Linux
VFS

révision 0024

Introduction to Linux Virtual File System

Janvier 2010

*Par Jérémy Cochoy [Zenol] (**Jeremy.Cochoy@gmail.com**)*

Sommaire

1. Avant-Propos

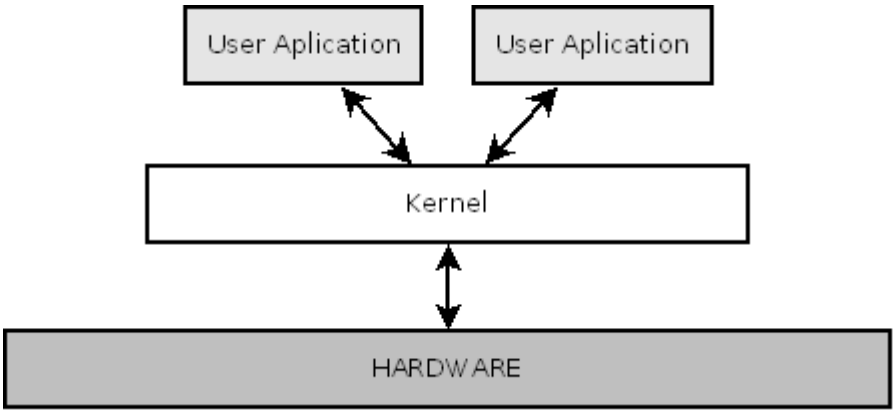
1.1. Terminologie

Tout au long de cet article, vous verrez apparaître des termes anglais, que je me refuse à traduire. Ils sont en effet tellement usités dans le milieu du développement kernel qu'il serait mal venu de les ignorer, ou de se priver de leur usage pour une traduction parfois floue et approximative. Ils seront donc traduits lors de leur première apparition (vous trouverez le terme francophone, dans les rares cas où il existe) puis utilisés comme de simples noms entrés dans la langue française, ou plus exactement, dans la longue liste de termes qu'emploie un développeur.

1.2. Près requis

Avant de lire ce document, il est fortement conseillé d'avoir de solides bases en C. En effet, les lignes de codes citées ne seront présentes qu'à titre d'exemple afin d'illustrer les propos qui s'y rapportent et aucun commentaire ne sera fait sur le comportement évident de ces dernières. Je rajouterais qu'une connaissance du monde UNIX et plus généralement de tout ce qui se rapporte au développement kernel peut faciliter la lecture de ce document.
Les personnes ayant déjà expérimenté le développement en kernel-land souhaiteront certainement passer au chapitre suivant.

1.3 Kernel

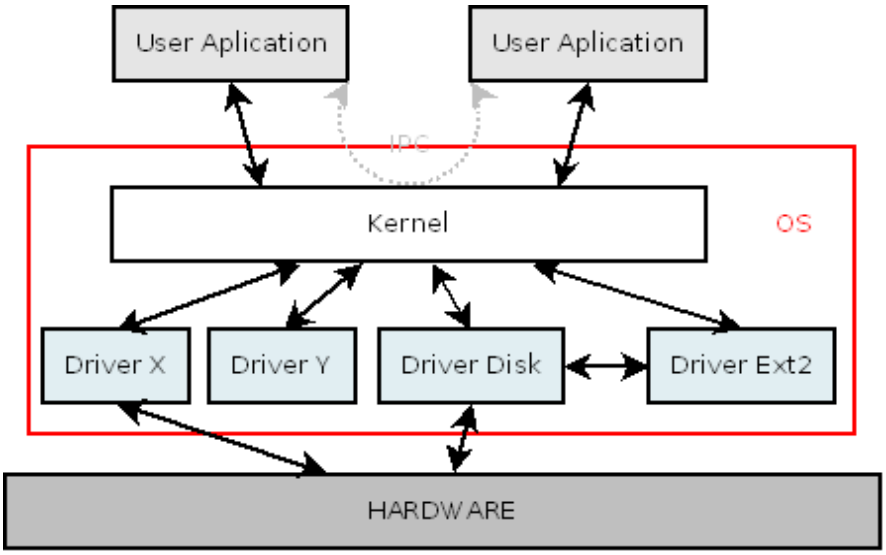


F1.3 1 : La place du kernel(Noyau) dans un ordinateur

Un système d'exploitation de type UNIX, dont fait partie Linux, est composé d'un ensemble d'applications ainsi que d'un élément central, que l'on nomme noyau. Les différents éléments constitutifs d'un système GNU/LINUX sont agencés et pré-configurés afin de former une distribution.
Ici, nous nous intéresserons particulièrement à une partie de ce noyau, le VFS.

Le noyau est l'ensemble du code d'un système d'exploitation qui a la tâche de gérer les différents processus devant s'exécuter simultanément, les droits utilisateur et administrateur, ainsi que la coordination des périphériques et les échanges de données entre ces derniers et les applications. Il fournit la couche d'abstraction nécessaire aux applications afin de s'exécuter indépendamment du matériel utilisé.

1.4 Drivers et modules



FI.4 1 : La place des drivers(Pilotes) et modules, ainsi que du kernel(Noyau) dans un OS(Système d'Exploitation)

Une partie du code qui constitue le noyau peut être subdivisée en petits éléments, que l'on nomme modules (ou drivers dans le cas où leur tâche est de contrôler un périphérique matériel).
Ces derniers sont caractérisés par leur fonction spécifique qui leur est propre et qui n'est pas directement liée aux tâches centrales du kernel. Ainsi, on retrouve dans cette catégorie tout ce qui se rapporte au contrôle des composants d'un ordinateur, c'est-à-dire les pilotes (drivers) de périphériques.
On citera pour exemple le contrôle du disque dur, mais aussi la réception des interruptions générées lors d'une frappe au clavier.
L'existence de drivers, qui peuvent être séparés du noyau et compilés séparément sous forme modulaire, est justifiée par leur fonction de "couche d'abstraction" entre le kernel et le code de contrôle des périphériques.

