TP module noyau Linux

J.-M Friedt

18 mars 2018

Notre objectif dans cet exposé est de maîtriser la mise en œuvre des interfaces de communication entre l'utilisateur et le matériel au travers du noyau Linux. Bien que nous ayons vu que divers niveaux d'abstraction (`script.bin` ou `devicetree.dtb`) vise à cacher la description du matériel au développeur, nous allons ici nous placer du point de vue de l'électronicien qui connaît son architecture matérielle et désire y accéder au travers de Linux pour en fournir les fonctionnalités à l'utilisateur.

1 Pourquoi travailler au niveau du noyau?

Dans la hiérarchie des niveaux d'abstraction permettant à un programme en espace utilisateur d'atteindre la matériel, le noyau fait office de gestionnaire des ressources. Il s'assure ainsi que chaque tâche a le droit de s'exécuter (ordonnanceur – *scheduler*), que les ressources sont attribuées de façon cohérente, et que les messages venant du matériel sont remontés à tous les programmes utilisateurs qui en ont besoin.

Un module [1] est un bout de code qui vient se connecter au noyau afin d'en compléter les fonctionnalités. Initialement introduit avec le noyau 1.2 de Linux (Mars 1995), il évite de devoir recompiler tout le noyau pour y ajouter une fonctionnalité telle que le support d'un nouveau périphérique matériel. Il est souvent possible d'accéder aux resources matérielles depuis l'espace utilisateur, mais ce faisant nous perdons les garanties de cohérence d'accès aux ressources proposées par le noyau (interdiction de charger deux fois un même module noyau par exemple). Certaines fonctionnalités (DMA, interruptions) ne sont simplement pas accessibles en dehors de l'espace noyau.

Depuis le noyau linux, la sélection des modules s'obtient en marquant par `M` les éléments accessibles par `make menuconfig`. La compilation des modules s'achève par `make modules` suivi de `make modules_install` qui placera le résultat dans `/var/lib/modules`.

Depuis buildroot, la configuration s'obtient par `make linux-menuconfig` et `make` de buildroot compilera à la fois le noyau et ses modules, pour les placer sur l'images qui sera flashée sur carte SD.

Tout développeur d'interfaces matérielles se doit de comprendre le fonctionnement d'un module noyau pour fournir le support logiciel nécessaire aux utilisateurs qui ne désirent par connaître les subtilités de l'accès au matériel.

Deux grandes classes de modules noyau sont chargés soit de transférer des blocs de données (par exemple vers un disque dur), soit des octets individuels [1] (caractères). Nous nous intéresserons au second cas, plus simple, et surtout moins difficile d'accès au niveau du matériel nécessaire à la démonstration.

Toutes les communications se font au travers de pseudo-fichiers situés dans le répertoire `/dev`. Un fichier peut être ouvert, fermé, et nous pouvons y écrire ou lire des données. Une méthode additionnelle qui n'a pas son équivalent dans un accès aux fichiers est une configuration des transactions : `ioctl` (Input/Output Control).

2 Structure et compilation d'un module noyau

Le minimum nécessaire pour attacher un module au noyau Linux est une fonction d'initialisation et une fonction pour libérer les ressources. L'exemple ci-dessous permet de se familiariser d'une part avec l'arborescence du noyau Linux, et d'autre part avec la méthode de compilation au travers d'un `Makefile` qui fait appel aux sources du noyau exécuté sur la plateforme cible.

Un module noyau minimaliste est proposé dans le listing ci-desssous, qui nous permettra de nous familiariser avec sa structure, sa compilation, et son utilisation.

Listing 1 – Premier exemple de module noyau

```
1 #include <linux/module.h>        /* Needed by all modules */
2 #include <linux/kernel.h>        /* Needed for KERN_INFO */
3 #include <linux/init.h>          /* Needed for the macros */
4
5 static int __init hello_start(void){printk(KERN_INFO "Hello\n");return 0;}
6 static void __exit hello_end(void) {printk(KERN_INFO "Goodbye\n");}
```

1. https://appusajeev.wordpress.com/2011/06/18/writing-a-linux-character-device-driver/

```
7
8 module_init(hello_start);
9 module_exit(hello_end);
```

Le module noyau contient nécessairement un point d'entrée et un point de sortie. Lorsque le modulé est lié au noyau (`insmod mon_noyau.ko`), la méthode `init()` est appelée. Lorsque le module est retiré du noyau (`rmmod mon_noyau`), la méthode `exit` est appelée. Ces méthodes sont des macros vers les fonctions `init_module()` et `cleanup_module()` (voir `linux/init.h` dans les sources d'un noyau Linux).

