PNL - 4I402 : Programmer dans le noyau

Version 17.01

Julien Sopena[1]

[1]julien.sopena@lip6.fr
Équipe RÉGAL - INRIA Rocquencourt
LIP6 - Université Pierre et Marie Curie

Master SAR 1ère année - 2017/2018

Grandes lignes du cours

Outline

Outline

Outline

Outline

Un module Linux c'est quoi ?

Definition
Un **module** est une bibliothèque chargée dynamiquement dans le noyau et pouvant générer un appel de fonction au moment de son chargement et de son déchargement.

Le noyau fourni deux macros pour enregistrer les fonctions à exécuter :

```
static int my_init(void) {
  ...
  return 0;
}
module_init(my_init);
```

```
static void my_exit(void) {
  ...
}
module_exit(my_exit);
```


Fichiers d'entêtes

Les modules doivent inclure un minimum de fichiers d'entêtes

```
/* API des modules */
#include <linux/module.h>

/* Si besoin : macro pour les fonctions init et exit */
#include <linux/init.h>

/* Si besoin : types, fonctions et macros de bases */
#include <linux/kernel.h>
```

Identification du module

Il est possible d'identifier le module en utilisant des macros spécifiques, le plus souvent placées au début du code source :

```
MODULE_DESCRIPTION("Hello World module");
MODULE_AUTHOR("Julien Sopena, LIP6");
MODULE_LICENSE("GPL");
```

```
modinfo helloworld.ko
    filename:       helloworld.ko
    description:    Hello World module
    author:         Julien Sopena, LIP6
    license:        GPL
    vermagic:       2.6.30-ARCH 686 gcc-4.4.1
    depends:
```


Exemple de module : helloworld.c

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/kernel.h>

MODULE_DESCRIPTION("Hello World module");
MODULE_AUTHOR("Julien Sopena, LIP6");
MODULE_LICENSE("GPL");

static int __init hello_init(void) {
        pr_info("Hello, world\n");
        return 0;
}
module_init(hello_init);

static void __exit hello_exit(void) {
        pr_info("Goodbye, cruel world\n");
}
module_exit(hello_exit);
```


Les macro `__init` et `__exit`

Definition

Les macros **`__init`** et **`__exit`** servent à optimiser l'emprunte mémoire du noyau, lorsque le code est compilé statiquement, en plaçant les fonctions dans des segments spécifiques :

- `.init.text` qui est supprimé après le boot du noyau ;
- `.exit.text` qui n'est jamais chargé.

Comme le montre leur définition dans `init.h`, ces macros sont tout simplement ignorées lorsque le code est compilé sous forme de module.

```
#ifndef MODULE
  #define __init   __attribute__ ((__section__ (".init.text")))
  #define __exit   __attribute__ ((__section__(".exit.text")))
#else
  #define __init
  #define __exit
#endif
```


Outline

Compiler un module

Le noyau est fourni avec un Makefile générique : `build/Makefile`

Le Makefile suivant construit `hello.ko` :

1. au lancement `KERNELRELEASE` n'est pas défini
2. récupération du répertoire courant
3. `make` sur le `Makefile` générique
   - 3.1 ce dernier défini un grand nombre de règles
   - 3.2 défini aussi des variables dont `KERNELRELEASE`
   - 3.3 source le makefile qui l'a lancé pour récupérer la(es) cible(s)
   - 3.4 compile le(s) module(s)

Pour les linux 2.4, l'extension des modules est **.o**.

À partir des noyaux 2.6 c'est **.ko** pour *kernel object*.

Compiler un module

```
ifneq ($(KERNELRELEASE),)

  obj-m += helloworld.o

else

  KERNELDIR ?= /lib/modules/$(shell uname -r)/build
  PWD := $(shell pwd)

all :
        make -C $(KERNELDIR) M=$(PWD) modules

clean:
        make -C $(KERNELDIR) M=$(PWD) clean

endif
```


Outline

Chargement et déchargement d'un module

Pour charger un module du noyau on utlise **insmod** :

```
insmod helloworld.ko
dmesg
   [177814.017370] Hello, world
```

Pour décharger un module du noyau on utlise **rmmod** :

```
rmmod helloworld
dmesg
   [177919.956567] Goodbye, cruel world
```

Outline

Exemple de module avec paramètres

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

static char *month = "January";
module_param(month, charp, 0660);
static int day = 1;
module_param(day, int, 0000);

static int __init hello_init(void) {
  pr_info("Hello ! We are on %d %s\n", day, month);
  return 0;
}
module_init(hello_init);

static void __exit hello_exit(void) {
  pr_info("Goodbye, cruel world\n");
}
module_exit(hello_exit);
```


Passer des paramètres aux modules

Par défaut les paramètres conservent leur valeur initiale :

```
insmod helloworld.ko
dmesg
  [180525.067016] Hello ! We are on 1 January
```

On peut passer les paramètres dans la ligne de commande :

```
insmod helloworld.ko month=December day=31
dmesg
  [181086.216097] Hello ! We are on 31 December
```

On peut aussi les fixer dans le fichier `/etc/modprobe.conf` :

```
options helloworld month=December day=31
```

```
modprobe helloworld
dmesg
  [181526.314020] Hello ! We are on 31 December
```


Outline

Edition dynamique des liens d'un module

Les modules sont chargés dynamiquement :

$\Longrightarrow$ **ils ne peuvent accéder qu'a des symboles qui ont explicitement été exportés pour eux.**

Par défaut il n'ont donc pas accès aux variables et aux fonctions du noyau même si celles-ci n'ont pas été déclarées comme `static` !

