Interactions entre espace utilisateur, noyau et matériel*

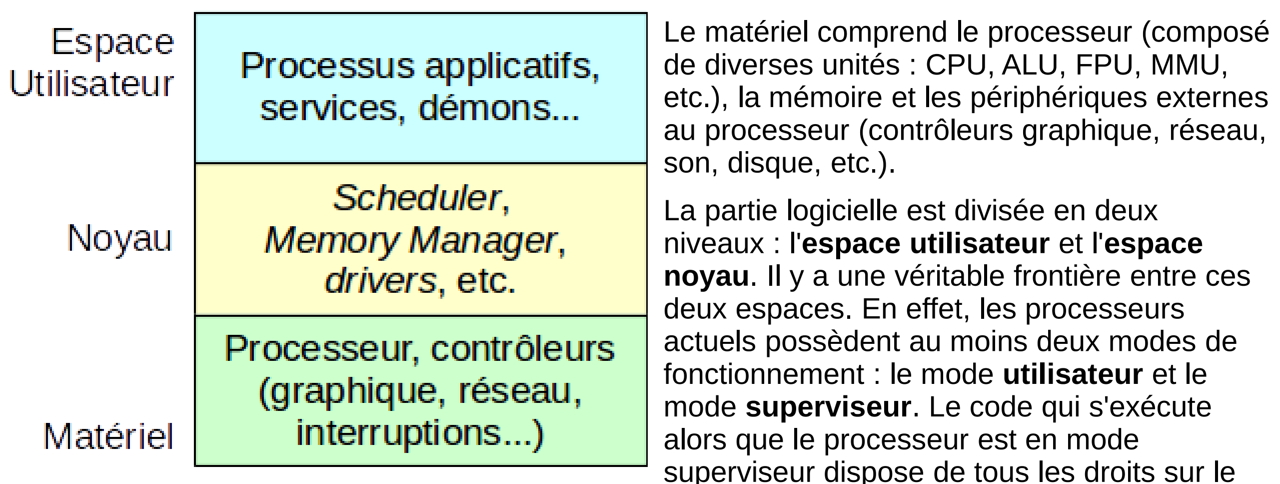
Christophe BLAESS

Pour pouvoir comprendre ou écrire du code kernel, il est important de bien assimiler les échanges d'informations entre les différents composants du système. Nous examinerons dans cet article les communications et notifications entre l'espace utilisateur (les applications), le noyau Linux et le matériel sous-jacent.

Lorsqu'on aborde la programmation dans le noyau Linux, il n'est pas rare de se sentir un peu submergé par l'ensemble des concepts présents dans ce nouvel environnement. Si l'adaptation d'un driver existant pour supporter un nouveau matériel est souvent assez facile, il n'en reste pas moins que pour écrire du code portable et robuste, certains mécanismes doivent être bien compris. Nous allons en voir quelques-uns dans cet article.

1 – Structure du système Linux

Le modèle des systèmes Gnu/Linux repose sur trois éléments distincts comme on le voit sur la figure 1.



Le matériel comprend le processeur (composé de diverses unités : CPU, ALU, FPU, MMU, etc.), la mémoire et les périphériques externes au processeur (contrôleurs graphique, réseau, son, disque, etc.).

La partie logicielle est divisée en deux niveaux : l'**espace utilisateur** et l'**espace noyau**. Il y a une véritable frontière entre ces deux espaces. En effet, les processeurs actuels possèdent au moins deux modes de fonctionnement : le mode **utilisateur** et le mode **superviseur**. Le code qui s'exécute alors que le processeur est en mode superviseur dispose de tous les droits sur le système. Il peut accéder directement au matériel, activer ou désactiver des interruptions, reconfigurer l'adressage mémoire via la MMU, etc. À l'inverse un code qui s'exécute en mode utilisateur n'a aucune de ces possibilités.

Le noyau Linux s'exécute en mode superviseur et tout le reste du système en mode utilisateur (y compris les commandes lancées avec `su` ou `sudo` qui n'ont rien à voir avec le mode superviseur du processeur). Nous avons donc deux mondes bien distincts, l'un totalement privilégié (celui du kernel) et l'autre complètement protégé, où aucune erreur de programmation n'aura de conséquence désastreuse sur le matériel ni sur le reste des applications.

Il y a trois manières de passer du mode utilisateur au mode noyau :

- Il existe une liste bien définie de points d'entrée que les programmes peuvent invoquer. Ce sont les **appels système** (par exemple : `open()`, `read()`, `write()`, `ioctl()`, `mmap()`, `fork()`, etc.) dont nous allons étudier le principe ci-après.

- Lorsqu'un périphérique doit notifier le système de l'occurrence d'un événement (par exemple

---

* Cet article est paru dans Gnu/Linux Magazine France, Hors-Série n° 87 « Kernel – le guide pour plonger au coeur de votre système Gnu/Linux », disponible sur `https://boutique.ed-diamond.com/les-guides/1088-gnulinux-magazine-hs-87.html`

l'arrivée d'un caractère sur un port série, la fin d'une lecture d'un bloc depuis le disque, la disponibilité d'une trame reçue sur un adaptateur Ethernet, etc.) il fait une demande d'**interruption**. Ce signal électronique indique au processeur d'arrêter son travail en cours, de sauvegarder son contexte d'exécution, et de se brancher à une adresse bien définie où se trouve un code capable de gérer l'événement survenu. Par la même occasion, le processeur bascule en mode superviseur, aussi toutes les fonctions de gestion – les *handlers* – d'interruptions se trouvent dans le noyau Linux.

