Pilotes de périphériques

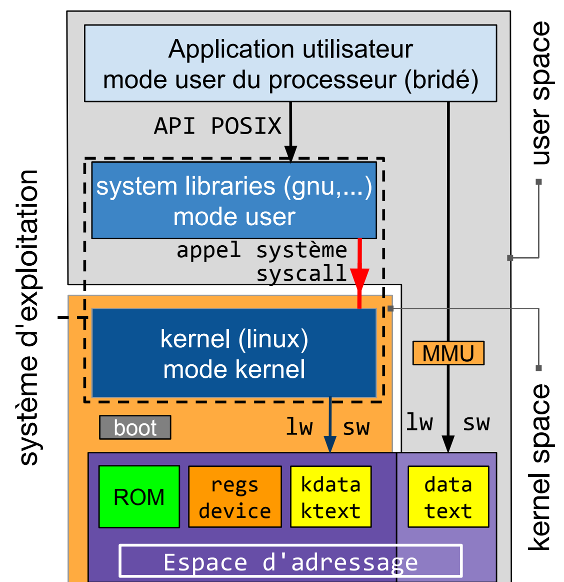
Module IOC — MU4IN109
Franck Wajsbürt

- http://doc.ubuntu-fr.org/tutoriel/tout_savoir_sur_les_modules_linux
- http://pficheux.free.fr/articles/lmf/drivers/
- https://broux.developpez.com/articles/c/driver-c-linux/

Système d'exploitation

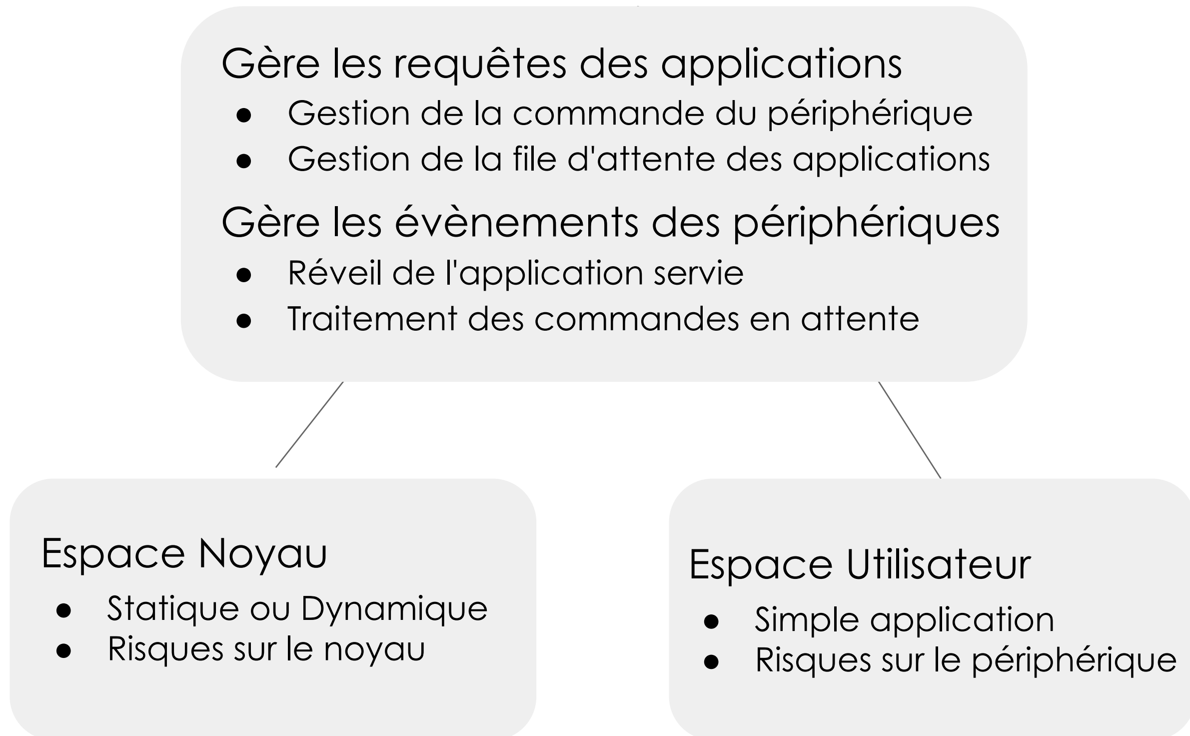
Le système d'exploitation est
un ensemble de programmes interfaçant
le matériel et les logiciels applicatifs.

