A la découverte des namespaces mount et uts

Rachid Koucha
[Ingénieur développement logiciel]

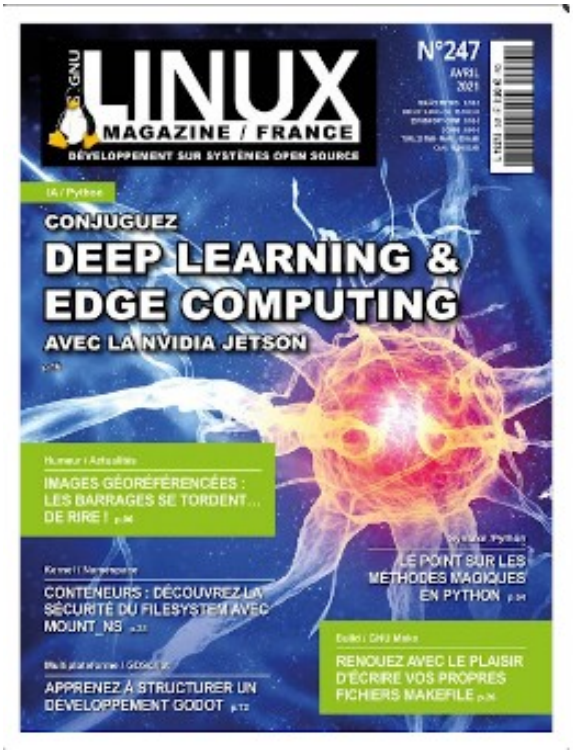
Le namespace mount, premier d'une longue série de namespaces, a été ajouté à Linux quelques années après chroot() afin d'offrir plus de possibilités et de sécurité dans l'isolation des systèmes de fichiers. Introduit peu après et indéniablement plus simple, le namespace uts permet d'instancier les noms de machine. Les conteneurs sont bien-entendus les premiers clients de ces fonctionnalités.

Table des matières

Avant-propos

Le code source des exemples utilisés dans cet article sont disponibles sur Github : https://github.com/Rachid-Koucha/linux_ns.

Cet article a été publié dans GNU Linux Magazine France n°247 du mois d'avril 2021 :



Introduction

Utilisé conjointement avec la notion de propagation des points de montage, le mount namespace (mount_ns) est une solution extrêmement flexible pour isoler tout ou partie du système de fichiers. Le namaspece uts (uts_ns), d'une utilisation plus aisée, donne la possibilité de voir la machine sous différents noms afin d'en simuler plusieurs. Ces deux mécanismes sont bien-entendus avant tout destinés aux conteneurs tels que LXC.

1 Le namespace MOUNT

Premier namespace implémenté dans le système Linux, il isole les points de montage dans le système de fichiers. Une entrée est dédiée à ce namespace dans le manuel en ligne de Linux : **man 7 mount_namespaces**.

1.1 Création

A la création du namespace, la liste des points de montage est la copie de celles du mount_ns du processus appelant **unshare()** ou **clone()**. Ensuite les processus associés à un mount_ns donné peuvent ajouter ou retirer des points de montage. Pour un processus d'identifiant **pid**, les points de montage vus de son mount_ns sont consultables dans le répertoire **/proc/<pid>** (cf. **man 5 proc**) via ces entrées:

- **mounts** : liste des systèmes de fichiers montés ;

- **mountinfo** : amélioration du précédent fichier avec prise en compte des types de propagation et des bind mounts ;

- **mountstats** : statistiques et informations de configuration sur les points de montage.

1.2 Types de propagations

Exécutons un sous-shell dans un nouveau mount_ns avec notre commande **shns**. Nous prenons soin de positionner la variable d'environnement **PS1** de sorte à différencier le prompt de celui du shell initial, dans le but de faciliter la lecture des copies d'écrans. Puis lançons la commande **lsblk** :

```
# PS1='SHNS# ' ./shns mnt
New namespace 'mnt'
SHNS# lsblk
NAME   MAJ:MIN RM   SIZE RO TYPE MOUNTPOINT
[...]
sda      8:0    0 465,8G  0 disk
`-sda1   8:1    0 465,8G  0 part /
sdb      8:16   0   2,7T  0 disk
|-sdb1   8:17   0 119,2G  0 part [SWAP]
`-sdb2   8:18   0   2,6T  0 part /home
sdc      8:32   0 119,2G  0 disk
|-sdc1   8:33   0  93,1G  0 part /tmp
|-sdc2   8:34   0     1K  0 part
`-sdc5   8:37   0  26,1G  0 part /var/tmp
```

Insérons une clé USB et relançons la commande **lsblk** pour voir quel nouveau périphérique apparaît pour la contrôler. Ici c'est **/dev/sdd1** :

```
SHNS# lsblk
NAME   MAJ:MIN RM   SIZE RO TYPE MOUNTPOINT
[...]
sda      8:0    0 465,8G  0 disk
`-sda1   8:1    0 465,8G  0 part /
sdb      8:16   0   2,7T  0 disk
|-sdb1   8:17   0 119,2G  0 part [SWAP]
`-sdb2   8:18   0   2,6T  0 part /home
sdc      8:32   0 119,2G  0 disk
|-sdc1   8:33   0  93,1G  0 part /tmp
|-sdc2   8:34   0     1K  0 part
`-sdc5   8:37   0  26,1G  0 part /var/tmp
```

```
sdd      8:48   1   7,5G  0 disk
`-sdd1   8:49   1   7,5G  0 part /media/rachid/USBKEY
```

Comme c'est la cas ici sur **/media**, la clé peut être montée automatiquement dans les deux mount_ns. Si c'est le cas, démontons-la :

```
SHNS# umount /dev/sdd1
SHNS# lsblk
NAME   MAJ:MIN RM   SIZE RO TYPE MOUNTPOINT
[...]
sda      8:0    0 465,8G  0 disk
`-sda1   8:1    0 465,8G  0 part /
sdb      8:16   0   2,7T  0 disk
|-sdb1   8:17   0 119,2G  0 part [SWAP]
`-sdb2   8:18   0   2,6T  0 part /home
sdc      8:32   0 119,2G  0 disk
|-sdc1   8:33   0  93,1G  0 part /tmp
|-sdc2   8:34   0     1K  0 part
`-sdc5   8:37   0  26,1G  0 part /var/tmp
sdd      8:48   1   7,5G  0 disk
`-sdd1   8:49   1   7,5G  0 part
```

Le démontage dans l'un des mount_ns se répercute dans l'autre. En effet, dans un autre terminal (dans le mount_ns initial donc), nous ne voyons plus le montage non plus :

```
# lsblk
NAME   MAJ:MIN RM   SIZE RO TYPE MOUNTPOINT
[...]
sdc      8:32   0 119,2G  0 disk
|-sdc1   8:33   0  93,1G  0 part /tmp
|-sdc2   8:34   0     1K  0 part
`-sdc5   8:37   0  26,1G  0 part /var/tmp
sdd      8:48   1   7,5G  0 disk
`-sdd1   8:49   1   7,5G  0 part
```

Montons la clé sur un autre répertoire (par exemple **/mnt**) à partir du sous-shell :

```
SHNS# mount /dev/sdd1 /mnt
SHNS# df
Filesystem      1K-blocks      Used  Available Use% Mounted on
/dev/sda1       479668904  11084140  444149156   3% /
udev              8148432         0    8148432   0% /dev
tmpfs             8175840     30996    8144844   1% /dev/shm
[...]
/dev/sdd1         7816240    157832    7658408   3% /mnt
```

La commande **df** dans le shell initial montre que la clé est aussi montée :

```
# df
Filesystem      1K-blocks      Used  Available Use% Mounted on
udev              8148432         0    8148432   0% /dev
tmpfs             1635172      1836    1633336   1% /run
/dev/sda1       479668904  11084140  444149156   3% /
[...]
/dev/sdd1         7816240    157832    7658408   3% /mnt
```

Un montage ou un démontage dans un nouveau mount_ns se propage donc sur le mount_ns initial.

