ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

DIPARTIMENTO DI INFORMATICA - SCIENZA E INGEGNERIA

Corso di Laurea in Informatica

Experiments on KVM-based system call virtualization

Supervisor:
Chiar.mo Prof.
Renzo Davoli

Presented by:
Fabio Murer

II Sessione
Anno accademico 2024/2025

This work is licensed under a Creative Commons “Attribution-ShareAlike 4.0 International” license.

Sommario

Le tecnologie di virtualizzazione hanno trasformato il modo in cui i moderni sistemi informatici vengono progettati, gestiti e messi in sicurezza. Tradizionalmente, le macchine virtuali di sistema offrono un forte isolamento emulando intere piattaforme hardware, ma ciò comporta un notevole carico in termini di prestazioni e risorse. Al contrario, le macchine virtuali di processo e le soluzioni basate su container forniscono una virtualizzazione leggera e modulare, ma si basano tipicamente su meccanismi software di intercettazione delle system call che possono introdurre degradazioni di prestazione e meno garanzie di sicurezza. Questa tesi presenta una nuova architettura che colma il divario tra questi due paradigmi applicando la virtualizzazione hardware alle macchine virtuali di processo. Ispirandosi al modello VUOS, il sistema proposto esegue applicazioni Linux non modificate all'interno di una macchina virtuale minimale basata su KVM, con un hypervisor in spazio utente responsabile dell'intercettazione e della gestione delle system call. Eliminando il bisogno di un kernel nel sistema ospite e sfruttando le funzionalità di virtualizzazione hardware, questo approccio permette di ottenere sandboxing efficiente e sicuro a livello di processo, consentendo inoltre la modifica dinamica della visione di sistema dell'applicazione.

Abstract

Virtualization technologies have transformed the way modern computing systems are designed, managed, and secured. Traditionally, full system virtual machines offer strong isolation by emulating entire hardware platforms, but this comes with significant performance and resource overhead. In contrast, process-level virtual machines and container-based approaches provide lightweight and modular virtualization, but typically rely on software-based system call interception mechanisms that can introduce performance bottlenecks and weaker security guarantees. This thesis presents a novel architecture that bridges the gap between these two paradigms by applying hardware-assisted virtualization to process-level virtual machines. Inspired by the VUOS model, the proposed system runs unmodified Linux applications inside a minimal KVM-based virtual machine, with a userspace hypervisor responsible for intercepting and managing system calls. By omitting the guest operating system kernel and leveraging hardware virtualization features, this approach enables efficient and secure process-level sandboxing while allowing dynamic modification of the application's system view.

Table of contents

Introduction

In modern computing, virtualization and containerization are key technologies that have changed how software is developed, deployed, and managed. They are used everywhere, from large cloud data centers to individual developer workstations. Virtual machines and containers offer practical methods for abstracting hardware resources, isolating system environments, and making software portable. Specifically, they enable multiple operating systems to run on a single physical machine, allow applications to be bundled with their dependencies into portable units, and provide secure sandboxes for running untrusted code.

As technology needs have changed, the industry has developed more lightweight and efficient forms of virtualization. Traditional full-system virtual machines provide strong isolation by emulating entire hardware systems, but they also bring significant performance overhead and require substantial resources. In response, operating system-level virtualization; commonly called containerization, has become popular. Containers, such as those managed by Docker and LXC, share the host system's kernel while isolating user-space environments. This approach offers faster startup times and lower resource usage, meeting the demand for powerful yet efficient virtualization solutions.

However, the current virtualization landscape presents a trade-off. On one side, full system virtualization, powered by hardware-assisted technologies like KVM, offers high security and performance by leveraging CPU features such as Intel VT-x and AMD-V. However, this power is exclusively applied to emulating entire machines, making it a heavy and monolithic solution. On the other side, process-level virtualization offers a more granular and flexible approach. A leading example is the VUOS model, which can alter a single process's view of the system by intercepting its system calls. While flexible, these systems typically rely on

software-based interception mechanisms, such as `ptrace()` or `seccomp-bpf`, which can introduce performance bottlenecks and offer a weaker isolation boundary compared to hardware-based methods.

This thesis introduces a new approach that aims to combine the strengths of both paradigms. The main objective is to apply KVM's hardware virtualization, traditionally used for full virtual machines to process-level and partial virtualization. By doing so, it becomes possible to use the speed and security of hardware-based interception together with the lightweight and modular properties of process-level virtualization. The goal is to create efficient and secure virtualization environment for individual processes, without the high overhead of running a complete guest operating system.

This work explores the design and implementation of a new architecture that repurposes KVM to operate on single processes rather than entire systems. The aim is to demonstrate that hardware virtualization is not tied to full system virtualization but can be a powerful tool for process-level virtualization, unlocking new possibilities for secure sandboxing, system call customization, and dynamic process environment modification.

1.1. Virtualization

Abstraction is a core concept in computer science that helps manage system complexity. By defining interfaces, abstraction hides low-level implementation details and enables different hardware and software layers to work together. For example, an operating system abstracts the complexity of hardware, exposing a simple API that developers use to build applications without worrying about memory management or device control.

However, abstraction has its limitations. Software and systems built on a specific abstraction layer are often tightly related to the underlying hardware or operating system. This can make it difficult to move software between different platforms or architectures; for instance, programs compiled for x86-64 cannot run on ARM machines due to differences in instruction sets and system interfaces.

Virtualization addresses these limitations by introducing an extra layer that separates software from hardware. Through virtualization, physical resources such as processors, memory, and devices are mapped into flexible, programmable virtual

resources. This allows one physical system to act as multiple independent virtual systems, or makes virtual systems appear consistent across different hardware platforms.

A classic example is disk virtualization, the operating system presents a filesystem abstraction, allowing users to work with files instead of raw disk sectors. Virtualization can build on this by creating multiple virtual disks, each appearing as a separate physical disk, but actually implemented as files on the underlying filesystem. Here, virtualization does not simply hide more details; it allows us to overcome hardware and software limitations by providing new, flexible interfaces.

In computer science, "virtual" refers to something that does not physically exist but appears real. A virtual machine is a software layer that creates a form of virtualization, providing an environment capable of running other software. Virtual machines are widely used in programming language runtimes (like the Java Virtual Machine), web hosting, and multi-OS servers.

In summary, virtualization extends abstraction by enabling interoperability and flexibility, overcoming the constraints imposed by hardware and software interfaces. It can add new layers or redefine existing ones, allowing for creating solutions to complex system design problems.

1.2. Virtual machines classification

Nowadays, there are many different types of virtual machines to satisfy a vast range of needs: programming languages (e.g., Java), multi-operating system servers, and sandboxing.

To better understand the types of virtual machines, this section discusses two virtual machine classification systems: one from Smith and Nair, focused on the user point of view, and another from the Virtual Square project, which focuses on technical features.

1.2.1. Smith & Nair Taxonomy

According to the Smith and Nair taxonomy [1], a machine can be understood in two ways. From the process point of view, a machine refers to the logical memory space, the set of CPU instructions and registers, and the interface between the operating system and hardware known as the Application Binary Interface (ABI).

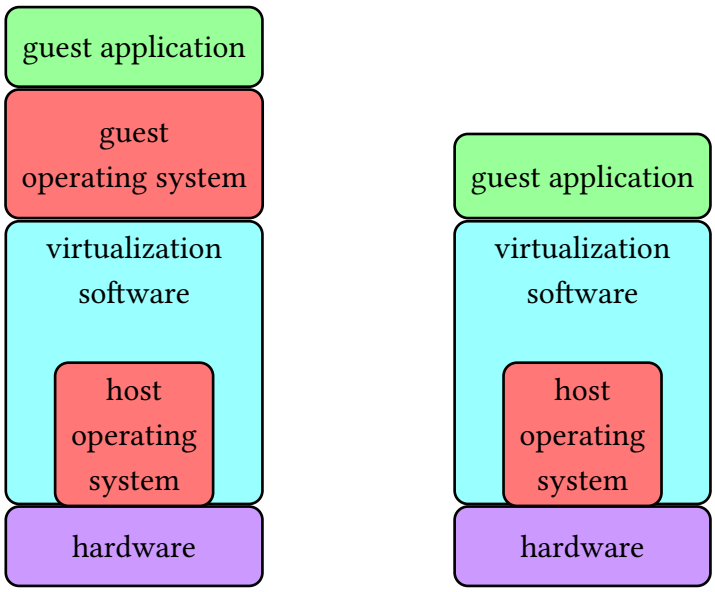

Figure 1: System Virtual Machine vs. Process Virtual Machine

In contrast, the system perspective treats the hardware as the machine, interacting through the Instruction Set Architecture (ISA). This distinction leads to two types of virtual machines. The process virtual machine provides an environment for running a single program, while the system virtual machine recreates the entire environment needed for an operating system and all its applications. Furthermore, Smith and Nair note that virtualization can either strictly imitate the real hardware to offer features such as isolation and security, or it can allow applications written for different architectures to run on the same hardware, improving compatibility and portability.

1.2.2. Virtual Square Taxonomy

The Virtual Square projects [2] focus on developing and collecting virtualization technologies for computer systems, operating systems, and networks; for this reason, this taxonomy provides general categories that are not limited to virtual machines.

1. *Host System Intrusiveness.* The VM can operate at different privilege levels on the host system:

   User-level access Virtualization runs as a regular user program, providing strong isolation but limited operations and potential performance issues. The reliability of the host is not compromised.

Superuser Access Requires superuser privileges, balancing intrusiveness and performance. However, it introduces security risks if used by untrusted users.

Kernel Patch or Module Involves modifying the operating system kernel, significantly improving efficiency but increasing intrusiveness and security risks. Administrative privileges are required.

Native Mode The monitor is integrated as the OS kernel or as a driver for the virtualized component.

2. *Consistency to the Lower Layer Interface.* VM may provide either the same interface as the underlying environment (Homogeneous Virtualization) or a different one (Heterogeneous Virtualization):

   Homogeneous Virtualization Offers the same environment interface, allowing services like access control and multiple virtual instances. It can be partial (affecting only some functions) and modular (combining different virtualizations)

   Heterogeneous Virtualization Provides an interface different from the underlying environment

3. *Paravirtualization.* Paravirtualization is an interface optimization for virtualization. Instead of full compatibility, it offers a similar but more efficient interface, requiring custom client software. This approach eliminates unnecessary features that are not relevant to the virtual environment, thereby improving performance.

1.3. State of the art

There are many virtualization technologies available today, each with its own unique characteristics and implementation details. This section focuses on the most widely used open source solutions in each virtualization category, highlighting how they differ in terms of architecture, functionality, and technical approach. To organize this overview, the two above-mentioned classification methods are used with the aim of not only classifying them according to their use cases but also the virtualization technology they use and how they are implemented.

1.3.1. Emulators/Heterogeneous virtual machines: QEMU

QEMU (Quick Emulator) [3] is a widely used, open-source virtualization platform that supports full system emulation. It is capable of emulating all major CPU architectures, like i386/x86-64, ARM, and RISC-V, which allows it to run various operating systems and virtual environments on a single host machine. This flexibility makes QEMU one of the most popular and adaptable virtualization solutions available.

QEMU can operate in different modes:

1. **User Mode emulation**: Allows individual Linux applications compiled for a different instruction set architecture (ISA) to run on the host system. This is achieved by translating system calls and dynamically translating instructions from the target architecture to the host architecture.
2. **System Emulation** (Figure 2): Emulates an entire computer system, including the CPU and peripherals, allowing the execution of full operating systems for different architectures on the same machine.
3. **KVM mode**: When used together with the Kernel-based Virtual Machine (KVM) module for Linux, QEMU can leverage hardware virtualization features of modern CPUs to run guest operating systems at near native speed, provided that the guest and host architectures match.

QEMU uses dynamic binary translation to achieve efficient emulation. The first time QEMU encounters an instruction from the guest, it translates it into the host's ISA and caches the result. Subsequent executions of the same instruction use the cached translation, improving performance and reducing overhead.

Following the Virtual Square taxonomy, QEMU has the lowest level of intrusiveness and user-privileged execution possible, and provides heterogeneous virtualization.

Following the Smith & Nair taxonomy, QEMU can be a process VM or a system VM depending on what operating mode is used, and can make conversions between different ISAs so that it can run code from other architectures.

