Virtualisation légère – TP n° 3

Linux Internals partie 1

Pierre-Olivier *nemunaire* Mercier

Mercredi 19 octobre 2022

Abstract

Ce premier TP consacré aux Linux Internals va nous permettre d'appréhender les notions de pseudos systèmes de fichiers, de *cgroups* ainsi que de *capabilities*.

Les exercices de ce cours sont à rendre au plus tard le mardi 8 novembre 2022 à 23 h 42. Consultez la dernière partie pour plus d'informations sur les éléments à rendre.

Sommaire

1 Statistiques et contrôle des processus

Maintenant que nous avons pu voir en détail comment utiliser Docker, nous allons tâcher d'appréhender les techniques qu'il met en œuvre. De nombreuses fonctionnalités du noyau Linux sont en effet sollicitées : certaines sont là depuis longtemps, quelques unes sont standardisées, d'autres sont plus récentes.

Dans un premier temps, nous allons aborder la manière dont le noyau Linux permet de contrôler les processus : que ce soit en collectant des informations sur eux ou en imposant certaines restrictions.

1.1 Prérequis

Pour pouvoir suivre les exemples et faire les exercices qui suivent, vous aurez besoin d'un noyau Linux récent (une version 5.x sera très bien). Il doit de plus être compilé avec les options suivantes (lorsqu'elles sont disponibles pour votre version) :

```
General setup  --->
    [*] Control Group support  --->
      [*]   Freezer cgroup subsystem
      [*]   PIDs cgroup subsystem
      [*]   Device controller for cgroups
      [*]   Cpuset support
      [*]   Simple CPU accounting cgroup subsystem
      [*]   Memory Resource Controller for Control Groups
      [*]   Group CPU scheduler  --->
        [*]   Group scheduling for SCHED_OTHER
        [*]   Group scheduling for SCHED_RR/FIFO
      <*>   Block IO controller
[*] Networking support  --->
    Networking options  --->
      [*] Network priority cgroup
      [*] Network classid cgroup
```

Si vous utilisez un noyau standard fourni par votre distribution, les options requises seront a priori déjà sélectionnées et vous n'aurez donc pas à compiler votre propre noyau. Néanmoins, nous allons nous interfacer avec le noyau, il est donc nécessaire d'avoir les en-têtes de votre noyau.

Sous Debian, vous pouvez les installer via le paquet au nom semblable à `linux-headers`. Le paquet porte le même nom sous Arch Linux et ses dérivés.

1.1.1 Vérification via `menuconfig`

L'arbre ci-dessus correspond aux options qui seront *built-in* (signalées par une `*`) ou installées en tant que module (signalées par un `M`). En effet, chaque noyau Linux peut être entièrement personnalisé en fonction des options et des pilotes que l'on voudra utiliser.

Pour parcourir l'arbre des options du noyau, il est nécessaire d'avoir les sources de celui-ci. Les dernières versions stables et encore maintenues sont disponibles sur la page d'accueil de https://kernel.org.

Dans les sources, on affiche la liste des options avec la commande :

```
make menuconfig
```

1.1.2 Vérification via `/boot/config-xxx`

Les distributions basées sur Debian ont pour habitude de placer le fichier de configuration ayant servi à compiler le noyau et ses modules dans le dossier `/boot`, aux côtés de l'image du noyau `vmlinuz-xxx`, de l'éventuel système de fichiers initial (`initramfs-xxx`) et des symboles de débogage `System.map-xxx`.

Ce fichier répertorie toutes les options qui ont été activées. Par rapport à l'arbre présenté ci-dessus, vous devriez trouver :

```
CONFIG_CGROUPS=y
CONFIG_CGROUP_FREEZER=y
CONFIG_CGROUP_PIDS=y
CONFIG_CGROUP_DEVICE=y
CONFIG_CPUSETS=y
CONFIG_CGROUP_CPUACCT=y
CONFIG_CGROUP_MEMCG=y
CONFIG_CGROUP_SCHED=y
CONFIG_BLK_CGROUP=y

CONFIG_NET=y
CONFIG_CGROUP_NET_PRIO=y
CONFIG_CGROUP_NET_CLASSID=y
```

1.1.3 Vérification via `/proc/config.gz`

Dans la plupart des autres distributions, la configuration est accessible à travers le fichier `/proc/config.gz`. Comme vous ne pouvez pas écrire dans `/proc` pour décompresser le fichier, utilisez les outils `zcat`, `zgrep`, ...

Vous devez retrouver les mêmes options que celles de la section précédente.

1.2 Pseudos systèmes de fichiers

Les systèmes Unix définissent le système de fichiers comme étant un arbre unique partant d'une racine[1] et où l'on peut placer au sein de son arborescence des points de montage. Ainsi, l'utilisateur définit généralement deux points de montage :

```
/dev/sda1 on / type ext4 (rw,relatime,data=ordered)
/dev/sda3 on /home type ext4 (rw,relatime,data=ordered)
```

Dans ce schéma, la racine correspond à la première partition du premier disque, et les fichiers des utilisateurs sont sur la troisième partition du premier disque.

1.2.1 Présentation des pseudos systèmes de fichiers

D'autres points de montage sont utilisés par le système : `/dev`, `/proc`, `/tmp`, ... Ces points de montage vont, la plupart du temps, être montés par le programme d'initialisation en utilisant des systèmes de fichiers virtuels, mis à disposition par le noyau.

[1] Consultez https://en.wikipedia.org/wiki/Filesystem_Hierarchy_Standard pour plus de détails sur l'arboresence.

Ces systèmes sont virtuels, car ils ne correspondent à aucune partition d'aucun disque : l'arborescence est créée de toute pièce par le noyau pour trier les informations mises à disposition, mais il n'est pas toujours possible d'y apporter des modifications.

Linux emploie de nombreux systèmes de fichiers virtuels :

— `/proc` : contient, principalement, la liste des processus (`top` et ses dérivés se contentent de lire les fichiers de ce point de montage) ;
— `/proc/sys` : contient la configuration du noyau ;
— `/sys` : contient des informations à propos du matériel (utilisées notamment par `udev` pour peupler `/dev`) et des périphériques (taille des tampons, clignotement des DELs, ...) ;
— `/sys/firmware/efi/efivars` : pour accéder et modifier les variables de l'UEFI ;
— ...

Tous ces systèmes de fichiers sont généralement exclusivement stockés en RAM. Pour rendre une modification persistante, il est nécessaire de modifier un fichier de configuration qui sera chargé par le programme d'initialisation. Par exemple, pour modifier les paramètres du noyau, on passe par le fichier `/etc/sysctl.conf` et le programme `sysctl`.

1.2.2 Consultation et modification

La consultation d'un élément se fait généralement à l'aide d'un simple `cat` :

```
42sh$ cat /sys/power/state
freeze mem
```

La modification d'un élément se fait avec `echo`, comme ceci :

```
42sh# echo mem > /sys/power/state
```

Vous devriez constater l'effet de cette commande sans plus attendre !

1.2.2.1 `procinfo`

Explorons le pseudo système de fichiers `/proc` pour écrire un script qui va afficher des informations sur un processus donné :

```
42sh$ ./procinfo $$
PID: 4242
Path: /bin/bash
Command line: bash
Working directory: /home/nemunaire/virli/
Root: /
State: S (sleeping)
Threads: 1

CGroups
=======
12:pids:/
11:net_prio:/
10:perf_event:/
9:net_cls:/
8:freezer:/
7:devices:/
6:memory:/
```

```
5:blkio:/
4:cpuacct:/
3:cpu:/
2:cpuset:/
1:name=openrc:/

Namespaces
==========
cgroup:[4026531835]
ipc:[4026531839]
mnt:[4026531840]
net:[4026531969]
pid:[4026531836]
user:[4026531837]
uts:[4026531838]
```

1.2.2.2 `batinfo.sh`, `cpuinfo.sh` Explorons le pseudo système de fichiers `/sys` pour écrire un script qui va, en fonction de ce que vous avez de disponible :

- afficher des statistiques sur votre batterie ;
- afficher des statistiques sur la fréquence du CPU.

`batinfo.sh` Voici un exemple d'utilisation :

```
42sh$ ./batinfo.sh
BAT0 Discharging
====
Capacity: 83% (Normal)
Voltage: 11.972000 V (minimal: 11.400000 V)
Energy: 18.290000/21.830000 Wh
Power: 7.937000 W
Remaining time: 2.304 h

BAT1 Unknown
====
Capacity: 83% (Normal)
Voltage: 11.972000 V (minimal: 11.400000 V)
Energy: 18.290000/21.830000 Wh
Power: 0.0 W
Remaining time: N/A
```

Pour les détails sur l'organisation de ce dossier, regardez :
https://www.kernel.org/doc/Documentation/power/power_supply_class.txt.

`cpuinfo.sh` Voici un exemple d'utilisation :

```
42sh$ ./cpuinfo.sh
cpu0
```

```
====
Current frequency: 2100384 Hz
Current governor: powersave
Allowed frequencies: between 500000 - 2100000 Hz
Thermal throttle count: 0

cpu1
====
Current frequency: 2099871 Hz
Current governor: powersave
Allowed frequencies: between 500000 - 2100000 Hz
Thermal throttle count: 0
```

N'hésitez pas à rajouter toute sorte d'informations intéressantes !

1.2.2.3 `rev_kdb_leds.sh`, `suspend_schedule.sh`

Maintenant que vous savez lire des informations dans `/sys`, tentons d'aller modifier le comportement de notre système. Au choix, réalisez l'un des scripts suivants, en fonction du matériel dont vous disposez :

- inverser l'état des diodes de votre clavier ;
- mettre en veille votre machine, en ayant programmé une heure de réveil.

