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 concur-
rent 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. This program was most commonly referred to as a kernel.

TODO

1.2 Objectives

The primary objective of this thesis is to introduce and benchmark the namespace abstraction
and see how it affects the performance of an application. In addition, each namespace will
be qualitatively evaluated against its ability to satisfy two security axioms - isolation and
noninterference between processes. The qualitative results will be contrasted against the results
of the performance benchmark to show the relationship between resource namespacing and
performance.

1.3 Content Structure

TODO

Chapter 2

Fundamentals

Section 2.1 summarises the concept of virtualisation. Section 2.2 introduces the concept of a namespace as defined in the Linux kernel and describes the individual namespace types and their effects on processes. Section 2.3 presents the benchmarking tool that will be used to assess the performance of a namespaced application.

2.1 Virtualisation

Virtualisation is the process of abstracting the execution environment of an application into a unit that is secure, manageable and performant. The primary value propositon of a virtualisation platform is the consolidation of independent and untrusted workloads onto a single server which cannot interfere with each other or with the host, and still exhibit optimal performance characteristics. Virtualisation is, in essence, a cost reduction technique for infrastructure providers. Unfortunately, security and performance are concepts that often conflict with each other, thereby rendering virtualisation into a particularly challenging domain.

Section 2.1.1 presents the fundamental axioms that define virtualisation - noninterference, isolation, and performance. The first two axioms will be used to assess the security properties that namespaces have, while the last one will be used to define a benchmark with metrics that measure the performance capabilities of a virtualised application.

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.

2.2 Namespaces

A namespace encapsulates a system resource and a collection of processes. Only processes that are part of the namespace can see and manipulate the resource. Namespaces are implemented in the kernel and are therefore a part of the trusted computing base. Container runtimes use them to establish a solid noninterference boundary between processes by restricting their view of the system. The system calls `clone`, `unshare`, `setns` and `ioctl`, as well as the `procfs` pseudo-filesystem are the primary interaction points between a user-space program and a namespace.

Every process is associated with a fixed-size set of namespaces, called the *namespace proxy*, that corresponds to the resources that it can access. In user-space, the proxy is contained within the `/proc/[pid]/ns` directory as a collection of symbolic links, each pointing to an inode number uniquely identifying a namespace that wraps the process identified by `pid`. Currently, there are seven different types of namespaces reflected in the proxy. Each will be discussed individually in the upcoming sections.

The lifecycle of a namespace is tightly coupled to the lifecycle of all processes that reside in it. This is reflected in how namespaces are created and destroyed. A namespace and a process can be atomically created via the `clone()` system call. `clone()` creates a child process that begins its lifecycle by executing a function pointer that returns the exit code with which the child process will terminate. In addition, the caller passes a set of configuration options that define the child's execution context. The parent can put the child in a freshly-allocated set of namespaces by ORing a set of flags and injecting the result into the configuration set. It is important to note that the configuration set is very flexible and defines the noninterference boundary between the child and its parent. The parent can use it to share its virtual memory pages, file-descriptor table and filesystem information with the child, thereby weakening the boundary. `unshare()` allows an already existing process to regulate its noninterference boundary by "disassociating parts of its execution context". In other words, a process can reverse the configuration options used when it was created. Hence, the process can detach itself into a new set of namespaces. Alternatively, `setns()` can be used by a process to join a namespace that already exists. The caller must have an open file descriptor referencing a namespace object from the `/proc/[pid]/ns` directory. Notice that a namespace cannot be created without having a process to reside in it. A namespace is implicitly destroyed by the kernel when all processes inside it terminate. However, a user-space process can keep a namespace alive by explicitly referencing it through a file descriptor obtained from the proxy.

Some namespaces are represented as n-ary trees and can be nested up to 32 times in order to create multiple noninterference boundaries. Relationships between namespaces can be queried via the `ioctl` system call that, in combination with the `NS_GET_PARENT` flag, returns an initialised read-only file descriptor pointing at the parent namespace.