Over the years, QEMU has become a crucial tool for system and kernel developers, as it allows them to test code for different architectures or kernel versions without needing to reboot or use multiple physical machines. QEMU can also be integrated with debugging tools like GDB, enabling detailed observation and troubleshooting of virtualized systems.

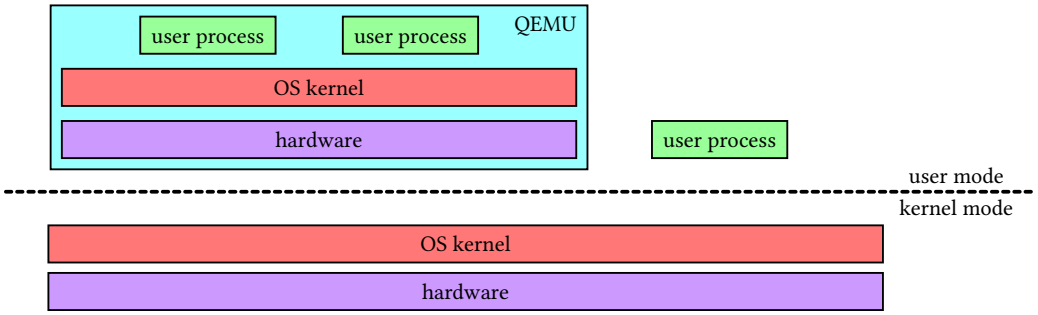

Figure 2: QEMU system mode emulation architecture

In summary, QEMU's ability to emulate diverse architectures, combined with its support for hardware virtualization through KVM, makes it an essential component for both developers and users who need to run multiple operating systems or test software across different platforms.

1.3.2. Homogeneous virtual machines: KVM

Kernel-based Virtual Machine (KVM) [4] is a virtualization infrastructure for the Linux kernel that leverages hardware-assisted virtualization features provided by modern processors, such as Intel VT-x and AMD-V. Unlike traditional hypervisors, which often require building significant portions of an operating system kernel from scratch, KVM transforms the Linux kernel itself into a virtual machine monitor.

KVM architecture relies on both kernel and userspace components. In the kernel, the KVM module exposes its functionality through a character device, `/dev/kvm`. User space programs, most notably QEMU, interact with this device to create and manage virtual machines. While KVM provides the core virtualization capabilities, QEMU is responsible for emulating hardware devices and managing VM I/O in user space. This separation allows KVM to focus on efficient execution of guest code, while QEMU handles device emulation.

Every KVM virtual machine is implemented as a standard Linux process, scheduled by the normal Linux scheduler. In addition to the traditional user and kernel modes of execution, KVM introduces a "guest mode". When a process is running in guest mode, it executes code of the virtualized operating system, with its own kernel and userspace. From the host system's perspective, KVM virtual machines appear as ordinary processes, and standard Linux tools (such as `ps` and `kill`) can be used to inspect or terminate them.

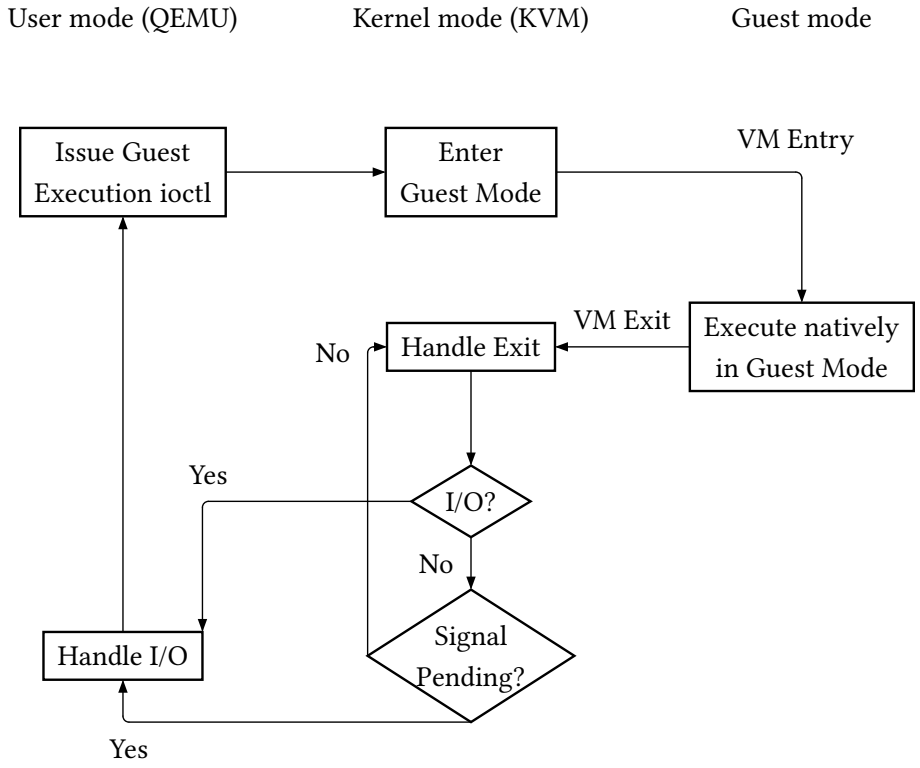

Figure 3: Guest Execution Loop

KVM's execution model can be described as a nested loop (Figure 3):

1. **Userspace Level**: The user space component requests the kernel to run guest code until an event (such as I/O or an external interrupt) requires intervention.
2. **Kernel Level**: The kernel uses the hardware virtualization features to enter guest mode. When an exit occurs (for example, due to a privileged instruction or hardware event), the kernel handles it or passes control back to user space.
3. **Hardware Level**: The CPU executes guest code directly and efficiently, trapping only when assistance is needed.

This design allows KVM to provide high performance and strong isolation between guests, making it a robust solution for server virtualization and cloud computing environments.

1.3.3. Operating System level Virtualization

Operating System-Level Virtualization is a virtualization method where the kernel of an operating system creates multiple isolated user space instances. Instead of virtualizing hardware, the kernel provides an abstraction by partitioning its resources into separate environments known as containers, zones, or partitions.

From a user's perspective, each of these containers appears as an independent computer, even though they all share the same underlying kernel.

1.3.3.1. LinuX Containers (LXC)

This technology leverages two key features of the Linux kernel: **namespaces** [5] and **control groups (cgroups)** [6]. Linux namespaces are a fundamental feature of the Linux kernel that isolates and virtualizes system resources. A namespace wraps a global system resource in an abstraction layer, making it appear to the processes within that namespace that they have their own dedicated instance of that resource. Any modifications to the resource are contained within the namespace, visible to other processes within the same namespace but completely invisible to processes outside of it.

By using these namespaces, LXC can create environments that are almost entirely isolated from one another, effectively creating a "sandbox" for processes. This functionality is similar in concept to the chroot command, which limits a process's view of the filesystem to a specific directory subtree. However, namespaces provide a much more comprehensive level of isolation that extends to PIDs, networking, user IDs, and more.

While namespaces provide isolation, control groups (cgroups) are used for resource management. They allow for the limitation and monitoring of resources like CPU time, system memory, and network bandwidth for a collection of processes.

By combining namespaces for isolation and cgroups for resource limiting, LXC provides a lightweight framework for creating system containers that behave much like full virtual machines but without the overhead of emulating hardware and running a separate kernel.

The creation and management of namespaces and cgroups traditionally require superuser privileges `CAP_SYS_ADMIN`, making LXC primarily a tool for system administration.

1.3.4. Process level virtual machines

A process-level virtual machine is a form of virtualization where the virtual machine monitor (VMM) directly interfaces with user processes. In this approach, service requests from processes are managed by the monitor itself, which provides a high-level interface for resource management. This is distinct from system-level virtualization, where the VMM emulates hardware to support the execution of complete operating systems.

1.3.4.1. User Mode Linux (UML)

User-Mode Linux [7] is a specially compiled Linux kernel designed to run as a standard user space process on a host Linux system.

The core principle of UML is to enable the execution of a complete Linux kernel environment within a user process. Consequently, every process running inside a UML instance is also a process, or more precisely, a thread on the host machine. This architecture is often utilized for kernel development and debugging, creating isolated environments for testing potentially harmful software, and deploying multiple distinct Linux environments on a single host without the overhead of full hardware virtualization.

The mechanism for virtualizing the system is centered on system call interception. A dedicated tracing thread within the UML monitor uses `ptrace()` [8] to intercept all system calls made by the processes running inside the virtual machine. When a virtualized process issues a system call, the tracing thread intercepts it, nullifies its effect on the host system, and redirects the process into the UML kernel. Once inside the UML kernel, tracing is disabled, allowing the UML kernel itself to make standard system calls to the host kernel. This effectively creates two modes: a "user mode" where process calls are virtualized, and a "kernel mode" where the UML kernel has privileged access to the host's services, mirroring the hardware-based privilege levels.

Beyond system call interception, User-Mode Linux (UML) virtualizes several other critical system components by leveraging host operating system mechanisms:

Signals and Traps Managed using the same `ptrace()` interception mechanism as system calls, allowing the UML kernel to handle them internally.

Devices and Timers Implemented using host signals. Device I/O is handled via `SIGIO`, which UML translates into a virtual IRQ, while clock interrupts are simulated using `SIGALRM`

Memory Faults The `SIGSEGV` signal is intercepted by UML, which then distinguishes between an actual illegal memory access, treated by forwarding the signal, and a standard page fault, handled with its own routines

Virtual Memory Physical RAM is simulated by a large file on the host system. UML's memory manager maps the virtual pages of its processes into this file

Host Filesystem Access Provided through a virtual filesystem called `hostfs`, which translates file operations from within UML into system calls on the host

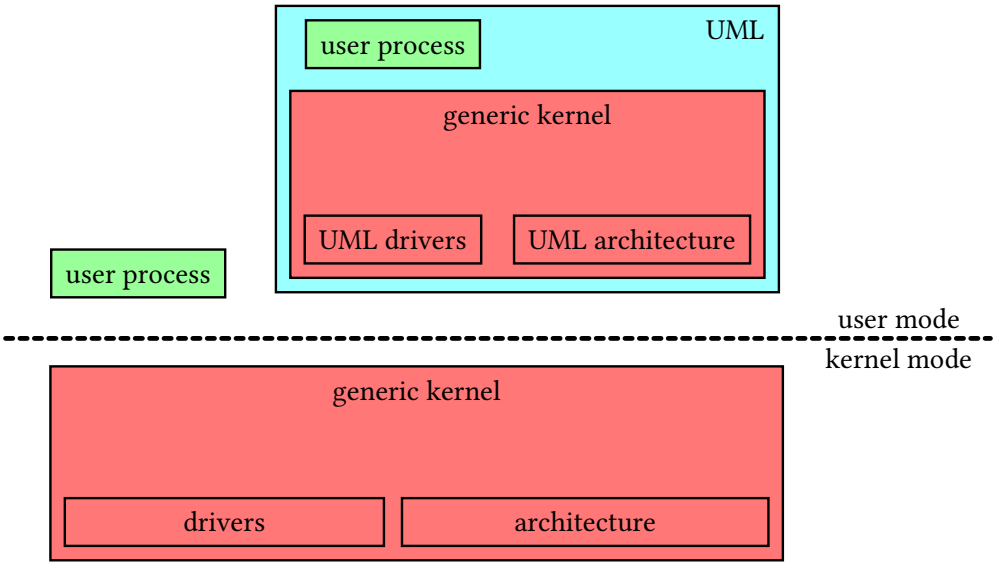

Figure 4: UML architecture

According to the taxonomy proposed by Smith and Nair, UML is classified as a process-level virtual machine with the same Instruction Set Architecture as the host. This signifies that UML does not perform binary translation; instead, it directly executes programs compiled for the host's architecture.

Following the Virtual Square taxonomy, UML is defined as a user-level solution, as it operates entirely as a standard process within the host operating system's user space. Because it exposes the same operating system interface as the host, it is further classified as a case of homogeneous virtualization.

1.3.4.2. gVisor

gVisor [9] is a technology developed to provide strong isolation for containerized applications by implementing a process-level virtual machine. It functions as an application kernel, written in the memory-safe language Go, which runs in userspace and provides a Linux-compatible interface to the sandboxed application. This approach allows gVisor to create a robust security boundary between the container and the host operating system, significantly reducing the attack surface of the host kernel. For integration with existing container ecosystems, gVisor includes an Open Container Initiative (OCI) [10] runtime named `runsc`, which allows it to be used seamlessly with container orchestration tools like Docker and Kubernetes.

