CSC5004/ASR8 — Cloud Computing Architectures

Lab work

Simple Container Engine

Namespaces, control groups, overlay filesystem and networking

Mathieu Bacou
mathieu.bacou@telecom-sudparis.eu

2022 – 2023
Télécom SudParis
Institut Mines-Télécom & Institut Polytechnique de Paris

Part I. Introduction

1. Overview

Containers are a lightweight form of virtualization at the level of the operating system (OS). The principle is to expose a virtual, isolated and constrained, view of the OS to a process. To build the core of a container, you need two features from the Linux kernel:

namespaces isolation from the OS;
control groups (cgroups) resource limits and monitoring.

Additionally, modern container engines provide higher-level administration features, among them:

process management spawn a new process in a container;
overlay filesystem filesystem made from read-only, de-duplicated layers;
networking configuration of the isolated network.

In this lab, you will build a simple container engine as a command line program named `contain`. It will run a command in a container crafted by hand from namespaces and cgroups. It will also have its own isolated overlay filesystem, and networking facilities.

You first set up the environment in part II. Then, the lab is structured into adding namespaces in part III, and cgroups in part IV. At this point, isolation and limits are done, and you make an overlay filesystem for the container in part V. Networking in finally done in part VI.

2. Prerequisite

Containers are built from Linux kernel's features, so you will need a Linux installation. You also need to work on an `ext4` filesystem for the features needed to set up the filesystem views of containers. If not available, you can use an Ubuntu virtual machine from Google Cloud Platform, but programming directly in the VM may be less convenient.

A hard dependency for the container engine is libnl, to provide networking to the containers.[1] The libnl library is used to interact with the netlink API of

[1]libnl website: `https://www.infradead.org/~tgr/libnl/`.

the Linux kernel, that manages networking. Most Linux distributions feature a development package for it (as listed on its website), so you can install it via your package manager. For instance: `sudo apt install libnl-route-3-dev`.

Warning

Make sure to install a *development* package (i.e., a package with development headers) of libnl *with the "route" module.*

The lab guides you to write the container engine in C, so knowledge of the language for systems programming is assumed. Writing the container engine as a shell script is also possible,[2] or in any other language that provides access to system facilities (possibly via libraries).

Part II. Setup

In this part, you download a container image that will be used in your tests; and you get, and learn about the development environment for this lab.

3. Container image

Managing container images a-la Docker is not a goal of this lab, however *your container will have an overlay filesystem* to run in. Thus, it needs a base image. In this lab, you will use Docker's image of Debian Bullseye[3] (the stable version at the time of writing), that has been extracted as a static archive. It means that you will have access to most commands available to a base Debian installation (e.g., that includes Python) to test your progress, as will be suggested by the lab subject.

Information

You do not need it for this lab, but know there are two ways to extract a

[2]In fact, the previous version of the lab was written in shell, and can be found here: `https://www-inf.telecom-sudparis.eu/COURS/CSC5004/practicals/simple-container-engine_2021.pdf`.

[3]Debian image on Docker Hub: `https://hub.docker.com/_/debian/`.

Docker image:

1. export the content of a created container with `docker export CONT_NAME | xz > contimg.tar.xz`;
2. save the content of a local image with `docker image save IMG_NAME | xz > img.tar.xz`.

In both cases, the content is extracted as a Tar archive to `stdout`, so one has to redirect it, e.g. to `xz` to compress it.

Move to your work directory, and execute the command in listing 1.

Listing 1: Download the Debian container image.

```
wget https://www-inf.telecom-sudparis.eu/COURS/CSC5004/practicals/\
simple-container-engine/debian-stable+python+iproute2.tar.xz
```

Question 1

What is the role of a container image? How do you think you will use its content in this lab?

4. Development environment

You will write C code in prepared source files. To get these, execute the command in listing 2 in your work directory.

Listing 2: Download and extract the sources.

```
curl https://www-inf.telecom-sudparis.eu/COURS/CSC5004/practicals/\
simple-container-engine.tgz | tar --extract --gzip
```

You now have 6 files in your directory:

"contain.c" the main source file of the container engine's command `contain`, *you write code there*;

"libcontain.c" a helper library for the main program, *you also write code there*;

"libcontain.h" the header file of the helper library, you don't have anything to write there;

"Makefile" a Makefile (to use with `make TARGET`) to setup and reset the environment, build the container engine, and run test commands;

"setup.sh", "reset.sh" scripts to respectively set up the environment once, and reset the environment between runs.

