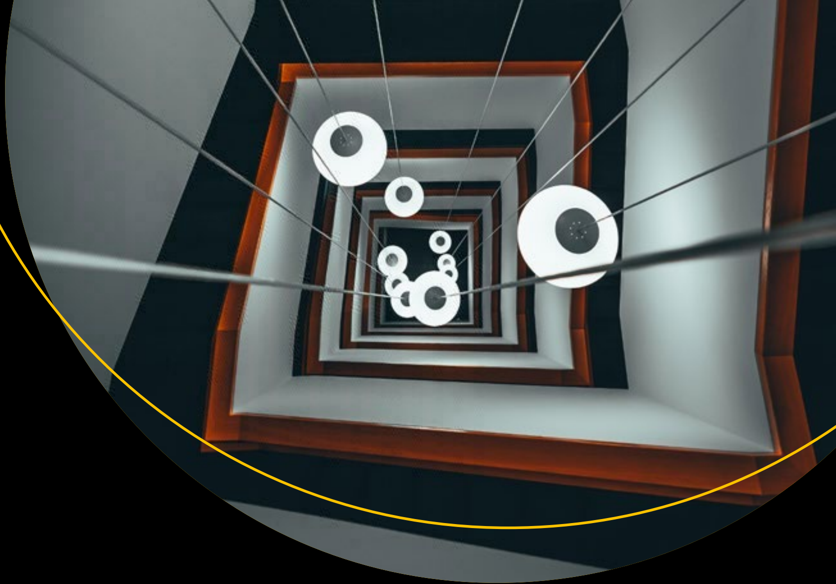


Linux Containers and Virtualization

A Kernel Perspective

Shashank Mohan Jain

Apress®

Linux Containers and Virtualization

A Kernel Perspective

Shashank Mohan Jain

Apress®

Linux Containers and Virtualization: A Kernel Perspective

Shashank Mohan Jain
Bengaluru, India

ISBN-13 (pbk): 978-1-4842-6282-5 ISBN-13 (electronic): 978-1-4842-6283-2
https://doi.org/10.1007/978-1-4842-6283-2

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Spandana Chatterjee
Development Editor: Matthew Moodie
Coordinating Editor: Shrikant Vishwakarma

Cover designed by eStudioCalamar

Cover image designed by Pexels

Distributed to the book trade worldwide by Springer Science+Business Media LLC, 1 New York Plaza, Suite 4600, New York, NY 10004. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/978-1-4842-6282-5. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

To my parents, my wife, and daughter, for being patient with me during the making of this book and always making that positive difference.

Table of Contents

About the Author

Shashank Mohan Jain has been working in the IT industry for nearly 20 years, mainly in the areas of cloud computing and distributed systems. He has keen interests in virtualization techniques, security, and complex, dynamic systems. Shashank has 25 software patents (many yet to be published) to his name in the area of cloud computing, IoT, and machine learning. He is a speaker at multiple reputed cloud conferences. Shashank also holds Sun, Microsoft, and Linux kernel certifications.

About the Technical Reviewer

Suresh Venkatasubramaniyan has a Ph.D. from the Indian Institute of Science in image forensics, compression, and encryption. He has close to 20 years of experience in machine learning and data mining. His areas of interests include natural language understanding, complex networks, and computational cognition. At present, he is a Principal Data Scientist with Walmart Labs. Previously, he was an R&D expert with SAP Labs and Accenture AI.

Introduction

The motivation for this book goes back to the words of Nobel Laureate and famous scientist Richard Feynman, “What I cannot create, I do not understand.”

The idea of the book was to develop a deep understanding of the world of virtualization and, in particular, go down the rabbit hole as far as Linux containers are concerned. Readers will get an understanding of what happens at the Linux operating system level when we talk of virtualization and Linux containers. The book explores the data structures involved in creating the isolation provided by Linux containers as well as the various resource control mechanisms.

The book will be helpful for people working in the area of cloud computing. Whether its development or DevOps, the book can take readers through the journey of what is really happening under the hood. It doesn’t cover the API level, but covers what happens below the APIs when we use Linux containers. By reading this book and going over the exercises, readers will get a decent understanding of how the world of containers works and will be able to better optimize and troubleshoot their deployments.

CHAPTER 1

Virtualization Basics

This book explains the basics of virtualization and will help you create your own container frameworks like Docker, but a slimmed-down version. Before we get into that process, we need to understand how the Linux kernel supports virtualization and how the evolution of the Linux kernel and CPUs helped advance virtual machines in terms of performance, which in turn led to the creation of containerization technologies.

The intent of this chapter is to explain what a virtual machine is and what is happening under the hood. We also look into some of the basics of hypervisors, which make it possible to run a virtual machine in a system.

History of Virtualization

Prior to the virtualization era, the only way to get full physical servers provisioned was via IT. This was a costly and time-consuming process. One of the major drawbacks of this method was that the machine's resources—like the CPU, memory, and disks—remained underutilized. To get around this, the notion of *virtualization* started to gain traction.

The history of virtualization goes back to the 1960s, when Jim Rymarczyk, who was a programmer with IBM, started virtualizing the IBM mainframe. IBM designed the CP-40 mainframe for internal usage. This system evolved into the CP-67, which used partition technology to run multiple applications at once. Finally came UNIX, which allowed multiple

S. M. Jain, *Linux Containers and Virtualization*, https://doi.org/10.1007/978-1-4842-6283-2_1

programs to run on the x86 hardware. Still the problem of portability remained. In the early 90s, Sun Microsystems came up with Java, which allowed the “write once run anywhere” paradigm to spread its wings. A user could now write a program in Java that could run across a variety of hardware architectures. Java did this by introducing intermediary code (called *bytecode*), which could then be executed on a Java runtime across different hardware architectures. This was the advent of *process-level virtualization,* whereby the Java runtime environment virtualized the POSIX layer.

In the late 1990s, VMware stepped in and launched its own virtualization model. This was related to virtualizing the actual hardware like the CPU, memory, disks, and so on. This meant that on top of the VMware software (also called the *hypervisor*), we could run operating systems themselves (called *guests*). This meant that developers were not restricted to just running Java programs, but could run any program meant to be run on the guest operating system. Around 2001, VMware launched the ESX and GSX servers. GSX was a Type 2 hypervisor so it needed an operating system like Windows to run guests. ESX was a Type 1 hypervisor, which allowed guest OSes to be run directly on the hypervisor.

What Is Virtualization?

Virtualization provides abstraction on top of the actual resources we want to virtualize. The level at which this abstraction is applied changes the way that different virtualization techniques look.

At a higher level, there are two major virtualization techniques based on the level of abstraction.

- Virtual machine (VM)-based
- Container-based

Apart from these two virtualizing techniques, there are other techniques, such as *unikernels*, which are lightweight single-purpose VMs. IBM is currently attempting to run unikernels as processes with projects like Nabla. In this book, we will mainly look at VM-based and container-based virtualizations only.

VM-Based Virtualization

The VM-based approach virtualizes the complete OS. The abstraction it presents to the VM are virtual devices like virtual disks, virtual CPUs, and virtual NICs. In other words, we can state that this is virtualizing the complete ISA (instruction set architecture); as an example, the x86 ISA.

With virtual machines, multiple OSes can share the same hardware resources, with virtualized representations of each of the resources available to the VM. For example, the OS on the virtual machine (also called the *guest*) can continue to do I/O operations on a disk (in this case, it's a virtual disk), thinking that it's the only OS running on the physical hardware (also called the *host*), although in actuality, it is shared by multiple virtual machines as well as by the host OS.

VMs are very similar to other processes in the host OS. VMs execute in a hardware-isolated virtual address space and at a lower privilege level than the host OS. The primary difference between a process and a VM is the ABI (Application Binary Interface) exposed by the host to the VM. In the case of a process, the exposed ABI has constructs like network sockets, FDs, and so on, whereas with a full-fledged OS virtualization, the ABI will have a virtual disk, a virtual CPU, virtual network cards, and so on.

Container-Based Virtualization

This form of virtualization doesn't abstract the hardware but uses techniques within the Linux kernel to isolate access paths for different resources. It carves out a logical boundary within the same operating system. As an example, we get a separate root file system, a separate process tree, a separate network subsystem, and so on.

Hypervisors

A special piece of software is used to virtualize the OS, called the *hypervisor*. The hypervisor itself has two parts:

- **Virtual Machine Monitor (VMM)**: Used for trapping and emulating the privileged instruction set (which only the kernel of the operating system can perform).
- **Device model**: Used for virtualizing the I/O devices.

Virtual Machine Monitor (VMM)

Since the hardware is not available directly on a virtual machine (although in some cases it can be), the VMM traps privileged instructions that access the hardware (like disk/network card) and executes these instructions on behalf of the virtual machine.

The VMM has to satisfy three properties (Popek and Goldberg, 1973):

- **Isolation**: Should isolate guests (VMs) from each other.
- **Equivalency**: Should behave the same, with or without virtualization. This means we run the majority (almost all) of the instructions on the physical hardware without any translation, and so on.

- **Performance**: Should perform as good as it does without any virtualization. This again means that the overhead of running a VM is minimal.

Some of the common functionalities of the VMM are as follows:

- Does not allow the VM to access privileged states; that is, things like manipulating the state of certain host registers should not be allowed from the VM. The VMM will always trap and emulate those calls.
- Handles exceptions and interrupts. If a network call (i.e., a request) was issued from within a virtual machine, it will be trapped in the VMM and emulated. On receipt of a response over the physical network/NIC, the CPU will generate an interrupt and deliver it to the actual virtual machine that it's addressed to.
- Handles CPU virtualization by running the majority of the instructions natively (within the virtual CPU of the VM) and only trapping for certain privileged instructions. This means the performance is almost as good as native code running directly on the hardware.
- Handles memory mapped I/O by mapping the calls to the virtual device-mapped memory in the guest to the actual physical device-mapped memory. For this, the VMM should control the physical memory mappings (Guest Physical memory to Host Physical memory). More details are covered in a later section of this chapter.

Device Model

The device model of the hypervisor handles the I/O virtualization again by trapping and emulating and then delivering interrupts back to the specific virtual machine.

Memory Virtualization

One of the critical challenges with virtualization is how to virtualize the memory. The guest OS should have the same behavior as the non-virtualized OS. This means that the guest OS should probably be at least made to feel that it controls the memory.

In the case of virtualization, the guest OS cannot be given direct access to the physical memory. What this means is that the guest OS should not be able to manipulate the hardware page tables, as this can lead to the guest taking control of the physical system.

Before we delve into how this is tackled, a basic understanding of memory virtualization is needed, even in the context of normal OS and hardware interactions.

The OS provides its processes a virtual view of memory; any access to the physical memory is intercepted and handled by the hardware component called the Memory Management Unit (MMU). The OS sets up the CR3 register (via a privileged instruction) and the MMU uses this entry to walk the page tables to determine the physical mapping. The OS also takes care of changing these mappings when allocation and deallocation of physical memory happens.

Now, in the case of virtualized guests, the behavior should be similar. The guest should not get direct access to the physical memory, but should be intercepted and handled by the VMM.

Basically, there are three memory abstractions involved when running a guest OS:

- **Guest Virtual memory**: This is what the process running on the guest OS sees.
- **Guest Physical memory**: This is what the guest OS sees.
- **System Physical memory**: This is what the VMM sees.

There are two possible approaches to handle this:

- Shadow page tables
- Nested page tables with hardware support

Shadow Page Tables

In the case of shadow page tables, the Guest Virtual memory is mapped directly to the System Physical memory via the VMM. This improves performance by avoiding one additional layer of translation. But this approach has a drawback. When there is a change to the guest page tables, the shadow page tables need to be updated. This means there has to be a trap and emulation into the VMM to handle this. The VMM can do this by marking the guest page tables as read-only. That way, any attempt by the guest OS to write to them causes a trap and the VMM can then update the shadow tables.

Nested Page Tables with Hardware Support

Intel and AMD provided a solution to this problem via hardware extensions. Intel provides something called an *Extended Page Table* (EPT), which allows the MMU to walk two page tables.

The first walk is from the Guest Virtual to the Guest Physical memory and the second walk is from the Guest Physical to the System Physical memory. Since all this translation now happens in the hardware, there is no need to maintain shadow page tables. Guest page tables are maintained by the guest OS and the other page table is maintained by the VMM.

With shadow page tables, the TLB cache (translation look-aside buffer, which is part of MMU) needs to be flushed on a context switch, that is, bringing up another VM. Whereas, in the case of an EPT, the hardware introduces a VM identifier via the address space identifier, which means TLB can have mappings for different VMs at the same time, which is a performance boost.

CPU Virtualization

Before we look into CPU virtualization, it would be interesting to understand how the protection rings are built into the x86 architecture. These rings allow the CPU to protect memory and control privileges and determine what code executes at what privilege level.

The x86 architecture uses the concept of *protection rings.* The kernel runs in the most privileged mode, Ring 0, and the user space used for running processes run is in Ring 3.

The hardware requires that all privileged instructions be executed in Ring 0. If any attempt is made to run a privileged instruction in Ring 3, the CPU generates a fault. The kernel has registered fault handlers and, based on the fault type, a fault handler is invoked. The corresponding fault handler does a sanity check on the fault and processes it. If a sanity check passes, the fault handler handles the execution on behalf of the process. In the case of VM-based virtualization, the VM is run as a process on the host OS, so if a fault is not handled, the whole VM could be killed.

At a high-level, privilege instruction execution from Ring 3 is controlled by a code segment register via the CPL (code privilege level) bit. All calls from Ring 3 are gated to Ring 0. As an example, a system call can be made by an instruction like `syscall` (from user space), which in turn sets the right CPL level and executes the kernel code with a higher privilege level. Any attempt to directly call high-privilege code from upper rings leads to a hardware fault.

The same concept applies to a virtualized OS. In this case, the guest is deprivileged and runs in Ring 1 and the process of the guest runs in Ring 3. The VMM itself runs in Ring 0. With fully virtualized guests, any privileged instruction has to be trapped and emulated. The VMM emulates the trapped instruction. Over and above the privileged instructions, the sensitive instructions also need to be trapped and emulated by the VMM.

Older versions of x86 CPU are not virtualizable, which means not all sensitive instructions are privileged. Instructions like SGDT, SIDT, and more can be executed in Ring 1 without being trapped. This can be harmful when running a guest OS, as this could allow the guest to peek at the host kernel data structures. This problem can be addressed in two ways:

- Binary translation in the case of full virtualization
- Paravirtualization in the case of XEN with hypercalls

Binary Translation in the Case of Full Virtualization

In this case, the guest OS is used without any changes. The instructions are trapped and emulated for the target environment. This causes a lot of performance overhead, as lots of instructions have to be trapped into the host/hypervisor and emulated.

Paravirtualization

To avoid the performance problems related to binary translation when using full virtualization, we use paravirtualization, wherein the guest knows that it is running in a virtualized environment and its interaction with the host is optimized to avoid excessive trapping. As an example, the device driver code is changed and split into two parts. One is the backend (which is with the hypervisor) and the other is the frontend, which is with the guest. The guest and host drivers now communicate over ring buffers. The ring buffer is allocated from the guest memory. Now the guest can accumulate/aggregate data within the ring buffer and make one *hypercall* (i.e., a call to the hypervisor, also called a *kick*) to signal that the data is ready to be drained. This avoids excessive traps from the guest to the host and is a performance win.

In 2005, x86 finally became virtualizable. They introduced one more ring, called Ring -1, which is also called *VMX (virtual machine extensions) root mode*. The VMM runs in VMX root mode and the guests run in non-root mode.

This means that guests can run in Ring 0 and, for the majority of the instructions, there is no trap. Privileged/sensitive instructions that guests need are executed by the VMM in root mode via the trap. We call these switches the *VM Exits* (i.e., the VMM takes over instruction executions from the guest) and *VM Entries* (the VM gains control from the VMM).

Apart from this, the virtualizable CPU manages a data structure called VMCS (VM control structure), and it has the state of the VM and registers. The CPU uses this information during the VM Entries and Exits. The VMCS structure is like `task_struct`, the data structure used to represent a process. One VMCS pointer points to the currently active VMCS. When there is a trap to the VMM, VMCS provides the state of all the guest registers, like the reason of exit, and so on.

Advantages of hardware-assisted virtualization are two-fold:

- No binary translation
- No OS modification

The problem is that the VM Entry and Exits are still heavy calls involving a lot of CPU cycles, as the complete VM state has to be saved and restored. Considerable work has gone into reducing the cycles of these entries and exits. Using paravirtualized drivers helps mitigate some of these performance concerns. The details are explained in the next section.

IO Virtualization

There are generally two modes of IO virtualization:

- Full virtualization
- Paravirtualization

Full Virtualization

With full virtualization, the guest does not know it's running on a hypervisor and the guest O/S doesn't need any changes to run on a hypervisor. Whenever the guest makes I/O calls, they are trapped on the hypervisor and the hypervisor performs the I/O on the physical device.

Paravirtualization

In this case, the guest OS is made aware that it's running in a virtualized environment and special drivers are loaded into the guest to take care of the I/O. The system calls for I/O are replaced with hypercalls.

Figure 1-1 shows the difference between paravirtualization and full virtualization.

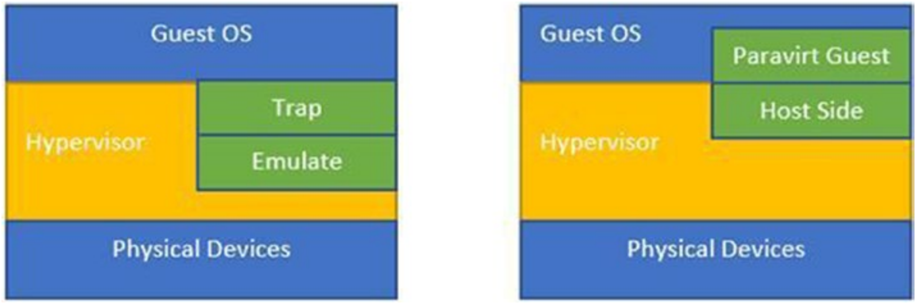


Figure 1-1. *Difference between full and paravirtualized drivers*

With the paravirtualized scenario, the guest-side drivers are called the frontend drivers and the host-side drivers are called the backend drivers. Virtio is the virtualization standard for implementing paravirtualized drivers. The frontend network or I/O drivers of the guest are implemented based on the Virtio standard and the frontend drivers are aware that they are running in a virtual environment. They work in tandem with the backend Virtio drivers of the hypervisor. This working mechanism of frontend and backend drivers helps achieve high-performance network and disk operations and is the reason for most of the performance benefits enjoyed by paravirtualization.

As mentioned, the frontend drivers on the guests implement a common set of interfaces, as described by the Virtio standard. When an I/O call has to be made from the process in the guest, the process invokes the frontend driver API and the driver passes the data packets to the corresponding backend driver through the *virtqueue* (the virtual queue).

The backend drivers can work in two ways:

- They can use QEMU emulation, which means the QEMU emulates the device call via system calls from the user space. This means that hypervisor lets the user space QEMU program make the actual device calls.

- They can use mechanisms like *vhost*, whereby the QEMU emulation is avoided and the hypervisor kernel makes the actual device call.

As mentioned, communication between frontend and backend Virtio drivers is done by the virtqueue abstraction. The virtqueue presents an API to interact, which allows it to enqueue and dequeue buffers. Depending on the driver type, they can use zero or more queues. In the case of a network driver, it uses two virtqueues—one queue for the request and the other to receive the packets. The Virtio block driver, on the other hand, uses only one virtqueue.

Consider this example of a network packet flow, where the guest wants to send some data over the network:

1. The guest initiates a network packet write via the guest kernel.
2. The paravirtualized drivers (Virtio) in guest take those buffers and put them into the virtqueue (`tx`).
3. The backend of the virtqueue is the worker thread, and it receives the buffers.
4. The buffers are then written to the tap device file descriptor. The tap device can be connected to a software bridge like an OVS or Linux bridge.
5. The other side of the bridge has a physical interface, which then takes the data out over the physical layer.

In this example, when a guest places the packets on the `tx` queue, it needs a mechanism to inform the host side that there are packets for handling. There is an interesting mechanism in Linux called `eventfd` that's used to notify the host side that there are events. The host watches the `eventfd` for changes.

A similar mechanism is used to send packets back to the guest.

As you saw in earlier sections, the hardware industry is catching up in the virtualization space and is providing more and more hardware virtualization, be it for CPUs (introducing a new ring) and instructions with `vt-x` or be it for memory (extended page tables).

Similarly, for I/O virtualization, hardware has a mechanism called an I/O memory management unit, which is similar to the memory management unit of CPU, but this is just for I/O-based memory. The concept is similar to CPU MMU, but here the device memory access is intercepted and mapped to allow different guests. Guests are physically mapped to different physical memory and access is controlled by the I/O MMU hardware. This provides the isolation needed for device access.

This feature can be used in conjunction with something called SRIOV (single root I/O virtualization), which allows an SRIOV-compatible device to be broken into multiple virtual functions. The basic idea is to bypass the hypervisor in the data path and use a pass-through mechanism, wherein VM directly communicates with the devices. Details of SRIOV are beyond the scope of this book. Curious users can follow these links for more about SRIOV:

https://blog.scottlowe.org/2009/12/02/what-is-sr-iov/

https://fir3net.com/Networking/Protocols/what-is-sr-iov-single-root-i-o-virtualization.html

CHAPTER 2

Hypervisors

In the previous chapter, we discussed what virtualization is and covered the types of virtualization—VM-based and container-based. In VM-based virtualization, we briefly discussed the role and importance of the hypervisor, which facilitates the creation of virtual machines.

In this chapter, we do a deep dive into hypervisors. Most of the chapter explains virtualization using components like the Linux Kernel Virtual Machine (KVM) and the Quick Emulator (QEMU). Based on these components, we then look at how VMs are created and how data flow between the guest and the hosts is facilitated.

Linux provides hypervisor facilities by using the QEMU in the user space and a specialized kernel module called the KVM (the Linux Kernel Virtual Machine). The KVM uses the Intel `vt-x` extension instruction set to isolate resources at the hardware level. Since the QEMU is a user space process, the kernel treats it like other processes from a scheduling perspective.

