Institut Polytechnique de Paris

M2 Parallel and distributed systems

Logo

Description automatically generated

Cloud Infrastructures CS5004

**SIMPLE CONTAINER ENGINE**

Authors:  
Jana Toljaga  
Timothée Zerbib  
Aleksandar Maksimovic  
Dimitrije Panic

Content

[1 Introduction 3](#_Toc125311371)

[2 Container image 3](#_Toc125311372)

[3 Namespaces 3](#_Toc125311373)

[4 Control groups 7](#_Toc125311374)

[5 Overlay filesystem 8](#_Toc125311375)

[6 Networking 9](#_Toc125311376)

# Introduction

Containers are a lightweight form of virtualization at the level of the operating system. Container engines can run multiple, isolated containers, and manages their lifecycle on the same operating system kernel.

The main goal of this lab was to learn how to implement a simple container engine that will provide isolation and constrain resources available for containers, leveraging namespaces and cgroups. Additionally, we’ve been able to setup networking and overlay file system into our container engine.

In the next chapters, we will present our solution as well as answers to all asked questions.

# Container image

*[Q1]* What is the role of a container image? How do you think you will use it in this lab when creating a container?

A container image is an executable code packed with its runtime so it can run an isolated process. It contains system libraries and tools needed to run a program on some containerization platform, such as Docker. A docker image consists of multiple layers each of which corresponds to a file system. This structure increases reusability of various components, so users don’t need to create everything from a scratch.

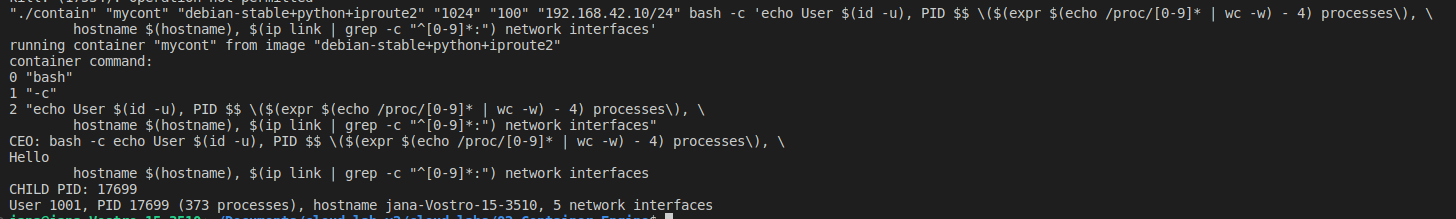
In the lab, we are using a Debian Docker image tat will serve as a base image in our container. The goal is to run a bash command in a containerized environment and Debian image will provide a source code of the most of the commands available in Debian instalation.

# Namespaces

In this section we will take a closer look at the implementation of our container engine step by step and explain everything that is necessary to build a container.

* 1. *Warmup: print and execute the containerized command*

In this section we printed the command specified through command line arguments and executed it natively using execvp syscall. We received the following output:

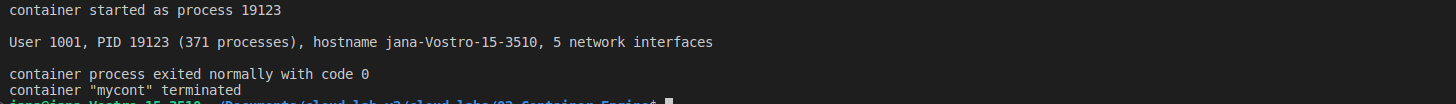


*[Q2]* Identify which namespaces are tested by this command. The command calls bash: where is the executed binary located?

Looking at the output above, echo is used to test four namespace dimensions: user namespace, pid namespace, hostname namespace and networking namspace. As we still haven’t created a container, echo command is executed natively on our host machine, therefore, a binary is located on our host machine at /*usr/*bin/bash.

* 1. *Execute the command in a subprocess*

Now, the goal is to create a subprocess calling clone3 syscall. At this point no cgroups or namespaces are specified. The output we received is shown below:

*[Q3]* What happens to the execution flow of your process when calling clone?

Clone system call will create a child process which runs concurrently with the parent process. Both processes will continue execution from the instruction after the clone. The return value is used to distinguish two processes and it is equal to 0 for the child process and child pid for the parent process. Child process will be the one executing our echo command (PID part of the output changed).