C'est un fonctionnement qui peut avoir ses avantages comme ses inconvénients quand on parle d'isolation. En effet, on pourrait vouloir ce fonctionnement ou au contraire éviter qu'un montage dans un conteneur se voit côté hôte. Tout dépend des besoins.

Cela a été anticipé (ou plutôt rectifié, comme le dit le manuel) : les mount_ns qui offraient une trop grande isolation au départ ont été assouplis avec l'ajout de la notion de **type de propagation des évènements**. Le fichier **/proc/<pid>/mountinfo** permet de voir cette valeur avec sa propre terminologie (cf. **man 7 mount_namespaces** à la rubrique « SHARED SUBTREES ») au niveau du septième champ des lignes affichées (situé juste avant le tiret, il peut être vide ou renseigné). Dans notre cas de figure, le contenu de ce fichier dans le sous-shell indique la valeur « shared » :

```
SHNS# cat /proc/$$/mountinfo
1116 473 8:2 / / rw,relatime shared:1 - ext4 /dev/sda2 rw,errors=remount-ro
[...]
1215 1166 0:25 / /dev/shm rw,nosuid,nodev shared:4 - tmpfs tmpfs rw
[...]
```

L'explication détaillée nécessiterait un article fastidieux. Pour cet exposé, nous nous contenterons de dire que le type de propagation détermine comment les montages/démontages sur les répertoires **situés directement sous un point de montage donné sont propagés**. Pour plus d'informations, il convient de se

référer à la documentation du noyau [1] et [2]. Il s'agit d'un drapeau qui peut prendre l'une de ces valeurs :

- **MS_SHARED** : Ce point de montage partage ses montages/démontages avec les autres membres de son groupe ;

- **MS_PRIVATE** : Ce point de montage ne partage pas ses montages/démontages ;

- **MS_SLAVE** : Les montages/démontages se propagent sous ce point de montage de la part d'un groupe maître. Par contre, ce point de montage ne partage pas les montages/démontages dans l'autre sens. C'est un compromis entre **MS_SHARED** et **MS_PRIVATE** ;

- **MS_UNBINDABLE** : en plus d'être **MS_PRIVATE**, ce point de montage ne peut pas être « bindé » (notion que nous verrons dans la suite).

Le drapeau est passé à l'appel système **mount()** dans le paramètres « mountflags » (cf. quatrième argument dans **man 2 mount**). Il est même possible d'appliquer les drapeaux précédents à toute une arborescence sous un point de montage en ajoutant le drapeau **MS_REC** (i.e. récursif). L'article [3] présente quelques cas concrets.

Ces drapeaux sont aussi proposés par la commande **mount** du shell à travers des options sur sa ligne de commande décrites dans la rubrique « Shared subtree operations » de **man 8 mount** : **--make-shared**, **--make-private**, **--make-slave** et **--make-unbindable**. Pour l'aspect récursif, ces options se déclinent avec l'ajout d'un « r » dans le nom : **--make-rshared**, **--make-rprivate**, **--make-rslave** et **--make-runbindable**.

Pour revenir à notre exemple ci-dessus, où la clé USB montée dans le mount_ns du sous-shell est aussi vue dans le mount_ns initial, s'explique par le fait que le répertoire **/** dans le mount_ns initial a le drapeau **MS_SHARED** (c'est la signification de la valeur « shared:1 » retournée par l'affichage de **mountinfo**).

La liste des points de montage de tout nouveau mount_ns étant la copie de celle du mount_ns du processus qui le crée, nous avons les mêmes propriétés dans le sous-shell :

```
SHNS# cat /proc/$$/mountinfo
[...]
579 557 8:1 / / rw,relatime shared:1 - ext4 /dev/sda1 rw,errors=remount-ro
[...]
```

A partir du sous-shell, démontons la clé USB (l'action se propage aussi au namespace initial) puis changeons le type de propagation **MS_SHARED** de **/** en **MS_SLAVE** avec l'option idoine de la commande **mount** :

```
SHNS# umount /mnt
SHNS# mount --make-slave /
SHNS# cat /proc/$$/mountinfo
[...]
579 557 8:1 / / rw,relatime master:1 - ext4 /dev/sda1 rw,errors=remount-ro
[...]
```

Nous vérifions que le type de propagation sur le mount_ns initial n'a pas changé (i.e. **shared**) :

```
# cat /proc/1/mountinfo
[...]
27 1 8:1 / / rw,relatime shared:1 - ext4 /dev/sda1 rw,errors=remount-ro
[...]
```

Avec l'opération **mount** précédente nous avons indiqué que le point de montage **/** dans le mount_ns du sous-shell est l'esclave du point de montage **/** dans le mount_ns initial. Donc un montage sur **/mnt** dans le sous-shell ne se propage plus dans le mount_ns initial :

```
SHNS# mount /dev/sdd1 /mnt
SHNS# df
Filesystem      1K-blocks      Used  Available Use% Mounted on
/dev/sda1       479668904  11084152  444149144   3% /
udev              8148432         0    8148432   0% /dev
tmpfs             8175840     30996    8144844   1% /dev/shm
[...]
/dev/sdd1         7816240    157832    7658408   3% /mnt
```

On vérifie que dans le mount_ns initial, la clé USB n'est pas vue sur **/mnt** :

```
# df
Filesystem      1K-blocks      Used  Available Use% Mounted on
udev              8148432         0    8148432   0% /dev
tmpfs             1635172      1836    1633336   1% /run
/dev/sda1       479668904  11084152  444149144   3% /
[...]
```

Démontons la clé USB dans le sous-shell :

```
SHNS# umount /mnt
```

Un montage/démontage dans le mount_ns initial sera propagé dans le mount_ns du sous-shell. Pour le vérifier, montons la clé dans le mount_ns initial :

```
# mount /dev/sdd1 /mnt
# df
Filesystem      1K-blocks      Used  Available Use% Mounted on
udev              8148432         0    8148432   0% /dev
tmpfs             1635172      1836    1633336   1% /run
/dev/sda1       479668904  11084152  444149144   3% /
tmpfs             8175840     30996    8144844   1% /dev/shm
[...]
/dev/sdd1         7816240    157832    7658408   3% /mnt
```

Si nous lançons **df** dans le sous-shell, on voit aussi la clé USB :

```
SHNS# df
Filesystem      1K-blocks      Used  Available Use% Mounted on
/dev/sda1       479668904  11084152  444149144   3% /
udev              8148432         0    8148432   0% /dev
tmpfs             8175840     30996    8144844   1% /dev/shm
[...]
/dev/sdd1         7816240    157832    7658408   3% /mnt
```

Nous pouvons démonter la clé côté sous-shell, elle restera montée dans le namespace initial. Par contre, si nous la démontons côté mount_ns initial, elle sera aussi démontée côté sous-shell. Nous avons ainsi mis en pratique le cas d'un point de montage en mode esclave : les montages/démontages se propagent sur lui par contre il ne les propagent pas dans le sens inverse.

