Développement du noyau et pilotes des périphériques

Langage de programmation

- Développé en langage C comme tout système unix.
- Une légère partie est implémentée en assembleur aussi
  - CPU et exceptions
  - Directives critiques des librairies
- Pas d'utilisation du C++
- Tout le code est compilé avec gcc
  - Plusieurs extensions spécifiques du gcc sont utilisées dans le noyau,

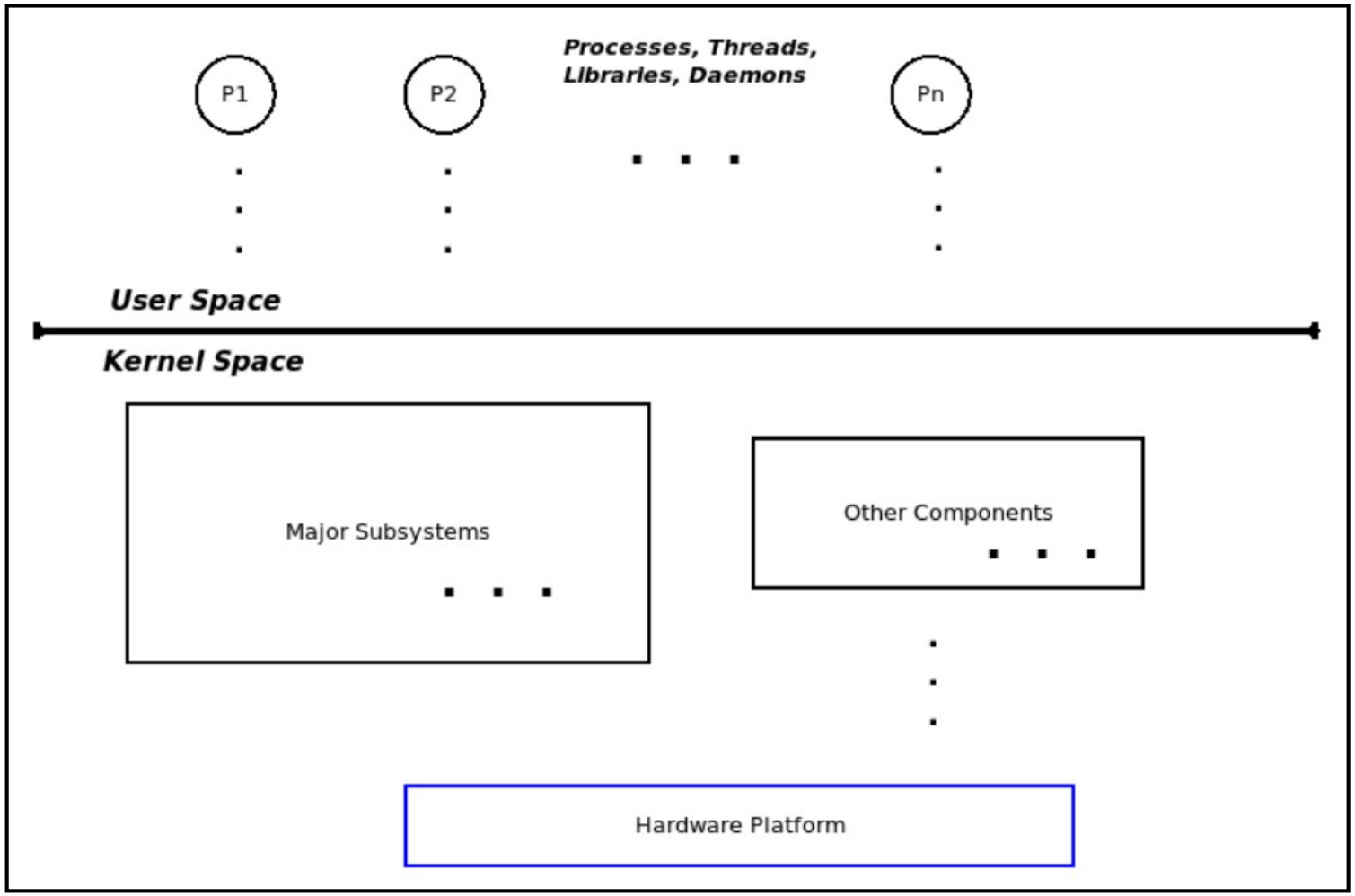
Portabilité

- Le noyau de linux code est conçu pour être portable
- Tout code dehors **`/arch`** devrait être portable
- À cette fin, le noyau fournit des macros et fonctions d'abstraction des détails d'une
  - Mémoire d'accès aux E/S
  - Mémoire barrières à apporter commande garanties si nécessaire
  - API DMA à gérer et actualiser la mémoire cache si nécessaire.
- Ne jamais utiliser nombres flottants dans le code du noyau.
  - Votre code pourrait être exécuté dans un processeur bas niveau ne supportant pas les nombres flottants.

