CSC5004/ASR8 — Cloud Computing Architectures

Practical

contain.sh

Namespaces, control groups, overlay filesystem and networking

Mathieu Bacou
mathieu.bacou@telecom-sudparis.eu

2021 – 2022
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.

On top of this, you can have capabilities to represent applications as containers, to implement the paradigm of micro-services, but those orchestrating features are out of the scope of this lesson.

In this practical, you will build a Bash script called `contain.sh` to run a command in a container crafted by hand that presents the previously listed features, and with minimal configuration (memory and CPU limits, networking...). You will first setup the environment in part II, and then execute the container's command in part III, piling namespaces one by one. Afterwards, part IV will guide you in applying control groups to the command. At this point, isolation and limits are done, and it is time to provide an overlay filesystem to the container in part V, and then networking in part VI.

2. Prerequisite

Containers are built from Linux kernel's features, so you will need a Linux installation. In particular, the subject is written for control groups v1. You can check it by issuing the command `ls /sys/fs/cgroup`: it should display at least directories named `memory` and `cpu`. You also need to work on an ext4 filesystem for 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 might be less convenient.

The practical proposes to write the container engine in Bash, so knowledge of shell scripting is assumed. Other languages can be used, but the practical is written to use files, OS and shell facilities. Languages may also provide (possibly via libraries) more convenient interfaces that you can use at your own discretion.

Part II.
Setup

In this part, you fetch a container image that will be used in your tests, and you set cgroups up for future usage.

3. Container image

Managing container images a-la Docker is not a goal of this practical, however *your container will have an overlay filesystem* to run in. Thus, it needs a base image. In this practical, you will use Docker's image of Python[1], that has been extracted as a static tarball. This way, you can write any example you want with a Python program, or use its embedded HTTP server implementation.

Information

You do not need it for this practical, but know there are two ways to extract a Docker image:

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

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

[1]Python image on Docker Hub: `https://hub.docker.com/_/python/`.

Jump into your work directory and execute the command in listing 1.

Listing 1: Download extracted Python container image

```
curl https://stark2.int-evry.fr/mbacou/csc5004/practicals/\
simple-container-engine/python3.tar.xz | tar -xz -C python3
```

Question 1

What is the role of a container image? How do you think you will use it in this practical when creating a container?

4. cgroups setup

We assume that the sysfs of the control groups is already mounted by your system. What you need to do here is *to create a cgroup that will act as the parent* of all your containers' cgroups, and to give your user the rights to manage cgroups underneath. We will limit the containers on memory and CPU usage, so we need to create the parent cgroups under the `memory` and the `cpu` controllers.

Information

Control groups v1 are managed as hierarchies: under each resource controller, there is a root cgroup, that contains every process of the system. All other cgroups are children of this root cgroup, and you can create children to those cgroups, etc. In addition, children cgroups inherit limits of their parent cgroup.

Execute the commands in listing 2: the first one creates the parent cgroups under the `memory` and the `cpu` controllers, and the second one sets your current user as their owner.

Listing 2: Setup control groups

```
sudo mkdir /sys/fs/cgroup/{memory,cpu}/containers
sudo chown -R $(id -u):$(id -g) /sys/fs/cgroup/{memory,cpu}/containers
```

Part III. Execute the command in isolation: namespaces

The starting point is to execute a specified command inside a namespace. You will use the command `unshare` to do so.

Information

`unshare` is a thin wrapper around the syscall of the same name. You can explore the use of the command with `man 1 unshare`, and complete this with `man 2 unshare` for the documentation of the syscall. The subject will warn you about pitfalls, however you will have to read the manual to answer a few questions related to `unshare` and its behavior.

5. Base running script: `contain.sh`

Actually, for some namespaces, additional steps will be required to configure the isolated resources from inside the namespace. Thus, our script `contain.sh` will run another script named `continit.sh` inside the namespaces; the latter will make the necessary configuration before executing the command given by the user. This architecture is summarized in fig. 1.

Let's start by writing the base of `contain.sh`. It is shown in listing 3.

