ps-hurd aux
USER      PID %CPU %MEM    SZ   RSS TT STAT   START    TIME COMMAND
0          830 96.0  0.0   155M 964K p0 S    4:41PM  0:00.00 ps-hurd -aux
0          829  1.5  0.1   148M 1.98M p0 Sw   4:41PM  0:00.00 /bin/bash
0          802  0.3  0.1   607M 3.05M  - S<o   4:39PM  0:00.96 /hurd/ext2fs --writable container.img
0          797  0.0  0.1   152M 2.37M p0 Sw   4:39PM  0:00.04 bash
bash-5.2# # namespace 2
bash-5.2# 

```

Figure 5.1: Setup two namespaces to monitor filesystem behavior

```

cj@cj: ~/Documents/research/hurd_imgs/bin/exp
File Edit View Search Terminal Help
root@debian:~/test# chroot container /bin/bash
bash-5.2# ps-hurd aux
USER      PID %CPU %MEM    SZ   RSS TT STAT   START    TIME COMMAND
0          832 120.0 0.0   155M 960K p1 S    4:42PM  0:00.00 ps-hurd -aux
0          831 11.0  0.1   148M 1.98M p1 Sw   4:41PM  0:00.01 /bin/bash
0          769  0.0  0.1   607M 3.05M  - S<o   4:37PM  0:01.03 /hurd/ext2fs --writable container.img
0          763  0.0  0.1   152M 2.38M p1 Sw   4:36PM  0:00.02 bash
bash-5.2# # namespace 1
bash-5.2# ls
bash: ls: command not found
bash-5.2# ps-hurd aux
bash: /bin/ps-hurd: Computer bought the farm
bash-5.2# pwd
/
bash-5.2# exit
root@debian:~/test# 

```

Figure 5.2: Failure of namespace 1 filesystem

In figure 5.4, two namespaces were created inside separate terminals (ssh). The procs process of NS1 is 842 and NS2 is 840. Then from the host system, pid 842

```

cj@cj: ~/Documents/research/hurd_imgs/bin/exp
File Edit View Search Terminal Help
bash: ls: command not found
bash-5.2# ps-hurd aux
bash: /bin/ps-hurd: Computer bought the farm
bash-5.2# pwd
/
bash-5.2# exit
root@debian:~/test# mke2fs container.img
mke2fs 1.46.6 (1-Feb-2023)
128-byte inodes cannot handle dates beyond 2038 and are deprecated
container.img contains a ext2 file system
        created on Fri Apr 25 17:37:35 2025
Proceed anyway? (y,N) y
Creating filesystem with 25600 4k blocks and 25600 inodes

Allocating group tables: done
Writing inode tables: done
Writing superblocks and filesystem accounting information: done

root@debian:~/test# settrans -a container /hurd/ext2fs \
--writable \
container.img

```

Figure 5.3: Re-invoke namespace 1 filesystem

```

cj@cj: ~/Documents/research/hurd_imgs/bin/exp
File Edit View Search Terminal Help
root@debian:~/test# # namespace 1
root@debian:~/test# chroot container /bin/bash
bash-5.2# settrans -ca /proc /hurd/procfs --compatible
bash-5.2# ls /proc
778 838 843      filesystems loadavg mounts self      stat  uptime  vmstat
784 842 cmdline hostinfo meminfo route  slabinfo swaps  version
bash-5.2# ps-hurd aux
USER      PID %CPU %MEM    SZ   RSS TT STAT   START    TIME COMMAND
0          845 120.0 0.0   155M 964K p1 S    5:18PM  0:00.00 ps-hurd -aux
0          842 0.1   0.0   155M 1.18M - S<o  5:18PM  0:00.00 /hurd/procfs --compatible
0          838 0.1   0.1   148M 2.01M p1 Sw   5:18PM  0:00.02 /bin/bash
0          784 0.1   0.1   607M 3.06M - S<o  5:16PM  0:00.87 /hurd/ext2fs --writable container.img
0          778 0.0   0.1   152M 2.37M p1 Sw   5:16PM  0:00.02 bash
bash-5.2#

```

```

cj@cj: ~/Documents/research/hurd_imgs/bin/exp
File Edit View Search Terminal Help
root@debian:~/unshare# # namespace 2
root@debian:~/unshare# chroot container /bin/bash
bash-5.2# settrans -ca /proc /hurd/procfs --compatible
bash-5.2# ls /proc
776 837 844      filesystems loadavg mounts self      stat  uptime  vmstat
796 840 cmdline hostinfo meminfo route  slabinfo swaps  version
bash-5.2# ps-hurd aux
USER      PID %CPU %MEM    SZ   RSS TT STAT   START    TIME COMMAND
0          846 120.0 0.0   155M 964K p2 S    5:19PM  0:00.00 ps-hurd -aux
0          837 0.3   0.1   148M 2.01M p2 Sw   5:18PM  0:00.02 /bin/bash
0          796 0.3   0.1   607M 3.03M - S<o  5:16PM  0:00.99 /hurd/ext2fs --writable container.img
0          776 0.0   0.1   152M 2.37M p2 Sw   5:16PM  0:00.05 bash
0          840 0.0   0.0   155M 1.18M - S<o  5:18PM  0:00.00 /hurd/procfs --compatible
bash-5.2#

```

Figure 5.4: Setup two namespaces to monitor procfs behavior

```

cj@cj: ~/Documents/research/hurd_imgs/bin/exp
File Edit View Search Terminal Help
root@debian:~/test# # namespace 1
root@debian:~/test# chroot container /bin/bash
bash-5.2# settrans -ca /proc /hurd/procfs --compatible
bash-5.2# ls /proc
778 838 843 filesystems loadavg mounts self stat uptime vmstat
784 842 cmdline hostinfo meminfo route slabinfo swaps version
bash-5.2# ps-hurd aux
USER      PID %CPU %MEM    SZ   RSS TT STAT   START     TIME COMMAND
0          845 120.0 0.0   155M 964K p1 S    5:18PM 0:00.00 ps-hurd -aux
0          842 0.1 0.0   155M 1.18M - S<o  5:18PM 0:00.00 /hurd/procfs --compatible
0          838 0.1 0.1   148M 2.01M p1 Sw  5:18PM 0:00.02 /bin/bash
0          784 0.1 0.1   607M 3.06M - S<o  5:16PM 0:00.87 /hurd/ext2fs --writable container.img
0          778 0.0 0.1   152M 2.37M p1 Sw  5:16PM 0:00.02 bash
bash-5.2# ls /proc
bash-5.2#

```

Figure 5.5: Failure of namespace 1 procfs