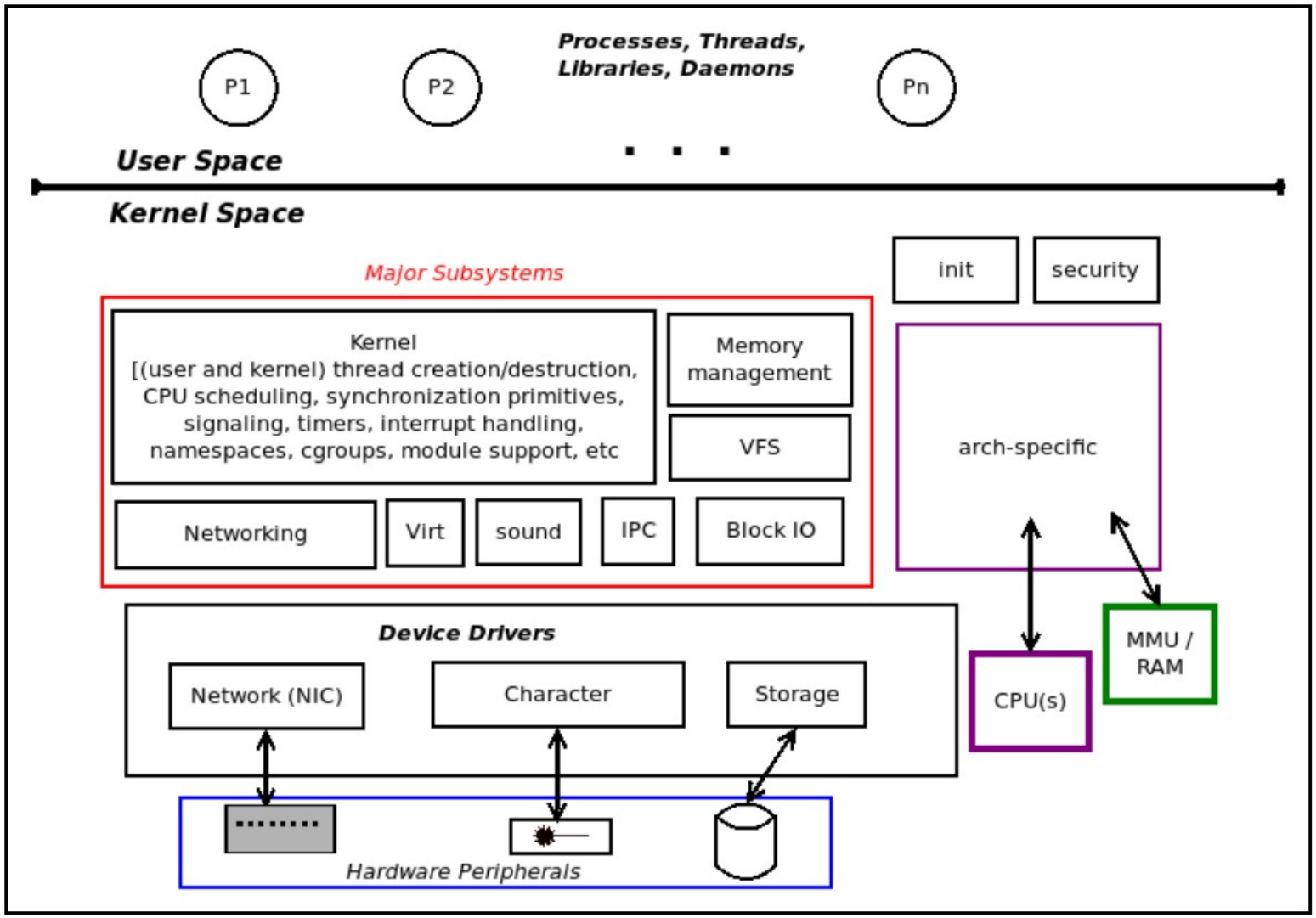
Contraintes de mémoire

- Pas de protection de la mémoire
- le noyau ne tente pas de se remettre des tentatives d’accès illégales à la mémoire. Il se contente d’afficher des message oops (kernel panic).
- Pile de taille fixe (4 ou 8 Ko).
- La mémoire SWAP n’est pas implémentée dans le noyau.

Architecture de base de Linux



Composants de l’espace noyau



Pilotes de l’espace utilisateur

- Des pilotes peuvent être développées dans l’espace utilisateur lorsque :
  - le noyau fournit un mécanisme permettant l’accès au matériel.
  - Le pilote n’a pas besoin d’accéder à la couche basse du noyau tels que les réseaux ou systèmes de fichiers.
  - Le driver n’a pas besoin de multiplexage : une seule application pour accéder au périphérique.
- Certaines classes de périphériques (imprimantes, scanners, ..) sont géré par l’espace utilisateurs et l’espace noyau

Pilotes de l’espace utilisateur

- Avantages:
  - Pas besoin de connaissance du code du noyau.
  - Les pilotes peuvent être écrit en tout langage de programmation.
  - Les pilotes peuvent être propriétaires.
  - Peuvent être utilisés avec SWAP.

Les modules du noyau

Définition : module noyau

- Module noyau / Pilote noyau (Kernel driver)
- Etendre dynamiquement les fonctionnalités du noyau
  - Pas de rédemarrage
  - Plug-and-play

- Le noyau doit être compilé avec l'option **`CONFIG_MODULES=y`**

Avantages des modules

- Les modules facilitent le développement des pilotes sans nécessité de redémarrage: load, test, unload, rebuild,…
- Intéressant pour garder une taille réduite du noyau.
- Utile pour réduire le temps du démarrage: pas de perte de temps pour initialiser les périphériques et fonctionnalités qui ne sont pas nécessaires pour le démarrage.
- **Une fois le module lancé, il dispose des droits absolus du contrôle du système ⇒ seul root peut charger/décharger les modules.**

Dépendances des modules

- Certains modules du noyau peuvent dépendre d'un autre module qui doit être chargé en premier.
  - Exemple: **le module ubifs dépend de ubi et mtd.**
- Les dépendances sont décrites dans :
  - **`/lib/modules/<version-kernel>/modules.dep`**
  - **`/lib/modules/<version-kernel>/modules.dep.bin`**
- Ces fichiers sont générés en exécutant: **make module_install**

Log du noyau

- En chargeant un module, ses informations sont disponibles dans le journal du noyau.

