Univerza v Ljubljani
Fakulteta za elektrotehniko

# Containerization via cgroups and namespaces

Name: Danilo Pejović

Date: 11.2.2022

# Table of contents

# 1. Introduction

This seminar will focus on the inner workings of container technology. Containerization is the type of virtualization that doesn't "trick" guest OS into believing that they have access to underlying hardware but instead isolates the process enough to where it operates as a separate machine. Due to its agility and ease of use container virtualization has become de facto industry standard. The goal of this seminar is to understand the underlying technology and use that knowledge to hopefully prevent hours of debugging deployment of applications for preventable reasons.

An example of such a problem was a project I was involved a few years ago when I just learned about Docker. I was trying to make docker deployment of a Django application which, due to quirks in the code, was expected to be run in a python virtual environment even if it is already separated in a container. I tried to go about that by simply activating the virtual environment as a command inside Dockerfile. That did not work because I didn't know the basics of how docker containers are built – every command in Dokcerfile builds a new layer on docker image (which is a very useful function that will be talked about later in this seminar), which led to my virtual environment being deactivated every time docker build would create a new layer of the image. That rendered python virtual environment useless and thus whole deployment was not possible. After several hacky attempts to solve that problem by activating the virtual environment every time I use new command in Dockerfile, I finally decided to read what virtual environments are and found a really simple solution for the problem. Since virtual environments only change 2 environmental variables, all I needed to do was export values I needed to those environment variables, and I would make changes that would transcend layers.

Right now, there are too many technologies that we interface with on daily basis, especially developers, that it is impossible to "do your homework" on everything. Sometimes short tutorials will have to suffice, but with technology so often used I feel it will be useful to better understand it and the development path that led to solutions we most often use today. Along with deep dive into the theory of Linux containers, during this seminar I will try to create a simple container of my own and explain the process it would take to give it part of the functionalities that modern container engines offer. When relevant, I will add short demos on Linux functionalities to further explain how they work.

## 2. Namespaces

Namespaces are logical isolations of processes within the Linux kernel. Changes made to a global resource are visible only to processes that are members of that namespace. Generally, a namespace is automatically torn down when the last process in that namespace terminates or moves to another namespace. As of the last Linux version following namespace types are available:

- UTS
- Mount
- PID
- Network
- Inter Process Communication (IPC)
- Control Group (cgroup)
- User

By default, every process we create is a member of a namespace of each type. Unless further configuration is done, the process will reside in root namespaces or in the same namespaces of a process that spawned them.

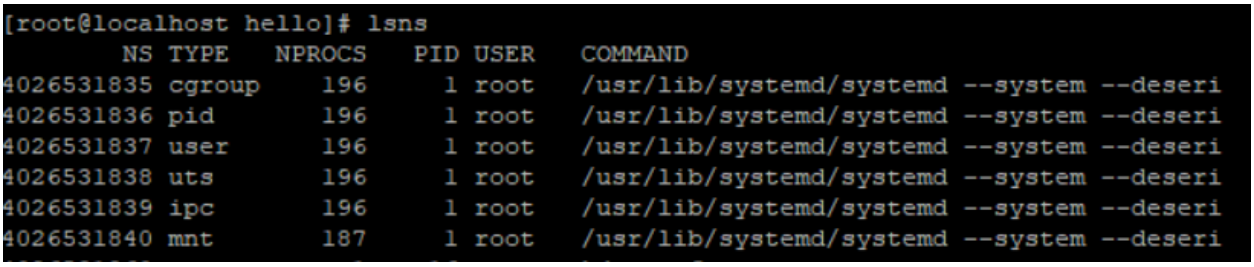


*Figure 1. Namespaces that terminal session process belongs to*

Namespace configuration is very flexible. For example, if we are developing an application on a server, it is possible to put database and web app as part of the same network namespace – assuring they have easy communication, while making sure they don't have access to the same files.

When you run a docker container, Docker creates a set of namespaces for that container – making sure that the docker container is isolated from the rest of the system.

Namespaces are created via system calls, and following 3 system calls are most commonly used with namespaces [1]:

- Clone: create child processes. Similar to system call fork(), with the difference that clone() calls offer more precise control over what pieces of execution context are shared between parent and child process. This call allows for child processes to be put into separate namespaces.
- Unshare: disables sharing a namespace with a parent process. Worth noting is that this system call does not spawn a new namespace to replace the namespace child process left.
- Setns: Attaches a process to an already existing namespace

## 2.1 Mount namespaces

The first type of namespaces to be added back in 2002, with a goal of restricting what files can each process view. When mounting or unmounting a filesystem, changes will be noticed by all processes that share their namespace. When it comes to container mount namespaces are important because they provide a way for different filesystem layouts (from host system filesystem layout) to exist inside a container [2]. As a quick demonstration of how mount namespaces operate, I will use 2 terminal sessions that belong to different namespaces and use that fact to make a file that the root user cannot see (which is unintuitive since we expect root user to have access to everything).

Having pulled centos 7 docker image earlier, I will spring up an instance of that image (so I don't do permanent damage to my virtual machine while I play around with namespaces), I will start a bash session in which I will create a new folder and then unshare that bash session from default namespace:

```
[root@localhost hello]# docker run -i -t --privileged eeb6ee3f44bd
[root@lbf7799bc172 /]# mkdir /tmp/mount_ns
[root@lbf7799bc172 /]# unshare -m /bin/bash
```

```
[root@lbf7799bc172 /]#  mount -n -t tmpfs tmpfs /tmp/mount_ns
```

*Figure 2. Moving bash session to new mount namespace and then mounting a folder to that namespace*

After opening a new terminal session to that same container, we can see that 2 terminal sessions share every namespace except the mount namespace:

```
cgroup -> cgroup:[4026531835]
ipc -> ipc:[4026532292]
mnt -> mnt:[4026532290]
net -> net:[4026532295]
pid -> pid:[4026532293]
pid_for_children -> pid:[4026532293]
time -> time:[4026531834]
time_for_children -> time:[4026531834]
user -> user:[4026531837]
uts -> uts:[4026532291]
```

```
cgroup -> cgroup:[4026531835]
ipc -> ipc:[4026532292]
mnt -> mnt:[4026532383]
net -> net:[4026532295]
pid -> pid:[4026532293]
pid_for_children -> pid:[4026532293]
time -> time:[4026531834]
time_for_children -> time:[4026531834]
user -> user:[4026531837]
uts -> uts:[4026532291]
```

*Figure 3. Namespace comparison of newly opened bash session (left) and session with unshared mount space(right)*

As we can see in Figure 3. Identifiers for all namespaces are identical except for the mount (mnt) namespace. Creating a file inside /tmp/mount_ns folder will only impact the namespace in which it was created, while files created outside that folder will be visible to both terminal sessions, since function unshare had a new namespace inherit all mountpoints of the default namespace.

```
[root@1bf7799bc172 mount_ns]# df -h
Filesystem            Size  Used Avail Use% Mounted on
overlay                13G  6.0G  6.6G  48% /
tmpfs                  64M     0   64M   0% /dev
shm                    64M     0   64M   0% /dev/shm
tmpfs                 405M     0  405M   0% /sys/fs/cgroup
/dev/mapper/cl-root    13G  6.0G  6.6G  48% /etc/hosts
tmpfs                 405M     0  405M   0% /tmp/mount_ns
```

*Figure 4. Mount points of newly created namespace*

```
[root@1bf7799bc172 test2]# ll
total 0
-rw-r--r--. 1 root root 0 Jan 30 19:35 test
-rw-r--r--. 1 root root 0 Jan 30 19:56 test2
-rw-r--r--. 1 root root 0 Jan 30 19:56 test3
[root@1bf7799bc172 test2]# cd ../mount_ns/
[root@1bf7799bc172 mount_ns]# ll
total 0
-rw-r--r--. 1 root root 0 Jan 30 19:29 test
-rw-r--r--. 1 root root 0 Jan 30 19:31 test2
```

```
[root@1bf7799bc172 test2]# ll
total 0
-rw-r--r--. 1 root root 0 Jan 30 19:35 test
-rw-r--r--. 1 root root 0 Jan 30 19:56 test2
-rw-r--r--. 1 root root 0 Jan 30 19:56 test3
[root@1bf7799bc172 test2]# cd ../mount_ns/
[root@1bf7799bc172 mount_ns]# ll
total 0
-rw-r--r--. 1 root root 0 Jan 30 19:32 test3
```