```

cj@cj: ~/Documents/research/hurd_imgs/bin/exp
File Edit View Search Terminal Help
root@debian:~/test# # namespace 1
root@debian:~/test# chroot container /bin/bash
bash-5.2# settrans -ca /proc /hurd/procfs --compatible
bash-5.2# ls /proc
778 838 843 filesystems loadavg mounts self stat uptime vmstat
784 842 cmdline hostinfo meminfo route slabinfo swaps version
bash-5.2# ps-hurd aux
USER      PID %CPU %MEM    SZ   RSS TT STAT   START     TIME COMMAND
0          845 120.0 0.0   155M 964K p1 S    5:18PM 0:00.00 ps-hurd -aux
0          842 0.1 0.0   155M 1.18M - S<o  5:18PM 0:00.00 /hurd/procfs --compatible
0          838 0.1 0.1   148M 2.01M p1 Sw  5:18PM 0:00.02 /bin/bash
0          784 0.1 0.1   607M 3.06M - S<o  5:16PM 0:00.87 /hurd/ext2fs --writable container.img
0          778 0.0 0.1   152M 2.37M p1 Sw  5:16PM 0:00.02 bash
bash-5.2# ls /proc
bash-5.2# settrans -ca /proc /hurd/procfs --compatible
bash-5.2# ls /proc
778 838 853 filesystems loadavg mounts self stat uptime vmstat
784 852 cmdline hostinfo meminfo route slabinfo swaps version
bash-5.2#

```

Figure 5.6: Re-invoke namespace 1 & 2 procfs translators

was killed to simulate procfs failure. Figure 5.5 shows the behavior of the NS1 after procfs failure. The host system and NS2 are not affected by the failure of NS1 procfs. This shows that the procfs of each namespace is isolated from one another and the host system. Therefore using the proposed design, other translators also can be invoked in each namespace after conducting a necessary assessment of the service to be invoked (because some services better to be shared with the host system like proc server, auth server) to have clean and clear separation of services between namespaces and the host system. The failed procfs process can be re-invoked to get the continuous service as shown in figure 5.6. This is not achievable in linux namespaces since all major services are handled by the kernel subsystems and failure of such subsystem will take down the

host including other namespaces.

Unchanged kernel behavior

When designing the hurd namespace architecture, only one new message passing interface introduced to the system which is `proc_create_namespace` while kernel is not modified. Hence kernel behavior is Unchanged from before and after namespace awareness. In contrast, linux kernel has been changed over the time to implement linux namespaces. This is also a significant benefit of hurd namespaces since using lesser codes in specially the kernel is lesser the bugs in code. Therefore we can consider unchanged kernel behavior ensure better isolation.

5.2.2 Quantitative Isolation

System calls are the fundamental interface between user-space applications and the operating system kernel, enabling privileged operations like I/O, process management, and resource allocation. The number of system calls can be considered as a part of the attack surface, as each call is a potential entry for attackers. For Linux, system calls mean entry points; for GNU/Mach, the attack surface focuses on message-passing interfaces.

Linux System Calls

Linux, a monolithic kernel, offers a fixed set of system calls, The number of system calls varies by kernel version and architecture, with recent data indicating 360-441 system calls for x86-64 in Linux 6.7, as derived from the `arch/x86/entry/syscalls/syscall_64.tbl` file in the kernel source (Torvalds 2025).

GNU/Mach System Calls

GNU/Mach uses message passing for most operations, contrasting with Linux's direct system calls. The attack surface focuses on the message-passing interfaces (197 - (GNU Project 2025)).

So, using a microkernel for namespace design can lower the attack surface of namespaces by about 45%–55%, since it reduces the number of exposed kernel interfaces.

Summary of Isolation Analysis

The primary focus of the proposed design was to design a system capable of supporting namespace isolation mechanisms within the scope of this work, rather than delivering a fully isolated environment. To assess the current implementation, isolation capabilities were evaluated using a combination of chroot and implemented unshare command, which together provides a container-like isolated environment. highlighted characteristics & results indicate that the proposed design effectively enhances process namespace isolation. While a complete analysis of isolation in microkernels requires long-term research and development, this work successfully demonstrates the feasibility of namespace isolation in Hurd, providing a robust foundation for future advancements in containerization within microkernel architectures.

5.3 Performance Comparison

