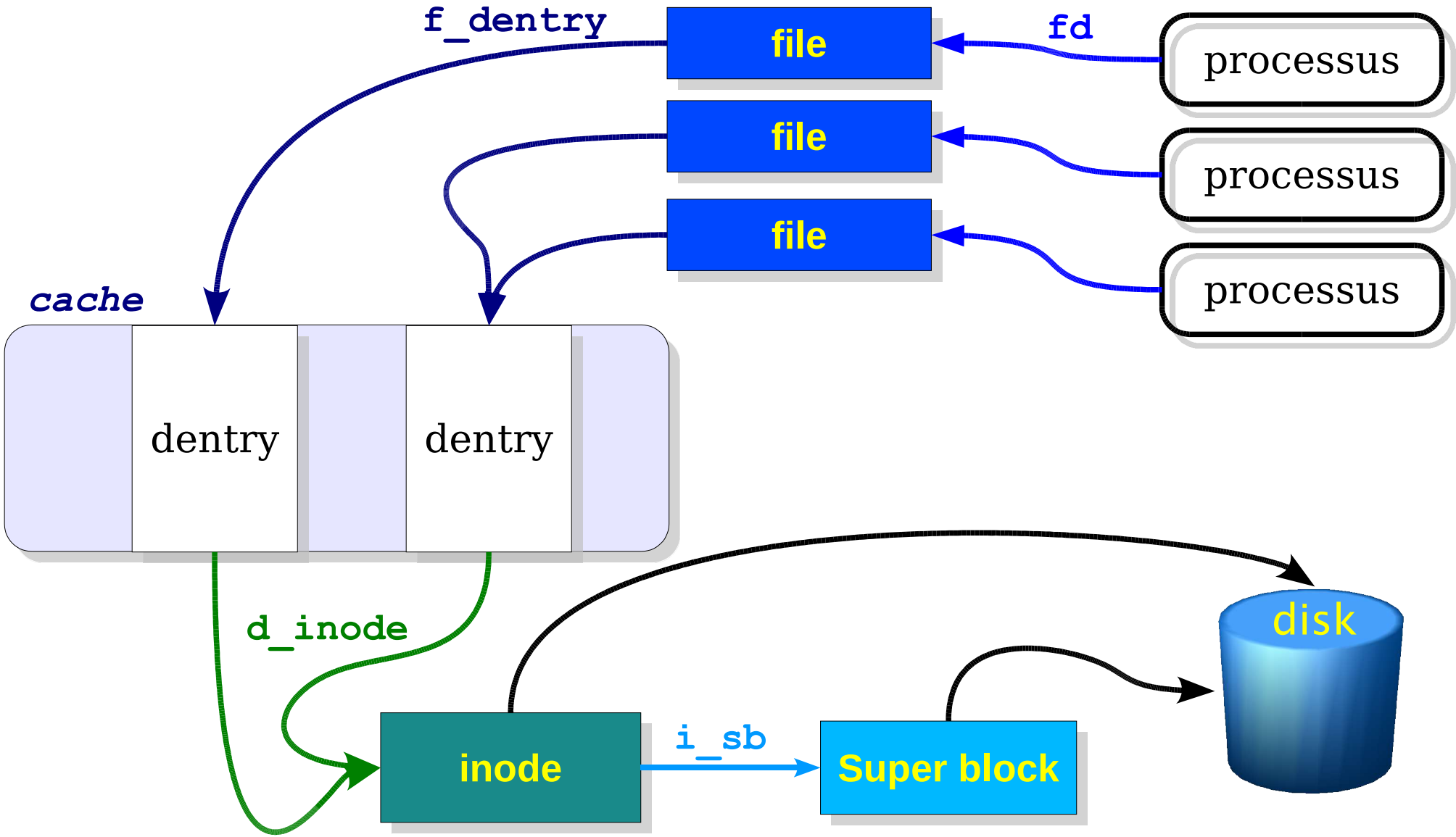
Le Virtual Filesystem Switch (VFS)