2.2.1 User namespace

Every process is associated with a set of credentials. The traditional UNIX security model defines these credentials as a set of unique numerical user and group identifiers. The *real* user and group identifiers represent the user to which the process belongs. They are read from the password database as part of the login procedure [Ker10]. Every process spawned by the user inherits the real user and group identifiers as part of its credentials. The credentials themselves further consist of an *effective* user and an effective group identifier, which are used by the kernel to authorise and execute operations on behalf of the process [Ker10]. For example, when a process attempts to read a file, the kernel checks if the effective user ID of the process matches the user ID of the owner of the file, or if the effective group ID matches the owner's group ID. If any of these predicates holds, then the effective user - and therefore the process - is deemed trustworthy to call the `read` system call on a file descriptor referencing the file. Typically, the effective user and group identifiers match their real counterparts. However, there are programs that can ambiently change the effective identifiers of a process to match those of the user that created them [Ker10]. These programs are known as *set-user-id* and *set-group-id* programs. Every executable file is associated with a set-user-id and a set-group-id permission bit that can be set by that file's owner. If the set-user-id permission bit is set and another user executes the program, the process obtains the effective user identifier of the file's owner, not the user that started the process [Ker10]. A classic example of a set-user-id program is `sudo`, which executes arbitrary commands as the root user. Since the `sudo` executable file is owned by the root user and has the set-user-id bit toggled, any user cam execute any command in a process with full privileges. Another example is the `passwd` utility which allows a user to change her password and therefore requires write access to the password database.

Notice that the UNIX security model differentiates only between privileged and unprivileged processes. The former are not subjected to permission checking and have full control of the system - they can reboot the system, set the system time, kill other processes and so on. If a set-user-id or a set-group-id program exhibits unexpected behaviour, either due to malicious manipulation or programming errors, this coarse-grained security model is unable to prevent a full system exploitation. The Linux capability scheme tries to solve this problem by unbinding privileges from the root user and dividing them into components called *capabilities*. A capability is represented as a single bit that is a part of a 64-bit wide bit mask, called a *capability set*. Every process is associated with a *permitted*, *inheritable*, *effective*, *bounding* and *ambient* capability set. In this model, the effective capability set is used by the kernel to do permission checking. The other sets define what can be stored in the effective set when loading executable files that happen to also have capabilities. The traditional UNIX credentials, paired with the capability sets, represent a process's security context.

A user namespace encapsulates the security contexts of its resident processes. When a process creates a new user namespace, its effective user identifier becomes the owner of the namespace

and is granted full capabilities within it. In other words, a non-privileged user on the host system can become `root` within its own user namespace. The kernel associates every resource and thus every other namespace type with a user namespace. A process is granted access to a resource in accordance with its credentials in the user namespace that owns the resource. Hence, on its own, a user namespace is not particularly useful because it does not own any resources. If created in pair with other namespaces, however, it becomes extremely useful.

```
1  $ id -u && cat /proc/$$/status | grep CapEff
2  1000
3  CapEff: 0000000000000000
4  $ unshare -U -r
5  $ id -u && cat /proc/$$/status | grep CapEff
6  0
7  CapEff: 000001ffffffffff
8  $ hostname mynewhostname
9  you must be root to change the host name
10 $ unshare --uts
11 $ hostname mynewhostname
12 $ hostname
13 mynewhostname
```

Code snippet 2.1: Example of resource ownership semantics with user namespaces

Example (2.1) highlights how the kernel authorises operations in a user namespace. First, the security context of the current process is shown. The user running the shell has no capabilities. The `unshare -U -r` command moves the shell into a newly-created user namespace. We can see that the process has obtained a superuser security context with a full effective capability set. Afterwards, an attempt is made to set the host name of the computer to an arbitrary string. The operation fails, indicating that the process has insufficient privileges, even though the required capability is in its effective set. Why? The kernel uses an additional namespace for encapsulating UNIX time-sharing (uts) operations. The shell's uts namespace is owned by the initial user namespace, in which our user is unprivileged as shown on lines 1-3. To make the new user namespace useful, we encapsulate the hostname resource in a new uts namespace and move the shell process there. Afterwards, the kernel uses the security contexts of the new user namespace to authorise any UNIX time-sharing operations.