- Quand un programme de l'espace utilisateur commet une erreur très grave (tentative de division par zéro, déréférencement d'un pointeur invalide, exécution d'un code assembleur illégal...), le processeur détecte directement le problème et notifie le système par un mécanisme proche de celui des interruptions que l'on nomme des **exceptions**. Les *handlers* d'exceptions sont automatiquement exécutés en mode superviseur, et se trouvent donc également dans le kernel. Leur réaction est d'envoyer au processus un signal (`SIGFPE`, `SIGSEGV`, `SIGILL`…) qui va généralement le tuer.

2 – Les appels système

2.1 – Entrée dans le kernel

Comment un appel système – une fonction implémentée dans la bibliothèque C, donc dans l'espace utilisateur – peut-il faire basculer le processeur en mode superviseur et exécuter du code dans le noyau ?

Examinons le fonctionnement de ce petit programme minimal qui invoque l'appel système `write()`.

hello.c :

```
#include <stdio.h>
#include <unistd.h>

int main(void)
{
	write(STDOUT_FILENO, "Hello\n", 6);
	return 0;
}
```

Nous compilons le fichier source et obtenons un exécutable :

```
$ gcc  hello.c  -o hello  -static
```

L'option `-static` est très importante. Elle demande à l'éditeur de liens d'inclure directement dans le fichier exécutable tout le code nécessaire à son fonctionnement, le rendant ainsi indépendant des bibliothèques dynamiques. Nous savons ainsi que tout le contenu de la fonction `write()` dans l'espace utilisateur est maintenant présent dans le fichier. Nous pouvons la désassembler et rechercher comment elle déclenche l'exécution de code dans le noyau. L'exemple a été tout d'abord compilé sur un processeur x86 64 bits.

```
$ objdump  -d  hello
[...]
000000000040105e <main>:
  40105e:       55                      push   %rbp
  40105f:       48 89 e5                mov    %rsp,%rbp
  401062:       ba 06 00 00 00          mov    $0x6,%edx
  401067:       be 04 35 49 00          mov    $0x493504,%esi
  40106c:       bf 01 00 00 00          mov    $0x1,%edi
  401071:       e8 1a 2e 03 00          callq  433e90 <__libc_write>
```

Inutile de connaître réellement l'assembleur x86 pour comprendre en substance ce morceau de code. Nous voyons que la fonction `main()` place les trois paramètres pour `write()` dans trois registres avant d'appeler la fonction `__libc_write()`. Comme nous avons compilé notre code avec l'option `-static`, cette dernière est également présente dans le fichier exécutable :

```
0000000000433e90 <__libc_write>:
  [...]
  433e99:       b8 01 00 00 00          mov    $0x1,%eax
```

```
 433e9e:       0f 05                   syscall
 433ea0:       48 3d 01 f0 ff ff       cmp    $0xfffffffffffff001,%rax
 433ea6:       0f 83 34 46 00 00       jae    4384e0 <__syscall_error>
 433eac:       c3                      retq
```

La fonction `__libc_write()` place la valeur `1` dans le registre `eax` et exécute l'instruction `syscall`. Cette instruction du processeur x86-64 sert à implémenter les points d'entrée du système d'exploitation. Elle va basculer le processeur en mode superviseur et sauter à une adresse donnée où l'on utilisera le contenu du registre `eax` pour sélectionner l'appel système à exécuter. En retour le registre `rax` contient une valeur négative en cas d'erreur.

En réitérant l'expérience sur un processeur Arm on observe :

```
0001f0f0 <__libc_write>:
   [...]
   1f104:       e3a07004        mov      r7, #4
   1f108:       ef000000        svc      0x00000000
```

Ici c'est le registre `r7` qui reçoit le numéro de l'appel-système – variable selon les architectures qu'il s'agit maintenant de `4` alors que sur x86-64 nous voyions `1`. L'instruction de déclenchement est `svc` (*service call*).

Sur un processeur x86 32 bits, l'opération est un peu plus compliquée, car il y a une indirection supplémentaire. La fonction `__libc_write` contient :

```
 806ce4b:       8b 54 24 10             mov    0x10(%esp),%edx
 806ce4f:       8b 4c 24 0c             mov    0xc(%esp),%ecx
 806ce53:       8b 5c 24 08             mov    0x8(%esp),%ebx
 806ce57:       b8 04 00 00 00          mov    $0x4,%eax
 806ce5c:       ff 15 f0 a9 0e 08       call   *0x80ea9f0
```

La dernière instruction signifie que l'on invoque une sous-routine dont l'adresse est elle-même stockée en mémoire en `0x080ea9f0`. Si on cherche le contenu de cet emplacement mémoire :

```
$ objdump  -s  --start-address=0x080ea9f0   hello
 80ea9f0   f0ef0608 90ad0908 07000000 7f030000  ................
 80eaa00   03000000 02000000 00100000 f0790908  .............y..
 […]
```

Les quatre premiers octets contiennent l'adresse mémoire de la commande de passage en mode superviseur. Comme l'architecture est *little endian*, il faut inverser les octets et lire : `0x0806eff0`. Voyons ce qui se cache à cette adresse :

```
$ objdump  -D  --start-address=0x0806eff0  hello
0806eff0 <_dl_sysinfo_int80>:
 806eff0:      cd 80                 int    $0x80
 806eff2:      c3                    ret
```

Sur un PC 32 bits, c'est donc une interruption logicielle déclenchée par le CPU lui-même (instruction assembleur `int`), qui permet de passer en mode superviseur et d'exécuter l'appel-système dont le numéro est stocké dans le registre `eax`. Le même numéro d'interruption – `0x80` – est employé pour tous les appels système. Les lecteurs ayant utilisé MS-DOS dans les années 90 se souviendront peut-être de l'interruption logicielle `0x21` qui permettait en suivant le même principe d'accéder depuis un programme en assembleur aux services du système d'exploitation. Ce fonctionnement est principalement employé sur les anciens processeurs, car les plus récents utilisent de préférence un mécanisme proche de celui des 64 bits.

