

PNL - MU4IN402 : Implémenter un module noyau Version 21.01

Julien Sopena¹

¹julien.sopena@lip6.fr
Équipe Delys - INRIA Paris
LIP6 - Sorbonne Université / CNRS

Master SAR 1^{ère} année - 2020/2021

Mise en garde

Méthodologie de développement dans le noyau

Les modules

- Mon premier module

- Compilation d'un module

- Chargement d'un module

- Module avec paramètres de chargement

- Liens avec le noyau

- Export : offrir de nouveaux symboles

- Dépendance : utiliser des symboles externes

- Patch ou module ?

Mise en garde

Méthodologie de développement dans le noyau

Les modules

Les dangers de la programmation noyau

Travailler directement au cœur du noyau peut le rendre instable et engendrer un `KERNEL PANIC`, le rendant donc inutilisable !

Avant toute installation d'un nouveau noyau, il est conseillé d'en enregistrer un autre dans le *bootloader* qui pourra servir de démarrage de secours.

Il est conseillé, si possible (voir plus loin), de travailler son code sous forme de modules qui pourront être chargés dynamiquement dans un système stable.

Mise en garde

Dans tous les cas, gardez à l'esprit qu'un bogue peut corrompre votre système de fichier ou un driver de périphériques.

Vous pouvez donc **perdre définitivement toutes les données** stockées dans un périphérique connecté à votre système.

Mise en garde

Méthodologie de développement dans le noyau

Les modules

Méthode 1 : travail en local.

La méthode la plus simple pour développer du code noyau, reste de **travailler en local** sur sa machine et de (dé)charger manuellement sur le noyau en courant.

Plusieurs méthodes existent pour récupérer des informations de débogage :

```
sudo tail -f /proc/kmsg
```

```
sudo tail -f /var/log/messages
```

```
dmesg [ -c ] [ -n niveau ] [ -s taille ]
```

```
syslogd
```

Méthode 1 : travail en local.

Avantage : facile à mettre en place et à utiliser.

Inconvénient : si le noyau devient instable, il peut être nécessaire

de rebooter. À préconiser pour de petits drivers, mais à déconseiller vivement pour des drivers complexes (réseau ou *vfs* par exemple).

C'est envisageable pour des modules, mais s'il est nécessaire de modifier le cœur même du noyau, alors cette technique est à éviter.

User Mode Linux ou **UML** est un noyau Linux compilé qui peut être exécuté dans l'espace utilisateur comme un simple programme. Il permet donc d'avoir plusieurs systèmes d'exploitation virtuels (principe de virtualisation) sur une seule machine physique hôte exécutant Linux.

Méthode 2 : User Mode Linux.

Avantage :

- ▶ Lancer des noyaux sans avoir besoin de redémarrer la machine.
- ▶ Si un *UML* plante, le système hôte n'est pas affecté.
- ▶ Un utilisateur sera root sur un *UML*, mais pas sur l'hôte.
- ▶ *gdb* peut servir à déboguer le noyau en développement puisqu'il est considéré comme un processus normal.
- ▶ Il permet de mettre en place un réseau complètement virtuel de machines Linux, pouvant communiquer entre elles. Il est alors possible des fonctionnalités réseau.

Inconvénient :

- ▶ Très lent, plutôt conçu pour des tests fonctionnels que pour la performance.
- ▶ Nécessite de patcher le noyau

Méthode 2 BIS : Kernel Mode Linux (KML).

Le **Kernel Mode Linux (KML)** est la technique réciproque de UML, permet d'exécuter dans le noyau un processus habituellement prévu pour l'espace user.

Tout comme pour UML, cela nécessite de patcher le noyau et d'activer la fonctionnalité lors de la Compilation du noyau. Les architectures supportées sont : *IA-32* et *AMD64*.

Actuellement, les binaires ne peuvent pas modifier les registres suivants : *CS*, *DS*, *SS* ou *FS*.

Méthode 3 : machine virtuelle.

Le développement de code noyau peut aussi se faire dans une **machine virtuelle**, s'exécutant au-dessus d'un OS "classique". Ces machines virtuelles sont dites de type 3 : elles permettent d'émuler une machine nue de façon à avoir un système d'exploitation à l'intérieur d'un autre. Les deux systèmes peuvent alors être différents.

Parmi de telles architectures, on peut citer :

- ▶ *QEMU*,
- ▶ *VMWARE*,
- ▶ *BOCHS*,
- ▶ *VirtualBox*.

Méthode 3 : machine virtuelle.

Voici un code permettant d'utiliser **QEMU** :

```
# Création du rootfs
mkdir iso
# Création de l'image ISO
mkisofs -o rootfs-dev.iso -J -R ./iso
# Cela peut être une copie d'un média
dd if=/dev/dvd of=dvd.iso # pour un dvd
dd if=/dev/cdrom of=cd.iso # pour un cdrom
dd if=/dev/scd0 of=cd.iso # pour cdrom scsi
# Simulation
qemu -boot d -cdrom ./rootfs-dev.iso
# Montage
sudo modprobe loop
sudo mount -o loop rootfs-dev.iso /mnt/disk
# Démontage
sudo umount mnt/disk
```

Méthode 3 : machine virtuelle.

Avantages :

- ▶ Très pratique pour développer à l'intérieur du noyau.
- ▶ Pas besoin de patcher le noyau comme avec *UML*.

Inconvénient :

- ▶ Dépend de la puissance de la machine hôte.
- ▶ Performance liée à la présence d'une virtualisation matérielle (type *Intel-VT*).
- ▶ Peut nécessiter de régénérer l'image à chaque fois que l'on souhaite la tester.

Méthode 4 : via une seconde machine.

