Virtualisation légère – TP n° 4

Linux Internals partie 2

Pierre-Olivier *nemunaire* Mercier

Mercredi 9 novembre 2022

Abstract

Le but de ce second TP sur les mécanismes internes du noyau va nous permettre d'utiliser les commandes et les appels système relatifs aux *namespaces* ainsi que d'appréhender la complexité des systèmes de fichiers.

Les exercices de ce cours sont à rendre au plus tard le mardi 15 novembre 2022 à 23 h 42. Consultez les sections matérialisées par un bandeau jaunes et un engrenage pour plus d'informations sur les éléments à rendre.

Sommaire

1 Des particularités de `mount`

Petite parenthèse avant de parler des *namespaces*, car nous avons encore besoin d'appréhender un certain nombre de concepts relatifs aux montages de systèmes de fichiers.

1.1 Les points de montage

Au premier abord, les points de montage dans l'arborescence d'un système de fichiers n'ont pas l'air d'être remplis de notions complexes : un répertoire peut être le point d'entrée d'un montage vers la partition d'un disque physique… ou d'une partition virtuelle, comme nous l'avons vu précédemment.

Mais avez-vous déjà essayé de monter la même partition d'un disque physique à deux endroits différents de votre arborescence ?

Si pour plein de raisons on pouvait se dire que cela ne devrait pas être autorisé, ce problème s'avère être à la base de beaucoup de fonctionnalités intéressantes. Le noyau va finalement décorréler les notions de montage, d'accès et d'accroche dans l'arborescence : et par exemple, une partition ne sera plus forcément démontée après un appel à `umount(2)`, mais le sera seulement lorsque cette partition n'aura plus d'accroches dans aucune arborescence.

La commande `findmnt(1)`, des `util-linux` nous permet d'avoir une vision arborescente des points de montage en cours d'utilisation.

```
TARGET                          SOURCE      FSTYPE     OPTIONS
/                               /dev/sda1   ext4       rw,data=ordered,...
  /proc                         proc        proc       rw,nosuid,nodev,...
  /sys                          sysfs       sysfs      rw,nosuid,nodev,...
    /sys/kernel/security        securityfs  securityfs rw,nosuid,nodev,...
    /sys/firmware/efi/efivars   efivarfs    efivarfs   ro,relatime
    /sys/fs/cgroup              cgroup_root tmpfs      rw,nosuid,...
      /sys/fs/cgroup/unified    none        cgroup2    rw,nsdelegate,...
      /sys/fs/cgroup/cpuset     cpuset      cgroup     rw,nosuid,cpuset,...
      /sys/fs/cgroup/cpu        cpu         cgroup     rw,nosuid,cpu,...
      /sys/fs/cgroup/cpuacct    cpuacct     cgroup     rw,nosuid,cpuacct,...
      /sys/fs/cgroup/blkio      blkio       cgroup     rw,nosuid,blkio,...
      /sys/fs/cgroup/memory     memory      cgroup     rw,nosuid,memory,...
      /sys/fs/cgroup/devices    devices     cgroup     rw,nosuid,devices,...
      /sys/fs/cgroup/freezer    freezer     cgroup     rw,nosuid,freezer,...
      /sys/fs/cgroup/net_cls    net_cls     cgroup     rw,nosuid,net_cls,...
      /sys/fs/cgroup/perf_event perf_event  cgroup     rw,nosuid,p_event,...
      /sys/fs/cgroup/net_prio   net_prio    cgroup     rw,nosuid,net_pri,...
      /sys/fs/cgroup/pids       pids        cgroup     rw,nosuid,pids,...
  /dev                          devtmpfs    devtmpfs   rw,nosuid,size=...
    /dev/pts                    devpts      devpts     rw,nosuid,gid=5,...
    /dev/shm                    tmpfs       tmpfs      rw
    /dev/mqueue                 mqueue      mqueue     rw,nosuid,nodev,...
  /home                         /dev/sda3   ext4       rw,nosuid,nodev,...
  /run                          tmpfs       tmpfs      rw,mode=755,...
  /tmp                          tmpfs       tmpfs      rw,nosuid,nodev,...
```

1.2 `bind` – montage miroir

Lorsque l'on souhaite monter à un deuxième endroit (ou plus) une partition, on utilise le *bind mount* :

```
mount --bind olddir newdir
```

Lorsque l'on souhaite `chroot` dans un système complet (par exemple lorsqu'on l'installe ou qu'on le répare via un *live CD*), il est nécessaire de dupliquer certains points de montage, tels que `/dev`, `/proc` et `/sys`.

Sans monter ces partitions, vous ne serez pas en mesure d'utiliser le système dans son intégralité : vous ne pourrez pas monter les partitions indiquées par le `/etc/fstab`, vous ne pourrez pas utiliser `top` ou `ps`, `sysctl` ne pourra pas accorder les paramètres du noyau, …

Pour que tout cela fonctionne, nous aurons besoin, au préalable, d'exécuter les commandes suivantes :

```
cd newroot
mount --bind /dev dev
mount --bind /proc proc
mount --bind /sys sys
```

En se `chroot`ant à nouveau dans cette nouvelle racine, tous nos outils fonctionneront comme prévu.

Tous ? … en fait non. Si l'on jette un œil à `findmnt(1)`, nous constatons par exemple que `/sys/fs/cgroup` dans notre nouvelle racine est vide, alors que celui de notre machine hôte contient bien les répertoires de nos *cgroups*.

`--bind` va se contenter d'attacher le système de fichiers (ou au moins une partie de celui-ci) à un autre endroit, sans se préoccuper des points de montages sous-jacents. Pour effectuer cette action récursivement, et donc monter au nouvel emplacement le système de fichier ainsi que tous les points d'accroche qu'il contient, il faut utiliser `--rbind`. Il serait donc plus correct de lancer :

```
cd newroot
mount --rbind /dev dev
mount -t proc none proc
mount --rbind /sys sys
```

1.3 Les types de propagation des points de montage

On distingue quatre variétés de propagation des montages pour un sous-arbre : partagé, esclave, privé et non-attachable.

Chacun va agir sur la manière dont seront propagées les nouvelles accroches au sein d'un système de fichiers attaché à plusieurs endroits.

1.3.1 partagé – shared mount

Dans un montage partagé, une nouvelle accroche sera propagée parmi tous les systèmes de fichiers de ce partage (on parle de *peer group*). Voyons avec un exemple :

```
# Création de notre répertoire de travail
mkdir /mnt/test-shared

# On s'assure que le dossier que l'on va utiliser pour nos tests utilise bien la
    politique shared
```

```
mount --make-shared /tmp

# Duplication de l'accroche, sans s'occuper des éventuels sous-accroches
mount --bind /tmp /mnt/test-shared
```

Si l'on attache un nouveau point de montage dans `/tmp` ou dans `/mnt/test-shared`, avec la politique `shared`, l'accroche sera propagée :

```
mkdir /mnt/test-shared/toto
mount -t tmpfs none /mnt/test-shared/toto
```

Un coup de `findmnt` nous montre l'existence de deux nouveaux points de montage. À `/mnt/test-shared/toto`, mais également à `/tmp/toto`.

1.3.2 esclave – slave mount

De la même manière que lorsque la propagation est partagée, cette politique propagera, mais seulement dans un sens. Le point de montage déclaré comme esclave ne propagera pas ses nouveaux points de montage à son *maître*.

```
# Suite de l'exemple précédent
cd /mnt/test-slave

# Duplication de l'accroche, sans s'occuper des éventuels sous-accroches
mount --bind /mnt/test-shared /mnt/test-slave

# On rend notre dossier esclave
mount --make-slave /mnt/test-slave
```

Si l'on effectue un montage dans `/mnt/test-shared` :

```
mkdir /mnt/test-shared/foo
mount -t tmpfs none /mnt/test-shared/foo
```

Le point de montage apparaît bien sous `/mnt/test-slave/foo`.

Par contre :

```
mkdir /mnt/test-slave/bar
mount -t tmpfs none /mnt/test-slave/bar
```

Le nouveau point de montage n'est pas propagé dans `/mnt/test-shared/bar`.

1.3.3 privé – private mount

C'est le mode le plus simple : ici les points de montage ne sont tout simplement pas propagés.

Pour forcer un point d'accroche à ne pas propager et à ne pas recevoir de propagation, on utilise l'option suivante :

```
mount --make-private mountpoint
```

1.3.4 non-attachable – unbindable mount

Ce mode interdira toute tentative d'attache à un autre endroit.

```
mount --make-unbindable /mnt/test-slave
```

Il ne sera pas possible de faire :

```
mkdir /mnt/test-unbindable
mount --bind /mnt/test-slave /mnt/test-unbindable
```

1.3.5 Propagation récursive

Les options que nous venons de voir s'appliquent sur un point de montage. Il existe les mêmes options pour les appliquer en cascade sur les points d'attache contenus dans leur sous-arbre :

```
mount --make-rshared mountpoint
mount --make-rslave mountpoint
mount --make-rprivate mountpoint
mount --make-runbindable mountpoint
```

1.4 Montage miroir de dossiers et de fichiers

Il n'est pas nécessaire que le point d'accroche que l'on cherche à dupliquer pointe sur un point de montage (c'est-à-dire, dans la plupart des cas : une partition ou un système de fichiers virtuel). Il peut parfaitement pointer sur un dossier, et même sur un simple fichier, à la manière d'un *hardlink*, mais que l'on pourrait faire entre plusieurs partitions et qui ne persisterait pas au redémarrage (le *hardlink* persiste au redémarrage, mais doit se faire au sein d'une même partition).

