Programmation noyau sous Linux Pilotes en mode caractère

Ce nouvel article de la série est consacré à la programmation des pilotes de périphériques en mode dit " caractère ". Les connaissances acquises lors du premier article consacré à l'API des modules Linux vont nous permettre d'aborder assez simplement la notion de " pilote Linux " qui, finalement, est un module dans une version un peu plus ardue.

Nous aborderons les sujets suivants :

- l'introduction aux pilotes Linux : les différents types de pilotes ;
- l'interface avec l'espace utilisateur : le répertoire **/dev** ;
- l'API des pilotes en mode caractère ;
- le transfert de données avec l'espace utilisateur ;
- l'allocation dynamique de mémoire ;
- la méthode **ioctl** ;
- les files d'attente.

Les exemples fournis sont inspirés de ceux développés par Stelian Pop pour la formation " noyau Linux " d'Open Wide.

Introduction aux pilotes Linux

Le système Linux étant de la famille UNIX, la structure d'un pilote Linux est relativement proche de celle d'un pilote générique UNIX. Un pilote est une portion de code exécutée dans l'espace du noyau. Il est chargé de faire l'interface entre un programme utilisateur et un composant matériel. Ce dernier point n'est pas forcément vrai puisqu'il existe des pilotes de périphériques virtuels tels les systèmes de fichier. En toute rigueur, un programme UNIX devrait systématiquement accéder à un périphérique au travers d'un pilote. Ce n'est pas toujours le cas, puisque nous avons vu dans plusieurs articles précédents qu'il était possible – sous Linux – d'accéder à une ressource matérielle grâce à la fonction **ioperm()**, puis aux fonctions **inb()** et **outb()**. Cette méthode est cependant peu recommandée et rarement utilisée sauf pour des cas simples ou très particuliers.
L'API d'un pilote Linux est très fortement inspirée de celle des pilotes UNIX des années 1970. Nous rappelons qu'elle est basée sur l'utilisation de fonctions ou méthodes permettant d'effectuer des actions basiques.

- une méthode **open()** permettant d'ouvrir le périphérique ;
- une méthode **close()** permettant de fermer (libérer) le périphérique. Sous Linux, cette méthode est nommée **release()** ;
- une méthode **read()** permettant de lire des données du périphérique ;
- une méthode **write()** permettant d'écrire des données sur le périphérique ;
- une méthode **ioctl()** (Input Output ConTroL) permettant de configurer le périphérique.

Il existe d'autres méthodes comme **lseek()**, **poll()** ou **mmap()**, mais elles sont moins fréquemment utilisées dans les pilotes simples.
Les méthodes sont activées depuis un programme de l'espace utilisateur en passant par les fichiers spéciaux du répertoire **/dev**. Ces fichiers sont appelés nœuds (nodes ou devices en anglais). Nous décrirons plus précisément ce répertoire plus loin dans le document.
Le pilote est conforme à l'API des modules Linux décrite dans l'article précédent. Du point de vue

fonctionnel, l'API des modules n'a rien à voir avec les pilotes. Elle permet simplement de charger dynamiquement le pilote dans l'espace du noyau en cours d'exécution.

Remarque :

Un pilote externe aux sources du noyau Linux sera toujours développé sous forme de module.

Il existe différents types de pilotes, liés aux types de périphériques qu'ils peuvent contrôler.

- Les pilote en mode caractère (char device driver). Ces pilotes sont destinés à manipuler les périphériques les plus courants avec lesquels ils échangent des données sous forme de flux d'octets de taille variable (minimum 1 octet). De ce fait, la plupart des pilotes de périphérique sous Linux seront en mode caractère. L'exemple le plus courant est le pilote de l'interface série RS-232.
- Les pilotes en mode bloc (block device driver). Ces pilotes sont destinés à manipuler des périphériques de stockage avec lesquels ils échangent des blocs de données (disque dur, CDROM, DVD, disque mémoire, etc.). La taille des blocs peut être de 512, 1024, 2048 (ou plus) octets suivant le périphérique.
- Les pilotes de périphériques réseau (network device driver). Ils sont destinés à gérer des contrôleurs réseau (exemple : carte Ethernet), mais aussi des piles de protocoles. Contrairement aux autres pilotes, ils n'utilisent pas d'entrée dans le répertoire **/dev**.

A cette liste, nous pouvons également ajouter quelques pilotes spéciaux le plus souvent dédiés à des bus ou des API particulières, citons entre autres :

- le bus PCI ;
- le bus USB ;
- les pilotes vidéo (V4L et V4L2).

La structure générale du noyau Linux est présentée dans la figure ci-dessous. On peut y voir le positionnement des principaux types de pilotes (schéma par Julien Gaulmin).