*Figure 5. Files that newly created namespace has access to (left) and default namespace (right)*

Folder /tmp/test2 was created outside of /tmp/mount_ns – as a result, both processes have equal access to it. But inside mount_ns folder they only have access to files created in their namespace. Root user has a way to get around this limitation by searching trough /proc directory, and finding process filesystem by its pid. By finding container pid on host machine and then bash pid on container – we can see those files as a root user of a machine that is hosting docker container.

```
root@danilo-VirtualBox:/proc/2403/root/proc/52/root/tmp/mount_ns# ll
total 4
drwxrwxrwt 2 root root   80 feb 11 17:10 ./
drwxrwxrwt 1 root root 4096 feb 11 17:00 ../
-rw-r--r-- 1 root root    0 feb 11 17:10 test1
-rw-r--r-- 1 root root    0 feb 11 17:09 test2
```

*Figure 6. Accessing files in a newly created namespace in a container from host machine*

Using this workaround we are even able to change files inside docker container directly from host machine, but that is highly discouraged as it can produce unpredictable bheaviours of docker container.

## 2.2 Process ID namespaces

Process ID identifies a process to a system, and it is integral that PID for every process is unique. PID namespaces separate PID number space, meaning that as long as two processes don't belong to the same namespace, they can have the same PID. This functionality allows containers to [2]:

- Suspend/resume processes
- Migrate containers to a new machine

as it allows processes to have their own PID regardless of host system. PID namespaces for the most part act like host PID number space. Process with namespace PID of 1 is called init process and is critical to the lifetime of a namespace. If init process of a namespace is destroyed, kernel will destroy all other processes in that namespace. In case a child process, whose parent is part of the same PID namespace, gets orphaned – init process will become a new parent of that process [17]. PID namespaces are one-way communication – while parent namespace will have information about all processes in all nested namespaces below it, child namespace does not have access to parent processes. This is very useful when it comes to security, especially in development – as a security breach in our container would not engender the whole server.

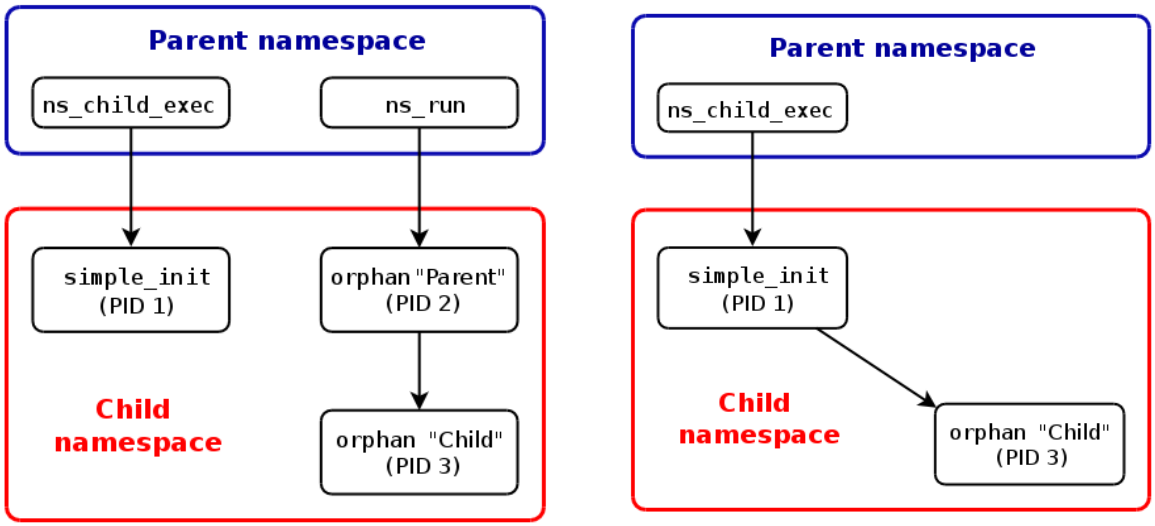


*Figure 7. Init process inheriting a child process from another process in a namespace [17]*

## 2.3 Inter Process Communication namespace

Interprocess communication namespaces provide a way of safely exchanging data and synchronizing the actions between threads and processes – for processes in the same namespace. An example of interprocess communication aspect is enabling processes in shared namespace sharing memory. IPC namespaces provide primitives such as semaphores, file locks, mutexes, … [3]

## 2.4 Unix Timesharing System (uts) namespace

UTS namespace allows you to separate hostnames for processes in different namespaces. Oftentimes this functionality is simply a convenience as most of the communication is done via IP and port numbers. [4]

```
[root@lbf7799bcl72 mount_ns]# hostname
lbf7799bcl72
[root@lbf7799bcl72 mount_ns]# unshare -u /bin/bash
[root@lbf7799bcl72 mount_ns]# hostname test-hostname
[root@lbf7799bcl72 mount_ns]# hostname
test-hostname
[root@lbf7799bcl72 mount_ns]#  cat /proc/sys/kernel/hostname
test-hostname
```

*Figure 8. Demo of setting up new UTS namespace*

## 2.5 User namespaces

User namespace is a way for a set of isolated processes to have a different set of permissions than the host system. It is integral in running containers as a non-root user on hosts system, while process is allowed to run as a privileged user inside a container. This removes a security risk, as breaking out of the container means having non-root access to the host machine. [2]

In CLI we can see that certain files belong to a certain user with a certain username – but to a computer that is simply a string that is mapped to a certain UID. Usually, unprivileged user UID are numbered 1000 or larger, and that UID is applied to the metadata of every file owned by that user.

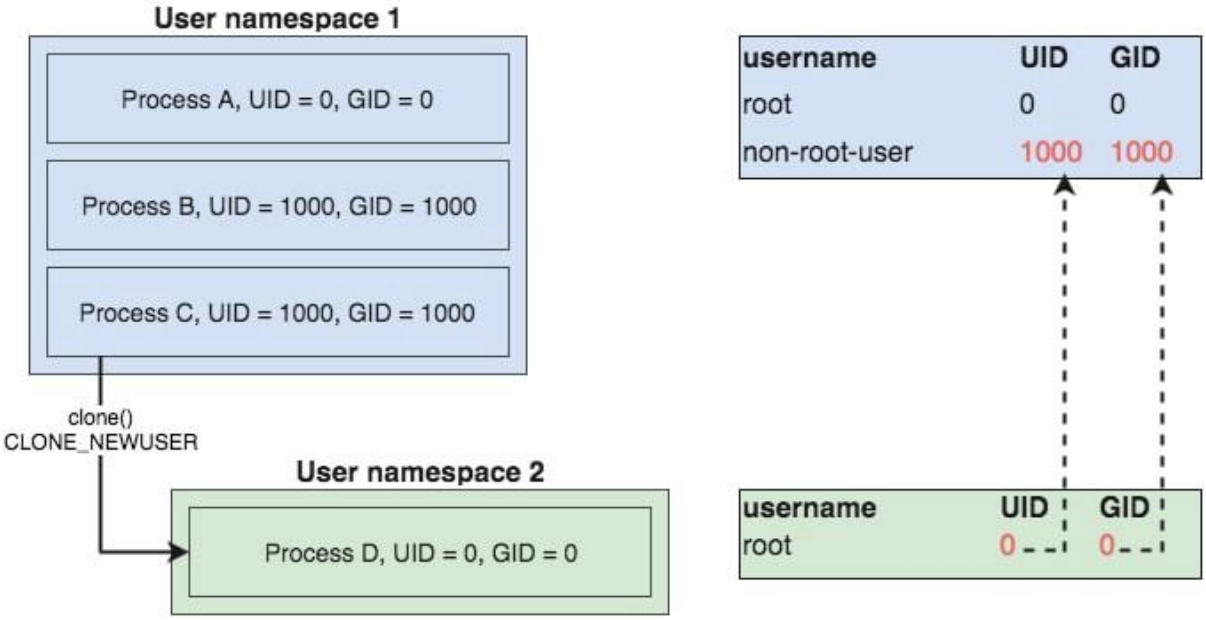


*Figure 9. User namespace representation [5]*

For the most part, user namespaces function similarly to PID namespaces – it is a way for the container to have a different set of permissions than the host system. User namespace will inherit permissions from a user who created it, which doesn't necessarily have to be a privileged user on the host system. User can see and interact with all files created by namespaces that have spawned from its UID, while reverse does not hold. User namespace governs all other namespaces – which means all namespace capabilities are determined by capabilities of its parent user namespace – so we can finely tune which processes can access and edit files, simply based on their user namespace.

