

# Containers

G. Lettieri

23 Nov. 2017

## 1 Introduction

The most important use-case for virtualization is the safe sharing of hardware resources among different applications. Typically, each virtual machine only runs just one application. We can observe that sharing the resources among applications is what Operating Systems should be designed for, so do we really need Virtual Machines at all for this use case? The answer is not so simple, since unfortunately existing OSs have increasingly become less good at isolating user applications from each other. However, *Containers* try to improve the situation by adding new isolation features to the traditional host OS. They are put forward as an alternative to Virtual Machines.

Containers (or jails) are a feature of some host operating systems, but we limit our considerations on Linux only, since this is what is commonly used in this area. Containers are a way to isolate a set of processes and make them think that they are the only ones running on the machine. The machine they see may feature only a subset of the resources actually available on the entire machine (e.g., less memory, less disk space, less CPUs, less network bandwidth). Many different containers may coexist on the same machine.

Containers are *not* virtual machines, even if they may look like ones in some cases. Processes running inside a container are normal processes running on the host kernel. There is no guest kernel running inside the container, and this is the most important limitation of containers with respect to virtual machines: you cannot run an arbitrary operating system in a container, since the kernel is shared with the host (Linux, in our case). The most important advantage of containers with respect to virtual machines is performance: there is no performance penalty in running an application inside a container compared to running it on the host. There is also a point of contention between containers and virtual machines, about which one is more secure. VMs are considered (by some) to be more secure of containers, because they have a smaller *attack surface*. By attack surface we mean the amount of code and features that a malicious attacker may probe for exploitable bugs: the entire host kernel, in the case of containers, and the hypervisor in the case of virtual machines. The KVM and Xen hypervisors, for example, are very small. In the KVM case, it must be noted that KVM is a small module, but it actually uses facilities from the rest of the linux kernel (for


scheduling, virtual memory, etc.). However, this is still less than the amount of code involved in the implementation of containers.

Linux Containers are implemented using two distinct kernel features: namespaces and control groups. We briefly examine each one in turn.

## 1.1 Namespaces

Namespaces provide a means to segregate system resources so that they can be hidden from selected processes. An old feature of Unix systems, which has a similar purpose, is the `chroot()` system call, which works as follows:

- For each process, the kernel remembers the inode of the process *root directory*;
- This directory is used as a starting point whenever the process passes the kernel (e.g., in a `open()`) a filesystem path that starts with “/”;
- whenever the kernel walks through the components of any filesystem path used by the process and reaches the process root directory, a subsequent “.” path element is ignored;
- only root can call `chroot()`;
- the process root directory is inherited by its children.

Normally, all processes have the same root directory, which coincides with the root directory of the file system. But, by using `chroot()`, we can make a subset of the filesystem look like it was the full filesystem for a set of processes. This is typically used to segregate untrusted processes that provide network services and, because of possible bugs in their implementations, may be forced by remote attackers to execute arbitrary code. The idea is to prepare a subtree in the filesystem that contains only the things that are needed for the execution of the server, and nothing else—a chroot environment. Then, the server process is started after a `chroot()` to the root directory of the chroot environment. Even if the server is subverted, it cannot access any file outside of the chroot environment. 

Chroot environments, however, are not full containers. Contrary to popular belief, in fact, not everything is a file in Unix. For example network interfaces, network ports, users and processes are not files. While we can have as many instances as we want of, say, `/etc/passwd`, each different and living in its own chroot environment, we can only have one port 80 throughout the system (thus, only one web server), only one process with pid 1 (thus, only one init process), and user and process ids will have the same meaning in all chroot environments. Thus, for example, a process running in a chroot environment will still be able to see all the processes running in the system, and it will be able to send signals to all the processes belonging to any user with the same user id as its own.

---

<sup>1</sup>Note that in the earlier implementations of the mechanism, root was able to escape a chroot environment, so this strategy was only effective if the server did not run as root.

Namespaces have been introduced to create something similar to chroot environments for all these other identifiers. Each process in Linux has its own network namespace, pid namespace, user namespace and a few others. Network interfaces and ports are only defined inside a namespace, and the same port number may be reused in two different namespaces without any ambiguity. The same holds true for processes and users. Normally, all processes share the same namespaces, but a process can start a new namespace that will be then inherited by all its children, grandchildren, and so on. This is done when the process is created using the `clone()` system call. This system call (taken from the Plan 9 OS) is the new, preferred way to create new processes in Linux, since it generalizes the behaviour of `fork()` and can be also used to implement `pthread_create()`. The idea is that both processes and threads share something with their creating process, and have a private copy of something else. For example, processes share open files with their parent, but have a private copy of all the process memory. Threads, instead, also share the process memory. The `clone()` system call is passed a set of flags using which the programmer may choose what to share and what to copy. This same system call has been extended to implement namespaces, essentially by adding flags for the sharing or copying of the network, pid, user namespaces and so on.

