

2009

Développement de pilote Linux

Pilote en mode block

→ Travail réalisé par :

- DHOUIB Khalil
- NEJI Wafa

GL5/G2

16/04/2009



I. Introduction

- Le système Linux étant de la famille UNIX, la structure d'un pilote Linux est relativement proche de celle d'un pilote générique UNIX. Un pilote est une portion de code exécutée dans l'espace du noyau. L'API d'un pilote Linux est très fortement inspirée de celle des pilotes UNIX des années 1970. Nous rappelons qu'elle est basée sur l'utilisation de fonctions ou méthodes permettant d'effectuer des actions basiques.

→ Les pilotes en mode bloc (block device driver): Ce sont des pilotes destinés à manipuler des périphériques de stockage avec lesquels ils échangent des blocs de données (disque dur, CDROM, DVD, disque mémoire, etc.). La taille des blocs peut être de 512, 1024, 2048 (ou plus) octets suivant le périphérique.

L'objectif de ce projet est donc le développement d'un driver en mode block. On est donc amené à réaliser un module noyau permettant de disposer d'un périphérique virtuel offrant une capacité mémoire de 1Ko.

II. Les bases du module :

- *Qu'est-ce qu'un module ? :*

→ Un module est un programme qui sera exécuté par le noyau, comme le driver d'un périphérique (USB, carte son, etc...). Généralement ce module tourne en mode noyau.

II.1. Chargement et déchargement de module :

→ Un module possède un point d'entrée et un point de sortie :

- Point d'entrée : `int MonDriver_init(void)`
- Point de sortie : `void MonDriver_exit(void)`

→ Pour dire au module que ces deux fonctions sont le point d'entrée et le point de sortie, nous utilisons ces deux macros :

```
/****** Declaration of the init and exit functions *****/
```

```
module_init(MonDriver_init);  
module_exit(MonDriver_exit);
```

Ces 2 fonctions sont automatiquement appelées, lors du chargement et du déchargement du module, avec « *insmod* » et « *rmmod* »

- La fonction `module_init(MonDriver_init)` doit s'occuper de préparer le terrain pour l'utilisation de notre module : *allocation mémoire, initialisation matérielle, etc...*

- La fonction `module_exit(MonDriver_exit)` doit quant à elle *défaire ce qui a été fait par la fonction module_init()*.

II.2 Description du module :

→ On peut décrire nos modules à l'aide des différentes macros présentes dans `<linux/module.h>`

```
/****** Description du module *****/
```

```
MODULE_DESCRIPTION( "Mon module" );
```

Place une description du module dans le fichier objet

```
MODULE_AUTHOR( "Neji Wafa & Dhouib Khalil" );
```

Place le nom de l'auteur dans le fichier objet

```
MODULE_LICENSE( "$LICENSE$" );
```

Indique le type de licence du module

II.3 Passage de paramètres au module:

→ Il serait bien utile de passer des paramètres à notre module. Une fonction et une macro sont donc disponibles pour cela :

```
{ module_param(nom, type,permissions)  
  MODULE_PARM_DESC(nom, desc)  
}
```

→ Dans notre cas, on va passer deux paramètres au module :

- 1^{er} paramètre Le nombre majeur «**MAJOR**» : Ce nombre majeur a pour but d'identifier notre driver.
- 2^{ème} paramètre : Le nom du périphérique «**DEVICE_NAME**»

```
/* ***** Passage de paramètres pour le modules ***** */  
  
static int MAJOR=0;  
static char *DEVICE_NAME = "mondevice";  
  
module_param( MAJOR, int, 0);  
MODULE_PARM_DESC(MAJOR, "Static major number (none = dynamic)");  
  
module_param(DEVICE_NAME, charp, 0000);  
MODULE_PARM_DESC(DEVICE_NAME, "The Name of Device");
```

1^{er} paramètre : Le nombre majeur pour identifier notre driver

2^{ème} paramètre : Le nom du périphérique

→ Voilà, maintenant qu'on a bien spécifié les bases d'un module, nous allons voir la création d'un driver en mode block et les interactions possibles avec le mode utilisateur, et le matériel.