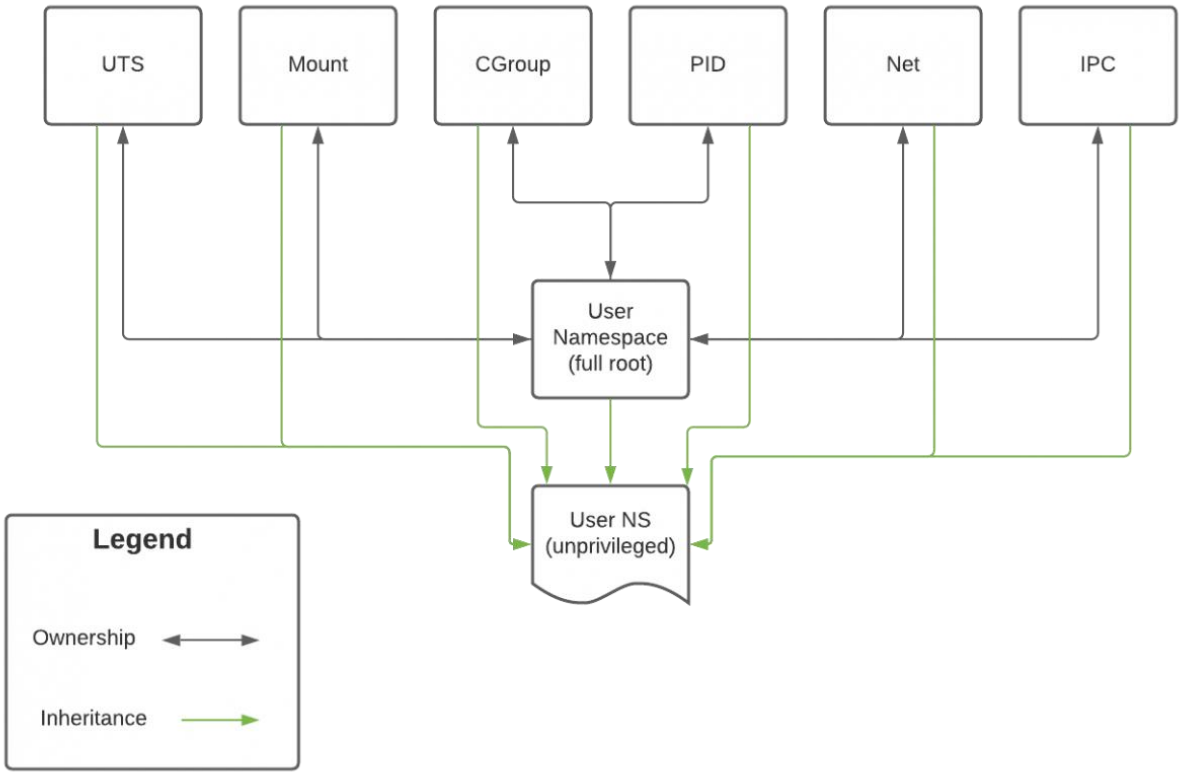


*Figure 10. Ownership and inheritance graph for namespaces[6]*

```
[root@1bf7799bc172 mount_ns]# su - danilo
[danilo@test-hostname ~]$ whoami
danilo
[danilo@test-hostname ~]$ PS1='\u@danilotest$ ' unshare -Ur
root@danilotest$ cat /proc/$$/uid_map
         0       1000          1
```

*Figure 11. Creating and entering a new user namespace created by an unprivileged user*

In Figure 9. the first number shows us that we are root user (uid of root user is 0) inside newly created namespace but that we are a regular unprivileged user in the parent namespace. Nesting another user namespace would result in:

```
root@danilotest$ PS1='\u@danilotest2$ ' unshare -Ur
root@danilotest2$ cat /proc/$$/uid_map
         0          0          1
```

*Figure 12. Nesting user namespaces*

Where we can see that we are root user in both child and parent namespace. If I try to install a package, I would need yum access, the system will not allow that:

```
root@danilotest2$ yum install -y iproute2
Loaded plugins: fastestmirror, ovl
Cannot open logfile /var/log/yum.log
ovl: Error while doing RPMdb copy-up:
[Errno 13] Permission denied: '/var/lib/rpm/.dbenv.lock'


[Errno 13] Permission denied: '/var/cache/yum/x86_64/7/.gpgkeyschecked.yum'
```

*Figure 13. Trying to install a package while inside a nested user namespace*

Since user who both of these user namespaces spawned from does not have permission to install packages via yum.

## 2.6 Network namespaces

Network namespaces provide isolation of the networking resources by making a logical copy of the network stack – allowing multiple processes to listen on the same port, as long as they don't share a network namespace. Like PID namespaces this is integral when we are trying to migrate containers from one machine to the other, as we don't have to re-map port mapping when migrating. For a short demo, we will use openvswitch and follow an example from [3]. By adding two network namespaces we can create a virtual bridge between them that would allow them to communicate:

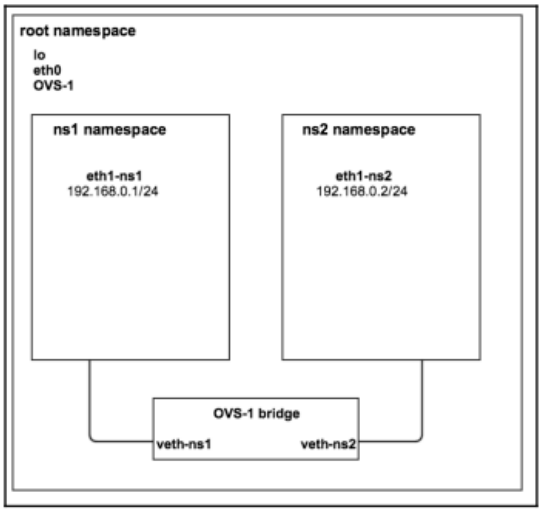


*Figure 14. Representation of bridge created by use of network namespaces [3]*

After creating two namespaces inside the host (in my case official openswitch image) with ''ip netns add namspace_name'' command we can see what interfaces are part of that namespace:

```
root@localhost:/# ip netns exec ns1 ip link
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN mode DEFAULT group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
```

