# Le fonctionnement des namespaces dans le noyau

**Rachid Koucha**
**[Ingénieur développement logiciel]**

***Après la présentation des structures de données supportant les namespaces, ce nouvel opus se consacre à la partie immergée dans le noyau des appels système.***

# Table des matières

# Table of Contents

# 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°245 du mois de février 2021 :

# Introduction

Nous continuons notre balade hors des sentiers battus dans le code source de **Linux 5.3.0** afin de nous pencher sur le fonctionnement des appels système dans le noyau.

# 1 NSFS

Le **V**irtual **F**ile system **S**witch (**VFS**) **[1]** de Linux est une couche d'abstraction qui présente à l'utilisateur un ensemble d'opérations génériques (p. ex. **`open()`**, **`close()`**, **`ioctl()`**...) en masquant les spécificités des différents types de systèmes de fichiers **[2]**. Le **N**ame**S**pace **F**ile **S**ystem (**NSFS**) est l'un d'eux et il est dédié aux namespaces **[3]**.

## 1.1 Intéractions avec PROCFS

**NSFS gère plus précisément les cibles des liens symboliques du répertoire `/proc/<pid>/ns`** . Son code source se trouve dans **`fs/nsfs.c`**. Il n'est pas monté explicitement par l'utilisateur mais de manière interne lors de l'initialisation du noyau (fonction **`nsfs_init()`** étiquetée avec **`__init`**) :

```
static struct file_system_type nsfs = {
        .name = "nsfs",
        .init_fs_context = nsfs_init_fs_context,
        .kill_sb = kill_anon_super,
};

void __init nsfs_init(void)
{
        nsfs_mnt = kern_mount(&nsfs);
[...]
        nsfs_mnt->mnt_sb->s_flags &= ~SB_NOUSER;
}
```

La figure 1 schématise la gestion des fichiers relatifs aux namespaces à travers **PROCFS** d'une part (les liens symboliques) et **NSFS** d'autre part (cibles des liens symboliques).

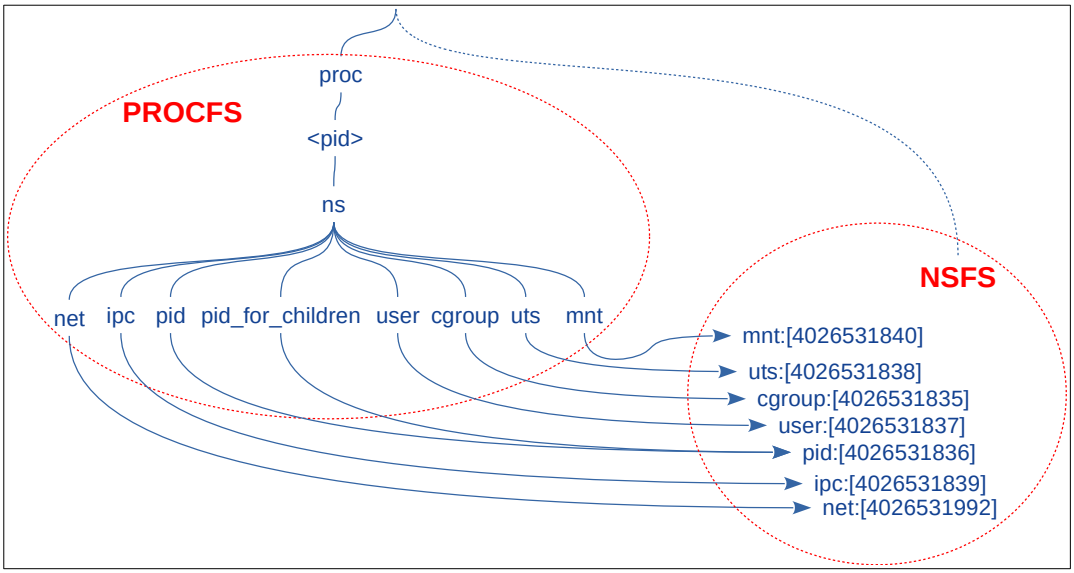


*Fig. 1: **NSFS** versus **PROCFS***

Notre programme **`linfo`** prend en paramètre le chemin d'un lien symbolique et affiche des informations sur le lien et sa cible en s'appuyant respectivement sur les appels système **`lstat()`** et **`stat()`** :

```
int main(int ac, char *av[])
{
[...]
```

```
  // Get information on the symbolic link
  rc = lstat(av[1], &stl);
[...]
  // Get the information on the target
  rc = stat(av[1], &stt);
[...]
  // Get the name of the target
  memset(lname, 0, sizeof(lname));
  ssz = readlink(av[1], lname, sizeof(lname));
[...]
  printf("Symbolic link:\n"
         "\tName: %s\n"
         "\tRights: 0%o\n"
         "\tDevice (major/minor): 0x%x/0x%x\n"
         "\tInode: 0x%x (%u)\n"
         "Target:\n"
         "\tName: %s\n"
         "\tRights: 0%o\n"
         "\tDevice (major/minor): 0x%x/0x%x\n"
         "\tInode: 0x%x (%u)\n"
         ,
         av[1],
         stl.st_mode & 0777,
         major(stl.st_dev), minor(stl.st_dev),
         (unsigned int)(stl.st_ino), (unsigned int)(stl.st_ino),
         lname,
         stt.st_mode & 0777,
         major(stt.st_dev), minor(stt.st_dev),
         (unsigned int)(stt.st_ino), (unsigned int)(stt.st_ino)
        );
```

Lancé avec le lien sur le net_ns associé au shell courant, **linfo** affiche d'abord les informations retournées par **procfs** (le lien symbolique) puis les informations retournées par **NSFS** (la cible du lien) :

```
$ ./linfo /proc/self/ns/net
Symbolic link:
       Name: /proc/self/ns/net
       Rights: 0777
       Device (major/minor): 0x0/0x5
       Inode: 0x14a61 (84577)
Target:
       Name: net:[4026531992]
       Rights: 0444
       Device (major/minor): 0x0/0x4
       Inode: 0xf0000098 (4026531992)
```

