Les namespaces ou l’art de se démultiplier

Rachid Koucha
[Ingénieur développement logiciel]

Notion indispensable aux mécanismes d’isolation mais plutôt méconnus du grand public, les namespaces sont devenus incontournables dans l’environnement Linux. Ils sont, le plus souvent, utilisés de manière implicite à travers les gestionnaires de conteneurs tels que LXC.

La notion de namespace **[1]** est apparue dans le système expérimental `Plan 9`. Développé par Bell Labs sous la direction de R. Pike et K. Thompson à partir de 1987, ce projet était destiné à pallier les problèmes du système Unix **[2]**.

Plan 9

Le nom « plan 9 » est inspiré du titre “Plan 9 from Outer Space”, film de science-fiction américain réalisé par Edward D. Wood et sorti sur les écrans en 1959 **[3]**. Classé dans la catégorie des nanars, il est devenu « collector ». Ce metteur en scène fantasque est d’ailleurs le personnage éponyme du biopic tourné par Tim Burton **[4]**.

Bien que le concept existait avant son apparition en 1991, le système Linux s’est d’abord cantonné dans son statut de « Unix-like » en proposant un fonctionnement identique à son modèle. L’appel système `chroot()` ainsi que la mémoire virtuelle des processus était alors les seuls mécanismes d’isolation. Linux a proposé un premier namespace dans sa version 2.4.19 en 2002 : le « namespace mount » **[5]**. Il n’était pas prévu d’en proposer d’autres mais afin de répondre aux besoins grandissants de la virtualisation des ressources informatiques, de nouveaux namespaces se sont avérés nécessaires et sont apparus à partir de 2006. Linux en compte sept actuellement. Et il est prévu d’en proposer de nouveaux dans les versions à venir.

Dans cette série d’articles, il n’est pas question de paraphraser ce qui se trouve dans le manuel de Linux ou les différents articles disponibles ici et là sur le Web. Nous allons plutôt passer en revue tous les namespaces et ce qui s’y rattache de manière pratique (voire ludique) à travers les outils disponibles, des

programmes écrits en shell et langage **C** et une petite plongée dans le code source du noyau. Ce sera l'occasion de combler quelques lacunes de la documentation, de comprendre certaines limitations voire même de pointer du doigt des faiblesses de l'implémentation. Nous nous accorderons aussi quelques digressions sur le gestionnaire de conteneurs **LXC [6]**, qui ne pourrait pas exister sans les namespaces.

Notes

- 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°239 de juillet/août 2020 :

1 Premier aperçu

1.1 Définition

A l'origine Linux, « Unix-like » par excellence, définissait des espaces de noms uniques pour différentes ressources système. Dans le jargon informatique, ils sont nommés « singletons ». Par exemple, les processus se voient attribuer un identifiant unique et global au système (pid). Il en est de même pour les utilisateurs (uid) et les groupes d'utilisateurs (gid). Avec l'avènement des namespaces (c.-à-d. « espaces de noms » en français), il est désormais possible d'instancier ces ensembles. L'unicité des ressources est ainsi garantie au sein d'un espace donné mais pas forcément d'un espace à l'autre. **Linux propose actuellement sept types de namespaces [7]** :

- Les points de montage du système de fichiers : « **mount namespaces** » (mount_ns) ;

- Les identifiants de processus : « **pid namespaces** »(pid_ns) ;

- Les noms de machine et de domaine : « **uts namespaces** » (uts_ns) ;

- Les ressources réseau (pile, protocoles, tables de routage, interfaces…) : « **network namespaces** » (net_ns) ;

- L'arborescence des cgroups : « **cgroup namespaces** » (cgroup_ns) ;

- Les identifiants de sécurité (identifiants d'utilisateur, identifiants de groupe…) : « **user namespaces** » (user_ns) ;

- Les IPC système V ainsi que les queues de message POSIX : « **ipc namespaces** » (ipc_ns).

Nous verrons par la suite que de nouveaux namespaces sont à l'étude ou apparaîtront dans les prochaines moutures de Linux.

Un processus est toujours associé à un namespace de chaque catégorie (cf. figure 1).

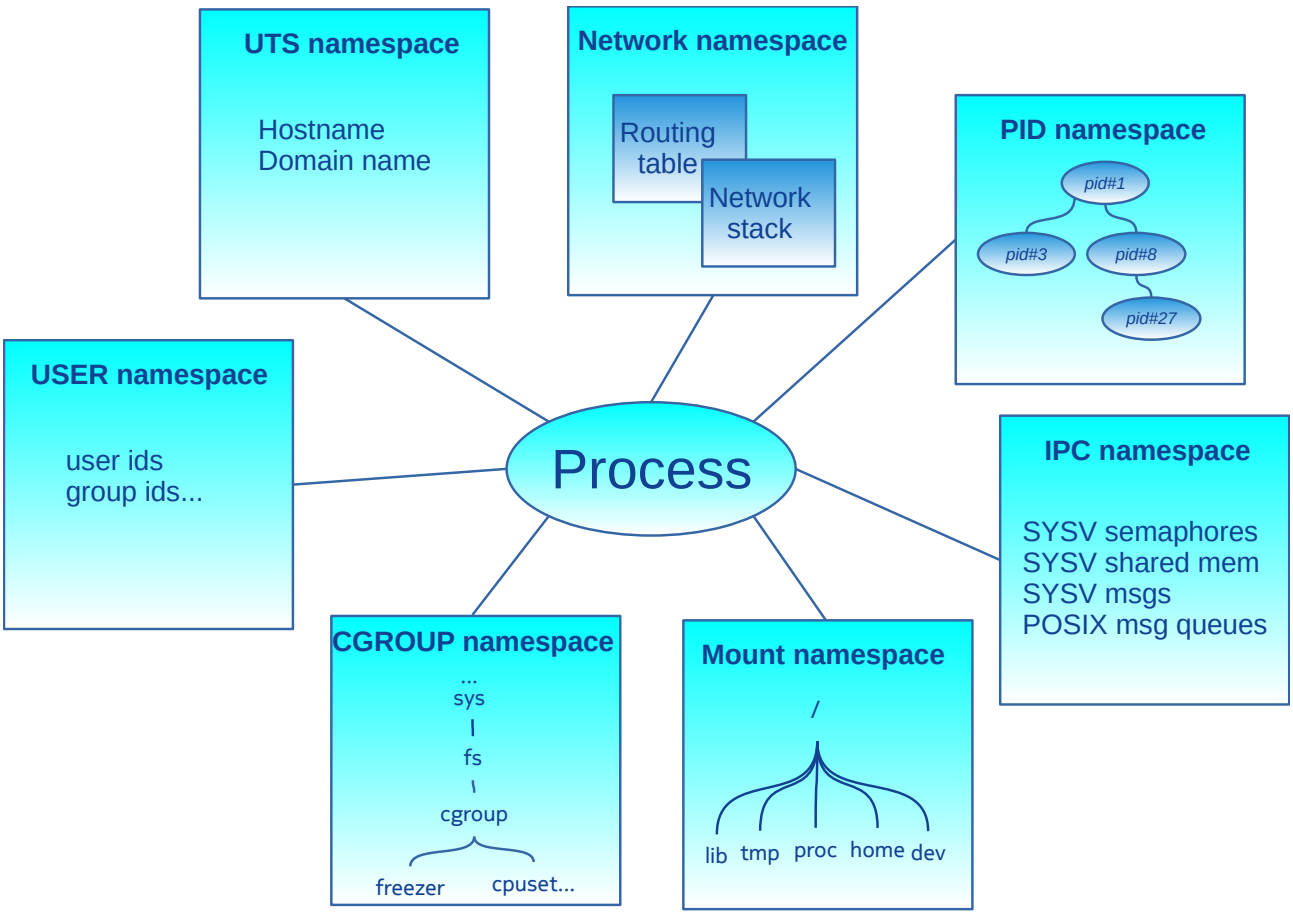

Fig. 1 : Les namespaces d'un processus

Un processus peut créer de nouveaux namespaces ou migrer d'un namespace à l'autre pour chaque catégorie.

Certains namespaces sont en plus hiérarchiques dans le sens où il y a une relation père-fils (c'est le cas des user_ns et pid_ns).

Une même ressource peut être visible dans un seul namespace (p. ex. un sémaphore) ou plusieurs namespaces, mais pas avec le même identifiant (p. ex. un processus a un pid dans tous les pid_ns en remontant de son pid_ns courant au pid_ns racine de la hiérarchie). La figure 2 illustre le cas d'un processus associé à un pid_ns fils. Il est identifié avec le pid 3 dans le pid_ns fils et avec le pid 13 dans le namespace parent.

