Programmation noyau sous Linux Partie 1 : API des modules Linux

Cette série d'articles a pour but de présenter les techniques de programmation dans l'espace du noyau Linux. Ce type de programmation requiert des connaissances particulières et il existe somme toute assez peu de documentation francophone et synthétique sur le sujet. Le sujet a déjà été traité partiellement lors de diverses publications dans GNU/Linux Magazine, mais il s'agit aujourd'hui de dispenser une information la plus exhaustive et à jour possible.

Nous aborderons des sujets comme :

- l'interface de programmation (API) des modules Linux, les contraintes techniques et légales associées ;
- la mise au point des programmes en espace noyau (KGDB).
- les différents types de pilotes de périphériques (caractère, bloc, réseau) ;
- l'utilisation des threads du noyau ;
- la gestion des interruptions ;
- la gestion du DMA et de **mmap()** ;
- les pilotes PCI, USB et Vidéo For Linux, version 2 (V4L2) ;

A la fin de la série, nous aborderons également la programmation de tâches temps réel " dur " en utilisant les principales extensions disponibles, soit XENOMAI, RTAI et RTLinux, bien que ce dernier ne fasse plus réellement partie de la nébuleuse du Logiciel libre.
A l'issue de ces articles, le lecteur attentif et motivé devrait être capable de développer des pilotes de périphériques simples pour Linux et suivre ainsi la célèbre citation de Linus Torvalds :
" We're back to the times when men were men and wrote their own device drivers. "

Résumé de l'article

Le présent article introduit l'interface de programmation des modules Linux (ou Application Programming Interface, API). Cette API est systématiquement utilisée pour la programmation dans l'espace du noyau, tant pour la programmation des pilotes de périphériques que de certaines extensions temps réel dur. Nous aborderons aujourd'hui les aspects techniques de cette API :

- les espaces mémoire de Linux ;
- les contraintes légales (GPL/LGPL) ;
- un exemple de module **Hello World** et son fichier Makefile associé ;
- l'installation des modules ;
- la dépendance entre les modules ;
- l'identification des modules ;
- le passage de paramètres ;
- la programmation d'un compteur régulier ;
- l'accès au répertoire **/proc**.

Nous insisterons sur les contraintes légales liées à la GPL, ce point étant fondamental bien que souvent mal connu des développeurs.

Les espaces mémoire de Linux

Les versions normales du noyau Linux utilisent deux espaces de mémoire :

- L'espace dit " utilisateur " (ou user space). Cet espace mémoire correspond à l'espace d'exécution des programmes applicatifs.
- L'espace dit " noyau " (ou kernel space). Cet espace correspond à l'espace d'exécution du noyau Linux et de ses extensions (modules).

L'architecture de Linux est décrite par le schéma ci-dessous (merci à Wikipedia).

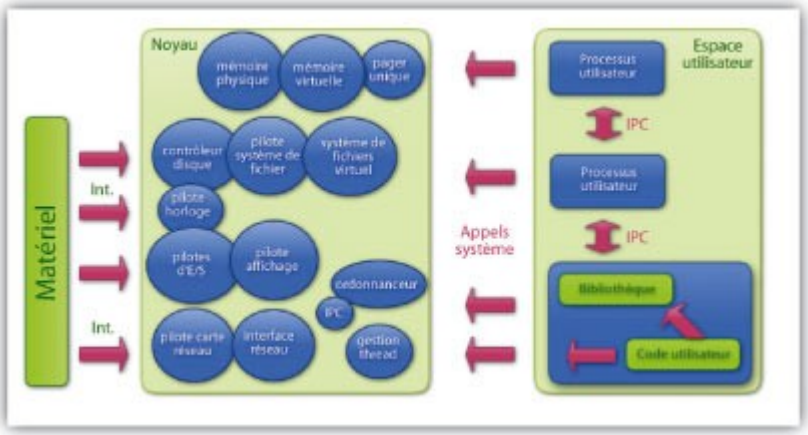

Figure 1 : Architecture de Linux

La plupart des développeurs travaillent dans l'espace utilisateur dans lequel résident la majorité des applications. Depuis une application, on peut cependant effectuer une requête au noyau en effectuant un appel système. Un cas simple est l'obtention du numéro d'identification du processus associé au programme en cours d'exécution (PID). Dans un programme classique, on l'obtient par le code C suivant :

```
/* Obtention du PID */
my_pid = getpid();
printf ("Mon PID est %d\n", my_pid);
```