Partie 2 : Opérations sur les Chemins et Répertoires

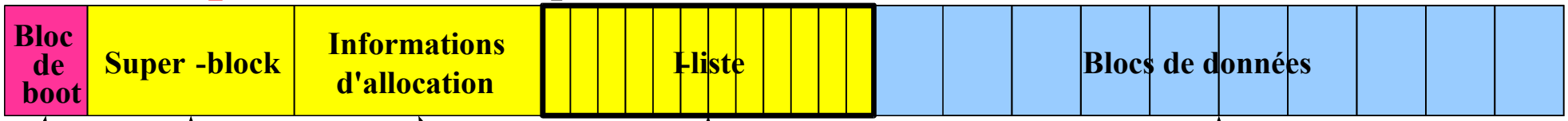
Rappel : les Principaux Protagonistes du VFS



Rappel : Identification d'un Fichier

- Un fichier est identifié de façon **unique** par
  - Un **identifiant de partition**
    - couple major/minor
  - Un identifiant de fichier **dans** la partition
    - numéro d'index (i-noeud/**inode**)

Contenu **possible** d'une partition Unix



Optionnel

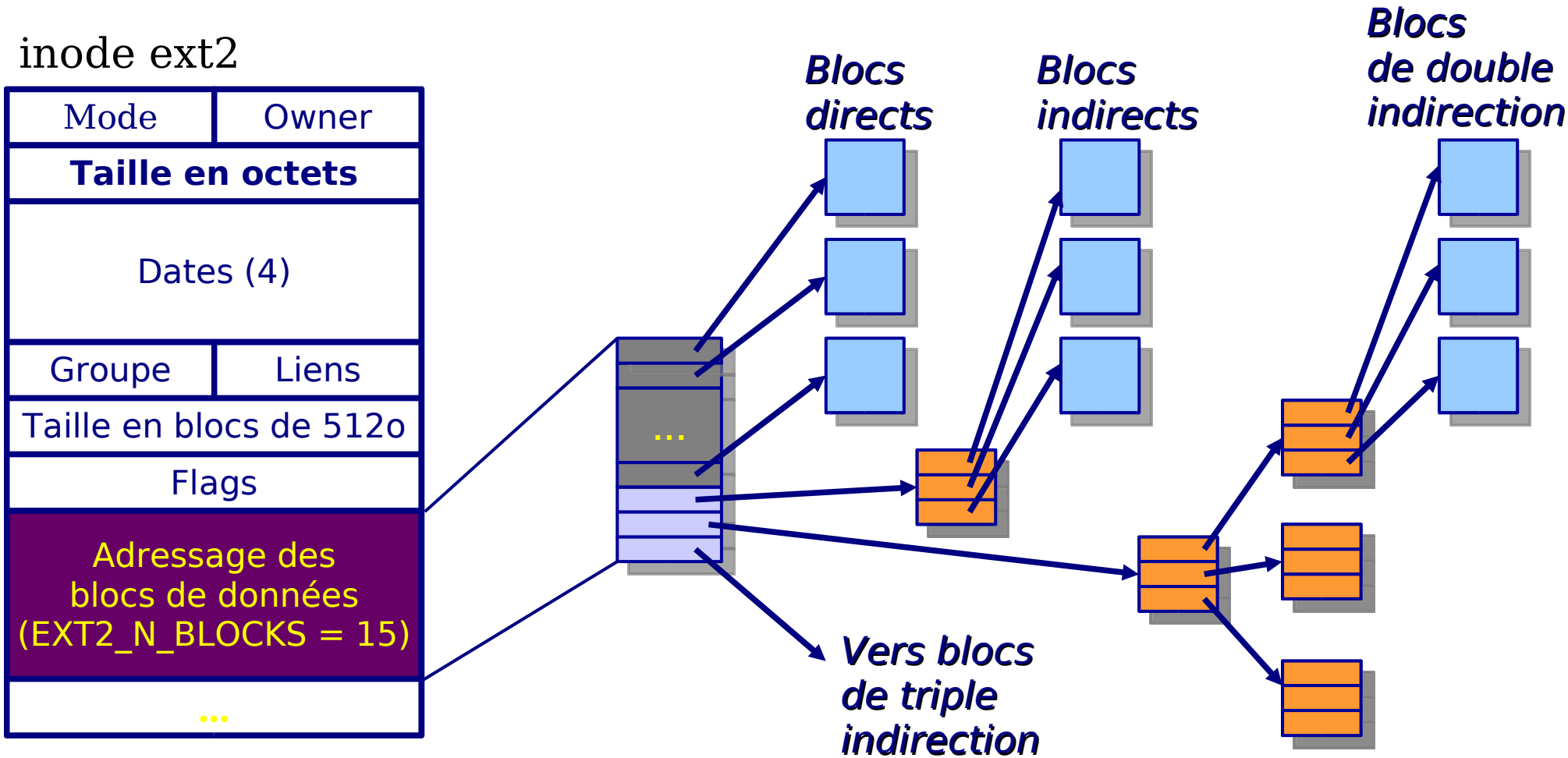
Information sur le système de fichiers (structure, dates màj, ...)

