

# Rapport final

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

## Auteurs :

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

## Sommaire

- I. Introduction
- II. Explication du travail réalisé
- III. Points à améliorer
- 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. Explication du travail réalisé

### 2.1 Fonctionnement général du shell

#### 2.1.1 Son rôle

Le projet se décompose en deux modules :

- Le SGF qui permet d'interagir avec le disque
- Le Shell qui sert d'interface avec le SGF, il s'agit d'appeler des primitives système grâce à des commandes

Le travail du Shell se décompose en plusieurs étapes :

- Lire les entrées utilisateur
- Formater l'entrée utilisateur
- Exécuter la commande associée à l'entrée utilisateur (en faisant appel à des primitives du SGF)

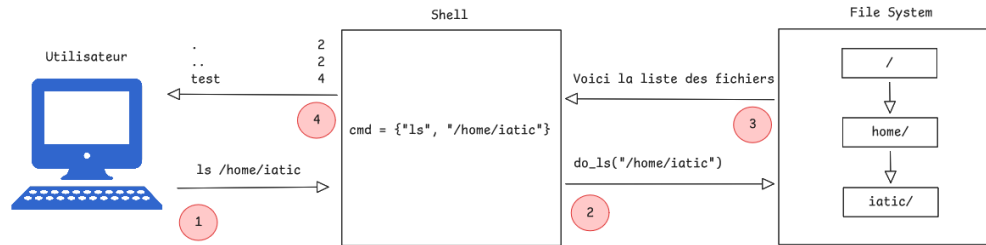


Figure 1: Schéma d'une utilisation classique d'un shell

### 2.1.2 Le mécanisme de pipe

Voici un exemple de commande utilisant un pipe

```
commande1 | commande2
```

Un pipe a pour utilité de passer les informations en sortie de la `commande1` en entrée à la `commande2`.

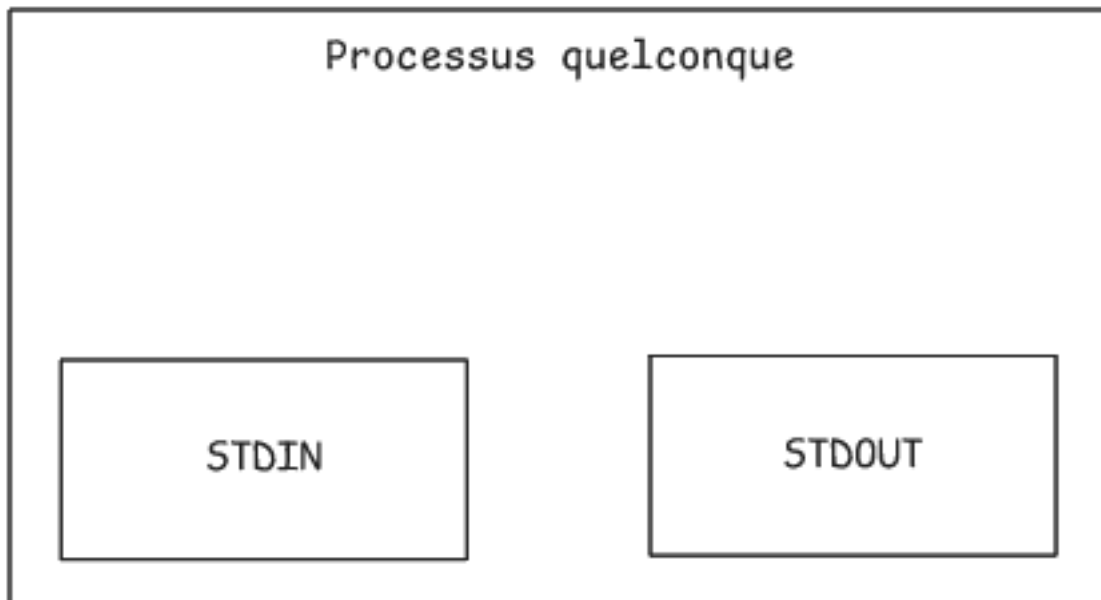


Figure 2: Schéma simplifié des canaux d'un processus

Dans notre shell nous faisons en sorte grâce au mécanisme de pipe anonyme en C d'écraser la sortie standard (STDOUT) du processus fils de `commande1` par la référence d'écriture du pipe et d'écraser l'entrée standard (STDIN) du processus fils de `commande2` par la référence de lecture du pipe.

Comme ça la première commande pense afficher son résultat sur sa sortie standard alors qu'en réalité elle écrit sur le pipe anonyme créé au préalable.

Pareillement pour la seconde commande qui elle pense lire sur son entrée standard mais lis en fait ce qu'à écrit le premier processus. En quelques sortes, la première commande passe le relais à la seconde.