*Figure 15. Newly created namespaces only have their loopback interface*

After creating virtual pair of interfaces they are automatically allocated to the default network namespace:

```
root@localhost:/# ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP mode DEFAULT group default qlen 1000
    link/ether 08:00:27:42:cd:7c brd ff:ff:ff:ff:ff:ff
3: enp0s8: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP mode DEFAULT group default qlen 1000
    link/ether 08:00:27:8d:50:12 brd ff:ff:ff:ff:ff:ff
4: virbr0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN mode DEFAULT group default qlen 1000
    link/ether 52:54:00:14:52:64 brd ff:ff:ff:ff:ff:ff
5: virbr0-nic: <BROADCAST,MULTICAST> mtu 1500 qdisc fq_codel master virbr0 state DOWN mode DEFAULT group default qlen 1000
    link/ether 52:54:00:14:52:64 brd ff:ff:ff:ff:ff:ff
6: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN mode DEFAULT group default
    link/ether 02:42:08:45:eb:02 brd ff:ff:ff:ff:ff:ff
7: ovs-system: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT group default qlen 1000
    link/ether 16:d5:ed:06:bb:d3 brd ff:ff:ff:ff:ff:ff
8: OVS-1: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT group default qlen 1000
    link/ether 3a:b5:e0:2e:1e:49 brd ff:ff:ff:ff:ff:ff
9: veth-ns1@eth1-ns1: <BROADCAST,MULTICAST,M-DOWN> mtu 1500 qdisc noop state DOWN mode DEFAULT group default qlen 1000
    link/ether 02:bc:31:81:da:74 brd ff:ff:ff:ff:ff:ff
10: eth1-ns1@veth-ns1: <BROADCAST,MULTICAST,M-DOWN> mtu 1500 qdisc noop state DOWN mode DEFAULT group default qlen 1000
    link/ether be:1a:a1:f0:84:4f brd ff:ff:ff:ff:ff:ff
11: veth-ns2@eth1-ns2: <BROADCAST,MULTICAST,M-DOWN> mtu 1500 qdisc noop state DOWN mode DEFAULT group default qlen 1000
    link/ether 2e:0e:07:54:6e:67 brd ff:ff:ff:ff:ff:ff
12: eth1-ns2@veth-ns2: <BROADCAST,MULTICAST,M-DOWN> mtu 1500 qdisc noop state DOWN mode DEFAULT group default qlen 1000
    link/ether 0a:7c:84:f3:66:02 brd ff:ff:ff:ff:ff:ff
```

*Figure 16. Default network namespace state after creating virtual interfaces*

Only after explicitly setting them to a new namespace we can divorce them from the default namespace, after giving interfaces in new namespaces IPs – they are able to communicate:

```
root@localhost:/# ip netns exec ns2 ip link set dev lo up
root@localhost:/#  ip netns exec ns2 ip link set dev eth1-ns2 up
root@localhost:/#  ip netns exec ns2 ip address add 192.168.0.2/24 dev eth1-ns2
root@localhost:/# ip netns exec ns2 ip a s
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
12: eth1-ns2@if11: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
    link/ether 0a:7c:84:f3:66:02 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 192.168.0.2/24 scope global eth1-ns2
       valid_lft forever preferred_lft forever
    inet6 fe80::87c:84ff:fef3:6602/64 scope link
       valid_lft forever preferred_lft forever
```

*Figure 17. Setting up a namespace for this demo*

```
root@localhost:/# ip netns exec ns1 ping -c 3 192.168.0.2
PING 192.168.0.2 (192.168.0.2) 56(84) bytes of data.
64 bytes from 192.168.0.2: icmp_seq=1 ttl=64 time=0.022 ms
64 bytes from 192.168.0.2: icmp_seq=2 ttl=64 time=0.031 ms
64 bytes from 192.168.0.2: icmp_seq=3 ttl=64 time=0.036 ms
```

*Figure 18. Pinging from ns1 to ns2*

This is integral for the operation of containers, especially when we are using it to host web applications since a network configuration of a web application is very different from a network configuration of a regular machine. Setting them up in the same namespace would open us up to unnecessary risks.

## 3. Control groups (cgroups)

In the previous chapter, we successfully restricted visibility and access to resources in order to facilitate safe and convenient running of processes – but even superficial knowledge of container technology tells us that that is not everything runtimes like Docker offer. What happens if one process is badly programmed and ''falls into'' an infinite loop that will consume a basically all system resources? If we are using only namespaces mentioned above, it is possible that the system will have to use last lines of defense, such as Out Of Memory Killer, to ensure that kernel has enough resources for basic functioning. Using control groups, we can control processes on our machine in a more centralized way. To illustrate the function of a basic example I will try and freeze one terminal session on an Ubuntu machine.

First, we need to create a folder where artifacts of the control group will be stored and then mount control group v2 to that folder [7]:

```
root@danilo-VirtualBox:/testcg# mkdir mygrp
root@danilo-VirtualBox:/testcg# cd mygrp/
root@danilo-VirtualBox:/testcg/mygrp# cd ..
root@danilo-VirtualBox:/testcg# mount -t cgroup2 none mygrp
```

*Figure 19. Creating and mounting a control group*

```
root@danilo-VirtualBox:/testcg/mygrp# ll
total 4
dr-xr-xr-x  5 root root    0 jan 31 22:32 ./
drwxr-xr-x  3 root root 4096 jan 31 23:40 ../
-r--r--r--  1 root root    0 jan 31 23:40 cgroup.controllers
-rw-r--r--  1 root root    0 jan 31 23:40 cgroup.max.depth
-rw-r--r--  1 root root    0 jan 31 23:40 cgroup.max.descendants
-rw-r--r--  1 root root    0 jan 31 22:32 cgroup.procs
-r--r--r--  1 root root    0 jan 31 23:40 cgroup.stat
-rw-r--r--  1 root root    0 jan 31 23:40 cgroup.subtree_control
-rw-r--r--  1 root root    0 jan 31 23:40 cgroup.threads
-rw-r--r--  1 root root    0 jan 31 23:40 cpu.pressure
-r--r--r--  1 root root    0 jan 31 23:40 cpu.stat
drwxr-xr-x  2 root root    0 jan 31 22:36 init.scope/
-rw-r--r--  1 root root    0 jan 31 23:40 io.pressure
-rw-r--r--  1 root root    0 jan 31 23:40 memory.pressure
drwxr-xr-x 90 root root    0 jan 31 23:38 system.slice/
drwxr-xr-x  3 root root    0 jan 31 22:32 user.slice/
```

*Figure 20. FIles inside a newly created control group*

By simply making a folder inside a cgroup mounting point we can make a child cgroup. Now by opening a new terminal and discovering its PID we can add process to tasks of that cgroup and control it from there:

```
root@danilo-VirtualBox:/testcg/mygrp# cd child/
root@danilo-VirtualBox:/testcg/mygrp/child# ll
total 0
drwxr-xr-x 2 root root 0 jan 31 23:41 ./
dr-xr-xr-x 6 root root 0 jan 31 23:41 ../
-r--r--r-- 1 root root 0 jan 31 23:41 cgroup.controllers
-r--r--r-- 1 root root 0 jan 31 23:41 cgroup.events
-rw-r--r-- 1 root root 0 jan 31 23:41 cgroup.freeze
-rw-r--r-- 1 root root 0 jan 31 23:41 cgroup.max.depth
-rw-r--r-- 1 root root 0 jan 31 23:41 cgroup.max.descendants
-rw-r--r-- 1 root root 0 jan 31 23:41 cgroup.procs
-r--r--r-- 1 root root 0 jan 31 23:41 cgroup.stat
-rw-r--r-- 1 root root 0 jan 31 23:41 cgroup.subtree_control
-rw-r--r-- 1 root root 0 jan 31 23:41 cgroup.threads
-rw-r--r-- 1 root root 0 jan 31 23:41 cgroup.type
-rw-r--r-- 1 root root 0 jan 31 23:41 cpu.pressure
-r--r--r-- 1 root root 0 jan 31 23:41 cpu.stat
-rw-r--r-- 1 root root 0 jan 31 23:41 io.pressure
-rw-r--r-- 1 root root 0 jan 31 23:41 memory.pressure
root@danilo-VirtualBox:/testcg/mygrp/child# echo 37792 > cgroup.subtree_control
bash: echo: write error: Invalid argument
root@danilo-VirtualBox:/testcg/mygrp/child# echo 37792 > cgroup.procs
root@danilo-VirtualBox:/testcg/mygrp/child# echo 1 > cgroup.freeze
```

*Figure 21. Child cgroup folder*

When value inside file cgroup.freeze is 1, all tasks in that group are frozen. Freezing is useful when we want to temporarily stop the process – either because it is working unexpectedly, we want to migrate it or we want to save a checkpoint.