GVisor achieves virtualization through a multiprocess architecture where each sandbox environment is composed of several key components.

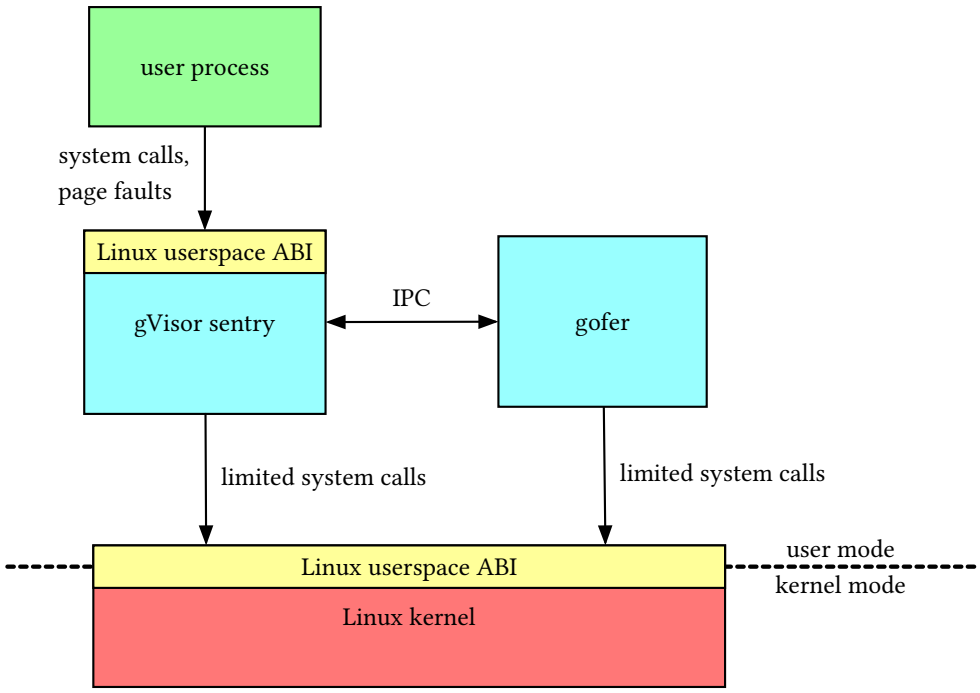

Figure 5: gVisor Systrap platform architecture

The Sentry This is the central component of gVisor, acting as the application kernel for the sandboxed container. The Sentry is responsible for implementing the Linux system call interface, managing memory, handling signals, and managing the process and threading model. When an application inside the sandbox makes a system call, the call is intercepted and handled entirely by the Sentry. Crucially, the Sentry does not pass the application's system calls directly to the host kernel. Instead, it performs the necessary operations itself, making its own limited set of system calls to the host only when essential, for example, to manage memory or threads.

The Gofer To further enhance isolation, the Sentry operates in a highly restricted environment with no direct access to the host's filesystem. All filesystem operations are delegated to a dedicated process called the Gofer. The Gofer is a standard host process that communicates with the Sentry over a 9P protocol channel [11]. It acts as a broker for filesystem access, mediating all requests from the sandbox and enforcing an additional layer of security.

The Application is a standard Linux binary, packaged within an OCI runtime bundle

gVisor employs different platforms to intercept system calls and page faults from the sandboxed application. The two primary platforms are Systrap and KVM.

KVM Platform This new and experimental platform leverages the Linux KVM subsystem. In this mode, the Sentry acts as both the guest operating system and the Virtual Machine Monitor. While it does not virtualize a full hardware set, it uses the virtualization extensions available in modern processors to enforce address space isolation and handle page faults. This provides strong security guarantees and performs best on bare-metal hosts where direct access to virtualization hardware is available.

Systrap Platform This is the default platform. It is based on the `seccomp-bpf` feature of the Linux kernel [12]. The platform configures seccomp to trap every system call made by the application. When a system call occurs, the kernel sends a `SIGSYS` signal to the process, which pauses its execution and transfers control to the Sentry. The Sentry then inspects the trapped process's state, emulates the system call, and resumes the process. The Systrap platform does not require hardware virtualization support, making it well-suited for environments where KVM is unavailable or nested, such as in cloud-based virtual machines.

gVisor provides enhanced security for containers by using a userspace application kernel, which significantly reduces the host kernel's attack surface. The main disadvantages are the performance overhead from system call interception and potential application incompatibilities due to an incomplete reimplementation of the Linux API. Therefore, gVisor is best suited for scenarios where security is a priority, such as running untrusted code and serverless platforms.

1.3.5. Process level partial virtualization

In a traditional computer system, the concept of a global view is fundamental: every process shares the same uniform vision of the system resources, including the network, devices, and file system, where it is running. However, a specific category of applications exists that alters this paradigm. While not strictly virtual machines in the classic sense, these systems introduce a form of virtualization at the process level.

1.3.5.1. Fakeroot

Fakeroot [13] and its successor, fakeroot-ng [14], are examples of process-level virtualization focused on the file system. Fakeroot is an application that provides a regular user with the perception of having administrative (root) privileges for file operations. By using fakeroot, a user can create and manage files that appear

to be owned by the root user, with specific ownership and permission settings. This functionality is achieved by intercepting file system related library calls, in the Fakeroot case, or by intercepting system calls related to file metadata in the new implementation, fakeroot-ng, to create a virtual, emulated view of ownership rights and permissions. The underlying file system is not actually modified; instead, the process running under fakeroot is presented with a modified perspective of file metadata.

1.3.5.2. Virtual Distributed Ethernet (VDE)

Virtual Distributed Ethernet (VDE) [15] is a tool that creates and manages virtual networks, allowing users to link computers across different locations on the Internet, design arbitrary network topologies, and create virtual LANs to connect them. VDE's interface with the real network is realized through the TUN/TAP virtual network drivers available in the Linux and macOS kernels. It is designed to be compatible with a wide range of virtualization solutions, including QEMU, KVM, User-Mode Linux (UML), VirtualBox, and Bochs.

The fundamental idea behind VDE is to allow users to build and manage complex virtual network topologies, independent of the restrictions imposed by the physical network infrastructure. VDE can create virtual switches, hubs, cables that behave much like a physical one and present an interface that is identical to a real network interface, both to the host system and to the virtual machines connected to it. The tunnel used to transport data-link-layer frames can rely on any streaming protocol. For example, an SSH connection can be used as the transport layer, effectively creating an encrypted Virtual Private Network (VPN) with VDE.

1.3.5.3. VUOS

A leading example of this approach is VUOS [16]. The core idea behind a VUOS is to allow a user to change the point of view of their processes. This is achieved by redirecting every system call to a hypervisor. The running process does not have immediate access to the services provided by the kernel; instead, each system call request is first intercepted and analyzed by the hypervisor. Based on the specific system call and its parameters, the hypervisor decides how to proceed.

If the hypervisor determines that the system call refers to an unmodified part of the process's view, it simply forwards the request to the underlying level for execution. This could mean passing the call to the real kernel or to a lower-level VUOS instance, as the architecture naturally supports nesting. From the hypervisor's point of view, both cases are handled identically.

In the other case, the hypervisor can choose to take actions and implement a change in the process's view. This may involve executing an existing system call

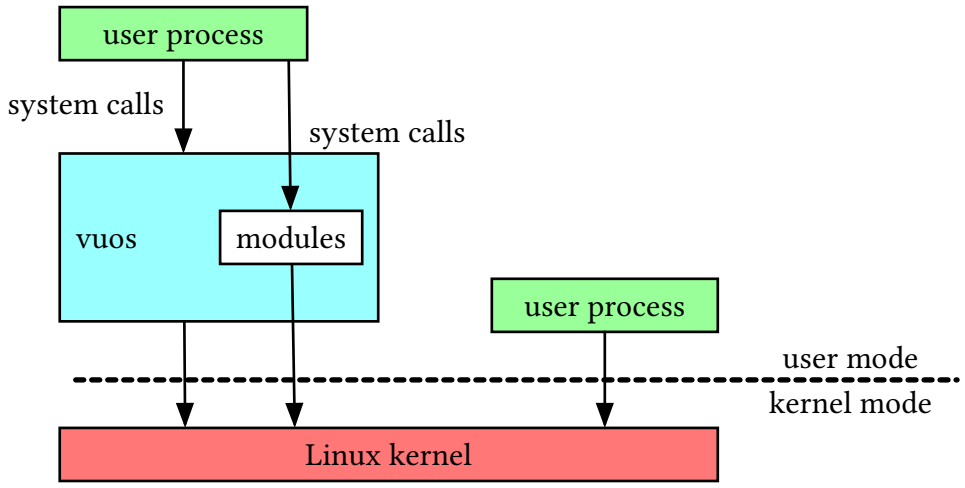

Figure 6: VUOS architecture

with modified semantics or even implementing a new system call not supported by the underlying kernel.

Umvu is the current VUOS hypervisor implementation developed as part of the Virtual Square project [17]. It functions as a user-level hypervisor designed to intercept system calls and modify their behavior according to the specific "view" of the calling process. Implemented as a system call virtual machine, umvu can dynamically load modules that alter the perspective of running processes.

A user-level implementation of the VUOS model, like umvu, is inherently slower than a potential kernel-level counterpart. However, this approach offers several significant advantages:

1. **No Administrative Privileges**: A user-level hypervisor does not require administrative (root) permissions to operate.
2. **Kernel Independence**: It does not depend on any specific features or support from the underlying Linux kernel.
3. **Simplified Development and Debugging**: Debugging a userspace application is considerably simpler and safer than debugging a kernel module.
4. **Kernel Integrity**: The host system's kernel code remains entirely unmodified, enhancing system stability and security.

The modularity of umvu is central to its design. It relies on dynamically loaded modules to redefine system call behavior, offering high flexibility. Developers can implement anything from simple modules that create a new virtual file system to complex modules that emulate a userspace kernel, making umvu a versatile solution for a wide range of virtualization-related tasks.

For its core task of system call interception, umvu can utilize two methods: the `ptrace()` system call, like UML, or the new faster `seccomp-bpf` method, like gVisor. Those methods achieve interception without requiring custom kernel features or loading a separate kernel module.

1.4. Objective

The objective of this thesis is to design and implement a prototype of a process-level virtual machine that operates as a userspace application using hardware virtualization through KVM. Unlike traditional KVM-based virtual machines, which typically run a complete guest operating system and kernel, the proposed prototype omits the guest kernel entirely; instead, it runs only the target application within the virtual machine. The userspace hypervisor functions as a minimal kernel, intercepting and managing system calls originating from the guest application.

This architecture draws inspiration from the VUOS model, which redefines the process's system view by rerouting system calls through a hypervisor. In the prototype, system call interception is achieved using hardware virtualization features, enabling the userspace hypervisor to inspect, modify, and execute system calls made by the guest application with the added benefits of hardware-assisted isolation and performance.

The main goal is to demonstrate the feasibility of this approach by running a subset of static and dynamic unmodified Linux ELF binaries inside the KVM virtual machine. The prototype will showcase how hardware virtualization can be adapted for process-level virtualization, enabling modification of the process's system view and providing secure sandboxing without the overhead of a full guest operating system. Furthermore, the thesis aims to discuss potential applications of this technology and to evaluate its performance compared to existing solutions.

Implementation

This chapter presents the design and implementation of **syskvm** (System Call KVM Virtual Machine) [18], the proof-of-concept process virtual machine developed for this thesis. The chapter shows how the KVM API is used to build and manage these minimal VMs, how it recreates a userspace execution environment inside the virtual machine, and the mechanisms it uses to intercept and handle system calls.

This implementation specifically targets the **x86-64** CPU architecture [19], which was chosen due to its prevalence in modern computing environments. As a result, certain aspects of the design are specific to x86-64. However, the core concepts presented in this thesis are not limited to a single architecture. This approach could be adapted to other architectures that provide similar virtualization extensions, with the necessary modifications to the architecture-specific components.

2.1. The KVM API

This section provides an introduction to KVM programming. The design of the KVM API is described, highlighting its hierarchical structure based on file descriptors and `ioctl` (Input/Output Control) system calls [20]. The chapter then explains how this API is used in practice to create a basic virtual machine used by syskvm, including the allocation of memory and the creation of a single virtual CPU (vcpu).

