



Cours de système

Modules

Sébastien Paumier



Définition

- module=code noyau pouvant être chargé et déchargé dynamiquement
- intérêts:
 - pouvoir ajouter des fonctionnalités sans toucher au code du noyau, pour éviter les risques d'erreurs
 - ne pas avoir à rebooter
- utilisation principale: les drivers



Installer un module

- les modules du système sont dans `/lib/modules/xxx/`
- `insmod` permet d'installer un seul module, sans vérifier ses dépendances
- `modprobe` installe un module en satisfaisant d'abord toutes les dépendances, décrites dans `/lib/modules/xxx/modules.dep`
- `rmmmod` désinstalle un module



Installer un module

- **depmod -a** examine récursivement tous les modules trouvés dans **/lib/modules/xxx** pour créer le fichier de dépendances
- **modinfo** donne des informations sur un module:

```
$>modinfo video
filename:      /lib/modules/2.6.31-22-generic/kernel/drivers/...
license:      GPL
description:   ACPI Video Driver
author:       Bruno Ducrot
srcversion:    870042C8E8178B98F3C5391
...
```



Versions

- puisqu'il s'agit de code noyau, un module est compilé pour un noyau précis
- il est fort probable qu'on ne puisse pas le chargé avec un noyau différent
- quand on installe un nouveau module, il vaut mieux le recompiler pour être sûr qu'il n'y aura pas de problème
- pour compiler un module, il faut avoir recompilé son noyau à partir des sources complètes



Compiler le noyau

- chaque distribution a ses propres patches sur le noyau
- étape 1: télécharger les sources correspondant au noyau que vous utilisez actuellement, obtenu avec `uname -r`
- sous Ubuntu, on obtient les sources du noyau courant avec la commande:

```
$>sudo apt-get source linux-image-`uname -r`
```



Compiler le noyau

- étape 2: copier le fichier de configuration du noyau courant trouvé dans `/boot` en un fichier `.config`
- étape 3: compiler le noyau

```
$>cd linux-2.6.31  
$>sudo bash  
#>cp /boot/config-2.6.31-22-generic .config  
#>make
```



init/exit

- un module possède une fonction d'initialisation et de libération, invoquées automatiquement au chargement et déchargement du module:
 - `int init_module(void)`
 - `void cleanup_module(void)`
- mais on peut personnaliser ces noms grâce aux macros `module_init` et `module_exit`



Affichage

- le noyau n'écrit pas sur un terminal, car tout terminal appartient à un processus utilisateur
- `printk` permet d'émettre des messages dans `/var/log/kern.log`
- on peut préfixe l'affichage par une constante indiquant le niveau de log (cf. `linux/kernel.h`)
- exemple: `printk(KERN_INFO "info\n");`



Informations

- grâce aux macros de `linux/module.h` on définit l'auteur, la description et la licence du module
- attention aux problèmes légaux liés aux licences!
- un premier module basique:

`module.c`



Compiler un module

- on utilise un Makefile très particulier:

```
LINUX_SRC_DIR=.....
```

```
EXTRA_CFLAGS += -I$(LINUX_SRC_DIR)/include
```

```
obj-m := module.o
```

```
all:
```

```
    make -C $(LINUX_SRC_DIR) M=`pwd` modules
```

```
clean:
```

```
    make -C $(LINUX_SRC_DIR) M=`pwd` clean
```

linux/Documentation/kbuild/makefiles.txt



Compiler un module

- cela crée un fichier .ko, prêt à être chargé:

```
$>make
```

```
make -C /home/paumier/tmp/linux-2.6.31 M=`pwd`
```

```
modules
```

```
make[1]: entrant dans le répertoire «  
/home/paumier/tmp/linux-2.6.31 »
```

```
CC [M] /home/.../module.o
```

```
Building modules, stage 2.
```

```
MODPOST 1 modules
```

```
CC /home/.../module.mod.o
```

```
LD [M] /home/.../module.ko
```

```
make[1]: quittant le répertoire «  
/home/paumier/tmp/linux-2.6.31 »
```