Un module est forcément lié à une version particulière d'un noyau, et devra être recompilé lors des mises à jour de ce dernier. Buildroot fournit un environnement cohérent de développement contenant la *toolchain* de cross-compilation et les sources du noyau, que nous trouverons dans `output/build/linux-`*version* de buildroot. Dans notre cas, le noyau linux est obtenu par `git` et contient donc, comme extension, l'identifiant de hashage.

Dans un premier temps, nous nous assurons que le compilateur est dans le chemin de recherche des exécutables :

```
export PATH=$PATH:$HOME/.../buildroot/output/host/usr/bin/
```

puis nous compilons le module – supposé être nommé `mymod.c`[2] pour devenir `mymod.ko` – au moyen du Makefile suivant :

```
1 obj-m += mymod.o
2
3 all:
4     make ARCH=arm CROSS_COMPILE=arm-buildroot-linux-uclibcgnueabihf- -C \
5   /home/jmfriedt/buildroot/output/build/linux-[...] M=$(PWD) modules
6
7 clean:
8     rm mymod.o mymod.ko
```

La seule méthode importante est évidemment `all`, qui fournit en argument de `make` le préfixe du compilateur (ici `arm-buildroot-linux-uclibcgnueabihf-` puisque le compilateur généré par `buildroot` se nomme `arm-buildroot-linux-uclibcgnueabihf-gcc`), l'emplacement des sources du noyau, et l'architecture cible.

Un module noyau se compile non-seulement pour la plateforme embarquée, mais aussi potentiellement pour l'hôte. Pour connaître la version du noyau qui tourne sur le PC, `uname -a`. S'assurer de la disponibilité des sources du noyau ou au moins des entêtes associés, par exemple par le paquet `linux-headers-4.1.0-2-all` sous Debian GNU/Linux à la date de rédaction de ce document. Sur PC, le Makefile sera de la forme

```
obj-m += t.o
all:
	make -C /usr/src/linux-headers-4.1.0-2-686-pae M=$(PWD) modules
```

Nous observons ici la puissance de `buildroot` qui fournit un environnement cohérent de travail avec une chaîne de compilation (`CROSS_COMPILE=arm-buildroot-linux-uclibcgnueabihf-`) et une arborescence des sources du noyau (`../buildroot-2017.05.2/output/build/linux-4.4.78`) qui est sollicitée lors de la compilation du module. Le principe du `Makefile` est d'ajouter le nom du nouveau module en cours de développement (`mymod`) à la liste des modules du noyau, et lancer la compilation du noyau par `make`. Les autres objets du noyau étant déjà compilés, seul notre module sera ainsi généré.

⚠**Attention** : en cas d'utilisation d'une carte comportant un noyau autre que celui de son buildroot, il faut synchroniser sa configuration du noyau avec celle de la carte. Pour ce faire, on pourra récuperer le fichier de configuration de Linux dans `/proc/configs.gz`, le placer dans le fichier `.config` du noyau utilisé par buildroot, effectuer un `make linux-menuconfig` pour informer buildroot de la mise à jour, et finalement `make` de buildroot pour refabriquer l'arborescence du noyau. Une fois ces tâches effectuées, le module que nous compilerons sera compatible avec le noyau exécuté sur la carte.

Le module est lié au noyau par `insmod mymod.ko` et retiré par `rmmod mymod`. La liste des modules liés au noyau est affichée par `lsmod`. Les messages du noyau sont soit affichés sur la console (dans le cas de la Redpitaya, sur le port série) ou dans les *logs* du noyau visualisés par `dmesg` ou `cat /var/log/messages` (ou parfois `/var/log/syslog`).

