Hochschule für Technik
und Wirtschaft Berlin

University of Applied Sciences

System Calls for Containerising and Managing Processes in Linux

A Thesis

Submitted in Partial Fulfillment of the Requirements for the Degree of

Master of Science (M.Sc.) in Applied Computer Science

at the

Berlin University of Applied Sciences (HTW)

First Supervisor:	Prof. Dr. Sebastian Bauer
Second Supervisor:	Dr. Michael Witt
Author:	B.Sc. Atanas Denkov
Matriculation Number:	s0559025
Submission Date:	04.10.2022

Contents

Chapter 1

Introduction

1.1 Motivation

Primitive support for multiprocessing in the form of basic context switching and dedicated input-output components was introduced in the late 1950s. Multiprocessing allowed for concurrent execution of multiple instructions at the cost of increased system complexity. Interleaved processes had a global unrestricted view of the system which inevitably led to unpredictable program behaviour. For example, programs had the ability to modify each other's memory and monopolise computer resources. Hence, to ensure correctness, every program had to carefully manage its interactions with hardware and all other processes in the system, which resulted in an unsustainable programming model.

The aforementioned issues were addressed by shifting the responsibility of resource management and process protection into a privileged control program that acted as an intermediary between hardware and user programs.

1.2 Objectives

1.3 Content Structure

Chapter 2

Fundamentals

2.1 Virtualisation

Chapter 2.1.1 introduces the fundamental axioms that define virtualisation - noninterference, isolation and performance. Chapter 2.1.2 presents hardware virtualisation as a concept, discusses the mechanisms it uses to satisfy the axioms, and highlights the trade-offs intrinsic to its design. Chapter 2.1.3 describes a more coarse-grained approach - operating system virtualisation - that has become the de-facto way of managing cloud workloads in multitenant environments.

2.1.1 Axioms

Noninterference

Codd et al. [Cod59] summarise the fundamental requirements of a multiprogramming system and emphasise the concept of noninterference between processes across space and time. *Spatial noninterference* is represented by all mechanisms that protect references to memory, disk and input-output devices [Cod59]. For example, memory segmentation is a method found in operating system kernels that assigns each process a dedicated portion of physical memory that is invisible to all other processes in the system. The kernel traps any attempt made by a process to access memory outside its allocated memory segment, thereby guaranteeing spatial noninterference [SGG18]. *Temporal noninterference* refers to those mechanisms that allocate execution time and protect against the monopolisation thereof [Cod59]. For instance, CPU scheduling is a technique that decides which process shall run on a core such that the core does not idle and all processes make sufficient runtime progress [SGG18]. The scheduling semantics, paired with an interrupt mechanism that makes sure that no process has hold of the core for too long, guarantee temporal noninterference.

Isolation

Anjali, Caraza-Harter, and Swift [ACS20] define isolation as the level of dependency that a virtualisation platform has towards the host kernel. We generalise this definition and say that *isolation* is the level of dependency that one piece of software has to another. Conceptually, isolation deals with explicit vertical relationships between software, and noninterference deals with implicit horizontal relationships between processes. Isolation can be quantified by counting the lines of external source code that a software executes to obtain a particular functionality. For example, Anjali, Caraza-Harter, and Swift [ACS20] count the lines of kernel code that a virtualisation platform executes when providing services to sandboxed applications. High counts indicate a strong dependency, i.e weak isolation, towards the kernel.

Performance

Randal [Ran20] defines performance as the contention between the overhead associated with isolating a process from its environment and the benefits of sharing resources between processes, i.e fully utilising the capacity of the underlying resource pool. Anjali, Caraza-Harter, and Swift [ACS20] use the same definition and contrast the isolation mechanisms provided by three different virtualisation platforms against processing unit, memory and input-output performance metrics. In particular, the authors define an application that computes prime numbers up to a limit. Since the workload is compute-bound, processing speed is measured and compared to the number of executed lines of code that reside in the `/arch`, `/virt` and `/arch/x86/kvm` subsystems of the Linux kernel. Manco et al. [Man17] use *same-host density* as a performance metric that measures the number of sandboxed applications that can be consolidated onto a single server. In addition, *boot, pause and unpause times* are also considered to be important performance indicators for particular use cases, such as elastic content delivery networks [Kue17] [Man17] and serverless computing.

