Les structures de données des namespaces dans le noyau

Rachid Koucha
[Ingénieur développement logiciel]

Après des premiers pas dans le monde des namespaces à travers l'étude des appels système et des utilitaires qui les mettent en œuvre en espace utilisateur, ce nouvel opus se consacre à leur implémentation dans le noyau.

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°243 de décembre 2020 :

Introduction

Lire le code du noyau **[1]** et l'exposer sous forme d'article est un exercice délicat car c'est susceptible d'ennuyer le lecteur. Ce n'est pas une étape nécessaire à la compréhension et l'utilisation des namespaces. Mais c'est indéniablement un plus de connaître les rouages de la partie immergée. Cet opus se concentre sur les structures de données associées aux namespaces dans le code source du noyau **Linux 5.3.0**.

1 Le descripteur de processus

En espace utilisateur on parle communément de processus et de threads pour désigner les unités d'exécution. Un processus est même plus précisément vu comme un ensemble de threads : le thread principal et les threads secondaires. Dans le noyau ce sont simplement des tâches. La tâche « thread group leader » représente le thread principal et les autres tâches représentent les threads secondaires.

A toute tâche est associé un descripteur (structure **task_struct**) au sein duquel se trouvent toutes les informations nécessaires à son suivi et fonctionnement. Il est déclaré dans le fichier **include/linux/sched.h** :

```
struct task_struct {
[...]
        void                            *stack;
[...]
        int                             prio;
        int                             static_prio;
        int                             normal_prio;
        unsigned int                    rt_priority;

        const struct sched_class        *sched_class;
[...]
        int                             nr_cpus_allowed;
        const cpumask_t                 *cpus_ptr;
        cpumask_t                       cpus_mask;
[...]
        pid_t                           pid;
        pid_t                           tgid;
[...]
        struct pid                      *thread_pid;
[...]
        const struct cred __rcu         *cred;
[...]
        char                            comm[TASK_COMM_LEN];
[...]
        struct nsproxy                  *nsproxy;
[...]
};
```

A l'image du pointeur **this** du langage C++ qui pointe sur l'objet courant, le pointeur **current** référence toujours le descripteur de la tâche en cours d'exécution. Il est largement utilisé dans le noyau pour accéder aux informations de la tâche courante.

Parmi les nombreux champs comme la pile, les identifiants, la priorité, le nom du programme ou l'affinité CPU pour n'en citer que quelques-uns, se trouve un pointeur sur la structure **nsproxy**.

2 Les namespaces d'un processus

2.1 Le nsproxy

La structure **nsproxy** contient **un ensemble de pointeurs référençant les namespaces auxquels la tâche est associée**. Elle est déclarée dans le fichier **include/linux/nsproxy.h** :

```
struct nsproxy {
        atomic_t count;
        struct uts_namespace *uts_ns;
        struct ipc_namespace *ipc_ns;
```

```
        struct mnt_namespace *mnt_ns;
        struct pid_namespace *pid_ns_for_children;
        struct net           *net_ns;
        struct cgroup_namespace *cgroup_ns;
};
```

On notera tout d'abord l'**utilisation du principe des compteurs de références** (champ généralement nommé **count** ou **kref**) dans les structures de données afin de provoquer la libération implicite de ces structures à partir du moment où elles ne sont plus utiles (c.-à-d. quand le compteur atteint la valeur 0). C'est l'un des modèles de conception de Linux **[2]**. L'allocation dynamique (sauf pour certains namespaces initiaux qui ne sont jamais désalloués) utilise l'allocateur **SLAB** **[3]** dédié aux objets de taille fixe.

Les champs **uts_ns**, **ipc_ns**, **mnt_ns**, **net_ns** et **cgroup_ns** pointent respectivement sur les descripteurs des namespaces UTS, IPC, mount, network et cgroup comme schématisé en figure 1.

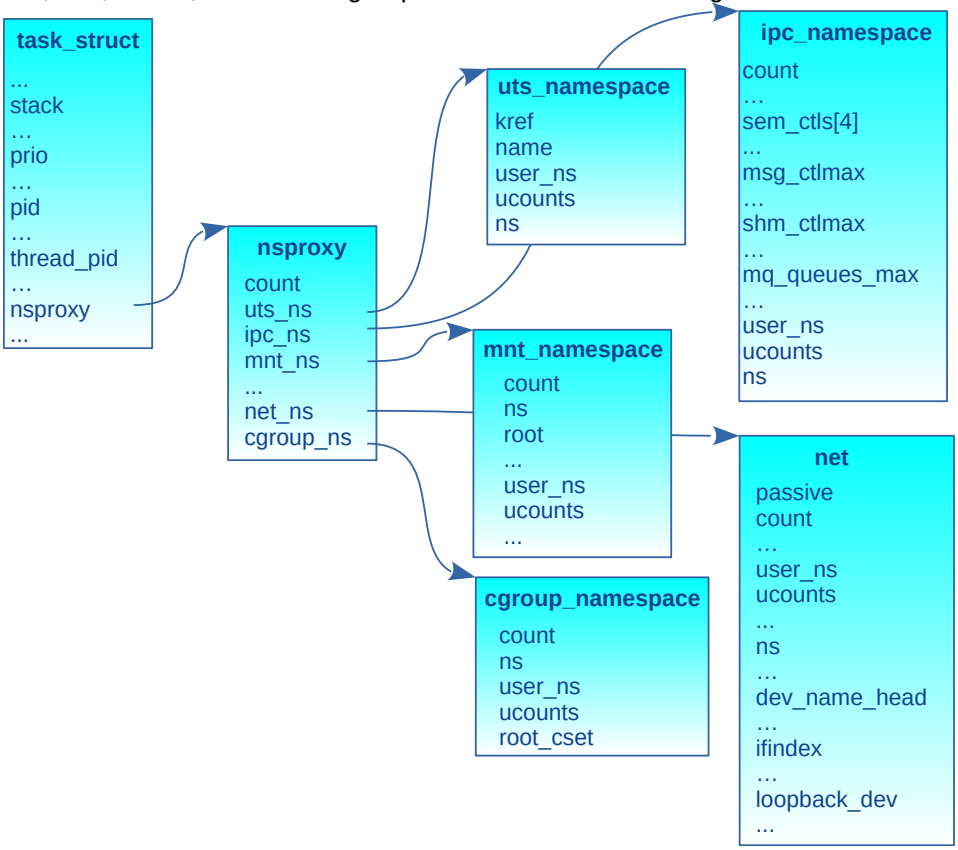

*Fig. 1: **nsproxy** d'une tâche*

2.2 Les namespaces hiérarchiques

Le user_ns et le pid_ns ne sont pas référencés directement par la structure **nsproxy**. Ils sont gérés différemment notamment à cause :

- De leur aspect hiérarchique (un user_ns ou un pid_ns est le fils du user_ns ou du pid_ns associé à la tâche qui l'a créé) ;

- Du fait que tout namespace est la propriété du user_ns associé à la tâche qui l'a créé ;

- Du fait qu'une tâche à un identifiant (c.-à-d. pid) différent dans chaque pid_ns et par conséquent la structure **task_struct** doit permettre d'accéder à tous ces identifiants et aux pid_ns associés ;

- Du fait qu'une tâche pouvant changer de user_ns, peut acquérir de nouveaux droits.

L'aspect hiérarchique est caractérisé par deux champs dans la structure décrivant le namespace :

- **parent** : pointeur sur la structure du namespace père. Quand ce champ est **NULL**, cela indique le sommet de la hiérarchie donc **un user_ns ou un pid_ns initial** (c.-à-d. **les namespaces créés au démarrage du système encore appelé « système hôte » quand on est dans un environnement de conteneurs**) ;

- **level** : entier désignant le niveau dans la hiérarchie. Le premier niveau est 0 (la racine ou niveau initial). La valeur augmente séquentiellement pour chaque nouveau namespace fils.

