NMV : Architecture du noyau Linux et construction de modules.

Version 1.05

Julien Sopena[1]

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

Master SAR 2ème année - NMV - 2012/2013

Grandes lignes du cours

Outline

Les différents types de noyaux

On distingue généralement 4 grands types de noyaux :

- Les noyaux monolithiques,
- Les micro-noyaux,
- Les noyaux monolithiques modulaires,
- Les exo-noyaux.

Noyaux monolithiques : Définition.

Définition

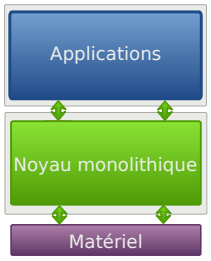
*Un **noyau monolithique** est un système d'exploitation dont l'ensemble des fonctionnalités du système et des pilotes sont regroupés dans un seul bloc de code et un seul bloc binaire généré à la compilation.*

Exemple

Premières versions de Linux, certains BSD ou certains anciens Unix.
Mais aussi des systèmes embarqués critiques : IOS de Cisco

L'aspect monolithique ne concerne pas forcement les sources qui peuvent être architecturées en bibliothèques bien distinctes les unes des autres. Cependant ce code très intégré est aussi compliqué à concevoir qu'à développer correctement.

Noyaux monolithiques : Schéma.



Noyaux monolithiques : Caractéristiques.

- S'exécute entièrement en *kernel space* $\Rightarrow \begin{cases} + \text{ Performances.} \\ -/+ \text{ Sécurité.} \end{cases}$
- Code optimisé pour une architecture $\Longrightarrow \begin{cases} + \text{ Performances.} \\ - \text{ Portabilité.} \end{cases}$
- Chargé entièrement en mémoire $\Rightarrow$ Problème de place mémoire
  $\Rightarrow$ Limite les fonctionnalités

Micro-noyaux : Définitions.

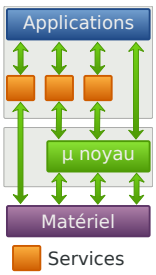
Définition

*Un **micro-noyau**, est le cœur d'un système d'exploitation cherchant à minimiser les fonctionnalités intégrées au noyau en plaçant la plus grande partie des services du système d'exploitation à l'extérieur de ce noyau, c'est-à-dire dans l'espace utilisateur. Ces fonctionnalités sont alors fournies par de petits serveurs indépendants possédant souvent leur propre espace d'adressage.*

Définition

*Un **micro-noyau enrichi** est l'ensemble logiciel formé par le micro-noyau et les services déplacés en espace utilisateur.*

Micro-noyaux : Schéma.



Micro-noyaux : Caractéristiques.

- Encombrement mémoire très faible ⇒ Systèmes embarqués
- Services bas-niveau hors-noyau ⇒
  - \+ Sécurité renforcée.
  - \+ Souplesse.
  - \- Performances limitées.
- Interaction micro-noyau/services par une interface standard ⇒
  - \+ Portabilité très importante.
  - \- Bride les capacités et la diversité.

Noyaux modulaires monolithiques : Définition.

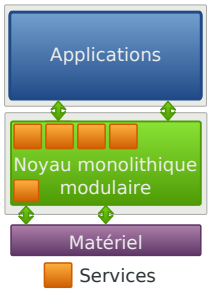
Définition

*Un **noyau monolithique modulaire** est un système d'exploitation dont les parties fondamentales du système sont regroupées dans un bloc de code unique (monolithique). Les autres fonctions, comme les pilotes matériels, sont regroupées en différents modules qui peuvent être séparés tant du point de vue du code que du point de vue binaire. Mais une fois chargées, ces fonctions s'exécutent en kernel space et partagent le même espace d'adressage que le coeur du noyau.*

Exemple

Linux, la plupart des BSD ou encore Solaris.

Noyaux modulaires monolithiques : Schéma.



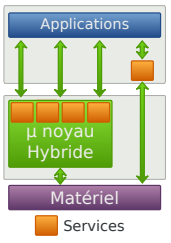
Noyaux modulaires monolithiques : Caractéristiques.

- Interaction direct avec le matériel ⟹
  - \+ Performances.
  - \- Portabilité.
- Code optimisé pour une architecture ⟹
  - \+ Performances.
  - \- Portabilité.
- Services sous forme de module ⟹
  - \+ Extensible.
  - \+ Portabilité.
- Modules exécutés en *kernel space* ⟹
  - \+ Performances.
  - \- **Sécurité.**

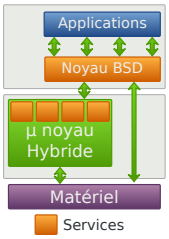
Micro-noyaux hybride : Définitions.

Définition

*Un **micro-noyau hybride**, est un micro noyau qui intègre directement sous forme de module certains services cruciaux, le reste des services restants à l'extérieur du noyau.*



Micro-noyaux hybride.



Exemple de Mach/XNU le "micro-noyau" d'apple".

Exo-noyaux : définition.

Définition

*Un **exo-noyau**, est un noyau minimal qui effectue uniquement un multiplexage sécurisé des ressources matérielles disponibles. Les applications peuvent sélectionner et utiliser des librairies formant le reste système d'exploitation ou implémenter les leurs.*

Exemple

Concept né du projet Aegis du MIT.

