Le Virtual Filesystem Switch (VFS)

Partie 1 : Opérations sur les Fichiers et Pilotes Caractères

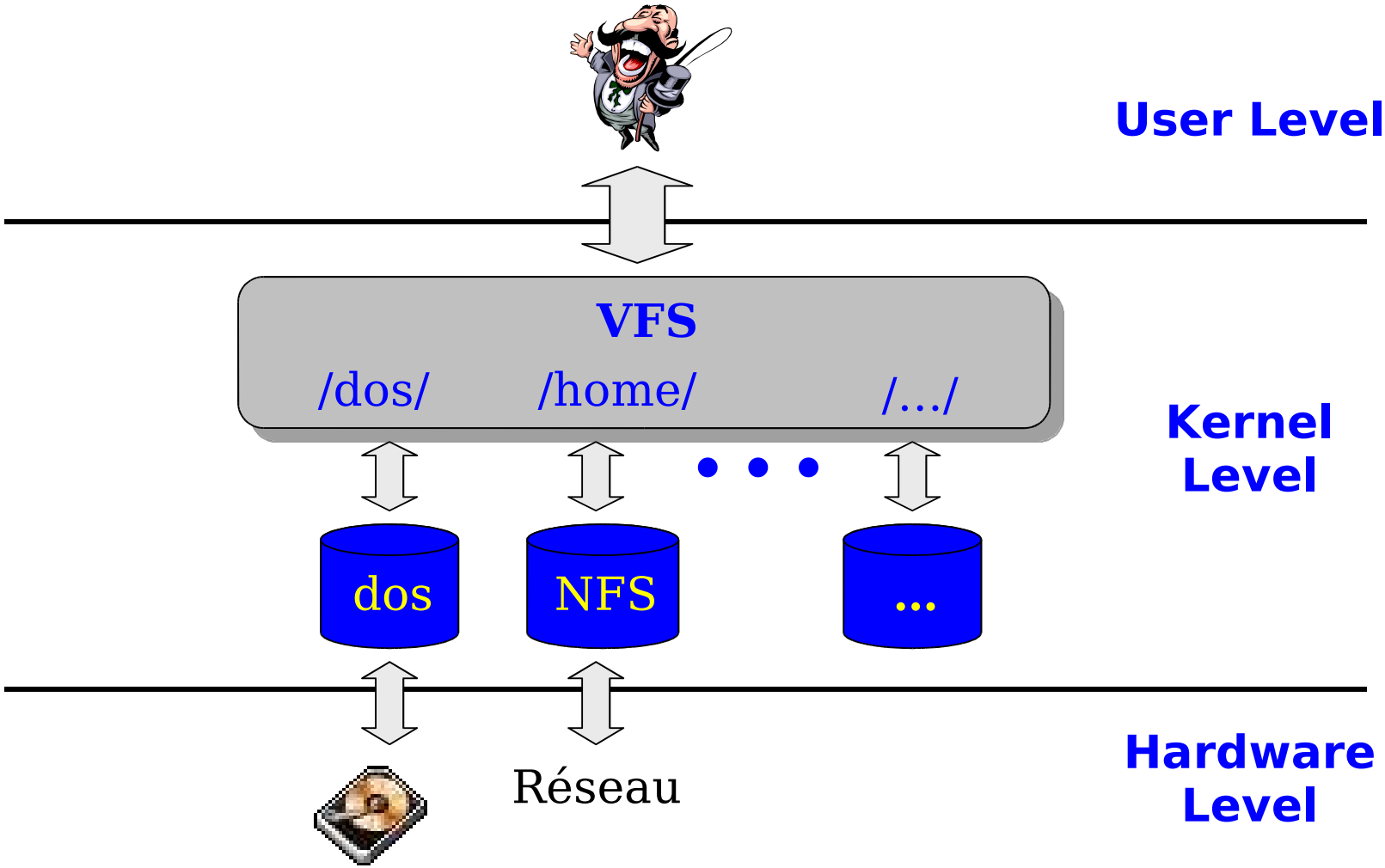
Définitions

- Un système de fichiers est :
  - **Structures de données** et d'**algorithmes**
    - Organisation des fichiers et répertoires
      - Quelles méta-données et comment les organiser ?
    - Gestion de l'espace du support physique
      - Comment rendre les E/S aussi performantes que possible ?
    - **Sémantique des opérations sur les fichiers** et répertoires
      - Qu'est supposée faire une "écriture" ?
  - Généralement homogène
    - Tous les fichiers et répertoires ont le même comportement

Définitions

- Le VFS est :
  - Point de vue utilisateur
    - Une interface unifiée vers différents SF
      - ext2/ext3,
      - Reiser,
      - iso9660 (cdrom),
      - NFS, ...
  - Point de vue architecte OS
    - Un moyen de factoriser le code
      - fiabilité/robustesse + allègement du code
  - Point de vue developpeur pilotes
    - Une API « classique »
    - Des services fournis par défaut

