NMV - NI436

TP 03 – Mes premiers modules

Maxime Lorrillere et Julien Sopena

Octobre 2015

Mise en garde : au cours de ce TP nous allons implémenter et charger des modules. Ces modules ont un accès à la totalité de la mémoire du noyau et peuvent donc corrompre son fonctionnement normal. Les conséquences peuvent être irrémédiables sur le système et sur vos données (corruption du système de fichier). C'est pourquoi il est recommandé de travailler sur un système test indépendant de votre machine de travail.

Recherche dans les sources : Pour l'ensemble de ce TP, vous pouvez utiliser comme moteur de recherche les sites lxr.free-electrons.com et lxr.missinglinkelectronics.com qui indexent les sources du noyau. Tous deux reposent sur le projet libre lxr, un utilitaire général d'indexation de code source et de références croisées qui permet la navigation au sein du code source en utilisant une technologie web.

Exercice 1 : Chargement dynamique de bibliothèques

Si les modules sont un moyen simple de développer, tester et distribuer de nouvelles fonctionnalités du noyau Linux, ils sont aussi très utiles pour les applications utilisateurs. Dans cet exercice nous allons étudier les différentes fonctionnalités offertes par les bibliothèque dynamiques qui sont à la base de la programmation des modules.

Question 1

Pour commencer récupérez depuis `/usr/data/sopena/nmv/TP-03` les fichiers : `cron_func.c`, `func.h`, et `nothing.c` ainsi que le `Makefile` associé et exécutez le programme `./cron_func` après l'avoir compilez. Ouvrez ensuite les sources pour compredre son fonctionnement.

Question 2

Dans un premier temps, on veut pouvoir modifier le comportement du programme sans le recompiler. L'idée est d'embarquer l'implémentation de `func()` dans une librairie dynamique `libfunc.so`.

Modifiez en conséquence votre `Makefile` et vérifiez que l'on puisse réimplémenter la nothing.c sans avoir à recompiler le programme `./cron_func`.

Question 3

Si l'utilisation massive de bibliothèques dynamiques permet d'économiser de la mémoire, elle peut poser des problèmes de sécurité. Il est en effet possible de rediriger un executable vers une version modifiée d'une bibliothèque.

En vous basant sur le squelette `my_read`, modifiez le comportement de `cron_func` pour que la saisie d'un 'r' execute le code d'une insertion (action du caractère 'i'). Vous utiliserez la variable `LD_PRELOAD` :

```
LD_PRELOAD=./libread.so ./cron_func
```

Question 4

On veut maintenant pouvoir modifier le comportement d'un programme en cours d'exécution, sans avoir à le redémarrer. A cette effet, l'appel système `dlopen` permet de recharger un symbole depuis une bibliothèque passée en paramètre.

Après avoir implémenté une nouvelle version de la fonction `func()` (respectant le prototype défini dans `func.h`). Modifier le `cron_func` pour qu'un `'i'` charge votre fonction et qu'un `'r'` restore l'implémentation originale.

Exercice 2 : Mes premiers modules

Question 1

Copiez depuis `/usr/data/sopena/nmv/TP-03` les sources du module *HelloWorld* et compiler les. Parmi les fichiers générés quel est celui/ceux qui seront chargés dans le noyau ?

Question 2

A l'aide de la commande **$(info xxx)** essayez de comprendre comment sont exécutées les lignes du *Makefile*.

Question 3

Chargez puis déchargez le module *HelloWorld*, puis listez les endroits où l'on peut lire le message **"Hello world**

Question 4

À partir du module *HelloWorld*, écrivez un module *HelloWorldParam* qui utilisera deux paramètres "whom" (une chaîne de caractères) et "howmany" (un entier) pour afficher :

```
insmod helloWorldParam.ko whom=julien howmany=3
   (0) Hello, julien
   (1) Hello, julien
   (2) Hello, julien
rmmod helloWorldParam
   Goodbye, julien
```

Question 5

Tester l'affichage des informations de module obtenu grâce à la commande `modinfo`. Vous modifierez si besoin votre code pour avoir une description de chaque paramètre.

Question 6

En modifiant si besoin le module *HelloWorldParam*, utilisez l'interface `/sys` pour changer (à posteriori) l'affichage de son déchargement.

Exercice 3 : Modification d'une variable du noyau à l'aide d'un module

Question 1

La variable globale **init_uts_ns** est utilisée dans le noyau pour stocker les informations retournées par l'appel système **uname**. Dans les sources du noyau, trouvez où est initialisée cette structure et analysez les champs qu'elle contient. Pour cela, vous pouvez être amenés à utiliser les sites d'indexation du noyau conseillé dans le préambule de ce sujet.

Est-ce que cette variable peut être accédée par tous les modules ?

Question 2

Créez un module dont le rôle sera de modifier le nom du noyau lors de son chargement.

Lors de son déchargement, il est impératif que le module restaurer le nom d'origine. Pourquoi ?

Exercice 4 : Cacher un module

Dans cet exercice on cherchera à dissimuler la présence d'un module dans la mémoire du noyau. Les techniques employées permettront d'étudier : la structuration des objets du noyau, leur référencement, mais aussi le fonctionnement du système de fichier **sysfs**.