`syscall` sur processeur x86-64, `svc` sur processeur Arm ou `int 0x80` sur x86-32 ont le même objectif : passer en mode superviseur et se brancher sur une routine dans le noyau. Cette routine est codée en assembleur. Sur une architecture x86-64, il s'agit de `entry_SYSCALL_64` dans le fichier `linux-4.7/arch/x86/entry/entry_64.S`, de `entry_SYSENTER_32` dans `linux-4.7/arch/x86/entry/entry_32.S` pour les x86-32 bits et de `vector_swi` dans `linux-4.7/arch/arm/kernel/entry-common.S` pour les processeur ARM. Le principe consiste alors à chercher l'adresse de la routine implémentant l'appel système demandé dans une table (la `sys_call_table` définie dans `linux-4.7/arch/x86/entry/syscalls/syscall_32.tbl` ou `syscall_64.tbl` ou encore `linux-4.7/arch/arm/kernel/calls.S`), puis à invoquer cette fonction, par exemple :

```
call    *sys_call_table(, %rax, 8)
```

La valeur 8 ci-dessus correspond à la taille des éléments de la table sur cette architecture, 8 octets soit 64 bits. Voici un aperçu du fichier `linux-4.7/arch/x86/entry/syscalls/syscall_64.tbl` qui sert à construire cette table :

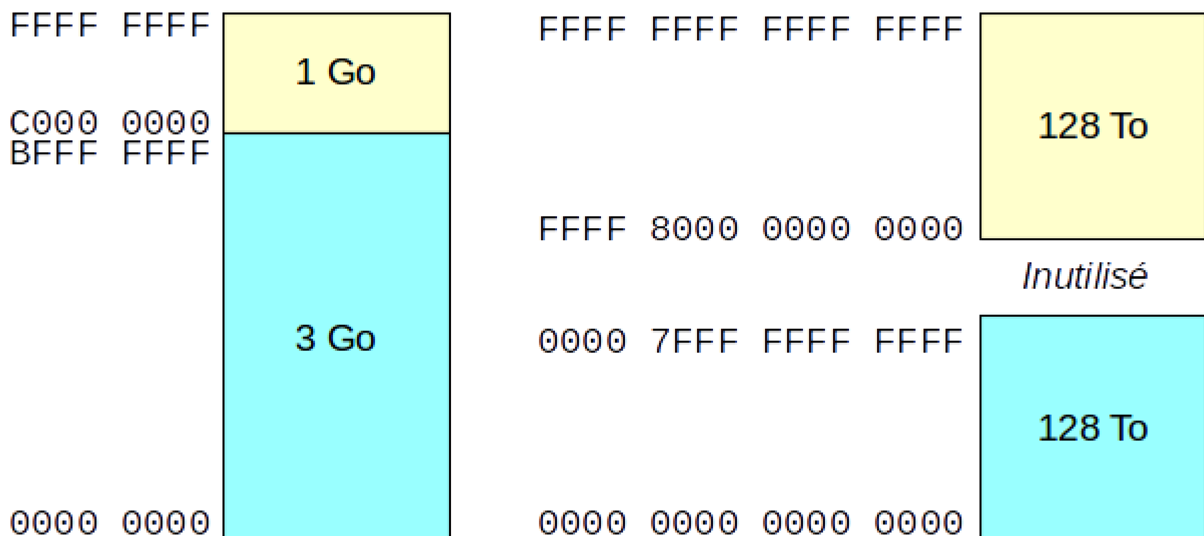
```
# The format is:
# <number> <abi> <name> <entry point>
#
# The abi is "common", "64" or "x32" for this file.
#
0       common  read                    sys_read
1       common  write                   sys_write
2       common  open                    sys_open
3       common  close                   sys_close
4       common  stat                    sys_newstat
5       common  fstat                   sys_newfstat
```

Nous voyons que le numéro de l'appel `write()` est bien `1` sur cette architecture comme nous l'avions observé en désassemblant l'appel système. La fonction exécutée sera `sys_write()` qui se trouve dans `linux-4.7/fs/read_write.c` et est commune à toutes les architectures.

Nous savons maintenant comment une application de l'espace utilisateur peut requérir l'aide du système en demandant l'exécution d'un appel système. Notons qu'il existe essentiellement deux types d'appels système : ceux qui offrent un service au processus appelant comme `getpid()`, `fork()`, `sigaction()`, `brk()`, etc. et ceux qui implémentent des méthodes associées à un descripteur comme `open()`, `close()`, `read()`, `write()`, `ioctl()`, `select()`, `mmap()`… Bien sûr il s'agit d'un descripteur pouvant représenter un fichier classique, mais également une socket, un périphérique caractère ou bloc, un tube de communication, etc.

2.2 – Communications avec l'espace utilisateur

L'espace d'adressage virtuel d'un processus s'étend classiquement de `0000 0000` à `BFFF FFFF` sur un processeur 32 bits et de `0000 0000 0000 0000` à `0000 7FFF FFFF FFFF` sur un processeur 64 bits. À l'intérieur de cet espace la MMU (*Memory Management Unit*) projette des pages de mémoire en assurant la conversion vers les adresses physiques.



L'espace réservé au noyau se situe généralement entre `C000 0000` et `FFFF FFFF` sur une architecture 32 bits, et entre `FFFF 8000 0000 0000` à `FFFF FFFF FFFF FFFF` sur 64 bits (on peut remarquer que les processeur 64 bits actuels ne gèrent « que » 256 To sur les 16 Eo qu'ils pourraient couvrir en théorie).

Un processus ne peut jamais accéder à la mémoire du kernel, car la MMU protège les adresses de ce dernier en les marquant comme accessibles en mode superviseur seulement. À l'inverse, pour des raisons de portabilité, un driver se trouvant dans le noyau **ne doit jamais accéder directement** à la mémoire d'un processus !