Nous verrons dans la partie *namespace* réseau une utilisation d'attache sur un fichier.

1.5 Déplacer un point de montage

À tout moment, il est possible de réorganiser les points de montage, en les déplaçant. Comme cela se fait sans démonter de partition, il est possible de le faire même si un fichier est en cours d'utilisation. Il faut cependant veiller à ce que les programmes susceptibles d'aller chercher un fichier à l'ancien emplacement soient prévenus du changement.

Pour déplacer un point de montage, on utilise l'option `--move` de `mount(8)` :

```
mount --move olddir newdir
```

Par exemple :

```
mount --move /dev /newroot/dev
```

Quand a-t-on besoin de déplacer un point de montage ?
Cette possibilité s'emploie notamment lorsque l'on souhaite changer la racine de notre système de fichiers : par exemple pour passer de l'*initramfs* au système démarré, ou encore de notre système hôte au système d'un conteneur, …

Aller plus loin

Voici quelques articles qui valent le détour, en lien avec les points de montage :

— Shared subtree (https://lwn.net/Articles/159077) et la documentation du noyau associée (https://kernel.org/doc/Documentation/filesystems/sharedsubtree.txt) ;
— Mount namespaces and shared subtrees : https://lwn.net/Articles/689856 ;
— Mount namespaces, mount propagation, and unbindable mounts : https://lwn.net/Articles/690679.

2 Les espaces de noms – *namespaces*

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.

Ces fonctionnalités sont très utiles pour éviter les dénis de service, mais nos processus ne sont pas particulièrement isolés du reste du système. On aimerait maintenant que nos processus n'aient pas accès à l'ensemble des fichiers, ne puissent pas interagir avec les autres processus, avoir leur propre pile réseau, … Voyons maintenant les *namespaces* qui vont nous permettre de faire cela.

2.1 Présentation des namespaces

Les espaces de noms du noyau, que l'on appelle *namespaces*, permettent de dupliquer certaines structures, habituellement considérées uniques pour le noyau, dans le but qu'un groupe de processus soit isolé d'autres processus, sur certains aspects de l'environnement dans lequel il s'exécute.

On en dénombre huit (le dernier ayant été ajouté dans Linux 5.6) : `cgroup`, `IPC`, `network`, `mount`, `PID`, `time`, `user` et `UTS`.

La notion d'espace de noms est relativement nouvelle et a été intégrée progressivement au sein du noyau Linux. Aussi, toutes les structures ne sont pas encore *containerisables* : le document fondateur[1] parle ainsi d'isoler les journaux d'événements ou encore les modules de sécurité (LSM, tels que AppArmor, SELinux, Yama, …).

Commençons par passer en revue rapidement les différents *namespaces*.

L'espace de noms `mount` Depuis Linux 2.4.19.

Cet espace de noms dissocie la liste des points de montage.

Chaque processus appartenant à un *namespace mount* différent peut monter, démonter et réorganiser à sa guise les points de montage, sans que cela n'ait d'impact sur les processus hors de cet espace de noms. Une partition ne sera donc pas nécessairement démontée après un appel à `umount(2)`, elle le sera lorsqu'elle aura effectivement été démontée de chaque *namespace mount* dans lequel elle était montée.

Il s'agit d'une version améliorée de nos bons vieux `chroot`, puisqu'il n'est plus possible de s'en échapper en remontant l'arborescence.

L'espace de noms `UTS` Depuis Linux 2.6.19.

Cet espace de noms isole le nom de machine et son domaine NIS.

L'espace de noms `IPC` Depuis Linux 2.6.19.

Cet espace de noms isole les objets IPC et les files de messages POSIX.

Une fois le *namespace* attaché à un processus, il ne peut alors plus parler qu'avec les autres processus de son espace de noms (lorsque ceux-ci passent par l'API IPC du noyau).

[1] https://www.kernel.org/doc/ols/2006/ols2006v1-pages-101-112.pdf

L'espace de noms `PID` Depuis Linux 2.6.24.

Cet espace de noms isole la liste des processus et virtualise leurs numéros.

Une fois dans un espace, le processus ne voit que le sous-arbre de processus également attachés à son espace. Il s'agit d'un sous-ensemble de l'arbre global de PID : les processus de tous les PID *namespaces* apparaissent donc dans l'arbre initial.

Pour chaque nouvel espace de noms de processus, une nouvelle numérotation est initiée. Ainsi, le premier processus de cet espace porte le numéro 1 et aura les mêmes propriétés que le processus `init` usuel ; entre autres, si un processus est rendu orphelin dans ce *namespace*, il devient un fils de ce processus, et non un fils de l'`init` de l'arbre global.

L'espace de nom `network` Depuis Linux 2.6.29.

Cet espace de noms fournit une isolation pour toutes les ressources associées aux réseaux : les interfaces, les piles protocolaires IPv4 et IPv6, les tables de routage, règles pare-feu, ports numérotés, etc.

Une interface réseau (`eth0`, `wlan0`, ...) ne peut se trouver que dans un seul espace de noms à la fois. Il est par contre possible de les déplacer.

Lorsque le *namespace* est libéré (généralement lorsque le dernier processus attaché à cet espace de noms se termine), les interfaces qui le composent sont ramenées dans l'espace initial/racine (et non pas dans l'espace parent, en cas d'imbrication).

L'espace de noms `user` Depuis Linux 3.8.

Cet espace de noms isole la liste des utilisateurs, des groupes, leurs identifiants, les *capabilities*, la racine et le trousseau de clefs du noyau.

La principale caractéristique est que les identifiants d'utilisateur et de groupe pour un processus peuvent être différents entre l'intérieur et l'extérieur de l'espace de noms. Il est donc possible, alors que l'on est un simple utilisateur à l'extérieur du *namespace*, d'avoir l'UID 0 dans le conteneur.

L'espace de noms `cgroup` Depuis Linux 4.6.

Cet espace de noms filtre l'arborescence des *Control Group* en changeant la racine vue par le processus. Au sein d'un *namespace* `cgroup`, la racine vue correspond en fait à un sous-groupe de l'arborescence globale.

Ainsi, un processus dans un *CGroup namespace* ne peut pas voir le contenu des sous-groupes parents (pouvant laisser fuiter des informations sur le reste du système). Cela peut également permettre de faciliter la migration de processus (d'un système à un autre) : l'arborescence des *cgroups* n'a alors plus d'importance car le processus ne voit que son groupe.

L'espace de noms `time` Depuis Linux 5.6.

Avec cet espace de noms, il n'est pas possible de virtualiser l'heure d'un de nos conteneurs (on peut seulement changer le fuseau horaire, puisqu'ils sont gérés par la `libc`). Les horloges virtualisées avec ce *namespace* sont les compteurs `CLOCK_MONOTONIC` et `CLOCK_BOOTTIME`.

Lorsque l'on souhaite mesurer un écoulement de temps, la méthode naïve consiste à enregistrer l'heure au départ de notre opération, puis à faire une soustraction avec l'heure de fin. Cette technique fonctionne bien, à partir du moment où l'on est sûr que l'horloge ne remontera pas dans le temps, parce qu'elle se synchronise ou que le changement d'heure été/hiver intervient, ... Pour palier ces situations imprévisibles, le noyau expose une horloge dite monotone (`CLOCK_MONOTONIC`) : cette horloge démarre à un entier abstrait

et s'incrèmente chaque seconde qui passe, sans jamais sauter de secondes, ni revenir en arrière. C'est une horloge fiable pour calculer des intervalles de temps.

De la même manière `CLOCK_BOOTTIME` mesure le temps qui s'écoule, mais prend en compte les moments où la machine est en veille (alors que `CLOCK_MONOTONIC` ne compte que les moments où la machine est en éveil).

Étant donné l'usage de ces deux horloges, en cas de migration d'un processus d'une machine à une autre, il convient de recopier l'état des horloges monotones qu'il utilise, afin que ses calculs ne soient pas chamboulés.

2.1.1 Prérequis

2.1.1.1 Noyau Linux Pour pouvoir suivre les exercices ci-après, vous devez disposer d'un noyau Linux, idéalement dans sa version 5.6 ou mieux. Il doit de plus être compilé avec les options suivantes (lorsqu'elles sont disponibles pour votre version) :

```
General setup  --->
    [*] Control Group support  --->
    -*- Namespaces support
      [*]   UTS namespace
      [*]   TIME namespace
      [*]   IPC namespace
      [*]   User namespace
      [*]   PID Namespaces
      [*]   Network namespace
[*] Networking support  --->
    Networking options  --->
      <M> 802.1d Ethernet Bridging
Device Drivers  --->
    [*] Network device support  --->
      <M>   MAC-VLAN support
      <M>   Virtual ethernet pair device
```

Les variables de configuration correspondantes sont :

```
CONFIG_CGROUPS=y

CONFIG_NAMESPACES=y
CONFIG_UTS_NS=y
CONFIG_TIME_NS=y
CONFIG_IPC_NS=y
CONFIG_USER_NS=y
CONFIG_PID_NS=y
CONFIG_NET_NS=y

CONFIG_NET=y
CONFIG_BRIDGE=m

CONFIG_NETDEVICES=y
CONFIG_MACVLAN=m
CONFIG_VETH=m
```

Référez-vous, si besoin, à la précédente configuration que l'on a faite pour la marche à suivre.