Before we discuss the QEMU and KVM, let's touch upon Intel's `vt-x` and its specific instruction set.

S. M. Jain, *Linux Containers and Virtualization*, https://doi.org/10.1007/978-1-4842-6283-2_2

The Intel Vt-x Instruction Set

Intel's virtualization technology (VT) comes in two flavors:

- Vt-x (for Intel x86 IA-32 and 64-bit architectures)
- Vt-i (for the Itanium processor line)

Functionalities wise, they are similar. To understand the need for virtualization support at the CPU level, let's quickly review how programs and the OS interact with the CPU, as well as how programs in VM interact with the CPU.

In the case of regular programs running on the host, the OS translates the program instructions into CPU instructions that are executed by the CPU.

In the case of a virtual machine, to run the programs within the virtual machine, the guest OS translates program instructions into virtual CPU instructions and the hypervisor then converts these into instructions for the physical CPU.

As we can see, for VM, the program instructions are translated twice—the program instructions are translated into virtual CPU instructions and the virtual CPU instructions are translated into physical CPU instructions.

This results in large performance overhead and slows down the virtual machine. CPU virtualization, like the `vt-x` feature, enables complete abstraction of the full prowess of the CPU to the virtual machine so that all the software in the VM can run without a performance hit; it runs as if it were on a dedicated CPU.

The `vt-x` also solves the problem whereby the x86 instructions architecture cannot be virtualized. According to the Popek Goldberg principle for virtualization (https://en.wikipedia.org/wiki/Popek_and_Goldberg_virtualization_requirements), all sensitive instructions must also be privileged. Privileged instructions cause a trap in user mode. In x86, some instructions are sensitive but not privileged. This means running them in the user space would not cause a trap. In effect, this means they are not virtualizable. An example of such an instruction is POPF.

`vt-x` simplifies the VMM software by closing virtualization holes by design:

- **Ring compression**: Prior to the introduction of `vt-x`, the guest OS would run in Ring 1 and the guest OS apps would run in Ring 3. To execute the privileged instructions in the guest OS, we need higher privileges, which are by default not available to the guest (due to security reasons). Therefore, to execute those instructions, we need to trap into the hypervisor (which runs in Ring 0 with more privileges), which can then execute the privileged instruction on behalf of the guest. This is called ring compression or deprivileging. `vt-x` avoids this by running the guest OS directly in Ring 0.
- **Non-trapping instructions**: Instructions like POPF on x86, which ideally should trap into the hypervisor as they are sensitive instructions, actually don't trap. This is a problem as we need program control to shift to the hypervisor for all sensitive instructions. `vt-x` addresses this by running the guest OS in Ring 0, where instructions like POPF can trap into the hypervisor running in Ring -1.
- **Excessive trapping**: Without `vt-x`, all sensitive and privileged instructions trap into the hypervisor in Ring 0. With `vt-x` this becomes configurable and depends on the VMM as to which instructions cause a trap and which can be safely handled in Ring 0. Details of this are beyond the scope of this book.

`vt-x` adds two more modes—the non-root mode (in Ring -1) is where VMM runs and the root mode (in Ring 0) is where the guest OS runs.

To understand how these modes are involved in program execution, lets look at an example. Say that a program is being executed in VM and, during the course of its execution, it makes a system call for I/O. As discussed in the previous chapter, guest programs in user space are executed in Ring 3. When the program makes an I/O call (which is a system call), these instructions are executed at the guest OS kernel level (Ring 0). The guest OS by itself cannot handle I/O calls so it delegates them to the VMM (Ring -1). When the execution goes from Ring 0 to Ring -1, it's called a *VMExit* and when the execution comes back from Ring -1 to Ring 0, it's called a *VMEntry*. This is all shown in Figure 2-1.

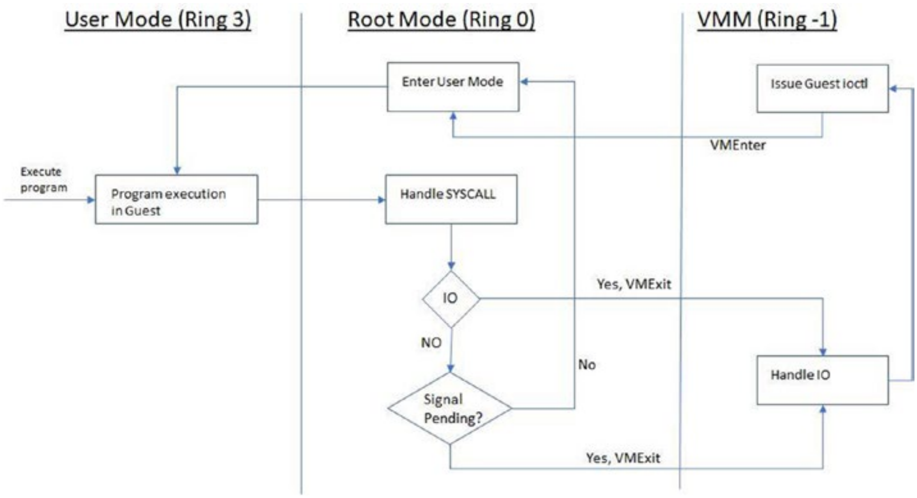


Figure 2-1. *Program execution in the guest with an I/O call*

Note Before we dive into the QEMU, as a side note, we want to bring your attention to some interesting projects in virtualization, like Dune, which runs a process within the VM environment rather than a complete OS. In root mode, it's the VMM that runs. This is the mode where the KVM runs.

The Quick Emulator (QEMU)

The QEMU runs as a user process and handles the KVM kernel module. It uses the `vt-x` extensions to provide the guest with an isolated environment from a memory and CPU perspective. The QEMU process owns the guest RAM and is either memory mapped via a file or is anonymous. VCPUs are scheduled on the physical CPUs.

The main difference between a normal process and a QEMU process is the code executed on those threads. In the case of the guest, since it's the virtualized machine, the code executes the software BIOS and the operating system.

Figure 2-2 shows how the QEMU interacts with the hypervisor.

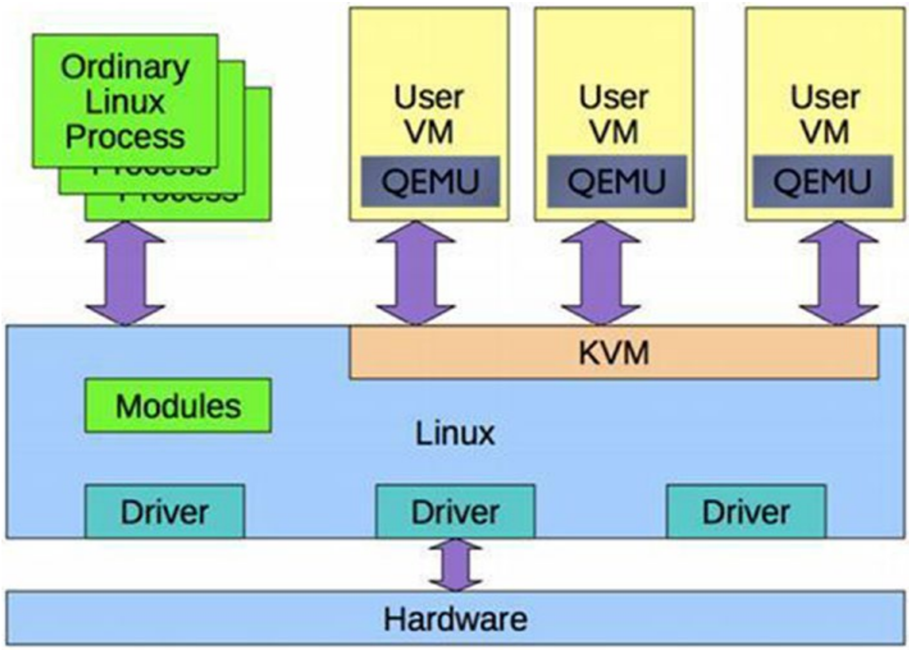


Figure 2-2. *QEMU interaction with the hypervisor*

The QEMU also dedicates a separate thread for I/O. This thread runs an event loop and is based on the non-blocking mechanism. It registers the file descriptors for I/O. The QEMU can use paravirtualized drivers like virtio to provide guests with virtio devices, such as `virtio-blk` for block devices and `virtio-net` for network devices. Figure 2-3 shows the specific components that facilitate communication between the guest and the host (hypervisor).

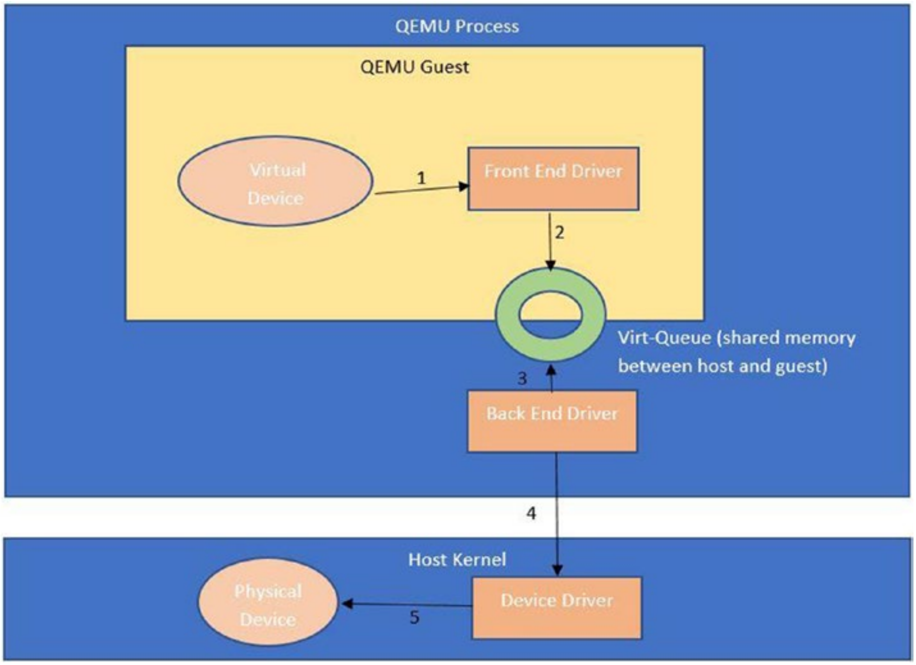


Figure 2-3. *How a virtual device in the guest OS interacts with the physical device in the hypervisor layer. The guest has a frontend device driver, while the host has a backend device driver and these two together facilitate communication between the guest and host OS*

In Figure 2-3, you see that the guest within the QEMU process implements the frontend driver, whereas the host implements the backend drivers. The communication between frontend and backend driver happens over specialized data structures, called *virtqueues*. Any packet that originates from the guest is first put into the virtqueue and the host side driver is notified over a hypercall, to drain the packet for actual processing to the device. There can be two variations of this packet flow, as follows:

- The packet from the guest is received by the QEMU and then pushed to the backend driver on the host. One example is `virtio-net`.
- The packet from the guest directly reaches the host via what is called a *vhost driver*. This bypasses the QEMU layer and is relatively faster.

Creating a VM Using the KVM Module

To create a VM, a set of `ioctl` calls has to be made to the kernel KVM module, which exposes a `/dev/kvm` device to the guest. In simplistic terms, these are the calls from the user space to create and launch a VM:

1. `KVM CREATE VM`: This command creates a new VM that has no virtual CPUs and no memory.
2. `KVM SET USER MEMORY REGION`: This command maps the user space memory for the VM.
3. `KVM CREATE IRQCHIP / KVM CREATE VCPU`: This command creates a hardware component like a virtual CPU and maps them with `vt-x` functionalities.

4. `KVM SET REGS / SREGS / KVM SET FPU / KVM SET CPUID / KVM SET MSRS / KVM SET VCPU EVENTS / KVM SET LAPIC`: These commands are hardware configurations.

5. `KVM RUN`: This command starts the VM.

`KVM RUN` starts the VM and internally it's the VMLaunch instruction invoked by the KVM kernel module that puts the VM code execution into non-root mode. It then changes the instruction pointer to the code location in the guest's memory. This is a slight over-simplification, as the module does much more to set up the VM, including setting up the VMCS (VM Control Section), and so on.

Vhost Based Data Communication

Any discussion about hypervisors would be incomplete without showing a concrete example. We'll look at an example of a network packet flow (depicted in Figure 2-4) in the context of the `vhost-net` device drivers. When we use the vhost mechanism, the QEMU is out of the data plane and there is direct communication between the guest and host over virtqueues. The QEMU remains in the control plane, where it sets up the vhost device on the kernel using the `ioctl` command:

```
/dev/vhost-net device
```

When the device is initialized, a kernel thread is created for the specific QEMU process. This thread handles the I/O for the specific guest. The thread listens to events on the host side, on the virtqueues. When an event arrives to drain the data (in virtio terminology, it's called a *kick*), the I/O thread drains the packet from the tx (transmission) queue of the guest. The thread then transmits this data to the tap device, which it makes it available to the underlying bridge/switch in order to transmit it downstream to an overlay or routing mechanism.

The KVM kernel module registers the `eventfd` for the guest. This a file descriptor that's registered for the guest (by the QEMU) with the KVM kernel module. The FD is registered against a guest I/O exit event (a kick), which drains the data.

What Is an eventfd?

So what basically is an *`eventfd`*? It's an interprocess communication (IPC) mechanism that offers a wait-notify facility between user space programs or between the kernel and the user space. The idea is simple. In the same way that we have FDs for files, we can create file descriptors for events. The benefit here is that the FDs can then be treated like other FDs and can be registered with mechanisms like `poll`, `select`, and `epoll`. The mechanisms can then facilitate a notification system when those FDs are written to.

The consumer thread can be made to wait on an epoll object via `epoll_wait`. Once the producer thread writes to the FD, the epoll mechanism will notify the consumer (again depending on the edge or level triggers) of the event.

Edge-triggered means that you only get notified when the event is detected (which takes place, say in an instant), while level-triggered means you get notified when the event is present (which will be true over a period of time).

For example, in an edge-triggered system, if you want a notification to signal you when data is available to read, you'll only get that notification when data was not available to read before, but now is. If you read some of the available data (so that some of the data is still available to read), you will not get another notification. If you read all of the available data, you will get another notification when new data becomes available to read again. In a level-triggered system, you'd get that notification whenever data is available to read.

The host uses an `eventfd` by using `ioeventfd` to send data from the guest to the host and `irqfd` to receive an interrupt from the host to the guest.

Another use case for `eventfds` is the out of memory (OOM) cgroup. The way this works is whenever the process exceeds the `memcg` limit, the OOM killer can decide to kill it or, if this behavior is disabled, the kernel can do the following

1. Create the `eventfd`.

2. Write the OOM event to the `eventfd`.

The process thread will block until the event is generated. Once the event is generated, the thread is woken up to react to the OOM notification.

The difference between `eventfd` and a Linux pipe is that the pipe needs two file descriptors, whereas `eventfd` just needs one.

The vhost I/O thread watches for the `eventfd`. Whenever the I/O event happens from the guest, the I/O thread for the guest gets informed that it has to drain the buffers from the tx queue.

Similar to `ioeventfd`, there is an `irqfd`. The QEMU user space also registers this (`irqfd`) FD for the guest. The guest driver listens for changes to those FDs. The reason for using this is to pass interrupts back to the guest to notify the guest side driver to process the packets. Taking the previous example, when the packets have to be sent back to the guest, the I/O thread fills up the rx queue (the receive queue) buffers for the guest and the interrupt injection is done to the guest via `irqfd`. In the reverse path of packet flow, the packets received on the host over the physical interface are put to the tap device. The thread that's interfacing with the tap device receives the packets to fill up the rx buffers for the guest. It then notifies the guest driver over `irqfds`. See Figure 2-4.

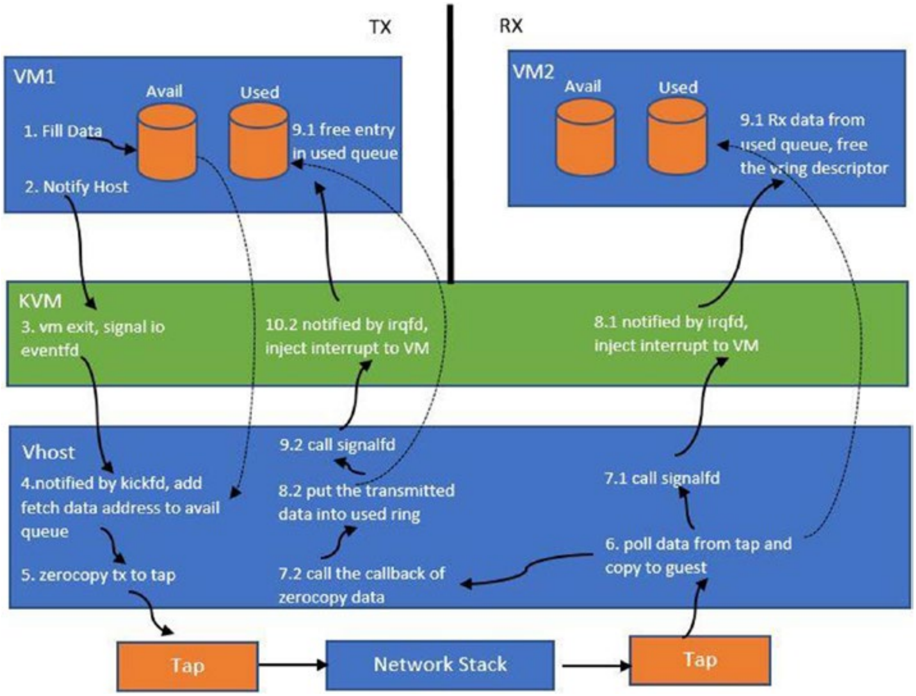


Figure 2-4. *Network packet flow*

Alternative Virtualization Mechanisms

After covering virtualization via VM-based mechanisms, it's time to briefly look at other means of virtualization that depart from container isolation, like the namespaces/cgroups-based mechanism we have with Docker. The motivation is to understand that it is possible to do the following:

- Reduce the interfaces exposed by different software layers like VMM in order to reduce attack vectors. The attack vectors can come in the form of exploits, like memory exploits that install malicious software or control the system by elevating privileges.
- Use hardware isolation to isolate the different containers/processes we run.

In summary, can we get the isolation levels of VMs with a reduced or minimalistic exposed machine interface and with a provisioning speed similar to that of containers.

We have already discussed how VMs, with the help of the VMM, isolate these workloads. The VMM exposes the machine model (x86 interface), whereas the container is exposing the POSIX interface. The VMM, with hardware virtualization, can isolate CPU, memory, and I/O (`vt-d`, SRIOV, and IOMMU). Containers that share the kernel provide this feature via namespaces and cgroups, but are still considered a weaker alternative to the hardware-based isolation techniques.

So is there a way to get the two worlds closer? One of the goals would be to reduce the attack vector by employing a minimalistic interface approach. What this means is that, instead of exposing complete POSIX to apps or a complete machine interface to the guest OS, we only provide what the app/OS needs. This is where we started to see the evolution of how the unikernel and the library OS started to happen.

Unikernels

Unikernels provide the mechanism, via toolchains, for preparing a minimalistic OS. This means if the application only needs network APIs, then the keyboard, mouse devices, and their drivers are not packaged. This reduces the attack vector considerably.

One of the problems with unikernels was that they had to be built across different models of device drivers. With the advent of I/O virtualization and virtio drivers, this problem is somewhat resolved, as the unikernels can now be built with exact virtio devices and the drivers needed for the apps on the guest. This means the guest can be a unikernel (Library OS) sitting on top of, say, a hypervisor like KVM. This still has limitations, as the QEMU or the user space part still has a good amount of codebase, all of which are subject to exploits.

To achieve further minimalism, one proposal was to package the VMM alongside the unikernel. What this means is that VMM now plays the role of the QEMU for the unikernel, but per instance. The VMM code is limited to the needed functionality and facilitates memory-based communication between the guest and the VMM. With this model, multiple VMMs can be made to sit on the hypervisor. The VMM role facilitates I/O and creates the guest unikernel using the hardware isolation capabilities.

The unikernel itself is a single process with no multi-threading capabilities, as shown in Figure 2-5.

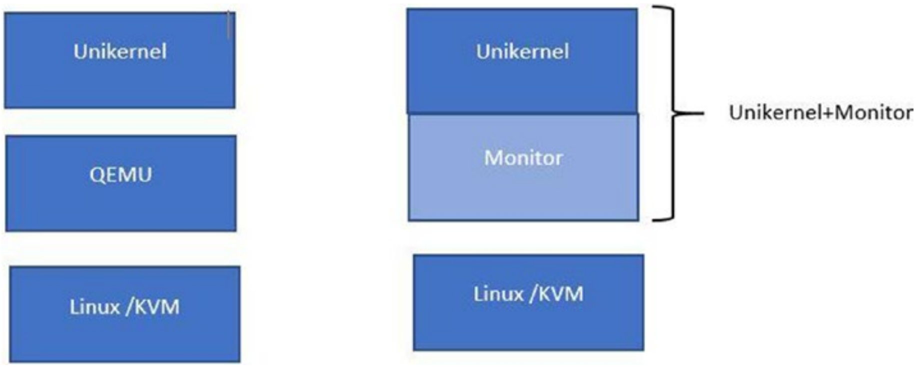


Figure 2-5. *The unikernel is a single process with no multi-threading capabilities*

In Figure 2-5, we can observe that the image on the left is running a VMM and the QEMU combined, to run unikernels on top, whereas the image on the right shows a VMM (monitor) like UKVM packaged alongside the unikernel. So basically we have reduced the code (the QEMU) and thereby have eliminated a significant attack vector. This is in line with the minimalistic interfaces approach we talked about previously.

Project Dune

A careful reader can easily make out that the vt-x isolation on the memory and CPU is not opinionated about running only a guest OS code in the guest's memory. Technically, we can provision different sandboxing mechanisms on top of this hardware isolation. This is precisely what Project Dune is doing. On top of the hardware isolation of vt-x, Dune doesn't spin a guest OS, but a Linux process. This means the process is made to run in Ring 0 of the CPU and has the machine interface exposed to it. The process can be made to sandbox by:

1. Running the trusted code of the process in Ring 0. This is basically the library that Dune calls libdune.

2. Running the untrusted code in Ring 3.

The Dune architecture is shown in Figure 2-6.