Les messages du noyau sont disponibles via la commande **`dmesg`** ou via le fichier **`/var/log/kern.log`** ou bien **`/dev/kmsg`**

- On peut écrire dans le log du noyau via une instruction root

```
echo ”info débogage” > /dev/kmsg
```

Utilitaires des modules

- **<nom_du_module>** : fichier du nom du module sans l'extension .ko
- **modinfo <nom_module>** : informations des modules dans **/lib/modules** sans le charger (paramètres, licences, description, dépendances,…)
- **insmod <chemin_module>** : charge le module indiqué dans le noyau.
- **modprobe <nom_module>** : charge un module et toutes ses dépendances.
- **lsmod** : liste les modules chargés (contenu du dossier **/proc/modules**)

Utilitaires des modules

- **`rmmod <nom_module>`** : décharger un module
- **`modprobe -r <nom_module>`** : décharger un module ainsi que ses dépendances.

Passage de paramètres aux modules

- Liste des paramètres disponibles pour un module
  - **`modinfo usb-storage`**
- Envoi des paramètres vers un module (exemple **`delay_use`** pour **`usb-storage`**)
  - **`insmod ./usb-storage.ko delay_use=0`**
  - Définir les paramètres dans « **`/etc/modprobe.conf`** » ou les fichiers du **`/etc/modprobe.d/`**
  - Statiquement depuis la CLI : **`usb-storage.delay_use=0`**
- Liste des valeurs des paramètres d'un module chargé

  `/sys/module/<nom>/parameters/`

Chargement d’un module au démarrage

- Les modules à charger au démarrage du système doivent être spécifiés dans le fichier

  `/etc/modules-load.d/<filename>.conf`

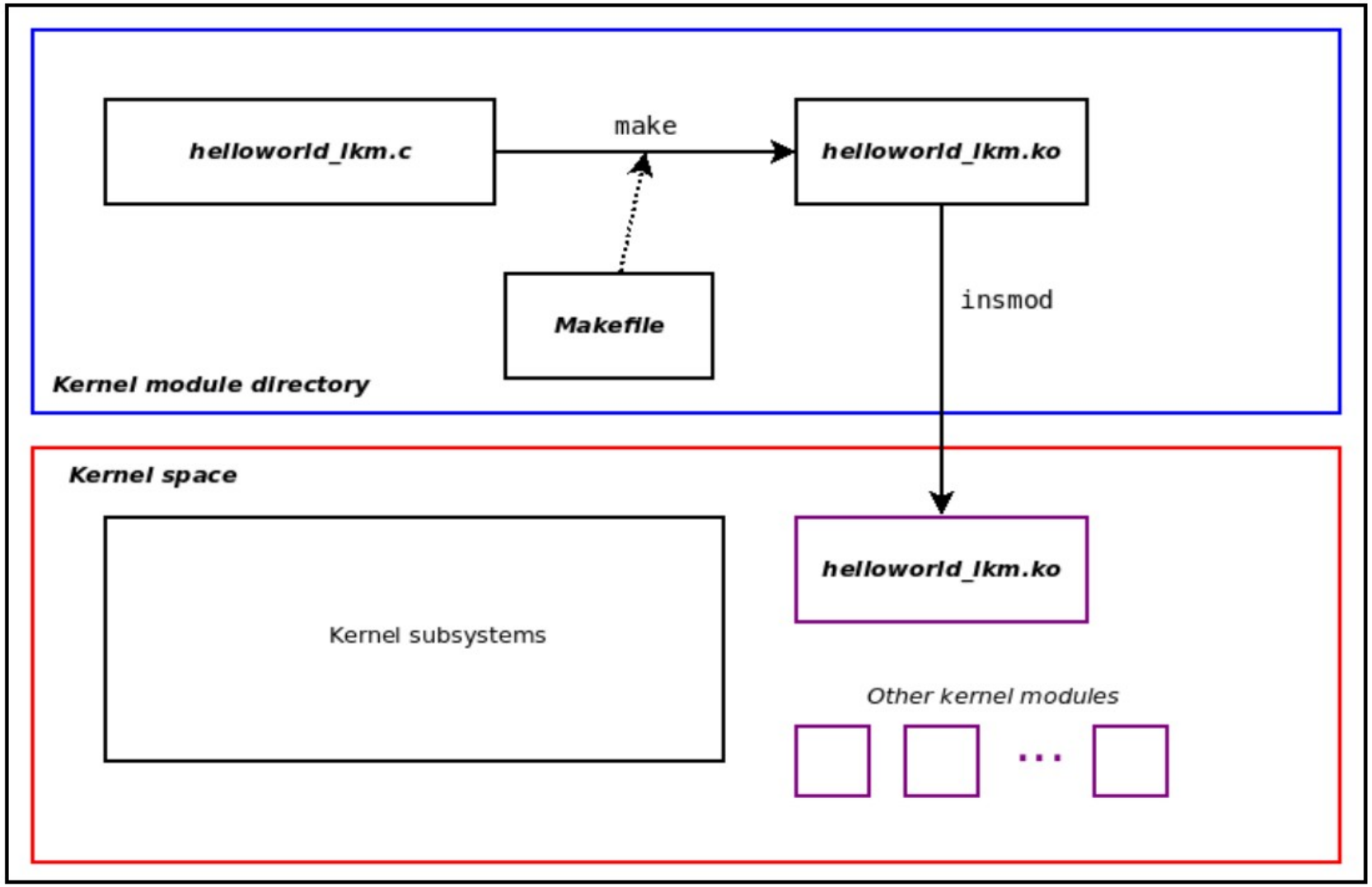
- Les modules à charger au démarrage du systèmes avec leurs dépendances

  `/etc/modprobe.d/<filename>.conf`

Framework LKM (Linux Kernel Module)

- Le framework LKM permet de compiler un morceau de code noyau en dehors de l'arborescence source du noyau, souvent appelé code "hors arborescence"
- Garder une certaine indépendance du module par rapport au noyau
- Insérer ou charger le module dans la mémoire du noyau, de le faire exécuter et d'effectuer son travail, puis de le retirer