Before the advent of user namespaces, a container runtime was required to run with superuser privileges, because the `clone()` and `unshare()` system calls require the `CAP_SYS_ADMIN` capability. Now, if the `CLONE_NEWUSER` flag is set, both system calls guarantee that the process will be moved in a new user namespace first, and its new security context will be used to authorise the creation of the remaining set of namespaces.
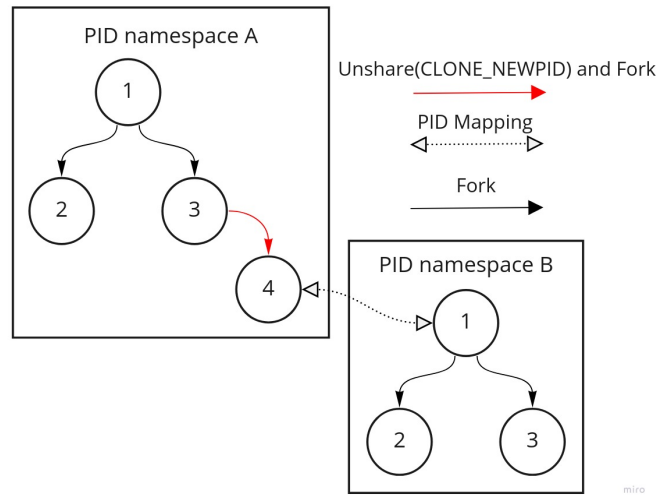
2.2.2 Process identifier namespace

Figure 2.1: PID namespace hierarchy

Process identiifer (PID) namespaces encapsulate the mechanism of assigning unique identifiers to their resident processes. Every PID namespace localises an associative array, called the identifier registry, capable of allocating up to 2^{22} unique numbers and mapping them to arbitrary pointers. Because of this localisation, processes in different namespaces can be mapped to the same identifier. The first process to join a PID namespace receives the process identifier 1 and acts as the init process for the entire namespace. This process becomes the parent of any orphaned children inside the namespace. Furthermore, if the init process terminates, all processes in the namespace terminate as well. In this case, even if a file descriptor to the namespace is kept open, i.e the namespace is kept alive by the kernel, no process is allowed to join.

Process identifier namespaces are organised into a hierarchy, as shown by Figure 2.1. When a process calls `unshare` with the `CLONE_NEWPID` flag set, the kernel does not detach the process into a new PID namespace. Doing so would require changing the identifier of the process. Instead, the kernel caches the new namespace in the process's namespace proxy and spawns the process's future children into it. The new process namespace is accessible from user-space via the `/proc/[pid]/ns/pid_for_children` file.

We denote N_i as the i-th process in namespace N. In Figure 2.1, A_1 forks two children A_2 and A_3. A_3 calls `unshare` and registers a new PID namespace in the kernel. The process subsequently forks A_4, which the kernel translates into B_1. Note that A_1 and A_2 can, by definition, interact with B_1 because $B_1 = A_4$. Hence, a process is visible to all of its peers in its namespace and to all direct ancestor namespaces. Conversely, B_2 and B_3 cannot see any processes in A. Joining a PID namespace is an irrevirtible operation, i.e if A_2 joins B, then it cannot go back.

PID namespaces can be nested arbitrarily up to 32 times. Container runtimes do not utilise this feature, because application workloads are orthogonal to each other. Nesting two application workloads, potentially stemming from two different tenants, in a hierarchy would enable processes resident in the ancestor namespace to kill the init process of the child namespace, which is in direct conflict with the noninterference property.

```c
/* pid_namespace.c */
int err = unshare(CLONE_NEWPID);
pid_t child = fork();
if (child == 0)
    child = fork();
    if (child == 0)
        execlp("sleep", "sleep", "60", (char *) NULL);
    else if (child > 0)
        exit(waitpid(child, NULL, 0) != child);
else
    return waitpid(child, NULL, 0) != child;
```

Code snippet 2.2: PID namespace creation pseudocode

Code snippet 2.2 demonstrates the creation of a new PID namespace with two processes. On lines 1 and 2, a new pid namespace is created through the `unshare` system call and its init process is forked. The init process creates a new child that sleeps for 60 seconds. All processes within this hierarchy await their children's completion. We can run this example in a shell and list the shell's children via the `ps` utility, as shown in Code snippet 2.3. You'll notice that the `sleep` command is visible from the ancestor namespace and can be killed, i.e directly interfered with.

```
$ ./pid-namespace &
[1] 78972
$ ps
PID TTY TIME CMD
62983 pts/1 00:00:00 bash
78972 pts/1 00:00:00 pid-namespace
78973 pts/1 00:00:00 pid-namespace
78974 pts/1 00:00:00 sleep
79027 pts/1 00:00:00 ps
```

Code snippet 2.3: Example of creating a PID namespace and listing all processes in the child namespace from a process in the parent namespace

2.2.3 Mount Namespace

Files are multiplexed across multiple devices, each with its own intrinsic implementation details for storing and managing data. The kernel abstracts the underlying idiosyncrasies by arranging all files into a hierarchy that can be accessed through a well-defined programming interface. The process of attaching a device's file system to this hierarchy is called *mounting*. The position in the hierarchy where the file system is attached is referred to as a *mount point*.

