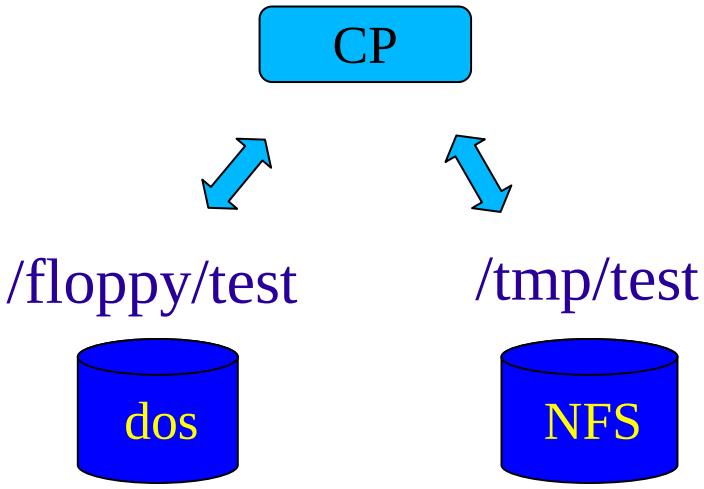
Le Virtual Filesystem Switch (VFS)

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

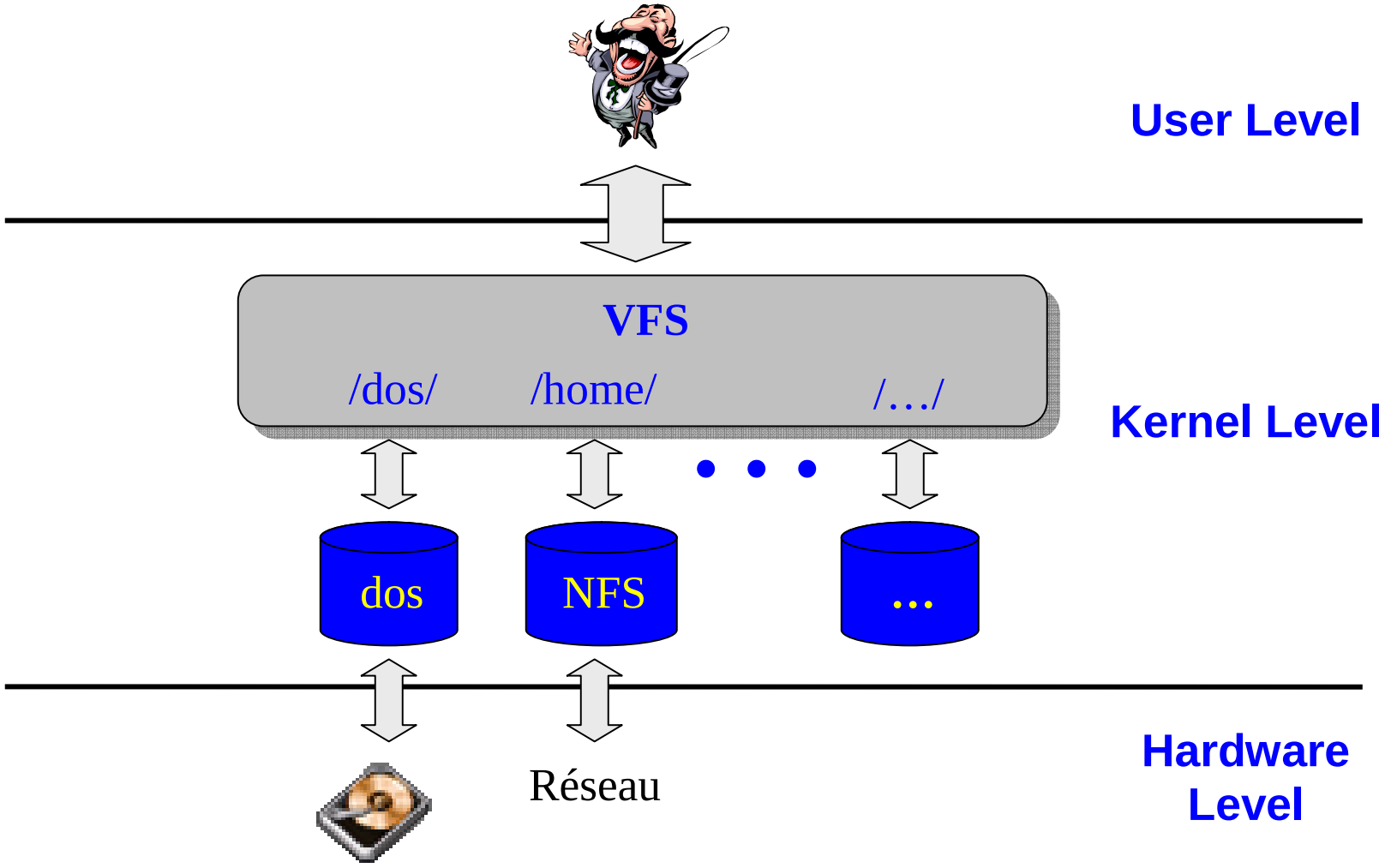
Copie de fichiers




```
inf = open ("/floppy/test", O_RDONLY,0);
outf = open ("/tmp/test",
   O_WRONLY|O_CREATE|O_TRUNC,0600);
do {
   l = read(inf, buf, 4096);
   write(outf, buf, l);
} (while l);
 close(outf);
 close(inf);
```