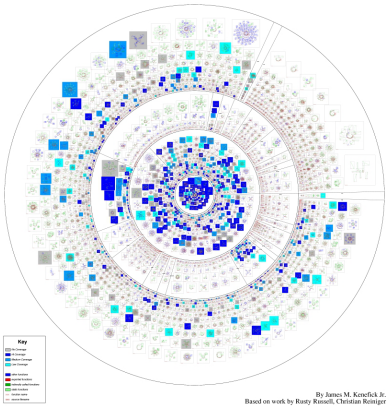
Si les *micro-noyaux* remonte le matériel au niveau d'une abstraction standard (*HAL*), les *exo-noyaux* descendent les applications à un niveau standard du matériel.

Exo-noyaux : Caractéristiques.

- Interaction direct avec le matériel ⟹
  - \+ Performances.
  - \- Portabilité.
- L'ensemble des services en *user mode* ⟹ Sécurité très importante.
- Gestion de ressources physiques au niveau applicatif ⇒
  - \+ Limite les contraintes
  - \+ Permet des applications spécifiques
  - \- Demande un nouveau type de programmation

Outline

Linux une Architecture en oignon.



Les fonctionnalités du noyau Linux.

Un noyau Linux offre 5 grandes fonctionnalités :

1. Gestion des processus
2. Gestion de la mémoire
3. Gestion des services réseaux
4. Gestion du stockage
5. Gestion de l'interface avec l'utilisateur

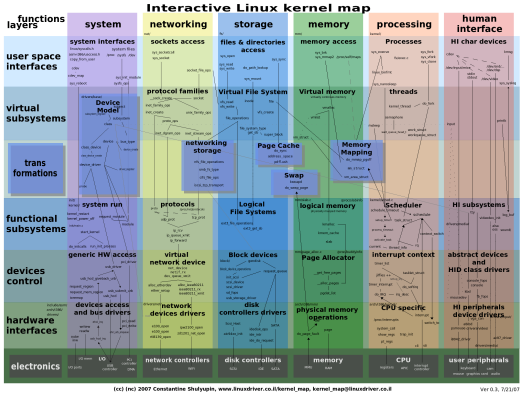
Les couches du noyau Linux.

Pour chacune des fonctionnalités du noyau Linux, on retrouve la même série d'abstractions :

1. Couche d'interaction avec le matériel
2. Couche d'abstraction du matériel
3. Couche d'interopérabilité entre les fonctionnalités
4. Couche de services
5. Couche d'abstraction des services

Vue simplifiée de l'architecture du noyau Linux.

Diagramme matriciel du noyau GNU Linux.



Outline

Qu'est ce que le VFS ?

Point de vue utilisateur :
Une interface unifiée vers différents sytème de fichiers, comme par exemple : ext2/ext3, ReiserFS, iso9660 (cdrom), NFS, ...

Point de vue architecte OS :
Un moyen de factoriser le code fiabilité/robustesse + allègement du code.

Point de vue developpeur pilotes :
Une API « classique », des services fournis par défaut.

VFS : une Conception Orientée Objet

Comme le reste du noyau le VFS est **codé en C** mais avec une **conception orientée objet**

Objet = une structure contenant :

- Des attributs (données)
- Des méthodes (pointeurs vers des fonctions)

Exemple d'objet : la structure **file** associée à un fichier ouvert

- Attributs, par exemples :
  - uid/gid du(des) processus ayant ouvert le fichier
  - mode : attributs d'ouverture (R/W/Append, ...)
  - offset : position courante (prochaine lecture/écriture)
- Méthodes, par exemples :
  - read, write, llseek, ioctl, mmap, poll, flush, open, ...

Les Principaux Objets de VFS

Le VFS est composé de quatres types d'objet :

1. **Super Block :**
   - Informations globales sur le support
   - Etat courant de la partition
   - Définie dans <linux/fs.h>
2. **I-node :**
   - Informations sur le fichier
   - Permissions, dates, adresse des blocks de données, ...
   - Définie dans <linux/fs.h>
3. **D-entry :**
   - Informations sur le lien (chemin vers le fichier)
   - Essentiellement pour des raison de perf (cache)
   - Définie dans <linux/dcache.h>
4. **File :**
   - Informations sur un fichier ouvert
   - Définie dans <linux/fs.h>

Architecture du VFS vue par un processus.

La structure : `struct file`

- Liens vers autres structures

```
struct list_head f_list;
struct dentry *f_dentry;
struct vfsmount *f_vfsmnt;
```

- Contrôle du fichier (opérations d'E/S)

```
struct file_operations *f_op;
```

- Etat courant du fichier

```
atomic_t f_count;
unsigned int f_flags;
mode_t f_mode;
loff_t f_pos;
unsigned long f_reada, f_ramax, f_raend, f_ralen, f_rawi
struct fown_struct f_owner;
unsigned int f_uid, f_gid;
int f_error;
```


Opérations sur Fichier Ouvert

Une opération pour chaque appel système concernant les fichiers déja ouverts, plus quelques « spécialités exotiques » :

```
llseek(file,offset,origin)
read(file,buf,count,offset)
write(file,buf,count,offset)
readdir(file,dirent,filldir_fn)
poll(file,poll_table)
ioctl(inode,file,cmd,arg)
mmap(file,vma)
open(inode,file)
flush(file)
release(inode,file)
fsync(file,dentry)
fasync(fd,file,on)
lock(file,cmd,file_lock)
readv(file,vector,count,offset)
writev(file,vector,count,offset)
sendpage(file,page,offset,size,pointer,fill)
get_unmapped_area(file,addr,len,offset,flags)
```


Lien Entre Appel Système et Opération

Les fonctions de la `libc` encapsule les opérations :