Pour la cible, on retrouve bien le numéro d'inode affiché dans le nom du fichier par la commande **ls** dans le répertoire :

```
$ ls -l /proc/self/ns/net
lrwxrwxrwx 1 rachid rachid 0 avril   6 11:43 /proc/self/ns/net -> 'net:[4026531992]'
```

Notre programme **fsinfo** reçoit en paramètre le nom d'un fichier et s'appuie sur l'appel système **statfs()** pour afficher des informations à propos de son système de fichiers :

```
int main(int ac, char *av[])
{
[...]
  // Get information on the file system
  rc = statfs(av[1], &st);
[...]
  printf("Type              : 0x%lx (%s)\n"
         "Block size       : %lu\n"
         "Total blocks     : %lu\n"
         "Total free blocks: %lu\n"
         "Total files      : %lu\n"
         "Max name length  : %lu\n"
         "Mount flags      : 0x%lx (%s)\n"
         ,
         (unsigned long)(st.f_type), get_fstype((unsigned long)(st.f_type)),
         (unsigned long)(st.f_bsize),
         (unsigned long)(st.f_blocks),
         (unsigned long)(st.f_bfree),
         (unsigned long)(st.f_files),
         (unsigned long)(st.f_namelen),
         (unsigned long)(st.f_flags), get_mntflags((unsigned long)(st.f_flags))
        );
[...]
```

Pour la cible d'un lien symbolique sur un namespace, il retourne bien le type **NSFS** :

```
$ ./fsinfo /proc/self/ns/pid
Type             : 0x6e736673 (NSFS)
Block size       : 4096
Total blocks     : 0
Total free blocks: 0
Total files      : 0
Max name length  : 255
Mount flags      : 0x20 (REMOUNT)
```

Pour le répertoire où se trouve ce même lien, c'est le type **PROCFS** :

```
$ ./fsinfo /proc/self/ns
Type             : 0x9fa0 (PROCFS)
Block size       : 4096
Total blocks     : 0
Total free blocks: 0
Total files      : 0
Max name length  : 255
Mount flags      : 0x102e (NODEV NOEXEC NOSUID REMOUNT BIND)
```

Sur l'appel système **readlink()** (utilisé dans le programme **linfo** pour récupérer la cible du lien symbolique) dans **/proc/<pid>/ns**, le système de fichiers **PROCFS** appelle le service interne **ns_get_name()** de **NSFS** pour retourner le nom des cibles (cf. fonction **proc_ns_readlink()** dans **fs/proc/namespaces.c**) :

```
static int proc_ns_readlink(struct dentry *dentry, char __user *buffer, int buflen)
{
        struct inode *inode = d_inode(dentry);
        const struct proc_ns_operations *ns_ops = PROC_I(inode)->ns_ops;
        struct task_struct *task;
        char name[50];
        int res = -EACCES;

        task = get_proc_task(inode);
[...]
        if (ptrace_may_access(task, PTRACE_MODE_READ_FSCREDS)) {
                res = ns_get_name(name, sizeof(name), task, ns_ops);
                if (res >= 0)
                        res = readlink_copy(buffer, buflen, name);
        }
        put_task_struct(task);
        return res;
}
```

Définie dans le fichier **fs/nsfs.c**, la fonction **ns_get_name()** construit le nom de la cible avec les informations du descripteur de namespace (c.-à-d. les champs **name**, **real_name** et **inum** de la structure **ns_common** vue dans l'article précédent) :

```
int ns_get_name(char *buf, size_t size, struct task_struct *task,
                        const struct proc_ns_operations *ns_ops)
{
        struct ns_common *ns;
        int res = -ENOENT;
        const char *name;
        ns = ns_ops->get(task);
        if (ns) {
                name = ns_ops->real_ns_name ? : ns_ops->name;
                res = snprintf(buf, size, "%s:[%u]", name, ns->inum);
                ns_ops->put(ns);
        }
        return res;
}
```

## 1.2 Les inodes

Lorsque la cible d'un lien symbolique dans **/proc/<pid>/ns** est demandée, l'opération **proc_ns_get_link()** dans **fs/proc/namespace.c** est déclenchée. Cette dernière appelle la fonction **ns_get_path()** dans **fs/nsfs.c** qui finit par appeler **__ns_get_path()** pour allouer l'inode et le dentry associé :

```
static void *__ns_get_path(struct path *path, struct ns_common *ns)
{
        struct vfsmount *mnt = nsfs_mnt;
        struct dentry *dentry;
        struct inode *inode;
        unsigned long d;
```

```
        rcu_read_lock();
        d = atomic_long_read(&ns->stashed);
        if (!d)
                goto slow;
        dentry = (struct dentry *)d;
        if (!lockref_get_not_dead(&dentry->d_lockref))
                goto slow;
        rcu_read_unlock();
        ns->ops->put(ns);
got_it:
        path->mnt = mntget(mnt);
        path->dentry = dentry;
        return NULL;
slow:
        rcu_read_unlock();
        inode = new_inode_pseudo(mnt->mnt_sb);
[...]
        inode->i_ino = ns->inum;
        inode->i_mtime = inode->i_atime = inode->i_ctime = current_time(inode);
        inode->i_flags |= S_IMMUTABLE;
        inode->i_mode = S_IFREG | S_IRUGO;
        inode->i_fop = &ns_file_operations;
        inode->i_private = ns;

        dentry = d_alloc_anon(mnt->mnt_sb);
[...]
        d_instantiate(dentry, inode);
        dentry->d_fsdata = (void *)ns->ops;
        d = atomic_long_cmpxchg(&ns->stashed, 0, (unsigned long)dentry);
[...]
        goto got_it;
}
```

L'inode ainsi alloué est présenté en figure 2.