## 1.2 Control groups

While namespaces can be used to hide and create private copies of all the system entities, they are not sufficient in isolating sets of processes so that they cannot interfere with each other. Processes may interfere also by abusing the system resources, e.g., allocating too much memory, using too much CPU time, or disk and network bandwidth. To properly implement containers, therefore, we also need to limit the usage of resources by the processes that live in the container. This is another thing that was not done very well before the introduction of *control groups*. The problem is that, before enforcing a limit on a set of processes, we need to know which processes belong to the set, and the processes must not be able to escape from the set. We would also like some flexibility in the definition of the set. Traditional Unix has a concept of process groups, but unfortunately any process is free to enter or leave a group. Processes are also grouped by user id (the user that is running them), but this is not very flexible; moreover, processes may temporarily switch their user id when they execute `setuid` programs.

Control groups, instead, are groups explicitly created by the administrator, who can later assign processes to them. The administrator may setup the system so that processes cannot escape their control group.

There are two implementations of the cgroups framework in Linux: version 1 and version 2. The second version is very recent and most installations are still using version 1.

Control groups can be organized in a tree-shaped *hierarchy*, like the one in Fig. [1](#). Each process in the system must belong to exactly one control group in the hierarchy, and therefore the hierarchy is a partition of the system processes.

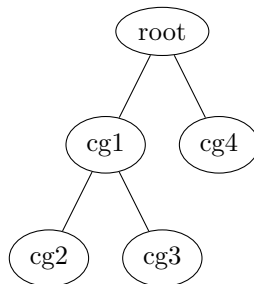


Figure 1: An example cgroup hierarchy

When a process creates a child process, the child inherits the control group of its parent. Note that some cgroups may be empty. Indeed, it is good practice to put processes only in the root cgroup and in the leaf cgroups, leaving all intermediated cgroups empty. E.g., in Fig. 1 cgroup cg1 should be left empty. In version 2 this has become mandatory.

In version 1 there may be many independent hierarchies. Each process must belong to a cgroup in *each* hierarchy, and therefore each hierarchy is a different partition of the processes. This was meant to enhance flexibility, but was later found to be very complex to implement and use; therefore, version 2 has dropped this feature: there is now only one cgroup hierarchy in the system.

Once we have the ability to reliably group processes in one or more hierarchy, we can start controlling their resource usage. To this aim, hierarchies can be linked to so-called *subsystems* (subsystem *controllers* would be a better name). Subsystems are used to control the resources assigned to the cgroups in the linked hierarchy. Some examples of existing subsystems are:

- memory** limits the amount of main memory used by each cgroup;
- cpu** limits the maximum fraction of CPU that each cgroup may use and may schedule the CPU based on cgroups *weights*;
- cpuacct** this is not much of a controller, since it only provides accounting information about the CPU usage of the cgroups;
- cpuset** on multi-cpu systems, limits the set of CPUs that may be used by each cgroup;
- pids** limits the number of processes that can be created in a cgroup.

Other subsystems control device access, block I/O and so on.

In version 1 each subsystem may be attached to at most one of the existing hierarchies, while in version 2 all the subsystems are attached to the single system hierarchy. A typical setup for version 1 is to have a separate hierarchy for each subsystem, with a few exceptions (e.g., cpu and cpuacct are usually attached to the same hierarchy).

The interpretation of the hierarchical relations among the cgroups is up to each subsystem. The general idea, however, is that limits imposed on parent cgroups should also be enforced on their descendants.

### 1.2.1 Using cgroups

Cgroups are managed via a pseudo-filesystem. The idea is to mimic the cgroups hierarchy in a pseudo-directory tree. The subsystems controllers then add their own pseudo-files to each pseudo-directory in the tree.

Let us consider an example, using the version 1 implementation. To create a new cgroups hierarchy, without any subsystem attached, we need commands like:

```
mkdir mytree
sudo mount -t cgroup -o none,name=myhierarchy cgroup mytree
```

Note the option **none** that says that we don't want to attach subsystems controllers to the hierarchy. Here we are also giving a name (**myhierarchy**) to the hierarchy.

Initially, the hierarchy consists only of the root node, represented by the **mytree** directory. We can see that **mytree** has already been populated by some pseudo-files (this is done automatically by the cgroup pseudo-filesystem implementation in the kernel). The most important one is **tasks**, which contains a list of all task that belong to this cgroup. If we try **cat mytree/tasks** we can see that it now contains all processes in the systems, since each process must belong to a cgroup in each hierarchy, and this hierarchy has only one cgroup yet.

To create a new cgroup, we use **mkdir**:

```
sudo mkdir mytree/cg1
```

We can now see that also **cg1** has been automatically populated with pseudo-files. In particular, the pseudo-file **mytree/cg1/tasks** is for the processes that belong to **cg1** and it is initially empty.