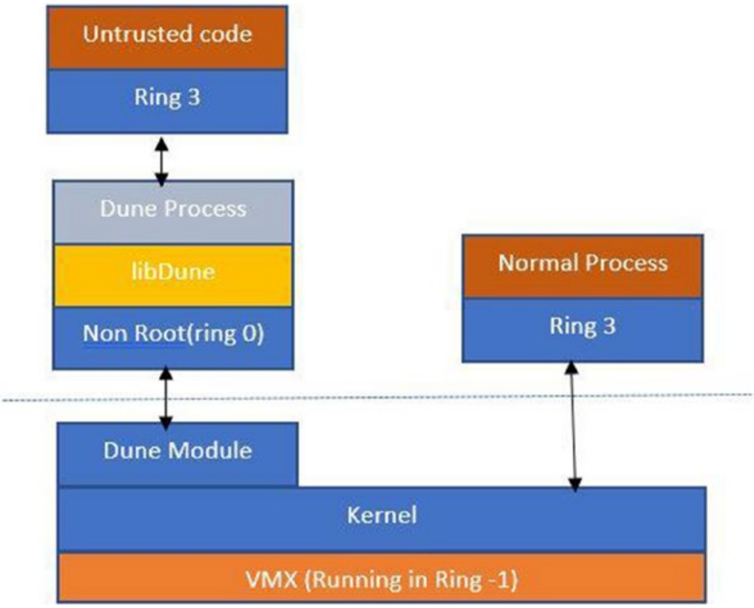


Figure 2-6. *The Dune architecture*

To bootstrap the process, Dune creates an operating environment, which entails setting up the page tables (the CR3 register is pointing to the root). It also sets up the IDT for the hardware exceptions. The trusted and untrusted code runs in the same address space, wherein the memory pages of the trusted code are protected by supervisor bits in page table entries. The system calls trap into the same process and are interposed with hypercalls to the VMM. For more details on Dune, check out `http://dune.scs.stanford.edu/`.

novm

novm is another type of hardware container used by the project. (It also uses the KVM APIs to create the VM via using the `/dev/kvm` device file.) Instead of presenting a disk interface to the VM, novm presents a file system (9p) interface back to the VM. This allows packaging of software that we want to provision as a container. There is no BIOS and the VMM simply puts the VM in 32-bit protected mode directly. This makes the provisioning process faster, because steps like device probing are not needed.

Summary of Alternate Virtualization Approaches

In summary, this chapter covered three approaches—one approach packaged a unikernel with a minimal OS interface, the second approach got rid of the OS interface and ran a process within Ring 0 directly, and the third approach provided a file system into the VM instead of block devices directly and optimized booting aspects.

These approaches provide good isolation at the hardware level and very fast spin-up times and might be a good fit for running serverless workloads and other cloud workloads.

Is this all? Of course not. We now have companies like Cloudflare and Fastly trying to address virtualization by offering isolation within a process. The intent is to use abilities of certain languages to have:

- Code flow isolation via control flow integrity
- Memory isolation
- Capability-based security

We could then use these primitives to build sandboxes within each process itself. This way, we can get even faster boot times for the code we want to execute.

WebAssembly is leading the innovation in this space. The basic idea is to run WebAssembly, aka Wasm modules, within the same process (the WASM runtime). Each module is isolated from the other modules, so we get one sandbox per tenant. This fits well into the serverless computer paradigms and probably prevents problems like cold start.

On a side note, there is a new functionality called *hotplug capability* that makes the devices dynamically available in the guest. They allow developers to dynamically resize the block devices as an example without restarting the guest. There is also the `hotplug-dimm` module, which allows developers to resize the RAM available to the guest.

CHAPTER 3

Namespaces

In this chapter, we touch upon an important aspect of Linux containers, called Linux namespaces. Namespaces allow the kernel to provide isolation by restricting the visibility of the kernel resources like mountpoints, network subsystems among processes scoped to different namespaces. Examples of such namespace visibilities are mount points and network subsystems.

Today, containers are the de facto cloud software provision mechanism. They provide fast spin-up times and have less overhead than a virtual machine. There are certain very specific reasons behind these features.

The VM-based virtualization emulates the hardware and provides an OS as the abstraction. This means that a bulk of the OS code and the device drivers are loaded as part of the provisioning. On other hand, containers virtualize the OS itself. This means that there are data structures within the kernel that facilitate this separation. Most of the time, we are not clear as to what is happening behind the covers.

Linux containers are made of three Linux kernel primitives:

- Linux namespaces
- cgroups
- Layered file systems

S. M. Jain, *Linux Containers and Virtualization*, https://doi.org/10.1007/978-1-4842-6283-2_3

A *namespace* is a logical isolation within the Linux kernel. A namespace controls visibility within the kernel. All the controls are defined at the process level. That means a namespace controls which resources within the kernel a process can see. Think of the Linux kernel as a guard protecting resources like OS memory, privileged CPU instructions, disks, and other resources that only kernel should be able to access. Applications running within user space should only access these resources via a trap, in which case the kernel takes over control and executes these instructions on behalf of the user space application. As an example, an application that wants to access a file on a disk will have to delegate this call to the kernel via a system call (which internally traps into the kernel) to the Linux kernel, which then executes this request on behalf of the application.

Since there could be many user space applications running in parallel on a single Linux kernel, we need a way to provide isolation between these user space-based applications. By isolation, we mean that there should be a kind of sandboxing of the individual application, so that certain resources in the application are confined to that sandbox. As an example, we would like to have file system sandbox, which would mean that within that sandbox, we could have our own view of the files. That way, multiple such sandboxes could be run over the same Linux kernel without interfering with each other.

The technique to achieve such sandboxing is done by a specific data structure in the Linux kernel, called the *namespace*.

Namespace Types

In this section, we explain the different namespaces that exist within the Linux kernel and discuss how they are realized within the kernel.

UTS

This namespace allows a process to see a separate hostname other than the actual global namespace one.

PID

The processes within the PID namespace have a different process tree. They have an `init` process with PID 1. At the data structure level though, the processes belong to one global process tree, which is visible only at the host level. Tools like `ps` or direct usage of the `/proc` file system from within the namespace will list the processes and their related resources for the process tree within the namespace.

Mount

This is one of the most important namespaces. It controls which mount points a process should see. If a process is within a namespace, it will only see the mounts within that namespace.

A small detour might be of help to explain how mount propagation works with containers. A mount in the kernel is represented by a data structure called `vfsmount`. All mounts form a tree-like structure, with a child mount structure holding a reference to the parent mount structure.

```
struct vfsmount {
      struct list_head mnt_hash;
      struct vfsmount *mnt_parent;    /* fs we are mounted on */
      struct dentry *mnt_mountpoint;  /* dentry of mountpoint */
      struct dentry *mnt_root;        /* root of the mounted tree*/
      struct super_block *mnt_sb;     /* pointer to superblock */
      struct list_head mnt_mounts;    /* list of children,
                                         anchored here */
```

```
    struct list_head mnt_child;     /* and going through their
                                       mnt_child */
    atomic_t mnt_count;
    int mnt_flags;
    char *mnt_devname;               /* Name of device e.g.
                                        /dev/dsk/hda1 */
    struct list_head mnt_list;
};
```

Whenever a mount operation is invoked, a `vfsmount` structure is created and the *dentry* of the mount point as well as the *dentry* of the mounted tree is populated. A *dentry* is a data structure that maps the inode to the filename.

Apart from mount, there is a bind mount, which allows a directory (instead of a device) to be mounted at a mount point. The process of bind mounting results in creating a `vfsmount` structure that points to the *dentry* of the directory.

Containers work on the concept of bind mounts. So, when a volume is created for a container, it's actually a bind mount of a directory within the host to a mount point within the container's file system. Since the mount happens within the mount namespace, the `vfsmount` structures are scoped to the mount namespace. This means that, by creating a bind mount of a directory, we can expose a volume within the namespace that's holding the container.

Network

A network namespace gives a container a separate set of network subsystems. This means that the process within the network namespace will see different network interfaces, routes, and iptables. This separates the container network from the host network. We will study this in more depth when we look at an example of the packet flow between two containers in different namespaces on the same host as well as containers in different namespaces within the same host.

IPC

This namespace scopes IPC constructs such as POSIX message queues. Between two processes within the same namespace, IPC is enabled, but it will be restricted if two processes in two different namespaces try to communicate over IPC.

Cgroup

This namespace restricts the visibility of the cgroup file system to the cgroup the process belongs to. Without this restriction, a process could peek at the global cgroups via the `/proc/self/cgroup` hierarchy. This namespace effectively virtualizes the cgroup itself.

Apart from the namespaces mentioned here, as of the writing of this book, there is one more namespace under discussion within the Linux community—called the time namespace.

Time

The time namespace has two main use cases:

- Changes the date and time inside a container
- Adjusts the clocks for a container restored from a checkpoint

The kernel provides access to several clocks: `CLOCK_REALTIME`, `CLOCK_MONOTONIC`, and `CLOCK_BOOTTIME`. The last two clocks are monotonous, but the start points for them are not well defined (currently it is system startup time, but the POSIX says "since an unspecified point in the past") and are different for each system. When a container migrates from one node to another, all the clocks are restored to their consistent states. In other words, they have to continue running from the same point where they were dumped.

Now that you have a basic idea about namespaces, we can study the details about how some of the data structures in the Linux kernel allow this separation when it comes to Linux containers. The term used for these structures is *Linux namespaces.*

The kernel represents each process as a `task_struct` data structure. If we detail this structure and list some of its members, we see the following:

```
/* task_struct member predeclarations (sorted alphabetically):
*/
struct audit_context;
struct backing_dev_info;
struct bio_list;
struct blk_plug;
struct capture_control;
struct cfs_rq;
struct fs_struct;
struct futex_pi_state;
struct io_context;
struct mempolicy;
struct nameidata;
struct nsproxy;
struct perf_event_context;
struct pid_namespace;
struct pipe_inode_info;
struct rcu_node;
struct reclaim_state;
struct robust_list_head;
struct root_domain;
struct rq;
struct sched_attr;
struct sched_param;
struct seq_file;
```

```
struct sighand_struct;
struct signal_struct;
struct task_delay_info;
struct task_group;
```

The nsproxy structure is a holder structure for the different namespaces that a task (process) belongs to.

```
struct nsproxy {
       atomic_t count;
       struct uts_namespace *uts_ns;
       struct ipc_namespace *ipc_ns;
       struct mnt_namespace *mnt_ns;
       struct pid_namespace *pid_ns_for_children;
       struct net           *net_ns;
       struct time_namespace *time_ns;
       struct time_namespace *time_ns_for_children;
       struct cgroup_namespace *cgroup_ns;
};
extern struct nsproxy init_nsproxy;
```

The nsproxy holds the eight namespace data structures. The missing one is the user namespace, which is part of the cred data structure in the task_struct.

There are three system calls that can be used to put tasks into specific namespaces. These are clone, unshare, and setns. The clone and setns calls result in creating a nsproxy object and then adding the specific namespaces needed for the task.

We will talk about network namespaces in this chapter. A network namespace is represented by a net structure. Part of that data structure is shown here:

```
struct net {
        /* First cache line can be often dirtied.
         * Do not place read-mostly fields here.
         */
        refcount_t          passive;        /* To decide when the
                                               network
                                             * namespace should
                                               be freed.
                                             */
        refcount_t           count;         /* To decided when
                                               the network
                                             *  namespace should
be shut down.
                                             */
        spinlock_t           rules_mod_lock;

        unsigned int         dev_unreg_count;

        unsigned int         dev_base_seq;   /* protected by
                                                rtnl_mutex */
        int                  ifindex;

        spinlock_t          nsid_lock;
        atomic_t            fnhe_genid;

        struct list_head    list;            /* list of network
                                                namespaces */
        struct list_head    exit_list;       /* To linked to
                                                call pernet exit
                                              * methods on dead
                                                net (
                                              * pernet_ops_rwsem
                                                read locked),
```

```
                                         * or to unregister
                                           pernet ops
                                         * (pernet_ops_rwsem
                                           write locked).
                                         */
        struct llist_node   cleanup_list;   /* namespaces on
                                           death row */

#ifdef CONFIG_KEYS
        struct key_tag          *key_domain; /* Key domain of
                                            operation tag */
#endif
        struct user_namespace   *user_ns;    /* Owning user
                                            namespace */
        struct ucounts           *ucounts;
        struct idr               netns_ids;

        struct ns_common    ns;

        struct list_head    dev_base_head;
        struct proc_dir_entry    *proc_net;
        struct proc_dir_entry    *proc_net_stat;

#ifdef CONFIG_SYSCTL
        struct ctl_table_set      sysctls;
#endif

        struct sock             *rtnl;         /* rtnetlink socket */
        struct sock             *genl_sock;

        struct uevent_sock      *uevent_sock;  /* uevent socket */

        struct hlist_head       *dev_name_head;
        struct hlist_head       *dev_index_head;
        struct raw_notifier_head     netdev_chain;
```

One of the elements of this data structure is the user namespace to which this network namespace belongs. Apart from that, the major structural part of this is `net_ns_ipv4`, which includes the routing table, net filter rules, and so on.

```
struct netns_ipv4 {
#ifdef CONFIG_SYSCTL
      struct ctl_table_header    *forw_hdr;
      struct ctl_table_header    *frags_hdr;
      struct ctl_table_header    *ipv4_hdr;
      struct ctl_table_header    *route_hdr;
      struct ctl_table_header    *xfrm4_hdr;
#endif
      struct ipv4_devconf        *devconf_all;
      struct ipv4_devconf        *devconf_dflt;
      struct ip_ra_chain __rcu *ra_chain;
      struct mutex         ra_mutex;
#ifdef CONFIG_IP_MULTIPLE_TABLES
      struct fib_rules_ops  *rules_ops;
      bool                       fib_has_custom_rules;
      unsigned int               fib_rules_require_fldissect;
      struct fib_table __rcu    *fib_main;
      struct fib_table __rcu    *fib_default;
#endif
      bool                  fib_has_custom_local_routes;
#ifdef CONFIG_IP_ROUTE_CLASSID
      Int                   fib_num_tclassid_users;
#endif

      struct hlist_head    *fib_table_hash;
      bool                  fib_offload_disabled;
      struct sock          *fibnl;
```

```
	struct sock  * __percpu	*icmp_sk;
	struct sock		*mc_autojoin_sk;

	struct inet_peer_base	*peers;
	struct sock  * __percpu	*tcp_sk;
	struct fqdir		*fqdir;
#ifdef CONFIG_NETFILTER
	struct xt_table		*iptable_filter;
	struct xt_table		*iptable_mangle;
	struct xt_table		*iptable_raw;
	struct xt_table		*arptable_filter;
#ifdef CONFIG_SECURITY
	struct xt_table		*iptable_security;
#endif
	struct xt_table		*nat_table;
#endif

	int sysctl_icmp_echo_ignore_all;
	int sysctl_icmp_echo_ignore_broadcasts;
	int sysctl_icmp_ignore_bogus_error_responses;
	int sysctl_icmp_ratelimit;
	int sysctl_icmp_ratemask;
	int sysctl_icmp_errors_use_inbound_ifaddr;

	struct local_ports ip_local_ports;

	int sysctl_tcp_ecn;
	int sysctl_tcp_ecn_fallback;

	int sysctl_ip_default_ttl;
	int sysctl_ip_no_pmtu_disc;
	int sysctl_ip_fwd_use_pmtu;
	int sysctl_ip_fwd_update_priority;
	int sysctl_ip_nonlocal_bind;
```

```
int sysctl_ip_autobind_reuse;
/* Shall we try to damage output packets if routing dev
   changes? */
int sysctl_ip_dynaddr;
```

This is how the iptables and routing rules are all scoped into the network namespace.

Other data structures of relevance here are the `net_device` (this is how the kernel represents the network card/device) and `sock` (a kernel representation of a socket data structure). These two structures allow the device to be scoped into a network namespace as well as the socket to be scoped to the namespace. Both these structures can be part of only one namespace at a time. We can move the device to a different namespaces via the `iproute2` utility.

Here are some of the user space commands to handle the network namespaces:

- `Ip netns add testns`: Adds a network namespace
- `Ip netns del testns`: Deletes the mentioned namespace
- `Ip netns exec testns sh`: Executes a shell within the `testns` namespace

Adding a Device to a Namespace

First, create a `veth` pair device (this device can be used to join two namespaces):

```
ip link add veth0 type veth peer name veth1
```

Then add one end of the `veth` pair to the network namespace `testns`:

```
ip link set veth1 netns testns
```

The other end (veth0) is in the host namespace and so any traffic sent to veth0 ends up on veth1 in the testns namespace.

Assume that we run an HTTP server in the testns namespace, which means the listener socket is scoped to the testns namespace, as explained previously in the sock data structure. So a TCP packet to be delivered to the IP and port of the application within the testns namespace would be delivered to the socket scoped within that namespace.

This is how the kernel virtualizes the operating system and various subsystems like networking, IPC, mounts, and so on.

Summary

In this chapter, we learned about the Linux namespaces and how they facilitate isolation between user space-based applications. We also looked into how different Linux kernel-based data structures are used to realize the different namespaces. Going forward, we will look into how Linux kernel provides resource limits to the different user space-based processes so that one process doesn't hog the resources of the operating system.

CHAPTER 4

Cgroups

In the previous chapter, we learned how to control visibility of Linux processes by using namespaces and learned how they are realized within the kernel. In this chapter, we touch upon another important aspect—resource control—which enables us to apply quotas to various kernel resources.

We learned about namespaces so we could restrict the visibility of resources for processes, which we did by putting the processes in separate namespaces. We also covered the data structures involved in the kernel, to get an understanding of how a namespace is realized within the Linux kernel.

Now we ask ourselves the question, as to whether restricting visibility is good enough for virtualization or do we need more. Assume we run `tenant1` processes in one namespace and `tenant2` processes in a separate namespace. Although the processes can't access each other's resources (mount points, process trees, and so on), as those resources are scoped to the individual namespace, we don't achieve true isolation just via this scoping.

As an example, what stops `tenant1` from launching a process that possibly could hog the CPU via an infinite loop? Flawed code can keep leaking memory (say, for example, it takes a big chunk of the OS page cache). A misbehaving process can create tons of processes via forking, launch a fork bomb, and crash the kernel.

S. M. Jain, *Linux Containers and Virtualization*, https://doi.org/10.1007/978-1-4842-6283-2_4

This means we need a way to introduce resource controls for processes within the namespace. This is achieved using a mechanism called *control groups,* commonly known as *cgroups*. cgroups work on the concept of cgroup controllers and are represented by a file system called `cgroupfs` in the Linux kernel.

The version of cgroups currently being used is cgroup v2. We explore some details about how cgroups work as well as some of the cgroup controllers seen in the kernel code. We also look at how the cgroups are realized within the Linux kernel. But before that, let's briefly see what cgroups are all about.

First, to use the cgroup, we need to mount the `cgroup` file system at a mount point, as follows:

```
mount -t cgroup2 none $MOUNT_POINT
```

The difference between cgroup version v1 and v2 is that, while mounting in v1, we could have specified the mount options to specify the controllers to enable, while in cgroup v2, no such mount option can be passed.