Pour échanger des données entre appel système et espace utilisateur deux possibilités s'offrent à nous :

- une **copie** des données avec les fonctions spécialisées `copy_to_user()` et `copy_from_user()` comme on le fait traditionnellement dans les appels système `read()`, `write()`, `ioctl()` par exemple,

- une **projection** directe des pages de mémoire du noyau dans l'espace d'adressage du processus appelant à l'aide de la fonction `remap_pfn_range()` ainsi que l'utilise l'appel système `mmap()`.

Voici un exemple de code noyau qui propose un appel-système `read()` pour l'espace utilisateur. Ce module extrêmement simple installe un simili-driver de type caractère, dont le fichier spécial apparaît automatiquement sous le nom `/dev/exemple-01`. Une lecture de ce fichier nous renvoie un message de salutation personnalisé (incluant le *PID* et le nom de la commande appelante).

exemple-01.c :

```
  #include <asm/uaccess.h>
 #include <linux/fs.h>
  #include <linux/miscdevice.h>
  #include <linux/module.h>
  #include <linux/sched.h>

static ssize_t ex_read(struct file *filp, char *u_buffer,
                            size_t max_lg, loff_t *offset)
{
  int lg;
  char k_msg[128];

 // Remplir une chaîne de salutation avec le PID de l'appelant.
  snprintf(k_msg, 128, "Hello '%s/%u'!\n", current->comm, current->pid);

 // Calculer la longueur restant à renvoyer.
  lg = strlen(k_msg) - (*offset);
  if (lg <= 0)
    return 0;
 // Tronquer si nécessaire (si le buffer fourni est trop petit).
  if (lg > max_lg)
    lg = max_lg;
 // Copier le message dans le buffer de l'espace utilisateur.
  if (copy_to_user(u_buffer, &k_msg[*offset], lg) != 0)
    return -EFAULT;
  *offset += lg;
  return lg;
}

  static struct file_operations ex_fops = {
    .owner   =  THIS_MODULE,
    .read    =  ex_read,
  };

  static struct miscdevice ex_misc = {
        .minor          = MISC_DYNAMIC_MINOR,
        .name           = THIS_MODULE->name,
        .fops           = & ex_fops,
  };

static int __init ex_init (void)
{
 // Initialiser un driver caractère de classe Misc.
  return misc_register(& ex_misc);
}

static void __exit ex_exit (void)
{
  misc_deregister(& ex_misc);
}

  module_init(ex_init);
  module_exit(ex_exit);
[...]
```

Après chargement du module, on peut consulter avec la commande `cat` le fichier spécial pour voir notre message :

```
[HS-12]$ sudo insmod exemple-01.ko
[HS-12]$ sudo cat /dev/exemple_01
Hello 'cat/16682'!
[HS-12]$ sudo cat /dev/exemple_01
Hello 'cat/16684'!
[HS-12]$ sudo rmmod exemple_01
```

L'inconvénient de ce mécanisme est de nécessiter une copie des données. Ceci prend du temps et est donc mal adapté aux communications avec un gros débit de données (flux vidéo par exemple). Dans ce cas on préfère projeter dans l'espace utilisateur des pages mémoire appartenant au driver. Ceci s'obtient avec la fonction `remap_pfn_range()` qui modifie la configuration de la MMU pour le processus appelant.

L'exemple suivant permet à un processus de projeter dans sa mémoire une page sur laquelle le kernel vient écrire toutes les secondes un message (le nombre de secondes écoulées depuis le 1er janvier 1970 et le complément en microsecondes).

Les exemples que je développe ici sont écrits en essayant de les rendre lisibles et compréhensibles. Ils ne suivent pas toujours les recommandations d'écriture du kernel (j'utilise plusieurs variables globales non-indispensables par exemple) et ne sont pas optimisés.

exemple-02.c :

```
[...]
  static char * ex_msg_string = NULL;
  static struct timer_list ex_timer;

static int ex_mmap (struct file * filp, struct vm_area_struct * vma)
{
 // Vérifier que la projection demandée ne soit pas trop grande.
  if ((unsigned long) (vma->vm_end - vma->vm_start) > PAGE_SIZE)
    return -EINVAL;
 // Réaliser la nouvelle projection
 return remap_pfn_range(vma,
                      (unsigned long) (vma->vm_start),
                      virt_to_phys(ex_msg_string) >> PAGE_SHIFT,
                      vma->vm_end - vma->vm_start,
                      vma->vm_page_prot);
}

static void ex_timer_function (unsigned long arg)
{
  struct timeval tv;

  // Lire l'heure et l'inscrire dans la page partagée.
  do_gettimeofday(& tv);
  snprintf(ex_msg_string, PAGE_SIZE, "\rTime: %ld.%06ld  ",
           tv.tv_sec, tv.tv_usec);
  // Reprogrammer le timer dans une seconde.
  mod_timer(& ex_timer, jiffies + HZ);
}

  static struct file_operations ex_fops = {
    .owner   =  THIS_MODULE,
    .mmap    =  ex_mmap,
  };

  static struct miscdevice ex_misc = {
        .minor          = MISC_DYNAMIC_MINOR,
        .name           = THIS_MODULE->name,
        .fops           = & ex_fops,
  };

static int __init ex_init (void)
{
  int err;

  ex_msg_string = kzalloc(PAGE_SIZE, GFP_KERNEL);
  if (! ex_msg_string)
    return -ENOMEM;
 // Le buffer est réservé par le noyau même s'il sera partagé
 // avec le processus appelant.
  SetPageReserved(virt_to_page(ex_msg_string));

 // Initialiser un timer à la seconde pour modifier le buffer.
```

```
  init_timer(& ex_timer);
  ex_timer.function = ex_timer_function;
  ex_timer.expires = jiffies + HZ;
  add_timer(& ex_timer);

 // Initialiser un driver caractère de classe Misc
  err = misc_register(& ex_misc);
  if (err != 0) {
    ClearPageReserved(virt_to_page(ex_msg_string));
    kfree(ex_msg_string);
  }
  return err;
}

static void __exit ex_exit (void)
{
  del_timer(& ex_timer);
  misc_deregister(& ex_misc);
  ClearPageReserved(virt_to_page(ex_msg_string));
  kfree(ex_msg_string);
}

[...]
```