1. En mode utilisateur
   - 1.1 Processus invoque operation (`open`,`read`, ...)
   - 1.2 `libc` prépare bascule en mode noyau
   - 1.3 déclenchement bascule ...
2. En mode noyau :
   - 2.1 point d'entrée : `sys_xxx` (`sys_open()`, `sys_read()`, ...)
   - 2.2 `sys_xxx()` fait des vérifications d'usage
   - 2.3 `sys_xxx()` invoque le traitement correspondant du VFS
   - 2.4 Gestion du retour :
     - retour > 0 : ok
     - retour < 0 : `-ERROR` (ex : `-EPERM`)
3. Retour au mode utilisateur :
   - 3.1 `libc` (mode ut) si retour < 0 : `errno` = retour, retour -1

Déroulement d'une Ouverture de Fichier :

`sys_open(filename,flags,mode)`

1. `getname()` : copie filename depuis espace utilisateur
2. `get_unused_fd()` : recherche decripteur libre
3. `filp_open(pathname,flags,mode)`
   - 3.1 `open_namei(pathname,flas,&nameidata)`
     - 3.1.1 `path_lookup(pathname, flags, &nameidata)` place dans nameidata **d_entry** correspondant à pathname (ou répertoire parent si création)
     - 3.1.2 nombreuses vérifications (verrous, permissions, ...)
   - 3.2 `dentry_open(dentry,mnt,flags)`
     - 3.2.1 allocation et initialisation de la `struct file` : `f_flags`, `f_mode`, `f_dentry`, `f_pos`, `f_count=1`, ...
     - 3.2.2 `f_op = d_entry->d_inode->i_fop`
     - 3.2.3 `struct file` chaîné dans liste `superblock`
4. ref vers `struct file` dans contexte processus (`files->fd[fd]`)
5. retourne `fd` (ou `-ERRNO`)

Déroulement d'une Lecture sur un fichier :

`sys_read(fd,buf,count)`

1. `fget(fd)`
   - 1.1 Incrémente le compteur utilisation (`f_count++`)
   - 1.2 retourne `struct file`
2. Opération autorisée (`f_mode`) ?
3. `locks_verify_area()` -> mandatory lock ?
4. invoque `file->f_op->read()`
   - 4.1 lecture depuis périphérique
   - 4.2 copie vers espace utilisateur
   - 4.3 met à jour `f_pos`
5. `fput()` : `file->f_count --`
6. Retourne nb octets effectivement transférés (ou `-ERRNO`)

Déroulement de la fermeture d'un fichier ouvert :

`sys_close(fd)` :

1. Retrouve struct file associée à `fd` à partir du contexte
2. Annule l'entrée de la `struct file` dans contexte
3. invoque `filp_close()`
   - 3.1 Invoque `file->f_op->flush()`
   - 3.2 Relâche les éventuels verrous
   - 3.3 Invoque `fput (f_count --)`
4. Si `f_count = 0`
   - 4.1 déclenche `release(inode,file)`
5. Si `nb_lien = 0` (car `unlink()` après ouverture)
   - 5.1 destruction physique du fichier
6. retourne code retour de `flush()`

Créer un Nouveau Système de fichiers

1. Créer un structure `file_system_type`
   - Ecrire une méthode `read_super()`
     - Remplir la structure `super_block`
     - Fournir les méthodes du `super_block` (champ `s_op`), en particulier `read_inode()`
2. Fournir les méthodes à surcharger au niveau de l'inode
   - Méthodes sur inodes : `struct inode_operations`
     - En général on fournit au moins lookup()
   - Méthodes sur fichiers ouverts : `struct file_operations`
   - Attachées à chaque inode par `read_super()`
3. Enregister la structure `file_system_type`
   - `register_file_system()`

Systèmes de fichiers exotiques

Remarque(s)

Normalement, un système de fichiers s'appuie sur un périphérique de stockage. Mais certains inventent/fabriquent leurs fichiers dynamiquement

Exemple

Le système de fichiers virtuels **procfs** :

- fabrique un répertoire exotique pour chaque processus
- fournit des répertoires et fichiers exotiques contenant :
  - Informations sur le matériel
  - Configuration du matériel

Principe de mise en oeuvre : `read_inode()` fournit des méthodes différentes selon l'inode demandée.

Outline

Liste des étapes pour générer un noyau

Étape pour la génération d'un noyau :

1. Récupération du code source du noyau et de ses patch
2. Vérification de l'intégrité des packages reçu
3. Application des patch sur le code source
4. Génération d'une configuration
5. Compilation
6. Installation

Étape 0 : Paquets nécessaires pour la compilation

Pour installer le noyau 2.6.x, assurez-vous d'avoir les paquets suivants (version minimum) :

- la librairie ncurses-5, certaines distributions l'appellent libncurses5 et libncurses5-dev (ou libncurses5-devel)
- l'utilitaire bzip2
- l'utilitaire gzip
- Gnu gcc 2.95.3 (commande : gcc –version)
- Gnu make 3.78 (commande : make –version)
- binutils 2.12 (commande : ld -v)
- util-linux 2.10 (commande : fdformat –version)
- module-init-tools 0.9.10 (commande : depmod -V)
- procps 3.1.13 (commande : ps –version)

Étape 1 : récupération d'une copie des sources officiels.

