Rapport : étude des systèmes de fichiers Ext

Léonard RIGAL, Adrian HEOUAIRI, Sevi DERVISHI, Edouard Patrick HOSSOU

20 mars 2024

Table des matières

1 Introduction **3**

2 L'organisation des recherches **3**
2.1 Equipe Fuse . . . 3
2.1.1 Compréhension de la documentation . . . 3
2.1.2 Lecture de projets existants . . . 3
2.2 Equipe Ext2 . . . 3
2.2.1 Quels sont les différents systèmes de fichiers ? . . . 3
2.2.2 Quelles sont les différences entre Ext2, Ext3 et Ext4 ? . . . 4

3 Mise en commun des recherches et code **4**
3.1 Premier faux départ . . . 4
3.2 Implémentation d'un "Ext2-like" . . . 4
3.2.1 Data-structure . . . 4
3.2.2 Implémentations des fonctions principales . . . 6

4 Tests **7**
4.1 Makefile . . . 7
4.2 Exécution du code . . . 7

5 Conclusion **7**

6 Annexe **8**

1 Introduction

Notre groupe a choisi de se concentrer sur l'étude des systèmes de fichiers et plus précisément sur ext. Pour cela, on a divisé le travail de la façon suivante : recherches sur les systèmes de fichiers d'un côté et implémentation avec fuse de l'autre.

2 L'organisation des recherches

2.1 Equipe Fuse

2.1.1 Compréhension de la documentation

2.1.2 Lecture de projets existants

2.2 Equipe Ext2

Afin de choisir quel sytème implémenter avec Fuse et comment, il nous a fallu faire des recherches sur les différents systèmes de fichiers sous Linux.

2.2.1 Quels sont les différents systèmes de fichiers ?

Il y a plusieurs manières de différentier un système de fichiers d'un autre : la taille limite des fichiers, la taille des partitions, la gestion des droits, la structure/l'organisation même du système de fichiers, la journalisation etc.
C'est ce qui va amener la compatibilité ou non-compatibilité d'un système de fichiers avec un système d'exploitation. Mais aussi les choix stratégiques pour choisir le bon système de fichiers associé au support (par exemple clé USB, SSD, etc).
Linux est capable d'exploiter différents systèmes de fichiers (même ceux qui ne sont pas natifs), comme par exemple ext2, ext3, ext4, FAT16, FAT32, NTFS, HFS, BtrFS, JFS, XFS, etc. Chaque système de fichiers réside sur un disque logique (les disques physiques peuvent être divisé en plusieurs disques logiques).
On a donc notre système "principale" monté à la racine / mais on peut monter d'autres systèmes grâce à la commande *mount* . C'est grâce à ça qu'on va pouvoir tester notre implémentation Fuse.
De manière générale, aujourd'hui le systèle de fichiers Linux est ext4 et pour Windows c'est NTFS. Pour les clé usb on a tendance à favoriser le Fat32. nous allons donc présenter ces trois systèmes :

Nom	Taille maximale d'un fichier	Taille maximale d'une partition	Journalisation	Gestion des droits d'accès*
Ext4	16 TiB	1 EiB	Oui	Oui
NTFS	16 TiB	256 TiB	Oui	Oui*
Fat32	4 GiB	8 TiB	Non	Non*

Organisation

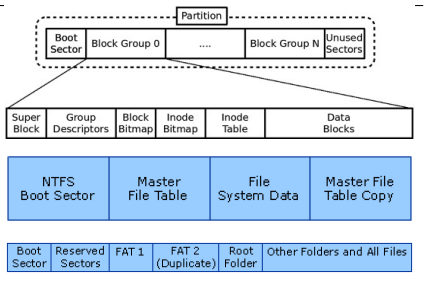

TABLE 1 – Comparaison des systèmes de fichiers

* : ici on parle de gestion des droits sous Linux quand on monte un système NTFS et Fat32
On ne va pas forcément entrer dans les détails de NTFS et FAT32 car ce n'est pas notre sujet ici, mais

il était important de les mentionner pour montrer la spécificité d'Ext.

A ÉTOFFER

2.2.2 Quelles sont les différences entre Ext2, Ext3 et Ext4 ?

Dans la sous partie précédente, on a comparé Ext4 à NTFS et FAT32. Ces comparaisons sont beaucoup plus simple car Ext4 et NTFS ont une organisation très différente, mais des caractéristiques proches. Cette organisation va alors jouer sur d'autres paramètres : vitesse lors du montage, création de dossier, accès aux données, fiabilité etc.