Bien entendu, l'accès à l'espace mémoire du noyau ne se limite pas à cela. La plupart du temps, les applications doivent accéder à des périphériques et donc à la mémoire. Si l'on respecte le principe des deux espaces séparés, l'accès à la mémoire ne se fait jamais directement, mais le plus souvent en passant par un composant dédié résidant dans l'espace du noyau. C'est une définition simple du pilote de périphérique. Le comportement normal du noyau Linux se base également sur la MMU (pour Memory Management Unit) un composant matériel inclus dans le processeur et qui permet la séparation physique de la mémoire. Telle est d'ailleurs l'origine du noyau Linux, puisque les premiers travaux de Linus Torvalds consistaient à utiliser le mode dit " protégé " (noyau) du processeur Intel 386, premier processeur de la gamme x86 à disposer d'une MMU.
Corollaire : dans le cas d'un environnement Linux sur un processeur disposant d'une MMU, les espaces mémoire sont physiquement séparés et il n'est donc pas possible qu'un programme utilisateur " plante " directement le système. Ce n'est bien sûr pas le cas d'un module Linux. La programmation et la mise au point des programmes en espace noyau est d'autant plus délicate. Un noyau Linux adapté peut fonctionner sur des processeurs sans MMU (on parle alors de µCLinux). L'API de programmation est similaire, mais il y a alors un seul espace de mémoire physique (plus de mode protégé).
Il est cependant possible d'accéder à un périphérique sans passer par un pilote en utilisant la

fonction **ioperm()**. Cette fonction permet de définir une zone de mémoire sur laquelle un programme utilisateur aura un accès direct en utilisant des fonctions d'entrée/sortie comme **inb()** et **outb()**. Elle est assez fréquemment utilisée en x86, mais elle n'est pas forcément disponible sur toutes les architectures.

Les contraintes légales

Certains lecteurs pourront s'étonner de la présence et de la taille d'un tel paragraphe dans un article technique. La raison est simple : Linux est de plus en plus utilisé à des fins industrielles et le respect des licences est fondamental dans ce cas puisqu'un problème légal peut avoir de lourdes conséquences. Par expérience, nous savons que de nombreux utilisateurs sont mal informés sur ce sujet et nous allons tenter en quelques lignes de dégager les points importants.
Le système Linux a une particularité : il y a dans une distribution Linux une grande quantité de code source provenant de projets libres externes (GNU, X, KDE, etc.) donc non spécifique à Linux. C'est la raison pour laquelle les puristes parlent de système " GNU-Linux " et non de système " Linux ". Au niveau des licences, la conséquence est la cohabitation de plusieurs licences libres.

- GNU GPL (General Public License) ;
- GNU LGPL (Lesser GPL) ;
- BSD (Licence Berkeley) ;
- X (Licence X Window/X.org) ;
- MPL (Mozilla Public License) ;
- etc.

Si l'on se place du point de vue du développeur, la GPL est plus " contraignante " que la plupart des autres licences libres, car elle est stricte sur deux points importants.

- 1. La notion de " travail dérivé ". Un code source dérivé – au sens large – d'un code sous GPL doit rester sous GPL.
- 2. L'édition de lien entre du code GPL et du code propriétaire n'est pas autorisée sauf dans certains cas très précis (voir la FAQ de la licence GPL). En particulier, et contrairement à certaines croyances, le fait d'effectuer une édition de lien dynamique (utilisation de bibliothèques partagées) n'est pas suffisante pour s'affranchir de la GPL.