La dernière méthode consiste à travailler sur une **seconde machine de développement**. Cette machine est reliée à la machine principale qui peut alors la monitorer.

De loin la technique la plus adaptée, car permet de développer au coeur du noyau ou bien des modules complexes.
Cette technique est de plus adaptée pour un usage embarqué.

Méthode 4 : via une seconde machine.

Avantages :

- ▶ Très pratique pour des développements sur le noyau même.
- ▶ Permet de déboguer (via le patch kdb et l'utilitaire kgdb) via la liaison série ou le réseau le noyau courant du second système en pouvant poser un point d'arrêt.

Inconvénient :

- ▶ Nécessite de disposer d'une seconde machine.

Sur la plate-forme de développement

- ▶ Pas de problème. Vous pouvez utiliser un convertisseur USB<-> série. Bien supporté par Linux. Ce périphérique apparaît en tant que `/dev/ttyUSB0`.

Sur la cible :

- ▶ Vérifiez si vous avez un port IrDA. C'est aussi un port série.
- ▶ Si vous avez une interface Ethernet, essayez de l'utiliser.
- ▶ Vous pouvez aussi connecter en JTAG directement les broches série du processeur (vérifiez d'abord les spécifications électriques!).

Mise en garde

Méthodologie de développement dans le noyau

Les modules

- Mon premier module

- Compilation d'un module

- Chargement d'un module

- Module avec paramètres de chargement

- Liens avec le noyau

- Export : offrir de nouveaux symboles

- Dépendance : utiliser des symboles externes

- Patch ou module ?

Mise en garde

Méthodologie de développement dans le noyau

Les modules

- Mon premier module

- Compilation d'un module

- Chargement d'un module

- Module avec paramètres de chargement

- Liens avec le noyau

- Export : offrir de nouveaux symboles

- Dépendance : utiliser des symboles externes

- Patch ou module ?

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 fournit deux macros pour enregistrer les fonctions à exécuter :

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

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

Les modules doivent inclure un minimum de fichiers d'en-têtes

```
/* 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 macros `__init` et `__exit`

Definition

Les macros `__init` et `__exit` servent à optimiser l'empreinte 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
```

Mise en garde

Méthodologie de développement dans le noyau

Les modules

- Mon premier module

- Compilation d'un module

- Chargement d'un module

- Module avec paramètres de chargement

- Liens avec le noyau

- Export : offrir de nouveaux symboles

- Dépendance : utiliser des symboles externes

- Patch ou module ?

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 définit un grand nombre de règles
 - 3.2 définit 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_PNL ?= /lib/modules/$(shell uname -r)/build
    PWD := $(shell pwd)

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

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

endif
```

Mise en garde

Méthodologie de développement dans le noyau

Les modules

- Mon premier module

- Compilation d'un module

- Chargement d'un module

- Module avec paramètres de chargement

- Liens avec le noyau

- Export : offrir de nouveaux symboles

- Dépendance : utiliser des symboles externes

- Patch ou module ?

Chargement et déchargement d'un module

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

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

Pour décharger un module du noyau on utilise **rmmmod** :

```
$ rmmmod helloworld
$ dmesg
[177919.956567] Goodbye, cruel world
```

Mise en garde

Méthodologie de développement dans le noyau

Les modules

- Mon premier module

- Compilation d'un module

- Chargement d'un module

- Module avec paramètres de chargement

- Liens avec le noyau

- Export : offrir de nouveaux symboles

- Dépendance : utiliser des symboles externes

- Patch ou module ?

Exemple de module avec paramètres

```
#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
```

Mise en garde

Méthodologie de développement dans le noyau

Les modules

- Mon premier module

- Compilation d'un module

- Chargement d'un module

- Module avec paramètres de chargement

- Liens avec le noyau

- Export : offrir de nouveaux symboles

- Dépendance : utiliser des symboles externes

- Patch ou module ?

Edition dynamique des liens d'un module

Les modules sont chargés dynamiquement :

⇒ ils ne peuvent accéder qu'à 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 chargés 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.

```
#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` nécessitent l'accès au symbole `pm_power_off()` du noyau.

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

Mise en garde

Méthodologie de développement dans le noyau

Les modules

- Mon premier module

- Compilation d'un module

- Chargement d'un module

- Module avec paramètres de chargement

- Liens avec le noyau

- Export : offrir de nouveaux symboles

- Dépendance : utiliser des symboles externes

- Patch ou module ?

Action d'un module

Pour agir, un module peut :

- ▶ modifier le comportement du noyau
⇒ modifier certaines structures ou pointeurs de fonction
- ▶ ordonnancer du code
⇒ créer un thread noyau ou ajouter un *work*
- ▶ offrir de nouvelles fonctionnalités pour d'autres modules
⇒ 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

```
#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);
```

Mise en garde

Méthodologie de développement dans le noyau

Les modules

- Mon premier module

- Compilation d'un module

- Chargement d'un module

- Module avec paramètres de chargement

- Liens avec le noyau

- Export : offrir de nouveaux symboles

- Dépendance : utiliser des symboles externes

- Patch ou module ?

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

```
#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 dépendant d'un autre module

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 installation 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"
```

Mise en garde

Méthodologie de développement dans le noyau

Les modules

- Mon premier module

- Compilation d'un module

- Chargement d'un module

- Module avec paramètres de chargement

- Liens avec le noyau

- Export : offrir de nouveaux symboles

- Dépendance : utiliser des symboles externes

- Patch ou module ?

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 champ dans le descripteur des processus ;
- ▶ Remplacer une fonction liée statiquement au noyau. Par exemple, modifier la manière dont les cadres de page sont alloués ;