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. This program was most commonly referred to as a kernel.

1.2 Objectives

The primary objective of this thesis is to create a set of synthetic workloads and measure their performance characteristics when deployed inside a sandboxed environment. The same workloads will also be executed natively within the standard noninterference boundary provided by the kernel. The measurements will be compared and evaluated in an attempt to outline the relationship between noninterference, isolation and performance. A command-line tool will be developed to run the workloads and generate the measurements so that the results are reproducible.

A custom container runtime will be designed, implemented and used by the command-line tool as the primary sandboxing mechanism. The system call interface used for sandboxing the workloads

will be discussed. The overhead introduced by the runtime to spawn a container will also be compared to the overhead associated with spawning a simple process.

1.3 Content Structure

Chapter 2 introduces the fundamental axioms, or trade-offs, in virtualisation technologies. These will be referred to throughout the entire document. The same chapter also introduces the concept of resource namespaces - the primary abstraction provided by the kernel to sandbox processes. Furthermore, the extended Berkley Packet Filter (eBPF) subsystem of the kernel is described, which we use to probe metrics from the kernel that measure workload performance.

Chapter 3 outlines the current state of research in operating-system virtualisation, highlights the relationships between noninterference, isolation and performance, and discusses modern architectures that try to maximise all three.

Chapter 4 provides a detailed description of the requirements and the software architecture of the runtime, benchmark tool, and the workloads. The measurements to be sampled from the kernel are introduced.

Chapter 5 describes the system calls used to create the sandbox environment. Important implementation details are also mentioned.

Chapter 6 shows the results of the performance measurements and highlights the differences between running the workloads inside and outside a sandbox. The results are evaluated and the primary performance bottlenecks are discussed.

Chapter 7 contains conclusive remarks, a brief summary of the results of the thesis and future work that can be done by other students in the field.

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 **??** 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 `proc` pseudo file system are the primary interaction points between a user-space program and a namespace (See Figure 2.1).

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 file system 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.

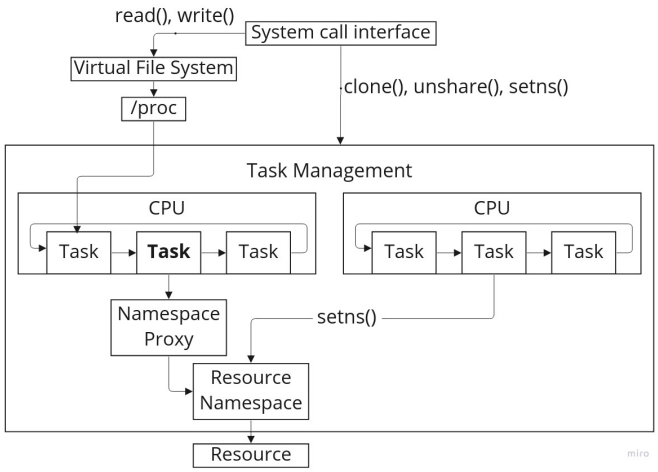

Figure 2.1: Namespace design inside the kernel. Every task is associated with a namespace proxy that can be manipulated through the virtual file system by reading and writing to files, or through the system call interface

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, it can map its real user identifier on the host system to the user identifier 0 in the new user namespace. 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: 000001fffffffff
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

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 (2.2). 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 (2.2), 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.

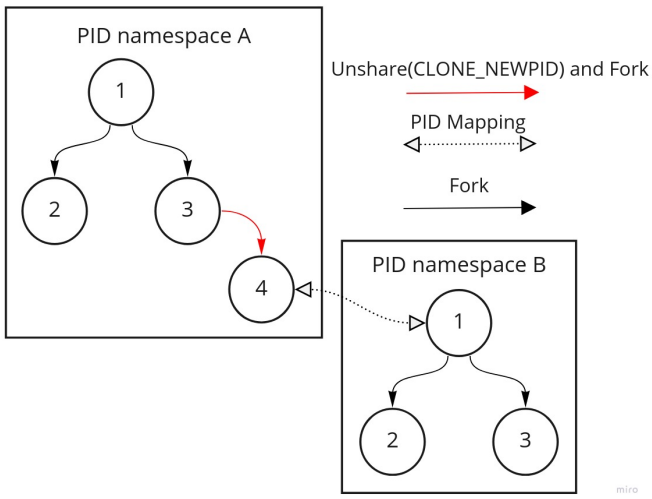

Figure 2.2: PID namespace hierarchy

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.

```
1  /* pid_namespace.c */
2  int err = unshare(CLONE_NEWPID);
3  pid_t child = fork();
4  if (child == 0)
5      child = fork();
6      if (child == 0)
7          execlp("sleep", "sleep", "60", (char *) NULL);
8      else if (child > 0)
9          exit(waitpid(child, NULL, 0) != child);
10 else
11     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.

