# PNL - Projet

Système de fichiers

**Redha Gouicem – Julien Sopena**

Dans ce projet, vous implémenterez un système de fichiers sous la forme d'un module compatible avec la version 4.9.83 de Linux. N'hésitez pas à utiliser un site de référencement croisé du code du noyau Linux pour vous aider (par exemple elixir). Il est fortement recommandé de s'inspirer des autres systèmes de fichiers présents dans le dossier `fs/` des sources du noyau, et de lire la documentation du fichier `Documentation/filesystems/vfs.txt`

La soumission, ainsi que le choix des groupes, sera fait via la plateforme `Moodle` de l'université. Vous devrez soumettre, pour chaque groupe, une archive `tar.gz` contenant votre rapport au format `pdf` ainsi que les sources de votre module noyau.

## Rappel : Lire attentivement l'énoncé !

## 1 Structures génériques du noyau Linux

### 1.1 Architecture

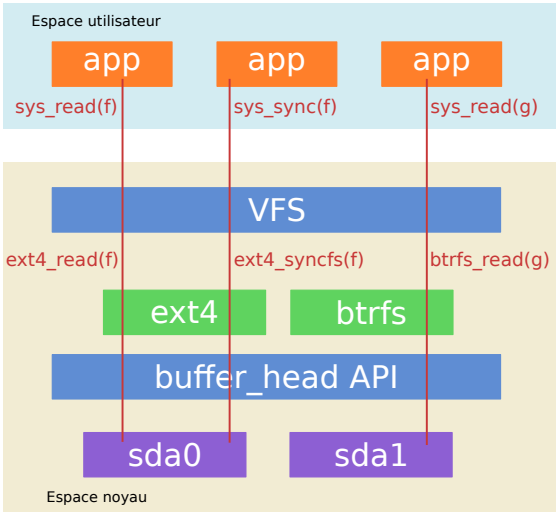



FIGURE 1 – Interface user/partition disque dans Linux (`sda0=ext4` et `sda1=btrfs`)

Figure 1, un appel système (eg. `sys_read()`, `sys_write()`) passe par le VFS, qui appelle l'implémentation du système de fichiers concerné. Les systèmes de fichiers (FS) interagissent avec les périphériques bloc via l'API `buffer_head` présentée plus loin dans ce sujet.

## 1.2 Structure de notre système de fichiers

Notre FS a la structure présentée en figure 2 :

— le *superblock* contient des métadonnées sur la partition, et est stocké dans le premier bloc de la partition (bloc 0),

— l'*inode store* contient toutes les données persistantes des inodes utilisables sur cette partition, et est stocké sur un nombre variable de blocs après le *superblock*,

— les blocs suivants contiennent 2 bitmaps : celui permettant de savoir si une inode est libre ou non, et celui permettant de savoir si un bloc est libre ou non,

— le reste de la partition contient les données.

L'unité d'accès à un disque est le bloc. Pour ce projet, un bloc est d'une taille fixe de 4096 octets. Pour plus de détails, se référer au fichier `pnlfs.h`.

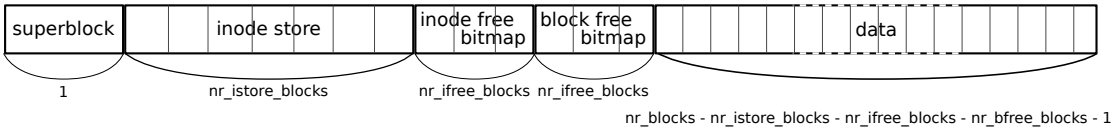
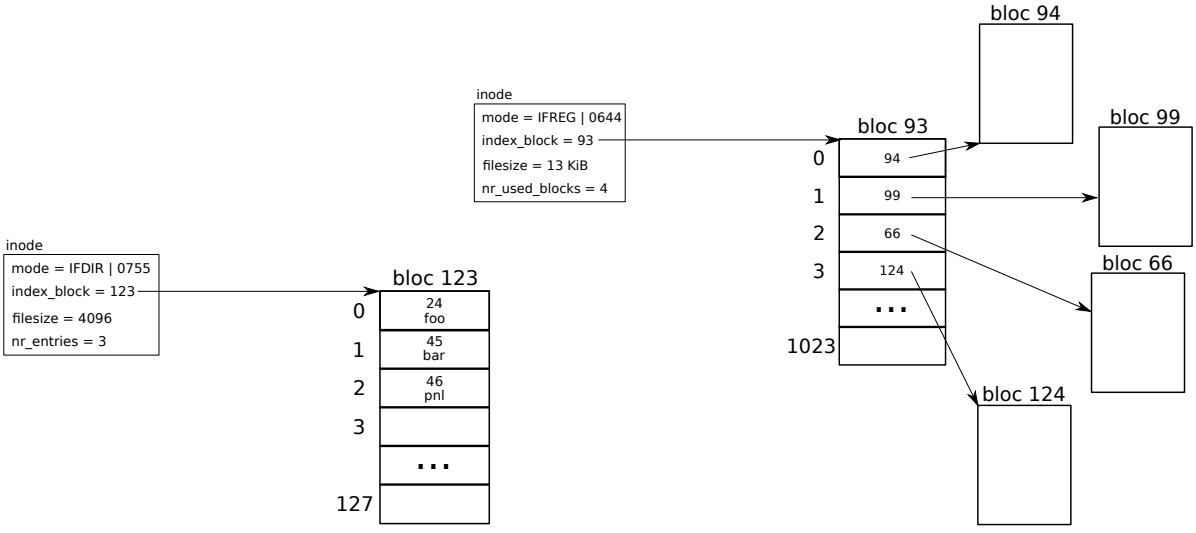