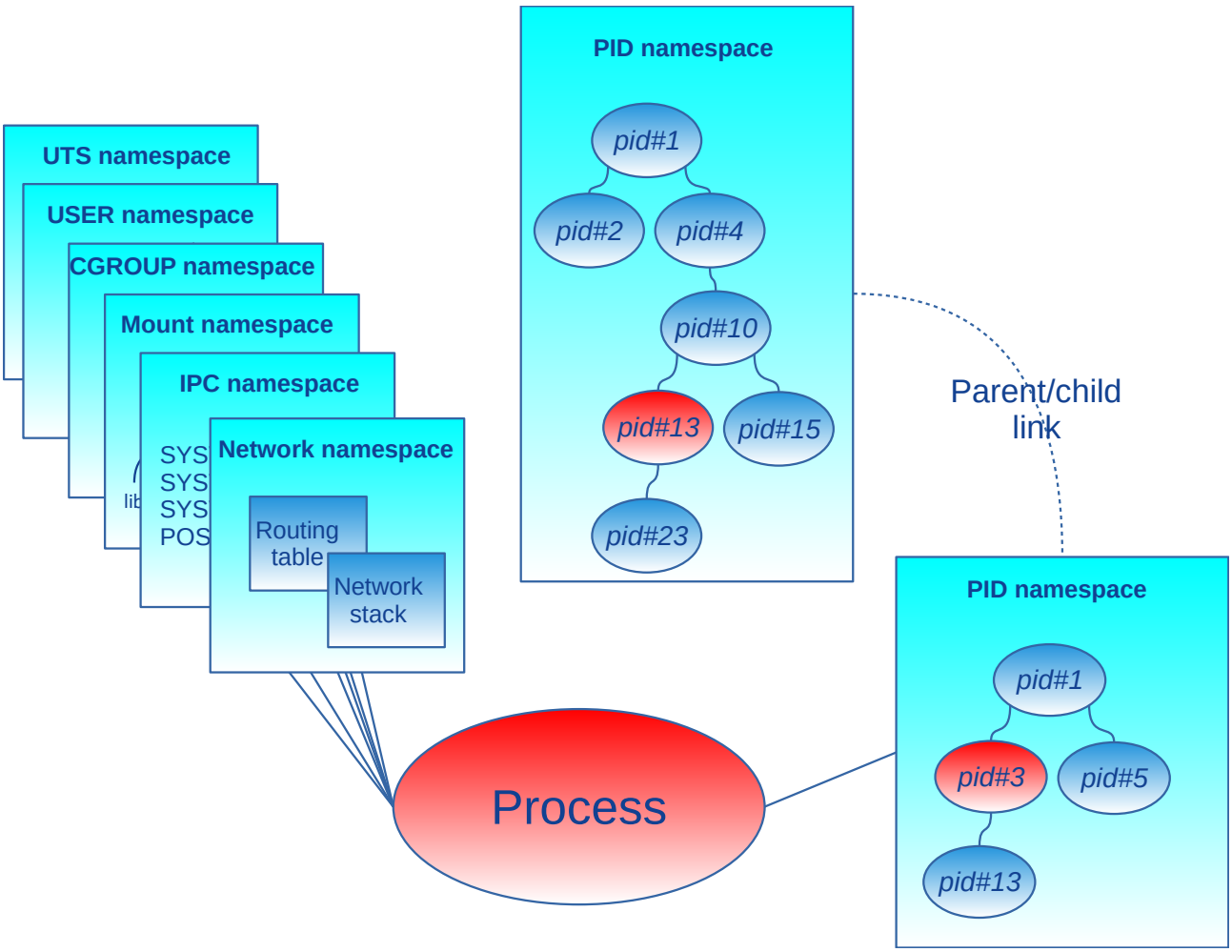

Fig. 2 : Processus vu de différents pid_ns

1.2 Besoin d'isolation

En guise de premier aperçu de l'utilité des namespaces, considérons un cas simple de développement et test d'un logiciel. Un système hôte est dédié aux tests unitaires. Parmi les ressources système dont le logiciel a besoin, il y a des sémaphores **Système V**. Le logiciel a une ampleur telle que différentes équipes de développeurs interviennent en même temps sur son code source pour la maintenance et les évolutions. Il faut le tester pour détecter les régressions et vérifier que les fonctionnalités ajoutées ont les comportements souhaités. Voici par exemple le programme **sem** qui crée un sémaphore avec la clé d'identification **0x12345** :

```
int main(void)
{
[...]
  rc = semget(0x12345, 1, IPC_CREAT|IPC_EXCL);
[...]
  printf("Semaphore id is: %d\n", rc);

  pause();

  return 0;

} // main
```

Un sémaphore étant identifié par une clé unique (cf. **man 2 semget**), il n'est pas possible de créer deux sémaphores avec une même clé sur une même machine. Si le programme précédent est lancé deux fois, une erreur est affichée lors de la seconde tentative d'exécution :

```
$ ./sem &
[2] 7235
Semaphore id is: 2
$ ./sem
semget(): 'File exists' (17)
```

Les tests du logiciel par les testeurs ne peuvent donc pas se faire en parallèle à moins d'utiliser des machines différentes. Pour des raisons de coûts notamment, il est difficile de multiplier le nombre de

machines. Dans un tel contexte, les équipes sont condamnées à lancer les tests séquentiellement. Cela va à l'encontre des principes de l'intégration continue **[8]** (c.-à-d. **CI**) destinée à augmenter l'efficacité de la mise au point et la livraison des corrections et nouvelles fonctionnalités. Le fameux « time to market » cher aux commerciaux, s'en voit fortement impacté.

Les solutions au problème ne manquent pas. Il est par exemple possible de configurer la valeur de la clé globalement (via une variable d'environnement par exemple) pour lui donner une valeur différente à chaque exécution ou d'attribuer des plages de valeurs autorisées aux différentes équipes. Peu importe les choix, on est dans une situation où il faut coordonner plusieurs intervenants sur une même machine. C'est une source de conflits en tout genre. Les collègues deviennent des belligérants.

C'est typiquement l'un des contextes où la notion de namespace a sa place. Les tests peuvent être mis en oeuvre en parallèle sur la même machine à condition d'utiliser des namespaces différents. Dans cet exemple, chaque test aura au minimum son propre ipc_ns où seront créés les sémaphores. Il n'y aura plus d'interférences entre les tests. Pour s'en convaincre, utilisons la commande **unshare** (détaillée plus tard) pour relancer chaque instance du programme précédent dans un ipc_ns dédié (option **-i** pour IPC). La manipulation de la majorité des namespaces, nécessite les droits du super utilisateur :

```
# unshare -i ./sem &
[1] 9365
Semaphore id is: 0
# unshare -i ./sem &
[2] 9366
Semaphore id is: 0
```

Nous constatons que la création sans erreur des sémaphores alors que les clés sont identiques. La commande **lsns** (détaillée plus tard), permet d'afficher la liste des namespaces auxquels un processus (option **-p**) est attaché. La première colonne est l'identifiant de namespace. Nous voyons que les deux processus évoluent dans les mêmes namespaces sauf pour les ipc_ns :

```
# lsns -p 9365
        NS TYPE   NPROCS   PID USER COMMAND
4026531835 cgroup    312     1 root /sbin/init splash
[...]
4026531992 net       306     1 root /sbin/init splash
4026532575 ipc         1  9365 root ./sem
# lsns -p 9366
        NS TYPE   NPROCS   PID USER COMMAND
4026531835 cgroup    312     1 root /sbin/init splash
[...]
4026531992 net       306     1 root /sbin/init splash
4026532703 ipc         1  9366 root ./sem
```

De manière plus générale, nous isolons au sein de namespaces les ressources nécessaires aux logiciels à exécuter. Dans les namespaces, tout se passe comme si nous étions seuls à utiliser la machine. On ne se pose plus la question des interférences éventuelles avec d'autres utilisateurs. **La notion de conteneur** est une application de ce principe.

1.3 Les conteneurs

Un conteneur est un ensemble de namespaces (un par catégorie) auxquels sont associés un ou plusieurs processus. Les ressources sont uniques dans un conteneur donné car elles sont allouées dans un namespace mais elles sont instanciées au niveau du système car il peut y avoir plusieurs namespaces pour chaque catégorie de ressource. **Le système d'exploitation est par contre commun** à tous les namespaces donc à tous les conteneurs. Au démarrage du système tous les processus sont associés aux **namespaces initiaux**. Dans un environnement de conteneurs, ces derniers constituent ce qu'on appelle le système hôte. Ils ne sont donc jamais désalloués. La figure 3 schématise un premier processus associé à des namespaces initiaux (uts_ns et net_ns) et non initiaux. Le second processus s'exécute au sein d'un conteneur car il est associé aux namespaces de ce dernier.