```
1 $ ./pid-namespace &
2 [1] 78972
3 $ ps
4 PID TTY TIME CMD
5 62983 pts/1 00:00:00 bash
6 78972 pts/1 00:00:00 pid-namespace
7 78973 pts/1 00:00:00 pid-namespace
8 78974 pts/1 00:00:00 sleep
9 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 mount and umount events are shared across mount namespaces that reference it or any mount points immediately below it in the file hierarchy. In essence, the propagation type governs the file system's noninterference boundary. Table (2.1) summarises the possible propagation types. The `/proc/self/mountinfo` file displays a multitude of mount point properties with respect to the mount namespace in which the process reading the file resides in.

In addition to mounting device file systems onto the file hierarchy, we can make already mounted subtrees visible at other locations in it. This concept is known as a *bind mount*. Bind mounts are commonly used by container engines to map directories on the host system to directories inside a container. Internally, the kernel copies the data structures of the original mount into a new mount point inside container's mount namespace. Changes in one mount point are reflected in the other using copy on write.

Propagation Type	Description
`MS_SHARED`	Mount and umount events are propagated across mount namespaces.
`MS_PRIVATE`	Mount and umount events are local. The mount point does not propagate, nor does it receive mount and umount events.
`MS_SLAVE`	Mount and umount events are propagated unidirectionally from a master set of mount points to a slave mount.
`MS_UNBINDABLE`	Mount point is private and cannot be bind mounted

Table 2.1: Table of mount point propagation types in Linux

2.2.4 Network namespace

Network namespaces encapsulate the full network stack and expose it only to their resident processes. This allows a collection of processes to operate on dedicated network interfaces, routing tables, firewall rules and UNIX domain sockets that are inaccessible to others.

The kernel imposes an important restriction. A physical network device can be attached only to a single network namespace. When the network namespace is released, the phyiscal device is returned back to the initial network namespace, i.e the host. To enable socket-based interprocess communication across network namespace boundaries, the physical devices need to be multiplexed on top of virtual interfaces.

All container engines implement the same default network configuration that allows processes in different network namespaces to communicate with each other. As part of its initialisation procedure, the container engine installs a bridge interface inside the root network namespace. A bridge can be thought of as a virtual network switch. It forwards packets between interfaces, called slaves, that are attached to it. Before detaching a process into a new network namespace, the container engine creates a virtual ethernet device. Virtual ethernet devices are a software equivalent of a patch cable. They are represented by a pair of interconnected endpoints. One end of the virtual ethernet device is moved into the network namespace of the process, and the other remains on the host and is attached to the bridge. Because the bridge acts as a switch, other network devices attached to it can communicate with a namespaced process. Figure (2.3) visualises such a network configuration.

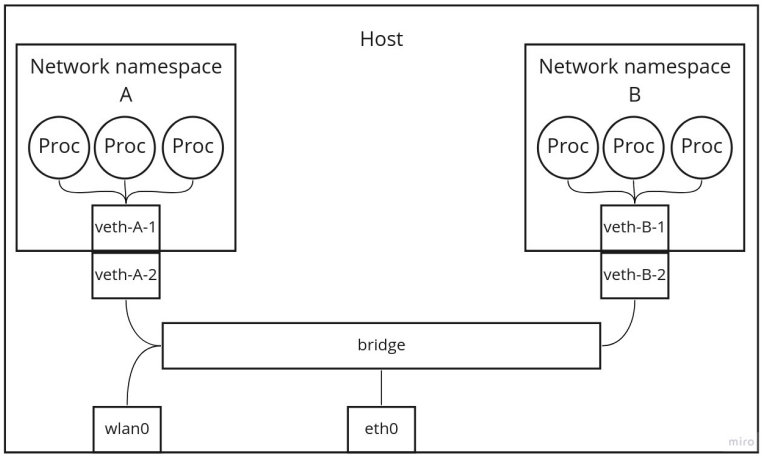

Figure 2.3: Network architecture that enables inter-container communication via bridge and virtual ethernet devices.

