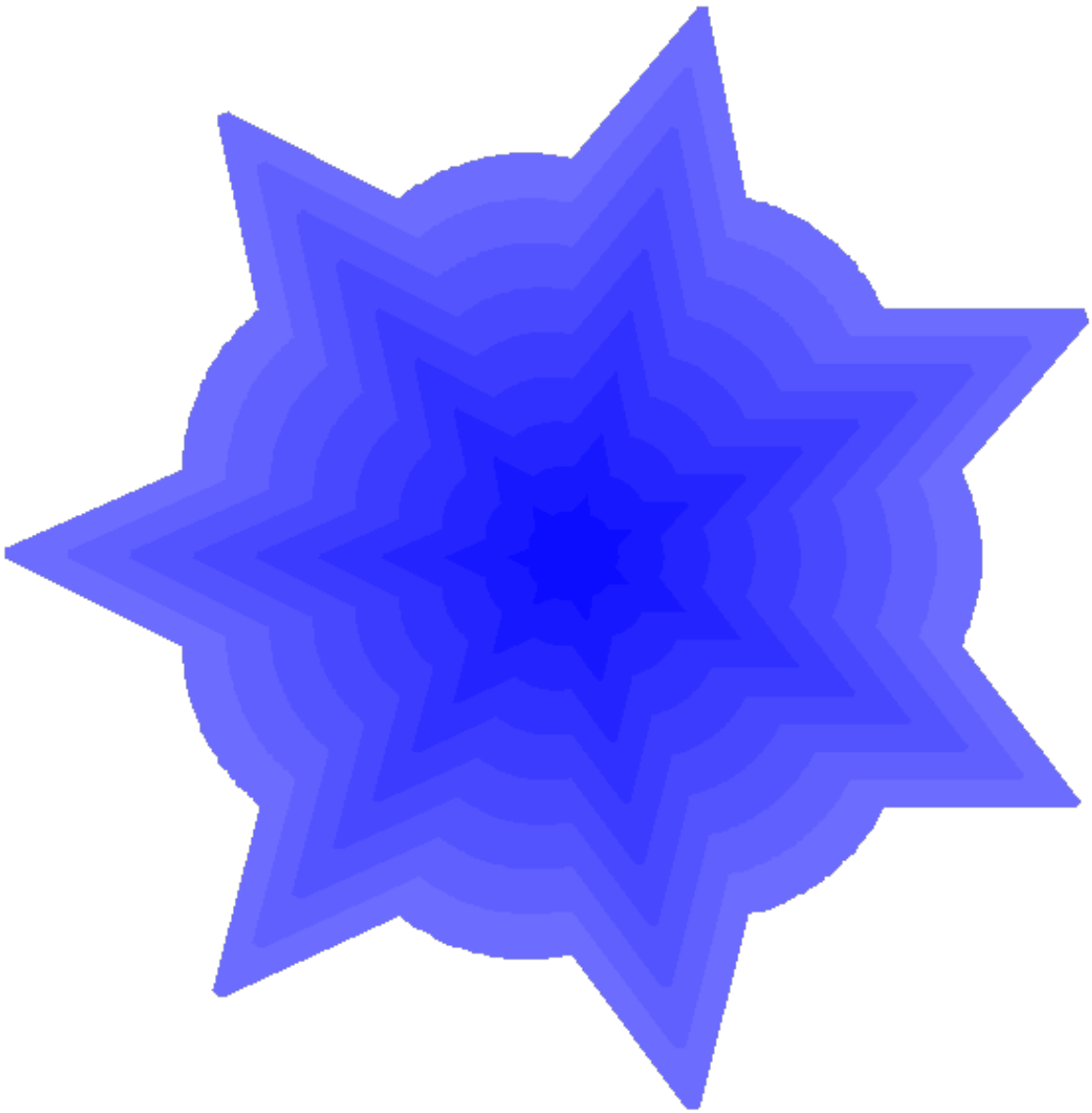
Débuter en kernel-programmation avec un driver en mode caractères



Débuter en kernel-programmation avec un driver en mode caractères (Bruggemann Eddie 2014).

Sommaire:

Débuter en kernel-programmation avec un driver en mode caractères (Bruggemann Eddie 2014).

Introduction:

Disposer des sources du noyaux :

Pour écrire du code pour le noyau, il faut disposer des sources de celui-ci car le noyau que utilise votre système Linux n'est rien d'autre qu'un gros fichier exécutable, lancer au boot, situer dans le dossier /boot.

Les sources de votre noyau actuel sont présentes dans le dossier du système :

```
/usr/src/linux-headers-$(uname -r)/
```

de votre système.

Celui-ci contient les dossiers suivants sous Ubuntu:

```
/arch                   :  Contient un dossier pour ce qui est spécifique a une architecture.
/block                  :  Driver en mode block.
/crypto                 :  Cryptage.
/Documentation          :  Documentation (Il faut la générer comme nous allons le voir).
/drivers                :  Drivers.
/firmware               :  Firmwares.
/fs                     :  Système de fichier.
/include                :  Fichiers d'en-tête a inclure pour la programmation noyau.
/init                   :  Initialisation du kernel.
/ipc                    :  IPC (Inter Process Communication).
/Kbuild                 :
/Kconfig                :
/kernel                 : Noyau du noyau.
/lib                    : Bibliothèques utilitaires.
/Makefile               : Fichier de compilation du noyau avec l'outil make.
/mm                     : Gestion de la mémoire (Memory Management).
/Module.symvers         :
/net                    : Résaux.
/samples                : Examples de tracepoints.
/scripts                : Scripts de génération du noyau.
/security               : Sécurité.
/sound                  : Audio.
/tools                  : Outils d'analyse de performances.
/ubuntu                 : Ubuntu.
/usr                    : User software pour l'initialisation du système de fichiers.
/virt                   : Virtualisation.
```

Pour générer la documentation sous Ubuntu :

Le paquet docbook-utils est requis.

Sous Ubuntu, la génération de la documentation n'est possible qu'avec les sources téléchargées avec la commande :

```
$ sudo apt-get source linux-image-$(uname -r)
```

Ensuite, il faut se placer dans le dossier des sources - il existe 3 commandes chacune pour un format de sortie différents afin de générer la documentation :

```
$ sudo make pdfdocs
$ sudo make psdocs
$ sudo make htmldocs
```

La documentation générer se trouve dans le format choisie dans le dossier :

```
/Documentation/DocBook/
```

Débuter en kernel-programmation avec un driver en mode caractères (Bruggemann Eddie 2014).

Spécificités du GNU C :

Le noyau n'est pas écrit en C ordinaire mais en GNU C qui est une version optimisée du langage C.

Si vous maîtrisez le langage C vous n'aurez aucun mal à assimiler les différences.

Il faut savoir qu'il ne vous sera pas possible de disposer des fonctions de la libc que vous connaissez comme dans un programme ordinaire, nie de compiler votre programme ni notre driver comme un programme ordinaire.

Mais le kernel vous met a disposition certaines fonctions analogues a celles de la libc pour certaines et bien d'autres qui sont spécifiques au noyau.

! Avec les changements de versions du noyau, les fonctions et les fichiers d'inclusion que vous utiliserait évoluent, la rétro-compatibilité est à essayer d'être maintenue mais il n'y a pas de garantie.

Notez que pour compiler notre driver, nous utiliseront l'outil **make** pour effectuer le linkage avec les sources du noyau.

Les spécificités du GNU C sont :

- Non utilisation des nombres a virgule flottante pour des questions de précisions :

  Car il faut savoir que selon l'architecture ceux-ci sont codés autrement, malgré le fait que le kernel fourni les types de données universelles pour la programmation noyau, mais les floats ne sont pas de la partie.

- Programmation orienter objet (P.O.O) :

  Les données et les fonctions doivent être strictement séparés. Les structures emploient beaucoup de pointeurs et sont définis comme propres au noyau dans un esprit de P.O.O., dont certaines sont écrits en langage d'assemblage : l'assembleur. Il existe notamment beaucoup de macros.