C'est la raison pour laquelle de nombreuses bibliothèques importantes sont diffusées sous Lesser-GPL (LGPL) et non sous GPL. Dans le cas de la LGPL, la contrainte de l'édition de lien disparaît. Le cas le plus flagrant est la GNU-Libc (glibc) utilisée par tous les programmes de l'espace utilisateur (sauf ceux qui sont compilés en mode statique). Pour cette raison, la GNU-Libc est diffusée sous LGPL.
En résumé, il est relativement aisé de développer du code propriétaire dans l'espace utilisateur de Linux.
Le cas de l'espace noyau est très différent. Si l'on reprend le début du paragraphe précédent, on note que cet espace est réservé au noyau Linux et à ses extensions (les modules). Le noyau Linux étant diffusé sous GPL, on en déduit que seule la GPL est théoriquement utilisable dans cet espace. Si l'on reprend les propos de Linus Torvalds sur ce sujet, les choses sont claires comme l'indique un courrier de Linus datant du 4 décembre 2003 et posté sur la liste LKML (Linux Kernel Mailing List) :
It is very clear: a kernel module is a derived work of the kernel by default. End of story.
You can then try to prove (through development history, etc.) that there would be major reasons why it's not really derived. But your argument seems to be that nothing is derived, which is clearly totally false, as you yourself admit when you replace "kernel" with "Harry Potter".
Linus

Un autre courrier posté le même jour est tout aussi clair :
So you can run the kernel and create non-GPL'd programs while running it to your hearts content. You can use it to control a nuclear submarine, and that's totally outside the scope of the license (but if you do, please
note that the license does not imply any kind of warranty or similar).

BUT YOU CAN NOT USE THE KERNEL HEADER FILES TO CREATE NON-GPL'D BINARIES.
Comprende?
Linus
Bien évidemment, il existe des modules Linux diffusés sous forme binaire, mais, en toute rigueur, ils ne devraient pas exister. Si l'on se place du coté des fabricants de matériel, il est cependant clair que la GPL est une contrainte forte, puisqu'il n'est théoriquement pas possible de placer des informations non diffusables (relevant de la propriété intellectuelle) dans l'espace du noyau. C'est la raison pour laquelle il existe une certaine tolérance vis-à-vis de ces écarts et qu'il est techniquement possible d'utiliser un module binaire.
Nous verrons plus loin que le module binaire est intimement lié à la version du noyau. De ce fait, un module binaire pourra être utilisé uniquement dans l'environnement de noyau ayant servi à sa génération, ce qui représente une contrainte non négligeable.
Lorsque l'on interroge Linus sur ce sujet, il tolère cet état de fait à partir du moment ou il ne s'agit pas de travail dérivé (voir l'interview par Alessandro Rubini datant de septembre 1998).
Alessandro : What is your position about the availability of Linux modules in binary-only form ?
Linus : I kind of accept them, but I never support them and I don't like them.
The reason I accept binary-only modules at all is that in many cases you have for example a device driver that is not written for Linux at all, but for example works on SCO Unix or other operating systems, and the manufacturer suddenly wakes up and notices that Linux has a larger audience than the other groups. And as a result he wants to port that driver to **Linux.**
En résumé, les choses sont assez claires.

- 1. Dans l'espace du noyau, SEULE LA GPL est théoriquement utilisable. L'utilisation d'une autre licence et, a fortiori, la diffusion de modules binaires relèvent de la tolérance.
- 2. La diffusion de modules binaires est très contraignante pour l'utilisateur final, puisque le module fonctionne uniquement pour la version de noyau prévue lors de la compilation du module.
- 3. Le fait que le module soit chargé dynamiquement ne change rien au problème de licence.
- 4. Dans le cas de problème de propriété intellectuelle, il est conseillé – si possible – d'isoler les portions de code concernées dans l'espace utilisateur. Les portions de code dans l'espace noyau devront alors être suffisamment génériques pour être diffusées sous GPL. En gros, cela correspond à réduire le niveau de complexité du module quitte à reporter la complexité dans l'espace utilisateur.