Listing 3: Base version of `contain.sh`.

```bash
#! /usr/bin/bash

# Flags for the unshare command, completed one namespace at a time
UNSHARE_FLAGS=""

# Note that we fork to background
unshare $UNSHARE_FLAGS ./continit.sh "$@" &
```

As you can see, the central command is `unshare`, that isolates a command (thus ”unsharing” the current environment). Its first argument beside flags, is our script to configure resources from the inside, `continit.sh`, to which the argument list, i.e. the user command to run in the container, is passed as-is. For now, the list of flags for `unshare` is empty so no namespace is created.

Listing 4 gives the initial code of `continit.sh`.

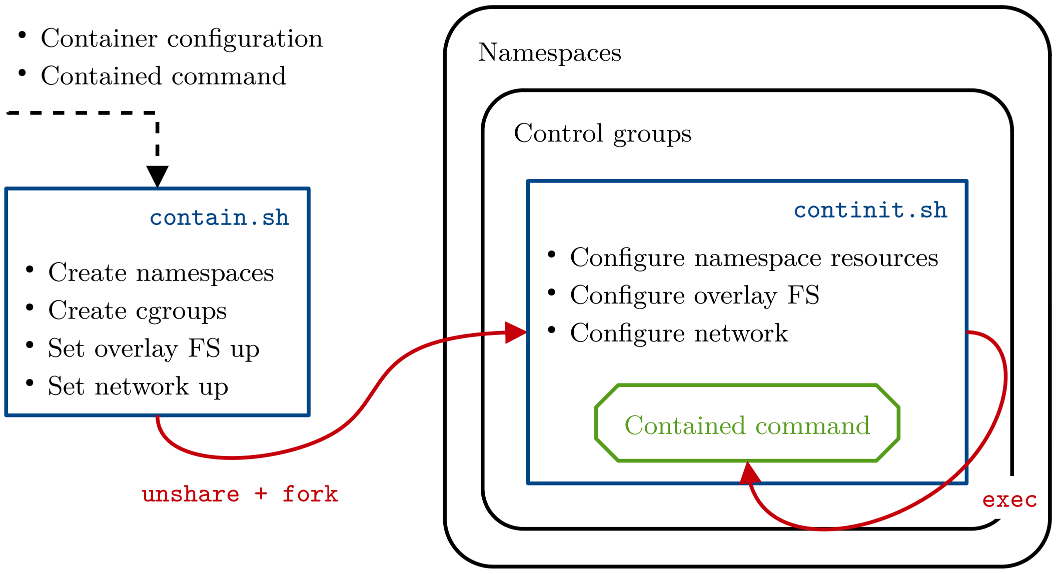

Figure 1: Architecture of the simple container engine, `contain.sh`.

Listing 4: Base version of `continit.sh`.

```bash
#! /usr/bin/bash

# Print the command to run (with a delimiter "!" to show Bash words)
cmd=$1
shift
echo -n "cmd: $cmd"
for word in "$@"; do
    echo -n " ! $word"
done
echo

# Replace the current process with the command to run
exec "$cmd" "$@"
```

Question 2

Why do we use `exec` to run the user's command? Think of what processes are spawned in the container, and of their PIDs.

Once both scripts are written (and a `chmod u+x contain.sh continit.sh` is done), test them. You can use the command given in listing 5, which is a good example to show you the effect of each namespace you will add afterwards.

Listing 5: Test command.

```
./contain.sh \
    bash -c 'echo User $(id -u), PID $$ \($(expr $(ps | wc -l) - 4) processes\), \
    hostname $(hostname), $(ip link | grep -c "^[0-9]*:") network interfaces'
```

Question 3

- Can you identify what namespaces are tested by this command?
- The command calls `bash`: where is the executed binary located?
- Why is it important to write the command to `bash` inside single quotes?

6. User namespace

We begin with the `user` namespace, because 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 knob to be set. This is most probably the case, but you can check it 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*.

Add the flags `--user --map-root-user` to `UNSHARE_FLAGS` to create the user namespace. Then, test your scripts again to confirm that the flag had an effect

Question 4

- What does the `--user` flag do exactly?
- What does the `--map-root-user` do? What does it mean in practical terms?
- What do you expect to see when running the test command?

7. Hostname namespace

Now we add the `uts` namespace, which is used in practice to isolate the hostname.