- Emploi de goto :

  Même si on ne les retrouvent pas souvent dans les programmes C actuels il sont légions dans l'écriture de driver car il permettent entre autres de sauter dans un bout de codes afin de dé-initialiser des données en cas d'erreur.

- Séparation de l'espace d'adressage :

  La mémoire est scindée en 2 espaces d'adressage distinct concernant les drivers et la programmation noyau :

  - Un espace d'adressage utilisateur (user-space).
  - Un espace d'adressage noyau (kernel-space).

  Nous effectueront les translations d'adresses nécessaire au transfert d'un espace d'adressage à un autre dans le cadre de l'application cliente du driver.

  D'ailleurs, ces deux espaces sont protégés l'un contre l'autre.

- Petites piles (stack) :

  Le noyau utilise des stack de 2 pages dont la taille dépend de l'architecture.

  - Architecture 64 bits : page de 8 K bytes * 2.
  - Architecture 32 bits : page de 4 K bytes * 2.

Débuter en kernel-programmation avec un driver en mode caractères (Bruggemann Eddie 2014).

- Importance de la synchronisation des données.
- Utilisation de fonction inline pour l'optimisation.
- Utilisation de l'assembleur de façon inline.
- Importance de la portabilité :

    Pour cela il existe un répertoire spécifique propre à chaque architecture prise en charge par le noyau Linux. Pour les choses variant d'une architecture à une autre, comme par exemple la définition des appels systèmes.

Débuter en kernel-programmation avec un driver en mode caractères (Bruggemann Eddie 2014).

Connaissance de l'outil make:

L'outil **make** permet l'automisation de la compilation d'un programme en permettant de définir les conditions sous lesquelles une commande sera lancée et de définir des variables et d'autre choses. Nous allons nous en servir ici afin d'effectuer le linkage vers les sources du noyau et générer le fichier module de notre driver.

! Il faut faire attention aux tabulations car make ne supporte pas les espaces remplaçant celles-ci.

Le fichier source est nommé driver.c et doit être situé dans un dossier, dont le nom ne porte pas a confusion Linux (évitez les caractères non-ASCII et espaces) et l'outil **make**, qui sera nommer driver.
Le fichier suivant doit être nommé : Makefile afin que la commande **make** reconnaisse notre fichier de compilation.

```
obj-m += driver.o
KERNELDIR ?= /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)
default:
	$(MAKE) -C $(KERNELDIR) M=$(PWD) modules
clean :
	$(MAKE) -C $(KERNELDIR) m=$(PWD) clean
```

Suite à quoi nous lançons la commande **make** sur le qui reconnait le fichier de compilation nommé Makefile:

```
$ cd driver
$ ls
driver.c   Makefile
$ sudo make
make -C /lib/modules/3.11.0-18-generic/build M=/home/edward/Bureau/driver modules
make[1]: entrant dans le répertoire « /usr/src/linux-headers-3.11.0-18-generic »
CC [M]  /home/edward/Bureau/driver/driver.o
Building modules, stage 2.
MODPOST 1 modules
CC      /home/edward/Bureau/driver/driver.mod.o
LD [M]  /home/edward/Bureau/driver/driver.ko
make[1]: quittant le répertoire « /usr/src/linux-headers-3.11.0-18-generic »
$ ls
driver.c   driver.mod.c  driver.o  modules.order
driver.ko  driver.mod.o  Makefile  Module.symvers
```

Débuter en kernel-programmation avec un driver en mode caractères (Bruggemann Eddie 2014).

Écriture et chargement d'un kernel-module:

Introduction a un kernel-module:

Dans ce premier bout de code (qui est un module chargeable à chaud) à l'opposé d'insérer le code dans le noyau lors de la compilation.

Nous allons utiliser une unique fonction du kernel qui permet d'écrire dans le fichier de log de celui-ci : /var/log/kern.log dont la fin est affichable avec la commande dmesg.

La fonction **printk** (print kernel) qui est analogue à la fonction printf() dont le flux de sortie est le fichier de log et qui a pour prototype :

```
int printk(const char *fmt, ...) ;
```

Dans ce fichier on peut insérer la nature du message dans les logs du noyau car il faut bien justifié la notification grâce au constante suivante :

```
#define KERN_EMERG      "<0>"           /* System unusable      */
#define KERN_ALERT      "<1>"           /* Need intervention    */
#define KERN_CRIT       "<2>"           /* System critic error  */
#define KERN_ERR        "<3>"           /* Sytem error occur    */
#define KERN_WARNING    "<4>"           /* Warning              */
#define KERN_NOTICE     "<5>"           /* Notice               */
#define KERN_INFO       "<6>"           /* Information          */
#define KERN_DEBUG      "<7>"           /* Debugging            */
```

De la manière suivante :

```
printk(KERN_DEBUG "Here is an message in the kernel logs\n") ;
```

A noter que selon la nature du message et la présence du caractère linefeed ('\n') l'écriture peut être différée.

Il existe aussi une fonction pour allouer dynamiquement de la mémoire **kmalloc**, a la manière de malloc, définis dans <linux/slab.h> qui a pour prototype :

```
void *kmalloc(size_t size, gfp_t flags);
```

Dont le premier argument définis le nombres de bytes a allouer limité a 128 K Bytes, pour de plus grand espace mémoire il faut se tourner vers la fonction vmalloc() qui alloue de la mémoire de façon virtuel sachant que l'espace d'adressage d'un module est limité a PAGE_SIZE fois 2.

Le deuxième argument induit le comportement de la fonction et dont les principale valeurs, définis dans <linux/mm.h>, sont :

```
GFP_USER    :   Alloue de la mémoire pour l'utilisateur.
                La fonction peut dormir.
GFP_KERNEL  :   Allocation de mémoire kernel normal
                en essayant de puiser dans la réserve de mémoire allouer au module.
                La fonction peut dormir.
FP_ATOMIC   :   Allocation sans dormir.
                Pour des fonctions ou cela n'est pas permis
                comme dans un interrupt handler.
```

La fonction retourne un pointeur sur l'espace mémoire allouer ou NULL en cas d'erreur.

L'espace mémoire allouer doit être libéré avec la fonction définis dans <linux/slab.h> :

```
void kfree(const void *)
```

Débuter en kernel-programmation avec un driver en mode caractères (Bruggemann Eddie 2014).

Structure d'un module:

Un module chargeable n'a pas de fonction main() mais un point d'entrée et de sortie correspondants à l'action d'accomplir quand on charge ou décharge le module.

Le point d'entrée d'un module peut se faire en définissant une fonction :

```
int init_module(void) {
            return 0 ; /* Le code de retour est important */
}
```

Et un point de sortie :

```
void cleanup_module(void) {
            return ;
}
```

Si l'on respecte strictement le nommage du point d'entrée et de sortie la fonction init_module sera appelée automatiquement lors du chargement du module, et la fonction cleanup_module au déchargement du module.

La fonction init_module sert à initialiser les ressources nécessaires au module, donc aussi à faire des testes pour l'initialisation ou l'enregistrement par le noyau d'un driver - par exemple : le sort du module dépend de son code de retour ; si celui-ci est différent de zéro le module ne sera pas chargé et la fonction cleanup_module sert a dé-initialiser les données et donc a nettoyer, d'où son nom.

Finalement, je précise qu'il est nécessaire de définir la licence du module afin que celui-ci soit chargeable par la macro:

```
MODULE_LICENSE() ;
```

qui accepte pour valeur définis dans <linux/module.h>:

```
"GPL"                           [GNU Public License v2 or later]
"GPL v2"                        [GNU Public License v2]
"GPL and additional rights"     [GNU Public License v2 rights and more]
"Dual BSD/GPL"                  [GNU Public License v2or BSD license choice]
"Dual MIT/GPL"                  [GNU Public License v2 or MIT license choice]
"Dual MPL/GPL"                  [GNU Public License v2 or Mozilla license choice]
"Proprietary"                   [Non free products]
```

Débuter en kernel-programmation avec un driver en mode caractères (Bruggemann Eddie 2014).