- Fournit une abstraction du matériel,
  - Des bibliothèques système (ex: libc)
  - Un noyau pour gérer les ressources
  - Des pilotes de périphériques
- Permet un usage sûr du matériel
  - Vérifie la légalité des requêtes
- Permet le partage du matériel entre
  - Plusieurs applications
  - Plusieurs utilisateurs



RaspberryPi utilise une distribution de GNU/linux nommée Raspian

Pilotes de périphériques



Pilote de périphérique noyau

- Linux dispose d'un noyau
  - → monolithique les services sont entièrement traités par le noyau
  - → modulaire les services peuvent être ajouté à chaud
  - → orienté objet les services sont définis par des objets et des méthodes
- Les pilotes de périphériques sont des extensions du noyau
- Périphérique = Device : c'est le composant matériel
- Pilote = Driver : c'est le code des commandes
- Les pilotes contiennent du code spécifique au périphérique
- Orienté Objet : tous les pilotes implémentent la même API
  - → mais chaque pilote propose une implémentation différente

Modèle de pilote Linux

- Dans la suite "pilote" signifie "pilote de périphérique"
- Un pilote dans le noyau peut être statique
  → inclus dans le noyau Linux à la compilation
- Un pilote peut être dynamique dans un module
  → compilé comme du code relogeable
  → chargé et déchargé à l'exécution
  → plusieurs pilotes peuvent avoir des comportements différents pour la même API
- Les pilotes sont orientés objets dans Linux et donc
  → La structure de donnée décrivant le périphérique contient des pointeurs sur les fonctions de l'API driver

Sous-systèmes dans Linux

- Linux est composé d'un ensemble de sous-système
  → file system
  → network system
  → memory manager
  → device drivers
  → etc.
- Il y a 3 types de pilotes qui dépendent
  → *character* échange d'octets
  → *block* échange de bloc d'octets
  → *network* contrôle des périphériques réseau
- Nous allons juste voir les pilotes de type *character*

Pilote pour périphérique de type caractère

- nommé char device pour pilote de type caractère
- Utilisé pour les périphériques pour lesquels on échange caractère par caractère
- Ils sont accessibles dans la partition devfs dans le répertoire « `/dev` »
  → chaque pilote est associé à un fichier
  avec 2 numéros : Major & Minor

Char dev — Major — minor

```
crw-rw-rw-   1 root    root     1,   3 Apr 11  2002 null
crw-------   1 root    root    10,   1 Apr 11  2002 psaux
crw-------   1 root    root     4,   1 Oct 28 03:04 tty1
crw-rw-rw-   1 root    tty      4,  64 Apr 11  2002 ttys0
crw-rw----   1 root    uucp     4,  65 Apr 11  2002 ttyS1
crw--w----   1 vcsa    tty      7,   1 Apr 11  2002 vcs1
crw--w----   1 vcsa    tty      7, 129 Apr 11  2002 vcsa1
crw-rw-rw-   1 root    root     1,   5 Apr 11  2002 zero
```

- numéros
  - Major : identifie le type de device
  - Minor : identifie l'instance du device