### 2.1.3 Le mécanisme de redirection

Le mécanisme de redirection consiste à rediriger la sortie d'une commande ou d'un ensemble de commande vers un fichier.

Pour cela on utilise les opérateurs `>` et `>>`.

- L'opérateur `>` redirige la sortie standard (STDOUT) de la dernière commande vers un fichier en écrasant les données déjà écrites dedans
- L'opérateur `>>` redirige la sortie standard (STDOUT) de la dernière commande vers un fichier sans écraser les données déjà écrites dedans

## 2.2 Fonctionnement général du SGF

Le SGF propose des primitives système afin de communiquer avec le disque comme :

- `create_file()`
- `create_directory()`
- `write_in_file()`
- `read_in_file()`
- `remove_inode()`

Dans un SGF type `ext4` le disque structure l'information sous forme d'inode et de blocs

- Un inode définit un fichier, un dossier ou bien un lien symbolique, il possède des permissions et pointe sur un bloc de donnée.
- Un bloc définit de la donnée brute, c'est un espace mémoire possédé par un inode et celui-ci est libre d'écrire ce qu'il veut dans cet espace

```
typedef struct inode {
    unsigned short perms; // rw-rw-rw-
    char filetype;
    int blocs[MAX_BLOCS];
} inode;

typedef struct bloc {
    char datas[MAX_BYTES_PER_BLOC];
} bloc;

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;
```

Dans le cas de notre projet, le disque est représenté sous forme de fichier.

## 2.3 Création du disque

Le processus de création du disque est réalisé grâce à un dump des données de la structure `disk` en mémoire sur un fichier intitulé `disk`.

On y initialise chacun des inodes à la constante `TYPE_NULL`, les blocs sont aussi initialisés à `-1`. Ensuite on crée la racine du système (`/`) et on lui alloue un bloc. Ensuite on initialise la racine elle-même en créant les deux dossiers `.` et `..` qui pointent sur le même répertoire (la racine).

## 2.4 Création d'un fichier

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

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

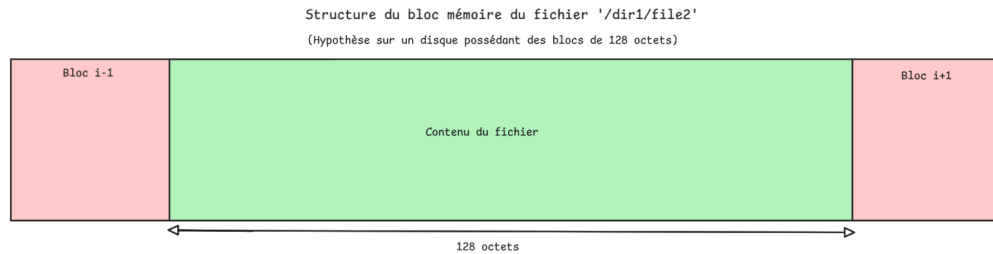


Figure 3: illustration du fonctionnement d'un fichier

- 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 donné 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)