III.Driver en mode block :

→ Les pilotes en mode bloc (block device driver): Ces pilotes sont destinés à manipuler des périphériques de stockage avec lesquels ils échangent des blocs de données (disque dur, CDROM, DVD, disque mémoire, etc.). La taille des blocs peut être de 512, 1024, 2048 (ou plus) octets suivant le périphérique (dans notre cas le bloc sera de taille 1Ko).

III. 1 Ajout d'un driver au noyau:

→ L'enregistrement du driver dans le noyau doit se faire lors de l'initialisation du driver ; c'est à dire lors du chargement du module, donc dans la fonction « **MonDriver_Init()** » ; en appelant la fonction : « **register_blkdev** » :

```
static int MonDriver_init(void)
{ int i;
  /* ETAPE 1 : Ajout d'un driver au noyau (Enregistrement du pilote):
     Cette étape permettra au périphérique d'être reconnu par le noyau
     On utilise la fonction register_blkdev(unsigned int major,const char* name) */

  1 MAJOR = register_blkdev( MAJOR, DEVICE_NAME);

  MAJOR : Numéro majeur du driver, 0
  indique que l'on souhaite une affectation
  dynamique.

  DEVICE_NAME : Nom du
  périphérique qui apparaîtra
  dans « /proc/devices »

  if (MAJOR < 0) {
    printk(KERN_WARNING "%s: Unable to obtain major number %d\n", DEVICE_NAME, MAJOR);
    return -EBUSY; }
}
```

→ 1 Lors de l'ajout d'un driver au noyau, le système lui affecte un nombre majeur. Ce nombre majeur a pour but d'identifier notre driver (Dans le cas de l'ajout d'un nouveau pilote par l'utilisateur, il sera préférable d'utiliser les fonctionnalités d'allocation dynamique de majeur plutôt que d'utiliser une valeur fixe, pour ceci il suffit d'initialiser le nombre majeur avec 0)

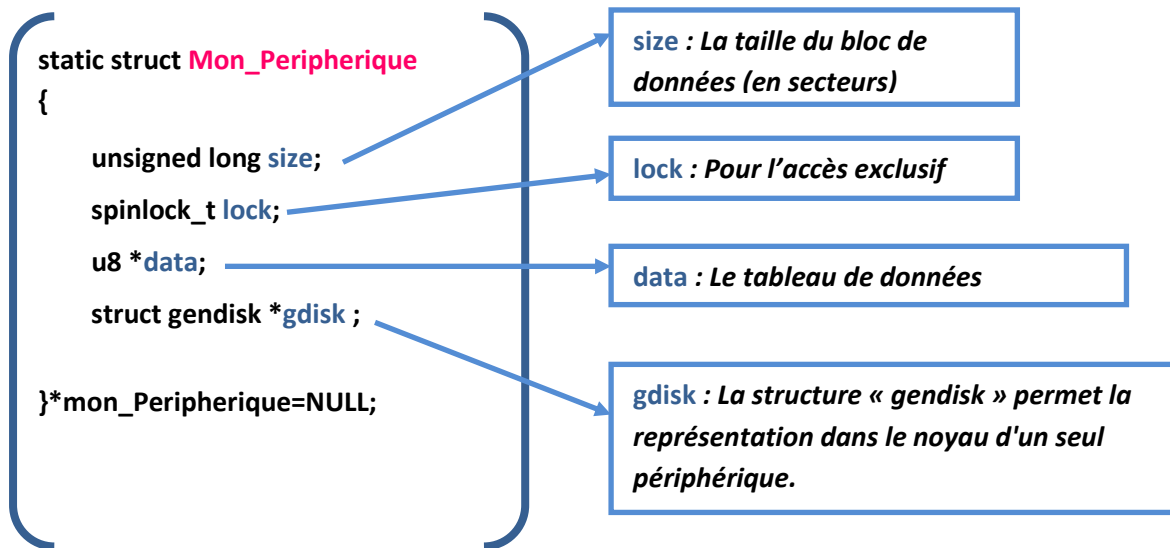
!Remarque :

→ De même, on supprimera le driver du noyau (lors du déchargement du module dans la fonction **MonDriver_exit()**), en appelant la fonction : « **unregister_blkdev** ».