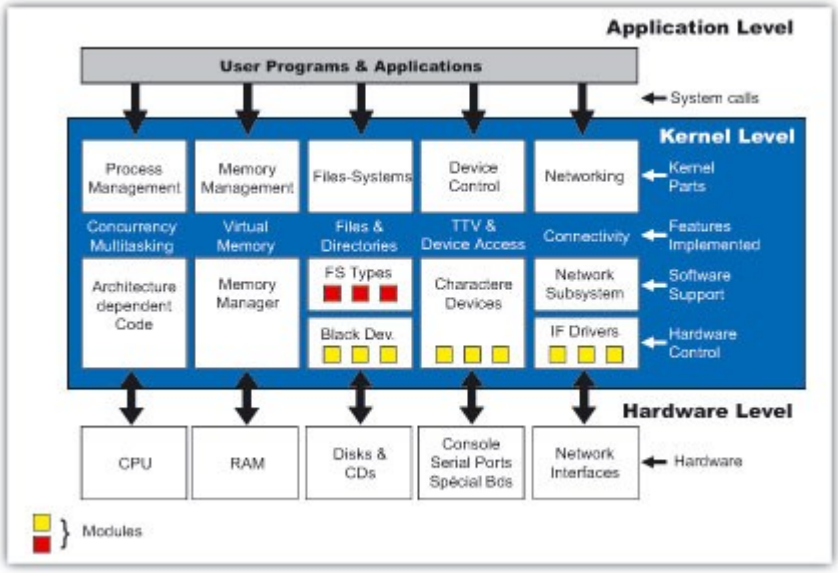

Figure 1 : Structure du noyau Linux

Dans le présent article, nous traiterons uniquement le cas des pilotes en mode caractère.

Quelques règles d'usage

La programmation d'un pilote est notoirement plus complexe que celle d'un programme en espace

utilisateur.

- Le noyau est un élément fondamental du système et un pilote mal conçu peut entraîner des dysfonctionnements importants, voire un arrêt du système, ce qui n'est pas le cas dans l'espace utilisateur.
- La mise au point dans l'espace noyau est complexe (voir l'article " Débogage dans l'espace noyau avec KGDB " dans le magazine LM 88).
- Les fonctions de la **glibc** ne sont pas disponibles dans l'espace noyau, mais certaines sont implémentées dans le répertoire **lib** des sources du noyau.
- Un pilote doit être programmé en C (pas de C++), mais la structure d'un pilote est " orientée objet ".
- En toute rigueur, on doit respecter le style de programmation défini dans le fichier **Documentation/CodingStyle** des sources du noyau.
- Un pilote doit prendre en compte les différentes architectures, particulièrement au niveau des " big-endians " et " little-endians ".

Tout cela doit inciter le lecteur à concevoir des pilotes les plus simples possibles et de reporter la difficulté sur l'espace utilisateur. De plus, nous rappelons une nouvelle fois qu'un pilote Linux doit en théorie être diffusé sous GPL, ce qui peut poser des problèmes par rapport à certains codes sensibles.

Le point de vue de l'espace utilisateur

Dans le cas d'un pilote en mode caractère, le périphérique est vu depuis un programme utilisateur comme un fichier spécial du répertoire **/dev**. Le fichier est dit " spécial ", car il n'occupe quasiment pas d'espace sur le disque (uniquement le point d'entrée). Le but du fichier spécial est uniquement de faire un lien entre l'espace utilisateur et le pilote de périphérique associé.

Principe du majeur/mineur

Si l'on regarde le fichier spécial associé au premier port série, on obtient :

```
$ ls -l /dev/ttyS0
crw-rw---- 1 root uucp 4, 64 oct 31 10:34 /dev/ttyS0
```

Le premier caractère identifie le type de périphérique, soit ici la lettre c pour caractère. Les neufs caractères suivants correspondent aux droits d'accès habituels sous Linux. Il en est de même pour le propriétaire et le groupe.
Par contre, les deux champs qui suivent sont particuliers aux fichiers spéciaux.

- La première valeur appelée " majeur " identifie le type de périphérique (ici le port série).
- La seconde valeur appelée " mineur " identifie l'instance du périphérique. Ces mineurs permettent de gérer plusieurs périphériques identiques avec le même pilote (donc le même majeur).

La valeur du majeur est unique. Dans le cas de Linux, la liste des majeurs réservés est disponible dans la documentation des sources du noyau, soit le fichier **Documentation/devices.txt**. La liste des majeurs actifs à un instant donné est disponible dans le fichier **/proc/devices**.

```
 $ cat /proc/devices
Character devices:
```

```
  1 mem
  4 /dev/vc/0
  4 tty
  4 ttyS
  5 /dev/tty
  5 /dev/console
...
```

Remarque :

L'utilisation erronée d'un numéro de majeur réservé entraînerait de fortes perturbations dans le fonctionnement du système !