Compilation et chargement d'un module



Structure d'un module en C

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

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

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

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

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

Contient les macros définit dans les sources du noyaux

informations récupérable par modinfo

fonction appelée lors du chargement du module

fonction appelée au déchargement du module

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

Compilation d'un module

- On utilise un compilateur **gcc**
- Les fichiers d'entête du noyau doivent être installés sur la machine
- En ligne de commandes

```
gcc -0 -DMODULE -D__KERNEL__ -c module.c
```

- En utilisant Makefile pour compiler deux modules **module1.c** et **module2.c**

```
obj-m += module1.o module2.o
all:
   make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
clean:
   make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Écrire des messages dans le journal du noyau

- La fonction **`printk`** est utilisée pour écrire des messages dans le journal noyau (fichier **`/dev/kmsg`**)

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

- **`fmt`** : chaine de formatage
- C'est le **`printf`** du noyau
- Il est possible de définir un niveau de priorité de 0 à 7 pour le message généré
- Par défaut, le niveau de priorité est définie par
  - **`CONFIG_DEFAULT_MESSAGE_LOGLEVEL`**

Valeurs par défaut de priorité

- Les paramètre par défauts pour niveaux de priorités peuvent être consultés dans le fichier **`/proc/sys/kernel/printk`**
- Exemple

- ❶ Niveau de priorité actuel de la console (tous les messages ayant un niveau inférieur vont apparaître)
- ❷ Niveau par défaut pour les messages
- ❸ Niveau minimum autorisé
- ❹ Niveau maximum autorisé

Niveaux de priorité des messages noyau

- Les niveaux de priorités sont définis dans **`include/linux/kern_levels.h`**

```
#define KERN_SOH "\001"                 /* ASCII Start Of Header */

#define KERN_EMERG KERN_SOH "0"         /* system is unusable */
#define KERN_ALERT KERN_SOH "1"         /* action must be taken immediately */
#define KERN_CRIT KERN_SOH "2"          /* critical conditions */
#define KERN_ERR KERN_SOH "3"           /* error conditions */
#define KERN_WARNING KERN_SOH "4"       /* warning conditions */
#define KERN_NOTICE KERN_SOH "5"        /* normal but significant condition
#define KERN_INFO KERN_SOH "6"          /* informational */
#define KERN_DEBUG KERN_SOH "7"         /* debug-level messages */
```

```
printk(KERN_INFO “Hello Kernel!”);
```

Goodbye `printk`

- Pour les nouveaux modules, il est recommandé d'untiliser les fonctions suivantes :
  - **`pr_<level>(...)`** : à utiliser avec les modules réguliers qui ne représentent pas un périphérique
  - **`dev_<level>(struct device *dev, ...)`** : à utiliser avec les périphériques et qui ne sont pas des périphériques réseaux
  - **`netdev_<level>(struct net_device *dev, ...)`** : à utiliser avec les périphériques réseaux

Goodbye `printk`

`pr_debug`, `pr_devel`	`dev_dbg`	`netdev_dbg`	Used for debug messages. `pr_devel()` is dead code. This means it is not compiled at all, so it's not present in the final binary unless `DEBUG` is defined. The preferred way to go is `pr_debug`.	7
`pr_info`	`dev_info`	`netdev_info`	You can use this for informational purposes, such as start up information at a driver initialization.	6
`pr_notice`	`dev_notice`	`netdev_notice`	This is a notice – nothing serious but notable, nevertheless. It is often used to report security events.	5
`pr_warning`	`dev_warn`	`netdev_warn`	A warning that means nothing serious by itself but might indicate problems.	4
`pr_err`	`dev_err`	`netdev_err`	An error condition, often used by drivers to indicate difficulties with hardware.	3
`pr_crit`	`dev_crit`	`netdev_crit`	A critical condition occurred, such as a serious hardware/software failure.	2
`pr_alert`	`dev_alert`	`netdev_alert`	Something bad happened and action must be taken immediately.	1
`pr_emerg`	`dev_emerg`	`netdev_emerg`	Emergency messages – the system is about to crash or is unstable.	0

Passage de paramètres à un module

- Les paramètres d'un module sont de type :
  - **`bool :`** le type booléen
  - **`short (ushort)`** : entier sur court 2 octets
  - **`int (uint)`** : entier sur 4 octets
  - **`long (ulong)`** : entier long sur 8 octets
  - **`charp`** : pointeur sur une chaine de caractères
- Les paramètres sont déclarés dans le module et on informe le noyau par les macros

```
static type nom;
module_param(nom,type,permissions);
MODULE_PARAM_DESC(nom,desc);
```

Passage de paramètres à un module

- **`module_param(name,type,perm)`** défini dans **`<linux/moduleparam.h>`**
  - **`name`** : nom de la variable utilisée comme paramètre
  - **`type`** : type du paramètre
  - **`perm`** : les permissions d'accès au fichier **`/sys/module/<module>/parameters/<param>`** tels que **`S_IWUSR, S_IRUSR, S_IXUSR, S_IRGRP, S_IWGRP, and S_IRUGO`**
    - **`S_I`** est un suffixe
    - **`R`** : lecture, **`W`** : écriture, **`X`** : exécution
    - **`USR`** = user, **`GRP`** = group, and **`UGO`** = user, group, and others

Passage de paramètres à un module

- Exemple

```
static int valeur=2;
module_param(valeur,int,0644);
MODULE_PARAM_DESC(valeur,”Ce paramètre est de type entier”);
```

Passage de paramètres à un module

- Paramètre de type tableau

```
module_param_array(tableau,type,permissions);
module_param_array(tableau,type,taille,permissions);
```

- Exemple