La figure 2 donne un exemple d'une hiérarchie de trois niveaux de user_ns. Ils sont liés par le pointeur **parent** et chaque niveau est marqué par le champ **level**.

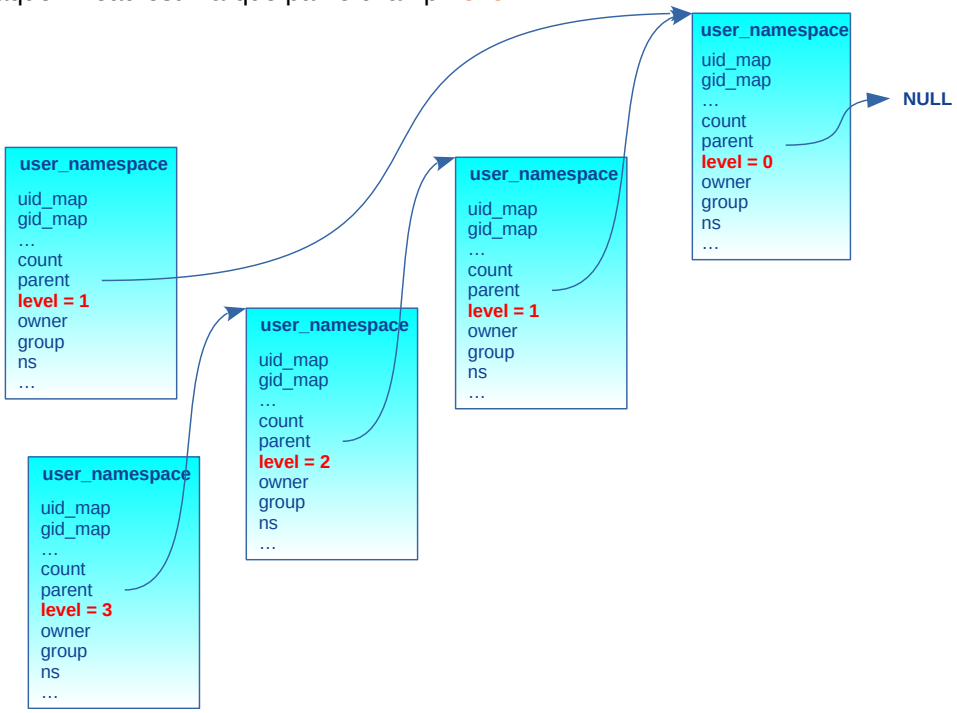

Fig. 2: Hiérarchie de user_ns

2.3 Le user_ns propriétaire

Toute structure décrivant un namespace possède un champ nommé **user_ns** qui pointe sur le descripteur du user_ns propriétaire comme indiqué sur la figure 3 avec les flèches rouges. On aurait pu complexifier le dessin en mettant un user_ns différent pour certains namespaces mais le cas le plus courant dans la pratique est qu'une tâche a le même user_ns pour tous ses namespaces.

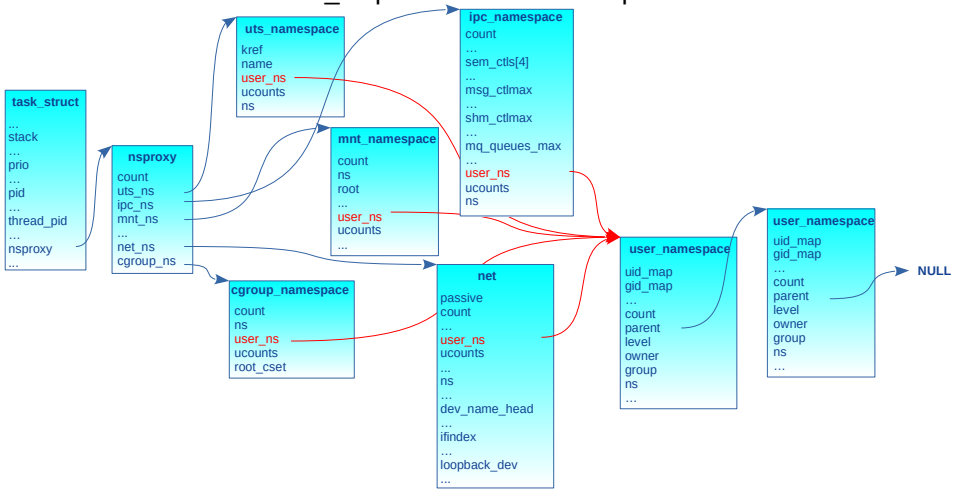

Fig. 3: user_ns des tâches

2.4 Gestion des identifiants de processus

Pour les pid_ns, une structure intermédiaire nommée **pid** a été mise en place pour refléter l'identifiant de processus dans tous les namespaces de la hiérarchie **[4]**. Elle est déclarée dans **include/linux/pid.h** :

```
struct upid {
```

```
        int nr;
        struct pid_namespace *ns;
};

struct pid
{
        refcount_t count;
        unsigned int level;
[...]
        struct upid numbers[1];
};
```

Le champ **level** est le niveau du processus dans la hiérarchie des pid_ns. Il fonctionne de la même manière que le même champ dans les descripteurs des pid_ns et user_ns vus plus haut (figure 2). Ce champ est l'indice maximum dans la table **numbers[]** dont les entrées sont décrites par la structure **upid**. Chaque entrée de cette table associe un identifiant au processus (champ **nr**) dans le pid_ns de niveau **level** de la hiérarchie (référencé par le champ **ns**).

La structure **pid** est allouée par la fonction interne **alloc_pid()** définie dans **kernel/pid.c**. Appelée lors de la création d'un processus fils (p. ex. **clone()**), son résultat est assigné au champ **thread_pid** du descripteur de tâche. La figure 4 illustre le propos en représentant le cas d'un processus créé dans un pid_ns de troisième niveau (c.-à-d. **level** 2).

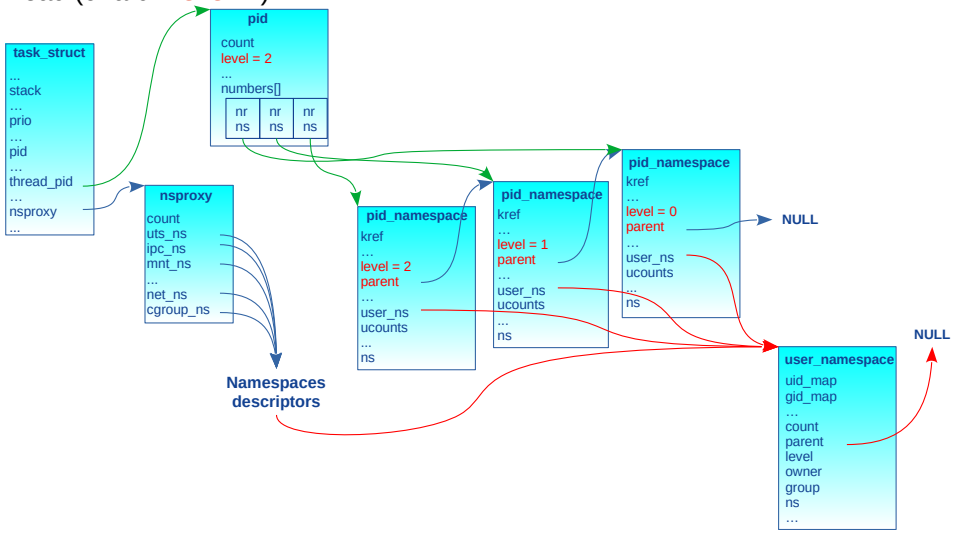

Fig. 4: Allocation des identifiants de processus

L'identifiant de processus (champ **nr**) dans chaque pid_ns est alloué via le mécanisme **IDR** du noyau **[5]** afin d'associer le pointeur sur le pid_ns avec l'identifiant de processus dans ce namespace. Cela permet à partir d'un identifiant de processus de retrouver le pointeur sur le descripteur de son pid_ns courant et inversement. Ainsi, la fonction **alloc_pid()** appelle **idr_alloc_cyclic()** avec le champ **idr** du pid_ns pour allouer un identifiant séquentiel unique à partir de la valeur 1 afin de renseigner le champ **nr** de chaque entrée dans la table **number[]**. Si le pid_ns est nouveau (c.-à-d. drapeau **CLONE_NEWPID** passé à **clone()**), la valeur de **nr** sera 1 (l'identifiant du processus **init**).

