POLITECNICO DI TORINO

Master degree course in Computer Engineering

Master Degree Thesis

OS-level virtualization with Linux containers: process isolation mechanisms and performance analysis of last generation container runtimes

Supervisor
Prof. Gianpiero Cabodi

Candidates
Gabriele Minì
Student ID: 257117
Davide Antonino Giorgio
Student ID: 262709

Accademic year 2019-2020

Abstract

The evolution of the IT world towards the massive adoption of cloud solutions has required the development of virtualization technologies to meet the needs of software scalability and portability. This has led to the emergence of new approaches to virtualization that adapt to these needs, thus efficiently supporting the architecture of distributed applications based on microservices. Our thesis work consists in the study of OS-level virtualization, in particular focusing on its implementation through the Linux kernel. We analyzed the in-kernel mechanisms that ensure process isolation, such as namespaces and cgroups. Using these we have developed a tool that allows the creation of an isolated environment from the host system for the execution of a process, implementing the following features: isolation and limitation of system resources through the use of namespaces and cgroups, system calls filtering through the use of seccomp and privileges management through Linux capabilities. Compared to traditional Hypervisor-based approaches, virtualization at the operating system level takes advantage of a shared kernel between the host machine and guest containers, reducing the overhead introduced by the management of a virtualized host kernel. By no longer using an Hypervisor (or Virtual Machine Monitor), container isolation is guaranteed thanks to isolation mechanisms offered by the kernel itself. We could therefore say that a Virtual Machine is to an Hypervisor as a container is to the kernel of the host machine. However, these solutions raise new security issues due to the use of a shared kernel. In recent years have been developed hybrid technologies that integrate the isolation obtained by traditional virtualization with the advantages of OS-level virtualization. These have found particular application in the development of the container runtimes, the layer of software that deals with the effective creation of the containers inside of tools like Docker or Podman. Our thesis work continues with the analysis of the overhead introduced by the adoption of hybrid solutions within runtime containers, both on performance and on the use of system resources. For this study, several applications executed within containers were considered and, as the runtime container varied, their performance was evaluated. Two hybrid runtimes (gVisor, Kata-runtime) and two traditional solutions (runc, crun) were chosen to perform the tests.

Contents

List of Tables

List of Figures

Chapter 1

Virtualization

The goal of this chapter is to give an overview of the different types of virtualization approaches and methodologies that allow the creation of a virtual environment. We will analyze which are the main problems that the implementation of virtualization has to face and their possible solutions.

1.1 Types of virtualization

It is not simple to give a single definition of virtualization since the are many definitions of it in literature [3] . Each definition is related to different aspects and reflects the main characteristics of each type of virtualization. Amit Singh defined it as: *"Virtualization is a framework or methodology of dividing the resources of a computer into multiple execution environments, by applying one or more concepts or technologies such as hardware and software partitioning, time-sharing, partial or complete machine simulation, emulation, quality of service, and many others."* [4]. This is one of the most general definition of virtualization which embrace concepts that belong to different ways of implementing virtualization as we will see later. The definition given by Joshua S. White and Adam W. Pilbeam focuses more on security, defining virtualization as: *"Virtualization is a mechanism permitting a single physical computer to run sets of code independently and in isolation from other sets."* [5]. This definition is closer to the notion of software containers, being a way of isolating the execution context of different applications. By creating different virtual environments we are ensuring that processes in a specific virtual context cannot tamper with processes in another one. This is a key feature for security, since today we have thousands of different applications that run together on a single server. Actually there are different approaches to virtualization that can be grouped in three main categories:

- **Server virtualization**
  Also known as CPU, machine or system virtualization [3]. It is the possibility to create complete virtual environment, called Virtual Machine (VM), that is able to run it's own operating system. With server virtualization we can have multiple operating systems running on a single machine, working in isolation from each other.

- **Resource virtualization**
  In this case it's a specific resource of the host which is virtualized. Resources can be both software, like domain names or certificates, or hardware, like memory, storage and network [7]. Depending on the resource which is virtualized we can distinguish between:

  – **Storage virtualization**
    Used to create a software abstraction of the physical storage system. In this way we can put together different physical drives to form a single logical entity.

  - **Memory virtualization**
    In this case what is being virtualized is the physical memory. This is done by adding an additional level to addresses translation. A virtual address refers to the memory seen by the guest applications whereas the physical memory is what the Guest O.S operates on. Next we have the machine memory which is the memory managed by the Virtual Machine Monitor and that the underlying machine has.
  - **Network virtualization**
    With Network virtualization both hardware and software resources can be puth together into a single logical entity in software [8]. This approach can happens between different physical networks, called external network virtualization, or to provide software based network functionality to a guest operating system, referred as internal network virtualization. An example of external network virtualization are VLANs whereas the creation of virtual network interfaces are the results of internal network virtualization.

- **Application virtualization**
  Application virtualization can be implemented in different contexts and it usually refers to two main cases [3]. A first definition of application virtualization is linked with the possibility of running an application on a local machine without actually installing it. The application is dislocated in a remote server and it is accessed through the network. The application runs in a small virtual environment with only the resources that are needed for its execution. In this way each user has its own isolated virtual environment where the application runs. On the other hand, a second definition of application virtualization [5] refers to the creation of a small virtual environment on a client machine that allows to emulate the execution context for the application. An example is the Java Virtual Machine (JVM) which allows the same java program to be run on different hosts, for example running a java application on both Linux and Windows.

Although there are different types of virtualization, we will focus on server virtualization, in which thanks to the addition of a software layer [7] called hypervisor (or VMM), multiple virtual environments (or VM) can be created on the same physical machine, each with its own operating system and set of applications.

1.2 Hypervisors

The essential component of server virtualization is called Virtual Machine Monitor (VMM) or Hypervisor. It allows to create a virtual environment on the host machine called Virtual Machine (VM). On top of it is possible to execute a set of virtualized guest operating systems that can share the hardware resource offered by the host.
Usually it is possible to classify Hypervisors in two different main types:

1. Type 1 - Native or bare metal

2. Type 2 - Hosted

According to *Robert P. Goldberg* and *Gerald J. Popek* [2] a VMM must respect three main characteristics:

1. **Equivalence**
   The VMM must provide an environment to the VMs which is essentially identical to the physical machine. With the term "essentially identical" it is intended that any program that run on top of the VMM should have an effect equal to that exhibited if the program had been executed on the physical machine directly. In any case there could be some exceptions caused by the availability of system resources due to the fact that we can have multiple virtual machines running concurrently.

2. **Efficiency**
   This characteristic requires that most of the virtual processor's instructions must be executed directly by the physical processor, without any software intervention by the Hypervisor. This requirement excludes emulators and simulators from the Hypervisor club.

3. **Resource control**
   The VMM must have complete control over resources. This statement is satisfied if:

   - A program running inside a virtual machine on top of the VMM can't access resources that were not explicitly allocated to it.
   - The VMM has the possiblity, under some circumstances, to regain control over the resources previously allocated.

1.2.1 Type 1 - Native or bare metal

Native Hypervisors run directly on the host's hardware. Adopting a bare metal architecture's schema, the Hypervisor runs directly on the hardware rather than going through an host operating system. A bare metal Hypervisor is more efficient than a hosted one having greater robustness and performance. Examples of this architecture are Microsoft Hyper-V, VMWare ESX Server and Xen.

1.2.2 Type 2 - Hosted

Hosted hypervisors are designed to run on top of a traditional operating system. In this case the VMM maintains a software-level representation of a physical hardware. As shown in 1.1, running in ring 3 as a normal application, all the actions that need to be performed (IO, memory management ecc ...) are mediated by the host operating system, which represents an additional software level beneath the VMM. Oracle VM VirtualBox is an example of a hosted hypervisor.

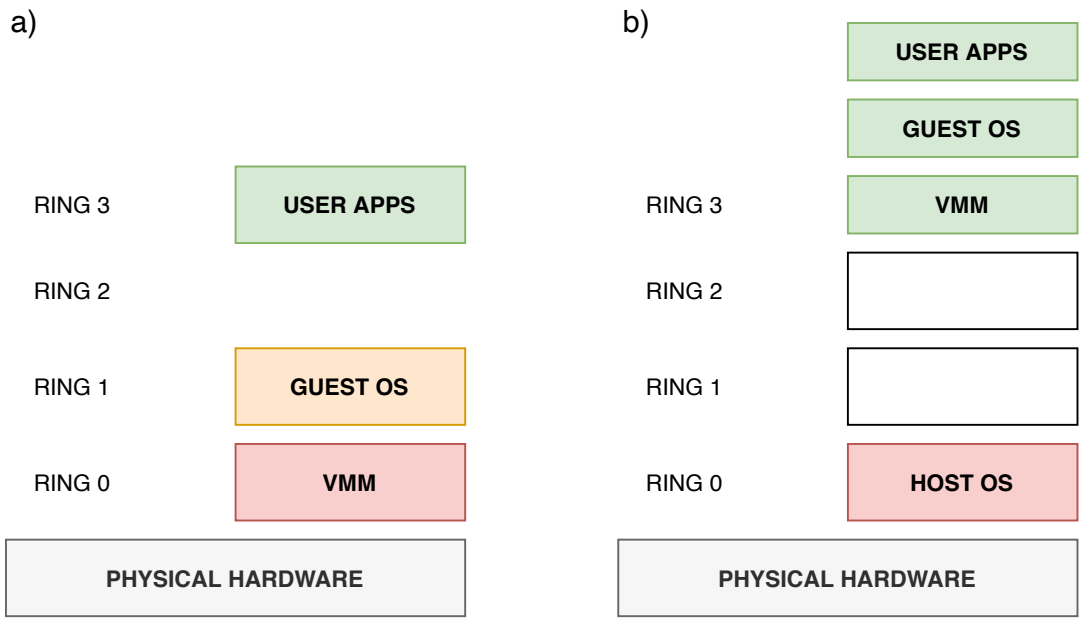

Figure 1.1. a. Bare metal Hypervisor , b. Hosted hypervisor (inspired from [6])

1.3 Virtualization requirements

Before introducing the different approaches to server virtualization it is important to understand the concept of "Virtualizable processor".
As described in a paper from 1974 by Goldberg and Popek [2], an instruction set is composed of three main types of instructions:

- **Control sensitive**
  An instruction is regarded as control sensitive if it accesses low-level machine resources (accessing devices, changing virtual to physical memory mapping ...) that should be managed by an OS or VMM.
- **Behaviour sensitive**
  Instructions whose behaviour is different depending on the resource's configuration. It is possible to distinguish between:
  - **Location sensitive**
    The instruction's behavior depends on its location in real memory.
  - **Mode sensitive**
    The instruction's behavior depends by the CPU mode (kernel or user mode).
- **Privileged**
  These are the instructions that can cause a *trap* when executed in a ring different from Ring 0.

According to the definitions given above, an instruction is generally referred as sensitive if it is either control sensitive or behavior sensitive. If an instruction is not generally sensitive then it is not problematic. Thanks to this latter definition, we can enunciate the following theorem:

Theorem 1 (Virtualizable CPU) *a CPU is defined as virtualizable if and only if the set of all sensitive instructions are a subset of the set of privileged instructions.*

This leads to the necessity of having all the sensitive instructions being privileged as well in order to cause a trap. In this way when a sensitive and privileged instruction is executed, a trap occurs and, by switching to kernel mode, the control is passed back to the VMM.

Problems arise because, to allow virtualization, the guest O.S must be moved from Ring0 to a lower privilege Ring, to let the Hypervisor run at

the lowest level (Ring0). If an instruction is sensitive but no privileged, it's execution behaviour may change since now it is no more executed at Ring0.

Additionally, not all the CPUs have been designed to support virtualization and so they do not respect Theorem 1. For example the famous Intel x86 architecture is not virtualizable.
As an example, the followings are some problems related to the the x86 architecture, as reported in [6]:

- **push** instruction is used to store a value into the stack. A register called *code segment* register has two bits that indicates the privilege level. A guest OS could use this instruction to understand that it is not executed in Ring 0. To solve this problem, *push* should cause a trap giving control to the VMM that will force a fake value of CPL (current privilege level).
- **pushf/popf** are used to manage (read or write) a register called %eflags. This contains a bit to enable interrupts. The problem is that if these instructions are executed in Ring 1 they will fail without raising a trap. Instead we would like to have these instructions to trap in order to respect the requirements of a virtualizable CPU architecture.

The different approaches to virtualization that will be described in the next section are solutions that can be adopted to make a CPU architecture virtualizable accordingly to the Goldberg and Popek requirement.

1.4 Approaches to server virtualization

To address server virtualization different approaches can be adopted:

1. Full virtualization
2. Para virtualization
3. Hybrid virtualization

Each of this solution has different scope and implementation methodology that differs in several aspects.
Our next discussion is based on two assumption:

- When we talk about virtualization solutions, we assume that the guest and the host use the same Instruction Set Architecture (ISA). There are possible also other scenarios, in which the guest and host ISA are different, but they are more closer to the emulation world. With emulation

each instruction of the guest code need to be translated, denying any form of direct execution. Conversely virtualization is implemented by abstracting the system ISA to support multiple guest OS'es.

- We are focusing on CPU virtualization, so all those techniques that makes a CPU architecture virtualizable. Each solution that will be introduced has a wider scopes, including memory, I/O and device virtualization. Actually this is not our focus, we will only make some hints about those topics comparing the efficiency of the different solutions.

1.4.1 Full Virtualization

With full virtualization a complete and isolated versions of the entire computer is virtualized, including all the hardware like CPU, memory, and I/O device,s allowing a guest OS to be run in isolation without any modification [8]. From the virtual machine point of view is like running on real hardware and the guest operating system does not notice the difference. In this way any application or complete operating system that is capable of running directly on the physical host is able to run inside a Virtual Machine.
There are different way to implement full virtualization that can be grouped in two major family [10]:

1. Software based
    - Hosted Interpretation
    - Trap and emulate with direct execution
    - Binary translation with direct execution
2. Hardware assisted

Hosted interpretation

As described in [6], this approach uses an hosted Hypervisor architecture since the VMM will run as a regular application on top of the host operating system. The hypervisor is in charge of abstracting the hardware in a software representation. The VMM reads each instruction of the code of the virtual machine and updates the hardware's state accordingly. Below is provided an example of the interpreter pseudo code.

```
while(1) {
    instr = fetch(virtHw.PC);
    virtHw.PC += 4;
switch(decode(instr)) {
    case ADD:
    //execute
        int sum = virtHw.regs[instr.r0] + virtHw.regs[instr.r1];
        virtHw.regs[instr.r0] = sum;
        break;
    case SUB:
//...etc...
```

Listing 1.1. Pseudo code of hosted interpreter (inspired from [6])

In this way the interpreter, maintaining a software representation of the hardware, prevents the guest OS from reading the current privilege level (CPL) of the physical CPU and can correctly implement the popf instruction, which would not trap, by referencing the virtual CPU state. The main advantage of this approach is the ease of management of privileged instructions. The interpreter has the possibility to handle those instructions in accord to different policy. Moreover, due to the fact that the hypervisor runs on top of the host O.S as a regular application, the host desktop operating system can still work concurrently with the guest O.S. Conversely this approach has bad performance due to the presence of the interpreter. The fetch-decode-execute cycle of the interpreter may use too much physical instruction for each guest instruction.

Actually this approach is mostly used by emulators such as BOCHS. In fact, no direct execution is allowed since every guest instruction will be emulated by a fetch-decode-execute cycle at software level, causing worse performance. This is needed because emulators theoretically allow to run guest code written for a totally different hardware architecture than the host machine. Anyway, despite the performances, this approach can also be a way of virtualizing a guest OS written for the same hardware as the host.

Trap and emulate with direct execution

The Ravello Community at Oracle [13] explains how this approach can be used to run directly on the CPU all those instructions (from the guest code) that can be executed in user mode. Instead, a privileged instruction will cause a trap giving the control back to the Hypervisor, which is in charge of emulating it and then continuing the execution.

The figure 1.2 below gives an overview of how trap and emulate actually works. The following example is inspired from [16]. The Guest tries to change

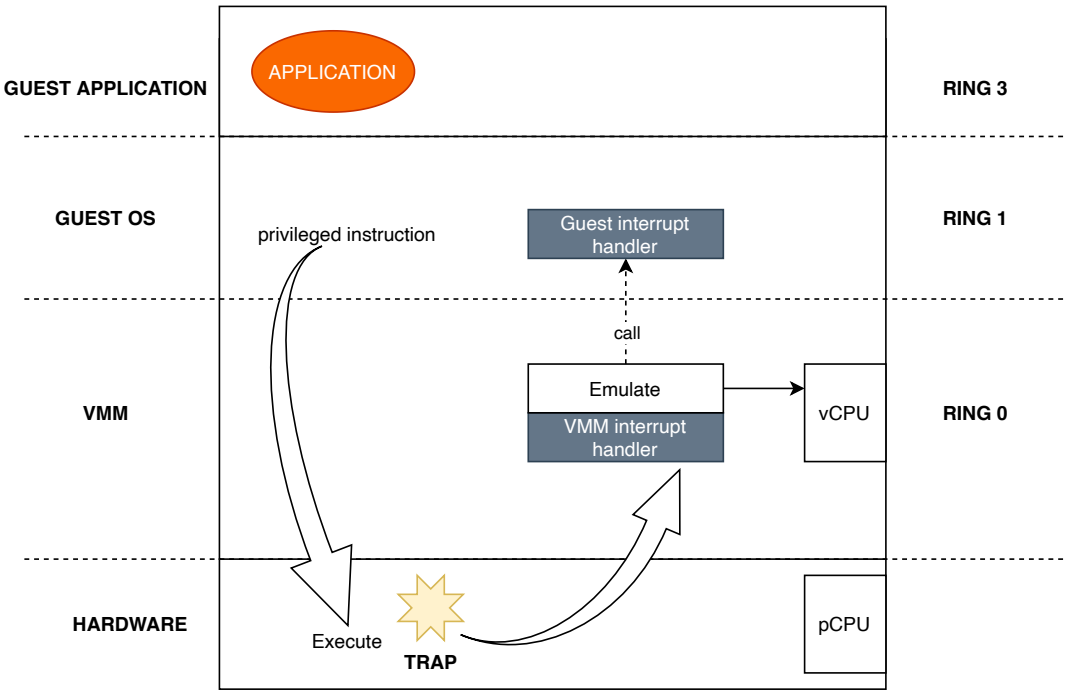

Figure 1.2. Trap-and-emulate approach to CPU virtualization

CPU flags which is a privileged operation. Doing this it would violate the ring protection because it is trying to access the hardware directly. Since the instruction is privileged and the guest kernel is running in Ring 1, a trap occurs.
To fool the guest of running in Ring 0, the VMM will emulate the instruction and set the new flags for guest's virtual CPU. This illusion is enforced by calling the guest's interrupt handler, so for the guest OS is like having the control of the hardware.
The main advantage of this approach is that most of the instructions do not require to trap and can be executed directly on the CPU in user mode [13]. On the other hand, the main limitation is that the processor must be "virtualizable", so it must respect the Goldberg-Popek requirement as we described before. A CPU architecture that can be completely virtualized with trap-and-emulate, is called "Classically virtualizable". Accordingly to this definition this approach can not be used with the x86 architecture due to the problem of sensitive but unprivileged instructions. For this reason

two new techniques has been introduced to implement the trap mechanism : binary translation and hardware assisted virtualization.

Binary translation with direct execution

The concept of binary translation was introduced as a way to implement virtualization to address processors that do not support the Goldberg-Popek requirement. These CPUs do not have a clean separation of privileged and non-privileged instructions. For example, the Intel x86 CPU line is one of them.
As described here [13] the idea is that the hypervisor reads each instruction of guest code before it runs. If the instruction is not a sensitive one it can be run natively on the physical CPU in user mode. On the other hand if the VMM find a non virtualizable (sensitive but not privileged) instructions, it translates it into something safer (virtualizable). This results in only the kernel code of the guest OS to be translated whereas guest application code can be executed directly. The translated code can be then executed directly by the Guest OS. As a consequence of this approach we get rid of all the sensitive instructions because they will be translated.
Also natively privileged instructions, the ones who would trap, can be managed in an efficient way by the binary translator. They can also be translated in order to avoid a trap and the consequently context switch to the VMM, improving performances with respect to the trap-and-emulate approach. As shown in figure 1.3 the guest OS would like to execute the CLI instruction to clear the IF flag in the EFLAGS register. Clearing the IF flag causes the processor to ignore maskable external interrupts [14]. When the BT reads the Guest OS code before execution, it will translate that instruction into something safer that will affect only virtual hardware (vCPU), allowing the guest OS to execute it in Ring1. It is clear that in this way we are avoiding traps for privileged operations.
However there are still some operations that are hard to manage efficiently. System calls triggered by guest applications, are trapped into the Hypervisor running in Ring 0, which is in charge of giving back control to the kernel of the guest OS. Traps cannot be directly managed by the guest OS for two main reasons:

1. The guest OS has been de-privileged to run in Ring1.

2. We want the VMM to intercept every privileged instruction to emulate it in a safe way.

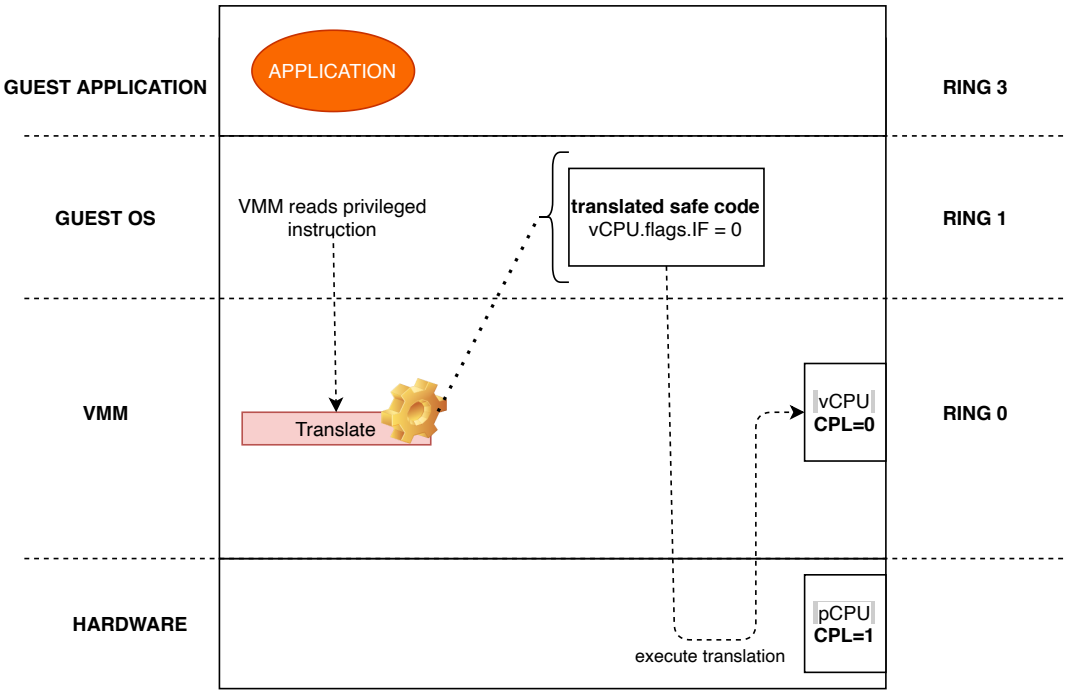

Figure 1.3. Binary translation with direct execution of a privileged instruction

This approach requires multiple context switches for the management of system calls which represents one of the main drawback. The following example is inspired from the article written by Johan De Gelas [17]. As we can see from figure 1.4 when a guest application invokes a system call it will trap to the VMM which is in charge of emulating it, translating the code and then giving back the control to the Guest OS which will execute the translated code. When the translated service routine is finished the guest OS will use SYSEXIT to switch back to the user application. However, being SYSEXIT a privileged operation, it will trap again into the VMM again which will then emulate it.

Also memory management is quite hard to be done in an efficient way. As sustained by Keith Adams and Ole Agesen [10] operations that are not privileged (e.g load and store) can actually cause a trap if they access sensitive data like page table. A simple BT would not be able to avoid those kind of traps, affecting the overall performance, since trapping is an expensive operation especially on modern CPU. For this reason better approaches has been developed with the main goal of reducing the number of traps. This is what is called an adaptive binary translator. The basic idea is that the execution

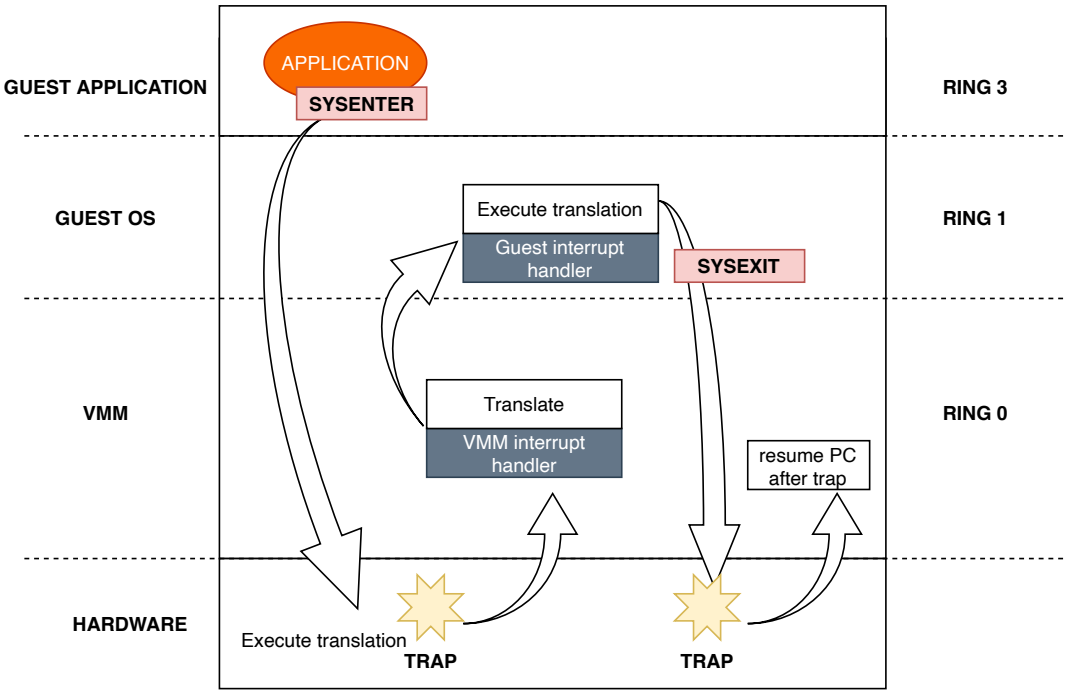

Figure 1.4. Syscall management in a Virtual Machine

of the translated code is analyzed and if some instructions frequently trap the BT will adapt their translator to avoid those traps.

The operation of binary translation can be done in two different ways [13]:

- **Statically**, by translating the executable code of an entire program.
- **Dynamically (on-demand)** grouping the code to be translated in smaller chunks. These are called basic blocks and their main characteristic is that they do not contain branch instructions. The performance of a binary translator are improved by saving the translations in order to reuse them later. In this way the overhead is limited to the first time the block is translated. In addition different blocks can be linked together by exploiting the branch instruction at the end of each chunk. This is called as block chaining.

To conclude, binary translation can be adopted to solve the problem of the stealthy x86 instructions by creating an x86 to x86 binary translator [10] to make the x86 architecture compilant with the oldberg and Popek requirements.

Hardware assisted

Hardware assisted virtualization, also referred as Native Virtualization [8], has been introduced to cope with the limitations of software based approaches to full virtualization, in particular the cost of binary translation. It is supported since 2006 by different generations of CPUs thanks to the introduction of the VT-x and AMD-V technologies from Intel and AMD respectively. Below is described the approach proposed by Intel to hardware assisted virtualization in order to clarify the concepts that make this possible.

The main idea is to add virtualization support to the hardware. To achieve this goal a new ideal ring, called Ring -1 in 1.5, has been added just below Ring 0 in the CPU architecture. There is where the VMM runs.