Creating a Sample cgroup

Let's create a sample cgroup called `mygrp`. To create a cgroup, we first need to create a folder where the cgroup artifacts are stored, as follows:

```
mkdir mygrp
```

Now we can create a cgroup using the following command (Note: `cgroup2` is supported in kernel version 4.12.0-rc5 onward. I am working on Ubuntu 19.04, which has kernel version Ubuntu 19.04 with Linux kernel 5.0.0-13.)

```
mount -t cgroup2 none mygrp
```

```
root@osboxes:~# mkdir mygrp
root@osboxes:~# mount -t cgroup2 none mygrp
root@osboxes:~# cd mygrp
root@osboxes:~/mygrp# ls -l
total 0
-r--r--r--  1 root root 0 Jul  2 00:29 cgroup.controllers
-rw-r--r--  1 root root 0 Jul  2 00:29 cgroup.max.depth
-rw-r--r--  1 root root 0 Jul  2 00:29 cgroup.max.descendants
-rw-r--r--  1 root root 0 Jul  2 00:29 cgroup.procs
-r--r--r--  1 root root 0 Jul  2 00:29 cgroup.stat
-rw-r--r--  1 root root 0 Jul  2 00:29 cgroup.subtree_control
-rw-r--r--  1 root root 0 Jul  2 00:29 cgroup.threads
drwxr-xr-x  2 root root 0 Jul  2 00:29 init.scope
drwxr-xr-x 52 root root 0 Jul  2 00:29 system.slice
drwxr-xr-x  4 root root 0 Jul  2 00:25 user.slice
root@osboxes:~/mygrp#
```


```
root@osboxes:~# mkdir mygrp
root@osboxes:~# mount -t cgroup2 none mygrp
root@osboxes:~# cd mygrp
root@osboxes:~/mygrp# ls -l
total 0
-r--r--r--  1 root root 0 Jul  2 00:29 cgroup.controllers
-rw-r--r--  1 root root 0 Jul  2 00:29 cgroup.max.depth
-rw-r--r--  1 root root 0 Jul  2 00:29 cgroup.max.descendants
-rw-r--r--  1 root root 0 Jul  2 00:29 cgroup.procs
-r--r--r--  1 root root 0 Jul  2 00:29 cgroup.stat
-rw-r--r--  1 root root 0 Jul  2 00:29 cgroup.subtree_control
-rw-r--r--  1 root root 0 Jul  2 00:29 cgroup.threads
drwxr-xr-x  2 root root 0 Jul  2 00:29 init.scope
drwxr-xr-x 52 root root 0 Jul  2 00:29 system.slice
drwxr-xr-x  4 root root 0 Jul  2 00:25 user.slice
root@osboxes:~/mygrp#
```


We created a directory called `mygrp` and then mounted the cgroup v2 file system on it. When we navigate inside the `mygrp` directory, we can see multiple files there:

`cgroup.controllers`: This file contains the supported controllers. All controllers that are not mounted on cgroup v1 will show up. Currently on my system, I have a cgroup v1 mounted by `systemd`. We can see that all the controllers are there.

```
root@osboxes:~/mygrp# mount | grep cgroup
tmpfs on /sys/fs/cgroup type tmpfs (ro,nosuid,nodev,noexec,mode=755)
cgroup2 on /sys/fs/cgroup/unified type cgroup2 (rw,nosuid,nodev,noexec,relatime)
cgroup on /sys/fs/cgroup/systemd type cgroup (rw,nosuid,nodev,noexec,relatime,xattr,name=systemd)
cgroup on /sys/fs/cgroup/blkio type cgroup (rw,nosuid,nodev,noexec,relatime,blkio)
cgroup on /sys/fs/cgroup/memory type cgroup (rw,nosuid,nodev,noexec,relatime,memory)
cgroup on /sys/fs/cgroup/perf_event type cgroup (rw,nosuid,nodev,noexec,relatime,perf_event)
cgroup on /sys/fs/cgroup/cpu,cpuacct type cgroup (rw,nosuid,nodev,noexec,relatime,cpu,cpuacct)
cgroup on /sys/fs/cgroup/hugetlb type cgroup (rw,nosuid,nodev,noexec,relatime,hugetlb)
cgroup on /sys/fs/cgroup/devices type cgroup (rw,nosuid,nodev,noexec,relatime,devices)
cgroup on /sys/fs/cgroup/freezer type cgroup (rw,nosuid,nodev,noexec,relatime,freezer)
cgroup on /sys/fs/cgroup/cpuset type cgroup (rw,nosuid,nodev,noexec,relatime,cpuset)
none on /root/mygrp type cgroup2 (rw,relatime)
```


```
root@osboxes:~/mygrp# mount | grep cgroup
tmpfs on /sys/fs/cgroup type tmpfs (ro,nosuid,nodev,noexec,mode=755)
cgroup2 on /sys/fs/cgroup/unified type cgroup2 (rw,nosuid,nodev,noexec,relatime)
cgroup on /sys/fs/cgroup/systemd type cgroup (rw,nosuid,nodev,noexec,relatime,xattr,name=systemd)
cgroup on /sys/fs/cgroup/blkio type cgroup (rw,nosuid,nodev,noexec,relatime,blkio)
cgroup on /sys/fs/cgroup/memory type cgroup (rw,nosuid,nodev,noexec,relatime,memory)
cgroup on /sys/fs/cgroup/perf_event type cgroup (rw,nosuid,nodev,noexec,relatime,perf_event)
cgroup on /sys/fs/cgroup/cpu,cpuacct type cgroup (rw,nosuid,nodev,noexec,relatime,cpu,cpuacct)
cgroup on /sys/fs/cgroup/hugetlb type cgroup (rw,nosuid,nodev,noexec,relatime,hugetlb)
cgroup on /sys/fs/cgroup/devices type cgroup (rw,nosuid,nodev,noexec,relatime,devices)
cgroup on /sys/fs/cgroup/freezer type cgroup (rw,nosuid,nodev,noexec,relatime,freezer)
cgroup on /sys/fs/cgroup/cpuset type cgroup (rw,nosuid,nodev,noexec,relatime,cpuset)
none on /root/mygrp type cgroup2 (rw,relatime)
```


Only after unmounting the controllers from v1 should v2 show these controllers. Sometimes we might need to add the kernel boot parameter `systemd.unified_cgroup_hierarchy=1` and reboot the kernel to make these changes effective. After making the changes on my machine, I see the following controllers:

```
root@osboxes:~# mount -t cgroup2 none mygrp
root@osboxes:~# cd mygrp/
root@osboxes:~/mygrp# ls
cgroup.controllers  cgroup.max.depth  cgroup.max.descendants  cgro
root@osboxes:~/mygrp# cat cgroup.controllers
cpu io memory
root@osboxes:~/mygrp# |
```

`Cgroup.procs`: This file contains the processes within the root cgroup. No PIDs will be there when the cgroup is freshly created. By writing the PIDs to this file, they become part of the cgroup.

`Cgroup.subtree_control`: This holds controllers that are enabled for the immediate subgroup.

Enabling and disabling controllers in the immediate subgroups of a parent is done only by writing into its `cgroup.subtree_control` file. So, for example, enabling the memory controller is done using this:

```
echo "+memory" > mygrp/cgroup.subtree_control
```

And disabling it is done using this:

```
echo "-memory" > mygrp/cgroup.subtree_control
```

`cgroup.events`: This is the cgroup core interface file. This interface file is unique to non-root subgroups. The `cgroup.events` file reflects the number of processes attached to the subgroup, and it consists of one item—`populated: value`. The value is `0` when there are no processes attached to that subgroup or its descendants, and `1` when there are one or more processes attached to that subgroup or its descendants.

Apart from these files, controller-specific interface files are also created. As an example, for memory controllers, a `memory.events` file is created, which can be monitored for events like OOM. Similarly, a PID controller has files like `pids.max` to avoid situations like a fork bomb.

In my example, I go ahead and create a child cgroup under `mygrp`. We can see the following files under the `child` directory:

```
root@osboxes:~/mygrp# cd child/
root@osboxes:~/mygrp/child# ls
cgroup.controllers  cgroup.max.depth        cgroup.procs  cgroup.subtree_control
cgroup.events       cgroup.max.descendants  cgroup.stat   cgroup.threads
```

```
cgroup.subtree_control  cgroup.type  cpu.stat    cpu.weight.nice  io.stat    memory.current  memory.high  memory.max  memory.oom.group
cgroup.threads          cpu.max      cpu.weight  io.max           io.weight  memory.events   memory.low   memory.min  memory.stat
```

We can see controller-specific files like `memory.max`. The interface file called `memory.events` lists the different events like `oom`, which can be enabled and disabled.

```
root@osboxes:~/mygrp/child# cat memory.events
low 0
high 0
max 0
oom 0
oom_kill 0
```

The next section explains how cgroups are implemented within the kernel and how they enable resource control.

Cgroup Types

There are different types of cgroups, based on which resources we want to control. Two of the cgroups we will cover are as follows:

- CPU: Provides CPU limits to user space processes
- Block I/O: Provides I/O limits on block devices for user space processes

CPU Cgroup

From the kernel perspective, let's see how a cgroup is realized. CPU cgroups can be realized on top of two schedulers:

- Completely fair scheduler
- Real-time scheduler

In this chapter, we discuss only the completely fair scheduler (CFS). The CPU cgroup provides two types of CPU resource control:

- `cpu.shares`: Contains an integer value that specifies a relative share of CPU time available to the tasks in a cgroup. For example, tasks in two cgroups that have `cpu.shares` set to 100 will receive equal CPU time, but tasks in a cgroup that have `cpu.shares` set to 200 receive twice the CPU time of the tasks in a cgroup where `cpu.shares` is set to 100. The value specified in the `cpu.shares` file must be 2 or higher.
- `cpu.cfs_quota_us`: Specifies the total amount of time in microseconds (µs, represented here as "us") for which all tasks in a cgroup can run during one period (as defined by `cpu.cfs_period_us`). As soon as tasks in a cgroup use all the time specified by the quota, they are stopped for the remainder of the time specified by the period and not allowed to run until the next period.
- `Cpu.cfs_period_us`*:* It is the period from which CPU quotas for cgroups (`cpu.cfs_quota_us`) are carved out and the quota and period parameters operate on a per CPU basis. Consider these examples:

- To allow the cgroup to be able to access a single CPU for 0.2 seconds of every second, set the `cpu.cfs_quota_us` to 200000 and `cpu.cfs_period_us` to 1000000.
- To allow a process to utilize 100% of a single CPU, set `cpu.cfs_quota_us` to 1000000 and `cpu.cfs_period_us` to 1000000.
- To allow a process to utilize 100% of two CPUs, set `cpu.cfs_quota_us` to 2000000 and `cpu.cfs_period_us` to 1000000.

To understand both of these control mechanisms, we can look into the aspects of the Linux CFS task scheduler. The aim of this scheduler is to grant a fair share of the CPU resources to all the tasks running on the system.

We can break up these tasks into two types:

- **CPU-intensive tasks**: Tasks like encryption, machine learning, query processing, and so on
- **I/O-intensive tasks**: Tasks that are using disk or network I/O like DB clients

The scheduler has the responsibility of scheduling both kinds of tasks. The CFS uses a concept of a `vruntime`. `vruntime` is a member of the `sched_entity` structure, which is a member of the `task_struct` structure (each process is represented in Linux by a `task_struct` structure):

```
struct task_struct {
        int prio, static_prio, normal_prio; unsigned int rt_priority;
        struct list_head run_list;
        const struct sched_class *sched_class;
        struct sched_entity se;
        unsigned int policy; cpumask_t cpus_allowed; unsigned
        int time_slice;
        }
```

```
struct sched_entity {
        /* For load-balancing: */
        struct load_weight          load;
        struct rb_node              run_node;
        struct list_head            group_node;
        unsigned int                on_rq;

        u64                         exec_start;
        u64                         sum_exec_runtime;
        u64                         vruntime;
        u64                         prev_sum_exec_runtime;

        u64                         nr_migrations;

        struct sched_statistics     statistics;

#ifdef CONFIG_FAIR_GROUP_SCHED
        Int                       depth;
        struct sched_entity       *parent;
        /* rq on which this entity is (to be) queued: */
        struct cfs_rq             *cfs_rq;
        /* rq "owned" by this entity/group: */
        struct cfs_rq             *my_q;
        /* cached value of my_q->h_nr_running */
        unsigned long             runnable_weight;
```

The `task_struct` has a reference to `sched_entity`, which holds a reference to `vruntime`.

`vruntime` is calculated using these steps:

1. Compute the time spent by the process on the CPU.
2. Weigh the computed running time against the number of runnable processes.

The kernel uses the update_curr function defined in the https://elixir.bootlin.com/linux/latest/source/kernel/sched/fair.c file.

```
/*
 * Update the current task's runtime statistics.
 */
static void update_curr(struct cfs_rq *cfs_rq)
{
        struct sched_entity *curr = cfs_rq->curr;
        u64 now = rq_clock_task(rq_of(cfs_rq));
        u64 delta_exec;

        if (unlikely(!curr))
                return;

        delta_exec = now - curr->exec_start;
        if (unlikely((s64)delta_exec <= 0))
                 return;

        curr->exec_start = now;

        schedstat_set(curr->statistics.exec_max,
                max(delta_exec, curr->statistics.exec_max));

        curr->sum_exec_runtime += delta_exec;
        schedstat_add(cfs_rq->exec_clock, delta_exec);

        curr->vruntime += calc_delta_fair(delta_exec, curr);
        update_min_vruntime(cfs_rq);

        if (entity_is_task(curr)) {
                struct task_struct *curtask = task_of(curr);

                trace_sched_stat_runtime(curtask, delta_exec,
                curr->vruntime);
                cgroup_account_cputime(curtask, delta_exec);
```

```
            account_group_exec_runtime(curtask, delta_exec);
        }

        account_cfs_rq_runtime(cfs_rq, delta_exec);
}
```

The function first calculates the delta_exec, which is the time spent by the current task on the CPU.

This delta_exec is then passed as a parameter to another function call, named calc_delta_fair.

This call will return the weighted value of the process runtime in relation to the number of runnable processes. Once vruntime is calculated, it's stored as part of the sched_entity structure.

Also, as part of updating the vruntime for the task, the update_curr function calls update_min_vruntime. This calculates the smallest value of vruntime among all runnable processes and adds it to a red black tree as the leftmost node. The CFS scheduler can then look into the red black tree to schedule the process that has the lowest vruntime.

Basically, the CFS scheduler schedules its heuristic's schedules and I/O-intensive tasks more frequently, but gives more time to the CPU-intensive tasks in a single run. This also could be understood from the vruntime concept discussed previously. Since I/O tasks are mostly waiting for network/disk, their vruntimes tend to be smaller than CPU tasks. That means the I/O tasks will be scheduled more frequently. The CPU-intensive tasks will get more time once they are scheduled to do the work. This way, CFS tries to attain a fair scheduling of tasks.

Let's stop for a minute and think about a potential problem this scheduling could lead to.

Assume you have two processes, A and B, belonging to different users. These processes each get 50% share of the CPU. Now say a user owning process A launches another process, called A1. Now CFS will give a 33% share to each process. This effectively means that users of process A and A1 now get 66% of the CPU. A classic example is a database like PostgreSQL, which creates processes per connection. As connections grow, the number of processes grow. If fair scheduling is in place, each connection would tend to take away the share of the other non-Postgre processes running on the same machine.

This problem led to what we call group scheduling. To understand this, let's look at other kernel data structure:

```
/* CFS-related fields in a runqueue */
struct cfs_rq {
	struct load_weight    load;
	unsigned int           nr_running;
	unsigned int           h_nr_running;     /* SCHED_{NORMAL,
	                                            BATCH,IDLE} */
	unsigned int          idle_h_nr_running; /* SCHED_IDLE */

	u64                   exec_clock;
	u64                   min_vruntime;
#ifndef CONFIG_64BIT
	u64                   min_vruntime_copy;
#endif

	struct rb_root_cached tasks_timeline;

	/*
	 * 'curr' points to currently running entity on this
	   cfs_rq.
	 * It is set to NULL otherwise (i.e. when none are
	   currently running).
	 */
	struct sched_entity *curr;
	struct sched_entity *next;
	struct sched_entity *last;
	struct sched_entity *skip;
```

This structure holds the number of runnable tasks in the nr_running member. The curr member is a pointer to the current running scheduling entity or the task.

Also, the sched_entity is now represented as a hierarchical data structure:

```
struct sched_entity {
/* For load-balancing: */
struct load_weight      load;
struct rb_node          run_node;
struct list_head        group_node;
unsigned int            on_rq;

u64                     exec_start;
u64                     sum_exec_runtime;
u64                     vruntime;
u64                     prev_sum_exec_runtime;

u64                     nr_migrations;

struct sched_statistics     statistics;

#ifdef CONFIG_FAIR_GROUP_SCHED
       Int                            depth;
       struct sched_entity      *parent;
       /* rq on which this entity is (to be) queued: */
       struct cfs_rq            *cfs_rq;
       /* rq "owned" by this entity/group: */
       struct cfs_rq            *my_q;
       /* cached value of my_q->h_nr_running */
       unsigned long             runnable_weight;
#endif
```

```
#ifdef CONFIG_SMP
       /*
        * Per entity load average tracking.
        *
        * Put into separate cache line so it does not
        * collide with read-mostly values above.
        */
         struct sched_avg avg;
#endif
};
```

This means there can now be sched_entities that are not associated with a process (task_struct). Instead, these entities can represent a group of processes. Each sched_entity now maintains a run queue of its own. A process can be moved to the child schedule entity, which means it will be part of the run queue that the child schedule entity has. This run queue can represent the processes in the group.

The code flow in scheduler would do the following.

Pick_next_entity is called to pick up the best candidate for scheduling. We assume that there is only one group running at this time. This means that the red black tree associated with the sched_entity process is blank. The method now tries to get the child sched_entity of the current sched_entity. It checks the cfs_rq, which has the processes of the group enqueued. The process is scheduled.

The vruntime is based on the weights of the processes within the group. This allows us to do fair scheduling and prevent processes within a group from impacting the CPU usage of processes within other groups.

Once we understand that processes can be placed into groups, let's see how bandwidth enforcement can be applied to the group. Another data structure called cfs_bandwidth, defined in sched.h, plays a role:

```
struct cfs_bandwidth {
#ifdef CONFIG_CFS_BANDWIDTH
        raw_spinlock_t          lock;
        ktime_t                 period;
        u64                 quota;
        u64                 runtime;
        s64                 hierarchical_quota;

        u8                  idle;
        u8                  period_active;
        u8                  distribute_running;
        u8                  slack_started;
        struct hrtimer          period_timer;
        struct hrtimer          slack_timer;
        struct list_head    throttled_cfs_rq;

        /* Statistics: */
        Int                 nr_periods;
        Int                 nr_throttled;
        u64                 throttled_time;
#endif
};
```

This structure keeps track of the runtime quota for the group. The `cff_bandwith_used` function is used to return a Boolean value when the check is made in the `account_cfs_rq_runtime` method of the fair scheduler implementation file. If no runtime quota remains, the `throttle_cfs_rq` method is invoked. It will dequeue the task from the run queue of the `sched_entity` and set the throttled flag. The function implementation is shown here:

```
static void throttle_cfs_rq(struct cfs_rq *cfs_rq)
{
```

```
struct rq *rq = rq_of(cfs_rq);
struct cfs_bandwidth *cfs_b = tg_cfs_bandwidth(cfs_rq->tg);
struct sched_entity *se;
long task_delta, idle_task_delta, dequeue = 1;
bool empty;

se = cfs_rq->tg->se[cpu_of(rq_of(cfs_rq))];

/* freeze hierarchy runnable averages while throttled */
rcu_read_lock();
walk_tg_tree_from(cfs_rq->tg, tg_throttle_down, tg_nop,
(void *)rq);
rcu_read_unlock();

task_delta = cfs_rq->h_nr_running;
idle_task_delta = cfs_rq->idle_h_nr_running;
for_each_sched_entity(se) {
    struct cfs_rq *qcfs_rq = cfs_rq_of(se);
    /* throttled entity or throttle-on-deactivate */
    if (!se->on_rq)
           break;

    if (dequeue) {
           dequeue_entity(qcfs_rq, se, DEQUEUE_SLEEP);
    } else {
           update_load_avg(qcfs_rq, se, 0);
           se_update_runnable(se);
    }

    qcfs_rq->h_nr_running -= task_delta;
    qcfs_rq->idle_h_nr_running -= idle_task_delta;

    if (qcfs_rq->load.weight)
           dequeue = 0;
```

```
        }

        if (!se)
            sub_nr_running(rq, task_delta);

        cfs_rq->throttled = 1;
        cfs_rq->throttled_clock = rq_clock(rq);
        raw_spin_lock(&cfs_b->lock);
        empty = list_empty(&cfs_b->throttled_cfs_rq);

        /*
        * Add to the _head_ of the list, so that an already-started
        * distribute_cfs_runtime will not see us. If disribute_
          cfs_runtime is
         * not running add to the tail so that later runqueues
           don't get starved.
         */
        if (cfs_b->distribute_running)
              list_add_rcu(&cfs_rq->throttled_list, &cfs_b-
              >throttled_cfs_rq);
        else
              list_add_tail_rcu(&cfs_rq->throttled_list,
              &cfs_b->throttled_cfs_rq);

        /*
         * If we're the first throttled task, make sure the
         bandwidth
         * timer is running.
         */
        if (empty)
              start_cfs_bandwidth(cfs_b);

        raw_spin_unlock(&cfs_b->lock);
}
```

This explains how the CPU cgroups allow tasks/processes to be grouped and can use the CPU shares mechanism to enforce fair scheduling within a group. This also explains how quota and bandwidth enforcement is accomplished within a group. We now discuss the other cgroup type, which enforces resource limits on block I/O.

Block I/O cgroups

The purpose of the block I/O cgroup is twofold:

- Provides fairness to the individual cgroup: Makes use of a scheduler called completely fair queuing.
- Does block i/o throttling: Enforces a quota on the block I/O (bytes as well as iops) per cgroup.

Before delving into details of how the cgroup for block I/O is implemented, we'll take a small detour to investigate how the Linux block I/O works. Figure 4-1 is a high-level block diagram of how the block I/O request flows through the user space to the device.

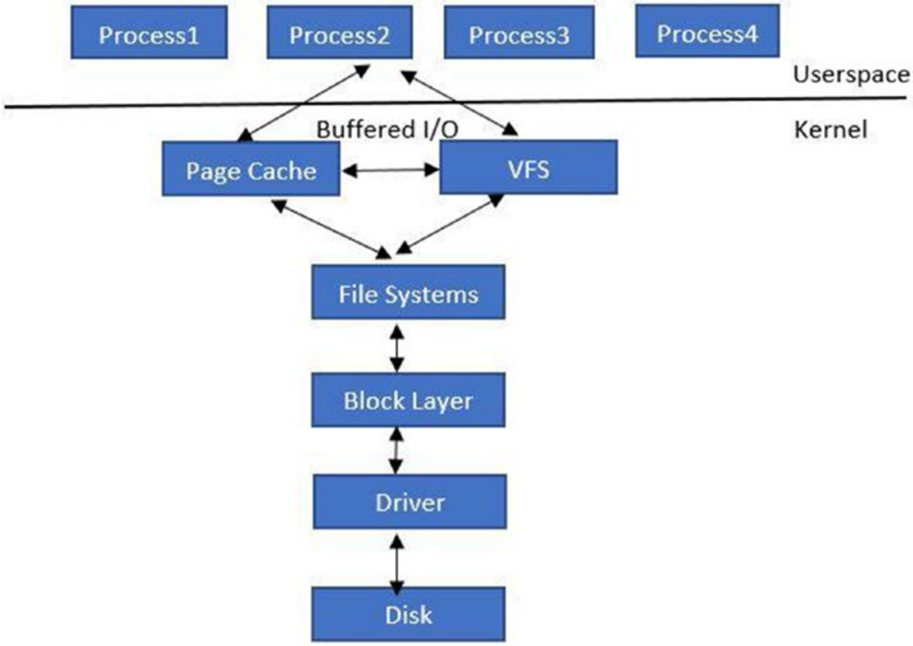


Figure 4-1. *The block I/O request flows through the user space to the device*

The application issues a read/write request via either the file system or via memory mapped files. In either case, the request hits the page cache (kernel buffer for caching file data). With a file system-based call, the virtual file system (VFS) handles the system call and invokes the underlying registered file system.

The next layer is the block layer where the actual I/O request is constructed. There are three important data structures within the block layer:

- `Request_queue`: A single queue architecture is where there is one request queue per device. This is the queue where the block layer, in tandem with the I/O scheduler, queues the request. The device driver drains the request queue and submits the request to the actual device.