2.1.2 Hardware Virtualisation

Popek and Goldberg [PG74] refer to the control program as a *virtual machine monitor* that ensures isolation and noninterference by providing every program with an environment that is "[...] effect identical with that demonstrated if the program had been run on the original machine directly" [PG74, p. 2]. This definition implies that a running program does not directly use the bare metal resources available. Instead, resources are emulated by the virtual machine monitor at the hardware level and presented as a dedicated physical system. Such an environment is called a *virtual machine*.

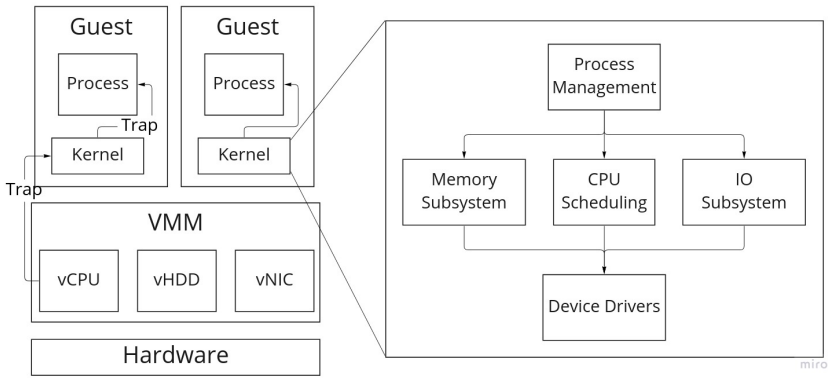

Figure 2.1: Hardware virtualisation architecture. Each guest runs a complete operating system. Privileged operations are trapped by the virtual machine monitor and emulated to provide hardware services.

Popek and Goldberg [PG74] define a requirement that the instruction-set architecture of a computer has to satisfy for it to be virtualisable. The instruction set must be segregated into three groups of instructions - privileged, sensitive and innocuous. An instruction is privileged if it requires changing the mode of execution from user to supervisor mode by means of a trap [PG74]. An instruction i is control-sensitive if, when applied to the current processor state S_1, results in a new state $i(S_1) = S_2$ such that the execution mode of S_2 does not equal that of S_1 or if S_2 has access to different resources than S_1 or both [PG74]. An instruction is behaviour-sensitive if its execution depends on the execution mode or its position in memory [PG74]. An instruction is innocuous if it is not sensitive. Given these definitions, a computer is virtualisable "[...] if the set of sensitive instructions for that computer is a subset of the set of privileged instructions" [PG74, p. 6]. If this criterion is met, the virtual machine monitor can trap all sensitive instructions and emulate each via a homomorphism $i : C_r \to C_v$ that maps the state space of the processor without the virtual machine monitor loaded C_r to the state space with the virtual machine monitor loaded C_v [PG74]. Innocuous instructions do not require protection, i.e a homomorphic mapping, and are directly executed by the processor [PG74].

Given the aforementioned homomorphism, a virtual machine can host a *guest kernel* (Figure 2.1) that runs completely in user mode. Whenever the guest kernel attempts to execute a privileged instruction, the virtual machine monitor traps the attempt and emulates the instruction. Consequently, the guest kernel does not have to be a part of the trusted computing base. Even if it is compromised or encounters an unrecoverable error condition, other virtual machines remain unaffected. As a result, the isolation boundary between user programs running in different virtual machines is stronger compared to processes running on a shared kernel.