## 3.1 Controllers

A cgroup controller is usually responsible for distributing a specific type of system resource along the hierarchy, although there are utility controllers that serve other purposes. When referring to cgroup v1, controllers are often called subsystems. Linux systems offer the following controllers [8]:

1. blkio (called ''io'' in version 2) – sets limits on input/output access to and from block devices (HDD, SSD,...)
2. cpuset – this controller assigns individual CPUs and memory nodes to tasks in a group
3. cpu – schedules access to CPU for tasks in a cgroup
4. cpuacct – Generates automatic reports on CPU resources used by a task. In cgroups version 2 this controller was combined with cpu controller into one controller.
5. freezer – is utility type controller that suspends or resumes tasks in a group
6. memory – sets limits on memory use by tasks and generates an automatic report on resources used by tasks

7. net_cls – tags network packets with class identified that linux traffic controller uses to identify from which task did packets come
8. net_prio – allows to dynamically set the priority of network traffic per network interface. Worth noting that controllers 7 and 8 exist only in cgroup version 1 – for version 2 their function was replaced by expansion of iptables package [9]
9. devices – manages access to device files. Includes both creation of new files and access to existing device files.
10. rdma – regulates distribution and accounting of RDMA (Remote Direct Memory Access) resources
11. pid – allows cgroup to stop any new task from forked or cloned after a specified limit is reached
12. hugetlb – limits usage of hugetbl( Huge Page Tables), which is used with processes that require bigger pages when being allocated memory

## 3.2 Cgroups version 1

Cgroups (version 1) was version rolled out roughly around the year 2007 and is still often used in some container runtimes. Cgroups are a hierarchical model where child cgroups inherits certain attributes from parent groups. The difference to many other hierarchical models on Linux systems is that multiple hierarchies can exist on a system at the same time. One process can belong to multiple cgroups as long as those cgroups are not part of the same hierarchy. Multiple subsystems can be combined into a single hierarchy.

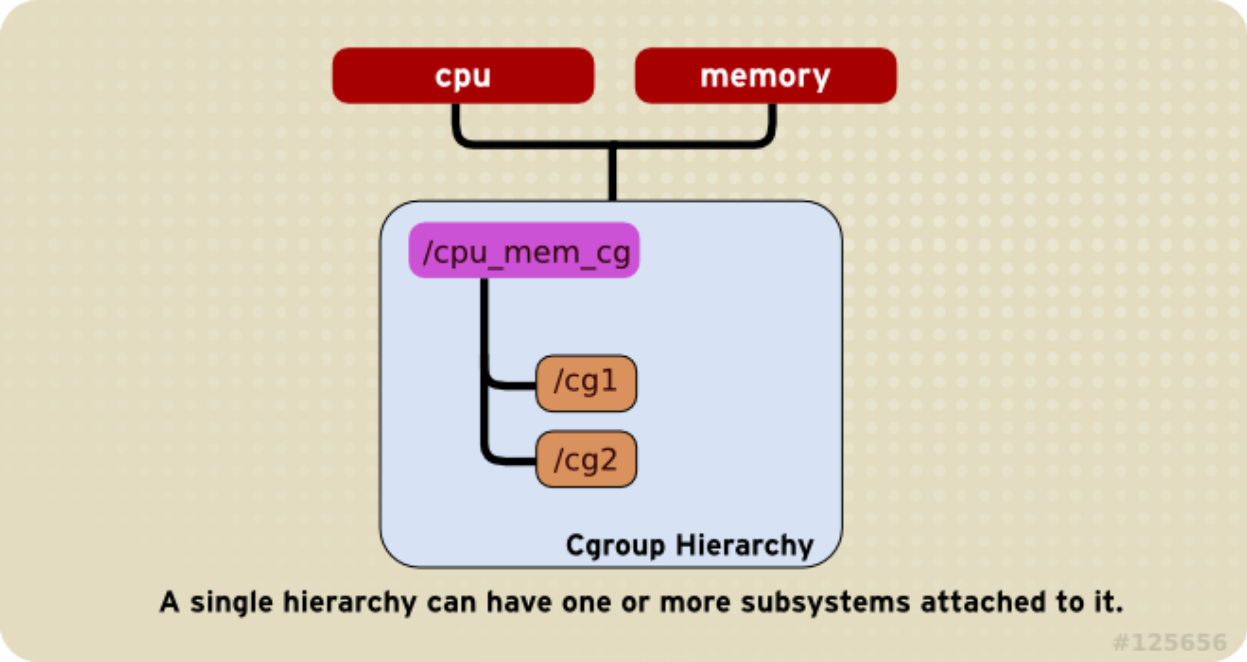


*Figure 22. Combining cpu and memory controllers [8]*

Processes are added to a cgroup by writing their pid into cgroups.procs file that is automatically created inside cgroup folder upon its creation. If the process forks itself during execution, child process inherits membership of a cgroup but can be moved to another cgroup without consequences. Once the process is spawned child and parent process are independent, from standpoint of cgroups. When we want to

combine subsystems into hierarchies we have to follow some rules, the most problematic of which is the following rule:

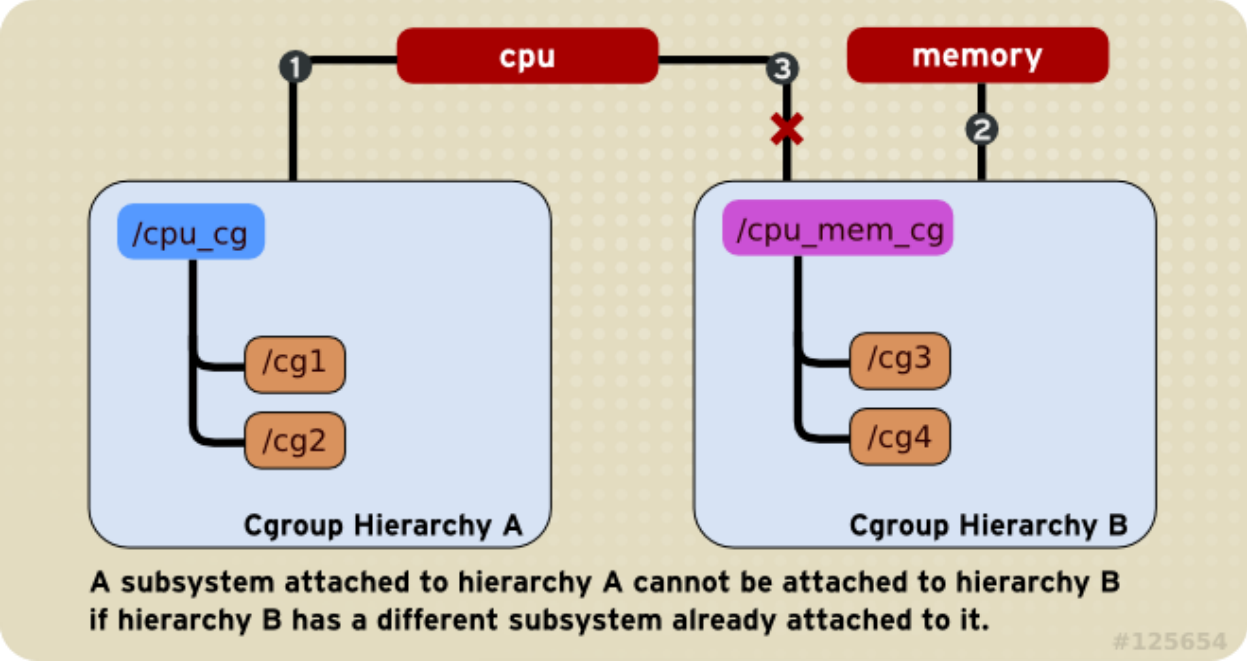


*Figure 23. Subsystem combination rule [8]*

This rule means that utility controllers (such as freezer) that would be useful in every hierarchy can only efficiently be used in one. This has led to developers generally just using separate hierarchies for every controller – which removed flexibility that combining subsystems offered but left all complications that were a result of allowing multiple hierarchies. In general hierarchies in version 1 would look like the following:

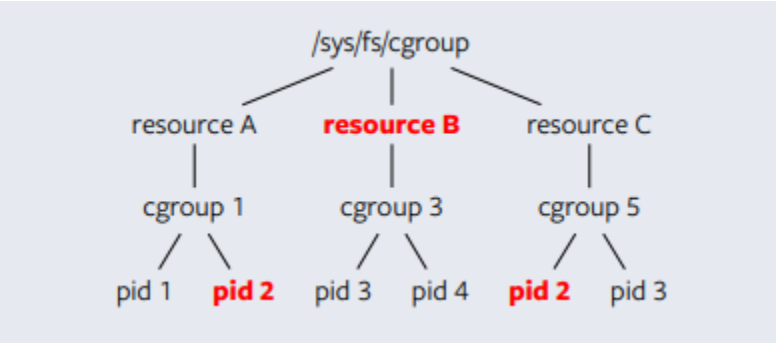


*Figure 24. Representation of hierarchies in the original version of cgroups [10]*

There we can see one process two groups in different hierarchies. Process by which we create cgroups for (as an example) cpu controller would be creating a folder inside a /sys/fs/cgroup/cpu location. That would create an empty cgroup to which we would add different processes.

Cgroup core (primarily responsible for hierarchically organizing processes) implementation was much more complicated in order to support combining subsystems. As a result of that, around 2013 work began on the second version of cgroups – which started getting widely used around early 2019.

## 3.2 Cgroups version 2

Main goal of new version of cgroups version 2 is to bring simplicity to hierarchies. Both versions can exist on the same machine, but that is not recommended as one resource can't be controlled by both cgroup1 and cgroup2.

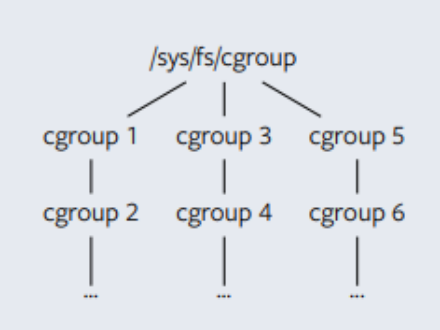


*Figure 25. Representation of cgroup2 hierarchy [10]*

We create a new cgroup by creating a new folder inside /sys/fc/cgroup folder and that creates empty cgroup2 [11]. We determine which controllers that group has access to by adding or removing those controllers in cgroup.controllers file in that cgroup.

```
root@danilo-VirtualBox:/sys/fs/cgroup/test# cat cgroup.controllers
cpuset cpu io memory hugetlb pids rdma misc
```

*Figure 26. Cgroup2 with access to various controllers*

Since I simply created a folder inside the root cgroup I have all controllers available. If I were to create sub-cgroup inside that one, I would see that that group has no controllers available. Nested hierarchies can only have access to controllers that are stated in their parent cgroup.subtree_control file.

```
root@danilo-VirtualBox:/sys/fs/cgroup/test/child_test# cat cgroup.controllers
root@danilo-VirtualBox:/sys/fs/cgroup/test/child_test# cd ..
root@danilo-VirtualBox:/sys/fs/cgroup/test# echo "+cpu +memory" > cgroup.subtree_control
root@danilo-VirtualBox:/sys/fs/cgroup/test# cd child_test/
root@danilo-VirtualBox:/sys/fs/cgroup/test/child_test# cat cgroup.controllers
cpu memory
```

*Figure 27. Giving subgroup access to a controller*

Adding a controller to a group will affect every cgroup belonging to that's group sub-hierarchy. Let's examine a simple example. Group A has access to cpu and memory controller, and has allowed child cgroup B to have access to memory controller. Group B then controls 2 processes.

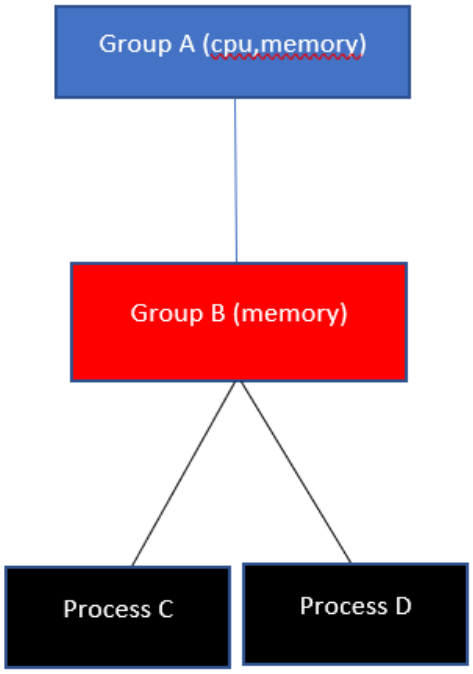


*Figure 28. Simple cgroup hierarchy*

Group A will control the distribution of CPU cycles and memory to group B. Since group B does not access cpu controller – processes C and D will compete freely on CPU cycles (that are available to this whole sub-hierarchy) but group B will control the division of memory available. Group B can't be overridden by group B – groups lower in the hierarchy can't grant more memory to their sub-groups (or processes) than the group higher in the hierarchy has granted them (or their ancestor) [12]. That is how we can make sure that Docker container processes will not use more resources than we have granted to that container. To avoid situations where child cgroups compete for resources against the internal processes of their parent, cgroups can only control the distribution of resources to children groups when they have no processes of their own. In practice, this means that only cgroups on the leaves of the hierarchy can control processes. In our example, group A can't control any internal process since it controls group B.

Cgroups2 allow delegating control of cgroups to less privileged users by granting them write access over a directory of a subgroup. This allowed the deployment of rootless containers, this was not possible with cgroup1 since giving such access to a non-privileged user was considered dangerous.

## 4. Creating a simple container

While my seminar is mostly based on researching the theory behind containers – it could be useful to make a basic alpine container to show how we can run a different linux distro on our host system while being mostly separate from it and draw a few parallels to how docker containers are run. Code for a basic container is not very complicated and mostly involves various system calls that are standardized, for the most part, I will be combining snippets from books and guides I was using to research the topic.

Program is written in Go language – which is a compiled programming language designed by Google in 2009. Go is syntactically similar to C and deeper understanding of Go is not needed to follow this chapter – even writing the code required relatively shallow knowledge as the tricky part of making a container is understanding the theory behind them rather than writing actual code. The goal of this container is to run a shell in which we can host a simple http server and do small tests. To do that we would need to install python inside the container and establish a network between the host machine and network namespace of our container.

First thing we need to do is create namespaces that we will eventually populate. I will do that inside function called run that will eventually call child function in which most of our setup will be done[5].

```
cmd.Stdin = os.Stdin
cmd.Stdout = os.Stdout
cmd.Stderr = os.Stderr
cmd.SysProcAttr = &syscall.SysProcAttr{
    Cloneflags: syscall.CLONE_NEWNS |
        syscall.CLONE_NEWUTS |
        syscall.CLONE_NEWIPC |
        syscall.CLONE_NEWPID |
        syscall.CLONE_NEWNET |
        syscall.CLONE_NEWUSER,
    Unshareflags: syscall.CLONE_NEWNS,
    // mapping user and groups to be root insite the container
UidMappings: []syscall.SysProcIDMap{
        {
            ContainerID: 0,
            HostID: os.Getuid(),
            Size: 1,
                },
                },
GidMappings: []syscall.SysProcIDMap{
                                    {
                    ContainerID: 0,
                            HostID: os.Getgid(),
                            Size: 1,
                            },
                        },
                }
```

*Figure 29. Creating new namespaces and adding user/group mapping*

We create new namespaces via system calls – interesting fact that originally mount namespace was named just namespace. It was first created namespace and creators probably thought no other will be needed. Under clone flags, I have one unshare flag which does the same thing as unshare function used when I was discussing mount namespaces. It says that the mount namespace within the container is not to be shared with the host.