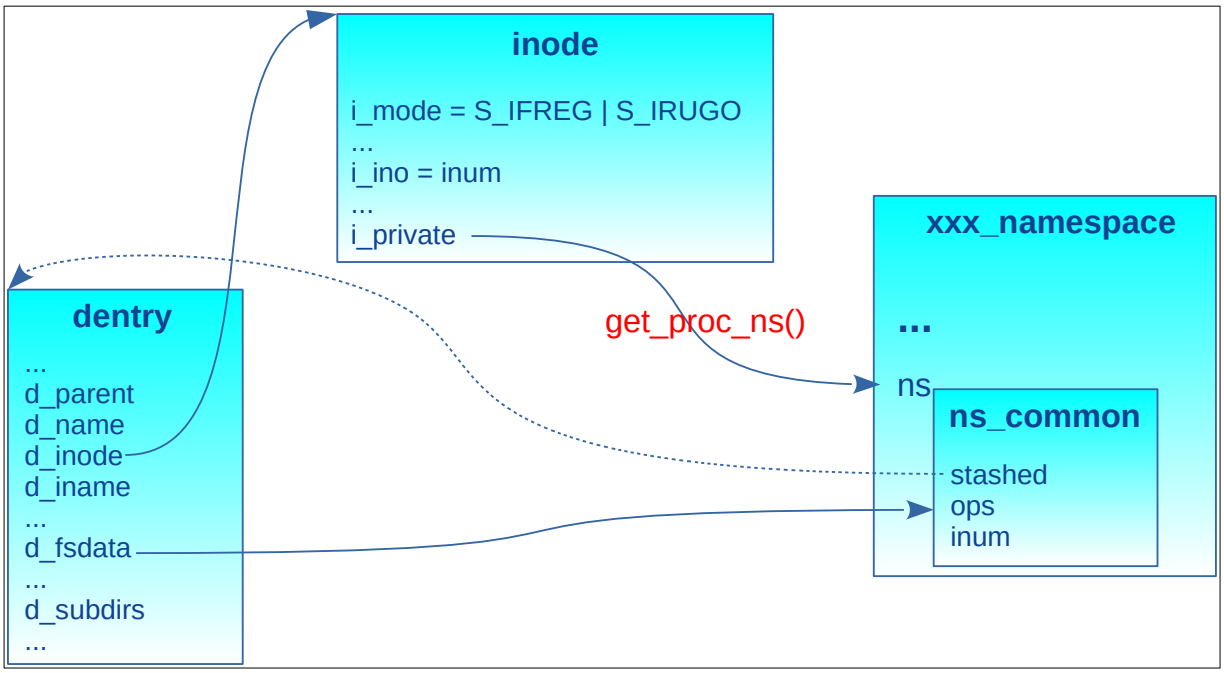


*Fig. 2: Inode dans **NSFS***

Le champ **ns** de type **ns_common** dans le descripteur de namespace est pointé par le champ **i_private** de la structure **inode**. Le numéro d'inode stocké dans le champ **inum** du namespace lors de son allocation, est assigné au champ **i_ino**. Le champ **stashed** de **ns_common** pointe sur la structure **dentry** liée à l'inode. Ce champ permet de savoir si l'inode est associé à un dentry ou pas car dans certaines situations où les ressources mémoire se raréfient, le **VFS** peut faire de « l'élagage » (c.-à-d. « prune » en anglais) dans le cache dentry de sorte à libérer des ressources. Dans ce cas, cela aboutit à l'appel de la fonction **ns_prune_dentry()** dans **fs/nsfs.c** qui marque le champ à 0 :

```
static void ns_prune_dentry(struct dentry *dentry)
{
        struct inode *inode = d_inode(dentry);
```

```
        if (inode) {
                struct ns_common *ns = inode->i_private;
                atomic_long_set(&ns->stashed, 0);
        }
}
```

## 1.3 Du descripteur de fichier au namespace

L'appel système **open()** sur un lien symbolique de namespace, aboutit via les fonctions précédentes, à l'allocation d'une structure **dentry**, de l'inode et la structure **file**. Un descripteur de fichier est retourné côté espace utilisateur.

Ensuite, avec les différentes fonctions de translation du **VFS**, le descripteur de fichier passé aux appels système tels que **ioctl()** ou **setns()**, permet de retrouver la structure **file** avec le service **proc_ns_fget()** définie dans **fs/nsfs.c** :

```
struct file *proc_ns_fget(int fd)
{
        struct file *file;

        file = fget(fd);
[...]
        if (file->f_op != &ns_file_operations)
                goto out_invalid;

        return file;
[...]
}
```

Le service **file_inode()** du fichier **include/linux/fs.h** retrouve l'inode à partir de la structure **file** :

```
static inline struct inode *file_inode(const struct file *f)
{
        return f->f_inode;
}
```

Le service **get_proc_ns()** du fichier **include/linux/proc_ns.h** retrouve un namespace à partir d'un pointeur sur une structure **inode** :

```
#define get_proc_ns(inode) ((struct ns_common *)(inode)->i_private)
```

Le champ **i_private** de l'inode pointe sur la structure du namespace et les opérations associées comme indiqué en figure 2.

## 1.4 Les montages

Un namespace est désalloué par le noyau à partir du moment où il n'est plus référencé (valeur 0 pour le compteur **kref** ou **count** vu dans l'article précédent). En général, il n'est plus référencé lorsqu'il n'y a plus aucune tâche associée. Cependant, comme on l'a vu avec la commande **ip** par exemple, il peut s'avérer nécessaire de garder un namespace actif même s'il n'a plus de tâche associée. Garder une tâche active liée au namespace est une solution mais ce n'est pas très économique en terme de ressources mémoire et CPU. L'astuce consiste à monter le fichier du namespace sur un autre fichier dans l'arborescence. Ainsi, tant qu'il est monté, le namespace reste référencé **[4]**. Reprenons l'exemple de création d'un net_ns avec la commande **ip** :

```
# ls -l /run/netns
ls: cannot access '/run/netns': No such file or directory
# ip netns add newnet
# ls -l /run/netns
total 0
-r--r--r-- 1 root root 0 avril  5 12:17 newnet
# cat /proc/$$/mountinfo
[...]
634 26 0:23 /netns /run/netns rw,nosuid,noexec,relatime shared:5 - tmpfs tmpfs rw,size=1635172k[...]
655 634 0:4 net:[4026532881] /run/netns/newnet rw shared:359 - nsfs nsfs rw
656 26 0:4 net:[4026532881] /run/netns/newnet rw shared:359 - nsfs nsfs rw
```

La commande a appelé **unshare(CLONE_NEWNET)** pour créer un nouveau net_ns. Une fois créé, la seule tâche associée au net_ns est la commande **ip** elle-même. Pour que ce net_ns ne disparaisse pas à la fin de la commande (c.-à-d. lorsqu'elle rend la main à l'opérateur), l'astuce consiste à monter **/proc/self/ns/net** (c.-à-d. la cible de ce lien symbolique !) sur **/run/netns/newnet** (c.-à-d. le nom passé en paramètre) afin de

laisser une référence sur le net_ns. Le fichier **mountinfo** montre le type de système de fichiers **NSFS** et le nom du fichier cible.