2.1.1. The API Model

The KVM API [21] is exposed to user space through the special device file `/dev/kvm`. Interaction with the KVM subsystem is performed by opening this file and issuing a series of `ioctl` calls on the resulting file descriptor. The API follows a hierarchical object model, where different file descriptors represent distinct components of the virtualized environment.

System ioctls Issued on the file descriptor obtained from `/dev/kvm`, these operations are used to query and configure global KVM attributes and to create new VMs via the `KVM_CREATE_VM` ioctl.

VM ioctls Once a VM is created, it is represented by a dedicated file descriptor. `ioctl`s sent to this VM fd manage the state of the entire virtual machine, such as defining guest physical memory, creating vCPUs, and managing virtual devices.

vcpu ioctls Each virtual CPU within a VM also has its own file descriptor, created with the `KVM_CREATE_VCPU` ioctl on a VM fd. These operations control the execution of a specific vCPU, including setting its registers, injecting interrupts, and running guest code. The primary operation on a vCPU fd is `KVM_RUN`, which starts or resumes guest execution.

By default, the `/dev/kvm` device on a Linux system is accessible only to the root user, as it provides direct access to hardware virtualization features. However, it is generally considered safe to grant access to this device to non-root users. For example, in the Debian distribution, installing the `qemu-system` package adjusts the permissions on `/dev/kvm` to allow broader user access, and some distributions even enable user-level access by default. The appropriate permission model is still a topic of discussion in the Linux community, but for this work, it is assumed that allowing normal users to access `/dev/kvm` is acceptable. This makes the presented technology feasible as a user-level, rather than a privileged, virtualization solution.

2.1.2. Building a Minimal Virtual Machine

The practical use of the KVM API is illustrated by showing the process of building a minimal VM, focusing on the essential steps involved in guest execution and handling VM exits.

1. **Accessing the KVM Subsystem**: The KVM device file is opened to obtain a system file descriptor.

```c
int kvm = open("/dev/kvm", O_RDWR | O_CLOEXEC);
```

2. **Creating the Virtual Machine**: With the system file descriptor, a VM can be created using the `KVM_CREATE_VM` ioctl. This call returns a new file descriptor that represents the VM instance.

```c
int vmfd = ioctl(kvm, KVM_CREATE_VM, 0);
```

3. **Allocating and Mapping Guest Memory**: A VM requires memory to function. It is the responsibility of the VMM to allocate this memory in its own address space and to instruct the kernel to map it into the guest's physical address space with the `KVM_SET_USER_MEMORY_REGION` ioctl. For this example, a page of memory is allocated.

```c
void* mem = mmap(NULL, 0x1000, PROT_READ | PROT_WRITE,
MAP_SHARED | MAP_ANONYMOUS, -1, 0);

struct kvm_userspace_memory_region region = {
  .slot = 0,
  .guest_phys_addr = 0x1000,
  .memory_size = 0x1000,
  .userspace_addr = mem,
};
ret = ioctl(vmfd, KVM_SET_USER_MEMORY_REGION, &region);
```

4. **Creating the Virtual CPU**: The `KVM_CREATE_VCPU` ioctl, invoked on the VM file descriptor, creates a vCPU and returns its corresponding file descriptor.

```c
int vcpufd = ioctl(vmfd, KVM_CREATE_VCPU, 0);
```

5. **Running the Guest and Handling VM Exits** The guest code is executed by invoking the `KVM_RUN` ioctl on the vCPU file descriptor. This call is blocking; it transfers control to the guest and returns only when a **VM exit** occurs. Typically, the VMM enters a loop, repeatedly calling `KVM_RUN` and handling the exit reasons. Information about each exit is provided in a shared data structure, `struct kvm_run`, which must be mapped into the VMM's address space using `mmap`

```c
// Map the kvm_run structure
size_t mmap_size = ioctl(kvm, KVM_GET_VCPU_MMAP_SIZE, NULL);
struct kvm_run *run = mmap(NULL, mmap_size, PROT_READ |
PROT_WRITE, MAP_SHARED, vcpufd, 0);
```

```
// Main execution loop
while (1) {
  ret = ioctl(vcpufd, KVM_RUN, NULL);

  switch (run->exit_reason) {
    // Handle different exit reasons
}
```

2.2. Recreating the Userspace Environment

When a virtual machine is created using KVM, it initially mimics a physical computer at the moment of power-on: it has uninitialized virtual hardware, no loaded code, and no operating system or firmware present. On a typical physical system, the boot process begins with firmware (BIOS or UEFI), which initializes the hardware and loads a bootloader from disk. The bootloader then loads and starts the operating system kernel, which in turn sets up the system environment and finally launches user programs.

However, in the architecture proposed in this thesis, the goal is to run a userspace application directly inside the VM, without a guest kernel, BIOS/UEFI, or bootloader. This means that all the work usually performed during the system boot process, such as setting up CPU state, memory mappings, and loading program code, must be handled explicitly by the VMM in user space, using KVM ioctls and related mechanisms.

This subchapter introduces the techniques and considerations involved in recreating the Linux userspace execution environment within such a minimal VM. It discusses how the VMM does the work of the firmware, bootloader, and kernel to prepare the virtual CPU and memory so that an unmodified Linux application can be executed as if it were running in a fully initialized operating system environment.

2.2.1. CPU Initialization

An important step of the VMM is to configure the virtual CPU to meet the expectations of a modern 64-bit Linux application. The x86-64 architecture has several operating modes [22, p. 63] (Figure 7), beginning in a 16-bit **real mode** for backward compatibility with older software. To execute usperspace programs, the

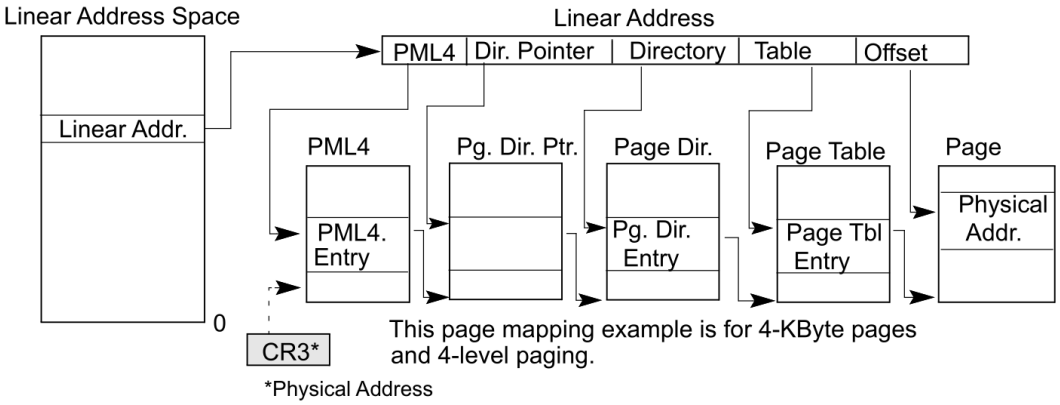

Figure 7: x86-64 4-level paging

CPU must be transitioned into **long mode**, the native 64-bit operating mode. This involves setting up memory management structures and enabling specific CPU features. The VMM performs this entire setup from user space before any guest code is executed.

Activating **long mode** requires the configuration of virtual memory, which is managed through a hierarchical paging structure [22, p. 109]. In this project's implementation, a **demand-paging** [23] approach is used for memory management. Rather than allocating the full page table hierarchy (PML4, PDPT, PD, PT) at startup, the VMM initially allocates only a single page for the top-level page table, the PML4. Alongside this, it sets up a Global Descriptor Table (GDT) to define the necessary code and data segments for 64-bit operation.

With this structure in place, the VMM can program the vCPU's control registers to trigger the switch to long mode. The steps are:

1. Enable Physical Address Extension (PAE) by setting the `PAE` bit in the `CR4` register.
2. Load the physical address of the top-level page table (PML4) into the `CR3` register.
3. Enable long mode by setting the `LME` (Long Mode Enable) bit in the Extended Feature Enable Register (`EFER`).
4. Enable paging by setting the `PG` and `PE` (Protection Enable) bits in the `CR0` register.

Once these steps are completed, the CPU enters 64-bit long mode. The VMM also initializes the segment registers (`CS`, `DS`, `SS`, etc.) to use the newly created GDT entries.

Simply entering long mode is not sufficient for running many modern applications. The C standard library (glibc) and many high-performance applications rely on

CPU features beyond the baseline x86-64 specification, particularly SIMD (Single Instruction, Multiple Data) extensions for vectorized computation.

The x86-64 architecture defines a set of microarchitecture levels [24] (v1, v2, v3, v4) that standardize these features:

x86-64-v1 The baseline, including SSE and SSE2.

x86-64-v2 Adds SSE3, SSSE3, SSE4.1, SSE4.2, and POPCNT, commonly required by modern toolchains.

x86-64-v3 Adds AVX, AVX2, and other features for more advanced vector processing.

x86-64-v4 Adds AVX-512 foundation instructions for high-performance computing.

The VMM must detect the capabilities of the host CPU and enable the corresponding features for the vCPU. This is accomplished by querying the vCPU's supported features via `KVM_GET_CPUID2` and then configuring the appropriate control registers.

The VMM creates this entire initialization process using a sequence of KVM `ioctl` calls. The general pattern is first to retrieve the vCPU's current state, modify it in userspace, and then write the updated state back to the vCPU. The primary `ioctl`s used for this are:

- `KVM_GET_SREGS2` / `KVM_SET_SREGS2`: To read and write special registers, including control registers (`CR0`, `CR3`, `CR4`), segment registers, and tables (`GDT`, `IDT`).
- `KVM_GET_REGS` / `KVM_SET_REGS`: For general-purpose registers.
- `KVM_GET_FPU` / `KVM_SET_FPU`: For the floating-point unit and SSE state.
- `KVM_GET_XCRS` / `KVM_SET_XCRS`: For extended control registers, which are essential for enabling features like AVX.

The overall process can be summarized by the following pseudo-code:

```c
// 1. Get the current state of all relevant register sets
ioctl(vcpufd, KVM_GET_SREGS2, &sregs);
ioctl(vcpufd, KVM_GET_FPU, &fpu);
ioctl(vcpufd, KVM_GET_XCRS, &xcrs);

// 2. Modify the state in userspace to prepare for long mode
//    - Set up GDT, IDT, and page tables
//    - Configure CR0, CR3, CR4, and EFER
cpu_init_long(&sregs, vm->memory);
```

```
// 3. Enable CPU features based on microarchitecture level
//    - Check CPUID results
//    - Configure XCR0 and other registers for SSE/AVX
cpu_init_vN(&sregs, &fpu, &xcrs);

// 4. Write the new, fully-initialized state back to the vCPU
ioctl(vcpufd, KVM_SET_SREGS2, &sregs);
ioctl(vcpufd, KVM_SET_FPU, &fpu);
ioctl(vcpufd, KVM_SET_XCRS, &xcrs);
```

This approach allows the VMM to construct a complete execution environment for a 64-bit application without needing a guest kernel or firmware, using only the controls provided by the KVM API.

2.2.2. Program Loading

Once the virtual CPU is initialized and the virtual machine environment is ready, the next step is to load the user program that will run inside the VM. On Linux systems, executable programs use the ELF (Executable and Linkable Format) file format [25]. ELF binaries can be either statically linked, where all code and data are included in a single file, or dynamically linked, relying on external shared libraries with symbol resolution at runtime.

When a new process is started in Linux, the kernel takes responsibility for reading the ELF binary and mapping its segments, such as code and data, into the process's address space at the correct virtual addresses. The kernel also prepares the initial process stack, placing the program's arguments, environment variables, and an auxiliary vector (`auxv`) containing system-specific information. Among the regions, the kernel map is the VDSO (Virtual Dynamically-linked Shared Object) [26]: a small shared object provided by the kernel that exposes certain system calls and functions directly to userspace, allowing them to be executed without the overhead of a kernel transition.

If the program is statically linked, the kernel can transfer control directly to its entry point. However, most modern programs are dynamically linked, meaning they depend on external shared libraries. In this case, the kernel loads the dynamic linker (`ld-linux.so`) [27], which is itself an ELF binary, into the process memory. On startup, the dynamic linker resolves the required symbols, loads any necessary shared libraries, and only then jumps to the actual program entry point.

Implementing a complete Linux-compatible program loader and memory manager for the VM is a complex and out-of-scope task. In particular, emulating the VDSO and properly managing Linux virtual memory areas would require reimplementing substantial parts of the Linux kernel.