Vue Schématique de l'Architecture



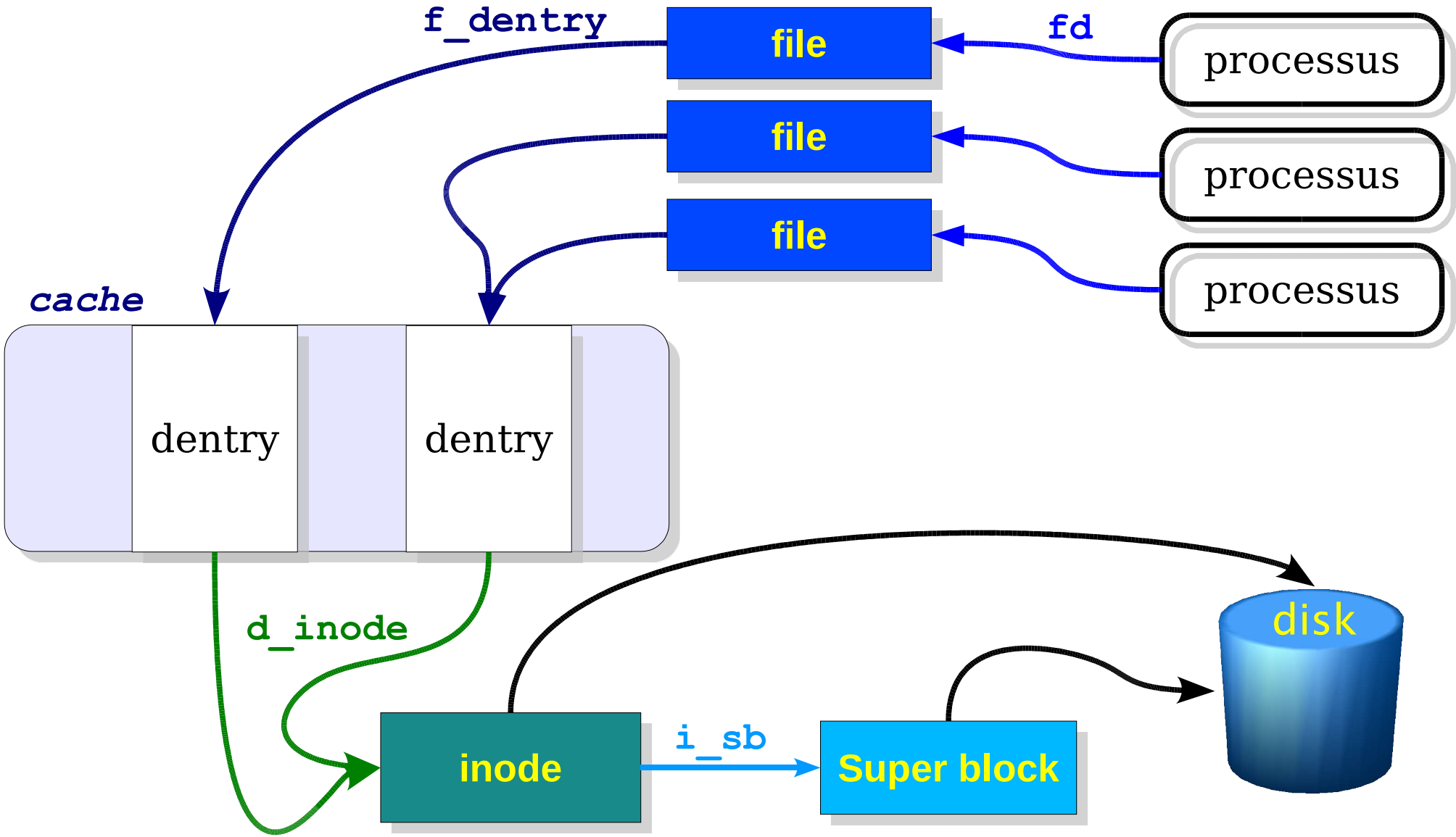
VFS: une Conception Orientée Objet

- Conception Objet en Langage C ?
  - Objet = struct(ure) contenant :
    - Des attributs (données)
    - Des méthodes (pointeurs vers des fonctions)
- Exemple d'objet : la structure **`file`**
  - Associée à un fichier ouvert
  - **Attributs** (quelques)
    - uid/gid du(des) processus ayant ouvert le fichier
    - mode : attributs d'ouverture (R/W/Append, ...)
    - pos : position courante (prochaine lecture/écriture)
  - **Méthodes** :
    - read, write, llseek, ioctl, mmap, poll, flush, open, ...

Les Principaux Objets de VFS

- Super Block : Informations globales sur le support
  - Etat courant de la partition
  - Définie dans `<linux/fs.h>`
- I-node : (méta)Informations sur le fichier
  - Permissions, dates, adresse des blocks de données, ...
  - Définie dans `<linux/fs.h>`
- D-entry : informations sur le lien (chemin vers le fichier)
  - Essentiellement pour des raison de perf (cache)
  - Définie dans `<linux/dcache.h>`
- File : informations sur un fichier ouvert
  - Définie dans `<linux/fs.h>`

Interactions entre Processus et VFS



La Structure 'file' (Fichier Ouvert)

Liens vers autres structures

```
struct file {
   struct list_head     f_list;
   struct dentry        *f_dentry;
   struct vfsmount      *f_vfsmnt;
```

Une de ces 3 listes:
- free_list
- anon_list (pas de sb)
- sb->s_files

Contrôle du fichier (opérations d'E/S)

```
   struct file_operations  *f_op;
```

Etat courant du fichier

```
   atomic_t             f_count;
   unsigned int         f_flags;
   mode_t               f_mode;
   loff_t               f_pos;
   unsigned long        f_reada, f_ramax, f_raend, f_ralen,
f_rawin;
   struct fown_struct   f_owner;
   unsigned int         f_uid, f_gid;
   int                  f_error;
```

Opérations sur Fichier Ouvert (struct file_o8s)

- Une pour chaque appel système sur un fichier ouvert

  + quelques « spécialités exotiques » ...