* 1. *User namespace*

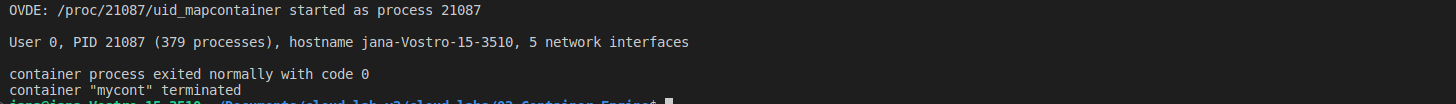
At this point we are configuring namespaces for our child process that will be used as a container process. This is done using CLONE\_NAMESPACE\_FLAGS to initialize flags field in clone\_args.

User namespace is used to isolate user IDs, so the process can have different user ID inside and outside the container. Confguring the clone flags field will create a new user namespace. Additionally, we needed to map the user ID to the root user inside the container writing that mapping in /proc/pid/uid\_map file.

*[Q4]* What change in the output do you expect when running the test command make run-namespaces?

We expected UID part of the output to change to 0. Echo command should be executed inside the container under the root user, even if outside the container that user is a non-root user.

We received exactly what we expected:



* 1. *Hostname namespace*

Similar to what we did before, a hostname namespace is created specifying a corresponding fag in clone\_args.flags. To set a new hostname it’s enough to call sethostname function inside the container.

*[Q5]* What change in the output do you expect when running the test command make run-namespaces? Why is the function finalize\_cont called from within the container?

We expected only the hostname part of the output to be changed to the one specified in command line arguments (mycont). The function finalize\_cont must be called from the container itself as root privileges are required for setting a hostname and our child process is running under the root user id. The sethostname function changes a hostname of the current process. The output can be seen below:

* 1. *PID namespace*

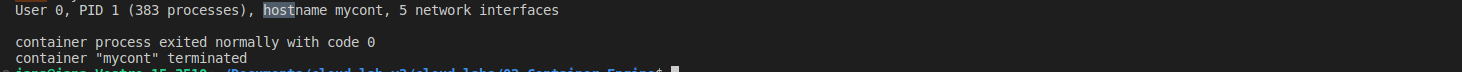
To configure PID namespace, we just added a new flag to clone\_args.flags, like before.

[Q6] Why is it necessary to have an “init” (PID 1) process in the container? Think of the special role of PID 1 in UNIX systems. What is the process with PID 1 in the container? What do you expect to see when running the test command? Read carefully the containerized test command, and explain.

In UNIX systems, the first process started by the operating system is given PID 1 and the same thing is happening when a new namespace is created. It is important for a couple of reasons:

* When the process with pid 1 die for any reason, all other processes are killed with KILL signal.
* When any process having children dies for any reason, its children are reparented to process with PID 1
* Many signals which have default action of Term do not have one for PID

Therefore, this process is the ancestor of all other processes in the container and it is, by default, used to take care of all orphaned processes and to forward signals to the host if necessary.



As for the output, only PID changed to 1. We also noticed that the number of process IDs in /proc directory is still equal to the number of processes in the host. This will be explained in chapter 3.7.

* 1. Network namespace

In this chapter we just added a flag to add a networking namespace, without any further configuration of network devices.

[Q7] What do you expect to see when running the test command?

After running the command we expected to see the change in number of network interfaces to 0 instead of 5. However, number of interfaces was 1.

After running ip link instruction from inside the container, we realised that 1 existing interface is a loopback interface (127.0.0.1) that is immediatelly created while creating the namespace. This interface is in down state by default, so ping to 127.0.0.1 would fail.

* 1. Mount namespace

Like for each other namespace, we specified the flag which corresponds to the mount namespace.

[Q8] You will not see any difference yet: why? Take the time to understand what the mount namespace isolates exactly.

Mount namespace isolates the list of mounts seen by the processes in each mount namespace instance. All processes in the new mount namespace will see all mounts inherited from the parent. However, if a child creates a new mount in a new mount namespace, a parent shouldn’t be able to see it. We still haven’t mounted anything in the child process, so the state remains the same.