Certaines communautés préfèrent réserver l'appellation « driver » aux pilotes de périphérique, et nommer module ce que l'on pourrait qualifier de « driver software ». Mais un module peut parfaitement être, soit compilé avec le noyau, soit compilé sous forme modulaire (figurant alors dans /lib/modules/<kernel version>)

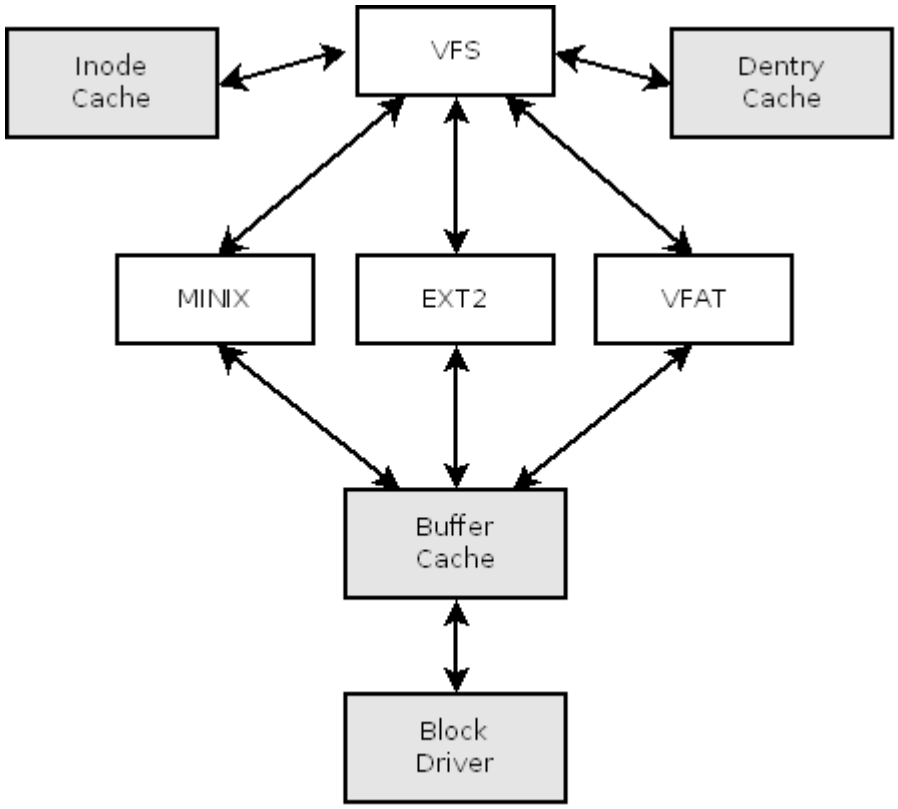
On trouve aussi parmi les modules, les implémentations d'algorithmes de cryptographie pour des raisons de performance, et enfin, sujet qui nous intéresse tout particulièrement, les drivers/modules de filesystem. Ces derniers sont l'implémentation d'un algorithme permettant de stocker une hiérarchie de fichiers sur un support, et résultent chacun du choix d'une représentation physique différente.

2. Introduction

2.1 VFS en quelques mots

VFS, pour Virtual File System est une solution choisie pour représenter une hiérarchie de fichiers dans le kernel Linux.
Dans une philosophie Linux où tout est fichier, aussi bien configuration/état du système (/etc et /proc) que périphériques (/dev), il est nécessaire de fournir une couche d'abstraction afin que les différents modules communiquent au kernel leur représentation sous forme de fichiers. Le kernel, à son tour, communiquera aux diverses applications via une certaine API, qui correspond aux fonctions système comme `open, close, read, write, mmap, …`

2.2 Structure global de VFS



F2.2 1 : Liens entre VFS et les filesystems réels

Ce schéma est un rapide résumé des interactions entre VFS et les modules de filesystem (MINIX, EXT2, VFAT, etc.)

Chaque filesystem obéit à différentes spécifications quant à la représentation des données sur le disque. L'emploi de VFS permet au kernel de ne pas avoir à tenir compte de leurs topologies.

VFS dispose d'une structure très proche du filesystem ext2, ce qui permet de minimaliser le code dédié à l'abstraction du filesystem, et de n'avoir que de simples copies de valeurs dans les structures appropriées. Dans la suite de ce document, je parlerai presque exclusivement des structures de VFS et non de celles spécifiques aux filesystem.

Lors de l'ajout d'un filesystem, ce qui se produit soit au démarrage de la machine, soit lorsque le kernel en éprouve la nécessité (si le driver est compilé sous forme modulaire, et le kernel configuré pour le charger dynamiquement) celui-ci s'enregistre auprès du kernel, en lui fournissant une structure, le Superblock, contenant des pointeurs de fonctions qui seront utilisés pour le parcours de l'arborescence, l'allocation et l'initialisation d'inodes, ainsi que de nombreuses autres opérations où il peut être nécessaire que le driver intervienne.

Retenez qu'une inode est une structure qui contient toutes sortes d'informations et métadonnées relatives à un fichier (l'exemple le plus éloquent est la date de modification), et qui de plus détient un numéro i_ino unique qui correspond au fichier. On notera aussi que le nom d'un fichier n'est pas stocké dans l'inode.

Par la suite, lorsqu'un programme demandera l'accès à une donnée dans l'arborescence de fichiers, ce seront ces fonctions, spécifiques au filesystem auquel appartient la donnée, qui seront appelées. Ainsi, si l'on étudie le comportement de la commande cat, on a tout d'abord la recherche du fichier dans l'arborescence de dossiers, puis l'ouverture du fichier, sa lecture, et enfin sa fermeture. Chacune de ces étapes nécessite l'intervention du driver de ce filesystem.

La recherche d'un nœud dans une arborescence depuis le disque est une opération qui s'avère très coûteuse en terme de performance. Pour ext2 ou minix, il est nécessaire de lire, au minimum, autant de blocks de données sur le disque qu'il existe de nœuds jusqu'au fichier. Si l'on tient compte des temps de latence du périphérique, et du fait que les différents blocks ne sont probablement pas agencés bien dans l'ordre sur ce dernier, les temps d'accès mémoire sont bien plus intéressants. C'est cette dernière proposition qui est à l'origine d'une importante optimisation : les caches d'Inodes et de Dentries.

On trouvera le code relatif à VFS dans /usr/src/linux-2.6.x.y/fs/.

2.3 Inodes Cache

L'inode cache est un espace mémoire réservé au stockage des dernières inodes lues afin d'optimiser la lecture de ces dernières sur le périphérique de stockage.