Écriture, compilation et insertion dans le noyau:

Voici le code source du fichier module1.c :

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

MODULE_LICENSE("GPL") ;
MODULE_DESCRIPTION("Module structure demonstration") ;
MODULE_AUTHOR("mrcyberfighter") ;

int init_module(void) {
  printk(KERN_DEBUG "Module inserted in kernel throught insmod !!!\n") ;
  return 0 ;
}

void cleanup_module(void) {
  printk(KERN_DEBUG "Module remove from kernel throught rmmod  !!!\n") ;
  return ;
}
```

Nous le compilons avec le fichier de compilation automatisé Makefile suivant :

```
obj-m += module1.o
KERNELDIR ?= /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)
default:
    $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
clean :
        $(MAKE) -C $(KERNELDIR) m=$(PWD) clean
```

```
$ sudo make
make -C /lib/modules/3.11.0-18-generic/build M=/home/edward/Bureau/dossier modules
make[1]: entrant dans le répertoire « /usr/src/linux-headers-3.11.0-18-generic »
CC [M]  /home/edward/Bureau/dossier/module1.o
Building modules, stage 2.
MODPOST 1 modules
CC      /home/edward/Bureau/dossier/module1.mod.o
LD [M]  /home/edward/Bureau/dossier/module1.ko
make[1]: quittant le répertoire « /usr/src/linux-headers-3.11.0-18-generic »
$ ls
Makefile   module1.ko     module1.mod.o  modules.order
module1.c  module1.mod.c  module1.o      Module.symvers
```

Puis nous procédons au chargement dans le noyau du module :

```
$ sudo insmod module1.ko
```

Vérifions si le module est présent dans le noyau :

```
$ lsmod
Module                  Size  Used by
module1                12426  0
...
```

Débuter en kernel-programmation avec un driver en mode caractères (Bruggemann Eddie 2014).

Regardons les métadonnées du module :

```
$ modinfo module1.ko
filename:       /home/edward/Bureau/module/module1.ko
author:         mrcyberfighter
description:    Module structure demonstration
license:        GPL
srcversion:     6941103565A089C4B575D10
depends:
vermagic:       3.11.0-18-generic SMP mod_unload modversions
```

Puis déchargeons le module :

```
$ sudo rmmod module1.ko
```

Et finalement regardons si le module a écrit dans les logs :

```
$ dmesg
...
[ 4264.823025] Module inserted in kernel throught insmod !!!
[ 4587.017096] Module remove from kernel throught rmmod  !!!
```

Débuter en kernel-programmation avec un driver en mode caractères (Bruggemann Eddie 2014).

Écriture d'un driver:

Introduction a un driver:

Il existe 2 types de drivers :

- Les drivers en mode caractères.
- Les drivers en mode bloc.

Qui sont soit chargeable a chaud dans le noyau grâce a la fonction insmod ou inséré dans le noyau a la compilation.

Un driver contrôle le périphérique. Dans notre cas nous souhaitons contrôler un périphérique virtuel, avec un driver en mode caractères. Il nous faudra donc créer un périphérique virtuel (un fichier spécial en mode caractères) dans le dossier /dev/.

Débuter en kernel-programmation avec un driver en mode caractères (Bruggemann Eddie 2014).

Fonctions nécesssaires:

Pour cela il faut procéder aux les étapes suivantes :

1. Attribuer un numéro majeur et mineur qui identifiera le **périphérique** que nous souhaitons contrôler:

   Le numéro peut être défini :

   de manière statique en demandant le numéro majeur que nous souhaitons, mais il faut savoir que le numéro majeur est codé sur un byte non-signé dans la plage de 0-255.

   Seuls les plages suivantes peuvent être utilisés:

   - 60 - 63.
   - 120 - 127
   - 240 - 255

   Le numéro ne doit pas être attribué déjà, vérifiez en lisant le fichier /proc/devices.

   ```
   $ cat /proc/devices
   ```

   Ou de manière dynamiquement en laissant le noyau réserver un numéro majeur garantie valide et non-attribuer.

2. Cela implique de réserver le numéro de **périphérique** en mode caractère.

   En réservant au même temps que l'allocation mémoire du périphérique avec la fonction :

   ```
   int alloc_chrdev_region(dev_t major,
                           unsigned int minor,
                           unsigned int minor_numbers,
                           const char *name) ;
   ```

   qui renvoie 0 en cas de succès.

3. Allouer de la mémoire pour instancier notre objet driver avec la fonction :

   Avec la fonction :

   ```
   void *cdev_alloc(void) ;
   ```

   qui renvoie un pointeur de type struct cdev sur l'objet périphérique en cas de succès sinon un pointeur NULL.

4. De lier l'objet périphérique avec une structure de type file_operations qui permet de prendre en charge les différentes actions que peut entreprendre le programme client du driver dont nous reparleront.

5. De lier l'objet driver au numéro référençant le périphérique par la fonction :

   ```
   int cdev_add(struct cdev *driver_object, dev_t dev_num, unsigned int major) ;
   ```

6. De créer une classe de notre périphérique avec la macro :

   ```
   class_create(owner, name)
   ```

7. De créer celui-ci finalement et de l'enregistrer dans le système de fichier /dev/ grâce a la la fonction :

```
struct device *device_create(struct class *cls,
                             struct device *parent,
                             dev_t devt,
                             void *drvdata,
                             const char *fmt,...)
```

8. De détruire les ressources crées en cas d'erreur et de retourner un code d'erreur grâce a un saut avec un goto lors d'un contrôle d'un code de retour :

    • Par la fonction :

```
void unregister_chrdev_region(dev_t dev_num,unsigned int minor_count) ;
```

    On restitue l'espace alloué pour notre périphérique et la référence au numéro majeur ayant minor_count numéros mineurs.

    • Par la fonction :

```
kobject_put(&driver_object->kobj) ;
```

    On restitue l'espace alloué à l'objet driver en décrémentant à zéro la référence dans le noyau.

    • Ont peut retourner un code d'erreur -EIO.

9. De détruire les ressources crées dans la fonction cleanup_module() :

    Grâce aux fonctions suivantes :

```
void device_destroy(struct class *cls, dev_t dev_num);
```

    Détruit le périphérique virtuel représenté par la classe du périphérqiue cls lier au numéro majeur dev_num.

```
void class_destroy(struct class *cls);
```

    Détruit la classe du périphérique virtuel.

```
void cdev_del(struct cdev *);
```

    Libère l'espace alloué à la structure du driver.

```
void unregister_chrdev_region(dev_t dev_num, unsigned int minor_count) ;
```

    Restitue l'espace alloué pour notre périphérique et la référence au numéro majeur ayant minor_count numéros mineurs.

Débuter en kernel-programmation avec un driver en mode caractères (Bruggemann Eddie 2014).

Écriture, compilation et insertion dans le noyau:

Assez de théorie - en pratique cela donne le code suivant :

```
#include <linux/module.h>  /** MODULE_LICENSE &&
                               MODULE_DESCRIPTION &&
                               MODULE_AUTHOR                */

#include <linux/cdev.h>    /** cdev_del() &&
                               cdev_add() &&
                               cdev_alloc()                 */

#include <linux/device.h>  /** class_create() &&
                               device_create()  &&
                               device_destroy()             */

#include <linux/fs.h>      /** alloc_chrdev_region() &&
                               unregister_chrdev_region()   */

MODULE_LICENSE("GPL") ;
MODULE_DESCRIPTION("Driver creating as module.") ;
MODULE_AUTHOR("mrcyberfighter") ;

#define DEVICE_NAME "devname"       /** Device name   */

static dev_t dev_num ;              /** Major number  */
static struct cdev *driver_object ; /** Driver object */
static struct class *device_class ; /** Device class  */


static int __init mod_init(void) {

    static int minor=0 ;
    static int minor_numbers=1 ;

    printk(KERN_DEBUG "start device creating\n") ;

    if (alloc_chrdev_region(&dev_num,minor,minor_numbers,DEVICE_NAME) < 0) {
      /** Allocate space for device and major number dev_num
          with minor number minor and minor_numbers count
          and as device name devname                        */

        printk(KERN_DEBUG "Device number and space alloctating error !!! \n") ;
        return -EIO ;
    }

    driver_object=cdev_alloc() ; /** Driver space allocating and
                                    instantiation as struct cdev object. */

    if (driver_object == NULL) {
      goto free_device_number ;
    }

    driver_object->owner=THIS_MODULE ;
```

Débuter en kernel-programmation avec un driver en mode caractères (Bruggemann Eddie 2014).