Le noyau offre deux macros pour exporter un symbole :

- `EXPORT_SYMBOL(s)` : le symbole est rendu accessible à tous modules chargé dans le noyau
- `EXPORT_SYMBOL_GPL(s)` : le symbole est accessible par tous les modules dont la licence est compatible avec la licence *GPL*

Exemple d'utilisation d'un symbole exporté

Voici le code d'un module `devil.c` qui éteint la machine en utilisant la fonction `pm_power_off()` du noyau Linux.

```c
#include <linux/module.h>
#include <linux/kernel.h>

MODULE_DESCRIPTION("Module qui eteint la machine");
MODULE_AUTHOR("Julien Sopena, LIP6");
MODULE_LICENSE("GPL");

static int __init devil_init(void)
{
        pr_info("C'est le debut de la fin >:)\n");

        if (pm_power_off)
                pm_power_off();

        return 0;
}
module_init(devil_init);
```


Exemple d'utilisation d'un symbole exporté

La compilation et le chargement du module `devil.ko` nécessite l'accès au symbole `pm_power_off()` du noyau.

Heureusement il est exporté dans `arch/x86/kernel/reboot.c`

```c
/*
 * Power off function, if any
 */
(*pm_power_off)(void);
EXPORT_SYMBOL(pm_power_off);
```

Le chargement du module provoque alors l'arrêt immédiat :

```
insmod devil.ko
   [  308.465263] C'est le debut de la fin >:)
   [  308.465557] acpi_power_off called
```


Outline

Action d'un module

Pour agir un module peut :

- modifier le comportement du noyau
  $\Longrightarrow$ modifier certaines structures ou pointeurs de fonction
- ordonnancer du code
  $\Longrightarrow$ créer un thread noyau ou ajouter un *work*
- offrir de nouvelles fonctionnalités pour d'autres modules
  $\Longrightarrow$ exporter un nouveau symbole

Dans ce dernier cas il doit utiliser les mêmes macros que le noyau :

- `EXPORT_SYMBOL(s)`
- `EXPORT_SYMBOL_GPL(s)`

Exemple de symbole exporté par un module

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

void print_var(const char *s, int i)
{
        pr_info("La valeur de \"%s\" est %d\n", s, i);
}
EXPORT_SYMBOL(print_var);

static int my_init(void)
{
        pr_info("Ajout de la fonction \"print_var\"\n");
        return 0;
}
module_init(my_init);

static void my_exit(void)
{
        pr_info("Dechargement de \"print_var\"\n");
}
module_exit(my_exit);
```


Outline

Dépendances de modules

Definition
Un module A **dépend** d'un module B si A utilise au moins un des symboles exportés par B.

Les dépendances des modules n'ont pas à être spécifiées explicitement par le créateur du module. Elles sont déduites automatiquement lors de la compilation du noyau.

Les dépendances des modules sont stockées dans :
`/lib/modules/<version>/modules.dep`

Ce fichier est mis à jour avec **depmod** :

```
depmod -a [<version>]
```


Exemple de module dépendant d'un autre module

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

extern void print_var(const char *s, int i);

static int day = 1;
module_param(day, int, 0000);

static int __init hello_init(void) {
  print_var("day", day);
  return 0;
}
module_init(hello_init);

static void __exit hello_exit(void) {
  printk(KERN_ALERT "Goodbye, cruel world\n");
}
module_exit(hello_exit);
```


Exemple de module avec dépendance

Avec `insmod` il faut veiller à insérer les modules dans l'ordre :

```
insmod helloworld.ko day=15
    insmod: ERROR: could not insert module helloworld.ko:
      Unknown symbol in module
```

```
insmod my_print.ko
insmod helloworld.ko day=15
dmesg
  [184644.868138] Ajout de la fonction "print_var"
  [184645.252412] La valeur de "day" est 15
```

Autre solution, utiliser `modprobe` après instalation des modules :

```
modprobe -v helloworld
    insmod /lib/modules/version\_noyau/extra/my\_print.ko
    insmod /lib/modules/version\_noyau/extra/helloworld.ko
```


Contrôle des dépendances au déchargement

Pour chaque module chargé, le noyau maintient la liste des modules qui utilisent au moins un de ses symboles exportés :

```
lsmod | grep helloworld
    helloworld             16384  0
    my_print               16384  1 helloworld
```

Au déchargement le noyau vérifie que cette liste est bien vide.

```
rmmod my_print
    rmmod: ERROR: Module my_print is in use by: helloworld
```

```
rmmod helloworld
rmmod my_print
dmesg
   [186513.878338] Goodbye, cruel world
   [186517.906542] Déchargement de la fonction "print_var
```


Outline

Module or not module : Opter pour le module.

Lorsque l'on développe une fonctionnalité pour le noyau, on peut :

- intégrer son **code dans le noyau** au travers d'un patch :

  elle est alors intégrée statiquement au noyau

- créer un nouveau **module** :

  elle pourra être chargée dynamiquement dans le noyau

Règle de choix

Pour maximiser ses chances d'intégration, il faut toujours choisir la solution du **module** lorsqu'elle est techniquement possible.

Avantages et limites des modules.

Avantages :

- Plus simple à développer ;
- Simplifie la diffusion ;
- Évite la surcharge du noyau ;
- Permet de résoudre les conflits ;
- Pas de perte en performance.

Limites :

- On ne peut pas modifier les structures internes du noyau. Par exemple, ajouter un champs dans le descripteur des processus ;
- Remplacer une fonction lié statiquement au noyau. Par exemple, modifier la manière dont les cadres de page sont alloués ;