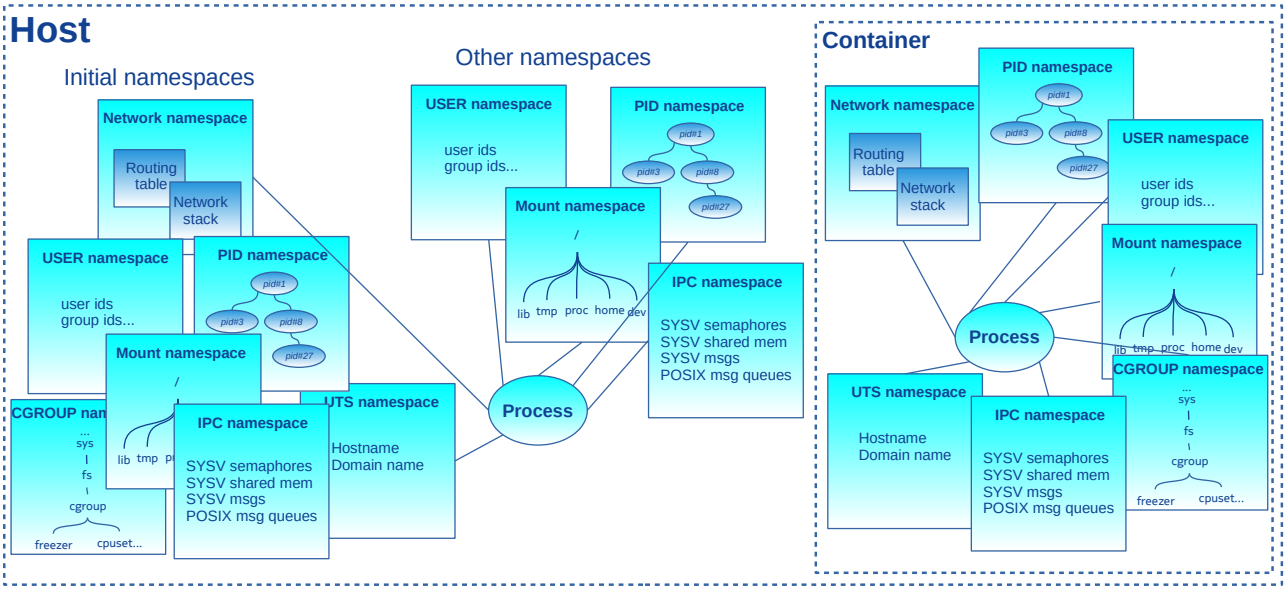

Fig. 3 : Les namespaces

1.4 Identification d'un conteneur

Un conteneur LXC est identifié par un nom. Il est passé avec l'option **`-n`** sur la ligne de commande de la plupart de ses outils. D'ailleurs dans les manuels en ligne, l'option figure dans la partie « COMMON OPTIONS ». Par exemple, en consultant le manuel de **`lxc-destroy`** (destruction d'un conteneur) :

```
$ man lxc-destroy
[...]
COMMON OPTIONS
[...]
       -n, --name=NAME
              Use container identifier NAME.  The container identifier format is an alphanumeric
string.
[...]
```

En interne, LXC référence un conteneur avec l'identifiant du premier processus lancé en son sein. Il est nommé **`init`** quand il s'agit d'un conteneur de type **`busybox`** **[9]** que nous utiliserons régulièrement pour étayer les propos de cette série d'articles. La mise en oeuvre d'un tel conteneur est présentée dans l'encadré « Mini-guide LXC ».

Mini-guide LXC

Ces articles utilisent la **version 2.1.1** de **LXC**.

Pour créer et démarrer un conteneur LXC simple nommé arbitrairement **bbox** et basé sur le template **busybox** (avec les droits du super utilisateur !) :

```
# lxc-create -n bbox -t busybox
# lxc-ls -f
NAME    STATE   AUTOSTART GROUPS IPV4 IPV6
bbox    STOPPED 0         -      -    -
# lxc-start -n bbox -s lxc.environment=PS1=\\h\#\\x20
# lxc-ls -f
NAME    STATE   AUTOSTART GROUPS IPV4       IPV6
bbox    RUNNING 0         -      10.0.3.203 -
```

L'option **-s** est facultative mais on l'utilise pour positionner la variable d'environnement **PS1** qui spécifie le format du **prompt** afin de différencier les copies d'écran qui illustrent les articles. Le script **lxc-start2** mis à disposition effectue ce travail :

```
# ./lxc-start2
Usage: lxc-start2 container_name
```

Pour accéder à la console du conteneur afin d'y lancer des commandes comme **ps** pour voir les processus qui tournent :

```
# lxc-console -n bbox -t 0
Connected to tty 0
Type <Ctrl+a q> to exit the console, <Ctrl+a Ctrl+a> to enter Ctrl+a itself
[...]
bbox# ps
PID   USER     COMMAND
    1 root     init
    4 root     /bin/syslogd
   14 root     /bin/udhcpc
[...]
bbox# hostname
bbox
```

On en a profité aussi pour lancer la commande **hostname**. Par défaut, un conteneur de type **busybox** prend le nom du conteneur (option **-n** de **lxc-create**) comme nom de hôte (i.e. **bbox**).

Comme indiqué dans la bannière d'accueil de la console, pour en sortir et revenir sur hôte, il faut taper la combinaison de touches **<CTRL>** + **<a>** + **<q>**.

Pour arrêter un conteneur, on a recours à la commande **lxc-stop** :

```
# lxc-ls -f
NAME    STATE   AUTOSTART GROUPS IPV4       IPV6
bbox    RUNNING 0         -      10.0.3.203 -
# lxc-stop -n bbox
# lxc-ls -f
NAME    STATE   AUTOSTART GROUPS IPV4 IPV6
bbox    STOPPED 0         -      -    -
```

Un fois arrêté, un conteneur est redémarré via **lxc-start** ou détruit définitivement avec **lxc-destroy** :

```
# lxc-destroy -n bbox
```

La figure 4, schématise un conteneur LXC.

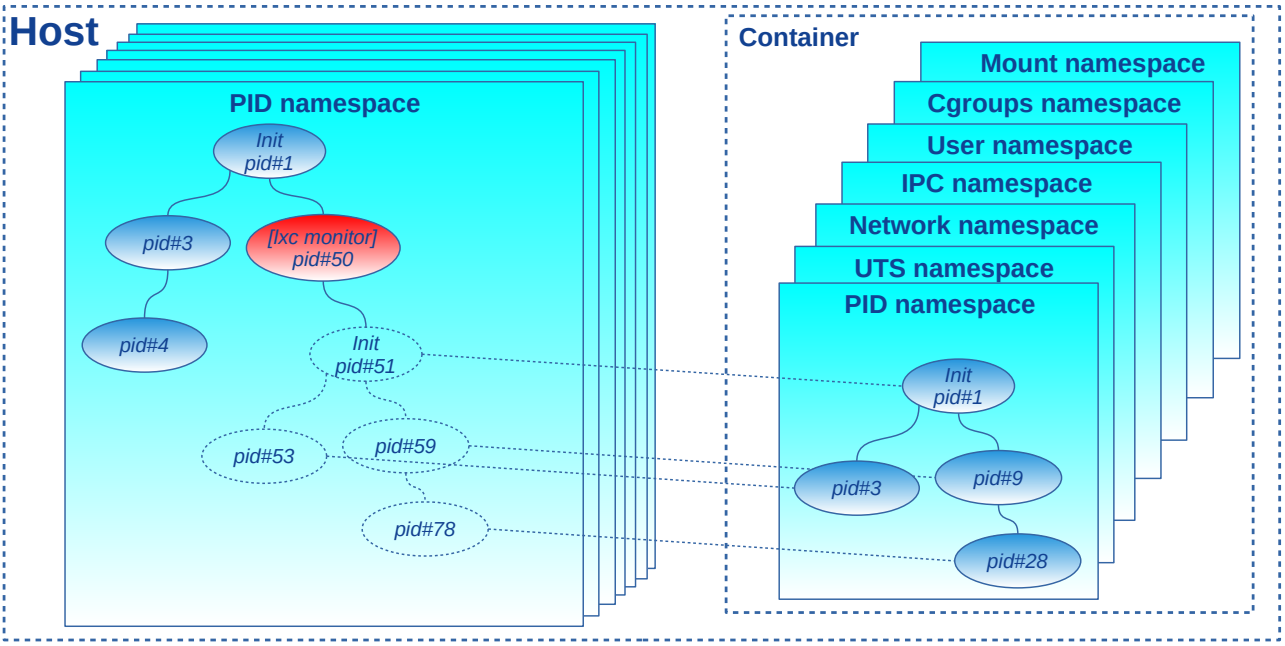

Fig. 4 : Conteneur LXC faisant tourner busybox

Nous notons que le processus **init** a le pid **1** dans le conteneur car il est le premier processus créé dans son pid_ns. Mais comme nous l'avons déjà dit et le reverrons plus en détails par la suite, **un pid_ns (tout comme le user_ns) a la particularité d'être hiérarchique**. Il y a une relation père-fils entre le namespace d'où est créé le namespace (le père) et le namespace nouvellement créé (le fils). Dans une telle hiérarchie, le père à la vision des ressources dans le fils mais l'inverse n'est pas vrai. Dans le cas des pid_ns, lorsqu'un processus est créé dans un namespace fils, il est identifié dans ce dernier mais il possède aussi un identifiant dans le namespace père. C'est la signification des liens en pointillé sur la figure 4. Le processus **init** de pid 1 dans le namespace fils est vu dans le namespace père avec le pid 51.