Télécharger les sources depuis le site :
http ://kernel.org/ (maintenu par la société Transmeta)

```
wget ftp://ftp.kernel.org/pub/linux/kernel/v2.6/linux-2.6.30.1.tar.gz
wget ftp://ftp.kernel.org/pub/linux/kernel/v2.6/linux-2.6.30.1.tar.gz.sign
```

Il peut être aussi nécessaire de récupérez une mise à jour (ensemble de correctifs) pour la version x.y.<z-1> :

```
wget ftp://ftp.kernel.org/pub/linux/kernel/v2.6/patch-2.6.30.1.bz2
wget ftp://ftp.kernel.org/pub/linux/kernel/v2.6/patch-2.6.30.1.bz2.sign
```


Étape 2 : vérification de l'intégrité des sources.

Le noyau est le code le plus critique du système, il est donc indispensable de vérifiez l'intégrité des sources :

```
gpg --verify linux-2.3.9.tar.gz.sign linux-2.3.9.tar.gz
gpg: Signature made Mon Oct 9 23:48:38 2000 PDT
     using DSA key ID 517D0F0E
gpg: Good signature from
     "Linux Kernel Archives Verification Key <ftpadmin@ke
```

Étape 3 : application des patchs (si nécessaire).

Commande patch : utilise la sortie (stdout) de la commande diff pour appliquer un ensemble de changements à une arborescence de fichiers sources.

```
patch -p<nb> < fichier_diff
  n: nombre de niveaux de répertoires à sauter
```

L'argument -p<nb> indique de tronquer <nb> répertoires dans les noms de fichiers affectés par le patch : utile si un patch s'applique à une arborescence qui n'est pas la même que la votre. Patches

Linux :

- Toujours à appliquer sur la version x.y.<z-1>
- Toujours prévu pour n=1 : patch -p1 < linux_patch

diff & patch

- diff : Comparaison de fichiers ligne par ligne
  - indique les lignes ajoutées ou supprimées ;
  - peut ignorer les casses, les tabulations, les espaces ;
  - option -u pour créer des patchs unifiés, avec plus d'informations.
- patch : Utilise la différence entre deux fichiers pour passer d'une version à l'autre.

Exemple

```
$ diff toto.c toto-orig.c > correction.patch
$ bzip2 correction.patch
$ bzcat correction.patch.bz2 | patch -p 0 toto.c
```


Création d'un patch : utilise diff

Les diff peuvent comparer des hiérarchies de fichiers (option -r) :

Exemple

```
$ cp -r linux-2.6.17 linux-2.6.17-orig
$ cd linux-2.6.17
$ vim [...]
$ cd ..
$ diff -r -u linux-2.6.17-orig linux-2.6.17 > \
      network-driver-b44.patch
$ gzip -9 network-driver-b44.patch

$ cd linux-2.6.17
$ zcat network-driver-b44.patch.gz | patch -p 1
```

L'option -p permet de laisser les chemins du patch au complet 0, en supprimant le premier dossier 1, etc.

Création d'un patch noyau

1. Téléchargez la dernière version des sources du noyau sur lequel vous travaillez.
2. Faites une copie de ces sources :

   ```
   rsync -a linux-2.6.9-rc2/ linux-2.6.9-rc2-patch/
   ```

3. Appliquez vos modifications aux sources copiées et testez les.
4. Créez un fichier correctif («patch») :

   ```
   diff -Nru linux-2.6.9-rc2/ linux-2.6.9-rc2-patch/ > pat
   ```

   Patchfile doit suivre une charte de nommage rappelant la version du noyau prise comme référence, le(s) bug(s) corrigé(s).
5. Comparez toujours la structure complète des sources (utilisable par patch -p1) Nom du fichier correctif : doit rappeler le problème résolu

Appliquer un retro-patch

A noter qu'il est possible d'inverser un patch de la façon suivante :

```
patch -R -p1 < ../patch-x.y.z
```


Étape 4 : définition la configuration du noyau

Éditer le Makefile pour définir la version et l'architecture de la cible :

```
VERSION = 2
PATCHLEVEL = 6
SUBLEVEL = 7
EXTRAVERSION = -monRootKit
```

La variable *EXTRAVERSION* permettra de distinguer l'image de votre noyau avec d'autres, compilées à partir des mêmes sources :

```
uname -r
  2.6.7-monRootKit
```


Étape 4 : définition la configuration du noyau

Définition

*La **configuration** d'un noyau, consiste à définir quelles fonctionnalités y seront intégrées et le cas échéant si elles le seront statiquement ou sous forme de module.*

Plusieurs méthodes peuvent être utilisées :

- édition de la configuration à la main.
- make config : mode texte.
- make menuconfig : interface ncurses.
- make oldconfig : chargement d'une ancienne configuration (demande des valeurs pour les nouveaux symboles).
- make xconfig (interface X utilisant *Qt*)
- make gconfig (interface X utilisant *Gtk*)

Où trouver la configuration actuelle du noyau

Les symboles de configuration du noyau sont stockés dans le fichier *.config* à la racine des sources.

Les fichiers de config des distributions sont généralement copiés dans /boot/ avec le binaire du noyau.

Pour un noyau 2.6 (s'il a été compilé avec l'option *CONFIG_IKCONFIG_PROC = y*), on peut aussi récupérer la configuration actuelle à partir du fichier virtuel suivant :

```
sudo zcat /proc/config.gz > /usr/src/linux/.config
```

Connaître son matériel

Pour connaître son matériel, on peut utiliser :

- lshwd, lspci, lsusb, ...
- dmidecode
- hdparam
- cat /proc/cpuinfo, /proc/meminfo, ...
- Et surtout dmesg

Étape 5 : compilation et installation du noyau

Compilation, est la phase la plus longue du processus :
45 minutes pour compiler un noyau 2.6.15.4 (38 Mo compressé) sur un portable Pentium 4 3,2 GHz avec 512 Mo de RAM.