FIGURE 2 – Structure d'une partition `pnlFS`

**Inodes** Chaque fichier et répertoire est décrit par une inode dans l'inode store. Une inode de répertoire (fig 3a) pointe vers un bloc contenant la liste des fichiers et répertoires contenus dans ce dernier. Une inode de fichier (fig 3b) pointe sur un bloc d'indirection vers des blocs contenant les données du fichier (pas de double ou triple indirection, ni de liens directs). L'*inode store* est un simple tableau d'inodes.




(a) Inode de répertoire (b) Inode de fichier

FIGURE 3 – Structures inode sur le disque

**Bitmaps** Les bitmaps d'inodes et blocs libres sont représentés sur le disque comme un tableau de bitmaps de 64 bits (ie. l'état du bloc 4 se situe dans le bit 4 du bitmap 0, l'état du bloc 70 se situe dans le bit 6 du bitmap 1, voir figure 4).

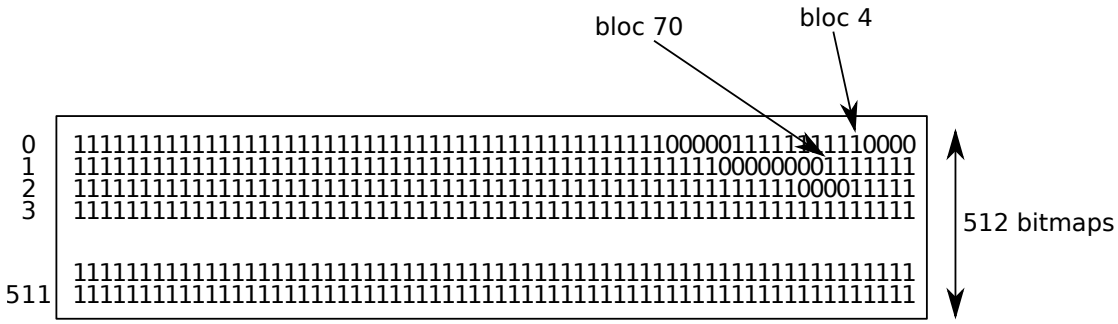



FIGURE 4 – Bitmap de blocs/inodes libres

**Endianness** Afin qu'un système de fichier soit portable entre plusieurs architectures matérielles d'*endianness* différentes, il est impératif de choisir un format d'écriture sur le disque et de lire/écrire correctement les métadonnées (*superbloc*, *inodes*, *bitmaps*). Pour ce projet, les métadonnées sont écrites sur le disque en *little endian* (type `__le32` dans `pnlfs.h`). Toute lecture/écriture depuis/vers le disque doit donc utiliser les fonctions `leX_to_cpu()` et `cpu_to_leX()`, avec X la taille de la donnée en bits (cf. `include/linux/byteorder/generic.h`). Une fois la donnée en mémoire, elle peut être manipulée normalement.

Pour plus de détails sur `pnlFS`, consulter le fichier `pnlfs.h`, ainsi que le code source du programme `mkfs-pnlfs`.

**Création d'une image disque** Vous pouvez créer une image disque de 30MiB avec notre système de fichiers grace au programme `mkfs-pnlfs` fourni :

```
dd if=/dev/zero of=disk.img bs=1M count=30
```

```
./mkfs-pnlfs disk.img
```

Cette image contient un fichier `foo` à la racine.