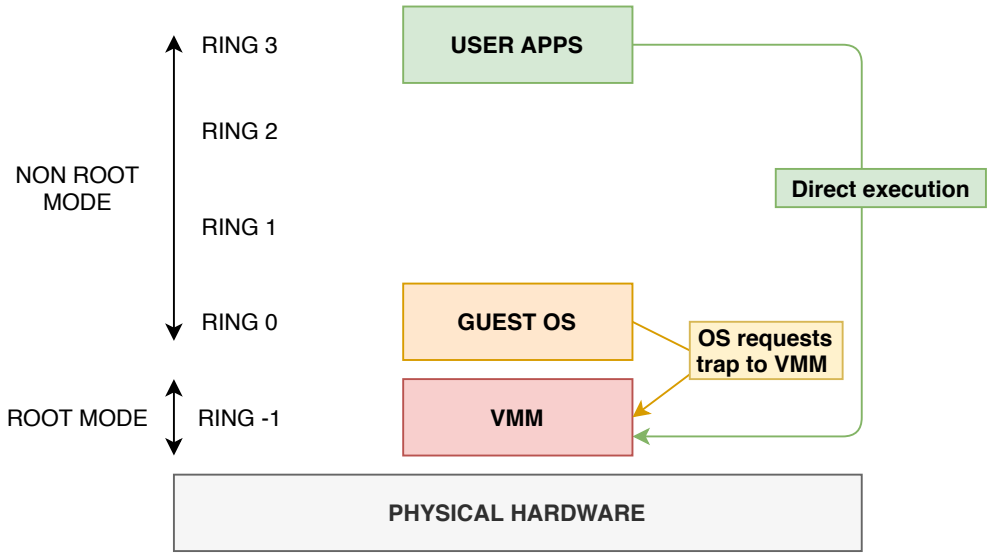

Figure 1.5. Intel VT-x Hardware assisted virtualization architecture (inspired from [8])

This allows the guest OS to be moved in Ring 0, avoiding the problem of running Ring 0 code in higher Rings. In this way it can access the hardware just as it normally would when running directly on a physical host. For this reason the guest OS can be virtualized without any modifications.

Accordingly two new CPU execution mode has been introduced, as described in [15]:

1. **Root mode** In this operational mode, the CPU works like a traditional CPU that does not exploit hardware assisted virtualization. This mode is used to run the Hypervisor.

2. **Non-root mode** This mode reflects the x86 CPU architecture having four privilege rings. Non-root mode is used to run the Guest OS. It's state is represented by the Virtual Machine Control Structure (VMCS) which controls how instructions from the guest OS are executed.

The Oracle corporation explains how all the sensitive operations executed in non-root mode trigger a transition by trapping to root mode, even in Ring 0. There are two different operations that allows to move from root mode to non-root mode and vice versa: "VM entry" and "VM exit". So the basic idea of hardware assisted virtualization is to use a trap-and-emulate but in a efficient way, since now we have hardware support. Even if being at Ring 0 means that the guest OS is using some of the host hardware directly, we still have a virtual machine. For example any I/O operations in Guest Ring 0 trigger a VM exit to let the VMM emulating the device. In particular, the state of the guest virtual machine (the state of its processor, of its memory and of its peripherals) is implemented as VMCS in the host memory. Inside the VMCS is also store control data to to determine which sensitive operations cause non-root code to transition to root code.

The main advantage of Hardware-assisted full virtualization is that now, having the guest OS running in Ring 0 we have no more all the problems related to the execution of sensitive instructions in a privileged Ring different from Ring 0 that were previously addressed by binary translation. Now the guest OS directly trap to the VMM. Thus hardware-assisted virtualization not only satisfy the Popek and Goldberg principle, but also improves performance because the emulation of privileged instructions and traps are managed by the hardware directly [18].

Hardware-assisted virtualization also have some limitations. First of all the overhead of switching between non-root mode to root mode (VM exits) is high affecting the performance for all those devices whose emulation leads to an high number of traps. [15]. Secondly the host CPU must provide the required support to hardware-assisted virtualization which is not available on all processors.

1.4.2 Paravirtualization

Paravirtualization is a virtualization technique that works differently from the full virtualization. The biggest difference is that the guest OS must be modified. This leads to two main consequences, which represent the major advantages of Paravirtualization:

1. Virtual guests know they are virtualized.

2. Only a subset of the complete hardware set of the physical machine is virtualized.

The idea behind Paravirtualization is that the guest OS source code is rewritten to remove instructions that are not virtualizable, allowing it to access resources through a virtual hardware API exposed by the hypervisor [8]. Consequently, unlike full virtualization, it is no more needed to perform the binary translation of unsafe instructions. Any non virtualizable instruction of the guest OS is replaced with calls to the hypervisor, known as *hypercalls*. System calls are also replaced with hypercalls as well. The hypervisor will then perform the task that the guest kernel should have done [19]. The major advantage of this approach is that calls to the hypervisor(hypercalls) do not trap, removing the expensive additional context switches required by binary translation in full virtualization. Moreover, the guest OS can now access hardware resources or perform kernel operation by making a direct request to the Hypervisor, thanks to the virtual hardware interfaces provided to him by the underlying additional sofware layer [11]. In this case is not needed to emulate the entire hardware because the guest OS is aware of being virtualized (differently from full virtualization). This leads to a simpler implementation of the Hypervisor.

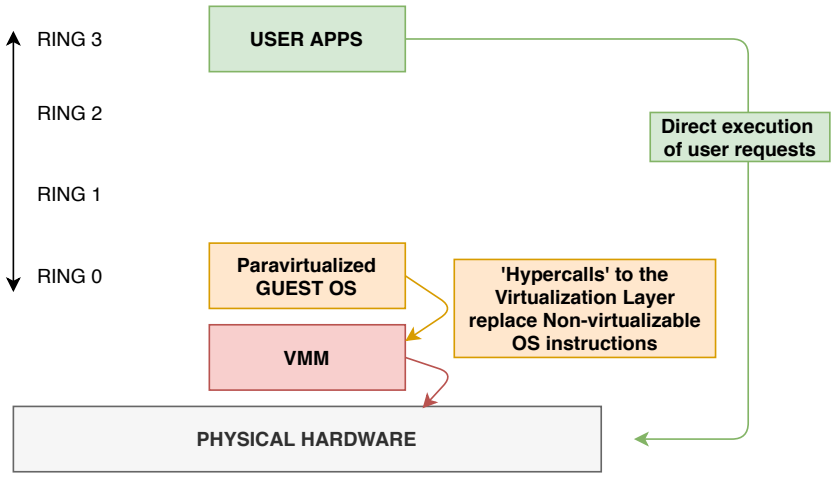

Figure 1.6. Paravirtualization on Intel x86 architecture (inspired from [8])

The goal of para-virtualization technique is to reduce the time needed to perform operations that otherwise would be difficult to execute in a virtual

environment [20]. On the other hand, in order to work, a guest operating system must be modified to take advantage of this technology. If is not possible to change the guest operating system (as is the case with Windows), it will be not possible to use paravirtualization. Solutions like Xen Hypervisor offers the possibilities to execute both a full virtualization or a para-virtualization solution.

1.4.3 Hybrid Virtualization

The basic idea behind hybrid virtualization is to bring together hardware assisted virtualization and Paravirtualization. Different studies [12] [35] have pointed out that hardware assisted virtualization performs better in CPU and memory virtualization whereas a software based approach like paravirtualization exposes the best features for I/O virtualization. An hybrid virtualization approach takes advantages from both these technologies and could be a promising solution since applications' workload is various, resulting in a mix of intensive CPU, memory and I/O operations. The main goal is to merge the best aspects of the two different approaches for performance maximization.

1.4.4 O.S Level Virtualization

Differently from the virtualization techniques described so far, operating system level virtualization uses a shared kernel between the host and the guest. This techniques removes the need of virtualizing the hardware leading to higher speed and performances. On the other hand the shared kernel represents one of the biggest limitation of this approach due to the fact that the guest must use the same OS as the host.

Operating system level virtualization it's originally born as an advanced implementation of the chroot jails. Chroot jails are based on the chroot (change root) command which allows to change the root directory for a process and all its children. In this way we have a process that is confined inside its directory. However, these solutions soon became unsafe. Starting from this idea, O.S level virtualization, also referred as "chroot on steroids", enforce isolation at process level, taking advantages of different isolation mechanisms that are usually built into the kernel itself. The main idea is that now, instead of having a complete virtual environment (what we previously called Virtual Machine), we have lighter and isolated context, usually referred as Container, made of a process or group of processes that has some restrictions

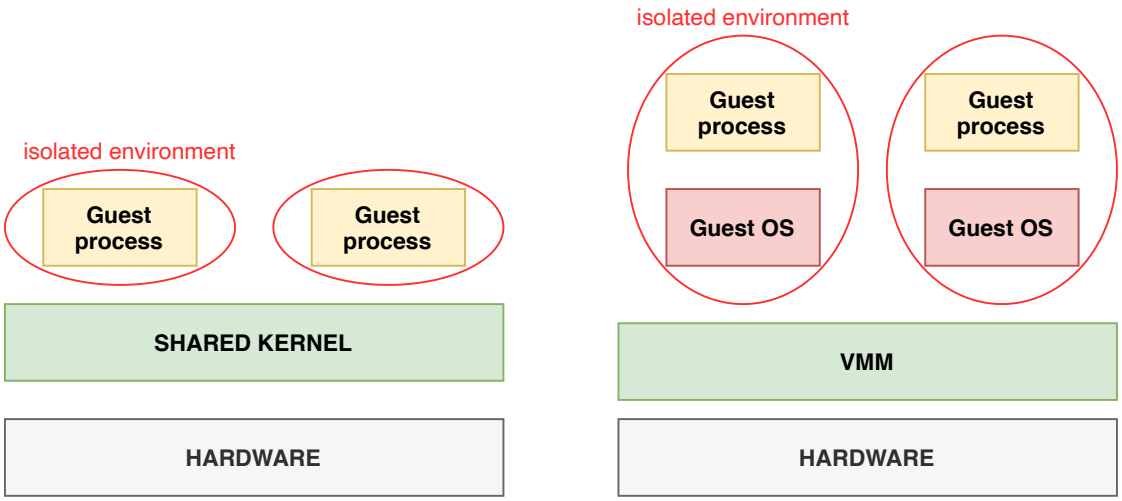

Figure 1.7. O.S virtualization architecture

over the resources that it can see and use. This kind of solution does not use an Hypervisor (or VMM) because the abstraction leverages on the kernel running on the host. Moreover, due to the fact that there is no more the need of emulating the hardware, this approach to virtualization is the less expensive. Generally, it is possible to distinguish two types of container:

- **System Container**
  Also referred as OS container, they expose the same functionalities of a traditional Virtual Machine but they are lightweight. Their goal is to create and environment which is as close as possible to a full operating system but without the need of a separate kernel. For this reason inside a system container we can have multiple processes and services running at the same time.

- **Application container**
  This kind of containers is used to run a single process that represents the service that we want to run. This embraces the microservices architectural pattern which has its concretization in the world of distributed applications.

Containers are easy to move, embracing the "build once run everywhere" paradigm. A container is described by an image (Docker) or template (openVZ), which represents the recipe to create the container. Then you can deploy the image on any host or environment that has a daemon installed to manage

and run containers. Containers come also with some limitations and drawbacks. As said before having a shared kernel means that we are unable to virtualize other O.S (what we have is linux on linux).

Moreover, the shared kernel is a single point of failure. If malicious code can exploit a vulnerability of the host all the containers that run on top of it should be considered compromised.

With respect to other forms of virtualization, The lack of a VMM (or hypervisor) and hardware virtualization leads to security and isolation totally being enforced at software level, increasing the attack surface. Due to the fact that Containers rely upon isolation mechanism that are embedded into the kernel, their specific implementation depends on the operating system on which they are built. Linux has been the most successful in this field and most of the technologies that exploit lightweight virtualization today make use of Linux containers.
However, that are also different implementations of container-like solutions for other operating system:

- **FreeBSD**, a UNIX operating system developed by the University of California, has its own concept of container which is called a freeBSD jail. [23]

- **Oracle Solaris**, developed by Sun Microsystems, introduced its concept of chrooted environment called Solaris Zones. They distinguish between two type of zones [22]:

  - **Global Zone**
    Each Solaris installation has exactly one global zone. It represents the default zone for the operating system and has control over all the processes running inside of it. It is used for administrative control over the system.

  - **Non Global Zone**
    They represent the container concept. Each non global zone is created under the system-wide Global Zone and consists of an isolated virtual environment.

- **Microsoft Windows** also have its own container solutions which are supported by Windows Server. It also supports linux containers ,thanks to the support of docker and a traditional virtual machine, in two different ways [21]:

– **Moby VM**
  This approach creates a complete Virtual Machine where the docker daemon will run and create our Linux Containers. The docker client runs in Windows but make calls into the docker daemon inside the VM.

– **LCOW**
  LCOW stands for Linix Container On Windows. Only a minimal linux kernel, just allowing the creation of containers, is virtualized. In this way each Linux Container has its own minimal virtual machine, in contrast with the Moby VM approach where a complete VM is shared between all the containers.

Chapter 2

Linux containers

A Linux container is no more than a process with some security and isolation mechanism that ensure it has its own view over the system resources and no possibilities to tamper with processes in other namespaces. Today there are several tools and libraries that help us in spawning and managing the lifecycle of containers like Docker, LXC, LXD, LibVirt and many others. Actually container does NOT run on this tools or libraries , they run on the Linux kernel.
The aim of this chapter is to illustrate which are the component of the Linux kernel that allow the creation of a process container and how they actually acts on the view that the container has on the system resources.
All the code presented in this chapter is taken from the Linux Kernel [29] and adapted in order to explain the concept we are interested in.

2.1 Namespaces

Namespaces are a Linux Kernel feature originated in 2002 and mainlined in Linux kernel v2.4.19, that allows to wrap system resources in a level of abstraction capable of isolating a set of system resources for a subset of processes.
Thanks to namespaces is possible to safely execute unknown programs in your server but in isolation from the rest of your system.
There are different types of isolation that can be achieved using namespacing:

1. Mount Namespace
2. Process Namespace
3. Network Namespace
4. IPC
5. UTS (UNIX Time Sharing)
6. User Namespace
7. Control group (cgroup) Namespace
8. Time Namespace

Each type of namespace will then affect some resources instead of others. The following table 2.1 lists all the namespaces, the flags that are associated to them in the linux kernel and which system resources they isolate.

Namespace	FLAG	Isolate
PID	`CLONE_NEWPID`	Process IDs
Network	`CLONE_NEWNET`	Network devices, stacks, ports, etc.
Time	`CLONE_NEWTIME`	Boot and monotonic clock
Mount	`CLONE_NEWNS`	Mount points
IPC	`CLONE_NEWIPC`	System V IPC, POSIX message queues
Cgroup	`CLONE_NEWCGROUP`	Cgroup root directory
User	`CLONE_NEWUSER`	User ID and Group ID
UTS	`CLONE_NEWUTS`	Hostname and NIS domain name

Table 2.1. Namespace overview [24]

2.1.1 Namespaces internals

Linux kernel namespaces are implemented using a dedicated structure called `nsproxy` 2.1 that contains a pointer to each namespace where the current proces resides (except the user namespace). Another parameter that is stored in this structure is a reference counter used to count the number of tasks having a reference to this structure. When two processes share all the namespaces the counter is incremented by one (e.g when fork is executed) [26]. Instead, when a single namespace is created (as a result of clone or unshare), the `nsproxy` is copied. For this reason each namespace has also its own separate reference count which represents the number of nsproxies that point to it and not the number of *tasks* [27], as we can see in figure 2.1. This is needed because sometimes we want to operate on a single namespace.

```
1  struct nsproxy {
2    atomic_t count;
3    struct uts_namespace *uts_ns;
4    struct ipc_namespace *ipc_ns;
5    struct mnt_namespace *mnt_ns;
6    struct pid_namespace *pid_ns_for_children;
7    struct net *net_ns;
8    struct time_namespace *time_ns;
9    struct time_namespace *time_ns_for_children;
10   struct cgroup_namespace *cgroup_ns;
11 };
```

Listing 2.1. `nsproxy` Linux kernel structure in `/include/linux/nsproxy.h`

As can be seen there is no a `pid_ns` entry representing the `pid` namespace of the current *task*, instead what we see is `pid_ns_for_children`. This entry refers to the namespace in which the children of the current task will be created but not the PID namespace of the current *task*. If we call `unshare()` or `setns()` with the flag `CLONE_NEWPID`, these calls do not change the PID namespace of the calling process, but instead the PID namespace for the future children of the calling process. Actually change the PID namespace of the current process is not possible because it would change its own idea of its PID (e.g obtained by `getpid()`) which would cause problems with many applications and libraries [25]. When a process's PID namespace is created, it's membership with a single PID namespace is defined, and cannot be changed anymore. This results in the fact that relationship between processes reflects the parental relationship between PID namespace, as reported in the linux

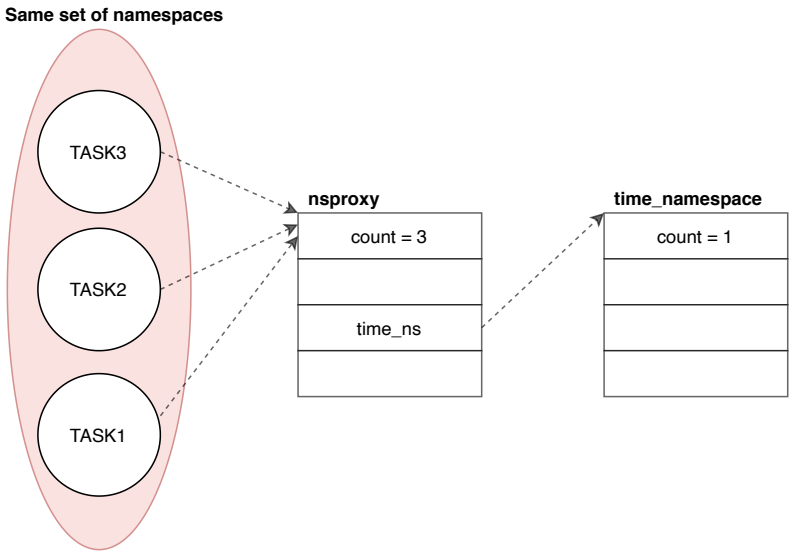

Figure 2.1. Namespace reference count

man of the pid namespace [25]. For this reason we do not find the PID namespace for the current task in `nsproxy` but in the `upid` 2.23 structure instead. As we can see from figure 2.1.1, when `unshare()` is executed, a new PID namespace for children is created but the calling process remains in its original namespace. If we want a new process to enter the new PID namespace we need to `fork`. The same thing applies for the `time_namespace`, in fact we have the `time_ns_for_children` entry in the `nsproxy` structure.
The user namespace is part of another structure called `struct cred` 2.2. This structure represents the security context of a *task*. It stores information about the owner of the process, its capabilities and various other.

```
1 struct cred {
2   [...]
3   struct user_namespace *user_ns;
4   [...]
5 }
```

Listing 2.2. `cred` Linux kernel structure in `include/linux/cred.h`

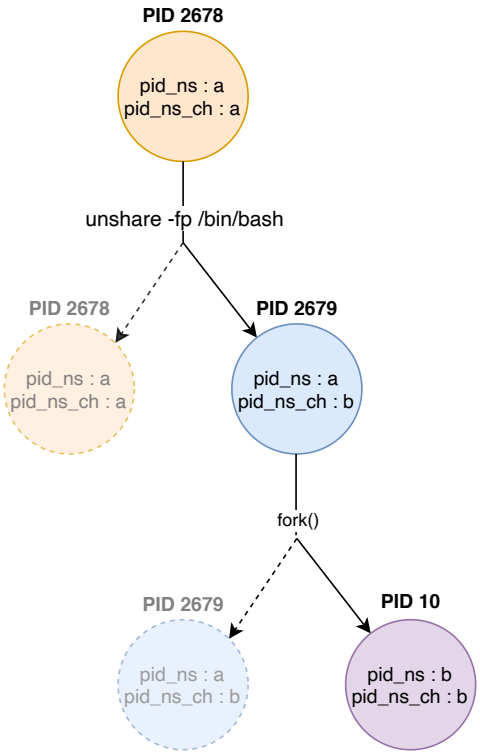

Figure 2.2. Execution of `unshare -pf /bin/bash`

These two data structures are hold together by the `task_struct` 2.3 structure that represents the single *task*. This structure is in charge of provide information like the status of the current *task*, its own `pid`, its parent and so on.

```
1 struct task_struct {
2   [...]
3   /* process credentials */
4   const struct cred __rcu *cred;
5   [...]
6   /* namespaces */
7   struct nsproxy *nsproxy;
8 }
```

Listing 2.3. `task_struct` Linux kernel structure in `include/linux/sched.h`

Namespaces are represented in the Kernel by a structure named `[ns_name]_namespace` 2.4 (for example `time_namespace` for the namespace

of type `TIME_NS`).

```
struct time_namespace {
  [...]
  struct user_namespace *user_ns;    //pointer to user_ns
  struct kref kref;                  //namespace ref count
  struct ns_common  ns;
}
```

Listing 2.4. `struct namespace` example

For most namespace structures there are some common fields:

- A pointer to the `user_namespace` is needed because each operation of the namespace will be controlled and according to the granted permissions accepted, in particular according to the assigned `UID` and `GID`. However, there are a set of other information regarding the user that are stored in the user namespace. When a namespace is created (that is different from the user one), it is automatically owned by the user namespace where the creating process resides. Each privileged operation performed by a task on a resource that is owned by another user namespace is checked by the kernel in order to validate it and check if it owns the required capability in that user namespace.
- A private `reference count`, as explained when introducing the `nsproxy`, to count the number of `nsproxy` structure pointing to it.
- A reference to a `struct ns_common` 2.5 structure.

```
struct ns_common {
  atomic_long_t stashed;                    //dentry
  const struct proc_ns_operations *ops;     //proc operations
  unsigned int inum;                        //inode number
};
```

Listing 2.5. `ns_common` Linux kernel structure in `/include/linux/ns_common.h`

The `ns_common` structure 2.5 collects all the abstractions of namespace in `procfs` as described in [26]. As we will discuss later, each file placed under `/proc/[pid]/ns/` is a *symbolic link* that points to an *inode* in a file system called *nsfs*, only visible to the kernel. Each of these links refers to a namespace for the process.

The field `stashed` is the dentry that is the "*glue*" between this file and the associated *inode number* of the namespace that is represented by the field `inum`. The `ops` field points to the `proc_ns_operations` structure 2.6. It includes the name and the type of the namespace (for example the type of the user namespace is `CLONE_NEWUSER`). Function `get()` is used to increment by one the private reference counter of the namespace and `put()` to decrement it. `install()` is used to install the process into the specified namesapce. It calls `get()` on the namespace we are attaching to and `put()` on the namespace we are detaching from.

```
1 struct proc_ns_operations {
2   const char *name;
3   const char *real_ns_name;
4   int type;
5   struct ns_common *(*get)(struct task_struct *task);
6   void (*put)(struct ns_common *ns);
7   int (*install)(struct nsset *nsset, struct ns_common *ns);
8   [...]
9 }
```

Listing 2.6. `proc_ns_operations` Linux kernel structure in `include/linux/proc_ns.h`

Each namespace will then define it's own `proc_ns_operations`.
Figure 2.3 shows the relational schema of the structures explained so far.

```
1 const struct proc_ns_operations utsns_operations = {
2   .name    = "uts",
3   .type    = CLONE_NEWUTS,
4   .get     = utsns_get,
5   .put     = utsns_put,
6   .install = utsns_install,
7   .owner   = utsns_owner,
8 };
```

Listing 2.7. `utsns_operations` example

Kernel operation over namespace

In `/kernel/nsproxy.c` are reported a set of functions used to operate with namespaces' kernel data structures. Some of these will be analyzed in the

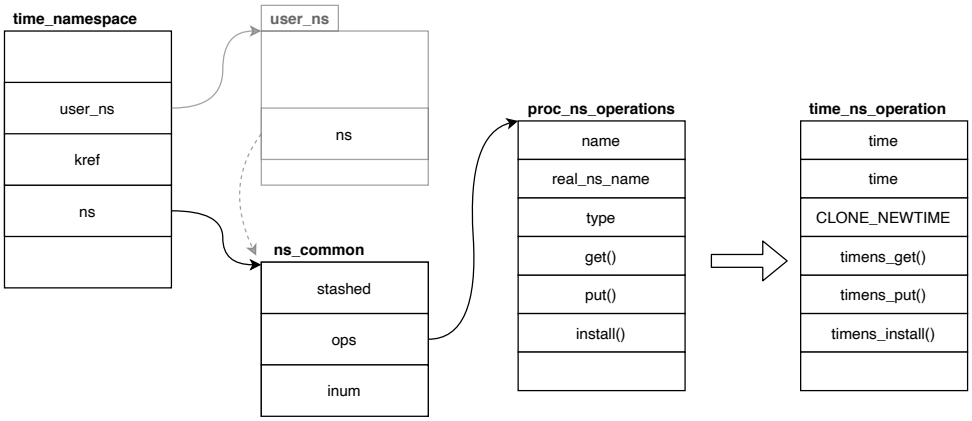

Figure 2.3. `ns_common` data structure relationship

following chapters.

switch_task_namespaces

The `switch_task_namespaces` 2.8 function is responsible for changing the namespace set of a task. This is done by replacing the `nsproxy` structure 2.1 with a new one which refers to the new set of namespaces.

```
1  void switch_task_namespaces(struct task_struct *p,
2                              struct nsproxy *new)
3  {
4    struct nsproxy *ns;
5
6    [...]
7
8    task_lock(p);
9
10   ns = p->nsproxy;
11   p->nsproxy = new;    /* Set the new nsproxy */
12
13   task_unlock(p);
14   /* If we are the last task having a reference to this nsporxy
      * struct then free the struct. */
15   if (ns && atomic_dec_and_test(&ns->count))
16     free_nsproxy(ns);
17 }
```

Listing 2.8. `switch_task_namespaces` in `/kernel/nsproxy.c`

As we said while introducing the `nsproxy` structure 2.1, the reference counter of the old `nsproxy` is atomically decremented since we detached from the old namespace set. If we are the last *task* that shares the old `nsproxy`, the count goes to zero and the old `nsproxy` can be freed.

unshare_nsproxy_namespaces

The `unshare_nsproxy_namespaces()` is in charge of creating a new `nsproxy` 2.1 which will contain a new set of namespace accordingly to the `unshare_flags`. It is used by the `unshare()` system call to detach the calling process from the specified namespaces and enter into a new ones.

```
1  int unshare_nsproxy_namespaces(unsigned long unshare_flags,
2                                 struct nsproxy **new_nsp,
3                                 struct cred *new_cred,
4                                 struct fs_struct *new_fs)
5  {
6    struct user_namespace *user_ns;
7    [...]
8    /* Being not part of the nsproxy struct, the user
     * namespace is create outside this function and stored in
     * the new_cred struct passed as third argumet.
9    */
10   user_ns = new_cred ? new_cred->user_ns : current_user_ns();
11
12   /* Create the new nsproxy holding the new namespaces. */
13   *new_nsp = create_new_namespaces(unshare_flags, current,
14                                    user_ns,
15                                    new_fs ? new_fs : current->fs);
16 }
```

Listing 2.9. `unshare_nsproxy_namespaces` in `/kernel/nsproxy.c`

create_new_namespaces

The `create_new_namespaces` 2.10 function is the core function to create a new set of namespaces. It is in charge of allocating the memory for the new `nsproxy` structure 2.1 and creating the corresponding set of namespaces. Steps that are achieved in this function are:

- create a new `nsproxy` structure through `create_nsproxy()`

- initialize each new namespace by calling functions in the form `copy_[namespace]()`. Depending on the value of the flags bit mask passed as first argument a new namespace is created or the old one is returned.