Pour accéder au contenu de notre driver, il faut un processus invoquant l'appel système `mmap()`. Voici un petit programme qui réclame la projection et en affiche le contenu tous les dixièmes de secondes.

mmap.c :

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/mman.h>

int main(int argc, char *argv[])
{
  int fd;
  char *ptr;

  if (argc != 2) {
    fprintf(stderr, "usage: %s <device>\n", argv[0]);
    exit(1);
  }
  if ((fd = open(argv[1], O_RDONLY, 0)) < 0) {
    perror(argv[1]);
    exit(1);
  }
  ptr = mmap(NULL, 32, PROT_READ, MAP_SHARED, fd, 0);
  if (ptr == MAP_FAILED) {
    perror("mmap");
    exit(1);
  }
  while (1) {
    fprintf(stderr, "%s", ptr);
    usleep(100000);
  }
  return 0;
}
```

Lorsqu'on charge le module et qu'on lance le processus, on voit l'heure évoluer toutes les secondes, nous permettant ainsi de vérifier que cette page est bien partagée entre l'espace utilisateur et celui du noyau.

```
[HS-12]$ sudo insmod exemple-02.ko
[HS-12]$ sudo ./mmap /dev/exemple_02
Time: 1473605894.894373    ^C
[HS-12]$ sudo rmmod exemple_02
```

2.3 – Concurrence d'accès et synchronisation

Dans l'exemple précédent, nous avons deux accès simultanés à la même page mémoire. Ceci est fortement déconseillé car des incohérences peuvent se produire si deux écritures se produisent en

même temps ou si une écriture modifie l'ensemble des données alors qu'une lecture est en cours. Il existe plusieurs moyens de synchroniser les opérations, nous allons voir le plus répandu lorsque les deux accès se font depuis des appels système : les **mutex**. (*MUTual EXclusion*).

Les *mutex* du kernel sont très proche de ceux que l'on emploie en programmation multi-threads. On définit un *mutex* pour protéger un objet des accès concurrents et l'on se discipline ensuite à prendre le *mutex* avec `mutex_lock_interruptible()` avant tout accès et à le rendre ensuite avec `mutex_lock()`. Le mot *interruptible* au moment du verrouillage fait référence au type de sommeil dans lequel tombera le processus appelant si le *mutex* est déjà pris : un sommeil *interruptible* peut se terminer prématurément si un signal survient (par exemple généré par les touches Contrôle-C) auquel cas nous devons quitter notre appel système en renvoyant l'erreur `ERESTARTSYS`.

Dans l'exemple suivant nous allons construire une sorte de petite *fifo*, une table dans laquelle on peut venir stocker des données avec un appel système `write()` puis les extraire avec un appel `read()`.

exemple-03.c :

```
[...]
  #define MAX_MESSAGES 8
  static int msg_table[MAX_MESSAGES];
  static int nb_msg = 0;
  DEFINE_MUTEX(msg_mtx);

static ssize_t ex_read(struct file *filp, char *u_buffer,
                            size_t max_lg, loff_t *offset)
{
  int lg;
  char k_msg[128];

 // Verrouiller l'accès à la table de messages.
  if (mutex_lock_interruptible(&msg_mtx) != 0)
    return -ERESTARTSYS;
 // S'il n'y a aucun message, indiquer une fin de fichier.
  if (nb_msg == 0) {
    mutex_unlock(&msg_mtx);
    return 0;
  }
 // Lire le premier message, et décaler les autres.
  sprintf(k_msg, "%d\n", msg_table[0]);
  nb_msg --;
  if (nb_msg > 0)
    memmove(msg_table, &msg_table[1], nb_msg*sizeof(int));
  mutex_unlock(&msg_mtx);
 // Renvoyer le message lu si le buffer fourni est assez grand.
  lg = strlen(k_msg);
  if(lg > max_lg)
    return -ENOMEM;
  if (copy_to_user(u_buffer, k_msg, lg) != 0)
    return -EFAULT;
  return lg;
}

static ssize_t ex_write(struct file *filp, const char *u_buffer,
                            size_t lg, loff_t *offset)
{
  char * k_msg;
  int value;

 // Allouer un buffer local et y copier le message transmis.
  k_msg = kmalloc(lg, GFP_KERNEL);
  if (k_msg < 0)
    return -ENOMEM;
  if (copy_from_user(k_msg, u_buffer, lg) != 0) {
    kfree(k_msg);
    return -EFAULT;
  }
 // Lire une valeur numérique depuis le buffer.
  if (sscanf(k_msg, "%d", & value) != 1) {
    kfree(k_msg);
    return -EINVAL;
  }
  kfree(k_msg);
 // Verrouiler l'accès à la table de message
```

```
  if (mutex_lock_interruptible(&msg_mtx) != 0)
    return -ERESTARTSYS;
 // Si la table est pleine, renvoyer une erreur.
  if (nb_msg == MAX_MESSAGES) {
    mutex_unlock(&msg_mtx);
    return -EBUSY;
  }
 //  Ajouter le message et libérer l'accès à la table.
  msg_table[nb_msg] = value;
  nb_msg ++;
  mutex_unlock(&msg_mtx);
  return lg;
}
[...]
```

Notre table est très petite (huit entrées) pour que l'on atteigne facilement la limite en testant le module.