Tables d'allocation (i-noeuds, blocs, etc)

Liste des i-noeuds (i-liste)

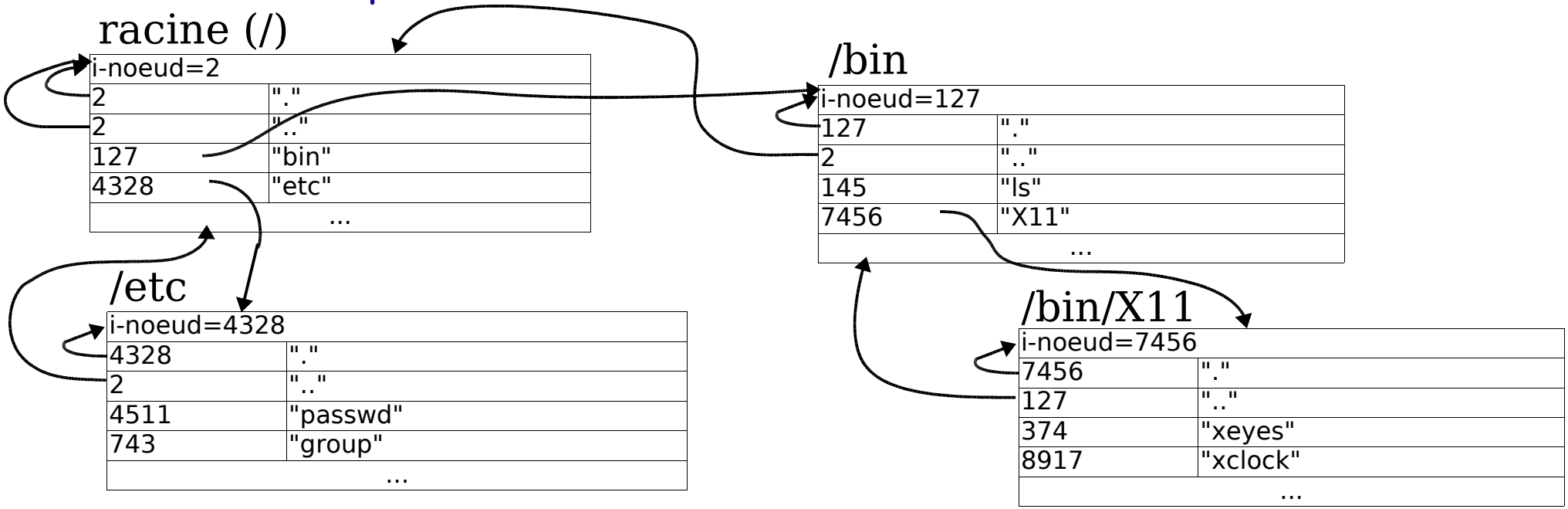
Blocs de données (contenu des fichiers et répertoires)

Structure **possible** d'un *inode* (ici ext2fs)



Chemins

- Méthode plus conviviale pour désigner un fichier
  - Plus convivial que <major,minor,inode>
- Mais il **faut retrouver** son i-node pour travailler avec
  - Q: Où se trouve l'association i-node <-> chemin ?
  - R: Dans les fichiers spéciaux **répertoire**
    - Répertoire = table associative