2.1.1.2 Paquets

Nous allons utiliser des programmes issus des `util-linux`, de `procps-ng` ainsi que ceux de la `libcap`.

Sous Debian et ses dérivés, ces paquets sont respectivement :

- `util-linux`
- `procps`
- `libcap2-bin`

Sous ArchLinux et ses dérivés, ces paquets sont respectivement :

- `util-linux`
- `procps-ng`
- `libcap`

2.1.1.3 À propos de la sécurité de l'espace de noms user

La sécurité du *namespace* `user` a souvent été remise en cause par le passé, on lui attribue de nombreuses vulnérabilités. Vous devriez notamment consulter à ce sujet :

- Security Implications of User Namespaces : https://blog.araj.me/security-implications-of-user-namespaces/ ;
- Anatomy of a user namespaces vulnerability : https://lwn.net/Articles/543273/ ;
- http://marc.info/?l=linux-kernel&m=135543612731939&w=2 ;
- http://marc.info/?l=linux-kernel&m=135545831607095&w=2.

De nombreux projets ont choisi de ne pas autoriser l'utilisation de cet espace de noms sans disposer de certaines *capabilities*[2].

De nombreuses distributions ont choisi d'utiliser un paramètre du noyau pour adapter le comportement.

?

Debian et ses dérivées Si vous utilisez Debian ou l'un de ses dérivés, vous devrez autoriser explicitement cette utilisation non-privilégiée :

```
42sh# sysctl -w kernel.unprivileged_userns_clone=1
```

Grsecurity D'autres patchs, tels que *grsecurity* ont fait le choix de désactiver cette possibilité sans laisser d'option pour la réactiver éventuellement à l'exécution. Pour avoir un comportement identique à celui de Debian, vous pouvez appliquer ce patch[a] sur vos sources incluant le patch de *grsecurity*.

[a] https://nemunai.re/post/user-ns-for-grsecurity/

2.2 Explorons les namespaces

Maintenant que nous avons quelques notions de base sur les espaces de nom, voyons quelles surprises nous réserve notre système…

[2] Sont nécessaires, conjointement : `CAP_SYS_ADMIN`, `CAP_SETUID` et `CAP_SETGID`.

2.2.1 Voir les namespaces de notre système

La première commande que l'on va utiliser est `lsns(8)`, afin d'afficher tous les *namespace*s actuels de notre machine.

```
42sh# lsns
        NS TYPE   NPROCS PID USER             COMMAND
4026531834 time   238      1 root             /sbin/init
4026531835 cgroup 238      1 root             /sbin/init
4026531836 pid    239      1 root             /sbin/init
4026531837 user   227      1 root             /sbin/init
4026531838 uts    231      1 root             /sbin/init
4026531839 ipc    228      1 root             /sbin/init
4026531840 net    228      1 root             /sbin/init
4026531841 mnt    223      1 root             /sbin/init
4026532230 uts      1    227 root               /usr/lib/systemd/systemd-udevd
4026532483 mnt      4    366 root               /usr/lib/systemd/systemd-userdbd
4026532485 mnt      1    363 systemd-resolve    /usr/lib/systemd/systemd-resolved
4026532486 mnt      1    364 systemd-timesync   /usr/lib/systemd/systemd-timesyncd
4026532491 mnt      1    381 root               /usr/lib/systemd/systemd-logind
4026532569 mnt      1    428 systemd-network    /usr/lib/systemd/systemd-networkd
4026531862 mnt      1     33 root             kdevtmpfs
[...]
```

Cette commande nous dévoile déjà de nombreuses choses :

— Chaque processus se trouve dans un *namespace* de chaque type : le noyau n'a pas de notion de « processus hôte » (sans *namespace*) et « processus contenerisé » (dans un *namespace*). Le processus initial de la machine se retrouve donc dans des espaces de nom, tout comme les processus d'un conteneur.
— On aperçoit un genre de hiérarchie dans certain cas.
— La première colonne nous renseigne sur l'identifiant du *namespace*.
— `kdevtmpfs` : un *thread* du noyau, s'exécute dans un espace de nom `mnt` dédié.

Vous verrez surement davantage de processus si vous exécutez cette commande sur une machine que vous utilisez : chaque conteneur y sera bien entendu listé, quelque soit la technologie sous-jacente (Docker, podman, CRI-O, snapd, …), mais chose plus étonnante, Chrome et Firefox tirent également parti des espaces de nom pour de la défense en profondeur.

2.2.2 Voir les namespaces d'un processus

Chaque processus lancé est donc rattaché à une liste d'espaces de nom, y compris s'il est issu du système de base (« l'hôte »).

Nous pouvons dès lors consulter le dossier `/proc/<PID>/ns/` de chaque processus, pour consulter les différents espaces de nom de nos processus.

Tous les processus ont la même liste de fichiers. Ils sont tous liés à un espace de noms par *namespace* utilisable avec la version noyau dont on dispose. D'une machine à l'autre, d'une version du noyau à l'autre, il est normal d'avoir une liste de *namespace*s différente, mais d'un processus à l'autre sur un même noyau, nous aurons les mêmes espaces de nom disponibles, donc les mêmes fichiers.

Ces fichiers sont en fait des liens symboliques un peu particuliers, car ils ne pointent pas vers une destination “valide” :

```
42sh$ ls -l /proc/self/ns
lrwxrwxrwx 1 nemunaire 1 oct. 23:42 cgroup -> 'cgroup:[4026531835]'
lrwxrwxrwx 1 nemunaire 1 oct. 23:42 ipc    -> 'ipc:[4026531839]'
lrwxrwxrwx 1 nemunaire 1 oct. 23:42 mnt    -> 'mnt:[4026531840]'
lrwxrwxrwx 1 nemunaire 1 oct. 23:42 net    -> 'net:[4026532008]'
lrwxrwxrwx 1 nemunaire 1 oct. 23:42 pid    -> 'pid:[4026531836]'
lrwxrwxrwx 1 nemunaire 1 oct. 23:42 pid_for_children -> 'pid:[4026531836]'
lrwxrwxrwx 1 nemunaire 1 oct. 23:42 time   -> 'time:[4026531834]'
lrwxrwxrwx 1 nemunaire 1 oct. 23:42 time_for_children -> 'time:[4026531834]'
lrwxrwxrwx 1 nemunaire 1 oct. 23:42 user   -> 'user:[4026531837]'
lrwxrwxrwx 1 nemunaire 1 oct. 23:42 uts    -> 'uts:[4026531838]'
```

Les liens référencent une structure du noyau résidant en mémoire. Les numéros entre crochets sont les *inodes* du système de fichiers `nsfs`, que l'on a pu voir dans la première colonne de `lsns(8)`. Ce sont les identifiants des espaces de nom.

Ces *inodes* seront les mêmes pour deux processus qui partagent le même espace de noms : la structure pointée sera identique. Elle sera par contre différente si l'espace de nom est différent, l'*inode* sera donc différent.

? **`*_for_children`** Vous avez peut-être remarqué des fichiers `*_for_children` dans le dossier `ns` de vos processus. Les espaces de noms *PID* et *Time*, lorsqu'on les change pour un processus, ne s'appliquent pas directement au processus en cours d'exécution, la dissociation de *namespace* ne pourra se faire que pour les processus/threads fils.
`pid_for_children` et `time_for_children` représentent donc les *namespace*s qui seront attribués aux processus fils lancés par un `clone(2)` ou un `fork(2)`.
En attendant notre processus courant conserve ses espaces de nom `pid` et `time`.

2.3 Utiliser les namespaces

2.3.1 S'isoler dans un nouveau namespace

Si l'on voit l'isolation procurée par les *namespace*s comme des machines virtuelles, on peut se dire qu'il suffit d'exécuter un appel système pour arriver dans un conteneur bien isolé. Cependant, le choix fait par les développeurs de Linux a été de laisser le choix des espaces de noms dont on veut se dissocier.

L'intérêt principal de cette approche, exploitée bien après la mise en avant du concept, est que l'utilisation des *namespace*s ne se limite pas seulement à des machines virtuelles légères. On retrouve ainsi dans Google Chrome et Firefox de nombreuses utilisations des *namespace*s dans le simple but d'accroître la sécurité de leur navigateur. Ainsi, les *threads* de rendu n'ont pas accès au réseau et sont cloisonnés de manière transparente pour l'utilisateur.

Nous allons voir dans cette partie plusieurs méthodes pour utiliser ces espaces de noms.