2.5 Allocation des namespaces

Lorsque les tâches sont associées aux mêmes namespaces, elles pointent toutes sur la même structure **nsproxy** comme indiqué en figure 5 (le pointeur est hérité de père en fils). Le champ **count** reflète bien-entendu chaque référence :

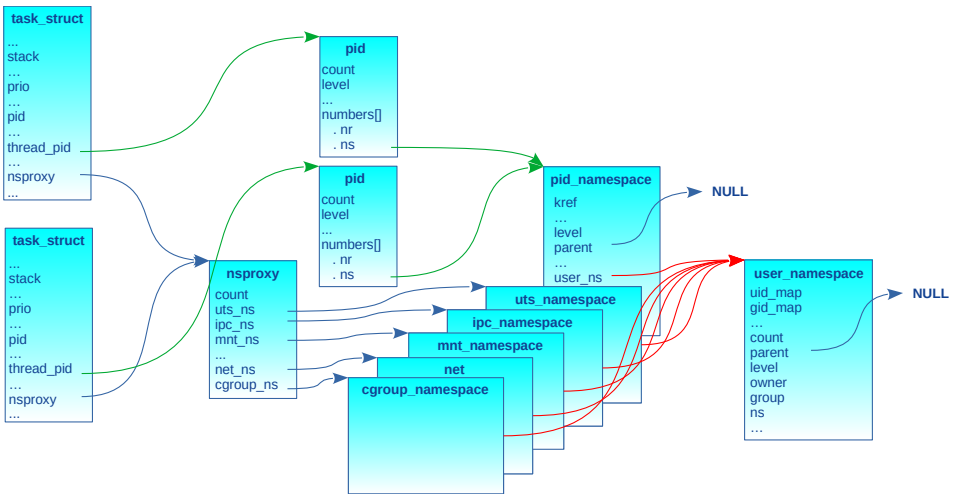

Fig. 5: Tâches associées aux mêmes namespaces

Si au moins un des namespaces diffère, une nouvelle structure **nsproxy** est allouée comme indiqué en figure 6 où l'uts_ns diffère entre les deux tâches.

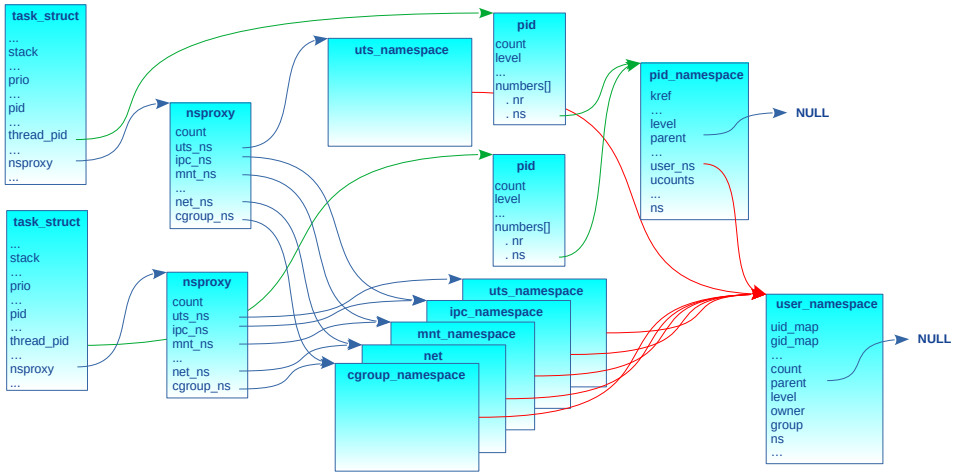

Fig. 6: Tâches avec un namespace (uts_ns) différent

C'est la fonction **copy_namespaces()** dans **kernel/nsproxy.c**, appelée par **clone()** lorsqu'une tâche crée un fils qui décide ou pas de l'allocation d'un nouveau **nsproxy** :

```
int copy_namespaces(unsigned long flags, struct task_struct *tsk)
{
        struct nsproxy *old_ns = tsk->nsproxy;
        struct user_namespace *user_ns = task_cred_xxx(tsk, user_ns);
        struct nsproxy *new_ns;
```

Si la condition indiquant qu'aucun drapeau **CLONE_NEWXXX** n'est passé, elle incrémente, via **get_nsproxy()**, le compteur de référence sur le **nsproxy** du processus père signifiant ainsi que le processus fils est associé aux mêmes namespaces que son père.

```
        if (likely(!(flags & (CLONE_NEWNS | CLONE_NEWUTS | CLONE_NEWIPC |
                              CLONE_NEWPID | CLONE_NEWNET |
                              CLONE_NEWCGROUP)))) {
                get_nsproxy(old_ns);
                return 0;
        }
```

Par contre, si au moins un des drapeaux est passé pour créer au moins un nouveau namespace, une nouvelle structure **nsproxy** est allouée en appelant **create_new_namespaces()**.

```
        new_ns = create_new_namespaces(flags, tsk, user_ns, tsk->fs);
[...]
        tsk->nsproxy = new_ns;
        return 0;
}
```

Les pointeurs sur les namespaces de la structure **nsproxy** nouvellement allouée sont soit une copie du même pointeur dans le **nsproxy** du père (c.-à-d. le drapeau **CLONE_NEWXXX** associé n'a pas été passé), soit

mis à jour pour référencer le namespace nouvellement alloué (c.-à-d. le drapeau **CLONE_NEWXXX** associé a été passé). On verra par la suite que dans ce dernier cas, l'opération « copy » des namespaces est appelée pour faire hériter aux namespaces du processus fils, les informations des namespaces du processus père.

Pour en revenir à la figure 6, elle illustre un processus père qui a créé un processus fils en l'associant à un nouvel uts_ns (c.-à-d. appel à **clone()** avec le drapeau **CLONE_NEWUTS**). Le processus engendré pointe sur une nouvelle structure **pid**. La structure **pid** pointe sur pid_ns du processus père mais son champ **nr** reflète le nouveau pid du processus nouvellement créé. Ce dernier a aussi un nouveau **nsproxy** dont un pointeur référence le descripteur du nouvel uts_ns tandis que les autres sont des copies de ceux du processus père car les autres namespaces sont partagés.

2.6 Accréditations des tâches

Les accréditations (c.-à-d. « credentials » en anglais) sont sous la forme d'une structure **cred** définie dans **include/linux/cred.h**. Elle contient toutes les informations d'identification (identifiants réels et effectifs d'utilisateur, identifiants réels et effectifs de groupe, les capacités...) et notamment un pointeur sur le descripteur de user_ns auquel elle s'applique :

```
struct cred {
[...]
        kuid_t          uid;            /* real UID of the task */
        kgid_t          gid;            /* real GID of the task */
[...]
        kuid_t          euid;           /* effective UID of the task */
        kgid_t          egid;           /* effective GID of the task */
[...]
        kernel_cap_t    cap_inheritable; /* caps our children can inherit */
        kernel_cap_t    cap_permitted; /* caps we're permitted */
        kernel_cap_t    cap_effective; /* caps we can actually use */
[...]
        struct user_struct *user;       /* real user ID subscription */
        struct user_namespace *user_ns; /* user_ns the caps and keyrings are relative to. */
[...]
} __randomize_layout;
```

Cette structure est référencée par le champ **cred** du descripteur de tâche **task_struct**. La figure 7 illustre un processus associé à ses namespaces. Ils sont ont tous été créés dans le même user_ns. Le champ **cred** du descripteur de tâche pointe sur ce même user_ns.

