# NMV - 5I453

# TP 08 – Appels systèmes

**Maxime Lorrillere et Julien Sopena**

**Octobre 2015**

**Environnement :** ce TP reprend l'environnement de programmation mis en place lors du deuxième TP. Il suppose entre autre l'utilisation dans qemu de l'image *nmv-tp.img*, ainsi que l'existence d'un fichier personnel *myHome.img* correspondant au `/root`. Il suppose aussi que vous ayez un fichier de configuration pour le noyau linux 4.2.3 et que vous ayez dans `/tmp` un exemplaire des sources.

## Exercice 1 : Mon premier appel système

Les appels systèmes sont des interfaces de communication fournies par le noyau pour permettre aux applications d'accéder à certaines de ses ressources. Un appel système est défini par son *numéro* et par ses paramètres. Le noyau possède une *table des appels systèmes* contenant les fonctions à exécuter : le numéro d'un appel système correspondant à un indice de cette table, et donc à sa fonction.

La plupart du temps, les appels systèmes sont exécutés par les applications par l'intermédiaire d'APIs, telles que POSIX ou la bibliothèque standard C. Lorsqu'aucune API ne fournit de fonction pour accéder à un appel système, la fonction `syscall` peut être utilisée pour accéder à l'appel système correspondant. Cette fonction exécute l'appel système, et s'il échoue (valeur de retour inférieure à 0), place le code d'erreur correspondant dans la variable `errno` et renvoie `-1`. Sinon elle renvoie la valeur retournée par l'appel système.

Les appels systèmes sont très dépendants de l'architecture utilisée. Dans l'architecture x86 64 bits (que vous utilisez en TP), la table des appels système est définie dans le fichier `arch/x86/entry/syscalls/syscall_64.tbl`. Cette table définit pour chaque appel système son numéro, son ABI (`common` pour nous), son nom et le nom de la fonction correspondante.

L'implémentation d'un appel système passe par l'utilisation de macros fournie par le noyau de la forme suivante :

```
#define SYSCALL_DEFINEx(nom, type1, nom1, type2, nom2, ...)
```

Ces macros définissent une fonction, dont le nom est `sys_nom` et prenant $x$ paramètres de types donnés. Le type de retour de la fonction ainsi définie sera `long`. Par convention, une valeur inférieure à 0 est un code d'erreur négatif.

Par exemple, un appel système ne prenant pas de paramètre, tel que l'appel système `sync`, peut être sera défini de la façon suivante :

```
SYSCALL_DEFINE0(sync)
{
    /* ... */
    return 0;
}
```

Un appel système prenant 2 paramètres, tel que l'appel système `chmod`, peut être défini de la façon suivante :

```
SYSCALL_DEFINE2(chmod, char __user *, filename, umode_t, mode)
{
    /* ... */
    return 0;
}
```

Dans ce cas, les paramètres de type pointeur doivent être manipulés avec précautions, en utilisant par exemple les fonctions `copy_to_user` et `copy_from_user`.

### Question 1

Ouvrez la page de manuel de la fonction `syscall`. Quel est le rôle de cette fonction ? Est-ce un appel système ? Quels sont ses paramètres et sa valeur de retour ?

### Question 2

Réalisez un petit programme C qui exécute l'appel système `kill`, dont le numéro est `__NR_kill`, en utilisant la fonction `syscall`.

### Question 3

Ajoutez un nouvel appel système à votre noyau, que vous appellerez `helloworld`. Lorsqu'il est exécuté, le noyau affichera le message *Hello world !"* dans le syslog.

### Question 4

On souhaite maintenant modifier dynamiquement l'affichage généré par l'appel système en y ajoutant un paramètre `what`. Par exemple, passer la valeur *"beer"* comme paramètre générera l'affichage *Hello beer !"* dans le syslog.

### Question 5

Plutôt que d'afficher le résultat dans le syslog, on souhaite renvoyer la chaîne générée dans un tampon fourni par l'utilisateur. Modifiez votre appel système en conséquence, celui-ci devra retourner la taille de la chaîne générée, ou un code d'erreur négatif si l'exécution de l'appel système échoue.

### Question 6

Lors de l'exécution de votre appel système, affichez le PID du processus courant dans le syslog. Que constatez vous ?

## Exercice 2 : Détourner un signal

Dans cet exercice, on cherchera dérouter les signaux adressés au processus précédemment dissimulé. Les techniques employées permettront d'étudier : le fonctionnement d'un appel système sous Linux, le rôle de la table des appels système et la table des symboles du noyau.

### Question 1

Après avoir lancer une commande `sleep 9000 &`, envoyez lui un signal `SIGCONT` puis renouvelez l'opération après l'avoir masqué.

En comparant vos observations au résultat de l'envoi du même signal à un processus inexistant, proposez une méthode permettant de lister les processus cachés.

### Question 2

Pour contrer cette détection nous allons dérouter le signal au niveau des appels système. Lancez la commande `strace` sur un `kill`. Quels appels système doit-on dérouter ?

### Question 3

L'ensemble des appels système passe par l'interruption `int 0x80`. Pour les différencier, on leur associe un numéro (*system call number*). Une fois basculé en mode noyau, ce numéro est utilisé pour chercher dans la **table des appels système** l'adresse de l'appel système.

Intercepter un appel système correspond donc à remplacer, dans la table des appels système, l'adresse originale par celle de votre fonction. Cette méthode nécessite de connaitre le début de la table noté par le symbole `sys_call_table`. Malheureusement, depuis la version 2.6, ce symbole n'est plus exporté.

Implémentez une fonction `find_sys_call_table` qui retourne l'adresse de la table, en sachant qu'elle se trouve entre l'adresse `&unlock_kernel` et l'adresse `&loops_per_jiffy`. Vous pouvez utiliser le fait que l'adresse du `sys_close` se trouve dans la case `__NR_close` (`<linux/unistd.h>`).

### Question 4

Comparer le résultat de votre fonction `find_sys_call_table` avec la valeur exportée dans le `/boot/System.map26`. Pourquoi ne peut-on pas directement utiliser cette valeur ?

### Question 5

Pourquoi linux exporte l'adresse des appels système (`<linux/syscalls.h>`), alors qu'on utilise la table des appels système pour les trouver ?

### Question 6

Maintenant que vous avez l'adresse de la table, intercepter le `sys_kill` avec une fonction qui retourne `-ESRCH` si le *pid* correspond au processus à cacher.

**Attention :** votre module doit conserver le fonctionnement normal des signaux pour les autres processus. D'autre part vous devez rétablir le comportement original lors du déchargement du module.