Ce qui intéresse particulièrement les internes de LXC, est l'identifiant du processus **init** dans le pid_ns père. En d'autres termes, le pid du processus vu du côté système hôte. Dans le cas de la figure 4, il s'agit de l'identifiant 51. Nous allons aussi beaucoup utiliser cet identifiant dans la suite de cet écrit pour « pénétrer » à l'intérieur des conteneurs. Il convient donc de savoir l'obtenir.

Comme chaque conteneur lancé via **lxc-start** est supervisé par un daemon nommé « [lxc monitor] chemin_lxc nom_conteneur » côté hôte. Le premier processus du conteneur (**init**) étant son fils, on peut déduire son pid en cherchant le fils du daemon :

```
# ps -ef | fgrep "[lxc monitor]"
root      8515  1652  0 21:18 ?        00:00:00 [lxc monitor] /var/lib/lxc bbox
# ps -ef | grep 8515
root      8515  1652  0 21:18 ?        00:00:00 [lxc monitor] /var/lib/lxc bbox
root      8520  8515  0 21:18 ?        00:00:00 init
```

Ce n'est rien d'autre que l'identifiant **PID** affiché par la commande **lxc-info** :

```
# lxc-info -n bbox
Name:           bbox
State:          RUNNING
PID:            8520
IP:             10.0.3.230
CPU use:        0.02 seconds
BlkIO use:      2.48 MiB
[...]
```

Comme on va avoir régulièrement besoin de cette information tout au long de cette série d'articles, nous nous facilitons la tâche avec notre script shell **lxc-pid**. Il reçoit en paramètre le nom d'un conteneur et affiche son identifiant en filtrant la valeur dans le résultat condensé (options **-H** et **-p**) de la commande **lxc-info** :

```
INIT_PID=`lxc-info -H -p -n $1 2>/dev/null`
[...]
  echo ${INIT_PID}
```

A l'exécution, cela donne :

```
# ./lxc-pid bbox
8520
```

1.5 Les namespaces d'un processus

Tout processus est lié à un seul namespace par type. Le répertoire **/proc/<pid>/ns** contient la liste des namespaces auxquels le processus d'identifiant **pid** est associé. Par exemple, les namespaces du processus **3542** sont :

```
$ ls -l /proc/3542/ns
total 0
lrwxrwxrwx 1 rachid rachid 0 janv.  21 11:46 cgroup -> 'cgroup:[4026531835]'
lrwxrwxrwx 1 rachid rachid 0 janv.  21 11:46 ipc -> 'ipc:[4026531839]'
lrwxrwxrwx 1 rachid rachid 0 janv.  21 11:46 mnt -> 'mnt:[4026531840]'
lrwxrwxrwx 1 rachid rachid 0 janv.  21 11:46 net -> 'net:[4026531992]'
lrwxrwxrwx 1 rachid rachid 0 janv.  21 11:46 pid -> 'pid:[4026531836]'
lrwxrwxrwx 1 rachid rachid 0 janv.  21 11:46 pid_for_children -> 'pid:[4026531836]'
lrwxrwxrwx 1 rachid rachid 0 janv.  21 11:46 user -> 'user:[4026531837]'
lrwxrwxrwx 1 rachid rachid 0 janv.  21 11:46 uts -> 'uts:[4026531838]'
```

Le contenu de ce répertoire est un ensemble de liens symboliques pointant sur le nom du namespace séparé par un « : » d'un numéro d'inode entre crochets. Le lien **pid_for_children** est une spécificité du pid_ns qui sera décrite plus tard. Le numéro d'inode sert à identifier les namespaces. Il est unique dans le système.

Pour obtenir un descripteur de fichier sur un namespace nécessaire à certains appels système vus plus bas, il suffira d'appeler **open()** sur la cible de ces liens symboliques. C'est un choix d'implémentation qui a été préféré à l'introduction d'un nouvel appel système qui prendrait en argument un identifiant de namespace et retournerait un descripteur de fichier **[10]**.

Nous verrons dans un article suivant que les cibles des liens symboliques sont des fichiers gérés par un système de fichiers interne au noyau dédié aux namespaces (**NSFS**).

1.6 Identification des namespaces

Un namespace est identifié de manière unique par deux entités : un numéro de périphérique et un numéro d'inode. Comparer les namespaces de deux processus revient par conséquent à comparer les champs **st_dev** et **st_ino** de la structure renvoyée par un appel système comme **stat()** sur la cible du lien symbolique correspondant. Notre programme **cmpns**, compare les namespaces de deux processus dont les identifiants sont passés en paramètres :

```
#define NB_NS_NAME 7
char *ns_name[NB_NS_NAME] = {
  "cgroup",
  "ipc",
  "mnt",
  "net",
  "pid",
  "user",
  "uts"
};

int main(int ac, char *av[])
{
[...]
  // Return 0, if the namespaces are equal
  // Return 1, otherwise
  rc = 0;

  // For all the namespaces
  for (i = 0; i < NB_NS_NAME; i ++) {

    // Build the pathnames of the namespaces
    snprintf(path1, sizeof(path1), "/proc/%s/ns/%s", av[1], ns_name[i]);
    snprintf(path2, sizeof(path2), "/proc/%s/ns/%s", av[2], ns_name[i]);

    rc = stat(path1, &stbuf1);
    rc = stat(path2, &stbuf2);

    // Comparison of the identifiers
    if ((stbuf1.st_dev == stbuf2.st_dev) &&
        (stbuf1.st_ino == stbuf2.st_ino)) {
      printf("%s is equal\n", ns_name[i]);
    } else {
      printf("%s is different\n", ns_name[i]);
      rc = 1;
    }
```

```
  } // End for

  return rc;
} // main
```

Si on le lance avec deux identifiants de processus associés aux mêmes namespaces, nous obtenons (**à lancer avec les droits du super utilisateur !**) :

```
# ./cmpns $$ 1
cgroup is equal
ipc is equal
mnt is equal
net is equal
pid is equal
user is equal
uts is equal
# echo $?
0
```

Par contre si on le lance avec deux identifiants de processus associés à des namespaces différents (par exemple, le processus courant et le processus **`init`** d'un conteneur). On voit que tous les namespaces sont différents sauf le user_ns :

```
# ./lxc-pid bbox
7246
# ./cmpns 7246 $$
cgroup is different
ipc is different
mnt is different
net is different
pid is different
user is equal
uts is different
# echo $?
1
```

Nous avons dit plus haut qu'un conteneur est un ensemble de namespaces dédiés. Hors l'affichage précédent montre que tous ses namespaces sont différents du système hôte sauf le user_ns. En fait, **un conteneur privilégié utilise le user_ns initial contrairement à un conteneur non privilégié qui a son propre user_ns**. Nous reviendrons sur cet aspect lors de l'étude des user_ns.

On notera que **nous avons utilisé l'appel système `stat()` et non pas `lstat()` qui nous donnerait les informations sur le lien symbolique lui-même et non pas sa cible**.

2 Les appels système

Les appels système manipulant les namespaces sont les suivants :

- **`clone()`** : création d'un processus en l'associant à de nouveaux namespaces ;

- **`setns()`** : association du processus appelant à un namespace existant ;

- **`unshare()`** : association du processus appelant à des nouveaux namespaces ;

- **`stat()`** : obtention de l'identifiant d'un namespace ;

- **`ioctl()`** : opérations diverses spécifiques aux namespaces.

2.1 clone()

`clone()` est l'appel système de Linux permettant de créer une tâche. Souvent méconnu du grand public, il est au coeur des bien plus célèbres services **`fork()`** et **`pthread_create()`**. Le prototype est le suivant (cf. **`man 2 clone`**) :

```
int clone(int (*fn)(void *), void *child_stack, int flags, void *arg, …);
```

En bref, cette fonction crée une nouvelle tâche avec pour point d'entrée la fonction **`fn`** (premier paramètre) qui recevra l'argument **`arg`** (quatrième paramètre) et la zone mémoire **`child_stack`** qui servira de pile d'exécution (second paramètre). Le troisième paramètre **`flags`** détermine le nombre et la nature des arguments qui suivent ainsi que ce qui est partagé entre la tâche appelante et la nouvelle tâche (informations sur le système de fichiers, espace mémoire, contexte d'entrées/sorties...). Parmi les drapeaux supportés, il y a ceux dédiés aux namespaces. Ils sont définis à raison d'un par namespace dans le fichier d'entête

<sched.h> pour associer la tâche créée à un nouveau :

- cgroup_ns (**CLONE_NEWCGROUP**) ;

- ipc_ns (**CLONE_NEWIPC**) ;

- net_ns (**CLONE_NEWNET**) ;

- pid_ns (**CLONE_NEWPID**) ;

- user_ns (**CLONE_NEWUSER**) ;

- uts_ns (**CLONE_NEWUTS**) ;

- mount_ns (**CLONE_NEWNS**).