`rev_kdb_leds.sh` Si vous avez :

- numlock On,
- capslock Off,
- scrolllock Off ;

Après avoir exécuté le script, vous devriez avoir :

- numlock Off,
- capslock On,
- scrolllock On.

Voici un exemple d'utilisation :

```
42sh# ./rev_kdb_leds.sh input20
```

`input20` correspond à l'identifiant de votre clavier, sous `/sys/class/input/`.

`suspend_schedule.sh` Votre script prendra en argument l'heure à laquelle votre machine doit être réveillée, avant de la mettre effectivement en veille.

Bien sûr, vous ne devez utiliser que des écritures (et des lectures) dans le système de fichiers `/sys`. Il n'est pas question de faire appel à un autre programme (vous pourriez cependant avoir besoin de `date(1)` pour faire les calculs horaires).

Voici un exemple d'utilisation :

```
42sh# ./suspend_schedule.sh 15:42
```

Vous aurez besoin de définir une alarme au niveau de votre RTC, via le fichier : `/sys/class/rtc/rtcX/wakealarm`

> **!** Attention au fuseau horaire utilisé par votre RTC, si votre système principal est Windows, elle utilisera sans doute le fuseau horaire courant. Sinon, ce sera UTC.

Un article très complet sur le sujet est disponible ici : https://www.linux.com/tutorials/wake-linux-rtc-alarm-clock/

1.3 Les cgroups

Les *cgroup*s (pour *Control Group*s) permettent de collecter des statistiques sur des **groupes de processus** (voire même, des threads !) et de leur attribuer des propriétés. Il est par exemple possible de leur imposer des limites d'utilisation de ressources ou d'altérer leur comportement : quantité de RAM, temps CPU, bande passante, …

Apparue dès Linux 2.6.24 (en 2008 !), les *cgroup*s sont répartis en différents sous-systèmes (*subsystem*), chacun étant responsable d'un type de ressources spécifique :

- `blkio` (`io` dans la v2) : limites et statistiques de bande passante sur les disques ;
- `cpu` : cycles CPU minimums garantis ;
- `cpuacct` (inclus dans `cpu` dans la v2) : statistiques du temps CPU utilisé ;
- `cpuset` : associe des tâches à un/des CPU particuliers (par exemple pour dédier un cœur du CPU à un programme, qui ne pourra alors utiliser que ce CPU et pas les autres) ;
- `devices` : règles de contrôle de création (`mknod`) et d'accès aux périphériques ;
- `freezer` : pour suspendre et reprendre l'exécution d'un groupe de tâches ;
- `hugetlb` : statistiques et limitation de l'usage de la fonctionnalité `HugeTLB` (permettant d'obtenir des pages mémoires plus grandes que 4 kB) ;
- `memory` : statistiques et limitation d'usage de la mémoire vive et de la *swap* ;
- `net_cls` (v1 seulement) : applique un `classid` à tous les paquets émis par les tâches du *cgroup*, pour filtrage par le pare-feu en sortie ;
- `net_prio` (v1 seulement) : surcharge la valeur de l'option de priorité `SO_PRIORITY`, ordonnant la file d'attente des paquets sortants ;
- `pids` : statistiques et limitation du nombre de processus ;
- …

Nous allons commencer par faire quelques tests avec le *cgroup freezer*, qui permet d'interrompre l'exécution d'un groupe de processus, puis de la reprendre lorsqu'on le décide.

1.3.1 Montage du freezer

En fonction de la configuration de votre système, vous allez vous trouver dans l'une de ces trois situations :

- Votre dossier `/sys/fs/cgroup` contient à la fois des fichiers `cgroup.*` et éventuellement des dossiers : vous avez une distribution moderne qui utilise la nouvelle version des `cgroups`.

— Votre dossier `/sys/fs/cgroup` contient d'autres dossiers au nom des sous-systèmes que l'on a listés ci-dessus : il s'agit des cgroups v1.
— Votre dossier `/sys/fs/cgroup` est vide ou inexistant, vous pouvez choisir d'utiliser la version de votre choix :
Pour utiliser la v1 :

```
mkdir /sys/fs/cgroup/freezer/
mount -t cgroup -o freezer none /sys/fs/cgroup/freezer/
```

Pour utiliser la v2 :

```
mkdir /sys/fs/cgroup/
mount -t cgroup2 none /sys/fs/cgroup/
```

Avant d'aller plus loin, notez que les exemples seront donnés pour les deux versions des cgroups à chaque fois.

Quelles sont les différences entre les deux versions des cgroups ? La principale différence entre les deux est la fusion des différents sous-systèmes au sein d'une même arborescence. Dans la première version, chaque sous-système disposait de sa propre arborescence et il fallait créer les groupes et associer les tâches pour chaque sous-système. Avec la seconde version, une seule création est nécessaire, quelque soit le nombre de sous-systèmes que l'on souhaite utiliser.

1.3.2 Création d'un nouveau groupe

Les *cgroup*s sont organisés autour d'une arborescence de groupe, où chaque groupe est représenté par un dossier. Il peut bien évidemment y avoir des sous-groupes, en créant des dossiers dans les dossiers existants, etc.

La première étape dans l'utilisation d'un *cgroup* est donc de créer un groupe.

Pour ce faire, il suffit de créer un nouveau dossier dans un groupe existant, par exemple la racine.

On commence par se rendre à la racine :

```
cd /sys/fs/cgroup/freezer/  # v1
cd /sys/fs/cgroup/          # v2
```

Puis on crée notre groupe :

```
mkdir virli
ls virli/
```

Nous avons maintenant un nouveau groupe de processus `virli` dans le *cgroup Freezer*. Comme il s'agit d'une hiérarchie, le groupe `virli` hérite des propriétés de son (ses) père(s).

1.3.3 Sélection de contrôleur (v2 seulement)

Du fait de l'unification de tous les sous-systèmes, si vous utilisez la seconde version, vous allez devoir activer le ou les contrôleurs dont vous avez besoin (tandis que dans la première version, on se rendait dans l'arborescence du sous-système que l'on voulait).

Pour activer le contrôleur *memory*, nous utilisons la commande suivante à la racine :

```
echo "+memory" > cgroup.subtree_control
```

! Si vous obtenez l'erreur `No such file or directory`, c'est sans doute que vous avez les cgroups v1 activé quelque part. Vous devriez plutôt utiliser la première version, le fait qu'elle soit active empêche l'utilisation de la v2 en parallèle.

On peut voir les contrôleurs actifs en consultant le fichier `virli/cgroup.controllers`.

Le contrôleur *freezer* est généralement activé par défaut, il n'y a pas besoin de l'activer.

1.3.4 Rattachement de processus

Pour le moment, ce nouveau groupe ne contient aucun processus, comme le montre le fichier `cgroup.procs` de notre groupe. Ce fichier contient la liste des processus rattachés à notre *cgroup*.

Ouvrons un nouveau terminal (c'est lui que l'on va geler), et récupérons son PID : `echo $$`.

Pour ajouter une tâche à ce groupe, cela se passe de cette manière :

```
echo $PID > /sys/fs/cgroup/{,freezer/}virli/cgroup.procs
```

Il faut ici remplacer `$PID` par le PID du shell que l'on a relevé juste avant.

? **Ne devrait-on pas utiliser >> pour ajouter le processus au fichier ?** Malgré l'utilisation de la redirection > (et non >>), il s'agit bel et bien d'un ajout au fichier et non d'un écrasement. Il faut garder en tête que le système de fichier est entièrement simulé et que certains comportements sont adaptés.

En validant cette commande, nous avons déplacé le processus dans ce groupe, il n'est alors plus dans aucun autre groupe (pour ce *cgroup*, il ne bouge pas dans les autres *cgroup*s de la v1).

? **Où sont placés les nouveaux processus ?** Les nouveaux processus héritent des groupes de leur père.
Si vous lancez un `top` dans votre nouveau terminal, son PID sera présent dans le fichier `cgroup.procs`.

1.3.5 Consultation de l'état

En affichant le contenu du dossier `virli`, nous pouvions constater que celui-ci contenait déjà un certain nombre de fichiers. Certains d'entre eux sont en lecture seule et permettent de lire des statistiques instantanées sur le groupe ; tandis que d'autres sont des propriétés que nous pouvons modifier.

Nous pouvons consulter l'état de gel du groupe en affichant le contenu du fichier :

```
42sh$ cat /sys/fs/cgroup/freezer/virli/freezer.state      # v1
42sh$ cat /sys/fs/cgroup/virli/cgroup.freeze              # v2
```

Pour plus d'informations sur les différents fichiers présents dans ce *cgroup*, consultez la documentation associée :
https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v1/freezer-subsystem.html

1.3.6 Changement d'état

Faisons exécuter à notre interpréteur une commande pour voir effectivement l'exécution s'arrêter. Si vous manquez d'inspiration, utilisez :

```
for i in $(seq 9999); do echo -n $i; sleep .1; echo -n " - "; sleep .1; done
```

Maintenant, nous allons donner l'ordre au noyau de ne plus allouer de temps de calcul à notre shell et ses fils :

```
echo FROZEN > /sys/fs/cgroup/freezer/virli/freezer.state  # v1
echo 1 > /sys/fs/cgroup/virli/cgroup.freeze               # v2
```

À cet instant, vous devriez voir votre compteur s'arrêter. Pour reprendre l'exécution :

```
echo THAWED > /sys/fs/cgroup/freezer/virli/freezer.state  # v1
echo 0 > /sys/fs/cgroup/virli/cgroup.freeze               # v2
```

Script de monitoring

À nous maintenant de concevoir un script de collecte de statistiques issues des *cgroups*, similaire à `telegraf`.