III. 2 Représentation interne et allocation mémoire du périphérique de stockage:

III. 2.1 Représentation interne du périphérique de stockage :

→ On se propose de définir la structure «*Mon_Péripherique*» permettant la représentation interne de notre périphérique de stockage :



III. 2.2 Allocation mémoire pour le périphérique de stockage:

→ En mode noyau, l'allocation mémoire se fait via la fonction ***kmalloc***, la désallocation, elle se fait via `kfree`.

→ Ces fonctions, sont très peu différentes des fonctions de la bibliothèque standard. La seule différence, est un argument supplémentaire pour la fonction `kmalloc()` : ***la priorité***. Cet argument peut-être :

- ***GFP_KERNEL*** : allocation normale de la mémoire du noyau
- `GFP_USER` : allocation mémoire pour le compte utilisateur (faible priorité)
- `GFP_ATOMIC` : alloue la mémoire à partir du gestionnaire d'interruptions

→ L'allocation mémoire pour notre périphérique de stockage se fait comme suit :

```
static int MonDriver_init(void)
{ int i;
/* ETAPE 1 */ MAJOR = register_blkdev( MAJOR, DEVICE_NAME);
if (MAJOR < 0) {
    printk(KERN_WARNING "%s: Unable to obtain major number %d\n", DEVICE_NAME, MAJOR);
    return -EBUSY; }

/* Etape 2 : Allocation mémoire pour le périphérique:
   On utilisera la fonction kmalloc(size_t size, int flags)*/

    mon_Peripherique = kmalloc(sizeof(struct Mon_Peripherique), GFP_KERNEL);

    if (mon_Peripherique == NULL) {
        printk(KERN_WARNING "%s: Error in allocating the device with kmalloc\n", DEVICE_NAME);
        goto out_unregister;
    }
}
```

GFP_KERNEL: *priorité, allocation normale de la mémoire du noyau.*

Au cas où l'allocation mémoire échoue, on supprimera le driver du en appelant la fonction « unregister_blkdev ».

```
out_unregister:
    unregister_blkdev(MAJOR, DEVICE_NAME);

    printk (KERN_NOTICE "%s: Unregister Function\n", DEVICE_NAME);
    return -ENOMEM;
```

III. 3 Initialisation du périphérique et allocation des mémoires sous-jacentes:

→ Cette étape permettra d'assurer la disponibilité du périphérique au système

III. 3.1 Définir la plage d'adresse

→ On commence par choisir la plage d'adresse occupée par notre périphérique. Pour ceci, on appelle la fonction « **memset** »

```
static int MonDriver_init(void)
{ int i;
/* ETAPE 1 */ MAJOR = register_blkdev( MAJOR, DEVICE_NAME);
if (MAJOR < 0) {
    printk(KERN_WARNING "%s: Unable to obtain major number %d\n", DEVICE_NAME, MAJOR);
    return -EBUSY; }
/* Etape 2 */ mon_Peripherique = kmalloc(sizeof(struct Mon_Peripherique), GFP_KERNEL);
if (mon_Peripherique == NULL) {
    printk(KERN_WARNING "%s: Error in allocating the device with kmalloc\n", DEVICE_NAME);
    goto out_unregister;}

/* ETAPE 3 : Initialisation du périphérique et allocation des mémoires sous-jacentes:
Cette étape permettra d'assurer la disponibilité du périphérique au système */

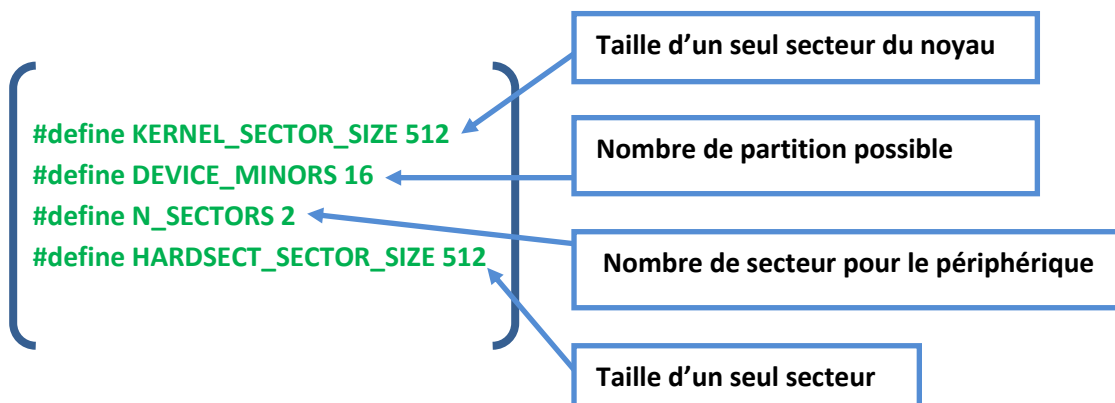
/*ETAPE 3.1: définition de la plage d'adresse*/

    memset(mon_Peripherique, 0, sizeof(struct Mon_Peripherique));
}
```

Spécification de la plage
d'adresse occupée par notre
périphérique.