```
[HS-12]$ sudo  insmod  exemple-03.ko
```

Écrivons et relisons une valeur :

```
[HS-12]$ sudo  sh  -c  "echo 11 > /dev/exemple_03"
[HS-12]$ sudo  cat  /dev/exemple_03
11
```

Si on essaye de lire alors que la table est vide, l'opération se termine immédiatement :

```
[HS-12]$ sudo cat /dev/exemple_03
```

Si on essaye d'écrire alors que la table est pleine, c'est l'erreur *busy* qui se produit :

```
[HS-12]$ for i in $(seq 1 9); do sudo sh -c "echo $i > /dev/exemple_03"; done
sh: ligne 0 : echo: erreur d'écriture : Périphérique ou ressource occupé
[HS-12]$ sudo cat /dev/exemple_03
1
2
3
4
5
6
7
8
[HS-12]$ sudo rmmod exemple_03
```

2.4 – Sommeil et ordonnancement

Notre exemple précédent marchait bien, mais on peut lui reprocher un détail : une tentative de lecture alors que la file est vide se termine immédiatement en fin-de-fichier. De même une écriture dans une table pleine échoue en erreur. On pourrait préférer que ces opérations restent bloquantes jusqu'à ce qu'elles soient possibles.

Pour cela, on va utiliser une structure de données permettant d'endormir le processus appelant jusqu'à ce qu'on le réveille explicitement : une **waitqueue**. Une *waitqueue* rappelle un peu la variable-condition qu'on peut trouver dans l'API des threads Posix. Lorsque je présente la *waitqueue* lors de mes sessions de formation je la compare souvent à une cloche, à côté de laquelle il est possible de s'endormir et sur laquelle on peut donner un coup. Si personne ne dort au moment du coup sonné cela n'a pas d'importance, il n'est pas mémorisé (au contraire par exemple d'un sémaphore).

Dans notre module, nous ajouterons deux *waitqueues* : une pour endormir la lecture si la table est vide et une pour endormir l'écriture quand la table est déjà pleine. Symétriquement c'est l'opération d'écriture qui viendra donner un coup sur la *waitqueue* « table vide » et l'opération de lecture pour la *waitqueue* « table pleine ».

Je ne présente ci-dessous que les portions de code modifiées :

```
exemple-04.c :
[...]
  static int msg_table[MAX_MESSAGES];
  static int nb_msg = 0;
  DEFINE_MUTEX(msg_mtx);
  DECLARE_WAIT_QUEUE_HEAD(msg_tbl_full_wq);
```

```
  DECLARE_WAIT_QUEUE_HEAD(msg_tbl_empty_wq);

static ssize_t ex_read(struct file *filp, char *u_buffer,
                            size_t max_lg, loff_t *offset)
{
  [...]
 // Verrouiller l'accès à la table de messages.
  if (mutex_lock_interruptible(&msg_mtx) != 0)
    return -ERESTARTSYS;
 // Tant qu'il n'y a aucun message dormir en attente dans la waitqueue.
  while (nb_msg == 0) {
    mutex_unlock(&msg_mtx);
    err = wait_event_interruptible(msg_tbl_empty_wq, nb_msg!=0);
    if (err != 0)
      return -ERESTARTSYS;
    if (mutex_lock_interruptible(&msg_mtx)!= 0)
      return -ERESTARTSYS;
  }
  [...]
 // Notifier un éventuel écrivain bloqué.
  wake_up_interruptible(&msg_tbl_full_wq);
  [...]
}

static ssize_t ex_write(struct file *filp, const char *u_buffer,
                            size_t lg, loff_t *offset)
{
  [...]
 // Tant que la table est pleine dormir en attente dans la waitqueue.
  while (nb_msg == MAX_MESSAGES) {
    mutex_unlock(&msg_mtx);
    err = wait_event_interruptible(msg_tbl_full_wq, nb_msg!=MAX_MESSAGES);
    if (err != 0)
      return -ERESTARTSYS;
    if (mutex_lock_interruptible(&msg_mtx)!= 0)
      return -ERESTARTSYS;
  }
  [...]
 // Notifier un éventuel lecteur bloqué.
  wake_up_interruptible(&msg_tbl_empty_wq);
  [...]
}
[...]
```

Les différences d'exécution avec l'exemple précédent sont surtout dynamiques, il est difficile d'en donner un compte-rendu ici. Précisons que :

- une lecture lorsque la file est vide reste bloquée en attendant une nouvelle valeur (envoyée avec `echo` depuis un autre termina) ou la pression sur Contrôle-C pour arrêter le processus,

- l'écriture de valeurs successives bloque lorsque la table est pleine (au bout de huit valeurs) jusqu'à ce qu'on vienne lire un élément depuis un autre terminal ou que l'on presse Contrôle-C.

Ce mécanisme de mise en sommeil et réveil depuis un autre contexte est crucial pour le fonctionnement du système. Examinez le résultat d'une commande « `ps aux` », vous verrez des dizaines de processus endormis (état *Sleeping*) dans des *waitqueues*.

3 – Communication avec le matériel

Les exemples que nous avons vus précédemment ne communiquent qu'entre appels système et espace utilisateur. Il n'y a pas de dialogue direct avec le matériel sous-jacent. C'est pourtant le rôle essentiel de la plupart des drivers.

3.1 – Entrées / sorties vers les périphériques

Pendant longtemps, la communication avec les périphériques externes au processeur demandait des instructions assembleur spéciales (`inb`, `inw`, `inl`, `outb`, `outw`, `outl`...) qui agissaient sur des ports d'entrées-sorties dans un espace d'adressage totalement distinct de la mémoire. Sur la plupart des processeurs actuels c'est terminé, la MMU est capable de projeter dans l'espace

d'adressage du processeur les points d'entrée des périphériques externes (les registres) de manière totalement équivalente à la mémoire classique.

Pour dialoguer avec un matériel il nous faut d'abord connaître l'adresse physique de ce périphérique. Deux cas se présentent en général :

- soit il est accessible par un bus (par exemple *PCI*) permettant d'énumérer les périphériques connectés et de négocier leurs conditions d'accès, auquel cas l'adresse physique nous est directement transmise dans notre méthode `probe()`, invoquée lors de la détection du matériel ;

- soit l'adresse est indiquée lors de la configuration du matériel (par exemple dans le *BIOS* sur un PC ou dans le *device tree* sur un processeur Arm) et l'on peut la connaître grâce à une fonction d'interrogation comme `platform_get_resource()`.

Connaissant l'adresse physique de notre matériel, nous pouvons demander une projection dans l'espace d'adressage du noyau avec (voir `linux-4.7/include/asm-generic/io.h`) :

