Systèmes et réseaux

Chapitre 4

Programmation noyau

Pablo Rauzy <pr@up8.edu>
pablo.rauzy.name/teaching/sr

Programmation noyau

- On parle de *programmation noyau* quand il s'agit de programmer du code qui va s'exécuter dans l'espace noyau.
  - Cela peut être le code du noyau lui-même bien sûr,
  - mais aussi le code de modules ou de pilotes par exemple.
- Programmer un module est une bonne façon d'essayer la programmation noyau.
- Documentation :
  - chercher sur le web...
  - dans la doc : `https://www.kernel.org/doc/`
  - dans les sources : `https://elixir.bootlin.com/`

Espace noyau

- Le code qui s'exécute dans l'espace noyau n'a pas accès aux bibliothèques existantes dans l'espace utilisateur. Y compris la libc.
  - Il faudra programmer en se passant de "`printf`", "`malloc`", etc.
- Il en va de même pour l'espace mémoire.
  - Il faudra passer par des fonctions ou interfaces spécifiques pour échanger des données vers et depuis l'espace utilisateur.

Module noyau

- Un *module noyau chargeable* (ou LKM, *loadable kernel module*) permet d'ajouter du code au noyau en cours d'exécution.
  - Cela permet aussi de compartimenter le code du noyau en projets séparés.
- `sudo apt install kmod`
  - `lsmod` liste les modules noyau chargés,
  - `insmod` charge un module dans le noyau,
  - `rmmod` décharge un module du noyau ;
  - voir aussi `modprobe`.
- On peut aussi explorer `/sys` et `/proc` :
  - `/proc/modules`
  - `/sys/module/*`

Préparation de l'environnement de développement

- Il vous est fortement conseillé de travailler dans une machine virtuelle !
- Installation des entêtes de développement noyau :
  - `sudo apt install linux-headers-amd64`
  - `sudo apt install linux-headers-$(uname -r)`

Un premier module

- Voyons maintenant concrètement comment écrire, compiler, et utiliser un premier module.

Exemple complet

coucou.c :

```c
#include <linux/module.h>
#include <linux/init.h>
#include <linux/printk.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Pablo");
MODULE_DESCRIPTION("A first module.");
MODULE_VERSION("0.1");

static int __init coucou_init (void)
{
  printk(KERN_INFO "coucou: module initializes\n");
  return 0;
}

static void __exit coucou_exit (void)
{
  printk(KERN_INFO "coucou: module exits\n");
}

module_init(coucou_init);
module_exit(coucou_exit);
```

Compilation du module

- La compilation doit se faire avec un Makefile :

```
1 #obj-m += coucou.o
2
3 all:
4         make -C /lib/modules/$(shell uname -r)/build/ M=$(PWD) modules
5 clean:
6         make -C /lib/modules/$(shell uname -r)/build/ M=$(PWD) clean
```

- Le résultat de la compilation sera un fichier **`coucou.ko`** (*kernel object*).

Utilisation du module

- `modinfo coucou.ko # affiche les infos sur le module`
- `dmesg -W # dans un autre terminal`
- `insmod coucou.ko # charge le module`
- `lsmod | grep coucou`
- `cat /sys/module/coucou/version`
- `rmmod coucou # décharge le module`

Remarques

- Quelques différences importantes avec les programmes s'exécutant en espace utilisateur :
  - Privilèges élevés !
  - Pas de nettoyage automatique !
  - Pas de descripteur de fichier ouvert par défaut !
  - Pas de `main` !

Privilèges élevés

- Votre code s'exécute avec les privilèges du noyau.
- "With great power comes great responsibility."
- Attention !

Pas de nettoyage automatique

- Lors de l'exécution d'un programme en espace utilisateur, une fois que le programme à terminé, le système récupère l'ensemble des ressources, notamment la mémoire.
- Ce n'est pas le cas avec un module noyau : votre fonction de sortie doit *impérativement* nettoyer le système de ce qui y a été ajouté par le module.

Pas de descripteur de fichier ouvert par défaut

- Contrairement à un programme normal :
  - Pas d'entrée standard.
  - Pas de sortie standard.
  - Pas de sortie d'erreur standard.
- `printk` et les fonction `pr_*` écrive dans les logs du noyau.
  - `/var/log/kern.log`
  - `sudo dmesg`

Pas de `main`

- Il n'y a pas de point d'entrée du programme.
- La fonction d'initialisation enregistre simplement auprès du noyau des gestionnaires d'évènements.
  - On peut comparer cette façon de fonctionner à la *programmation événementielle*.

Passage de paramètre

- Voyons le code d'un deuxième module, qui prend un paramètre.

Code du deuxième module

`salut.c` :

```c
#include <linux/module.h>
#include <linux/init.h>
#include <linux/printk.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Pablo");
MODULE_DESCRIPTION("A second module, with a parameter.");
MODULE_VERSION("0.1");

static char *who = "tout le monde";
module_param(who, charp, S_IRUGO | S_IWUSR);
MODULE_PARM_DESC(who, "Who to salute?");

static int __init salut_init (void)
{
  printk(KERN_INFO "salut: Salut %s !\n", who);
  return 0;
}

static void __exit salut_exit (void)
{
  printk(KERN_INFO "salut: Bye %s !\n", who);
}

module_init(salut_init);
module_exit(salut_exit);
```

Utilisation du module et de son paramètre

- `modinfo salut.ko`
- `insmod salut.ko who=Pablo`
- `ls /sys/module/salut/parameters`
- `echo -n "quelqu'un d'autre" > /sys/module/salut/parameters/who`
- `rmmod salut`

Voyons ensemble un module plus complexe

- Pour l'instant nos modules ne font rien d'intéressant pendant leur "vie".
- Écrivons ensemble un module **`accum`** qui :
  - crée un fichier virtuel **`/proc/accum`** qui accumule les nombres qu'on y écrit ;
  - crée un paramètre sysfs **`multiplier`** et s'en sert lors de l'ajout.