Flexibilité de l'Architecture VFS

- On vient de voir
  - Le contenu **possible** d'une partition
  - La structure **possible** d'un inode
- On peut trouver des choses **très** différentes !
  - Mais le VFS masque les différences
    - Les chemins s'utilisent toujours de la même façon
    - Les fichiers sont **tous** identifiés par un inode
  - Le VFS offre des **fonctionnalités génériques**
    - Déjà vues : struct file_operations
      - ouvrir, lire, ecrire un fichier ...
    - A voir : manipuler des inodes, des chemins, des blocs disques, des partitions ...

Exemple de Fonctionnalité Générique

- Conversion d'un chemin en inode
  - Par exemple lors de open("/toto/titi", flags)
    - Pour trouver l'inode de titi il faut d'abord trouver toto
    - Pour trouver toto il faut d'abord trouver "/" ...
    - Algorithme itératif
      - Le VFS connaît l'i-node de la racine
        - Information liée au point de montage
      - Lecture du contenu de "/"
        - Pour retrouver le **numéro** inode de "toto"
      - Lecture du **contenu de l'inode** de "toto"
      - Lecture du **contenu du fichier** "toto"
        - Pour retrouver le **numéro** de l'inode de "titi"
      - Lecture du **contenu de l'inode** de titi

Les Objets *dentry*

- dentry = directory entry (entrée de répertoire)
  - Cache des noms
    - Table de hachage (clef = nom de fichier)
    - Mémorise le résultat d'une recherche d'i-node
    - Optimise les résolutions multiples rapprochées
      - trouver un fichier, l'éditer, le compiler, le rééditer, ...
  - Peut-être considéré comme une forme très spécialisée de chaîne de caractères
    - S'utilise partout où on a besoin de manipuler un nom de fichier (qu'il faudra forcément convertir en inode)
      - Lister le contenu d'un répertoire
      - Suivre un lien (dur ou symbolique)
      - Supprimer un fichier
      - ...

Conversion d'un Chemin en I-node : Mise en Oeuvre

- Mise en oeuvre déléguée au pilote du SF
- Méthode fournie par le pilote : lookup(inode,dentry)
  - Rôle : trouver un fichier dans un répertoire
    - Exemple : rechercher "titi" dans <inode_de_toto>
  - **opération associée à l'inode**
  - inode : répertoire parent
    - Paramètre indispensable car la même méthode est utilisée par de nombreux répertoires
      - En C++ on ferait simplement inode.lookup(dentry) car on dispose du pointeur **this**. Mais en C, pas de pointeur this ...
  - dentry : structure à compléter en retour
    - Contient déjà le nom du fichier
    - Il faut ajouter l'inode (ou NULL si non trouvé)

Comment Implémenter lookup() ?

- Il faut recourir à des primitives de bas niveau
  - **Lire un bloc disque appartenant à un fichier**
    - En deux temps :
      - Fonction primitive : int bmap(inode,blk)
        - retourne le numéro du bloc de la partition correspondant au bloc blk du fichier désigné par inode
          - En fait, invoque une méthode attachée à l'inode ...
      - Fonction primitive bh=bread(dev,block,size)
        - retrourne handler vers bloc dans le cache des blocs
  - **Trouver le contenu d'un inode sachant son numéro**
    - Opération read_inode() du **super_block** ...
      - Car l'emplacement, la taille, le contenu de l'inode dépendent de l'implémentation du SF
        - Seul le pilote ext2 sait lire les inode d'une partition ext2

Quel est le Point de Départ de Tout ce Bazar ?