Le résultat de la compilation est transféré vers la carte Redpitaya (NFS) pour chargement par `insmod mymod.ko`. Nous en validons le bon fonctionnement en observant les messages fournis par le noyau dans ses logs au moyen de `dmesg` :

```
# insmod mymod.ko
# dmesg | tail -1
[  249.879087] Hello
# rmmod mymod.ko
# dmesg | tail -1
[  256.571319] Goodbye
```

2. on se gardera de nommer ce module `kernel.c` : dans ce cas, la compilation se concluera bien, mais par la génération d'un objet qui n'est pas issu de notre code source mais d'un fichier présent dans l'arborescence du noyau !

Ces informations sont reproduites dans `/var/log/messages` que l'on pourra consulter continuellement par `tail -f`.

L'exemple est extrait des sources du noyau, et accessible à `modules/example/example.c`.

En cas d'échec de la compilation du module, il est fondamental de se remémorer qu'un module noyau est compilé pour une configuration d'un noyau. En cas de perte de cette configuration, il est toujours possible de la retrouver au moyen de `scripts/extract-ikconfig` dont la sortie sera sauvegardée dans un fichier `.config` permettant de régénérer un environnement de travail identique à celui proposé par le noyau en cours d'exécution.

Les capacités de communication de notre module avec le systeme sont cependant restreintes : nous allons les étendre dans le contexte d'un périphérique de caractères (*char device*).

3 Communication au travers de `/dev` : interaction avec l'utilisateur

Un module qui se contente de se charger et de se décharger ne présente que peu d'intérêt. Afin d'interagir avec l'utilisateur, nous devons fournir des méthodes de type lecture et écriture. Par convention, les pseudo fichiers permettant le lien entre un module noyau et un programme utilisateur se trouvent dans `/dev`. Ces fichiers se comportent comme des tuyaux reliant les deux espaces (noyau et utilisateur) dans lesquels circulent des caractères (opposés à des blocs de données dans le cas qui nous intéresse ici). La création d'un pseudo-fichier se fait par l'administrateur au moyen de la commande `mknod /dev/jmf c 90 0` avec les arguments qui indiquent qu'il s'agit d'un *character device* (`c`) d'identifiant majeur 90 (nature du périphérique) et d'identifiant mineur 0 (indice dans la liste de ce périphérique). Sur PC, on ajouter `-m 777` pour créer le nœud de communication avec les permissions de lecture et d'écriture pour tout utilisateur : `mknod /dev/jmf c 90 0 -m 777`.

⚠En cas de message de ressource occupée lors de l'insertion du module (erreur -16 – EBUSY), il se peut que le nombre majeur 90 soit déjà occupé : même s'il n'apparaît pas dans la liste de `/dev`, la liste des périphériques dans `/proc/devices` indiquera tous les major numbers occupés. Par exemple sur Redpitaya, le major number 90 est occupé par le pilote `mtd`. Sur PC, le major 99 est occupé par ppdev.

La structure de données qui définit quelle fonction est appelée lors de l'ouverture et la fermeture du tuyau (`open` et `close` en espace utilisateur) ainsi que l'écriture ou la lecture de données dans le tuyau (`write` et `read` respectivement depuis l'espace utilisateur) est nommée `file_operations`.