Pour les exemples précédents, nous avons utilisé notre programme **shns** au lieu de la commande **unshare** car l'option **-m** de cette dernière déclenche l'appel système **mount()** avec les drapeaux **MS_REC|MS_PRIVATE** sur **/** entre l'appel à **unshare(CLONE_NEWNS)** et l'exécution de la commande. Il suffit de lancer la commande sous le contrôle de **strace** pour s'en convaincre :

```
# strace -f unshare -m /bin/sh
[...]
unshare(CLONE_NEWNS)                    = 0
mount("none", "/", NULL, MS_REC|MS_PRIVATE, NULL) = 0
execve("/bin/sh", ["/bin/sh"], 0x7ffeede107a8 /* 25 vars */) = 0
[...]
```

En d'autres termes, tous les points de montage de l'arborescence sont « privatisés » dans le nouveau mount_ns. Par conséquent, tout montage/démontage dans le sous-shell n'est pas propagé dans le mount_ns initial et inversement, les montages/démontage dans le mount_ns initial ne se propage pas dans le sous-shell. Dans le fichier **mountinfo**, le septième champ des lignes (situé juste avant le tiret) est vide dans ce cas :

```
# cat /proc/$$/mountinfo
1314 1311 8:2 / / rw,relatime - ext4 /dev/sda2 rw,errors=remount-ro
1315 1314 0:6 / /dev rw,nosuid,relatime - devtmpfs udev rw,size=7869980k,nr_inodes=1967495,mode=755
1316 1315 0:23 / /dev/pts rw,nosuid,noexec,relatime - devpts devpts rw,gid=5,mode=620,ptmxmode=000
```

Il est en réalité possible de changer ce comportement par défaut de la commande **unshare** avec l'option **--propagation** plus facile à comprendre maintenant que la notion de type de propagation a été présenté (cf. **[3]** pour les exemples d'utilisation).

Le système de fichiers du conteneur LXC **busybox** est monté en mode esclave sur tous les points de montage de l'arborescence (i.e. **MS_REC|MS_SLAVE**). Lançons le conteneur **bbox** comme indiqué dans l'encadré du premier article de notre série **[8]** et affichons le contenu de **mountinfo** dans sa console :

```
# lxc-console -n bbox -t 0
[...]
bbox# cat /proc/$$/mountinfo
1298 1241 8:1 /var/lib/lxc/bbox/rootfs / rw,relatime master:1 - ext4 /dev/sda1 rw,errors=remount-ro
[...]
```

Le système de fichiers du conteneur préparé par le template **busybox** lors de l'appel à **lxc-create**, est créé dans **/var/lib/lxc/bbox/rootfs**. Si nous montons une clé USB sur le répertoire **/var/lib/lxc/bbox/rootfs**, du côté hôte :

```
# mount /dev/sdd1 /var/lib/lxc/bbox/rootfs/mnt
# df
Filesystem      1K-blocks      Used  Available Use% Mounted on
udev              8148432         0    8148432   0% /dev
tmpfs             1635172      1828    1633344   1% /run
```

```
/dev/sda1       479668904  11102416  444130880   3% /
[...]
/dev/sdd1      1921771720 386188936 1437892368  22% /var/lib/lxc/bbox/rootfs/mnt
```

Cette clé sera vue dans **/mnt** côté conteneur conformément à la propagation **MS_SLAVE** positionnée :

```
bbox# df
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/sda1            479668904  11102416 444130880   2% /
none                       492         4       488   1% /dev
[...]
/dev/sdd1            1921771720 386188936 1437892368  21% /mnt
```

Démontons la clé USB côté conteneur :

```
bbox# umount /mnt
bbox# df
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/sda1            479668904  11102416 444130880   2% /
none                       492         4       488   1% /dev
```

Elle reste montée côté hôte :

```
# df
Filesystem      1K-blocks      Used  Available Use% Mounted on
udev              8148432         0    8148432   0% /dev
tmpfs             1635172      1828    1633344   1% /run
/dev/sda1       479668904  11102416  444130880   3% /
[...]
/dev/sdd1      1921771720 386188936 1437892368  22% /var/lib/lxc/bbox/rootfs/mnt
```

Après la notion de type de propagation des évènements de montage/démontage, un autre besoin se fait sentir au niveau des conteneurs : le partage partiel du système de fichiers avec le hôte.

1.3 Bind mount

Un conteneur démarrant avec son propre mount_ns, a au départ la copie de la liste des points de montage du hôte. En d'autres termes, le **rootfs** du hôte est partagé avec le celui du conteneur. Mais les conteneurs font le plus souvent un **pivot_root()**, variante plus sûre de **chroot()**, pour avoir leur propre **rootfs** dans un sous-arbre du système de fichiers du hôte. Par exemple, le conteneur **busybox** de LXC, « localise » son **rootfs** dans le sous arbre **/var/lib/lxc/<nom_conteneur>/rootfs** (créé au moment de l'appel à **lxc-create**). La figure 1 illustre cela pour **bbox**.

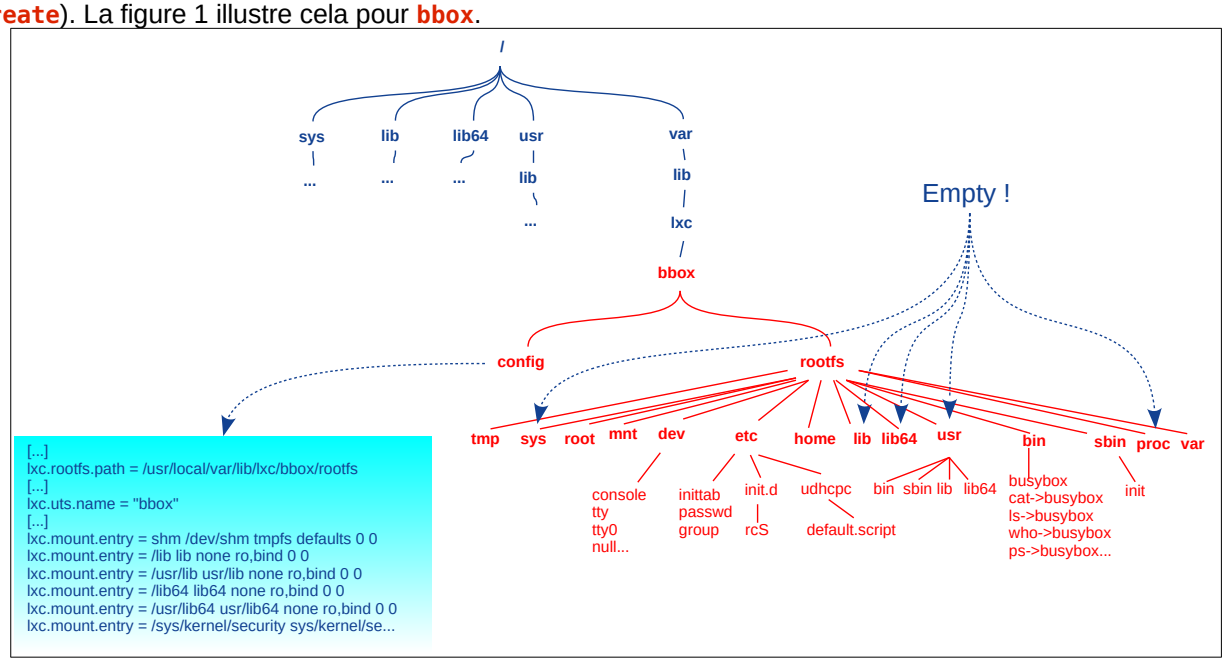


Fig. 1 : Espace du conteneur après lxc-create

Affichons sur hôte le contenu de ce répertoire :

```
# ls -la /var/lib/lxc/bbox/rootfs
total 68
drwxr-xr-x 17 root root 4096 févr.  5 14:35 .
drwxrwx---  3 root root 4096 févr.  4 18:40 ..
```

```
drwxr-xr-x  2 root root 4096 janv.  25 20:27 bin
drwxr-xr-x  4 root root 4096 janv.  25 20:27 dev
drwxr-xr-x  4 root root 4096 janv.  25 20:27 etc
drwxr-xr-x  2 root root 4096 janv.  25 20:27 home
drwxr-xr-x  2 root root 4096 janv.  25 20:27 lib
[...]
drwxr-xr-x  6 root root 4096 janv.  25 20:27 usr
drwxr-xr-x  4 root root 4096 janv.  25 20:27 var
```