Utiliser le module

- comme on n'a qu'un seul module sans dépendance, on peut le charger simplement avec:

```
sudo insmod module.ko
```

- on peut alors le voir avec **lsmod** et le décharger avec:

```
sudo rmmod module
```

- on peut consulter ses messages avec **tail /var/log/kern.log**



Paramètres

- la macro `module_param` permet de définir des paramètres pouvant être passés lors du chargement du module:
 - `module_param(foo, type, perm);`
- `foo`=nom de la variable globale qui va recevoir la valeur du paramètre
- les types possibles sont: `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `charp` (chaîne de caractères), `bool`, `invbool`



Paramètres

- pour un paramètre **charp**, le noyau va automatiquement allouer de la mémoire qu'il faudra rendre avec **kfree** au déchargement du module
- si **perm** est non nul, un fichier est créé dans **/sys/module/nom/parameters** et on peut accéder ainsi à la valeur du paramètre par le système de fichiers

```
include/linux/stat.h
```



Paramètres

- on peut aussi déclarer des tableaux avec `module_param_array`
- on décrit les paramètres avec `MODULE_PARM_DESC(nom, description);`
- on passe les valeurs des paramètres grâce à `insmod`:

`module_params.c`

```
$>sudo insmod module_params.ko name=coucou  
$>tail /var/log/kern.log  
... [95208.181578] This is a kernel hello world!  
... [95208.181586] value=5 name=coucou
```



Drivers

- la majorité des modules servent à écrire des pilotes de périphériques
- communication avec un périphérique via un fichier spécial dans `/dev`
- périphériques en mode bloc:
 - buffers de taille fixe
 - possibilité de ne pas traiter les requêtes dans l'ordre (ex: disque dur qui charge les blocs dans l'ordre qui arrange le bras de lecture)



Drivers

- périphériques en mode caractère:
 - buffers de taille variable
 - traitement séquentiel des requêtes
- un périphérique est caractérisé par deux nombres:
 - nombre majeur: indique quel driver utiliser
 - nombre mineur: identité du périphérique, au cas où le driver en gère plusieurs



Device files

- on crée un fichier de périphérique avec la commande **mknod**
- exemple:
 - **mknod -m 0666 /dev/pouet c 16 64**
- crée avec les droits **rw-rw-rw-** un fichier **/dev/pouet** en mode caractère
- pour y accéder, le noyau utilisera le driver associé au numéro majeur 16



Drivers

- un driver doit s'enregistrer avec un numéro majeur sur un fichier de périphérique
- `int register_chrdev(unsigned int major, const char* name, const struct file_operations* ops);`
- `ops` est une structure désignant la liste des fonctions définies pour ce driver

```
include/linux/fs.h
```



Drivers

- avec un numéro majeur nul, le noyau en choisira automatiquement un disponible
- il suffira ensuite de créer un fichier de périphérique avec ce numéro
- exemple: driver qui compte combien d'octets sont écrits et combien d'appels à **open** et **write** ont été effectués



open/release

- les fonctions **open** et **release** sont invoquées quand un processus veut accéder au fichier de périphérique
- problème: le module ne doit pas être déchargé pendant qu'un processus utilise le driver
- solution: indiquer dans **open** que le driver est utilisé avec
try_module_get(THIS_MODULE);



open/release

- attention: il faut absolument faire l'opération inverse `module_put(THIS_MODULE);` dans `release`, sinon on aura un module immortel que seul un reboot pourra enlever!