## 2.5 Création d'un dossier

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

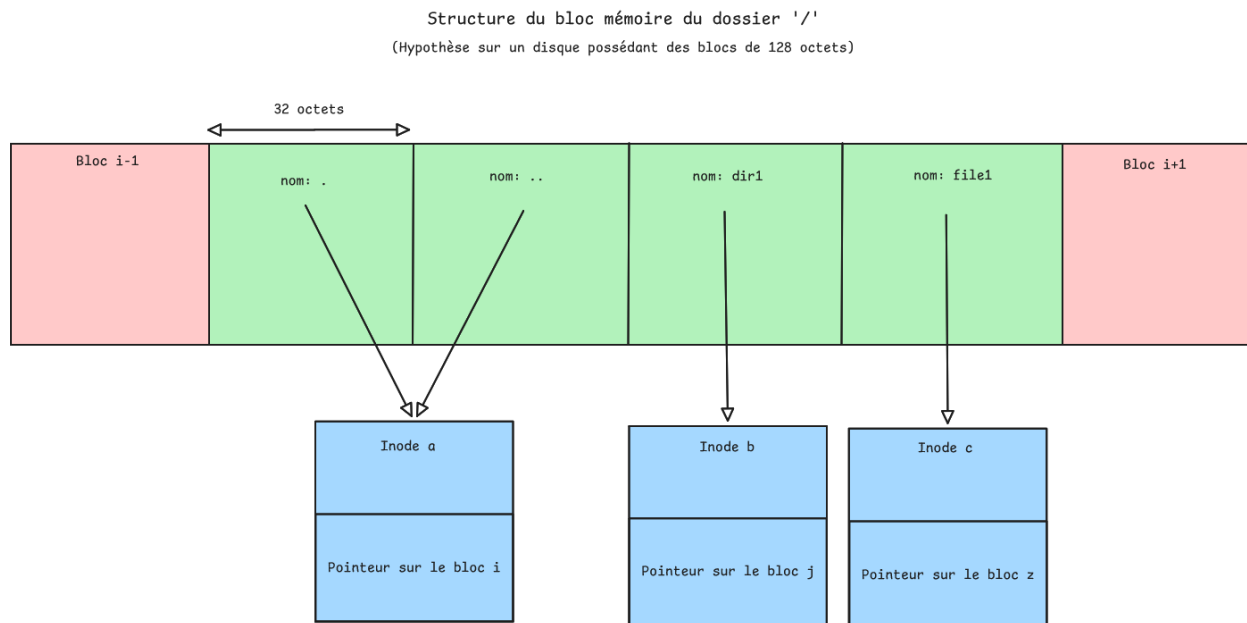


Figure 4: illustration du fonctionnement d'un dossier

La primitive fonctionne de la manière suivante :

- On trouve un bloc de donné 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)

## 2.6 Le nommage d'un inode

Un inode n'a pas de nom en soit, en fait son nom est stocké dans le parent. Et un répertoire stocke les noms des répertoires ainsi qu'une référence à chacun des inodes.

Dans notre SGF, nous avons choisi de stocker les noms/référence comme la figure ci-dessous

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

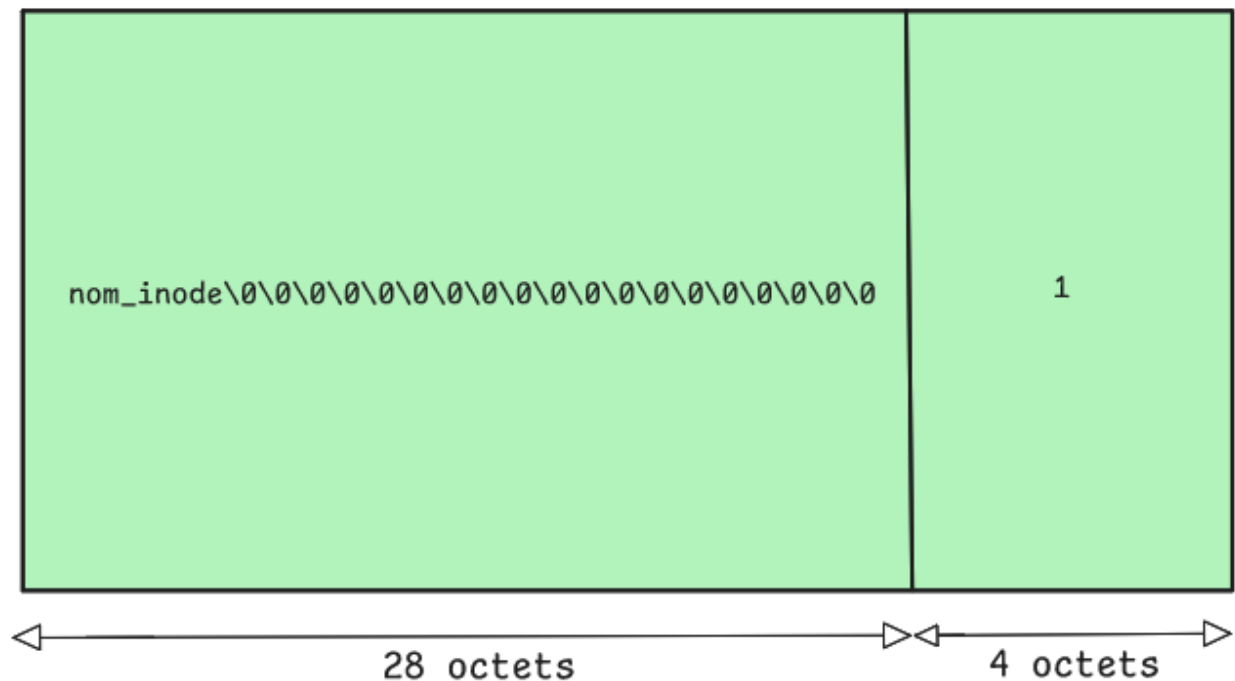


Figure 5: illustration de la méthode de stockage du nom d'un fichier

Sur un bloc de 1024 octets ça fait 32 inodes max référencés dans le dossier.

## 2.7 Les commandes et leurs fonctionnement

### 2.7.1 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 :

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

### 2.7.2 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.

### 2.7.3 Les commandes **mkdir** et **touch**

Ces deux commandes sont sensiblement les mêmes, c'est pour cela qu'elles sont dans la même section.