2.3.1.1 Dans son shell

De la même manière que l'on peut utiliser l'appel système `chroot(2)` depuis un shell via la commande `chroot(1)`, la commande `unshare(1)` permet de faire le nécessaire pour lancer l'appel système `unshare(2)`, puis, tout comme `chroot(1)`, exécuter le programme passé en paramètre.

En fonction des options qui lui sont passées, `unshare(1)` va créer le/les nouveaux *namespaces* et placer le processus dedans.

Par exemple, nous pouvons modifier sans crainte le nom de notre machine, si nous sommes passés dans un autre *namespace* `UTS` :

```
42sh# hostname --fqdn
koala.zoo.paris
42sh# sudo unshare -u /bin/bash
  bash# hostname --fqdn
  koala.zoo.paris
  bash# hostname lynx.zoo.paris
  bash# hostname --fqdn
  lynx.zoo.paris
  bash# exit
42sh# hostname --fqdn
koala.zoo.paris
```

Nous avons pu ici modifier le nom de la machine, sans que cela n'affecte notre machine hôte.

2.3.1.2 Les appels système

L'appel système par excellence pour contrôler l'isolation d'un nouveau processus est `clone(2)`.

Ce *syscall*, propre à Linux, crée habituellement un nouveau processus enfant de notre processus courant (mais il peut aussi créer des *threads*, on va voir qu'il fait beaucoup de choses), comme `fork(2)` (qui lui est un appel système POSIX). Mais il prend en plus de nombreux paramètres. L'isolement ou non du processus se fait en fonction des *flags* qui sont passés. On retrouve donc :

- `CLONE_NEWNS`,
- `CLONE_NEWUTS`,
- `CLONE_NEWIPC`,
- `CLONE_NEWPID`,
- `CLONE_NEWNET`,
- `CLONE_NEWUSER`,
- `CLONE_NEWCGROUP`,
- `CLONE_NEWTIME`.

> Le nom du *flag* `CLONE_NEWNS` est historique et assez peu explicite contrairement aux autres : il désigne en fait l'espace de nom `mount`.
> Au départ, les *namespaces* ont étés pensés pour former un tout : une couche d'isolation complète pour les processus. Mais lors des développements suivants, il s'est avéré pratique de pouvoir choisir finement de quels aspects on souhaitait se dissocier.
> C'est ainsi que pour chaque nouveau *namespace*, un nouveau *flag* est introduit.

On peut bien entendu cumuler un ou plusieurs de ces *flags*, et les combiner avec d'autres attendus par `clone(2)`.

Ces mêmes *flags* sont utilisés lors des appels à `unshare(2)` ou `setns(2)`, que nous verrons plus tard.

>_

Pour créer un nouveau processus qui sera à la fois dans un nouvel espace de noms réseau et dans un nouveau *namespace* cgroup, on écrirait un code C semblable à :

```c
#include <sched.h>

#define STACKSIZE (1024 * 1024)
static char child_stack[STACKSIZE];

int clone_flags = CLONE_CGROUP | CLONE_NEWNET | SIGCHLD;

pid_t pid = clone(do_execvp,          // First function executed by child
                  child_stack + STACKSIZE, // Assume stack grows downward
                  clone_flags,            // clone specials flags
                  args);                  // Arguments to pass to
                                          //    do_execvp
```

Dans cet exemple, le processus fils créé disposera d'un nouvel espace de noms pour les *CGroups* et disposera d'une nouvelle pile réseau.

?

Quel est le rôle du flag `SIGCHLD` ?

Lorsque l'on crée un nouveau processus, on ajoute l'option `SIGCHLD` afin d'être notifié par signal lorsque notre processus fil a terminé son exécution. Cela permet d'être réveillé de notre `wait(2)`.

L'appel système `clone(2)` va donc créer un nouveau processus, ou un nouveau *thread*, mais parfois on souhaite juste isoler notre processus actuel.

2.3.1.2.1 `unshare`

Lorsque l'on souhaite faire entrer notre processus courant ou notre *thread* dans un nouvel espace de noms, on peut utiliser l'appel système `unshare(2)`. Celui-ci prend uniquement en argument une liste de *flags* des *namespaces* dont on souhaite se dissocier.

!

Le comportement de `unshare(2)` (ou `unshare(3)`) avec les *namespaces PID* et *Time* n'est pas celui que l'on peut attendre !

En effet, après avoir appelé `unshare`, le processus reste tout de même dans son *namespace Time* ou *PID* d'origine, seuls ses enfants (après un appel à `fork(2)` ou `clone(2)` par exemple) seront dans le nouveau *namespace*.
Cela poserait trop de problème de faire changer le PID d'un processus en cours d'exécution, de même qu'il serait impensable que la `CLOCK_MONOTONIC` puisse faire un saut en avant ou en arrière. Alors le choix qui a été fait est que seuls les fils créés après l'appel à `unshare(2)` seront concrètement dans le nouveau *namespace*. C'est dans cette situation que `pid` et `pid_for_children` peuvent être différents dans le dossier `/proc/<PID>/ns`.
On ne remarque pas cette bizarrerie avec `clone(2)`, car il crée déjà un nouveau processus, son PID et sa `CLOCK_MONOTONIC` sont directement à la bonne valeur dès l'exécution de la fonction `fn`.

Nous avons vu comment créer et nous dissocier d'un espace de nom. Maintenant voyons comment en rejoindre un déjà existant.

2.3.2 Rejoindre un namespace

Rejoindre un espace de noms se fait en utilisant l'appel système `setns(2)`, ou la commande `nsenter(1)`. Il est nécessaire de donner en argument respectivement un *file descriptor* ou le chemin vers le fichier, lien symbolique, représentant l'espace de nom (dans `/proc/<PID>/ns/...`).

Une particularité de ces fichiers, que l'on ne peut pas afficher (leurs liens ne pointent pas sur des fichiers que l'on peut atteindre), c'est que l'on peut les ouvrir avec `open(2)` pour obtenir un *file descriptor* que l'on pourra passer à `setns(2)`.

Pour les commandes *shell*, il convient de donner en argument le chemin vers le lien symbolique : la commande se chargera d'`open(2)` le fichier pour obtenir le *file descriptor* nécessaire.

Exemple C
Voici un exemple de code C utilisant `setns(2)` :

```c
#define _GNU_SOURCE
#include <fcntl.h>
#include <sched.h>
#include <stdlib.h>

// ./a.out /proc/PID/ns/FILE cmd args...

int
main(int argc, char *argv[])
{
    int fd = open(argv[1], O_RDONLY);
    if (fd == -1)
    {
      perror("open");
      return EXIT_FAILURE;
    }

    if (setns(fd, 0) == -1)
    {
      perror("setns");
      return EXIT_FAILURE;
    }

    execvp(argv[2], &argv[2]);

    perror("execve");
    return EXIT_FAILURE;
}
```

Ce programme prend au minimum deux arguments : - le chemin d'un fichier d'espace de nom que l'on souhaite rejoindre (le chemin vers le lien symbolique donc) ; - le programme (et ses arguments) que l'on souhaite souhaite exécuter une fois que l'on a rejoint l'espace de noms ciblé.

Dans un premier temps, on ouvre le fichier passé en paramètre afin d'obtenir un *file descriptor* de la structure du noyau.

On passe ensuite ce *file descriptor* en argument de l'appel système `setns(2)`. Et enfin on exécute la commande données dans les derniers paramètres.

Qu'attend le deuxième argument de `setns(2)` ?

Il s'agit d'une contrainte sur le type d'espace de nom que l'on souhaite rejoindre, dans le cas où on ne souhaite pas rejoindre n'importe quel espace de nom.
Une fois encore, on utilisera les mêmes options `CLONE_NEW*` lorsque l'on attendra un type particulier d'espace de nom, ou 0 pour autoriser tous les types.

Peut-on connaître le type de namespace à partir du file descriptor ?

Il est possible de récupérer le type d'espace de nom en passant notre *file descriptor* à `ioctl(2)`, avec le *flag* `NS_GET_NSTYPE` :

```
nstype = ioctl(fd, NS_GET_NSTYPE);
if (nstype & CLONE_NEWUTS != 0) {
    ... // This is a file descriptor to an UTS namespace
}
```

Exemple shell

Dans un shell, nous utiliserons la commande `nsenter(1)` :

```
42sh# unshare --uts /bin/bash
  inutsns# echo $$
  42
  inutsns# hostname jument
  # Keep this shell active to perform nexts steps, in another shell.

42sh# hostname
chaton
42sh# nsenter --uts=/proc/42/ns/uts /bin/bash
  inutsns# hostname
  jument
```

Avec `nsenter(1)`, il est possible de cibler un processus particulier, sans aller nécessairement faire référence aux fichiers de `/proc`, en utilisant l'option `--target` :

```
42sh# unshare --uts /bin/bash
  inutsns# echo $$
  42
  inutsns# hostname jument
  # Keep this shell active to perform nexts steps, in another shell.
```

```
42sh# hostname
chaton
42sh# nsenter --target 42 --uts /bin/bash
  inutsns# hostname
  jument
```

Les options supplémentaires `--uts`, `--mount`, `--ipc`, `--pid`, … ne prennent alors pas d'argument, mais désignent les espaces de noms du processus ciblé que l'on souhaite rejoindre.