What we have created so far is just a very limited user on our host – what we need to do now is download a filesystem. As alpine is very popular as a minimalist system on container technologies, I opted for that distribution. I placed it in a /root/alpine/rootfs location. We can mount the location of alpine binaries with command [13]:

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

*Figure 30. Mounting a directory*

Next task is to pivot the root of that namespace to the location of our binaries (so we can't escape the container and have access to files that should be isolated from this container). This turned out to be relatively tricky as there are a few rules that need to be followed [14]:

1. Both old and new root have to be directories
2. Place where we will move files of old root after pivoting to new root has to be underneath new root
3. We can't move old root to a place where some other namespace is mounted
4. New root and old root copying places can't be on the same filesystem as the current root.

Last requirement caused some problems, as I would get unexpected crashes. My best guess is that kernel checks not filesystems but mounts for equality:

```
    if (new_mnt == root_mnt || old_mnt == root_mnt)
            goto out4; /* loop, on the same file system  */
```

*Figure 31. Relevant snippet of Linux source code*

I got around this problem by bind-mounting new root to something (in my example, itself) – which allowed me to pivot to my system location. If I understood their code correctly this is similar to how LXC container runtime pivots root directory of their containers. With that finished we now have a relatively decent level of separation from the host machine – it is not possible to just wander out of the container.

```
/ # ls
bin          etc          home         media        opt          root         sbin         sys          tmp          var
dev          forkbomb.sh  lib          mnt          proc         run          srv          testll       usr
/ # hostname
DaniloAlpineContainer
```

*Figure 32. Look inside of a container with a new root directory*

Next, we should limit what resources can this container use. As I already created cgroups in example for hierarchies, we can use those same cgroups to limit resources this container can use. As a proof of concept pid and memory controller will be good enough. First, we need to add pid controller to child_test cgroup via commands shown in the cgroups chapter. Cgroups are controlled by the host system so it is important that they appear in code before we pivot the root.

```
 // using cgroup created in previous example to limit memory usage of our process
 parent_cgroup := "/sys/fs/cgroup/test"
 leaf := filepath.Join(parent_cgroup, "child_test")
 // Limiting cgroup to 10 processes
 must(ioutil.WriteFile(filepath.Join(leaf, "pids.max"), []byte("10"), 0700))
 // Limiting this cgroup to 10 Mb of memory
 must(ioutil.WriteFile(filepath.Join(leaf, "memory.max"), []byte("10M"), 0700))
 // adding this process to cgroup
 must(ioutil.WriteFile(filepath.Join(leaf, "cgroup.procs"),[]byte(strconv.Itoa(os.Getpid())), 0700))
```

*Figure 33. Relevant code snippet*

As an example of advantage that limiting process resources gives us I will do a fork bomb (type of ddos attack) inside of my container. Code of fork bomb is very simple:

```
fork() {
        fork | fork &
                }
fork
```

```
/ # vi forkbomb.sh
/ # sh forkbomb.sh
/ # forkbomb.sh: line 2: can't fork: Resource temporarily unavailable
forkbomb.sh: forkbomb.sh: line 2: can't fork: Resource temporarily unavailable
line 2: can't fork: Resource temporarily unavailable
^C
/ # ps
/bin/sh: can't fork: Resource temporarily unavailable
/ # ^C
/ # ps
/bin/sh: can't fork: Resource temporarily unavailable
```

*Figure 34. Container running out of allowed processes*

As we see, the container instantly runs out of pids as there can only be 10 active processes at one time. Memory would be limited for similar purposes as containers could require too much memory either due to an attack or bug in the service that runs on it.

```
/ # ps -ef
PID   USER     TIME  COMMAND
    1 root      0:00 /proc/self/exe child /bin/sh
    6 root      0:00 /bin/sh
   13 root      0:00 ps -ef
```

*Figure 35. Container can't see processes running outside of it*

Right now, we have a container that can run at a sufficient level and won't interfere with our host system by hoarding critical resources – but it still can't communicate with the outside world (or host system) via network connections since it belongs to an effectively empty network namespace. The easiest way to connect it to the internet is to create a bridge between it and the host device, connect it via veths and make sure that all traffic from the container is going through that bridge [15]. To accomplish that I will use netsetgo library. Since we need to both configure host and container – we will have to run parts of this code in both run and child functions. Also, we will have to configure iptables on the host with the following commands:

```
iptables -tnat -N netsetgo
iptables -tnat -A PREROUTING -m addrtype --dst-type LOCAL -j netsetgo
iptables -tnat -A OUTPUT ! -d 127.0.0.0/8 -m addrtype --dst-type LOCAL -j netsetgo
iptables -tnat -A POSTROUTING -s 10.10.10.0/24 ! -o brg0 -j MASQUERADE
iptables -tnat -A netsetgo -i brg0 -j RETURN
```

as well as enabling ip forwarding:

```
echo 1 > /proc/sys/net/ipv4/ip_forward
```

Now by running netsetgo with the correct pid of our child process we will set up a bridge and give our process IP address.

```
/ # ping google.com
PING google.com (142.250.201.206): 56 data bytes
64 bytes from 142.250.201.206: seq=0 ttl=56 time=14.247 ms
64 bytes from 142.250.201.206: seq=1 ttl=56 time=14.775 ms
64 bytes from 142.250.201.206: seq=2 ttl=56 time=13.583 ms
64 bytes from 142.250.201.206: seq=3 ttl=56 time=14.983 ms
64 bytes from 142.250.201.206: seq=4 ttl=56 time=21.807 ms
64 bytes from 142.250.201.206: seq=5 ttl=56 time=21.791 ms
```

*Figure 36. Container succesfuly pings google.com*

With that, we are now able to install python and run a http server on our container. This container is now functional and can be upgraded from within itself – but it is still a very basic version of a container that has massive drawbacks compared to optimized runtimes such as docker. The biggest drawback is that I can't just migrate (or copy it via images) and migrate it to another machine – it is tailored specifically for my machine, and it would take some work to move it somewhere else.