Add the `--uts` flag to `UNSHARE_FLAGS`. Then, modify `continit.sh` to set the hostname inside the namespace (use the command `hostname`) with the value given as its first argument, and update `contain.sh` to pass the hostname to it (the hostname should also become an argument of `contain.sh`).

Test your scripts again; don't forget to pass the hostname as an argument to `contain.sh`.

Question 5

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

8. PID namespace

Now we add the `pid` namespace.

Add the `--pid --fork` flags to `UNSHARE_FLAGS`. The second flag tells `unshare` to fork to run the given command. This is necessary to make the command the "init" process (PID 1) of the namespace.

Warning

If you forget the `--fork` flag, you will get the following error: "bash: fork: Cannot allocate memory".

Test your scripts again.

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

9. Network namespace

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

Add the `--net` flag to `UNSHARE_FLAGS`.

Test your scripts again.

Question 7

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

10. Mount namespace

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

Start by adding the `--mount` to `UNSHARE_FLAGS`, and run the test command.

Question 8

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

As said just before, we ignore the overlay filesystem for now. To emulate this, in `contain.sh`, we will create a folder for the container and *bind-mount* the folder of the image to this new folder (note that we need superuser privileges to use `mount`). The image (or rather, its folder) should now become an argument to `contain.sh`. The path to this folder is passed to `continit.sh` in the `unshare` call. A summary of these modifications is given in listing 6.

Listing 6: Container filesystem setup (without overlay filesystem) in `contain.sh`.

```
hostname=$1
image=$2
shift 2
# ...
# The name of the container's FS directory is built from the given hostname and
# image name
contfs=$image-$hostname
# Use --parents to ignore error when directory exists
mkdir --parents "$contfs"
# Simulate mounting the overlay FS by bind-mounting the image directory
sudo mount --rbind "$image" "$contfs"

unshare $UNSHARE_FLAGS ./continit.sh "$hostname" "$contfs" "$@" &
```

Then, the main work is done in `continit.sh`. In this script, we change the working directory to the container's filesystem and mount special filesystems (`proc`, `sysfs` and `tmpfs`), before using `chroot` to change the root of the running process to the current directory. This is shown in listing 7.

Information

`chroot` (a wrapper command and a syscall) changes the apparent root directory of the process. In some way, it is a limited form of containerization, and is still used to implement complete containerization solutions.

Listing 7: Container filesystem setup in `continit.sh`.

```bash
hostname=$1
fs=$2
shift 2
# ...
# Change current directory to the container's filesystem
cd "$fs"
# Mount special filesystems
mount -t proc proc proc
mount -t sysfs none sys
mount -t tmpfs none tmp

# Execute chroot and make it execute the container's command
exec chroot . "$cmd" "$@"
```

Test your scripts again (don't forget to add the image name "python3" to the command line) by running the command `ls /home` inside your container.

Question 9

- Why is the container's filesystem mounted from `contain.sh`?
- Why are the special filesystems `proc`, `sysfs` and `tmpfs` mounted from `continit.sh`?
- What do you expect to see when running the new test command `ls /home`?

To finish, clean up by unmounting the container's filesystem with `sudo umount`.

11. Summary: namespace isolation

As a reference, listings 8 and 9 give examples of the scripts you should have at this point.

Listing 8: `contain.sh`: namespace isolation.

```bash
#! /usr/bin/bash

hostname=$1
image=$2
shift 2

```

```
# Flags for the unshare command, completed one namespace at a time
UNSHARE_FLAGS="--user --map-root-user\
    --uts\
    --pid --fork\
    --net\
    --mount"

# The name of the container's FS directory is built from the given hostname and
# image name
contfs=$image-$hostname
# Use --parents to ignore error when directory exists
mkdir --parents "$contfs"
# Simulate mounting the overlay FS by bind-mounting the image directory
sudo mount --rbind "$image" "$contfs"

# Note that we fork to background
unshare $UNSHARE_FLAGS ./continit.sh "$hostname" "$contfs" "$@" &
```