Dans un premier temps, commençons par afficher dans la console, la quantité de mémoire utilisée par le groupe monitoré.

Vous pouvez utiliser un programme comme `memhog`[a] pour remplir rapidement votre mémoire.

[a] Ce programme fait partie du paquet `numactl`, mais vous trouverez une version modifiée plus adaptée à nos tests sur https://nemunai.re/post/slow-memhog.

```
42sh# ./monitor group_name memhog 500
~~~ 13595 ~~~ Current memory usage: 75194368
~~~ 13595 ~~~ Current memory usage: 150290432
~~~ 13595 ~~~ Current memory usage: 223690752
~~~ 13595 ~~~ Current memory usage: 296828928
~~~ 13595 ~~~ Current memory usage: 368001024
~~~ 13595 ~~~ Current memory usage: 438517760
~~~ 13595 ~~~ Current memory usage: 480329728
~~~ 13595 ~~~ Current memory usage: 155648
```

Le modèle de sortie standard de votre script `monitor` n'a pas d'importance, il doit cependant être possible d'y trouver des statistiques intéressantes, dont la quantité de mémoire utilisée. Ici nous affichons au début le PID du processus, ce qui peut simplifier le débogage du script.

Il s'utilise de la manière suivante :

```
./monitor group_name prog [args [...]]
```

Où :

— `group_name` correspond au nom du/des *cgroup*(s) à créer/rejoindre.
— `prog [args [...]]` est la commande que l'on souhaite monitorer, à exécuter dans le *cgroup*.

Si vous n'avez pas le *cgroup memory*, il est possible qu'il ne soit pas activé par défaut par votre système. Si vous êtes dans ce cas, essayez d'ajouter `cgroup_enable=memory` à la ligne de commande de votre noyau.

Gardez ce script dans un coin, nous allons le compléter dans les sections suivantes.

1.3.7 Fixer des limites

Au-delà de la simple consultation, les *cgroup*s peuvent servir à limiter la quantité de ressources mises à disposition à un groupe de processus.

Pour définir une limite, nous allons écrire la valeur dans le fichier correspondant à une valeur limite, comme par exemple `memory.max_usage_in_bytes` (v1) ou `memory.max` (v2), qui limite le nombre d'octets que notre groupe de processus va pouvoir allouer au maximum :

```
# cgroup v1
42sh$ cat /sys/fs/cgroup/memory/virli/memory.max_usage_in_bytes
0
# 0 = Aucune limite
42sh$ echo 4M > /sys/fs/cgroup/memory/virli/memory.max_usage_in_bytes
# Maintenant, la limite est à 4MB, vérifions...
42sh$ cat /sys/fs/cgroup/memory/virli/memory.max_usage_in_bytes
4194304
```

```
# cgroup v2
42sh$ cat /sys/fs/cgroup/virli/memory.max
max
# max = Aucune limite
42sh$ echo 4M > /sys/fs/cgroup/virli/memory.max
# Maintenant, la limite est à 4MB, vérifions...
42sh$ cat /sys/fs/cgroup/virli/memory.max
4194304
```

Chaque *cgroup*s définit de nombreux indicateurs et possède de nombreux limiteurs, n'hésitez pas à consulter la documentation associée à chaque *cgroup*.

Mettez à jour votre script de monitoring pour prendre en compte les limites que vous avez définies :

```
42sh# mkdir /sys/fs/cgroup...
42sh# echo 512M > /sys/fs/cgroup.../memory.max_usage_in_bytes
```

```
42sh# ./monitor group_name memhog 500
~~~ 13595 ~~~ Current memory usage: 75194368/550502400 (13%)
~~~ 13595 ~~~ Current memory usage: 150290432/550502400 (27%)
~~~ 13595 ~~~ Current memory usage: 223690752/550502400 (40%)
~~~ 13595 ~~~ Current memory usage: 296828928/550502400 (53%)
~~~ 13595 ~~~ Current memory usage: 368001024/550502400 (66%)
~~~ 13595 ~~~ Current memory usage: 438517760/550502400 (79%)
~~~ 13595 ~~~ Current memory usage: 480329728/550502400 (87%)
~~~ 13595 ~~~ Current memory usage: 155648/550502400 (0%)
```

Pour aller plus loin

Pour tout connaître en détail, la série d'articles de Neil Brown sur les Control groups[2] est excellente ! Plus cet article sur la version 2[3].

Exercice (obligatoire pour les SRS – optionnel pour les GISTRE)

Poursuivons notre script de monitoring afin d'envoyer nos résultats vers InfluxDB : nous l'appellerons `./telegraf.sh`.

1.3.8 Rappel d'InfluxDB

Commençons par lancer le conteneur Docker d'InfluxDB (pour éviter de l'installer sur notre machine) :

```
docker container run --name mytsdb -d -p 8086:8086 influxdb:1.8
```

Nous utilisons la version 1.8 d'influxDB qui est plus simple à administrer pour faire des tests. Vous pouvez partir sur la version 2, une API compatible avec la version 1 est disponible, elle est plus simple à utiliser à partir d'un shell.

Il nous faut ensuite créer une base de données pour y stocker nos métriques. Voici comment on s'était débrouillé précédemment pour interagir avec InfluxDB :

```
docker container exec -i mytsdb influx <<EOF
CREATE DATABASE metrics;
SHOW DATABASES;
EOF
```

Vérifiez que la base de données `metrics` a bien été créée.

[2] https://lwn.net/Articles/604609/
[3] https://lwn.net/Articles/679786/

1.3.9 Monitoring vers InfluxDB

Maintenant, envoyons nos données vers la base https://docs.influxdata.com/influxdb/v1.8/guides/write_data/ :

```
curl -i -XPOST 'http://localhost:8086/write?db=metrics' --data-binary \
    "$my_cgroup_name memory.usage_in_bytes=$(cat .../my_cgroup_name/memory.
    usage_in_bytes)"
```

Pour vérifier que les données ont bien été ajoutées, nous pouvons effectuer la requête suivante dans notre client `influx` :

```
SELECT * from "$my_cgroup_name";
```

1.3.10 Monitorer davantage de données

Liste non exhaustive de données à monitorer :

— Nombre d'IOs effectué ;
— nombre d'octets lus/écrits sur les disques ;
— temps de calcul utilisé (`userspace`, `system`, tout confondu) ;
— ...

Tous les cgroups existants dans le dernier noyau publié ont leur documentation accessible ici :

— v1 : https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v1/index.html
— v2 : https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v2.html

1.3.11 Permettre à l'utilisateur de monitorer des processus

Maintenant, séparons notre script en deux parties afin qu'un utilisateur normal (non-root) puisse utiliser la partie monitoring de notre script. La procédure sera différente suivant la version des *cgroup*s que vous utilisez.

1.3.11.1 cgroups v1

Un premier script doit s'occuper de créer le(s) *cgroup*s et lui attribuer les bons droits (`chown $EUID`), tandis que le deuxième va effectuer le monitoring, sans privilèges particuliers.

Exemple

```
42sh$ sudo ./telegraf_init.sh my_cgroup_name
42sh$ ./telegraf.sh my_cgroup_name memhog 500
```

1.3.11.2 cgroups v2

On part du principe que `systemd` est le système d'initialisation de votre machine et qu'il gère lui-même l'arborescence.

Dans cette version des *cgroup*s, on peut toujours déléguer une partie de l'arborescence à un utilisateur, au moyen d'un `chown`, cependant une contrainte supplémentaire entre en jeu. Lorsque

l'on veut déplacer un processus, il faut avoir les droits d'écriture à la fois sur la destination, mais également sur la source, et sur tous les nœuds de l'arborescence que l'on peut croiser lors du parcourt de l'arbre.

Cela signifie, qu'après avoir `mkdir -p virli/child1 virli/child2 && chown -R $EUID:$EUID virli`, il est possible de déplacer un processus de :

```
/virli/child1
```

vers

```
/virli/child2
```

car on dispose bien des droits d'écriture sur l'ancêtre commun le plus proche (à savoir `/virli`).

Les problèmes surviennent lorsque l'on souhaite déplacer un processus que `systemd` a placé dans :

```
/user.slice/user@$UID.service/app.slice/app-alacritty-0a1b2c3d4e5f6e7d9c
```

Il ne sera alors pas possible de le déplacer vers `/virli/child1`, car l'ancêtre commun le plus proche est la racine des *cgroup*s, sur laquelle l'utilisateur n'a pas les droits d'écriture.

Pour faire cet exercice, vous allez donc devoir rechercher l'ancêtre commun le plus proche sur lequel votre utilisateur a les droits d'écritures.

En utilisant `systemd` :

```
42sh$ systemctl --user show | grep ControlGroup
ControlGroup=/user.slice/user-1234.slice/user@1234.service
```

Ou bien en recherchant dans l'arborescence des *cgroup*s votre processus :

```
42sh$ find /sys/fs/cgroup/ -name cgroup.procs -exec grep -l $$ {} \;
```

… puis en remontant tant que l'on dispose des droits d'écritures.

Le script `telegraf_init.sh` devra retourner le chemin vers le dossier (éventuellement) créé à partir du nom passé en paramètre, depuis la racine des *cgroup*s.

Ce retour servira de premier argument au script `telegraf.sh`.

Exemple

```
42sh$ ./telegraf_init.sh my_cgroup_name
/user.slice/user@1234.service/my_cgroup_name
42sh$ ./telegraf.sh "/user.slice/user@1234.service/my_cgroup_name" memhog 500
```

2 Gestion de la mémoire

Linux a une gestion de la mémoire bien particulière[4] : en effet, par défaut, `malloc(3)` ne retournera jamais `NULL`. En se basant sur l'euristique qu'un bloc mémoire demandé ne sera pas utilisé directement et que de nombreux process ne feront pas un usage total des blocs qu'ils ont alloués, le noyau permet d'allouer plus de mémoire qu'il n'y en a réellement disponible. La mémoire est ainsi utilisée de manière plus efficace.