- return the created `nsproxy` structure 2.1.

```
1  static struct nsproxy *create_new_namespaces(
2                                  unsigned long flags,
3                                  struct task_struct *tsk,
4                                  struct user_namespace *user_ns,
5                                  struct fs_struct *new_fs)
6  {
7    struct nsproxy *new_nsp;
8    // Create new nsproxy.
9    new_nsp = create_nsproxy();
10   // Mount namespace.
11   new_nsp->mnt_ns = copy_mnt_ns(flags, tsk->nsproxy->mnt_ns,
12                                 user_ns, new_fs);
13   // UTS namespace.
14   new_nsp->uts_ns = copy_utsname(flags, user_ns,
                                    tsk->nsproxy->uts_ns);
15   // IPC namespace.
16   new_nsp->ipc_ns = copy_ipcs(flags, user_ns,
                                 tsk->nsproxy->ipc_ns);
17   // PID namespace.
18   new_nsp->pid_ns_for_children = copy_pid_ns(flags, user_ns,
19                               tsk->nsproxy->pid_ns_for_children);
20   // Cgroup namespace.
21   new_nsp->cgroup_ns = copy_cgroup_ns(flags, user_ns,
22                                       tsk->nsproxy->cgroup_ns);
23   // Network namespace.
24   new_nsp->net_ns = copy_net_ns(flags, user_ns,
25                                 tsk->nsproxy->net_ns);
26   // Time namespace.
27   new_nsp->time_ns_for_children = copy_time_ns(flags,  user_ns,
                                 tsk->nsproxy->time_ns_for_children);
28   new_nsp->time_ns = get_time_ns(tsk->nsproxy->time_ns);
29
30   return new_nsp;
31 }
```

Listing 2.10. Linux kernel `create_new_namespaces` in `/kernel/nsproxy.c`

2.1.2 Namespace API

Namespaces come with an API which consists of three system calls:

1. `clone()`
2. `unshare()`
3. `setns()`

`clone`

One way to create a namespace is to use the `clone` syscall. There are several implementation of the `clone` function like `clone(2)` and `clone(3)`. The difference is that the `clone(3)` receives as argument a `clone_args` structure, instead `clone(2)` receives a set of parameters that are passed individually. The `glibc` wrapper prototype of `clone(2)` is reported below:

```
1 int clone(int (*child_func)(void *),
2           void *child_stack,
3           int flags, void *arg);
```

Listing 2.11. *glibc* wrapper for `clone()`

Before introducing how `clone` can be used to manage namespaces let's spend a few words about how it works. Basically `clone` allows to control which parts of the execution context (virtual memory, open file descriptors...) are shared between the calling process and the child process. Moreover, it allows the child process to be created in a different set of namespaces than its father. This topic will be discussed later on.
Differently from the `fork` syscall the child process created by `clone` starts its execution from the function pointed by the first argument provided to the system call: `child_func`. As desrcribed in the Linux man page of clone [30], when `child_func` returns the child terminates. The function executed by the child (`child_func`) returns to the parent an integer that represents the exit status of the cloned process. The last argument `arg` is passed as argument of the function `child_func`.
The `clone` system call is very interesting because it represents the unifying implementation shared between processes and threads, as written here [31]. Eli Bendersky explains how threads and processes have always been seen as two completely different entities. Actually this is not true. From the

Linux kernel point of view there is no difference between process (which are the results of `fork`) and threads (the result of `pthread_create`). They are both represented by the same concept of *task* which is implemented by the `task_struct` kernel data structure. In Linux, threads are *task* that share their memory space whereas processes are *task* that don't share resources. Despite threads and processes are managed by different libraries and APIs they are both created thanks to the `clone` system call.
Every feature that differentiates threads and processes is obtained by passing different flags 2.1 to `clone`. For this reason it's better not to see threads and processes as two completely different concepts but as variants of the same concept, starting a concurrent *task*. The major differences are only in what is shared between the parent *task* and the child one.

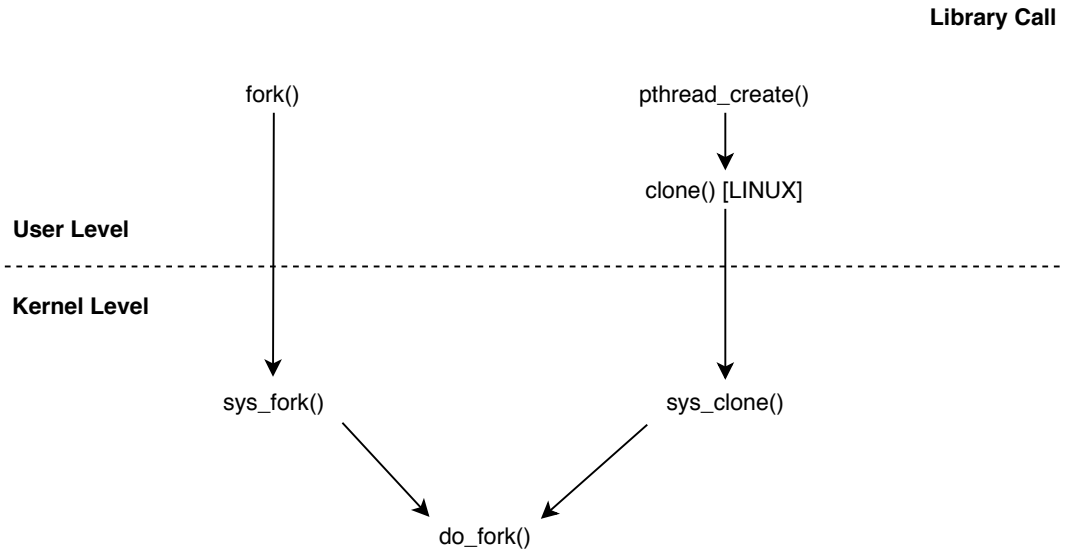

Figure 2.4. *process* and *thread* creation calls chain

`clone` and namespaces

The `clone` system call allows to create a new *task* in a separate set of namespaces. By passing as parameter a 32-bit mask flag, defined by ORing different namespace flags (reported in table 2.1), clone will be in charge of creating the specified namespaces and put the created task inside of them. As we can see by the definition of the `clone` system call 2.12 the only action performed is to call `_do_fork` 2.13 passing as parameter the `kernel_clone_args` data structure. This acts like a wrapper for the parameters of `clone`.

```
SYSCALL_DEFINE5(clone, unsigned long, clone_flags,
      unsigned long, newsp,
      int __user *, parent_tidptr,
      unsigned long, tls,
      int __user *, child_tidptr) {
  struct kernel_clone_args args = {
    .flags     = (lower_32_bits(clone_flags) & ~CSIGNAL),
    .pidfd     = parent_tidptr,
    .child_tid  = child_tidptr,
    .parent_tid = parent_tidptr,
    .exit_signal  = (lower_32_bits(clone_flags) & CSIGNAL),
    .stack     = newsp,
    .tls     = tls,
  };

  if (!legacy_clone_args_valid(&args))
    return -EINVAL;

  return _do_fork(&args);
}
```

Listing 2.12. `clone` syscall ABI definition

The `_do_fork` is the core function to create a new *task*. Note that we are talking about *tasks* and not *threads* or *processes* since, as explained before, the difference between *threads* and *processes* it's just a matter of flags passed to `clone`. Actually `_do_fork` performs different actions to initialize the new *task* structure but the only function we are interested in is the `copy_process`. This function is in charge of copying different data structure from the parent to the child, including namespaces. Depending on the flags of the `kernel_clone_args` structure it will create a new namespace for each flag ad assign it to the `ns_proxy` structure of the child.

```
long _do_fork(struct kernel_clone_args *args) {
    struct pid *pid;
    struct task_struct *p;
    /* create a new process as copy of the old one */
    p = copy_process(NULL, trace, NUMA_NO_NODE, args);
    [...]
    /* put the task in the runqueue and then run it */
    wake_up_new_task(p);
}
```

Listing 2.13. `_do_fork` implementation in `/kernel/fork.c`

unshare()

The unshare system call allows the current *task* (process or thread) to dissociate parts of its execution context. It creates a new set of namespaces, in accord with the flags passed as argument, and attach the current process to it. Its prototype is the following:

```
1 int unshare(int flags);
```

Listing 2.14. Linux unshare system call prototype

In the Linux kernel the unshare system call is defined as follow. Looking at the syscall definition 2.15, the number one at the end of SYSCALL_DEFINE1 indicates that the unshare syscall takes only one input parameter (the unshare_flags bit mask).

```
1 SYSCALL_DEFINE1(unshare, unsigned long, unshare_flags)
2 {
3   return ksys_unshare(unshare_flags);
4 }
```

Listing 2.15. unshare syscall in /kernel/fork.c

The ksys_unshare 2.16 function takes as argument the flags bit mask and it is in charge for associating the current task to the a new set of namespces. The following steps are performed:

1. calls unshare_userns() to create the new user_ns, filling the new_cred structure. This is not done in unshare_nsproxy_namespaces because as we said before the user_ns is not part of the struct ns_proxy.

2. calls unshare_nsproxy_namespaces() 2.9 in order to create a new nsproxy structure. The unshare_nsproxy_namespaces 2.9 function is responsible for creating a new nsproxy structure that will be used by the task to leave its old set of namespaces and associates with the new one in accord to the flags passed as first argument. In particular, a new set of namespaces will be created or copied from the parent nsproxy structure.

3. when the new nsproxy structure is created, a new call to switch_task_namespace(current, new_nsproxy) 2.8 is done in order

to switch the old and the new `ns_proxy` in the `task_struct` of the current *task*.

4. calls `commit_creds()` to install the new `cred` structure, containing the new user namespace, to the current task.

```
int ksys_unshare(unsigned long unshare_flags)
{
  struct fs_struct *fs, *new_fs = NULL;
  struct files_struct *fd, *new_fd = NULL;
  struct cred *new_cred = NULL;
  struct nsproxy *new_nsproxy = NULL;

  /* Create the new_cred struct if necessary. Internally checks
   * if the CLONE_NEWUSR flag is actually set
   */
  unshare_userns(unshare_flags, &new_cred);

  /* create the new ns proxy struct for the new namespaces */
  unshare_nsproxy_namespaces(unshare_flags, &new_nsproxy,
                             new_cred, new_fs);
  /* set to the current task the new nsproxy struct */
  if (new_nsproxy)
    switch_task_namespaces(current, new_nsproxy);
  [...]
  if (new_cred) {
    /* Install the new user namespace */
    commit_creds(new_cred);
    new_cred = NULL;
  }
}
```

Listing 2.16. `ksys_unshare` in `/kernel/fork.c`

`setns()`

The `setns()` system call allows a process or thread to move into a different set of namespaces. Its prototype is reported below:

```
int setns(int fd, int nstype);
```

Listing 2.17. Setns syscall prototype

Both the arguments can assume different meanings:

- `fd` refers to a `/proc/[pid]/ns/` link (`nsfd`) then nstype indicates the type of the namespace we are associating to. This allows only to associate with one namespace at a time. It is also possibile to pass 0 as nstype. In this case any type of namespace is allowd to be joined.
- fd refers to a PID file descriptor (pidfd) then setns moves the calling thread into one or more of the same namespaces as the target process referred to by fd. In this case nstype is a flag bit mask which indicates the namespaces we want to associate with. Specifying 0 as nstype is not valid together with a pidfd.

Taking a look at the implementation we can see that, differently from the unshare syscall definition 2.15, `SYSCAL_DEFINE2` ends with 2 that indicates the two parameters needed as argument.
A new structure called `nsset` 2.18 is used to include all bits needed to install a partial or complete new set of namespaces as reported in [27] at line 45. It's definition it's reported below:

```
1 struct nsset {
2   unsigned flags;
3   struct nsproxy *nsproxy;
4   const struct cred *cred;
5   struct fs_struct *fs;
6 };
```

Listing 2.18. `nsset` Linux kernel structure in `linux/include/linux/nsproxy.h`

The main tasks performed by `setns` are:

- calls `proc_ns_file(file)` to check if fd refers to a symbolic link inside the procfs. If true, it calls `get_proc_ns()` to obtained the corresponding `ns_common` structure otherwise fd is referred to a pid file descriptor and the set of flags is checked.
- calls `prepare_nsset(flags , &nsset)` to create the `nsset->nsproxy` structure. Internally it will call create_new_namespaces() passing 0 as first parameter (flags bit mask). This implies that only an nsproxy is created by calling the create_nsproxy() method, but no new namespace is created (setns() does not create new namespace but just attach current task to an exhisting namespace).

- calls again `proc_ns_file(file)` to distinguish between the two execution cases. If fd refers to a symbolic link in /proc/[pid]/ns/ we can attach to only one namespace and validate_ns is called. Internally, thanks to the proc_ns_operations struct of ns_common it will call the install() function of the namespace we want to attach to. During installation the reference count of the namespace we are detaching from is decremented and the count of the target namesapce is incremented. Then the new namespace is saved into the nsset->nsproxy. In the other case validate_nsset() is called which is responsible for accessing the nsproxy of the target task and associating the current task with the specified namespaces.

- At the end the commit_nsset function will internally call `switch_task_namespaces` 2.8 in order to save the new nsproxy into the structure of the current task.

```
1  SYSCALL_DEFINE2(setns, int, fd, int, flags)
2  {
3    struct file *file;
4    struct ns_common *ns = NULL;
5    struct nsset nsset = {};
6
7    file = fget(fd);
8
9    [...]
10
11   if (proc_ns_file(file)) {
12     /* fd refers to a symbolic link in /proc/[pid]/ns/, we take
        * the ns_common struct.
13      */
14     ns = get_proc_ns(file_inode(file));
15     /* check flag type passed as parameter against the type
        * taken from ns_common.
16      */
17     if (flags && (ns->ops->type != flags))
18       err = -EINVAL;
19   flags = ns->ops->type;
20   } else if (!IS_ERR(pidfd_pid(file))) {
21         /* fd refers to a pid file descriptor. */
22         err = check_setns_flags(flags);
23   } else {
24         err = -EINVAL;
25   }
```

```
26
27    [...]
28
29    /* Create nsset->nsproxy and nsset->cred. */
30    prepare_nsset(flags, &nsset);
31
32    if (proc_ns_file(file))
33      /* fd refers to a symbolic link in /proc/[pid]/ns/ */
34      validate_ns(&nsset, ns);
35    else
36      /* fd refers to a process file descriptor, ns_common not
37       *  needed.
38       */
39      validate_nsset(&nsset, file->private_data);
40
41    [...]
42
43    /* save new namespace set into current task. */
44    commit_nsset(&nsset);
45
46    [...]
}
```

Listing 2.19. Linux `setns` system call in `/kernel/nsproxy.c`

Procfs

Each process has a dedicated /proc/PID/ns directory inside the proc filesystem where we it is possible to find information about the namespaces it belongs to.

```
1 $ ls -l /proc/$$/ns          # $$ is replaced by shell's PID
2   total 0
3   lrwxrwxrwx. 1 mtk mtk 0 Jan  8 04:12 ipc -> ipc:[4026531839]
4   lrwxrwxrwx. 1 mtk mtk 0 Jan  8 04:12 mnt -> mnt:[4026531840]
5   lrwxrwxrwx. 1 mtk mtk 0 Jan  8 04:12 net -> net:[4026531956]
6   lrwxrwxrwx. 1 mtk mtk 0 Jan  8 04:12 pid -> pid:[4026531836]
7   lrwxrwxrwx. 1 mtk mtk 0 Jan  8 04:12 user -> user:[4026531837]
8   lrwxrwxrwx. 1 mtk mtk 0 Jan  8 04:12 uts -> uts:[4026531838]
```

Listing 2.20. Proc virtual filesystem

Each number enclosed by square brackets is the inode number for the corresponding symbolic link. If two different processes are in the same namespace then the inode number will be the same for the corresponding symbolic link.

Actual these entries are not only usefull to check if two processes are in the same namespace, but they can also be used to create a persistent namespace. When a new namespace is created it will persist as long as it has a process member of it. When all the processes leave the namespace it can be freed. Alternatively, a new namespace can be made persistent by doing a bind mount of /proc/pid/ns/[entry] files to another path in a private filesystem. A persistent namespace lives even if there are no processes member of it (except for the PID namespace which requires a permanent init process to run). We can enter in a persistent namespace by using nsenter specifying the path to the bind mount. A persistent namespace can be deleted by using umount to remove the bind mount.

2.1.3 PID Namespace

As described by *Mahmud Ridwan* [28], in the past years the Linux kernel has maintained a single process tree. Processes in the same tree can inspect another process, attach a tracer to it or may even be able to kill it. Today, the process namespace allows to have "nested" process trees that are isolated from each other. Two process that are in two different and not related process trees can not be know the existence of the other, so they can not inspect or kill it. In Linux, the root process has process identifier equal to `1` and it is designed as root of the process tree.
All other processes starts below it in the tree. Thanks to *process namespace* is possible to create a new tree starting from a new process with PID 1. This process will be the root of the new nested tree but it will be attached to the parent tree.
As presented in 2.5 PID namespace are a hierarchical architecture, so the processes of the parent PID namespace are aware of the existence of the son namespace processes but not vice versa. So a process that stands in a child PID namespace can not see another process that sits in a parent PID namespace.

As you can see, with the introduction of the *process namespace*, a single process can have multiple PIDs associated with it. Each of them is associated with one namespace. A pid namesapce is represented in the Linux kernel by the `pid_namespace` 2.21 structure.

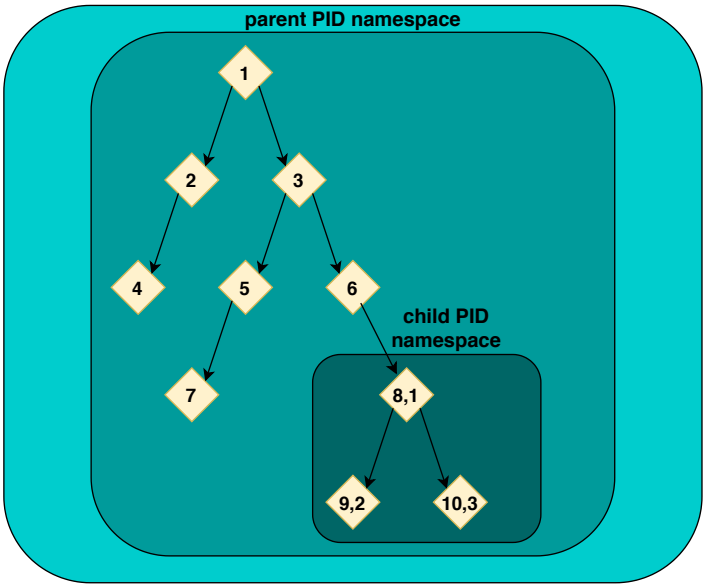

Figure 2.5. Process Namespace tree example inspired from [28]

```
1 struct pid_namespace {
2   struct task_struct *child_reaper;
3   unsigned int level;
4   struct pid_namespace *parent;
5   [...]
6 }
```

Listing 2.21. `pid_namespace` structure

The main fields of this structure, which are relevant for us, are:

- `parent`
  The parent of a PID namespace is the PID namespace of the process that created the namespace using `clone` or `unshare()`.

- `child_reaper`
  Pointer to the task_struct structure of the task which is in charge for becoming the parent of orphaned process. Usually this is done by the init process of each namespace.

- `level`
  Depth in the namespace hierarchy.

The `pid` 2.22 structure is the Kernel level representation of the PID concept.

```
struct pid {
  unsigned int level;
  struct upid numbers[1];
  [...]
}
```

Listing 2.22. Linux kernel `pid` structure

The field level represents the depth of the namespace in which the task holding a reference of the pid structure was created. Numbers is an array of upid structure. Due to the introduction of the pid namespace, a task can have multiple PIDs that are are relevant in their specific PID namespace. For this reason the structure `upid` 2.23 was added to maintain together the couple (PID number, namespace).

```
struct upid {
  int nr;                       // the PID number
  struct pid_namespace *ns;     // namespace where PID is relevant
}
```

Listing 2.23. Linux kernel `upid` structure

The `task_struct` structure has a reference called `thread_pid` which holds information about the PID structure of the current *task*. It also have two other entries which refers to the global PID and TGID respectively. As written by Pavel Emelyanov and Kir Kolyshkin [32], Global IDs are unique in the entire system, just like the old PIDs were. These are only useful when the PID value is not going to leave the kernel.

```
struct task_struct {
  pid_t        pid;             //global pid
  pid_t        tgid;            //global tgid
  struct pid   *thread_pid;     //reference to pid structure
  [...]

}
```

Listing 2.24. `task_struct`'s entries relevant for `pid` namespace

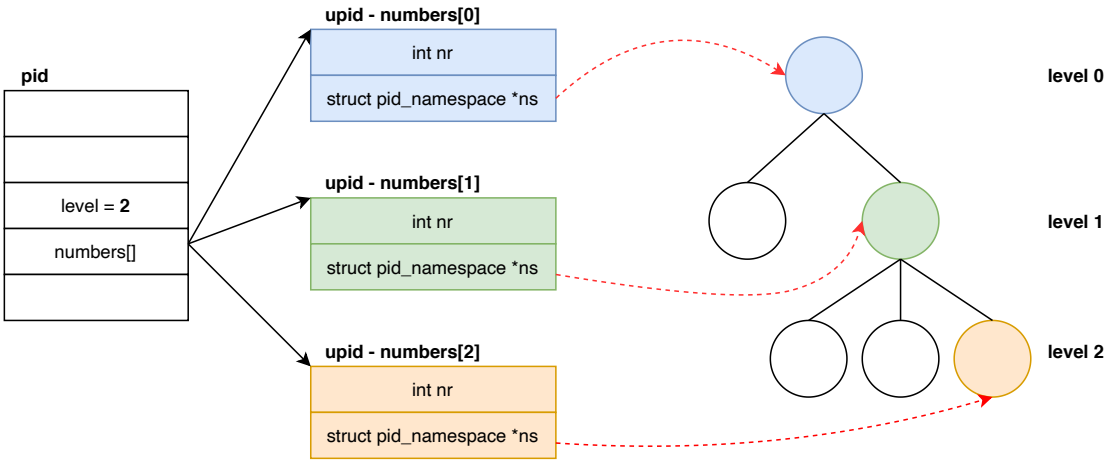

Figure 2.6. graphic representation of numbers

A more detailed explanation about how the PID for a new task is allocated is reported in *How process IDs are allocated* in the appendix A.

2.1.4 Network Namespace

A vital feature for namespaces is networking. Network namespaces allows to isolate the complete network stack, in particular allows two processes to perceive a different network setup. An example of what can be done using network namespaces is:

- see different interfaces in each namespace
- having different routing table
- set different firewalling rules

Logically each network namespace can be considered as a copy of the Linux network stack. In particular, a physical network device can live in a certain network namespace and when it is destroyed, it is moved back to the parent network namespace.
Is also possible to use a *virtual network device pair* 2.1.4 to provide a network tunnel between the namespace borders and another namespace.
When a new *Network namespace* is spawned, the process inherits its network namespace from its parent.

Virtual Ethernet Devices

Virtual Ethernet Devices acts as a tunnel between a network device and a namespace or between two network namespaces. Anything that enters in one ends, comes out through the other end as in a real Ethernet connection between two real nodes. Is also possible to use a `veth` as a standalone network device.
Virtual Ethernet Devices are always creates as interconnected pairs and are an abstraction of a wire that allows the traffic to flow between the two ends.

```
1 $ #!/bin/bash
2 $ ip link add <p1-name> type veth peer name <p2-name>
```

Listing 2.25. How to create a Virtual Ethernet Device pair

```
1 $ #!/bin/bash
2 $ ip link set <p2-name> netns <netnsname>
```

Listing 2.26. How to move one pair to a network namespace

These two commands can be also chained together.

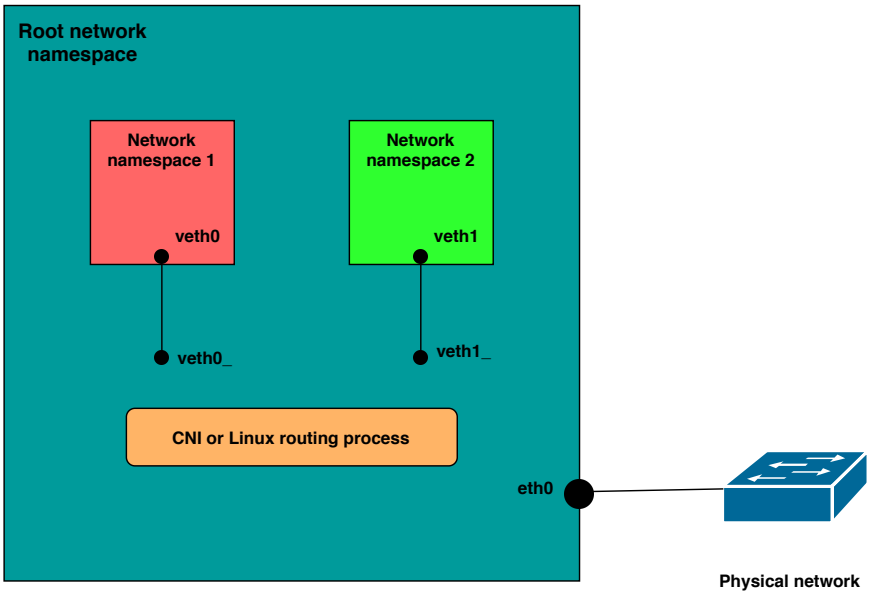

Figure 2.7. Simple Network namespace interconnection scheme

There are different network solutions that can be used to interconnect VM's and network namespaces, an example is *Project Calico*[36] or *Polycube Network* [37].

How to interconnect two Network namespaces

In this example is reported how to interconnect two different *Network namespaces*. This is only a demonstrative example, more complex situations based on Ethernet bridges, and even route packets between namespaces can be created.

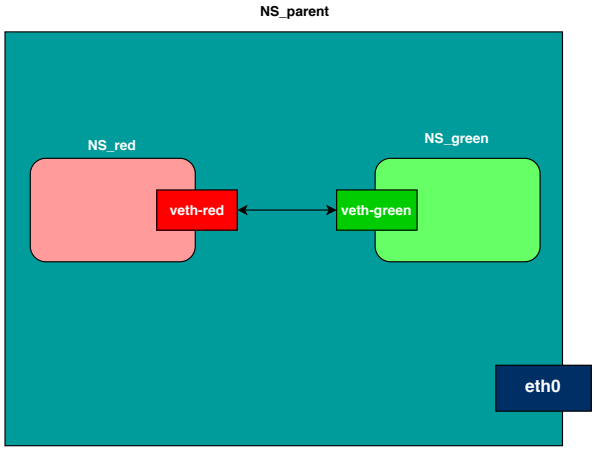

Figure 2.8. Simple Network namespace interconnection scheme

```
$ #! /bin/bash
// creating namespaces
$ ip netns add NS_red
$ ip netns add NS_green

// creating a new veth pair
$ ip link add veth-red type veth peer name veth-green

// move veth-red veth to NS_red and veth-green to NS_green
$ ip link set veth-red netns NS_red
$ ip link set veth-green netns NS_green

// assign a new ip address to NS_red and NS_green
$ ip netns exec NS_red addr add 192.168.8.1 dev veth-red
$ ip netns exec NS_green addr add 192.168.8.2 dev veth-green
```

```
16
17 // set veth-red and veth-green up
18 $ ip netns exec NS_red link set veth-red up
19 $ ip netns exec NS_green link set veth-green up
```

Listing 2.27. commands to interconnect two *Network namespaces*

2.1.5 UNIX Time Sharing namesapce

The UNIX Time Sharing namesapce (`CLONE_NEW_UTS`) provides isolation of two system identifiers [33]:

- Hostname
- NIS domain name

If these identifiers are changed they will be valid only in their own UTS namespace. The hostname can be set using `sethostname`. The NIS domain name is set by `setdomainname`.

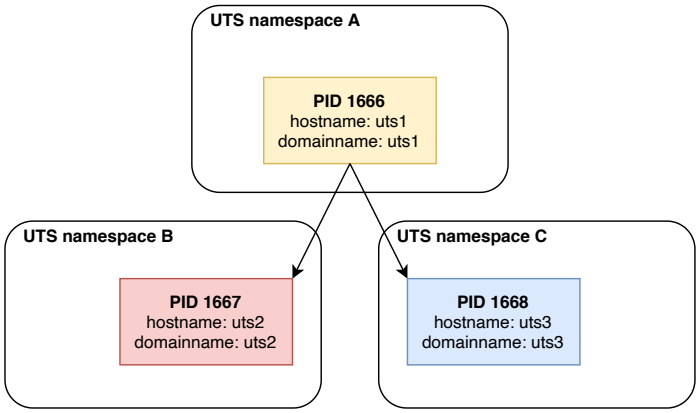

Figure 2.9. UTS namespace

The UTS namespace is defined in the Linux kernel by the following structure:

```
1 struct uts_namespace {
2   struct kref kref;
3   struct new_utsname name;
4   struct user_namespace *user_ns;
```

```
5   struct ucounts *ucounts;
6   struct ns_common ns;
7 }
```

Listing 2.28. `uts_namespace` structure

Most of the fields are common to other namespaces and has been already explained in the "Namespaces internals" section.