La configuration LXC pour ce conteneur spécifie ce chemin comme **rootfs** :

```
# cat /usr/local/var/lib/lxc/bbox/config
[...]
lxc.rootfs.path = dir:/usr/local/var/lib/lxc/bbox/rootfs
[...]
```

A l'appel de **lxc-start**, le conteneur **bbox** a sa racine située à cet endroit. Son contenu est donc le même mais dans un nouveau mount_ns :

```
# lxc-console -n bbox -t 0
[...]
bbox# ls -l /
total 64
drwxr-xr-x   17 root     root          4096 Feb  5 13:35 .
drwxr-xr-x   17 root     root          4096 Feb  5 13:35 ..
drwxr-xr-x    2 root     root          4096 Jan 25 19:27 bin
drwxr-xr-x    3 root     root           340 Feb  5 13:35 dev
drwxr-xr-x    4 root     root          4096 Jan 25 19:27 etc
drwxr-xr-x    2 root     root          4096 Jan 25 19:27 home
drwxr-xr-x  153 root     root         12288 Jan 24 16:03 lib
[...]
drwxr-xr-x    6 root     root          4096 Jan 25 19:27 usr
drwxr-xr-x    4 root     root          4096 Jan 25 19:27 var
```

La commande **lxc-create** a renseigné le fichier de configuration et tous les répertoires. En particulier **/bin** avec l'exécutable **busybox** et les **applets** en liens symboliques :

```
bbox# ls -l /bin
total 2076
lrwxrwxrwx    1 root     root             7 Jan 25 19:27 [ -> busybox
lrwxrwxrwx    1 root     root             7 Jan 25 19:27 [[ -> busybox
lrwxrwxrwx    1 root     root             7 Jan 25 19:27 acpid -> busybox
lrwxrwxrwx    1 root     root             7 Jan 25 19:27 adjtimex -> busybox
lrwxrwxrwx    1 root     root             7 Jan 25 19:27 ar -> busybox
lrwxrwxrwx    1 root     root             7 Jan 25 19:27 arch -> busybox
lrwxrwxrwx    1 root     root             7 Jan 25 19:27 arp -> busybox
[...]
```

Mais ce serait un gaspillage de place de copier les nombreuses librairies généralement situées dans **/lib** ou **/usr/lib** (**libc**, **libc++**, …) dans les mêmes répertoires côté conteneur (i.e. Dans **/var/lib/lxc/bbox/rootfs/lib** et **/var/lib/lxc/bbox/rootfs/usr/lib**).

D'où l'utilisation du mécanisme de « bind mounting » **[4]** qui consiste à voir un répertoire ou un fichier donné à un autre endroit dans le système de fichiers. Il s'agit du drapeau **MS_BIND** (aperçu au paragraphe précédent) passé dans les options de l'appel système **mount()**. Pour un conteneur, il est possible de dire qu'on veut voir certains sous-arbres du hôte dans le **rootfs** à l'aide du mot clé « bind » dans l'option de configuration **lxc.mount.entry**. C'est un **mount()** avec **MS_BIND** qui se cache derrière.

La configuration LXC pour le conteneur **bbox** spécifie que les répertoires sur hôte comme **/lib** et quelques autres sont « bind-montés » dans son **rootfs** :

```
# cat /var/lib/lxc/bbox/config
[...]
lxc.mount.entry = /lib lib none ro,bind 0 0
lxc.mount.entry = /usr/lib usr/lib none ro,bind 0 0
lxc.mount.entry = /lib64 lib64 none ro,bind 0 0
lxc.mount.entry = /usr/lib64 usr/lib64 none ro,bind 0 0
lxc.mount.entry = /sys/kernel/security sys/kernel/security none ro,bind,optional 0 0
```

D'où la vision côté conteneur des fichiers du hôte dans ces répertoires. Par exemple, le premier **lxc.mount.entry** dans la copie d'écran précédente indique que **/lib** sur hôte est « bind-monté » sur **lib** dans le répertoire racine côté conteneur. On a aussi pris soin de spécifier le mode « lecture seulement » (i.e. **ro**) pour ne pas altérer les fichiers du hôte à partir du conteneur :

```
bbox# ls -l /lib
total 30412
[...]
-rwxr-xr-x    1 root     root         78784 Nov  7 06:15 klibc-KzNL5rI0ooqhK-koTVzHy10DW4w.so
```

```
drwxr-xr-x    2 root     root          4096 Oct 17 12:26 language-selector
lrwxrwxrwx    1 root     root            25 Sep 16 14:56 ld-linux.so.2 -> i386-linux-gnu/ld-2.30.so
lrwxrwxrwx    1 root     root            23 Aug  5  2019 libOpenColorIO.so.1 ->
libOpenColorIO.so.1.1.1
-rw-r--r--    1 root     root       1084984 Aug  5  2019 libOpenColorIO.so.1.1.1
[...]
```

Cela implique bien-sûr que toute modification des fichiers dans ces répertoires côté hôte (suite à une mise à jour système par exemple), sera répercutée de manière transparente dans le conteneur car le principe du « bind montage » fait que ce dernier a exactement la même vue sur le contenu de ces répertoires.

Dans **lxc-start**, les « bind-montages » ont lieu avant le **pivot_root()**, période pendant laquelle la commande a la vue sur le **rootfs** du hôte. Mais ces montages restent valides après car le manuel de **mount()** précise bien qu'ils peuvent traverser les différents systèmes de fichiers (les points de montage) et les **chroot()** (cela comprend les **pivot_root()**) :

```
$ man 2 mount
[...]
Creating a bind mount
       If mountflags includes MS_BIND (available since Linux 2.4), then perform a bind mount.
       A bind mount makes a file or a directory subtree  visible  at  another  point
       within the single directory hierarchy.  Bind mounts may cross filesystem boundaries
       and span chroot(2) jails.
[...]
```

La figure 2 représente les « bind-mounts » dans le conteneur **bbox**.

