



Rapport de conception

Projet Système 2025/2026 – Système de Gestion de Fichiers (SGF)

Membres de l'équipe :

- CHOISY Alexis
- DEGAT Teddy
- DA SILVA FERREIRA Lucas
- FOURNIE Baptiste
- FATIHI Youssef

Encadrante : ABOUDA Dhekra

Date de remise : 2 mai 2026

Sommaire

- [I. Introduction](#)
- [II. Architecture et Modélisation](#)
- [III. Algorithmes Principaux](#)
- [IV. Conclusion](#)

I. Introduction

Ce projet consiste à réaliser un SGF (Système de Gestion de Fichier) dans le cadre d'un projet.

Le cahier des charges impose la réalisation de ce SGF en se basant sur un système d'inode et de blocs stockés (ext3fs ou ext4fs) sur un volume afin de persister les données.

Ce rapport a pour but de montrer l'avancement du projet, notamment concernant la conception et l'explication du fonctionnement des primitives du SGF.

II. Architecture et Modélisation

Pour rappel, le projet est structuré de cette manière :

```
typedef struct env {
    char* key;
    char* value;
} env;

/**
 * @struct inode
 * @brief Un inode est un fichier, il possède des permissions, un type
 * (répertoire par exemple) et pointe sur des blocs de données
```

```

*/
typedef struct inode {
    unsigned short perms; // rwxrwxrwx
    char filetype;
    int blocs[MAX_BLOCS];
} inode;

/**
 * @struct bloc
 * @brief Un bloc possède un tableau de données brut concernant des inodes
 */
typedef struct bloc {
    char datas[MAX_BYTES_PER_BLOC];
} bloc;

/**
 * @struct disk
 * @brief Un disque est une liste d'inodes qui pointent sur des blocs de
donnée
 */
typedef struct disk {
    char owned_blocs[MAX_BLOCS]; // 1 si possédé par un inode, 0 si libre
    inode inodes[MAX_INODE];
    bloc blocs[MAX_BLOCS];
} disk;

// pour 10 inode qui a 30 blocs de chacun 1024 octets, on a 30720 octets,
soit
// 30,7 Ko sur le disque

```

Entre temps nous avons jugé important de rajouter

```
char owned_blocs[MAX_BLOCS]
```

pour certaines commandes afin de déterminer si un bloc est libre ou non (par exemple pour la commande `df`).

Nous avons aussi rajouté une structure `env` sous forme d'un dictionnaire mis à disposition pour le shell (par exemple la variable 'PWD' qui sert pour parcourir l'arborescence). `env` utilise le design pattern singleton, c'est à dire qu'une seule instance de cet objet existe et est fréquentée ou modifiée quand cela est nécessaire.

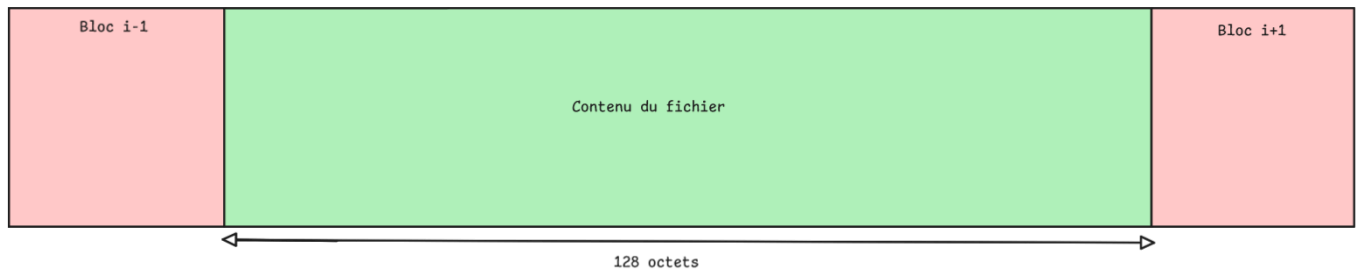
III. Algorithmes Principaux

Ici sera expliqué le fonctionnement des primitives du SGF ainsi que certaines commandes du shell associé.