Listing 9: `continit.sh`: namespace isolation.

```
#! /usr/bin/bash

hostname=$1
fs=$2
shift 2

cmd=$1
shift
echo -n "cmd: $cmd"
for word in "$@"; do
    echo -n " ! $word"
done
echo

# HOSTNAME
hostname "$hostname"

# MOUNT
# Change current directory to the container's filesystem
cd "$fs"
# Mount special filesystems
mount -t proc proc proc
mount -t sysfs none sys
mount -t tmpfs none tmp

# Replace the current process with the command to run
# Execute chroot and make it execute the container's command
exec chroot . "$cmd" "$@"
```

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

Control groups (cgroups) are controlled from the shell via the special filesystem `cgroup`, that is usually mounted by your distribution under `/sys/fs/cgroups`. The interface consists in:

- creating directories with `mkdir`[2];
- displaying file content with `cat`;
- writing to files with `echo value > controlfile`.

All the work in this part is done in `contain.sh`, before calling `unshare`.

This script will now take as additional parameters the memory and CPU limits, expressed respectively in MB and in percentage of CPU.

12. Memory control group

First, we must create the container's cgroup, using `mkdir` under the parent cgroup created in section 4. Then, we write the memory limit to the cgroup's file `memory.limit_in_bytes`.

This will create and set up the cgroup. All that is left to do, is to jump into the cgroup by writing a PID in the cgroup's file `cgroup.procs`. We will write the PID of the current process, i.e. of `contain.sh`, so that the call to `unshare`, and its fork to `continit.sh` that then `executes` `chroot` before `executing` the user command, happen in the cgroup from the beginning.

We propose the addition of code shown in listing 10.

Listing 10: Memory control group setup in `contain.sh`.

```
# Parent cgroup of our containers' cgroups
PARENT_CGROUP=containers
# ...
memory_MB=$1
shift 1
# Convert memory limit from MB to B
memory=$(expr "$memory_MB" '*' 1024 '*' 1024)
# ...
```

[2]As you did in section 4 when you created parent cgroups under the `memory` and `cpu` controllers.

```
memory_cgroup="/sys/fs/cgroup/memory/$PARENT_CGROUP/$hostname"
mkdir "$memory_cgroup"
echo $memory > "$memory_cgroup/memory.limit_in_bytes"

echo $$ > "$memory_cgroup/cgroup.procs"
```

13. CPU control group

The same steps are to be followed to set the CPU cgroup up.

To apply a limit, the file to write is `cpu.cfs_quota_us`: it sets the quota of a scheduler timeslice (in µs) allocated to the processes in the cgroup.

Information

The files `cpu.cfs_period_us` and `cpu.cfs_quota_us` control the scheduler of the Linux kernel, called Completely Fair Scheduler (CFS). The former sets the period of scheduling, in µs, while the latter sets the amount of time, in µs, allocated to the processes during a timeslice.

Question 10

What are the effects of setting:

- a shorter or longer scheduling period?
- a lower or higher quota allocation?

Also, what are the downsides?

The interface of our script `contain.sh` asks for a percentage of CPU, so this value must be converted to a quota in µs with a simple computation shown in listing 11 that reads the scheduler allocation period from `cpu.cfs_period_us`.

Listing 11: Conversion of CPU allocation in `contain.sh`.

```
# ...
memory_MB=$1
cpu_perc=$2
shift 2
# ...
cpu_cgroup="/sys/fs/cgroup/cpu/$PARENT_CGROUP/$hostname"
mkdir "$cpu_cgroup"
echo $(expr "$cpu_perc" '*' $(cat "$cpu_cgroup/cpu.cfs_period_us") / 100) > \
    "$cpu_cgroup"/cpu.cfs_quota_us

echo $$ > "$cpu_cgroup/cgroup.procs"
```

14. Summary: resource limits

Test your script. Don't forget that you now need to pass the memory and CPU limits to `contain.sh`. You can try with the command shown in listing 12 to check that the containerized process indeed runs in the cgroups.

Listing 12: Test command for cgroups.

```
./contain.sh mycont python3 128 50 bash -c 'cat /proc/self/cgroup | grep mycont'
```

Clean up by removing the container's cgroups with the `rmdir` command, as well as unmounting the mounted container filesystem like before.

As a reference, listing 13 gives an example of the script `contain.sh` you should have at this point.