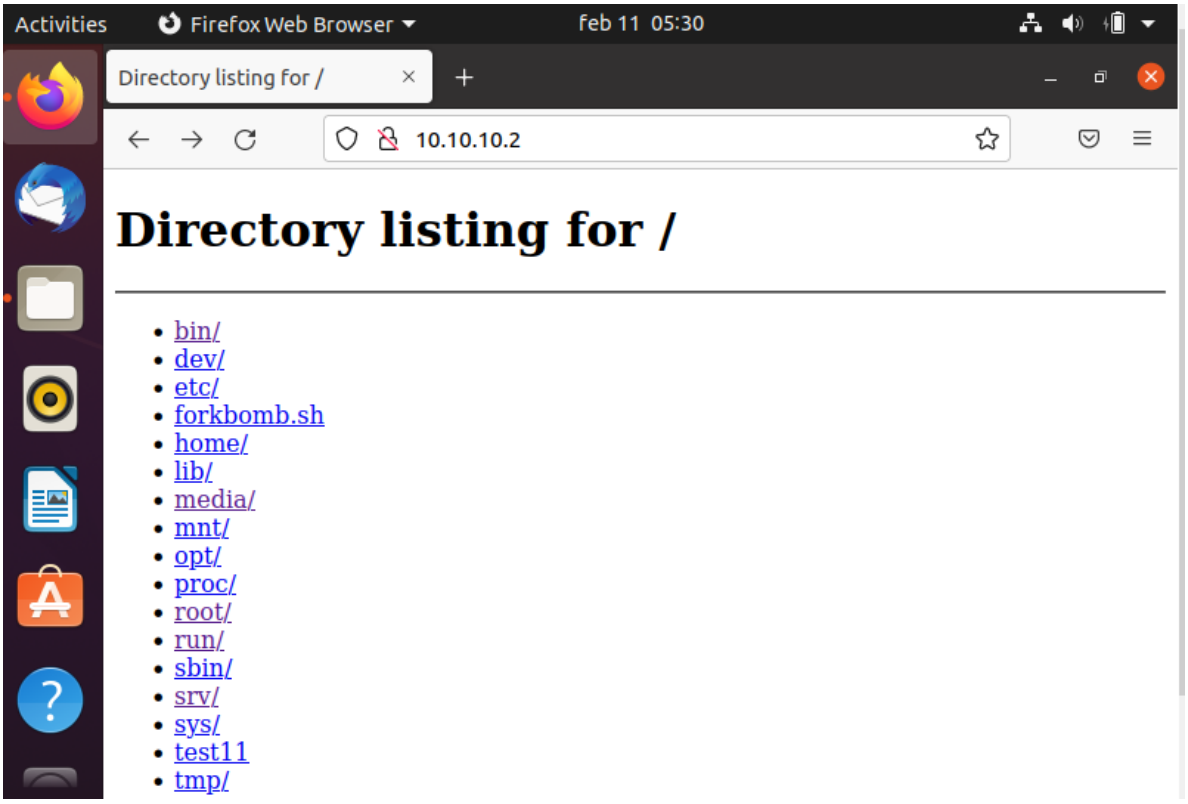


*Figure 37. Http server is visible from the host machine*

Also, if we were to run multiple instances of this container on a machine it would take up a lot of storage space, as for every container we have we would need to copy the whole file system. In such situations, Docker would take advantage of overlay file system and employ copy-on-write strategy.

Copy-on-write is a way of sharing and copying files for maximum efficiency. When I downloaded tarball of alpine I extracted it and worked with it- effectively doubling its size. When I build an image or run a container in docker it adds a thin writeable layer on top of the currently active layer, every layer not in use is read-only. Effectively, instead of copying, containers simply mount the image. When the existing file in a read-only layer is modified, the storage device starts copy-on-write strategy. It will search through layers for which file to modify (from oldest to newest layer) and then performs a copy_up operation [16]. Copy_up operation copies modified file to writeable layer and container would then use that modified file from that point on. So, layers of docker images only store changes from previous layers.

Multiple instances of the same container can read files they do not modify from the same place. With that said, this is not recommended for files that write in a lot of different files because copy_up function has a lot of overhead. Such processes (like databases) should probably be running as volumes that are independent of the running container and are designed to be efficient for I/O.

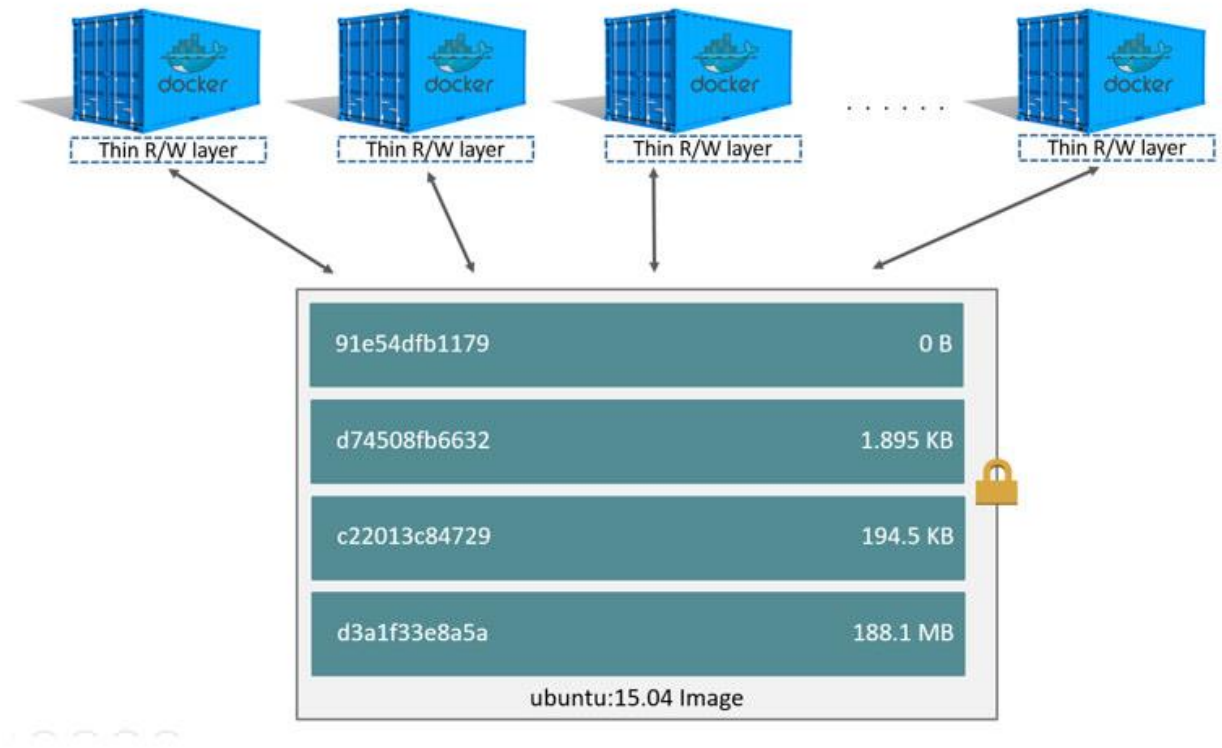


*Figure 38. Multiple docker containers using same read only layers of an image*

This system has had a large part in why Docker has spread so much, as it combined ease of use (setting up a basic web server takes only one command) and it is more efficient than something even a skilled programmer can make in a reasonable amount of time.

## 5. References


[1] Philipp Schmied, Linux Container primitives: An Introduction to namespaces; 29.10.2019; https://www.schutzwerk.com/en/43/posts/namespaces_01_intro/

[2] Steve Ovens; Building containers by hand: The PID namespace; Redhat; 30.4.2021; https://www.redhat.com/sysadmin/pid-namespace

[3] Konstantin Ivanov; Containerization with LXC; Packt publishing; 2017

[4] Steve Ovens; The 7 most used linux namespaces; Redhat; 11.1.2021; https://www.redhat.com/sysadmin/pid-namespace

[5] Ed King; Namespaces in Go - User; Medium; 13.12.2016; https://medium.com/@teddyking/namespaces-in-go-user-a54ef9476f2a

[6] Steve Ovens; Building a Linux container by hand using namespaces; Redhat; 8.3.2021; https://www.redhat.com/sysadmin/building-container-namespaces

[7] Shashank Mohan Jain; Linux Containers and Virtualization: A Kernel Perspective; Apress; 2020

[8] Resource Management Guide: Chapter 1 Introduction to control groups; Redhat; https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/ch01

[9] Linux manual page – cgroups(7); Linux; 27.08.2021; https://man7.org/linux/man-pages/man7/cgroups.7.html

[10] Chris Down; cgroupv2: Linux's new unified control group system; Facebook; https://chrisdown.name/talks/cgroupv2/cgroupv2-fosdem.pdf

[11] Tejun Heo; Linux documentation: cgroup-v2; 2015 https://www.kernel.org/doc/Documentation/cgroup-v2.txt

[12] Creating and organizing cgroups; Facebook; 2019 https://facebookmicrosites.github.io/cgroup2/docs/create-cgroups.html

[13] Liz Rice; Containers from Scratch; Gotocon; 2018 https://gotoams.nl/2018/sessions/429/containers-from-scratch

[14] Linux manual page: pivot_root(2); Linux; 22.03.2021 https://man7.org/linux/man-pages/man2/pivot_root.2.html

[15] Ed King; Namespaces in Go – Network; Medium; 9.1.2017; https://medium.com/@teddyking/namespaces-in-go-network-fdcf63e76100

[16] Docker documentation: About storage drivers; Docker; 2021 https://docs.docker.com/storage/storagedriver/

[17] Michael Kerrisk; Namespaces in operation, part 4: more on PID namespaces; LWN; 23.1.2013; https://lwn.net/Articles/532748/