  - llseek(file,offset,origin)
  - read(file,buf,count,offset)
  - write(file,buf,count,offset)
  - readdir(file,dirent,filldir_fn)
  - poll(file,poll_table)
  - ioctl(inode,file,cmd,arg)
  - mmap(file,vma)
  - open(inode,file)
  - flush(file)
  - release(inode,file)
  - fsync(file,dentry)
  - fasync(fd,file,on)
  - lock(file,cmd,file_lock)
  - readv (file,vector,count,offset)
  - writev (file,vector,count,offset)
  - sendpage(file,page,offset, size,pointer,fill)
  - get_unmapped_area(file,addr, len,offset,flags)

Traitement supplémentaire en fin d'ouverture

Lien Entre Appel Système et Opération

- En mode utilisateur
  - Processus invoque operation (open(2), read(2), ...)
  - libc prépare bascule en mode noyau
  - déclenchement bascule ...
- En mode noyau
  - point d'entrée = sys_xxx (sys_open(), sys_read(), ...)
  - sys_xxx() fait des vérifications d'usage
  - sys_xxx() invoque le traitement correspondant du VFS
  - retour > 0 : ok
  - retour < 0 : -ERROR (ex : -EPERM)
  - bascule en mode utilisateur
- libc (mode ut) si retour < 0 : errno = retour, retour -1

Déroulement d'une Ouverture de Fichier

- sys_open(filename,flags,mode)
  1. getname() : copie filename depuis espace utilisateur
  2. get_unused_fd() : recherche decripteur libre
  3. filp_open(pathname,flags,mode)
     1. open_namei(pathname,flas,&nameidata)
        1. ...
        2. ...
     2. dentry_open(dentry,mnt,flags)
        1. ...
        2. ...
        3. ...
  4. ref vers struct file dans contexte processus (files->fd[fd])
  5. retour fd (ou -ERRNO)

Déroulement d'une ouverture de fichier

filp_open(pathname,flags,mode)

1. open_namei(pathname, flags, &nameidata)
   1. path_lookup(pathname, flags, &nameidata) :
      place dans nameidata d_entry correspondant à pathname
      (ou répertoire parent si création)
   3. nombreuses vérifications (verrous, permissions, ...)
2. dentry_open(dentry, mnt, flags)
   1. alloc et init de la struct file
      f_flags, f_mode, f_dentry, f_pos, f_count=1 ...
   2. **f_op = d_entry->d_inode->i_fop**
   3. struct file chaîné dans liste superblock

Déroulement d'une Lecture/Ecriture

- Fonctionnement similaire
- Ex: sys_read(fd,buf,count)
  1. fget(fd) -> struct file + compteur utilisation (f_count++)
  2. Opération autorisée (f_mode) ?
  3. locks_verify_area() -> mandatory lock ?
  4. invoque file->f_op->read()
     1. lecture depuis périphérique
     2. copie vers espace utilisateur
     3. met à jour f_pos
  5. fput() : file->f_count --
  6. Retourne nb octets effectivement transférés (ou -ERRNO)
     - 0 = EOF

Déroulement d'une Fermeture de Fichier

- sys_close(fd) :
  1. Retrouve struct file associée à fd à partir du contexte
  2. Annule l'entrée de la struct file dans contexte
  3. invoque filp_close()
     1. invoque file->f_op->flush()
     2. Relâche les éventuels verrous
     3. invoque fput() (f_count --)
  4. retourne code retour de flush()
- Remarque : Effets de bord possibles
  - Si f_count = 0 : déclenche release(inode,file)
    - Si nb lien = 0 (car unlink() après ouverture) : destruction physique du fichier

Définition Partielle des f_ops

- Principe de surcharge
  - L'absence de définition d'une méthode est courante
    - Traitement minimal assuré par VFS
    - Seule la définition owner est obligatoire
  - Méthodes non définies = pointeur NULL dans struct file_operations
    - Déclaration « étiquetée »
      - Astuce pour ne pas se soucier de l'ordre des éléments
      - Exemple :

```
struct file_operations chardev_fops = {
  llseek: chardev_llseek,
  read:   chardev_read,
  write:  chardev_write,
  owner:  THIS_MODULE,
};
```

équiva lent à:
SET_MODULE_OWNER(&chardev_fops)