```
static int myintArray[2] = { -1, -1 };
static int arrSize = 0;
module_param_array(myintArray, int, &arrSize, 0000);
MODULE_PARM_DESC(myintArray, "An array of integers");
```

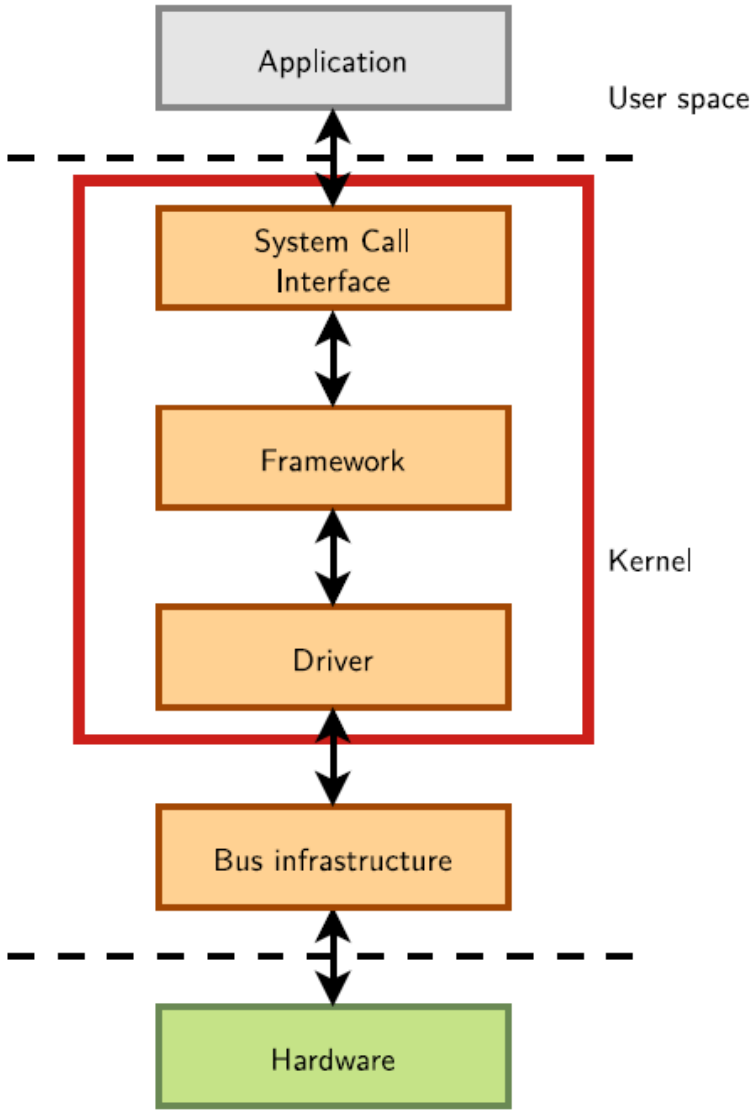
LAB: implémentation d’un module du noyau

- Développement et configuration d’un module personnalisé du noyau pour accéder aux périphériques internes.
- Passage de paramètres à un module.
- Exécution depuis et en dehors de l’arborescence du noyau.

Gestion des périphériques avec Linux

Le noyau et pilotes

- Un pilote est toujours interfacé avec:
  - Un **framework**: permettant la configuration du matériel concerné.
  - **Une infrastructure de bus:** une partie de l'architecture permettant de communiquer avec le matériel.



Types des périphériques

- **Périphériques réseaux:** représentés en tant qu'interfaces réseaux affichées avec **`ifconfig`**.
- **Périphériques blocs**: utilisés pour permettre aux applications de l'espace utilisateur d'accéder aux périphériques de stockage (disques, usb). On peut les trouver dans le dossier **`/dev`**.
- **Périphériques caractères:** utilisés pour permettre aux applications de l'espace utilisateur d'accéder à tous les autres types de périphériques (graphique, son, caméra, communication série,…). On les trouve également dans le dossier **`/dev`**

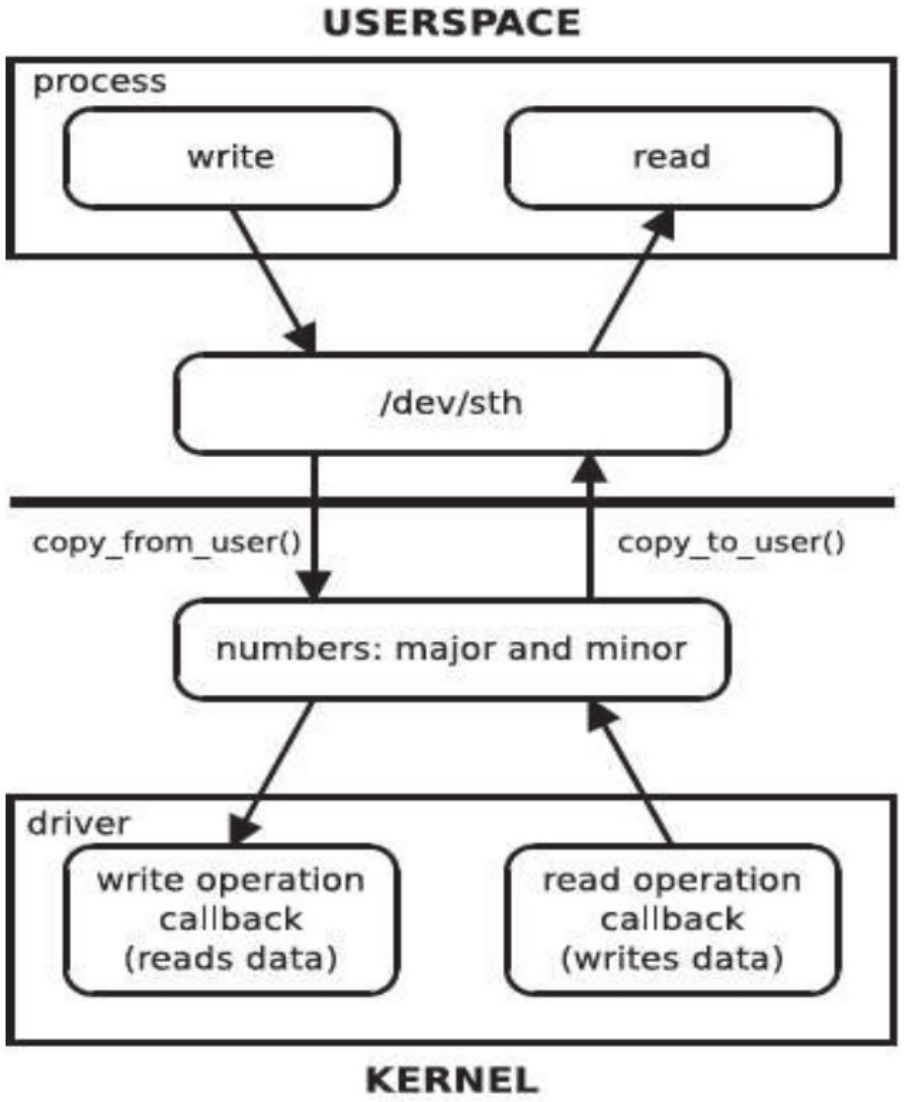
Nombres majeurs et mineurs

- Dans les noyau, tous les périphériques blocs et caractères sont identifiés en utilisant un nombre majeur et un nombre mineur.
  - Le numéro majeur indique typiquement la famille des appareils.
  - Le numéro mineur indique les numéro des appareil (lorsqu'il y a plusieurs ports série par exemple)