A FINIR

3 Mise en commun des recherches et code

3.1 Premier faux départ

3.2 Implémentation d'un "Ext2-like"

Il s'agit ici d'une vision très simplifiée d'un système Ext2, c'est pour ça qu'on le définit comme "Ext2-like", car beaucoup de détails diffèrent.

3.2.1 Data-structure

Notre structure de données est définie dans *data_structure.h* et ses fonctions associées sont implémentées dans *data_strucure.c* .

Ici on définit deux structures : un fe4_inode et un fe4_dirent.

```
 // An inode knows its own inode number (index) because of stat.st_ino
// Doesn't contain filename
typedef struct fe4_inode fe4_inode;
struct fe4_inode {
  struct stat stat;
  char contents[CONTENTS_SIZE]; // Must not be accessed directly
};

typedef struct fe4_dirent fe4_dirent;
struct fe4_dirent {
  ino_t inode_number;
  char filename[256];
};
```

Notre fichier racine aura toujours pour node id 0.
Notre structure est un buffer de 256 i-node, avec les i-nodes stockés directement en RAM. Dans notre cas, les valeurs *uid_t* et *gid_t* seront toujours l'utilisateur courant. A savoir aussi que les noms de fichiers ne pourront pas faire plus que 256 octets.

Regardons maintenant de plus près les fonctions implémentées pour la structure :

```
 void init_inodes(void);
```

Pour cette fonction, comme dit précédemment, on va créer 256 i-node. Tant qu'ils ne sont pas utilisés, leur taille est initialisée à -1. Et on défini le premier dossier pour la racine. On arrive donc à

notre deuxième fonction :

```
fe4_inode *get_new_dir_inode(ino_t parent_inode_number);
```

Cette fonction prend l'id du parent et renvoi un i-node. Dans le cadre de la création du dossier racine, l'id du dossier parent et le même que l'id du dossier lui-même : 0.
En premier on va récupérer le premier i-node disponible. Pour ça, c'est très simple, il suffit de parcourir notre tableau d'i-node et de récupérer le premier qui a une taille égale à -1 et de renvoyer son id.
On va alors appeler la fonction *get_inode_at*qui renvoi l'i-node associé à l'id.
Ensuite on fait un memset et on passe tout à 0. Et là on définit les différentes valeurs de l'i-node (id, uid,gid,droits,etc).
Enfin, on oublie pas de créer les dirent suivants : dossier courant et le dossier parent, qui sont respectivement **.** et **..** . Quand on créer un dossier on a donc automatiquement deux dossiers qui nous permettent de nous déplacer.
Passons maintenant à la prochaine fonction :

```
fe4_inode *get_new_file_inode(void);
```

Ici, nous allons créer non pas un i-node pour un dossier mais pour un fichier. Le fonctionnement est un peu similaire par rapport à la fonction précédente, mais clairement simplifié car ici il n'est question que d'un simple fichier.

```
ssize_t read_inode(const fe4_inode *inode, void *buf, size_t count, off_t offset);
ssize_t write_inode(fe4_inode *inode, const void *buf, size_t count, off_t offset);
```

C'est ici, entre autres, qu'on comprend tout l'intérêt de Fuse. Pour les fonctions de **read** et **write**, Fuse nous permet d'éviter l'utilisation de file descriptor et renvoie la position de fin après l'opération.

```
fe4_inode *get_inode_from_path(const char *path);
```

Obsolete ? ? ?

```
int get_nb_children(const fe4_inode *inode); // Returns the number of children. Some may have "/"

fe4_dirent *get_dirent_at(fe4_inode *parent, int index);

fe4_inode *get_inode_at(ino_t index);
```

Ces fonctions sont plutôt triviales à comprendre.

```
void add_dirent_to_inode(fe4_inode *inode, const fe4_dirent *dirent);
```

Cette fonction permet de créer un dossier dans un autre. Déjà pour commencer on vérifie que l'i-node donné est bien un dossier grâce au mode **S_ISDIR** .Ensuite on récupère l'id du prochain noeud disponible dans le dirent et on appel la fonction *write_inode* .

```
void delete_inode_at(ino_t index);
```

Ici, contrairement à ce que son nom l'indique, on ne supprime pas vraiment l'i-node, mais on exprime qu'il est libre en passant sa taille à -1.

```
void delete_dirent_at(fe4_inode *inode, int index);
```