To set up the environment, execute the command in listing 3.

Information

The `make` target `setup` runs the script "setup.sh" that:

1. extracts the container image;
2. enables the cgroup controllers `cpu` and `memory` in the root cgroup for the child cgroups;
3. creates a parent cgroup for all the containers that your engine spawns;
4. enables the cgroup controllers `cpu` and `memory` in the above parent cgroup for the child cgroups (i.e., the containers);
5. creates and sets up a network bridge (i.e., a virtual switch) to provide networking to the containers.

Listing 3: Set up the environment.

```
make setup
```

5. General instructions

As explained above, you work in "contain.c" and "libcontain.c", where you replace placeholder sections of code delimited by the marks shown in listing 4. The code sections indicate the related question(s) in the lab subject.

Listing 4: Illustration of code sections to replace.

```
/*** STUDENT CODE BELOW (qN, qM...) ***/

return 0;

/*** STUDENT CODE ABOVE (qN, qM...) ***/
```

In those sections, comments recapitulate what you have to code there. They also include the functions that you are expected to use to answer the question, as well as *references to their documentations*. You are expected to read them to understand how to use the indicated functions. The references you will find in these comments are:

`PAGE(SECTION)` read the manual page `PAGE` in the section `SECTION` with `man SECTION PAGE`;

`libcontain` not actually documentation, this reference means that the function is implemented in "libcontain.c" (and it may have an instructive documentation comment);

plain reference the name of a macro defined in "libcontain.h" or "contain.c" that you probably have to use here;

`cgroup doc` a section of the Linux kernel documentation about cgroups, especially the memory and CPU controllers at `https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v2.html`;

`libnl` read the documentation for libnl at `https://www-inf.telecom-sudparis.eu/COURS/CSC5004/libnl/`, in particular the “Routing Family Library (libnl-route)” section, and its API reference at `https://www-inf.telecom-sudparis.eu/COURS/CSC5004/libnl/api/group__rtnl.html`.[4]

To complete the code sections, you should begin by understanding what the enclosing function is doing. Read its preamble, i.e., the prepared variables above the code section you have to complete: you will probably have to use them. You are also advised to get familiar with the structure that represents a container: `struct container`; and its sub-structures.

Finally, when writing code, you should always check the return value of the function you use. Their documentation will tell you exactly how they behave, but in general terms they return `-1` on error. Then, you can use the following macros, defined in "libcontain.h", to handle the error:

`raise_err` print an error message, print an automatic description of the error code from the previous function,[5] and immediately return `-1`;

`raise_msg` same as `raise_err` but without printing the automatic description of the error code, you will prefer it when dealing with libnl functions.[6]

Part III. Execute the command in isolation: namespaces

The starting point of your container engine, is to execute a specified command inside a namespace. You use the system call `clone` to create a subprocess directly in namespaces. More precisely, you use the `clone3` variant that is more modern and offers the required features.

[4]You will encounter uses of libnl when managing the containers’ network.

[5]This is `errno`; you can learn more about `errno` at `errno(3)`.

[6]libnl does not use `errno`, so `raise_err` would always prints an unrelated error message.

6. Warmup: print and execute the containerized command

Coding task 1

Write the function `print_container_cmd`, that prints the command to run in the container, and its arguments.

To test your code, execute `make run-namespaces`. You should see the following output:

```
sudo "./contain" "mycont" "debian-stable+python+iproute2" "1024" "100" "192.168.42.10/24" bash
        -c 'echo User $(id -u), PID $$ \($(expr $(echo /proc/[0-9]* | wc -w) - 4)
                processes\), \
    hostname $(hostname), $(ip link | grep -c "^[0-9]*:") network interfaces'
running container "mycont" from image "debian-stable+python+iproute2"
container command: "bash" "-c" "echo User $(id -u), PID $$ \($(expr $(echo /proc/[0-9]*
        | wc -w) - 4) processes\), \
    hostname $(hostname), $(ip link | grep -c "^[0-9]*:") network interfaces"
```

Question 2

- Identify which namespaces are tested by this command.
- The command calls `bash`: where is the executed binary located?

Coding task 2

Execute the command to containerize.

`cont_args` is already prepared to be used by the `execvp` system call suggested in the code block to fill.

You should obtain additional output when running `make run-namespaces`:

```
User 1000, PID 58050 (202 process), hostname BacomputerW-MKIII, 4 network interfaces
```

You may have different numbers and hostname; after adding the namespaces to build a container (in the following sections), the output will change to reflect the new state of the container.

