Rapport de laboratoir CSEL - Samy Francelet et Romain Crettenand

Modules noyaux

Exercice 1 - Génération d'un module noyau

Afin de tester la génération et l'insertion d'un module noyau, nous avons développé un module qui affiche un message dans le debug lorsqu'il est chargé. Le code source est disponible le suivant:

```
// best.c
#include <linux/module.h> // needed by all modules
#include <linux/init.h>  // needed for macros
#include <linux/kernel.h> // needed for debugging

#include <linux/moduleparam.h> // needed for module parameters

static char* text = "This is the best module ever made!";
module_param(text, charp, 0664);
static int  elements = 1;
module_param(elements, int, 0);

static int __init best_init(void)
{
    pr_info ("Linux module 01 best loaded\n");
    pr_debug ("  text: %s\n  elements: %d\n", text, elements);
    return 0;
}

static void __exit best_exit(void)
{
    pr_info ("Linux module best unloaded\n");
}

module_init (best_init);
module_exit (best_exit);

MODULE_AUTHOR ("Samy Francelet <samy.francelet@ik.me>");
MODULE_DESCRIPTION ("Module best");
MODULE_LICENSE ("GPL");
```

Lors de son insertion, le module affiche le message suivant (obtenu avec dmesg):

```
[  678.337447] Linux module 01 best loaded
[  678.341360]   text: This is the best module ever made!
[  678.341360]   elements: 1
```

Comparaison des commandes lsmod et cat /proc/modules:

- lsmod

```
Module                  Size  Used by    Tainted: G
mymodule               16384  0
```

- cat /proc/modules

```
mymodule 16384 0 - Live 0xffff80000122b000 (O)
```

Les deux commandes affichent le nom du module, sa taille et le nombre de processus qui l'utilisent. La commande lsmod affiche également si le module est marqué comme "tainted" (c'est-à-dire si le module a été compilé avec une version différente du noyau). La commande cat /proc/modules affiche également l'état du module (Live, Loading, Unloading, etc.).

Lors de son retrait, le module affiche le message suivant (obtenu avec dmesg):

```
[  742.206592] Linux module best unloaded
```

Pour permettre l'installation du module avec la commande modprobe, il faut ajouter la ligne suivante dans le Makefile:

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

indiquant que le module doit être installé dans le répertoire $(MODPATH) lors de l'installation.

Exercice 2 - Paramètres de module

Dans cet exercice, il fallait créer un module pouvant recevoir des paramètres de la ligne de commande.

Pour cela, nous avons utilisé la fonction module_param() qui permet de définir des paramètres de la ligne de commande. Cette fonction prend en paramètre le nom du paramètre, son type, et les permissions d'accès.

Nous avons défini deux paramètres: un paramètre de type chaîne de caractères et un paramètre de type entier. Le premier paramètre est initialisé à la chaîne de caractères "This is the best module ever made!" et le second paramètre est initialisé à 1.

Pour tester le module avec des paramètres différents, nous avons utilisé la commande suivante:

```
modprobe mymodule.ko text="This is the worst module ever made!" elements=1234
```

Donnant le résultat suivant:

```
[ 2097.961276] Linux module 01 best loaded
[ 2097.965210]   text: This is the worst module ever made!
[ 2097.965210]   elements: 1234
```

Exercice 3 - cat /proc/sys/kernel/printk

La command cat /proc/sys/kernel/printk affiche le résultat suivant:

```
7       4       1       7
```

Le resultat montre les niveaux de priorité minimum des messages affichés par le noyau (dans l'ordre: *current*, *default*, *minimum*, boot-time-default*). Les niveaux de priorité sont les suivants:

- 0: Emergency
- 1: Alert
- 2: Critical
- 3: Error
- 4: Warning
- 5: Notice
- 6: Info
- 7: Debug

Exercice 4 - Gestion de la mémoire

Dans cet exercice, il fallait allouer dynamiquement de la mémoire pour spécifier le nombre d'éléments à créer, et initialiser ces éléments avec un texte passé en paramètre. Chaque élément doit posséder un identifiant unique. Les éléments doivent être détruits lors de la suppression du module.