To address this, the prototype developed in this thesis uses a technique that "borrows" the memory layout built by the host Linux kernel when it loads a program. Instead of writing a full ELF loader and dynamic linker in the VMM, the host kernel is used to create a reference process that sets up the program's memory exactly as Linux would. To do this, the VMM creates a helper or "reference" process, here called the **mmlayout** process, by forking and then executing the target program under `ptrace`. This child process is stopped immediately after loading, before any user code is executed. This guarantees that the process memory layout (including ELF loading, all mappings, stack, and auxiliary vectors) is exactly as Linux would provide, but no instructions of the target program have run.

By stopping the process at this stage, the VM Monitor can safely use it as a blueprint for the guest's memory. Memory for the guest is then populated on demand: whenever the guest running inside the VM triggers a page fault (because it tries to access an unmapped page), the VM Monitor simply copies the relevant page from the reference process via the special `/proc/<pid>/mem` file provided

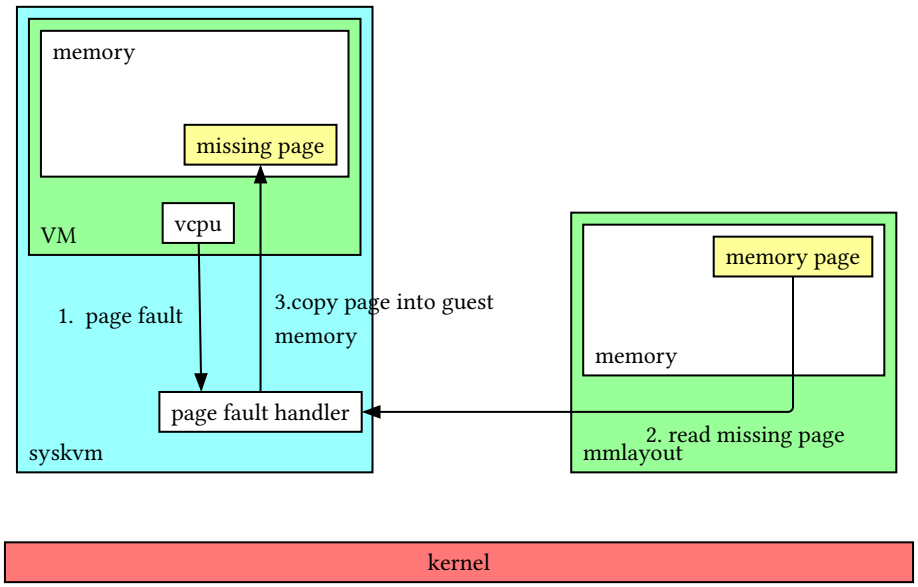

Figure 8: Syskvm page faults resolutions

by Linux. This mechanism allows the VM to "steal" the work already performed by the kernel and dynamic linker, such as mapping all ELF segments and shared libraries, without having to reimplement any of the complex loading logic.

One compatibility issue occurs with the VDSO. Because the VDSO is a region provided by the kernel and not present in the guest VM, any attempt to use it inside the guest would fail. To avoid breaking compatibility with the glibc dynamic linker and related code, the VM Monitor disables the VDSO mapping in the reference process by marking the `AT_SYSINFO_EHDR` entry in the auxiliary vector as invalid. This signals to glibc that it should not attempt to use the VDSO, falling back to traditional system calls instead.

Overall, this technique allows syskvm to recreate the Linux userspace execution environment for arbitrary programs, without the need for a full in-VM implementation of ELF loading and virtual memory management. The VM Monitor copies memory pages on demand from the reference process, using the host kernel's own facilities to provide a compatible execution context for the guest application.

2.3. The Interception Mechanism: Handling Exceptions and System Calls

Operating systems rely on exceptions [22, p. 199] to manage unexpected or special conditions encountered during program execution. Exceptions include events such as page faults, invalid instructions, and division errors. In this implementation, exception handling is not only used for correct execution but is also the main mechanism for intercepting system calls issued by the guest application.

2.3.1. Exceptions Interception

In x86-64 architecture, exceptions are delivered to the processor through the Interrupt Descriptor Table (IDT) [22, p. 207]. Each entry in the IDT corresponds to a specific exception or interrupt and defines how the CPU should handle it.

To intercept all exceptions made by the guest code, the Virtual Machine Monitor initializes the guest's IDT so that all 256 entries are configured as interrupt gates with the Present (P) flag cleared (set to 0). This means that, from the guest's perspective, no exception handlers are implemented. When the guest triggers any exception, the CPU attempts to deliver it but fails due to the absent handler,

causing a **double fault**. Since the **double fault** handler is also not present, a **triple fault** occurs, which results in the virtual CPU exiting to userspace with the `KVM_EXIT_SHUTDOWN` exit reason.

After such a VM exit, the VMM can inspect the `exception.nr` field in the `kvm_vcpu_events` structure, retrievable with the `KVM_GET_VCPU_EVENTS` ioctl, to determine which exception was originally raised. Despite the name, after a `KVM_EXIT_SHUTDOWN`, the VM can resume execution normally by issuing another `KVM_RUN` ioctl. This method allows the VMM to trap and analyze all exception events in the guest efficiently.

2.3.2. System Calls Interception

In x86-64 Linux, system calls [28] can be performed via two main mechanisms: The legacy `int 0x80` software interrupt method works by executing a specific software interrupt instruction `int 0x80`, which causes the CPU to switch from user mode to kernel mode using the interrupt descriptor table, and the modern `syscall` instruction [29, p. 1428], introduced with the x86-64 architecture, provides a much faster way to transition from user mode to kernel mode. When `syscall` is executed, the CPU switches to a predefined kernel entry point, set by special model-specific registers (MSR) such as `IA32_LSTAR`.

Most modern software uses the `syscall` instruction. In this thesis, only the `syscall` method is supported for intercepting system calls from the guest application.

System call interception is achieved by deliberately disabling the `syscall/sysret` instruction pair in the guest. This is done by clearing the SCE (System Call Enable) bit (bit 0) in the `IA32_EFER` Model-Specific Register of the virtual CPU. As a result, when the guest executes a `syscall` instruction, it triggers an Invalid Opcode exception (#UD) because the instruction is not enabled.

This #UD exception is handled using the general exception interception mechanism described above: the absence of a handler leads to a triple fault and a `KVM_EXIT_SHUTDOWN` VM exit. The VMM then determines whether the exception was caused by a `syscall` instruction by inspecting the guest's instruction pointer (`rip`) and verifying that the memory at that address contains the `syscall` opcode (`0F 05`).

If a `syscall` is detected, the VMM reads the guest's register state to identify the system call number (stored in `rax`) and its arguments (`rdi`, `rsi`, `rdx`, etc.). The system call is then handled and executed by the VMM as appropriate. After the system call is processed, the VMM advances the instruction pointer by two

bytes, the length of the `syscall` opcode, to ensure that guest execution resumes at the instruction following the system call. This approach guarantees precise interception of all guest system calls, enabling the VMM to inspect, modify, and execute them as needed.

The overall logic for handling exits within the VMM's main loop can be conceptualized with the following pseudo-code:

```c
// Main VMM execution loop
while (running) {
  ioctl(vcpufd, KVM_RUN, NULL);

  switch (run->exit_reason) {
    // ... other exit reasons ...

    case KVM_EXIT_SHUTDOWN:
      struct kvm_regs regs = vm_get_regs(vm);
      struct kvm_sregs sregs = vm_get_sregs(vm);
      struct kvm_vcpu_events events = vm_get_vcpu_events(vm);

      switch (events.exception.nr) {
        case INVALID_OPCODE_EXCEPTION: // #UD
          if (is_syscall_instruction(guest_memory, regs.rip)) {
            handle_system_call(&regs);

            // Skip the 2-byte syscall instruction
            regs.rip += 2;
          } else {
            // Handle an actual invalid opcode
          }
          break;

        case PAGE_FAULT_EXCEPTION: // #PF
          // Handle demand-paging by copying from reference process
          handle_page_fault(guest_memory, sregs.cr2);
          break;

        // ... handle other exceptions ...
      }
      break;
```

```
  }
}
```

2.4. System Calls Execution

Once a system call is intercepted from the guest application running inside the KVM-based virtual machine, it must be executed. In conventional process-level or partial virtual machines, such as those using `seccomp-bpf` or `ptrace`, for example, VUOS and gVisor, the intercepted system call can normally be delegated back to the host kernel. The hypervisor can inspect, modify, or filter the system call, and then allow the host kernel to execute it on behalf of the virtualized process. This works because the virtualized process remains within the control of the host kernel's resources with direct access to its execution state and memory.

However, in the architecture described in this thesis, this direct delegation is not possible. The guest application runs inside a virtualized KVM VM; therefore, its resources, such as memory and CPU state, are isolated from the host and are not directly visible to the host kernel as a standard process. When the guest application issues a system call, the hypervisor, running outside the VM, cannot directly ask the host kernel to execute that system call as if it came from the guest.

As a result, the hypervisor (syskvm) must take an extra step: it must emulate the system call from the outside. This is achieved by invoking the system call, or an equivalent operation, in its own context, and then copying any changes back into the guest's execution state and memory. For system calls that involve pointers to guest memory, for example, reading from or writing to a buffer, the host kernel expects that pointer to be a valid address within the VMM's own memory space, not the guest's, so syskvm first needs to copy the relevant data from the guest memory into its own address space. After the system call is performed, any modified data must be copied back into the guest. This ensures that the effects of the system call are visible to the virtualized application as expected.

This approach introduces other problems, especially for system calls that have close interactions with the hardware or the kernel's internal state, like memory mapping or setting CPU registers. In these cases, syskvm cannot simply forward the request to the host kernel directly, because doing so would affect only the hypervisor's resources, not those of the guest. To execute them, the syskvm must

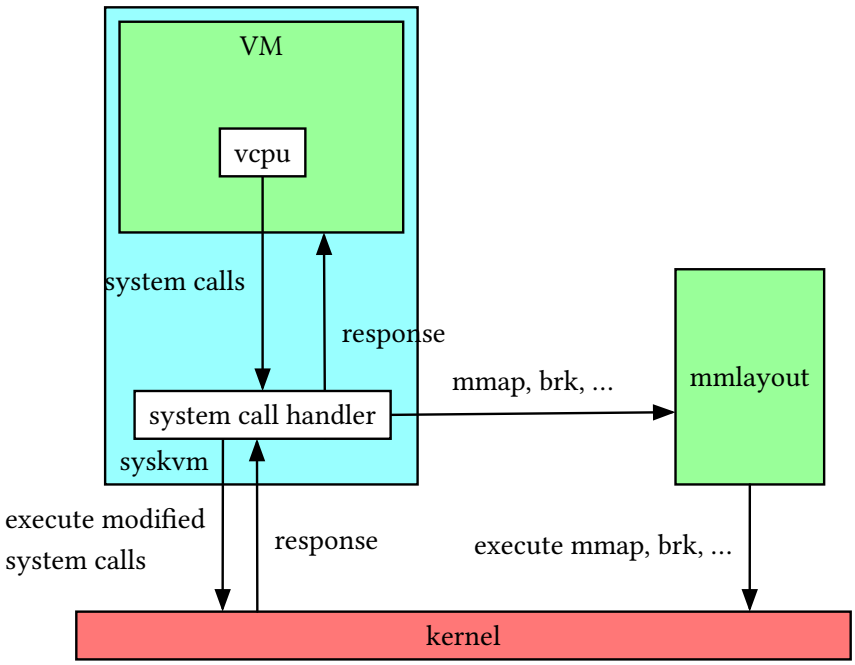

Figure 9: Syskvm System Calls Execution Mechanism

implement or emulate these operations itself. This often requires re-implementing some functionality typically provided by the operating system, such as memory management.

To avoid duplicating large parts of the OS, this implementation reuses the **mmlayout** method discussed above. For memory management system calls such as `mmap` and `brk`, syskvm executes the call within the **mmlayout** reference process using `ptrace`, altering its memory layout. Thanks to the on-demand page copying mechanism, these changes are automatically reflected in the guest's address space whenever the guest faults on a page. While this approach introduces some performance overhead due to the additional `ptrace` calls, it significantly simplifies the implementation and provides a straightforward solution that is Linux-compatible.