Pilotes de Périphériques Caractères

- Cas particulier de fichiers du VFS
  - Fichier spécial :
    - dont le « comportement » est programmé spécialement
      - Programme = traitement spécifique associée à chaque opération d'E/S
      - file_operations : open(), read(), write(), ioctl()
        - Autres que celles prévues pour les fichiers normaux
    - Le programme est choisi à partir du **major number**
  - Pilote de périphérique
    - Code spécifique dans le noyau (module ou liaison statique)
    - Init : Associe son programme à un major
      - Fournit une struct file_operations à utiliser
      - Lors du boot ou du chargement du module
        - Conservation d'état possible entre 2 chargements du module

Numéros Major/Minor d'un Périphérique

- Chaque périphérique est associé à un couple (major,minor)

```
toto_$ ls -al /dev/zero /dev/null /dev/ttyS[0-4]
crw-rw-rw-    1 root     root       1,   3 avr 11  2002 /dev/null
crw-rw----    1 root     uucp       4,  64 avr 11  2002 /dev/ttyS0
crw-rw----    1 root     uucp       4,  65 avr 11  2002 /dev/ttyS1
crw-rw----    1 root     uucp       4,  66 avr 11  2002 /dev/ttyS2
crw-rw----    1 root     uucp       4,  67 avr 11  2002 /dev/ttyS3
crw-rw-rw-    1 root     root       1,   5 avr 11  2002 /dev/zero
```

  - Le major permet au VFS d'identifier le pilote
  - Le minor permet au pilote d'identifier le périphérique

Aiguillage vers le Code selon major/minor

- Un même pilote peut prendre en charge plusieurs périphériques !
  - Le VFS ne s'occupe pas vraiment du minor
    - Se contente d'aiguiller vers un pilote (major)
  - Lors d'une E/S, le pilote peut "remonter" jusqu'à l'inode
    - Aiguillage "interne" vers le périphérique
      - A la charge du (programmeur du) pilote
      - major et minor sont combinés dans le champ i_rdev (inode)
        - Manipulation à l'aide de macros :
          - MAJOR(dev),
          - MINOR(dev),
          - MKDEV(major,minor)

Enregistrement d'un Pilote Caractère

- Par exemple dans init_module() :
  - register_chrdev(major,name,&file_ops)
- Problème : quel major choisir pour un nouveau pilote ?
  - Le nombre de major est très limité : 256
    - Limite corrigée dans noyau 2.6
  - Solution 1 : majors réservés pour expé
    - 60-63, 120-127, 240-244
    - Pratique pour phase devel, mais inadapté pour diffusion
  - Solution 2 : réclamer un major attitré
    - Possible, mais beaucoup de concurrence
    - Uniquement lorsque état du devel. est avancé
  - Solution 3 : Allocation dynamique

Utilisation d'un « Major Dynamique »

- register_chrdev() avec paramètre major = 0
  - Résultat : major attribué si >0, erreur sinon
  - Comment l'utilisateur (root) peut-il savoir quel major a été affecté ?
    - indispensable pour exécuter mknod avec bons arguments
    - Solution : trace (printk) ou /proc/devices
  - Problème : le fichier spécial de périphérique ne peut-être créé qu'après initialisation
    - Cad après insmod()
    - Solution robuste : script ad-hoc
    - Solution optimiste en cas de de/re-chargements successifs
      - le noyau a tendance à donner le même major dynamique

Exemple de Makefile pour Major Dynamique

```
CFLAGS = ...
MODS_DIR=/lib/modules/$(shell uname -r)/$(shell basename `pwd`)
MODS= chardev.o

all: chardev
mods: $(MODS)
$(MODS): $(MODS:.o=.c)

$(MODS_DIR):
    mkdir $@
$(MODS_DIR)/chardev.o install: $(MODS_DIR) $(MODS)
    cp $(MODS) $(MODS_DIR)
    depmod -a
uninstall:
    rm -rf $(MODS_DIR)
    depmod -a
ins: $(MODS_DIR)/chardev.o
    modprobe chardev
del:
    modprobe -r chardev
    rm -f chardev
chardev: ins
    mknod chardev c $(shell awk '$$2 == "chardev" {print $$1;}' /
proc/devices) 0
```