- Autrement dit, comment installer un nouveau SF ?
  - register_filesystem(struct file_system_type *fs)
    - même principe que register_chrdev(), mais installe toute la mécanique pour un SF au lieu d'un seul fichier device
    - Fournit la methode read_super()
      - Invoquée lors du **montage**
      - Doit compléter ce qui manque dans la struct super_block

```
struct file_system_type {
        const char *name;
        int fs_flags;
        struct super_block *(*read_super) (struct super_block
*, void *, int);
        struct module *owner;
        struct file_system_type * next;
        struct list_head fs_supers;
};
```

La Structure du Super Block (struct super_block)

struct list_head	s_list
kdev_t	s_dev
unsigned long	s_blocksize
unisgned char	s_blocksize_bits
unsigned char	s_dirt
unsigned long long	s_maxbytes
struct file_system_type *	s_type
struct super_operations *	s_op
struct dquot_operations *	dq_op
unsigned long	s_flags
unisgned long	s_magic
struct dentry *	s_root
struct rw_semaphore	s_umount
struct semaphore	s_lock
int	s_count
atomic_t	s_active
struct list_head	s_dirty
struct list_head	s_locked_inodes
struct list_head	s_files
struct block_device *	s_bdev
struct list_head	s_instances
struct quot_mount_options	s_dquot
union	u

Liste des super blocks

Vrai si modification

Taille Maximum des fichiers

Identifiant arbitraire a priori unique

Point de montage

inodes modifiés

inodes en cours d'E/S

Fichiers ouverts

Superblocks des SF montés du même type

Opérations du Super Block (struct super_operations)

- read_inode(inode) : lecture disque et remplissage de l'objet inode identifié par inode->i_ino
- read_inode2(inode,p) : version pour ino 64 bits (Reiser)
- dirty_inode(inode) (maj journal)
- write_inode(inode,flag) : flag = synchrone/asynchrone
- put_inode(inode)
- delete_inode(inode)
- put_super(super) : lors du démontage
- write_super(super)
- unlock_fs(super)
- statfs(super,buf) : statistiques (par exemple pour df)
- remount_fs(super,flags,data)
- clear_inode(inode) : idem put_inode + libération mémoire
- umount_begin(super) : interruption montage (NFS)
- fh_to_dentry()/dentry_to_fh() : utilisé par NFS
- show_options(seq_file,vfsmount) : pour l'affichage d'options spécifiques

i-node

- Deux versions existent !!
  - **En mémoire** :structure interne du VFS, générique
    - <linux/fs.h>
    - Beaucoup de choses (trop pour un transparent !)
  - **Sur disque** : structure dépendante du SF sous-jascent
    - Exemple : struct ext2_inode_info
    - Champ 'u' de la structure VFS d'inode
      - u = union
  - Certaines informations de la structure disque sont dupliquées dans la structure VFS

i-node en Mémoire

- Etats possibles de l'i-node = combinaison de :
  - I_DIRTY : i-node ayant besoin d'être écrit sur disque
    - Ecriture différée
    - En fait plusieurs variantes : I_DIRTY_SYNC, I_DIRTY_DATASYNC, I_DIRTY_PAGE
  - I_LOCK : en cours d'E/S
  - I_FREEING : en cours de libération
  - I_CLEAR : inutile (et invalide)
  - I_NEW : en cours de création
- Chaque i-node appartient à l'une de ces 3 listes
  - i-node valides inutilisés : cache disque
  - i-nodes valides et utilisés
  - i-nodes sales : a écrire

Les méthodes de l'objet i-node (inode_operations)

- create(dir,dentry,mode)
  - création d'un inode disque à partir de son nom (dentry)
- lookup(dir,entry)
- link(old_dentry,dir,new_dentry)
- unlink(dir,dentry)
- symlink(dir,dentry,symname)
- mkdir(dir,dentry,mode)
- rmdir(dir,dentry)
- mknod(dir,dentry,mode,rdev)
- rename(old_dir,old_dentry, new_dir,new_dentry)
- readlink(dentry,buffer,buflen)
- follow_link(inode,dir)
- truncate(inode)
- permission(inode,mask)
- revalidate(dentry)
  - Mise à jour attributs dupliqués (NFS)
- setattr(dentry,iattr)
  - Mise à jour des attribut de l'i-node (prop, perm, ...)
- getattr(dentry,iattr)
  - Récupération attributs

D'où Viennent les Méthodes de l'inode ?