# 2 Remarques utiles

— Ne pas hésiter à regénérer une image propre avec `mkfs-pnlfs` en cas de problèmes
— De même, ne pas hésiter à redémarré la VM pour avoir un Linux sain
— Compilation séparée d'un module noyau (dans le Makefile) :

```
obj-m += foo.o bar.o  # création de 2 modules (foo.ko et bar.ko)
foo-objs := toto.o fifi.o # dépendances de foo.ko (et donc toto.c et fifi.c)
bar-objs := tutu.o fifi.o # dépendances de bar.ko
ccflags-y := -DDEBUG -O1 # flags passés à gcc à la compilation
```

— Valeurs de retour des fonctions, voir les fichiers :
  • `include/uapi/asm-generic/errno-base.h`
  • `include/uapi/asm-generic/errno.h`
  • Conversion errno → pointeur : `ERR_PTR(errno)`
  • Conversion pointeur → errno : `PTR_ERR(errno)`

# 3 Enregistrement du système de fichiers

La première étape consiste à enregistrer notre FS lors de l'insertion de notre module. La `struct file_system_type` décrit un système de fichiers :

```
struct file_system_type {
        const char *name;
        int fs_flags;
        struct dentry *(*mount) (struct file_system_type *, int,
                                 const char *, void *);
        void (*kill_sb) (struct super_block *);
        struct module *owner;
        struct file_system_type *next;
};
```

Créer un système de fichiers en définissant une `struct file_system_type` en suivant la documentation du noyau. Dans un premier temps, laisser les fonctions `mount()` et `kill_sb()` vides, et enregistrer votre système de fichiers avec la fonction `register_filesystem()`. Pour retirer votre FS, utiliser la fonction `unregister_filesystem()`.
Vérifier que votre FS est présent dans le fichier `/proc/filesystems`
*Note :* notre FS requiert un périphérique...

# 4 Montage d'une partition

**Fonction mount()** La fonction `mount()` de la `struct file_system_type` est appelée pendant l'appel système `sys_mount`. Le noyau fourni des fonctions simplifiant le montage de périphériques selon leur type ( `mount_bdev()` par exemple). Ces fonctions prennent en paramètre un callback que nous compléterons plus tard (vide pour le moment, retournant -1).

**Fonction kill_sb()** Cette fonction est appelée lors de la destruction d'une partition (appel système `sys_umount`). Elle doit appeler la fonction `kill_block_super()` sur le superbloc en paramètre.

Pour monter une image `test.img` dans le dossier `dir/`, utiliser la commande :
`mount -t <fsname> -o loop test.img dir/`
*Note :* l'option `-o loop` permet de monter le fichier `test.img` sur un des périphériques `/dev/loop*`
Vérifier que le montage d'une partition échoue (puisque votre callback renvoie -1) et que toutes vos fonctions sont appelées (noter l'ordre d'appel).

# 5 Superbloc

Maintenant que notre FS est enregistré et permet de monter une partition, il faut lire les métadonnées de la partition afin de correctement manipuler les données. Une partition est représentée par une `struct super_block` :

```
struct super_block {
        unsigned long                   s_magic;     /* FS magic number */
        unsigned long                   s_blocksize;/* Taille d'un bloc */
        loff_t                          s_maxbytes; /* Taille max d'un fichier */
        const struct super_operations *s_op;        /* Handler pour superblock */
```

```
        void                            *s_fs_info; /* info fs specifique */
        struct dentry                   *s_root;    /* dentry racine */
};
```

**Callback fill_super() [partie 1]** Le callback `fill_super()` passée à `mount_bdev()` lors du `mount` doit :

- initialiser la `struct super_block` passée en paramètre,
- allouer les données spécifiques à notre FS,
- créer l'inode racine ainsi que la dentry correspondante.

Dans un premier temps, nous n'allons implémenter que les deux premiers points. Écrire un callback `fill_super()` initialisant les trois premiers champs de la `struct super_block` présentés ci-dessus. Lire ensuite le super bloc ( `struct pnl_superblock` dans `pnlfs.h`) sur le périphérique disque à l'aide de l'API `buffer_head` (annexe 12.1). Le champ `s_fs_info` peut être utilisé pour stocker des données spécifiques au FS, comme celles de la `struct pnlfs_sb_info` (`pnlfs.h`). Pour le moment, retourner une erreur même en cas de succès pour éviter un crash du noyau, car la fonction est pour le moment incomplète (il manque l'inode racine attendue par le VFS).
*Note :* Attention, les bitmaps doivent être copiés par lignes de 64 bits !