```
void __iomem *ioremap_nocache(phys_addr_t offset, size_t size);
```

L'adresse physique est passée en premier argument, suivie de la taille à projeter. L'adresse renvoyée se trouve dans l'espace d'adressage virtuel du kernel. On voit que le pointeur est marqué d'un attribut spécifique : `__iomem`. En effet, les registres auxquels on accède avec cette projection doivent être lus et écrits directement. Ceci signifie que l'on doit désactiver le cache du processeur sur ces zones d'adressage, pour éviter que deux lectures successives d'une adresse renvoient la même valeur (celle du cache) alors que l'état indiqué dans le registre du périphérique a changé. En outre on doit éviter que le compilateur lui-même se mêle de réordonner les accès pour faire des optimisations. Par exemple, il existe des périphériques où l'on doit écrire successivement le poids fort puis le poids faible d'une valeur à la même adresse pour la transmettre complètement. Il serait malvenu que le compilateur supprime la première écriture au prétexte que d'après lui la seconde suffira. Pour cela on évite les accès en lecture / écriture directs sur la plage de pointeurs renvoyés, en préférant les instructions suivantes qui agissent sur des pointeurs `__iomem`.

```
u8   ioread8   (void __iomem *addr);
u16  ioread16  (void __iomem *addr);
u32  ioread32  (void __iomem *addr);
void iowrite8  (u8 value, void __iomem *addr);
void iowrite16 (u16 value, void __iomem *addr);
void iowrite32 (u32 value, void __iomem *addr);
```

Outre l'accès en lecture ou écriture à l'adresse d'entrée-sortie, ces fonctions contiennent des barrières mémoires – instructions fictives qui imposent un point d'arrêt à l'optimiseur du compilateur – assurant ainsi que les registres indiqués seront lus ou écrits aussi souvent que nécessaire.

On notera que la projection d'un périphérique dans la mémoire du kernel peut également s'accompagner du re-projection dans l'espace utilisateur par l'intermédiaire du `mmap()` vu plus haut.

3.2 - Notification par interruption

Nous avons rapidement évoqué le principe des interruptions plus haut. Lorsqu'un périphérique considère qu'il a une information urgente à transmettre au système, il lève un signal électrique à destination du contrôleur d'interruption (APIC *Advanced Programmable Interrupt Controller*). Ce dernier envoie une demande d'interruption (IRQ *Interrupt Request*) au processeur. Celui-ci sauvegarde son contexte de travail (registres, pile…) et se branche sur une adresse dépendant du numéro d'interruption pour y exécuter un gestionnaire (*handler*) avant de reprendre son activité précédente.

Le *handler* de bas-niveau est écrit dans le noyau Linux standard et n'est généralement pas modifié. Nous pouvons néanmoins facilement lui fournir une fonction de plus haut-niveau (ISR *Interrupt Service Routine*) qu'il invoquera. Si plusieurs ISR sont fournies, elles sont toutes appelées successivement.

La fonction (voir `linux-4.7/include/linux/interrupt.h`)

```
int request_irq (unsigned int irq, irq_handler_t handler,
                 unsigned long flags, const char *name, void *id)
```

permet d'installer une routine de service sur l'interruption dont le numéro est passé en premier argument. Notre routine doit avoir la forme

```
irqreturn_t  my_function (int irq, void *id)
```

Elle reçoit en argument le numéro de l'interruption ayant déclenché son invocation (pour le cas où la même routine servent à traiter plusieurs interruptions), suivi du pointeur générique que nous avons fourni en dernier argument de `request_irq()` et qui peut par exemple viser une structure personnalisée contenant les informations personnelles du driver. La fonction doit interroger son matériel pour déterminer si c'est bien lui qui a provoqué l'interruption auquel cas elle renverra `IRQ_HANDLED`. À l'inverse elle renverra `IRQ_NONE` si l'interruption a été déclenchée par un autre périphérique dont le driver sera invoqué par la suite.

Pendant le déroulement d'une routine de service d'interruption on se trouve dans un contexte particulier qui empêche certaines opérations : pas d'accès à l'espace utilisateur (aucun processus ne nous a appelé) et surtout pas de mise en sommeil (il n'y a pas de processus à endormir). Impossible donc d'utiliser des *mutex*.

On rencontre toutefois très souvent un schéma où la routine d'interruption reçoit des données (provenant par exemple d'un périphérique d'acquisition) et les stocke dans un buffer en attendant qu'un appel système `read()` vienne les collecter et les envoyer à l'applicatif se trouvant dans l'espace utilisateur. Il est donc indispensable de disposer d'un moyen d'empêcher les accès concurrents au buffer et aux variables servant à gérer celui-ci. Ce mécanisme existe et se nomme le **spinlock**. Assez similaire dans son usage au *mutex*, le *spinlock* assure une attente active sans sommeil. Ainsi les fonctions `spin_lock()` et `spin_unlock()` peuvent être invoquées depuis une routine d'interruption.

Dans un appel système, ce sont les fonctions `spin_lock_irqsave()` et `spin_unlock_irqrestore()` que l'on appellera afin de prendre le *spinlock* et de couper les interruptions sur le processeur appelant puis de restaurer les interruptions en relâchant le *spinlock*. Bien que ce mécanisme paraisse bizarre au premier abord, il faut savoir que c'est un moyen parfaitement adapté pour synchroniser appels système et routine d'interruption de manière parfaitement portable, que l'architecture soit uni-cœur ou multi-cœur.

Pour voir un exemple très simple de *handler* d'interruption, je reprends le principe de l'exemple précédent en le modifiant comme suit :

- la table ne contient plus des nombres mais des horodatages en secondes depuis le 01/01/1970 et microsecondes,

- l'appel système `write()` a disparu ainsi que la *waitqueue* « table pleine »,

- un *handler* d'interruption viendra ajouter son horodatage à chaque déclenchement et l'appel système `read()` renverra les horodatages vers l'espace utilisateur,

- le *mutex* a été remplacé par un *spinlock*.

exemple-05.c :