2.4.1. Implementing Virtual Views Through System Call Modification

Process-level partial virtualization can be achieved through system call interception, a technique that allows a hypervisor to control and modify the behavior of system calls made by the application. This method is central to the design of **umvu**, and is also adopted in this project.

To demonstrate this mechanism, syskvm implements a virtualized view of specific system files: `/etc/passwd` [30] and `/etc/shadow` [31]. In the guest environment, these files present different contents and permissions than on the host system. This is accomplished by intercepting the `open` system call: when the virtualized application attempts to open either of these files, the VMM does not forward the request to the host kernel. Instead, it creates a new in-memory file using the `memfd_create` system call, writes the desired virtual content to this temporary file, and returns its file descriptor to the guest. All subsequent read and write operations by the application are directed to this virtual file, giving the illusion that it is interacting with the real system files, even if the actual host files remain unchanged.

2.5. Guest Debugging via the GDB Remote Protocol

During the development of this project, the ability to inspect the execution of the virtualized guest program proved helpful in troubleshooting and analysis. Traditional debugging on Linux is accomplished using tools such as GDB [32], which interact with running processes via the kernel's `ptrace` interface. However, in this architecture, the guest program runs inside a KVM-based virtual machine, making it inaccessible to conventional debugging methods.

To provide users with a familiar and powerful debugging environment, the Virtual Machine Monitor implements the GDB Remote Serial Protocol (RSP) [33]. This allows standard GDB clients to connect to the VMM and debug the guest application as if it were running natively.

The implementation uses **mini-gdbstub** [34], a lightweight library that handles the GDB Remote Serial Protocol communication. Mini-gdbstub acts as the server component, translating requests from the GDB client into calls to the VMM, which in turn manipulates the state of the virtualized guest using the KVM API.

The integration process involves implementing a set of callback functions required by the mini-gdbstub library. These functions provide the necessary operations on the guest's state, including reading and writing memory, accessing registers, managing breakpoints, and controlling the execution flow.

The implementation of debugging functionalities relies on the ability to manipulate both the memory and register state of the virtualized guest program. This provides the base for debugging features like memory inspection and register modification.

Handling breakpoints, however, requires further steps. On x86-64 systems, breakpoints can be implemented either as hardware breakpoints, which use the CPU's debug registers but are limited in number, or as software breakpoints. In this project, software breakpoints are chosen because they do not impose practical limits on the number of breakpoints that can be set. The mechanism involves saving the original byte at the target program address and then overwriting it with the special `0xCC` opcode (the `int3` instruction). When the guest program reaches this address during execution, a debug exception is triggered, and the virtual machine exits with the `KVM_EXIT_DEBUG` code. The VMM can then restore the original instruction to remove the breakpoint and resume normal execution.

Single-stepping and breakpoint execution control are managed through the KVM api. By using the `KVM_SET_GUEST_DEBUG` ioctl and configuring the `struct kvm_guest_debug` structure with the appropriate flags, the VMM can enable features such as software breakpoints and single-stepping. For example, enabling `KVM_GUESTDBG_SINGLESTEP` allows the VMM to execute the guest program one instruction at a time, while the `KVM_GUESTDBG_USE_SW_BP` flag activates software breakpoint handling. When any of these events are triggered, KVM notifies the VMM by exiting the guest execution, enabling it to synchronize with the debugger and respond to user commands from the GDB client.

From the user's perspective, interacting with the system through GDB, these mechanisms make debugging the virtualized program feel very similar to debugging a native process. All basic GDB operations, including setting breakpoints, stepping through code, examining memory, and modifying registers, are supported, providing a simple debugging experience for applications running inside the virtual machine.

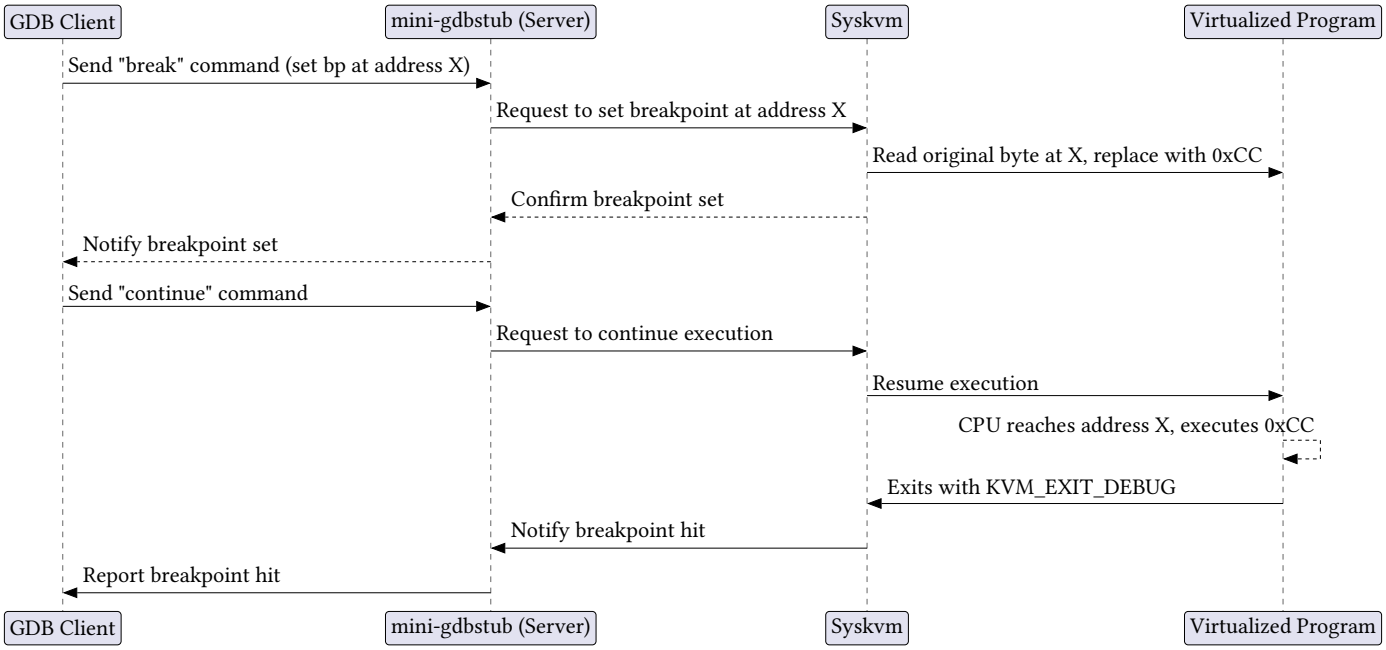

Figure 10: Communication flow when a user sets a breakpoint via GDB

Performance Analysis

This chapter presents a performance comparison of the syskvm prototype with popular process-level virtualization methods (Table 1): **QEMU user-mode emulation** (qemu-x86_64), **VUOS** (umvu in `seccomp-bpf` mode), and **gVisor** in both `seccomp-bpf` and `KVM` modes (`runsc`). The objective is to evaluate the improvements offered by **syskvm**'s new KVM-based system call interception method over traditional approaches.

A direct comparison between these tools is inherently complex due to significant differences in their architecture and scope. Therefore, for a more equitable comparison, this analysis focuses only on the overhead introduced by each virtualization technology.

Execution Mode	Virtualization	Tecnology	Startup Overhead
QEMU (qemu-x86-64)	Passthrough	Dynamic Binary Translation	Low
syskvm	Partial	KVM	Medium
VUOS (umvu)	Partial	Seccomp-bpf	Medium
gVisor (runsc-systrap)	Full	Seccomp-bpf	High
gVisor (runsc-kvm)	Full	KVM	High

Table 1: Comparison of tested virtualization technologies

3.1. Testing Environment

The tests were executed inside an LXC container running Debian 13, created using Proxmox VE [35] version 8.3.5 (Linux kernel 6.8.12). The container was granted direct access to the host's `/dev/kvm` device to avoid hardware virtualization overhead. The container was configured with 4 cores of an Intel Xeon Silver 4110 CPU at 2.10GHz and 4 GB of DDR4 RAM at 2666 MHz.

The versions of the benchmarked programs are as follows:

- **qemu-x86_64**: version 10.0.2
- **umvu**: version 0.9.2
- **runsc**: version release-20250820.0

The **hyperfine**[36] command-line tool (version 1.19.0) was used to conduct the benchmarks. Hyperfine provides statistically robust measurements by automating the execution and timing process. Each benchmark was configured with 3 warm-up runs, followed by a minimum of 10 measured execution runs (Chapter 5.1).

3.1.1. Benchmark Programs

Three distinct test programs were developed to test different aspects of process execution:

1. **System Call Throughput (`test-syscall` Chapter 5.2.2):** This program executes the `getpid()` system call 100.000 times in a loop. It is designed to measure the overhead of system call interception and emulation, which is the most critical performance metric for this thesis.
2. **CPU-Bound Workload (`test-cpu` Chapter 5.2.3):** This program performs a simple subtraction operation within a doubly nested loop, iterating 50.000 times in each loop. This test is designed to measure raw computational performance.
3. **I/O Performance (`test-io` Chapter 5.2.4):** This program reads 8 KB chunks from the `/dev/urandom` character device and writes it to the `tmp/trash` file in a loop 10.000 times. It aims to measure the performance of I/O operations, which typically involve system calls and data transfer between the guest and host.

For gVisor, which operates on OCI containers, a minimal container image was created to execute these test programs (Chapter 5.2.5).

3.2. Results and Analysis

This section presents the results obtained from the three benchmark programs. The performance of each virtualization tool is compared against a "Native" baseline, which represents the execution of the test program directly on the host system without any virtualization layer.

3.2.1. System Call Overhead

The system call benchmark is the most revealing test for this study, as it directly measures the efficiency of the core interception mechanism.

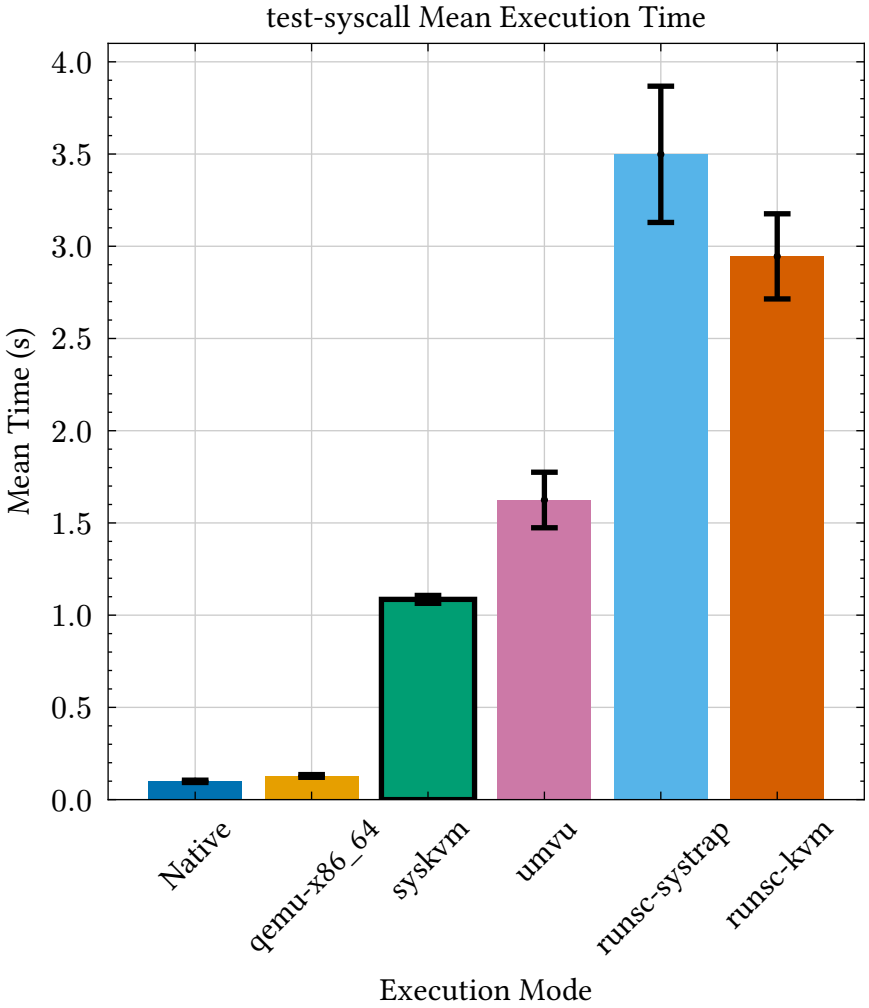

Figure 11: Benchmark results for the test-syscall program

System call interception is where most process-level virtualization approaches introduce significant overhead. Seccomp-bpf, the method used by VUOS and gVisor (systrap), is known for its high overhead in syscall-heavy workloads. Syskvm, by leveraging KVM to trap system calls via hardware exceptions, runs roughly **1.5 times faster** than umvu and is significantly faster than both gVisor

modes (Table 2). While all virtualization methods introduce substantial overhead compared to native execution, syskvm's approach proves to be a more performant alternative for this specific task.