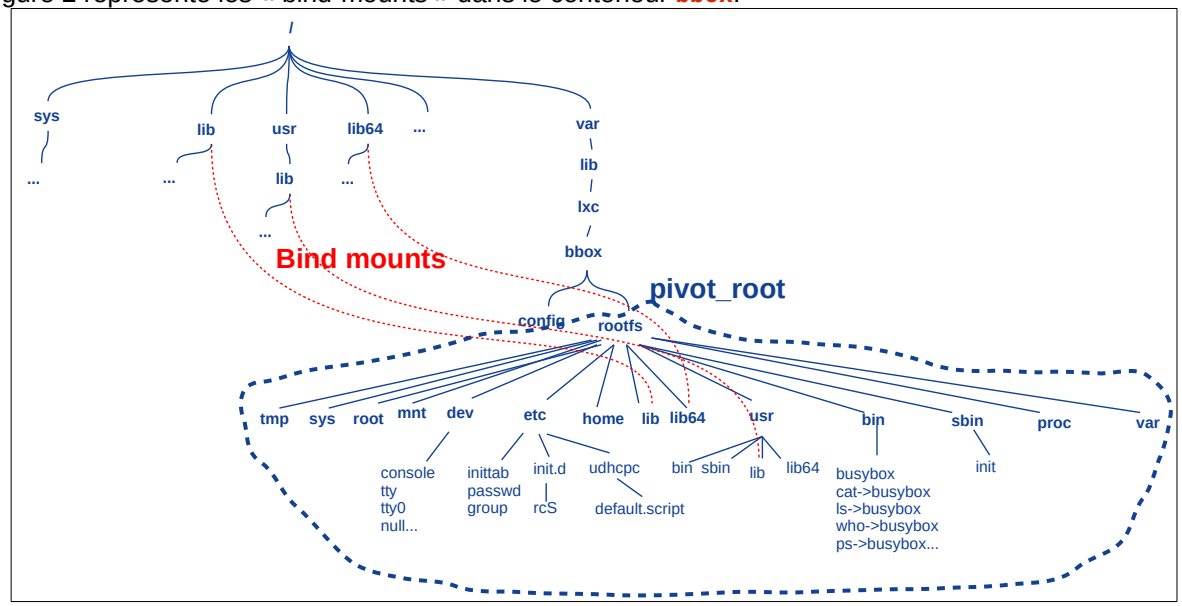


Fig. 2 : Bind-mounts dans le conteneur

En résumé, **le mécanisme de « bind-mount » donne la possibilité au système hôte de partager des parties de son système de fichiers avec les conteneurs**.

1.4 Les namespaces de la commande ip

Lors de l'étude de la commande **ip** **[10]** et notamment de son objet **netns**, nous avons vu qu'il est possible de créer de nouveau net_ns ou de s'attacher à des net_ns existant. Nous avons vaguement évoqué la notion de « bind montage » pour expliquer comment la commande s'est attachée au net_ns d'un conteneur LXC. Nous pouvons maintenant expliquer plus précisément le principe de la commande **ip netns attach <name> <pid>** utilisée. Le but de cette commande est d'attacher un nom **<name>** à un net_ns existant auquel un processus d'identifiant **<pid>** est associé (par exemple le processus **init** d'un conteneur) :

1. Création du répertoire **/run/netns** où seront créés les noms des namespaces au sens de la commande **ip** : **mkdir**("/run/netns", **S_IRWXU|S_IRGRP|S_IXGRP|S_IROTH|S_IXOTH**) ;

2. « Bind montage » du répertoire **/run/netns** sur lui-même afin d'en faire un point de montage : **mount**("/run/netns", "/run/netns", "none", **MS_BIND | MS_REC**, NULL) ;

2. Positionnement du type de propagation en **MS_SHARED** : **mount**("", "/run/netns", "none", MS_SHARED | MS_REC, NULL) ;

3. Création du fichier qui servira de nom au namespace au sens de la commande **ip** : **open**("/run/netns/**<name>**", **O_RDONLY|O_CREAT|O_EXCL**, 0) ;

4. « Bind montage » du lien symbolique du net_ns du processus cible sur le fichier précédent : **mount**("/proc/**<pid>**/ns/net", "/run/netns/**<name>**", "none", **MS_BIND**, NULL).

Pour la création d'un net_ns avec la commande **ip netns add name**, les étapes sont les mêmes mais avec en plus l'appel à **unshare()** pour créer le namespace et un « bind montage » de **/proc/self/ns/net** sur **/run/netns/<name>** :

[...]

4. **unshare**(**CLONE_NEWNET**) ;

5. **mount**("/proc/**self**/ns/net", "/run/netns/**<name>**", "none", **MS_BIND**, NULL).

Ainsi, la liste des net_ns auxquels la commande **ip** a associé un nom est la liste des fichiers dans le répertoire **/run/netns**. La commande **ip netns list** affiche le contenu de ce répertoire :

```
# ls -l /run/netns/
total 0
-r--r--r-- 1 root root 0 mars   23 13:13 bbox_nsnet
-r--r--r-- 1 root root 0 mars   23 10:07 net2
-r--r--r-- 1 root root 0 mars   23 10:41 other
# ip netns list
bbox_nsnet (id: 0)
other
net2
```

Les fichiers de ce répertoire sont donc des « bind montages » des cibles (dans le système de fichiers **NSFS** **[5]**) des liens symboliques **/proc/<pid>/ns/net** associés aux net_ns. On les voit dans le fichier **mountinfo** même si le noyau n'affiche pas explicitement le mot « bind » :

```
# cat proc/$$/mountinfo
[...]
1004 26 0:23 /netns /run/netns rw,nosuid,noexec,relatime shared:5 - tmpfs tmpfs
rw,size=1635172k,mode=755
1320 1004 0:4 net:[4026532586] /run/netns/net2 rw shared:661 - nsfs nsfs rw
1321 26 0:4 net:[4026532586] /run/netns/net2 rw shared:661 - nsfs nsfs rw
1374 1004 0:4 net:[4026533004] /run/netns/other rw shared:740 - nsfs nsfs rw
1375 26 0:4 net:[4026533004] /run/netns/other rw shared:740 - nsfs nsfs rw
508 1004 0:4 net:[4026533069] /run/netns/bbox_nsnet rw shared:284 - nsfs nsfs rw
509 26 0:4 net:[4026533069] /run/netns/bbox_nsnet rw shared:284 - nsfs nsfs rw
```

La figure 3 schématise le résultat des actions précédentes. La cible du fichier **/proc/<conteneur_init_pid>/ns/net** est « bind-monté » sur **/run/netns/bbox_nsnet**. Les deux autres namespaces (**other** et **net2**) ont été créés de toute pièce par des processus exécutant **ip netns add**. Ces deux processus ont disparu à la fin des commandes **ip** correspondantes mais le « bind-montage » empêche les net_ns de disparaître. La commande **ip netns del** démontera les fichiers et provoquera implicitement la disparition des net_ns non liés à un processus.