Listing 13: `contain.sh`: resource limits.

```
#! /usr/bin/bash

# Parent cgroup of our containers' cgroups
PARENT_CGROUP=containers

hostname=$1
image=$2
shift 2
memory_MB=$1
cpu_perc=$2
shift 2
# Convert memory limit from MB to B
memory=$(expr "$memory_MB" '*' 1024 '*' 1024)

# Flags for the unshare command, completed one namespace at a time
UNSHARE_FLAGS="--user --map-root-user\
    --uts\
    --pid --fork\
    --net\
    --mount"

# The name of the container's FS directory is built from the given hostname and
# image name
contfs=$image-$hostname
# Use --parents to ignore error when directory exists
mkdir --parents "$contfs"
# Simulate mounting the overlay FS by bind-mounting the image directory
sudo mount --rbind "$image" "$contfs"

memory_cgroup="/sys/fs/cgroup/memory/$PARENT_CGROUP/$hostname"
mkdir "$memory_cgroup"
echo $memory > "$memory_cgroup/memory.limit_in_bytes"

cpu_cgroup="/sys/fs/cgroup/cpu/$PARENT_CGROUP/$hostname"
mkdir "$cpu_cgroup"
echo $(expr $cpu_perc '*' $(cat "$cpu_cgroup/cpu.cfs_period_us") / 100) > \
    "$cpu_cgroup"/cpu.cfs_quota_us

echo $$ > "$memory_cgroup/cgroup.procs"
echo $$ > "$cpu_cgroup/cgroup.procs"
```

```

# Note that we fork to background
unshare $UNSHARE_FLAGS ./continit.sh "$hostname" "$contfs" "$@" &
```

With this, the process is now properly containerized:[3] 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.

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 11

- 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 calling the `mount` command to mount 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,

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

4. the mountpoint, i.e., where to actually mount the overlay filesystem for use by the container.

Question 12

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 constitue the container's isolated root. Note that the path to the container's filesystem that is passed to `continit.sh` must be changed to point to the overlay filesystem mount point.

An example of commands to set the overlay filesystem up, replacing the previous commands in `contain.sh` to bind-mount the container's image, is given in listing 14. Nothing needs to be done in `continit.sh`.

Listing 14: Overlay filesystem setup in `contain.sh`.

```
# ...
# The name of the container's FS directory is built from the given hostname and
# image name
contfs="$image-$hostname"
# Use --parents to ignore error when directories exist
mkdir --parents "$contfs"/{.diff,.workdir,run}
# Mount the overlay FS under the container's directory, in "run"
sudo mount -t overlay overlay \
    -o lowerdir="$image",upperdir="$contfs/.diff",workdir="$contfs/.workdir" \
    "$contfs/run"
# ...
unshare $UNSHARE_FLAGS ./continit.sh "$hostname" "$contfs/run" "$@" &
```

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 your scripts. You can check the proper usage of the overlay filesystem with the commands in listing 15.

Listing 15: Test command for overlay filesystem.

```
# Create a file in the container, and check its presence
./contain.sh mycont python3 128 50 bash -c 'touch testfile; ls'
# Check the original image to see that the file is not present
ls python3
```

Clean up by removing cgroups as before, and unmounting the overlay filesystem with `sudo umount python3-mycont/run`.

Part VI. Communicating with containers: networking

The final part of our minimal container engine is to give network capabilities to the container. There are two halves:

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

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

Information

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

Warning

The practical does not cover network access to the container from outside the host. To achieve this, one would create NAT routing rules 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 creating a container engine.

15. Network configuration: virtual Ethernet

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 13

How do you use a veth pair to provide networking to a container? The rough idea is to use it to “traverse” the network namespace of a container.

veth pairs require basic IP configuration; `contain.sh` is to be modified to expect a new argument: the container’s IP address. veth management and IP configuration is done by the command `ip` (note that we need superuser privileges to use this command).

Notice that a part of the network configuration must be done from inside the container (setting the IP address and the route), i.e. by `continit.sh`. However, the veth device will be inserted into the network namespace *after calling* `unshare` and starting the execution of `continit.sh`, because the namespace must exist; and the PID of its first process (as seen from outside the PID namespace) is required to insert the device into it. Thus, some synchronization is required to have the script wait until the interface is available.