Présentation de l'API des modules

La suite du document décrit l'API des modules Linux pour le noyau 2.6. Comme à l'accoutumée, nous nous baserons sur un exemple simple qui évoluera au fur et à mesure de la démonstration.

Historique et introduction

Les modules Linux furent introduits en 1995 lors de la sortie de la version 1.2 du noyau. Nous rappelons qu'il existe deux manières de valider le support d'un périphérique dans le noyau Linux.

- 1. Compiler ce pilote dans la partie statique du noyau (le fichier **bzImage** ou **vmlinuz**)
- 2. Compiler ce pilote en tant que module. Ce dernier sera chargé ultérieurement si nécessaire. Nous verrons plus tard que le chargement peut s'effectuer manuellement ou bien de manière automatique.

Si nous devons ajouter le support d'un périphérique inconnu du noyau, il est évident que la première solution n'est pas envisageable, puisqu'elle impliquerait de modifier le code source du noyau. De ce fait, nous traiterons uniquement le cas des modules, car dans un système Linux moderne (soit depuis 1995), un pilote est forcément un module (mais un module n'est pas forcément un pilote). On peut faire une analogie entre un module et une bibliothèque dynamique qui serait chargée dans l'espace utilisateur par la primitive **dlopen()**. Le dé-chargement du module est similaire à l'utilisation de la primitive **dlclose()** dans l'espace utilisateur.

```
Exemple du module " Hello World "
```

Afin de simplifier le problème, nous allons partir d'un exemple de module simple (le traditionnel Hello World). Malgré sa simplicité, la mise en place de ce module nous permettra de découvrir les concepts fondamentaux de la programmation dans l'espace noyau. Pour tester le module présenté, il faut bien entendu disposer de la chaîne de compilation GNU, mais également des sources du noyau 2.6 livrés avec la distribution ou bien chargés depuis http://www.kernel.org (vanilla kernel).
La référence de l'API utilisée sera celle du noyau 2.6.

- structure du code source d'un module ;
- fonctions **module_init()** et **module_exit()** ;
- utilisation de **printk()** et interface avec le système de trace du système ;
- fichier Makefile associé ;
- macros d'identification du module ;
- test de chargement/déchargement ;
- installation dans l'arborescence du noyau.

Le fonctionnement du module en question est trivial : au chargement, il affiche le message Hello World dans le système de trace. Au déchargement, il affiche le message Goodbye cruel world.
Un module minimal comme le nôtre contient au moins les fichiers d'en-tête suivants. Le fichier linux/kernel.h est nécessaire dès lors que l'on utilise la fonction **printk()** qui est l'homologue de **printf()** en espace noyau. L'accès aux fichiers d'en-tête est relatif au répertoire des sources du noyau, soit en général **/usr/src/linux-2.6.x**.

```
#include <linux/module.h>  /* API des modules */
#include <linux/kernel.h>  /* Pour KERN_INFO dans printk() */
```

Le paramètre **KERN_INFO** indique le niveau d'erreur associé au message. Dans notre cas, il s'agit d'un simple message de trace. Les valeurs sont définies dans **linux/kernel.h**.

```
#define    KERN_EMERG     "<0>"     /* system is unusable            */
#define    KERN_ALERT     "<1>"     /* action must be taken immediately  */
#define    KERN_CRIT     "<2>"     /* critical conditions        */
#define    KERN_ERR     "<3>"     /* error conditions            */
#define    KERN_WARNING  "<4>"    /* warning conditions             */
#define    KERN_NOTICE "<5>"    /* normal but significant condition  */
#define    KERN_INFO     “<6>”    /* informational            */
```

```
#define    KERN_DEBUG     "<7>"    /* debug-level messages         */
```

Nous définissons la fonction d'initialisation appelée lors du chargement du module. Dans notre cas, la fonction se limite à l'affichage d'un message.

```
static int __init hello_init(void)
{
    printk(KERN_INFO "Hello world!\n");
    return 0;
}
```

Le principe est le même pour la fonction liée au déchargement.