Mais évidemment, cela peut donner lieu à des situations où, au moment où un processus se met à utiliser un nouveau bloc de mémoire (reçu du noyau d'un appel précédent à `malloc(3)`) qu'il n'utilisait pas jusque là, le noyau se trouve dans l'impossibilité d'attribuer un bloc physiquement disponible, car il n'y en a tout simplement plus (y compris via le swap).

Puisque le noyau ne peut pas honorer sa promesse et qu'il n'a plus la possibilité de retourner `NULL` au programme qui réclamme sa mémoire (il s'agit sans doute d'une simple assignation de variable à ce stade), il faut trouver une solution si l'on veut pouvoir continuer l'exécution du programme.

Le noyau pourrait choisir de suspendre l'exécution du processus tant qu'il n'y a pas de nouveau bloc mémoire disponible… mais, à moins qu'un processus se termine ou libère de la mémoire, on risque de se retrouver face à une situation de faillite où tous les processus seraient suspendus, sans garantie qu'une solution se produise à un moment donné.

Pour être certain de récupérer de la mémoire, le noyau n'a pas d'autre solution que de tuer un processus. L'issue la plus simple est de tuer le processus qui est en train d'accéder à la plage de mémoire que le noyau ne peut pas honorer. Pour autant, ce n'est pas une raison pour tuer ce processus, car il est peut-être vital pour le système (peut-être est-ce `init` qui est en train de gérer le lancement d'un nouveau daemon). On dit alors que le noyau est *Out-Of-Memory*.

Pour se sortir d'une telle situation, et après avoir tenté de vider les caches, il lance son arme ultime pour retrouver au plus vite de la mémoire : l'*Out-Of-Memory killer*.

2.1 Trouver un coupable

Tous les processus, au cours de leur exécution, disposent d'un score permettant au système de savoir à tout moment quel processus apporterait le plus de gain à être éliminé, si la mémoire venait à manquer.

Lorsqu'une situation d'OOM est déclarée, le noyau tue le processus avec le score le plus élevé à ce moment là.

Pour connaître le score actuel d'un processus, on affiche le contenu du fichier `oom_score` dans son dossier de `/proc` :

```
42sh$ cat /proc/self/oom_score
666
42sh$ cat /proc/1/oom_score
0
```

Le score est établi entre 0 et 1000.

Le but étant de récupérer le plus vite possible, le plus de mémoire possible, le score est établi selon la quantité de mémoire que le processus occupe.

[4] Cela dépend de la valeur de `/proc/sys/vm/overcommit_memory`, généralement 0. Voir https://www.kernel.org/doc/html/latest/vm/overcommit-accounting.html.

Voici un script pour visualiser les programmes ayant les plus gros scores sur votre système :

```
ps -A | while read pid _ _ comm; do
    echo $(cat /proc/$pid/oom_score) $comm;
done | sort -h
```

A priori, la dernière ligne montre le processus ayant le plus de chance d'être tué en cas de passage proche de l'OOM-killer.

À vous de jouer Continuons l'exercice précédent où nous avions fixé les limites de mémoire que pouvaient réserver les processus de notre groupe. Que se passe-t-il alors si `memhog` dépasse la quantité de mémoire autorisée au sein du `cgroup` ?

```
42sh$ echo 512M > /sys/fs/cgroup/memory/MYNAME/memory.limit_in_bytes
42sh$ ./monitor MYNAME memhog 500
.............. # OK
42sh$ ./monitor MYNAME memhog 700
.............Killed
```

Eh oui, l'OOM-killer passe également lorsqu'un `cgroup` atteint la limite de mémoire qui lui est réservé. Dans ce cas évidemment, les processus pris en compte sont ceux contenus dans le `cgroup`.

2.2 Esquiver l'OOM killer ?

Le passage de l'OOM killer relevant parfois de la roulette russe, il peut être intéressant, pour certains processus, de vouloir faire en sorte qu'ils aient moins de chance d'arriver en tête du classement, même s'ils occupent beaucoup de mémoire. On pourrait par exemple vouloir que des services importants ou des programmes contenant notre travail en cours (potentiellement non-enregistré !) ne soient pas ciblés à moins de ne plus avoir d'autre choix.

Le sujet étant très épineux, et aucune solution ou algorithme n'ayant réellement démontré sa supériorité pour évincer la tâche idéale, nous avons la possibilité de modifier le score à la hausse ou à la baisse.

Le fichier `oom_score_adj` peut contenir un valeur entre -1000 et +1000, cette valeur vient s'ajouter au score du processus à chaque prochain calcul. Une valeur négative va faire réduire le score et réduire d'autant les chances que le processus soit ciblé, tandis qu'une valeur positive va augmenter le score et donc accroître les chances que le processus soit ciblé.

La valeur spéciale de `-1000` fait que le processus ne sera pas considéré comme une cible potentielle. Au même titre que les *thread*s du noyau ou le processus `init` du système.

2.3 Être notifié sur l'état de la mémoire

Au sein d'un *cgroup memory*, les fichiers `memory.oom_control` (v1) ou `memory.events` (v2) peuvent être utilisés afin de recevoir une notification du noyau avant que l'OOM-killer ne s'attaque à un processus de ce groupe.

L'idée est de créer un *eventfd* avec `eventfd(2)`, d'ouvrir le fichier `memory.oom_control` ou `memory.events` du groupe pour lequel on veut être notifié, pour obtenir un *file descriptor* ; enfin écrire dans le fichier

`cgroup.event_control` le numéro de l'*eventfd* puis du *file descriptor* sur `memory.oom_control`, séparé par une espace.

On attend ensuite la notification avec :

```
uint64_t ret;
read(<fd of eventfd()>, &ret, sizeof(ret));
```

D'autres notifications peuvent être mises en place, selon le même principe sur d'autres fichiers, notamment `memory.usage_in_bytes`, pour être prévenu dès lors que l'on franchit dans un sens ou dans l'autre un certain palier. Le palier qui nous intéresse est à indiquer comme troisième argument à `cgroup.event_control`.

Dans la version 2 des *cgroups*, il est même possible d'utiliser `inotify` pour surveiller les changements.

3 Gestion des privilèges

3.1 Les capabilities

Historiquement, dans la tradition UNIX, on distinguait deux catégories de processus :

— les processus *privilégiés* : dont l'identifiant numérique de son utilisateur est 0 ;
— les processus *non-privilégiés* : dont l'identifiant numérique de son utilisateur n'est pas 0.

Lors des différents tests de permission faits par le noyau, les processus privilégiés outrepassaient ces tests, tandis que les autres devaient passer les tests de l'*effective UID*, *effective GID*, et autres groupes supplémentaires…

Dans les années 90, ce système s'est rélévé être un peu trop basique et conduisait régulièrement à des abus, au moyen de vulnérabilités trouvées dans les programmes *setuid root*.

Avant de regarder plus en détail les solutions qui ont été apportées à ce problème, commençons par mettre le doigt sur les difficultés.

3.1.1 `setuid` : être `root` le temps de l'exécution

De manière générale, un processus s'exécute dans le contexte de privilège de l'utilisateur qui l'a démarré. Ainsi, lorsque l'on souhaite supprimer un fichier ou un répertoire, en tant que simple utilisateur on ne pourra supprimer que ses propres fichiers, et en aucun cas on ne pourra supprimer … la racine par exemple.

Il y a cependant des tâches nécessitant des privilèges, que l'on souhaite pouvoir réaliser en tant que simple utilisateur. C'est le cas notamment de la modification de son mot de passe : il serait inconcevable de devoir demander à son administrateur à chaque fois que l'on souhaite modifier son mot de passe. Pourtant bien que l'on puisse lire le fichier `/etc/passwd`, seul `root` peut y apporter des modifications (il en est de même pour `/etc/shadow` qui contient les hashs des mots de passe).

C'est ainsi qu'est apparu le `suid-bit` parmi les modes de fichiers. Lorsque ce bit est défini sur un binaire exécutable, au moment de l'exécution, le contexte passe à celui du propriétaire du fichier (`root` si le propriétaire est `root`, mais cela fonctionne quelque soit le propriétaire du fichier : on ne devient pas `root`, mais bien l'utilisateur propriétaire).

Un autre exemple : pour émettre un ping, il est nécessaire d'envoyer des paquets ICMP. À la différence des datagrammes UDP ou des segments TCP, il n'est pas forcément simple d'envoyer des paquets ICMP lorsque l'on est simple utilisateur, car l'usage du protocole ICMP dans une socket est restreint : il faut soit être super-utilisateur, soit que le noyau ait été configuré pour autoriser certains utilisateurs à envoyer des `ECHO_REQUEST`.

Pour permettre à tous les utilisateurs de pouvoir envoyer des ping, le programme est donc généralement *setuid root*, pour permettre à n'importe quel utilisateur de prendre les droits du super-utilisateur, le temps de l'exécution du programme.

Les problèmes surviennent lorsque l'on découvre des vulnérabilités dans les programmes *setuid root*. En effet, s'il devient possible pour un utilisateur d'exécuter du code arbitraire, ce code sera exécuté avec les privilèges de l'utilisateur *root* ! Dans le cas de `ping`, on se retrouverait alors à pouvoir lire l'intégralité de la mémoire, alors que l'on avait juste besoin d'écrire sur une interface réseau.