Mount namespaces encapsulate the list of mount points that their resident processes can see. When a process detaches into a new mount namespace, it inherits an exact replica of the parent's mount points. This does not necessarily mean that a change made by the parent to an underlying mount point remains invisible to the child, or vice versa. Every mount point is associated with a dedicated *propagation type* that determines whether or not file system events are shared across mount namespaces that reference it. In essence, the propagation types governs the file system's noninterference boundary.

2.2.4 Network Namespace

2.2.5 Inter-process Communication Namespace

2.2.6 Time Sharing Namespace

2.2.7 Control Group Namespace

2.3 Extended Berkley Packet Filters (eBPF)

Chapter 3

State of research

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. This is the only feasible architecture on a general-purpose kernel such as Linux. 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 concept of a process itself would have to satisfy all three virtualisation axioms. Examples of such operating systems include the seL4 *microkernel* - the first operating system to be formally verified as free of programming errors [Kle09], and Google's Fuchsia - described by Pagano, Verderame, and Merlo [PVM21].

An application, referred to as a *container*, is defined as an encapsulation of "[...] a software component and all its dependencies in a format that is self-describing and portable, so that any compliant container runtime can run it without extra dependencies, regardless of the underlying machine and the contents of the container" [Ini16, p. 1]. A *container runtime* is the user-space daemon program responsible for bringing this portable but static representation of an application into execution. In runtime, a container consists of a collection of processes that share a restricted view of the system's resources. For example, every container "believes" it has a dedicated network stack with its own network interfaces, routing tables and packet-forwarding rules. All processes in the container can access and manipulate that network stack, but no other process outside the container has that capability. The container runtime configures this invariant and the operating system enforces it by namespacing system resources. The Open Containers Initiative (OCI) [Ini15] has developed a runtime specification that standardises the operations a container runtime needs to support. Most importantly, it must allow an external process called a *container engine* to hook into the lifecycle of a container. The container engine can use these hooks to manage all the containers on a single host system. In addition, the engine can attach network and storage to containers, thereby allowing processes in different sandboxes to share state and communicate with each other, if required. At the highest level of abstraction sits an orchestration

platform that manages containers on multiple hosts by interacting with the container engine on each system. This platform constantly monitors node and container health and dynamically multiplexes workloads based on various system properties of the cluster as to ensure maximum application availability.

It is important to note that, by definition, the kernel is assumed to be trustworthy. It has full control of all hardware resources and can access and modify the execution environment of every process on the system. In other words, noninterference between the kernel and user processes is not guaranteed. Therefore, if the kernel is compromised, all processes on the system become untrustworthy. It follows that if a process compromises the kernel, it transitively interferes with all other processes on the system. Hardening the operating system by implementing various security features such as mandatory access control has been the primary approach for protection against such scenarios. However, the size of a monolithic general-purpose kernel is too large to adequately create a threat model that captures all possible vectors of attack. This problem is of particular concern to infrastructure providers whose entire business model revolves around consolidating hundreds of potentially malicious client applications on the same server, all of which share the same kernel and are allowed to directly interact with it via its system call interface.

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 reduces start times and memory consumption. Priedhorsky and Randles [PR17] use containers 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. Felter et al. [Fel15] show the exact same thing and further conclude that the Docker container engine is resource-friendlier and faster than the Kernel Virtual Machine (KVM) when stressing the "memory, IPC, network and filesystem subsystems" Felter et al. [Fel15, p. 1] of the Linux kernel by running a database inside a virtual environment and evaulating its performance via the SysBench OLTP benchmark [ZK04].