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

Puis nous affectons les pointeurs de fonctions grâce aux deux macros définies dans **linux/init.h**.

```
module_init(hello_init);
module_exit(hello_exit);
```

La génération du module nécessite des options de compilation spécifiques. Depuis l'apparition du noyau 2.6, la structure d'un fichier Makefile de module est simplifiée, car elle utilise l'arborescence des sources du noyau.
On définit tout d'abord le chemin d'accès au répertoire pointant vers les sources du noyau. Le module est compilé pour une version de noyau donnée. En théorie, il ne peut pas fonctionner sur une autre version. Le test de compatibilité s'effectue sur la version du noyau, soit par exemple 2.6.12-14mdk, obtenu grâce à la commande **uname -r**. Au niveau du Makefile, on peut calculer dynamiquement la valeur du chemin d'accès.

```
KDIR= /lib/modules/$(shell uname -r)/build
```

On définit ensuite le nom du fichier objet produit de la compilation.

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

La cible **all** effectue la compilation du module, puis l'édition de liens afin d'obtenir un ficher **.ko** (pour Kernel Object).

```
all:
        $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules
```

La cible **install** permet d'installer le module dans l'arborescence du noyau (sous-répertoire de **/lib/modules**).

```
install:
	$(MAKE) -C $(KDIR) M=$(PWD) modules_install
```

La cible **clean** efface tous les fichiers binaires.

```
clean:
	rm -f *~
	$(MAKE) -C $(KDIR) M=$(PWD) clean
```

La compilation s'effectue comme en utilisant la commande **make**.

```
$ make
make -C /lib/modules/2.6.12-14mdk/build SUBDIRS=/path/hello_world modules
make[1]: Entering directory `/usr/src/linux-2.6.12-14mdk'
  CC [M]  /path/hello_world/helloworld.o
  Building modules, stage 2.
  MODPOST
  CC      /path/hello_world/helloworld.mod.o
  LD [M]  /path/hello_world/helloworld.ko
make[1]: Leaving directory `/usr/src/linux-2.6.12-14mdk'
```

Notez que le système de compilation n'affiche pas les détails. Si l'on désire changer ce comportement, il suffit de passer l'option **V=1** soit :

```
$ make V=1
```

On peut d'ores et déjà tester le fonctionnement du module en utilisant la commande **insmod**, extraite du paquetage **module-init-tools**. Notez que seul l'utilisateur root peut charger et décharger des modules.

```
# insmod helloworld.ko
```

On peut vérifier la présence du module avec la commande **lsmod**. La sortie indique la taille mémoire occupée par le module.
La structure des modules étant arborescente, on affiche également la liste des modules utilisant le module en question.

```
# lsmod
Module                  Size  Used by
...
helloworld              992   0
```

Le message provenant du module est affiché dans le tampon tournant du noyau (ou ring buffer) à l'aide de la commande **dmesg**. Il est également traité par le démon **syslogd** en fonction de la configuration définie dans le fichier **/etc/syslog.conf**. Dans le cas de notre système, les messages sont visibles sur **/var/log/kernel/info** et **/var/log/messages**.

```
$ dmesg
...
Hello world!
# tail /var/log/kernel/info
...
Aug 30 15:24:53 localhost kernel: Hello world!
# tail /var/log/messages
...
Aug 30 15:24:53 localhost kernel: Hello world!
```

On peut décharger le module en utilisant la commande **rmmod**. On note que l'on utilise le nom du module et non plus le nom du fichier.

```
# rmmod helloworld
```

A terme, il est nécessaire d'installer le module dans l'arborescence du noyau correspondant. Pour ce faire, on utilise le but **install**. On utilise l'option **V=1** afin de visualiser précisément le répertoire de stockage des modules.

```
# make install V=1
make -C /lib/modules/2.6.12-14mdk/build
M=/home/pierre/docs/articles/lmf/kernel_programming/01-
linux_modules/exemples/hello_world modules_install
...
mkdir -p /lib/modules/2.6.12-14mdk/extra; cp
/home/pierre/docs/articles/lmf/kernel_programming/01-
linux_modules/exemples/hello_world/helloworld.ko
/lib/modules/2.6.12-14mdk/extra
make[1]: Leaving directory `/usr/src/linux-2.6.12-14mdk'
```