- Request: The request represents the single I/O request to be delivered to the I/O device. The request is made of a list of bio structures.
- **Bio**: The bio structure is the basic container for block I/O. Within the kernel is the bio structure. Defined in <linux/bio.h>, this structure represents block I/O operations that are in flight (active) as a list of segments. A segment is a chunk of a buffer that is contiguous in memory.

Diagrammatically, bio is shown in Figure 4-2.

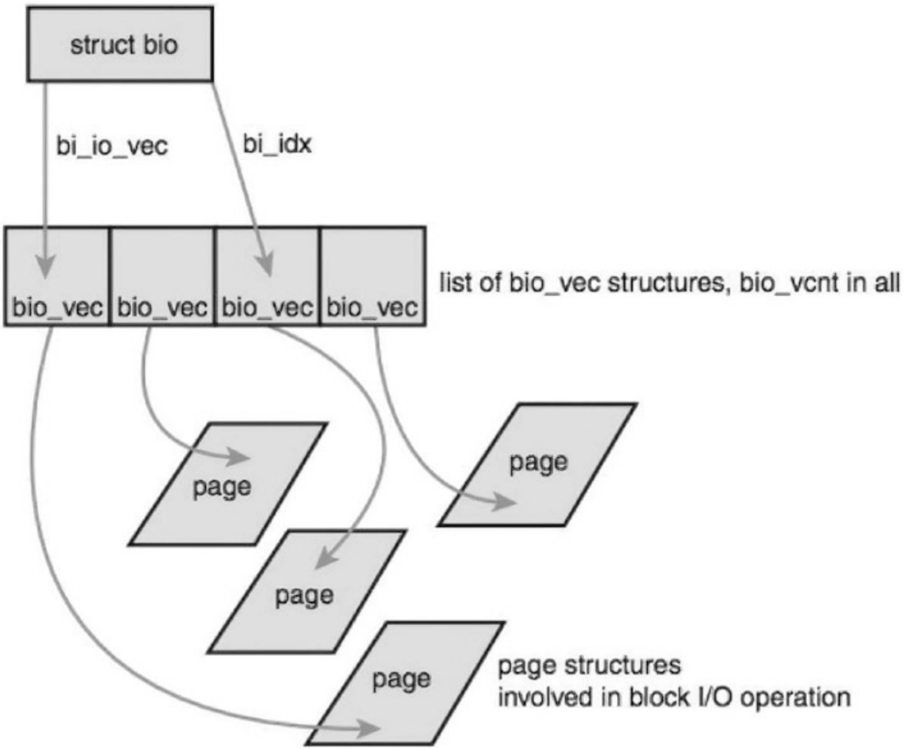


Figure 4-2. *The bio structure represents block I/O operations that are in flight (active) as a list of segments*

bio_vec represents a specific segment and has a pointer to the page holding the block data at a specific offset.

The requests are submitted to the request queue and drained by the device driver. The important data structures involved in implementing the block I/O cgroup within the Linux kernel are shown here:

```
struct blkcg {
   struct cgroup_subsys_state    css;
    spinlock_t                   lock;
   struct radix_tree_root        blkg_tree;
   struct blkcg_gq __rcu        *blkg_hint;
   struct hlist_head             blkg_list;
   struct blkcg_policy_data     *cpd[BLKCG_MAX_POLS];
   struct list_head all_blkcgs_node; #ifdef
   CONFIG_CGROUP_WRITEBACK
   struct list_head              cgwb_list;
   refcount_t                    cgwb_refcnt;
#endif
};
```

This structure represents the block I/O cgroup. Each block I/O cgroup is mapped to a request queue, which we explained previously.

```
/* association between a blk cgroup and a request queue */
struct blkcg_gq {
   /* Pointer to the associated request_queue */
   struct request_queue    *q;
   struct list_head         q_node;
   struct hlist_node        blkcg_node;
   struct blkcg            *blkcg;

/*
*  Each blkg gets congested separately and the congestion state is
```

```
*  propagated to the matching bdi_writeback_congested.
*/

   struct bdi_writeback_congested     *wb_congested;

   /* all non-root blkcg_gq's are guaranteed to have access to
      parent */
   struct blkcg_gq        *parent;

   /* request allocation list for this blkcg-q pair */
   struct request_list     rl;
   /* reference count */
   atomic_t               refcnt;
   /* is this blkg online? protected by both blkcg and q locks */

   Bool                        online;

   struct blkg_rwstat          stat_bytes;
   struct blkg_rwstat          stat_ios;

   struct blkg_policy_data   *pd[BLKCG_MAX_POLS];

   struct rcu_head            rcu_head;

atomic_t                      use_delay;
atomic64_t                    delay_nsec;
atomic64_t                    delay_start;
u64                           last_delay;
int                           last_use;
};
```

Each request queue is associated with a block I/O cgroup.

Understanding Fairness

By fairness, we mean that each cgroup should get a fair share of the I/O issued to the device. To accomplish this, a CFQ (Complete Fair Queuing) scheduler must be configured. Without cgroups in place, the CFQ scheduler assigns each process a queue and then gives a time slice to each queue, thereby handling fairness.

A *service tree* is a list of active queues/process on which the scheduler runs. So basically, the CFQ scheduler services requests from the queues on the service tree.

With cgroup in place, the concept of a *CFQ group* is introduced. Now, instead of scheduling per process, the scheduling happens at the group level. This means each cgroup has multiple service trees on which the group queues are scheduled. Then there is a global service tree on which the CFQ groups are scheduled.

The CFQ group structure is defined as follows:

```
struct cfq_group {
   /* must be the first member */
   struct blkg_policy_data pd;

   /* group service_tree member */
   struct rb_node rb_node;

   /* group service_tree key */
u64 vdisktime;

    /*
    * The number of active cfqgs and sum of their weights under this
    * cfqg. This covers this cfqg's leaf_weight and all children's
    * weights, but does not cover weights of further descendants.
    *
    * If a cfqg is on the service tree, it's active. An active cfqg
```

```
 * also activates its parent and contributes to the
   children_weight
 * of the parent.
 */
int nr_active;

unsigned int children_weight;

 /*
 * vfraction is the fraction of vdisktime that the tasks in this
 * cfqg are entitled to. This is determined by compounding the
 * ratios walking up from this cfqg to the root.
 *
 * It is in fixed point w/ CFQ_SERVICE_SHIFT and the sum of all
 * vfractions on a service tree is approximately 1. The sum may
 * deviate a bit due to rounding errors and fluctuations
   caused by
 * cfqgs entering and leaving the service tree.
 */
* unsigned int vfraction;

/*
 * There are two weights - (internal) weight is the weight
   of this
 * cfqg against the sibling cfqgs. leaf_weight is the weight of
 * this cfqg against the child cfqgs. For the root cfqg, both
 * weights are kept in sync for backward compatibility.
 */

unsigned int weight;
unsigned int new_weight;
unsigned int dev_weight;
```

```
unsigned int leaf_weight;
unsigned int new_leaf_weight;
unsigned int dev_leaf_weight;

/* number of cfqq currently on this group */
int nr_cfqq;
/*
 * Per group busy queues average. Useful for workload slice calc.
 * We create the array for each prio class but at runtime it
   is used
 * only for RT and BE class and slot for IDLE class remains unused.
 * This is primarily done to avoid confusion and a gcc warning.

*/

unsigned int
busy_queues_avg[CFQ_PRIO_NR]; /*
 *rr lists of queues with requests. We maintain service
  trees for
 *RT and BE classes. These trees are subdivided in subclasses
 * of SYNC, SYNC_NOIDLE and ASYNC based on workload type. For
 * the IDLE class there is no subclassification and all the
   CFQ queues go on
 * a single tree service_tree_idle.
 * Counts are embedded in the cfq_rb_root
 */

struct cfq_rb_root service_trees[2][3];
struct cfq_rb_root service_tree_idle;

u64 saved_wl_slice;
enum wl_type_t saved_wl_type;
enum wl_class_t saved_wl_class;
```

```
    /* number of requests that are on the dispatch list or
       inside driver */
   int dispatched;
   struct cfq_ttime ttime;
   struct cfqg_stats stats;      /* stats for this cfqg */

   /* async queue for each priority case */ struct
   cfq_queue *async_cfqq[2][IOPRIO_BE_NR]; struct
   cfq_queue *async_idle_cfqq;

};
```

Each CFQ group contains an "io weight" value that can be configured in cgroup. The CFQG's (CFQ groups) `vdisktime` decides its position on the "cfqg service tree," and then it's charged according to the "io weight,"

Understanding Throttling

Throttling provides a means to apply resource limits to the block I/O. This enables the kernel to control the max block I/O that a user space process can get. The kernel realizes this via the block I/O cgroup.

Throttling the block I/O per cgroup is done using a set of different functions. The first function is `blk_throttl_bio` and it's defined in `blk-throttle.c` (see https://elixir.bootlin.com/linux/latest/source/block/blk-throttle.c):

```
bool blk_throtl_bio(struct request_queue *q, struct blkcg_gq
            *blkg, struct bio *bio)
{
   struct throtl_qnode *qn = NULL;
   struct throtl_grp *tg = blkg_to_tg(blkg ?: q-
   >root_blkg); struct throtl_service_queue *sq; bool rw =
   bio_data_dir(bio);
```

```
bool throttled = false;
struct throtl_data *td = tg->td;
WARN_ON_ONCE(!rcu_read_lock_held());

/* see throtl_charge_bio() */
if (bio_flagged(bio, BIO_THROTTLED) || !tg->has_rules[rw])
goto out;

spin_lock_irq(q->queue_lock);


throtl_update_latency_buckets(td);


if (unlikely(blk_queue_bypass(q)))
goto out_unlock;

blk_throtl_assoc_bio(tg, bio);
blk_throtl_update_idletime(tg);

sq = &tg->service_queue;

again:

  while (true) {
     if (tg->last_low_overflow_time[rw] == 0) tg-
          >last_low_overflow_time[rw] = jiffies;
      throtl_downgrade_check(tg);
      throtl_upgrade_check(tg);
     /* throtl is FIFO - if bios are already queued, should queue
*/
     if (sq->nr_queued[rw])
         break;

     /* if above limits, break to queue */
     if (!tg_may_dispatch(tg, bio, NULL)) {
         tg->last_low_overflow_time[rw] = jiffies;
```

```
        if (throtl_can_upgrade(td, tg)) {
               throtl_upgrade_state(td);
            goto again;
        }
        break;
}

/* within limits, let's charge and dispatch directly */
throtl_charge_bio(tg, bio);
/*
 * We need to trim slice even when bios are not being queued
 * otherwise it might happen that a bio is not queued for
 * a long time and slice keep on extending and trim is not
 * called for a long time. Now if limits are reduced suddenly
 * we take into account all the IO dispatched so far at new
 * low rate and * newly queued IO gets a really long dispatch
 * time.
 *

 * So keep on trimming slice even if bio is not queued. */
throtl_trim_slice(tg, rw);

 /*
 * @bio passed through this layer without being throttled.
 * Climb up the ladder.  If we’re already at the top, it
 * can be executed directly.
 **/
qn = &tg->qnode_on_parent[rw];

sq = sq->parent_sq;
tg = sq_to_tg(sq);
if (!tg)
    goto out_unlock;
```

```
    }

    /* out-of-limit, queue to @tg */
    throtl_log(sq, "[%c] bio. bdisp=%llu sz=%u bps=%llu
  iodisp=%u iops=%u queued=%d/%d",

       rw == READ ? 'R' : 'W',
      tg->bytes_disp[rw], bio->bi_iter.bi_size,
       tg_bps_limit(tg, rw),
      tg->io_disp[rw], tg_iops_limit(tg, rw),
       sq->nr_queued[READ], sq->nr_queued[WRITE]);

    tg->last_low_overflow_time[rw] = jiffies;

    td->nr_queued[rw]++;
    throtl_add_bio_tg(bio, qn, tg);
    throttled = true;

      /*
      * Update @tg's dispatch time and force schedule dispatch
 if @tg
      * was empty before @bio. The forced scheduling isn't likely to
      * cause undue delay as @bio is likely to be dispatched
        directly if
      * @tg's disptime is not in the future.
      */
        if (tg->flags & THROTL_TG_WAS_EMPTY) {
            tg_update_disptime(tg);
            throtl_schedule_next_dispatch(tg->service_queue.
            parent_sq,
       true);
       }

     out_unlock:
```

```
      spin_unlock_irq(q->queue_lock);

   out:
     bio_set_flag(bio, BIO_THROTTLED);

   #ifdef CONFIG_BLK_DEV_THROTTLING_LOW
      if (throttled || !td->track_bio_latency) bio->
          bi_issue.value |= BIO_ISSUE_THROTL_SKIP_LATENCY;
     #endif

   return throttled;
   }
```

The following code snippet checks if the `bio` can be dispatched to be pushed to the device driver:

```
   if (!tg_may_dispatch(tg, bio, NULL)) { tg-
               >last_low_overflow_time[rw] = jiffies;
               if (throtl_can_upgrade(td, tg)) {
                       throtl_upgrade_state(td);
                   goto again;
               }

               break;
   }
```

The `tg_may_dispatch` definition is shown here:

```
   static bool tg_may_dispatch(struct throtl_grp *tg, struct
bio  *bio, unsigned long *wait)
{
    bool rw = bio_data_dir(bio);
    unsigned long bps_wait = 0, iops_wait = 0, max_wait = 0;

    /*
```

```
 * Currently the whole state machine of group depends on
   first bio
 * queued in the group bio list. So one should not be
   calling
 * this function with a different bio if there are other bios
 * queued.
 * /
 BUG_ON(tg->service_queue.nr_queued[rw] &&
   bio != throtl_peek_queued(&tg->service_queue.queued[rw]));

 /* If tg->bps = -1, then BW is unlimited */
    if (tg_bps_limit(tg, rw) == U64_MAX &&
         tg_iops_limit(tg, rw) == UINT_MAX) {
        if (wait)
             *wait = 0;
        return true;
   }

 /*
 * If the previous slice expired, start a new one, otherwise
 * renew/extend the existing slice to make sure it is at
   least throtl_slice interval
 * long since now. The new slice is started only for empty
   throttle
 * group. If there is queued bio, that means there should be an
 * active slice and it should be extended instead.
 * /

if (throtl_slice_used(tg, rw) &&
   !(tg->service_queue.nr_queued[rw]))
       throtl_start_new_slice(tg, rw);
   else {
        if (time_before(tg->slice_end[rw],
```

```
                jiffies + tg->td->throtl_slice))
                  throtl_extend_slice(tg, rw,
                    jiffies + tg->td->throtl_slice);
    }

    if (tg_with_in_bps_limit(tg, bio, &bps_wait) &&
        tg_with_in_iops_limit(tg, bio, &iops_wait)) {
         if (wait)
           *wait = 0;
        return true;
   }

   max_wait = max(bps_wait, iops_wait);

   if (wait)
         *wait = max_wait;

    if (time_before(tg->slice_end[rw], jiffies + max_wait))
       throtl_extend_slice(tg, rw, jiffies + max_wait);

    return false;

   The snippet
   if (tg_with_in_bps_limit(tg, bio, &bps_wait) &&
      tg_with_in_iops_limit(tg, bio, &iops_wait)) {
        if (wait)
         *wait = 0;
        return true;
}
```

This determines if the bio is within the limits for that cgroup or not. As evident, it checks both the bytes per sec limit as well as the I/O per sec limit for the cgroup.

If the limit is not exceeded, the bio is first charged to the cgroup:

```
/* within limits, let's charge and dispatch directly */ throtl_
charge_bio(tg, bio);

static void throtl_charge_bio(struct throtl_grp *tg, struct bio
*bio) {
   bool rw = bio_data_dir(bio);
   unsigned int bio_size =
                        throtl_bio_data_size (bio);

    /* Charge the bio to the group */
    tg->bytes_disp[rw] += bio_size;
    tg->io_disp[rw]++;
    tg->last_bytes_disp[rw] += bio_size;
    tg->last_io_disp[rw]++;

     /*
    * BIO_THROTTLED is used to prevent the same bio to be throttled
    * more than once as a throttled bio will go through
      blk-throtl the
    * second time when it eventually gets issued.  Set it when a bio
    * is being charged to a tg.
    */

     if (!bio_flagged(bio, BIO_THROTTLED))
          bio_set_flag(bio, BIO_THROTTLED);
     }
```

This function charges the bio (the bytes and iops) to the throttle group. It then passes the bio up to the parent, as evident in the following code:

```
/*
     * @bio passed through this layer without being throttled.
     * Climb up the ladder.  If we're already at the top, it
```

```
     * can be executed directly.
     */
     qn = &tg->qnode_on_parent[rw];
     sq = sq->parent_sq;
     tg = sq_to_tg(sq);
```

If the limits are exceeded, the code takes a different flow. The following code snippet is called:

```
    throtl_add_bio_tg(bio, qn, tg);
    throttled = true;
```

Let's look at the `throtl_add_bio_tg` function in more detail:

```
     /**
     * throtl_add_bio_tg - add a bio to the specified throtl_grp
     * @bio: bio to add
     * @qn: qnode to use
     * @tg: the target throtl_grp

   *

    Add @bio to @tg's service_queue using @qn. If @qn is not
    specified,
    tg->qnode_on_self[] is used.
    */

static void throtl_add_bio_tg(struct bio *bio, struct
throtl_qnode *qn,
             struct throtl_grp *tg)
{

    struct throtl_service_queue *sq = &tg-
    >service_queue; bool rw = bio_data_dir(bio);
```

```
 if (!qn)
       qn = &tg->qnode_on_self[rw];

  /*
  * If @tg doesn't currently have any bios queued in the same
  * direction, queueing @bio can change when @tg should be
  * dispatched.  Mark that @tg was empty.  This is automatically
  * cleared on the next tg_update_disptime().

*/
if (!sq->nr_queued[rw])
    tg->flags |= THROTL_TG_WAS_EMPTY;

throtl_qnode_add_bio(bio, qn, &sq->queued[rw]);


sq->nr_queued[rw]++;
throtl_enqueue_tg(tg);
}
```

This function adds the bio to the throttle service queue. This queue acts as a mechanism to throttle the bio requests. The service request is then drained later.

```
/**

  *blk_throtl_drain - drain throttled bios
  *@q: request_queue to drain throttled bios for

  *Dispatch all currently throttled bios on @q through
  - >make_request_fn().

 */

void blk_throtl_drain(struct request_queue *q)
__releases(q->queue_lock) __acquires(q->queue_lock)
{
```

```
   struct throtl_data *td = q-
   >td; struct blkcg_gq *blkg;

   struct cgroup_subsys_state *pos_css;
   struct bio *bio;
   int rw;

   queue_lockdep_assert_held(q);

   rcu_read_lock();

     /*
     * Drain each tg while doing post-order walk on the blkg tree, so
     * that all bios are propagated to td->service_queue.  It'd be
     * better to walk service_queue tree directly but blkg walk is
     * easier.
    */

   blkg_for_each_descendant_post(blkg, pos_css, td->queue-
   >root_blkg)
      tg_drain_bios(&blkg_to_tg(blkg)->service_queue);

    /* finally, transfer bios from top-level tg's into the td */
    tg_drain_bios(&td->service_queue);

    rcu_read_unlock();
    spin_unlock_irq(q->queue_lock);

   /* all bios now should be in td->service_queue, issue them
   */ for (rw = READ; rw <= WRITE; rw++)
          while ((bio = throtl_pop_queued(&td-
>service_queue.queued[rw],
                                     NULL)))
                 generic_make_request(bio);
    spin_lock_irq(q->queue_lock);
```

CHAPTER 5

Layered File Systems

In the previous chapters, we learned about namespaces and cgroups. In this chapter, we touch upon another interesting aspect of the container ecosystem, which is the layered file system. We discuss how it enables file-sharing on the host and how this helps run multiple containers on the host.

In previous chapters, we addressed topics of process isolation via Linux namespaces and resource control for individual processes via cgroups. Now we delve into the topic of layered file systems, which constitute the third building block of the Linux container, after namespaces and cgroups.

Lets start by discussing what a file system is.

A File System Primer

The Linux philosophy is to treat everything as a file. As an example, socket, pipe, and block devices are all represented as files in Linux.

The file systems in Linux act as containers to abstract the underlying storage in the case of block devices. For non-block devices like sockets and pipes, there are file systems in memory that have operations which can be invoked using the standard file system API.

Linux abstracts all file systems using a layer called the Virtual File System (VFS). All file systems register with the VFS. The VFS has the following important data structures:

S. M. Jain, *Linux Containers and Virtualization*, https://doi.org/10.1007/978-1-4842-6283-2_5

- **File**: This represents the open file and captures the information, like offset, and so on. The user space has a handle to an opened file via a structure called the file descriptor. This is the handle used to interface with the file system.
- **Inode**: This is mapped 1:1 to the file. The *inode* is one of the most critical structures and holds the metadata about the file. As an example, it includes in which data blocks the file data is stored and which access permissions are on the file. This info is part of the inode. Inodes are also stored on disk by the specific file system, but there is a representation in memory that's part of the VFS layer. The file system is responsible for enumerating the VFS inode structure.
- **Dentry**: This is the mapping between filename and inode. This is an in-memory structure and is not stored on disk. This is mainly relevant to lookup and path traversal.
- **Superblock**: This structure holds all the information about the file system, including how many blocks are there, the device name, and so on. This structure is enumerated and brought into memory during a mount operation.

Each of these data structures holds pointers to their specific operations. As an example, *file* has `file_ops` for reading and writing and *superblock* has operations via `super_ops` to mount, unmount, and so on.

The mount operation creates a `vfsmount` data structure, which holds a reference to a new superblock structure created from the file system to be mounted on the disk. The dentry has a reference to the `vfsmount`. This is where the VFS distinguishes between a directory and a mount point. During a traversal, the `vfsmount` is found in a dentry, the inode number 2 on the mounted device is used (inode 2 is reserved for the root directory).

So how does this all fit together in the case of a block device? Say that the user space process makes a call to read a file. The system call is made to the kernel. The VFS checks the path and determines if there are dentries cached from the root. As it traverses and finds the right dentry, it locates the inode for the file to be opened. Once the inode is located, the permissions are checked and the data blocks are loaded from the disk into the OS page cache. The same data is moved into the user space of the process.