To this end, we will use a named pipe with `mkfifo`, and redirect a file descriptor (FD) to it. When `unshare` is called and `continit.sh` is forked, the latter retains the open FDs. Thus, it can block on reading the FD, and receive the name of the container’s veth device as well as its IP address.

To summarize:

- in `contain.sh`:
  1. get a new argument: the container’s IP address,
  2. redirect a file descriptor to a pipe to be used for cross-namespace communication,
  3. launch `continit.sh`,
  4. create the veth pair and insert one end inside the container,
  5. send the container its name and its IP address via the pipe;
- in `continit.sh`:
  1. block reading from the pipe,
  2. configure the veth end inside the container.

An example of the scripts updates is given in listings 16 and 17.

Listing 16: veth configuration in `contain.sh`.

```
ipaddr=$1
shift 1
# ...
# Create a named pipe for cross-namespace communication
mkfifo "$hostname.pipe"
```

```
# Redirect file descriptor 3 to the pipe
exec 3<>"$hostname.pipe"
# ...
unshare $UNSHARE_FLAGS ./continit.sh "$hostname" "$contfs/run" "$@" &
continit_pid=$!

# vethOutXXX is the end on the host, vethInXXX is the end in the container
sudo ip link add "vethOut$hostname" type veth peer name "vethIn$hostname"
sudo ip link set "vethOut$hostname" up
# Insert the container's end inside the net namespace
sudo ip link set "vethIn$hostname" netns $continit_pid

echo "vethIn$hostname" "$ipaddr" >&3
rm "$hostname.pipe"
```

Listing 17: veth configuration in `continit.sh`.

```
# ...
# Block waiting for veth pair end and IP address
read vethitf ipaddr <&3

# Notice that sudo is not needed here: we are mapped to the root user
ip link set "$vethitf" up
ip addr add $ipaddr dev "$vethitf"
# Set the default route through the veth device
ip route add default dev "$vethitf"
#...
```

Test your scripts. Don't forget that you now need to pass an IP address (complete with a netmask length, e.g. `192.168.42.10/24`) to `contain.sh`. Confirm that the container now has a network interface configured with the given IP address. An example of a test command is given in listing 18.

Listing 18: Test command for veth configuration.

```
./contain.sh mycont python3 128 50 192.168.42.10/24 ip addr
```

Information

veth pairs are automatically destroyed when the namespace that contains one end is destroyed.

16. Network access to the container

Finally, let's set up network access to the container. This is very simple: you will create a permanent bridge on the host.

Information

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

On container creation, the end of the veth pair that is left on the host is added to the bridge. This will provide access to the container from the host, as well as mutual access between containers on the same bridge.

Listing 19 gives the commands to set up the bridge from your shell.

Warning

You only have to run the commands in listing 19 once, in your terminal, to set up the bridge. After the practical, you can remove the bridge with `ip link delete contbr0 type bridge`.

Listing 19: Bridge configuration on the host.

```
# The bridge is named contbr0
sudo ip link add name contbr0 type bridge
sudo ip link set contbr0 up
# Give an IP address to the bridge
sudo ip addr add 192.168.42.1/24 dev contbr0
```

To add the end of the veth pair to the bridge, update `contain.sh` with the command given in listing 20.

Listing 20: veth and bridge configuration in `contain.sh`.

```
# ...
sudo ip link set "vethOut$hostname" master contbr0
```

Once the bridge is set up on the host, test your scripts. For example, you can run two containers trying to ping each other. A more interesting example is to run a web server (the one provided by Python in its standard library) and then try to send a request to it from the host. This is shown in listing 21.

Listing 21: Test command for network access.

```
./contain.sh myserver python3 128 50 192.168.42.10/24 python -m http.server
# Once returned to the shell
wget 192.168.42.10:8000
# Or use your web browser
```

Warning

We didn't see how to kill a contained process. When testing networking with a `ping` command or with the Python server, you can stop a container by sending `SIGKILL` to its init process. To get the latter's PID, you can use `pgrep`.

Question 14

- By extension of how you connected two containers to each other trying to ping 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 air gapped 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?[a]
- 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)?[b]

[a]Interestingly, this setup, which can be seen as a proxy pattern, cannot be easily achieved with Docker.