- Pour le processus utilisateur, les détails d'implémentation du FS ne sont pas importants

Vue Schématique de l'Architecture



Modèle VFS

- L'idée derrière le VFS est de fournir un modèle commun pour représenter tous les systèmes de fichier
- Basé sur le modèle de fichier traditionnel des Unix
  - Permet d'avoir de bonnes performances dans le cas général
- Chaque système de fichier doit fournir une implémentation utilisant le modèle VFS
  - Certains FS sont très différents du VFS
  - Les informations manquantes sont construites au vol

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 développeur pilotes
    - Une API « classique »
    - Des services fournis par défaut

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, ...

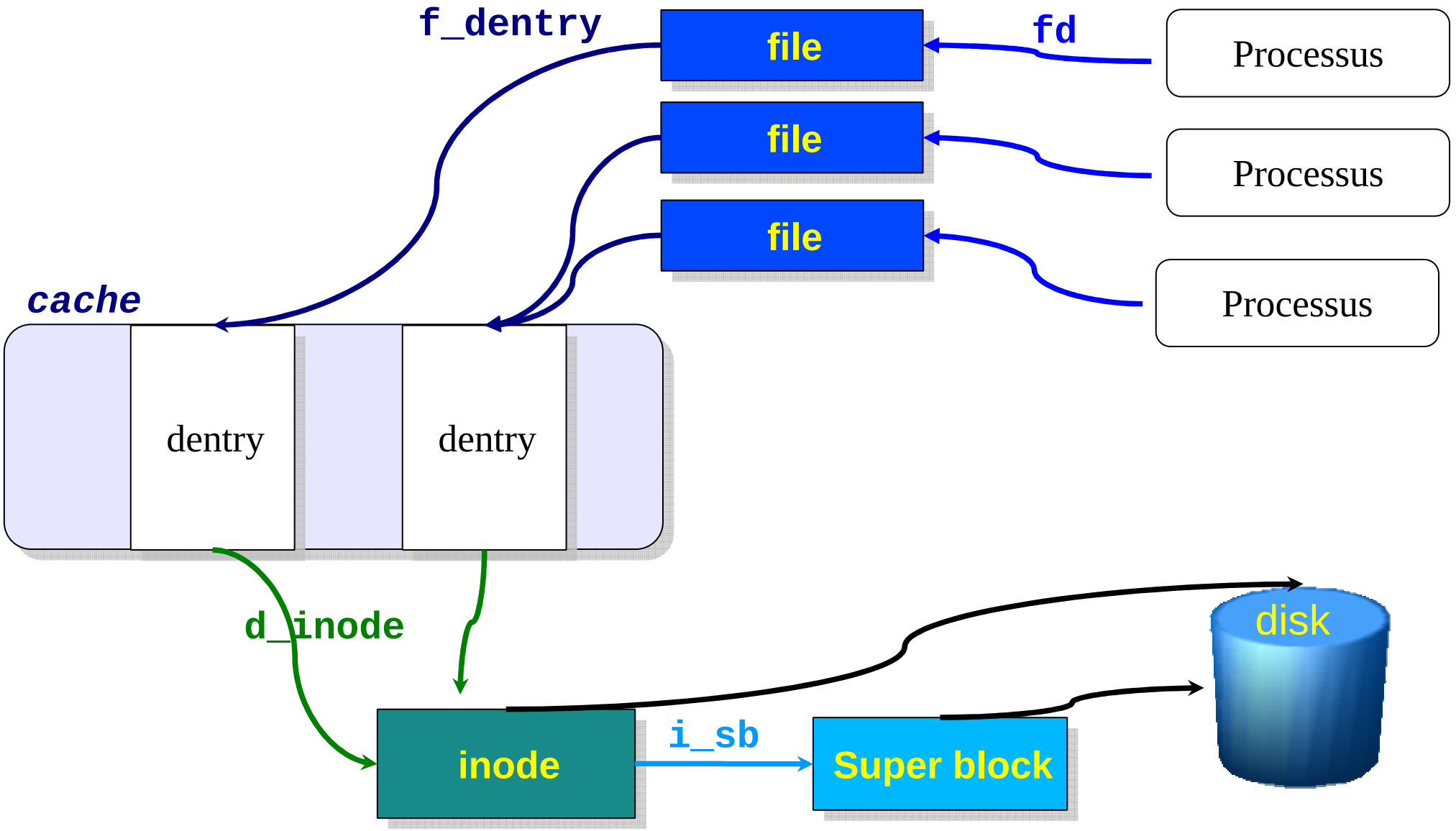
Les Principaux Objets de VFS

- File : Représente tout ce sur quoi on peut lire/écrire
  - Informations sur un fichier ouvert par un processus
  - N'existe que dans la mémoire du kernel
  - Fournit l'implémentation des opérations disponibles sur le fichier
  - Définie dans `<linux/fs.h>`
- Inode : Représente un objet dans le système de fichiers
  - Peut être un fichier, un répertoire, un lien symbolique…
  - Chaque Inode a un numéro unique (*inode number*) qui permet d'identifier l'objet correspondant
  - Certains objets ont un inode mais pas de structure file (liens symboliques) et vice-versa
  - Définie dans `<linux/fs.h>`

Les Principaux Objets de VFS

- Dentry : informations sur le lien entre une entrée de répertoire et le fichier correspondant
  - On ne veut pas manipuler des inodes mais des noms
  - L'association nom-inode peut être coûteuse
  - VFS maintient une liste de nom actifs ou récemment utilisés dans un arbre, le *dcache*
  - Les nœuds de cet arbre sont des *dentries*
  - Les *dentries* sont des intermédiaires entre les *files* et les *inodes*
  - Définie dans `<linux/dcache.h>`
- Super Block : Informations globales sur le FS
  - État courant de la partition
  - 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;
```

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;
    void *                  private_data;
```

La Structure 'file' (Fichier Ouvert)

- Représente un fichier ouvert par un processus, souvent abrégé en *filp* (pointeur sur struc file)
- `loff_t f_ppos;`
  - Position de la prochaine opération de lecture ou d'écriture (si pas O_APPEND)