7. Execute the command in a subprocess

To start containerizing the command, start the subprocess of the container.

Coding task 3

Complete `clone_to_container` to call the `clone3` system call.

For this question, only the field `exit_signal` of `clone_args` is set.

Hint: the glibc does not have a wrapper for `clone3`, so you have to call it using `syscall`.

Question 3

What happens to the execution flow of your process when calling `clone`?

There are no namespace for now, because you have not set any flags for the `clone` system call (via the macro `CLONE_NAMESPACE_FLAGS`).

Coding task 4

In the code executed by the parent process, wait for the termination of the child container process, using the already declared variable `wstatus`.

You should have the output below when running `make run-namespaces`:

```
container started as process 10151


User 1000, PID 10151 (210 process), hostname BacomputerW-MKIII, 4 network interfaces

container process exited normally with code 0
container "mycont" terminated
```

8. User namespace

Start with the `user` namespace. Creating other namespaces requires privileges that are given when creating a `user` namespace.

Warning

Creating a user namespace without elevated privileges requires a sysctl configuration. Check with `sysctl kernel.unprivileged_userns_clone`: it must say that the value is `1`. If this is not the case, run the following command: `sudo sysctl kernel.unprivileged_userns_clone=1`. Note that this setting *does not survive a reboot.*

To create the container subprocess in a user namespace, it is enough to set the corresponding flag in the `clone_args` structure for the `clone` system call. However, *it does not change your user ID inside the container*, which means that you will execute the containerized command under a user ID that maps to nothing inside the container. So a second step is to map the root user (user ID 0) inside the container, to your user ID.

Coding task 5

Complete the macro `CLONE_NAMESPACE_FLAGS` to create a user namespace, and use it to set the field `flags` in the `clone_args` structure.

Then, complete `cgroup_map_root_user` in "libcontain.c" to map the root user in the container.

Finally, call `cgroup_map_root_user` in `finalize_host` in "contain.c".

Question 4

What change in the output do you expect when running the test command `make run-namespaces`? Be specific.

9. Hostname namespace

Now add the `uts` namespace, which is used in practice to isolate the hostname. Then set the container's hostname.

Coding task 6

Complete the macro `CLONE_NAMESPACE_FLAGS` to create a UTS namespace.

Then, complete `finalize_cont` to set the hostname of the container.

Question 5

- 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?

10. PID namespace

Now add the `pid` namespace. With this, `clone` will create a new process hierarchy starting with the PID 1. However, the processes on the host will remain visible to the container for now.

Coding task 7

Complete the macro `CLONE_NAMESPACE_FLAGS` to create a PID namespace.

Question 6

- 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.
- Which process (what program) has got PID 1 in your container?
- What do you expect to see when running the test command? Read carefully the containerized test command, and explain.

11. Network namespace

Now add the `net` namespace. Note however that you won't be doing any networking configuration until part VI.

Coding task 8

Complete the macro `CLONE_NAMESPACE_FLAGS` with a network namespace.

Question 7

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

12. Mount namespace

Finally, add the `mount` namespace. Despite not integrating the overlay filesystem for now (until part V), this is the most complex namespace to set up.

Coding task 9

Complete the macro `CLONE_NAMESPACE_FLAGS` with a mount namespace.

Question 8

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

The overlay filesystem will be set up later, but you will mount an isolated filesystem for the container nonetheless, using a bind mount. Creating the filesystem of the container is done by `contfs_make`, that is called by the `contain` function before creating the container subprocess.

Coding task 10

In "libcontain.c", complete `contfs_make` to create the directory that will be the mountpoint for the container filesystem.

Then, complete `contfs_mount` to actually mount the container filesystem. *You can ignore* `mount_options` *for now*, and make *a bind mount* of the container image to the root path defined in `cont_fs`.

Hint: a bind mount has no filesystem type.

From now, test your program using `make run-namespace-mount`.

Warning

The previous test command used by `make run-namespaces` will fail, because privileged rights are required to mount.

Question 9

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.

The main work is now done in `finalize_cont`.

Coding task 11

Complete `finalize_cont`:

- mount special filesystems (`proc`, `sysfs` and `tmpfs`) using the function `contfs_mount_pseudo_fs` from "libcontain.c";
- change the working directory to the container's filesystem root;
- use `chroot` to change the root of the running process to the current directory (which is now the root of its dedicated filesystem).

Information

The `chroot` system call changes the apparent root directory of the process. In some way, it is a limited form of containerization, and as you can see, is still used to implement complete containerization solutions.