To move a process to the new cgroup we need to write its pid into the **tasks** file inside **cg1**:

```
sudo -s
echo $$ > mytree/cg1/tasks
```

With the first command we start a root shell. The special variable **\$\$** contains the pid of the current shell, so now our new shell and all the processes that it creates will belong to this cgroup. Note that, since a process must be in only one cgroup for each hierarchy, the root shell has also been removed from its old cgroup (the root one, in this case). If we now run **cat mytree/cg1/tasks** we should see two pids: the pid of our shell and the pid of the process created by our shell to execute **cat**. If we run the command several times we should see that the second pid changes every time.

If we exit from the root shell we should now see that the `tasks` file has become empty. To remove a control group we can use `rmdir`

```
sudo rmdir mytree/cg1
```

This can only be done if the `tasks` file of the cgroup is empty. Note that, unlike a “normal” `rmdir`, deleting the `cg1` directory will succeed even if it looks not-empty, due to the pseudo-files it contains. Again, this is done automatically by the kernel.

A hierarchy with subgroups is kept in the kernel even if its pseudo-filesystem is not mounted anywhere. The mounted pseudo-filesystem just gives userspace access to the in-kernel hierarchy, which in this case is identified by its name (`myhierarchy`, in the example). The pseudo-filesystem can also be mounted several times in different locations, and all the mount points will give access to the same information. The in-kernel data structures are freed when they are no longer needed (i.e., when there is no subgroup and no mount-point).

To create a hierarchy with one or more subsystem controllers attached we can issue a command like

```
mkdir mytree2
sudo mount -t cgroup -o cpu,memory,pids cgroup mytree2
```

where we are trying to create a hierarchy and attaching to it the `cpu`, `memory` and `pids` subsystems (note that we are not giving a name this time, since the set of attached subsystems is also sufficient to identify a hierarchy).

If you try to issue the above command you will probably obtain an error, since a subsystem cannot be attached to more than one hierarchy, and your Linux distribution has very likely already attached these subsystems at boot. All hierarchies are typically mounted in subdirectories of `/sys/fs/cgroup`, each named after the corresponding set of attached subsystems. E.g., in an Ubuntu OS (or any OS managed via `systemd`) we can interact with the `cpu` subsystem (and its attached hierarchy) by looking inside `/sys/fs/cgroup/cpu`. Here we find the usual cgroup related pseudo-files (e.g., `tasks`) plus the pseudo-files added by the subsystem controller. Their names follow the pattern “*subsystem.parameter*”.

As an example, we can take a look at the `cpu.shares` pseudo-file. This is added by the `cpu` subsystem and can be used to assign a “relative weight” to each cgroup. These weights are used to assign a fraction of the available CPU time to each cgroup according to the following rules:

- the root cgroup is assigned 100% of the time;
- if a cgroup has  $n$  child subgroups with weights  $w_1, w_2, \dots, w_n$ , the  $i$ -th child is assigned the fraction  $w_i / \sum_{j=1}^n w_j$  of the parent time.

Note that the CPU is assigned in a *work conserving* fashion: if any cgroup is not using all of its CPU time, the remaining time is made available for the other cgroups.

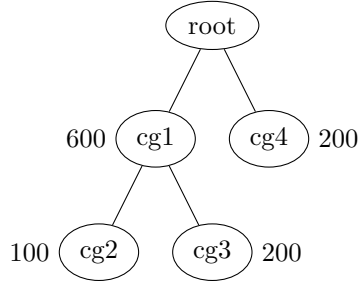


Figure 2: An example assignment of CPU weights to the hierarchy of Fig. 1

For example, let us consider the assignment of weights given in Figure 2. We can create the hierarchy and assign the shares with the following commands as root

```

cd /sys/fs/cgroup/cpu
mkdir cg1
echo 600 > cg1/cpu.shares
mkdir cg1/cg2 cg1/cg3
echo 100 > cg1/cg2/cpu.shares
echo 200 > cg1/cg3/cpu.shares
mkdir cg4
echo 200 > cg4/cpu.shares

```

If we call  $F_i$  the fraction of CPU-time assigned to cgroup  $cg_i$ , we have

$$\begin{aligned}
 F_1 &= \frac{600}{600 + 200} = \frac{3}{4}, \\
 F_2 &= \frac{100}{100 + 200} F_1 = \frac{1}{3} F_1 = \frac{1}{4}, \\
 F_3 &= \frac{200}{100 + 200} F_1 = \frac{2}{3} F_1 = \frac{1}{2}, \\
 F_4 &= \frac{200}{600 + 200} = \frac{1}{4}.
 \end{aligned}$$

It is easy to check the above setup and calculations by creating a program that uses all of its available CPU-time (e.g., a simple infinite loop), running it several times, moving the corresponding processes into the available subgroups and observing the CPU times reported by `top`. Note that, even if modern systems have many CPUs, it is simpler to use only one CPU for this experiment. This can be accomplished by using the `taskset` utility, which can run a program by pinning the process on the given set of CPUs. E.g.,

```
sudo taskset -c 0 ./loop
```

will run the `./loop` program only on CPU 0.