Création de module dynamique

Cette aplication fait référence au magazine Linux Magazine N° 17 réalisé par Pierre FICHEUX.

1 – Création de module

L'étude suivante permet de comprendre et de réaliser un pilote de périphérique pour Linux. Ce pilote sera chargé de manière dynamique et donc ne sera pas intégré au noyau Linux. Ce pilote constitue un module chargeable. La version du noyau utilisée est la 2.4 avec une distribution Mandrake 9.0. Cette application ne fonctionne pas pour une version antérieure du noyau.

Il existe plusieurs type de pilote de périphérique:

- Pilotes en mode caractère (char driver). Permet l'échange de données en forme binaire. Exemple port de laison série, port parallèle...
- Pilotes en mode bloc (block drivers). Exemple disque dur.
- Pilotes réseau (network drivers). Ils pemettent de contrôler des ressources réseau.

Nous nous intéresserons uniquement aux pilotes en mode caractère. Ceux ci sont associés aux travers des fichiers spéciaux appelé noeud (node). Ces fichiers sont situés dans le répertoire /dev:

```
#ls  -l /dev/tty*
total 0
crw-rw----    1 herve    tty        4,  64 jan  1  1970 0
crw-rw----    1 herve    tty        4,  65 jan  1  1970 1
```

Le premier caractère 'c' indique que le pilote est est mode caractère. Le '4' est l'indice majeur (major) permettant l'identification du pilote. Le '64' et '65' est l'indice mineur (minor) indique une sous adresse. Cela permet de départager plusieurs périphériques identiques.

```
#ll /dev/ide/host0/bus0/target0/lun0/disc
brw-------    1 root     root       3,   0 jan  1  1970 /dev/ide /
host0/bus0/target0/lun0/disc
```

Ce périphérique qui est le disque dur indique qu'il fonctionne en mode bloc. Les indices majeur et mineur sont normalisés. Le fichier /usr/linux/Documentation/devices.txt indique ces valeurs normalisés. Voici un extrait de ce fichier réservé aux consoles et ports séries :

```
 4 char      TTY devices
               0 = /dev/tty0           Current virtual console

               1 = /dev/tty1           First virtual console
                 ...
              63 = /dev/tty63    63rd virtual console
              64 = /dev/ttyS0    First UART serial port
                 ...
             255 = /dev/ttyS191  192nd UART serial port
```

On peut faire le lien avec le port série (4). D'après ce fichier, on doit utiliser l'indice majeur 10 pour des périphériques divers (`10 char Non-serial mice, misc features`). L'indice mineur sera alloué de manière automatique au chargement du module.

2 - Ecriture et lecture du module

Le périphérique utilisé est un système composé de 8 leds commandées par le port parallèle via un Latch (74LS373). Les données du latch sont prises en compte par la broche autofeed (14).

La valeur transmise au module doit être une chaîne de caractère formant un nombre décimale. Pour écrire une donnée avec le Shell :

```
#echo 125 > /dev/pportled
```

En fait ce module ne recoit qu'une chaîne de caractères et convertie cette chaîne en nombre. On peut très bien écrire echo “titi“> /dev/pportled, mais là le module renvoie une érreur.

Pour lire le périphérique :

```
#dd bs=1 count=1 < /dev/pportled
```

La valeur retournée sera un caractère représenant le nombre décimale.

Pour accéder à ce périphérique en langage C on doit d'abord ouvrir le périphérique comme un fichier :

```
int fModule = open (“/dev/pportled“, O_RDWR);
```

Pour écrire :

```
char * sCommande = “55“;
write (fModule, sCommande, strlen(sCommande));
```

Pour lire :

```
char sBuffer[4];
read (fModule, sBuffer, 1);
```

Pour le fermer :

```
close (fModule);
```

Il est possible de controler l'opération effectuée, car chaque fonctions renvoient un code d'erreur.

Quel est l'intérêt d'utiliser un module alors que l'on peut accéder directement au périphérique en langage C par exemple?

- Tout d'abord on écrit qu'une fois se périphérique pour plusieurs programme. Il est alors plus facile le maintenir que l'ensemble des programmes.

Oui, mais si je n'est qu'un seul programme?

- On peut donner des droits d'utilisation d'un module et le faire apartenit à un groupe, alors que si le programme accède directement au périphérique, il ne peut s'éxécuter qu'en « root ».

3 - Accès aux modules par le Shell

- rmmod : Cette commande permet de décharger un module de la mémoire. L'option -a permet d'enlever tous les modules qui ne sont pas utilisés.
- lsmod : Affiche la liste des modules chargés en mémoire.
- insmod : Installe un module en mémoire.
- mknod : Permet de créer un noeud pour un module.
- modinfo : Affiche des informations à propos d'un module du noyau (Auteur, description, paramètres...)
- depmod : Gestion des dépendances entre les modules du noyau.
- modprobe : insertion et extraction automatique d un module dans le noyau avec résolution des dépendances (chargement des autres modules nécessaires).