Question 10

- 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?

Now that isolation is set up, the next step is to put resource limits on the container using cgroups.

Part IV. Put resource limits on the command: control groups

The interface to control cgroups is directory and file operations:

- creating directories with `mkdir` to create cgroups;[7]

[7]Under the root of the cgroup hierarchy that was created by `make setup` beforehand.

- opening files with `open` (and derivative);
- reading and writing to opened files under cgroup directories to read and set limits.

13. Control group creation

First, you need to create the control group. This is done by the `cgroup_make` function that is called by `contain` before creating the container subprocess.

Coding task 12

In "libcontain.c", complete `cgroup_make` to create the cgroup; you should set the access rights of the created directory to something sensible.

Note that you also have to open the created directory and store the file descriptor (FD) in `cgroup->fd`. Here, you must pass 4 special flags to `open`, see `open(2)` and `O_PATH`.

Then, you must instruct the call to `clone` to actually create the container subprocess in the cgroup.

Coding task 13

Complete `clone_to_container` so that the call to `clone` will put the subprocess in the cgroup.

Test your code with `make run-cgroups`.

Question 11

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

At this point, the container subprocess lives under a cgroup. The setup for this lab enabled the cgroup controllers `cpu` and `memory`, but you have not set any limit yet.

14. Imposing limits to the cgroup

The limits are set in `cgroup_make` in "libcontain.c", that calls the functions `cgroup_limit_memory` and `cgroup_limit_cpu` to set respectively, the memory limit and the CPU limit.

Coding task 14

Complete `cgroup_limit_memory` in "libcontain.c".

The process to apply the limit is explained in the comments. In particular, the functions you should use to manipulate the string data are listed. This is also where you use the FD to the cgroup's directory that you previously stored in `cgroup->fd` in `cgroup_make`.

Coding task 15

Complete `cgroup_limit_cpu` in "libcontain.c".

The instructions are similar to setting the memory limit in the previous coding task. However, the process is slightly more complex because you must compute the CPU limit from the CPU percentage you are given in arguments, and the current scheduler period that you read from the control file.

Information

There is no way from inside the container, to discover the resource limits set by the cgroups.

Question 12

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.

With this, the process is now properly containerized:[8] it is isolated from the host system, and resource limits are applied.

Let's move forward with the implementation of common container engine features: usage of an overlay filesystem, and basic networking configuration.

[8]Except that a few namespaces and resource limits are missing, but those are trivial to add by following the same procedures.

Part V. Reusable container images: overlay filesystem

In this part, you will replace bind-mounting the container image by mounting it in a proper overlay filesystem. In other words, the container will now run in a filesystem built as layers.

Question 13

- What is the container image in the container's filesystem layers?
- How are filesystem layers used in container images?
- How are filesystem layers 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?

Creating an overlay filesystem is done by `mount`ing a filesystem of type `overlay`. Mounting it actually involves four different directories:

1. the `lowerdir`, which contains the immutable base of the overlay filesystem;
2. the `upperdir`, which contains files that differ from the base layer;
3. the `workdir`, which serves as a working directory for the overlay filesystem driver, it has no correlation with the concepts of containerization seen in the lecture,
4. the mountpoint, i.e., where to actually mount the overlay filesystem for use by the container.

Question 14

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

The mountpoint folder will constitute the container's isolated root.

Coding task 16

In `contfs_mount` in "libcontain.c", prepare `mount_options` with the mount options for an overlay filesystem. Then, mount the container image as an overlay filesystem.

Hint: the mount origin is already specified in `mount_options`, so it does not matter in the call to `mount`.

Information

It is possible to pile overlay filesystems up, where the mount point (the run layer) of a filesystem is used as the `lowerdir` of another one. With this method, you can achieve image layers just like Docker.

Test with `make run-overlayfs`, to observe that a file created by the containerized command does not appear in the base container image directory.

The only feature left to implement in your simple container engine, is networking.

Part VI. Communicating with containers: networking

Providing networking to the container is done in two steps:

1. the network configuration of the container;
2. the network access to the container.

To achieve them, you will use virtual ethernet devices (veth) and a bridge.

Information

This is the networking mode “bridge” in the Docker world.

Warning

The lab does not cover network access to the container from outside the host. To do so requires using NAT with iptables to route packets between the container and the physical network device, which is a general network configuration task, and is not linked to writing a container engine.

Information

veth devices are *pairs of virtual Ethernet devices* provided by the Linux kernel: packets sent to one end are received on the other end, and vice-versa.