The `new_utsname` 2.29 field, as the name suggest, is specific for the UTS namespace and holds together different information, including the *nodename* and the *domain name*, which are the only ones that can be modified.

```
1 struct new_utsname {
2   /* Operating system name (e.g., "Linux") */
3   char sysname[__NEW_UTS_LEN + 1];
4   /* Network node hostname */
5   char nodename[__NEW_UTS_LEN + 1];
6   /* Operating system release (e.g., "2.6.28") */
7   char release[__NEW_UTS_LEN + 1];
8   /* Operating system version */
9   char version[__NEW_UTS_LEN + 1];
10  /* Hardware identifier */
11  char machine[__NEW_UTS_LEN + 1];
12  /* NIS or YP domain name */
13  char domainname[__NEW_UTS_LEN + 1];
14 };
```

Listing 2.29. `new_utsname` structure

2.1.6 User namespace

The user namespace (`CLONE_NEW_USR`) is responsible for isolating security identifiers and attributes including user IDs, group IDs and capabilities. In particular, the user namespace allows to map user and group IDs over different namespaces. Thanks to this feature is possible to have a process running as root inside a user namespace, but having no privilege outside of it. In this way the process is allowed to do operations, which require privilege, only on resources owned by that user namespace. As described in the Linux man page of the user namespace [51], each resource that is controlled by a namespace (that is not a user one) can be used by the task that has an user namespace

owner of these namespaces. Moreover, there are still many privileged operations that affect resources that are not managed by any namespace type, for example, loading a kernel module or creating a device. This kind of operations requires a process with privileges in the root user namespace. Each process is a member of exactly one user namespace.
The user namespace is defined by the `user_namespace` 2.30 structure in the Kernel.

```
struct user_namespace {
  struct uid_gid_map  uid_map;
  struct uid_gid_map  gid_map;
  struct user_namespace *parent;
  int      level;
  kuid_t   owner;
  kgid_t   group;
  [...]
}
```

Listing 2.30. Linux kernel `user_namespace` structure

The field `owner` and `group` are the effective user ID and group ID of the creating process of the user namespace. The field `level` is the depth of the user namespace in the user namespaces' hiearchy.
Fields `uid_map` and `gid_map` defines the mapping of uid/gid between the child task inside the user namespace and the parent (which sits outside).
The mapping is represented by the `uid_gid_extent` 2.31 structure. A reference to this structure is kept inside the `uid_gid_map` structure of the `uid_map` and `gid_map` field.

```
struct uid_gid_extent {
  u32 first;
  u32 lower_first;
  u32 count;
};
```

Listing 2.31. `uid_gid_extent` structure

The first operation that is done when a new process is created in a new user namespace is defining the mapping between the UID and the GID of the process by writing information to the `/proc/[PID]/uid_map` and `/proc/[PID]/gid_map` files associated to the interested process in the user

namespace. These files consists of one or more lines, each of which contains three values separated by white space:

- `ID-inside-ns`
  Starting ID for the range inside the current user namespace.
- `ID-outside-ns`
  Defines the starting point of the ID range outside the considered user namespace. How this field must be interpreted depends if the reading process (*PIDr*) and the observed process PID are or not in the same namespace.
  In particular:
  - If *PIDr* and PID belong to different user namespaces, then `id-outside-ns` refers to the start ID in *PIDr*'s user namespace.
  - if *PIDr* and PID are in the same user namespace, then `id-outside-ns` refers to the start ID in PID's parent user namespace.
- `length`
  Length of the mapped range.

These three values are mapped to the `first`, `lower_first` and `count` fields of the uid_gid_extent structure respectively.
As shown in figure 2.10, the child user namespace owns only the second uts namespace. If process A attempts to change the hostname, it will succeed because being a member of the child user namespace (that owns the second uts namespace) it has the privileges to do such operation. Conversely if process A tries to bind to a reserved socket port, since it is a member of the initial network namespace (which is owned by the initial user namespace) it will fail because the user namespace it is a member of does not own the initial network namespace.

2.1.7 Mount namespace

Mount namespace (`CLONE_NEWNS`) was the first namespace type added to Linux (2002 - Linux 2.4.19) [38]. It is used to isolate the list of mount points seen by the processes in each namespace instance. Consequently, each process that resides in a different mount namespace will see its own list of mount points. In other words, a mount namespace represents the set of mounted file system that are sees by the considered process.
Remember that a mount point represents the place of a particular dataset (a

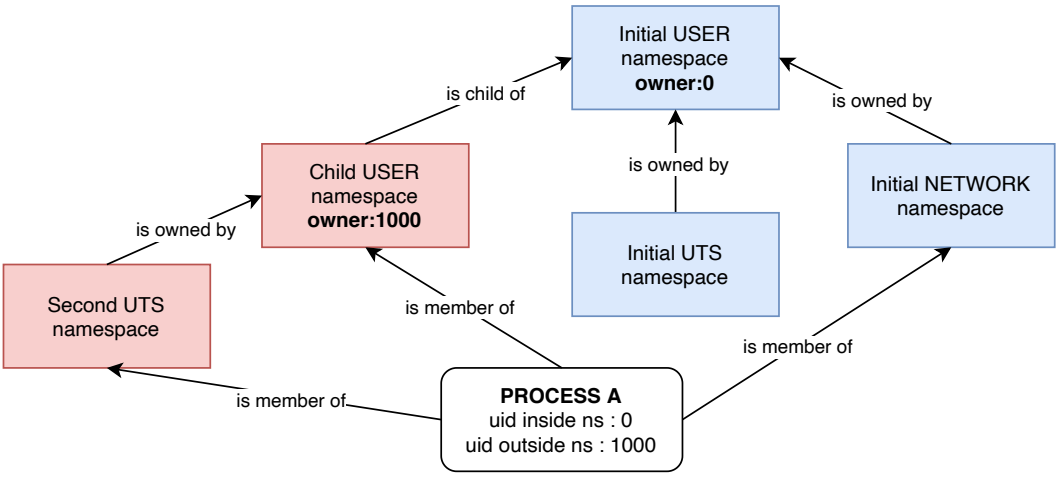

Figure 2.10. User namespace relationship [52].

device or file system) in a particular directory hierarchy. When the mount namespace is created, the list of the mount points is initialized differently according to the function used to create it (`clone` or `unshare()`):

- Using `clone`, the mount point of the child's namespace is a copy of the mount point present in the parent's namespace. In this case, the child namespace will get its own copy of the data about mounted filesystem from the parent. The new mount namespace process can change his set of mounted file systems without affecting the parent.
- Using `unshare()`, the mount point of the child's namespace is a copy of the caller's previous mount namespace. In this way the process associate with `unshare()` is moved to the new namespace.

After the `clone` or `unshare()` call, starting for a copy of the mount namespace of the parent, mounts points can be added or removed in each namespace using `mount()` and `umount()` but these changes are visible only in the process associate with the considered namespace and not in other mount namespaces.

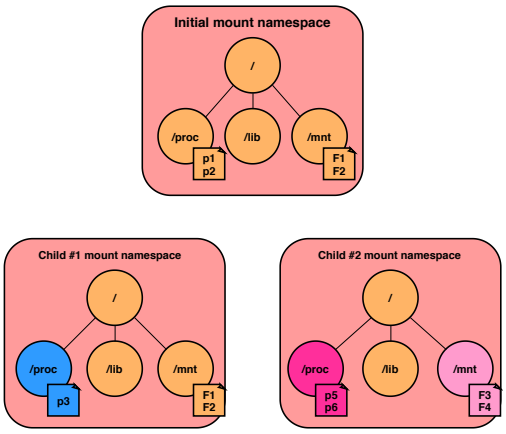

Figure 2.11. mount namespaces representation

Often, mount namespaces are combined with `chroot()` style isolation in order to isolate the filesystem that is sees by the new process to a portion of a single directory hierarchy.
It is also possible to mount the /texttt/proc pseudo filesystem for the considered process to limit the vision over processes that are only child of it. In this way, the child process will see only a set of processes that are in the same pid namespace where it resides (or event that of the pid namespaces children of this).

```
1    /* Create directory for mount point */
2    mkdir(mount_point, 0555);
3    mount("proc", mount_point, "proc", 0, NULL);
4    printf("Mounting procfs at %s\n", mount_point);
```

Listing 2.32. example of proc mount point

Shared subtrees

Shared subtrees are a Linux Kernel feature that allows to automatically mount or unmount events between namespaces. This means that, for example, mounting an optical disk in a mount namespace can trigger a mount of that disk in other mount namespace in automatic way. It is useful if for example is necessary to add a new optical disk in the system.

2.1.8 IPC namespace

The IPC namespace (`CLONE_NEWIPC`) [39] is used to isolating System V IPC objects and POSIX message queues. This namespace is defined in the Linux Kernel by the `ipx_namespace` structure. In particular, each IPC namespace has its own copy of System V IPC identifiers and its own copy of POSIX message queue filesystem. Each IPC object that is create in a specific IPC namespace will be only visible by all processes that resides in the same IPC namespace. If an IPC namespace is destroyed, all IP objects inside it are automatically removed.
The interfaces that are distinct in each IPC namespace are:

- The System V interface under `/proc/sys/kernel`
- The System V interface under `/proc/sysvipc`
- The POSIX message queue interfaces in `/proc/sys/fs/mqueue`

2.1.9 Time namespace

The time namespace [40] was originally proposed in 2018 but it enters in the mainline of the Linux kernel in the version 5.6 (early 2020).
This namespace allows the virtualization of two different system clocks that can not be usually modified:

- `CLOCK_MONOTONIC`: represents the absolute monotonic elapsed time since some arbitrary fixed time in the past.
- `CLOCK_BOOTTIME`: is similar to `CLOCK_MONOTONIC` but it also includes the elapsed time since the boot of the system.

It is not possible to create a new time namespace using `clone`, do to it is necessary to use `unshare()` using `CLONE_NEWTIME` as flag. It will change the value of `/proc/[pid]/ns/time_for_children` but the current task time namespace is not replaced. However, this will have an effect on its child.

2.2 Control groups

Control groups (also called "*Cgroups*") are a Linux Kernel feature developed for the first time in 2006 by *Google* engineers *Paul Menage* and *Rohit Seth* and mainlined in 2007 (*cgroups v1*)[41]. A second version (*cgroups v2*) was

developed by *Tejun Heo* in 2016.[42]
Cgroups provides a mechanism to isolate ad aggregate sets of tasks (and their children) into a group characterized by special behaviors.[43] When the flag `CLONE_NEWCGROUP` is passed to `clone` or `unshare()`, the process is pushed inside a new croup namespace and its current cgroup directories become the cgroup root directories of the new namespace.

Cgrops basic concepts

Like processes, Cgroups are organized hierarchically and each child cgroup inherit some characteristics from his parent. But the two models (*cgroups* and *processes*) are quite different:

- **process model:** the `init` process is the common parent of all processes on a Linux System. Due to this fact, the process model in Linux is a single hierarchy tree.
- **cgroup model:** Depending on the version (*v1* or *v2*) multiple cgroups hierarchies can be present in the same system. Each hierarchy can be attached to one o more subsystem. The main difference between *cgroups v1* and *cgroups v2* is that in the second version there is only one single hierarchy and there is a difference between processes and threads.
  Moreover, if a process is moved to a new cgroup, the restrictions are applied to all its threads.

The basics blocks of the cgroup infrastructure are:

- **Resource controller - subsystem**: represent a single resource of the machine, such as `memory`, `devices`, `cpu`, `pids` number etc...
- **Hierarchy**: The set of cgroups that are arranged in a tree. Each hierarchy can be attached to one or more resource controller and it is associated to a specific instance of the cgroup virtual filesystem.
- **Cgroup**: The node of the hierarchy. Each node is composed by one task.
- **Task**: The task of the process that is located inside a specific cgroup.

Cgroup organization

At each cgroup namespace is associated a basepoint directory (`/container` in this example) that is pointed by each task that resides inside it. For each

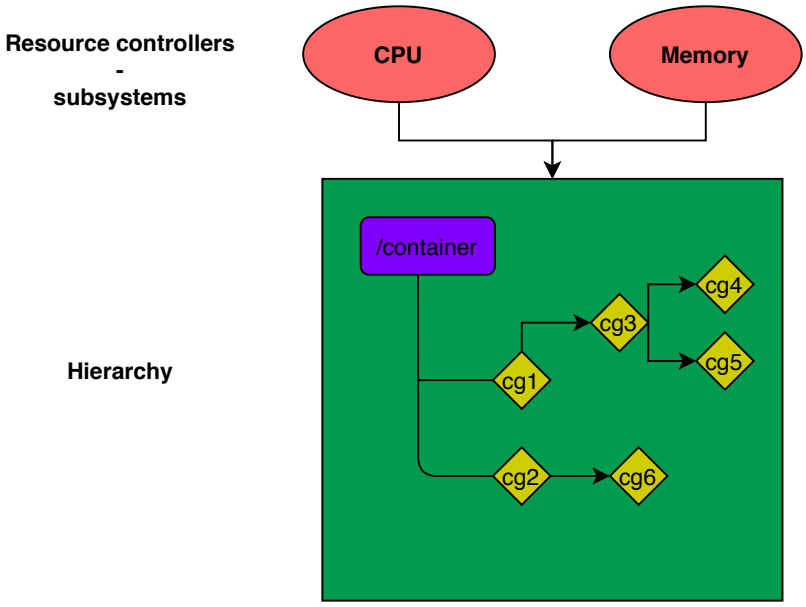

Figure 2.12. Control group organization

process identifier, a file under `/proc/[pid]/cgroup` reports the lists of base-pointers for the associated cgroup namespace of the considered subsystem.

```
12:devices:/user.slice
11:pids:/container
10:freezer:/
9:cpuset:/
8:blkio:/
7:cpu,cpuacct:/
6:memory:/
5:rdma:/
4:perf_event:/
3:net_cls,net_prio:/
2:hugetlb:/
1:name=systemd:/user.slice/user-1000.slice/session-2.scope
0::/user.slice/user-1000.slice/session-2.scope
```

In this example, `/container` is the baseline for the `pids` **subsystem** of our **task**. The `/sys/fs/cgroup/pids/` directory of our system will contains the information about the considered cgroup. Thanks to the cgroup namespace, the baseline directory of the new process created by `clone` or `unshare()` will be his root directory "/".
Another interesting file is `/proc/cgroups` that reports information about cgroup namespaces and hierarchies in the observed system:

```
#subsys_name    hierarchy    num_cgroups  enabled
cpuset              9              1          1
cpu                 7              1          1
cpuacct             7              1          1
blkio               8              1          1
memory              6              1          1
devices             12             75         1
freezer             10             2          1
net_cls             3              1          1
perf_event          4              1          1
net_prio            3              1          1
hugetlb             2              1          1
pids                11             81         1
rdma                5              1          1
```

Relationship between hierarchies are allowed but there are few limitations acts to reduce the complexity of the problem:

- A single hierarchy can have one o more subsystems attached to it
- A subsystem, that is attached to a hierarchy, cannot be attached to another hierarchy that is already attached to another subsystem.
- A single task cannot be even in two different cgroups that resides in the same hierarchy. When it is moved in another cgroup, automatically the system will remove it from the old hierarchy.
- A forked task inherits the exact same cgroups as its parent task but can be moved to different cgroups as needed.

Cgroup version 1 and version 2

When cgroup v1 was released, developers starts to take an interest in this new feature and in the following years the fast and uncoordinated development of controllers lead inconsistencies in the first implementation of control groups. In particular, in this version of cgroup, each controller can be mounted by different cgroups where there is not difference between processes and threads. The membership in each cgroup was per task. This flexibility is also a major problem of resource management. Cgroup v2 was designed using an *unified hierarchy* where a process can be member of one cgroup. When a process is moved to another cgroup, all its threads are moved to the same target.

Chapter 3

Nscage: lightweight process isolation tool

Nscage [46] is a lightweight process isolation tool, written in C, based on Linux Containers and related technologies (namespaces, cgroups, Linux capabilities and seccomp-bpf syscalls filtering). This tool is intended to provide a secure and isolated execution environment for running a process inside of it, reducing the possible effects of security breaches on the host system. It can also be used to create a testing environment for applications under development to reduce the effects of possible errors on the system.

3.1 Motivations

Nowadays lightweight virtualization is a weapon used by many. Microservice based architecture is increasingly used and containers are the backbone of it. There are several software like Docker, or more in general all those tools regarded as containers manager, that allow us to easily create and manage containers. The problem is that the use of this applications actually hide the isolation mechanisms that are used to create a container, loosing the concept of what really is a Linux Container. The development of this tool gave us a clear idea on how containers are spawned inside a container manager and which are the steps that are needed to be performed in order to enforce isolation at process level. It allowed us to directly work with the namespace API that the kernel of Linux offers and to face the problems related to the development of software at low level. Generally speaking, the operations that are performed in our tool to create a container are the core steps done by the different container engines that reside below container managers.

3.2 Related works

These are some works, using container technologies such namespaces and cgroups, that inspired us in the creation of this tool.

- **Nsroot**
  Nsroot [61] has been developed in *C* by two students of the Arctic University of Norway, Tromsø. They used namesapces and pivot-root to create an isolate environment to run third party tools required by the META-pipe metagenomics data analysis service. A specific biological data analysis usually requires different tools that are arranged in a pipeline. Due to the fact that the pipeline consists of third party tools that may contains untrusted code not properly tested, it is important to isolate the individual pipeline tool executions from each other.
- **Nsjail**
  Nsjail [62] is a light-weight process isolation tool, written in *C++*, that makes use of Linux namespaces, cgroup and syscalls filtering to ensure isolation between different processes. It has been developed by Google to provide a lightweight solution to containarization of processes.
- **Charliecloud**
  Charliecloud [63] is a tool, written in Python and *C* and developed at Los Alamos, that proposes a lightweight approach based on Linux containers for high performance computing (HPC) centers. It provides user defined software stacks (UDSS) for HPC systems allowing users to bring their own software stacks to the system. This includes the possibility of introducing flexible new features that could increase center's attack surface. For this reason, to ensure

performance and security, they choose a lightweight approach based on Linux Container instead of complete virtual machines to run user code.

- **Firejail**
  Firejail [64] is a program, written in *C*, that uses Linux namespaces, cgroups, seccomp-bpf and Linux capabilities to create a sandbox to run untrusted application safely. It allow to create an isolated environment for any kind of processes, including graphical applications.

3.3 Usage

To use *nscage* it is first of all needed a root filesystem. This can be manually created or taken from a Docker container by simply extracting a tar archive from it using the below command.

```
$ #!/bin/bash
$ docker container export -o rootfs.tar <container_id>
```

This will be the new root of the isolated process. The CLI of nsroot is simple and includes few options that are listed beloew:

```
$ #!/bin/bash
$ ./nscage -h

Usage: sudo ./nscage <options> <entrypoint>

<options> should be:
    - a run the new process in a complete new set of
        namespace, this does not includes user namespace.
    - U create an user namespace for the new process
    - c <resources> cgrops used to limit resources.

    Where <resources> can be one or more of the following:
    - M <memory_limit>  [1-4294967296]  default: 1073741824 (1GB)
    - C <%_cpu_shares>  [1-100]         default: 25
    - P <max_pids>      [10-32768]      default: 64
    - I <io_weight>     [10-1000]       default: 10
```

Listing 3.1. `nscage` usage

When *nscage* is executed, a new set of namespace is created thanks to `clone` 2.1.2 and then the new process is jailed inside the new root filesystem using `pivot_root`. What pivot_root does is setting the new root mount for the process to the specified root_fs while moving the old root into a directory that can be later be unmounted

and deleted. This solution is to be preferred over `chroot` because there are several chroot-jail escapes [65] that cannot be exploited with pivot_root. To summarize, these are the main steps executed by the nscage main process to setup the new child environment:

- the parent creates the new cgroup namespace configuration if required (`-c` option)
- Clone is called to create a new child process in a new set of namespaces.
- The parent setup the network namespace for the child. This includes the set up of a veth pair between the parent's and the child's namespace.
- If a new user namespace is created, the parent defines the UID and GID mapping for the child.

The child process also needs to configure its resources and privileges, following these steps:

- If a new user namespace is created, the child waits the UID - GID mapping from the parent and then set its own UID and GID.
- Sets its new hostname
- Prepares the root filesystem for the execution of pivot_root. This includes mounting all the necessary filesystems and devices.
- Drops a set of capabilities.
- Applies a system call filtering security layer using seccomp.
- Calls execvp to execute the entrypoint command in an isolated environment.

All the steps briefly described here are analyzed deeply in the next sections, focusing on each namespace.

3.3.1 Namespaces

In this section we will analyze the main steps needed to create the new process and configuration of namespaces considered most interesting for the case study. Different setups are needed to allow proper operation in each namespace.

User namespace

The use of a new user namespace is optional (`-U` option). This allows the creation of a privileged or unprivileged container. If the -U option is set, clone will guarantee that the user namespace will be created first and delegated the owner of the other non user namespaces. In this way the containerized process gains a full set of capabilities over the resources that it actually owns (the ones that are created by clone alongside the user namespace). For example it is able to set a new *hostname* because it is a resource in its UTS_NAMESPACE whose it is the owner.

If a new user namespace is created, the parent need to define a mapping between the root user inside the container and the UID and GID seen from outside the user namespace (parent side). In this way we have the conteinerized process having root privileges inside the container but being unprivileged for operations outside its user namespace. These are the so called rootless containers, and they are more secure than being root on the host. We decided to make the user namespace optional because it also introduces some limitations that can compromise the correct operation of different applications (e.g. the possibility to create a new device with mknod). Without the user namespace option (`-U`), the child process is created in the same user namespace of its parent. In this case the privileged container has UID 0 mapped to the host's UID 0. This kind of containers are not root-safe and they must be intended to run only thrusted workloads. They are also called rootfull containers. The privileges of a rootfull container can be anyway controlled by reducing its capabilities set and syscalls.

UTS namespace

After the user namespaced has been configured, the child process is free to set its hostname. This will only affect the hostname as seen by the container process inside its new uts namespace.

Mount namespace

The mount namespace is mandatory for the new process that is created by `clone`. Before executing pivot_root to move the root mount point of the container and umounting the old root it is needed to prepare the target root filesystem. This includes the following operations:

1. Remount everything as private. In this way all the following mounts won't be propagated outside the current mount namepace. This is also needed to avoid pivot_root to fail.
2. Ensure that the rootfs of the container is a mount point. This can be done by bind mounting the rootfs on itself. This is another requirement for the execution of pivot_root.
3. Mount all the filesystems that are needed for the container inside its rootfs. These are listed in table 3.2.
4. Depending on the user namespace, if the -U option is not set devices are created through *mknod* otherwise they are bind mounted from the parent. This operation is done because creating a device is forbidden in a *rootless* container even if it has `CAP_MKNOD` in its capabilities set. This is a known limitation when creating a container in a new user namespace because devices are a system resource that is not namespaced.
5. Create all the symbolic links as shown in table 3.1.

From	To
/proc/self/fd	/dev/fd
/proc/self/fd/0	/dev/stdin
/proc/self/fd/1	/dev/stdout
/proc/self/fd/2	/dev/stderr
/dev/pts/ptmx	/dev/ptmx

Table 3.1. Symbolic links in nscage

It is mandatory to perform the actions listed in *point 3* before unmounting the old root (done after the execution of pivot_root). This is due to the presence of a check in the Linux kernel, in particular the `mount_too_revealing()` B.1 function, that doesn't allow to mount a filesystem if it is not already present in the list of mountpoints of the process calling mount. When a new mount namespace is created, the process inside the new namespace receive a copy of the mount points of its parent. This list however is cleared when the old root is unmounted, denying the container to mount any filesystem. So it is nedeed to mount everyhting we need in the container before performing the unmount of the old root. Afer the rootfs has been configured, we are ready to jail the child process inside of it executing pivot_root, unmouting and deleting the old root.

Path	Type
/proc	proc
/dev	tmpfs
/dev/shm	tmpfs
/dev/mqueue	mqueue
/dev/pts	devpts
/sys	sysfs

Table 3.2. Mounted filesystems with their respective types [54]

Network namespace

The network namespace set up comes with a default configuration which consists in the creation of a veth pair 2.1.4 between the newly created network namespace and the host network namespace. *Netlink* [48] has been used to communicate and make requests between the user level process and the underlying Linux kernel. This

software layer exposes a set of API based on top of socket that can be used by an application to communicate with the kernel. A more detailed description of the protocol is reported in *RFC3549* [49].
The following operations are performed to configure the network namespace of the child process allowing it to have internet connectivity:

- Two veth devices are created (a veth-pair).
- An ip address is assigned to the veth device in the parent namespace and it is set up.
- The veth device of the child is moved in the child network namespace
- An ip adress is assigned to the veth device in the child namespace and it is set up as well as the loopback interface.
- A default route is added to the routing table of the container
- Packet forwarding in Linux is disabled by default so it is needed to turn it on.
- NAT need to be configured in order to let containers access the outside world. In this case the NAT will translate the source IP address of the container to the ip address of the physical interface of the host.

These operations correspond to the bash commands shown in listing 2.27 and have been embedded in our C code thanks to the `iptc.h` library.

```
$ #!/bin/bash
$ ip link add veth1 type veth peer name vpeer1

$ ip link set vpeer1 netns ns1

$ ip addr add 10.1.1.1/24 dev veth1
$ ip link set veth1 up

$ ip netns exec ns1 ip addr add 10.1.1.2/24 dev vpeer1
$ ip netns exec ns1 ip link set vpeer1 up
$ ip netns exec ns1 ip link set lo up

$ ip netns exec ns1 ip route add default via 10.1.1.1
$ echo 1 > /proc/sys/net/ipv4/ip_forward

$ iptables -t nat -A POSTROUTING -s 10.1.1.0/255.255.255.0\
            -o eth0 -j MASQUERADE

$ iptables -A FORWARD -i eth0 -o veth1 -j ACCEPT
$ iptables -A FORWARD -o eth0 -i veth1 -j ACCEPT
```

Listing 3.2. Network namespace configuration

3.3.2 Control groups

In `nscage` the following subsystems can be controlled thanks to cgroup:

- `-M`: memory limit
- `-C`: percentage of cpu shares
- `-P`: max pids
- `-I`: io default weight

The parent process, for each resource controller, will create a new cgroup folder that will become the new cgroup root for the child.

Figure 3.1 shows the data structures relationships that implement the cgroups system for the child process.

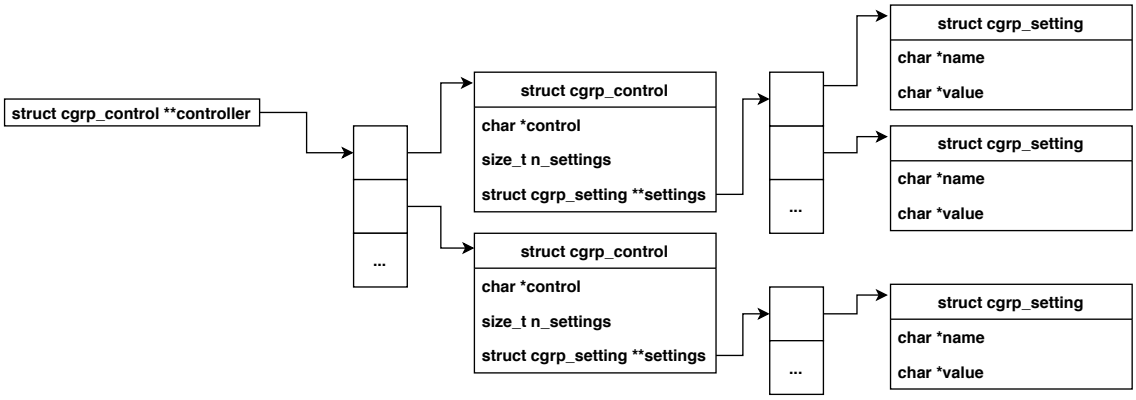

Figure 3.1. cgroup strucutres in nscage

As shown in figure 3.1, the controller represents the cgroup configuration to be applied to the child process. It holds a struct cgrp_control for each resource controller (e.g. cpu, memory...). Each cgrp_control structure defines the limitations for the controlled *subsystem*. Each of these limitations is represented by the cgrp_setting struct (e.g.memory.limit_in_bytes and memory.kmem.limit_in_bytes). For example, if we want to limit the max number of pids for a process to 100, the content of the cgrp_control structure related to the pid subsystem will be like:

- `cgrp_control->control = "/sys/fs/cgroup/pids"`
- `cgrp_control->n_setting = 1`
- `cgrp_control->settings[0]->name = "pids.max"`
- `cgrp_control->settings[0]->value = "100"`

Because we are using *cgroup v1*, a new file called `pids.max`, having "100" as content, will be placed under `/sys/fs/cgroup/pids/<hostname>` (where hostname is the hostname of the container).