Pour cela, nous avons utilisé la fonction kzalloc() qui permet d'allouer de la mémoire et de l'initialiser à 0. Cette fonction prend en paramètre la taille de la mémoire à allouer et la priorité d'allocation de la mémoire.

Résultat de l'installation/désinstallation du module:

```
[ 3188.026403] Linux module 01 best loaded
[ 3188.030311]   text: Hello
[ 3188.030311]   elements: 12
[ 3196.895052] Element 0: Hello
[ 3196.897994] Element 1: Hello
[ 3196.900925] Element 2: Hello
[ 3196.903818] Element 3: Hello
[ 3196.906717] Element 4: Hello
[ 3196.909599] Element 5: Hello
[ 3196.912493] Element 6: Hello
[ 3196.915383] Element 7: Hello
[ 3196.918277] Element 8: Hello
[ 3196.921156] Element 9: Hello
[ 3196.924050] Element 10: Hello
[ 3196.927026] Element 11: Hello
[ 3196.929992] Number of elements: 12
[ 3196.933407] All elements deleted (12 out of 12)
[ 3196.937942] Linux module best unloaded
```

Exercice 5 - Accès aux entrées/sorties

Le but de cet exercice était de créer un module qui peut accéder aux entrées/sorties du système. Pour cela, nous avons utilisé la fonction request_mem_region() qui permet de demander l'accès à une région mémoire spécifique. Cette fonction prend en paramètre l'adresse de la région mémoire et la taille de la région mémoire.

Ensuite si l'accès à la région mémoire est autorisé, nous avons utilisé la fonction ioremap() qui permet de mapper une région mémoire physique dans l'espace d'adressage virtuel. Cette fonction prend en paramètre l'adresse de la région mémoire physique et la taille de la région mémoire (typiquement 1 page, soit 0x1000 octets).

Avec la fonction iorea(), nous pouvons accéder à la mémoire physique à l'adresse virtuelle retournée par la fonction ioremap(). Ce qui nous permet de lire, par exemple, la valeur de température du processeur.

Exercice 6 - Threads du noyau

Afin de créer un thread du noyau, nous avons utilisé la fonction kthread_run() qui permet de créer et de lancer un thread du noyau. Cette fonction prend en paramètre la fonction à exécuter, les paramètres de la fonction, et le nom du thread.

Dans la fonction à exécuter, nous avons utilisé la fonction kthread_should_stop() dans une boucle qui permet de vérifier si le thread doit être arrêté. Cette fonction retourne 1 si le thread doit être arrêté, 0 sinon. Cela diffère du fonctionnement des threads utilisateurs qui utilisent simplement des boucles infinies.

Au retrait du module, nous avons utilisé la fonction kthread_stop() qui permet d'arrêter un thread du noyau.

Exercice 7 - Mise en sommeil

Dans cet exercice, il fallait créer deux threads du noyau qui s'exécutent en parallèle. Le premier thread doit attendre une notification de réveil pour s'exécuter. Le second thread transmet la notification de réveil au premier thread via une waitqueue.

Pour cela, nous avons utilisé la fonction wait_event_interruptible() qui permet de mettre un thread en attente d'une notification. Cette fonction prend en paramètre une waitqueue et une fonction de test. Cette fonction retourne 0 si la notification a été reçue, -ERESTARTSYS si le thread a été interrompu.

Exercice 8 - Gestion des interruptions

Dans cet exercice, il fallait générer une interruption quand un bouton est appuyé.

Pour cela, nous avons d'abord utilisé la fonction gpio_request() qui permet de demander l'accès à un port GPIO. Cette fonction prend en paramètre le numéro du port GPIO et le nom du port GPIO.