Adapter la configuration de votre noyau en ne choisissant que les modules nécessaires à votre matériel. Cela peut diviser le temps de compilation par 30 et économiser des centaines de MB !

Pour un 2.4 :

```
make dep && make clean && make bzImage && make modules
```

Pour un 2.6 :

```
make && make modules_install && make install
```


Fichiers générés après la compilation

- **vmlinux :** Image brute du noyau Linux, non compressée
- **System.map :** Fichier de mapping des symboles Listes des adresses des symboles des primitives inclues dans le noyau linux compilé.
- **arch/<arch>/boot/zImage :** Image du noyau compressée avec zlib
- **arch/<arch>/boot/bzImage :** Image du noyau compressée aussi avec zlib. Généralement suffisamment petite pour tenir sur une disquette ! Image par défaut sur i386.

La commande file donne certaines informations sur le noyau linux :

```
file /boot/vmlinuz-2.6.17-10mdv
  /boot/vmlinuz26: Linux kernel x86 boot executable bzImage,
                   version 2.6.30-ARCH (root@T-POWA-LX) #1,
                   RO-rootFS, root_dev 0x905, swap_dev 0x1
```


Fichiers installés après un make install

- **/boot/vmlinuz-<version> :** Image du noyau
- **/boot/System.map-<version> :** Stocke les adresses des symboles (primitives systèmes) du noyau
- **/boot/initrd-<version>.img :** Initial RAM disk, contenant les modules nécessaires pour monter le système de fichier root. make install lance mkinitrd !
- **/etc/grub.conf ou /etc/lilo.conf :** make install met à jour les fichiers de configuration de votre bootloader pour supporter votre nouveau noyau ! Il relance /sbin/lilo si LILO est votre bootloader.

Fichiers installés après un make module_install

- **/lib/modules/<version>/ :** Modules noyau et autres fichiers
- **build/ :** Tout ce qui est nécessaire pour construire des modules pour ce noyau : fichier .config (build/.config), informations sur les symboles des modules(build/module.symVers), headers du noyau (build/include/)
- **kernel/ :** Fichiers modules .ko (Kernel Object), avec la même structure de répertoires que dans les sources.
- **/lib/modules/<version>/ :** Aliases des modules pour insmod et modprobe. Exemple :

```
alias sound-service-?-0 snd_mixer_oss.
```

- **modules.dep :** Dépendances des modules pour insmod et modprobe. Aussi utilisé pour ne copier que les modules nécessaires dans un système de fichier minimal.
- **modules.symbols :** Dit à quel module appartient un symbole donné.

Symboles des primitives du noyau

Exemple de fichier de mapping des symboles des primitives du noyau

```
cat /boot/System.map
  00100000 A phys_startup_32
  bfffe400 A __kernel_vsyscall
  bfffe410 A SYSENTER_RETURN
  bfffe420 A __kernel_sigreturn
  bfffe440 A __kernel_rt_sigreturn
  c0100000 A _text
  c0100000 T startup_32
  c01000a4 T startup_32_smp
  c0100124 t checkCPUtype
  c01001a5 t is486
  c01001ac t is386
  c0100210 t L6
  c0100212 t check_x87
  c010023a t setup_idt
  c0100257 t rp_sidt
  (...)
```


Outline

Particularité de la programmation noyau

Le noyau GNU Linux est le coeur du système d'exploitation. Travailler directement au coeur 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 avoir un autre de référence enregistré au niveau du BOOT loader connu pour ne pas poser de problème lors du démarrage.

De même il est conseiller, si possible (voir plus loin), de travailler son code sous forme de module. Ce dernier peut alors se charger dynamiquement dans un système stable.

Mise en garde

Les modules s'exécutant dans l'espace noyau le chargement d'un module bugger peut corrompre le système.

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 debugage :

```
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. A 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.

Méthode 2 : User Mode Linux

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 débuguer 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 réseaux.

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 permetant 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 recopie 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 lié 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 un 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 un seconde machine.

Avantages :

- Très pratique pour des développement sur le noyau même.
- Permet de débuguer (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.

Remplacement le port série pour le debugage

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 !).

Outline

API noyau :

ATTENTION

Lorsque vous allez programmer dans le noyau, oubliez toutes les bibliothèques que vous aviez l'habitude d'utiliser et en premier lieu la *libc*.

Heureusement, vous ne serez pas totalement démuni. Le noyau est totalement autonome et implémente lui même une série de fonctionnalités de base.

L'ensemble de ces fonctions disponibles est parfaitement documenté dans la documentation **kernel-api** :

```
linux/Documentation/DocBook/kernel-api.tmpl
```


API noyau : Affichage.

Une des fonctionnalités nécessaire à tout développement est celle de l'affichage. Avec **printk()**, le noyau offre une fonction au fonctionnement quasi identique au classic *printf()*.

Il existe cependant quelques différences :

- Toute chaine de caractères est censée être préfixée par une valeur de priorité. Le fichier *kernel.h* définis 8 niveaux qui vont de **KERN_EMREG** à **KERN_DEBUG**.
- Le flux est récupéré par **klogd**, peut passer dans un *syslogd* et finit généralement dans */var/log/kern.log*.

Exemple