The page cache is an interesting optimization in the OS. All reads and writes (except direct I/O) happen over the page cache. The page cache itself is represented by a data structure called the `address_space`. This `address_space` holds a tree of memory pages and the file inode holds a reference to that `address_space` data structure.

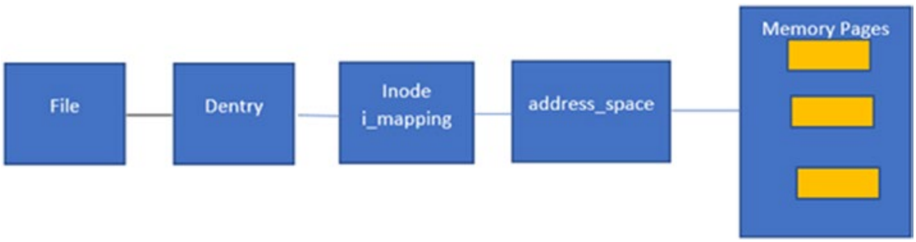


Figure 5-1. *Mapping a file to a page cache*

Figure 5-1 shows how a file maps into the page cache. This is also the key to understanding how operations like `mmap` for memory mapped files work. We will cover that when we cover file systems like `tmpfs` and shared memory IPC primitives.

If the file read request is in the page cache (which is determined via the `address_space` structure of the file's inode), the data is served from there.

Whenever a write call is made on the file via the file descriptor, the writes are first written to the page cache. The memory pages are marked dirty and the Linux kernel uses the write-back cache mechanism, which means there are threads in the background (called `pdflush`) that drain the page cache and write to the physical disk via the block driver. The mechanism of marking pages dirty doesn't happen at the page level. Pages can be 4KB in size and even a minimal change will then cause a full page write.

To avoid that, there are structures that have more fine-grained granularity and represent a disk block in memory. These structures are called *buffer heads*. For example, if the block size is 512 bytes, there are eight buffer heads and one page in the page cache.

That way, individual blocks can be marked dirty and made part of the writes.

The buffers can be explicitly flushed to disk via these system calls:

- `Sync()`: Flushes all dirty buffers to disk.
- `Fsync(fd)`: Flushes only the file-specific dirty buffers to disk, including the changes to inode.
- `Fdatasync(fd)`: Flushes only the dirty data buffers of the file to disk. Doesn't flush the inodes.

Here's an example of how this sync process works:

1. Check if the superblock is dirty.
2. Write back the superblock.
3. Iterate over each inode from the inode list:
    a. If the inode is dirty, write it back.
    b. If the page cache of the inode is dirty, write it back.
    c. Clear the dirty flag.

Figure 5-2 shows the file system's different layers under the kernel.

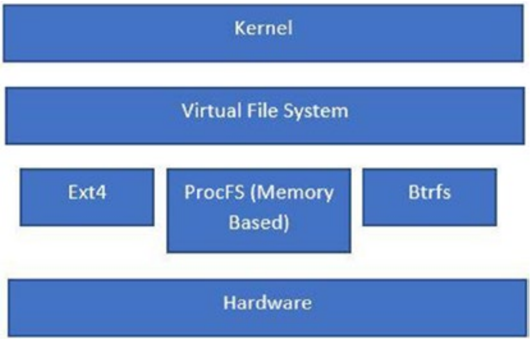


Figure 5-2. *The different layers of a file system under the kernel*

Examples of different kinds of file systems include:

- **Ext4**: This file system is used to access the underlying block devices.
- **ProcFS**: This is an in-memory file system and is used to provide features. This is also called a *pseudo file system*.

A Few Words on Pseudo File Systems

Recall that the general philosophy of Linux is that everything is a file. Working on that premise, there are file systems that expose some of the kernel's resources over the file interface. We call them pseudo file systems. One such file system is `procfs`.

The `procfs` file system is mounted on the `rootfs` under the `proc` directory. The data under `procfs` is not persisted and all operations happen in memory.

Some of the structures exposed via `procfs` are explained in the following table:

Structure	Description
/proc/cpuinfo	CPU details like cores, CPU size, make, etc.
/proc/meminfo	Information about physical memory
/proc/interrupts	Information about interrupts and handlers
/proc/vmstat	Virtual memory stats
/proc/filesystems	Active file systems on the kernel
/proc/mounts	Current mounts and devices; this will be specific to the mount namespace
/proc/uptime	Time since the kernel was up
/proc/stat	System statistics
/proc/net	Network-related structures like TCP sockets, files, etc. proc also exposes some process-specific information via files
/proc/pid/cmdline	Command-line name of the process
/proc/pid/environ	Environment variables of the process
/proc/pid/mem	Virtual memory of the process
/proc/pid/maps	Mapping of the virtual memory
/proc/pid/fdinfo	Open file descriptors of the process
/proc/pid/task	Details of the child processes

Layered File Systems

Now that you have a better understanding of the file systems in Linux, it's time to take a look at the layered file systems in Linux.

The layered file system allows files to be shared on disk, thereby saving space. Since these files are shared in memory (loaded in page cache), a layered file system allows optimal space utilization as well as faster bootup.

Consider an example of running ten Cassandra databases on the same host, each database running its own namespaces. If we have separate file systems for each database's different inodes, we don't enjoy these advantages:

- Memory sharing
- Sharing on disk

Whereas in the case of a layered file system, the file system is broken into layers and each layer is a read-only file system. Since these layers are shared across the containers on the same host, they tend to use storage optimally. And, since the inodes are the same, they refer to the same OS page cache. This makes things optimal from all aspects.

Compare this to VM-based provisioning, where each `rootfs` is provisioned as a disk. This means they all have different inode representations on the host and there is no optimal storage as compared to the containers.

Hypervisors also tend to reach optimization using techniques like KSM (Kernel Same Page Merging) so they can de-duplicate across VMs for the same pages.

Next, we discuss the concept of union file systems, which is a type of layered file system.

The Union File System

According to Wikipedia, the union file system is a file system service for Linux, FreeBSD, and NetBSD that implements a union mount for other file systems. It allows files and directories of separate file systems, known as *branches*, to be transparently overlaid, forming a single coherent file system. The contents of any directories that have the same path within the merged branches will be seen together in a single merged directory, within the new virtual file system.

So basically, a union file system allows you to take different file systems and create a union of their contents, with the top layer providing a view of all the files underlying it. If duplicate files are found, the top layer supersedes the layers below it.

OverlayFS

This section looks at OverlayFS as one example of a union FS. OverlayFS has been part of the Linux Kernel since 3.18. It overlays (as the name suggests) the contents of one directory onto other. The source directories can be on different disks or file systems.

With OverlayFS v1, there were only two layers, and they were used to create a unified layer, as shown in Figure 5-3.

Figure 5-3. *OverlayFS v1 with two layers (upper and lower)*

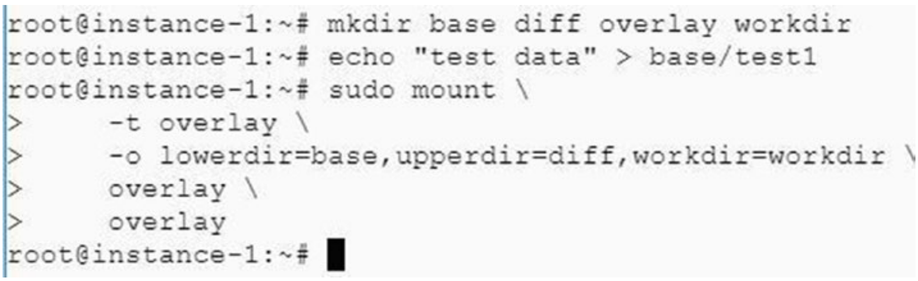
OverlayFS v2 has three layers:

- **Base**: This is the base layer. This is primarily read-only.
- **Overlay**: This layer provides visibility from the base layer and allows one to add new files/directories. If any files from the base layer change, they are stored in the next layer.

- **Diff**: The changes made in the overlay layer are stored in the diff layer. Any changes to files in the base layer lead to copying the file from the base layer to the diff layer. The changes are then written in the diff layer.

Lets look at an example of how OverlayFS v2 works:

```
root@instance-1: mkdir base diff overlay workdir
root@instance-1: echo "test data" > base/test1
root@instance-1: sudo mount \
>     -t overlay \
>     -o lowerdir=base,upperdir=diff,workdir=workdir \
>     overlay \
>     overlay
root@instance-1:-# I
```



```
root@instance-1:~# mkdir base diff overlay workdir
root@instance-1:~# echo "test data" > base/test1
root@instance-1:~# sudo mount \
>     -t overlay \
>     -o lowerdir=base,upperdir=diff,workdir=workdir \
>     overlay \
>     overlay
root@instance-1:~#
```

We create a file in the overlay directory and can see that it appears in diff:

```
root@instance-1:/overlay# touch test2
root@instance-1:/overlay# ls
test1 test2
root@instance-1:-/overlay# cd ../diff
root@instance-1:-/diff# ls
test2
```

```
root@instance-1:~/overlay# touch test2
root@instance-1:~/overlay# ls
test1  test2
root@instance-1:~/overlay# cd ../dif
-bash: cd: ../dif: No such file or directory
root@instance-1:~/overlay# cd ../diff
root@instance-1:~/diff# ls
test2
```

We now modify the `test1` file:

```
root@instance-1:~/diff# nano test1
root@instance-1:~/diff# cd ../overlay/
root@instance-1:~/overlay# nano test1
root@instance-1:~/overlay# cat test1
test data
. Modifying
root@instance-1:~/overlay# cd ../base
root@instance-1:~/base# cat test1
test data
root@instance-1:~/base#
```

If we check the file in the `diff` directory, we see the changed file. However, if we go to the `base` directory, we still see the old file. This means that when we modified the file in the `base` directory, it was copied to the `diff` directory first, after which the changes were made.

After these examples are executed, if users wanted to do a cleanup of resources, they could execute the following command to unmount the OverlayFS:

```
root@instance-1: umount overlay
```

After the unmount is complete, the directories can also be removed if desired.

Lets now think about how container engines like Docker implement this process. There is an Overlay2 storage driver in Docker, which you can find out more about at `https://github.com/moby/moby/blob/master/daemon/graphdriver/overlay2/overlay.go`.

Docker creates multiple read layers (base layers) and one read/write layer called the container layer (in our case, the overlay layer).

The multiple read layers can be shared across different containers on the same host, thereby attaining very high optimization. As hinted at earlier, since we have the same file system and the same inodes, the OS page cache is also shared across all containers on the same host.

Contrary to this, if we see a Docker driver device mapper, since it gives a virtual disk for each layer, we might not experience the sharing we get with OverlayFS .But now, even with the device mapper usage in Docker, we can pass the `-shared-rootfs` option to the daemon to share the `rootfs`. This basically works by creating a device for the first container base image and then doing bind mounts for subsequent containers. The bind mounts allow us to preserve the same inodes and therefore the page cache is shared.

CHAPTER 6

Creating a Simple Container Framework

In the previous chapters, we learned about the important building blocks of the container framework, like namespaces, cgroups, and layered file systems. In this chapter, we use that knowledge to build a simple container framework and learn how these building blocks make up the container framework.

Since we have covered the basics of what constitutes a container, it is time to look at how to write your own simple container. By end of this chapter, you will have created your own simple container using namespace isolation.

Let's get started.

I have tested the commands mentioned in the chapter on Ubuntu 19.04 with Linux Kernel 5.0.0-13.

The first command we explore is called `unshare`. This command allows you to unshare a set of namespaces from the host.

The UTS Namespace

We will enter a new `uts` namespace and change the hostname within that namespace.

S. M. Jain, *Linux Containers and Virtualization*, https://doi.org/10.1007/978-1-4842-6283-2_6

```
root@osboxes:~# unshare -u /bin/bash
root@osboxes:~# hostname test
root@osboxes:~# hostname
test
root@osboxes:~# exit
exit
root@osboxes:~# hostname
osboxes
```

When we entered the UTS namespace, we changed the hostname to `test` and this is what is reflected within that namespace. Once we exit and re-enter the `host` namespace, we get the `host` namespace.

The command `unshare -u /bin/bash` creates the `uts` namespace and executes our process (`/bin/bash`) within that namespace. The careful reader might observe that if we don't change the hostname after entering the namespace, we still get the hostname of the host. This is not desirable, as we need a way to set this before executing our program within the namespace.

This is where we will explore writing a container using Golang (also called Go) and then set up namespaces before we launch the process within the container. We will be writing the container in Golang, so we need to have Golang installed on the VM or on the machine on which we are working. (For Golang installation, visit https://golang.org/doc/install.)

Golang is the most common systems programming language around. It is used to create container runtimes like Docker, as well as container orchestration engines like Swarm and Kubernetes. Apart from that, it has been used in various other systems programming settings. It's a good idea to have a decent understanding of Golang before you delve into the code in this chapter.

Golang Installation

Here are the quick Golang install commands:

root@osboxes:~#wget https://dl.google.com/go/go1.12.7.linux-amd64.tar.gz

root@osboxes:~# `tar -C /usr/local -xzf go1.12.7.linux-amd64.tar.gz`

You can add the following line to `/root/.profile` to add the Golang binaries to the system `PATH` variable:

root@osboxes:~# `export PATH=$PATH:/usr/local/go/bin`

Then run this command in your terminal:

root@osboxes:~# source ~/.profile

To check if Go (Golang) is installed properly, you can run this command:

root@osboxes:~# go version

If the installation was successful, you should see the following output:

Now we will build a container with only a namespace and then keep modifying the program to add more functionalities, like shell support, `rootfs`, networking, and cgroups.

Building a Container with a Namespace

Let's revisit Linux namespaces briefly before we build the container. Namespaces are in the Linux kernel, similar to sandbox kernel resources like file systems, process trees, message queues, and semaphores, as well as network components like devices, sockets, and routing rules.

Namespaces isolate processes within their own execution sandbox so that they run completely isolated from other processes in different namespaces.

There are six namespaces:

- **PID namespace**: The processes within the PID namespace have a different process tree. They have an `init` process with a PID of 1.
- **Mount namespace**: This namespace controls which mount points a process can see. If a process is within a namespace, it will only see the mounts within that namespace.
- **UTS namespace**: This allows a process to see a different namespace than the actual global namespace.
- **Network namespace**: This namespace gives a different network view within a namespace. Network constructs like ports, iptables, and so on, are scoped within the namespace.
- **IPC namespace**: This namespace confines interprocess communication structures like pipes within a specific namespace.
- **User-namespace**: This namespace allows for a separate user and group view within the namespace.

We don't discuss the cgroup namespace here, which also allows the cgroups to be scoped into their own namespaces.

Now let's get our hands dirty and create a Go class called `myuts.go`. Copy the following snippet and use `go build myuts.go` to get the `myuts` binary. Also execute the **myuts** binary as the root user.

```
package main

import (

  "fmt"
  "os"
  "os/exec"
  "syscall"

)

func main() {

  cmd := exec.Command("/bin/bash")
  // The statements below refer to the input, output and error
streams of the process created (cmd)
  cmd.Stdin = os.Stdin
  cmd.Stdout = os.Stdout
  cmd.Stderr = os.Stderr
  //setting an environment variable
  cmd.Env = []string{"name=shashank"}
  // the command below creates a UTS namespace for the process
        cmd.SysProcAttr = &syscall.SysProcAttr{
              Cloneflags: syscall.CLONE_NEWUTS,
  }
```

```
  if err := cmd.Run(); err != nil {
        fmt.Printf("Error running the /bin/bash command - %s\n", err)
        os.Exit(1)
  }
}
```

This is a simple Go program that executes a shell, sets up the I/O streams for the process, and then sets one `env` variable. Then it uses the following command:

```
cmd.SysProcAttr = &syscall.SysProcAttr{
        Cloneflags: syscall.CLONE_NEWUTS,
  }
```

It then passes the `CLONE` flags (in this case, we just pass UTS as the Clone flag). The clone flags control which namespaces are created for the process.

After that, we build and run this Golang process. We can see whether the new namespace was created by using the `proc` file system and checking the `proc/<<pid>>/ns`:

```
root@osboxes:~/book_prep# ls -li /proc/self/ns/uts
60086 lrwxrwxrwx 1 root root 0 Apr 13 10:10 /proc/self/ns/uts -
> 'uts:[4026531838]'
root@osboxes:~/book_prep# ./myuts
root@osboxes:/root/book_prep# ls -li /proc/self/ns/uts
60099 lrwxrwxrwx 1 root root 0 Apr 13 10:10 /proc/self/ns/uts -
> 'uts:[4026532505]'
root@osboxes:/root/book_prep#exit
```

First, we print the namespace of the host and then we print the namespace of the container we are in.

We can see that the `uts` namespaces are different.

Adding More Namespaces

In the previous section, we displayed how a UTS namespace could be created. In this section, we add more namespaces.

First, we add more clone flags, in order to create more namespaces for the container we are creating.

```
package main

import (
    "fmt"
    "os"
    "os/exec"
    "syscall"
)

func main() {
    cmd := exec.Command("/bin/bash")

    cmd.Stdin = os.Stdin
    cmd.Stdout = os.Stdout
    cmd.Stderr = os.Stderr

    cmd.Env = []string{"name=shashank"}
    //command below creates the UTS, PID and IPC , NETWORK and
      USERNAMESPACES
    cmd.SysProcAttr = &syscall.SysProcAttr{
            Cloneflags: syscall.CLONE_NEWNS |
                    syscall.CLONE_NEWUTS |
                    syscall.CLONE_NEWIPC |
                    syscall.CLONE_NEWPID |
                    syscall.CLONE_NEWNET |
                    syscall.CLONE_NEWUSER,
    }
```

```
    if err := cmd.Run(); err != nil {
      fmt.Printf("Error running the /bin/bash command - %s\n", err)
      os.Exit(1)
    }
}
```

Here we added more namespaces via the clone flag. We build and run the program as follows:

```
root@osboxes:~/book_prep# ./myuts
nobody@osboxes:/root/book_prep$ ls -li /proc/self/ns/ total 0
63290 lrwxrwxrwx 1 nobody nogroup 0 Apr 13 10:14 cgroup ->
'cgroup:[4026531835]'
63285 lrwxrwxrwx 1 nobody nogroup 0 Apr 13 10:14 ipc ->
'ipc:[4026532508]'
63289 lrwxrwxrwx 1 nobody nogroup 0 Apr 13 10:14 mnt ->
'mnt:[4026532506]'
63283 lrwxrwxrwx 1 nobody nogroup 0 Apr 13 10:14 net ->
'net:[4026532511]'
63286 lrwxrwxrwx 1 nobody nogroup 0 Apr 13 10:14 pid ->
'pid:[4026532509]'
63287 lrwxrwxrwx 1 nobody nogroup 0 Apr 13 10:14 pid_for_
children -> 'pid:[4026532509]'
63288 lrwxrwxrwx 1 nobody nogroup 0 Apr 13 10:14 user ->
'user:[4026532505]'
63284 lrwxrwxrwx 1 nobody nogroup 0 Apr 13 10:14 uts ->
'uts:[4026532507]'
```

We have the namespaces this container belongs to. Now we see that the ownership belongs to nobody. This is because we also used a user-namespace as a clone flag. The container is now within a new user-namespace. User-namespaces require that we map the user from the namespace to the host. Since we have not done anything yet, we still see nobody as the user.

We now add user mapping to the code:

```
package main

import (
        "fmt"
        "os"
        "os/exec"
        "syscall"
)

func main() {
        cmd := exec.Command("/bin/bash")

        cmd.Stdin = os.Stdin
        cmd.Stdout = os.Stdout
        cmd.Stderr = os.Stderr

        cmd.Env = []string{"name=shashank"}
    //command below creates the UTS, PID and IPC , NETWORK and
    // USERNAMESPACES and adds the user and group mappings.

        cmd.SysProcAttr = &syscall.SysProcAttr{
                Cloneflags: syscall.CLONE_NEWNS |
                        syscall.CLONE_NEWUTS |
                        syscall.CLONE_NEWIPC |
                        syscall.CLONE_NEWPID |
                        syscall.CLONE_NEWNET |
                        syscall.CLONE_NEWUSER,
        UidMappings: []syscall.SysProcIDMap{
                        {
                                ContainerID: 0,
                                HostID: os.Getuid(),
                                Size: 1,
                        },
```

```
                },
                GidMappings: []syscall.SysProcIDMap{
                        {
                                ContainerID: 0,
                                HostID: os.Getgid(),
                                Size: 1,
                        },
                },
        }
        if err := cmd.Run(); err != nil {
        fmt.Printf("Error running the /bin/bash command -
        %s\n", err)
        os.Exit(1)
        }
}
```

You can see that we have `UidMappings` and `GidMappings`. We have a field called `ContainerID`, which we are setting to `0`. This means we are mapping the `uid` and `gid 0` within the container to the `uid` and `gid` of the user who launched the process.

There is one interesting aspect I would like to touch upon in the context of user-namespaces. We don't need to be the root on the host in order to create a user-namespace. This provides a way to create namespaces and thereby containers without being the root on the machine, which means it's a big security win as providing root access to a process can be hazardous. If programs are launched as the root, any compromise to those programs can give root privileges to the attacker. In turn, the whole machine gets compromised.

We can technically be non-root on the host and then create a user-namespace and other namespaces within that user-namespace. Mind you, all the other namespaces, if launched without a user-namespace, will need root access.

If we take the previous example, where we are passing all the flags together, the system first creates a user-namespace and places all the other namespaces within that user-namespace.

I cannot cover the user-namespace topic in its entirety here, but it is an interesting area for curious readers to explore. One area I can mention straightaway is the area of Docker builds, wherein we need root access to build an image within a container. This is necessary for many reasons, as we need some layered file systems mounted within the container and creating a new mount requires root privilege.

The same holds for setting up virtual network devices like veth pairs in order to wire containers to the host. Having said that, there has been momentum in the area of *rootless* containers, which allow developers to run containers without the root. If you want to read about this in more detail, you can explore this topic at the following: `https://rootlesscontaine.rs/` and `https://github.com/rootless-containers`.