In order to fully guarantee spatial noninterference between processes, the virtual machine monitor must be in full control of the host system's memory. There are two primary methods to do this - *shadow paging* and *extended page tables*. The former mechanism is considered first. The virtual

machine monitor maintains a nested page table per guest, also called a *shadow page table* [SN05]. In turn, the guest kernel maintains a page table per process. Whenever the guest kernel schedules a new process for execution, it modifies the *page-table base register* to point to the page table for that process [SN05]. The virtual machine monitor intercepts this attempt and transparently updates the page table pointer to point to the guest's shadow page table corresponding to that process [SN05]. Note that the virtual machine monitor has to traverse the shadow page table for that guest in order to find the nested entry corresponding to the process. Afterwards, the memory management unit takes care of translating the virtual memory addresses of the guest and updating the *translation lookaside buffer*. Alternatively, the memory management unit may be "virtualisation-aware" in the sense that it knows there are two page tables it needs to traverse - the page table that maps guest virtual memory to guest "physical memory", and the page table that maps guest physical memory to actual physical memory. The former is maintained by the guest kernel, whilst the latter is maintained by the virtual machine monitor. The extended page table approach is up to 50% faster than shadow paging [Esx06] because table walks are done in hardware - by the memory management unit. Nevertheless, maintaining page table data structures inside the virtual machine monitor and the guests leads to memory pressure, which is further amplified by the fact that guests, their applications and the virtual machine monitor all share the same physical memory [SGG18].

The spatial noninterference property necessitates that the virtual machine monitor manage all input-output devices and their interactions with the guests. This is accomplished by the already introduced trap-and-emulate pattern. When an application within a virtual machine issues a system call requesting some form of input-output, the request is processed by the I/O stack inside the guest. At the lowest level of the stack, the device driver issues a command to the device, typically by writing to memory specifically assigned to the device, or by calling specific input-output instructions [SGG18]. Either way, the virtual machine monitor intercepts this and traverses its own I/O stack, which remaps guest and real input-output addresses and forwards the request to a physical device [WR12]. After processing the request, the physical device triggers an interrupt that is caught by the virtual machine monitor and transformed into a virtual equivalent that is sent to the virtual machine that issued the request. To reduce the overhead associated with interrupt processing, the virtual machine monitor can batch multiple events together and use a single interrupt to notify the guest kernel [WR12]. Still, a request must traverse two input-output stacks. The same holds for the response. In addition, hardware optimisations such as direct memory access are emulated in software, which further degrades performance. This, however, can be mitigated by integrating an input-output memory management unit that remaps all direct memory accesses of a device on the host to an address space in the guest.

The cost of hardware virtualisation becomes apparent when measuring same-host density and boot times. Manco et al. [Man17] consider memory consumption and on-disk image size as the primary limiting factors. The authors measure the time it takes to create and boot virtual machines using the Xen virtual machine monitor and show the negative effects that on-disk

image size has by starting images with varying sizes by manually "[...] injecting binary objects into the uncompressed image file" [Man17, p. 3]. As the number of consolidated virtual instances increases and the image size grows, creation and boot times increase linearly. Furthermore, the authors show that creating and starting a process directly on the host is, on average, two orders of magnitude faster. Lv et al. [Lv12] also evaluate Xen and state that processing units spend 25% of their total cycles in hypervisor mode instead of executing guest applications when running "[...] SPEC's first benchmark addressing performance evaluation of datacenter servers used in virtualised server consolidation" [Lv12, p. 2], which includes components such as a web, database and application server.

2.1.3 Operating System Virtualisation

Operating system virtualisation refers to all mechanisms that enable the creation of secure and isolated application environments that run on top of a shared kernel. Conventionally, these mechanisms are baked into the kernel and are therefore part of the trusted computing base. The kernel may expose these through its system-call interface, thereby allowing a user-space daemon program to provide an automated facility for creating and orchestrating sandboxed environments. Alternatively, the kernel may treat every software component, including its own subsystems, as an entity to be wrapped in a sandbox. In that case, the environment in which every program is instantiated would have to satisfy all three virtualisation axioms.