```
printk(KERN_DEBUG "Au retour de f() : i=%i", i);
```


API noyau : Manipulation de la mémoire.

L'allocation de mémoire dans le noyau se fait grâce à la fonction **kmalloc()**. Pendant noyau de la fonction *malloc()*, cette fonction présente des caractéristiques propres :

- Rapide (à moins qu'il ne soit bloqué en attente de pages)
- N'initialise pas la zone allouée
- La zone allouée est contiguë en RAM physique
- Allocation par taille de $2^n - k$ (k : quelques octets de gestion) : Ne demandez pas 1024 quand vous avez besoin de 1000 : vous recevriez 2048 !

Exemple

```
data = kmalloc(sizeof(*data), GFP_KERNEL);
```


Organisation de la mémoire.

Remarque

Il est possible d'étendre l'utilisation de la mémoire au delà des 4 Go Via l'utilisation de la **mémoire haute** (**ZONE_HIGHMEM**).

API noyau : Options du kmalloc.

Les options du *kmalloc* sont définies dans *include/linux/gfp.h* (GFP pour : Get Free Pages) :

- **GFP_KERNEL :** Allocation mémoire standard du noyau. Peut être bloquante. Bien pour la plupart des cas.
- **GFP_ATOMIC :** Allocation de RAM depuis les gestionnaires d'interruption ou le code non lié aux processus utilisateurs. Jamais bloquante.
- **GFP_USER :** Alloue de la mémoire pour les processus utilisateurs. Peut être bloquante. Priorité la plus basse.
- **GFP_NOIO :** Peut être bloquante, mais aucune action sur les E/S ne sera exécutée.
- **GFP_NOFS :** Peut être bloquante, mais aucune opération sur les systèmes de fichier ne sera lancée.
- **GFS_HIGHUSER :** Allocation de pages en mémoire haute en espace utilisateur. Peut être bloquante. Priorité basse.

API noyau : Autres options du kmalloc.

Autres macros définissant des options supplémentaires et pouvant être ajoutées avec l'opérateur | :

- **__GFP_DMA :** Allocation dans la zone DMA
- **__GFP_HIGHMEM :** Allocation en mémoire étendue (x86 et sparc)
- **__GFP_REPEAT :** Demande d'essayer plusieurs fois. Peut se bloquer, mais moins probable.
- **__GFP_NOFAIL :** Ne doit pas échouer. N'abandonne jamais. Attention : à n'utiliser qu'en cas de nécessité !
- **__GFP_NORETRY :** Si l'allocation échoue, n'essaye pas d'obtenir de page libre.

API noyau : Manipulation de la mémoire par page.

Pour l'allocation de grosses tranches mémoire, il existe une série de fonctions plus appropriées que *kmalloc*, puisqu'elles fonctionnent par **page mémoire** :

- **unsigned long get_zeroed_page(int flags) :** Retourne un pointeur vers une page libre et la remplit avec des zéros
- **unsigned long __get_free_page(int flags) :** Identique, mais le contenu n'est pas initialisé
- **unsigned long __get_free_pages(int flags, unsigned long order) :** Retourne un pointeur sur une zone mémoire de plusieurs pages continues en mémoire physique avec *order*= log_2(nombre de pages).

API noyau : Mapper des adresses physiques

La fonction **vmalloc()** peut être utilisé pour obtenir des zones mémoire continues dans l'espace d'adresse virtuel, même si les pages peuvent ne pas être continues en mémoire physique :

```
void *vmalloc(unsigned long size);
void vfree(void *addr);
```

La fonction **ioremap()** ne fait pas d'allocation, mais fait correspondre le segment donné en mémoire physique dans l'espace d'adressage virtuel.

```
void *ioremap(unsigned long phys_addr, unsigned long size
void iounmap(void *address);
```


API noyau : Les *wait queues*.

Les **wait queues** sont une liste de tâches endormies, en attente d'une ressource : lecture d'un *pipe*, attente d'un paquet sur une interface réesaux, etc.

Pour s'enregistrer dans l'une de ces queues, un processus on peut utiliser une des deux fonctions suivantes :

- **sleep_on(queue) :** Le processus s'endort et ne sera réveiller que la ressource sera disponible.
- **interruptible_sleep_on(queue) :** Le processus peut aussi être réveillé par un signal.

Lorsque la ressource est prête, son **handler** reveille tous les processus de la liste avec un appel à la fonction **wake_upqueue**.

API noyau : Les *wait queues*.

Attention

Comme avec les *pthread_cond_wait()*, il est possible que l'un des processus réveillés ne soit activé par le *scheduler* qu'une fois la ressource utilisée par d'autre.

Il faut donc toujours tester la disponibilité de la ressource après chaque sortie d'une *wait queue*.

API noyau : Les *task queues*.

Les **task queues** permettent à un processus de procrastiner une ou plusieurs tâches.

Chaque *task queues* contient une liste chainée de structures contenant un pointeur de fonction (la tâche) et un pointeur de donnée (l'objet de la tache).

A chaque fois qu'une *task queues* est exécutée, toutes les fonctions enregistrées sont exécutées, une à une, avec leurs données en paramètre.

API noyau : Appels systèmes.

Le noyau offre aux applications un ensemble d'appels système (plus de 200) pour réaliser des tâches simple vue de l'application mais complexe vu du noyau.

Si l'on peut utiliser les librairies standard, on peut très bien utiliser ces appels systèmes au sein même d'un code noyau.

Un bon programmeur système n'aura donc pas de mal à coder dans le noyau.

API noyau : Convention de retour

Les fonctions du noyau suivent la même convention de retour que les appels système :

- **positif ou nul** en cas de succès
- **négatif** en cas d'erreur (opposé de la valeur *errno*).

Si la fonction retourne un pointeur, on utilise une autre convention :

- le codes d'erreur est re-encodé par la macro **ERR_PTR()** et est retourné comme un pointeur.
- la fonction appelante utilise la macro **IS_ERR()** pour déterminer s'il s'agit d'un code d'erreur, au quel cas la macro **PTR_ERR()** permet de l'extraire.

API noyau : Convention de retour

Exemple