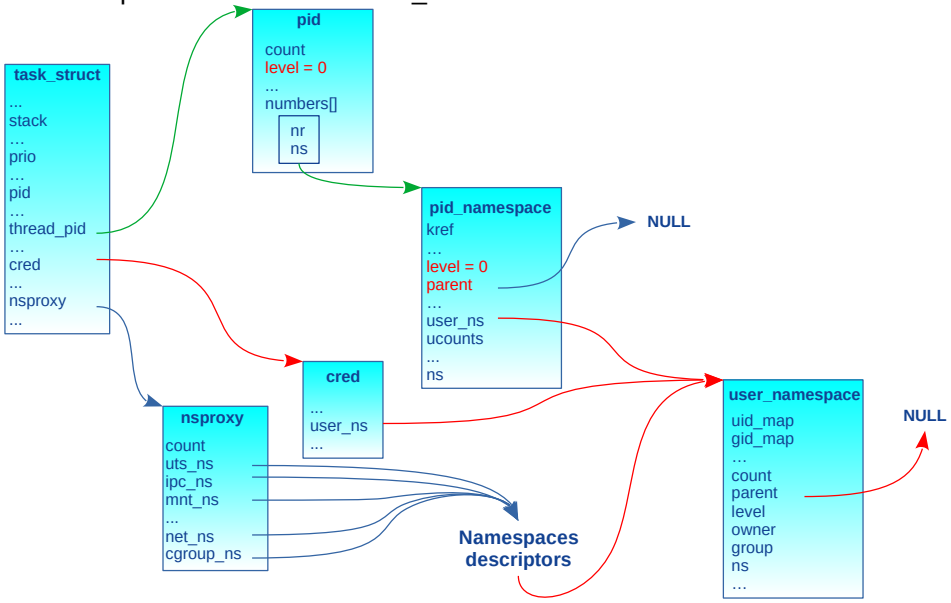

Fig. 7: User_ns des accréditations d'un processus

Nous reviendrons sur ces aspects dans un article dédié aux user_ns mais on peut d'ores et déjà retenir qu'un processus peut changer de user_ns (via l'appel système **setns()**). Dans ces conditions, le pointeur **cred** du descripteur de tâche est mis à jour pour pointer sur une structure **cred** nouvellement allouée et renseignée avec les nouvelles accréditations et le pointeur sur le descripteur du user_ns cible.

3 Les descripteurs de namespaces

3.1 Les informations communes

Dans tous les descripteurs de namespaces, il y a un champ nommé **ns** de type **struct ns_common**. C'est un ensemble de données et opérations communes à tous les namespaces. Cette structure est définie dans **include/linux/ns_common.h** :

```
struct ns_common {
        atomic_long_t stashed;
        const struct proc_ns_operations *ops;
        unsigned int inum;
};
```

A partir d'un pointeur sur ce champ, il est possible de retrouver l'adresse de départ du descripteur de namespace englobant avec la macro **container_of(ptr, type, member)** définie dans **include/linux/kernel.h** et basée sur la macro **offsetof(type, member)** de la librairie C. Pour chaque type de namespace, il existe un service nommé **to_<namespace_type>()** utilisant **container_of()** afin de retrouver l'adresse de début du descripteur. Par exemple, un descripteur d'ipc_ns est retrouvé comme suit :

```
static inline struct ipc_namespace *to_ipc_ns(struct ns_common *ns)
{
        return container_of(ns, struct ipc_namespace, ns);
}
```

La figure 8 schématise l'opération.

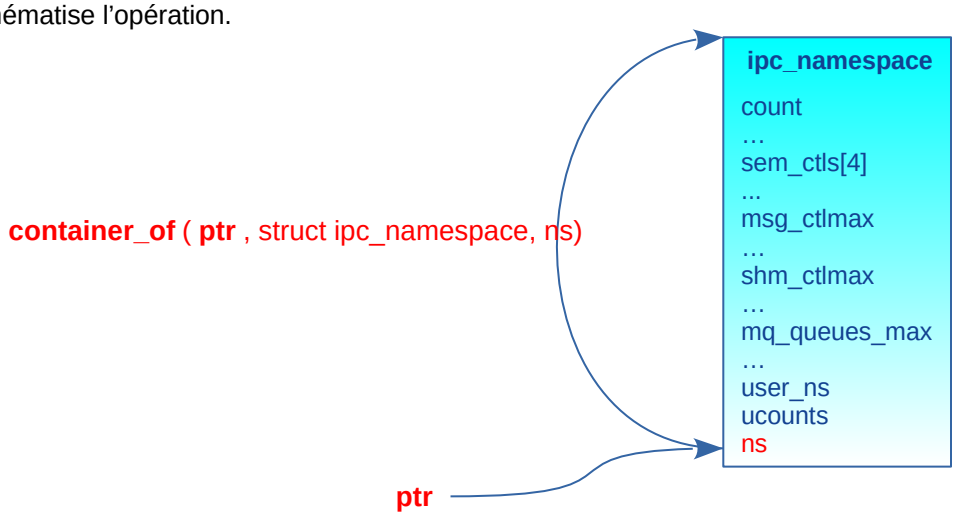

*Fig. 8: Du champ **ns** au descripteur de namespace*

Le champ **inum** est le numéro d'inode unique global au système attribué à la création du namespace. C'est la valeur affichée dans les liens symboliques lorsqu'on liste le répertoire **/proc/<pid>/ns** :

```
$ ls -l /proc/$$/ns
total 0
dr-x--x--x 2 rachid rachid 0 mars   10 18:45 ./
dr-xr-xr-x 9 rachid rachid 0 mars   10 08:36 ../
lrwxrwxrwx 1 rachid rachid 0 mars   10 18:45 cgroup -> 'cgroup:[4026531835]'
lrwxrwxrwx 1 rachid rachid 0 mars   10 18:45 ipc -> 'ipc:[4026531839]'
[...]
lrwxrwxrwx 1 rachid rachid 0 mars   10 18:45 user -> 'user:[4026531837]'
lrwxrwxrwx 1 rachid rachid 0 mars   10 18:45 uts -> 'uts:[4026531838]'
```

Hormis certains namespaces initiaux pour lesquels la valeur est fixée (on le verra par la suite), **inum** est positionné avec le résultat de la fonction **proc_alloc_inum()** définie dans **fs/proc/generic.c** qui retourne une valeur entre **PROC_DYNAMIC_FIRST** (**0xf0000000**) and **0xffffffff** (mécanisme **IDA** du noyau **[5]**). Avec le champ **stashed**, il permet de manipuler les namespaces à travers le système de fichiers **NSFS** que nous reverrons dans le prochain article.

Le type du champ **ops** est défini dans **include/linux/proc_ns.h** :

```
struct proc_ns_operations {
        const char *name;
        const char *real_ns_name;
```

```
        int type;
        struct ns_common *(*get)(struct task_struct *task);
        void (*put)(struct ns_common *ns);
        int (*install)(struct nsproxy *nsproxy, struct ns_common *ns);
        struct user_namespace *(*owner)(struct ns_common *ns);
        struct ns_common *(*get_parent)(struct ns_common *ns);
} __randomize_layout;
```

Les champs ont la signification suivante :

- **name** est le nom du namespace (utilisé pour le nommage dans le répertoire **/proc/<pid>/ns**) ;

- **real_ns_name** est le nom du namespace pour le namespace transitoire « pid_for_children » utilisé lors de la création des pid_ns ;

- **type** est le drapeau **CLONE_NEWXXX** associé au namespace (p. ex. **CLONE_NEWUTS**). Il est utilisé par l'ioctl **NS_GET_NSTYPE** ;

- **get()** incrémente le compteur de références sur le namespace. Opération préalable pour utiliser le namespace ;

- **put()** décrémente le compteur de références. Opération exécutée après utilisation du namespace. Si le compteur atteint la valeur 0, le namespace est désalloué ;

- **install()** est déclenché lors de l'appel système **setns()** ;

- **owner()** retourne le user_ns propriétaire de ce namespace. Il est utilisé par l'ioctl **NS_GET_USERNS** ;

- **get_parent()** retourne le namespace parent. Il ne concerne que les namespaces hiérarchiques user_ns et pid_ns. Il est utilisé par l'ioctl **NS_GET_PARENT**.