Unlike hardware virtualisation, this architecture does not use hardware emulation as an isolation primitive. This means that shadow pages need not be maintained per virtual environment. Input-output operations need only traverse the kernel's stack without any address translations and with the additional performance benefit of direct memory access. As a result, the isolation overhead is lower compared to a virtual machine, which allows more applications to be consolidated onto a single server. Furthermore, guests do not boot up complete operating system images, which practically eliminates boot times. Priedhorsky and Randles [PR17] use operating system virtualisation in high-performance computing clusters to run user-defined compute jobs and show that the imposed performance penalties are, at most, negligible compared to vanilla processes that have no additional isolation.

List of Figures

List of Tables

Source Code Content

References

[ACS20] Anjali, Tyler Caraza-Harter, and Michael M. Swift. "Blending Containers and Virtual Machines: A Study of Firecracker and GVisor". In: *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE '20. Lausanne, Switzerland: Association for Computing Machinery, 2020, pp. 101–113. ISBN: 9781450375542. DOI: 10.1145/3381052.3381315. URL: https://doi.org/10.1145/3381052.3381315.

[Cod59] E. F. Codd et al. "Multiprogramming STRETCH: Feasibility Considerations". In: *Commun. ACM* 2.11 (Nov. 1959), pp. 13–17. ISSN: 0001-0782. DOI: 10.1145/368481.368502. URL: https://doi.org/10.1145/368481.368502.

[Esx06] Vmware Esx. "Performance Evaluation of Intel EPT Hardware Assist". In: 2006.

[Kue17] Simon Kuenzer et al. "Unikernels Everywhere: The Case for Elastic CDNs". In: *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE '17. Xi'an, China: Association for Computing Machinery, 2017, pp. 15–29. ISBN: 9781450349482. DOI: 10.1145/3050748.3050757. URL: https://doi.org/10.1145/3050748.3050757.

[Lv12] Hui Lv et al. "Virtualization Challenges: A View from Server Consolidation Perspective". In: *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*. VEE '12. London, England, UK: Association for Computing Machinery, 2012, pp. 15–26. ISBN: 9781450311762. DOI: 10.1145/2151024.2151030. URL: https://doi.org/10.1145/2151024.2151030.

[Man17] Filipe Manco et al. "My VM is Lighter (and Safer) than Your Container". In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP '17. Shanghai, China: Association for Computing Machinery, 2017, pp. 218–233. ISBN: 9781450350853. DOI: 10.1145/3132747.3132763. URL: https://doi.org/10.1145/3132747.3132763.

[PG74] Gerald J. Popek and Robert P. Goldberg. "Formal Requirements for Virtualizable Third Generation Architectures". In: *Commun. ACM* 17.7 (July 1974), pp. 412–421. ISSN: 0001-0782. DOI: 10.1145/361011.361073. URL: https://doi.org/10.1145/361011.361073.

[PR17] Reid Priedhorsky and Tim Randles. "Charliecloud: Unprivileged Containers for User-Defined Software Stacks in HPC". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '17. Denver, Colorado: Association for Computing Machinery, 2017. ISBN: 9781450351140. DOI: 10.1145/3126908.3126925. URL: https://doi.org/10.1145/3126908.3126925.

[Ran20] Allison Randal. "The Ideal Versus the Real: Revisiting the History of Virtual Machines and Containers". In: *ACM Comput. Surv.* 53.1 (Feb. 2020). ISSN: 0360-0300. DOI: 10.1145/3365199. URL: https://doi.org/10.1145/3365199.

[SGG18] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*. 10th. Wiley Publishing, 2018. ISBN: 9781119320913.

[SN05] Jim Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005. ISBN: 1558609105.

[WR12] Carl Waldspurger and Mendel Rosenblum. "I/O Virtualization". In: *Commun. ACM* 55.1 (Jan. 2012), pp. 66–73. ISSN: 0001-0782. DOI: 10.1145/2063176.2063194. URL: https://doi.org/10.1145/2063176.2063194.