Il s'agit d'une table de hachage unique et commune à tous les systèmes de fichiers où chaque entrée est un pointeur vers une liste d'inodes possédant toutes le même hash.

```
inode.c L77: static struct hlist_head  *inode_hashtable __read_mostly;
```

F2.3 1 : La table de hachage des inodes

Le hache correspondant à chaque inode dépend de son numéro (i_ino) et de l'adresse du VFS Superblock du filesystem auquel il appartient (on peut donc différencier plusieurs points de montage entre eux, car ils possèdent chacun leur propre VFS Superblock).

```
inode.c L573: static unsigned long hash(struct super_block *sb, unsigned long hashval);
```

F2.3 2 : La fonction de hachage pour une inode

Nb: On notera qu'une recherche dans cette liste provoque la libération « asynchrone » des inodes marquées comme 'I_FREEING | I_CLEAR | I_WILL_FREE'.

Quand le système nécessite d'accéder à une inode 'inode', il calcule son hash, puis la recherche dans la liste située dans inode_hashtable[hash(sb, inode->i_ino)].
L'inode portant le même numéro et appartenant au même système. C'est ce qu'illustre l'extrait *F2.3.3*.

```
inode.c L558:
static struct inode *find_inode_fast(struct super_block *sb,
                                     struct hlist_head *head, unsigned long ino)
{
        //...
        hlist_for_each_entry(inode, node, head, i_hash) { //On each inode from the inode list
                if (inode->i_ino != ino) //Not the same i_ino
                        continue;
                if (inode->i_sb != sb) //Not the same superblock
                        continue;
        //...
        return inode ? Inode : NULL; //Return inode found or NULL
}
```

F2.3 2 : La fonction de hachage pour une inode

Quand le (file)système a besoin de récupérer une inode (via `iget_locked` ou `iget5_locked`), VFS commence par rechercher dans la liste des inodes existantes via la table de hachage.

Si l'inode en question est introuvable, elle sera alors allouée, verrouillée puis initialisée, avant d'être renvoyée. Le système incrémentera alors i_count qui correspond au nombre d'utilisations simultanées de l'inode. (Une inode avec un i_count = 0 correspond à une inode qui devra être libérée par le système. La mémoire en question pourra alors être à nouveau allouée pour une autre inode).

Lorsque le système charge une inode depuis le périphérique de stockage pour la première fois, il initialise i_count à 1, ce qui signifie que cette inode n'est utilisée que dans un seul contexte, par exemple pour un éditeur de texte. Si à nouveau, l'inode est demandée dans un second contexte, par exemple suite à une commande cat, alors i_count est incrémenté. Par la suite, si l'éditeur est quitté, imaginons avant que la commande cat ne termine son exécution, alors l'inode verra son champ i_count repasser à la valeur 1, et l'inode sera conservée par le noyau et continuera à être utilisée par cat. Enfin, la commande cat terminée, i_count passera à 0 et sera libéré.

Afin de signaler que l'utilisation d'une inode est terminée, on utilisera iput ce qui diminuera i_count et si nécessaire appellera les fonctions nécessaires à la libération de l'inode.

On constate donc que lors de l'utilisation simultanée de la même inode dans plusieurs contextes, seul une unique instance subsiste en mémoire.

Les inodes qui ne sont plus utilisées (sans dentry lié, aucun file descriptor*(descripteur de fichier)* ouvert) sont placées dans la liste des inodes inutilisées et seront libérées par la

suite via dispose_list().

2.4 Dentry Cache

On appelle Dentries*(directory entries / entrées de répertoire)* les structures qui contiennent l'association entre un nom de fichier et le numéro d'inode correspondant. Elles sont générées par la fonction lookup, implémentée dans chaque filesystem, qui prend en paramètre l'inode du répertoire parcouru (le premier est la racine "/") et le nom du fichier/répertoire. Elle doit remplir une structure dentry qui lui est passée en paramètre, soit par l'inode correspondante, soit par l'inode invalide : 'NULL' (qui est associée à l'inexistante de l'inode recherchée).

Un exemple flagrant, pour vous rendre compte des répercussions du dentry cache sur les performances est de simplement lister ('ls') un répertoire qui n'a pas été accédé depuis le montage du disque. Renouveler l'opération une seconde fois, la différence de temps est alors flagrante.

A noter que le DentryCache est un « maître » de l'InodeCache, c'est à dire que si une dentry existe dans le cache, alors l'inode associée existe nécessairement. De même, si une dentry est détruite, alors l'inode associée est aussi détruite.

```
dcache.c L9:
/*
 * Notes on the allocation strategy:
 *
 * The dcache is a master of the icache - whenever a dcache entry
 * exists, the inode will always exist. "iput()" is done either when
 * the dcache entry is deleted or garbage collected.
 */
```

F2.4 1 : Stratégie de gestion mémoire

Tout comme l'Inode Cache, on retrouve la table de hashage :

```
dcache.c L63: static struct hlist_head *dentry_hashtable __read_mostly;
```

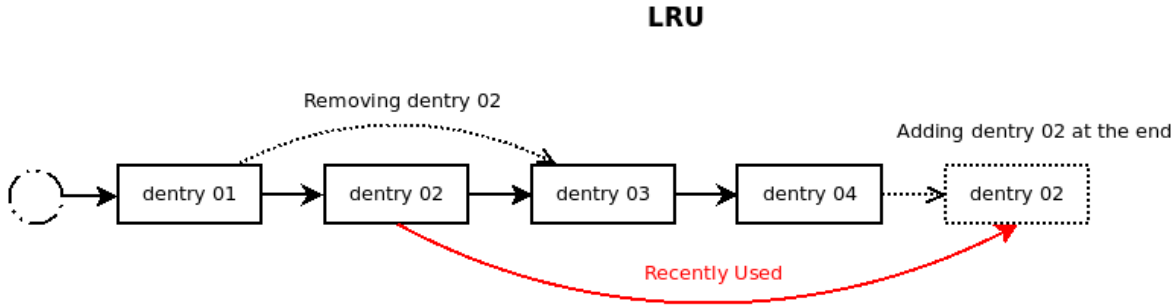
F2.4 2 : La table de hachage des dentries

La fonction de hashage des Dentries diffère de celle des inodes. On assiste à un pré-hashage à partir du nom complet de la dentry, puis la valeur résultat est re-hashée avec l'adresse de la dentry parente de celle-ci.

```
dcache.c L1111:
static inline struct hlist_head *d_hash(struct dentry *parent,
                                        unsigned long hash)
{
        hash += ((unsigned long) parent ^ GOLDEN_RATIO_PRIME) / L1_CACHE_BYTES;
        hash = hash ^ ((hash ^ GOLDEN_RATIO_PRIME) >> D_HASHBITS);
       return dentry_hashtable + (hash & D_HASHMASK);
}
```

F2.4 3 : La fonction de hashage des dentries

Afin d'améliorer les performances, le système maintient une list des Last Recently Used (*LRU : Derniers Récemment Utilisés*) dentries contenues dans le cache. Le système dispose de deux LRU Lists. Plus une dentry est utilisée, plus elle sera située vers la fin. Lors de la création d'une dentry, elle est ajoutée à la fin de la première LRU List. Dans le cas où le cache est plein, c'est le premier maillon de la liste qui sera réutilisé pour être placé en fin de liste. Si la dentry est à nouveau utilisée, alors elle sera migrée dans la liste de second niveau, où à chaque utilisation la dentry sera repositionnée en fin de liste. On peut ainsi aisément trouver les dentries les moins utilisées (positionnées en début de liste) afin de les libérer si nécessaire.