Ainsi, la commande **ip** pourra accéder de nouveau au net_ns pour y exécuter des commandes ou y migrer des interfaces réseau. Listons les net_ns créés :

```
# ip netns list
newnet
```

Listons les interfaces réseau dans le net_ns à l'aide de la commande **ip link list** associée à ce net_ns. On y trouve uniquement l'interface **loopback** car c'est la seule interface disponible dans tout nouveau net_ns. On remarquera qu'elle a l'indice 1 comme nous l'avions mentionné dans l'article précédent :

```
# ip netns exec newnet ip link list
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN mode DEFAULT group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
```

# 2 Les appels système

## 2.1 Les identifiants

Un certain nombre de fonctions et macros « helpers » ont été définies en interne :

- **pid_nr()** : Identifiant de processus (global) vu du pid_ns initial ;

- **pid_vnr()** : Identifiant de processus (virtuel) vu du pid_ns courant ;

- **pid_nr_ns()** : Identifiant de processus (virtuel) vu du pid_ns passé en paramètre ;

- **pid_<xid>_nr()** : Identifiant (global) vu du namespace initial ;

- **pid_<xid>_vnr()** : Identifiant (virtuel) vu du namespace courant ;

- **pid_<xid>_nr_ns()** : Identifiant (virtuel) vu du namespace passé en paramètre.

Ces dernières sont largement mises à contribution dans les appels système tels que **getpid()**, **gettid()**, **getgid()**, **getuid()** et autres afin de les **virtualiser (c.-à-d. leur faire retourner un un résultat correspondant aux namespaces auxquels le processus appelant est associé)**.

## 2.2 ioctl

Le point d'entrée **ioctl** du système de fichiers **NSFS** est la fonction **ns_ioctl()** dans le fichier **fs/nsfs.c**. Le début de la fonction retrouve l'inode à partir de la structure **file** passée en paramètre. Puis la structure **ns_common** du namespace cible est retrouvée (variable locale **ns**) depuis l'inode comme on l'a vu précédemment (cf. figure 2) :

```
static long ns_ioctl(struct file *filp, unsigned int ioctl,
                        unsigned long arg)
{
        struct user_namespace *user_ns;
        struct ns_common *ns = get_proc_ns(file_inode(filp));
        uid_t __user *argp;
        uid_t uid;
```

Dans l'article précédent, nous avons vu que le champ **ops** de la structure **ns_common** contient des informations et opérations communes à tous les namespaces. Elles vont être exploitées dans la suite.

Comme souvent avec les fonctions **ioctl()**, l'agorithme consiste en un « switch/case » pour chaque opération supportée :

```
        switch (ioctl) {
```

L'opération **NS_GET_USERNS** déclenche le point d'entrée **owner** afin de retourner le descripteur de fichier sur le user_ns propriétaire :

```
        case NS_GET_USERNS:
                return open_related_ns(ns, ns_get_owner);
```

L'opération **NS_GET_PARENT** déclenche le point d'entrée **get_parent** afin de retourner le descripteur de fichier sur le namespace parent. Le retour erreur **EINVAL** est pour les namespaces non hiérarchiques (c.-à-d. autres que user_ns et pid_ns). Leur champ **get_parent** est **NULL** :

```
	case NS_GET_PARENT:
		if (!ns->ops->get_parent)
			return -EINVAL;
		return open_related_ns(ns, ns->ops->get_parent);
```

L'opération **NS_GET_TYPE** retourne la valeur du champ **type** (c.-à-d. un drapeau **CLONE_NEWXXX**) :

```
	case NS_GET_NSTYPE:
		return ns->ops->type;
```

L'opération **NS_GET_OWNER_UID** ne s'applique qu'aux user_ns (sous peine de retour **EINVAL**) et retourne l'identifiant d'utilisateur d'un user_ns :

```
	case NS_GET_OWNER_UID:
		if (ns->ops->type != CLONE_NEWUSER)
			return -EINVAL;
		user_ns = container_of(ns, struct user_namespace, ns);
		argp = (uid_t __user *) arg;
		uid = from_kuid_munged(current_user_ns(), user_ns->owner);
		return put_user(uid, argp);
```

Un mauvais identifiant d'opération tombe sous le coup du cas « default » pour retourner **ENOTTY**. Ce code d'erreur peut prêter à confusion car dans le cas où les paramètres n'ont pas les valeurs attendues, on utilise généralement **EINVAL**. Mais c'est devenu une règle dans les ioctls des drivers et systèmes de fichiers afin de signifier que la commande passée n'est pas connue **[5]** :

```
	default:
		return -ENOTTY;
	}
}
```

On comprend maintenant la signification du nom du fichier d'entête **<linux/nsfs.h>** nécessaire à l'utilisation de **ioctl()** côté espace utilisateur :

```
[...]
/* Returns a file descriptor that refers to an owning user namespace */
#define NS_GET_USERNS		_IO(NSIO, 0x1)
/* Returns a file descriptor that refers to a parent namespace */
#define NS_GET_PARENT		_IO(NSIO, 0x2)
/* Returns the type of namespace (CLONE_NEW* value) referred to by
   file descriptor */
#define NS_GET_NSTYPE		_IO(NSIO, 0x3)
/* Get owner UID (in the caller's user namespace) for a user namespace */
#define NS_GET_OWNER_UID	_IO(NSIO, 0x4)
[...]
```