```
[...]
  #define MAX_TV 8
  static struct timeval tv_table[MAX_TV];
  static int nb_tv = 0;
  static spinlock_t tv_spl;
  DECLARE_WAIT_QUEUE_HEAD(tv_tbl_empty_wq);
 // Le numero d'interruption est un paramètre du module.
  static int tv_irq = 10;
  module_param_named(irq, tv_irq, int, 0600);

static ssize_t ex_read(struct file *filp, char *u_buffer,
                            size_t max_lg, loff_t *offset)
{
  unsigned long irqs;
  int   err;
  char  k_msg[128];
  int   lg;

 // Verrouiller l'accès à la table des horodatages.
```

```
  spin_lock_irqsave(&tv_spl, irqs);
 // Tant qu'il n'y a aucun horodatage dormir en attente dans la waitqueue.
  while (nb_tv == 0) {
    spin_unlock_irqrestore(&tv_spl, irqs);
    err = wait_event_interruptible(tv_tbl_empty_wq, nb_tv!=0);
    if (err != 0)
      return -ERESTARTSYS;
    spin_lock_irqsave(&tv_spl, irqs);
  }
 // Lire le premier message, et décaler les autres.
  sprintf(k_msg, "%ld.%06ld\n", tv_table[0].tv_sec, tv_table[0].tv_usec);
  nb_tv --;
  if (nb_tv > 0)
    memmove(tv_table, &tv_table[1], nb_tv*sizeof(struct timeval));
  spin_unlock_irqrestore(&tv_spl, irqs);
 // Renvoyer le message lu si le buffer fourni est assez grand.
  [...]
}

static irqreturn_t ex_irq_handler(int irq, void * id)
{
  spin_lock(&tv_spl);
  if (nb_tv < MAX_TV) {
    do_gettimeofday(&(tv_table[nb_tv]));
    nb_tv ++;
  }
  spin_unlock(&tv_spl);
 // Notifier un éventuel lecteur bloqué.
  wake_up_interruptible(&tv_tbl_empty_wq);
  return IRQ_HANDLED;
}

static int __init ex_init (void)
{
  int err;
  spin_lock_init(&tv_spl);
  err = request_irq(tv_irq, ex_irq_handler, IRQF_SHARED,
                    THIS_MODULE->name, THIS_MODULE->name);
  if (err != 0)
    return err;
  return misc_register(& ex_misc);
}

[...]
```

Au chargement de ce module, on peut préciser le numéro d'interruption à horodater. Je propose de se placer sur l'interruption « souris » du système. Pour déterminer celle-ci, le plus simple est d'exécuter :

```
[HS-12]$ watch -n 0,1 cat /proc/interrupts
```

et d'observer quel numéro évolue lorsque l'on clique (par exemple 17 sur mon poste). Le numéro est passé en argument sur la ligne de `insmod`.

```
[HS-12]$ sudo  insmod  exemple-05.ko  irq=17
[HS-12]$ sudo cat /dev/exemple_05
1473635000.673606
1473635001.680602
1473635004.715601
1473635005.598596
1473635005.686594
1473635005.918591
1473635006.006595
^C
[HS-12]$ sudo  rmmod  exemple-05.ko
[HS-12]$
```

Conclusion

Cet article est une courte introduction à la programmation noyau, et nous n'avons fait qu'effleurer les mécanismes mis en œuvre pour les entrées/sorties et le traitement des interruptions. Nous n'avons pas parlé des traitements différés (*bottom halves* et *threaded interrupt*) d'interruptions,

des autres appels système que `read()`, `write()` et `mmap()` (par exemple `ioctl()`, `select()`, etc.), non plus que de la détection des périphériques (méthodes `probe()` et `disconnect()` du driver) ni de la gestion de la mémoire du kernel (`kmalloc()`, `vmalloc()`, `kmap()`, etc.). J'espère avoir néanmoins fourni une petite boîte à outils permettant de lire assez facilement le code source des drivers du kernel, et de mieux comprendre les interactions entre les applications et le noyau.

Pour aller plus loin

Je conseille la lecture régulière du site LWN (*Linux Weekly News*), et plus particulièrement sa rubrique Kernel (`https://lwn.net/Kernel`). La consultation de la dernière édition est réservée aux abonnés mais les précédentes sont lisibles par tous. Naturellement je vous encourage à vous abonner pour soutenir l'excellent travail de documentation de ce site.

Pour apprendre à produire du code de bonne qualité pour le noyau Linux, il existe le projet **`Eudyptula Challenge`** (http://eudyptula-challenge.org/) : une fois inscrit – gratuitement – vous recevrez un exercice par mail, auquel vous devrez répondre pour passer au niveau suivant, etc. La qualité des exercices est excellente, mais ce projet est un peu victime de son succès, et il faut parfois attendre plusieurs semaines pour avoir la correction de notre réponse.

Enfin, je propose des sessions de formation professionnelle pour apprendre la programmation noyau sous Linux, n'hésitez pas à me contacter par mail.