3.1.2 Et ainsi les privilèges furent séparés

Depuis Linux 2.2 (en 1998), les différentes actions réclamant des privilèges sont regroupées en catégories que l'on appelle *capabilities*. Chacune donne accès à un certain nombre d'actions, on trouve notamment :

- `CAP_CHOWN` : permet de modifier le propriétaire d'un fichier de manière arbitraire ;
- `CAP_KILL` : permet de tuer n'importe quel processus ;
- `CAP_SYS_BOOT` : permet d'arrêter ou de redémarrer la machine ;
- `CAP_SYS_MODULE` : permet de charger et décharger des modules ;
- et beaucoup d'autres, il y en a environ 41 en tout (ça dépend de la version du noyau) !

(Petit point historique, Linux n'est pas à l'origine du concept de *capabilities*, il s'agit au départ de la norme POSIX 1003.1e, mais celle-ci s'est éteinte après le 17ème *draft*. Il devait y être standardisé de nombreux composants améliorant la sécurité des systèmes POSIX, notamment les *capabilities* POSIX, les *ACL POSIX*, …)

Ainsi, `ping` pourrait se contenter de `CAP_NET_RAW`, une *capability* qui permet notamment d'envoyer des données brutes sur n'importe quelle *socket*, sans passer par les types de *socket* plus restreints, mais accessibles aux utilisateurs.

C'est peut-être encore beaucoup, mais au moins, une vulnérabilité dans `ping` ne permettrait plus d'accéder à tous les fichiers ou à toute la mémoire.

? **Peut-on faire mieux pour `ping` ?** Un paramètre existe bien depuis 2011 dans le noyau : `net.ipv4.ping_group_range`. Mais ce n'est que depuis 2020 que les distributions comme Fedora et Ubuntu se mettent à fournir par défaut une configuration qui permette de se passer de *capabilities* pour lancer `ping`.
Cette option prend un intervalle d'identifiant numérique de groupes autorisés à créer de `ECHO_REQUEST`. Par défaut la valeur invalide de `1 0` est définie, ce qui signifie qu'aucun groupe n'est autorisé à créer des `ECHO_REQUEST` à moins d'être privilégié.

! Nous allons par la suite travailler avec le binaire `ping` pour appréhender les *capabilities*. Si vous vous rendez compte que votre binaire `ping` est dans le cas figure décrit juste avant (avec une distribution ayant mis en œuvre l'option `net.ipv4.ping_group_range`), vous pouvez retrouver le comportement historique en désactivant l'option :

```
42sh# sysctl net.ipv4.ping_group_range="1 0"
```

Pas d'inquiétudes, le paramètre est changé de manière temporaire seulement, au prochain redémarrage il reprendra sa valeur définie par la distribution.

3.1.3 Les ensembles de capabilities

Tout d'abord, il faut noter que chaque *thread* dispose de 5 ensembles de *capabilities*. Au cours de son exécution, il peut changer ses ensembles de *capabilities*. Ceux-ci sont utilisés de la façon suivante :

— ***bounding*** (B) : c'est l'ensemble limitant des *capabilities* qui pourront faire l'objet d'un calcul lors des prochaines exécutions. Quelques soient les *capabilities* que peut nous faire gagner une exécution, si la *capability* ne se trouve pas dans le *bounding set*, elle ne sera pas considérée et il sera impossible de l'obtenir. L'option `--cap-drop` de Docker modifie cet ensemble pour restreindre les *capabilities* disponibles dans un conteneur ;
— ***effective*** (E) : il s'agit des *capabilities* actuellement actives, qui seront vérifiées lors des tests de privilèges ;
— ***permitted*** (P) : ce sont les *capabilities* que la tâche peut placer dans l'ensemble *effective* via `capset(2)`. L'ensemble *effective* ne peut pas avoir de *capabilities* qui ne sont pas dans l'ensemble *permitted* ;
— ***inheritable*** (I) : est utilisé au moment de la résolution des *capabilities* lors de l'exécution d'un nouveau processus. Il s'agit des *capabilities* qui seront transmises au processus fil. À moins d'avoir la *capability* `CAP_SETPCAP`, cet ensemble ne peut pas avoir plus de *capability* que celles présentent dans l'ensemble *permitted* ;
— ***ambient*** (A) : existe depuis Linux 4.3 pour permettre aux utilisateurs non-`root` de conserver des *capabilities* d'une exécution à l'autre (sans que l'ensemble des binaires soient marqués avec toutes les *capabilities* dans leur ensemble **inheritable**)[5]. L'ensemble évolue automatiquement à la baisse si une *capability* n'est plus *permitted*.

3.1.4 Les attributs de fichier étendus

Une grosse majorité des systèmes de fichiers (ext[234], XFS, btrfs, ...) permet d'enregistrer, pour chaque fichier, des attributs (dits attributs *étendus*, par opposition aux attributs *réguliers* qui sont réservés à l'usage du système de fichiers, tels que le propriétaire, le groupe, les modes du fichier, ...).

Sous Linux, les attributs sont regroupés dans des espaces de noms :

— *security* : espace utilisé par les modules de sécurité du noyau, tel que SELinux, ... ;
— *system* : espace utilisé par le noyau pour stocker des objets système, tels que les ACL POSIX ;
— *trusted*: espace dont la lecture et l'écriture est limité au super-utilisateur ;
— *user* : modifiable sans restriction, à partir du moment où l'on est le propriétaire du fichier.

Par exemple, on peut définir un attribut sur un fichier comme cela :

```
42sh$ echo 'Hello World!' > toto
42sh$ setfattr -n user.foo -v bar toto
42sh$ getfattr -d toto
# file: toto
user.foo="bar"
```

Lorsque l'on est propriétaire du fichier, on peut modifier les attributs des espaces *security*, *system* et *user*. Évidemment, *root* peut aussi intervenir et par exemple les ACL POSIX[6] (espace *system*) :

```
42sh$ sudo chown root:root toto && sudo chmod o-r toto
42sh$ cat toto
cat: toto: Permission denied
42sh$ sudo setfacl -m u:$USER:r toto
42sh$ cat toto
Hello World!
```

[5]Le problème ayant donné naissance aux *ambient capabilities* est décrit dans cet échange : https://lwn.net/Articles/636533/
[6]Les ACL POSIX sont des permissions supplémentaires qui viennent s'ajouter aux modes standards du fichier (propriétaire, groupe, reste du monde). Avec les ACL POSIX, on peut doonner des droits à un ou plusieurs utilisateurs ou groupe, de manière spécifique.

Dans cet exemple, bien que les droits UNIX traditionnels ne vous donnent pas accès au fichier, les ACL POSIX vous autorisent à le lire.

Vous pouvez voir ces attributs bruts avec la commande :

```
42sh$ getfattr -d -m "^system" toto
# file: toto
system.posix_acl_access=0sgAAEAD/////AgAEOgDAEAA/////xAABAD////8=
```

Il s'agit d'une structure du noyau encodée en base64 (la valeur débute par `0s`, de la même manière que l'on a l'habitude de reconnaître l'hexadécimal avec `0x`, voir `getfattr(1)`), pas forcément très lisible en l'état. On utilisera plutôt `getfacl` pour la version lisible.

>_

Les structures utilisées pour stocker les ACL POSIX dans les attributs étendus sont définies dans `linux/posix_acl_xattr.h` :

D'abord un en-tête, composé de la version avec laquelle la suite des octets a été enregistrée (actuellement 2) :

```
struct posix_acl_xattr_header {
    uint32_t        a_version;
};
```

Puis on trouve directement la liste d'entrée(s) :

```
struct posix_acl_xattr_entry {
    uint16_t        e_tag;
    uint16_t        e_perm;
    uint32_t        e_id;
};
```

Le *tag* identifie de quel type d'entrée il s'agit (propriétaire, utilisateur, groupe, masque, reste du monde, ...). Les *perm*issions sont un champ de bits pour la lecture, écriture et exécution. Enfin l'*id*entifiant renseigne sur le numéro d'utilisateur ou de groupe à qui s'applique l'entrée.

Les constantes utilisées sont définies dans `linux/posix_acl.h`.

3.1.5 Capabilities et attributs étendus pour `ping`

De la même manière que l'on peut définir de façon plus fine les droits d'accès par utilisateur, un attribut de l'espace de nom *security* peut être défini pour accroître les *capabilities* d'un processus lorsqu'il est lancé par un utilisateur non-privilégié. On peut voir le *setuid root* comme l'utilisation de cet attribut, qui accroîtrait l'ensemble des *capabilities*.

Si votre distribution profite de ces attributs étendus, vous devriez obtenir :

```
42sh$ getfattr -d -m "^security" $(which ping)
# file: bin/ping
security.capability=0sAQAAAgAgAAAAAAAAAAAAAAAAAAA=
```

Suivant votre distribution, d'autres programmes sont susceptibles de profiter des attributs étendus pour se passer du *setuid root*, par exemple : `chvt`, `beep`, `iftop`, `mii-tool`, `mtr`, `nethogs`, ...

Comme pour les ACL POSIX, une structure du noyau est enregistrée comme attribut du fichier ; et on peut l'afficher dans sa version plus lisible avec `getcap` :

```
42sh$ getcap $(which ping)
/bin/ping = cap_net_raw+ep
```

>_

La structure utilisée pour stocker les *capabilities* dans les attributs étendus est définie dans `linux/capability.h` :

```
struct vfs_cap_data {
    uint32_t   magic_etc;
    struct {
        uint32_t permitted;
        uint32_t inheritable;
    } data[VFS_CAP_U32];
};
```

La valeur `magic` contient la version sur 1 octet, puis 3 octets sont réservés pour des *flags*. Actuellement un seul *flag* existe, il s'agit de `VFS_CAP_FLAGS_EFFECTIVE` qui détermine si la liste effective de *capabilities* du programme doit être remplie avec les *capabilities permitted* si elle doit rester vide (auquel cas ce sera au programme de s'ajouter les *capabilities* au cours de l'exécution).