Dans le cas de l'ajout d'un nouveau pilote par l'utilisateur, il sera préférable d'utiliser les fonctionnalités d'allocation dynamique de majeur ou de mineur plutôt que d'utiliser une valeur fixe. Ce point sera abordé dans la description de l'API.
La création d'un nouveau fichier spécial s'effectue grâce à la commande **mknod**. Si l'on devait créer le fichier spécial **/dev/ttyS0**, on utiliserait la commande :

```
# mknod /dev/ttyS0 c 4 64
```

Pour supprimer un fichier spécial, on utilise la commande classique **rm**.

```
# rm -f /dev/ttyS0
```

Limites du système de majeur/mineur

Cette approche est simple, mais elle pose un problème de duplication d'information entre l'espace utilisateur et l'espace noyau. En effet, les fichiers spéciaux sont créés dans **/dev** avec des valeurs passées à **mknod** ou bien à des utilitaires chargés de créer une liste de fichiers spéciaux (exemple : **MAKEDEV**). Pour que le système fonctionne, il faut que ces mêmes valeurs soient utilisées dans le code source des pilotes.
L'approche **devfs** permettait d'améliorer les choses pour le noyau 2.4. Grâce à **devfs**, les périphériques ne sont plus manipulés sous forme de majeur/mineur, mais sous forme de nom. L'entrée dans **/dev** est créée dynamiquement lors de l'insertion du pilote. Cependant le code de **devfs** est situé dans le noyau ce qui pose des problèmes de nommage et de compatibilité avec le standard LSB (Linux Standard Base) qui ne traite pas l'espace noyau.
De ce fait, une autre approche appelée udev est désormais utilisée pour le noyau 2.6 (même si la compatibilité devfs est conservée). Le principal intérêt de **udev** est de fonctionner entièrement en espace utilisateur. Un démon nommé **udevd** reçoit des instructions du noyau afin de procéder à la création de l'entrée dans **/dev** grâce à des règles modifiables par l'administrateur. Techniquement parlant, **udev** est basé sur deux composants liés au noyau 2.6.

- **hotplug** : un démon chargé de la gestion de l'insertion/suppression des périphériques ;
- **sysfs** (monté sur **/sys**) : un système de fichier virtuel similaire à **/proc** décrivant les périphériques connectés au système.

Le lien vers un article complet concernant **udev** est disponible dans la bibliographie.

API de programmation

Nous avons vu, dans l'article précédent, l'API de programmation des modules Linux. Sachant que, dans notre cas, un pilote est systématiquement un module, cette API sera utilisée, enrichie de plusieurs éléments permettant de définir les méthodes décrites au début de l'article.
Comme nous l'avons fait pour les précédents articles et afin de faciliter la compréhension, nous baserons la démonstration sur un exemple simple et évolutif.

La structure file_operations

Cette structure dont le type est décrit dans le fichier **<linux/fs.h>** permet de définir les méthodes du pilote. La tradition veut que l'on préfixe le nom de la structure et des méthodes par le nom du pilote (ici **mydriver**).

```
static struct file_operations mydriver_fops = {
        .owner =        THIS_MODULE,
        .read = mydriver_read,
        .write =        mydriver_write,
        .open = mydriver_open,
        .release =      mydriver_release,
};
```

De part la nécessité de définir les symboles correspondant aux méthodes, on placera la déclaration de la structure après la définition des méthodes. De ce fait, l'architecture générale du pilote sera la suivante :
1. Déclaration des en-têtes.
2. Déclaration des macro-instructions identifiant le pilote.
3. Déclaration des variables.
4. Définition des méthodes.
5. Définition de la structure **file_operations**.
6. Définition des points d'entrée **module_init()** et **module_exit()** du module.

Déclarations à effectuer

Dans le cas simple qui nous intéresse, les déclarations sont similaires à celles que nous avons effectuées pour l'exemple des modules de l'article précédent.

```
#include <linux/kernel.h>       /* printk() */
#include <linux/module.h>       /* modules */
#include <linux/init.h> /* module_{init,exit}() */
#include <linux/fs.h>         /* file_operations */
MODULE_DESCRIPTION("mydriver1");
MODULE_AUTHOR("Stelian Pop/Pierre Ficheux, Open Wide");
MODULE_LICENSE(“GPL”);
```

La seule nouveauté est la présence du fichier d'en-tête **<linux/fs.h>** nécessaire à la définition de la structure **file_operations**.