In it, two network namespaces, and a wlan interface inside the root namespace, can exchange data over the bridge. Code snippet (2.4) shows how to configure such a network.

```
1  $ ip netns add A
2  $ ip netns add B
3  $ ip link add veth-A-1 netns A type veth peer name veth-A-2
4  $ ip link add veth-B-1 netns B type veth peer name veth-B-2
5  $ ip netns exec A ip addr add "192.168.1.1/24" dev veth-A-1
6  $ ip netns exec B ip addr add "192.168.1.2/24" dev veth-B-1
7  $ ip netns exec A ip link set veth-A-1 up
8  $ ip netns exec B ip link set veth-B-1 up
9  $ ip link set veth-A-2 up
10 $ ip link set veth-B-2 up
11 $ ip link add name br-AB-0 type bridge
12 $ ip link set br-AB-0 up
13 $ ip link set veth-A-2 master br-AB-0
14 $ ip link set veth-B-2 master br-AB-0
15 $ ip addr add 192.168.1.10/24 brd 192.168.1.255 dev br-AB-0
```

Code snippet 2.4: Commands for configuring the network as shown in Figure (2.3)

On lines 1-2, two network namespaces A and B are created. On line 3, a virtual ethernet cable is created, whose ends are represented by `veth-A-1` and `veth-A-2`. The former is injected into namespace A, whilst the latter is kept in the root namespace. The same process is executed on line 4 for namespace B. The isolated ends of the virtual devices are initialised and assigned unique IP addresses that are a part of the same logical network. The other ends are initialised as well. On lines 11-14 a bridge is created and the virtual ethernet cables are attached to it. At this point, A and B can communicate effectively with each other. The host, however, interprets

the bridge solely as an L2 packet forwarder that cannot interact with the other namespaces. On line 15, the bridge is made a participant of the isolated network by being assigned an IP address. Now, both namespaces and the host can communicate with each other. However, a mechanism for routing traffic from the bridge to a different node in the host network or the internet is missing.

2.2.5 Interprocess communication namespace

An interprocess communication (IPC) namespace encapsulates System V objects and POSIX message queues and makes them visible only to its resident processes. Every IPC namespace has a dedicated POSIX message queue filesystem that cannot be accessed by processes outside the namespace. When all processes in an IPC namespace terminate, all IPC objects are automatically destroyed.

2.2.6 UNIX time sharing namespace

UNIX Time-Sharing (uts) namespaces encapsulate the system's host name and domain name. When creating a new uts namespace, the kernel simply copies the system identifiers of the previous namespace and exposes the copied versions to all residents of the namespace. The residents are free to change both identifiers without affecting the actual values on the host. Example (2.1) already highlights how to detach a shell into a new uts namespace and set its host name.

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" [Ini16c, 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.

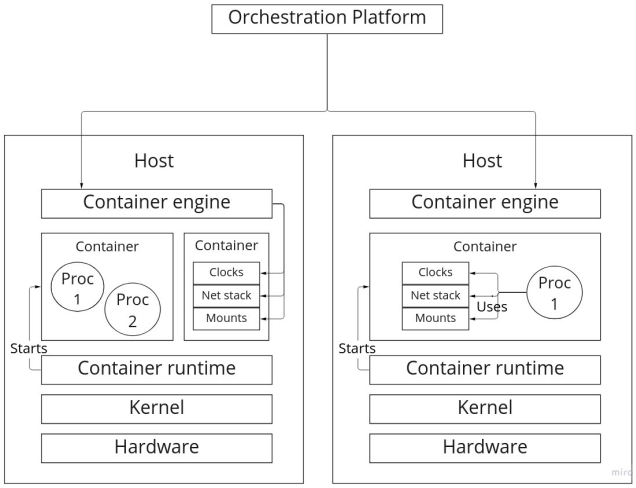

Figure 3.1: Operating system virtualisation architecture using containers. The container runtime starts containers on a single host. A user process can see a bundle of resources allocated to it by the kernel. The kernel guarantees that a process cannot see any other resources. The container engine manages 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

Concept

This chapter introduces the functional and non-functional requirements of two applications - the container runtime and the benchmarking tool. In addition, architectural diagrams are provided and discussed. Example usage of both applications is shown. This chapter also contains a thorough justification of the workloads deployed by the benchmarking tool as well as the variables used to measure the performance of the sandboxed workloads.