- Ils sont statiquement alloués et identiques dans tous les systèmes linux.

Tout est fichier

- Le concept le plus important de Unix est son principe du tout est fichier.
- Il permet aux applications de manipuler tous les objets systèmes via une simple API (**`open, read, write, close,etc.`**)
- Les périphériques sont ainsi représentés par des fichiers spéciaux : **device files**.
  - Un fichier spécial associant un nom de fichier visible dans l'espace utilisateur au triplet (**type, majeur, mineur**) compréhensible par le noyau.
  - Tous les fichiers périphériques sont par convention dans le dossier **`/dev`**

Interaction avec les pilotes



Exemple de fichier spéciaux

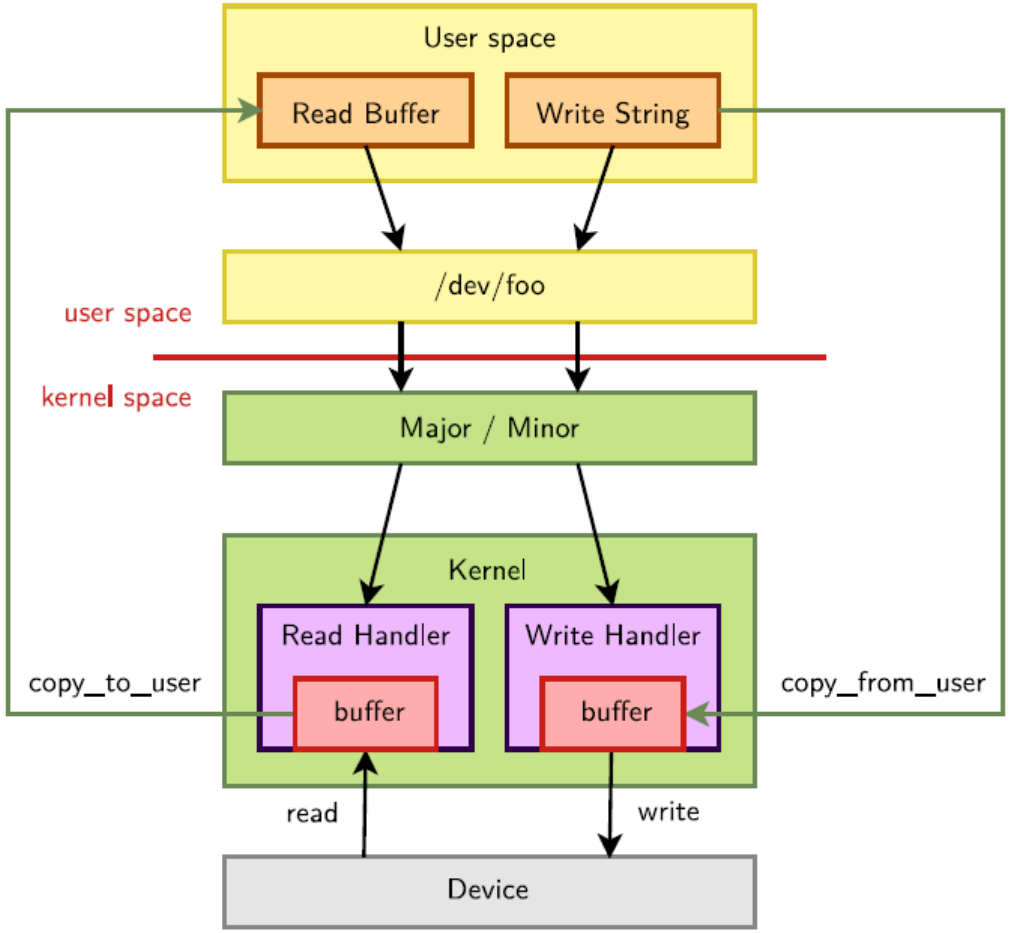
```
$ ls -l /dev/ttyS0 /dev/tty1 /dev/sda1 /dev/sda2 /dev/zero
brw-rw---- 1 root disk    8,  1 2011-05-27 08:56 /dev/sda1
brw-rw---- 1 root disk    8,  2 2011-05-27 08:56 /dev/sda2
crw------- 1 root root    4,  1 2011-05-27 08:57 /dev/tty1
crw-rw---- 1 root dialout 4, 64 2011-05-27 08:56 /dev/ttyS0
crw-rw-rw- 1 root root    1,  5 2011-05-27 08:56 /dev/zero
```

Périphérique en mode caractère

Introduction

- Pour une application, le périphérique caractère est un **fichier**.
- Le pilote du périphérique doit ainsi implémenter les opérations de gestion de fichiers (**open, close, read, write,…)**
- Linux fait en sorte d'appeler les opérations nécessaire lorsque le périphérique est appelé depuis l'espace utilisateur.

Périphériques caractères: architecture complète



Structure de données d'un driver caractère

- Un driver caractère est représenté par une instance de la structure **`cdev`** définie dans **`include/linux/cdev.h`**