```
    if (cdev_add(driver_object,dev_num,1)) { /** register the driver object
                                               to the kernel
                                               with the device number dev_num
                                               and counts of minor numbers 1 */
        goto free_device ;
    }

    device_class=class_create(THIS_MODULE,DEVICE_NAME) ;
    /** create new device class               */

    device_create(device_class,NULL,dev_num,NULL,"%s",DEVICE_NAME) ;
    /** register device in /dev/  filesystem  */

    if (device_class == NULL) {
      printk(KERN_DEBUG "Device class creating error !!!\n") ;
      goto free_device ;
    }

    printk(KERN_DEBUG "Device successfull created !!!\n") ;
    return 0 ;

    /** error handling goto ; */
    free_device :
      printk(KERN_DEBUG  "Driver registering error or\n"
                         "Device class creation error.\n"
                         "Must free allocated space for device.\n") ;
      kobject_put(&driver_object->kobj) ;

    free_device_number :
      printk(KERN_DEBUG "chrdev registering error !!!\n") ;
      unregister_chrdev_region(dev_num,1) ;
      return -EIO ;
}

static void __exit mod_exit(void) {

  device_destroy(device_class,dev_num) ;
  /** destroy device */

  class_destroy(device_class) ;
  /** destroy device class */

  cdev_del(driver_object) ;
  /** free driver_object memory space */

  unregister_chrdev_region(dev_num,1) ;
  /** unregister dev_t dev_num and minors count 1 */

  printk(KERN_DEBUG "Device destruction succesfull !!!\n") ;
  return ;

}

module_init(mod_init) ; /** Register init_module macro.    */
module_exit(mod_exit) ; /** Register cleanup_module macro. */
```

Remarquer que nous n'avons pas utiliser la règle de nommage des fonctions d'initialisation (**init_module**) et de dé-initialisation (**cleanup_module**) et l'ajout d'un mot-clef aux fonctions d'initialisation et de dé-initialisation sous forme de : **__exit __init**

Qui servent à la spécification de l'allocation mémoire du noyau des fonctions. Notez simplement que **__init** et **__exit** sont à placer devant le nom de la fonction - il existe d'autres spécificateurs notamment pour les variables et fonctions d'un driver pour device hotplugable.

Ici elles n'ont pas de répercussion pour un module chargeable mais sont une bonne habitude à prendre.

Nous nous servons des macros **module_init()** et **module_exit()** qui servent a référencer les fonctions d'initialisation et de dé-initialisation afin que l'on puissent les nommées comme bons vous semble.

Après exécution du fichier Makefile correspondant au fichier driver.c ont charge le module-driver grâce a la fonction insmod et que l'on consulte le fichier.

```
/proc/devices
```

Qui répertorie les périphériques et est diviser en 2 parties:

- Les périphériques en mode caractères.
- Les périphériques en mode bloc.

```
$ sudo make
$ sudo insmod driver.ko
$ cat /proc/devices
Character devices:
1 mem
...
250 devname
...
254 rtc

Block devices:
1 ramdisk
...
254 mdp
```

Ont s'aperçoit de la création de notre périphérique **"devname"** avec comme numéro majeur 250.

```
$ sudo rmmod driver.ko
$ cat /proc/devices
Character devices:
1 mem
...
254 rtc

Block devices:
1 ramdisk
...
254 mdp
```

Après déchargement du module-driver **"devname"** n'est plus présent comme périphérique.

Débuter en kernel-programmation avec un driver en mode caractères (Bruggemann Eddie 2014).

La structure file_operations:

Introduction a la structure file_operations:

Afin de lier notre driver au requêtes d'un programme client, il existe une structure définie dans : <linux/fs.h>

Qui a pour prototype :

```
struct file_operations {
          struct module *owner;
       loff_t (*llseek) (struct file *, loff_t, int);
       ssize_t (*read) (struct file *, char __user *,
                                    size_t, loff_t *);
       ssize_t (*write) (struct file *, const char __user *,
                                       size_t, loff_t *);
       ssize_t (*aio_read) (struct kiocb *, const struct iovec *,
                                       unsigned long, loff_t);
       ssize_t (*aio_write) (struct kiocb *, const struct iovec *,
                                         unsigned long, loff_t);
       int (*iterate) (struct file *, struct dir_context *);
       unsigned int (*poll) (struct file *, struct poll_table_struct *);
       long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
       long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
       int (*mmap) (struct file *, struct vm_area_struct *);
       int (*open) (struct inode *, struct file *);
       int (*flush) (struct file *, fl_owner_t id);
       int (*release) (struct inode *, struct file *);
       int (*fsync) (struct file *, loff_t, loff_t, int datasync);
       int (*aio_fsync) (struct kiocb *, int datasync);
       int (*fasync) (int, struct file *, int);
       int (*lock) (struct file *, int, struct file_lock *);
       ssize_t (*sendpage) (struct file *, struct page *, int, size_t,
                                                        loff_t *, int);
       unsigned long (*get_unmapped_area)(struct file *, unsigned long,
                      unsigned long, unsigned long, unsigned long);
       int (*check_flags)(int);
       int (*flock) (struct file *, int, struct file_lock *);
       ssize_t (*splice_write)(struct pipe_inode_info *, struct file *,
                                      loff_t *, size_t, unsigned int);
       ssize_t (*splice_read)(struct file *, loff_t *,
                              struct pipe_inode_info *, size_t, unsigned int);
       int (*setlease)(struct file *, long, struct file_lock **);
       long (*fallocate)(struct file *file, int mode, loff_t offset,
                     loff_t len);
       int (*show_fdinfo)(struct seq_file *m, struct file *f);
};
```

Comme vous voyez, elle contient des pointeurs vers diverses fonctions et c'est en définissant ces fonctions qu'on va lier une action du programme client au driver chargé dans le noyau. Vous avez sûrement remarqué que le prototype ne donne seulement le type des paramètres (c'est spécifique des prototypes des sources du kernel).

Débuter en kernel-programmation avec un driver en mode caractères (Bruggemann Eddie 2014).

Nous allons nous intéresser aux fonctions et à leurs paramètres:

1. open().
2. release() Équivalent a close().
3. read()
4. write().

Dit au passage, la fonction unlock_ioctl correspond à la fonction ioctl .

La fonction compat_ioctl permet la prise en charge sur un système 64 bits la compatibilité d'un processus 32 bits grâce au transtypage du 3ième argument de qui est de type :

```
unsigned long           sur un système 32 bits.
void *                  sur un système 64 bits.
```

Description de la fonction open() :

Déclaration :

```
static int *open(struct inode *device_file, struct file * instance) ;
```

La structure inode représente le périphérique que nous souhaitons contrôler. Elle est définie dans <linux/fs.h>.

On peut s'en servir pour vérifier

1. Les droits d'accès.
2. Le propriétaire.
3. Le numéro majeur et mineur du périphérique.

Et bien d'autres choses (consultez les membres dans les sources).

La structure file représente l'instance du driver et est définie dans <linux/fs.h>.

On peut s'en servir pour :

1. Vérifier les droits d'accès (utile dans d'autres fonctions ou il n'y a pas de structure inode passer en argument).
2. Vérifier le mode d'accès (O_NONBLOCK).

Et bien d'autres choses (consultez les membres dans les sources).

Cette fonction doit retourner 0 en cas de succès sinon un code d'erreur.

Cette fonction est appelée dans le driver à chaque ouverture par une application du périphérique que nous souhaitons contrôler.

Débuter en kernel-programmation avec un driver en mode caractères (Bruggemann Eddie 2014).

Description de la fonction release():

Déclaration :