[b]The networking mode "overlay" of Docker requires the set up of an 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 practical, but also not really interesting, and requires two host machines to be demonstrated.

17. Summary: network configuration

As a reference, listings 22 and 23 give examples of the scripts you should have at this point. This is their final version.

Listing 22: `contain.sh`: network configuration.

```bash
#! /usr/bin/bash

# Parent cgroup of our containers' cgroups
PARENT_CGROUP=containers

hostname=$1
image=$2
```

```
shift 2
memory_MB=$1
cpu_perc=$2
shift 2
ipaddr=$1
shift 1

# Convert memory limit from MB to B
memory=$(expr "$memory_MB" '*' 1024 '*' 1024)

# Flags for the unshare command, completed one namespace at a time
UNSHARE_FLAGS="--user --map-root-user\
    --uts\
    --pid --fork\
    --net\
    --mount"

# Create a named pipe for cross-namespace communication
mkfifo "$hostname.pipe"
# Redirect file descriptor 3 to the pipe
exec 3<>"$hostname.pipe"

# The name of the container's FS directory is built from the given hostname and
# image name
contfs=$image-$hostname
# Use --parents to ignore error when directories exist
mkdir --parents "$contfs"/{.diff,.workdir,run}
# Mount the overlay FS under the container's directory, in "run"
sudo mount -t overlay overlay \
    -o lowerdir="$image",upperdir="$contfs/.diff",workdir="$contfs/.workdir" \
    "$contfs/run"

memory_cgroup="/sys/fs/cgroup/memory/$PARENT_CGROUP/$hostname"
mkdir "$memory_cgroup"
echo $memory > "$memory_cgroup/memory.limit_in_bytes"

cpu_cgroup="/sys/fs/cgroup/cpu/$PARENT_CGROUP/$hostname"
mkdir "$cpu_cgroup"
echo $(expr "$cpu_perc" '*' $(cat "$cpu_cgroup/cpu.cfs_period_us") / 100) > \
    "$cpu_cgroup/cpu.cfs_quota_us"

echo $$ > "$memory_cgroup/cgroup.procs"
echo $$ > "$cpu_cgroup/cgroup.procs"

# Note that we fork to background
unshare $UNSHARE_FLAGS ./continit.sh "$hostname" "$contfs/run" "$@" &
continit_pid=$!

# vethOutXXX is the end on the host, vethInXXX is the end in the container
sudo ip link add "vethOut$hostname" type veth peer name "vethIn$hostname"
sudo ip link set "vethOut$hostname" up
# Insert the container's end inside the net namespace
sudo ip link set "vethIn$hostname" netns $continit_pid
sudo ip link set "vethOut$hostname" master contbr0

echo "vethIn$hostname" "$ipaddr" >&3
rm "$hostname.pipe"
```

Listing 23: `continit.sh`: network configuration.

```bash
#! /usr/bin/bash

hostname=$1
fs=$2
shift 2

cmd="$1"
shift
echo -n "cmd: $cmd"
for word in "$@"; do
        echo -n " ! $word"
done
echo

# HOSTNAME
hostname "$hostname"

# MOUNT
# Change current directory to the container's filesystem
cd "$fs"
# Mount special filesystems
mount -t proc proc proc
mount -t sysfs none sys
mount -t tmpfs none tmp

# Block waiting for veth pair end and IP address
read vethitf ipaddr <&3

# Notice that sudo is not needed here: we are mapped to the root user
ip link set $vethitf up
ip addr add $ipaddr dev $vethitf
# Set the default route through the veth device
ip route add default dev $vethitf

# Execute chroot and make it execute the container's command
exec chroot . "$cmd" "$@"
```

Congratulations! You wrote a simple container engine with two Bash scripts.

Part VII. Conclusion

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

To go further, you can discover the interfaces to those features that are available to programming languages. For example, in C, you use syscalls to create namespaces and mount the overlay filesystem, and it would certainly be easier to build a nicer interface to your container engine. Beside the fact that a few namespaces and cgroups are missing, your engine also lacks proper container management: can you list running containers? How can you stop a container? How would you handle the destruction of its resources?