Pour la réalisation du premier module « hello », on vérifie le chargement du module en lisant le tampon des messages du noyau.

- dmesg : pemet de lire le tampon des messages du noyau
- tail -n: pemet d'afficher n lignes en partant de la fin d'un fichier.

3.1 - Exemple du module « hello »

```
#include <linux/module.h>

int module_init(void)
{
  printk("<1>Hello, world\n");
  return 0;
}

void module_exit(void)
{
  printk("<1>Goodbye cruel world");
}
```

Compilons ce module:

cc -O -DMODULE -D__KERNEL__ -c hello.c

- Petit paragraphe sur la compilation :

L'option D permet de définir une macro qui est contenu dans un fichier header (.h) ou source (.c). Ici les macros MODULE et __KERNEL__ sont définies. On les retrouve par exemple dans le fichier /usr/include/linux/module.h :

ligne 181:
```
#ifdef __KERNEL__
#define HAVE_INTER_MODULE
```

ligne 199 :
```
#if defined(MODULE) && !defined(__GENKSYMS__)
```

J'ose espérer que vous n'avez eu aucune erreur de compilation. Le programme inclu le fichier module.h. La fonction printk permet d'écrire dans le tampon des messages. Les deux fonctions sont des macros définis dans le le fichier module.h, pemettent le chargement du module (module_init) et le déchargement (module_exit). Ce sont les points d'entrées du module.

NB : Il n'est pas nécessaie d'être « root » pour compiler ce module.

3.2 Chargeons...

Pour avoir accès à cette commande, vous devez être « root » :

```
#insmode hello.o
hello.o: kernel-module version mismatch
        hello.o was compiled for kernel version 2.4.19-16mdkcustom
        while this kernel is version 2.4.19-16mdk.
```

Surprise, le module ne se charge pas... C'est normal, le noyau est en 2.4.19-16mdk, alors que le lien vers version de la source linux est en 2.4.19-16mdkcustom et j'ai compilé avec la version « custom ». Si vous ne voulez pas ce type d'erreur, recompiler le noyau linux. (Voir le fichier NoyauMandrake.sxw).

Dans ce cas ajoutons l'option -f (force)

```
#insmod -f hello.o
Warning: kernel-module version mismatch
        hello.o was compiled for kernel version 2.4.19-16mdkcustom
        while this kernel is version 2.4.19-16mdk
Warning: loading hello.o will taint the kernel: no license
  See http://www.tux.org/lkml/#export-tainted for information about
tainted modules
Warning: loading hello.o will taint the kernel: forced load
Module hello loaded, with warnings
```

Vérifions si le module est bien chargé :

```
#dmesg | tail -1
Hello, world
```

Visualiser le module :

```
#lsmod
Module                  Size  Used by    Tainted: PF
hello                    304   0  (unused)
```

Et déchargeons ce module :

```
#rmmod hello
#dmesg | tail -1
Goodbye cruel world
```

Passons au chose sérieuses. La création du module pportled permettant de piloter des leds sur le port parallèle d'un PC.

4 - Module pportled

Tous les fichiers d'entête sité font références aux sources fournis avec Linux et sont situés dans /usr/include/linux.

file_operation :

```
struct file_operations {
       struct module *owner;
       loff_t (*llseek) (struct file *, loff_t, int);
       ssize_t (*read) (struct file *, char *, size_t, loff_t *);
       ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
       int (*readdir) (struct file *, void *, filldir_t);
       unsigned int (*poll) (struct file *, struct poll_table_struct *);
       int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned
long);
       int (*mmap) (struct file *, struct vm_area_struct *);
       int (*open) (struct inode *, struct file *);
       int (*flush) (struct file *);
       int (*release) (struct inode *, struct file *);
       int (*fsync) (struct file *, struct dentry *, int datasync);
       int (*fasync) (int, struct file *, int);
       int (*lock) (struct file *, int, struct file_lock *);
       ssize_t (*readv) (struct file *, const struct iovec *, unsigned long,
loff_t *);
       ssize_t (*writev) (struct file *, const struct iovec *, unsigned long,
loff_t *);
       ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *,
int);
       unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned
long, unsigned long, unsigned long);
       int (*dmapi_map_event) (struct file *, struct vm_area_struct *, unsigned
int);
};
```

Cette structure contient toutes les méthodes servant au kernel afin d'accéder aux fonctions du driver. Celle ci sont définies dans cette structure comme des pointeurs de fonctions (voir fichier d'entête fs.h). Il faut bien penser que toutes ces fonctions et paramètres sont passées par le noyau. Les derniers noyaux 2.2 et 2.4 utilisent une syntaxe de déclaration spécifique à GCC (owner, read...) qui évite de remplacer les méthodes non implémentées par la valeur NULL.

```
struct file_operations pportled_fops =
{
  owner   : THIS_MODULE,            // module *owner
  read    : pportled_read,    // Lecture du port
  write   : pportled_write,   // Ecriture du port
  open    : pportled_open,    // ouverture du port
  release : pportled_close,   // fermuture du port
};
```

THIS_MODULE est une macro servant à définir le module créé.