- **touch** fait appel à la primitive `create_file()` avec quelques vérification au préalable (par exemple l'emplacement dans l'arborescence pour créer le fichier)

`create_file()` fonctionne de la manière suivante :

1. On essaie de trouver si il reste un inode disponible, si non on renvoie un message d'erreur
2. On essaie de trouver si il rest un bloc disponible, si non on renvoie un message d'erreur
3. On trouve de l'espace dans le répertoire parent pour pouvoir référencer l'inode dans l'arborescence
4. On écrit le nom du fichier dans le répertoire parent, ses permissions, son bloc alloué et son numéro d'inode

En ce qui concerne **mkdir** cette commande fait appel `create_directory()` qui s'occupe de faire concrètement le même travail que `create_file()` à la seule différence que l'inode est de type `TYPE_DIRECTORY` et non de `TYPE_FILE`. (à noter que dans un système UNIX tout est fichier).

### 2.7.4 Les commandes **rmdir** et **rm**

Comme **touch** et **mkdir**, **rmdir** et **rm** sont très similaires.

Ces deux commandes sont encore plus similaires que les deux précédentes car leur seule différence est seulement la vérification si l'inode est un fichier ou un répertoire.

Ils font appel à la même primitive `remove_inode()` qui fonctionne de la manière suivante :

1. On vérifie si le dossier à supprimer est la racine, si oui on renvoie une erreur
2. On efface la référence de l'inode à supprimer dans le parent
3. On efface tout le contenu de l'inode en réinitialisant le contenu du bloc avec des 0
4. On précise que le bloc supprimé est désormais libre
5. On définit l'inode comme `TYPE_NULL`

### 2.7.5 La commande **cat**

**cat** permet de voir le contenu des fichiers en lui précisant le nom d'un fichier grâce à la primitive `read_in_file()`

cette primitive est très simple car elle se contente de lire la donnée brute du bloc et renvoyer ce contenu sous forme de `char *`.

Il est aussi possible de faire appel à **cat** sans spécifier de fichier, si c'est le cas alors le programme va juste écouter son entrée standard jusqu'à que l'utilisateur tue le processus grâce à un `SIGINT` par exemple.

### 2.7.6 La commande **df**

Cette commande affiche le nombre de bloc restant sur le disque, le nombre d'inode ainsi que l'espace de stockage restant sur le système.

Exemple :

```
iatic@tartempion /> df
Free blocs : 28
Inode left : 28
Space left : 14.34Kb
```

### 2.7.7 La commande `echo`

Comme `cd`, `echo` est une commande built-in au Shell, c'est à dire que contrairement aux autres commande, c'est une commande qui est normalement implémenté par le shell lui même et ce n'est pas un exécutable.

`echo` se contente de rediriger ce qu'il reçoit en argument vers sa sortie standard, il regarde au préalable si dans la chaine qu'on lui a passé, y est inclut une variable d'environnement (`$PWD` par exemple), si c'est le cas, il l'affiche.

## III. Points à améliorer

Bien que le SGF et son shell soient fonctionnels et respectent les primitives de base demandées, plusieurs axes d'amélioration permettront de rapprocher le système d'un comportement de type UNIX/ext4 réel :

- **Gestion de la mémoire et robustesse** : Actuellement, le code présente des risques de *buffer overflow*, notamment lors de la manipulation des chemins de fichiers ou des arguments dans le shell. L'utilisation systématique de fonctions sécurisées et une validation stricte de la taille des entrées utilisateur sont nécessaires pour sécuriser le système.
- **Mise en œuvre concrète des permissions** : Bien que la structure `inode` intègre un champ `perms`, les mécanismes de vérification lors de l'appel aux primitives ne sont pas encore activés. Il faudrait l'implémenter pour interdire, par exemple, l'écriture dans un fichier en lecture seule.
- **Système de gestion des utilisateurs** : Le SGF ne gère pour l'instant qu'un utilisateur unique (équivalent à un accès *root* permanent). L'ajout d'une table des utilisateurs, d'une commande `su/login` et le stockage de l'UID (User ID) dans l'inode permettraient une isolation complète des données.
- **Allocation dynamique et fragmentation** : Actuellement, la taille des fichiers est contrainte par un nombre fixe de blocs contigus (`MAX_BLOCS`). L'introduction de pointeurs indirects dans l'inode permettrait de supporter des fichiers de taille variable et plus importante sans gaspiller l'espace disque.

## IV. Conclusion

Ce projet nous a permis de concevoir et de réaliser l'architecture complète d'un Système de Gestion de Fichiers (SGF) simplifié ainsi que son interface en ligne de commande. En modélisant un disque virtuel par un simple fichier binaire, nous avons pu manipuler concrètement les concepts fondamentaux des systèmes d'exploitation : la structure et le cycle de vie des inodes, l'indexation des blocs de données, le fonctionnement interne d'un répertoire et le parsing de commandes d'un shell.

Malgré les limitations actuelles en termes de sécurité mémoire et de gestion des droits, le système développé offre une base solide, logique et fonctionnelle, illustrant parfaitement la maxime UNIX selon laquelle tout est fichier.