La dénomination **CLONE_NEWNS** relative au mount_ns déroge à la règle de nommage utilisée pour les autres namespaces (i.e. **CLONE_NEW<nom de namespace abrégé>**). Au lieu de « NS » pour « NameSpace », on aurait pu s'attendre à quelque chose comme « MOUNT ». En fait, le mount_ns étant le premier namespace proposé par Linux, les développeurs de l'époque ne pensaient pas que d'autres suivraient.

La figure 5 schématise le fonctionnement de **clone()** auquel on a passé les drapeaux **CLONE_NEWUTS** et **CLONE_NEWIPC**. Le processus fils est associé à de nouveaux ipc_ns et uts_ns et hérite les autres de son père.

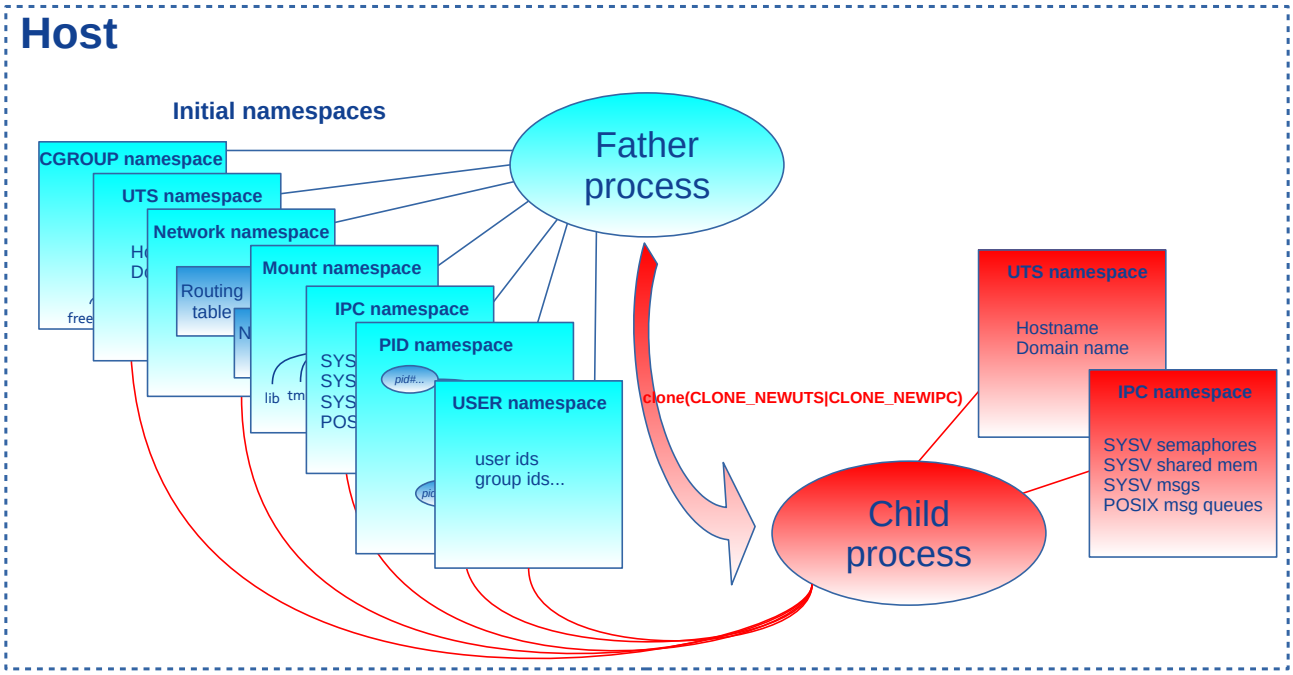

*Fig. 5 : Mise en oeuvre de **clone()***

Notre programme **clonens** utilise cet appel système pour engendrer et associer un processus à de nouveaux namespaces dont les noms sont passés en argument :

```
#define NB_NS_NAME 7

struct ns_type_t
{
  char *name;
  int   flag;
} ns[NB_NS_NAME] = {
  { "cgroup", CLONE_NEWCGROUP },
  { "ipc",    CLONE_NEWIPC    },
  { "mnt",    CLONE_NEWNS     },
  { "net",    CLONE_NEWNET    },
  { "pid",    CLONE_NEWPID    },
  { "user",   CLONE_NEWUSER   },
  { "uts",    CLONE_NEWUTS    }
};

int ns_name2type(const char *name)
{
  int i;
```

```
  for (i = 0; i < NB_NS_NAME; i ++) {
    if (!strcmp(ns[i].name, name)) {
      return ns[i].flag;
    }
  }

  return -1;

} // ns_name2type

static int child(void *p)
{
[...]
  pause();

  return 0;

} // child

int main(int ac, char *av[])
{
[...]
  // For all the namespaces passed as parameter, determine the
  // clone flag
  flags = 0;
  for (i = 1; i < ac; i ++) {
    // Translate the name into type
    rc = ns_name2type(av[i]);
[...]
    // Add the clone flag to the mask
    flags |= rc;
  } // End for

  // Clone a process
  cloneid = clone(child, child_stack + CHILD_STACK_SZ, SIGCHLD | flags, 0);
[...]
  printf("Created process %d in requested namespaces\n", cloneid);

  // Wait for the end of the process
  waitpid(cloneid, 0, 0);

  return 0;
} // main
```

Les noms de namespaces passés sur la ligne de commande sont convertis en drapeaux **CLONE_NEW*** via la fonction **ns_name2type()** pour être passés à l'appel système **clone()** qui crée un processus fils. Ce dernier exécute la fonction **child()** qui se met en attente avec l'appel système **pause()** tandis que le processus père attend sa terminaison via l'appel à **waitpid()**.

Lançons ce programme pour créer un processus associé à de nouveaux pid_ns et uts_ns :

```
# ./clonens pid uts
Created process 14020 in requested namespaces
```

Dans un autre terminal, utilisons notre programme **cmpns** pour comparer ses namespaces à celui du shell courant :

```
# ./cmpns 14020 $$
cgroup is equal
ipc is equal
mnt is equal
net is equal
pid is different
user is equal
uts is different
```

Les pid_ns et uts_ns sont différents, comme attendu. Ce qui est aussi vérifiable en listant le contenu du répertoire **/proc/<pid>/ns** pour les deux processus.

Le service **lxc-start** qui démarre un conteneur, s'appuie entre autres sur cet appel système pour créer le premier processus du conteneur et l'associer aux namespaces qui le caractérisent.

2.2 setns()

setns() permet au processus appelant de s'associer à un namespace existant. Le prototype est le suivant (cf. **man 2 setns**) :

```
int setns(int fd, int nstype);
```

Le premier paramètre **fd** est le descripteur du lien symbolique du namespace cible et le second paramètre **nstype**, qui a plus une fonction de mise au point qu'une réelle utilité, est le drapeau correspondant au type de namespace cible. Par type, on entend une valeur parmi les drapeaux **CLONE_NEW*** vus lors de la description de **clone()**. Ce paramètre peut être mis à **0** ou sinon doit avoir obligatoirement la valeur du drapeau correspondant au namespace associé au premier paramètre sous peine de retour erreur !

Le service **lxc-attach** qui exécute un programme dans un conteneur, s'appuie sur cet appel système. Il crée un processus fils sur hôte. Le fils s'associe à tous les namespaces qui caractérisent le conteneur cible avant d'exécuter le programme demandé.

Sur la même idée, notre programme **execns** prend en paramètres le pid d'un processus cible, les noms de namespaces du processus cible auxquels les associer (tous par défaut ou avec l'option **all**) et une commande à exécuter dans ces namespaces (par défaut **/bin/sh**). Le principe consiste à ouvrir un à un les cibles des liens symboliques dans le répertoire **/proc/<pid_cible>/ns** et à passer le descripteur de fichier associé à **setns()** afin d'associer le processus aux namespaces correspondant. Ensuite un processus fils est créé via **fork()** (**il hérite des namespaces de son père**) pour exécuter le programme demandé. L'étape de création d'un processus fils peut paraître inutile étant donné qu'une fois migré dans les namespaces cibles, le processus père pourrait exécuter le programme lui-même plutôt que de le sous-traiter à un fils. Mais une particularité des pid_ns empêche de faire cela : **lorsqu'un processus change de pid_ns, seuls ses fils sont effectivement associés à ce nouveau namespace**.

```
#define NB_NS_NAME 7

struct ns_list_t {
  char *name;
  int   selected;
  int   flag;
} ns_list[NB_NS_NAME] = {
  { "ipc",    0, CLONE_NEWIPC    },
  { "pid",    0, CLONE_NEWPID    },
[...]
  { "cgroup", 0, CLONE_NEWCGROUP },
  { "mnt",    0, CLONE_NEWNS     }
};

[...]
int main(int ac, char *av[])
{
[...]
  // Target pid
  tpid = atoi(p);
[...]
  // For each namespace of the target process:
  //   . Build the pathname to the symbolic link
  //   . Get the identifier and compare it to the calling process one
  //   . If different, enter into the namespace
  for (i = 0; i < NB_NS_NAME; i ++) {
    // If this namespace is requested
    if (ns_list[i].selected) {

      // Build the pathname
[...]
      snprintf(tpath, sizeof(tpath), "/proc/%d/ns/%s", tpid, ns_list[i].name);
[...]
      // Open the target namespace symbolic link
      fd = open(tpath, O_RDONLY);

      // Enter into the target namespace
      rc = setns(fd, ns_list[i].flag);
[...]
        printf("Moved into namespace %s\n", ns_list[i].name);
[...]
      close(fd);
    } // End if namespace selected
  } // End for

  // Fork a child process
  child = fork();
  if (!child) {

    // Child process
[...]
      cmd = av[2];
      execv(cmd, &(av[2]));
[...]
```

```
  } else {

    // Father process

    int status;

    // Wait for the end of the child process
    rc = waitpid(child, &status, 0);
[...]
    printf("program's status: %d (0x%x)\n", status, status);
  }

  return 0;
} // main
```