Performance comparisons were conducted between Linux & Hurd. It's important to highlight that GNU Hurd used inside QEMU with KVM optimizations since hurd is not recommended to run on bare-metal. Since the limited capabilities of hurd(Qemu), comparing bare-metal linux with better hardware requirements is not fair. Therefore ubuntu (16.04.7) server 32-bit (i386) version was contained in an image to compare with GNU/hurd. That way both systems have same configurations as shown in figure 5.2. Each test was executed five times on each instances, and the average values were considered for analysis. Each analysis was performed on three situations in each system. Those three situations/states are:

1. **Directly on the host:** Without using any namespace feature, analysis were performed on the host system.

2. **Simple namespace Isolation:** Used process namespace isolation (unshare command) without using a jail environment(chroot).
3. **Enhanced Namespace Isolation:** Used process namespace isolate combined with the jail filesystem isolation (chroot).

Table 5.2: Test Environment Configuration: Linux vs GNU/Hurd

Category	Linux (Ubuntu 22.04)	GNU/Hurd (Debian Hurd 2020)
OS Version	Ubuntu 16.04.7 server	Debian GNU/Hurd-0.9
Architecture	32-bit(i386)	32-bit (i386)
Filesystem	ext2	ext2
Virtualization	KVM Virtual Machine	KVM Virtual Machine
CPU	1 Core	1 Core
RAM	3 GB	3 GB

5.3.1 CPU

Figure 5.7 illustrates the duration taken by each environment to execute a benchmark tasked with calculating the first 5,761,455 primes (freshe 2025).

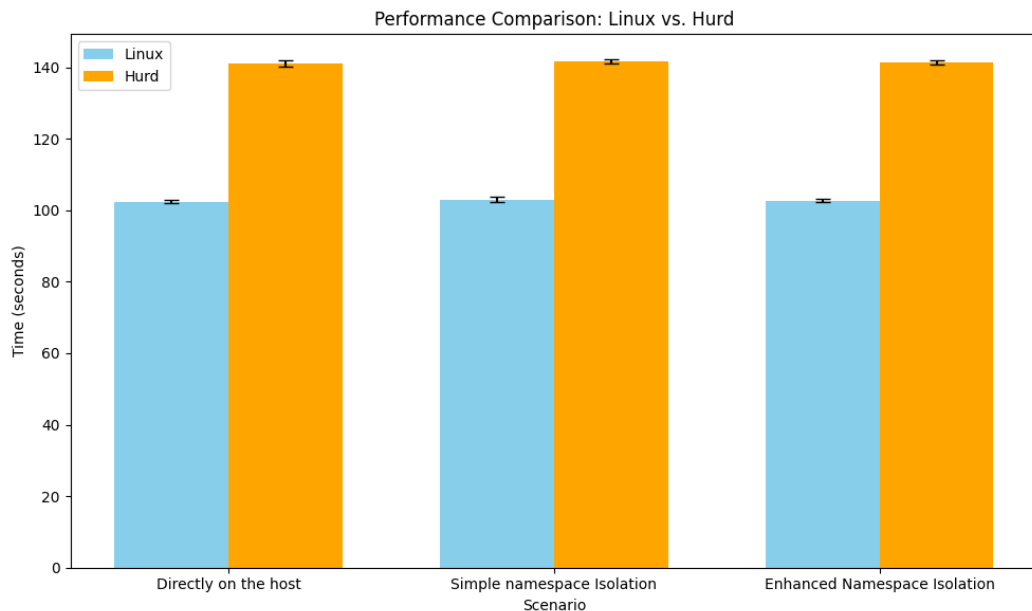


Figure 5.7: CPU Performance Comparison: Linux vs Hurd

Linux takes about 102-103 seconds on average, while Hurd takes 141-142 seconds—a difference of approximately 39 seconds. This is expected due to Linux’s monolithic

kernel design, which handles system calls and thread management more efficiently, versus Hurd’s microkernel architecture, where inter-process communication (IPC) between servers introduces overhead. For a CPU-intensive task like prime number crunching, Hurd’s IPC overhead likely slows down memory access speeds.

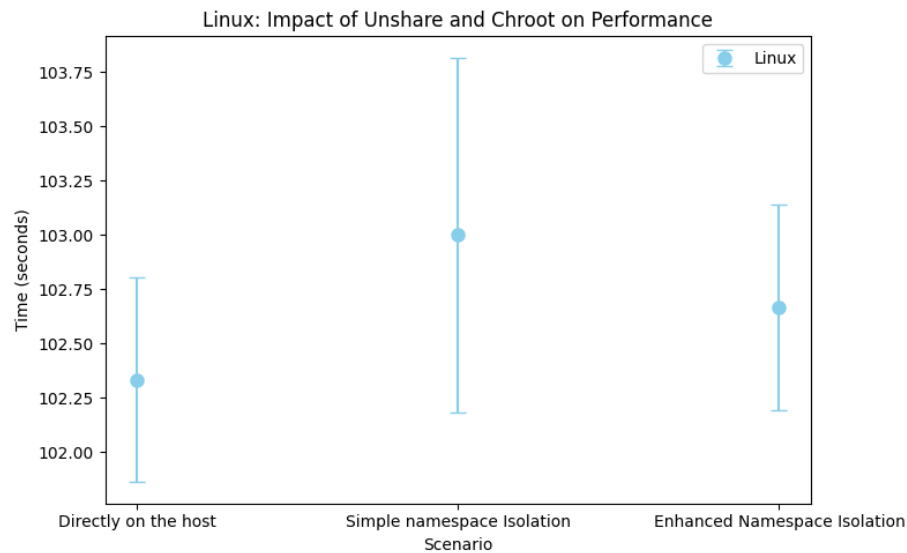


Figure 5.8: Linux: Impact of Enhance namespace isolation on cpu performance

Line plot 5.8 shows the average time for Linux across the three scenarios, with error bars. The time difference across scenarios is less than 1 second, which is negligible for a benchmark running over 100 seconds. The unshare process namespace isolates the PID space, adding a small overhead for process management, but this is minimal for a CPU-bound task like cputest. Adding chroot (filesystem isolation) has almost no impact, as the benchmark doesn’t heavily rely on filesystem access beyond initial loading.

Line plot 5.9 shows the average time for Hurd across the three scenarios, with error bars. Similar to Linux, the time difference across scenarios is less than 1 second, which is negligible for a benchmark running over 140 seconds. Despite Hurd’s slower baseline performance, the unshare process namespace and chroot add minimal overhead, mirroring the behavior seen in Linux. This suggests that the isolation mechanisms (PID namespace, filesystem jail) do not significantly affect CPU-bound workloads, even on Hurd’s microkernel architecture.

Both Linux and Hurd show the same pattern: a maximum increase of 0.67 seconds

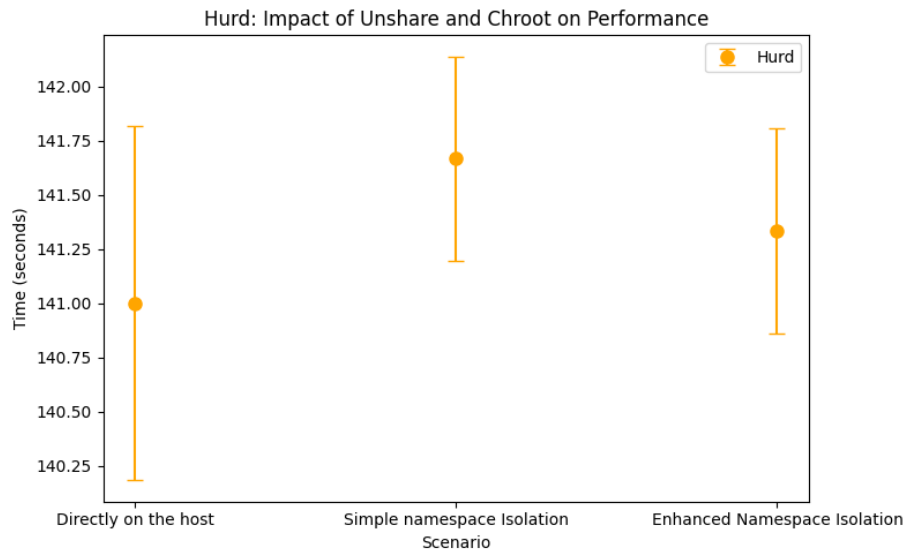


Figure 5.9: Hurd: Impact of Enhance namespace isolation on cpu performance

(0.5% of runtime) when adding unshare, and even less with both unshare and chroot. This consistency across systems, despite Hurd’s slower baseline, shows that the proposed namespace isolation design doesn’t exacerbate cpu-bound performance issues in hurd, making it a viable approach for process isolation in Hurd.

5.3.2 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 1,000 such files were generated to evaluate the performance and effectiveness of the file system logic and interfaces (Bacik 2025).

Figure 5.10 shows application overhead of hurd namespace’s filesystem vs linux namespace’s filesystem. Application Overhead reflects time spent outside filesystem-related system calls, indicating kernel efficiency and scheduling overhead. Hurd’s overhead is 6x higher, reflecting microkernel IPC overhead, while Linux handles filesystem calls more directly.

Figure 5.11 shows how application overhead of each scenario impact on hurd and linux. Both show a slight decrease, suggesting chroot may streamline some operations, though the effect is small.

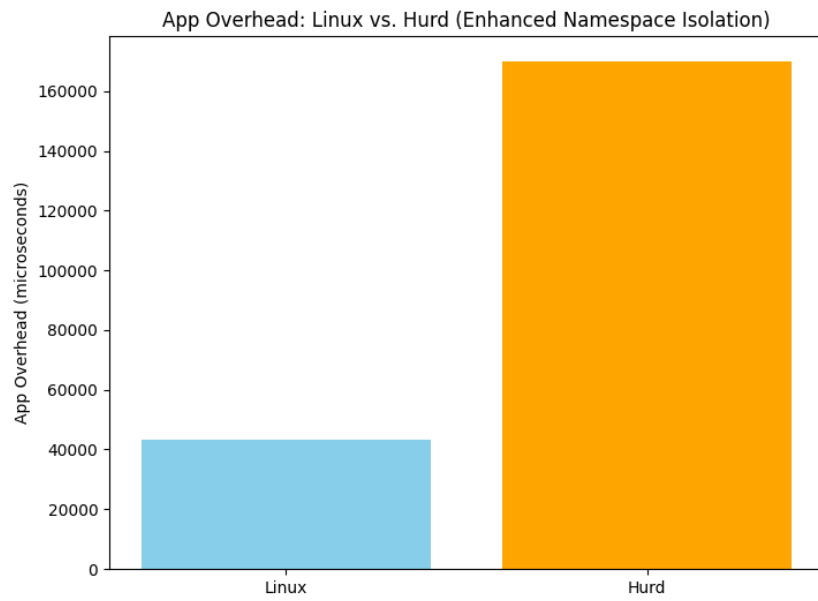


Figure 5.10: Application Overhead:Linux vs Hurd

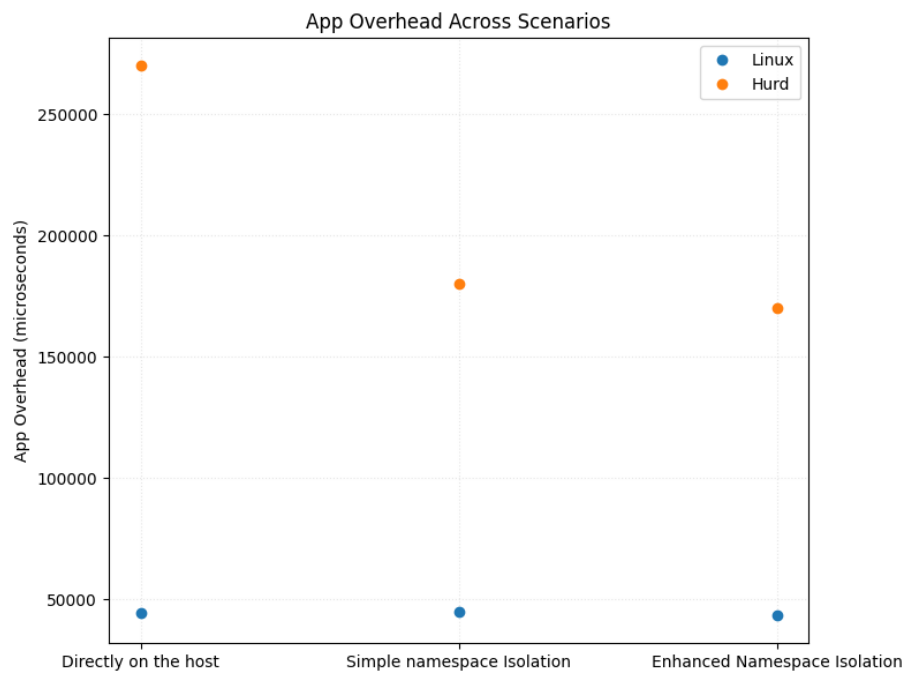


Figure 5.11: App Overhead across scenarios

The graphs provide a clear visual comparison of filesystem performance between Linux and Hurd, as well as the impacts of unshare and chroot. Hurd's high Application Overhead highlights microkernel challenges.

5.3.3 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 (ssvb 2025).