Parmi les informations communes, nous verrons aussi que les descripteurs de namespaces ont tous un champ nommé **ucounts**. Il permet de compter les namespaces actifs afin d'en limiter le nombre aux valeurs maximales **[6]** exportées dans les fichiers sous **/proc/sys/user** (cf. **man 7 namespaces**) :

```
$ ls -l /proc/sys/user
total 0
-rw-r--r-- 1 root root 0 avril   2 09:29 max_cgroup_namespaces
[...]
-rw-r--r-- 1 root root 0 avril   2 09:29 max_ipc_namespaces
-rw-r--r-- 1 root root 0 avril   2 09:29 max_mnt_namespaces
-rw-r--r-- 1 root root 0 avril   2 09:29 max_net_namespaces
-rw-r--r-- 1 root root 0 avril   2 09:29 max_pid_namespaces
-rw-r--r-- 1 root root 0 avril   2 09:29 max_user_namespaces
-rw-r--r-- 1 root root 0 avril   2 09:29 max_uts_namespaces
```

3.2 Virtualisation de PROCFS

Le système de fichiers **PROCFS** (monté sur **/proc**) permet de consulter voire modifier des informations dans le noyau. Certaines de ces informations sont globales et par conséquent restent inchangées peu importe les namespaces à partir desquels nous les consultons. Par contre, d'autres informations sont stockées ou liées aux descripteurs de namespaces. Dans ce cas, les valeurs exportées dépendent des namespaces à partir desquels nous les consultons. **On parle alors de fichiers ou répertoires virtualisés**.

3.3 Sécurisation des structures

La plupart des structures de données présentées sont étiquetées avec la macro **__randomize_layout**. C'est une extension ajoutée sous forme de greffon, au compilateur **GCC** dans la chaîne de compilation du noyau Linux pour réordonner les champs des structures de manière aléatoire pendant la compilation. Le but est de contrer les attaques pirates qui se basent sur la connaissance de l'ordre des champs dans les structures **[11]**. Cela permet de définir un nouvel attribut nommé **randomize_layout** que le noyau encapsule avec cette macro dans le fichier **include/linux/compiler-gcc.h** :

```
#define __randomize_layout __attribute__((randomize_layout))
```

3.4 Le user_ns

Introduit dans Linux 2.6.23 et complété dans Linux 3.8 [7], ce namespace est décrit par la structure **user_namespace** définie dans le fichier d'entête **include/linux/user_namespace.h** :

```
struct user_namespace {
        struct uid_gid_map     uid_map;
        struct uid_gid_map     gid_map;
[...]
        atomic_t               count;
        struct user_namespace  *parent;
        int                    level;
        kuid_t                 owner;
        kgid_t                 group;
        struct ns_common       ns;
        unsigned long          flags;
[...]
        struct work_struct     work;
[...]
        struct ucounts         *ucounts;
        int ucount_max[UCOUNT_COUNTS];
} __randomize_layout;
```

Les champs **uid_map** et **gid_map** définissent les mappings utilisateur. Ils sont respectivement exposés en espace utilisateur via les fichiers **/proc/<pid>/uid_map** et **/proc/<pid>/gid_map**. De même, le champ **flags** permet de gérer l'autorisation d'appel à **setgroups()** via le fichier **/proc/<pid>/setgroups** utilisé lors du mapping des identifiants d'utilisateurs et groupes. Ce sera détaillé dans un article consacré au user_ns.

Ce namespace étant hiérarchique, il y a les champs **parent** et **level** comme indiqué précédemment.

Les champs **owner** et **group** sont respectivement les identifiants effectifs d'utilisateur et de groupe de la tâche à l'origine de la création de ce user_ns.

Le champ **ucount_max[]** gère le comptage du nombre de namespaces par types (une entrée pour chaque) pour ce user_ns [6].

3.5 Le pid_ns

Introduit dans Linux 2.6.24 [7], ce namespace est décrit par la structure **pid_namespace** définie dans le fichier d'entête **include/linux/pid_namespace.h** :

```
struct pid_namespace {
        struct kref kref;
        struct idr idr;
        struct rcu_head rcu;
        unsigned int pid_allocated;
        struct task_struct *child_reaper;
        struct kmem_cache *pid_cachep;
        unsigned int level;
        struct pid_namespace *parent;
[...]
        struct user_namespace *user_ns;
        struct ucounts *ucounts;
        struct work_struct proc_work;
        kgid_t pid_gid;
        int hide_pid;
        int reboot;    /* group exit code if this pidns was rebooted */
        struct ns_common ns;
} __randomize_layout;
```

Comme le user_ns, c'est un namespace hiérarchique (d'où la présence des champs **parent** et **level**). On notera cependant que le nombre maximum de niveau est limité pour les pid_ns. Il est défini avec la constante interne **MAX_PID_NS_LEVEL** (égale à 32 dans **kernel/pid_namespace.c**). Cette valeur permet de limiter la taille du tableau **numbers[]** dans la structure **pid** vue plus haut.

Le champ **pid_cachep** est le cache dans lequel sont alloués les structures **pid** des tâches associées à ce namespace. L'initialisation du cache définit une taille de structure par rapport au niveau du pid_ns (champ **level**). Ceci afin de dimensionner au plus juste le nombre d'entrées **upid** dans son champ **numbers[]**. La

formule utilisée est :

```
len = sizeof(struct pid) + level * sizeof(struct upid);
```

Le champ **idr** permet d'allouer les identifiants de processus de manière séquentielle dans le pid_ns à l'aide du mécanisme **IDR** vu plus haut.

Le champ **pid_allocated** est le nombre courant de tâches dans le namespace.

Le champ **child_reaper** pointe sur le descripteur de processus (c.-à-d. processus **init**) qui joue de rôle de faucheur des processus orphelins. C'est le premier processus créé dans le namespace.

Le champ **user_ns** pointe sur le descripteur du user_ns propriétaire c'est-à-dire le user_ns associé au processus qui a créé ce namespace.

Le champ **hide_pid** correspond à l'option **hidepid** de montage du système de fichiers **/proc** et peut prendre les valeurs suivantes : **HIDEPID_OFF** (0), **HIDEPID_NO_ACCESS** (1), **HIDEPID_INVISIBLE** (2). La documentation du noyau **[8]** décrit la fonction de chacune de ces valeurs.

Le champ **reboot** est positionné avec l'identifiant du signal **SIGHUP** ou **SIGINT** en fonction des paramètres passés à l'appel système **reboot()**. Des détails supplémentaires seront donnés dans l'article relatif aux pid_ns.

3.6 Le net_ns

Introduit dans Linux 2.6.24 et complété dans Linux 2.6.29 **[7]**, ce namespace est décrit par la structure **net** définie dans **include/net/net_namespace.h** :

```
struct net {
        refcount_t              passive;        /* To decide when the network
                                                 * namespace should be freed.
                                                 */
        refcount_t              count;          /* To decided when the network
                                                 *  namespace should be shut down.
[...]
        struct list_head        list;           /* list of network namespaces */
[...]
        struct user_namespace   *user_ns;       /* Owning user namespace */
        struct ucounts          *ucounts;
        spinlock_t              nsid_lock;
        struct idr              netns_ids;

        struct ns_common        ns;
        struct proc_dir_entry           *proc_net;
        struct proc_dir_entry           *proc_net_stat;
[...]
        struct list_head        dev_base_head;
        struct hlist_head       *dev_name_head;
        struct hlist_head       *dev_index_head;
[...]
        int                     ifindex;
[...]
        struct net_device       *loopback_dev;          /* The loopback */
        struct netns_core       core;
        struct netns_mib        mib;
        struct netns_packet     packet;
        struct netns_unix       unx;
        struct netns_nexthop    nexthop;
        struct netns_ipv4       ipv4;
[...]
} __randomize_layout;
```

Le champ **list** est un lien dans une liste globale de net_ns. Cela sert à l'algorithmique interne pendant les phases de désallocation des namespaces.