Avec ce programme, on peut ainsi exécuter une commande dans un conteneur comme on le ferait avec l'utilitaire **lxc-attach**. Dans l'exemple suivant, le process **init** d'un conteneur a l'identifiant **14010**. On passe cet identifiant et le programme **/bin/sh** en paramètre à **execns**. On en profite pour positionner la variable d'environnement **PS1** afin d'afficher le nom de host dans le prompt du shell (la variable est héritée par le shell créé). La mécanique mise en oeuvre est schématisée en figure 6.

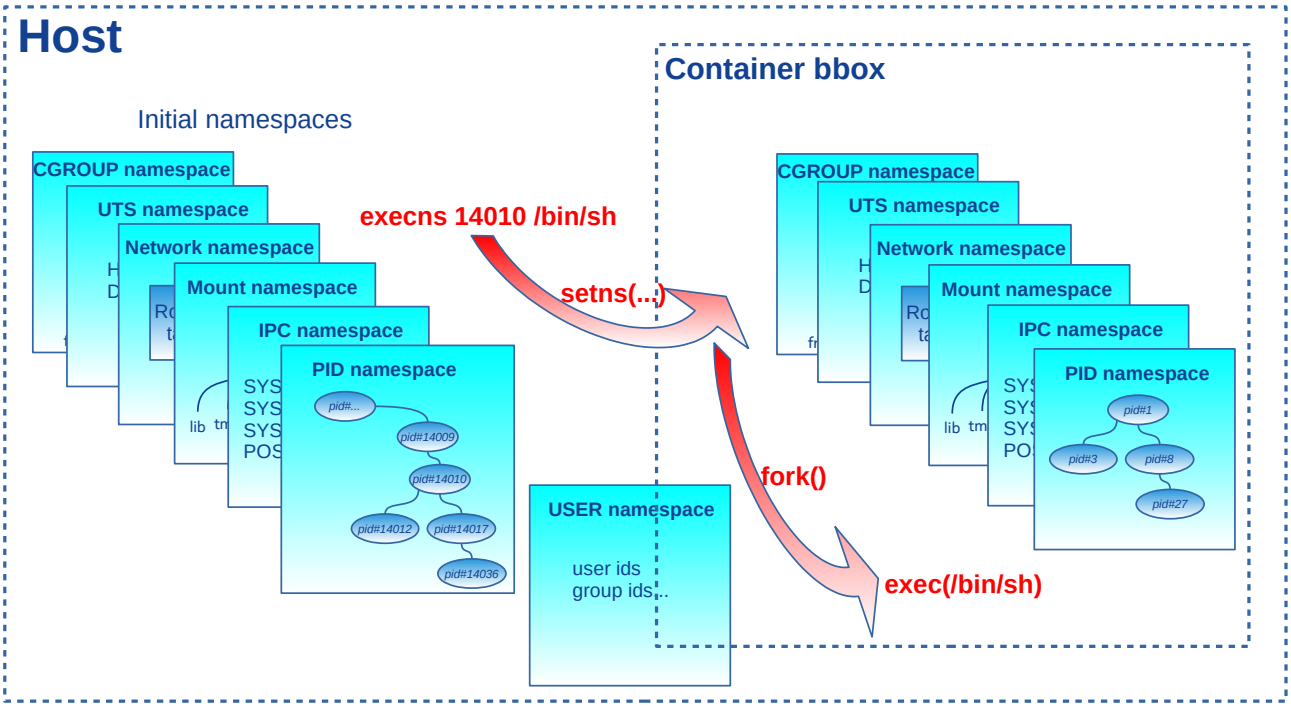

*Fig. 6 : **setns()** dans un conteneur*

Dans le shell ainsi créé, la commande **ps** montre bien la liste des processus du conteneur et non pas celle du système hôte. On appelle **exit** pour sortir du shell et donc rendre la main au processus père qui affiche le statut de retour de la commande.

```
# ./lxc-pid bbox
14010
# PS1=\\h\#\\x20 ./execns 14010 /bin/sh
Moved into namespace ipc
[...]
Moved into namespace cgroup
Moved into namespace mnt
[...]
bbox# ps
PID   USER     COMMAND
    1 root     init
    4 root     /bin/syslogd
   14 root     /bin/udhcpc
   15 root     /bin/getty -L tty1 115200 vt100
[...]
bbox# exit
program's status: 0 (0x0)
```

Le service **setns()** est aussi utilisé par la commande **lxc-start** à travers les trois options **--share-net**, **--share-ipc** et **--share-uts** pour respectivement partager le net_ns, l'ipc_ns et l'uts_ns d'un autre processus

ou conteneur. L'option est accompagnée d'un pid ou un nom de conteneur. Dans le second cas, le pid du processus **init** du conteneur est récupéré à partir du nom. Puis LXC ouvre le fichier **/proc/<pid>/ns/[net|ipc|uts]** et utilise l'appel système **setns()** pour migrer le conteneur dans le namespace correspondant.

2.3 unshare()

L'appel système **unshare()** permet au processus appelant de se désassocier d'une ou plusieurs parties du contexte d'exécution. Le prototype est le suivant (cf. **man 2 unshare**) :

```
int unshare(int flags);
```

Dans le cadre des namespaces, la fonction reçoit en paramètre le masque des drapeaux associés (i.e. les constantes **CLONE_NEW*** vues dans la description de **clone()**) afin de créer de nouveau namespaces et les associer à l'appelant.

C'est par exemple utilisé avec le drapeau **CLONE_NEWNS** dans l'utilitaire **lxc-create** afin de créer le système de fichiers du conteneur dans un mount_ns séparé de manière à éviter la pollution du mount_ns initial (le système de fichiers du hôte).

La figure 7 illustre le comportement de l'appel système lorsqu'il est appelé avec les drapeaux **CLONE_NEWUTS** et **CLONE_NEWIPC**. Un nouvel uts_ns ainsi qu'un nouvel ipc_ns sont créés. Le processus appelant se détache des namespaces d'origine correspondants pour être attaché à ces derniers. Mais rien ne change pour les autres namespaces.