```
asmlinkage long sys_open(const char* filename) {
  char* tmp;
  int fd, error;

  tmp = getname(filename);
  fd = PTR_ERR(tmp);
  if (! IS_ERR(tmp)) {
    fd = get_unused_fd();
    if(fd >= 0) {
      \* On peut ouvrir le fichier *\
      \* Et retourner le file descriptor *\
    }
  }
  return fd;
}
```


API noyau : Table des symboles.

Le noyau maintient une table des symboles destinée à l'édition de liens dynamiques lors de l'insertion des modules. Ces symboles sont visibles dans **/proc/ksyms**.

Tout symbole qui peut être utilisé dans un module doit être explicitement exporté avec la macro **EXPORT_SYMBOL()**.

Si un module utilise des symboles d'un autre module, il est dit dépendant de ce dernier : commande *depmod*.

API noyau : Types génériques

Pour assurer la portabilité des drivers d'une architecture à une autre, il faut utiliser des types génériques internes au noyau et définis dans *linux/types.h* :

```
u8 : unsigned byte (8 bits)
u16 : unsigned word (16 bits)
u32 :unsigned 32-bit value
u64 : unsigned 64-bit value
s8 : signed byte (8 bits)
s16 : signed word (16 bits)
s32 : signed 32-bit value
s64 : signed 64-bit value
```

Exemple de fonction issue du bus i2c :

```
s32 i2c_smbus_write_byte(struct i2c_client *client, u8 valu
```


API noyau : Types génériques

Pour des variables pouvant être visibles du userspace (ex : *ioctl*) il est nécessaire d'utiliser cette notation :

```
__u8 unsigned byte (8 bits)
__u16 unsigned word (16 bits)
__u32 unsigned 32-bit value
__u64 unsigned 64-bit value
__s8 signed byte (8 bits)
__s16 signed word (16 bits)
__s32 signed 32-bit value
__s64 signed 64-bit value
```

Exemple d'utilisation, lors de l'envoi d'un message de contrôle à un device USB :

```
struct usbdevfs_ctrltransfer {
  __u8 requesttype; __u8 request;
  __u16 value; __u16 index; __u16 length;
  __u32 timeout; /* in milliseconds */
  void *data;
#define USBDEVFS_CONTROL_IOWR('U', 0,
struct
usbdevfs_ctrltransfer)
```


Outline

Concurrence et synchronisation : problèmes.

Le problème des accès concurrents à une ressource critique du noyau peut avoir une cause matérielle et/ou logicielle :

- **Préemption :** Depuis sa version 2.6, Linux est devenu un noyau préemptif. Un processus peut être interrompu au milieu d'un code noyau et laissé sa place à un autre processus.
- **Multi-processeurs :** Avec l'arrivée des machines multi-processeurs, se pause le problème de l'exécution parallèle de code noyau.

Concurrence et synchronisation : Les Solutions.

Plusieurs solutions sont envisageables pour résoudre le problème des accès concurrents :

1. Bloquer les interruptions
2. Opérations atomiques
3. Big Kernel Lock
4. Sémaphores
5. Spinlocks

Concurrence et synchronisation : Ininterruptible.

Dans une architecture mono-processeur, le problème résulte uniquement de la préemption. On peut donc le résoudre en désactivant les interruptions : le code devient alors **non-préemptible**.

Pour ce faire, on peut utiliser les macro *local_irq_disable()* et *local_irq_enable()* qui utilisent l'instruction assembleur *cli* (resp. *sti*) pour désactiver (resp. activer) les interruptions **sur le processeur local**.

Exemple

```
local_irq_disable();
/* code non préemptible ... */
local_irq_enable();
```


Concurrence et synchronisation : Opérations atomiques

Il possible d'éviter le problème d'accès concurrent en utilisant des fonctions garantissant l'atomicité d'une opération.

Ces fonctions sont dépendantes de l'architecture matérielle et sont définies dans **linux/include/asm/atomic.h**.

Exemple

Si *p* est un pointeur d'entier on peut utiliser : *atomic_inc(p)*, *atomic_set(p,i)* et *atomic_add(p)*.

Concurrence et synchronisation : BKL

Dans un système multiprocesseur on ne peut résoudre le problème en bloquant les interruptions. Une solution consiste alors à verrouiller l'ensemble du noyau avec un **Big Kernel Lock (BKL)**.

Avantage : C'est la solution la plus simple.

Inconvénient : C'est extrêmement couteux car on bloque l'ensemble des processeurs.

Exemple

```
lock_kernel();
/* critical region ... */
unlock_kernel();
```


Concurrence et synchronisation : BKL

Fin annoncée du BKL

Suite à une longue discussion sur la liste diffusion du noyau, il a été décidé par Linus Torvalds de supprimer progressivement le Big Kernel Lock.

Le travail va se dérouler dans la branche "kill-the-BKL" mais il sera sans doute possible aux utilisateurs du noyau principal d'activer une nouvelle option (CONFIG_DEBUG_BKL) afin de participer eux aussi à la chasse aux bugs.

Concurrence et synchronisation : Sémaphores

Les **sémaphores** permettent de réaliser une synchronisation entre les processus. Cette méthode pause tout de même le problème de l'attente passive qui peut conduire à un changement de contexte.

Exemple

```
struct semaphore mr_sem;
sema_init(&mr_sem, 1);
/* usage count is 1 */
if (down_interruptible(&mr_sem))
/* semaphore not acquired; received a signal ... */
/* critical region (semaphore acquired) ... */
up(&mr_sem);
```

Comme le processus s'endort pour attendre le sémaphore, cette solution est réservée au code noyau s'exécutant en *contexte utilisateur*.

Concurrence et synchronisation : Spinlocks

Les **spinlocks** sont une implémentation de sémaphore avec attente active. Leur manipulation est plus complexe que celle de sémaphores "passifs".