F2.4 4 : Déplacement d'une dentry à la fin d'une LRU List

2.5 Monter un filesystem

Monter un filesystem est un évènement qui peut sembler banal dans la vie de tous les jours d'un utilisateur de Linux. La commande est de ce type :

```
$ mount -t iso9660 -o ro /dev/sr0 /mnt/cdrom
```

F2.5 1 : Monter un cdrom

Ici, mount ne fait que passer la main au kernel avec les arguments suivants :

- Le type de filesystem (iso9660)
- Le périphérique (/dev/sr0)
- Le point de montage /mnt/cdrom
- Une chaine de caractères destinée à être interprétée par le filesystem (ici aucune)
- Divers flags (ici ro)

Ceci se fait via la fonction do_mount(). Cette dernière commence par récupérer les flags de montage, vérifie l'existence du point de montage puis passe la main à do_new_mount dans l'exemple ci dessus.

C'est alors que do_new_mount() tente d'interpréter la chaine fstype et de récupérer la structure décrivant le filesystem correspondant. Dans le cas d'une compilation sous forme de module, ce dernier pourra être chargé par le système quand il devient nécessaire[α].

Vient alors le tour de vfs_kern_mount() qui vas créer la structure vfsmount où seront stockées les informations relatives au montage. C'est alors que la fonction get_sb du filesystem est appelée. A ce

stade, rien ne garantit que la chaine passée en paramètre corresponde à un périphérique valide[β].

Si cette dernière opération est une réussite, alors le noyau tentera de placer le filesystem au point de montage associé. En cas d'échec, le superblock alloué(et la structure vfsmount) est libéré.

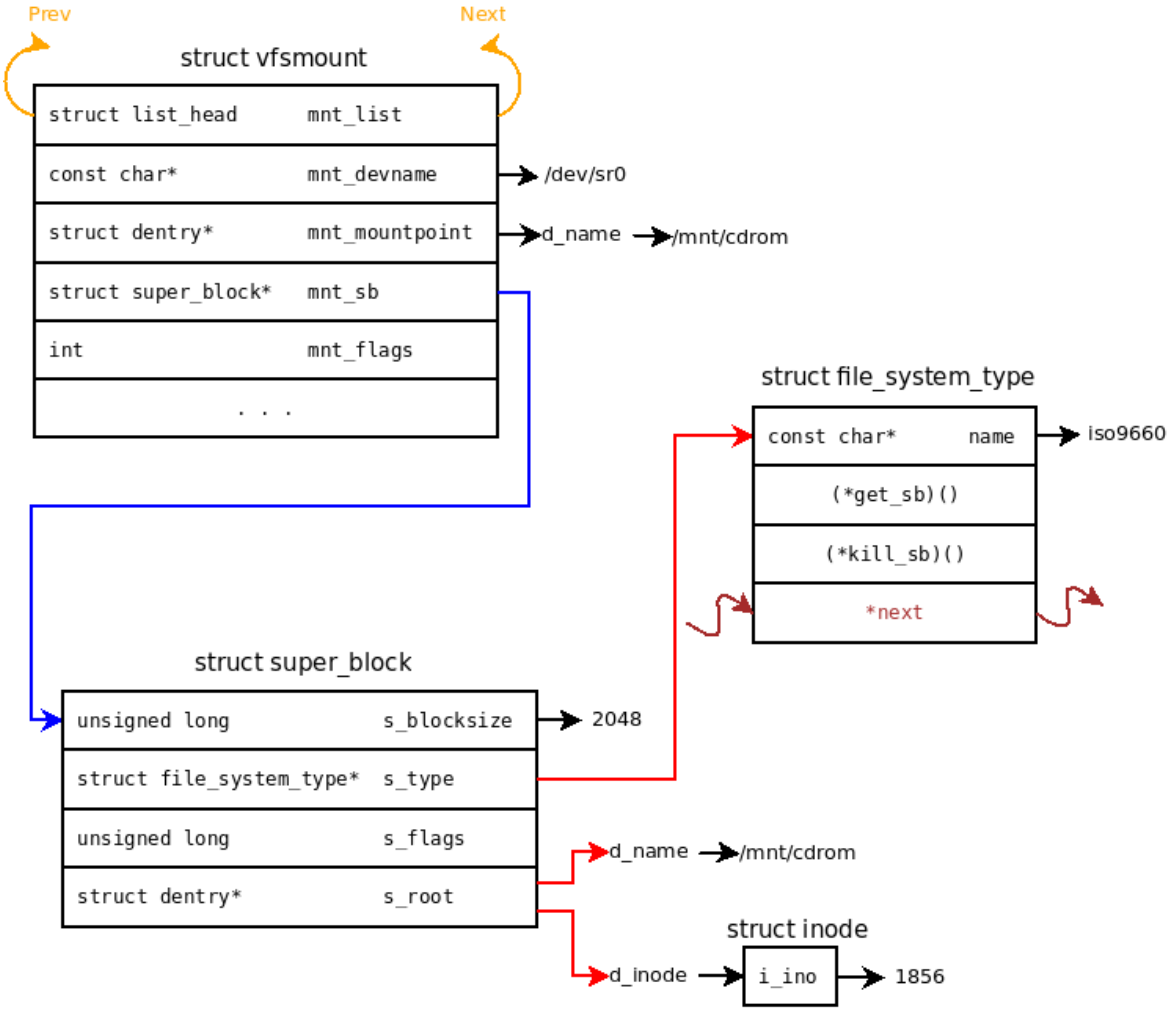
```
namespace.c L1905: long do_mount(char *dev_name, char *dir_name, char *type_page,
namespace.c L1906:                unsigned long flags, void *data_page)

super.c L912: struct vfsmount *
super.c L913: vfs_kern_mount(struct file_system_type *type, int flags, const char *name,
              void *data)

super.c L997: struct vfsmount *
super.c L998: do_kern_mount(const char *fstype, int flags, const char *name, void *data)
```

F2.5 2 : Fonctions utilisé pour le nouveau montage d'un filesystem

Dans le cas où la commande précédente serait un succès, nous aurions quelque chose de similaire à la figure *F2.5 3*.