L'allocation/libération du majeur ou du mineur

Chronologiquement, la fonction **module_init()** est la première à être exécutée lors du chargement du module. C'est donc dans cette fonction que l'on doit effectuer l'allocation du majeur ou du mineur suivant les cas. Si l'on part du principe que l'allocation sera dynamique, il y a deux solutions :
1. Allouer dynamiquement un majeur. Ce majeur sera ensuite visible dans **/proc/devices**.
2. Allouer dynamiquement un mineur lié au majeur du pilote **misc** (correspondant à la valeur 10). Dans ce cas, le mineur alloué sera visible dans **/proc/misc**.
Juste avant la définition de la fonction d'initialisation, nous plaçons la déclaration de la structure **file_operations**.

```
static struct file_operations mydriver1_fops = {
        .owner =        THIS_MODULE,
        .read = mydriver1_read,
        .write =        mydriver1_write,
        .open = mydriver1_open,
        .release =      mydriver1_release,
};
```

Dans le premier cas, le majeur par défaut vaut zéro – ce qui correspond à un majeur dynamique – mais il est possible de passer une valeur en paramètre du module, soit :

```
static int major = 0; /* Major number */
module_param(major, int, 0644);
MODULE_PARM_DESC(major, "Static major number (none = dynamic)");
```

L'enregistrement du pilote s'effectue grâce à la fonction **register_chrdev()**. Si le majeur vaut zéro, le système retournera un majeur dynamique. La définition de la fonction d'initialisation du module est la suivante :

```
static int __init mydriver1_init(void)
{
     int ret;
     ret = register_chrdev(major, "mydriver1", &mydriver1_fops);
     if (ret < 0) {
             printk(KERN_WARNING "mydriver1: unable to get a major\n");
             return ret;
     }
     if (major == 0)
        major = ret; /* dynamic value */
        printk(KERN_INFO "mydriver1: successfully loaded with major %d\n",
major);
        return 0;
}
```

La libération du majeur alloué s'effectue dans la fonction de déchargement du module dont le code source est le suivant. Pour cela, on utilise la fonction **unregister_chrdev()**.

```
 static void __exit mydriver1_exit(void)
{
        if (unregister_chrdev(major, "mydriver1") < 0) {
        printk(KERN_WARNING "mydriver1: error while unregistering\n");
        return;
```

```
        }
        printk(KERN_INFO "mydriver1: successfully unloaded\n");
}
```

La fin du programme correspond simplement à l'API des modules Linux.

```
module_init(mydriver1_init);
module_exit(mydriver1_exit);
```

Dans le cas de l'utilisation du pilote **misc**, la procédure à utiliser est un peu différente. Il faut tout d'abord inclure le fichier d'en-tête **<linux/miscdevice.h>**, puis déclarer une structure décrivant le pilote.

```
#include <linux/miscdevice.h>   /* misc driver interface */
static struct miscdevice mymisc; /* Misc device handler */
```

Dans la fonction d'initialisation du module, on alloue dynamiquement le mineur au lieu du majeur en utilisant la fonction **misc_register**().

```
 static int __init mydriver2_init(void)
{
  int ret;
  mymisc.minor = MISC_DYNAMIC_MINOR;
  mymisc.name = "mydriver2";
  mymisc.fops = &mydriver2_fops;
  ret = misc_register(&mymisc);
  if (ret < 0) {
    printk(KERN_WARNING "mydriver2: unable to get a dynamic minor\n");
    return ret;
  }
  return 0;
}
```

Pour libérer le mineur alloué, on utilise la fonction **misc_deregister**().

```
static void __exit mydriver2_exit(void)
{
  misc_deregister(&mymisc);
  printk(KERN_INFO „mydriver2: successfully unloaded\n“);
}
```

Les méthodes open() et release()

Ces deux méthodes sont les points d'entrée principaux d'un pilote UNIX, même si, sous Linux, certaines tâches d'initialisation peuvent être reportées dans les fonctions **module_init**() et **module_exit**(). La méthode **open()** est réservée à des tâches d'initialisation matérielle du périphérique (exemple : initialisation matérielle d'un bus ou allocation dynamique de mémoire). La méthode **release()** effectue la libération des ressources allouées. Dans le cas de l'exemple, les méthodes effectuent uniquement un affichage.

```
static int mydriver1_open(struct inode *inode, struct file *file)
{
        printk(KERN_INFO "mydriver1: open()\n");
        return 0;
}
static int mydriver1_release(struct inode *inode, struct file *file)
{
        printk(KERN_INFO "mydriver1: release()\n");
        return 0;
}
```