```
static int *release(struct inode *, struct file *);
```

Les paramètres sont les mêmes que pour la fonction open() ;

Cette fonction doit retourner 0 en cas de succès sinon un code d'erreur.

Cette fonction est appelée à chaque fermeture par une application du périphérique que nous souhaitons contrôler.

Débuter en kernel-programmation avec un driver en mode caractères (Bruggemann Eddie 2014).

Description de la fonction read() :

Déclaration :

```
static size_t *read(struct file *instance, char __user  *userbuffer,
                                    size_t count, loff_t *offset) ;
```

Cette fonction est appelée dans le driver à chaque opération de lecture fait par l'application qui communique avec le périphérique que nous souhaitons contrôler.

- La structure file joue le même rôle que dans la fonction open().
- Le paramètre userbuffer:

  Notez au passage le mot-clé __user devant le paramètre userbuffer qui signifie que l'adresse pointer se trouve dans l'espace utilisateur (user-space), pointé sur cette adresse. Ceci est l'argument le plus important, car comme énoncé précédemment l'espace mémoire utilisateur (celui de l'application cliente du périphérique que nous souhaitons contrôler) et séparé de l'espace mémoire du noyau (dans lequel s'exécute le driver). Il faut faire une translation d'adresse entre les deux afin de copier - dans l'espace utilisateur - les bytes (que l'application qui communique avec notre périphérique) demandent à lire, et renvoyer le nombre de bytes copiés dans celui-ci.

  Pour cela il existe plusieurs fonctions fournis par les sources du noyau définis dans <asm/uaccess.h> dont il existe un fichier propre à chaque architecture.

  ```
  copy_to_user(void *dst, const void *src, unsigned long count) ;
  ```

  copie count bytes depuis *src dans l'espace d'adresse utilisateur *dst et renvoie le nombre de bytes non-copiés.

  ou la macro qui ne supporte que des adresses de type char* ou int* :

  ```
  put_user(const void *to_copy, void *dst) ;
  ```

  Qui renvoies 0 en cas de succès, sinon -EFAULT.
  (Remarquer la conversion en nombre négatif du code d'erreur typique de la programmation driver du noyau).

  Par exemple pour renvoyer un "Hello world" à l'application qui tente de lire notre périphérique :

  ```
  copy_to_user(userbuffer, »Hello world »,strlen(Hello world)+1) ;
  ```

  ou

  ```
  put_user("Hello world",userbuffer) ;
  ```

  Ces fonctions font une vérification de la validité des adresses ce que l'on peut empêcher (pour des questions de performance) en employant les fonction sous-jacentes de ces fonctions qui ne vérifie pas la validité des adresse :

  ```
  __copy_to_user() ;
  __put_user() ;
  ```

Finalement ont peut vérifier la validité d'une adresse avec la macro:

```
access_ok(type, addr, size) ;
```

type : Type of d'access: VERIFY_READ ou VERIFY_WRITE. addr : adresse a vérifier. Size : nombre de bytes a vérifier.

- L'argument count correspond au nombre de bytes demandé par l'application communiquant avec notre périphérique - à lire depuis le périphérique.
- L'argument offset est la position dans l'argument userbuffer à partir duquel l'application communiquant avec notre périphérique souhaite lire.
- La fonction doit renvoyer le nombre de bytes copiés dans l'espace utilisateur de l'application communiquant avec notre périphérique.

Débuter en kernel-programmation avec un driver en mode caractères (Bruggemann Eddie 2014).

Description de la fonction write() :

Déclaration :

```
ssize_t *write(struct file *instance, const char __user *userbuffer,
                                      size_t count, loff_t *offset) ;
```

Cette fonction est appelée dans le driver pour une opération d'écriture (faîtes par l'application communiquant avec le périphérique que nous souhaitons contrôler.

- La structure file joue le même rôle que dans la fonction open().
- Le pointeur userbuffer pointe sur l'adresse où sont stockés les données que le client désire communiquer au périphérique.

  Il faut faire une translation d'adresse entre les deux, afin de copier depuis l'espace utilisateur les bytes que l'application communiquant avec notre périphérique demande à écrire - et renvoyer le nombre de bytes copiés dans celui-ci.

  Pour cela il existe plusieurs fonctions fournis par les sources du noyau définis dans <asm/uaccess.h> dont il existe un fichier propre à chaque architecture.

  ```
  copy_from_user(void *src, const void __user *dst, unsigned long count);
  ```

  copie count bytes depuis *src dans l'espace d'adresse utilisateur *dst et renvoie le nombre de bytes non-copiés.

  ou la macro qui ne supporte que des adresses de type char* ou int* :

  ```
  get_user(void *dst,const void *to_copy) ;
  ```

  Qui renvoies 0 en cas de succès, sinon -EFAULT.

  Ou la fonction qui ne supporte que du texte ASCII :

  ```
  strncpy_user(char *dst,const char __user *src,long count) ;
  ```

  A copier seulement depuis l'espace utilisateur dans l'espace du noyau avec la fonction :

  ```
  strnlen_user(const char __user *src,long count) ;
  ```

  Renvoie la taille du string pointer par src terminator ('0') compris minimum count long sinon 0 en cas d'erreur (adresse invalide, pas droit d'accès au données).

  Il existe des fonctions sous-jacentes qui ne vérifient pas la validité des adresses.

  ```
  __copy_from_user() ;
  __get_user() ;
  ```

- L'argument count correspond au nombre de bytes écrit par l'application communiquant avec notre périphérique.
- L'argument offset est la position dans l'argument userbuffer à partir duquel l'application communiquant avec notre périphérique désire écrire des données.
- La fonction doit renvoyer le nombre de bytes copiés dans l'espace du noyau.

Débuter en kernel-programmation avec un driver en mode caractères (Bruggemann Eddie 2014).

Un exemple Hello world:

Voici un exemple, qui renvoie "Hello World" à chaque tentative de lecture en écrivant ce que l'application communiquant avec notre périphérique lui envoie dans le fichier de log.

Le code créant le périhpérique et driver de celui-ci: hello_world_driver.c

```
#include <linux/module.h>  /** MODULE_LICENSE &&
                               MODULE_DESCRIPTION &&
                               MODULE_AUTHOR               */

#include <linux/cdev.h>    /** cdev_del() &&
                               cdev_add() &&
                               cdev_alloc()                */

#include <linux/device.h>  /** class_create() &&
                               device_create()  &&
                               device_destroy()            */
#include <linux/fs.h>      /** alloc_chrdev_region() &&
                               unregister_chrdev_region()  */

#include <linux/string.h>  /** strlen()                    */

#include <linux/fs.h>      /** struct file_operations      */

#include <asm/uaccess.h>   /** copy_to_user() &&
                               copy_from_user()            */

MODULE_LICENSE("GPL") ;
MODULE_DESCRIPTION("Hello World device driver") ;
MODULE_AUTHOR("None") ;

#define DEVICE_NAME "devname"

static dev_t dev_num ;
static struct cdev *driver_object ;
static struct class *device_class ;

static char hello_msg[]="hello world" ;

static ssize_t
file_ops_read(struct file *instance,char __user *read_buffer,
                                size_t count,loff_t *offset ) {

  ssize_t to_copy, not_copy ;

  to_copy=min(count,strlen(hello_msg)+1) ;

  not_copy=copy_to_user(read_buffer,hello_msg,to_copy) ;

  printk(KERN_DEBUG "%s send to application !!!\n",hello_msg) ;
  if (not_copy) {
    printk(KERN_DEBUG "copy_to_user() error !!!\n") ;
  }
  return to_copy - not_copy ;
}
```

Débuter en kernel-programmation avec un driver en mode caractères (Bruggemann Eddie 2014).