Contrôle de Périphérique avec *ioctl*

- Fonction à tout faire pour
  - envoyer des séquences de contrôle
  - Récupérer des informations du device
  - Ou les deux en même temps (opération atomique)
- Prototype utilisateur
  - `int ioctl(int fd, int cmd, ...)`

equivalent à `char *argp`

- Méthode de la structure file_operations
  - `int (*ioctl)(inode,file,cmd,arg)`
    - cmd : unsigned int désignant la commande
      - traitement typiquement dans un switch/case
    - arg : paramètre (optionnel)
      - unsigned long utilisé pour recevoir et/ou retourner des données par reférence

Choix du Numéro de Commande

- Solution 1 : Choisir un numéro quelconque
  - Ca fonctionne (à quelques exceptions près)
  - Mais ce n'est pas recommandé (pas très social)
    - Chaque commande doit être reservée exclusivement à un type de device
- Solution 2 : Utiliser le schéma de numérotation Linux

Schéma de Numérotation des Commandes ioctl Linux

- Choisir un **numéro magique** de device sur 16 bits
  - Documentation/ioctl-number.txt : Liste des numéros a priori encore disponibles
- Choisir un **numéro de séquence** pour la commande (8 bits)
- Préciser la **direction** de transfert du paramètre
  - lecture, écriture, aucun ou les deux
  - sens : user -> noyau

Construction du Numéro de Commande

- Macros définies dans <asm/ioctl.h>
  - **_IO**(mag,seq) : sans direction
  - **_IOR**(mag,seq,type) : direction kernel -> user
  - **_IOW**(mag,seq,type) : direction user -> kernel
  - **_IOWR**(mag,seq,type) : deux directions
    - type est informel : permet d'expliciter la taille des données
      - Vérifications éventuelles par le pilote

Extraction des Numéros de Commande

- Macros définies dans <asm/ioctl.h>
  - **_IOC_DIR**(num)
    - Renvoie combinaison binaire de _IOC_READ et _IOC_WRITE
    - Ex : _IOC_DIR(num) & (_IOC_READ|_IOC_WRITE)
  - **_IOC_TYPE**(num) -> numéro magique
  - **_IOC_NR**(num) -> numéro séquence
  - **_IOC_SIZE**(num) -> taille attendue des données (sizeof)

Erreurs et Code de Retour

- Impossible de retourner une valeur négative en cas de succès
  - Obligation d'utiliser le paramètre, par reférence
- Les erreurs que doit (devrait) détecter le pilote
  - `-ENOTTY` : commande non supportée
  - `-EPERM` : Droits insuffisants
    - Le pilote peut faire des vérifications supplémentaires
      - Par exemple tester les capabilities (<linux/capability.h>)
  - `-EFAULT` : référence invalide
    - Le pilote **doit** vérifier la validité de l'adresse en mémoire utilisateur !
      - Faire confiance à l'utilisateur est une faute grave
      - La zone mémoire peut-être valide mais indisponible (swap ...)

Utilisation et Vérification des Adresses en Espace Util.

- Vérification (<asm/uaccess.h>)
  - `int access_ok(type, void *addr, u. long size)`
    - **type** : `VERIFY_READ` ou `VERIFY_WRITE`
      - `VERIFY_WRITE` inclut `VERIFY_READ`
    - retour : 1 si ok, 0 si erreur
- Utilisation (transfert) des données
  - Fonctions optimisées pour types de base
    - `{put|get}_user(datum,ptr)` : vérification forte
      - Redondant avec access_ok
    - `__{put|get}_user(datum,ptr)` : vérification faible
      - Sans access_ok
    - Retournent 0 ou -EFAULT
  - Fonctions classiques
    - `copy_to_user()/copy_from_user()`