Dans cet exemple, une écriture dans `/dev/jmf` se traduit par un message dans les logs du noyau contenant la chaîne de caractères écrite.

```
1  // mknod /dev/jmf c 90 0
2  #include <linux/module.h>       /* Needed by all modules */
3  #include <linux/kernel.h>       /* Needed for KERN_INFO */
4  #include <linux/init.h>         /* Needed for the macros */
5  #include <linux/fs.h>           // define fops
6  #include <asm/uaccess.h>
7
8  static int dev_open(struct inode *inod,struct file *fil);
9  static ssize_t dev_read(struct file *fil,char *buff,size_t len,loff_t *off);
10 static ssize_t dev_write(struct file *fil,const char *buff,size_t len,loff_t *off);
11 static int dev_rls(struct inode *inod,struct file *fil);
12
13 char buf[15]="Hello␣read\n\0";
14
15 int hello_start(void); // declaration pour eviter les warnings
16 void hello_end(void);
17
18 static struct file_operations fops=
19 {.read=dev_read,
20  .open=dev_open,
21  .write=dev_write,
22  .release=dev_rls,
23 };
24
25 int hello_start()  // init_module(void)
26 {int t=register_chrdev(90,"jmf",&fops);  // major = 90
27  if (t<0) printk(KERN_ALERT "registration␣failed\n");
28     else printk(KERN_ALERT "registration␣success\n");
29  printk(KERN_INFO "Hello\n");
30  return t;
31 }
32
```

```
33 void hello_end() // cleanup_module(void)
34 {printk(KERN_INFO "Goodbye\n");
35  unregister_chrdev(90,"jmf");
36 }
37
38 static int dev_rls(struct inode *inod,struct file *fil)
39 {printk(KERN_ALERT "bye\n");return 0;}
40
41 static int dev_open(struct inode *inod,struct file *fil)
42 {printk(KERN_ALERT "open\n");return 0;}
43
44 static ssize_t dev_read(struct file *fil,char *buff,size_t len,loff_t *off)
45 {int readPos=0;
46  printk(KERN_ALERT "read\n");
47  while (len && (buf[readPos]!=0))
48    {put_user(buf[readPos],buff++);
49     readPos++;
50     len--;
51    }
52  return readPos;
53 }
54
55 static ssize_t dev_write(struct file *fil,const char *buff,size_t len,loff_t *off)
56 {int l=0,mylen;
57  printk(KERN_ALERT "write␣");
58  if (len>14) mylen=14; else mylen=len;
59  for (l=0;l<mylen;l++) get_user(buf[l],buff+l);
60  // Essayer avec vvv : kernel panic car acces en userspace
61  // for (l=0;l<mylen;l++) buf[l]=buff[l];
62  buf[mylen]=0;
63  printk(KERN_ALERT "%s␣\n",buf);
64  return len;
65 }
66
67 module_init(hello_start);
68 module_exit(hello_end);
```

nous solicitons ce pilote par

```
# echo "Hello" > /dev/jmf
[  258.408111] open
[  258.414953] write
[  258.416874] Hello
[  258.416874]
[  258.420911] bye
# dd if=/dev/jmf bs=10 count=1
[  282.774521] open
[  282.776557] read
Hello[  282.778454] bye

0+1 records in
0+1 records out
```

```
# echo "toto" > /dev/jmf [  290.922228] open

[  290.924908] write
[  290.926933] toto
[  290.926933]
[  290.930904] bye
# dd if=/dev/jmf bs=10 count=1
[  293.484720] open
[  293.486761] read
toto[  293.488659] bye

0+1 records in
0+1 records out
```

Cet exemple contient une modification au module proposé ci-dessus afin que le message écrit soit celui lu auparavant : implémenter cette modification.

Ce module implémente les communications entre espace utilisateur et espace noyau au travers du pseudo-fichier `/dev/jmf`. Un fichier dans le répertoire `/dev` n'est pas un vrai fichier mais simplement une passerelle permettant de faire transiter des données. Sa création s'obtient par `mknod /dev/jmf c 90 0` avec 90 le l'identifiant (nombre) majeur et 0 l'identifiant mineur de ce pseudo-fichier. L'identifiant majeur reste constant pour les pseudo-fichiers liés à un même type de périphérique, tandis que l'identifiant mineur est incrémenté pour chaque nouvelle instance de ce périphérique. Nous vérifions que le périphérique de communication est bien reconnu par le noyau en consultant `/proc/devices` qui contient l'entrée `jmf` avec le major number que nous lui avons associé (90, qui est libre comme nous le vérifions par `ls -l /dev/`). Cependant, sur la carte Olinuxino-A13micro, le répertoire `/dev/` est régénéré à chaque lancement (*reboot*) : il faudra donc prendre soin de re-créer l'entrée `/dev/jmf` à chaque réinitialisation de la carte.

4 La méthode `ioctl`

Une méthode fort peu élégante – notamment parce qu'elle brise le concept de "tout est fichier" d'unix, est l'ioctl. Cette méthode permet de compléter les fonctions qui ne sont pas accessibles par `read` et `write`, et est notamment utilisée pour la configuration d'un périphérique (par exemple, régler la vitesse d'échantillonnage d'une carte son).