3.2.2. CPU Performance

This test evaluates the impact of virtualization on pure computational workloads that do not involve kernel interactions.

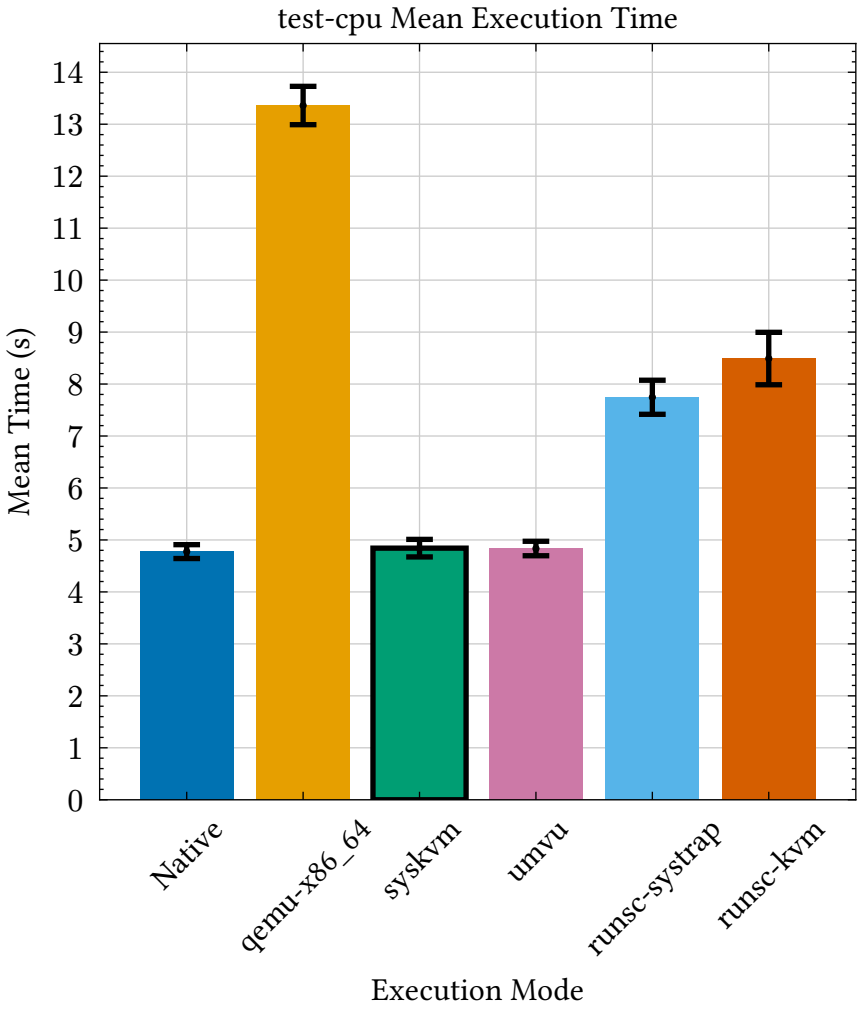

Figure 12: Benchmark results for the test-cpu program

Thanks to hardware virtualization, syskvm achieves near-native performance, running only **1.01 times slower** the baseline (Table 3). The small overhead is primarily due to VM startup and a few VM exit events (e.g., page faults). Performance is equivalent to VUOS (umvu), confirming that syskvm does not introduce CPU bottlenecks compared to other partial virtualization methods. In contrast, QEMU's dynamic binary translation imposes significant overhead, while gVisor's more complex architecture also results in a noticeable slowdown.

3.2.3. I/O Performance

This benchmark measures the efficiency of handling I/O operations, which involve a combination of system calls and data transfers between the guest and host address spaces.

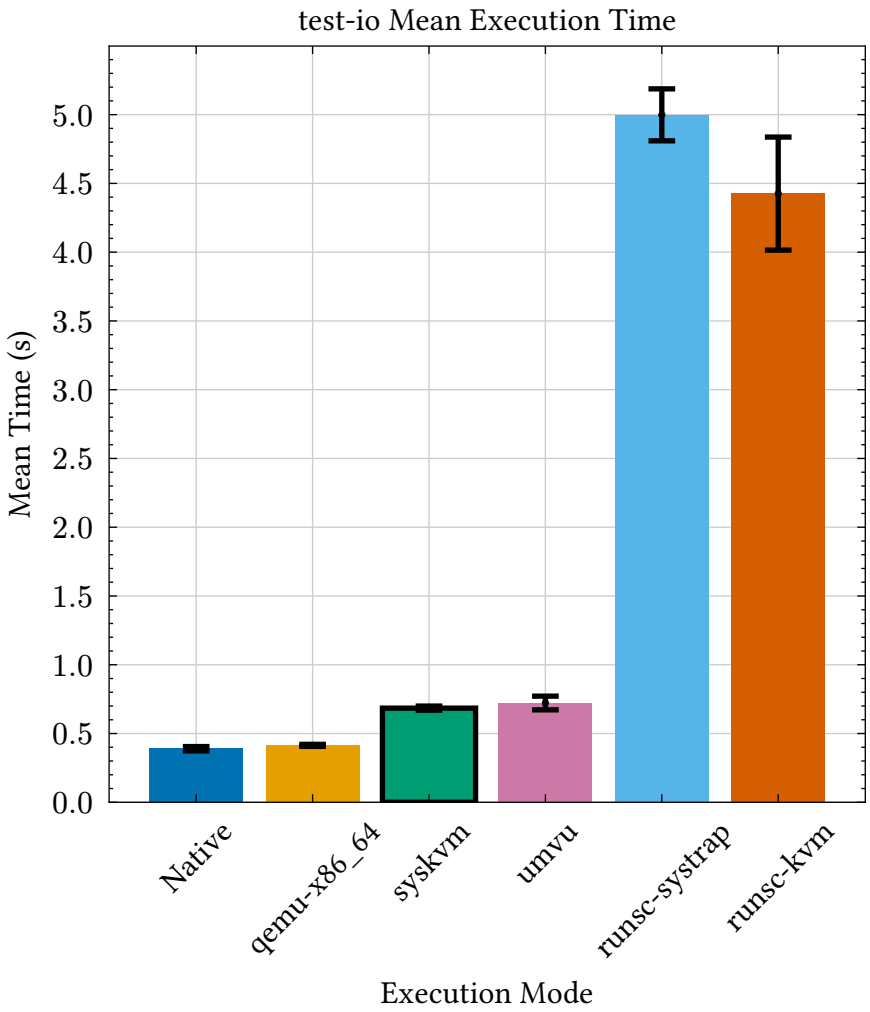

Figure 13: Benchmark results for the test-io program

Reading from a device is about **1.75 times slower** in syskvm than native execution (Table 4). This slowdown is primarily due to the need to copy data between guest and host memory, which is needed in this type of virtualization. syskvm's performance is nearly identical to VUOS, showing that the new method does not worsen I/O throughput compared to previous solutions.

Conclusions and Future Developments

This thesis introduced **syskvm**, a novel approach that brings hardware virtualization techniques to process-level and partial virtualization. Syskvm fills a previously unexplored space in the virtualization spectrum. As UML stands to VUOS, then QEMU-KVM stands to syskvm; it represents the process-level analog of hardware-accelerated system virtualization, providing new methods and opportunities for system call interception and application isolation.

4.1. Achievements and Contributions

The contributions of this thesis can be assessed from three perspectives: performance, security, and expressiveness.

Performance The performance analysis in Chapter 3 confirmed that Syskvm achieves its main goal of providing a more efficient mechanism for system call interception compared to traditional software-based approaches like `seccomp-bpf` and `ptrace`. Benchmarks show that syskvm outperforms these methods in system call-intensive workloads, delivers near-native CPU performance, and does not introduce additional I/O overhead compared to similar tools. This demonstrates that hardware-assisted virtualization can be successfully applied at the process level, reducing the overhead typically associated with software-based isolation.

Security By relying on hardware virtualization features instead of kernel mechanisms like `ptrace` or `seccomp-bpf`, syskvm reduces the host kernel's attack surface. The interface with KVM is minimal, and the isolation is enforced by the CPU itself. This hardware-enforced boundary between the guest process and the host is generally considered more secure than software-based techniques, offering stronger guarantees against kernel exploits and other vulnerabilities.

Expressiveness Syskvm unlocks new possibilities that are not available with traditional process-level virtualization. Because the guest code runs inside a KVM virtual machine, it can execute privileged instructions (i.e., code intended for ring 0) without any risk to the host system. This capability is similar to that provided by **Dune**[37], a system that uses virtualization hardware to give user-level applications safe access to privileged CPU features. However, unlike Dune, `syskvm` does not require a custom kernel module, making it more accessible. This feature can turn `syskvm` into a powerful tool for education and experimentation, allowing developers and students to explore the x86-64 architecture by writing simple ELF programs instead of entire kernels.

4.2. Future Developments

While the syskvm prototype demonstrates the viability of this approach, it is far from a complete and production-ready virtualization solution. The following areas represent promising directions for future work.

Integration and System Development:

VUOS Integration The most natural next step is to integrate `syskvm`'s interception mechanism as a new backend for the VUOS system. This would allow VUOS to leverage hardware virtualization for creating partial virtualizations, offering better performance and a security alternative to its existing `seccomp-bpf` method.

OCI Container Runtime The architecture could be extended to implement a full container runtime, similar to gVisor, leveraging hardware virtualization for isolation.

Technical Enhancements:

Memory Management The current memory mapping mechanism, which clones the layout of a reference process, should be replaced with a more robust and stable solution. Implementing a dedicated ELF loader and memory manager within syskvm would be essential for production readiness.

Multithreading Support The current prototype is limited to single-threaded applications. To support more software, a scheduling mechanism and support for multiple vCPUs must be implemented. This would involve handling system calls, such as `clone`, and managing shared resources between threads.

System Call Completeness The current implementation supports a limited subset of system calls. Future work must focus on expanding this support to cover more complex functionalities, including signal handling and resource sharing.

4.3. Final Remarks

In conclusion, this thesis has successfully introduced and tested a new method for process-level virtualization that combines the modularity and lightweight nature of process-based virtualization with the performance and security of hardware-assisted virtualization. Syskvm has shown that this approach is not only feasible but also offers some benefits, opening new virtualization possibilities.

Appendices

5.1. Benchmarking Script

A helper script that test a given program with hyperfine across all execution modes.

```bash
#!/bin/bash

# Check if a program path is provided
if [ $# -eq 0 ]; then
  echo "Usage: $0 <program_path>"
  exit 1
fi

# Get the program path
PROGRAM_PATH="$1"

# Extract program name (last part of the path without any directory)
PROGRAM_NAME=$(basename "$PROGRAM_PATH")
COINTAINER_NAME="${PROGRAM_NAME#./}-oci"

# Run the benchmark with hyperfine
echo "Running benchmark for $PROGRAM_PATH"
echo "Results will be saved to /tmp/$PROGRAM_NAME.json"
```

```
echo

hyperfine -w 3 --export-json "/tmp/$PROGRAM_NAME.json" \
  "$PROGRAM_PATH" \
  "qemu-x86_64 $PROGRAM_PATH" \
  "../../release/syskvm -- $PROGRAM_PATH" \
  "umvu $PROGRAM_PATH" \
  "sudo docker run --runtime=runsc-systrap $COINTAINER_NAME" \
  "sudo docker run --runtime=runsc-kvm $COINTAINER_NAME"
```

5.2. Test programs

All programs are compiled with **gcc** version 14.2 and `-static -nostdlib` flags.