D'une manière similaire à notre dernier exemple, depuis Linux 5.8, `setns(2)` peut utiliser comme premier argument, un *file descriptor* pointant un processus (`pidfd`).

Eh oui, outre l'obtention d'un *file descriptor* sur un lien symbolique étrange d'une structure du noyau, il est possible d'en obtenir un sur un processus :

— soit en réalisant un `open(2)` d'un dossier `/proc/<PID>` ;
— soit en utilisant l'appel système `pidfd_open(2)`, en précisant l'identifiant du processus dont on souhaite obtenir le *file descriptor* ;
— soit en retour d'un `clone(2)` avec l'option `CLONE_PIDFD`.

```
int fd = pidfd_open(42, 0);
setns(fd, CLONE_NEWUSER | CLONE_NEWNET | CLONE_NEWUTS);
```

À travers cet exemple, on cherche à récupérer un *file descriptor* pour le processus 42. On le passe ensuite à `setns(2)` en précisant que l'on ne souhaite rejoindre que les espaces de noms Utilisateurs, Réseau et UTS (nom de la machine).

cmps.sh

2.3.3 Comparaison de namespace – `cmpns.sh`

Les *namespaces* d'un programme sont exposés sous forme de liens symboliques dans le répertoire `/proc/<PID>/ns/`.

Deux programmes qui partagent un même *namespace* auront un lien vers le même *inode*.

Écrivons un script, `cmpns`, permettant de déterminer si deux programmes s'exécutent dans les mêmes *namespaces*. On ignorera les *namespaces* `*_for_children`, car ils ne font pas partie du cycle d'exécution que l'on cherche à comparer.

En shell, vous aurez besoin de `grep(1)` et de `readlink(1)`.

```
42sh$ docker run -d influxdb
42sh$ ./cmpns $(pgrep influxd) $(pgrep init)
  - cgroup: differ
  - ipc: differ
  - mnt: differ
  - net: differ
  - pid: differ
  - time: same
  - user: same
  - uts: same
```

Exemples

```
42sh$ ./cmpns $(pgrep init) self
  - cgroup: same
  - ipc: same
  - mnt: same
  - net: same
  - pid: same
  - time: same
  - user: same
  - uts: same
```

Ici, `self` fait référence au processus actuellement exécuté (comme il existe un dossier `/proc/self/`, vous n'avez pas besoin de gérer de cas particulier pour ça !).

Et pourquoi pas :

```
42sh# unshare -m ./cmpns $$ self
  - cgroup: same
  - ipc: same
  - mnt: differ
  - net: same
  - pid: same
  - time: same
  - user: same
  - uts: same
```

2.3.4 Durée de vie d'un namespace

Le noyau tient à jour un compteur de références pour chaque *namespace*. Dès qu'une référence tombe à 0, la structure de l'espace de noms est automatiquement libérée, les points de montage sont démontés, les interfaces réseaux sont réattribuées à l'espace de noms initial, …

Ce compteur évolue selon plusieurs critères, et principalement selon le nombre de processus qui l'utilisent. C'est-à-dire que, la plupart du temps, le *namespace* est libéré lorsque le dernier processus s'exécutant dedans se termine.

Lorsque l'on a besoin de référencer un *namespace* (par exemple pour le faire persister après le dernier processus), on peut utiliser un `mount bind` :

```
42sh# touch /tmp/ns/myrefns
42sh# mount --bind /proc/<PID>/ns/mount /tmp/ns/myrefns
```

De cette manière, même si le lien initial n'existe plus (si le `<PID>` s'est terminé), `/tmp/ns/myrefns` pointera toujours au bon endroit.

Il est aussi tout à fait possible d'utiliser directement ce fichier pour obtenir un descripteur de fichier valide vers le *namespace* (pour passer à `setns(2)`).

Faire persister un namespace ?

?

Il n'est pas possible de faire persister un espace de noms d'un reboot à l'autre.

Même en étant attaché à un fichier du disque, il s'agit d'un pointeur vers une structure du noyau, qui ne persistera pas au redémarrage.

Aller plus loin

Je vous recommande la lecture du *man* `namespaces(7)` introduisant et énumérant les *namespaces*.

Pour tout connaître en détails, la série d'articles de Michael Kerrisk sur les *namespaces*[3] est excellente ! Auquel il faut ajouter l'article sur le plus récent `cgroup` *namespace*[4] et le petit dernier sur le *namespace* `time`[5].

[3] https://lwn.net/Articles/531114/
[4] https://lwn.net/Articles/621006/
[5] https://lwn.net/Articles/766089/

2.4 Le namespace `network`

Voyons maintenant plus en détail les différents espaces de nom, leurs caractéristiques et leurs usages ; en commençant par le *namespace* `network`.

2.4.1 Introduction

L'espace de noms `network`, comme son nom l'indique permet de virtualiser tout ce qui est en lien avec le réseau : les interfaces, les ports, les routes, les règles de filtrage, etc.

En entrant dans un nouvel espace de noms `network`, on se retrouve dans un environnement qui n'a plus qu'une interface de *loopback* :

```
42sh# unshare --net ip a
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN group default qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
```

Bien que portant le même nom que l'interface de *loopback* de notre environnement principal, il s'agit bien de deux interfaces isolées l'une de l'autre.

Afin d'amener du réseau à notre nouvel espace de nom, il va falloir lui attribuer des interface. En fait, nous allons pouvoir déplacer nos interfaces réseaux, dans le *namespace* vers lequel elle doit être accessible. Une interface donnée ne peut se trouver que dans un seul *namespace* à la fois.

Qui dit nouvelle pile réseau, dit également que les ports qui sont assignés dans l'espace principal, ne le sont plus dans le conteneur : il est donc possible de lancer un serveur web sans qu'il n'entre en conflit avec celui d'un autre espace de noms.

2.4.2 Premiers pas avec `ip netns`

La suite d'outils `iproute2` propose une interface simplifiée pour utiliser le *namespace* `network` : `ip netns`.

Nous pouvons tout d'abord créer un nouvel espace de noms :

```
42sh# ip netns add virli
```

La technique utilisée ici pour avoir des *namespaces* nommés est la même que celle que nous avons vue dans la première partie sur les *namespaces* : via un `mount --bind` dans le dossier `/var/run/netns/`. Cela permet de faire persister le namespace malgré le fait que plus aucun processus ne s'y exécute.
Nous pouvons créer artificiellement des entrées pour `ip netns` avec les quelques commandes suivantes :

```
# On affiche la liste des netns déjà créés
42sh# ip netns
virli

# On crée un fichier pour servir de réceptacle au bind mount
```

>_

```
42sh# touch /var/run/netns/foo

# On crée un nouveau namespace net, puis on bind tout de suite le
# fichier de namespace vers le fichier que l'on vient de créer
42sh# unshare --net \
    mount --bind /proc/self/ns/net /var/run/netns/foo

# Testons si cela a bien marché
42sh# ip netns
foo  virli
42sh# ip netns exec foo ip link
1: lo: <LOOPBACK> mut 65536 qdisc noop state DOWN mode DEFAULT group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
```

Les fichiers utilisés par `ip netns` ne sont donc rien de plus que des *bind-mount*. Ce qui explique qu'ils soient persistant même sans processus s'exécutant à l'intérieur.

Maintenant que notre *namespace* est créé, nous pouvons regarder s'il contient des interfaces :

```
42sh# ip netns exec virli ip link
1: lo: <LOOPBACK> mut 65536 qdisc noop state DOWN mode DEFAULT group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
```

Cette commande ne nous montre que l'interface de *loopback*, car nous n'avons pour l'instant pas encore attaché la moindre interface.

D'ailleurs, cette interface est rapportée comme étant désactivée, activons-la via la commande :

```
42sh# ip netns exec virli ip link set dev lo up
```

À ce stade, nous pouvons déjà commencer à lancer un `ping` sur cette interface :

```
42sh# ip netns exec virli ping 127.0.0.1
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.038 ms
...
```

2.4.3 Virtual Ethernet

Étant donné qu'une interface réseau ne peut être présente que dans un seul espace de noms à la fois, il n'est pas bien pratique d'imposer d'avoir une interface physique par conteneur, d'autant plus si l'on a plusieurs centaines de conteneurs à gérer.

Une technique couramment employée consiste à créer une interface virtuelle de type `veth` :

```
42sh# ip link add veth0 type veth peer name veth1
```

Une interface `veth` se comporte comme un tube bidirectionnel : tout ce qui entre d'un côté sort de l'autre et inversement. La commande précédente a donc créé deux interfaces `veth0` et `veth1` : les paquets envoyés sur `veth0` sont donc reçus par `veth1` et les paquets envoyés à `veth1` sont reçus par `veth0`.