```
static ssize_t
file_ops_write(struct file *instance,const char __user *write_buffer,
                                          size_t count,loff_t *offset ) {
  ssize_t to_copy, not_copy ;


  char kernel_buf[128] ;
  memset(kernel_buf,'\0',128) ;
  to_copy=min(count,strlen(write_buffer)+1) ;

  not_copy=copy_from_user(kernel_buf,write_buffer,to_copy) ;
  printk(KERN_ALERT "Application send: %s !!!\n",kernel_buf) ;
  return to_copy-not_copy;
}

static int
file_ops_open(struct inode *device_file,struct file *instance ) {

  printk(KERN_DEBUG "device open function triggering !!!\n") ;
  printk(KERN_DEBUG "driver major: %d minor %d !!!\n",
                    imajor(device_file),iminor(device_file)) ;

  if (instance->f_flags & O_RDWR) {
    printk(KERN_DEBUG "device open in rw mode !!!\n") ;
  }
  else if (instance->f_flags & O_RDONLY) {
    printk(KERN_DEBUG "error: device open in read-only mode !!!\n") ;
    return -EIO ;
  }
  else if (instance->f_flags & O_WRONLY) {
    printk(KERN_DEBUG "error: device open in read-only mode !!!\n") ;
    return -EIO ;
  }
  return 0 ;
}

static int
file_ops_release(struct inode *device_file,struct file *instance ) {
  printk(KERN_DEBUG "device release function triggering !!!\n") ;
  return 0 ;
}

/** Define file_operations structure
    to bind to driver for client actions
    processing.                          */

struct file_operations fops = {
  .read         = file_ops_read  ,
  .write        = file_ops_write ,
  .open         = file_ops_open  ,
  .release      = file_ops_release  ,
} ;
```

Débuter en kernel-programmation avec un driver en mode caractères (Bruggemann Eddie 2014).

```
static int
__init my_device_init(void) {

  static int minor=0 ;
  static int minor_numbers=1 ;

  printk(KERN_DEBUG "start device creating\n") ;

  if (alloc_chrdev_region(&dev_num,minor,minor_numbers,DEVICE_NAME) < 0) {
    /** Allocate space for device and major number dev_num
        with minor number minor and minor_numbers count
        and as device name devname                          */

      printk(KERN_DEBUG "Device number and space alloctating error !!! \n") ;
      return -EIO ;
  }

  driver_object=cdev_alloc() ; /** Driver space allocating and
                                  instantiation as struct cdev object. */

  if (driver_object == NULL) {
    goto free_device_number ;
  }

  driver_object->owner=THIS_MODULE ;
  driver_object->ops=&fops ;

  if (cdev_add(driver_object,dev_num,1)) { /** register the driver object
                                               to the kernel            */
        goto free_device ;
  }

  device_class=class_create(THIS_MODULE,DEVICE_NAME) ;
  /** create new device class               */

  device_create(device_class,NULL,dev_num,NULL,"%s",DEVICE_NAME) ;
  /** register device in /dev/  filesystem  */

  if (device_class == NULL) {
        printk(KERN_DEBUG "Device class creating error !!!\n") ;
        goto free_device ;
  }

  printk(KERN_DEBUG "Device successfull created !!!\n") ;
  return 0 ;

  /* Errors handlong gotos */
  free_device :
    printk(KERN_DEBUG "Must free allocated space for device.\n") ;
    kobject_put(&driver_object->kobj) ;
  free_device_number :
    printk(KERN_DEBUG "chrdev registering error !!!\n") ;
    unregister_chrdev_region(dev_num,1) ;
    return -EIO ;
}
```

Débuter en kernel-programmation avec un driver en mode caractères (Bruggemann Eddie 2014).

```
static void
__exit my_device_exit(void) {

  device_destroy(device_class,dev_num) ;
  /** destroy device */

  class_destroy(device_class) ;
  /** destroy device class */

  cdev_del(driver_object) ;
  /** free driver_object memory space */

  unregister_chrdev_region(dev_num,1) ;
  /** unregister dev_t dev_num and minors count 1 */

  printk(KERN_DEBUG "Driver destruction succesfull !!!\n") ;
  return ;

}

module_init(my_device_init) ;
module_exit(my_device_exit) ;
```

Et voici le code de l'application client de notre driver: hello_world_client.c

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

int main(void) {
  int fd ;
  if ((fd=open("/dev/devname",O_RDWR | O_NOCTTY ) ) >= 0) {
    /** Device opening succesfull */
    fprintf(stdout,"open devname succesfull !!!\n") ;
  }
  else {
    fprintf(stderr,"open err) %m\n") ;
    exit(EXIT_FAILURE) ;
  }

  int ctrl=1 ;
  while (ctrl) {

    char *rd_buf=calloc(128,sizeof(char)) ;
    char *wr_buf=calloc(128,sizeof(char)) ;

    if (read(fd,rd_buf,(size_t) 128) > 0) {
      /** Reading from device succcessfull */
      fprintf(stdout,"device devname answer: %s\n",rd_buf) ;
    }
    else {
      fprintf(stderr,"rd err) %m\n") ;
      exit(EXIT_FAILURE) ;
    }

    fprintf(stdout,"Enter message to send to device\n") ;

    fgets(wr_buf,128,stdin) ;

    wr_buf[strlen(wr_buf)-1]='\0' ;
    /** Disable the linefeed character added from fgets() */

    if (strcmp(wr_buf,"stop") == 0) {
      /** We stop the commnunication with the device */
      ctrl=0 ;
      break ;
    }

    if (write(fd,wr_buf,strlen(wr_buf)) > 0) {
      /** Message sending to device succcessfull */
      fprintf(stdout,"message send to device:\n%s\n",wr_buf) ;
    }
```

```
    else {
      fprintf(stderr,"wr err) %m\n") ;
      exit(EXIT_FAILURE) ;
    }

    free(rd_buf) ;
    free(wr_buf) ;
  }
  close(fd) ;
  exit(EXIT_SUCCESS) ;
}
```

Débuter en kernel-programmation avec un driver en mode caractères (Bruggemann Eddie 2014).

Après avoir exécuter le fichier Makefile correspondant au fichier hello_world_driver.c et après compilation de l'application client du driver hello_world_client.c . Vous pourrez les faire communiquer ainsi:

```
$ sudo make
$ sudo insmod hello_world_driver.ko
$ gcc hello_world_client.c -o device_client
$ sudo device_client # Sinon changer les droits d'accès de /dev/devname
open devname succesfull !!!
device devname answer: hello world
Enter message to send to device
hi you device
message send to device:
hi you device
device devname answer: hello world
Enter message to send to device
stop
$
$ dmesg
[ 6784.816991] start device creating
[ 6784.817151] Device successfull created !!!
[ 6805.066774] device open function triggering !!!
[ 6805.066779] driver major: 250 minor 0 !!!
[ 6805.066781] device open in rw mode !!!
[ 6805.066866] hello world send to application !!!
[ 6815.855711] Application send: hi you device !!!
[ 6815.855743] hello world send to application !!!
[ 6818.785004] device release function triggering !!!
```

Débuter en kernel-programmation avec un driver en mode caractères (Bruggemann Eddie 2014).

Un exemple de driver de périphérique virtuel:

Introduction a l'exemple avancé:

Nous allons écrire un driver pour un périphérique virtuel avec quelques notions de programmation noyau en plus.

Nous allons nous servir du type de données atomique : **atomic_t**

- Pour gérer le nombre d'application se connectant au périphérique. Nous souhaitons qu'une seule application puisse se connecter.
- Pour le buffer de lecture et d'écriture.