Standardisation des numéros major/minor

Le fichier Documentation/devices.txt
contient la liste des numéros major et minor dans linux

```
  0             Unnamed devices (e.g. non-device mounts)
                  0 = reserved as null device number
                See block major 144, 145, 146 for expansion areas.

  1 char        Memory devices
                  1 = /dev/mem          Physical memory access
                  2 = /dev/kmem         Kernel virtual memory access
                  3 = /dev/null         Null device
                  4 = /dev/port         I/O port access
[...]

  1 block       RAM disk
                  0 = /dev/ram0         First RAM disk
                  1 = /dev/ram1         Second RAM disk
                    ...
                250 = /dev/initrd       Initial RAM disk

                Older kernels had /dev/ramdisk (1, 1) here.
                /dev/initrd refers to a RAM disk which was preloaded
                by the boot loader; newer kernels use /dev/ram0 for
                the initrd.

  2 char        Pseudo-TTY masters
                  0 = /dev/ptyp0 First PTY master
                  1 = /dev/ptyp1 Second PTY master
                    ...
                255 = /dev/ptyef 256th PTY master

[...]

  2 block       Floppy disks
                  0 = /dev/fd0          Controller 0, drive 0, autodetect
                  1 = /dev/fd1          Controller 0, drive 1, autodetect
```

```
 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

[...]

 10 char        Non-serial mice, misc features
                  0 = /dev/logibm       Logitech bus mouse
                  1 = /dev/psaux PS/2-style mouse port
                  2 = /dev/inportbm     Microsoft Inport bus mouse
                  3 = /dev/atibm ATI XL bus mouse
                ...

                240-254                 Reserved for local use
                255                     Reserved for MISC_DYNAMIC_MINOR

[...]

240-254 char    LOCAL/EXPERIMENTAL USE

240-254 block   LOCAL/EXPERIMENTAL USE
                Allocated for local/experimental use.  For devices not
                assigned official numbers, these ranges should be
                used in order to avoid conflicting with future assignments.
```

Concevoir un pilote de périphérique

- Définir les services attendus
  → uniquement les accès et pas les comportements
  → les comportements seront proposés dans la bibliothèque système (exécuté en mode utilisateur)
- Doit réutiliser le code des autres sous-systèmes
  → USB, interruptions, etc
- Le code doit être réentrant parce que le même pilote peut être utilisé par plusieurs instances du périphérique
  Il faut donc éviter l'usage des variables globales dans les fonctions
- Ecrire le code des opérations de l'API (open, read, …)
- Créer un module noyau pour ce pilote

Qu'est-ce qu'un Module du noyau ?

- Morceau de code permettant d'ajouter des services dans le noyau
  - pilotes de périphériques
  - appels systèmes
  - protocole réseau
- Un module est chargé dynamiquement dans le noyau sans recompilation du noyau et sans redémarrage
- Un module peut aussi être déchargé
- Un module s'exécute avec les droits du noyau
- Nous allons utiliser les modules pour l'ajout d'un pilote de périphérique

Connaître les modules installés

Pour connaître les modules installés : `lsmod`

```
pi@raspberrypi ~ $ lsmod
Module                  Size  Used by
snd_bcm2835            21342  0
snd_pcm                93100  1 snd_bcm2835
snd_seq                61097  0
snd_seq_device          7209  1 snd_seq
snd_timer              23007  2 snd_pcm,snd_seq
snd                    67211  5 snd_bcm2835,snd_timer,snd_pcm,...
i2c_bcm2708             6200  0
spi_bcm2708             6018  0
evdev                  11000  2
joydev                  9766  0
8192cu                569633  0
uio_pdrv_genirq         3666  0
uio                     9897  1 uio_pdrv_genirq
```

Les modules peuvent dépendre les uns des autres,
→ l'ordre de chargement est important

Avoir des informations des modules installés

Pour avoir des informations sur un modules : `modinfo`

```
pi@raspberrypi ~ $ modinfo i2c_bcm2708
filename:       /lib/modules/3.18.7+/kernel/drivers/i2c/busses/i2c-bcm2708.ko
alias:          platform:bcm2708_i2c
license:        GPL v2
author:         Chris Boot <bootc@bootc.net>
description:    BSC controller driver for Broadcom BCM2708
srcversion:     0ACD78A7932FB3F3042F78B
alias:          of:N*T*Cbrcm,bcm2708-i2c*
depends:
intree:         Y
vermagic:       3.18.7+ preempt mod_unload modversions ARMv6
parm:           baudrate:The I2C baudrate (uint)
parm:           combined:Use combined transactions (bool)
```