Next, we created a container’s mountpoint and mounted container’s file system to it. We used a bind mount. A bind mount takes an existing directory tree and replicates it under a different point. The directories and files in the bind mount are the same as the original. It creates a reference, not a copy.

[Q9] Again, you will not see any difference yet: why? Take the time to understand what the test command does, and what is the current working directory of the container process.

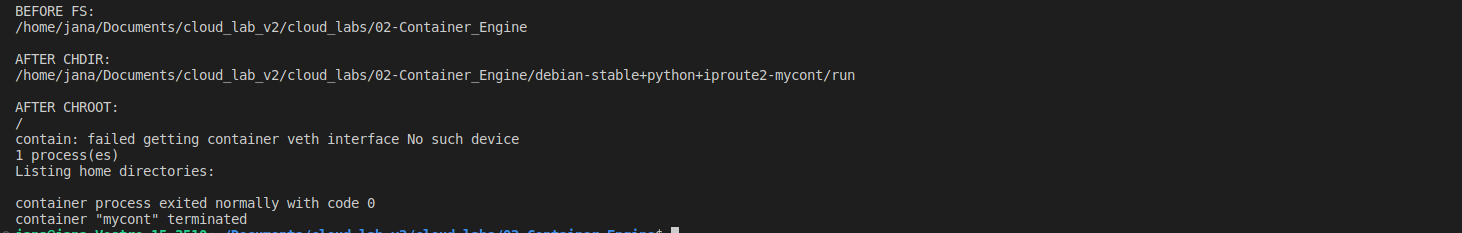
Here we just created needed directories and later a child process mounted container’s file system. However, we still haven’t changed neither a home nor a working directory and one of the things test command checks is the home directory, that is still the same as parent’s. Secondly, test command checks the number of process IDs in proc, and it is still the same as before.

Finally, we mounted a special file systems (proc among them) and changed home, working and root directory.

[Q10] Why is the container’s filesystem mounted from the host side, before creating the container subprocess? Why are the special filesystems proc, sysfs and tmpfs mounted from finalize\_cont, i.e., from the container side? What do you expect to see when running the test command?

Creating namespaces haven’t created new special file systems such as proc, sysfs and tmpf, but instead these special file systems are by default bind mounted from external parent’s / file system. This is why we were seeing so far host processes when counting process ids in /proc directory. It is necessary to mount these file systems explicitly in a container’s process because their content depends on the process that does mounting. For example, mounting /proc directory will fill proc with PIDs visible from the namespace that mounts it.

As a result, we expected to see only 1 process in /proc directory and listing home directories to be empty, as it is in the output below. We additionally printed pwd while changing working, home and root directory to see the difference.



# Control groups

In this chapter we will explain how we constrained resources, such as cpu and memory, using control groups and we will answare all asked questions.

* 1. Control group creation

To create a cgroup we created a directory under /sys/fs/cgroup/ that will present our cgroup. We gave write privileges only to the creator of the directory, parent process. Child process shouldn’t be able to change its resource limits. Later, we used clone arguments to set a file descriptor of that directory and added a flag CLONE\_INTO\_CGROUP.

[Q11] What do you expect to see when running the test command?

**We expected to see the cgroup that we created in the/proc/self/cgroup for a child process as an output of the test command. The real output we got is shown below.

* 1. Imposing limits to the cgroup

In this section, we specified limits for CPU and memory in our cgroup. To constrain memory we had to write a requested limit to a memory.high file in the cgroup. Similar procedure is done for cpu as well, changing a default cpu.max file. There we wrote the cpu period and cpu limit calculated from cpu period and given cpu percentage.

[Q12] How do you think the inability of reading the resource limits from inside the container can affect containerized applications? Give examples of programs and runtimes.

The inability to know the amount of memory or cpu that is available for containerized application can affect runtimes focused on memory or cpu management. For example, when Java application is executed inside the container, JVM needs to dynamically assign cpu and memory limits based on the amount of available resources. However, if JVM can’t get that information from the cgroup of the container, it is not possible for JVM to be configured to respect these constraints and recognize when program exceeds container’s limits.

# Overlay filesystem

In this chapter we will explain what the overlay filesystem is and how we configured it for our container, while answering a couple of questions.