Espace utilisateur

- notre fonction **read** va retourner à l'utilisateur une chaîne de caractères donnant des statistiques sur l'utilisation du fichier de périphérique
- problème: les données du noyau et celles du processus utilisateur ne sont pas dans les mêmes espaces d'adressages
- solution: utiliser **copy_to_user**

char_device.c



Dans le noyau

- avant d'invoquer notre **open**, le noyau fait ses tests de contrôle d'accès au fichier de périphérique
- comme on est dans le noyau, on a accès à tout et on peut faire ce qu'on veut
- exemple: fichier qui ne pourra pas être ouvert deux fois de suite par le même utilisateur effectif, même pas root

`permissions.c`



Paramétrage

- avec **ioctl**, on peut changer dynamiquement les paramètres d'un périphérique précis, et non pas ceux du driver tout entier
- il faut définir des commandes représentées par des entiers
- exemple: interdire l'accès à un utilisateur précis

`permissions2.c`

`my_ioctl.c`



Opérations bloquantes

- quand le noyau doit faire une opération bloquante, il doit endormir le processus utilisateur jusqu'à ce que l'opération soit possible
- pour cela, on doit placer le processus dans une file d'attente avec une macro:
 - `wait_event(queue, condition)`
- et le réveiller avec `wake_up(&queue)` quand la condition doit être réévaluée



wait_event

- cette macro ne réveille pas le processus en cas de signal, ce qui peut être gênant, par exemple en empêchant l'utilisateur de tuer le processus avec un SIGINT
- pour être aussi réveillé en cas de signal, utiliser `wait_event_interruptible`
- si on veut aussi un timeout, on peut utiliser `wait_event_timeout` et `wait_event_interruptible_timeout` avec une durée en jiffies



wait_event

- quand il y a un appel à **wake_up**, tous les processus en attente d'une condition sont réveillés
- ils doivent tester s'ils ont été réveillé à cause d'un signal
- si c'est le cas, et si ce n'est pas un signal que le processus voulait bloquer, on doit rendre la main au processus utilisateur en retournant `-ERESTARTSYS`



Réveil

- **IMPORTANT:** quand on réveille les processus en attente avec `wake_up`, on doit impérativement redonner la main à l'ordonnanceur avec `schedule`
- sinon, le processus courant peut changer aussitôt la condition avant que les autres aient eu le temps de la tester, ce qui peut les bloquer indéfiniment s'ils ne sont pas ordonnancés au bon moment



wait_event

- si on a été réveillé par un signal bloqué, on doit se réendormir
- on place donc le code de test de réveil dans une boucle:

```
while (condition) {
    int i, is_signal=0;
    wait_event_interruptible(queue, !condition);
    for (i=0; i<_NSIG_WORDS && !is_signal; i++) {
        is_signal=current->pending.signal.sig[i] &
            ~current->blocked.sig[i];
    }
    if (is_signal) return -EINTR;
}
```



Mode non bloquant

- si le processus utilisateur a émis le souhait de travailler en mode non bloquant, on doit respecter son choix et rendre tout de suite la main avec `-EAGAIN` en cas de blocage

```
if (condition) {
    if (filp->f_flags & O_NONBLOCK) {
        /* Non blocking operation ? We return */
        return -EAGAIN;
    }
    ...
}
```



Écriture bloquante

- exemple: driver qui bloque en écriture si un autre processus est déjà en train d'écrire dans le fichier de périphérique
- on simule une longue écriture en attendant avec
`wait_event_interruptible_timeout`

`blocking.c`

`morse_1.c`



Concurrence

- hélas, on n'est jamais à l'abri d'une architecture SMP et il faut protéger tout accès à des données globales
- solutions:
 - les variables atomiques `atomic_t`,
 - les spinlocks `spinlock_t`
 - les sémaphores

`morse_2.c`

```
arch/x86/include/asm/atomic.h  
include/linux/spinlock.h  
include/linux/semaphore.h
```



Allmighty modules

- en tant que code noyau, on a accès à tout ce qui est visible dans le noyau
- exemple: jouer avec les diodes du clavier
- on peut rechercher le driver de clavier associé à la console courante (celle qui a la main sur le clavier) et utiliser son `ioctl` pour changer l'état des diodes



Allmighty modules

- problème: il faut avoir la capacité `CAP_SYS_TTY_CONFIG` pour pouvoir faire ça, ce qui nous oblige à être root, sauf si...
- un code noyau peut tout faire, y compris s'attribuer les pleins pouvoirs quand il le souhaite
- d'où, le bouquet final:

`morse_3.c`



Pour aller plus loin:
<http://www.makelinux.net/ldd3/>

The end