III. 3.2 Initialisation de la taille du périphérique et du tableau de données :

→ Avant d'initialiser la taille du bloc de données du périphérique du stockage, il sera bien nécessaire de définir quelques variables statiques qui seront utilisés dans le reste de notre code :



→ On commence par définir la taille du bloc de données en secteurs. Dans notre cas, on à besoin de 1Ko de mémoire : c'est à dire 1024 octets, or la taille d'un seul secteur étant de 512 octet, on à donc besoin donc de deux secteurs.

→ On alloue ensuite l'espace mémoire relatif au bloc de données, on obtient alors le code suivant :

```
static int MonDriver_init(void)
{ int i;
  /* ..... */

  /* ETAPE 3.2
  /*initialisation taille périphérique */
  mon_Peripherique->size = N_SECTORS*HARDSECT_SECTOR_SIZE;

  mon_Peripherique->data = vmalloc(mon_Peripherique->size);

  for (i=0; i<mon_Peripherique->size;i++)
    memcpy(mon_Peripherique->data + i,"", 1);
}
```

Spécification de la taille
du bloc de données

Allocation mémoire pour
le bloc de données

Cette boucle permet d'initialiser
le block de données avec le vide
et d'éviter par la suite
l'apparition des caractères
spéciaux lors de l'exécution

III. 3.3 Allocation de la 'request queue' (file de demande) relatif au périphérique :

→ On commence tout d'abord par définir la variable « **req_queue** » de type « **request_queue** » permettant la représentation de la file de demande pour la lecture ou l'écriture depuis et vers notre périphérique de stockage.

```
static struct request_queue *req_queue; /* The device request queue */
```

→ La valeur de cette variable sera affecté par la suite à « **mon_Peripherique->gdisk->queue** » (Voir la section « **II.3.4.2 Initialisation de la structure gendisk** »)

!Remarque :

Il serait bien important d'allouer et initialiser « **Miniblock->lock** » avant de procéder à l'allocation de la request queue :

```
static int MonDriver_init(void)
{
  /* ..... */
  spin_lock_init(&mon_Peripherique->lock);
}
```

→ En effet, l'allocation de la 'request_queue' nécessite :

1

1. **La request function (PériphériqueRW_request):** c'est la fonction qui assure les opérations de lecture et écriture depuis et vers le périphérique (*voir la section II. 5 Implémentation de la request function*)

2

2. **Le spinlock (mon_Périphérique->lock):** permettant le contrôle d'accès à la request_queue. L'utilisation du spinlock permet d'empêcher le noyau d'accepter de nouvelles requêtes(R/W) tant que la request function « PériphériqueRW_request » manipule ce périphérique.

→ La fonction permettant l'allocation de la 'request_queue' est « **blk_init_queue** » :

```
static int MonDriver_init(void)
{
    /* ..... */
    spin_lock_init(&mon_Périphérique->lock);

    /*ETAPE 3.3 Allocation de la 'request queue' */
    req_queue = blk_init_queue( PériphériqueRW_request, &mon_Périphérique->lock);

    if (req_queue == NULL) {
        printk(KERN_WARNING "%s: error in blk_init_queue\n", DEVICE_NAME);
        goto out_free;
    }
}
```

```
out_free:
    vfree(mon_Périphérique->data);
    kfree(mon_Périphérique);
```

Au cas où l'allocation de la 'request_queue' échoue, on libérera l'espace mémoire associé au bloc de données et à la structure représentant le périphérique du stockage

III. 3.4 Allocation mémoire, initialisation et installation de la structure « gendisk »

→ Comme on l'a déjà mentionné, la structure « **gendisk** » permet la représentation dans le noyau de notre périphérique de stockage.