4.1 Container runtime

The container runtime is the component responsible for wrapping a user-defined binary in a sandbox. It will be used by the benchmarking tool to wrap workloads in sandboxes.

4.1.1 Functional requirements

The Open Containers Initiative standardises the operations [Ini16a] that a container runtime needs to support.

i The container runtime must provide clients with state information for a container given its unique identifier.

ii The container runtime must provide clients with the ability to create a new container. Users must supply the runtime with a unique identifier for the container and a path to a container bundle. The latter consists of a root filesystem and a configuration file that defines, amongst other things, the path to the user-defined binary and the set of namespaces it will reside in.

iii The container runtime must provide clients with the ability to start a container. Users must provide the unique identifier of the container to start. This operation must execute the user-defined binary in the sandboxed environment.

iv The container runtime must allow users to kill the container process. Users are required to provide the unique identifier of the container and the signal to be sent to the container.

v The container runtime must allow users to delete the container. The delete operation must remove all resources allocated in (ii)

vi The container runtime must allow external applications to hook into well-defined points of a container's lifecycle.

It is important to note that the container runtime's only responsibility is to sandbox processes. An external application, such as the benchmarking tool discussed later, must have the ability to interact with the runtime for the purpose of configuring the sandboxed environment. This includes operations such as creating network topologies that interconnect different containers or allocating shared filesystems. From this, requirement (vi) has been derived.

4.1.2 Non-functional requirements

All requirements specified in this section are ranked in order of importance.

i The container runtime must not pollute or damage the host system via its operation. The runtime manages sensible resources, such as mount points and devices. It must in no way cause side effects on the host, leading to operational failure. This is also an important factor for allowing reproducibility of the work. Users must be able to use the runtime without fear of damaging their system.

ii The container runtime must support unprivileged containers, i.e containers that run without root privileges on the host system. This is in and of itself a functional requirement for running containers in multitenant environments.

iii The container runtime must be implemented in a programming language with manual memory management. Satisfying this requirement will ensure that future work aimed at measuring container boot times will not be hindered by unpredictable perturbations introduced by a garbage collector.

iv The container runtime must consist of a library component and an executable component. This will allow users to implement their own abstractions on top of the library for other research-related purposes.

4.1.3 Architecture

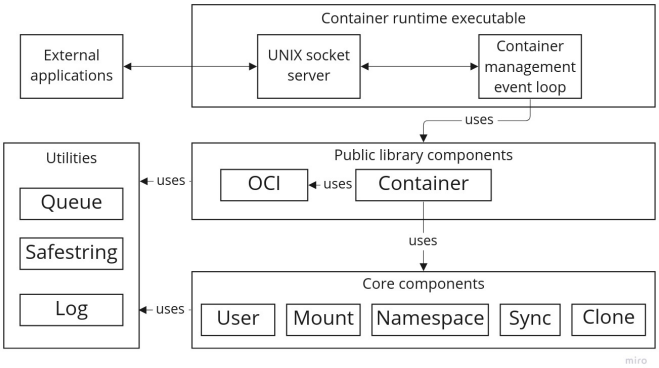

Figure 4.1: High-level overview of the runtime architecture

The architecture of the runtime is shown in Figure 4.1. It consists of a library and an executable that uses it to provide container management services to external applications. The library is split into three parts - utilities, core components and public components.

The utilities provide common data structures such as linked lists and queues. They also contain procedures for safe string manipulation, a logger, and a multitude of helper macros that enable the safe management of resources such as file descriptors and heap memory. The core components directly interface with the kernel. They are responsible for configuring the container, i.e the execution environment of the user-defined application. On top, the public library components use the core components to provide a simple interface for creating, starting, killing and deleting containers. A container is represented as an opaque pointer and is only allowed to be accessed through library functions.