Dans d'autres pilotes plus complexes, on peut entre autres tester la validité du mineur. Voici l'exemple du pilote du port parallèle.

```
 static int lp_open(struct inode * inode, struct file * file)
{
        unsigned int minor = iminor(inode);
        if (minor >= LP_NO)
                return -ENXIO;
        ...
}
```

Les méthodes read() et write()

Ces méthodes sont, en général, les plus utilisées, car elles permettent d'échanger des données avec le périphérique grâce à un tampon **buf**. Dans l'exemple simple qui suit, le code se contente d'afficher un message. Nous verrons plus loin les fonctions à utiliser pour échanger des données entre l'espace utilisateur et l'espace noyau.

```
 static ssize_t mydriver1_read(struct file *file, char *buf, size_t count,
loff_t *ppos)
{
        printk(KERN_INFO "mydriver1: read()\n");
        return count;
}
static ssize_t mydriver1_write(struct file *file, const char *buf, size_t
count, loff_t *ppos)
{
        printk(KERN_INFO "mydriver1: write()\n");
        return count;
}
```

Compilation et test du pilote

Le fichier **Makefile** est identique à celui utilisé pour les exemples de modules Linux.

```
 KDIR= /lib/modules/$(shell uname -r)/build
obj-m := mydriver1.o
all:
        $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules
install:
        $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules_install
        depmod -a
```

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

On teste le premier module en effectuant la compilation, puis l'insertion du module grâce à **insmod**.

```
$ make
# insmod mydriver1.ko
# dmesg
mydriver1: successfully loaded with major 253
```

Le système a affecté dynamiquement le majeur 253 au nouveau pilote, cependant le fichier spécial dans **/dev** n'est pas créé par défaut. Pour ce faire, on doit utiliser la commande **mknod** pour créer le fichier associé que nous appellerons **mydriver1**.

```
 # mknod /dev/mydriver1 c 253 0
```

En pratique, on peut utiliser un script simple permettant de charger le module et de créer le fichier spécial en lisant **/proc/devices**.

```
#!/bin/sh
if [ $# -eq 0 ]; then
    echo "Usage: $0 module_name"
    exit 1
fi
insmod ${1}.ko
X=`grep $1 /proc/devices`
if [ "$X" != "" ]; then
    set $X
    rm -f /dev/$2
    mknod /dev/$2 c $1 0
else
    echo "Module $1 not loaded !"
    exit 1
fi
```

A partir de là, le pilote est accessible depuis l'espace utilisateur. Pour tester le pilote, il n'est pas nécessaire d'écrire un programme, puis le langage shell fournit les moyens grâce aux opérateurs de redirection. On peut tester l'ouverture, puis la fermeture par :

```
# < /dev/mydriver1
```

Ce qui provoque l'affichage suivant dans les traces du noyau :

```
mydriver1: open()
mydriver1: release()
```

On teste l'écriture par :

```
 # echo x > /dev/mydriver1
```

Ce qui provoque l'affichage suivant dans les traces du noyau :

```
mydriver1: open()
mydriver1: write()
mydriver1: release()
```

Pour tester la lecture, on peut utiliser la commande **dd** :

```
dd bs=1 count=1 < /dev/mydriver1
```

Ce qui provoque l'affichage suivant dans les traces du noyau :

```
mydriver1: open()
mydriver1: read()
mydriver1: release()
```

Dans le cas du pilote utilisant l'allocation dynamique du mineur, la procédure est la même. On insère le module.

```
# insmod mydriver2.ko
```

Le point d'entrée correspondant apparaît automatiquement dans **/proc/misc** et dans le répertoire **/dev**.

```
# cat /proc/misc
 62 mydriver2
 63 device-mapper
175 agpgart
144 nvram
228 hpet
135 rtc
231 snapshot
# ls -l /dev/mydriver2
crw------- 1 root root 10, 62 nov  5 13:59 /dev/mydriver2
```

Cette apparition " magique " dans **/dev** n'en est pas vraiment une et la création du fichier spécial est effectuée par **udev** déjà cité au début de l'article (si le démon **udevd** est activé).
Le fonctionnement de udev est basé sur le système de fichier **sysfs** monté sur **/sys**. Le fichier spécial est créé, car un pilote de type misc possède une entrée dans le répertoire **/sys/class/misc**. Cette entrée est présente grâce à l'appel à class_device_create() présent dans la fonction **misc_register()**.
L'absence de classe dans le cas de l'allocation dynamique du majeur (voir l'exemple précédent, soit **mydriver1**) empêche la création automatique du fichier spécial.
Si nous revenons à l'exemple **mydriver2**, nous obtenons :

```
# ls -l /sys/class/misc
total 0
drwxr-xr-x 2 root root 0 nov  5 12:32 agpgart
drwxr-xr-x 2 root root 0 nov  5 12:32 device-mapper
drwxr-xr-x 2 root root 0 nov  5 12:32 hpet
drwxr-xr-x 2 root root 0 nov  5 14:02 mydriver2
drwxr-xr-x 2 root root 0 nov  5 12:32 nvram
drwxr-xr-x 2 root root 0 nov  5 12:32 rtc
drwxr-xr-x 2 root root 0 nov  5 12:32 snapshot
```