L'opération non documentée **SIOCGSKNS** **[6]** qui retourne un descripteur de fichier sur le net_ns auquel une socket est associée, est géré dans **net/socket.c** :

```
static long sock_ioctl(struct file *file, unsigned cmd, unsigned long arg)
{
	struct socket *sock;
	struct sock *sk;
	void __user *argp = (void __user *)arg;
	int pid, err;
	struct net *net;

	sock = file->private_data;
	sk = sock->sk;
	net = sock_net(sk);
[...]
		case SIOCGSKNS:
			err = -EPERM;
			if (!ns_capable(net->user_ns, CAP_NET_ADMIN))
				break;

			err = open_related_ns(&net->ns, get_net_ns);
			break;
[...]
```

De la structure **file** est déduit le descripteur de socket duquel on récupère une référence sur le net_ns associé. Ensuite la fonction vérifie que l'appelant a bien la capacité **CAP_NET_ADMIN** dans le user_ns propriétaire du net_ns avec le service interne **ns_capable()** **[7]**. Enfin, la fonction **open_related_ns()** de **fs/nsfs.c** est invoquée pour retourner un descripteur de fichier sur le net_ns.

## 2.3 clone

Le fichier **kernel/fork.c** contient l'implémentation de l'appel système **clone()**.

```
SYSCALL_DEFINE5(clone, unsigned long, clone_flags, unsigned long, newsp,
                int __user *, parent_tidptr,
                int __user *, child_tidptr,
                unsigned long, tls)
{
        struct kernel_clone_args args = {
                .flags          = (clone_flags & ~CSIGNAL),
                .pidfd          = parent_tidptr,
                .child_tid      = child_tidptr,
                .parent_tid     = parent_tidptr,
                .exit_signal    = (clone_flags & CSIGNAL),
                .stack          = newsp,
                .tls            = tls,
        };
[...]
        return _do_fork(&args);
}
```

La fonction **clone()** appelle **copy_process()** à partir de **_do_fork()** pour dupliquer la tâche appelante :

```
/*
 * This creates a new process as a copy of the old one,
 * but does not actually start it yet.
 *
 * It copies the registers, and all the appropriate
 * parts of the process environment (as per the clone
 * flags). The actual kick-off is left to the caller.
 */
static __latent_entropy struct task_struct *copy_process(
                                        struct pid *pid,
                                        int trace,
                                        int node,
                                        struct kernel_clone_args *args)
{
[...]
```

La fonction interdit la combinaison des drapeaux **CLONE_NEWNS** et **CLONE_FS** car deux processus évoluant dans des mount_ns différents ne peuvent pas partager le répertoire racine ou le répertoire courant. Elle interdit aussi la combinaison des drapeaux **CLONE_NEWUSER** et **CLONE_FS** car deux processus évoluant dans des user_ns différents ne peuvent pas partager les mêmes informations dans le système de fichier pour raisons de sécurité :

```
        if ((clone_flags & (CLONE_NEWNS|CLONE_FS)) == (CLONE_NEWNS|CLONE_FS))
                return ERR_PTR(-EINVAL);

        if ((clone_flags & (CLONE_NEWUSER|CLONE_FS)) == (CLONE_NEWUSER|CLONE_FS))
                return ERR_PTR(-EINVAL);
```

**A moins que le drapeau CLONE_NEWUSER ne soit passé**, la fonction ne peut être exécutée qu'à la condition où l'utilisateur a la capacité **CAP_SYS_ADMIN**. Nous verrons la raison de cette exception dans l'article dédié au user_ns :

```
        if ((clone_flags & CLONE_NEWUSER) && !unprivileged_userns_clone)
                if (!capable(CAP_SYS_ADMIN))
                        return ERR_PTR(-EPERM);
```

Par lesure de sécurité, tous les threads d'un processus doivent résider dans le même user_ns et pid_ns. Par conséquent, la fonction interdit la création d'un thread (drapeau **CLONE_THREAD** passé par **pthread_create()** de la librairie C) dans un nouveau user_ns ou pid_ns. Cela comprend aussi le cas où un processus fait un premier appel à **clone(CLONE_NEWPID)** pour créer un premier fils dans un nouveau pid_ns puis appelle **pthread_create()** pour créer un thread (sans ce contrôle, le nouveau thread s'exécuterait aussi dans le nouveau pid_ns bien que **CLONE_NEWPID** n'est pas passé).

```
        if (clone_flags & CLONE_THREAD) {
                if ((clone_flags & (CLONE_NEWUSER | CLONE_NEWPID)) ||
                    (task_active_pid_ns(current) !=
                                current->nsproxy->pid_ns_for_children))
                        return ERR_PTR(-EINVAL);
        }
```

Ensuite parmi les nombreuses autres actions, il y a la copie (héritage) des informations de sécurité (« credentials ») via **copy_creds()** et des namespaces du processus appelant via **copy_namespaces()**. La

copie des informations de sécurité **donne toutes les capacités au nouveau processus si le drapeau `CLONE_NEWUSER` est passé** (le nouvel user_ns associé est d'ailleurs créé à ce moment là). De plus, cela nous permet de souligner que **dès lors où le drapeau `CLONE_NEWUSER` est passé, il est traité en premier** ! Nous verrons l'importance de ces points lors dans l'article dédié aux user_ns :

```
[...]
        retval = copy_creds(p, clone_flags);
[...]
        retval = copy_namespaces(clone_flags, p);
```

Enfin, la fonction **`alloc_pid()`** est appelée pour allouer une structure **`pid`** afin de créer et mémoriser l'identifiant de la nouvelle tâche dans tous les pid_ns en remontant jusqu'à la racine. On notera que c'est ici que le pointeur **`pid_ns_for_children`** du **`nsproxy`** est utilisé pour référencer le pid_ns. Ce dernier pointe sur le pid_ns de la tâche appelante (le père) sauf si le drapeau **`CLONE_NEWPID`** a été passé auquel cas il pointe sur le pid_ns nouvellement alloué :

```
                pid = alloc_pid(p->nsproxy->pid_ns_for_children);
```