Memory Bandwidth

Memory bandwidth measures how much data can be transferred per second during operations like memory copies. Under enhanced namespace isolation, comparison was done between linux and hurd.

Linux typically delivers high bandwidth due to its optimized, in-kernel memory management as shown in figure 5.12. Hurd’s microkernel design delegates memory management to user-space servers, introducing inter-process communication (IPC) overhead. This often reduces bandwidth as shown in figure 5.12.

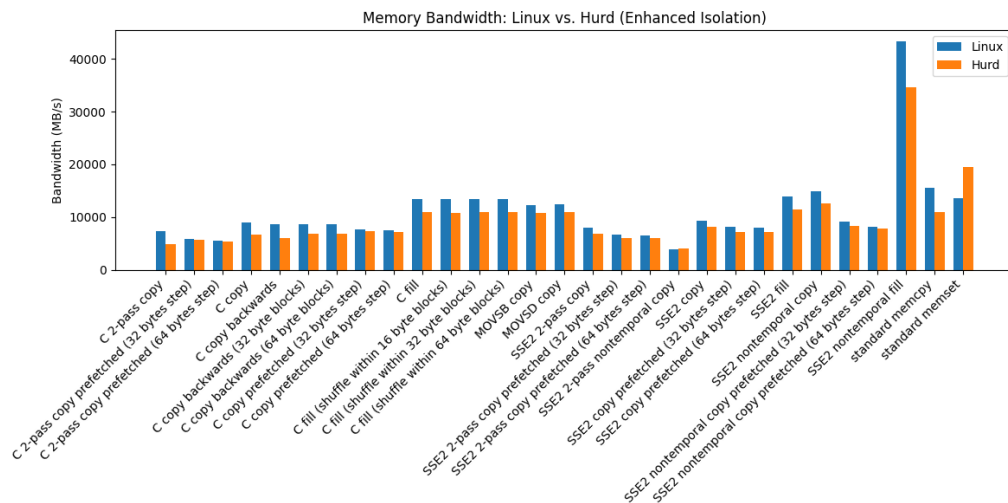


Figure 5.12: Memory Bandwidth: Linux vs Hurd namespace

The enhanced isolation setup on hurd amplify this overhead (figure 5.13) slightly due to additional isolation layer.

Linux’s tight kernel integration keeps isolation overhead tiny. The negligible impact shows enhanced isolation adds almost no friction to its memory pipeline, even for optimized operations (figure 5.14).

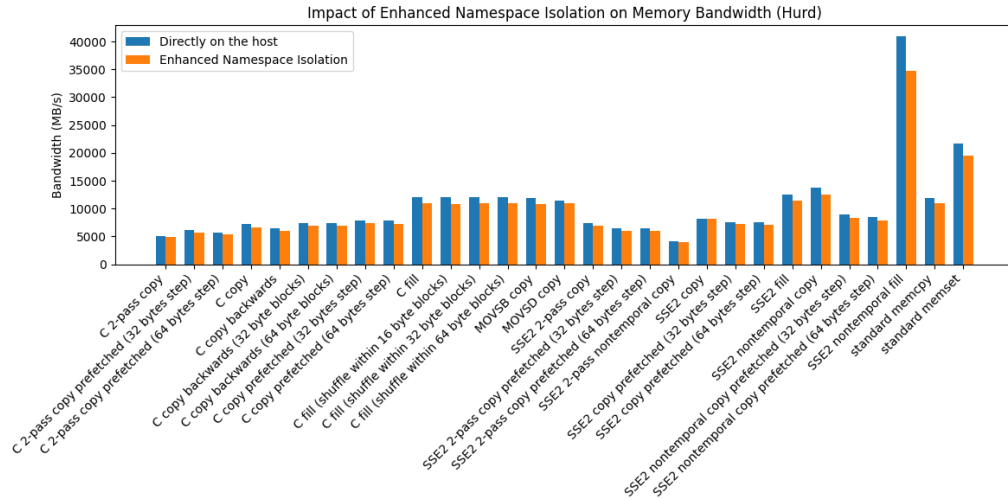


Figure 5.13: Namespace Isolation on Memory Bandwidth: hurd

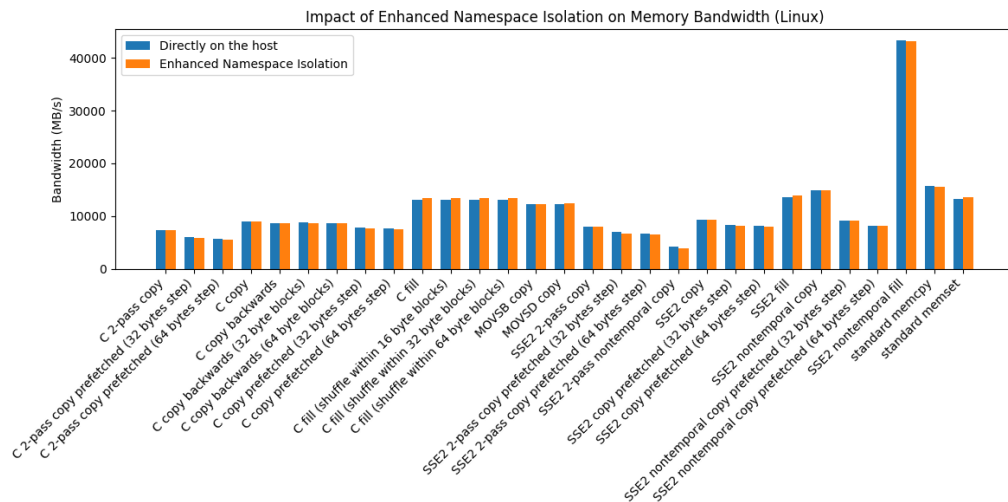


Figure 5.14: Namespace Isolation on Memory Bandwidth: linux

Memory Latency

Memory latency measures the time for a single random read, varying with block size. Comparisons was done in three scenarios to get insights on how namespace isolation affects this. Figure 5.15 shows memory latency on linux vs hurd in enhanced isolation scenario. Hurd's latency is 118.0 ns, 15.8% higher than Linux's 101.9 ns. Linux handles large memory accesses faster, thanks to direct kernel management. Hurd's microkernel IPC slows things down, as memory requests bounce between processes, adding delays.

Figure 5.16 examine how isolation affects Hurd's memory latency. Latency rises from 112.0 ns to 118.0 ns, a 5.4% increase.

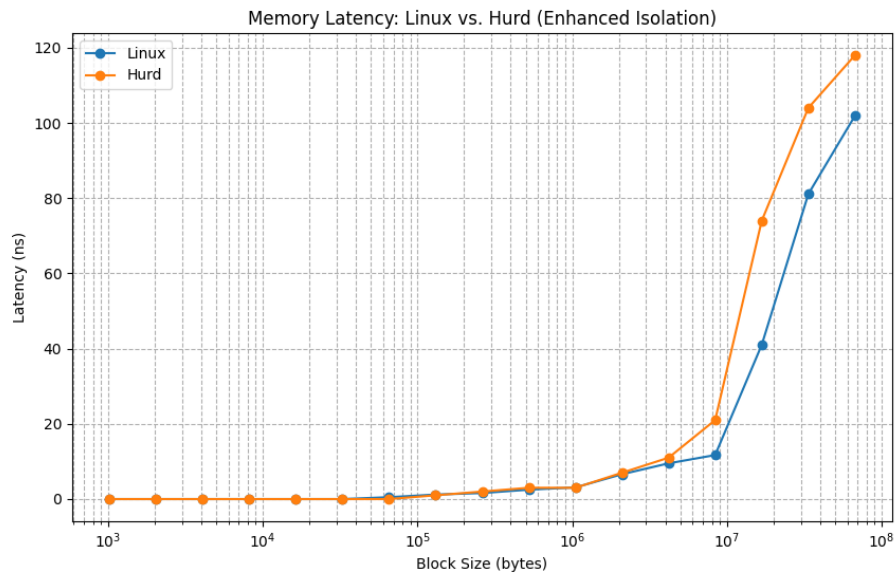


Figure 5.15: Memory Latency: Linux vs Hurd namespace

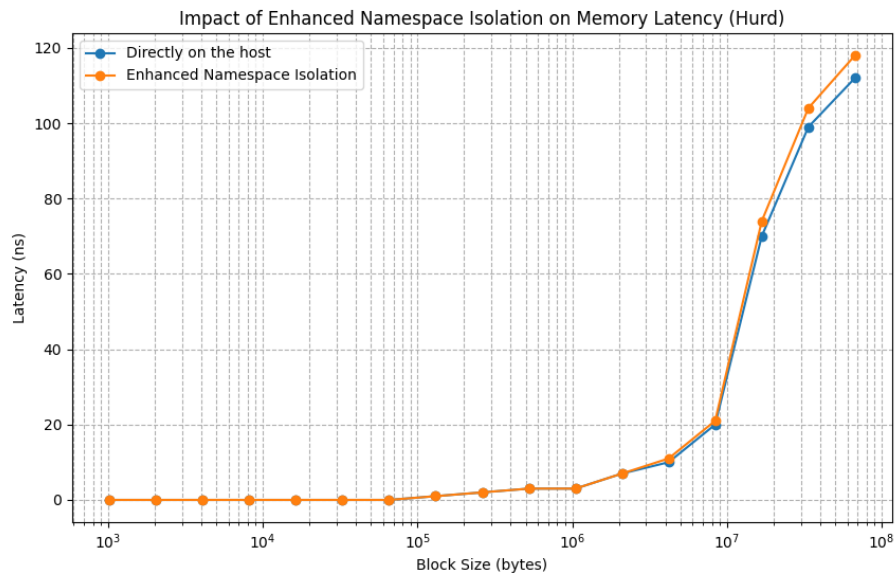


Figure 5.16: Namespace Isolation on Memory Latency: hurd

Figure 5.17 examine how isolation affects Linux’s memory latency. Latency edges up from 100.9 ns to 101.9 ns, a 1.0% bump ($(101.9 - 100.9) / 100.9 \times 100 \approx 1.0$). This tiny increase reflects Linux’s efficiency.

Linux’s monolithic design yields better performance, while Hurd’s microkernel shows moderate penalties. The 7–15% bandwidth drop and 5.4% latency rise, plus lower baseline performance, are noticeable but acceptable for non-critical tasks where security

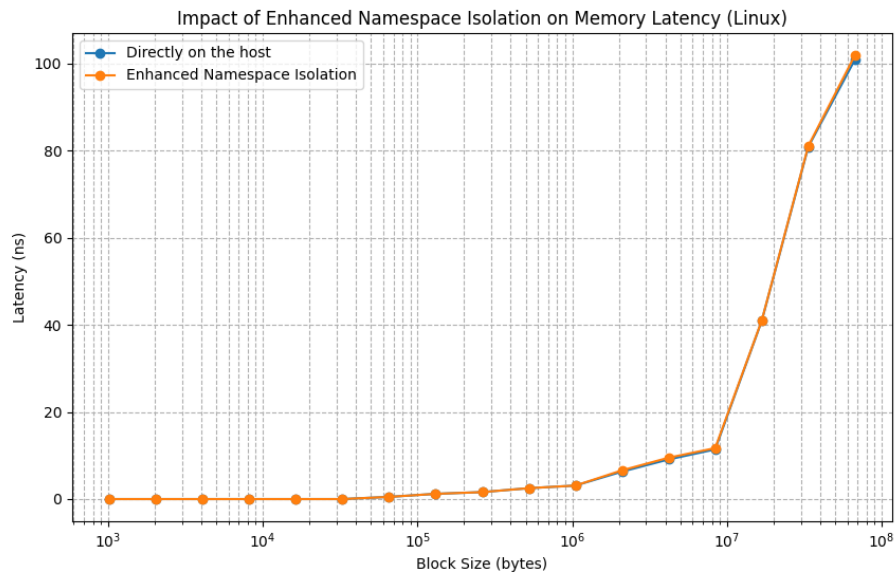


Figure 5.17: Namespace Isolation on Memory Latency: linux

or modularity matters more. Performance-sensitive apps might need optimization. But Hurd's lack of modern enhancements compared to Linux, specially in memory management, can explain the observed differences in the tinymembench memory benchmark results. Since the main focus of this research is to improve isolation using microkernel, moderate performance panalties can be opt-out for now expecting future enhancements. But here the important point is, the proposed design of GNU/hurd namespaces doesn't add noticeable overhead to the performance of the system.

Chapter 6

Conclusions

6.1 Limitations

While the proposed design successfully introduces namespace awareness into GNU/Hurd through modifications to the proc server: enabling process namespace isolation, several limitations highlight areas for future improvements. These limitations span performance, filesystem operations, isolation scope, security policies, namespace coverage.

Performance Overhead

The implementation has performance overheads in memory operations, as evidenced by tinymembench results. These overheads stem from Hurd’s microkernel architecture, where IPC between user-space servers introduces additional latency and reduces throughput. While acceptable for non-critical workloads, this performance gap may limit the design’s suitability for memory-intensive applications.

Scope of Isolation

The current design is limited to process namespace isolation, covering functionalities like process enumeration (e.g., `getallproc`) within the proc server. However, it does not prevent other cross-namespace interactions, such as a process sending signals (e.g., via `kill`) to processes outside its namespace. This partial isolation isn’t enough for the full

process separation needed for strong containerization, which means processes could still interfere with each other across namespaces, creating potential risks. Extending namespace awareness to all proc server functionalities, such as signal handling or resource queries, is necessary to achieve full process isolation, a task that was infeasible within the project’s timeframe due to the time taken to modifying multiple server interfaces.

Security Policies