On peut augmenter le niveau de trace du démon udevd en modifiant le fichier **/etc/udev/udev.conf**.

```
# udev.conf
# The initial syslog(3) priority: "err", "info", "debug" or its
# numerical equivalent. For runtime debugging, the daemons internal
# state can be changed with: "udevcontrol log_priority=<value>".
#udev_log=”err”
udev_log=“debug“
```

Lors de l’insertion du module **mydriver2.ko**, on obtient les traces suivantes dans le fichier **/var/log/messages** :

```
Nov  5 13:59:48 dhcp-588-2 udevd[438]: udev_event_run: seq 792 forked, pid
[5983], ‘add’ ‘module’, 0 seconds old
Nov  5 13:59:48 dhcp-588-2 udevd[438]: udev_event_run: seq 793 forked, pid
[5984], ‘add’ ‘misc’, 0 seconds old
Nov  5 13:59:48 dhcp-588-2 udevd-event[5983]: udev_rules_get_run: rule applied,
‘mydriver2’ is ignored
Nov  5 13:59:48 dhcp-588-2 udevd-event[5983]: udev_device_event: device event
will be ignored
Nov  5 13:59:48 dhcp-588-2 udevd-event[5983]: udev_event_run: seq 792 finished
Nov  5 13:59:48 dhcp-588-2 udevd[438]: udev_done: seq 792, pid [5983] exit with
0, 0 seconds old
Nov  5 13:59:48 dhcp-588-2 udevd-event[5984]: udev_rules_get_name: no node name
set, will use kernel name ‘mydriver2’
Nov  5 13:59:48 dhcp-588-2 udevd-event[5984]: create_node: creating device node
‘/dev/mydriver2’, major = ‘10’, minor = ‘62’, mode = ‘0600’, uid = ‘0’, gid =
‘0’
```

L’utilisation du pilote **misc** est donc intéressante, car elle évite l’étape de création du point d’entrée correspondant au majeur dynamique dont la valeur peut changer suivant la configuration du système. Le point d’entrée disparaît lorsque l’on décharge le module. Par contre, chaque pilote **misc** ne peut utiliser qu’un seul mineur.

```
 # rmmod mydriver2
# ls -l /dev/mydriver2
ls: /dev/mydriver2: Aucun fichier ou répertoire de ce type
# cat /proc/misc
 63 device-mapper
175 agpgart
144 nvram
228 hpet
135 rtc
231 snapshot
```

Echange de données avec le pilote

Les programmes utilisateurs et les pilotes ne fonctionnant pas dans le même espace de mémoire, il est nécessaire d'utiliser des fonctions dédiées pour écrire ou lire des données vers ou en provenance du pilote. Les fonctions à utiliser sont déclarées dans **<asm/uaccess.h>**.

```
- copy_from_user(void *to, void *from, unsigned long size)
- copy_to_user(void *to, void *from, unsigned long size)
```

L'exemple que nous proposons utilise un tampon statique – un tableau de 64 caractères – permettant de recevoir des données par une écriture depuis l'espace utilisateur, puis de renvoyer ces mêmes données lors d'une lecture depuis cet espace utilisateur. Pour cela, nous devons déjà inclure le fichier **<asm/uaccess.h>**.

```
#include <asm/uaccess.h>        /* copy_{from,to}_user() */
```

Il faut ensuite déclarer le tampon, ainsi que le nombre d'éléments disponibles :

```
#define BUF_SIZE 64
static char buffer[BUF_SIZE];
static size_t num = 0; /* Number of available bytes in the buffer */
```

La méthode **write()** lit les données provenant de l'espace utilisateur dans la limite de la taille du tampon.

```
 static ssize_t mydriver3_write(struct file *file, const char *buf, size_t
count, loff_t *ppos)
{
  size_t real;
  real = min((size_t)BUF_SIZE, count);
  if (real)
    if (copy_from_user(buffer, buf, real))
      return -EFAULT;
  num = real; /* Destructive write (overwrite previous data if any) */
  printk(KERN_DEBUG "mydriver3: wrote %d/%d chars %s\n", real, count, buffer);
  return real;
}
```