[Q13] What is the container image in the container’s filesystem layers? How are filesystem layers used in container images? How filesystem layers are used when running a container? What happens when a containerized process writes to a file? Specifically, why is it slow for a containerized process to write to a file provided by its image?

Container images are base layers for container’s filesystem. Layers in a container image are read-only and reusable by other images built on top of that. It is possible to add a writable layer on top of underlying layers, which is considered a container’s layer. All writes to the container that add new or modify existing data are stored in this writable layer, using copy on write strategy. When the container is deleted, the writable layer is also deleted. The underlying image remains unchanged.

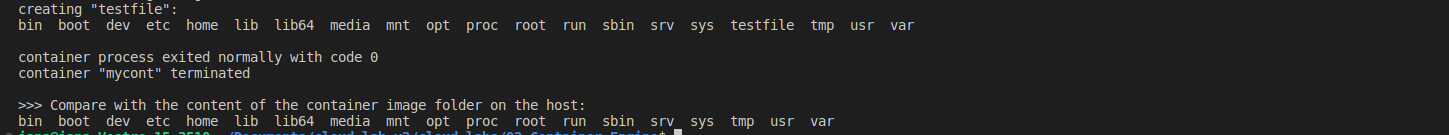
Copy on write technique is used by container engines to provide a fast boot of the container, but then a slow-down is visible on each first write, as file first needs to be copied to writable layer, before any modifications.

To enable an overlay filesystem, we just needed to mount the image filesystem of type overlay, specifying lowedir, upperdir and workdir parameters.

[Q14] How do the lowerdir, upperdir and mountpoint map to the concepts of a running container filesystem as seen in the lecture?

Lowerdir represents a base image layer, while upperdir corresponds to a container’s image layer. Both of them are read-only and all changes made in upper layers won’t affect them. A mountpoint is a running layer. It is writable and uses copy on write to modify lower layers.

The output we received running the corresponding command is below. It is obvious that testfile created inside the container doesn’t exist in the container image directory in the host.



# Networking

In this chapter we will explain how we configured networking for the container and how we accessed it via network. Networking here is used to access the container from the host. We used virtual ethernet devices for that purpose.

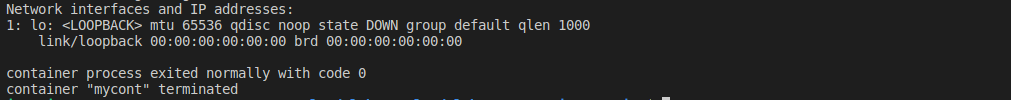
[Q15] How would you use a veth pair to provide networking to a container? The rough idea is to use it to “cross” the boundary of the network namespace of a container.

Once we configured a veth pair, the communication between the host and the container is established. Networking protocols could be implemented between the host (real network device) and someone else. All received messages could be routed from the host to the container through veth pair if the container is a destination.

* 1. Synchronization between container engine and container

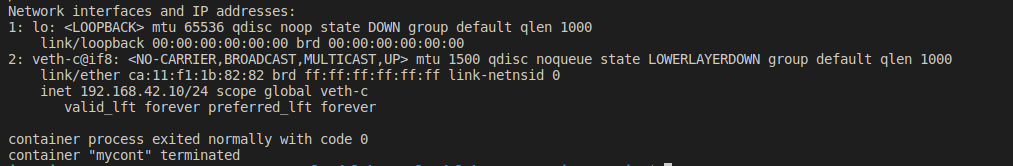
Before adding networking using veth pairs, we first set up a synchronization between the host and the container that is needed for configuring networking correctly. For synchronization purposes we used signals and signal handlers. What we wanted to accomplish with the help of synchronization is to have the veth pair created by the parent process after cloning the child process with net namespace, but before child specified its ip address. We used SIGUSR 1 to signal the waiting child process and activate a handler that sets a flag to unblock the container. Container waits in a loop for a flag, pausing in each iteration to avoid busy wait.

The output that we first produced is shown below. The only network interface we have so far is a default loopback interface.