5.2.1. syscall.h

```c
long syscall_syscall(long syscall_number, long arg1, long arg2,
long arg3, long arg4, long arg5, long arg6) {

    register long syscall_no __asm__("rax") = syscall_number;
    register long a1 __asm__("rdi") = arg1;
    register long a2 __asm__("rsi") = arg2;
    register long a3 __asm__("rdx") = arg3;
    register long a4 __asm__("r10") = arg4;
    register long a5 __asm__("r8") = arg5;
    register long a6 __asm__("r9") = arg6;
    __asm__("syscall");

    if (syscall_no < 0) {
        return -syscall_no;
    }

    return syscall_no;
}
```

5.2.2. test-syscall

```c
#include <asm/unistd_64.h>
```

```c
#include "syscall.h"

#define ITERATIONS 100000

void __attribute__((noreturn)) __attribute__((section(".start")))
_start(void) {

    for (int i = 0; i < ITERATIONS; i++) {
        syscall_syscall(__NR_getpid, 0, 0, 0, 0, 0, 0);
    }

    syscall_syscall(__NR_exit_group, 0, 0, 0, 0, 0, 0); // exit
    succes
    for (;;) {
        __asm__("hlt");
    }
}
```

5.2.3. test-cpu

```c
#include "syscall.h"
#include <asm/unistd_64.h>

#define ITERATIONS 50000

void __attribute__((noreturn)) __attribute__((section(".start")))
_start(void) {
    long x = 0;
    for (long i = 0; i < ITERATIONS; i++) {
        for (long j = 0; j < ITERATIONS; j++) {
            x = i - j;
        }
    }

    syscall_syscall(__NR_exit_group, x, 0, 0, 0, 0, 0);
    for (;;){
        __asm__("hlt");
    }
}
```

5.2.4. test-io

```c
#include <asm/unistd_64.h>
#include <fcntl.h>
#include "syscall.h"

#define ITERATIONS 10000

// File permissions (0644 = owner rw, group/others r)
#define FILE_MODE 0644

// Buffer size (8KB)
#define BUFFER_SIZE (8 * 1024)

void __attribute__((noreturn)) __attribute__((section(".start")))
_start(void) {
    // Buffer to store data from urandom
    char buffer[BUFFER_SIZE];

    // Open /dev/urandom
    const char urandom_path[] = "/dev/urandom";
    long urandom_fd = syscall_syscall(__NR_open,
    (long)urandom_path, O_RDONLY, 0, 0, 0, 0);

    // Check if open failed
    if (urandom_fd < 0) {
        syscall_syscall(__NR_exit_group, 1, 0, 0, 0, 0, 0); // Exit
        with error
    }

    // Open /tmp/trash file for writing (create if doesn't exist,
    truncate if it does)
    const char trash_path[] = "/tmp/trash";
    long trash_fd = syscall_syscall(__NR_open, (long)trash_path,
    O_WRONLY | O_CREAT | O_TRUNC, FILE_MODE, 0, 0, 0);

    // Check if open failed
    if (trash_fd < 0) {
        syscall_syscall(__NR_exit_group, 2, 0, 0, 0, 0, 0); // Exit
        with error
    }
```

```
    // Read from urandom and write to /tmp/trash 1000 times
    for (int i = 0; i < ITERATIONS; i++) {
        // Read 8KB from urandom
        long bytes_read = syscall_syscall(__NR_read, urandom_fd,
        (long)buffer, BUFFER_SIZE, 0, 0, 0);

        // Write to /tmp/trash
        long bytes_written = syscall_syscall(__NR_write, trash_fd,
        (long)buffer, bytes_read, 0, 0, 0);

        // Check if write failed
        if (bytes_written != bytes_read) {
            syscall_syscall(__NR_exit_group, 4, 0, 0, 0, 0, 0); //
            Exit with error
        }
    }

    // Close files
    syscall_syscall(__NR_close, urandom_fd, 0, 0, 0, 0, 0);
    syscall_syscall(__NR_close, trash_fd, 0, 0, 0, 0, 0);

    // Exit successfully
    syscall_syscall(__NR_exit_group, 0, 0, 0, 0, 0, 0);

    for (;;) {
        __asm__("hlt");
    }
}
```

5.2.5. Oci Image creation

Images are created with Docker, A configuration file is provided, the others are analogous.

```
FROM gcc as builder                                          dockerfile
COPY test-syscall.c /test-syscall.c
COPY syscall.h /syscall.h
RUN gcc -static -nostdlib -o /test-syscall /test-syscall.c

FROM scratch
```

```
COPY --from=builder /test-syscall /test-syscall
ENTRYPOINT ["/test-syscall"]
```

5.3. Testing Results

Execution Mode	Mean (s)	Min (s)	Max (s)	Std Dev (s)
Native	0.09914	0.08885	0.11368	0.00624
QEMU (qemu-x86_64)	0.12829	0.11534	0.14207	0.0074
syskvm	1.08555	1.05198	1.12528	0.02144
VUOS (umvu)	1.62438	1.39918	1.74405	0.15057
gVisor (runsc-systrap)	3.49845	3.11699	4.1765	0.36938
gVisor (runsc-kvm)	2.94526	2.63266	3.35682	0.23079

Table 2: test-syscall

Execution Mode	Mean (s)	Min (s)	Max (s)	Std Dev (s)
Native	4.77555	4.58267	4.94491	0.13336
QEMU (qemu-x86_64)	13.3599	12.77341	13.88561	0.36898
syskvm	4.84193	4.57062	5.02616	0.16882
VUOS (umvu)	4.83511	4.57009	4.96906	0.13923
gVisor (runsc-systrap)	7.74564	7.35382	8.34634	0.32775
gVisor (runsc-kvm)	8.4908	7.59433	9.08327	0.50444

Table 3: test-cpu

Execution Mode	Mean (s)	Min (s)	Max (s)	Std Dev (s)
Native	0.38946	0.37364	0.41689	0.01564
QEMU (qemu-x86_64)	0.41381	0.4021	0.42559	0.00737
syskvm	0.68448	0.66106	0.71515	0.01499
VUOS (umvu)	0.72188	0.66515	0.78333	0.0498
gVisor (runsc-systrap)	4.99847	4.71494	5.30736	0.18901
gVisor (runsc-kvm)	4.4258	3.53995	5.15081	0.41105

Table 4: test-io

Bibliography

[1] J. Smith and R. Nair, "The architecture of virtual machines," *Computer*, vol. 38, no. 5, pp. 32–38, 2005, doi: 10.1109/MC.2005.173.

[2] R. Davoli and M. Goldweber, "Virtual Square: Users, Programmers & Developers Guide," *Lulu Book*, 2011.

[3] "Documentation/Platforms - QEMU." [Online]. Available: https://wiki.qemu.org/Documentation/Platforms

[4] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the Linux virtual machine monitor," in *Proceedings of the Linux symposium*, 2007, pp. 225–230.

[5] E. W. Biederman and L. Networx, "Multiple instances of the global linux namespaces," in *Proceedings of the Linux Symposium*, 2006, pp. 101–112.

[6] "Control Groups; The Linux Kernel documentation." [Online]. Available: https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v1/cgroups.html

[7] J. Dike, "User-mode linux," in *5th Annual Linux Showcase & Conference (ALS 01)*, 2001.

[8] "ptrace(2) - Linux manual page ." [Online]. Available: https://www.man7.org/linux/man-pages/man2/ptrace.2.html

[9] "The Container Security Platform - gVisor." [Online]. Available: https://gvisor.dev/

[10] "Open Container Initiative." [Online]. Available: https://opencontainers.org/

[11] “Plan 9 Remote Resource Protocol.” [Online]. Available: https://ericvh.github.io/9p-rfc/rfc9p2000.html

[12] J. Jia *et al.*, “Programmable system call security with ebpf,” *arXiv preprint arXiv:2302.10366*, 2023.

[13] “FakeRoot - Debian Wiki.” [Online]. Available: https://wiki.debian.org/FakeRoot

[14] “Fakeroot NG.” [Online]. Available: https://fakeroot-ng.lingnu.com/index.php?title=Main_Page

[15] R. Davoli, “VDE: virtual distributed Ethernet,” in *First International Conference on Testbeds and Research Infrastructures for the DEvelopment of NeTworks and COMmunities*, 2005, pp. 213–220. doi: 10.1109/TRIDNT.2005.38.

[16] L. Bassi, D. Berardi, and R. Davoli, “VUOS: A User-Space Hypervisor Based on System Call Hijacking,” in *Computer Safety, Reliability, and Security. SAFECOMP 2024 Workshops*, A. Ceccarelli, M. Trapp, A. Bondavalli, E. Schoitsch, B. Gallina, and F. Bitsch, Eds., Cham: Springer Nature Switzerland, 2024, pp. 296–307.

[17] “VirtualSquare.” [Online]. Available: https://wiki.virtualsquare.org/

[18] “Syskvm: System call KVM virtual machine Github repository.” [Online]. Available: https://github.com/fabiomurer/syskvm

[19] Intel Corporation, “Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture.” [Online]. Available: https://cdrdv2.intel.com/v1/dl/getContent/671436

[20] “ioctl(2) - Linux manual page.” [Online]. Available: https://www.man7.org/linux/man-pages/man2/ioctl.2.html

[21] “The Definitive KVM (Kernel-based Virtual Machine) API Documentation - The Linux Kernel documentation.” [Online]. Available: https://docs.kernel.org/virt/kvm/api.html

[22] Intel Corporation, “Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3: System Programming Guide.” [Online]. Available: https://cdrdv2.intel.com/v1/dl/getContent/671447

[23] T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner, “One-Level Storage System,” *IRE Transactions on Electronic Computers*, no. 2, pp. 223–235, 1962, doi: 10.1109/TEC.1962.5219356.

[24] “x86-64 Microarchitecture Levels.” [Online]. Available: https://en.wikipedia.org/wiki/X86-64#Microarchitecture_levels

[25] T. I. Standard, “Executable and linking format (ELF) specification version 1.2,” *TIS Committee*, pp. 1–106, 1995.

[26] “vdso(7) - Linux manual page.” [Online]. Available: https://www.man7.org/linux/man-pages/man7/vdso.7.html

[27] “ld.so(8) - Linux manual page.” [Online]. Available: https://www.man7.org/linux/man-pages/man8/ld.so.8.html

[28] “syscall(2) - Linux manual page.” [Online]. Available: https://linuxman7.com/linux/man-pages/man2/syscall.2.html

[29] Intel Corporation, “Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2: Instruction Set Reference, A-Z.” [Online]. Available: https://cdrdv2.intel.com/v1/dl/getContent/671110

[30] “passwd(5) - Linux manual page.” [Online]. Available: https://man7.org/linux/man-pages/man5/passwd.5.html

[31] “shadow(5) - Linux manual page.” [Online]. Available: https://man7.org/linux/man-pages/man5/shadow.5.html

[32] R. Stallman, R. Pesch, S. Shebs, and others, “Debugging with GDB,” *Free Software Foundation*, vol. 675, 1988.

[33] “Remote Protocol (Debugging with GDB).” [Online]. Available: https://sourceware.org/gdb/current/onlinedocs/gdb.html/Remote-Protocol.html

[34] “mini-gdbstub: An implementation of the GDB Remote Serial Protocol to help you adding debug mode on emulator.” [Online]. Available: https://github.com/RinHizakura/mini-gdbstub

[35] “Proxmox Virtual Environment.” [Online]. Available: https://www.proxmox.com/en/products/proxmox-virtual-environment/overview

[36] “Hyperfine: A command-line benchmarking tool.” [Online]. Available: https://github.com/sharkdp/hyperfine/tree/master

[37] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis, "Dune: Safe user-level access to privileged {CPU} features", in *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, 2012, pp. 335–348.

Acknowledgements

I want to thank my family and the Azienda Regionale per il Diritto agli Studi Superiori dell'Emilia Romagna (ER.GO) for supporting me and allowing me to complete my university studies. Without their support, this thesis would not even exist.

I'm deeply grateful to my supervisor, Prof. Renzo Davoli, who, with his extraordinary ability to share knowledge and his immense passion for teaching, guided me through the creation of this thesis. Thank you also for the constant support of the ADMstaff project, which made my university journey rich with opportunities and wonderful friendships.

I thank my friends, the people from ADMstaff, and from Polisportiva G. Masi Orienteering: their company and help made my years of study pleasant and serene.

Finally, my thanks go to all the developers of free and open-source software, who make computer science a unique discipline in terms of sharing and collaboration. Without your work, this thesis would not have been possible.