α : La liste des systèmes de fichier actuellement chargée dans le kernel est disponible via la commande : « cat /proc/filesystems ».

β : Bien sur, le binaire mount peut très bien effectuer lui même toutes sortes de vérifications. Nous ne nous intéressons ici qu'à l'aspect kernel.



F2.5 3 : Résultat d'une commande mount pour un cdrom

2.6 Démonter un filesystem

Une fois les tâches de l'utilisateur accomplies, que ce soit lors de l'arrêt du système ou bien simplement car l'utilisateur souhaite par exemple déconnecter un périphérique de stockage amovible, ou encore un administrateur effectuer une maintenance, il est alors nécessaire de démonter le filesystem.

```
$ umount /mnt/cdrom
```

F2.5 1 : Démonter un cdrom

Le démontage (unmounting) d'un filesystem est bien sur l'action réciproque du montage. Un système de fichiers ne peut être démonté tant qu'il est utilisé par une application. Il est donc nécessaire que toutes les inodes détenues par le filesystem puissent être libérées. Suite à une commande umount, le noyau recherche dans le cache des inodes toutes les inodes appartenant au système de fichiers qui doit être démonté. Elles sont alors libérées

(si nécessaire, les modifications sont écrites sur le disque). Le noyau récupère alors le superblock, et là encore, si nécessaire, écrit les modifications sur le disque. La mémoire est alors libérée, la structure vfsmount elle aussi supprimée de la mnt_list et libérée.

3. Structures élémentaires

3.1. File System Type

```
fs.h L1738: struct file_system_type {                                   //...
```

Cette structure décrit un filesystem pour qu'il puisse être enregistré dans la liste des FS disponible.

Voici un exemple simple :

```
//FS type
static struct file_system_type yfs_fs_type =
  {
   .name       = "yfs",
   .get_sb     = yfs_get_sb,          //Get superblock (mount)
   .kill_sb    = yfs_kill_sb,         //Kill superblock (umount)
   .owner      = THIS_MODULE,         //The filesystem owner(this module)
   .fs_flags   = FS_REQUIRES_DEV,     //Tell we need a block device
 };
```

3.1. Inode

```
fs.h L719: struct inode {                                               //...
```

La structure inode contient toutes les informations relatives à l'état d'un fichier[α]. Voici le détail des champs les plus usuels.

i_ino

numéro identificateur propre à chaque inode. Pour certains systèmes de fichiers qui n'ont pas d'inode fixe (FAT32) ce numéro est généré dynamiquement par exemple par iunique. Pour des systèmes de fichiers comme ext2/3, ce numéro est propre à chaque fichier, les ids 0 et 1 ne sont pas utilisables, la racine (/) se voyant alors attribuer l'inode numéro 2.
(Les valeurs 0 et 1 son respectivement :
Une valeur invalide
Une node utilisée pour lier tous les blocks considérés comme défectueux du disque.)

i_count

Nombre de contextes d'utilisation de l'inode. Incrémenté à chaque ouverture d'un fichier, et décrémenté lors de sa fermeture.

i_mode

Ils'agit du chmod du fichier/répertoire ainsi que du type de fichier. Ce peut être un fichier, un répertoire, un lien dur, un lien symbolique, un char device, un block device, un point de montage... Généralement on se contente de gérer les cas qui relèvent du filesystem (Lien, fichier et répertoire).
La liste des valeurs est disponible dans include/linux/stat.h

α : A noter qu'il sera parfois nécessaire d'adjoindre des données supplémentaires à une inode, par exemple la place des premiers blocs du fichier correspondant. On créera alors une superstructure qui contient les données et l'inode, et on utilisera les macros fournies par le kernel comme `container_of`. Ceci sera abordé à nouveau plus tard dans ce document.

i_nlink

Nombre de liens physiques pointant sur la même inode. Dans le cas d'un système de fichiers comme ext3, c'est cette valeur qui permet de savoir lors de la suppression d'un fichier si l'inode et le contenu associé doivent être détruits ou bien conservés pour un autre fichier.
Un lien physique (ln src_file link_name) incrémente cette valeur, et la suppression d'un des liens (Il n'y a pas « d'original » ou de « premier fichier », toutes les entrées vers ce fichier/cette inode se valent) entrainera la décrémentation de cette valeur. Quand i_nlink atteint 0, le système sait que le fichier peut être considéré comme « détruit » et les données libérées (c'est le filesystem qui se chargera de cette tâche).

i_uid && i_gid

Ce sont respectivement l'identifiant de l'utilisateur possédant le fichier et le groupe auquel appartient le fichier. La correspondance peut parfois être faite en observant les fichiers /etc/passwd et /etc/group.

i_size

La taille (en octets) du fichier/répertoire considéré. Par exemple, les répertoires ont "souvent" une taille multiple de 4096 octets (correspondant à une page mémoire sous certaines architectures).
Il s'agit bien de la taille des données contenues dans le fichier, et non de l'espace utilisé sur le périphérique de stockage.

i_atime && i_mtime && i_ctime,

Ce sont respectivement les derniers timestamp :
Du dernier accès à un fichier (lecture)
De la dernière modification d'un fichier
Du dernier changement de statut du fichier (par exemple, chmod)

i_blocks

Le nombre de blocs utilisés pour stocker les données du fichier/répertoire associé à l'inode. Notez toutefois qu'un filesystem peut fonctionner en se passant de cette valeur.
Maintenir sa valeur à jour rend les informations fournies par stat et d'autres commandes plus exactes.

i_op

Un lien vers la structure « inode operations » liée à ce fichier. (cf: inode operations)

i_fop

Un lien vers la structure « file operations » liée à ce fichier. (cf: file operations)

i_aops

Un lien vers la structure « address_space_operations » liée à ce fichier. (cf: address_space_operations)

3.2. Superblock

```
fs.h L1316: struct super_block {                                    //...
```

Cette structure que nous avons déjà évoquée contient les informations cruciales du filesystem monté.
Toutefois, il n'y a que deux éléments essentiels que vous devez impérativement définir vous même lors de l'appel de fill_super :

s_op

Un lien vers la structure « super opérations » liée au superblock. Cette structure est cruciale puisque c'est elle qui fournira les fonctions élémentaires qui permettent de parcourir l'arborescence. (cf: super opérations)

s_root

La dentry liée à l'inode décrivant le répertoire / (dentry pouvant être allouée avec int d_alloc_root(struct *inode) une fois la structure inode de / construite par vos soins).