The implementation lacks integration with advanced security policies, such as those provided by seccomp or SELinux in Linux, which enforce fine-grained access controls. Without such policies, the design cannot fully mitigate risks like privilege escalation or unauthorized resource access, limiting its security guarantees compared to mature container systems.

Support for Other Namespaces

The design currently supports only process namespaces, leaving other critical namespace types—such as network, mount, and user namespaces—unaddressed. In Linux, these namespaces evolved over a decade, with features like network namespaces integrated by 2019 to handle netlink notifications (Smalley 2019a), and SELinux support for containers maturing by 2022 (Moore 2023). Implementing these in Hurd would require extending namespace awareness to additional system servers and potentially developing new translators to manage namespace-specific resources, a complex endeavor beyond the project’s scope but feasible given the design’s extensible framework.

Despite these limitations, the design’s modularity and alignment with Hurd’s micro-kernel principles offer a robust foundation for addressing these gaps. The use of system servers and translators, as demonstrated with the proc server and procfs, provides a scalable mechanism to implement other namespaces and isolation features. For instance, re-invoking pfinet translator inside a process namespace could enable network namespace isolation, while a namespace-aware ext2fs translator could support mount namespaces,

mirroring Linux’s comprehensive namespace model. Additionally, integrating security policies inspired by SELinux, which enforces mandatory access controls, could enhance the design’s security profile, making it a viable platform for containerization in microkernel architectures with continued research and development.

6.2 Future Works

This research began by exploring the applicability of microkernels to enhance container security, investigating various OS designs and their characteristics, with a focus on microkernels like GNU/Hurd. Building on prior discussions of OS reliability issues and microkernel-based solutions proposed by Tanenbaum, Herder, and Bos (2006), which emphasize the security and isolation benefits of microkernels—key requirements for containerization—this work implemented namespace awareness in Hurd through a new `unshare` command, modifying the `proc` server to enable process namespace isolation according to the proposed design. Combined with `chroot`, this approach achieves container-like isolation, addressing some of the security and isolation challenges inherent in traditional container technologies like Docker and Singularity, which share the host OS kernel and thus face vulnerabilities. Performance evaluations reveal acceptable overheads in Hurd, with memory bandwidth and latency impacts of 7–15% and 5.4%. The proposed design perfectly complements `mach` microkernel isolation features such as isolated IPCs and message passing showing potential to implement other namespace isolation directly with the provided process namespace isolation.

Since the proposed design provide the foundation, future research should focus on extending namespace awareness to additional system servers and translators, leveraging Hurd’s modular architecture to achieve comprehensive resource isolation. Additionally, integrating advanced security policies, such as `seccomp` or SELinux-inspired mechanisms, into Hurd’s `proc` server could mitigate risks like privilege escalation, enhancing container security beyond what current capabilities offer (Moore 2023), (*namespace_selinux(8)* - *Linux manual page* 2025).

This research proves that microkernel-based approaches remain promising, specially given Hurd's inherent isolation advantages. With growing security concerns, developing microkernel-based operating systems capable of containerization is not merely an option but a high-priority requirement, and this research lays a foundational framework for achieving that goal through extensible namespace support and enhanced isolation mechanisms.

Bibliography

- Bacik, Josef (2025). *fs_mark: A simple file system benchmark*. https://github.com/josefbacik/fs_mark. Accessed: 2025-04-14.
- Bentaleb, Ouafa et al. (2022). “Containerization technologies: Taxonomies, applications and challenges”. In: *The Journal of Supercomputing* 78.1, pp. 1144–1181.
- Bhardwaj, Aditya and C Rama Krishna (2021). “Virtualization in cloud computing: Moving from hypervisor to containerization—a survey”. In: *Arabian Journal for Science and Engineering* 46.9, pp. 8585–8601.
- Biederman, Eric W and Linux Networx (2006). “Multiple instances of the global linux namespaces”. In: *Proceedings of the Linux Symposium*. Vol. 1. 1. Citeseer, pp. 101–112.
- Crosby, Simon and David Brown (2006). “The Virtualization Reality: Are hypervisors the new foundation for system software?” In: *Queue* 4.10, pp. 34–41.