Les champ **proc_net** ainsi que **proc_net_stat** concernent les fichiers dans les répertoires **/proc/net** ainsi que les fichiers de statistiques dans **/proc/net/stat** (cf. **man 5 proc**).

Les interfaces assignées au namespace sont chaînées dans différentes listes référencées par les champs **dev_base_head**, **dev_name_head** et **dev_index_head**. Elles permettent de retrouver une interface par son nom ou son index.

Le champ **ifindex** est le dernier index attribué aux interfaces installées dans ce net_ns. Sa **valeur commence à 1 et est réservée à l'interface loopback**. Les index ne sont pas forcément séquentiels car lorsqu'une interface migre dans un namespace, elle ne change son index que s'il est déjà attribué dans le net_ns cible. De même si une interface quitte le namespace, les interfaces restantes ne sont pas renumérotées.

Le champ **loopback_dev** pointe sur l'interface **loopback**. Comme on l'a dit précédemment, tout net_ns a une interface **loopback** dédiée.

Les structures suivantes nommées **netns_XXX** sont toutes les fonctions réseau virtualisées (table de routage, filtres pare-feu...).

La structure **net_device** décrivant un périphérique réseau et définie dans **include/linux/netdevice.h**, contient un champ nommé **nd_net** de type **possible_net_t** pour référencer le net_ns auquel appartient le périphérique :

```
struct net_device {
        char                    name[IFNAMSIZ];
        struct hlist_node       name_hlist;
[...]
        struct list_head        dev_list;
[...]
        int                     ifindex;
[...]
        unsigned int            flags;
[...]

        struct hlist_node       index_hlist;
[...]
        netdev_features_t       features;
[...]
possible_net_t                  nd_net; /* Network namespace this network device is inside */
[...]

};
```

A l'initialisation, le driver d'interface réseau alloue (via le service **alloc_netdev_mqs()** ou équivallent) et initialise une structure **net_device** **[9]**. Cette dernière est chainée dans les listes du namespaces associé avec les champs **name_hlist**, **dev_list** et **index_hlist**.

La structure **possible_net_t** définie dans **include/net/net_namespace.h** est une coquille au-dessus d'un pointeur sur la structure **net** décrivant le namespace :

```
typedef struct {
#ifdef CONFIG_NET_NS
        struct net *net;
#endif
} possible_net_t;
```

La figure 9 représente une tâche associée à ses namespaces avec une vue simplifiée de deux interfaces réseau attachées à son net_ns. Le premier descripteur **net_device** de la liste correspond bien-entendu à l'interface **loopback**.

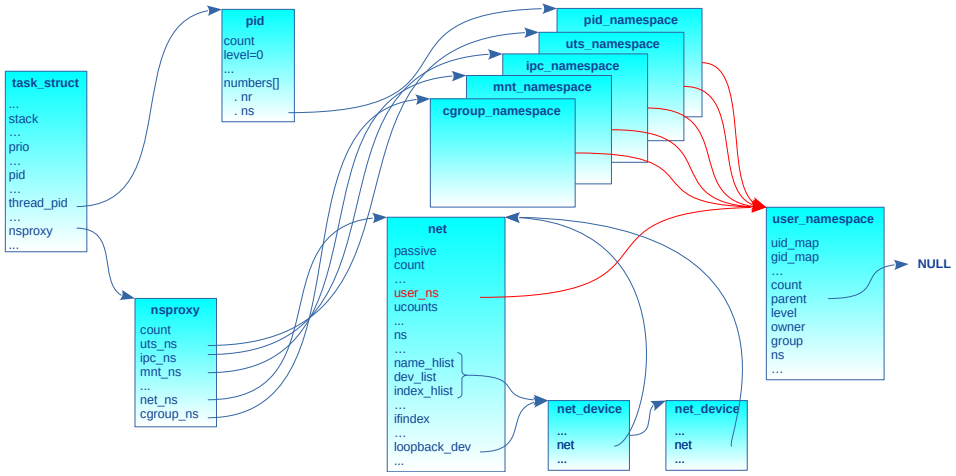

Fig. 9: Les interfaces réseau dans un net_ns

Précisons comme nous l'avons mentionné dans un précédent article, que ce qui détermine la possibilité pour une interface réseau de migrer ou pas dans un autre namespace est la présence du drapeau **NETIF_F_NETNS_LOCAL** positionné dans le champ **features** de la structure **net_device** dans la fonction d'initialisation du driver. Par exemple, l'interface **loopback** s'initialise comme suit dans **drivers/net/loopback.c** car elle ne peut pas migrer :

```
        dev->features         = NETIF_F_SG | NETIF_F_FRAGLIST
[...]
                | NETIF_F_NETNS_LOCAL
                | NETIF_F_VLAN_CHALLENGED
                | NETIF_F_LOOPBACK;
```

Sur demande de migration d'une interface réseau, la fonction suivante est appelée dans **net/core/dev.c**. Elle sort en erreur si le drapeau est positionné :

```
int dev_change_net_namespace(struct net_device *dev, struct net *net, const char *pat)
{
[...]
        /* Don't allow namespace local devices to be moved. */
        err = -EINVAL;
        if (dev->features & NETIF_F_NETNS_LOCAL)
                goto out;
[...]
```

Le livre [10] donne plus de détails sur le sujet.

3.7 L'uts_ns

Introduit dans Linux 2.6.19 [7], ce namespace est décrit par la structure **uts_namespace** définie dans **include/linux/utsname.h** :

```
struct uts_namespace {
        struct kref kref;
        struct new_utsname name;
        struct user_namespace *user_ns;
        struct ucounts *ucounts;
        struct ns_common ns;
} __randomize_layout;
```

Le champ **name** est une structure **new_utsname** définie dans **include/uapi/linux/utsname.h** :

```
#define __NEW_UTS_LEN 64
[...]
struct new_utsname {
        char sysname[__NEW_UTS_LEN + 1];
        char nodename[__NEW_UTS_LEN + 1];
[...]
        char domainname[__NEW_UTS_LEN + 1];
};
```

Ce namespace est le plus simple de tous. Le nom de nœud et le nom de domaine 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()**.

3.8 L'ipc_ns

Introduit dans Linux 2.6.19 [7] pour les IPC système V, il a été complété avec les queues de messages POSIX dans la version 2.6.30. Ce namespace est décrit par la structure **ipc_namespace** définie dans **include/linux/ipc_namespace.h** :

```
struct ipc_namespace {
        refcount_t      count;
        struct ipc_ids ids[3];

        int             sem_ctls[4];
        int             used_sems;

        unsigned int    msg_ctlmax;
[...]
        atomic_t        msg_hdrs;

        size_t          shm_ctlmax;
[...]
```

```
        int             shm_ctlmni;
[...]
        /* The kern_mount of the mqueuefs sb.  We take a ref on it */
        struct vfsmount       *mq_mnt;
[...]
        unsigned int    mq_msgsize_default;

        /* user_ns which owns the ipc ns */
        struct user_namespace *user_ns;
        struct ucounts *ucounts;

        struct ns_common ns;
} __randomize_layout;
```

Le champ **ids[]** a trois entrées pour mémoriser les identifiants d'IPC système V des trois types possibles :

```
#define IPC_SEM_IDS    0 /* Semaphores */
#define IPC_MSG_IDS    1 /* Message queues */
#define IPC_SHM_IDS    2 /* Shared memory */
```

Comme les identifiants de tâches vus plus haut, ils sont alloués par le mécanisme **IDR**.