Deux entiers de 32 bits représentent ensuite les *capabilities* de 0 à 31 sous forme de champ de bits, un entier pour la liste des *capabilities* permises, un autre pour les *capabilities* héritables. Comme il y a une quarantaine de *capabilities*, celles de 32 à 40 se retrouvent à la suite, dans deux nouveaux entiers de 32 bits. C'est pour cela que `data` est un tableau, avec `VFS_CAP_U32` valant 2, car on a deux fois nos 2 entiers de 32 bits à la suite.

Il s'agit de la version 2 de la structure. La version 1 était utilisée lorsqu'il n'y avait encore que moins de 33 *capabilities* : il n'y avait alors pas de tableau, seulement les deux entiers de 32 bits. On remarque que les deux versions sont facilement compatibles entre-elles, la seconde version étendant simplement la première.

Une version 3 existe, la structure obtient un champ supplémentaire `rootid` :

```
struct vfs_ns_cap_data {
    uint32_t   magic_etc;
    struct {
        uint32_t permitted;
        uint32_t inheritable;
    } data[VFS_CAP_U32];
    uint32_t rootid;
};
```

Ce nouveau champ `rootid` est utilisé pour stocker un identifiant d'utilisateur `root` au sein d'un *namespace* User. C'est utile pour pouvoir jouer avec les *capabilities* au sein d'un conteneur non-privilégié. S'il vaut autre chose que 0 et que l'on ne se trouve pas dans un *namespace* User, les *capabilities* ne seront pas appliquées.

3.1.6 Gagner des capabilities

C'est à l'exécution (`execve(2)`) que sont calculés les nouveaux ensembles de *capabilities* : c'est donc uniquement à ce moment que l'on peut en gagner de nouvelles. Voici comment les différents ensembles du nouveau processus sont calculés :

$$p'_A = (\text{file caps or setuid or setgid } ?\ \emptyset\ :\ p_A)$$
$$p'_P = (p_I\ \&\ f_I)\ |\ (f_P\ \&\ p_B)\ |\ p'_A$$
$$p'_E = f_E\ ?\ p'_P\ :\ p'_A$$
$$p'_I = p_I$$
$$p'_B = p_B$$

Avec p le processus avant l'exécution, f le fichier exécutable donnant naissance au nouveau processus et p' le nouveau processus.

On remarque que sans les *ambients capabilities*, on perd systématiquement les *capabilities* dont on disposait avant l'`execve(2)`, car f_I n'est que très rarement défini sur un exécutable. Dans le cas général, on récupère donc soit f_P, soit p_A (les deux étant exclusifs : si f_P est défini, p'_A est vide).

Bien entendu, lorsque l'on se trouve dans le contexte d'exécution de `root` (ou que l'on exécute un binaire *setuid root*), ces calculs sont biaisés, car le super-utilisateur a normalement toutes les *capabilities* (mais toujours limitées par l'ensemble *bounding*) : f_P et f_I contiennent alors toutes les *capabilities*, indifféremment du fichier considéré. Les calculs peuvent alors être simplifiés en :

$$p'_P = (p_I\ \&\ p_B)$$
$$p'_E = p'_P$$

Il y a cependant une exception, lorsque l'utilisateur réel n'est pas `root` (comme par exemple face à un binaire *setuid root*, seul l'utilisateur effectif change) : dans ce cas, si le fichier expose des *capabilities*, seulement celles-ci sont gagnées.

? **Peut-on placer des capabilities sur un script ?** De la même manière qu'il n'est pas possible d'avoir de script *setuid root* sous Linux[a], ajouter des *capabilities* à un script ne permettra pas d'en gagner. Le calcul s'effectue en effet sur les *capabilities* de l'exécutable de l'interpréteur et non sur celles du script.

[a] https://unix.stackexchange.com/a/2910

3.1.7 Gérer ses capabilities

Un *thread* peut agir comme il le souhaite sur les ensembles *effective*, *permitted* et *inheritable*. À condition bien sûr de ne jamais dépasser les *capabilities* déjà contenues dans l'ensemble *permitted*, sauf si l'on dispose de la *capability* `CAP_SETPCAP` : dans ce cas, on se retrouve limité seulement par l'ensemble *bounding*.

>_

On utilise les appels système `capget(2)` et `capset(2)` pour respectivement connaître les *capabilities* actuelles de notre fil d'exécution et pour les écraser. Ces appels système utilisent une structure d'en-tête et une structure de données, qui sont définies dans `linux/capability.h` :

```
#define _LINUX_CAPABILITY_VERSION_1  0x19980330
#define _LINUX_CAPABILITY_U32S_1     1

        /* V2 added in Linux 2.6.25; deprecated */
#define _LINUX_CAPABILITY_VERSION_2  0x20071026
#define _LINUX_CAPABILITY_U32S_2     2

        /* V3 added in Linux 2.6.26 */
#define _LINUX_CAPABILITY_VERSION_3  0x20080522
#define _LINUX_CAPABILITY_U32S_3     2

typedef struct __user_cap_header_struct {
   uint32_t version;
   int pid;
} *cap_user_header_t;

typedef struct __user_cap_data_struct {
   uint32_t effective;
   uint32_t permitted;
   uint32_t inheritable;
} *cap_user_data_t;
```

La structure d'en-tête permet de renseigner sur la version de la structure de données que l'on souhaite utiliser ou recevoir. Comme pour les *capabilities* dans les attributs étendus, la première version était utilisée lorsqu'il y avait moins de 33 *capabilities*, ce qui permettait de tout stocker sur un seul entier de 32 bits non signé. Les versions 2 et 3 sont identiques et permettent de récupérer les *capabilities* au moyen d'un tableau de 2 structures.

!

Les versions 2 et 3 ici ne sont pas comparables aux versions 2 et 3 de nos structures `vfs_cap_data` et `vfs_ns_cap_data` : il n'y a pas de notion de *namespace* ici.
Il y a eu un couac dans les en-têtes distribués avec Linux 2.6.25, causant des *buffers overflow* dans les applications qui n'avaient pas prévu de gérer les versions. Cela a été corrigé dans la version 2.6.26 : un avertissement est consigné dans les journaux système en cas d'utilisation malheureuse de la version 2.

Dans la pratique, on préférera utiliser `cap_get_proc(3)` et `cap_set_proc(3)` fournis par la `libcap`.

Pour agir sur l'ensemble *bounding*, il faut disposer de la *capability* `CAP_SETPCAP`[7]. Lorsque l'on retire une *capability* de cet ensemble, elle n'est pas retirée des autres ensembles et on peut donc continuer de bénéficier des privilèges qu'elle accorde.

Il faut bien penser, lorsque l'on retire une *capability* de l'ensemble *bounding*, à également la retirer de l'ensemble *inheritable*, sans quoi si le programme exécuté a la *capability* en question dans ses attributs, celle-ci sera préservée (revoir la formule pour p'_P, l'ensemble *bounding* n'est pas considéré avec l'ensemble

[7]En effet, avant Linux 2.6.25, cet ensemble était utilisé par tout le système, pas seulement pas le *thread* courant et ses fils.

inheritable).

>_ La consultation et la modification de l'ensemble *bounding* se fait au moyen de `prctl(2)`, en utilisant les paramètres `PR_CAPBSET_READ` ou `PR_CAPBSET_DROP`.
Le second paramètre attendu est l'une des constantes représentant une *capability*.

Avec `PR_CAPBSET_READ`, `prctl(2)` retournera 0 si la *capability* ne fait pas partie de l'ensemble *bounding*, ou 1 si elle est bien présente.
Avec `PR_CAPBSET_DROP`, `prctl(2)` retirera la *capability* passée en argument de l'ensemble *bounding*. Une fois cette action effectuée, il est impossible de revenir en arrière.

Dans la pratique, on préférera utiliser `cap_get_bound(3)` et `cap_drop_bound(3)` fournis par la `libcap`.

Enfin, le *thread* peut aussi modifier à sa guise l'ensemble *ambient*, à condition que les *capabilities* ajoutées soient dans les ensembles *permitted* et *inheritable*.

>_ On consulte et modifie l'ensemble *ambient* avec `prctl(2)` à qui l'on passe comme premier paramètre `PR_CAP_AMBIENT`. Le second paramètre permet de préciser l'action que l'on veut réaliser :

- `PR_CAP_AMBIENT_RAISE` : ajoute la *capability* précisée comme troisième paramètre ;
- `PR_CAP_AMBIENT_LOWER` : retire la *capability* précisée comme troisième paramètre ;
- `PR_CAP_AMBIENT_IS_SET` : retourne 1 si la *capability* précisée comme troisième paramètre fait partie de l'ensemble *ambient*, sinon retourne 0 ;
- `PR_CAP_AMBIENT_CLEAR_ALL` : vide l'ensemble *ambient*.

3.1.8 Explorer les capabilities avec un shell

La `libcap`, au travers des paquets `libcap2-bin` (Debian et ses dérivées) ou `libcap` (Alpine, Archlinux, Gentoo et leurs dérivées), apporte le programme `capsh(1)`, très utile pour appréhender les *capabilities*.

```
42sh# capsh --drop=cap_net_raw -- -c "/bin/ping nemunai.re"
/bin/bash: line 1: /bin/ping: Operation not permitted
```

Une autre commande intéressante est `pscap(1)`, de la `libcap-ng`. Parmi tous les programmes en cours d'exécution, cet utilitaire va afficher tous les programmes s'exécutant actuellement avec des *capabilities*, en indiquant pour chacun lesquelles sont actives et disponibles.