3.1. La primitive `create_file()`

Avant d'expliquer le fonctionnement de la primitive, il est nécessaire d'expliquer comment fonctionne un fichier dans un système ext3fs

Structure du bloc mémoire du fichier '/dir1/file2'
(Hypothèse sur un disque possédant des blocs de 128 octets)



Il s'agit simplement d'un inode qui pointe sur un ou plusieurs bloc de donnée (ici 128 octets). Ce bloc décrit le contenu du fichier. L'inode contient

- Si l'inode concerne un fichier, un répertoire ou un lien symbolique
- les droits de l'inode concerné (droit de lecture/écriture/exécution)

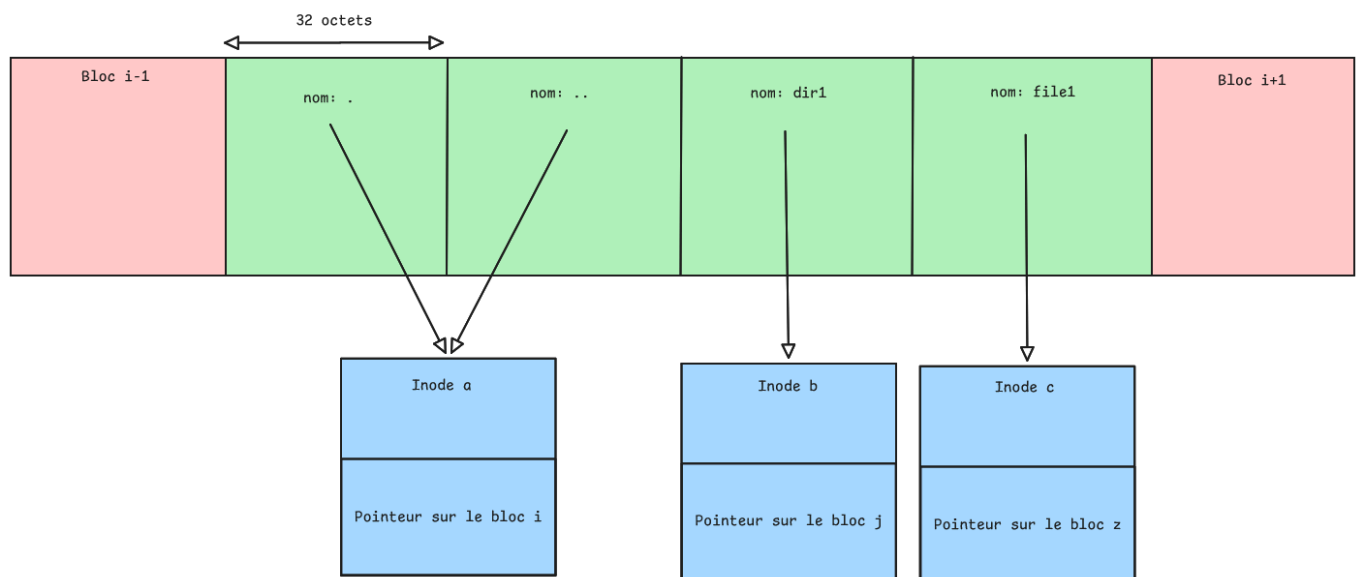
La primitive fonctionne de la manière suivante :

- On trouve un bloc de donnée libre
- Une fois ce bloc trouvé, on trouve un espace libre de 32 octets (28 pour le nom du fichier et 4 pour la référence à l'index de l'inode concerné)

3.2. La primitive `create_directory()`

Premièrement il faut savoir comment se structure un dossier dans un système ext3fs

Structure du bloc mémoire du dossier '/'
(Hypothèse sur un disque possédant des blocs de 128 octets)



La primitive fonctionne de la manière suivante :

- On trouve un bloc de donnée libre
- Une fois ce bloc trouvé, on trouve un espace libre de 32 octets (28 pour le nom du fichier et 4 pour la référence à l'index de l'inode concerné)
- On créer un dossier '.' qui pointe sur l'inode du dossier en création
- On créer un dossier '..' qui pointe sur le parent de l'inode en création

à noter qu'un dossier n'est qu'un fichier qui structure l'information d'une manière différente (en faisant stockant le nom et la référence vers les inodes de ses fils)

3.3. La primitive `write_in_file()`

Il s'agit simplement d'écrire des données dans un bloc de données soit en écrasant les données par des nouvelles (mode 'w') soit en rajoutant les données à la suite de celles déjà existantes (mode 'a').