Détaillons la fonction **`copy_namespace()`** définie dans **`kernel/nsproxy.c`**. Si aucun des drapeaux **`CLONE_NEWXXX`** n'est passé, cela signifie que le processus créé hérite des namespaces de l'appelant (la variable locale **`old_ns`** pointe sur le **`nsproxy`** de la tâche appelante). **Dans ce cas, on n'alloue pas un nouveau `nsproxy` mais la nouvelle tâche va pointer sur le `nsproxy` de son père** en incrémentant son compteur de références (on a vu cela dans l'article précédent):

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

**Si au moins un des drapeaux `CLONE_NEWXXX` est passé, alors on va obligatoirement allouer une nouvelle structure `nsproxy`**. Les capacités du processus appelant sont de nouveau vérifiées (Rappel : si le drapeau **`CLONE_NEWUSER`** a été passé, toutes les capacités sont activées) :

```
        if (!ns_capable(user_ns, CAP_SYS_ADMIN))
                return -EPERM;
```

La combinaison des drapeaux **`CLONE_NEWIPC`** et **`CLONE_SYSVSEM`** n'est pas autorisée car dans un nouvel ipc_ns, les sémaphores de l'appelant ne sont plus accessible et par conséquent il n'est pas possible de partager la liste des **`semadj`** (mécanisme qui sera détaillé dans l'article dédié aux ipc_ns) :

```
        if ((flags & (CLONE_NEWIPC | CLONE_SYSVSEM)) ==
                (CLONE_NEWIPC | CLONE_SYSVSEM))
                return -EINVAL;
```

Ensuite est appelée la fonction **`create_new_namespaces()`** située dans le même fichier. Cette fonction alloue une nouvelle structure **`nsproxy`**. Pour chaque drapeau **`CLONE_NEWXXX`** passé en argument à **`clone()`**, le namespace associé est allouée et le pointeur associé dans le **`nsproxy`** pointe dessus. Pour les autres namespaces dont les drapeaux ne sont pas spécifiés, il n'y a pas d'allocation de nouveaux namespaces mais seulement une référence au namespace de l'appelant (avec incrémentation du compteur de références) et le pointeur correspondant du **`nsproxy`** pointe dessus :

```
        new_ns = create_new_namespaces(flags, tsk, user_ns, tsk->fs);
```

Enfin, le **`nsproxy`** ainsi alloué est assigné au champ **`nsproxy`** de la nouvelle tâche :

```
        tsk->nsproxy = new_ns;
        return 0;
}
```

La figure 3 schématise l'opération **`clone(CLONE_NEWUTS)`**. Les actions et structures mises en oeuvre pour la nouvelle tâche sont en orange. Les pointeur du **`nsproxy`** nouvellement alloué sont identiques à ceux du processus père sauf pour l'uts_ns qui pointe sur le descripteur d'uts_ns nouvellement alloué. Pour des raisons de lisibilité, nous n'avons pas représenté la structure **`pid`**.

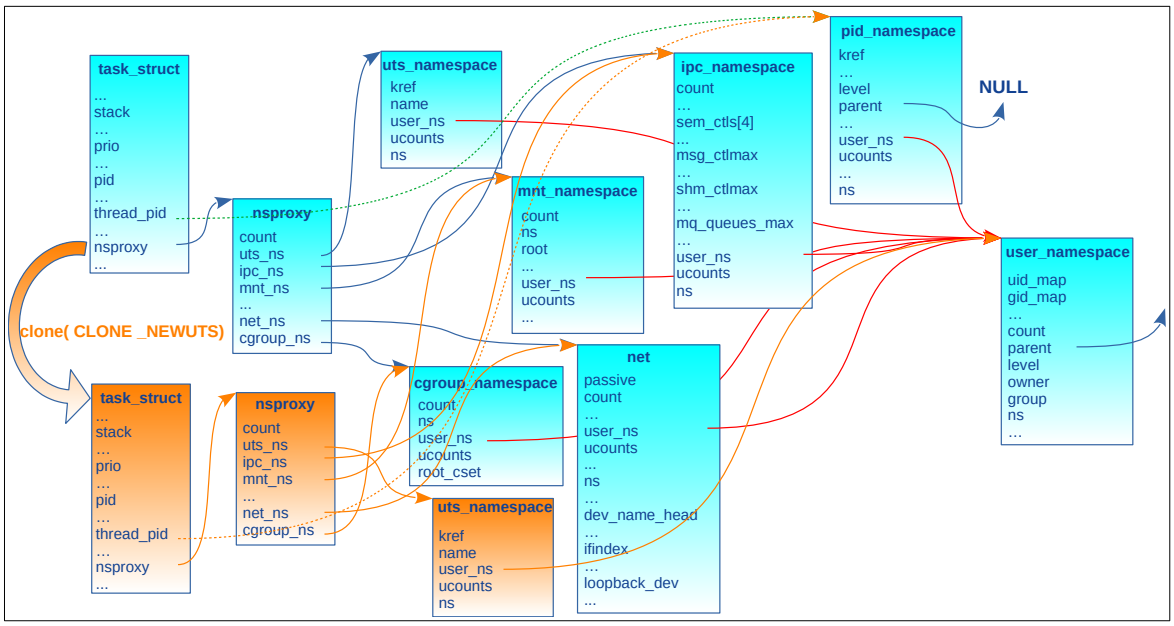


*Fig. 3: Appel système **clone()** avec **CLONE_NEWUTS***

## 2.4 unshare

Le fichier **kernel/fork.c** contient l'implémentation de l'appel système **unshare()** :

```
SYSCALL_DEFINE1(unshare, unsigned long, unshare_flags)
{
	return ksys_unshare(unshare_flags);
}
```

Tout le travail est fait dans la fonction **ksys_unshare()** située dans le même fichier. **unshare()** accepte bon nombre de drapeaux en paramètres. Nous nous concentrerons uniquement sur ceux concernant les namespaces. La fonction commence par positionner implicitement les drapeaux **CLONE_THREAD** et **CLONE_FS** si **CLONE_NEWUSER** est passé :

```
int ksys_unshare(unsigned long unshare_flags)
{
	struct fs_struct *fs, *new_fs = NULL;
	struct files_struct *fd, *new_fd = NULL;
	struct cred *new_cred = NULL;
	struct nsproxy *new_nsproxy = NULL;
	int do_sysvsem = 0;
	int err;

	/*
	 * If unsharing a user namespace must also unshare the thread group
	 * and unshare the filesystem root and working directories.
	 */
	if (unshare_flags & CLONE_NEWUSER)
		unshare_flags |= CLONE_THREAD | CLONE_FS;
```