Pour réaliser ces étapes dans un langage de programmation, nous allons passer par Netlink. Il s'agit d'une interface de communication entre le noyau et l'espace utilisateur. Il faut commencer par créer une `socket` pour avoir accès à l'API Netlink, nous pourrons ensuite envoyer des requêtes, comme celle nous permettant de créer notre interface `veth`.
Un message Netlink a une structure, alignée sur 4 octets, contenant un en-tête et des données. Le format de l'en-tête est décrit dans le RFC 3549[a] :

```
struct nlmsghdr {
    uint32_t nlmsg_len;    // Length of message including header
    uint16_t nlmsg_type;   // Type of message content
    uint16_t nlmsg_flags;  // Additional flags
    uint32_t nlmsg_seq;    // Sequence number
    uint32_t nlmsg_pid;    // Sender port ID
};
```

Parmi les fonctionnalités de Netlink, nous allons utiliser le module NIS (Network Interface Service)[^REF3549NIS]. Il spécifie le format par lequel doivent commencer les données liées à l'administration d'interfaces réseau.

```
struct ifinfomsg {
    uint8_t  ifi_family;   // AF_UNSPEC
    // uint8_t  Reserved
    uint16_t ifi_type;     // Device type
    int32_t  ifi_index;    // Interface index
    uint32_t ifi_flags;    // Device flags
    uint32_t ifi_change;   // change mask
};
```

Le module NIS a besoin que les données soient transmises sous forme d'*attributs Netlink*. Ces attributs fournissent un moyen de segmenter la charge utile en sous-sections. Un attribut a une taille et un type, en plus de sa charge utile.

```
struct rtattr {
    uint16_t rta_len;    // Length of option
    uint16_t rta_type;   // Type of option
    // Data follows
};
```

Le *payload* du message Netlink sera donc transmis comme une liste d'attributs (où chaque attribut peut à son tour avoir des attributs imbriqués).

```
struct rtattr {
  unsigned short  rta_len;
  unsigned short  rta_type;
};
```

En se basant sur du code d'`ip link`[b], on peut reconstituer la communication suivante :

```
#define MAX_PAYLOAD 1024

struct nlreq {
    struct nlmsghdr hdr;   // Netlink message header
    struct ifinfomsg msg;  // First data filled with NIS module info
    char buf[MAX_PAYLOAD]; // Remaining payload
```

```
};

int
create_veth(char *ifname, char *peername)
{
    // Create the netlink socket
    int sock_fd = socket(PF_NETLINK, SOCK_RAW | SOCK_CLOEXEC, NETLINK_ROUTE);
    if (sock_fd < 0)
        return -1;

    uint16_t flags =
              NLM_F_REQUEST  // We build a request message
              | NLM_F_CREATE // Create a device if it doesn't exist
              | NLM_F_EXCL   // Do nothing if it already exists
              | NLM_F_ACK;   // Except an ack as reply or an error

    // Initialise request message
    struct nl_req req = {
        .hdr.nlmsg_len = NLMSG_LENGTH(sizeof(struct ifinfomsg)),
        .hdr.nlmsg_flags = flags,
        .hdr.nlmsg_type = RTM_NEWLINK,
        .msg.ifi_family = PF_NETLINK,
    };
    struct nlmsghdr *hdr = &req.hdr;
    int maxlen = sizeof(req);

    // Attribute r0 with veth info
    addattr_l(hdr, maxlen, IFLA_IFNAME, ifname, strlen(ifname) + 1);

    // Attribute r1 nested within r1, containing iface info
    struct rtattr *linfo =
            addattr_nest(n, maxlen, IFLA_LINKINFO);
    // Specify the device type is veth
    addattr_l(hdr, maxlen, IFLA_INFO_KIND, "veth", 5);

    // r2: another nested attribute
    struct rtattr *linfodata =
        addattr_nest(hdr, maxlen, IFLA_INFO_DATA);

    // r3: nested attribute, contains the peer name
    struct rtattr *peerinfo =
            addattr_nest(n, maxlen, VETH_INFO_PEER);
    n->nlmsg_len += sizeof(struct ifinfomsg);
    addattr_l(n, maxlen, IFLA_IFNAME, peername, strlen(peername) + 1);
    addattr_nest_end(hdr, peerinfo);

    addattr_nest_end(hdr, linfodata);
    addattr_nest_end(hdr, linfo);
```

```
    // Send the message
    sendmsg(sock_fd, &req, 0);
}
```

Maintenant que l'on a notre interface virtuelle, nous pouvons envoyer un second message pour la déplacer dans un nouveau *namespace*[c] :

```
    // Initialise request message
    struct nl_req req = {
            .hdr.nlmsg_len = NLMSG_LENGTH(sizeof(struct ifinfomsg)),
            .hdr.nlmsg_flags = NLM_F_REQUEST | NLM_F_ACK,
            .hdr.nlmsg_type = RTM_NEWLINK,
            .msg.ifi_family = PF_NETLINK,
    };

    addattr_l(&req.hdr, sizeof(req), IFLA_NET_NS_FD, &netns, 4);
    addattr_l(&req.hdr, sizeof(req), IFLA_IFNAME,
              ifname, strlen(ifname) + 1);

    sendmsg(sock_fd, &req, 0);
```

[a] https://tools.ietf.org/html/rfc3549
[b] https://github.com/shemminger/iproute2/blob/main/ip/link_veth.c
[c] https://github.com/shemminger/iproute2/blob/main/ip/iplink.c#L677

Dans cette configuration, ces deux interfaces ne sont pas très utiles, mais si l'on place l'une des deux extrémités dans un autre *namespace* `network`, il devient alors possible de réaliser un échange de paquets entre les deux.

Pour déplacer `veth1` dans notre *namespace* `virli` :

```
42sh# ip link set veth1 netns virli
```

Il ne reste maintenant plus qu'à assigner une IP à chacune des interfaces :

```
42sh# ip netns exec virli ip a add 10.10.10.42/24 dev veth1
42sh# ip a add 10.10.10.41/24 dev veth0
```

Dès lors[6], nous pouvons `pinger` chaque extrêmité :

```
42sh# ping 10.10.10.42
- et -
42sh# ip netns exec virli ping 10.10.10.41
```

Il ne reste donc pas grand chose à faire pour fournir Internet à notre conteneur : via un peu de NAT ou grâce à un pont Ethernet.

2.4.4 Les autres types d'interfaces

Le bridge ou le NAT obligera tous les paquets à passer à travers de nombreuses couches du noyau. Utiliser les interfaces *veth* est plutôt simple et disponible partout, mais c'est loin d'être la technique la plus rapide ou la moins gourmande.

[6] Il peut être nécessaire d'activer chaque lien, via `ip link set vethX up`.

Voyons ensemble les autres possibilités à notre disposition.

2.4.4.1 VLAN

Il est possible d'attribuer juste une interface de VLAN, si l'on a un switch supportant la technologie 802.1q derrière notre machine.

```
42sh# ip link add link eth0 name eth0.100 type vlan id 100
42sh# ip link set dev eth0.100 up
42sh# ip link set eth0.100 netns virli
```

On attribuera alors à chaque conteneur une interface de VLAN différente. Cela peut donner lieu à une configuration de switch(s) assez complexe.

2.4.4.2 MACVLAN

Lorsque l'on n'a pas assez de carte ethernet et que le switch ne supporte pas les VLAN, le noyau met à disposition un routage basé sur les adresses MAC : le MACVLAN. S'il est activé dans votre noyau, vous allez avoir le choix entre l'un des quatre modes : *private*, VEPA, *bridge* ou *passthru*.

Quel que soit le mode choisi, les paquets en provenance d'autres machines et à destination d'une MAC seront délivrés à l'interface possédant ladite MAC. Les différences entre les modes se trouvent au niveau de la communication entre les interfaces.

2.4.4.2.1 VEPA

Dans ce mode, tous les paquets sortants sont directement envoyés sur l'interface Ethernet de sortie, sans qu'aucun routage préalable n'ait été effectué. Ainsi, si un paquet est à destination d'un des autres conteneurs de la machine, c'est à l'équipement réseau derrière la machine de rerouter le paquet vers la machine émettrice (par exemple un switch 802.1Qbg).

Pour construire une nouvelle interface de ce type :

```
42sh# ip link add link eth0 mac0 type macvlan mode vepa
```

2.4.4.2.2 Private

À la différence du mode *VEPA*, si un paquet émis par un conteneur à destination d'un autre conteneur est réfléchi par un switch, le paquet ne sera pas délivré.

Dans ce mode, on est donc assuré qu'aucun conteneur ne pourra parler à un autre conteneur de la même machine.

```
42sh# ip link add link eth0 mac1 type macvlan mode private
```

2.4.4.2.3 Bridge

En mode *Bridge*, les paquets sont routés selon leur adresse MAC : si jamais une adresse MAC est connue, le paquet est délivré à l'interface MACVLAN correspondante ; dans le cas contraire, le paquet est envoyé sur l'interface de sortie.

Pour construire une nouvelle interface de ce type :

```
42sh# ip link add link eth0 mac2 type macvlan mode bridge
```

2.4.4.2.4 passthru

Enfin, le mode *passthru* permet de récupérer le contrôle sur tout ce qu'il reste du périphérique initial (notamment pour lui changer sa MAC propre, ou pour activer le mode de promiscuité).

L'intérêt est surtout de pouvoir donner cette interface à un conteneur ou une machine virtuelle, sans lui donner un accès complet à l'interface physique (et notamment aux autres MACVLAN).

On construit l'interface en mode *passthru* de cette façon :

```
42sh# ip link add link eth0 mac3 type macvlan mode passthru
```

Une seule interface MACVLAN peut être en mode *passthru* par interface physique.