Pour initialiser une variable de type atomic_t statiquement nous allons utiliser la macro:

```
ATOMIC_INIT()
```

définis dans <asm/atomic.h>

qui initialise une variable de type **atomic_t** avec l'argument passé en paramètre.

```
static atomic_t access_counter = ATOMIC_INIT(-1) ;
/** atomic access counter declaration and initialisation. */

static atomic_t bytes_read_available = ATOMIC_INIT(-1) ;
/** atomic bytes to read checker variable declaration and initialisation.  */

static atomic_t bytes_write_available = ATOMIC_INIT(-1) ;
 /** atomic bytes to write checker variable declaration and initialisation. */
```

Puis nous allons définir deux macros qui vérifie respectivement si les données sont disponibles en lecture et en écriture :

```
#define RD_OK ( atomic_read(&bytes_read_available) != 0)
 /** Macro to check if data available to be read.  */

#define WR_OK ( atomic_read(&bytes_write_available) != 0)
/** Macro to check if data available to be written.  */
```

Nous allons aussi nous occuper de l'aspect de la disponibilité de données en faisant dormir le périphérique - en attendant que les données soit effectivement disponibles et en implémentant un type de données prenant la forme d'une queue. De type **wait_queue_head_t** définis dans <linux/wait.h>.

Une queue est une structure de données fournis par le noyau, fonctionnant selon le principe que l'on peut écrire et lire des données dans/depuis celle-ci sous forme d'accès FIFO (First In First Out).

Mais ce type de données est une wait_queue_head_t signifiant que l'accès à la queue est soumis a une mise en attente souhaitant qu'un événement se produise. Nous allons utiliser la fonction :

```
wait_event_interruptible(wait_queue_head_t queue,condition) ;
```

définis dans <linux/wait.h>

Tout cela endort notre périphérique - en attendant un signal - quand il arrive, le signal produit la vérification de la condition (celle-ci décide si le périphérique continue à dormir ou pas).

Débuter en kernel-programmation avec un driver en mode caractères (Bruggemann Eddie 2014).

Afin de générer des nombres aléatoires de calculs, nous allons faire une sorte de process list (commande ps) que nous enregistrerons dans une structure - dont les membres nous donnent matière a générer deux nombres aléatoires - nous prendrons qu'un seul élément (l'index nous est fourni par le nombre des changements de contexte du processus actuellement en cours d'exécution, accessible par la macro **current**) :

- Le temps impartis au processus pour s'exécuter dans le processeur (celui-ci est calculé par ordonnanceur).
- Le temps passé dans le processeur depuis son lancement.

Débuter en kernel-programmation avec un driver en mode caractères (Bruggemann Eddie 2014).

Code d'un driver de périphérique virtuel:

Voici un exemple, demande a faire une addition a la première tentative de lecture puis enregistre votre réponse envoyer par la fonction write de l'application communiquante avec notre périphérique et vous dit a la deuxième tentative de lecture si le résultat est exact en alternance.

Le code créant le périhpérique et driver de celui-ci: calc_device_driver.c

```
#include <linux/module.h>  /** MODULE_LICENSE &&
                               MODULE_DESCRIPTION &&
                               MODULE_AUTHOR               */

#include <linux/cdev.h>    /** cdev_del() &&
                               cdev_add() &&
                               cdev_alloc()              */

#include <linux/device.h>  /** class_create() &&
                               device_create()  &&
                               device_destroy()          */

#include <linux/fs.h>      /** alloc_chrdev_region() &&
                               unregister_chrdev_region() */

#include <linux/string.h>  /** strncpy()                 */

#include <linux/fs.h>      /** struct file_operations    */

#include <asm/uaccess.h>   /** copy_to_user() &&
                               copy_from_user            */

#include <linux/sched.h>   /** struct task_struct        */


MODULE_LICENSE("GPL") ;
MODULE_DESCRIPTION("Addition device driver") ;
MODULE_AUTHOR("None") ;

#define DEVICE_NAME "devname"

static dev_t dev_num ;
static struct cdev *device_object ;
static struct class *device_class ;

static wait_queue_head_t wait_read, wait_write ;

static atomic_t access_counter = ATOMIC_INIT(-1) ;
/** atomic access counter. */

static atomic_t bytes_read_available = ATOMIC_INIT(-1) ;
/** atomic bytes to read checker variable initialisation.  */

static atomic_t bytes_write_available = ATOMIC_INIT(-1) ;
/** atomic bytes to write checker variable initialisation. */
```

```
#define RD_OK ( atomic_read(&bytes_read_available) != 0)
/** Macro to check if data available to be read.          */

#define WR_OK ( atomic_read(&bytes_write_available) != 0)
/** Macro to check if data available to be written.       */


unsigned long result ; /** Addition result */

char res_str[128] ;
/** char buffer for getting answers from client application */

int status=0 ;
/** Switcher:
*  Sending adddition.
*  Sending result check.
************************/

struct Data_Random {
  /** Data structure for process list */
  unsigned long value1 ;
  unsigned long value2 ;
} random[256] ;

u16 c ;
unsigned long value1 ; /** Operand 1  */
unsigned long value2 ; /** Operand 2  */

void register_random_data(void) {

  struct task_struct *task ; /** Iterator */

  c=0 ;

  for_each_process(task) {
    /** Registering 256 processes in run. */

    random[c].value1=task->se.vruntime ;
    /** The amount of time given for the process to stay in the processor. */

    random[c].value2=task->se.exec_start ;
    /** The time the process spend in the processor since start.          */

    c++ ;

    if (c == 256) {
      break ;
    }

  }
  return ;
}
```

```
void set_random_data(void) {
  struct task_struct *task ;

  task=current ; /** Get current process */

  value1=random[(task->fpu_counter < c) ? task->fpu_counter : 0 ].value1 % 255 ;
  /** task->fpu_counter The number of consecutive
      context switches that the FPU is used.   */

  value2=random[(task->fpu_counter < c) ? task->fpu_counter : 0 ].value2 % 255 ;
  /** task->fpu_counter The number of consecutive
      context switches that the FPU is used.   */

  return ;

}

static int
file_ops_open(struct inode *device_file,struct file *instance ) {

  if (instance->f_flags & O_RDWR) {
    printk(KERN_ALERT "device open in rw mode !!!\n") ;

    if (atomic_inc_and_test(&access_counter) == 1) {
      /** We permit only one client application at the same time
      *  by testing atomic_inc_and_test(&access_counter)
      *  who return 1 only if access_counter == 1 after incrementing
      ************************************************************/
      return 0 ;
    }
    else {
      return -EBUSY ;
    }
  }
  else {
    printk(KERN_ALERT "device open in wrong mode !!!\n") ;
    return -EIO ;

  }

  printk(KERN_DEBUG "device open function triggering !!!\n") ;
  return 0 ;
}



static int
file_ops_release(struct inode *device_file,struct file *instance ) {
  atomic_dec(&access_counter) ;
  /** Release access control lock */
  printk(KERN_DEBUG "device release function triggering !!!\n") ;
  return 0 ;
}
```