**Fonction put_super()** Les opérations agissant sur le super bloc sont définies dans la structure `struct super_operations` ci-dessous. Définir une `struct super_operations` et implémenter la fonction `put_super()`. Cette fonction doit défaire ce que la fonction `fill_super()` a fait. Une fois cette fonction écrite, affecter votre `struct super_operations` au champ `s_op` dans `fill_super()`.

```
struct super_operations {
        void (*put_super) (struct super_block *);
        struct inode *(*alloc_inode) (struct super_block *);
        void (*destroy_inode) (struct inode *);
        int (*write_inode) (struct inode *, struct writeback_control *);
        int (*sync_fs) (struct super_block *, int);
        int (*statfs) (struct dentry *, struct kstatfs *);
};
```

**Création de l'inode racine** Afin de compléter la fonction `fill_super()`, il faut désormais initialiser l'inode racine, et donc lire une inode sur le disque. Puisque la lecture d'une inode sur le disque sera une opération nécessaire à plusieurs reprises dans notre FS, nous allons écrire le constructeur et le destructeur d'une inode, ainsi qu'une fonction de lecture d'inode.

Le constructeur et le destructeur d'inode correspondent aux fonctions `alloc_inode()` et `destroy_inode()` de la `struct super_operations`. Le constructeur doit allouer une structure `struct pnlfs_inode_info`, initialiser la `struct inode` qu'elle embarque avec la fonction `inode_init_once()` et renvoyer l'adresse de cette `struct inode`. Le destructeur doit simplement désallouer les données allouées par le constructeur.

Une fois ces fonctions écrites, écrire une fonction avec la signature suivante pour lire une inode sur le disque :
`struct inode *pnlfs_iget(struct super_block *sb, unsigned long ino);`
Cette fonction devra demander une `struct inode` au VFS (qui appellera le constructeur défini précédemment) avec la fonction `iget_locked()`. Si cette inode était en cache, il suffit de la retourner. Sinon, vous devez lire cette inode sur le disque et initialiser les champs ci-dessous. Vous pouvez initialiser toutes les dates avec la macro `CURRENT_TIME`, et les champs `i_op` et `i_fop` avec des structures vides pour le moment.

```
struct inode {
        umode_t                         i_mode;
        const struct inode_operations *i_op;
        const struct file_operations  *i_fop;
        struct super_block             *i_sb;
        unsigned long                   i_ino;      /* numero d'inode */
        loff_t                          i_size;     /* taille occupée en octets */
        blkcnt_t                        i_blocks;   /* nb blocs occupés */
        struct timespec                 i_atime;    /* date dernier acces */
        struct timespec                 i_mtime;    /* date derniere modif */
        struct timespec                 i_ctime;    /* date creation */
};
```

En cas de succès, retourner l'inode après l'avoir débloqué avec la fonction `unlock_new_inode()`. En cas d'échec, retourner une erreur après avoir libéré l'inode avec la fonction `iget_failed()`.

**Callback fill_super() [partie 2]** À l'aide de ces nouvelles fonctions, récupérer l'inode racine du disque (inode 0), définir ses propriétaires avec la fonction `inode_init_owner()`, puis la déclarer comme inode racine (champ `s_root` de la `struct super_block`) avec la fonction `d_make_root()`.

Le montage et démontage de partitions doivent maintenant fonctionner sans erreur.

# 6 Recherche d'inode

L'accès au dossier où l'image est montée est toujours impossible car le VFS n'est pas capable de trouver l'inode correspondante au nom du fichier dans le cache des dentry. Pour résoudre ce problème, implémenter la fonction `lookup()` de la `struct inode_operations`. Vous pouvez pour cela suivre la documentation du VFS (cf. 12.4), à l'exception de l'incrémentation du champ `i_count` de l'inode (probablement une erreur de la doc...).

Il est maintenant possible d'accéder au dossier racine de votre image montée !

# 7 Opération sur les répertoires

Afin de pouvoir connaître la contenu d'un répertoire, implémenter la fonction `iterate_shared()` de votre `struct file_operations`. Cette fonction est appelée de façon itérative avec un contexte maintenu entre les appels. Ce contexte contient notamment la liste des fichiers présents dans le répertoire, ainsi que leur nombre. Ajouter les fichiers particuliers . et .. avec la fonction `dir_emit_dots()`, puis ajouter, sur le même modèle, les autres fichiers présents dans ce répertoire avec la fonction `dir_emit()` (pour le type, vous pouvez utiliser `DT_UNKNOWN`).

Vous devriez désormais pouvoir lister le contenu d'un répertoire avec la commande `ls`.