* 1. Network configuration: virtual Ethernet

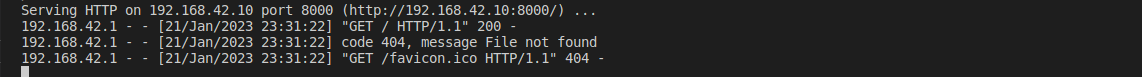
Firstly, to configure veth networking we created a veth pair, configured host end of the veth pair and put the other end to the guest nmespace in the host process. Later, after awakening of the guest process, we configured guest IP address and guest end of veth pair. This protocol gave us the following output:

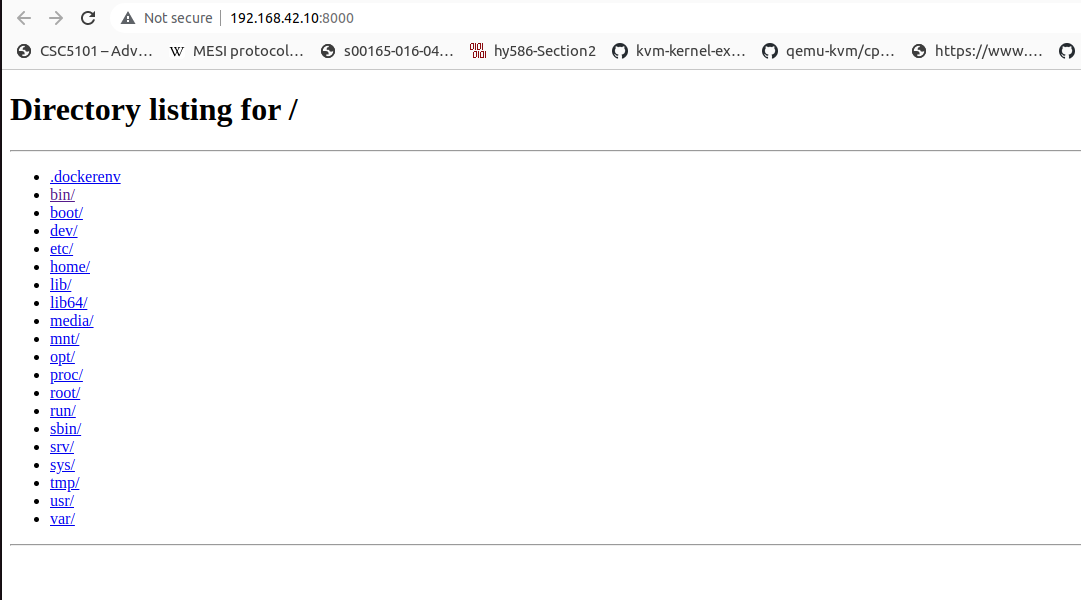


In addition to the loopback interface, now a new veth interface can be seen with a new IP address specified as a command line argument.

* 1. Network access to the container

The only thing that is left to be done is enabling a bridge that is created by make setup and used to route packages using ip addresses. Now, we are able to run a webserver and access it through the browser. The following images show the output of get requests we made and a webpage we received from a browser.





[Q16] How would you set up the network to have two containers communicate with each other?a By extension of how you connect two containers to each other, how would you handle networking isolation between groups of containers? 19 In other words, how would you have containers A and B connected to each other, and C and D connected to each other, but with A and B completely isolated from C and D? How could you have a setup where A is connected to B, B is connected to C, but A and C are still isolated at the network level?

It should be enough to have ends of both veth pairs belonging to different containers connected to the same bridge. Therefore, as bridge enables access between two connected sides, to ensure network isolation between two groups of containers we would just create two separate bridges (one connecting A and B, and another connecting C and D). The same thing could be applied for the second example, as well. We could create a bridge for A-B connection and then another one for B-C.

[Q17] The networking model of Kubernetes is to have containers of a pod share networking capabilities. How do you think it is implemented? There are two possibilities, but only one is actually used by Kubernetes, that involves only a namespace trick. You implemented the networking mode “bridge” of Docker. How would you implement the networking modes “none” (i.e., absolutely no networking) and “host” (i.e., absolutely no networking isolation between the container and the host)?

The networking model of pods in Kubernates is implemented by putting containers of the same pod into the same networking namespace, that would give them the same capabilities, the same ip address and port space. They would communicate with each other using localhost.

To implement a none mode, containers should be put in separate namespaces without configuring veth interfaces for them. To implement a host mode we would just avoid creating networking namespaces for containers.