- Debian GNU/Hurd Maintainers (2023). *Debian GNU/Hurd 2023 "Bookworm" - Unofficial hurd-i386*. <https://cdimage.debian.org/cdimage/ports/stable/hurd-i386/README.txt>. URL: <https://cdimage.debian.org/cdimage/ports/stable/hurd-i386/README.txt>.
- Debian Project (2025). *Debian GNU/Hurd — Installation*. Accessed: 2025-04-24. URL: <https://www.debian.org/ports/hurd/hurd-install>.
- Dennis, Jack B. and Earl C. Van Horn (Mar. 1966). “Programming semantics for multiprogrammed computations”. In: *Commun. ACM* 9.3, pp. 143–155. ISSN: 0001-0782. DOI: 10.1145/365230.365252. URL: <https://doi.org/10.1145/365230.365252>.
- Elphinstone, Kevin and Gernot Heiser (2013). “From L3 to seL4 what have we learnt in 20 years of L4 microkernels?” In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP ’13. Farmington, Pennsylvania: Association for Computing Machinery, pp. 133–150. ISBN: 9781450323888. DOI: 10.1145/2517349.2522720. URL: <https://doi.org/10.1145/2517349.2522720>.
- freshe (2025). *c-cpu-bench: Minimal C CPU benchmark with benchmarking framework*. <https://github.com/freshe/c-cpu-bench>. Accessed: 2025-04-14.
- “GNU Hurd/ Documentation” (June 2017). In: URL: <https://www.gnu.org/software/hurd/documentation.html>.
- “GNU Hurd/ Documentation” (May 2024). In: URL: <https://darnassus.sceen.net/%7Ehurd-web/documentation/>.
- GNU Project (2016a). *Running a chroot in GNU/Hurd*. <https://www.gnu.org/software/hurd/hurd/running/chroot.html>. Accessed: 2025-04-24. URL: <https://www.gnu.org/software/hurd/hurd/running/chroot.html>.

- GNU Project (2016b). *Running a chroot in GNU/Hurd*. <https://darnassus.sceen.net/~hurd-web/hurd/running/chroot/>. Accessed: 2025-04-24. URL: <https://darnassus.sceen.net/~hurd-web/hurd/running/chroot/>.
- (2025). *GNU Mach Microkernel*. <https://git.savannah.gnu.org/git/hurd/gnumach.git/>. Accessed: 2025-04-15. Free Software Foundation.
- gnu.org (2008-11-13a). *Inter Process Communication*. Accessed from gnu.org. URL: <https://www.gnu.org/software/hurd/gnumach-doc/Inter-Process-Communication.html#Inter-Process-Communication> (visited on 04/14/2025).
- (2008-11-13b). *The GNU Mach Reference Manual*. Accessed from gnu.org. URL: <https://www.gnu.org/software/hurd/gnumach-doc/> (visited on 04/14/2025).
- gnu.org/ (2003a). *The GNU/Hurd User's Guide: Bootstrap*. URL: https://www.gnu.org/software/hurd/users-guide/using_gnuhurd.html#Bootstrap (visited on 04/14/2025).
- (2003b). *The GNU/Hurd User's Guide: Translators*. URL: https://www.gnu.org/software/hurd/users-guide/using_gnuhurd.html#Translators (visited on 04/14/2025).
- (Mar. 2017). *SubHurd*. URL: <https://www.gnu.org/software/hurd/hurd/subhurd.html> (visited on 04/14/2025).
- Hakamian, A. and A. Rahmani (2015). "Evaluation of isolation in virtual machine environments encounter in effective attacks against memory". In: *Security and Communication Networks* 8. DOI: 10.1002/sec.1374.
- Hallyn, Serge E. (2013). *Namespaces in operation, part 1*. Published on LWN.net. URL: <https://lwn.net/Articles/531114/> (visited on 04/14/2025).
- Hevner, Alan (Jan. 2007). "A Three Cycle View of Design Science Research". In: *Scandinavian Journal of Information Systems* 19.
- IBM (Feb. 2025). *Virtual Machines*. URL: <https://www.ibm.com/think/topics/virtual-machines> (visited on 04/14/2025).
- Isaac, Odun-Ayo et al. (Apr. 2021). "An Overview of Microkernel Based Operating Systems". In: *IOP Conference Series: Materials Science and Engineering* 1107.1, p. 012052. DOI: 10.1088/1757-899X/1107/1/012052. URL: <https://dx.doi.org/10.1088/1757-899X/1107/1/012052>.
- Kamp, Poul-Henning and Robert NM Watson (2000). "Jails: Confining the omnipotent root". In: *Proceedings of the 2nd International SANE Conference*. Vol. 43, p. 116.
- Kerrisk, Michael (2025). *unshare(2) - Linux manual page*. Accessed from man7.org. URL: <https://man7.org/linux/man-pages/man2/unshare.2.html> (visited on 04/14/2025).
- Klein, Gerwin et al. (2014). "Comprehensive formal verification of an OS microkernel". In: *ACM Transactions on Computer Systems (TOCS)* 32.1, pp. 1–70.
- Landaeta, Pedro Selencio (2024). "Uso de PROTEUS y la CYCLONE II para la Enseñanza Práctica de VHDL y FPGA por medio de Software y Hardware". In: *Perfiles de Ingeniería* 21.22, pp. 60–76.
- Levin, R. et al. (1975). "Policy/mechanism separation in Hydra". In: *Proceedings of the Fifth ACM Symposium on Operating Systems Principles*. SOSP '75. Austin, Texas, USA: Association for Computing Machinery, pp. 132–140. ISBN: 9781450378635.

- DOI: 10.1145/800213.806531. URL: <https://doi.org/10.1145/800213.806531>.
- Liedtke, J. (Dec. 1995). “On micro-kernel construction”. In: *SIGOPS Oper. Syst. Rev.* 29.5, pp. 237–250. ISSN: 0163-5980. DOI: 10.1145/224057.224075. URL: <https://doi.org/10.1145/224057.224075>.