Le drapeau **CLONE_FS** est aussi implicitement positionné s'il y a création d'un nouveau mount_ns :

```
	/*
	 * If unsharing namespace, must also unshare filesystem information.
	 */
	if (unshare_flags & CLONE_NEWNS)
		unshare_flags |= CLONE_FS;
```

Comme pour **clone()** vu précédemment, si **CLONE_NEWUSER** n'est pas passé alors la fonction retourne en erreur (**EPERM**) si l'appelant n'a pas la capacité **CAP_SYS_ADMIN**. Lors de l'étude détaillée des user_ns, nous verrons pourquoi il est important d'autoriser la fonction lorsque que la création d'un user_ns est demandé :

```
	if ((unshare_flags & CLONE_NEWUSER) && !unprivileged_userns_clone) {
		err = -EPERM;
		if (!capable(CAP_SYS_ADMIN))
			goto bad_unshare_out;
	}
```

Un certains nombre de contrôles de cohérence et de validité sur les drapeaux est effectué :

```
	err = check_unshare_flags(unshare_flags);
```

Si le drapeau **CLONE_NEWUSER** est passé, comme précédemment souligné pour **clone()**, **il est traité en premier** en allouant un nouveau user_ns (appel à **unshare_userns()**) puis les autres namespaces sont créés conformément aux autres drapeaux passés en paramètres (appel à **unshare_nsproxy_namespaces()** qui appelle **create_new_namespaces()** vue avec **clone()**). On notera que si **CLONE_NEWIPC** est passé, **un nouveau système de fichiers mqueue** (généralement monté sur **/dev/mqueue** côté espace utilisateur) **est monté en interne pour le nouvel ipc_ns**. L'ancien est « oublié ». En d'autres termes, les noms des queues de message POSIX créées par la tâche courante ne seront plus accessible pour une destruction dans le système de fichiers car la tâche ne les verra plus à partir de son nouvel ipc_ns. Par contre, les descripteurs de fichiers actuellement ouverts sur les queues de message permettent toujours d'y accéder pour émettre et recevoir des messages ! On reverra cela dans un exemple lors de l'étude détaillée des ipc_ns.

```
	err = unshare_userns(unshare_flags, &new_cred);
[...]
	err = unshare_nsproxy_namespaces(unshare_flags, &new_nsproxy, new_cred, new_fs);
```

Si le drapeau **CLONE_NEWIPC** a été passé (la variable locale **do_sysvsem** vaut 1), la fonction **exit_sem()** est appelée pour mettre en œuvre le mécanisme **semadj** afin d'éviter les inter-blocages en annulant les opérations en cours sur les sémaphores. Nous reviendrons sur ce sujet dans le paragraphe dédié à l'ipc_ns.

```
	if (new_fs || new_fd || do_sysvsem || new_cred || new_nsproxy) {
		if (do_sysvsem) {
			/*
			 * CLONE_SYSVSEM is equivalent to sys_exit().
			 */
			exit_sem(current);
		}
```

Toujours pour le drapeau **CLONE_NEWIPC**, **exit_shm()** est appelée pour les identifiants de segments de mémoire partagée. Les segments créés par la tâche courante sont marqués orphelins (c.-à-d. l'identifiant de tâche créatrice est effacé afin de pouvoir les détruire plus tard dans l'ipc_ns que l'on quitte). Puis la liste des segments créés est mise à 0 pour la tâche courante. De plus, si le fichier **/proc/sys/kernel/shm_rmid_forced** est différent de 0, alors les segments créés par la tâche et non encore attachés (c.-à-d. non mappés par au moins une tâche) sont détruits.

```
		if (unshare_flags & CLONE_NEWIPC) {
			/* Orphan segments in old ns (see sem above). */
			exit_shm(current);
			shm_init_task(current);
		}
```

Ensuite on met à jour le champ **nsproxy** de la tâche appelante avec le nouveau **nsproxy** alloué si au moins un des drapeaux **CLONE_NEWXXX** autre que **CLONE_NEWUSER** est passé en paramètre à l'aide de **switch_task_namespaces()** . Cette dernière décrémente le compteur de référence sur le **nsproxy** courant et sa désallocation si le compteur atteint 0 :

```
		if (new_nsproxy)
			switch_task_namespaces(current, new_nsproxy);
```

Enfin, si **CLONE_NEWUSER** a été passé, le nouveau user_ns est « attaché » à la tâche courante :

```
		if (new_cred) {
			/* Install the new user namespace */
			commit_creds(new_cred);
			new_cred = NULL;
		}
	}
```

La figure 4 schématise l'opération **unshare(CLONE_NEWUTS)**. Les actions et structures mises en oeuvre sont en orange.