To finalize the configuration of the child's cgroup, its pid must be written under `/sys/fs/cgroup/pids/container/tasks`.

To do this operation it is possible to write inside `/sys/fs/cgroup/pids/container/tasks` the pid tasks of the process that is hosted inside the new created cgroup. The value `"0"` can be also written to put the current process inside the new cgroup, it means "the writing process".
Finally, a new pid cgroup root for the new process is linked to the new's one so `/proc/self/cgroup` will contains `pids:/` if it is observed inside the new container. Is important to notice that in the parent cgroup namespace, the file `/proc/[container_pid]/cgroup` is not affected to the cgroup namespace of the child so it will report a pids entry like `pids:/container`.

3.3.3 Capabilities

Starting from Kernel 2.2 a security step about root privileged has been done. Capabilities offer a more accurate granularity instead of having just *privileged* and *unprivileged* processes. Historically, UNIX implementation was characterized by two main categories of processes:

- **privileged**: user ID equal to 0, referred to *superuser* or *root*
- **unprivileged**: user ID different from 0

Privileged process can bypass all kernel permission check, instead an unprivileged process is subject to a full permission checking. Capabilities was released in 1999 (kernel version 2.2) in order to add a more complex root permission management and avoid the historical binary system of privileged and non-privileged processes.
The final goal of capabilities is to split all the possibles kernel calls into groups of related functionalities [45] and assign each of them a slice of privileges. Capabilities are saved into the filesystem (like the bit `suid`).

Why we need to use capabilities?

Capabilities are used to allow userspace processes to gain a slice of root privileges when executed. In this way, any attacks conducted through these executables (for example exploiting a bug) will be limited to the use of a restricted portion of possible system calls avoiding a possible privilege escalation.

Types of capabilities

There are different sets of capabilities that are own by a thread:

- **Permitted**
  This is the subset that can be assumed by the thread.
- **Inheritable**
  This is the set that is preserved after execve.

- **Effective**
  Set of capabilities used to the kernel to perform permission checks for the thread.
- **Bounding** (since Linux 2.6.25)
  This is a mechanism used to limiting the capabilities that are gained after execve.
- **Ambient** (since Linux 4.3)
  This set of capabilities is preserved after execve. Ambient capabilities are added to the permitted set and assigned to the effective set when execve is called.

In *nscage* we decided to implement a capability security layer to reduce the root level of the process into the container, in the case the user namespace is not used. Inheritable and ambient capabilities are cleaned to be empty, in this way isn't possible for the child process to regain (through a file capability) the capabilities that are dropped. Then we defined a bounding set where we listed the capabilities to drop for the child process. An example of the capabilities that decided to drop:

- `CAP_SETID` whithout user namespacing, allows to modify a setuid executable without modify the setuid bit. In this case another program can produce a new executable that can switch setuid to root and `execve` a root bash and make a privilege escalation. [55]
- `CAP_BLOCK_SUSPEND` that prevents the system from suspending.
- `CAP_MKNOD` without user namespacing allows programs to create new devices that con correspond to real devices. With this capability a program can create a new device and associate it to a disk and then mount it. The result is that the mounted disk can be used inside the new namespace. In nscage all devices are created before dropping this capability.
- `CAP_SYS_ADMIN` avoid to use `mount`, `vm86` and other privileged operations.
- `CAP_SYS_BOOT` allows programs to restart the system (`reboot` syscall)
- `CAP_SYS_TIME` programs inside the namespace can not change the system time.

The full list of dropped capabilities is presented in B.1

3.3.4 Seccomp

In nscage a Seccomp security layer is implemented to reduce the set of system calls that can be used by the containerized process. The main goal is to reduce the possible attack surface exposed by the kernel by limiting the syscalls that the container is able to execute. In particular we used the Seccomp-bpf [56] extension, which allows to install a filter attaching it to every system call. In this way, differently from the Seccomp strict mode where only four system calls are allowed (read, write, _exit and sigreturn), it is possible to filter syscalls based on their arguments. When a process

runs with Seccomp enabled, its mode is stored in the file `/proc/self/status` under the field Seccomp. A mode value of 2 indicates that we are operating in seccomp filter mode. If the process inside the container fork itself its child will inherits parent's filter or if it makes a call to execve (as done for executing the entrypoint of the container) the filters are preserved across this call. Just to give an example, we want do deny the creation of a new user namespace from inside the container. This is made by adding a Seccomp rule to our filter that checks if between the arguments passed to clone or unshare the `CLONE_NEWUSER` is present and in the case deny its execution. If the contained process was allowed to create a new user namespace, it would regain, inside that namespace, the capabilities previously dropped. The seccomp filter's context has been initialized with the `SCMP_ACT_ALLOW` flag in order to implement a black listing policy, so the filter will have no effect on system calls that does not match our security profile. In the case the process calls a black listed system call it will receive an EPERM return value.
The Docker default Seccomp profile, which can be found in the Docker's documentantion [58], provides a reasonable list of system calls that can lead to security issues. Some of these restrictions anyway have already been enforced by dropping related capabilities (e.g. a call to mount will fail anyway since it requires the `CAP_SYS_ADMIN` which can be dropped from the capabilties set of the process).

Chapter 4

Performance comparison of secure container runtimes

The use of a shared kernel represents one of the biggest risks related to containers' security. For this reason, most cloud infrastructure managers have opted to use an approach base on virtual machines to create a secure container execution environment. The main disadvantage of this approach is that you lose all the advantages of flexibility and ease of management of containers because several virtual machines need to be managed and configured manually. For this reason, in recent years, solutions have been developed that maintain the ease of container management and guarantee a high level of security. Obviously, everything has a cost and the addition of different levels of indirection to enforce container security introduces an overhead in terms of resources consumed and execution performance.
In this chapter an analysis is made to quantify this added cost. For this study two container runtimes were analyzed that implement additional security mechanisms such as *runsc* (part of the gVisor project) and *kata-runtime* (part of the Kata-containers project). The results obtained with these two new approaches based on security were compared with those obtained using two traditional container runtimes such as *runc* (Docker's default container runtime) and *crun* (developed by RedHat).

4.1 OCI compliant container runtimes

A container runtime is the actual layer of software that takes care of container creation and lifecycle management. In 2015 docker together with other large container companies defined the so-called Open Container Initiative - OCI. The OCI currently includes two specifications: the Runtime Specification (runtime-spec) and the Image Specification (image-spec). For the purpose of our thesis and the replicability of our experiments we will only introduce the Runtime Specification. The OCI Runtime-spec defines a set of interfaces that a runtime must have in order to be compliant with the standard. The standard defines different operations over containers that need to be implemented(State, Create, Start, Kill, Delete, Hooks) and also the states of the container lifecycle(created, running, stopped).

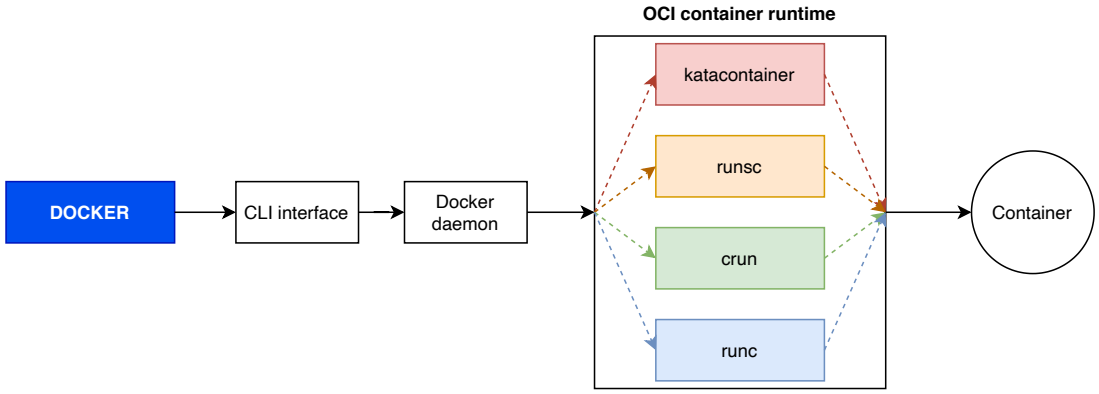

Figure 4.1. OCI compliant container runtimes

Additional states or operations can be defined by the runtimes but they must be different from the ones reported in the runtime specification. For example runc together with Docker defines additional state such as Paused and Stopped and new operations to manage the container in these new states like pause and resume.
The biggest advantage of introducing this standard is that using the same container manager we can change different runtimes simply by changing the appropriate configuration file, without changing the command used to interact with containers.

4.2 Kata Containers

Kata containers [70] is an hybrid container runtime that integrates the strong isolation obtained by a traditional virtualization approach with the efficiency and flexibility of containers. It was born from the merge of two open source projects: Hyper runV and Intel Clear Containers. It takes advantages from both the projects having

good performance (Intel Clear Containers) and supporting multiple CPU architecture with different hypervisors (Hyper runV) [71]. The idea of this technique is to isolate each container instance using a minimal Virtual Machine with a dedicated kernel. By doing this containers do not share anymore the same kernel, which represented the main security drawback, because they have their own isolate execution context enforced by a virtual machine. Figure 4.2 shows how the Kata container solution ensure isolation by adding a level of indirection using traditional virtualization technologies, in this case hardware assisted virtualization. Thanks to this additional level of isolation, if malicious code is executed inside a container, it could no longer use the shared kernel of the host machine to compromise other containers, because it is confined in its guest kernel. Actually, using traditional virtualization to ensure the isolation of containers is not a new solution. Most cloud infrastructures today take advantage of the creation of virtual machines to enforce isolation at hardware level, providing a secure execution context where to run multiple containers. Doing so, however, you lose all the advantages that characterize the use of containers. Kata container's goal is to bring these two technologies together while maintaining all the advantages of containers. This approach give us the possibility to bring the flexibility of the containers ensuring and high level of isolation typically associated with a Virtual Machine.

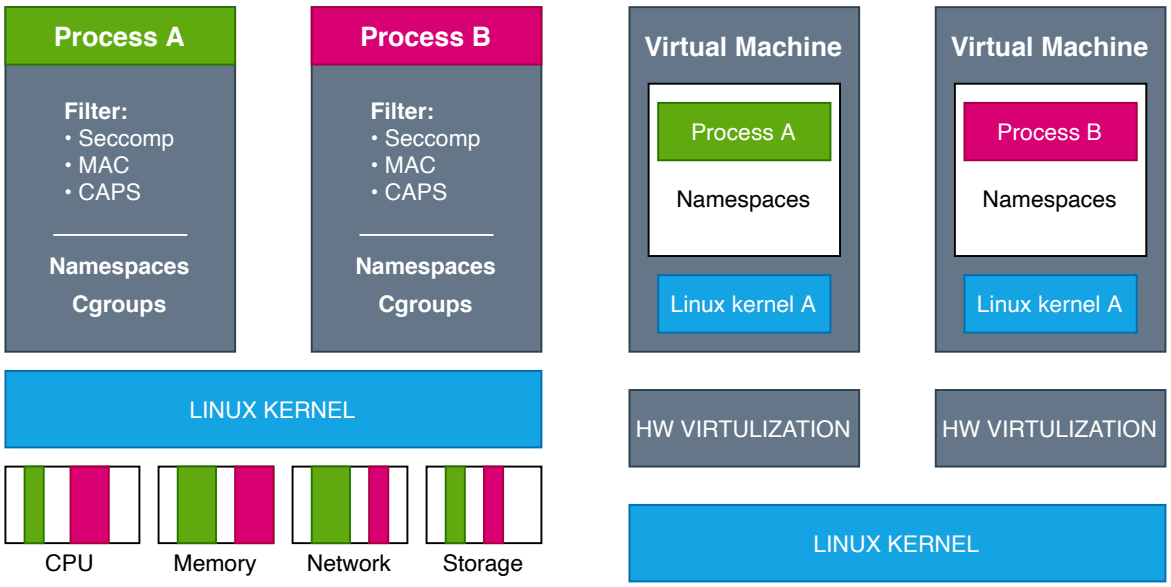

Figure 4.2. Kata Containers isolation (inspired from [66])

Kata Containers virtualization

Kata Containers uses an hardware-assisted virtualization approach to support the execution of virtual machines. For this reason a CPU that supports the Intel VT-x extension is needed. Kata Container can be configured to run with different Hypervisors but for the purpose of our thesis we focused in its implementation with QEMU/KVM (which is the default for Kata). As reported by Rick Donato in [67], QEMU is a type 2 hypervisor, sometimes referred as userland Hypervisor since it runs on top of an host, that is used for hardware emulation and virtualization. This must not be confused with hardware assisted virtualization (exploited by KVM) since QEMU is used to emulate different physical devices like CPU, disk, USB etc. QEMU can be used on its own to virtualize all the resources needed to run a virtual machine but since this process is performed totally in software it is rather slow. For this reason it is used in combination with KVM that, by the use of hardware assisted virtualization, ensures better performance taking the role of an accelerator. KVM is a Linux kernel moudule that acts as a bare metal hypervisor offering a full virtualization solution for Linux on x86 hardware supporting virtualization extensions (Intel VT or AMD-V).

Kata-runtime, which is the container runtime of the Kata Containers project, creates a QEMU/KVM virtual machine for each container that is going to be created. As shown in listing 4.1, after creating a container through Docker specifying to use kata-runtime configured with QEMU/KVM, we can see a process running the qemu executable. This is the process representing the virtual machine that hosts our container. Moreover it possible to notice that kvm is passed as an accelerator to qemu.

```
#!/bin/bash
$ docker run --rm --runtime=kata-qemu -d ubuntu /bin/bash
$ ps -ef | grep qemu

root 10206 [...] /opt/kata/bin/qemu-system-x86_64 -machine pc,
accel=kvm
```

Listing 4.1. Kata containers virtualization

Kata Containers architecture

The Kata Containers project is made of three main packages which are described in [72] and by Sebastien Boeuf in [73]:

- **Kata-runtime**
  The kata-runtime is the container runtime. It is fully compliant with the OCI

specification in the creation of a container and the management of its lifecycle. It also implements some other commands that are not part of the OCI specification but that are actually implemented in runc (the default container runtime for Docker) and that Docker assumes. Kata-runtime uses a gRPC protocol to communicate with the agent in order to forward container management commands. This protocol is also used bring the stdout, stdin and stderr from the container manager (eg. Docker) to the container and vice-versa. The kata-runtime is responsible for the creation of the QEMU/KVM virtual machine and committing the actual creation of the container to the kata-agent process inside the VM.

- **Kata-shim**
  After a container is created, it should be possible to interact with him for monitoring or I/O purpose and this is possibile by referring to it by its PID. The problem with Kata container is that the container its actually spawned inside a Virtual Machine and we can't access it on the host by its PID. For this reason the Kata-shim has been introduced to mimic the container behaviour on the host side. Thanks to the kata-shim container managers can interact with the container inside the VM as they would do with a normail container.

- **Kata-proxy**
  The Kata-proxy is a component that is created for each Virtual Machine and it is used to handle multiplexing and demultiplexing of container management commands and I/O streams.

Figure 4.3 graphically shows all the components previously described. Particular attention should be given to the process called kata-agent inside the VM. This process acts as a supervisor for containers and their processes. Its role is similar to the role that runc has inside docker, in fact it uses libcontainer for the creation of containers and the management of their lifecycle (sharing most of the code with runc). It's execution context is called a sandbox and it is defined by a set of namespaces (NS, UTS, IPC and PID).

As an example (inspired from [72]) by executing the following command *docker run ubuntu echo "Hello world"* the following steps are performed:

1. The hypervisor uses the guest kernel to load a minimal root file system for the guest O.S. This is also called a mini O/S. The only service running in the context of the guest O.S is the init process.

2. The init process will start the kata-agent.

3. The kata-agent process is responsible for the creation of the execution context for the container that will run the echo "Hello world" command. The container is created in the same way as runc does.

4. The kata-agent will the set the rootfs (ubuntu in this case) for the container and then execute the echo "Hello world" command inside the container.

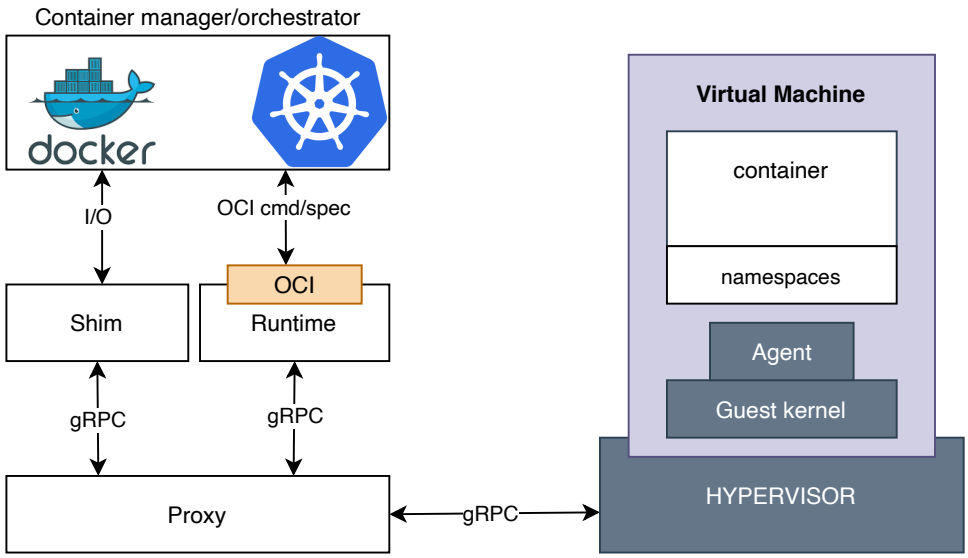

Figure 4.3. Kata Containers components (inspired from [66])

4.3 gVisor

gVisor [74] [75] is an open source application kernel developed by *Google* that implements a new approach of provisioning a virtualized environment moving system calls interfaces, that are normally implemented by the host kernel, into an user space kernel. In particular, it implements more than 200 Linux system calls in a memory safe sandbox. In fact, to avoid memory issues like memory leaks, this project is written in *GO* so that it can take advantage of its memory sandbox. A memory safe application like gVisor has the following security features:

- strong types
- built-in bounds checks
- no unitialized variables
- no use-after-free
- no stack overflow
- built-in race detector

A component of *gVisor* is an *Open Container Initiative* (OCI) compliant runtime called *runsc* that makes it easy to work with other container managers like *Docker* or *Podman*. In order to provide a strong isolation layer between containers and the host Kernel, there are two different approaches:

- **Machine-level virtualization**: A virtualized hardware is exposed to a guest kernel via a Hypervisors. For example, using *KVM* or *Xen*.

- **Rule-based execution**: A fine grained security policy is applied to an application container using solutions like Seccomp, SELinux or AppArmor.

gVisor provides a third isolation mechanism. In particular, each application system call is intercepted and executed by a new independent kernel. In this way it is possible to provide a flexible resource footprint but the price to pay is the application compatibility and an higher per-system call overhead. The high level architecture of gVisor is showed in 4.4

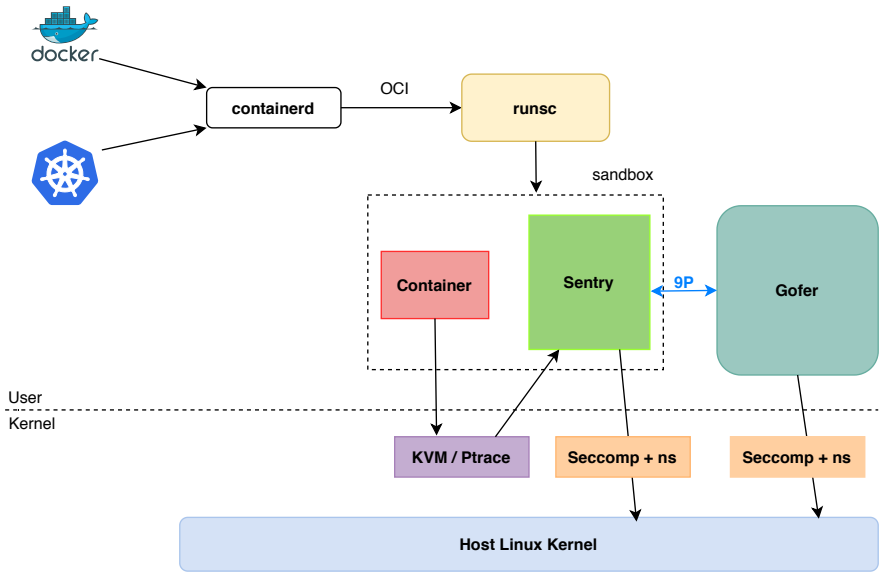

Figure 4.4. gVisor architecture I (inspired from [68])

4.3.1 Internals

gVisor provides a sandbox where containers can run in isolation, it is composed by a set of processes:

- **Sentry**: This component is the kernel that runs the container, it intercepts and responds to system calls made from the application. A single **Sentry** is present in each isolated sandbox. It is the larger component in gVisor. It is an application kernel that implements functionalities like: system calls, signal delivery, memory managment, page fault logic, threading model and more. Each system call is intercepted by a component called "*Platform*" [79] and then redirected to the *Sentry* component. The default *platform* is *ptrace* [81] used with the request `PTRACE_SYSEMU` as argument but gVisor can be also configured to use *KVM* [82]. During this process the system calls do not pass through the host system kernel, them are redirected to sentry that will execute

the functionalities. After this phase, as a normal user space application, Sentry will produce system calls to the host kernel in order to support these operations but does not allow the application to control them directly. Filesystem access calls are executed by the *Gofer*.

- **Gofer**: This component provides file system access to the containers. Each *Gofer* process is started when a new container is created. It communicates with the *Sentry* process using the 9P network protocol [80] developed by *Bell Labs*. The *Gofer* mediates each access to the filesystem tree providing and extra layer of isolation.

Netstack

gVisor implements a dedicated user space network stack that was written from scratch exclusively for the project. It is called *Netstack* and it is totally written in *GO*, it was designed to be modular, flexible and self-contained. Networking grows quickly and unfortunately *Netstack* is actually one of the performance bottleneck of the entire project. The causes are related to the fact that many performances related RFCs like `RACK` [86] and `BBR` [87] (that is a *congestion control algorithm*) are not yet implemented. On the other side, it allows to reduce drastically the amount of system calls that sentry uses to communicate with the host. In particular it uses only three system calls to write and read packets. Anyway gVisor offers the possibility to bypass the own network stack and uses the host's one. This functionality is called *passthrough* and unlock the possibility to use fifteen additionally syscalls from Sentry to the host kernel. Moreover this allows to create file descriptors that exposes the Sentry to a file-based attach. Using this feature, gVisor will starts to use the socket API (that normally is not used) increasing the attach surface.

4.3.2 Costs

In general, regarding performances, gVisor introduces a cost over native containers that can be divided in two main categories [76]:

- **Structural costs**: This type of costs are given by the design choice introduced by gVisor. For example, Sentry require a certain amount of userspace memory to run correctly.
- **Implementation costs**: Sentry implements a system calls surface that compare to more mature solutions introduces an impact in term of performance on the execution of the system calls. An example is the gVisor network stack (called *netstack*) that is less CPU efficient than the native one.

The full list of syscalls that are supported by gVisor is reported in the documentation [83][84][85]. Note that an user space application is compatible with gVisor only if it uses only syscalls that are implemented in *Sentry*.

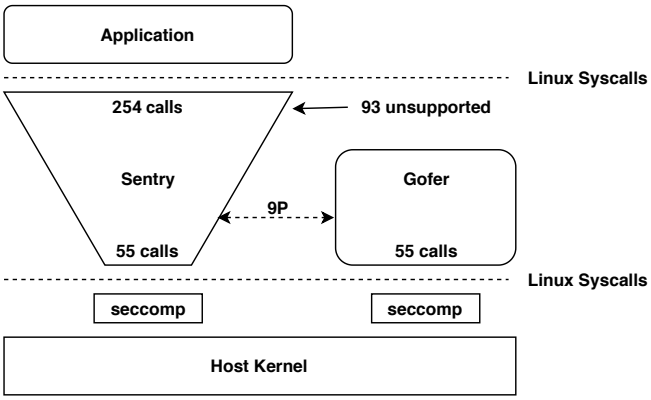

Figure 4.5. gVisor architecture II (inspired from [69])

4.4 Benchmarking

In this section we will illustrate the setup that we used to execute the tests. For each system resource that we analyzed we give a brief description of the tests executed to stress that resource and the related metrics that are of our interest.

4.4.1 Testing environment

The system configuration that we used for all the following tests is composed as follows:

Component	Specifications
CPU	Intel Core i5 6600
RAM	2x4GB DDR4 2133Mhz
Storage	SSD 32GB
OS	Ubuntu Server 18.04.5 LTS
Linux Kernel	4.15.0-123-generic
Docker	version 19.03.6, build 369ce74a3c
Podman	version 2.1.1
runc	1.0.1-dev
crun	0.15.1.3-7d65
runsc	1.0.1-dev
kata-runtime	1.11.1

Table 4.1. System configuration for the benchmark

We decided to install two different container manager:

- Docker
- Podman

And the following container runtimes:

- runc
- crun
- runsc from the gVisor project
- kata-runtime
  - kata-qemu (kata-runtime using 9pfs as shared filesystem)
  - kata-qemu-virtiofs (kata-runtime using viritofs as shared filesystem)

In the next sections we will use the term *sandbox* as an alternative to *container*. The reason is that with Kata-containers and gVisor, as opposed to runc and crun, the container execution environment is made of multiple processes that are needed by the runtime architecture itself.

4.4.2 Boot time

Single container boot time

The purpose of this test is to understand how long does it take to create a container depending on the container runtime that is used. To run this test we have developed a *bash* script that take into consideration four different times:

1. **t_0**: represents the origin time.
2. **t_c**: the time the container is created.
3. **t_s**: the time when the container was last started.
4. **t_f**: the time when the container last exited.

These times were taken thanks to the *docker inspect* command, which returns a JSON array containing different information about the container. From the JSON we extract the fileds *Created*, *StartedAt* and *FinishedAt*. By taking these times we were able to compute three interesting measures:

1. **t_created** $= t\_c - t\_0$
2. **t_started** $= t\_s - t\_c$
3. **t_execution** $= t\_f - t\_s$

These refer to the time needed to create the container, perform its start up and the time taken by the command (specified in docker run) to be executed inside the container. The script takes as parameter the container runtime and the number of iteration the test will be executed. The result is the average calculated on the total number of iterations. To evaluate the boot time of a container we are not interested in the execution time of the command so we just execute the /bin/true executable inside the container, which simply returns true.

Real boot time on a set of containers

The aim of this test is to measure the time taken to create multiple containers, in particular seventy instances. To measure the startup time of a container we used `/usr/bin/time` and the real time was computed as the sum of user and kernel time spent.

4.4.3 CPU

CPU prime

The cpu performance has been tested with sysbench using the command shown below.

```
1 $ #!/bin/bash
2 $ sysbench --test=cpu --cpu-max-prime=200000 run
```

Listing 4.2. CPU performance benchmark

This benchmark consist in a CPU intensive workload to compute all the prime numbers in the range specified by the *–cpu-max-prime* field.
The metric we are interested in is the number of events per second consumed by the CPU.