Question 1

A l'aide de la commande **lsmod** vérifiez que le module *helloWord* est bien chargé dans le noyau.

Question 2

Sachant qu'un **lsmod** parcours la liste des modules, implémentez un module *hideModule* qui reste invisible à cette commande.

Question 3

S'il reste caché pour la commande **lsmod**, votre module laisse encore des traces dans le système de fichier **sysfs**. En effet, au chargement d'un module l'ensemble de ses paramètres, ainsi qu'un certain nombre d'informations, sont enregistrés dans des fichiers du répertoire */sys/modules/xxx*.

Commencez par vérifier que votre module est bien visible.

Question 4

Le **sysfs** est un système fichier spécial semblable à */proc*. Il a été introduit pour le compléter (voir le remplacer). Mais contrairement au */proc*, il présente une vue hiérarchique des informations : à chaque niveau correspond un objet : *substytem*, *kset*, *kobject*, attributs (voir figure **??**).

Ces objets ne correspondent pas à des données stockées sur disque. Ils ne sont pas non plus complètement virtualisés, i.e., recalculés à chaque lecture. En fait, l'ensemble des données du *sysfs* est stocké dans la mémoire du noyau. Effacer une donnée du *sysfs* revient donc à la dé-référencer puis à libérer l'espace mémoire correspondant.

Ajoutez une fonction qui efface les traces du module dans le *sysfs*, sachant que les références du **kobject** correspondant à un module, ainsi que celles de ses **attributs**, sont enregistrées dans la structure *module*.

Exercice 5 : Les limites des modules

Les modules permettent d'étendre le fonctionnalités d'un noyau sans même avoir besoin de redémarrer la machine. Cependant, ils ne peuvent pas tout faire, et il est parfois nécessaire de modifier directement le code du noyau et de le recompiler pour implémenter la fonctionnalité désirée. Dans cet exercice, les modules que nous vous demandons de créer **ne peuvent fonctionner sans modifications du noyau**.

Question 1

On souhaite réaliser un module qui affiche des informations sur les *superblocks* chargés en mémoire par le noyau, comme par exemple l'UUID du superblock et son type de système de fichiers. Pour cela, il est nécessaire de parcourir la liste des `struct super_block`. Pour faciliter cette opération, le noyau propose la fonction `iterate_supers`, qui effectue ce parcours à votre place et exécute la fonction passée en paramètre pour chaque superblock.

À l'aide de cette fonction, créez le module `show_sb` qui parcours la liste des superblocks et produit un affichage similaire à celui-ci :

```
uuid=00000000-0000-0000-0000-000000000000 type=rootfs
uuid=00000000-0000-0000-0000-000000000000 type=bdev
...
uuid=f842acdd-39c7-4678-bb9f-670592fef78c type=ext4
uuid=00000000-0000-0000-0000-000000000000 type=tmpfs
```

Vous trouverez les informations sur la structure `struc super_block` et sur la fonction `iterate_supers` dans le fichier *include/linux/fs.h*. Comme pour les exercices précédents, vous pourrez vous aider de LXR pour trouver des exemples d'utilisation.

Question 2

On souhaite désormais créer le module `update_sb`. Basé sur le code du précédent module, celui-ci doit afficher, lors de son chargement, la liste des superblocks d'un système de fichiers particulier passé en paramètre (ext4 par exemple). Il dois également être capable d'afficher la date de son dernier affichage.

Il va donc être nécessaire de conserver individuelement cette date pour chaque superblock. En effet, une partition *ext4* peu avoir été ajoutée après l'affichage des infos des superblocks de ce type. Dans ce cas sa date diffère de la date des autres superblocks *ext4*.

Pour cela, vous pourrez utiliser la fonction `get_fs_type` pour obtenir une structure qui représente un type de système de fichiers, et la fonction `iterate_super_type` qui parcours la liste des superblocks d'un type particulier.

Important : l'objet retourné par `get_fs_type` a un compteur de références. Vous **devez** rendre cette référence quand vous avez terminé, en utilisant la fonction `put_filesystem`.

Vous pourrez aussi utiliser la fonction `getnstimeofday` (*include/linux/timekeeping.h*) qui permet d'obtenir la date courante.

Au final, vous devriez obtenir un affichage similaire à celui-ci :

```
# insmod ./update_super.ko type=ext4
uuid=f842acdd-39c7-4678-bb9f-670592fef78c type=ext4 time=0.000000000
uuid=3413b1a2-7c1f-429e-9de8-1b4f52f23fd3 type=ext4 time=0.000000000
# rmmod update_super.ko
# insmod ./update_super.ko type=proc
uuid=00000000-0000-0000-0000-000000000000 type=proc time=0.000000000
# rmmod update_super.ko
# insmod ./update_super.ko type=ext4
uuid=f842acdd-39c7-4678-bb9f-670592fef78c type=ext4 time=1445346469.983057
uuid=3413b1a2-7c1f-429e-9de8-1b4f52f23fd3 type=ext4 time=1445346469.983058
```