3.3. Super Operations

```
fs.h L1555: struct super_operations {                               //...
```

Cette structure est capitale. Elle contient les pointeurs sur les fonctions qui permettront la dés-allocation du superblock, l'écriture d'une inode sur le disque, sa suppression, sa libération, la création d'une inode, ou encore remonter le filesystem. Les noms des champs sont on ne peut plus explicites.

Voici les fonctions essentielles, la liste complète et documentée figure dans vfs.txt fourni dans la documentation du kernel.

```
//Super operations
static struct super_operations yourfs_super_operations =
  {
    .alloc_inode        = yfs_alloc_inode,          //Allocate inode's memory
    .destroy_inode      = yfs_destroy_inode,        //Free inode's memory
    .write_inode        = yfs_write_inode,          //Write inode's data on device
    .delete_inode       = yfs_delete_inode,         //Inode Destruction
    .put_super          = yfs_put_super,            //Free superblock
    .statfs             = simple_statfs             /* simple statfs */,
    .remount_fs         = NULL,                     //Remount the filesystem
  };
```

F3.3 1 : Une structure Super Opérations

3.4. Inode Operations

Cette structure fournit les pointeurs de fonctions qui permettent d'agir sur le fichier/répertoire. On distinguera donc le cas d'un fichier contenant simplement des données d'un répertoire contenant une liste d'inodes nommées.

Voici deux exemples simples pour ces deux cas. Vous trouverez aussi un exemple pour les liens symboliques. N'hésitez pas à consulter la documentation et les headers des structures

pour avoir la liste complète des champs disponibles.

```
//Repertoires
struct inode_operations yfs_dir_iops =
  {
        //Permet de resoudre un chemin
        .lookup     = yfs_lookup,

        //Crée un fichier, peut se contenter d'appeler mknod
        .create     = yfs_create,
        //Crée un noeud (fichier, répertoire, char/block device, ...)
        .mknod      = yfs_mknod,
        //Crée un repertoire, peut se contenter d'adapter i_nlink et appeler mknod
        .mkdir      = yfs_mkdir,
        //Supprime un repertoire du filesystem
        .rmdir      = yfs_rmdir,
        //Lien "dure"
        .link       = yfs_link,
        //Supprime un fichier
        .unlink     = yfs_unlink,
        //Crée un lien symbolique. On pourra utiliser page_symlink
        .symlink    = yfs_symlink,
  };
```

```
//Fichiers
struct inode_operations yfs_file_iops =
  {
        //Libère les bloques de l'inode suite à un redimensionnement
        .truncate           = sfs_truncate,
        //Permet de récupérer les attributs du fichier
        .getattr            = yfs_getattr,
  };
```

```
//Liens symbolique
struct inode_operations yfs_symlink_iops =
  {
        .readlink             = generic_readlink,
        .follow_link          = page_follow_link_light,
        .put_link             = page_put_link,
        .getattr              = yfs_getattr,
  };
```

3.5. File Operations

A nouveau, les opérations qui peuvent être effectuées sur un nœud dépendent de l'inode considérée. De plus, vous pouvez choisir de rendre la lecture d'un répertoire comme d'un fichier possible ou impossible, etc.

Voici quelques exemples minimaux :

```
struct file_operations yfs_file_ops =
 {
        .llseek             = generic_file_llseek,
        .read               = do_sync_read,
        .aio_read           = generic_file_aio_read,
        .write              = do_sync_write,
        .aio_write          = generic_file_aio_write,
        .mmap               = generic_file_mmap,
        .splice_read        = generic_file_splice_read,
  };
```

```
struct file_operations  yfs_dir_ops =
  {
        .read               = generic_read_dir,
        .readdir            = yfs_readdir,
  };
```

On constate que le kernel fournit de quoi assurer nombre d'opérations élémentaires. Ceci sous réserve de fournir une structure dans le champ a_ops valide.

3.6. Address Space Operations

La structure « Address Space Operations » permet de faire abstraction de la discontinuité des données enregistrées sur le périphérique et de les représenter comme une succession de pages mémoire connexe. La compréhension de cet outil est essentielle.
Dans l'exemple suivant, nous présenterons une utilisation simple où nous nous contenterons de renvoyer le numéro du block contenant la Nième page mémoire d'un fichier. Ceci n'est qu'un cas particulier, et vous retrouverez cette structure dans les filesystem virtuels, qui ne requièrent pas de périphérique.

```
//Maping virtual connex memory of a file into physical blocks
struct address_space_operations sfs_address_space_ops =
  {
        .readpage      = yfs_readpage,
        .writepage     = yfs_writepage,
        .sync_page     = block_sync_page,
        .write_begin   = yfs_write_begin,
        .write_end     = generic_write_end,
        .bmap   = sfs_bmap,
  };
```

```
//Read page with yfs_get_block
static int yfs_readpage
(struct file *file, struct page *page)
{
  printk(KERN_DEBUG "yfs_readpage\n");
  return block_read_full_page(page, yfs_get_block);
}

//Write page with yfs_get_block
static int yfs_writepage
(struct page *page, struct writeback_control *wbc)
{
  printk(KERN_DEBUG "yfs_writepage\n");
  return block_write_full_page(page, yfs_get_block, wbc);
}

//Prepare Write page
int __yfs_write_begin
(struct file *file, struct address_space *mapping,
 loff_t pos, unsigned len, unsigned flags,
 struct page **pagep, void **fsdata)
{
  printk(KERN_DEBUG "__yfs_write_begin\n");
  //Called by us, so pagep is initialised
  return block_write_begin(file, mapping, pos, len, flags, pagep, fsdata, yfs_get_block);
}

//Prepare Write page with yfs_get_block
static int yfs_write_begin
(struct file *file, struct address_space *mapping,
 loff_t pos, unsigned len, unsigned flags,
 struct page **pagep, void **fsdata)
{
  printk(KERN_DEBUG "yfs_write_begin\n");
  //Called by kernel, *pagep is uninitialised!
  *pagep = NULL;
  return block_write_begin(file, mapping, pos, len, flags, pagep, fsdata, yfs_get_block);
}
```