Une fois le port GPIO accédé, nous avons utilisé la fonction request_irq() qui permet de demander l'accès à une interruption. Cette fonction prend en paramètre le numéro de l'interruption, la fonction à exécuter lors de l'interruption, le type d'interruption, le nom de l'interruption, et un pointeur vers des données.

Dans la fonction à exécuter (de type irq_handler_t), nous retournons IRQ_HANDLED pour indiquer que l'interruption a été traitée. Dans le cas d'une interruption différée, nous retournons IRQ_WAKE_THREAD (il aurait fallu utiliser la fonction request_threaded_irq() pour cela).

Pilotes de périphériques

Exercice 1 - Pilotes orientés mémoire

Dans cet exercice, il fallait réaliser un pilote orienté mémoire permettant de mapper en espace utilisateur les registres du microprocesseur en utilisant /dev/mem. Ce pilote permet de lire le Chip-ID.

Pour ce faire, nous avons procédé ainsi:

- Ouvrir le fichier /dev/mem avec la fonction open()
- Mapper la mémoire physique du Chip-ID dans l'espace d'adressage virtuel avec la fonction mmap(), avec les paramètres suivants:
    - NULL: laisse le kernel choisir l'adresse virtuelle
    - pz: une taille de page

    - PROT_READ | PROT_WRITE: autorise la lecture et l'écriture
    - MAP_SHARED: partage la mémoire avec d'autres processus
    - fd: le descripteur de fichier du fichier /dev/mem précédemment ouvert
    - l'offset du Chip-ID dans le fichier /dev/mem (soit dev_addr - (dev_addr % pz))
- Lu les registres du Chip-ID à l'adresse virtuelle retournée par mmap()
- Dé-mapper la mémoire avec la fonction munmap()
- Fermer le fichier /dev/mem avec la fonction close()

Pilotes orientés caractères

Exercice 2

Dans cet exercice, il fallait réaliser un pilote orienté caractère capable de stocker dans une variable globale au module les données reçues par l'opération write et de les restituer par l'opération read. Pour tester le module, nous utilisons les commandes echo et cat.

Pour réaliser un pilote orienté caractère, nous devons définir les fonctions suivantes:

- open(): appelée lors de l'ouverture du fichier
- release(): appelée lors de la fermeture du fichier
- read(): appelée lors d'une lecture du fichier (cat)
- write(): appelée lors d'une écriture dans le fichier (echo >)

Ces fonctions sont enregistrées dans la structure file_operations avec la fonction cdev_init() lors de l'initialisation du pilote. Avant d'utiliser cdev_init(), il faut enregistrer le numéro majeur du pilote avec la fonction alloc_chrdev_region().

Une fois le pilote enregistré et installer, nous devons lui associer un fichier avec la commande mknod :

```
mknod /dev/mydriver c 511 0
```

Une fois le fichier associé, nous pouvons utiliser le pilote avec les commandes echo et cat:

```
echo "Coucou CSEL!" > /dev/mydriver
cat /dev/mydriver
Coucou CSEL!
```

Résultats des messages debug:

```
[ 2712.768790] mydriver: registered dev with major 511 and minor 0
[ 2783.119330] mydriver: open operation, major:511, minor:0
[ 2783.124777] mydriver: opened for read & write
[ 2783.129196] mydriver: at0
[ 2783.131866] mydriver: write operation, wrote=13
[ 2783.136448] mydriver: release operation
[ 2795.284020] mydriver: open operation, major:511, minor:0
[ 2795.289423] mydriver: opened for read & write
[ 2795.294013] mydriver: read operation, read=10000
[ 2795.298939] mydriver: read operation, read=0
[ 2795.303361] mydriver: release operation
```

Exercice 3

Ici, il faut étendre le pilote précédent pour qu'on puisse spécifier le nombre d'instances du pilote à créer lors de l'installation du module.

Comme dans l'exercice 2 sur les modules noyaux, nous devons utiliser un paramètre du module pour spécifier le nombre d'instances du pilote.