Débuter en kernel-programmation avec un driver en mode caractères (Bruggemann Eddie 2014).

```
static ssize_t
file_ops_read(struct file *instance,char __user *read_buffer,
                                size_t count,loff_t *offset ) {

  size_t to_copy, not_copy ;
  char buffer_kern_space[128] ; /** kernel data-space buffer */

  if ((instance->f_flags & O_NONBLOCK) && ! RD_OK) {
    /** case O_NONBLOCK modus and data to read not available. */
    printk(KERN_DEBUG "device read non-block and not ready !!!\n") ;
    return -EAGAIN ;
  }

  if (wait_event_interruptible(wait_read,RD_OK)) {
    /** Wait until data available throught RD_OK macro check.  */
    printk(KERN_DEBUG "device read not wake up error !!!\n") ;
    return -ERESTART ;
  }

  to_copy=min((size_t) atomic_read(&bytes_read_available),count) ;
  /** atomic read operation and return gthe number of bytes available. */

  if (status == 0) {
    register_random_data() ;
    set_random_data() ;

    result = value1 + value2 ;

    sprintf(buffer_kern_space,"\nHello compute:\n %lu + %lu",value1,value2) ;
    /** copy readed data in kernel buffer to answers */
    status=1 ;
  }
  else {
    if (result == simple_strtoul(res_str,NULL,0)) {
      sprintf(buffer_kern_space,"RIGHT\n") ;
      /** Wrinting answers in kernel buffer to send to client. */
    }
    else {
      sprintf(buffer_kern_space,"WRONG\n") ;
      /** Wrinting answers in kernel buffer to send to client. */
    }
    status=0 ;
  }

  not_copy=copy_to_user(read_buffer,buffer_kern_space,to_copy) ;
  /** copy readed data in kernel-space buffer to return to client application.
    * and return number of not read data
    ************************************************************************/

  atomic_sub(to_copy-not_copy,&bytes_read_available) ;
  /** atomic data to read computing */

  return to_copy - not_copy ;

}
```

```
static ssize_t
file_ops_write(struct file *instance,const char __user *write_buffer,
                                        size_t count,loff_t *offset ) {

  size_t to_copy, not_copy ;

  printk(KERN_DEBUG "device write function triggering !!!\n") ;

  if ((instance->f_flags & O_NONBLOCK) && ! WR_OK) {
    /** case O_NONBLOCK modus and data to write not available. */
    printk(KERN_DEBUG "device write non-block and not ready !!!\n") ;
    return -EAGAIN ;
  }

  if (wait_event_interruptible(wait_write,WR_OK)) {
    /** Wait until data available throught WR_OK macro check.  */
    printk(KERN_DEBUG "device write not wake up error !!!\n") ;
    return -ERESTART ;
  }

  memset(res_str,'\0',128) ;

  to_copy=min((size_t) atomic_read(&bytes_write_available),count) ;
  /** atomic read operation and return adress from end of read data. */

  not_copy=copy_from_user(res_str,write_buffer,to_copy) ;

  atomic_sub(to_copy-not_copy,&bytes_write_available) ;
  /** atomic data to read computing */


  return to_copy - not_copy ;

}



struct file_operations fops = {
  .read                 = file_ops_read  ,
  .write        = file_ops_write ,
  .open                 = file_ops_open  ,
  .release      = file_ops_release  ,

} ;
```

Débuter en kernel-programmation avec un driver en mode caractères (Bruggemann Eddie 2014).

```
static int __init my_device_init(void) {
  static int minor=0 ;
  static int minor_numbers=1 ;

  printk(KERN_DEBUG "start device creating !!!\n") ;

  if (alloc_chrdev_region(&dev_num,minor,minor_numbers,DEVICE_NAME) < 0) {
    /** allocate space for device dev_t dev_num with
        minor number minor and minor_numbers count
        as device name DEVICE_NAME */

    printk(KERN_DEBUG "chrdev alloc error !!! \n") ;
    return -EIO ;
  }

  device_object=cdev_alloc() ;
  /** device instantiation with space allocation as struct cdev. */

  if (device_object == NULL) {
    goto free_device_number ;
  }

  device_object->ops=&fops ;
  device_object->owner=THIS_MODULE ;

  if (cdev_add(device_object,dev_num,1)) {
    /** register the device object to the kernel with
        the device number dev_num and
        counts of minor numbers 1 */

    goto free_device ;
  }

  device_class=class_create(THIS_MODULE,DEVICE_NAME) ;
  /** create new device class */
  device_create(device_class,NULL,dev_num,NULL,"%s",DEVICE_NAME) ;
  /** register device in /dev/  filesystem */

  if (device_class == NULL) {
    printk(KERN_DEBUG "device class creating error !!!\n") ;
    return -1 ;
  }

  printk(KERN_DEBUG "device successfull created !!!\n") ;
  return 0 ;

  /** error handling goto ; */
  free_device :
    printk(KERN_DEBUG "kobject triggering\n") ;
    kobject_put(&device_object->kobj) ;
  free_device_number :
    printk(KERN_DEBUG "chrdev registering error !!!\n") ;
    unregister_chrdev_region(dev_num,1) ;
    return -EIO ;
}
```

Débuter en kernel-programmation avec un driver en mode caractères (Bruggemann Eddie 2014).

```
static void __exit my_device_exit(void) {

  device_destroy(device_class,dev_num) ;
  /** destroy device */
  class_destroy(device_class) ;
  /** destroy device class */

  cdev_del(device_object) ;
  /** free device_object memory space */
  unregister_chrdev_region(dev_num,1) ;
  /** unregister dev_t dev_num and minors count 1 */

  printk(KERN_DEBUG "driver destruction succesfull !!!\n") ;
  return ;

}

module_init(my_device_init) ;
module_exit(my_device_exit) ;
```

Débuter en kernel-programmation avec un driver en mode caractères (Bruggemann Eddie 2014).

Et voici le code de l'application client de notre driver: calc_device_client.c

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

int main(void) {
  int fd ;
  if ((fd=open("/dev/devname",O_RDWR| O_NOCTTY ) ) >= 0) {
    fprintf(stdout,"open devname succesfull !!!\n") ;
  }
  else {
    fprintf(stderr,"1) %m\n") ;
    exit(EXIT_FAILURE) ;
  }


  int c=0 ;
  while (c < 5) {
    char *rd_buf=malloc(128) ;
    if (read(fd,rd_buf,(size_t) 128) > 0) {
      fprintf(stdout,"\ndevname send:\n%s\n",rd_buf) ;
    }
    else {
      fprintf(stderr,"2) %m\n") ;
      exit(EXIT_FAILURE) ;
    }
    free(rd_buf) ;

    fprintf(stdout,"\nEnter answer to send to driver\n>>> ") ;
    char *write_buf=malloc(128) ;

    fgets(write_buf,128,stdin) ;
    write_buf[strlen(write_buf)-1]='\0' ;

    if (write(fd,write_buf,strlen(write_buf)) > 0) {
      fprintf(stdout,"\nsend answer: [ \"%s\" ] to devname\n",write_buf) ;
    }
    else {
      fprintf(stderr,"3) %m\n") ;
      exit(EXIT_FAILURE) ;
    }
    free(write_buf) ;
    char *response=malloc(128) ;
    if (read(fd,response,(size_t) 128) > 0) {
      fprintf(stdout,"\ndevname answer:\n%s\n",response) ;
    }
    else {
      fprintf(stderr,"2) %m\n") ;
      exit(EXIT_FAILURE) ;
    }
```

```
    free(response) ;
    c++ ;
    sleep(1) ;
  }
  close(fd) ;

  exit(EXIT_SUCCESS) ;

}
```

Voila vous n'avez qu'a exécuter le fichier Makefile a inséré le module dans le noyau avec la commande insmod et a compiler et lancer le client...

Débuter en kernel-programmation avec un driver en mode caractères (Bruggemann Eddie 2014).

Sources et Remerciements:

Sources:

Livre: Linux-Treiber entwickeln (3rd Edition).
publication: Mars 2011.
Pages: 598 pages.
Éditeur: Dpunkt Verlag.
Auteurs: Jürgen Quade & Eva-Katharina Kunst.
Langue: Deutsch.
Note: Le livre contient un index de 400 kernel-fonctions avec prototype et description.

Livre: Linux Kernel Development (3rd Edition)
publication: 22 juin 2010
Pages: 440 pages.
Éditeur: Addison Wesley.
Auteur: Robert Love.
Langue: English.

Tutoriel: Ecriture de driver sous Linux grâce au Langage C
publication: Le 4 janvier 2007 - Mis à jour le 21 février 2007
Auteur: Roux Benjamin
Langue: Français.

Remerciements:

Auteur: Bruggemann Eddie.
Corrections: Pinta Oleander.
Publication: 28/03/2014.
Remerciement: Merci a ma famille pour leurs soutient et aux docteurs.
Conseil: Rien n'équivaut a la lectures des sources du kernel pour apprendre et comprendre la kernel-programmation.