On complètera par exemple le listing précédent par les fonctions

```
1  // static int dev_ioctl(struct inode *inod,struct file *fil, unsigned int cmd,
2  //       unsigned long arg); // pre-2.6.35
3  static long dev_ioctl(struct file *f, unsigned int cmd, unsigned long arg); // actuel
4
5  static struct file_operations fops=
6  {.read=dev_read,
7   .open=dev_open,
8   .unlocked_ioctl=dev_ioctl,
9   .write=dev_write,
10  .release=dev_rls,};
11
12 // static int dev_ioctl(struct inode *inod,struct file *fil, unsigned int cmd,
13 //     unsigned long arg) // pre-2.6.35
14 static long dev_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
15 {switch ( cmd )
16  {case 0: printk(KERN_ALERT "ioctl0");break;
17   case 1: printk(KERN_ALERT "ioctl1");break;
18   default:printk(KERN_ALERT "unknown␣ioctl");break;
19  }
20  return 0;
21 }
```

L'exécution de ce programme se traduit par la séquence suivante :

```
# lsmod
Module                  Size  Used by
g_ether                44895  0
# insmod mymod.ko
# ./ioctl
# tail /var/log/messages
Jan  1 00:11:45 buildroot kern.alert kernel: [  705.740703] registration success
Jan  1 00:11:45 buildroot kern.info kernel: [  705.745730] Hello
Jan  1 00:11:50 buildroot kern.alert kernel: [  710.805798] open
Jan  1 00:11:50 buildroot kern.alert kernel: [  710.808491] ioctl1
Jan  1 00:11:50 buildroot kern.alert kernel: [  710.818059] ioctl0
Jan  1 00:11:50 buildroot kern.alert kernel: [  710.820838] bye
```

D'un point de vue utilisateur, la fonction `ioctl()` s'appelle exactement comme `read()` ou `write()` mais prend 3 arguments : le descripteur de fichier, le numéro de l'ioctl (en cohérence avec la déclaration dans le noyau, ici 0 ou 1) et éventuellement un argument. La page 2 du manuel de `ioctl` est explicite sur ces points. Depuis le noyau, le passage d'argument [3] vers l'espace utilisateur s'obtient au moyen de `put_user()` pour une valeur scalaire ou `copy_to_user` pour un tableau (zone mémoire).

5 Passer de l'adressage virtuel à l'adressage physique

Bien que de nombreux systèmes embarqués ne s'encombrent pas de gestionnaire matériel de mémoire (*Memory Management Unit* – MMU), un système d'exploitation multitâche tel que Linux en fait pleinement usage et justifie donc l'utilisation d'un microcontrôleur suffisamment puissant pour être muni d'un tel périphérique. L'organisation de la mémoire, gérée par la MMU, n'est en principe pas du ressort du développeur, *sauf* lorsqu'il veut accéder à un emplacement physique connu, tel que par exemple les registres de configuration du matériel. Dans ce cas, il nous faut effectuer la convertion inverse, de l'espace d'adressage virtuel vers l'espace d'adressage réel. Au niveau de l'espace utilisateur, cette fonctionnalité est fournie par `mmap()`. En espace utilisateur, il s'agit de `ioremap()`.

L'électronicien ne connaît que la notion d'adresse physique d'un périphérique : il s'agit de l'adresse associée à chaque registre dans la documentation du microcontrôleur. L'informaticien ne connaît que la notion d'adresse virtuelle, telle que

3. `http://www.makelinux.net/ldd3/chp-6-sect-1`

manipulée par le système d'exploitation qui tourne au dessus de la MMU. Nous devons faire le lien entre ces deux mondes pour les faire communiquer.

L'instruction [1, chap.9] qui, au niveau du noyau, fait ce lien est `void *ioremap(unsigned long phys_addr, unsigned long size);` Nous allons illustrer son utilisation en complétant les méthodes appelées lors de l'insertion du module pour allumer une LED, et éteidre la LED lorsque le module est retiré de la mémoire. Cet exemple est purement académique puisqu'en pratique, le module `linux-*/drivers/gpio/gpio-sunxi.c` implémente ces fonctionnalités, et bien d'autres, pour cacher les subtilités du matériel au développeur.