La méthode **read()** retourne ces mêmes données à l'espace utilisateur et vide le tampon.

```
static ssize_t mydriver3_read(struct file *file, char *buf, size_t count,
loff_t *ppos)
{
  size_t real;
  real = min(num, count);
  if (real)
    if (copy_to_user(buf, buffer, real))
      return -EFAULT;
  num = 0; /* Destructive read (no more data after a read) */
  printk(KERN_DEBUG "mydriver3: read %d/%d chars %s\n", real, count, buffer);
  return real;
```

```
}
```

Pour tester le pilote, nous utilisons la même méthode que précédemment. Tout d'abord, nous chargeons le module, puis nous écrivons une chaîne de caractères sur le fichier spécial associé.

```
# insmod mydrive3.ko
# echo -n salut > /dev/mydriver3
```

Ce qui provoque l'affichage suivant dans les traces du noyau :

```
mydriver3: open()
mydriver3: wrote 5/5 chars salut
mydriver3: release()
```

Nous pouvons ensuite lire le contenu du tampon stocké par le pilote.

```
# dd bs=5 count=1 < /dev/mydriver3 2> /dev/null
salut
#
```

Ce programme en C permet d'effectuer le même test :

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <string.h>
main (int ac, char **av)
{
  char buf[64];
  int n, fd;
  fd = open (av[1], O_RDWR);
  if (fd < 0) {
    perror ("open");
    exit (1);
  }
  n = write (fd, av[2], strlen(av[2]));
  read (fd, buf, n);
  printf ("buf= %s\n");
  close (fd);
}
```

On utilise le programme comme suit :

```
# ./mydriver3_user /dev/mydriver3 salut
buf= salut
```

Allocation dynamique de mémoire

Dans l'espace utilisateur, l'allocation et la libération dynamique de mémoire s'effectuent grâce aux fonctions **malloc()** et **free()**. Ces fonctions ont leurs équivalents dans l'espace du noyau, en

l'occurrence **kmalloc()** et **kfree()**. La taille allouée par la fonction kmalloc() est cependant limitée à 128 Ko de mémoire ce qui peut être insuffisant pour certains pilotes. Dans ce cas, il est possible d'utiliser des fonctions plus complexes comme **__get_free_page()/free_pages()** qui alloue/libère plusieurs pages de mémoire contiguës de mémoire réelle ou bien **vmalloc()/vfree()** qui alloue un espace continu de mémoire virtuelle.
Pour illustrer cela, nous allons modifier l'exemple précédent pour qu'il utilise un tampon de mémoire dynamique au lieu d'un tampon statique.
Il faut tout d'abord ajouter le fichier d'en-tête décrivant les fonctions.

```
#include <linux/slab.h>         /* kmalloc()/kfree() */
```

La taille du tampon est par défaut de 64 octets, mais peut être modifiée en passant le paramètre au chargement du module.

```
static size_t buf_size = 64; /* Buffer size */
module_param(buf_size, int, 0644);
MODULE_PARM_DESC(buf_size, "Buffer size");
```

Le tampon est désormais un pointeur.

```
static char *buffer;             /* copy_from/to_user buffer */
```

L'allocation du tampon s'effectue dans la fonction m**odule_init()**. Le paramètre **GFP_KERNEL** indique une allocation normale de mémoire (pouvant " dormir "), alors que le paramètre **GFP_ATOMIC** indique une allocation " atomique ", ce qui est nécessaire dans certains cas (exemple : une fonction de traitement d'interruption).

```
 buffer = (char *)kmalloc(buf_size, GFP_KERNEL);
if (buffer != NULL) {
  printk(KERN_DEBUG "mydriver4: allocated a %d bytes buffer\n", buf_size);
} else {
  printk(KERN_WARNING "mydriver4: unable to allocate a %d bytes buffer\n",
buf_size);
  misc_deregister(&mymisc);
  return -ENOMEM;
}
```

La libération de mémoire s'effectue dans la fonction **module_exit()**.

```
 static void __exit mydriver4_exit(void)
{
  kfree (buffer);
  misc_deregister(&mymisc);
  printk(KERN_INFO "mydriver4: successfully unloaded\n");
}
```

La méthode ioctl()

Cette méthode est utilisée pour effectuer les opérations inaccessibles aux méthodes **read()** et **write()**. En général, cela correspond à des actions de configuration matérielle du périphérique (exemple : modifier la vitesse du port série). Il faut noter que la méthode **ioctl()** permet également de lire des valeurs à partir du pilote.
Dans notre exemple, nous considérons deux modes de fonctionnement (MODE1 et MODE2) affectés à la variable **my_mode** du pilote. La méthode **ioctl()** permet de lire et d'écrire les modes depuis l'espace utilisateur.