```
struct cdev {
        struct kobject kobj;
        struct module *owner;
        const struct file_operations *ops;
        dev_t dev;
        [...]
};
```

Opérations sur les fichiers

```
struct file_operations {
    struct module *owner;
    ssize_t (*read) (struct file *, char __user *,
        size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *,
        size_t, loff_t *);
    long (*unlocked_ioctl) (struct file *, unsigned int,
        unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*release) (struct inode *, struct file *);
    ...
};
```

open() & release()

- `int open(struct inode *i, struct file *f)`
  - Appelée lors de l'ouverture d'un fichier périphérique.
    - **Struct inode:** l'inode de l'objet concerné.
    - **Struct file:** structure créée lors de l'ouverture du fichier contenant des informations d'accès (position actuelle de lecture, mode d'ouverture,…)
- `int release(struct inode *i, struct file *f)`
  - Appelée lors de la fermeture du fichier.
- `ssize_t read(struct file *f, char user *buf, size_t sz, loff_t *off)`
  - Lecture d'une partie du fichier

Structure de données d'un fichier sur le disque

- Au niveau du noyau, un fichier stocké dans un disque est décrit par la structure **`inode`**

```
struct inode {
        [...]
        union {
                struct pipe_inode_info *i_pipe;
                struct cdev  *i_cdev;
                char *i_link;
                unsigned i_dir_seq;
        };
        [...]
}
```

Structure de données d'un fichier ouvert

- Un fichier ouvert associé à un processus avec une descripteur de fichier est décrit par la structure **`file`**

```
struct file {
        [...]
        struct path f_path;
        struct inode *f_inode;
        const struct file_operations *f_op;
        loff_t f_pos;
        void *private_data;
        [...]
}
```

Identification d'un périphérique

- Le noyau identifie un périphérique à travers une valeur entière non signée de 32 bits : le type **`dev_t`**
  - 12 bits du poids fort : le numéro majeur
  - 20 bits restants : le numéro mineur
- Le macro **`MKDEV(Majeur,Mineur)`** permet de construire l'identifiant du périphérique
- Quelques macros définis dans **`include/linux/kdev_t.h`**

```
#define MINORBITS 20
#define MINORMASK ((1U << MINORBITS) - 1)

#define MAJOR(dev) ((unsigned int) ((dev) >> MINORBITS))
#define MINOR(dev) ((unsigned int) ((dev) & MINORMASK))
#define MKDEV(ma,mi) (((ma) << MINORBITS) | (mi))
```

Allocation du numéro majeur

- Deux manières pour allouer un numéro majeur
  - Allocation (enregistrement) statique : connaitre au préalable le numéro majeur
  - Allocation dynamique : le noyau alloue à la volée le numéro majeur

Allocation statique du numéro majeur

- L'allocation statique s'effectue à travers la fonction **`register_chrdev_region`**

```
int register_chrdev_region(dev_t first, unsigned int count,char *name);
```

`first` : le numéro majeur

`count` : nombre de périphériques mineurs

`name` : nom associé au périphérique

Retourne 0 en cas de succès

☞ Risque de conflit si le numéro majeur est déjà associé à un autre périphérique

- L'opération inverse

```
void unregister_chrdev_region(dev_t first, unsigned int count);
```

Initialisation d’un périphérique

- L’initialisation spécifie les opérations sur le fichier associé au périphérique

```
void cdev_init(struct cdev *cdev,const struct file_operations *fops);
```

`cdev` : pointeur sur l’instance de périphérique caractère à créer

`fops` : les opérations fichier implémentées

Ajout d’un périphérique caractère

- Ajouter un périphérique caractère au noyau

```
int cdev_add (struct cdev * p, dev_t dev, unsigned count);
```

`p` : pointeur sur l’instance de périphérique

`dev` : l’identifiant du périphérique

`count` : nombre de numéros mineurs

Retourne 0 en cas de succès, sinon une valeur négative

- Opération inverse

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

Étapes pour initialiser un périphérique en mode caractère

```
/* Define mayor number */
#define MY_MAJOR_NUM XXX

dev_t dev = MKDEV(MY_MAJOR_NUM, 0); /* Get first device identifier */

/* Allocate a number of devices */
ret = register_chrdev_region(dev, 1, "my_char_device");
if (ret < 0) {
        pr_info("Unable to allocate mayor number %d\n", MY_MAJOR_NUM);
        return ret;
}