Aller plus loin

Pour approfondir les différentes techniques de routage, je vous recommande cet article : Linux Containers and Networking[7].

Appliqué à Docker, vous apprécierez cet article : Understanding Docker Networking Drivers and their use cases[8].

[7] https://blog.flameeyes.eu/2010/09/linux-containers-and-networking
[8] https://www.docker.com/blog/understanding-docker-networking-drivers-use-cases/

2.5 Le namespace PID

2.5.1 Introduction

L'espace de noms PID est celui qui va nous permettre d'isoler un sous-arbre de processus en créant un nouvel arbre, qui aura son propre processus considéré comme l'init.

Contrairement aux autres *namespaces* où l'on peut demander à se séparer du *namespace* en question à n'importe quel moment de l'exécution du processus, via unshare(2) ou setns(2) par exemple, ici, le changement ne sera valable qu'après le prochain fork(2) (ou similaire). En effet, l'espace de noms n'est pas changé, afin que le processus ne change pas de PID en cours de route, puisqu'il dépend du *namespace* dans lequel il se trouve.

2.5.2 Isolons !

Première étape s'isoler :

```
42sh# unshare --pid --fork /bin/bash
    inpidns# echo $$
    1
```

?

Qu'est-ce qu'il se passe sans l'option --fork ?

Nous utilisons ici l'option --fork, pour que le passage dans le nouvel espace de noms des PID soit effectif (cf. Introduction). Si on l'omet, voici ce qu'il se passe :

```
42sh# unshare --pid /bin/sh
    inpidns# echo $$
    23456
```

Ce n'est apparemment pas ce que l'on souhaitait : on est toujours dans l'ancien *namespace*, puisqu'on a juste unshare(2), sans fork(2). Si l'on va plus loin et que l'on fork en demandant à sh d'exécuter un nouveau processus :

```
42sh# unshare --pid /bin/sh
    inpidns# echo $$
    34567
    inpidns# sh
      inpidns# echo $$
      1
```

C'est d'ailleurs le bon moment pour regarder le contenu de notre fichier pid_for_children pour le processus 34567 :

```
42sh# ls -l /proc/34567/ns/pid*
lrwxrwxrwx 1 nemunaire 1 oct. 23:42 pid    -> 'pid:[4026531836]'
lrwxrwxrwx 1 nemunaire 1 oct. 23:42 pid_for_children -> 'pid:[4026532993]'
```

On remarque bien que l'on se trouve dans un entre-deux : pid pointe toujours sur l'*inode* de l'arbre de PID initial, tandis que pid_for_children est prêt à transmettre le nouvel *inode* aux nouveaux processus.

Pourquoi ça ne fonctionne pas du tout avec bash ?

?

Si on utilise bash à la place de sh dans les exemples précédents, toujours sans l'option --fork, on obtient une erreur plutôt étrange :

```
42sh# unshare --pid /bin/bash
    inpidns# echo $$
    65432
    inpidns# sh
    -bash: fork: Cannot allocate memory
```

Il se trouve que bash commence par fork(2) lui-même afin de réaliser un certain nombre d'opérations. Notre PID 1, le premier PID de notre conteneur, a donc été alloué à un processus d'initialisation de bash, qui s'est terminé depuis.

Le comportement du noyau, lorsque le PID 1 se termine, est de lancer un *kernel panic* (car c'est un processus indispensable, notamment de part son rôle de parent pour tous les processus orphelin). Au sein d'un *namespace* PID qui n'est pas le *namespace* racine, le noyau appelle la fonction disable_pid_allocation qui retire le *flag* PIDNS_HASH_ADDING de l'espace de nom. Le fait de ne pas avoir PIDNS_HASH_ADDING fait retourner ENOMEM à la fonction alloc_pid appelée par fork(2) et clone(2). On n'a donc pas d'autre choix que de quitter ce *namespace* pour en recréer un nouveau.

Un coup d'œil à top ou ps aux devrait nous montrer que l'on est maintenant le seul processus … pourtant, il n'en est rien, ces deux commandes continuent d'afficher la liste complète des processus de notre système.

Cela est dû au fait que ces deux programmes, sous Linux, se basent sur le contenu de /proc. D'ailleurs, si l'on affiche le PID du processus courant echo $$, on obtient bien 1.

En l'état, beaucoup d'informations sont divulguées. Mais il n'est pas possible de monter le bon /proc car il serait également monté pour les processus de notre système initial. Pour s'en sortir, il est nécessaire de s'isoler dans un *namespace* mount séparé.

2.5.3 Double isolation : ajout du namespace mount

Voici la nouvelle ligne de commande que l'on va utiliser :

```
42sh# unshare --pid --mount --fork --mount-proc /bin/bash
```

Avec l'option --mount-proc, unshare va s'occuper de monter le nouveau /proc.

Cette fois, top et ps nous rapportent bien que l'on est seul dans notre *namespace*.

2.5.4 Arborescence à l'extérieur du namespace

Lors de notre première tentative de top, lorsque /proc était encore monté sur le procfs de l'espace de noms initial : notre processus (au PID 1 dans son nouveau *namespace*) était présent dans l'arborescence de l'espace initial avec un PID dans la continuité des autres processus, étonnant !

En fait, l'isolation consiste en une virtualisation des numéros du processus : la plupart des processus du système initial ne sont pas accessibles, et ceux qui font partie de l'espace de noms créé disposent d'une nouvelle numérotation. Et c'est cette nouvelle numérotation qui est montrée au processus.

Si l'on veut interagir avec ce processus depuis un de ses espaces de noms parent, il faut le faire avec son identifiant de processus du même *namespace* que le processus appelant.

2.5.5 Processus orphelins et `nsenter`

Au sein d'un *namespace*, le processus au PID 1 est considéré comme le programme `init`, les mêmes propriétés s'appliquent donc.

Si un processus est orphelin, il est donc affiché comme étant fils du PID 1 dans son *namespace*[9] ; il n'est pas sorti de l'espace de noms.

Lorsqu'on lance un processus via `nsenter(1)` ou `setns(2)`, cela crée un processus qui n'est sans doute pas un fils direct du processus d'`init` de notre conteneur. Malgré tout, même s'il est affiché comme n'étant pas un fils à l'extérieur du conteneur, les propriétés d'`init` sont biens appliquées à l'intérieur pour conserver un comportement cohérent.

Aller plus loin

N'hésitez pas à jeter un œil à la page de manuel consacrée à cet espace de noms : `pid_namespaces(7)`.

[9] en réalité, ce comportement est lié à la propriété `PR_SET_CHILD_SUBREAPER`, qui peut être définie pour n'importe quel processus de l'arborescence. Le processus au PID 1 hérite forcément de cette propriété ; il va donc récupérer tous les orphelins, si aucun de leurs parents n'a la propriété définie.

2.6 Le namespace `mount`

L'espace de noms `mount` permet d'isoler la vision du système de fichiers qu'ont un processus et ses fils.

Peut-être que l'on peut trouver avec ça, un moyen de faire un `chroot` plus sûr ?

Attention il convient de prendre garde aux types de liaison existant entre vos points de montage (voir la partie sur les particularités des points de montage), car les montages et démontages pourraient alors être répercutés dans l'espace de noms parent.

Une manière rapide pour s'assurer que nos modifications ne sortiront pas de notre *namespace* est d'appliquer le type esclave à l'ensemble de nos points de montage, récursivement, dès que l'on est entré dans notre nouvel espace de noms.

```
mount --make-rslave /
```

2.6.1 Préparation du changement de racine

Nous allons essayer de changer la racine de notre système de fichier. À la différence d'un `chroot (2)`, changer de racine est quelque chose d'un peu plus sportif car il s'agit de ne plus avoir aucune trace de l'ancienne racine. Au moins ici, il ne sera certainement pas possible de revenir en arrière dans l'arborescence !

Pour l'instant, votre système utilise sans doute la partition d'un disque physique comme racine de son système de fichier. Le changement de racine, va nous permettre d'utiliser un autre système.

Bien sûr, nous n'allons pas changer la racine de votre système hôte, nous allons faire cela dans un *namespace* qui nous permet d'avoir des points de montage virtuels. Le changement de racine sera donc effectif uniquement dans cet espace de noms.

2.6.1.1 L'environnement

Pour pouvoir changer de racine, il est nécessaire que la nouvelle racine soit la racine d'un point de montage, comme l'explique `pivot_root(2)`. En effet, il serait encore possible hypothétiquement de remonter dans l'arborescence si l'on ne se trouvait pas à la racine d'une partition au moment du basculement.

Si vous n'avez pas de partition à disposition, vous pouvez utiliser un `tmpfs` :

```
42sh# mkdir /mnt/newroot
42sh# mount -t tmpfs none /mnt/newroot
```

Placez ensuite dans cette nouvelle racine le système de votre choix.

myswitch_root.sh

2.6.2 Changer de racine – `myswitch_root.sh`

Voici les grandes étapes du changement de racine :

1. S'isoler dans les *namespaces* adéquats ;
2. Démonter ou déplacer toutes les partitions de l'ancienne racine vers la nouvelle racine ;
3. `pivot_root` !