Google's gVisor [Goo18] attempts to sustain the performance advantages of containers whilst introducing an additional isolation boundary between the kernel and each container. The authors implement a substantial portion of the system call interface in a user-space process called Sentry. Their dedicated container runtime calls out to Sentry instead of the kernel when issuing system calls.
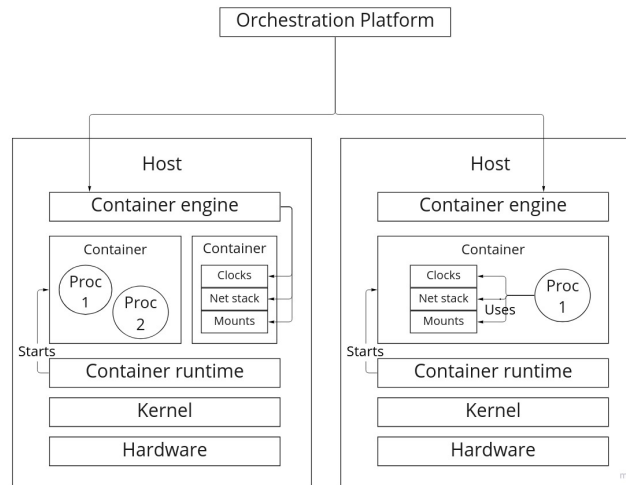
Figure 3.1: Operating system virtualisation architecture using containers. The container runtime starts containers on a single host. A user process can see a bundle of resources allocated to it by the kernel. The kernel guarantees that a process cannot see any other resources. The container engine manages all containers on a single system and allocates storage and networking to create explicit paths between containers. The orchestration platform talks to all engines inside a cluster to provide automatic workload management.

However, Young et al. [You19] show that network and memory allocation performance greatly suffer. This can be partially attributed to the fact that the Sentry process is implemented in a garbage-collected language and lacks the fine-grained optimisations contained in the Linux kernel. Agache et al. [Aga20] take a different approach and try to fuse the security of virtual machines with the performance of containers by programming a custom virtual machine monitor called Firecracker that runs on top of KVM. Firecracker completely relies on the Linux kernel for memory management, CPU scheduling and block I/O. To reduce its attack surface, the virtual machine monitor sacrifices portability by supporting a limited set of emulated network and block devices. To further strengthen the noninterference boundary, the devices have configurable built-in rate limiters that can control the number of operations per second, e.g disk/packets per second. Unlike a traditional container runtime, Firecracker's rate-limiting implementation does not rely on the kernel, which makes its isolation boundary to the kernel stronger. Anjali, Caraza-Harter, and Swift [ACS20] evaluate both gVisor and Firecracker and show that the latter "[...] is effective at reducing the frequency of kernel code invocations, but had a much smaller impact on reducing the footprint of kernel code" [ACS20, p. 12].

Chapter 4

Appendix A

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.

[Aga20]   Alexandru Agache et al. "Firecracker: Lightweight Virtualization for Serverless Applications". In: *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 419–434. ISBN: 978-1-939133-13-7. URL: https://www.usenix.org/conference/nsdi20/presentation/agache.

[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.

[Fel15]   Wes Felter et al. "An updated performance comparison of virtual machines and Linux containers". In: *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. Mar. 2015, pp. 171–172. DOI: 10.1109/ISPASS.2015.7095802.

[Ker10]   Michael Kerrisk. *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. 1st. USA: No Starch Press, 2010. ISBN: 1593272200.

[Kle09]   Gerwin Klein et al. "SeL4: Formal Verification of an OS Kernel". In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. SOSP '09. Big Sky, Montana, USA: Association for Computing Machinery, 2009, pp. 207–220. ISBN: 9781605587523. DOI: 10.1145/1629575.1629596. URL: https://doi.org/10.1145/1629575.1629596.

[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.

[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.

[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.

[PVM21]   Francesco Pagano, Luca Verderame, and Alessio Merlo. "Understanding Fuchsia Security". In: *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications* 12.3 (2021), pp. 47–64.

[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.

[You19]   Ethan G. Young et al. "The True Cost of Containing: A gVisor Case Study". In: *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*. Renton, WA: USENIX Association, July 2019. URL: https://www.usenix.org/conference/hotcloud19/presentation/young.

Online References

[Goo18]   Google. *gVisor*. 2018. URL: https://github.com/google/gvisor (visited on 05/26/2022).

[Ini15]   Open Container Initiative. *Open Containers Initiative Website*. 2015. URL: https://opencontainers.org/ (visited on 05/26/2022).

[Ini16]   Open Container Initiative. *The 5 principles of Standard Containers*. 2016. URL: https://github.com/opencontainers/runtime-spec/blob/main/principles.md (visited on 05/26/2022).

[ZK04]   Peter Zaitsev and Alexey Kopytov. *Sysbench OLTP*. 2004. URL: https://github.com/akopytov/sysbench (visited on 05/26/2022).