```
/* ioctl cmds */
#define SET_MODE 0
#define GET_MODE 1
/* ioctl valid args */
#define MODE1    1
#define MODE2    2
static int my_mode = MODE1;
```

Le code de la méthode **ioctl()** est le suivant. Il faut noter que la lecture du mode nécessite l'utilisation de la fonction **copy_to_user()** pour retourner la valeur à l'espace utilisateur. De même, l'utilisation de **copy_from_user()** sera nécessaire si le paramètre envoyé au pilote est passé par un pointeur de structure et non par un type de donnée simple.

```
 static int mydriver5_ioctl(struct inode *inode, struct file *file,
                   unsigned int cmd, unsigned long arg)
{
  printk(KERN_DEBUG "mydriver5: ioctl()\n");
  switch (cmd) {
  case SET_MODE :
    switch (arg) {
    case MODE1 :
      my_mode = MODE1;
      break;
    case MODE2 :
      my_mode = MODE2;
      break;
    default :
      printk(KERN_WARNING "mydriver5: arg %x unsupported in the SET_MODE ioctl
command\n", (int)arg);
            return -EINVAL;
    }
    break;
  case GET_MODE: /* Send my_mode value to user space */
    if (copy_to_user((void*)arg, &my_mode, sizeof(int))) {
      printk(KERN_WARNING "mydriver5: copy_to_user error\n");
      return -EFAULT;
    }
    break;
  default :
    printk(KERN_WARNING "mydriver5: 0x%x unsupported ioctl command\n", cmd);
    return -EINVAL;
  }
  return 0;
}
```

Il est bien entendu nécessaire d'ajouter la ligne suivante à la définition de la structure **file_operations** utilisée.

```
  .ioctl   =      mydriver5_ioctl,
```

Coté utilisateur, le code qui suit permet de tester la méthode :

```
  for (i = 1 ; i < 4 ; i++) {
    printf ("setting mode= %d\n", i);
    if ((n = ioctl (fd, SET_MODE, i)) < 0)
      perror ("ioctl");
    if ((n = ioctl (fd, GET_MODE, &arg)) < 0)
      perror ("ioctl");
    else
      printf ("arg= %d\n", arg);
  }
```

Ce qui donne à l'exécution :

```
 # ./ioctl_test /dev/mydriver5
setting mode= 1
arg= 1
setting mode= 2
arg= 2
setting mode= 3
ioctl: Invalid argument
arg= 2
```

Les files d'attente

Une file d'attente permet à un processus de libérer le processeur lorsqu'il est en attente de données. Le processus pourra être réveillé soit par un signal, soit par l'arrivée des données. L'exemple présenté est dérivé de celui du paragraphe concernant l'allocation dynamique de mémoire.
Nous déclarons tout d'abord une file d'attente.

```
static DECLARE_WAIT_QUEUE_HEAD(read_wait_queue); /* Read wait queue */
```

Dans la méthode **read()**, nous ajoutons la séquence de code suivante, qui indique au processus de s'endormir s'il n'y a pas de données disponibles.

```
  /* Sleep if no data available. */
  ret = wait_event_interruptible(read_wait_queue, num != 0);
  if (ret < 0) {
    printk(KERN_DEBUG "mydriver6: woke up by signal\n");
    return -ERESTARTSYS;
  }
```

Dans la méthode **write()**, nous ajoutons une ligne permettant de réveiller le processus, puisque des données sont alors disponibles.

```
 num = real; /* Destructive write (overwrite previous data if any) */
  /* Wake up one blocked processes in read_wait_queue */
  wake_up_interruptible(&read_wait_queue);
  printk(KERN_DEBUG "mydriver6: wrote %d/%d chars %s\n", real, count, buffer);
```

Nous déroulons la séquence de test. L'écriture puis la lecture des données s'effectuent comme dans l'exemple précédent.

```
# echo -n salut > /dev/mydriver6
# dd bs=5 count=1 < /dev/mydriver6 2> /dev/null
salut
#
```

Par contre, une deuxième lecture des données bloque le **read**(), puisqu'il n'y a plus de données disponibles. On débloque la lecture par un [Ctrl]-[C] ou en effectuant une nouvelle écriture sur le fichier spécial associé depuis un autre terminal.

```
# echo -n 12345 > /dev/mydriver6 2> /dev/null
12345
#
```

Conclusion

Nous avons abordé ici quelques éléments permettant de comprendre la structure des pilotes en mode caractère. Certains points importants, comme la gestion des interruptions, la méthode mmap() ou la création de threads en espace noyau, n'ont pas été traités. Ils seront abordés lors d'un prochain épisode !