La fonction yfs_get_block associe une partie d'un fichier au block où se situent les données de cette partie. Voici un aperçu de ce à quoi peut ressembler cette fonction :

```
//Associate logical block to physical block
int yfs_get_block
(struct inode *inode, sector_t iblock, struct buffer_head *bh_result, int create)
{
  //...
  int                   err = -EIO;

  //Invalid negativ block number!
  if (iblock < 0)
    {
      printk("YFS-warning: In get_block, invalid iblock = %u\n", (unsigned int)iblock);
      return err;
    }

  //Search the grate block from inode's data and store result in blk
  if((err = yfs_find_block(inode, /* ... */ &blk)) < 0)
    goto err;
  //Block found
  if (blk)
    goto map;

  //Get a new block and mark it used in fs block's list.
  if((err = yfs_alloc_block(inode, /* .. */ &blk) < 0))
    goto err;

  //Map block into bh
 map:
  map_bh(bh_result, inode->i_sb, blk);
  return 0;

 err:
  return err;
}
```

Ici, tout le travail est relégué à yfs_find_block et yfs_alloc_block.

4. Initialisation d'un module de filesystem

Nous allons présenter ici quelques lignes de code correspondant aux actions élémentaires que tout module de filesystem effectue.

4.1. Initialisation du module

Avant toute chose, il est nécessaire de rédiger l'entry point du module, qui correspond au code appelé au chargement du module.

Voici donc un exemple simple et plutôt minimaliste.

```
//Module Entry point
int     yfs_init_module(void)
{
  int   err;

  printk("YFS-fs: init_module\n");

  //Allocate inode cache
  yfs_inode_cache = kmem_cache_create("yfs_inode_cache",
                                      sizeof(struct yfs_inode_info),
                                      0,
                                      0,
                                      yfs_init_once);
  if (!yfs_inode_cache)
    return -ENOMEM;

  //Register filesystem
  err = register_filesystem(&yfs_fs_type);
  return (err);
}

//...
module_init(yfs_init_module);
```

On notera la création d'un cache (yfs_inode_cache est une globale) afin de disposer d'un pool mémoire pour l'allocation des inodes du filesystem. yfs_inode_info est une structure qui englobe une `struct inode*`, car comme précisé plus haut, nous pourrions nécessiter des données supplémentaires, telle que la position des bloques d'un fichier sur le disque.

La fonction yfs_init_once sera appelée à l'allocation d'une inode. Ici, elle se contente d'appeler `inode_init_once(&inode->vfs_inode)` où vfs_inode est une `struct inode`.

4.2. Libération du module

Une fois son devoir accompli, le module doit libérer les ressources allouées, et supprimer le filesystem de la liste des fs disponibles.

Ce que font ces quelques lignes :

```
//Module Exit
static void     sfs_cleanup_module(void)
{
  printk("SFS-fs: cleanup_module\n");

  //Free inode cache
  kmem_cache_destroy(sfs_inode_cache);

  //Unregister FS
  unregister_filesystem(&sfs_fs_type);
}

module_exit(sfs_cleanup_module);
```

4.3. Allouer une inode

Afin d'illustrer une utilisation du cache créé dans l'exemple précédent, voici quelques lignes de deux fonctions d'une structure super_opérations.

```
struct inode*
sfs_alloc_inode(struct super_block *sb)
{
  struct yfs_inode_info *ii;

  printk(KERN_DEBUG "YFS: alloc_inode\n");
  if (!(ii = kmem_cache_alloc(yfs_inode_cache, GFP_KERNEL)))
    return NULL;
  return &ii->vfs_inode;
}
```

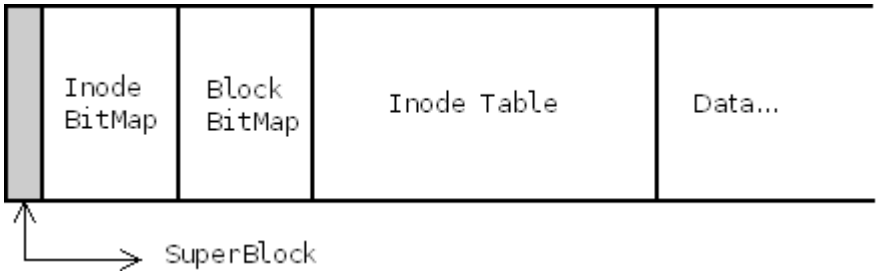
```
static void
yfs_destroy_inode(struct inode *inode)
{
  printk(KERN_DEBUG "YFS: destroy_inode\n");

  kmem_cache_free(yfs_inode_cache, inode);
}
```

5. Quelques informations concernant

5.1. La segmentation en blocks pour les blocks device

Comme je l'ai laissé entendre, sans jamais développer le sujet, les données d'un fichier (ou répertoire) sont souvent fragmentées sur le disque. Cela dépend de la topologie du filesystem, mais il est bien souvent plus simple de prendre le premier block libre que l'on trouve que de déplacer d'importantes masses de données sous prétexte que l'on se refuse à fragmenter un fichier.

Dans le cas de minix, la partition est divisée comme ceci :



Les deux premiers éléments dressent une liste des inodes disponibles sur la partition, puis des blocks disponibles (c'est à dire l'espace de la partition en octets divisé par la taille d'un block, elle aussi en octets. Le superblock est alors le premier block de cette liste).

Suit la table des inodes, qui est l'espace réservé aux inodes. Ainsi, le nombre d'inodes maximum sur une partition est défini lors du formatage de celle-ci.

Pour un système type ext2, on retrouvera une structure similaire, qui se répète à plusieurs reprises sur la partition, et dispose donc de copies du superblock en cas de corruption.

5.2. Lookup

Je n'ai fait que mentionner le nom de cette fonction, pourtant cruciale.

Lookup est la fonction qui permet de résoudre un path (chemin) vers un fichier/répertoire, et donc d'obtenir la dentry de cette élément.

Concrètement, la résolution de « /dir/subdir/file » produira les appels suivants :

```
dir_dentry    = lookup(root_dentry, « dir »)
subdir_dentry = lookup(dir_dentry, « subdir »)
file_dentry   = lookup(subdir_dentry, « file »)
```

Nous disposons de la dentry associée au point de montage grâce au super_block. Par la suite, on parcourt l'arbre (si bien sûr les nœuds existent).

Cette fonction est implémentée par le filesystem. Elle a un comportement similaire à readdir, puisqu'il lui faut lire le contenu d'un dossier pour obtenir l'inode du nœud suivant, et créer/récupérer la dentry associée.