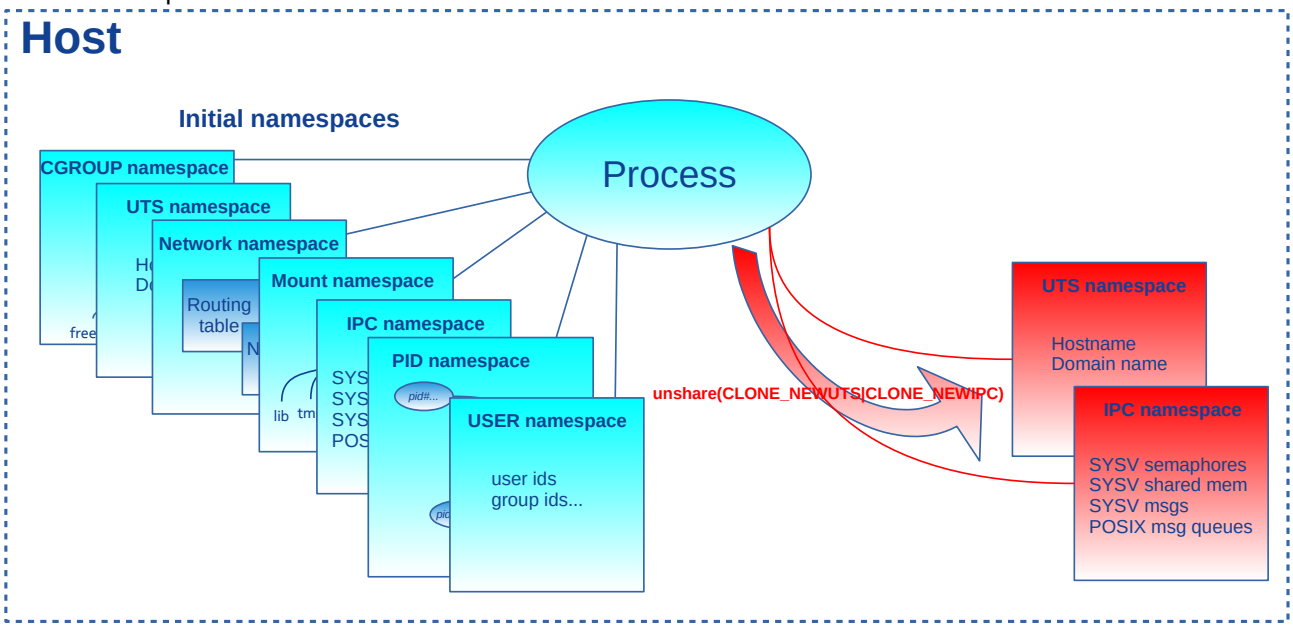

*Fig. 7 : Mise en oeuvre de **unshare()***

Notre programme **shns** donne la possibilité de créer un shell dans de nouveaux namespaces dont les noms sont passés en arguments (tous par défaut ou avec l'option **all**). Pour tous les namespaces demandés, le programme construit le masque de drapeaux associés et le passe à la fonction **unshare()**. Puis, un processus fils est créé (il hérite des nouveaux namespaces) via **fork()** pour exécuter le shell avec **execv()**.

```
#define NB_NS_NAME 7

struct ns_list_t {
  char *name;
  int   selected;
  int   flag;
} ns_list[NB_NS_NAME] = {
  { "ipc",    0, CLONE_NEWIPC    },
  { "pid",    0, CLONE_NEWPID    },
[...]
  { "cgroup", 0, CLONE_NEWCGROUP },
  { "mnt",    0, CLONE_NEWNS     }
};

// Set the "selected" field in ns_list[] for the given
// namespace or all (if keyword "all" is passed)
static int select_ns(const char *ns)
{
```

```
 [...]
} // select_ns


int main(int ac, char *av[])
{
[...]
  // For each namespace of the target process, set the flag
  flags = 0;
  for (i = 0; i < NB_NS_NAME; i ++) {
    if (ns_list[i].selected) {
      printf("New namespace '%s'\n", ns_list[i].name);
      flags |= ns_list[i].flag;
    } // End if namespace selected
  } // End for

  // Create brand new namespaces
  rc = unshare(flags);
[...]
  // Fork a child process
  child = fork();
  if (!child) {

    // Child process

    char *av_cmd[] = { DEFAULT_CMD, NULL };

    execv(av_cmd[0], av_cmd);
[...]
  } else {

    // Father process

    int status;

    // Wait for the end of the child process
    rc = waitpid(child, &status, 0);
[...]
    printf("program's status: %d (0x%x)\n", status, status);
  }

  return 0;
} // main
```

L'exécution de **shns** sans paramètre lance un shell dans de nouveaux namespaces. L'affichage de l'identifiant de process du shell courant donne la valeur **1** car le premier processus créé dans un nouveau pid_ns a pour identifiant **1** :

```
# PS1="SHNS# " ./shns
New namespace 'ipc'
New namespace 'pid'
New namespace 'net'
New namespace 'user'
[...]
SHNS# echo $$
1
```

Dans un autre shell, nous cherchons le pid du processus shell fils de **shns** et pouvons vérifier que de nouveaux namespaces ont été créés en les comparant aux namespaces initiaux à l'aide de notre programme **cmpns** :

```
# pidof shns
14729
# ps -ef | fgrep 14729 | fgrep /bin/sh
root     14730 14729  0 17:04 pts/0    00:00:00 /bin/sh
# ./cmpns $$ 14730
cgroup is different
ipc is different
mnt is different
[...]
```

2.4 stat()

Déjà évoqué plus haut et de manière pratique dans les programmes **cmpns** et **execns**, l'appel système **stat()** permet entre autres de récupérer le couple « numéro de périphérique, numéro d'inode » de la cible du lien symbolique d'un namespace. Ce duo constitue l'identifiant unique d'un namespace. Muni de cette information, il est par exemple possible de comparer les namespaces.

Notre programme **idns** prend en paramètres le pid d'un processus et les noms de namespaces (tous par défaut ou avec le mot clé **all**) afin d'afficher les identifiants correspondants :

```
int main(int ac, char *av[])
{
[...]
  // For all the namespaces passed as parameter
  for (i = 0; ns[i]; i ++) {
[...]
    // Build the pathname of the namespace
    snprintf(tpath, sizeof(tpath), "/proc/%s/ns/%s", av[1], ns[i]);

    // Get the information of the namespace
    rc = stat(tpath, &st);
[...]
    printf("%s [Device,Inode]: [%lu,%lu]\n", tpath, st.st_dev, st.st_ino);

  } // End for

  return 0;

} // main
```

Pour les namespaces initiaux et un conteneur, **idns** retourne par exemple ceci :

```
# ./idns $$
/proc/2078/ns/cgroup [Device,Inode]: [4,4026531835]
/proc/2078/ns/ipc [Device,Inode]: [4,4026531839]
[...]
/proc/2078/ns/uts [Device,Inode]: [4,4026531838]
/proc/2078/ns/user [Device,Inode]: [4,4026531837]
# ./lxc-pid bbox
2687
# ./idns 2687
/proc/2687/ns/cgroup [Device,Inode]: [4,4026532477]
/proc/2687/ns/ipc [Device,Inode]: [4,4026532414]
[...]
/proc/2687/ns/uts [Device,Inode]: [4,4026532413]
/proc/2687/ns/user [Device,Inode]: [4,4026531837]
```

Le conteneur étant privilégié, il est associé au même user_ns que celui de l'hôte.

2.5 ioctl()

Quelques opérations relatives aux namespaces sont disponibles via l'appel système **ioctl()**. Toutes les opérations, sauf **SIOCGSKNS**, sont documentées dans **man 2 ioctl_ns**.

Dans sa version 2.1.1, LXC n'a pas du tout recours à ces opérations. Le champ d'application peut donc paraître assez limité.

2.5.1 NS_GET_USERNS

Cette opération retourne le descripteur de fichier référençant le user_ns auquel appartient un namespace (c.-à-d. le user_ns à partir duquel il a été créé). La fonction reçoit le descripteur de fichier **fd** du namespace et retourne le descripteur **new_fd** du user_ns auquel il appartient. Le prototype est le suivant :

```
#include <sys/ioctl.h>
#include <linux/nsfs.h>

new_fd = ioctl(fd, NS_GET_USERNS);
```

La notion d'appartenance à un user_ns est importante pour des raisons de sécurité. Tout ce qui est fait dans un namespace donnée, est conditionné par le degré de privilèges configuré dans le user_ns auquel le namespace appartient.

Notre programme **ownerns** utilise cet **ioctl** pour déterminer le user_ns auxquels appartiennent les namespaces d'un processus donné (identifiant passé en premier paramètre). Les namespaces sont énumérés par leur nom sur le reste de la ligne de commande (tous par défaut ou avec le mot clé **all**) :

```
int main(int ac, char *av[])
[...]
  // For all the namespaces passed as parameter
  for (rc = 0, i = 0; ns[i]; i ++) {
[...]
    // Build the pathname of the namespace
```

```
    snprintf(tpath, sizeof(tpath), "/proc/%s/ns/%s", av[1], ns[i]);

    // Open the pathname of the namespace
    tfd = open(tpath, O_RDONLY);

    // Get a file descriptor on the user_ns to which this namespace
    // belongs to
    ufd = ioctl(tfd, NS_GET_USERNS);

    if (ufd < 0) {
      ERR("ioctl(%s, NS_GET_USERNS): '%m' (%d)\n", tpath, errno);
      // Continue with other namespaces
      rc = 1;
    } else {

      fstat(ufd, &st);

      printf("%s belongs to [Device,Inode]: [%lu,%lu]\n", tpath, st.st_dev, st.st_ino);
    }

    close(tfd);
    close(ufd);
  } // End for

  return rc;

} // main
```

Appelé sur hôte avec l'identifiant du shell courant, le programme affiche bien l'identifiant du user_ns initial :

```
# ./ownerns $$ all
/proc/4770/ns/cgroup belongs to [Device,Inode]: [4,4026531837]
/proc/4770/ns/ipc belongs to [Device,Inode]: [4,4026531837]
[...]
ERROR@main#75: ioctl(/proc/4770/ns/user, NS_GET_USERNS): 'Operation not permitted' (1)
```

L'erreur affichée pour le user_ns est conforme au manuel qui précise que **EPERM** est retourné si on tente d'obtenir le parent du user_ns initial.