Question 15

How do you use a veth pair to provide networking to a container?

The rough outline of this section is to have the container engine create the veth pair, putting one side in the container; and to have the container wait for its veth interface, and configure it (set its IP address).

In other words, some synchronization is needed between the container engine and the container: creating the veth pair requires that the container's network namespace be already created (to put the container end of the veth pair in it), but the container side must wait for the container engine to do that before setting the IP address from within the namespace.

15. Synchronization between container engine and container

Setting aside the networking for this section, you must add code to have the container subprocess wait for a signal from the container engine. The latter will send the signal when it is done adding the veth interface to the container. The skeleton of this machinery is already in place:

1. the container engine parent process calls `set_cont_wait` to set a signal handler before creating the container process;[9]
2. the container subprocess calls `cont_wait` to wait for the signal;
3. the parent process calls `cont_start` to signal the subprocess to start.

[9]This is required to avoid race conditions where the signal could be delivered to the container process before it set up the signal handler.

Coding task 17

Complete `set_cont_wait` to declare a signal handler (e.g., on `SIGUSR1`) for the container process. You must also write your own signal handler.

Then, complete `cont_wait` to make the container subprocess wait on the execution of the signal handler (use a condition variable).

Finally, complete `cont_start` to make the container engine signal the container subprocess (e.g., with `SIGUSR1` as suggested above).

Test your code with `make run-networking`. Your container process now waits for the parent container engine to finish its setup (write your signal handler to make clear what happens).

Take also note of the output of `make run-networking`: it will change as you add the container end of the veth pair in the next section.

16. Network configuration: virtual Ethernet

With the synchronization mechanism between the host and the container in place, you can add networking to the container.

Coding task 18

Complete `finalize_host` to make the network from the host by calling `contnet_make_host`, and implement the latter in "libcontain.c".

Then, complete `finalize_cont` to set up the network from the container by calling `contnet_make_cont`, and implement the latter in "libcontain.c".

These pieces of code use libnl. Comments will help you find the functions you need in its documentation. Take the time to understand what variables are prepared for you in each function, to use in your code.

Run again the test command `make run-networking`. Confirm that the container now has a network interface configured with the given IP address.

Information

veth pairs are automatically destroyed when the namespace that contains one end is destroyed, or when any end of the pair is destroyed.

17. Network access to the container

Finally, let's set up network access to the container. This is very simple: `make setup` created a bridge on the host, and you plug the host end of the veth pair into the bridge. This will provide access to the container from the host, as well as mutual access between containers on the same bridge.

Information

Bridges can be seen as *virtual switches* provided by the Linux kernel: they route packets to network interfaces based on IP addresses.

Coding task 19

Complete `contnet_connect_host_bridge` in "libcontain.c" to put the host end of the veth pair in the bridge.

Test your code by running the command `make run-networking-webserver`. It starts a Python webserver reachable at `http://192.168.42.10:8000` (that simple serves the files in the container image): open this link in your web browser to confirm that the containerized webserver has network.

Warning

To exit the container process, simply send it `SIGINT` by hitting Ctrl-C: signals are forwards from your container engine to the container process.

Question 16

In this question, answer by describing how you would use veth pairs, bridges, and any other element you would find useful.

- 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? 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?[b]

[a]The IP address of your container is hardcoded in the Makefile. To test your answer, run the commands manually and give different IP addresses to the containers.
[b]Interestingly, this "proxy" setup cannot be easily achieved with Docker.

Question 17

- 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)?[a]

[a]The networking mode "overlay" of Docker requires the set up of a technology of networking overlay, which ends up creating a virtual interface that you simply add to a container's network namespace. This is out of the scope of this lab, but also not really interesting, and requires two host machines to be demonstrated.

Congratulations! You wrote a simple container engine with a few lines of C code ! Remark how the longest and hardest parts were to set up the filesystem and the networking, but running a process in isolation and under resource limits only revolves around the `clone` system call.

Part VII. Conclusion

In this lab, you manipulated all the low-level features that, once put together one by one, constitute a very functional container engine: isolation, limits, overlay filesystem and networking. This gives a glimpse of the work achieved by industry-ready container engines to bring container technology to the public.

To go further, your engine lacks container management: you could implement listing running containers, stopping a container and destroying its resources. You could also implement the three missing namespaces. Run the command `ls -l /proc/self/ns` on the host, in your container and in a Docker container to list them: `ipc`, `cgroups` and `time`.