# 8 Opérations sur les inodes

La plupart des fonctionnalités d'un FS sont effectuées au niveau de l'inode, via la structure `struct inode_operations`. Implémentez les fonctions suivantes, en vous aidant de la documentation du VFS dans les sources du noyau, des annexes (12.1, 12.2, 12.3) et du code d'autres systèmes de fichiers de Linux :

```
struct inode_operations {
        struct dentry *(*lookup) (struct inode *, struct dentry *,
                                  unsigned int);
        int (*create) (struct inode *, struct dentry *, umode_t, bool);
        int (*unlink) (struct inode *, struct dentry *);
        int (*mkdir) (struct inode *, struct dentry *, umode_t);
        int (*rmdir) (struct inode *, struct dentry *);
        int (*rename) (struct inode *, struct dentry *,
                       struct inode *, struct dentry *, unsigned int);
};
```

Vous devriez désormais être capable de créer, renommer et supprimer des fichiers et des répertoires dans votre FS.

# 9 Opérations sur les fichiers

Les opérations sur les fichiers sont implémentées dans la `struct file_operations` de l'inode. Implémenter les fonctions suivantes, en plus de la fonction `iterate_shared()` précédemment implémentée :

```
struct file_operations {
        struct module *owner;
        int (*iterate_shared) (struct file *, struct dir_context *);
        ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
        ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
};
```

*Note :* pour la fonction `write`, vous devez déplacer le curseur à la fin du fichier si ce dernier a été ouvert avec le flag `O_APPEND`.

Vous devriez désormais pouvoir lire et écrire des fichiers dans votre FS.

# 10 Persistance

Si la partition est démontée, les modifications ne sont jamais écrites sur le disque (notamment les informations contenues dans le super bloc ou les inodes). Pour rendre notre FS persistant, implémenter les fonctions `sync_fs()` et `write_inode()` de la `struct super_operations`. Ces fonctions sont appelées par l'appel système `sys_sync` afin de forcer l'écriture des données sur le disque dur, par un daemon périodique, ainsi que lors du démontage d'une partition. `sync_fs()` doit synchroniser les données globales au FS (super bloc, bitmaps), et `write_inode()` est appelé sur les inodes marquées *dirty* afin qu'elles soient écrites sur le disque.

# 11 Pour aller plus loin...

Vous pouvez implémenter d'autres fonctionnalités pour votre système de fichiers, telles que :

- la fonction `statfs` du super bloc, pour l'appel à `df`,
- la gestion des liens symboliques,
- la gestion des tmpfiles,
- la gestion du page cache,
- ou une autre de votre choix !

# 12 Annexe

## 12.1 Buffer head (include/linux/buffer_head.h)

`struct buffer_head *sb_bread(struct super_block *sb, sector_t block bno);` : renvoie un pointeur sur le buffer en cache du bloc `bno` de la partition `sb`. La taille du buffer lu correspond à `sb->s_blocksize`, et les données de ce buffer sont accessible via le champ `b_data` de la `struct buffer_head` renvoyée.

`void mark_buffer_dirty(struct buffer_head *bh);` : signal que le buffer `bh` a été modifié et devra être recopié sur le disque.

`int sync_dirty_buffer(struct buffer_head *bh);` : force l'écriture du buffer `bh` sur le disque. Renvoie 0 en cas de succès.

`void brelse(struct buffer_head *bh);` : relâche la référence sur le buffer `bh`.

## 12.2 Inode (include/linux/fs.h)

`void mark_inode_dirty(struct inode *inode);` : signale une modification de l'inode. Lorsque celle-ci sera libérée, la fonction `write_inode()` sera appelée sur cette inode.

`void iput(struct inode *inode);` : relâche la référence sur l'inode. Si le compteur atteint 0, la focntion `destroy_inode()` sera appelée sur cette inode.

## 12.3 Bitmap (include/linux/bitmap.h)

`unsigned long find_first_bit(const unsigned long *addr, unsigned long size);` : renvoie la position du premier bit à 1 dans le bitmap `addr` de taille `size`.

`void bitmap_clear(unsigned long *map, unsigned int start, int len);` : met à 0 les bits `start` à `start + len` du bitmap `map`.

`void bitmap_set(unsigned long *map, unsigned int start, int len);` : met à 1 les bits `start` à `start + len` du bitmap `map`.

## 12.4 Références

**Documentation du VFS** `https://elixir.bootlin.com/linux/v4.9.83/source/Documentation/filesystems/vfs.txt`

**Article LWN expliquant la création d'un système de fichiers** `https://lwn.net/Articles/13325/`