The runtime executable uses the container component to create and manage an in-memory queue of containers. Furthermore, an additional thread implements an event loop that monitors state changes of all containers in the run queue and reaps their resident processes to avoid leaking identifiers. The main thread of execution implements a UNIX socket server that accepts requests from external applications, e.g the benchmark tool, to satisfy the functional requirements defined in the previous paragraph.

When a process is detached into a new user namespace, it's real user identifier is replaced by the kernel with the overflow identifier, also known as the *nobody* user, which has no access to file objects that are not world readable and writable. For this reason, the container runtime must map a range of user identifers from the root user namespace into a range of user identifiers in the user namespace of the process. The user component in Figure 4.1 is responsible for this functionality.

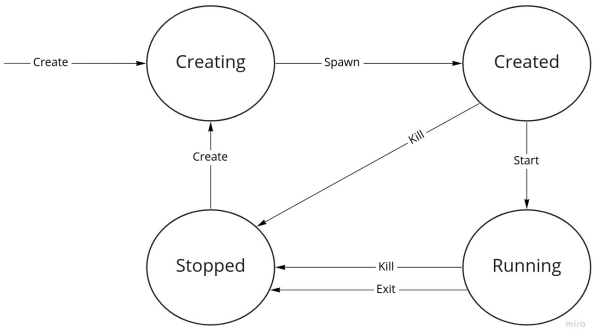

Figure 4.2: State transition diagram of container states. Every state transition is triggered by an operation, whose name is specified in the arrow.

Every container has a dedicated root filesystem with its own set of device nodes, pseudo filesystems, applications and libraries. The mount component in Figure 4.1 atomically changes the container's root mount to a user-defined directory that holds its new root file system. It then creates private mount points for all necessary pseudo filesystems, e.g `proc`, `sys`, and `mqueue`. In addition, this component creates private device nodes for the container such as `/dev/null` and `/dev/urandom`.

The namespace component simply wraps some of the namespace system calls and provides a mechanism to enumerate a contiguous sequence of namespaces.

The clone component wraps the raw `clone3()` system call and the `glibc` wrapper into a portable (at least for `x86_64` and `aarch64`) set of functions.

The Open Containers Initiative (OCI) component is responsible for parsing a JSON configuration file that defines the container's execution environment. Code snippet 8.1 is an example of such a file. In addition, this component provides a mechanism for executing an arbitrary program, called a hook, that receives the container's state through its standard input stream as a JSON string. Hooks are provided by external applications as part of the JSON configuration file and are executed when a container transitions into a new state. A container's state transition diagram is shown in Figure 4.2. For each unique state transition, a set of hooks gets executed. This design is defined as part of the runtime specification [Ini16b].

The synchronisation component defines an inter-process communication mechanism between the container runtime and a container. The typical communication flow follows the request-response pattern - one end produces a request and blocks until the other end consumes it and produces a response. However, to enable concurrent work, a peer can "fire" a message and forget about the response. The requests and responses are simple numeric values with semantic meaning.