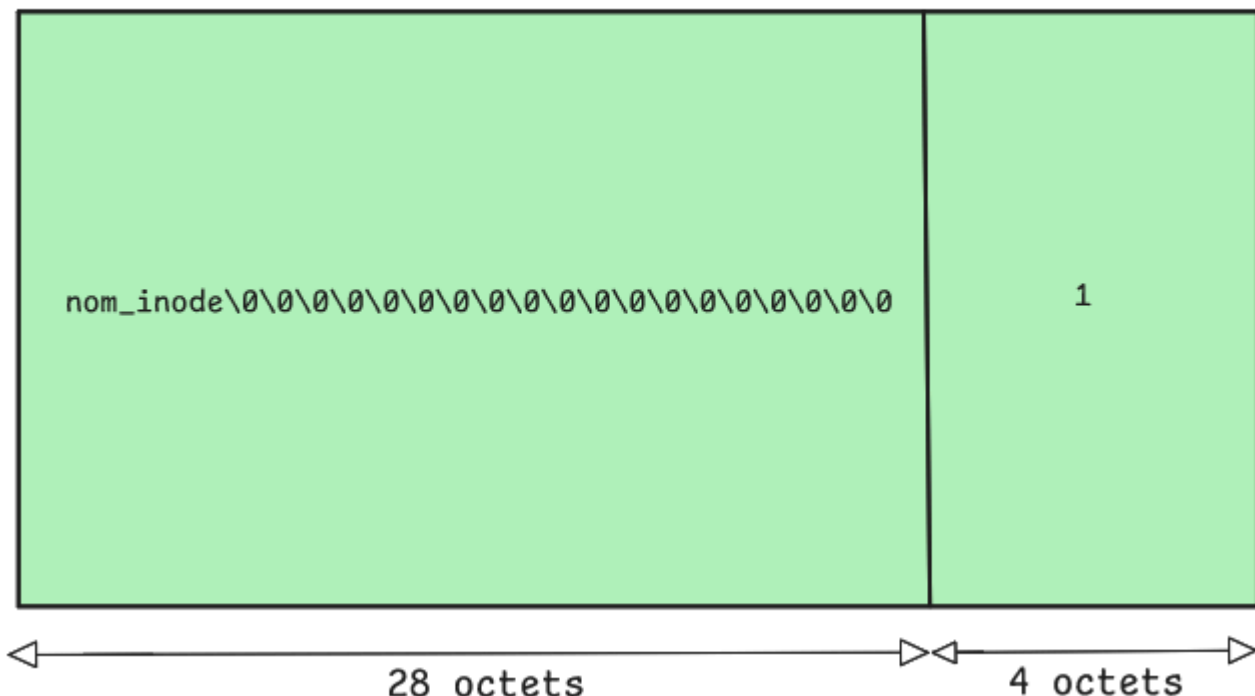
3.4. La commande cd

Contrairement aux autres commandes du projet, `cd` est une commande assez spéciale car elle est toujours implémentée par le shell lui même et non déjà existante dans le système. En bref, elle est propre à chaque shell (bash, zsh, fish...). Dans les faits ça ne change pas grand chose mais l'implémentation n'est pas dans le fichier `disk.c` mais `exec.c` qui s'occupe de tout l'aspect exécution de commande (et non implémentation à la base).

Le plus long est le parsing du chemin, le fait de séparer les noms des dossiers des '/'. Par exemple pour le chemin `/dir1/dir2/` on va devoir parcourir `dir1` puis `dir2`. Pour cela, il existe une fonction `find_dir_inode_by_name()` qui comme son nom l'indique sert à retrouver un inode grâce à son nom et comme on le sait déjà, le nom d'un inode est stocké dans le/les blocs du parent.

En bref, on va commencer du tout début du/des blocs du parent jusqu'à trouver (ou non) le nom du dossier donné en paramètre. Et comme les noms sont stockés de cette manière :

Comment est stocké le nom d'un inode dans son parent



Une fois l'inode associé au nom du dernier dossier dans le chemin trouvé, on change la variable d'environnement 'PWD'.

3.5. La commande `ls`

`ls` est très similaire à `cd` sur le fonctionnement à la seule différence que `ls` affiche le contenu du dossier avec l'inode associé.

IV. Conclusion

En résumé, cette phase intermédiaire de conception nous a permis de poser des fondations solides et cohérentes pour notre Système de Gestion de Fichiers. L'architecture globale, basée sur le modèle des inodes et des blocs de données (façon ext3/ext4), est désormais clairement modélisée et structurée en mémoire.

Le développement des primitives fondamentales — comme la création et l'écriture de fichiers ou l'arborescence des dossiers — couplé aux premières commandes vitales du shell (`cd`, `ls`), prouve la viabilité de notre approche. L'ajout de structures d'optimisation comme `owned_blocs` ou l'environnement global (`env`) montre également notre capacité à adapter le cahier des charges aux défis techniques rencontrés en cours de route.

Bien que le projet soit encore en phase de développement, cette étape de mi-parcours valide nos choix algorithmiques et architecturaux. Les bases sont désormais saines et opérationnelles pour entamer la suite de l'implémentation et aboutir à un système complet et persistant d'ici le rendu final.