On note que les modules développés par l'utilisateur sont installés dans le répertoire **/lib/modules/version_de_noyau/extra**. On peut, bien entendu, les installer dans un autre sous-répertoire de **/lib/modules/version_de_noyau** en utilisant directement la commande **cp**.
Lorsque le module est installé, on peut alors utiliser la commande **modprobe** associée au nom du module et non plus au nom du fichier **.ko**.

```
 # modprobe helloworld
```

Le principal avantage de **modprobe** est de gérer les dépendances des modules. Avant de charger un module, **modprobe** charge automatiquement tous les modules nécessaires. Dans le paragraphe suivant, nous allons décrire un exemple de dépendance de module.

Identification du module

Il est possible d'identifier le module en utilisant des macros spécifiques, le plus souvent placées au début du code source. Dans le cas de notre module de test, on peut ajouter la description suivante.

```
MODULE_DESCRIPTION("Hello World module");
MODULE_AUTHOR("Pierre Ficheux, Open Wide");
```

```
MODULE_LICENSE(“GPL”);
```

Ces informations sont utilisées par la commande **modinfo**.

```
# modinfo helloworld.ko
filename:       helloworld.ko
description:    Hello World module
author:         Pierre Ficheux, Open Wide
license:        GPL
vermagic:       2.6.12-14mdk 686 gcc-4.0
depends:
```

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.
```

Cela indique le chargement d’un module utilisant une licence inconnue (propriétaire ?) alors que comme décrit au début de cet article, seule la GPL est autorisée. De même, la déclaration d’une licence non-GPL (exemple : **MY_LICENSE**) donne un résultat identique.

```
 module license ‘MY_LICENSE’ taints kernel.
```

On remarque cependant que le module est tout de même chargé en dépit de l’apparition du message. Le message a un but informatif avant tout.

Dépendance des modules

A partir de l’exemple précédent, nous pouvons créer un module légèrement différent. Au lieu d’utiliser directement la fonction **printk()**, nous définissons une fonction **my_printk()** dans un module externe **my_printk.c**. La structure est similaire à celle du module précédent, sauf que la fonction **my_printk()** est exportée donc visible des autres modules. Si l’on utilise le mot-clé **EXPORT_SYMBOL_GPL**, l’accès à la fonction sera réservé aux modules déclarant une licence GPL par la macro **MODULE_LICENSE**. Dans le cas contraire, il faudra utiliser le mot-clé **EXPORT_SYMBOL**.

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

Dans le code source du nouveau module **Hello World**, on utilise la fonction.

```
void my_printk (char *);
static int __init hello_init(void)
{
```

```
        my_printk("Hello world!");
        return 0;
}
```

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 in module
# insmod ../my_printk/my_printk.ko
# insmod helloworld2.ko
```

Cette méthode est laborieuse en cas de dépendances multiples. L'approche correcte est d'installer les modules dans l'arborescence du noyau et d'utiliser la commande **modprobe**. Cette commande utilise le fichier **modules.dep** créé par l'appel à depmod chargé de calculer les dépendances entre les modules. Le fichier est stocké dans l'arborescence binaire du noyau (**/lib/modules/version_de_noyau**). La structure du fichier **modules.dep** est simple, mais efficace. Dans notre cas, on remarque la dépendance du module **helloworld2** par rapport au module **my_printk**.

```
/
lib/modules/2.6.17.11/extra/helloworld2.ko:/lib/modules/2.6.17.11/extra/my_prin
tk.ko
/lib/modules/2.6.17.11/extra/my_printk.ko:
```

Pour tester cela, on installe les modules en question.

```
# make install
# cd ../my_printk
# make install
# depmod -a
```

On peut ensuite utiliser la commande **modprobe**.