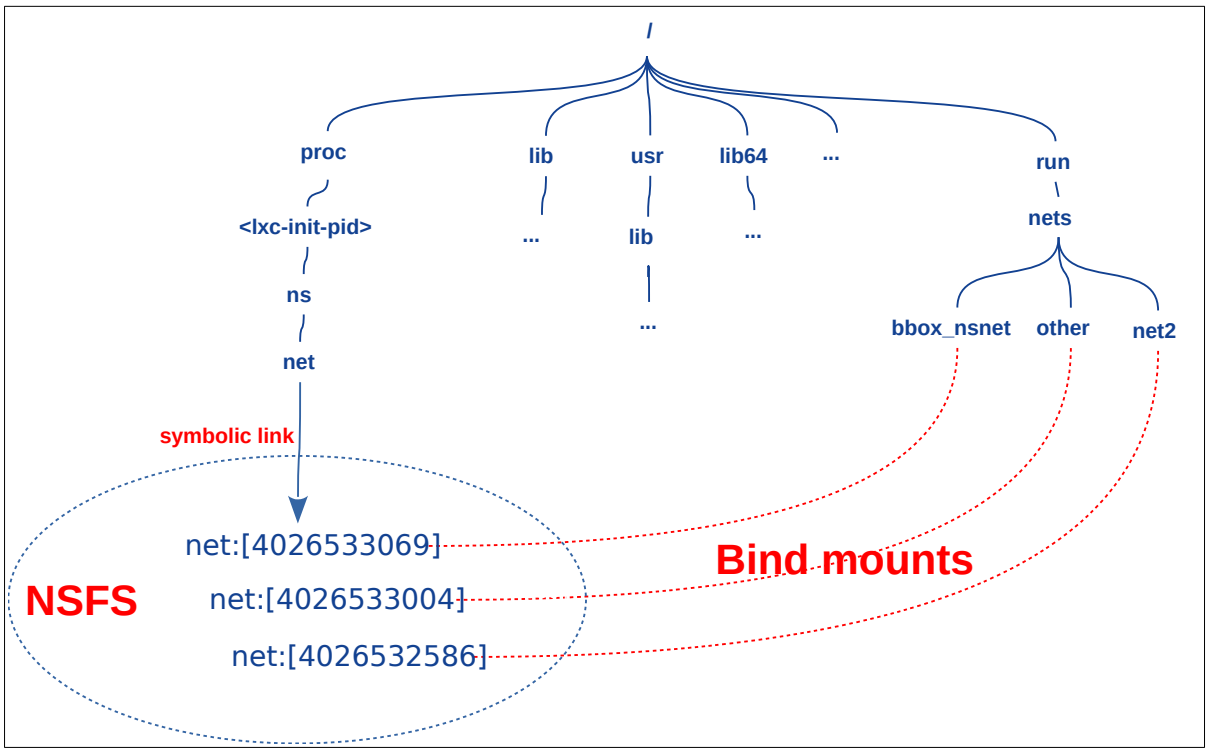


Fig. 3 : Les « bind montages » de la commande ip

Ainsi, pour effectuer des actions dans un net_ns auquel est associé un nom, la commande **ip** n'aura plus qu'à ouvrir le fichier associé dans **/run/netns**. Ce dernier étant un point de « bind montage » de la cible du lien symbolique du namespace, le descripteur de fichier qui en résulte peut être passé à l'appel système **setns()** pour entrer dans le namespace correspondant.

1.5 Emulation de terminal dans LXC

Lorsqu'un conteneur émule un système Linux minimum sans interface graphique, il peut mettre en oeuvre des terminaux physique (**tty**) sur lesquels tournent des processus **getty** pour gérer les connexions des utilisateurs. Lorsqu'un utilisateur saisit son nom d'utilisateur, **getty** exécute le programme **login** pour gérer la saisie du mot de passe et exécuter le shell associé à l'utilisateur dans le fichier **/etc/passwd**.

Dans un conteneur LXC, le nombre de terminaux est configurable avec le paramètre **lxc.tty.max**. Par exemple, le conteneur **busybox** a un terminal :

```
# cat /var/lib/lxc/bbox/config
[...]
lxc.tty.max = 1
[...]
```

Le fichier **/etc/inittab** est configuré pour demander au processus **init** de lancer un **getty** sur ce terminal :

```
# lxc-console -n bbox -t 0
bbox# cat /etc/inittab
::sysinit:/etc/init.d/rcS
tty1::respawn:/bin/getty -L tty1 115200 vt100
console::askfirst:/bin/sh
bbox# ps
PID   USER     COMMAND
    1 root     init
    4 root     /bin/syslogd
   14 root     /bin/udhcpc
   15 root     /bin/getty -L tty1 115200 vt100
   16 root     /bin/sh
   18 root     {ps} /bin/sh
```

Pour se connecter au terminal, il suffit de spécifier son numéro sur la ligne de commande de **lxc-console** (dans la copie d'écran précédente, on s'est connecté sur la console de numéro 0) et utiliser **root** comme nom d'utilisateur (défini par défaut par le template passé à **lxc-create**). Mais comme son mot de passe

n'est pas défini, il faut lui en attribuer un via la console :

```
bbox# passwd
passwd: no record of root in /etc/shadow, using /etc/passwd
Changing password for root
New password:
Retype password:
passwd: password for root changed by root
```

Ensuite on peut se connecter via le terminal numéro 1 :

```
# lxc-console -n bbox -t 1

Connected to tty 1
Type <Ctrl+a q> to exit the console, <Ctrl+a Ctrl+a> to enter Ctrl+a itself

bbox login: root
Password:

BusyBox v1.30.1 (Ubuntu 1:1.30.1-4ubuntu4) built-in shell (ash)
Enter 'help' for a list of built-in commands.

~ #
```

Dans les faits, un conteneur n'a pas de périphérique terminal car ces derniers sont attribués au système hôte. Pour pallier ce manque, LXC monte un système de fichiers **devpts** dans le conteneur (dimensionné avec un nombre de **pty** spécifié par le paramètre de configuration **lxc.pty.max**) et effectue un « bind montage » des pseudo-terminaux **/dev/pts/<x>** (**pty**) sur le fichier **/dev/tty<x+1>**. Par exemple :

```
mount("/dev/pts/0", "/dev/tty1", 0x7f0c146fc13f, MS_BIND, NULL)
```

Pour la console du conteneur, c'est aussi un « bind montage » mais c'est fait avant le **pivot_root()** et avec le premier pseudo-terminal libre sur hôte au moment du démarrage du conteneur. Par exemple, si le premier **pty** libre est le numéro 6 lors de l'appel à **lxc-start**, LXC va effectuer le montage suivant sur hôte :

```
mount("/dev/pts/6", "/var/lib/lxc/bbox/rootfs/dev/console", 0x7f0c146fc13f, MS_BIND, NULL)
```

Ce montage est persistant après le **pivot_root()**.

Le fichier **mountinfo** dans le conteneur se présente comme suit pour les terminaux :

```
bbox# cat /proc/1/mountinfo
[...]
609 586 0:22 /6 /dev/console rw,nosuid,noexec,relatime master:3 - devpts devpts
rw,gid=5,mode=620,ptmxmode=000
497 586 0:63 / /dev/pts rw,relatime - devpts devpts rw,gid=5,mode=620,ptmxmode=666,max=1
[...]
505 586 0:63 /0 /dev/tty1 rw,relatime - devpts devpts rw,gid=5,mode=620,ptmxmode=666,max=1
```

Le fichier **/dev/console** est « bind monté » sur **/dev/pts/6** (côté hôte !), le fichier **/dev/tty1** est « bind monté » sur **/dev/pts/0** (côté conteneur !) dans le système de fichier **devpts** monté à cet effet (dimensionné par le paramètre de configuration **lxc.pty.max**).