Il y a des informations sur les paramètres : leur nom, leur rôle et leur type

Charger / décharger un module

Commandes de chargement : `insmod` et `modprobe -a`

```
insmod <module> [module parameters]
    module est le cheminom (pathname) complet
modprobe -a  <module> [module parameters]
    Cette commande est plus intelligente, elle gère les dépendances
    et les alias pour ne pas avoir à taper le nom complet
```

Commandes de déchargement : `rmmod` et `modprobe -r`

```
rmmod <module>
modprobe -r <module>
```

La commande `dmesg`, qui affiche les messages du système, informe du travail réalisé.

Passer des paramètres aux modules

Les paramètres sont donnés au moment de leur chargement

```
pi@raspberrypi ~ $ modinfo i2c_bcm2708
filename:       /lib/modules/3.18.7+/kernel/drivers/i2c/busses/i2c-bcm2708.ko
[...]
parm:           baudrate:The I2C baudrate (uint)
[...]
pi@raspberrypi ~ $ sudo rmmod i2c_bcm2708

pi@raspberrypi ~ $ sudo insmod (écrit ici sur 3 lignes mais il n'y en a qu'une en vrai)
                     /lib/modules/3.18.7+/kernel/drivers/i2c/busses/i2c-bcm2708.ko
                     baudrate=100000
pi@raspberrypi ~ $ dmesg
[...]
[23865.018959] bcm2708_i2c_init_pinmode(1,2)
[23865.018997] bcm2708_i2c_init_pinmode(1,3)
[23865.022374] bcm2708_i2c 20804000.i2c: BSC1 Controller at 0x20804000 (irq 79)
                                                                 (baudrate 100000)
```

Charger des modules au boot

On peut demander le chargement des modules au moment du boot en mettant leur nom dans le fichier

`/etc/modules`

L'ordre est important, si le module A dépend du module B
alors B doit être chargé avant A

Écrire un module

```
#define MODULE
#include <linux/module.h>
#include <linux/init.h>
```

Contient les macros
définit dans les sources du noyaux

```
MODULE_AUTHOR("skyrunner");
MODULE_DESCRIPTION("exemple de module");
MODULE_SUPPORTED_DEVICE("none");
MODULE_LICENSE("none");
```

informations
récupérable par modinfo

```
static int __init mon_module_init(void)
{
    printk(KERN_DEBUG "Hello World !\n");
    return 0;
}
```

fonction appelée lors du
chargement du module
remplacer **`mon_module`**
par le nom du module

```
static void __exit mon_module_cleanup(void)
{
    printk(KERN_DEBUG "Goodbye World!\n");
}
```

fonction appelée au
déchargement du module
remplacer **`mon_module`**
par le nom du module

```
module_init(mon_module_init);
module_exit(mon_module_cleanup);
```

informe le noyau du nom
des fonctions de chargement
et de déchargement

compilation

Pour compiler, il faut les sources du noyau.

Il est déconseillé de compiler les modules directement sur la raspberrypi pour éviter de devoir copier les sources du noyau sur la SDcard, il est préférable de crosscompiler.

On compile hors des sources du noyau.

`Makefile → génère nom_module.ko`

```
# source du noyau
KERNELDIR=/users/enseig/franck/peri/linux-rpi-3.18.y

# prefix du cross-compilateur
CROSS_COMPILE ?= bcm2708hardfp-

# fichier objet contenu dans le module
obj-m += nom_module.o

# appel de make dans le répertoire KERNELDIR : target modules
make    -C $(KERNELDIR) ARCH=arm\
        CROSS_COMPILE=$(CROSS_COMPILE) M=$(PWD) modules
```


Passage des paramètres

Les paramètres d'un module sont de type :

- short (entier court, 2 octet),
- int (entier, 4 octets),
- long (entier long)
- charp (chaînes de caractères).