Les noms des champs qui suivent mettent en exergue quatre parties : la première dédiée aux sémaphores (**sem_ctls[]** et **used_sems**), la seconde dédiée aux queues de messages (**msg_ctlmax** à **msg_hdrs**), la troisième dédiée aux segments de mémoire partagée (**shm_ctlmax** à **shm_ctlmni**) et la dernière dédiées aux queues de messages POSIX (**mq_mnt** à **mq_msgsize_default**). Ces valeurs sont pour la plupart exportées dans les répertoires **/proc/sys/kernel**, **/proc/sysvipc** et **/proc/sys/fs/mqueue**. (cf. **man 7 namespaces**).

Le champ **mq_mnt** est une référence sur le système de fichiers **mqueuefs** normalement monté sur **/dev/mqueue** en espace utilisateur. C'est l'endroit où sont créés les fichiers associés aux queues de messages POSIX.

Dans l'article dédié à l'ipc_ns, nous expliquerons la raison pour laquelle ce namespace ne concerne pas les sémaphores et segments de mémoire partagée POSIX.

3.9 Le mount_ns

Introduit dans la version 2.4.19 **[7]**, c'est le premier namespace de Linux. Il est décrit par la structure **mnt_namespace** définie dans **fs/mount.h** :

```
struct mnt_namespace {
        atomic_t                count;
        struct ns_common        ns;
        struct mount * root;
        struct list_head        list;
        struct user_namespace   *user_ns;
        struct ucounts          *ucounts;
        u64                     seq;    /* Sequence number to prevent loops */
        wait_queue_head_t poll;
        u64 event;
        unsigned int            mounts; /* # of mounts in the namespace */
        unsigned int            pending_mounts;
} __randomize_layout;
```

3.10 Le cgroup_ns

Introduit dans Linux 4.6 **[7]**, c'est le dernier né des namespaces dans Linux même si nous verrons par la suite que de nouveaux namespaces sont en préparation ou en cours de discussion. Il est décrit par la structure **cgroup_namespace** définie dans **include/linux/cgroup.h** :

```
struct cgroup_namespace {
        refcount_t              count;
        struct ns_common        ns;
        struct user_namespace   *user_ns;
        struct ucounts          *ucounts;
        struct css_set          *root_cset;
};
```

Le champ **root_cset** référence les états des sous-systèmes (**C**group **S**ubsystem **S**tate) et donne aux tâches associées au namespace une vue spécifique de la hiérarchie. Dans un contexte de conteneurs, cela sert à

limiter la vue sur une partie de l'arborescence globale des cgroups (cf. **Documentation/cgroups/namespace.txt**).

4 Les namespaces initiaux

Au démarrage, les processus sont associés aux namespaces initiaux. **Ils constituent le système hôte dans un environnement de conteneurs**. Certains sont définis et initialisés de manière statique tandis que les autres le sont de manière dynamique.

Les namespaces initiaux ne devant jamais être désalloués, leur champ « compteur de références » (**count** ou **kref**) est initialisé à une valeur supérieure ou égale à 1.

Citons le user_ns initial en premier car il est référencé en tant que propriétaire de tous les autres namespaces initiaux via leur champ **user_ns**. Il est défini de manière statique dans **kernel/user.c** :

```
struct user_namespace init_user_ns = {
        .uid_map = { [...] },
        .gid_map = { [...] },
[...]
        .count = ATOMIC_INIT(3),
        .owner = GLOBAL_ROOT_UID,
        .group = GLOBAL_ROOT_GID,
        .ns.inum = PROC_USER_INIT_INO,
#ifdef CONFIG_USER_NS
        .ns.ops = &userns_operations,
#endif
        .flags = USERNS_INIT_FLAGS,
[...]
};
```

Tout champ n'apparaissant pas dans l'initialisation statique est par défaut initialisé à 0. Le champ **parent** n'apparaissant pas, il est donc initialisé à 0 (c.-à-d. **NULL**) pour indiquer qu'on est au sommet (racine) de la hiérarchie des user_ns.

Le mount_ns initial est créé dynamiquement à l'initialisation du système (étiquette **__init**) via l'appel à la fonction **init_mount_tree()** dans le fichier **fs/namespace.c** :

```
static void __init init_mount_tree(void)
{
        struct vfsmount *mnt;
        struct mount *m;
        struct mnt_namespace *ns;
        struct path root;

        mnt = vfs_kern_mount(&rootfs_fs_type, 0, "rootfs", NULL);
[...]
        ns = alloc_mnt_ns(&init_user_ns, false);
        if (IS_ERR(ns))
                panic("Can't allocate initial namespace");
        m = real_mount(mnt);
        m->mnt_ns = ns;
        ns->root = m;
        ns->mounts = 1;
        list_add(&m->mnt_list, &ns->list);
        init_task.nsproxy->mnt_ns = ns;
        get_mnt_ns(ns);

        root.mnt = mnt;
        root.dentry = mnt->mnt_root;
        mnt->mnt_flags |= MNT_LOCKED;

        set_fs_pwd(current->fs, &root);
        set_fs_root(current->fs, &root);
}
```

L'appel à la fonction **get_mnt_ns()** permet d'incrémenter le compteur de références du namespace de sorte à le rendre persistant (c.-à-d. non désallouable).

Les autres namespaces initiaux sont créés de manière statique.

Le cgroup_ns initial est défini dans **kernel/cgroup/cgroup.c** :

```
struct cgroup_namespace init_cgroup_ns = {
```

```
        .count          = REFCOUNT_INIT(2),
        .user_ns        = &init_user_ns,
        .ns.ops         = &cgroupns_operations,
        .ns.inum        = PROC_CGROUP_INIT_INO,
        .root_cset      = &init_css_set,
};
```

L'ipc_ns initial est défini dans **ipc/msgutil.c** :

```
struct ipc_namespace init_ipc_ns = {
        .count          = REFCOUNT_INIT(1),
        .user_ns = &init_user_ns,
        .ns.inum = PROC_IPC_INIT_INO,
#ifdef CONFIG_IPC_NS
        .ns.ops = &ipcns_operations,
#endif
};
```

Le net_ns initial est défini dans **net/core/net_namespace.c** :

```
struct net init_net = {
        .count          = REFCOUNT_INIT(1),
        .dev_base_head = LIST_HEAD_INIT(init_net.dev_base_head),
[...]
};
```

L'initialisation est complétée par des fonctions étiquetées **__init** et appelées lors du démarrage du système (cf. **include/linux/init.h**) pour par exemple enregistrer l'interface **loopback** de sorte à ce qu'elle soit la première dans le net_ns initial.

L'uts_ns initial est défini dans **init/version.c** :

```
struct uts_namespace init_uts_ns = {
        .kref = KREF_INIT(2),
        .name = {
                .sysname        = UTS_SYSNAME,
                .nodename       = UTS_NODENAME,
[...]
                .domainname     = UTS_DOMAINNAME,
        },
        .user_ns = &init_user_ns,
        .ns.inum = PROC_UTS_INIT_INO,
#ifdef CONFIG_UTS_NS
        .ns.ops = &utsns_operations,
#endif
};
```

Le pid_ns initial est défini dans **kernel/pid.c** :

```
struct pid init_struct_pid = {
        .count          = REFCOUNT_INIT(1),
[...]
        .level          = 0,
        .numbers        = { {
                .nr             = 0,
                .ns             = &init_pid_ns,
        }, }
};
[...]
struct pid_namespace init_pid_ns = {
        .kref = KREF_INIT(2),
        .idr = IDR_INIT(init_pid_ns.idr),
        .pid_allocated = PIDNS_ADDING,
        .level = 0,
        .child_reaper = &init_task,
        .user_ns = &init_user_ns,
        .ns.inum = PROC_PID_INIT_INO,
#ifdef CONFIG_PID_NS
        .ns.ops = &pidns_operations,
#endif
};
```

Comme pour le user_ns, le champ **parent** (n'apparaissant pas) est à 0 pour indiquer qu'on est au sommet de la hiérarchie.