Nous n'allons pas entrer plus que cela dans les détails de la gestion des terminaux dans LXC sous peine de nous égarer. Nous nous contentons de dire que **lxc-console** se connecte aux terminaux ainsi configuré via le superviseur du conteneur (processus **[lxc monitor]**) qui garde en interne des descripteurs de fichiers ouverts sur les terminaux du conteneur. L'option **-t** permet de choisir le terminal sur lequel on se connecte (0 pour la console, 1 pour le tty1 et ainsi de suite). La figure 4 schématise la configuration du conteneur **bbox** :

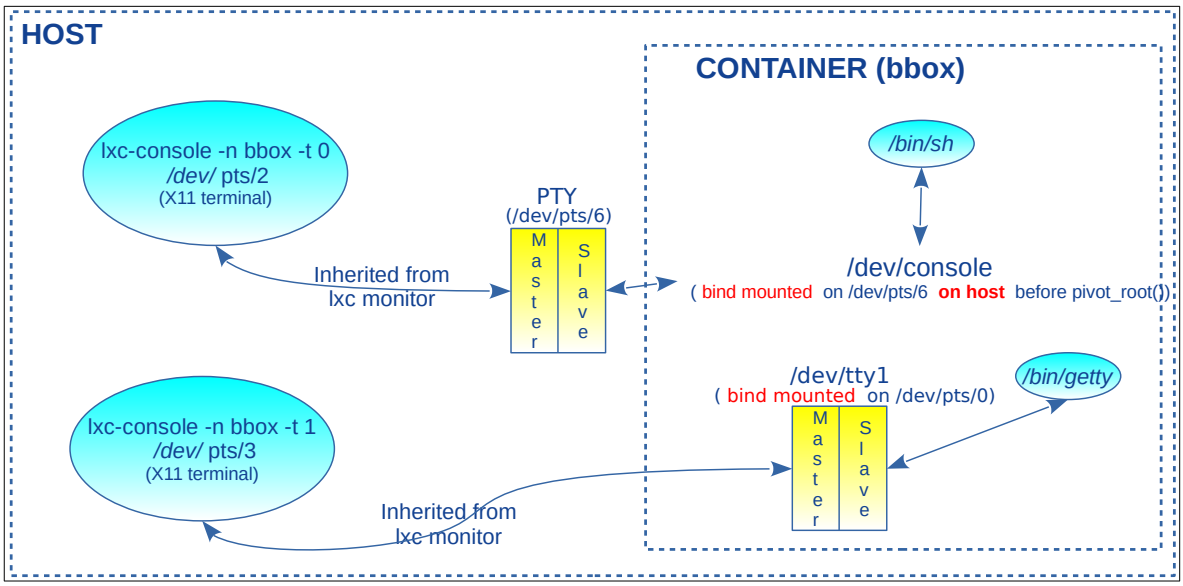


Fig. 4 : Les terminaux de LXC

1.6 Montage hors namespace initial

Comme nous l'avons vu, un namespace est la propriété d'un user_ns. Pour des raisons de sécurité, si le user_ns n'est pas initial, il y a des restrictions sur certaines opérations et en particulier les montages de systèmes de fichiers **[6]** même si l'utilisateur est privilégié dans le namespace.

Dans le noyau de Linux, un drapeau a été introduit dans le fichier **include/linux/fs.h** afin d'indiquer si un système de fichier peut être monté ou pas dans un user_ns non initial :

```
#define FS_USERNS_MOUNT               8       /* Can be mounted by userns root */
```

L'appel système **mount()** défini dans **fs/namespace.c**, aboutit à l'appel d'une fonction de service interne **mount_capable()** localisée dans **fs/super.c** :

```
bool mount_capable(struct fs_context *fc)
{
        if (!(fc->fs_type->fs_flags & FS_USERNS_MOUNT))
                return capable(CAP_SYS_ADMIN);
        else
                return ns_capable(fc->user_ns, CAP_SYS_ADMIN);
}
```

La fonction est chargée de déterminer si l'utilisateur appelant a la capacité **CAP_SYS_ADMIN** pour effectuer l'opération de montage :

- Si le système de fichiers à monter n'a pas le drapeau **FS_USERNS_MOUNT**, la fonction vérifie la capacité dans le user_ns initial. Autrement dit, si l'appelant n'est pas le super utilisateur au démarrage du système, il n'aura en général pas le droit de faire le montage ;

- Si le système de fichiers à monter a le drapeau **FS_USERNS_MOUNT**, la fonction vérifie la capacité dans le user_ns courant de l'utilisateur. Dans ce cas, il suffira à l'appelant d'être super utilisateur dans ce user_ns pour avoir le droit d'effectuer le montage (et même s'il correspond à un utilisateur ordinaire dans le user_ns supérieur via la fonction de mapping vue dans l'étude des user_ns !).

Au départ assez restreinte, une recherche rapide dans les sources du noyau Linux 5.3, montre que la liste des systèmes de fichiers autorisés à être montés dans les user_ns non initiaux s'est étoffée. Nous pouvons citer entre autres : **cgroup**, **cgroup2**, **cpuset**, **proc**, **ramfs**, **devpts**, **ext2**, **ext3**, **ext4**, **fuse**, **aufs**, **sysfs**, **shiftfs**, **overlay**, **mqueue**, **binder** et **tmpfs**.

2 Le namespace UTS

Ce namespace isole deux identifiants : le nom de nœud et le nom de domaine. Ce sont respectivement les champs **nodename** et **domainname** de la structure **utsname** renseignée par l'appel système **uname()**. Ces

informations peuvent être modifiées par les services **sethostname()** et **setdomainname()**. Le nom « uts » n'est pas très explicite voire trompeur car il est l'acronyme de « **U**nix **T**ime **S**haring ». C'est une réminiscence du système Unix reconduite dans Linux. Dans le code source du noyau, nous la retrouvons pour tout ce qui touche à l'identification du système : **UTS_RELEASE**, **UTS_VERSION**, **UTS_MACHINE**...

Le nom de domaine trouvait son utilité dans le service **NIS** (auparavant **Yellow pages**) de la société Sun Microsystem **[7]** mais ce service a été supplanté par **LDAP** et **DNS**. Il n'est plus très utile de nos jours. Par contre, le nom de nœud (ou machine) est encore très usité. Il sert d'alias pour l'adresse internet de la machine.

Nous avons par exemple vu dans **[8]** le script template **lxc-busybox** utilise le nom du conteneur (option **-n** des commandes) comme nom de machine (paramètre de configuration **lxc.uts.name**). Le fichier de configuration est en général visible dans **/var/lib/lxc/<nom_de_conteneur>/config** :

```
# cat /var/lib/lxc/bbox/config
[...]
lxc.uts.name = "bbox"
[...]
```

Le programme d'exemple **setnshost** reçoit en paramètres l'identifiant et un nom de machine à positionner[8] dans l'uts_ns d'un processus cible. L'outil migre dans le namespace en invoquant **setns()** avec le descripteur de la cible du lien symbolique correspondant (i.e. **/proc/<pid_processus_cible/ns/uts**) puis appelle **sethostname()** avec le nom demandé :

```
int main(int ac, char *av[])
{
int          fd, rc;
char         tpath[256];
[...]
  // Build the pathname of the uts_ns
  snprintf(tpath, sizeof(tpath), "/proc/%s/ns/uts", av[1]);

  // Open the target uts_ns symbolic link
  fd = open(tpath, O_RDONLY);

  // Enter into the target namespace
  rc = setns(fd, CLONE_NEWUTS);
[...]
  close(fd);

  rc = sethostname(av[2], strlen(av[2]));
[...]
  return 0;
} // main
```

On peut ainsi modifier le nom de machine d'un conteneur en passant le pid de son processus **init** à **setnshost**. Vérifions tout d'abord le nom de machine courant dans le conteneur **bbox** :

```
# lxc-console -n bbox -t 0
[...]
bbox# hostname
bbox
```

Dans un autre terminal, renommons le nom de hôte du conteneur en **foo** (à ne pas confondre avec le nom du conteneur qui restera **bbox** !) en prenant soin de vérifier avant et après que le nom côté hôte (ici c'est **rachid-pc**) ne change pas :

```
# ./lxc-pid bbox
4338
# hostname
rachid-pc
# ./setnshost 4338 foo
# hostname
rachid-pc
```

Dans le conteneur, affichons le nom de hôte pour vérifier la modification (un appui sur **<RETURN>** convient aussi vu que le nom de host est affiché dans le prompt) :

```
bbox#
foo# hostname
foo
```