Ils sont déclarés dans le module et on informe le noyau par des macros

```
static type nom;
module_param(nom, type, permissions)
MODULE_PARM_DESC(nom, desc) → sera affiché avec modinfo
```

Exemple :

```
static int param;

module_param(param, int, 0); // 0 : ne pas exposer param dans sysfs
MODULE_PARM_DESC(param, "Un paramètre de ce module");

static int __init mon_module_init(void)
{
    printk(KERN_DEBUG "Hello World !\n");
    printk(KERN_DEBUG "param=%d !\n", param);
    return 0;
}
```

```
$ insmod ./module.ko param=2
```

Pilotes de périphérique

- Un pilote est un module qui permet d'échanger avec un périphérique.
- Il existe plusieurs types de pilote, nous nous intéressons à ceux de type caractère dont les échanges se font avec une granularité caractère.
- Un pilote doit d'abord être enregistré dans le noyau au chargement du module, puis devra être supprimé du noyau au déchargement.
- À l'enregistrement,
    - associer un numéro Major au pilote de ce périphérique, on peut choisir ce numéro, ou le laisser choisir par le système.
    - associer un liste de méthodes standards (open, read, write, ioctl,…) qui définissent le comportement du pilote
- On ajoute une entrée dans le système de fichier dans le répertoire /dev en utilisant le numéro Major qui permet de lui attribuer un nom afin que l'utilisateur puisse l'ouvrir (avec open) et l'utiliser (avec read, write, ioctl)

Enregistrement / Suppression d'un pilote

```
int register_chrdev( unsigned char major, const char *name,
                     struct file_operations *fops);
void unregister_chrdev(  unsigned int major, const char *name);
```

`register_chrdev`

- major : numéro Major du driver, `0` pour une affectation dynamique.
- name : nom du périphérique qui apparaîtra dans le fichier `/proc/devices` avec le numéro majeur
- fops : pointeur vers la structure des pointeurs de fonction. Ils définissent les fonctions appelées par les appels systèmes (read...) côté utilisateur.
- renvoient `0` ou `major` si tout se passe bien.

`unregister_chrdev`

- major : numéro majeur du driver, celui utilisé dans register_chrdev
- name : nom du périphérique utilisé dans register_chrdev

Opérations

Structure des fonctions de comportement du pilote

```
struct file_operations fops =
{
    .open     = my_open_function,
    .read     = my_read_function,
    .write    = my_write_function,
    .release  = my_release_function  /* appelée par le dernier close */
};
```

Les prototypes sont bien sûr imposés

Il en existe beaucoup d'autres :

- `ioctl()`, `select()`, `llseek()`, `aioread()`, `aiowrite()`, `readdir()`, `poll()`, `unlocked_ioctl()`, `compar_ioctl()`, `mmap()`, `flush()`, `fsync()`, `aio_fsync()`, `aio_fsync()`, `fasync()`, `lock()`, `sendpage()`, `get_unmapped_area()`, `check_flags()`, `dir_notify()`, `flock()`, `splice_write()`, `splice_read()`, `setlease()`, …
- Celles qui ne sont pas redéfinies sont initialisées avec un comportement par défaut qui rend `EINVAL`

Principales opérations

`open` Ouverture du périphérique. Cette méthode effectue le plus souvent la détection et l'initialisation du hardware lorsque cela est nécessaire.

`read` Lecture des données sur le périphérique (dans l'espace du noyau) puis déplacement des données dans l'espace utilisateur appelant.

`write` Ecriture des données depuis l'espace utilisateur vers l'espace noyau puis envoi des données au périphérique.

`close` Fermeture de l'accès. Cette méthode pourra exécuter des actions matérielles nécessaires à la fermeture du périphérique.