  - Pourquoi cette information ici et pas dans l'inode?
- `struct list_head f_list;`
  - Un fichier fait partie d'une liste chaînée
  - Soit une liste de fichiers non utilisés
    - Cache
    - Sécurité grâce au NB_RESERVED_FILES pour root
  - Soit une liste de fichiers utilisés
- `atomic_t f_count;`
  - Compteur d'utilisations du fichiers
  - Nombres de processus + kernel
- `struct file_operations *f_op;`
  - Pointeur vers une structure contenant l'implémentation des opérations disponibles sur ce fichier
- `void * private_data;`
  - Informations spécifiques au drivers, conservées entre les appels

La structure file_operations

- Une pour chaque appel système sur un fichier ouvert

- 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,onlock(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)

Et un champs

- struct module *owner;

Opérations

- `llseek(file,offset,origin)`
  - Déplace le pointeur de lecture/écriture du fichier
  - default_llseek de fs/read_write.c utilisé si aucun spécifié
- `read(file,buf,count,offset)`
  - Essaie de lire *count* octets et les met dans le *buffer*
  - La lecture commence a offset et la méthode **doit** modifier sa valeur ensuite
- `write(file,buf,count,offset)`
  - Écriture
- `open(inode,file)`
  - Ouvre un fichier en créeant un nouvel objet *file* et en le reliant au *inode*
- `flush(file)`
  - Appelé quand une référence à un fichier est fermée, i.e. quand f_count est décrémenté
- `release(inode,file)`
  - Appelé quand la dernière référence à un fichier est fermée
  - Permet de faire le ménage (*private_data*)
- `struct module * owner;`
  - Pointeur vers le module à qui appartient ce fichier
  - Permet de maintenir le compteur d'utilisation

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

Écriture d'un module

- On ne considère que les *characteres devices*
  - Famille de périphériques qui sont accédés comme un flot d'octets
  - La principale différence entre un *char device* et un fichier est qu'on peut avancer/reculer dans un fichier et en général pas dans un c.d
  - Il existe aussi les *block devices* et les *network interfaces*
- Étant donné un *device*, on peut écrire un module pour le rendre accessible à travers VFS
  - Écrire les méthodes nécessaires (*open…*)
  - Relier ces méthodes au *file_ops*
  - Indiquer au kernel que le module gère les accès au device