Les ressources qui nous aident à comprendre l'organisation des registres de configuration des GPIOs sont décrites à `http://redpitaya.readthedocs.io/en/latest/developerGuide/125-14/extent.html` et la documentation incontournable – à défaut d'être digeste – `https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf`. L'adressage des registres liés aux GPIO est décrit dans le chapitre 14 du manuel de l'utilisateur.

Nous complétons le module noyau que nous avons rédigé jusqu'ici par

```
static void __iomem *jmf_gpio;   //int jmf_gpio;
#define IO_BASE2 0xe000a000

int hello_start()  // init_module(void)
{int delay = 1;
 unsigned int stat;
[... gestion horloge GPIO ...]

 if (request_mem_region(IO_BASE2,0x2E4,"GPIO test")==NULL)
     printk(KERN_ALERT "mem request failed");
 jmf_gpio = (void __iomem*)ioremap(IO_BASE2, 0x2E4);
 writel(1<<mio,jmf_gpio+0x204);
 writel(1<<mio,jmf_gpio+0x208);
 writel(1<<mio,jmf_gpio+0x40);

 return 0;
}

void hello_end() // cleanup_module(void)
{printk(KERN_INFO "Goodbye\n");
 release_mem_region(IO_BASE2, 0x2e4);
}
```

en pensant à exploiter les fichiers d'entête `linux/io.h` et `linux/ioport.h`. Ce code ne peut se comprendre qu'en consultant [2, p.1347] reproduit ci-dessous, avec la description des divers registres associés aux GPIO, et [2, p.1580] pour la gestion des horloges.

6 Ajouter un *timer* périodique

Le noyau fournit un accès à des *timers* pour cadencer des tâches. Nous nous servons de telles ressources pour faire clignoter la diode périodiquement et afficher un message dans les *logs* du noyau.

Afin d'indiquer que la carte Redpitaya est en vie, nous désirons représenter le battement de son cœur par une diode qui clignote périodiquement. Le noyau Linux fournit une méthode sous forme de *timer* [1, chap.7].

```
struct timer_list exp_timer;

static void do_something(unsigned long data)
{printk(KERN_ALERT "plop");
 jmf_stat=0x200-jmf_stat;
 writel(jmf_stat,(void*)jmf_gpio+36*6+0x10); // JMFXX
 mod_timer(&exp_timer, jiffies + HZ);
}

int hello_start()  // init_module(void)
{...
 init_timer_on_stack(&exp_timer);
 exp_timer.expires = jiffies + delay * HZ; // HZ specifies number of clock ticks/second
 exp_timer.data = 0;
 exp_timer.function = do_something;
 add_timer(&exp_timer);
...
```

```
18 }
19
20 void hello_end() // cleanup_module(void)
21 {...
22  del_timer(&exp_timer);
23 }
```

Nous constatons dans `dmesg | tail` que le timer est bien appelé périodiquement, toutes les secondes.

```
[ 5697.089643] registration success
[ 5697.093780] Hello
[ 5698.094083] plop
[ 5699.094146] plop
[ 5700.094227] plop
[ 5701.094286] plop
[ 5702.094348] plop
```

Observer la diode verte à coté du connecteur audio lorsque le module est chargé en mémoire.

7 Sémaphore, mutex et spinlock [3]

Un pilote qui fournit une méthode `read` communique continuellement des informations au processus en espace utilisateur qui les requiert (`cat < /dev/pilote`). Cette condition de fonctionnement n'est pas représentative d'une application pratique, dans laquelle un producteur met un certain temps à générer les données qui seront communiquées à l'espace utilisateur : la méthode `read` se comporte alors comme le consommateur de données. Dans cette architecture de producteur-consommateur, il faut d'une part un mécanisme pour bloquer `read` tant que les données n'ont pas été produites, et d'autre part un mécanisme pour garantir la cohérence des accès à la zone mémoire commune aux deux fonctions de production et de consommation. La première fonctionnalité est fournie par le sémaphore, la seconde par le mutex et son implémentation plus rapide (mais plus gourmande en ressources), le spinlock.