Ouverture :

```
static int pportled_open (struct inode *inode, struct file *flip)
```

Afin de supporter le chargement dynamique, le noyau met en place un système de comptage pour chaque driver afin de connaître le nombre de procéssus utilisant ce driver . La macro MOD_INC_USE_COUNT permet l'utilisation de ce système.

Les paramètres de cette fonction sont:
struct inode : Permet de retrouver les informations telle que l'utilisateur, groupe, le temps d'accès, la taille, les indices majeur et mineur... (Cette structure est définie dans fs.h)
struct file : C'est la structure de fichier caractérisé par le driver ouvert. Contient différents champs comme f_mode (Type d'acces du fichier lecture seule...), f_pos (position courante du fichier), private_data (données privées du drivers)...

Fermeture:

```
static int pportled_close (struct inode *inode, struct file *flip)
```

Cette fonction est appelée via le programme applicatif afin de fermer le système de fichier. A ce moment le driver doit libérer l'espace mémoire alloué (*private_data). De même, il est nécessaire d'utiliser la macro MODE_DEC_USE_COUNT. Si ce compteur est à 0, le noyau peut alors enlever le driver de la mémoire.

Lecture :

```
static ssize_t pportled_read (struct file *pFlip, char *sBuff,
size_t count, loff_t *ppos)
```

pFlip : Structure de fichier du driver.
sBuff : Buffer de données.
count : Taille des données à lire (size type).
ppos : Pointeur de position (long offset type).

Les données peuvent être copiées du noyau à l'espace utilisateurs par les fonctions copy_to_user(unsigned long dest, unsigned long src, unsigned long len) ou put_user (expression, addr) .

Ecriture :

```
static ssize_t pportled_write (struct file *pFilp, const char
*sBuff, size_t count, loff_t *ppos)
```

Les arguments sont identiques à la fonction de lecture. Ici les données doivent transférées de l'espace utilisateur au noyau par les fonctions get_user (lvalue, addr) ou copy_from_user (unsigned long dest, unsigned long src, unsigned long len).

Ces fonctions de transfert de mémoire sont déclarées dans le fichier asm/uaccess.h.

Point d'entrée :

Comme pour « hello », les deux fonctions permetant les points d'entrées sont module_init et module_exit. La fonctions misc_register permet d'enregister le pilote de caractère auprès du noyau (prototype : fs.h , source : linux/fs/device.c).

D'autres fonctions comme register_chrdev existe afin d'enregister le pilote, mais doivent être utilisées pour les modules non « misc ».

Certaines macros permetent de générer la description du module (module.h) :

- MODULE_AUTHOR() : Auteur du module.
- MODULE_DESCRIPTION() : Description du module.
- MODULE_LICENSE() : Accepte la licence Free Software Module .
- MODULE_SUPPORTED_DEVICE() : Peut être utilisé par kmod.
- MODULE_PARM() : Utilisé pour vérifier les paramètres donnés au module.
- MODULE_PARM_DESC() : Description des paramètres.

module.h :

```
#define MODULE_AUTHOR(name)                                              \
const char __module_author[] __attribute__((section(".modinfo")))= \
"author=" name

#define MODULE_DESCRIPTION(desc)         \
const char __module_description[] __attribute__((section
(".modinfo"))) =   \
"description=" desc
```

Etant donné que ce module est de type misc, un ensemble de fonctions permettent une gestion de chargement, et d'attribution du numéro mineur automatiquement. Le fichier miscdevice.h contient toutes des fonctions et structure permettant ces opérations. Tout d'abord, la structure miscdevice :

```
struct miscdevice
{
      int minor;                              // indice mineur
      const char *name;                       // nom du module
      struct file_operations *fops;
      struct miscdevice * next, * prev;
      devfs_handle_t devfs_handle;
};
```

Il faut renseigner une variable du type de cette structure, puis d'appeler la fonction misc_register contenant l'adresse de cette variable. La macro MISC_DYNAMIC_MINOR, permet la création du noeud de manière automatique

```
// Calcul dynamique du mineur
pportled_dev.minor = MISC_DYNAMIC_MINOR;

pportled_dev.name = "pportled";
pportled_dev.fops = &pportled_fops;

retval = misc_register (&pportled_dev);
```

Après compilation, il ne reste plus qu'à charger le module (inmod -f pportled). Le noeud est créé automatiquement, avec le numéro mineur. Il ne faut donc pas faire un mknod.

```
#ll /dev/pportled
lr-xr-xr-x    1 root     root           13 oct 21 21:30 /dev/pportled
-> misc/pportled
#ll /dev/misc/pportled
crwxrwxrwx    1 root     root      10,  63 jan  1  1970 /
dev/misc/pportled
```