- Ces méthodes sont spécifiques à chaque SF
  - Exemple : unlink(dir,dentry)
    - Décrémente le nombre de lien
    - Libère les blocs si nombre de liens=0
    - La façon de réprésenter un lien, de gérer les blocs libres ou utilisés est spécifique au SF
- Par commodité, le VFS fournit une version générique
  - Mais si les versions génériques ne conviennent pas, il faut les **surcharger**
- Où se fait la surcharge ?
  - Au moment de la **création** de l'inode
    - méthode read_inode() du super_block

Et les Méthodes des Fichiers ?

- Exactement comme les méthodes de l'inode
  - Car elles sont aussi spécifiques du FS
  - Elles ont aussi une implémentation générique par défaut
  - On peut avoir besoin de les surcharger
    - Par exemple write() de NFS doit envoyer les données écrites au serveur NFS …
  - Chaque inode contient un champ **i_fop**
    - Pointeur vers la struct file_operations
    - Affecté lors de la création de l'inode par read_inode()
- Remarque :
  - C'est exactement la même structure que pour un périphérique caractère
    - On peut faire un SF ne contenant que des FIFOs …

Modification des Attributs d'un i-*node*

- Certains AS modifient les attributs des i-nodes
  - utimes(2), chown(2), chgrp(2), ...
- Les point d'entrée de ces AS ont tous un fonctionnement similaire :
  - Remplir une `struct iattr` (<linux/fs.h>)

```
struct iattr {
    unsigned int ia_valid;
    umode_t      ia_mode;
    uid_t        ia_uid;
    gid_t        ia_gid;
    loff_t       ia_size;
    time_t       ia_atime;
    time_t       ia_mtime;
    time_t       ia_ctime;
    unsigned int ia_attr_flags;
};
```

  - invoquer `notify_change(dentry,iattr)`
    - invoque méthode setattr de l'inode

Créer un Nouveau SF : On Récapitule...

- Créer un structure file_system_type
  - Ecrire une méthode read_super()
    - Remplir la structure super_block
    - Fournir les méthodes du super_block (champ s_op)
      - En particulier read_inode()
- Fournir les méthodes à surcharger au niveau de l'inode
  - Méthodes sur inodes : struct inode_operations
    - En général on fournit au moins lookup()
  - Méthodes sur fichiers ouverts : struct file_operations
  - Attachées à chaque inode par read_super()
- Enregistrer la structure file_system_type
  - register_file_system()

Systèmes de Fichiers Exotiques

- Normalement, un SF s'appuie sur un périphérique de stockage
- Un SF exotique invente/fabrique des fichiers dynamiquement
  - Exemple : procfs
    - Fabrique un répertoire exotique pour chaque processus
    - Fournit des répertoire et fichiers exotiques
      - Informations sur le matériel
      - Configuration du matériel
- Principe de mise en oeuvre
  - read_inode() fournit des méthodes **différentes** selon l'inode demandé

Retour sur la Structure *dentry*

struct module(linux/module.h)

atomic_t	d_count
unisgned int	d_flags
struct inode *	d_inode
struct dentry *	d_parent
struct list_head	d_hash
struct list_head	d_lru
struct list_head	d_child
struct list_head	d_subdirs
struct list_head	d_alias
int	d_mounted
struct qstr	d_name
unisgned long	d_time
str. dentry_operations *	d_op
struct super_block *	d_sb
unisgned long	d_vfs_flags
void *	d_fsdata
unsigned char *	d_iname

Chaînage avec autres entrées ayant même clef dans hash table

Inutilisé mais encore valide : élimination avec politique LRU

Liste des fils du répertoire parent

Sous répertoires du répertoire

Liste de toutes les autres dentry pour le même inode

Etats Possibles d'une *dentry*

- Free
  - Aucune information valide (dans slab allocator)
- Unused
  - d_count = 0, mais d_inode valide
  - Susceptible d'être recyclé (politique LRU)
- In Use
  - d_count > 0, d_inode valide
- Negative
  - d_inode = NULL
    - Résolution de fichier inexistant
    - Fichier supprimé
  - Persistant dans le cache car info utile (nom fichier)