`unlocked_ioctl`
Paramétrage du périphérique depuis un programme utilisateur.

`select` mise en attente d'un programme utilisateur sur un ensemble de descripteurs de fichiers jusqu'à la réception de données ou la présence de place pour un write.

Implémentation des fonctions

```
static int
my_open_function(struct inode *inode, struct file *file) {
    printk(KERN_DEBUG "open()\n");
    return 0;
}

static ssize_t
my_read_function(struct file *file, char *buf, size_t count, loff_t *ppos) {
    printk(KERN_DEBUG "read()\n");
    return 0;
}

static ssize_t
my_write_function(struct file *file, const char *buf, size_t count, loff_t *ppos)
{
    printk(KERN_DEBUG "write()\n");
    return 0;
}

static int
my_release_function(struct inode *inode, struct file *file) {
    printk(KERN_DEBUG "close()\n");
    return 0;
}

long
my_unlocked_ioctl (struct file *, unsigned int, unsigned long); // vu plus tard
```


Type des arguments

`struct inode *inode` : nœud d'index contenant les métadonnées du fichier
File type (e.g., regular file, directory, device).
Owner, Group, Permissions, ...

`struct file *file` : structure sur le fichier ouvert
mode d'ouverture, offset de lecture/écriture, flags,
opérations (open(), read()...) private_data,,

`char *buf` : buffer d'échange dans l'espace utilisateur

`size_t count` : taille des données dans le buffer

`loff_t *ppos` : pointeur sur l'offset dans le fichier

open / release

En général, la méthode `open` réalise ces différentes opérations :

- Incrémentation du compteur d'utilisation
- Contrôle d'erreur au niveau matériel
- Initialisation du périphérique
- Identification du nombre mineur
- Allocation et remplissage de la structure privée file->private_data

Le rôle de la méthode `release` est tout simplement le contraire

- Libérer ce que open a alloué
- Éteindre le périphérique
- Décrémenter le compteur d'utilisation

création d'un noeud dans le /dev

mknod /dev/mydriver c major minor

Le numéro major est celui attribué lors de l'enregistrement,
si c'est le noyau qui l'a choisi, il se trouve dans /proc/device

```
pi@raspberrypi ~ $ cat /proc/devices
Character devices:
  1 mem
  4 /dev/vc/0
  4 tty
  5 /dev/tty
  5 /dev/console
  5 /dev/@ptmx
  5 ttyprintk
  [...]
```

Le numéro minor est un numéro d'instance.

L'effacement est un simple : `sudo rm`

Création d'un device "scripté"

L'insertion d'un module driver passe par un script bash qui

- charge le module driver
- recherche le major dans /proc/devices avec une commande awk
- créer le device

Le script insdev suppose que le nom du module est celui du driver

insdev

```
#!/bin/sh
module=$1
shift
/sbin/insmod ./$module.ko $* || exit 1
rm -f /dev/$module
major=$(awk "\$2==\"$module\" {print \$1}" /proc/devices)
mknod /dev/$module c $major 0
chmod 666 /dev/$module
```

$ sudo insdev device params…

destruction d'un device "scripté"

L'effacement d'un module driver passe par un script bash qui

- décharge le module driver
- efface le device

rmdev suppose que le nom du driver est le même que le nom du module

rmdev

```
#!/bin/sh
module=$1

/sbin/rmmod $module || exit 1
rm -f /dev/$module
```

```
$ sudo rmdev driver
```

Test d'un driver

```
echo "bonjour" > /dev/mydriver
```

→ permet d'invoquer **open, write, release**

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

→ permet d'invoquer **open, read, release**

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

int main(void)
{
     int file = open("/dev/mydriver", O_RDWR);

     if(file < 0) {
          perror("open");
          exit(errno);
     }
     write(file, "hello", 6);
     close(file);

     return 0;
}
```


TME

L'objectif du TME va être de contrôler
les leds et le bouton poussoir via un driver