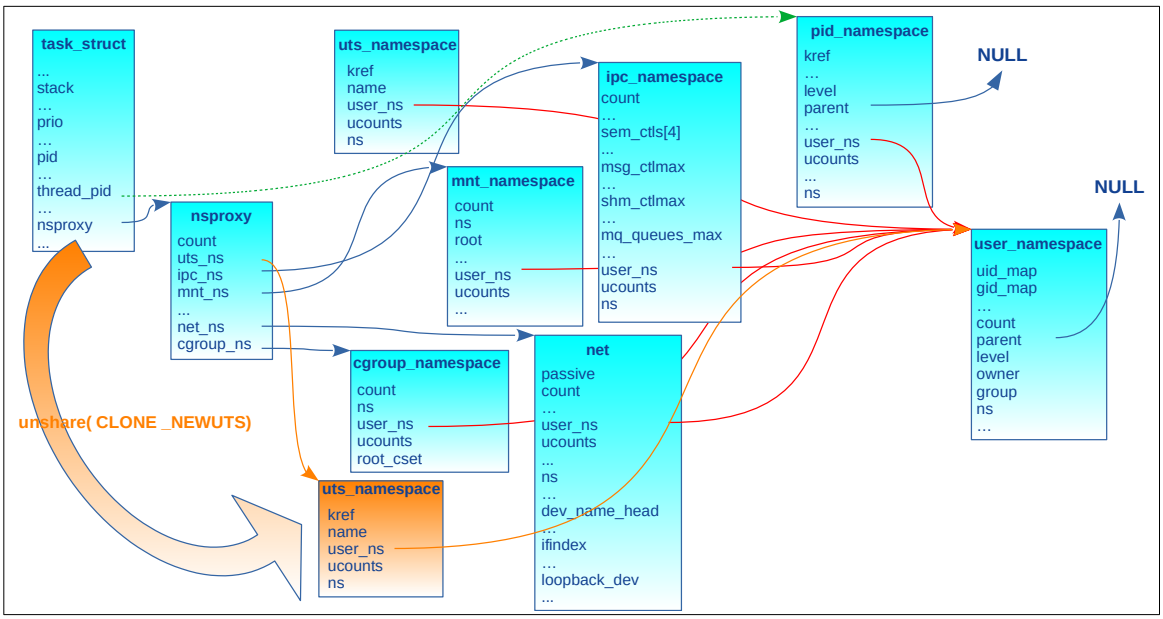


*Fig. 4: Appel système unshare()*

On notera pour conclure que dans le cas de la création d'un nouveau pid_ns (drapeau **CLONE_NEWPID**), l'appelant n'est pas associé au nouveau pid_ns. Le champ **pid_ns_for_children** de son **nsproxy** est modifié pour le référencer afin qu'un prochain appel à **clone()** (sans le drapeau **CLONE_NEWPID**) s'appuie dessus pour y associer la nouvelle tâche. Dans un prochain article, nous expliquerons la raison de ce fonctionnement particulier pour le drapeau **CLONE_NEWPID**.

## 2.5 setns

L'appel système **setns()** est défini dans le fichier **kernel/nsproxy.c**. Le descripteur de fichier **fd** passé en paramètre, permet de retrouver la référence sur le descripteur du namespace cible comme on l'a déjà vu (variable locale **ns**). Si le paramètre **nstype** est différent de 0, on vérifie qu'il correspond bien au type du namespace lié au descripteur de fichier (sinon l'erreur **EINVAL** est retournée) :

```
SYSCALL_DEFINE2(setns, int, fd, int, nstype)
{
        struct task_struct *tsk = current;
        struct nsproxy *new_nsproxy;
        struct file *file;
        struct ns_common *ns;
        int err;

        file = proc_ns_fget(fd);
...]
        err = -EINVAL;
        ns = get_proc_ns(file_inode(file));
        if (nstype && (ns->ops->type != nstype))
                goto out;
```

La fonction **create_new_namespaces()** déjà vue lors de l'étude de **clone()** est alors appelée pour allouer une nouvelle structure **nsproxy**. Le premier paramètre « flags » étant à 0, ses pointeurs vont référencer les mêmes structures que le **nsproxy** de la tâche appelante mais leur compteur de références est incrémenté.

```
        new_nsproxy = create_new_namespaces(0, tsk, current_user_ns(), tsk→fs);
```

Puis la fonction **install()** du namespace cible (c.-à-d. correspondant au descripteur de fichier passé en paramètre) est appelé. Cette fonction vérifie la présence de la capacité **CAP_SYS_ADMIN** sous peine de retourner l'erreur **EPERM**, décrémente le compteur de références sur le namespace source (celui dans lequel l'appelant se trouve), effectue quelques actions d'intégrité dans le namespace source, incrémente le compteur de références sur le namespace cible et fait pointer le nouveau **nsproxy** dessus :

```
        err = ns->ops->install(new_nsproxy, ns);
```

Parmi les opérations d'intégrité, on peut citer l'exemple où le namespace cible serait un ipc_ns : le mécanisme **semadj** est déclenché pour l'ipc_ns source avec un appel à **exit_sem()**. Dans le cas où le namespace cible est un user_ns, on retourne en erreur (**EINVAL**) si l'appelant est une tâche d'un processsus

multithreadé (car tous les threads d'un processus doivent être dans le même user_ns).

Enfin le champ **nsproxy** de la tâche courante est remplacé pour laisser place à celui que l'on vient de créer avec l'un de ses pointeurs référençant le namespace cible. L'ancien **nsproxy** est libéré si la décrémentation de son compteur de références aboutit à la valeur 0 :

```
        switch_task_namespaces(tsk, new_nsproxy);
```

En conclusion, la tâche appelante se retrouve avec un nouveau **nsproxy** identique au précédent sauf pour le pointeur associé au namespace cible. Cette opération est identique à l'opération **unshare()** car elle alloue un nouveau **nsproxy**. Mais **unshare()** crée un nouveau namespace alors que **setns()** fait pointer son **nsproxy** sur un namespace existant.

# Conclusion

Ainsi s'achève notre plongée dans le noyau pour observer de près l'implémentation des namespaces. Le but n'était pas de comprendre tous les détails mais plutôt de donner des clés à qui voudrait approfondir le sujet ou comprendre certaines limitations que nous évoquerons par la suite.

A partir du prochain article, nous repasserons en espace utilisateur pour commencer notre passage en revue de chaque namespace de manière pratique à l'aide de programmes d'exemples.

# Références


**[1]** Overview of the Linux Virtual File System : https://www.kernel.org/doc/Documentation/filesystems/vfs.txt

**[2]** Anatomy of the Linux virtual file system switch : https://developer.ibm.com/technologies/linux/tutorials/l-virtual-filesystem-switch/

**[3]** The namespace file system (NSFS) : https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=e149ed2b805fefdccf7ccdfc19eca22fdd4514ac

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

**[5]** J. Corbet, A. Rubini, G. Kroah-Hartman, « Linux Device Drivers » (3rd Edition), O'Reilly, February 2005

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

**[7]** Linux capabilities support for user namespaces : https://lwn.net/Articles/420624/