/* Initialize the cdev structure and add it to kernel space */
cdev_init(&my_dev, &my_dev_fops);
ret= cdev_add(&my_dev, dev, 1);
if (ret < 0) {
        unregister_chrdev_region(dev, 1);
        pr_info("Unable to add cdev\n")
         return ret;
}
```

Échange de données avec l’espace utilisateur

- Le code du noyau n’est pas autorisé à accéder à l’espace utilisateur, il faut utiliser une fonction de type :**memcpy()**
  - Cette procédure n’est pas prise en compte par toutes les architectures.
  - Si l’adresse indiquée est invalide, ça causera une erreur de segmentation.
- Pour garder le code du noyau portable et bien sécurisé, il faut procéder proprement en utilisant des fonctionnalités du noyau.

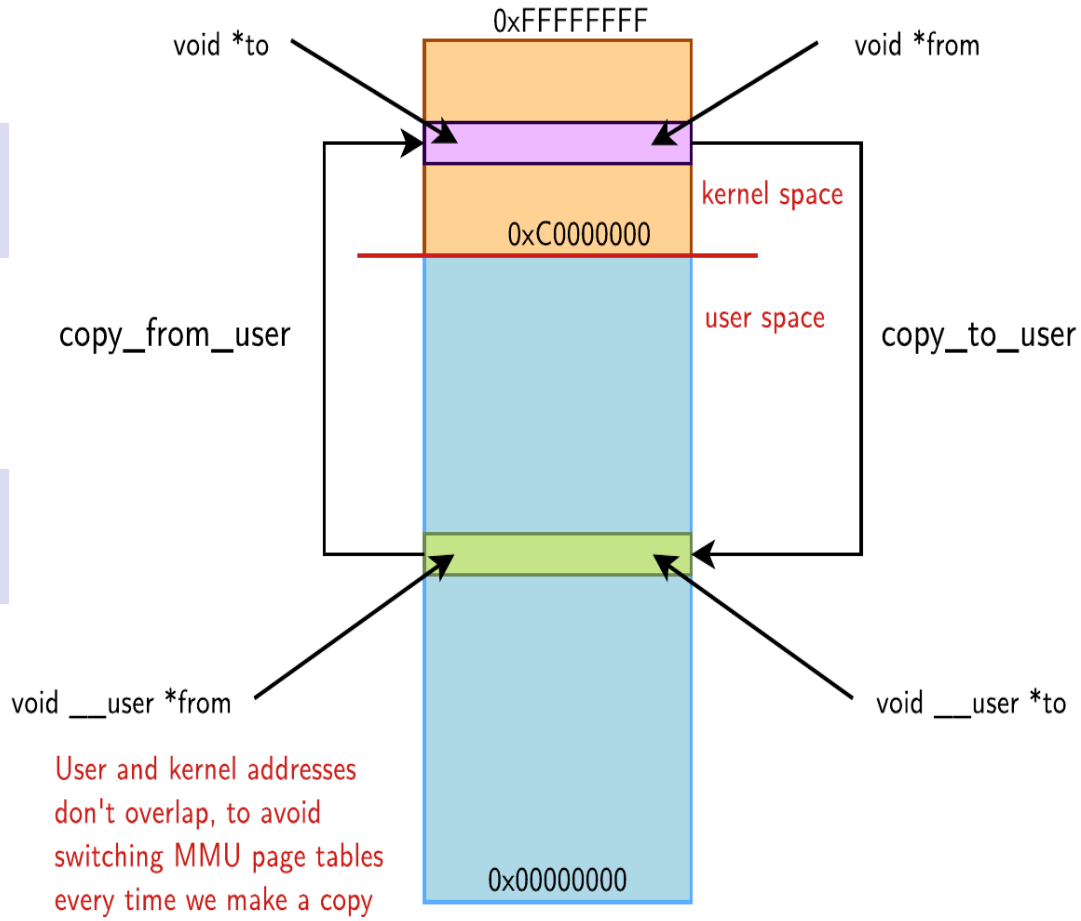
Échange de données avec l'espace utilisateur

- Deux macros permettent l'échanges de valeurs uniques
  - **`get_user(v, ptr);`**
    Le contenu du pointeur **`ptr`** dans l'espace utilisateur est affecté à la variable du noyau **`v`**
  - **`put_user(v, ptr);`**
    Le contenu de la variable du noyau **`v`** est affecté à l'espace utilisateur pointé par **`ptr`**
  - Les deux macros retournent **`0`** en cas de succès, sinon **`-EFAULT`**

Échange des données avec l'espace utilisateur

```
unsigned long copy_to_user(void __user *to,const
void *from, unsigned long n)
```

```
unsigned long copy_from_user(void *to,const void
__user *from, unsigned long n)
```



Opérations sur les fichiers : **`write()`**

- La fonction **`write()`** est utilisée pour l'envoie de données au périphérique. Elle est appelée à chaque fois que l'utilisateur envoie des données au périphérique

```
ssize_t (*write) (struct file *filp, const char __user *buf, size_t count,
                  loff_t *pos);
```

`filp` : pointeur de fichier
`buf` : tampon de données provenant de l'espace utilisateur
`count` : la taille de données de la requête de transmission
`pos` : indique la position à partir de laquelle les données doivent être écrites dans la fichier

La valeur de retour est le nombre d'octets écris dans le fichier.

Opérations sur les fichiers : **`read()`**

- La fonction **`read()`** est utilisée pour récevoir, dans l'espace utilisateur, les données envoyées par le périphérique

```
ssize_t (*read) (struct file *filp, const char __user *buf, size_t count,
                 loff_t *pos);
```

`filp` : pointeur de fichier
`buf` : tampon de données pour l'envoie de données vers l'espace utilisateur
`count` : la taille de données de la requête de transmission
`pos` : indique la position à partir de laquelle les données doivent être écrites dans la fichier

La valeur de retour est le nombre d'octets lus.

Opérations sur les fichiers : ioctl ()

- La fonction **`ioctl()`** est utilisée pour envoyer des commandes spécifiques au périphérique sous forme d'appel système (arrêt, initialisation, configuration, suspension, etc.)

```
long unlocked_ioctl(struct file *filp, unsigned int cmd, unsigned long arg);
```

`filp` : pointeur de fichier
`cmd` : tampon de données pour l'envoie de données vers l'espace utilisateur
`arg` : la taille de données de la requête de transmission

La valeur de retour est le nombre d'octets lus.