Pour créer plusieurs instances du pilote, nous devons modifier l'appel à cdev_init() pour lui passer le nombre d'instances du pilote. Il faut également allouer dynamiquemenent un buffer par instance du pilote. Afin de pouvoir accéder au bon buffer lors des opérations read et write, nous devons passer le bon buffer dans le pointeur file_operations->private_data durant l'ouverture du fichier (open). Il suffit ensuite d'utiliser ce pointeur dans les fonctions read et write.

Exercice 4

Maintenant que nous avons un pilote orienté caractère, nous allons développer une petite application qui utilise ce pilote.

Ici le code source de l'application:

```c
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>
#include <unistd.h>

static const char* text =
    "\n"
    "bonjour le monde\n"
    "ce mois d'octobre est plutot humide...\n"
    "ce n'est qu'un petit texte de test...\n";

static const char* text2 =
    "\n"
    "et voici un complement au premier text..\n"
    "ce n'est qu'un deuxieme petit texte de test...\n";

static const char* blabla =
    "blabla blabla blabla blabla blabla blabla blabla\n";

int main(int argc, char* argv[])
{
    if (argc <= 1) return 0;

    int fdw = open(argv[1], O_RDWR);
    write(fdw, argv[1], strlen(argv[1]));
    write(fdw, text, strlen(text));
    write(fdw, text2, strlen(text2));

    int s;
    do {
        s = write(fdw, blabla, strlen(blabla));
    } while (s >= 0);
    close(fdw);

    int fdr = open(argv[1], O_RDONLY);
    while (1) {
        char buff[100];
        ssize_t sz = read(fdr, buff, sizeof(buff) - 1);
        if (sz <= 0) break;
        buff[sizeof(buff) - 1] = 0;
        printf("%s", buff);
    }
    close(fdr);

    return 0;
}
```

Ce programme prend en paramètre le nom du fichier associé au pilote et écrit dans ce fichier puis lit le contenu du fichier. Il affiche finalement le contenu du fichier et répète l'opération pour chaque texte.

sysfs

Exercice 5

Dans cet exercice, il fallait créer un pilote orienté caractère qui valide les fonctionnalités de sysfs.

Pour ce faire, nous devons utiliser la fonction class_create() pour créer une classe de pilotes. Elle prend en paramètre THIS_MODULE et le nom de la classe. Cette fonction retourne un pointeur sur la classe créée.

Avec ce pointeur, nous pouvons créer un device avec la fonction device_create(). Elle prend en paramètre la classe et le nom du device. Cette fonction retourne un pointeur sur le device créé.

Finalement, pour installer le pilote dans sysfs, nous devons utiliser la fonction device_create_file(). Elle prend en paramètre le device et un pointeur sur la structure device_attribute, contenant le show et le store.

Pour supprimer le pilote de sysfs, nous devons utiliser la fonction device_remove_file() pour supprimer le fichier associé, la fonction device_destroy() pour supprimer le device et la fonction class_destroy() pour supprimer la classe.

Exercice 5.1

Ici, il fallait ajouter au module de l'exercice 5 les opérations read et write de l'exercice 3.

Exercice 7 - Opérations bloquantes

Dans cet exercice, le but était de créer une application utilisant les entrées et sorties bloquantes afin de signaler une interruption matérielle venant d'un switch.

Pour ce faire, nous utilisons une wait_queue et un flag pour signaler l'interruption. Cela permettra à la fonction d'interruption de générer un événement dans la wait_queue et de quittancer l'interruption. Cette fonctionnalité nécessite l'opération poll() dans notre pilote, qui permettra d'accéder à des ressources de manière bloquante. Dans la fonction poll, nous utilisons la fonction poll_wait() avec notre wait_queue, et préparons le flag pour signaler que l'interruption n'a pas eu lieu.

Dans la fonction d'interruption, nous utilisons la fonction wake_up_interruptible() sur notre wait_queue pour signaler l'interruption. Nous utilisons la fonction wake_up_interruptible() car elle permet de quitter la fonction poll().