Toutes les informations de la structure **new_utsname**, vue lors de létude des namespaces dans le noyau **[9]**, sont exportées en espace utilisateur via **/proc/sys/kernel** :

```
$ ls -l /proc/sys/kernel
[...]
```

```
-rw-r--r-- 1 root root 0 avril  10 08:24 domainname
[...]
-rw-r--r-- 1 root root 0 avril  10 08:24 hostname
[...]
-r--r--r-- 1 root root 0 avril  10 08:24 osrelease
-r--r--r-- 1 root root 0 avril  10 11:49 ostype
[...]
-r--r--r-- 1 root root 0 avril  10 11:49 version
[...]
```

L'implémentation est dans le fichier **kernel/utsname_sysctl.c** :

```
static struct ctl_table uts_kern_table[] = {
	{
		.procname	= "ostype",
		.data		= init_uts_ns.name.sysname,
		.maxlen		= sizeof(init_uts_ns.name.sysname),
		.mode		= 0444,
		.proc_handler	= proc_do_uts_string,
	},
	{
		.procname	= "osrelease",
		.data		= init_uts_ns.name.release,
		.maxlen		= sizeof(init_uts_ns.name.release),
		.mode		= 0444,
		.proc_handler	= proc_do_uts_string,
	},
	{
		.procname	= "version",
		.data		= init_uts_ns.name.version,
		.maxlen		= sizeof(init_uts_ns.name.version),
		.mode		= 0444,
		.proc_handler	= proc_do_uts_string,
	},
	{
		.procname	= "hostname",
		.data		= init_uts_ns.name.nodename,
		.maxlen		= sizeof(init_uts_ns.name.nodename),
		.mode		= 0644,
		.proc_handler	= proc_do_uts_string,
		.poll		= &hostname_poll,
	},
	{
		.procname	= "domainname",
		.data		= init_uts_ns.name.domainname,
		.maxlen		= sizeof(init_uts_ns.name.domainname),
		.mode		= 0644,
		.proc_handler	= proc_do_uts_string,
		.poll		= &domainname_poll,
	},
	{}
};
```

Tous les champs sont accessibles en lecture (**mode** égal à 0444) et deux (**hostname** et **domainname**) sont en plus modifiables par le super utilisateur (**mode** égal à 0644). L'accès à ces fichiers déclenche la fonction **proc_do_uts_string()**. Cette dernière obtient l'adresse du champ désiré de la structure **new_utsname** de l'uts_ns de la tâche courante en appelant la fonction **get_uts()** :

```
static void *get_uts(struct ctl_table *table)
{
  char *which = table->data; // @ of the field in init_uts_ns (got from  uts_kern_table[])
  struct uts_namespace *uts_ns;

 uts_ns = current->nsproxy->uts_ns; // @uts_ns of the current task (current uts_ns)
  which = (which - (char *)&init_uts_ns) + (char *)uts_ns; // @field in new_utsname of the current
uts_ns

  return which;
}
```

Il est par conséquent possible de consulter le fichier **/proc/sys/kernel/hostname** du côté hôte pour avoir la valeur dans le namespace initial :

```
$ cat /proc/sys/kernel/hostname
rachid-pc
```

Tandis que côté conteneur, l'affichage correspond à ce qui a été modifié par notre programme :

```
foo# cat /proc/sys/kernel/hostname
```

```
foo
```

Les valeurs de **domainname** et **hostname** sont ainsi consultables et modifiables dans **/proc** au lieu de passer par les appels système.

Le bémol de cette seconde méthode tient au fait que **/proc/sys** est un répertoire sensible du point de vue des données qu'il exporte (nous le soulignerons aussi lors de l'étude des ipc_ns). Dans le contexte d'un conteneur, **/proc/sys** est souvent monté en lecture seule pour raisons de sécurité. La modification directe des fichiers **/proc/sys/kernel/{hostname,domainname}** est donc en général impossible. D'où l'intérêt de passer par les appels système **sethostname()** et **setdomainname()** qui modifient les champs **nodename** et **domainname** dans la structure **new_utsname** pointée par le **nsproxy** de la tâche appelante sans avoir à passer par le système de fichiers. Le code source de **sethostname()** se présente en effet comme suit dans le fichier **kernel/sys.c** du noyau :

```
SYSCALL_DEFINE2(sethostname, char __user *, name, int, len)
{
        int errno;
        char tmp[__NEW_UTS_LEN];

        if (!ns_capable(current->nsproxy->uts_ns->user_ns, CAP_SYS_ADMIN))
                return -EPERM;
[...]
        errno = -EFAULT;
        if (!copy_from_user(tmp, name, len)) {
                struct new_utsname *u;

                down_write(&uts_sem);
                u = utsname();
                memcpy(u->nodename, tmp, len);
                memset(u->nodename + len, 0, sizeof(u->nodename) - len);
                errno = 0;
                uts_proc_notify(UTS_PROC_HOSTNAME);
                up_write(&uts_sem);
        }
        return errno;
}
```

La fonction interne **utsname()**, définie dans **include/linux/utsname.h**, retourne un pointeur sur la structure **new_utsname** référencée par le **nsproxy** de la tâche courante :

```
static inline struct new_utsname *utsname(void)
{
        return &current->nsproxy->uts_ns->name;
}
```

En résumé, **sethostname()** et **setdomainname()** bâti sur le même algorithme, modifient respectivement les champs **current->nsproxy->uts_ns->name.nodename** et **current->nsproxy->uts_ns->name.domainname** après avoir vérifié que l'utilisateur a la capacité **CAP_SYS_ADMIN** (apanage du super utilisateur en général).

Conclusion

En donnant la possibilité de voir et isoler certaines parties de l'arborescence des fichiers du système grâce aux différents types de propagation et le mécanisme de « bind montage », les mount_ns permettent d'optimiser au mieux l'emprunte et la sécurité du système de fichiers des conteneurs.

L'uts_ns, le plus simple des namespaces, permet de nommer les conteneurs.

Références

[1] Les sous-arbres partagés : https://www.kernel.org/doc/Documentation/filesystems/sharedsubtree.txt

[2] Mount namespaces and shared subtrees : https://lwn.net/Articles/689856/

[3] Mount namespaces, mount propagation, and unbindable mounts : https://lwn.net/Articles/690679/

[4] Mount namespaces and shared subtrees : https://lwn.net/Articles/689856/

[5] R. Koucha, « Le fonctionnement des namespaces dans le noyau », GNU/Linux magazine n°245, février 2021 : https://connect.ed-diamond.com/GNU-Linux-Magazine/glmf-245/le-fonctionnement-des-namespaces-dans-le-noyau

[6] Filesystem mounts in user namespaces : https://lwn.net/Articles/652468/

[7] Network Information Service : https://en.wikipedia.org/wiki/Network_Information_Service

[8] R. Koucha, « Les namespaces ou l'art de se démultiplier », GNU/Linux magazine n°239, juillet/août 2020 : https://connect.ed-diamond.com/GNU-Linux-Magazine/glmf-239/les-namespaces-ou-l-art-de-se-demultiplier

[9] R. Koucha, « Les structures de données des namespaces dans le noyau », GNU/Linux magazine n°239, juillet/août 2020 : https://connect.ed-diamond.com/GNU-Linux-Magazine/glmf-243/les-structures-de-donnees-des-namespaces-dans-le-noyau

[10] R. Koucha, « Les utilitaires relatifs aux namespaces », GNU/Linux magazine n°240, septembre 2020 : https://connect.ed-diamond.com/GNU-Linux-Magazine/glmf-240/les-utilitaires-relatifs-aux-namespaces