- Loepere, Keith (1992). *Mach 3 kernel principles*.
- Moore, Paul (2023). “SELinux and Namespaces”. In: *LWN.net*. URL: <https://lwn.net/Articles/885004/> (visited on 04/22/2025).
- Morabito, Roberto, Jimmy Kjällman, and Miika Komu (2015). “Hypervisors vs. lightweight virtualization: a performance comparison”. In: *2015 IEEE International Conference on cloud engineering*. IEEE, pp. 386–393.
- namespace_selinux(8) - Linux manual page* (2025). Accessed from linux.die.net. URL: https://linux.die.net/man/8/namespace_selinux (visited on 04/22/2025).
- Perera, B. Ravin (Apr. 2024). “Improving Low-Level Isolation of Containers: Leveraging Microkernel Design”. In.
- Popek, G. J. and R. P. Goldberg (1974). “Formal requirements for virtualizable third generation architectures”. In: *Communications of the ACM* 17.7, pp. 412–421. DOI: 10.1145/361011.361073.
- Prins, T (2022). “Containerization in Trusted Computing”. In: *In Proceedings of the Ground Vehicle Systems Engineering and Technology Symposium (GVSETS)*.
- R. Espinola, Stallman’s Dream (2009). “GNU Hurd/ Documentation”. In: URL: <https://raulespinola.wordpress.com/2009/02/12/el-sueno-de-stallman-gnu-hurd/>.
- Raho, Moritz et al. (2015). “KVM, Xen and Docker: A performance analysis for ARM based NFV and cloud computing”. In: *2015 IEEE 3rd Workshop on Advances in Information, Electronic and Electrical Engineering (AIEEE)*. IEEE, pp. 1–8.
- Rana et al. (June 2023). “A Survey on Microkernel Based Operating Systems and Their Essential Key Components”. In: DOI: 10.2139/ssrn.4467406. URL: <http://dx.doi.org/10.2139/ssrn.4467406>.
- Randal, Allison (Feb. 2020). “The Ideal Versus the Real: Revisiting the History of Virtual Machines and Containers”. In: *ACM Comput. Surv.* 53.1. ISSN: 0360-0300. DOI: 10.1145/3365199. URL: <https://doi.org/10.1145/3365199>.
- Saltzer, Jerome H, David P Reed, and David D Clark (1984). “End-to-end arguments in system design”. In: *ACM Transactions on Computer Systems (TOCS)* 2.4, pp. 277–288.
- Sharma, Srinarayan and Young Park (2011). “Virtualization: A review and future directions executive overview”. In: *American Journal of Information Technology* 1, pp. 1–37.
- Smalley, Stephen (2016). *SELinux: Add support for per-namespace policy*. <https://lore.kernel.org/selinux/1459958221.7680.2.camel@gmail.com/>. Accessed: 2025-04-22.
- (2019a). *selinux: add support for SELinux policy namespaces*. <https://lore.kernel.org/selinux/20191015132528.13519-8-sds@tycho.nsa.gov/t/>. Linux Kernel Mailing List, Patch Submission. URL: <https://lore.kernel.org/selinux/20191015132528.13519-8-sds@tycho.nsa.gov/t/>.

- Smalley, Stephen (2019b). *SELinux: Patch series for namespace support*. <https://lore.kernel.org/selinux/20191015132528.13519-8-sds@tycho.nsa.gov/>. Accessed: 2025-04-22.
- ssvb (2025). *tinymembench: Simple memory benchmark for ARM and x86 systems*. <https://github.com/ssvb/tinymembench>. Accessed: 2025-04-14.
- Stankov, Ivan and Grisha Spasov (2006). “Discussion of microkernel and monolithic kernel approaches”. In: *International Scientific Conference Computer Science*. sn.
- Tamane, Sharvari (2015). “A review on virtualization: A cloud technology”. In: *International Journal on Recent and Innovation Trends in Computing and Communication* 3.7, pp. 4582–4585.
- Tanenbaum, Andrew S (1995). “A comparison of three microkernels”. In: *The Journal of Supercomputing* 9, pp. 7–22.
- Tanenbaum, Andrew S, Jorrit N Herder, and Herbert Bos (2006). “Can we make operating systems reliable and secure?” In: *Computer* 39.5, pp. 44–51.
- Tanenbaum, Andrew S and Albert S Woodhull (1997). *Operating systems: design and implementation*. Vol. 68. Prentice Hall Englewood Cliffs.
- The Linux Kernel Community (2025a). *Cgroup v2 - Linux kernel documentation*. Accessed from kernel.org. URL: <https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v2.html> (visited on 04/14/2025).
- (2025b). *Namespaces - Linux kernel documentation*. Accessed from kernel.org. URL: <https://www.kernel.org/doc/html/latest/admin-guide/namespaces/index.html> (visited on 04/14/2025).
- Torvalds, Linus (2025). *The Linux Kernel Source Tree*. <https://github.com/torvalds/linux>. Accessed: 2025-04-14.
- Uhlig, Rich et al. (2005). “Intel virtualization technology”. In: *Computer* 38.5, pp. 48–56.
- VMWARE, INC (2007). “Understanding Full Virtualization”. In: *Paravirtualization and Hardware Assist*.
- Walfield, Neal H. and Marcus Brinkmann (July 2007). “A critique of the GNU hurd multi-server operating system”. In: *SIGOPS Oper. Syst. Rev.* 41.4, pp. 30–39. ISSN: 0163-5980. DOI: 10.1145/1278901.1278907. URL: <https://doi.org/10.1145/1278901.1278907>.
- Watada, Junzo et al. (2019). “Emerging trends, techniques and open issues of containerization: A review”. In: *IEEE Access* 7, pp. 152443–152472.

Appendix

Listing 6.1: runHurd script

