SYS1

Filesystem

20xx

Version 1

Jules AUBERT <jules1.aubert@epita.fr>

Contents

1 Introduction **2**
1.1 Filesystem, container and more . . . 2

2 Filesystem **3**
2.1 Hierachy . . . 3
2.1.1 Mount . . . 3
2.2 Filesystem . . . 4
2.3 Chroot . . . 4
2.3.1 initrd and initramfs . . . 5
2.4 Container . . . 5
2.5 Namespace . . . 5
2.6 Cgroup . . . 6
2.7 Capabilities . . . 8

1 Introduction

1.1 Filesystem, container and more

The purpose of this course is to help you have a better knowledge about Linux filesystems, containers and other resources. It is a melting pot of several knowledge easy to learn and test. You will dive into some Linux knowledge to understand how containers tools work (LXC, podman, docker, ...).

To a full comprehension of the last chapter, you should read first the **network programming** document.

For each new command or function or whatever, you should always read its manpage.

2 Filesystem

2.1 Hierachy

Is Linux just a set of directories? Maybe not. Try:

```
$ man 7 hier
...
$
```

The `hier(7)` man page provides an overview of the standard filesystem hierarchy on Unix-like systems. It describes the purpose and typical contents of directories such as `/`, `/bin`, `/etc`, `/usr`, `/var`, and more. Understanding this layout is essential for navigating the system, writing scripts, configuring services, and developing system-level programs. It also helps clarify which directories are read-only, which are variable or temporary, and which are used during boot or multi-user runtime.

Let's dive into `/dev/input`. After executing a command, move your mouse.

```
$ sudo cat /dev/input/mouse0
...
CTRL^C
$
```

```
$ sudo xxd /dev/input/mouse0
...
CTRL^C
$
```

And what about those files in `/dev/`?

```
$ sudo dd if=/dev/sda of=/tmp/data bs=10k status=progress
...
$ file /tmp/data
...
```

The entry point to your disk is a simple file.

`/dev/` contains all the device files for your Linux system. We will have a closer look in the next semester. ;)

2.1.1 Mount

If you want to know what is **mounted** on your filesystem, use the `mount(1)` command.

```
$ mount
(...)
proc on /proc type proc (rw,nosuid,nodev,noexec,relatime)
(...)
sys on /sys type sysfs (rw,nosuid,nodev,noexec,relatime)
(...)
dev on /dev type devtmpfs (rw,nosuid,relatime,size=4045724k,nr_inodes=1011431,mode=755,inode64)
(...)
tmpfs on /tmp type tmpfs (rw,nosuid,nodev,nr_inodes=1048576,inode64)
(...)
$
```

`/proc/`, `/sys/` and other directories are special mounted points in RAM.

2.2 Filesystem

You are going to download a filesystem and work with it. It existes several filesystem, based on Linux distributions. We will use **Ubuntu**. If you want a tiny filesystem you can use **Alpine**, but it is very, very small with few binaries.

```
$ wget
https://dl-cdn.alpinelinux.org/alpine/v3.16/releases/x86_64/alpine-minirootfs-3.16.2-x86_64.tar.gz
-O rootfs.tar.gz
$ mkdir rootfs
$ sudo tar xf rootfs.tar.gz -C rootfs
$
```

2.3 Chroot

`chroot(1)` is a program to **ch**ange the **root** of the userspace program (the shell, for example).

We will change our **root filesystem** to the one we have just downloaded. Only for one shell, of course.

chroot is just a wrapper around `pivot-root(2)`.

```
$ sudo chroot rootfs /bin/sh
(rootfs) # exit
$
```

The `/bin/sh` used as a parameter is the one inside the filesystem we have just downloaded. If it does not exist, search for another shell or another path inside. Let's execute some command inside our "*container*".

```
(rootfs) # ls
/bin/sh: ls: not found
(rootfs) # echo *
bin dev etc home lib media mnt opt proc root run sbin srv sys tmp usr var
(rootfs) # /bin/ls
bin    dev    etc    home   lib    media  mnt    opt    proc   root   run    sbin   srv    sys    tmp
    usr    var
(rootfs) # /bin/ps
PID   USER     TIME  COMMAND
(rootfs) #
```

`ps(1)` is failing. Why?

`ps(1)` is a parsing tool going to some files from `/proc/` and `/sys/`. Both are **virtual filesystems**. See them as kernel's API exposing information.

Let's try some malware commands... :)

```
# In the parent filesystem (the real filesystem)
$ vlc&
# Get the processus ID printed immedialtly
$
# In the 'container'
(rootfs) # /bin/kill VLC_PROCESSUS_ID
```

You just killed a parent process from a container... Wait what?

Do not worry, there's a simple and historical explanation.

2.3.1 initrd and initramfs

How does a filesystem get mounted at first on boot?

The bootloader executes the kernel and loads a tiny Linux distro named initrd (or initramfs for newer Linux). It loads the drivers then pivots into the your distro filesystem. No security, only pivot_rooting the root filesystem.

2.4 Container

What is a container exactly?

A container is an isolated user-space instance built on top of the host Linux kernel. It relies on namespaces to provide isolation (PID, mount, network, UTS, IPC, user) and cgroups to limit or monitor resource usage. At its core, a container can be seen as a process launched in a new namespace with a custom root filesystem. Unlike virtual machines, containers do not emulate hardware or run separate kernelsthey simply isolate and control processes more efficiently within the same OS kernel.

How can we isolate our container? We will use the `namespaces(7)`.