```
spinlock_t mon_lock = SPIN_LOCK_UNLOCKED;
unsigned long flags;
spin_lock_irqsave(&mon_lock, flags);
/* critical section ... */
spin_unlock_irqrestore(&mon_lock, flags);
```

Les spinlocks ne produisent aucun code si le noyau est compilé mode non-préemptible et sans support SMP.

Outline

Module or not module : Opter pour le module.

Lorsque l'on développe une fonctionnalité pour le noyau, on doit choisir entre :

- intégrer son **code dans le noyau** au travers d'un patch. Elle sera alors intégrer statiquement au noyau.
- créer un nouveau **module** que l'on pourra charger dynamiquement dans le noyau.

Règle de choix

Toujours choisir la solution du module si 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.

Un module Linux c'est quoi ?

Définition

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.

Un module minimal comme le nôtre contient au moins les fichiers d'en-tête suivants :

```
#include <linux/module.h>   /* API des modules */
#include <linux/kernel.h>   /* Si besoin : KERN_INFO dans
#include <linux/init.h>     /* Si besoin : fonction d'init
```

Un module peut enregistrer des fonctions à exécuter lors de son chargement et de son déchargement :

```
module_init(pointeur_fonction_init);
module_exit(pointeur_fonction_exit);
```


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:
```


Licence de distribution du module

Depuis le noyau 2.6, la définition de la licence est nécessaire.

Dans le cas contraire, on obtient un message d'erreur dans les traces du noyau :

```
module license 'unspecified' taints kernel.
```


Exemple de module : helloworld.c

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

static int hello_init(void) {
  printk(KERN_ALERT "Hello, world\n");
  return 0;
}

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

module_init(hello_init);

module_exit(hello_exit);

MODULE_LICENSE("GPL");
```


Compiler un module

Le Makefile ci-dessous est réutilisable pour tout module Linux 2.6. Lancez juste make pour construire le fichier hello.ko

```
# Makefile pour le module hello
obj-m := hello.o
KDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)
default:
  $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules
```

Il est à noter que les modules sont seulement compilés et pas linkés. Le linkage s'effectuant lors du chargement du driver dans le noyau Linux.

Dans les linux 2.4, l'extension des modules était **.o**. Désormais, avec la famille des 2.6, c'est **.ko** pour *kernel object*.

Chargement et déchargement d'un module

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

```
insmod helloworld
```

Résultat au chargement :

```
Hello, world
```

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

```
rmmod helloworld
```

Résultat au déchargement :

```
Goodbye, cruel world
```


Exemple de module avec paramètre : hello_parma.c

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/moduleparam.h>
MODULE_LICENSE("GPL");
static char *whom = "world";
module_param(whom, charp, 0);
static int howmany = 1;
module_param(howmany, int, 0);
static int hello_init(void) {
  int i;
  for (i = 0; i < howmany; i++)
    printk(KERN_ALERT "(%d) Hello, %s\n", i, whom);
  return 0;
}
static void hello_exit(void) {
  printk(KERN_ALERT "Goodbye, cruel %s\n", whom);
}
module_init(hello_init);
module_exit(hello_exit);
```

Passer des paramètres aux modules

Il existe 3 façons de passer des paramètres à un module :

- Avec insmod ou modprobe :
  `insmod ./hello_param.ko howmany=2 whom=universe`
- Avec modprobe en changeant le fichier /etc/modprobe.conf :
  `options hello_param howmany=2 whom=universe`
- Avec la ligne de commande du noyau, lorsque le module est lié statiquement au noyau :
  `options hello_param.howmany=2 hello_param.whom=universe`

Dépendances de modules

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, grâce aux symboles exportés par le module :
A **dépend de** B si A utilise un symbole exporté par B.

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

Ce fichier est mis à jour (en tant que root) avec **depmod** :

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

La commande **modprobe** permet de charger un module avec toutes ses dépendances.

Exemple de module avec dépendance

Dans l'exemple helloword.c, on remplace *printk()* par *my_printk()*

Puis implémente cette fonction dans un autre module *my_printk*.
Notons que ce module n'a pas de fonction *init* :

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

MODULE_LICENSE("GPL");
void my_printk (char *s)
{
   printk (KERN_INFO "my_printk: %s\n", s);
}
EXPORT_SYMBOL_GPL(my_printk);
```


Exemple de module avec dépendance

Lorsque les deux modules sont compilés, on doit tout d'abord insérer le module définissant la fonction afin d'éviter une erreur de dépendance.

```
insmod helloworld2.ko
    insmod: error inserting 'helloworld2.ko': -1 Unknown symbol
inmodule
insmod ../my_printk/my_printk.ko
insmod helloworld2.ko
```

Autre solution, utiliser modprobe après instalation des modules :

```
modprobe -v helloworld2
    insmod /lib/modules/version_noyau/extra/my_printk.ko
    insmod /lib/modules/version_noyau/extra/helloworld2.ko
```