III. 3.4.1 Allocation mémoire pour la structure « gendisk » :

→ La structure « **gendisk** » est alloué dynamiquement via la fonction '**struct gendisk *alloc_disk(int minors)**'. L'argument '*minors*' représente le nombre de partitions maximum que peut avoir le périphérique.

```
static int MonDriver_init(void)
{
    /* ..... */
    /* ETAPE 3.4 : Allocation mémoire, Initialisation et installation de la structure gendisk */

    /*ETAPE 3.4 .1: Allocation mémoire pour gendisk*/

    mon_Peripherique->gdisk = alloc_disk(DEVICE_MINORS);

    if (!mon_Peripherique->gdisk) {
        printk(KERN_WARNING "%s: Error in allocating the struct gendisk: alloc_disk\n",
        DEVICE_NAME);
        goto out_free;
    }
}
```

Allocation mémoire pour gendisk.

Au cas où l'allocation de la 'request_queue' échoue, on libérera l'espace mémoire associé au bloc de données et à la structure représentant le périphérique du stockage

!Remarque :

→ On supprimera la structure « gendisk » du noyau (lors du déchargement du module dans la fonction **MonDriver_exit()**), en appelant la fonction : « **void del_gendisk(struct gendisk *gd)** ».

III. 3.4.2 Initialisation de la structure gendisk :

→ La structure « *gendisk* » possède des champs qui doivent être initialisés par le pilote.

```
static int MonDriver_init(void)
{
/*.....*/
/* ETAPE 3.4 : Allocation mémoire, Initialisation et installation de la structure gendisk */

/*ETAPE 3.4 .1: Allocation mémoire pour gendisk*/
/...../

/* ETAPE 3.4.2 : Initialisation de la structure gendisk:*/

mon_Peripherique->gendisk->major = MAJOR;
mon_Peripherique->gendisk->minor = DEVICE_MINORS;
mon_Peripherique->gendisk->first_minor = 0;

mon_Peripherique->gendisk->fops = &block_dev_op;

mon_Peripherique->gendisk->queue = req_queue;

mon_Peripherique->gendisk->private_data = mon_Peripherique;

snprintf(mon_Peripherique->gendisk->disk_name,10, "%s", DEVICE_NAME+'0');
set_capacity(mon_Peripherique->gendisk, N_SECTORS*(HARDSECT_SECTOR_SIZE/KERNEL_SECTOR_SIZE));
}
```

Ces champs décrivent les nombres utilisés par le périphérique

Association des méthodes du pilote (voir la section « [II.4 Implémentation des appels systèmes](#) »)

Structure utilisée par le noyau pour contrôler les demandes de lecture/écriture depuis et vers le périphérique.