2.6.2.1 S'isoler

Notre but étant de démonter toutes les partitions superflues, nous allons devoir nous isoler sur :

— les points de montages, ça semble évident ;
— les PIDs : car on ne pourra pas démonter une partition en cours d'utilisation. S'il n'y a pas de processus, il n'y a personne pour nous empêcher de démonter une partition !
— les autres *namespaces* ne sont pas forcément nécessaires.

Isolons-nous :

```
42sh# unshare -p -m -f --mount-proc
```

Avant de pouvoir commencer à démonter les partitions, il faut s'assurer que les démontages ne se propagent pas via une politique de *shared mount*.

2.6.2.2 Dissocier la propagation des démontages

Commençons donc par étiqueter tous nos points de montage (de ce *namespace*), comme esclaves :

```
42sh# mount --make-rslave /
```

2.6.2.3 Démonter tout !

À vous maintenant de démonter vos points d'attache. Il ne devrait vous rester après cette étape que : `/`, `/dev`, `/sys`, `/proc`, `/run` et leurs fils.

2.6.2.4 Switch !

À ce stade, dans votre console, vous avez plusieurs solutions : utiliser `switch_root(8)` ou `pivot_root(8)`. La première abstrait plus de choses que la seconde.

`switch_root`

Cette commande s'occupe de déplacer les partitions restantes pour vous, et lance la première commande (*init*) de votre choix.

`pivot_root`

Cette commande, plus proche du fonctionnement de l'appel système `pivot_root(2)`, requiert de notre part que nous ayons préalablement déplacé les partitions systèmes à leur place dans la nouvelle racine.

L'appel de la commande sert à intervertir les deux racines ; elle prend en argument :

— le chemin de la nouvelle racine,
— le chemin dans la nouvelle racine où placer l'ancienne.

Une fois le pivot effectué, on peut démonter l'ancienne racine.

Pour lancer la première commande dans la nouvelle racine, on passe généralement par :

```
42sh# exec chroot / command
```

Erreurs courantes

Voici une liste des erreurs les plus courantes que vous allez sans doute rencontrer :

EINVAL
- La nouvelle racine n'est pas un point de montage.
- Le chemin où placer l'ancienne racine n'est pas sur la nouvelle racine.
- Le point de montage de la nouvelle ou l'ancienne racine utilise le type de propagation *shared*.

EBUSY La nouvelle racine ou le chemin où mettre l'ancienne racine se trouve sur la racine actuelle.
ENOTDIR La nouvelle racine ou le chemin où mettre l'ancienne racine ne sont pas des dossiers.
EPERM Le processus appelant ne dispose pas de la capability `CAP_SYS_ADMIN`.

Vous pouvez retrouver toutes les erreurs dans le manuel de `pivot_root(2)`.

Assemblage

Vous devriez maintenant pouvoir réaliser un script `myswitch_root.sh` qui enchaîne toutes les étapes précédentes.

On considère préalablement que l'environnement est propice à la réalisation de ce script :

```
42sh# mkdir -p /mnt/newroot
42sh# mount -t tmpfs none /mnt/newroot
42sh# wget https://dl-cdn.alpinelinux.org/alpine/v3.14/releases/x86_64/alpine-
    minirootfs-3.14.8-x86_64.tar.gz
42sh# tar xpf alpine-minirootfs-*.tar.gz -C /mnt/newroot
42sh# cd /
```

Puis on s'attend à le lancer ainsi :

```
42sh# ~/myswitch_root.sh /mnt/newroot /bin/bash
  innewns# _
```

2.7 Le namespace user

2.7.1 Introduction

L'espace de noms `user` est plutôt pratique car il permet de virtualiser la liste et les droits des utilisateurs.

Par exemple, on va pouvoir entrer dans un conteneur en tant que super-utilisateur à partir d'un compte d'un simple utilisateur. Il nous sera alors possible d'effectuer toutes les actions privilégiées dont nous pourrions avoir besoin à l'intérieur de cet espace de noms, sans que cela ne réduise la sécurité des composants à l'extérieur de cet espace.

2.7.2 Comportement vis-à-vis des autres namespaces

Alors qu'il est normalement nécessaire d'avoir des privilèges pour créer de nouveaux espaces de noms, en commençant par demander un *namespace* utilisateur, on obtient les privilèges requis pour créer tous les autres types de *namespaces*.

Grâce à cette technique, il est possible de lancer des conteneurs en tant que simple utilisateur ; les projets `podman` et Singularity, ... reposent en grande partie sur cela.

2.7.3 Correspondance des utilisateurs et des groupes

Comme pour les autres espaces de noms, le *namespace* `user` permet de ne garder dans le nouvel espace, que les utilisateurs et les groupes utiles au processus, en les renumérotant au passage si besoin.

2.7.3.1 L'utilisateur -2 : nobody

Lorsque l'on arrive dans un nouvel espace, aucun utilisateur ni groupe n'est défini. Dans cette situation, tous les identifiants d'utilisateur et de groupe, renvoyés par le noyau sont à -2 ; valeur qui correspond par convention à l'utilisateur *nobody* et au groupe *nogroup*.

-1 étant réservé pour indiquer une erreur dans le retour d'une commande, ou la non-modification d'un paramètre passé en argument d'une fonction.

2.7.3.2 `uid_map` et `gid_map`

Pour établir la correspondance, une fois que l'on a créé le nouveau *namespace*, ces deux fichiers, accessibles dans `/proc/self/`, peuvent être écrits une fois.

`uid_map`

Sur chaque ligne, on doit indiquer :

— L'identifiant marquant le début de la plage d'utilisateurs, pour le processus en question.
— L'identifiant marquant le début de la plage d'utilisateurs, pour le processus affichant le fichier.
— La taille de la plage.

Par exemple, le *namespace* `user` initial défini la correspondance suivante :

```
42sh$ cat /proc/self/uid_map
    0   0   4294967295
```

Cela signifie que les utilisateurs dont l'identifiant court de 0 à `MAX_INT - 2` inclus, dans cet espace de noms, correspondent aux utilisateurs allant de 0 à `MAX_INT - 1` inclus, pour le processus affichant ce fichier.

Lorsque l'on crée un *namespace* `user`, généralement, la correspondance vaut :

```
42sh$ cat /proc/self/uid_map
    0   1000    1
```

Dans cette situation, on comprend que notre processus considère que l'utilisateur root, dans le conteneur équivaut à l'utilisateur 1000 hors de l'espace de noms.

`gid_map`

Le principe est identique pour ce fichier, mais agit sur les correspondances des groupes au lieu des utilisateurs.

Il y a cependant un subtilité car il faut écrire la chaîne `deny` dans le fichier `setgroups` avant de modifier `gid_map`.

2.7.4 Utilisation basique de l'espace de noms

Avec `unshare(1)`, voici comment, en tant que simple utilisateur, se dissocier de plusieurs *namespace*s en gardant un environnement cohérent, dans lequel on devient le super-utilisateur :

```
42sh$ unshare --mount --pid --mount-proc --fork --net --user \
          --map-root-user bash
```

Un `capsh --print` nous montre que l'on est bien `root` et que l'on possède toutes les *capabilities*. Cependant, cela ne signifie pas que l'on a tous les droits sur le système ; il y a plusieurs niveaux de validation qui entrent en jeu. L'idée étant que l'on a été désigné root dans son conteneur, on devrait pouvoir y faire ce que l'on veut, tant que l'on n'agit pas en dehors.

Aller plus loin

N'hésitez pas à jeter un œil à la page du manuel consacrée à ce *namespace* : `user_namespaces(7)`.

mydocker_exec

2.8 docker exec

Après voir lu la partie concernant les *namespaces*, vous avez dû comprendre qu'un `docker exec`, n'était donc rien de plus qu'un `nsenter(1)`.

Réécrivons, en quelques lignes, la commande `docker exec` !

Pour savoir si vous avez réussi, comparez les sorties des commandes :

- `ip address` ;
- `hostname` ;
- `mount` ;
- `ps -aux` ;
- ...

Voici quelques exemples pour tester :

```
42sh$ docker run --name mywebsrv -d -p 80:80 nginx
d63ceae863956f8312aca60b7a57fbcc1fdf679ae4c90c5d9455405005d4980a
42sh$ docker container inspect --format '{{ .State.Pid }}' mywebsrv
234269

42sh# ./mydocker_exec mywebsrv ip address
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
      inet 127.0.0.1/8 scope host lo
          valid_lft forever preferred_lft forever
13: eth0@if14: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 state UP
     link/ether 02:42:ac:11:00:02 brd ff:[...]:ff link-netnsid 0
     inet 172.17.0.1/16 scope global eth0
         valid_lft forever preferred_lft forever

42sh# hostname
koala.zoo.paris
42sh# ./mydocker_exec mywebsrv hostname
d63ceae86395

42sh# ./mydocker_exec mywebsrv mount
42sh# ./mydocker_exec mywebsrv ps aux
...
```

3 Rendu

Est attendu d'ici le cours suivant :

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

3.1 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) :

```
./cmpns.sh
./mydocker_exec.sh
./myswitch_root.sh
```

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 `rendu4`, ... 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).