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

Remarque :

Pour que le système de dépendances fonctionne correctement, l'arborescence binaire du noyau doit contenir le fichier **Module.symvers**. En cas d'erreur, on obtient le message suivant à la compilation des modules :

```
  WARNING: Symbol version dump /usr/src/version_noyau/Module.symvers
           is missing; modules will have no dependencies and modversions.
```

Passage de paramètres

Il est fréquent de vouloir passer des paramètres à un module avec une approche similaire à ce que l'on fait en espace utilisateur :

```
$ mon_programme 3 une_chaîne 17 28
```

Il existe pour cela des macros définies dans le fichier **include/linux/moduleparam.h**. Dans le cas présent, nous allons ajouter le passage de paramètres à l'exemple traité dans cet article.

- entier ;
- chaîne de caractères ;
- tableau d'entiers.

Au niveau du code source, cela implique les modifications suivantes. Tout d'abord, on définit la liste des variables réservées au traitement des paramètres.

```
/* Paramètres: entier, chaîne, tableau */
#define MAX_STRING     10
#define MAX_TAB        16
static int entier;
static char chaine[MAX_STRING];
static int tableau[MAX_TAB];
```

Puis, on utilise les macros dédiées. Le dernier paramètre (0644) définit les droits d'accès au paramètre dans le répertoire **/sys/module/nom_du_module/parameters/nom_du_paramètre**.

```
module_param(entier, int, 0644);
module_param_array(tableau, int, NULL, 0644);
module_param_string(chaine, chaine, sizeof(chaine), 0644);
```

On peut également ajouter la description des paramètres, accessible par la commande **modinfo** (voir paragraphe précédent).

```
MODULE_PARM_DESC(entier, "Un entier");
MODULE_PARM_DESC(chaine, "Une chaîne de caractères");
MODULE_PARM_DESC(tableau, "Un tableau d'entiers séparés par des virgules");
```

Dans la fonction d'initialisation, on utilise la fonction **printk()** pour afficher les valeurs des paramètres.

```
  int i;
  printk(KERN_INFO "entier= %d chaine= %s\n", entier, chaine);
  for (i = 0  ; i < MAX_STRING ; i++)
    printk(KERN_INFO "tableau[%d] = %d\n", i,  tableau[i]);
```

Après compilation du module, on peut alors tester son chargement. On obtient l'affichage des valeurs des paramètres dans le système de trace.

```
# modinfo helloworld3.ko
filename:       helloworld3.ko
```

```
description:    Hello World module
author:         Pierre Ficheux, Open Wide
license:        GPL
vermagic:       2.6.12-14mdk 686 gcc-4.0
depends:
parm:           chaine:Une chaîne de caractères (string)
parm:           tableau:Un tableau d'entiers séparés par des virgules (array of
int)
parm:           entier:Un entier (int)
# insmod helloworld3.ko entier=10 chaine=essai tableau=1,2,3,4
# dmesg
entier= 10 chaine= essai
tableau[0] = 1
tableau[1] = 2
tableau[2] = 3
tableau[3] = 4
tableau[4] = 0
tableau[5] = 0
tableau[6] = 0
tableau[7] = 0
tableau[8] = 0
tableau[9] = 0
```

```
# modinfo helloworld3.ko
filename:       helloworld3.ko
description:    Hello World module
author:         Pierre Ficheux, Open Wide
license:        GPL
vermagic:       2.6.12-14mdk 686 gcc-4.0
depends:
parm:           chaine:Une chaîne de caractères (string)
parm:           tableau:Un tableau d'entiers séparés par des virgules (array of
int)
parm:           entier:Un entier (int)
# insmod helloworld3.ko entier=10 chaine=essai tableau=1,2,3,4
# dmesg
entier= 10 chaine= essai
tableau[0] = 1
tableau[1] = 2
tableau[2] = 3
tableau[3] = 4
tableau[4] = 0
tableau[5] = 0
tableau[6] = 0
tableau[7] = 0
tableau[8] = 0
tableau[9] = 0
```

Lorsque le module est installé, on peut utiliser le fichier **/etc/modprobe.conf** pour indiquer les paramètres :

```
options helloworld3 entier=10 chaine=essai tableau=1,2,3,4
```

Remarque :

La modification du fichier nécessite l'appel à la commande **depmod -a**.