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 »
      - 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,
};
```

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
- La création d'un fichier périphérique se fait avec la commande *mknod*

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)
  - Aiguillage "interne" vers le périphérique
    - A la charge du (programmeur du) pilote
- Le module manipule un seul type représentant le major et le minor
  - Accessible par le champs `i_rdev` de l'inode
  - Du type `dev_t` ou `kdev_t` suivant la version du noyau (32 bits en 2.6, 12 pour le major, 20 pour le minor)
  - Manipulation à l'aide de macros :
    - `MAJOR(dev)`,
    - `MINOR(dev)`,
    - `MKDEV(major,minor)`

Enregistrement d'un Pilote Caractère

- Par exemple dans init_module() :
  - Appel de *int register_chrdev(unsigned int major, const char *name, struct file_operations *fops);*
  - La suppression se fait avec *int unregister_chrdev(unsigned int major, const char *name);*
- 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 expérimentations
    - 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
```

Opérations étendues sur des devices caractère

- Le VFS masque le matériel
- On peut avoir besoin d'envoyer des commandes au matériel
- Première solution : utiliser le flot de données
  - Certains caractères sont interprétés par le driver comme devant faire une action
  - Utilisé par exemple dans les terminaux (tty)
  - Mais difficile à programmer
  - Interdit l'utilisation de certains caractères comme données utiles
- Deuxième solution : envoyer directement des commandes au driver
  - Utilisation de l'appel système *ioctl*

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, ...)`
  - En pratique pas un nombre variable d'arguments mais un `char* argp` mais pas de type checking à la compilation
- Méthode de la structure file_operations
  - `int (*ioctl)(struct inode inode,struct file * file,unsigned int cmd,unsigned long 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 référence

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
    - Sinon risque d'envoyer n'importe quoi au mauvais device…
- Solution 2 : Utiliser le schéma de numérotation Linux
  - Historiquement 16 bits divisé en 2
    - 8 bits comme magic number du device
    - 8 bits pour un numéro de commande unique pour le device
  - Maintenant, utilisation de 4 nombres

Schéma de Numérotation des Commandes ioctl Linux

- Type
  - Le *magic number* sur 8 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 tel que vu **par** l'application
  - _IOC_NONE
  - _IOC_READ : l'application recevra des données du device
  - _IOC_WRITE
  - _IOC_READ | _IOC_WRITE
- Size
  - Indique la taille des données (8-14 bits suivant les architectures)
  - Valeur disponible avec la macro _IOC_SIZEBITS
  - Optionnel et non contraignant

Schéma de Numérotation des Commandes ioctl Linux

- Des macros sont disponibles dans *<asm/ioctl.h>* pour construire les numéros de commande
- _IO(type,number)
  - Commande sans échange de données
- _IOR(type, number, data)
  - Commande pour une lecture des données
  - *data* est la structure pour stocker les données, un sizeof sera fait pour calculer sa taille
- _IOW et _IORW…
- Et pour les opérations inverses
  - _IOC_DIR, IOC_TYPE, _IOC_NR et _IOC_SIZE

Utilisation d'un paramètre

- Si l'utilisateur a spécifié un 3ème argument, alors `unsigned long arg` a un sens
  - Sinon sa valeur n'est pas utilisable
  - Vérification du sens de la commande
- 2 cas possible
  - C'est un entier, on peut l'utiliser directement
  - C'est un pointeur, c'est plus compliqué
- Si pointeur, c'est une adresse en user space
  - Le kernel doit vérifier la validité de l'adresse
  - Si l'adresse n'est pas valide, le processeur va générer une exception, et le kernel OOPSera (suivant la version)
  - Verification explicite avec `int access_ok(int type, const void *addr, unsigned long size);`

Utilisation d'un paramètre

- Une fois la vérification faite, on peut transférer les données
- Pour des structures arbitraires
  - `unsigned long copy_to_user(void *to, const void *from, unsigned long count);`
  - `unsigned long copy_from_user(void *to, const void *from, unsigned long count);`
- Optimisation pour les données de taille bien choisies (1,2,4,8 octets suivant les plateformes), macros dans *<asm/uaccess.h>*
  - `put_user(datanum, ptr)`
  - `get_user(local,ptr)`
  - La vérification de la validité de l'adresse est effectuée par ces macros
  - On peut s'en passer avec les versions `__put_user` et `__get_user`

Erreurs et Code de Retour

- Utilisation du paramètre pour retourner un succés
- 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**