Pour permettre au pilote de pointer sur les données internes

Définir le nombre de secteur associé à gendisk

III. 3.4.3 Installation de la structure gendisk

→ L'allocation mémoire de la structure « gendisk » ne permet pas à cette dernière d'être valide pour le système. Pour assurer cette validité, il suffit d'appeler la fonction « **void add_disk(struct gendisk *gd)** ». En invoquant la méthode **add_disk**; le pilote sera prêt pour recevoir les requêtes

```
static int MonDriver_init(void)
{
    /* ..... */
    /* ETAPE 3.4 : Allocation mémoire, Initialisation et installation de la structure gendisk */

    /*ETAPE 3.4 .1: Allocation mémoire pour gendisk*/
    /*.....*/

    /* ETAPE 3.4.2 : Initialisation de la structure gendisk:*/
    /*.....*/

    /* ETAPE 3.4.3 : Installer la structure gendisk */

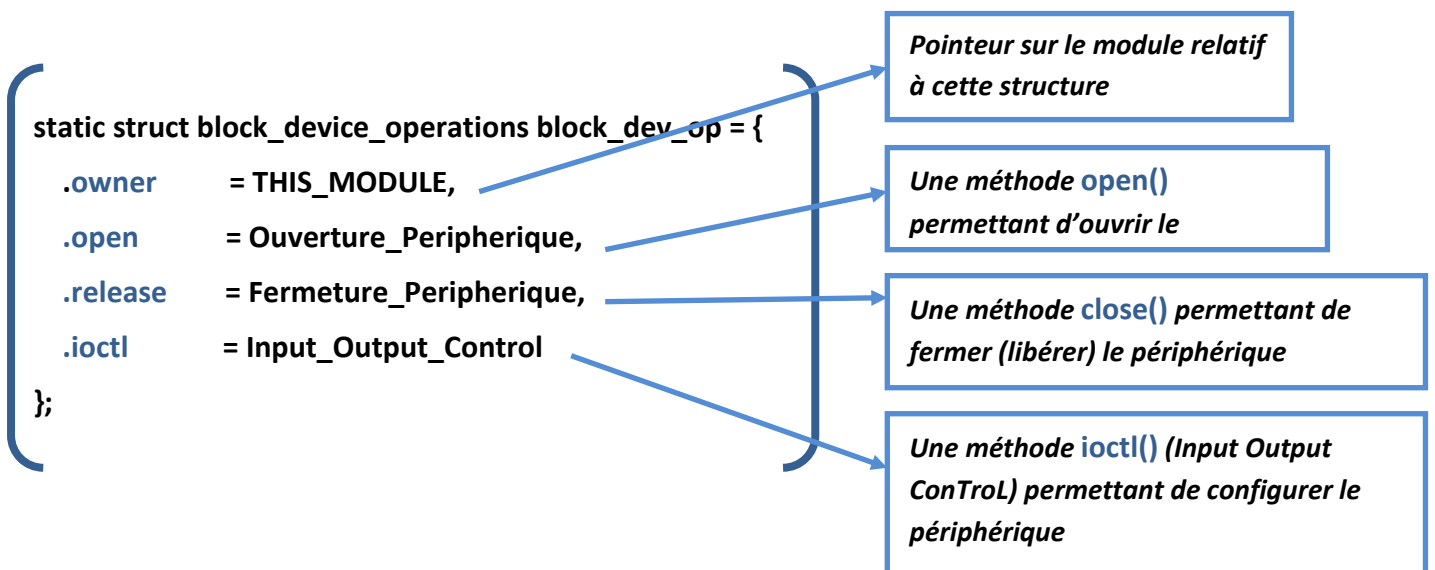
    add_disk(mon_Peripherique->gdisk);

}
```

II. 4 Implémentation des appels systèmes (méthodes de pilote)

La structure « **block_device_operations** » dont le type est décrit dans le fichier <linux/fs.h> permet de définir les méthodes du pilote.

→ On commence donc par définir la variable « **block_dev_op** » de type « **block_device_operations** » pour définir les méthodes de pilote à implémenter



→ Ces méthodes sont activées depuis un programme de l'espace utilisateur en passant par les fichiers spéciaux du répertoire */dev*. Ces fichiers sont appelés nœuds. Ces fonctions renvoient 0 (ou >0) en cas de succès, une valeur négative sinon.

II. 4.1 Méthodes open et release

→ Dans la méthode « *Ouverture_Péripherique* » (open) , on se contente du Remplissage de la structure privée qui sera placée dans « file->private_data », et d'afficher un message annonçant l'ouverture du périphérique. Quand à la méthode « *Fermeture_Péripherique* » release, elle permet uniquement l'affichage d'un message annonçant la fermeture du périphérique

```
/****** Ouverture du périphérique *****/
```

```
static int Ouverture_Péripherique(struct inode *inode, struct file *filp)
```

```
{
```

```
    mon_Péripherique = inode->i_bdev->bd_disk->private_data;
```

```
    filp->private_data = mon_Péripherique;
```

```
    printk (KERN_NOTICE "%s: Open Function\n", DEVICE_NAME);
```

```
    return 0;
```

```
}
```

```
/****** Fin Ouverture du périphérique *****/
```

```
/****** Fermeture du périphérique *****/
```

```
static int Fermeture_Péripherique(struct inode *inode, struct file *filp)
```

```
{
```

```
    printk (KERN_NOTICE "%s: Release Function\n", DEVICE_NAME);
```

```
    return 0;
```

```
}
```

```
/****** Fin Fermeture du périphérique *****/
```

Méthode relative à open

Méthode relative à release

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

- Contrôle d'erreur au niveau matériel.
- Initialisation du périphérique
- Identification du nombre mineur

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

- Libérer ce que open a alloué
- Éteindre la périphérique

II. 4.2 Méthode ioctl :

→ Dans le cadre d'un pilote, la méthode ioctl sert généralement à contrôler le périphérique. Cette fonction permet de passer des commandes particulières au périphérique. Les commandes sont codées sur un entier et peuvent avoir un argument. L'argument peut-être un entier ou un pointeur vers une structure.