What we have achieved thus far is the ability to launch a process within a set of namespaces. But we definitely need more. We need a way to initialize these namespaces before we launch the container.

Back to the program we created. Let's build and run it:

```
root@osboxes:~/book_prep# ./myuts
root@osboxes:/root/book_prep# whoami
root

root@osboxes:/root/book_prep# id
uid=0(root) gid=0(root) groups=0(root)
```

Now we see that the user within the container is the root.

The program checks the first argument. If the first command is run, then the program executes `/proc/self/exe`, which is simply saying execute yourself (`/proc/self/exe` is the copy of the binary image of the caller itself).

One might ask why we need to execute `/proc/self/exe`. When we execute this command, it launches the same binary with some arguments (in our case, we pass `fork` as the argument to it). Once we are into different namespaces, we need some setup for the namespaces, like setting the hostname, before we launch the process within the container.

Executing `/proc/self/exe` gives us the opportunity to set up the namespaces like so:

1. Set the hostname.
2. Within the mount namespace, we do a pivot root, which allows us to switch the root file system. It does this by copying the old root to some other directory and making the new path the new root. This pivot root has to be done from within the mount namespace, as we don't want to move the `rootfs` off the host. We also mount the `proc` file system. This is done because the mount namespace inherits the `proc` of the host and we want a `proc` mount within the mount namespace.
3. Once the namespaces are initialized and set up, we invoke the container process (in this case, the shell).

Running this program launches the shell into a sandbox confined by the `proc`, mount, and `uts` namespace.

Now we work on initializing the namespaces before launching the process within the container. In the following example, we will have a different hostname in the `uts` namespace. In the following code, we make the required changes.

We have a function parent that:

1. Clones the namespaces.
2. Launches the same process again via `/proc/self/exe` and passes a child as the parameter.

Now the process is called again. Checks in the main function lead to invocations of the child function. Now you can see that we cloned the namespaces. All we do now is change the hostname to `myhost` within the `uts` namespace. Once that is done, we invoke the binary passed as the command-line parameter (in this case, `/bin/bash`).

Launching a Shell Program Within the Container

In previous sections, we explained how to create different Linux namespaces. In this section, we explain how to enter those namespaces. Entering the confines of the namespaces can be done by launching a program/process within the namespaces. The following program launches a shell program within these namespaces.

```
package main

import (
        "fmt"
        "os"
        "os/exec"
        "syscall"
)

func main() {

switch os.Args[1] {
        case "parent":
                parent()
        case "child":
                child()
        default:
               panic("help")
```

```
        }
}
// the parent function invoked from the main program which sets
up the needed namespaces
func parent() {

        cmd := exec.Command("/proc/self/exe",
append([]string{"child"}, os.Args[2:]...)...)

        cmd.Stdin = os.Stdin
        cmd.Stdout = os.Stdout
        cmd.Stderr = os.Stderr

        cmd.Env = []string{"name=shashank"}

        cmd.SysProcAttr = &syscall.SysProcAttr{
                Cloneflags: syscall.CLONE_NEWNS |
                        syscall.CLONE_NEWUTS |
                        syscall.CLONE_NEWIPC |
                        syscall.CLONE_NEWPID |
                        syscall.CLONE_NEWNET |
                        syscall.CLONE_NEWUSER,
        UidMappings: []syscall.SysProcIDMap{
                        {
                                ContainerID: 0,
                                HostID: os.Getuid(),
                                Size: 1,
                        },
               },
               GidMappings: []syscall.SysProcIDMap{
                      {
                                ContainerID: 0,
                                HostID: os.Getgid(),
```

```
                            Size: 1,
                    },
                },
        }
        must(cmd.Run())
}
// this is the child process which is a copy of the parent
   program itself.
func child () {
cmd := exec.Command(os.Args[2], os.Args[3:]...)
        cmd.Stdin = os.Stdin
        cmd.Stdout = os.Stdout
        cmd.Stderr = os.Stderr
//the command below sets the hostname to myhost. Idea here is
  to showcase the use of UTS namespace
must(syscall.Sethostname([]byte("myhost")))
// this command executes the shell which is passed as a program
   argument
must(cmd.Run())
}
func must(err error) {
        if err != nil {
                fmt.Printf("Error - %s\n", err)
        }
}
```

Upon executing the program, we can launch the binary within the new namespaces. Also note that the hostname is set to `myhost`:

```
root@osboxes:~/book_prep# ./myuts parent /bin/bash
root@myhost:/root/book_prep# hostname
myhost
root@myhost:/root/book_prep#
```

After the `uts` namespace, it's time to get more adventurous. We now will work on initializing the mount namespace.

One thing to understand here is that all mounts from the host are inherited within the mount namespace. Therefore, we need a mechanism to clear the mounts and only make mounts for the mount namespace visible within that namespace.

Before we move ahead, one of the things to understand conceptually is the system call `pivot_root`. This system call allows us to change the root file system for the process. It mounts the old root to some other directory (in the following example, the author used `pivot_root` as the directory to mount the old root on) `else` and mounts the new directory on `/`. This allows us to clear all the host mounts within the namespace.

Again, we need to be inside the mount namespace before we do the `pivot_root`. Since we already have a hook on namespace initialization (via the `/proc/self/exe` hack), we need to introduce a pivot root mechanism.

Providing Root File System

We will use the `rootfs` from busybox, which you can download from https://github.com/allthingssecurity/containerbook (`busybox.tar`).

After downloading `busybox.tar`, extract it to `/root/book_prep/rootfs` in your system. This location is referred to in this code as the location of `rootfs`. As shown in Figure 6-1, the contents of the `/root/book_prep/rootfs` should look the same on your system.

```
root@osboxes:~/book_prep/rootfs# ls -l
total 48
drwxr-xr-x 2 root root 12288 Jun 23  2016 bin
drwxr-xr-x 2 sys  sys   4096 Jun 23  2016 dev
drwxr-xr-x 2 root root  4096 Jun 23  2016 etc
drwxr-xr-x 2   99   99  4096 Jun 23  2016 home
drwxr-xr-x 2 root root  4096 Jun 23  2016 lib
lrwxrwxrwx 1 root root     3 Jun 23  2016 lib64 -> lib
drwxr-xr-x 2 root root  4096 Jul 11 07:55 proc
drwxr-xr-x 2 root root  4096 Jun 23  2016 root
drwxrwxrwt 2 root root  4096 Jun 23  2016 tmp
drwxr-xr-x 3 root root  4096 Jun 23  2016 usr
drwxr-xr-x 4 root root  4096 Jun 23  2016 var
root@osboxes:~/book_prep/rootfs#
```

Figure 6-1. *The contents of the /root/book_prep/rootfs path*

After extracting the `rootfs`, we can see the directory structure under the `rootfs` directory.

```
root@osboxes:~/book_prep/rootfs# ls
bin   dev   etc   home   lib   lib64   root   tmp   usr   var
root@osboxes:~/book_prep/rootfs# cd ..
root@osboxes:~/book_prep# |
```

The following program does a pivot root to the `rootfs` within the mount namespace.

The mount namespace becomes important, as it allows us to sandbox the file system mounts. This is one way to get an isolated view of the file system hierarchy and see what is present on the host or on different sandboxes running on the same host.

As an example, assume there are two sandboxes—`sandboxA` and `sandboxB`—running on the host. When `sandboxA` gets its own mounts, its file system sees a different and isolated mount from what `sandboxB` sees, and neither can see the mounts of the host. This provides security at the file system level, as individual sandboxes cannot access files from different sandboxes or from the host.

```
//providing rootfile system
package main

import (
        "fmt"
        "os"
        "os/exec"
        "path/filepath"
        "syscall"
)

func main() {

switch os.Args[1] {
        case "parent":
                parent()
        case "child":
                child()
        default:
               panic("help")
        }
}

func pivotRoot(newroot string) error {
        putold := filepath.Join(newroot, "/.pivot_root")

        //bind mount newroot to itself - this is a slight hack
          needed to satisfy the
        //pivot_root requirement that newroot and putold must
          not be on the same
        //filesystem as the current root
        if err := syscall.Mount(newroot, newroot, "", syscall.
        MS_BIND|syscall.MS_REC, ""); err != nil {
                 return err
```

```
        }

        // create putold directory
        if err := os.MkdirAll(putold, 0700); err != nil
                { return err
        }

        // call pivot_root
        if err := syscall.PivotRoot(newroot, putold); err != nil {
           return err
        }

        // ensure current working directory is set to new
        root if err := os.Chdir("/"); err != nil {
                return err
        }

        //umount putold, which now lives at /.pivot_root putold
          = "/.pivot_root"
        if err := syscall.Unmount(putold, syscall.MNT_DETACH);
        err !=
        nil {
                return err
        }

        // remove putold
        if err := os.RemoveAll(putold); err != nil
                { return err
        }

        return nil
}
```

```
func parent() {

        cmd := exec.Command("/proc/self/exe", append([]
        string{"child"}, os.Args[2:]...)...)

        cmd.Stdin = os.Stdin
        cmd.Stdout = os.Stdout
        cmd.Stderr = os.Stderr

        cmd.Env = []string{"name=shashank"}

        cmd.SysProcAttr = &syscall.SysProcAttr{
                 Cloneflags: syscall.CLONE_NEWNS |
                         syscall.CLONE_NEWUTS |
                         syscall.CLONE_NEWIPC |
                         syscall.CLONE_NEWPID |
        syscall.CLONE_NEWNET |
                      syscall.CLONE_NEWUSER,
         UidMappings: []syscall.SysProcIDMap{
                      {
                              ContainerID: 0,
                              HostID: os.Getuid(),
                              Size: 1,
                      },
                },
                GidMappings: []syscall.SysProcIDMap{
                        {
                             ContainerID: 0,
                             HostID: os.Getgid(),
                             Size: 1,
                        },
                },
         }
```

```
        must(cmd.Run())
}

func child () {

cmd := exec.Command(os.Args[2], os.Args[3:]...)
        cmd.Stdin = os.Stdin
        cmd.Stdout = os.Stdout
        cmd.Stderr = os.Stderr
must(syscall.Sethostname([]byte("myhost")))

        if err := pivotRoot("/root/book_prep/rootfs"); err != nil
                { fmt.Printf("Error running pivot_root - %s\n",
                err) os.Exit(1)
        }
must(cmd.Run())
}

func must(err error) {
        if err != nil {
                fmt.Printf("Error - %s\n", err)

        }
}
```

After executing the following program:

```
root@osboxes:~/book_prep# ./myuts parent /bin/sh
/ # ls
bin     dev     etc     home    lib     lib64   root    tmp     usr     var
/ # hostname
myhost
/ # id
uid=0(root) gid=0(root) groups=0(root)
/ # |
```

We can see the directories under `rootfs` and see that the hostname has changed. We can also see the `uid` as `0` (the root within the container).

We still have a problem. The `proc` mount is not there. We need the `proc` mount to provide information about different processes running within the namespace and as an interface to the kernel for other utilities, as explained in the pseudo file systems in earlier chapters. We need to mount the `proc` file system within the mount namespace.

The Mount Proc File System

We add the new `mountProc` function to the program:

```
package main

import (
        "fmt"
        "os"
        "os/exec"
        "path/filepath"
        "syscall"
)

func main() {

switch os.Args[1] {
        case "parent":
                parent()
        case "child":
                child()
        default:
               panic("help")
        }
}
```

```
func pivotRoot(newroot string) error {
        putold := filepath.Join(newroot, "/.pivot_root")

        // bind mount newroot to itself - this is a slight hack
           needed to satisfy the
        // pivot_root requirement that newroot and putold must
           not be on the same
        // filesystem as the current root
        // if err := syscall.Mount(newroot, newroot, "",
            syscall.MS_BIND|syscall.MS_REC, ""); err != nil {
                 return err
        }

        // create putold directory
        if err := os.MkdirAll(putold, 0700); err != nil {
                return err
        }

        // call pivot_root
        if err := syscall.PivotRoot(newroot, putold); err != nil {
                return err
        }

        // ensure current working directory is set to new root
        if err := os.Chdir("/"); err != nil {
                return err
        }

        // umount putold, which now lives at /.pivot_root
           putold = "/.pivot_root"
        if err := syscall.Unmount(putold, syscall.MNT_DETACH);
        err !=
        nil {
```

```
                return err
        }

        // remove putold
        if err := os.RemoveAll(putold); err != nil
                { return err
        }

        return nil
}

func parent() {

        cmd := exec.Command("/proc/self/exe", append([]
               string{"child"}, os.Args[2:]...)...)

        cmd.Stdin = os.Stdin
        cmd.Stdout = os.Stdout
        cmd.Stderr = os.Stderr
        cmd.Env = []string{"name=shashank"}

        cmd.SysProcAttr = &syscall.SysProcAttr{
                 Cloneflags: syscall.CLONE_NEWNS |
                         syscall.CLONE_NEWUTS |
                         syscall.CLONE_NEWIPC |
                         syscall.CLONE_NEWPID |
        syscall.CLONE_NEWNET |
                         syscall.CLONE_NEWUSER,
         UidMappings: []syscall.SysProcIDMap{
                         {
                                 ContainerID: 0,
                                 HostID: os.Getuid(),
                                 Size: 1,
                         },
                },
```

```
                GidMappings: []syscall.SysProcIDMap{
                        {
                                ContainerID: 0,
                                HostID: os.Getgid(),
                                Size: 1,
                        },
                },
        }
        must(cmd.Run())
}
func child () {
cmd := exec.Command(os.Args[2], os.Args[3:]...)
        cmd.Stdin = os.Stdin
        cmd.Stdout = os.Stdout
        cmd.Stderr = os.Stderr
//make a call to mountProc function which would mount the proc
filesystem to the already
//created mount namespace
must(mountProc("/root/book_prep/rootfs"))
must(syscall.Sethostname([]byte("myhost")))
        if err := pivotRoot("/root/book_prep/rootfs"); err != nil
                { fmt.Printf("Error running pivot_root - %s\n",
                err) os.Exit(1)
        }
must(cmd.Run())
}
```

```
func must(err error) {
        if err != nil {
                 fmt.Printf("Error - %s\n", err)
        }
}

// this function mounts the proc filesystem within the
// new mount namespace
func mountProc(newroot string) error {
        source := "proc"
        target := filepath.Join(newroot, "/proc")
        fstype := "proc"
        flags := 0
        data := ""
//make a Mount system call to mount the proc filesystem within
the mount namespace
        os.MkdirAll(target, 0755)
        if err := syscall.Mount(
                source,
                target,
                fstype,
                uintptr(flags),
                data,
        ); err != nil {
                return err
        }

        return nil
}
```

Now, when we run `ps` inside the container to list the processes running within the sandbox, we get the output shown here. The reason for this is that `ps` uses the `/proc` file system.

```
root@osboxes:~/book_prep# ./myuts parent /bin/sh
/ # ps
PID   USER     TIME   COMMAND
    1 root      0:00 /proc/self/exe child /bin/sh
    6 root      0:00 /bin/sh
    7 root      0:00 ps
/ # |
```

We can use the `nsenter` command to enter the created container namespaces. To try that, let the created container be in the running state and open another Linux terminal. Then run this command:

ps -ef | grep /bin/sh

You should see the output shown here. In my case, my container's PID is `5387`. Users should use the PIDs on their machines.

```
root@osboxes:~# ps -ef | grep /bin/sh
root        5387   3829  0 14:00 pts/1    00:00:00 ./myuts parent /bin/sh
root        5392   5387  0 14:00 pts/1    00:00:00 /proc/self/exe child /bin/sh
root        5397   5392  0 14:00 pts/1    00:00:00 /bin/sh
root        5574   5560  0 14:04 pts/0    00:00:00 grep --color=auto /bin/sh
root@osboxes:~# nsenter -a -t 5397 /bin/bash
nsenter: failed to execute /bin/bash: No such file or directory
root@osboxes:~# nsenter -a -t 5397 /bin/sh
/ # |
```

Executing `nsenter -a -t 5387 /bin/sh` allows this shell to be created in the namespaces of the process with the PID `5387`, as shown.

Enabling the Network for the Container

In previous sections, we created a container with `uts`, PID, and mount namespaces. We didn't add the network namespace. In this section, we discuss how to set up network namespaces for the container.

Before we delve into the networking topic, I will provide a small primer on virtual devices in Linux, which are essential for understanding container-based networks, or for that matter any virtual networking.

Virtual Networking a Small Primer

In a virtualized world, there is a need to send packets across virtual machines to the actual physical devices, between virtual machines, or between different containers. We need a mechanism to use virtualized devices in this way. Linux provides a mechanism to create virtual network devices, called `tun` and `tap`. The `tun` device acts at Layer 3 of the network stack, which means it receives the IP packets. The `tap` device acts at Layer 2, where it receives raw Ethernet packets.

Now one might ask, what are these devices used for? Consider a scenario where `containerA` needs to send packets outbound to another container. The packets from one packet are transmitted to the host machine, which smartly uses a `tap` device to pass the packet to a software bridge. The bridge can then be connected to another container.

Let's see how these tap devices work with a simple example. Here, I create two tap devices, called `mytap1` and `mytap2`:

```
jain_sm@instance-1:~$ sudo su
root@instance-1:/home/jain_sm# ip tuntap add name mytap1 mode tap
root@instance-1:/home/jain_sm# ip tuntap add name mytap2 mode tap
```

Listing the `tap` devices, we can see there are two network interfaces:

```
root@instance-1:/home/jain_sm# ip addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: ens4: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1460 qdisc mq state UP group default qlen 1000
    link/ether 42:01:0a:80:00:02 brd ff:ff:ff:ff:ff:ff
    inet 10.128.0.2/32 brd 10.128.0.2 scope global ens4
       valid_lft forever preferred_lft forever
    inet6 fe80::4001:aff:fe80:2/64 scope link
       valid_lft forever preferred_lft forever
3: mytap1: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN group default qlen 1000
    link/ether 1e:34:fc:78:28:f6 brd ff:ff:ff:ff:ff:ff
    inet 10.0.0.10/32 scope global mytap1
       valid_lft forever preferred_lft forever
4: mytap2: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN group default qlen 1000
    link/ether 36:df:e9:2c:ad:76 brd ff:ff:ff:ff:ff:ff
    inet 10.0.0.11/32 scope global mytap2
       valid_lft forever preferred_lft forever
```

We assign IP addresses to these devices:

```
root@instance-1:/home/jain_sm# ip addr add 10.0.0.10 dev mytap1
root@instance-1:/home/jain_sm# ip addr add 10.0.0.11 dev mytap2
```

Running a simple `ping` from one device to other results in the following:

```
root@instance-1:/home/jain_sm# ping -I 10.0.0.10 -c1 10.0.0.11
PING 10.0.0.11 (10.0.0.11) from 10.0.0.10 : 56(84) bytes of data.
64 bytes from 10.0.0.11: icmp_seq=1 ttl=64 time=0.054 ms

--- 10.0.0.11 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.054/0.054/0.054/0.000 ms
```

In these examples, we explicitly created two `tap` devices and tried a `ping` between the two.

We can also use veth pairs, which can be thought of as virtual cables that connect the virtual devices. They are used in `openstack` to connect software bridges.

First, we create a veth pair as follows:

```
root@instance-1:/home/jain_sm# ip link add firsttap type veth peer name secondtap
```

This creates two tap interfaces, called firstap and secondtap.

Now, we add IP addresses to the tap devices and run a ping:

```
root@instance-1:/home/jain_sm# ip addr add 10.0.0.12 dev firsttap
root@instance-1:/home/jain_sm# ip addr add 10.0.0.13 dev secondtap
root@instance-1:/home/jain_sm# ping -I 10.0.0.12 -c1 10.0.0.13
PING 10.0.0.13 (10.0.0.13) from 10.0.0.12 : 56(84) bytes of data.
64 bytes from 10.0.0.13: icmp_seq=1 ttl=64 time=0.032 ms
```

With a basic understanding of tun and tap devices, let's move on to how the networking set up should work between the namespace created for the container and the host's namespace. For that process, we follow these steps:

1. Create a Linux bridge on the host.
2. Create a veth pair.
3. One end of veth pair must be connected to the bridge.
4. The other end of the bridge must be connected to the network interface on the container namespace.

These steps are illustrated in Figure 6-2.