7.1 Sémaphore

Le sémaphore se comporte ici comme un compteur qui mémorise le nombre de fois que les données sont produites. Un consommateur qui cherche à obtenir une donnée décrémente le sémaphore : si le sémaphore est déjà à 0, le consommateur (ici la méthode `read`) est bloquée jusqu'à ce qu'un producteur ait incrémenté le sémaphore. Un sémaphore incrémenté plusieurs fois permettra au consommateur d'obtenir plusieurs données.

La définition des constantes et prototypes associés aux spinlocks se résument en

```
1 #include <linux/semaphore.h>
2 struct semaphore mysem;
3 sema_init(&mysem, 0);       // init the semaphore as empty
4 down (&mysem);
5 up (&mysem);
```

Simuler la production périodique de données en incrémentant un sémaphore dans le gestionnaire d'un *timer*. Bloquer la méthode `read` tant que les données n'ont pas été produites.

Que se passe-t-il lorsque nous effectuons une lecture sur le périphérique accessible par `/dev/` alors que le sémaphore s'est déjà incrémenté plusieurs fois ?

Parmi les 5 fonctions au cœur d'un pilote communiquant – init, exit, open, read, release – laquelle permet de ne produire des données que lorsque l'utilisateur en a besoin ? Proposer un mécanisme pour ne produire des informations (incrément du sémaphore) que lorsque l'utilisateur en a besoin.

7.2 Mutex et spinlock

Les données produites sont stockées dans une mémoire tampon servant à échanger les informations entre producteur et consommateur. Un mécanisme doit être mis en œuvre pour garantir la cohérence entre les données produites et lues : il ne faut pas écraser une mémoire tampon en cours de lecture, et ne pas chercher à lire des données en cours de production. Le mutex (*MUTually EXclusive*) fournit un vérou binaire – bloqué ou débloqué – qui interdit à deux processus d'accéder simultanément à la même zone mémoire. Avant d'accéder au tampon, chaque processus (producteur ou consommateur) bloque le mutex, interdisant l'autre processus qui chercherait lui même à bloquer le mutex de continuer son exécution. Une fois l'opération – lecture ou écriture – achevée, le mutex est débloqué et l'exécution du processus en attente se poursuit.

La méthode des mutex permet de placer la tâche en mode veille jusqu'à ce que le mutex soit débloqué. Passer la tâche en mode veille libère des ressources du processeur, mais nécessite un changement de contexte qui peut s'avérer lourd et long si les opérations de lecture et écriture nécessitent une faible lattence. Une alternative est le spinlock, qui effectue les mêmes opérations mais en maintenant la tâche active et en testant continuellement l'état du vérou : dans ce cas, le processeur est continuellement sollicité, mais la lattence est moindre et le changement de contexte ne s'opère plus.

La définition des constantes et prototypes associés aux mutex se résument en

```
#include <linux/mutex.h>
struct mutex mymutex;
mutex_init(&mymutex);
mutex_lock(&mymutex);
mutex_unlock(&mymutex);
```

La définition des constantes et prototypes associés aux spinlocks se résument en

```
#include <linux/spinlock.h>
static DEFINE_SPINLOCK(myspin);
spin_lock_init(&myspin);
spin_lock(&myspin);
spin_unlock(&myspin);
```

Proposer une implémentation par mutex garantissant la cohérence entre écriture et lecture. Démontrer en modifiant le tampon contenant le texte affiché lors de la requête à `read`, en y insérant le décompte d'un compteur qui s'incrémente à chaque appel du timer simulant la production de données.

Au lieu de modifier le contenu du tableau de caractères contenant le message à afficher dans le gestionnaire de *timer*, créer une tâche qui sera ordonnancée lorsque le *scheduler* en aura le temps : on placera les opérations coûteuses en ressources (mutex, manipulation du tampon) dans cette tâche.

Références

[1] J. Corbet, A. Rubini, & G. Kroah-Hartman, *Linux Device Drivers, 3rd Ed.*, O'Reilly, disponible à `http://lwn.net/Kernel/LDD3/`

[2] *Zynq-7000 All Programmable SoC Technical Reference Manual, rev. 6 Dec. 2017*, Xilinx (2017)

[3] C. Blaess, *Interactions entre espace utilisateur, noyau et matériel*, GNU/Linux Magazine France Hors Série 87 (Nov.–Déc. 2016)