→ Prototype:

```
int ioctl(struct inode *inode, struct file *file, unsigned int cmd, unsigned long arg)
```


Si la commande n'existe pas pour le pilote, l'appel système retournera EINVAL.

→ Dans notre cas, la méthode ioctl « **Input_Output_Control** » prend en compte une seule commande, permettant de définir et de donner des informations sur l'aspect physique (géométrique) du périphérique.

```
int Input_Output_Control (struct inode *inode, struct file *filp, unsigned int cmd, unsigned long arg)
{
    long size;
    struct hd_geometry geo; //geometry cylinders and array of sectors.
    mon_Peripherique = filp->private_data;

    switch(cmd) {
        case HDIO_GETGEO:
            /*
             * Get geometry: since we are a virtual device, we have to make
             * up something plausible. So we claim 16 sectors, four heads,
             * and calculate the corresponding number of cylinders. We set the
             * start of data at sector four.
             */
            size = mon_Peripherique->size*(HARDSECT_SECTOR_SIZE/KERNEL_SECTOR_SIZE);
            geo.cylinders = (size & ~0x3f) >> 6;
            geo.heads = 4;
            geo.sectors = 16;
            geo.start = 4;
            if (copy_to_user((void *) arg, &geo, sizeof(geo)))
                return -EFAULT;
            return 0;
        }

    printk (KERN_NOTICE "minibd: IOCTL Function\n");
    return -ENOTTY;
}
```



II. 5 Implémentation de la request function

→ La **request function** (*PériphériqueRW_request*) est la fonction qui assure les opérations de lecture et écriture depuis et vers le périphérique :

```
static void PeripheriqueRW_request(struct request_queue *req_q)
```

```
{
```

```
    struct request *req; /* block I/O request to execute */
```

Extraire la demande de la file à l'aide de la fonction « *elv_next_request* »

```
    while ((req = elv_next_request(req_q)) != NULL) {
```

```
        if (! blk_fs_request(req)) {
```

On utilise « *blk_fs_request* » pour vérifier qu'il s'agit bien d'une request valide.

```
            printk (KERN_NOTICE "Skip non-CMD request\n");
```

```
            end_request(req, 0);
```

Si la request n'est pas valide, on utilise la fonction « *void end_request(struct request *req, int succeeded)* »; et on passe 0 pour l'argument *succeeded*.

```
            /* the request is not successfully completed */
```

```
            continue;
```

```
        }
```

```
        PeripheriqueIO_Transfer(mon_Peripherique, req->sector, req->current_nr_sectors, req->buffer, rq_data_dir(req));
```

La fonction « *PeripheriqueIO_Transfer* » effectue les opérations de lecture/écriture

```
        end_request(req, 1);
```

```
    }
```

```
}
```

Si la request est valide, on passe 1 pour l'argument *succeeded*.

→ C'est dans la fonction « *PeripheriqueIO_Transfer* » que s'effectuent effectivement les opérations de lecture/écriture depuis et vers le périphérique :

```
/* Manipulation des requetes I/O du périphérique */
```

```
static void PeripheriqueIO_Transfer(struct Mon_Peripherique *dev,unsigned long sector,  
unsigned long nsect, char *buffer, int write)
```

```
{  
    unsigned long offset = sector*KERNEL_SECTOR_SIZE;  
    unsigned long nbytes = nsect*KERNEL_SECTOR_SIZE;  
  
    if ((offset + nbytes) > dev->size) {  
        printk (KERN_NOTICE "%s: Beyond-end write (%ld %ld)\n", DEVICE_NAME, offset, nbytes);  
        return;  
    }  
  
    if (write){  
        memcpy(dev->data + offset, buffer, nbytes);  
        printk (KERN_NOTICE "%s: Write with %ld with %ld \n", DEVICE_NAME, offset, nbytes);  
    }  
  
    else{  
        memcpy(buffer, dev->data + offset, nbytes); /* copy to buffer */  
        printk (KERN_NOTICE "%s: Read with %ld with %ld \n", DEVICE_NAME, offset, nbytes);  
    }  
}
```