Visualisateur de capabilities d'un processus

Écrivons maintenant un programme permettant de voir les *capabilities* d'un processus :

```
42sh$ ./view_caps 1
cap_user_header_t
-----------------
Version: 20080522
```

```
PID: 1

cap_user_data_t
---------------
effective: 0x3fffffffff
    CAP_AUDIT_CONTROL
    CAP_AUDIT_READ
[...]
    CAP_SYS_TIME
    CAP_SYS_TTY_CONFIG
    CAP_SYSLOG
    CAP_WAKE_ALARM
permitted: 0x3fffffffff
    CAP_AUDIT_CONTROL
    CAP_AUDIT_READ
[...]
    CAP_SYS_TIME
    CAP_SYS_TTY_CONFIG
    CAP_SYSLOG
    CAP_WAKE_ALARM
inheritable: 0x0
```

Appelé sans argument, `view_caps` affichera les *capabilities* du processus courant, avec en plus les informations sur son ensemble *ambient* et *bounding* :

```
42sh# ./view_caps
cap_user_header_t
-----------------
Version: 20080522
PID: 42

cap_user_data_t
---------------
effective: 0x3fffffffff
    CAP_AUDIT_CONTROL
    CAP_AUDIT_READ
[...]
    CAP_SYSLOG
    CAP_WAKE_ALARM
permitted: 0x3fffffffff
    CAP_AUDIT_CONTROL
    CAP_AUDIT_READ
[...]
    CAP_SYSLOG
    CAP_WAKE_ALARM
inheritable: 0x0

ambient:
    CAP_AUDIT_CONTROL
    CAP_AUDIT_READ
```

```
[...]
    CAP_SYSLOG
    CAP_WAKE_ALARM

bounding:
    CAP_AUDIT_CONTROL
    CAP_AUDIT_READ
[...]
    CAP_SYSLOG
    CAP_WAKE_ALARM
```

```
42sh$ ./view_caps
cap_user_header_t
-----------------
Version: 20080522
PID: 42

cap_user_data_t
---------------
effective: 0x0
permitted: 0x0
inheritable: 0x0

ambient:

bounding:
    CAP_AUDIT_CONTROL
    CAP_AUDIT_READ
[...]
    CAP_SYSLOG
    CAP_WAKE_ALARM
```

Il peut être intéressant de lancer `view_caps` au sein d'un conteneur Docker pour voir évoluer l'ensemble *bounding*, ou bien d'utiliser `capsh(1)` en amont.

Pour aller plus loin

Je vous recommande la lecture des *man* suivants :

— `capabilities(7)` : énumérant tous les *capabilities*, leur utilisation, etc. ;
— `xattrs(7)` : à propos des attributs étendus.

Et de ces quelques articles :

— Linux Capabilities: Why They Exist and How They Work : https://blog.container-solutions.com/linux-capabilities-why-they-exist-and-how-they-work
— Guidelines for extended attributes : https://www.freedesktop.org/wiki/CommonExtendedAttributes/
— File-based capabilities : https://lwn.net/Articles/211883/

— A bid to resurrect Linux capabilities : https://lwn.net/Articles/199004/
— False Boundaries and Arbitrary Code Execution : https://forums.grsecurity.net/viewtopic.php?f=7&t=2522#p10271
— Linux Capabilities on HackTricks : https://book.hacktricks.xyz/linux-unix/privilege-escalation/linux-capabilities
— POSIX Access Control Lists on Linux : https://www.usenix.org/legacy/publications/library/proceedings/usenix03/tech/freenix03/full_papers/gruenbacher/gruenbacher_html/main.html

3.2 Secure Computing Mode

Plus connue sous l'acronyme *seccomp*, cette fonctionnalité du noyau Linux permet de restreindre les appels système qu'un processus est autorisé à utiliser. En cas d'appel non autorisé, le processus fautif est directement tué (`SIGKILL`) par le noyau, ou, lorsque c'est précisé, un code `errno` particulier peut être renvoyé au programme.

Depuis la version 3.17 du noyau, l'appel système `seccomp(2)` permet de faire entrer le processus courant dans ce mode. En effet, c'est le processus lui-même qui déclare au noyau qu'il peut désormais se contenter d'une liste réduite d'appels système ; à l'inverse des politiques de sécurité comme SELinux ou AppArmor, qui encapsulent les programmes pour toute la durée de leur exécution.

Seccomp est particulièrement utile lorsqu'un processus a terminé son initialisation (ne dépendant en général pas de données sujettes à l'exploitation de vulnérabilité) et doit commencer à entrer dans des portions de code promptes aux vulnérabilités : c'est notamment le cas des moteurs de rendus des navigateurs Firefox et Chrome.

3.2.1 Prérequis

L'utilisation de `seccomp` nécessite d'avoir un noyau compilé avec l'option `CONFIG_SECCOMP`.

La bibliothèque `libseccomp` est également indispensable car l'appel système `seccomp(2)` n'a pas de *wrapper* dans la libc, vous devrez donc passer par cette bibliothèque.

3.2.2 `MODE_STRICT`

Le mode traditionnel de *seccomp* est de ne permettre uniquement l'utilisation des appels système `read(2)`, `write(2)` (sur les descripteurs de fichier déjà ouvert), `_exit(2)` et `sigreturn(2)`.

Historiquement, avant la création de l'appel système `seccomp(2)`, on activait ce mode via :

```
prctl(PR_SET_SECCOMP, SECCOMP_MODE_STRICT);
```

Une fois passé cet appel système, toute entrée dans un appel système non autorisé conduit à un `SIGKILL` du processus.

3.2.3 `MODE_FILTER`

Plus modulable que le mode strict, le mode de filtrage permet une grande amplitude en permettant au programmeur de définir finement quels appels système le programme est autorisé à faire ou non, et quelle sentence est exécutée en cas de faute.

Notons que les processus fils issus (`fork(2)` ou `clone(2)`) d'un processus auquel est appliqué un filtre `seccomp`, héritent également de ce filtre.

La construction de ce filtre est faite de manière programmatique, via des règles BPF (`Berkeley Packet Filter`). On passe ensuite ce filtre BPF en argument de l'appel système :

```
struct sock_filter filter[];
struct sock_fprog prog = {
    .len = (unsigned short) (sizeof(filter) / sizeof(filter[0])),
    .filter = filter,
};
seccomp(SECCOMP_SET_MODE_FILTER, 0, &prog);
```

Écrivez un programme filtrant un appel système, à l'aide de `seccomp` :

```
42sh$ ./syscall_filter sleep 5
sleep: cannot read realtime clock: Operation not permitted
42sh$
```

Dans cet exemple, l'appel système filtré est `nanosleep(2)` (on utilise `strace(2)` pour le découvrir).

4 Registres

Nous allons appréhender le fonctionnement d'un registre OCI, en essayant de récupérer les couches de quelques images (Debian, Ubuntu, hello, ...) : dans un premier temps en nous préoccupant simplement de la couche la plus basse (qui ne contient pas de modification ou de suppression : chaque fichier est normal).

4.1 Authentification

L'authentification est facultative et est laissée à l'appréciation du fournisseur de service. Étant donné que nous allons utiliser le Docker Hub, le registre par défaut de `docker`, nous allons devoir nous plier à son mécanisme d'authentification : chaque requête au registre doit être effectuée avec un jeton, que l'on obtient en s'authentifiant auprès d'un service dédié. Ce service peut délivrer un jeton sans authentifier l'interlocuteur, en restant anonyme ; dans ce cas, on ne pourra accéder qu'aux images publiques. Ça tombe bien, c'est ce qui nous intéresse !

Il n'en reste pas moins que le jeton est forgé pour un service donné (ici `registry.docker.io`) et avec un objectif bien cerné (pour nous, on souhaite récupérer le contenu du dépôt[8] `hello-world` : `repository:hello-world:pull`). Ce qui nous donne :

```
42sh$ curl "https://auth.docker.io/token?service=registry.docker.io&" \
  "scope=repository:library/hello-world:pull" | jq .
```

```
{
  "token": "lUWXBCZzg2TGNUdmMy...daVZxGTj0eh",
  "access_token": "eyJhbGciOiJSUzI1NiIsI...N5q469M3ZkL_HA",
  "expires_in": 300,
  "issued_at": "2012-12-12T12:12:12.123456789Z"
}
```

C'est le `token` qu'il faudra fournir lors de nos prochaines requêtes au registre.

Avec `jq`, on peut l'extraire grâce à :

```
| jq -r .token
```

Le token expire ! Pensez à le renouveler régulièrement.

En cas d'erreur inexplicable, vous pouvez ajouter un `-v` à la ligne de commande `curl`, afin d'afficher les en-têtes. Prêtez une attention toute particulière à `Www-Authenticate`.

4.2 Lecture de l'index d'images

Une fois en possession de notre jeton, nous pouvons maintenant demander l'index d'images à notre registre :

[8] Dans un registre, les fichiers qui composent l'image forment un dépôt (*repository*).

```
curl -s \
  -H "Authorization: Bearer ${TOKEN}" \
  -H "Accept: application/vnd.docker.distribution.manifest.list.v2+json" \
  "https://registry-1.docker.io/v2/library/hello-world/manifests/latest" | jq .
```

Dans la liste des *manifests* retournés, nous devons récupérer son `digest`. Dans tout l'écosystème OCI, les `digest` servent à la fois de chemin d'accès et de somme de contrôle.

4.3 Lecture du manifest

Demandons maintenant le *manifest* correspondant à notre matériel et à notre système d'exploitation :

```
curl -s \
  -H "Authorization: Bearer ${TOKEN}" \
  -H "Accept: ${MEDIATYPE}" \
  "https://registry-1.docker.io/v2/library/hello-world/manifests/
    ${MNFST_DGST}"
```

Nous voici donc maintenant avec le *manifest* de notre image. Nous pouvons constater qu'il n'a bien qu'une seule couche, ouf !