2.5 Namespace

A namespace is a kernel feature that provides isolation of global system resources between sets of processes. Each namespace type isolates a specific aspect of the system. For example, the PID namespace gives processes their own process ID space, the mount namespace allows different views of the filesystem, and the network namespace provides separate network interfaces and routing tables. Namespaces enable containers by ensuring that processes inside them have a limited and private view of the system, while still running on the same kernel as the host.

Namespace	**Effect / Isolation scope**
`mnt` (Mount)	Isolates the set of mount points and the filesystem hierarchy. Each namespace can have its own view of the mounted filesystems.
`pid` (Process ID)	Provides each namespace with its own PID number space. Processes inside see a different PID hierarchy than outside.
`net` (Network)	Gives processes their own network stack: interfaces, routing tables, firewall rules, etc.
`uts` (UNIX Timesharing System)	Allows each namespace to have its own hostname and domain name.
`ipc` (Interprocess Communication)	Isolates System V IPC and POSIX message queues.
`user`	Isolates user and group ID mappings. Allows unprivileged users to have root privileges inside a container.
`cgroup`	Isolates the view and management of control groups (resource limitations, accounting, etc.).

Table 1: Linux namespace types and their purposes

We will use `unshare(1)` to isolate our processus inside the container. For your knowledge, behind these technics are present some syscalls, as `clone(2)`, `setns(2)`, `unshare(2)` (obviously). By the way, if you did not know, `clone(2)` is the syscall hidden behind `fork(2)`.

```
$ sudo mount -t proc /proc rootfs/proc
$ sudo unshare -f -p -n --mount-proc=rootfs/proc chroot rootfs /bin/sh
```

The parameters used are:

- -f: fork the executed program, chroot here
- -p: unshare the PID namespace
- -n: unshare the network namespace ; if you do not add it, you can see your network interfaces with `ip(1)`, if you add it, you will not see your network interfaces

And now, inside the container:

```
(rootfs) # /bin/ps
...
(rootfs) # /bin/kill VLC_PROCESSUS_ID # it fails
```

`/bin/sh` has now the pid '1' and we can not kill parent's process. We are isolated!

2.6 Cgroup

Control groups (cgroups) are a Linux kernel feature that allow the system to limit, account for, and isolate the resource usage of process groups. Cgroups can control resources such as CPU time, memory, disk I/O, and network bandwidth. They are organized in hierarchies, with each hierarchy attached to one or more controllers (e.g., `cpu`, `memory`, ...). Processes placed in the same cgroup share resource limits and accounting, and new limits can be applied dynamically. Cgroups are a core component of containerization, used to enforce resource constraints and provide better security and performance isolation.

Let's give it a try! You can replace `APPING` with anything else.

```
$ sudo mkdir /sys/fs/cgroup/APPING
$ cd /sys/fs/cgroup/APPING
$ ls
...
$
```

It populated itself.

Let's limit a processus max memory to **100MB**.

```
$ echo $((100 * 1024 * 1024)) | sudo tee memory.max
104857600
```

Here are two Python scripts doing the same thing. One is in Python2 and the other one Python3.

```
#!/usr/bin/env python2

f = open("/dev/urandom", "r")
data = ""
i = 0

while True:
    data += str(f.read(10000000))
    i += i
    print '%dmb' % (i * 10,)
```

```python
#!/usr/bin/env python3

f = open("/dev/urandom", "rb")
data = ""
i = 0

while True:
    data += str(f.read(10000000))
    i += i
    print(f"{i * 10}mb")
```

They will eat 10Mb by 10Mb data from the special file `/dev/urandom`. It is a special file outputting random data without limit. `man 4 urandom`.

Let's open a new shell and try to limits its memory.

Shell number 1 is the one executing the script. Shell number 2 is the one reducing the RAM to shell number 1. My example may differs from your execution.

```
# Shell 1
$ echo $$ # Get the shell PID
6969
$
# Shell 2
$ echo 6969 | sudo tee cgroup.procs
or
$ sudo cgexec -g memory:APPING ./memory.py
# Shell 1
$ ./memory.py
10mb
20mb
30mb
40mb
50mb
60mb
70mb
80mb
90mb
100mb
110mb
120mb
CTRL^C
$
```

It did not work... Actually, it did. ;)

Replay the script, you will see a display slowdown when getting to 100MB. Why? Because the shell limited processus is **swapping** its memory into a **swap file**. Swap is space on the disk used as RAM. It permits to virtually have more RAM but is slower than real RAM.

Let's limit the maximum memory to be swapped to 10MB.

```
# Shell 2
$ echo $((10 * 1024 * 1024)) | sudo tee memory.swap.max
10485760
```

Rexecute the script the same way again, from shell number 1. You will get a segfault, because now the processus lacks of memory space.

2.7 Capabilities

Linux capabilities are a fine-grained access control mechanism that breaks down the privileges traditionally associated with the root user into distinct units. Instead of granting full root access, the kernel allows processes to be assigned only the specific privileges they need, improving security through the principle of least privilege. For example, `CAP_NET_ADMIN` allows managing network interfaces, while `CAP_SYS_CHROOT` permits calling `chroot()`. Capabilities can be added or dropped per process, and are especially useful in containerized environments to reduce the attack surface while retaining necessary functionality.

You can learn more at `man 7 capabilities`

Remember in the network programming course when I say that the ports from 0 to 1023 are reserved ports and only `root` can use them? Let's change the rules for a moment.

To be sure, try to execute the server with a low port number, like 80.

```
$ ./server 127.0.0.1 80
server: Cannot bind the service.
$ sudo setcap cap_net_bind_service+ep ./server
$ ./server 127.0.0.1 80
CTRL^C
$
```

It works!