4.4.4 Memory

The tests that compose the memory test suite are listed in table 4.2, which gives a short overview about the metrics we are interested in.

Memory footprint

The aim of this test is to compute the memory footprint for a single instance of a sandbox. Here for memory footprint it is intended the amount of memory required by the entire sandbox to run a container (in the case of runc and crun its just the container itself). Each container is created from an Ubuntu image and the

Test	Purpose
Memory footprint	Compute the memory used by single instance of a Sandbox.
Memory access	Compute the operations per second performed in raw memory accesses.
Memory allocation	Compute the time needed to allocate 1GB of memory with different chunk's sizes.
Memory usage	Compute the overall system memory utilization of N sandbox instances running in parallel.

Table 4.2. Memory test suite

entrypoint is /bin/bash. Containers are started in detach mode (`-d` option) and to prevent them from exiting we have kept their standard input open via the `-i` option. The memory usage for each container is taken from its cgroup associated with the memory controller. Despite this information is accessible from the cgroup fs under `/sys/fs/cgroup/memory/docker/<container-id>/memory.usage_in_bytes` it is better not to refer to this value as suggested in [92].
Instead we can obtain the memory used by the sum of *rss*, *cache*, and *swap* fields from the `/sys/fs/cgroup/memory/docker/<container-id>/memory.stat` file. The test was repeated multiple times (N=20) and we take the average value of memory used for a single container instance.

Cgroups in Katacontainer

Since the Kata-containers architecture implies many different componenets (Virtual Machine, kata-shim, kata-proxy) for the management of a container, plus an additional process (virtiofsd) in case of kata-qemu-virtiofs, to have a correct measure of how much this ecosystem actually costs, it is necessary to put the overall sandbox created inside a single cgroup. This is possible by setting the `sandbox_cgroup_only=true` [103] flag inside the kata-runtime configuration file under `/opt/kata/share/defaults/kata-containers/configuration-<runtime-name>.toml`.

Memory usage

This benchmark is based on a sequence of 50 containers that are executed in parallel and in detach mode using docker as container manager. The image that is used for the test is *alpine* and the entrypoint is the default one: `/bin/sh`.
The purpose of this test is to compute how the memory utilization of the system

changes by increasing the number of container instances being run.

Memory access

To test the raw memory accesses we used sysbench. The command is reported below:

```
$ #!/bin/bash
$ sysbench --test=memory --memory-block-size=4k \
  --memory-total-size=100G --num-threads=1 run
```

Listing 4.3. Raw memory accesses benchmark

This test will allocate a buffer of size *–memory-block-size* and then performs a read or write operation until reaching the total size specified by the *–memory-total-size*. The default operations performed are sequential writes but this can be changed through the *–memory-oper* and *–memory-access-mode* options.
Note that for this test, since there are no differences in the memory management system of kata-qemu and kata-qemu-virtiofs the results will only show one kata-runtime configuration. The metrics we are interested in is the number of operation performed per second.

Memory allocation

To stress the memory management system we developed a little program in C that allocate a buffer multiple times using malloc, until reaching 1GB of allocated memory and without freeing it. The test was performed multiple times changing the size of the allocated buffer. As for the memory access test the results will only show one configuration of kata-runtime. The metrics that were collected are the number of allocations per second and the total time needed to allocate of 1GB of memory.

4.4.5 Networking

Net I/O

The networking performance has been tested using `iperf3` in different configurations. In particular, the download and upload throughput has been measured as follows:

- to measure the **download** speed an `iperf3` server has been placed in a container using the four different runtimes. The `iperf3` client has been always placed in a container using runc which uses the host kernel network stack.
- to measure the **upload** speed, a `iperf3` server was placed in a container using runc while the client was executed in another container using the four different runtimes.

Using the `-J` flag of `iperf3` it is possible to retrieve the test result in JSON format and then analyze it. The observed metric is the throughput in Gbit per second.

Round Trip Time

In this test we measured the average Round Trip Time elapsed between the host and a container booted with different runtimes. The result is the RTT average on 100 ICMP echo requests between two containers started by the same runtime.

4.4.6 System calls execution time

This test was developed to compute the overhead introduced by each sandbox int the execution of different system calls. In particular we launched one million system calls (`getpid()` [88], `getcwd()` [89] and `fopen()` [90]) under different scenarios and we measured the average elapsed time of execution.
We decided to use those system calls because each of them is executed differently in gVisor. There are three main ways in which a system call can be managed by Sentry:

- Execute the system call directly in Sentry (getcwd).
- Execute the System call by making a call to the host kernel (getpid).
- Execute the system call trough Gofer (fopen).

The metric observed is the average execution time of each system call. Note that for this test each file has been placed in a shared volume with the host because talking with the gVisor community we found that there is a difference when opening a file. That's because the root filesystem is for exclusive use of the container and can be cached more aggressively. While files in container mounts can be changed externally and require revalidation on every access.

4.4.7 I/O

Testing the I/O by performing reading and writing operations over files has been one of the most complex benchmark, because there are a lot of different factors to be taken into consideration (which are strictly dependant on the architecture adopted by each runtime). To perform this test we used FIO [101] using this command:

```
$ #!/bin/bash
$ fio --filename=<filename> --direct=0 --buffered=1 \
  --name=seq_write --fallocate=none --ioengine=sync \
  --bs=<block-size> --invalidate=1 --thread=0 --numjobs=1 \
  --group_reporting=1 --size=1G --filesize=1G --rw=<mode>
```

Listing 4.4. File I/O benchmark

As the command shows, we tested buffered I/O (`-buffered=1 -direct=0`), this means that every read and write operation is performed thanks to the page cache or buffer cache (unified starting from Linux kernel version 2.4). This choice is motivated by two main reasons:

1. Talking with different developers of the Kata-containers project we find out that direct IO has none effect in host cache for shared filesystem (the container's rootfs is shared between the host and the guest thanks to 9pfs or virtio-fs filesystem). This means that, when testing direct IO with kata-runtime, having a virtual machine implies two level of page caches (guest and host page cache). Using the `-direct=1` option in FIO the host page cache is still used, resulting in kata-runtime performing better that runc and crun (this has no sense), which correctly bypass the only page cache available to them. This actually depends on the disk cache mode the qemu VM is started with, in the case of Kata-containers is Writeback [98]. With this cache mode, when a guest container open a file with the `O_DIRECT` option, write operations are reported to it as completed when the host page cache is written [96].
2. As reported here [102], the open system call actually does not support opening files with the option `O_DIRECT`, leading to the impossibility of testing direct IO with runsc.

For these reason the only way we had to make equal comparison was to use buffered I/O. This also preclude testing asynchronous I/O (`-ioengine=libaio`) since, as reported here [97], using AIO on file opened without `O_DIRECT` lead to non asynchronous behaviour. Instead, synchronous I/O has been tested (`-ioegine=sync`) to run fio tests with buffered I/O enabled (`-buffered=1`).
Another problem we had was that for reading operations, kata-qemu and kata-qemu-virtiofs performed better than runc or crun (also this has no sense). The problem is probably related to the fact that FIO, before running a reading test, has to layout the file. Then to avoid reading it directly from the cache, FIO provides the option `-invalidate=1` to invalidate caches before reading. As said before, this had no effect on the host cache for shared-fs in Kata-qemu resulting in better reading performance than runc and crun. The same thing applies for Sentry in runsc since it implements its own memory management. One solution we found was to avoid laying out the file before reading it and instead use an already existing file.
With regards to kata-runtime, the test were performed with the configurations shown in table 4.3.

By setting the cache mode to *none* the guest page cache is disabled resulting in any data changes being pushed to the host immediately. Actually, virtio-fs supports different caching mode (other than none) that can be set using the configuration file of kata-qemu-virtiofs. But this is not possible using 9pfs, as reported here [98], which by default uses none.
By bypassing the guest page cache we can make an equal comparison with runc and crun, since both solutions (runc/crun and kata-qemu) will see their IO operations

runtime	cache mode	DAX
kata-qemu (9pfs)	none	not used
kata-qemu-virtiofs	none	disabled

Table 4.3. Kata-runtime configuration

as completed when they hit the host page cache. This also applies in the case of gVisor, since as reported here [99] and here [100] (under Files section), read and write operations (performed by the selected sync ioengine) for files whose host file descriptors is available, does not use the Sentry page cache.

4.4.8 Real application benchmarking

Machine learning model training

This test consist in a CPU-bound workload. It performs the training of a Convolutional Neural Network (CNN) made of two convolutional layers and three fully connected layer. To build and train the CNN we used the pytorch deep learning library (in its CPU version, which allows to train a model only by means of CPU). The CFAR10 database has been used. The CNN is quite small, due to the simplicity of the dataset we used but can unsure a good stress workload for the CPU. The aim of this test is to compute the time needed to train the neural network.

Redis

Redis is an open source in-memory key-value store database written in C. It is actually the most used key-value store database in the world and it is maintained by Redis Labs. It supports complex data structures (such as hashes, lists, sets and more), on-disk persistence, high scalability jobs and typically it is configured as cache to support near real-time performance applications.
This software is also equipped with an integrated benchmarking tool that allows to test the operation of the server under different conditions. In particular, we decided to test the performance using a single container running *redis*. The final score is measured in *request per second* (ops/s) handled by the server. We configured the benchmarking tool to spawn fifty clients that each ran sixteen operations in parallel until each operation is executed one million times. These operations are executed in parallel like in a real scenario and each client does not wait for the server's response before sending a subsequent request. Each test is executed twenty times for each runtime.

Apache Spark

Apache Spark is an open source distributed general purpose computing framework used for big data processing. It can be used in different configurations and its is possible to use it with different APIs and programming languages. To test the performance of this application we executed a *word count problem* [94] on a file of 5GB. The experiment was deployed in two different configurations:

- single worker configuration using the official Jupyter image present in the Docker Hub [95].
- a standalone cluster configuration with two workers.

The API that was used for the test is PySpark. The metrics that we analyzed are: the CPU usage (inside and outside the container) and the execution time to complete the Job. For each runtime the test has been executed five times.

Tea Store

Teastore [77] is a distributed micro-services based application, developed by the Descartes Research Group at the University of Würzburg, that can be used for benchmarking and testing purpose. It emulates the behaviour of a web shop and its architecture is made of five different services, each with a specific performance characteristic: WebUI, Authentication, Recommender, Persistence Provider and Image Provider. Additionally to these five services, a registry service is used to manage the communication and the instances of all other ones. Moreover a database is required, in this case MariaDB is used. The easiest way to deploy the TeaStore application (and also the one that perfectly fits our thesis's purpose) is to use Docker. The services can be deployed in a single container or it is also possible to use a container instance for each service (being closer to the architecture of an application based on microservices). The second configuration has been adopted since it is suggested for benchmarking and testing. To test the application we used Apache JMeter [78] with the default stress script provided by the developer of Teastore. The script include the execution of 8 different operations (e.g. login, list products, add product to cart etc.) and we configure it with a *loop_count* of 1000 and 5 concurrent clients. This means that a total of 40000 (1000*8*5) operations will be executed in parallel by 5 different clients. The metrics we are interested in are: the time to manage all the 40000 request, the average number of requests consumed per second and the error rate.

4.5 Results

4.5.1 Boot time

Single container boot time

Figure 4.6 shows the result obtained by the computation of the boot time of a single container instance per container runtime. It is evident that, with respect to traditional solutions (runc e crun), the hypervisor based approach proposed by Kata-containers is the most expensive in terms of time. This overhead is mostly due to the fact that the creation of a container via kata-runtime requires a virtual machine to be created and moreover the startup of kata-shim and kata-proxy processes. It is also noticeable that there are no significant differences between kata-qemu and kata-qemu-virtiofs. On the other hand runsc (gVisor) introduce a little overhead being 12,7% slower than runc. Its booting process requires the startup of the Sentry (which represents the sandbox itself) and the Gofer process, but anyway it is less heavy than starting up a Virtual Machine. As we expected, crun performs better that any other solution having been developed in C to achieve better performance.

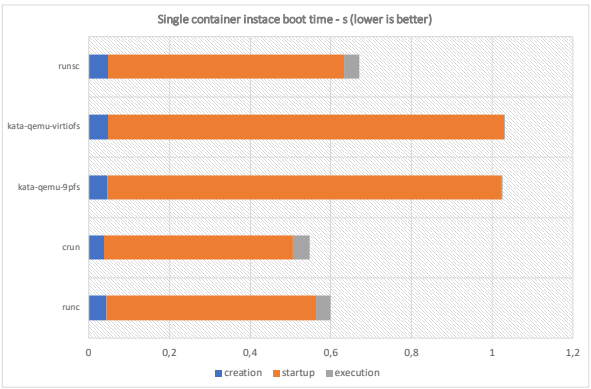

Figure 4.6. boot time

Multiple containers real boot time

Image 4.7 shows a sequence of seventy container instances executed in parallel. Is interesting to note that the deploy time increases considerably when the hypervisor based solutions reach the memory limit of the system. In this case it starts to use the swap file and the performance of the system is adversely affected.

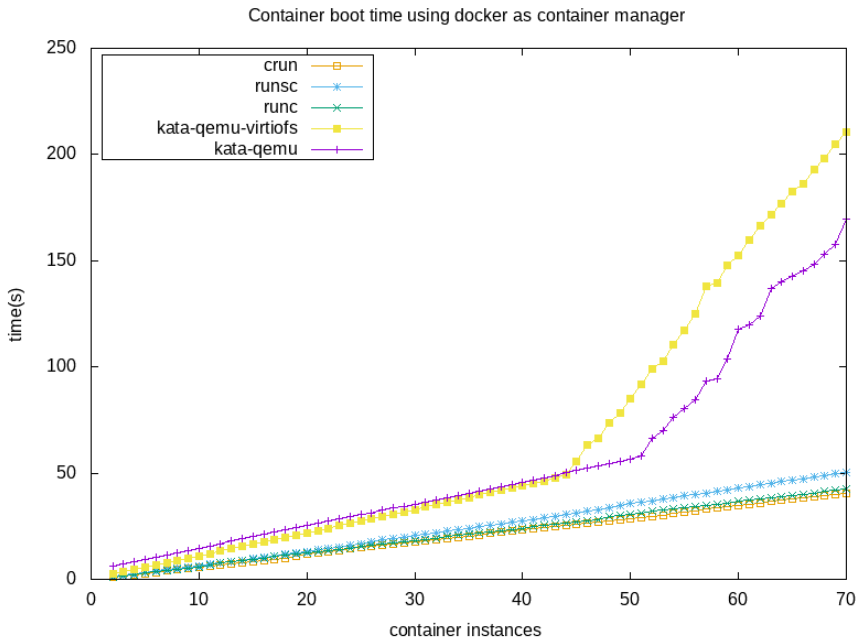

Figure 4.7. boot time on 70 container instances

In this test, the difference regarding the real boot time between runc and crun is minimal. With regards to gVisor the overhead introduced is limited and only becomes significant when the number of container instances grow. Regarding Kata-containers, the impact on the real boot time is considerable compared to others. Classic container-based solutions are therefore more scalable than the hypervisor-based runtimes.
Anyway the boot time is also influenced by the container manager that is used to start the container. In the figure 4.8 is possible to see that starting seventy containers in a row using *Podman* as container manager, the real time elapsed is different changing the container manager. In particular, in this case we used *Podman* to compare its result with the result of docker. The difference to the 70th container is of 11.36 seconds. In order to measure the boot time of each runtime, we would have to start the container using only the runtime but we wanted to put ourselves in a real scenario starting the procedure from a container manager. The difference in the result obtained between the use of Docker or Podman is given by the fact that Podman is a deamonless application, instead Docker must first communicate with *dockerd*.

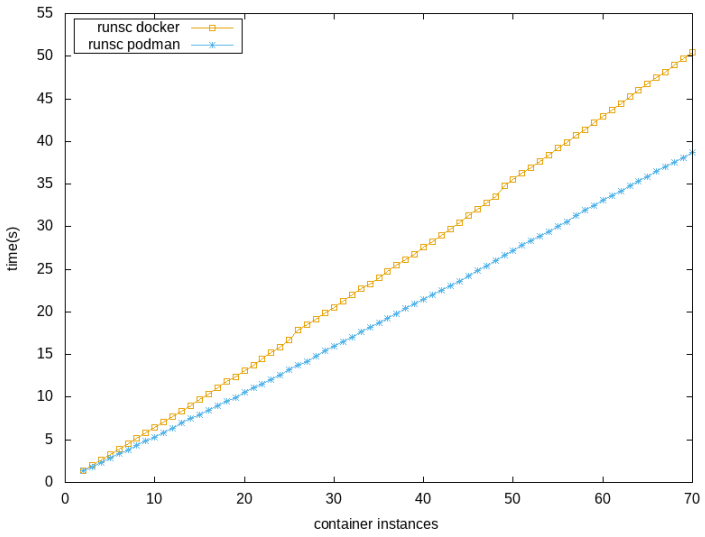

Figure 4.8. real time using two different container manager on the same runtime

4.5.2 Memory

Memory footprint

Figure 4.9 shows the memory footprint for each container runtime. As previously noted runc and crun are the less expensive as they require only the container process itself to run, with a memory requirement of about 50KB. The biggest discrepancy can be seen using runsc and kata-runtime (in both its configurations). This overhead, with respect to traditional runtimes, is mainly due to the fact that the execution of a container created through runsc or kata-runtime (*9p* or virtiofs) requires the presence of additional processes to ensure the security features on which they are based and to support the runtime's architecture. Runsc needs to allocate memory for the Sentry and the Gofer for each container that is created, having a memory footprint of about ~8MB. Kata-runtime introduces the biggest overhead in terms of memory since it requires a virtual machine (a *QEMU* process), a *kata-shim* and a *kata-proxy* process for the execution of a container.

One interesting thing to note is how kata-qemu-virtiofs has a greater impact on memory (~105MB) than kata-qemu (~92MB).This results is justified by the presence of an additional process called virtiofsd, which represents the virtio-fs daemon running on the host side.

Memory usage

Figure 4.10 shows how the different memory requirement of each runtime have significant impacts in a scalability scenario. In particular the memory overhead introduced

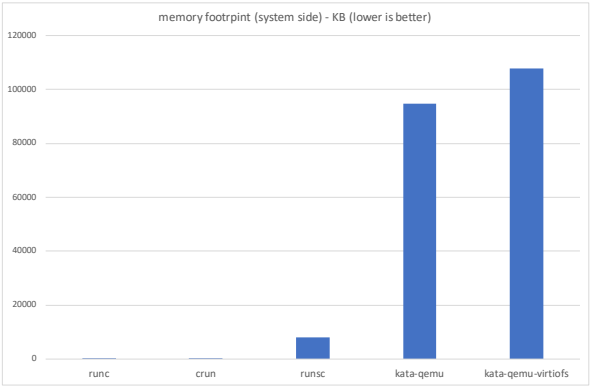

Figure 4.9. Memory footprint of a single container instance

by runc and crun is limited allowing an higher density of containers instances on a single machine. Additional processes used to create the gVisor sandbox (Sentry and Gofer) has a limited memory overhead making runsc a good solution when an high number of container instances are required. In fact after 50 containers only 5,61% of the total memory of the system is used. Kata-runtime is instead hungry for memory and this affects the maximum number of container instances that can be executed simultaneously. Kata-qemu-virtiofs and kata-qemu have a memory requirement that is respectively 2155x and 1897x times greater than the average of runc and crun.

In particular kata-qemu-virtiofs has a bigger impact on memory than kata-qemu, requiring after 50 containers are created, 6.3GB of memory against the 5.1GB taken by kata-qemu. This can also be seen in figure 4.7 where after the creation of just ~45 containers, kata-qemu-virtiofs starts using the swap file, affecting the startup time and the overall performance of the system.

Memory access

The graphs reported in figures 4.11 4.12 4.13 4.14 show the result obtained by testing raw memory accesses with sysbench. As expected crun and runc have not significant differences. Gvisor has the highest overhead in both readings and writings. For what concerns sequential writes gVisor tends to perform about 11% less ops/s than runc. This overhead decreases to 7% increasing the block size up to 128KB. With regards to random writes, the overhead increases considerably. In fact gVisor performs about 62% less ops/s than runc. As noted also for sequential writes, gVisor performance improves with a larger block size, reducing the overhead to 47,4%. The same behavior is reflected in reading operations.
The overhead introduced by gVisor in accessing memory is mainly due to the fact

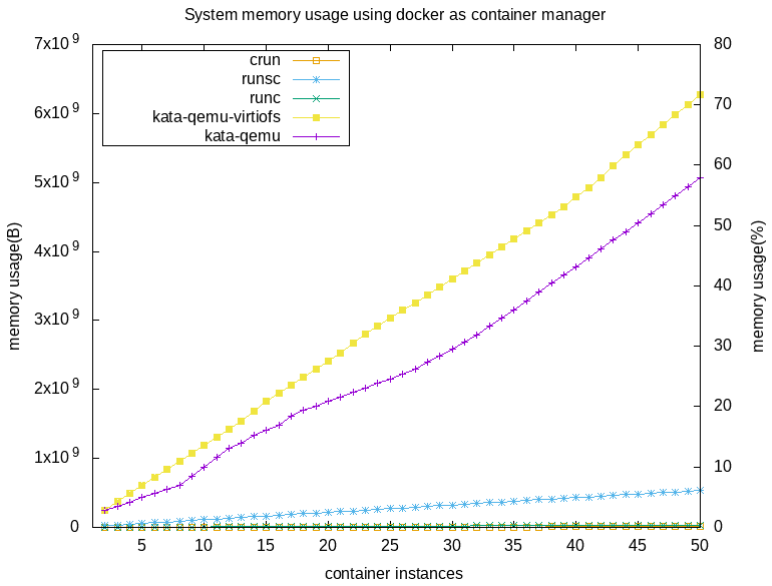

Figure 4.10. memory benchmarking test over 50 containers using different runtimes

that it implements its own memory management system (demand-paging) in Sentry, resulting in two level of mappings for each memory page, one from the guest application to the user space kernel (Sentry) and another from the user space kernel to the host. On the other hand, the use of a VM, as in Kata-containers, has a reduced overhead in memory accesses. As with gVisor, the higher penalty is achieved with smaller chunk sizes. In particular, with Kata-containers are made 3% less sequential writings per second than runc but this difference goes to 0 increasing the size of the buffer. Moreover there is no significant overhead at all in random writings. In conclusion, with Kata-runtime it is not introduced any significant overhead. This can be justified by the use of hardware memory virtualization technology in QEMU/KVM that removes the needs of a two level page mapping. In particular QEMU/KVM use Intel extended page table(EPT) thanks to which it is possible to map directly guest physical addresses to host physical without the need of traversing the host page table. So, once the mappings are defined, there is no additional overhead in accessing memory.

Memory allocation

Figure 4.15 and 4.16 show the results obtained by running the memory allocation test. Similarly to what obtained with the memory access test, gVisor introduce the biggest overhead. Kata-qemu instead performs as fast as runc and crun introducing a very little overhead with respect to them, of about 12% for block sizes less or equal

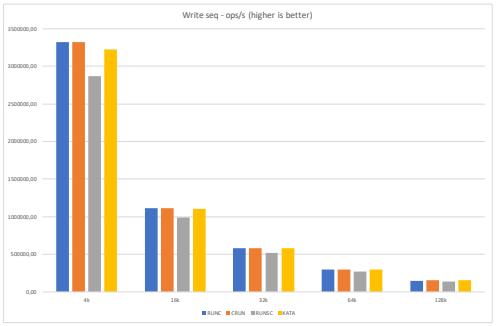

Figure 4.11. Memory sequential write

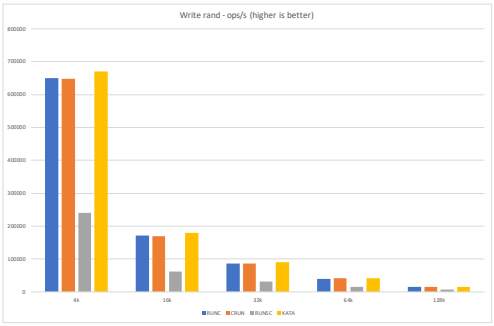

Figure 4.12. Memory random write

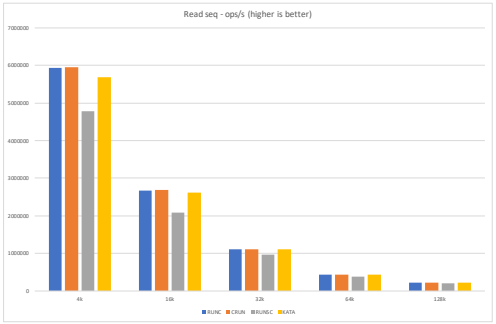

Figure 4.13. Memory sequential read

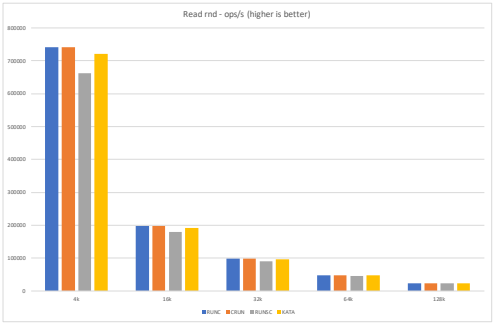

Figure 4.14. Memory random read

16KB.

4.5.3 Networking

Net I/O

Figure 4.17 shows the execution of a network test based on the `iperf3` tool. As it is possible to see, there is no difference in performance between classic solutions such as runc and crun. Regarding gVisor, as reported in 4.3.1, its implementation worsens networking performance.
A network stack totally in userspace has in fact a big impact on the performance but it guarantees an additional layer of isolation. The loopback interface is too defined in userspace and all networking aspects are managed by *Sentry*. The download and upload throughput reached by runsc (using netstack) is of 13.19Gbit/s and 13.74

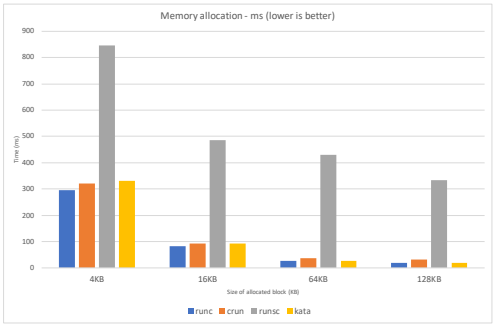

Figure 4.15. Allocation time

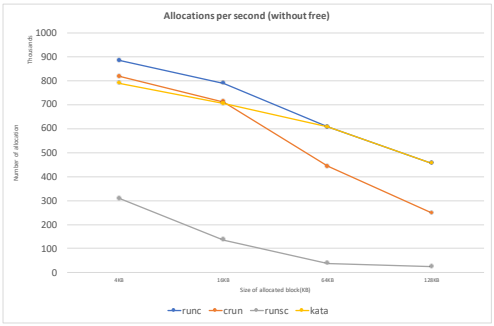

Figure 4.16. Allocations per second

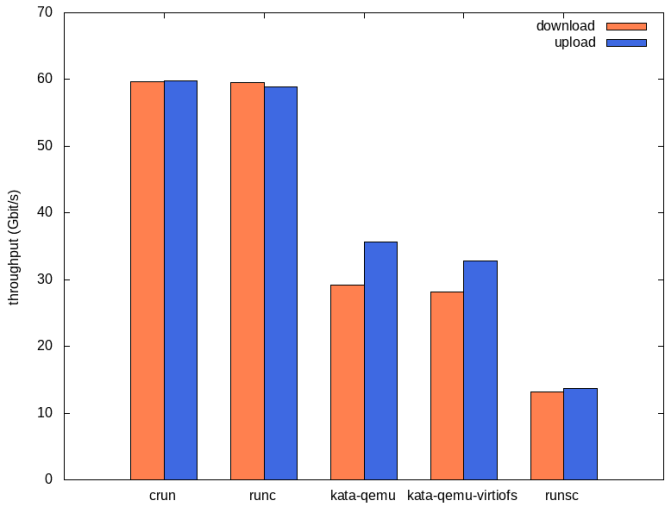

Figure 4.17. network iperf3-based benchmarking

Gbis/s respectively, which is only the 0,22x of the one obtained with runc. For high-intensive network applications, it is possible to cut off this feature and enable the *network passthrough* functionality that allows to use the host network stack but degrading the security of the environment.
Regarding hipervisor-based solutions like *kata-quemu* and *kata-quemu-virtiofs* they reach approximately the 50% of the throughput obtained by runc and crun that take advantage of the host network stack. This penalty is caused by the overhead introduced by the presence of the virtual machine.