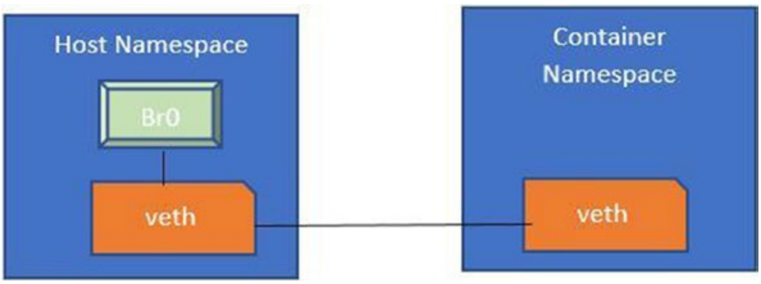


Figure 6-2. *Networking between the container's namespace and the host's namespace*

Now we modify the code to enable the network namespace:

```
package main

import (
   "fmt"
   "os"
   "os/exec"
   "path/filepath"
   "syscall"
   "time"
   "net"
)

func main() {

    switch os.Args[1] {

        case "parent":

            parent()

        case "child":

            child()

        default:

            panic("help")

        }
}

func waitForNetwork() error {

    maxWait := time.Second * 3

     checkInterval := time.Second
```

```
     timeStarted := time.Now()

     for {

           interfaces, err := net.Interfaces()

           if err != nil {

              return err

         }

         // pretty basic check ...

         // > 1 as a lo device will already
         exist if len(interfaces) > 1 {

              return nil

         }

         if time.Since(timeStarted) > maxWait {

         return fmt.Errorf("Timeout after %s waiting for
         network", maxWait)

         }

         time.Sleep(checkInterval)
         }
}

// The function allows mounting of proc filesystem
func mountProc(newroot string) error {

    source := "proc"

    target := filepath.Join(newroot, "/proc")

    fstype := "proc"
```

```
    flags := 0
    data := ""
    os.MkdirAll(target, 0755)
    if err := syscall.Mount(
         source,
         target,
         fstype,
         uintptr(flags),
         data,
    ); err != nil {
         return err
    }
    return nil
}
// this function allows to pivot the root filesystem. This allows us
// to have the root filesystem available in the sandbox
func pivotRoot(newroot string) error {
    putold := filepath.Join(newroot, "/.pivot_root")
    // bind mount newroot to itself - this is a slight hack
       needed to satisfy the
    // pivot_root requirement that newroot and putold must not
       be on the
    //same
```

```
    //filesystem as the current root

    if err := syscall.Mount(newroot, newroot, "",
syscall.MS_BIND|syscall.MS_REC, ""); err != nil {

        return err

    }

     // create putold directory

     if err := os.MkdirAll(putold, 0700); err != nil {

          return err

     }

     // call pivot_root

     if err := syscall.PivotRoot(newroot, putold); err != nil {
          return err

     }

     // ensure current working directory is set to new
     root if err := os.Chdir("/"); err != nil {

          return err

    }

    umount putold, which now lives at
    /.pivot_root putold = "/.pivot_root"

    if err := syscall.Unmount(putold, syscall.MNT_DETACH); err
!= nil {
            return err

    }
```

```
    // remove putold
    if err := os.RemoveAll(putold); err != nil {
         return err
    }
    return nil
}
func parent() {
      cmd := exec.Command("/proc/self/exe", append([]
string{"child"}, os.Args[2:]...)...)
      cmd.Stdin = os.Stdin
      cmd.Stdout = os.Stdout
      cmd.Stderr = os.Stderr
      cmd.Env = []string{"name=shashank"}
      cmd.SysProcAttr = &syscall.SysProcAttr{
           Cloneflags: syscall.CLONE_NEWNS |
               syscall.CLONE_NEWUTS |
               syscall.CLONE_NEWIPC |
               syscall.CLONE_NEWPID |
               syscall.CLONE_NEWNET |
               syscall.CLONE_NEWUSER,
      UidMappings: []syscall.SysProcIDMap{
```

```
            {
                ContainerID: 0,
                HostID: os.Getuid(),
                Size: 1,
            },
},
      GidMappings: []syscall.SysProcIDMap{
            {
                ContainerID: 0,
                HostID: os.Getgid(),
                Size: 1,
            },
      },
}
must(cmd.Start())
pid := fmt.Sprintf("%d", cmd.Process.Pid)
// Code below does the following
// Creates the bridge on the host
// Creates the veth pair
// Attaches one end of veth to bridge
// Attaches the other end to the network namespace. This is
   interesting
// as we now have access to the host side and the network side
   until // we block.
```

```
netsetgoCmd := exec.Command("/usr/local/bin/netsetgo", "-pid", pid)
    if err := netsetgoCmd.Run(); err != nil {
          fmt.Printf("Error running netsetgo - %s\n", err)
          os.Exit(1)
    }
    if err := cmd.Wait(); err != nil {
         fmt.Printf("Error waiting for reexec.Command - %s\n", err)
         os.Exit(1)
}
}
func child () {
cmd := exec.Command(os.Args[2], os.Args[3:]...)
     cmd.Stdin = os.Stdin
     cmd.Stdout = os.Stdout
     cmd.Stderr = os.Stderr
must(mountProc("/root/book_prep/rootfs"))
//must(syscall.Mount("proc", "proc", "proc", 0, ""))
must(syscall.Sethostname([]byte("myhost")))
    if err := pivotRoot("/root/book_prep/rootfs"); err != nil {
          fmt.Printf("Error running pivot_root - %s\n", err)
          os.Exit(1)
```

```
    }
//must(syscall.Mount("proc", "proc", "proc", 0, ""))
if err := waitForNetwork(); err != nil {
     fmt.Printf("Error waiting for network - %s\n", err)
     os.Exit(1)
}
if err := cmd.Run(); err != nil {
     fmt.Printf("Error starting the reexec.Command - %s\n", err)
     os.Exit(1)
}
//must(cmd.Run())
}
func must(err error) {
    if err != nil {
        fmt.Printf("Error - %s\n", err)
    }
}
```

There are a few aspects that are worth considering here. In the earlier code examples, we initialized namespaces (like changing the hostname and pivot root) in the child method. Then we launched the shell (`/bin/sh`) within the namespaces.

This mechanism worked because we just needed to initialize the namespaces, and that was being done within the namespaces themselves. When it comes to the network namespace, we need to carry out certain activities like the following:

- Create a bridge on the host.
- Create the veth pair and make one end connect to the bridge on the host and the other end within the namespace.

The problem with the current way is that when we launch the shell, we remain in the namespace until we purposely exit it. So, we need a way to return the code immediately in the API so we can execute the network setup on the host and join the veth pairs.

Fortunately, the `cmd.Run` command can be broken into two parts.

- `Cmd.Start()` returns immediately.
- `Cmd.Wait()` blocks until the shell is exited.

We use this to our advantage in the `parent` method. We execute the `cmd.Start` method, which returns immediately.

After the `start` method, we use a library called `netsetgo` created by Ed King from Pivotal. It does the following.

1. Creates the bridge on the host.
2. Creates the veth pair.
3. Attaches one end of the veth to the bridge.
4. Attaches the other end to the network namespace. This is interesting, as we now have access to the host side and the network side until we block.

Follow the instructions to download and install `netsetgo`:

```
wget "https://github.com/teddyking/netsetgo/releases/download/0.0.1/netsetgo"

sudo mv netsetgo /usr/local/bin/

sudo chown root:root /usr/local/bin/netsetgo

sudo chmod 4755 /usr/local/bin/netsetgo
```

In fact, a lot of these explanations are adapted from his examples.

The related code snippet is shown here:

```
must(cmd.Start())

pid := fmt.Sprintf("%d", cmd.Process.Pid)

netsetgoCmd := exec.Command("/usr/local/bin/netsetgo", "-pid", pid)

if err := netsetgoCmd.Run(); err != nil {

     fmt.Printf("Error running netsetgo - %s\n", err)

     os.Exit(1)

}

if err := cmd.Wait(); err != nil {

     fmt.Printf("Error waiting for reexec.Command - %s\n", err)

     os.Exit(1)

}
```

Once this is done, we use `cmd.Wait()`, which relaunches the program (`/proc/self/exe`). Then we execute the child process and go ahead with all the other initializations. After the initializations, we can launch the shell within the namespaces.

Next, we should verify the network communication from the host to the container and from the container to the host. First run this program:

```
/myuts parent /bin/sh
```

Within the container shell, run the `ifconfig` command. You should see the container's IP address, as shown here.

```
root@osboxes:~/src# ./myuts parent /bin/sh
/ #
/ # ifconfig
veth1     Link encap:Ethernet  HWaddr DE:BC:93:2E:D3:6F
          inet addr:10.10.10.2  Bcast:0.0.0.0  Mask:255.255.255.0
          inet6 addr: fe80::dcbc:93ff:fe2e:d36f/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:21 errors:0 dropped:0 overruns:0 frame:0
          TX packets:7 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:2676 (2.6 KiB)  TX bytes:586 (586.0 B)

/ #
```

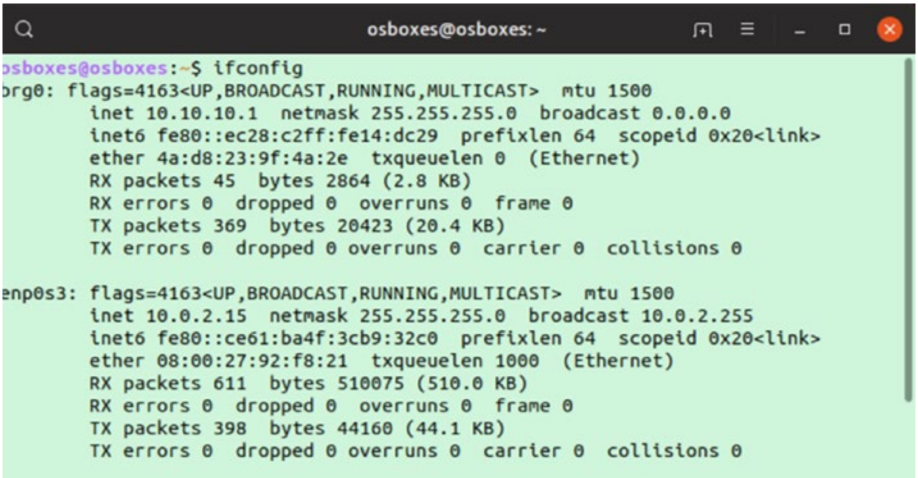
Keep the container running and open another terminal (a bash shell) on the host. Run the following command, which pings the container's IP:

```
ping 10.10.10.2
```

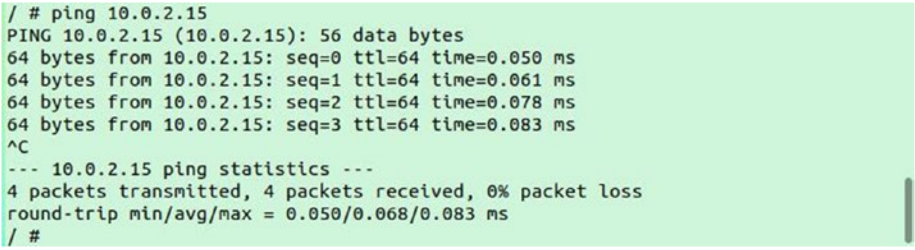
```
osboxes@osboxes: ~
osboxes@osboxes:~$ ping 10.10.10.2
PING 10.10.10.2 (10.10.10.2) 56(84) bytes of data.
64 bytes from 10.10.10.2: icmp_seq=1 ttl=64 time=0.098 ms
64 bytes from 10.10.10.2: icmp_seq=2 ttl=64 time=0.045 ms
^C
--- 10.10.10.2 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 24ms
rtt min/avg/max/mdev = 0.045/0.071/0.098/0.027 ms
osboxes@osboxes:~$
```

Note that we are able to ping the container's IP address from the host.

Now try the pinging the host IP address from the container. First, get the host IP address by running the `ifconfig` command. As you can see here, my host IP address is `10.0.2.15`:



Now ping this host IP from the container:



As you can see, we could ping from the container to the host as well as from the host to the container, so networking communication is working both ways.

Let's recap what we have achieved thus far.

- We created a container with `unshare` and demonstrated the ability to change the hostname within a `uts` namespace.
- We created a container with Golang with namespaces like UTS and user-namespaces.

- We add mount namespaces and demonstrated how a separate `proc` file system can be mounted within the namespace.
- We added network capabilities to the namespace, which allow us to communicate between the container namespaces and the host namespace.

Enabling Cgroups for the Container

We earlier mounted a cgroup on `/root/mygrp`. We created a directory child within it. Now we will put our process within the cgroup and cap its maximum memory.

Here is the sample code snippet:

```
func enableCgroup() {
    cgroups := "/root/mygrp"
    pids := filepath.Join(cgroups, "child")

    must(ioutil.WriteFile(filepath.Join(pids, "memory.max"), []
byte("2M"), 0700))


    must(ioutil.WriteFile(filepath.Join(pids, "cgroup.procs"),
    []byte(strconv.Itoa(os.Getpid())), 0700))
}
```

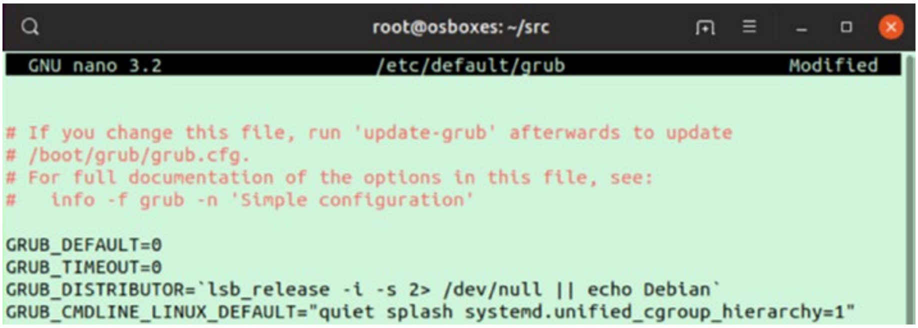
In this code snippet, we add the PID of the process we create within the container (`/bin/sh`) to the `cgroup.procs` file and cap the maximum memory for the process to 2MB.

Before executing this code, you need to make one configuration change to the OS. Open the `/etc/default/grub` file using Nano or your favorite editor:

```
nano /etc/default/grub
```

In this file, you have to modify the `GRUB_CMDLINE_LINUX_DEFAULT` key to add `systemd.unified_cgroup_hierarchy=1`. Refer the following image for clarification.

```
GRUB_CMDLINE_LINUX_DEFAULT="quiet splash systemd.unified_
cgroup_hierarchy=1"
```



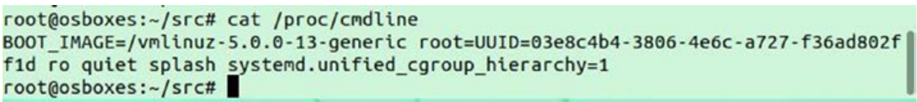
```
root@osboxes: ~/src
GNU nano 3.2                /etc/default/grub                Modified

# If you change this file, run 'update-grub' afterwards to update
# /boot/grub/grub.cfg.
# For full documentation of the options in this file, see:
#   info -f grub -n 'Simple configuration'

GRUB_DEFAULT=0
GRUB_TIMEOUT=0
GRUB_DISTRIBUTOR=`lsb_release -i -s 2> /dev/null || echo Debian`
GRUB_CMDLINE_LINUX_DEFAULT="quiet splash systemd.unified_cgroup_hierarchy=1"
```

After the update, run the command and reboot the system:

```
sudo update-grub
```

After the system reboots, run this command:

```
cat /proc/cmdline
```



```
root@osboxes:~/src# cat /proc/cmdline
BOOT_IMAGE=/vmlinuz-5.0.0-13-generic root=UUID=03e8c4b4-3806-4e6c-a727-f36ad802f
f1d ro quiet splash systemd.unified_cgroup_hierarchy=1
root@osboxes:~/src#
```

You should see `systemd.unified_cgroup_hierarchy=1` as the `BOOT_IMAGE` key in the `/proc/cmdline`.

To create a cgroup, run the following commands in the terminal. Use the same folders we used in the program.

```
mkdir -p /root/mygrp

mount -t cgroup2 none /root/mygrp

mkdir -p /root/mygrp/child
```

Now you can run this program:

```
package main

import (
        "fmt"
        "io/ioutil"
        "os"
        "os/exec"
        "path/filepath"
        "strconv"
        "syscall"
        "time"
        "net"
)

func main() {

switch os.Args[1] {
        case "parent":
                parent()
        case "child":
                child()
        default:
               panic("help")
        }

}
```

```
func enableCgroup() {
        cgroups := "/root/mygrp"
        pids := filepath.Join(cgroups, "child")

        must(ioutil.WriteFile(filepath.Join(pids,
"memory.max"), []byte("2M"), 0700))

        must(ioutil.WriteFile(filepath.Join(pids,
"cgroup.procs"), []byte(strconv.Itoa(os.Getpid())), 0700))

}

func waitForNetwork() error {
        maxWait := time.Second * 3
        checkInterval := time.Second
        timeStarted := time.Now()

        for {
                interfaces, err := net.Interfaces()
                if err != nil {

                        return err
            }

            // pretty basic check ...
            // > 1 as a lo device will already exist

            if len(interfaces) > 1 {
                    return nil
            }

            if time.Since(timeStarted) > maxWait {
                    return fmt.Errorf("Timeout after %s waiting
                    for network", maxWait)
            }
```

```
            time.Sleep(checkInterval)
        }
}

func mountProc(newroot string) error {
        source := "proc"
        target := filepath.Join(newroot, "/proc")
        fstype := "proc"
        flags := 0
        data := ""

       os.MkdirAll(target, 0755)
       if err := syscall.Mount(
               source,
               target,
               fstype,
               uintptr(flags),
               data,
       ); err != nil {
               return err
       }

       return nil
}

func pivotRoot(newroot string) error {
        putold := filepath.Join(newroot, "/.pivot_root")

        // bind mount newroot to itself - this is a slight hack
           needed
       // to satisfy the pivot_root requirement that newroot
          and putold
       // must not be on the same filesystem as the current root
```

```
         if err := syscall.Mount(newroot, newroot, "",
syscall.MS_BIND|syscall.MS_REC, ""); err != nil {
        return err
       }

       // create putold directory
       if err := os.MkdirAll(putold, 0700); err != nil
               { return err
       }

       // call pivot_root
       if err := syscall.PivotRoot(newroot, putold); err != nil
               { return err
       }

       // ensure current working directory is set to new
       root if err := os.Chdir("/"); err != nil {
                    return err
       }

       // umount putold, which now lives at
       /.pivot_root putold = "/.pivot_root"

       if err := syscall.Unmount(putold, syscall.MNT_DETACH); err !=
nil {
               return err
       }
       // remove putold
       if err := os.RemoveAll(putold); err != nil {
               return err
       }
       return nil
}
```

```
func parent() {

        cmd := exec.Command("/proc/self/exe", append([]
               string{"child"}, os.Args[2:]...)...)

        cmd.Stdin = os.Stdin
        cmd.Stdout = os.Stdout
        cmd.Stderr = os.Stderr

        cmd.Env = []string{"name=shashank"}

        cmd.SysProcAttr = &syscall.SysProcAttr{
                Cloneflags: syscall.CLONE_NEWNS |
                        syscall.CLONE_NEWUTS |
                        syscall.CLONE_NEWIPC |
                        syscall.CLONE_NEWPID |
                        syscall.CLONE_NEWNET |
                        syscall.CLONE_NEWUSER,
        UidMappings: []syscall.SysProcIDMap{
              {
                        ContainerID: 0,
                        HostID: os.Getuid(),
                        Size: 1,
              },
        },

        GidMappings: []syscall.SysProcIDMap{
              {
                         ContainerID: 0,
                         HostID: os.Getgid(),
                         Size: 1,
              },
        },

}
```

```
    must(cmd.Start())

pid := fmt.Sprintf("%d", cmd.Process.Pid)
netsetgoCmd := exec.Command("/usr/local/bin/netsetgo", "-pid",
pid) if err := netsetgoCmd.Run(); err != nil {
     fmt.Printf("Error running netsetgo - %s\n", err)
     os.Exit(1)
}

if err := cmd.Wait(); err != nil {
     fmt.Printf("Error waiting for reexec.Command - %s\n", err)
     os.Exit(1)
}

}

Func child () {
//enable the cgroup functionality

enableCgroup()
cmd := exec.Command(os.Args[2], os.Args[3:]...)
        cmd.Stdin = os.Stdin
        cmd.Stdout = os.Stdout
        cmd.Stderr = os.Stderr

must(mountProc("/root/book_prep/rootfs"))
//must(syscall.Mount("proc", "proc", "proc", 0, ""))

must(syscall.Sethostname([]byte("myhost")))

        if err := pivotRoot("/root/book_prep/rootfs"); err != nil
             { fmt.Printf("Error running pivot_root - %s\n",
             err) os.Exit(1)
        }

//must(syscall.Mount("proc", "proc", "proc", 0, ""))
```

```
if err := waitForNetwork(); err != nil {
            fmt.Printf("Error waiting for network - %s\n", err)
           os.Exit(1)
        }

if err := cmd.Run(); err != nil {
          fmt.Printf("Error starting the reexec.Command - %s\n", err)

          os.Exit(1)
        }

//must(cmd.Run())

}
func must(err error) {
        if err != nil {
             fmt.Printf("Error - %s\n", err)
        }
}
```

Figure 6-3 shows the process PID added to the cgroup and the value stored in the memory.max file, which we defined in the program.

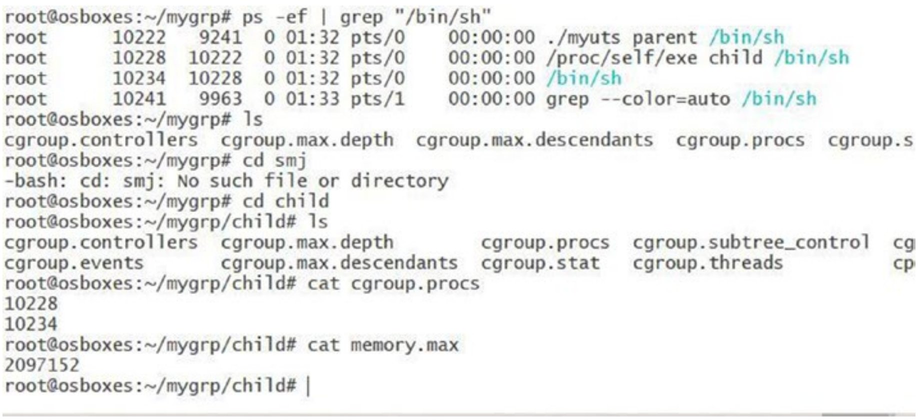


Figure 6-3. *The process PID added to the cgroup and the value stored in the memory.max file*

Summary

In the book, we covered the basics of virtualization. We delved into how virtualization works and the basic techniques used to achieve it. We covered different packet flow scenarios, as to how communication happens from a VM to a hypervisor.

The book covered the specifics of Linux containers (namespaces, cgroups, and union file systems) and how containers are realized within the Linux kernel. We took a stab at writing a Linux container and saw how, with some simple programming, we can create a simple container runtime like Docker.

You are advised to go over each exercise and try different combinations of the code. As an example, you could do the following:

1. Try a new `rootfs` rather than busybox.
2. Try container-to-container networking.
3. Play with more resource controls.
4. Run an HTTP server within one container and an HTTP client within other container and establish a communication over HTTP.

You should now have a decent idea as to what happens under the hood within a container. Therefore, when you use different container orchestrators like Kubernetes or Swarm, you'll more easily understand what is actually happening.

Index

A

B

C, D

E

S. M. Jain, *Linux Containers and Virtualization*, https://doi.org/10.1007/978-1-4842-6283-2

U

V, W, X, Y, Z