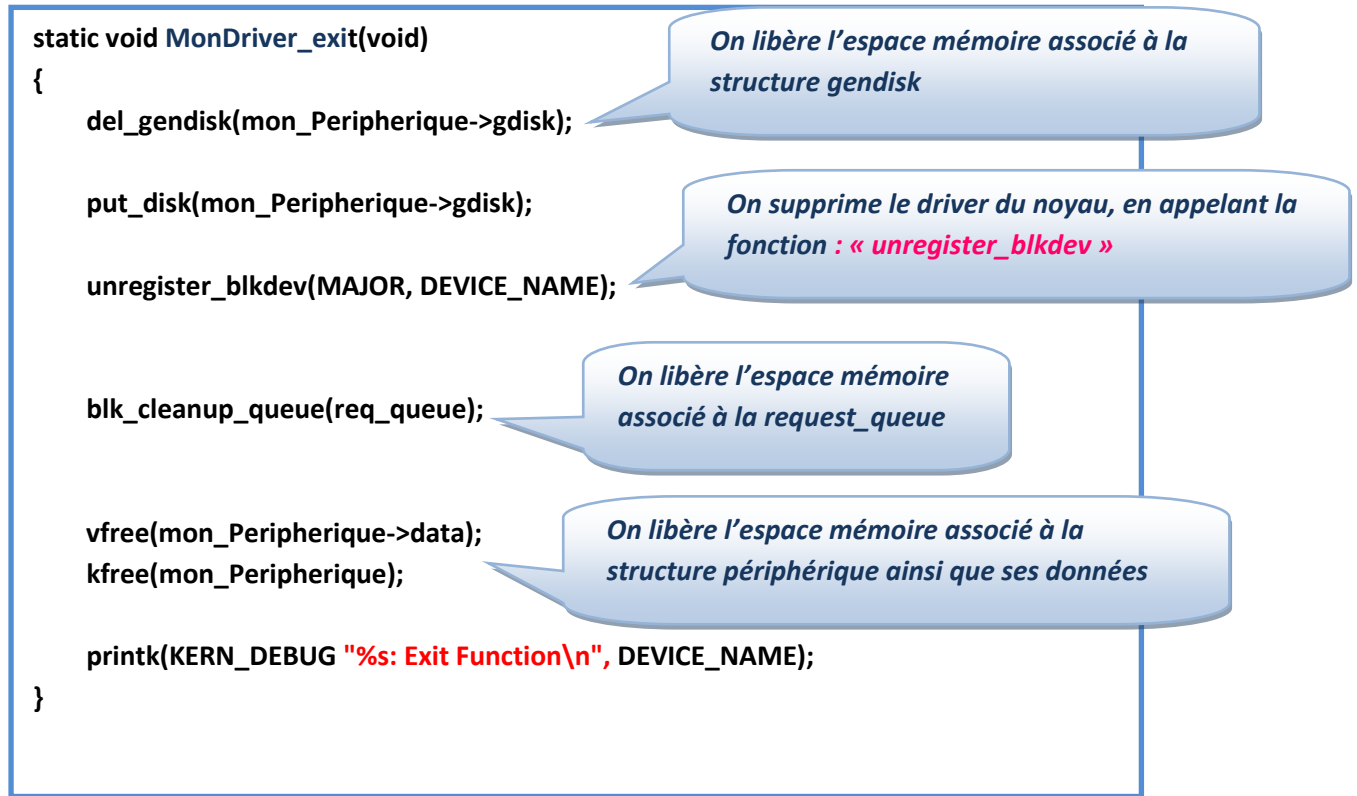
S'assurer qu'on n'effectue pas la copie à la fin du périphérique virtuel.

Manipuler l'opération d'écriture dans le périphérique

Manipuler l'opération de lecture depuis le périphérique

II. 6 Fonction pour le déchargement du module

→ Le rôle de la méthode « **MonDriver_exit** » est tout simplement libérer de ce que « **MonDriver_init** » a alloué.



III. Conclusion :

Une fois le driver compilé et chargé, il faut tester son lien, avec le mode utilisateur.

On a donc bien réussi à réaliser un pilote en mode block permettant de disposer d'un périphérique virtuel offrant une capacité mémoire de 1Ko.