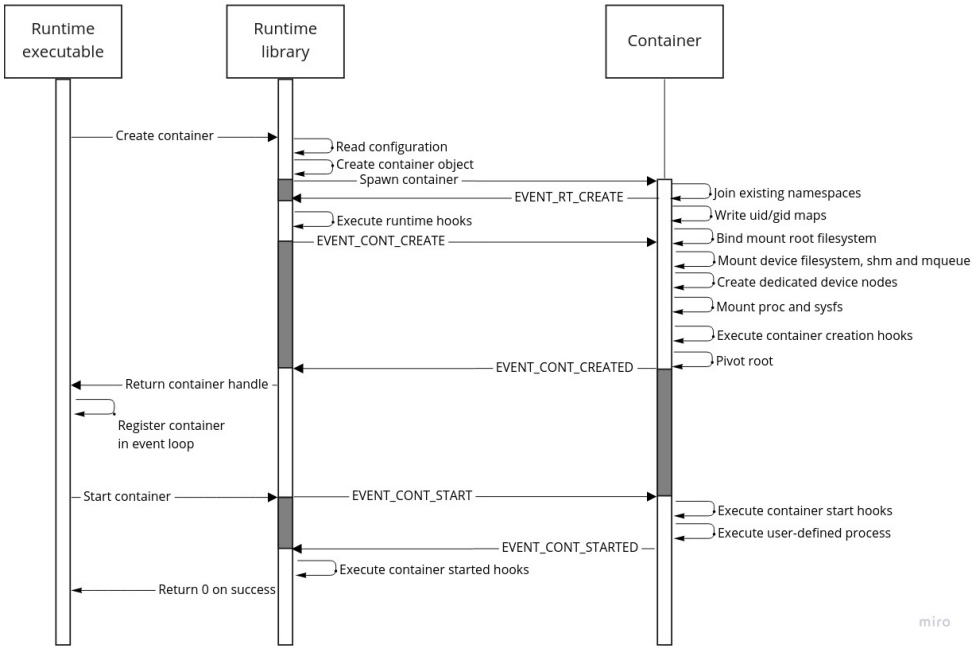

Figure 4.3: Sequence diagram highlighting the container creation and start operations

The container component brings all of the core components together. The sequence diagram in Figure 4.3 shows the procedures executed to create and start a container. The runtime executable invokes the create operation, as defined in requirement (ii). The container component reads the configuration file and allocates an in-memory representation for the container. It then spawns the container process, which creates a freshly-allocated set of namespaces and joins already existing namespaces, depending on the namespace configuration specified in the file. It then transitions into the "creating" state and notifies the runtime to execute the corresponding hooks by sending an `EVENT_RT_CREATE` through the synchronisation component. The runtime receives the request, executes the runtime hooks, and notifies the child that it should transition into the "created" state by executing the container creation hooks and atomically swapping the old root mount with the new root filesystem. The container process then blocks indefinitely until it is killed or it receives a start request from the runtime executable. The latter will trigger yet another hook execution procedure, it will move the container into the "running" state and will afterwards run the user-defined binary. This is detected by the runtime, a set of post-start hooks is triggered, and the runtime executable is notified of the operation's completion.

After the container is created, the executable is responsible for managing the container process. It does so by registering the container with an event loop that polls container states and reaps processes that have terminated, which happens when the container transitions into the "stopped" state - either by exiting or being manually killed by the executable through a signal.

4.2 Benchmark

Benchmarking is a tool to test performance in a controlled way, allowing different system and software configurations to be compared and analysed.

4.2.1 Functional requirements

4.2.2 Non-functional requirements

i The benchmark must be repeatable to facilitate comparisons

ii The benchmark must be observable, either through visualisations or formatted text outputs, so that it can be analysed

4.2.3 Methodology

4.2.4 Workloads

4.2.5 Command-line interface

Chapter 5

Implementation

Chapter 6

Results

Chapter 7

Conclusion

Chapter 8

Appendix A

```
{
    "process": {
        "args": [
        "nsbench-disk-workload"
        ],
        "cwd": "/"
    },
    "root": {
        "path": "./workloads/rootfs/disk",
        "readonly": false
    },
    "namespaces": [
        {
        "type": "user"
        },
        {
        "type": "net"
        },
        {
        "type": "ipc"
        },
        {
        "type": "mnt"
        },
        {
        "type": "uts"
        },
        {
        "type": "pid"
        }
    ],
    "uid_mappings": [
        {
```

```
        "container_id": 0,
        "host_id": 1000,
        "size": 1
        }
    ],
    "gid_mappings": [
        {
        "container_id": 0,
        "host_id": 1000,
        "size": 1
        }
    ],
    "hostname": "nsbench-disk-workload",
    "hooks": {
        "on_runtime_create": [
            {
                "path": "/usr/bin/setup-network",
                "args": ["-a", "192.168.0.101"],
                "env": ["PATH=/usr/bin"],
                "timeout": 2
            }
        ],
        "on_container_created": [],
        "on_container_start": [],
        "on_container_started": [],
        "on_container_stopped": []
    }
}
```

Code snippet 8.1: Open Containers Initiative Configuration File Example

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).

[Ini16a] Open Container Initiative. *Operations*. 2016. URL: https://github.com/opencontainers/runtime-spec/blob/main/runtime.md#operations (visited on 08/20/2022).

[Ini16b] Open Container Initiative. *Operations*. 2016. URL: https://github.com/opencontainers/runtime-spec/blob/main/runtime.md#lifecycle (visited on 08/20/2022).

[Ini16c] 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).