La même commande pour le processus **init** d'un conteneur, nous obtenons la même erreur pour le user_ns car le conteneur étant privilégié, il est associé à celui du hôte (c.-à-d. initial) :

```
# ./lxc-pid bbox
5071
# ./ownerns 5071 all
/proc/5071/ns/cgroup belongs to [Device,Inode]: [4,4026531837]
/proc/5071/ns/ipc belongs to [Device,Inode]: [4,4026531837]
[...]
ERROR@main#75: ioctl(/proc/5071/ns/user, NS_GET_USERNS): 'Operation not permitted' (1)
```

Si nous lançons un shell dans un nouveau pid_ns, net_ns, uts_ns et user_ns avec les options respectives **-p**, **-n**, **-u** et **-U** de l'utilitaire **unshare** (que nous verrons dans le prochain article) :

```
# unshare -p -n -u -U /bin/sh
$ echo $$
15357
```

Dans un autre terminal, la commande **owner_ns** indique que le net_ns et l'uts_ns appartiennent au nouvel user_ns (i.e. **[4,4026532935]**) par contre le pid_ns et le user_ns continuent à appartenir au user_ns initial (i.e. **[4,4026531837]**) :

```
# ./ownerns 15357 all
/proc/15357/ns/cgroup belongs to [Device,Inode]: [4,4026531837]
/proc/15357/ns/ipc belongs to [Device,Inode]: [4,4026531837]
/proc/15357/ns/mnt belongs to [Device,Inode]: [4,4026531837]
/proc/15357/ns/net belongs to [Device,Inode]: [4,4026532935]
/proc/15357/ns/pid belongs to [Device,Inode]: [4,4026531837]
/proc/15357/ns/uts belongs to [Device,Inode]: [4,4026532935]
/proc/15357/ns/user belongs to [Device,Inode]: [4,4026531837]
```

Tout cela s'explique par les raisons suivantes :

- Le nouvel user_ns appartient au user_ns à partir duquel il est créé. Ici c'est le user_ns initial ;

- Le manuel des user_ns spécifie que **clone()** ou **unshare()** (appel système appelé par la commande **unshare**) avec entre autres le drapeau **CLONE_NEWUSER**, garantissent que le user_ns est créé en premier. Par conséquent, les namespaces créés dans la foulée appartiendront à ce nouveau user_ns. D'où le fait que les nouveaux net_ns et uts_ns de notre exemple appartiennent au user_ns nouvellement créé. Mais quid du pid_ns ? Il devrait suivre la même règle ?

- Lorsqu'un processus appelle **unshare()** ou **setns()** avec le drapeau **CLONE_NEWPID**, **ce sont ses fils qui entrent dans le nouveau pid_ns**. Le père, donc l'appelant de ces appels système, est cantonné à son pid_ns courant qui appartient au user_ns initial. Comme l'utilitaire **unshare** n'invoque pas **fork()** mais juste **execv()** pour lancer le shell, ce dernier reste dans le pid_ns initial. Par contre, les fils créés par le processus lancés par **unshare** (ici la commande **/bin/sh**) entreront dans le nouveau pid_ns qui par conséquent appartiendra au nouvel user_ns.

Par la suite, la présentation de l'utilitaire **unshare** et ensuite du pid_ns permettront d'éclaircir ce propos car le shell lancé comme précédemment ne peut lancer qu'une première commande puis il affiche une erreur de **fork()** pour les suivantes :

```
# unshare -p -n -u -U /bin/sh
$ date
mardi 28 janvier 2020, 21:50:31 (UTC+0100)
$ ls
/bin/sh: 2: Cannot fork
```

2.5.2 NS_GET_PARENT

Cette opération retourne le descripteur de fichier (**new_fd**) référençant le namespace parent du namespace dont le descripteur de fichier est passé en paramètre (**fd**). Le prototype est le suivant :

```
#include <sys/ioctl.h>
#include <linux/nsfs.h>

new_fd = ioctl(fd, NS_GET_PARENT);
```

Cela ne s'applique qu'aux namespaces hiérarchiques. Seuls deux le sont : le user_ns et le pid_ns. Pour tout autre namespace, l'erreur **EINVAL** est retournée.

Notre programme **parentns** est une copie de **ownerns** où l'opération **NS_GET_USERNS** a été remplacée par **NS_GET_PARENT**. Appelé avec un namespace non hiérarchique en paramètre, l'opération retourne l'erreur **EINVAL** conformément au manuel :

```
# ./parentns $$ uts
ERROR@main#89: ioctl(/proc/2193/ns/uts, NS_GET_PARENT): 'Invalid argument' (22)
```

Avec un namespace hiérarchique mais initial, l'opération retourne l'erreur **EPERM** conformément au manuel :

```
# ./parentns $$ user
ERROR@main#89: ioctl(/proc/13484/ns/user, NS_GET_PARENT): 'Operation not permitted' (1)
```

Si on l'appelle pour les namespaces pid et user d'un conteneur, on obtient l'identifiant du namespace père uniquement pour le pid_ns car le conteneur étant privilégié, évolue dans le user_ns initial pour lequel l'opération retourne **EPERM** :

```
# ./lxc-pid bbox
3316
# ./parentns 3316 pid user
/proc/3316/ns/pid is child of [Device,Inode]: [4,4026531836]
ERROR@main#89: ioctl(/proc/3316/ns/user, NS_GET_PARENT): 'Operation not permitted' (1)
```

2.5.3 NS_GET_NSTYPE

Cette opération retourne le type de namespace (c.-à-d. le drapeau **CLONE_XXX** vu avec les appels système précédents) dont le descripteur de fichier est passé en paramètre (**fd**). Le prototype est le suivant :

```
#include <sys/ioctl.h>
#include <linux/nsfs.h>

nstype = ioctl(fd, NS_GET_NSTYPE);
```

2.5.4 NS_GET_OWNER_UID

Cette opération retourne l'identifiant d'utilisateur (troisième paramètre **uid**) du processus qui a créé un user_ns dont le descripteur de fichier est passé en paramètre (**fd**). Le prototype est le suivant :

```
#include <sys/ioctl.h>
#include <linux/nsfs.h>

rc = ioctl(fd, NS_GET_OWNER_UID, &uid);
```

Nous parlons ici de l'identifiant effectif de l'utilisateur du processus qui a créé le namespace. A ne pas confondre avec l'opération **NS_GET_USERNS** qui retourne le descripteur de fichier du user_ns auquel

appartient un namespace donné.

Notre programme **userns** reçoit en paramètre le pid d'un processus et affiche le nom et l'identifiant de l'utilisateur de son user_ns :

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

  // Open the pathname of the namespace
  tfd = open(tpath, O_RDONLY);
[...]
  // Get the owner's uid to which this namespace belongs to
  rc = ioctl(tfd, NS_GET_OWNER_UID, &uid);
[...]
  // Get the owner's password entry to get his name
  pw = getpwuid(uid);
[...]
  printf("%s belongs to user: '%s' (%d)\n", tpath, pw->pw_name, uid);
[...]
} // main
```

Avec l'identifiant du shell courant, le programme affiche par exemple :

```
# ./userns $$
/proc/2078/ns/user belongs to user: 'root' (0)
```

C'est effectivement sous l'identité du super-utilisateur que Linux démarre et crée ses namespaces initiaux.

2.5.5 SIOCGSKNS

Cette opération est à part car elle n'est pas documentée. Elle a semble-t-il été introduite pour les opérations de « dump/restore » **[11]** des conteneurs **OpenVZ [12]** afin d'obtenir un descripteur de fichier **fd** sur le net_ns auquel un descripteur de socket **sd** est associé. Le prototype est le suivant :

```
#include <sys/ioctl.h>
#include <linux/sockios.h>

fd = ioctl(sd, SIOCGSKNS);
```

Conclusion

La mise en oeuvre des appels système à travers les programmes d'exemple qui ont jalonné cet article, a d'ores et déjà révélé des subtilités sur les namespaces. Certaines ont été expliquées mais d'autres juste mentionnées pour ne pas alourdir le propos. Nous ne manquerons pas d'y revenir en détail dans les articles suivants ! Le second opus sera consacré aux utilitaires basés sur ces appels système.

Références

[1] The Use of Name Spaces in Plan 9 : http://doc.cat-v.org/plan_9/4th_edition/papers/names

[2] Le système Plan 9 : https://fr.wikipedia.org/wiki/Plan_9_from_Bell_Labs

[3] Plan 9 from outer space : https://fr.wikipedia.org/wiki/Plan_9_from_Outer_Space

[4] Le film « Ed Wood » de Tim Burton : https://fr.wikipedia.org/wiki/Ed_Wood_(film)

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

[6] Linux containers (LXC) : https://linuxcontainers.org/lxc/

[7] Les namespaces de Linux : https://en.wikipedia.org/wiki/Linux_namespaces

[8] L'intégration continue : https://fr.wikipedia.org/wiki/Int%C3%A9gration_continue

[9] BusyBox, le couteau suisse de Linux embarqué : https://busybox.net/

[10] Namespace file descriptors : https://lwn.net/Articles/407495/

[11] Add an ioctl to get a socket network namespace : https://lore.kernel.org/patchwork/patch/728774/

[12] OpenVz : https://fr.wikipedia.org/wiki/OpenVZ