Le lecteur perspicace notera que le champ **nr** de **init_struct_pid** est initialisé à 0 alors que nous avons dit plus haut que l'allocation des valeurs pour ce champ utilise le mécanisme **IDR** avec une plage qui commence à 1. Cela reste tout de même cohérent car le second processus créé provoquera l'appel de la fonction **alloc_pid()** qui obtiendra le premier identifiant non utilisé dans la plage autrement dit la valeur 1. La

particularité ici est que seul le pid_ns initial a une première tâche d'identifiant 0 et la seconde d'identifiant 1. Les pid_ns descendants ont l'identifiant 1 pour la première tâche.

Les numéros d'inode des namespaces initiaux sont prédéfinis dans **include/linux/proc_ns.h** :

```
enum {
        PROC_ROOT_INO         = 1,
        PROC_IPC_INIT_INO     = 0xEFFFFFFFU,
        PROC_UTS_INIT_INO     = 0xEFFFFFFEU,
        PROC_USER_INIT_INO    = 0xEFFFFFFDU,
        PROC_PID_INIT_INO     = 0xEFFFFFFCU,
        PROC_CGROUP_INIT_INO  = 0xEFFFFFFBU,
};
```

Cette énumération ne contient pas de numéros pour le net_ns et le mount_ns. Leurs inodes sont alloués dynamiquement au démarrage du système. La connaissance de ces valeurs permet de savoir si un processus est associé aux namespaces initiaux lorsqu'on liste le répertoire **/proc/<pid>/ns**. **C'est une astuce qui n'a rien d'officiel** ! Nous listons ci-dessous les numéros d'inodes des namespaces du processus **init** (pid 1) et vérifions que les valeurs affichées sont égales au contenu de l'énumération vu que ce processus est associé aux namespaces initiaux :

```
# ls -l /proc/1/ns
total 0
lrwxrwxrwx [...] cgroup -> 'cgroup:[4026531835]'    # 0xEFFFFFFBU (PROC_CGROUP_INIT_INO)
lrwxrwxrwx [...] ipc -> 'ipc:[4026531839]'          # 0xEFFFFFFFU (PROC_IPC_INIT_INO)
lrwxrwxrwx [...] mnt -> 'mnt:[4026531840]'
lrwxrwxrwx [...] net -> 'net:[4026531992]'
lrwxrwxrwx [...] pid -> 'pid:[4026531836]'          # 0xEFFFFFFCU (PROC_PID_INIT_INO)
lrwxrwxrwx [...] pid_for_children -> 'pid:[4026531836]'
lrwxrwxrwx [...] user -> 'user:[4026531837]'        # 0xEFFFFFFDU (PROC_USER_INIT_INO)
lrwxrwxrwx [...] uts -> 'uts:[4026531838]'          # 0xEFFFFFFEU (PROC_UTS_INIT_INO)
```

La structure **nsproxy** référençant les namespaces initiaux est définie dans **kernel/nsproxy.c** :

```
struct nsproxy init_nsproxy = {
        .count                  = ATOMIC_INIT(1),
        .uts_ns                         = &init_uts_ns,
#if defined(CONFIG_POSIX_MQUEUE) || defined(CONFIG_SYSVIPC)
        .ipc_ns                         = &init_ipc_ns,
#endif
        .mnt_ns                         = NULL,
        .pid_ns_for_children    = &init_pid_ns,
#ifdef CONFIG_NET
        .net_ns                         = &init_net,
#endif
#ifdef CONFIG_CGROUPS
        .cgroup_ns              = &init_cgroup_ns,
#endif
};
```

Le champ **mnt_ns** est initialisé à **NULL** mais nous avons vu plus haut que la fonction d'initialisation du mount_ns initial positionne ce champ avec l'adresse de la structure du namespace allouée dynamiquement :

```
static void __init init_mount_tree(void)
{
[...]
        ns = alloc_mnt_ns(&init_user_ns, false);
[...]
        init_task.nsproxy->mnt_ns = ns;
[...]
}
```

Le descripteur de la première tâche du système est **init_task**. Elle est appelée **swapper**. Elle est définie dans **init/init_task.c**. Son champ **nsproxy** pointe évidemment sur **init_nsproxy** :

```
struct task_struct init_task [...] = {
[...]
        .state          = 0,
        .stack          = init_stack,
        .usage          = REFCOUNT_INIT(2),
        .flags          = PF_KTHREAD,
        .prio           = MAX_PRIO - 20,
        .static_prio    = MAX_PRIO - 20,
        .normal_prio    = MAX_PRIO - 20,
        .policy                 = SCHED_NORMAL,
[...]
        .comm           = INIT_TASK_COMM,    // "swapper"
[...]
        .thread_pid     = &init_struct_pid,
```

```
[...]
        .nsproxy        = &init_nsproxy,
[...]
};
```

La figure 10 illustre les namespaces initiaux au démarrage du système.

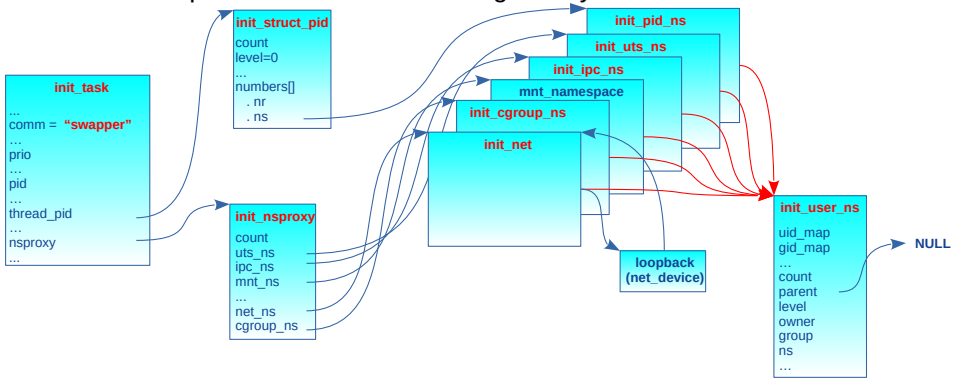

Fig. 10: Les namespaces initiaux

Conclusion

Cet article est indéniablement sorti des sentiers battus en prenant le risque d'une plongée dans les sources du noyau de Linux. Nous avons tenté de ne pas être trop rébarbatifs dans cette présentation des structures de données. C'est une étape préalable qui nous est apparue nécessaire afin d'expliquer le fonctionnement interne des appels système dans le prochain article.

Références

[1] Premier post de L. Torvalds sur usenet : **http://opendotdotdot.blogspot.com/2006/03/linus-torvalds-first-usenet-posting.html**

[2] Linux kernel design patterns : **https://lwn.net/Articles/336224/**

[3] Slab allocation : **https://en.wikipedia.org/wiki/Slab_allocation**

[4] PID namespaces in the 2.6.24 kernel : **https://lwn.net/Articles/259217/**

[5] ID allocation : **https://www.kernel.org/doc/html/latest/core-api/idr.html**

[6] sysctl limits for namespaces : **https://lwn.net/Articles/694968/**

[7] Namespaces overview : **https://lwn.net/Articles/531114/**

[8] Options de montage de /proc : **https://www.kernel.org/doc/Documentation/filesystems/proc.txt**

[9] Network Devices : **https://www.kernel.org/doc/Documentation/networking/netdevices.txt**

[10] Rosen R., « Linux Kernel Networking: Implementation and Theory », Apress, 2014

[11] Randomizing structure layout : **https://lwn.net/Articles/722293/**