4.4 Récupération de la configuration et de la première couche

Les deux éléments que l'on cherche à récupérer vont se trouver dans le répertoire `blobs`, il ne s'agit en effet plus de *manifest*. Si les *manifests* sont toujours stockés par le registre lui-même, les blobs peuvent être délégués à un autre service, par exemple dans le cloud, chez Amazon S3, un CDN, etc.

Pour récupérer la configuration de l'image :

```
curl -s --location \
  -H "Authorization: Bearer ${TOKEN}" \
  "https://registry-1.docker.io/v2/library/hello-world/blobs/${CNF_DIGEST}"
```

Enfin, armé du `digest` de notre couche, il ne nous reste plus qu'à la demander gentiment :

```
wget --header "Authorization: Bearer ${TOKEN}" \
  "https://registry-1.docker.io/v2/library/hello-world/blobs/${LAYER_DGST}"
```

4.5 Extraction

Le type indiqué par le *manifest* pour cette couche était :

```
application/vnd.docker.image.rootfs.diff.tar.gzip
```

Il s'agit donc d'une tarball compressée au format gzip :

```
mkdir rootfs
tar xzf ${DL_LAYER} -C rootfs
```

Et voilà, nous avons extrait notre première image, nous devrions pouvoir :

```
42sh# chroot rootfs /hello
Hello from Docker!
[...]
```

Assemblage Réalisez un script pour automatiser l'ensemble de ces étapes :

```
42sh$ cd $(mktemp)

42sh$ ./registry_play library/hello-world:latest

42sh$ find
.
./rootfs
./rootfs/hello

42sh# chroot rootfs /hello
Hello from Docker!
[...]
```

5 En route vers la contenerisation

Nous avons vu un certain nombre de fonctionnalités offertes par le noyau Linux pour limiter, autoriser ou contraindre différents usages des ressources de notre machine.

Il est temps maintenant de commencer à parler d'isolation, mais … cela fait déjà beaucoup à digérer pour aujourd'hui, alors on va se contenter de parler d'un mécanisme que vous connaissez sans doute déjà.

5.1 L'isolation … avec chroot

Depuis les premières versions d'Unix, il est possible de changer le répertoire vu comme étant la racine du système de fichiers. En anglais : *change root*: `chroot`. Le processus effectuant cette action ainsi que tous ses fils verront donc une racine différente du reste du système.

>_

Le noyau stocke le chemin de la racine courante dans les informations de notre processus :

```
// From linux/sched.h
struct task_struct {
    [...]
    /* Filesystem information: */
    struct fs_struct        *fs;
    [...]
};

// From linux/fs_struct.h
struct fs_struct {
    int users;
    spinlock_t lock;
    seqcount_spinlock_t seq;
    int umask;
    int in_exec;
    struct path root;
    struct path pwd;
};
```

Ici dans `root`.
On retrouve la racine exposée sous forme de lien symbolique dans `/proc/$$/root`.

Pour se créer un environnement afin de changer notre racine, il va falloir commencer par créer le dossier de notre nouvelle racine, peu importe où dans l'arborescence :

```
mkdir newroot
```

Nous allons ensuite remplir ce dossier afin qu'il soit vraiment utilisable comme une racine : rien n'est strictement obligatoire, on s'assure simplement d'avoir de quoi bidouiller : un shell sera amplement suffisant pour commencer.

5.1.1 busybox

Queques mots, pour commencer, à propos du projet Busybox : c'est un programme couteau-suisse qui implémente tous les binaires vitaux pour avoir un système fonctionnel et utilisable : `ls`, `sh`, `cat`, mais aussi `init`, `mdev` (un `udev`-like, cela permet de découvrir les périphériques attachés afin de les exposer dans `/dev` notamment). C'est un programme *linké* statiquement, c'est-à-dire qu'il ne va pas chercher ni charger de bibliothèque dynamique à son lancement. Il se suffit donc à lui-même dans un *chroot*, car il n'a pas de dépendances. Nous pouvons donc tester notre première isolation :

```
cp $(which busybox) newroot/
chroot newroot /busybox ash
```

Nous voici donc maintenant dans un nouveau shell (il s'agit d'`ash`, le shell de `busybox`).

?

Faut-il être `root` pour faire un `chroot` ? Oui, seul un utilisateur avec la *capability* `CAP_SYS_CHROOT` peut le faire !

En fait, il est possible de duper le système avec un fichier `/etc/passwd` et `/etc/shadow` que l'on maîtrise, et un binaire *setuid root* tel que `su` : `su` pourra nous demander le mot de passe `root` ou d'un autre utilisateur, mais comme on a la maîtrise du fichier `/etc/shadow`, on aura mis préalablement une valeur qui nous arrange.

Jusque-là … ça fonctionne, rien de surprenant ! Mais qu'en est-il pour `bash` :

```
42sh$ cp $(which bash) newroot/
42sh# chroot newroot /bash
chroot: failed to run command 'bash': No such file or directory
```

De quel fichier est-il question ici ?

5.1.2 debootstrap, pacstrap

`debootstrap` est le programme utilisé par l'installeur des distributions Debian et ses dérivés. Il permet d'installer dans un dossier (en général, ce dossier correspond au point de montage de la nouvelle racine choisie par l'utilisateur lors de l'installation) le système de base.

```
debootstrap bullseye newroot/ http://httpredir.debian.org/debian/
```

`pacstrap` est le programme équivalent pour Arch Linux. Alors que `debootstrap` peut s'utiliser depuis n'importe quel environnement ou distribution, `pacstrap` nécessite d'avoir installé et configuré `pacman` (le gestionnaire de paquets d'Arch Linux), ce qui est le cas si vous êtes sous Arch Linux ou ses dérivés.

```
pacstrap newroot/
```

Dans les deux cas, nous nous retrouvons avec un dossier `newroot` contenant une distribution complète minimale, dans laquelle nous pouvons entrer :

```
chroot newroot/ bash
```

5.1.3 Gentoo

http://gentoo.mirrors.ovh.net/gentoo-distfiles/releases/amd64/autobuilds/current-stage3-amd64-openrc/stage3-amd64-openrc-20211128T170532Z.tar.xz

```
tar xpf stage3-amd64-*.tar.xz -C newroot/
```

(L'avantage de télécharger l'archive de Gentoo est que l'on a déjà gcc dans un environnement qui tient dans 200 MB.)

Comme pour les autres distributions vues précédemment, nous pouvons entrer dans notre nouvelle racine comme ceci :

```
chroot newroot/ bash
```

5.1.4 Alpine

https://dl-cdn.alpinelinux.org/alpine/v3.14/releases/x86_64/alpine-minirootfs-3.14.2-x86_64.tar.gz

```
tar xpf alpine-minirootfs-*.tar.xz -C newroot/
```

Alpine se contentant de Busybox pour son système de base, nous n'avons pas `bash`, mais on peut tout de même lancer `ash` :

```
chroot newroot/ ash
```

5.1.5 Utiliser une image OCI ?

Pourquoi pas : nous avons réalisé précédemment un script pour interagir avec le registre, c'est le moment de l'utiliser !

```
./registry_play.sh library/hello-world:latest
```

Si on se contente de l'image `hello-world`, on va pouvoir exécuter le binaire principal :

```
chroot rootfs/ /hello
```

Bien sûr, des images de base comme `debian` ou `alpine` devraient fonctionner également sans difficulté.

Exercice (SRS seulement)

Écrivons maintenant un programme dont le seul but est de s'échapper du `chroot` :

```
make escape
echo bar > ../foo
chroot .
```

Dans le nouvel environnement, vous ne devriez pas pouvoir faire :

```
cat ../foo
```

Mais une fois votre programme `escape` exécuté, vous devriez pouvoir !

```
(chroot) 42sh# ./escape
  bash# cat /path/to/foo
```

6 Rendu

Est attendu d'ici le cours suivant :

— vos réponses à l'évaluation du cours,
— [SRS] tous les exercices de ce TP,
— [GISTRE] les premiers paliers du projet final.

6.1 Arborescence attendue (SRS)

Tous les fichiers identifiés comme étant à rendre sont à placer dans un dépôt Git privé, que vous partagerez avec votre professeur.

Voici une arborescence type (vous pourriez avoir des fichiers supplémentaires) :

```
./pseudofs/procinfo
./pseudofs/cpuinfo.sh // batinfo.sh
./pseudofs/suspend_schedule.sh // rev_kdb_leds.sh
./cgroups/monitor
./cgroups/telegraf
./cgroups/telegraf_init
./caps/view_caps.c
./seccomp/syscall_filter.c
./docker/registry_play
./chroot/escape.c
```

Votre rendu sera pris en compte en faisant un tag **signé par votre clef PGP**. Consultez les détails du rendu (nom du tag, ...) sur la page dédiée au projet sur la plateforme de rendu.

?

Si vous utilisez un seul dépôt pour tous vos rendus, vous **DEVRIEZ** créer une branche distincte pour chaque rendu :

```
42sh$ git checkout --orphan renduX
42sh$ git reset
42sh$ rm -r *
42sh$   # Créer l'arborescence de rendu ici
```

Pour retrouver ensuite vos rendus des travaux précédents :

```
42sh$ git checkout renduY
-- ou --
42sh$ git checkout master
...
```

Chaque branche est complètement indépendante l'une de l'autre. Vous pouvez avoir les exercices du TP1 sur `master`, les exercices du TP3 sur `rendu3`, ... ce qui vous permet d'avoir une arborescence correspondant à ce qui est demandé, sans pour autant perdre votre travail (ou le rendre plus difficile d'accès).