5.3. Readdir

Cette fonction est chargée de lister le contenu d'un répertoire, en fournissant des paires (<nom de fichier>,<inode id>).

Dans un filesystem type minix, un dossier est simplement un fichier subdivisé en structure contenant ces paires. Ajouter un lien physique revient alors simplement à ajouter une structure pointant sur une inode existante. De même, supprimer un lien vers un fichier, c'est simplement supprimer cette structure (si l'inode du fichier considéré se retrouve avec un i_nlink de 0, c'est qu'elle doit être supprimée).
On comprend alors pourquoi le nom d'un fichier ne figure pas dans une inode, mais bien dans chaque dossier lié à cette inode.

Voici un exemple de ce à quoi peut ressembler cette fonction :

```
//Dans cette exemple, la taille d'une structure elementaire representant une paire n'est
pas fixe.
static int      yfs_readdir
(struct file *file, void *dirent, filldir_t filldir)
{
  //Get page index
  unsigned long pidx            = file->f_pos >> PAGE_CACHE_SHIFT;
  //Get data offset
  unsigned long offset          = file->f_pos & ~PAGE_CACHE_MASK;
  //File's inode
  struct inode* inode           = file->f_dentry->d_inode;
  //Parent directory
  struct inode* parent          = file->f_dentry->d_parent->d_inode;
  //How much pages in this file?
  unsigned long cnt_pages       = yfs_count_pages(inode);
  //Page used
  struct page   *page           = NULL;
  //yfs directory entry
  struct yfs_dirent*            dent;
  //Page address
  char                          *kaddr;
  //Len name
  int   len;

  //Page doesn't exist!
  if (file->f_pos > inode->i_size + IDX_ADD)
    goto done;

  //On each page
  for(;pidx < cnt_pages; pidx++)
    {
      //Get page or next page
      page = yfs_get_page(inode, pidx);
      if (IS_ERR(page))
        continue;

      //Lock page
      lock_page(page);

      //End is ino = 0 (An entry can't be broken into 2 pages!)
      kaddr = (void*)page_address(page);
      dent = (void*)kaddr + offset;
      while (dent->d_ino)
        {
          len = strlen(dent->d_name);
          offset = (char*)dent - kaddr;
          //Fill file (unknow type, fs will know by checking inode latter)
          if (filldir(dirent, dent->d_name, len,
                      file->f_pos = ((pidx << PAGE_CACHE_SHIFT) | offset),
                      dent->d_ino, DT_UNKNOWN))
            {
              //Unlock the page
              unlock_page(page);
              //filldir error
              yfs_put_page(page);
              goto out;
            }
          dent = yfs_next_dentry(dent, len);
        }
```

```
      //Unlock and put page
      unlock_page(page);
      yfs_put_page(page);
      offset = 0;
    }

  //. and .. are special directory
  if (offset == 0 && filldir(dirent, ".", 1,
                             file->f_pos = ((pidx << PAGE_CACHE_SHIFT) | offset),
                             inode->i_ino, DT_DIR))
    goto out;
  offset = 1;
  filldir(dirent, "..", 2,
          file->f_pos = ((pidx << PAGE_CACHE_SHIFT) | offset),
          parent->i_ino, DT_DIR);

   //Finish, . .. and others writed.
  done:
   file->f_pos = inode->i_size + IDX_ADD + 1;
   return 1;

   //Out, all dirent aren't already writed
  out:
   return 0;
}
```

6. Pour aller plus loin...

Il y a tant de chose à dire, je n'ai pas abordé les buffer_head, je n'ai que succinctement décrit la façon dont un filesystem maintient une liste des inodes et blocks utilisés, je n'ai pas non plus mentionné les différences structurelles entre minix et FAT, je n'ai que brièvement abordé la structure d'un répertoire... Il y a tant à dire, bien assez pour écrire un livre. Je n'ai malheureusement pas la prétention de pouvoir réaliser un tel ouvrage, alors je vous renverrai vers ceux existants qui sont certainement bien plus clairs que tout ce que je pourrais écrire.

J'espère avec cet article avoir éveillé votre curiosité, vous avoir fait découvrir ce domaine et peut être vous amener à en lire plus sur le sujet.

Pour mieux comprendre le fonctionnement d'un filesystem, rien ne vaut l'analyse d'un filesystem existant et fonctionnel, voire (si vous disposez du temps nécessaire) l'implémentation d'un filesystem.

Je recommande alors particulièrement la lecture du code source du filesystem minix, qui vous évitera de vous perdre dans la complexité du filesystem et de vous concentrer sur l'interface avec le kernel.

Le code de ce dernier est clair, explicite, concis, et particulièrement représentatif de l'implémentation d'un filesystem sous linux (on retrouvera en effet la même hiérarchie, la même logique de gestion d'erreurs et de segmentation des fonctions que dans un filesystem plus complexe comme ext2).

Le lien suivant correspond au repository git de l'implémentation de minix dans le kernel 2.6 : http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=tree;f=fs/minix;hb=HEAD

- - -

Vous pouvez aussi, si vous le souhaitez, consulter le code d'un filesystem (non fonctionnel) minimaliste inspiré de minix où le code est commenté (rédigé dans l'optique de mieux comprendre ce domaine) : http://zenol.fr/trac/dev/browser/sfs

- - -

Enfin, gardez à l'esprit que les exemples de ce texte ne sont qu'une infime partie des possibilités offertes par VFS, et ne font pas règle de conduite. Vous trouverez de nombreux modules de filesystem qui nécessiteront d'autres fonctionnalités, n'utilisant pas ou peu les fonctions fournies par défaut car inadaptées. De plus, un filesystem n'est pas nécessairement destiné à être utilisé toujours monté sur un block device (pensez à /proc) et certains fs sont adaptés à certains périphériques (comme par exemple Unified Flash File System)

Bibliographie

[DAV 97] David A. Rusling, « The VFS Inode », Avril 1997
[RAV 07] Ravi Kiran UVS, « Writing a Simple File System », Octobre 2007
[WIK 10] Wikipedia, « VFS », Aujourd'hui
[LLD 01] Alessandro Rubini & Jonathan Corbet, « Linux Device Drivers », 2001
[TAN 08] Andrew Tanenbaum , « Systèmes d'exploitation », Septembre 2008

Remerciements

Merci à Furr pour sa relecture orthographique, à dut et à toute l'équipe de developpez.com pour leur soutien.