Round Trip Time

As it is possible to see in the graph reported in figure 4.18, hybrid solutions introduces a significant overhead compared to classic container solutions. Differently from what obtained in the iperf3 test, runsc introduces a smaller overhead compared to kata-qemu and kata-qemu-virtiofs. In particular, the latency introduced is of: 0,3206ms for runsc, 0,3336ms for kata-qemu and 0,3517ms for kata-qemu-virtiofs compared to runc and crun.

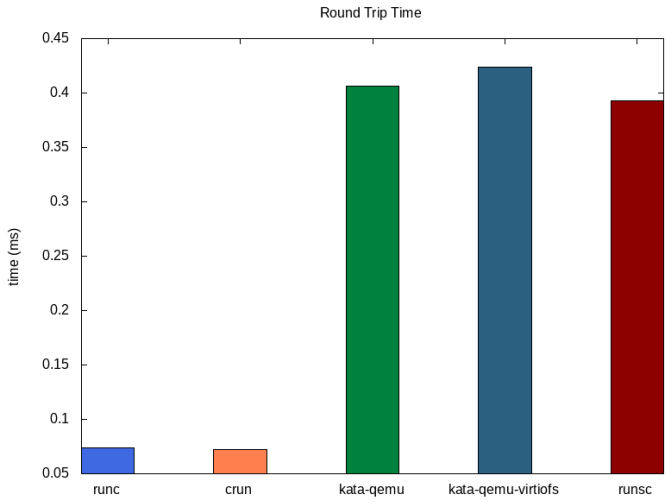

Figure 4.18. Round Trip Time

4.5.4 System calls execution time

Figure 4.19 4.20 and 4.21 show the result obtained by the system call execution test on three different system calls. For what concerns getpid and getcwd, runsc introduces the biggest overhead due to the system call interception exploited by the ptrace platform. This represents the most significant structural cost, moreover the implementation of syscalls entirely in user space is slower than the more mature native ones. How it is possible to see in 4.19 and 4.20, both the system calls are executed with similar behaviors. In particular runsc performs 25,4x and 22,25x times slower than runc for the execution of getpid and getcwd respectively. Instead, in the case of *fopen* 4.21, runsc is 5,51x times slower than classic solutions.

With regards to Kata-containers, it does not introduce any significant overhead in the execution of getpid and getcwd. On the other hand, for the executuon of fopen kata-qemu-virtiofs and kata-qemu result being 2,54x and 17,35x times slower than runc and crun. This is due to the fact that they use different methods to access

the shared filesystem (*9pfs* using the 9p network protocol or *virtio-fs* using *FUSE requests*).

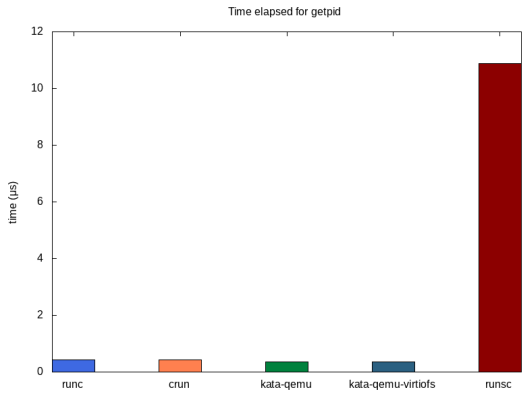

Figure 4.19. `getpid` execution time

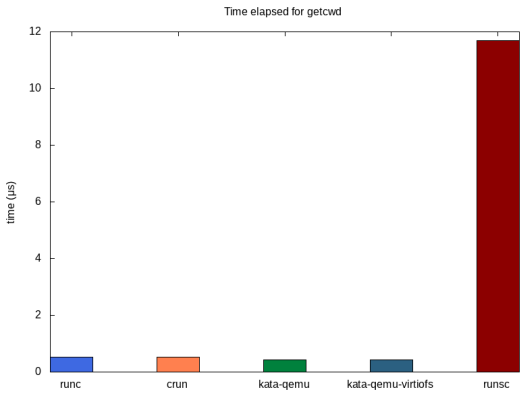

Figure 4.20. `getcwd` execution time

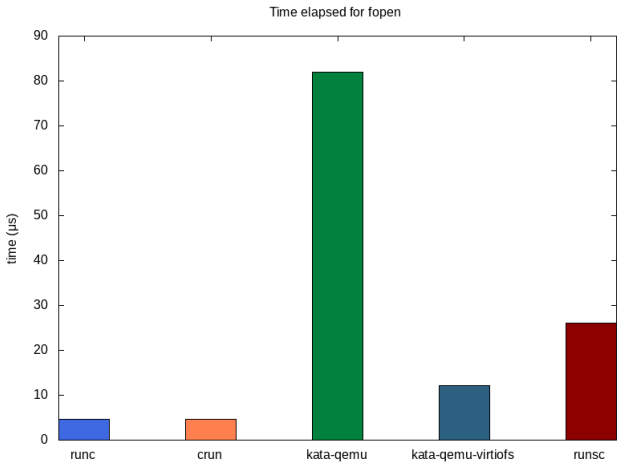

Figure 4.21. `fopen` [90] execution time

4.5.5 CPU

CPU prime

The result for the sysbench CPU benchmark are shown in figure 4.22. In this case we can see that both kata-qemu (in both its configurations) and runsc do not impose any significant overhead, in terms of events per second managed by the cpu, with respect to runc and crun.

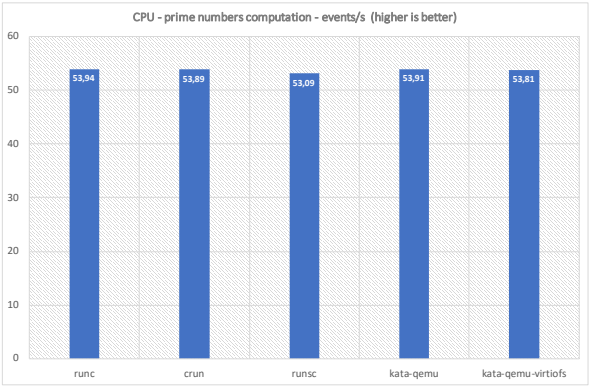

Figure 4.22. CPU - prime numbers computation

4.5.6 I/O

Figure 4.23 shows the result obtained by each runtime when performing sequential writes. As first impression it can be seen that there is a significant overhead introduced by secure runtimes in file I/O operations. With regards to runsc, it is evident that it tents to performs better with increasing block size. This can be justified by the implementation of the memory management system of Sentry (since we are using buffered I/O) that instead of having page based file, it uses a range based memory allocation approach. So issuing bigger requests results in fewer memory lookups. Instead the biggest cost is paid with smaller request, in this case runsc reaches a throughput of about 47MB/s whereas runc and crun score an average of 831MB/s. Starting from a block size of 32KB runsc outperfoms over kata-runtime, reaching a throughput of 745MB/s.
With regards to kata-runtime, for smaller block sizes it performs better than runsc introducing a smaller overhead. An interesting thing to note is that kata-qemu-virtiofs seems to suffer for block size of 4KB and 32KB, performing worse than kata-qemu. Talking with the Kata-containers community, it results that in some cases kata-qemu has better performance than kata-qemu-virtiofs, but this is still under investigation. Conversely, as expected, kata-qemu-virtiofs outperforms over kata-qemu in any other case. In any case, there is a singnificant overhead introduced by the use of an hypervisor based solutions. In particular kata-qemu reaches an average throughput of 200MB/s, which represents in the best case 1/4 of what runc achieves. On the other hand, kata-qemu-virtiofs got its best score with 16KB block size, reaching a throughput of 406MB/s, being however far from the result obtained by traditional runtimes.

Regarding sequential readings, figure 4.24 shows the result obtained by the FIO

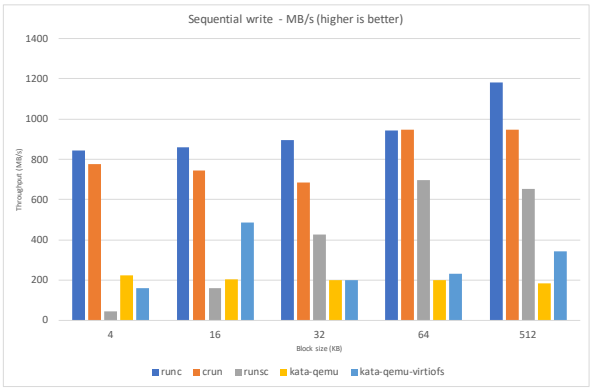

Figure 4.23. Sequential write throughput

test. These results are quite different from the ones obtained when performing writing operations since the overhead introduced by the more secure solutions (kata-runtime and runsc) is minimal. We think that in this case we are limited by the hardware that we had at our disposal. The most relevant thing that can be noticed is the overhead introduced by runsc when performing reading operation with block size less or equal 16KB.

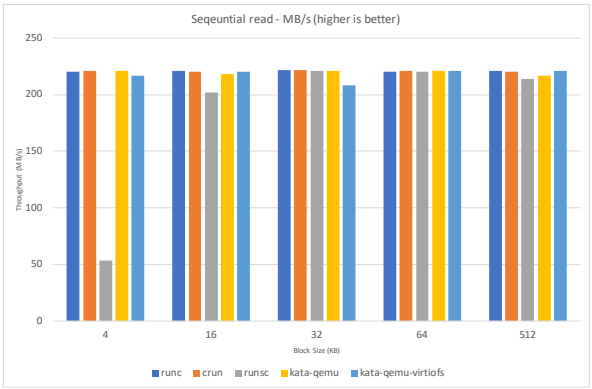

Figure 4.24. Sequential read throughput

4.5.7 Real application benchmarking

Machine learning model training

Figure 4.25 shows the result obtained by training a machine learning model. The most evident thing is that runsc imposes the most significant overhead being 1,5x slower compared to runc. Actually this does not depends on a CPU overhead, as when demonstrated by the result obtained with the sysbench benchmark. By profiling the execution of the python script we realize that this overhead is mostly due to the use of operations over sockets, so NET I/O operations. Instead, both kata-qemu and kata-qemu-virtiofs imposes a minimum penalty compared to runc, requiring 1.07x more time than runc. This is because the hardware assisted virtualization exploited by KVM allows to have performances almost equal to the native ones for this kind of workload.

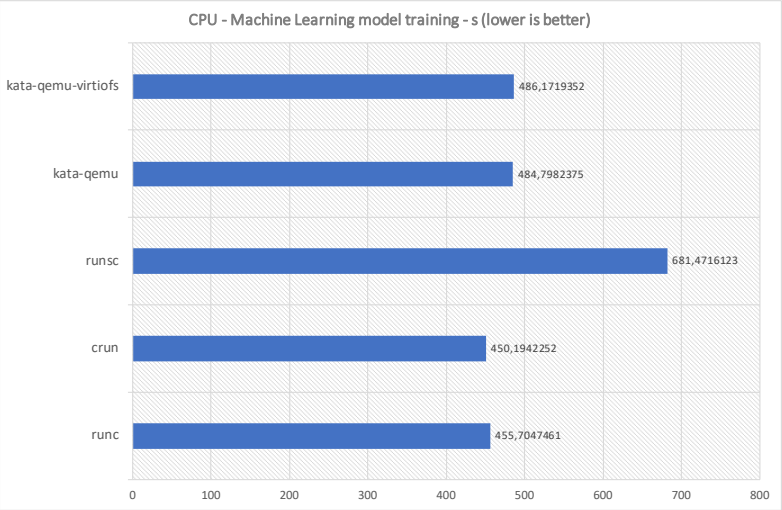

Figure 4.25. CPU - machine learning model training

To run this test, the default Seccomp security profile has been disabled via the `-security-opt seccomp=unconfined` option of the docker run command. Since the first runs of this test, with seccomp enabled, we have had too marked differences in terms of time, especially between runc and crun. This made us have doubts because, on the execution of the container, runc and crun should have the same results since they do not require any addiotional software layer (as for kata-runtime or runsc). By doing an analysis of the python code executed by the model training script, we found that the default Seccomp profile of runc slows down the work of the Python

interpreter to execute the associated byte code, leading to to the wrong results. This is actually an open issue in runc and it is due to missing support for a seccomp flag called `SECCOMP_FILTER_FLAG_SPEC_ALLOW`, which is instead implemented in the default seccomp configuration of crun [91].

Redis

As described in the Redis test section, we decided to test the performances of this in-memory key-value store database. Figure 4.26 shows the score obtained by each runtime. Classic container solutions like crun and runc have a throughput of 1894060 requests per second when using a single container instance to host the database. This result is no different from running a Redis server directly on the host operating system. There is a clear difference when using secure runtimes. GVisor is significantly slower than other solutions, in particular it reaches a maximum throughput speed of about 145182 operations per second, which represents only the 7,66% of the one obtained with runc and crun. This result is probably obtained by the implementation of the network stack entirely in user space, as previously noted. Regarding hypervisor-based solutions, the throughput achieved is about 912087 ops/s both for kata-qemu and kata-qemu-virtiofs, which represents the 48% of the score obtained by runc and crun.

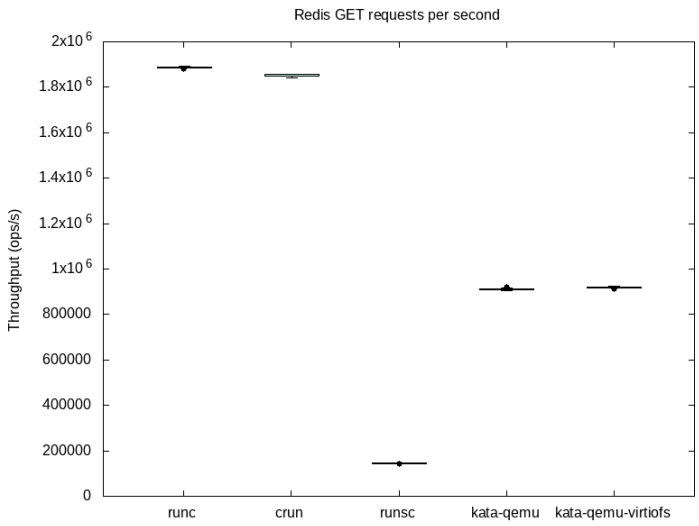

Figure 4.26. Redis benchmark throughput measured in requests per second

In the end, we can say that this test is definitely dependent on the performance of the network stack that the various solutions have.

Apache Spark

The default seccomp security profile was disabled as reported in the CPU M.L. test to avoid the performance impact on it given by the seccomp layer.
As it is possible to notice in the chart 4.27, runc and crun have similar performance, finishing the word count problem [94] job in 650 seconds. Instead, regarding the hybrid solutions, gVisor completes the work in 940 seconds, being 1,44x times slower than the average of runc and crun. The result obtained by hypervisor-based solutions is slightly different: kata-qemu using 9p ends the job in 760 second (1,15x slower than runc and crun), instead the kata-qemu-virtiofs version is able to finish the work in 729 seconds (1,11x of runc and crun).

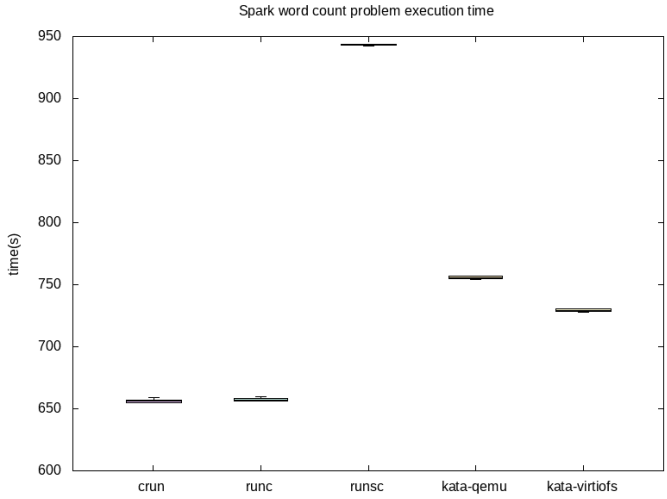

Figure 4.27. Spark word count problem execution time (lower is better)

Apache Spark single node system CPU usage

During the test we traced the cpu usage of the whole system and the cpu usage that is seen by the container. In particular we used the *dockerd* API to collect data for the container. To observe the cpu usage of the system, we looked at `/proc/stat`. In this case the job is executed by one core of our system. So the total amount of system CPU usage is about 25%. The chart 4.28 shows how the same spark job influences the system cpu usage when it runs in different containers. In the chart 4.29 is reported the CPU usage for each container from the point of view of the containers themselves. Each container consumes 100% of the CPU during the Job.

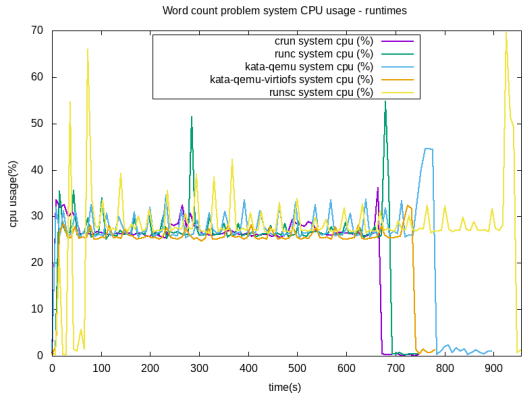

Figure 4.28. Spark word count problem system cpu usage

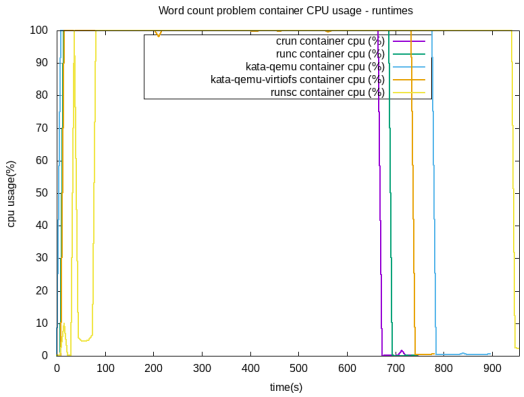

Figure 4.29. Spark word count problem container cpu usage

Apache Spark single node system memory usage

The image 4.30 shows the memory usage of each container during the execution of the word count problem using the single node configuration of Spark. The value of the memory usage is taken directly from the cgroup of the running container. The figure shows a progressive increase in memory usage for all technologies. The overhead introduced by the secure solutions is given by the cost of the processes to manage the sandbox.

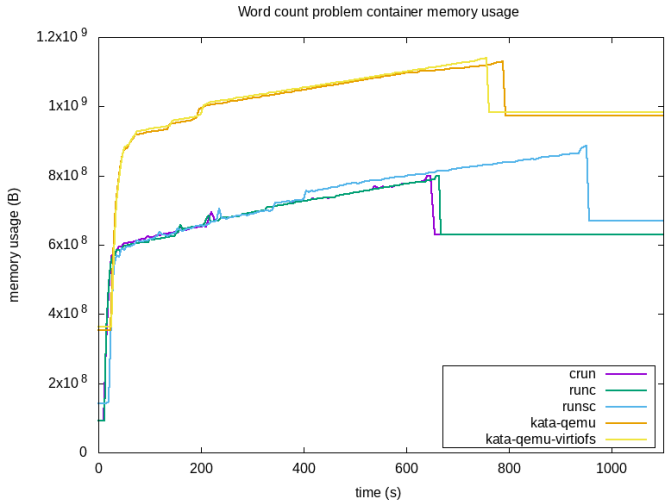

Figure 4.30. Spark word count problem container memory usage

Apache Spark standalone cluster analysis

Another test that we done in our experiments is based on the same *word count problem* [94] of the test reported in 4.5.7 but on a Apache Spark standalone cluster. In this case we deployed a cluster with two workers using PySpark as API. Each worker has `3GB` of memory and one single core of the CPU. The input file is the same that was used for the previous experiment. A JupyterLab interface was used to interact with the cluster. The architecture of the Apache Spark cluster used is exposed in 4.31.

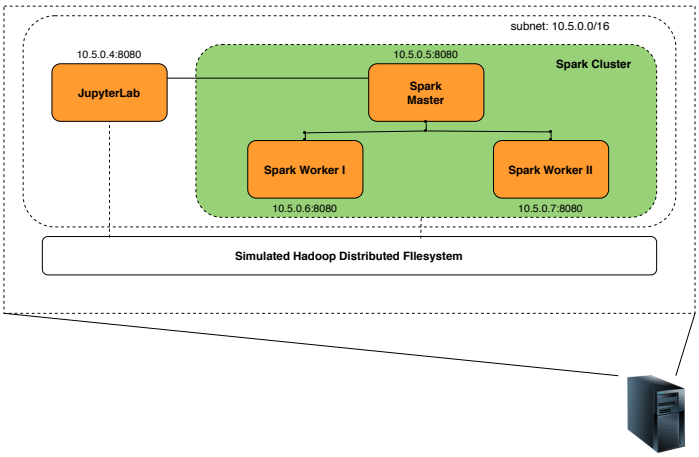

Figure 4.31. Apache Spark cluster architecture

As shown in figure 4.32, the execution time of the job has been drastically reduced. Crun and runc takes about 206 seconds to complete the Job, runsc 627s (3,05x slower) and regarding Kata-containers: 354s (1,73x slower) for 9p and 334 (1,61x slower) for the virtiofs version. Compared to the test with only one node we notice the presence of outliers in the graph, these depends on the instantaneous load on the machine during the execution process.

Tea Store

In the figure 4.33 are shown the results obtained by measuring the deploy time for each kind of services. Here for deploy time it is intended the total time required from the creation of the container to the availability of the service. As we can see the result reflects what we obtained during the boot time test. Kata-qemu introduces the the biggest overhead requiring a total of 149,2 s to deploy the entire application (2,1x slower than runc). This time overhead is mitigated by the use of kata-qemu-virtiofs which takes 110,3 s to start up all the services (1,55x than runc). GVisor imposes a minor overhead with respect to kata-qemu taking in total 136,7 s (almost twice as much as run c, 1,92x slower). An important thing to notice is the time

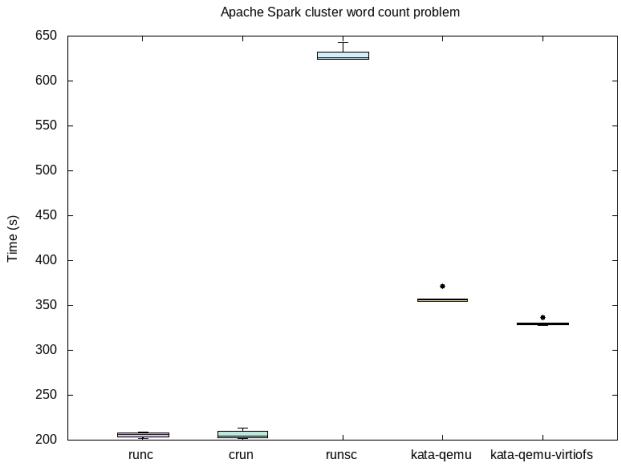

Figure 4.32. Apache Spark cluster word count problem execution time

taken by runsc(gVisor) to deploy the persistence service. We have in fact analyzed the resources consumed during the start up of this service and we find out that it is highly NET I/O bound workload, as shown in figure 4.34. As noted by testing networking in gVisor, starting the persistence service takes more time than others.

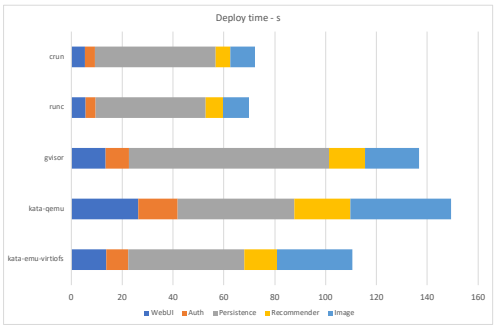

Figure 4.33. TeaStore services deploy time

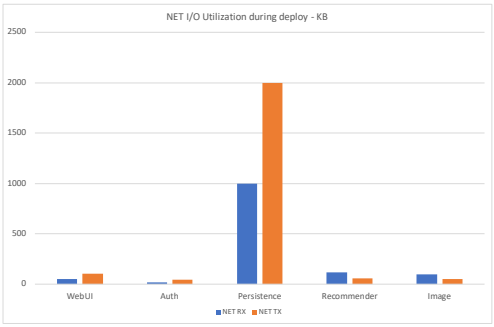

Figure 4.34. NET I/O during services deploy

Figure 4.35 shows the impact on system memory for the deployment of the TeaStore application. It is clear than kata-runtime has the highest memory requirement whereas runsc results being less expensive. But in general there is a significant overhead, in terms of memory, introduced by secure runtimes .

Figure 4.36 illustrates the result obtained by running the JMeter stress test. Here it is clearly noticeable how the use of additional software layers to improve the

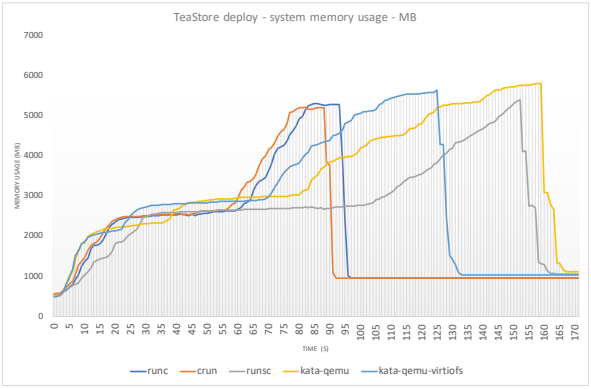

Figure 4.35. System memory consumption during services' deploy

security of containers (as in gVisor and Kata-containers) have a negative effect on the performance. GVisor is the slowest, handling all the 40000 requests performed by JMeter in 342 seconds having, reaching the management of 116 ops/s. Kata-qemu e Kata-qemu-virtiofs obtain similar results, serving all the requests in 295 and 291 seconds respectively and reaching an average throughput of 136.55 ops/s. As expected runc and crun outperforms with respects to gVisor and Kata-containers, managing all the request in about 159 s and consuming an average of 252.3 ops/s.

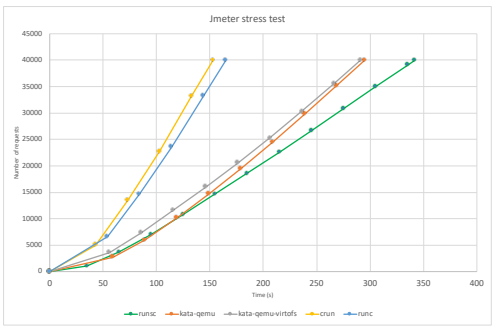

Figure 4.36. JMeter stress test

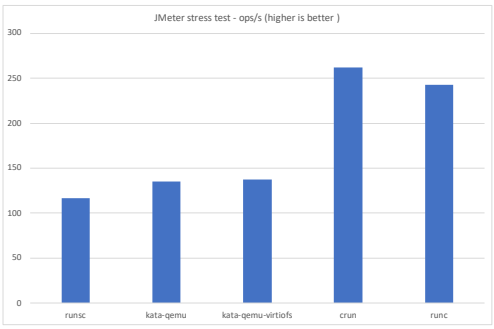

Figure 4.37. OPS/s handled during JMeter stress test

It also important to notice that with runsc we obtain an error rate of 0.01%, with 5 failed requests over a total of 40000. Instead kata-qemu and kata-qemu-virtiofs both got an error rate of 0.03% failing to serve 13 and 11 request respectively.

4.6 Overall results

Tables 4.4 - 4.5 shows the results obtained in each test by the different container runtimes. To better understand the overhead introduced by the use of secure runtimes, tables 4.6 - 4.7 illustrates for each test the cost of adopting these solutions with respect to the average result obtained by runc and crun. Note that for memory allocation and memory acess tests the results for kata-qemu-virtiofs are not reported since there are no differences in the memory management system between the two kata-runtime configurations.