Un exemple plus complet

Le module qui suit reprend la fonctionnalité du programme square utilisé dans l'article " Temps réel sous LINUX (reloaded) ", paru dans le hors-série numéro 24 de Linux Magazine. Nous rappelons que le but du programme était de générer un signal carré de période 50 ms sur le port parallèle d'un PC. Nous remarquons l'utilisation de la fonction **ioperm()** qui permet d'accéder à un port physique depuis un programme en espace utilisateur (adresse 0x378, soit le port parallèle du PC).

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <asm/io.h>
#define LPT 0x378
int ioperm();
int main(int argc, char **argv)
{
    setuid(0);
    /* Pour accéder au port depuis l'espace utilisateur */          if
(ioperm(LPT, 1, 1) < 0) {
        perror("ioperm()");
        exit(-1);
    }
    while(1) {
        outb(0x01, LPT);        /* Niveau à 1 */
        usleep(50000);    /* Attente de 50 ms */
        outb(0x00, LPT);  /* Niveau à 0 */
        usleep(50000);
    }
    return(0);
}
```

En plus de la gestion du compteur de période 50 ms, le module équivalent utilise le répertoire /proc pour transmettre à l'espace utilisateur le nombre d'arrivées à échéance du compteur. Au niveau du code source, on peut dégager les principaux éléments.
En premier lieu, des fichiers d'en-têtes en plus de ceux utilisés jusqu'à présent :

```
#include <linux/timer.h>        /* Gestion du compteur */
#include <linux/proc_fs.h>      /* Pour /proc */
```

On définit ensuite les paramètres du compteur :

```
#define LPT         0x378       /* Port parallèle */
#define PERIOD      HZ/20       /* 50 ms = 1000 / 20 s */
static struct timer_list timer; /* Définition du compteur */
static int timer_cnt;           /* Nombre de ticks */
static int nibl=0x01;        /* Valeur à écrire sur le pp */
```

Au niveau de la fonction de chargement, on ajoute l'initialisation du compteur et la création de l'entrée dans le répertoire **/proc**. Le compteur est associé à une fonction exécutée lors de l'échéance. La variable **jiffies** correspond au nombre de ticks d'horloge système depuis le démarrage de la

machine. La constante **HZ** représente le nombre de ticks d'horloge par seconde. Elle vaut 1000 dans le cas du noyau 2.6 (100 pour le noyau 2.4). Pour une période de 50 ms, la valeur est donc 1000/20.

```
  /* Initialize the timer */
  init_timer(&timer);
  timer.function = timer_handler;
  timer.data = 0;
  mod_timer(&timer, jiffies + PERIOD);
```

La fonction **timer_handler()** est la suivante :

```
 static void timer_handler(unsigned long arg)
{
  timer_cnt++;
  /* Ecriture sur le port parallèle */
  outb(nibl, LPT);
  nibl = ~nibl;    /* de 0 à 1 et vice versa... */
  /* compteur armé */
  mod_timer(&timer, jiffies + PERIOD);
}
```

La définition d'une entrée **/proc** est également associée à une fonction appelée lors de l'accès au fichier correspondant, soit **/proc/square** dans notre cas.

```
  /* Entrée /proc */
  proc_entry = create_proc_entry("square", 0, NULL);
  if (proc_entry) {
    proc_entry->get_info = square_get_info;
  } else {
    printk(KERN_ERR "Can't create proc entry for square\n");
    return -EAGAIN;
  }
```

La fonction **square_get_info()** est la suivante :

```
static int square_get_info(char *page, char **start, off_t off, int count)
{
  int len = 0;
  len += sprintf(page + len, "timer_cnt= %d\n", timer_cnt);
  return len;
}
```

Après compilation, on peut tester le fonctionnement du module par les commandes suivantes :

```
# insmod square_mod.ko
# cat /proc/square
timer_cnt= 1591
```