```
1  #!/bin/bash
2  if [ -n "${1}" ]; then
3      port=2222
4      ram=1
5      if [ -n "${2}" ]; then
6          port=$2
7      fi
8      if [ -n "${3}" ]; then
9          ram=$3
10     fi
11     echo "File_name_is_$1"
12     echo "Trying_to_Run_the_img_file"
13     kvm \
14         -net user,hostfwd=tcp:127.0.0.1:$(echo $port)-:22 -
15         net nic,model=e1000 \
16         -drive format=raw,file=$(echo $1),cache=writeback -m
17         $(echo $ram)G &
18     #if [ "${$?}" -eq "0" ]; then
19         #echo "success"
20     #fi
21 else
22     echo "Missing_file_name"
23 fi
```



```

com 2 out of range
lpr0: at atbus1, port = 378x, spl = 6d, pic = 7.
RTC time is 2025-02-19 19:08:16
module 0: acpi --host-priv-port=${host-port} --device-master-port=${device-port}
--next-task=${pci-task} $(acpi-task=task-create) $(task-resume)
module 1: pci-arbiter --next-task=${disk-task} $(pci-task=task-create)
module 2: rumpdisk --next-task=${fs-task} $(disk-task=task-create)
module 3: ext2fs --readonly --multiboot-command-line=${kernel-command-line} --ex
ec-server-task=$(exec-task) -T typed ${root} $(fs-task=task-create)
module 4: exec /hurd/exec $(exec-task=task-create)
5 multiboot modules
task loaded: acpi --host-priv-port=1 --device-master-port=2 --next-task=3
task loaded: pci-arbiter --next-task=1
task loaded: rumpdisk --next-task=1
task loaded: ext2fs --readonly --multiboot-command-line=root=part:2:device:hd0 -
-exec-server-task=1 -T typed part:2:device:hd0
task loaded: exec /hurd/exec

start acpi pci rumpdisk Kernel is already driving an IDE device, skipping
probing disks
Hurd server bootstrap: ext2fsIpart:2:device:hd0I exec startup proc authHELLO FRO
M NEW PROC SERUER, going to main loop for process messages...
/hurd/startup: ../../startup/startup.c:878: launch_core_servers: Unexpected erro
r: (ipc/mig) server type check failure.
-

```

System failure due to libc errors

```

struct proc *
create_init_proc (void)
{
    static const uid_t zero;
    struct proc *p;
    const char *rootsname = "root";

    p = allocate_proc (MACH_PORT_NULL);
    assert_backtrace (p);

    /*
    by charith
    */
    p->p_namespace = NULL;
    /*
    end charith
    */
    p->p_pid = HURD_PID_INIT;

    p->p_parent = p;
    p->p_sib = 0;
    p->p_prevsib = &p->p_ochild;
    p->p_ochild = p;
    p->p_parentset = 1;

    p->p_deadmsg = 1; /* Force initial "re-"fetch of msgport. */
    p->p_important = 1;

    p->p_id = make_ids (&zero, 1, &zero, 0);
    assert_backtrace (p->p_id);

    p->p_loginleader = 1;
    p->p_login = malloc (sizeof (struct login) + strlen (rootsname) + 1);
    assert_backtrace (p->p_login);

    p->p_login->l_refcnt = 1;
}

```

The create_init_proc function

Index

ARP poisoning, 20

cgroups, 18

chroot, 18

Docker, 18

GNU Hurd Microkernel, 25

GNU project, 3

L4 Microkernel, 24

L4RE Microkernel, 24

LXC, 18

Mach Microkernel, 23

Minix 3 Microkernel, 25

modular monolithic kernels, 22

namespaces, 18

QNX Neutrino Microkernel, 25

Seccomp, 18

SeL4 Microkernel, 24

Sing#, 30

Singularity, 19