Resource	Test	runc	crun	kata-qemu	kata-qemu-virtiofs	runsc	Unit
-	boot time	0,59970	0,54697	1,02660	1,02941	0,67126	s
CPU	prime	53,94	53,89	53,91	53,81	53,09	events/s
Network	iperf3 upload	58,890	59,760	35,609	32,859	13,744	Gbit/s
*	iperf3 downl.	59,599	59,659	29,125	28,123	13,194	Gbit/s
*	RTT	0,0741	0,0725	0,4061	0,4242	0,3931	ms
System calls	getpid	0,41929	0,43636	0,36381	0,36745	10,87000	µs
*	getcwd	0,52517	0,52514	0,42946	0,42459	11,69000	µs
*	fopen	47,26	47,10	818,98	120,08	260	µs
File I/O	4K write	831	832,25	204,75	166,75	46,825	MB/s
*	16K write	1216	657,5	195,25	406,25	163,5	MB/s
*	64K write	887	730,25	199,75	170,75	422,75	MB/s
*	256K write	937	831	199,25	261,25	683,75	MB/s
*	512K write	1045,25	883,25	193,25	317,75	745,25	MB/s
*	4K read	220	221	221	217	53,3	MB/s
*	16K read	221	220	218	220	202	MB/s
*	64K read	222	222	221	208	221	MB/s
*	256K read	220	221	221	221	220	MB/s
*	512K read	221	220	217	221	214	MB/s
ML	training	455,70	450,19	484,80	486,17	681,47	s
Redis	throughput	1890540	1855584	912087	919117	144258	ops/s
Spark single	Ex. time	656,071	652,532	755,870	729,442	944,057	s
Spark cluster	Ex. time	206,071	204,125	355,870	330,442	626,154	s
TeaStore	depoly	69,93	72,20	149,29	110,39	136,73	s
*	Jmeter	165	153	295	291	342	s

Table 4.4. Overall results

Resource	Test	runc	crun	kata-qemu	kata-qemu-virtiofs	runsc	Unit
Memory	footprint	51	49	94798	107665	8099	KB
*	4K allocation	296	320	332	-	847	ms
*	16K allocation	83	92	93	-	486	ms
*	64K allocation	27	37	27	-	430	ms
*	128K allocation	18	33	18	-	335	ms
*	4K seq write	3319981	3322365	3229130	-	2866565	ops/s
*	16K seq write	1111194	1111279	1100772	-	988153	ops/s
*	32K seq write	580022	579933	577276	-	520303	ops/s
*	64K seq write	301396	301468	300804	-	270466	ops/s
*	128K seq write	150133	153393	153302	-	138859	ops/s
*	4K rnd write	649574	648375	670354	-	241261	ops/s
*	16K rnd write	171279	169721	178891	-	62072	ops/s
*	32K rnd write	87064	86906	90063	-	31428	ops/s
*	64K rnd write	39891	40961	40912	-	15890	ops/s
*	128K rnd write	14980	15100	15315	-	7868	ops/s
*	4K seq read	5927936	5957716	5682787	-	4777526	ops/s
*	16K seq read	2673321	2682456	2624538	-	2093980	ops/s
*	32K seq read	1115403	1115890	1102448	-	960440	ops/s
*	64K seq read	439455	439688	438916	-	381200	ops/s
*	128K seq read	224790	226019	225280	-	196545	ops/s
*	4K rnd read	741754	741308	720753	-	661997	ops/s
*	16K rnd read	198124	198267	192556	-	179934	ops/s
*	32K rnd read	98080	97971	97203	-	90999	ops/s
*	64K rnd read	48250	48316	48249	-	45270	ops/s
*	128K rnd read	24101	24178	24297	-	22870	ops/s

Table 4.5. Overall results

Test	avg runc/crun	Kata-9p	Kata-virtiofs	runsc
Boot time	0,57333 s	1,8x	1,8x	1,2x
CPU prime	53,915 ops/s	1x	1x	1x
Network upload	59,325 Gbit/s	0,49x	0,47x	0,22x
Network downl.	59,239 Gbit/s	0,48x	0,47x	0,22x
RTT	0,0733 ms	5,54x	5,78x	5,78x
Syscalls getpid	0,4278285 µs	0,85x	0,85x	25,40x
Syscalls getcwd	0,5251605 µs	0,81x	0,80x	22,25x
Syscalls fopen	47,18 µs	17,35x	2,54x	5,51x
I/O 4K write	831,63 MB/s	0,25x	0,20x	0,06x
I/O 16K write	936,75 MB/s	0,21x	0,43x	0,17
I/O 32K write	808,623 MB/s	0,25x	0,21x	0,52
I/O 64K write	884 MB/s	0,23x	0,30x	0,77
I/O 512K write	964,25 MB/s	0,20x	0,33x	0,77
I/O 4K read	220,5 MB/s	1x	0,98x	0,24
I/O 16K read	220,5 MB/s	0,99x	1x	0,92
I/O 32K read	222 MB/s	1x	0,94x	1x
I/O 64K read	220,5 MB/s	1x	1x	1,00x
I/O 512K read	220,5 MB/s	0,98x	1x	0,97x
ML	452,95	1,07x	1,07x	1,50x
Redis	1873062 ops/s	0,48x	0,49x	0,08x
Spark single	654,3015 s	1,15x	1,11x	1,44x
Spark cluster	205,098 s	1,73x	1,61x	3,05x
TeaStore deploy	71,065 s	2,1x	1,55x	1,92x
TeaStore Jmeter	159 s	1,85x	1,83x	2,15x

Table 4.6. Performance penalties of secure container runtimes

Test	avg runc/crun	Kata-9p	Kata-virtiofs	runsc
Memory footprint	49,9705 KB	1897,08x	2155,57x	162,08x
Memory alloc 4K	308 ms	1,08x	-	2,75x
Memory alloc 16K	87,5 ms	1,06x	-	5,55x
Memory alloc 64K	32 ms	0,84x	-	13,44x
Memory alloc 128K	25,5 ms	0,71x	-	13,14x
Memory 4K seq write	3321173 ops/s	1x	-	0,86x
Memory 16K seq write	1111236,5 ops/s	1x	-	0,89x
Memory 32K seq write	579977,5 ops/s	1x	-	0,90x
Memory 64K seq write	301432 ops/s	1x	-	0,90x
Memory 128K seq write	151763 ops/s	1x	-	0,91x
Memory 4K rnd write	648974,5 ops/s	1,03x	-	0,37x
Memory 16K rnd write	170500 ops/s	1,05x	-	0,36x
Memory 32K rnd write	86985 ops/s	1,04x	-	0,36x
Memory 64K rnd write	40426 ops/s	1,01x	-	0,39x
Memory 128K rnd write	15040 ops/s	1,02x	-	0,52x
Memory 4K seq read	5942826 ops/s	0,96x	-	0,80x
Memory 16K seq read	2677888,5 ops/s	0,98x	-	0,78x
Memory 32K seq read	1115646,5 ops/s	0,99x	-	0,86x
Memory 64K seq read	439571,5 ops/s	1x	-	0,87x
Memory 128K seq read	225404,5 ops/s	1x	-	0,87x
Memory 4K rnd read	741531 ops/s	0,97x	-	0,89x
Memory 16K rnd read	198195,5 ops/s	0,97x	-	0,91x
Memory 32K rnd read	98025,5 ops/s	0,99x	-	0,93x
Memory 64K rnd read	48283 ops/s	1x	-	0,94x
Memory 128K rnd read	24139,5 ops/s	1,01x	-	0,95x

Table 4.7. Performance penalties of secure container runtimes

Chapter 5

Conlusions

The purpose of this thesis was to study the implementation of OS-level virtualization through the Linux kernel and how, from recent developments, it has integrated different aspects of traditional virtualization based on Virtual Machine to enforce the security of software containers. In the first chapter of our thesis the concept of virtualization and the different techniques that allow its implementation are introduced. This served us to explain the concept of virtualization at the operating system level and how it relates with traditional virtualization approaches. Starting from chapter 2 our work is mainly divided in two parts. In the first part (chapter 2 and 3) we analyzed the mechanisms offered by the Linux kernel that ensure the isolation of a process, resulting in the creation of a container. This was consolidated by the development of a lightweight process isolation tool, that leveraging namespaces and cgroups and enforcing security through Linux capabilities and Seccomp, allows the creation of an isolated environment for the execution of a process.
This helped us to understand which steps are necessary for the creation of a container, in order to have a clear idea of the role of a container runtime within the architecture of a container manager like Docker.
The second part of our work (chapter 4) is focused on the most recently developed OCI-compliant container runtimes that has integrated different mechanisms of traditional virtualization to provide additional layers of security for the execution of a container. In particular we characterized and quantified the overhead introduced by these new secure container runtimes with respect to traditional solutions. To do this, several tests and benchmarks were performed to analyze the consumption of system resources depending on the type of sandbox used. Our work takes into consideration four different container runtimes than can be divided in two main groups:

- Traditional runtimes
- Secure runtimes

As traditional runtimes we have chosen runc and crun, being the most popular and used in the containers world. With regards to secure container runtimes, runsc from

the gVisor project and kata-runtime from Kata-containers were the best candidates being completely integrated with Docker and given their growth in recent years through constant development and improvements. The tests that we did take into consideration different system resources (memory, CPU, I/O, networking) and some real application use cases. Our results show that no significant CPU overhead is introduced when using secure runtimes. This has a positive feedback for all those applications characterized by intensive CPU-bound workload such as machine learning or data processing.
The biggest cost can be found in the memory requirement for a single container instance (what we called the memory footprint). In this case the entire container sandbox used by Kata-containers introduced the biggest overhead whereas using runsc from gVisor the cost is considerably reduced. This has significant consequences in all those scalability scenarios where a high number of container instances are required. Conversely, the penalty introduced in raw memory accesses is relatively high for runsc while kata-runtime introduced a little overhead. Regarding file I/O, both runsc and kata-runtime have a significant overhead being far from the performance obtained by traditional solution. This affects considerably all those applications that need to maintain a state (e.g databases) performing an high number of reading and writing operations trough the use of persistent storage. These costs are also relevant in applications such as web server, which are strictly bound to filesystem and networking operations. In fact a relevant overhead is also introduced when perfoming NET I/O operations. In this case kata-runtime outperforms runsc, but they are both far from runc and crun. Furthermore runsc introduces an additional cost in the management of system calls. The configuration of runsc we have chosen, using ptrace as a platform, has the highest overhead compared to other runtimes.

In conclusion, our results show that the adoption of secure container runtimes has a significant cost with respect to traditional solutions. They definitely ensure better security (this is outside the scope of our thesis) by introducing additional layer of indirection, but all of this comes at a cost.

Bibliography

[1] Oracle, "Reasons to Use Virtualization"

[2] Gerald J. Popek and Robert P. Goldberg, “Formal requirements for virtualizable third generation architectures“, *doi:10.1145/361011.361073*

[3] Radhwan Y. Ameen, Asmaa Y. Hamo, 2013, "Survey of server virtualization"

[4] Amit Singh, 2004, "An Introduction to Virtualization" http://webx.ubi.pt/~hgil/utils/Virtualization_Introduction.html,

[5] Joshua S. White, Adam W. Pilbeam, 2010 “A Survey of Virtualization Technologies With Performance Testing“

[6] Harvard lectures, 2017, "Virtualization" https://www.eecs.harvard.edu/~cs161/notes/virtualization.pdf

[7] Padhy, Rabi & Patra, Manas & Satapathy, Suresh. (2020). "virtualization techniques & technologies: state-of-the-art"

[8] Fatma Bazargan, Chan Yeob Yeun, Mohamed Jamal Zemerly, 2012, "State-of-the-Art of Virtualization, its Security Threats and Deployment Models"

[9] Matthew Chapman, Daniel J. Magenheimer, Parthasarathy Ranganathan, 2007, "MagiXen: Combining Binary Translation and Virtualization"

[10] Keith Adams, Ole Agesen, "A Comparison of Software and Hardware Techniques for x86 Virtualization"

[11] Valeria Cardellini, corso di sistemi distribuiti e cloud computing, 2019/2020, "Virtualizazzione"

[12] Qian Lin, Zhengwei Qi, Jiewei Wu, Yaozu Dong, Haibing Guan, 2011, "Optimizing virtual machines using hybrid virtualization"

[13] Oracle, November 6 2013, "Nested Virtualization With Binary Translation: Back to the Future" https://blogs.oracle.com/ravello/nested-virtualization-with-binary-translation

[14] Clear Interrupt Flag, http://www.jaist.ac.jp/iscenter-new/mpc/altix/altixdata/opt/intel/vtune/doc/users_guide/mergedProjects/analyzer_ec/mergedProjects/reference_olh/mergedProjects/instructions/instruct32_hh/vc32.htm

[15] 10.4. Details About Hardware Virtualization https://www.virtualbox.org/manual/ch10.html#hwvirt-details

[16] Glenn Willen, Mike Cui, 15-410 Fall 2006 "Virtualization"
http://www.cs.cmu.edu/~410-f06/lectures/L31_Virtualization.pdf
[17] Johan De Gelas, March 17 2008
https://www.anandtech.com/show/2480/5
[18] "Hardware-Assisted Virtualization Explained"
Hardware-AssistedVirtualizationExplained
[19] "Paravirtualization Explained"
https://networksandservers.blogspot.com/2011/11/para-is-english-affix-of-greek-origin.html
[20] "Paravirtualization"
https://en.wikipedia.org/wiki/Paravirtualization
[21] Daniel Prizmant, December 12 2019, "What I Learned from Reverse Engineering Windows Containers"
https://unit42.paloaltonetworks.com/what-i-learned-from-reverse-engineering-windows-containers/
[22] Oracle zones
https://docs.oracle.com/cd/E23824_01/html/821-1460/zones.intro-5.html
[23] FreeBSD
https://www.freebsd.org/it/
[24] Namespaces
https://man7.org/linux/man-pages/man7/namespaces.7.html
[25] PID namespace
https://man7.org/linux/man-pages/man7/pid_namespaces.7.html
[26] "Linux Kernel Namespace Implementation: Introduction to namespace API"
https://titanwolf.org/Network/Articles/Article?AID=740fc46c-d325-4c22-a720-b5c259551c87#gsc.tab=0
[27] Nsproxy reference count
https://github.com/torvalds/linux/blob/master/include/linux/nsproxy.h
[28] Process Namespace, Mahmud Ridwan
https://www.toptal.com/linux/separation-anxiety-isolating-your-system-with-linux-namespaces
[29] Linux kernel
https://github.com/torvalds/linux
[30] Clone Linux man
https://man7.org/linux/man-pages/man2/clone.2.html
[31] Eli Bendersky, August 01 2018, "Launching Linux threads and processes with clone"
https://eli.thegreenplace.net/2018/launching-linux-threads-and-processes-with-clone/
[32] Pavel Emelyanov and Kir Kolyshkin, November 19 2007, "PID namespaces in

the 2.6.24 kernel"
https://lwn.net/Articles/259217/
[33] UTS namespace
https://man7.org/linux/man-pages/man7/uts_namespaces.7.html
[34] Emiliano Casalicchio, Stefano Iannucci,19 January 2020,"The State-of-the-Art in Container Technologies: Application, Orchestration and Security"
[35] Jun Nakajima and Asit K. Mallick,2007,"Hybrid-Virtualization—Enhanced Virtualization for Linux"
[36] Project Calico
https://www.projectcalico.org/
[37] Polycube Network
https://polycube.network/
[38] Mount namespace
https://man7.org/linux/man-pages/man7/mount_namespaces.7.html
[39] IPC namespace
https://man7.org/linux/man-pages/man7/ipc_namespaces.7.html
[40] Time namespace
https://man7.org/linux/man-pages/man7/time_namespaces.7.html
[41] Paul Menage, “Cgroups“, Linux Kernel documentation
https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt
[42] Tejun Heo, “Control Group v2“, Linux Kernel documentation
https://www.kernel.org/doc/Documentation/cgroup-v2.txt
[43] Milan Navrátil, Peter Ondrejka, Eva Majoršinová, Martin Prpič, Rüdiger Landmann, Douglas Silas, “Resource management guide“, Red Hat Customer portal
https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/
[44] Michael Barcarella, 2002, “Taking advantage of Linux capabilities“, Linux Journal
[45] Adrian Mouat, 2019, “Linux Capabilities: Why They Exist and How They Work“
[46] Davide Antonino Giorgio, Gabriele Minì, “nscage“
https://github.com/DavideAG/Understanding-containers
[47] *crun* github repository
https://github.com/containers/crun
[48] Netlink Library
http://www.infradead.org/~tgr/libnl/doc/core.html
[49] RFC3549, Netlink RFC
https://tools.ietf.org/html/rfc3549#section-2.3.2
[50] LXC Security, “Linux containers“
https://linuxcontainers.org/lxc/security/

[51] User namespace, Linux manual
https://man7.org/linux/man-pages/man7/user_namespaces.7.html
[52] Michael Kerrisk, 2019, "Understanding user namespaces"
[53] Rik Janssen, University of Amsterdam, February 11, 2018, Categorizing container escape methodologies in multi-tenant environments
https://work.delaat.net/rp/2017-2018/p80/report.pdf
[54] runc, Container Specification - v1
https://github.com/opencontainers/runc/blob/master/libcontainer/SPEC.md
[55] Lizzie Dixon, privilege escalation without `CAP_SETID` and user namespace
https://blog.lizzie.io/linux-containers-in-500-loc.html#fn.16
[56] Seccomp bpf, Linux kernel documentation
https://www.kernel.org/doc/html/v4.16/userspace-api/seccomp_filter.html
[57] Sebastian Krahmer, email and description of docker VMM breakout
https://www.openwall.com/lists/oss-security/2014/06/18/4
[58] Docker's seccomp documentation
https://github.com/docker/docker.github.io/blob/master/engine/security/seccomp.md
[59] Moby, Docker's seccomp default profile
https://github.com/moby/moby/blob/b248de7e332b6e67b08a8981f68060e6ae629ccf/profiles/seccomp/default.json#L52
[60] Lizzie Dixon, Docker System Calls
https://blog.lizzie.io/linux-containers-in-500-loc.html#org8504d16
[61] Inge Alexander Raknes, Bjørn Fjukstad, Lars Ailo Bongo, "nsroot: Minimalist Process Isolation Tool Implemented With Linux Namespaces"
[62] "A light-weight process isolation tool, making use of Linux namespaces and seccomp-bpf syscall filters"
https://nsjail.dev/#more-info
[63] CharlieCloud
https://github.com/hpc/charliecloud
[64] Firejail
https://firejail.wordpress.com
[65] Chroot jail escape example
https://filippo.io/escaping-a-chroot-jail-slash-1/
[66] Kata Containers isolation and components scheme
https://katacontainers.io/learn/
[67] Rick Donato, January 22 2020, "What Is the Difference between QEMU and KVM?"
https://www.packetflow.co.uk/what-is-the-difference-between-qemu-and-kvm/
[68] gVisor architecture scheme

https://gvisor.dev/blog/2019/11/18/gvisor-security-basics-part-1/)
[69] gVisor architecture second scheme https://www.usenix.org/system/files/hotcloud19-paper-young.pdf
[70] Kata Containers
https://katacontainers.io
[71] "The speed of containers, the security of VMs"
https://katacontainers.io/collateral/kata-containers-1pager.pdf
[72] Kata Containers Architecture
https://github.com/kata-containers/documentation/blob/master/design/architecture.md
[73] Kata Containers, story of a container runtime
https://www.openstack.org/videos/summits/denver-2019/kata-containers-story-of-a-container-runtime-1
[74] gVisor
https://gvisor.dev
[75] What is gVisor?
https://gvisor.dev/docs/
[76] gVisor costs
https://gvisor.dev/docs/architecture_guide/performance/
[77] J'oakim von Kistowski and Simon Eismann and Norbert Schmitt and Andr'e Bauer and Johannes Grohmann and Samuel Kounev, September 2018, TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research
[78] Apache JMeter™
https://github.com/apache/jmeter
[79] gVisor Platform
https://gvisor.dev/docs/architecture_guide/platforms/
[80] 9P protocol, Wikipedia
https://en.wikipedia.org/wiki/9P_(protocol)
[81] gVisor ptrace platform source code
https://github.com/google/gvisor/blob/master/pkg/sentry/kernel/ptrace.go
[82] Humble Devassy Chirammal, Prasad Mukhedkar, and Anil Vettathu. Mastering KVM virtualization. Packt Publishing Ltd, 2016.
[83] AMD64 syscalls supproted by gVisor, gVisor official documentation
https://gvisor.dev/docs/user_guide/compatibility/linux/amd64/
[84] Allowed syscalls from Sentry to Host
https://github.com/google/gvisor/blob/master/runsc/boot/filter/config.go#L28
[85] Allowed syscalls from Gofer to Host
https://github.com/google/gvisor/blob/master/runsc/fsgofer/filter/config.go#L27

[86] RAC RFC, IETF
https://tools.ietf.org/id/draft-ietf-tcpm-rack-03.html
[87] BBR RFC, IETF
https://tools.ietf.org/html/draft-cardwell-iccrg-bbr-congestion-control-00
[88] getpid, Linux manual
https://man7.org/linux/man-pages/man2/getpid.2.html
[89] getpid, Linux manual
https://man7.org/linux/man-pages/man2/getcwd.2.html
[90] fopen, Linux manual
https://man7.org/linux/man-pages/man3/fopen.3.html
[91] support seccomp flags such as SECCOMP_FILTER_FLAG_SPEC_ALLOW,
https://github.com/opencontainers/runc/issues/2430
[92] Memory Resource Controller
https://www.kernel.org/doc/Documentation/cgroup-v1/memory.txt
[93] Katacontainers, runtime, issue 3071
https://github.com/kata-containers/runtime/issues/3071
[94] Apache Spark word count problem, GitHub
https://github.com/apache/spark/blob/master/examples/src/main/python/wordcount.py
[95] Official jupyter image, Docker Hub
https://hub.docker.com/r/jupyter/pyspark-notebook
[96] Son-Hai Ha, Daniele Venzano, Patrick Brown, Pietro Michiardi, On the Impact of Virtualization on the I/O Performance of Analytic Workloads
[97] Kata-containers, Issue 512, Is it valid to set direct=0 when using libaio?
https://github.com/axboe/fio/issues/512
[98] Kata-containers, Issue 560, Testing Disk IO performance
https://github.com/kata-containers/tests/issues/560
[99] GVisor, Issue 142, sentry io performance
https://github.com/google/gvisor/issues/142
[100] Gvisor resource model
https://gvisor.dev/docs/architecture_guide/resources/
[101] FIO - Flexible I/O tester
https://fio.readthedocs.io/en/latest/fio_doc.html
[102] Linux/amd64 syscalls compatibility
https://gvisor.dev/docs/user_guide/compatibility/linux/amd64/
[103] SandboxCgroupOnly
https://github.com/kata-containers/documentation/blob/master/design/host-cgroups.md

Appendix A

How process IDs are allocated

As wirtten in PID Namespace 2.1.3, in Linux , a process may have more than one pid associated with it.
A `pid structure` is allocated by `alloc_pid` presented in `/kernel/pid.c`. In the code reported below A.1 - A.2 it is possible notice that the `alloc_pid` is composed by two main functions:

- `kmem_cache_alloc` is used to keeps a cache of pre-allocated structures. Since the allocation of a `pid structure` is performed frequently, a cache is used instead of allocating a `pid struct` from the main memory using `kmalloc`. In this way is possible to use a preallocated memory optimizing allocation time.
- `idr_alloc` and `idr_alloc_cyclic` are used to retrive the id value and save it in `upid.nr`. This value will be stored in one of the `upid` structures presents in `pid->numbers` related to the corresponding `pid namespace`. The `set_tid_size` value, that represents the size of the `set_tid` array, is used to tells to `alloc_pid` which PID to set for a process.
  `idr_alloc` and `idr_alloc_cyclic` allocates an unused ID in the range specified by the third parameter and fourth parameter.

After that the pid is stored inside the correct `upid` structure `pid->numbers`. Starting from the pid namespace where the process resides, a pid is assigned going up the namespaces hierarchy through to the root PID namespace.

```
1  struct pid *alloc_pid(struct pid_namespace *ns, pid_t *set_tid,
2          size_t set_tid_size)
3  {
4    struct pid *pid;
5    int i, nr;
6    struct pid_namespace *tmp;
7    struct upid *upid;
8
9    [...]
10   pid = kmem_cache_alloc(ns->pid_cachep, GFP_KERNEL);
11   [...]
12   tmp = ns;
13   pid->level = ns->level;
14
15   for (i = ns->level; i >= 0; i--) {
16     int tid = 0;
17     if (set_tid_size) {     /* getting the deepest tid */
18       tid = set_tid[ns->level - i];
19         [...]
20       set_tid_size--;
21     }
22     [...]
23     if (tid) {
24         /* used to allocate an id that is the pid value */
25       nr = idr_alloc(&tmp->idr, NULL, tid,
26                 tid + 1, GFP_ATOMIC);
27       [...]
28     } else {
29       int pid_min = 1;
30       /* used to allocate ciclically an id, 1 in this case */
31       nr = idr_alloc_cyclic(&tmp->idr, NULL, pid_min,
32                  pid_max, GFP_ATOMIC);
33     }
34         [...]
35     pid->numbers[i].nr = nr;
36     pid->numbers[i].ns = tmp;
37     tmp = tmp->parent;
38   }
39     [...]
40 }
```

Listing A.1. `pid` allocation in `alloc_pid` - /kernel/pid.c

```
1  struct pid *alloc_pid(struct pid_namespace *ns, pid_t *set_tid,
2          size_t set_tid_size)
3  {
4      [...]
5    refcount_set(&pid->count, 1);
6      [...]
7    upid = pid->numbers + ns->level;
8      [...]
9    for ( ; upid >= pid->numbers; --upid) {
10     /* Make the PID visible to find_pid_ns. */
11     idr_replace(&upid->ns->idr, pid, upid->nr);
12     upid->ns->pid_allocated++;
13   }
14   [...]
15   return pid;
16 }
```

Listing A.2. used to give visibility to the `pid` in `alloc_pid` - /kernel/pid.c

Before returning the pid structure, the final step is to make the pid visible to `find_pid_ns`. `find_pid_ns` is a function that can be used to finds the pid in the namespace specified using the corresponding `struct upid`.

Appendix B

Nscage

B.1 mount_too_revealing source code

```
1  static bool mount_too_revealing(const struct super_block *sb,
       int *new_mnt_flags)
2  {
3    const unsigned long required_iflags = SB_I_NOEXEC|SB_I_NODEV;
4    struct mnt_namespace *ns = current->nsproxy->mnt_ns;
5    unsigned long s_iflags;
6
7    if (ns->user_ns == &init_user_ns)
8      return false;
9
10   /* Can this filesystem be too revealing? */
11   s_iflags = sb->s_iflags;
12   if (!(s_iflags & SB_I_USERNS_VISIBLE))
13     return false;
14
15   if ((s_iflags & required_iflags) != required_iflags) {
16     WARN_ONCE(1, "Expected s_iflags to contain 0x%lx\n",
17         required_iflags);
18     return true;
19   }
20
21   return !mnt_already_visible(ns, sb, new_mnt_flags);
22 }
```

Listing B.1. mount_too_revealing() - /fs/namespace.c

B.2 List of dropped capabilities

Capability
CAP_NET_BROADCAST
CAP_SYS_MODULE
CAP_SYS_RAWIO
CAP_SYS_PACCT
CAP_SYS_ADMIN
CAP_SYS_NICE
CAP_SYS_RESOURCE
CAP_SYS_TIME
CAP_SYS_TTY_CONFIG
CAP_AUDIT_CONTROL
CAP_MAC_OVERRIDE
CAP_MAC_ADMIN
CAP_NET_ADMIN
CAP_SYSLOG
CAP_DAC_READ_SEARCH
CAP_LINUX_IMMUTABLE
CAP_IPC_LOCK
CAP_IPC_OWNER
CAP_SYS_PTRACE
CAP_SYS_BOOT
CAP_LEASE
CAP_WAKE_ALARM
CAP_BLOCK_SUSPEND
CAP_MKNOD
CAP_AUDIT_READ
CAP_AUDIT_WRITE
CAP_FSETID
CAP_SETFCAP

Table B.1. List of dropped capabilities in nscage