Premièrement on vérifie bien que l'i-node est bien un dirent. **PAS SUR DE BIEN COMPRENDRE CA**

```
fe4_dirent *dirents = (fe4_dirent *)inode->contents;
strcpy(dirents[index].filename, "/");
```

3.2.2 Implémentations des fonctions principales

Les fonctions principales appelées par notre programme se trouvent dans le *main.c*.
Elles sont prédéfinies dans la structure *fe4_oper* avec chaque champ auquel on attribue un pointeur de fonction :

```
static const struct fuse_operations fe4_oper = {
    .init           = fe4_init,
    .getattr        = fe4_getattr,
    .readdir        = fe4_readdir,
    .open           = fe4_open,
    .read           = fe4_read,
    .mknod          = fe4_mknod,
    .mkdir          = fe4_mkdir,
    .truncate       = fe4_truncate,
    .write          = fe4_write,
    .unlink         = fe4_unlink,
    .rmdir          = fe4_rmdir,
    .rename         = fe4_rename,
};
```

Commençons par **mkdir** :

```
static int fe4_mkdir(const char *path, mode_t mode);
```

Tout d'abord, on vérifie que le dossier n'existe pas déjà.
Ensuite on vérifie que le chemin existe déjà : donc à la fois que l'inode existe mais aussi qu'il est bien un dirent.
Ensuite on va appeler la fonction *get_new_dir_inode* afin de créer le dirent, ensuite on le place au bonne endroit suivant le path donné grace à *add_dirent_to_inode* .

Maintenant l'implémentation de **mknod** :

```
static int fe4_mknod(const char *path, mode_t mode, dev_t dev);
```

On vérifie déjà deux choses : premièrement que c'est bien un fichier et deuxièmement qu'il n'existe pas déjà.
Ensuite c'est les même opération que pour **mkdir** mais on utilise la fonction *get_new_file_inode* .

Pour **getattr** :

```
static int fe4_getattr(const char *path, struct stat *stbuf, struct fuse_file_info *fi) ;
```

Ici c'est plutôt simple, on va renvoyer le *stat* du noeud s'il existe sinon une erreur.

Afin de pouvoir utiliser la commande *ls* , on a ici besoin d'implémenter **readdir** :

```
static int fe4_readdir(const char *path, void *buf, fuse_fill_dir_t filler,
                       off_t offset, struct fuse_file_info *fi,
                       enum fuse_readdir_flags flags);
```

Déjà, on commence à s'assurer que le noeud existe et qu'il s'agisse bien d'un dossier. Ensuite on parcourt tous les enfants du noeud et on passe le filename dans le buffer grâce à la fonction Fuse *filler* .

Les prochaines fonctions **open** , **read** et **write** sont codées selon le même principe :

```
static int fe4_open(const char *path, struct fuse_file_info *fi);
static int fe4_read(const char *path, char *buf, size_t size,
                    off_t offset, struct fuse_file_info *fi);
static int fe4_write(const char *path, const char *buf, size_t size,
                    off_t offset, struct fuse_file_info *fi);
```

En effet, pour les trois, on vérifie que l'i-node existe et qu'il est bien un fichier. Pour **open**, l'opération consiste seulement à accéder à l'i-node et agir selon les flags (**VERIFIER SI C'EST JUSTE**. Concernant **read** et **write** , les fonctions appellent respectivement *read_inode* et *write_inode* .

Pour la fonction **truncate** ,le fonctionnement est globalement similaire :

```
static int fe4_truncate(const char *path, off_t size, struct fuse_file_info *fi) ;
```

ICI IL Y A UN TODO PUT ZERO, A FAIRE ?

Regardons maintenant **unlink** et **rmdir** :

```
static int fe4_unlink(const char *path);
static int fe4_rmdir(const char *path);
```

4 Tests

4.1 Makefile

4.2 Exécution du code

5 Conclusion

6 Annexe

Les liens sont cliquables

https://doc.ubuntu-fr.org/utilisateurs/felixp/systeme_de_fichiers

https://learn.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2003/cc781134(v=ws.10)?redirec

https://learn.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2003/cc776720(v=ws.10)

https://www.kernel.org/doc/html/latest/filesystems/ntfs3.html

https://fr.wikipedia.org/wiki/NTFS

https://www.pjrc.com/tech/8051/ide/fat32.html

https://fr.wikipedia.org/wiki/File_Allocation_Table